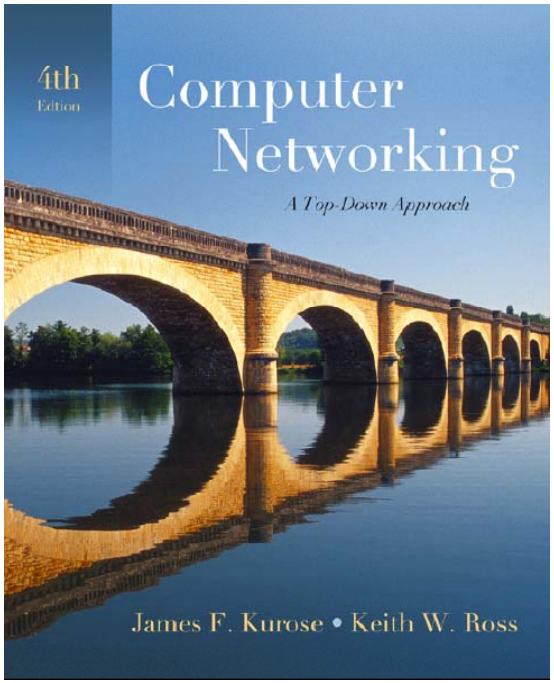
Chapter 3
Transport Layer
傳輸層



Computer Networking: A Top Down Approach 4th edition.
Jim Kurose, Keith Ross
Addison-Wesley, July 2007.

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

Chapter 3: Transport Layer

Our goals:

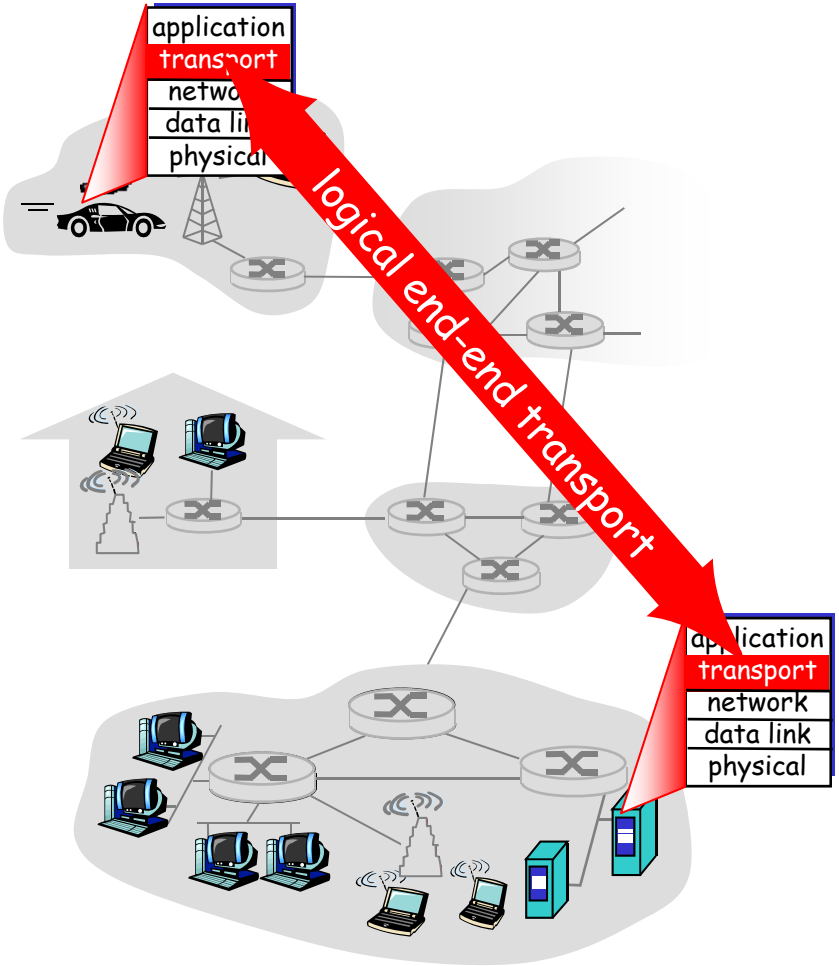
- understand principles behind transport layer services:
  - multiplexing/demultiplexing
    多工與解多工
  - reliable data transfer
    可信賴的資料傳輸
  - flow control
    流量控制
  - congestion control
    壅塞控制
- learn about transport layer protocols in the Internet:
  - UDP: connectionless transport
  - TCP: connection-oriented transport
  - TCP congestion control

Chapter 3 outline

- 3.1 Transport-layer services
  傳輸層服務
- 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

Transport services and protocols

- provide *logical communication* 邏輯通訊 between app processes running on different hosts
- transport protocols run in end systems
  - send side: breaks app messages into segments 區段, passes to network layer
  - rcv side: reassembles segments into messages, passes to app layer
- more than one transport protocol available to apps
  - Internet: TCP and UDP



Transport vs. network layer

傳輸層與網路層之間的關係

- *network layer:* logical communication between hosts
- *transport layer:* logical communication between processes
  - relies on, enhances, network layer services

Household analogy:例子

12 kids sending letters to 12 kids

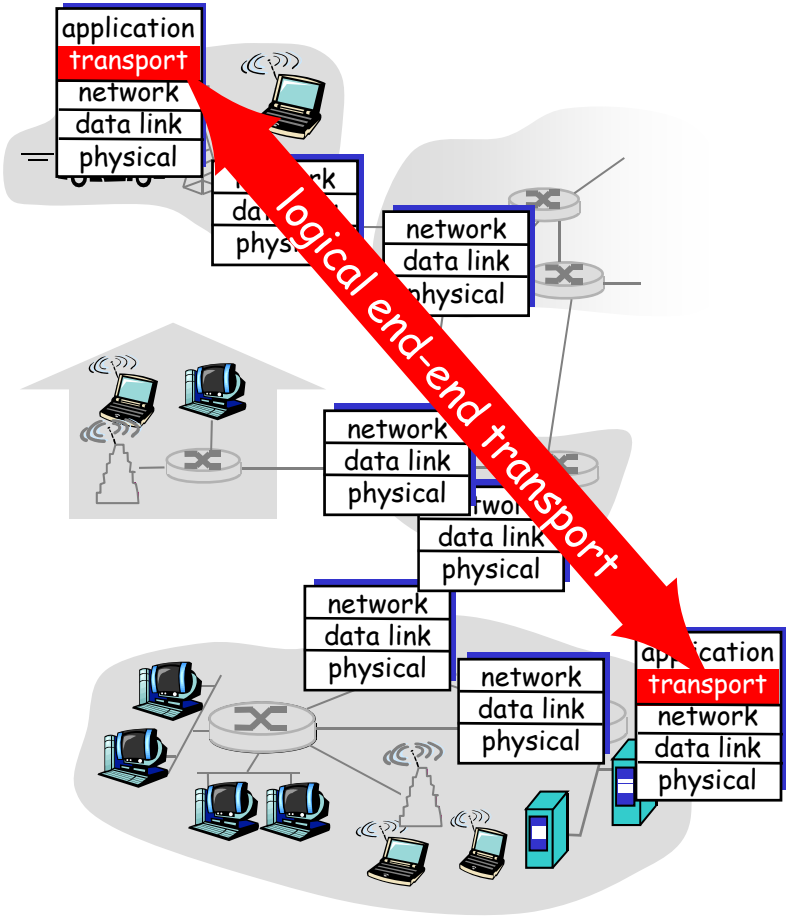
12小孩寄信給12個小孩

- processes = kids
- app messages = letters in envelopes
- hosts = houses
- transport protocol = Ann and Bill
- network-layer protocol = postal service

Internet transport-layer protocols

網際網路傳輸層協定

- reliable, in-order delivery (TCP)
  - congestion control
  - flow control
  - connection setup
- unreliable, unordered delivery: UDP
  - no-frills extension of "best-effort" IP
- services not available:
  - delay guarantees
  - bandwidth guarantees



Chapter 3 outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
  多工與解多工
- 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

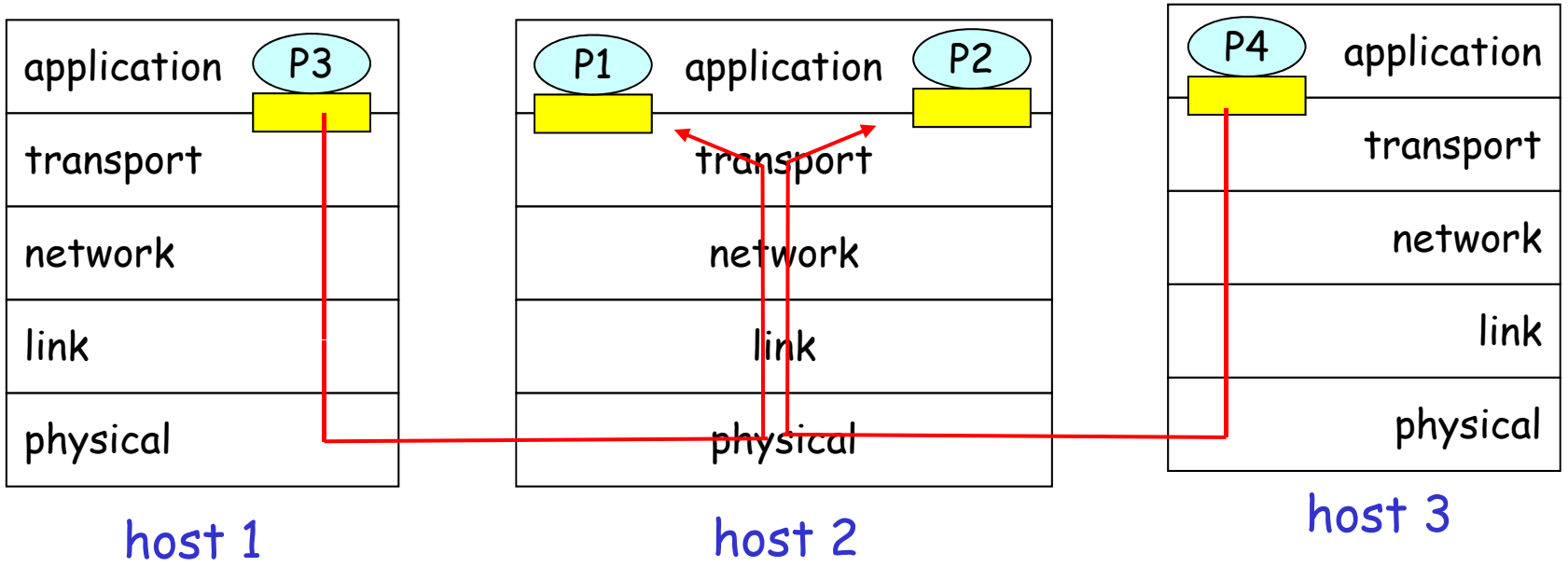
Multiplexing/demultiplexing

多工與解多工

Demultiplexing at rcv host:
delivering received segments to correct socket
在接收端解多工

Multiplexing at send host:
gathering data from multiple sockets, enveloping data with header (later used for demultiplexing)
在傳送端多工

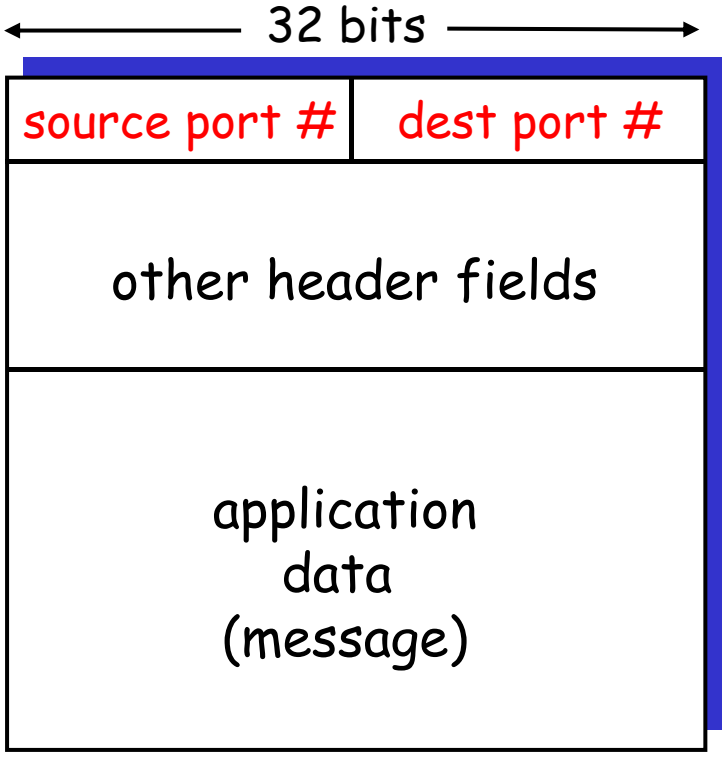
= socket = process



How demultiplexing works

解多工的運作方式

- host receives IP datagrams
  主機收到IP封包後
  - each datagram has source IP address, destination IP address
  - each datagram carries 1 transport-layer segment
    每個封包都攜帶一個傳輸層的區段
  - each segment has source, destination port number
    每個區段都有port number
- host uses IP addresses & port numbers to direct segment to appropriate socket
  用IP Address及port number決定



TCP/UDP segment format

Connectionless demultiplexing
UDP的解多工

- Create sockets with port numbers:

```
DatagramSocket mySocket1 = new
   DatagramSocket(12534);
DatagramSocket mySocket2 = new
   DatagramSocket(12535);
```

- UDP socket identified by two-tuple:

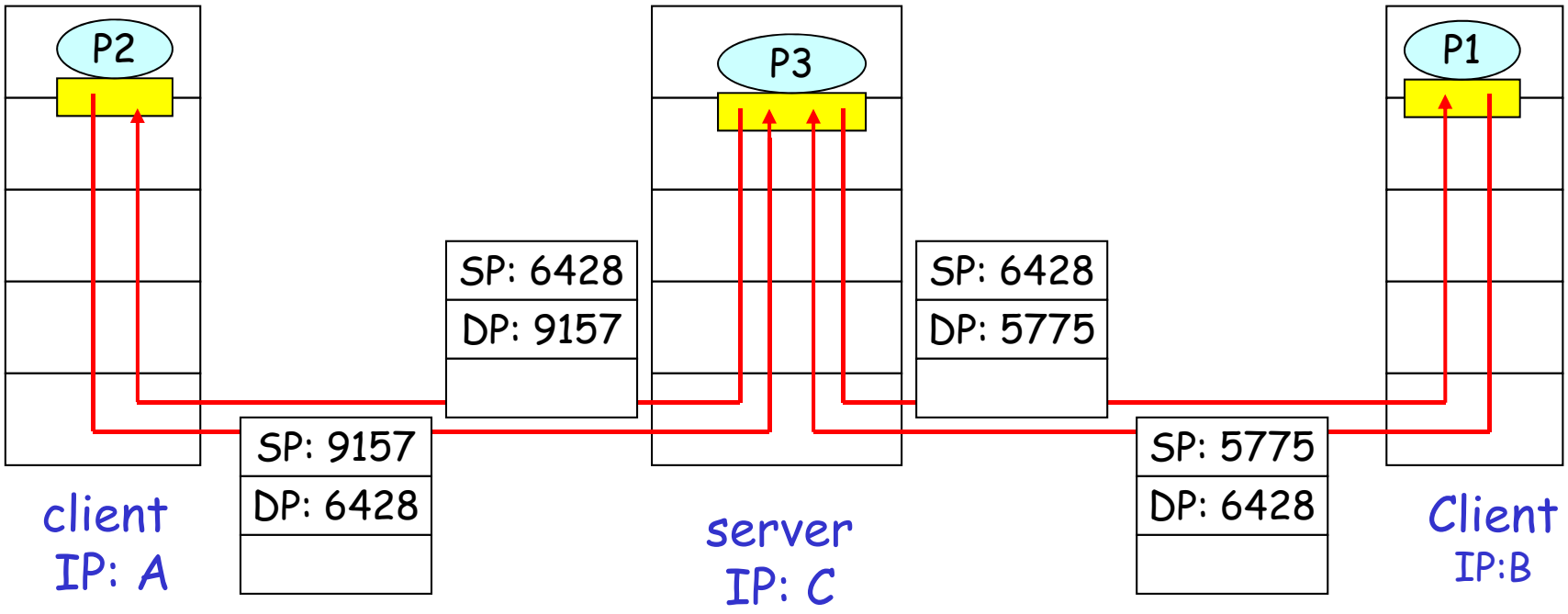
(dest IP address, dest port number)

- When host receives UDP segment:
  - checks destination port number in segment
  - directs UDP segment to socket with that port number
- IP datagrams with different source IP addresses and/or source port numbers directed to same socket

返回位址

Connectionless demux (cont)

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



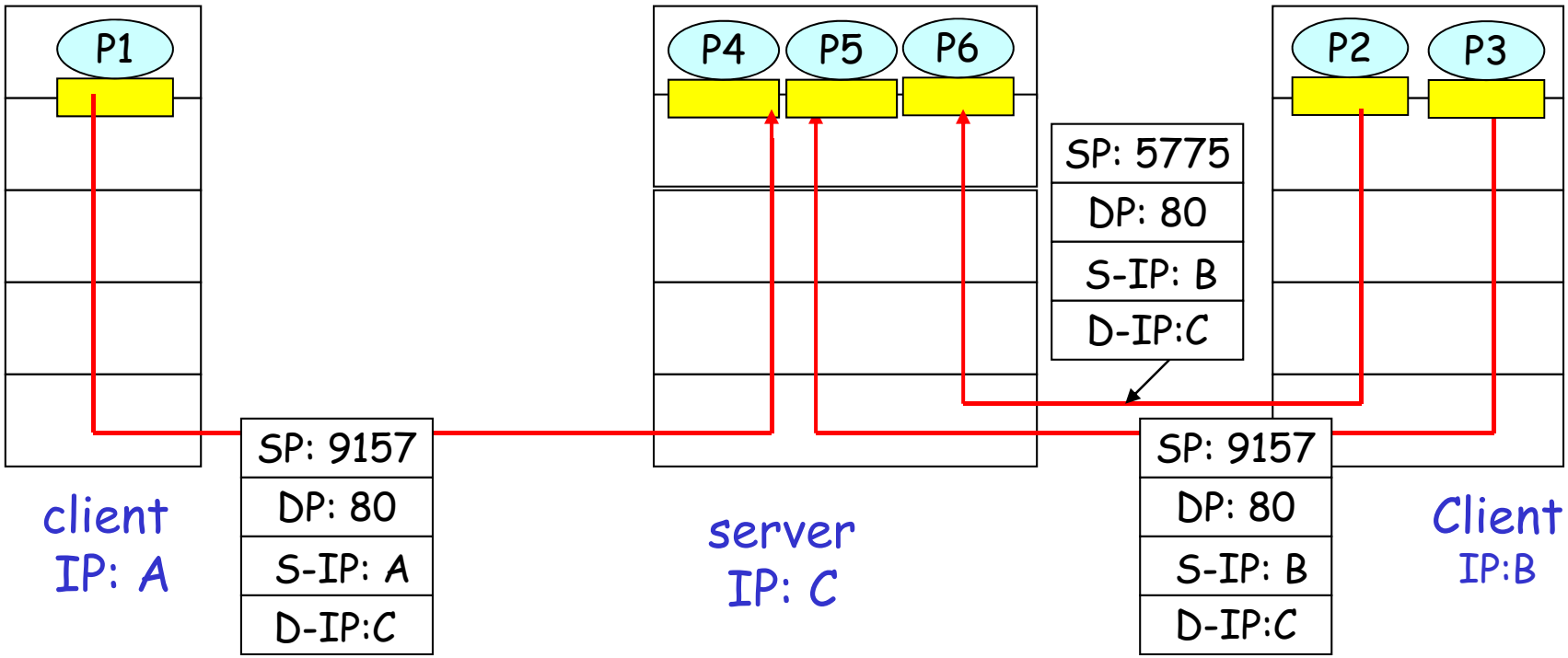
SP provides "return address"

Connection-oriented demux
TCP的解多工

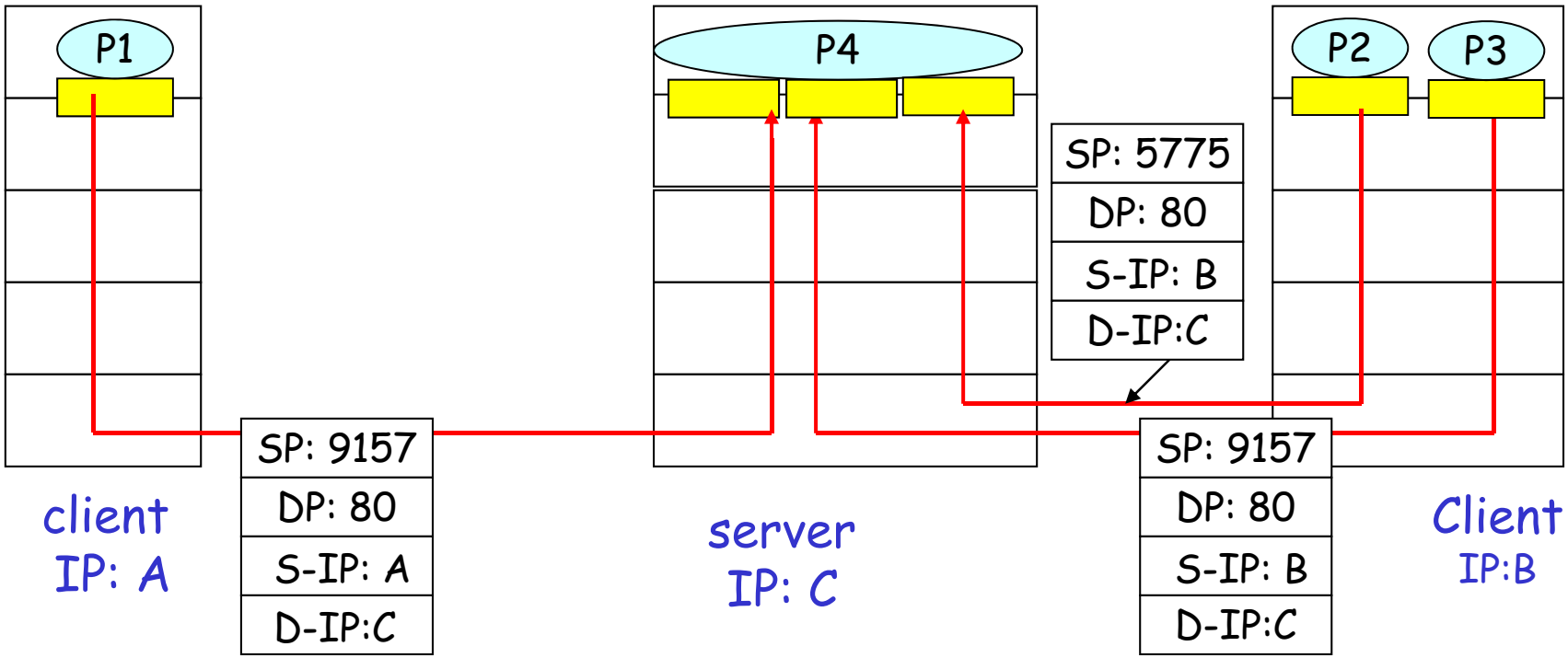
- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number

    因為需要事先建立連線
- recv host uses all four values to direct segment to appropriate socket
- Server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- Web servers have different sockets for each connecting client
  - non-persistent HTTP will have different socket for each request

Connection-oriented demux (cont)



Connection-oriented demux: Threaded Web Server



Chapter 3 outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
  無連結傳輸：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

UDP: User Datagram Protocol [RFC 768]

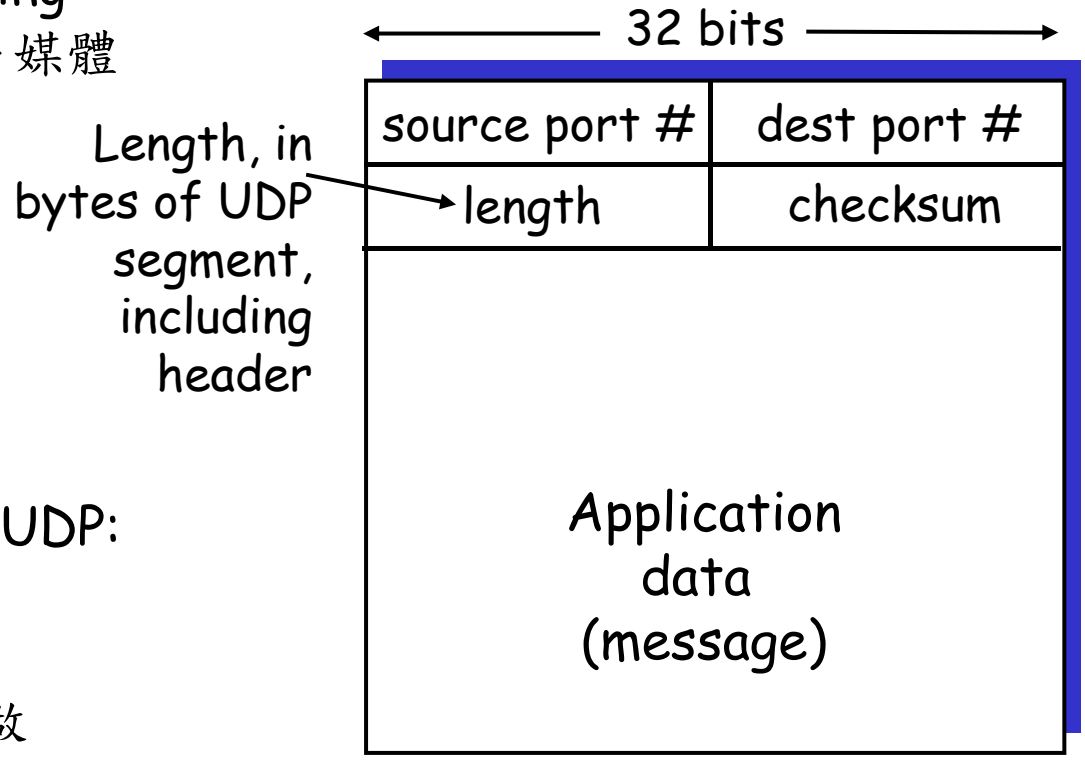
- "no frills," "bare bones" Internet transport protocol
- "best effort 盡力去做" service, UDP segments may be:
  - Lost 封包可能遺失
  - delivered out of order to app 封包可能不依順序
- *connectionless:*
  - no handshaking between UDP sender, receiver 沒有握手機制
  - each UDP segment handled independently of others 各自獨立

Why is there a UDP?

- no connection establishment (which can add delay) 減少delay
- simple: no connection state at sender, receiver
- small segment header 標頭負擔較小
- no congestion control: UDP can blast away as fast as desired 沒有壅塞控制的機制

UDP: more

- often used for streaming multimedia apps 串流多媒體
  - loss tolerant
  - rate sensitive
- other UDP uses
  - DNS
  - SNMP
- reliable transfer over UDP: add reliability at application layer
  未達成的服務由應用層做
  - application-specific error recovery!



UDP segment format

UDP checksum 錯誤偵測

Goal: detect "errors" (e.g., flipped bits) in transmitted segment 偵測錯誤

Sender:

- treat segment contents as sequence of 16-bit integers
- checksum: addition (1's complement sum) of segment contents
- sender puts checksum value into UDP checksum field

Receiver:

- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - NO - error detected
  - YES - no error detected. *But maybe errors nonetheless?* More later ....

身份證號碼驗證 check-sum

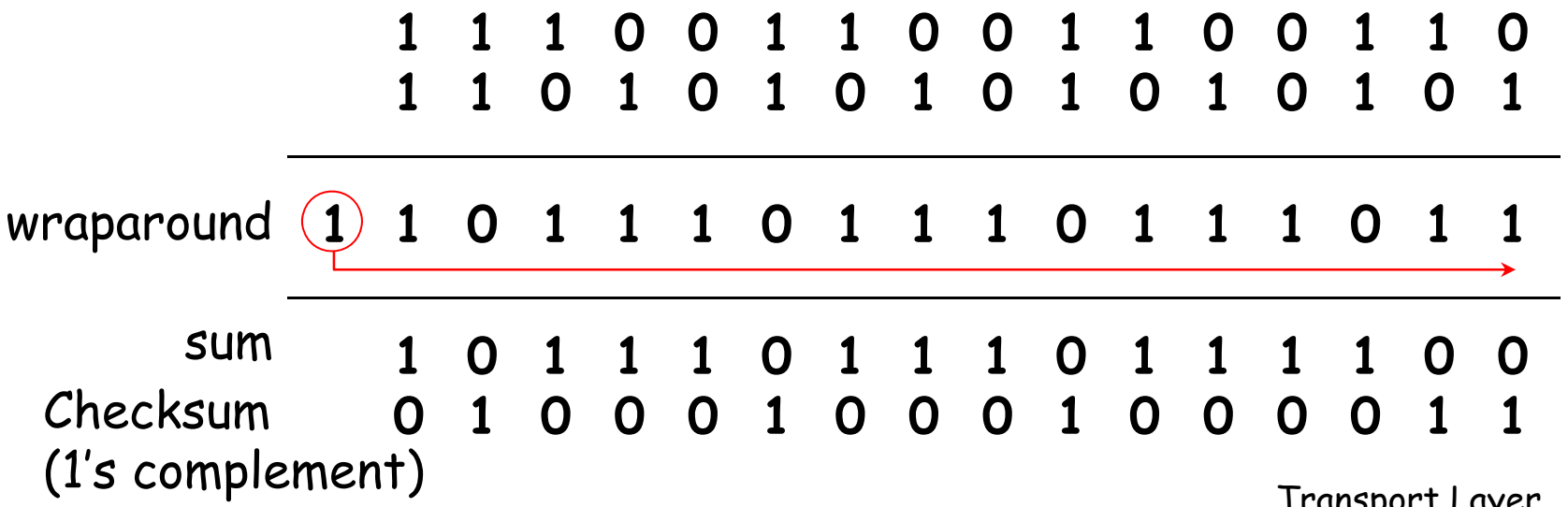
1. 英文代號以下表轉換成數字
   A=10 台北市　J=18 新竹縣 S=26 高雄縣
   B=11 台中市　K=19 苗栗縣 T=27 屏東縣
   C=12 基隆市　L=20 台中縣 U=28 花蓮縣
   D=13 台南市　M=21 南投縣 V=29 台東縣
   E=14 高雄市　N=22 彰化縣 * W=32 金門縣
   F=15 台北縣　O=35 新竹市 X=30 澎湖縣
   G=16 宜蘭縣　P=23 雲林縣 Y=31 陽明山
   H=17 桃園縣　Q=24 嘉義縣 * Z=33 連江縣
   * I=34 嘉義市 R=25 台南縣
2. 英文轉成的數字, 個位數乘９再加上十位數
3. 各數字從右到左依次乘１、２、３、４・・・・８
4. 求出(2),(3)之和
5. 求出(4)除10後之餘數,用10減該餘數,結果就是檢查碼，若餘數為0，檢查碼就是 0。

6. http://web.htps.tn.edu.tw/cen/other/files/pp/index.htm

Internet Checksum Example

網際網路驗證碼

- Note
  - When adding numbers, a carryout from the most significant bit needs to be added to the result
- Example: add two 16-bit integers

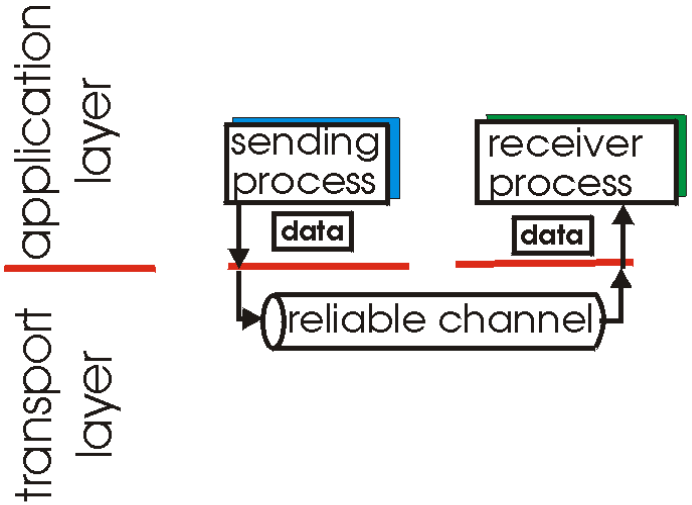


Chapter 3 outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
  可靠資料傳輸原理
- 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

Principles of Reliable data transfer

- important in app., transport, link layers 應用層、傳輸層、連結層
- top-10 list of important networking topics!

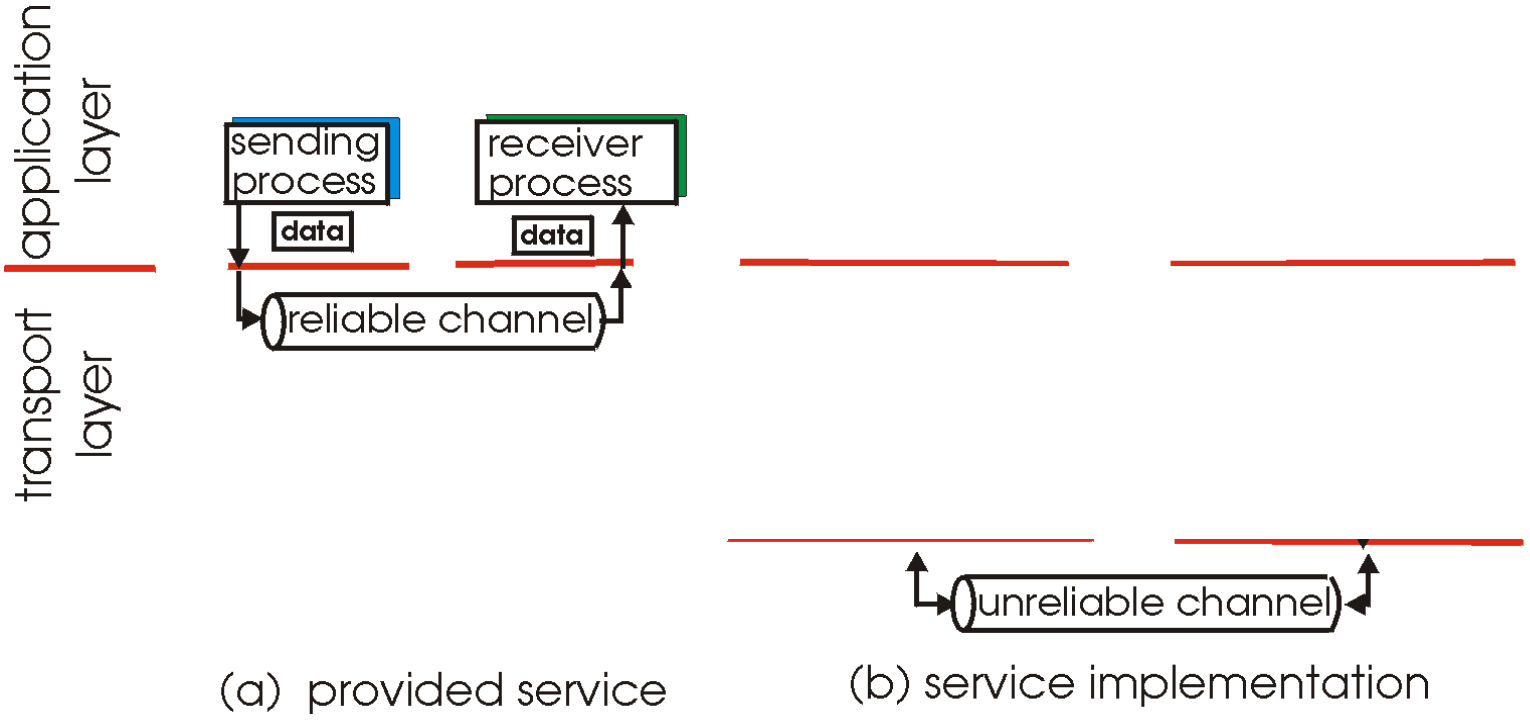


(a) provided service

- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Principles of Reliable data transfer

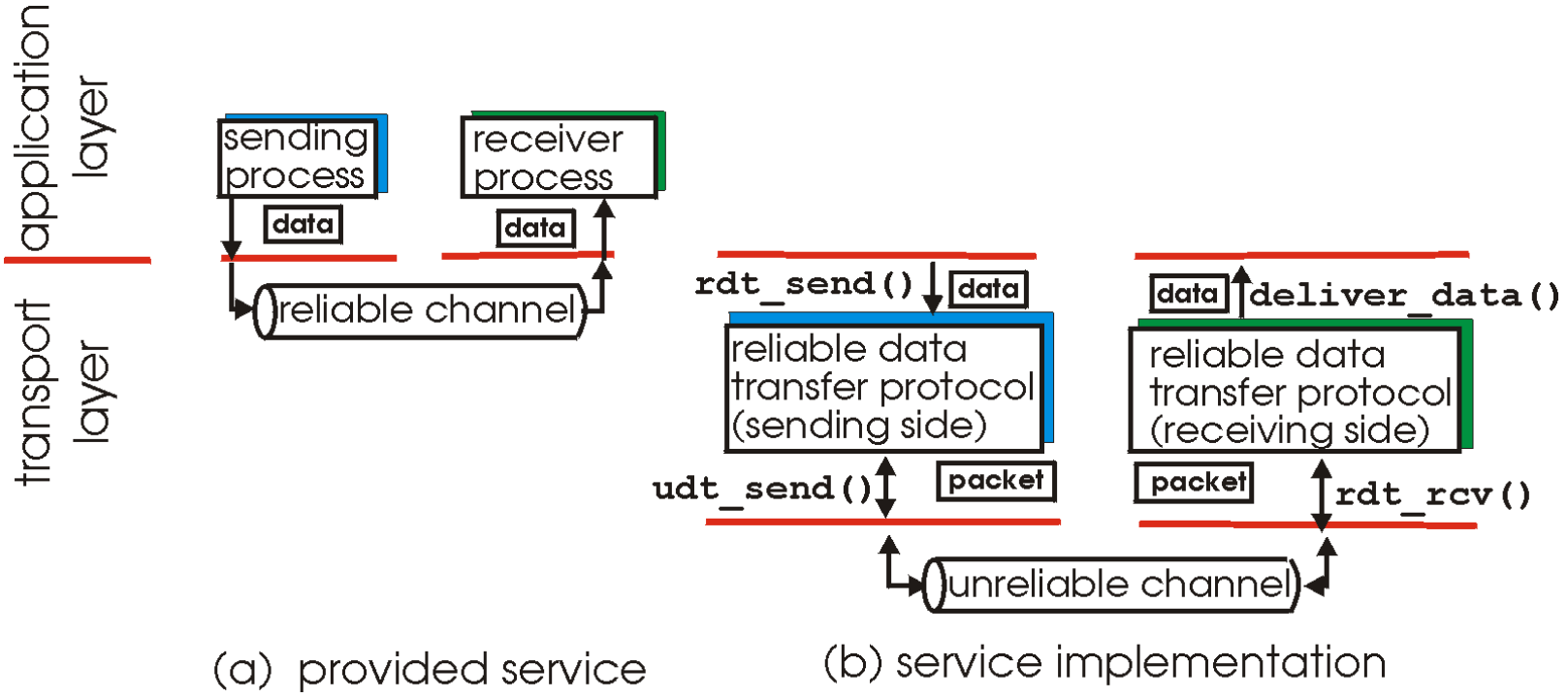
- important in app., transport, link layers
- top-10 list of important networking topics!



- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Principles of Reliable data transfer

- important in app., transport, link layers
- top-10 list of important networking topics!

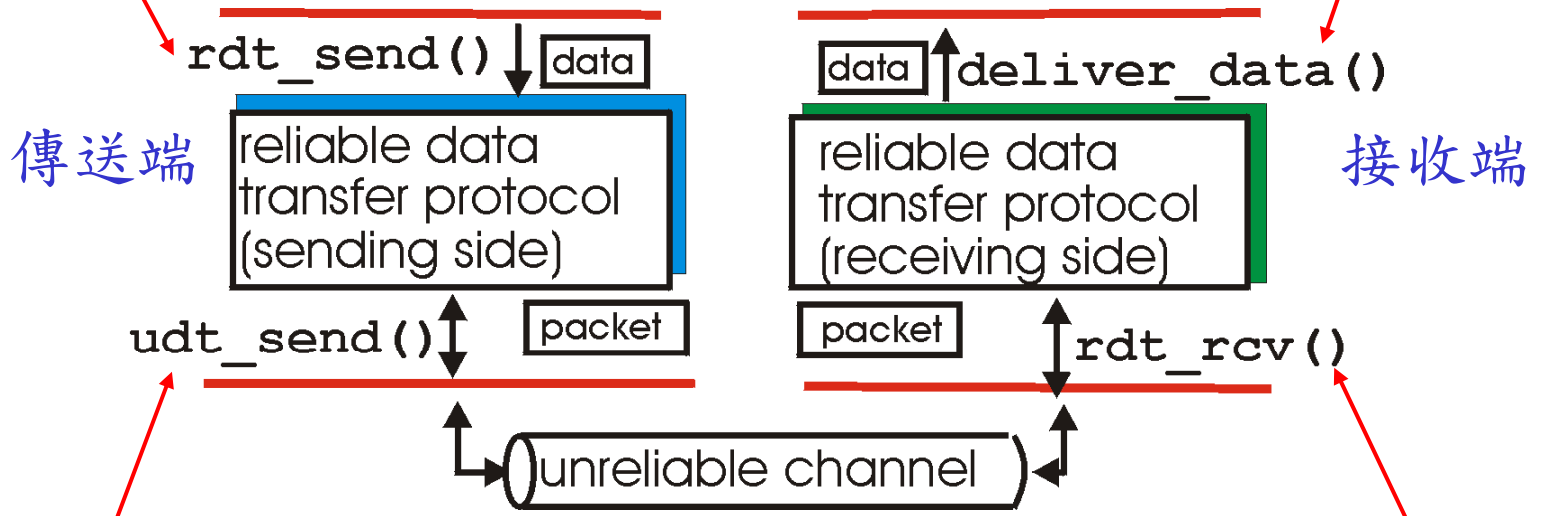


- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Reliable data transfer: getting started

rdt_send(): 由上層應用程式呼叫，可以把資料傳送給接收端的上層

deliver_data(): 將資料傳送到上層



udt_send(): 由rdt呼叫，可以將封包傳送給另一端

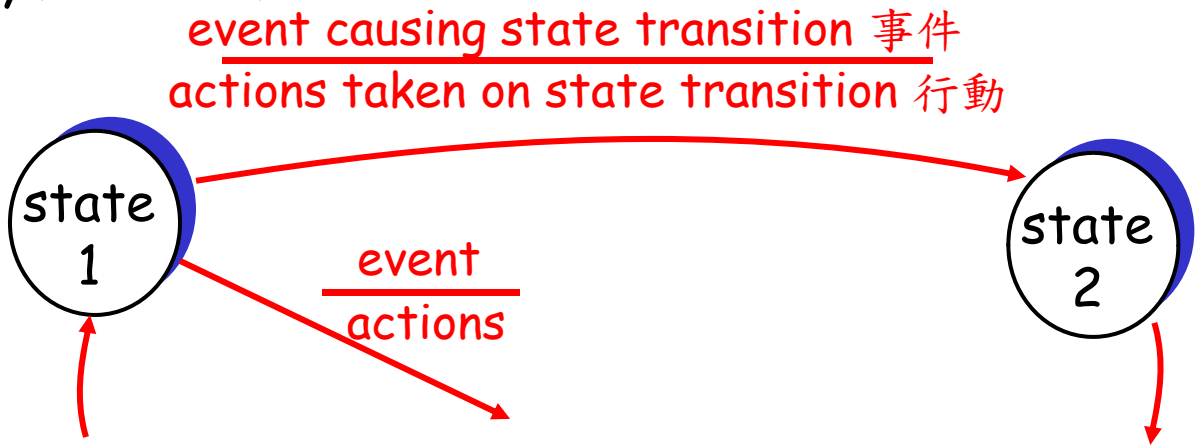
rdt_rcv(): 當封包抵達時，呼叫此函式

Reliable data transfer: getting started

We'll:

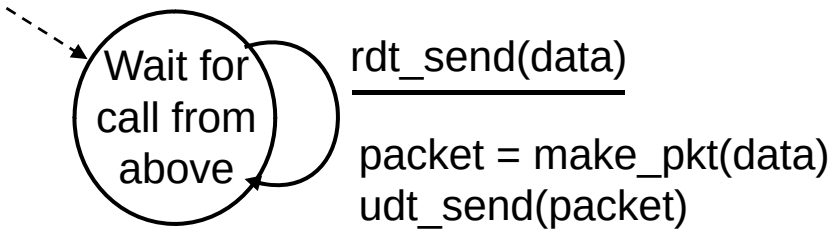
- incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- consider only unidirectional data transfer
  - but control info will flow on both directions!
- use finite state machines (FSM有限狀態機) to specify sender, receiver

state: when in this "state" next state uniquely determined by next event

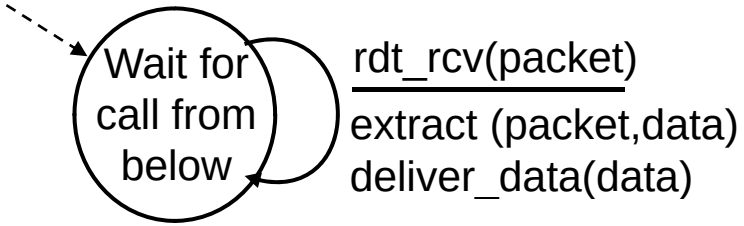


Rdt1.0: reliable transfer over a reliable channel

- underlying channel perfectly reliable
  完全可信賴的底層通道
  - no bit errors 資料不會錯誤
  - no loss of packets 封包不會遺失
- separate FSMs for sender, receiver:
  - sender sends data into underlying channel
  - receiver read data from underlying channel



Sender
傳送端

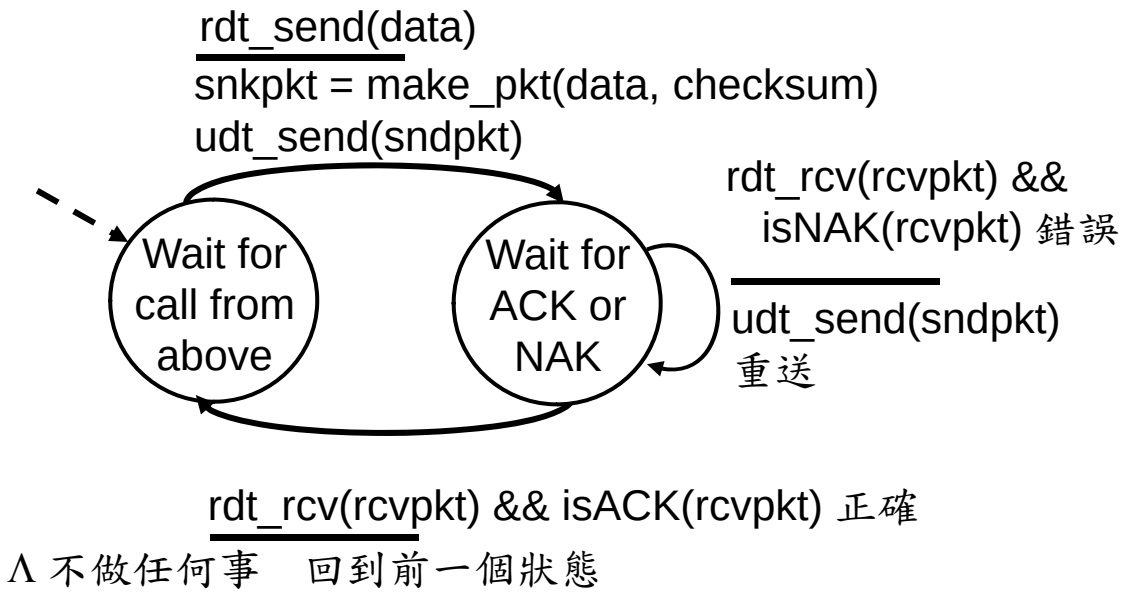


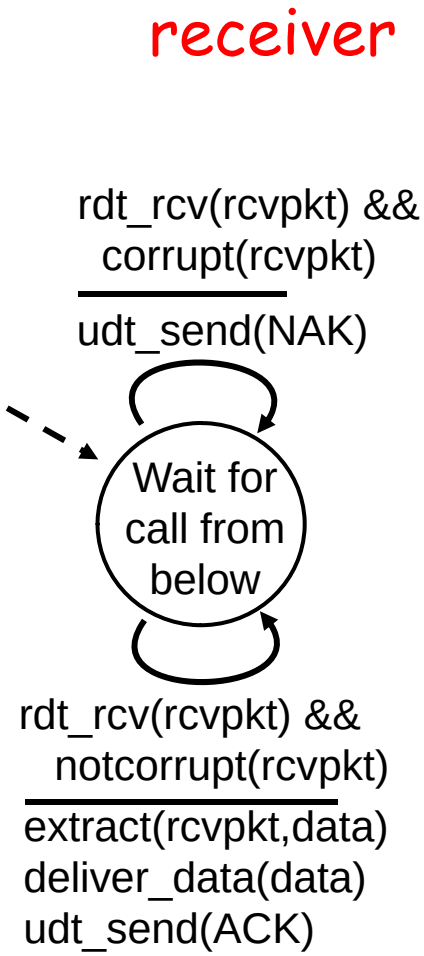
Receiver
接收端

Rdt2.0: channel with bit errors 會有位元錯誤

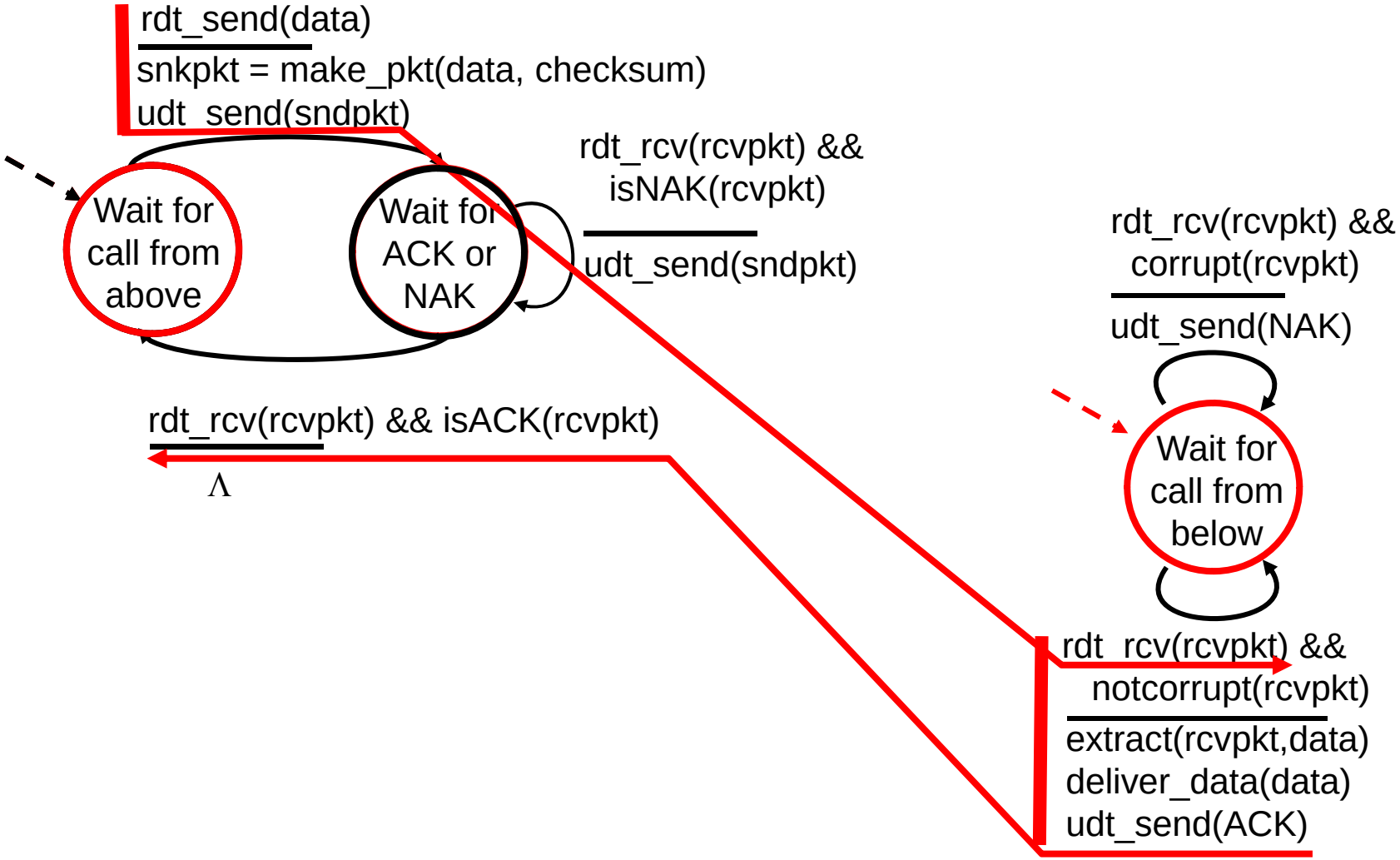
- underlying channel may flip bits in packet
  - checksum to detect bit errors 用驗證碼偵測錯誤
- *the* question: how to recover from errors:
  - *acknowledgements (ACKs):* receiver explicitly tells sender that pkt received OK 使用【正確】訊息
  - *negative acknowledgements (NAKs):* receiver explicitly tells sender that pkt had errors 使用【錯誤】訊息
  - sender retransmits pkt on receipt of NAK
    當沒收到【正確】或收到【錯誤】時重送
- new mechanisms in **`rdt2.0`** (beyond **`rdt1.0`**):
  - error detection
  - receiver feedback: control msgs (ACK,NAK) rcvr->sender

rdt2.0: FSM specification

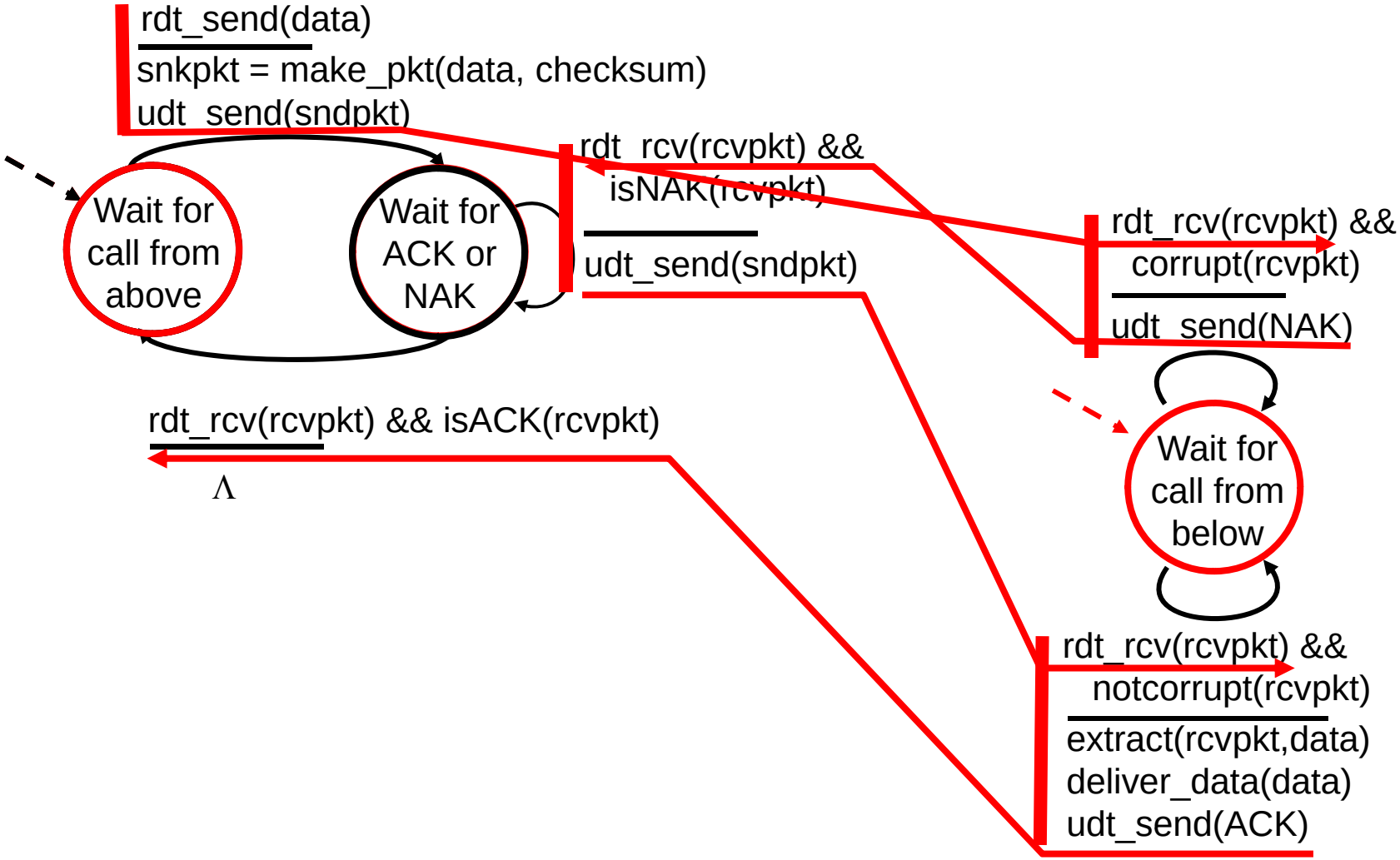




rdt2.0: operation with no errors



rdt2.0: error scenario



rdt2.0 has a fatal flaw! 致命問題

What happens if ACK/NAK corrupted?
如果連ACK/NAK也錯了

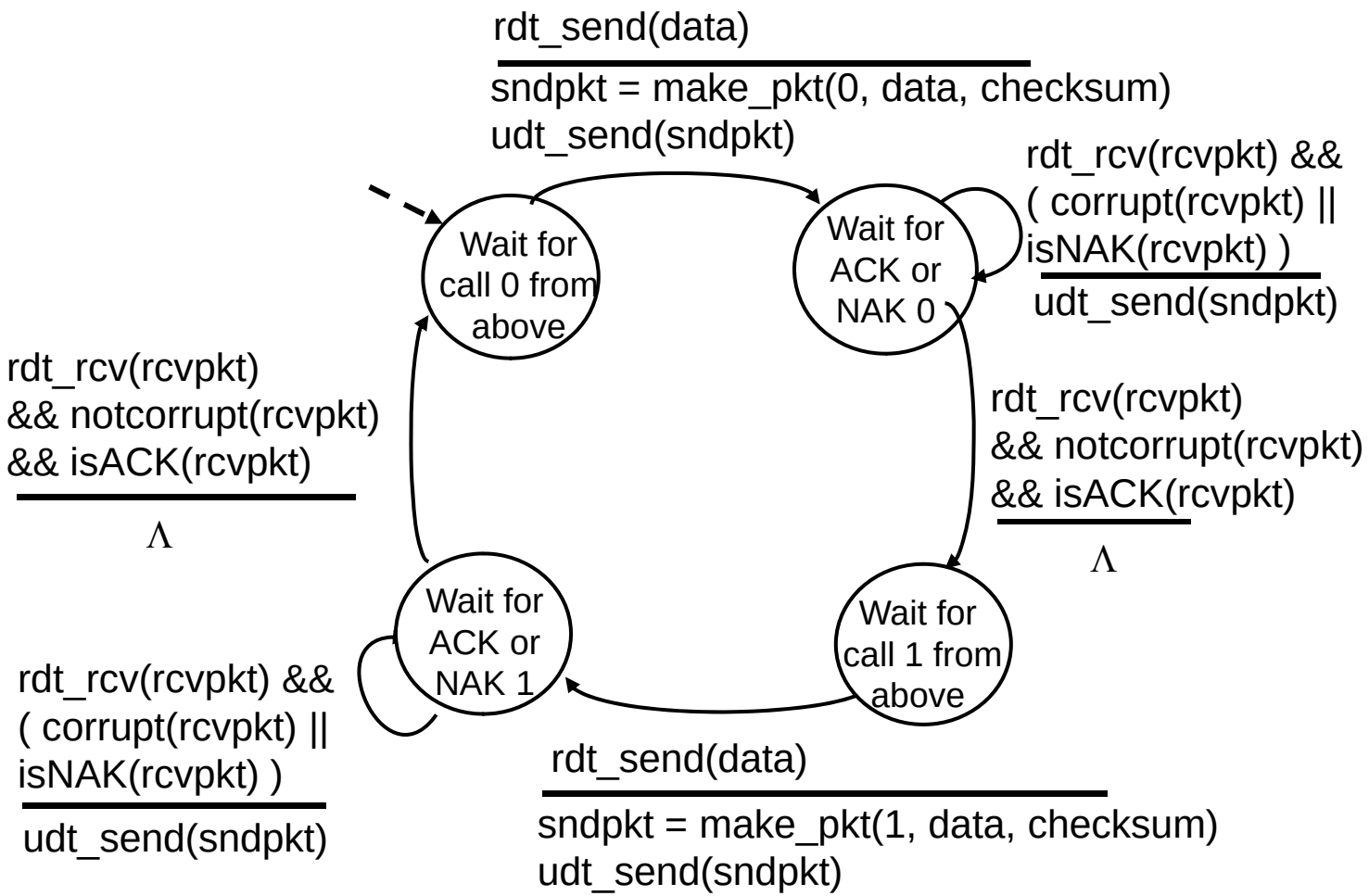
- sender doesn't know what happened at receiver!
  傳送端不知道接收端的狀態
- can't just retransmit: possible duplicate
  不能重送：可能發生重覆傳送的問題

Handling duplicates:

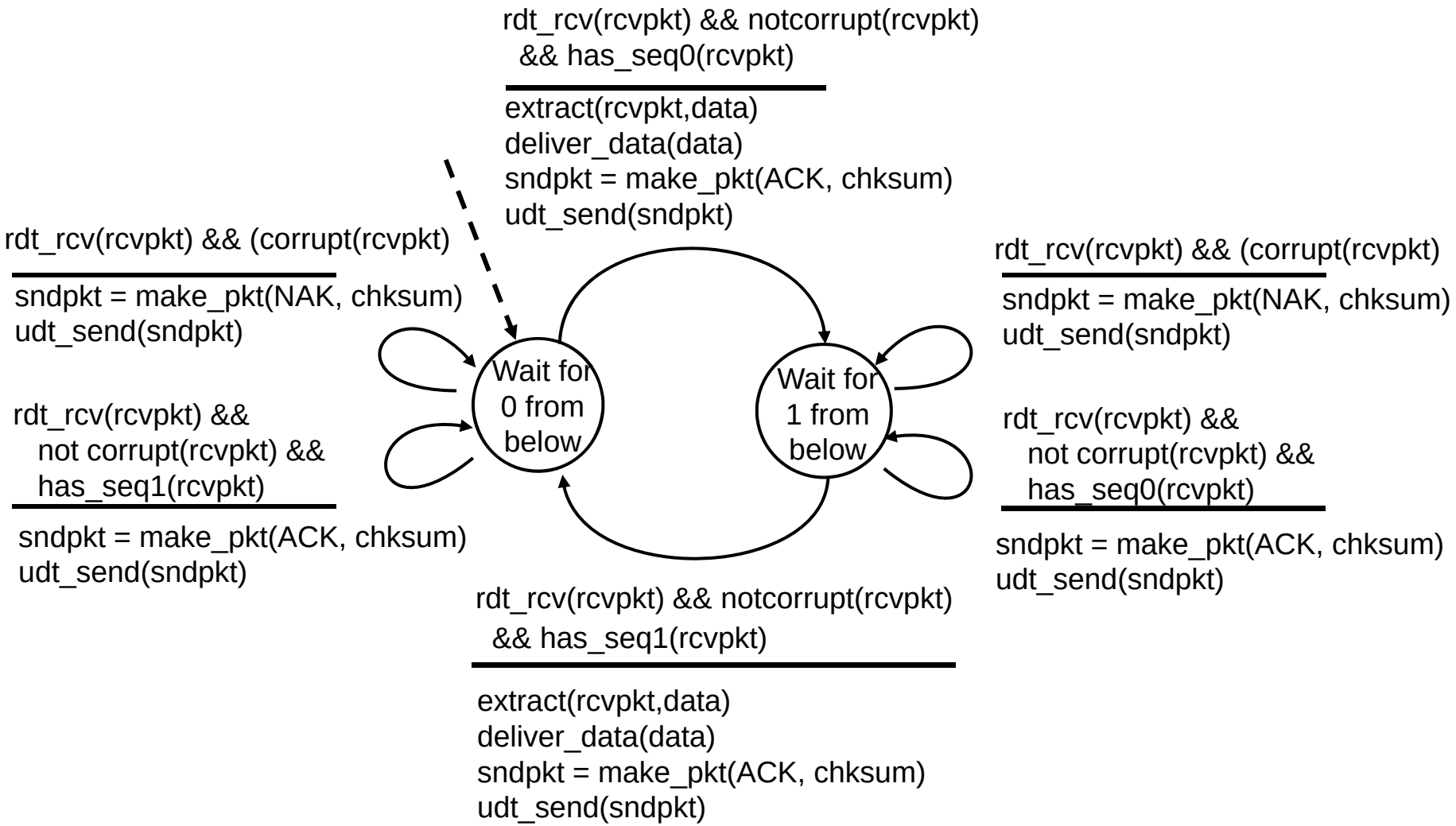
- sender retransmits current pkt if ACK/NAK garbled
- sender adds *sequence number* to each pkt
  增加序號
- receiver discards (doesn't deliver up) duplicate pkt
  丟棄相同序號的封包

stop and wait
Sender sends one packet, then waits for receiver Response
停止與等待協定

rdt2.1: sender, handles garbled ACK/NAKs



rdt2.1: receiver, handles garbled ACK/NAKs



rdt2.1: discussion

Sender:

- seq # added to pkt
- two seq. #'s (0,1) will suffice. Why?
  為什麼用兩個序號(0,1)就夠了
- must check if received ACK/NAK corrupted
- twice as many states
  - state must "remember" whether "current" pkt has 0 or 1 seq. #

Receiver:

- must check if received packet is duplicate
  - state indicates whether 0 or 1 is expected pkt seq #
- note: receiver can *not* know if its last ACK/NAK received OK at sender

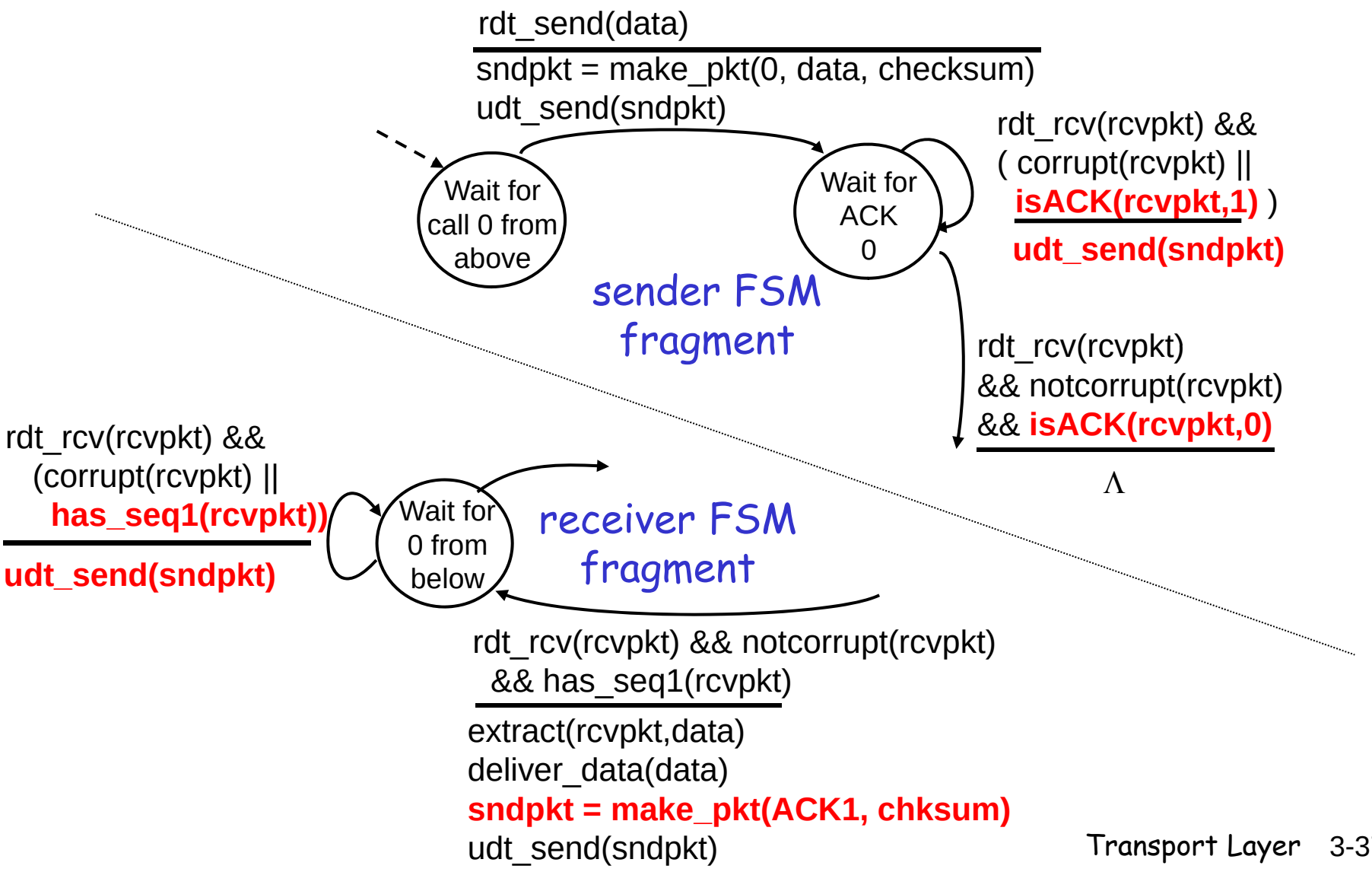
rdt2.2: a NAK-free protocol
不用NAK的協定

- same functionality as rdt2.1, using ACKs only
  只用ACK，不用NAK
- instead of NAK, receiver sends ACK for last pkt received OK
  - receiver must *explicitly* include seq # of pkt being ACKed

  告訴傳送端，最後一個成功接收的封包序號
- duplicate ACK at sender results in same action as NAK: *retransmit current pkt*

  重覆收到相同的ACK，表示有封包錯誤，因此重新傳送封包

rdt2.2: sender, receiver fragments



rdt3.0: channels with errors *and* loss

位元錯誤及封包遺失

New assumption:

underlying channel can also lose packets (data or ACKs)

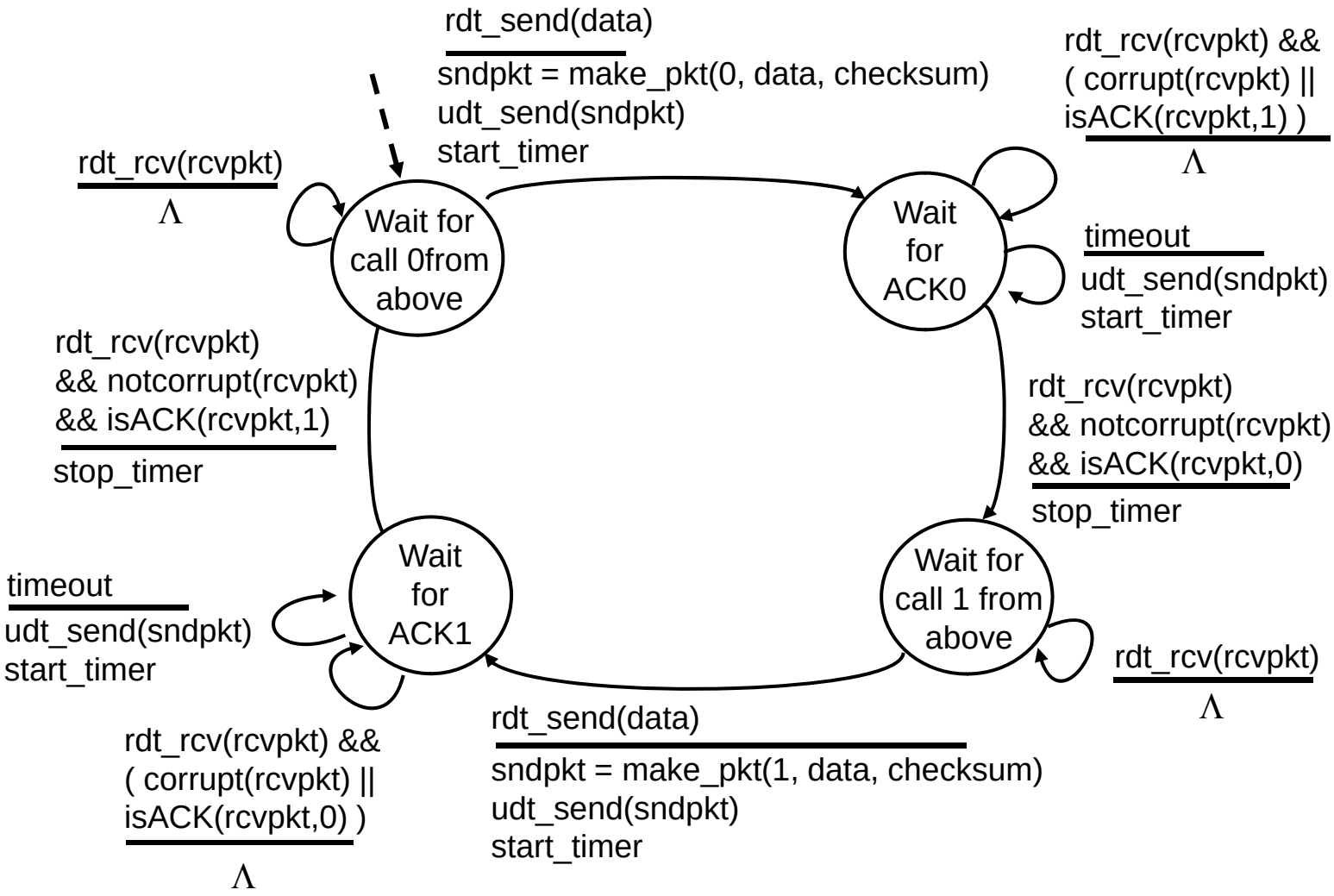
- checksum, seq. #, ACKs, retransmissions will be of help, but not enough

Approach: sender waits "reasonable" amount of time for ACK (timeout)

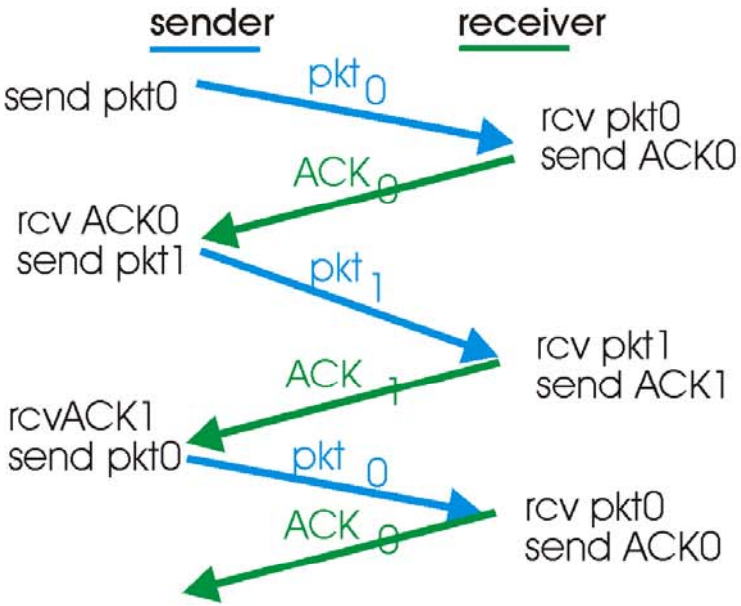
等待一段"合理的"時間

- retransmits if no ACK received in this time
- if pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate, but use of seq. #'s already handles this
  - receiver must specify seq # of pkt being ACKed
- requires countdown timer

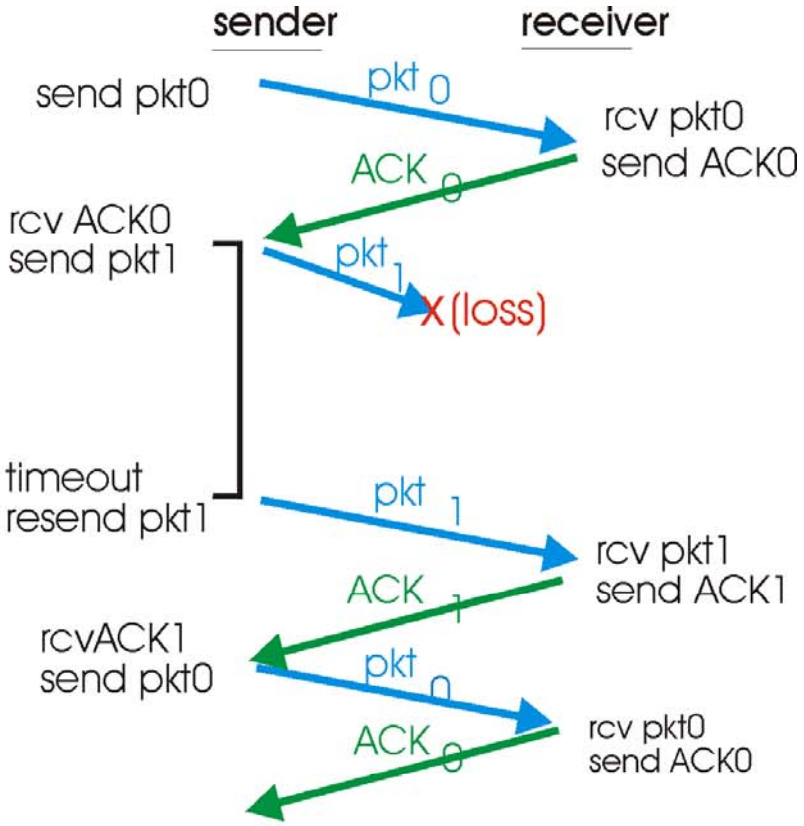
rdt3.0 sender



rdt3.0 in action

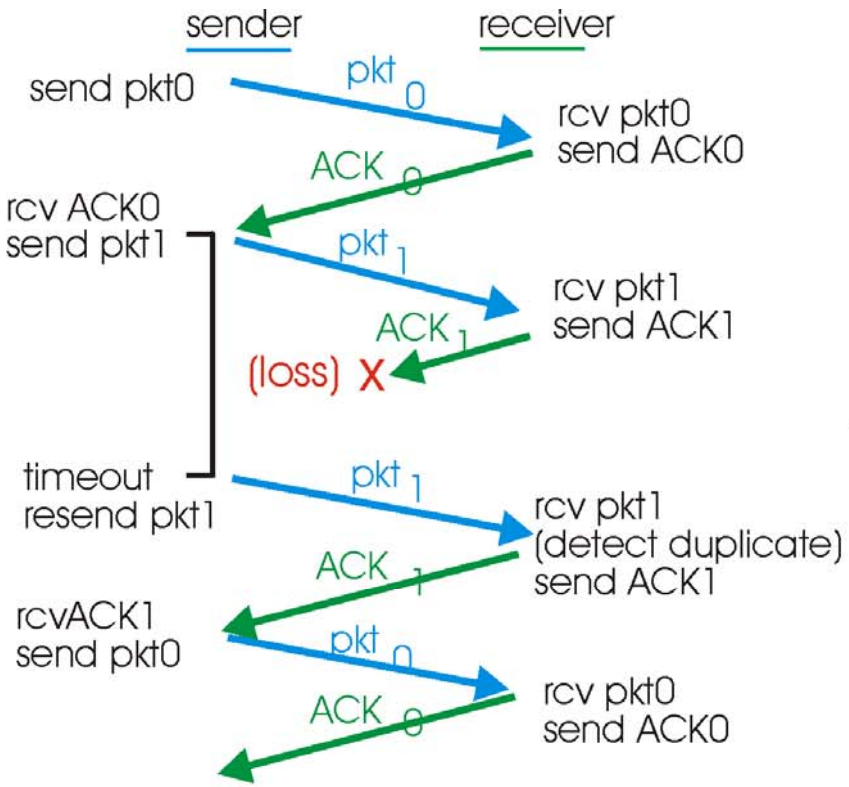


(a) operation with no loss

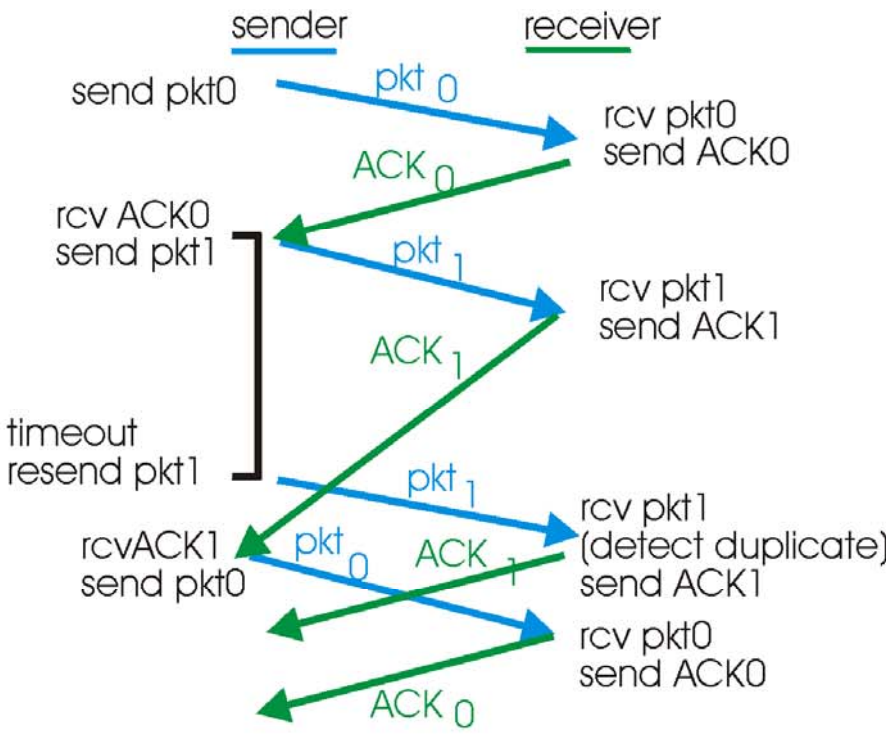


(b) lost packet

rdt3.0 in action



(c) lost ACK



(d) premature timeout

Performance of rdt3.0

- rdt3.0 works, but performance stinks
- example: 1 Gbps link, 15 ms e-e prop. delay, 1KB packet:

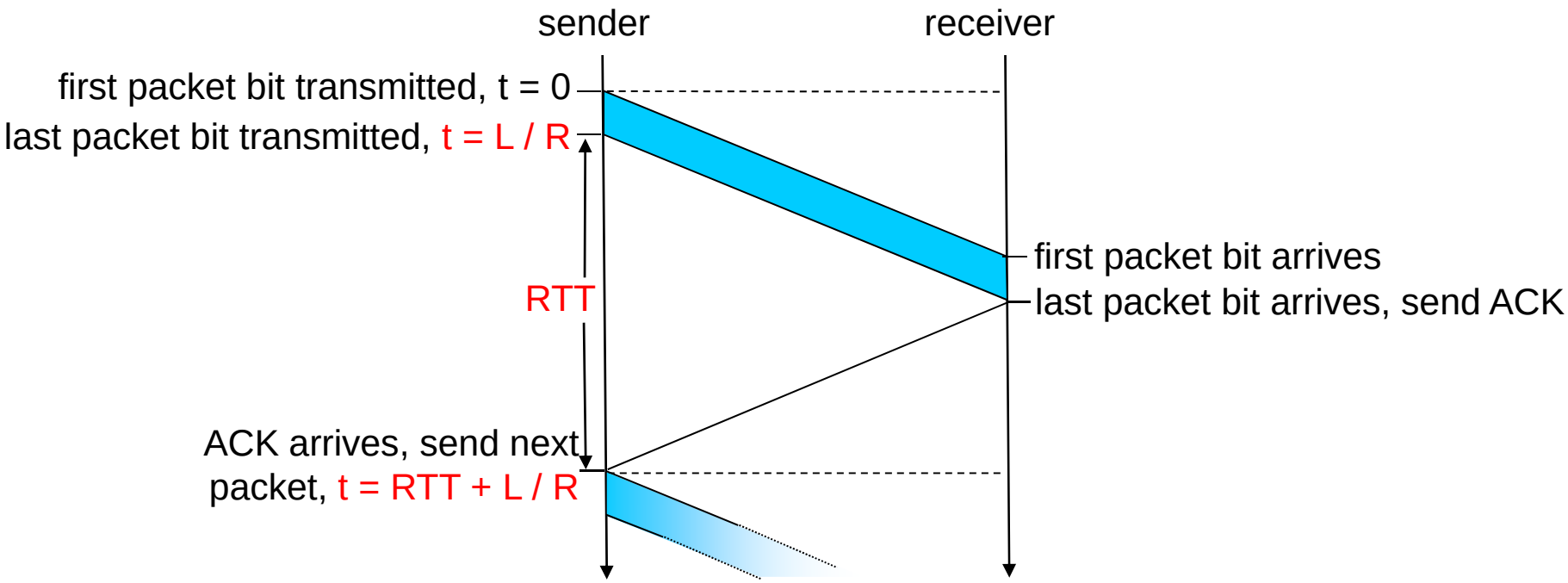
$$T_{transmit} = \frac{L \text{ (packet length in bits)}}{R \text{ (transmission rate, bps)}} = \frac{8kb/pkt}{10**9 \text{ b/sec}} = 8 \text{ microsec}$$

  - U_{sender}: utilization – fraction of time sender busy sending

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

  - 1KB pkt every 30 msec -> 33kB/sec thruput over 1 Gbps link
  - network protocol limits use of physical resources!

rdt3.0: stop-and-wait operation

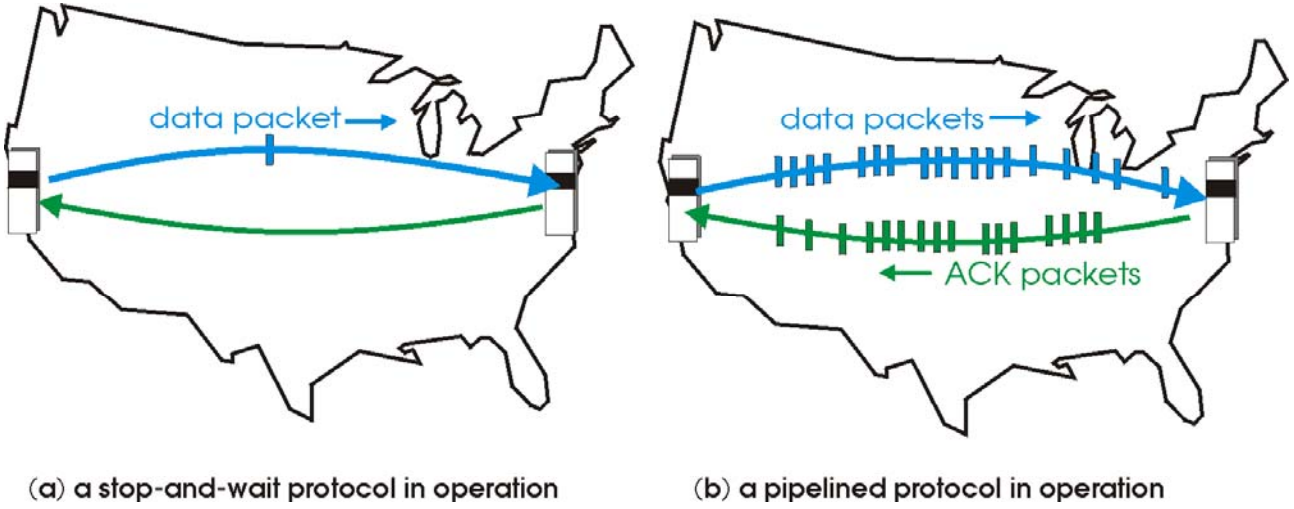


$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

Pipelined protocols (平行)管線傳送

Pipelining: sender allows multiple, "in-flight", yet-to-be-acknowledged pkts

- range of sequence numbers must be increased
- buffering at sender and/or receiver

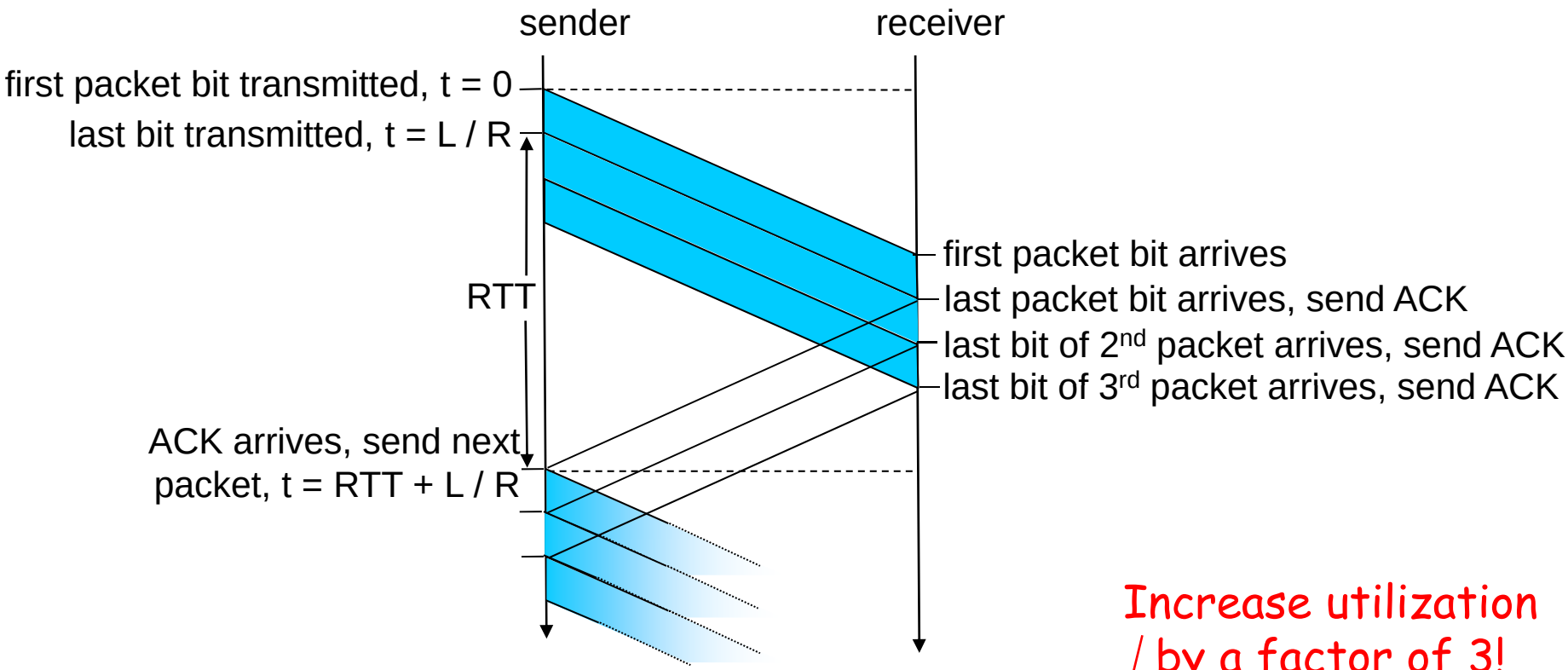


(a) a stop-and-wait protocol in operation

(b) a pipelined protocol in operation

- Two generic forms of pipelined protocols: *go-Back-N, selective repeat*

Pipelining: increased utilization



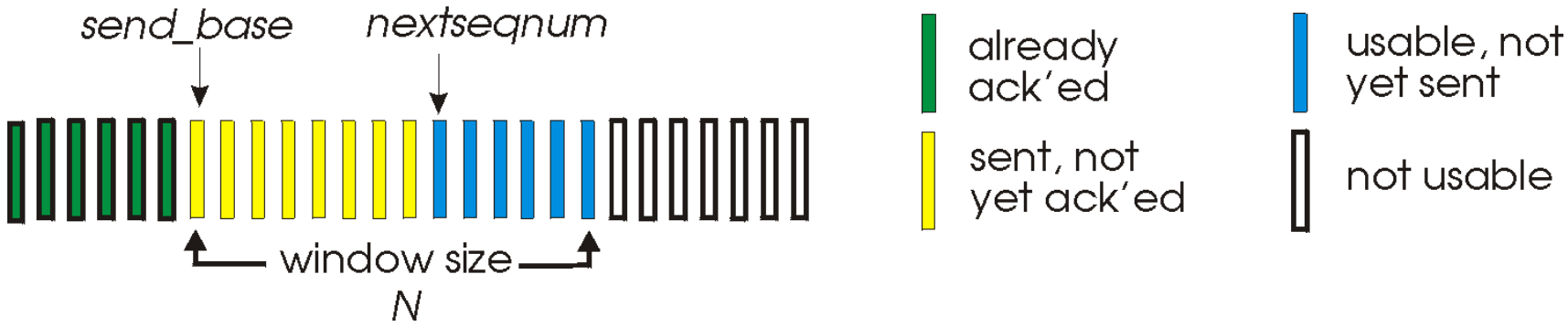
Increase utilization by a factor of 3!

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

Go-Back-N

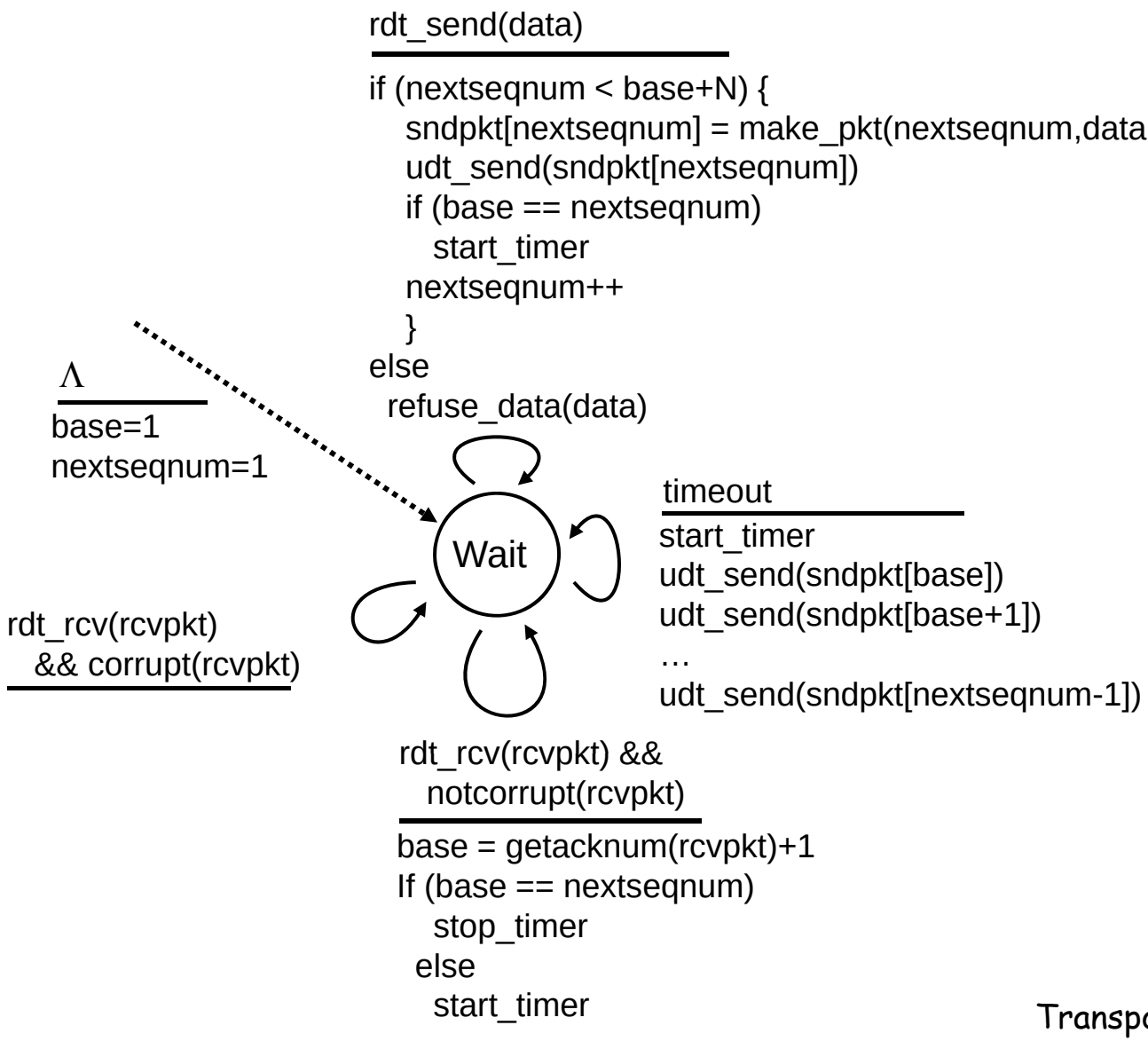
Sender:

- k-bit seq # in pkt header (用k-bit來表示序號)
- "window" of up to N, consecutive unack'ed pkts allowed

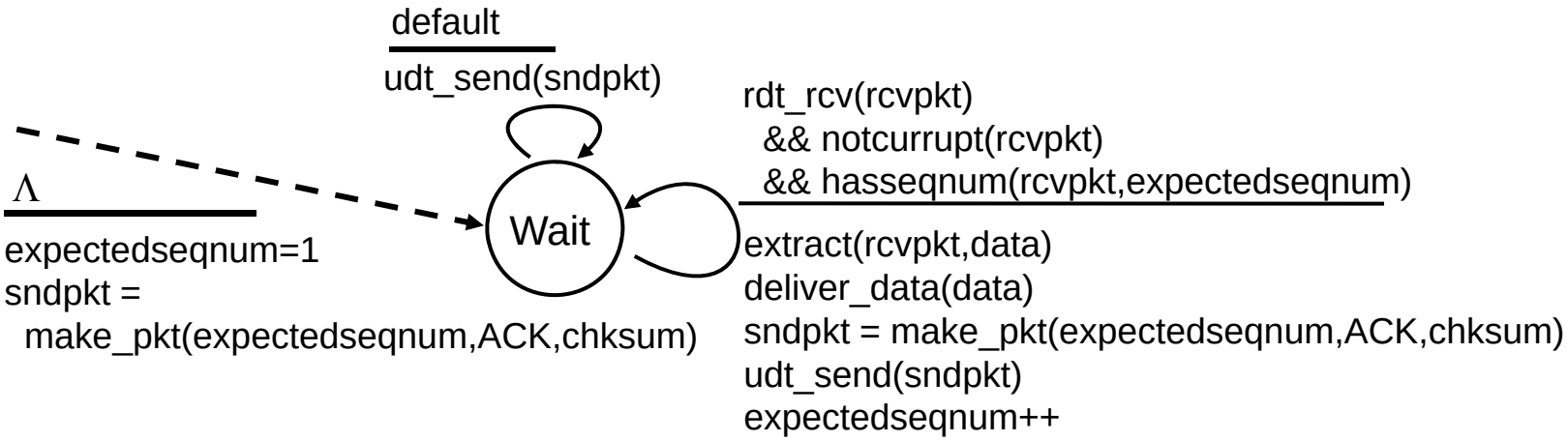


- ACK(n): ACKs all pkts up to, including seq # n - "cumulative ACK"
  - may receive duplicate ACKs (see receiver)
- timer for in-flight pkt 設重傳時間
- *timeout(n):* retransmit pkt n and all higher seq # pkts in window
  重傳沒收到ACK的封包，以及seq#較大的封包

GBN: sender extended FSM



GBN: receiver extended FSM

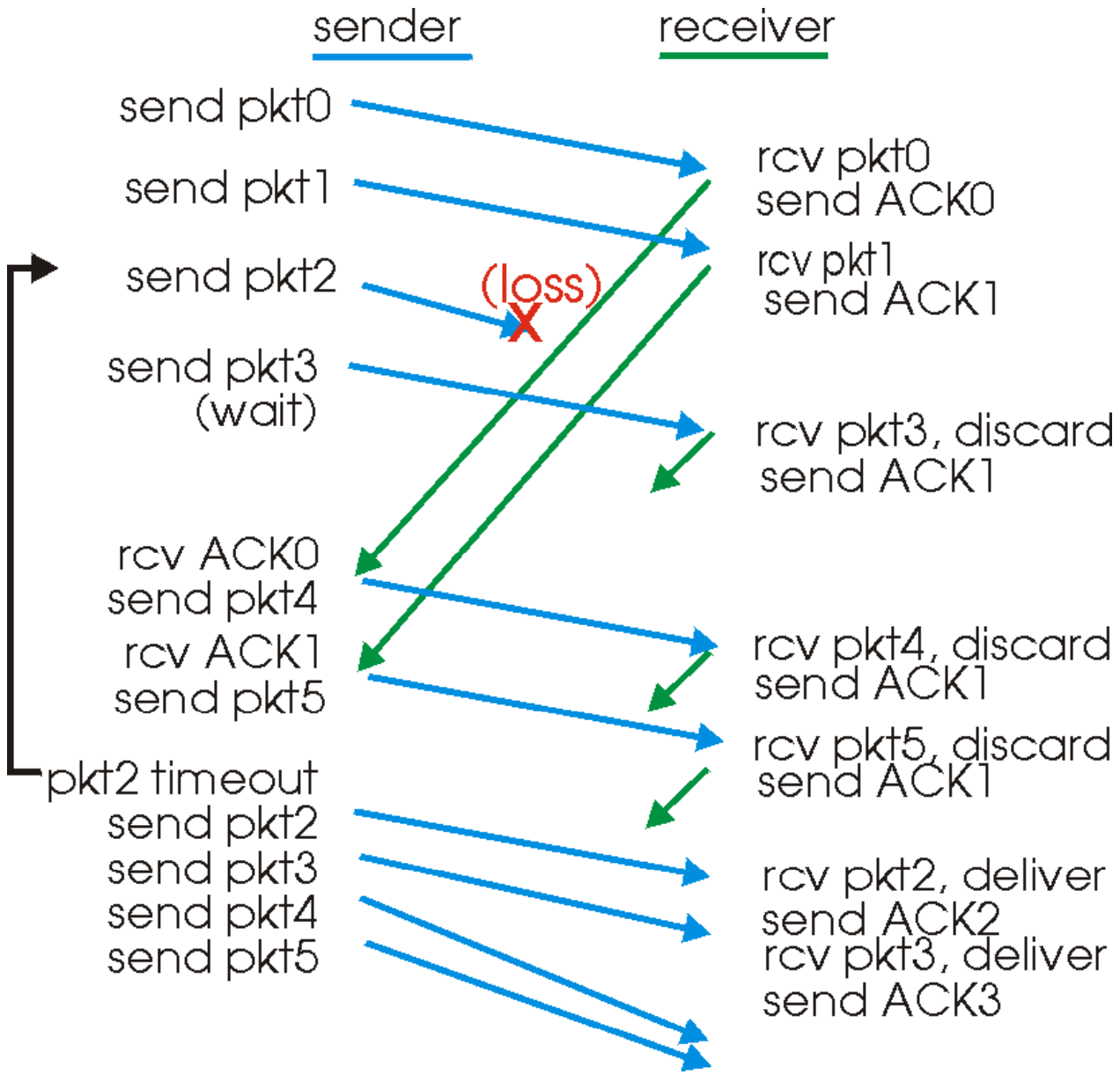


ACK-only: always send ACK for correctly-received pkt with highest *in-order* seq # (ACK最大且依序到達的封包)

- may generate duplicate ACKs
- need only remember **expectedseqnum**
  只要記得下一個要收的封包序號即可

- out-of-order pkt: 不照順序抵達的封包處理方法
  - discard (don't buffer) -> no receiver buffering! 捨棄
  - Re-ACK pkt with highest in-order seq #

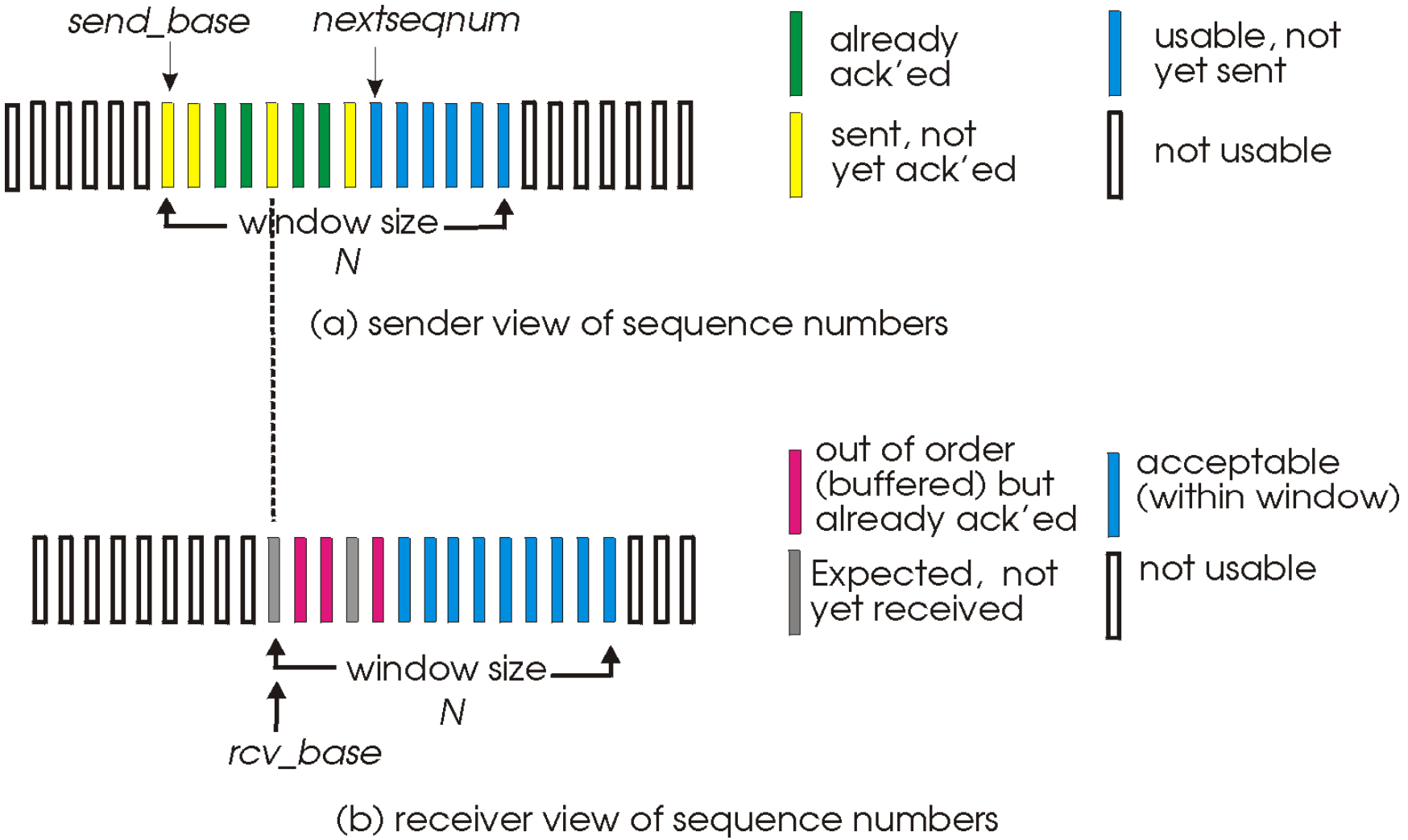
GBN in action



Selective Repeat

- receiver *individually* acknowledges all correctly received pkts 各自ACK正確收到的封包
  - buffers pkts, as needed, for eventual in-order delivery to upper layer 有buffer可用
- sender only resends pkts for which ACK not received 只要送沒有ACK的封包
  - sender timer for each unACKed pkt
- sender window
  - N consecutive seq #'s
  - again limits seq #s of sent, unACKed pkts

Selective repeat: sender, receiver windows



Selective repeat

sender

data from above :

- if next available seq # in window, send pkt
  如果下一個可送的序號在window內，就送出

timeout(n):

- resend pkt n, restart timer
  重新傳送packet, 重設計時器

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

- mark pkt n as received
- if n smallest unACKed pkt, advance window base to next unACKed seq #

receiver

pkt n in [rcvbase, rcvbase+N-1]

- send ACK(n)
- out-of-order: buffer
- in-order: deliver (also deliver buffered, in-order pkts), advance window to next not-yet-received pkt
  移動window

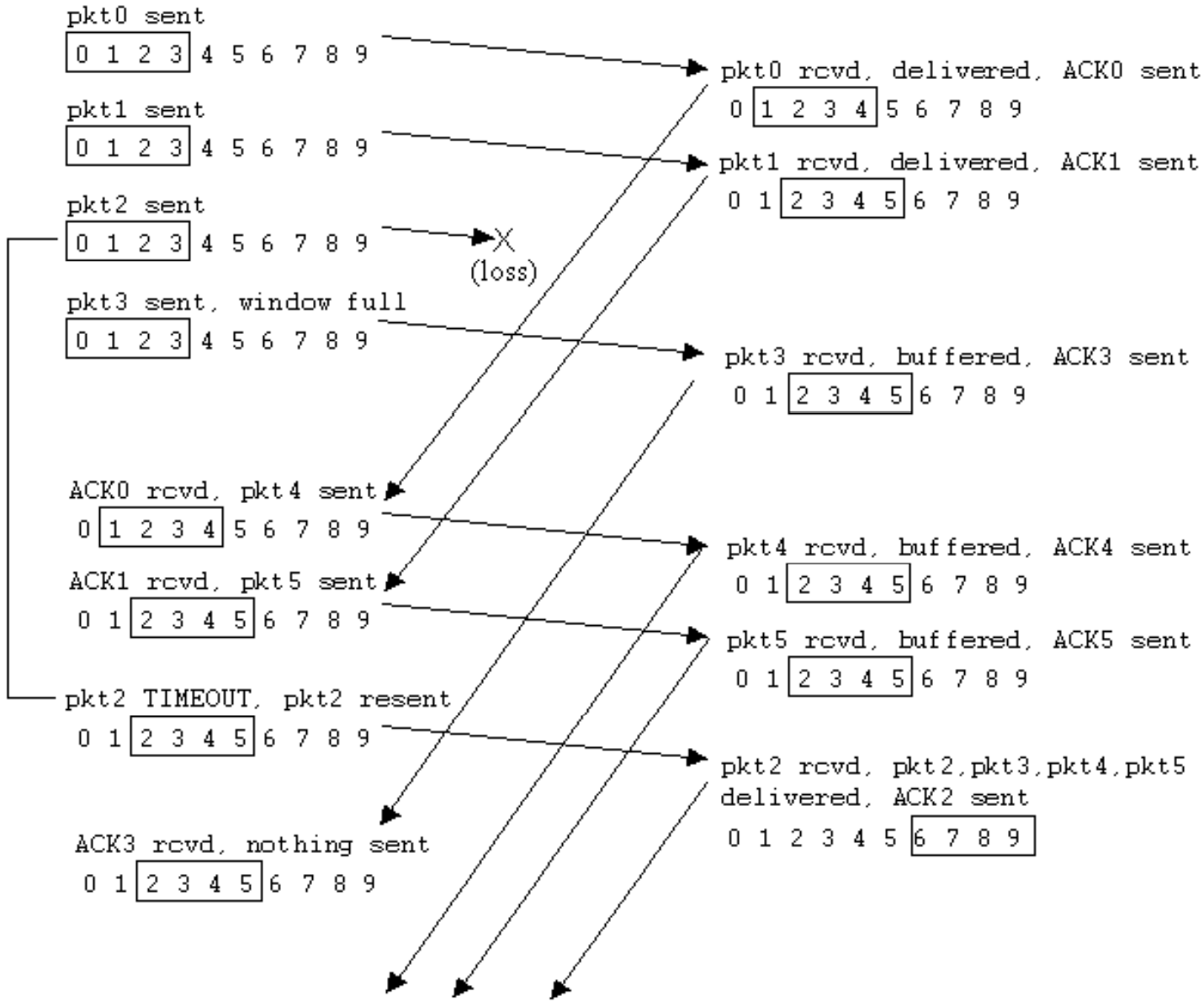
pkt n in [rcvbase-N,rcvbase-1]

- ACK(n)

otherwise:

- ignore

Selective repeat in action



Selective repeat: dilemma 問題來了

- Example:
- seq #'s: 0, 1, 2, 3
- window size=3

- receiver sees no difference in two scenarios!
- incorrectly passes duplicate data as new in (a)

- Q: what relationship between seq # size and window size? (Homework)