Tecnología Digital IV: Redes de Computadoras

Clase 8: Nivel de Transporte - Parte 2

Lucio Santi & Emmanuel Iarussi

Licenciatura en Tecnología Digital
Universidad Torcuato Di Tella

3 de abril de 2025

Agenda

- Servicios del nivel de transporte
- Multiplexación y demultiplexación
- Transporte no orientado a conexión: UDP
- Transferencia de datos confiable

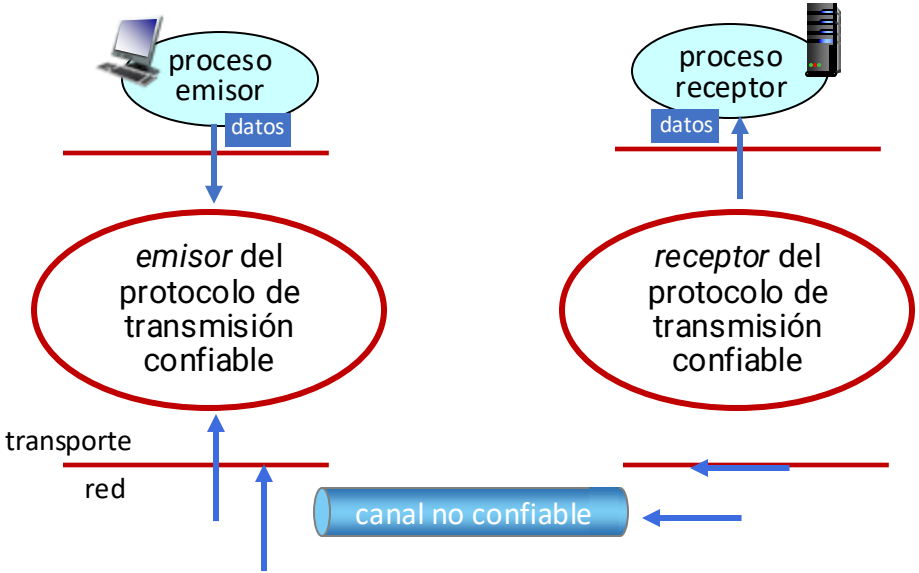
Protocolos confiables

Transmisión confiable: conceptos

servicio confiable: abstracción

Transmisión confiable: conceptos

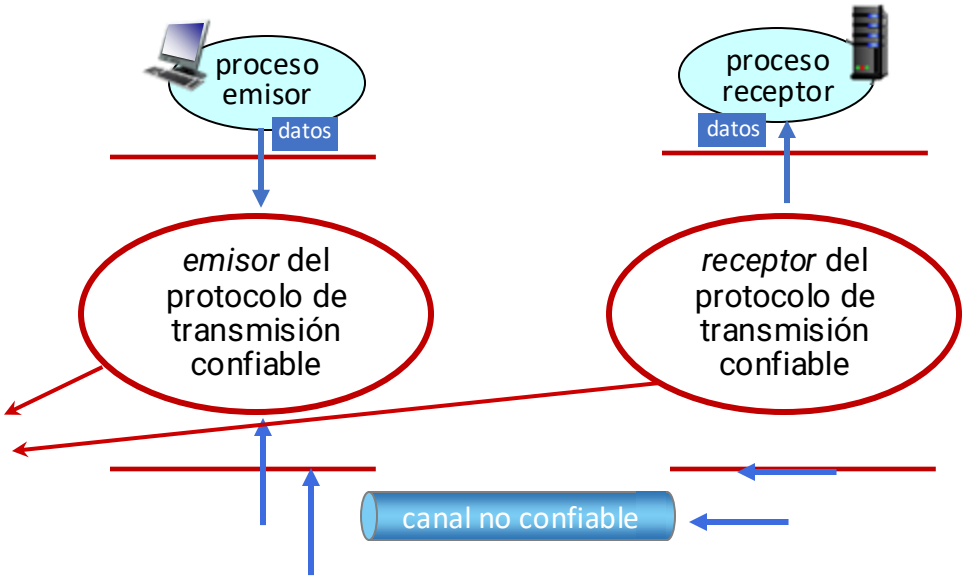
servicio confiable: abstracción



servicio confiable: implementación

Transmisión confiable: conceptos

La **complejidad** del protocolo depende fuertemente de las particularidades del canal no confiable subyacente (ej. si corrompe, pierde y/o reordena datos)

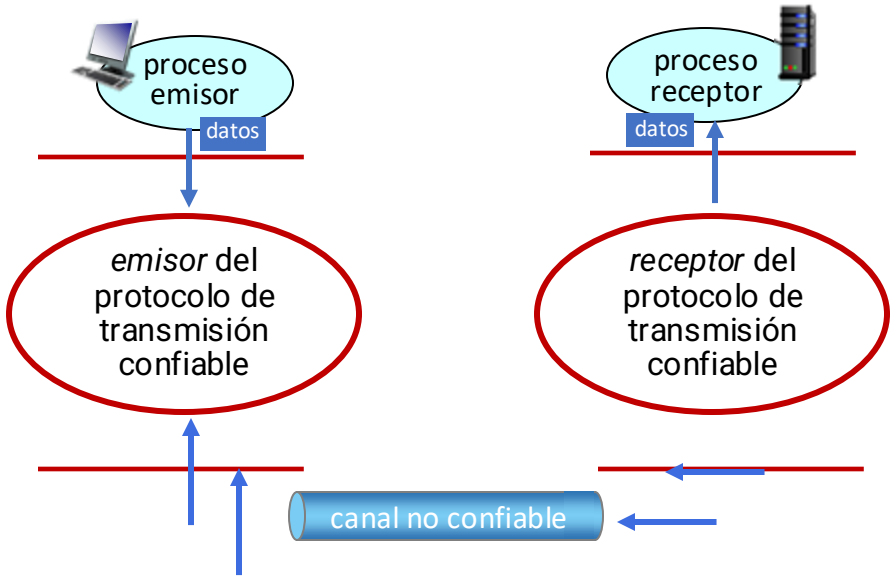


servicio confiable: implementación

Transmisión confiable: conceptos

Los interlocutores **no conocen** el estado del otro (e.g., el emisor no puede saber si un mensaje fue recibido)

- La única forma de conocer dicho estado es mediante el **intercambio de mensajes (que puede fallar!)**

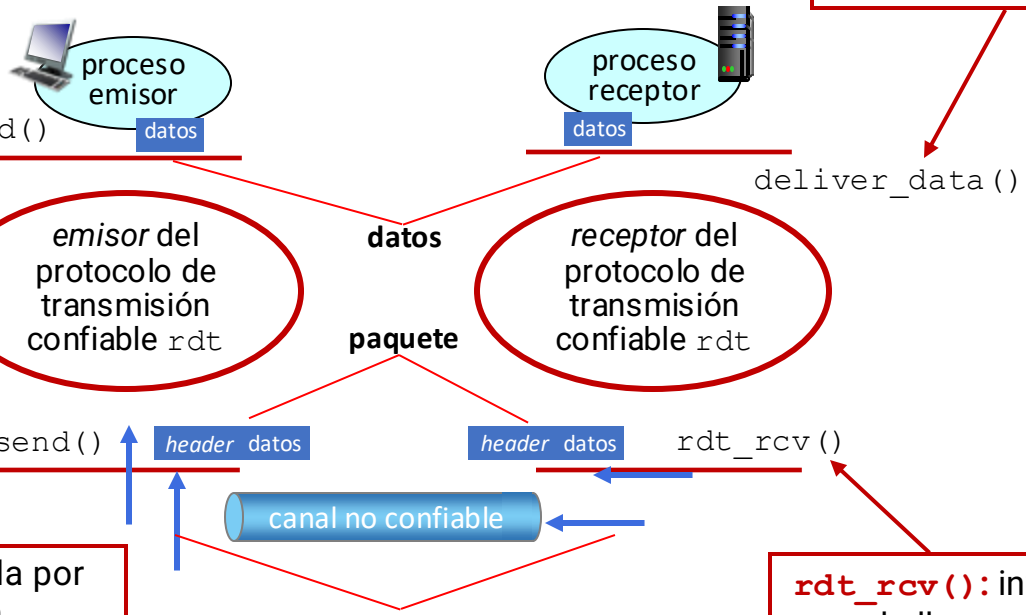


servicio confiable: implementación

API (abstracta) de un protocolo confiable

`rdt_send()`: invocada desde la app para pasar datos a entregar al nivel de aplicación del receptor

`deliver_data()`: invocada por `rdt` para entregar datos a la capa de aplicación



`udt_send()`: invocada por `rdt` para transmitir un paquete al receptor sobre el canal no confiable

`rdt_rcv()`: invocada cuando llega un paquete por el canal

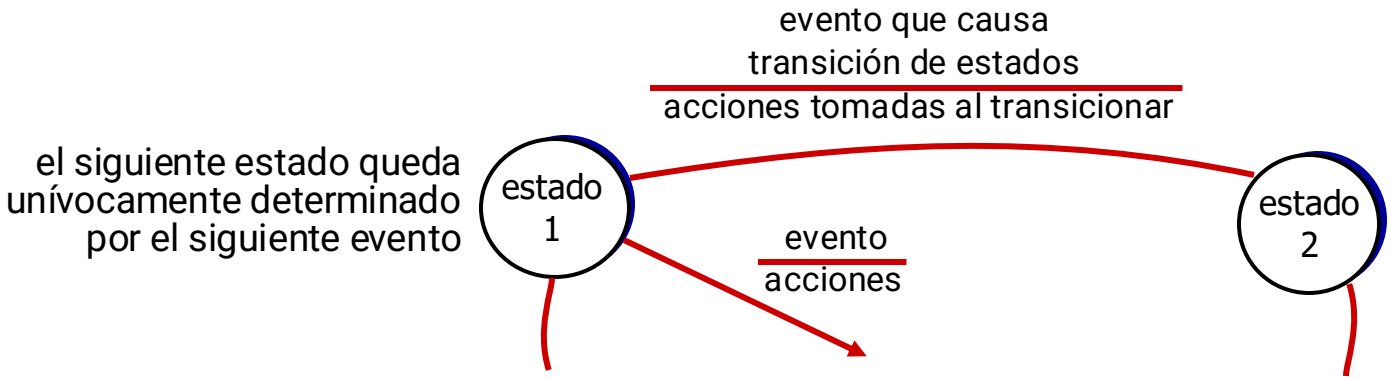
API (abstracta) de un protocolo confiable

- `rdt_send(data)` crea un paquete que contiene los datos (mediante la acción `make_pkt(data)`) y los envía a la capa subyacente (utilizando la acción `udt_send(packet)`)

- `rdt_rcv(packet)` recibe un paquete del canal, remueve el encabezado mediante `extract(packet,data)` y lo pasa a la capa del nivel superior mediante la acción `deliver_data(data)`

Diseño de un protocolo confiable

- Vamos a diseñar *incrementalmente* las características de un protocolo de transmisión confiable, `rdt` (*Reliable Data Transfer*)
- Motivación para entender la complejidad de protocolos como TCP
- Consideramos flujo unidireccional de datos (pero no así de información de control)
- Utilizaremos **máquinas de estados finitos** (FSMs) para especificar el protocolo:

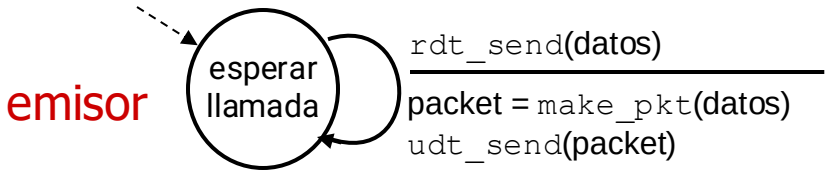


rdt1.0: transmisión confiable sobre un canal confiable

- Comenzamos asumiendo que el canal subyacente es **perfectamente confiable**
  - No introduce errores en los bits
  - Tampoco pierde paquetes
- Tenemos una FSM para el emisor y otra para el receptor:
  - El emisor envía datos al canal subyacente
  - El receptor lee datos del canal

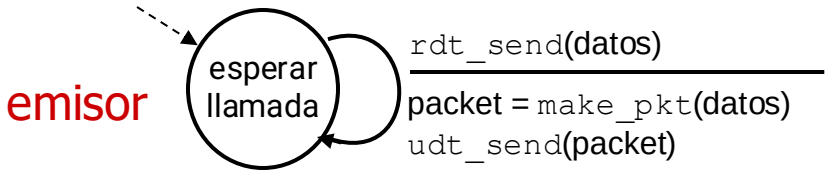
rdt1.0: transmisión confiable sobre un canal confiable

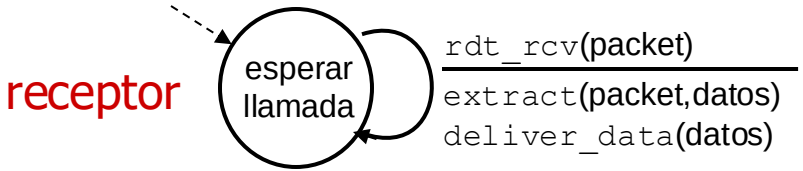
- Comenzamos asumiendo que el canal subyacente es **perfectamente confiable**
  - No introduce errores en los bits
  - Tampoco pierde paquetes
- Tenemos una FSM para el emisor y otra para el receptor:
  - El emisor envía datos al canal subyacente
  - El receptor lee datos del canal



rdt1.0: transmisión confiable sobre un canal confiable

- Comenzamos asumiendo que el canal subyacente es **perfectamente confiable**
  - No introduce errores en los bits
  - Tampoco pierde paquetes
- Tenemos una FSM para el emisor y otra para el receptor:
  - El emisor envía datos al canal subyacente
  - El receptor lee datos del canal





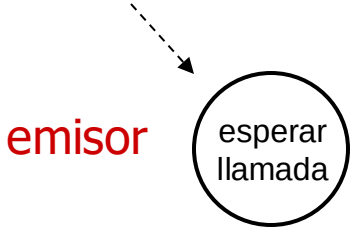
rdt2.0: transmisión sobre un canal con errores

- El canal subyacente puede corromper bits en los paquetes
  - Se pueden detectar vía *checksum*
- ¿Cómo podemos recuperarnos de los errores?
  - *Acknowledgements (**ACK**s):* el receptor indica explícitamente que el paquete llegó bien
  - *Negative Acknowledgements (**NAK**s):* el receptor indica explícitamente que el paquete llegó con errores
  - El emisor retransmite el paquete al recibir un NAK

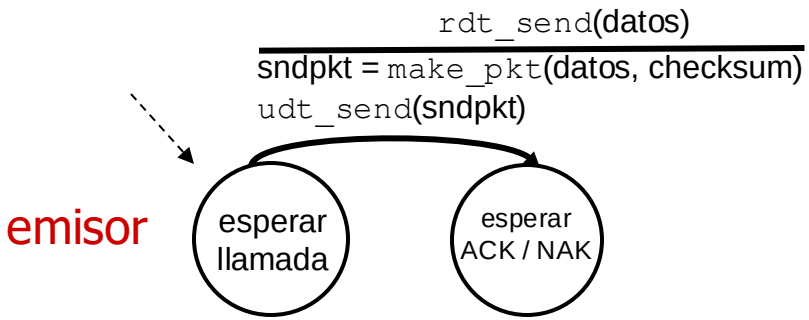
stop and wait

el emisor envía un paquete y luego espera la respuesta del receptor

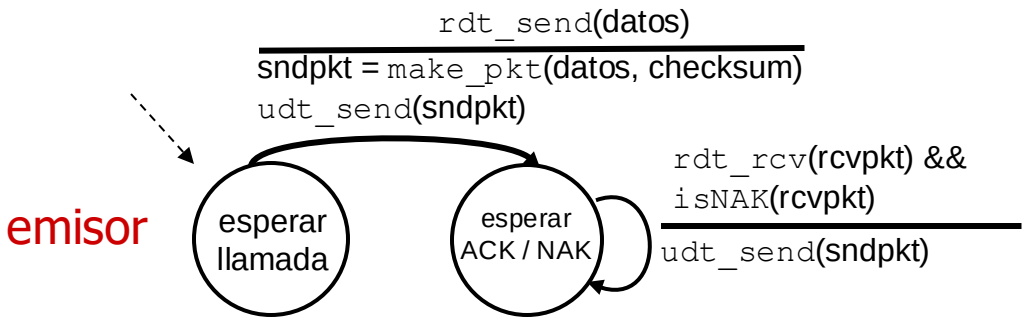
rdt2.0: especificación



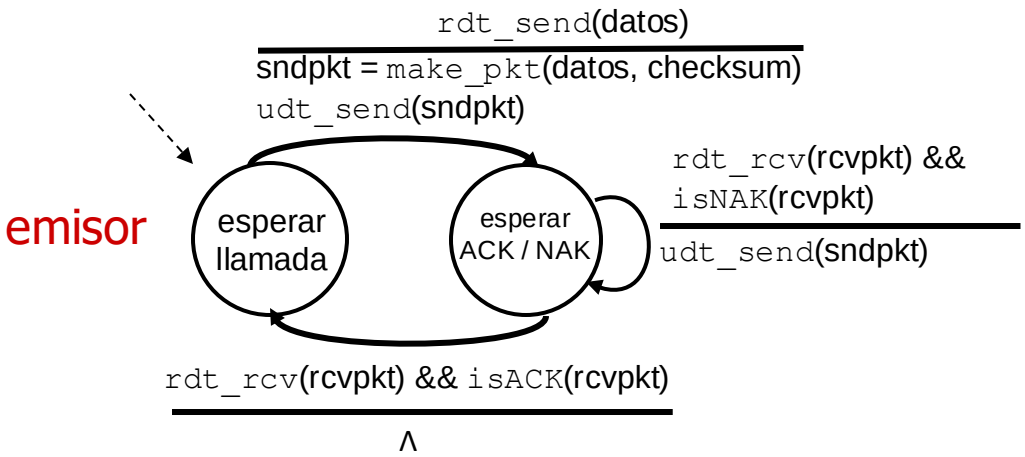
rdt2.0: especificación



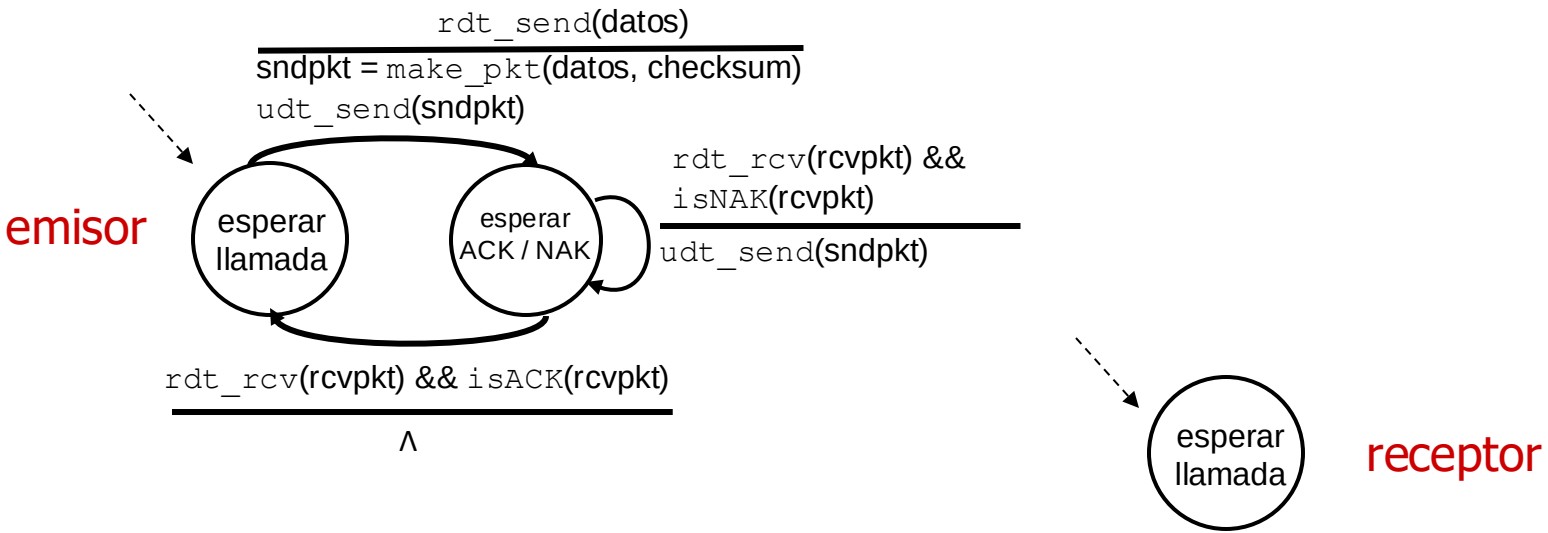
rdt2.0: especificación



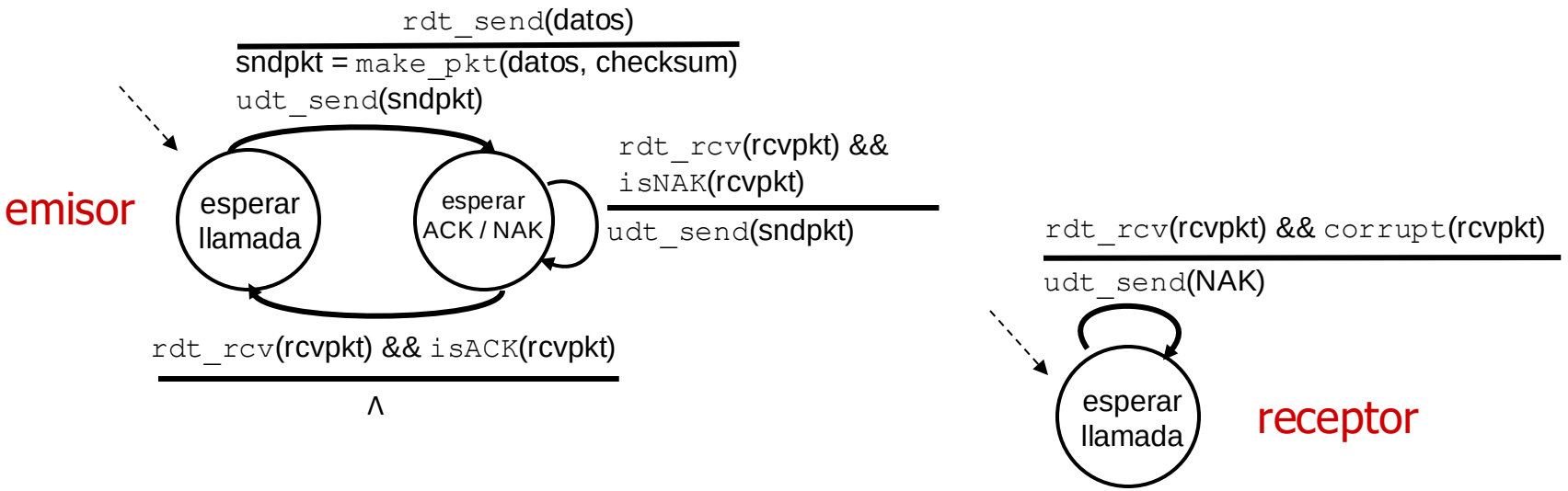
rdt2.0: especificación



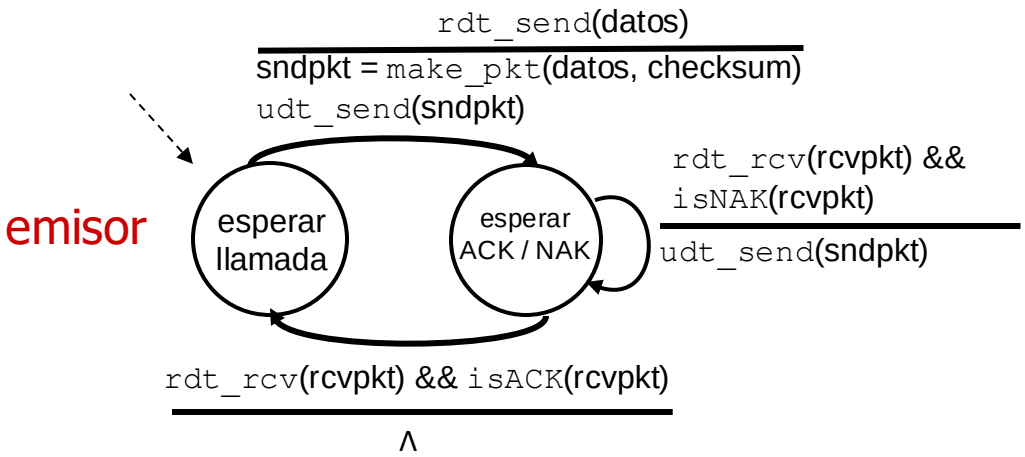
rdt2.0: especificación



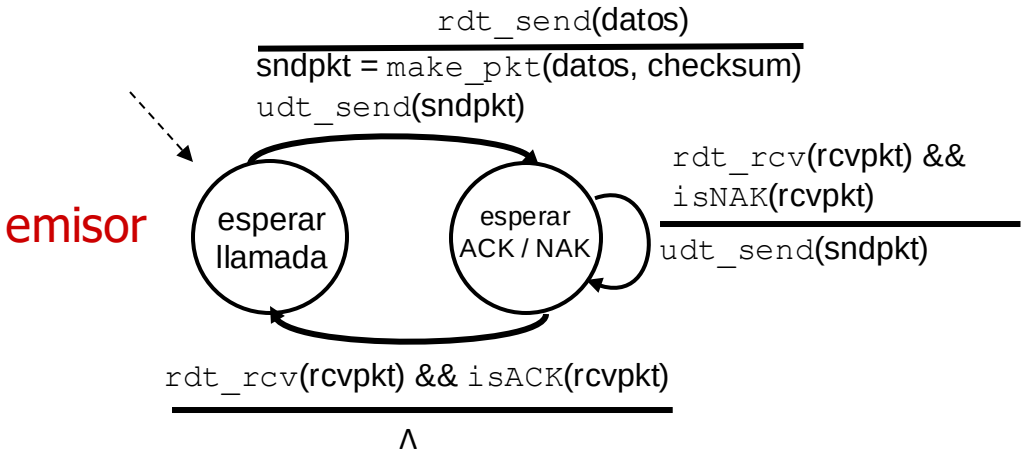
rdt2.0: especificación

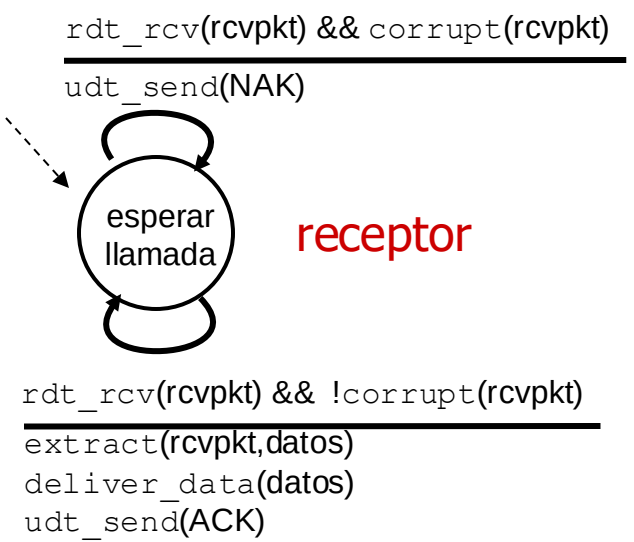


rdt2.0: especificación



rdt2.0: especificación





Nota: el emisor **no tiene forma** de conocer el estado del receptor a menos que el mismo se comunique explícitamente

rdt2.0: ACKs/NAKs corruptos

Este protocolo **falla** si hay corrupción en ACKs/NAKs

- El emisor **no tiene forma** de saber qué ocurrió en el receptor
- Podríamos retransmitir, pero puede generar datos duplicados

Manejo de duplicados:

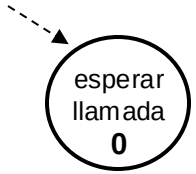
- El emisor retransmite el paquete actual ante corrupción de ACK/NAK
- Agrega un **número de secuencia** a cada paquete
- El receptor descarta paquetes duplicados valiéndose de este valor

stop and wait

el emisor envía un paquete y luego espera la respuesta del receptor

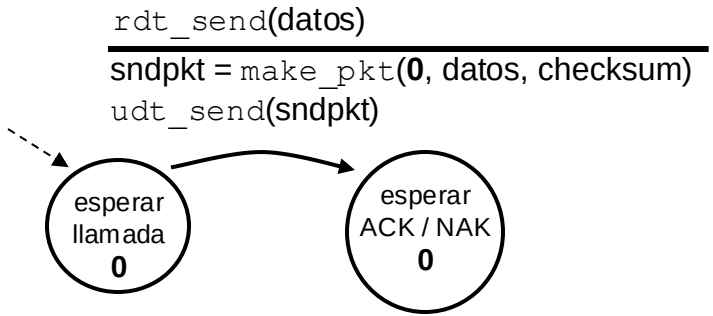
rdt2.1: manejo de ACKs/NAKs corruptos

emisor



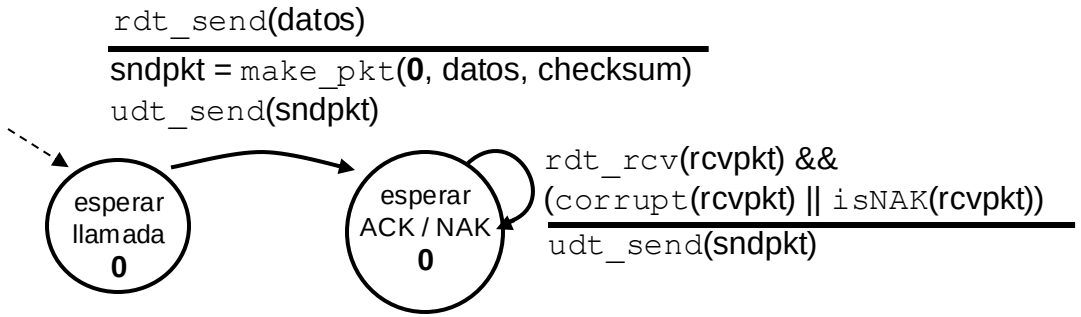
rdt2.1: manejo de ACKs/NAKs corruptos

emisor



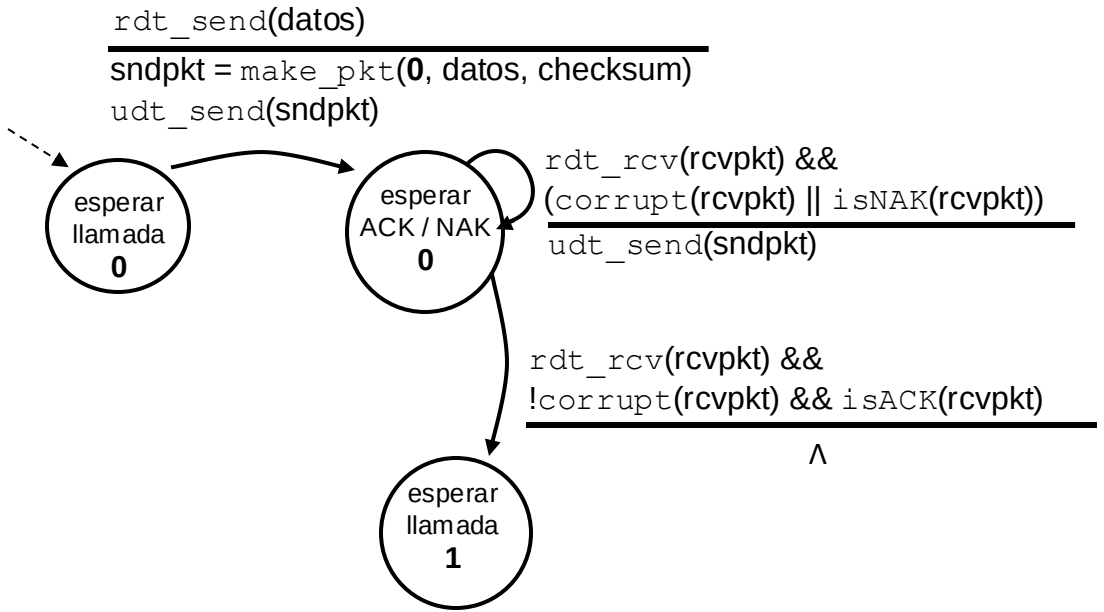
rdt2.1: manejo de ACKs/NAKs corruptos

emisor



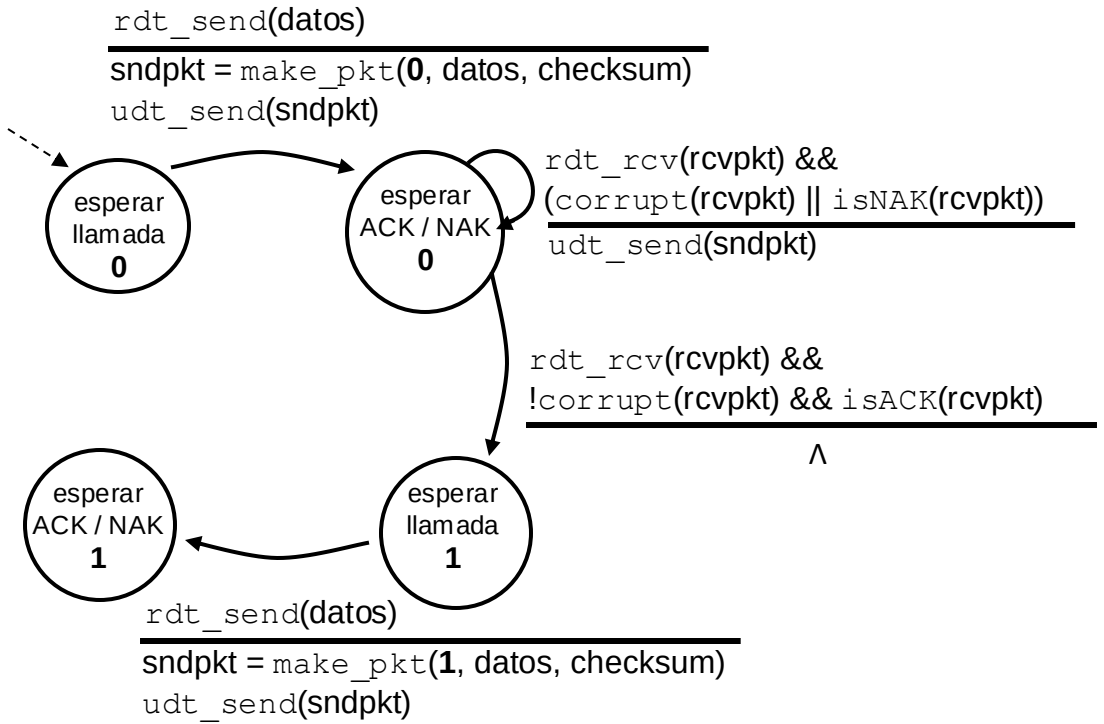
rdt2.1: manejo de ACKs/NAKs corruptos

emisor



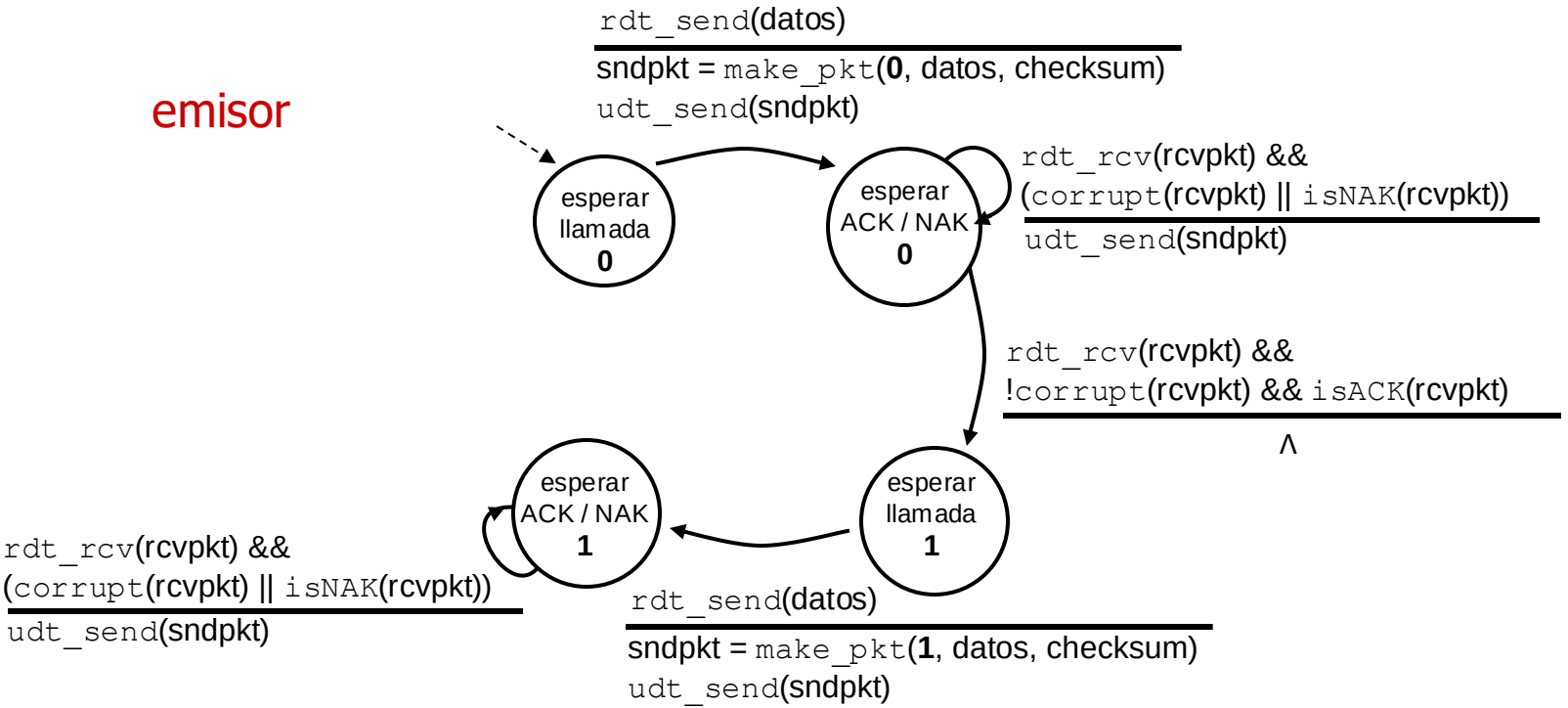
rdt2.1: manejo de ACKs/NAKs corruptos

emisor

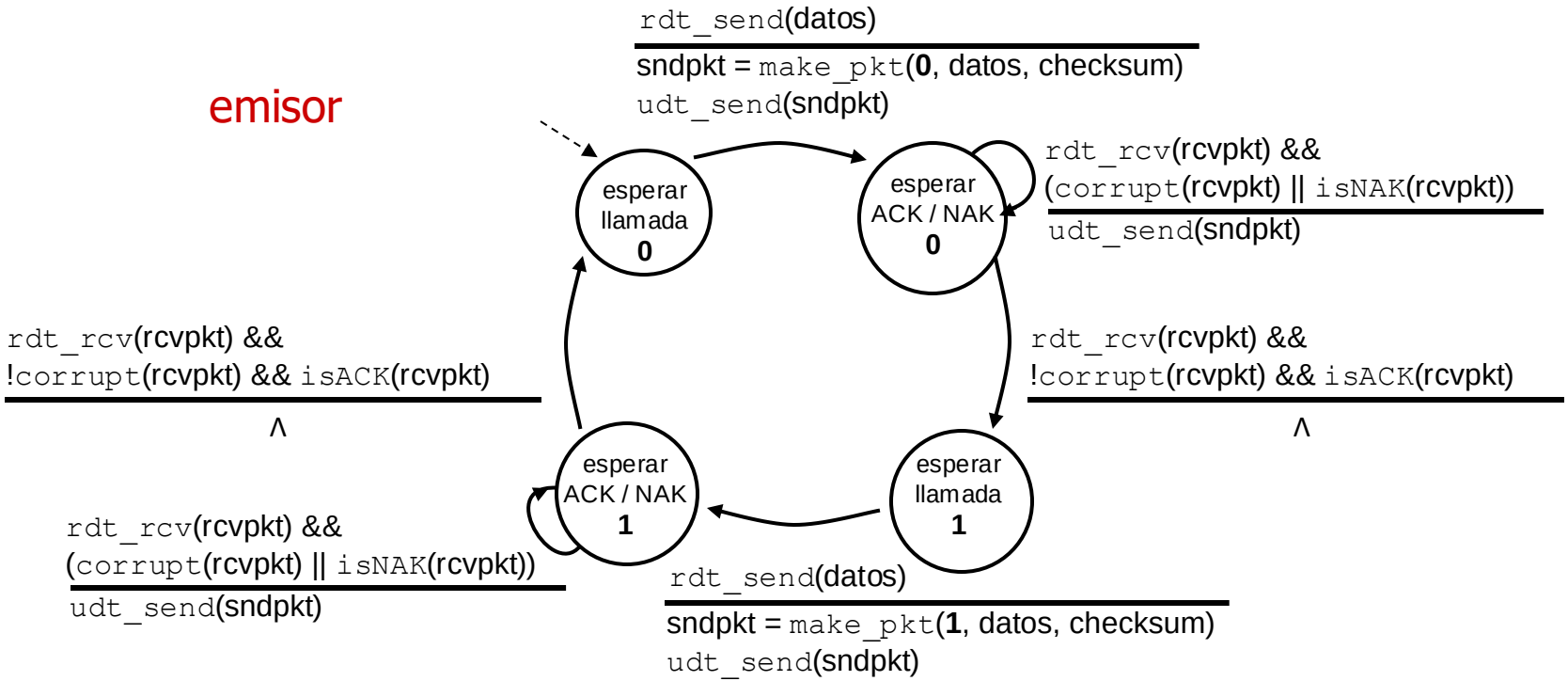


rdt2.1: manejo de ACKs/NAKs corruptos

emisor

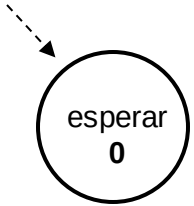


rdt2.1: manejo de ACKs/NAKs corruptos



rdt2.1: manejo de ACKs/NAKs corruptos

receptor

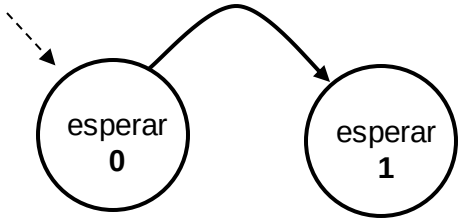


rdt2.1: manejo de ACKs/NAKs corruptos

receptor

`rdt_rcv`(rcvpkt) &&
!`corrupt`(rcvpkt) && `has_seq`(rcvpkt, **0**)

`extract`(rcvpkt,datos)
`deliver_data`(datos)
sndpkt = `make_pkt`(ACK, checksum)
`udt_send`(sndpkt)

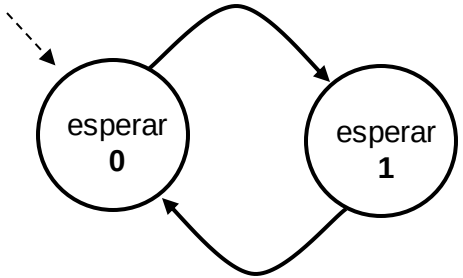


rdt2.1: manejo de ACKs/NAKs corruptos

receptor

`rdt_rcv`(rcvpkt) &&
!`corrupt`(rcvpkt) && `has_seq`(rcvpkt, **0**)

`extract`(rcvpkt,datos)
`deliver_data`(datos)
sndpkt = `make_pkt`(ACK, checksum)
`udt_send`(sndpkt)



`rdt_rcv`(rcvpkt) &&
!`corrupt`(rcvpkt) && `has_seq`(rcvpkt, **1**)

`extract`(rcvpkt,datos)
`deliver_data`(datos)
sndpkt = `make_pkt`(ACK, checksum)
`udt_send`(sndpkt)

rdt2.1: manejo de ACKs/NAKs corruptos

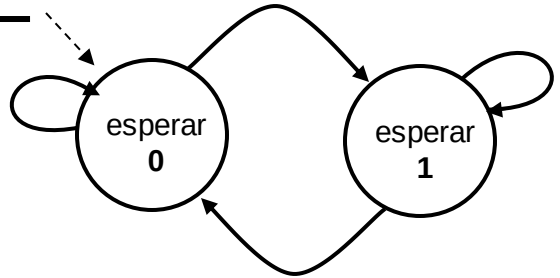
receptor

`rdt_rcv`(rcvpkt) &&
!`corrupt`(rcvpkt) && `has_seq`(rcvpkt, **0**)

`extract`(rcvpkt,datos)
`deliver_data`(datos)
sndpkt = `make_pkt`(ACK, checksum)
`udt_send`(sndpkt)

`rdt_rcv`(rcvpkt) && `corrupt`(rcvpkt)

sndpkt = `make_pkt`(NAK, checksum)
`udt_send`(sndpkt)



`rdt_rcv`(rcvpkt) && `corrupt`(rcvpkt)

sndpkt = `make_pkt`(NAK, checksum)
`udt_send`(sndpkt)

`rdt_rcv`(rcvpkt) &&
!`corrupt`(rcvpkt) && `has_seq`(rcvpkt, **1**)

`extract`(rcvpkt,datos)
`deliver_data`(datos)
sndpkt = `make_pkt`(ACK, checksum)
`udt_send`(sndpkt)

rdt2.1: manejo de ACKs/NAKs corruptos

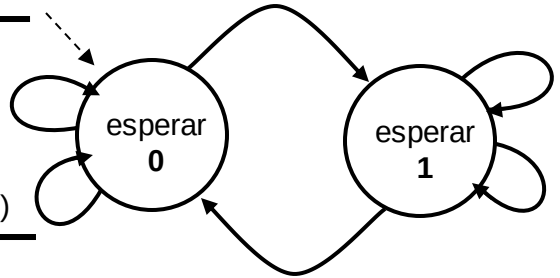
receptor

`rdt_rcv`(rcvpkt) && `corrupt`(rcvpkt)

sndpkt = `make_pkt`(NAK, checksum)
`udt_send`(sndpkt)

`rdt_rcv`(rcvpkt) &&
!`corrupt`(rcvpkt) && `has_seq`(rcvpkt, **1**)

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



`rdt_rcv`(rcvpkt) &&
!`corrupt`(rcvpkt) && `has_seq`(rcvpkt, **0**)

`extract`(rcvpkt,datos)
`deliver_data`(datos)
sndpkt = `make_pkt`(ACK, checksum)
`udt_send`(sndpkt)

`rdt_rcv`(rcvpkt) && `corrupt`(rcvpkt)

sndpkt = `make_pkt`(NAK, checksum)
`udt_send`(sndpkt)

`rdt_rcv`(rcvpkt) &&
!`corrupt`(rcvpkt) && `has_seq`(rcvpkt, **0**)

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

`rdt_rcv`(rcvpkt) &&
!`corrupt`(rcvpkt) && `has_seq`(rcvpkt, **1**)

`extract`(rcvpkt,datos)
`deliver_data`(datos)
sndpkt = `make_pkt`(ACK, checksum)
`udt_send`(sndpkt)

rdt2.1: conclusiones

Emisor

- Agrega un **número de secuencia** a los paquetes
- Sólo utiliza **dos** números de secuencia
- Debe comprobar **corrupción** de los ACKs/NAKs recibidos
- **Duplica** la cantidad de estados
  - El estado debe recordar el número de secuencia del paquete esperado

Receptor

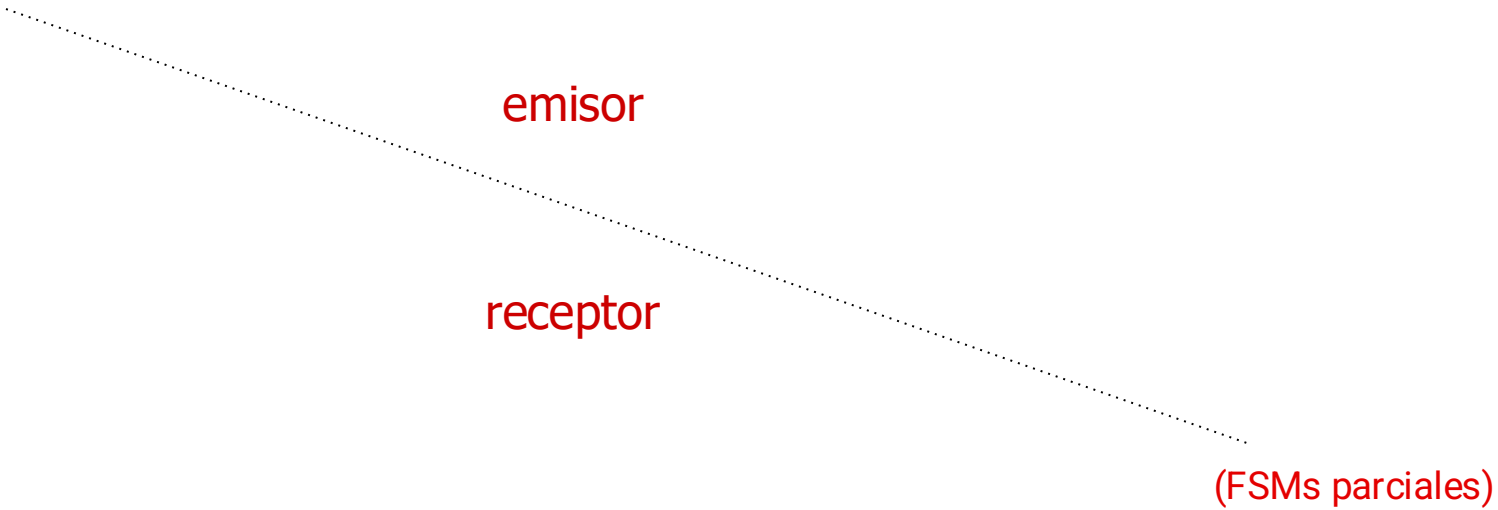
- Debe comprobar si el paquete recibido es un **duplicado**
  - El estado indica el número de secuencia del paquete esperado
- El receptor **no puede saber** si su último ACK/NAK llegó correctamente al emisor

rdt2.2: eliminando los NAKs

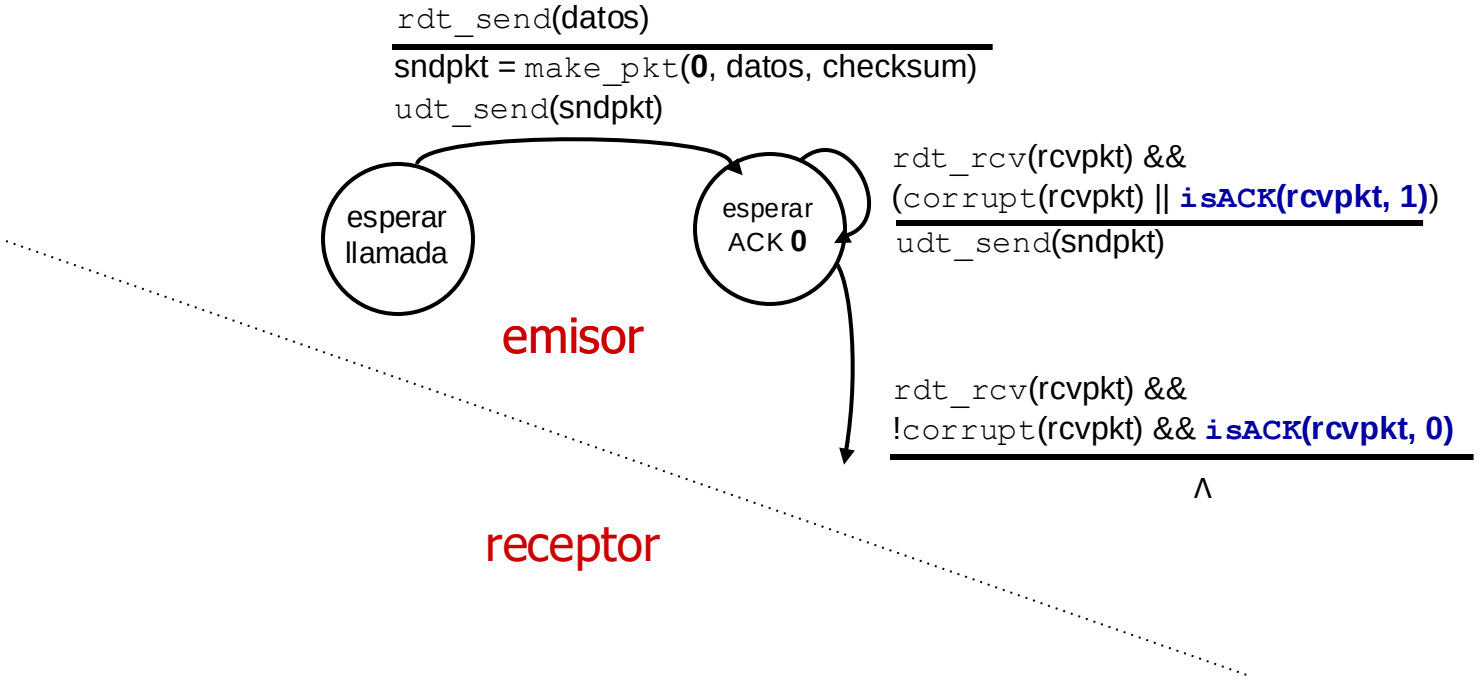
- Misma funcionalidad que `rdt2.1` pero **sólo empleando ACKs**
- En vez de un NAK, el receptor envía un ACK del último paquete recibido correctamente
  - El receptor debe indicar explícitamente el número de secuencia del paquete ACKeado
- Un ACK duplicado en el emisor dispara la misma acción que un NAK: **retransmitir** el paquete actual

TCP utiliza este enfoque (más sobre esto luego)

rdt2.2: eliminando los NAKs

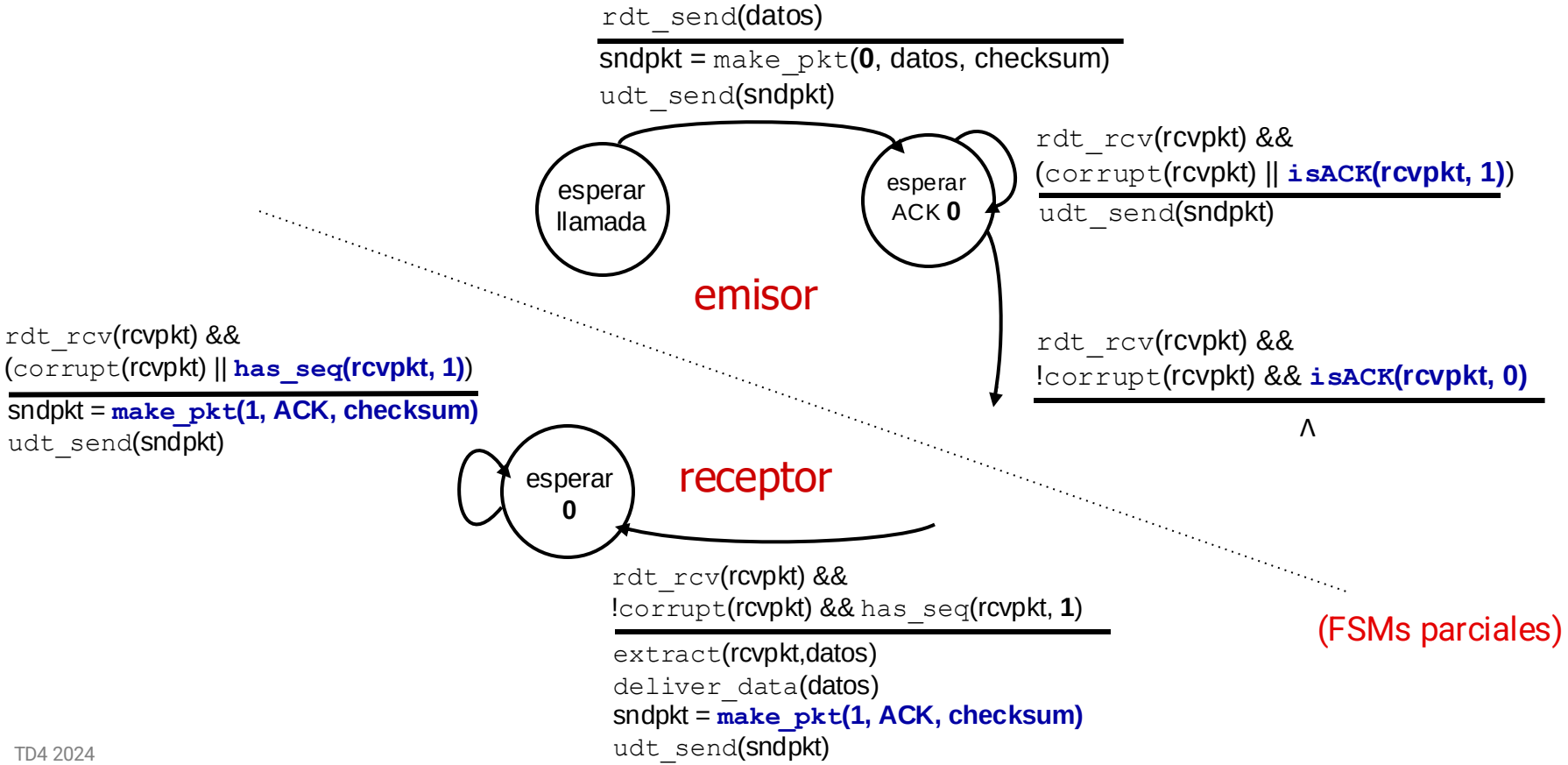


rdt2.2: eliminando los NAKs



(FSMs parciales)

rdt2.2: eliminando los NAKs



rdt3.0: transmisión sobre un canal con errores y pérdidas

Nueva suposición: el canal subyacente, además de introducir errores, puede **perder paquetes** (datos y/o ACKs)

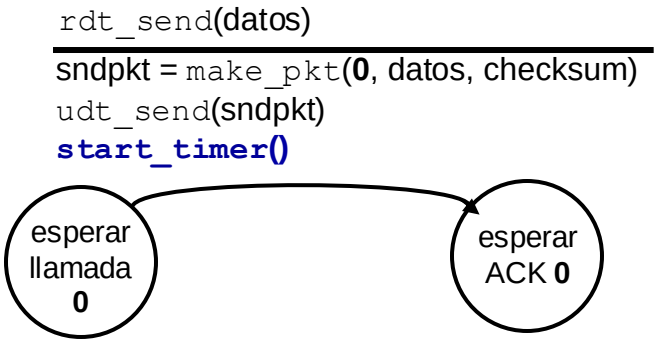
- Necesitamos checksums, números de secuencia, ACKs y retransmisiones, pero con esto solo **no alcanza**

Enfoque: el emisor **espera** un ACK durante cierto tiempo

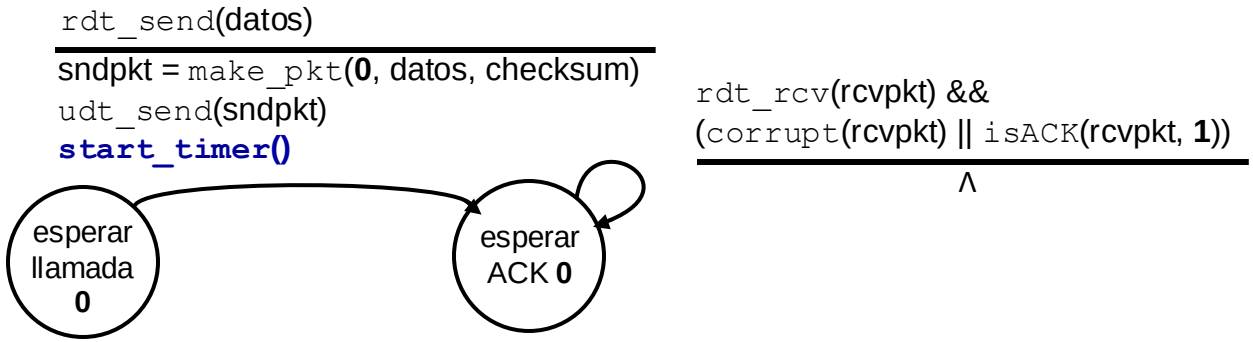
- Si no llega, **retransmite** el paquete
- Si hay demoras (en vez de pérdidas efectivas),
  - La retransmisión causará duplicados (pero ya lo contemplamos con los números de secuencia)
  - El receptor debe indicar el número de secuencia del paquete ACKeado
- La *espera* se controla con un ***timer*** que produce interrupciones cada cierto intervalo de tiempo (***timeout***)

rdt3.0: especificación del emisor

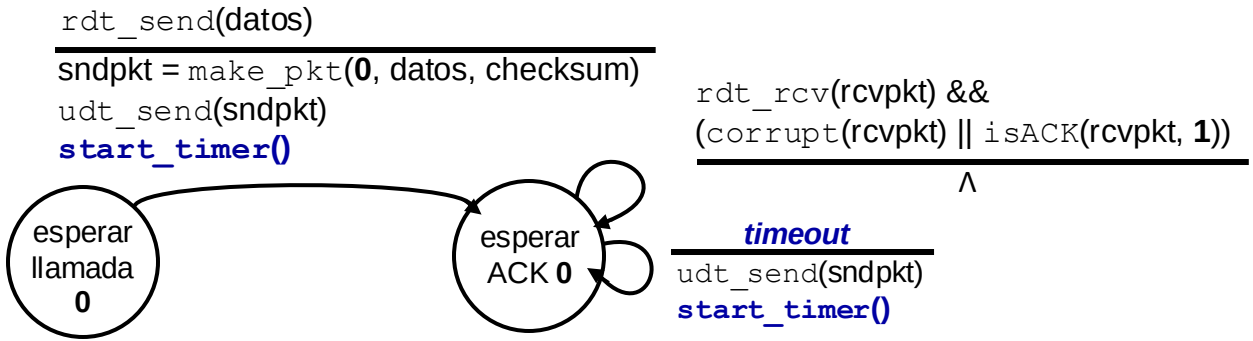
rdt3.0: especificación del emisor



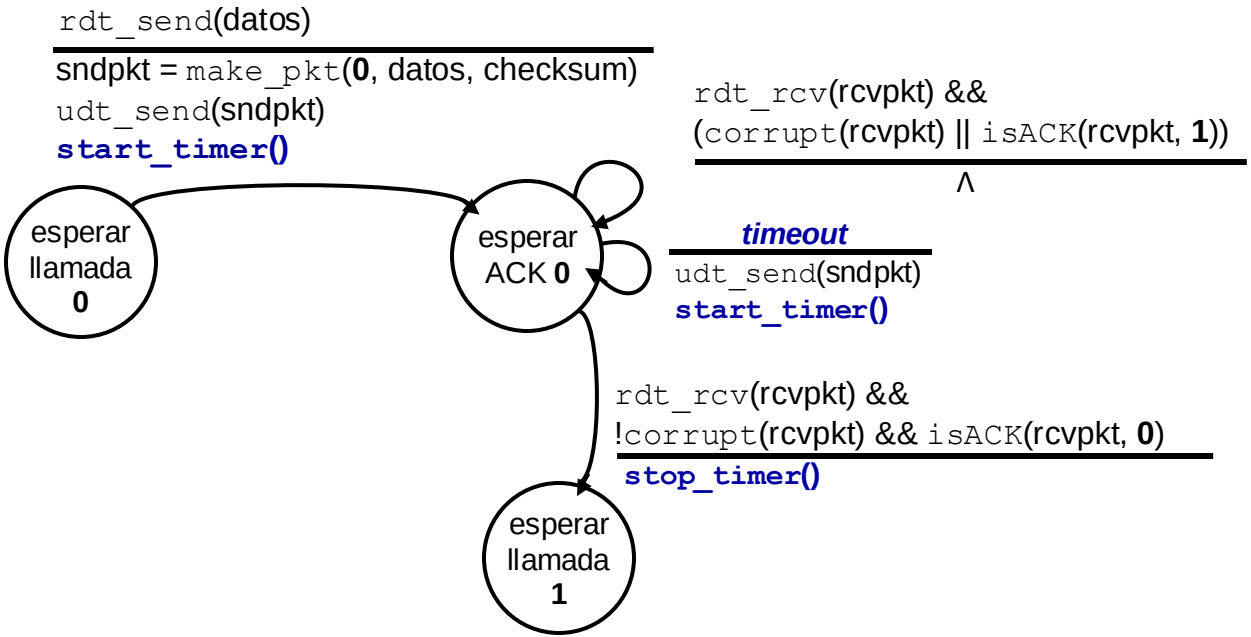
rdt3.0: especificación del emisor



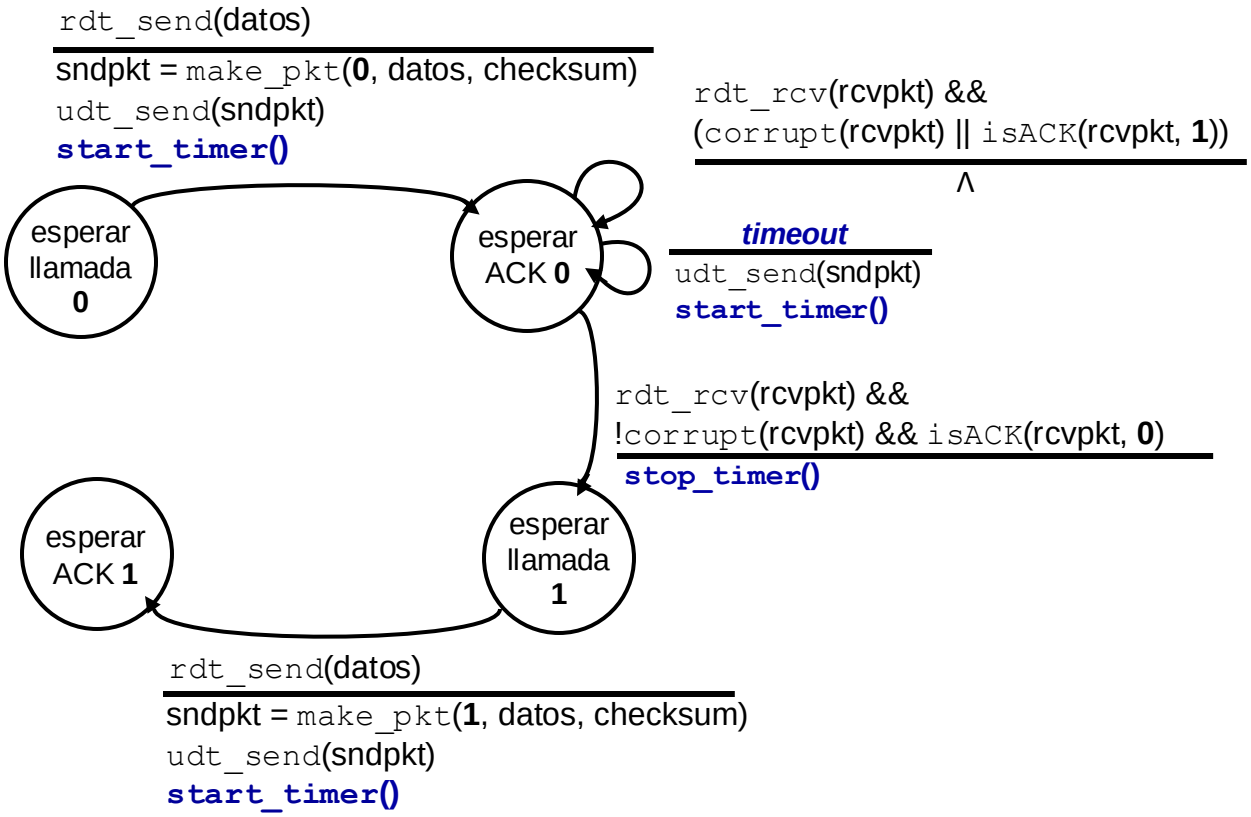
rdt3.0: especificación del emisor



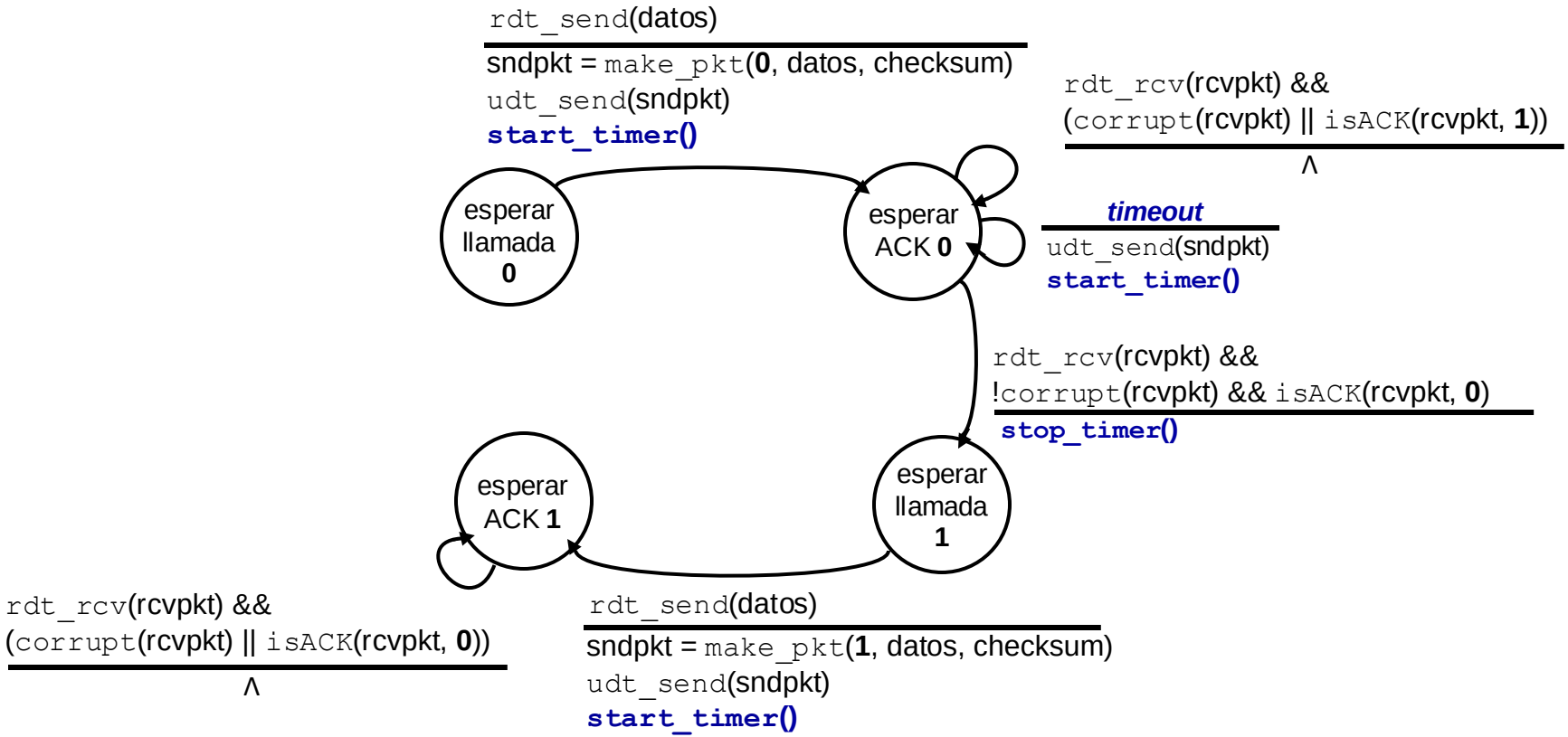
rdt3.0: especificación del emisor



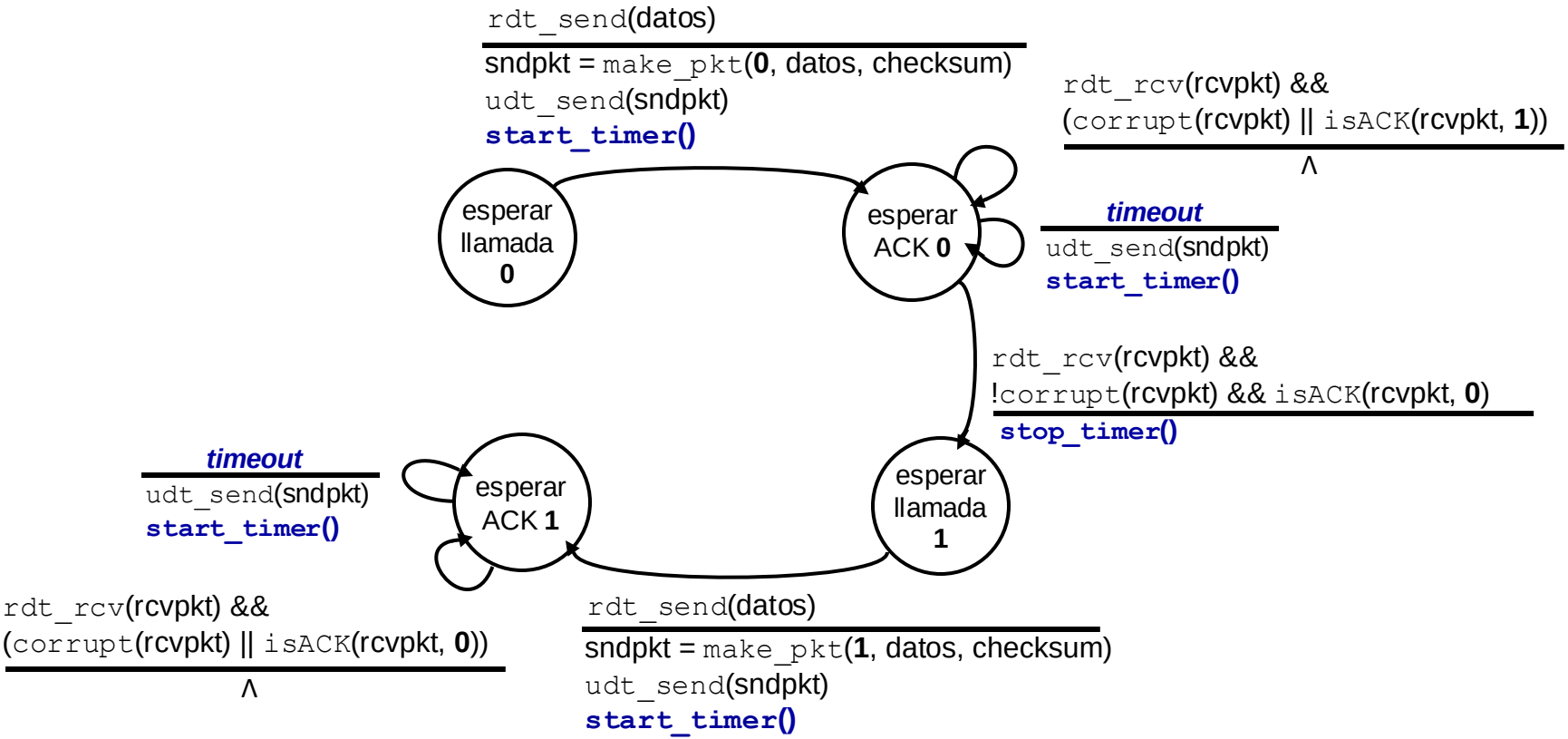
rdt3.0: especificación del emisor



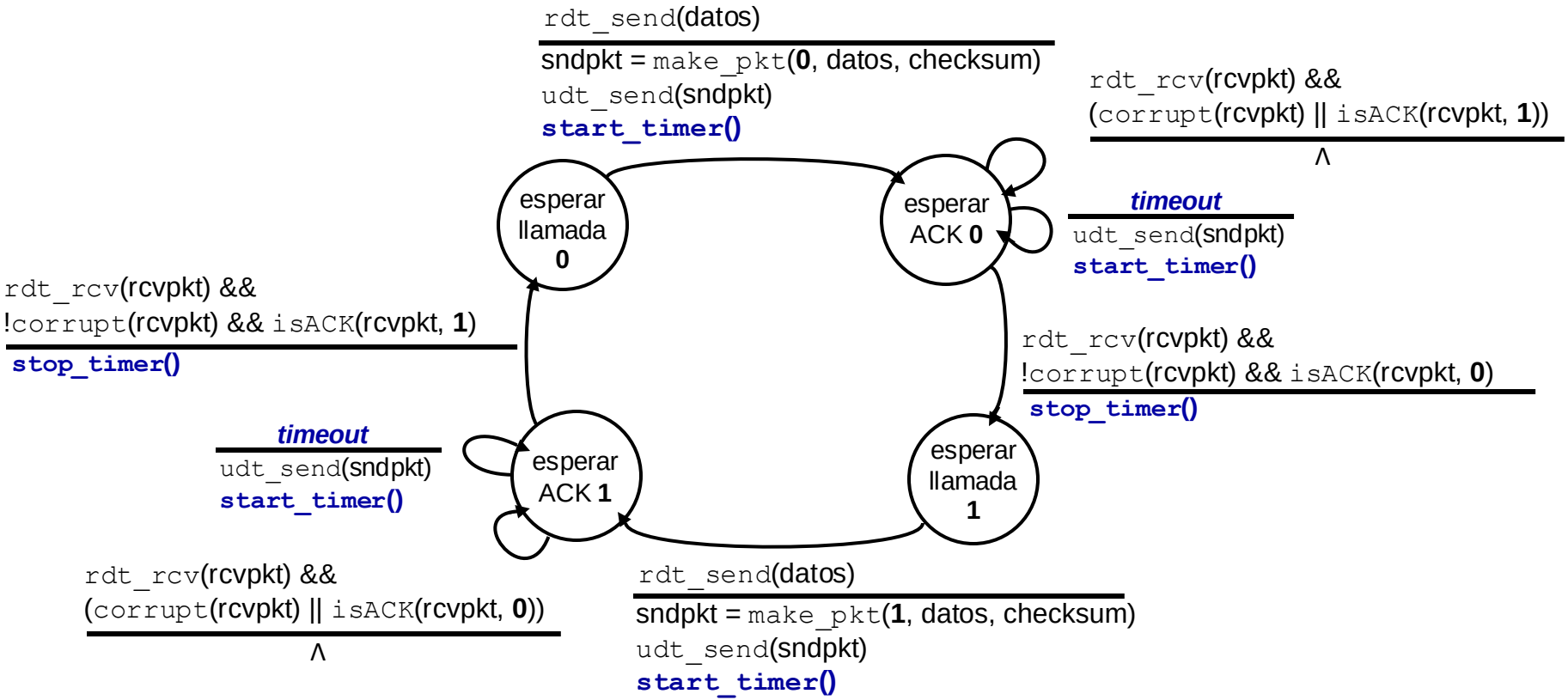
rdt3.0: especificación del emisor



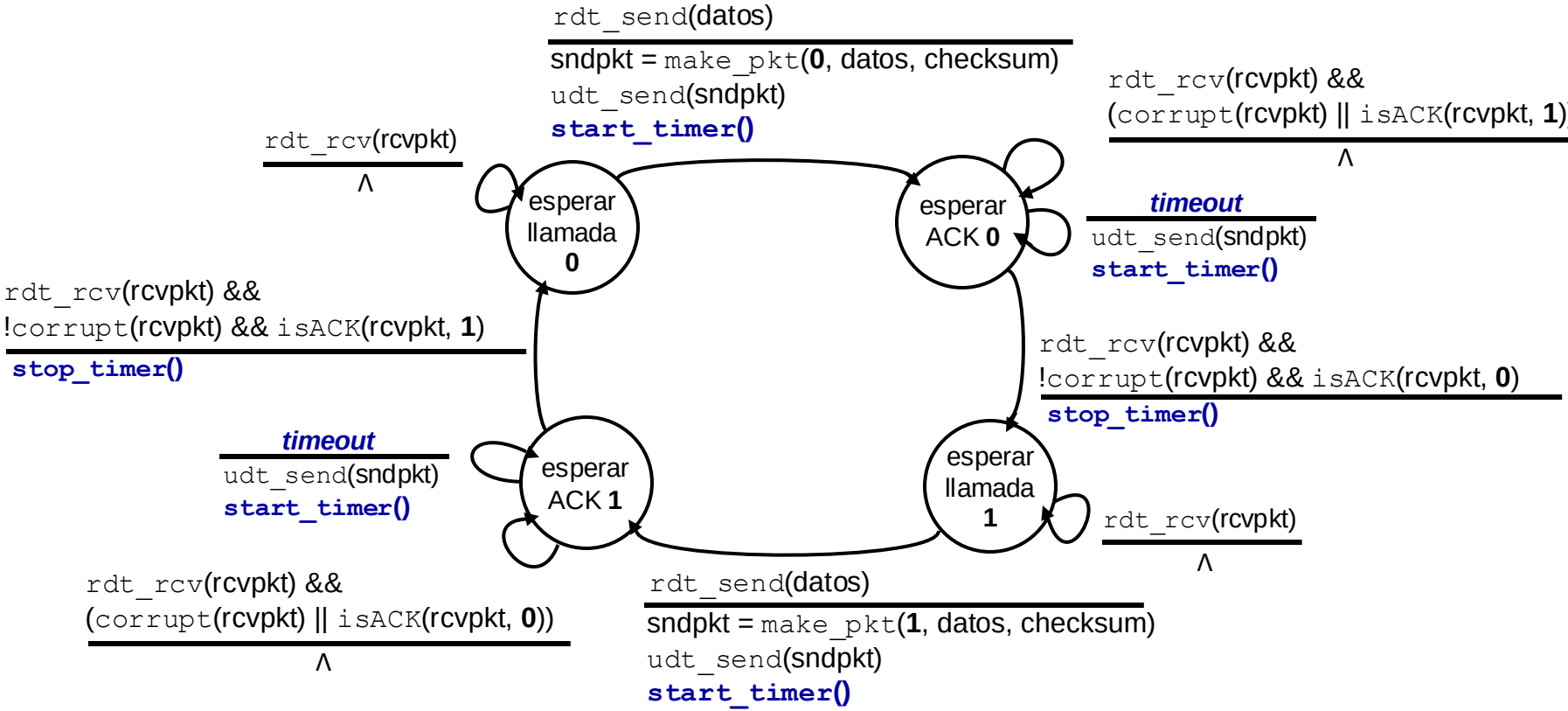
rdt3.0: especificación del emisor



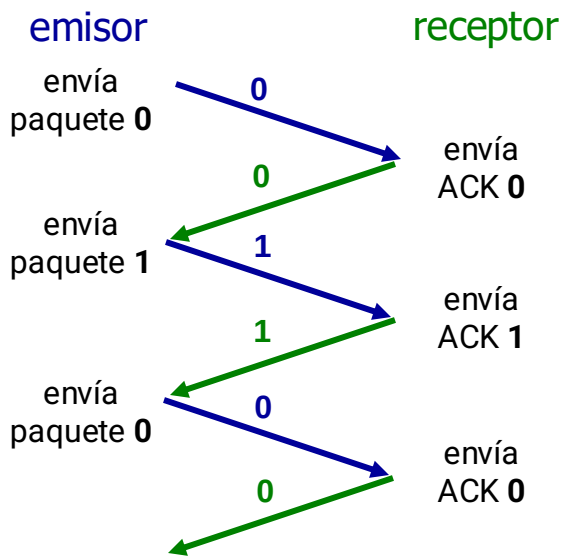
rdt3.0: especificación del emisor



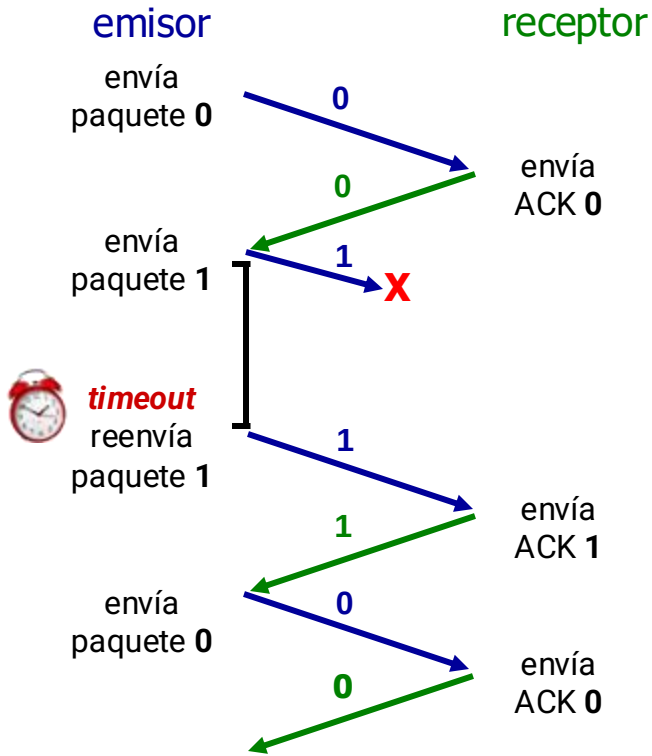
rdt3.0: especificación del emisor



rdt3.0: escenarios

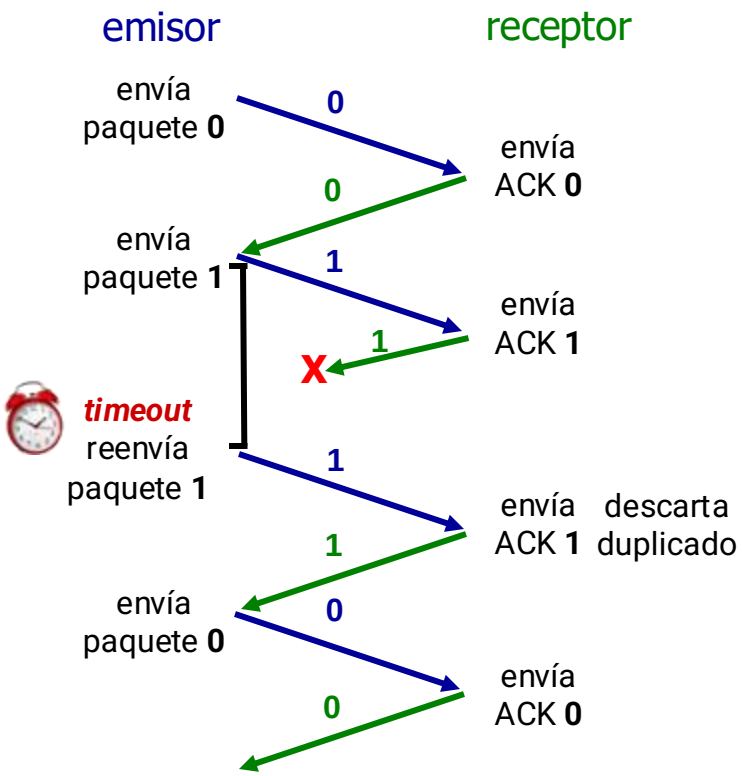


(a) sin pérdida

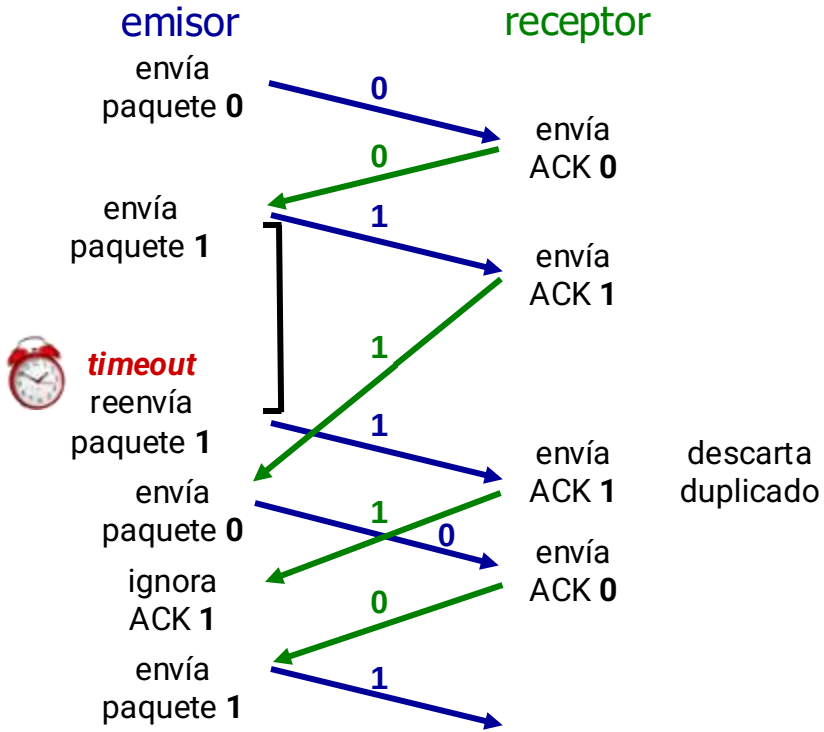


(b) pérdida de paquete (datos)

rdt3.0: escenarios



(c) pérdida de paquete (ACK)

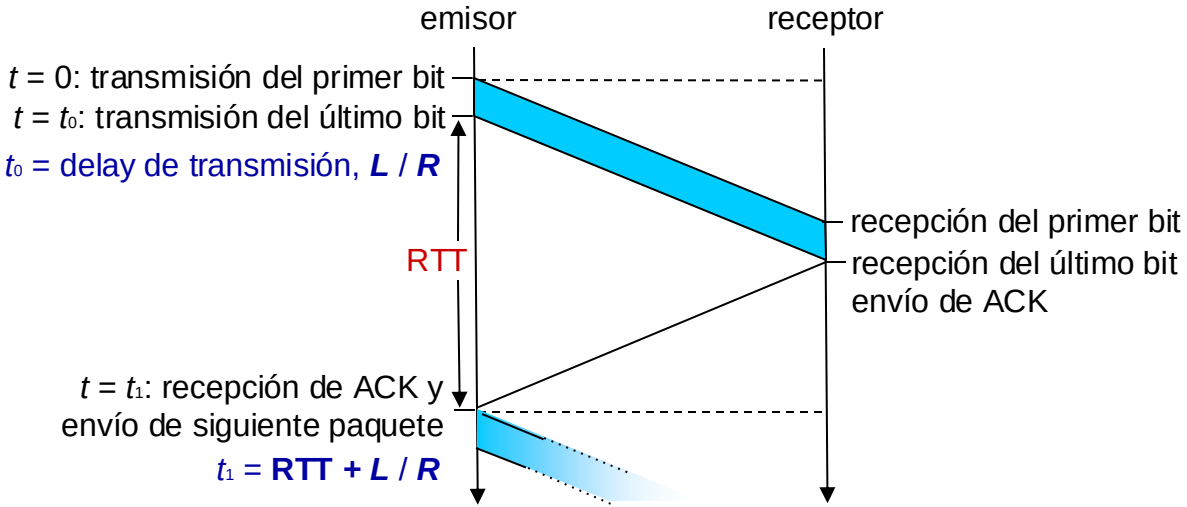


(d) ACK demorado

rdt3.0: rendimiento (*stop-and-wait*)

- Veamos cómo estimar la **utilización del canal *U*** por parte del emisor

$$U = \frac{L / R}{\text{RTT} + L / R}$$



- Si e.g. L = 8 kb, R = 1 Gbps y RTT = 30 ms, ***U* = 0.00027**
  - Rendimiento **muy pobre**
- El protocolo **limita el rendimiento** del canal subyacente