

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

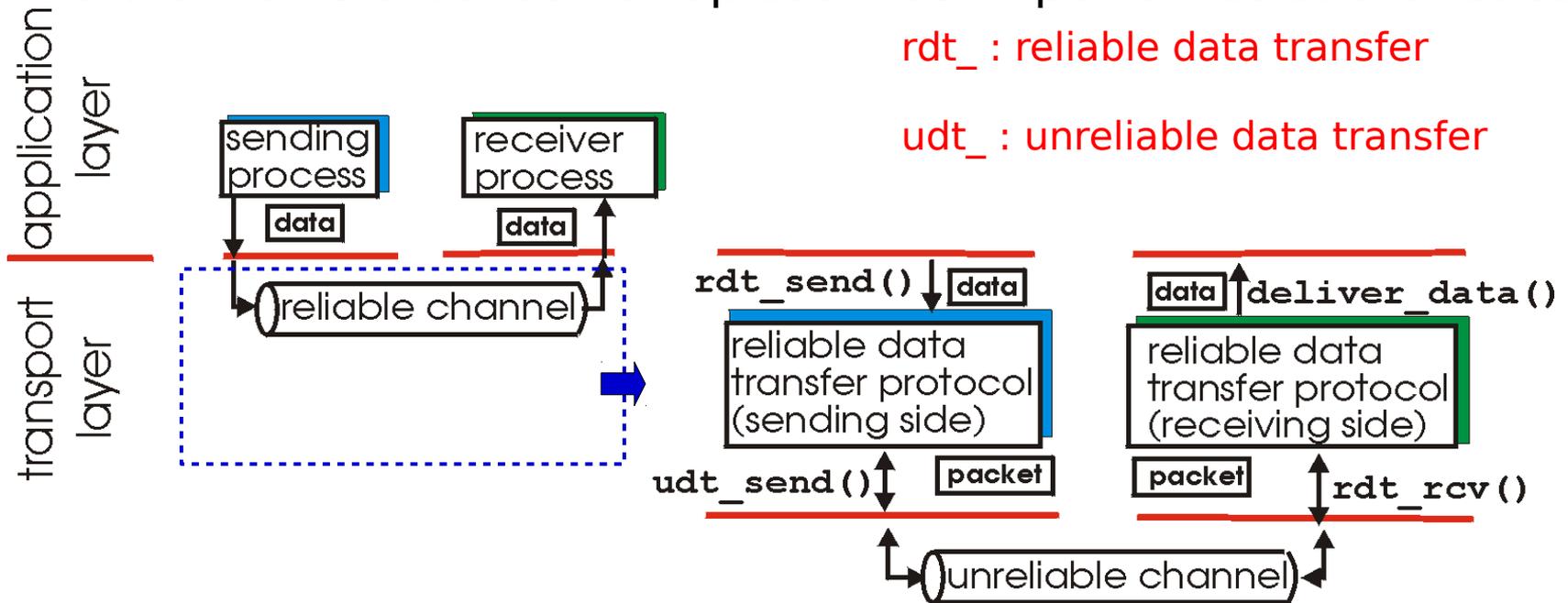
- ❑ 3.1 Servicios de la capa transporte
- ❑ 3.2 Multiplexing y demultiplexing
- ❑ 3.3 Transporte sin conexión: UDP
- ❑ 3.4 Principios de transferencia confiable de datos
- ❑ 3.5 Transporte orientado a la conexión: TCP
 - Estructura de un segmento
 - Transferencia confiable de datos
 - Control de flujo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

Principios de transferencia confiable de datos

- ❑ **Es un tópico importante en capas de aplicación, transporte y enlace de datos**
- ❑ Está en la lista de los 10 tópicos más importantes sobre redes !

rdt_ : reliable data transfer

udt_ : unreliable data transfer



(a) provided service

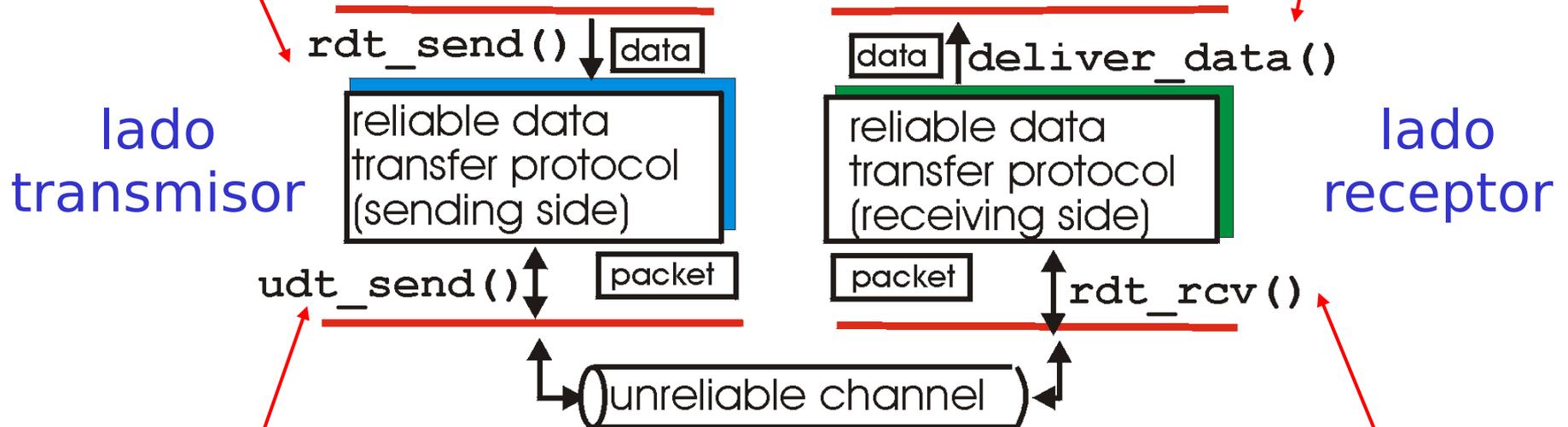
(b) service implementation

- ❑ 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

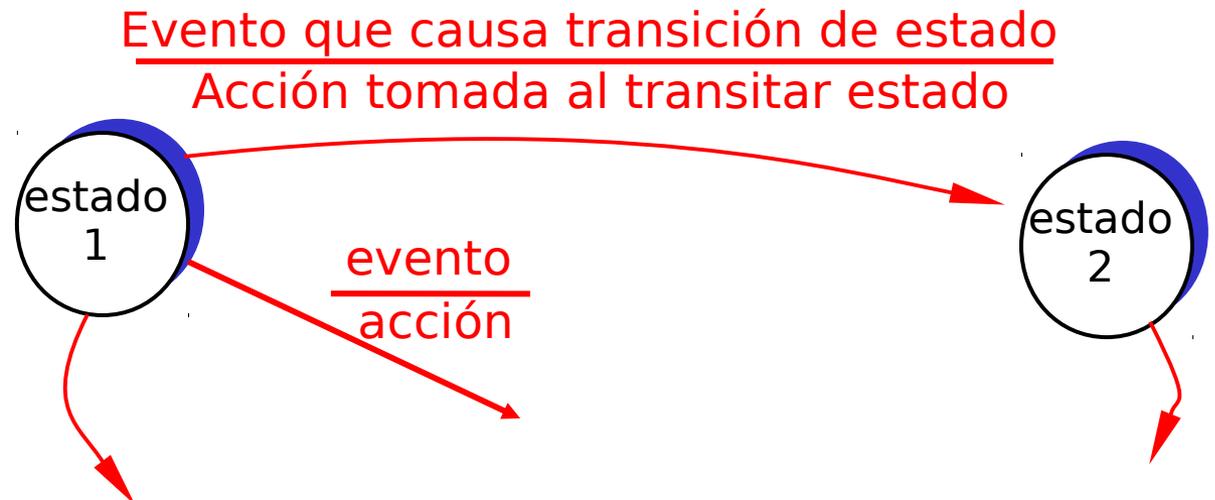
rdt_rcv(): llamada cuando un paquete llega al lado receptor

Transferencia confiable de datos: primeros aspectos

Pasos a seguir:

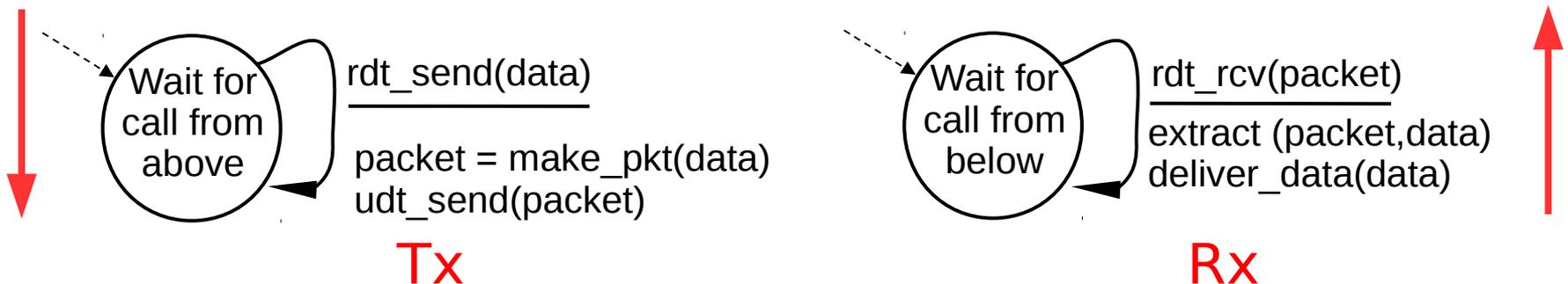
- ❑ Desarrollaremos incrementalmente los lados Tx y Rx del protocolo de transferencia confiable (rdt)
- ❑ Consideraremos sólo transferencias de datos unidireccionales
 - Pero la información de control fluirá en ambas direcciones!
- ❑ Usaremos máquina de estados finitos (Finite State Machine) para especificar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



Rdt2.0: Canal con **bits errados**

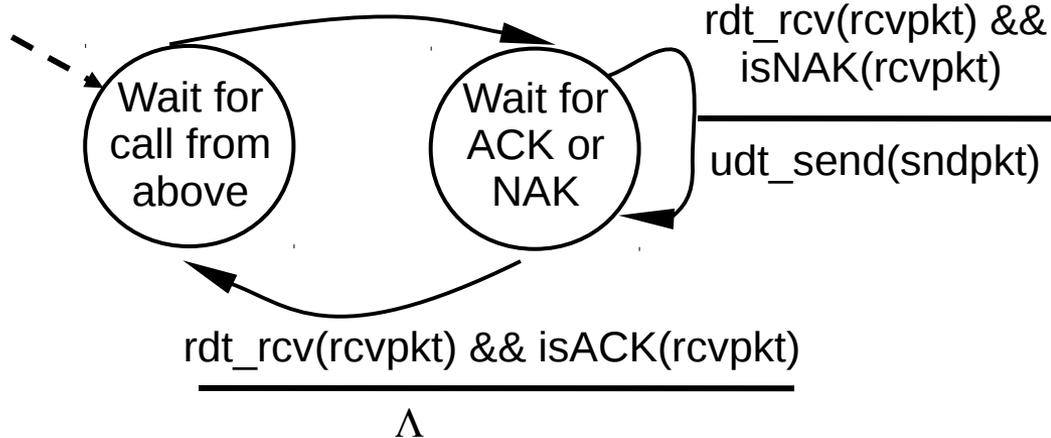
- Canal inferior puede invertir bits del paquete
 - Usamos checksum para detectar los errores de bits
 - Supondremos que no se pierden paquetes ni hay desorden
- La pregunta: ¿Cómo recuperarnos de errores?:
 - *acknowledgements (ACKs)*: - *acuses de recibo*: receptor explícitamente le dice al Tx que 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

rdt_send(data)

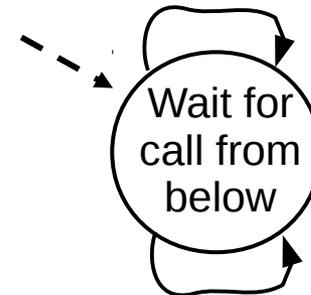
sndpkt = make_pkt(data, checksum)
 udt_send(sndpkt)



$\Lambda \equiv$ hacer nada

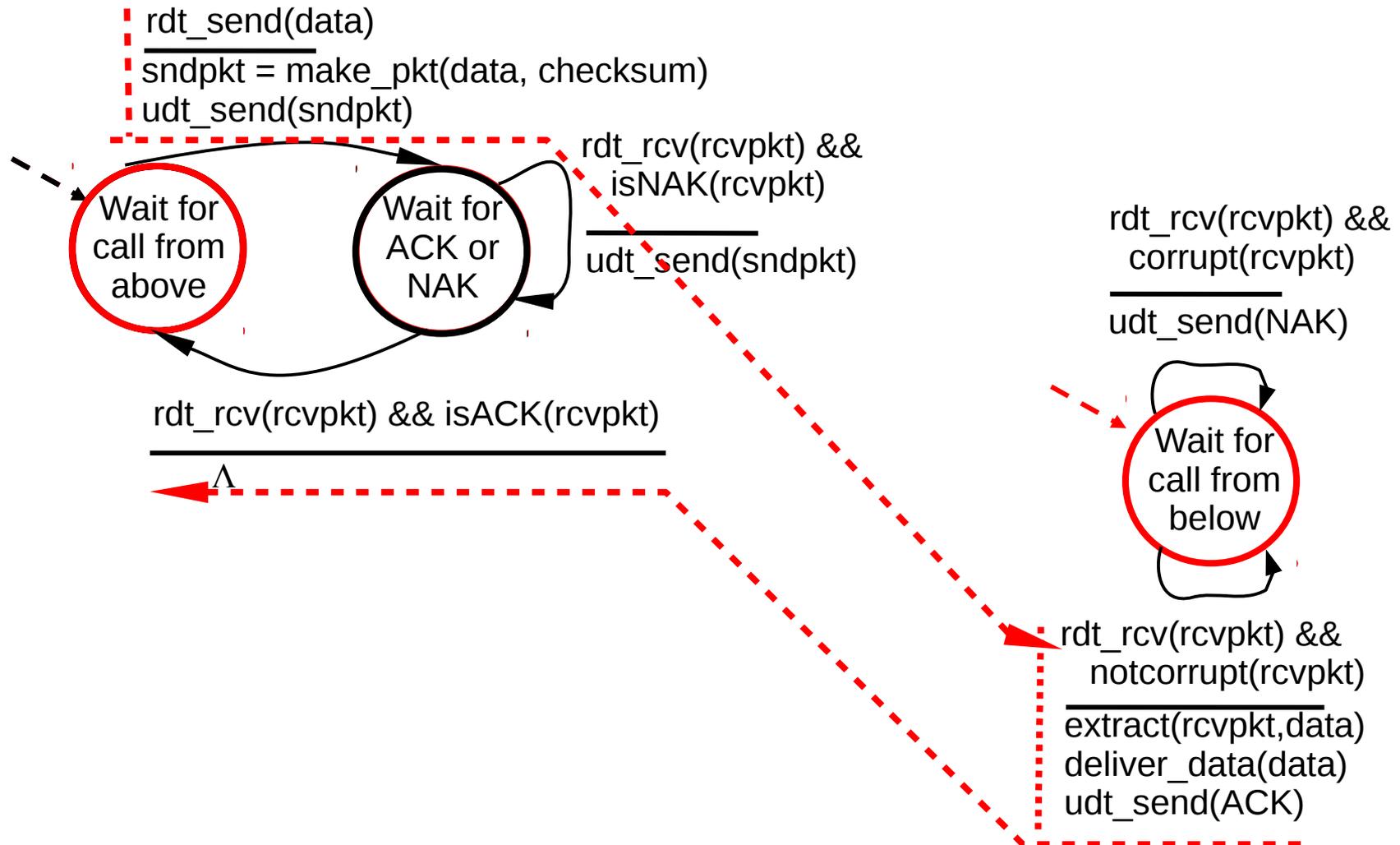
Rx

rdt_rcv(rcvpkt) && corrupt(rcvpkt)
 udt_send(NAK)

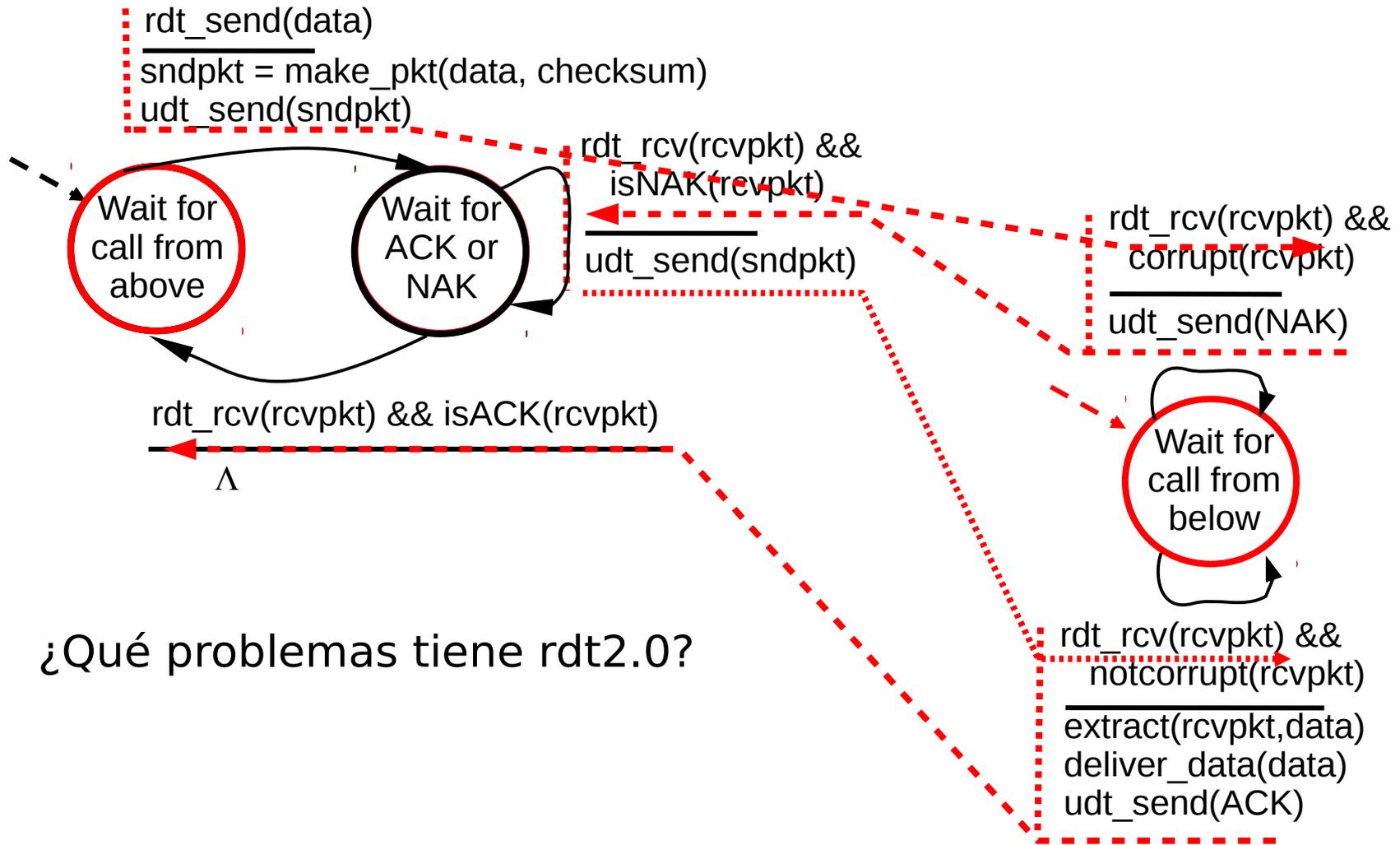


rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
 extract(rcvpkt, data)
 deliver_data(data)
 udt_send(ACK)

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?

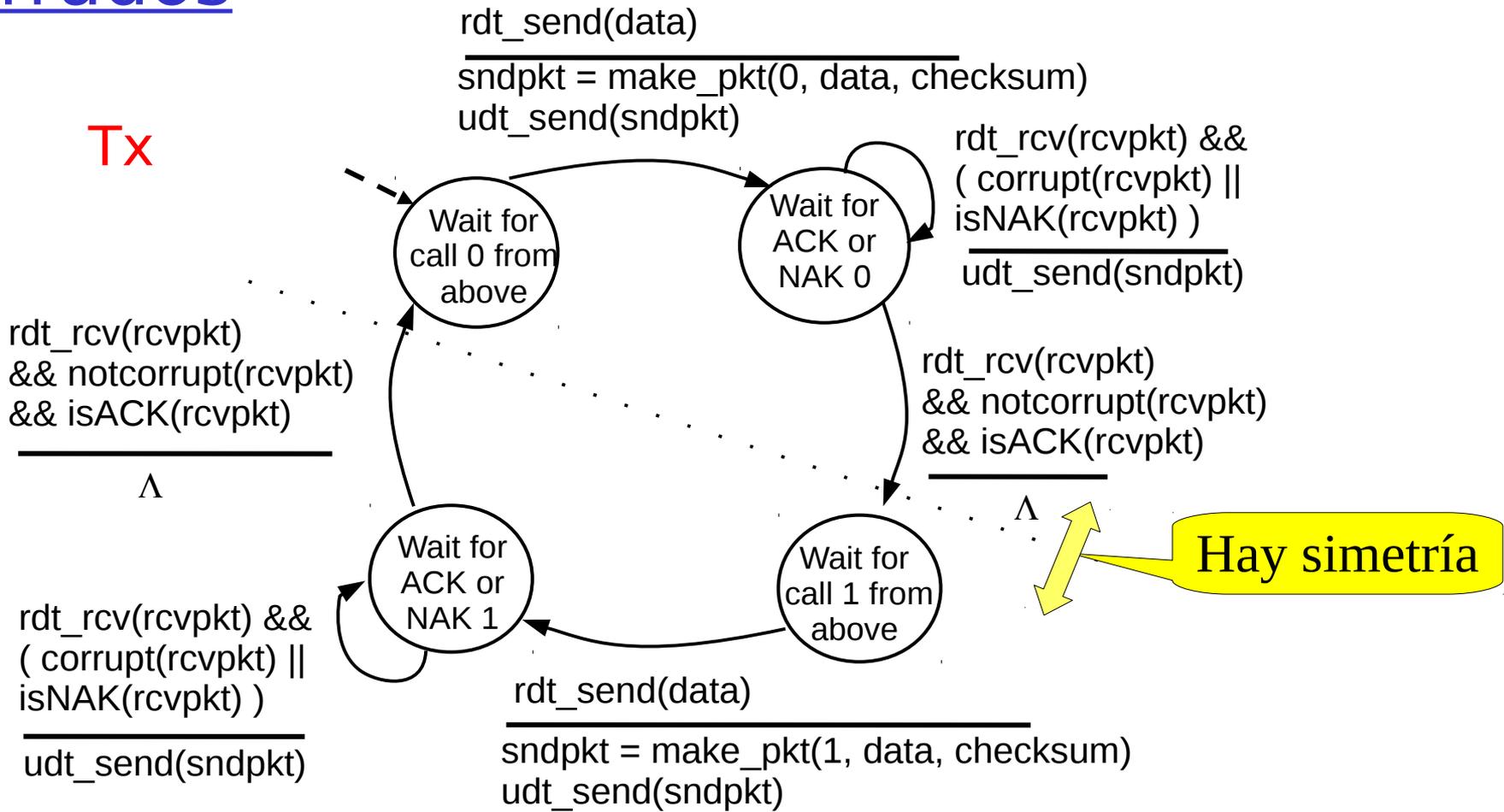
- ❑ Tx no sabe qué pasó en el receptor!
- ❑ Idea, retransmitir paquete ante la llegada de un ACK o NAK dañado.
- ❑ No puede sólo retransmitir: generaría posible duplicado
- ❑ Surge necesidad de poner números de secuencia para detectar duplicados.

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

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

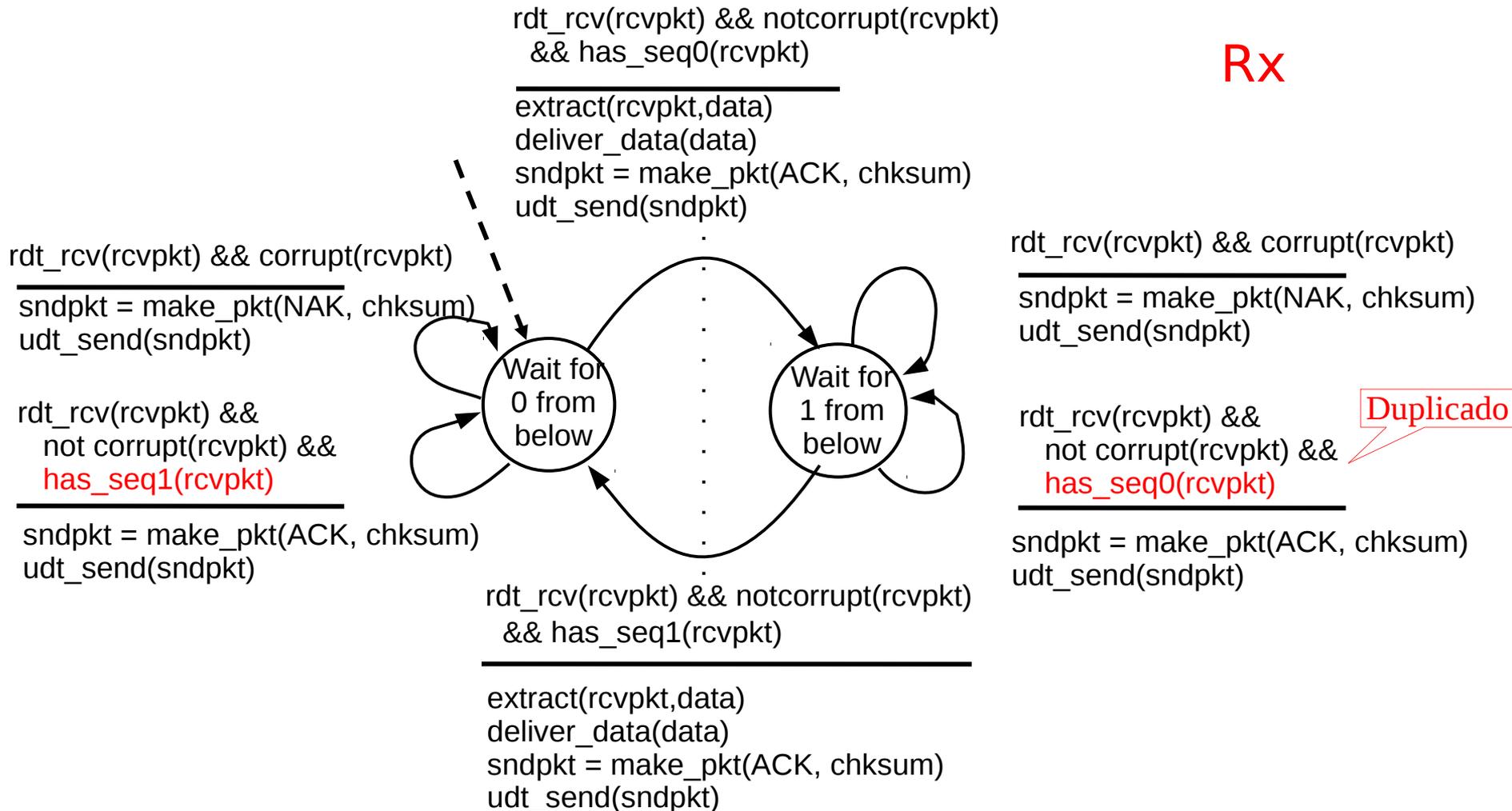
Tx



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 podemos enviar NAK de un paquete que nunca llegó.
- ❑ Preparándonos para la pérdida de paquetes, es mejor prescindir de los NAK.
- ❑ 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 suponiendo que no hay pérdidas

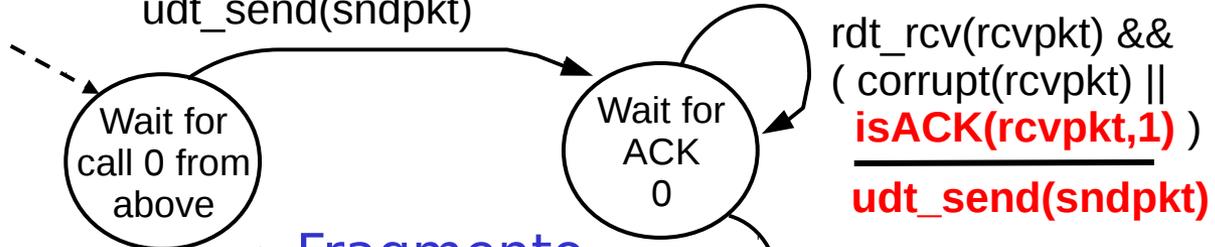
¿Cómo usted compara usar sólo NAK versus usar sólo ACK?

- ❑ ¿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 al trabajar sólo con NAK.
- ❑ 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

Lado Tx

rdt_send(data)
 sndpkt = make_pkt(0, data, checksum)
 udt_send(sndpkt)



Lado Rx

Fragmento MSF Tx

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

Λ

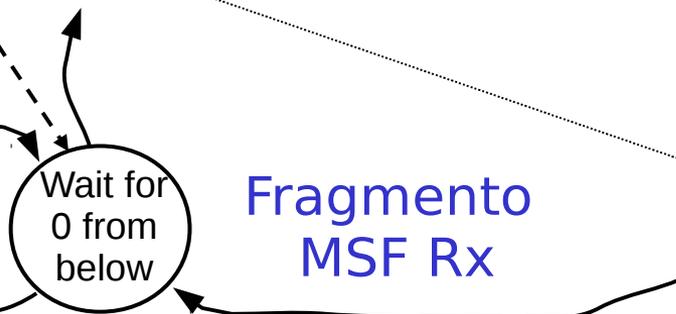
Fragmento MSF Rx

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

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

rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || has_seq1(rcvpkt))

sndpkt=make_pkt(ACK,1,checksum)
 udt_send(sndpkt)



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, 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

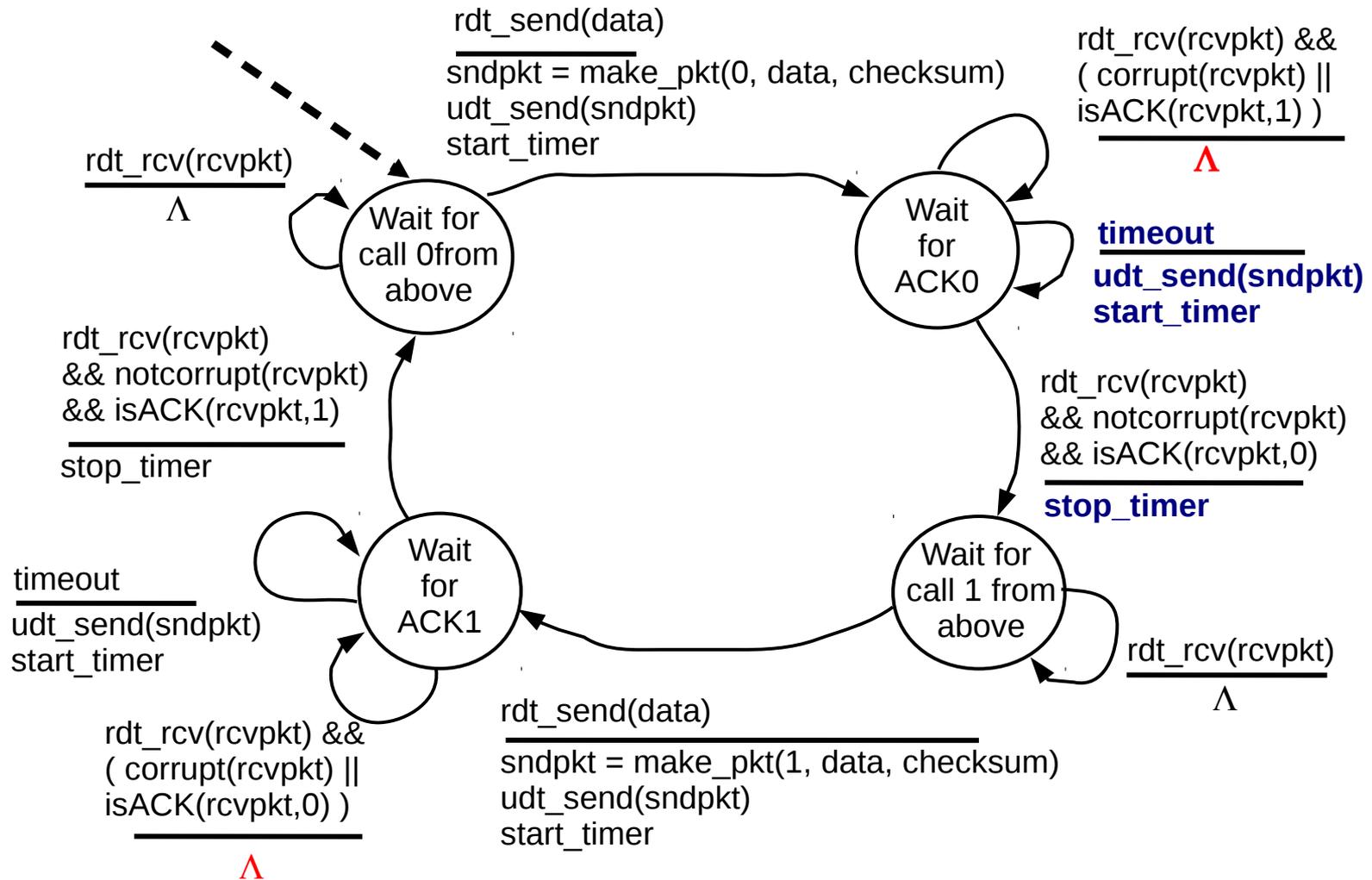
stop and wait

Tx envía un paquete,
Luego para y espera por la respuesta del Rx
Si no llega, hace re-envío hasta que llegue ACK.

Estrategia: transmisor espera un tiempo “razonable” por el ACK

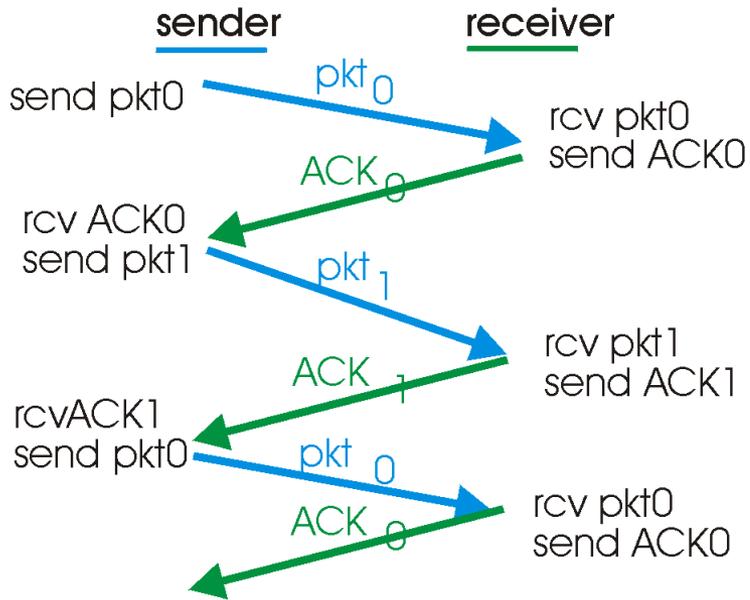
- ❑ Retransmitir si no se recibe ACK en este tiempo
- ❑ Si el paquete (o ACK) está retardado (no perdido):
 - La retransmisión será un duplicado, pero el uso de #'s de secuencia ya maneja esto
 - Receptor debe especificar el # de secuencia del paquete siendo confirmado en el ACK
- ❑ Se requiere un temporizador.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

rdt3.0 Transmisor

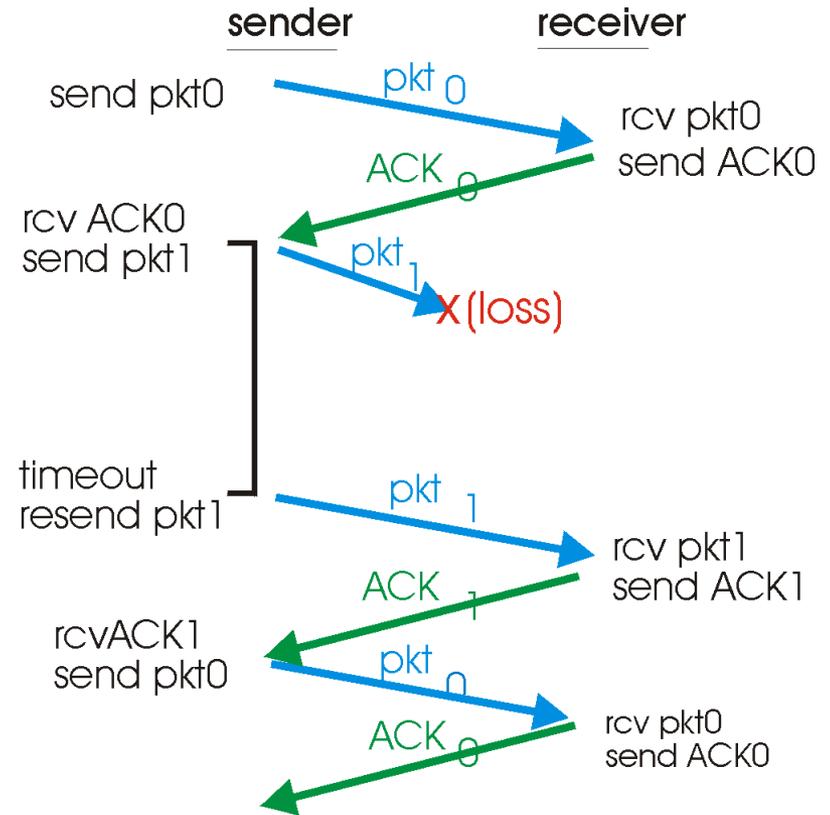


Hay simetría en los estados con # sec.=0, 1

rdt3.0 en acción



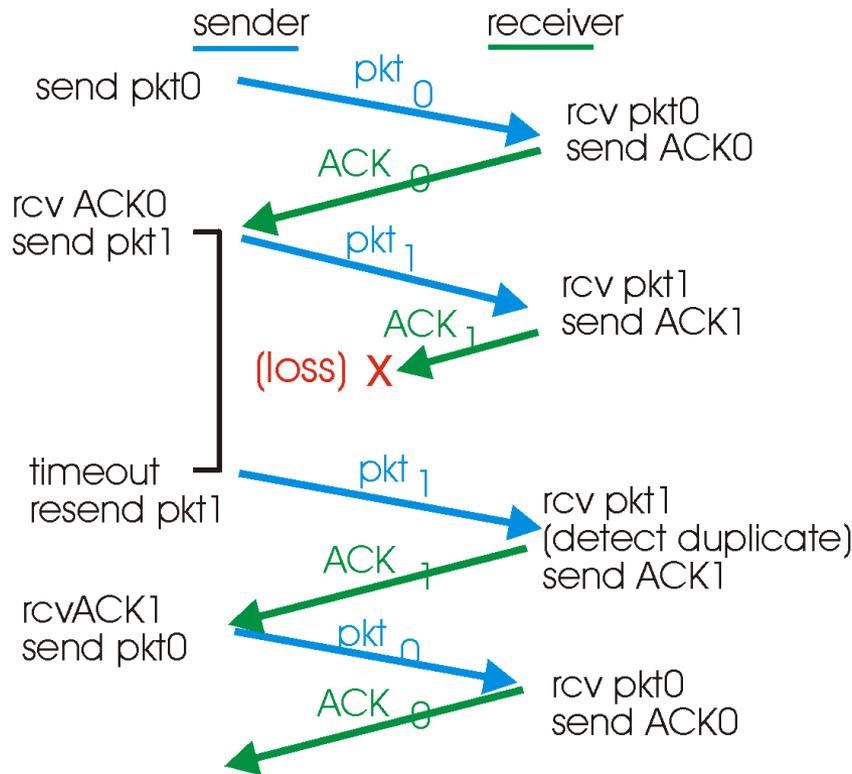
a) Operación **sin** pérdidas



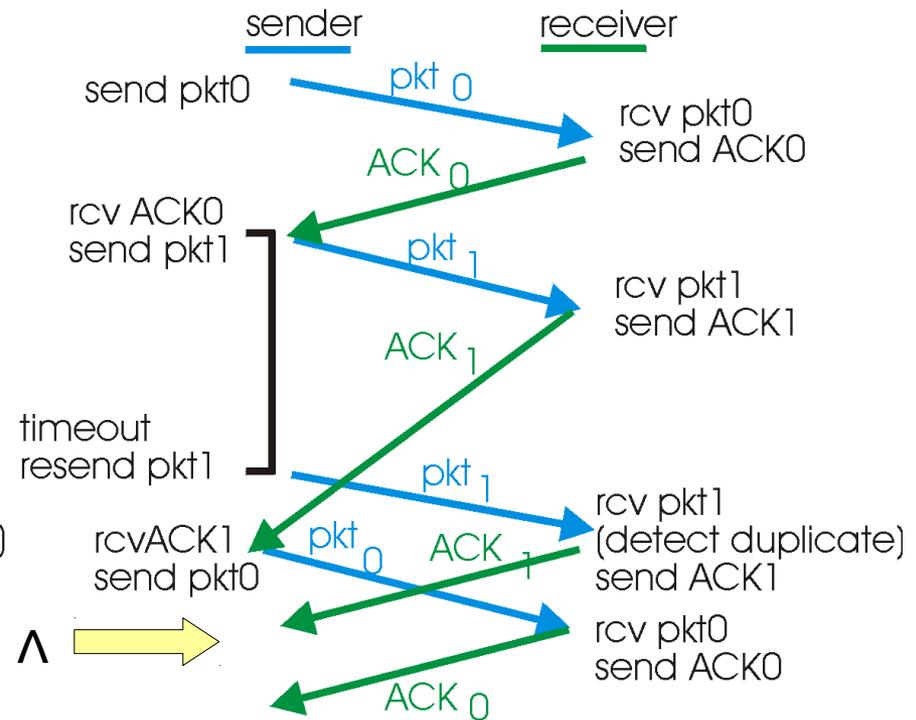
b) Operación **con** pérdidas

Es fundamental suponer que los paquetes no cambian su orden durante su trayecto

rdt3.0 en acción



c) Pérdida de ACK



d) Timeout prematuro

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
 - ❑ 3.4.2 Protocolos con Pipeline: Go-Back-N y Selective Repeat.
- ❑ 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