Computer Networking Notes Ch. 3

傳輸層提供的服務

傳輸層

- 提供邏輯上的通訊功能。
- 應用程式提供傳輸層提供的邏輯通訊發送訊息（Message），不需要考慮內部的物理細節。
- 傳輸層基本上分為 TCP 與 UDP，且通常只在邊緣系統上負責。

傳輸層的運作原理

- 傳輸層會從應用程式接收到的訊息（Message）轉成多個訊息區段（Segment）。
- 多個訊息區段（Segment）會將其封裝給網路層，並交由網路層發送到指定的目的地。
    - 網路層主要是負責兩個主機（Host）的邏輯通訊。
    - 傳輸層主要是負責兩個程序（Process）的邏輯通訊。

TCP、UDP 與 IP

- TCP
    - 提供可靠的連接傳輸方式。
    - 提供可靠傳輸服務、壅塞控制。
    - 在 TCP 上的訊息分組稱為 Segment。
- UDP
    - 提供不可靠，無連接的傳輸方式。
    - 在 UDP 上的訊息分組稱為 Datagram。
- IP
    - 屬於網路層的範疇。
    - 盡力交付的服務：IP 會盡最大的努力在通訊的主機之間傳輸訊息區段。
    - 不可靠服務：IP 不做任何的確保，單純傳送資訊。

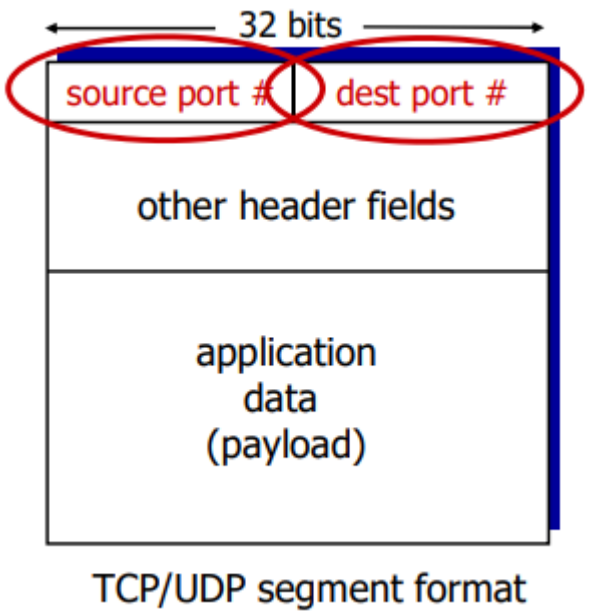
多路複用（Multiplexing）與多路分解（Demultiplexing）

多路複用與多路分解的概述

- 多路複用與多路分解
    - 網路層提供從主機到主機之間傳輸服務，延伸到應用程式從程序到程序之間的交付服務。
- 多路分解
    - 在目標主機上，將從網路層獲得到的訊息區段中的資料，交付到對應的 socket 中。
- 多路複用
    - 在來源主機上不同的 socket 上蒐集 segment，並在 segment 上封裝 header，
      多路分解可藉由 header 分解成多個 segment，並將 segment 傳送至網路層。

Segment 的架構

- 都會有來源主機與目標主機的 IP、Port。
- 都會持有一個傳輸層的區段。
- 主機使用 IP 地址與 Port，來將訊息指向適合的 socket 上。



TCP/UDP segment format

無連接的多路複用與多路分解

> 待補

連接的多路複用與多路分解

> 待補

User Datagram Protocol

UDP 的簡介

- UDP 的簡介
  - 沒有多餘的裝飾、極簡的網路傳輸協定
  - 盡力而為服務：可能會 loss 或者傳輸亂序的資料給應用程式。
  - 無連接協定：不須 handshaking，UDP Segment
- Why use UDP?
  - 不用 handshaking。
  - 不需要在 sender、receiver 紀錄狀態。
  - 極小的 header size。
  - 不用壅塞控制。
- Where to use UDP?
  - 影音串流平台
  - DNS
  - SNMP
  - HTTP/3（通常會新增為了可靠傳輸所需要的特性，以及新增壅塞控制到應用層上）。

UDP 的傳輸層動作

- UDP Sender 的動作
- UDP Receiver 的動作
  - 從 IP 取得 segment
  - 使用 checksum 驗證 segment 的資料

UDP Checksum

- 發送方對 segment 的訊息區段內所有 16bits 的數值進行總和，將總和的值進行反運算當作驗證碼。
- 接收方接收到 segment，總和後加上驗證碼，若出現 0 則代表 segment 有誤。

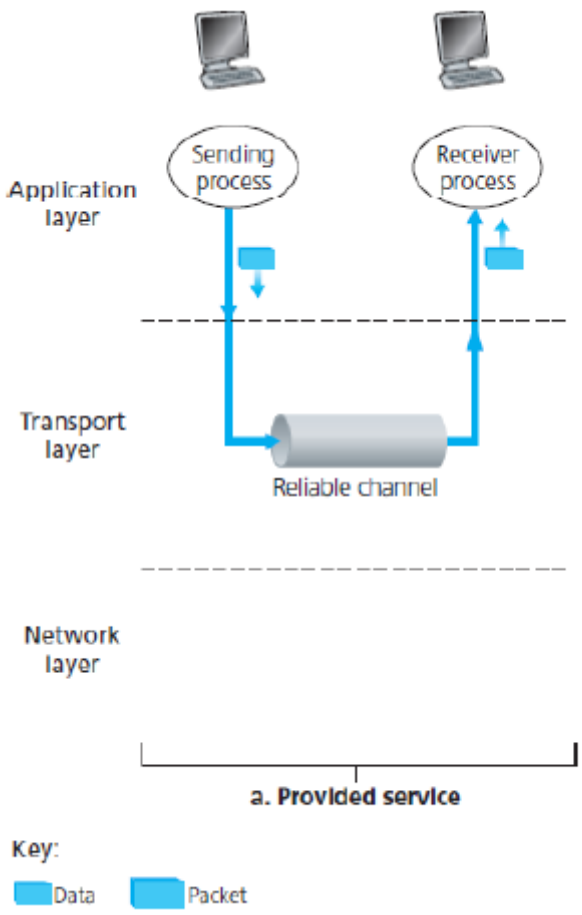
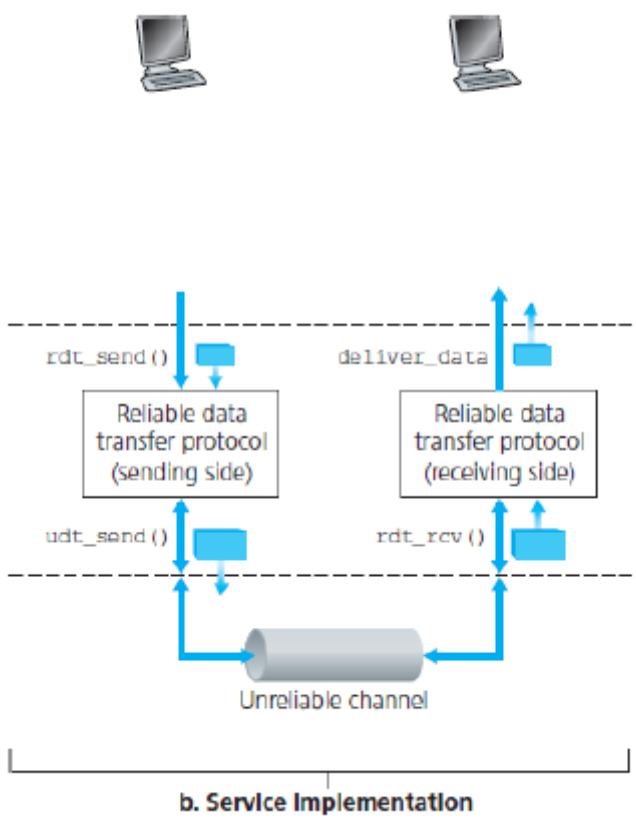
UDP 的比較

- UDP 提供錯誤偵測，但不包含錯誤修復，僅能放棄訊息區段或者發出警告。
- TCP 提供了可靠傳輸，先見「可靠資料傳輸原理」後見 TCP 傳輸原理。

可靠資料傳輸原理

可靠資料傳輸原理

- 期望提供的功能：
  - 資料能夠藉由一條可靠的通道進行傳輸。
- 實際上功能上的實作：
  - 利用可靠資料傳輸協定（Reliable Data Transfer Protocol），將資料藉由不可靠的通道（例如 IP）進行傳輸。

期望	實作
	

rdt 1.0: Reliable transfer over a Reliable channel

介紹 rdt 1.0

- 考慮底層通道是完美的
  - 沒有 bit 錯誤的問題
  - 沒有掉封包的問題
- 發送端與接收端可以形成一個有限的狀態機
  - sender 等待上層指示，製作 packet 與發送 packet 到底層通道內。
  - receiver 等待下層指示，接收 packet 與傳遞資料至上層。

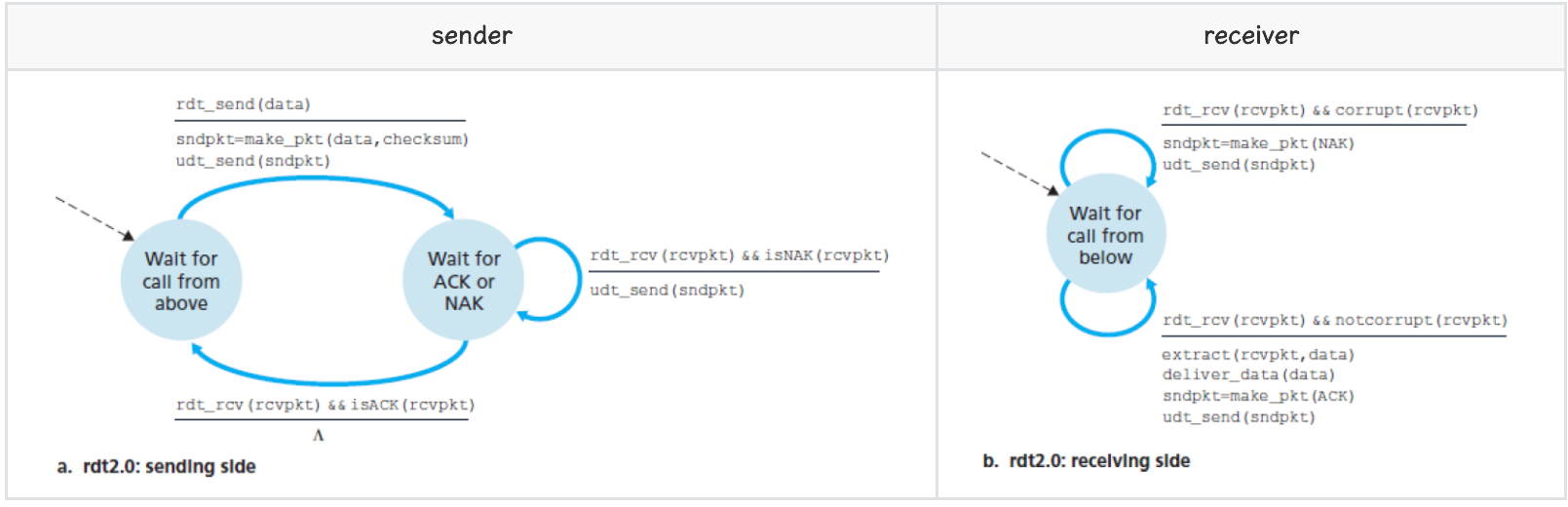
rdt 2.0: Channel with bit errors

介紹 rdt 2.0

- 底層通道較趨近真實的情況，訊息區段是有可能在傳輸過程受損的。
- 接收方得讓發送方知道哪些內容正確被接收，哪些內容需要重傳。
  - 這樣的想法使用自動重傳請求 ARQ (Automatic Repeat reQuest) 協議來實踐。
- 使用 ARQ 之外，還需要三種協議功能來處理 bit error 的問題：
  - 錯誤檢測
  - 接收方反饋：往接收方發送 ACK (0) 與 NAK (1)。
  - 重傳：接收方收到有錯誤的區段時，發送端重新傳輸該訊息區段。

因此，rdt 2.0 可以畫成以下的狀態機

- sender：等待上層指示發送資訊至底層通道，等待 ACK 或 NAK 資訊，根據資訊決定是否傳輸或者重傳。
- receiver：等待下層指示，如果收到了資訊，但資訊是錯誤的，發送 NAK，否則發送 ACK 並且傳遞資料給上層。



rdt 2.0 這樣的協議也被稱作停等（stop-and-wait）協議。

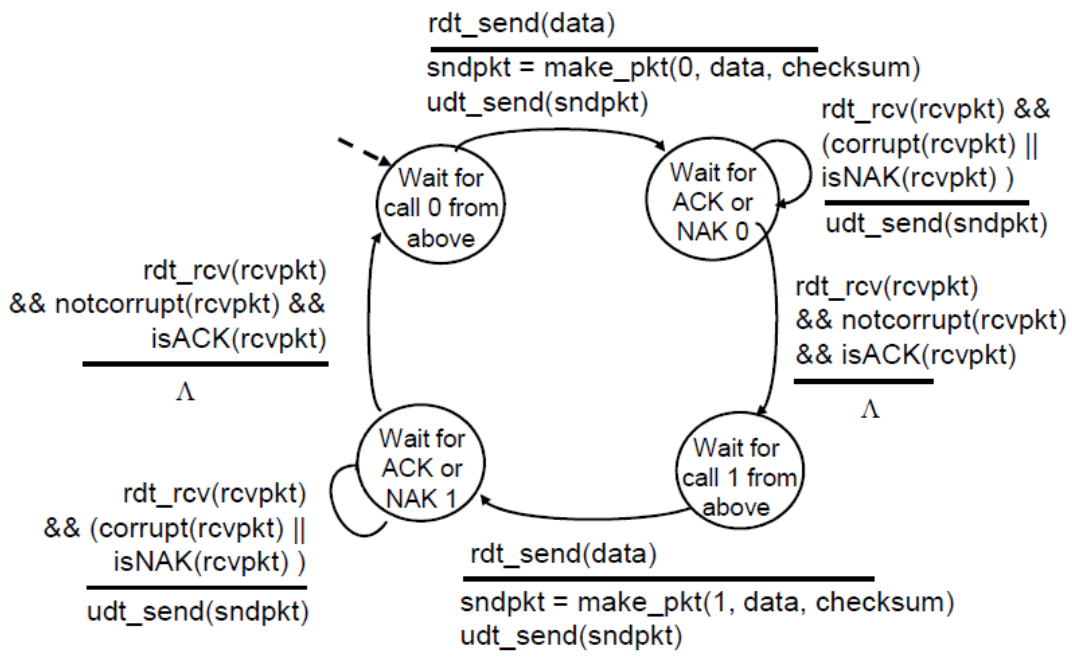
rdt 2.0 的問題

- 回傳的 ACK & NAK 同時也可能是錯的，也就是 ACK 變成了 NAK，或者 NAK 變成了 ACK。
  - 不太可能再次重傳，可能會 duplicate。
- 解決問題的簡單方式
  - 目前假設的問題：只有 bit 會錯誤，但不會掉封包。
  - 因此讓封包帶有序號，檢查序號即可知道是否為重傳。

rdt 2.1

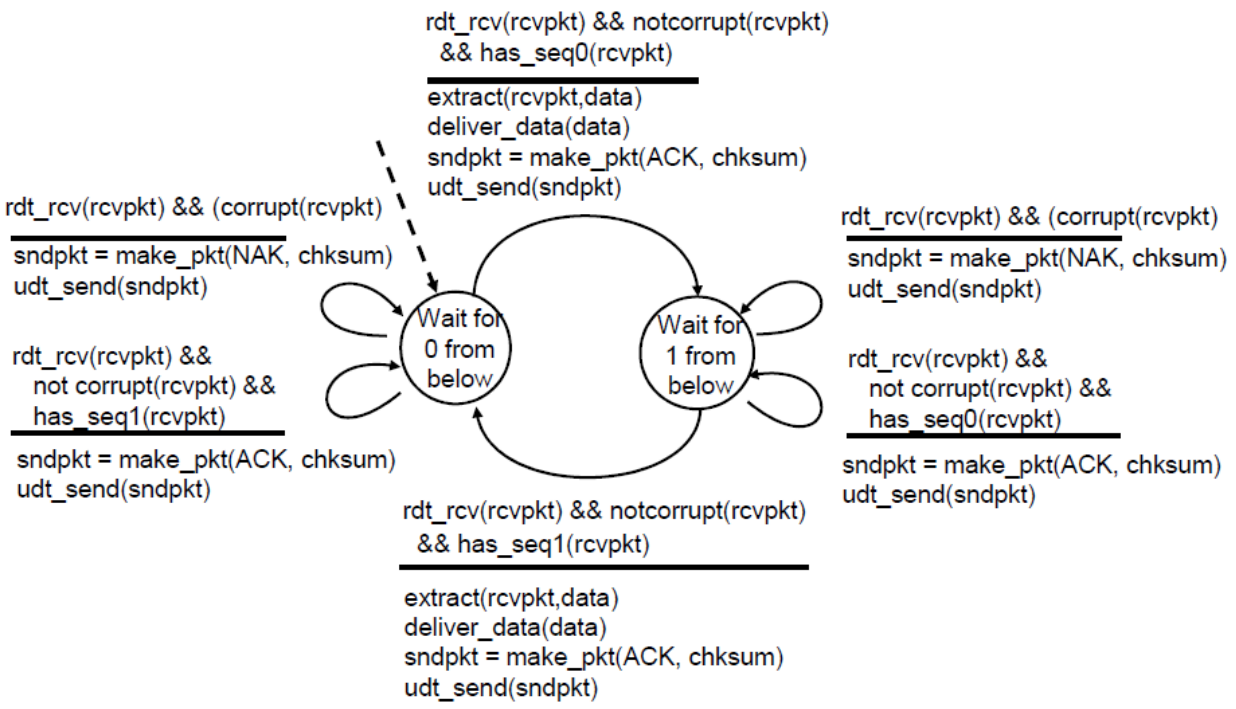
Sender 可以分成以下的狀態機：

1. 等待上層傳送 0 信號，接著傳送序號為 0 的 segment
2. 等待剛剛傳送序號為 0 的封包回傳 ACK 或者 NAK
   - 如果收到了 NAK 或者封包錯誤，那麼重傳
3. 等待上層傳送 1 信號，接著傳送序號為 1 的 segment
4. 等待剛剛傳送序號為 1 的封包回傳 ACK 或者 NAK
   - 如果收到了 NAK 或者封包錯誤，那麼重傳



Receiver 可以分成以下的狀態機：

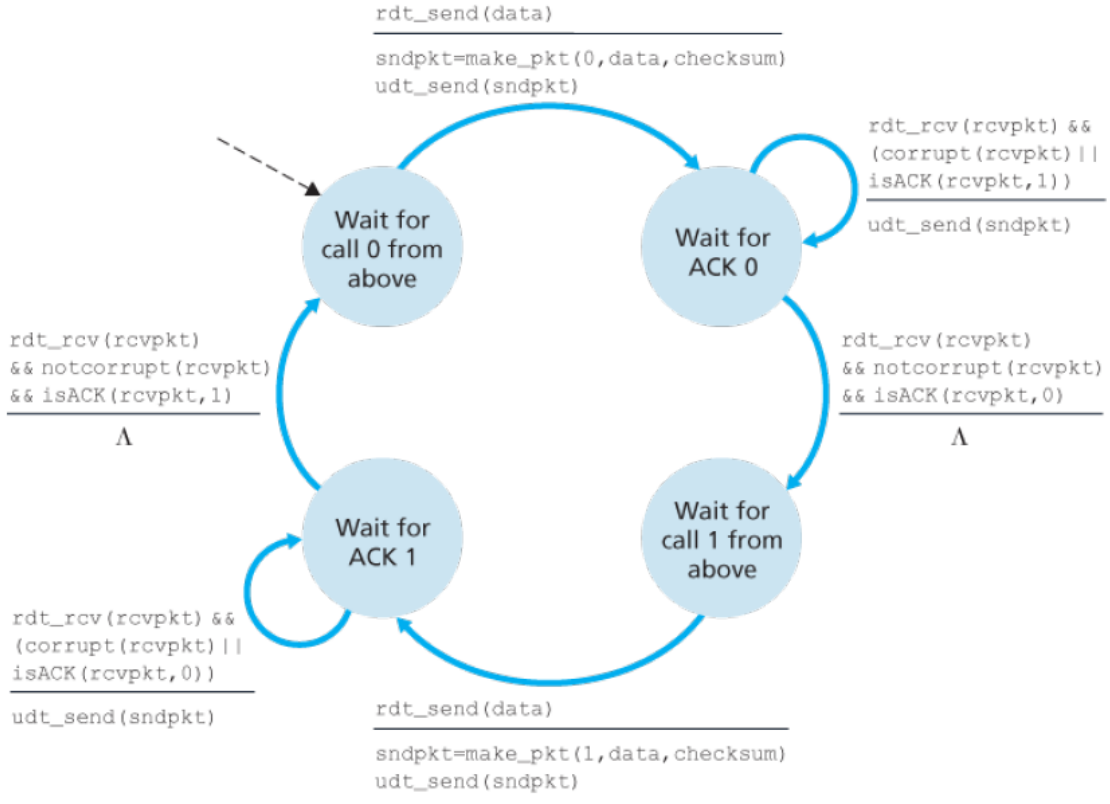
1. 接收到了信號 0，確認 segment 是序號 0 且沒有問題，將資料傳至上層並傳送 ACK 的 packet。
   - 若接收到了 segment 且 segment 是錯的，傳送 NAK 的 packet。
   - 若接收到了 segment 且 segment 是對的，但 segment 有序號 1，傳送 ACK。
2. 接收到了信號 1，確認 segment 是序號 1 且沒有問題，將資料傳至上層並傳送 ACK 的 packet。
   - 若接收到了 segment 且 segment 是錯的，傳送 NAK 的 packet。
   - 若接收到了 segment 且 segment 是對了，但 segment 有序號 0，傳送 ACK。



rdt 2.2

與 rdt 2.1 不同的地方：由於發送端只要收到了兩個同組的 ACK 後，確定該封包為冗餘封包，就可以知道接收方沒有正確接收到分組。

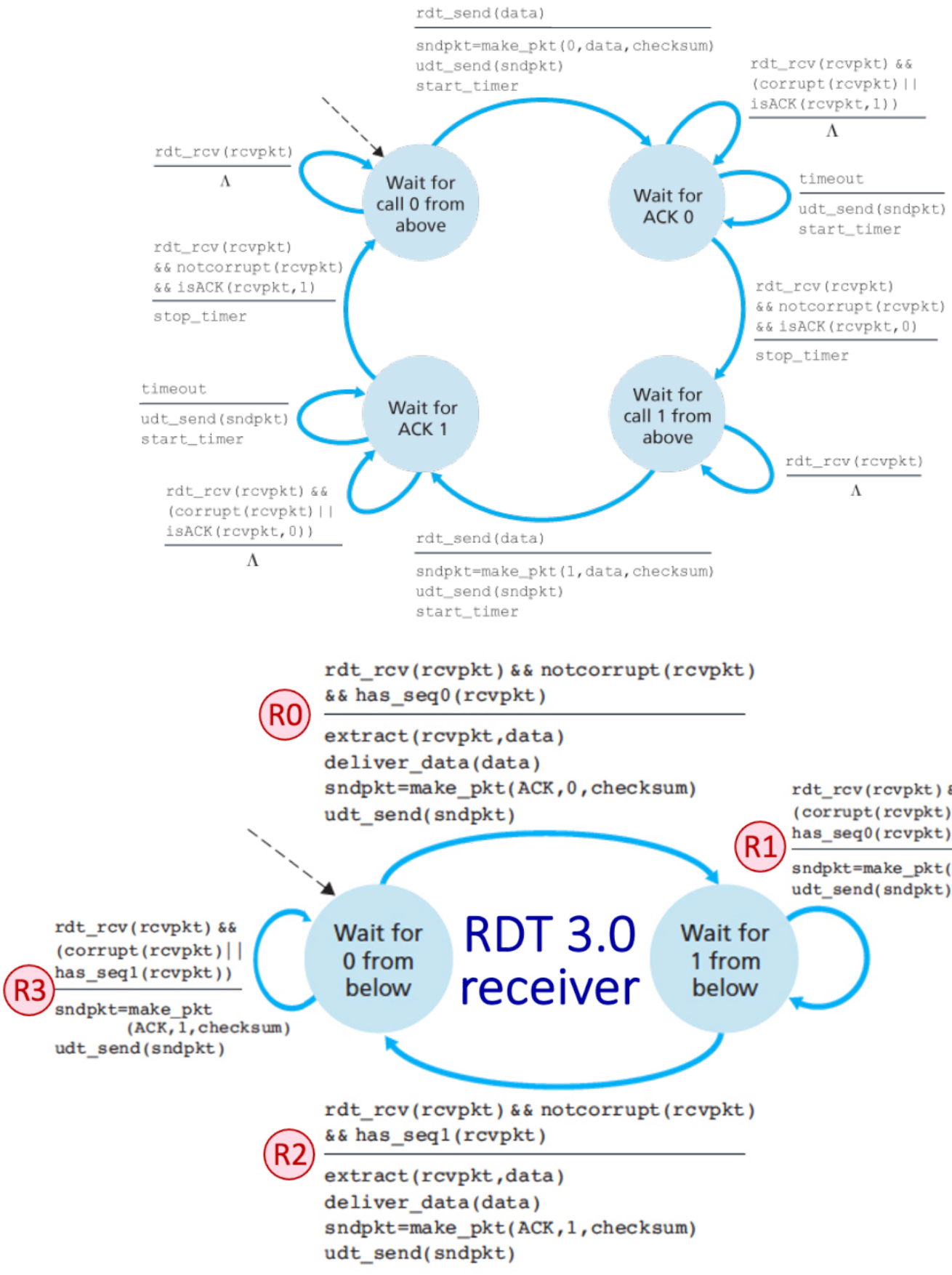
因此 rdt 2.2 拋棄了發送 NAK 的特性，如下面的發送方狀態圖。



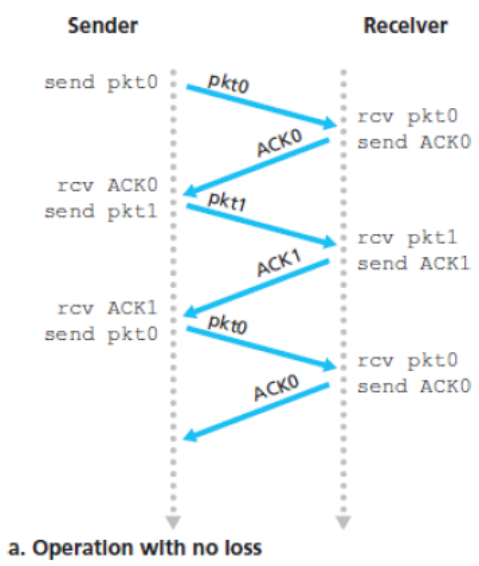
rdt 3.0

- 底層通道較趨近真實的情況，訊息區段是有可能在傳輸過程受損之外，也有可能掉封包。
- 讓發送方負責檢測和恢復掉封包的動作，如果發送方可以等待足夠長的時間來確定封包已丟失，那麼他只需要重傳即可。
  - 等多久呢？讓發送方選一個適當的時間值。
  - 冗餘封包？rdt 2.2 處理掉了這個問題。
- 實現等待足夠長的時間來處理掉封包的問題，需要一個倒數計時器，且滿足以下的需求
  - 每次發送一個 segment 時，啟動一個定時器
  - 收到 response 的時候，定時器採取適當的措施
  - 可以終止計時器

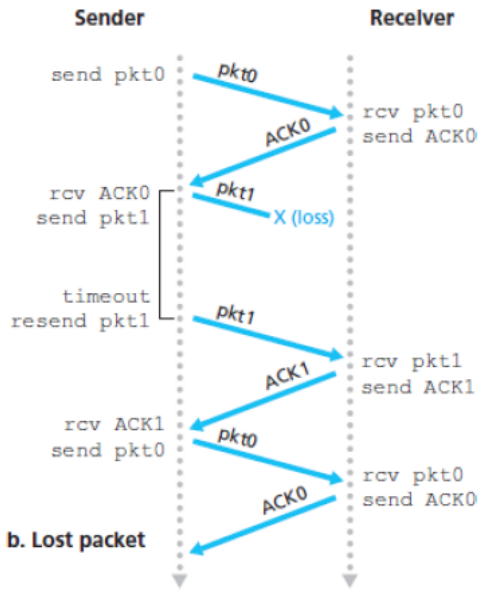
下圖呈現了 rdt 3.0 的發送與接收狀態圖



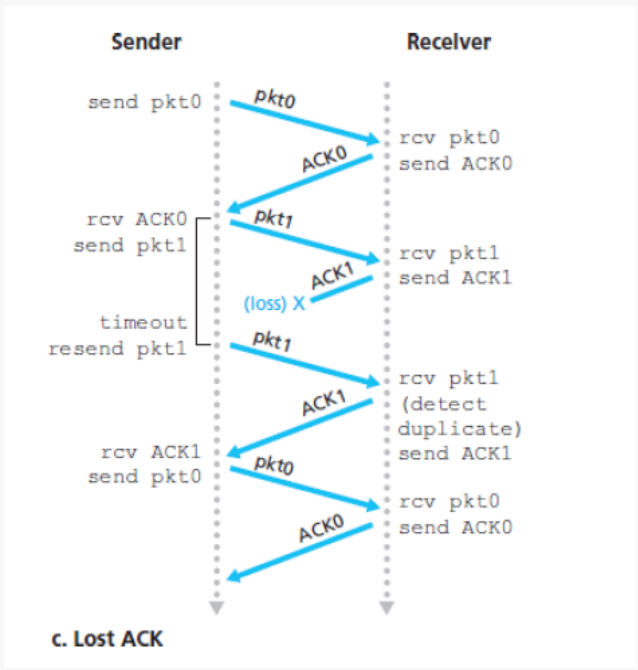
rdt 3.0 可以使用以下的圖，來呈現遇到問題時解決的方式



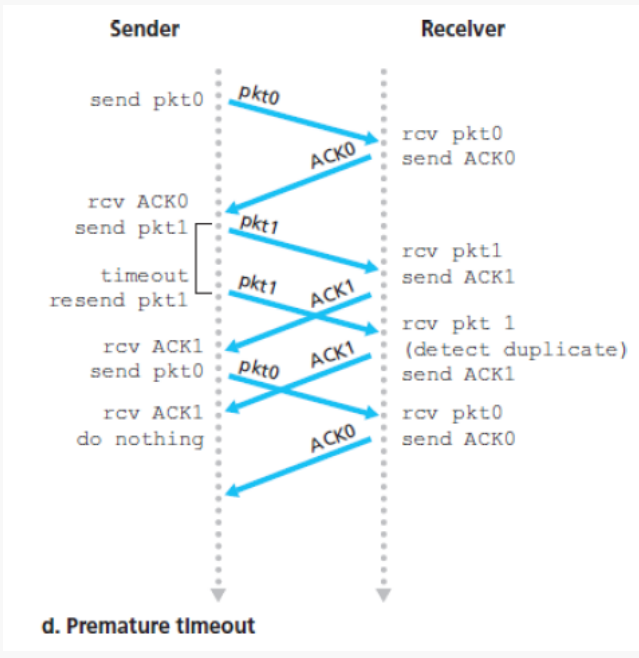
a. Operation with no loss



b. Lost packet



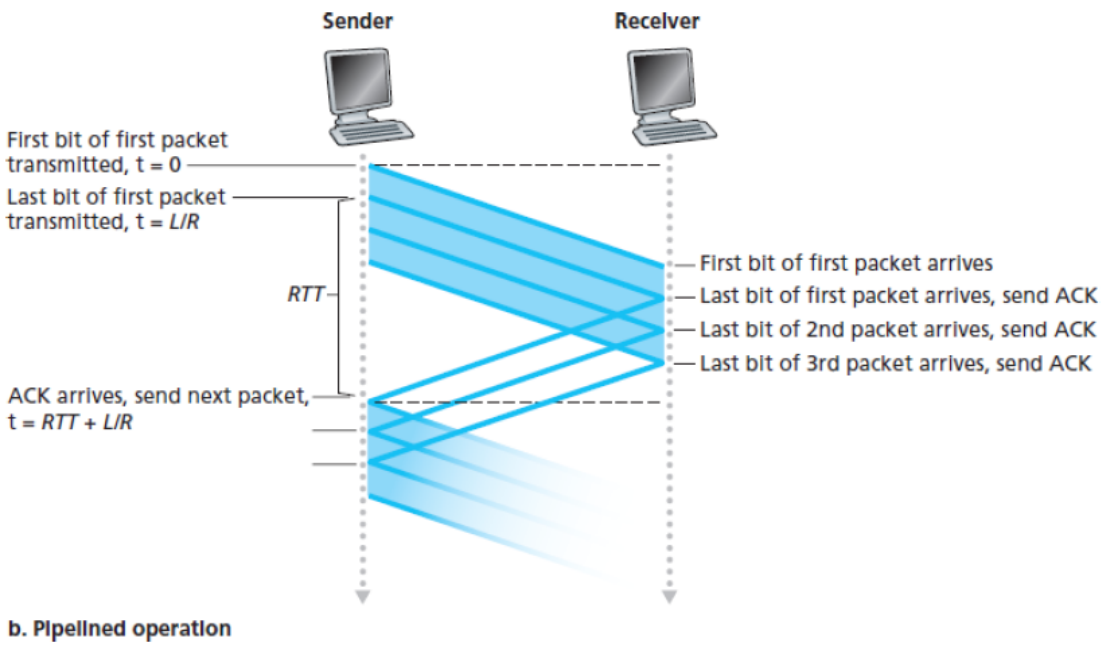
c. Lost ACK



d. Premature timeout

流水線可靠資料傳輸協定

- 停等協定的效能很糟糕，原因是考慮 RTT 的影響後，發送封包到等待封包回來的時間太長，利用率過低
- 所以我們應該不停地發送封包而不等待確認，使用流水線技術。



b. Pipelined operation

- 同時需要處理以下的事項：
  - 增加序號範圍，必須保證送出去的封包序號不一樣，避免衝突。
  - 發送方與接收方也許沒辦法暫存 segment，因為會一次性地吃很多個 segment，所以遇到正確的就要儲存。
  - 利用回退 N 步（Go-Back-N, GBN）以及選擇重傳（Selective Repeat, SR）來解決丟失、損壞與超時的問題。

Go-Back-N

- 我們有好幾個 segment 等著發送，我們可以使用流水線的方式進行發送。
- 我們使用 slide windows 的方式進行發送，若有 10 個 segment，我們一開始發送 [0, 1, 2, 3]，接著在收到 ACK 0 的時候發送 [1, 2, 3, 4] 等等，其中決定一次發送幾個 segment 的準則，來自於 Go-Back-N 所指定的 N 的數量。
- GBN 必須要，也處理了以下的事項：
  - 發送：發送時先確定 windows 是否已滿，滿了就等待，沒滿就發送。

    - 收到一個 ACK：滑動窗口，發送下一個 segment。
    - 超時事件：確定哪個 segment 超時，接著將該 base 超時的 segment 的 slide windows 全部重送。
        - 例如 2 超時了，且 N=4，接收端拋棄 [3, 4, 5] 已經傳送成功的封包，[2, 3, 4, 5] 都會再重送。
- GBN 有以下的優點：
    - 解決了需要按序交付的問題。
    - 不需要暫存失序的分組。
- GBN 同時隱含以下的問題：
    - 丟棄所有的失序分組有點浪費，可能會造成需要重傳多次的問題。

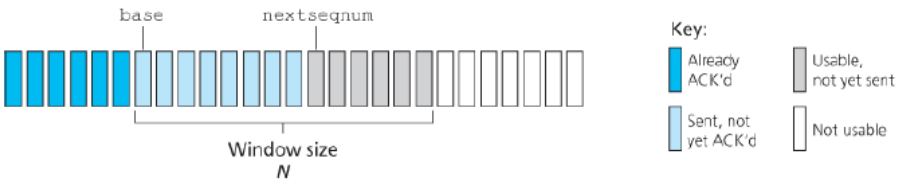


Figure 3.19 Sender's view of sequence numbers in Go-Back-N

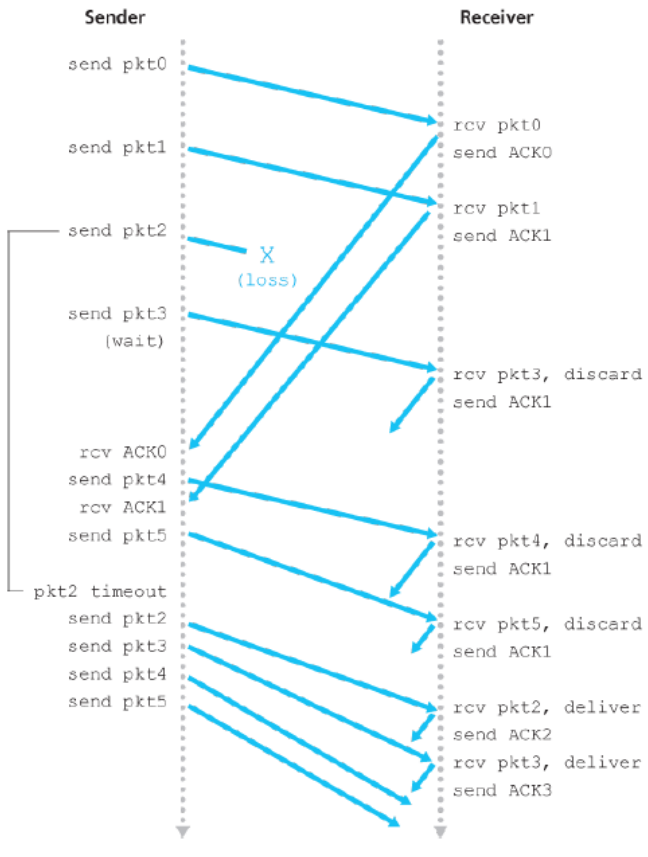
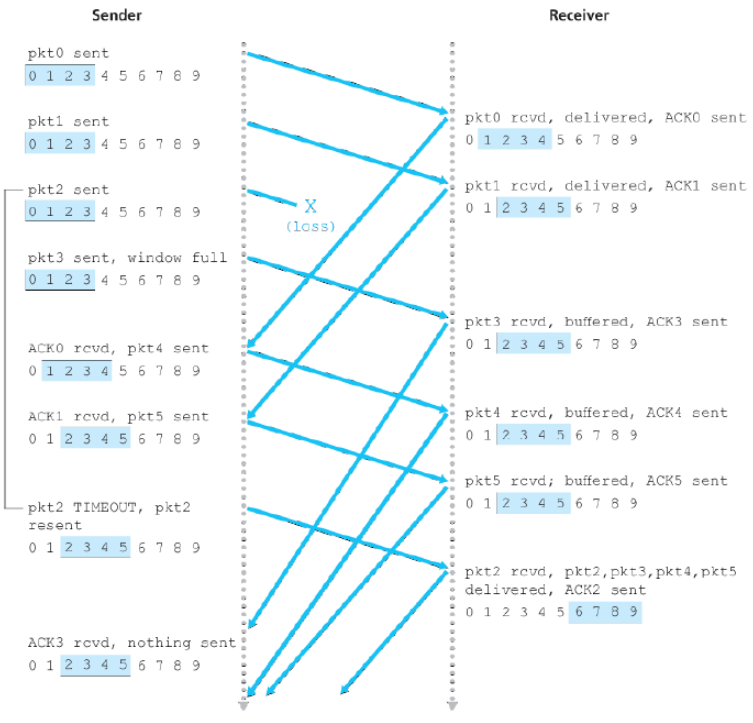


Figure 3.22 Go-Back-N in operation

Selective Repeat

- 我們有好幾個 segment 等著發送，我們可以使用流水線的方式進行發送。
- 我們使用 slide windows 的方式進行發送，若有 10 個 segment，我們一開始發送 [0, 1, 2, 3]，接著在收到 ACK 0 的時候發送 [1, 2, 3, 4] 等等，其中決定一次發送幾個 segment 的準則。
- 比起 GBN，SR 改變了以下的事情：
    - 超時事件：確定哪個 segment 超時，接著將該 base 超時的 segment 重送。
        - 例如已經傳送 [2, 3, 4, 5]，2 超時了，就會再次傳送 2 這個 segment，接著傳送 6, 7...
- SR 有以下的優點：
    - 利用暫存解決掉 GBN 浪費的問題
- SR 有以下的問題：
    - 當今天序號的範圍是有限的（例如 0 1 2 3 循環）時，若沒有適當的選取範圍，會導致因為 Receiver 不知道 Sender 傳了什麼，而導致誤解發生。
        - 窗口長度必須要小於等於序號空間大小的一半。

正常運作	不當選取範圍
 Figure 3.26 SR operation