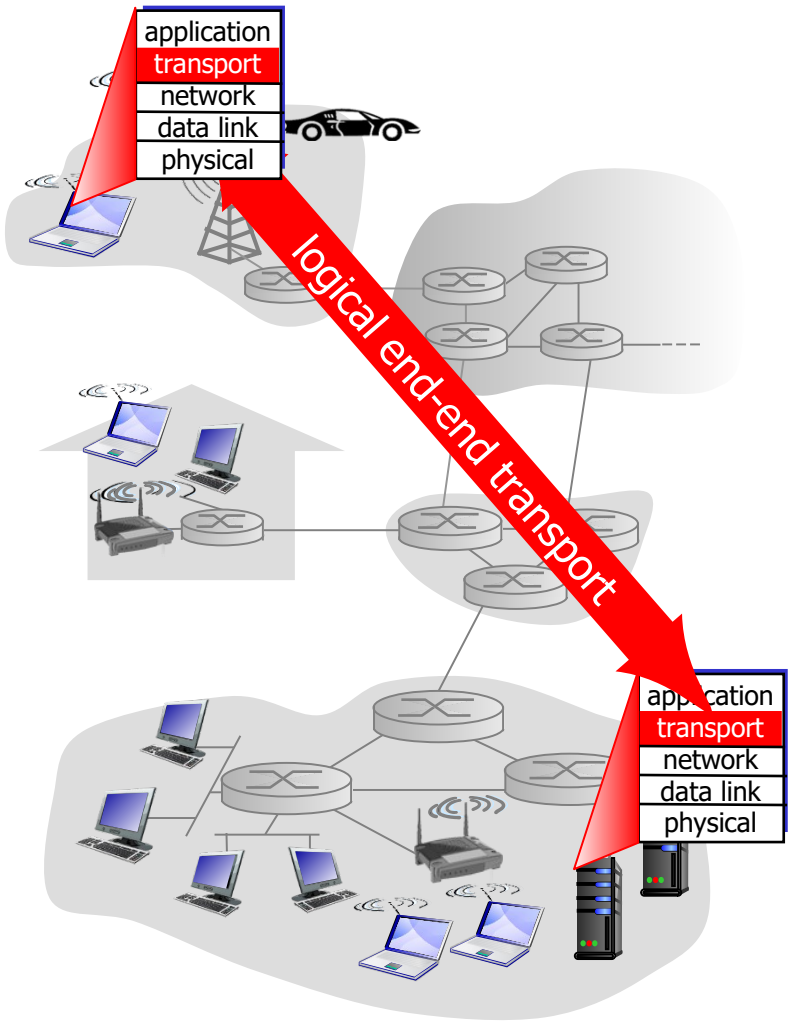
Capa de Transporte

Capa de Transporte

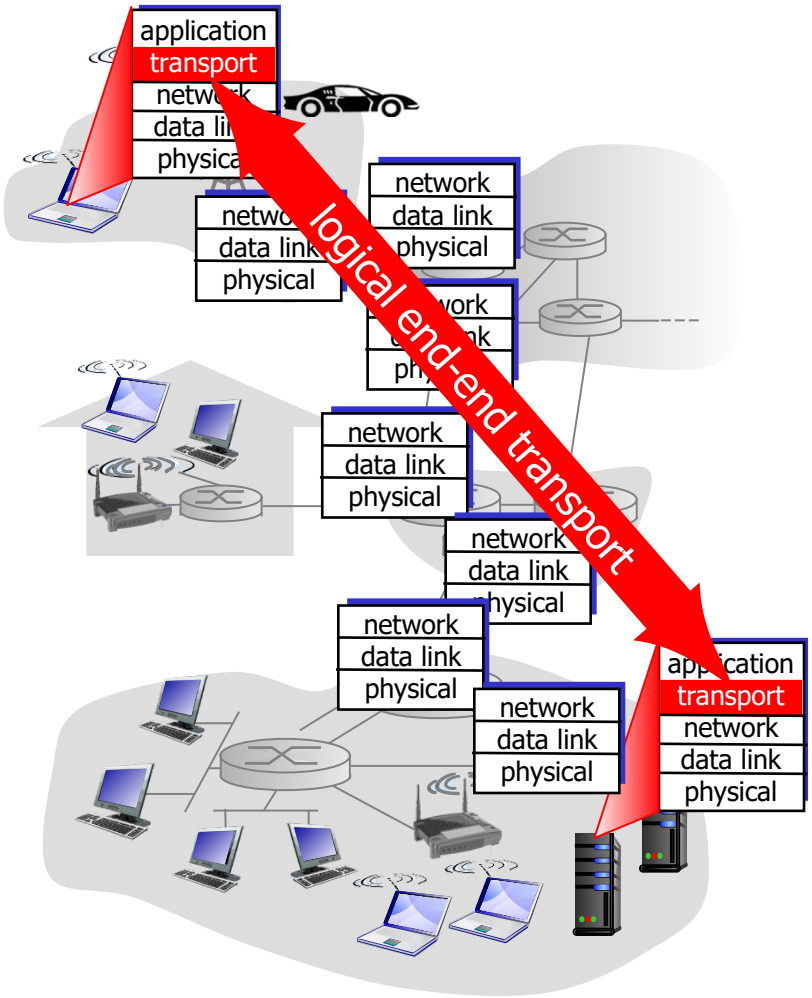
- Proporcionar una **comunicación lógica** a la entidades de capa de Aplicación que residen en máquinas diferentes
- La capa de Transporte dialoga extremo a extremo (entre equipos finales):
  - *Transmisor:* "parte" los mensajes de capa de aplicación en **segmentos/datagramas**, para luego entregarlos a la capa de red.
  - *Receptor:* re-ensamblar los **segmentos** en los mensajes "originales" y entregarlos a la capa de aplicación.
- Pueden existir mas de una alternativa para la elección de capa de Transporte.

  En Internet encontramos TCP y UDP.



Capa de Transporte - ¿Porqué otra capa más?

- Capa de red:
  - identificador de host (ejemplo IP)
  - “encaminar” los paquetes del origen a destino
  - comunicación lógica entre hosts
  - utilizar los servicios de capa de enlace
- Capa de transporte:
  - Comunicación lógica entre procesos o aplicaciones en hosts diferentes.
  - Utiliza los servicios de capa de red.
  - **Mejora o adapta** los servicios de capa de red, a los ofrecidos a capa de transporte, por ejemplo servicio con garantía de entrega.
- Dimensiones de calidad de servicio en redes:
  - **Garantía de entrega**
  - **Tasa de transferencia** efectiva (throughput, descartes, priorización).
  - **Requerimientos de tiempo** (delay, jitter)
  - **Seguridad** (autenticación, secreto, integridad, no repudio)



Capa de Transporte en Internet

- Servicios que ofrece la capa de Transporte:
  - Orientado a Conexión y Confiable
  - No Orientado a Conexión y No confiable
- **Confiable = Garantía de entrega** (no significa seguro)
- Orientado a Conexión y Confiable (**TCP**):
  - Control de Flujo
  - Control de Congestión
  - Entrega ordenada
  - Establecimiento y finalización de conexión
- **No** orientado a conexión y **No** confiable (**UDP**):
  - No hay garantía de entrega ni de orden. Es una extensión de capa de red.
- Dimensiones que **NO atiende** la capa de Transporte en Internet
  - Garantías temporales (delay y jitter)
  - Garantías de ancho de banda y priorización
  - “Seguridad” (formalmente TLS es capa de aplicación)

Funciones que debe realizar la capa de Transporte

- Para poder brindar servicios a la capa de aplicación la capa de transporte debe generalmente realizar las siguientes funciones:
  - Direccionamiento (identificador en capa de transporte)
  - Control de errores
  - Secuenciamiento
  - Control de flujo
  - Control de congestión
  - Multiplexado
  - Manejo de buffers y temporizadores

Capa de Transporte – Multiplexado y DeMultiplexado

- ¿Porqué necesito un identificador en capa de Transporte?
  - **Servidor:** lo usual es disponer de varias aplicaciones (web y ssh).
  - **Cliente:** accediendo a diferentes servicios remotos
- El identificador de capa de Transporte me permite poder utilizar varias aplicaciones tanto en el cliente como en el servidor.

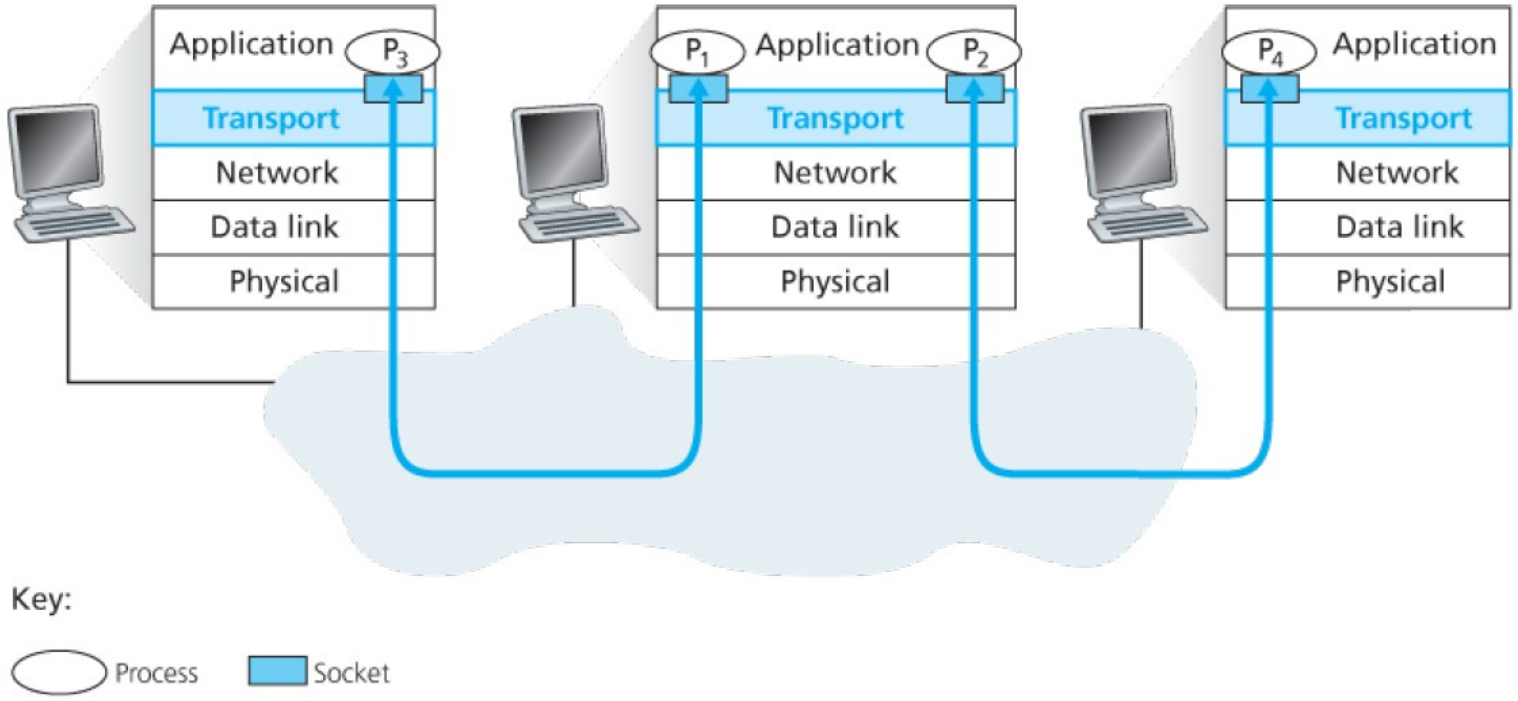


Figure 3.2 Transport-layer multiplexing and demultiplexing

Capa de Transporte – MUX/DMUX y Socket

- **TCP/UDP utilizan un identificador de 16 bits que se llama “puerto”.**

  Forma parte el encabezado que agregan las entidades de Transporte.

- **Socket:** Una abstracción lógica de una conexión entre aplicaciones
  - Identificación host origen (IP de origen)
  - Identificación host destino (IP de destino)
  - Identificación de Aplicación cliente (puerto de origen)
  - Identificación de Aplicación servidor (puerto destino)
  - Tipo de socket (UDP o TCP)

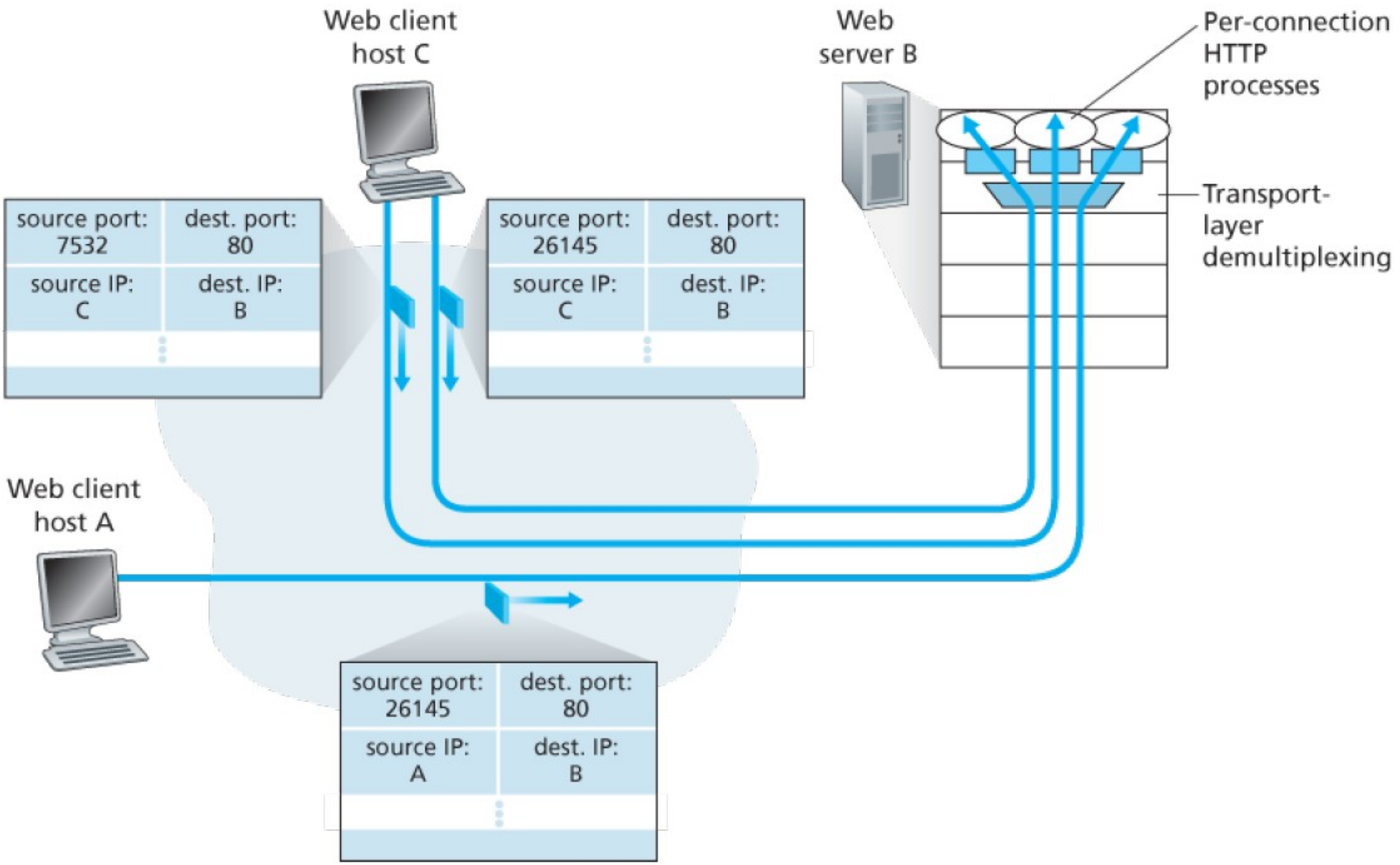
- **Demultiplexación:** la capa de transporte revisa el puerto destino en el segmento/datagrama, de acuerdo al puerto de destino “entrega” los datos a la aplicación correspondiente.

  Ejemplo solicitudes **http** y **https**

- **Multiplexación:** la capa de transporte, en cada socket, recibe datos de las diferentes aplicaciones,en cada caso agrega los encabezados de capa de transporte con ello los puertos de origen y destino.

- **Curiosos:** Se necesitan estrategias para programar aplicaciones, las usuales son copia de proceso (**fork**) dedicado al socket, o hilo de proceso (**threads**) dedicado al socket.

Capa de Transporte – MUX/DMUX y Socket



Capa de Transporte - Direccionamiento

- En general dos opciones: estática o dinámica.
- **Puerto de origen:** Por lo general la dirección IP de origen y el puerto de origen los elije el Sistema Operativo
- **Estática:** "puertos bien conocidos" una aplicación servidor escucha en un puerto fijo, asignado de forma estática.
  - HTTP puerto 80
  - HTTPS puerto 443
  - SSH puerto 22
  - SMTP puerto 25
  - Etc

  Deberían registrarse, o de no hacerlo evitar colisionar.
- **Dinámicos:** consulta a un directorio
  - Portmaper de Unix: Puerto fijo de inicio, se negocia un nuevo puerto.
  - Nuevo Registro DNS tipo **SRV**

    _xmpp-client._tcp.example.net. 86400 IN SRV 5 0 5222 server.example.net.

    _xmpp-server._tcp.example.net. 86400 IN SRV 5 0 5269 server.example.net.

Capa de Transporte – Errores en los datos

- Un error en los datos es la alternación de los bits (“1” por “0” o “0” por “1”) en los **datagramas/segmentos** intercambiados.
- Para poder identificar si hubieron errores necesito agregar información adicional. **Ej:** C.I. 1.234.567 → 1.234.567**-2** (función de todos los dígitos)
- Para el manejo de errores en la transmisión, se utilizan dos estrategias:
- **Detección de errores**
  - Mediante algún algoritmo se detecta si los datos recibidos son los esperados
  - Si no lo son, se descartan. En caso necesario se deberá pedir retransmisión
- **Corrección de errores**
  - Al transmitir los datos, se les agrega suficiente redundancia para poder corregir errores
  - Puede evitar retransmisiones (a cambio de mayor overhead)
- En general en capa de Transporte solo se realiza detección de errores
- En **capa de enlace** se suelen utilizar algoritmos de detección más complejos. En medios físicos inalámbricos (red celular) o enlaces de alta velocidad (100 Gbps), es posible utilizar corrección de errores **(FEC)**.

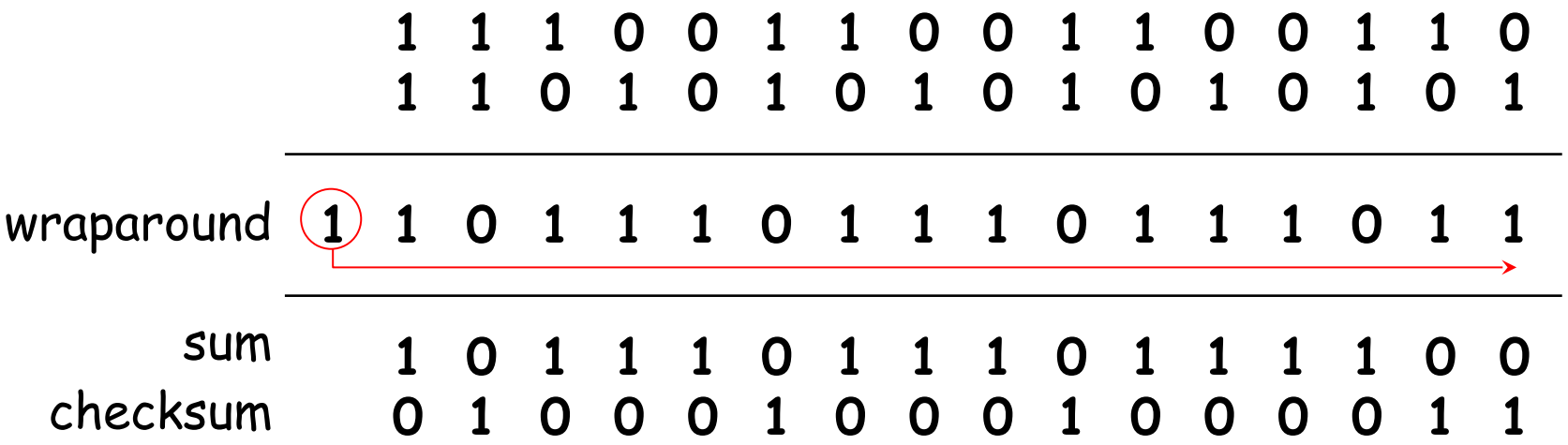
Capa de Transporte – Detección de Errores

- En general se recurre a las siguientes técnicas en la capa de transporte:
  - Suma de comprobación o **checksum**
  - Hay otras técnicas pero no suelen ser usadas en esta capa (se verán algunas durante el estudio de la capa de enlace).
- Suma de Comprobación:
  - Se divide el mensaje en palabras de largo fijo (típicamente 2 o 4 bytes), se rellena con “ceros”.
  - Se realiza una “suma” de todas las palabras.
  - Se envía el resultado junto con el mensaje.
  - En el receptor, se realiza la misma operación y se verifica el resultado.
  - En caso de diferencias, se declara que hubo error y se descarta.

Capa de Transporte – TCP/UDP Checksum

- **TCP/UDP:** Simple suma de comprobación de 16 bits
- El mensaje se divide en palabras de 16 bits, y se hace la suma en complemento a 1
- Se realiza el complemento a 1 (invertir 1s y 0s)
- Se hace sobre los datos, el encabezado capa 4, y un **"pseudo encabezado"** con información de capa 3

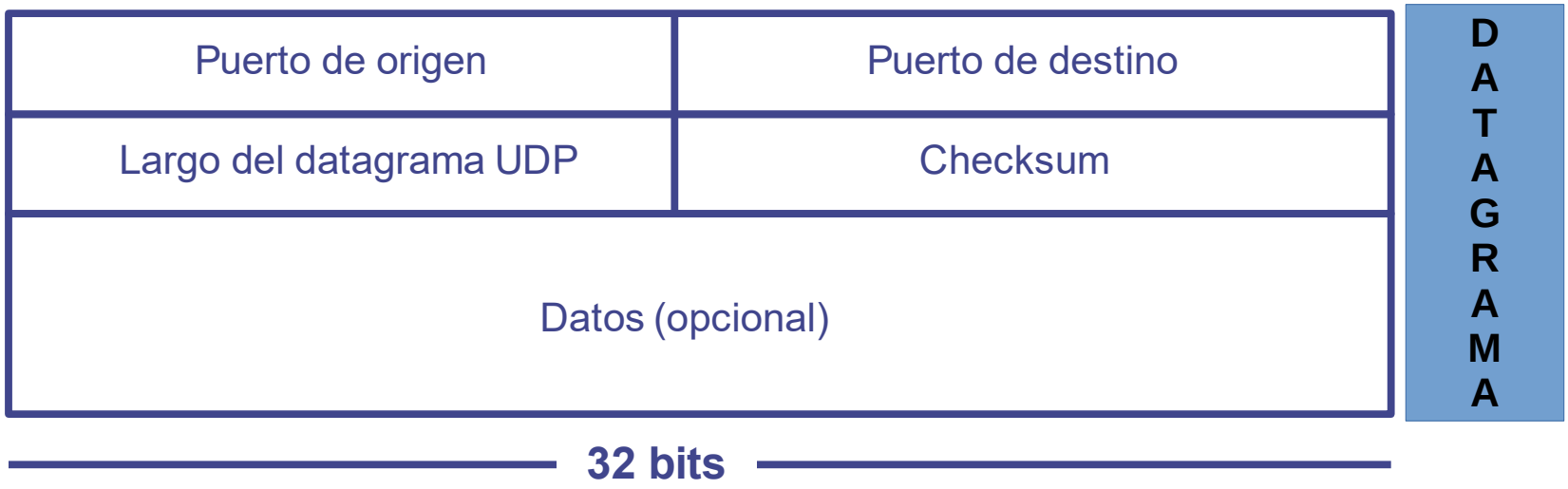
example: add two 16-bit integers



Capa de Transporte – UDP

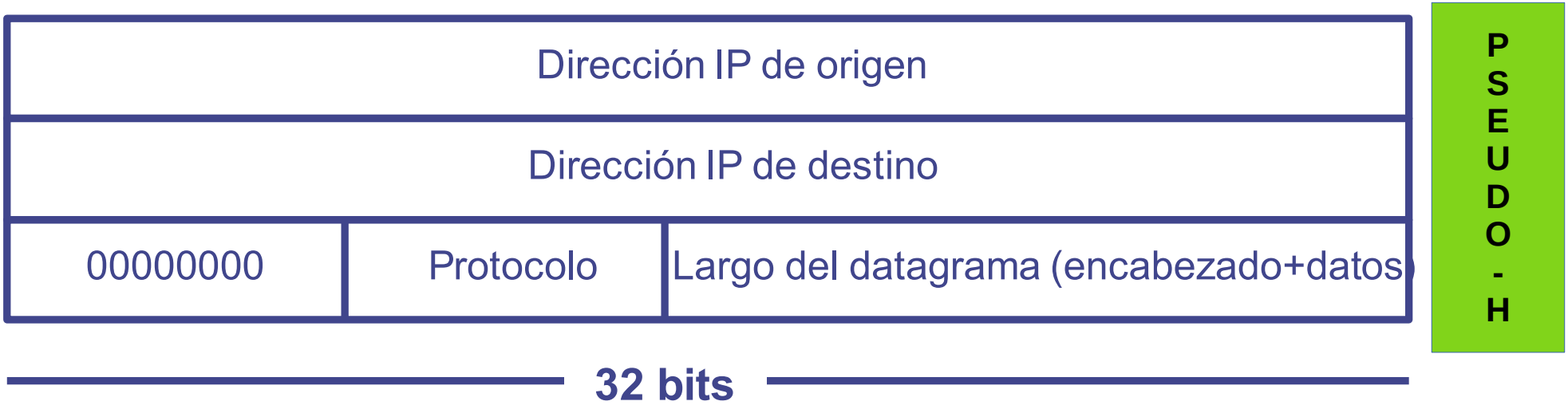
- **UDP:**
  - Servicio no orientado a Conexión
  - No confiable (no hay garantía de entrega)
  - Identificar la aplicación de origen y destino.
  - “Detectar” errores
  - **No tiene control de flujo** (capacidad del receptor)
  - **No tiene control de congestión** (capacidad de la “red”)

Encabezado UDP:

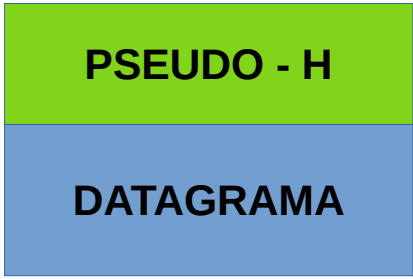


Capa de Transporte – UDP y Pseudo Encabezado

Pseudo Encabezado UDP (Checksum):



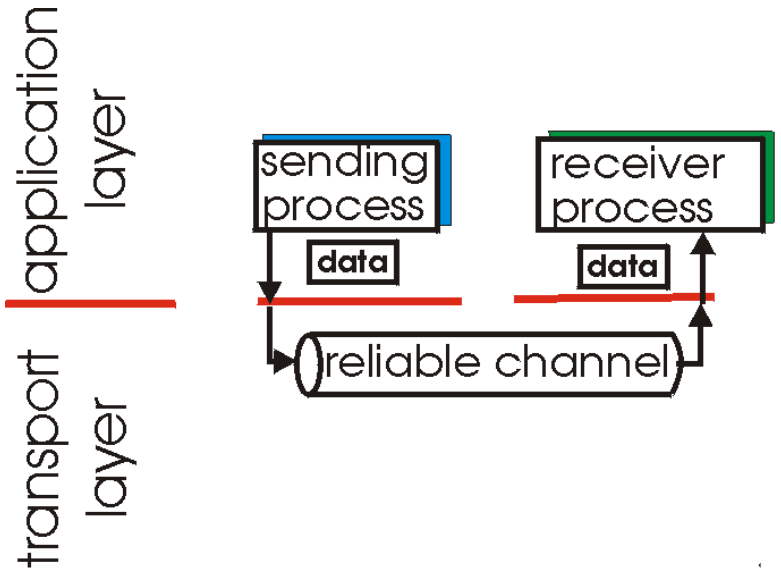
- Campo Cheksum del datagrama en “0..0” (todos ceros) al momento de calcular (**Pseudo-H** + **Datagrama**)
- El resultado luego se completa el campo Checksum



- **Violación a la independencia de capas**
  - Se debe “recalcular” en escenarios con NAT
  - Se modifica al cambiar de versión de IP!

Principios de Transferencia de Datos “Confiable”

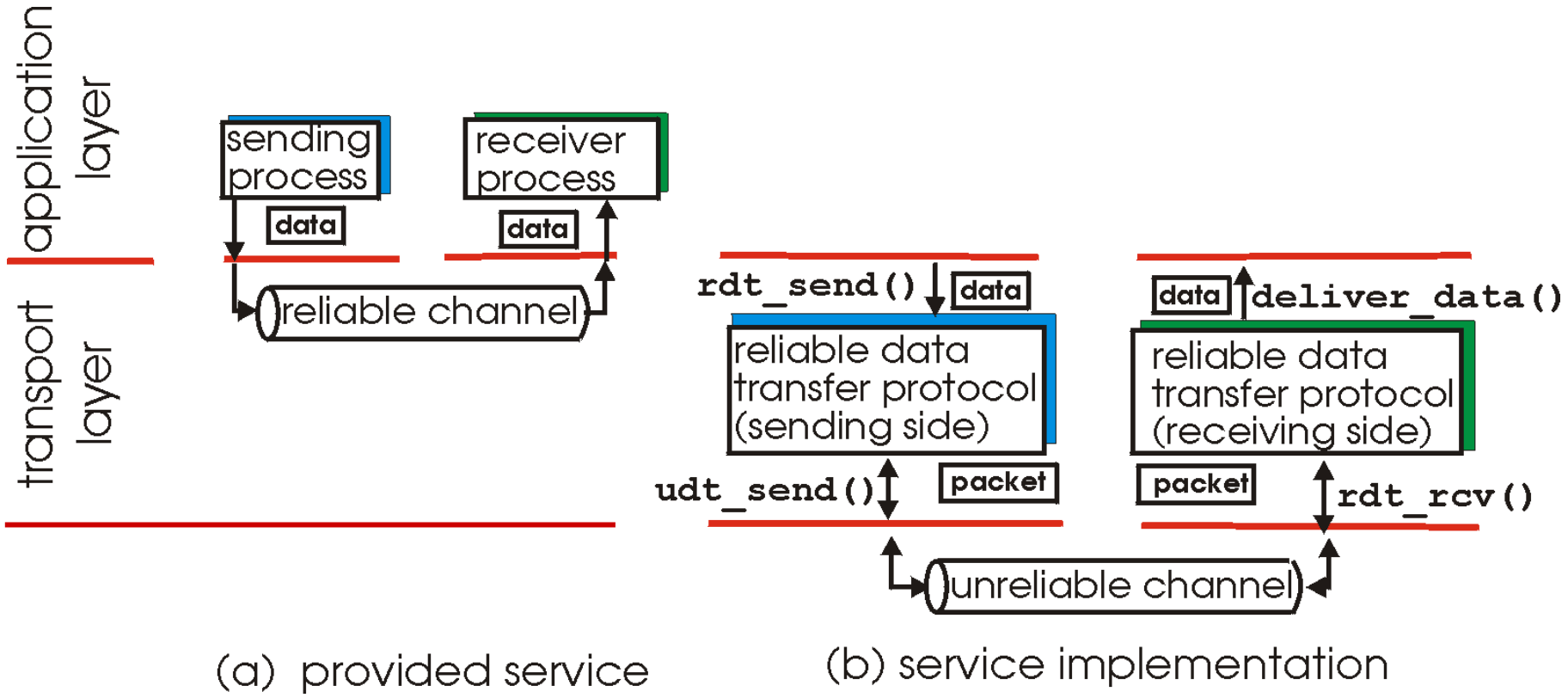
- **Un canal “confiable”:** aquel que permite la transferencia de datos, “0” y “1”, sin alterarlos.
- La abstracción del servicio **“canal confiable”** que ofrece la capa de transporte a la capa de aplicación debe de ser independiente de los servicios de las capas inferiores.
- **Puede no ser tan simple:** TCP utiliza los servicios de IP (best effort), ocurren pérdidas y los paquetes pueden llegar desordenados.



(a) provided service

Principios de Transferencia de Datos “Confiable”

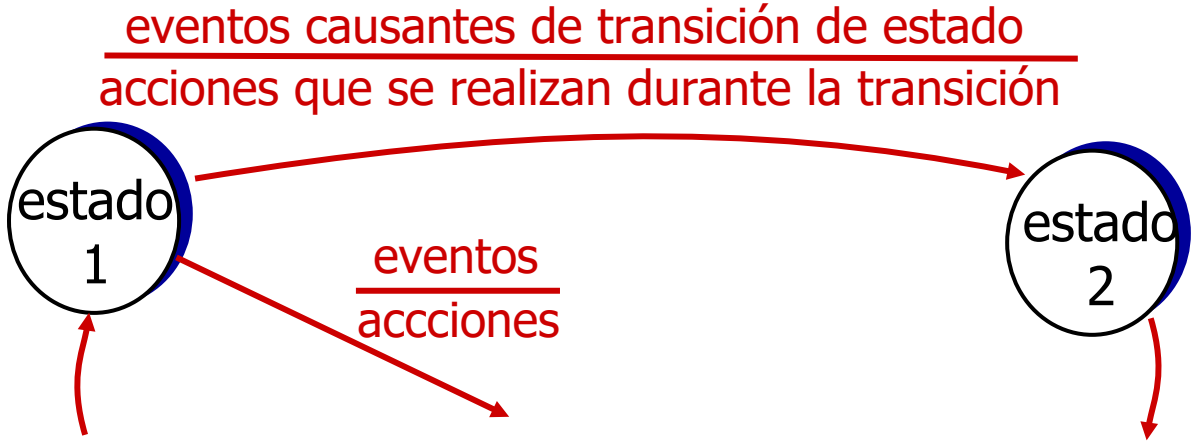
- **Un canal “confiable”:** aquel que permite la transferencia de datos, “0” y “1”, sin alterarlos.
- La abstracción del servicio **“canal confiable”** que ofrece la capa de transporte a la capa de aplicación debe de ser independiente de los servicios de las capas inferiores.
- **Puede no ser tan simple:** TCP utiliza los servicios de IP (best effort), ocurren pérdidas y los paquetes pueden llegar desordenados.



Construcción de protocolo - RDT Confiable

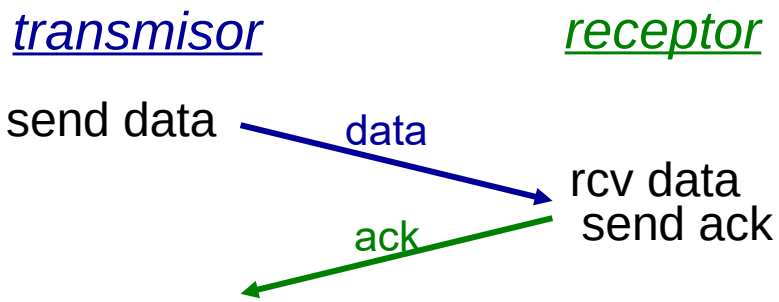
- **RDT** = Reable Data Transfer
- Construcción Evolutiva de un protocolo comenzando de un modelo simple y luego retirando las simplificaciones.
- Vamos a Trabajar: Como Diagrama de **estados** finitos o como Diagrama de **intercambio** (eventos y acciones).

Estado: la evolución a otro estado solo depende de un nuevo evento



Evento y acciones: llegada de datos o información de control (encabezados).

Envío de datos o información de control



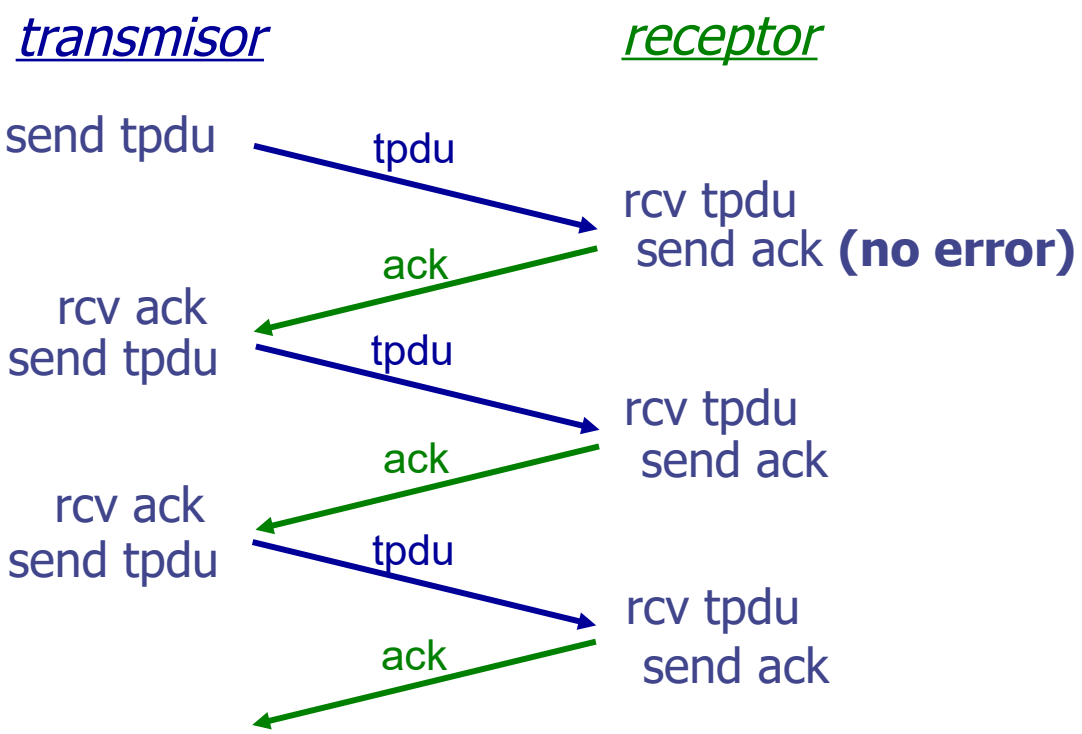
Protocolo Simple en una red con ruido - RDT 2.0

- Un Transmisor y un Receptor, solo envía datos de aplicación el transmisor.
- Errores en Capas Inferiores: Los segmentos se pueden corromper o perder totalmente.
- **Receptor:** Se agrega código detector de errores (TCP Checksum)
- **¿Transmisor?**
  - **ACK:** sólo se reconocen (ACK) los segmentos que llegan bien (el receptor envía una segmento indicando que se recibió correctamente).
  - **Reloj:** Necesito de Temporizadores para recuperación de errores cuando no llega el reconocimiento.
- **Transport Protocol Data Unit (TPDU):** De forma genérica, a los datos de aplicación con la información de control (encabezados).

  En TCP lo llamamos segmentos
- **Simplificación: TPDU de largo fijo.**

Protocolo Simple en una red con ruido - RDT 2.0

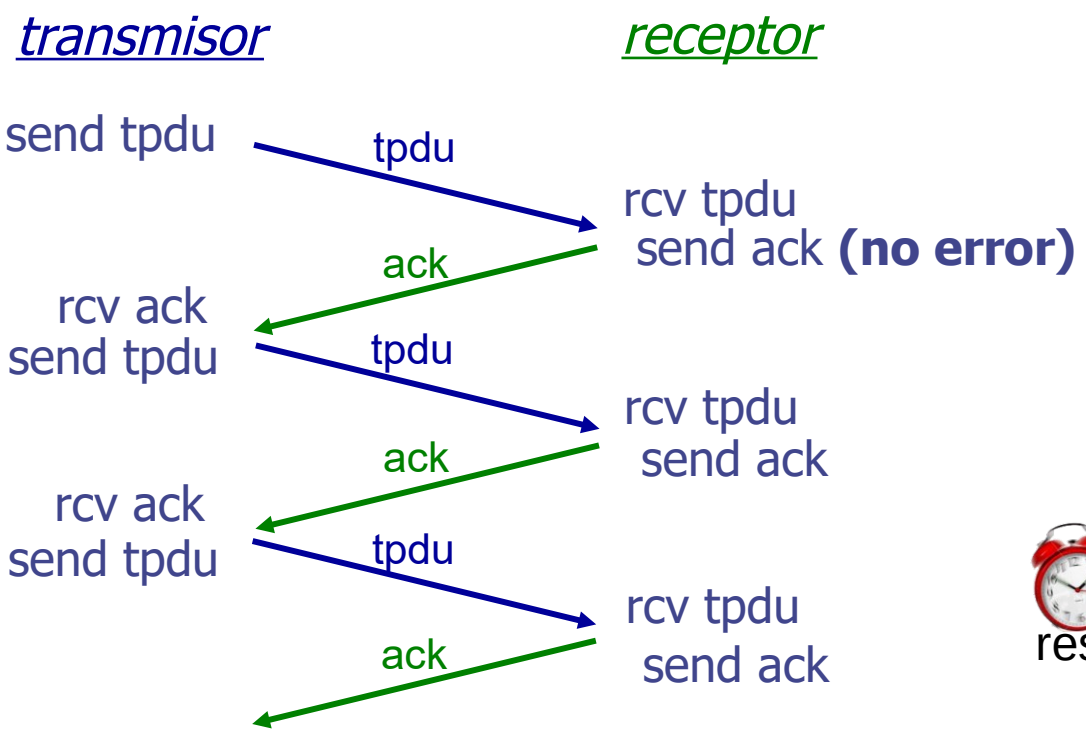
send tpdu ~ udt_send
rcv_tpdu ~ rdt_recv



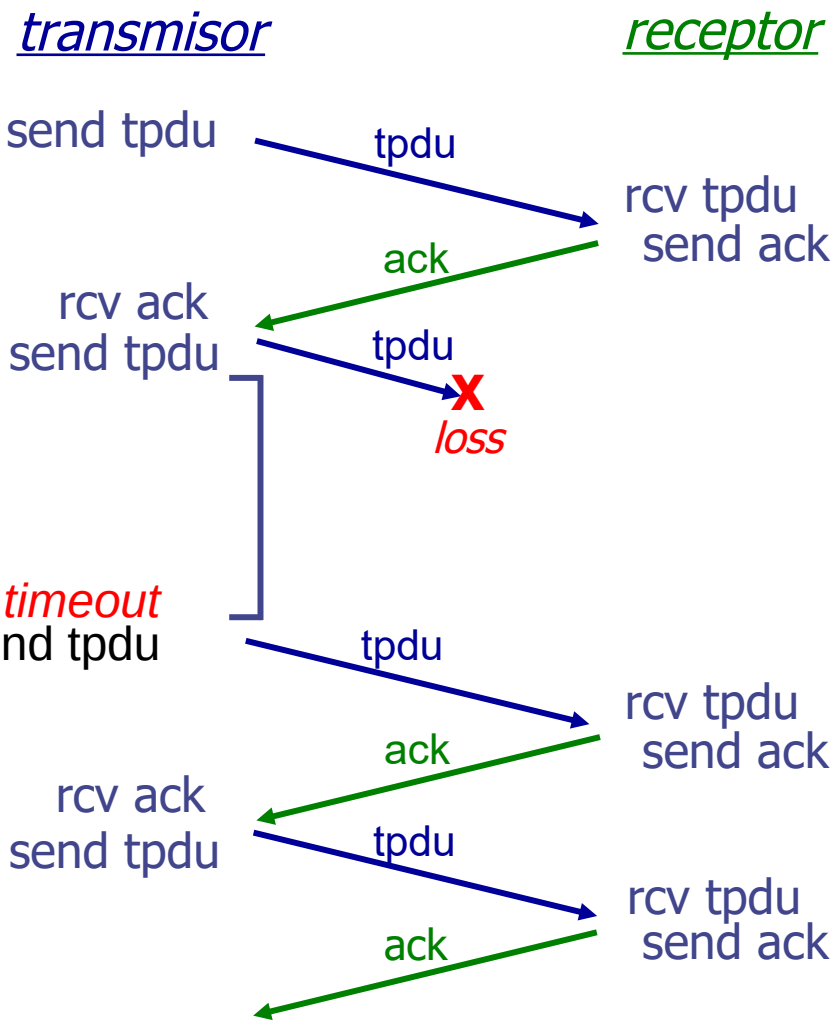
(a) Sin pérdidas

(b) Con pérdidas

Protocolo Simple en una red con ruido - RDT 2.0



(a) Sin pérdidas



(b) Con pérdidas

Protocolo Simple en una red con ruido - RDT 2.0

transmisor

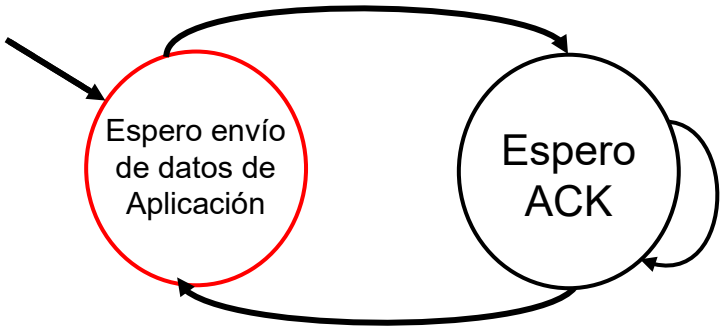
eventos

acciones

receptor

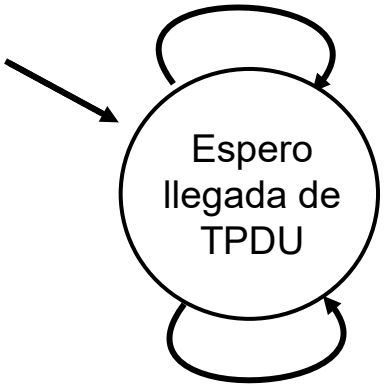
rdt_send(data)

sndpkt = make_pkt(data, checksum)
udt_send(sndpkt)
start_retran_timer



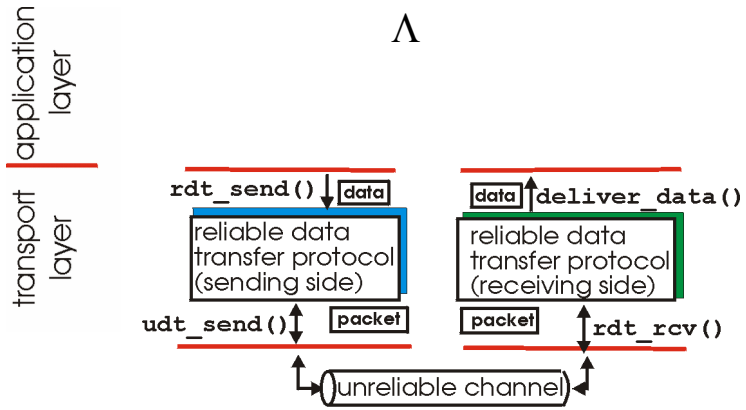
rdt_rcv(rcvpkt) &&
corrupt(rcvpkt) ||
timeout

udt_send(sndpkt)
start_retran_timer



rdt_rcv(rcvpkt) && isACK(rcvpkt) && notcorrupt(rcvpkt)

Λ



Protocolo Simple en una red con ruido - RDT 2.0

transmisor

eventos

acciones

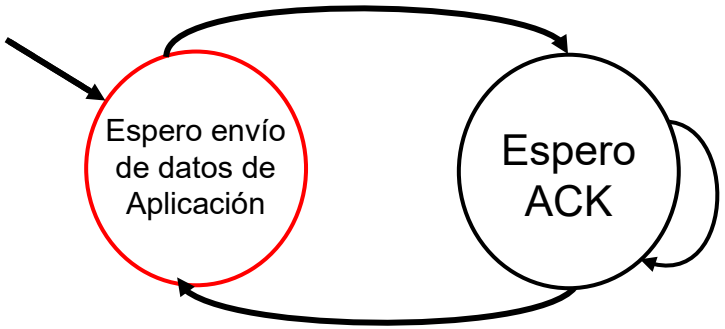
receptor

rdt_send(data)

sndpkt = make_pkt(data, checksum)
udt_send(sndpkt)
start_retran_timer

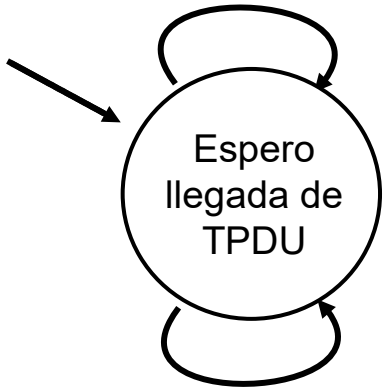
rdt_rcv(rcvpkt) &&
corrupt(rcvpkt)

Λ



rdt_rcv(rcvpkt) &&
corrupt(rcvpkt) ||
timeout

udt_send(sndpkt)
start_retran_timer

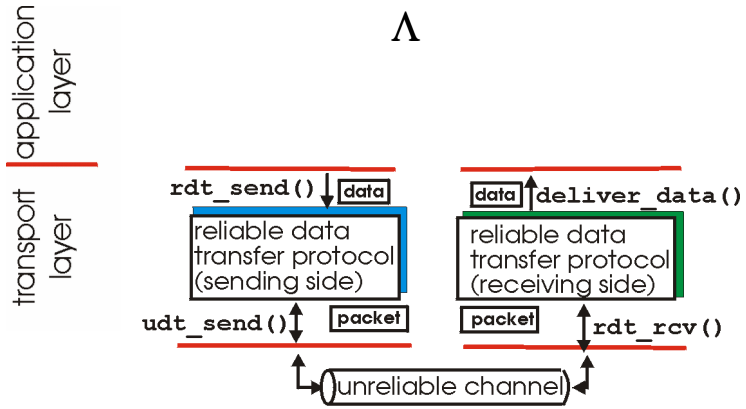


rdt_rcv(rcvpkt) && isACK(rcvpkt) && notcorrupt(rcvpkt)

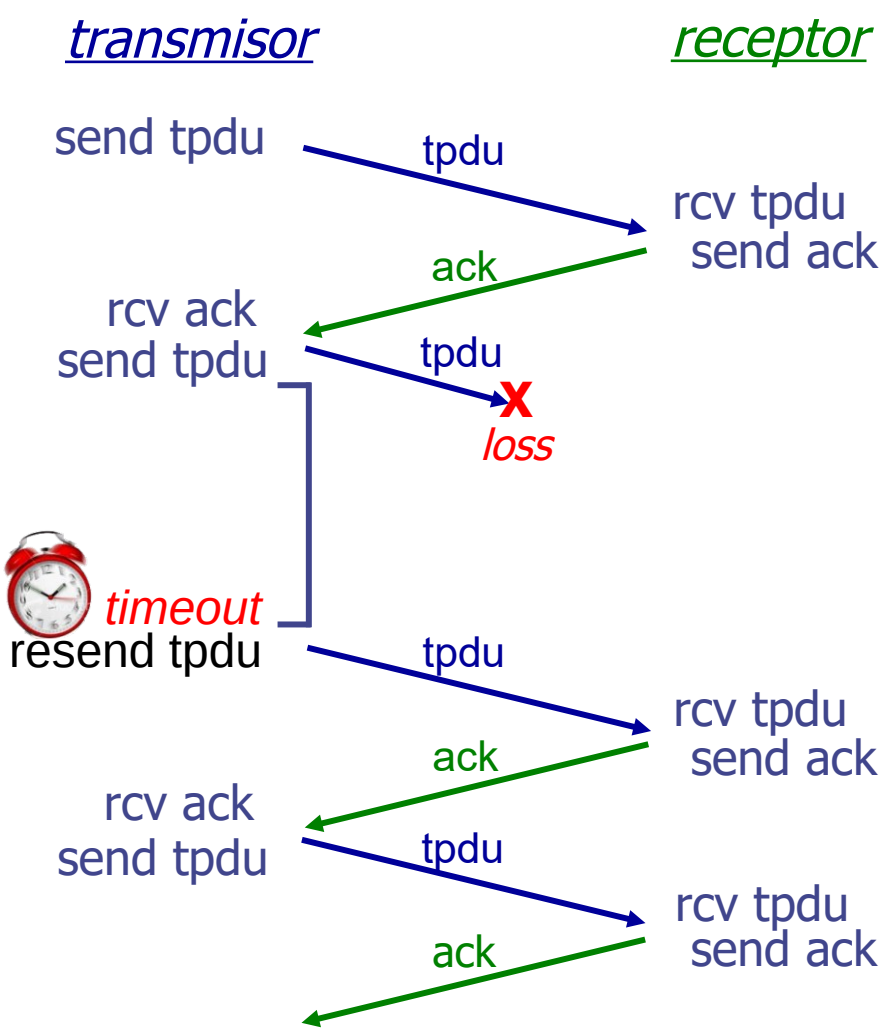
Λ

rdt_rcv(rcvpkt) &&
notcorrupt(rcvpkt)

extract(rcvpkt,data)
deliver_data(data)
udt_send(ACK)

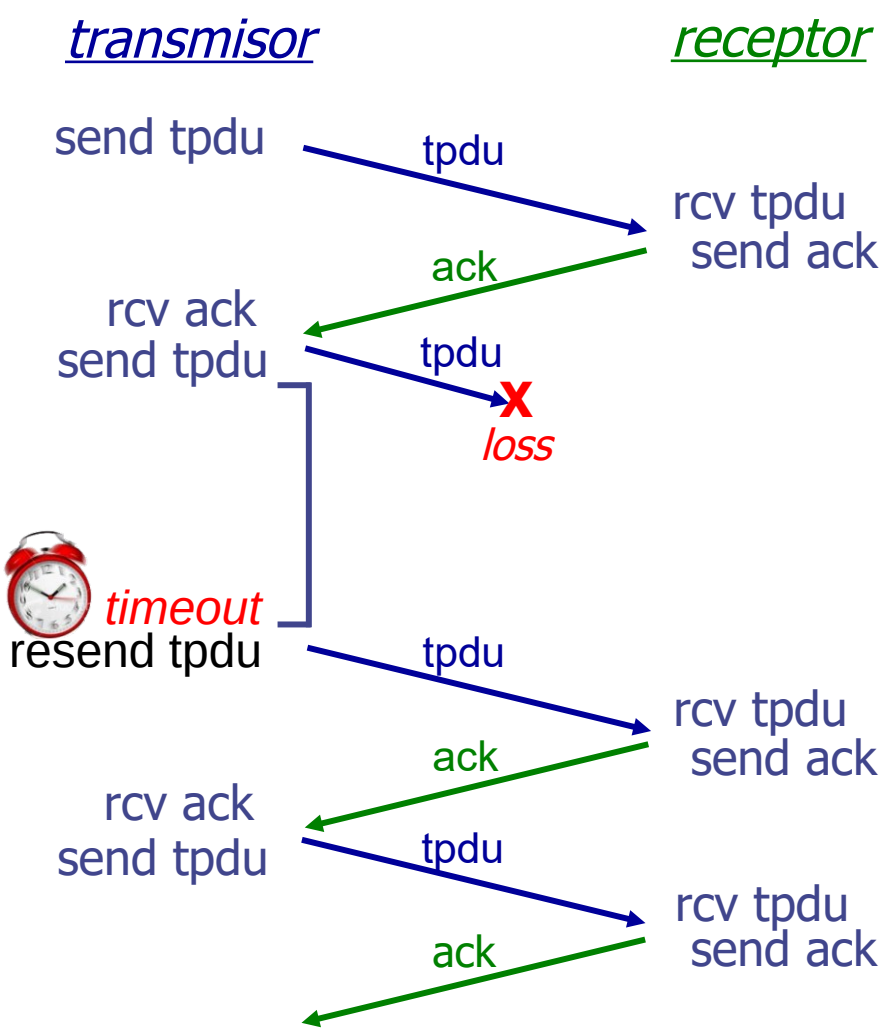


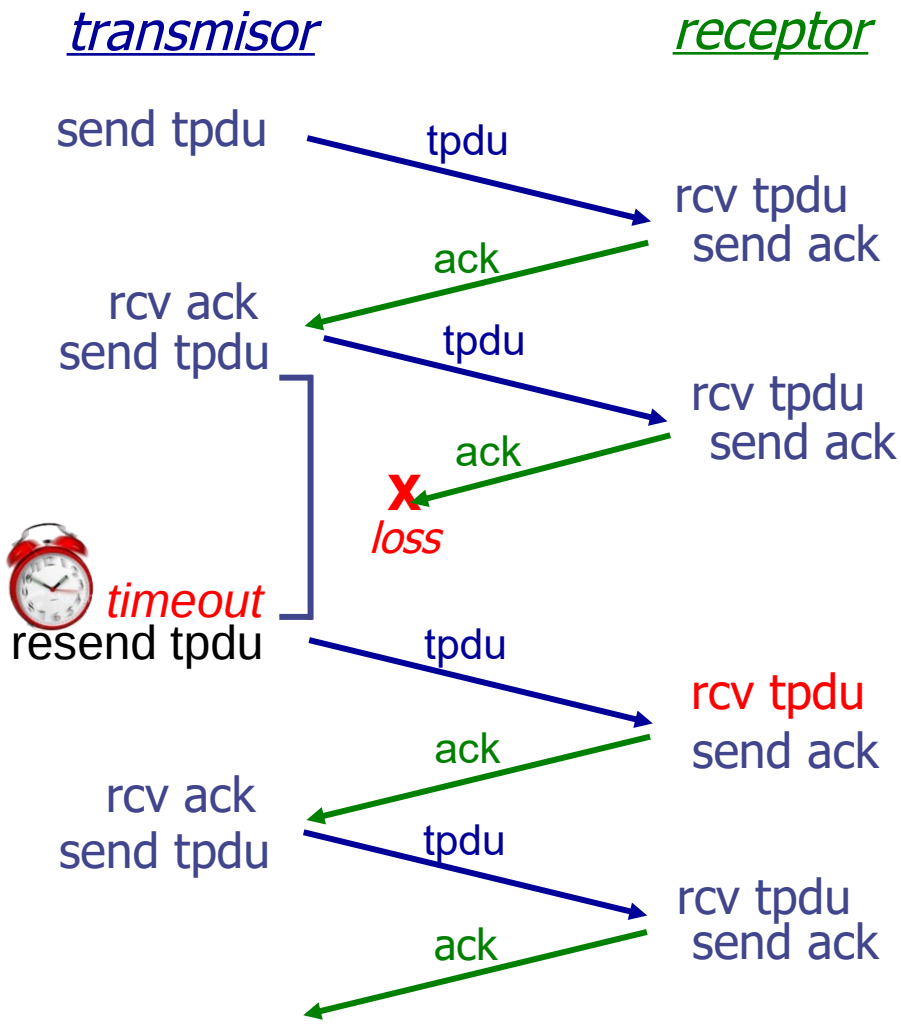
¿Pérdida de TPDU (segmento) o de reconocimiento (ACK)?



transmisor receptor

¿Pérdida de TPDU (segmento) o de reconocimiento (ACK)?



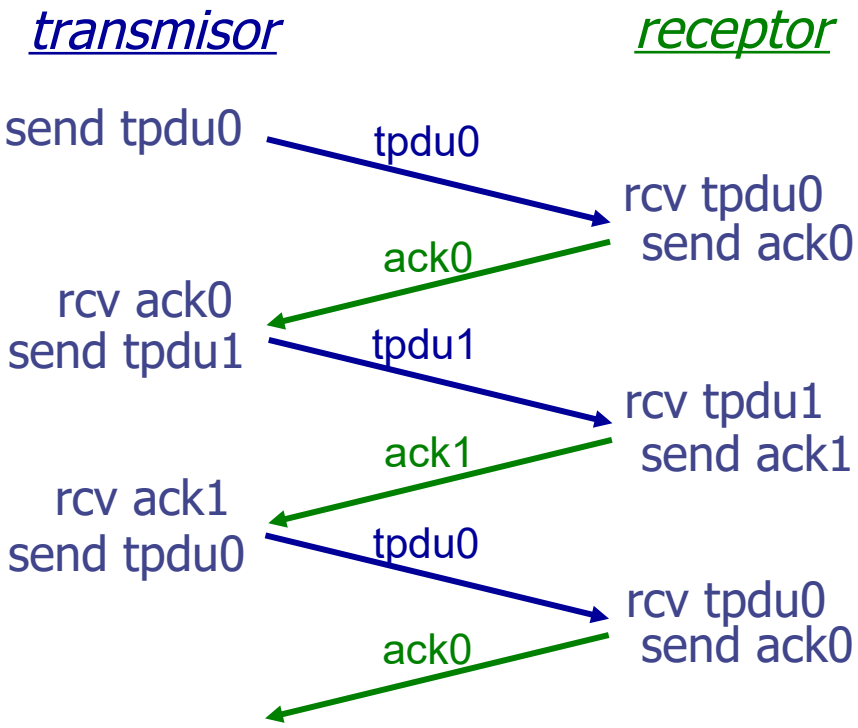


¿Pérdida de TPDU (segmento) o de reconocimiento (ACK)?

- **Problema:** se pierde ACK, el transmisor reenvía, se duplica una TPDU en el receptor.
  - Necesidad distinguir entre una TPDU nueva de una re-enviada, surgen los **números de secuencia.**
- **¿Cuántos números de secuencia se necesitan?**
  - En este protocolo solo **2**: 0 y 1 pues hasta no recibir reconocimiento del 0 no intento mandar el 1
  - Cuando se espera la TPDU 0 se rechazan las que no sean 0, al recibirse la 0 se espera por la 1
- **Simplificación:** La “red” no re-ordena TPDU (recordar que los paquetes pueden tomar caminos distintos)

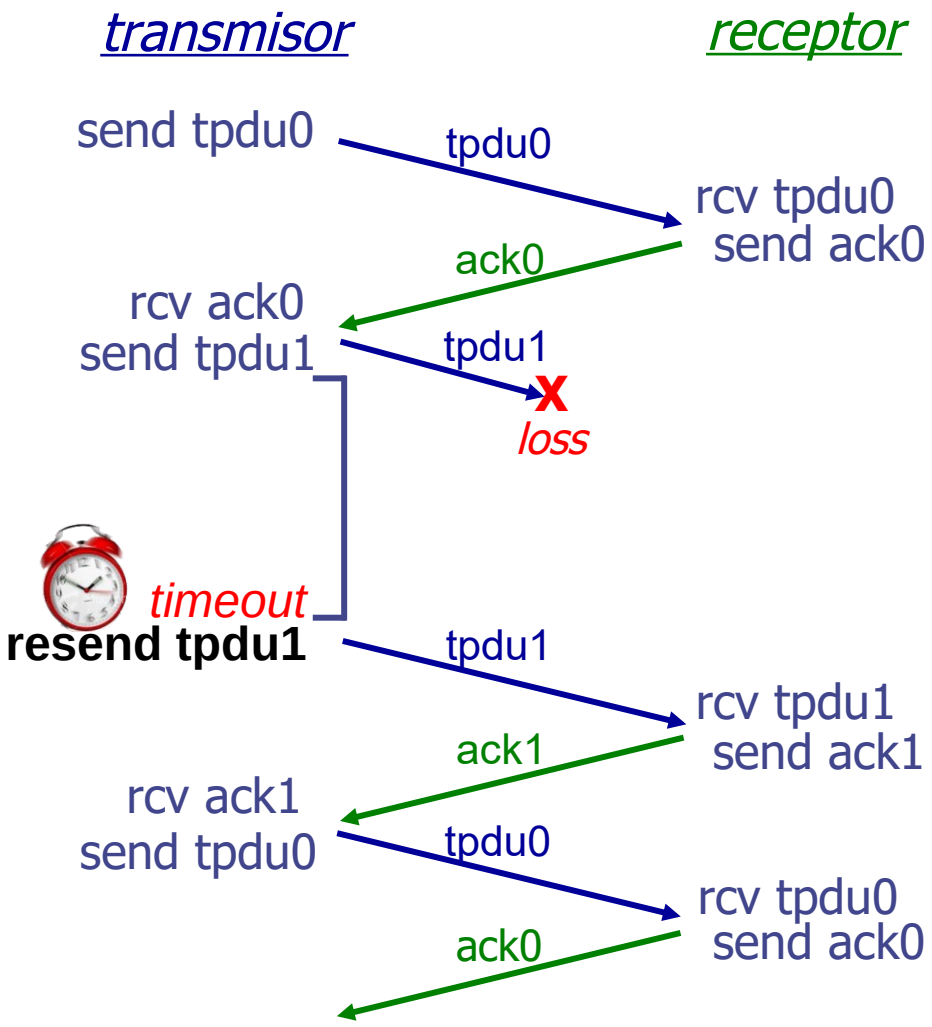
Protocolo Simple RDT 2.1 – Número de secuencia n = 1

send tpdu ~ udt_send
rcv_tpdu ~ rdt_recv



(a) Sin pérdida

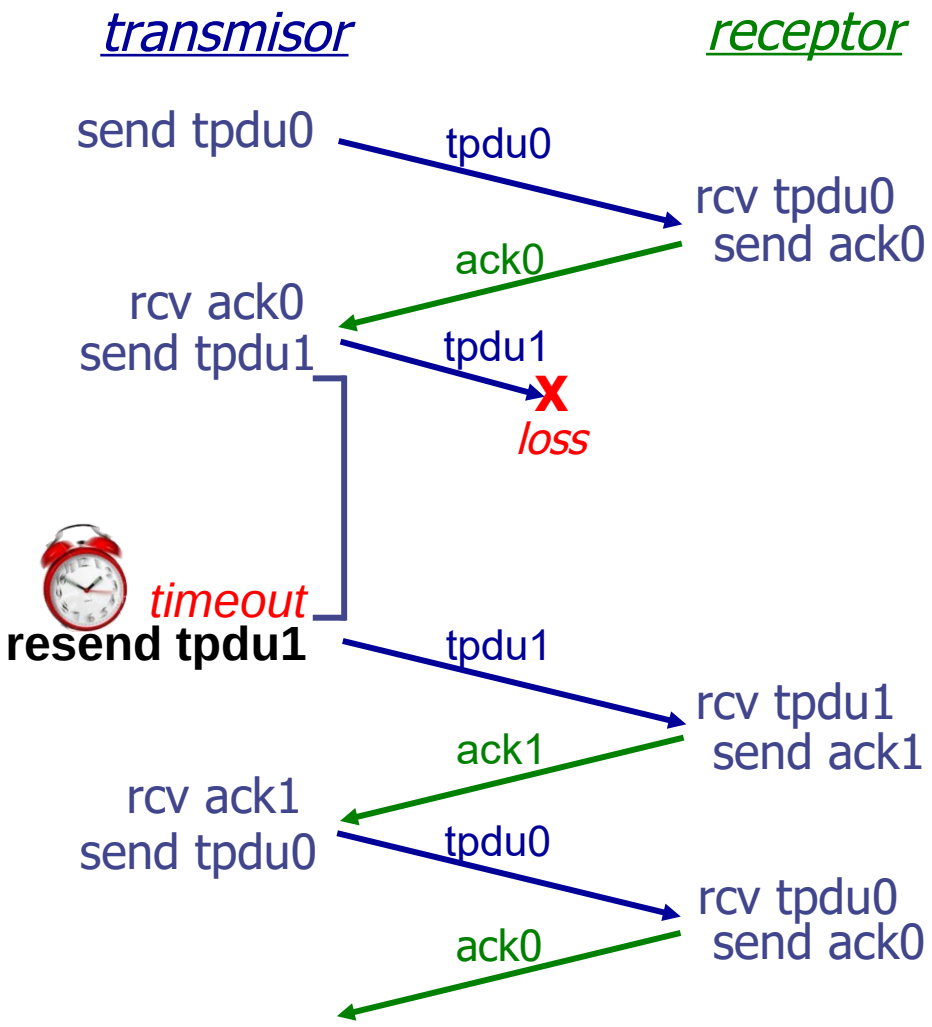
Protocolo Simple RDT 2.1 – Número de secuencia n = 1



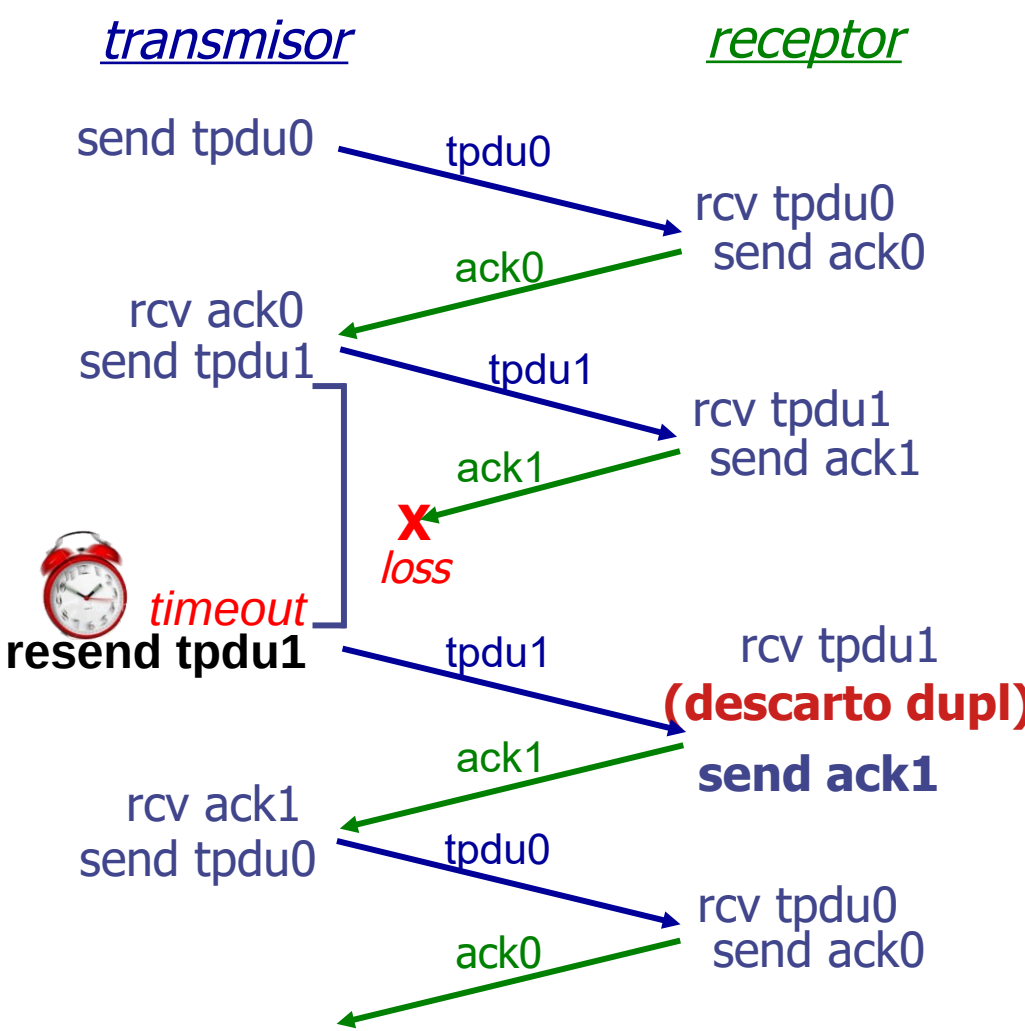
(a) TPDU pérdida

(b) ACK perdido

Protocolo Simple RDT 2.1 – Número de secuencia n = 1

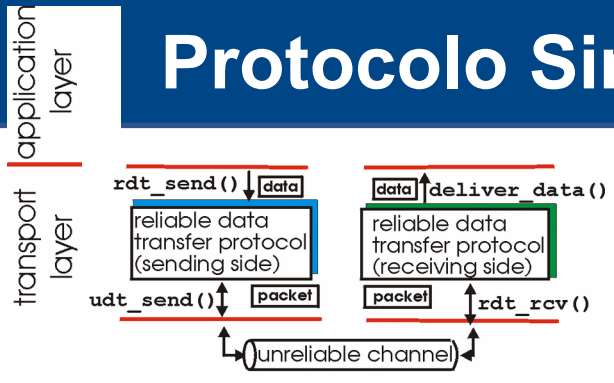


(a) TPDU pérdida



(b) ACK perdido

Protocolo Simple RDT 2.1 – Estados Transmisor



rdt_send(data)

sndpkt0 = make_pkt(**0**, data, checksum)
udt_send(sndpkt0)
start_timer

rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isACK(rcvpkt,**1**))

Λ

rdt_rcv(rcvpkt)

Λ

Desde capa Aplicación 0

Espero ACK 0

timeout

udt_send(**sndpkt0**)
start_timer

rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& **isACK**(rcvpkt,**1**)

stoptimer

rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& **isACK**(rcvpkt,**0**)

stoptimer

Re-envío TPDU

timeout

udt_send(**sndpkt1**)
start_timer

Espero ACK 1

Desde capa Aplicación 1

rdt_rcv(rcvpkt)

Λ

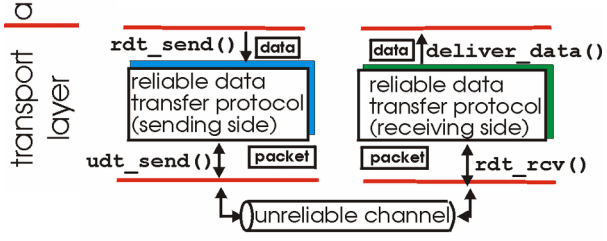
rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isACK(rcvpkt,**0**))

Λ

rdt_send(data)

sndpkt1 = make_pkt(**1**, data, checksum)
udt_send(sndpkt1)
start_timer

Protocolo Simple RDT 2.1 – Estados Receptor



rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& **has_seq0**(rcvpkt)

extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(**ACK0**, chksum)
udt_send(sndpkt)

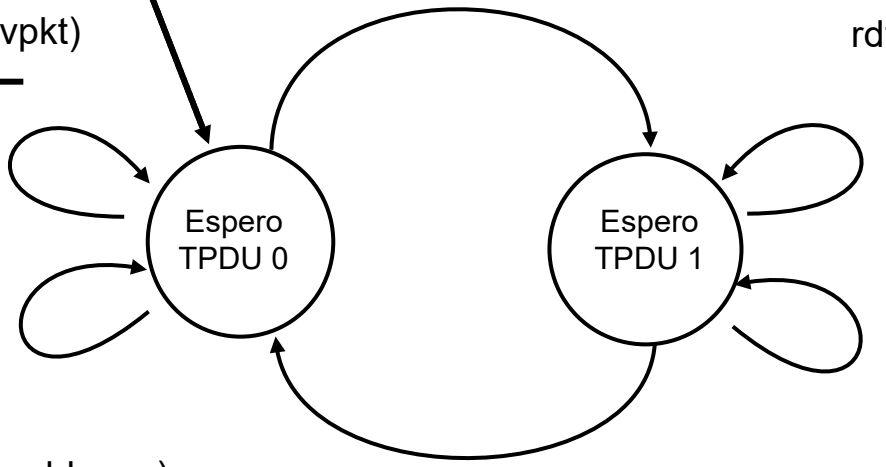
Espero TPDU 0 y llega TPDU 0

rdt_rcv(rcvpkt) && (corrupt(rcvpkt)

Λ

rdt_rcv(rcvpkt) && (corrupt(rcvpkt)

Λ



rdt_rcv(rcvpkt) &&
not corrupt(rcvpkt) &&
has_seq1(rcvpkt)

sndpkt = make_pkt(**ACK1**, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
not corrupt(rcvpkt) &&
has_seq0(rcvpkt)

sndpkt = make_pkt(**ACK0**, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& **has_seq1**(rcvpkt)

extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(**ACK1**, chksum)
udt_send(sndpkt)

Espero TPDU 0 y llega TPDU 1 (retransmisión) Envío último ACK

Protocolo Simple - Eficiencia

- A este tipo de protocolos se les llama protocolos de "**stop and wait**" o "**stop and go**" (envío TPDU y espero ACK).
- Ejemplo: enlace 1 Gbps link (**R**), **RTT** 30 ms, 8000 bit trama (**L** ~"TPDU/segmento"):

$$D_{serailización} = \frac{L}{R} = \frac{8000\ bits}{10^9\ bits/sec} = 8\ microsecs$$

U $_{transmisor}$: *utilización* – fracción de tiempo que en transmisor está transmitiendo

$$U_{transmisor} = \frac{L/R}{RTT+L/R} = \frac{0.008}{30 + 0.008} = 0{,}00027\ (0{,}027\ \%)$$

$$B_{Efectiva} = \frac{L}{RTT+L/R} = \frac{8000\ bits\ x1000}{(30 + 0.008)} = 266.596\ bps$$

- **Eficiencia:** sensible a "**RTT** x **R**"

Protocolo Simple – Mejoras de Eficiencia

- Es habitual el uso de la red para transmitir datos en ambas direcciones (números de **Sec** y **ACK**)
- Como el campo **ACK** es parte del **encabezado**, puedo enviar información en la misma TPDU que lleva el ACK

  Esto se llama piggybacking

- Problema: Si no tengo datos, ¿espero? ¿Cuánto?
  - POCO: debo mandar una TPDU solo con ACK
  - MUCHO: el transmisor retransmite porque expira un temporizador

Protocolo Simple – Mejoras de Eficiencia

- **Pipeline o Tubería:** ¿Me cambia la eficiencia que existan varias TPDU en camino al receptor sin ser reconocidas?

  Por ejemplo: envío 1000 TPDUs

$$U_{transmisor} = \frac{1000 \times L/R}{RTT + L/R} = \frac{8}{30 + 0.008} = 0{,}27\ (27\ \%)$$

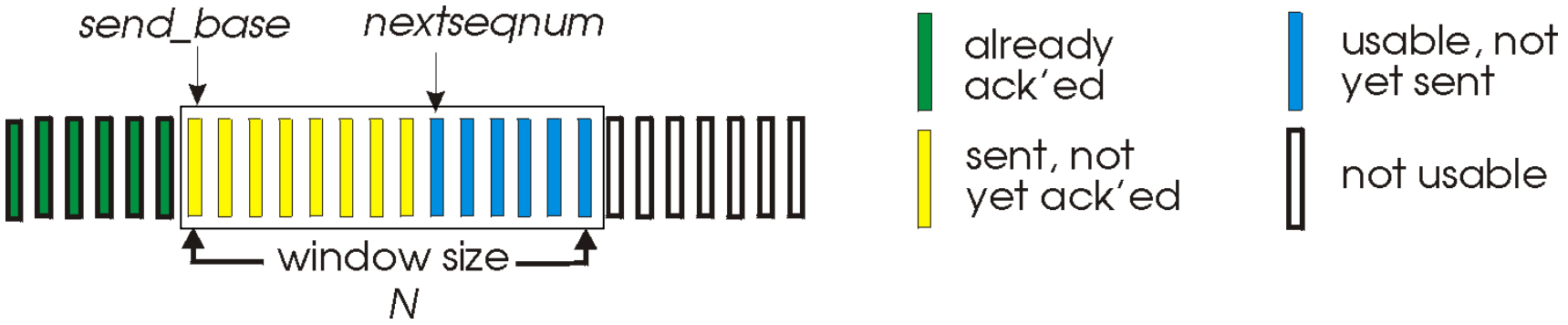
- Enviar varias TPDU incrementa la complejidad de nuestro protocolo, debemos **incrementar** la cantidad de bits destinados para **números de secuencia** (identificar que TPDU no llegó).
- Surge el concepto de **Ventana Deslizante**
- El transmisor puede implementar dos estrategias:
  - **Go Back N:** El receptor **NO** almacena TPDU desordenadas, el transmisor re-envía todo lo que no tuvo un ACK.
  - **Selective Repeater:** El receptor permite almacenar TPDU **desordenadas**. De “alguna forma” identifica las TPDU faltantes y el transmisor las re-envía.

Mejoras de Performance – Ventanas Deslizantes

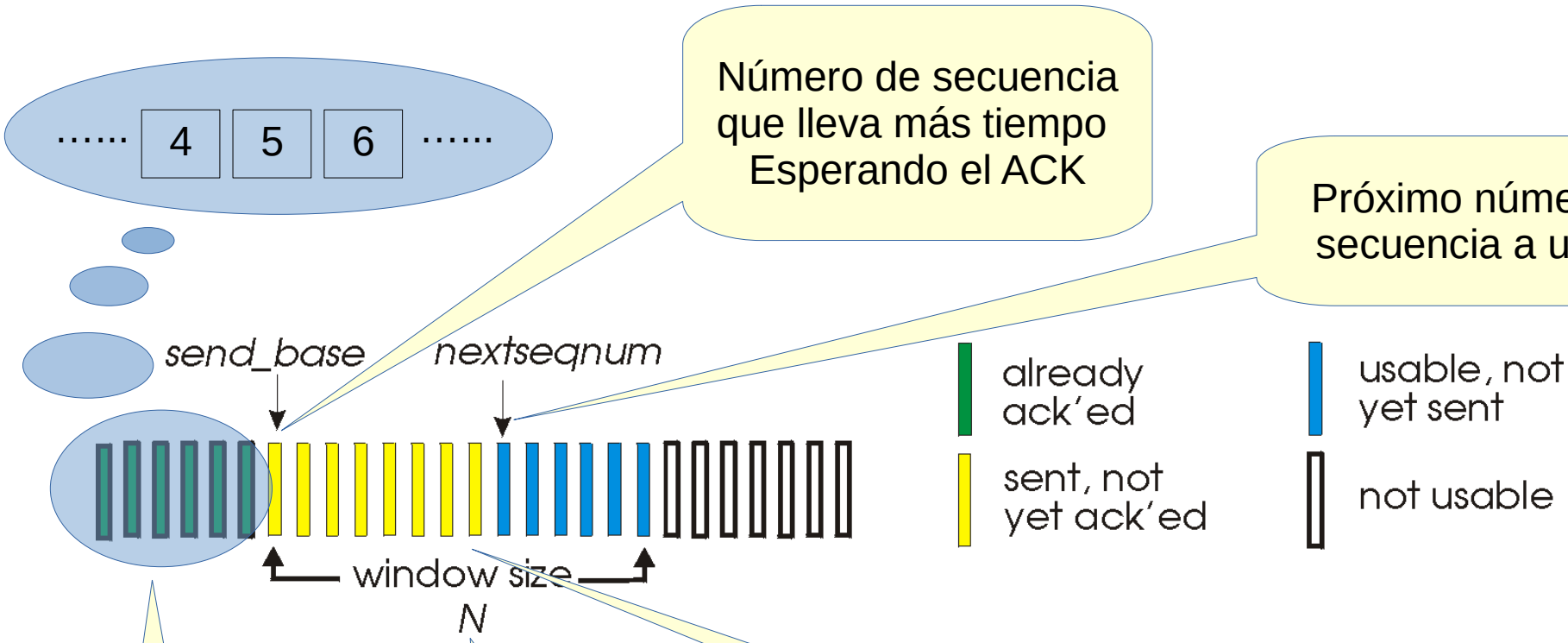
- Si tenemos n bits para el N° de secuencia:

  Números de secuencia de 0 a 2^{n-1} = MAX_SEQ

- **Ventana de transmisión (TX)**
  - números de secuencia de las TPDU que puedo enviar, o ya han sido enviadas y aún no han sido reconocidas
- **Ventana de recepción (RX)**
  - números de secuencia de las TPDU que se pueden aceptar en un determinado instante
- Las ventanas de TX y RX no tienen por que tener los mismos límites ni el mismo tamaño



Mejoras de Performance – Ventanas Deslizantes



Mejoras de Performance – Ventanas Deslizantes

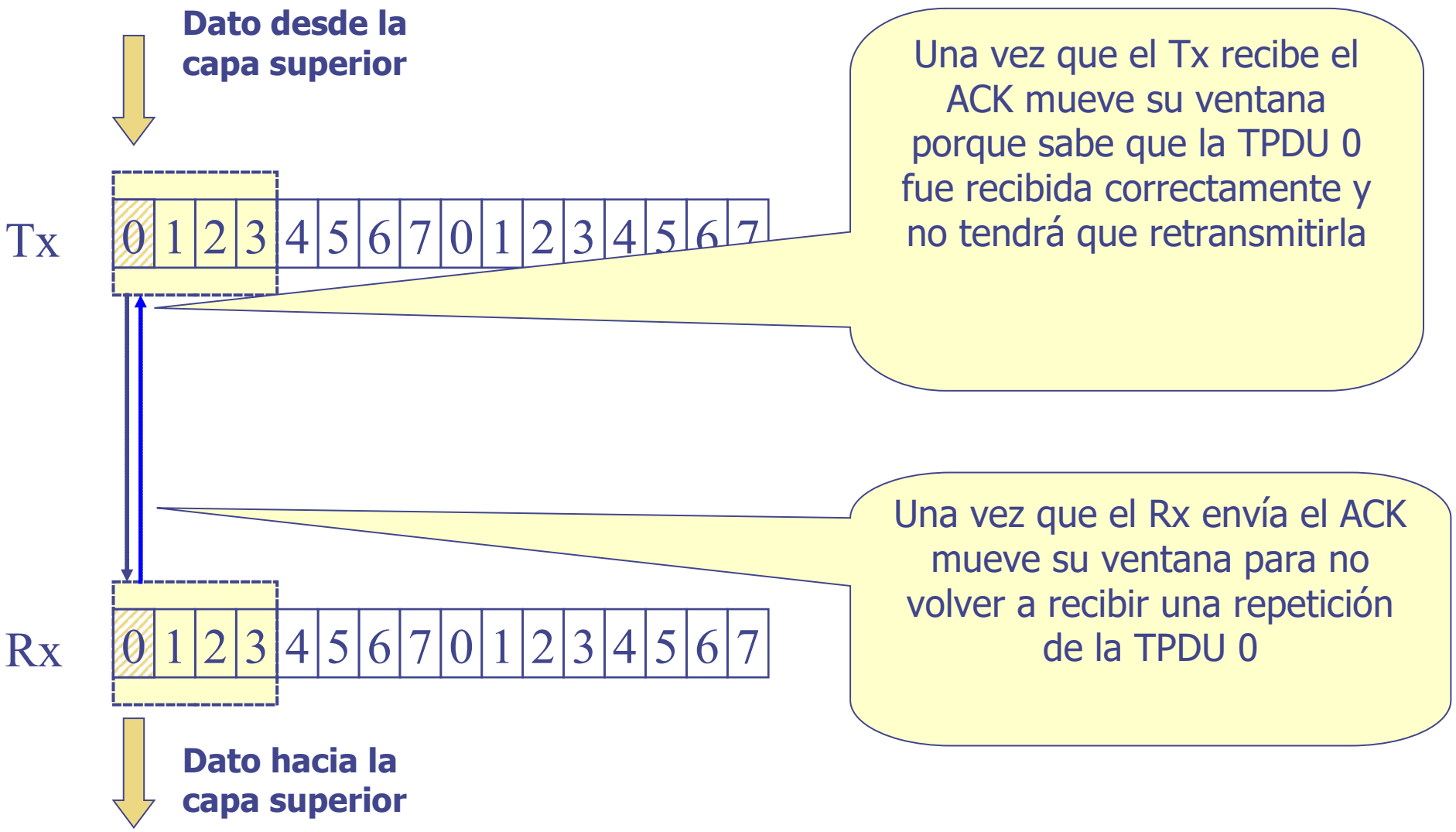
- El uso de ventanas permite a la capa de transporte más libertad en el orden de transmisión y recepción de las TPDUs

  PERO: agrega complejidad
- Si el receptor acepta TPDUs en desorden, debe guardarlas hasta recibir las faltantes para entregarlas en orden a la capa de aplicación (necesidad de buffers).
- El transmisor debe mantener una copia de las TPDUs enviadas en un buffer para re-enviarlas si no recibe reconocimiento
- Se deben gestionar varios temporizadores (uno por TPDU).

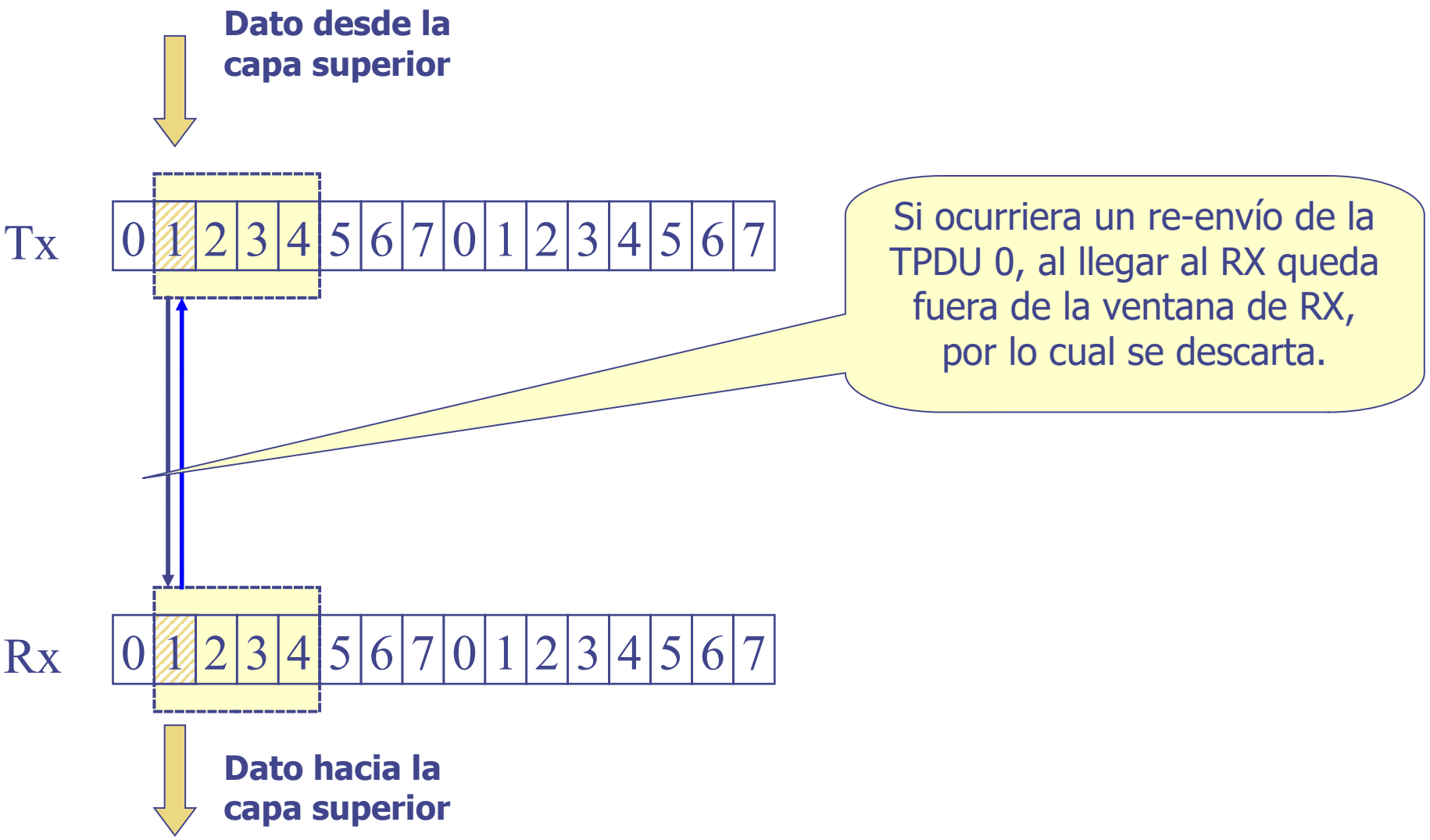
Mejoras de Performance – Ventanas Deslizantes

- Si no se dispone lugar en la ventana de TX no se deben aceptar datos de la aplicación
- La ventana del receptor son los números de secuencia que puede aceptar en las TPDUs
- Si se recibe algo fuera de la ventana se descarta
- Puede ser necesario enviar un reconocimiento a modo de refresco de estado (lo último reconocido hasta ese nomento).
- Si lo recibido coincide con el límite inferior, se entrega a la capa de aplicación los datos, se envía ACK y se avanza la ventana ("**desliza**")

Ventanas Delizantes

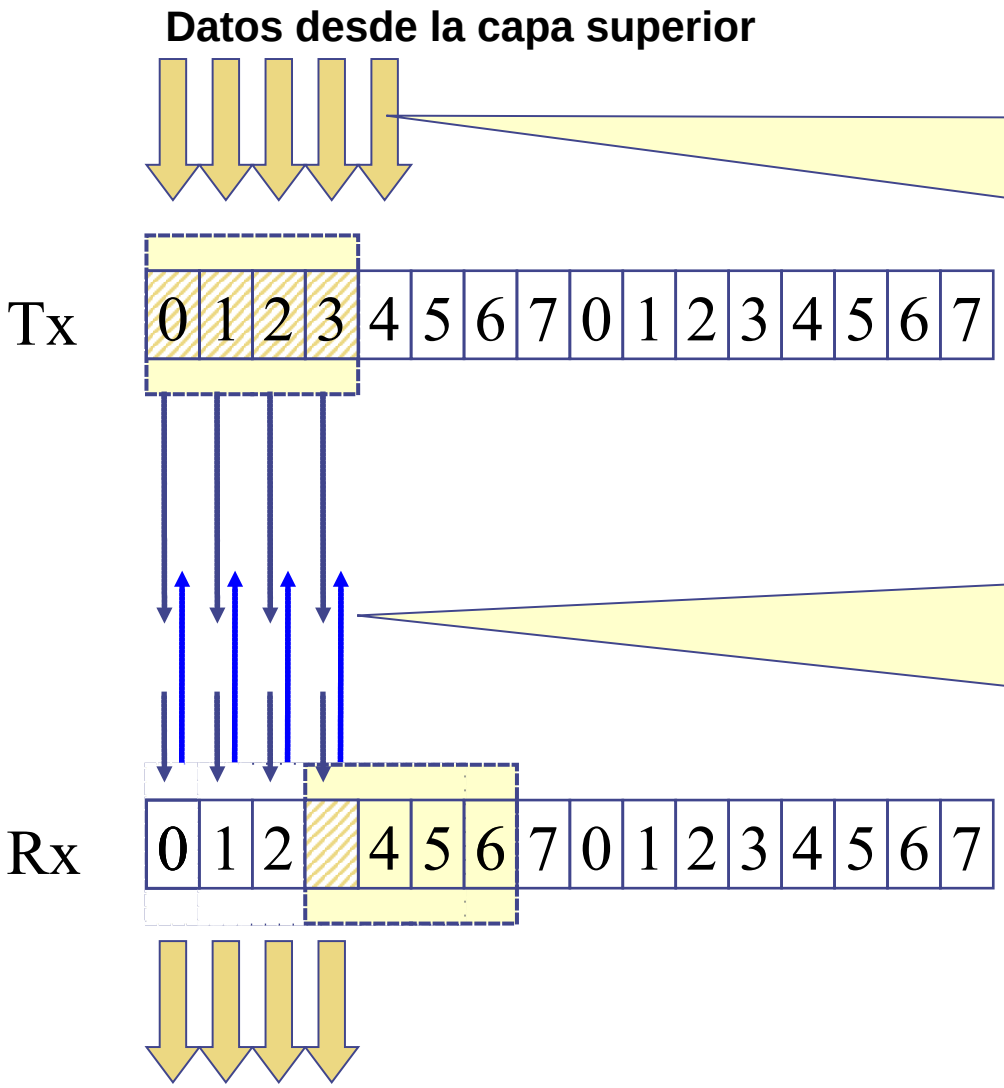


Ventanas Delizantes



Ventanas Delizantes

cuando los reconocimientos (ACK) demoran

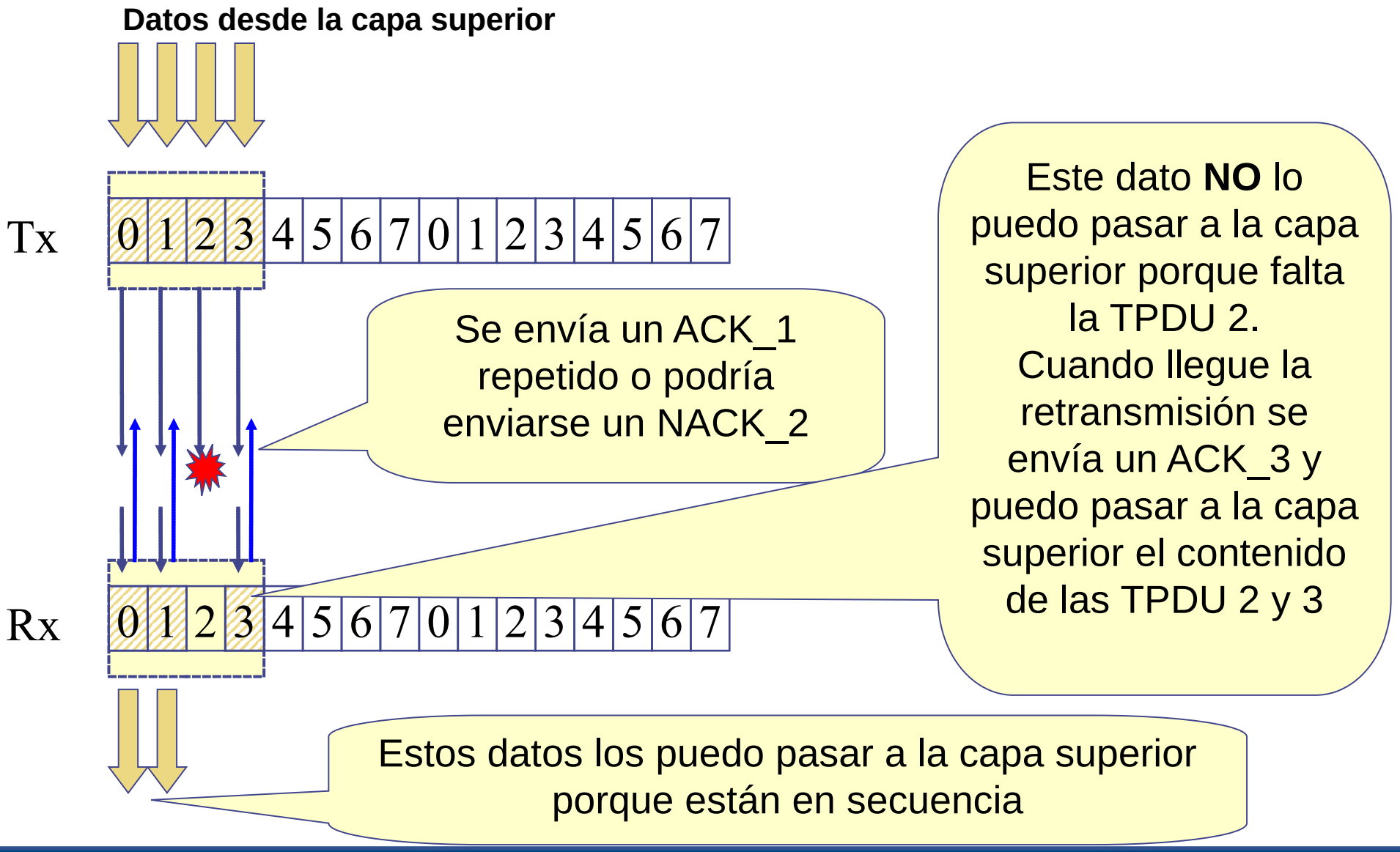


Este dato no puede aceptarse porque no hay espacio en la ventana

A medida que el Tx comienza a recibir los ACKs puede mover su ventana y aceptar nuevos datos de la capa superior

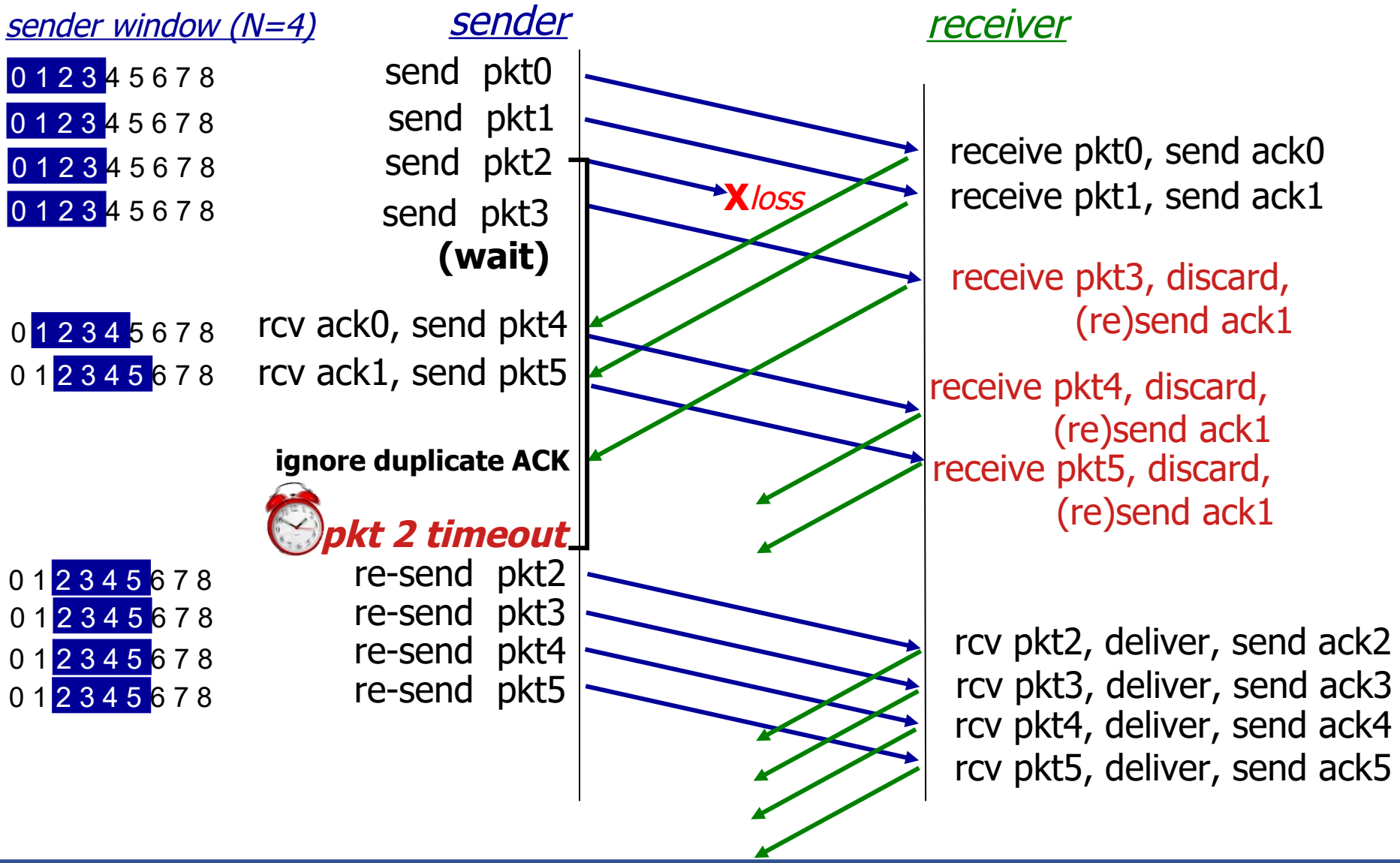
Ventanas Delizantes

cuando hay errores y se pierden datos



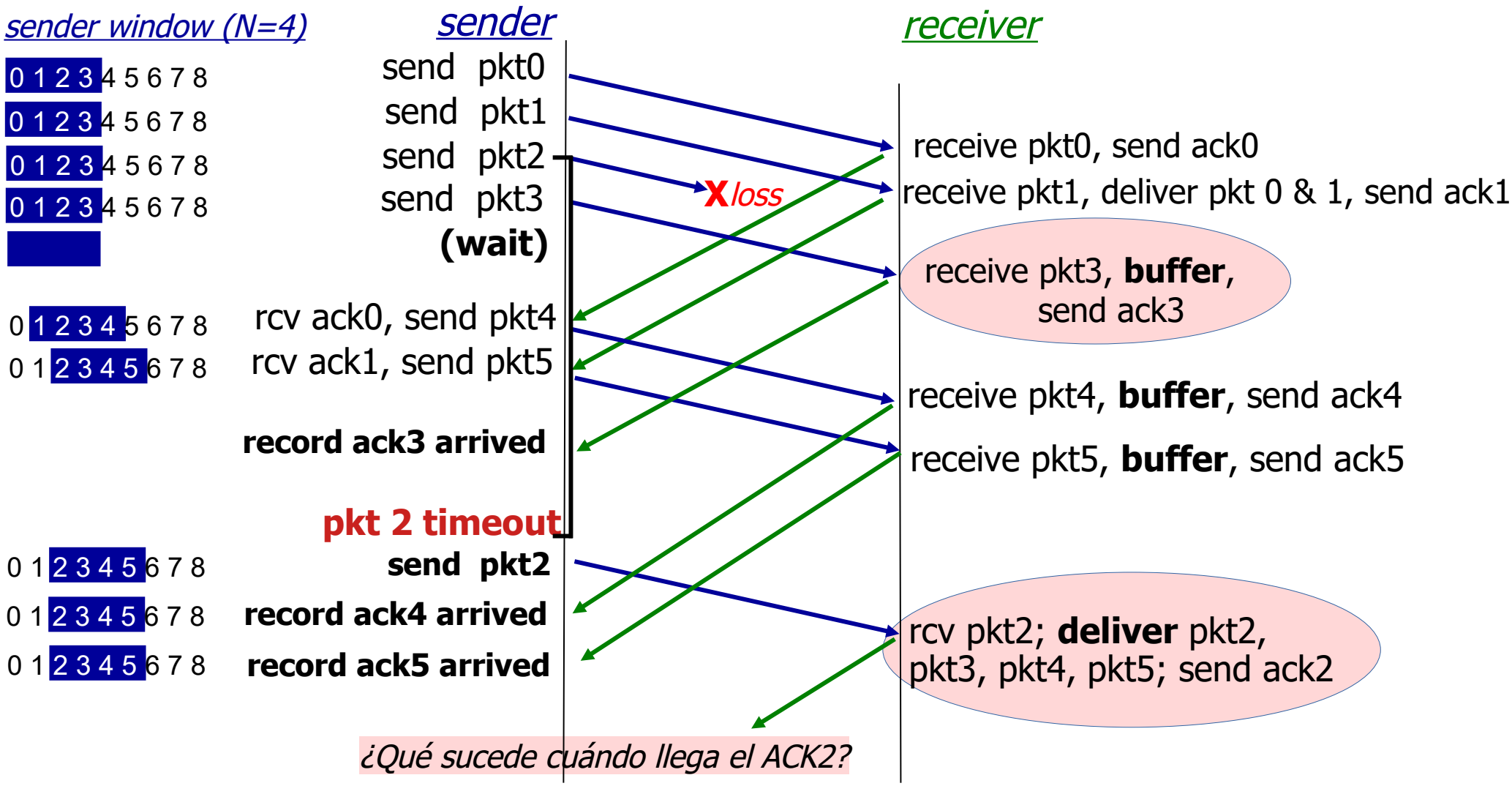
Go Back N – TPDU deben llegar ordenadas

- Los ACK son acumulativos, ACK=X reconoce todos los números de secuencia hasta X, frente a la detección de una pérdida, reitero el último ACK



Selective Repeat – Permito TPDU desordenadas

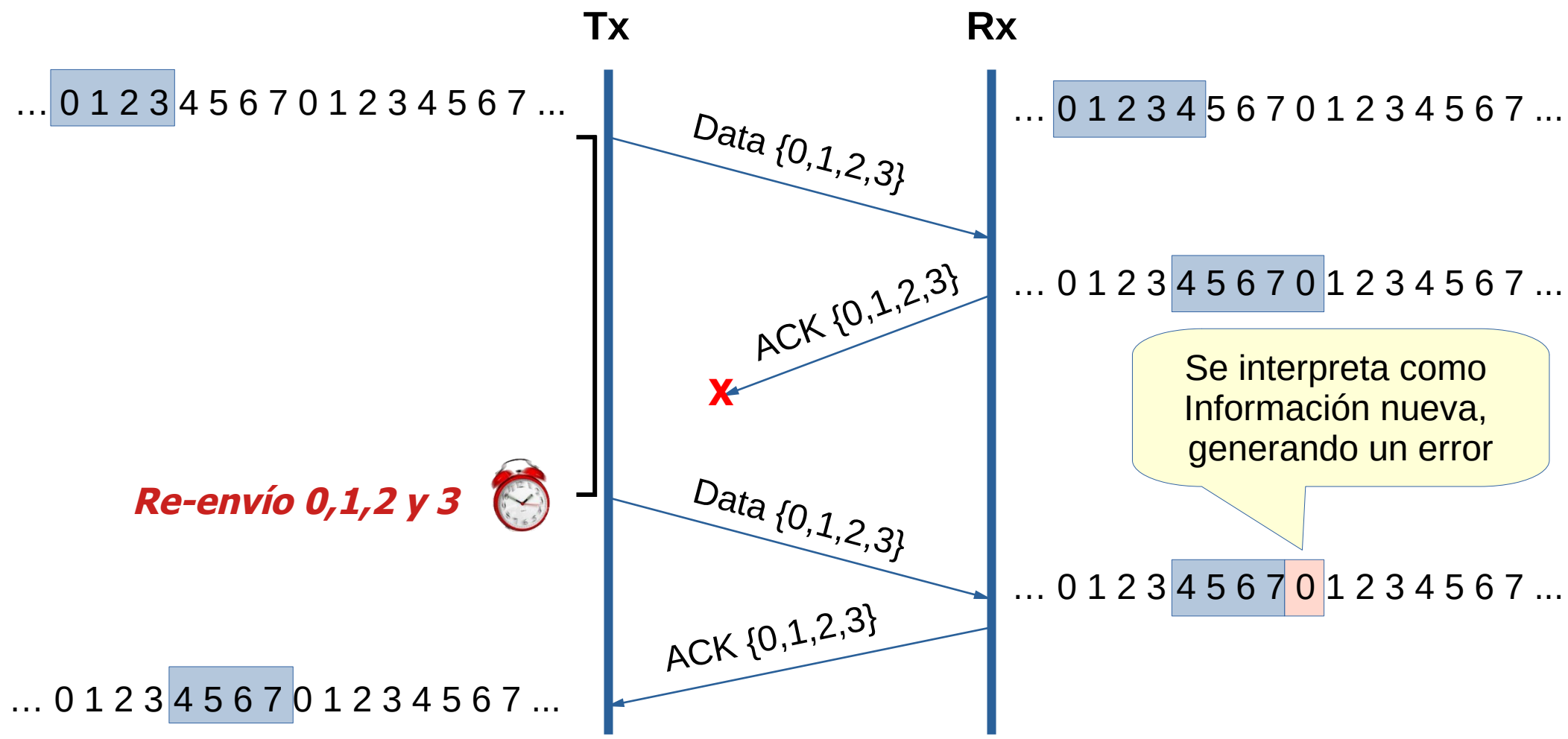
- Los ACK son **selectivos**, ACK=X solo reconoce el número de secuencia X. El transmisor podría detectar la pérdida y acelerar la re-transmisión.
- Existe la alternativa de enviar NACK (negative ACK) ó ACK acumulativos (TCP).



Ventanas Deslizantes – Tamaño Máximo de Ventana

- **Ventana de recepción W_{RX}**
  - Si **W_{RX}** = MAX_SEQ (Ejemplo: MAX_SEQ=7)
    - TX envía 0 a 3, RX los recibe bien y envía ACKs
    - RX avanza la ventana para admitir 4 a 2
    - Se pierden los ACKs y el TX retransmite el 0
    - El "0" cae en la ventana del RX, se acepta como nuevo
  - Falla: la nueva ventana se superpone con la anterior
- Para evitar la falla: **W_{RX}** <=(MAX_SEQ+1)/2
- **Buffers (memoria) VS Ventana (disponible)**
  - Buffer = memoria asignada al receptor
  - **ventana de recepción =** "número de buffers disponibles del receptor". No la cantidad (universo) de números de secuencia
  - Es posible utilizar de un espacio de números de secuencia grande, e igual trabajar con equipos con poca capacidad de memoria.

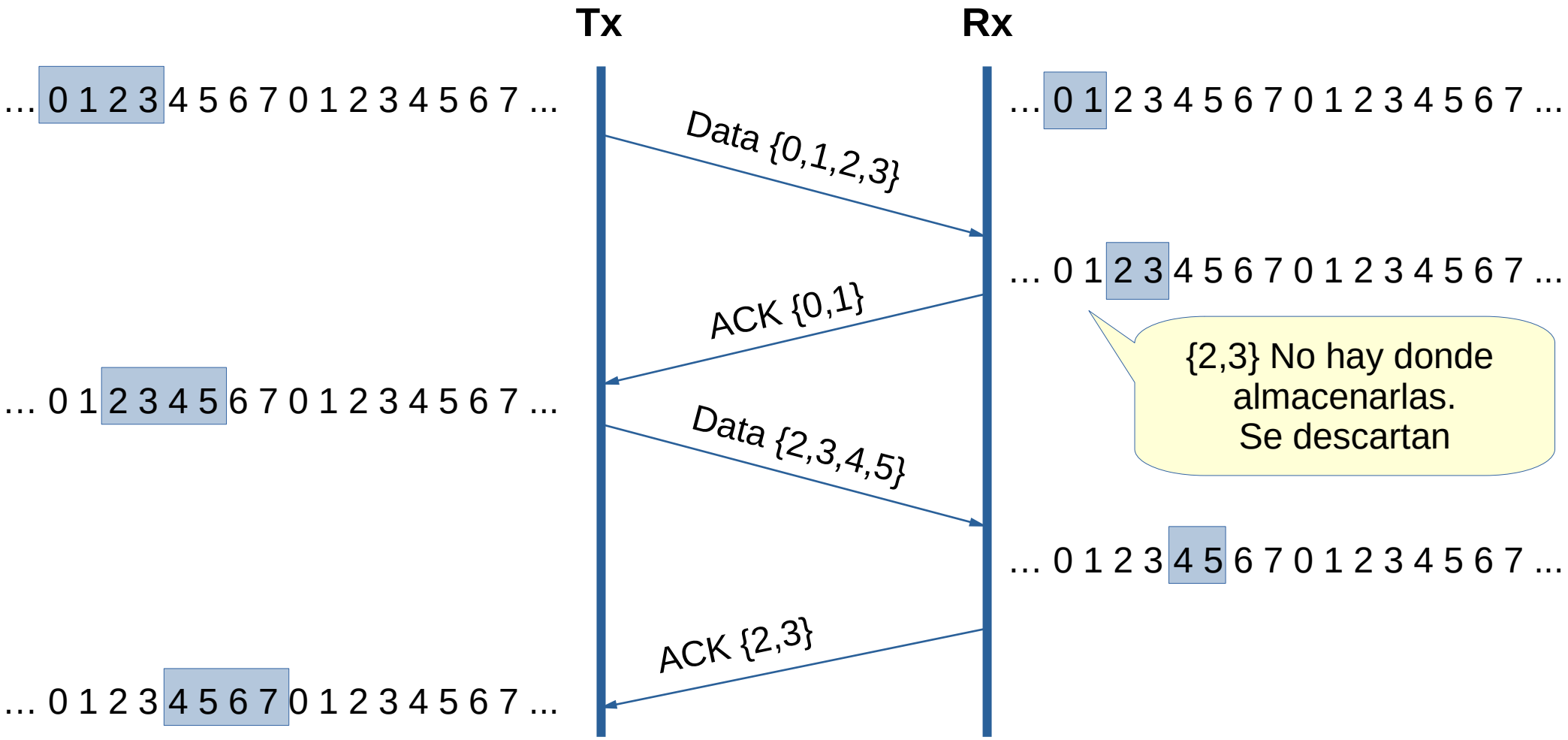
Ventanas Deslizantes – Tamaño Máximo de Ventana



Al deslizar la ventana de RX, la ventana “vieja” y la “nueva” tienen números de secuencia en común.

Para evitar el escenario: $W_{RX} \leq (MAX\_SEQ+1)/2$

Ventanas Deslizantes – Tamaño Máximo de Ventana



Transmito TPDUs que se que si llegan al destino, este no las aceptará. No tiene sentido práctico

$W_{Tx} \leq W_{Rx}$ para evitarlo (**control de flujo**)

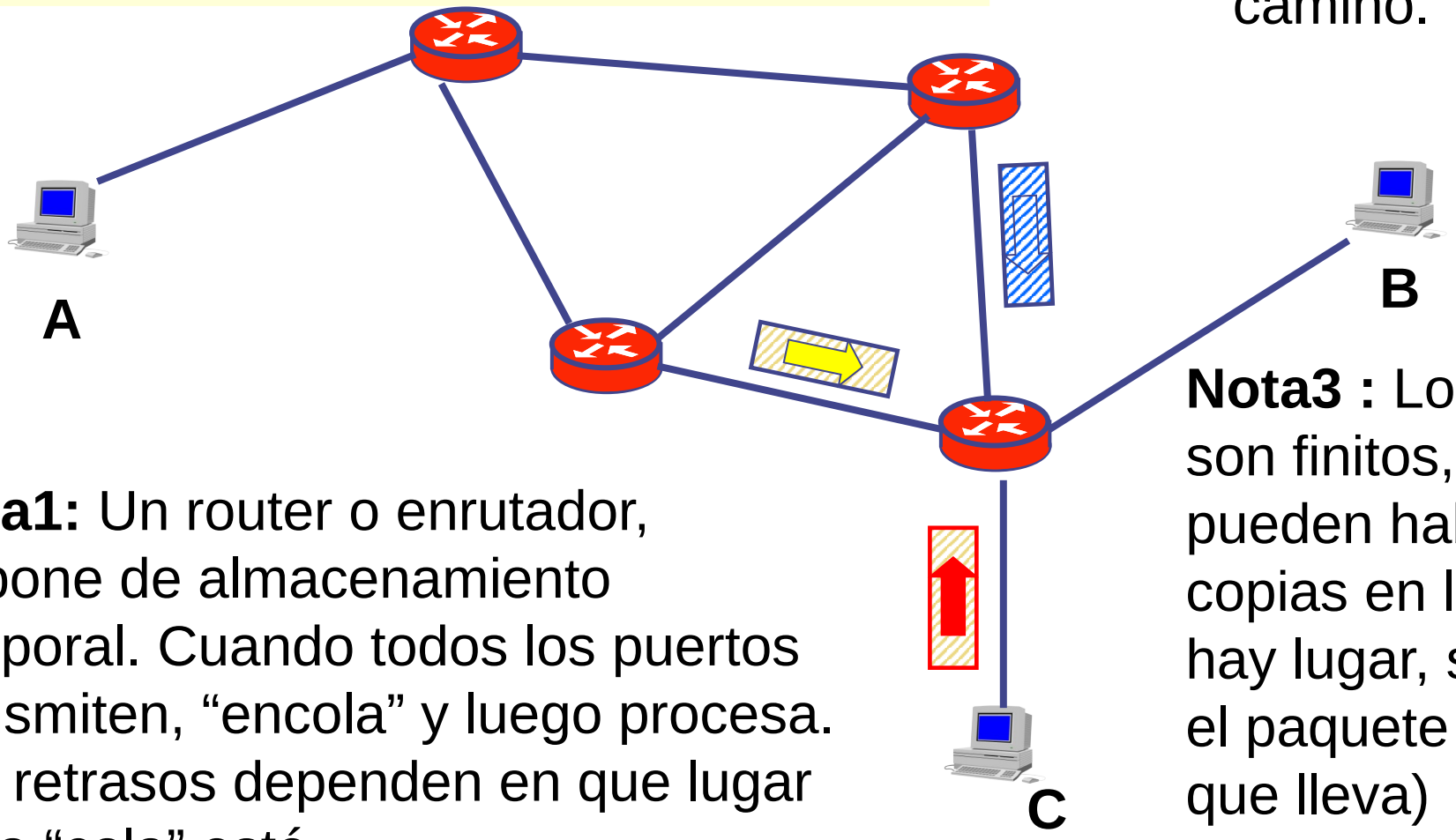
Caso contrario, aún cumpliendo $W_{RX} \leq (MAX\_SEQ+1)/2$, se requiere además que $W_{Tx} + W_{Rx} <= (MAX\_SEQ+1)$

Motivando Conceptos – Retardo Variable y Descartes

Recordar : Una **TPDU** (segmento) "viaja" en la carga útil de un paquete (**NPDU**).

Lo que suceda con los **paquetes**, afecta a los **segmentos** que lleva.

Nota2 : Tantas "colas" (buffers) como routers en el camino.



Nota1: Un router o enrutador, dispone de almacenamiento temporal. Cuando todos los puertos transmiten, "encola" y luego procesa. Los retrasos dependen en que lugar de la "cola" esté.

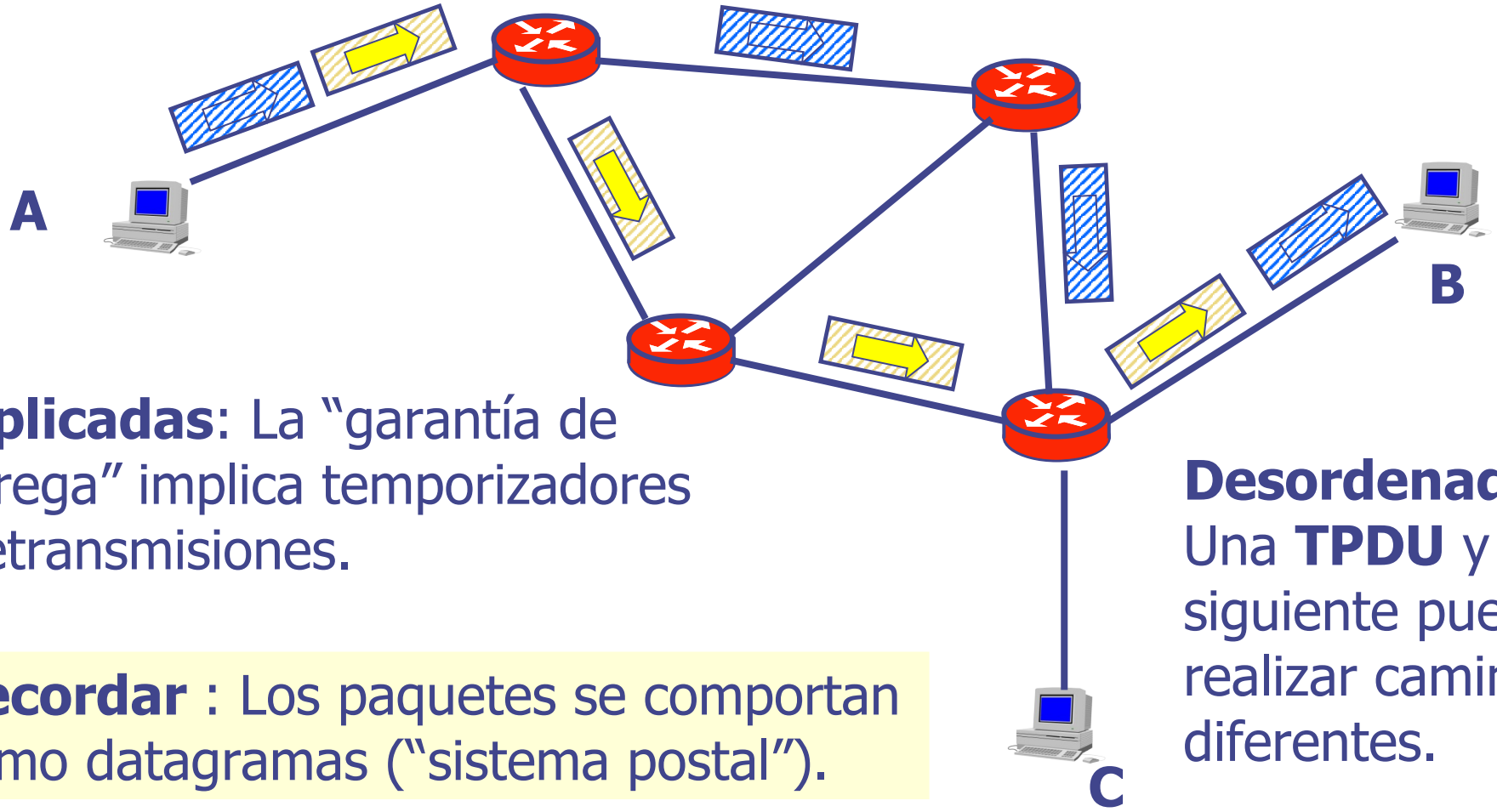
Nota3 : Los buffers son finitos, de no serlo pueden haber varias copias en la fila. Si no hay lugar, se descarta el paquete (y la **TPDU** que lleva)

Motivando Conceptos – Duplicadas y Desordenadas

Recordar : Una **TPDU** "viaja" en la carga útil de un paquete (**NPDU**).

Lo que suceda con los **paquetes**, afecta a los **segmentos** que lleva.

Duplicadas: Una **TPDU** y su reenvío pueden tomar diferentes caminos.



Duplicadas: La "garantía de entrega" implica temporizadores y retransmisiones.

Desordenadas: Una **TPDU** y la siguiente pueden realizar caminos diferentes.

Recordar : Los paquetes se comportan cómo datagramas ("sistema postal").

Ventanas Deslizantes y Números de Secuencia

- Cada retransmisión por expiración del temporizador genera un "**potencial**" duplicado.
- La subred podría tener duplicados que aparecen en el receptor cientos de **ms** luego del arribo del paquete original, podrían tener un número de secuencia correspondiente a la nueva ventana
  - Si los números de secuencia se "**reciclaron**"
- Para poder solucionar este problema, se precisa tener una **cota** superior del **tiempo de vida de un paquete** (TPDU) en la red, y **suficientes números de secuencia** para evitar que suceda.
- Debe haber suficientes números de secuencia y utilizarse a una velocidad suficientemente baja como para que al reutilizar un número, **todos los duplicados de TPDU anteriores (o sus ACK)** con dicho número hayan desaparecido.
- Se necesita **más números** de secuencia que si la red no pudiera **re-ordenar** los paquetes.

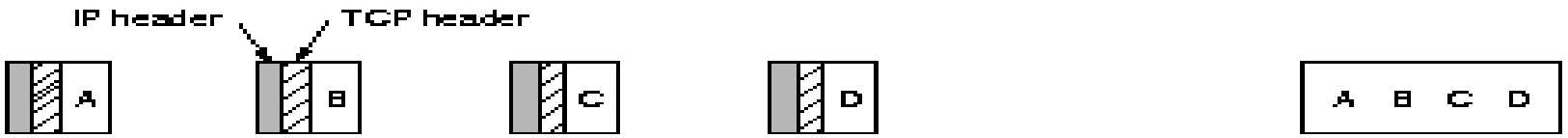
TCP – Connection Oriented Protocol

- **Orientado a Conexión:** previo al intercambio de datos, las entenidades de transporte deben acordar algunos parámetros. Estos se negocian en los primeros segmentos.
- **Objetivo:** Flujo confiable de bytes sobre una red no confiable
  - Debe funcionar sobre IP (no da garantías)
  - Diferentes tecnologías de red en el medio
  - Robusto frente a problemas de la red
- Recibe flujo de la capa superior y lo divide en bloques que envía en segmentos independientes (“uno en cada paquete IP”)
- El receptor los reensambla
- **¿Números de secuencia por TPDU (segmento)?**
  - **Tamaño fijo de segmentos.**
  - **Modo “mensaje” vs modo “flujo”**
  - **Forzaría a hacer relleno de segmentos o solo trabajar en “modo mensaje”.**
  - **Los ACK del receptor también deberían ser del mismo largo**

=> Los números de secuencia identifican los byte del flujo de datos de aplicación

TCP – Connection Oriented Protocol

- TCP **NO mantiene** las fronteras entre los bloques recibidos de la capa de aplicación, ni los recibidos de la red
- Ejemplo: Un mensaje (ABCD) es enviado en 4 paquetes de capa 3 separados, pero devueltos a la aplicación destino como un conjunto de bytes ABCD

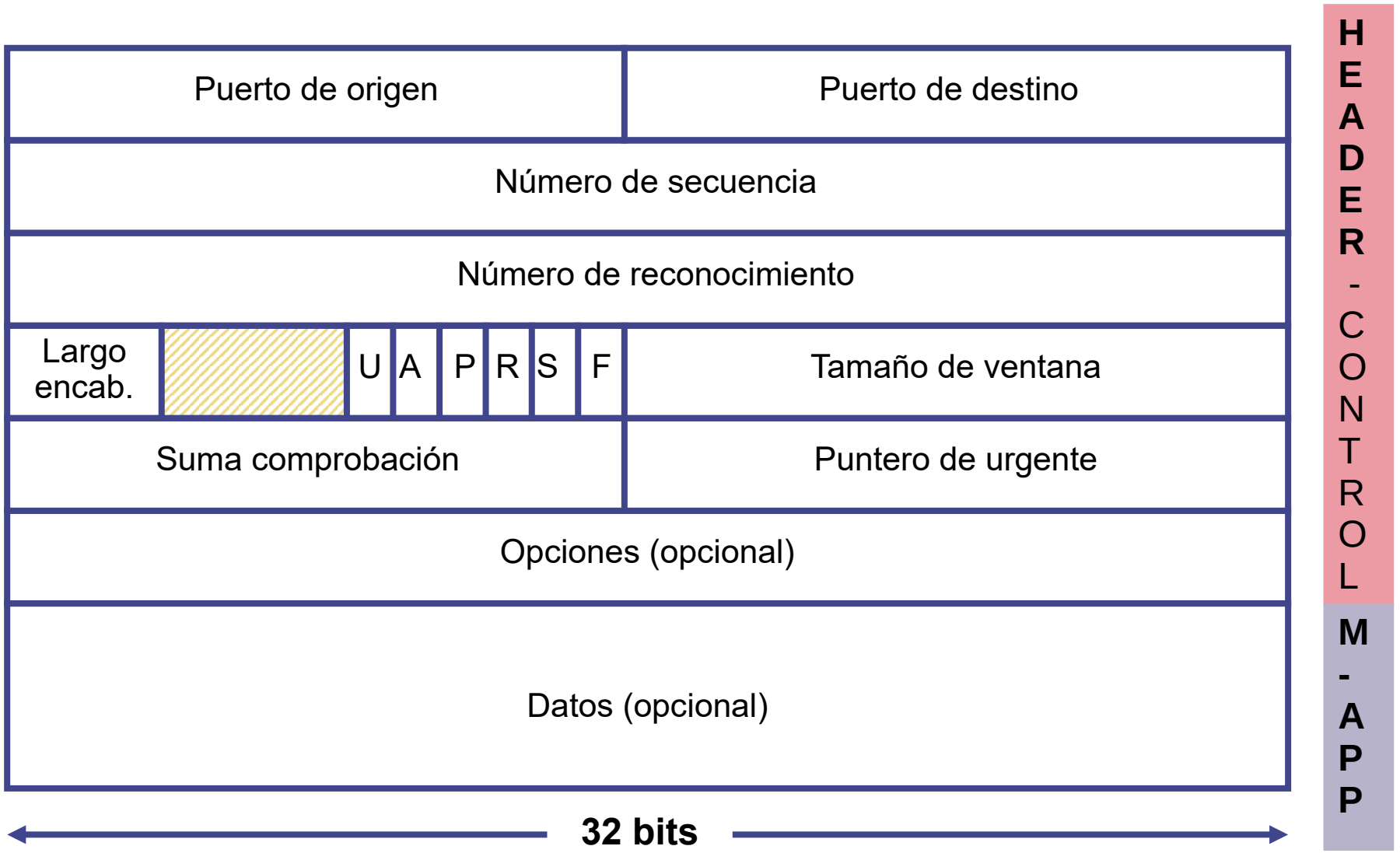


- Modo mensaje: la capa de transporte reconstruye el mensaje antes de entregarlo a la capa de aplicación.
- Modo flujo: la capa de transporte no se proecupa por las fronteras, reconstruye el flujo de bytes y lo entrega a la capa de aplicación en el orden que fueron entregados.

Protocolo TCP

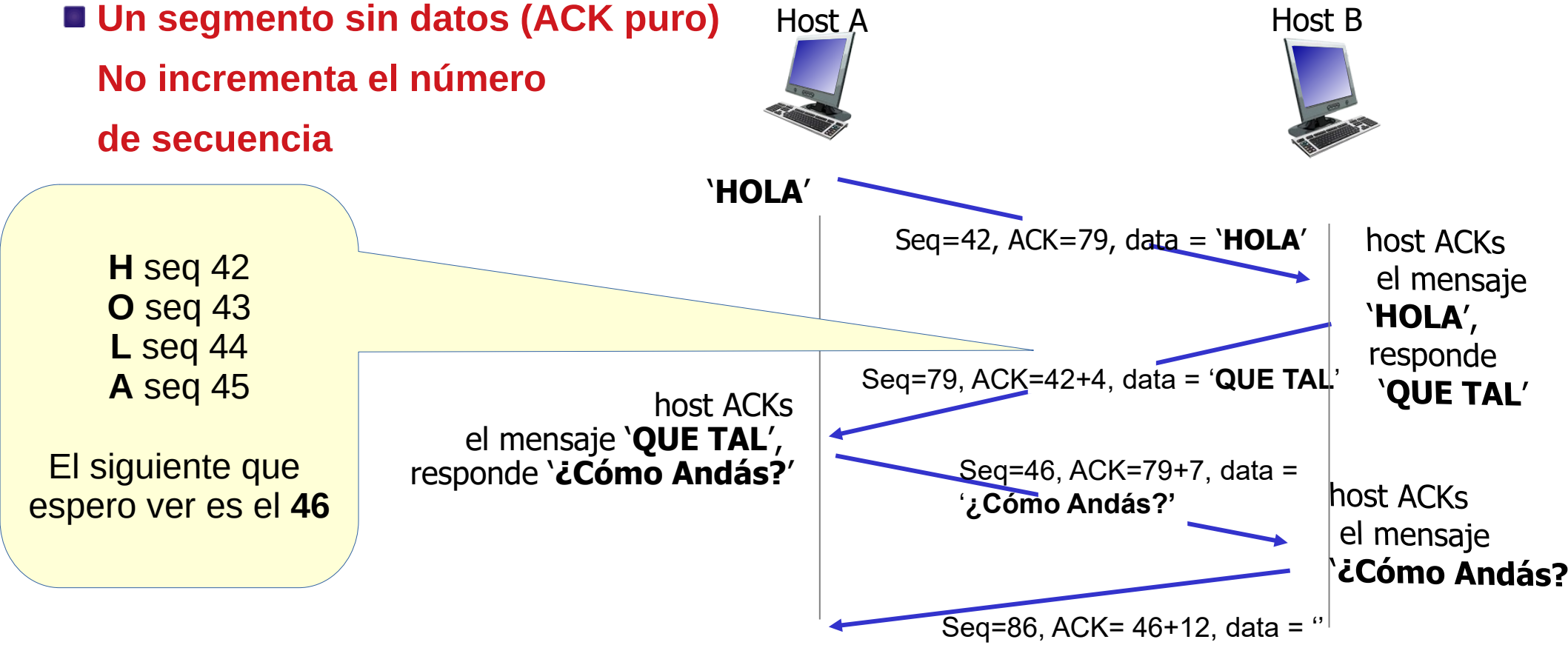
- Unidad de datos **(TPDU) = Segmento**
- Número de secuencia de 32 bits
- Se numeran los bytes, no los segmentos
- Reconocimientos acumulativos:
  - ACK = **x**, reconoce todos los bytes **x-1**, **x-2**, **x- 3**, …..
  - El siguiente byte que espera es **x**
- Encabezado de 20 bytes (+ opciones)
- Tamaño máximo del segmento
  - carga del paquete IP: máximo 64 Kbytes
  - TCP trata de evitar fragmentación en capas inferiores. Por ello se autolimita a la **MTU** (maximum transfer unit) de la red, típico 1500 bytes.
- Utiliza protocolo de ventanas deslizantes de tamaño de ventana variable

Formato del Segmento TCP



TCP – Reconocimientos y Números de Secuencia

- **Reconocimientos:**
  - Campo específico notificando que el segmento lleva un reconocimiento (**flag ACK**).
  - **Número de reconocimiento** (recordar acumulativo)
- **Número se secuencia**:
  - El número de secuencia del segmento corresponde al primer byte de datos de capa de aplicación.
  - **Un segmento sin datos (ACK puro) No incrementa el número de secuencia**



TCP – Flags o Banderas

- Un Flag es un capo de 1 bit, la bandera está levantada (**activa**) si el **bit = 1** o baja (**desactivada**) si el **bit = 0**.
- **U** - hay datos urgentes
- **A** - Campo de reconocimiento válido (en N.º de ACK)
- **P** - Push (se pide celeridad para enviar los datos a la capa de aplicación)
- **R** - Reset (cierre abrupto de conexión)
- **S** - Syn (sincronización inicial de números de secuencia durante el establecimiento de conexión)
- **F** - Fin (solicitud de fin de conexión)

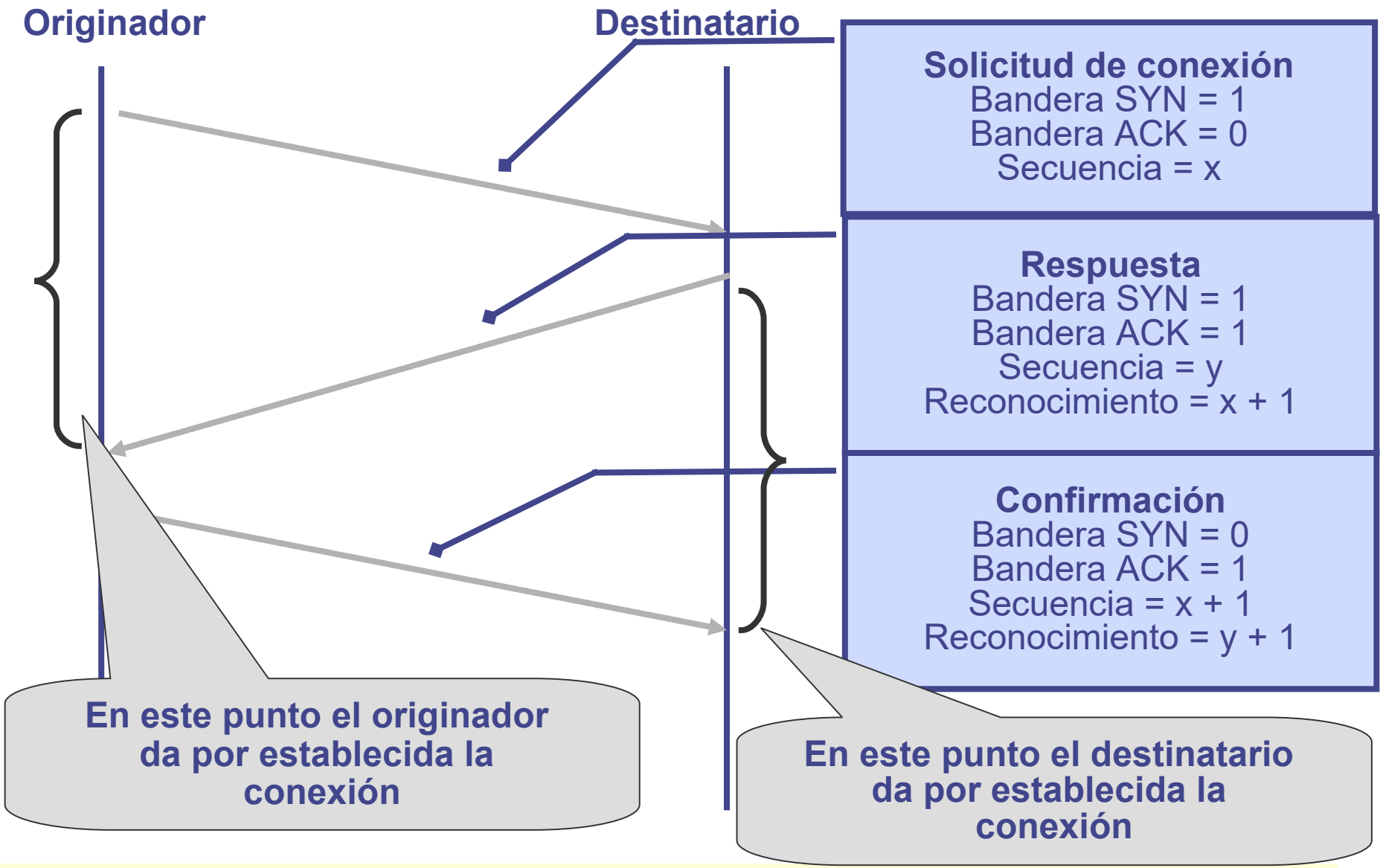
TCP – Opciones

- El campo de Opciones permite intercambiar datos no obligatorios
- Se han ido agregando nuevas opciones
  - Maximum Segment Size (**MSS**)
  - Escala de la ventana (**WSCALE**)
  - Asentimiento selectivo (SACK)
  - Marcas de tiempo (**Timestamp**)
  - Asentimiento negativo (NAK)
  - Otras

TCP – Opciones

- **MSS:** observamos el tamaño máximo de paquete que podemos enviar (dado por las capas inferiores), y descontamos los encabezados. Se envía en el primer segmento
  - Por ejemplo, en ethernet el máximo de trama (MTU) es 1500 bytes, le restamos 20 bytes de encabezado IPv4, restamos 20 bytes de encabezado TCP (sin opciones), el MSS es 1460
- NAK: avisar que no se recibió un determinado segmento (no se suele utilizar)
- Otras: veremos luego

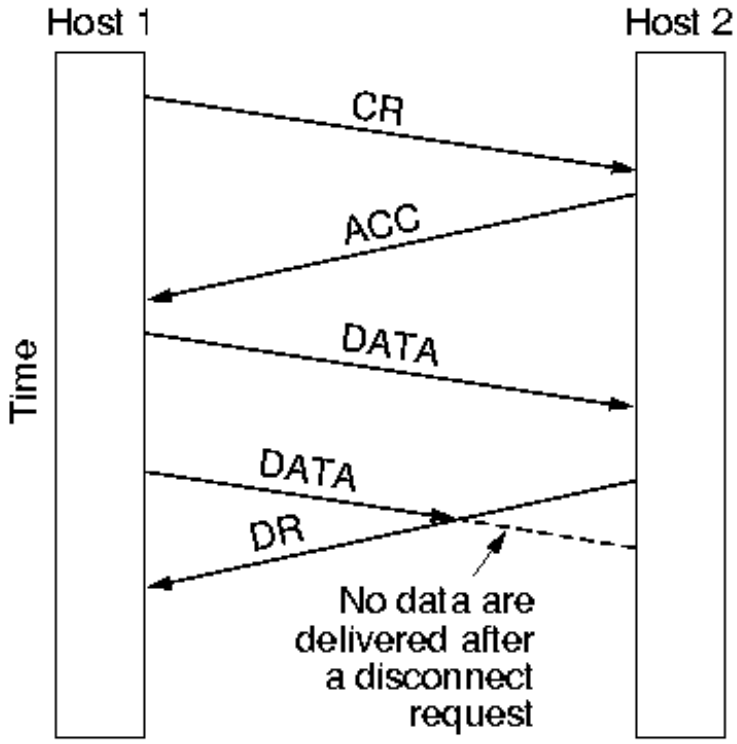
TCP – Inicio de Conexión



Se conoce como el establecimiento de tres vía o **three way handshake.**
Al finalizar, ambos extremos están de acuerdo en iniciar la conexión y sincronizan (conocen) sus números de secuencia inicial (y también el MSS).

TCP – Fin de Conexión

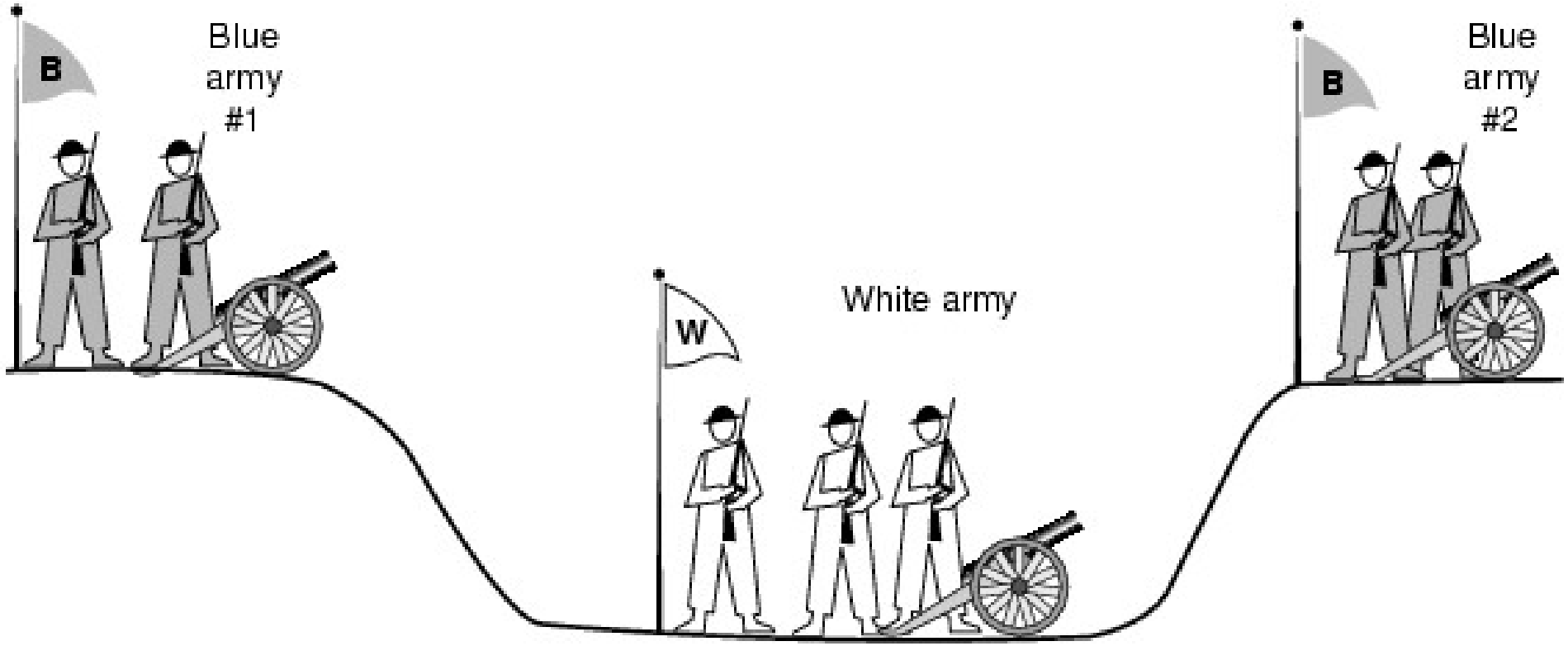
- Terminar una conexión:
  - Simétrica
    - Se intenta asegurar que no haya pérdida de datos, asegurando que ambos extremos estén de acuerdo en cerrar la conexión
    - se cierran separadamente ambos sentidos
    - dificultad “problema de los dos ejércitos”
  - Asimétrica
    - Uno de los extremos cierra la conexión unilateralmente
    - No garantiza que no haya pérdida de datos



Fin de Conexión - Problema de los dos ejércitos

Escenario: El ejército azul se encuentra destacado en las colinas, el blanco en el valle. El ejército azul es numéricamente superior al blanco, pero se encuentra divido en #1 y #2. El ejército blanco es superior numéricamente a ambas divisiones por separado.

Regla (simplificación): La batalla la gana el ejército que tenga más soldados

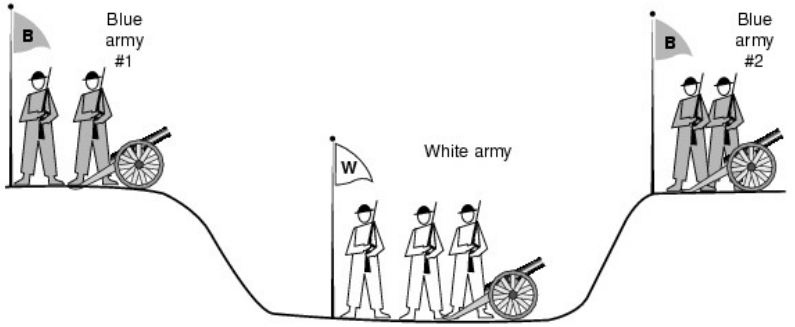


Problema: El ejército azul desea coordinar un ataque entre #1 y #2 para ganar la batalla, pero los mensajes entre ellos pasan por el valle, pudiendo ser capturados (vistos, eliminados, pero no alterados).

Fin de Conexión - Problema de los dos ejércitos

Blue (**#1** + **#2**) army **>** White army => **Blue win**

White army **>** Blue **#1** army, Blue **#2** army => **White win**



1. **#1** decide coordinar un ataque al amanecer, envía un mensaje a #2.

   ¿**#1** ataca al amanecer o debe esperar algo?

2. **#2** recibe el mensaje, y está de acuerdo, envía una confirmación.

   ¿**#2** ataca al amanecer o debe esperar algo?

3. **#1** recibe la confirmación de #2, envía una confirmación a #2.

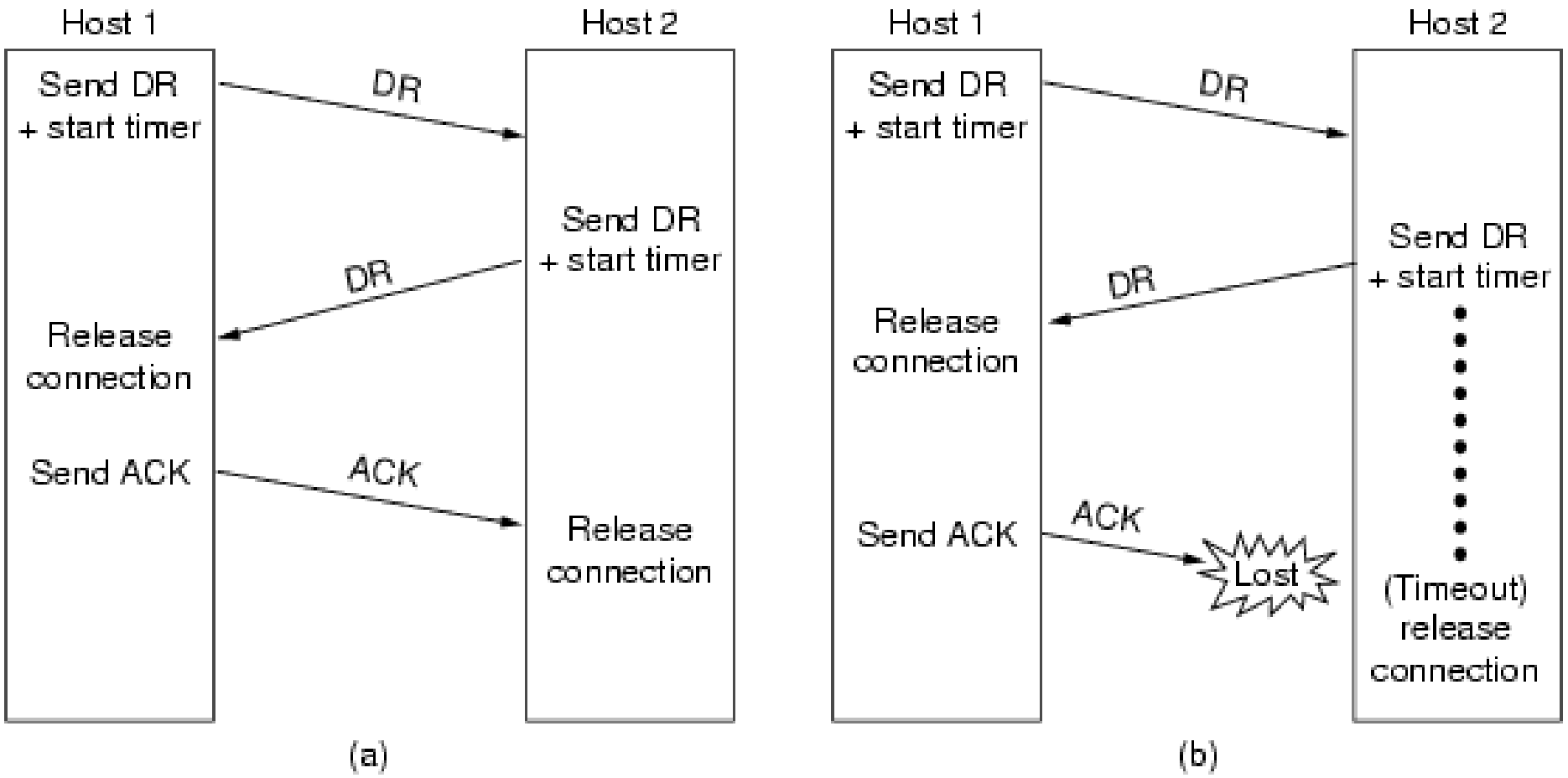
   ¿**#1** ataca al amanecer o debe esperar algo?

……

N. Siempre termino teniendo que confirmar el mensaje, no tiene solución.

Fin de Conexión – Pérdida de algunos segmentos

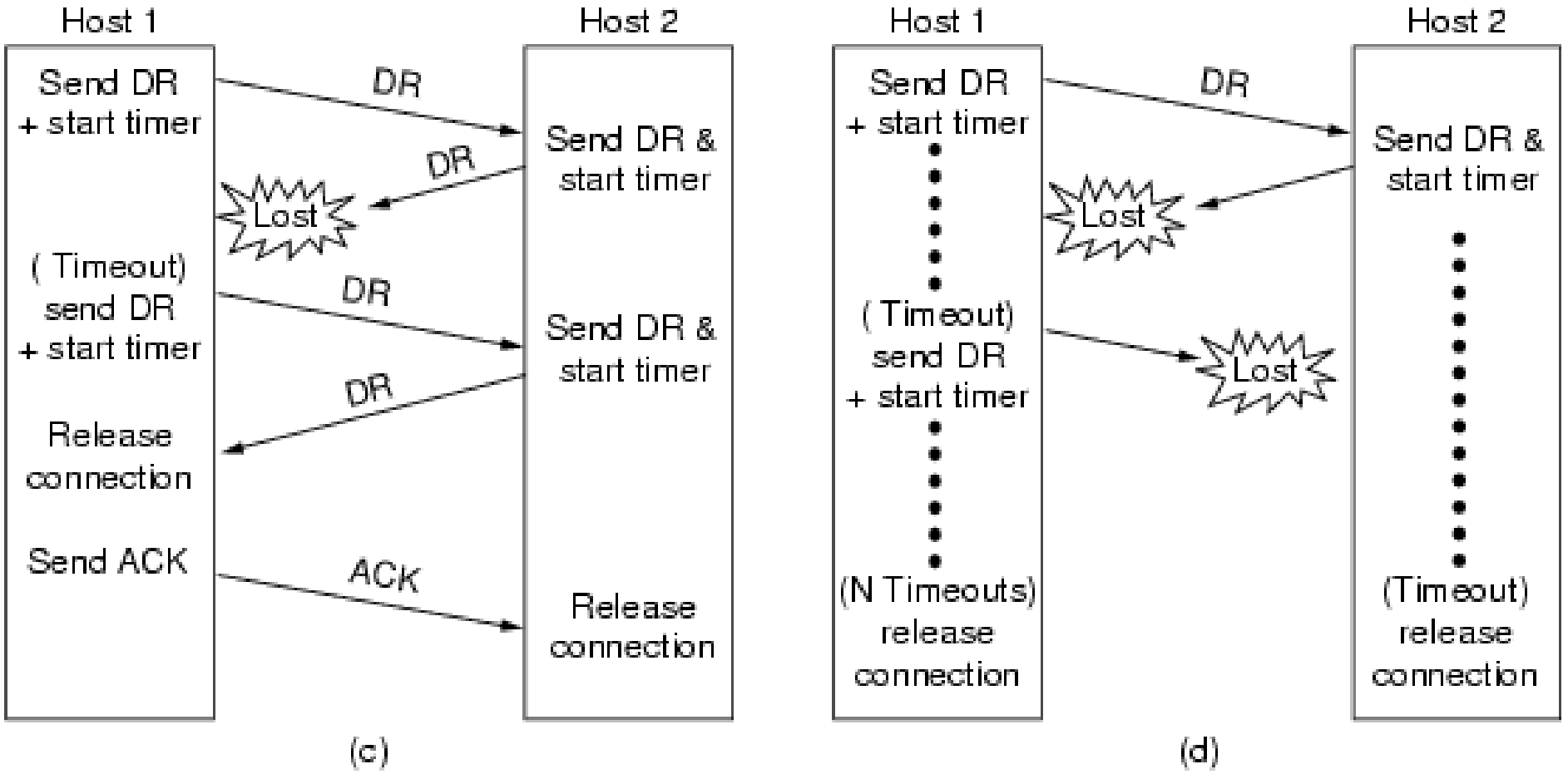
Ambos lados vieron el DR, formalmente no habría pérdida de información, alcanza esperar un tiempo T.



Variantes: retransmitir el DR cada T_R y hasta N_R veces, para vel el ACK.

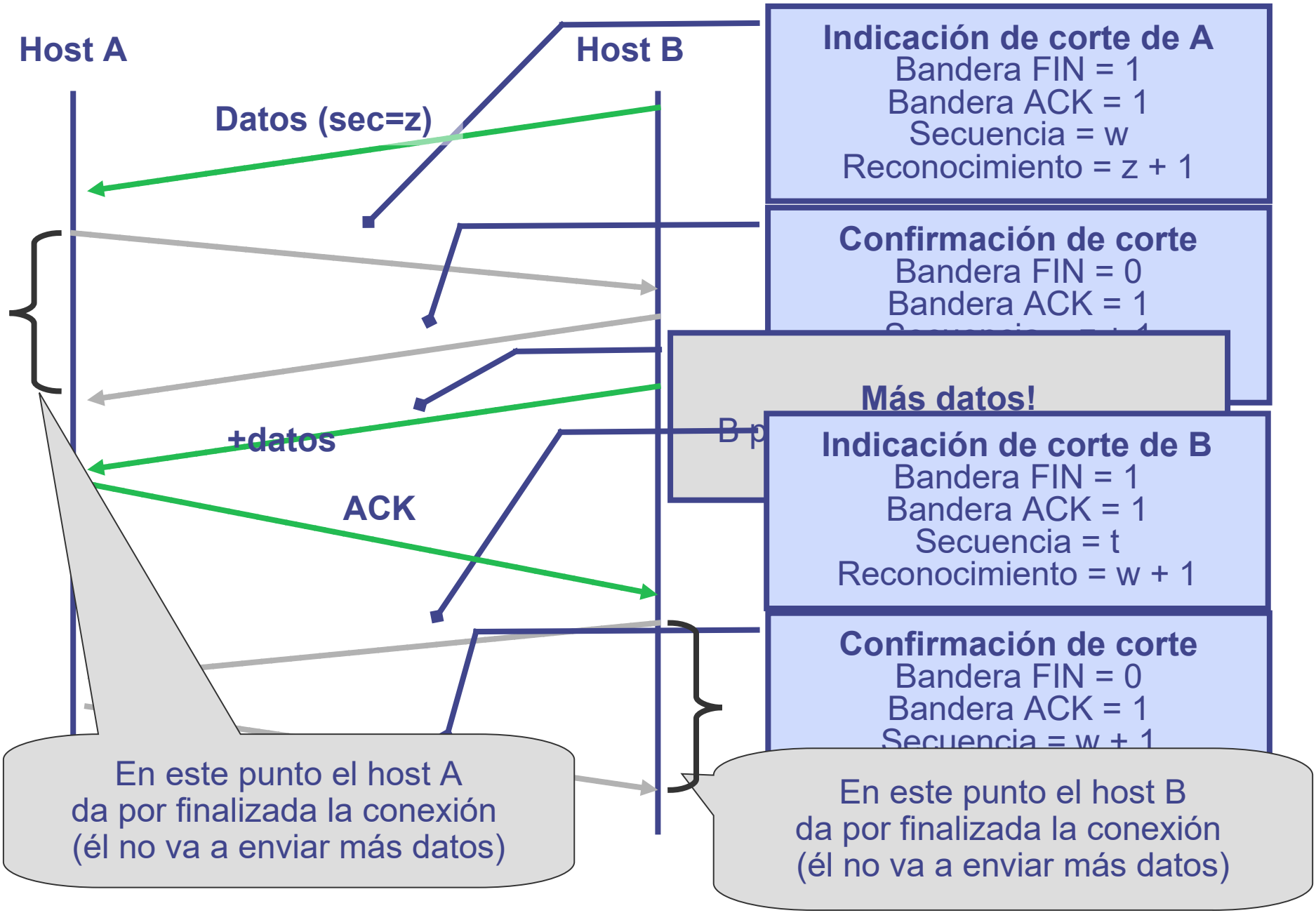
Fin de Conexión – Pérdida de algunos segmentos

EL caso (d) es un “caso de borde”, “evento raro” que N retransmisiones se pierdan, es muy probable que ocurra un corte de cables.

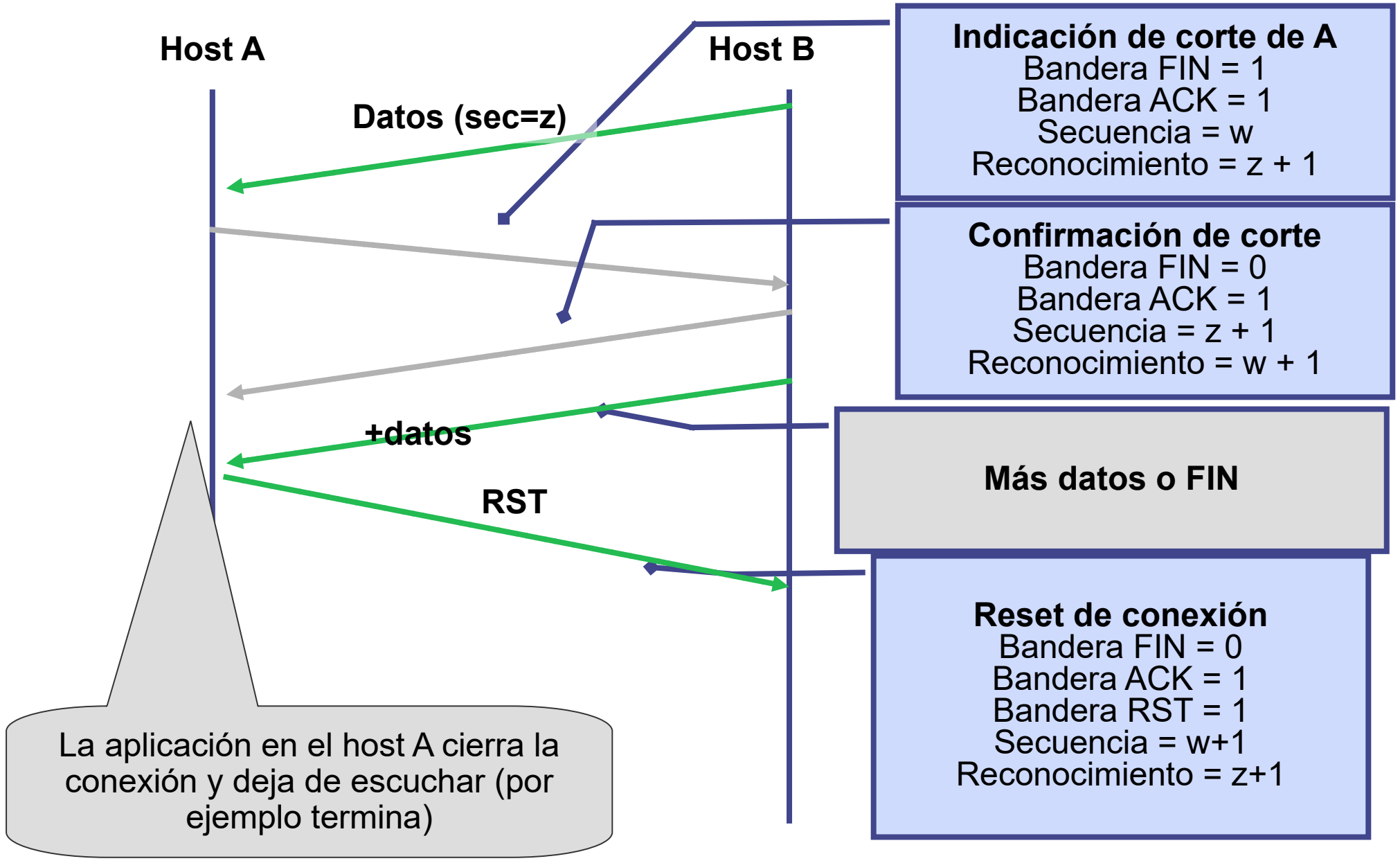


Con varios temporizadores (DR, ACK) y contadores de eventos (N Timeouts), es posible finalizar la conexión con garantía de ambas partes

TCP - Fin de Conexión Simétrica



TCP - Fin de Conexión Asimétrica

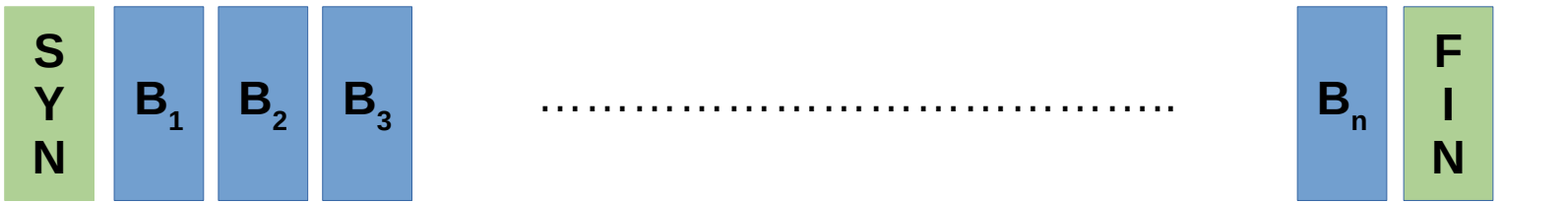


TCP - Fin de Conexión

- Simétrica:
  - Flag **FIN**
  - Cuatro vías: cada sentido finaliza y se reconoce la finalización.
  - Tres vías: caso particular del anterior, al enviar el ACK por el FIN del transmisor, también notifico mi intensión de finalizar la conexión (FIN).
- Asimétrica:
  - Flag **RST**
  - Los datos enviados luego de la confirmación de corte de B hacia A se pierden.
  - Los últimos datos enviados por B y reconocidos son hasta el byte “z”, cualquier otro dato posterior debe asumirse que se perdió (a menos que recibamos ACK)

TCP – Números de Secuencia y Reconocimiento

- En TCP se numeran los bytes, no los segmentos
  - Un segmento sin datos no incrementa el número de secuencia
- El establecimiento **(SYN)** y finalización de conexión **(FIN)** consumen 1 número de secuencia cada uno.
- En el reconocimiento se indica el próximo nº de secuencia esperado
- Los reconocimientos son acumulativos **ACK(x)** => reconozco hasta el **x-1** y el siguiente que “espero ver” es el **x**

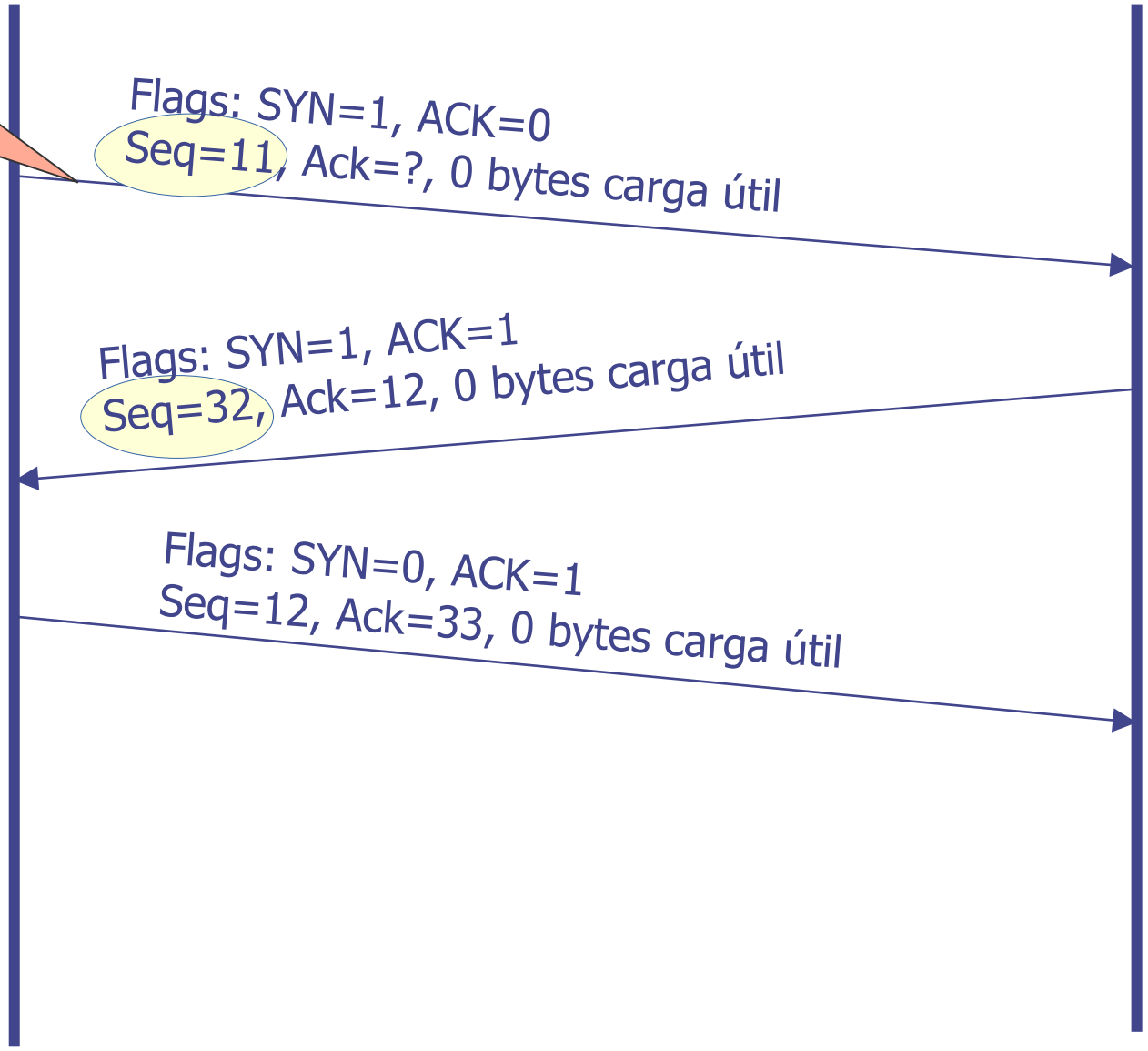


TCP – Ejemplo de Establecimiento de Conexión

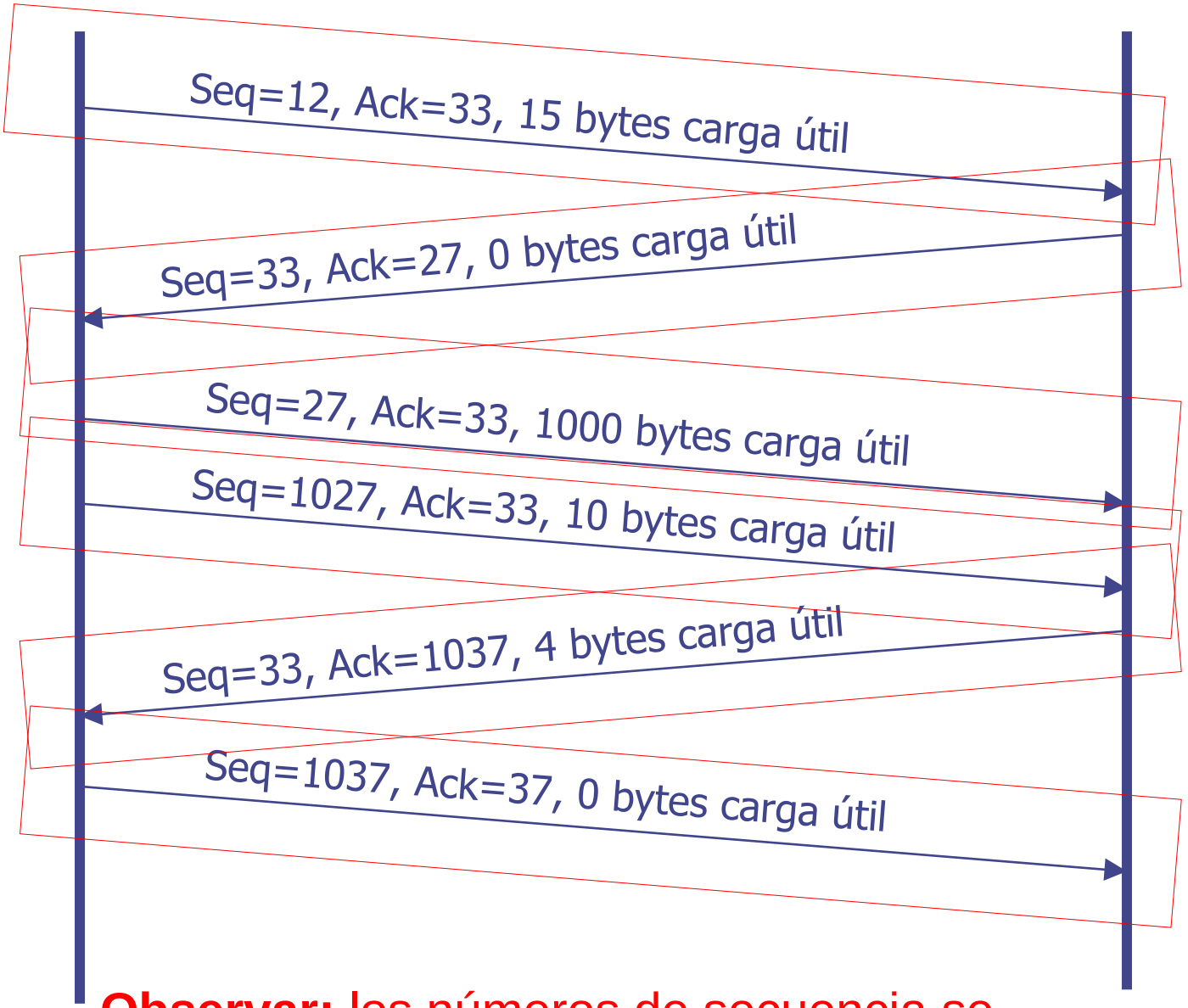
¿Cómo elijen los números de secuencia iniciales?

Si comienzan siempre desde cero la secuencia es predecible. Permite realizar ataques de Denegación de Servicio (DoS)

Los números iniciales en cada extremo de elijen de forma aleatoria e independiente del otro extremo

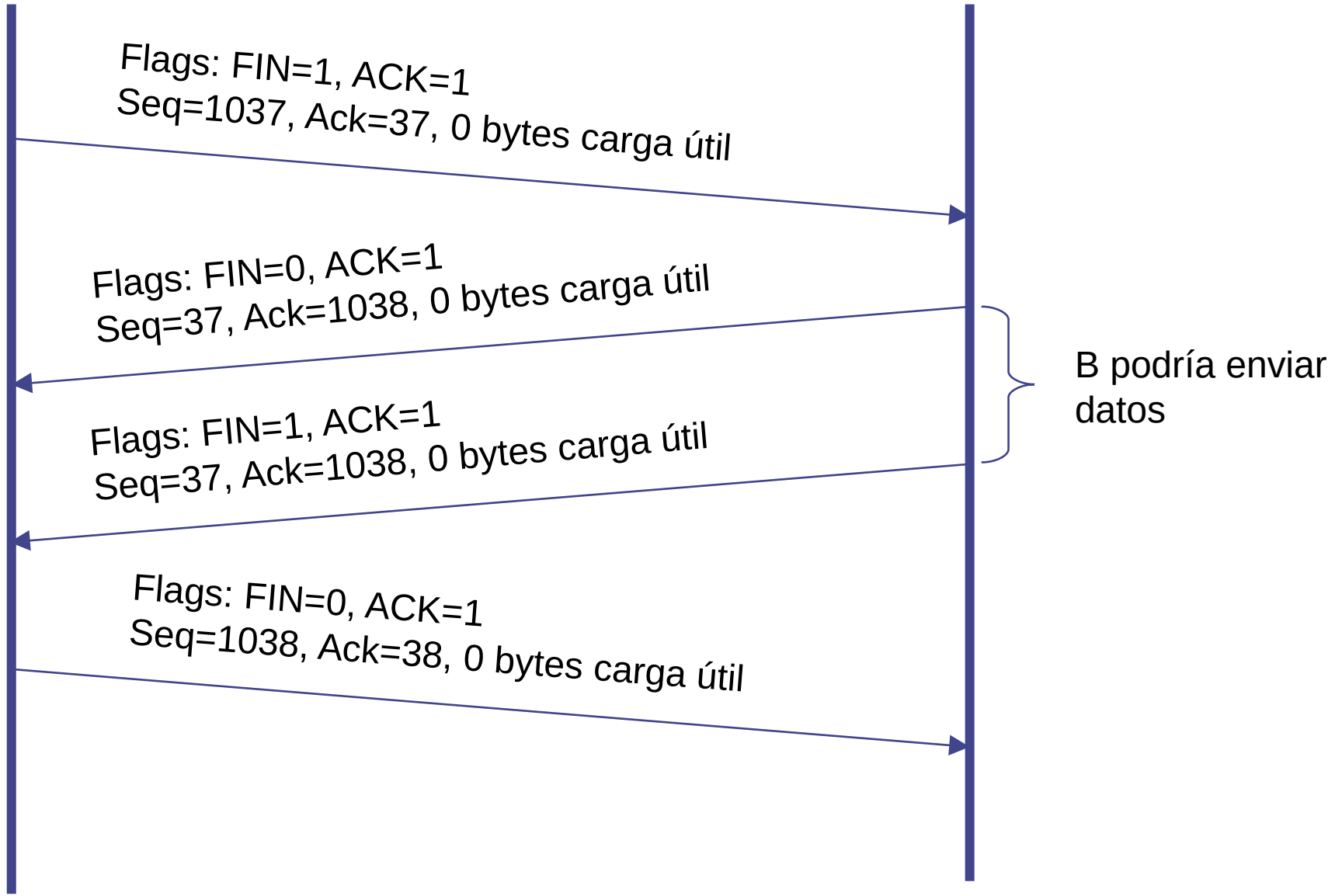


TCP – Ejemplo de Transmisión de Datos



Observar: los números de secuencia se incrementan de acuerdo a los bytes enviados

TCP – Ejemplo de Fin de Conexión Simétrico



TCP - Control de Flujo

- **El objetivo del control de flujo es adaptar el envío de datos a la capacidad del receptor.**

  Evitar que un transmisor “rápido” sature a un receptor “lento” o sobrecargado.

- Usualmente se basa en limitar la cantidad de datos que puede enviar el transmisor.
  - El receptor informa sobre el tamaño de la ventana (tamaño de buffer disponible) en cada segmento

    Campo "Tamaño de ventana” RXWIN en encabezado TCP
- El transmisor no puede tener en tránsito más que RWIN bytes a partir del último byte reconocido
  - **De esta manera, el receptor controla la cantidad máxima de datos que el transmisor puede enviarle en cada momento**
- Ejemplo: no enviar reconocimientos hasta tener buffers libres.
  Problema: fuerzo retransmisión por timeout

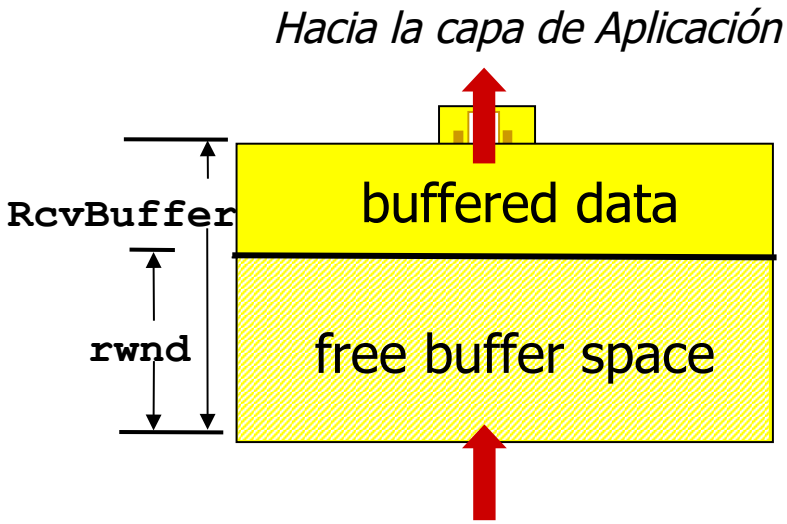
TCP - Buffers y Control de Flujo

- Las máquinas pueden manejar varias conexiones TCP simultáneas, por lo que se requiere manejo dinámico de buffers.
  - Cada conexión entre entidades de transporte tiene su respectiva ventana de transmisión $\mathbf{W_{Tx}}$, ventana de recepción $\mathbf{W_{Rx}}$ y los buffer correspondientes.
- El uso de buffers dinámicos hace conveniente separar el asentimiento (ACK) del segmento, de la señalización/notificación del tamaño de la ventana.
  - Notifico la **recepción** de los datos previos pero no tengo espacio para recibir más datos. Enviar **ACK** para evitar las retransmisiones y **RXWIN = 0**.
  - Cuando la aplicación lee del buffer y libera espacio, al tener espacio disponible, actualizo el tamaño de ventana.

TCP - Buffers y Control de Flujo

Recordar:

- Número de buffers libres del receptor = ventana de recepción (Windows Size) y no la cantidad de números de secuencia
- Es posible utilizar de un espacio de números de secuencia grande, e igual trabajar con equipos con poca capacidad de memoria



Receptor buffering

Analogía:
Estacionamiento del shopping que reporta los espacios libres.

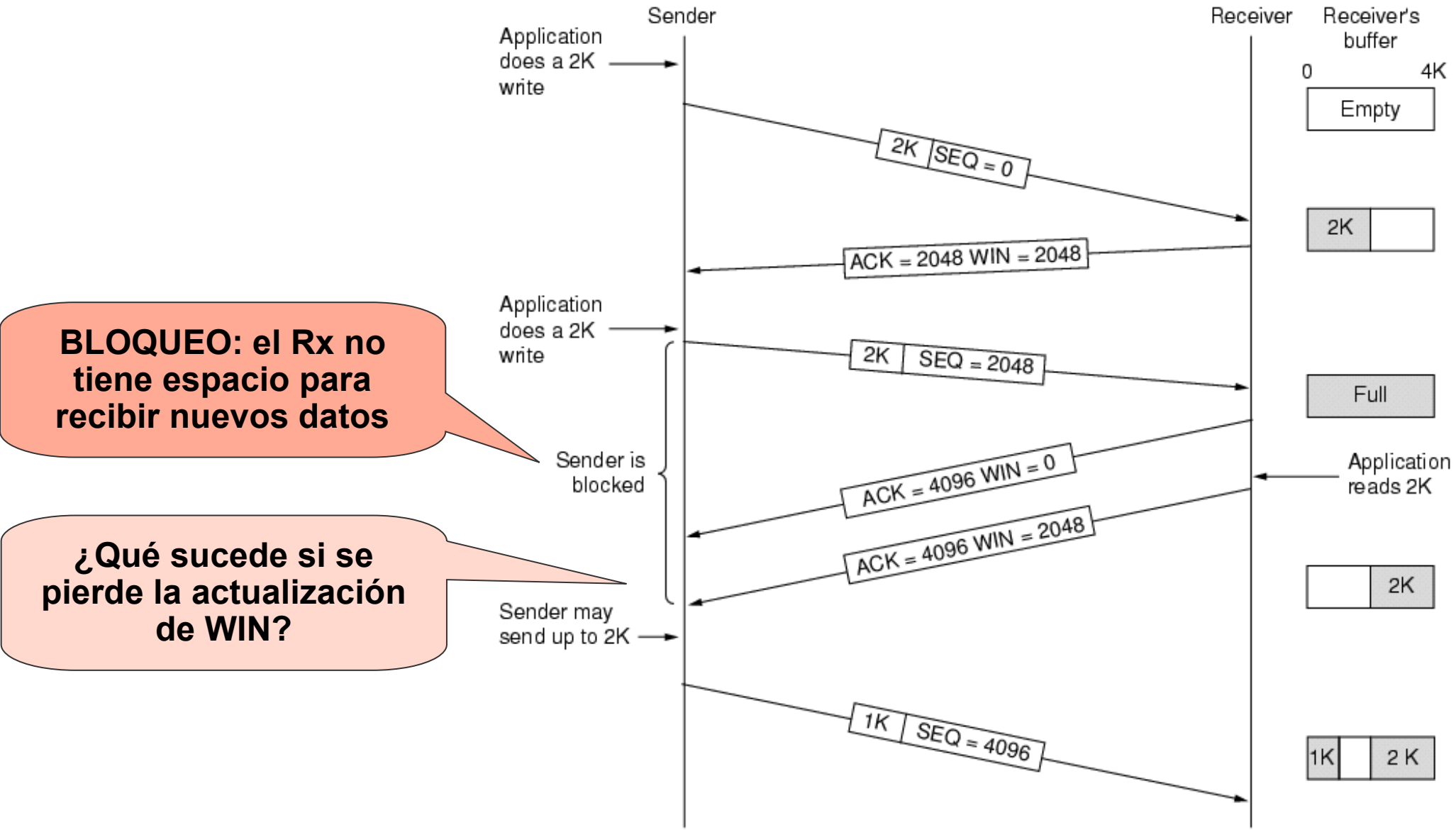
TCP – Gestión de Ventana del Receptor

- Podría destinarse buffer fijo en el receptor
  - La aplicación puede solicitarlo
- Muchos sistemas operativos adaptan automáticamente el tamaño del buffer
- Un buffer muy chico puede limitar la tasa de transmisión (¿cómo?)
- Si la aplicación no lee los datos suficientemente rápido, la ventana del receptor decrecerá
  - Puede llegar a “0” si se llena todo el buffer
- ¿Qué pasa si se pierde el mensaje de actualización de ventana luego de haber anunciado ventana “0”?
  - Posible deadlock
  - Solución: **pedido de actualización de ventana**

TCP – Anuncios de Ventana

- **Problema:** Se anuncia ventana 0, y siguiente anuncio con actualización de ventana, se pierde
  - Debe evitarse el bloqueo
  - Para ello, el emisor enviará un segmento que fuerce una respuesta del receptor (prueba para forzar re-anuncio de ventana):
    - Enviar un segmento con número de secuencia menor al actual (y sin datos)
    - O enviar el próximo byte (que podría ser descartado)

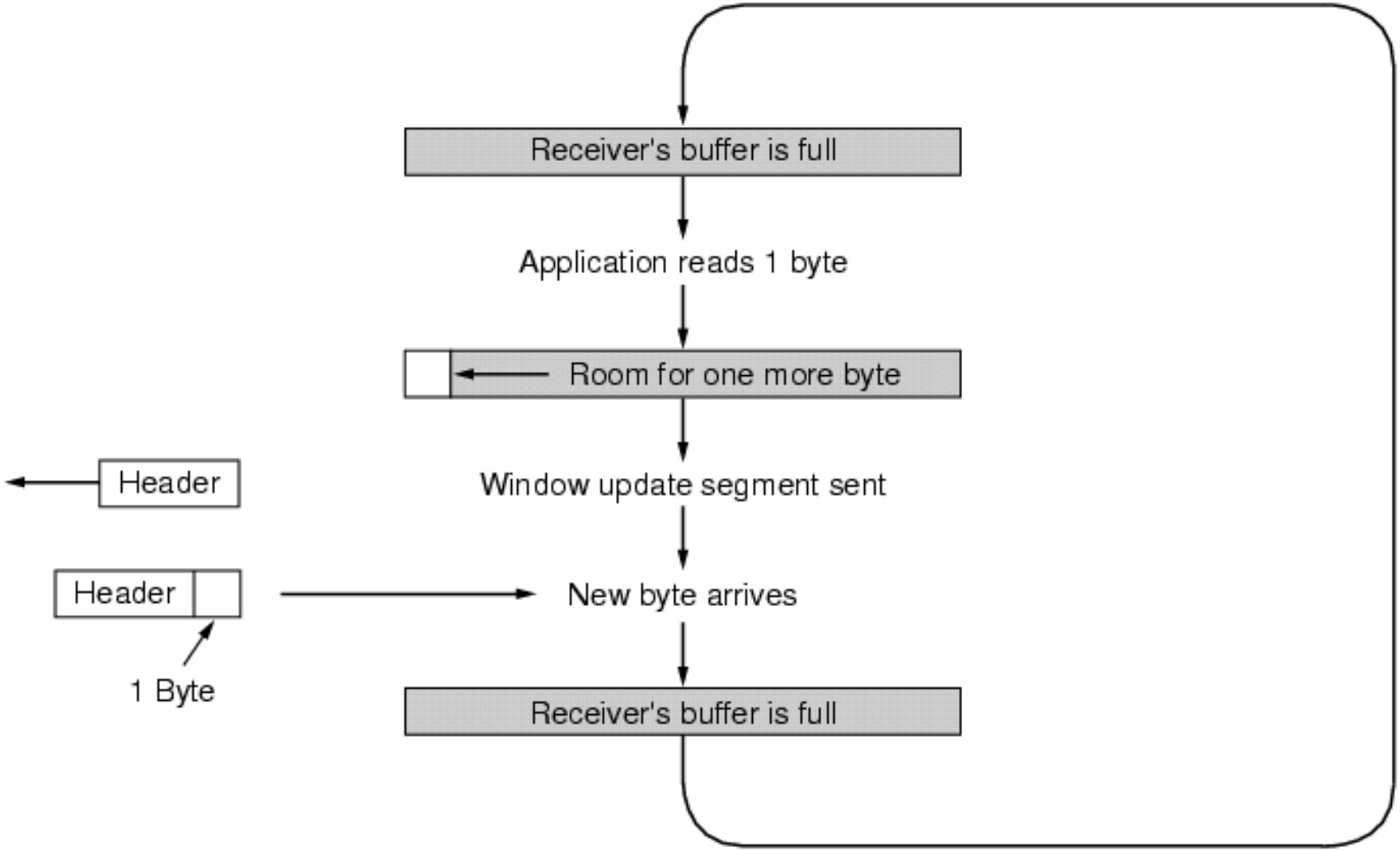
TCP – Gestión de Ventana



TCP – Problemas de Performance

- Objetivo: evitar muchos segmentos “chicos”
- Posibilidad de retardar el envío de reconocimiento (hasta 500 ms) para esperar a tener datos para transmitir (piggybacking)
- Algoritmo de Nagle (en transmisor)
  - esperar el ack del primer segmento y mientras bufferear (acumular los nuevos bytes).
  - se puede enviar también cuando se llena media ventana o el tamaño máximo del segmento
  - Malo en aplicaciones interactivas remotas (mouse)
- Síndrome de la ventana tonta (solución de Clark)
  - Aviso de ventana de 1 byte (o muy pequeña)
  - Clark: No avisar disponibilidad de ventana hasta tener libre segmento máximo o mitad del buffer

TCP – Sindrome de Ventana Tonta



TCP – Control de ancho de banda controlando Windows Size

W = tamaño de ventana (ej. 64 KB)
R = 80 Mbps o 10 MBps

ts =Tiempo de serialización (cuánto demoro en enviar W) = 6,4 ms

ts(ACK) = 20 bytes/ 10 MBps = 0,002 ms

RTT = 160 ms

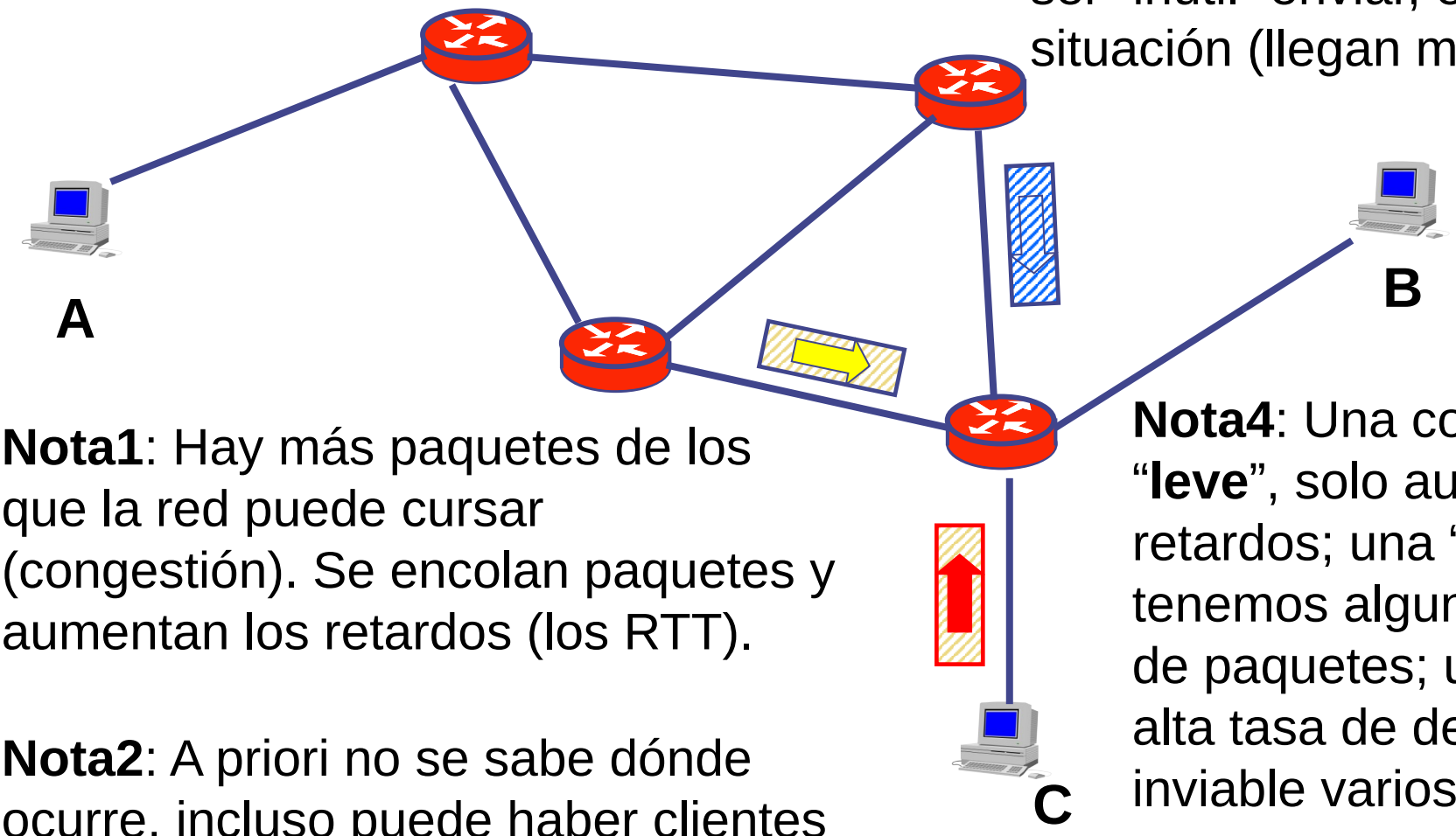
$$Beff = \frac{W}{ts + RTT + ts(ACK)} \sim \frac{W}{RTT} = 400\,kBps$$

Controlando W (formalmente ventana de transmisión < W) controlo el ancho de banda efectivo (máximo).

RTT no lo controlamos, depende dónde se encuentran los interlocutores y el estado (nivel de utilización) de la red.

Congestión – Motivación del Problema

Nota3: Cuando ocurre, si no disminuyo la tasa de envío puede ser "inútil" enviar, empeoramos la situación (llegan menos).



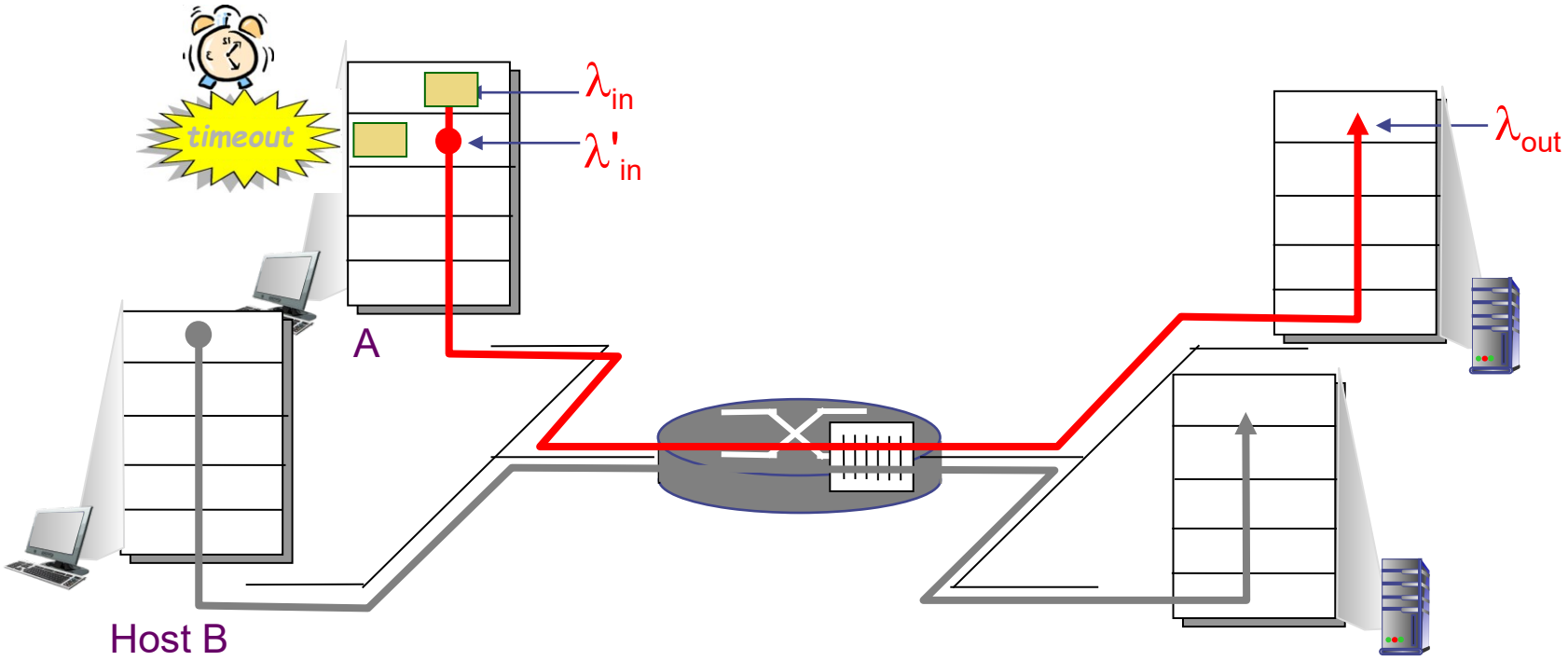
Nota1: Hay más paquetes de los que la red puede cursar (congestión). Se encolan paquetes y aumentan los retardos (los RTT).

Nota2: A priori no se sabe dónde ocurre, incluso puede haber clientes que no la experimenten.

Nota4: Una congestión "**leve**", solo aumentan los retardos; una "**moderada**", tenemos algunos descartes de paquetes; una "**severa**", alta tasa de descartes, hace inviable varios servicios.

CONGESTIÓN

Dos transmisiones en simultáneo



λ_{in}- bytes/segundo versiones “originales”

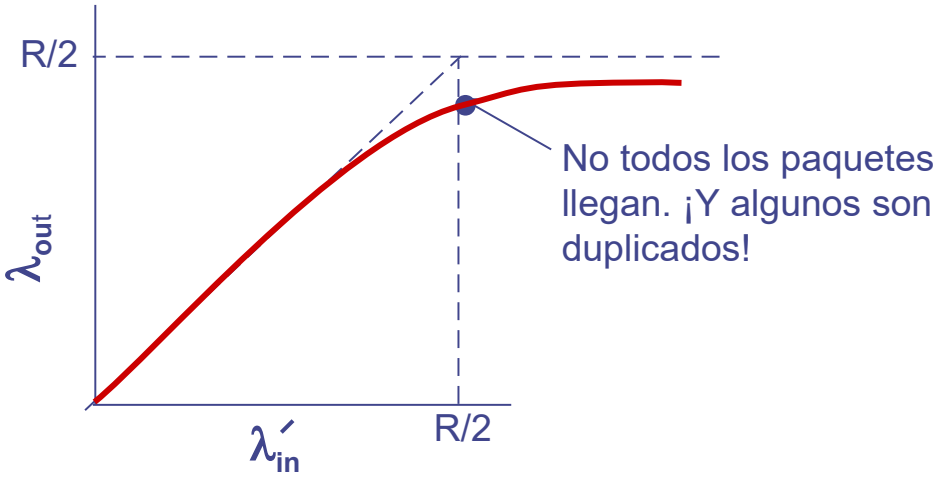
λ'_{in} - bytes/segundo versiones “originales” + retransmisiones por timeout

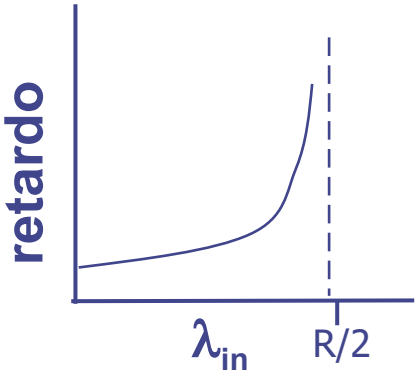
λ_{out} – bytes/segundo tasa “efectiva” de aplicación $\lambda_{out} \leq \lambda'_{in}$

R – Capacidad del Link (bps)

CONGESTIÓN

Dos equipos (transparencia anterior) transmitiendo al mismo tiempo, cada uno puede utiliza como máximo R/2.





- El retardo “promedio” aumenta fuertemente al acercarme a la tasa R/2 (la “fila” ya tiene paquetes esperando).
- Se presentan pérdidas (asumiendo buffers finitos).
- Se retransmiten segmentos innecesariamente (por timeout debido a la demora por avanzar en la “fila”).
- Mantener o incrementar la tasa de envío $\boldsymbol{\lambda'_{in}}$ genera mas copias en camino, en la “fila” del router, llegan menos a destino (disminuye $\boldsymbol{\lambda_{out}}$)
- En capa de transporte con mecanismo de ventanas deslizantes, solo se permite enviar lo autorizado por la capacidad de $\boldsymbol{W_{TX}}$ (limita el $\boldsymbol{\lambda_{in}}$)

CONGESTIÓN

- La congestión ocurre cuando se sobrepasa la capacidad de algún (al menos uno) elemento en la red
  - Típicamente algún enlace o la CPU de algún enrutador
- Los efectos observados incluyen **aumento de retardo** (RTT) y **pérdidas** (retransmisiones)
  - Si no se controla, lleva a un uso muy ineficiente de la red
- Puede participar la **capa de transporte**, la **capa de red** (todo el trayecto entre el origen y el destino), **o ambas**.
- **Concepto de Multiplexado Estadístico de Tráfico**
- Para detectar la congestión requerimos “alguna” **realimentación** sobre el estado de carga de la red
- **Objetivo:** enviar exactamente la cantidad de datos que la red puede transmitir
- **Dificultad:** no conocemos la capacidad instantánea disponible

Control de Congestión

- No enviar más datos que los que la red puede aceptar
  - **Complementario al control de flujo**, donde solo intervienen la capa de transporte de los extremos.
- Idea: “Entubado”
  - Si tenemos disponible un ancho de banda B, y el retardo de ida y vuelta es 2T, queremos enviar:

    W = B*2T (por ahora asumamos que W < 64 KB)

    ¿Por qué?
- **Problema:** ¿Cómo conocer o estimar B?

TCP – Control de CONGESTIÓN

- **Hipótesis:** las pérdidas de paquetes son por congestión (los enlaces son de buena calidad, tasa de errores del medio despreciable)
  - Problema en enlaces inalámbricos con muchas pérdidas
- El **transmisor** utiliza una nueva ventana llamada **ventana de congestión**, que actualiza dinámicamente de acuerdo a las condiciones de la red
  - Mantiene también un valor "Umbral"
- El transmisor no permite que haya en tránsito más bytes que los que indica la ventana de congestión.
- ¿Porqué una ventana de congestión?

  No deja de ser una variable que estimamos, y en “forma de ventana” porque ya sabemos que ajustando la ventana del tranmisor, ajustamos la tasa de envío.

TCP – Control de CONGESTIÓN

- Llamamos **cwnd** al tamaño de la ventana de congestión
- Se define también un tamaño de umbral (**ssthresh**)
- Iremos variando el tamaño cwnd
- Se identifican 2 períodos bien definidos:
  - **Slow start:** al comienzo, al no conocer la capacidad disponible, se comienza con una ventana chica pero se hace crecer rápidamente
  - **Congestion Avoidance:** al (posiblemente) acercarnos a la congestión, hacemos crecer despacio la ventana
- Ante pérdidas, disminuimos drásticamente la tasa de transmisión. **¿Cómo?** Disminuyendo la ventana.

TCP – Ventana de Congestión (cwnd) – Slow Start

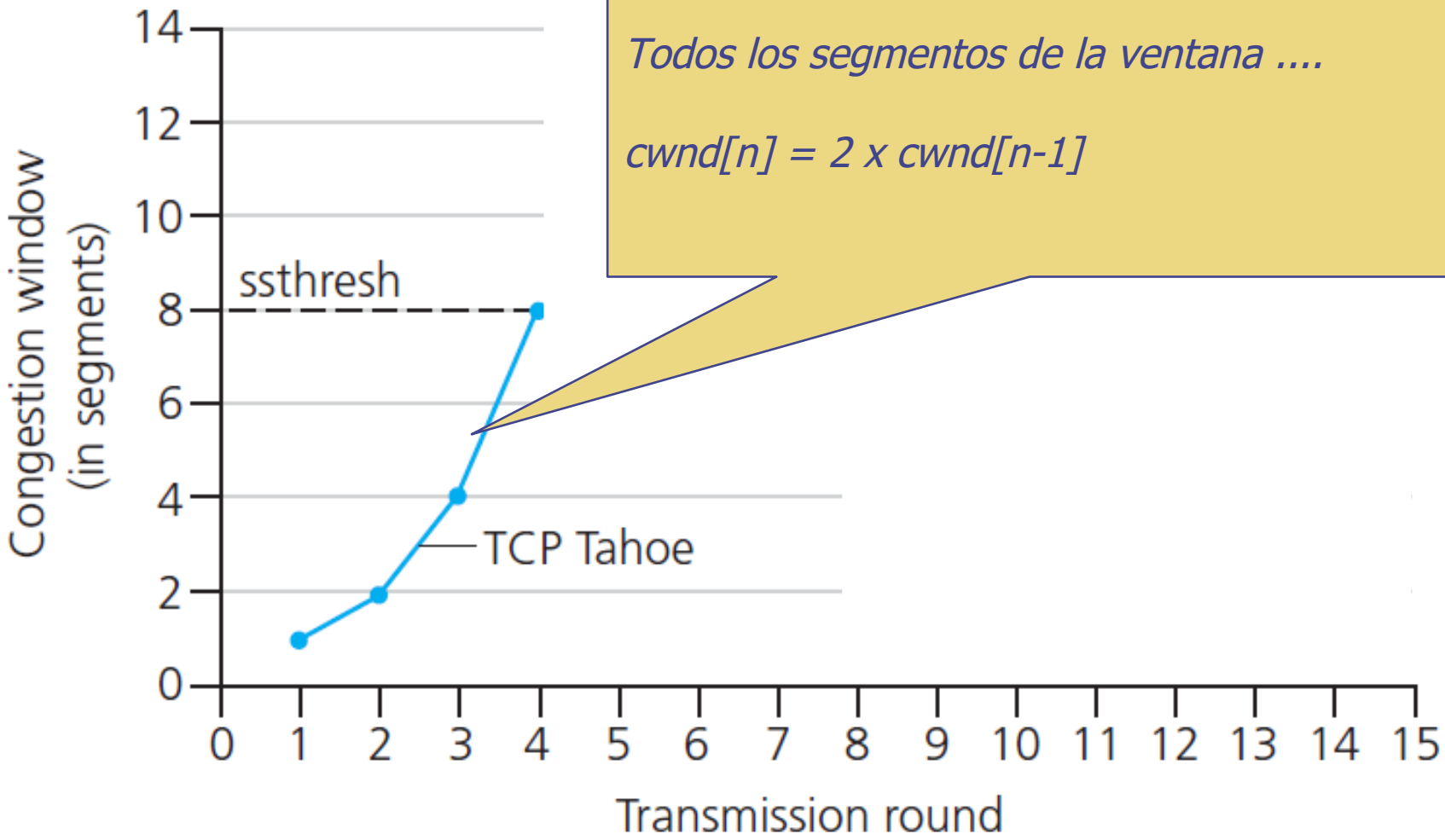
- Valor inicial de **cwnd = 1 MSS**
  - Versiones modernas suelen comenzar con un tamaño un poco más grande
- Por cada byte reconocido, agrega 1 byte a la ventana (a cwnd)
  - Al reconocerse toda la ventana enviada, esta se duplica
- Si tenemos pérdidas, bajamos el valor de cwnd
- Comportamiento se mantiene hasta que cwnd alcanza al valor del umbral (**ssthresh**)

TCP – Ventana de Congestión (cwnd) – Slow Start

cwnd[n] = cwnd[n-1] + [Número de bytes ACK]

Todos los segmentos de la ventana ....

cwnd[n] = 2 x cwnd[n-1]



TCP – Ventana de Congestión – Congestion Avoidnance

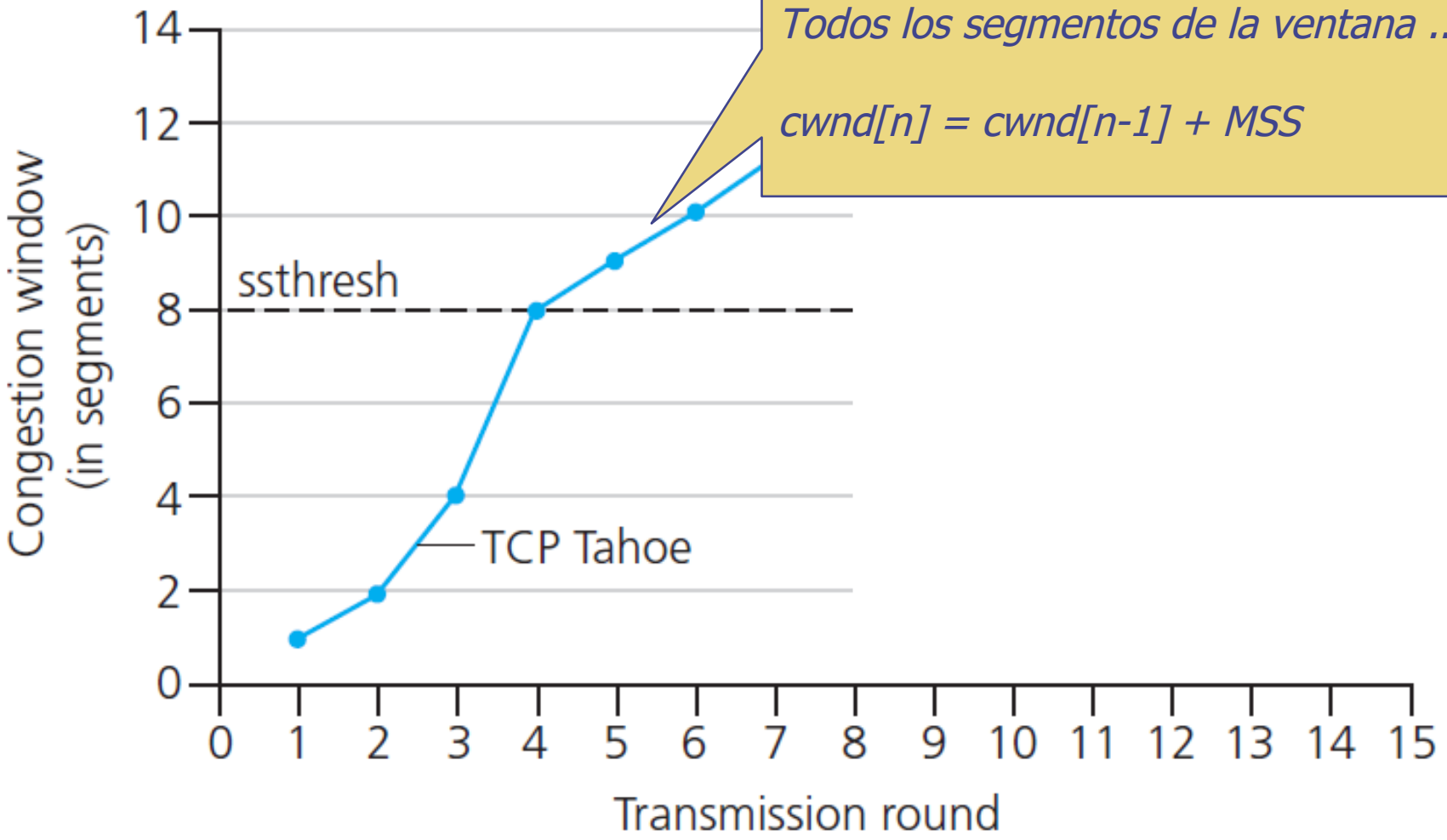
- A partir del umbral se asume que podemos estar más cerca del punto donde ocurre congestión
  - Se incrementa cwnd de forma lineal
  - Se aumenta en 1 MSS por cada ventana completa reconocida
- ¿Si no hay pérdidas, podríamos crecer indefinidamente?
  - Usualmente nos termina limitando la ventana del receptor, también se debe seguir respetando el control de flujo.

TCP – Ventana de Congestión – Congestion Avoidnance

cwnd[n] = cwnd[n-1] + (MSS/cwnd[n-1]) x [Número de bytes ACK]

Todos los segmentos de la ventana ....

cwnd[n] = cwnd[n-1] + MSS



TCP – Comportamiento ante pérdidas

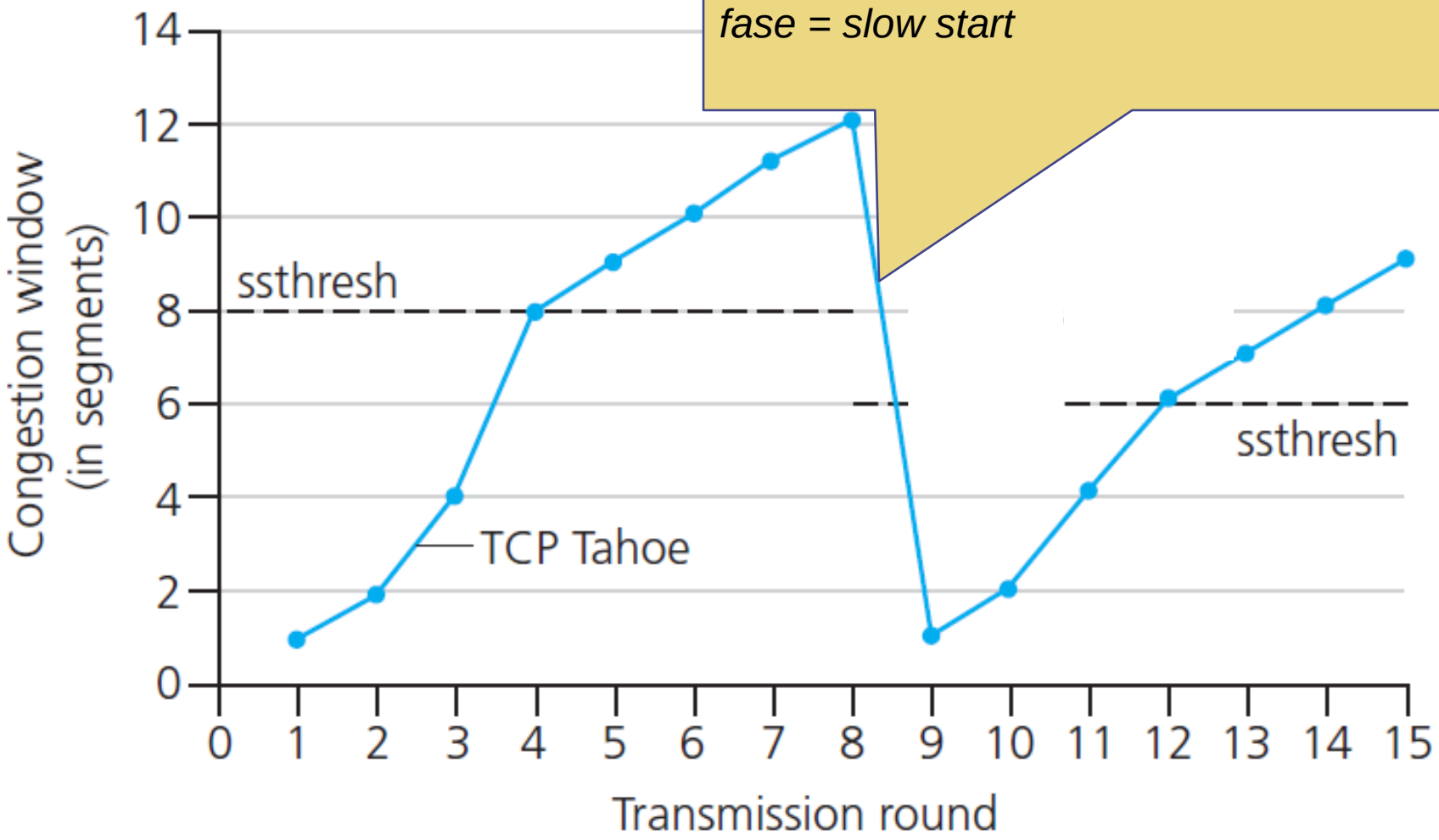
- **Timeout = Congestión**
  - Disminuir la tasa de transmisión
  - Disminuir la cwnd

- **Timeout:**
  - fijar umbral (ssthresh) a la mitad del valor de la ventana cwnd actual (NO a la mitad del umbral actual)
  - Disminuir la ventana (cwnd) a 1 MSS
  - Retomar en fase slow start

TCP – Ventana de Congestión – Timeout

cwnd[n] = 1 MSS
ssthresh[n] = cnwd[n-1] /2 (“la última ventana”)

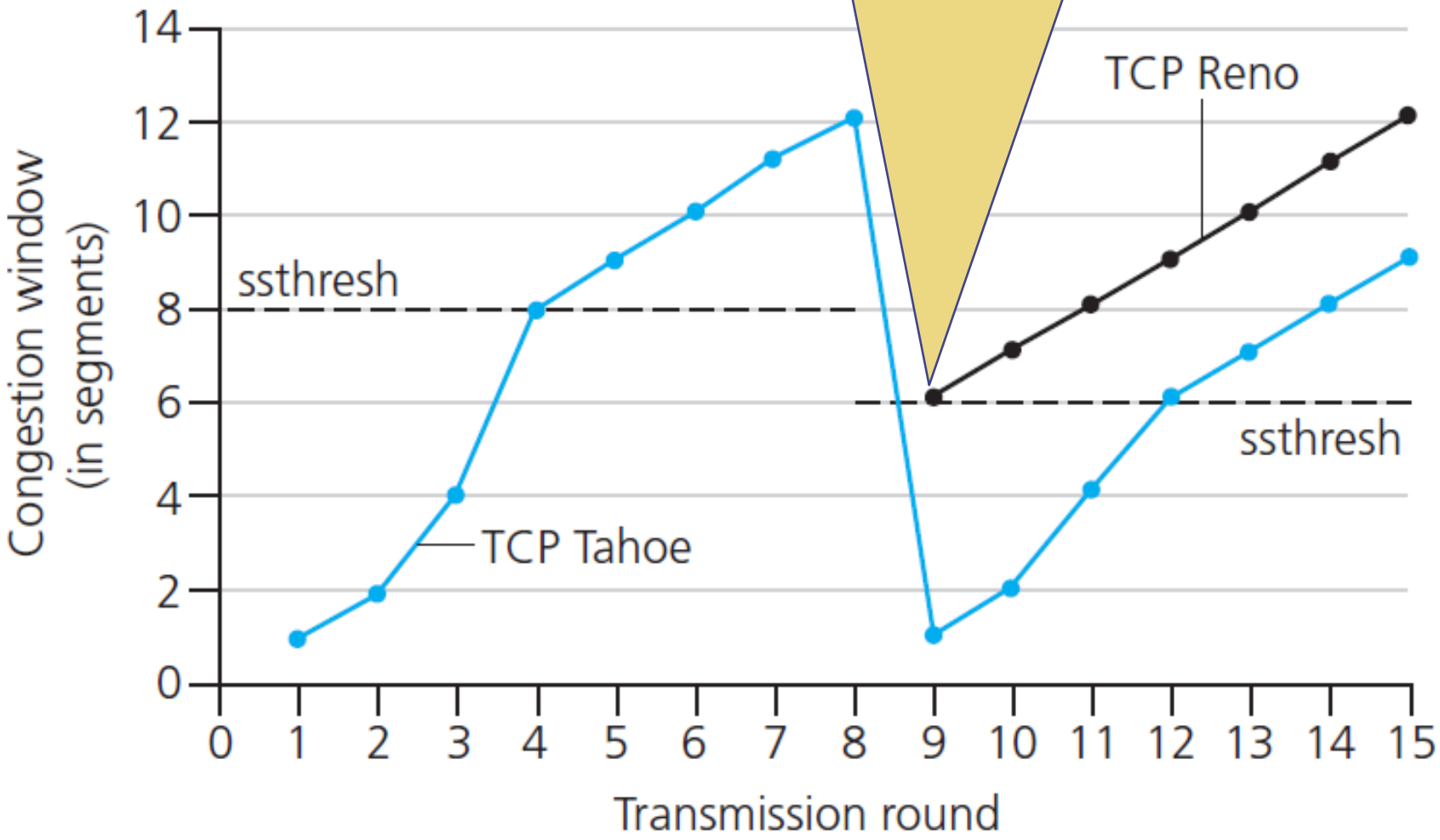
fase = slow start



TCP – Comportamiento ante pérdidas

ssthresh[n] = cnwd[n-1] /2 (“la última ventana”)
cwnd[n] = ssthresh[n]

fase = congestion avoidance



TCP – Comportamiento ante pérdidas

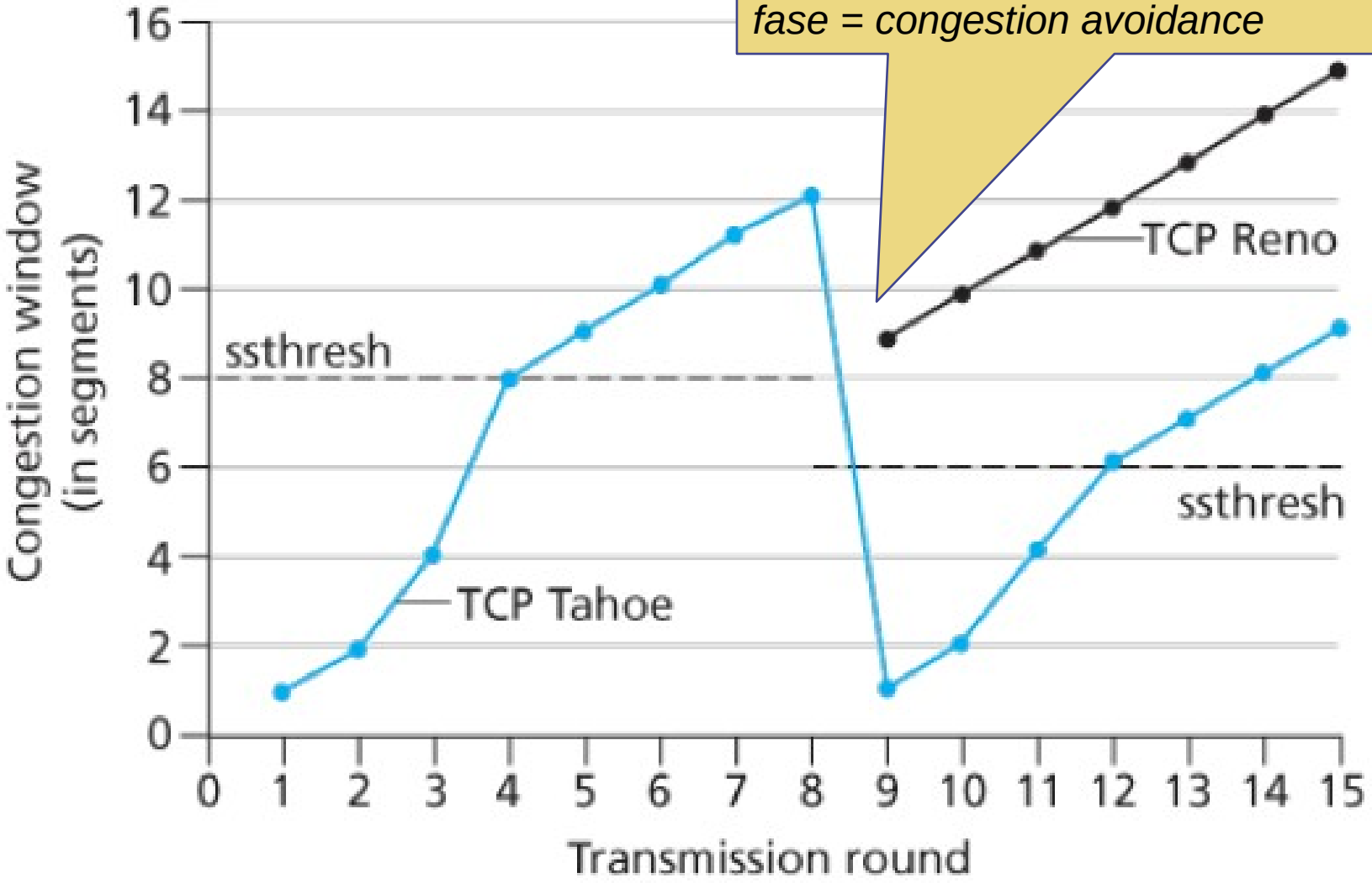
- No todos los escenarios de pérdidas son similares.
- **Reconocimientos repetidos:**
  - Recordemos el caso en que envío 4 segmentos y se perdió solo el primero. Se repite el ACK(Z) tres veces. **Fast Recovery**
  - **Se siguen recibiendo segmentos**, así que se asume que la congestión no es tan importante.

- Un poco de historia y actualidad:
  - **Hay variantes de TCP** de acuerdo a como reaccionar ante pérdidas: Tahoe (original), Reno, Vegas, CUBIC, ECN, etc.
  - Modificar el comportamiento frente a la congestión es un tema debatido y resistido por la IETF, sobre todo por **fairness** en la distribución del ancho de banda.
  - Google implementó **QUIC**, originalmente para intercambio interno en sus centros de datos, actualmente el Browser Chrome lo puede utilizar.

TCP – Comportamiento 3 ACK duplicados

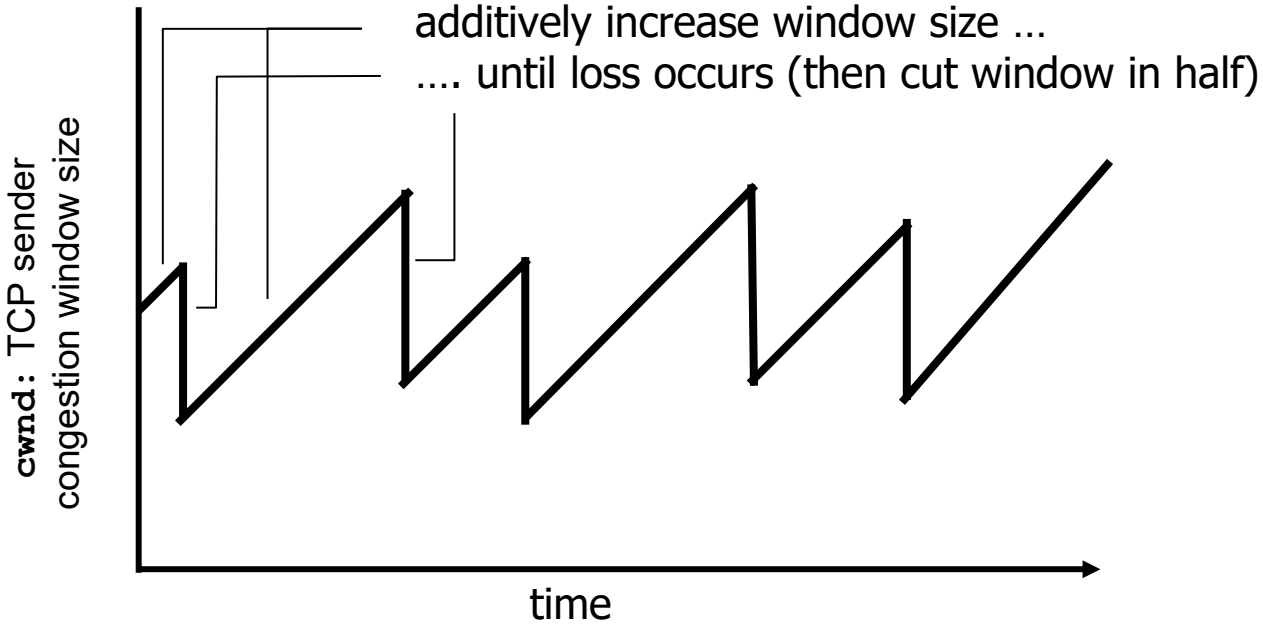
ssthresh[n] = cnwd[n-1] /2 ("la última ventana")
cwnd[n] = ssthresh[n] + 3 ***MSS***
(los 3 segmentos posteriores en el buffer)

Fast Recovery = retransmite "el faltante"
fase = congestion avoidance



TCP – Control de Congestión y diente de sierra

- El mecanismo de prueba para encontrar la congestión, es básicamente aumentar el envío hasta detectar una pérdida.
- Si consideramos la fase de "**congestion avoidance**":
  - Incremento aditivo de cwnd en 1 MSS cada RTT
  - Decremento multiplicativo, cwnd a la mitad de cwnd antes de la pérdida.
  - Produce un efecto de **diente de sierra**, que puede "**sincronizarse**" entre los transmisores de segmentos (paquetes) que compartan el punto de congestión.
  - **TCP Reno**



TCP – Ventana de Congestión

- El gráfico es una versión simplificada
  - "**Todos los segmentos**" de forma instantánea (idealización cuando el tiempo de serialización es mucho menor que el valor de RTT).
  - Observar que el comportamiento esta basado en RTT, que varía dinámicamente.
  - **¿Cómo elijo el valor de los temporizadores para los timeout?**

TCP – Gestión de Temporizadores

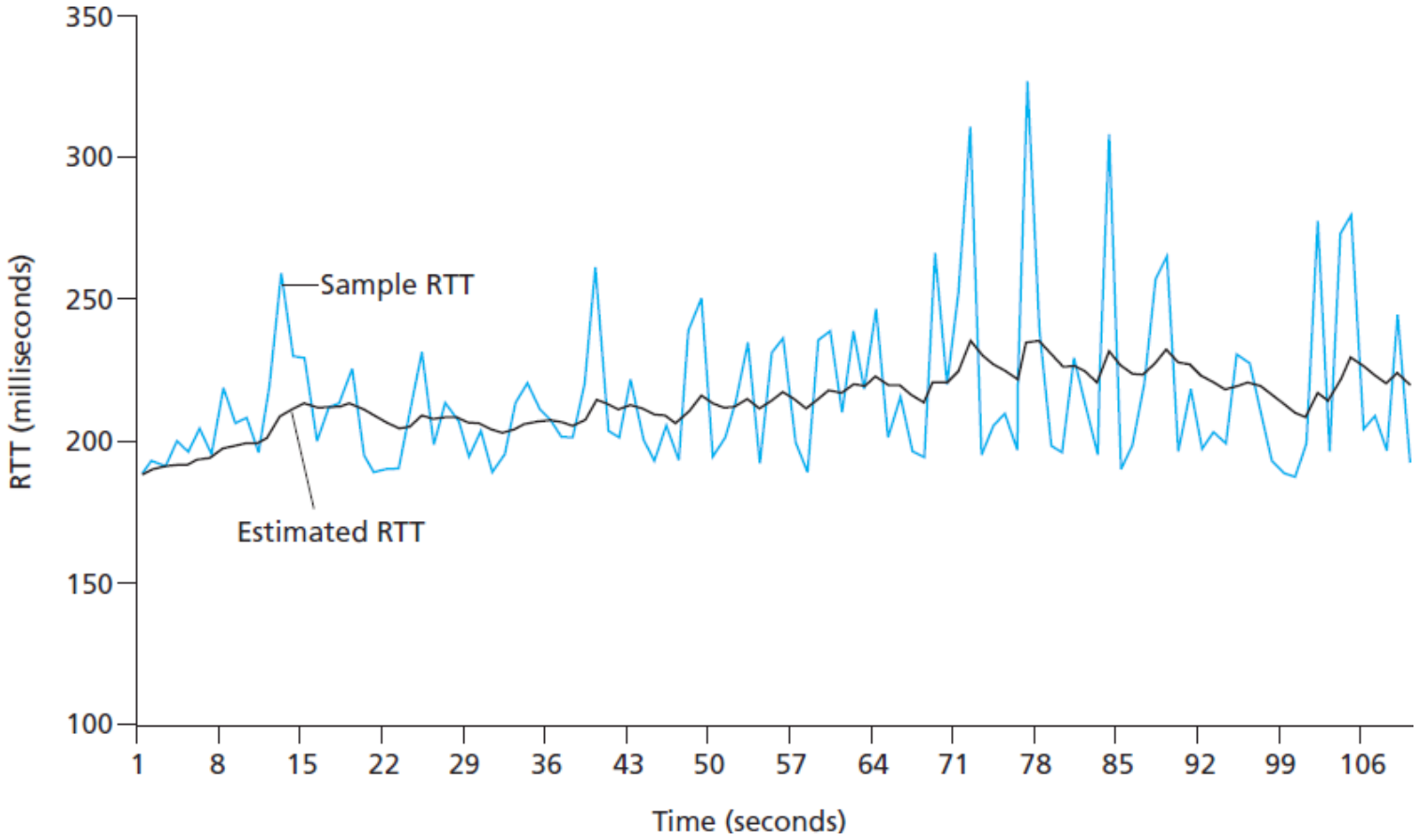
- Varios temporizadores
  - el más importante es el de retransmisión
- **Jacobson:**

  - **M[n]** = medida del RTT (llegada del ACK) obtenida por el segmento n-1.
  - **RTT[n]** = Estimador de RTT en el instante n.
  - **D[n]** = Estimador de desviación estándar en el instante n.

  - **RTT[n]** = a RTT[n-1] + (1 - a) M[n] a = 7/8
  - **D[n]** = b D[n-1] + (1 – b) |RTT[n-1] – M[n]| b = ¾

  - **Timeout[n]** = RTT[n] + 4 * D[n]
    Valor de timeout para el envío del segmento n
    Intuitivamente: Si llega mas tarde que la media mas cuatro veces la desviación estandar, es un evento “raro”.

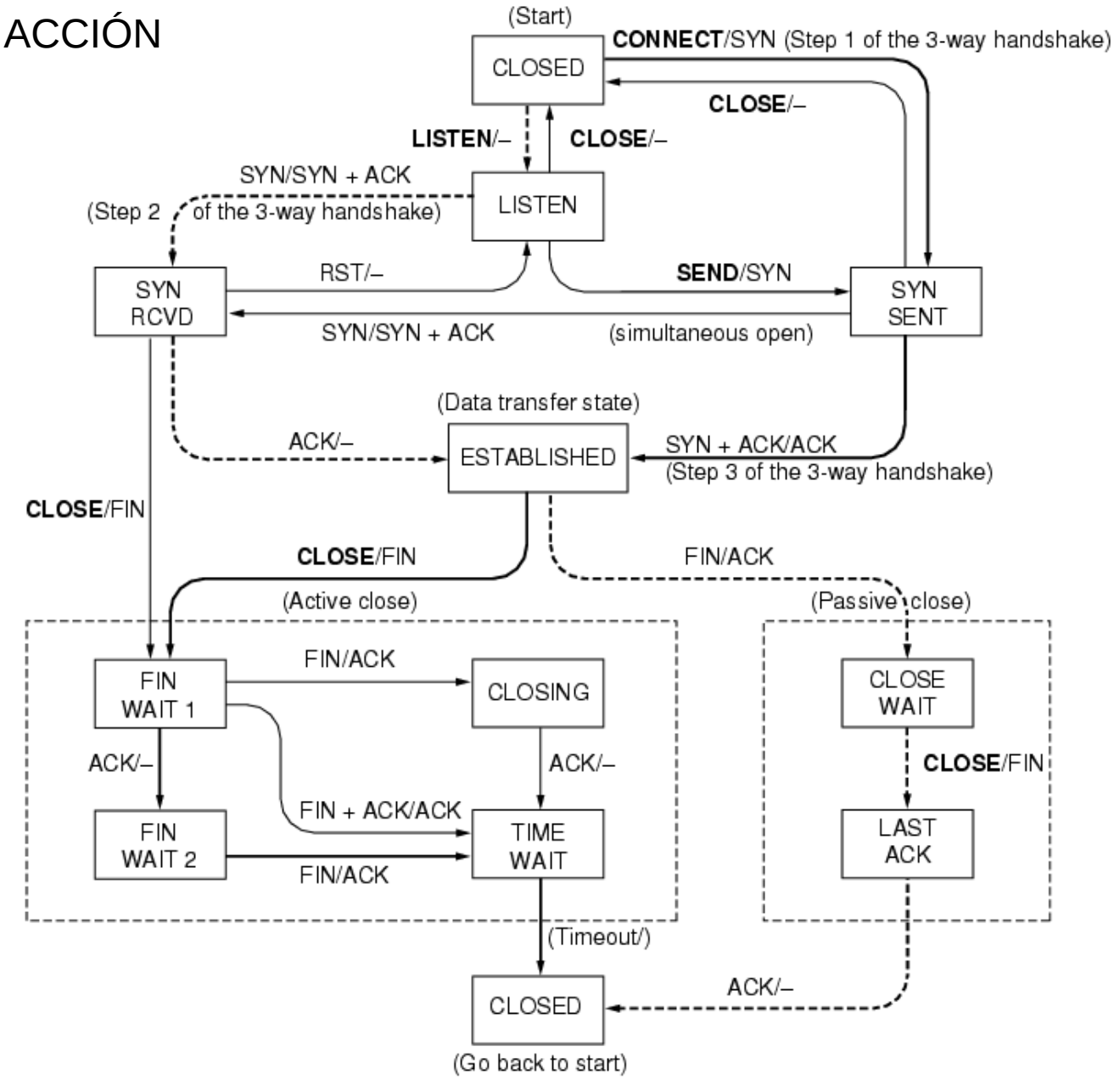
TCP – Gestión de Temporizadores



- Variantes agregadas por Karn:
  - No calcular sobre retransmisiones
  - Se duplica el timeout a cada pérdida (en el caso de varias pérdidas, el estimador de RTT no se puede actualizar)
- Moderno: se utiliza la opción Timestamp para mejorar el cálculo del RTT
- Hay varios temporizadores más.

TCP – Estados

ENTRADA / ACCIÓN

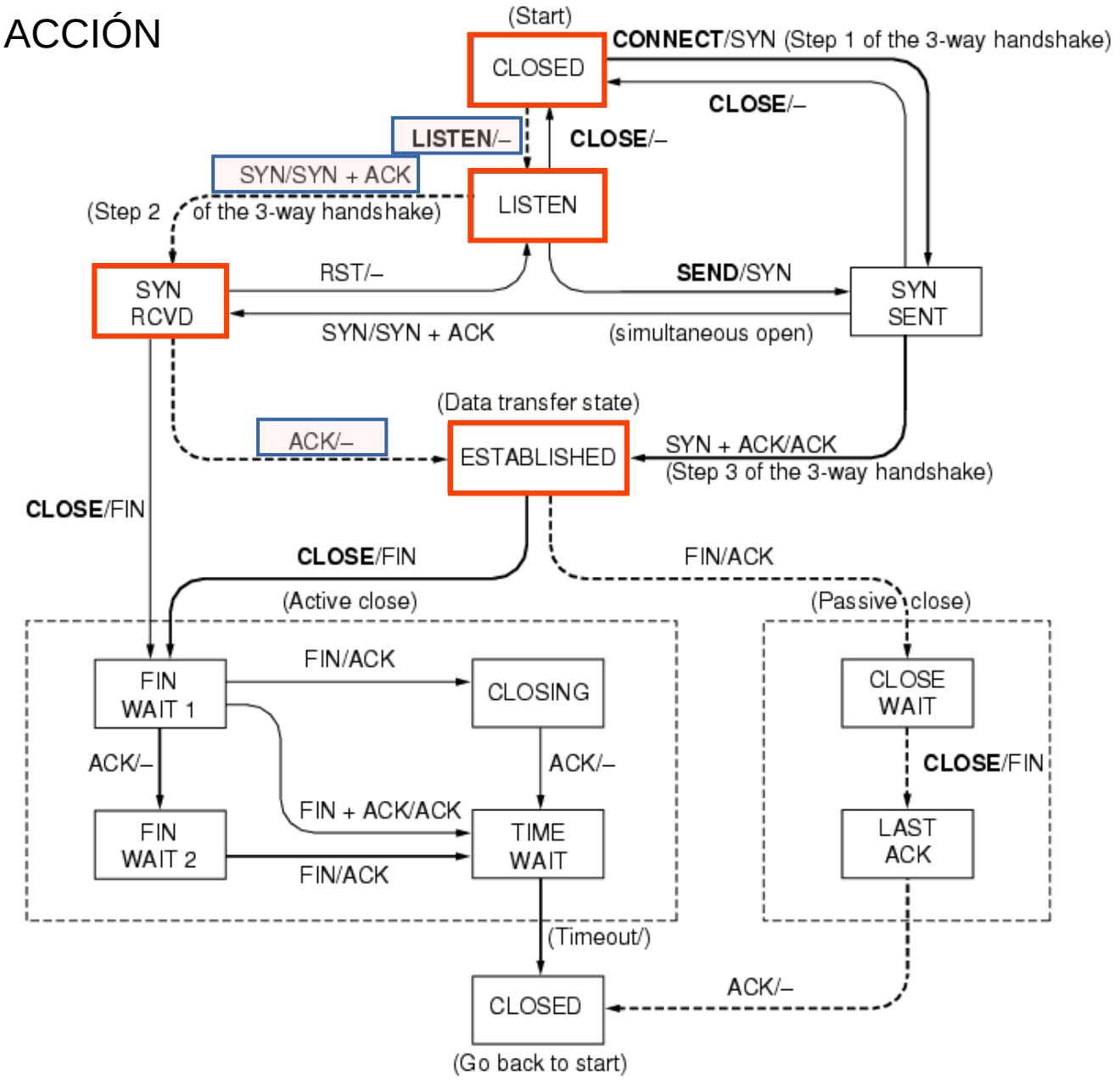


TCP – Significado de los Estados

State	Description
CLOSED	No connection is active or pending
LISTEN	The server is waiting for an incoming call
SYN RCVD	A connection request has arrived; wait for ACK
SYN SENT	The application has started to open a connection
ESTABLISHED	The normal data transfer state
FIN WAIT 1	The application has said it is finished
FIN WAIT 2	The other side has agreed to release
TIMED WAIT	Wait for all packets to die off
CLOSING	Both sides have tried to close simultaneously
CLOSE WAIT	The other side has initiated a release
LAST ACK	Wait for all packets to die off

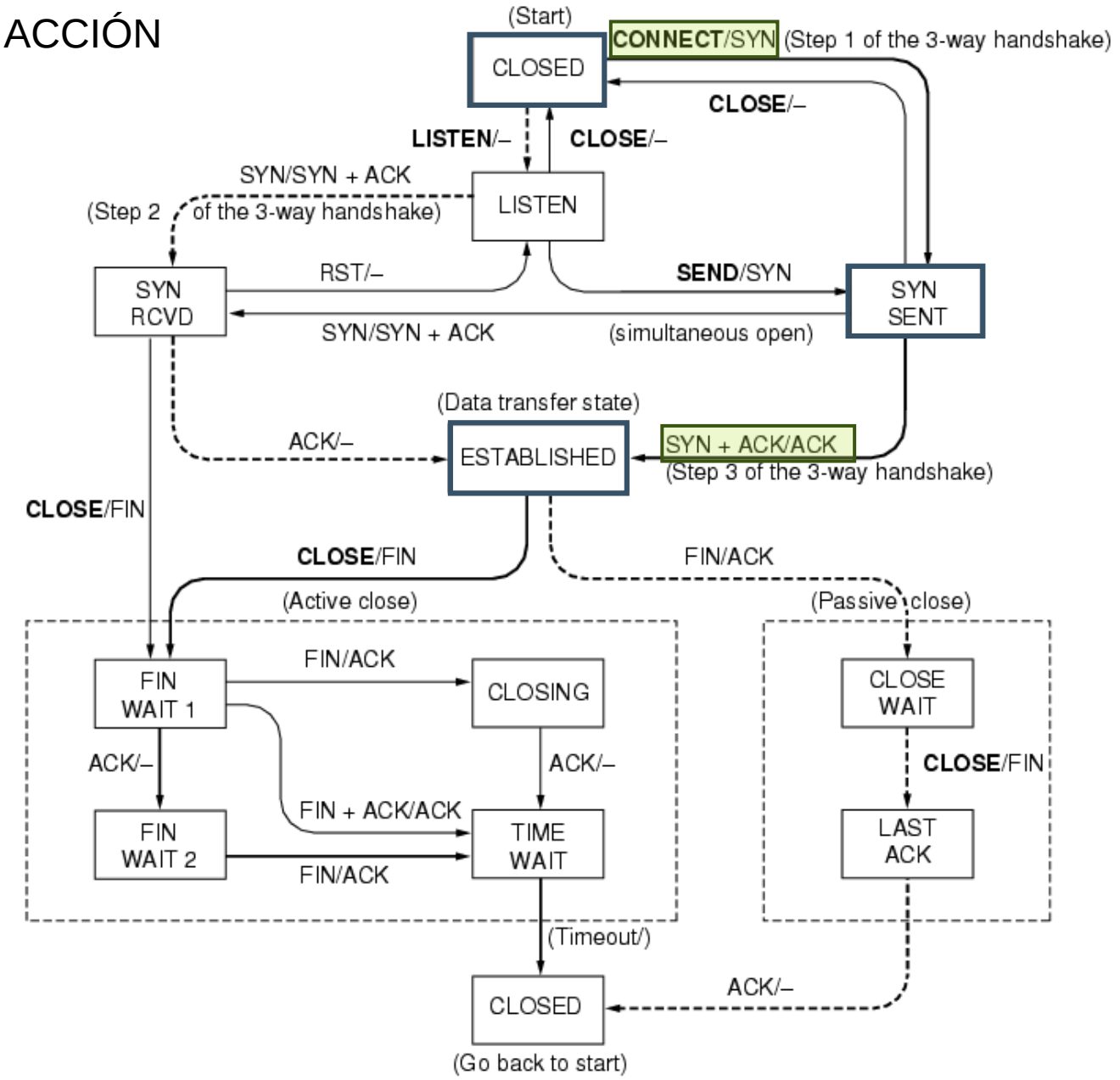
TCP – Estados – Inicio de conexión – Lado Servidor

ENTRADA / ACCIÓN



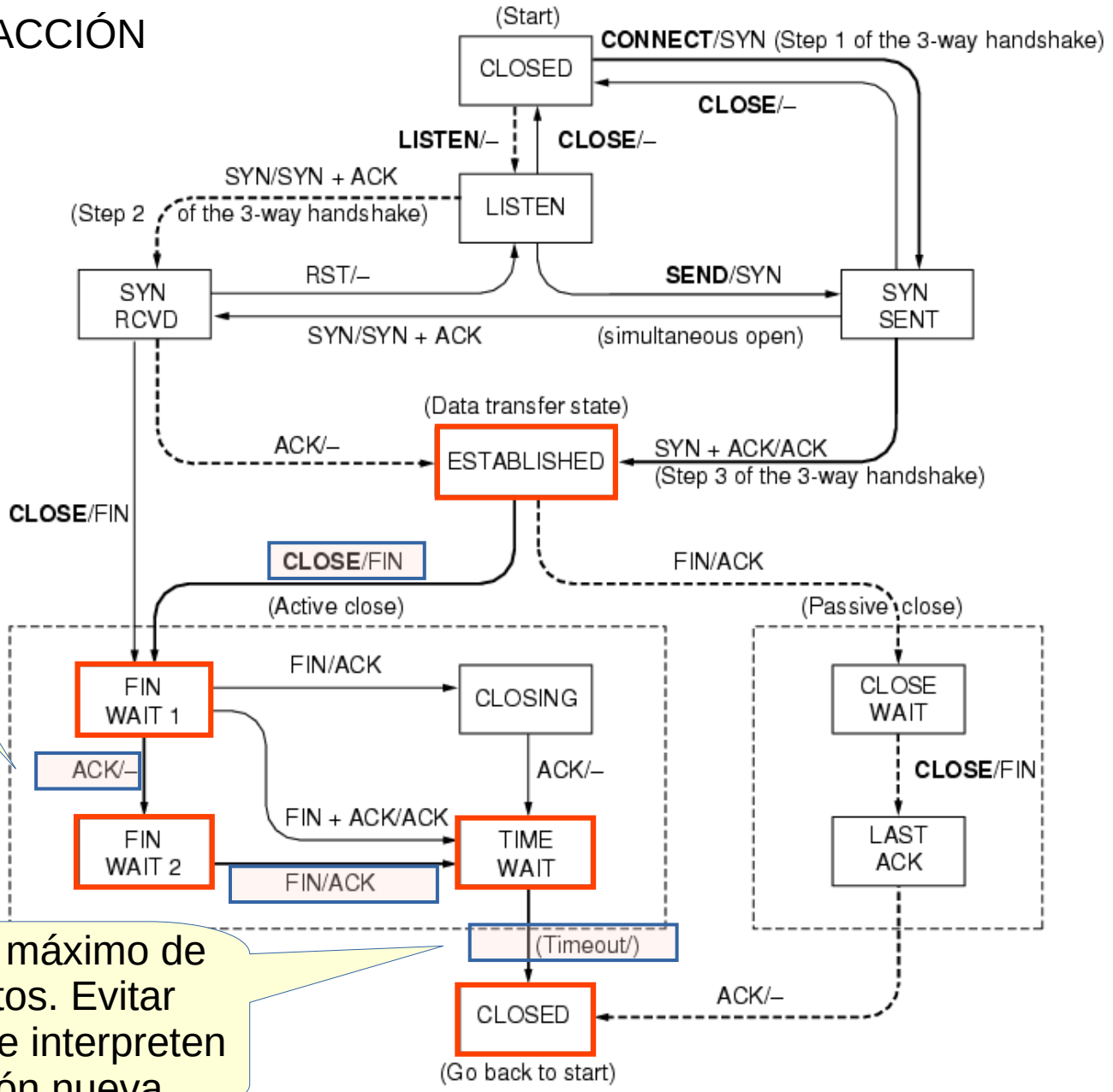
TCP – Estados – Inicio de conexión – Lado Cliente

ENTRADA / ACCIÓN



TCP – Estados – Fin de conexión activa

ENTRADA / ACCIÓN

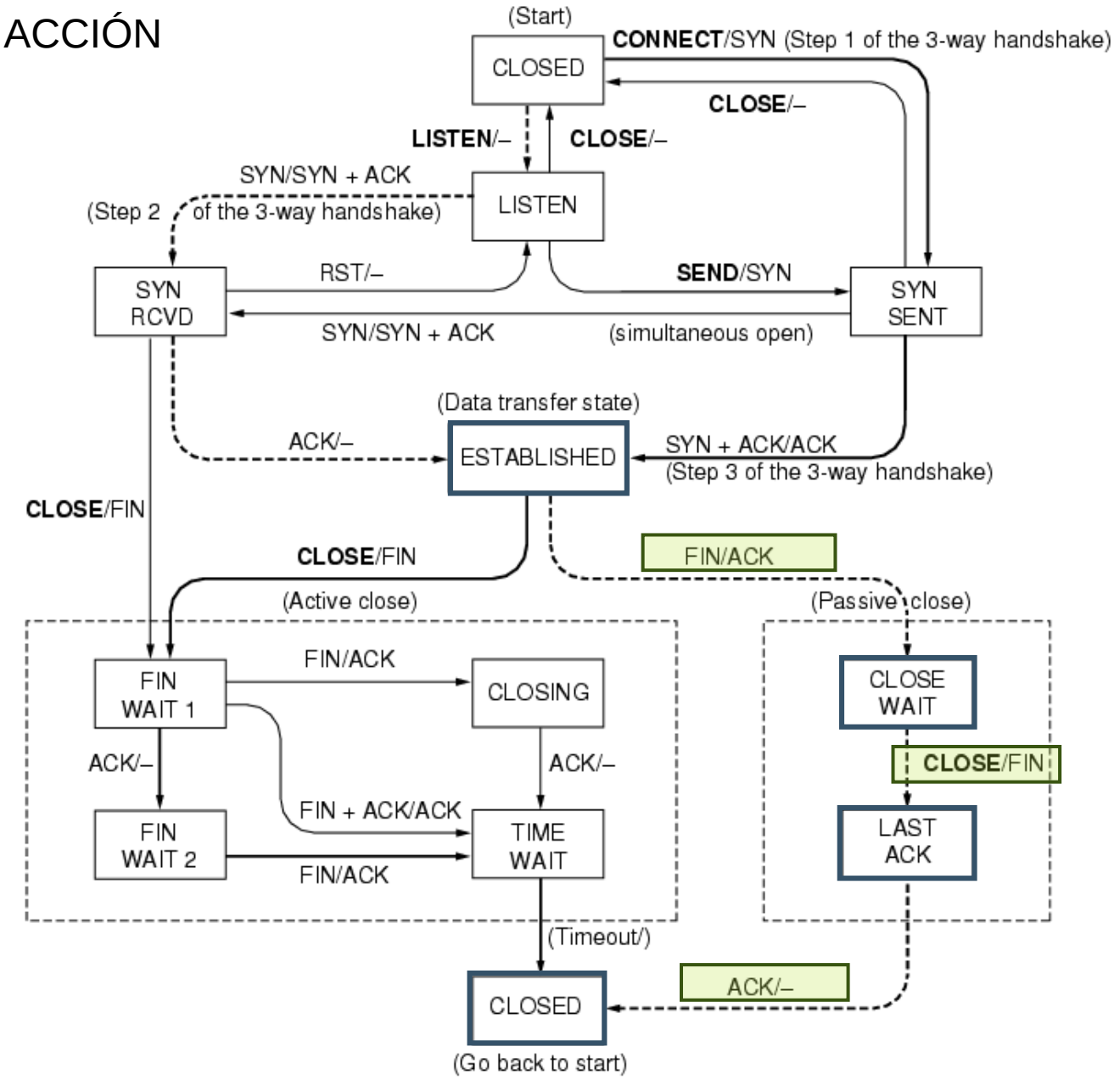


Finalización de 4 vías.

Espera un tiempo máximo de vida de segmentos. Evitar duplicados viejos se interpreten como información nueva.

TCP – Estados – Fin de conexión – Lado Pasivo

ENTRADA / ACCIÓN



TCP – Verificación de los Estados

Es posible verificar en el sistema operativo el estado de las conexiones, permite detectar algunos tipos de ataques (TIME_WAIT), puertos “tcp” esperando peticiones (LISTEN), conexiones con otros sistemas (ESTABLISHED).

```
valdes@ampere:~$ netstat -a
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 0.0.0.0:netbios-ssn     0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:9869            0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:47821           0.0.0.0:*               LISTEN
tcp        0      0 localhost:nut           0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:5926            0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:2633            0.0.0.0:*               LISTEN
tcp        0      0 ampere.fing.edu.uy:ssh  ampere.fing.edu.u:48912 ESTABLISHED
tcp        0      0 ampere.fing.edu.uy:ssh  uruguay.fing.edu.:38320 ESTABLISHED
tcp        0      0 ampere.fing.edu.u:51422 ceibo.fing.edu.uy:ssh   ESTABLISHED
tcp        0      0 ampere.fing.edu.u:48934 ampere.fing.edu.uy:ssh  ESTABLISHED
tcp        0      0 localhost:49366         localhost:2633          TIME_WAIT
tcp        0      0 ampere.fing.edu.uy:ssh  r167-58-254-39.di:58547 ESTABLISHED
tcp        0      0 ampere.fing.edu.u:60350 ohm.fing.edu.uy:ssh     ESTABLISHED
tcp        0      0 ampere.fing.edu.uy:nut  ceibo.fing.edu.uy:55704 ESTABLISHED
tcp        0      0 ampere.fing.edu.u:37134 hertz.fing.edu.uy:ssh   ESTABLISHED
tcp        0     53 ampere.fing.edu.uy:ssh  115.78.10.115:44346     FIN_WAIT1
```

TCP – Transisión de estados Estados a Establecida

client state

CLOSED

choose init seq num, x
send TCP SYN msg

SYN SENT

SYNbit=1, Seq=x

choose init seq num, y
send TCP SYNACK
msg, acking SYN

server state

LISTEN

SYN RCVD

SYNbit=1, Seq=y
ACKbit=1; ACKnum=x+1

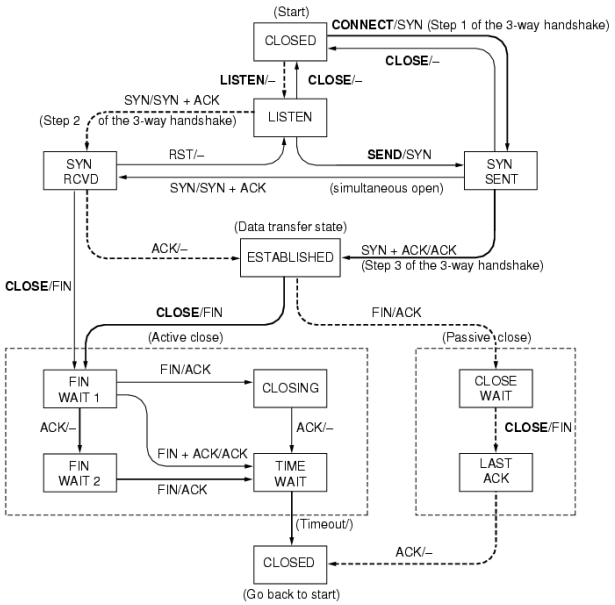
ESTABLISHED

received SYNACK(x)
indicates server is live;
send ACK for SYNACK;
this segment may contain
client-to-server data

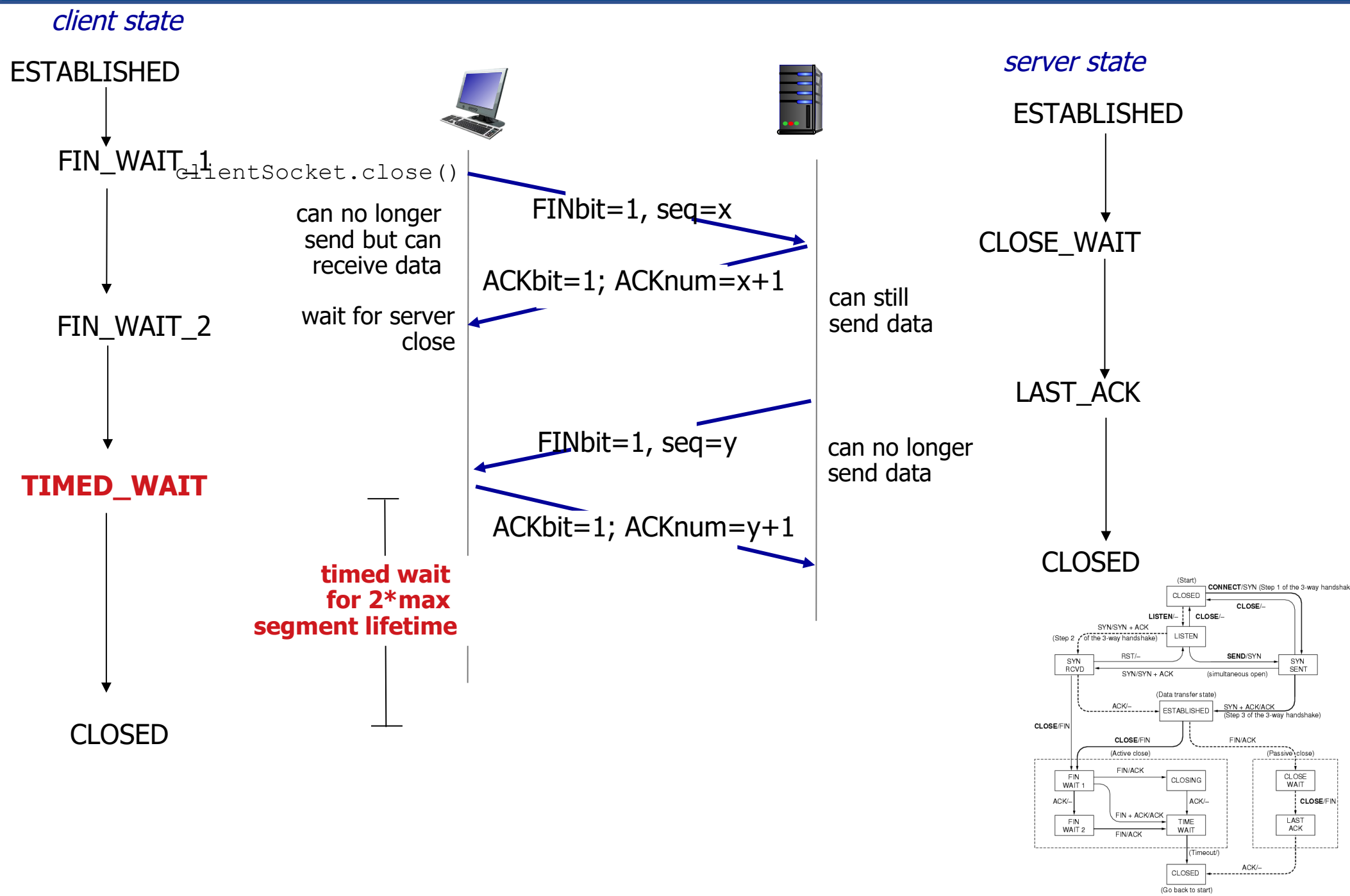
ACKbit=1, ACKnum=y+1

received ACK(y)
indicates client is live

ESTABLISHED



TCP – Transisión de estados Estados a Cerrado



TCP – Modificaciones

- Window Scale: ventana máxima de 64 KB (representada en 16 bits) “quedó chica”
  - Calculen la velocidad máxima de transmisión en un enlace de 1Gbps y 100ms de RTT con una ventana de receptor de 64 KB (5.2 Mbps)
  - Opción "**escala de ventana**“: indica cuantos bits "0" agregar a la derecha del valor de buffer
  - Se negocia la escala al comienzo de la conexión (syn), formalmente 2^escala * WS (máximo 14)

    A 1 Gbps y 100 ms RTT y ventana de receptor 2^(16+14) el ancho de banda máximo es ~86 Gbps.
- SACK (Reconocimiento selectivo)
  - Permite indicar qué segmentos se recibieron correctamente posteriores al indicado en el campo ACK del segmento.

TCP – Modificaciones

- Enlace de 56 Kbps -> los números de secuencia cubren $\mathbf{W_{max}}$ (2^{31}) en 3.6 días
- 1 Gbps -> cubren $\mathbf{W_{max}}$ en 17 segundos!!
  - **Tiempo de vida máximo asumido por TCP de un paquete en la red: 120 Segundos!**
- Solución: Se utiliza el número de secuencia junto con una opción **Timestamp** para detectar duplicados viejos
  - Receptor debe verificar que el valor de timestamp sea **monótono creciente**, descartando segmentos antiguos
  - Timestamp también sirve para mejorar el cálculo del RTT
  - **Observación:** utilizar Windows Scale, nos lleva a utilizar timestamp (están en la misma RFC).

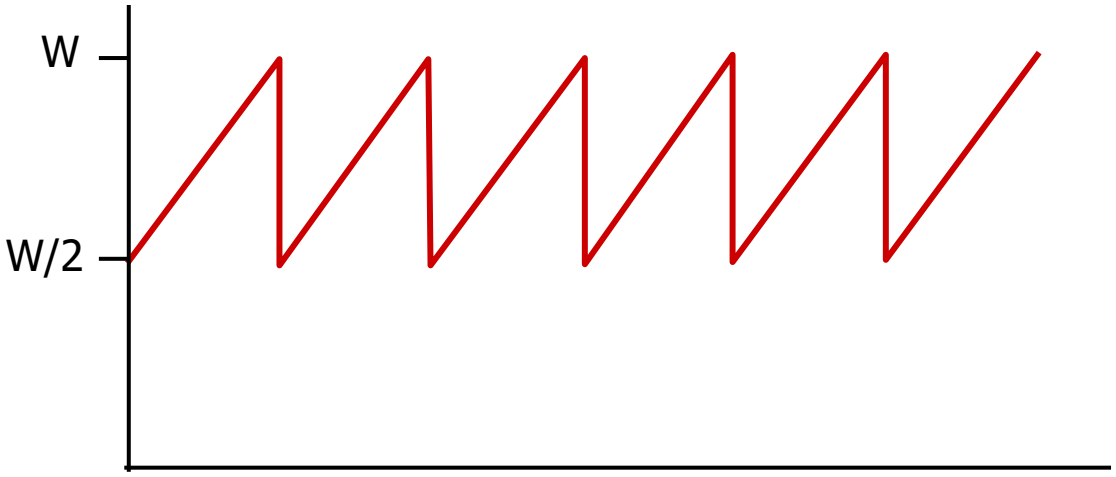
TCP – Throuhput

- ¿Cuál es el avg. TCP throughput en función window size (**W**), y el **RTT**?
- **(Control de Flujo, W = flujo)**

$$\text{avg TCP thruput} = \frac{W}{RTT} \text{ bytes/sec}$$

- **(Control de Congestión, W = cwnd)**

  Supuestos: ignorando slow start, siempre estoy enviando datos, cuando alcanzo cwnd = W se produce una pérdida, TCP RENO.
  - avg. window size (# in-flight bytes) is ¾ W
  - avg. throughput is 3/4W per RTT



$$\text{avg TCP thruput} = \frac{3}{4}\frac{W}{RTT} \text{ bytes/sec}$$

TCP – Throughput y tasa de pérdidas

- throughput en términos de **L** la probabilidad de pérdida o error de segmento [Mathis 1997]:

$$\text{TCP throughput} = \frac{1.22 \cdot MSS}{RTT\sqrt{L}}$$

- **Ejemplo:** MSS = 1460 bytes y RTT = 100 ms
- L = 10^{-2} => Th ~ 1,42 Mbps
- L = 10^{-4} => Th ~ 14,2 Mbps
- L = 10^{-6} => Th ~ 142,5 Mbps
- L = 10^{-7} => Th ~ 451 Mbps
- L = 10^{-8} => Th ~ 1,42 Gbps
- L = 10^{-9} => Th ~ 4,51 Gbps
- L = 10^{-10} => Th ~ 14,25 Gbps

Si bien la tasa de intercambio depende de la tasa de errores, este límite es por conexión TCP (pueden haber varias en simultáneo).

A tasas razonables el límite "seguramente" lo imponga la capa física