CAPA TRANSPORTE CAPITULO 3

Semestre B2010

Prof. Andrés Arcia-Moret

Basasdo en el laminario de Kurose-Ross 5ta edición

Introducción / Servicios Transporte

- Protocolo capa-T garantiza la comunicación lógica entre procesos remotos.
- Implementado en los extremos (no en routers)
- Unidad de trabajo: el segmento.
- TCP pasa a IP (sender) y luego IP a pasa a TCP (receiver)
- ¿Cuáles? TCP, UDP y DCCP.

Relación Capa Transp/Red

- Transporte → comunicación entre procesos.
- Red → comunicación entre hosts.
- Ej: Distribuir el correo en un caserio.

1/ Se agrupan los correos (msg) de persona (proceso) dentro cada casa (host)

2/ Los empleados de la oficina de correo fungen de Protocolo de Transporte (y de capa Sockets)

3/ El servicio postal (el transporte aereo, terrestre) es la capa de red

Modelo de Servicio

- Si se sustituyen los empleados dentro del servicio postal por otros (Ej. menos estrictos) el modelo de servicio **cambia**. Inclusive, radicalmente.
- Caso de UDP vs. TCP

Vista General (1/2)

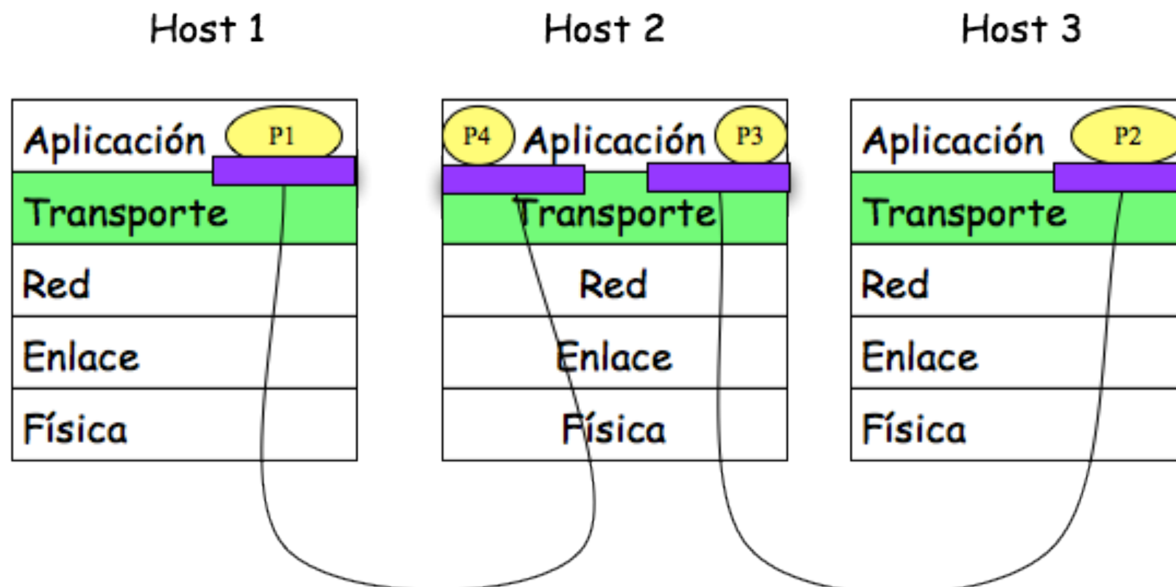
- UDP → servicio no confiable, sin conexión
- TCP → servicio confiable, orientado a conexión (ambos usan segmentos).
- Capa de Red → IP (no confiable) → 1 IP = 1 host
  - Servicio sin garantías de: entrega, orden e integridad.
- TCP extiende servicio de IP entre 2 procesos.
- Hay multiplexado y demultiplexado de segmentos.

Vista General (2/2)

- Entrega de Proceso a Proceso + Chequeo Errores = UDP.
- TCP ➔ entrega confiable = control de flujo + numero de seq + ACKs + timers
- Control de congestion → complejidad añadida

Mux/Demux

- Extensión de entrega host-to-host a process-to-process.
- ¿Cómo se dirije la data de la entrada del host al proceso?
  - A través de los sockets.



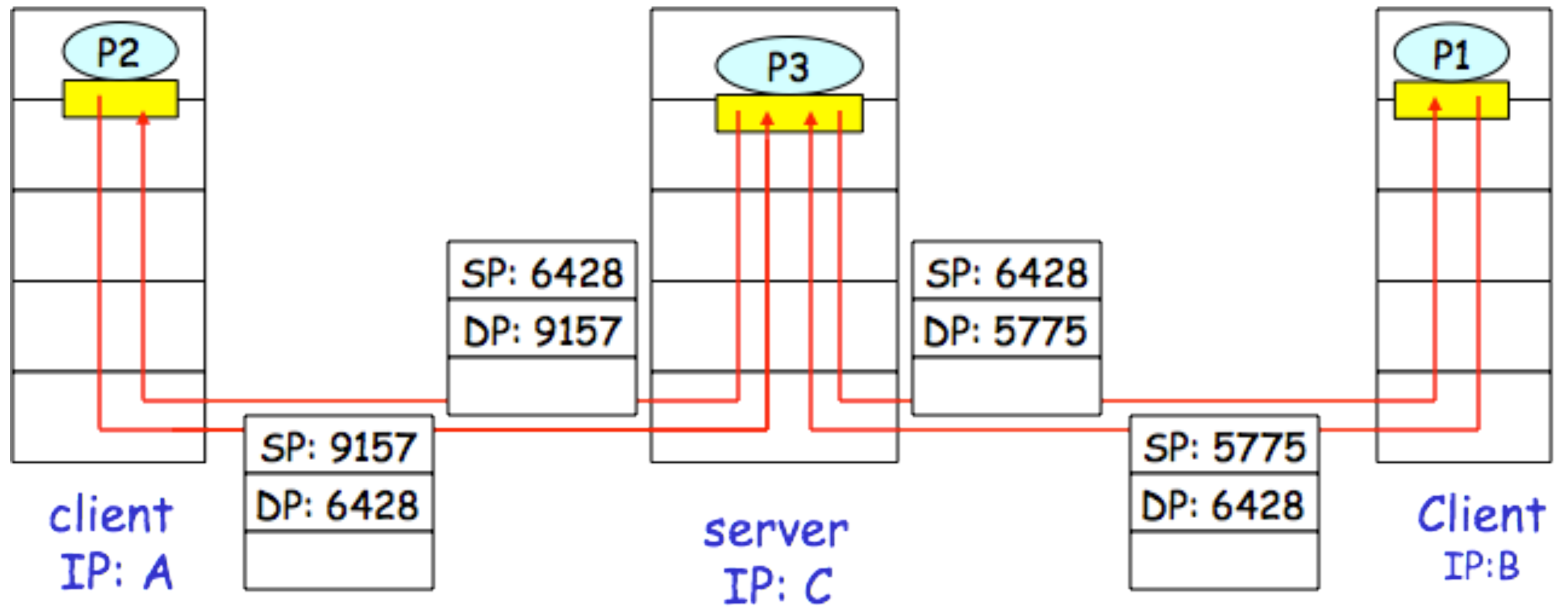
Mux/Demux para UDP

- Un socket debe tener UIDs (0-65535 → puerto)
  - De 0 a 1023 son bien conocidos (www.iana.org)
- Data de capa transporte y va a sockets directamente.
- Socket **cliente** asignado por el **kernel**, Socket **servidor** por el **programador.**
- Se crea paquete con socket origen y destino (usado en el retorno) .
- Basado en IP/Best Effort de receptor.
- Socket UDP = IP dest + Puerto dest (poco importa el origen).

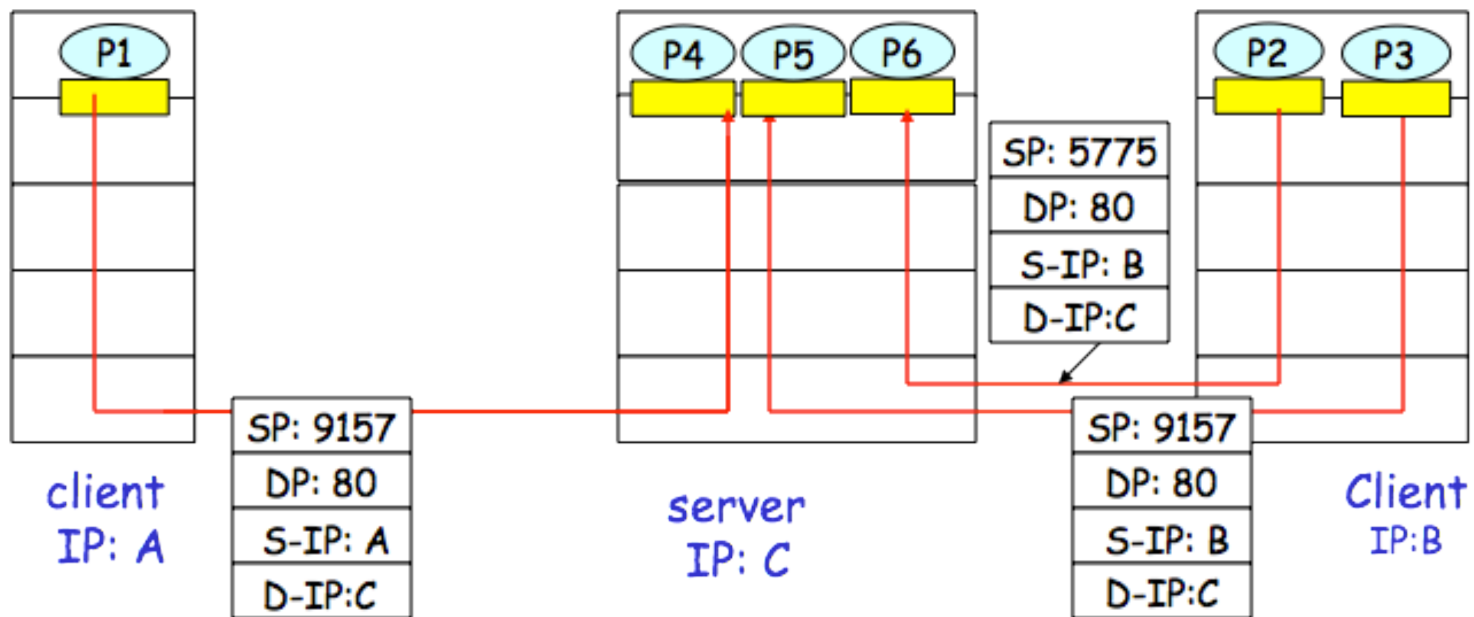
Mux/Demux para TCP

- Identificado por la 4-tupla:
  - <IP src, Port src, IP tgt, Port tgt>
- Dos segmentos con distinto origen van a sockets distintos.
- La aplicación dispone de un socket de bienvenida.
- Cuando llega una petición se crea socket nuevo.

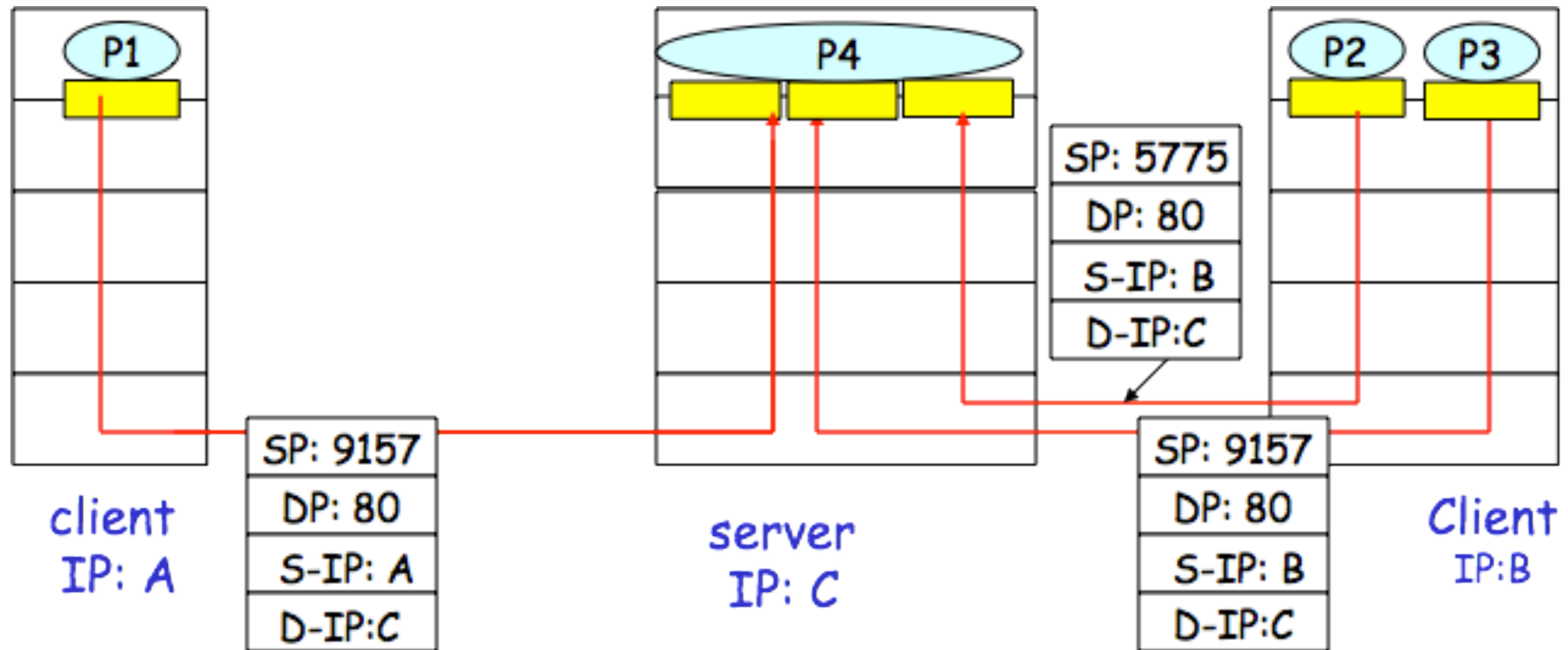
Ejemplo Demux 1 (UDP)



Ejemplo Demux 2 (TCP)



Ejemplo Demux 2 (TCP)



User Datagram Protocol

- Diseño de un protocolo de transporte atrevido:
  - Los datos pasan directos de la capa aplicación a la red y de la red a la aplicación
  - Hay que proveer multiplexación y demultiplexación.
  - No hay handshaking ⟶ sin conexión, encapsular IP + hacer entrega Best-Effort.
  - Ej: DNS envia cada pregunta directamente en un segmento UDP.

¿Por qué UDP en vez de TCP?

- Control de la data se envía a la aplicación
  - UDP → Data → encapsula segmento → capa de red
  - TCP → Data → encapsula segmento → **Control Congestión** → capa de red
- No hay handshaking para conexión
  - TCP hace a la aplicación mucho más lenta (ej: *obtención de documentos con HTTP*)
  - DNS es rápido porque usa UDP

¿Por qué UDP en vez de TCP?

- UDP no tiene estado de la conexión
  - Ocupa menos recursos (ej: procesador y memoria)
  - Soporta más clientes ⟶ es más escalable
- UDP tiene encabezado más pequeño, 8 bytes contra 20 bytes (TCP).
- UDP bueno para protocolos acumulativos (es decir, que mensajes más recientes prevalecen sobre los otros).

Aplic. Inet y Prot. de Transporte

Aplicación	Capa Aplic.	Protocolo Transporte.
e-mail	SMTP	TCP
acceso remoto	Telnet	TCP
Web	HTTP	TCP
Trans. Archivos	FTP	TCP
Serv. Archivos	NFS	UDP
Streaming	propietario	UDP o TCP
Telefonia	propietario	UDP o TCP
Admin. Redes	SNMP	UDP
Ruteo	RIP	UDP
Nombres	DNS	UDP

Composición segmento UDP

- DNS → pkt tiene una pregunta o una respuesta
- Streaming → pkt tiene muestras (samples)
- En el receptor
  - Demultiplexar al host correcto
  - Hacer checksum para verificar integridad

Checksum UDP

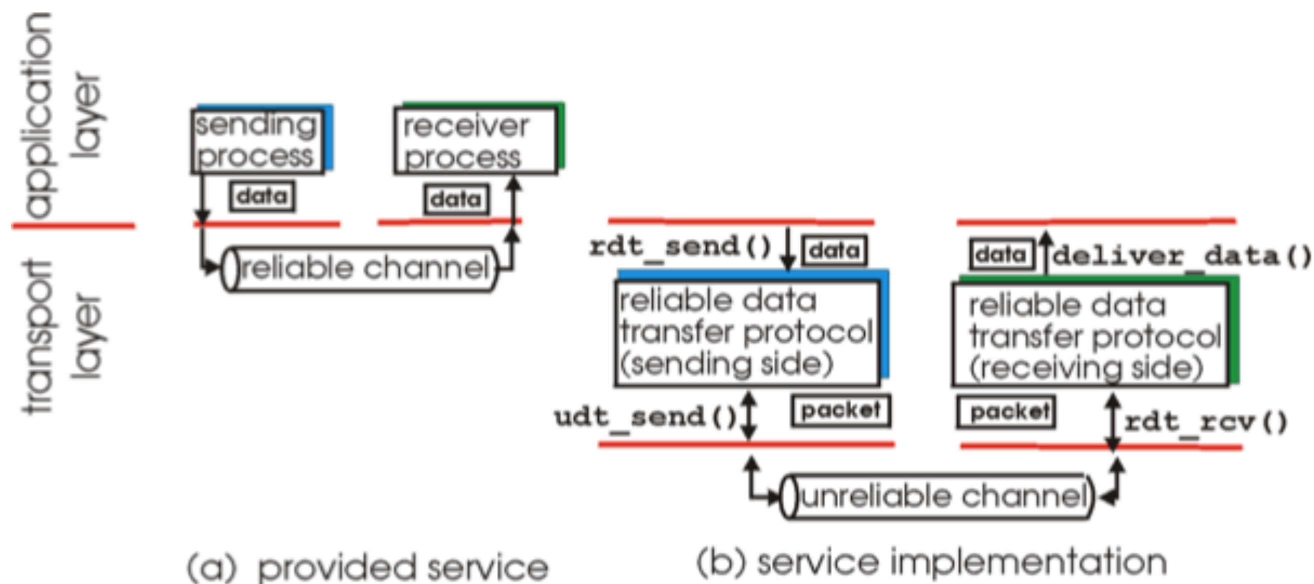
- Hacer el complemento a 1 de la suma de todas las palabras de 16 bits (con envoltura del acarreo final).
- Colocar el complemento 1 en el encabezado UDP.
- Al hacer el “O” logico debería dar todos en uno.
- UDP hace un chequeo de extremo (principio de extremo-a-extremo).

Transmisión Confiable.

- Veremos principios de transmisión aplicables a cualquier capa.
- Transporte de Datos está en el **Top-10** de problemas fundamentales de redes
- Todos los bits llegan y en orden.

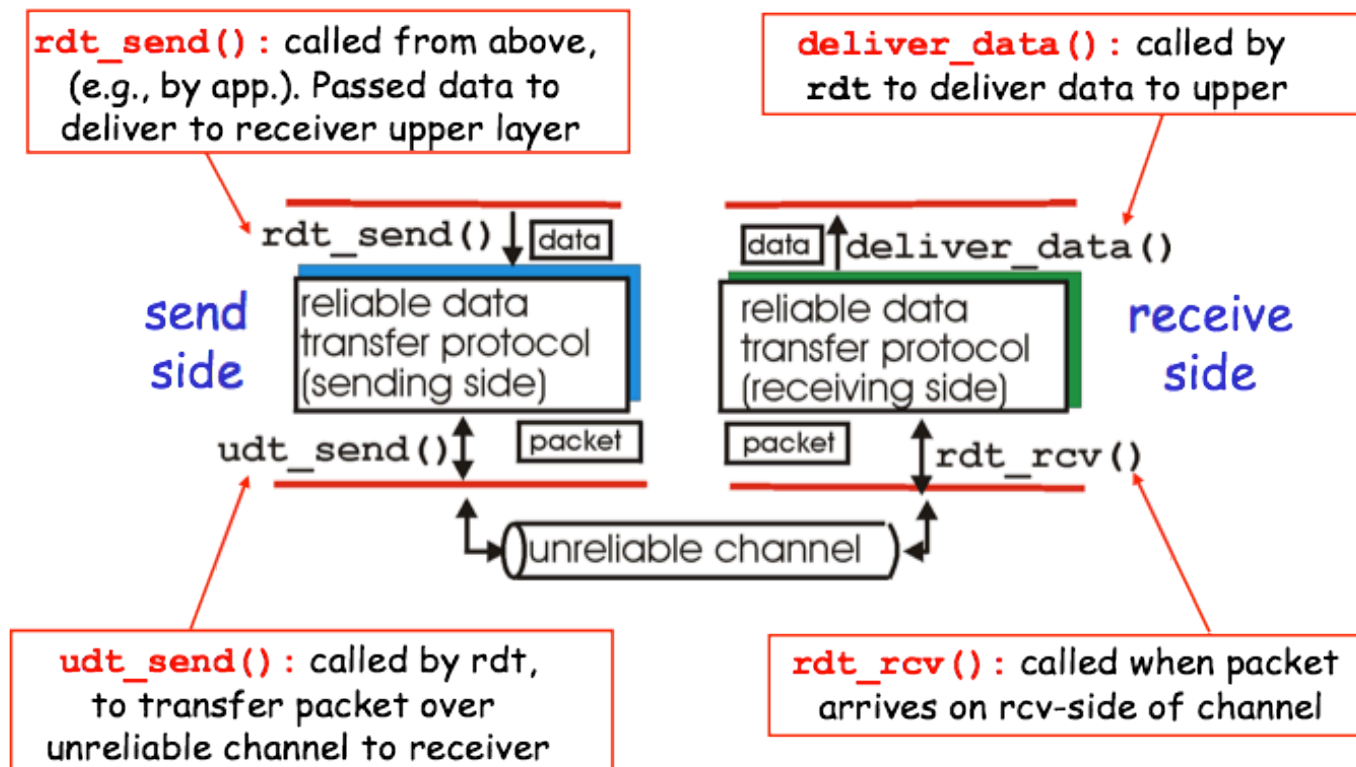
Desarrollo de un Protocolo Confiable

rdt: reliable data transfer
udt: unreliable data transfer



(a) provided service (b) service implementation

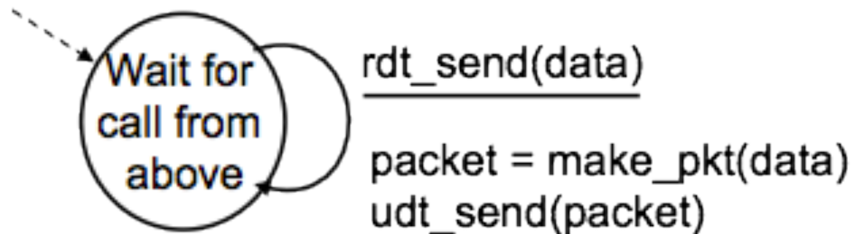
Desarrollo de un Protocolo Confiable

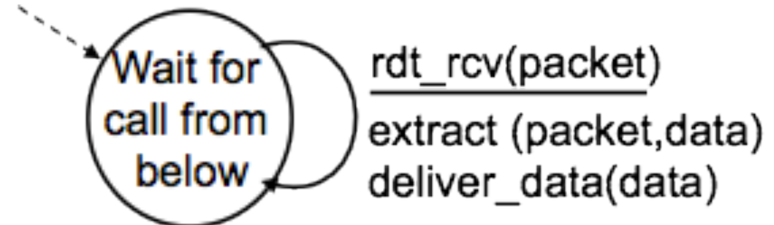


rdt 1.0, en canal perfecto

- Protocolo basado en un canal confiable
  - No hay errores en bits
  - No hay pérdida de datos
- Se transmite del emisor al receptor (bidireccional no más complejo pero si más engorroso de explicar)
- Delay *CERO* entre el cliente y servidor ⟶ receptor no será inundado.

rdt 1.0, en canal perfecto





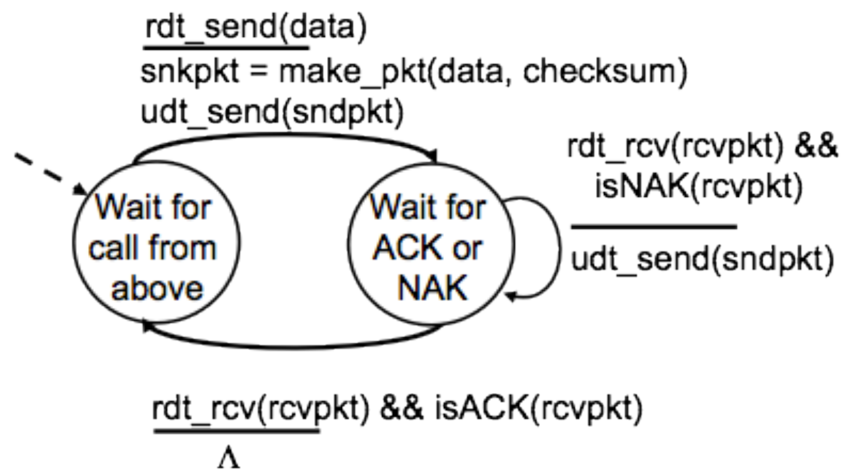
sender

receiver

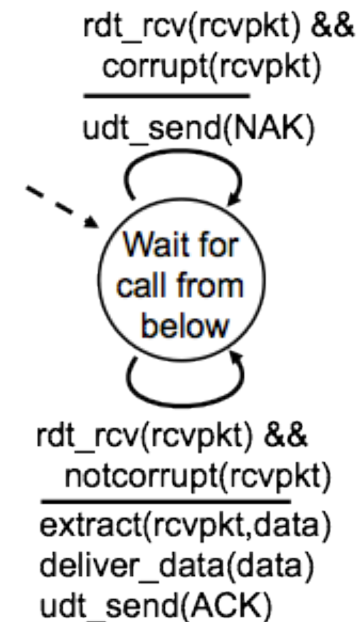
rdt 2.0, en canal con errores

- Modelo más realista: errores en la propagación, transmision o almacenamiento.
- Uso de reconocimientos positivos (**ACKs**) o negativos (**repita por favor, NACKs**). ⟶ Protocolo ARQ.
- Se bloquea a la espera de recepción ⟶ Conocido como **stop-and-wait**.
- Mecanismos nuevos: detección de error, retroalimentación (ACK / NACK).

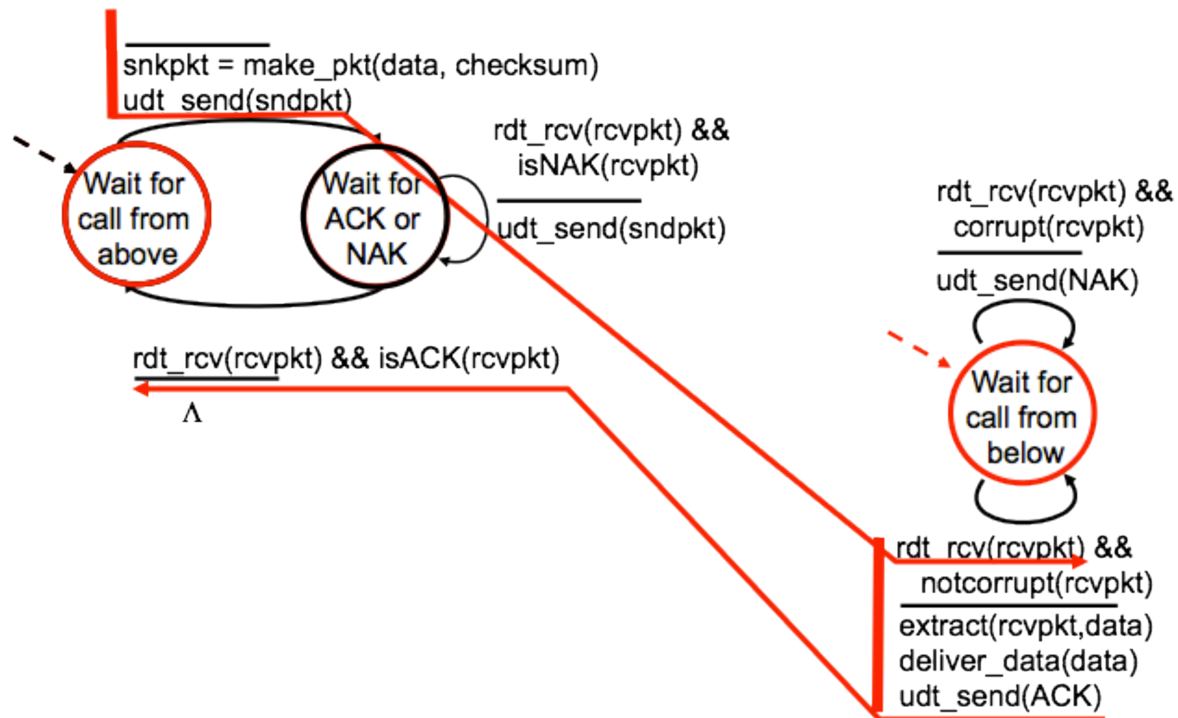
rdt 2.0, en canal con errores



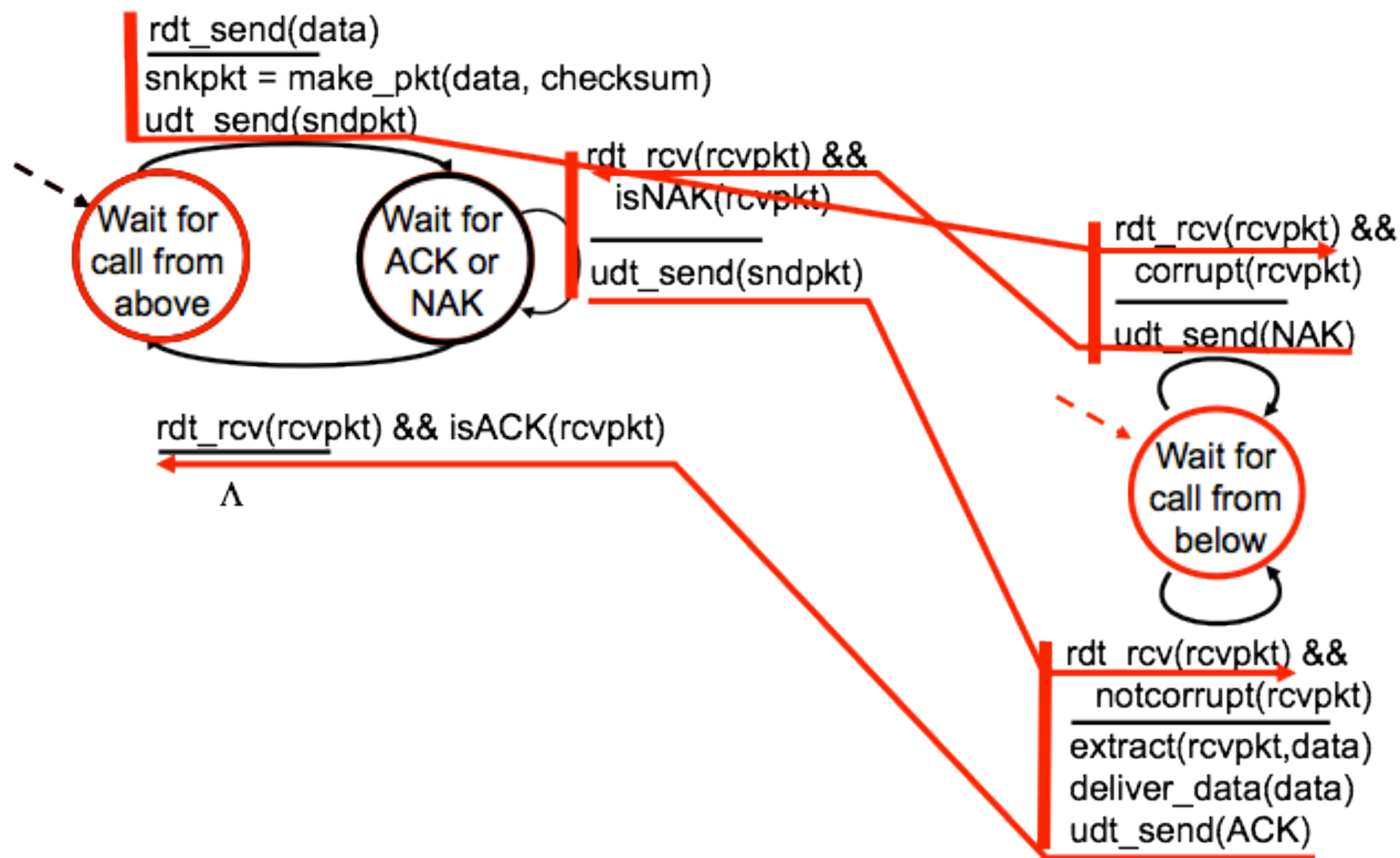
receiver



rdt 2.0, operación sin error (1)



rdt 2.0, operación con error



¿Cómo manejar errores de bits?

- Mecanismo de detección de error (checksum, extra bits, etc.)
- Mecanismo de Feedback del receptor
  - Necesidad debida a la distancia ⟶ se usan ACKs y NACKs.
- Mecanismo a través de retransmisión.

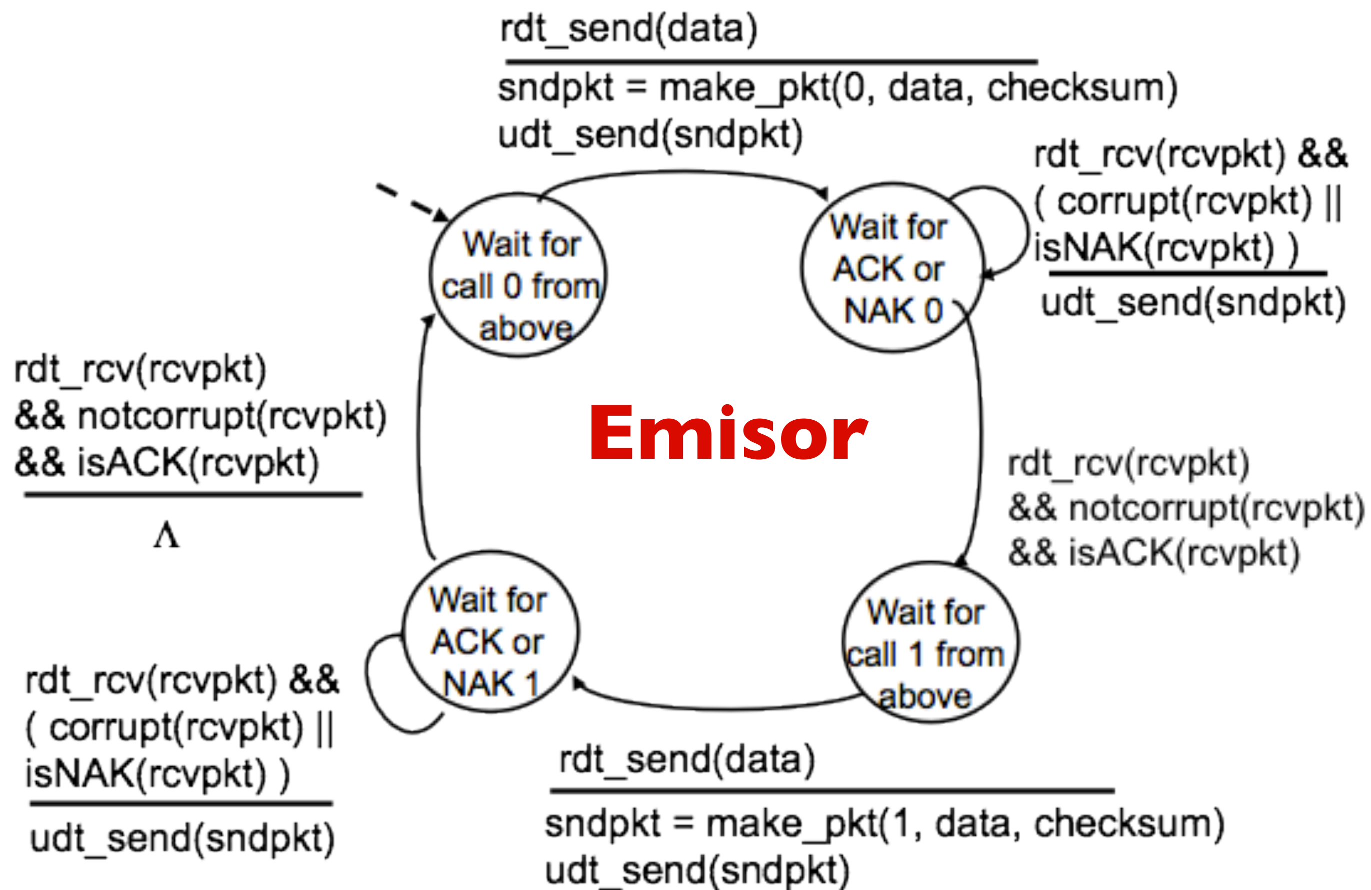
¿Cómo mejorar rdt 2.0?

- ¿Qué sucede si un ACK o NACK se corrompe?
- ¿Cómo se corrige ésta limitación?
  - Pedir que repita lo que acaba de enviar, pero y ¿si se vuelve a corromper?
  - Añadir suficientes bits para auto-corregir el paquete.
  - Reenviar (el que envia **data**) siempre que se reciba un ACK corrompido
  - Pero el receptor nunca sabrá si su ACK/NACK llegó bien
- Solución:
  - Colocar números de secuencia para saber si es una retransmisión.
  - Para **stop-and-wait** 1 bit es suficiente.

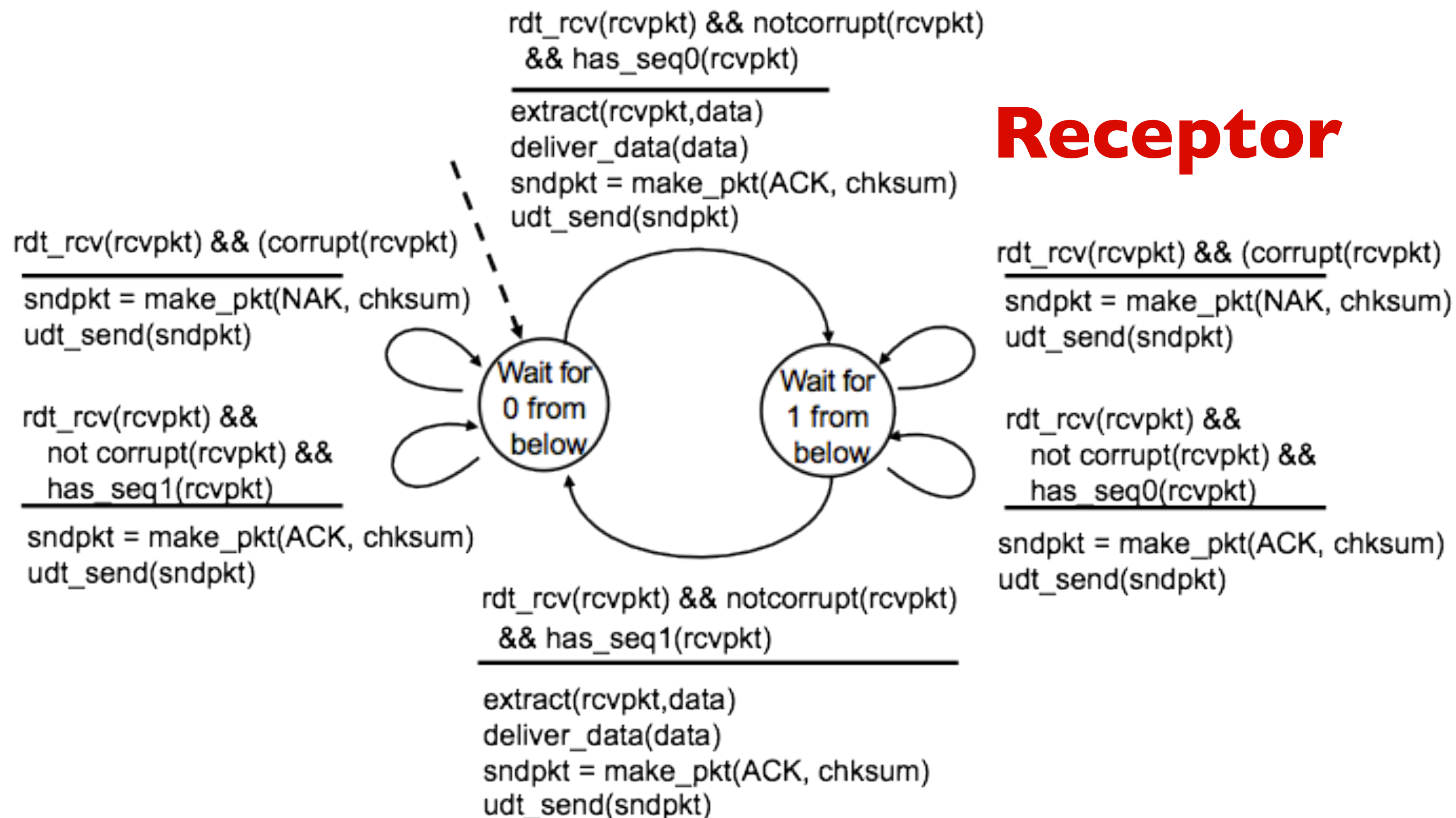
rdt 2.1, maneja ACK/NACKs desordenados

- Se añaden números de secuencia (0 y 1) y es suficiente (¿Por qué?)
- Estados de “0” son espejo de los estados para “1”.
- Considera:
  - data fuera de orden ⟶ reportada por ACKs duplicados
  - si ACK / NACK esta corrompido ⟶ emisor reenvia data inmediatam.
  - data corrompida en el receptor ⟶ (re)enviar NACK

rdt 2.1, maneja ACK/NACKs desordenados



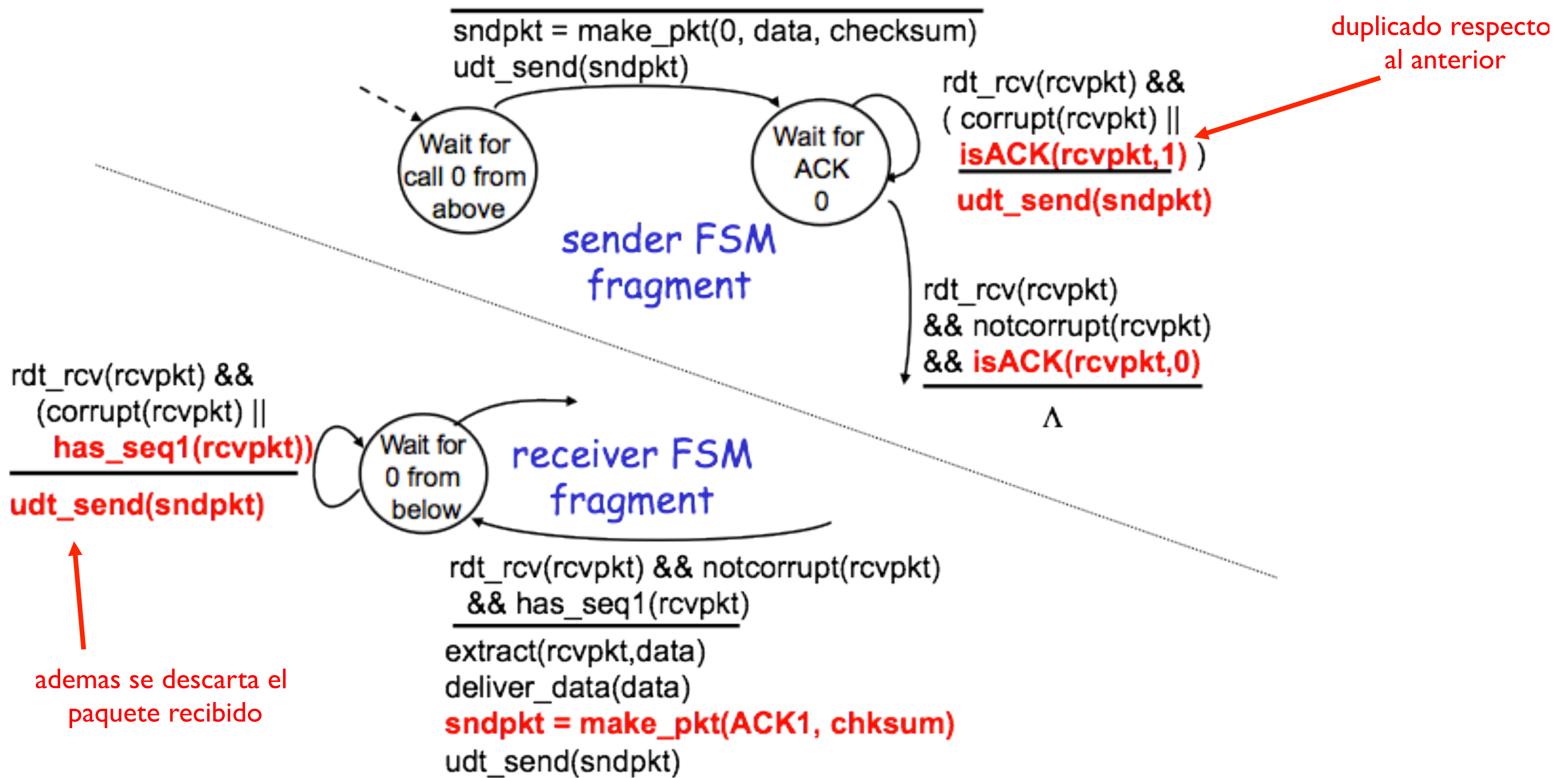
rdt 2.1, maneja ACK/NACKs desordenados



rdt 2.2, SIN NACKs

- Se puede mejorar aun la versión 2.1 enviando solamente ACKs duplicados
- 1 ACK duplicado (dupack) indica que el siguiente paquete al número de secuencia no llegó bien.
- Se agrega el número de secuencia del ACK.

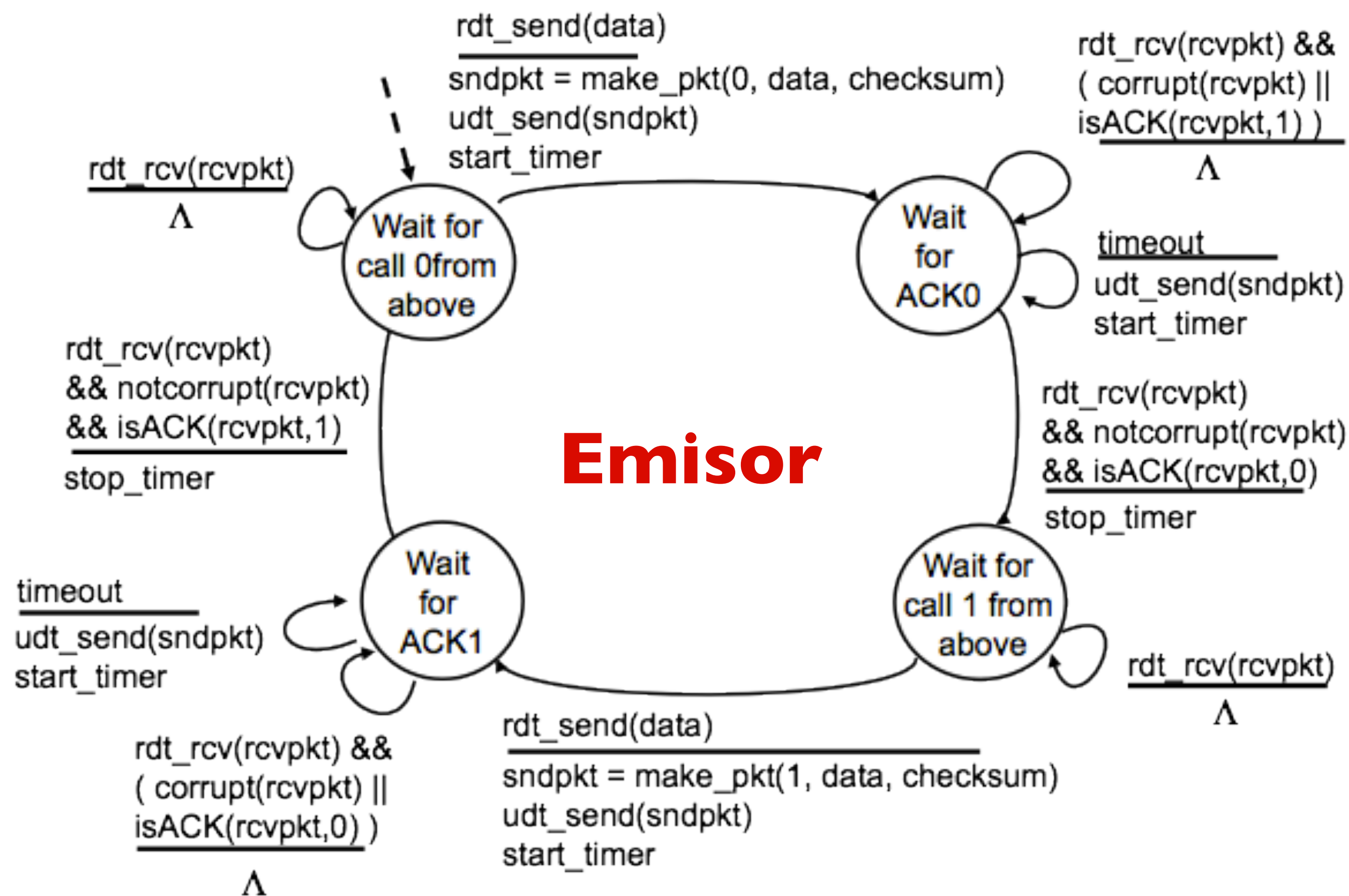
rdt 2.2, SIN NACKs



rdt 3.0, para canal ruidoso y pérdidas

- Además de corromperse los paquetes, se pierden
  - Lo anterior ayuda: checksum, número de secuencia, ACKs, retransmisiones, pero es <u>NO SUFICIENTE</u>
- ¿Cómo detectar perdidas? ¿Qué hacer luego?
- Protocolo que detecte y recupere pkt perdidos.
- Esperar al menos 1 RTT (puede faltar precisión)
  - Introduce duplicados
- Retransmitir es **la solución** para el que envía
  - Pero se necesita un Temporizador o *Timer*
- Se le llama <u>**protocolo de bit alternado**</u>

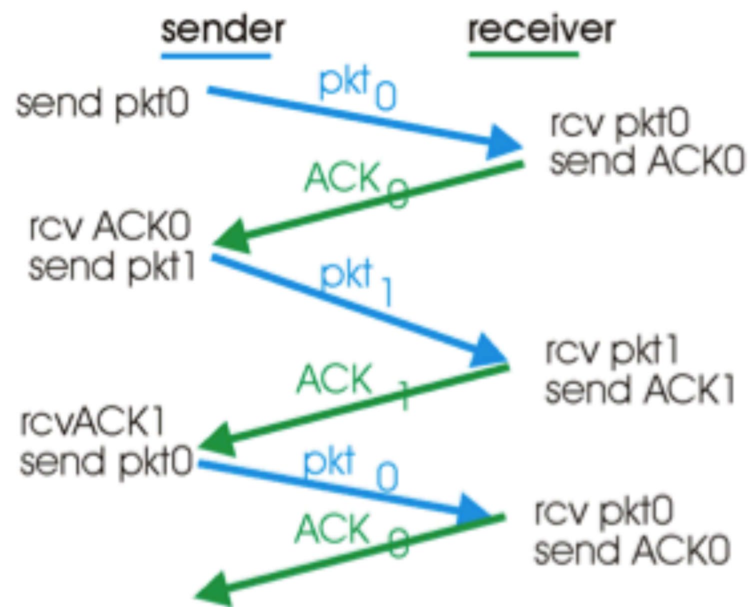
rdt 3.0, para canal ruidoso y pérdidas



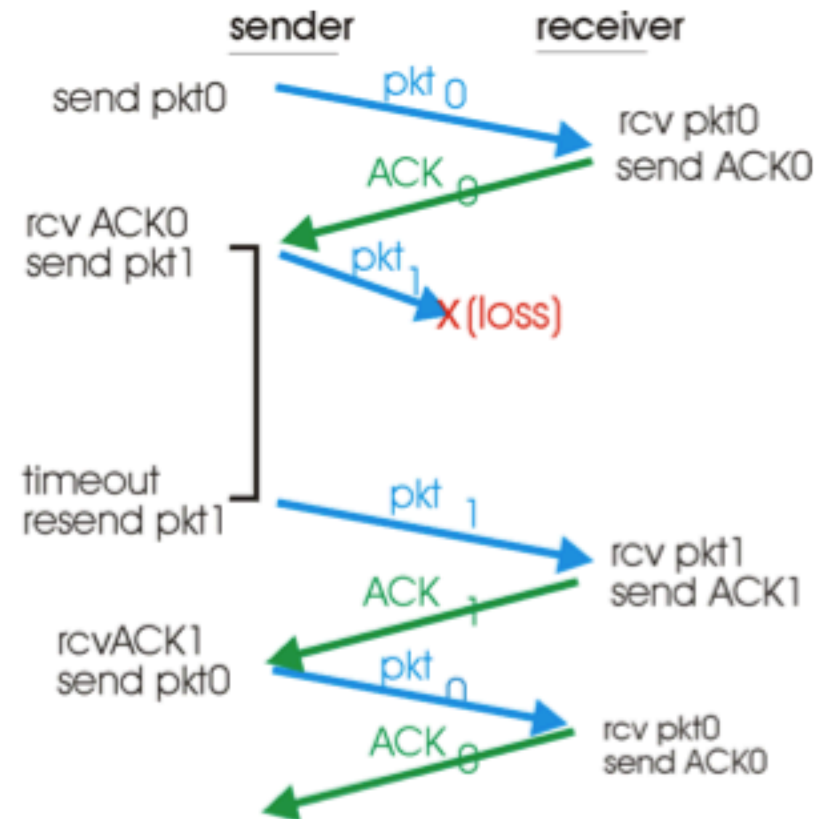
rdt 3.0, para canal ruidoso y pérdidas

¿y el receptor?

rdt 3.0 con diagrama de flechas

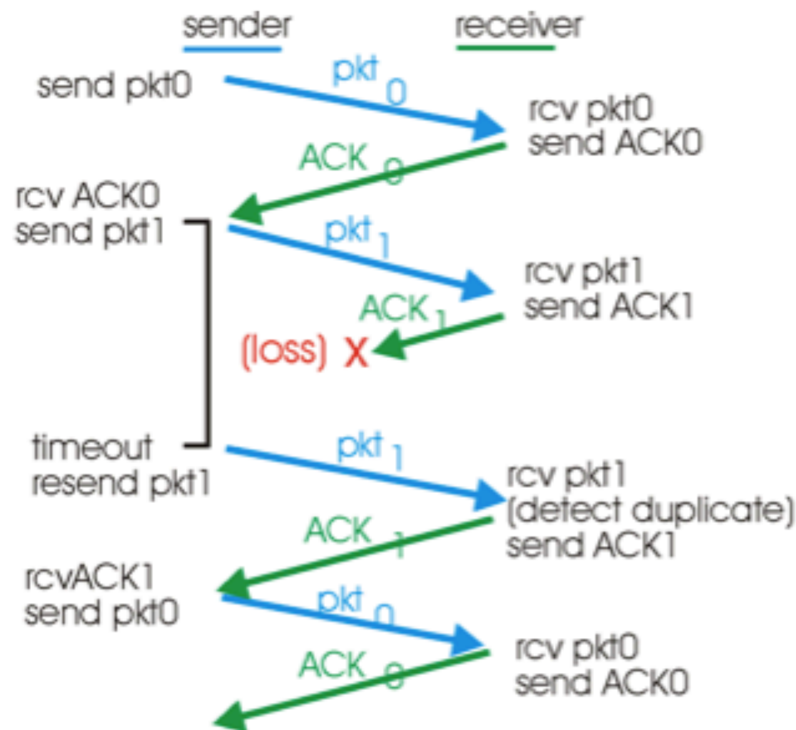


(a) operation with no loss

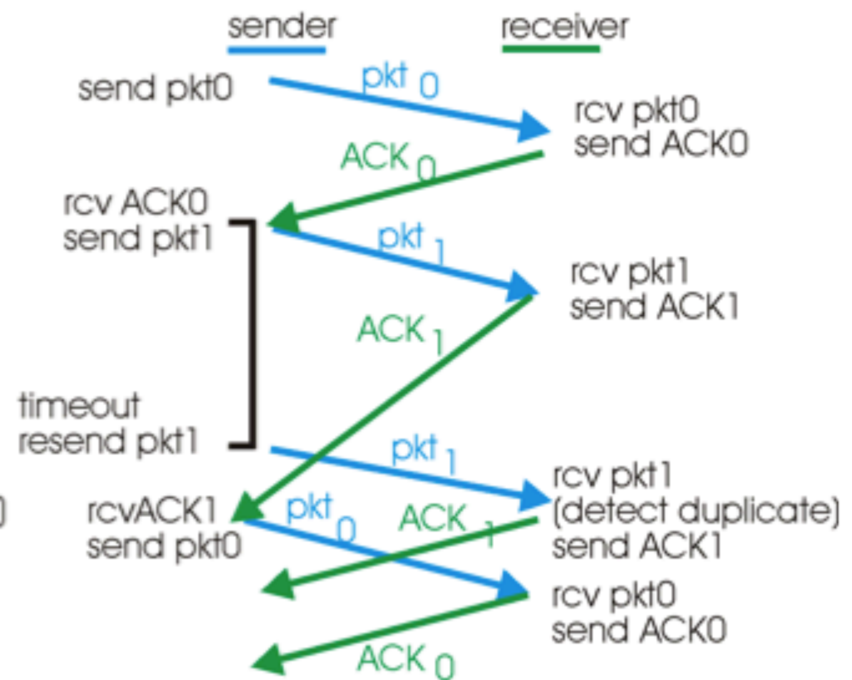


(b) lost packet

rdt 3.0 con diagrama de flechas



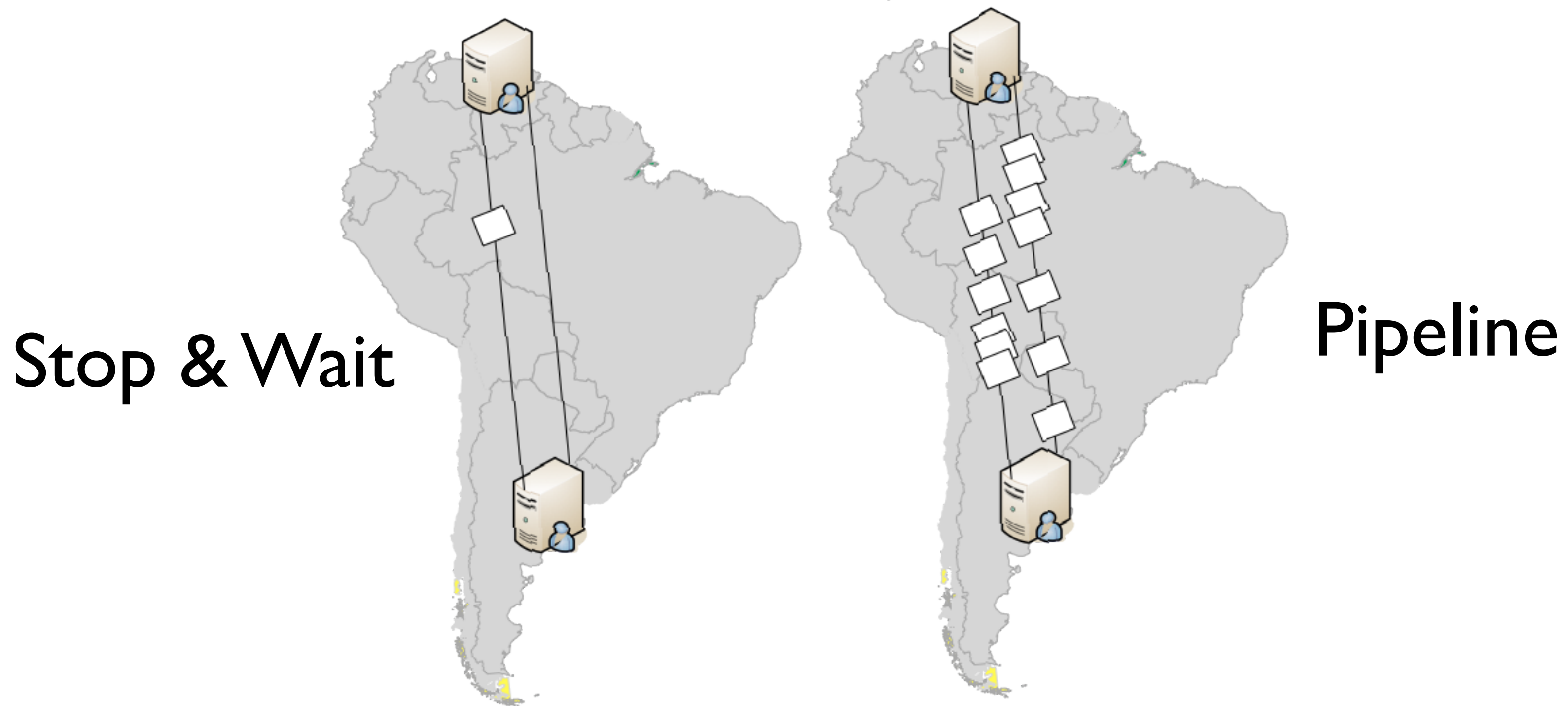
(c) lost ACK



(d) premature timeout

Rendimiento de rdt 3.0

- rdt 3.0 es un protocolo stop-&-wait → **no muy eficiente que digamos.**
- Suponga un servidor en Mérida-Vnzla y el otro en Buenos Aires-Argentina:



Rendimiento de rdt 3.0

- Supongamos que en RTT es de 30 ms, la tasa de transferencia R es de 1 Gbps, el tamaño del paquete es de L=1000 bytes.
- Tiempo de transmisión del paquete:

$$d_{trans} = \frac{L}{R} = \frac{8000\ bits/packet}{10^9\ bits/sec} = 8\mu s$$

- Utilización del canal:

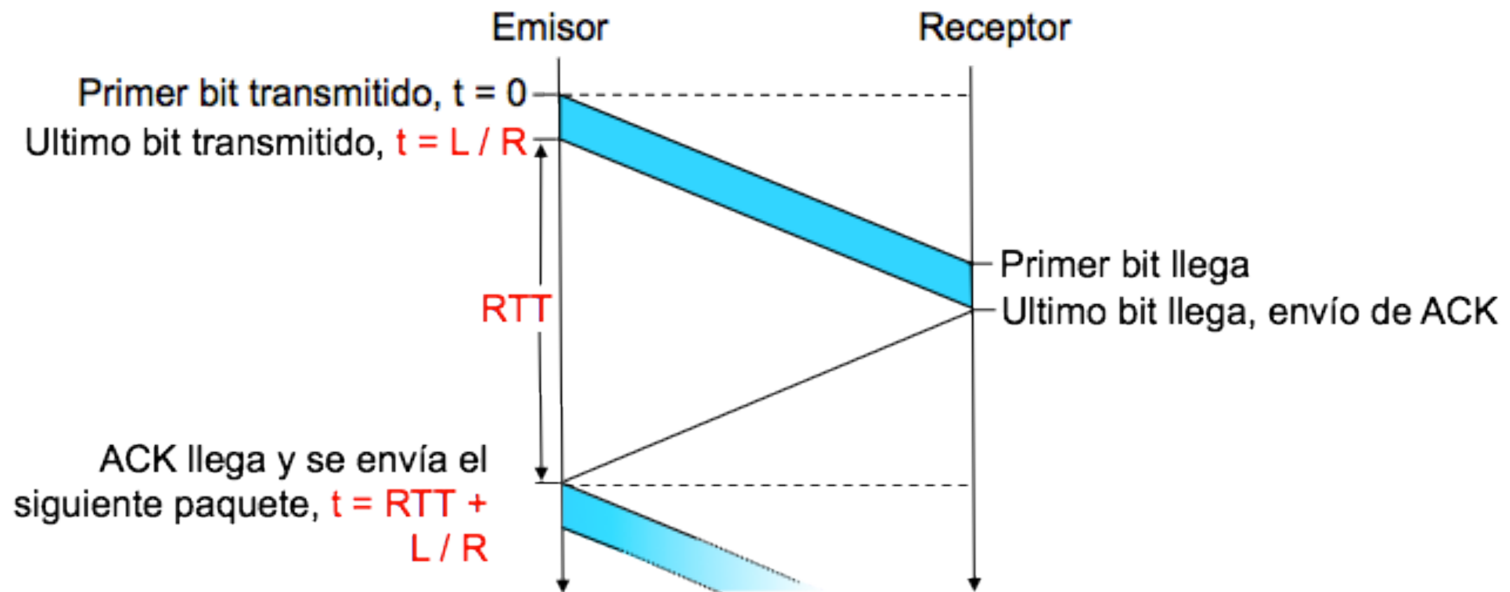
$$U_{sender} = \frac{L/R}{RTT+L/R} = \frac{0.008}{30.008} = 0.00027$$

¿Cómo aumentar el rendimiento?

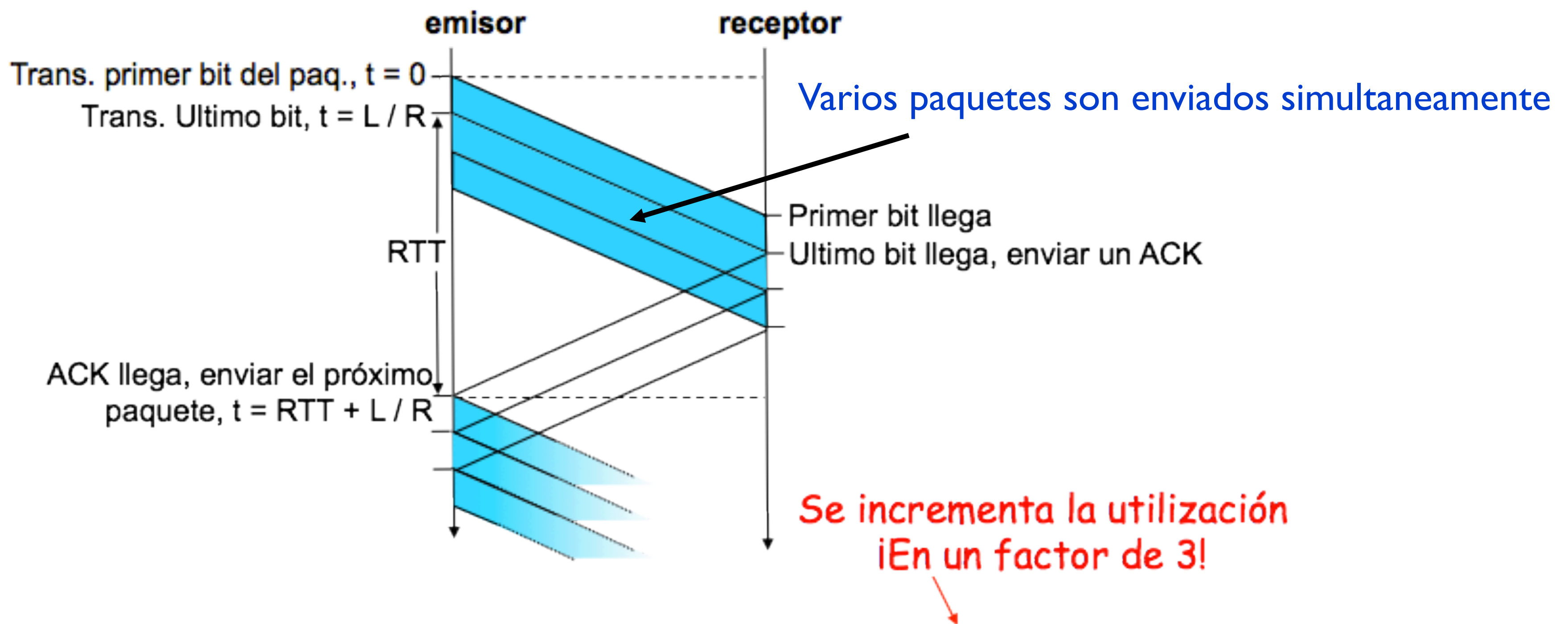
- Envío de múltiples paquetes al mismo tiempo ⟶ hacer **pipelining** (entubamiento).
- Consecuencias:
  - Se requieren más números de secuencia.
  - Buferisación debe ser más grande (tanto cliente como servidor).
  - Capacidad en buffer depende del método de retransmisión.

Recuperación de Errores

- Go-Back-N

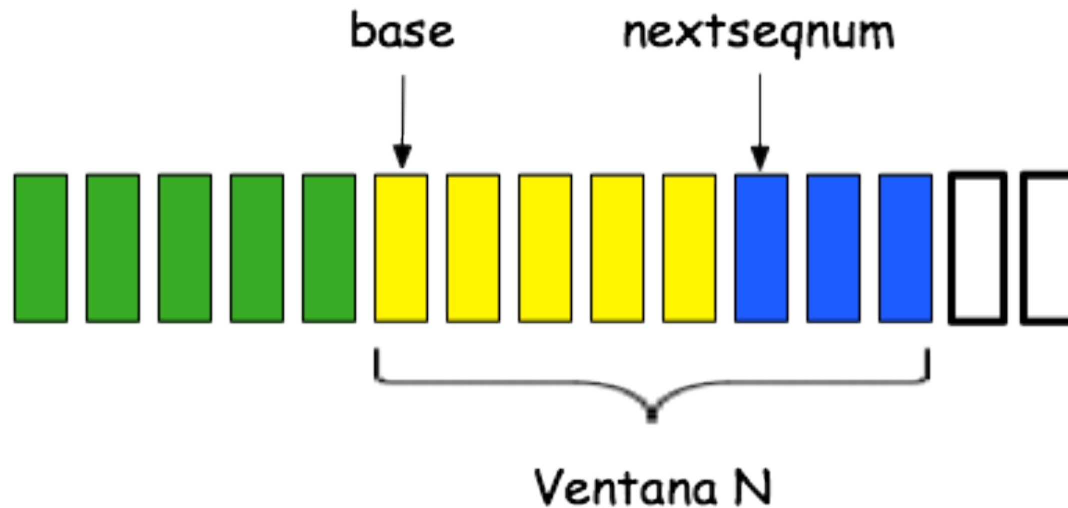


Entubamiento (pipelining)



$$U_{sender} = \frac{3 * L / R}{RTT + L / R} = \frac{.024}{30.008} = 0.0008$$

Ventana de Transmisión



Enviados y bien rcb.

Permitidos para enviar pero esperando

Enviados y no ACK

Bloqueados

Go-Back-N

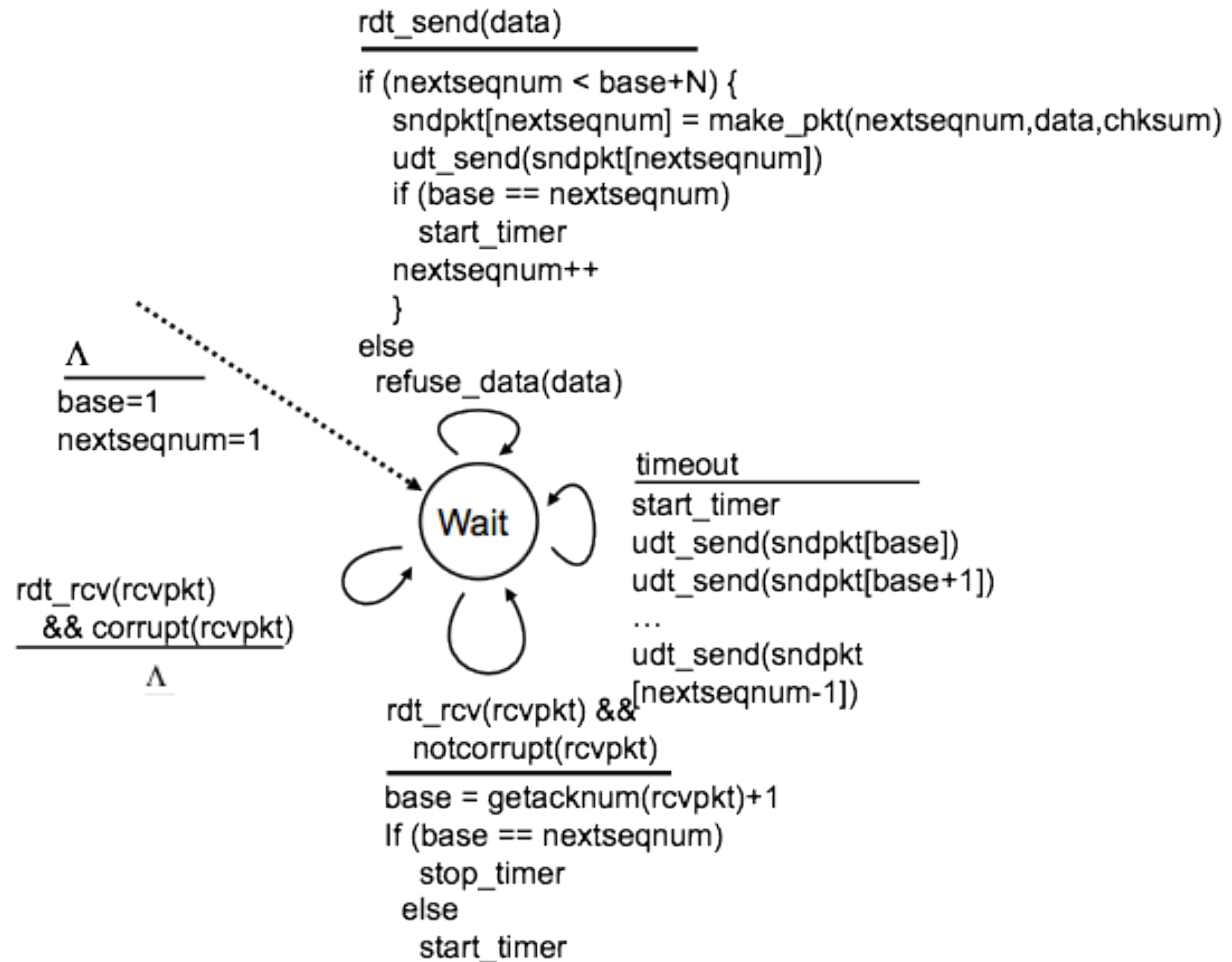
- Restringido a máximo N paquetes en transito.
- 4 intervalos en la secuencia
  - 1.- [0, base-1], 2.- [base, nextseqnum-1], 3.- [nextseqnum, base+N-1] 4.- [base+N, ∞].
- Protocolo de "ventana deslizante" restringido a tamaño de ventana "N".
- Número de secuencia formado con K bits genera números entre $[0,2^k-1]$.
  - Se puede ver como un anillo de 2^k números. Se solapa de 2^k a 0.

Comparación

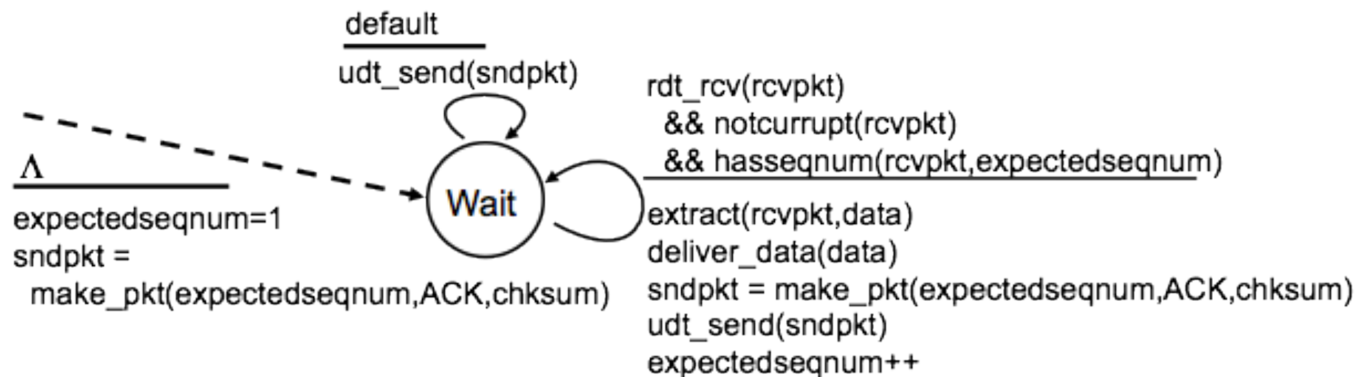
Go-back-N	Selective Repeat
Hasta N paquetes sin ACK	Hasta N paquetes sin ACK
Envío de ACK acumulativo.	Envío de ACK por paquete recibido
• Timer para el paquete más antiguo. • Retransmisión de la ventana completa.	• Timer por paquete. • Retransmisión de 1 paquete a la vez.

Extensión para Go-Back-N

Emisor



Extensión para Go-Back-N



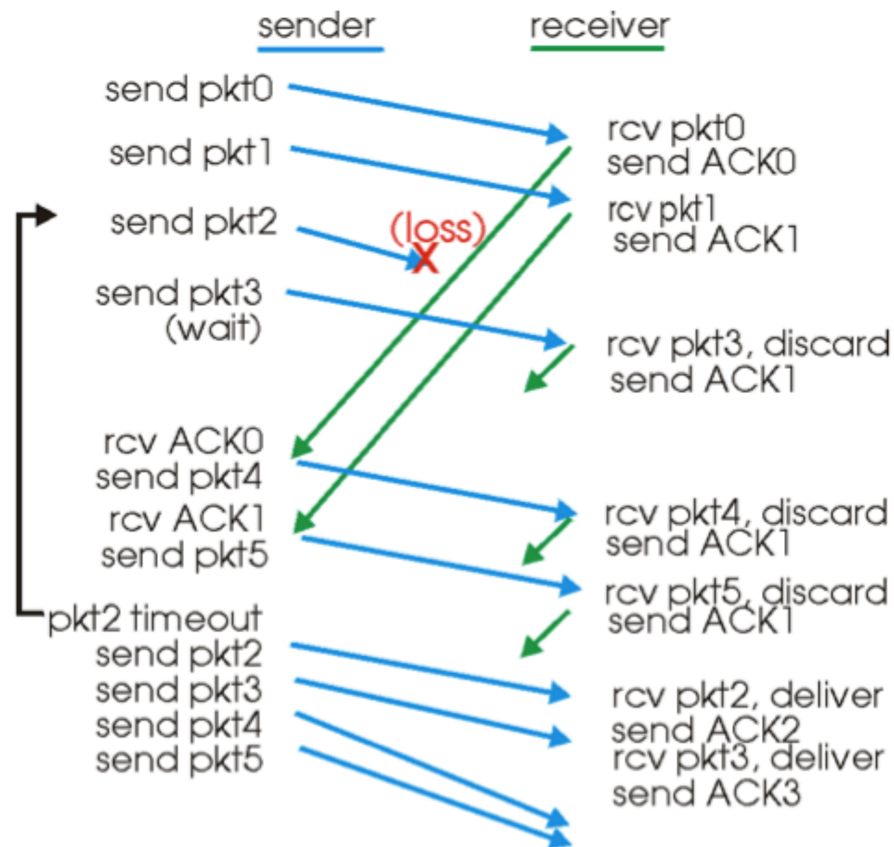
receptor

- Siempre envia ACKs con el número de secuencia recibido más alto.
- Puede generar ACKs duplicados.
- Debe recordar el número de secuencia esperado.
- Si hay paquete fuera de orden:
  - 1) descartar 2) enviar un ACK con el número de secuencia más alto.

Tipos de Eventos

- Invocación desde arriba
  - Si la ventana no está llena ⟶ enviar paquete.
  - Si no, indicar que no hay capacidad de envio (excepción)
- Recepción de un ACK
  - Tipo acumulativo
- Timeout
  - Se aplica la politica: Ir N paquetes hacia atras, como máximo y reenviar
  - Note que se puede **empeorar** la transmisión, pues retransmite paquetes que probablemente ya llegaron.

Ejemplo



Resumen

- Go-Back-N utiliza casi todas las políticas de transmisión confiable:
  - Números de Secuencia
  - ACK acumulativos
  - Checksums
  - Temporizador / Retransmisión

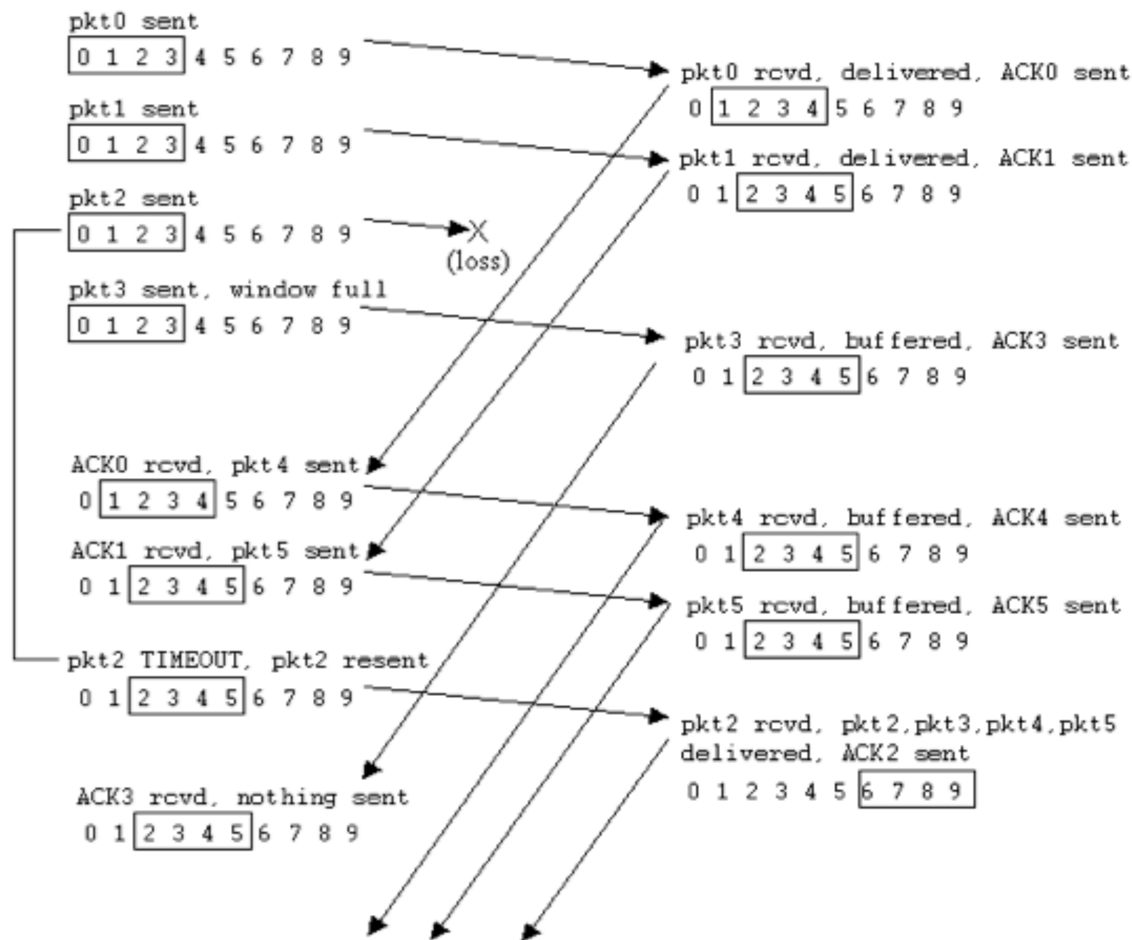
Selective Repeat

- GBN permite potencialmente "llenar el tubo".
- Un solo error en GBN produce retransmisiones "innecesarias".
- El receptor reconoce (envia ACKs) por cada paquete de datos (no envia acumulado)

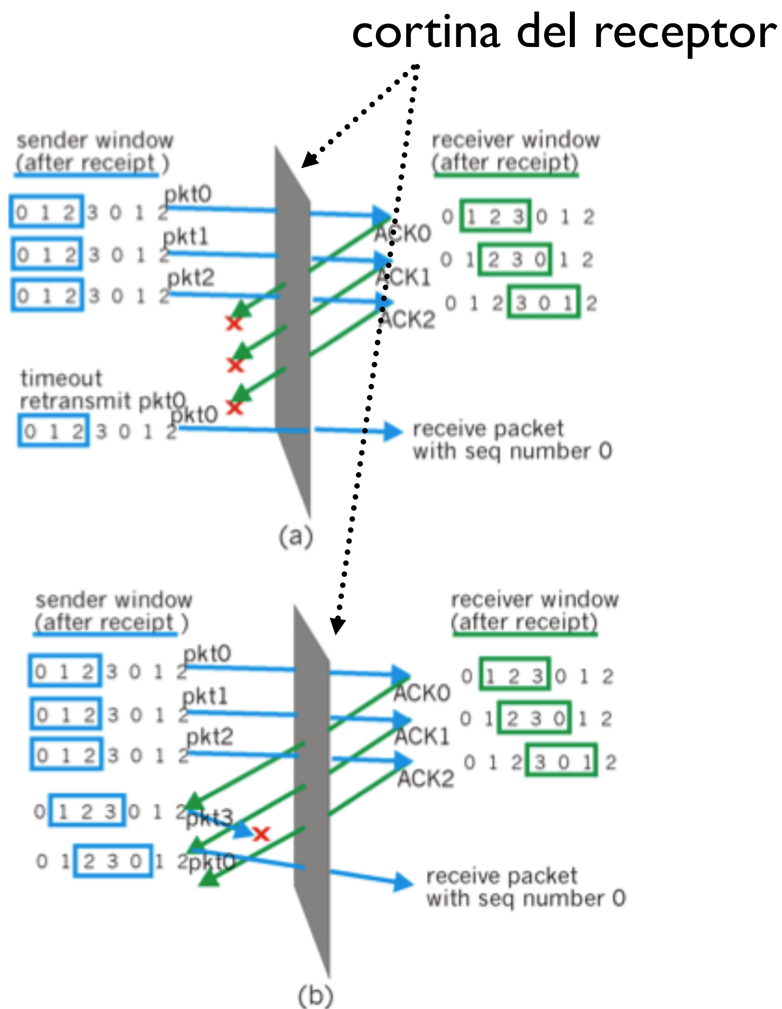
Algoritmos para Selective Repeat

Emisor	Receptor
Data desde arriba: * Si hay disponibilidad en la ventana —› enviar. Timeout (n): * Reenviar el paquete **n** * Reiniciar timer T(n) entre [base,base+N] * Marcar paquete como recibido * Avanzar ventana.	Paquete n en [rcvbase, rcvbase+N-1] * Enviar ACK(n). * <u>fuera de orden</u>: bufferizar * <u>en orden</u>: entregar todo el conjunto * avanzar ventana Paquete n en [rcvbase-N, rcvbase-1] * No Enviar Nada. Por omisión: * Ignorar

Ejemplo de Selective Repeat



Dilema del SR: ¿Cómo diferenciar las pérdidas?

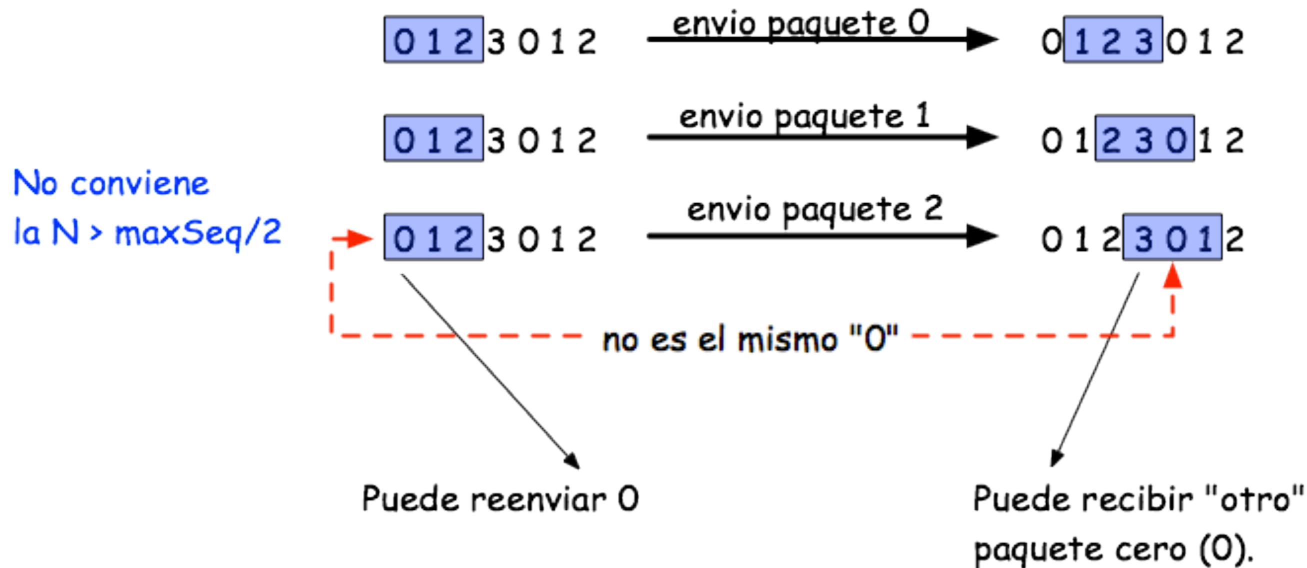


- Desde el **receptor** los 2 escenarios son iguales. (pero, en (a) hay retransmisión incorrecta.)
- No hay manera de distinguir una **retransmisión** de un **nuevo elemento**.

- Condiciones:

- Una ventana ha de ser la mitad o menos del max. número de secuencia para no solapar los bordes.
- Hay **reordenamiento** cuando hay varios saltos entre los 2 nodos.

Solapamiento de Bordes



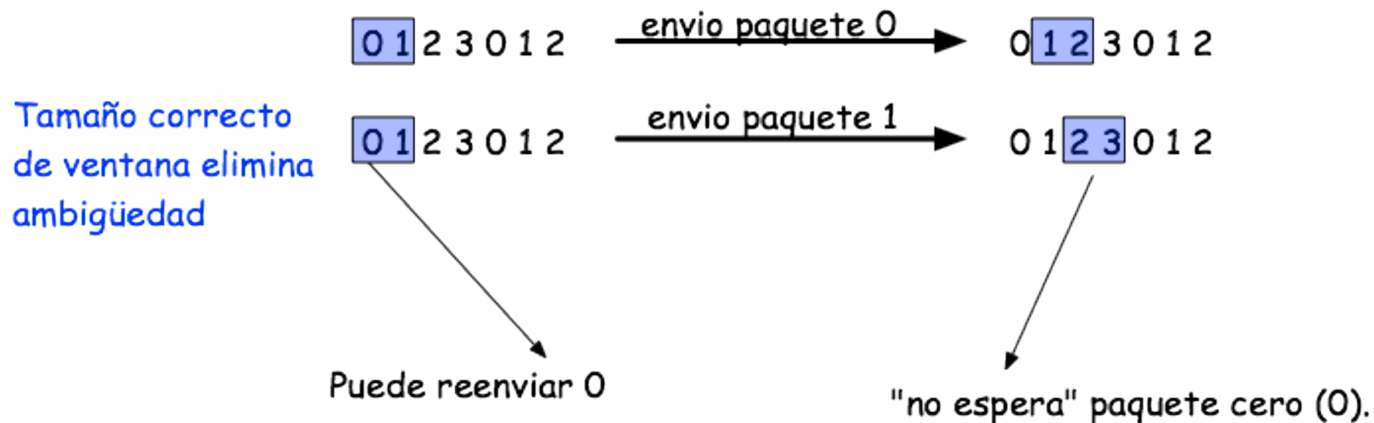


Tabla Resúmen

- El par de protocolos avanzados para transporte utilizan todos los mecanísmos vistos:
  - Checksum
  - Timer
  - Números de Secuencia
  - ACKs
  - NACKs
  - Ventanas/Entubamiento (pipelining)

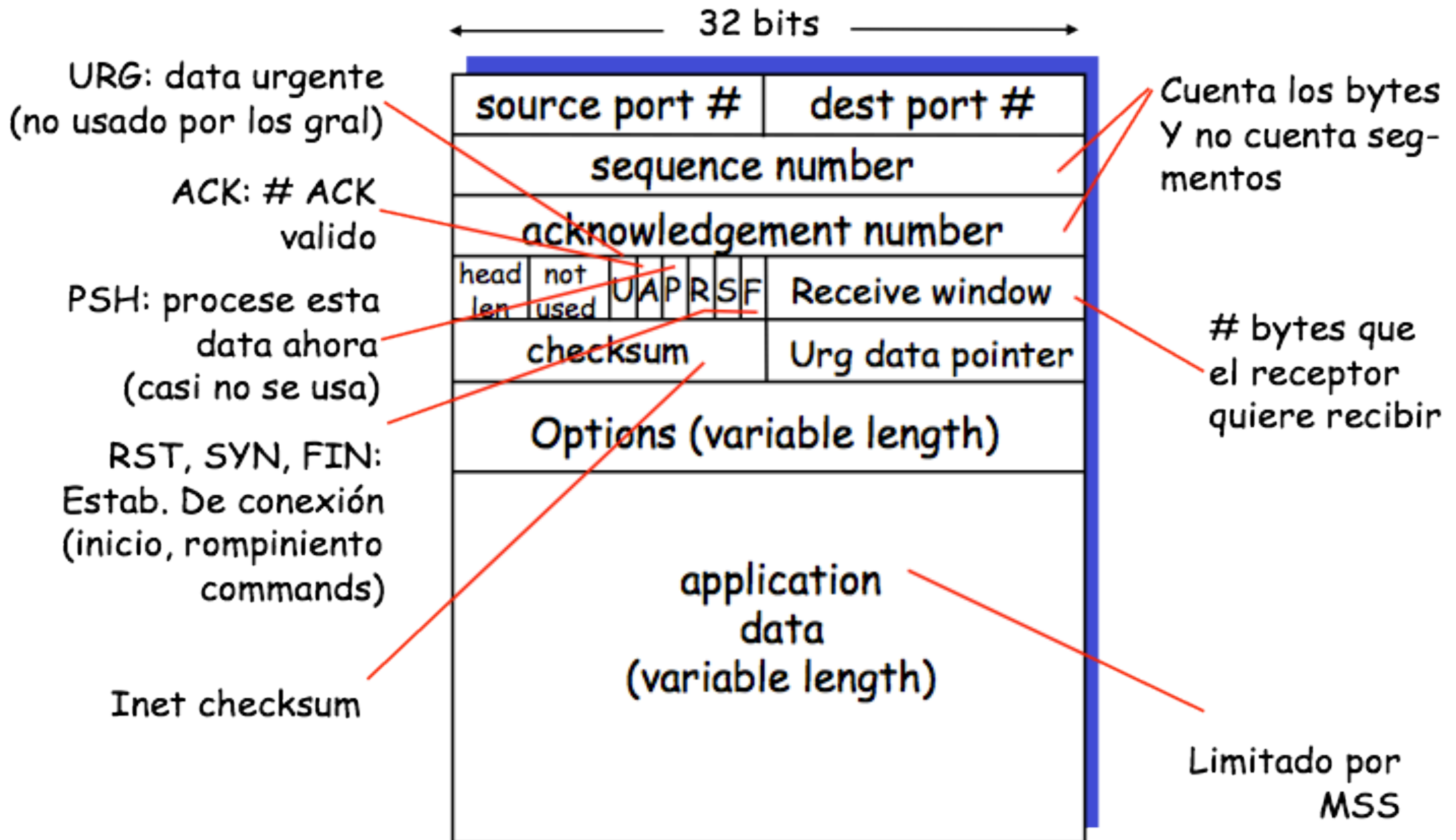
TCP: protocolo orientado a conexión

- **Punto a punto**: un emisor y un receptor
- **Flujo de datos en orden y confiable** (no hay límites en msgs)
- **Entubado** (crtl de *congestión* + ctrl de *flujo* controla tamaño de ventana de transmisión)
- **Buffers** para el envío / recepción

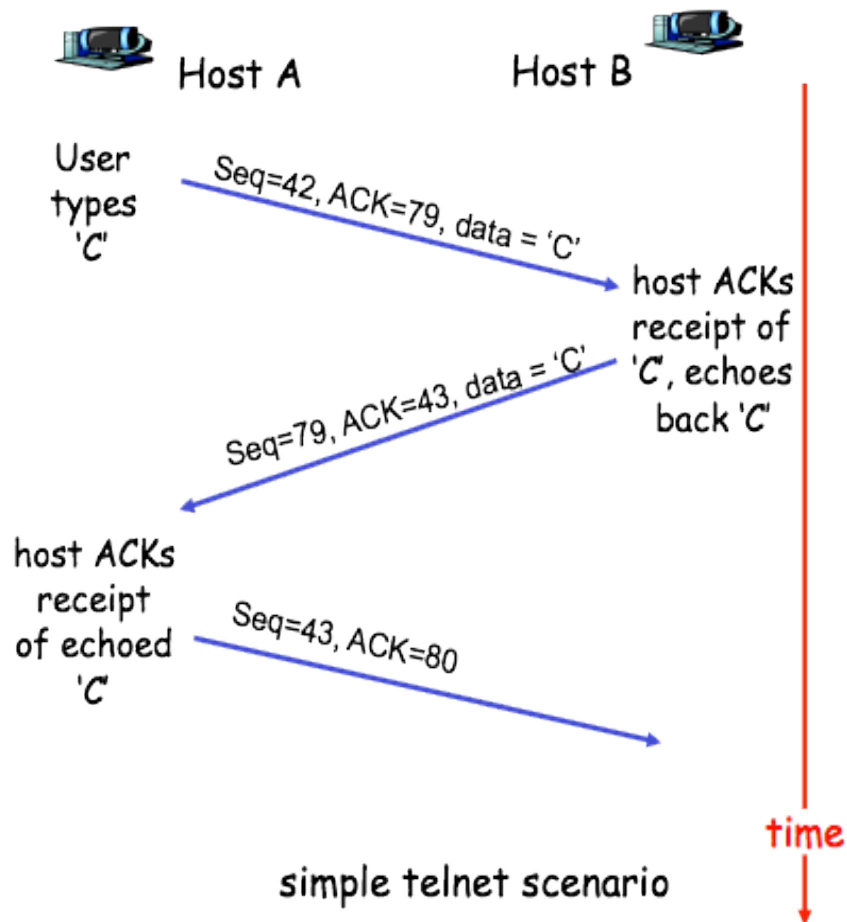
TCP: protocolo orientado a conexión

- **Data transmitida en full duplex**: data bidireccional, depende de talla máxima de paquete transmitible (MSS, Ej. 1460, 536 y 512).
- **Orientado a conexión**: handshaking
- **Control de flujo**: el que envia no satura al receptor.

Estructura del Segmento



Números de Secuencia y número de ACKs



Seq. Número del primer byte de la cadena.

ACK. Próximo byte esperado en la cadena (acumulativo)

Preg: y ¿Cómo se manejan los segmentos fuera de orden?
R: decisión libre del implementador

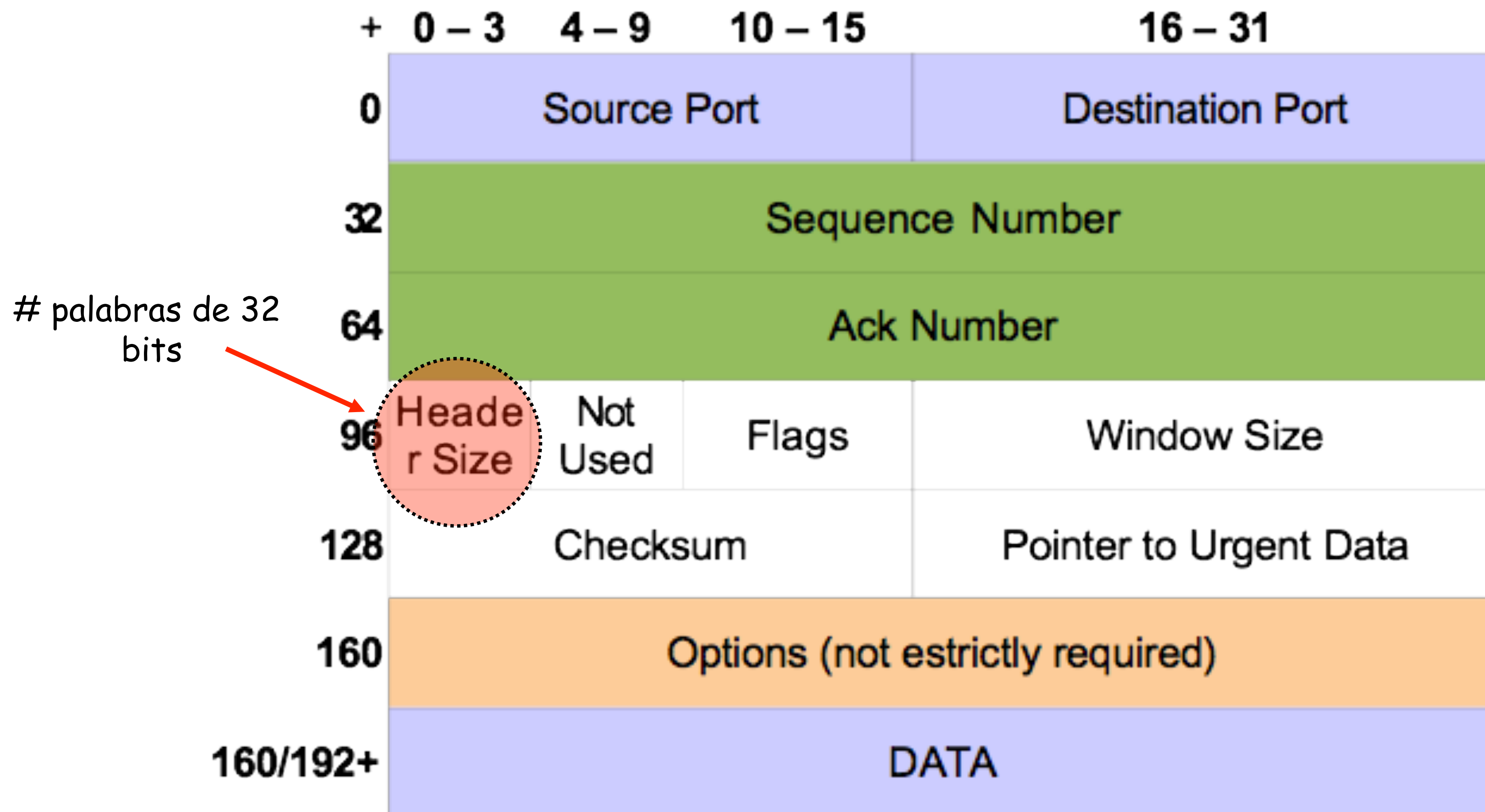
Detalles de la Conexión

- Se puede abrir una conexión entre un cliente y un servidor:
  - Socket clientsocket = new Socket("hostname", portNumber);
- Primeros 2 segmentos no llevan datos (SYN, SYN +ACK), el tercero **podría** llevar data (3-way-handshake).
- Cuando la conexión se establece se puede empezar a enviar data entre los 2 extremos.
- Se busca la trama más grande de la ruta en la capa enlace (MTU) y se calcula el tamaño del segmento más grande MSS (maximum segment size).

Detalles de la conexión

- **Cada lado** de la conexión tiene su propio **buffer** ⟶ de envio y de recepción.
- El emisor no puede innudar al receptor ⟶ receptor anuncia su capacidad máxima del buffer.
- Cuando se envia todo un archivo todos los paquetes son de tamaño MSS **menos** el último que suele ser más pequeño (¿puede explicar por qué?).

Encabezado TCP (2)

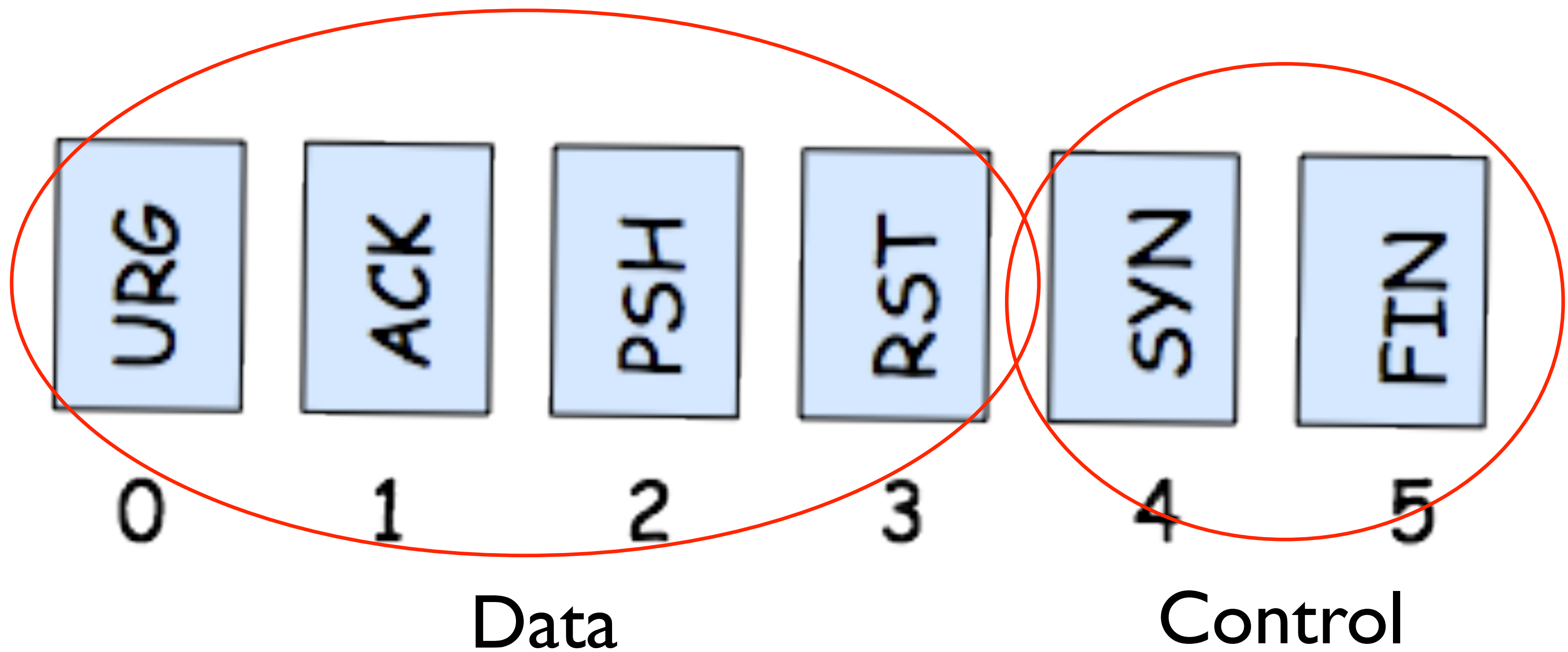


Opciones TCP

- El campo opciones de TCP se puede usar, entre otras opciones :
  - Negociar la talla máxima de segmento (MSS)
  - Escala de la ventana (para redes de alta velocidad)
  - Estampillas temporales
  - Negociar la proporción de ACKs en el control de congestión (RFC 5690)

¿Cuantos bytes máximo son permitidos en el espacio de opciones de un Paquete TCP?

Bits del encabezado



Nros. de secuencia & ACKs

- TCP ve todos los datos **no estructurados** pero ordenados en forma de bytes.
  - Además TCP es **full-duplex**
- Ej: **Archivo** de 500.000 bytes, **segm** 1000 bytes: primer byte numerado 0, primer byte siguente paquete numerado 1000 ...
- **Un ACK** corresponde al próximo byte esperado por el receptor.
- TCP provee ACKs hasta el último byte recibido.
  - Los ACKs son **acumulativos**

Paquetes fuera de secuencia

- Hay dos opciones para tratar **paquetes fuera de secuencia**:
  1. Descartarlos ⟶ hace el diseño del receptor más simple
  2. Guardarlos y solicitar el reenvío de los huecos faltantes ⟶ ¡Lo que hace TCP!
- Números de secuencia aleatorios para:
  - Evitar confundir conexiones reencarnadas.
  - Protegerse contra ataques (SYN cookies).

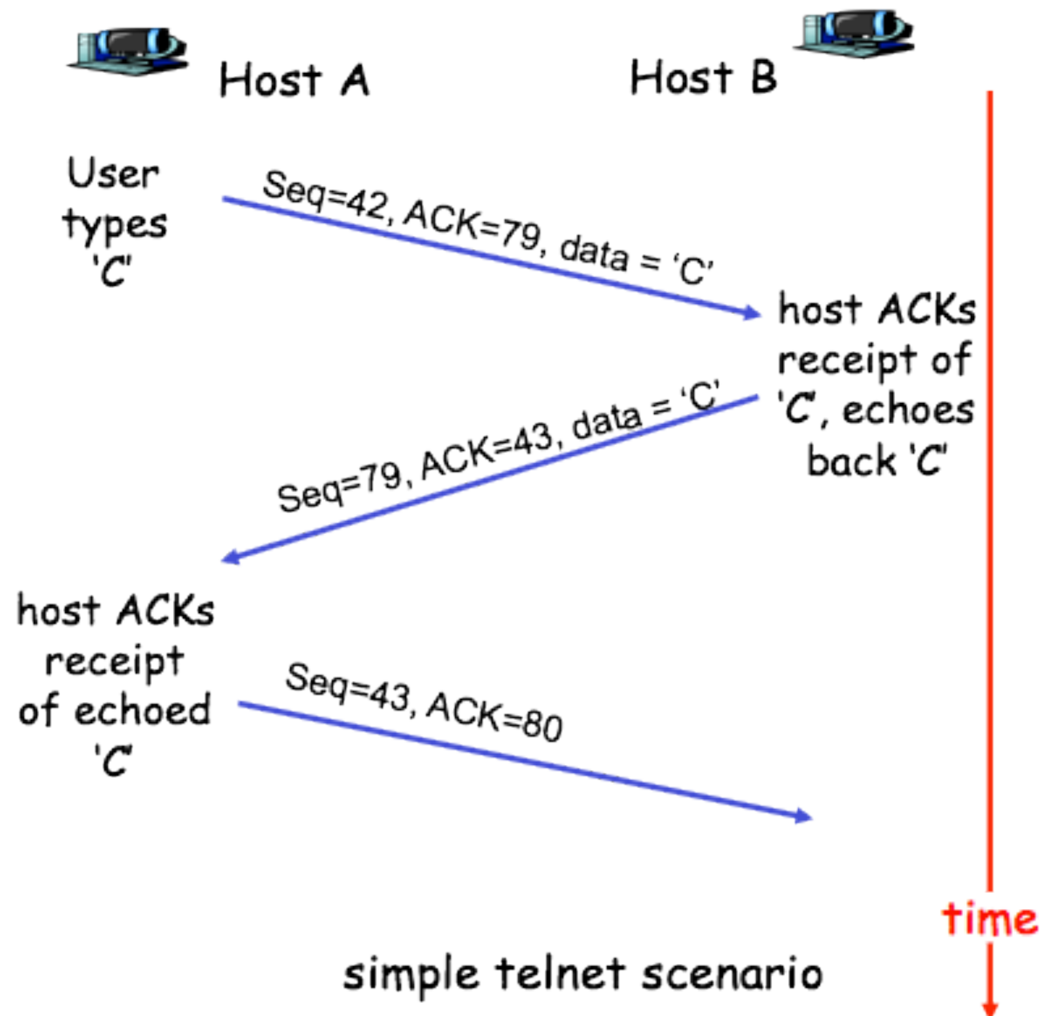
SYN Cookie

- Servidor no sabe si SYN es legitimo o no…
  - Servidor crea una un número de secuencia con una función hash compleja; F_h(SrcPort,DstPort,SrcAd,DstAd, SecretNumber) para el SYN+ACK
  - Servidor no guarda estado
- Cliente responde SYN+ACK+1
- Servidor vuelve a calcular el mismo número de secuencia que debe ser igual al anterior.
  - El servidor no ha guardado estatus y el cliente ha de ser LEGITIMO.

Caso de Estudio: TELNET

- También corresponde al caso de SSH
- Protocolo capa aplicación para **operación remota.**
- Ilustra bien el uso de números de secuencia y ACKs.
- Se envía un caracter al **srv** y éste lo envía de vuelta (*eco de retorno*) ⟶ indica que ha sido procesado.
- Como la comunicación es full-duplex los ACKs se pueden *cargar a cuestas (piggybacking)*.

Ejemplo Telnet



Estimación del Round Trip Time (RTT) -- (1/3)

- Conceptualmente sencillo pero hay sutilezas en el cálculo del mécanismo de timeout/retransmisión
- Pregunta obvia: ¿Cuanto espero?
  - Respuesta corta: más que 1 (un) RTT
  - Respuesta larga: RFC 2988 o *Jacobson, V. Congestion Avoidance and Control. In Proceedings of SIGCOMM '88 (August 1988), (Stanford, CA) ACM.*

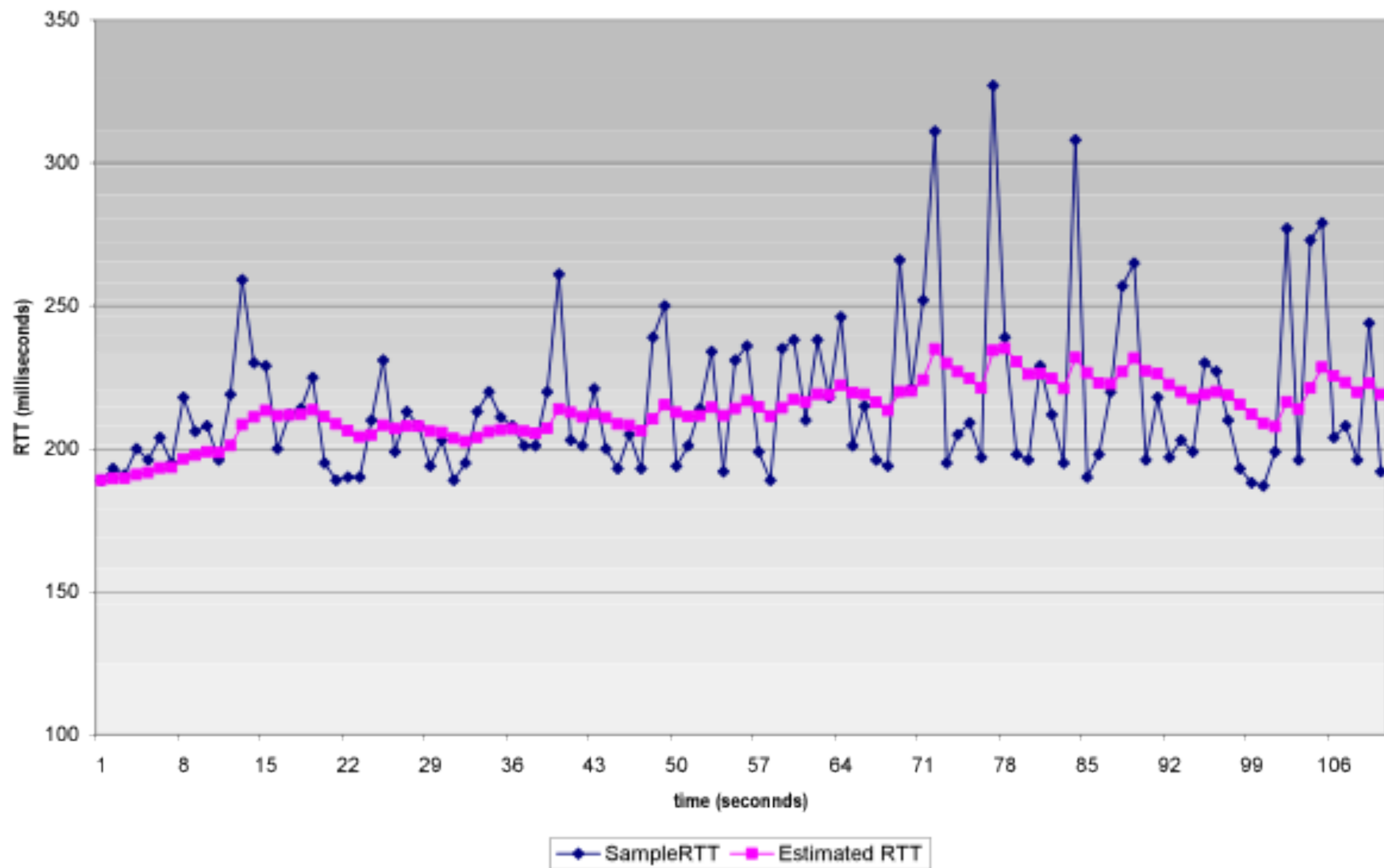
Estimación del RTT (2/3)

- **Muestra del RTT**: se toma cuando se recibe un ACK (una vez por RTT).
- TCP mantiene un promedio: `EstimatedRTT`
  - `EstimatedRTT = (1-` α `) EstimatedRTT +` α `.SampleRTT`
  - α `= 0.125 (1/8) según RFC 2988`
- **Premisa de diseño**: más peso a viejos valores que a los nuevos.

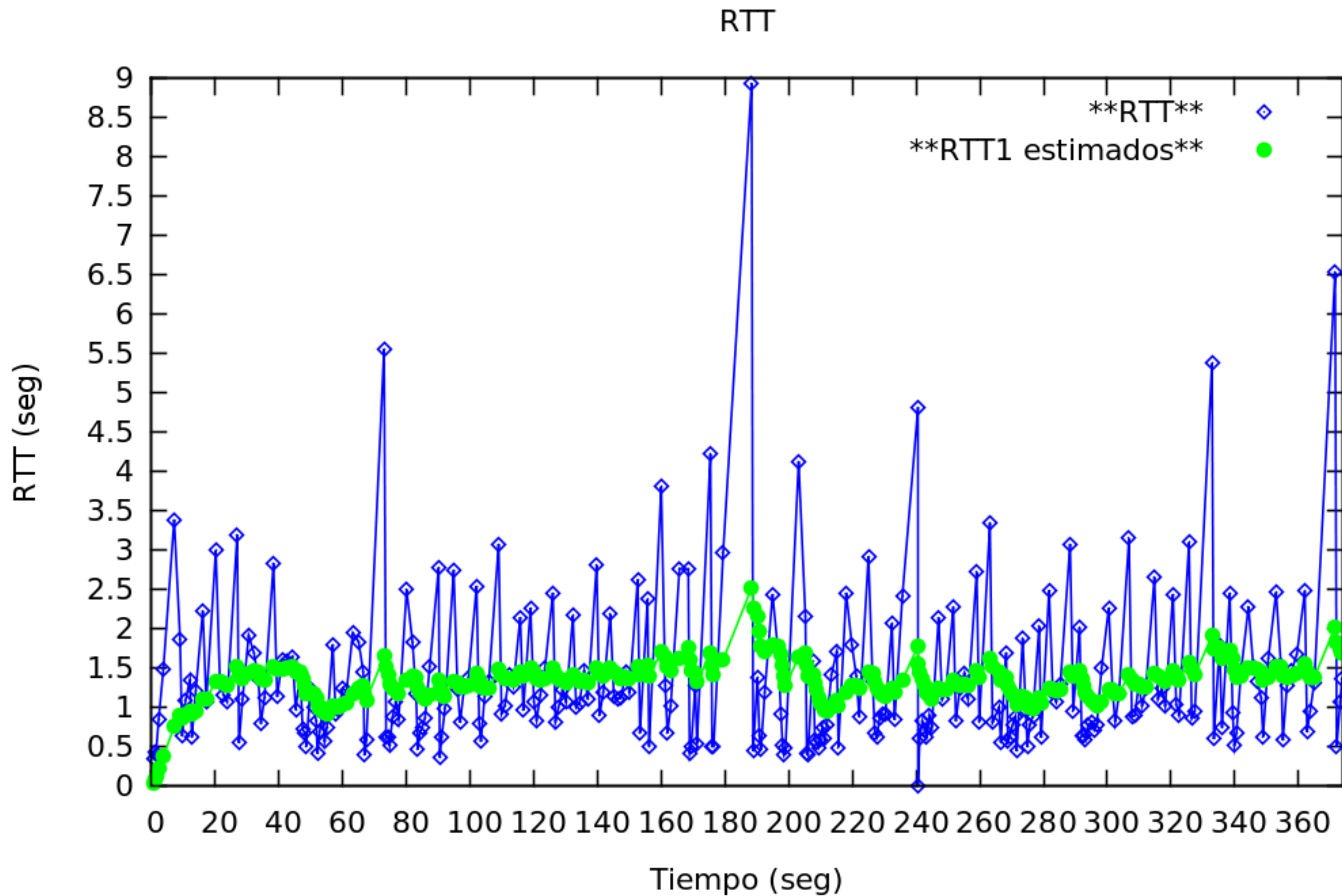
Estimación RTT (3/3)

- El método estadistico para `EstimatedRTT` ⟶ Exponential Weighted Moving Average (decrece exponencialmente el peso de una muestra).
- También se usa la variabilidad:
  - **`DevRTT = (1-ß)DevRTT + ß.|SampleRTT-EstimatedRTT|`**
  - **`ß=0.25 (recomendado)`**

Ejemplo de RTT

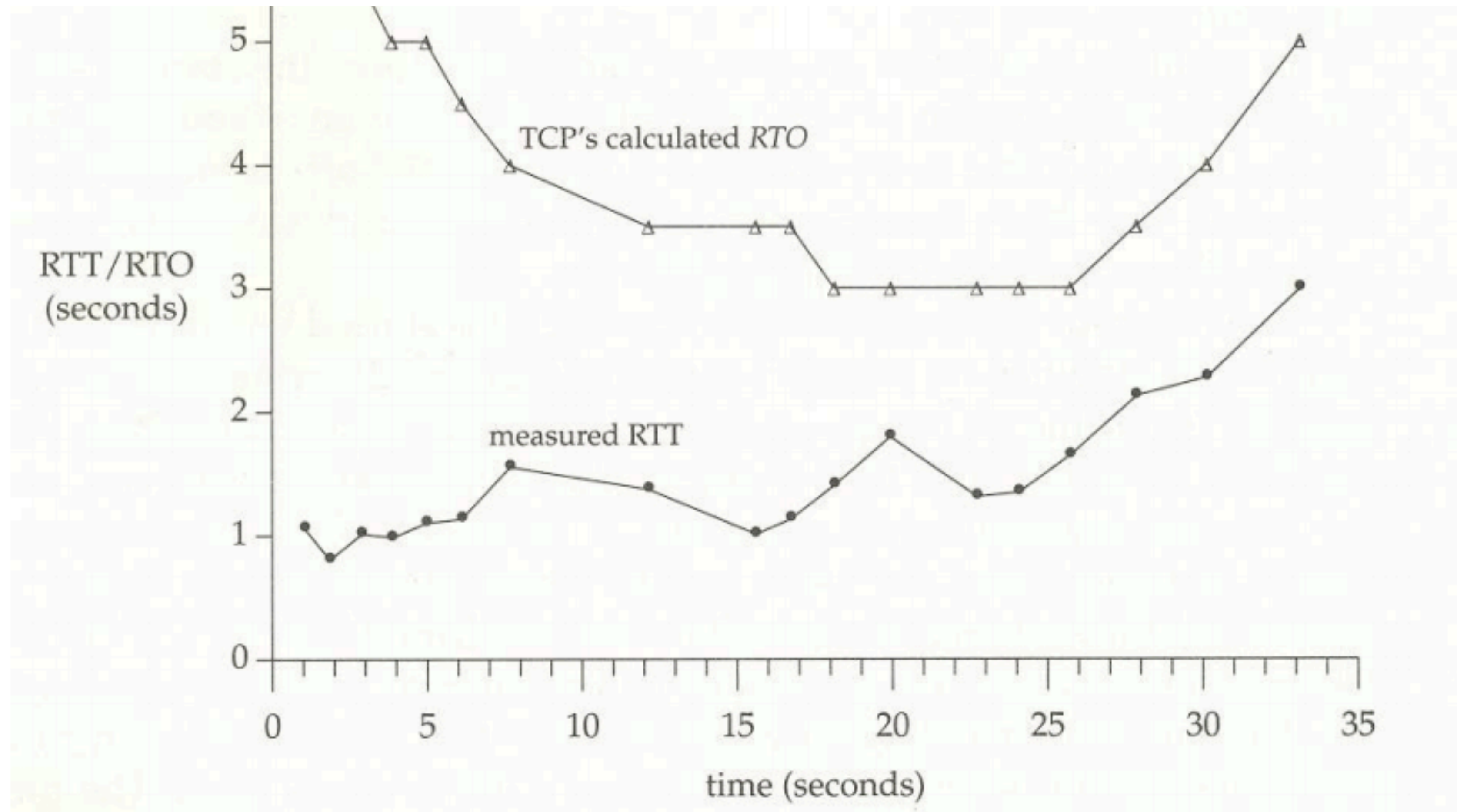


Ejemplo de RTT: de Inter a CANTV



Transferencia de Archivo de 4MB. Tomada de Rincon & Uzcategui, proyecto final, sem A2010.

Dinamica del Timeout



Tomada del R. Stevens, TCP/IP Illustrated Vol I

Pero, ¿Cuanto vale RTO?

- Debe calcularse no mucho mas grande que el EstimatedRTT
- Intuitivamente: grande cuando hay mucha **fluctuación** y pequeño cuando hay poca
- Calculo del timeout:
  - `Timeout = EstimatedRTT + 4 x DevRTT`

Transmisión Confiable

- TCP crea una capa de transmisión confiable encima de IP
- La data que lee el proceso que recibe, debe ser no corrompida, sin huecos, sin duplicado y en secuencia.
  - Principio de Transmisión Confiable: *Garantizar que lo que envio se recibe.*
- Aunque **muchos timers** es buena idea, es pesado para la máquina
  - Se recomienda 1 (un) solo timer para retransmisiones (aún para varios segmentos)

Algoritmo de Transmisión Confiable

- Uso de los ACKs duplicados
- Data enviada en una sola dirección en una transmisión "infinita"
- 3 eventos mayores:
  - Data enviada por la aplicación
  - Timeout
  - Recepción de un ACK
- Timer se asocia con el segmento más viejo no reconocido.

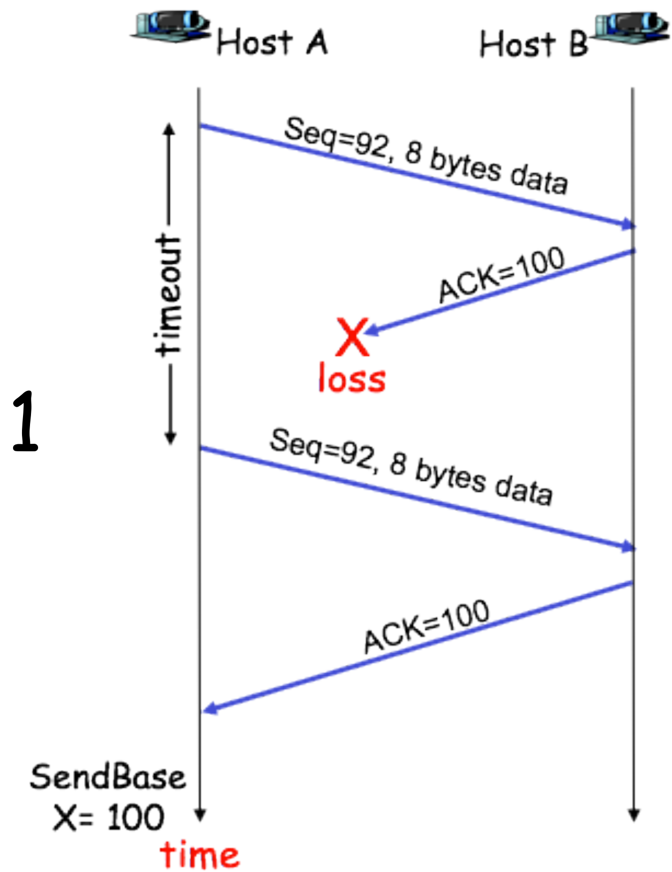
Algoritmo del Emisor

```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

 loop (forever) {
   switch(event)

   event: data received from application above
       create TCP segment with sequence number NextSeqNum
       if (timer currently not running)
           start timer
       pass segment to IP
       NextSeqNum = NextSeqNum + length(data)

    event: timer timeout
       retransmit not-yet-acknowledged segment with
            smallest sequence number
       start timer

    event: ACK received, with ACK field value of y
       if (y > SendBase) {
           SendBase = y
          if (there are currently not-yet-acknowledged segments)
                start timer
          }

  }  /* end of loop forever */
```

- SendBase-1 es el ACK mas recientemente recibido
- Ejemplo:
  - Si SendBase-1 = 71.
  - **y = 73** —› (y > SendBase) entonces la data es nueva.

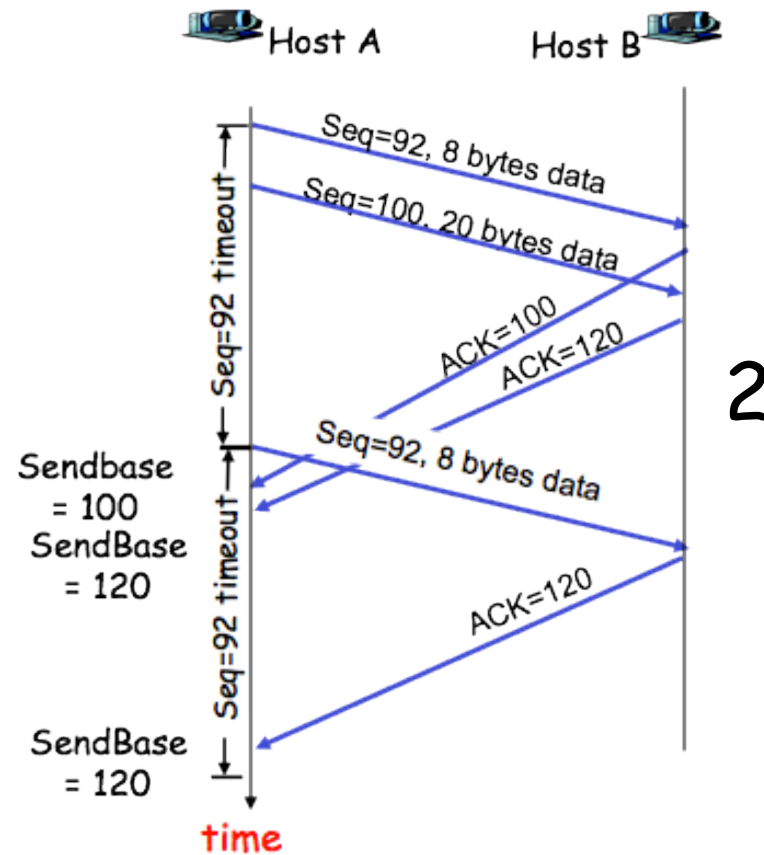
Casos Especiales

1



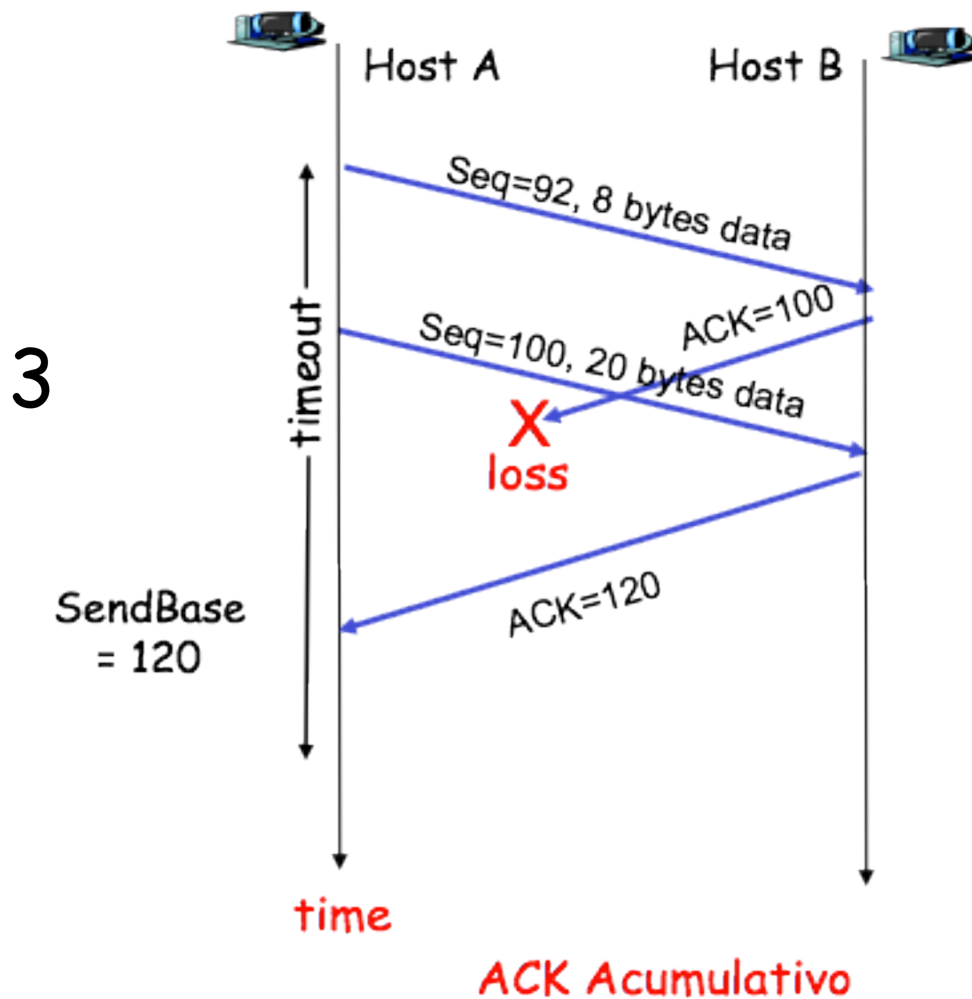
Descarte de Transmisión por Perdida de un ACK

2



Un Timeout prematuro

Casos Especiales



Incremento del RTO

- **Longitud del timeout**: después de una retransmisión por timeout, se dobla el valor.
  - RTO = 0.75 (Timeout) → 1.5 (Timeout) → 3.0 segs.
- Si timer se calcula después de haber recibido un ACK → se toma a partir de la formula con EstimatedRTT & devRTT.
- TCP actua "educadamente" en periodos de congestión.

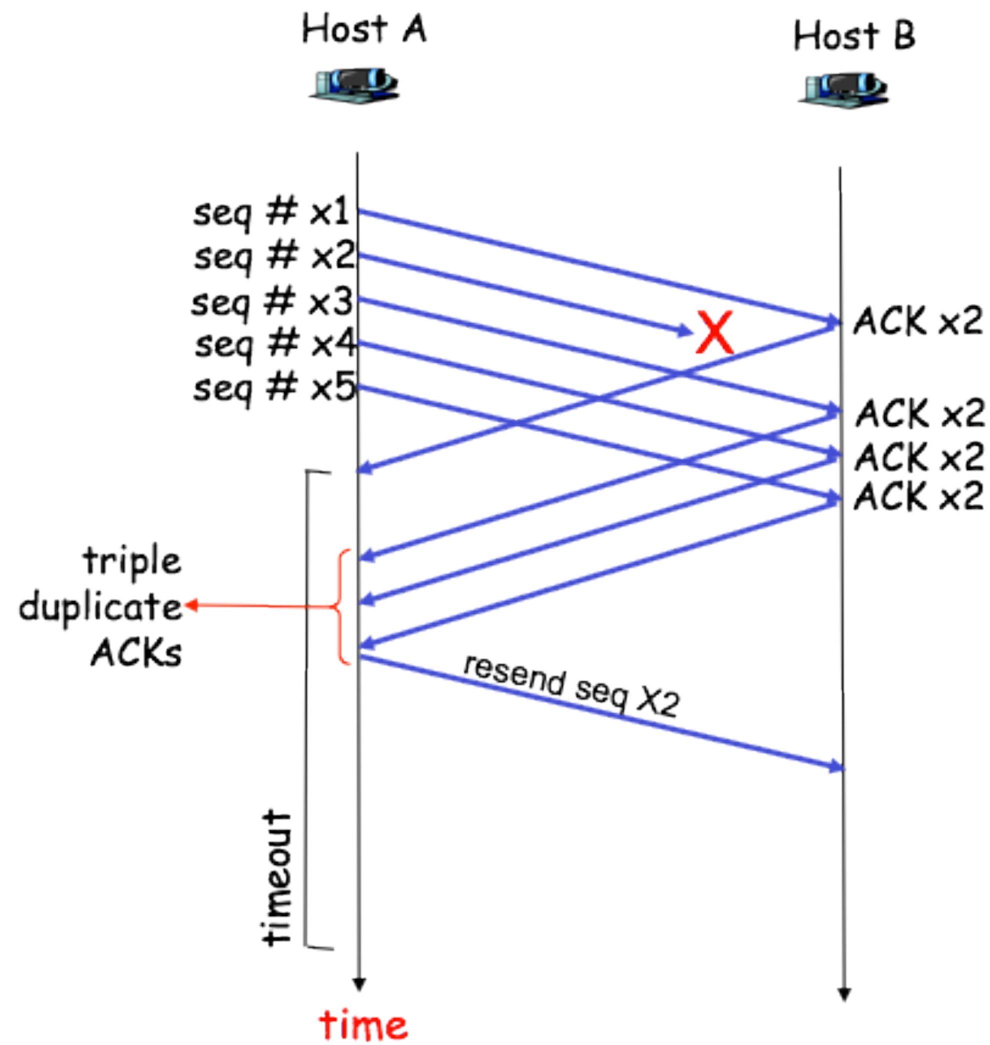
Retransmisión Rapida

- Problema con los timeouts es la **lentitud** en la recuperación.
- El emisor puede **detectar pérdidas** con los ACKs duplicados.
- Debe recibir **3 ACKs duplicados** para enviar el segmento.
- ¿En que condiciones se generan los duplicados?

Reglas de Retransmisión
(RFC 1122, 2581)

Llegada de un segmento en orden	Si es el primero ⟶ esperar hasta 500ms Si es el segundo ⟶ enviar ACK inmediatamente
Llegada de un segmento en DESorden	Enviar un ACK duplicado
Llegada de un segmento que cierra un hueco	Enviar un ACK del segmento final del hueco

Ejemplo retransmisión



Mejora al emisor
Retransmisión Rapida


```
event: ACK received, with ACK field value of y
          if (y > SendBase) {
              SendBase = y
              if (there are currently not-yet-acknowledged segments)
                  start timer
            }
          else {
               increment count of dup ACKs received for y
               if (count of dup ACKs received for y = 3) {
                   resend segment with sequence number y
               }
```

Recepción de un ACK duplicado

Retransmisión rápida

¿GBN o Select. Repeat para TCP?

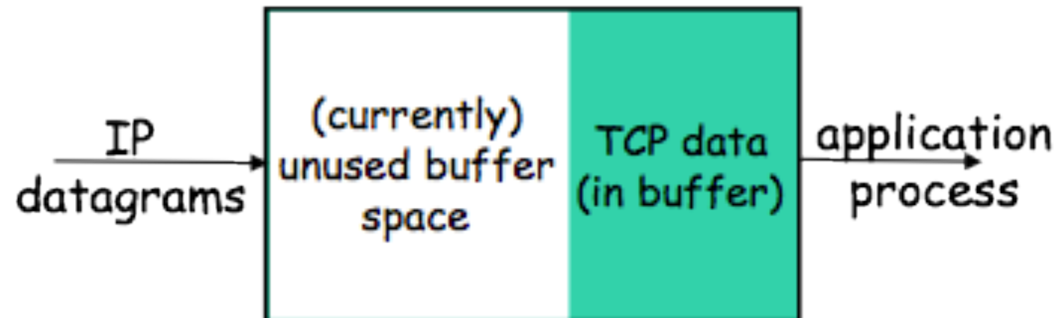
- ¿Qué hace TCP, Go-Back-N o Selective Repeat?
  - TCP mantiene el # de secuencia reconocido más pequeño (sendbase y nextseqnum)
    - Se parecen **pero** GBN no guarda los números fuera de orden
    - GBN retransmite **toda** la ventana, TCP solo el paquete indicado
- Si se usa SACK, TCP se parecería más a S.R.
  - SACK es aun más eficiente que SR pues usa 1 solo timer.

Control de Flujo

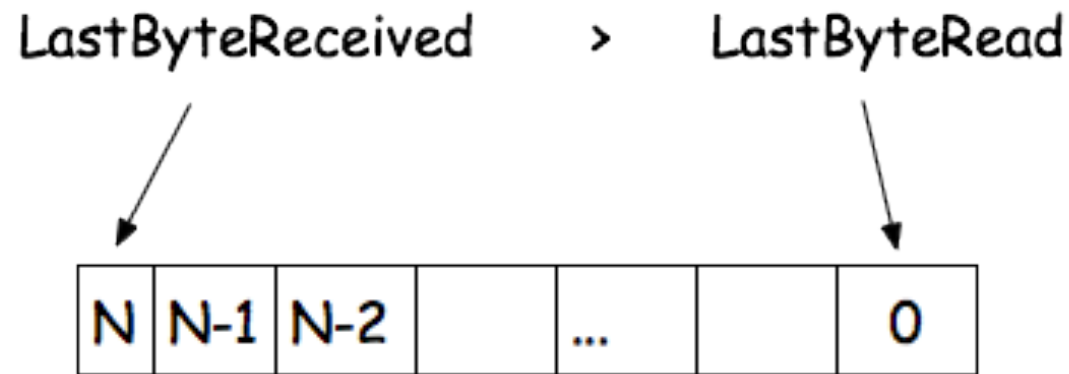
- Cada ente (cliente o servidor) tiene un buffer para almacenar data.
- Si el servidor está ocupado puede no leer la data del buffer ⟶ el emisor podría **saturar** al receptor.
- El control de flujo controla la posibilidad de saturación (es un servicio de control de la velocidad entre la recepción y el procesamiento)
  - Se hace con una **ventana anunciada** llamada **rwnd**

Control de Flujo

- Note bien que control de flujo NO ES control de congestion
- El control de flujo se usa en cada ACK.



Buffer de Recepción



LastByteReceived - LastByteRead ≤ RcvBuffer

rwnd = RcvBuffer - (LastByteReceived-LastByteRead)

Del lado del cliente

- Cuidar que:
  - LastByteSent-LastByteAcked <= rwnd
- ¿Qué sucedería si despues de anunciar **rwnd=0** no hay más ACKs que enviar?
  - El emisor debe "probar" al receptor periodicamente con 1 byte

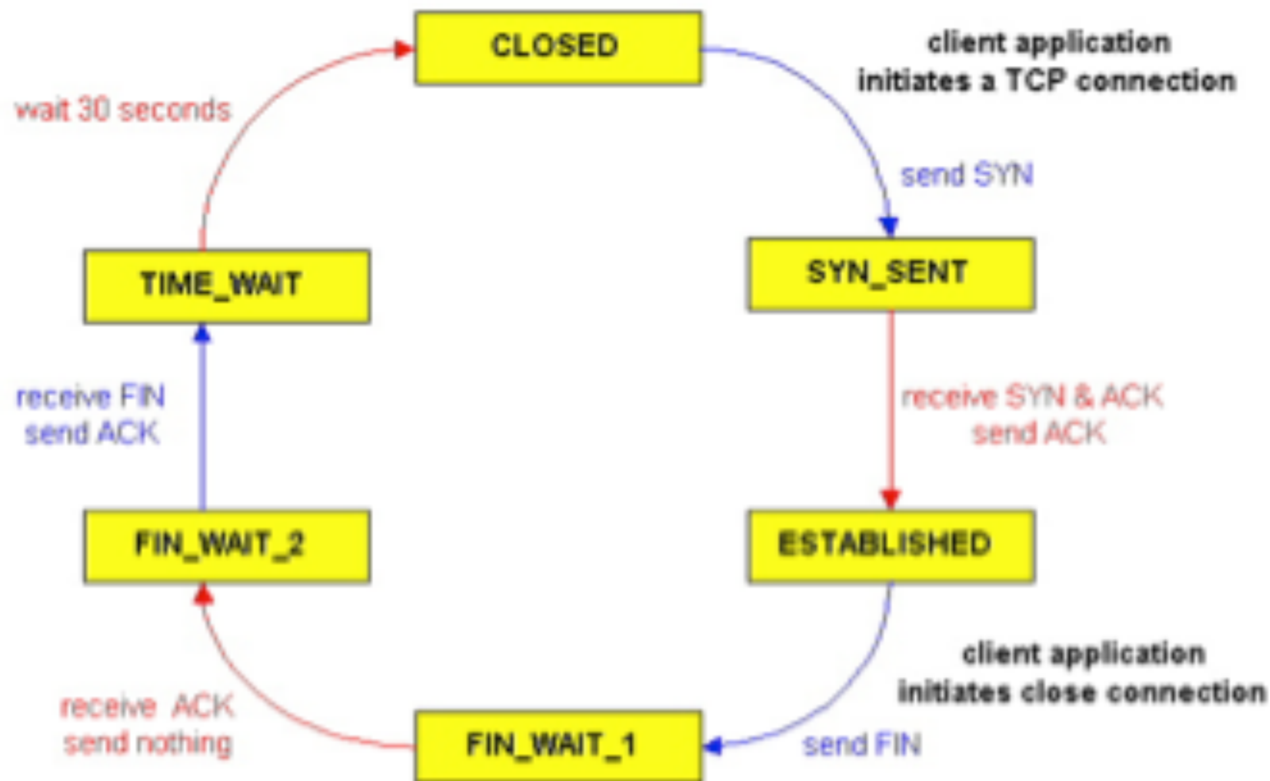
Manejo de Conexión TCP

- Veremos como abrir y cerrar una conexión
  - Influye en el delay (transacciones muy cortas, calidad de servicio)
- Como se establece la conexión
  - Enviar paquete especial al servidor (SYN) y un número aleatorio para la secuencia (client_isn)
  - Si el paquete llega, el ACK es fijado en **client_isn** +1 y se envia el **server_isn** (paquete SYN+ACK).
  - Enviar un reconocimiento con server_isn+1 (SYN=off). En el servidor se apartan buffers y estructuras.

Cierre de Conexión

- Se hace con segmentos especiales que tienen el bit FIN encendido.
- Los recursos son devueltos al SO.
- Cada cierre de cada via en independiente.
- Después del cierre pasan entre 30 seg y 2 min para que todos los **recursos** sean efectivamente devueltos.

Máquina de Estados TCP



Máquina de Estados de TCP