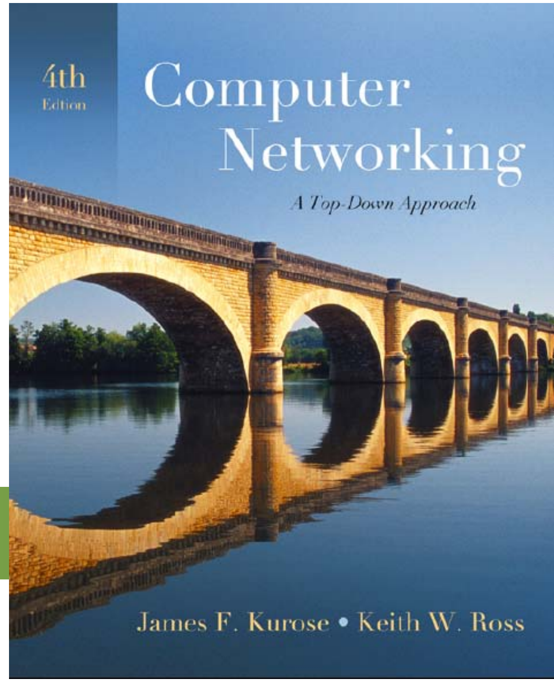
電腦網路

Computer Networks

http://www.cyut.edu.tw/~hcchu

Chapter 3: Transport Layer



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

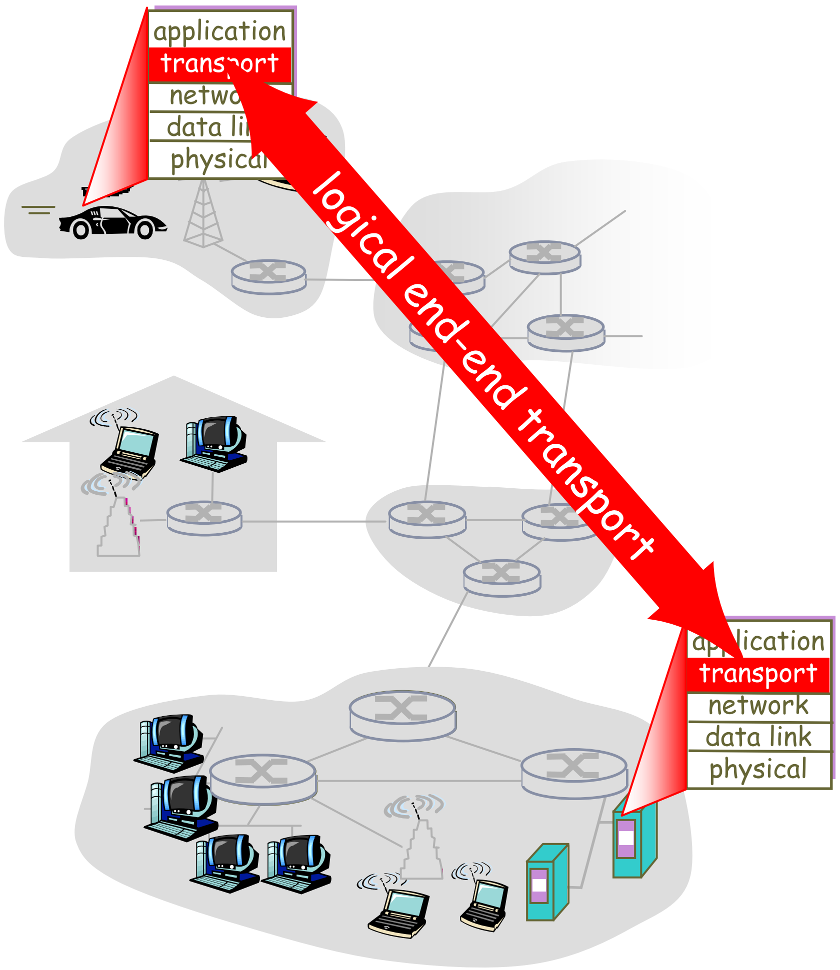
Outline

- **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**

3.1 Transport-layer services
傳輸層服務

Transport services and protocols

- **Provide *logical communication* between app processes running on different hosts**
  提供不同主機上執行應用程式之間的邏輯通訊
- **Transport protocols run in end systems**
  在終端系統間執行的傳輸協定
  - Send side: breaks app messages into segments, passes to network layer 傳送端：將應用程式的訊息分割成資料分段、傳送到網路層
  - Rcv side: reassembles segments into messages, passes to app layer
    接收端：將資料分段重組成訊息、傳給應用層
- **More than one transport protocol available to apps** 應用層可用的傳輸協定超過一個
  - Internet: TCP and UDP

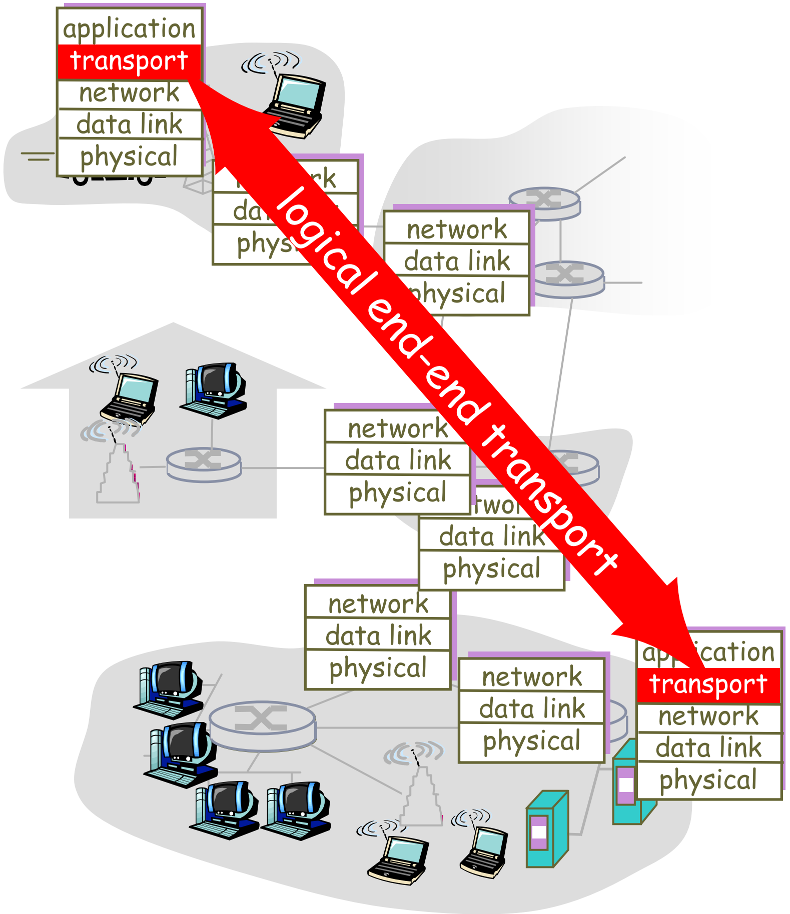

application
transport
network
data link
physical
logical end-end transport
application
transport
network
data link
physical

Transport vs. Network layer
傳輸 vs. 網路層

- ***Network layer:*** **logical communication between hosts**
  網路層：主機之間的邏輯通訊
- ***Transport layer:*** **logical communication between processes**
  傳輸層：行程之間的邏輯通訊
  - Relies on, enhances, network layer services
    依賴、 增強、 網路層服務

Internet transport-layer protocols
網際網路傳輸層協定

- **Reliable, in-order delivery (TCP)**
  可靠的、 有序的遞送
  - Congestion control壅塞控制
  - Flow control流量控制
  - Connection setup連線建立
- **Unreliable, unordered delivery: UDP**
  不可靠的、 無序的遞送
  - No-frills extension of "best-effort" IP
    "盡全力"的 IP的精簡延伸



3.2 Multiplexing and demultiplexing 多工和解多工

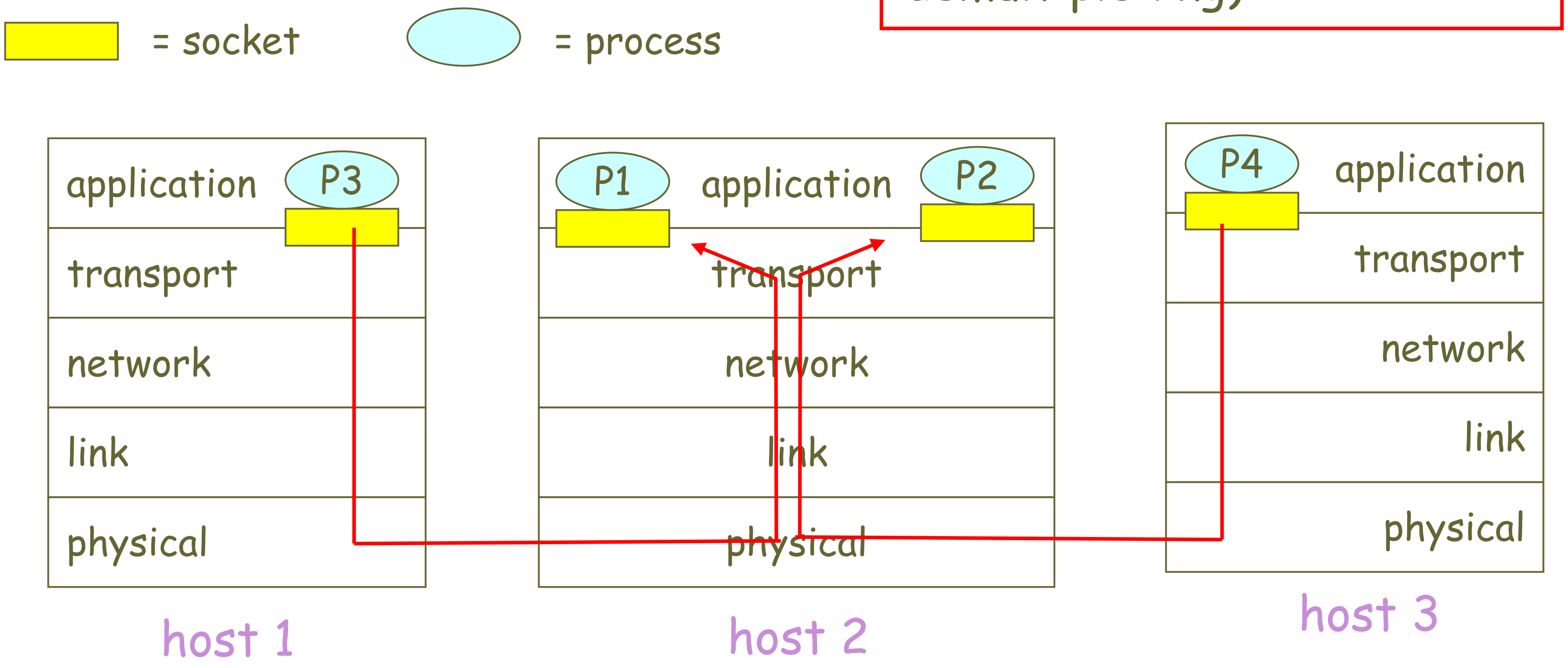
Multiplexing/demultiplexing

Demultiplexing at rcv host:

Delivering received segments to correct socket

Multiplexing at send host:

Gathering data from multiple sockets, enveloping data with header (later used for demultiplexing)



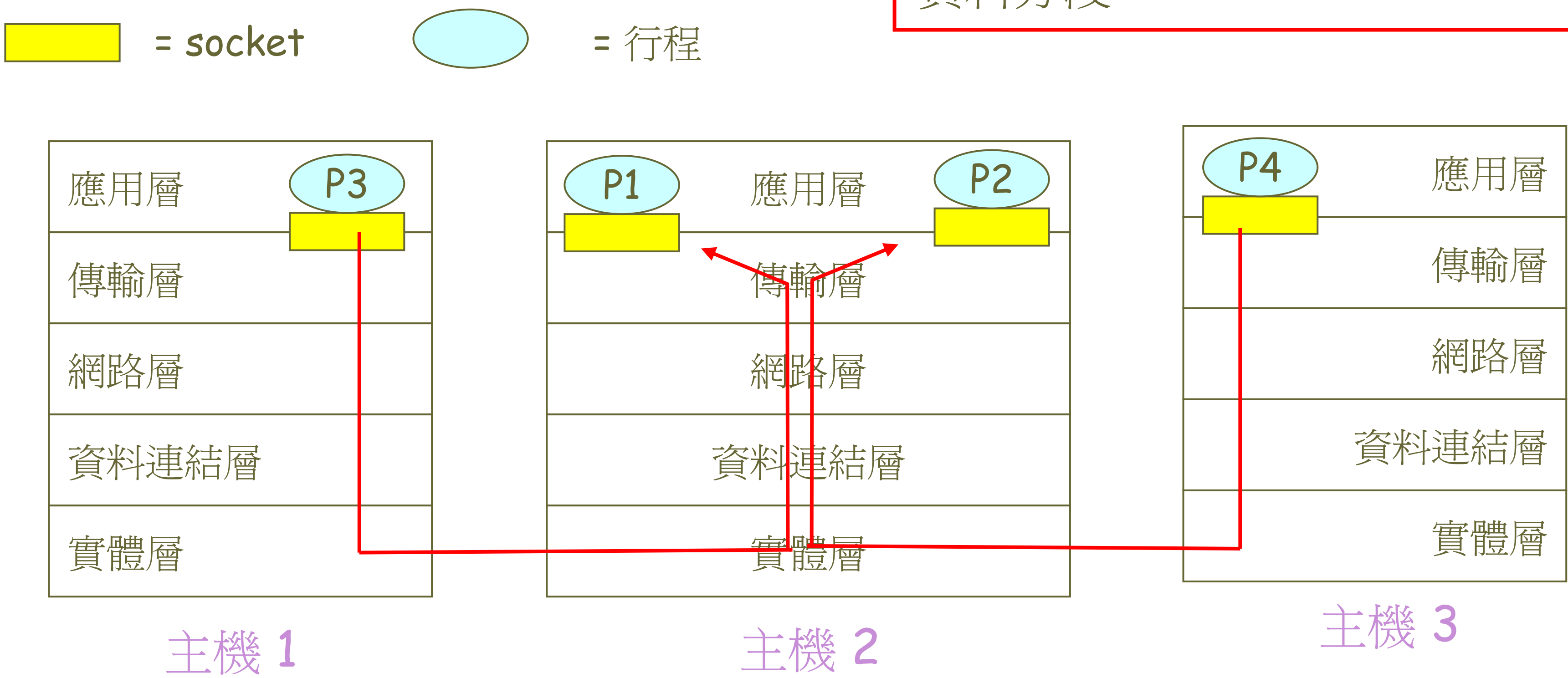
多工/解多工

接收端主機的解多工：

將收到的資料分段
傳送給正確的socket

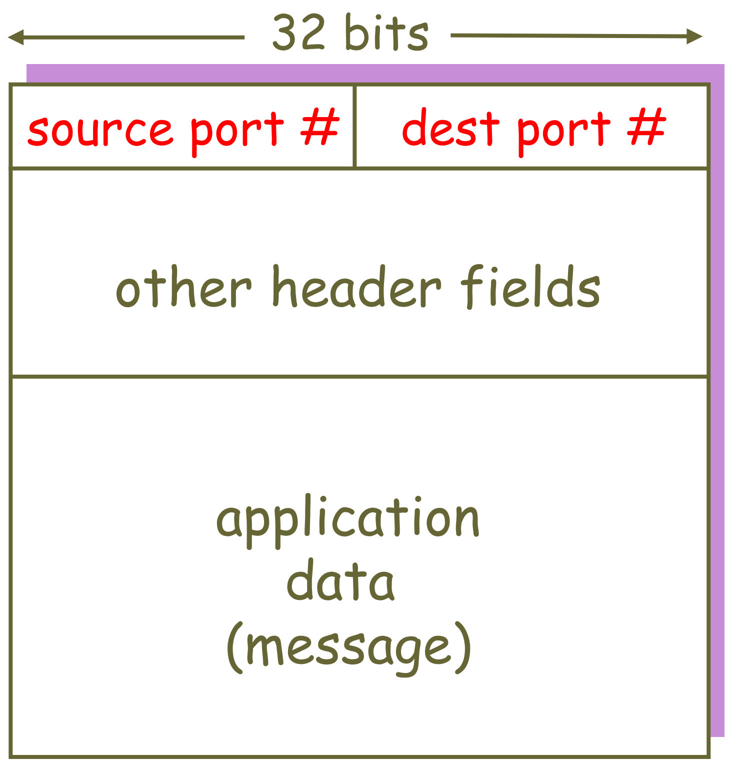
傳送端主機的多工：

收集多個socket的資料、
用標頭 (稍後將用在解多工)
將每個資料片段封裝成
資料分段



How demultiplexing works
解多工如何運作

- **Host receives IP datagrams**
  主機收到 IP 資料段
  - Each datagram has source IP address, destination IP address
    每一個資料段都擁有來源端 IP位址以及目的端IP位址
  - Each datagram carries 1 transport-layer segment
    每一個資料段載送 1 個傳輸層資料分段
  - Each segment has source, destination port number
    每一個資料分段都擁有來源端以及目的端埠號
- **Host uses IP addresses & port numbers to direct segment to appropriate socket主機使用 IP 位址以及埠號將資料分段送到正確的socket**



TCP/UDP segment format

Connectionless demultiplexing
無連線的解多工

- **Create sockets with port numbers:**
  以埠號產生socket

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

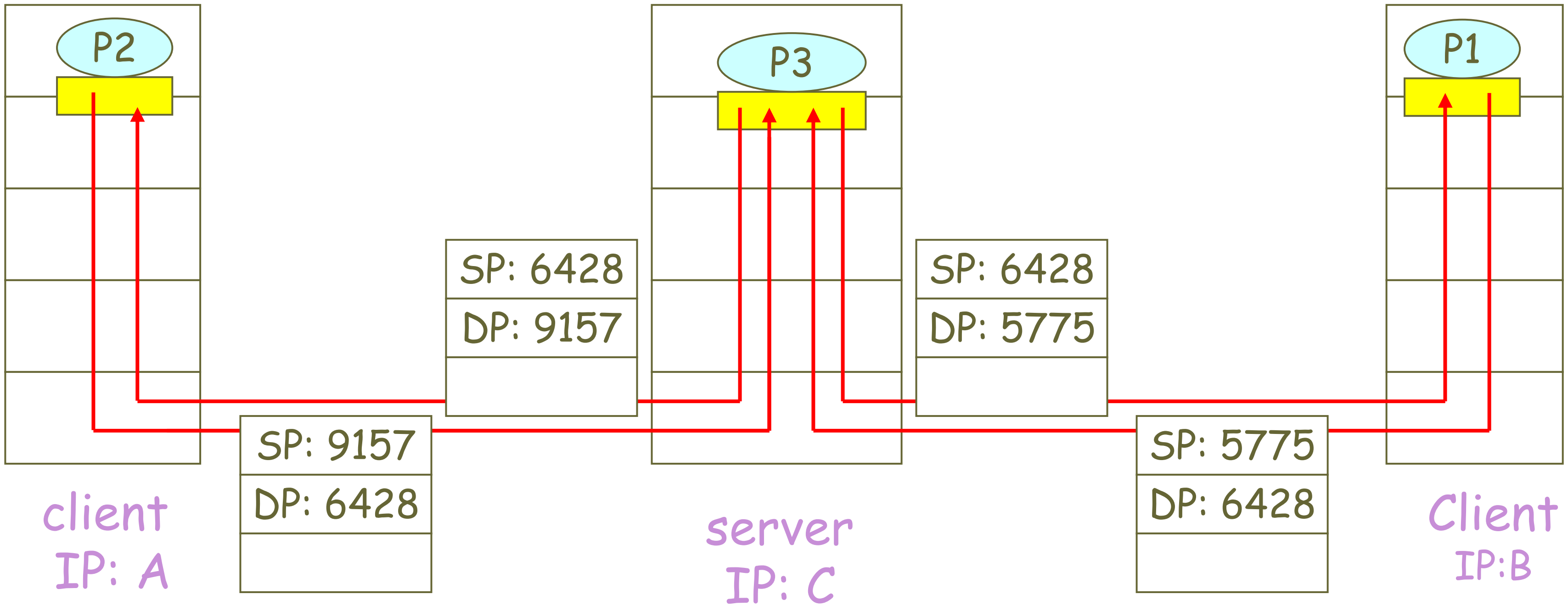
- **UDP socket identified by two-tuple:**
  以兩組資料識別 UDP socket

(dest IP address, dest port number)

- **When host receives UDP segment: 當主機收到 UDP 資料分段時**
  - Checks destination port number in segment確認資料分段中的來源端埠號
  - Directs UDP segment to socket with that port number以此埠號將UDP資料分段傳送到socket
- **IP datagrams with different source IP addresses and/or source port numbers directed to same socket 具有不同來源端 IP 位址的IP 資料段 和/或 來源端埠號會被送到同一個 socket**

Connectionless demux (cont)

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



SP provides "return address"

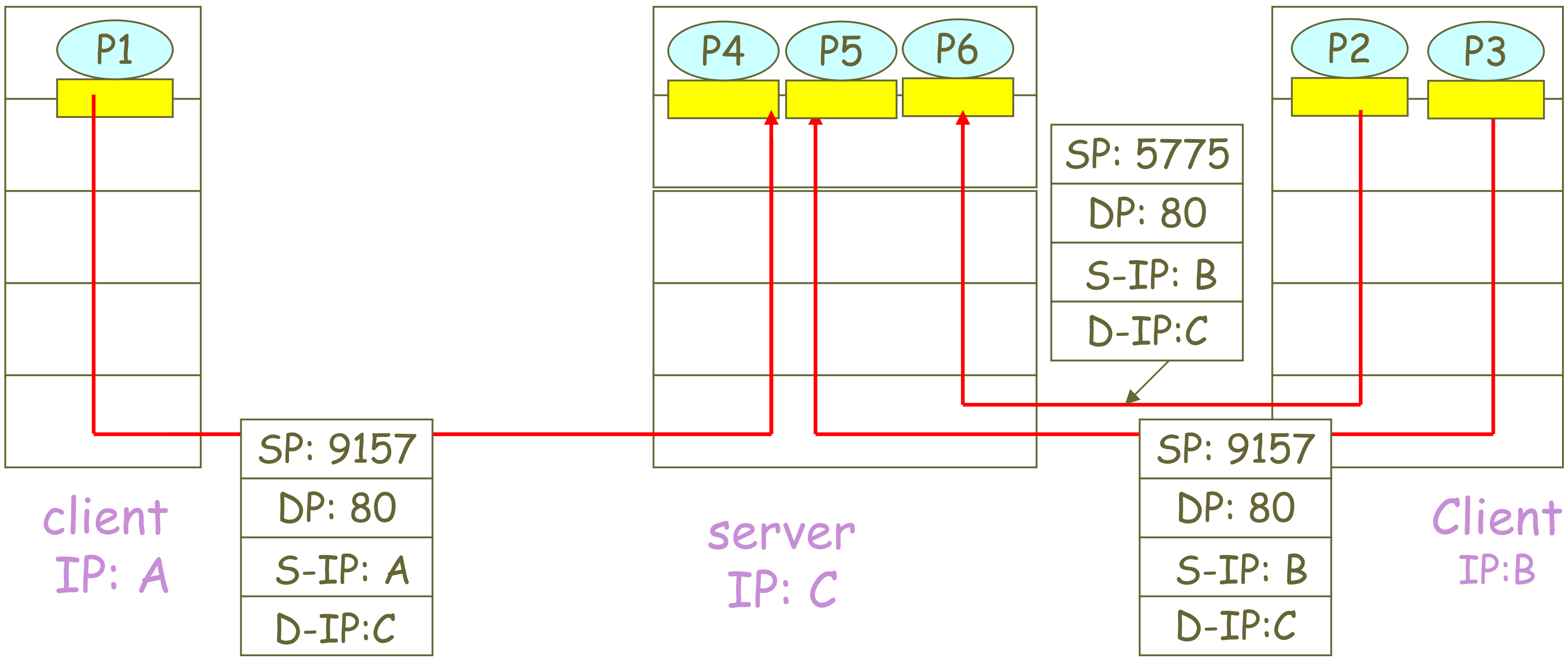
SP: Source Port
DP: Dest. Port

Connection-oriented demux

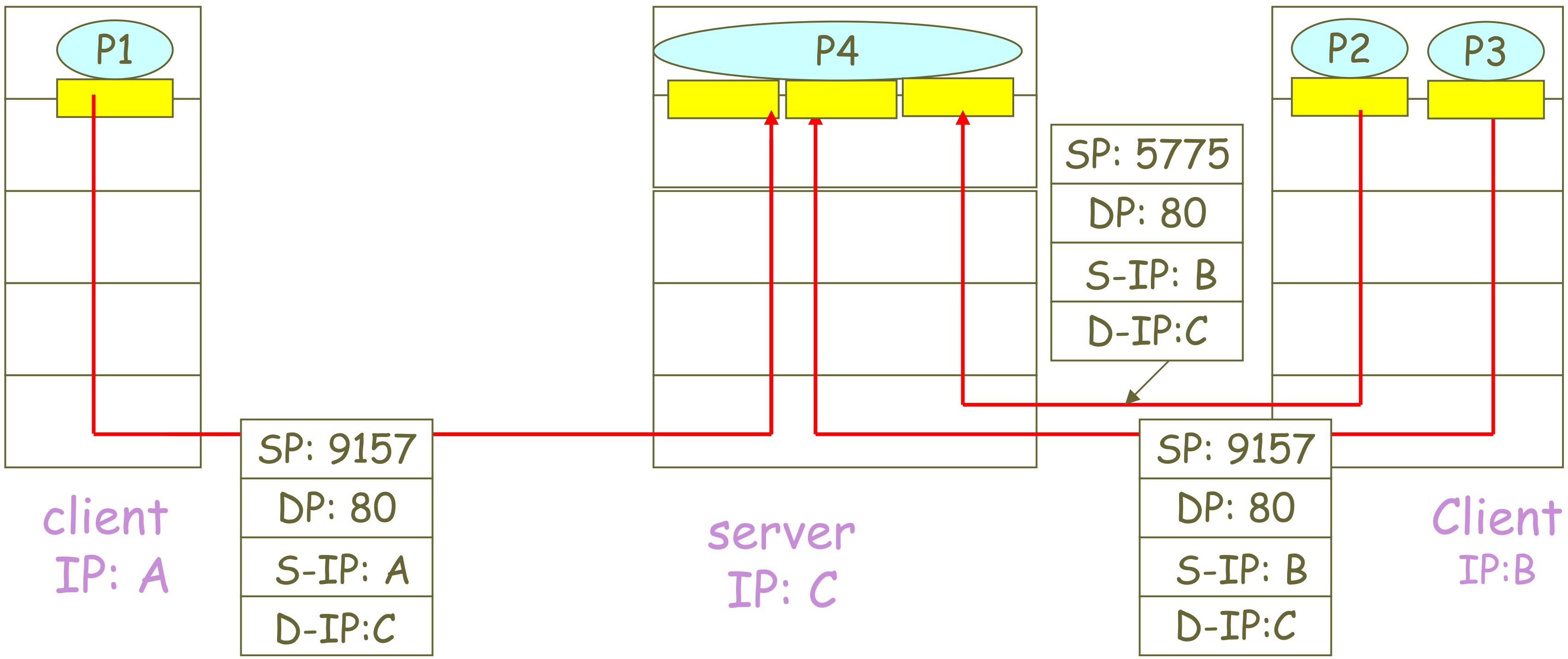
- **TCP socket identified by 4-tuple:**
  TCP socket 以四組資料加以識別
  - source IP address 來源端 IP 位址
  - source port number 來源端埠號
  - dest IP address 目的端 IP 位址
  - dest port number 目的端埠號
- **Recv host uses all four values to direct segment to appropriate socket**
  接收端主機使用全部的四個數值將資料分段送到適當的 **socket**

- **Server host may support many simultaneous TCP sockets:**
  伺服端主機可能同時支援許多**TCP sockets**
  - Each socket identified by its own 4-tuple
    每個 socket 以它自己的四組資料加以識別
- **Web servers have different sockets for each connecting client**
  Web 伺服器針對連結到它的每一個用戶端都有不同的**socket**
  - Non-persistent HTTP will have different socket for each request
    非永久性 HTTP 針對每一次的請求都有不同的 socket

Connection-oriented demux (cont)



Connection-oriented demux: Threaded Web Server



3.3 Connectionless transport: UDP 無傳輸連線UDP

UDP: User Datagram Protocol
[RFC 768]

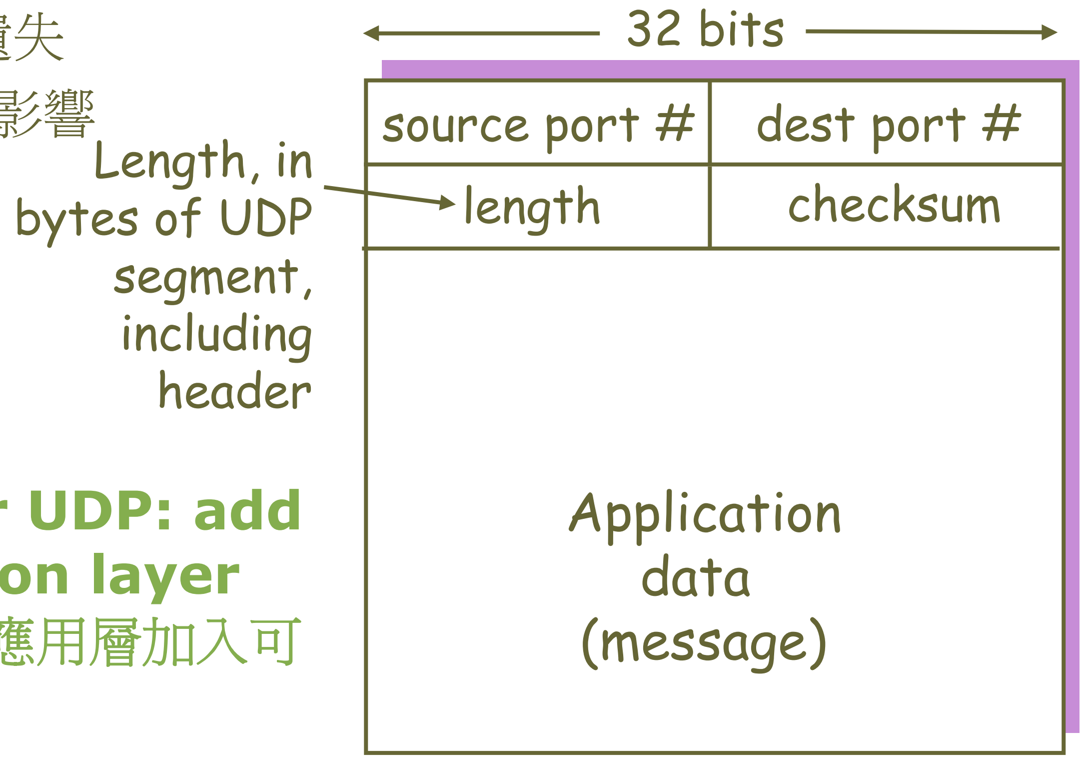
- “No frills,” “bare bones” Internet transport protocol 實際的、精簡的網際網路傳輸協定
- “Best effort” service, UDP segments may be: “盡全力” 的服務、 UDP 資料分段可能
  - Lost遺失
  - Delivered out of order to app不按順序傳送給應用程式
- ***Connectionless:***非預接式服務
  - No handshaking between UDP sender, receiver 在 UDP 傳送端和接收單之間沒有交握程序
  - Each UDP segment handled independently of others 每一個 UDP 資料分段的處理和其它資料分段是獨立的

Why is there a UDP?
爲什麼會使用 UDP?

- No connection establishment (which can add delay)
  不需建立連線 (會增加延遲)
- Simple: no connection state at sender, receiver
  簡單： 在傳送端和接收端不需維持連線狀態
- Small segment header較小的封包標頭
- No congestion control: UDP can blast away as fast as desired
  沒有壅塞控制： UDP 可以僅可能地快速傳送資料

UDP: more

- **Often used for streaming multimedia apps**
  通常用在串流的多媒體應用程式
  - Loss tolerant可以容忍遺失
  - Rate sensitive易受速率影響
- **Other UDP uses**
  其他使用 **UDP** 的有
  - DNS
  - SNMP
- **Reliable transfer over UDP: add reliability at application layer**
  使用**UDP**的可靠傳輸： 在應用層加入可靠性的機制
  - Application-specific error recovery!
    應用層指定的錯誤復原



UDP segment format

UDP checksum檢查和

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

目標： 偵測傳送的資料分段中的 "錯誤" (例如:被翻轉的位元)

Sender:

- Treat segment contents as sequence of 16-bit integers
  將資料分段的內容視爲一列16位元的整數
- Checksum: addition (1's complement sum) of segment contents檢查和：資料分段內容的加法 (1的補數和)
- Sender puts checksum value into UDP checksum field傳送端將檢查和的值放入UDP的檢查和欄位

Receiver:

- Compute checksum of received segment計算收到的資料分段的檢查和
- Check if computed checksum equals checksum field value:確認計算出來的檢查和是否和檢查和欄位中的相等
  - NO - error detected偵測到錯誤
  - YES - no error detected. *But maybe errors nonetheless?*沒有偵測到錯誤。但是仍然可能有錯誤

Internet Checksum Example網際網路的檢查和範例

❖ Note

- When adding numbers, a carryout from the most significant bit needs to be added to the result
當數字加總時、最高位元的進位必須被加回結果中

❖ Example: add two 16-bit integers

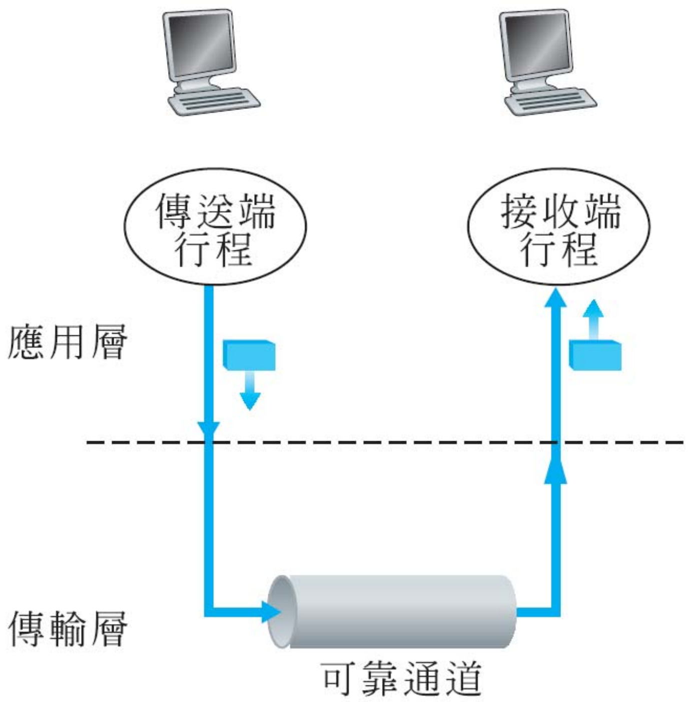
加總兩個 16 位元的整數

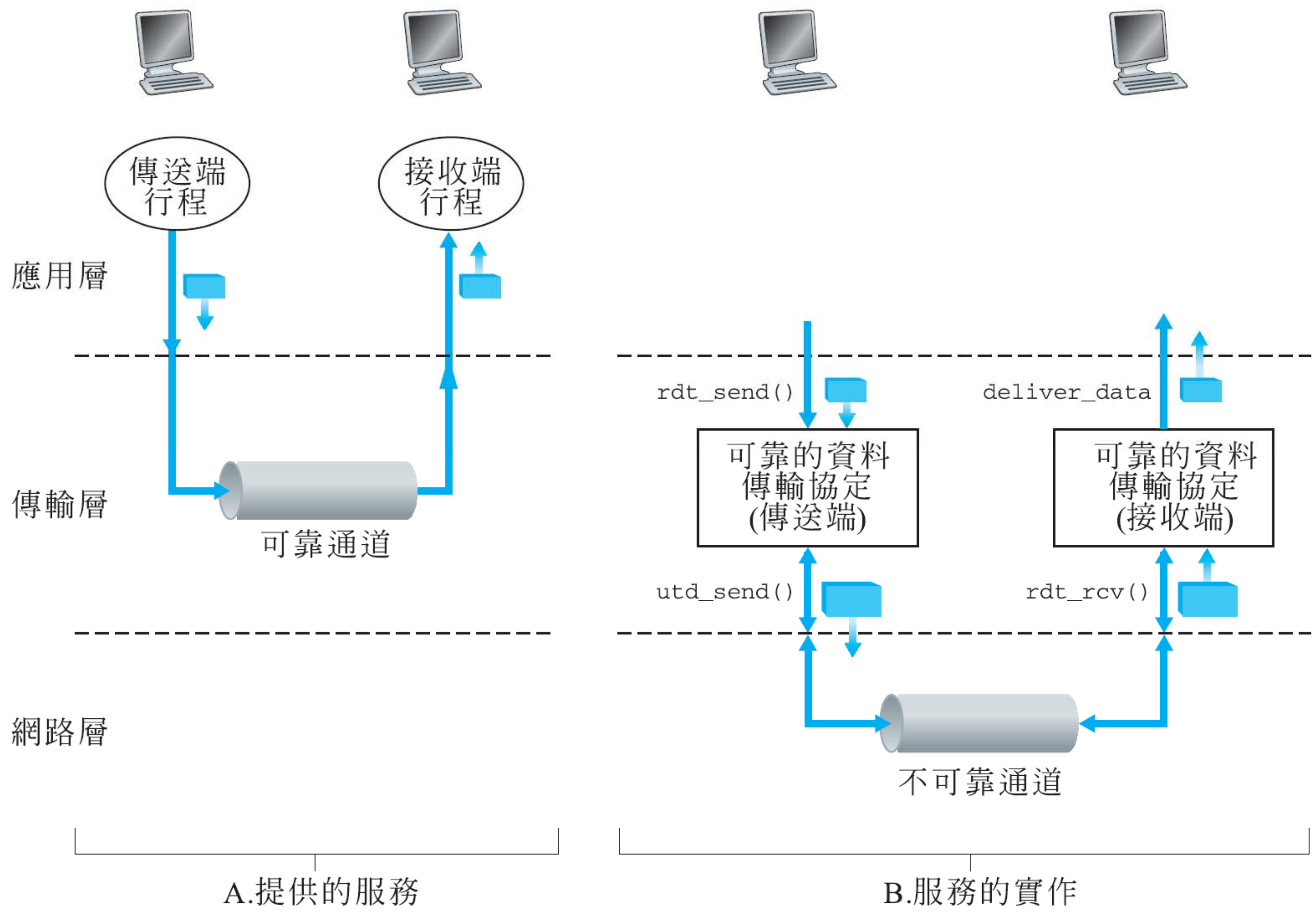
		1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
		1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
Wraparound 繞回去	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
sum		1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum		0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

3.4 Principles of reliable data transfer可靠資料傳輸的原理

Principles of Reliable data transfer可靠資料傳輸的原理

- **Important in app., transport, link layers**
  在應用層、傳輸層、資料連結層中都是很重要的
- **Characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)**
  不可靠通道的特性決定了可靠資料傳輸協定 **(rdt)** 的複雜性

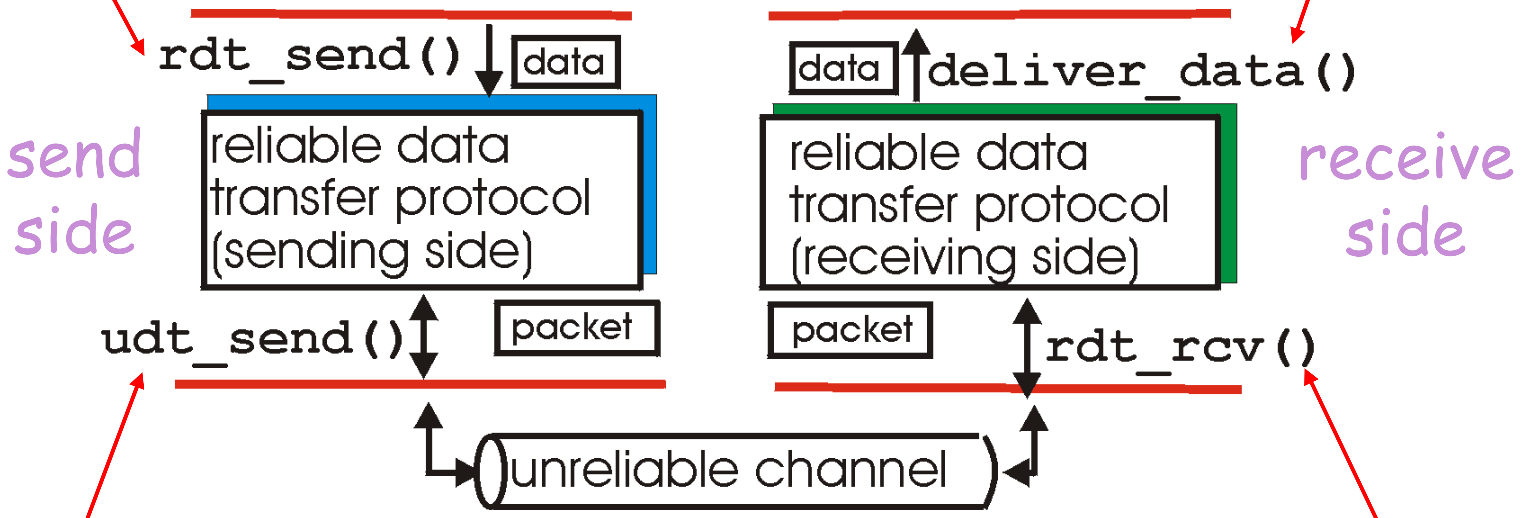



傳送端
行程
接收端
行程
應用層
傳輸層
網路層
可靠通道
rdt_send()
deliver_data
可靠的資料
傳輸協定
(傳送端)
可靠的資料
傳輸協定
(接收端)
utd_send()
rdt_rcv()
不可靠通道
A.提供的服務
B.服務的實作

Reliable data transfer: getting started

rdt_send(): called from above, (e.g., by app.). Passed data to deliver to receiver upper layer

deliver_data(): called by **rdt** to deliver data to upper



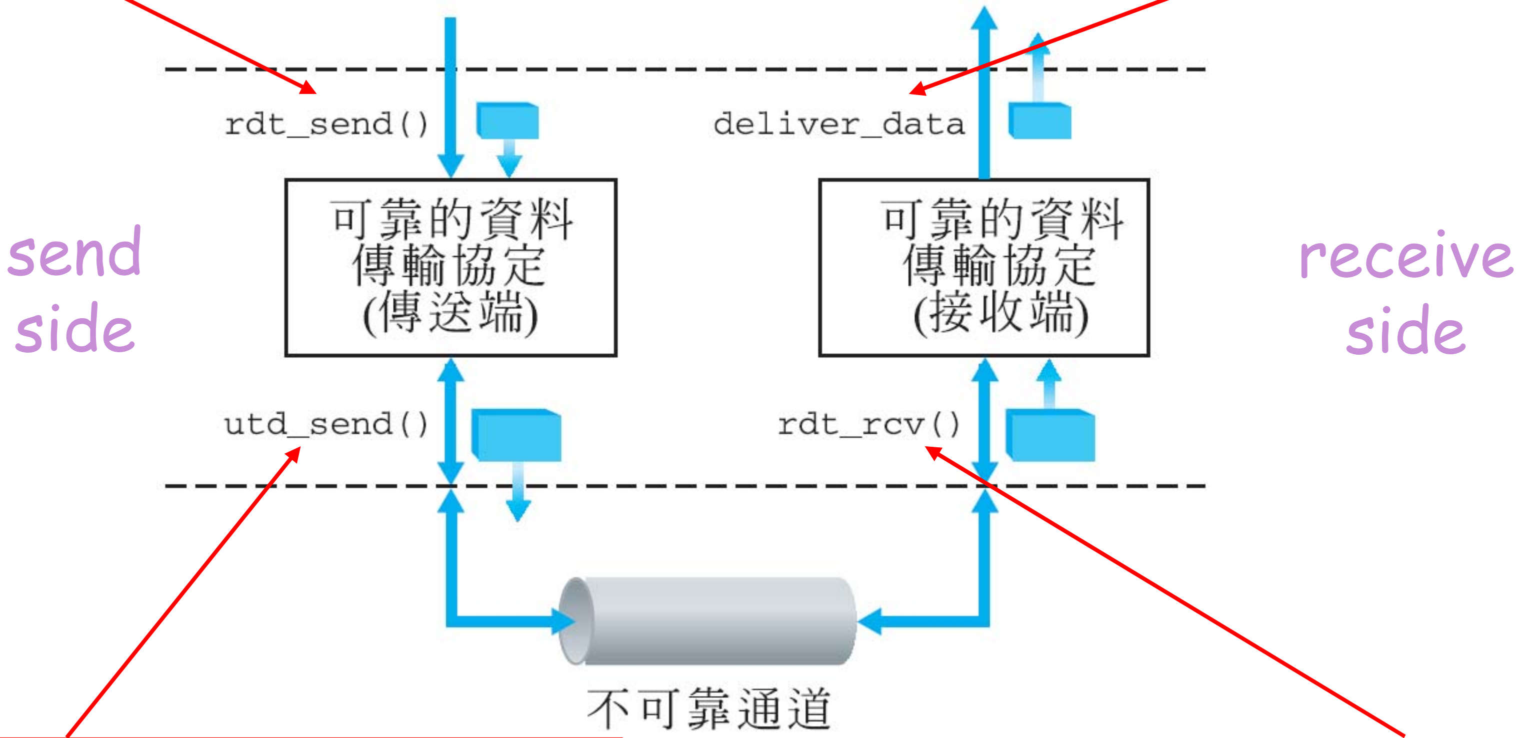
udt_send(): called by rdt, to transfer packet over unreliable channel to receiver

rdt_rcv(): called when packet arrives on rcv-side of channel

可靠的資料傳輸： 開始

rdt_send()： 被上層呼叫、 (例如應用層). 將資料傳遞給接收端的上層協定

deliver_data()： 被 rdt 呼叫、將資料傳送到上層



udt_send()： 被rdt呼叫、經由不可靠的通道將封包傳送給接收端

rdt_rcv()： 當封包抵達接收端的通道時被呼叫

參考內容(不考)

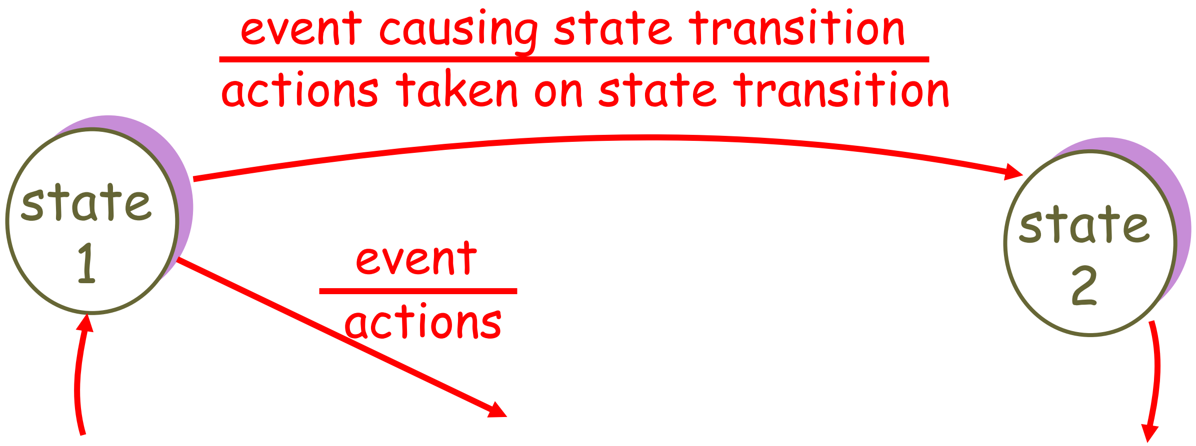
pp. 30-64

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) to specify sender, receiver**

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



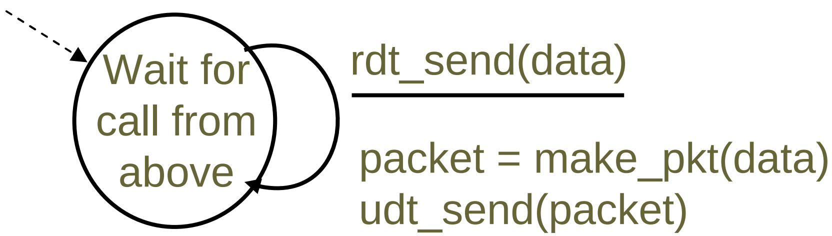
Rdt1.0: reliable transfer over a reliable channel

Underlying channel perfectly reliable

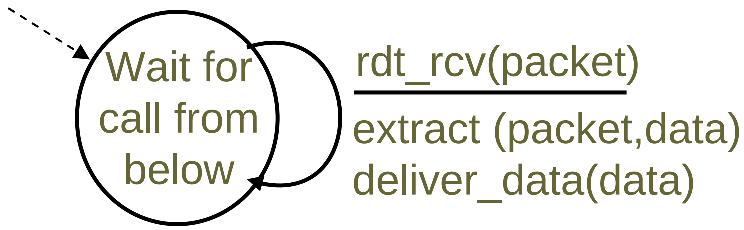
- No bit errors
- No loss of packets

Separate FSMs for sender, receiver:

- Sender sends data into underlying channel
- Receiver read data from underlying channel



sender

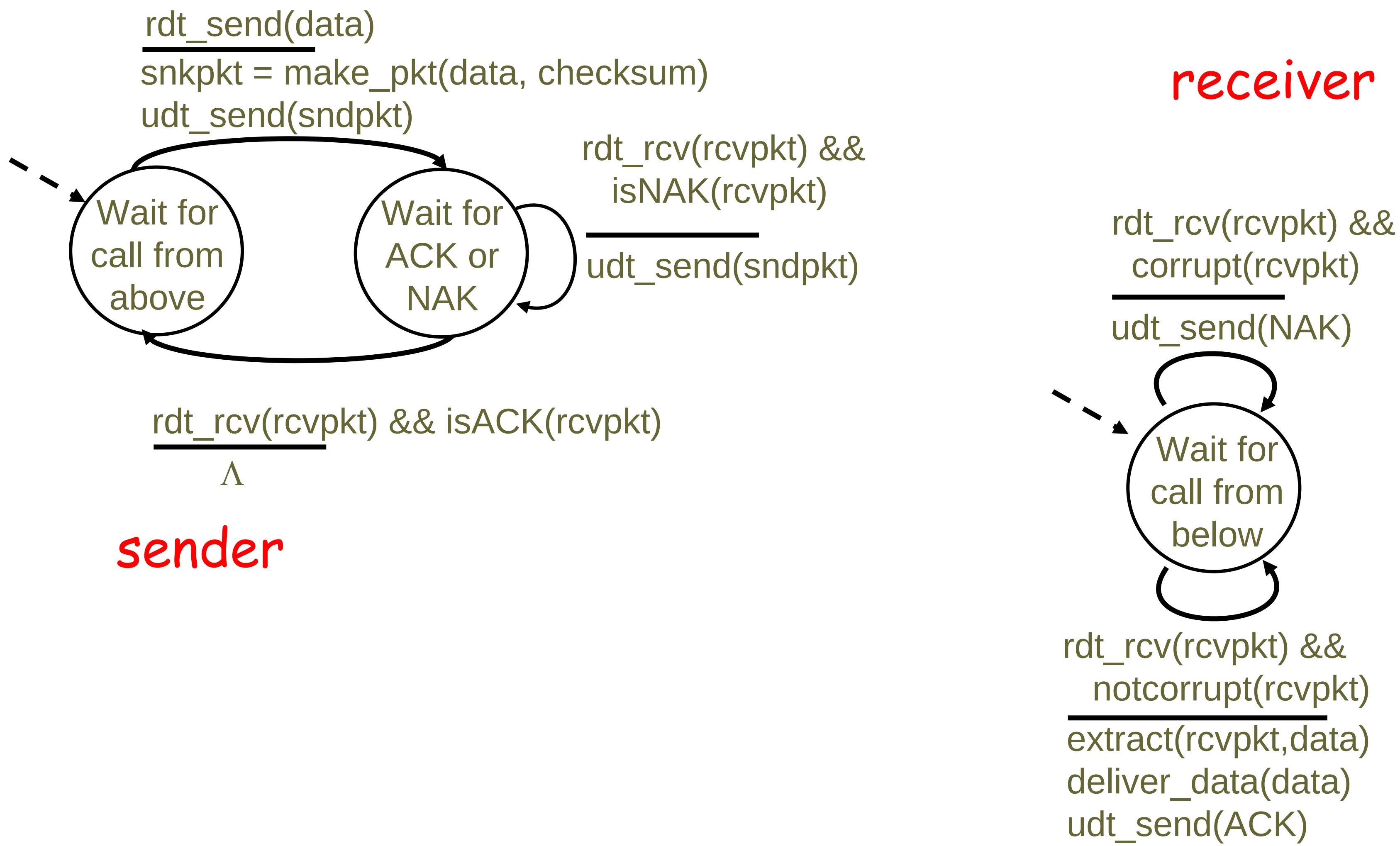


receiver

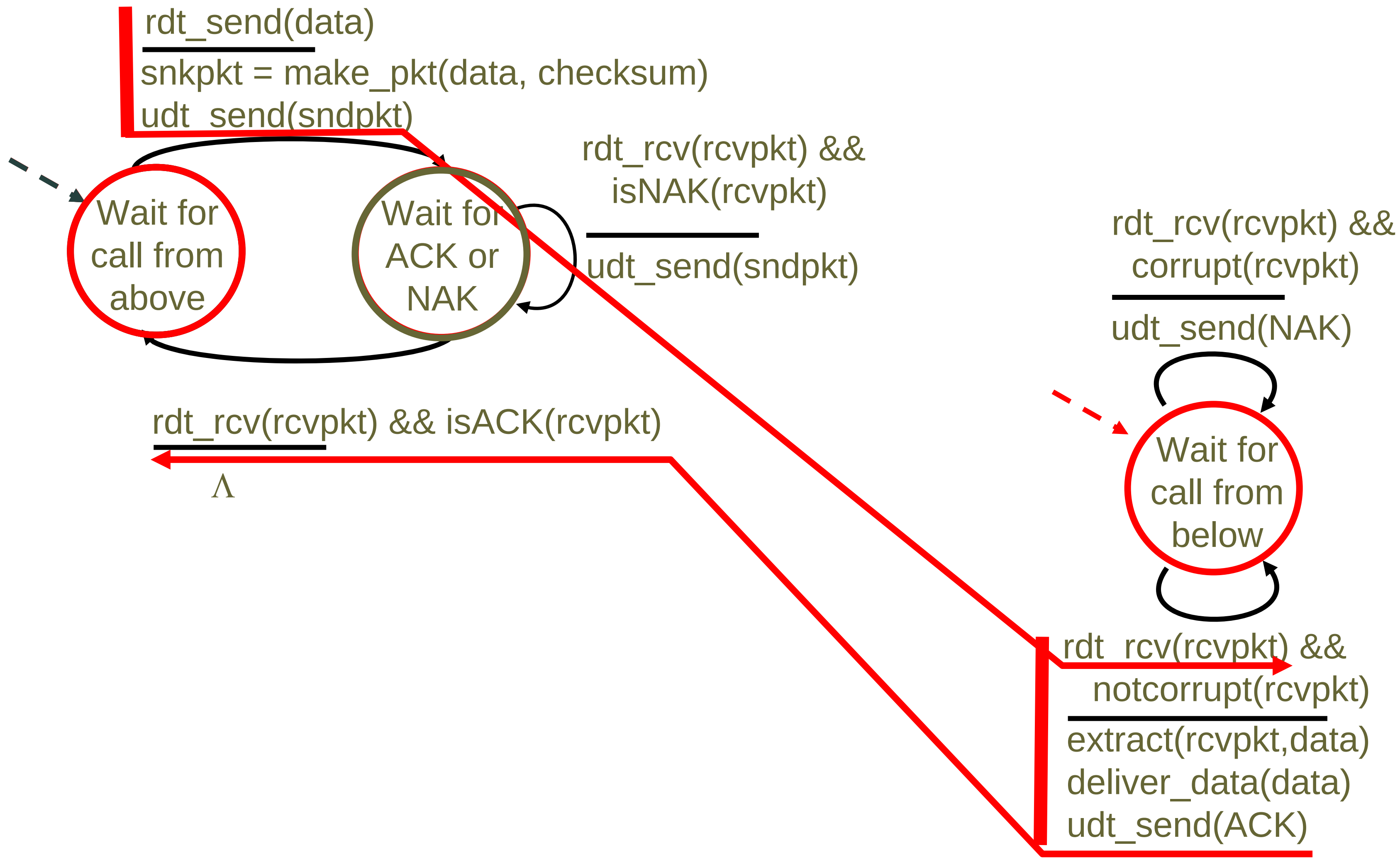
Rdt2.0: channel with bit errors

- **Underlying channel may flip bits in packet**
  - Checksum to detect bit errors
- ***The* question: how to recover from errors:**
  - *Acknowledgements (ACKs):* receiver explicitly tells sender that pkt received OK
  - *Negative acknowledgements (NAKs):* receiver explicitly tells sender that pkt had errors
  - Sender retransmits pkt on receipt of NAK
- **New mechanisms in `rdt2.0` (beyond `rdt1.0`):**
  - Error detection
  - Receiver feedback: control msgs (ACK,NAK) rcvr->sender

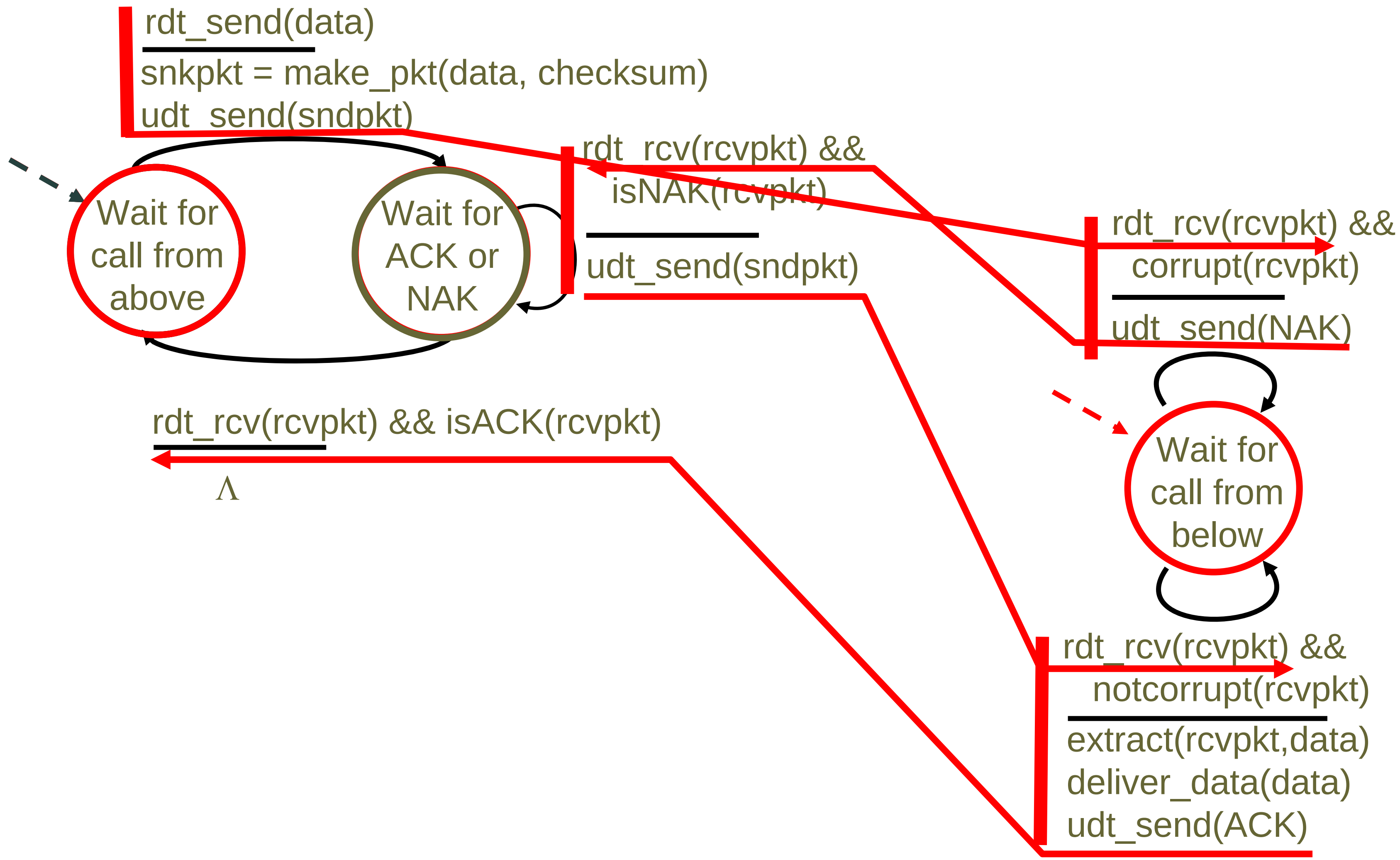
rdt2.0: FSM specification



rdt2.0: operation with no errors



rdt2.0: error scenario



rdt2.0 has a fatal flaw!

What happens if ACK/NAK corrupted?

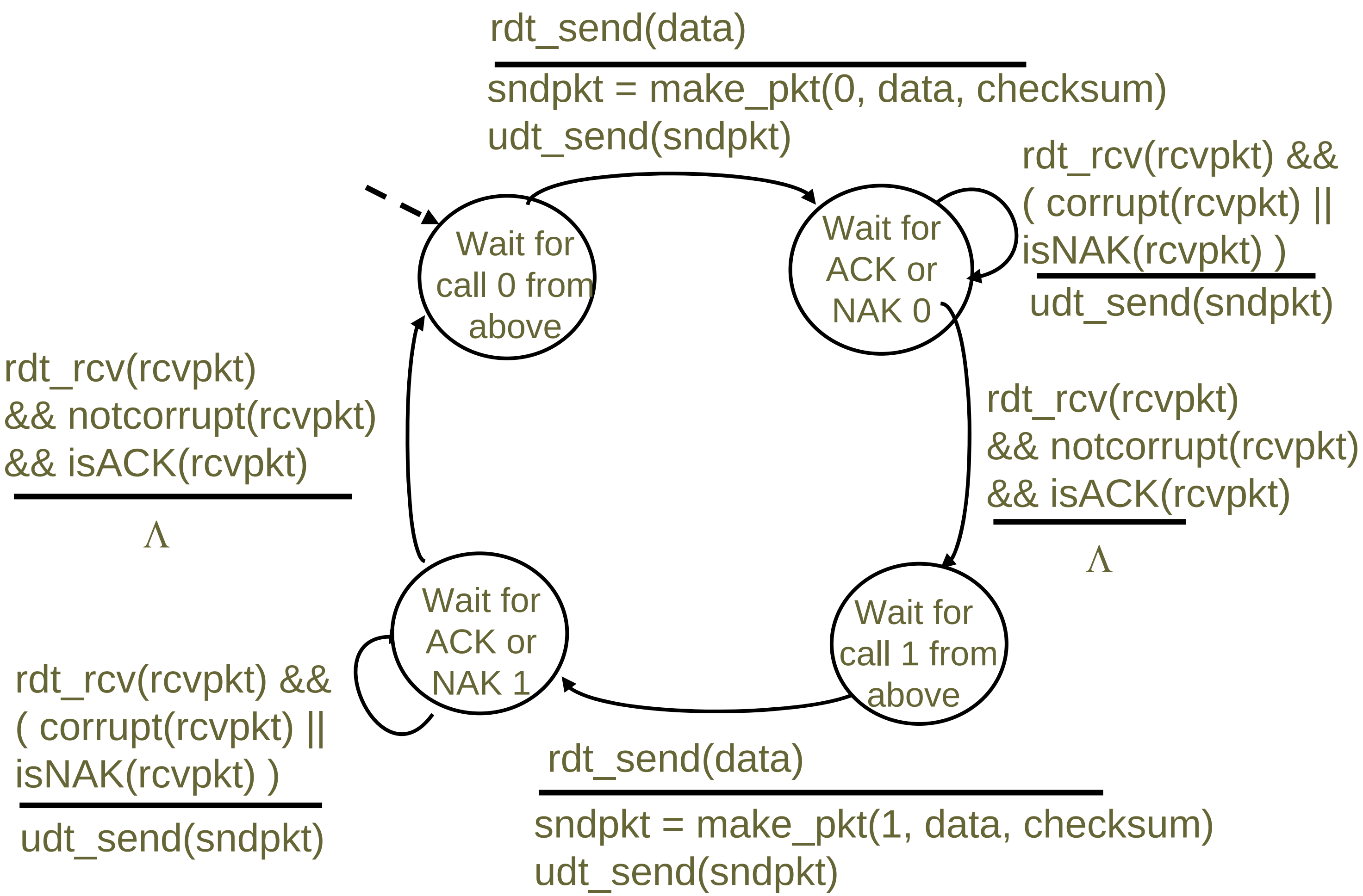
- Sender doesn't know what happened at receiver!
- Can't just retransmit: possible duplicate

Handling duplicates:

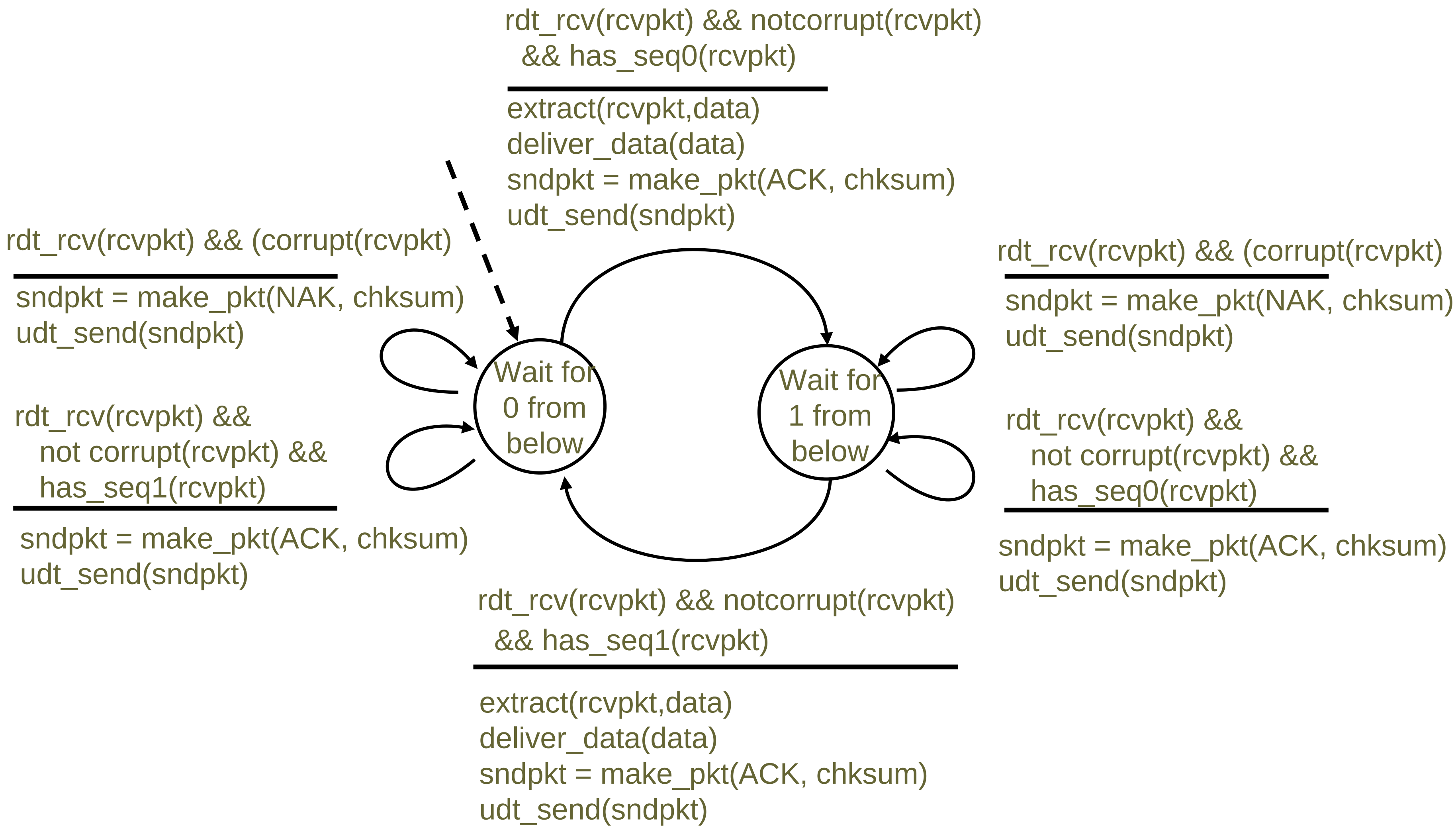
- Sender retransmits current pkt if ACK/NAK garbled
- Sender adds *sequence number* to each pkt
- Receiver discards (doesn't deliver up) duplicate pkt

stop and wait
Sender sends one packet, then waits for receiver response

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?
- Must check if received ACK/NAK corrupted
- Twice as many states
  - State must "remember" whether "current" pkt has 0 or 1 seq. #

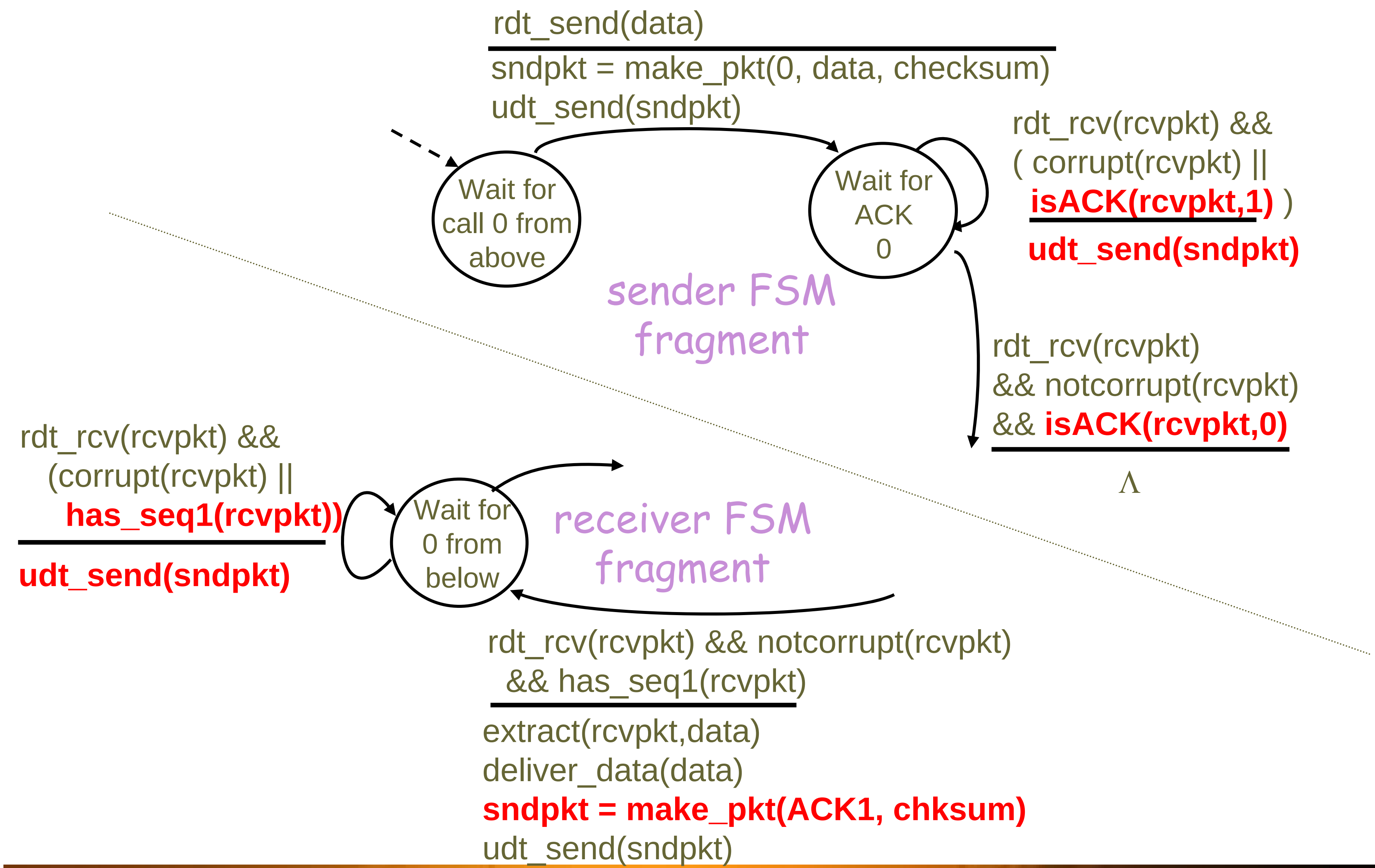
Receiver:

- 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

- **Same functionality as rdt2.1, using ACKs only**
- **Instead of NAK, receiver sends ACK for last pkt received OK**
  - Receiver must *explicitly* include seq # of pkt being ACKed
- **Duplicate ACK at sender results in same action as NAK: *retransmit current pkt***

rdt2.2: sender, receiver fragments



rdt3.0: channels with errors *and* loss

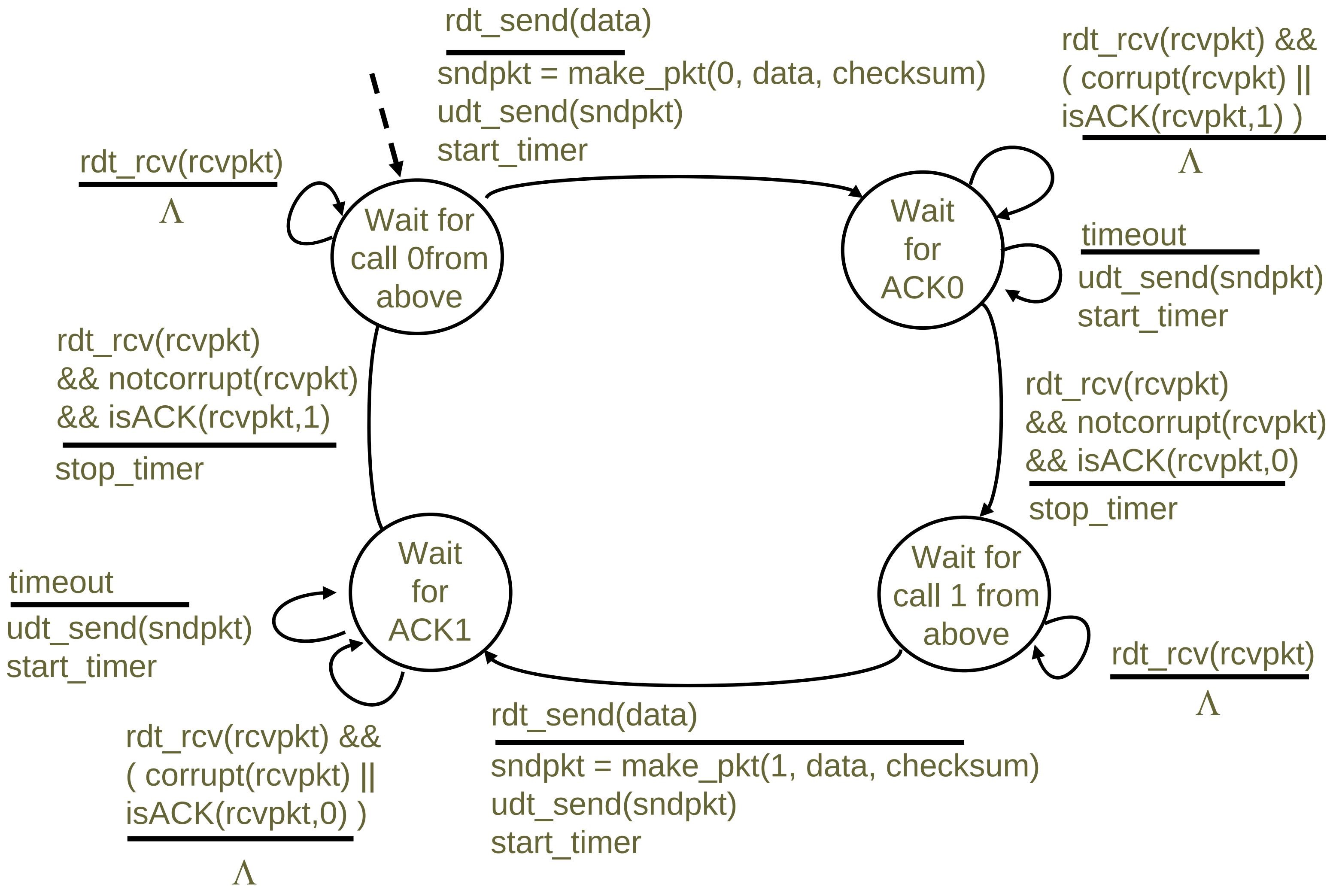
New assumption: underlying channel can also lose packets (data or ACKs)

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

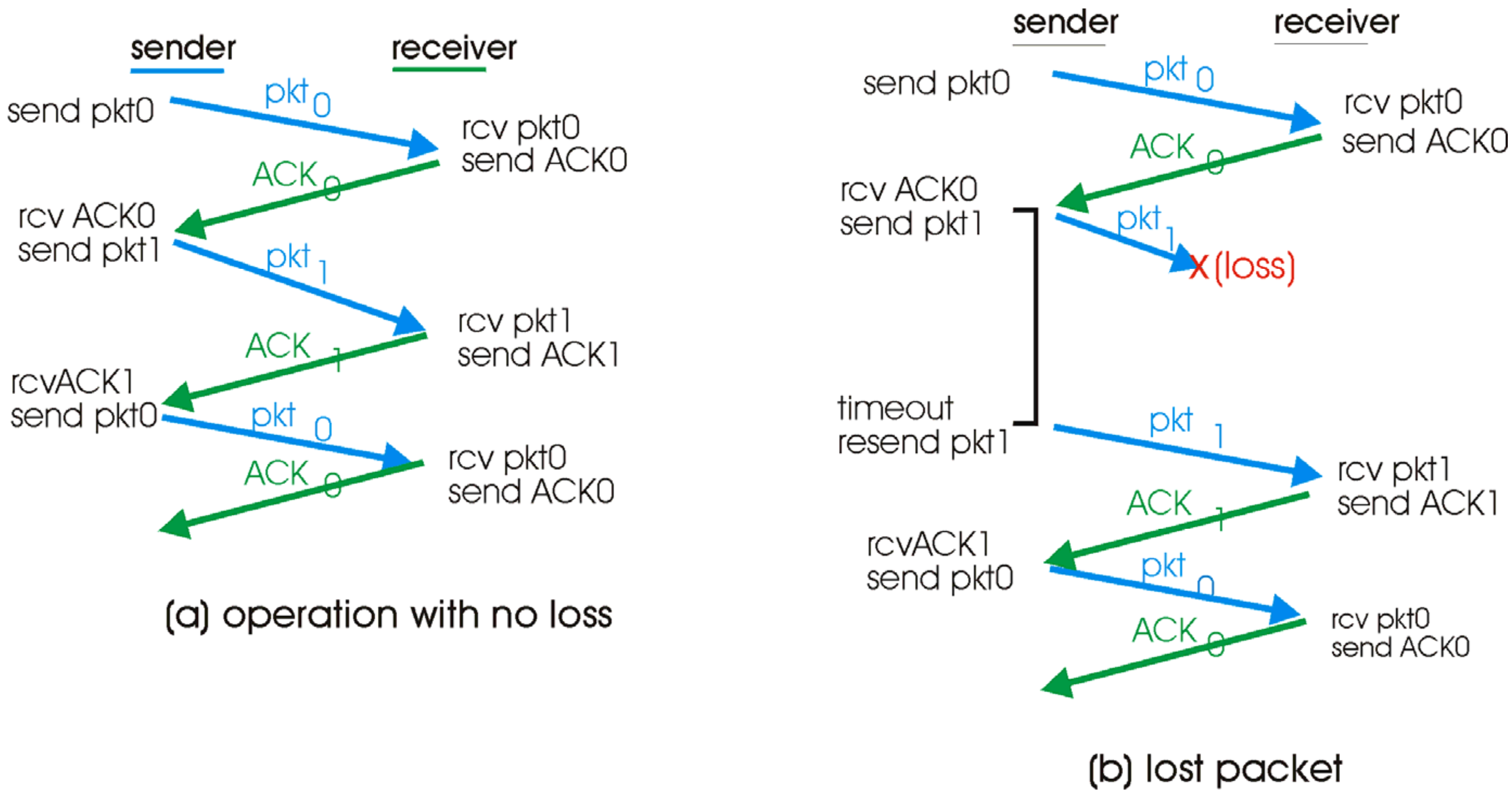
Approach: sender waits "reasonable" amount of time for ACK

- 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

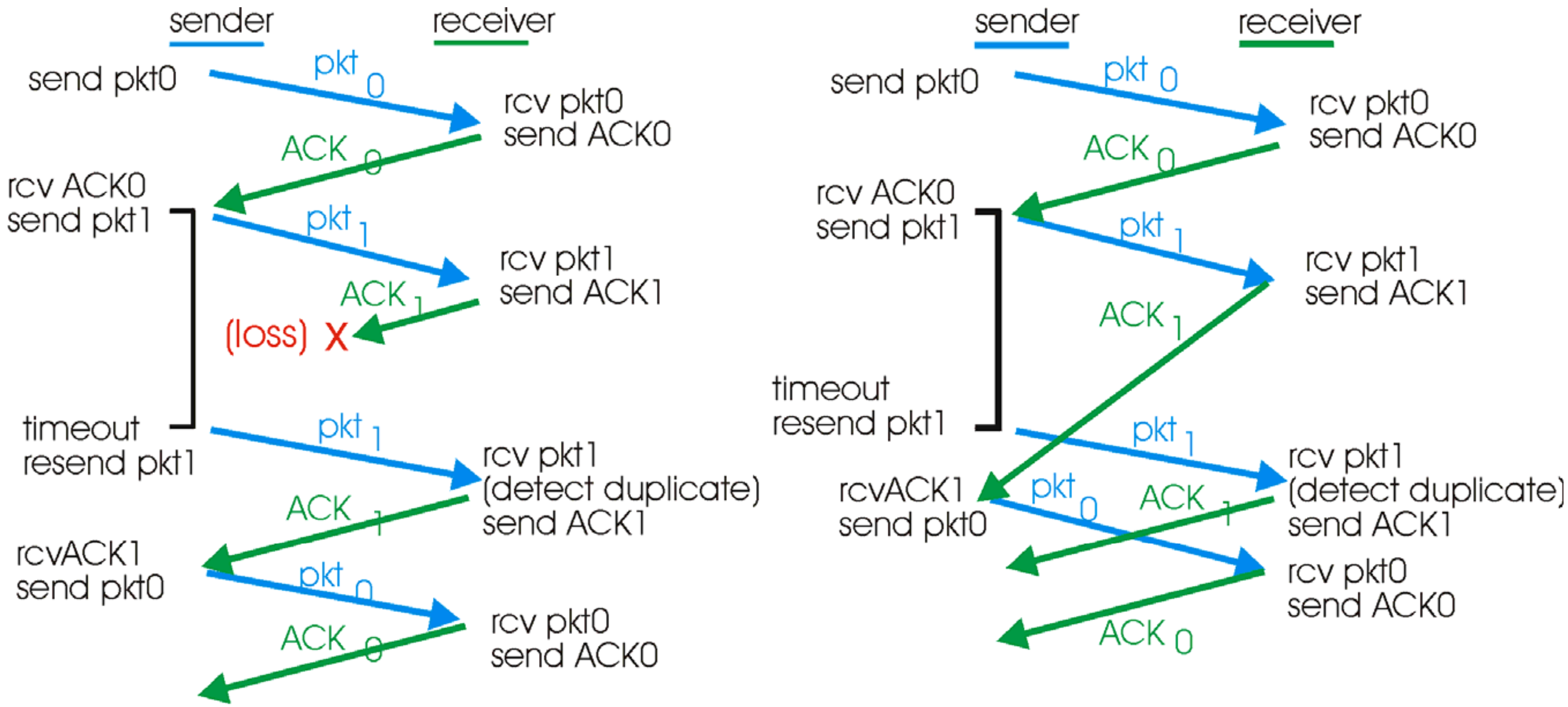
rdt3.0 sender



rdt3.0 in action



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:

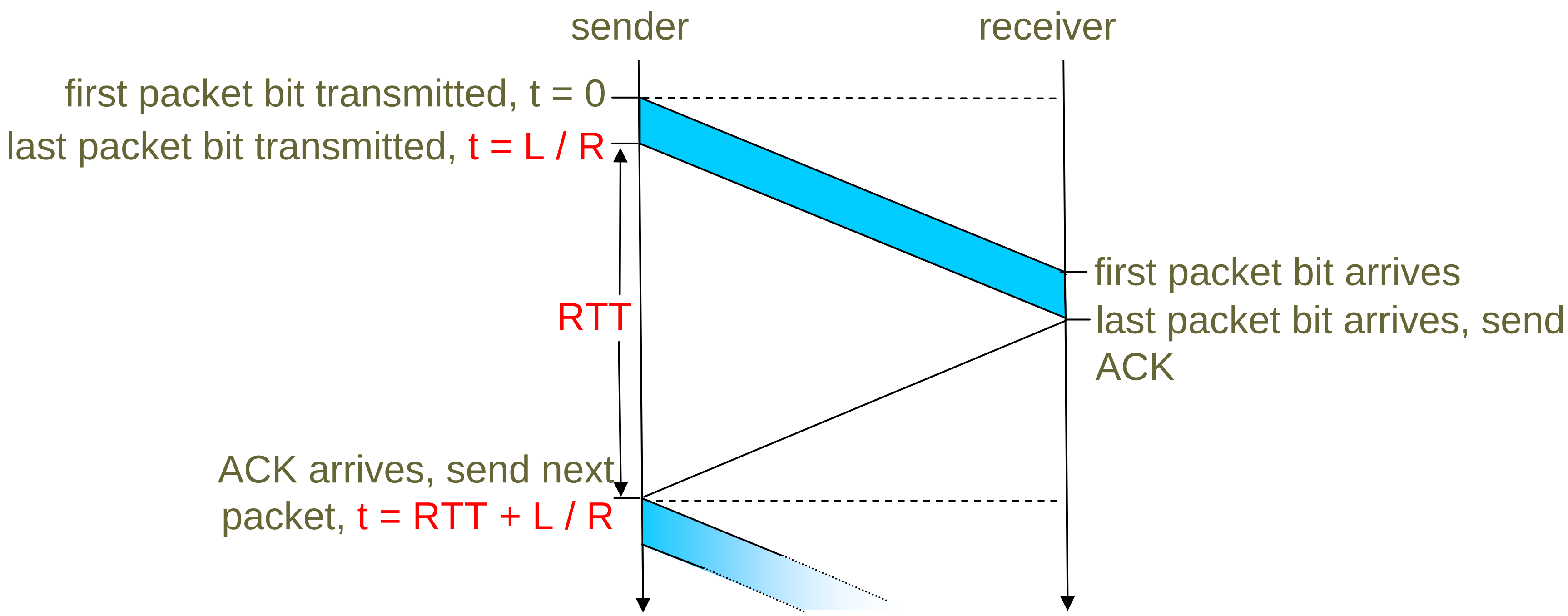
$$T_{transmit} = \frac{L \text{ (packet length in bits)}}{R \text{ (transmission rate, bps)}} = \frac{8\text{kb/pkt}}{10^{**}9 \text{ b/sec}} = 8 \text{ microsec}$$

- 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

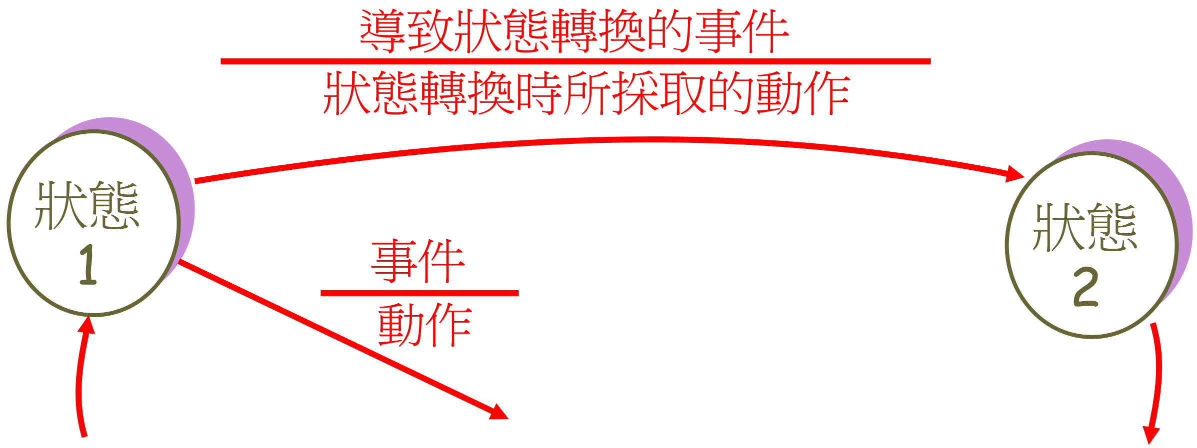


$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

可靠的資料傳輸： 開始

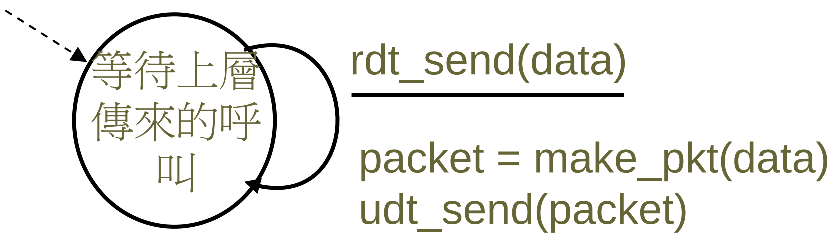
- 我們將會：
- 漸進式地建立傳送端、接收端的可靠資料傳輸協定 **(rdt)**
- 只探討單向的資料傳輸
  - 但是控制資訊會在雙向流動!
- 使用有限狀態機 **(FSM)**指定傳送端、 接收端

狀態： 在這個"狀態"時、下一個狀態將唯一地被下一個事件所決定

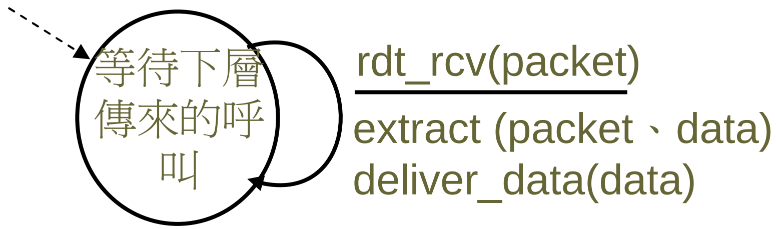


Rdt1.0： 使用可靠通道的可靠傳輸

- 底層的通道是完全可靠的
  - 沒有位元錯誤
  - 沒有資料遺失
- 傳送端和接收端擁有各自的 **FSM**：
  - 傳送端將資料送入底層的通道
  - 接收端從底層的通道接收資料



傳送端

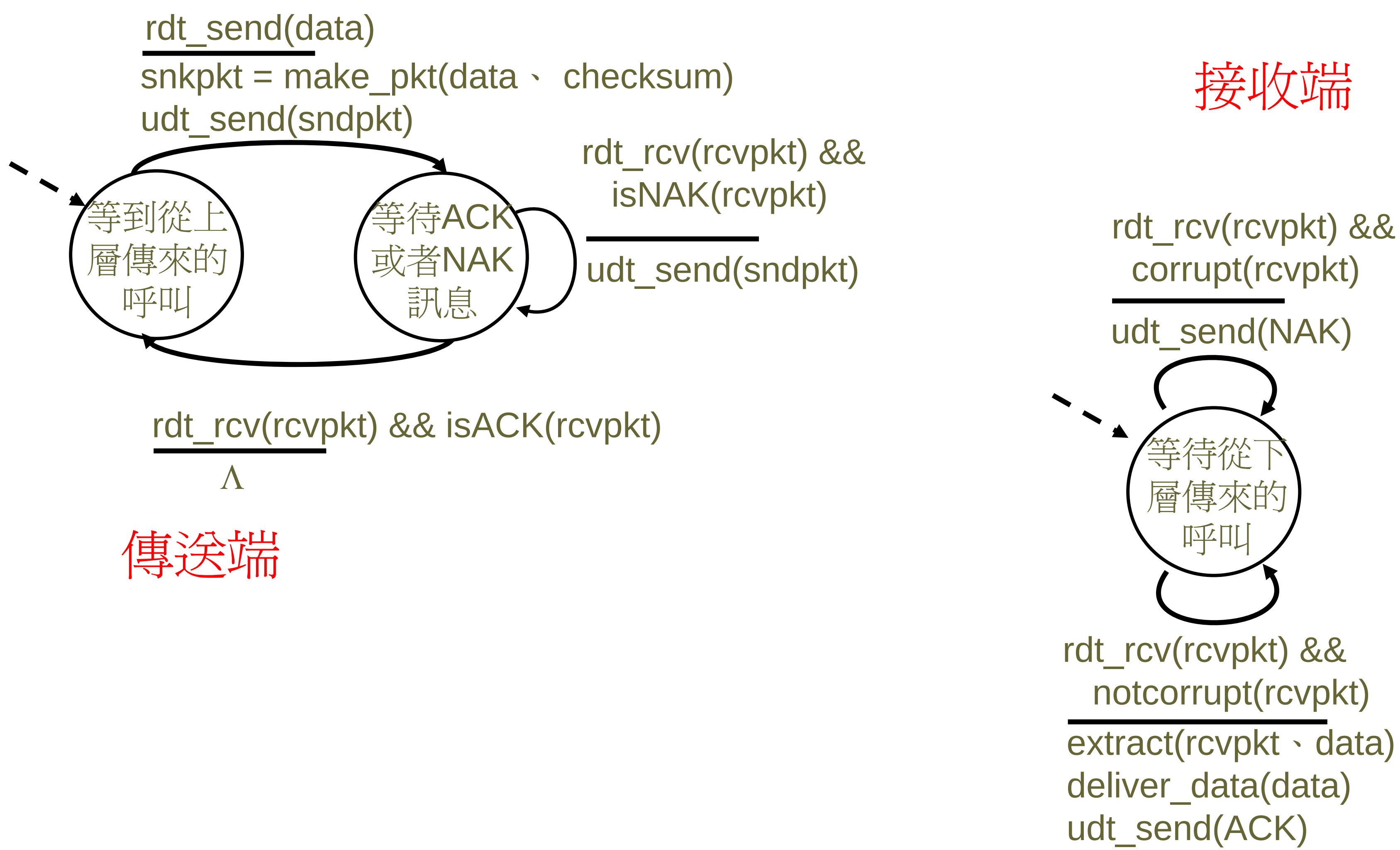


接收端

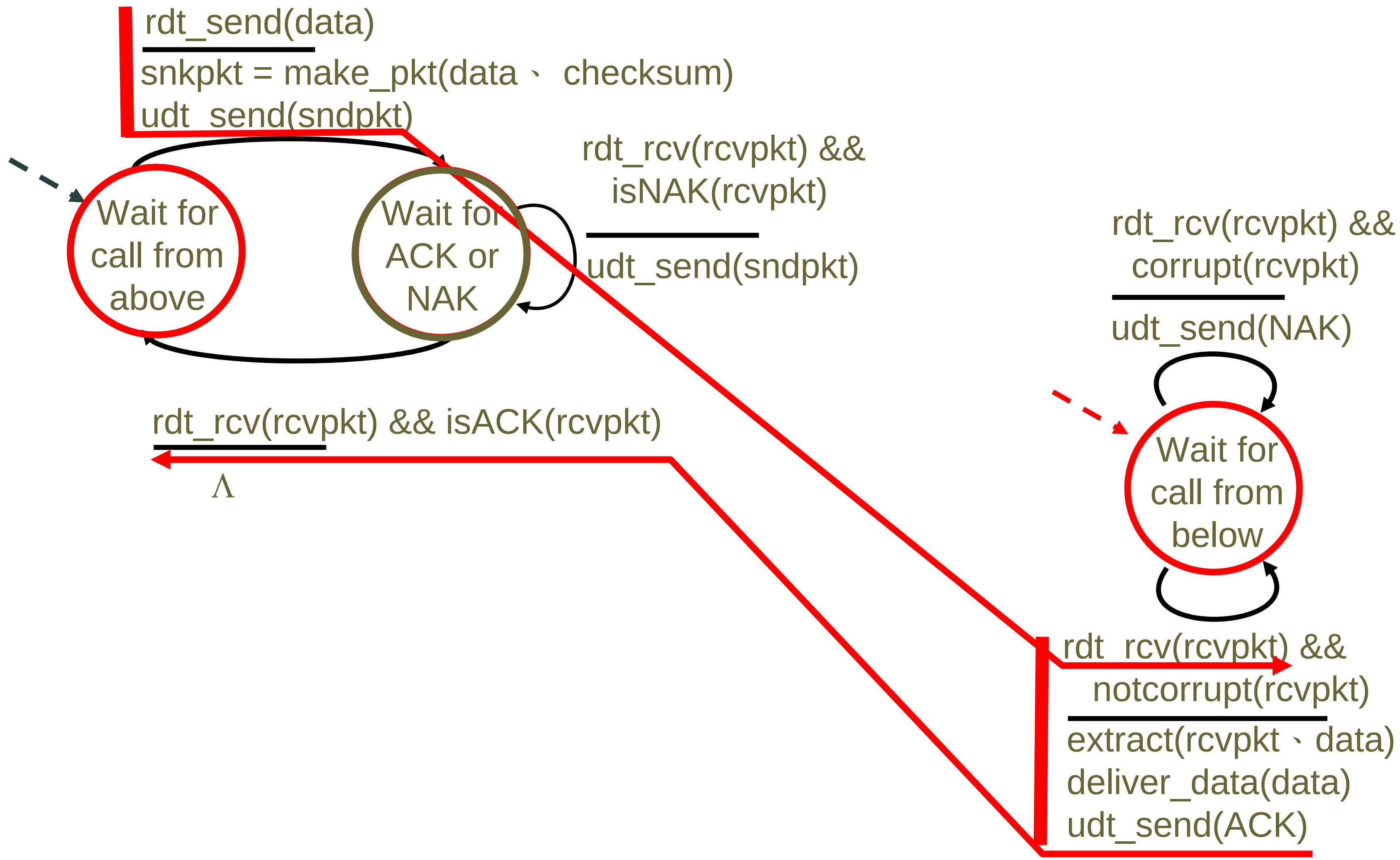
Rdt2.0： 可能產生位元錯誤的通道

- 底層的通道可能會將封包中的位元翻轉
  - 偵測位元錯誤的檢查和
- 問題： 如何回復錯誤：
  - 確認 (ACKs)： 接收端明確地告訴傳送端封包的傳送 OK
  - 否定確認 (NAKs)： 接收端明確地告訴傳送端封包的傳送有問題
  - 當收到NAK時、傳送端會重傳封包
- rdt2.0 的新機制 (超出rdt1.0)：
  - 錯誤偵測
  - 接收端回饋： 控制訊息 (ACK、NAK) 接收端->傳送端

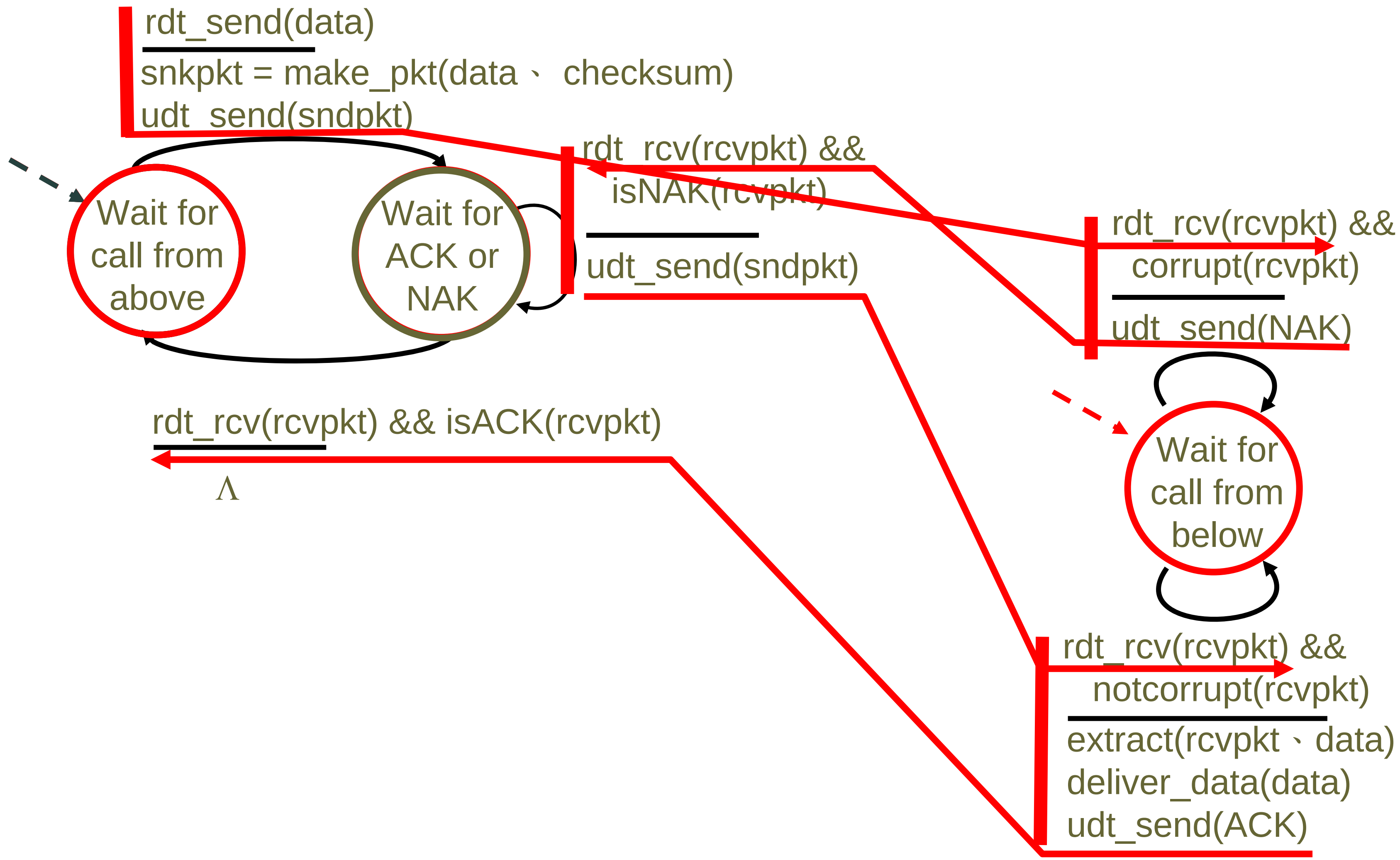
rdt2.0： FSM 說明


rdt_send(data)
snkpkt = make_pkt(data、 checksum)
udt_send(sndpkt)
等到從上層傳來的呼叫
等待ACK或者NAK訊息
rdt_rcv(rcvpkt) && isNAK(rcvpkt)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && isACK(rcvpkt)
Λ
傳送端
接收端
rdt_rcv(rcvpkt) && corrupt(rcvpkt)
udt_send(NAK)
等待從下層傳來的呼叫
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
extract(rcvpkt、data)
deliver_data(data)
udt_send(ACK)

rdt2.0： 沒有錯誤時的運作



rdt2.0： 發生錯誤的情況



rdt2.0 有一個致命的缺點!

假如 **ACK/NAK** 損毀了會如何?

- 傳送端不知道接收端發生了什麼事!
- 沒辦法直接重傳： 可能會重複

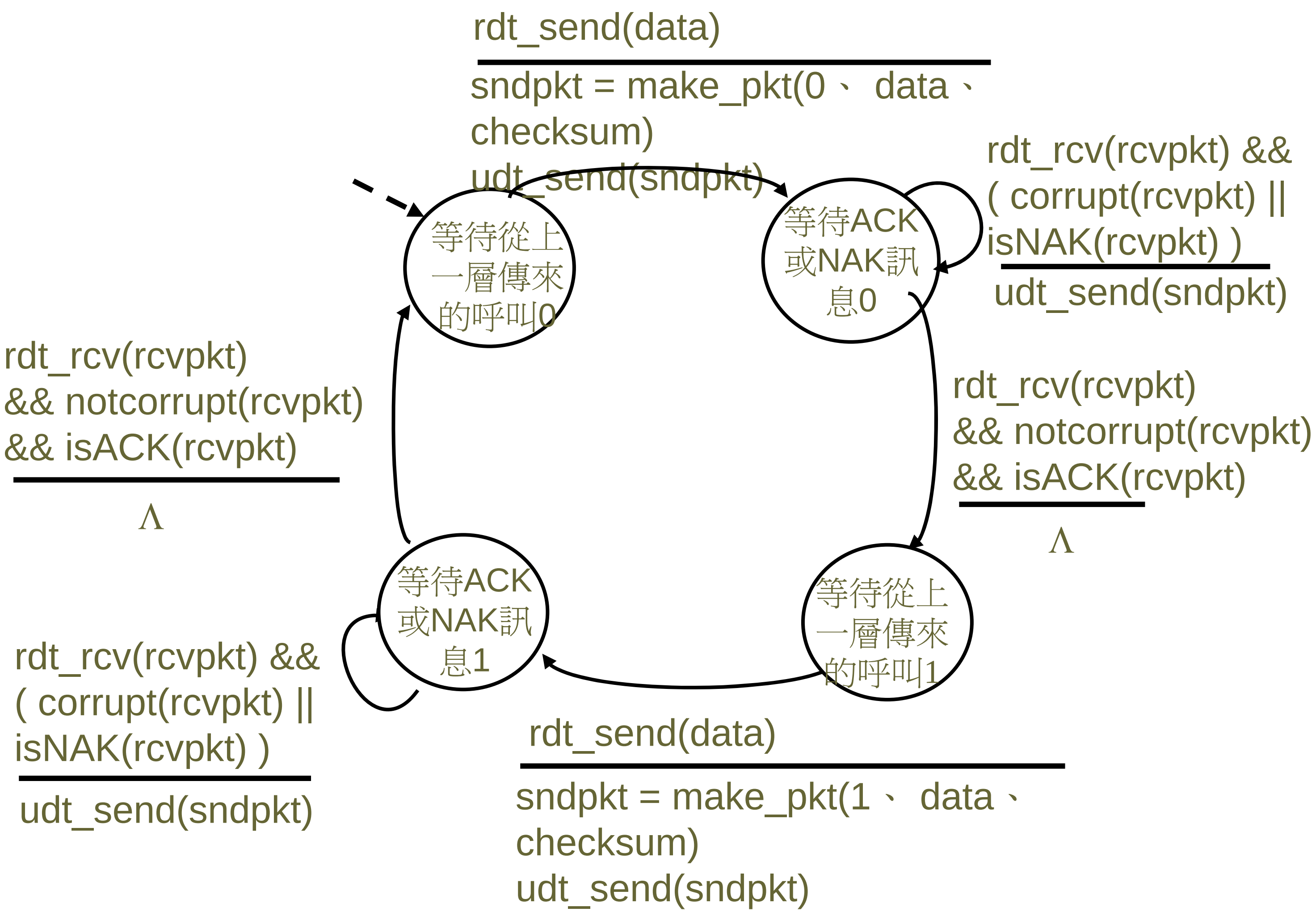
重複的處理：

- 假如 **ACK/NAK**損壞了、傳送端會重新傳送目前的封包
- 傳送端會在每個封包加上序號
- 接收端或刪掉 (不往上傳) 重複的封包

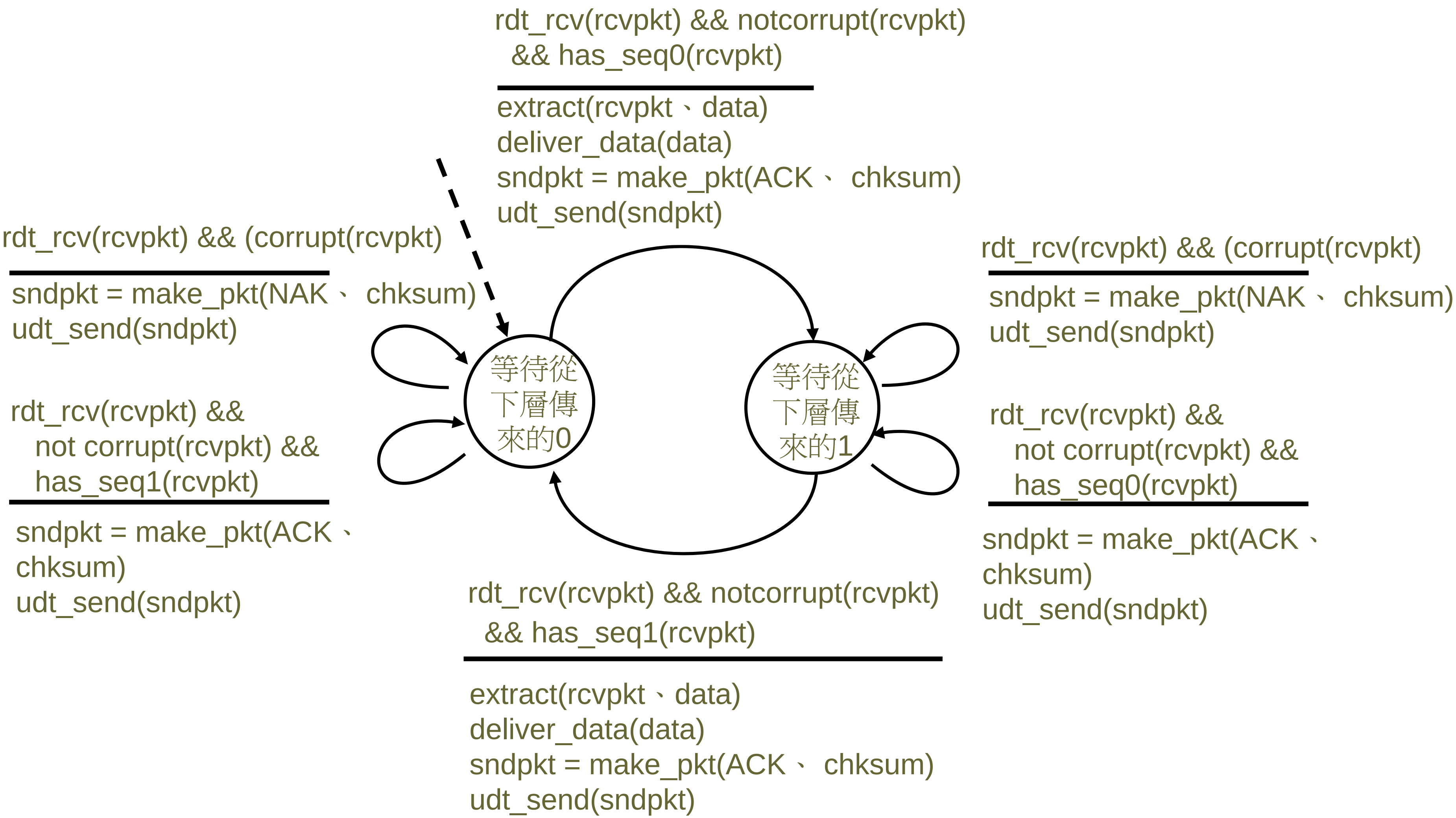
停止以及等待

傳送端傳送一個封包、並等待接收端的回應

rdt2.1： 傳送端、 處理損毀的 ACK/NAK



rdt2.1： 接收端、 處理損毀的 ACK/NAK



rdt2.1： 討論

傳送端：

- 在封包加入序號
- 兩個序號 (0、1) 就足夠了。為什麼？
- 必須檢查收到的 ACK/NAK 是否損毀
- 兩倍數量的狀態
  - 狀態必須"記得" "目前的"封包序號為 0 或是 1

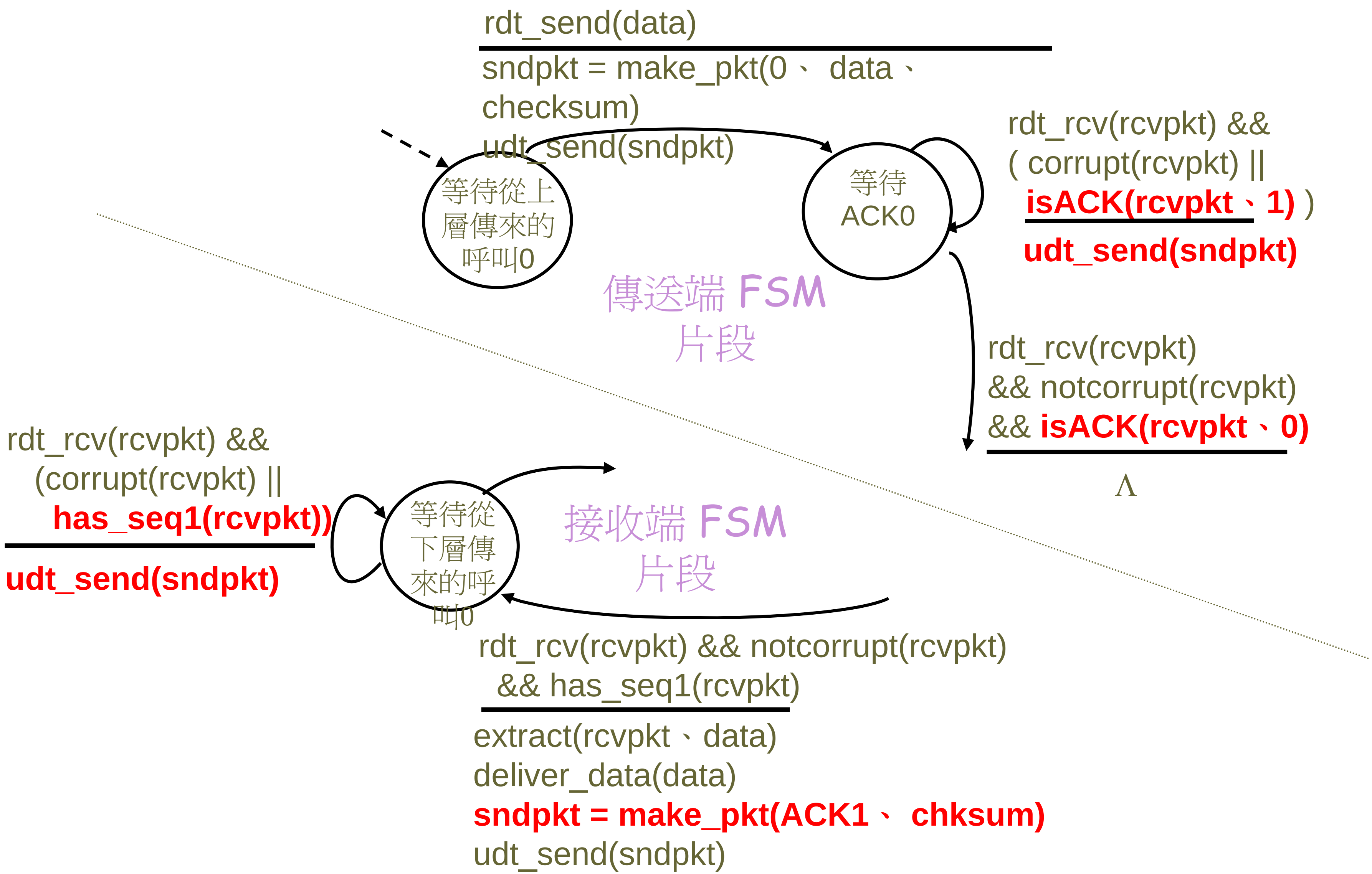
接收端：

- 必須確認接收端封包是否重複
  - 狀態表示 0 或 1 是否為所預期的封包序號
- 注意： 接收端無法得知它的最後一個 ACK/NAK 是否在傳送端被接收無誤

rdt2.2： 不採用NAK訊息的協定

- 與 **rdt2.1** 同樣的功能、但只使用**ACK**
- 不使用**NAK**、 接收端傳送 **ACK** 表示最後一個封包接收正確
  - 接收端必須明確地加上經過確認封包的序號
- 在傳送端收到重複的 **ACK** 導致與 **NAK** 相同的行爲：重新傳送目前的封包

rdt2.2： 傳送端、 接收端片段



rdt3.0： 使用會發生錯誤及遺失封包的通道

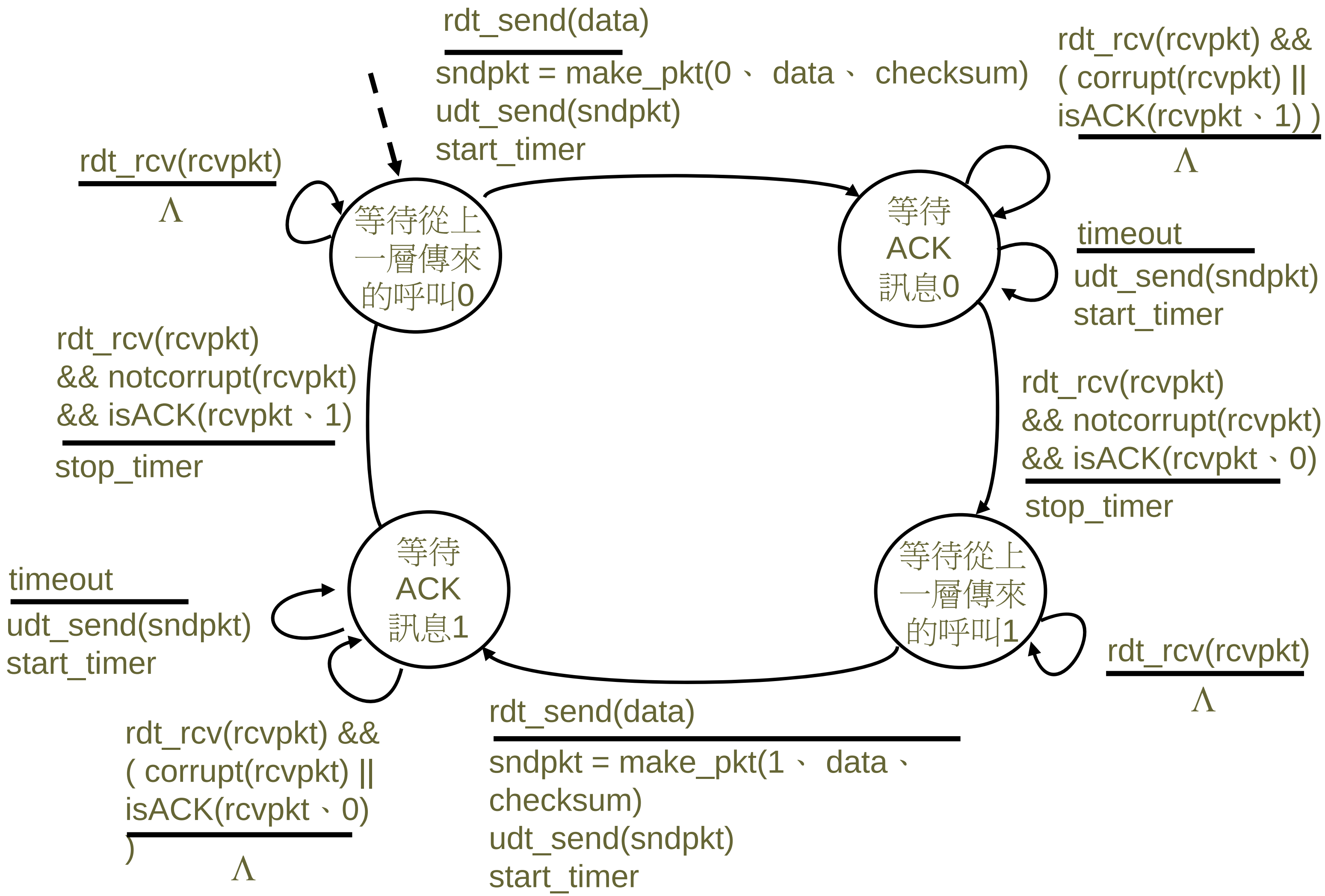
新的假設： 底層的頻道也可能遺失封包 (資料或 ACK)

- 檢查和、 序號、 ACK、 重傳都是有幫助的、但是卻不夠

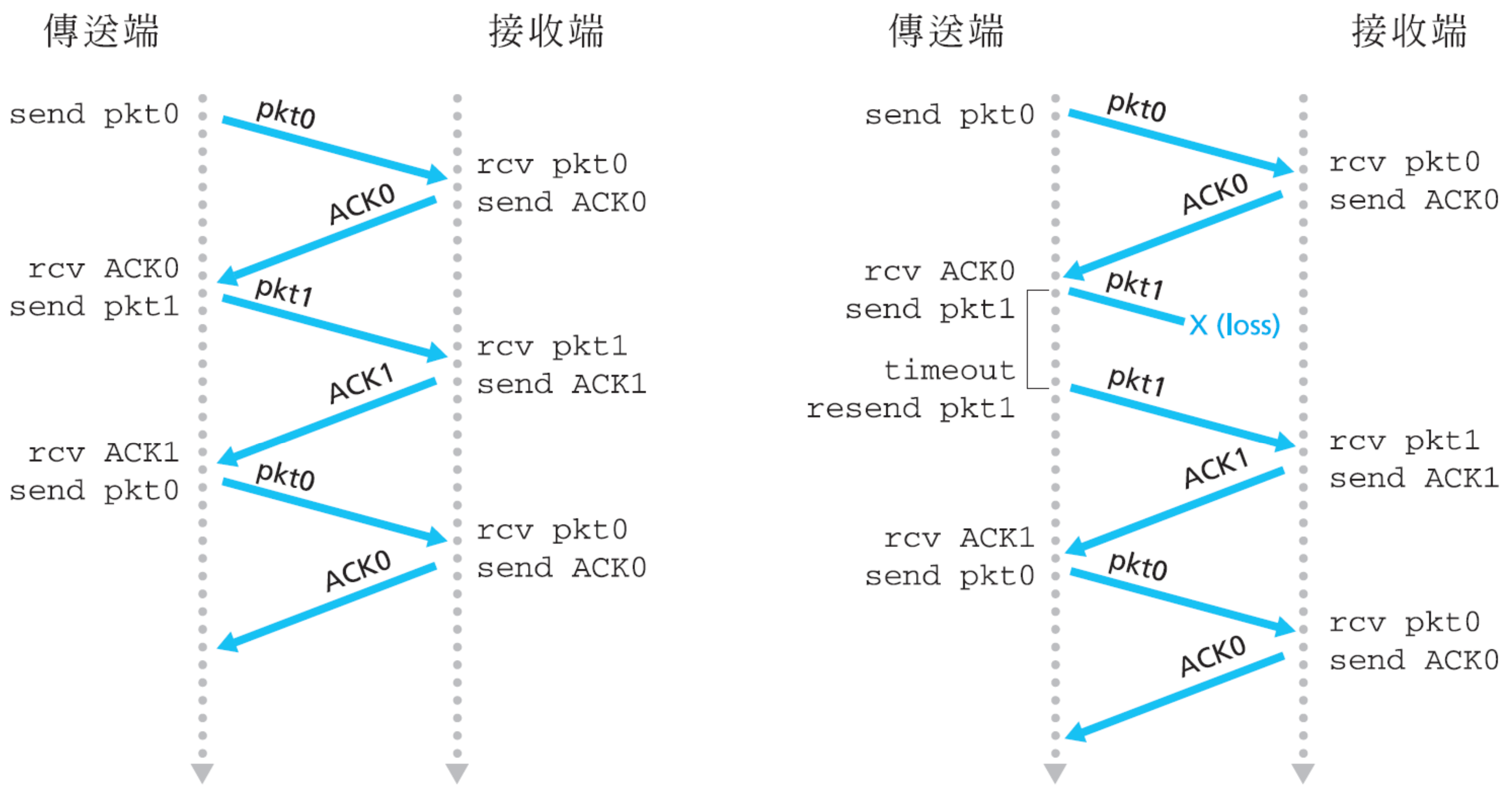
方法： 傳送端等待ACK “合理的” 時間

- 假如在這段時間內沒有收到 ACK、則重傳
- 假如封包 (或 ACK) 只是延遲了 (沒有遺失)：
  - 重傳會導致重複、但是序號的使用能夠處理這個情況
  - 接收端必須指定確認的封包序號
- 需要倒數計時器

rdt3.0 傳送端

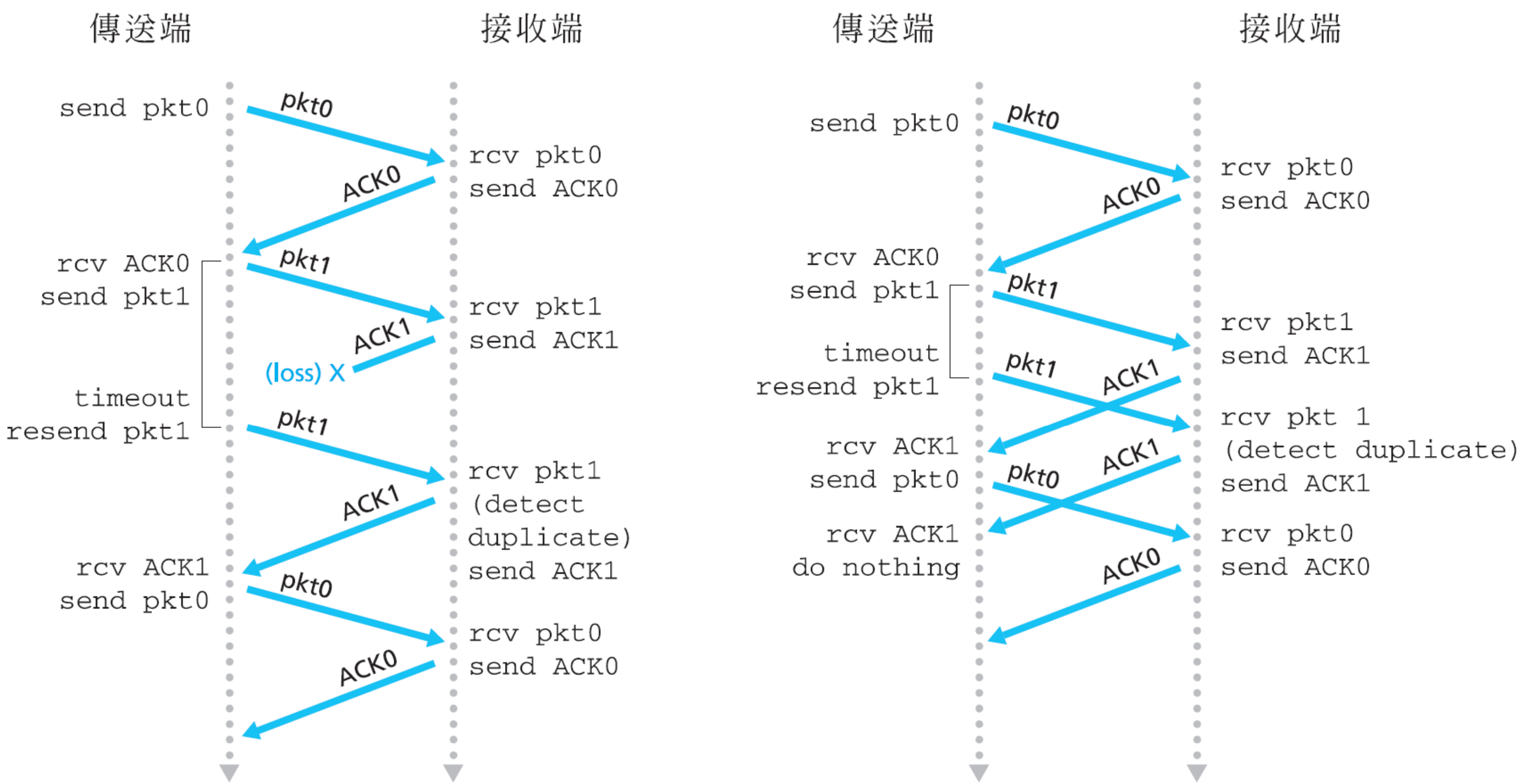

rdt_send(data)
sndpkt = make_pkt(0、 data、 checksum)
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isACK(rcvpkt、1))
Λ
rdt_rcv(rcvpkt)
Λ
等待從上一層傳來的呼叫0
等待ACK訊息0
timeout
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt、1)
stop_timer
rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt、0)
stop_timer
等待ACK訊息1
等待從上一層傳來的呼叫1
timeout
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt)
Λ
rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isACK(rcvpkt、0)
)
Λ
rdt_send(data)
sndpkt = make_pkt(1、 data、
checksum)
udt_send(sndpkt)
start_timer

rdt3.0 的運作



a. 沒有發生遺失的運作　**b.** 遺失封包

rdt3.0 的運作



c. 遺失ACK

d. 過早的逾時

rdt3.0的效能

- **rdt3.0** 能夠運作、 但是效能很糟
- 範例： **1 Gbps** 的連結、 **15** 毫秒 終端對終端傳遞延遲、 **1KB** 的封包：

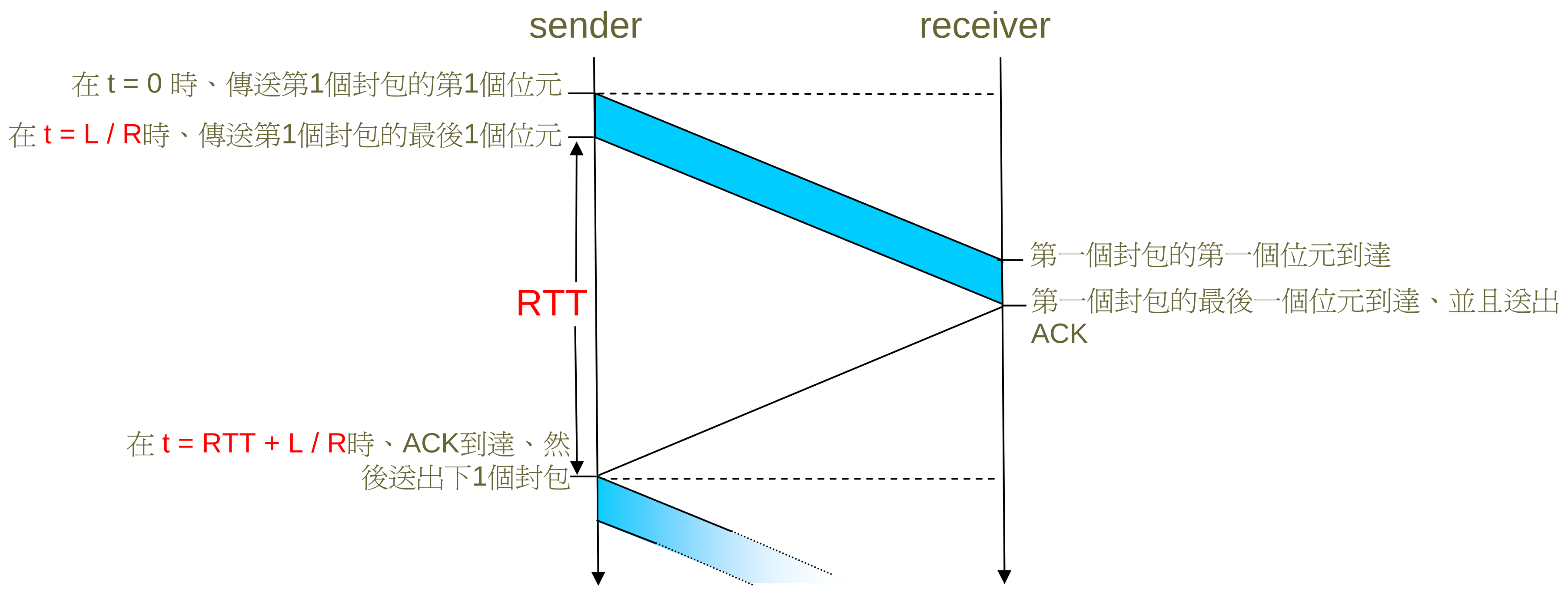
$$T_{transmit} = \frac{L\text{ (封包長度位元)}}{R\text{ (傳送速率、 bps)}} = \frac{8\text{kb/pkt}}{10^{**}9\text{ b/sec}} = 8\text{ 毫秒}$$

- U_{sender}： 使用率 – 傳送端將位元傳入通道的時間比例

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

- 每 30 毫秒 1KB 封包 -> 33kB/sec 生產量在 1 Gbps 連結上
- 網路協定限制了實體資源的使用!

rdt3.0： 停止並等待的機制

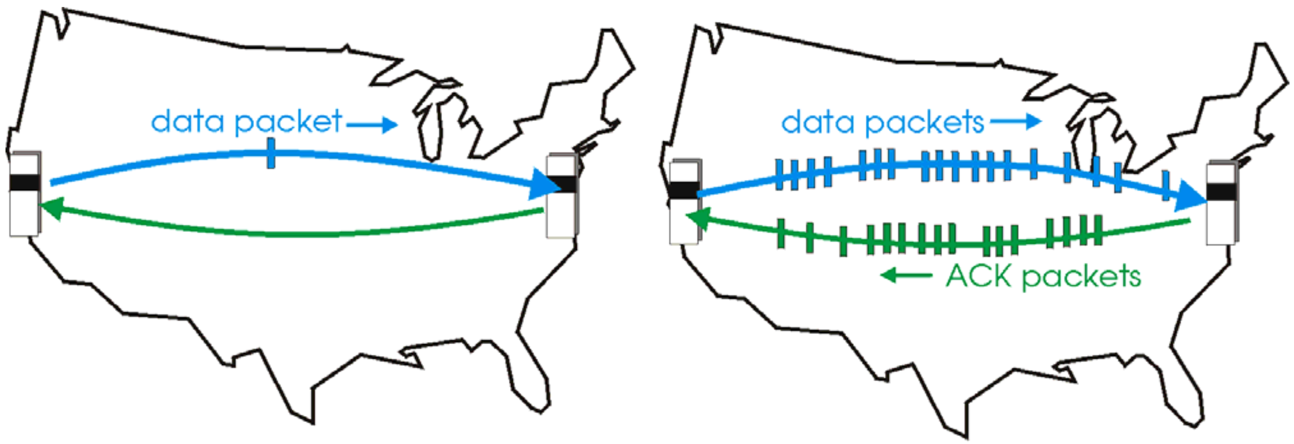


$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

Pipelined protocols管線化協定

Pipelining: **sender allows multiple, "in-flight", yet-to-be-acknowledged pkts**
管線化： 傳送端允許多個、"飛行中的"、 還沒有被確認的封包

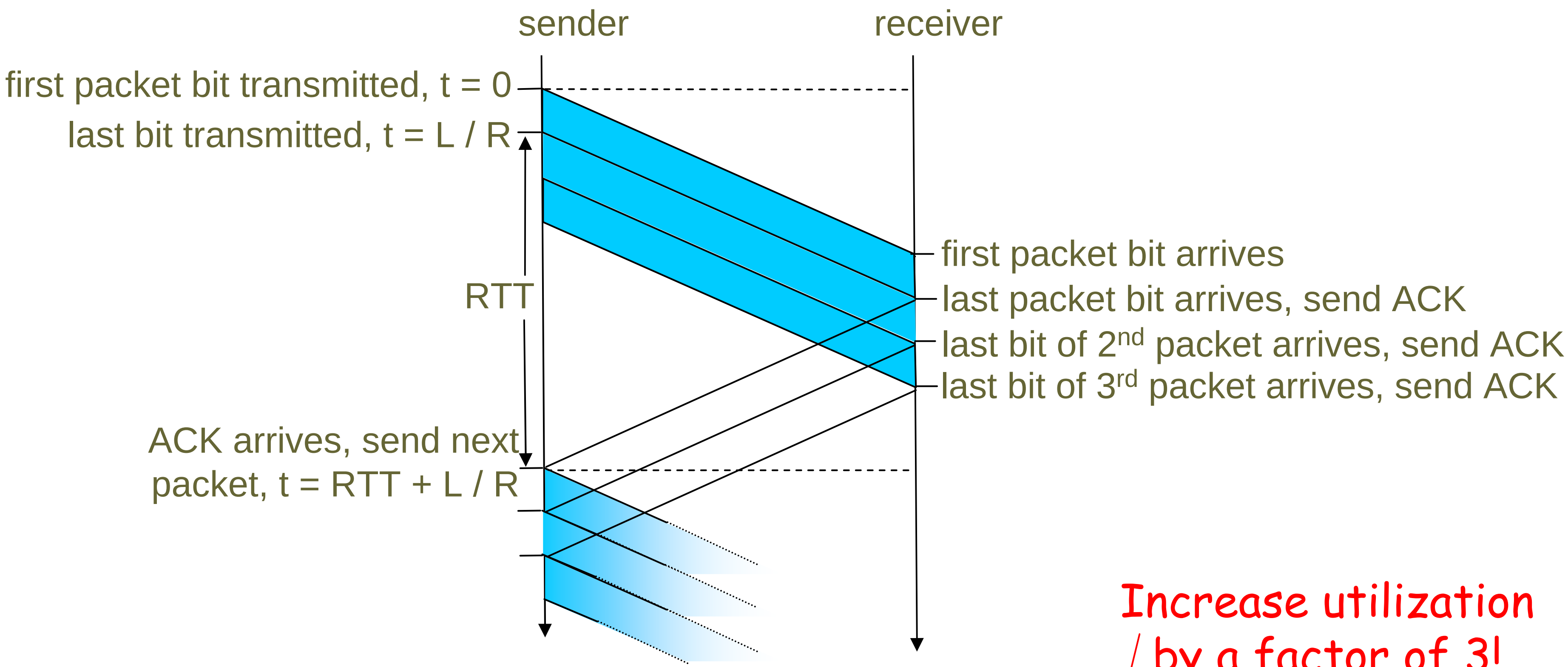
- range of sequence numbers must be increased序號的範圍必須增加
- buffering at sender and/or receiver傳送端 和/或 接收端需要暫存器



(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***
兩種管線化協定的一般性型態： 回送**N**、 選擇性重複

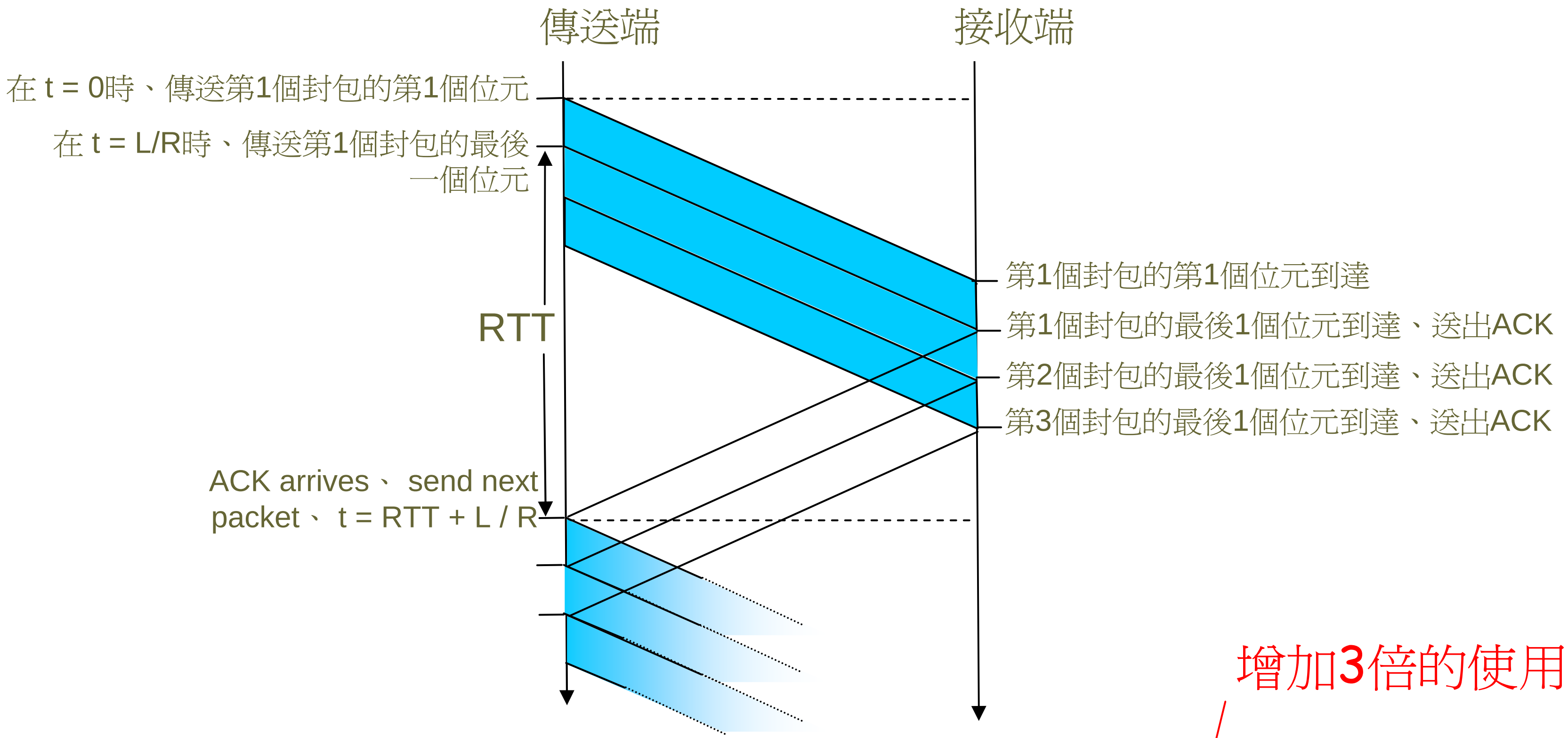
Pipelining: increased utilization
增加使用率



Increase utilization by a factor of 3!

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

管線化： 增加使用率



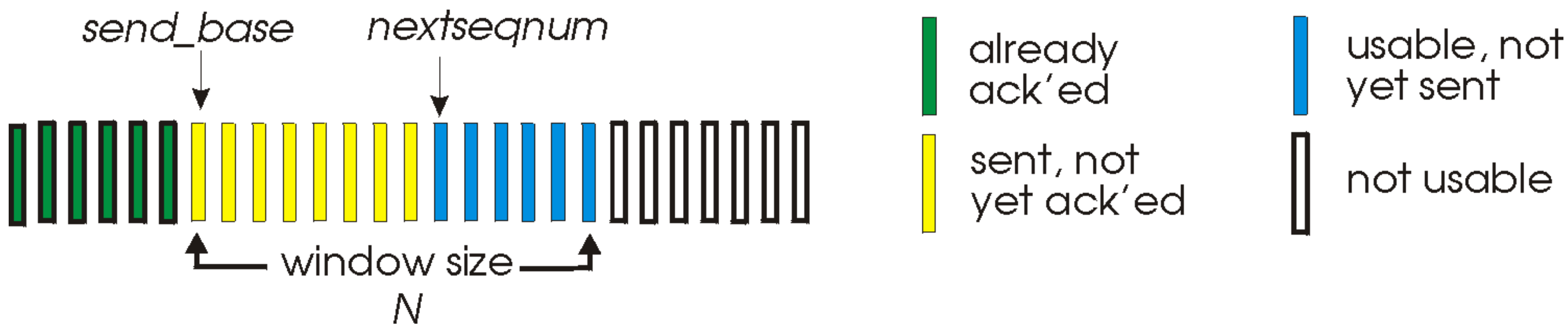
增加3倍的使用率!

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

Go-Back-N 回送N

Sender:

- **k-bit seq # in pkt header**
  封包標頭的 **k-**位元序號
- **"window" of up to N, consecutive unack'ed pkts allowed**
  大小最多爲**N**的"視窗"、允許連續的未被確認的封包

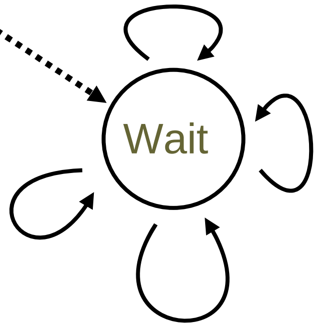


- **ACK(n): ACKs all pkts up to, including seq # n - "cumulative ACK"**
  ACK(n)： 確認小於或等於序號 **n** 的所有封包 **- "**累積式確認**"**
  - May receive duplicate ACKs (see receiver)
    可能會收到重複的確認 (見接收端)
- **Timer for each in-flight pkt**
  某個傳送中的封包都使用一個計時器
- ***timeout(n):*** **retransmit pkt n and all higher seq # pkts in window**
  重傳封包 **n** 以及在視窗中序號高於 **n** 的全部封包

GBN: sender extended FSM

```
rdt_send(data)
____________________
if (nextseqnum < base+N) {
    sndpkt[nextseqnum] = make_pkt(nextseqnum,data,chksum)
    udt_send(sndpkt[nextseqnum])
    if (base == nextseqnum)
       start_timer
    nextseqnum++
    }
else
  refuse_data(data)
```

```
Λ
__________
base=1
nextseqnum=1
```

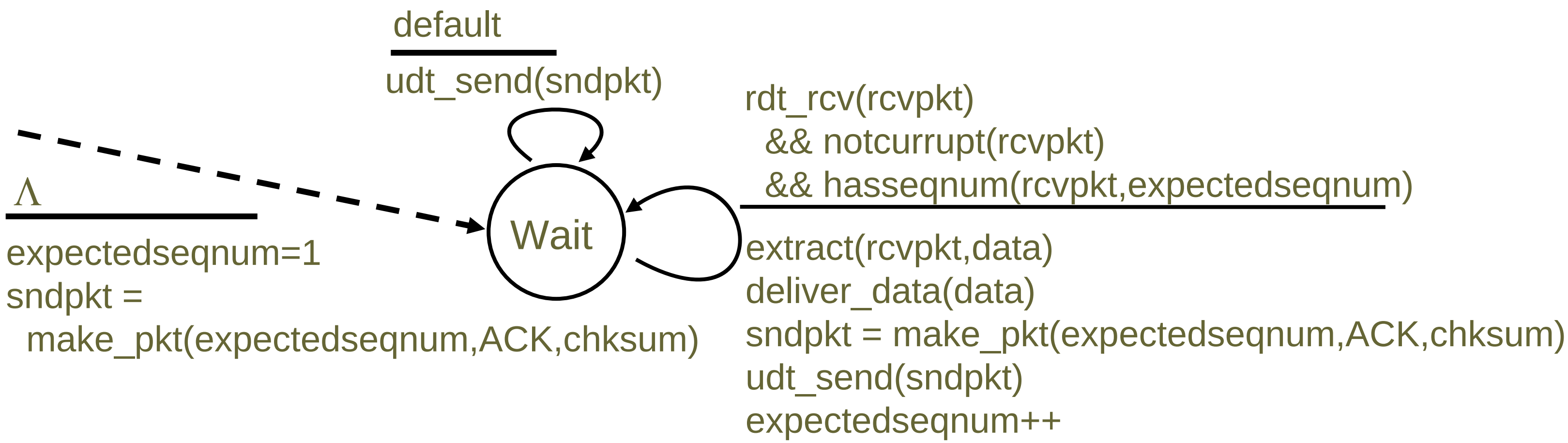



```
timeout
____________________
start_timer
udt_send(sndpkt[base])
udt_send(sndpkt[base+1])
…
udt_send(sndpkt[nextseqnum-1])
```

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

```
rdt_rcv(rcvpkt) &&
   notcorrupt(rcvpkt)
____________________
base = getacknum(rcvpkt)+1
If (base == nextseqnum)
    stop_timer
  else
    start_timer
```

GBN: receiver extended FSM

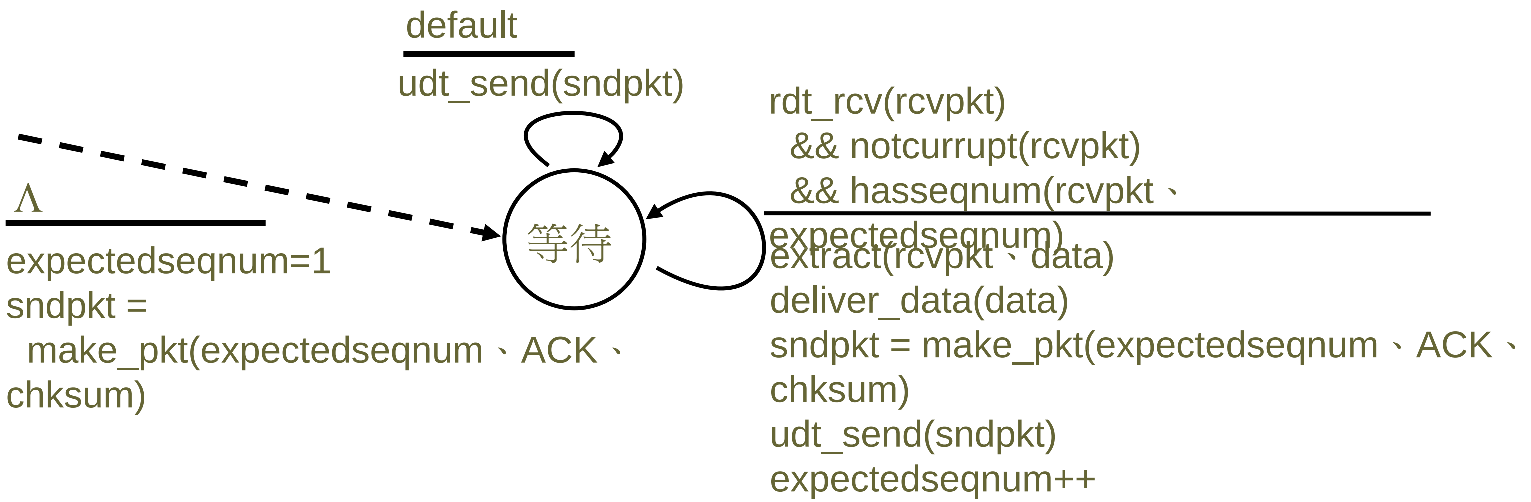


ACK-only: always send ACK for correctly-received pkt with highest *in-order* seq #

- may generate duplicate ACKs
- need only remember **`expectedseqnum`**

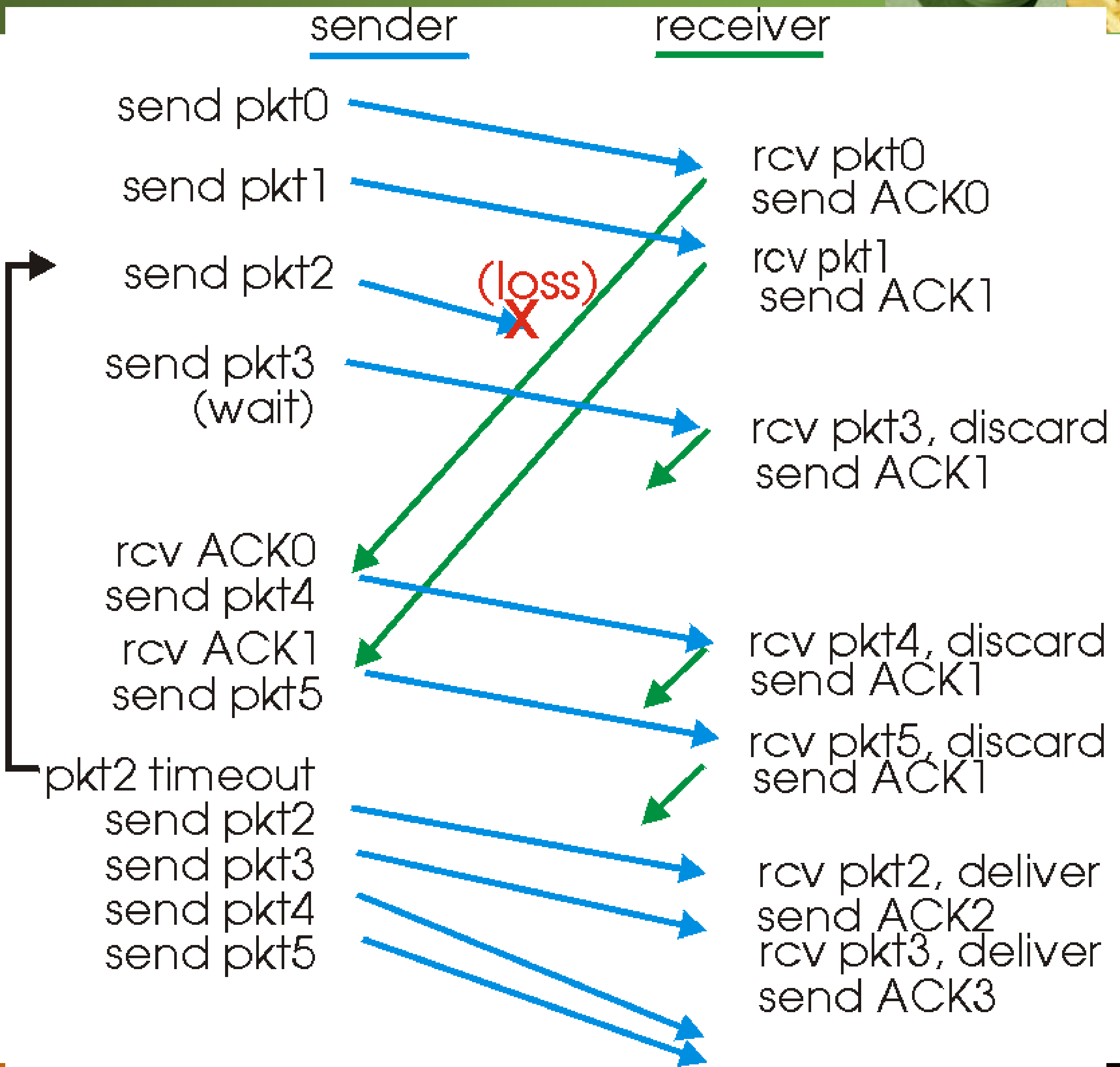
- **out-of-order pkt:**
  - discard (don't buffer) -> no receiver buffering!
  - Re-ACK pkt with highest in-order seq #

GBN： 接收端的擴充 FSM



- 只使用**ACK**： 只為接收順序正確的封包傳送 **ACK**
  - 可能會產生重複的ACK
  - 只需要記住 **expectedseqnum**
- 順序不正確的封包：
  - 刪除 (不會暫存) -> 接收端沒有暫存器!
  - 重新回應最高的順序正確封包

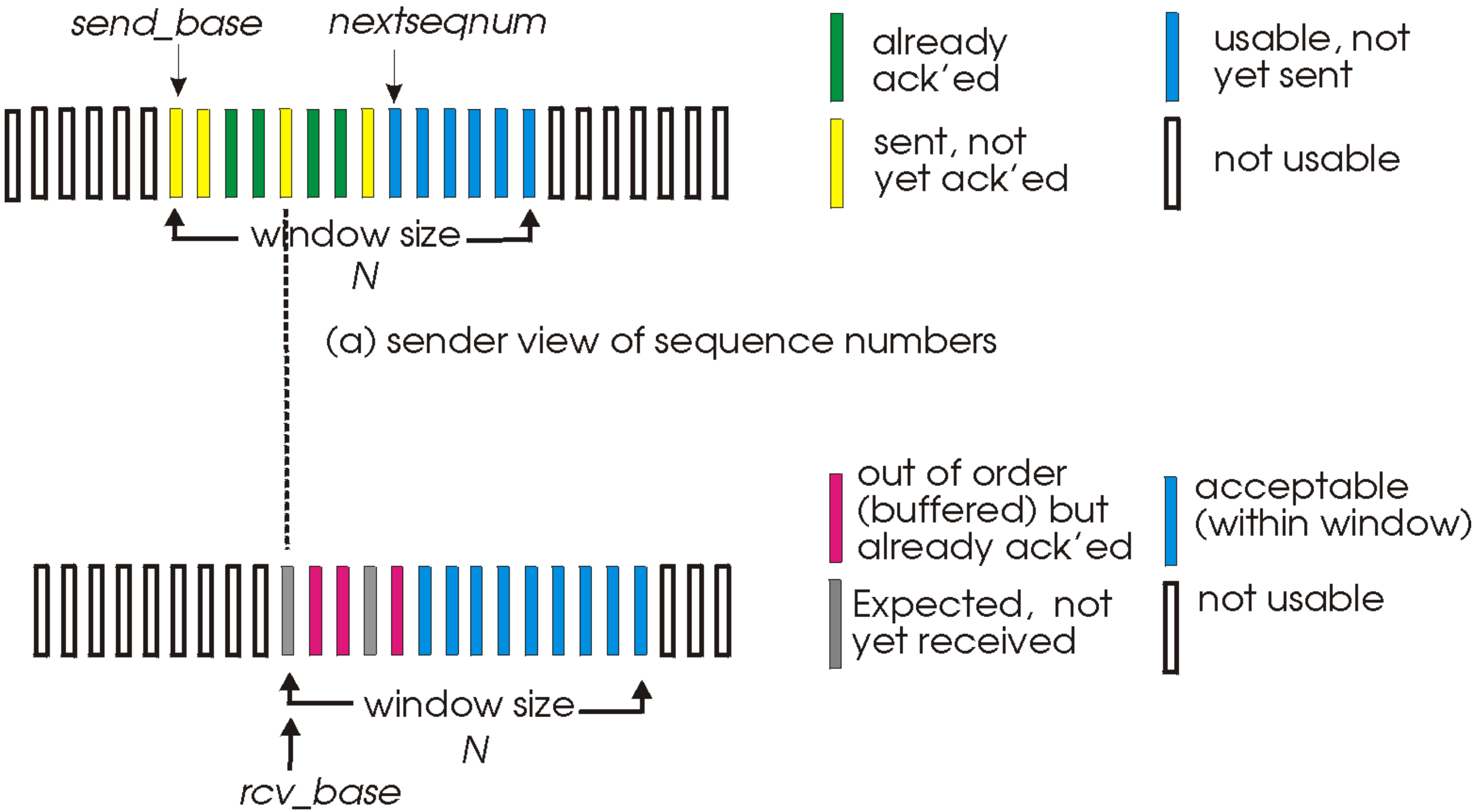
GBN in action



Selective Repeat選擇性重複

- **Receiver *individually* acknowledges all correctly received pkts**
  接收端分別確認所有正確接收的封包
  - Buffers pkts, as needed, for eventual in-order delivery to upper layer
    依需要暫存封包、 最終會依序傳送到上一層
- **Sender only resends pkts for which ACK not received**
  傳送端只重傳沒有收到 ACK 的封包
  - Sender timer for each unACKed pkt
    傳送端針對每一個未確認的封包需要一個計時器
- **Sender window傳送端視窗**
  - N consecutive seq #'s　N 個連續的序號
  - Again limits seq #s of sent, unACKed pkts
    再次、用來限制傳送出去的、 未確認的封包序號

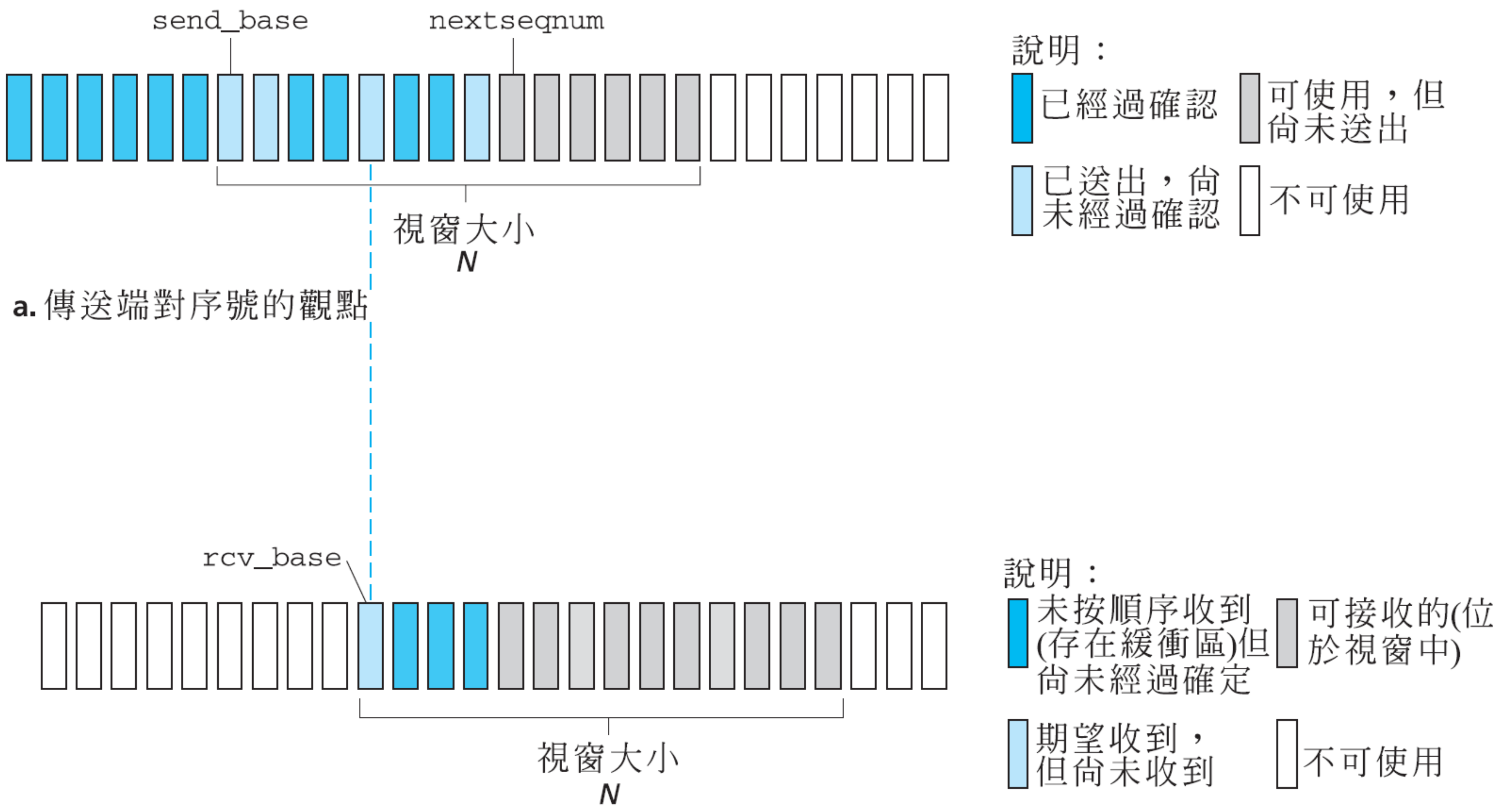
Selective repeat: sender, receiver windows



(a) sender view of sequence numbers

(b) receiver view of sequence numbers

選擇性重複： 傳送端、 接收端視窗



a. 傳送端對序號的觀點

b. 接收端對序號的觀點

Selective repeat

sender

Data from above :

- If next available seq # in window, send pkt

Timeout(n):

- Resend pkt n, restart timer

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

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

- ACK(n)

otherwise:

- Ignore

選擇性重複

傳送端

- 來自上層的資料 ：
- 假如下一個可用的序號在視窗內、則傳送封包
- **timeout(n)**：
- 重送封包 **n**、 重新啓動計時器
- **ACK(n)** 在 **[sendbase**、**sendbase+N]**中：
- 將封包 **n** 標示爲已收到的
- 假如 **n** 爲未確認的封包中最小的、將視窗的 **base** 往前移到下一個未回應的序號

接收端

封包**n** 在 **[rcvbase**、**rcvbase+N-1]**中

- 傳送 **ACK(n)**
- 不正確的順序： 暫存區
- 正確順序： 遞送 (也遞送暫存區內順序錯誤的封包)、將視窗前進到下一個未接收的封包

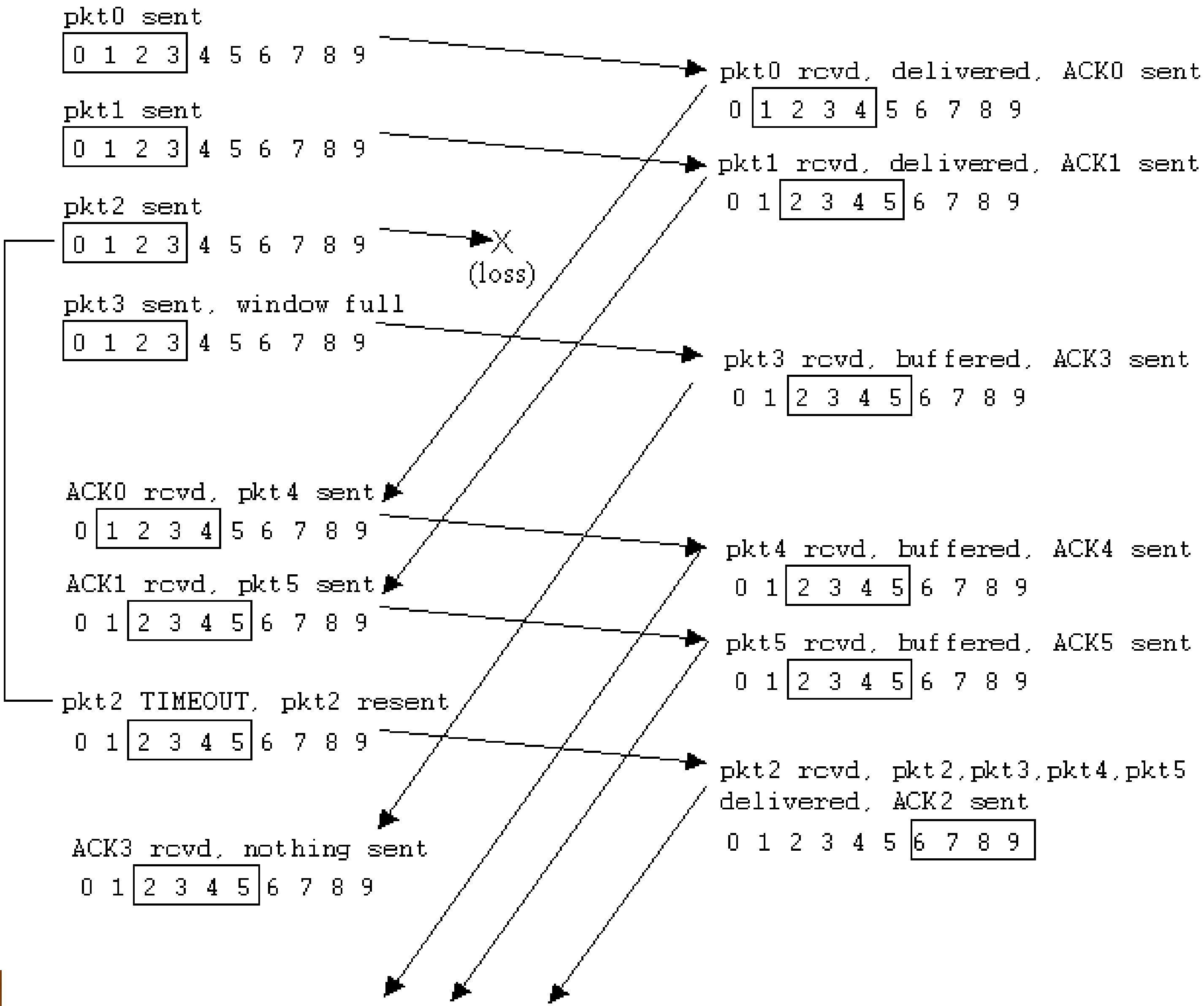
封包 **n** 在 **[rcvbase-N**、**rcvbase-1]**中

- **ACK(n)**

否則：

- 忽略該封包

Selective repeat in action

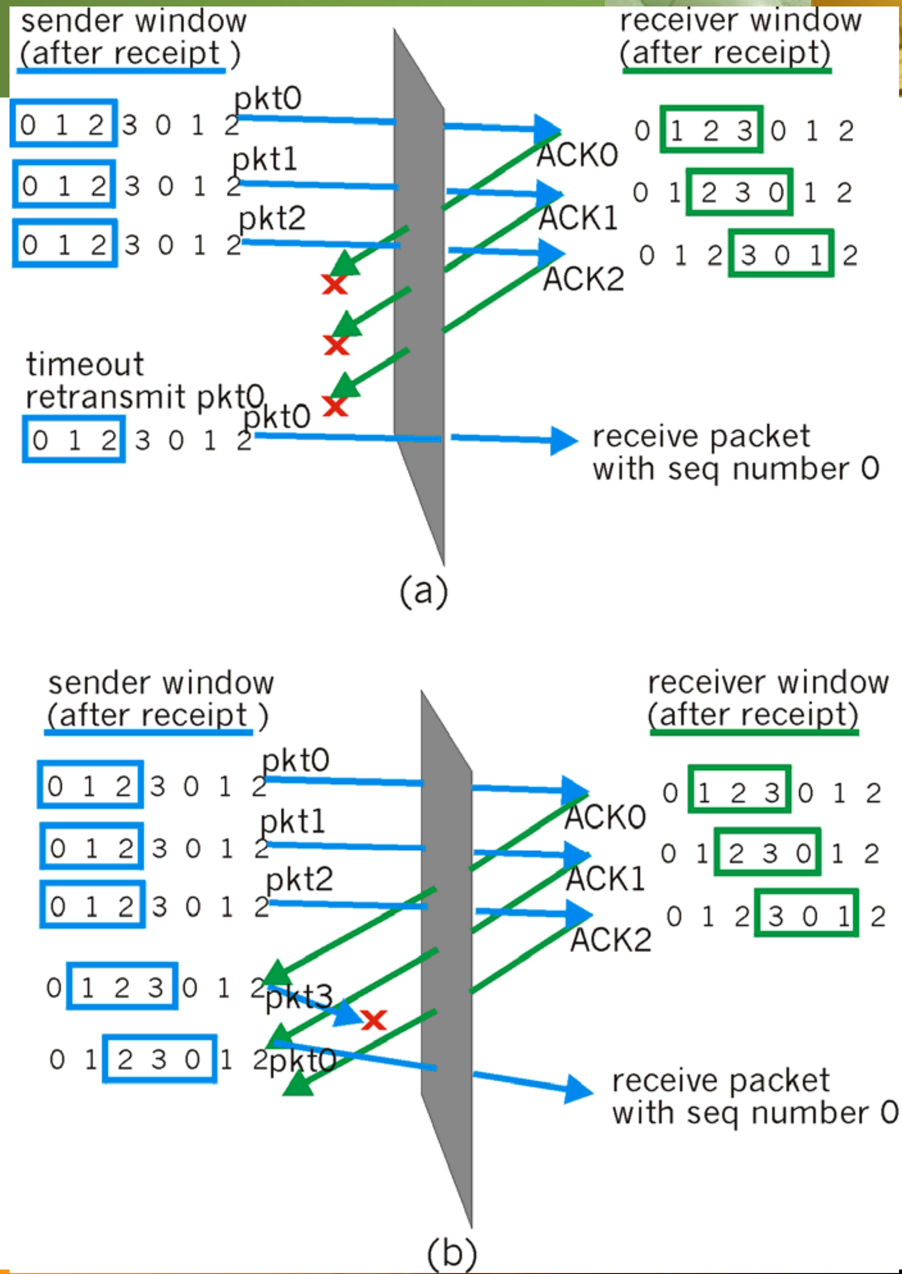


Selective repeat: dilemma
選擇性重複： 困境

Example:

- Seq #'s: 0, 1, 2, 3
- Window size=3

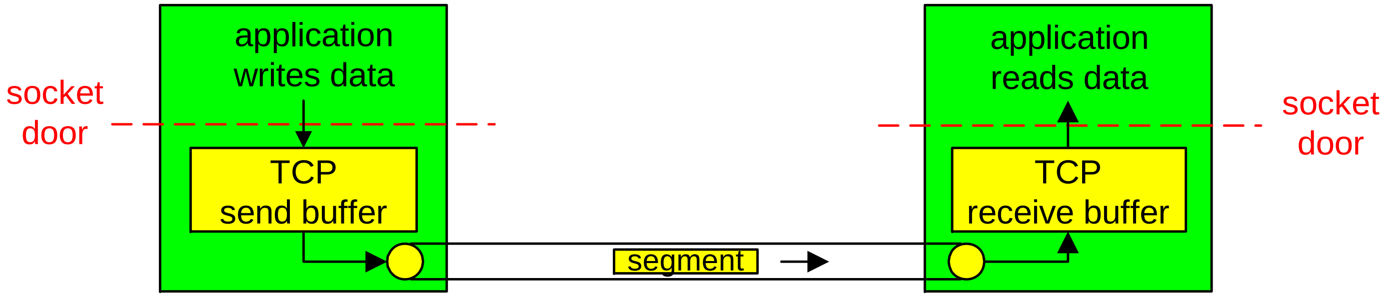
- Receiver sees no difference in two scenarios!接收端無法分辨兩種情況的差別
- Incorrectly passes duplicate data as new in (a)
  不正確地重新傳送重複的資料、如同 (a)



3.5 Connection-oriented transport: TCP 連線導向傳輸

TCP: Overview RFCs: 793, 1122, 1323, 2018, 2581

- **Point-to-point:**點對點
  - One sender, one receiver一個傳送端、 一個接收端
- **Reliable, in-order *byte steam:*** 可靠的、 有順序的位元組串流
  - No "message boundaries"沒有 "訊息界線"
- **Pipelined:**管線化
  - TCP congestion and flow control set window size TCP壅塞控制和流量控制設定視窗大小
- ***Send & receive buffers***傳送端和接收端暫存器



- **Full duplex data:**全雙工資料傳輸
  - Bi-directional data flow in same connection
    同一個連結中、雙向的資料流
  - MSS: maximum segment size最大資料分段大小
- **Connection-oriented:**連線導向
  - Handshaking (exchange of control msgs) init's sender, receiver state before data exchange
    交握程序 (控制訊息的交換) 在資料開始交換之前、設定傳送端和接收端的狀態
- **Flow controlled:**流量控制
  - Sender will not overwhelm receiver
    傳送端不會超過接收端

TCP segment structure

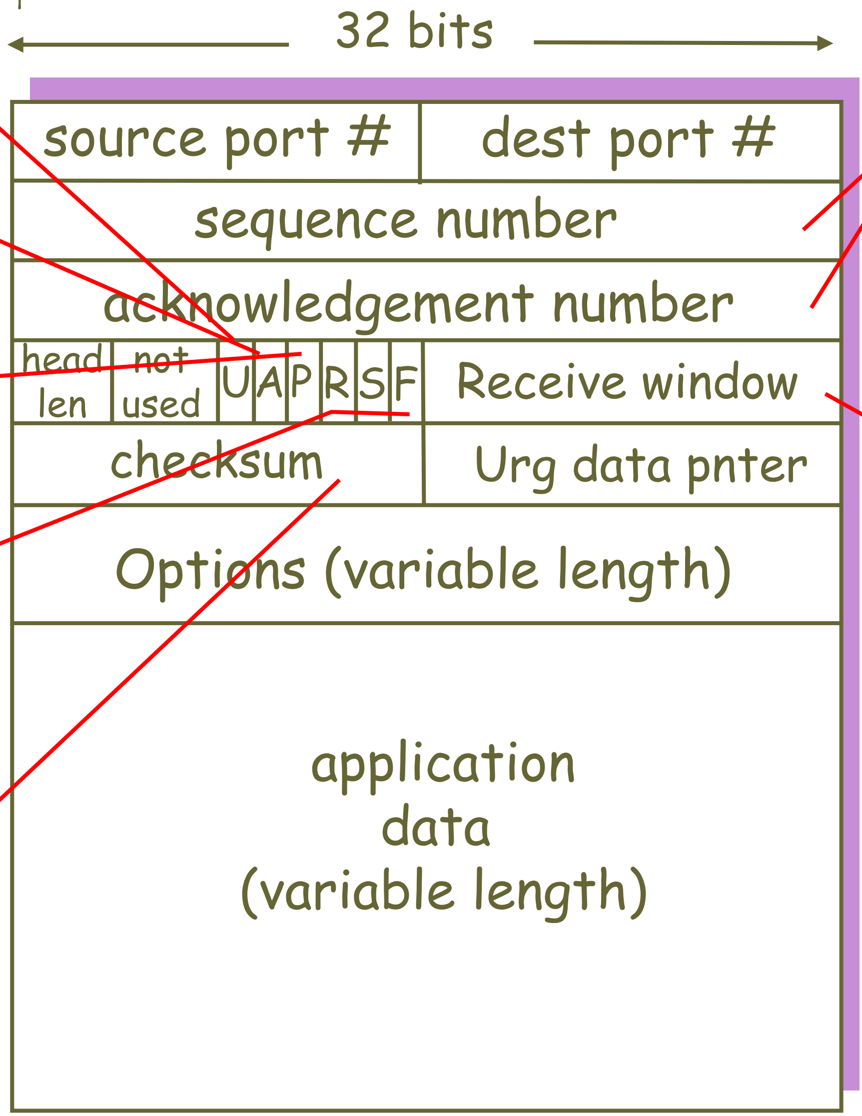
URG: urgent data緊急資料
(generally not used)

ACK: ACK #
valid

PSH: push data now
馬上將資料送出
(generally not used)

RST, SYN, FIN:
connection estab連線建立(setup設定, teardown中斷, commands指令)

Internet checksum
網際網路檢查和
(as in UDP)



Counting by
bytes of data以資料位元組計算
(not segments
非資料分段)

bytes
rcvr willing
to accept
接收端願意
接收的位元組數

TCP seq. #'s and ACKs

Seq. #'s:

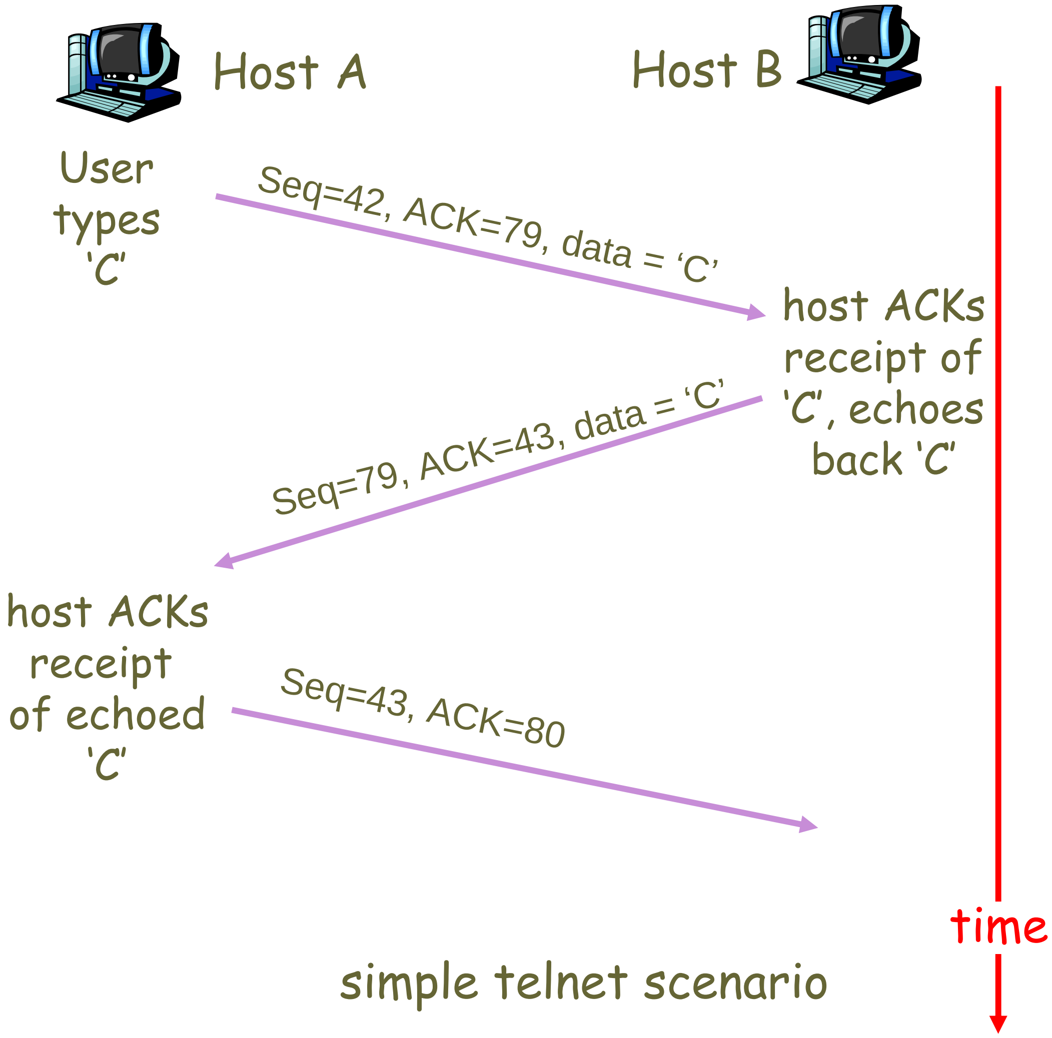
- byte stream "number" of first byte in segment's data

ACKs:

- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

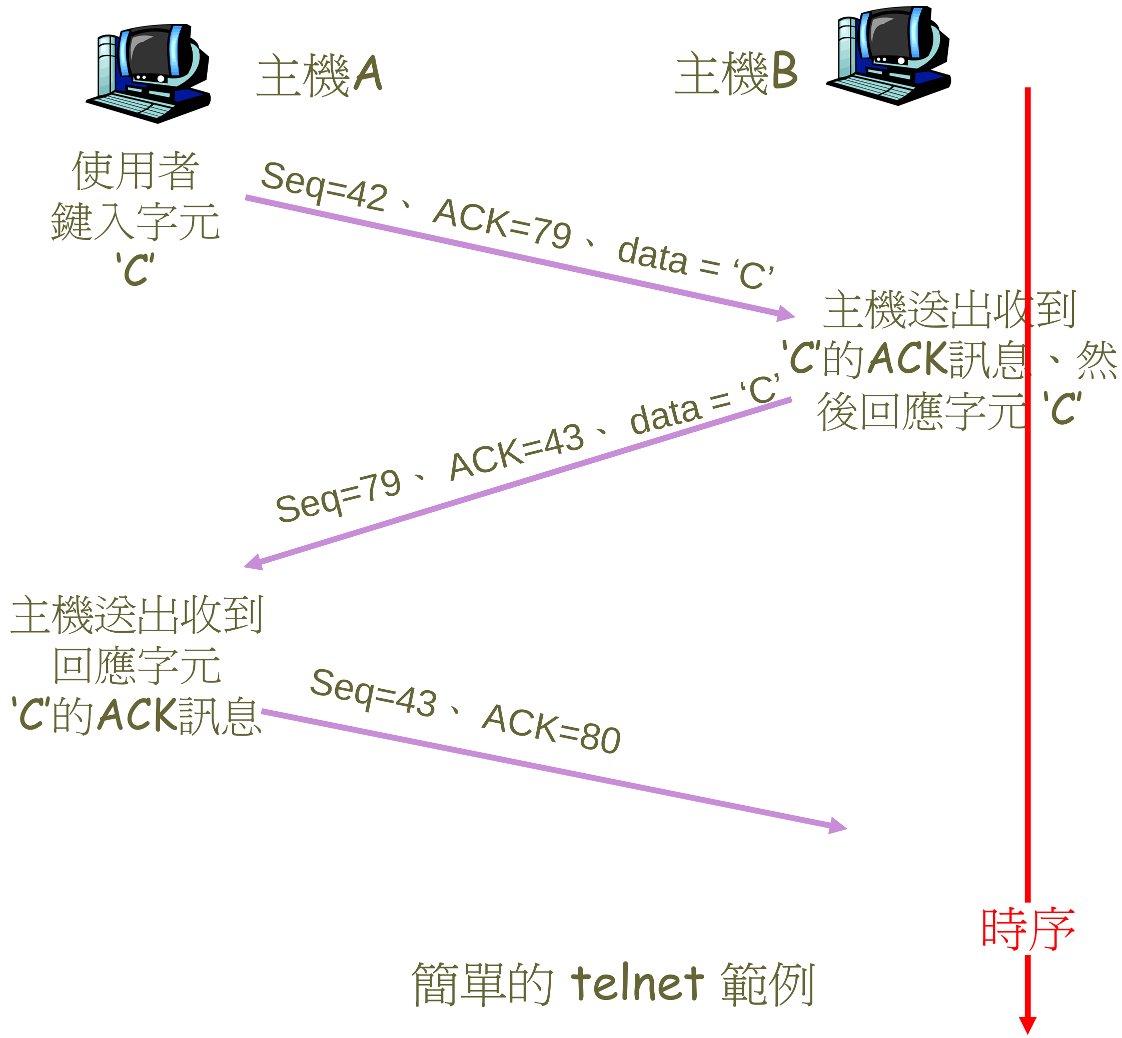
- A: TCP spec doesn't say, - up to implementor



simple telnet scenario

TCP 序號與確認

- 序號：
  - 資料分段中、第一個位元的位元組串流 "編號"
- 確認：
  - 另一端期待的下一個位元組序號
  - 累積式確認
- 問題： 接收端如何處理順序不正確的資料分段
  - 答： TCP 規格中未限制、取決於程式開發者



簡單的 telnet 範例

TCP Round Trip Time and Timeout

Q: how to set TCP timeout value?

- longer than RTT
  - but RTT varies
- too short: premature timeout
  - unnecessary retransmissions
- too long: slow reaction to segment loss

Q: how to estimate RTT?

- `SampleRTT`: measured time from segment transmission until ACK receipt
  - ignore retransmissions
- `SampleRTT` will vary, want estimated RTT "smoother"
  - average several recent measurements, not just current **`SampleRTT`**

TCP 來回傳遞時間以及逾時

- 問： 如何設定 TCP 的逾時值?
- 比 RTT 長
  - 但是 RTT 是不固定的
- 太短： 過早逾時
  - 不需要重新傳送
- 太長： 太晚對資料分段遺失作出反應

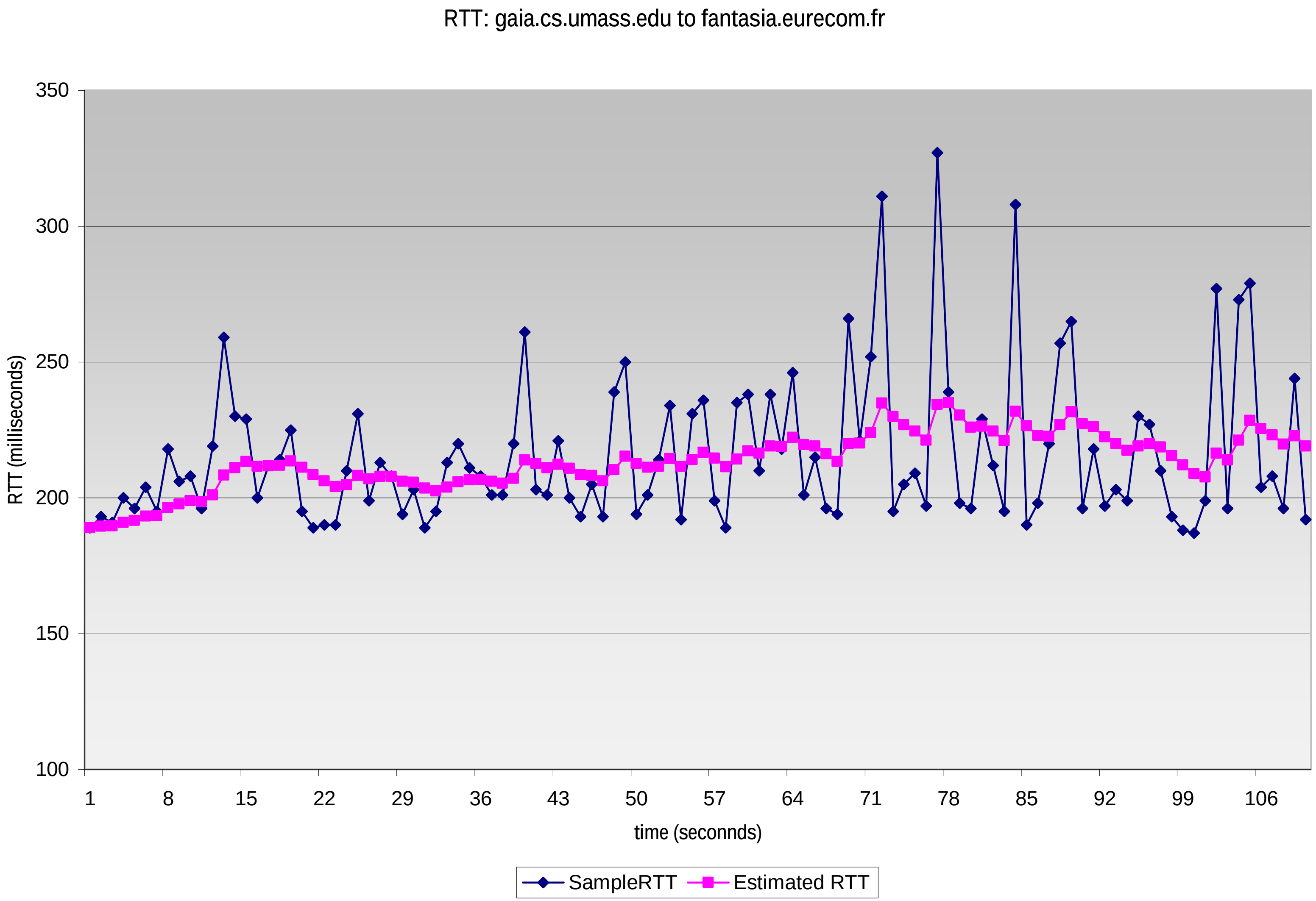
- 問： 如何估計來回傳遞時間
- (RTT)?
- 樣本RTT： 測量資料分段傳送出去到收到確認所需的時間
  - 忽略重傳
- 樣本**RTT**會有所變動、我們想要讓預估的**RTT** “更平滑”
  - 將好幾個最近的測量值做平均、而非目前的樣本**RTT**

TCP Round Trip Time and Timeout

`EstimatedRTT = (1- α)*EstimatedRTT + α*SampleRTT`

- **Exponential weighted moving average**
  指數加權移動平均值
- **Influence of past sample decreases exponentially fast**
  過去樣本的影響將以指數速率減少
- **Typical value建議值:** α = **0.125**

Example RTT estimation:



TCP Round Trip Time and Timeout

Setting the timeout

- `EstimtedRTT` **plus "safety margin"**
  - large variation in **`EstimatedRTT`** **->** larger safety margin
- **First estimate of how much SampleRTT deviates from EstimatedRTT:**

```
DevRTT = (1-β)*DevRTT +
             β*|SampleRTT-EstimatedRTT|

(typically, β = 0.25)
```

Then set timeout interval:

```
TimeoutInterval = EstimatedRTT + 4*DevRTT
```

TCP來回傳遞時間以及逾時

設定逾時間隔

- EstimtedRTT 加上 “安全邊界”
  - **EstimatedRTT** 的變動很大 **->** 大的安全邊界
- 首先估計 **SampleRTT** 與 **EstimatedRTT** 的差距：

$$DevRTT = (1-\beta)*DevRTT + \beta*|SampleRTT-EstimatedRTT|$$

(通常、 β = 0.25)

接著設定逾時間隔：

$$TimeoutInterval = EstimatedRTT + 4*DevRTT$$

TCP reliable data transfer

- **TCP creates rdt service on top of IP's unreliable service**
  TCP 在 **IP** 的不可靠服務上建立 **rdt** 服務
- **Pipelined segments**
  管線化的分段
- **Cumulative acks**
  累積式確認
- **TCP uses single retransmission timer TCP**
  使用單一的重新傳送計時器
- **Retransmissions are triggered by:**
  重新傳送的觸發
  - Timeout events逾時事件
  - Duplicate acks重複的ack
- **Initially consider simplified TCP sender:**
  一開始先考慮簡化的**TCP**傳送端
  - Ignore duplicate acks
    忽略重複的ack
  - Ignore flow control, congestion control
    忽略流量控制、壅塞控制

TCP sender events:

data rcvd from app:

- Create segment with seq #
- seq # is byte-stream number of first data byte in segment
- start timer if not already running (think of timer as for oldest unacked segment)
- expiration interval: `TimeOutInterval`

timeout:

- retransmit segment that caused timeout
- restart timer

Ack rcvd:

- If acknowledges previously unacked segments
  - update what is known to be acked
  - start timer if there are outstanding segments

TCP 傳送端事件：

從應用程式收到資料：

- 產生含有序號的資料分段
- 序號是資料分段中、第一個資料位元組的位元組串流編號
- 假如計時器尚未執行、啓動計時器 (將計時器想成與最久的未確認資料分段有關)
- 逾時時間：**TimeOutInterval**

逾時：

- 傳新傳送導致逾時的資料分段
- 重新啓動計時器

收到Ack：

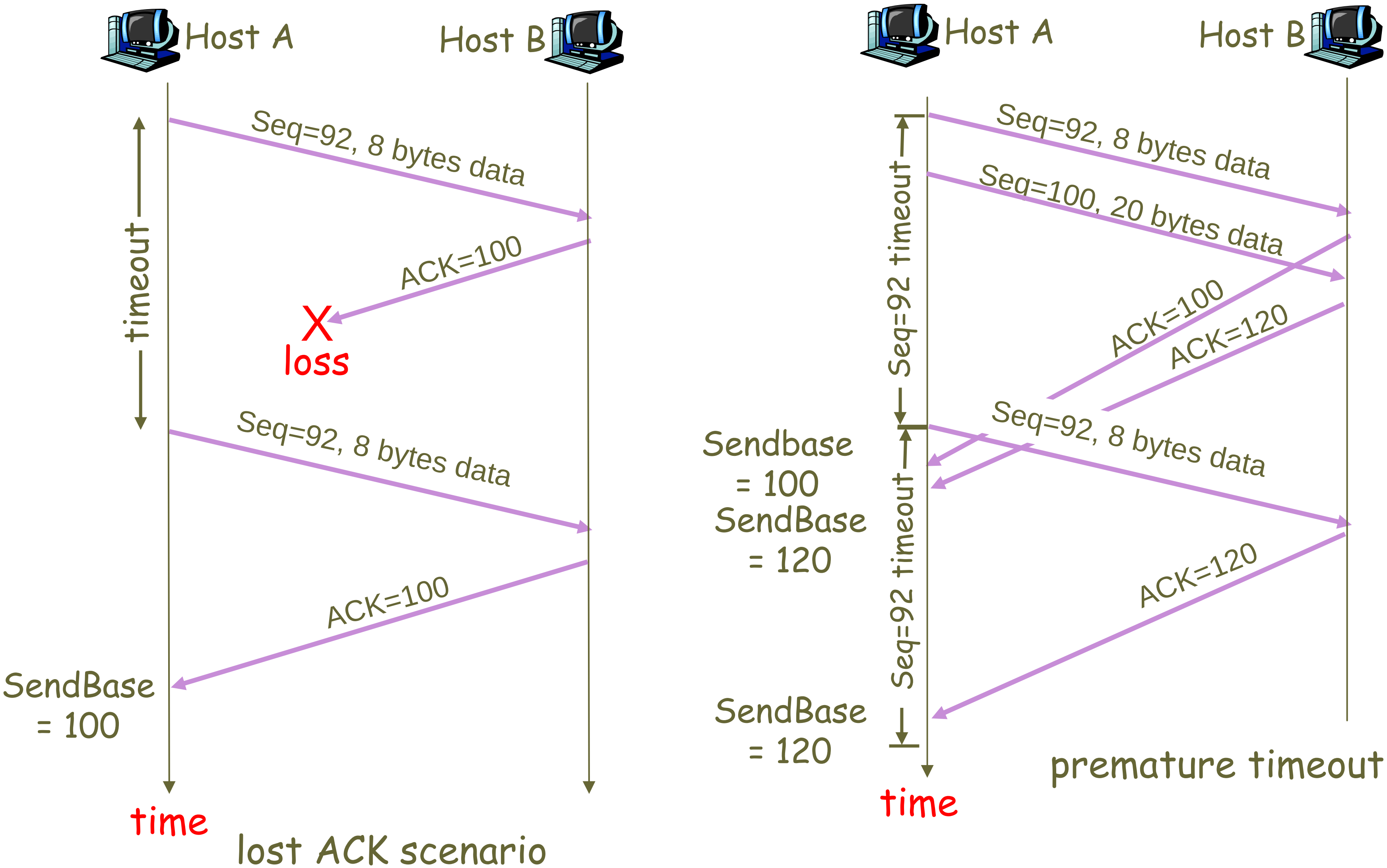
- 假如確認爲之前未確認的資料分段
  - 更新已確認的狀態
  - 假如還有未確認的資料分段、重新啓動計時器

```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

 loop (forever) {
   switch(event)

   event: data received from application above
        create TCP segment with sequence number NextSeqNum
        if (timer currently not running)
              start timer
        pass segment to IP
        NextSeqNum = NextSeqNum + length(data)

    event: timer timeout
        retransmit not-yet-acknowledged segment with
                smallest sequence number
        start timer

    event: ACK received, with ACK field value of y
        if (y > SendBase) {
              SendBase = y
             if (there are currently not-yet-acknowledged segments)
                     start timer
             }

 }  /* end of loop forever */
```

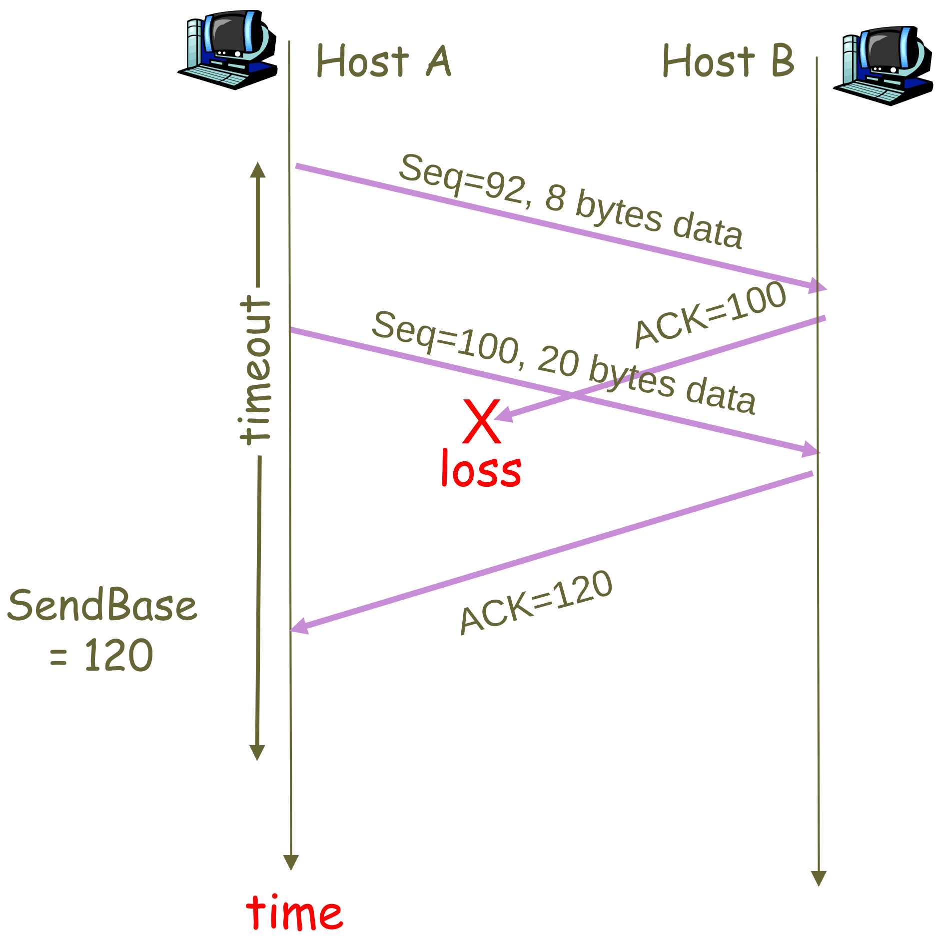
Comment:
- SendBase-1: last cumulatively ack'ed byte

Example:
- SendBase-1 = 71; y= 73, so the rcvr wants 73+ ; y > SendBase, so that new data is acked

TCP: retransmission scenarios



TCP retransmission scenarios (more)



Cumulative ACK scenario

TCP ACK generation [RFC 1122, RFC 2581]

Event at Receiver	TCP Receiver action
Arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
Arrival of in-order segment with expected seq #. One other segment has ACK pending	Immediately send single cumulative ACK, ACKing both in-order segments
Arrival of out-of-order segment higher-than-expect seq. # . Gap detected	Immediately send *duplicate ACK*, indicating seq. # of next expected byte
Arrival of segment that partially or completely fills gap	Immediate send ACK, provided that segment startsat lower end of gap

TCP ACK 的產生 [RFC 1122、 RFC 2581]

接收端的事件	TCP 接收端的動作
內含預設序號的資料分段按照順序到達。所有在預期序號之前的資料都已經確認。	延後出發ACK。等待另一個應依順序到達的資料分段、等待最多500毫秒。若下一個依序資料分段未在此時間間隔內到達、則送出ACK。
內含預期序號的資料分段按照順序到達。另一個依序到達的資料分段正在等待ACK傳送。	立刻送出單一的累積式ACK、確認這兩個依照序號到達的資料分段。
未依照順序且序號超過預期序號的資料分段到達。偵測到序號中斷的情況。	立刻送出重複的ACK、指出下一個預期到達爲組的序號 (就是序號中斷範圍中的較低序號)。
資料分段的到達、可以部分或完全填滿已接收資料的中斷	即刻送出ACK、如果資料從中斷的較低序號端開始填滿。

Fast Retransmit

- **Time-out period often relatively long:**
  - long delay before resending lost packet
- **Detect lost segments via duplicate ACKs.**
  - Sender often sends many segments back-to-back
  - If segment is lost, there will likely be many duplicate ACKs.
- **If sender receives 3 ACKs for the same data, it supposes that segment after ACKed data was lost:**
  - fast retransmit: resend segment before timer expires

快速重新傳送

- 逾時間隔通常相對地太長：
  - 在重傳遺失的封包前會有很長的延遲
- 經由重複的**ACK**偵測到資料分段的遺失
  - 傳送端經常連續傳送許多資料分段
  - 假如資料分段遺失了、可能會有許多大量的重複ACK

- 假如傳送端接收到**3**個**ACK**、它會假設已確認之後的資料已經遺失了：
  - 快速重新傳送： 在計時器逾期之前、會先傳送資料分段

Fast retransmit algorithm:

```
event: ACK received, with ACK field value of y
           if (y > SendBase) {
               SendBase = y
               if (there are currently not-yet-acknowledged segments)
                   start timer
              }
           else {
                increment count of dup ACKs received for y
                if (count of dup ACKs received for y = 3) {
                    resend segment with sequence number y
                 }
```

a duplicate ACK for already ACKed segment

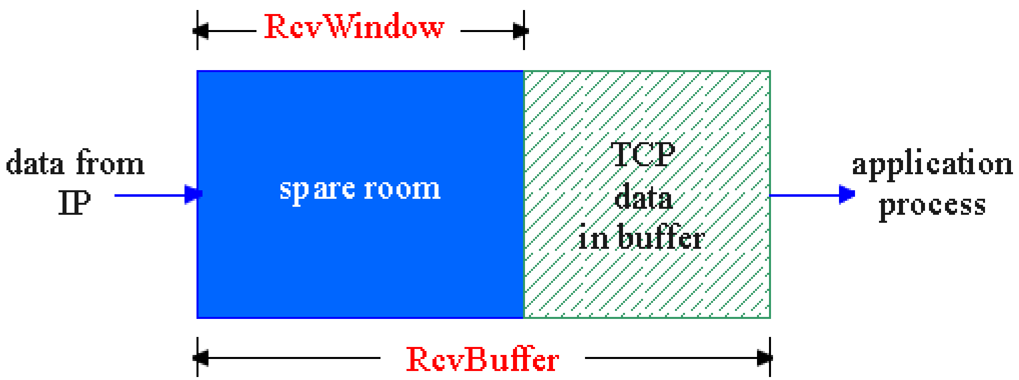
fast retransmit

TCP Flow Control

- Receive side of TCP connection has a receive buffer:

flow control

Sender won't overflow receiver's buffer by transmitting too much, too fast



- Speed-matching service: matching the send rate to the receiving app's drain rate

- App process may be slow at reading from buffer

TCP 流量控制

- TCP連線的接收端有一個接收緩衝區：

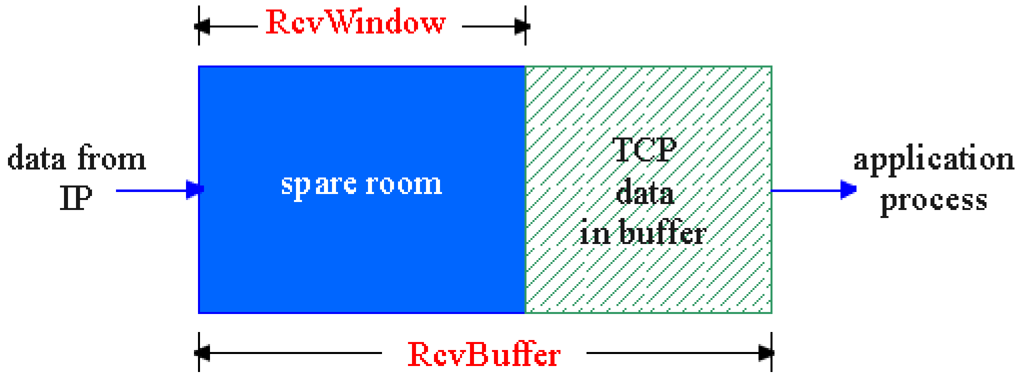
- 應用程式的行程也許會以較慢的速度從緩衝區讀取資料

流量控制

傳送端不會傳送太多太快的資料超過接收端的緩衝區

- 速度調整服務： 調整傳送端的速度與接收端應用程式能負擔的速度'相符

TCP Flow control: how it works



(Suppose TCP receiver discards out-of-order segments)

- spare room in buffer

= `RcvWindow`

= `RcvBuffer-[LastByteRcvd - LastByteRead]`

- Rcvr advertises spare room by including value of `RcvWindow` in segments
- Sender limits unACKed data to `RcvWindow`
  - guarantees receive buffer doesn't overflow

TCP 流量控制： 如何運作

(假設 TCP 接收端會將順序不正確的資料分段捨棄)

- 緩衝區內的剩餘空間

= RcvWindow

= RcvBuffer-[LastByteRcvd - LastByteRead]

- 接收端將**RcvWindow**值包含在資料分段裡、以告知剩餘的空間
- 傳送端限制未確認的資料在 RcvWindow之下
  - 保證接收端緩衝區不會溢出

TCP Connection Management

Recall: **TCP sender, receiver establish "connection" before exchanging data segments**

- **initialize TCP variables:**
  - seq. #s
  - buffers, flow control info (e.g. `RcvWindow`)
- ***client:* connection initiator**

```
Socket clientSocket = new
 Socket("hostname","port

 number");
```

- ***server:* contacted by client**

```
Socket connectionSocket =
 welcomeSocket.accept();
```

Three way handshake:

Step 1: **client host sends TCP SYN segment to server**

- specifies initial seq #
- no data

Step 2: **server host receives SYN, replies with SYNACK segment**

- server allocates buffers
- specifies server initial seq. #

Step 3: **client receives SYNACK, replies with ACK segment, which may contain data**

TCP 連線管理

回想： TCP 傳送端、接收端在交換資料分段之前、會先建立"連線"

- 將 TCP 變數初始化：
  - 序號
  - 緩衝區、流量控制資訊 (例如 `RcvWindow`)
- 用戶端： 開始連線者

```
Socket clientSocket = new
 Socket("hostname"、"port
 number");
```

- 伺服端： 被用戶端聯繫

```
Socket connectionSocket =
 welcomeSocket.accept();
```

三路交握：

步驟 1： 用戶端主機傳送 TCP SYN 資料分段到伺服器

- 指定初始的序號
- 沒有資料

步驟 2： 伺服端主機收到 SYN、以 SYNACK 資料分段回應

- 伺服端配置緩衝區
- 指定伺服端的初始序號

步驟 3： 用戶端收到 SYNACK、回應 ACK 資料分段、可能含有資料

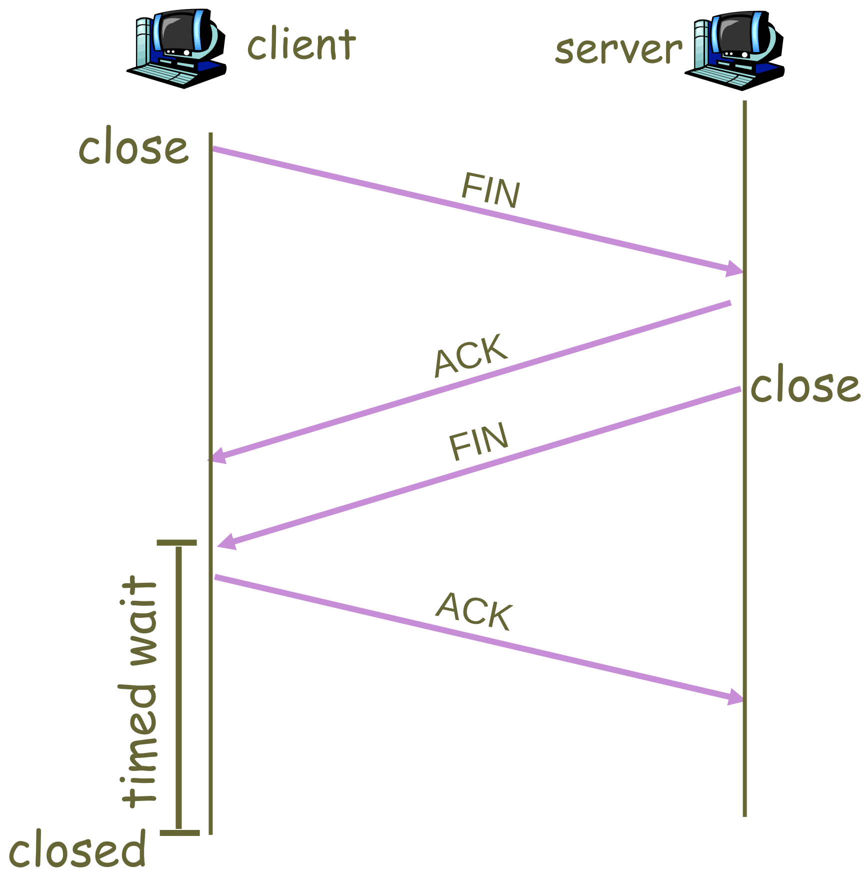
TCP Connection Management (cont.)

Closing a connection:

client closes socket:
`clientSocket.close();`

Step 1: **client end system sends TCP FIN control segment to server**

Step 2: **server receives FIN, replies with ACK. Closes connection, sends FIN.**



TCP 連線管理 (續)

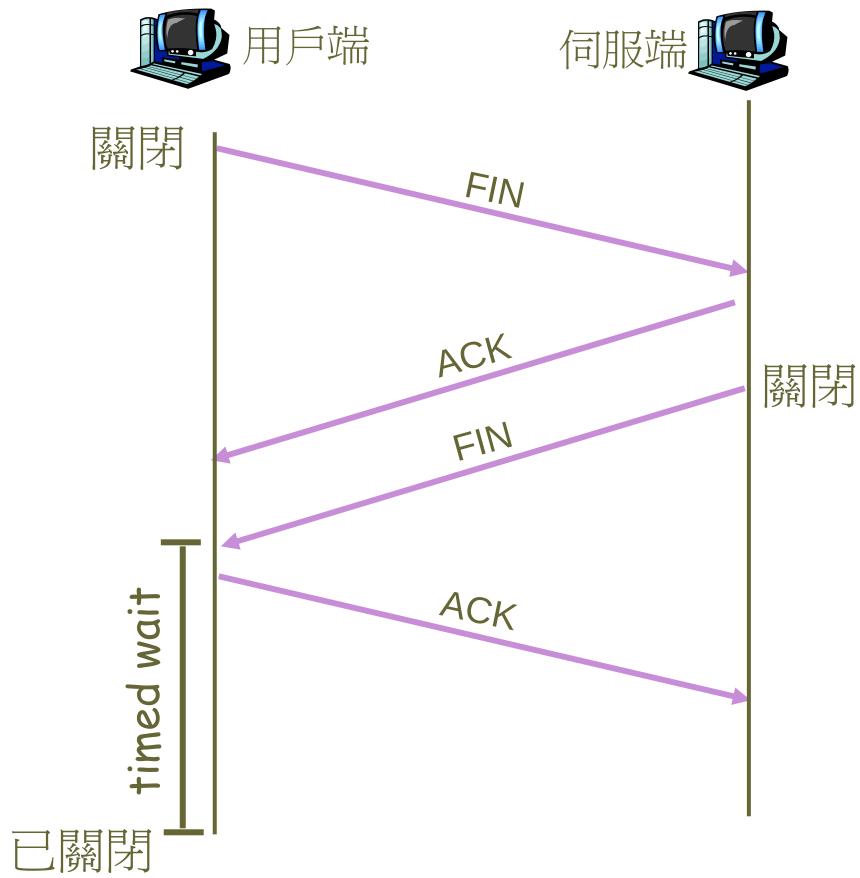
關閉連線：

用戶端關閉 **socket**：

```
clientSocket.close();
```

步驟 1： 用戶端終端系統傳送 **TCP FIN**控制分段到伺服端

步驟 2： 伺服端 接收到**FIN**、以 **ACK** 回應。關閉連線、傳送 **FIN**。



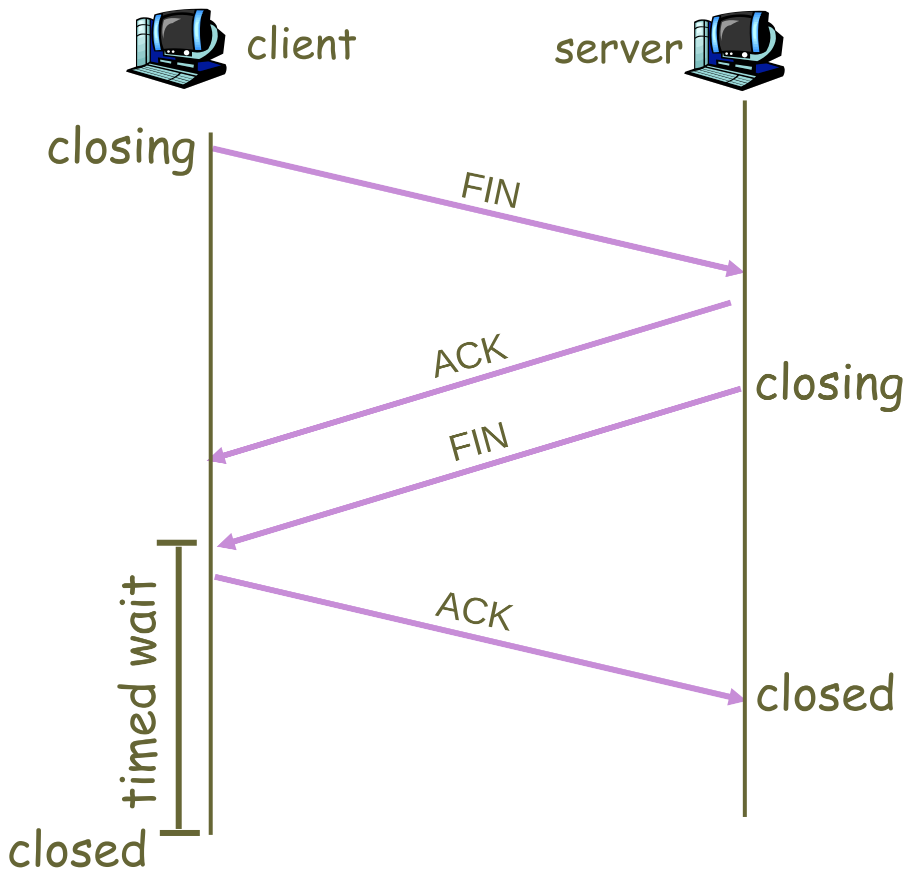
TCP Connection Management (cont.)

Step 3: **client receives FIN, replies with ACK.**

- Enters "timed wait" - will respond with ACK to received FINs

Step 4: **server, receives ACK. Connection closed.**

Note: **with small modification, can handle simultaneous FINs.**



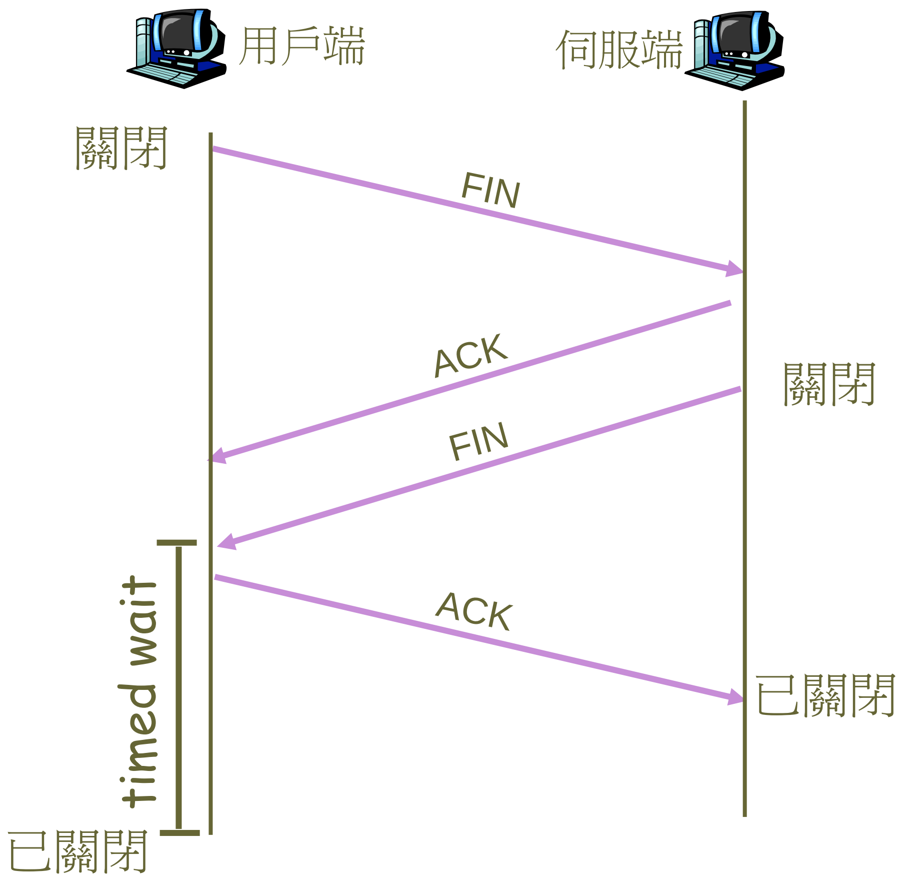
TCP 連線管理 (續)

步驟 3： 用戶端 收到 **FIN**、回應 **ACK** 訊息。

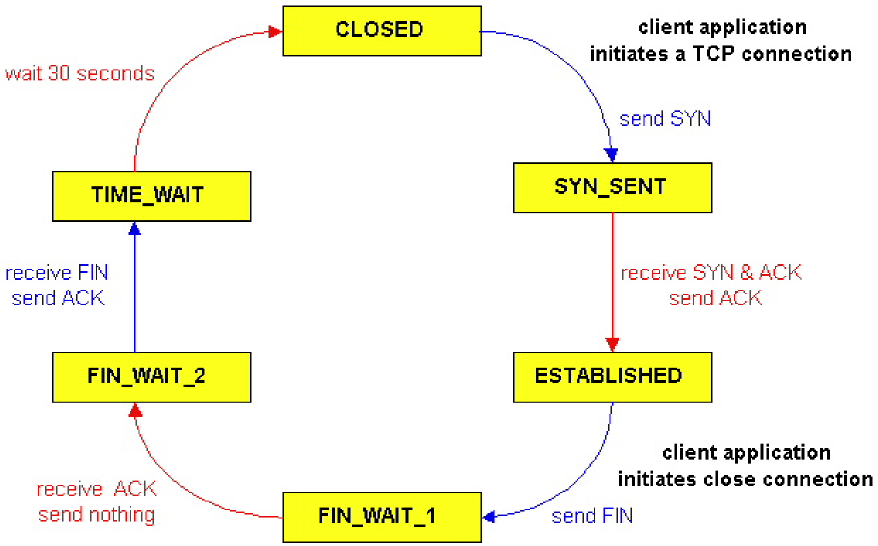
- 進入 "等待計時" - 對接收到的 FIN 做確認的回應

步驟 4： 伺服端、收到**ACK**。連線關閉。

注意： 做一點小修改、 可以處理同時的 **FIN**。

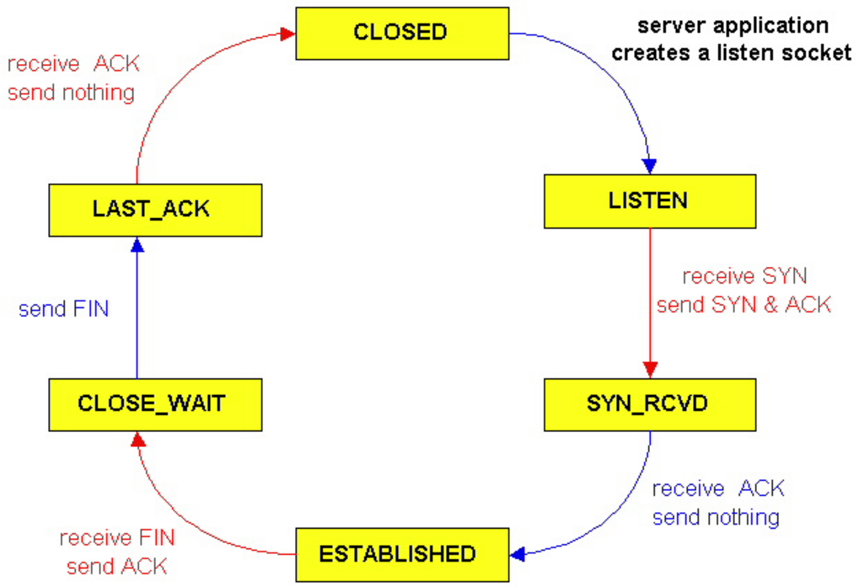


TCP Connection Management (cont)

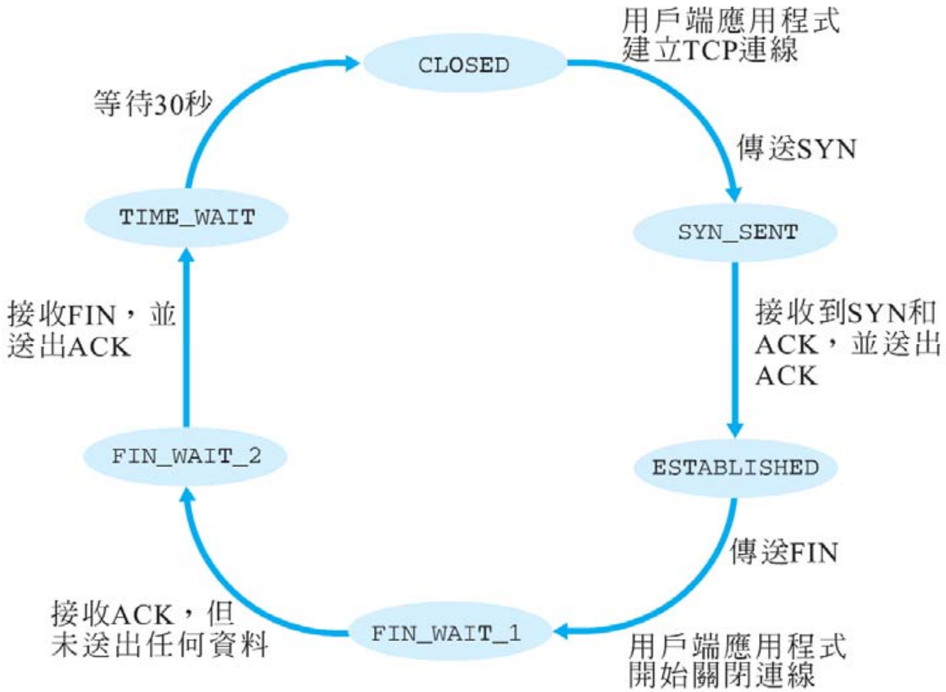


TCP client lifecycle

TCP server lifecycle

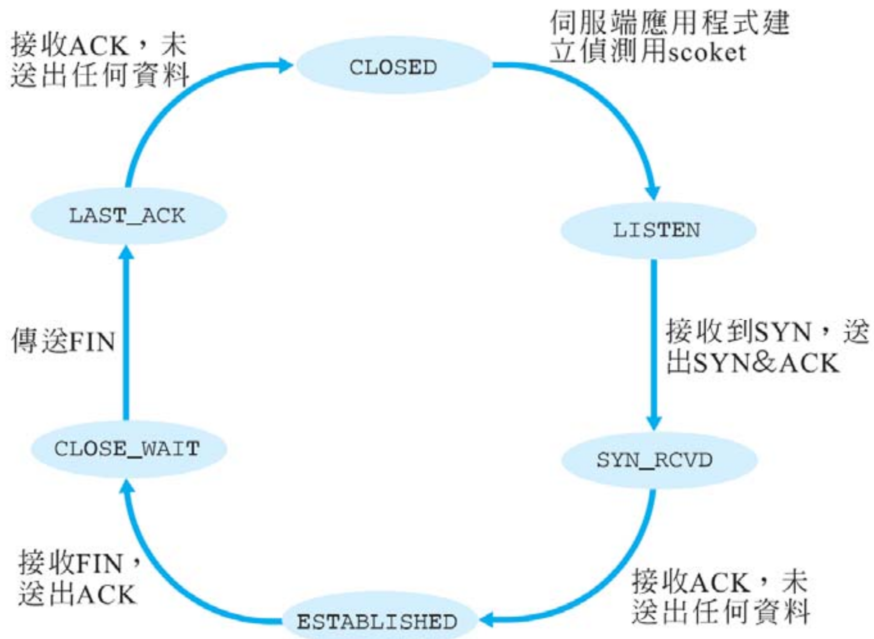


TCP連線管理 (續)



TCP 用戶端
生命週期

TCP 伺服端
生命週期



3.6 Principles of congestion control

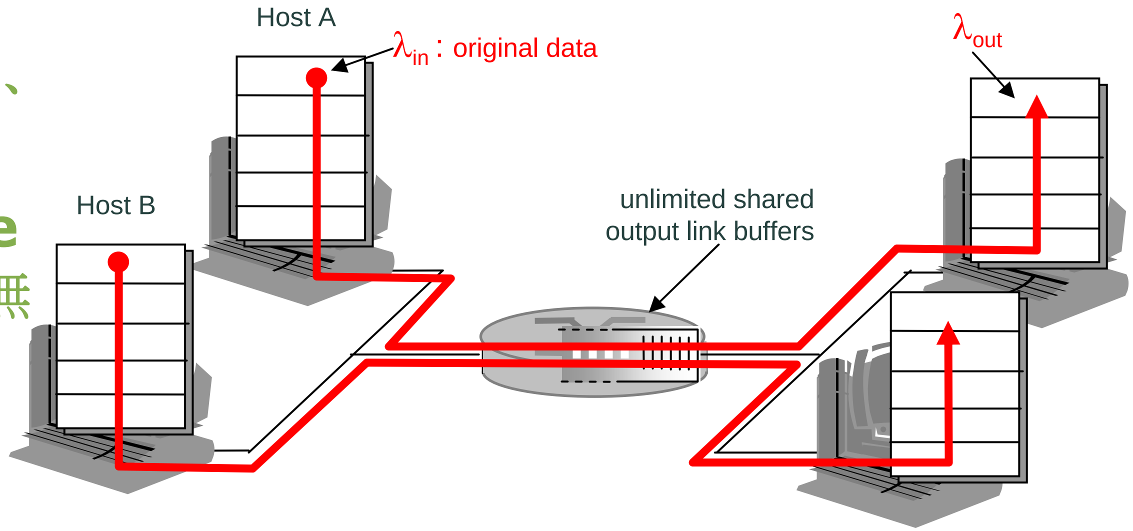
Principles of Congestion Control

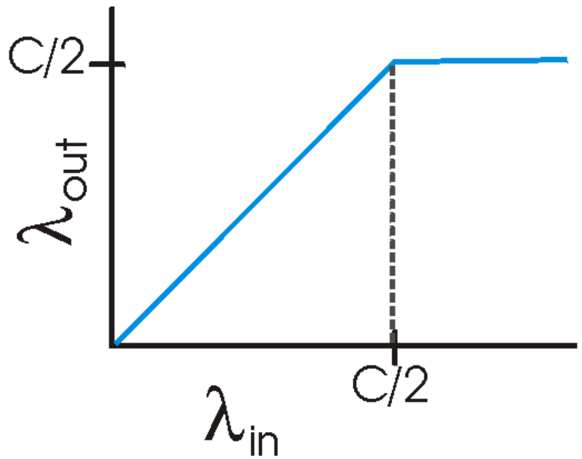
Congestion:

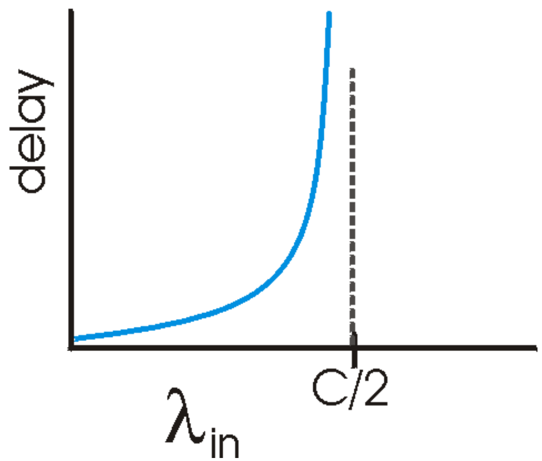
- **Informally: "too many sources sending too much data too fast for *network* to handle"**
  非正式地："太多的來源端傳送太多的資料、對網路來說太快、超過能處理的速度"
- **Different from flow control**與流量控制不同**!**
- **Manifestations**表現形式**:**
  - Lost packets (buffer overflow at routers)
    封包遺失 (路由器緩衝區溢出)
  - Long delays (queueing in router buffers)
    長的延遲 (在路由器緩衝區佇列中等待)

Causes/costs of congestion壅塞的原因和代價: scenario 1

- **Two senders, two receivers**兩個傳送端、兩個接收端
- **One router, infinite buffers**一個路由器、無限的緩衝區
- **No retransmission**沒有重傳機制



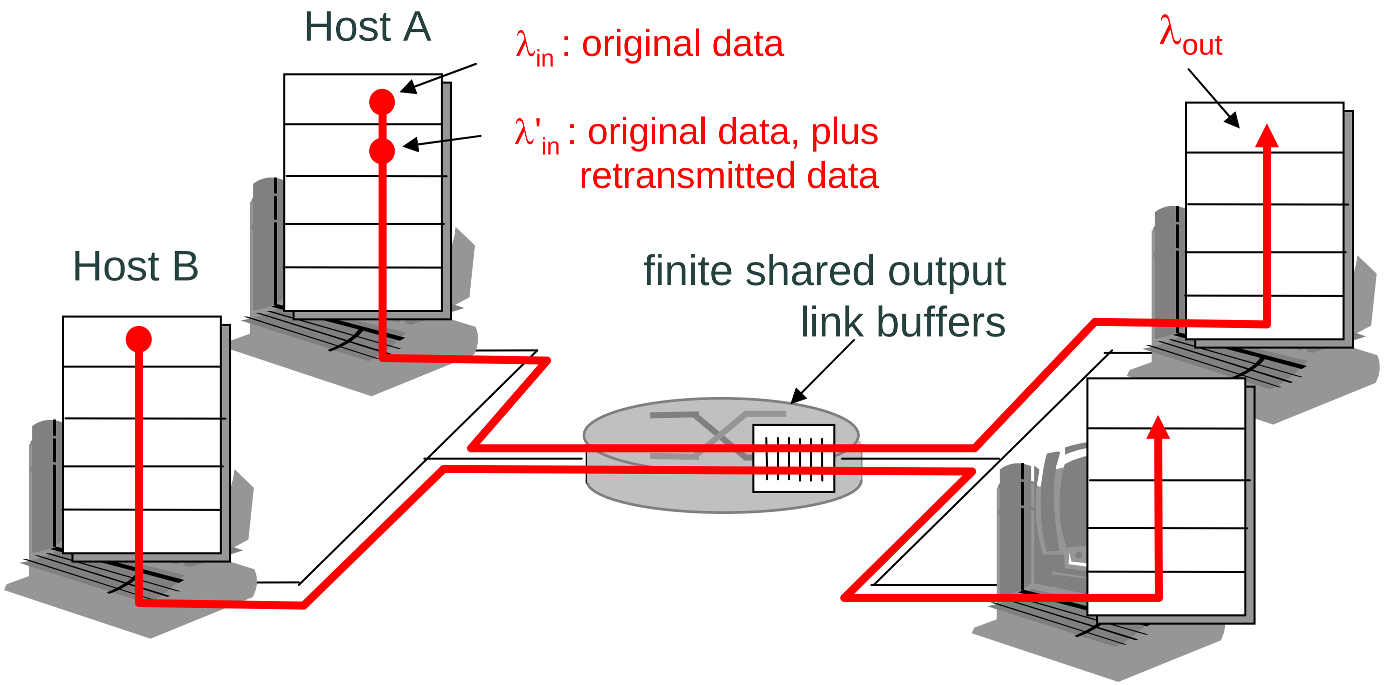




- **Large delays when congested**當壅塞時會有很長的延遲
- **Maximum achievable throughput**最大的可達成流通量

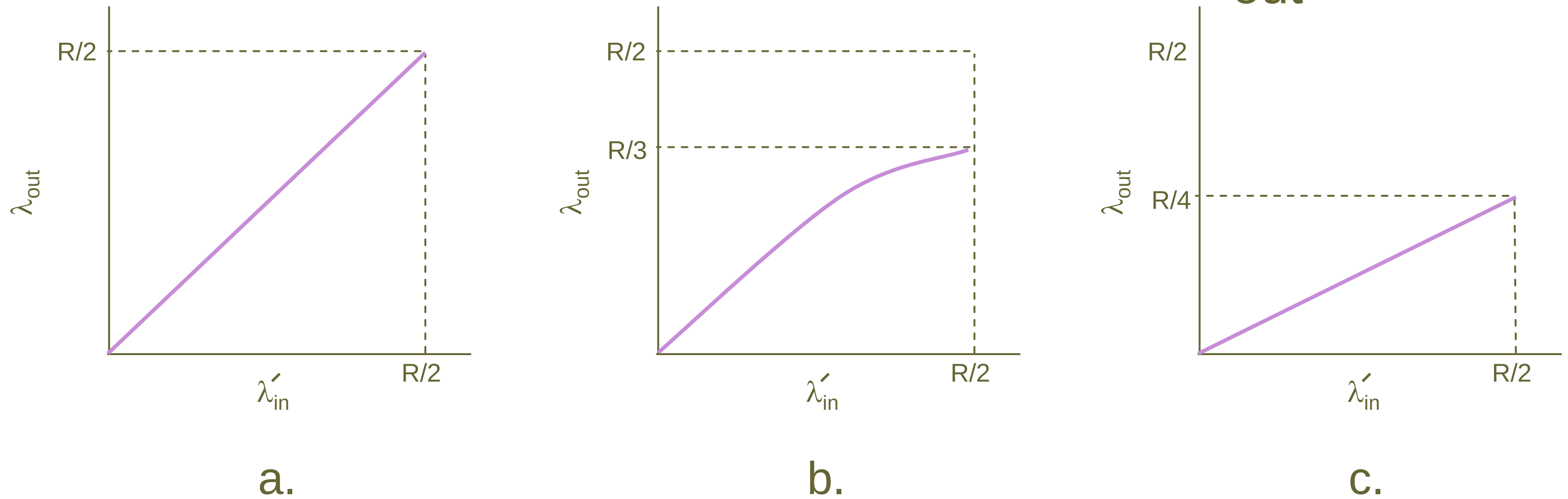
Causes/costs of congestion: scenario 2

- **One router, *finite* buffers**
  一個路由器、*有限的* 緩衝區
- **Sender retransmission of lost packet**傳送端會重新傳送遺失的封包



Causes/costs of congestion: scenario 2

- **Always:** $\lambda_{in} = \lambda_{out}$ **(goodput)**
- **"perfect" retransmission only when loss:** $\lambda'_{in} > \lambda_{out}$
- **Retransmission of delayed (not lost) packet makes** λ'_{in} **larger (than perfect case) for same** λ_{out}

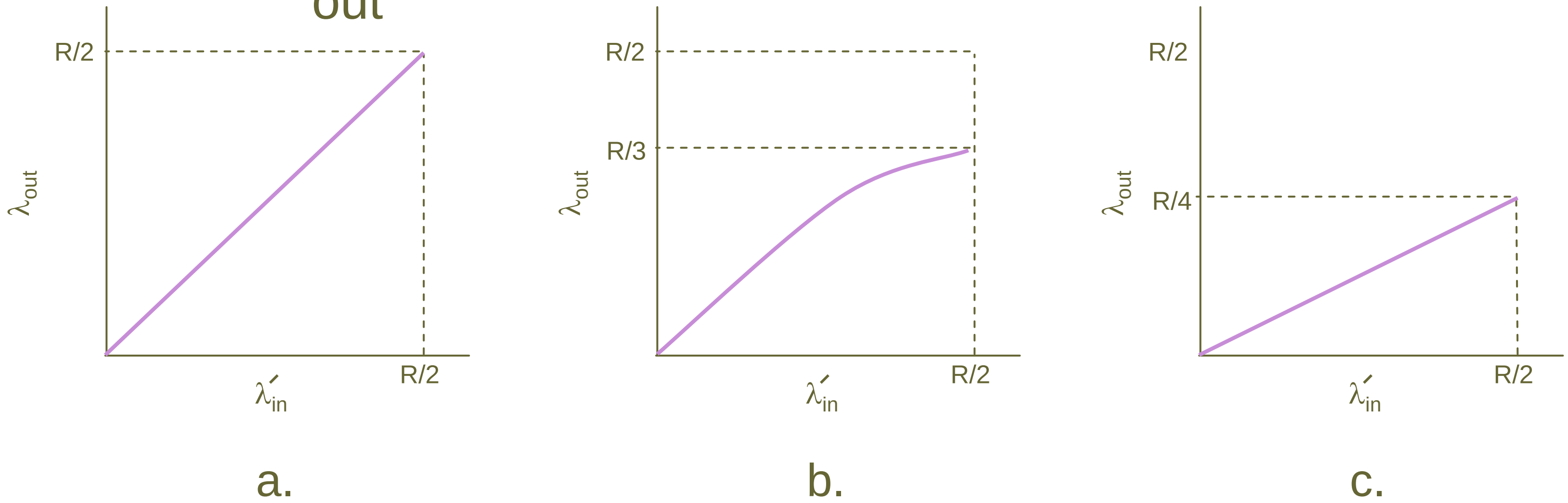


"costs" of congestion:

- **More work (retrans) for given "goodput"**
- **Unneeded retransmissions: link carries multiple copies of pkt**

壅塞的原因和代價： 情況 2

- 總是： $\lambda_{in} = \lambda_{out}$ (goodput、實際產量)
- “理想的” 重新傳送、只在遺失：$\lambda'_{in} > \lambda_{out}$
- 傳送延遲的封包 (並非遺失) 會使的 λ'_{in} 較大 (大於理想狀況)、在相同的 λ_{out} 下



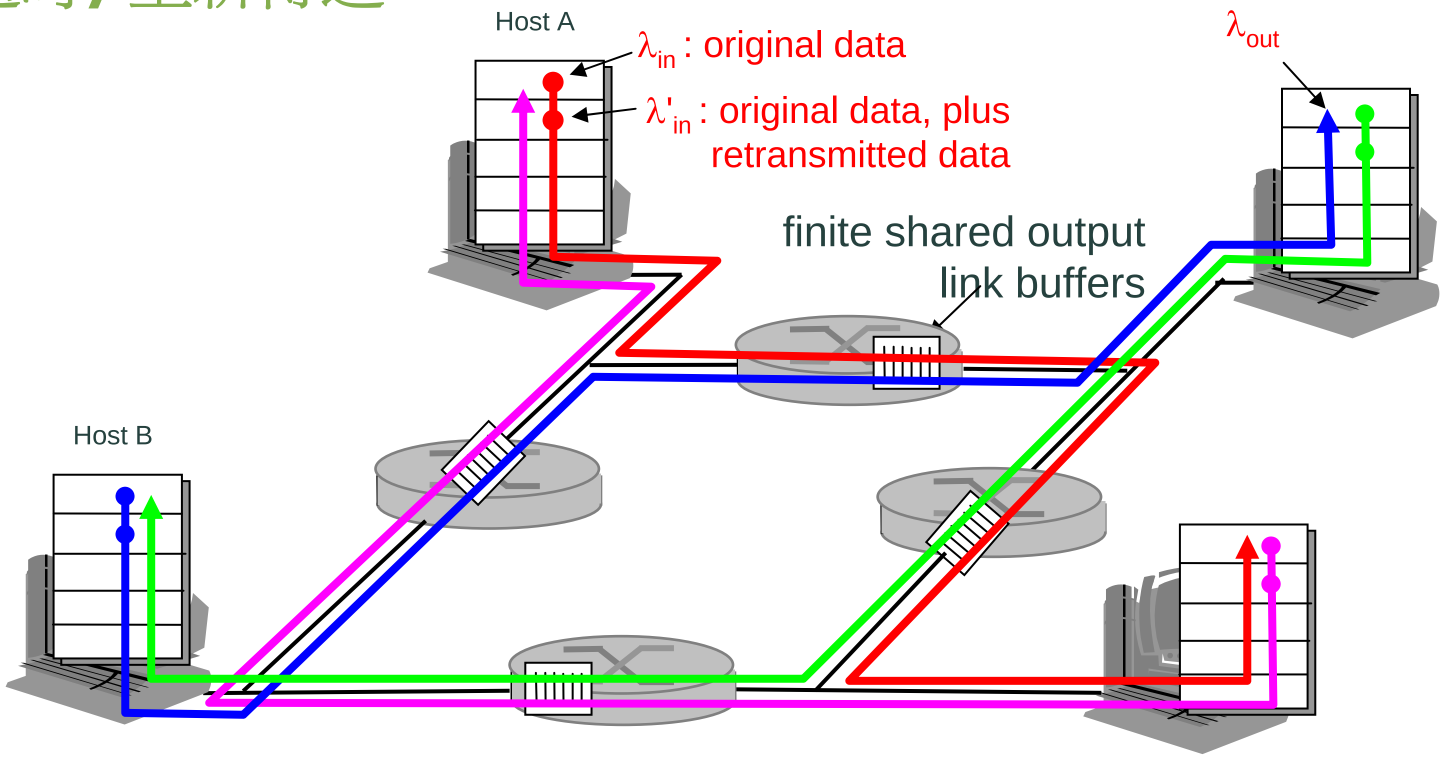
壅塞的”代價”：

- 對給定的 “實際產量”(goodput)、會有更多的工作 (重新傳輸)
- 不需要的重新傳輸： 連結必須負擔多個封包的副本

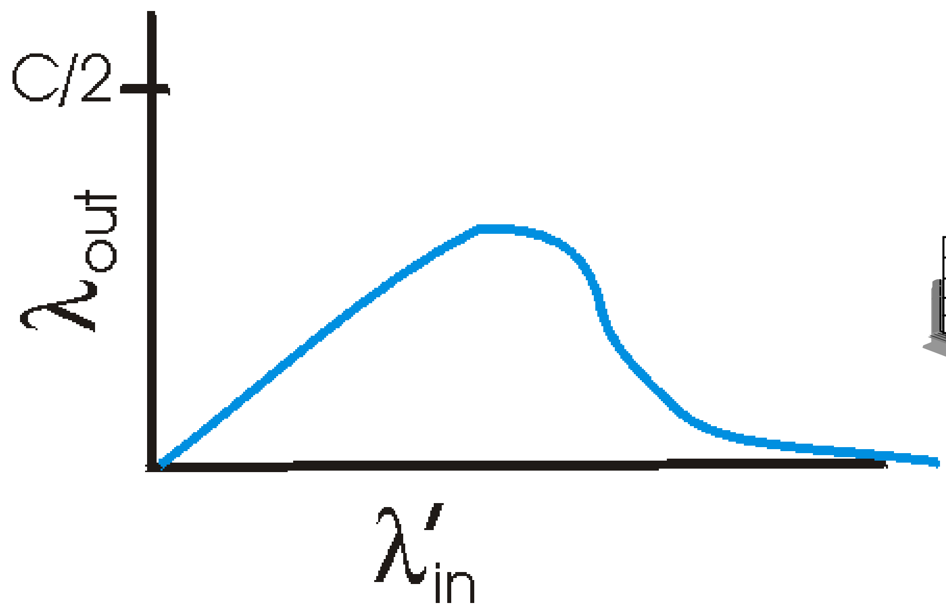
Causes/costs of congestion: scenario 3

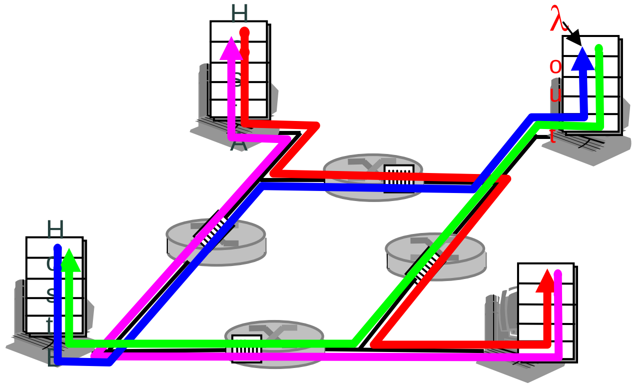
- Four senders四個傳送端
- Multihop paths多次轉接路徑
- Timeout/retransmit 逾時/重新傳送

Q: What happens λ_{in} as λ'_{in} and increase **?**



Causes/costs of congestion: scenario 3





Another "cost" of congestion壅塞的另一個代價:

- **When packet dropped, any "upstream transmission capacity used for that packet was wasted!**當封包被丟掉時、此封包所使用到的任何"上游"傳送容量就被浪費掉了

Approaches towards congestion control

Two broad approaches towards congestion control:

End-end congestion control:

- No explicit feedback from network
- Congestion inferred from end-system observed loss, delay
- Approach taken by TCP

Network-assisted congestion control:

- Routers provide feedback to end systems
  - Single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM)
  - Explicit rate sender should send at

壅塞控制的方法

壅塞控制的兩個主要方法：

端點對端點壅塞控制：

- 網路層並沒有提供明顯的協助
- 根據中端系統觀察到的遺失及延遲來判斷壅塞
- **TCP** 採用的方法

網路協助的壅塞控制：

- 路由器提供協助給終端系統
  - 以一個位元來表示壅塞 (SNA、 DECbit、 TCP/IP ECN、 ATM)
  - 傳送端應該傳送的明確速率

Case study: ATM ABR congestion control

ABR: available bit rate:

- **"elastic service"**
- **If sender's path "underloaded":**
  - Sender should use available bandwidth
- **If sender's path congested:**
  - Sender throttled to minimum guaranteed rate

RM (resource management) cells:

- **Sent by sender, interspersed with data cells**
- **Bits in RM cell set by switches ("*network-assisted*")**
  - NI bit: no increase in rate (mild congestion)
  - CI bit: congestion indication
- **RM cells returned to sender by receiver, with bits intact**

案例研究： ATM ABR 壅塞控制

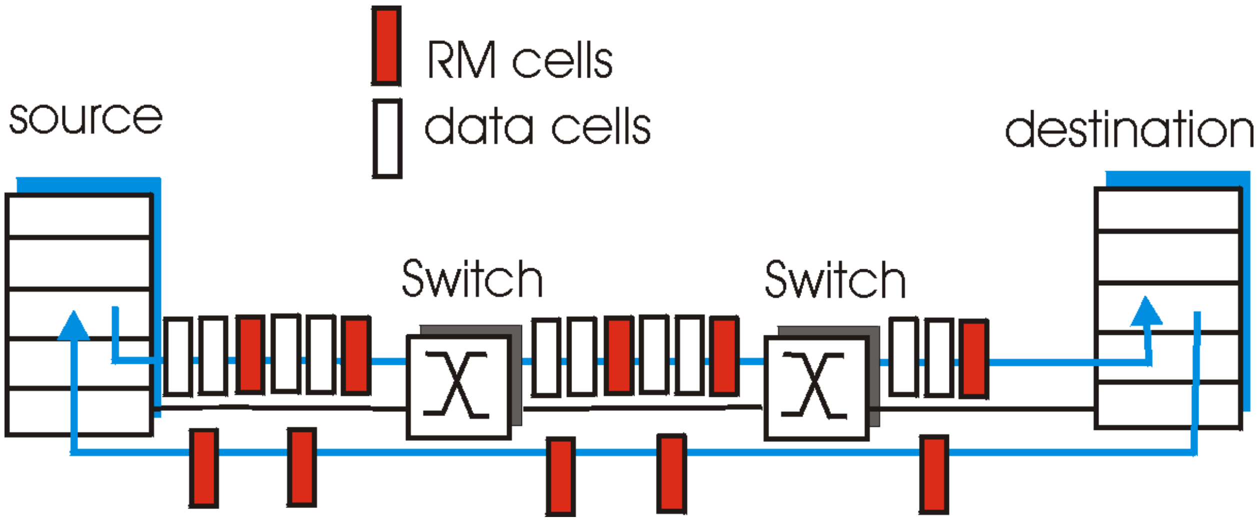
ABR： 可用的位元速率：

- “彈性的服務”
- 假如傳送端路徑 “負載量很低”時：
  - 傳送端可以利用可用的頻寬
- 假如傳送端路徑壅塞時：
  - 傳送端減速到最小的保證速率

RM (資源管理) 封包單位：

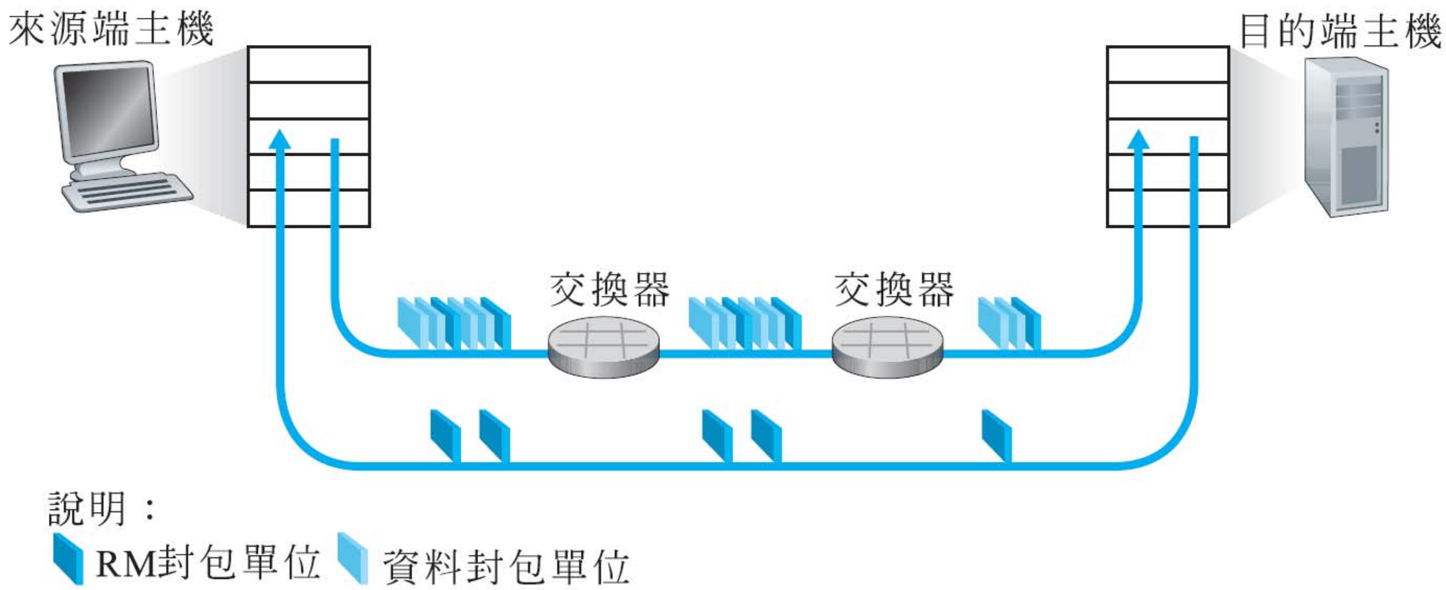
- 傳送端所傳送的、配置在資料封包單位中
- RM封包單位中的位元、由交換器設定 (“網路協助”)
  - NI 位元： 不增加速率 (輕微壅塞)
  - CI 位元： 壅塞指示
- RM 封包單位的位元由接收端原封不動地送回給傳送端

Case study: ATM ABR congestion control



- **Two-byte ER (explicit rate) field in RM cell**
  - Congested switch may lower ER value in cell
  - Sender' send rate thus maximum supportable rate on path
- **EFCI bit in data cells: set to 1 in congested switch**
  - If data cell preceding RM cell has EFCI set, sender sets CI bit in returned RM cell

案例研究： ATM ABR 壅塞控制



- ❖ **RM封包單位中、兩個位元組的 ER (明確速率) 欄位**
  - 壅塞的交換器會降低封包單位中的 ER 值
  - 因此、傳送端的傳送速率爲路徑上最低可支援速率
- ❖ **資料封包單位中的EFCI 位元：在壅塞的交換器中設定爲1**
  - 假如 RM 封包單位之前的資料封包的EFCI都被設定、則傳送端會將CI位元設定在回傳的RM封包單位中

3.7 TCP congestion control

TCP Congestion Control: details

- **Sender limits transmission:**
  `LastByteSent-LastByteAcked` $\leq$ `CongWin`
- **Roughly,**

$$\boxed{\text{rate} = \frac{\text{CongWin}}{\text{RTT}} \text{ Bytes/sec}}$$

- `CongWin` **is dynamic, function of perceived network congestion**

How does sender perceive congestion?

- **Loss event = timeout *or* 3 duplicate acks**
- **TCP sender reduces rate (`CongWin`) after loss event**

Three mechanisms:

- AIMD
- slow start
- conservative after timeout events

TCP 壅塞控制： 細節

- 傳送端限制速率：
  LastByteSent-LastByteAcked
  ≤ CongWin
- 大致上、

$$\boxed{\text{rate} = \frac{\text{CongWin}}{\text{RTT}}\ \text{Bytes/sec}}$$

- CongWin 是動態的、 是察覺的網路壅塞函數

<u>傳送端如何察覺到壅塞狀況?</u>

- 遺失事件 = 逾時或3個重複的ack
- 在遺失事件之後、TCP 傳送端會降低速率 (CongWin)

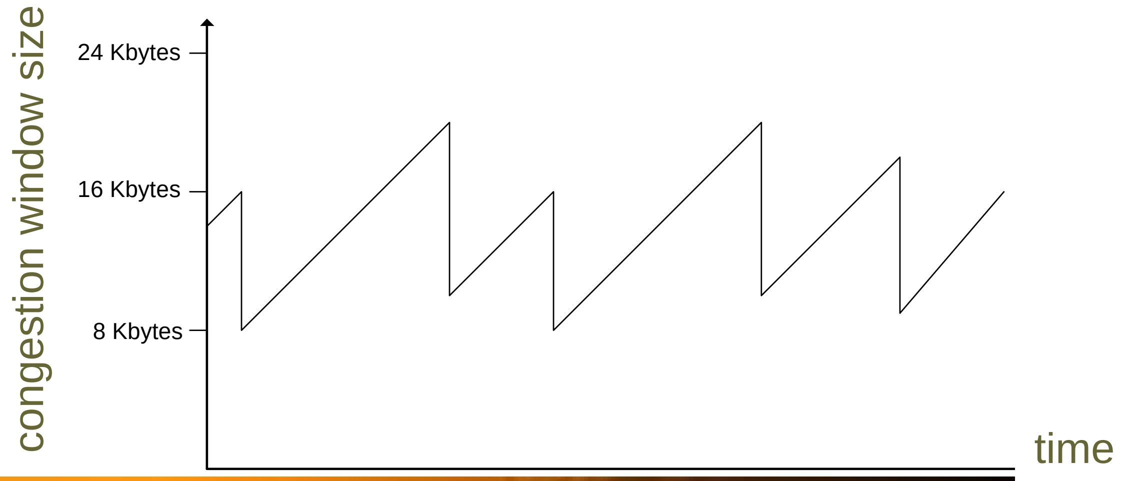
<u>三個機制：</u>

- AIMD
- 緩數啓動
- 發生逾時事件後的保守態度

TCP congestion control: additive increase, multiplicative decrease TCP 壅塞控制： 累積遞增、倍數遞減

- ***Approach:*** increase transmission rate (window size), probing for usable bandwidth, until loss occurs
  增加傳送速率 (視窗大小)、 探測可用的頻寬、 直到發生遺失的狀況
  - *Additive increase累積遞增:* increase **CongWin** by 1 MSS every RTT until loss detected 每個 RTT將 **CongWin** 加 1、直到發生遺失
  - *Multiplicative decrease倍數遞減:* cut **CongWin** in half after loss 在發生遺失之後、將 **CongWin** 減爲一半

Saw tooth behavior看到鋸齒形式: probing for bandwidth頻寬的探測



AIMD (Additive-Increase, Multiplicative-Decrease)累加遞增、倍數遞減

- **TCP congestion control is for the sender to reduce its sending rate (by decreasing its congestion window size, CongWin)** 讓傳送端在發生遺失事件時，降低它的傳送速率（減少**CongWin** 大小）
  - Loss➔ CongWin size減少一半
  - No loss➔ CongWin zise每次加1
  - Additive-Increase(累加遞增)稱爲congestion avoidance（壅塞迴避）

TCP Slow Start

- **When connection begins, `CongWin` = 1 MSS**
  - Example: MSS = 500 bytes & RTT = 200 msec
  - Initial rate = 20 kbps
- **Available bandwidth may be >> MSS/RTT**
  - Desirable to quickly ramp up to respectable rate
- **When connection begins, increase rate exponentially fast until first loss event**

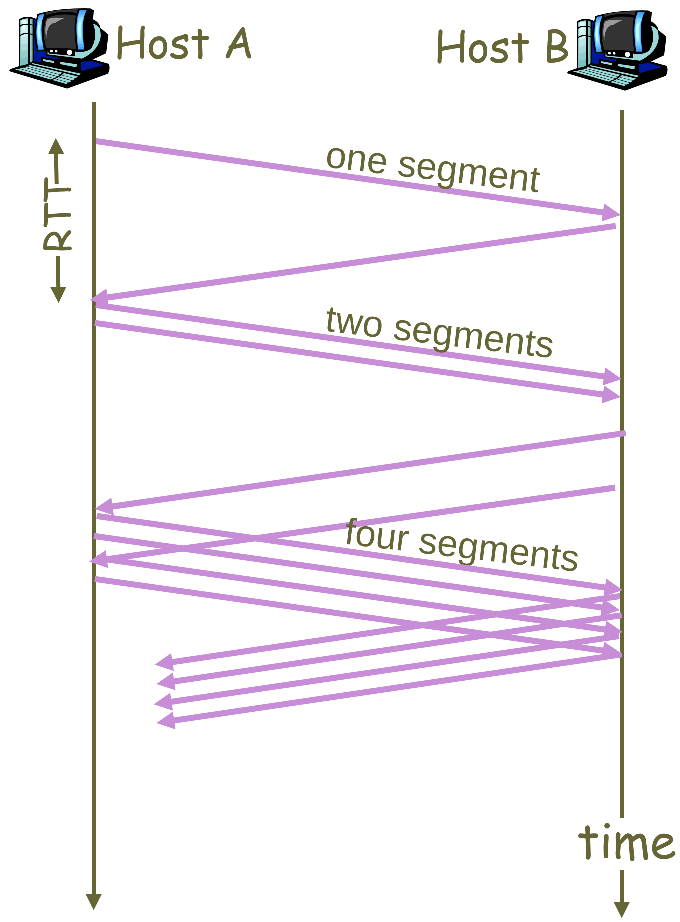
TCP 緩速啓動

- 當連線一開始時、`CongWin` **= 1 MSS**
  - 範例： MSS = 500 位元組 & RTT = 200 毫秒
  - 初始速率 = 20 kbps
- 可用的頻寬可能 **>> MSS/RTT**
  - 想要快速地增加到可接受的速率
- 當連結開始時、以指數型式增加速率、直到第一個遺失發生

MSS: maximum segment size

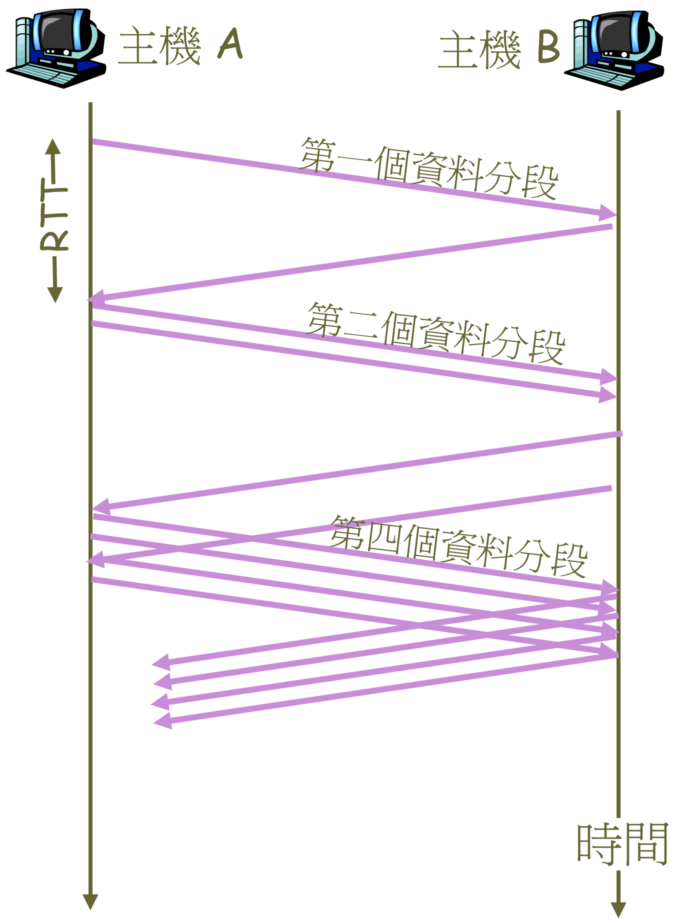
TCP Slow Start (more)

- **When connection begins, increase rate exponentially until first loss event:**
  - Double **`CongWin`** every RTT
  - Done by incrementing **`CongWin`** for every ACK received
- **Summary: initial rate is slow but ramps up exponentially fast**



TCP 緩速啓動 (更多)

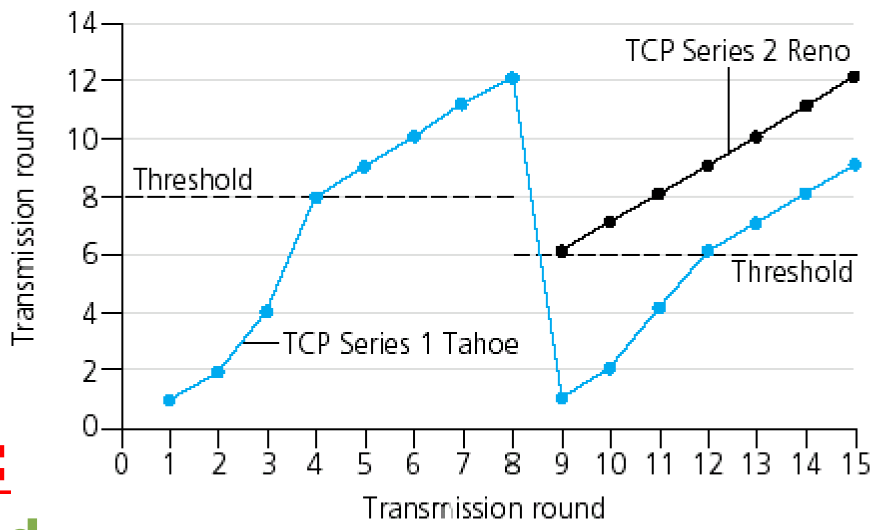
- 當連結開始時、以指數型式增加速率、直到第一個遺失事件發生：
  - 在每次的 RTT、將 **`CongWin`** 增為一倍
  - 每次收到 ACK 時、會增加 **`CongWin`**
- <u>總結：</u> 開始的速率是緩慢的、但會以指數形式快速增加速率



Refinement

Q: When should the exponential increase switch to linear?

A: When `CongWin` gets to 1/2 of its value before timeout.



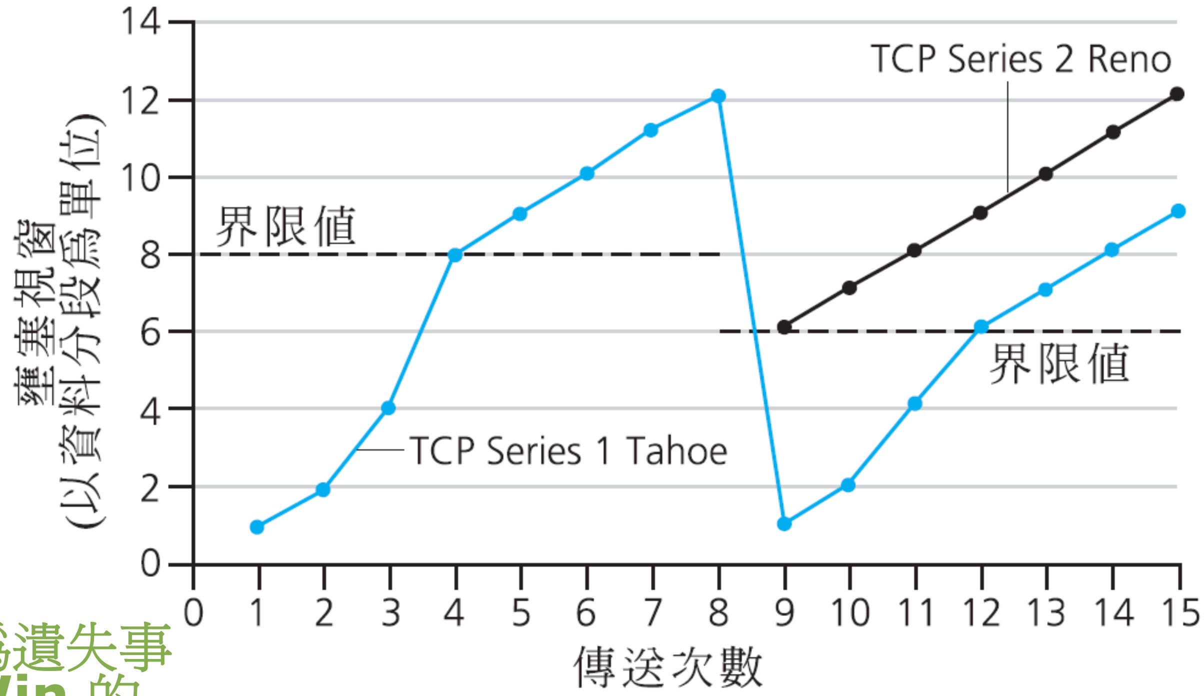
Implementation:

- **Variable Threshold**
- **At loss event, Threshold is set to 1/2 of CongWin just before loss event**

再改良

Q： 指數形式的增長什麼時候會轉換爲線性的**?**

A： 當 CongWin 到達逾時事件發生前的一半大小



實作：

- 變數 **Threshold**
- 在遺失事件發生時、**Threshold** 會被設爲遺失事件發生之前的**CongWin** 的 **1/2**。

Refinement: inferring loss

- **After 3 dup ACKs:**
  - `CongWin` is cut in half
  - Window then grows linearly
- **But after timeout event:**
  - `CongWin` instead set to 1 MSS;
  - Window then grows exponentially
  - To a threshold, then grows linearly

Philosophy:

- 3 dup ACKs indicates network capable of delivering some segments
- timeout indicates a "more alarming" congestion scenario

再改良： 推論遺失

❖ 在三個重複的**ACK**之後：
  - **CongWin** 減爲一半
  - 視窗接下來會線性成長

❖ 但是在逾時事件後：
  - **CongWin** 設爲 1 MSS;
  - 視窗會以指數增長
  - 到一個門檻、接著以線性成長

哲學：

- 3個重複的 *ACKs* 表示網路有能力傳送某些資料分段
- 逾時表示較爲嚴重的壅塞狀況

Summary: TCP Congestion Control

- **When** `CongWin` **is below** `Threshold`**, sender in slow-start phase, window grows exponentially.**
- **When** `CongWin` **is above** `Threshold`**, sender is in congestion-avoidance phase, window grows linearly.**
- **When a triple duplicate ACK occurs,** `Threshold` **set to** `CongWin/2` **and** `CongWin` **set to** `Threshold`**.**
- **When timeout occurs,** `Threshold` **set to** `CongWin/2` **and** `CongWin` **is set to 1 MSS.**

總結： TCP 壅塞控制

- 當 CongWin 在 Threshold 之下且傳送端在緩速啓動階段時、視窗以指數成長。
- 當 CongWin 在 Threshold 之上且傳送端在壅塞避免階段、視窗以線性成長。
- 當 三個重複的 **ACK** 發生時、將 Threshold 設定爲 CongWin/2 且 CongWin 設定爲 Threshold。
- 當 逾時 發生時、 Threshold 設定爲 CongWin/2 且 CongWin 設定爲 **1 MSS**。

TCP sender congestion control

State	Event	TCP Sender Action	Commentary
Slow Start (SS)	ACK receipt for previously unacked data	CongWin = CongWin + MSS, If (CongWin > Threshold) set state to “Congestion Avoidance”	Resulting in a doubling of CongWin every RTT
Congestion Avoidance (CA)	ACK receipt for previously unacked data	CongWin = CongWin+MSS * (MSS/CongWin)	Additive increase, resulting in increase of CongWin by 1 MSS every RTT
SS or CA	Loss event detected by triple duplicate ACK	Threshold = CongWin/2, CongWin = Threshold, Set state to “Congestion Avoidance”	Fast recovery, implementing multiplicative decrease. CongWin will not drop below 1 MSS.
SS or CA	Timeout	Threshold = CongWin/2, CongWin = 1 MSS, Set state to “Slow Start”	Enter slow start
SS or CA	Duplicate ACK	Increment duplicate ACK count for segment being acked	CongWin and Threshold not changed

TCP 傳送端壅塞控制

狀態	事件	TCP 傳送端動作	註解
緩速啓動 (SS)	收到下一個時確認資料的ACK	CongWin = CongWin + MSS、如果(CongWin > Threshold) 設定狀態爲「壅塞避免」	導致在每個RTT時間內CongWin數值的倍增
壅塞避免 (CA)	收到下一個待確認資料的ACK	CongWin = CongWin+MSS * (MSS/CongWin)	累加遞增、導致CongWin在每個RTT時間內增加1MSS
SS or CA	偵測到三個重複ACK的遺失事件	Threshold = CongWin/2、CongWin = Threshold、設定狀態爲「壅塞避免」	快速回復、採用倍數遞減。CongWin值不會低於1MSS
SS or CA	逾時	Threshold = CongWin/2、CongWin = 1 MSS、設定狀態爲「緩速啓動」	進入緩速啓動
SS or CA	重複 ACK	增加資料分段被確認的重複ACK記數	CongWin及Threshold不會改變

TCP throughput

- What's the average throughout of TCP as a function of window size and RTT?
  - Ignore slow start
- Let W be the window size when loss occurs.
- When window is W, throughput is W/RTT
- Just after loss, window drops to W/2, throughput to W/2RTT.
- Average throughout: .75 W/RTT

TCP 流通量

- **TCP**的平均流通量爲何？以視窗大小以及**RTT**值的函數表示**?**
  - 忽略緩慢啓動階段
- 令 **W** 爲遺失發生時的視窗大小
- 當視窗大小爲**W**時、流通量爲 **W/RTT**
- 在遺失發生之後、視窗馬上降爲 **W/2**、流通量爲 **W/2RTT**
- 平均流通量： **.75 W/RTT**

TCP Futures: TCP over "long, fat pipes"

- Example: 1500 byte segments, 100ms RTT, want 10 Gbps throughput
- Requires window size W = 83,333 in-flight segments
- Throughput in terms of loss rate:

$$\frac{1.22 \cdot MSS}{RTT\sqrt{L}}$$

- ➜ $L = 2 \cdot 10^{-10}$
- New versions of TCP for high-speed

TCP 的未來：TCP 壅塞控制演化了很長的時間

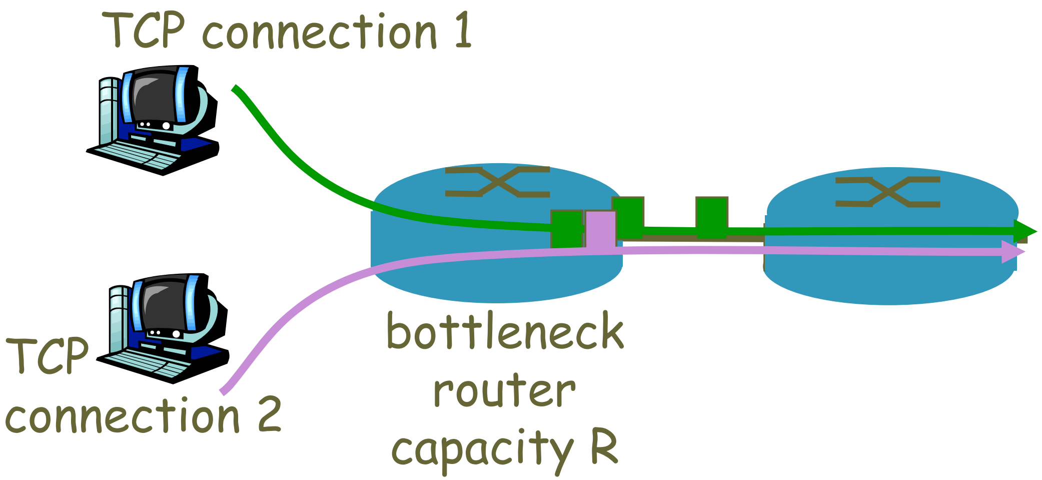
- 範例： **1500** 位元組資料分段、**100** 毫秒 **RTT**、想要達到 **10 Gbps** 的流通量
- 需要視窗大小 **W = 83,333** 傳輸的資料分段
  (10Gbps=(W/MSS)/RTT
  ➔W=10G/(MSS*RTT)
  =10G/(1500*8*100ms)=83333…)
- 以遺失率計算流通量：

$$\frac{1.22 \cdot MSS}{RTT\sqrt{L}}$$

- **➜ L = $2 \cdot 10^{-10}$** （很低的**loss rate**）
- 我們需要高速環境下的新版**TCP!**

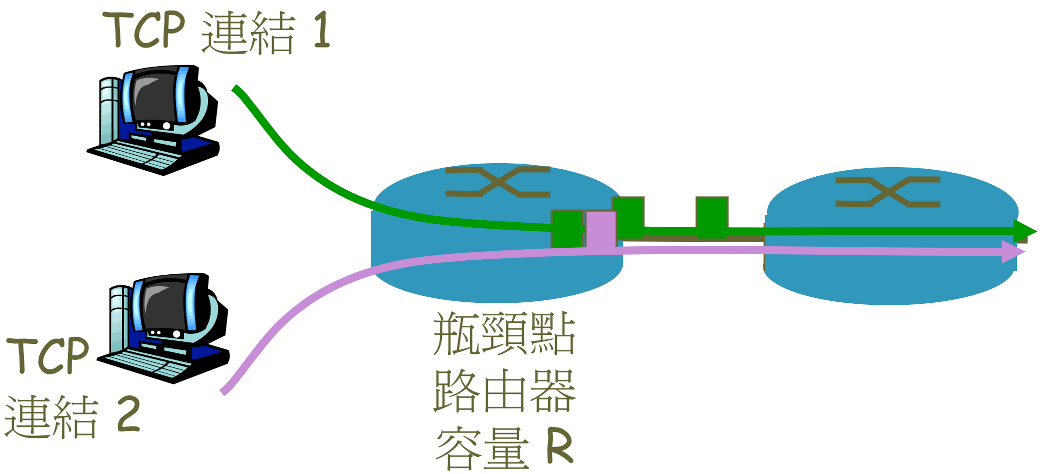
TCP Fairness

Fairness goal: if K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K



TCP 公平性

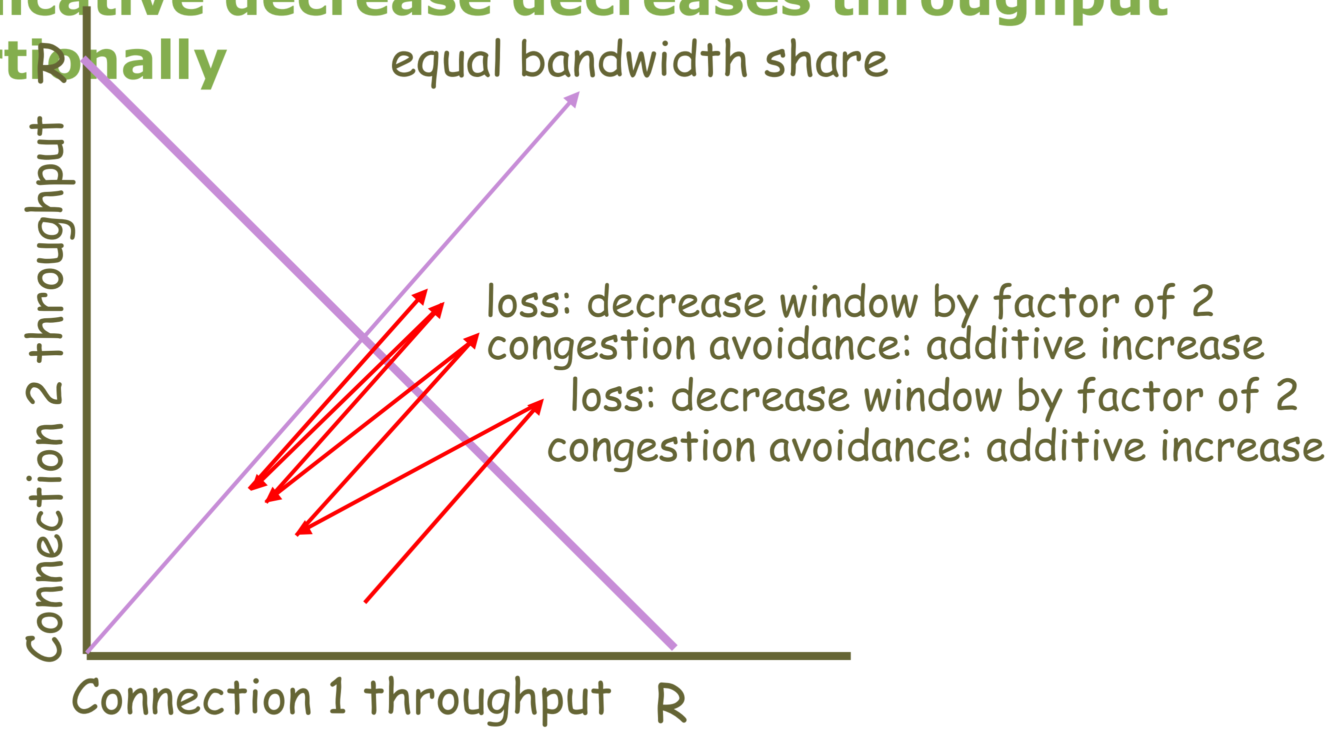
公平性目標： 假如有 **K** 條 **TCP** 會談連線、分享同一個瓶頸點連結的頻寬 **R**、每一個應該有 **R/K** 的平均速率



Why is TCP fair?

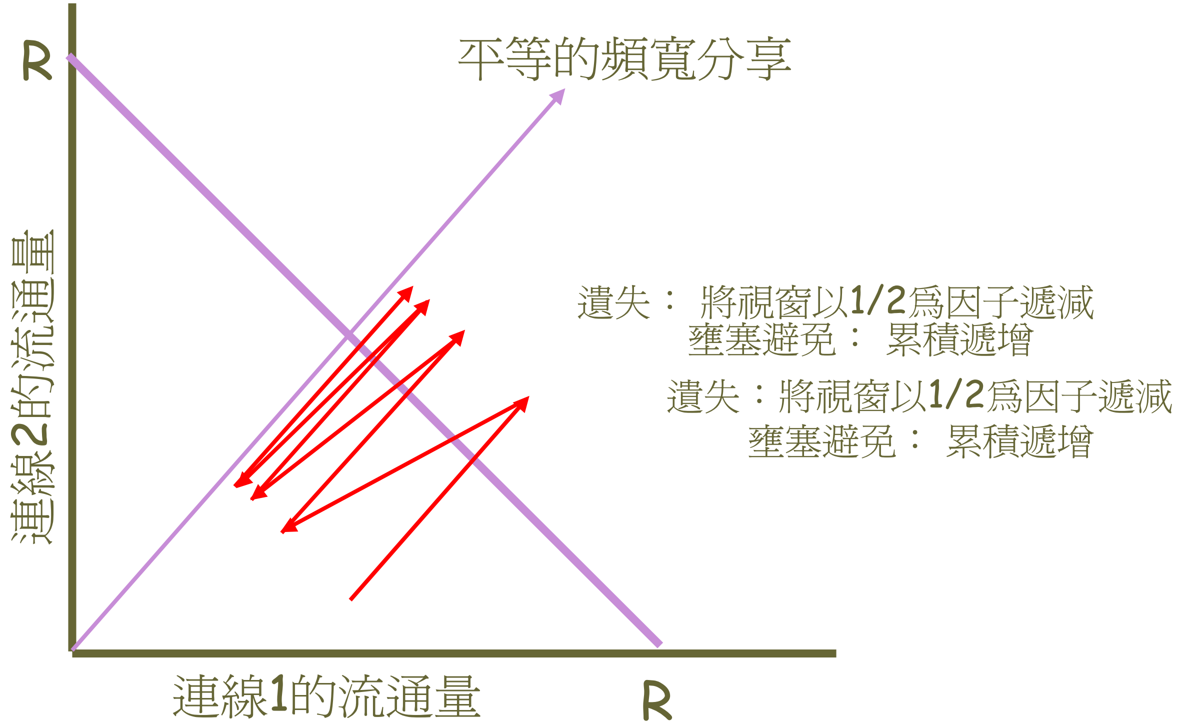
Two competing sessions:

- Additive increase gives slope of 1, as throughout increases
- Multiplicative decrease decreases throughput proportionally



TCP 為什麼公平?

- 兩個互相競爭的會談連線：
- 隨著流通量增加、累積遞增會導致 **1** 的斜率
- 倍數遞減會使得流通量成比例地遞減



Fairness (more)

Fairness and UDP

- **Multimedia apps often do not use TCP**
  - Do not want rate throttled by congestion control
- **Instead use UDP:**
  - Pump audio/video at constant rate, tolerate packet loss
- **Research area: TCP friendly**

Fairness and parallel TCP connections

- **Nothing prevents app from opening parallel connections between 2 hosts.**
- **Web browsers do this**
- **Example: link of rate R supporting 9 connections;**
  - New app asks for 1 TCP, gets rate R/10
  - New app asks for 11 TCPs, gets R/2 !

公平性 (更多)

- 公平性和 UDP
- 多媒體應用程式通常不會使用 TCP
  - 不想藉壅塞控制限制速率
- 使用 UDP 來取代：
  - 以固定速率將音訊/視訊送入網路、容忍封包遺失
- 研究領域： TCP 的友善性

- 公平性以及平行的TCP連結
- 無法防止應用程式在兩個主機間開啓平行的連線
- Web 瀏覽器會這樣做
- 範例： 速率 R的連結支援 supporting 9 個程式;
  - 新的應用程式要求 1 個 TCP、則得到 R/10 的速率
  - 新的應用程式要求 11 個 TCP、則得到 R/2 的速率！