

# Chapter 3

## Transport Layer

### A note on the use of these ppt slides:

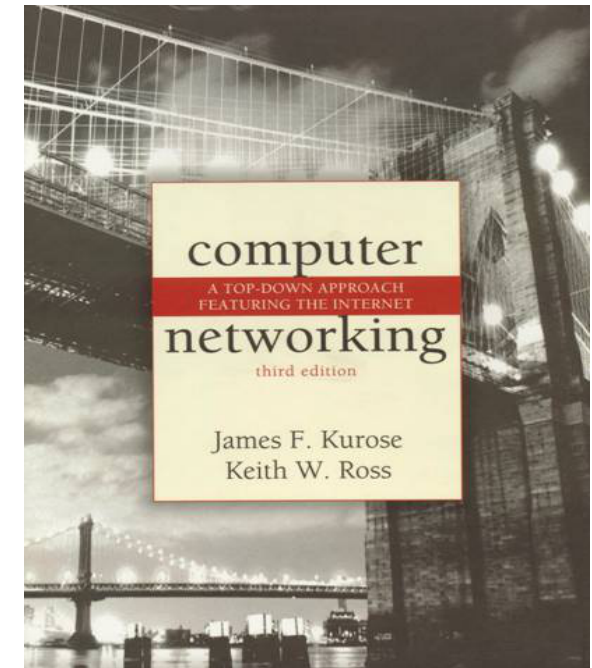
We're making these slides freely available to all (faculty, students, readers). They're in PowerPoint form so you can add, modify, and delete slides (including this one) and slide content to suit your needs. They obviously represent a *lot* of work on our part. In return for use, we only ask the following:

If you use these slides (e.g., in a class) in substantially unaltered form, that you mention their source (after all, we'd like people to use our book!)

If you post any slides in substantially unaltered form on a www site, that you note that they are adapted from (or perhaps identical to) our slides, and note our copyright of this material.

Thanks and enjoy! JFK/KWR

All material copyright 1996-2004  
J.F Kurose and K.W. Ross, All Rights Reserved



*Computer Networking:  
A Top Down Approach  
Featuring the Internet,  
3<sup>rd</sup> edition.*

Jim Kurose, Keith Ross  
Addison-Wesley, July  
2004.

# Capa de Transporte

- Objetivos
  - Comprender los principios detrás de los servicios de la capa de transporte
    - Multiplexado/demultiplexado
    - Transferencia de datos confiable
    - Control de flujo
    - Control de congestión
- Aprender sobre los protocolos de la capa de transporte en Internet
  - UDP: transporte sin conexión
  - TCP: transporte orientado a conexión
  - Control de congestión en TCP

# Capa de Transporte

## 3.1 Servicios de la capa de transporte

### 3.2 Multiplexado y demultiplexado

### 3.3 Transporte sin conexión: UDP

### 3.4 Principios de transferencia de datos confiable

### 3.5 Transporte orientado a conexión: TCP

Estructura de segmentos

Transferencia de datos confiable

Control de flujo

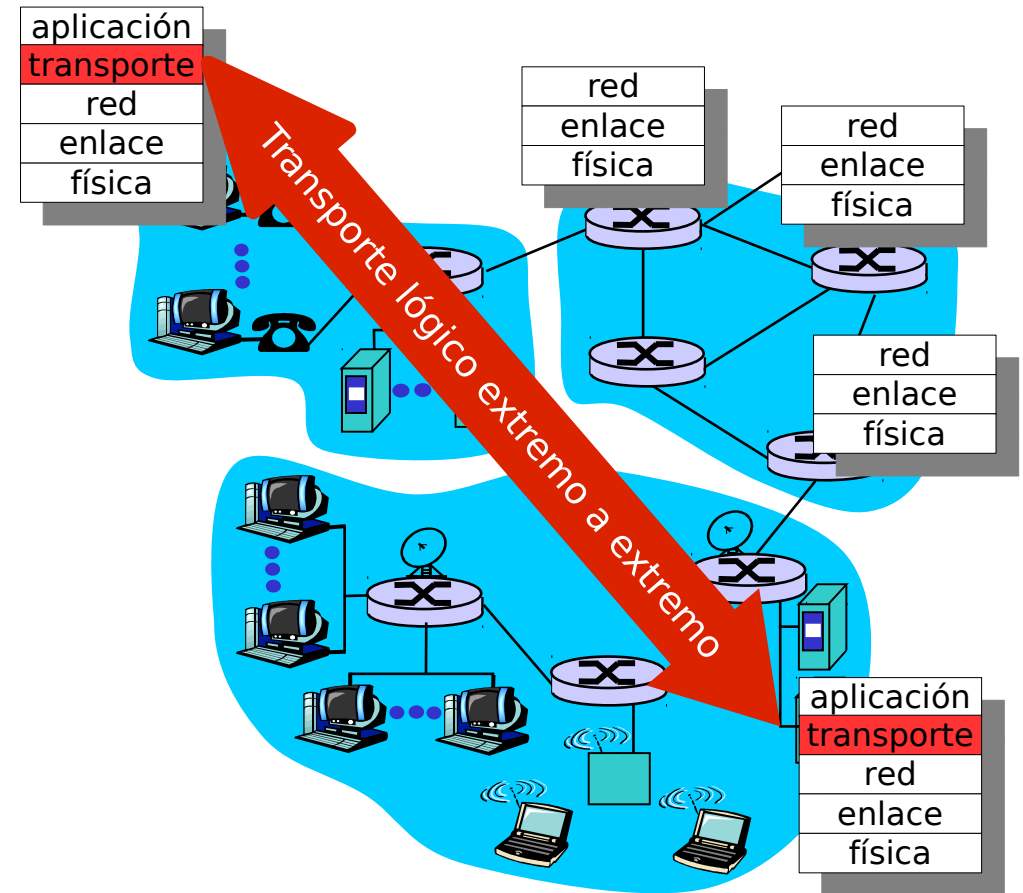
Administración de conexiones

### 3.6 Principios de control de congestión

### 3.7 Control de congestión en TCP

# Servicios y protocolos de Transporte

- Proveer comunicación lógica entre procesos de aplicación que corren en diferentes hosts
- Los protocolos de Transporte corren en sistemas finales
  - Emisor: descompone los mensajes de aplicación en segmentos, los pasa a la capa de red
  - Receptor: rearma los segmentos en mensajes, los pasa a la capa de aplicación
  - Más de un protocolo de transporte disponible a las aplicaciones
    - Internet: TCP y UDP



# Capa de Red vs. Transporte

Capa de red

Comunicación lógica **entre  
hosts**

Capa de transporte

Comunicación lógica **entre  
procesos**

Descansa sobre, y mejora, los  
servicios de la capa de red

Analogía de la casa

12 personas envían cartas a  
otras 12 personas

Procesos = personas

Mensajes de aplicación =  
cartas en sus sobres

Hosts = casas

Protocolo de Transporte =  
personas mensajeros

Protocolo de capa de red =  
servicio postal

# Protocolos de transporte en Internet

Entrega confiable, en orden  
(TCP)

- Control de congestión

- Control de flujo

- Establecimiento de conexión

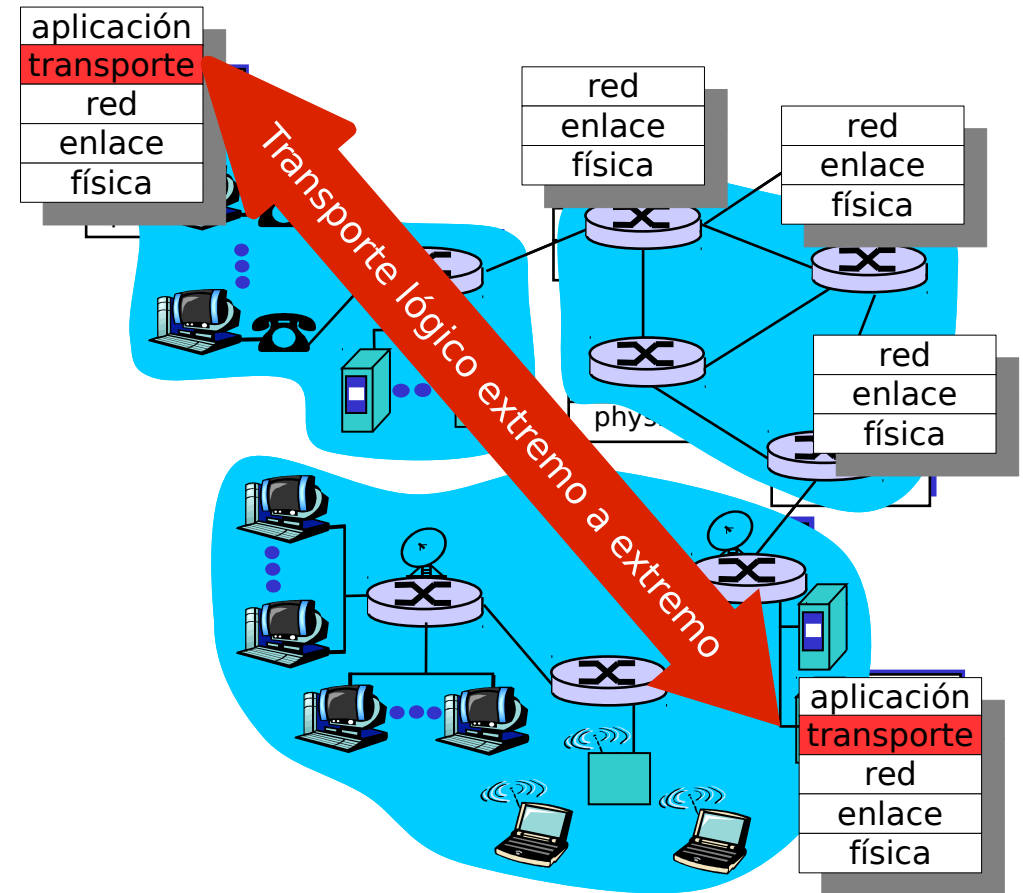
Entrega no confiable, sin orden  
(UDP)

- Extensión simple del IP “best-effort”

Servicios no disponibles

- Garantías de retardo

- Garantías de ancho de banda



# Capa de Transporte

3.1 Servicios de la capa de transporte

3.2 Multiplexado y demultiplexado

3.3 Transporte sin conexión: UDP

3.4 Principios de transferencia de datos confiable

3.5 Transporte orientado a conexión: TCP

Estructura de segmentos

Transferencia de datos confiable



Control de flujo

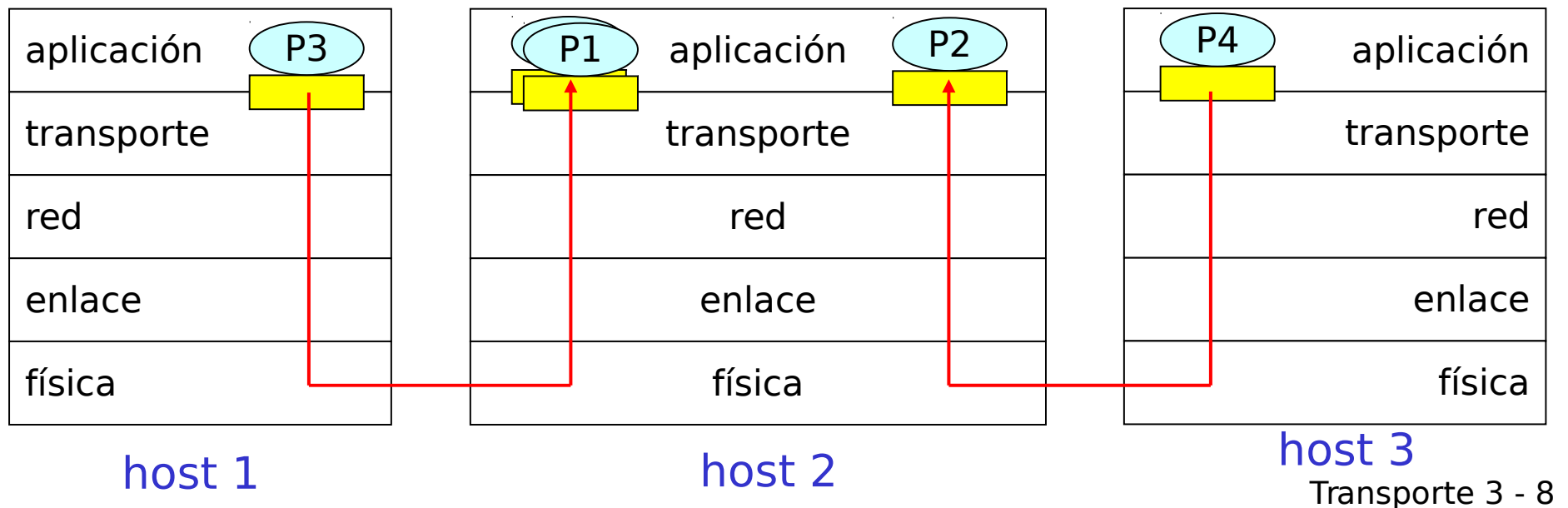
Administración de conexiones

3.6 Principios de control de congestión

3.7 Control de congestión en TCP

# Multiplexado/demultiplexado

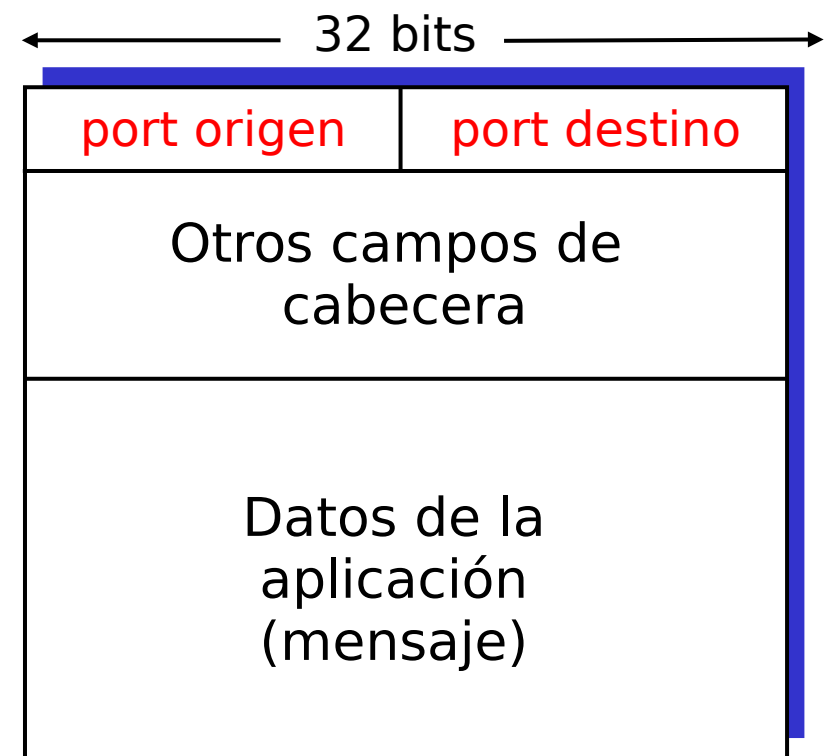
- Demultiplexado en el receptor
  - Enviando segmentos recibidos al socket correcto
  - Multiplexado en el emisor
  - Reuniendo datos de múltiples sockets, ensobrando datos con cabeceras (luego usadas para demultiplexar)
-  = socket       = proceso





# Cómo funciona el demultiplexado

- Host recibe datagramas IP
  - Cada datagrama tiene direcciones IP origen y destino
  - Cada datagrama lleva 1 segmento de capa de transporte
  - Cada segmento tiene números de port origen y destino
    - Números de port bien conocidos para aplicaciones específicas
- Host usa direcciones IP y números de port para dirigir segmentos al socket correcto



Formato de segmento TCP/UDP

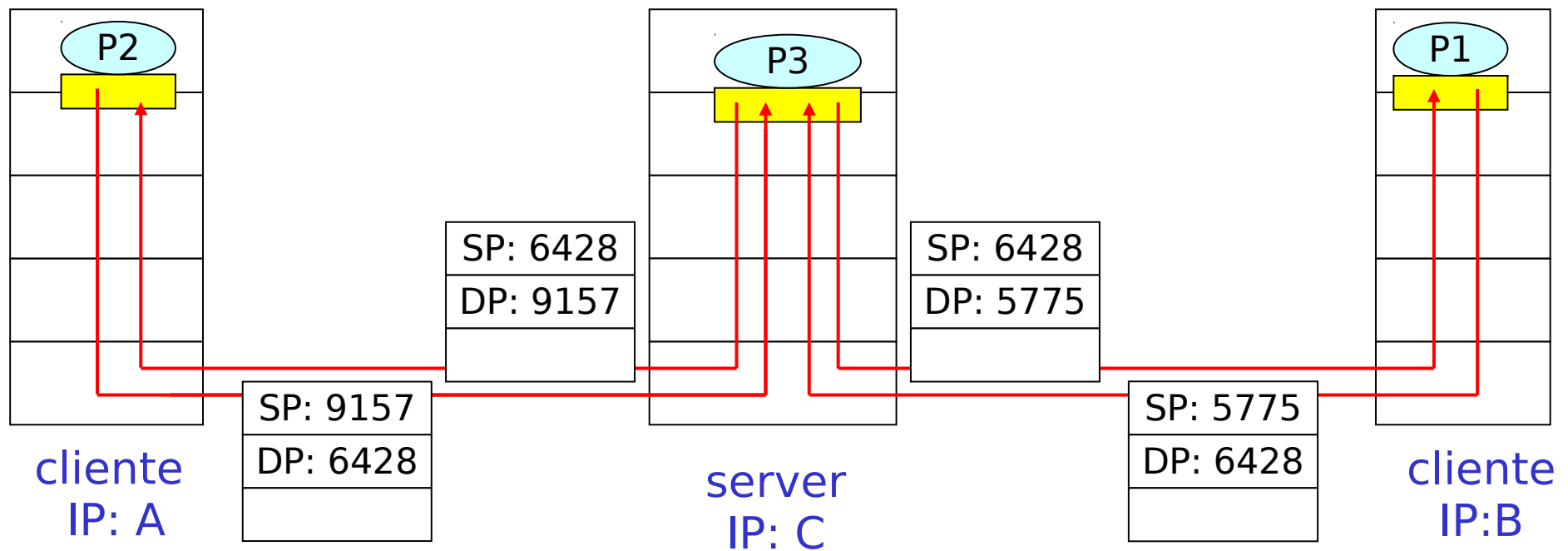
# Demultiplexado sin conexión

- Sockets asociados a números de port

```
DatagramSocket mySocket1 = new DatagramSocket(9111);
DatagramSocket mySocket2 = new DatagramSocket(9222);
```
- Socket UDP identificado por el par (dirección IP destino, número de port destino)
- Cuando el host recibe segmento UDP:
  - Observa port destino en el segmento
  - Dirige el segmento UDP al socket con ese número de port
- Datagramas IP con diferente dirección IP origen y/o port origen son dirigidos al mismo socket

# Demultiplexado sin conexión

```
DatagramSocket serverSocket = new DatagramSocket(6428);
```

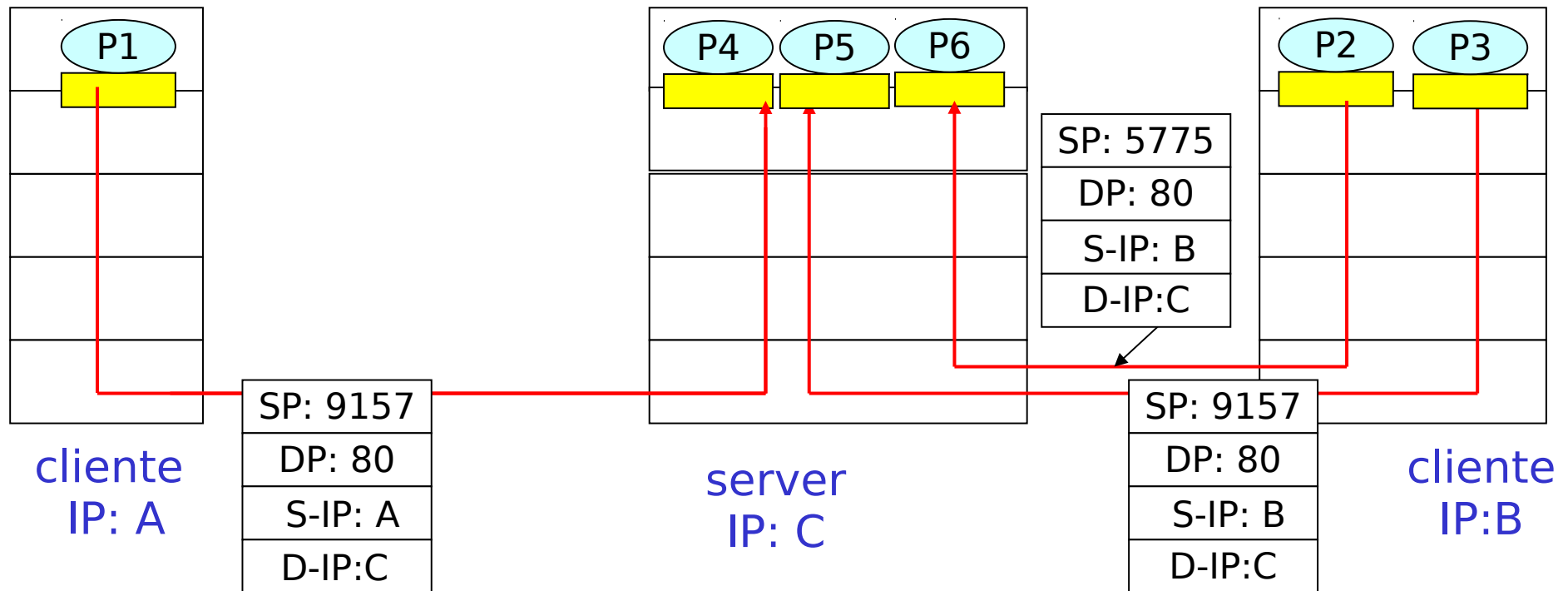


SP = Port origen  
DP = Port destino  
Port origen ofrece la dirección de retorno

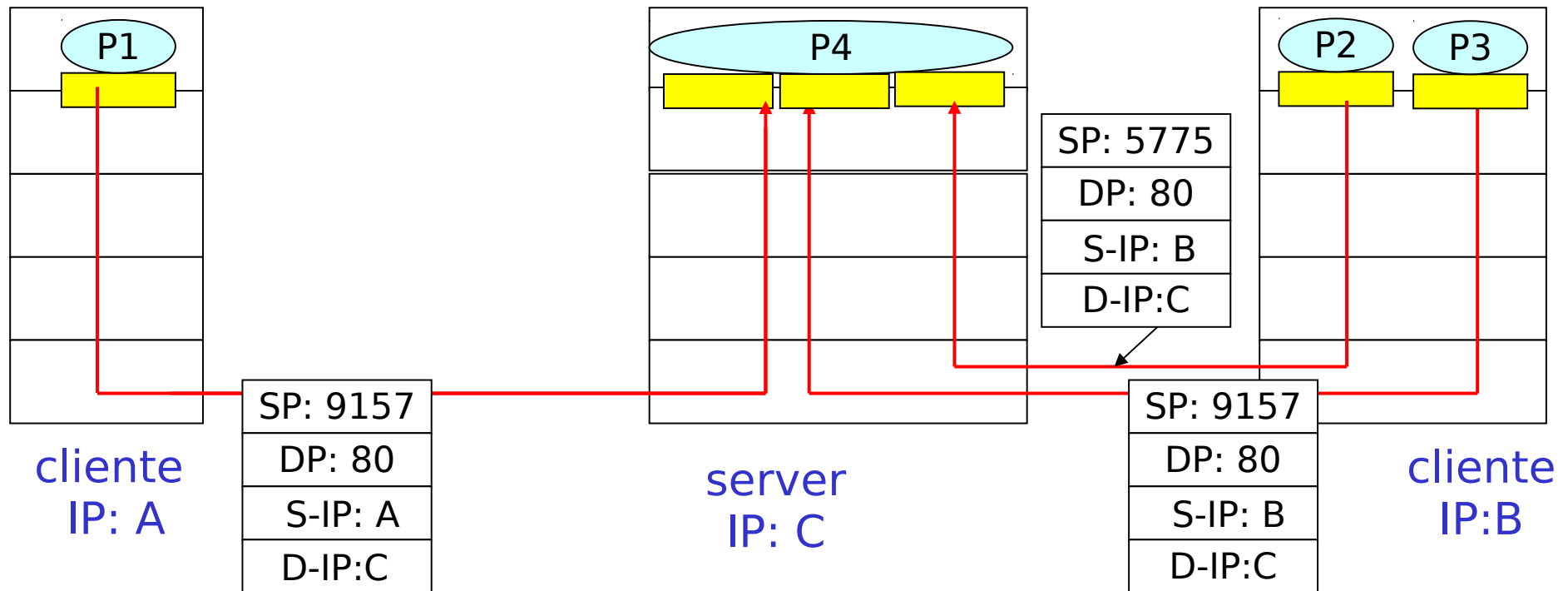
# Demultiplexado con conexión

- Socket TCP identificado por una 4-tupla
  - Dirección IP origen
  - Número de port origen
  - Dirección IP destino
  - Número de port destino
- Host receptor usa los cuatro valores para dirigir el segmento al socket correcto
- Host server puede soportar muchos sockets TCP simultáneos
  - Cada socket está identificado por su propia 4-tupla
- Los Web Servers tienen diferentes sockets para cada cliente que se conecta
  - HTTP no persistente tendrá diferente socket para cada request

# Demultiplexado con conexión



# Demultiplexado con conexión (Web Server c/threads)



# Capa de Transporte

3.1 Servicios de la capa de transporte

3.2 Multiplexado y demultiplexado

3.3 Transporte sin conexión:  
UDP

3.4 Principios de transferencia de datos confiable

3.5 Transporte orientado a conexión: TCP

Estructura de segmentos

Transferencia de datos confiable

Control de flujo

Administración de conexiones

3.6 Principios de control de congestión

3.7 Control de congestión en TCP

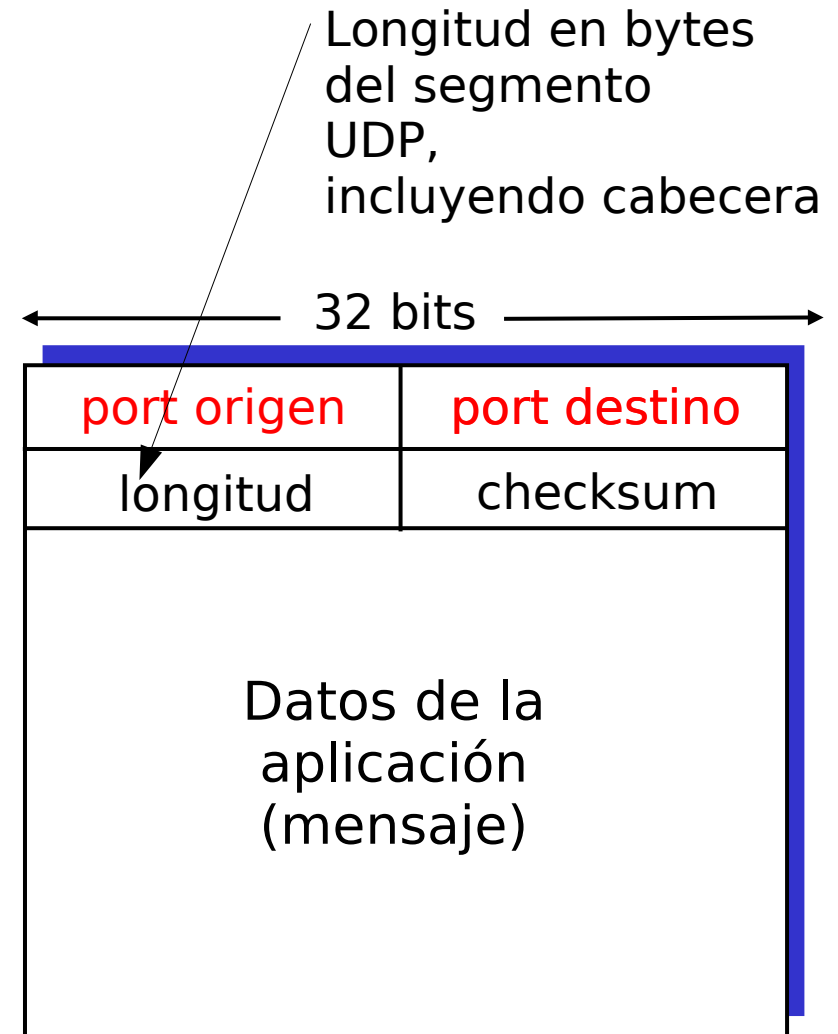
# UDP: User Datagram Protocol [RFC 768]

- Protocolo de transporte de Internet simple, esquelético
- Servicio “best effort”
- Los segmentos UDP pueden
  - Perderse
  - Entregarse fuera de orden
- Sin conexión:
  - No hay acuerdo previo entre emisor y receptor
  - Cada segmento UDP es independiente de los demás
- Por qué existe UDP?
- No hay establecimiento de conexión (que puede agregar retardo)
- Es simple: no hay estado de conexión en emisor ni receptor
- Cabecera de segmento pequeña
- Sin control de congestión, UDP no sufre demoras debido a dicho control



# UDP

- Usado con frecuencia para aplicaciones de *streaming* multimedia
  - Tolerantes a pérdidas
  - Sensibles a *rate*
- Otros usos de UDP
  - DNS
  - SNMP
- Transporte confiable sobre UDP: agregar confiabilidad en la capa de aplicación
  - Recuperación de errores específico de la aplicación



Formato de segmento UDP

# UDP checksum

- Objetivo
  - Detectar “errores” (bits intercambiados) en el segmento transmitido
- Emisor:
  - Trata el contenido del segmento como secuencias de enteros de 16 bits
  - Calcula el checksum (suma en complemento a 1 del contenido del segmento)
  - Coloca el valor del checksum en el campo *checksum* del mensaje UDP
- Receptor
  - Calcula checksum del segmento recibido
  - Verifica si el computado es igual al valor del campo checksum
    - NO -> Detectamos error
    - SI -> No detectamos error

*(¿pero puede haber error de todas maneras?)*

# Ejemplo de checksum

## RFC 1071: Internet Checksum

Al sumar números tenemos en cuenta el bit de acarreo que se suma al resultado

Ejemplo: sumar dos enteros de 16 bits

		1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
		1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
		<hr/>															
acarreo	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
		<hr/>															
suma		1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum		0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

# Capa de Transporte

3.1 Servicios de la capa de transporte

3.2 Multiplexado y demultiplexado

3.3 Transporte sin conexión: UDP

3.4 Principios de transferencia de datos confiable

3.5 Transporte orientado a conexión: TCP

Estructura de segmentos

Transferencia de datos confiable

Control de flujo

Administración de conexiones

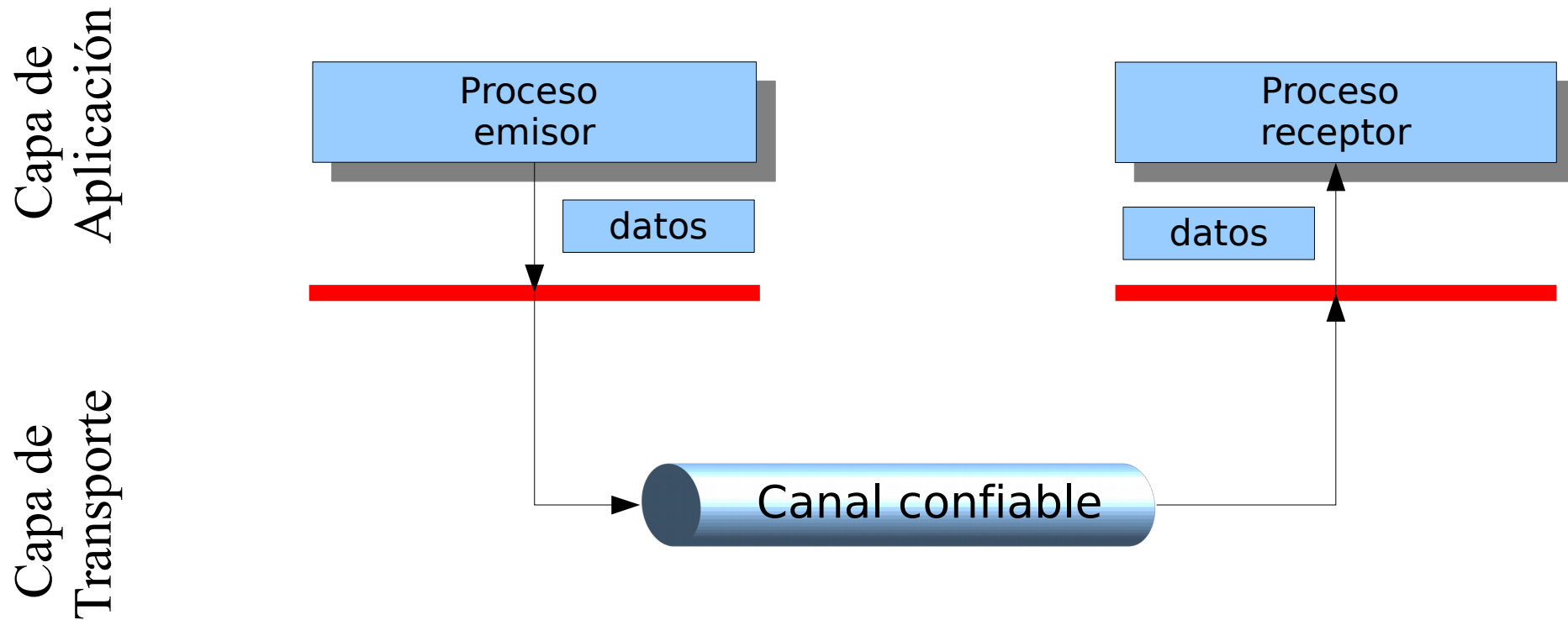
3.6 Principios de control de congestión

3.7 Control de congestión en TCP

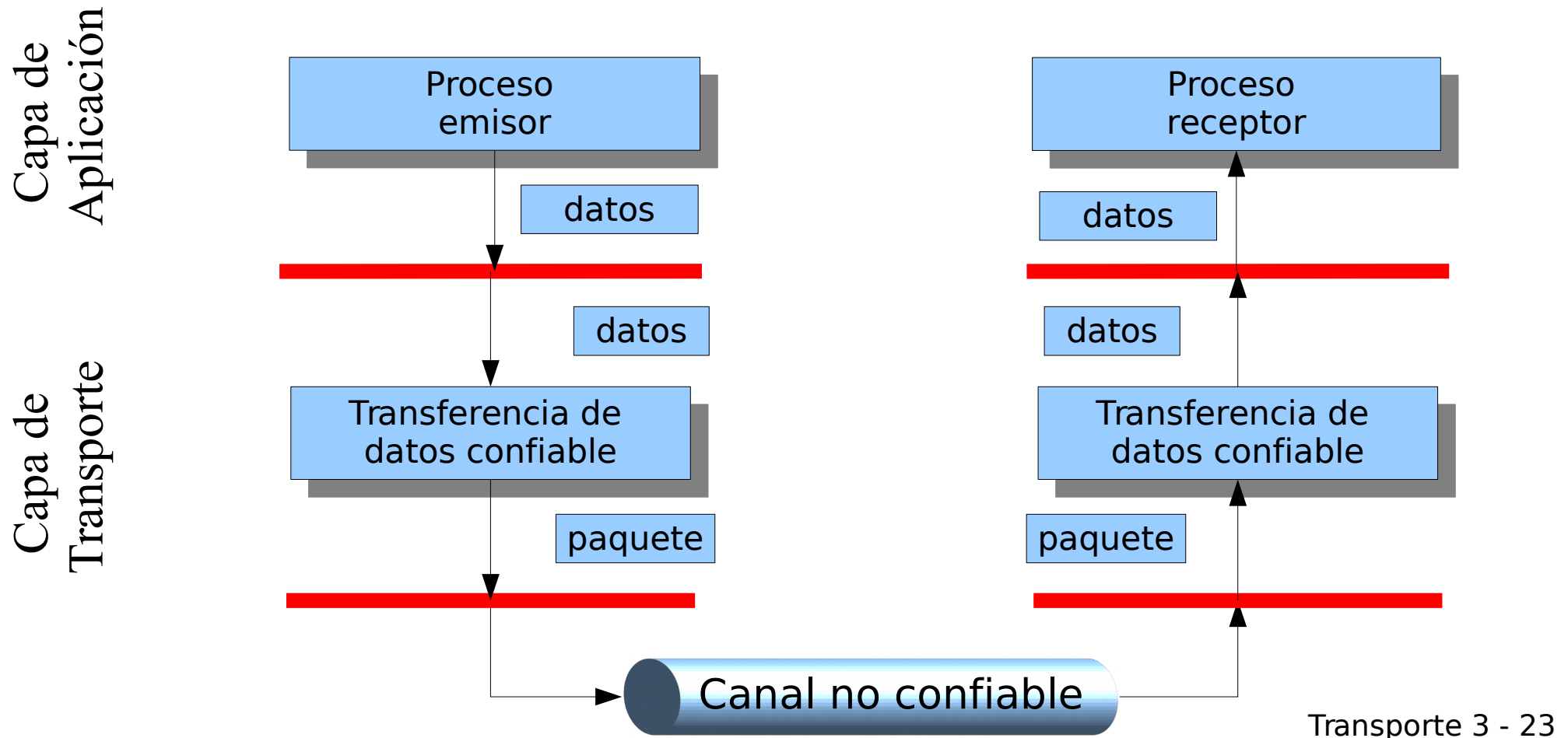
# Principios de transferencia de datos confiable

- Importante en capas de aplicación, transporte, enlace
- ¡En los *top-10* de los temas importantes de Redes!
- Las características del canal no confiable determinarán la complejidad del protocolo de datos confiable (**rdt**)
- Servicio provisto e implementación

# Servicio provisto



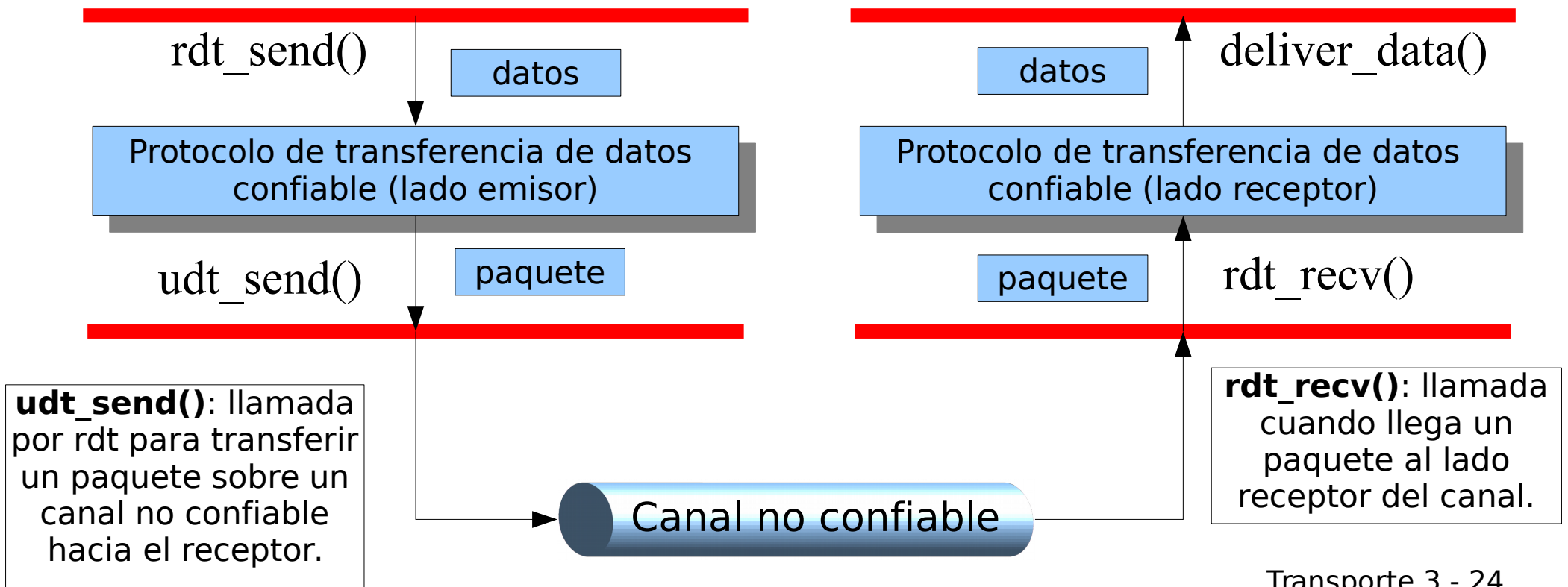
# Implementación del servicio



# Transferencia confiable de datos

**rdt\_send():** llamada desde arriba (por la aplicación). Se le pasan datos a enviar a la capa superior del receptor

**deliver\_data():** llamada por rdt para enviar datos a la capa superior

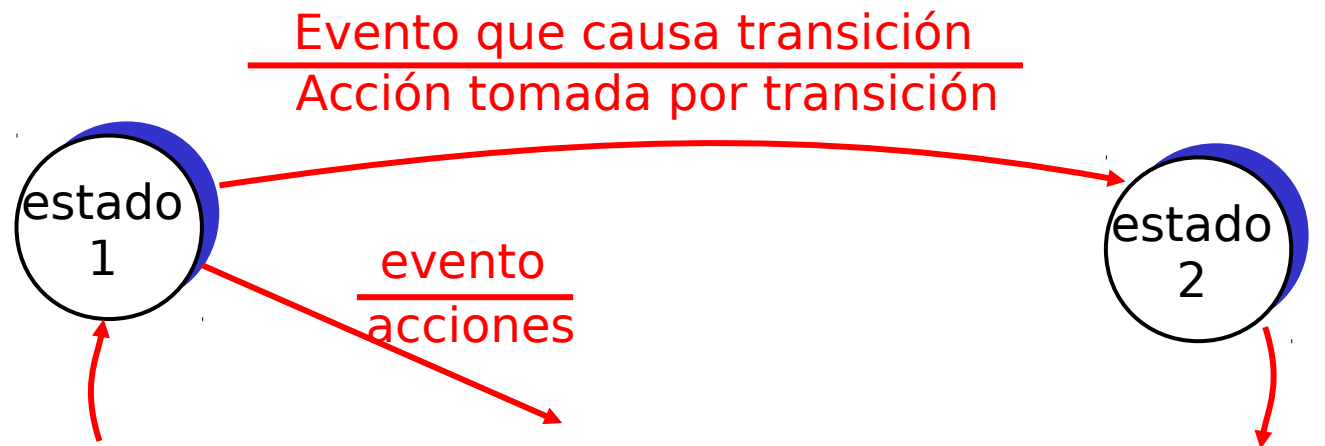




# Transferencia confiable de datos

- Desarrollaremos incrementalmente los lados emisor y receptor del protocolo rdt de transferencia confiable de datos
- Sólo consideraremos transferencia de datos unidireccional
  - Sin embargo, la información de control fluirá en ambas direcciones
- Usaremos máquinas de estados finitos para especificar emisor y receptor

Cuando nos hallamos en un “estado”, el siguiente estado está determinado de forma unívoca por el próximo evento



# RDT1.0

## Transferencia confiable sobre un canal confiable

El canal subyacente es perfectamente confiable

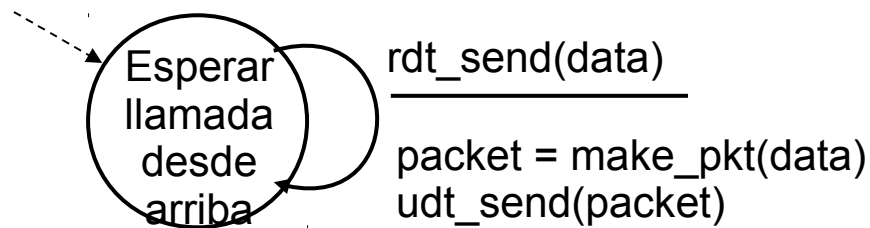
no hay errores de bits

no hay pérdida de paquetes

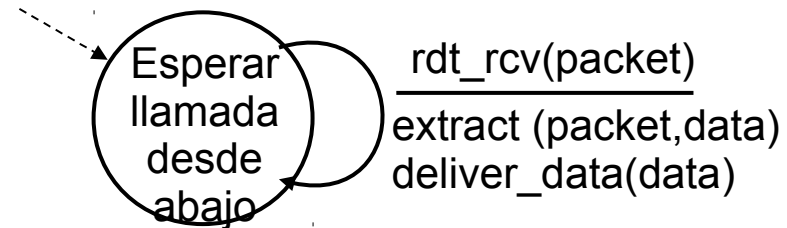
Automátas separados para emisor y receptor

Emisor envía datos al canal subyacente

El receptor lee datos del canal subyacente



emisor



receptor

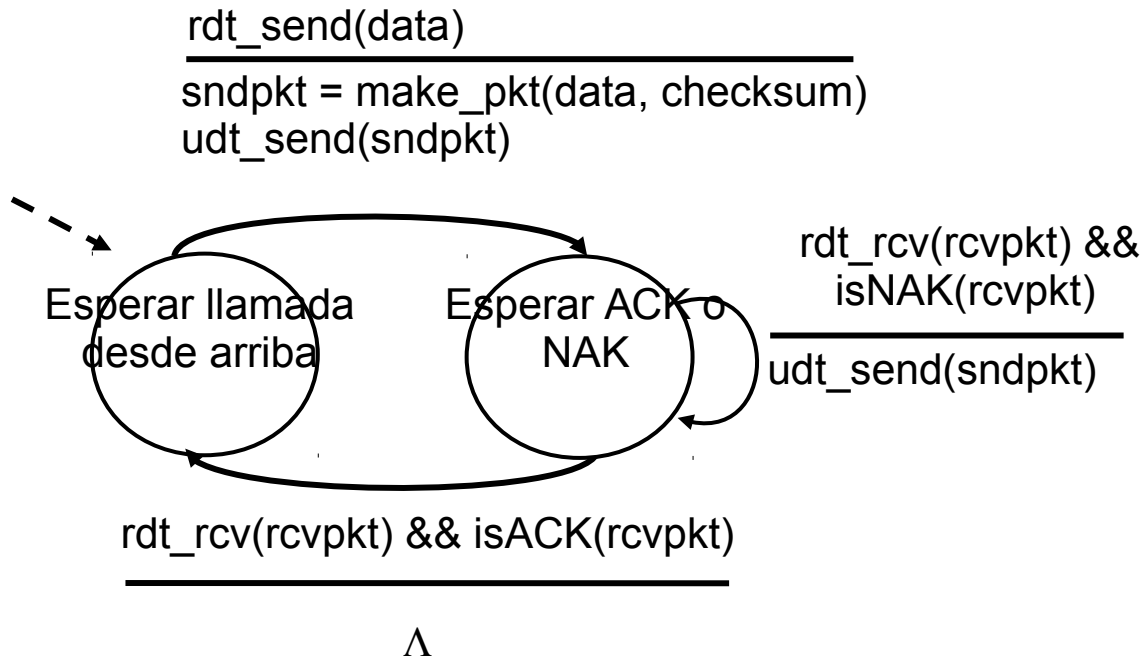
# RDT2.0

## Canal con errores de bits

- El canal subyacente puede cometer errores de bits (intercambiando 0 por 1 en el paquete)
  - El checksum permite detectar errores de bits
- Cómo recuperarse de los errores
  - Reconocimientos o acknowledgements (ACKs): el receptor explícitamente le dice al emisor que recibió el paquete OK
  - Reconocimientos negativos (NAKs): el receptor explícitamente dice al emisor que el paquete tenía errores
  - El emisor retransmite el paquete al recibir el NAK
- Nuevos mecanismos en RDT2.0
  - Detección de errores
  - Feedback del receptor: Mensajes de control ACK/NAK de receptor a emisor

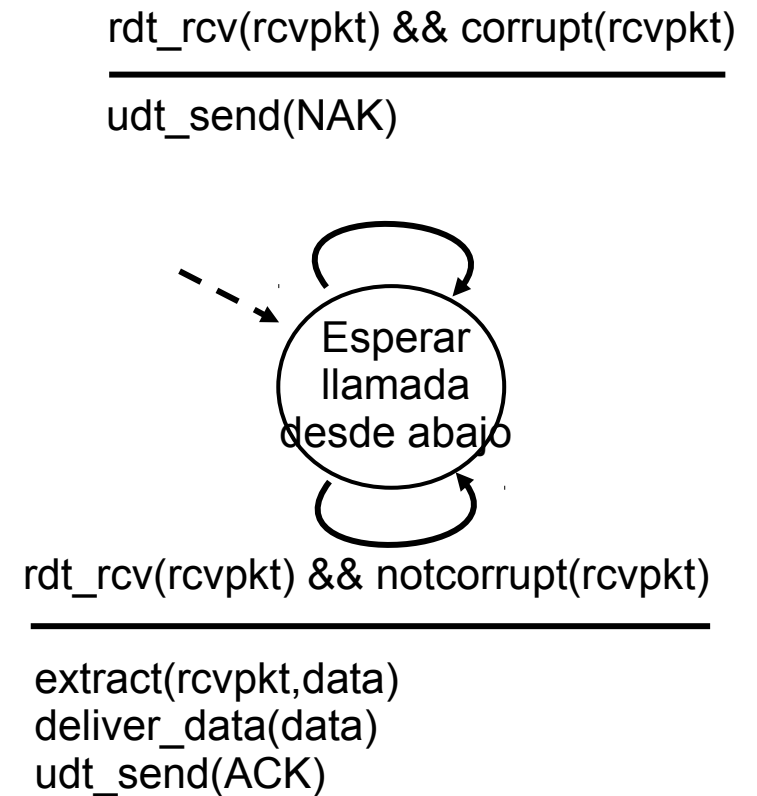
# RDT2.0

## Especificación del autómata finito



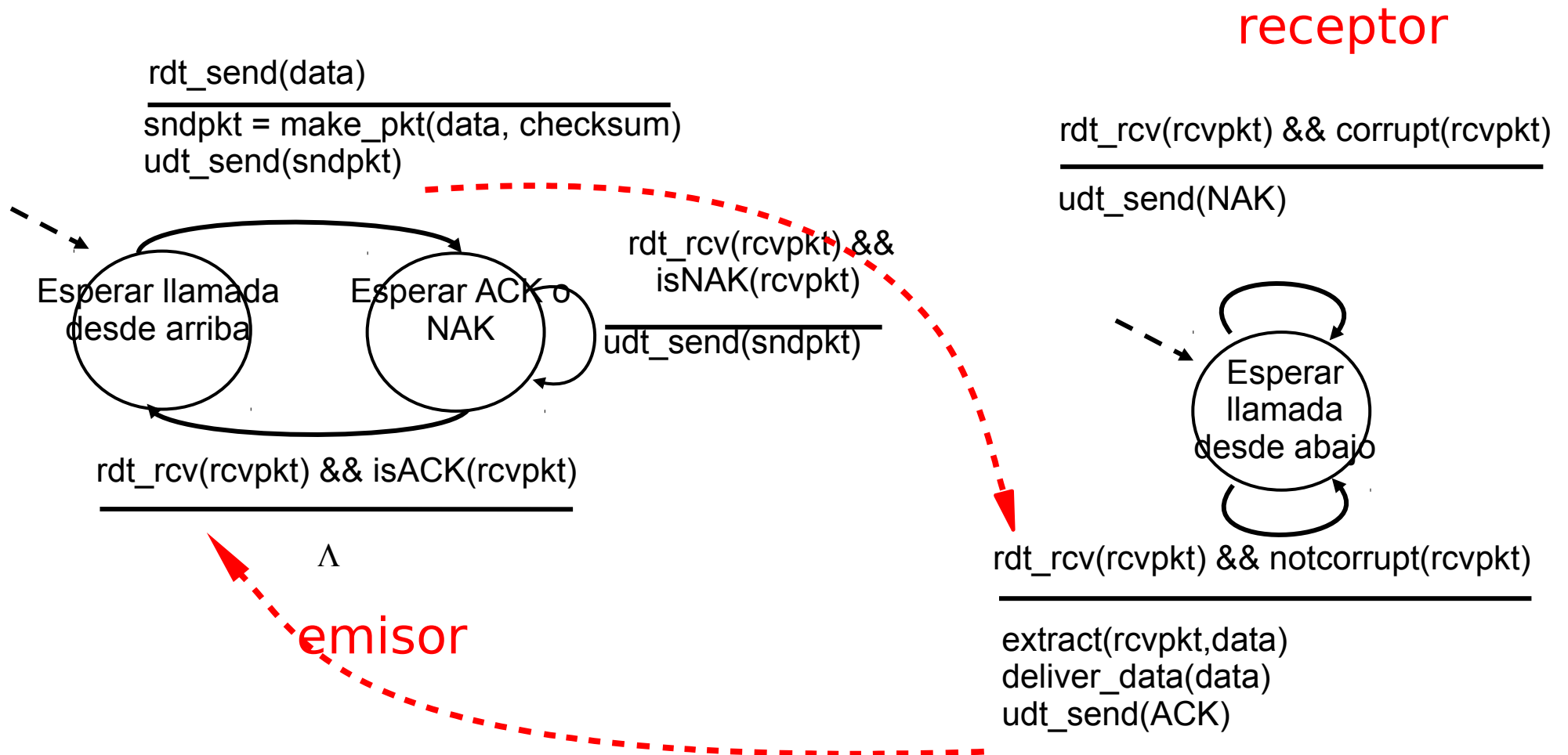
emisor

receptor



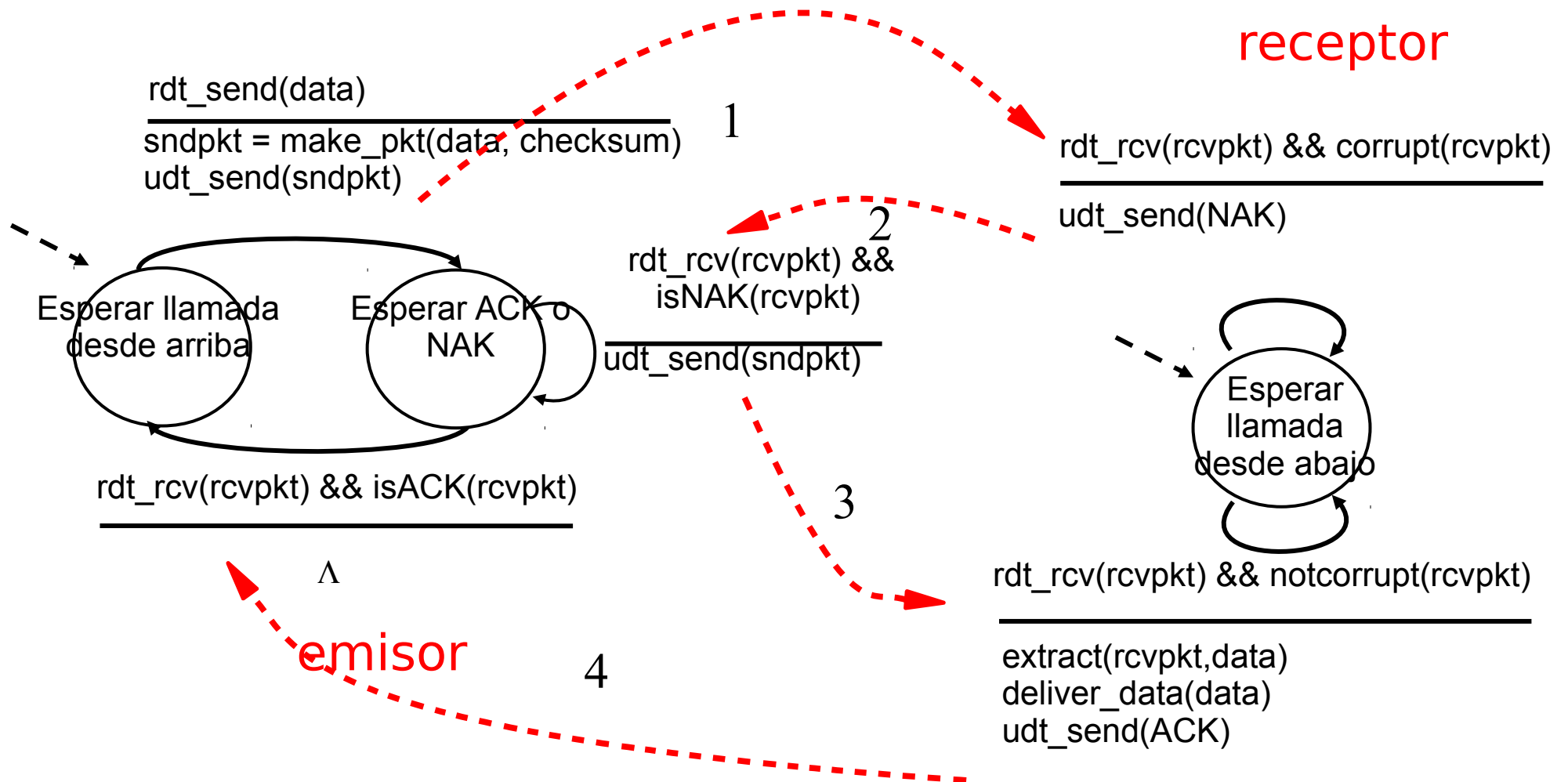
# RDT2.0

## Operación sin errores



# RDT2.0

## Escenario de error



# RDT2.0

## Tiene un defecto fatal!

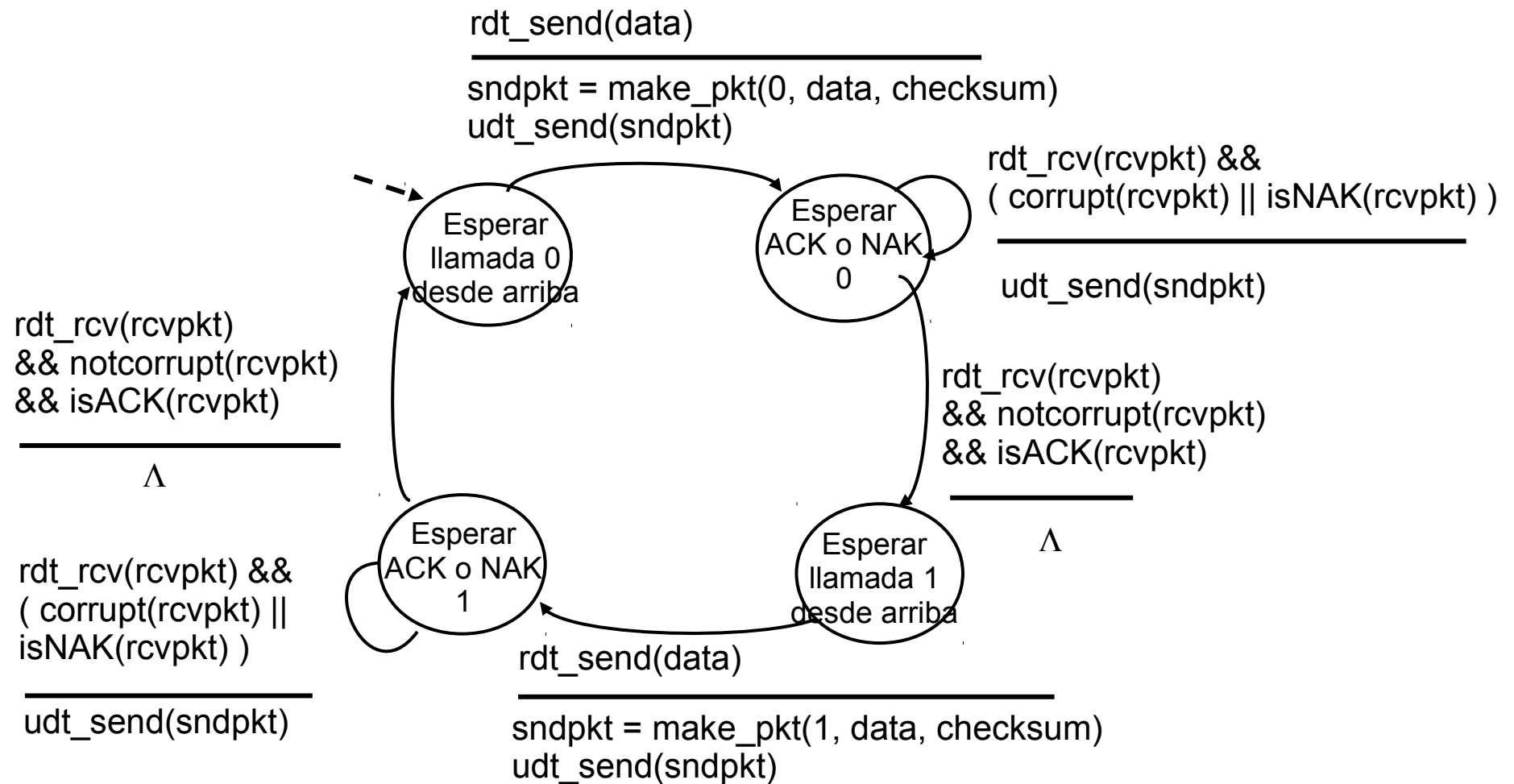
- Qué pasa si está corrupto el ACK o NAK?
  - El emisor no sabe qué pasó en el receptor
  - No se puede simplemente retransmitir, podríamos crear un duplicado
- Cómo manejar los duplicados
  - El emisor agrega un número de secuencia a cada paquete
  - El emisor retransmite el paquete actual si el ACK o NAK llega corrupto
  - El receptor descarta el paquete duplicado (no lo entrega a la entidad superior)

### — stop and wait —

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

# RDT2.1

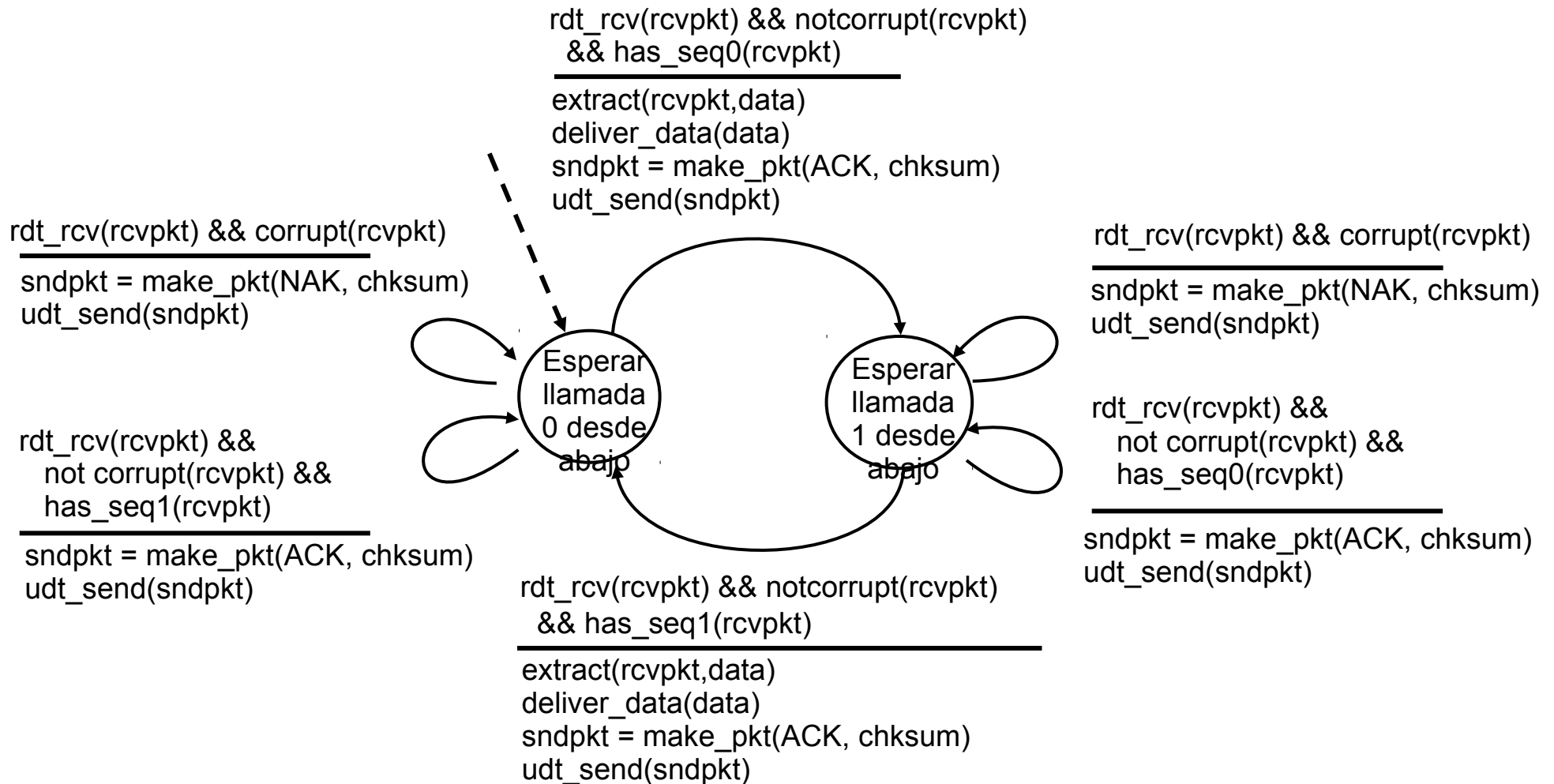
## Maneja los ACK/NAK corruptos - Emisor





# RDT2.1

## Maneja los ACK/NAK corruptos - Receptor



# RDT2.1

## Discusión

- Emisor
  - Se agrega número de secuencia al paquete
  - Basta con una secuencia de dos números (0,1)
    - Por qué?
  - Debe verificar si el ACK/NAK recibido está corrupto
  - El doble de estados
    - El estado debe “recordar” si el paquete actual tiene número de secuencia 0 o 1
- Receptor
  - Debe verificar si el paquete recibido es duplicado
    - El estado indica si se espera paquete con número de secuencia 0 o 1
  - El receptor no puede saber si su último ACK/NAK llegó OK al emisor

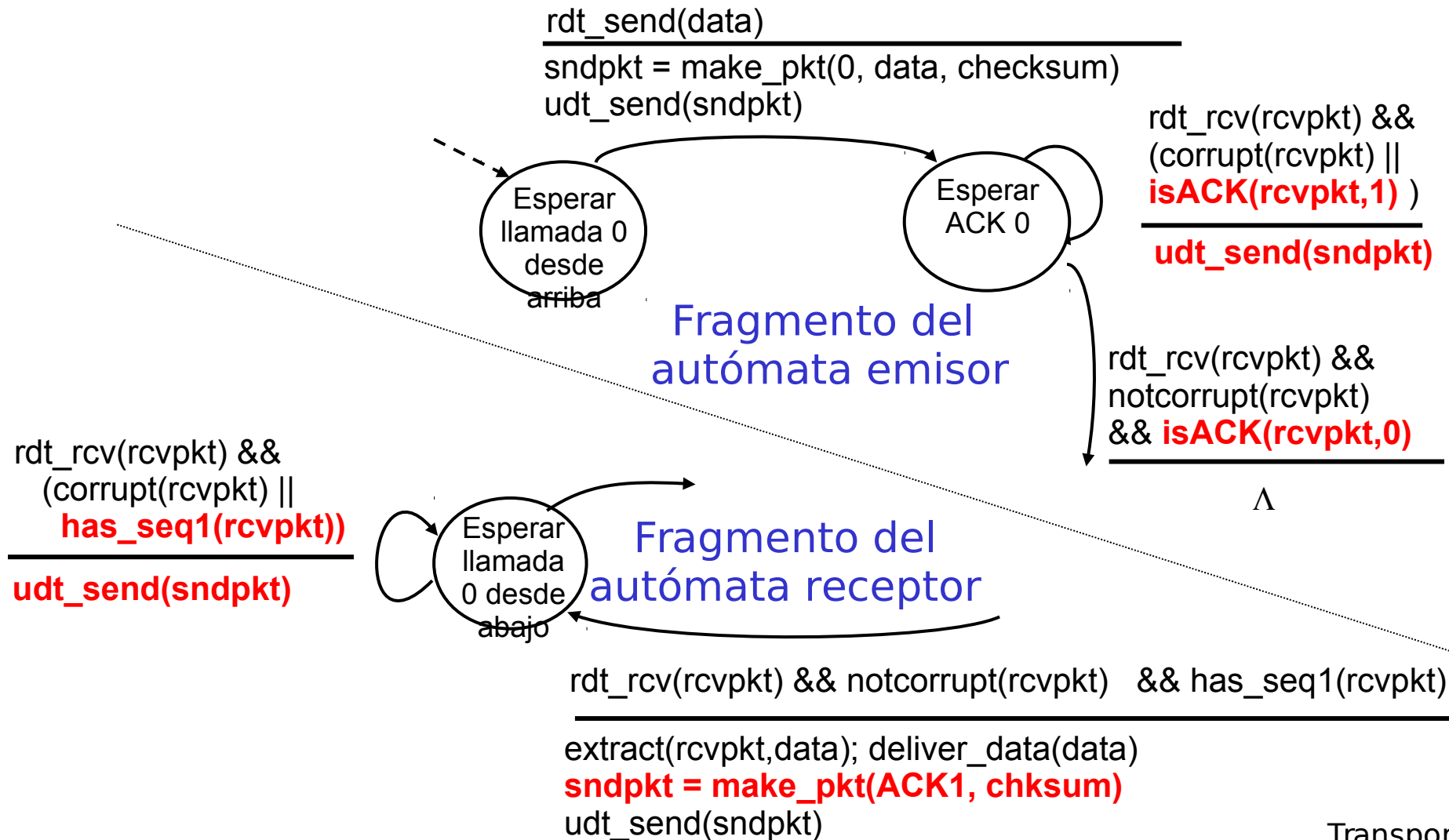
# RDT2.2

## Un protocolo sin NAKs

- Misma funcionalidad que RDT2.1 pero usando solamente ACKs
- En lugar de NAK, el receptor envía ACK para el último paquete recibido OK
  - El receptor debe explícitamente incluir el número de secuencia del paquete que está reconociendo
- Un ACK duplicado en el emisor resulta en la misma acción que un NAK: retransmitir el paquete actual

# RDT2.2

## Fragmentos de emisor y receptor

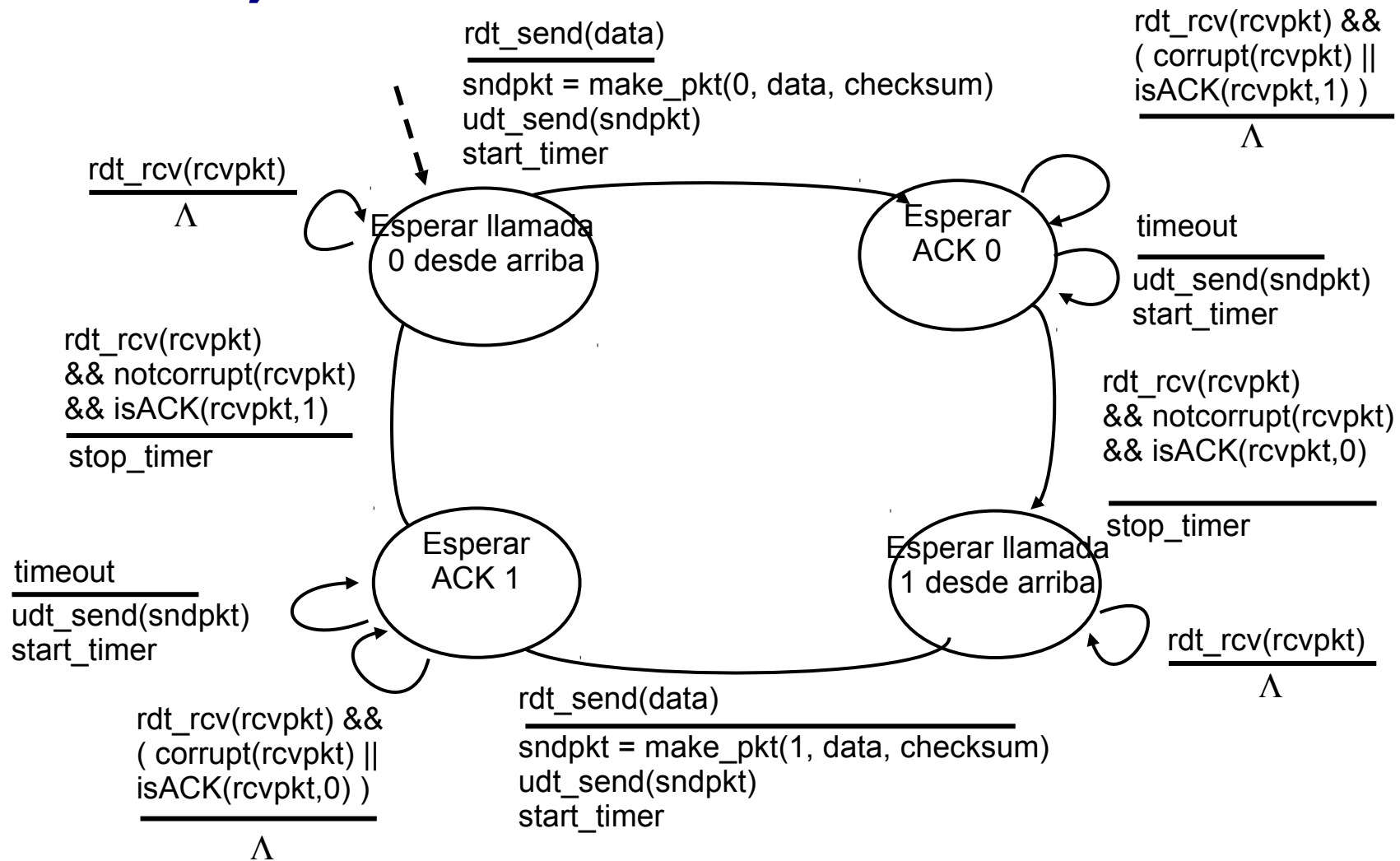


# RDT3.0

## Canales con errores y pérdidas

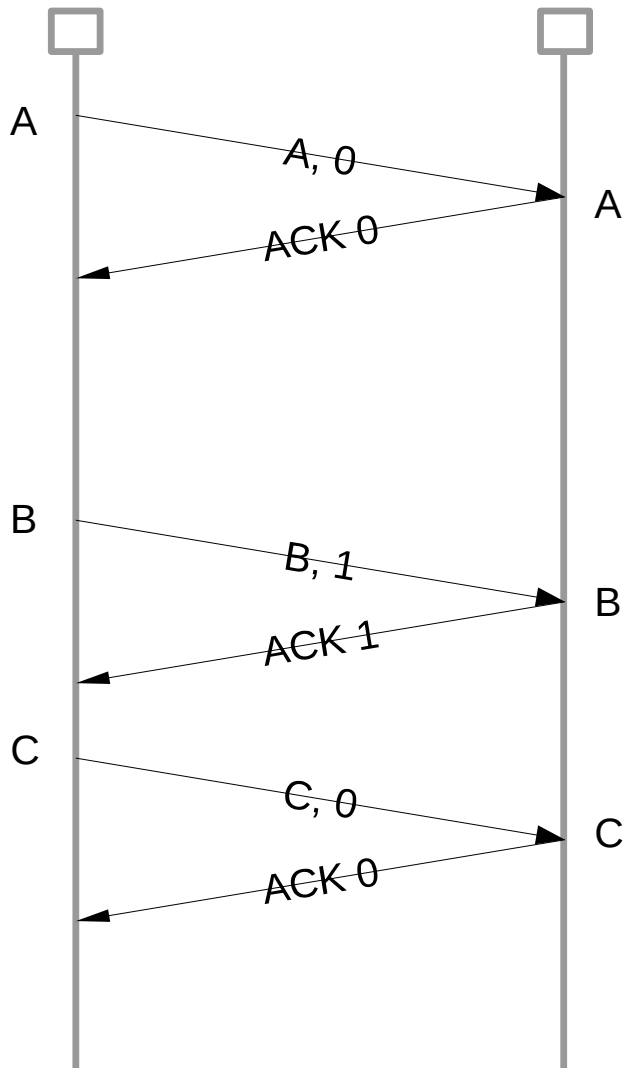
- Nueva suposición
- El canal subyacente puede también perder paquetes (datos o ACKs)
  - Los checksums, números de secuencia, ACKs y retransmisiones serán útiles pero no suficientes
- IDEA: El emisor espera una cantidad de tiempo “razonable” para que llegue el ACK
- Retransmite si no se recibe ACK en ese tiempo
- Si el paquete (o ACK) solamente se demoró (no se ha perdido):
  - La retransmisión generará un duplicado...
  - ¡Pero ya está solucionado con los números de secuencia!
- El receptor debe especificar el número de secuencia del paquete que está reconociendo
- Requiere un *timer* o reloj de cuenta a cero

# RDT3.0, emisor

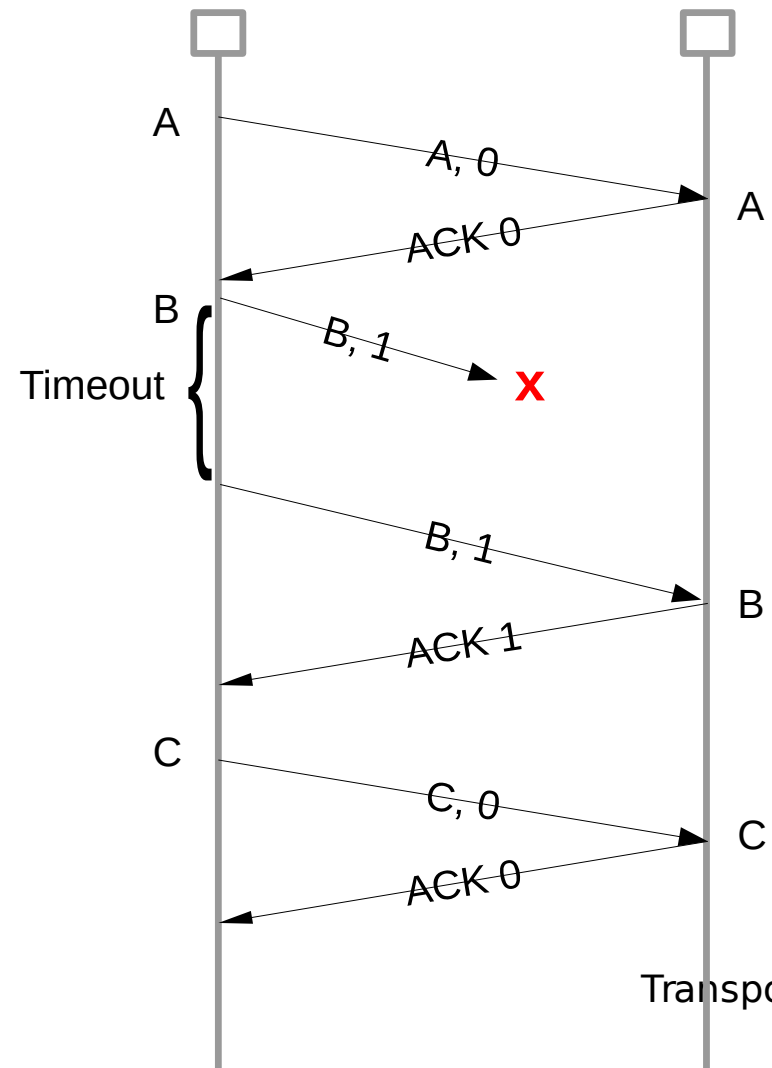


# RDT3.0 en acción

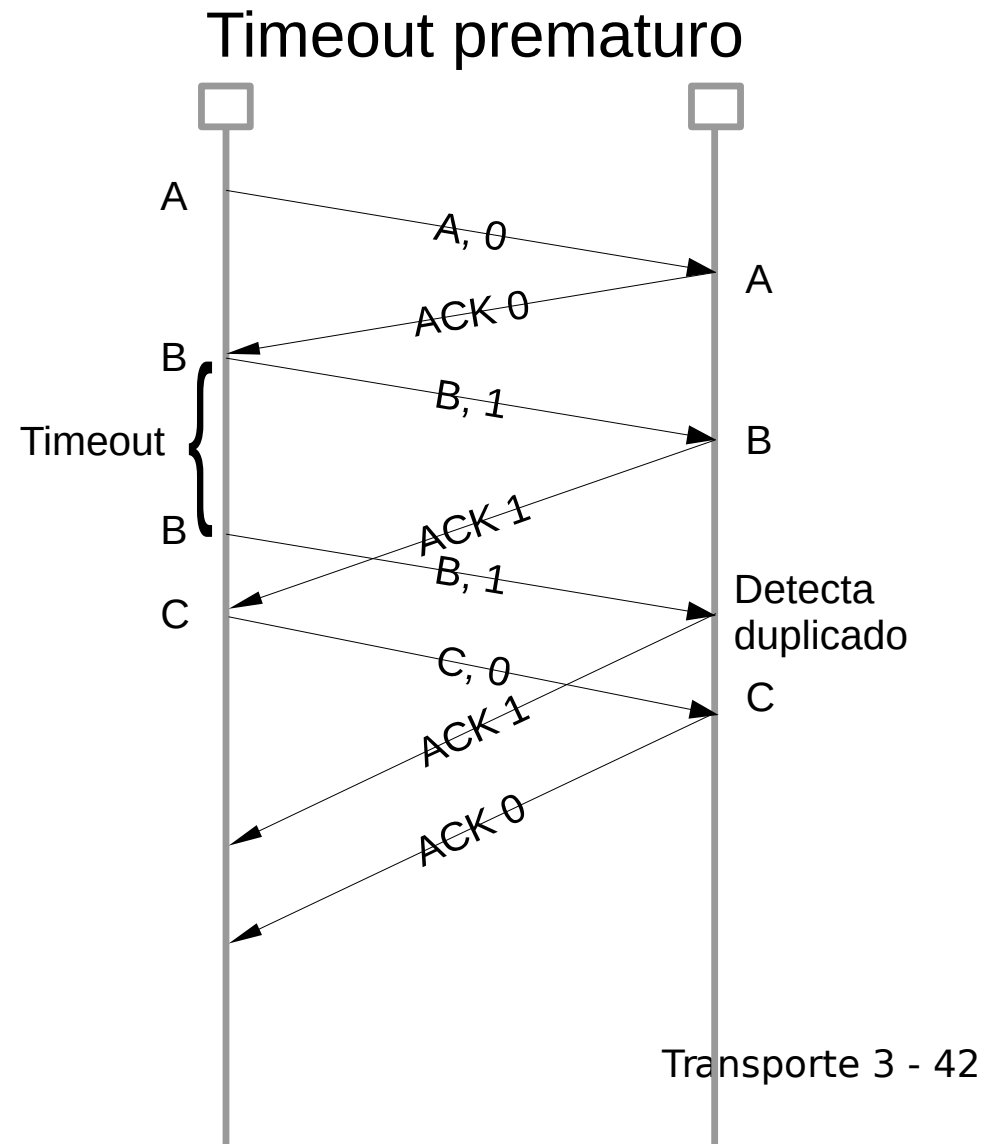
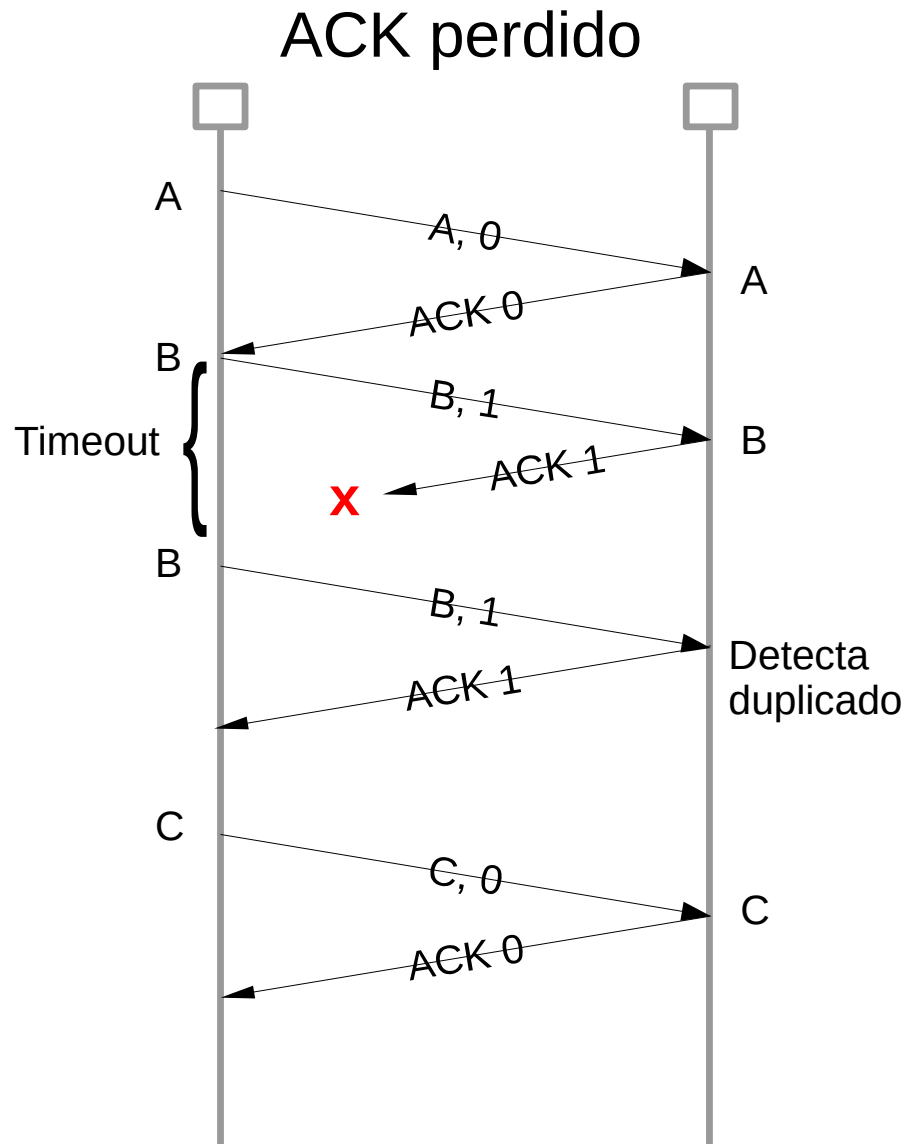
Operación sin pérdida



Paquete perdido

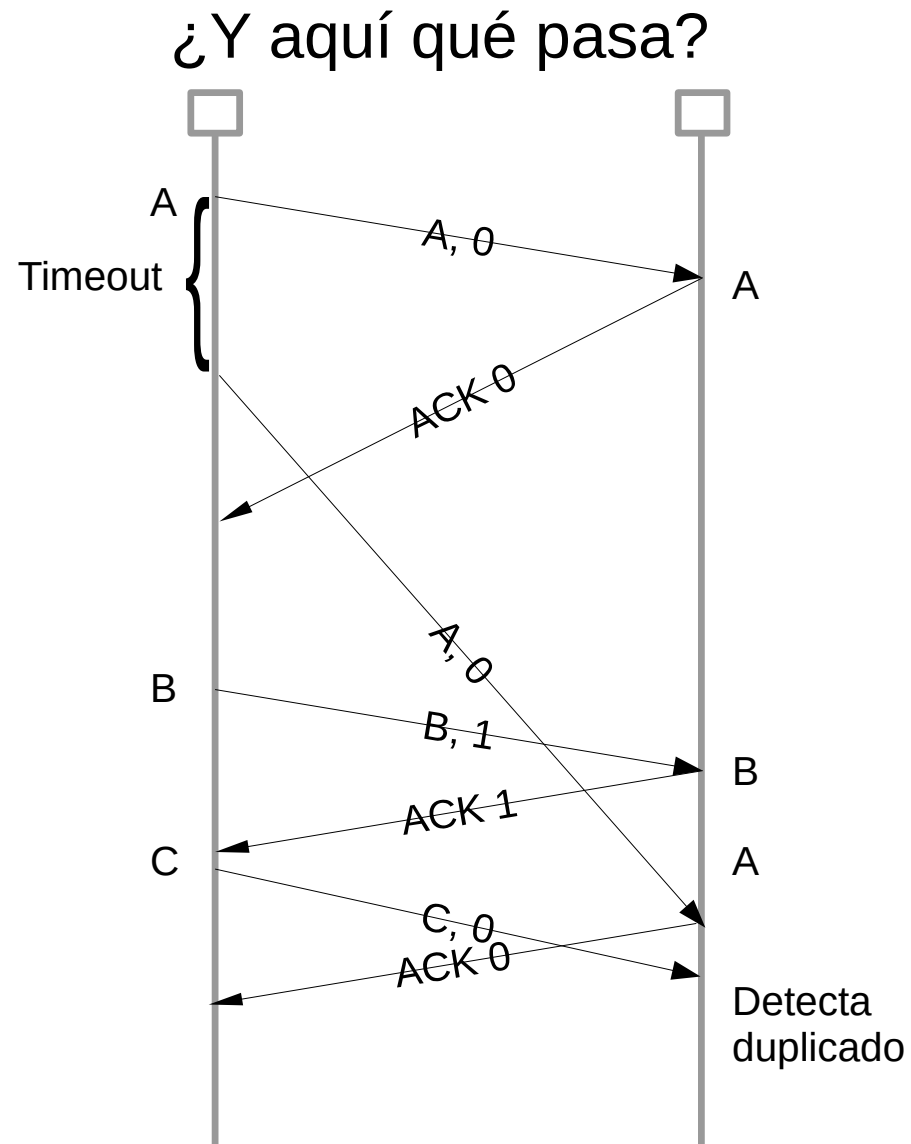


# RDT3.0 en acción





# RDT3.0 en acción



# Performance de RDT3.0

RDT3.0 funciona, pero con mala performance

Ejemplo: enlace de Gbps, retardo de propagación extremo a extremo de 15 ms (RTT = 30 ms), paquetes de 1KB

Designemos con  $U_{emisor}$  la utilización (fracción del tiempo en que el emisor está ocupado transmitiendo)

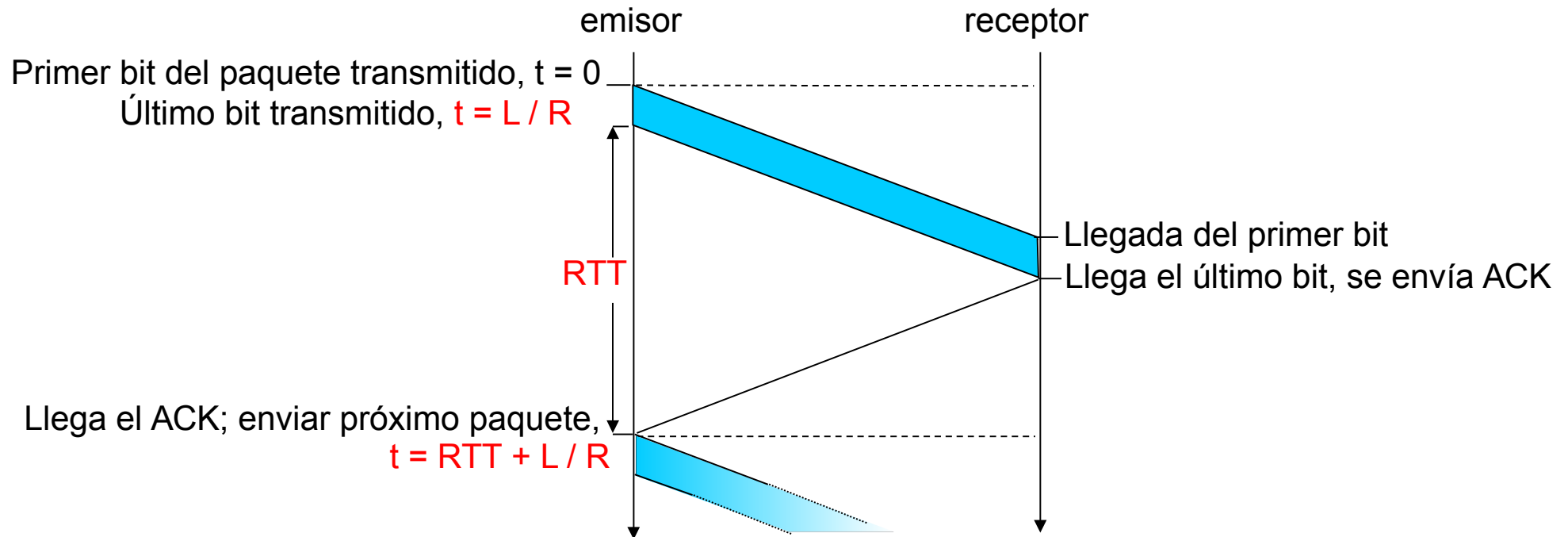
Un paquete de 1KB cada 30  $\mu s$  -> 267 kb/s de throughput sobre el enlace de 1Gbps

El protocolo utilizado limita el uso de los recursos físicos

$$T_{transm} = \frac{L(\text{longitud del paquete en bits})}{R(\text{velocidad de transmisión en bps})} = \frac{8000b}{10^9 \frac{b}{s}} = 8 \mu s$$

$$U_{emisor} = \frac{L/R}{RTT + L/R} = \frac{.008}{30.008} = 0.00027$$

# RDT3.0: operación stop-and-wait



$$U_{emisor} = \frac{L/R}{RTT + L/R} = \frac{.008}{30.008} = 0.00027$$

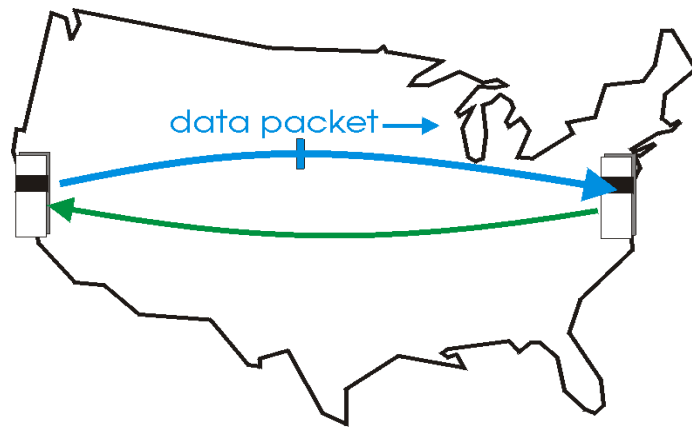
# Protocolos con pipeline

Pipelining: el emisor libera múltiples paquetes “en vuelo”, sin sus reconocimientos

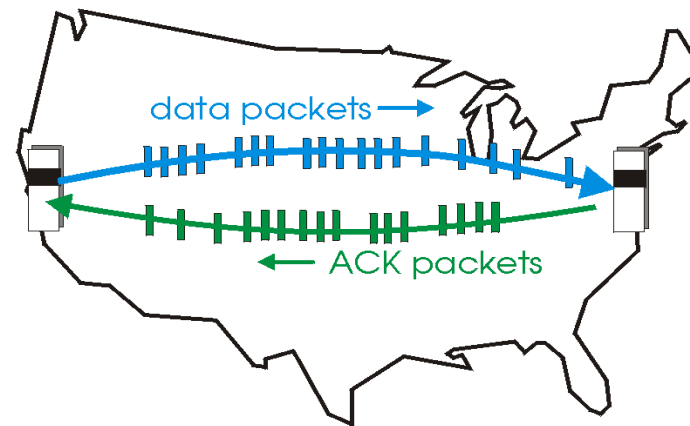
El rango de números de secuencia debe incrementarse

Debe existir buffering en emisor y/o receptor

Dos formas genéricas de protocolos con pipeline: Atrás N (go-Back-N) y repetición selectiva (selective repeat)

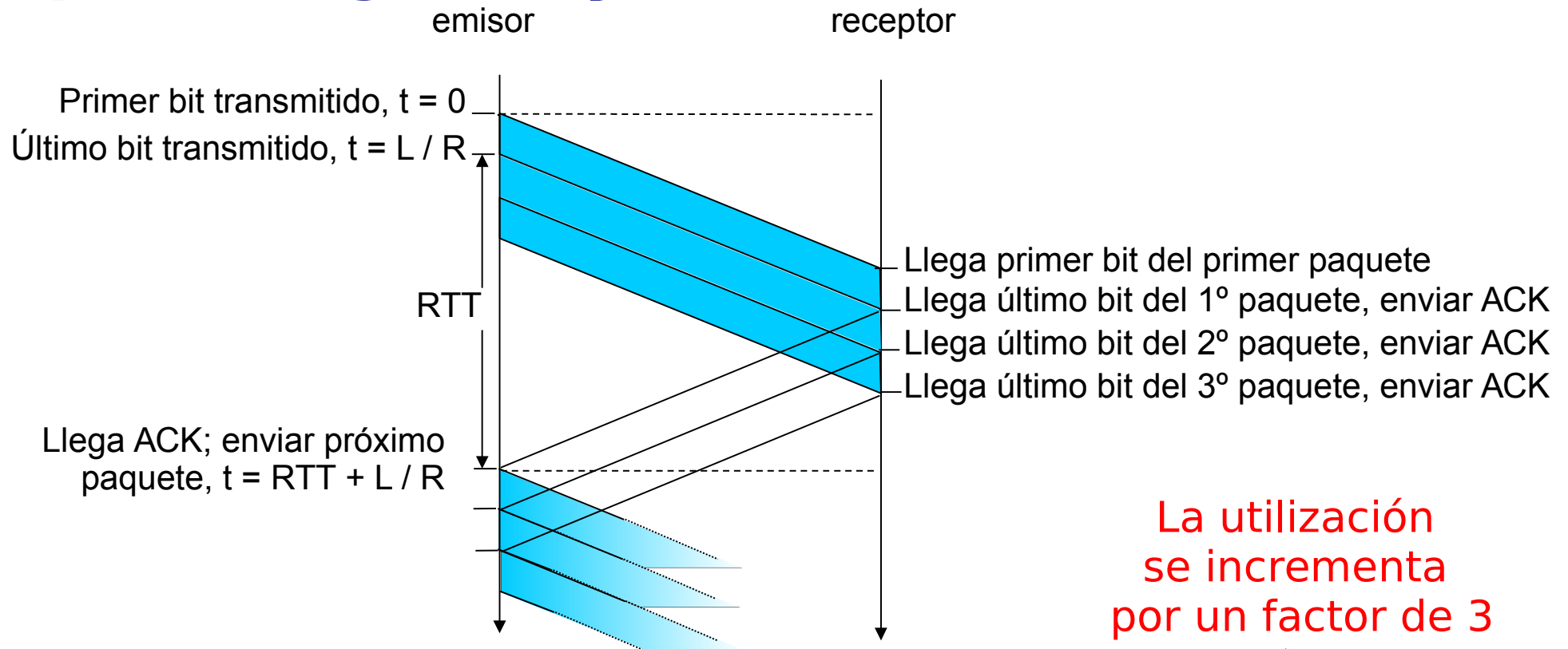


(a) a stop-and-wait protocol in operation



(b) a pipelined protocol in operation

# Pipelining: mayor utilización



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

# Estrategias con pipeline

## Go-back-N (GBN)

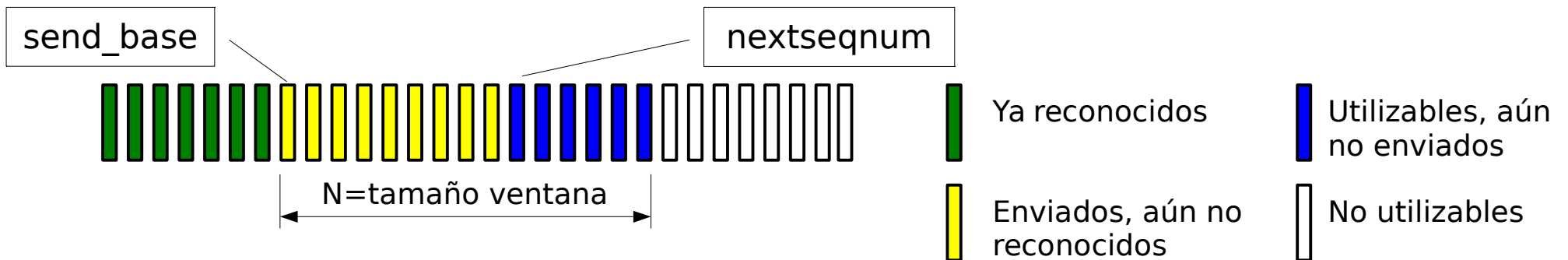
- El emisor puede mantener hasta N paquetes sin ACK
  - El receptor sólo envía reconocimientos acumulativos y no tiene un buffer
  - No emite reconocimientos cuando ve un “hueco”
- Emisor mantiene timer para el paquete más antiguo sin reconocer
  - Al expirar, retransmite todo lo que esté sin reconocimiento

## Selective Repeat (SR)

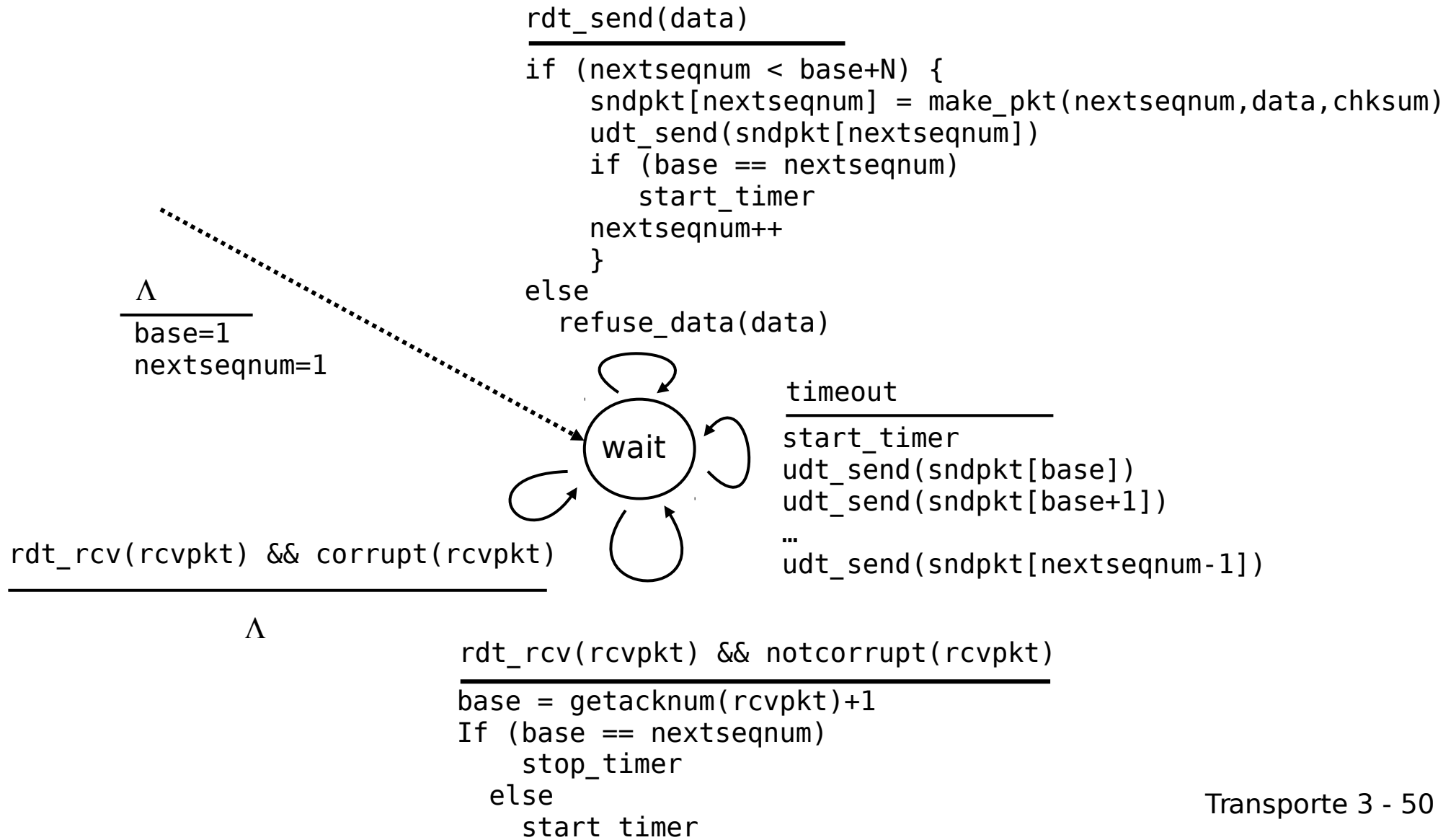
- El emisor puede mantener hasta N paquetes sin ACK
  - El receptor envía reconocimientos individuales por cada paquete y bufferiza una secuencia de paquetes
- Emisor mantiene timer para cada paquete sin reconocer
  - Al expirar, retransmite únicamente ese paquete sin reconocimiento

# Go-Back-N (GBN)

- Emisor
  - Números de secuencia sobre k bits en la cabecera del paquete
  - “Ventana” de hasta N paquetes consecutivos sin reconocer
  - ACK(n): Reconoce todos los paquetes hasta el que tiene número de secuencia N inclusive (“ACK acumulativo”)
    - Pueden recibirse ACKs duplicados
  - Timer para cada paquete en vuelo
    - timeout(n): retransmitir paquete n y todos los que tienen número de secuencia superior en la ventana



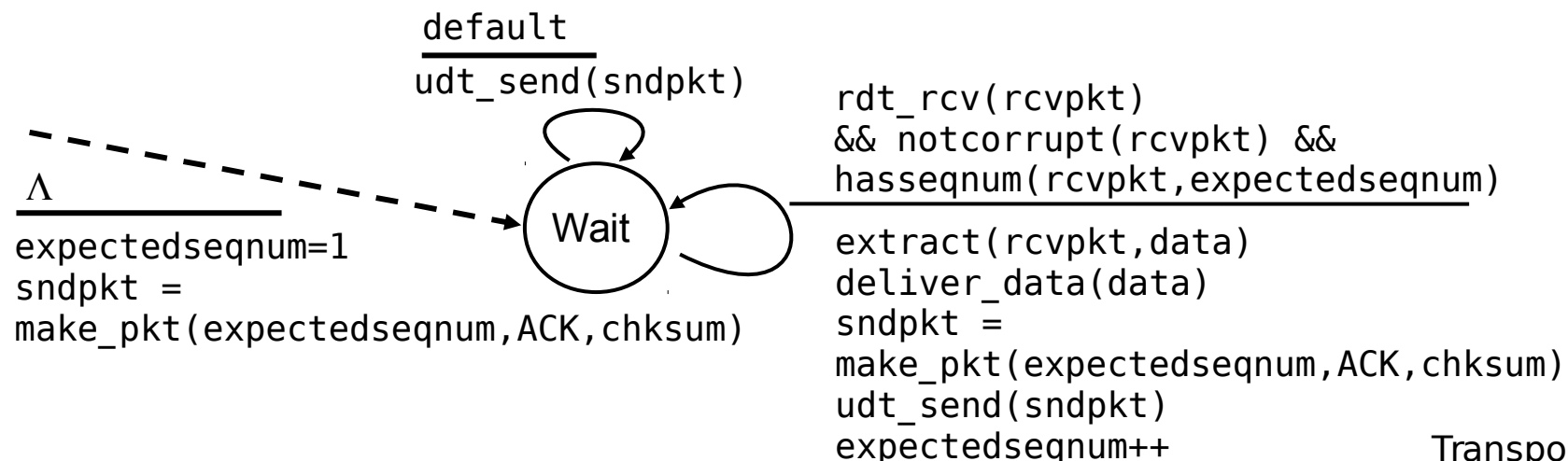
# GBN: Autómata extendido del emisor





# GBN: Autómata extendido del receptor

- ACK-solamente
  - Siempre envía ACK para el paquete recibido correctamente con el número de secuencia en orden correcto más alto
    - Puede generar ACKs duplicados
    - Solamente hace falta recordar expectedseqnum
- Paquete fuera de orden
  - Descartar (no bufferizar) -> no hay bufferización en el receptor
  - Re-reconocer paquete con # sec más alto en orden correcto



# GBN en acción

## Emisor (N=4)

0 1 2 3 4 5 6 7 8  
0 1 2 3 4 5 6 7 8  
0 1 2 3 4 5 6 7 8  
0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8  
0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8  
0 1 2 3 4 5 6 7 8  
0 1 2 3 4 5 6 7 8  
0 1 2 3 4 5 6 7 8

## Emisor

Envía 0  
Envía 1  
Envía 2  
Envía 3  
(espera)

Recibe A0, envía 4  
Recibe A1, envía 5

Ignora A1 duplicado

Timeout paquete 2

Envía 2  
Envía 3  
Envía 4  
Envía 5

## Receptor

Recibe 0, envía A0  
Recibe 1, envía A1

Recibe 3, descarta,  
reenvía A1

Recibe 4, descarta,  
reenvía A1

Recibe 5, descarta,  
reenvía A1

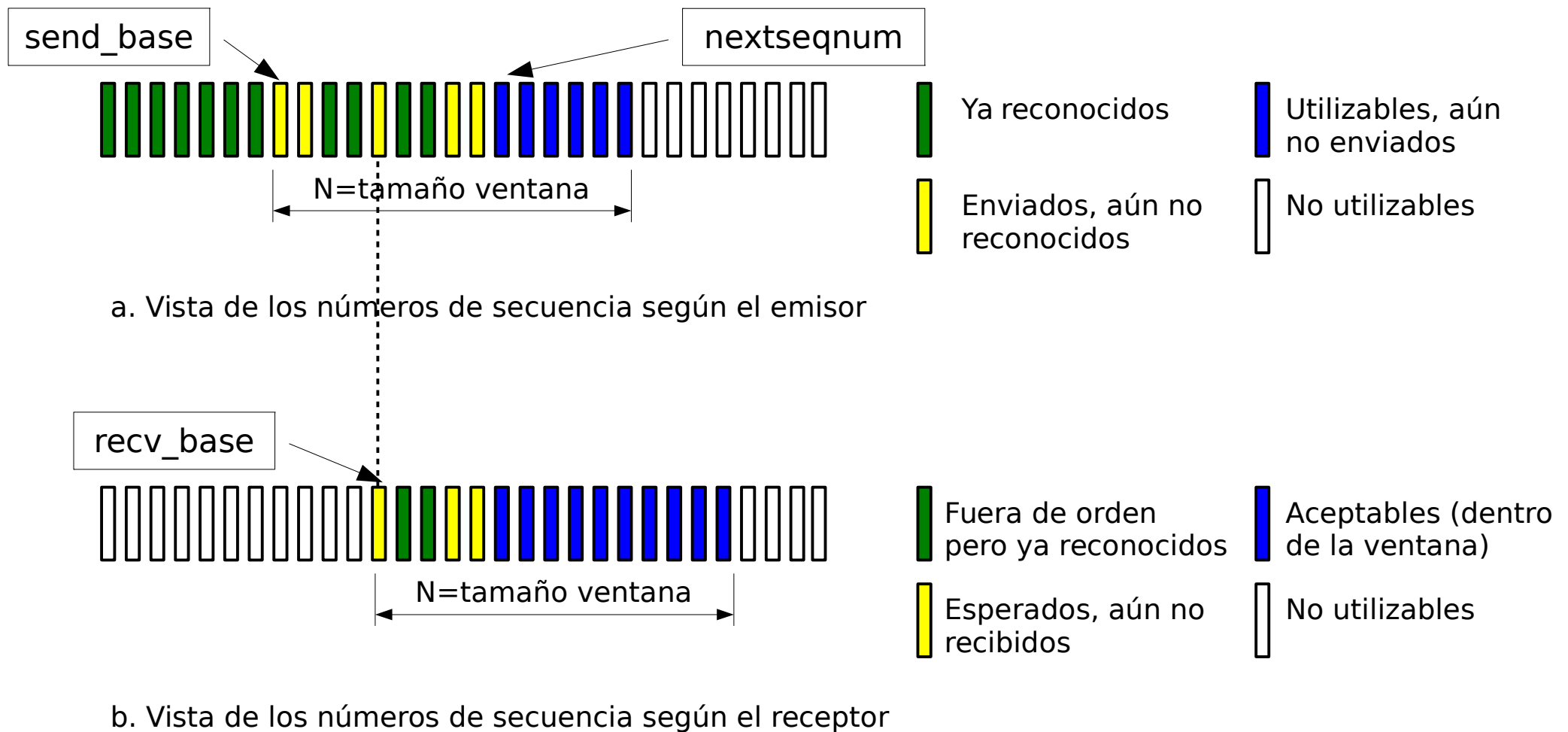
Recibe 2, entrega, envía A2  
Recibe 3, entrega, envía A3  
Recibe 4, entrega, envía A4  
Recibe 5, entrega, envía A5

# SR - Repetición selectiva

- El receptor individualmente reconoce todos los paquetes correctamente recibidos
  - Bufferiza paquetes según se necesite para eventualmente entregarlos en orden a la capa superior
- El emisor solamente reenvía paquetes para los cuales no ha recibido ACK
  - El emisor mantiene un timer para cada paquete sin ACK
- Ventana en el emisor
  - N números de secuencia consecutivos
  - Nuevamente se limitan los números de secuencia de los paquetes enviados, aún sin reconocimiento

# Repetición selectiva

## Ventanas emisor y receptor



# Repetición selectiva

## Emisor

Datos desde arriba:

Si el siguiente número de  
secuencia está en 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 sin  
reconocer más pequeño,  
avanzar la base de ventana  
al siguiente número de  
secuencia sin reconocer

## Receptor

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

Enviar ACK(n)

Si fuera de orden: bufferizar

Si en orden: entregar (también  
entregar los paquetes  
bufferizados, en orden),  
avanzar ventana al siguiente  
paquete aún no recibido

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

ACK(n)

En otro caso

Ignorar

# Repetición selectiva en acción

## Emisor (N=4)

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

## Emisor

Envía 0

Envía 1

Envía 2

Envía 3

(espera)

Recibe A0, envía 4

Recibe A1, envía 5

Marca 3 Llegado

Timeout paquete 2

Envía 2

Marca 4 Llegado

Marca 5 Llegado

## Receptor

Recibe 0, envía A0

Recibe 1, envía A1

Recibe 3, bufferiza,  
envía A3

Recibe 4, buff, envía A4  
Recibe 5, buff, envía A5

Recibe 2, entrega 2 a 5,  
envía A2

# Repetición selectiva

## Ejemplo

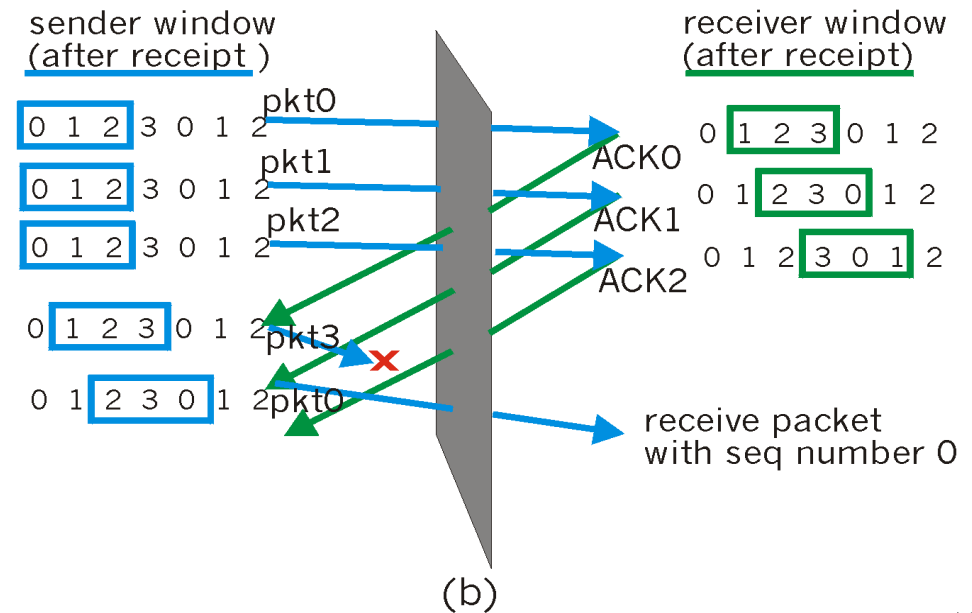
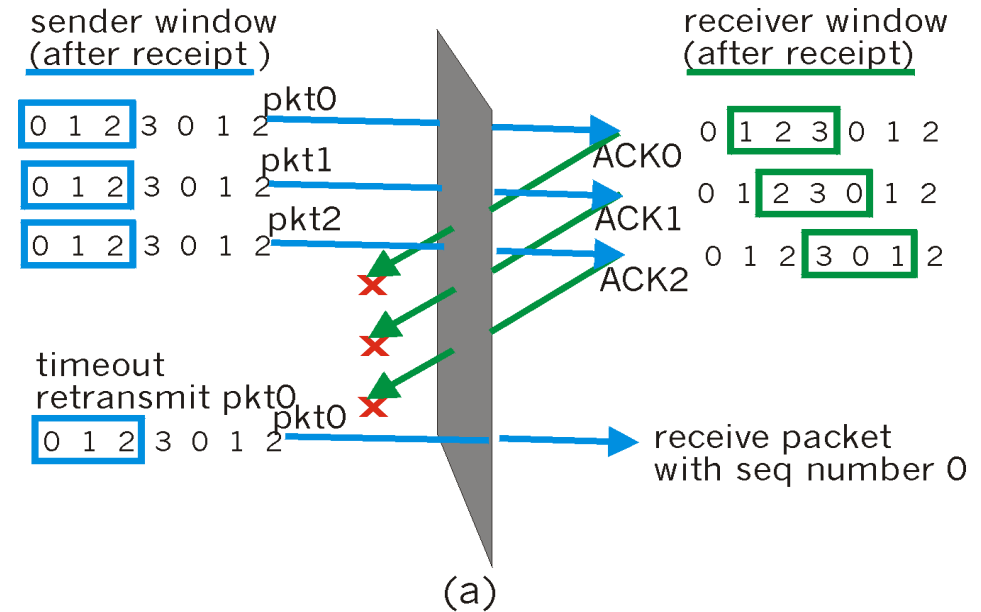
Números de secuencia 0, 1, 2, 3

Tamaño de ventana = 3

El receptor no ve ninguna diferencia en dos escenarios

Incorrectamente pasa datos duplicados como nuevos en (a)

Qué relación deberá haber entre el tamaño de la secuencia de números y el tamaño de la ventana?



# Chapter 3 outline

3.1 Transport-layer services

3.2 Multiplexing and demultiplexing

3.3 Connectionless transport: UDP

3.4 Principles of reliable data transfer

3.5 Connection-oriented transport: TCP

segment structure

reliable data transfer

flow control

connection management

3.6 Principles of congestion control

3.7 TCP congestion control