計算機網路概論

傳輸層通訊協議

© All rights reserved. No part of this publication and file may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without prior written permission of Professor Nen-Fu Huang (E-mail: nfhuang@cs.nthu.edu.tw).

大綱

- 端對端通訊協議介紹
- 簡易反多工協議 (如 UDP)
- 可靠位元組串流協議 (如 TCP)

端對端通訊協議

- 傳輸層通訊協議通常期望能提供
  - 保證的訊息傳遞
  - 訊息傳遞順序與發送順序相同
  - 一個訊息最多只會傳遞一份
  - 支援任意大小的訊息
  - 支援發送者與接收者的同步
  - 允許接收者對發送者執行流量控制
  - 支援一台主機上執行多個不同應用程式

端對端通訊協議

- 傳輸層下層的網路層服務 (如:IP網路) 典型限制:
  - 丟棄訊息
  - 重新排序訊息
  - 傳遞多份相同訊息
  - 傳遞的訊息有大小限制
  - 訊息傳遞可能有任意長短的時間延遲

  → 不可靠服務

端對端通訊協議

- 傳輸層通訊協議的挑戰
  - 設計演算法來將下層不可靠的網路層服務變成應用程式所需的服務
  - *不可靠服務* → *不可靠服務* **(UDP)**
  - *不可靠服務* → *可靠服務* **(TCP)**

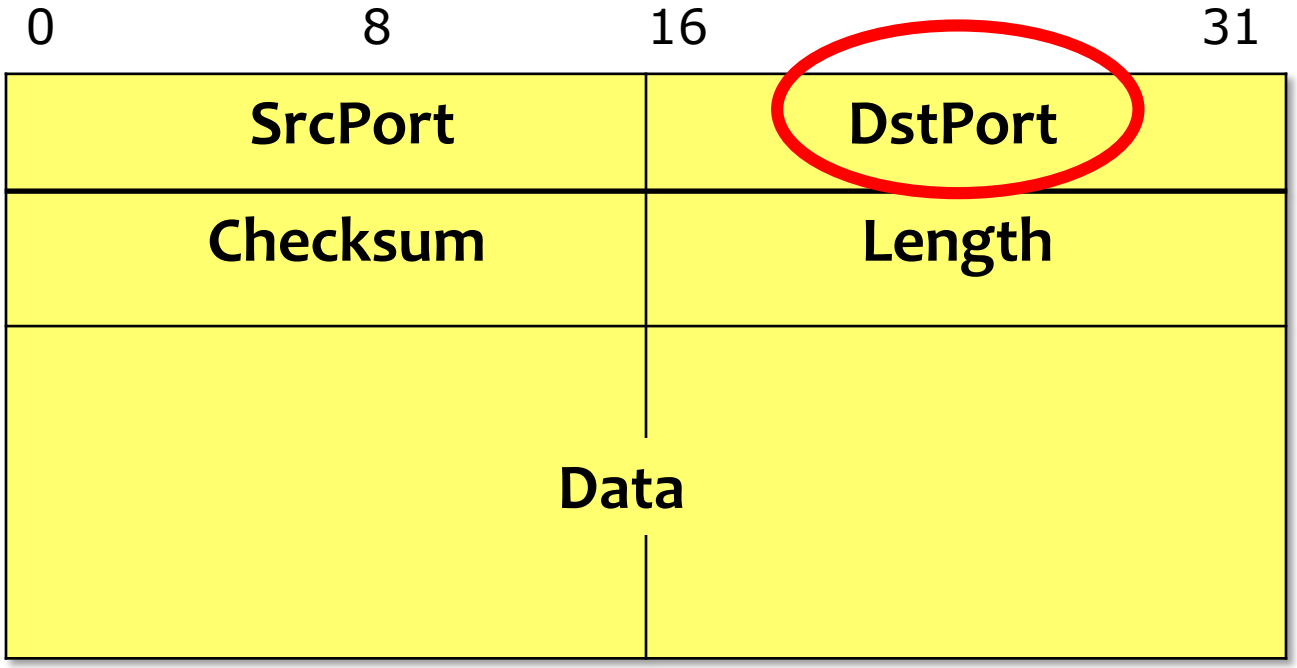
大綱

- 端對端通訊協議介紹
- 簡易反多工協議 (如 UDP)
- 可靠位元組串流協議 (如 TCP)

簡易反多工器 (UDP)

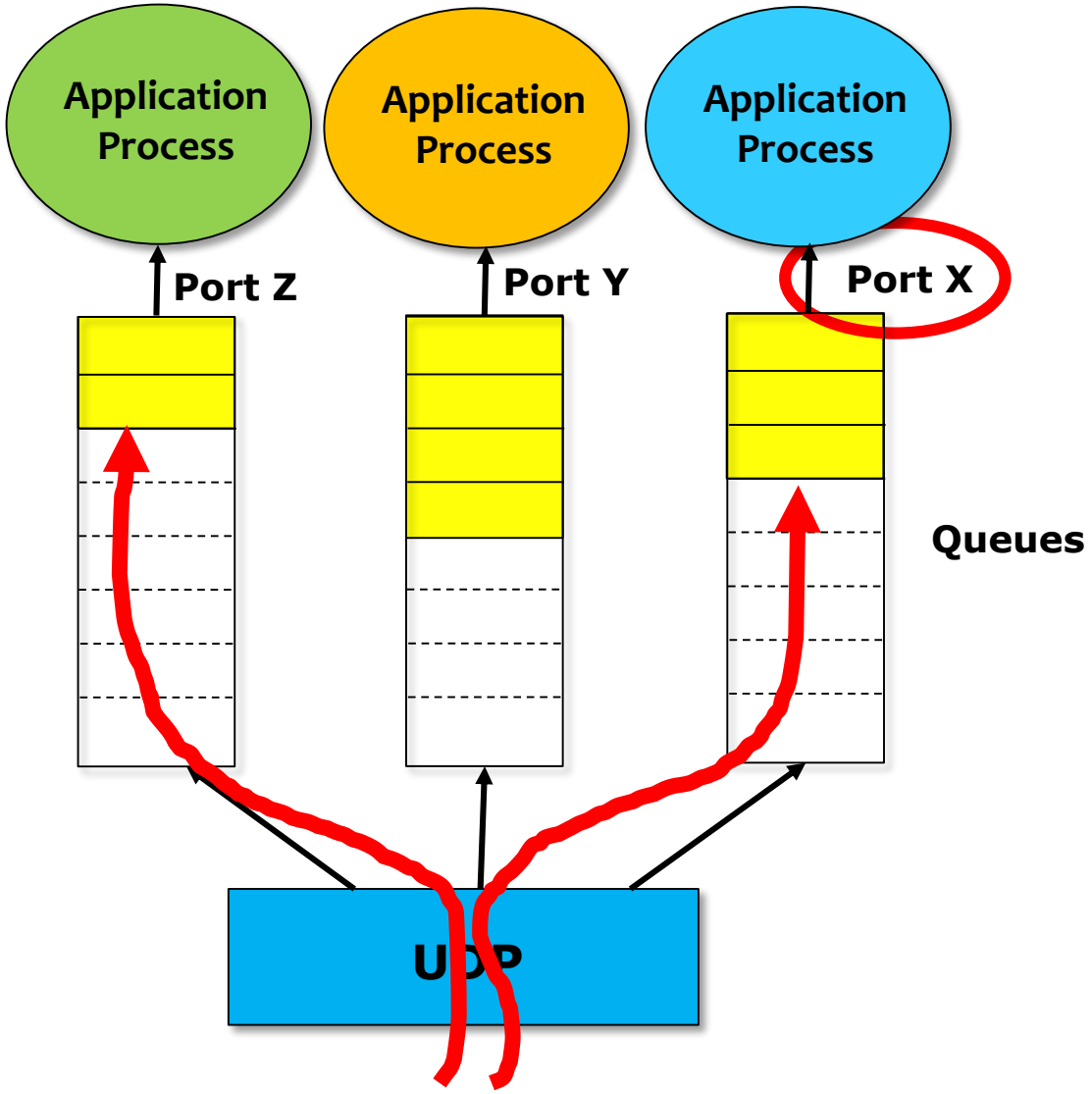
- 將下層的主機到主機遞送傳輸服務擴展成 程序對程序的通訊服務
- 增加一層反多工(demultiplexing)，讓在同一個主機上的多個應用程式可共享網路.

簡易反多工器 (UDP)



UDP 標頭格式

簡易反多工器 (UDP)



UDP 封包反多工器

大綱

- 端對端通訊協議介紹
- 簡易反多工協議 (如 UDP)
- 可靠位元組串流協議 (如 TCP)

可靠位元組串流協議 (TCP)

- 相較於 UDP，傳輸控制通訊協議 (Transmission Control Protocol，即TCP) 提供以下服務
  - 可靠的服務 (Reliable)
  - 連線導向(Connection oriented)
  - 字元組串流服務(byte-stream service)

流量控制 VS 壅塞控制

- 流量控制(Flow control)用來預防發送者傳送過多的流量使接收者超出可負荷的容量
- 壅塞控制(Congestion control) 用來預防網路被注入過多的資料而造成路由器/交換器或鏈路超出負荷

端對端問題

- TCP 運行於互聯網上，非於一個點對點的鏈結上
- TCP 滑動視窗(sliding window)演算法必須考慮:
  - TCP 支援互聯網上不同主機上應用程式間建立邏輯連線(logical connections)
  - 不同 TCP 連線的 RTT 時間可能差異極大
  - 封包在互聯網上傳遞時可能重新排序

端對端問題

- TCP 需要一個機制讓一個連線的兩端皆可得知另一端提供什麼資源給該連線
- TCP 需要一個機制讓發送端得知網路的負荷量

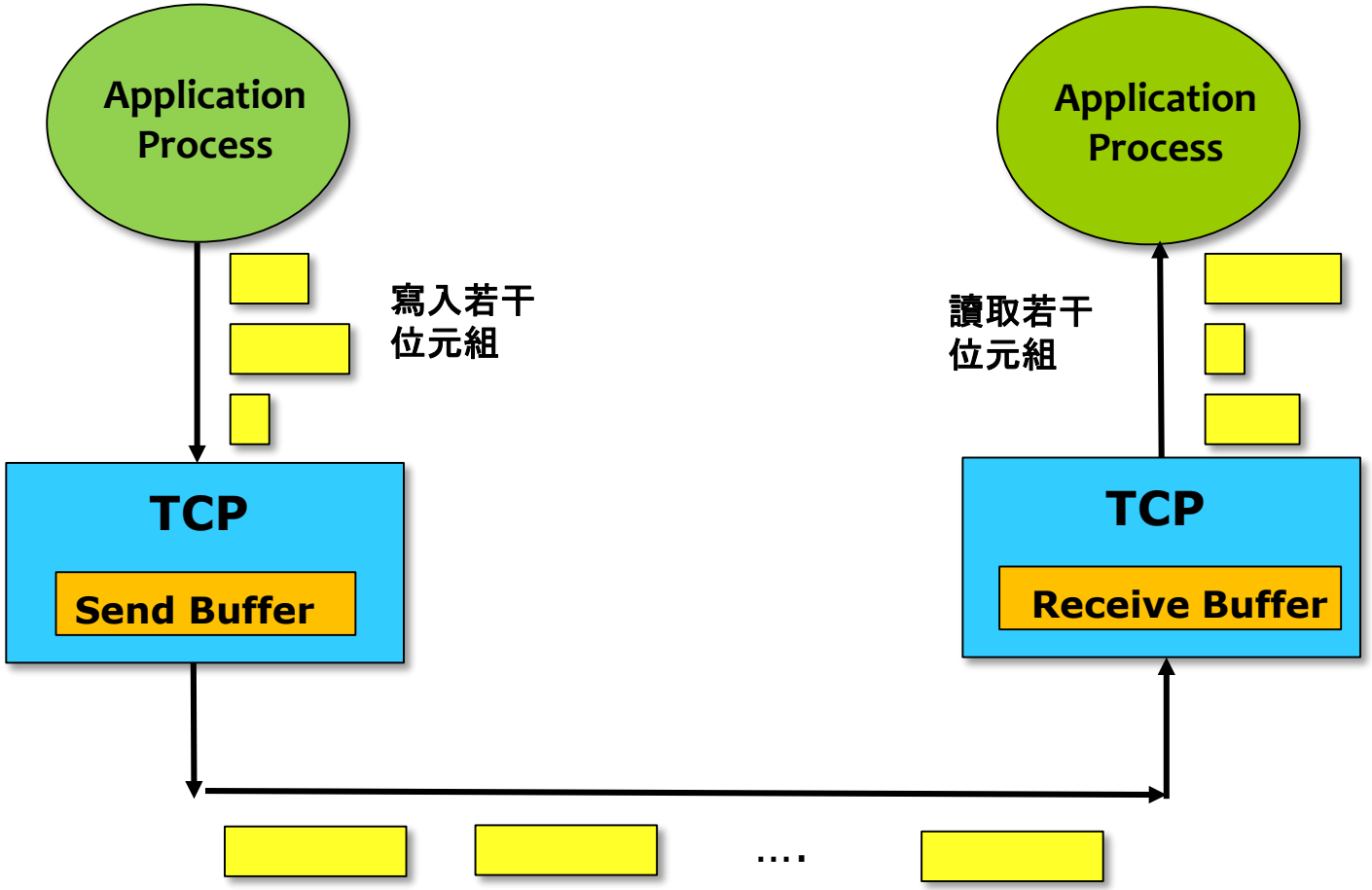
TCP 資料段

- TCP 是一個位元組導向通訊協議
- 傳送端將若干位元組寫入一個 TCP 連線，接收端則從該 TCP 連線讀取位元組
- 然而， TCP 不會在互聯網上傳輸各別的位元組

TCP 資料段

- TCP 傳送端將應用程式傳送的位元組先儲存起來, 等位元組數量形成一個合理大小的資料段，再將此資料段送至 TCP 的目的端主機
- TCP目的端會將資料段內容放入接收緩衝器，接收端應用程式再由此緩衝器讀取位元組
- 在 TCP 連線上傳送的封包稱為 資料段 (*segments*)

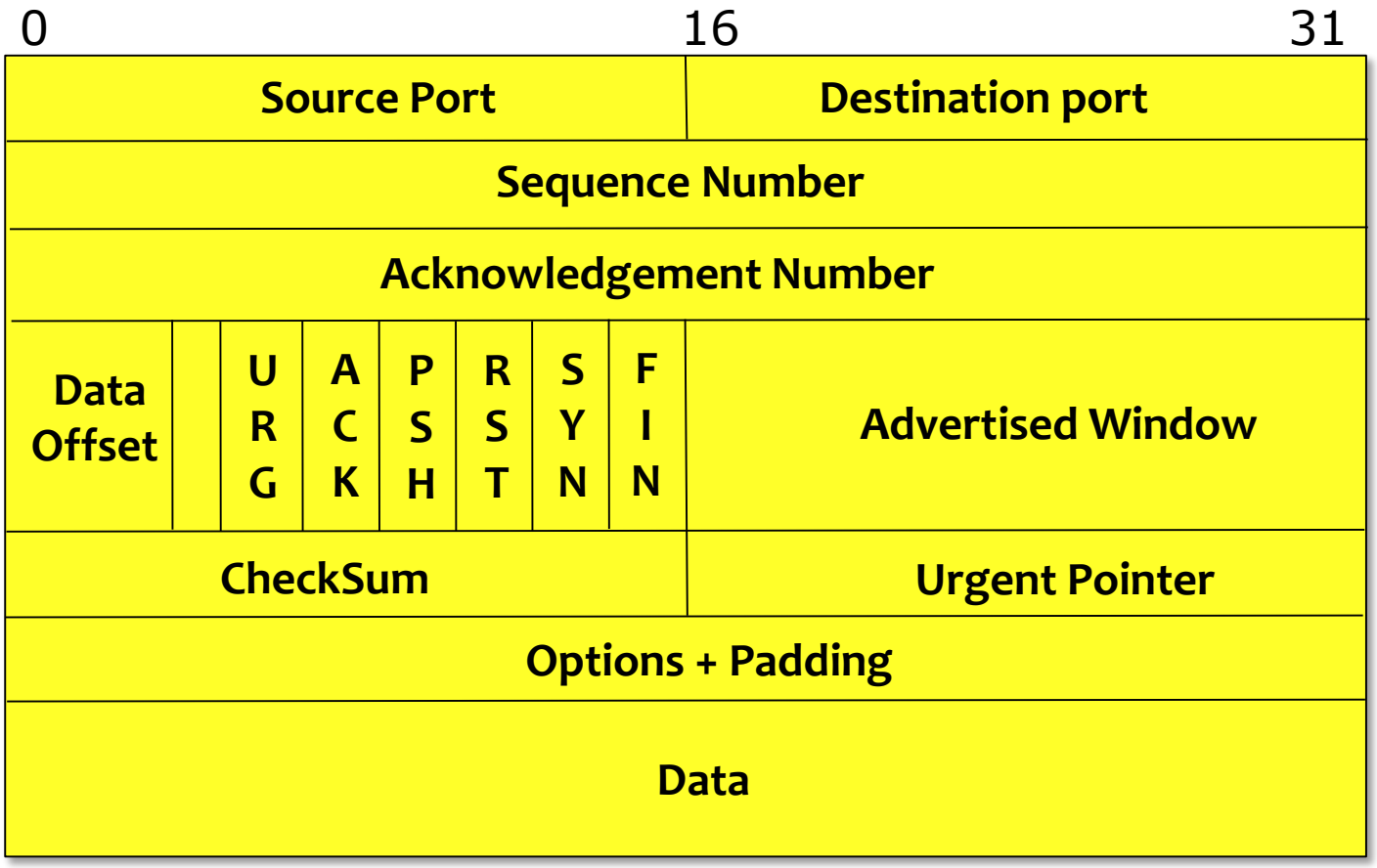
TCP 資料段



傳輸資料段

TCP 控管位元組串流的方式

TCP 標頭



TCP 標頭 格式

TCP 標頭

- SrcPort 與 DstPort 欄位分別代表來源端口號(source port)與目的端口號(destination port)
- Acknowledgment、SequenceNum與AdvertisedWindow欄位用於TCP滑動視窗演算法
- 因為TCP為位元組導向協議，資料的每個位元組皆有一個序號；而SequenceNum 欄位為此資料段上的第一個資料位元組的序號
- Acknowledgment 與 AdvertisedWindow 欄位攜帶的是反方向上資料傳輸的流量控制相關資訊

TCP 標頭

- 6-位元旗標欄位用來傳遞 TCP 連線兩端之間的控制訊息
- 可能的旗標有 SYN、FIN、RESET、PUSH、URG 與 ACK
- SYN 與 FIN 旗標分別於建立與中止TCP 連接時使用
- ACK 旗標 = 1 表示 Acknowledgment 欄位上的值是有效的，表示接收者應該注意此欄位值

TCP 標頭

- URG 旗標表示此資料段包含緊急資料。當此旗標被設為 1 ， UrgPtr 欄位值代表此資料段上的非緊急資料的起始位置
- 因緊急資料會放在資料段主體的前面部分，由這裡一直到 UrgPtr 所指的位置之間皆為緊急資料
- PUSH 旗標表示發送者啟動推擠動作，會指示TCP接收端應通知其接收應用程式有此動作

TCP 標頭

- RESET 旗標表示接收端因為困惑於接收到一個無預期接收的資料段—因此想要中止此連線
- 最後，Checksum 檢查碼欄位 使用方式跟在UDP的一樣— 它由TCP Header、TCP data 與 pseudoheader (由來源端位址、目地端位址與 IP header長度組成) 計算得出

TCP 連線管理

- TCP 傳送端與接收端在交換資料段之前會先建立“連線”
- TCP 連線初始變數:
  - 序號
  - 緩衝器, 流量控制訊息 (e.g. RcvWindow)
- *Client*: 連線啟動者

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

- *Server*: 等待 client 接觸

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

TCP 連線管理

三向交握 (Three-way handshake):

Step 1: Client 發送 TCP **SYN** 資料段 給 server

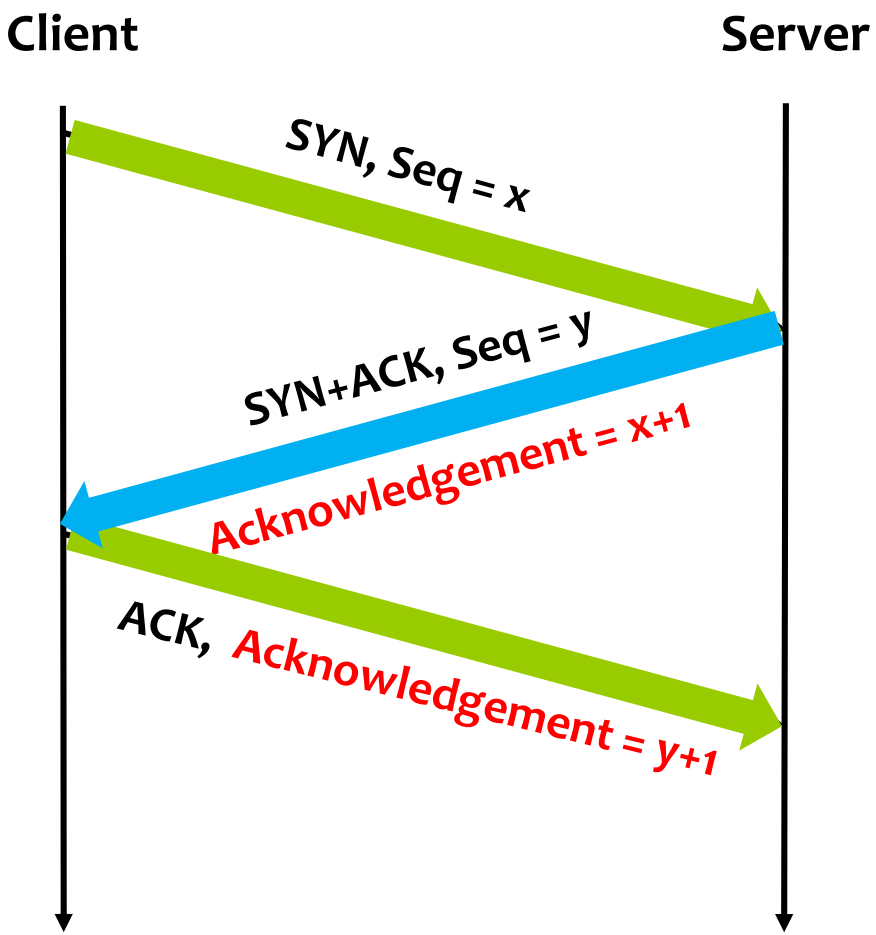
- 註明一開始的序號
- 不包含資料

Step 2: Server 收到 SYN，回傳**SYN/ACK** 資料段

- server 配置緩衝器
- 註明 server 一開始的序號

Step 3: client 收到 SYN/ACK，回傳可能帶有資料的**ACK** 資料段

TCP 連線建立



三向交握 演算法的時間軸圖

TCP 連線管理 (接續)

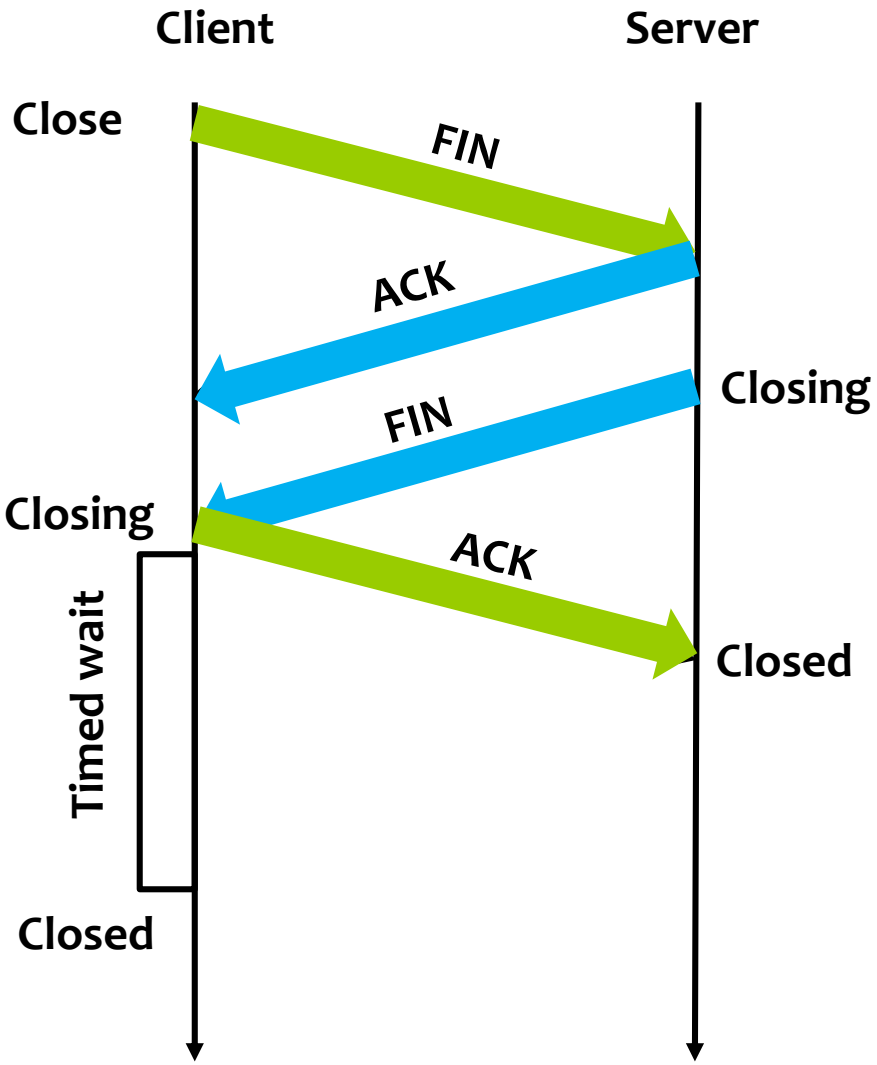
關閉連接:

client 關閉 socket:

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

Step 1: Client 發送 TCP FIN 控制資料段給 server

Step 2: Server 收到 FIN，回傳 ACK。關閉連線，發送FIN。



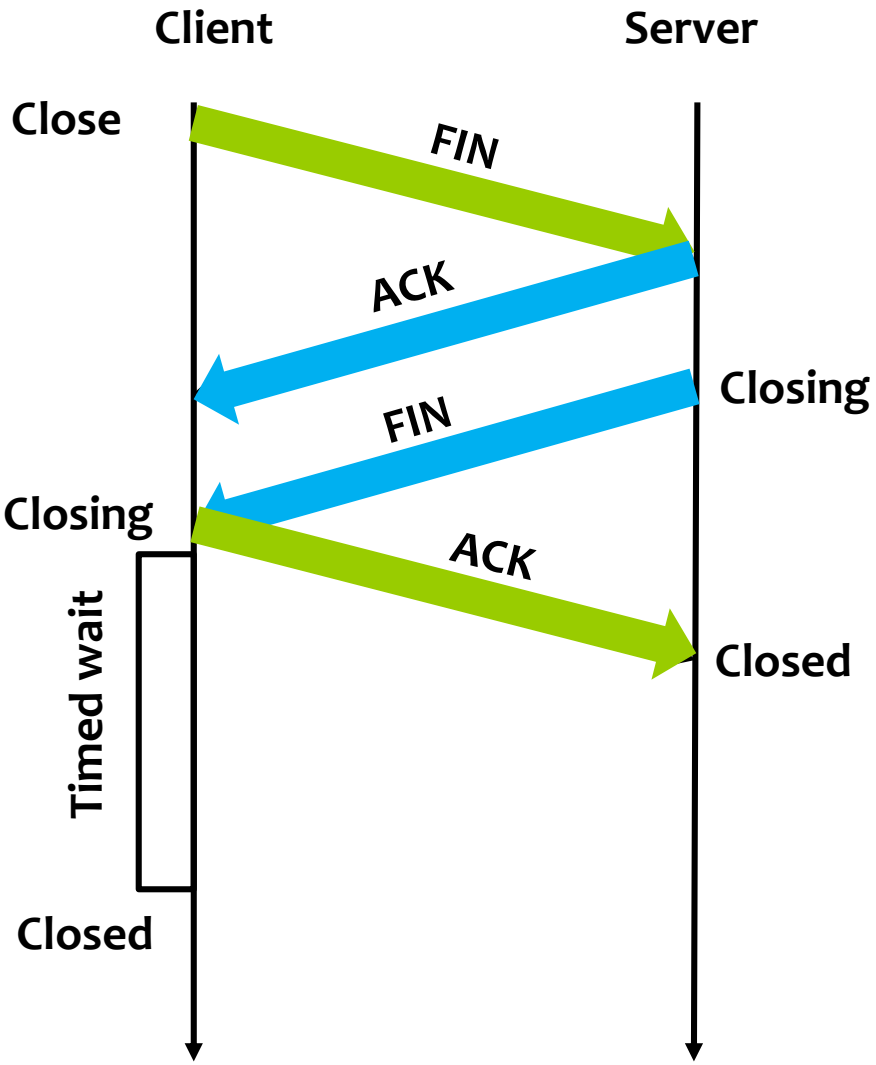
TCP 連線管理 (接續)

Step 3: Client 接收 **FIN**，回傳 **ACK**。

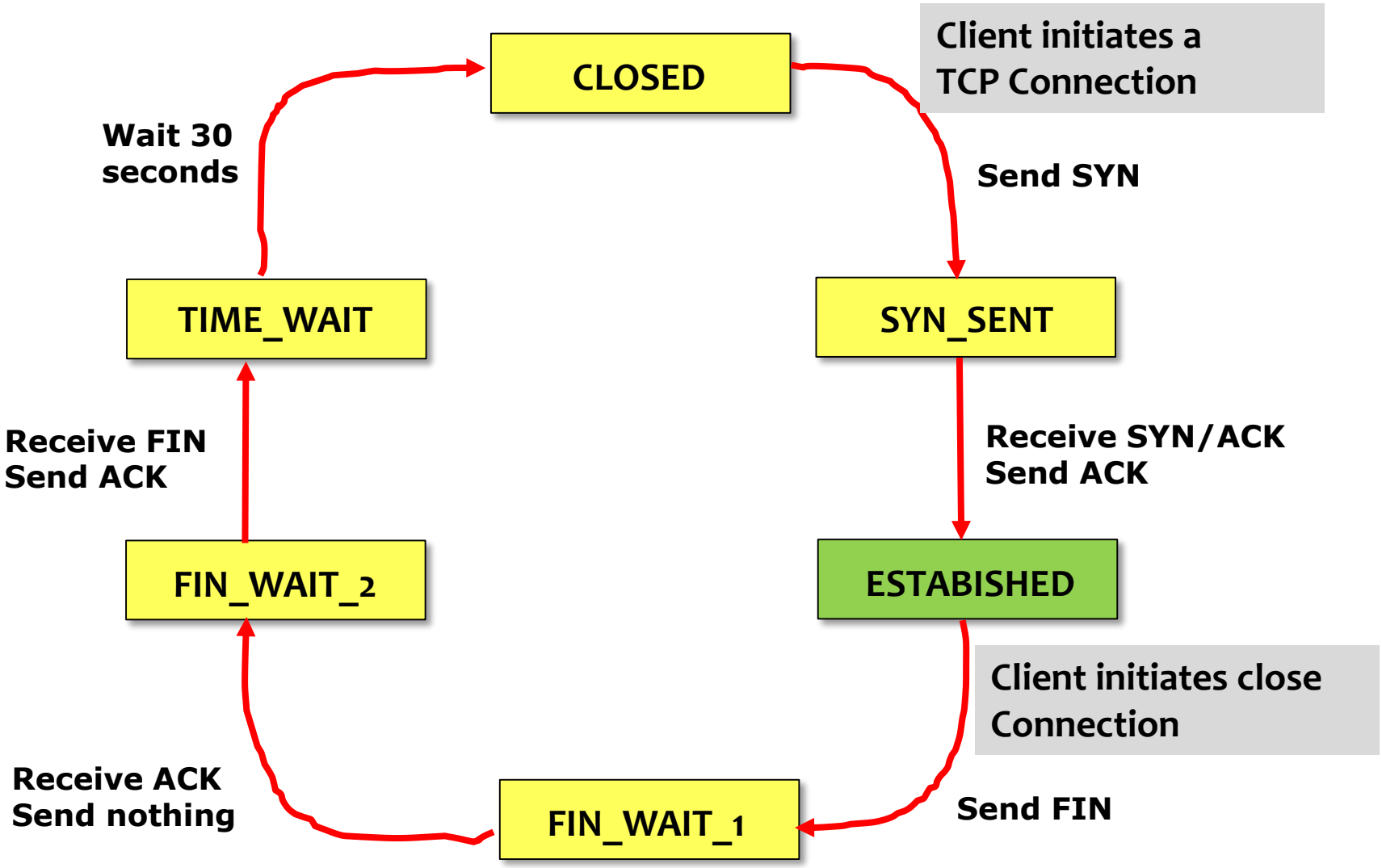
- 進入 **"timed wait" –**當接收到 **FINs** 將回傳 **ACK**

Step 4: Server 接收**ACK** ，連線關閉。

Note: 做一些小更改後，可處理同時多個 **FINs**

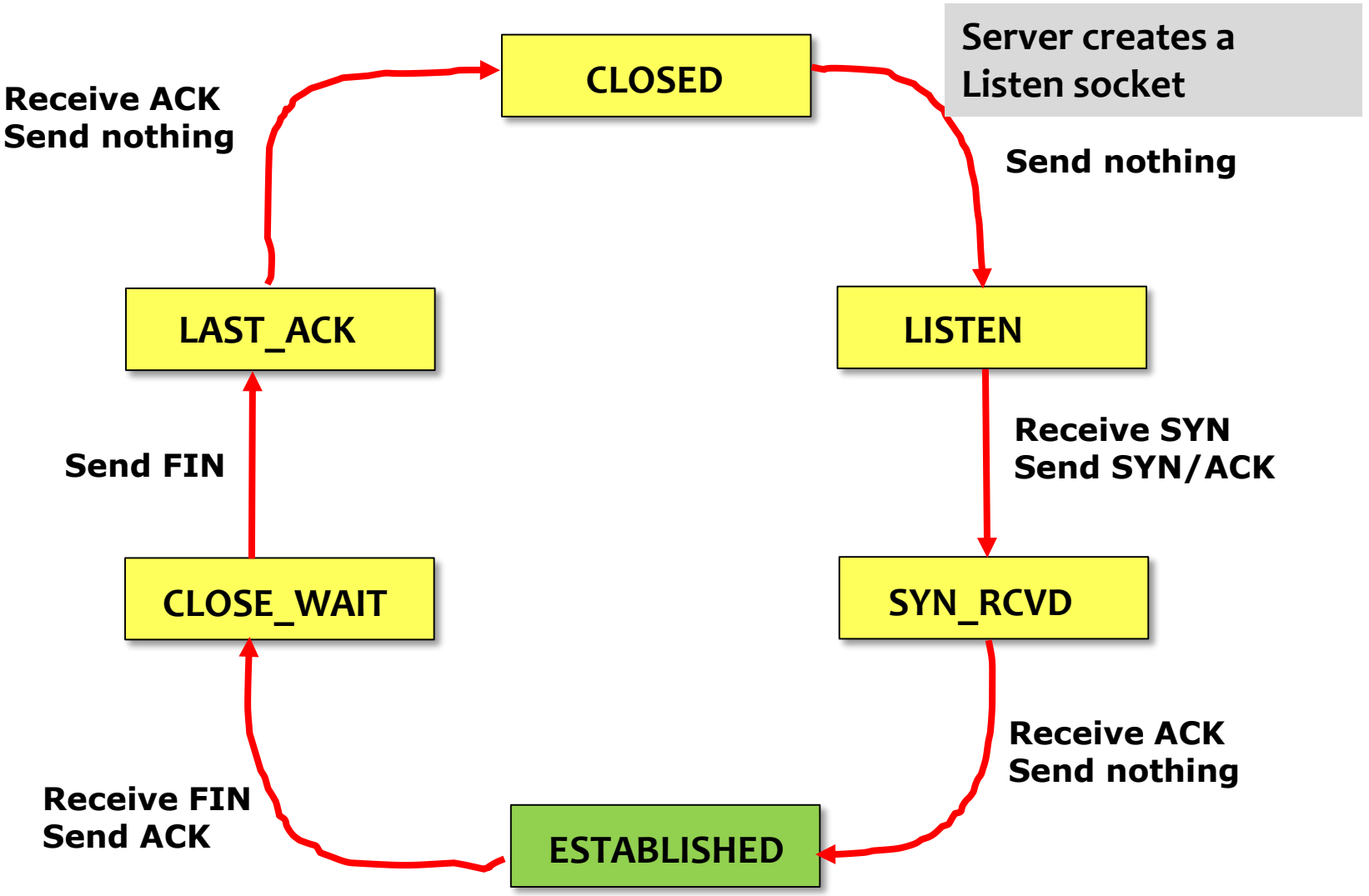


TCP 連線管理 (接續)



TCP client 狀態圖

TCP 連線管理 (接續)



TCP server 狀態圖

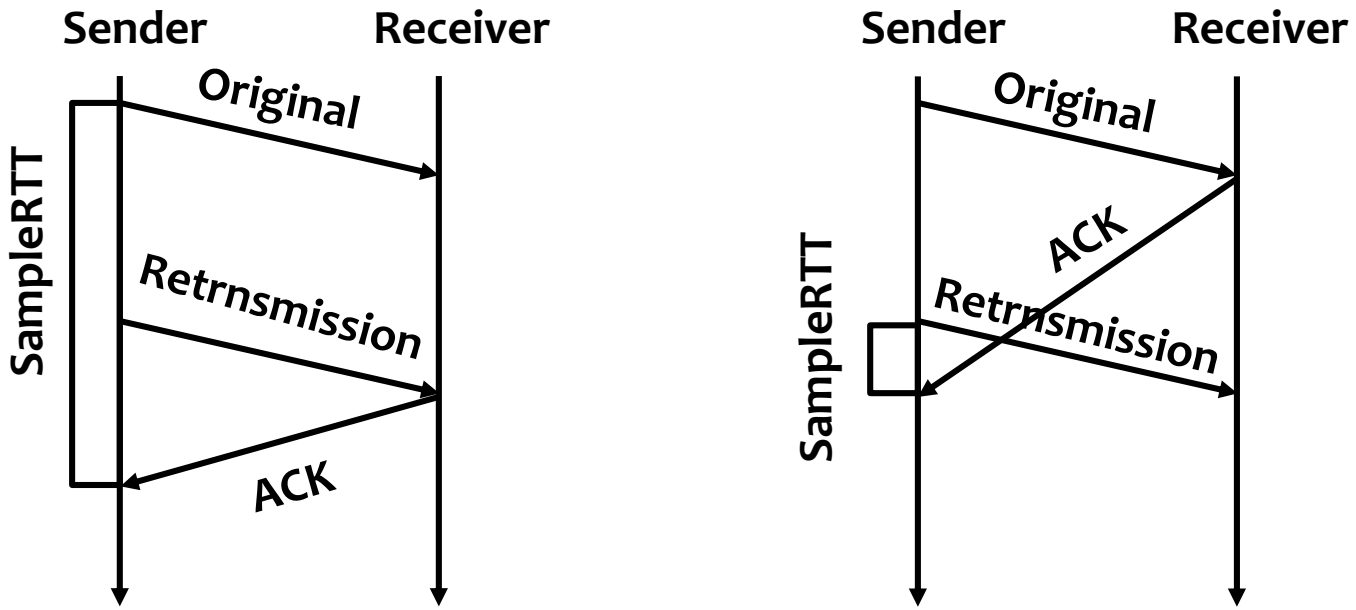
用於重送的逾時器值設定

- 原始演算法
  - 為每一對 資料段/ ACK 量測其 SampleRTT 值
  - 計算 比重化 (weighted) RTT 平均值
    - EstRTT = α x EstRTT + (1 - α)x SampleRTT
    - α 值介於 0.8 與 0.9 之間
  - 基於 EstRTT 來設定逾時時限值 (Timeout)
    - TimeOut = 2 x EstRTT

用於重送的逾時器值設定

- **SampleRTT 值的計算問題**
  - 當重送一個資料段後，立即收到一個 **ACK**
    - 計算 **RTT** 時，無法判定**此 ACK 是屬於第一次傳輸或是第二次傳輸 (重送)**

用於重送的逾時器值設定



判斷 ACK 屬於那一個封包可能產生的問題

(a) ACK 屬於第一次傳送封包 (正確應該是重送的)

(b) ACK 屬於重送封包 (正確應該是第一次傳送的)

Karn/Partridge 演算法

- 重新傳輸時不計算 RTT 值
- 在每次重送之後 逾時器值加倍
- **Karn-Partridge** 演算法改良原先的方法 ，但它無法消除壅塞問題
- 我們需要了解逾時器值與壅塞之間的關係
  - 如果逾時器值設定太小，你可能會重送不需重送的資料段，而增加網路負載

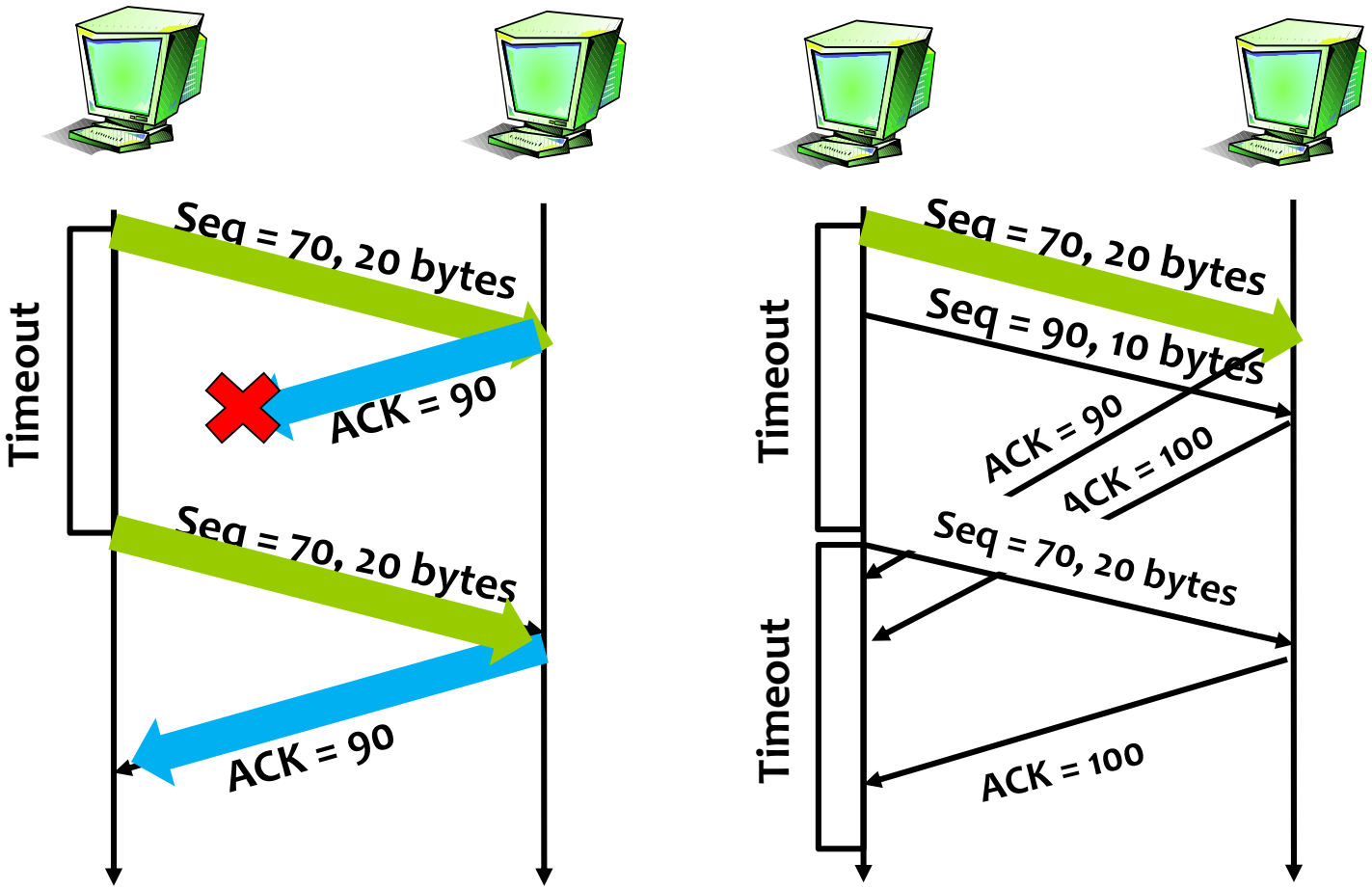
Karn/Partridge 演算法

- 原先計算方法的主要問題在於沒有考慮 SampleRTT 值之間的變異性 (variance)
- 如果 SampleRTT 值之間的變異小
  - 那麼 EstimatedRTT 值可信任度較高
  - 計算逾時器值時不需要將它乘以 2
- 如果 SampleRTT 值之間的變異大
  - 逾時器值不應該與 Estimated RTT值過於緊密

Jacobson/Karels 演算法

- Jacobson/Karels 提出一個新的 TCP 重送方案
- Difference = SampleRTT – EstimatedRTT
- EstimatedRTT = EstimatedRTT + (δ × Difference)
- Deviation = Deviation + δ (|Difference| – Deviation)
  - 其中 δ 是介於 0 與 1 之間的參數
- TimeOut = μ × EstimatedRTT + ϕ × Deviation
  - 基於經驗，$\mu = 1$ 以及 $\phi = 4$
  - 因此，當變異值很小， 逾時器值接近EstimatedRTT; 當變異值很大，會使 Deviation 值主宰計算結果

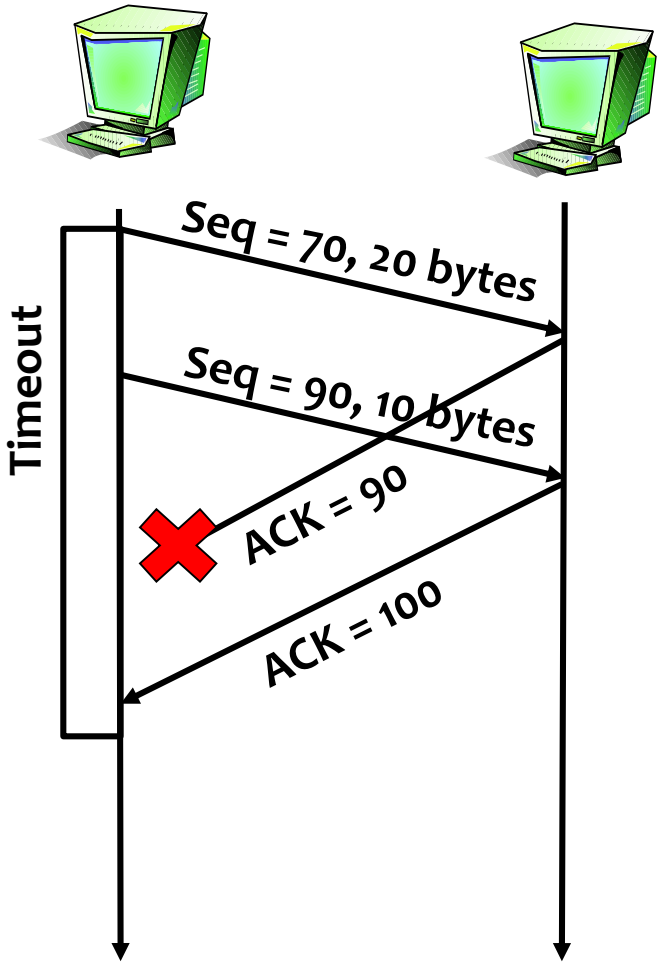
TCP 重送情境



ACK 遺失

ACK 延遲

TCP 重送情境



累積式 ACKs

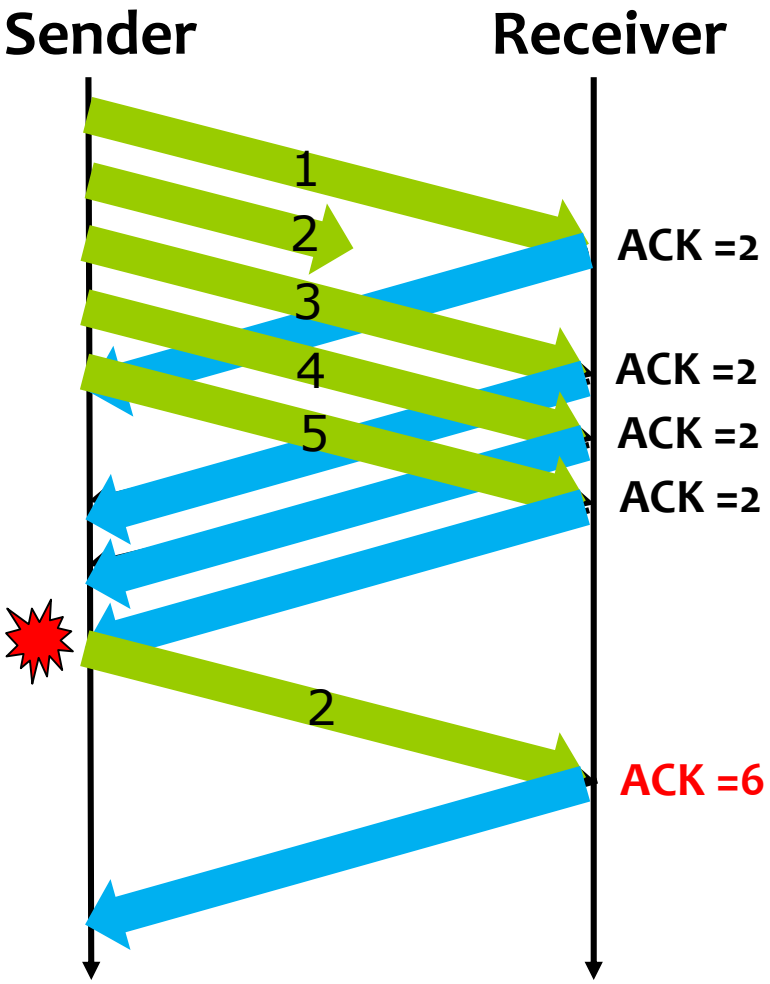
TCP 快速重送

- 快速重送 (Fast Retransmit)
  - 每當資料封包抵達接收端，接收端皆回傳一個回覆序號，即使此序號已經回覆過
  - 因此， 當一個非依序的封包非抵達接收端時 — TCP 會重送最近一次發送過的回覆
  - 這個再次傳輸的相同回覆被稱為 重複回覆 (*duplicate ACK*)

TCP 快速重送

- 快速重送 (Fast Retransmit)
  - 當傳送端看見重複回覆，就表示另一端一定是接收到失序的封包，也就是先前傳送的封包可能已經遺失
  - 因為先前傳送的封包也有可能只是延遲抵達而非遺失，因此傳送端會等到看見若干個重複回覆後, 才重送該遺失封包
  - TCP 會看見三個重複回覆後, 才重送該封包

TCP 快速重送範例



TCP 壅塞控制

- TCP壅塞控制的概念是讓每個傳送端決定目前網路有多大容量，以便知道自己可以同時傳送多少封包
- TCP 被稱為自我時序調節(*self-clocking*) ，也就是透過 **ACKs** 來調節封包傳輸的節奏

TCP 壅塞控制

- 線性增加倍數減少法 Additive Increase Multiplicative Decrease (AIMD)
  - *CongestionWindow*: 用來限制傳送端在一個連線上同時間能傳送多少資料
  - CongestionWindow 之於壅塞控制，有如 Advertised window 之於流量控制
  - 未回覆資料的最大允許值現在則是 CongestionWindow 與 Advertised window 的最小值
  - 傳輸速率(Transmission rate)

$$\text{Rate} = \frac{\text{CongestionWindow}}{\text{RTT}} \text{ 位元組/每秒}$$

TCP 壅塞控制

- **Additive Increase Multiplicative Decrease (AIMD)**
  - **TCP 的有效視窗 (effective window)被修正為以下算式:**
    - **MaxWindow = MIN (CongestionWindow, AdvertisedWindow)**
    - **EffectiveWindow = MaxWindow – (LastByteSent – LastByteAcked).**
  - **TCP 傳送端允許傳送的速度不能超過以下兩者最慢速度**
    - 網路 (基於壅塞控制)
    - 目的地主機 (基於流量控制)

TCP 壅塞控制

- Additive Increase Multiplicative Decrease (AIMD)
  - 如何決定CongestionWindow 值？
  - AdvertisedWindow 是接收端發送的值
  - 但是沒有人會傳送一個適當的 CongestionWindow 給TCP的傳送端
  - TCP 傳送端決定 CongestionWindow 值是基於它觀察到的網路壅塞程度
    - 當壅塞程度增加, 減少 congestion window 值
    - 當壅塞程度減少, 增加 congestion window 值
  - 此稱為 線性增加倍數減少法 *additive increase/ multiplicative decrease (AIMD)*

TCP 壅塞控制

- Additive Increase Multiplicative Decrease (AIMD)
  - 傳送端如何判定網路壅塞，進而縮減 congestion window 值？
    - TCP 將封包遺失 (收到三個重複回覆) 視為壅塞的訊號，然後減低傳輸速率
    - 每當發生封包遺失，傳送端便將 CongestionWindow 值減半
    - 這對應 AIMD 機制中的倍數減少 (multiplicative decrease) 的部分

TCP 壅塞控制

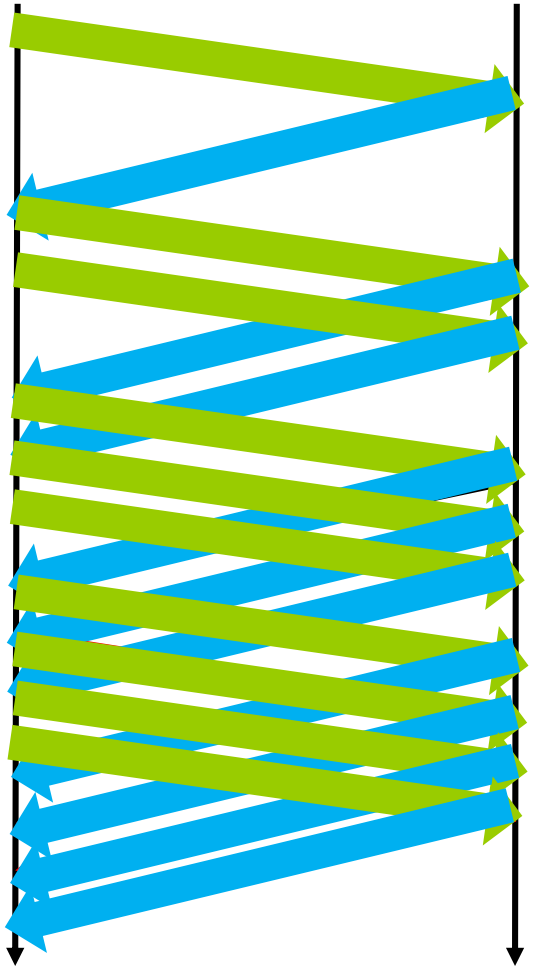
- **Additive Increase Multiplicative Decrease (AIMD)**
  - 雖然 CongestionWindow 值是以位元組來計算，但若從封包數量的角度思考會更容易理解倍數減少 (multiplicative decrease)的道理
    - 例如: 假設 CongestionWindow 值目前設為 16 個封包. 當偵測到封包遺失，CongestionWindow 值會設為 8
    - 持續的封包遺失會造成 CongestionWindow 值再減至 4 ，再減至 2 ，最終減至 1.
    - CongestionWindow 值不可小於單一封包的最大值 ，以 TCP 術語來說, 就是資料段最大值 *maximum segment size (MSS)*

TCP 壅塞控制

- Additive Increase Multiplicative Decrease (AIMD)
  - 當網路有新增的容量 (壅塞減輕), 可增加 CongestionWindow 值
  - 每當傳送端成功送出數量等於 CongestionWindow 值的封包 (所有在上一個 RTT 期間傳送的封包都成功收到回覆)，便將 CongestionWindow 值加上 MSS 值(相當於增加一個最大封包的量)

TCP 壅塞控制

- Additive Increase Multiplicative Decrease (AIMD)



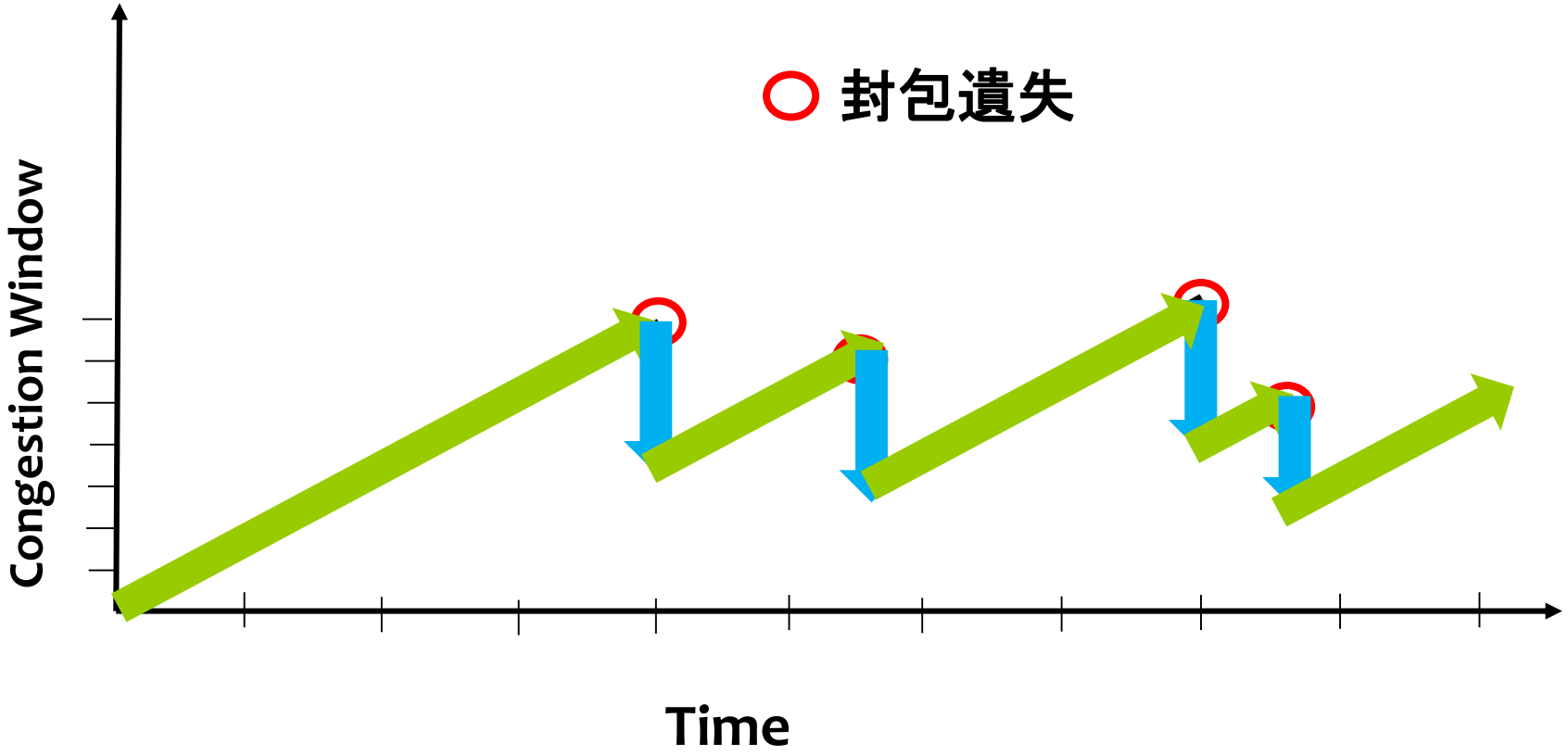
每經過一個 RTT 時間, 就增加一個封包傳輸量 (MSS)

TCP 壅塞控制

- **Additive Increase Multiplicative Decrease (AIMD)**
  - TCP 不等前一個 RTT 中所有 ACK 都收到才將 CongestionWindow 加上 MSS 值，而是每收到一個 ACK 就先加一部分 (CW 可以較早滑動)
  - 例如
    - CW = 5 x MSS,每收到一個 ACK 就先加 1/5 MSS
    - CW = 8 x MSS,每收到一個 ACK 就先加 1/8 MSS
  - Increment = MSS × (MSS/CongestionWindow)
  - CongestionWindow += Increment

TCP 壅塞控制

- **Additive Increase Multiplicative Decrease (AIMD)**
- 壅塞視窗成長路徑: 鋸齒狀



TCP 壅塞控制

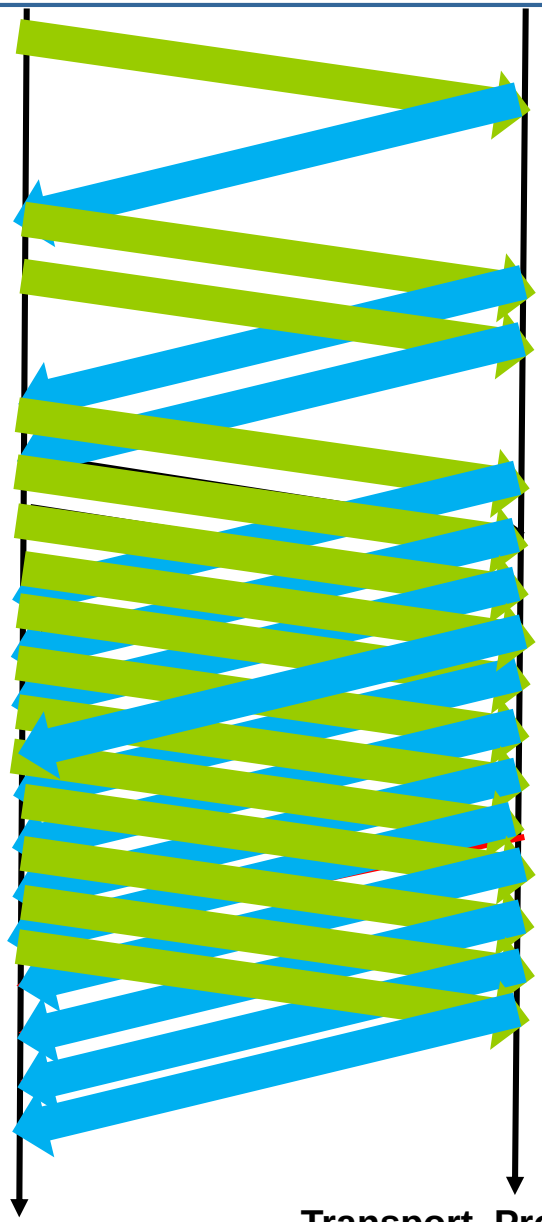
- 慢啟動 (Slow Start)
  - 當傳送端傳輸速度接近網路的容量時, 線性增加是不錯的機制. 但如果要探測網路的容量極限, 從連線一開始就使用線性增加會過於緩慢 , 需要花很長的時間
  - 慢啟動: 從冷啟動 (cold start) 開始快速增加 CongestionWindow 值的機制
  - 慢啟動以指數方式增加 CongestionWindow值, 而非線性方式增加

TCP 壅塞控制

- 慢啟動
  - 起初， CongestionWindow 值 = 1 封包值 (MSS)
  - 例如: MSS = 500 位元組, RTT = 200 msec
    - 初始速率 = 20 kbps
  - 當傳送端收到此封包的 ACK， TCP 將 CongestionWindow 值加 MSS，然後發送兩個封包.
  - 每收到一個 ACK 就加 1 封包 (事實上是加 MSS 值)
  - 一旦收到對應的兩個 ACKs ， TCP 將 CongestionWindow 值加 2 (2xMSS) — 每個 ACK 加一 (MSS) — 然後送出四個封包.
  - TCP 每經過一個 RTT 就會將傳送封包的數量加倍.

TCP 壅塞控制 (慢啟動)

- 慢啟動
- 當連線開始時，指數增加傳輸速率，直到封包首次遺失
  - 每 RTT 時間 `CongWin` 加倍
  - 以每次收到 ACK 就增加 `CongWin` 的方式實現
- 初始傳輸速率較慢，但會呈指數上升



慢啟動期間的封包傳送過程

TCP 壅塞控制

- 傳送端收到三個重複回覆 (3 dup ACKs) :
  - CongWin 減半
  - CongWin 呈線性增長
- 但逾時器發生逾時事件:
  - CongWin 設為 1 MSS ;
  - CongWin 呈指數增長
  - 增長到臨界值後， 改為線性增長

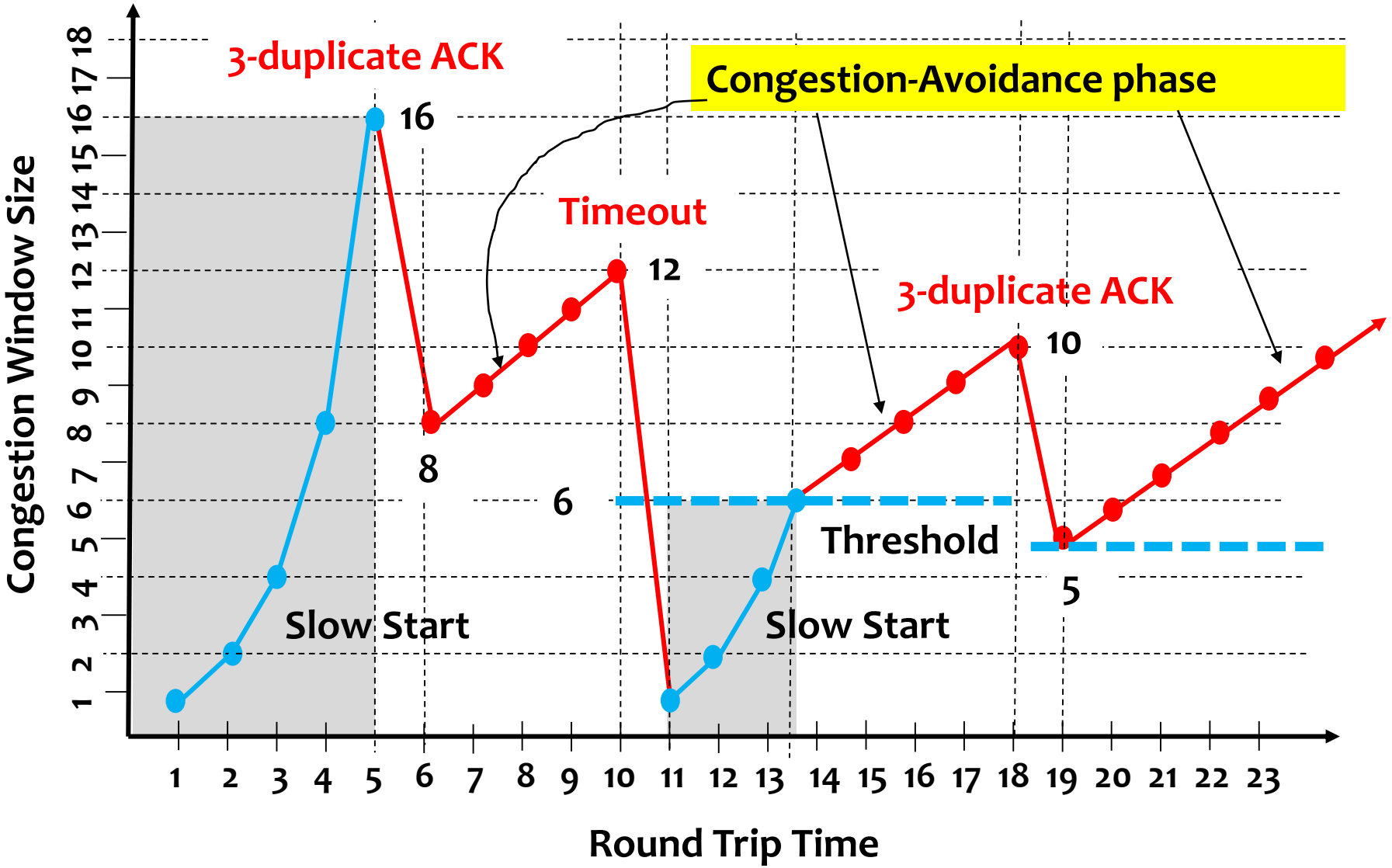
提醒

- 3 dup ACKs 顯示網路雖然壅塞, 但還可以傳送一些封包
- 逾時事件則顯示網路壅塞更為嚴重

TCP 壅塞控制

- 結論：
- 當 CongWin 低於臨界值，傳送端處於慢啟動 (slow-start) 階段， CongWin 會呈指數增長
- 當 CongWin 高於臨界值，傳送端處於雍塞預防 (congestion-avoidance) 階段，CongWin 會呈線性增長
- 當發生收到三個重複回覆，臨界值會設為 CongWin/2 ，CongWin 會設為臨界值
- 當發生逾時事件，臨界值會設為 CongWin/2，CongWin會設為 1 MSS.

TCP 壅塞控制


3-duplicate ACK
Congestion-Avoidance phase
Timeout
3-duplicate ACK
16
12
10
8
6
5
Threshold
Slow Start
Slow Start
Congestion Window Size
Round Trip Time
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23

TCP 傳送端壅塞控制

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 吞吐量

- TCP的平均吞吐量 (average throughout) 要如何用 window size 與 RTT 來表示？
  - 忽略慢啟動期間
- 令 W 為封包遺失發生時的 CongWin 值
- 當 CongWin 值為 W 時，吞吐量為 W/RTT
- 就在封包遺失後，CongWin 值減為 W/2，吞吐量則減為 W/2RTT
- 平均吞吐量: 0.75 W/RTT

結論

- 我們介紹了如何將主機之間的封包遞送服務轉換為程序之間的通訊管道
- UDP 提供不可靠的傳輸服務
- TCP 提供可靠的傳輸服務
  - 三向交握連線建立機制
  - TCP 連線狀態圖
  - TCP 逾時器時限值計算
  - TCP 重送情境
  - TCP 快速重送機制

結論

- TCP 壅塞控制機制
  - AIMD (線性增加倍數減少)
  - 慢啟動 (Slow start)
  - 三個重複回覆 (3-duplicate ACKs, 封包遺失)
  - 逾時器逾時事件 (Timeout)