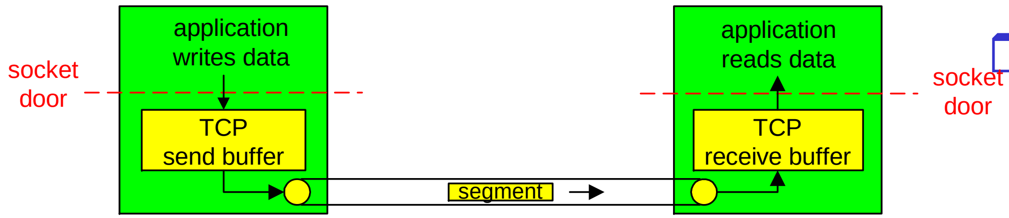
Chapter 3 outline

- 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

TCP: Overview TCP簡介

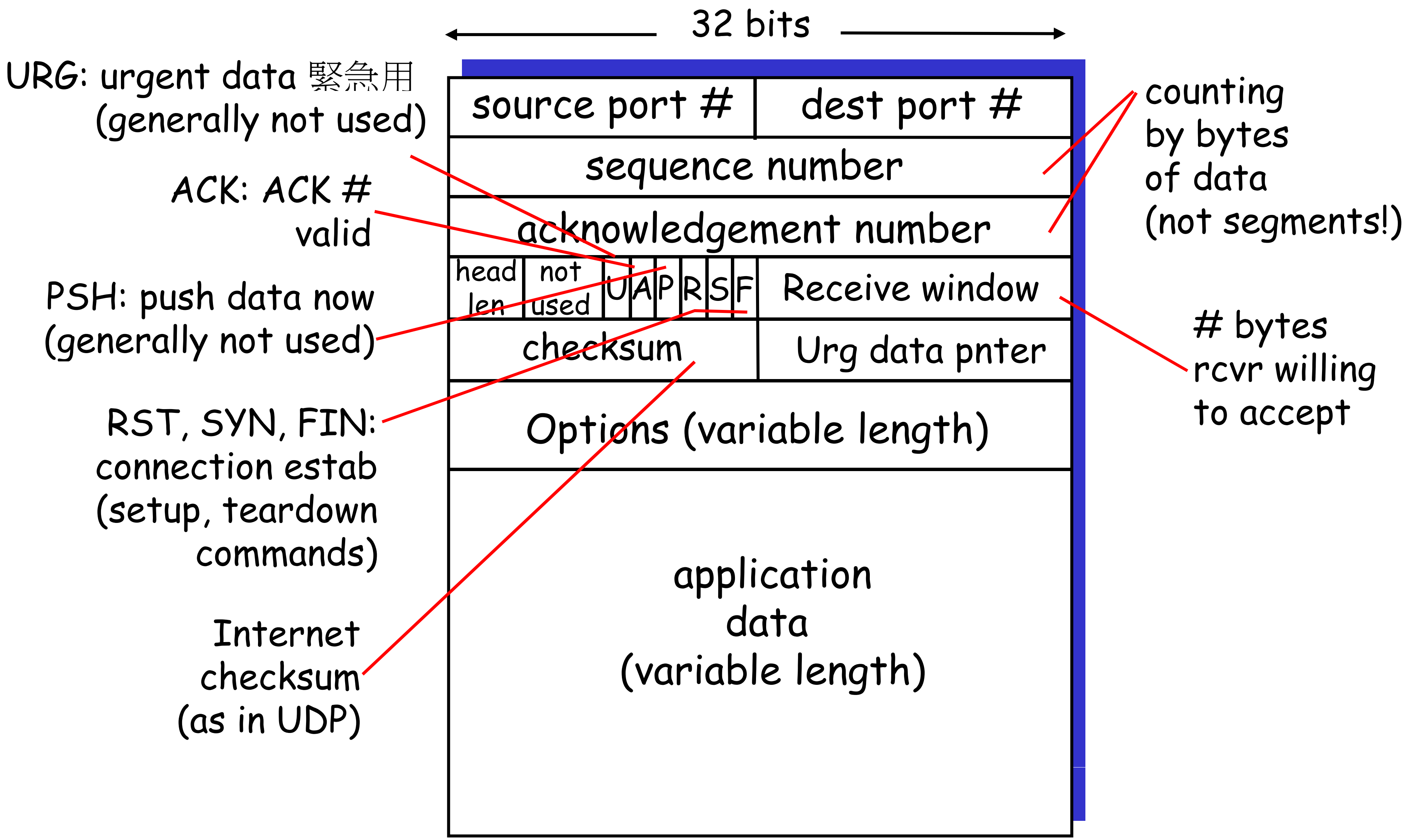
RFCs: 793, 1122, 1323, 2018, 2581

- point-to-point: 點對點
  - one sender, one receiver
- reliable, in-order *byte steam:* *可信賴的傳輸*
  - no "message boundaries"
- pipelined: 管線平行傳輸
  - TCP congestion and flow control set window size
- *send & receive buffers* *雙方都有buffer*



- full duplex data: 全雙工
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- connection-oriented: 連結導向
  - handshaking (exchange of control msgs) init's sender, receiver state before data exchange
- flow controlled: 流量控制
  - sender will not overwhelm receiver

TCP segment structure 區段結構



TCP seq. #'s and ACKs
TCP 序號及確認號碼

Seq. #'s:

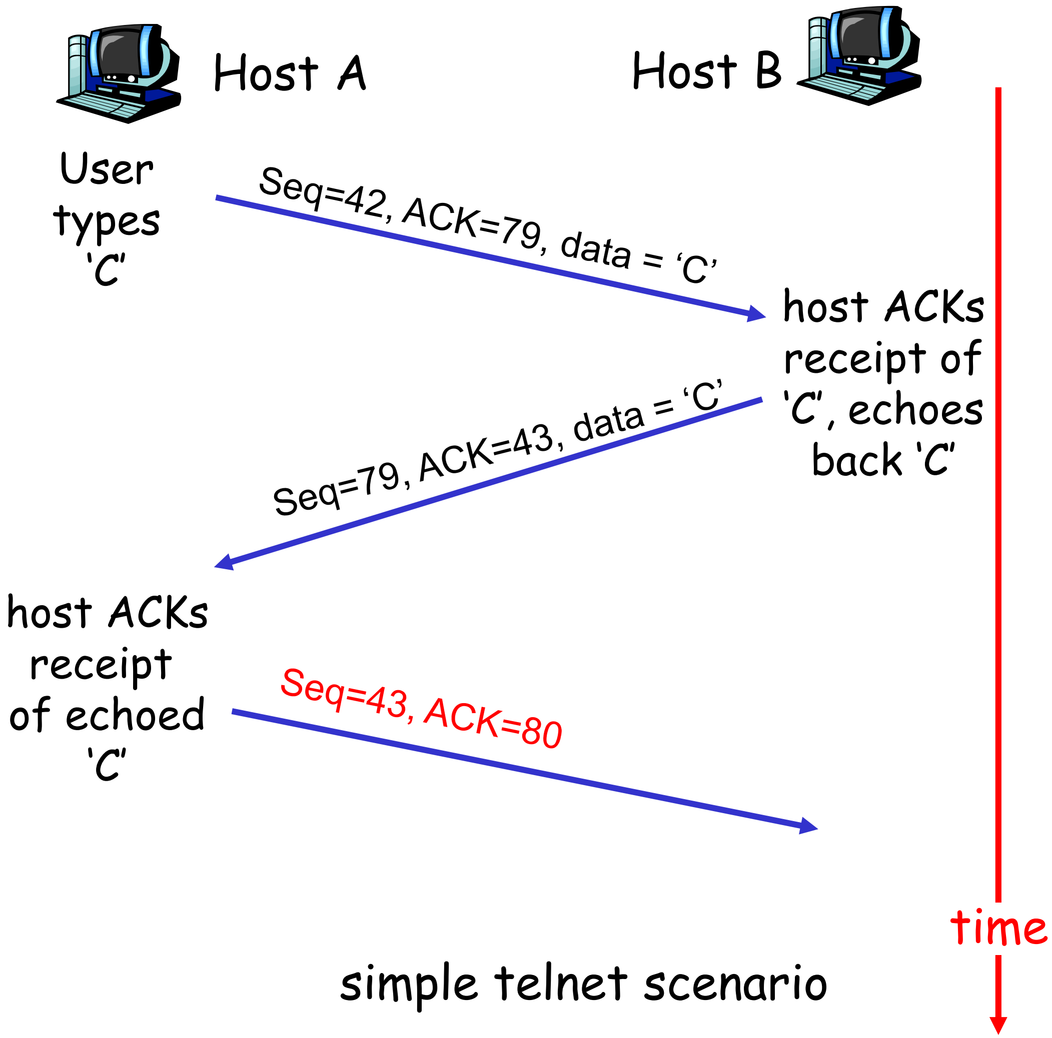
- byte stream "number" of first byte in segment's data
  第一個byte的編號

ACKs: 下一個預計收到的序號

- 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 Round Trip Time and Timeout

Q: how to set TCP timeout value?
如何決定TCP timeout

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

Q: how to estimate RTT?
如何估計RTT?

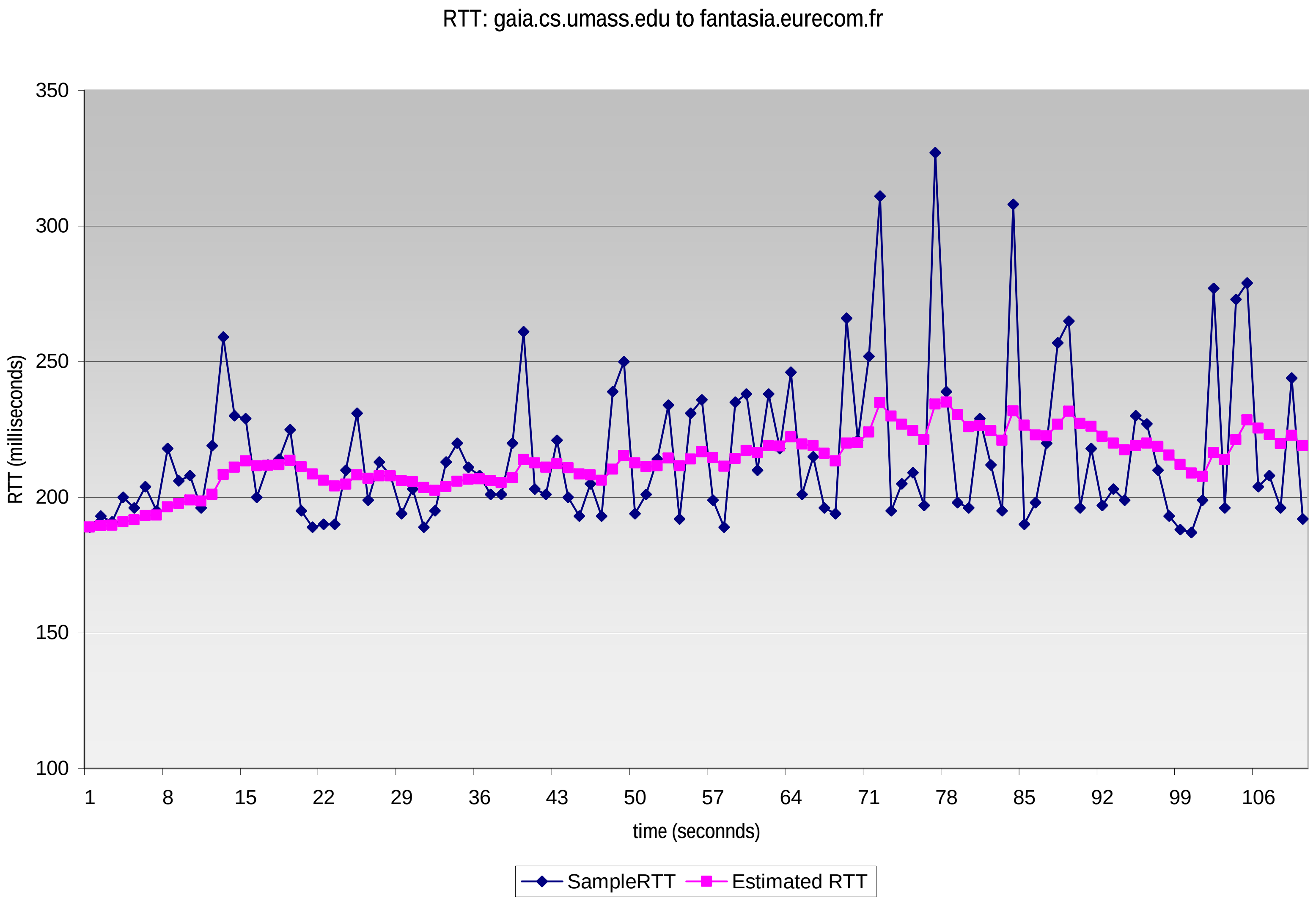
- **SampleRTT**: measured time from segment transmission until ACK receipt
  - ignore retransmissions
- **SampleRTT** will vary, want estimated RTT "smoother"
  - average several recent measurements, not just current **SampleRTT**

TCP Round Trip Time and Timeout

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

- Exponential weighted moving average
- influence of past sample decreases exponentially fast
- typical value: $\alpha = 0.125$

Example RTT estimation:



TCP Round Trip Time and Timeout

Setting the timeout 設定timeout值

- **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
```

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
    可信賴的資料傳輸
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

TCP reliable data transfer
TCP可信賴的資料傳輸

- TCP creates rdt service on top of IP's unreliable service
- Pipelined segments 管線平行傳輸
- Cumulative acks 累積式確認
- TCP uses single retransmission timer 單一重送計時器
- Retransmissions are triggered by: 重送
  - timeout events
  - duplicate acks
- Initially consider simplified TCP sender:
  - ignore duplicate acks
  - ignore flow control, congestion control

TCP sender events: 高度簡化的TCP

data rcvd from app:
- Create segment with seq # (亂數選擇)
- 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 重設

Ack rcvd:
- If acknowledges previously unacked segments
  - update what is known to be acked 最後未被確認的segment
  - start timer if there are outstanding segments

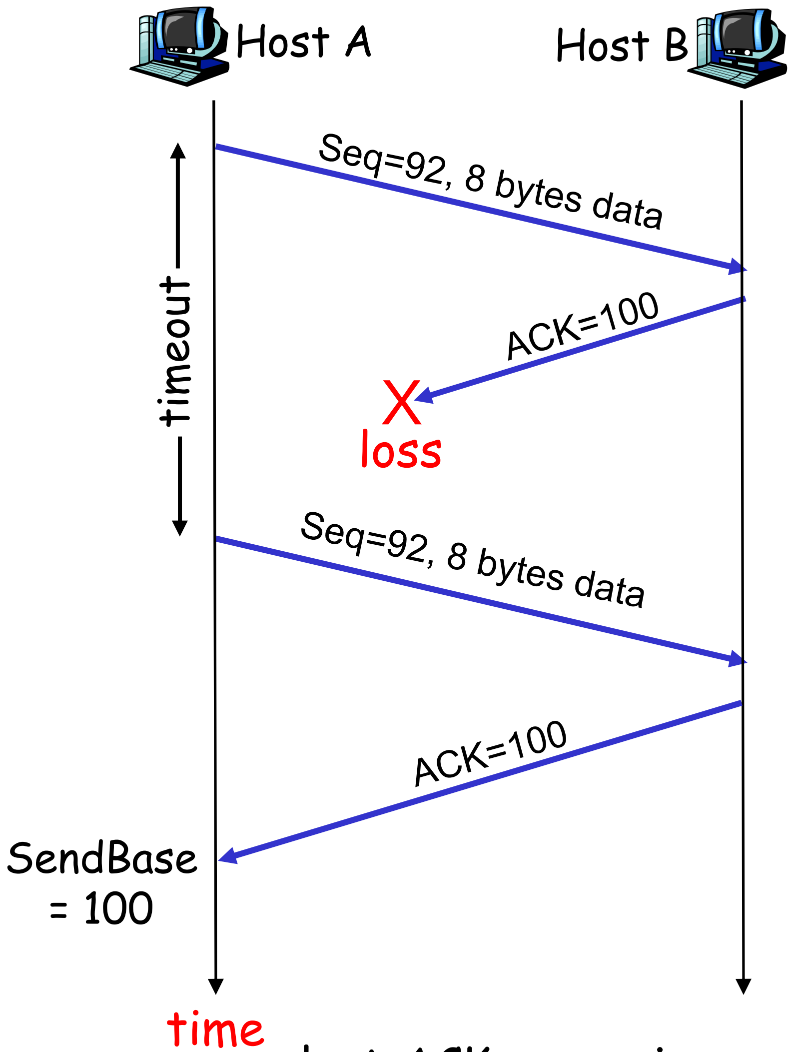
```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

 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 */
```

TCP sender (simplified)

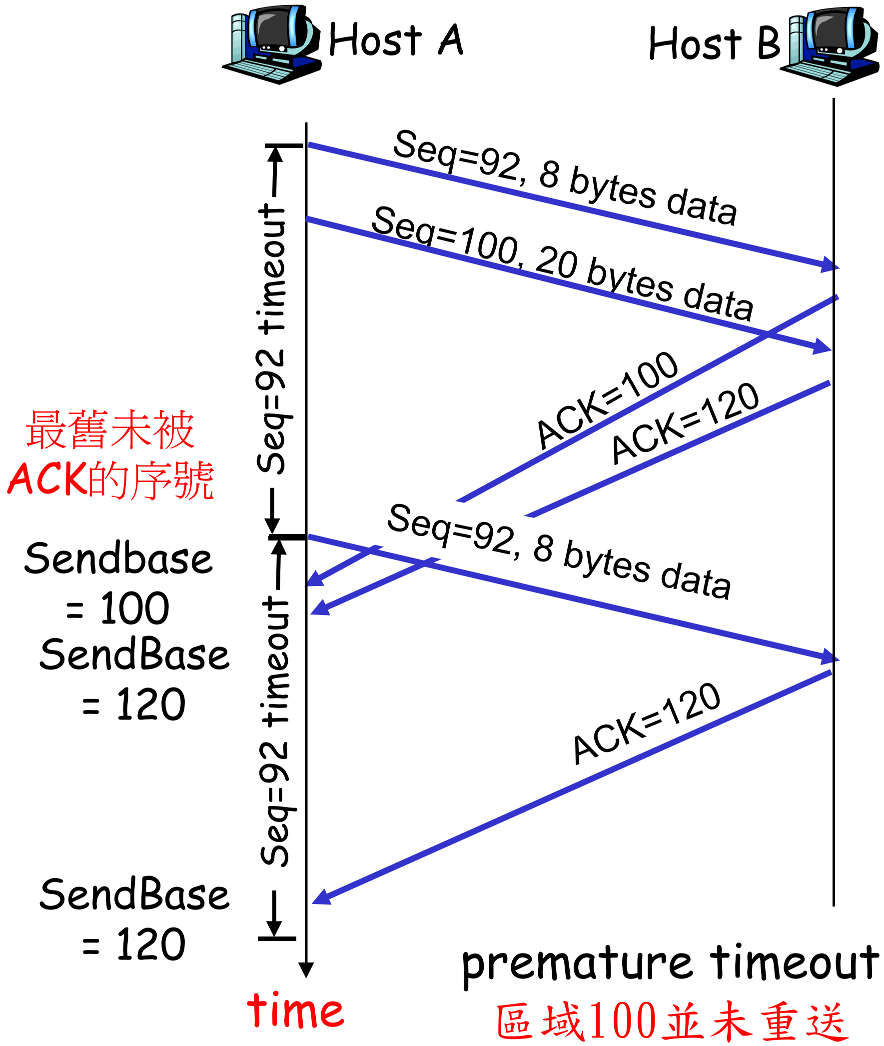
Comment:
- SendBase-1: last cumulatively ack'ed byte

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

TCP: retransmission scenarios



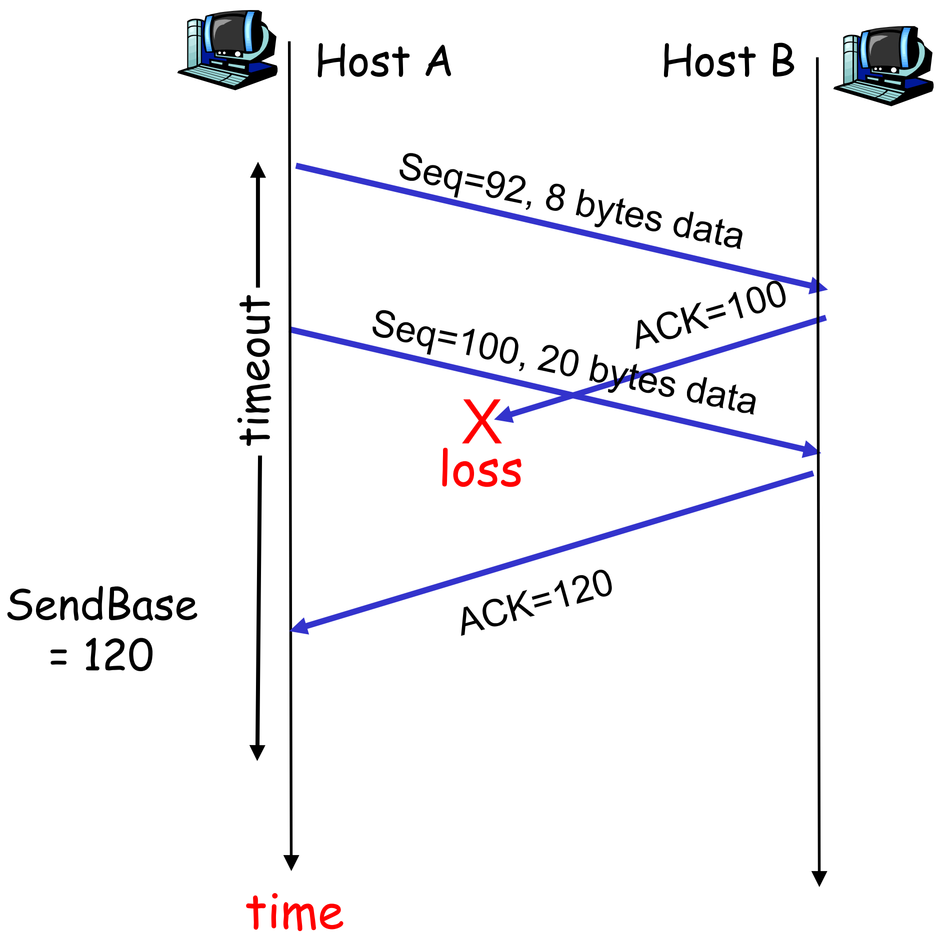
lost ACK scenario



premature timeout

區域100並未重送

TCP retransmission scenarios (more)



Cumulative ACK scenario

累積式確認可避免多餘的重送

Fast Retransmit 快速重送

- 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 duplicated ACKs for the same data, it supposes that segment after ACKed data was lost:
  - fast retransmit: resend segment before timer expires

重覆收到三個一樣的ACK時執行快速重送！

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

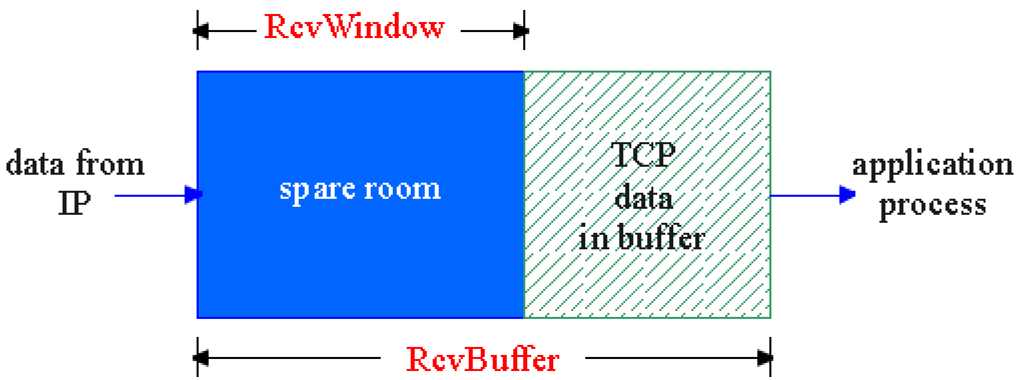
fast retransmit

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
    流量控制
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

TCP Flow Control 流量控制

- receive side of TCP connection has a receive buffer: 接收端緩衝區



- app process may be slow at reading from buffer

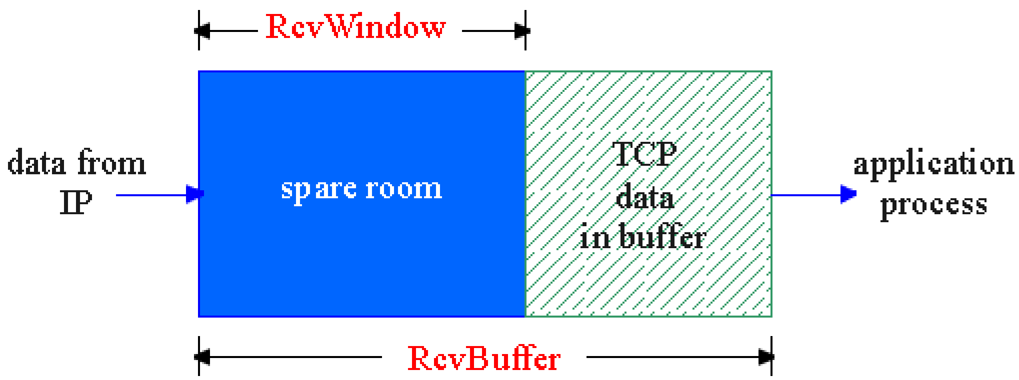
flow control

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

傳送端會依接收端的緩衝區情形決定傳送速度

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

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

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 連線管理
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

TCP Connection Management 連線管理

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:

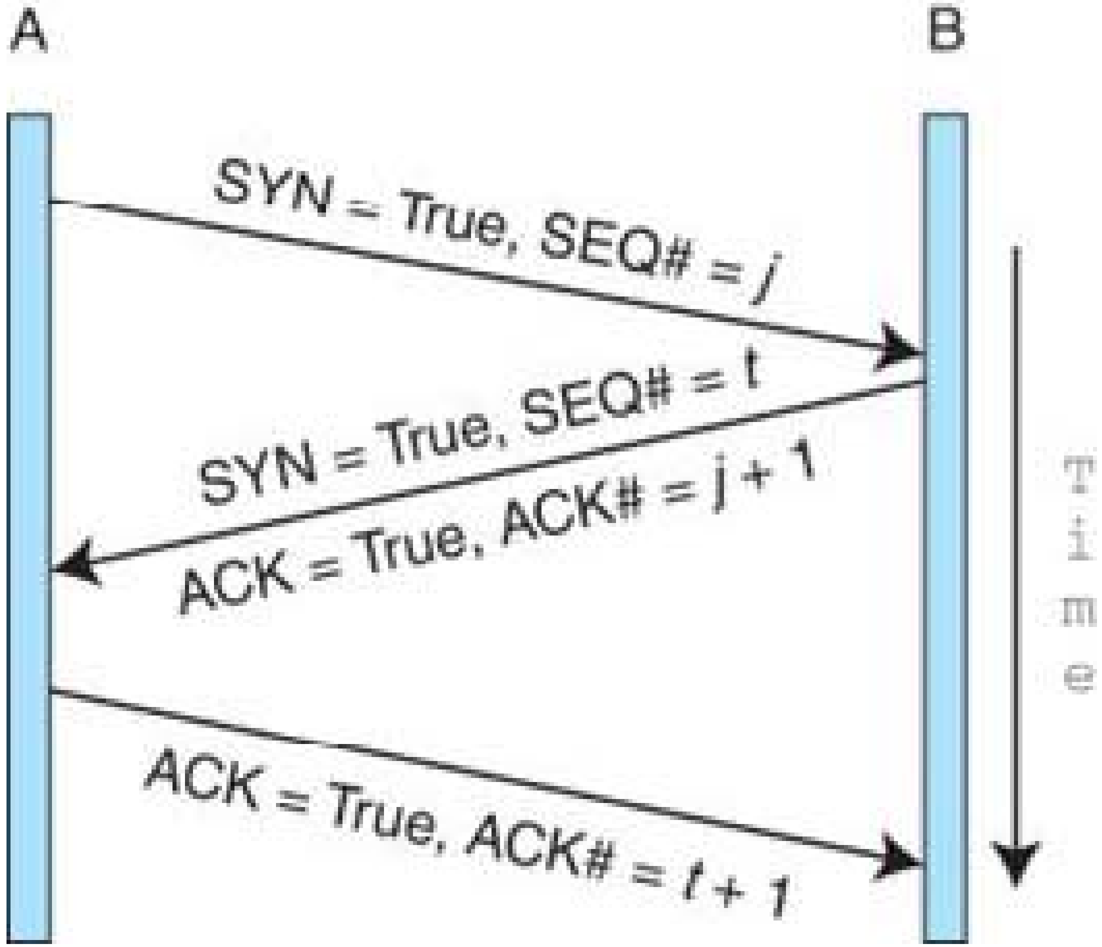
三方握手

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

Three Way Handshake



TCP Connection Management (cont.)

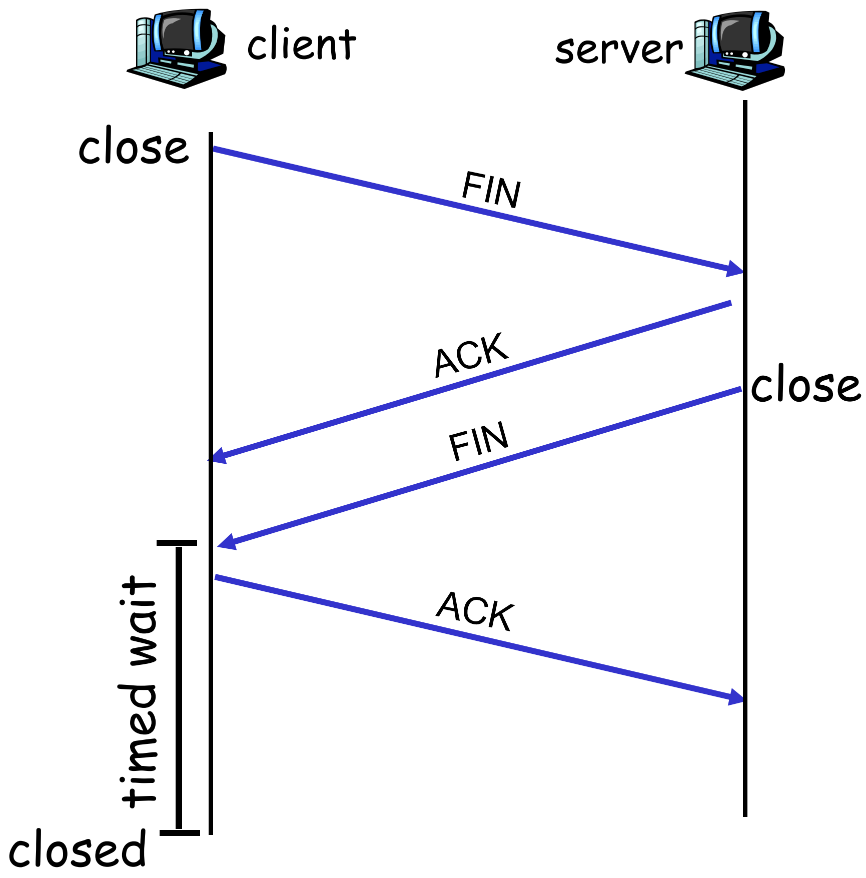
Closing a connection:

關閉連線

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.



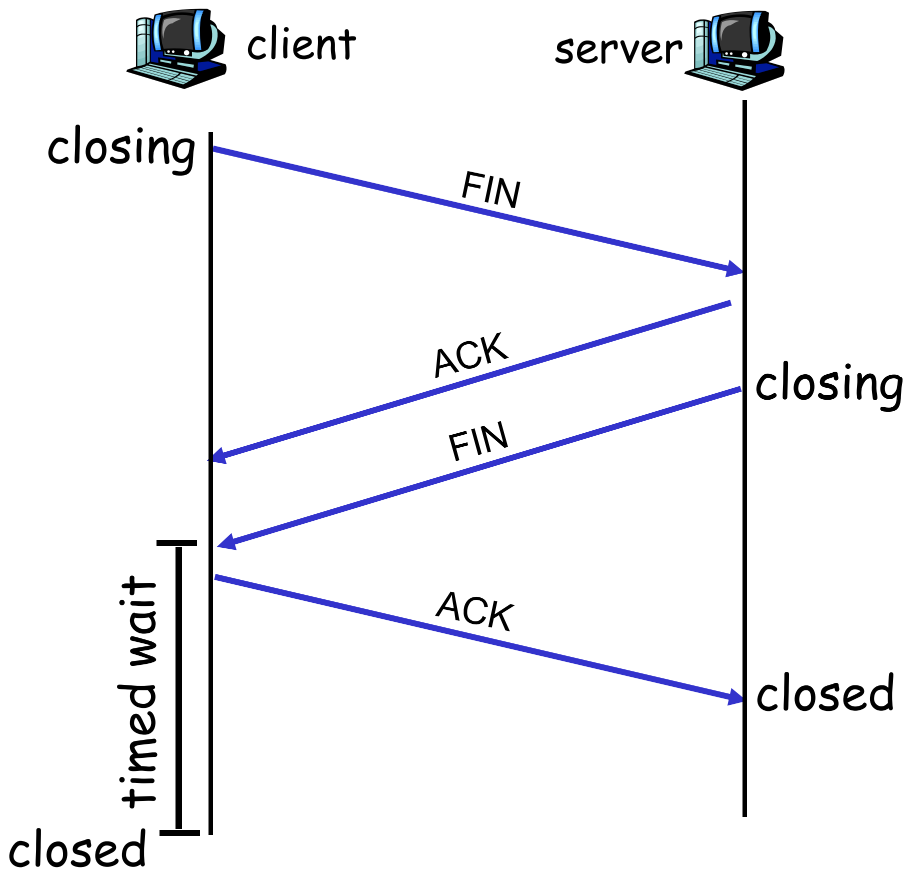
TCP Connection Management (cont.)

Step 3: client receives FIN, replies with ACK.

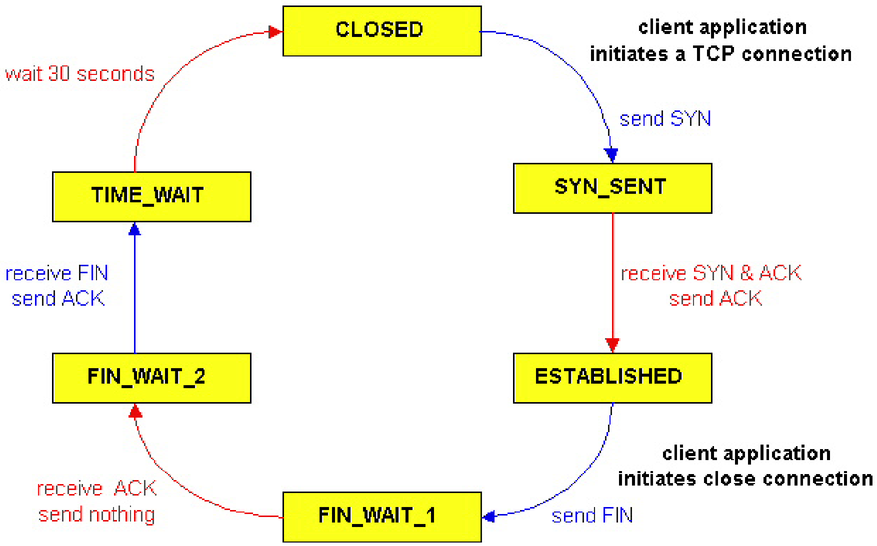
- Enters "timed wait" - will respond with ACK to received FINs

Step 4: server, receives ACK. Connection closed.

Note: with small modification, can handle simultaneous FINs.



TCP Connection Management (cont)



TCP client lifecycle

TCP server lifecycle