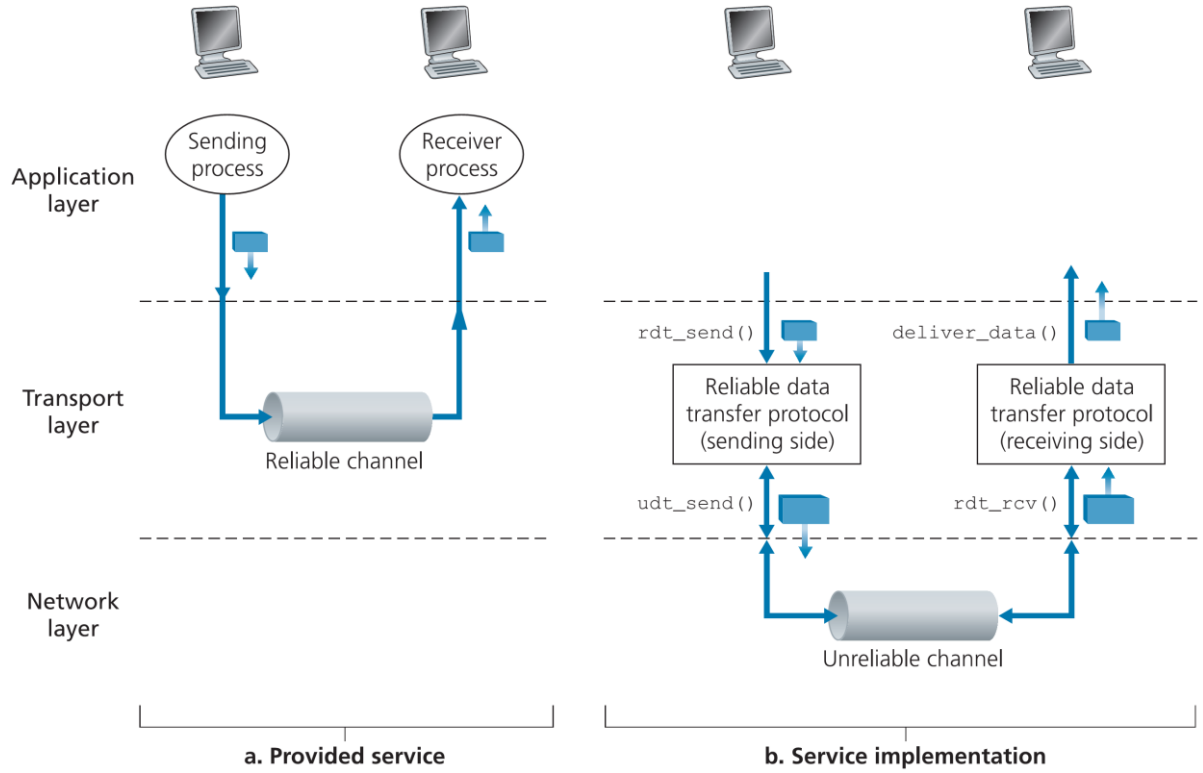
RDT

TRANSFERENCIA FIABLE DE DATOS

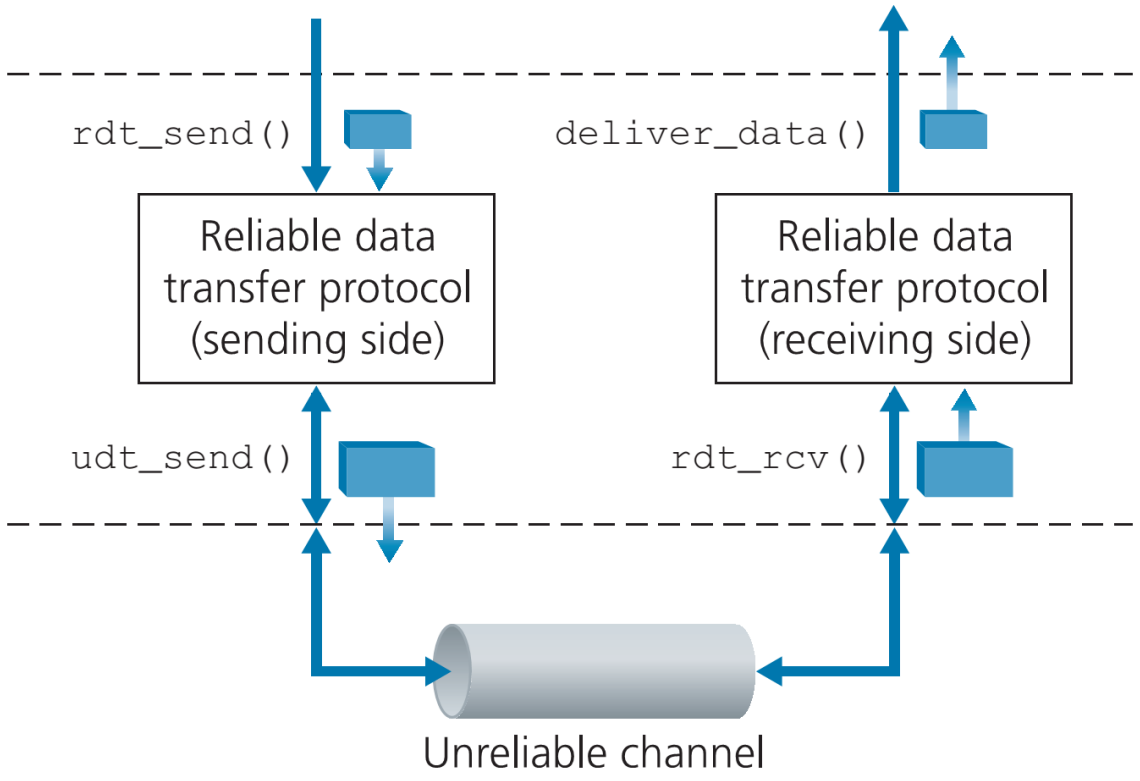
Principios de transferencia de datos fiable

- Importante en las capas de aplicación, transporte y enlace
- En la lista de los 10 tópicos más importantes de redes!



- Las características del canal no fiable determinan la complejidad del protocolo RDT

Transferencia de datos fiable: comenzando



rdt_send(): llamado de arriba, (e.g., por una aplicación). Pasa datos para entregar a la capa superior del receptor

deliver_data(): llamado por rdt para entregar datos a la capa superior

udt_send(): llamado por rdt, para transferir un paquete sobre un canal no fiable al receptor

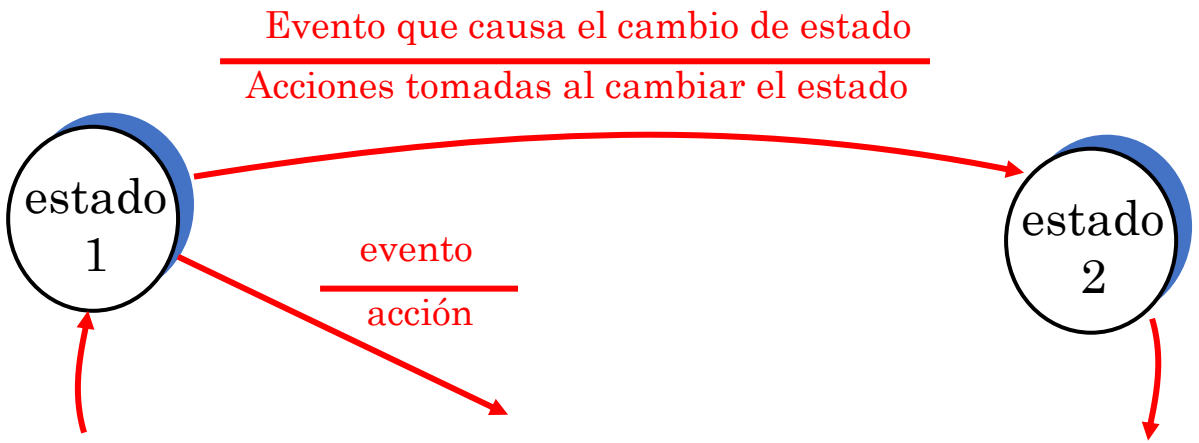
rdt_rcv(): llamado cuando llega un paquete al lado receptor del canal

Transferencia de datos fiable: comenzando

¿Qué haremos? :

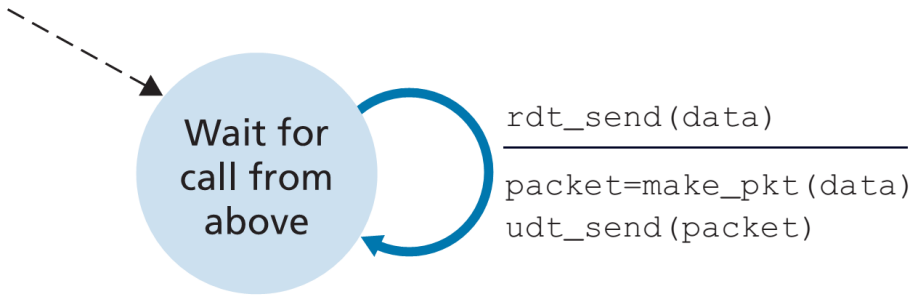
- Desarrollar incrementalmente los extremos emisor y receptor del protocolo de transferencia de datos fiable (rdt)
- Considerar solo la transferencia de datos unidireccional
  - Pero la información de control fluirá en ambas direcciones!
- Usar máquinas de estado finito (FSM) para especificar el emisor y el receptor

estado: estando en este "estado", el siguiente estado se determina unívocamente por el siguiente evento

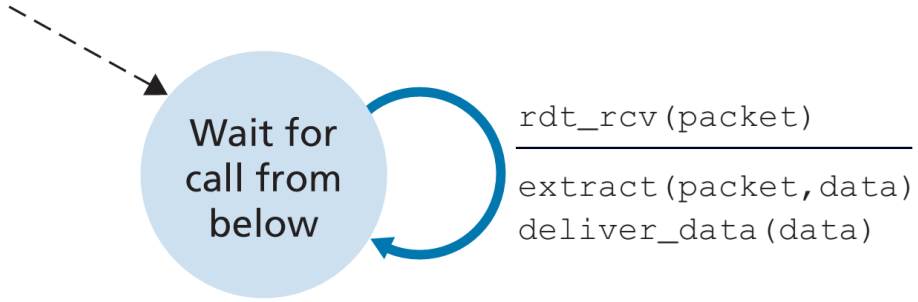


Rdt1.0: Transferencia fiable sobre un canal fiable

- Canal subyascente perfectamente fiable
  - No hay errores de bit
  - No hay pérdida de paquetes
- FSMs separados para emisor, receptor:
  - Emisor envia datos al canal subyascente
  - Receptor lee datos del canal subyascente



emisor



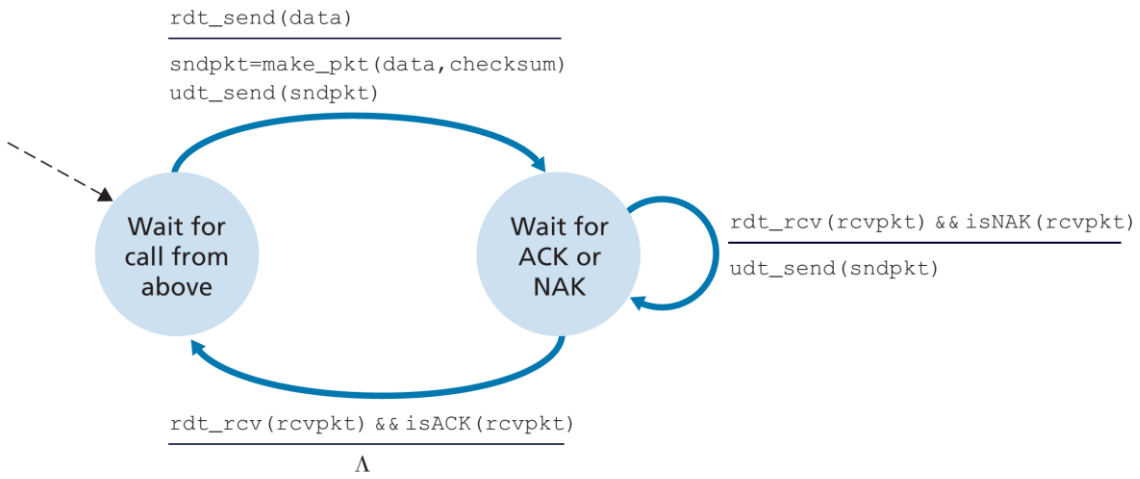
receptor

Rdt2.0: canal con errores de bit

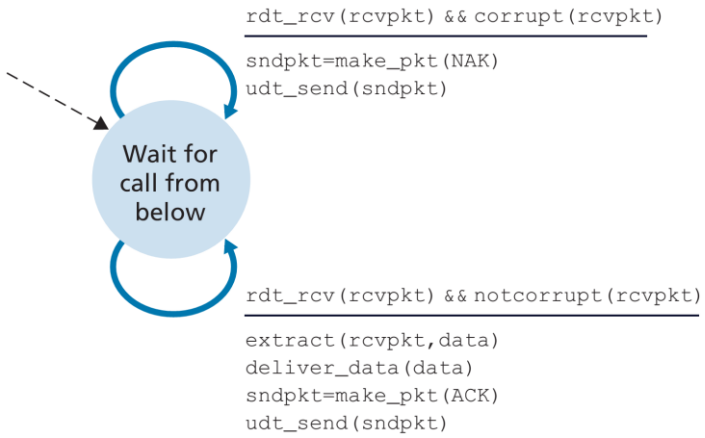
- Canal subyascente puede invertir los bits en un paquete
  - checksum para detectar errores de bit
- *La pregunta:* como recuperarse de errores:
  - *confirmación (ACKs):* el receptor explícitamente le dice al emisor que un paquete se recibio bien
  - *Confirmación negativa (NAKs):* el receptor explícitamente le dice al emisor que el paquete tuvo errores
  - El emisor retransmite el paquete al recibir un NAK
- Nuevos mecanismos en ***rdt2.0*** (sobre ***rdt1.0***):
  - Detección de errores
  - Retroalimentación del receptor: mensajes de control (ACK, NAK) del receptor al emisor

rdt2.0: Especificación FSM

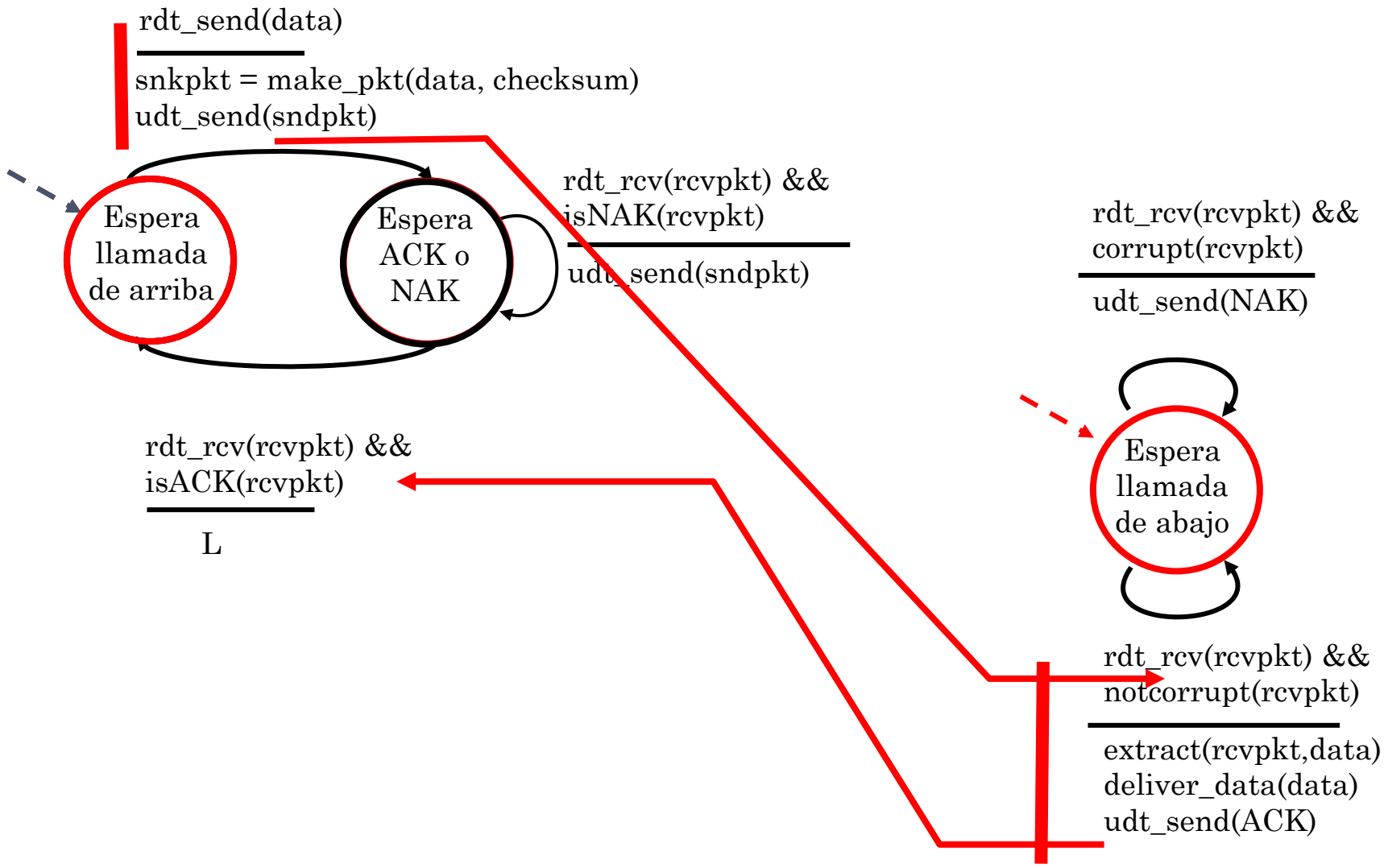
emisor



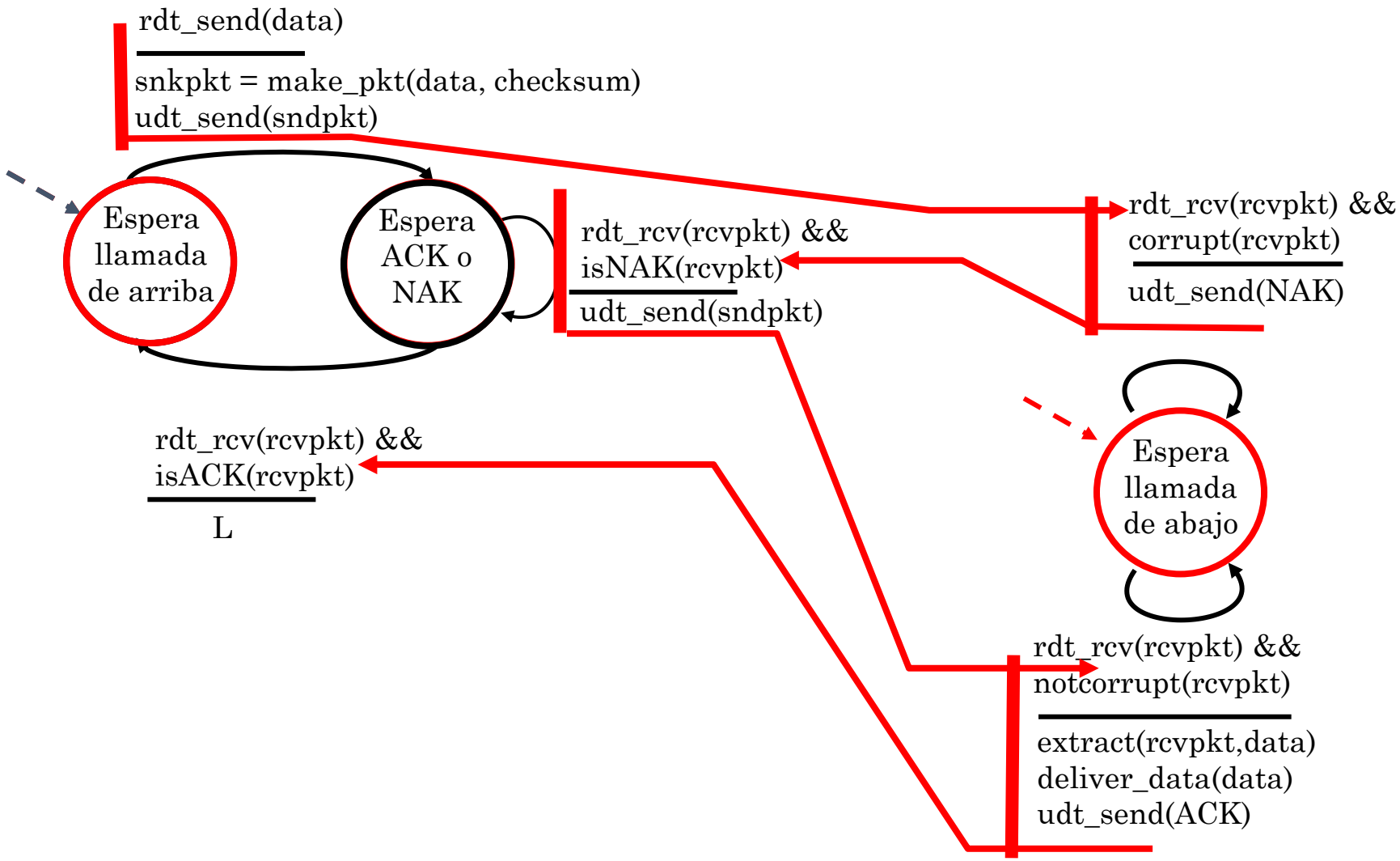
receptor



rdt2.0: operación sin errores



rdt2.0: Escenario con error



rdt2.0 tiene una falla fatal!

¿Qué ocurre si los ACK/NAK se corrompen?

- El emisor no sabe que ocurrió en el receptor!
- No puede retransmitir simplemente: posible duplicado

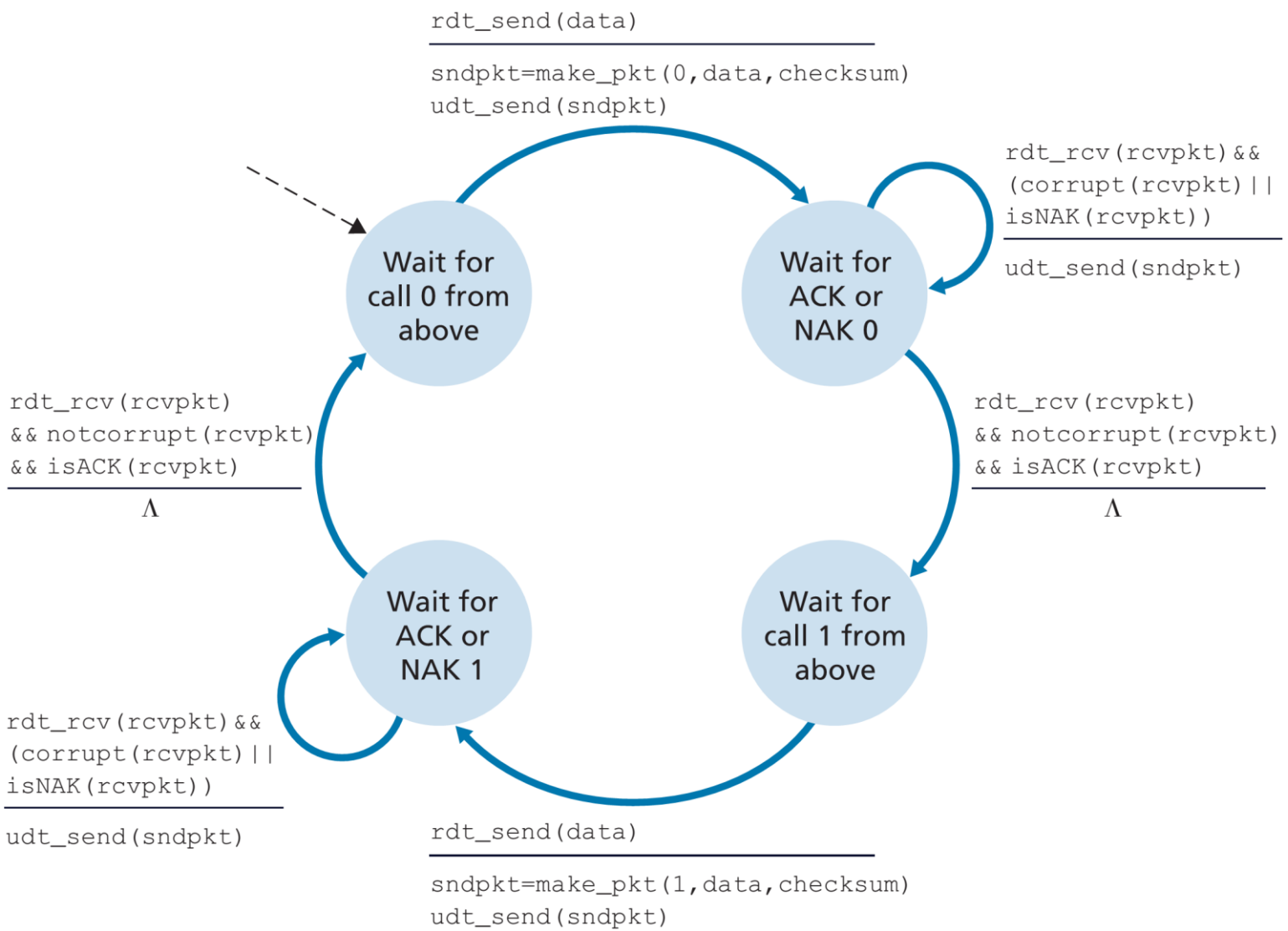
Manejar duplicados:

- El emisor retransmite el paquete actual si el ACK/NAK está dañado
- El emisor agrega un *número de secuencia* a cada paquete
- El receptor descarta (no entrega hacia arriba) el paquete duplicado

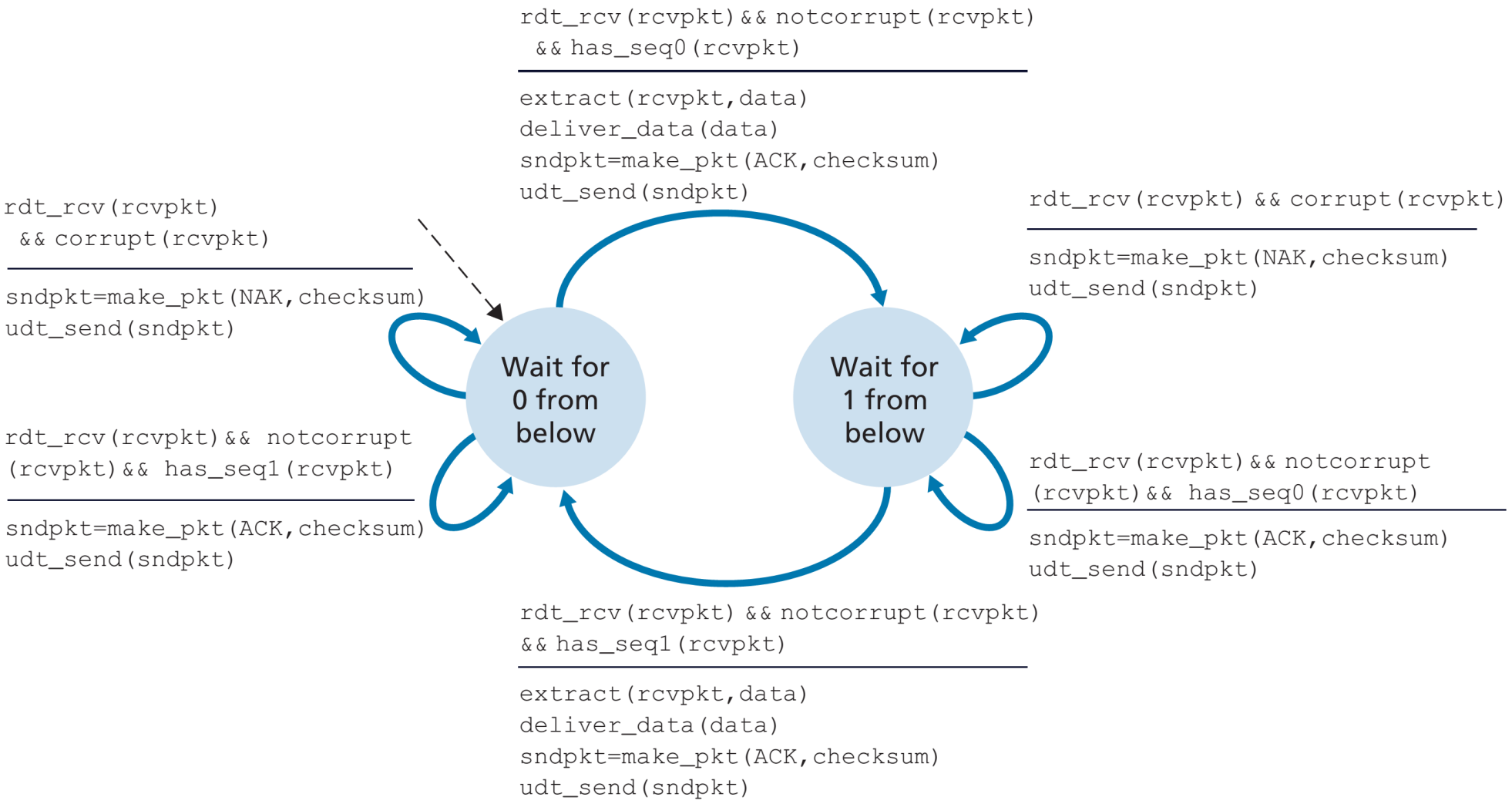
Parada y espera (Stop and wait)

El emisor envia un paquete y espera la respuesta del receptor

rdt2.1: El emisor maneja ACK/NAKs dañados



rdt2.1: el receptor maneja ACK/NAKs dañados



rdt2.1: Discusión

Emisor:

- #sec agregado al paquete
- Dos #sec (0,1) serán suficientes. ¿Porqué?
- Debe verificar si el ACK/NAK recibido esta dañado
- Dos veces mas estados
  - Un estado debe “recordar” si el paquete “actual” tiene #sec 0 o 1

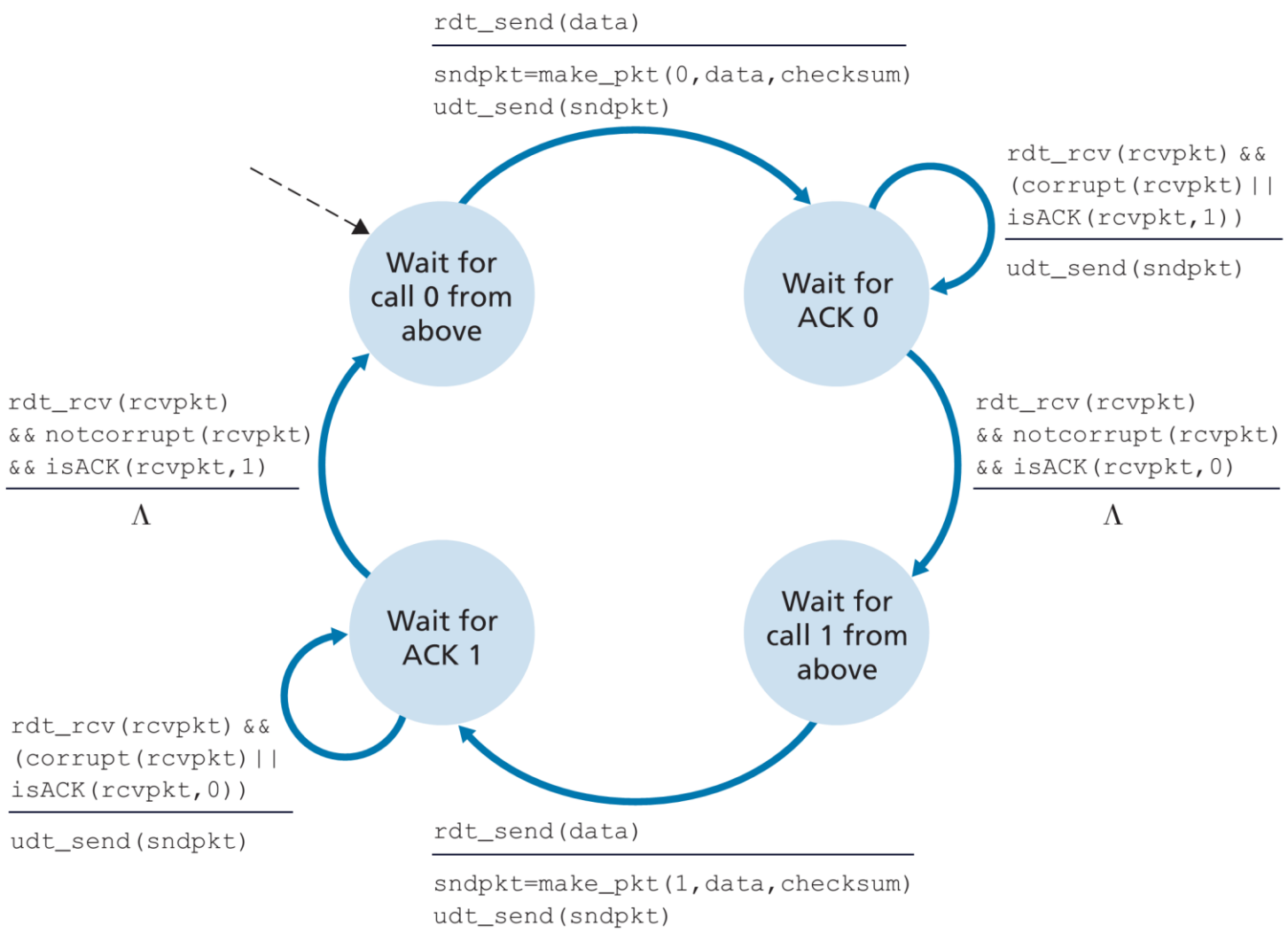
Receptor:

- Debe verificar si el paquete recibido es duplicado
  - El estado indica si el #sec esperado es 0 o 1
- **Nota**: el receptor no puede saber si su último ACK/NAK fue recibido correctamente en el emisor

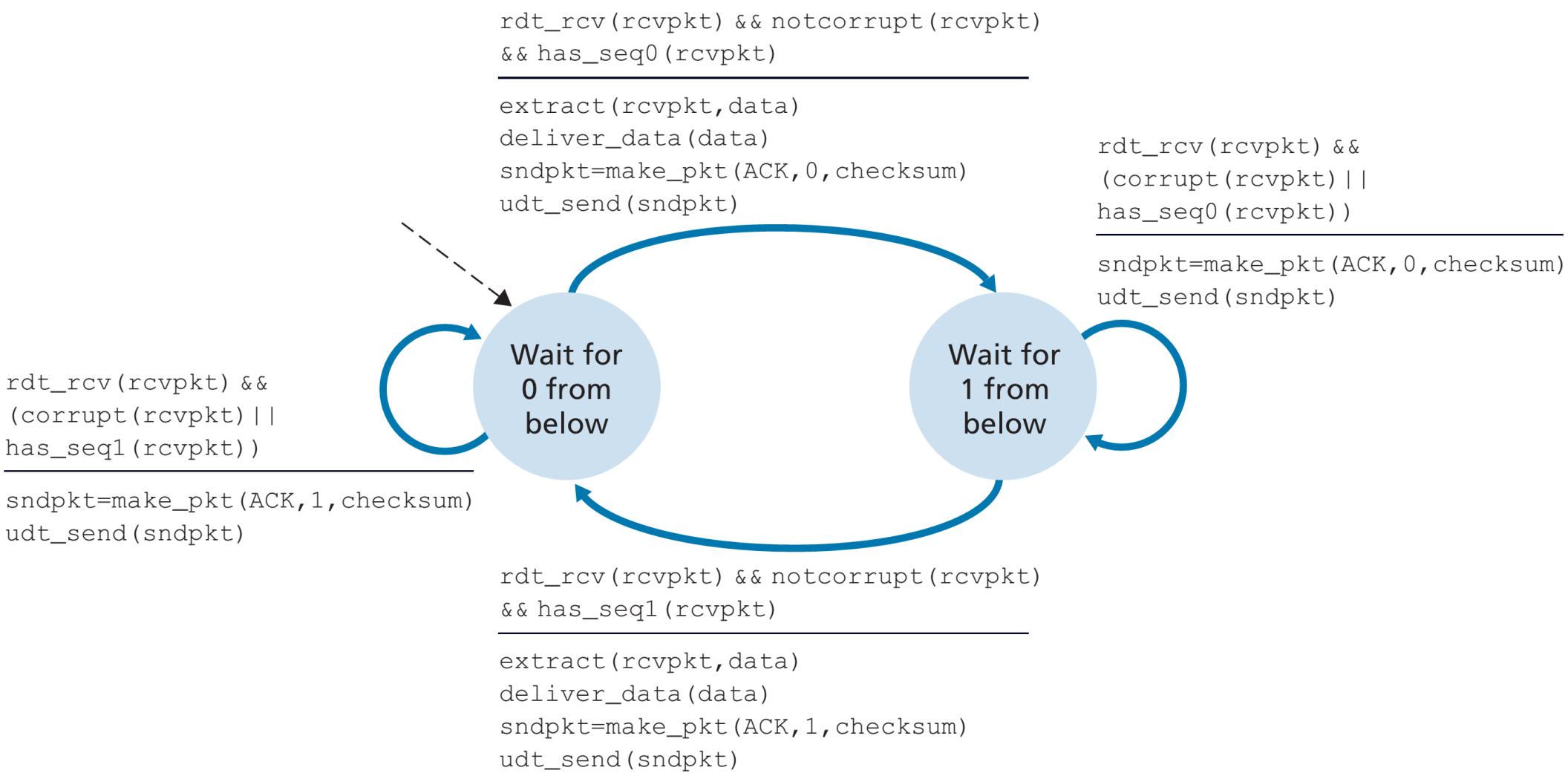
rdt2.2: Un protocolo libre de NAK

- La misma funcionalidad que rdt2.1, usando solo ACKs
- En vez de un NAK, el receptor envía un ACK por el último paquete recibido OK
  - El receptor debe incluir explícitamente el #sec del paquete confirmado
- Un ACK duplicado en el emisor resulta en la misma acción que un NAK: se retransmite el paquete actual

rdt2.2: Emisor



rdt2.2: Receptor



rdt3.0: Canal con errores y pérdida

Nueva presunción:

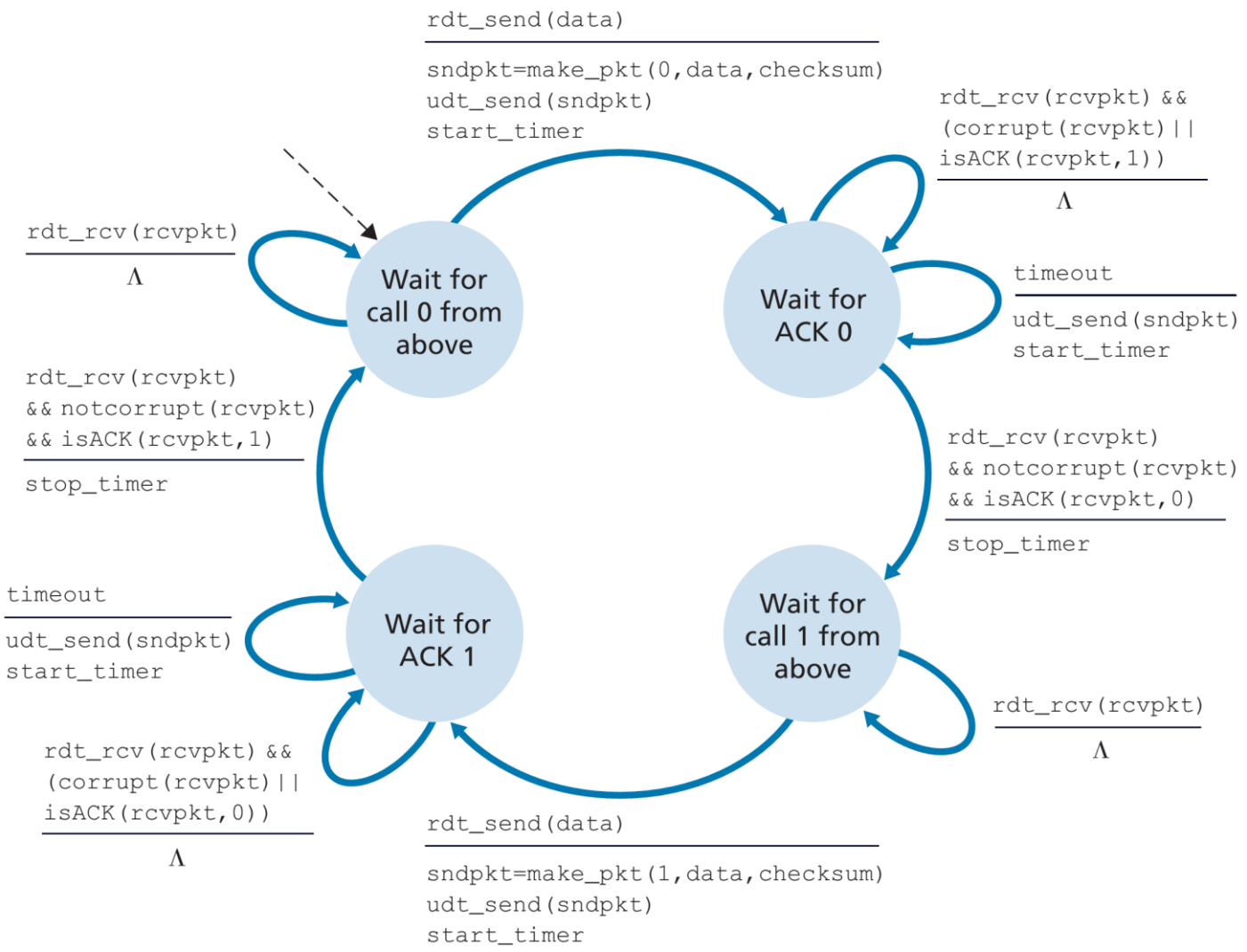
El canal subyacente puede también perder paquetes de datos o ACKs

- Suma de verificación, #sec, ACKs y retransmisiones son de ayuda, pero no suficientes

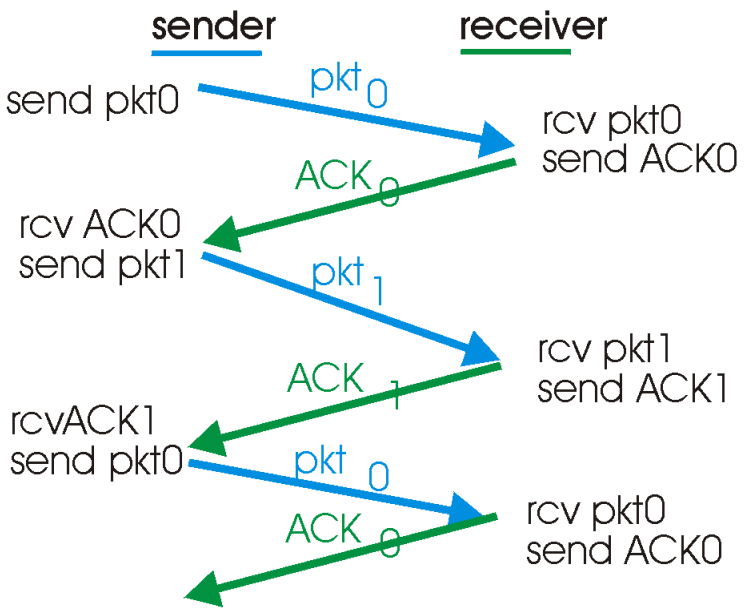
Enfoque: el emisor espera por un tiempo "razonable" un ACK

- Retransmite si no se recibe ningún ACK en ese tiempo
- Si el paquete (o ACK) solo se retrasó (no se perdió) :
  - La retransmisión generará un duplicado, pero el uso de #sec resuelve este problema
  - El receptor debe especificar #sec del paquete confirmado
- Requiere de un contador descendente

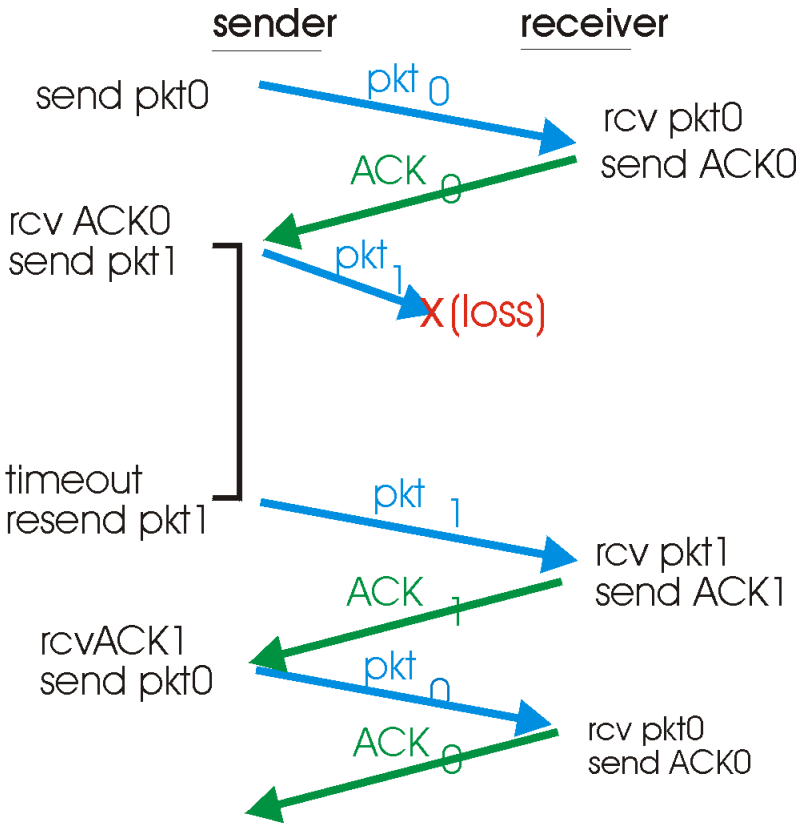
rdt3.0 emisor



rdt3.0 en acción

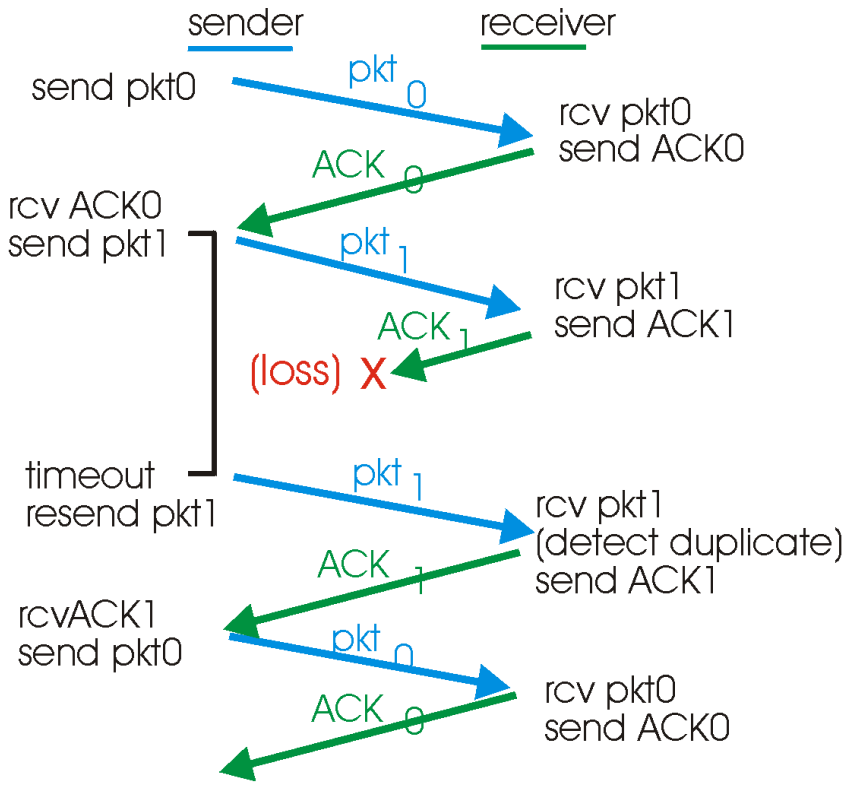


(a) operation with no loss

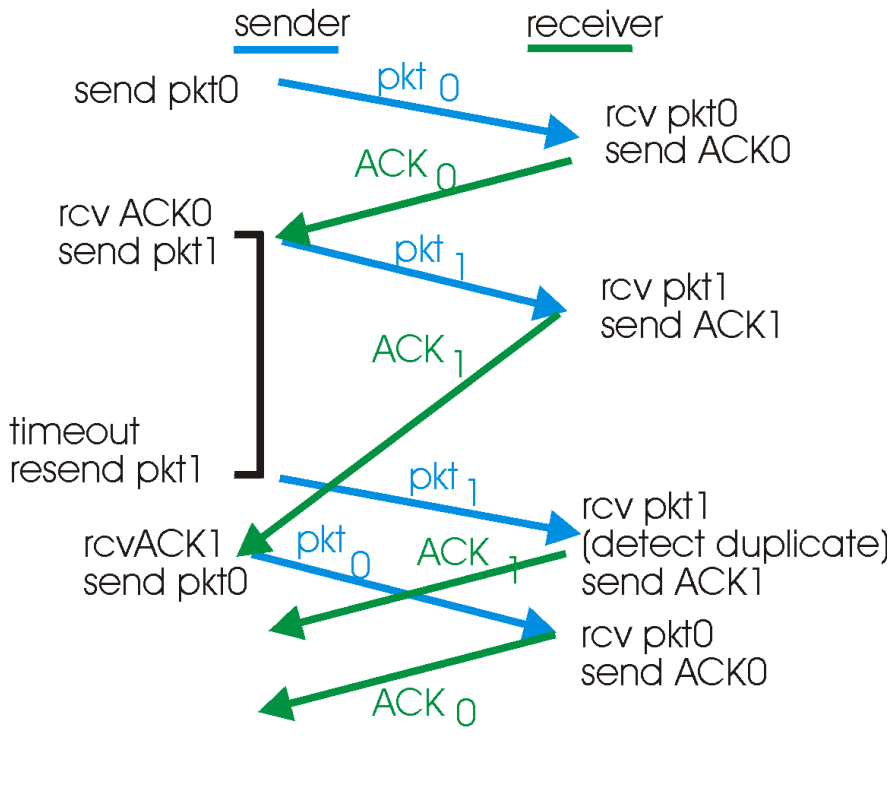


(b) lost packet

rdt3.0 en acción



(c) lost ACK



(d) premature timeout

Desempeño de rdt3.0

- rdt3.0 funciona, pero el desempeño es bajo
- Ejemplo:

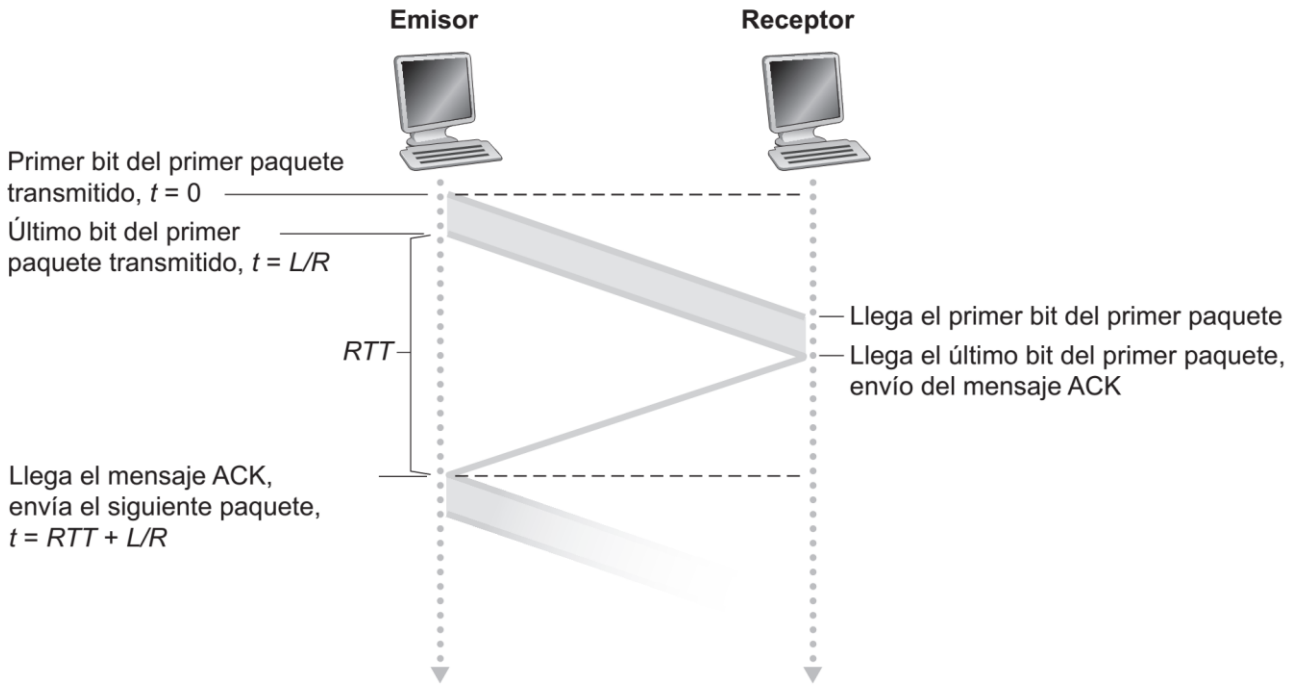
  En un enlace de 1 Gbps, con retardo de propagación de 15 ms, y tamaño de paquete de 8000 bits:

$$d_{trans} = \frac{L}{R} = \frac{8000b}{10^9 bps} = 8\mu s$$

Utilización del emisor

- U_{emisor} : Fracción de tiempo que el emisor está ocupado enviando
- *RTT* : Tiempo de ida y vuelta (Round Trip Time)

rdt3.0: operación de parada y espera



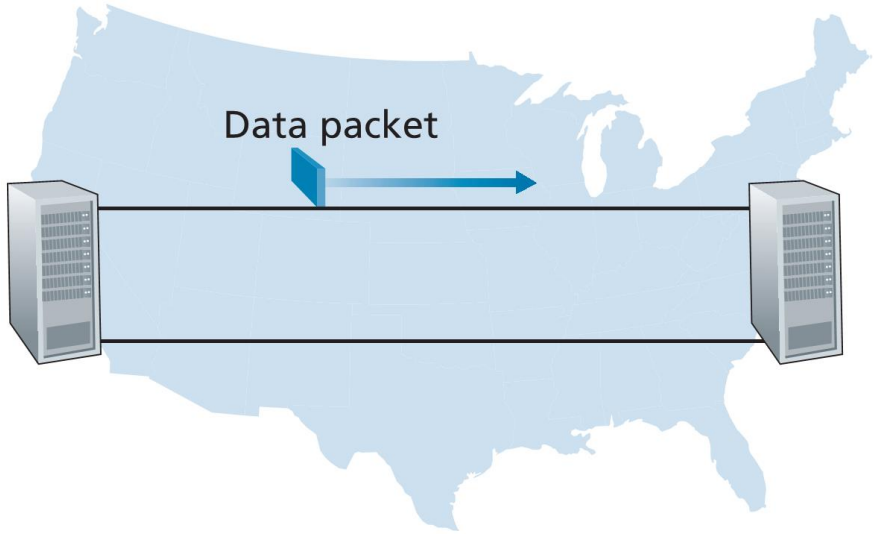
$$U_{emisor} = \frac{L/R}{RTT + L/R} = \frac{0.008ms}{30.008ms} = 0.00027$$

- 1 PDU de 1KB cada 30 ms → 33KBps (267Kbps) aún sobre un enlace de 1Gbps
- El protocolo de red limita el uso de los recursos físicos!

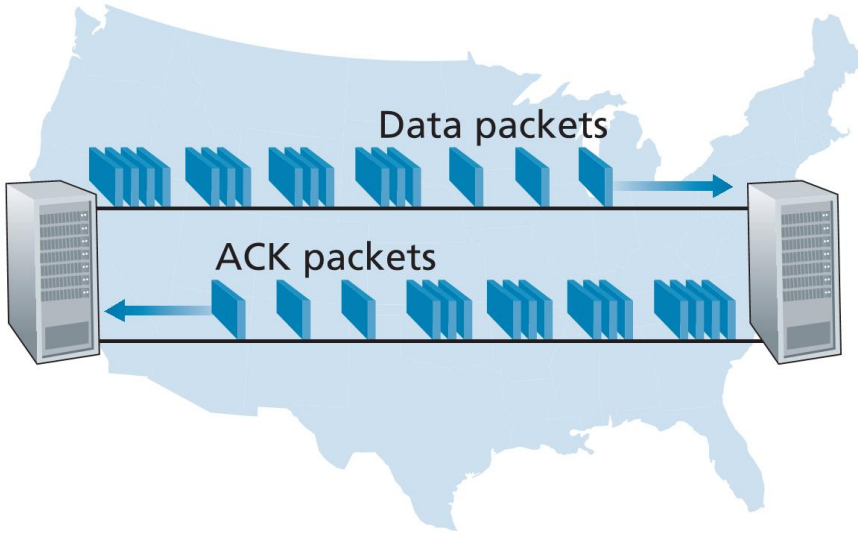
Protocolos entubados (encauzados)

Encauzamiento: el emisor permite múltiples paquetes, “en camino”, por confirmar

- El rango de números de secuencia debe incrementarse
- Buffers en el emisor y receptor



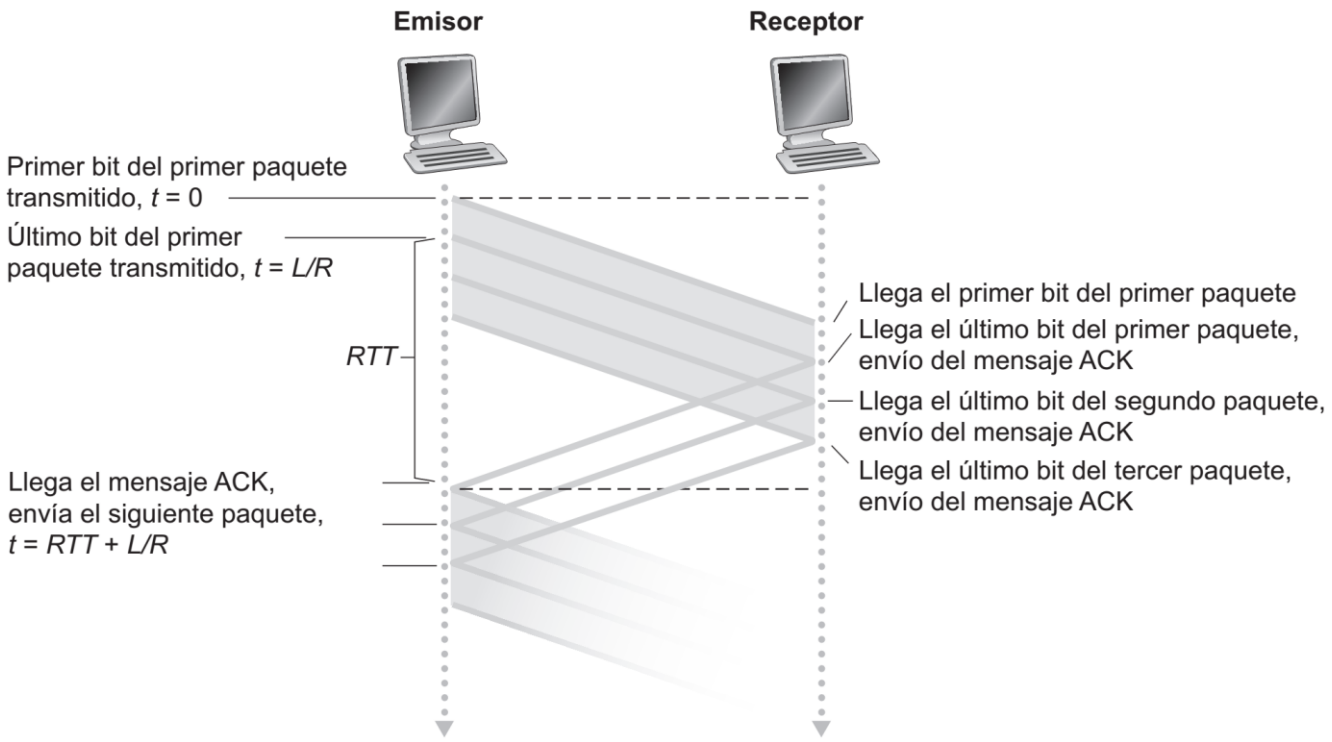
a. A stop-and-wait protocol in operation



b. A pipelined protocol in operation

- Dos formas genéricas de protocolos encauzados: *regresar a N, repetición selectiva*

Encauzamiento: utilización mejorada



$$U_{emisor} = \frac{3 \times (L/R)}{RTT + L/R} = \frac{0.024}{30.008} = 0.0008$$

Incrementa la utilización en un factor de 3!

Protocolos encauzados

Regresar a N:

- El emisor puede tener hasta N paquetes sin confirmar en el cauce
- El receptor solo envía ACKs acumulativos
  - No confirma un paquete si existe un espacio
- El emisor tiene un temporizador para el paquete mas viejo sin confirmar
  - Si el temporizador expira, retransmite todos los paquetes no confirmados

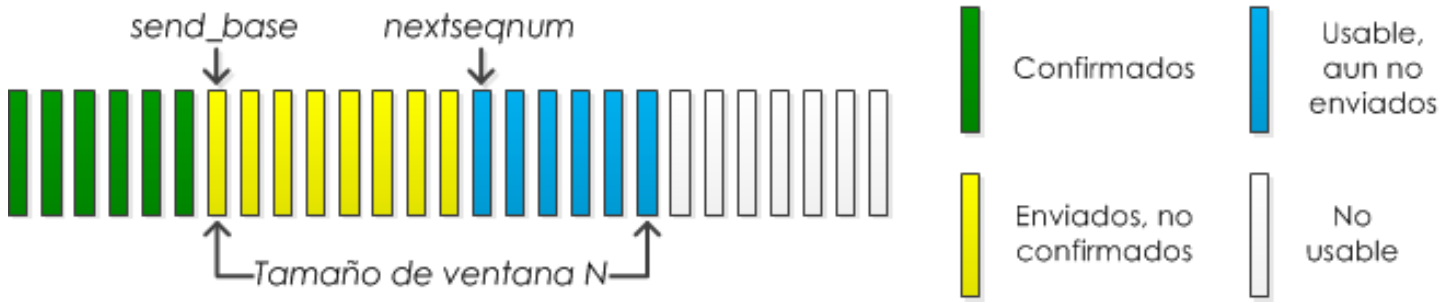
Repetición selectiva:

- El emisor puede tener hasta N paquetes sin confirmar en el cauce
- El receptor confirma paquetes individuales
- El emisor mantiene un temporizador por cada paquete sin confirmar
  - Cuando expira el temporizador, retransmite solo el paquete no confirmado

Regresar a N

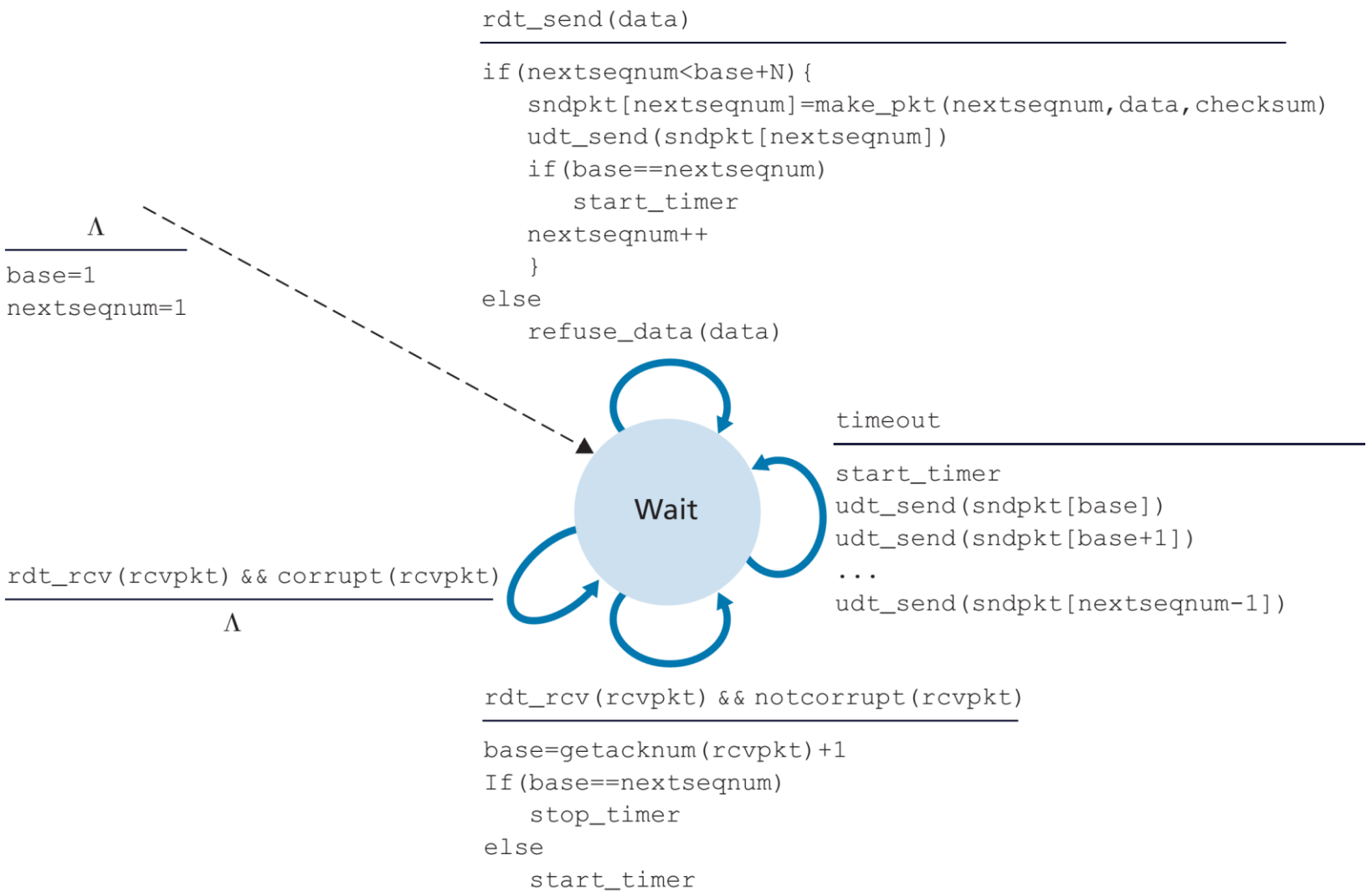
Emisor:

- #sec de k bits en encabezado de paquete
- “Ventana” de hasta N, paquetes consecutivos sin confirmar permitidos

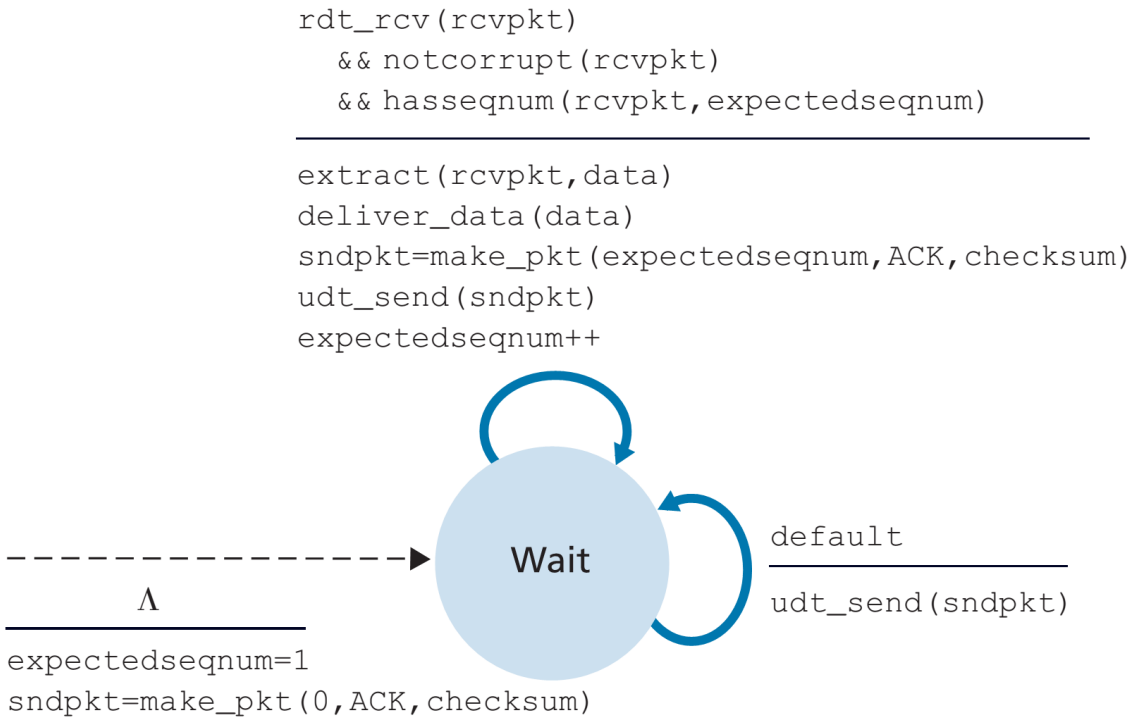


- ACK(n): Confirma todos los paquetes hasta *#sec n* inclusive – “ACK acumulativo”
  - Puede recibir ACKs duplicados (ver receptor)
- Timer para cada paquete en camino
- *timeout(n):* Retransmite el paquete *n* y todos los paquetes con *#sec* mayor en la ventana

Regresar a N: FSM extendido del emisor

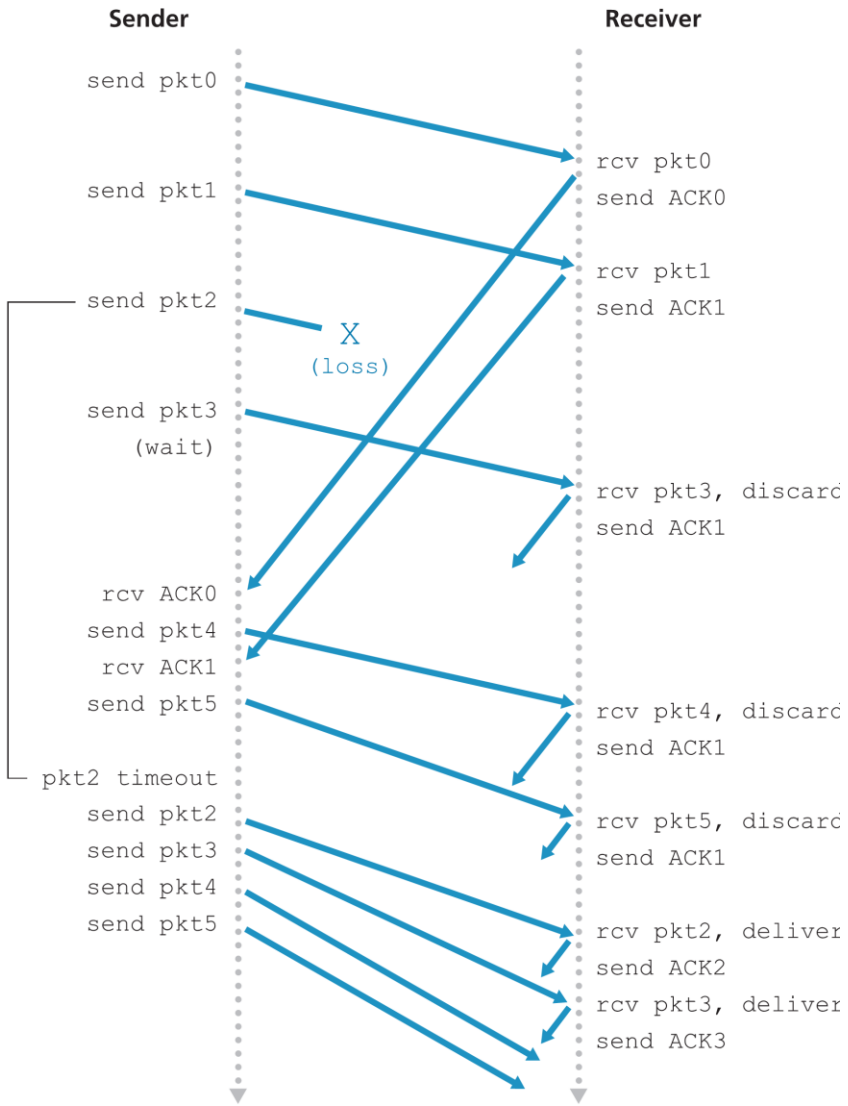


Regresar a N: FSM extendido del receptor



- ACK-only: siempre envía un ACK por paquete correctamente recibido con el mayor #sec *en orden*
  - Puede generar ACKs duplicados
  - Solo necesita recordar **expectedseqnum**
- Paquetes fuera de orden:
  - descartar (no poner en buffer) -> no requiere buffer en el receptor!
  - Reconfirmar el paquete con el mayor #sec en orden

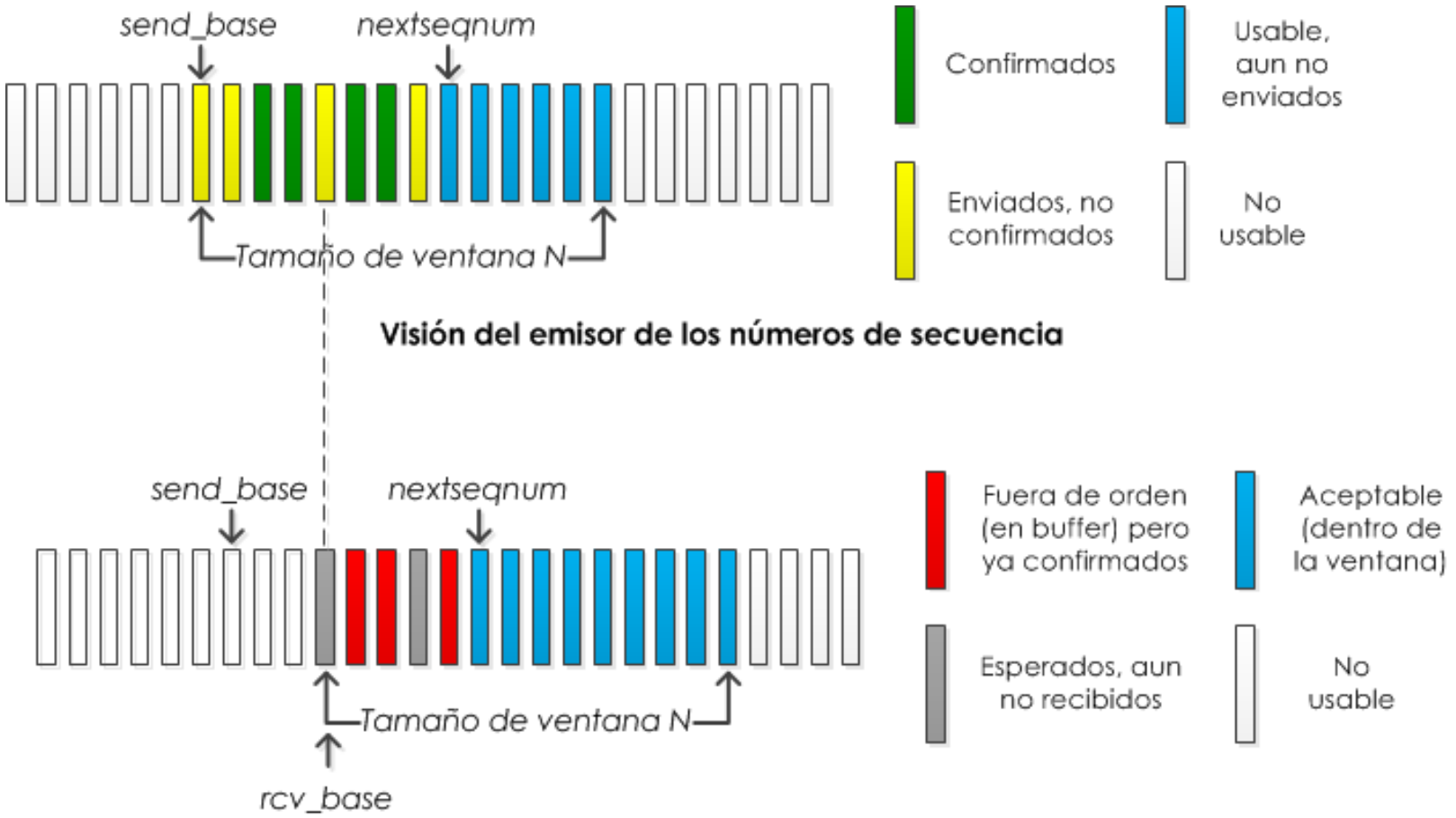
Regresar a N en acción



Repetición selectiva

- El receptor confirma individualmente todos los paquetes correctamente recibidos
  - Pone en buffer los paquetes, según se requiera, para su eventual entrega ordenada a la capa superior
- El emisor solo reenvía los paquetes para los cuales no se recibió un ACK
  - Un timer por cada paquete no confirmado en el emisor
- Ventana del emisor
  - N #sec consecutivos
  - Limita los #sec de paquetes enviados sin confirmar

Repetición selectiva: ventanas de emisor y receptor



Repetición selectiva

Emisor

Datos desde arriba :

- Si el siguiente #sec disponible esta en la ventana, enviar paquete

Timeout(n):

- Reenviar paquete n, reiniciar timer

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

- Marcar paquete n como recibido
- Si n es el paquete menor sin confirmar, avanzar la base de la ventana al siguiente #sec sin confirmar

Receptor

Paquete n en [rcvbase, rcvbase+N-1]

- enviar ACK(n)
- Fuera de orden: buffer
- En orden: entregar (también entregar paquetes ordenados en buffer), avanzar la ventana al siguiente paquete por recibir

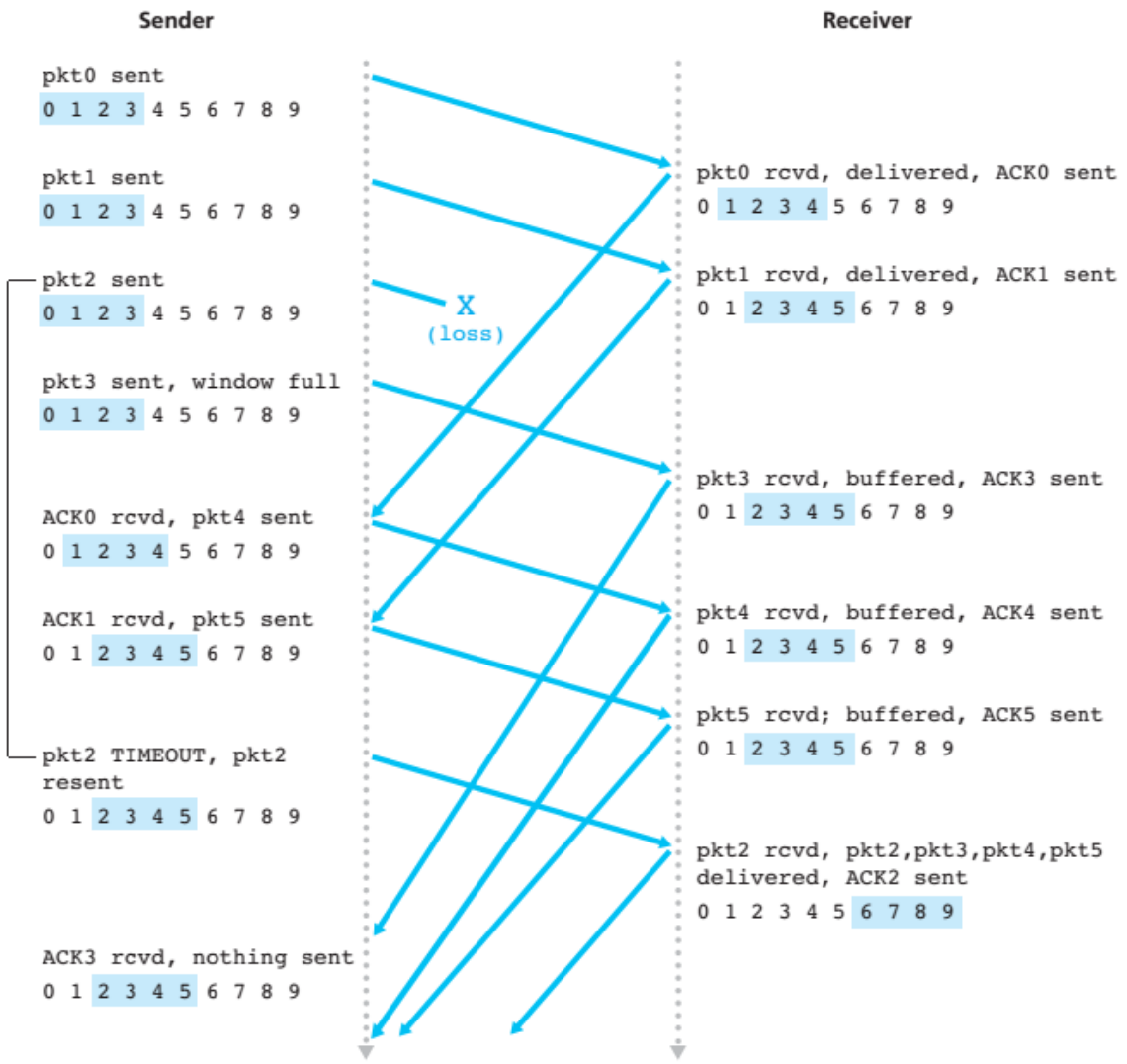
Paquete n en [rcvbase-N,rcvbase-1]

- Paquete duplicado. Enviar ACK(n)

En otro caso:

- Ignorar paquete

Repetición selectiva en acción



Repetición selectiva: dilema

Ejemplo:

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

- El receptor no ve ninguna diferencia en los dos escenarios!
- Erroneamente pasa datos duplicados como nuevos en (a)

Preg: ¿Que relación existe entre el tamaño de #sec y el tamaño de la ventana?