Capítulo 3: Capa Transporte - II

ELO322: Redes de Computadores

Agustín J. González

Este material está basado en:

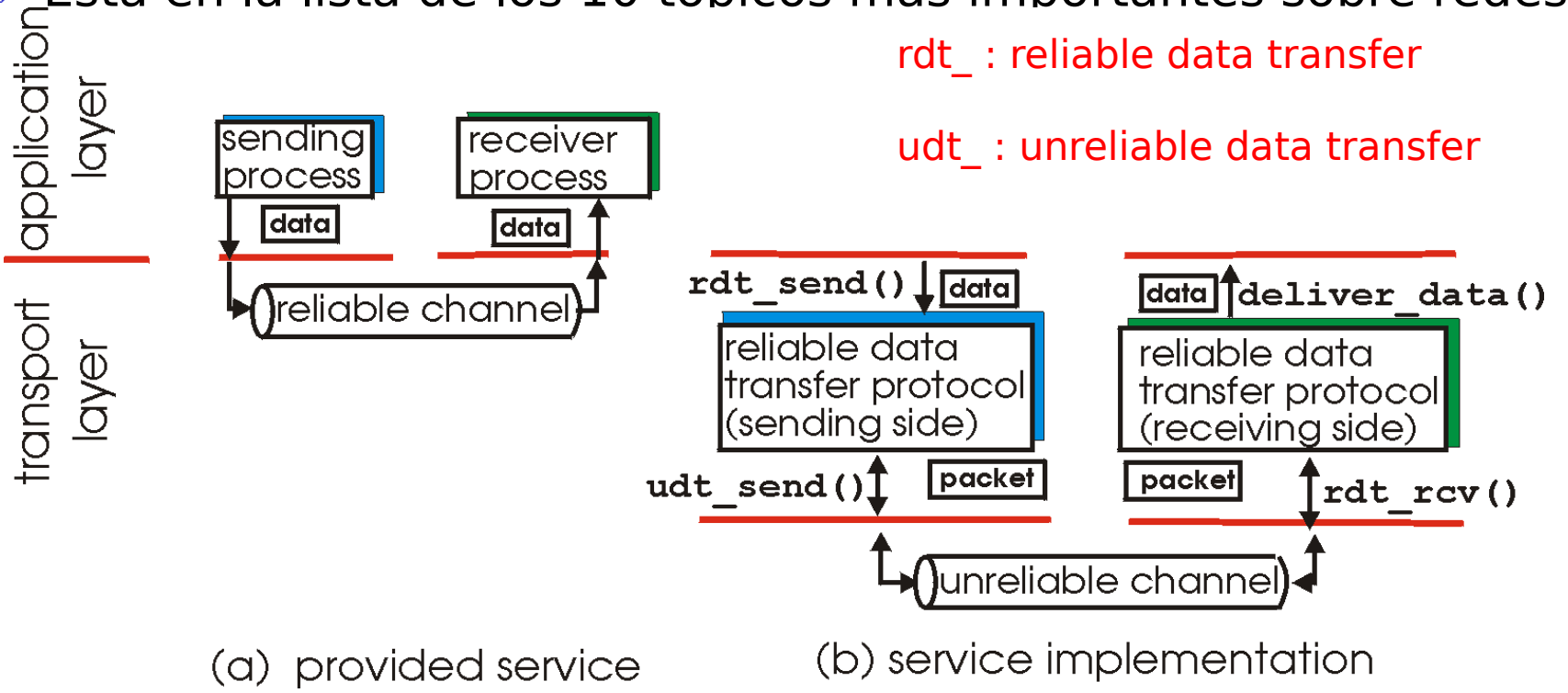
- Material de apoyo al texto *Computer Networking: A Top Down Approach Featuring the Internet.* Jim Kurose, Keith Ross.

Capítulo 3: Continuación

- 3.1 Servicios de la capa transporte
- 3.2 Multiplexing y demultiplexing
- 3.3 Transporte sin conexión: UDP
- 3.4 Principios de transferencia confiable de datos
- 3.5 Transporte orientado a la conexión: TCP
  - Estructura de un segmento
  - Transferencia confiable de datos
  - Control de flujo
  - Gestión de la conexión
- 3.6 Principios del control de congestión
- 3.7 Control de congestión en TCP

Principios de transferencia confiable de datos

- **Importante en capas de aplicación, transporte y enlace de datos**
- Está en la lista de los 10 tópicos más importantes sobre redes !

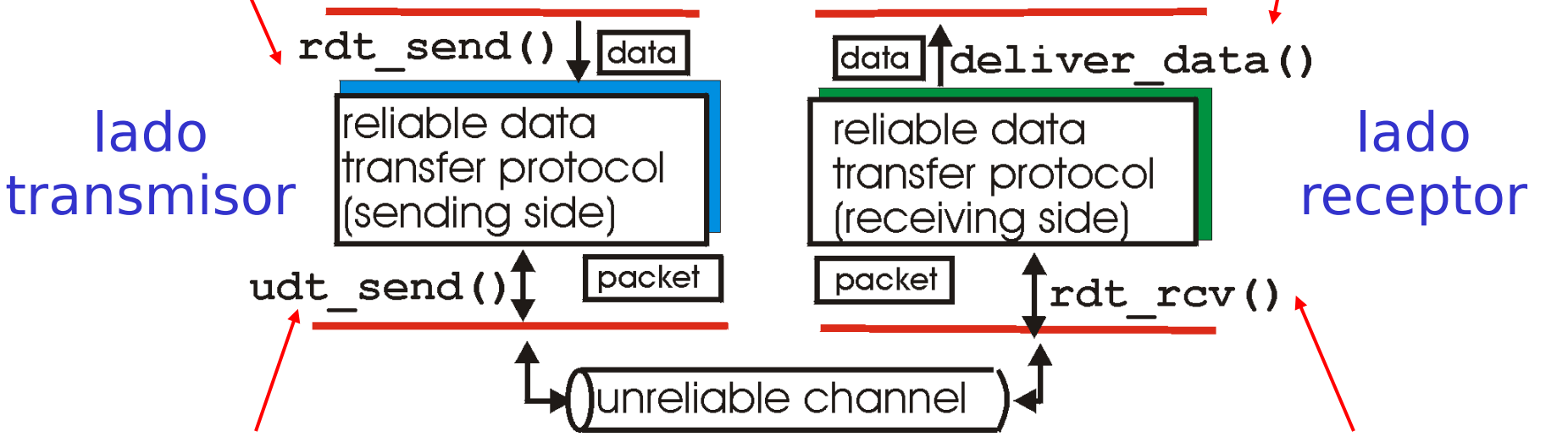


- Las características del canal no-confiable determinarán la complejidad del protocolo de datos confiable (reliable data transfer - rdt)

Transferencia confiable de datos: primeros aspectos (notación para esta parte)

`rdt_send()`: llamado desde arriba, (e.g., por aplicación). Recibe datos a entregar a la capa superior del lado receptor

`deliver_data()`: llamado por **`rdt`** para entregar los datos al nivel superior



`udt_send()`: llamado por rdt para transferir paquetes al receptor vía un canal no confiable

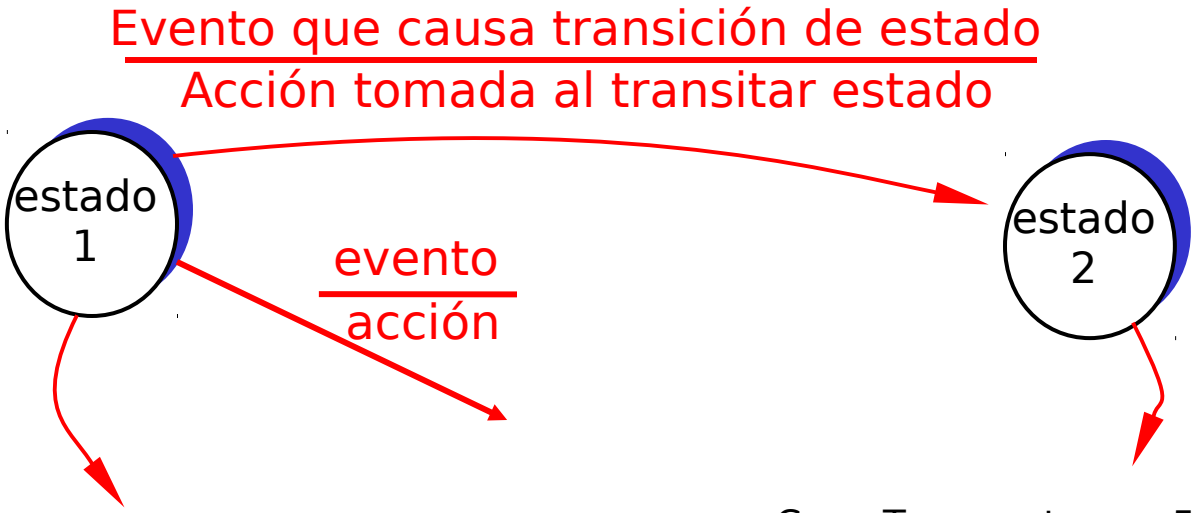
`rdt_rcv()`: llamada cuando un paquete llega al lado receptor del canal

Transferencia confiable de datos: primeros aspectos

Pasos a seguir:

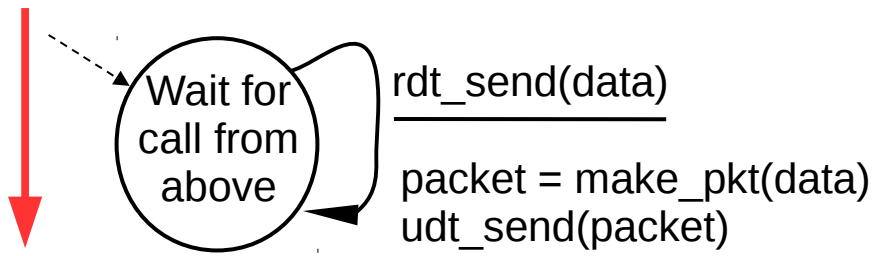
- Desarrollaremos incrementalmente los lados Tx y Rx del protocolo de transferencia confiable (rdt)
- Consideraremos sólo transferencias de datos unidireccionales
  - Pero la información de control fluirá en ambas direcciones!
- Usaremos máquina de estados finitos (Finite State Machine) para especificar el Tx y Rx

estado: cuando estamos en un "estado", el próximo es determinado sólo por el próximo evento

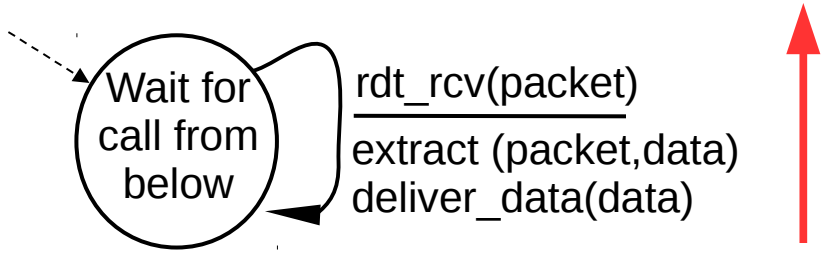


Rdt1.0: transferencia confiable sobre canal confiable (las bases)

- Canal subyacente utilizado es perfectamente confiable (caso ideal)
  - no hay errores de bit
  - no hay pérdida de paquetes
  - No hay cambio de orden en los paquetes
- Distintas MEFs (Máquina de Estados Finita) para el transmisor y receptor:
  - transmisor envía datos al canal inferior
  - receptor lee datos desde el canal inferior



Tx



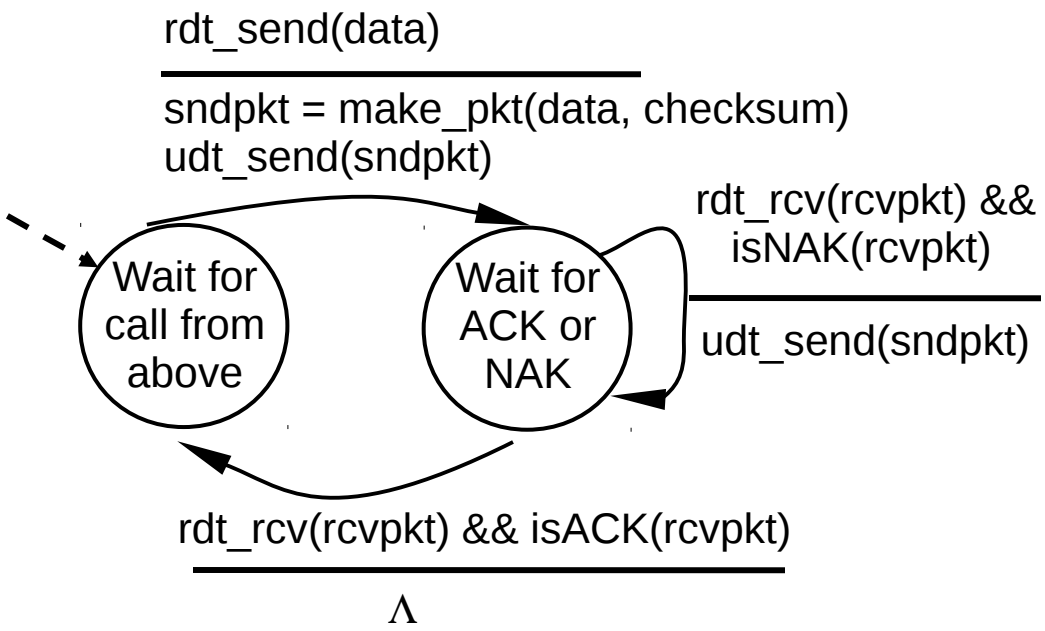
Rx

Rdt2.0: Canal con bits errados

- Canal inferior puede invertir bits del paquete
  - Usamos checksum para detectar los errores de bits
  - Supondremos que no se pierden paquetes ni hay desorden
- La pregunta: ¿Cómo recuperarnos de errores?:
  - *acknowledgements (ACKs):- acuses de recibo:* receptor explícitamente le dice al Tx que el paquete llegó OK
  - *negative acknowledgements (NAKs): – acuses de recibo negativos:* receptor explícitamente le dice al Tx que el paquete tiene errores.
  - Tx re-transmite el paquete al recibir el NAK
- Nuevos mecanismos en **`rdt2.0`** (sobre **`rdt1.0`**):
  - Detección de errores
  - Realimentación del receptor: mensajes de control (ACK, NAK) Tx <-------- Rx

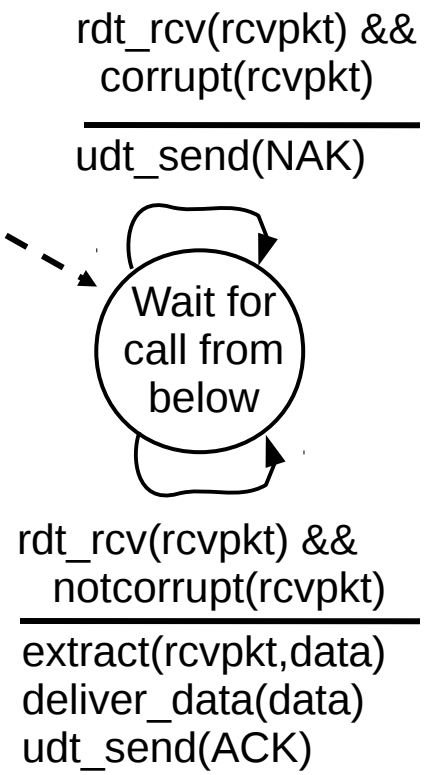
rdt2.0: Especificación de la MEF

Tx

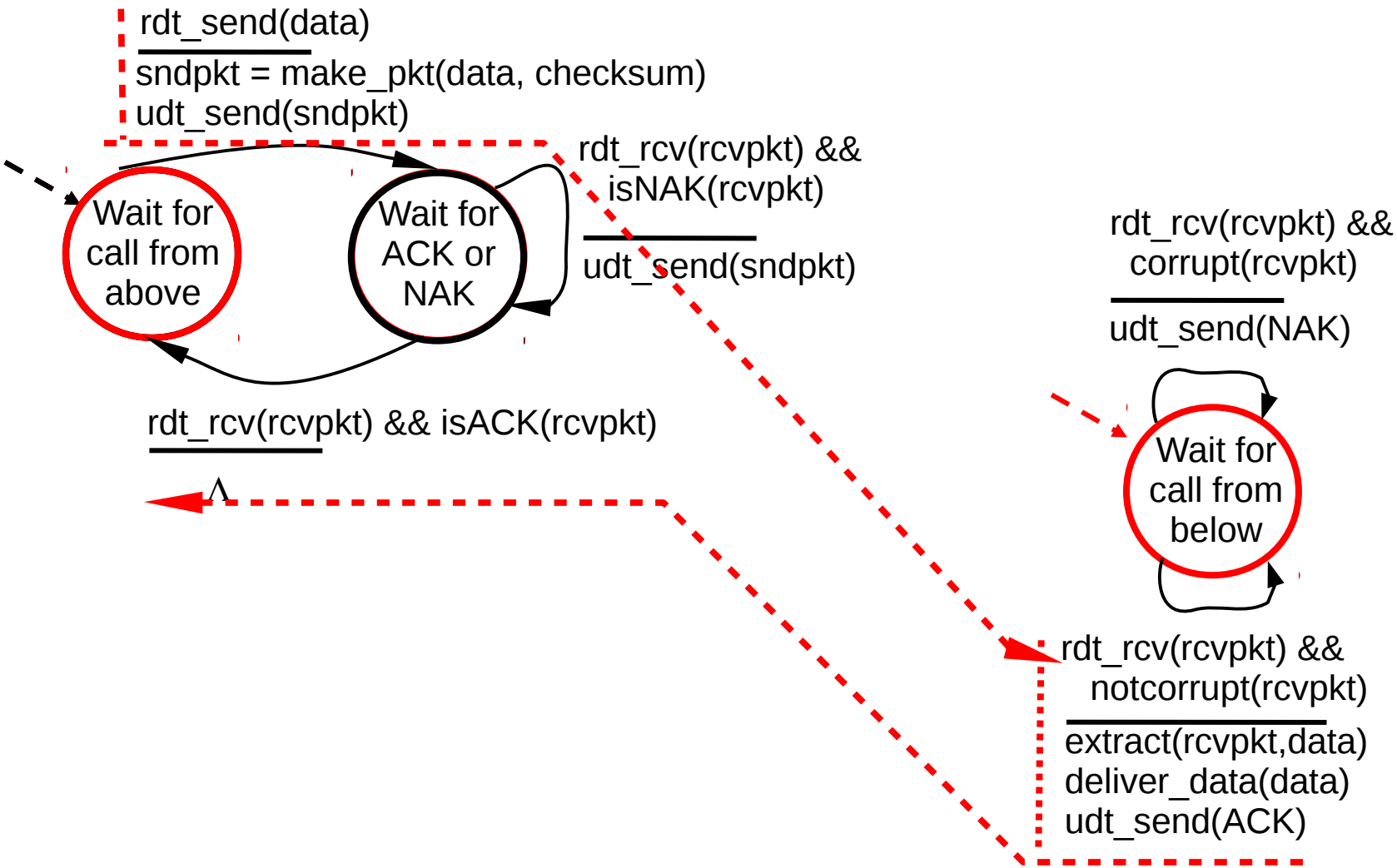


Λ: hacer nada

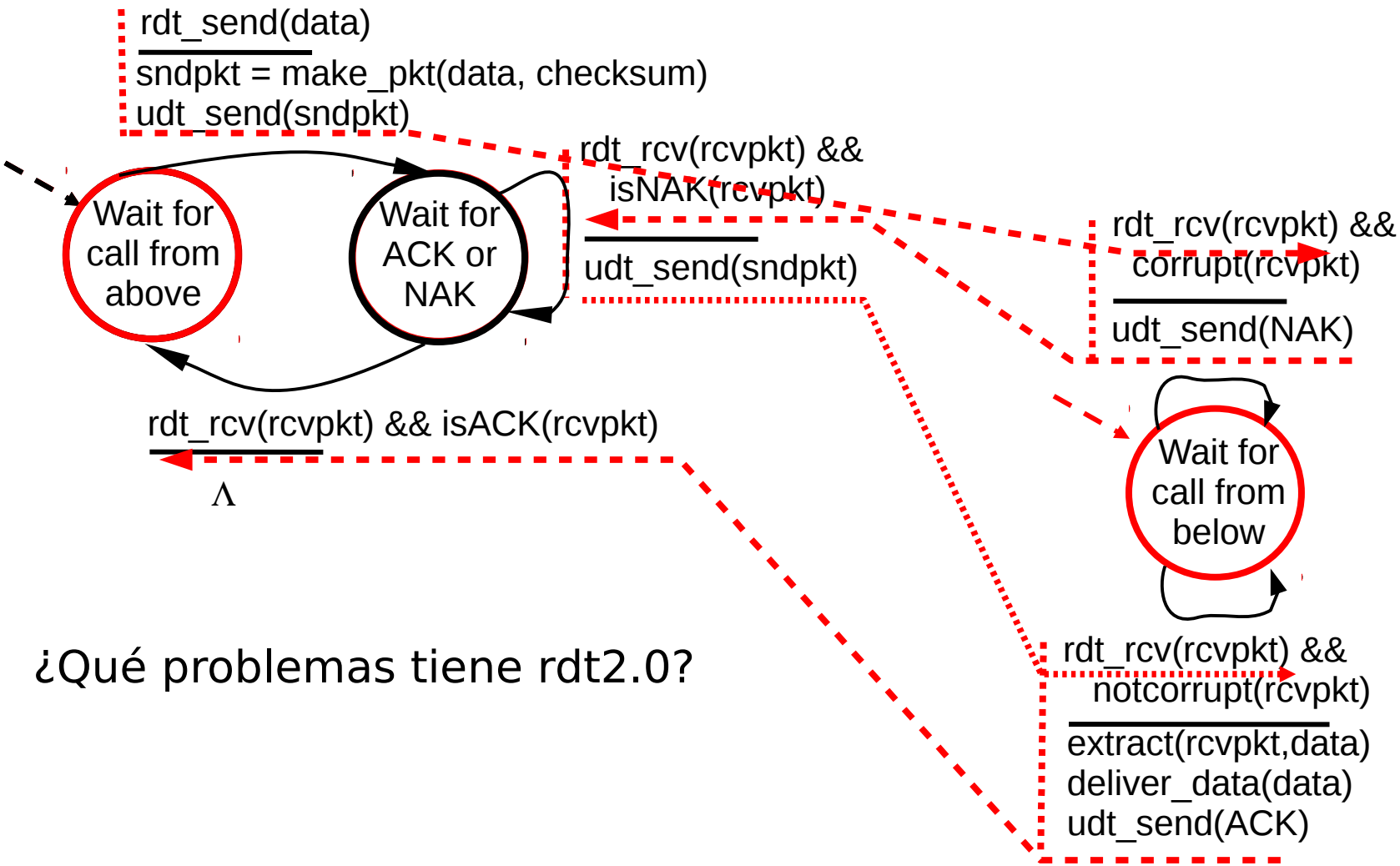
Rx



rdt2.0: operación sin errores



rdt2.0: operación con error y una retransmisión



¿Qué problemas tiene rdt2.0?

rdt2.0 tiene una falla fatal!

¿Qué pasa si se corrompe el ACK/NAK?

- Tx no sabe qué pasó en el receptor!
- Idea, retransmitir paquete ante la llegada de un ACK o NAK dañado.
- No puede sólo retransmitir: generaría posible duplicado
- Surge necesidad de poner números de secuencia para detectar duplicados.

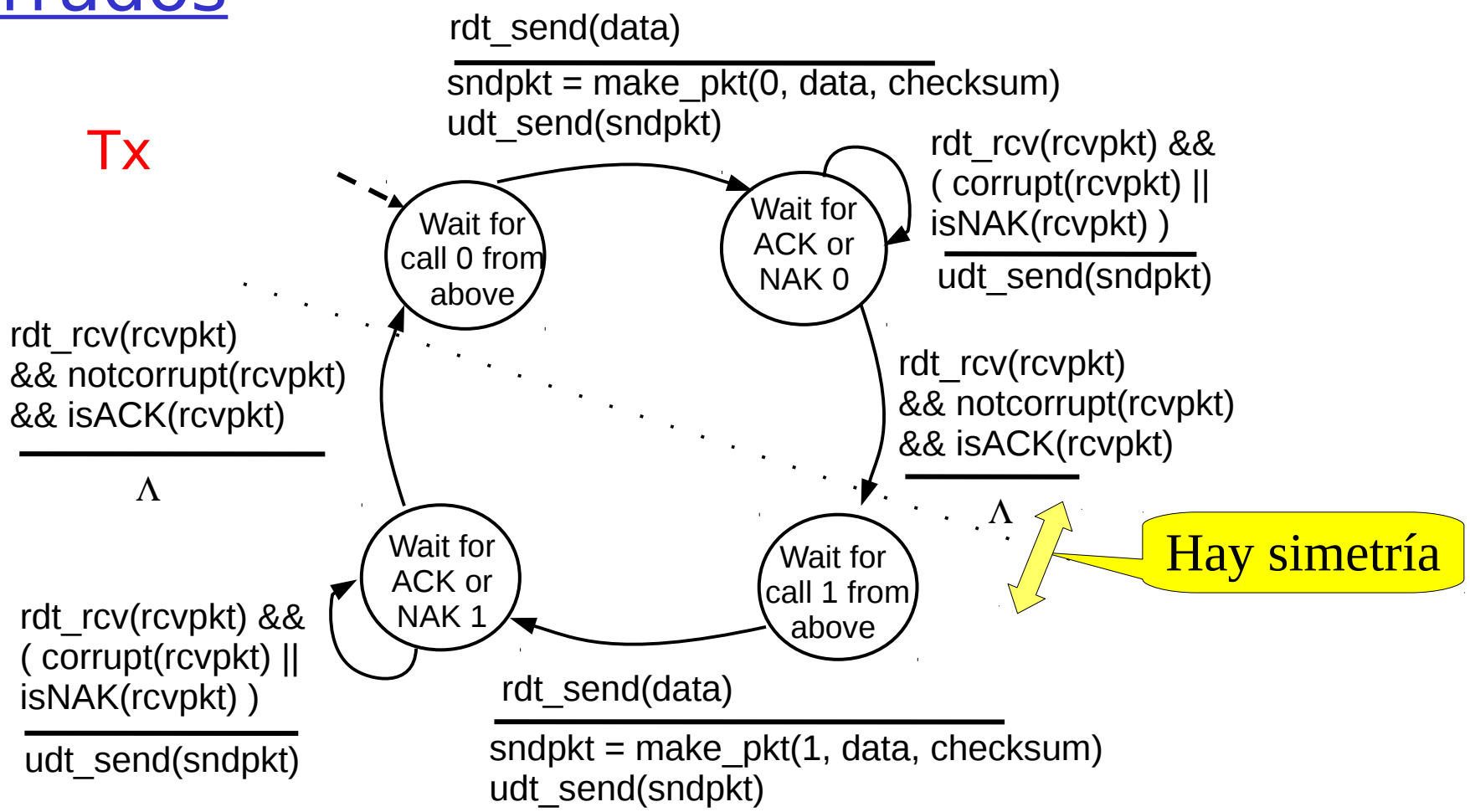
Manejo de duplicados:

- Tx agrega *números de secuencia* a cada paquete
- Tx retransmite el paquete actual si ACK/NAK llega mal
- El receptor descarta (no entrega hacia arriba) los paquetes duplicados

stop and wait

Tx envía un paquete,
Luego para y espera por la respuesta del Rx

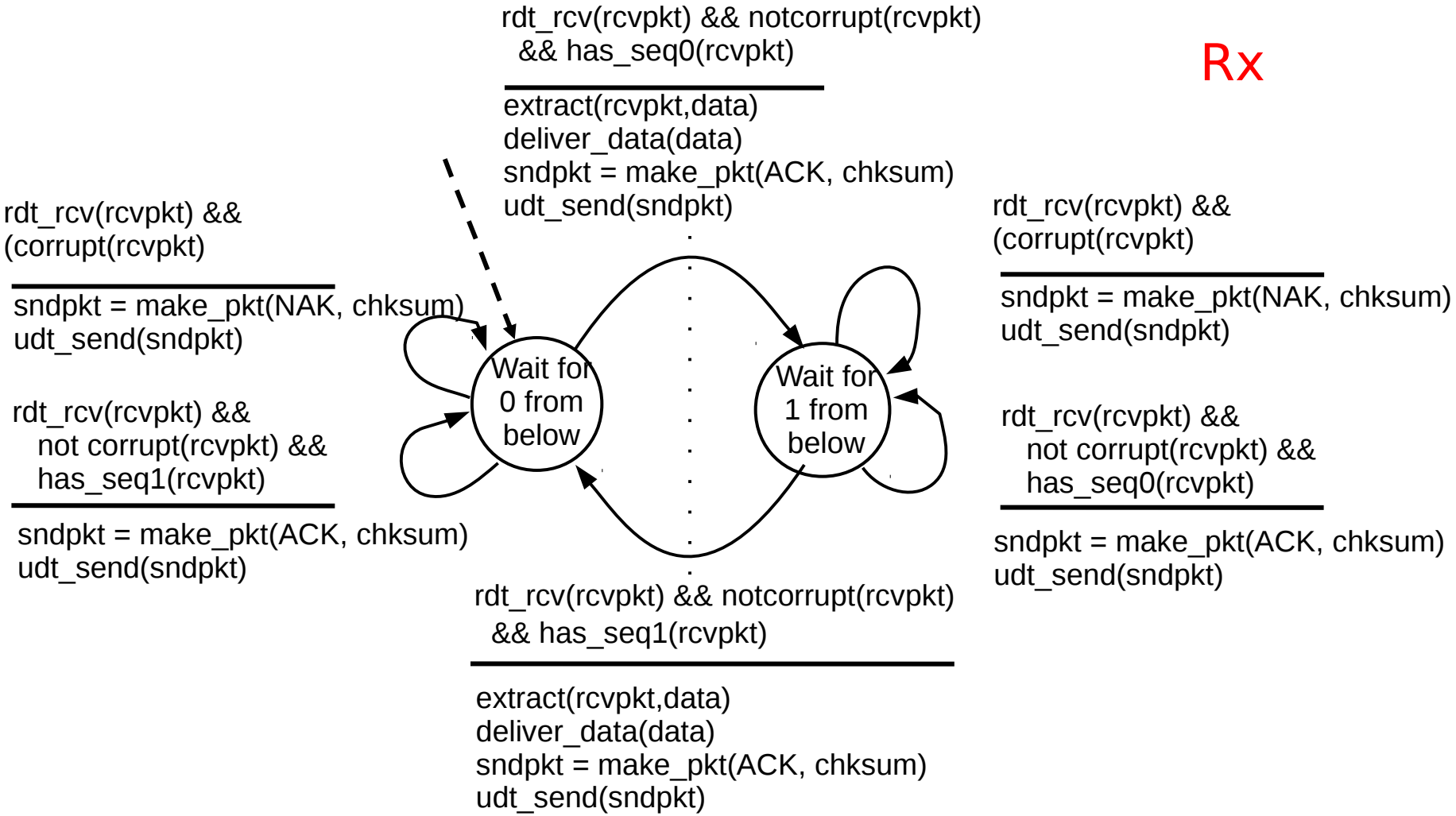
rdt2.1: Tx, manejo de ACK/NAKs errados



Transmisor incluye # de secuencia para permitir al receptor descartar duplicados

rdt2.1: Receptor, manejo de ACK/NAKs errados

Rx



rdt2.1: discusión

Transmisor:

- Agrega # secuencia al paquete
- 2 #'s (0,1) de secuencia son suficientes, por qué?
- Debe chequear si el ACK/NAK recibido está corrupto.
- El doble del número de estados
  - Estado debe "recordar" si paquete "actual" tiene # de secuencia 0 ó 1

Receptor:

- Debe chequear si el paquete recibido es duplicado
  - Estado indica si el número de secuencia esperado es 0 ó 1
- Nota: el receptor *no* puede saber si su último ACK/NAK fue recibido OK por el Tx

¿Podemos adaptar rdt2.1 para tolerar pérdidas de paquetes?

rdt2.2: un protocolo libre de NAK

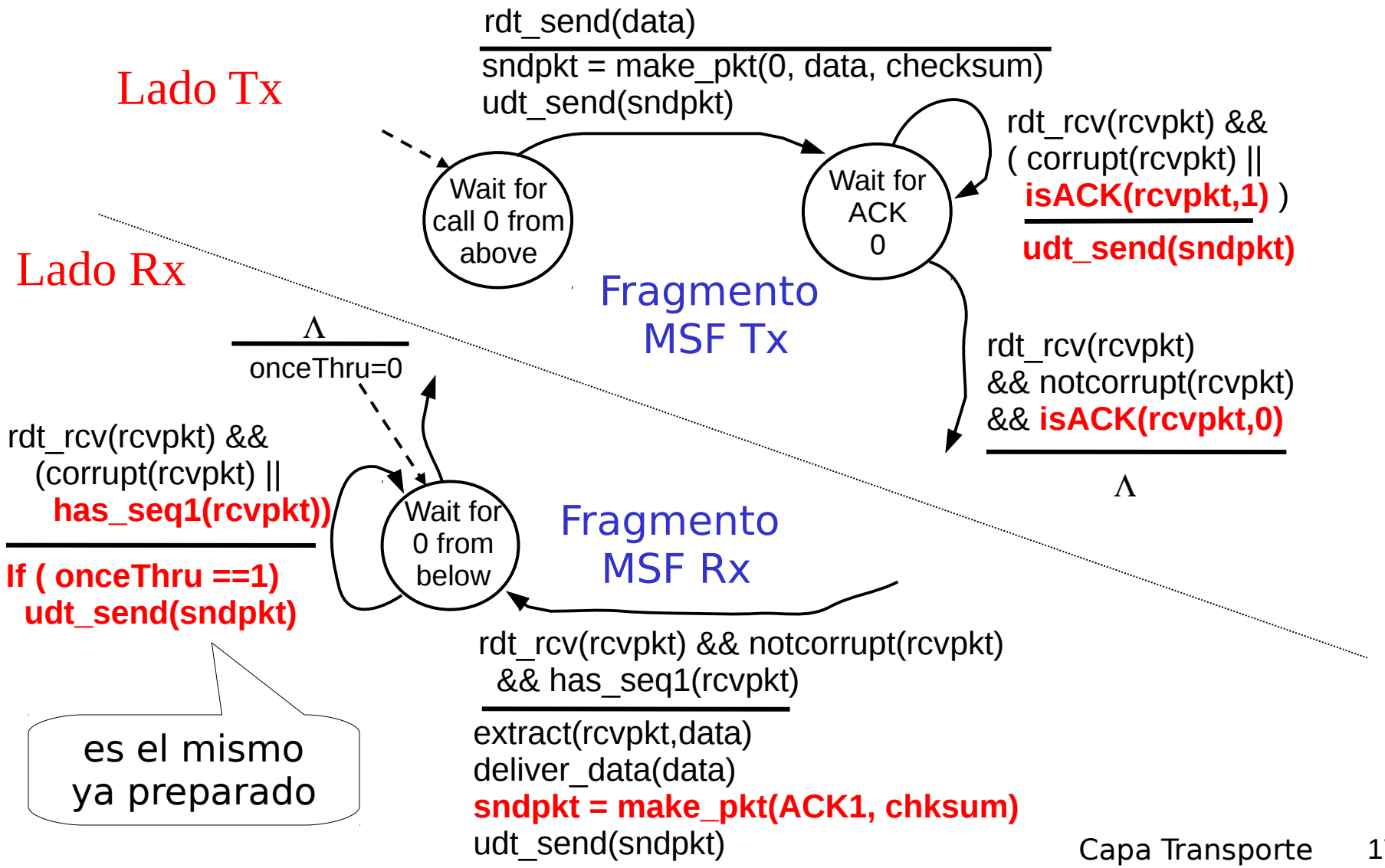
- No posee problemas, pero preparándonos para la pérdida de paquetes, es mejor prescindir de los NAK.
- No podemos enviar NAK de un paquete que nunca llegó.
- Se busca la misma funcionalidad que rdt2.1, usando sólo ACKs
- En lugar de NAK, el receptor re-envía ACK del último paquete recibido OK
  - Receptor debe *explícitamente* incluir # de secuencia del paquete siendo confirmado con el ACK
- ACK duplicados en el Tx resulta en la misma acción que NAK: *retransmitir paquete actual*

En rdt2.2 seguiremos asumiendo que no hay pérdidas

¿Cómo usted compara usar sólo NAK versus usar sólo ACK?

- ¿Qué le dice su madre/padre: llámame al llegar a destino -ACK- o llámame si tienes algún problema -NAK?
- Si no hay pérdidas, se podría ahorrar mensajes si se trabaja sólo con NAK; sin embargo, esto no permite hacer control de flujo pues no tenemos evento para decidir el envío de otro paquete. Deberíamos usar un timer, si no llega NAK, enviar nuevo paquete con el timer. El caso exitoso resulta lento.
- Si pérdidas son posibles, el uso de NAK debe descartarse, pues no podemos distinguir entre un paquete que llegó bien de uno perdido.
- Resumen: sí podemos usar solo ACKs, pero no solo NAKs.

rdt2.2: Fragmentos del Transmisor y receptor



Hasta aquí

- Si el canal es ideal, el mecanismo es simple: solo enviar los datos (rdt 1.0).
- Si hay errores, pero no se pierden paquetes, usar ACK y NAK. (rdt 2.0)
- Si los ACK o NAK también pueden venir con errores,en este caso el tx re-envía el paquete; entonces debemos usar número de secuencia para eliminar duplicados. (rdt 2.1)
- Se puede evitar NAK, enviando ACK duplicados en lugar de NAK, entonces debemos usar número de secuencia para detectar ACK duplicados (ver rdt 2.2)

rdt3.0: Canales con errores y pérdidas

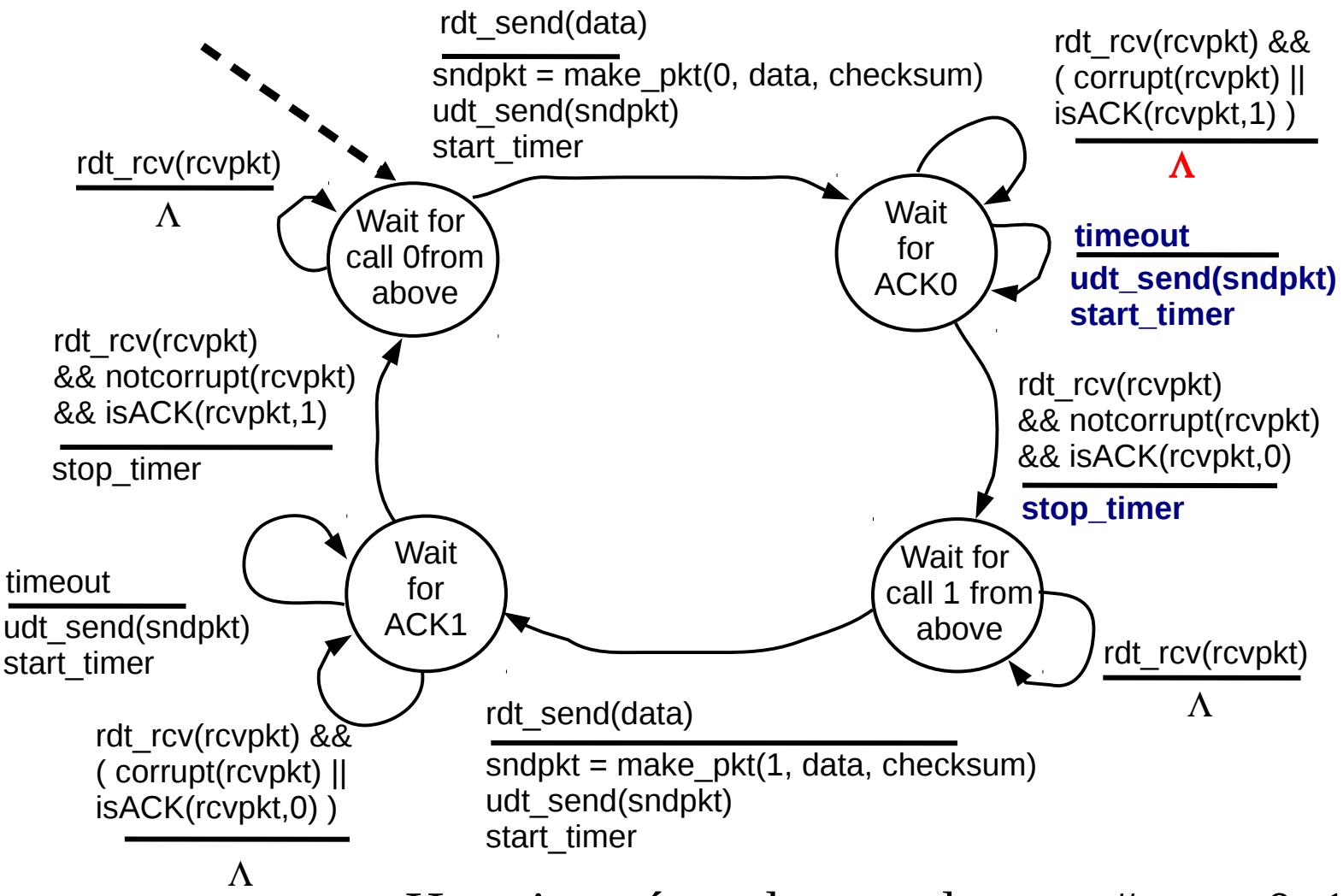
Suposición nueva:

- Canal subyacente también puede perder paquetes (de datos o ACKs)
  - checksum, # de secuencias, ACKs, y retransmisiones ayudan pero no son suficientes

Estrategia: transmisor espera un tiempo "razonable" por el ACK

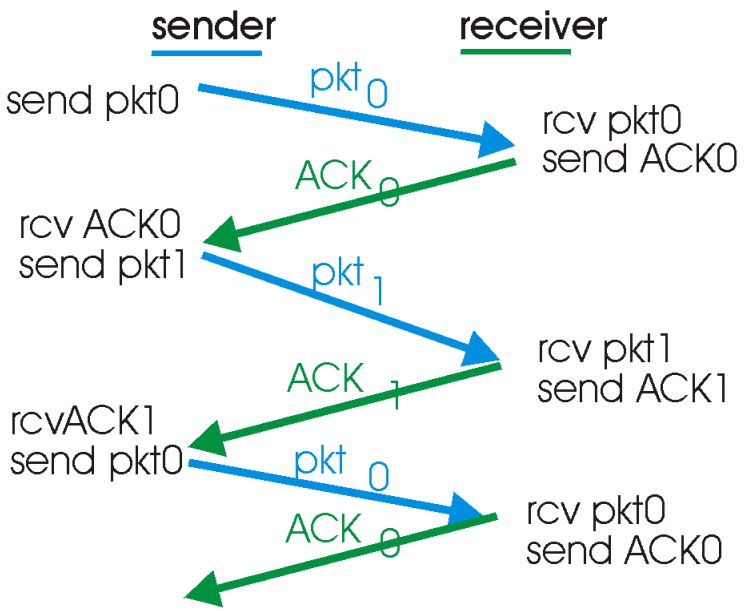
- Retransmitir si no se recibe ACK en este tiempo
- Si el paquete (o ACK) está retardado (no perdido):
  - La retransmisión será un duplicado, pero el uso de #'s de secuencia ya maneja esto
  - Receptor debe especificar el # de secuencia del paquete siendo confirmado en el ACK
- Se requiere un temporizador.
- Este protocolo se conoce como: Stop and wait protocol (parar y esperar)

rdt3.0 Transmisor

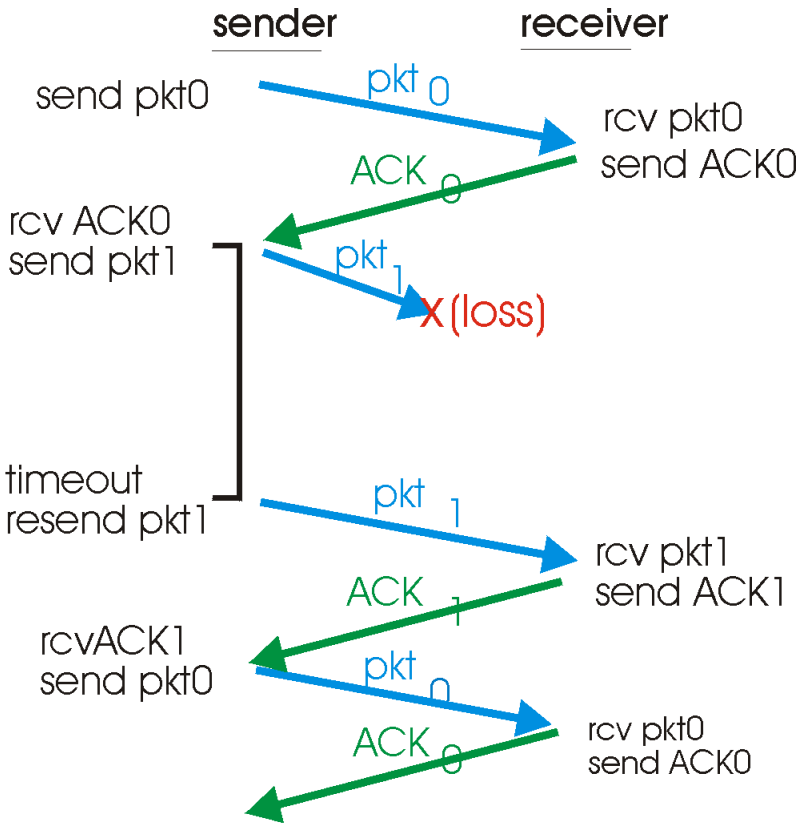


Hay simetría en los estados con # sec.=0, 1

rdt3.0 en acción

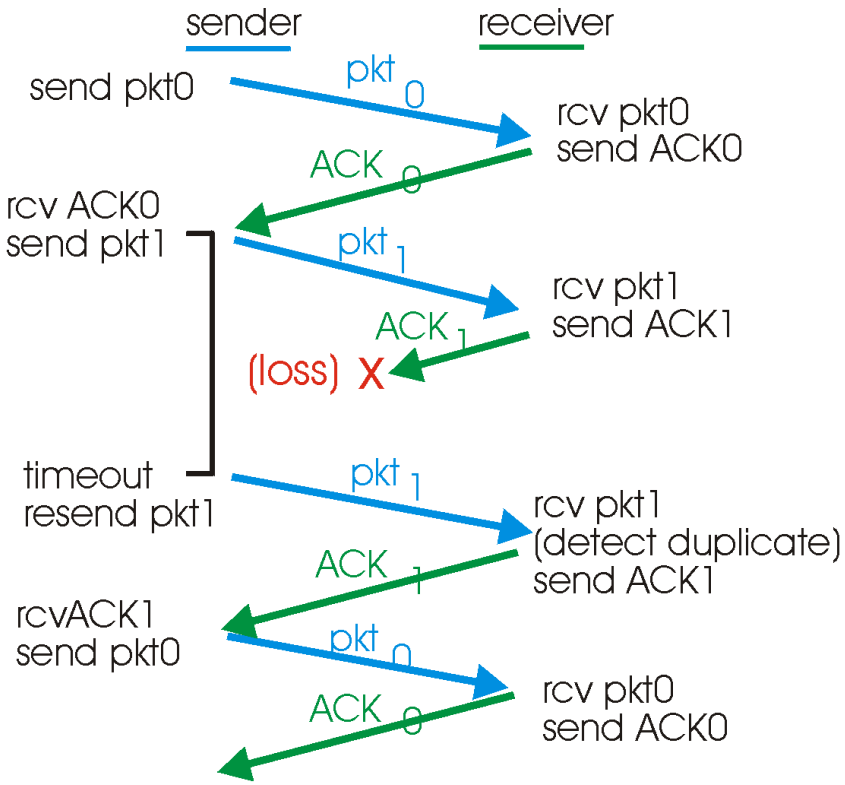


a) Operación **sin** pérdidas

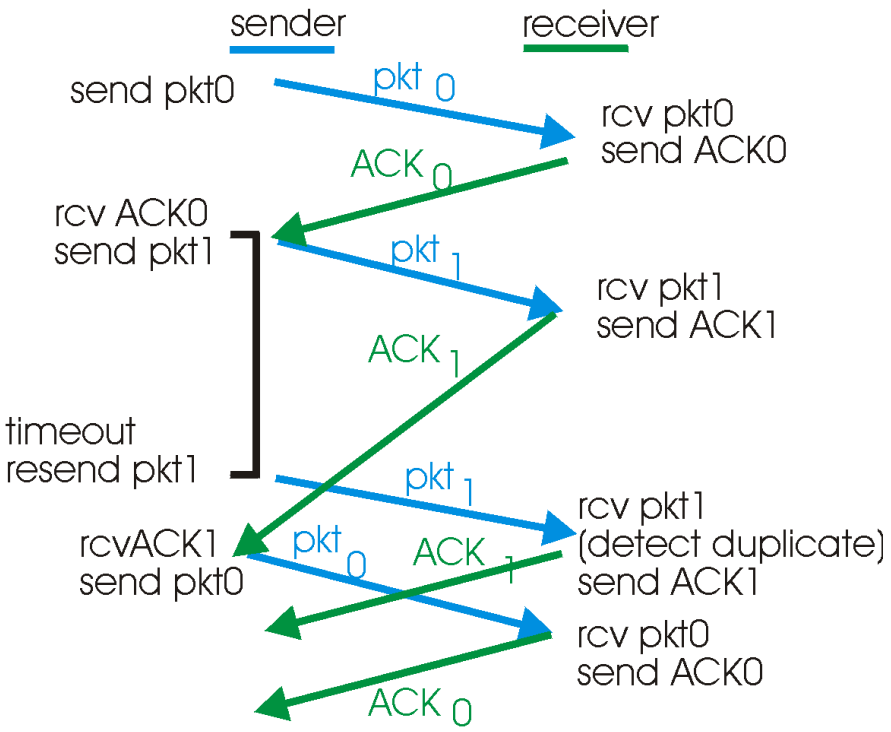


b) Operación **con** pérdidas

rdt3.0 en acción

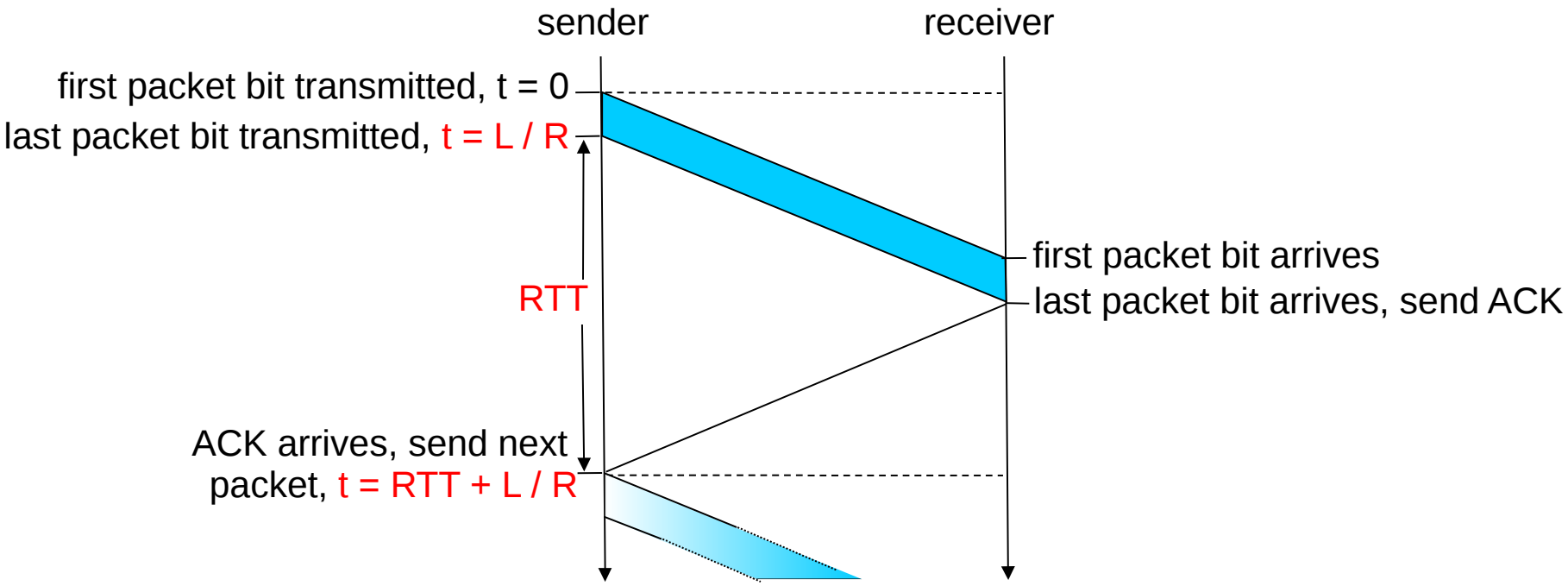


c) Pérdida de ACK



d) Timeout prematuro

rdt3.0: protocolo stop & wait



$$Utilización_{transmisor} = \frac{L/R}{RTT + L/R} = \frac{0{,}008}{30{,}008} = 0{,}00027 = 0{,}027$$

Baja utilización, ¿recuerdan cómo se mejora esto?

Desempeño de rdt3.0

- rdt3.0 funciona, pero su desempeño es malo
- Ejemplo: R = enlace de 1 Gbps, 15 ms de retardo extremo a extremo, L = paquetes de 1KB, RTT = 30ms.

$$T_{transmitir} = \frac{L}{R} = \frac{8\,Kb/paquete}{10^9\,b/s} = 8\mu s$$

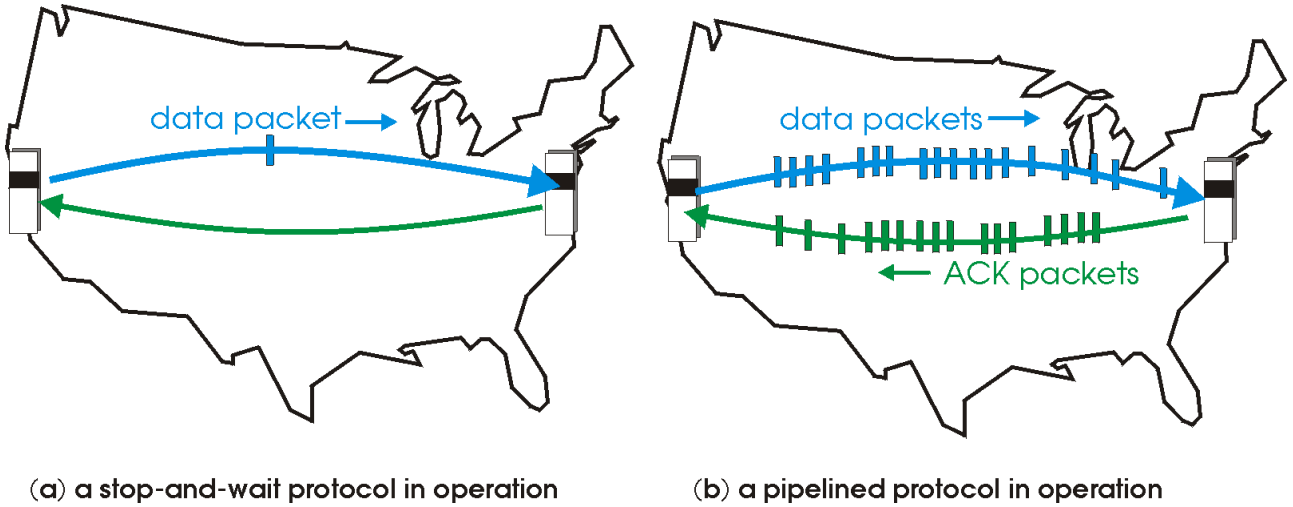
$$U_{transmisor} = \frac{L/R}{RTT + L/R} = \frac{0.008}{30.008} = 0.00027 = 0.027\ \%$$

  - $U_{transmisor}$: utilización del transmisor o canal = fracción de tiempo que el transmisor/canal está ocupado transmitiendo
  - 1 paquete de 1KB cada ~30 ms -> 33kB/s tasa de transmisión en enlace de 1 Gbps
  - Protocolo de red limita el uso de los recursos físicos!

Protocolos con Pipeline

Con Pipeline: Transmisor permite múltiples paquetes en tránsito con acuse de recibo pendiente

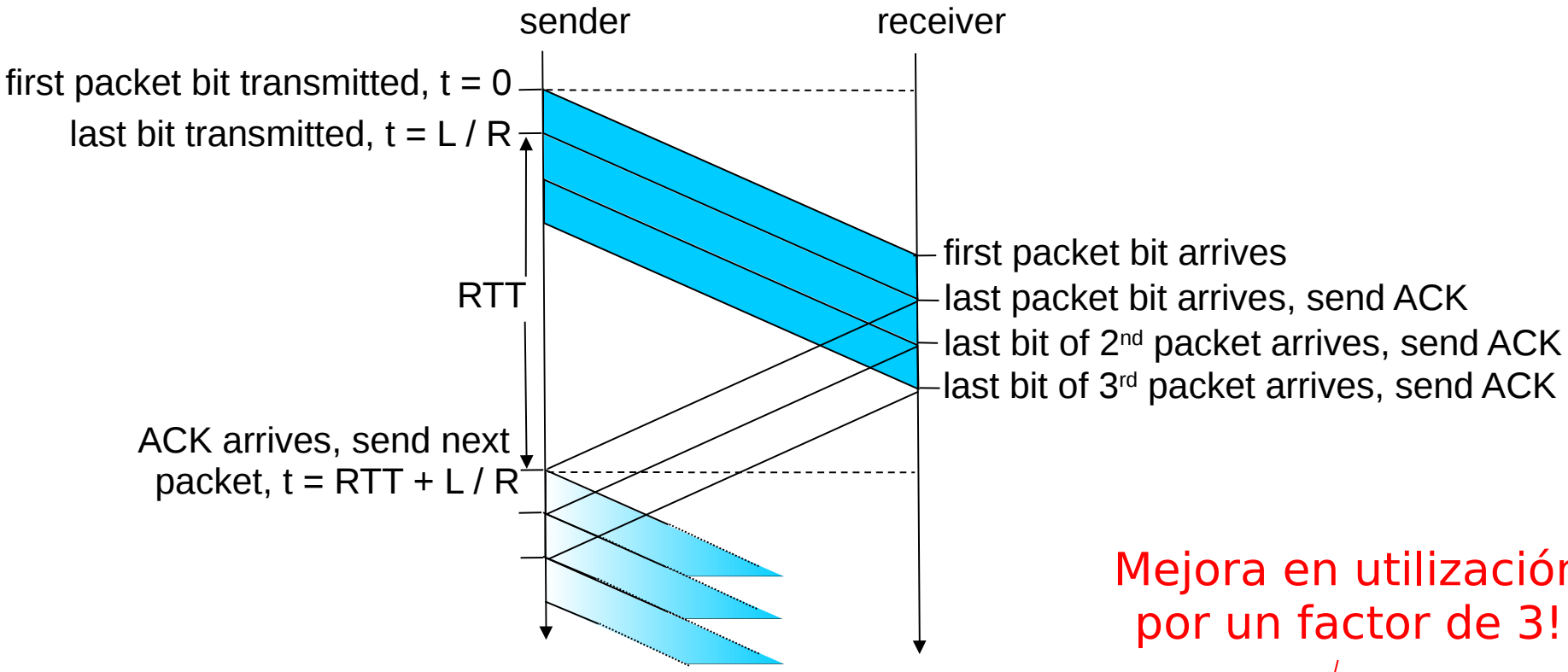
- El rango de los números de secuencia debe ser aumentado
- Se requiere buffers en el Tx y/o Rx



(a) a stop-and-wait protocol in operation (b) a pipelined protocol in operation

- Hay dos formas genéricas de protocolos con pipeline: *go-Back-N y selective repeat (repetición selectiva)*

Pipelining: utilización mejorada



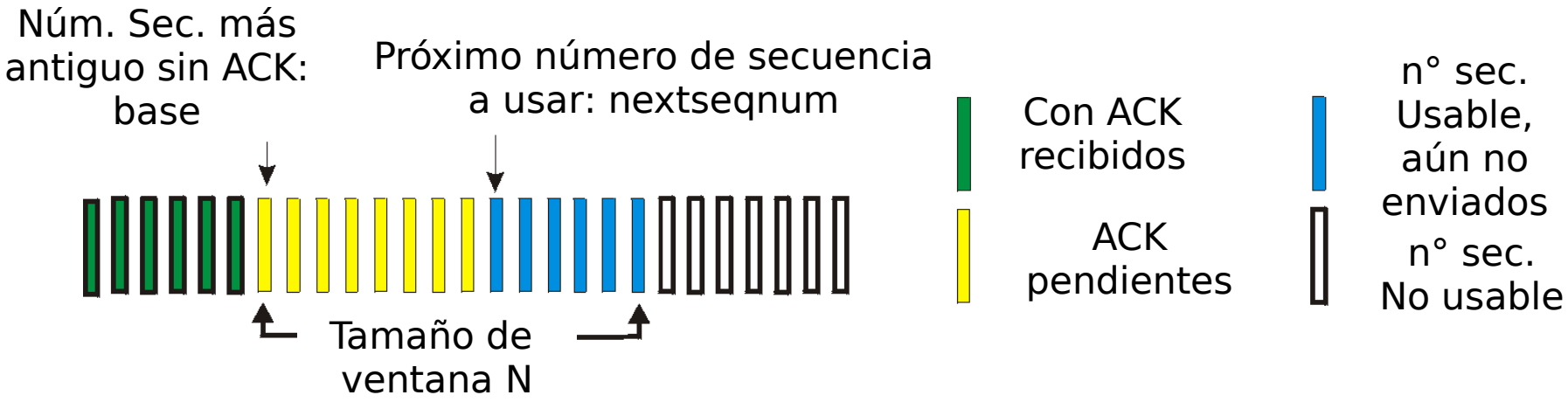
Mejora en utilización por un factor de 3!

$$U_{sender} = \frac{3 * L / R}{RTT + L / R} = \frac{.024}{30.008} = 0.0008 = 0.08\%$$

Go-Back-N

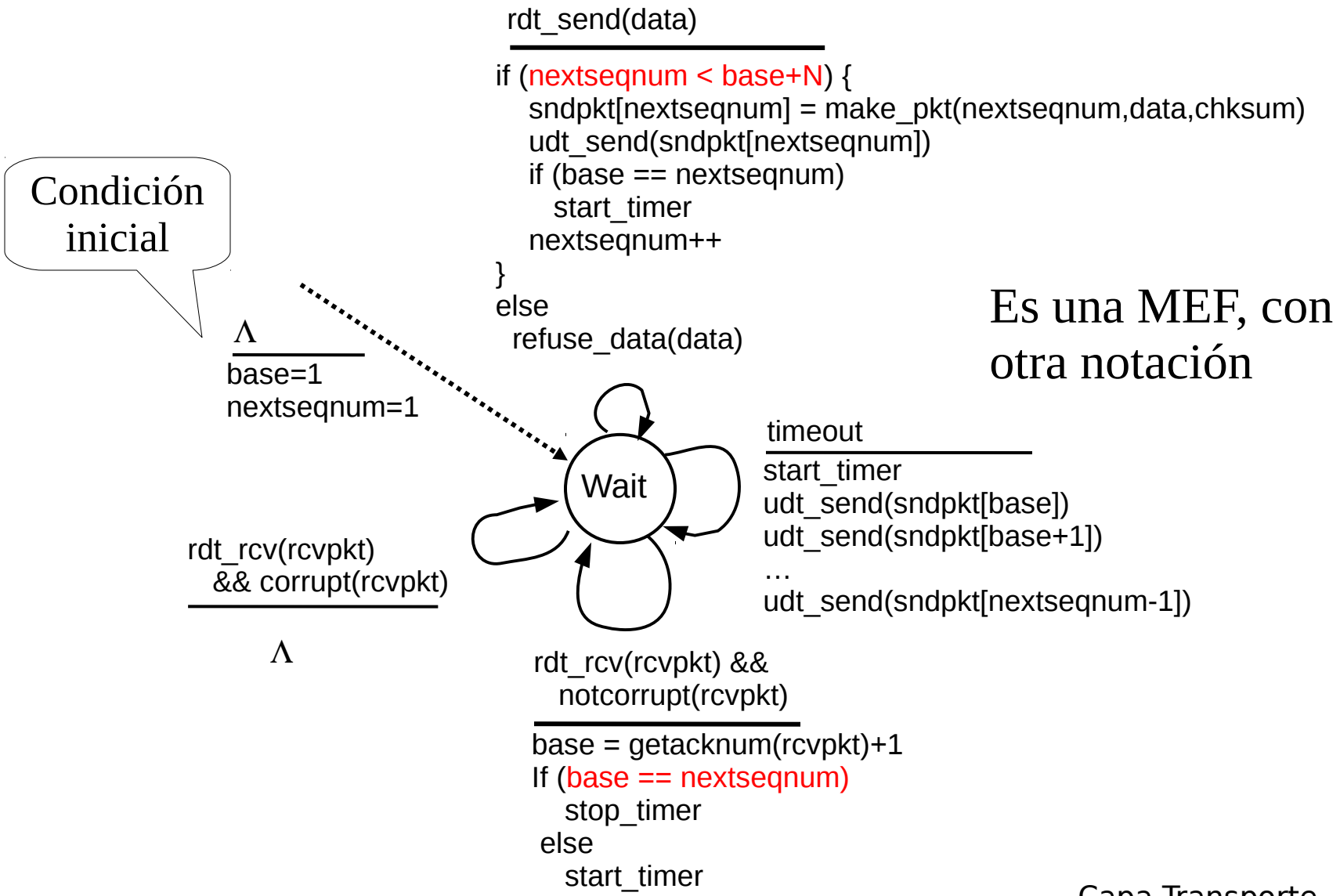
Transmisor:

- # de secuencia de k-bits en el encabezado del paquete
- "ventana" de hasta N paquetes consecutivos con acuse de recibo pendiente



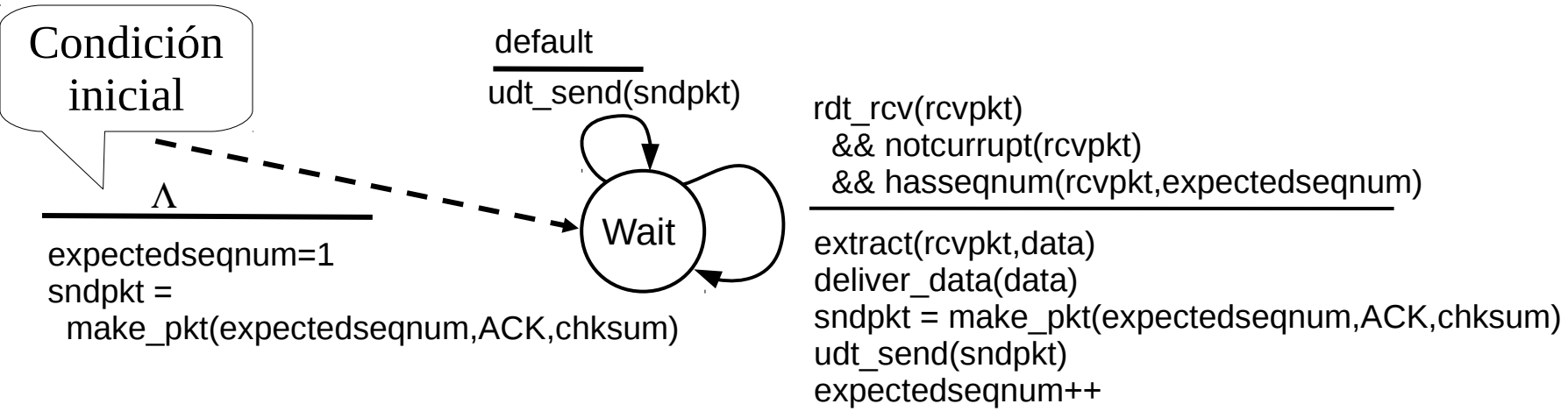
- Cuando llega un ACK(n): da acuse de recibo a todos los paquetes previos, incluyendo aquel con # de secuencia n; corresponde a un "acuse de recibo acumulado"
  - Podría recibir ACKs duplicados (ver receptor)
- Usa un timer para manejar la espera de ack de paquete en tránsito
- *timeout(n):* retransmitir paquete n y todos los paquetes con número de secuencia siguientes en la ventana

GBN: MEF extendida del Transmisor



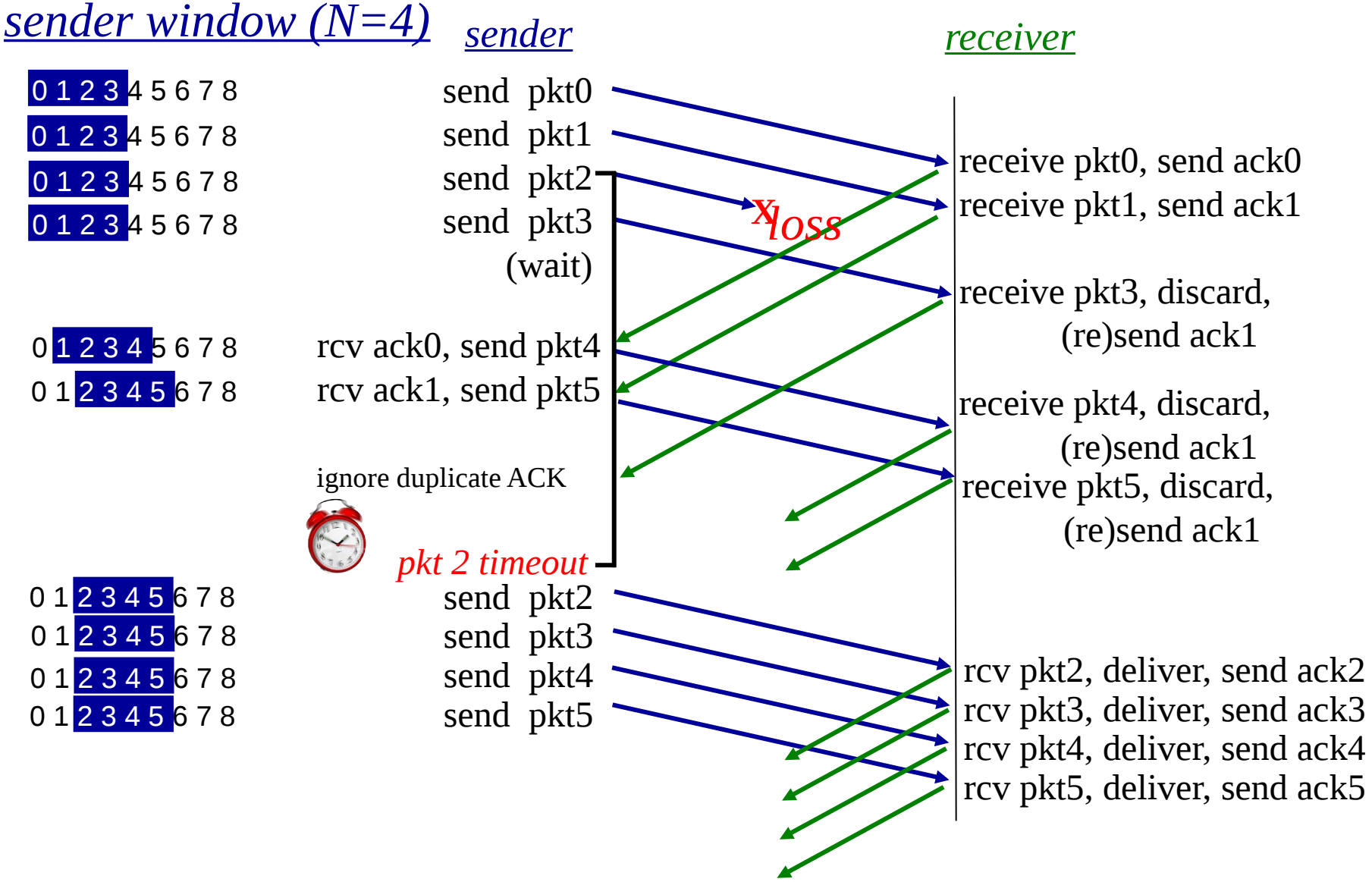
Es una MEF, con otra notación

GBN: MEF extendida del Receptor



- Usa sólo ACK: siempre envía ACK de paquete correctamente recibido con el # de secuencia mayor en orden
  - Puede generar ACKs duplicados. ¿Cuándo? Ver animación
  - Sólo necesita recordar **`expectedseqnum`**
- Paquetes fuera de orden:
  - Descartarlos (no almacenar en buffer) => no requiere buffer en receptor! Sólo para almacenar el paquete recibido.
  - Re-envía ACK del paquete de mayor número de secuencia en orden

GBN en acción



¿Para qué re-enviar paquetes correctamente recibidos?

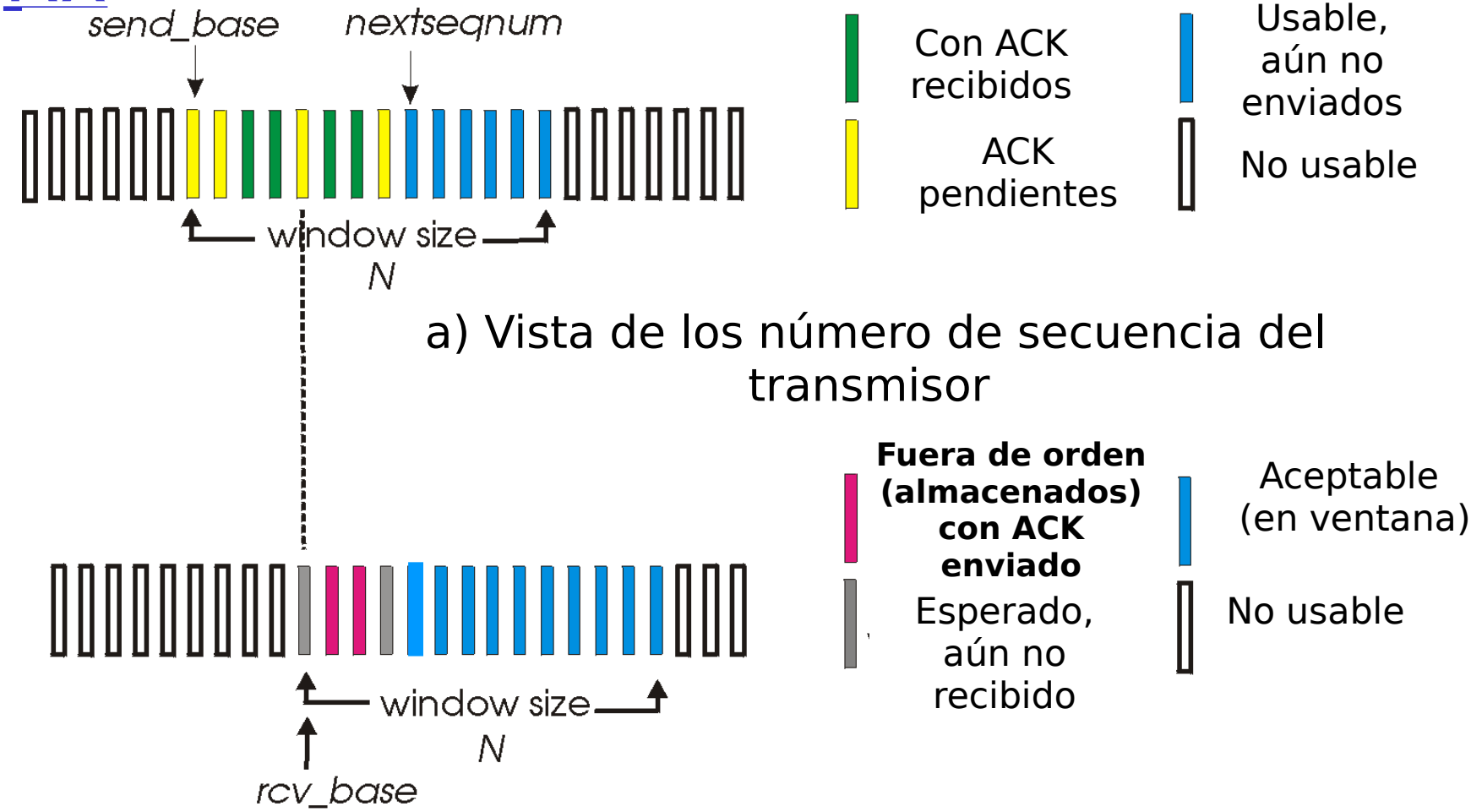
Go-Back-N: Análisis versión texto guía

- Idea Básica:
  - Tx: Enviar hasta completar ventana.
  - Rx: Sólo aceptar paquete correcto y en orden
- En caso de error o pérdida:
  - Tx: Lo detecta por timeout y retransmite todo desde el perdido o dañado en adelante.
- Reflexionar:
  - La pérdida sólo es detectada por el Tx luego de un timeout. Pero éste se reinicia con cada ACK que no sea el último. Convendría tener un timer por paquete enviado? Ocuparía más timers.
  - Por qué reiniciar timer ante ACK distinto del último?
  - ¿Por qué acá un ACK duplicado no es considerado como paquete perdido?

Selective Repeat (repetición selectiva)

- Receptor envía acuse de recibo *individuales* de todos los paquetes recibidos
  - Almacena paquetes en buffer, según necesidad para su entrega en orden a la capa superior
- Transmisor sólo re-envía los paquetes sin ACK recibidos
  - Transmisor usa un timer por cada paquete sin ACK
- Ventana del Transmisor
  - Es la cantidad de números de secuencia consecutivos que puede enviar.
  - Nuevamente limita los #s de secuencia de paquetes enviados sin ACK
- Existe ventana en Receptor

Selective repeat: Ventanas de Tx y Rx



a) Vista de los número de secuencia del transmisor

b) Vista de los número de secuencia del receptor

Selective repeat (repetición selectiva)

Transmisor

Llegan datos desde arriba:

- Si el próximo # de sec. está en ventana, enviar paquete

timeout(n):

- Re-enviar sólo paquete n, re-iniciar timer

ACK(n) en [sendbase,sendbase+N]:

- Marcar paquete n como recibido, parar su timer
- Si n es el paquete más antiguo sin ACK, avanzar la base de la ventana al próximo # de sec. sin ACK.

Receptor

Llega paquete n en [rcvbase, rcvbase+N-1]

- Enviar ACK(n)
- Si está fuera de orden: almacenar en buffer
- En-orden: entregar a capa superior (también entregar paquetes en orden del buffer), avanzar ventana al paquete próximo aún no recibido

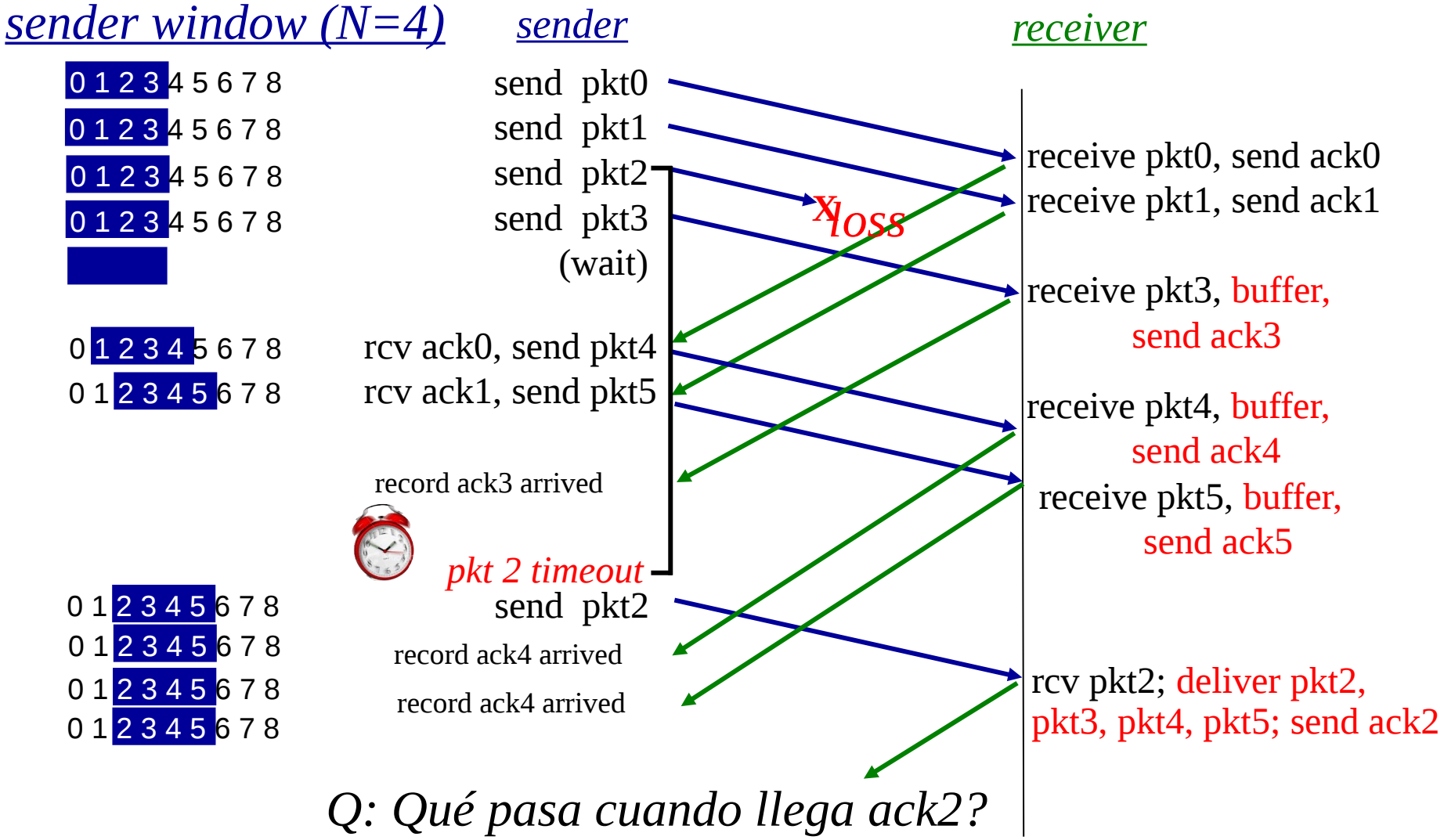
paquete n en [rcvbase-N, rcvbase-1]

- Enviar ACK(n)

Otro caso:

- ignorarlo

Repetición Selectiva en Acción



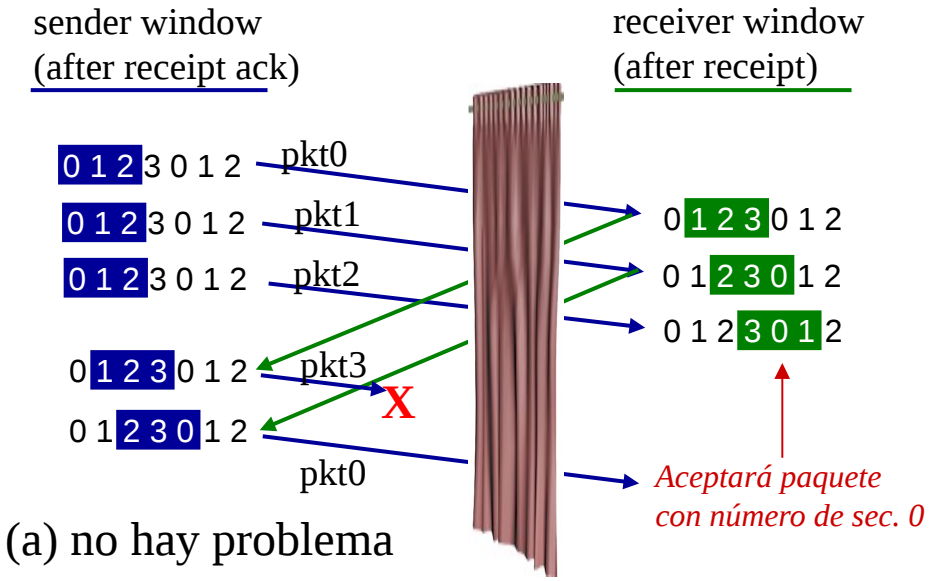
Dilema de la repetición Selectiva

Ejemplo:

- #s de sec.: 0, 1, 2, 3
- Tamaño de ventana=3

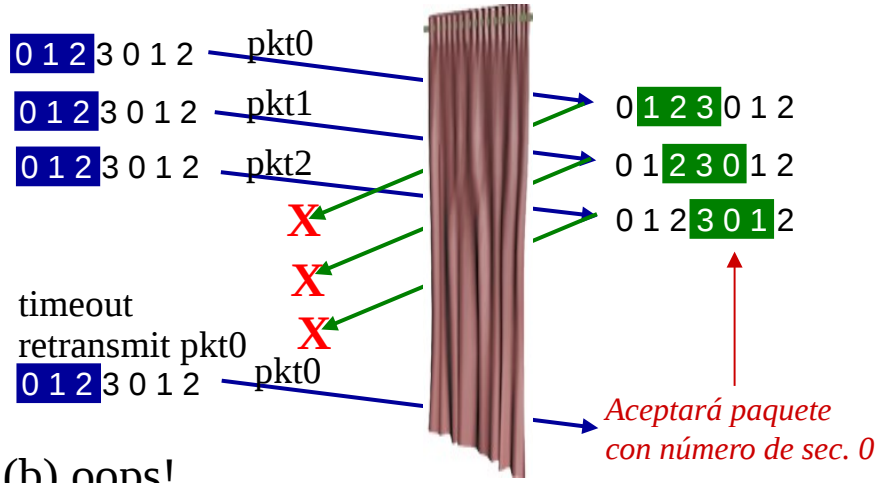
- Rx no ve diferencia en los dos escenarios!
- Pasa incorrectamente datos como nuevos en (b)

Q: ¿Qué relación debe existir entre el # de sec. y el tamaño de ventana?



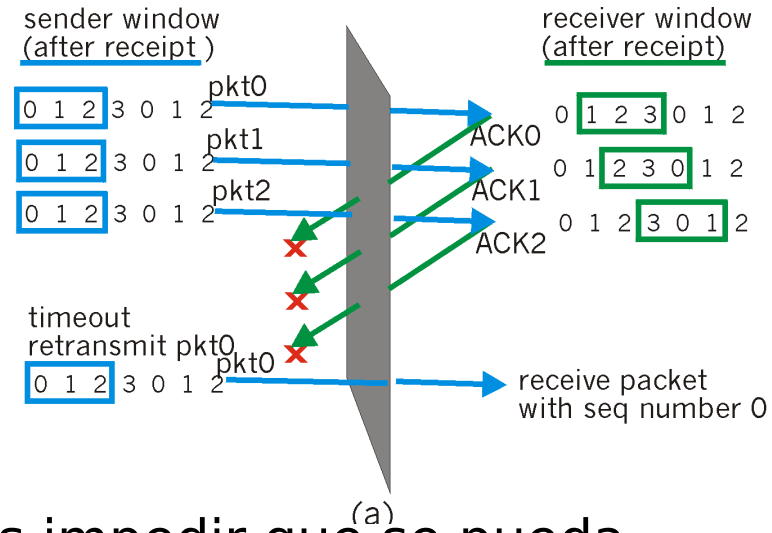
(a) no hay problema

El receptor no puede ver el lado Tx.
Igual acción de Rx en ambos casos!
Algo está (muy) mal!



(b) oops!

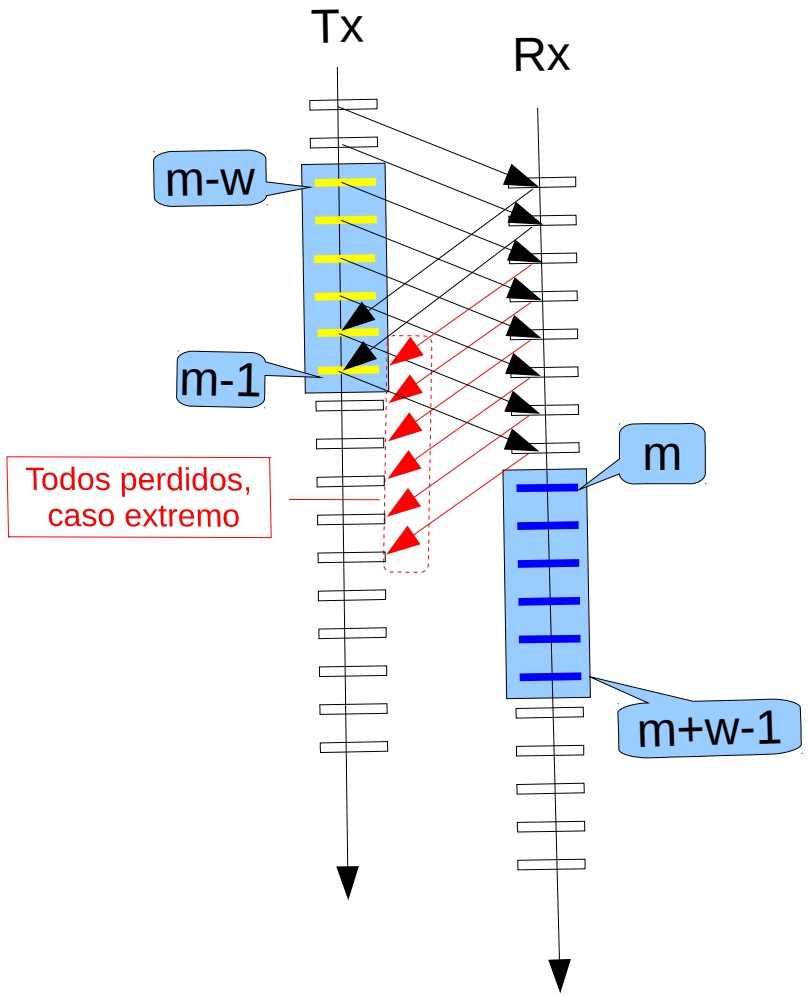
Q: ¿Qué relación debe existir entre el # de sec. y el tamaño de ventana?



- La clave para evitar este problema es impedir que se pueda producir el escenario de la figura adjunta.
- Supongamos que la ventana de recepción es [m,m+w-1], por lo tanto Rx ha recibido y enviado ACK del paquete m-1 y los w-1 paquetes previos a éste.
- Si ninguno de estos ACK han sido recibidos por el Tx la ventana del transmisor será [m-w,m-1].
- Así, el mayor número de secuencia de la ventana del Rx será m+w-1 y el límite inferior de la ventana del Tx será m-w.
- Para que Rx tome el paquete m-w como duplicado, su número de secuencia debe caber fuera de su ventana.
- Luego debemos tener un *rango de números de secuencia k* tan grande como para acomodar (m+w-1)-(m-w)+1=2w números de secuencia, luego k >= 2w.
- Q:¿Qué relación debe existir en el caso Go-Back-N?

Tamaño máximo de ventana en Selective Repeat en más detalle

- Rx espera paquetes en [m,m+w-1]
- Tx habiendo enviado toda su ventana, hace timeout al no recibir los acuses de recibos y re-envía paquete con secuencia m-w.
- Para que m-w sea interpretado como duplicado debo tener números de secuencia distintos para ambas ventanas; luego, # de secuencia debes ser al menos m+w-1-(m-w)+1 = 2w.



Capítulo 3: Continuación

- 3.1 Servicios de la capa transporte
- 3.2 Multiplexing y demultiplexing
- 3.3 Transporte sin conexión: UDP
- 3.4 Principios de transferencia confiable de datos
- 3.5 Transporte orientado a la conexión: TCP
  - Estructura de un segmento
  - Transferencia confiable de datos
  - Control de flujo
  - Gestión de la conexión
- 3.6 Principios del control de congestión
- 3.7 Control de congestión en TCP