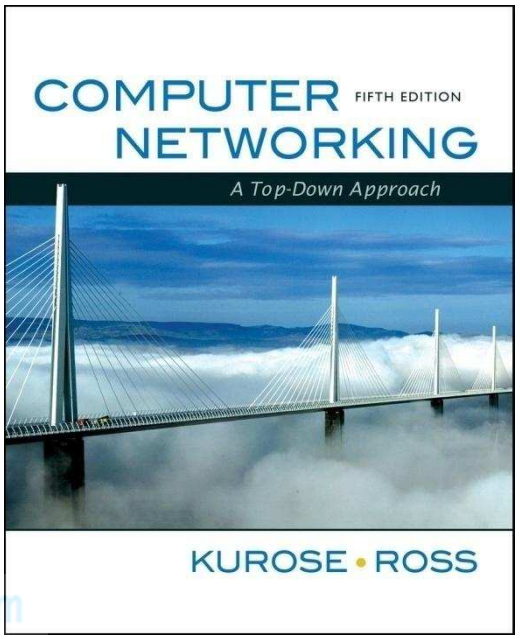
Chapter 3
Transport Layer



A note on the use of these ppt slides:

We're making these slides freely available to all (faculty, students, readers). They're in PowerPoint form so you can add, modify, and delete slides (including this one) and slide content to suit your needs. They obviously represent a *lot* of work on our part. In return for use, we only ask the following:

- If you use these slides (e.g., in a class) in substantially unaltered form, that you mention their source (after all, we'd like people to use our book!)
- If you post any slides in substantially unaltered form on a www site, that you note that they are adapted from (or perhaps identical to) our slides, and note our copyright of this material.

Thanks and enjoy! JFK/KWR

Computer Networking: A Top Down Approach 5th edition. Jim Kurose, Keith Ross Addison-Wesley, April 2009.

Chapter 3: Transport Layer

Our goals:

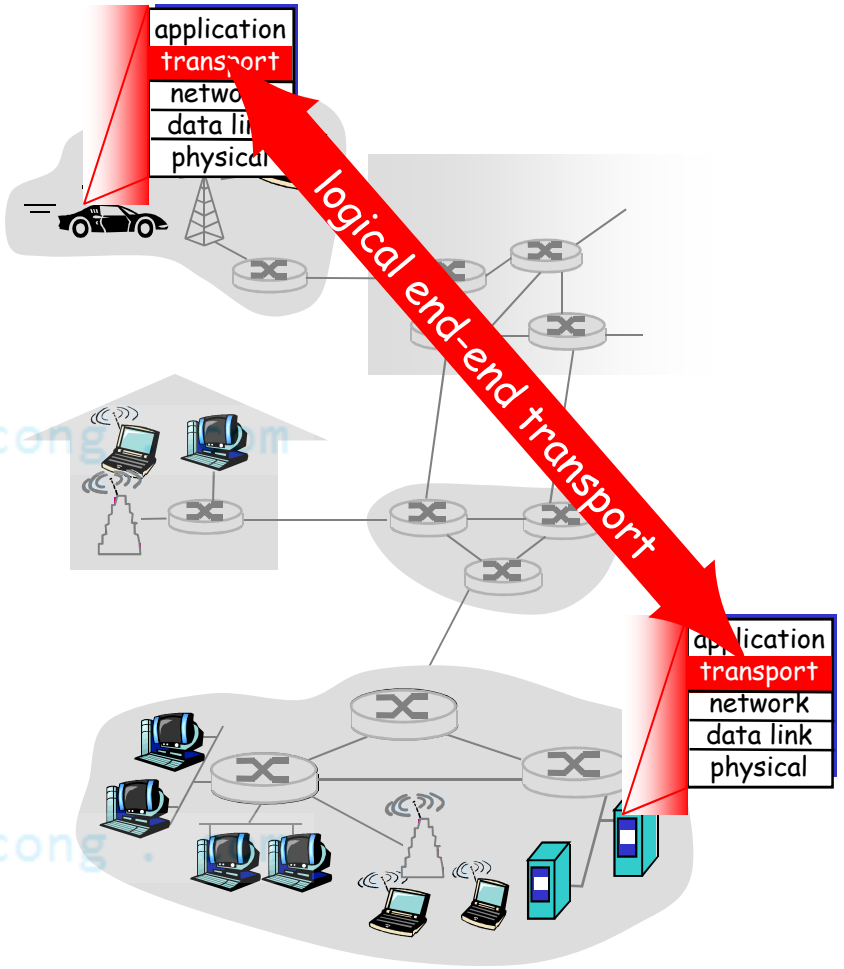
- understand principles behind transport layer services:
  - multiplexing/demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- learn about transport layer protocols in the Internet:
  - UDP: connectionless transport
  - TCP: connection-oriented transport
  - TCP congestion control

Chapter 3 outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

Transport services and protocols

- cung cấp *truyền thông logic* [*logical communication*] giữa các tiến trình ứng dụng đang chạy trên các host khác nhau.
- Các giao thức vận chuyển chạy trên các host
  - Phía gửi: chia message ở tầng Transport thành các segments, sau đó gửi các segment này xuống tầng Network
  - Phía nhận: ráp các segments thành các message, sau đó gửi lên tầng Application
- Có nhiều hơn 2 giao thức vận chuyển để các ứng dụng mạng sd
  - Internet: TCP và UDP



Transport vs. network layer

- *Tầng Network:* truyền thông logic giữa các hosts
- *Tầng Transport :* truyền thông logic giữa các tiến trình
  - relies on, enhances, network layer services

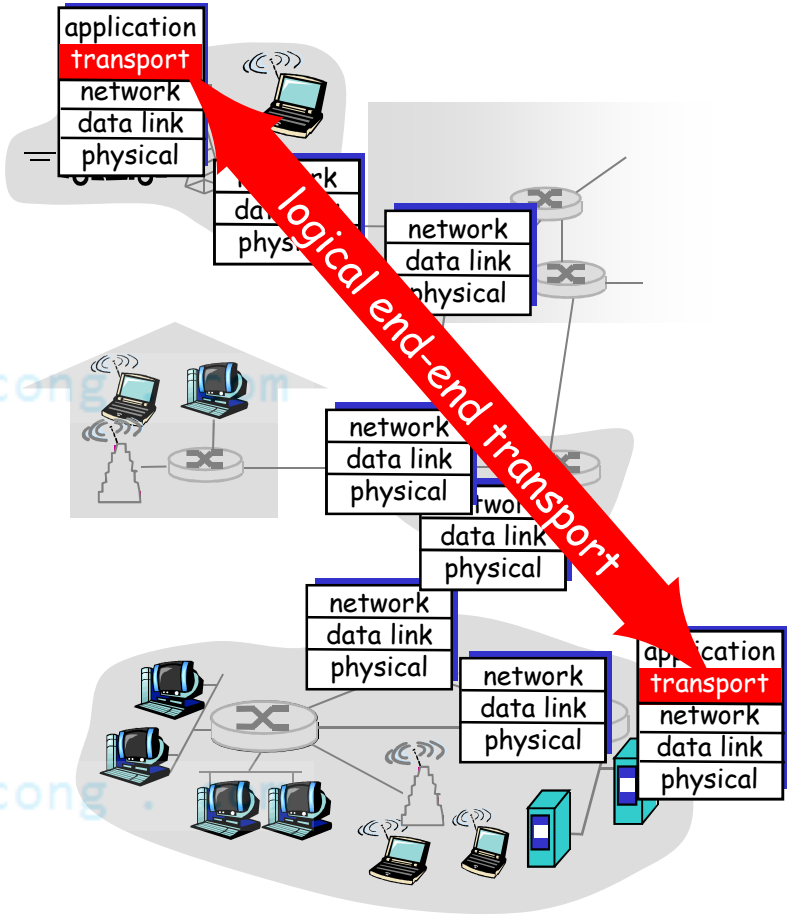
Household analogy:

12 đứa trẻ gửi thư cho 12 đứa trẻ khác.

- Tiến trình = đứa trẻ
- Message ở tầng Ứng dụng = thư trong phong bì
- hosts = gia đình
- Giao thức vận chuyển = Ann và Bill
- Giao thức ở tầng mạng = Dịch vụ bưu chính

Internet transport-layer protocols

- Vận chuyển bảo đảm [reliable], đúng thứ tự [in-order] (TCP)
  - congestion control
  - flow control
  - connection setup
- Vận chuyển ko bảo đảm [unreliable], không đúng thứ tự : UDP
  - no-frills extension of "best-effort" IP
- Các dịch vụ ko cung cấp:
  - delay guarantees
  - bandwidth guarantees



M1 no-frills: bình dân?
No-frills or no frills is a term used to describe any service or product for which the non-essential features have been removed to keep the price low.
MVC, 5/3/2010

cuu duong than cong . com

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

Multiplexing/demultiplexing

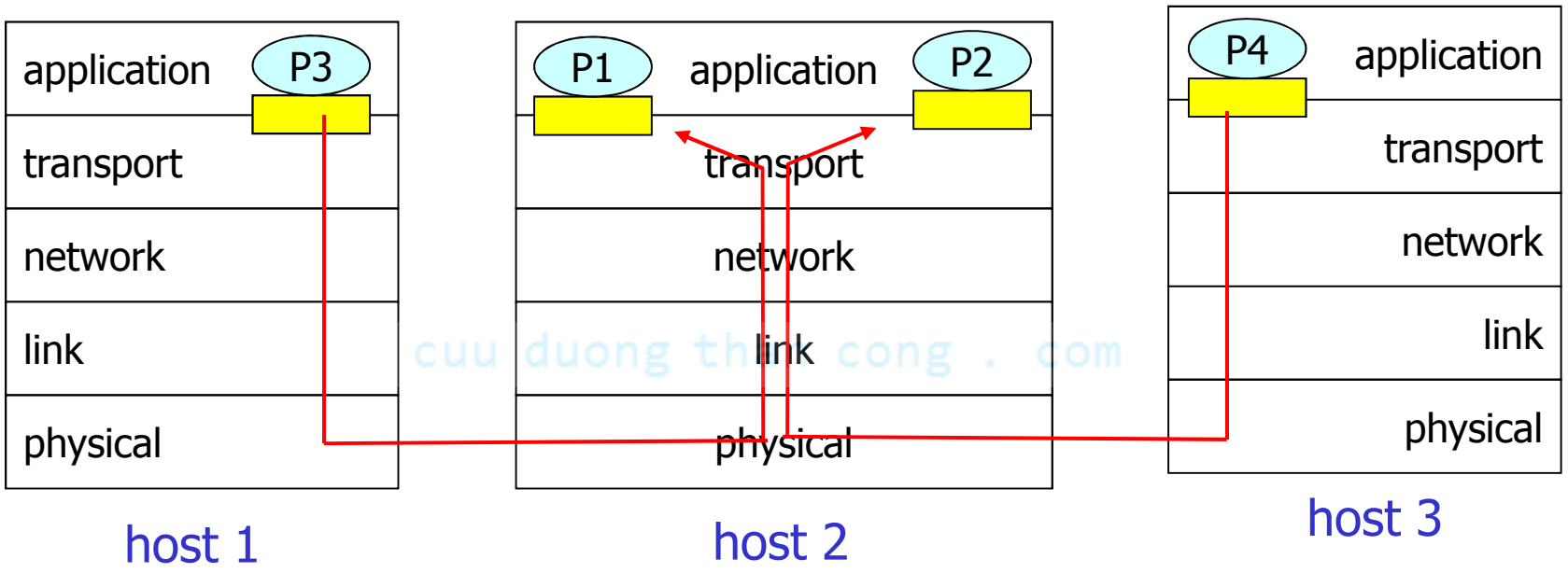
Demultiplexing ở host nhận:

Chuyển segment nhận được đến đúng socket

Multiplexing ở host gửi:

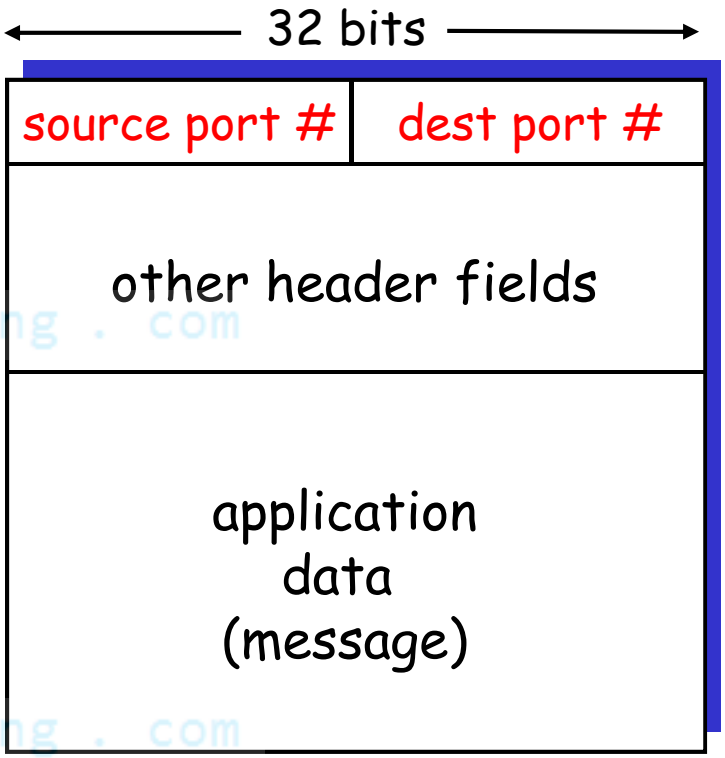
Thu thập các mẩu dữ liệu từ các socket, bao bọc mỗi mẩu dữ liệu trong 1 gói, có gắn thêm header (để dùng khi demultiplexing ở host nhận)

= socket = process



How demultiplexing works

- Host bên nhận nhận IP datagrams
  - Mỗi datagram có source IP address, destination IP address
  - Mỗi datagram mang 1 segment (là đơn vị dữ liệu ở Tầng Transport)
  - Mỗi segment có *source port number*, *destination port number* .
- Host bên nhận sử dụng địa chỉ IP và số hiệu port để chuyển giao segment đến socket tương ứng.
- Hai cách demultiplexing khác nhau ứng với cách truyền thông hướng kết nối (connection-oriented)và phi kết nối (connectionless)



TCP/UDP segment format

Connectionless demultiplexing

- Tạo sockets với số hiệu port:

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

- Khi host nhận được UDP segment
  - Kiểm tra destination port number trong segment
  - Chuyển UDP segment đến socket tương ứng với số hiệu port đó.

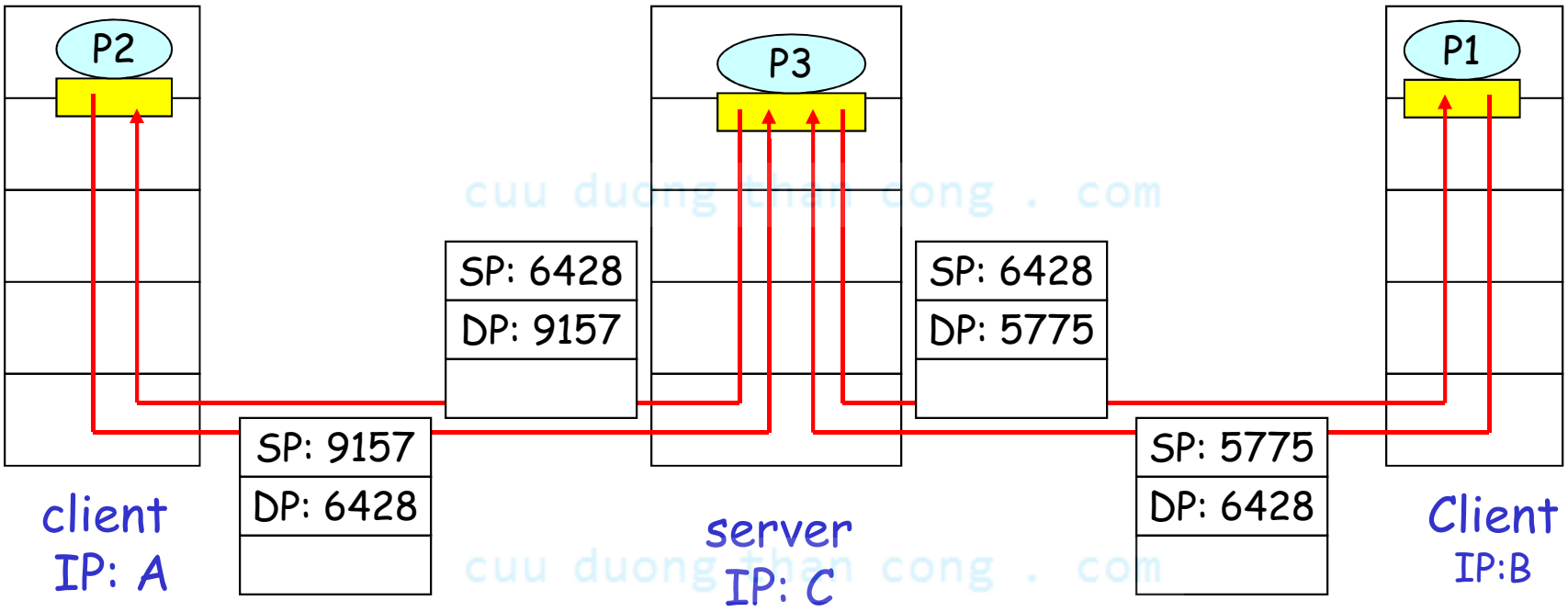
- Socket của UDP được xác định bằng cặp:

(dest IP address, dest port number)

=> Dù các IP Datagram có khác nhau tại **source IP adress** và/hoặc **source port number**, nhưng chúng có cùng **destination IP address** và **destination port number**, thì chúng cũng sẽ được chuyển đến cùng 1 socket.

Connectionless demux (cont)

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



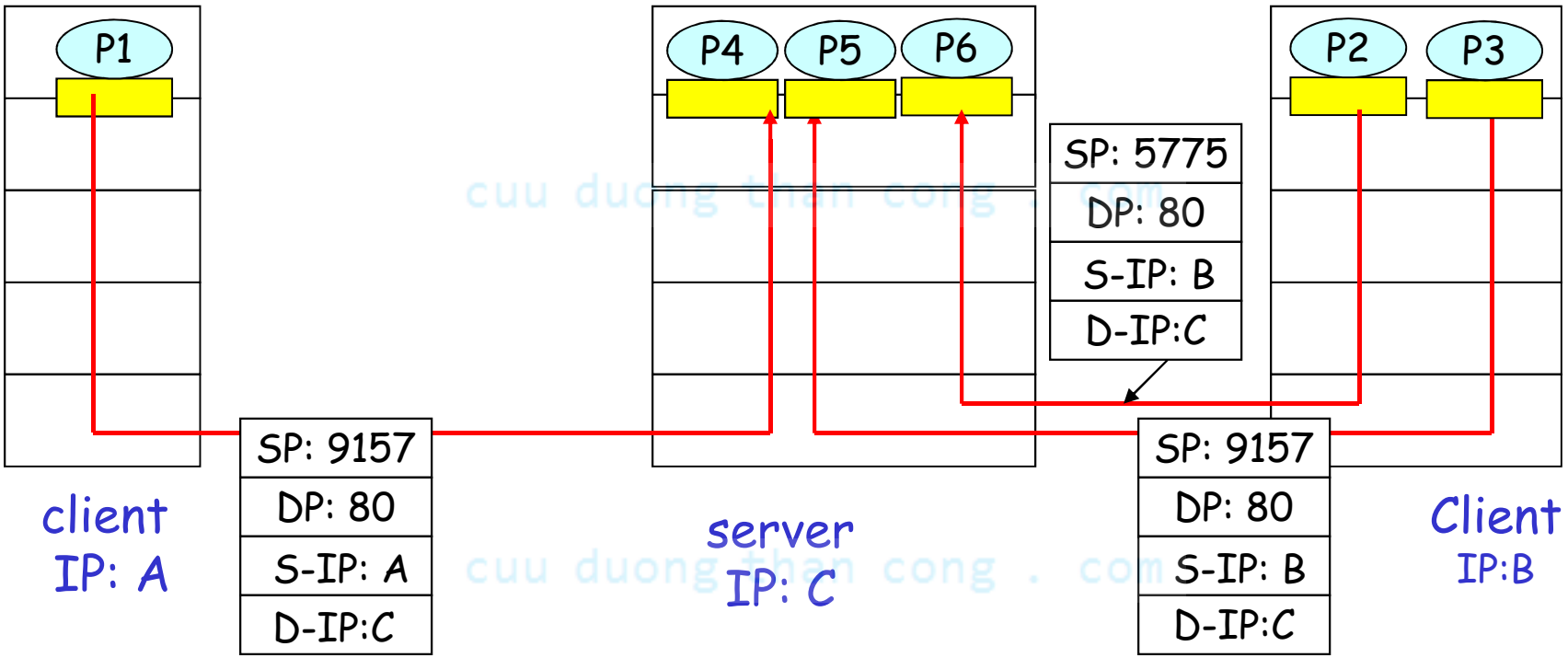
SP: Source port number; DP: Destination port number
SP nhằm cung cấp "địa chỉ quay về"

Connection-oriented demux

- TCP socket được xác định bằng bộ bốn:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- Host bên nhận sử dụng cả 4 giá trị này để chuyển/hướng segment tới socket tương ứng.

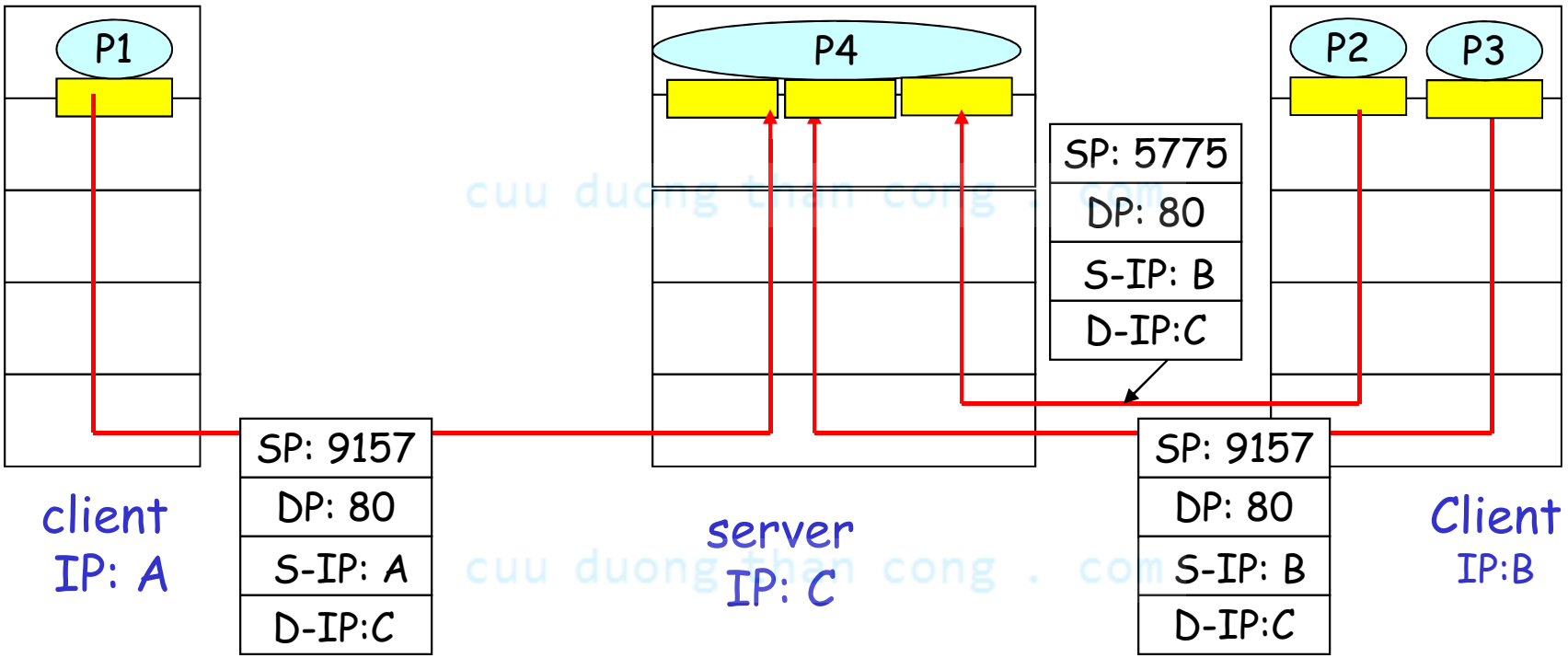
- Server host có thể cung cấp nhiều TCP socket đồng thời.
  - Mỗi socket được xác định bằng bộ bốn của chính nó.
- Web server có các socket khác nhau cho mỗi client kết nối đến.
  - non-persistent HTTP sẽ có các socket khác nhau cho mỗi request.

Connection-oriented demux (cont)



Dù cho B và A vô tình sử dụng trùng 1 SP (9157), nhưng Server C vẫn truyền thông chính xác với từng Client B, A do chúng có S-IP khác nhau

Connection-oriented demux: Threaded Web Server



Vì lý do hiệu năng, Web Server sử dụng "một processs với nhiều socket". Khi đó, process sử dụng các sub-process (được gọi là thread - *tiểu trình*), mỗi thread phụ trách 1 socket

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

UDP: User Datagram Protocol [RFC 768]

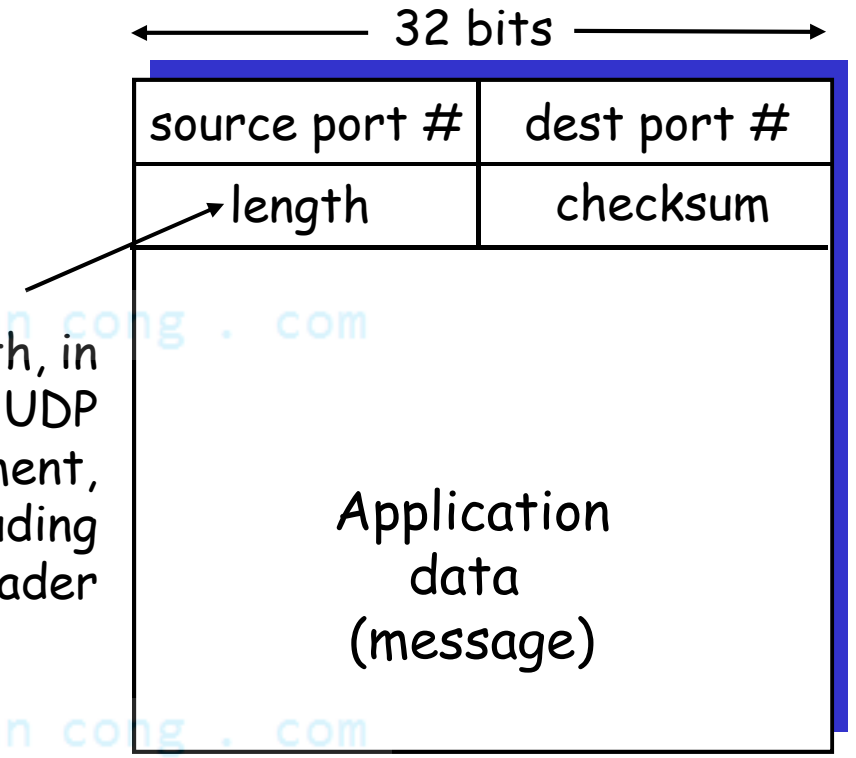
- Là giao thức vận chuyển đơn giản.
- UDP segments có thể bị:
  - Mất
  - Chuyển giao cho các ứng dụng ko đúng thứ tự
- *Phi kết nối [connectionless]:*
  - Trong UDP, ko có "thủ tục bắt tay" giữa bên gửi và bên nhận.
  - Mỗi UDP segment được xử lý độc lập với các segments khác.

Tại sao cần có UDP?

- Ko cần thiết lập kết nối (là yếu tố làm chậm)
- Đơn giản: ko cần quản lý trạng thái kết nối tại bên gửi và bên nhận.
- Kích thước phần header nhỏ (8 byte)
- Ko quan tâm đến sự ùn tắt mạng

UDP: more

- Thường được các ứng dụng streaming multimedia sử dụng, do chúng
  - Cho phép mất d/liệu
  - Cần tốc độ (cao) ổn định
- Các ứng dụng khác dùng UDP
  - DNS
  - SNMP
- Truyền d/liệu bảo đảm với UDP: cần bổ sung khả năng **phát hiện và sửa lỗi** ở tầng Ứng dụng.



UDP segment format

UDP checksum

Mục đích: phát hiện "lỗi" (e.g., flipped bits) trong các segment được truyền theo UDP.

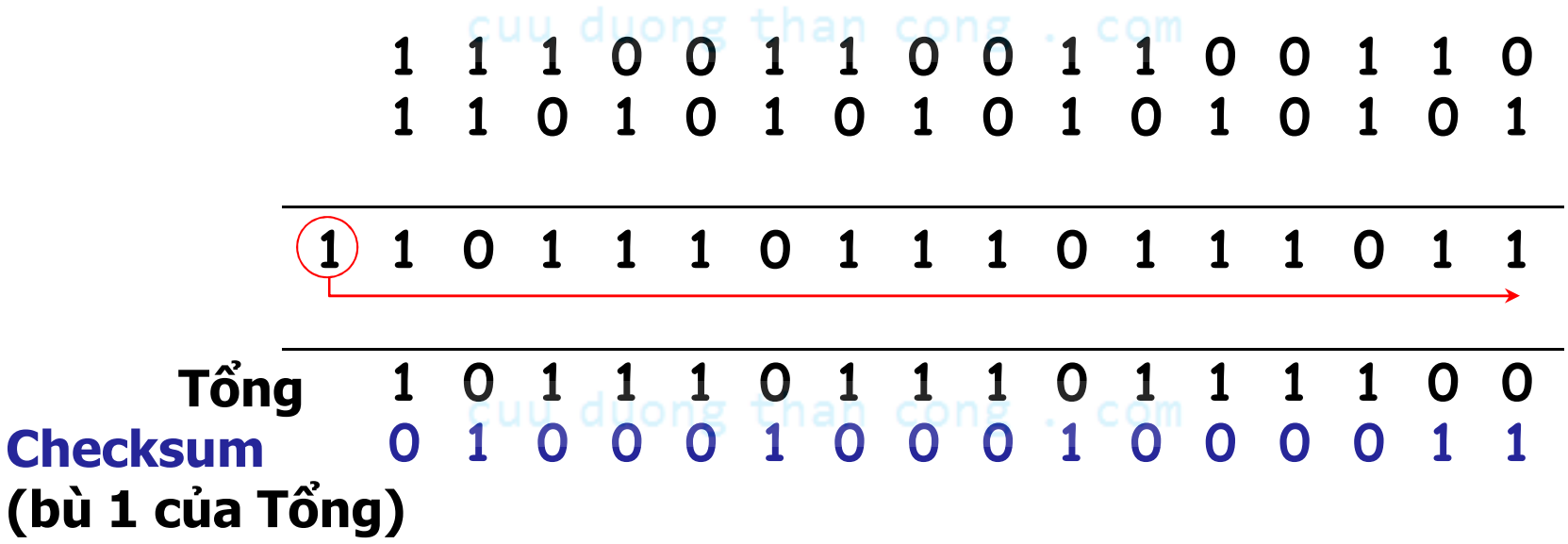
- Bên gửi:
- Coi nội dung segment như chuỗi các số nguyên 16 bit.
- Checksum: bù 1 của tổng các số nguyên 16 bit (trong nội dung segment)
- Bên gửi đặt trị checksum vào trường checksum trong UDP segment.

Bên nhận:

- Tính lại checksum của segment nhận được.
- Kiểm tra trị tính được có bằng với trị checksum trong segment nhận được.
  - KHÔNG BẰNG –lỗi được phát hiện
  - BẰNG- không có lỗi được phát hiện (*nhưng vẫn có thể có lỗi*).

Internet Checksum Example

- Ghi chú
  - Khi cộng, bít nhớ ở ngoài cùng bên trái cần được thêm vào kết quả=> tạo thành Tổng.
- Ví dụ: cộng 2 số nguyên 16-bit.

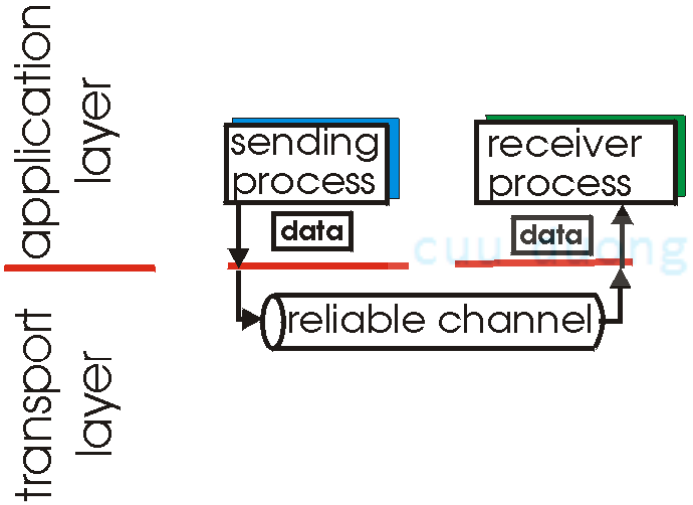


Chapter 3 outline

- 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

Principles of Reliable data transfer

- Là chủ đề quan trong ở tầng application, transport và link
- Nằm trong 10 chủ đề về mạng quan trọng nhất.

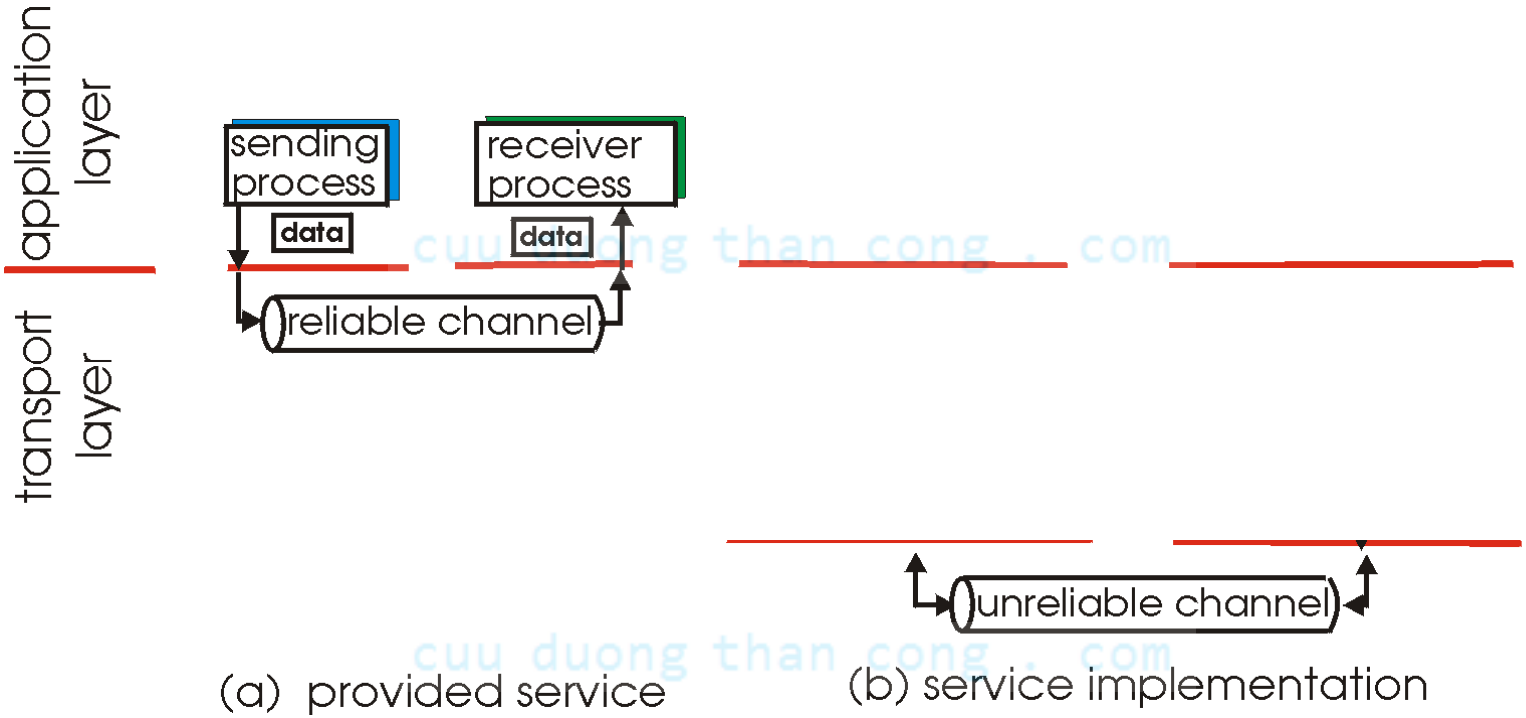


(a) provided service

- Các đặc tính của *kênh truyền không đáng tin cậy* (*unreliable channel*) sẽ quyết định độ phức tạp của *giao thức truyền dữ liệu đáng tin cậy (reliable data transfer protocol – viết tắt rdt)*.

Principles of Reliable data transfer

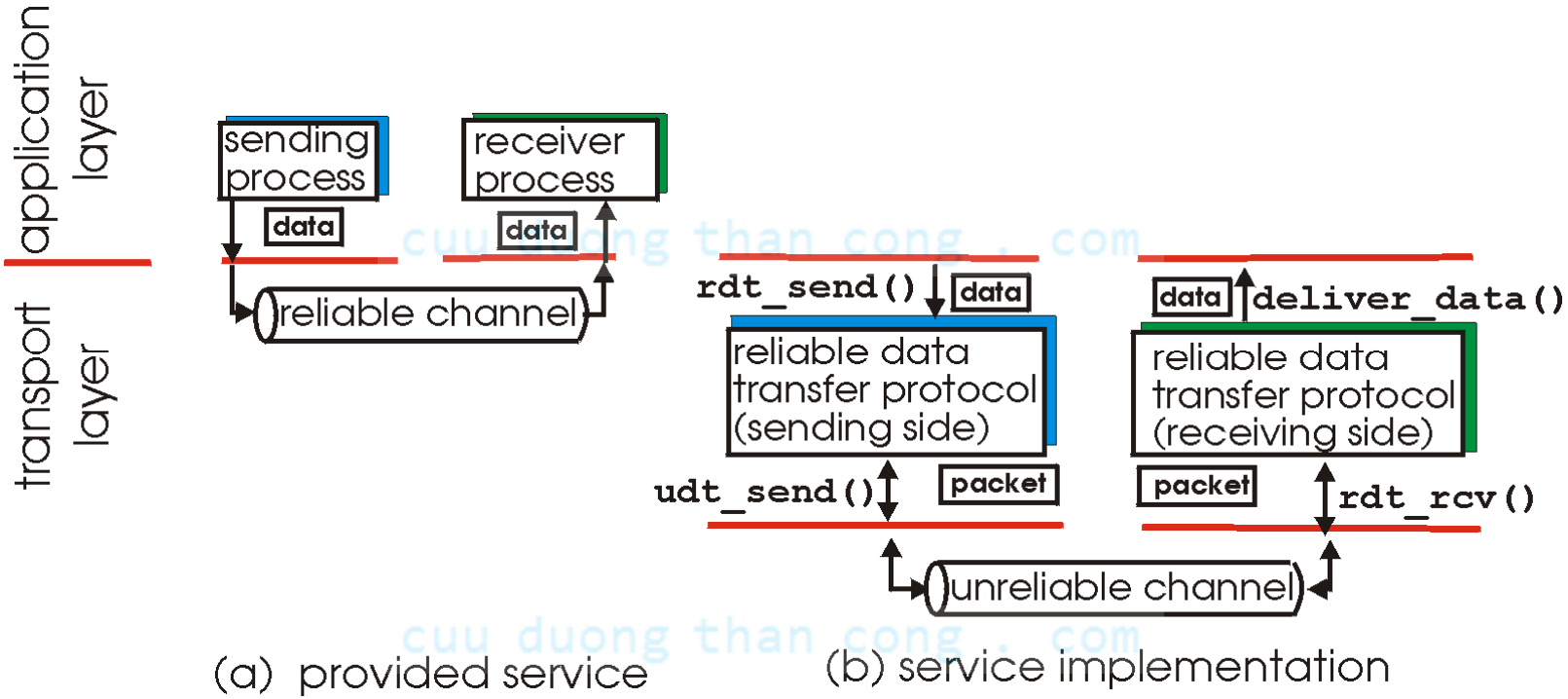
- Là chủ đề quan trọng ở tầng application, transport và link
- Nằm trong 10 chủ đề về mạng quan trọng nhất.



- Các đặc tính của *kênh truyền không đáng tin cậy* (*unreliable channel*) sẽ quyết định độ phức tạp của *giao thức truyền dữ liệu đáng tin cậy (reliable data transfer protocol – viết tắt rdt)*

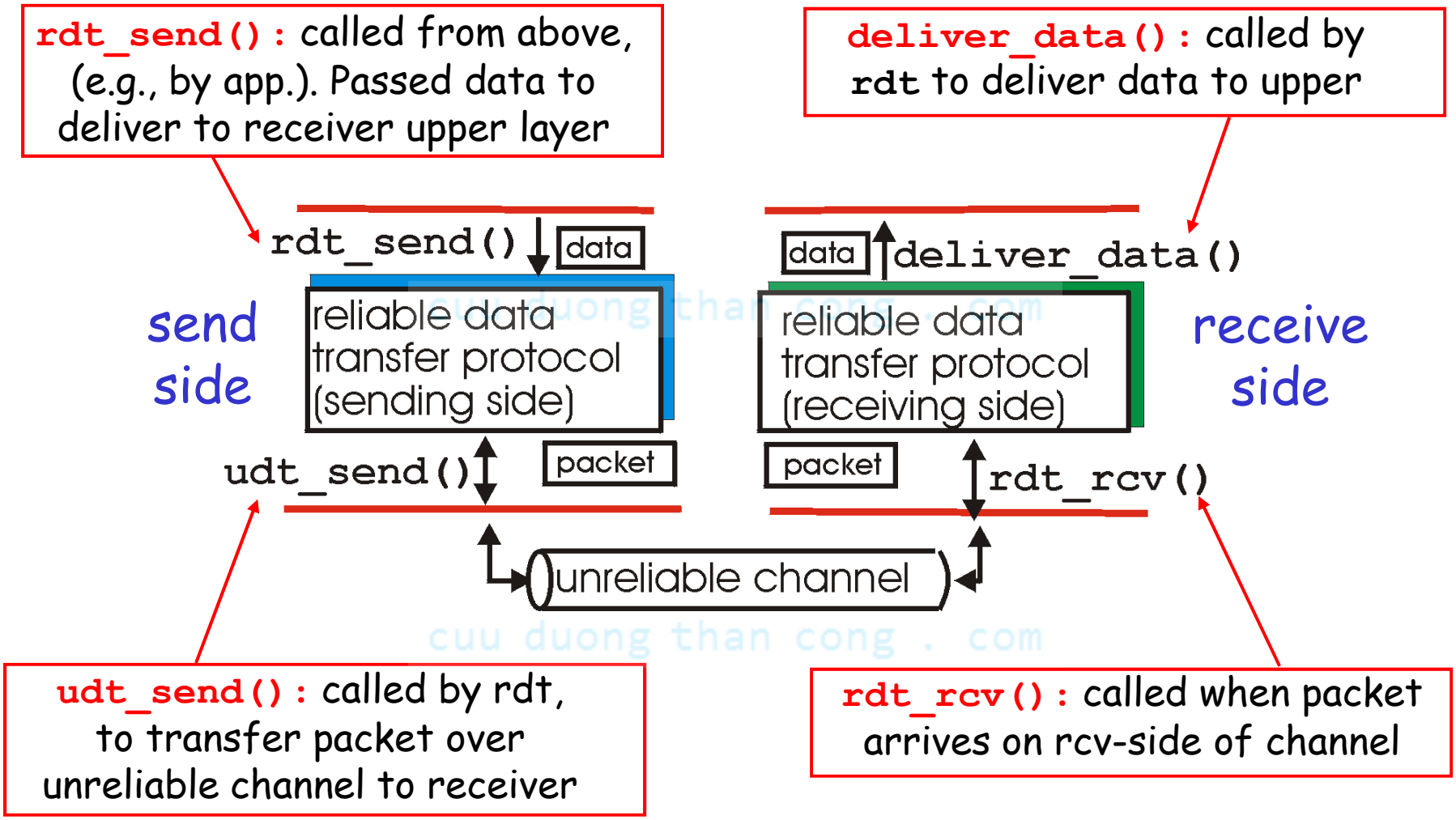
Principles of Reliable data transfer

- Là chủ đề quan trong ở tầng application, transport và link
- Nằm trong 10 chủ đề quan trọng nhất về mạng máy tính.



- Các đặc tính của *kênh truyền không đáng tin cậy* (*unreliable channel*) sẽ quyết định độ phức tạp của *giao thức truyền dữ liệu đáng tin cậy (reliable data transfer protocol – viết tắt rdt)*

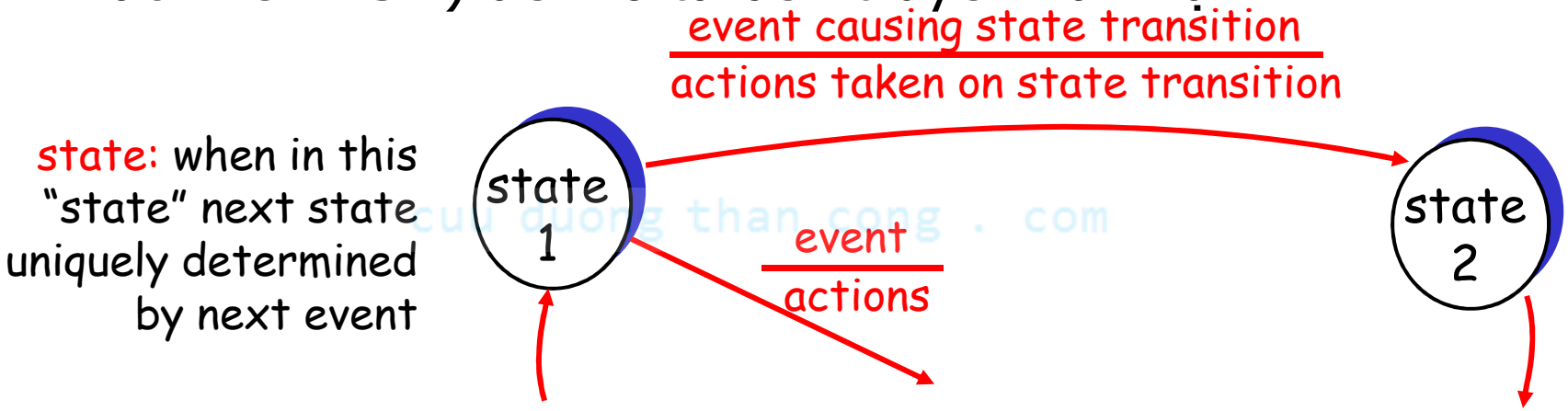
Reliable data transfer: getting started



Reliable data transfer: getting started

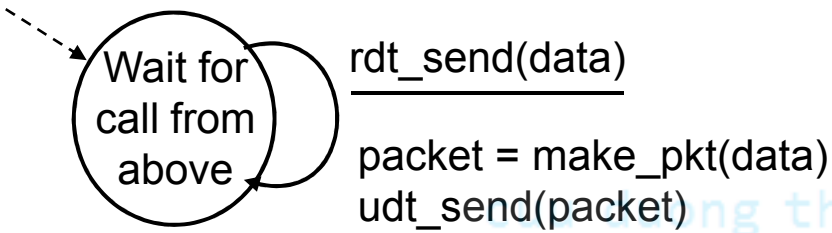
Chúng ta sẽ:

- Phát triển dần giao thức *truyền dữ liệu đáng tin cậy* (rdt) cho cả 2 phía truyền và nhận.
- Chỉ xem xét trường hợp truyền dữ liệu 1 chiều
  - Nhưng thông tin điều khiển sẽ chạy theo cả 2 chiều!
- Sử dụng *máy trạng thái hữu hạn* (finite-state machine - FSM) để mô tả bên truyền và nhận.

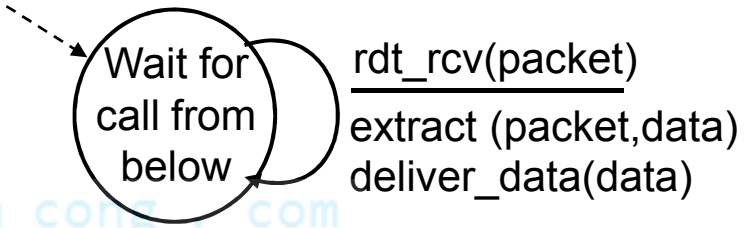


Rdt1.0: reliable transfer over a reliable channel

- Giả định: Kênh truyền bên dưới truyền dữ liệu hoàn toàn đáng tin cậy.
  - Không sai bit.
  - Không mất gói tin
- Có 2 FSMs riêng cho bên truyền và nhận:
  - Bên truyền gửi dữ liệu xuống kênh truyền bên dưới.
  - Bên nhận đọc dữ liệu từ kênh truyền bên dưới.



Bên truyền

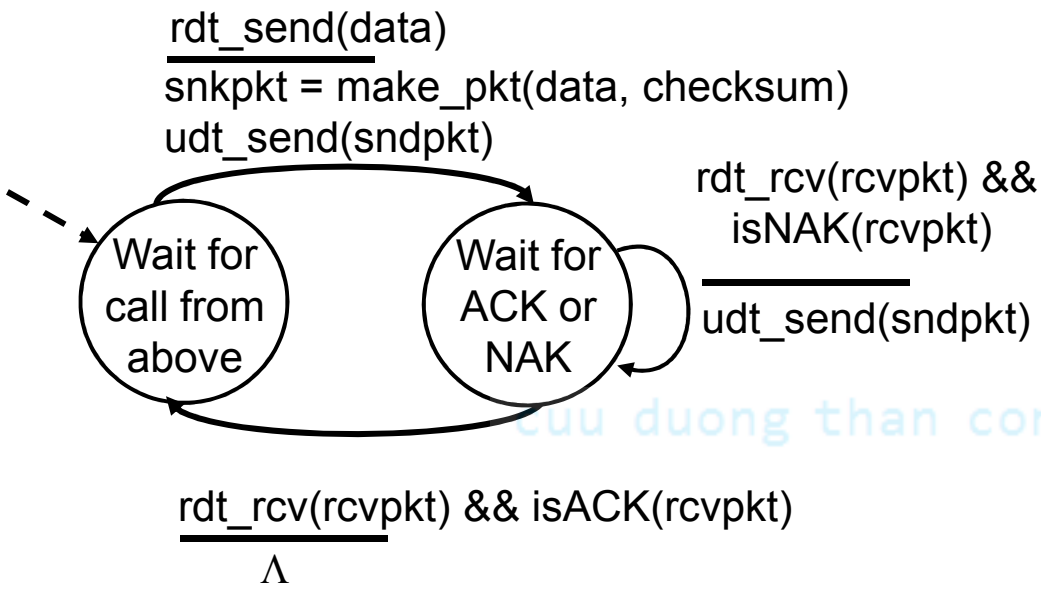


Bên nhận

Rdt2.0: channel with bit errors

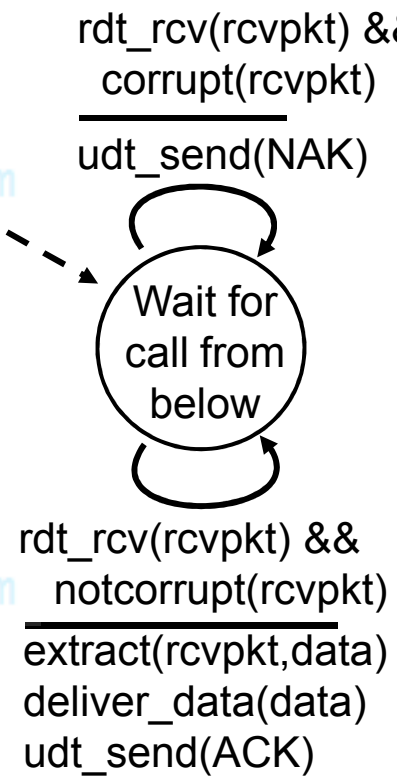
- Kênh truyền bên dưới có thể truyền bit bị sai.
  - Dùng checksum để phát hiện lỗi bit sai
- *Vấn đề*: làm thế nào để sửa lỗi:
  - *Xác nhận đúng(acknowledgements -ACKs):* bên nhận nói rõ cho bên gửi biết là gói tin đã được nhận đúng.
  - *Xác nhận lỗi: negative acknowledgements (NAKs):* bên nhận nói rõ cho bên gửi biết là gói tin nhận được đã bị lỗi.
  - Bên gửi truyền lại gói tin khi nhận được NAK.
- Cơ chế mới trong **`rdt2.0`** (beyond **`rdt1.0`**):
  - Phát hiện lỗi (error detection)
  - Phản hồi của bên nhận: bên nhận gửi cho bên gửi các thông điệp xác nhận (thông điệp ACK, hoặc thông điệp NAK)

rdt2.0: FSM specification

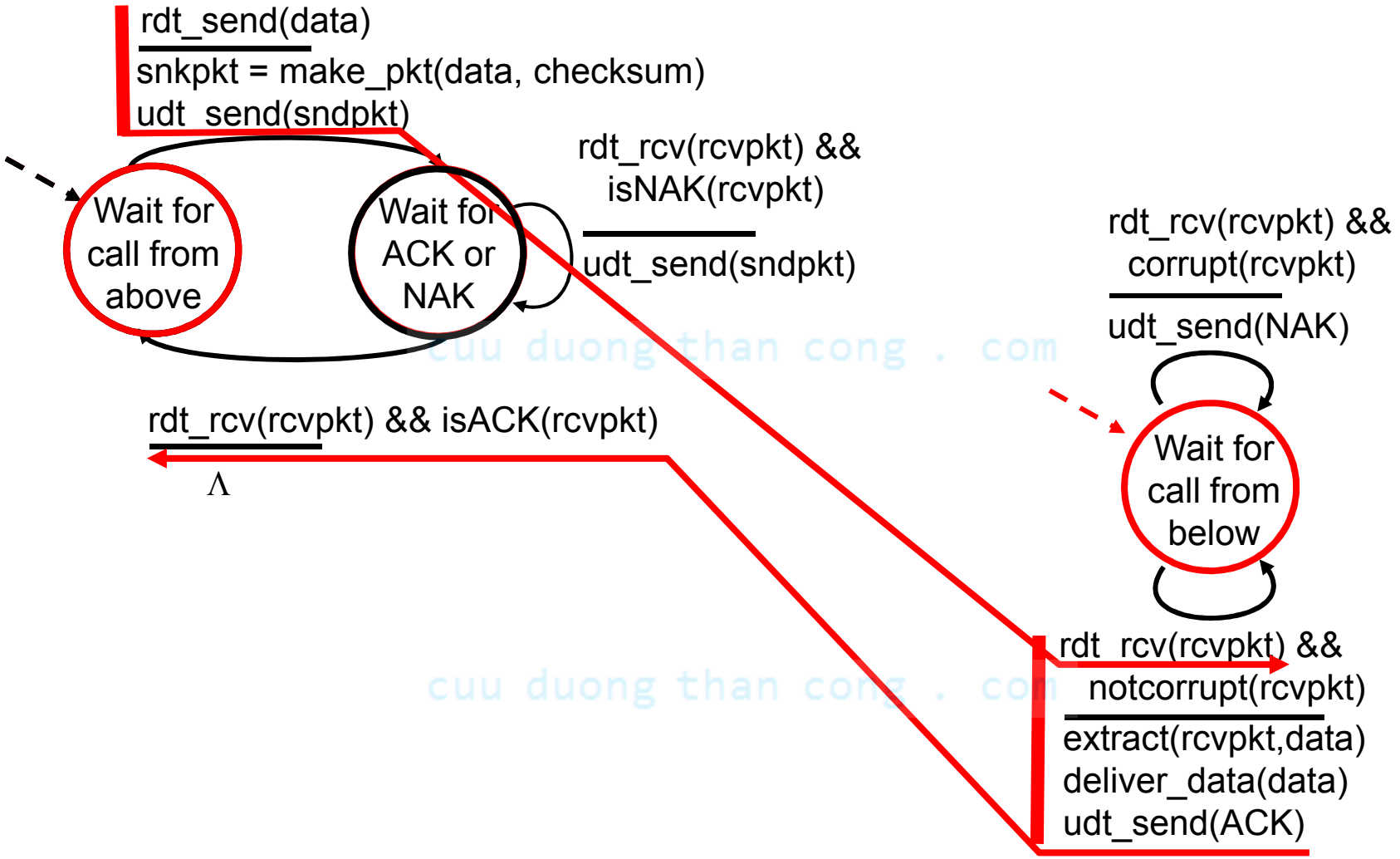


Bên truyền

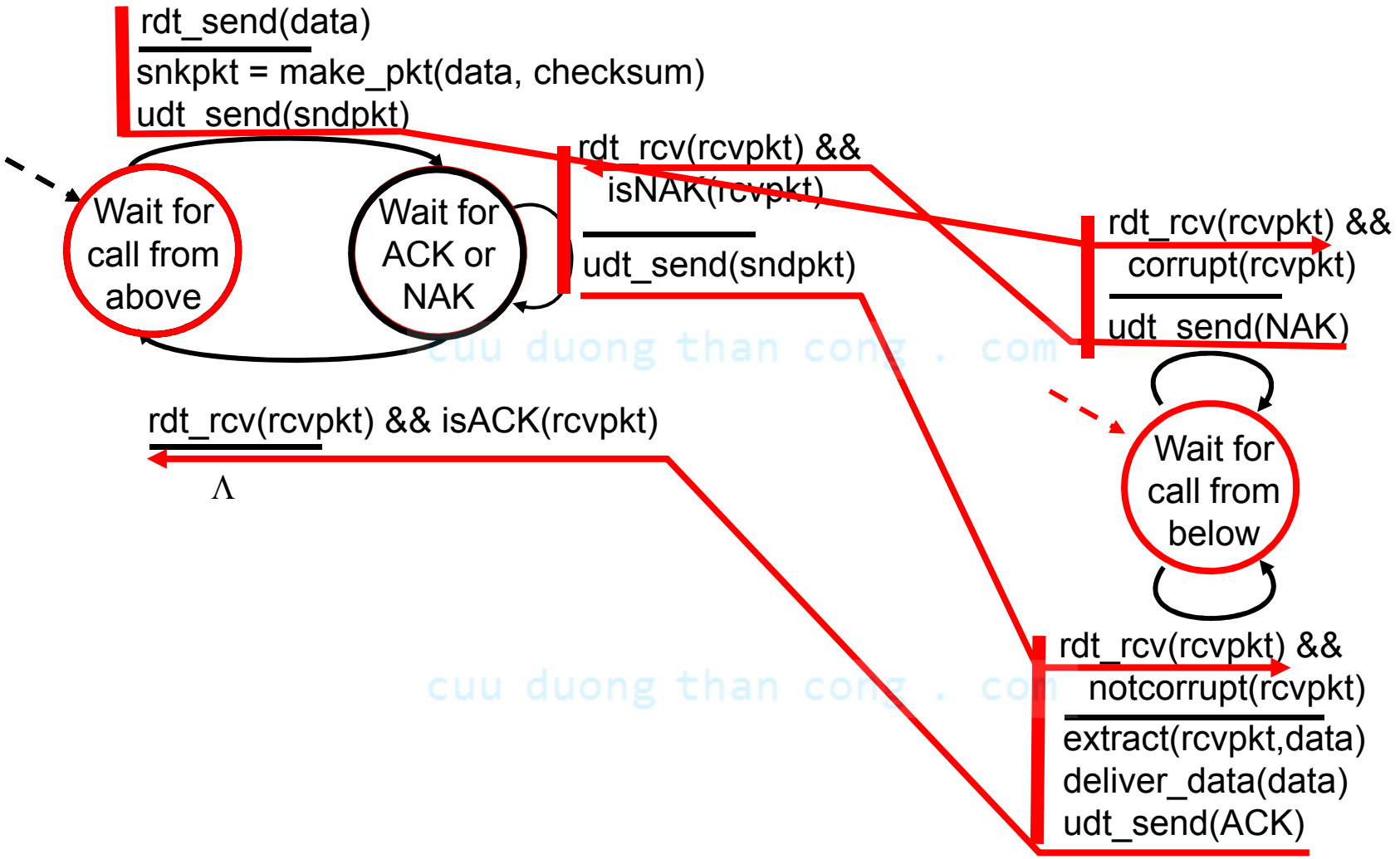
Bên nhận



rdt2.0: operation with no errors



rdt2.0: error scenario



rdt2.0 has a fatal flaw!

Điều gì xảy ra nếu ACK/NAK bị mất/lỗi?

- Bên gửi không biết những gì đã xảy ra ở bên nhận!
- Bên gửi cần truyền lại: nhưng có thể xảy ra hiện tượng trùng lặp gói tin.

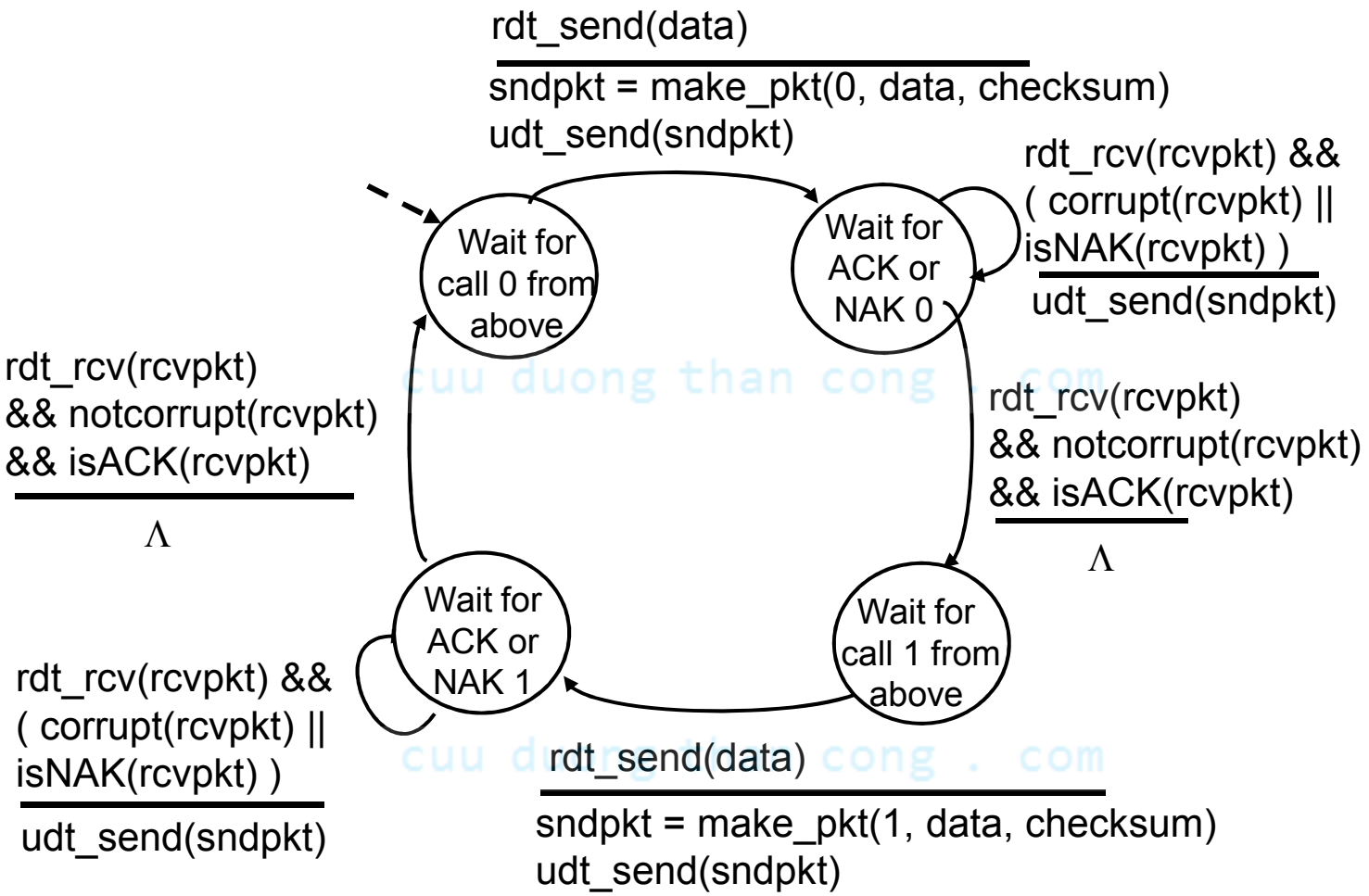
Xử lý trùng lặp gói tin:

- Bên gửi truyền lại gói tin hiện thời nếu như ACK/NCK bị mất/lỗi, ko thể nhận ra.
- Bên gửi thêm 1 số thứ tự vào mỗi gói tin.
- Bên nhận vứt bỏ các gói tin bị trùng (so số thứ tự), mà không truyền lên tầng trên.

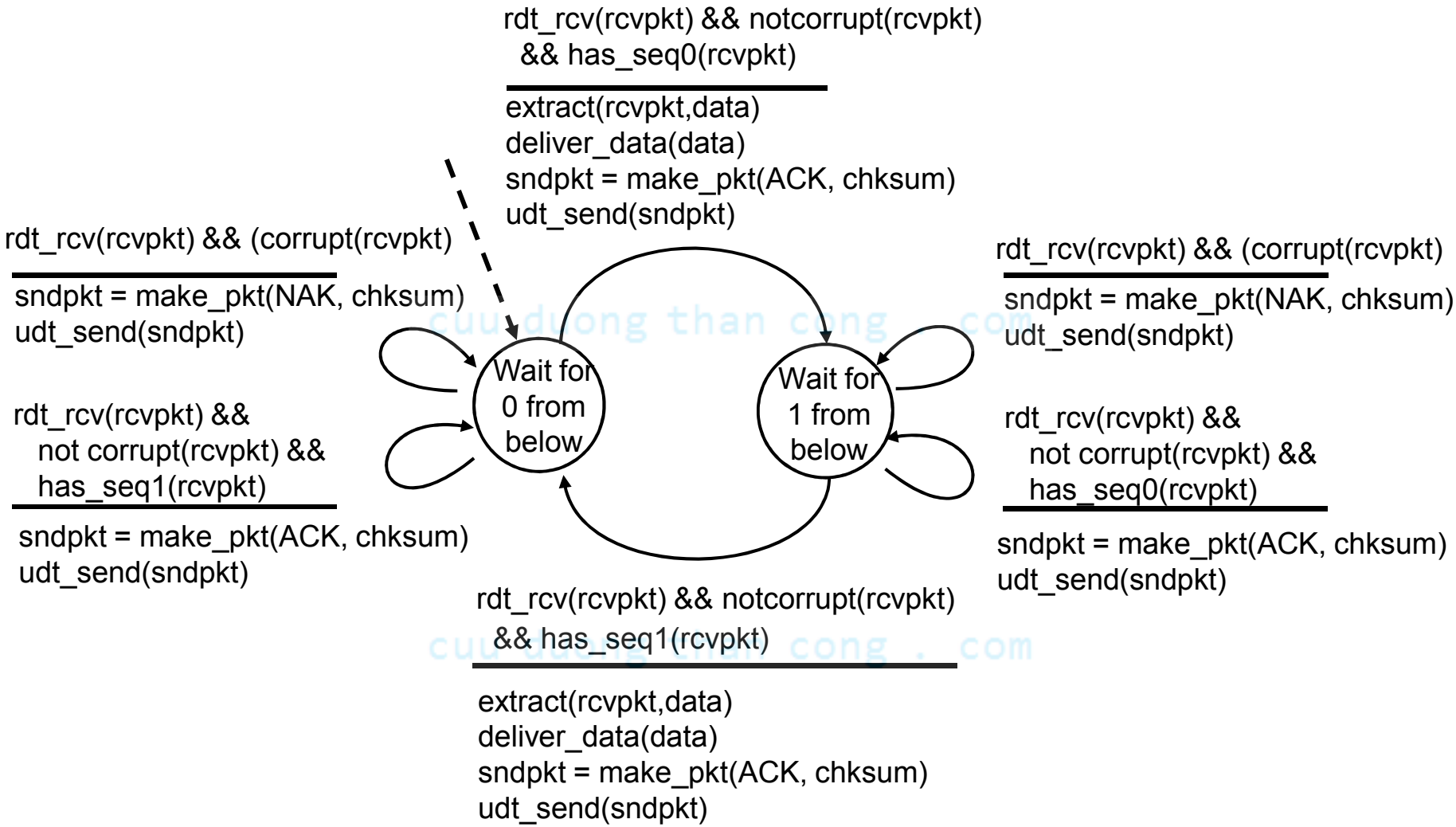
stop and wait

Bên gửi gửi 1 gói tin, sau đó chờ bên nhận phản ứng lại.

rdt2.1: sender, handles garbled ACK/NAKs



rdt2.1: receiver, handles garbled ACK/NAKs



rdt2.1: discussion

Bên gửi:

- Số thứ tự được thêm vào gói tin.
- Dùng hai giá trị (0,1) cho số thứ tự là đủ. Vì sao?
- Cần phải kiểm tra lại xem các xác nhận ACK/NAK có bị lỗi không.
- Gấp đôi số trạng thái
  - ở mỗi trạng thái, phải "nhớ" gói tin hiện thời có số thứ tự là 0 hay 1.

Bên nhận:

- Phải kiểm tra xem liệu gói tin nhận được có bị trùng lặp hay ko?
  - Trạng thái cho biết mong chờ gói tin có stt là 0 hay 1
- Lưu ý: Sau khi gửi ACK/NAK, bên nhận không thể biết liệu gói tin ACK/NAK sau cùng có đến được ở bên gửi hay không.

rdt2.2: a NAK-free protocol

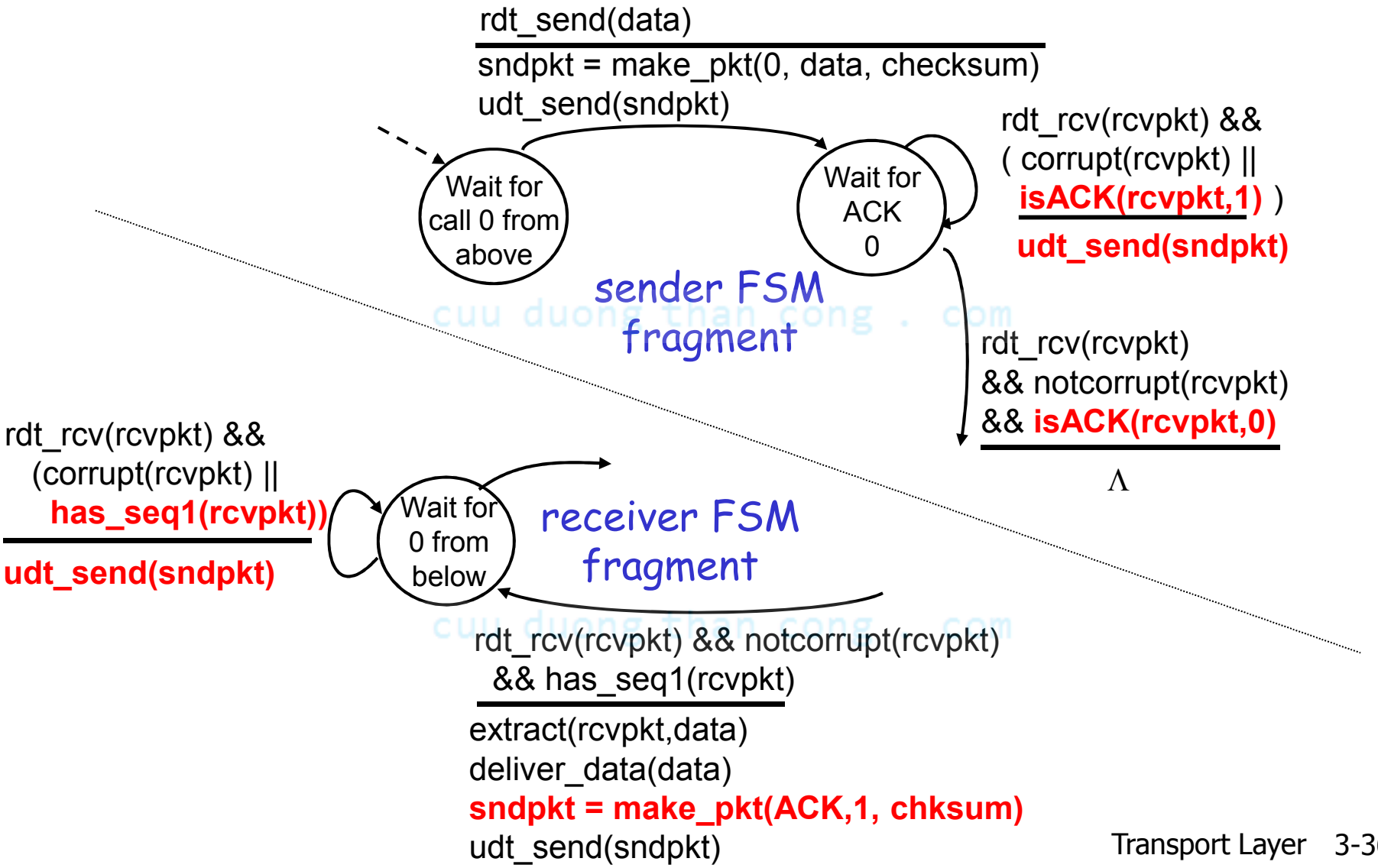
- Làm việc tương tự như rdt2.1, nhưng chỉ sử dụng ACKs.
- Thay cho NAK, bên nhận có thể sử dụng ACK với số thứ tự SAI, để tạo ra cùng tác động của như NAK trong rdt2.1 : bắt bên gửi phải truyền lại
  - Bên nhận phải **đưa số thứ tự** vào gói tin ACK
- Việc trùng lắp ACK ở bên gửi dẫn đến cùng một hành động như khi nhận NAK: *gửi lại gói tin hiện thời* .

Slide 35

c1 Ý này thay thế cho ý trước, yêu cầu SV sửa lại trong slide.
cuong, 10/12/2010

cuu duong than cong . com

rdt2.2: sender, receiver fragments

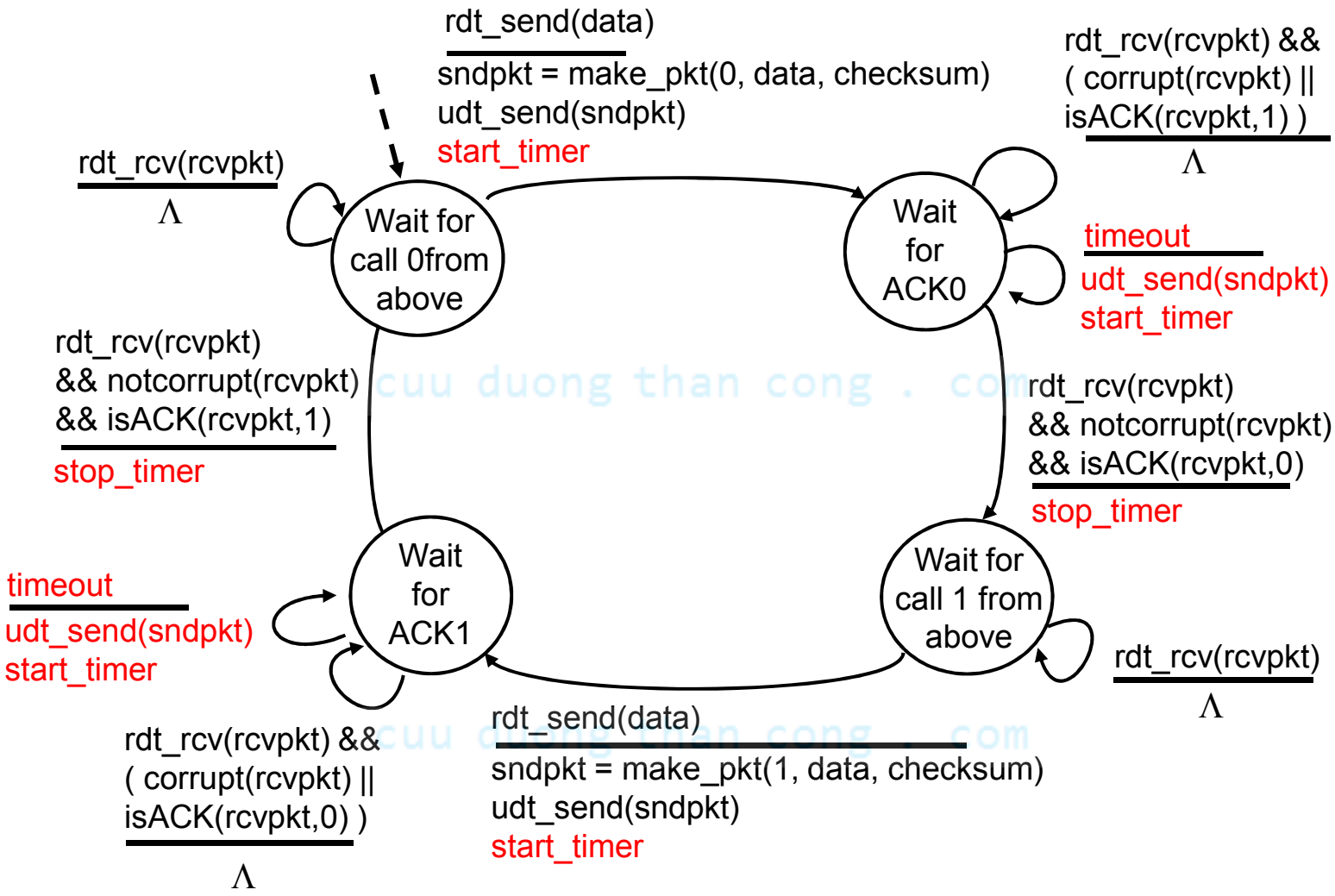


rdt3.0: channels with errors *and* loss

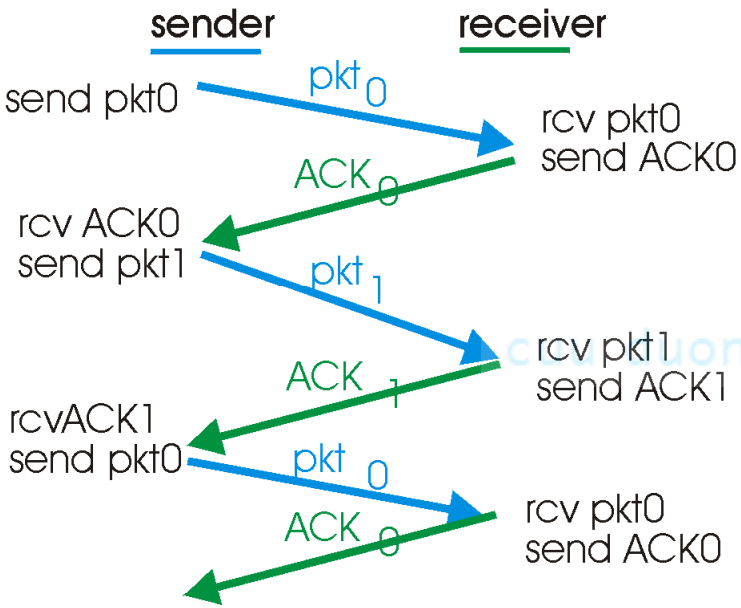
- Giả định mới: kênh truyền bên dưới cũng có thể làm mất gói tin (dữ liệu hoặc ACK)
  - Kỹ thuật checksum, đánh số thứ tự, dùng ACKs, kỹ thuật truyền lại là cần thiết, nhưng chưa đủ

- Một cách giải quyết: bên gửi chờ gói tin ACK một thời gian "hợp lý".
- Bên gửi sẽ truyền lại nếu không nhận được gói tin ACK trong khoảng thời gian này.
- Nếu gói tin (dữ liệu hoặc ACK) chỉ chậm đến (chứ không mất):
  - Việc truyền lại sẽ gây tình trạng trùng lắp gói tin, nhưng sử dụng *số thứ tự* sẽ xử lý được tình trạng này.
  - Bên nhận phải chỉ định số thứ tự của gói tin ACK
- Cần đồng hồ đếm giờ.

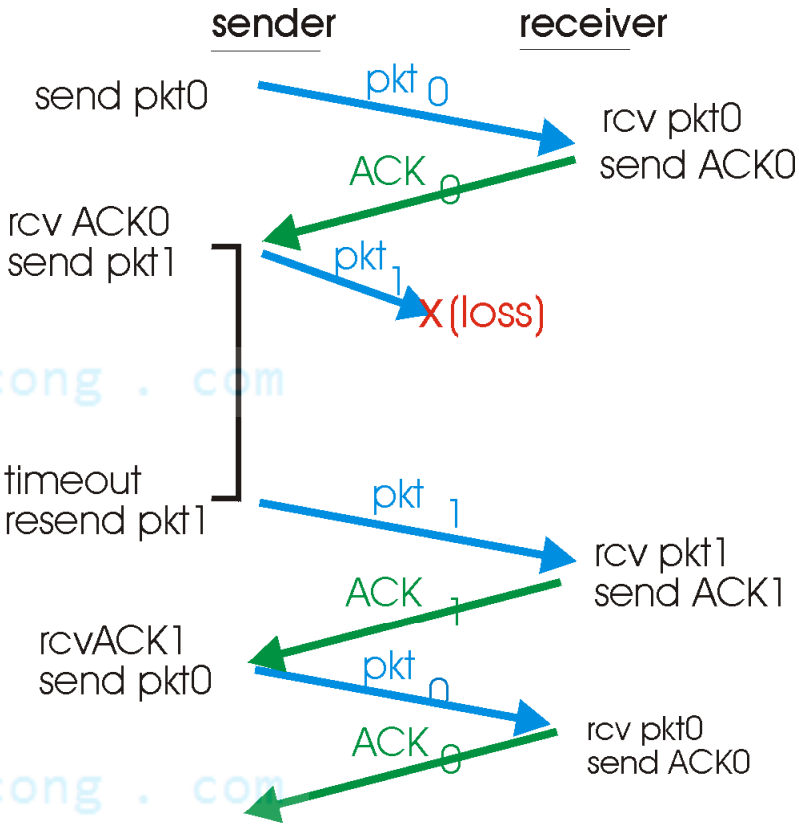
rdt3.0 sender



rdt3.0 in action

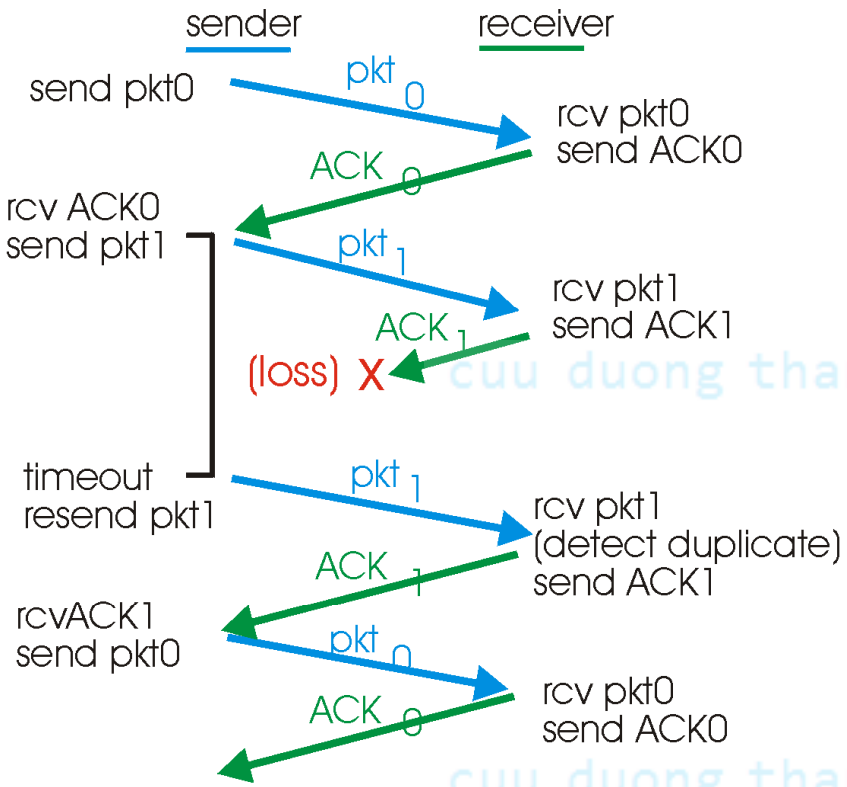


(a) operation with no loss

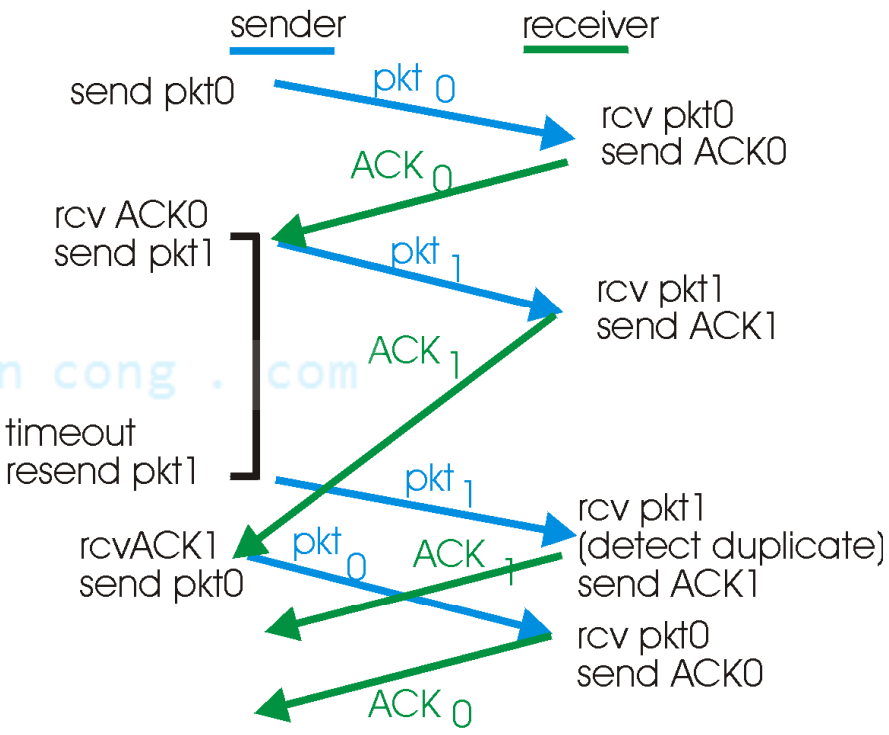


(b) lost packet

rdt3.0 in action



(c) lost ACK



(d) premature timeout

Performance of rdt3.0

- rdt3.0 có thể vận hành trong điều kiện thực tế, nhưng chưa được hiệu quả.
- VD: cho đường truyền có transmission rate 1 Gbps, có độ trễ lan truyền (prop. delay) 15 ms, với gói tin 8000 bit:

Transmission delay

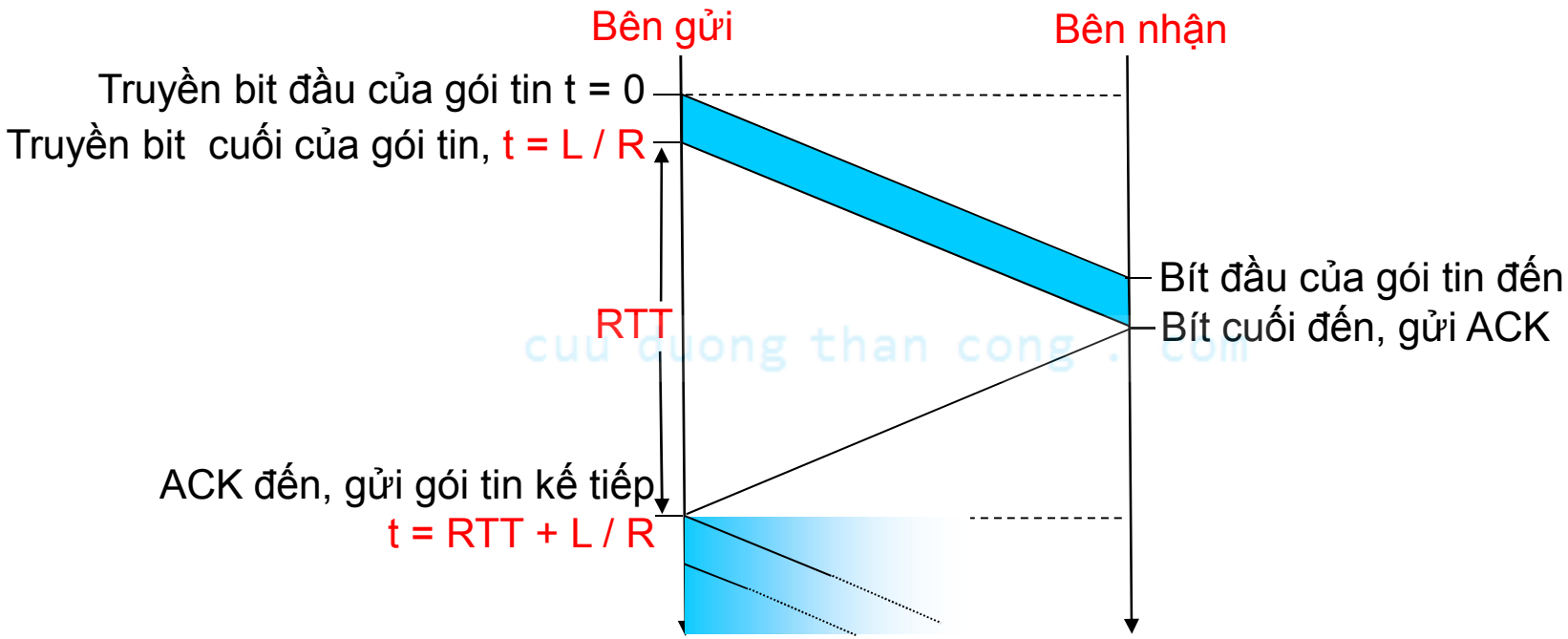
$$d_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bps}} = 8 \times 10^{-3} ms$$

- Gọi U_{sender}: hiệu quả dùng đường truyền = phần thời gian sử dụng đường truyền cho việc truyền dữ liệu

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

- Với đường truyền 1 Gbps, chỉ truyền được gói tin 1KB sau mỗi 30 ms => tức truyền dược 33kB/sec
- Giao thức rdt3.0 làm giới hạn khả năng của đường truyền vật lý!

rdt3.0: stop-and-wait operation

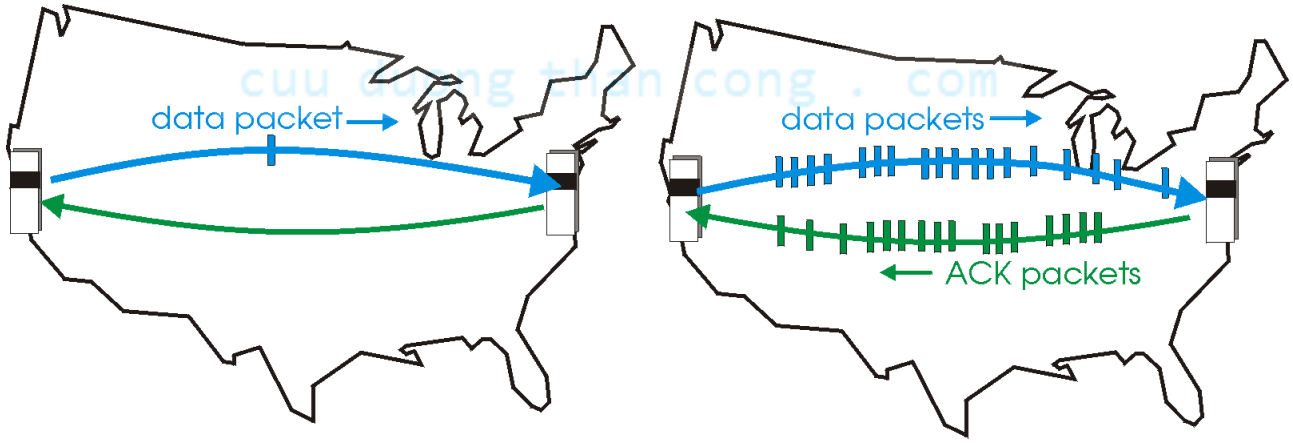


$$U_{sender} = \frac{L/R}{RTT + L/R} = \frac{.008}{30.008} = 0.00027$$

Pipelined protocols

Pipelining: bên gửi cho phép gửi nhiều gói tin cùng lúc, dù chưa nhận được ACK.

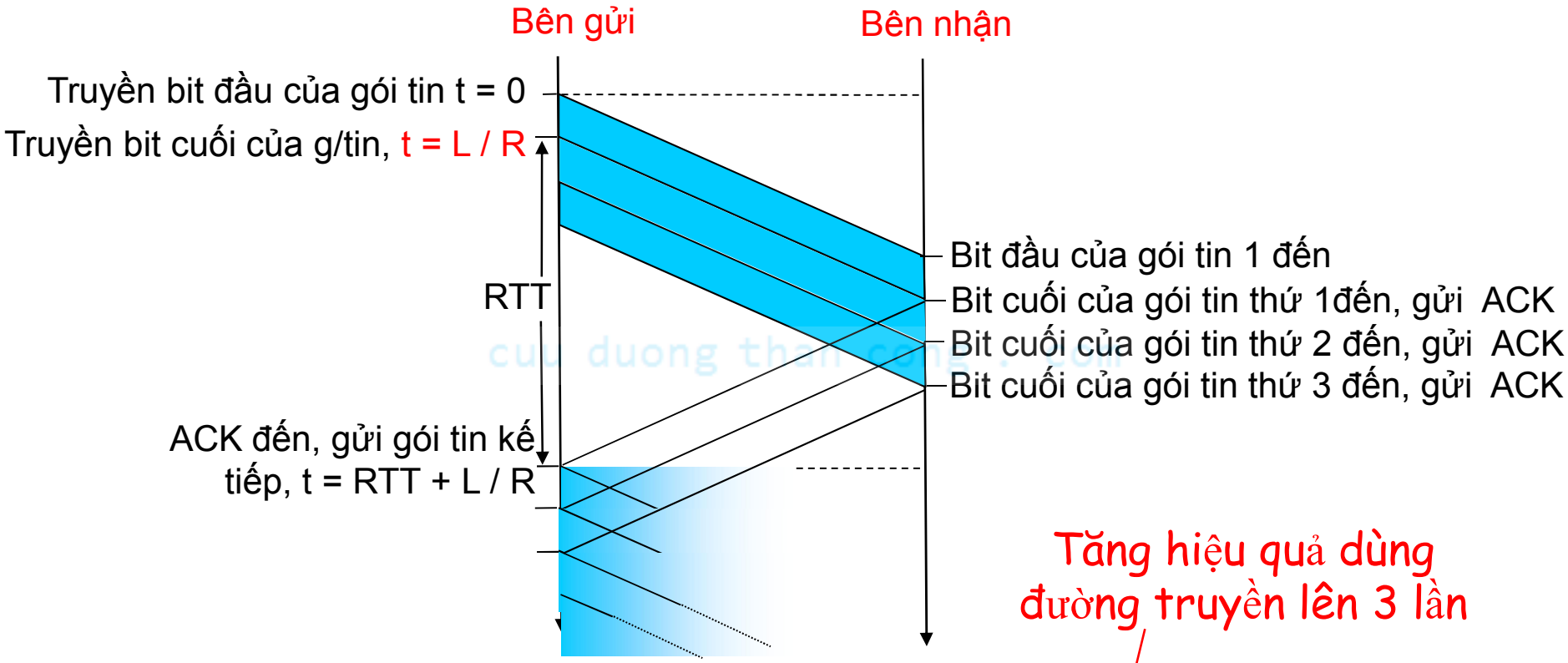
- Vùng đánh số thứ tự phải được mở rộng thêm.
- Cần vùng đệm ở cả bên gửi và bên nhận



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

- Hai giao thức dạng pipeline: *go-Back-N, selective repeat*

Pipelining: increased utilization



Tăng hiệu quả dùng đường truyền lên 3 lần

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

Pipelining Protocols

Go-back-N: tổng quan

- *Bên gửi:* cho phép lên đến N gói-tin-chưa-được-xác-nhận trong pipeline
- *Bên nhận:* chỉ gửi các cumulative ACKs
  - Không gửi ACK nếu các gói-tin-dữ-liệu đến ko đúng thứ tự.
- *Bên gửi:* đếm giờ cho gói tin "cũ nhất" chưa-được-xác-nhận
  - Nếu quá giờ: truyền lại tất cả các gói-tin-chưa-được-xác -nhận.

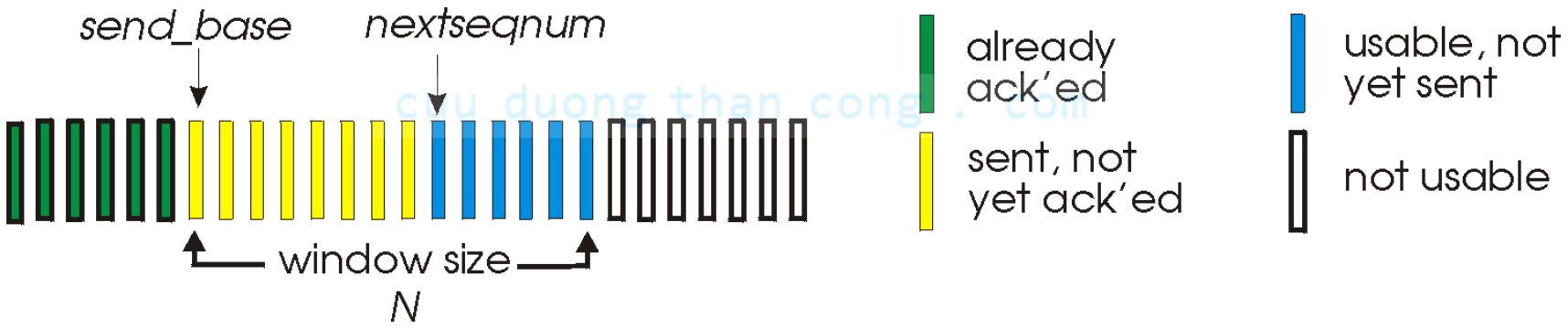
Selective Repeat: tổng quan

- *Bên gửi:* cho phép lên đến N gói-tin-chưa-được-xác-nhận trong pipeline
- *Bên nhận:* Xác nhận cho từng gói tin cụ thể
- *Bên gửi:* đếm giờ cho từng gói tin chưa được xác nhận
  - Nếu quá giờ: chỉ truyền lại gói tin chưa được xác nhận

Go-Back-N

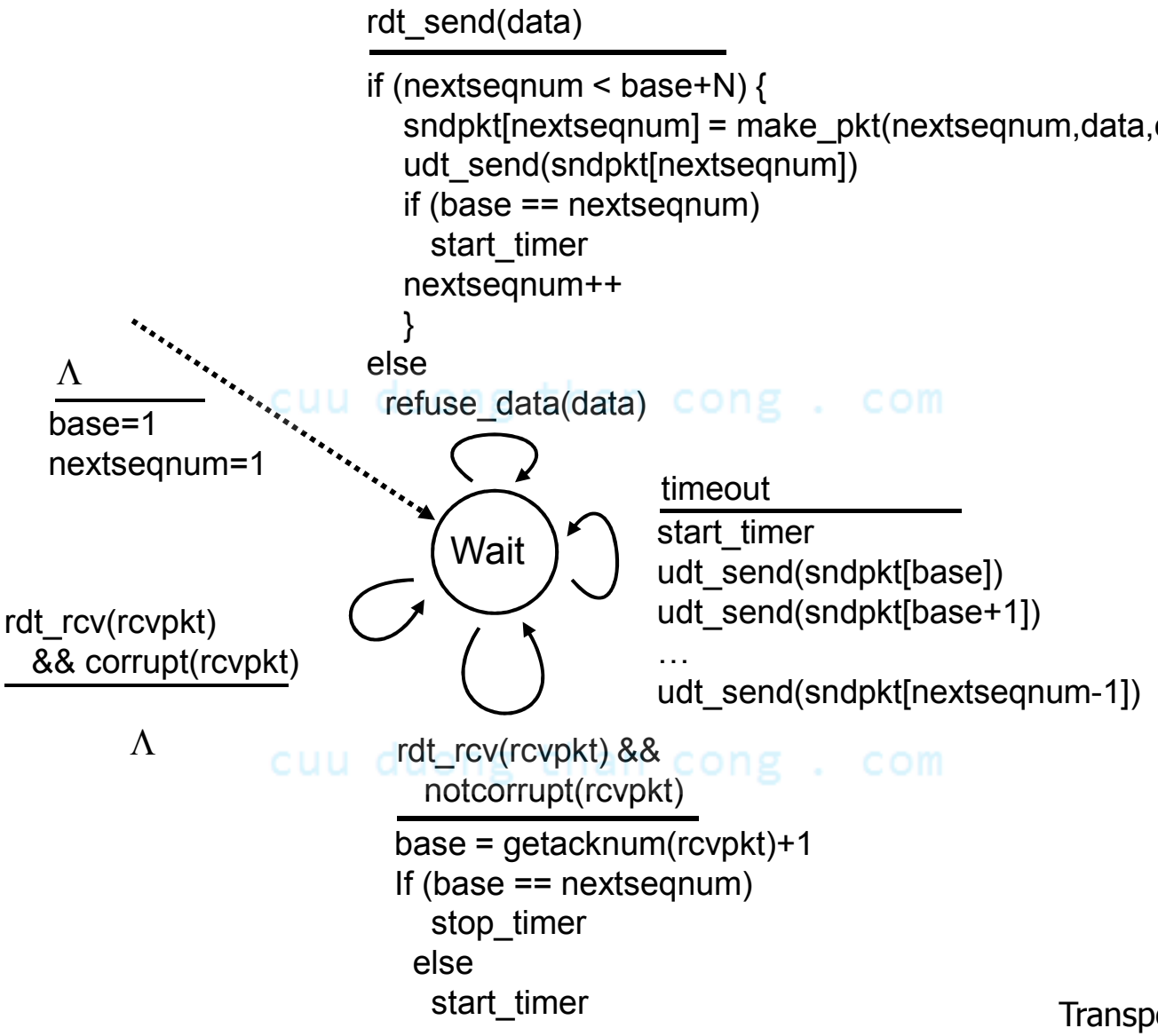
Bên gửi:

- Sử dụng k-bit cho trường số thứ tự trong phần header gói tin
- "cửa sổ" kích thước N, cho phép có nhiều gói-tin-chưa-được-xác-nhận.

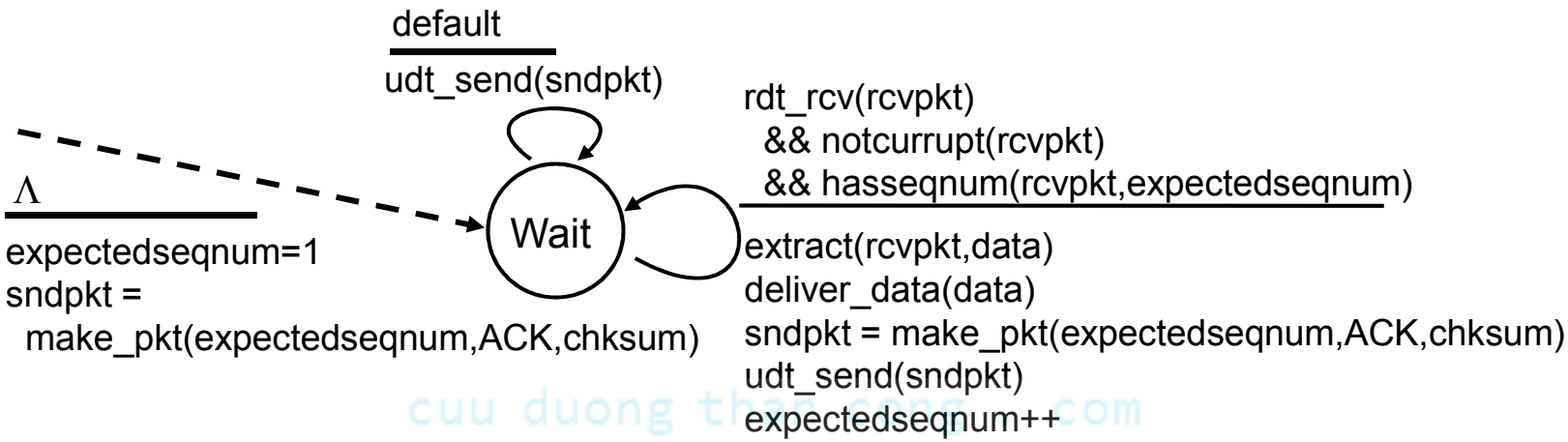


- ACK(n): xác nhận tất cả các gói tin tới số thứ tự n -"cumulative ACK"
  - Có thể nhận trùng gói tin xác nhận (xem bên nhận)
- Sử dụng bộ đếm giờ (timer) cho mỗi gói tin đang truyền
- *timeout(n):* truyền lại gói tin n và tất cả các gói tin có số thứ tự cao hơn trong cửa sổ.

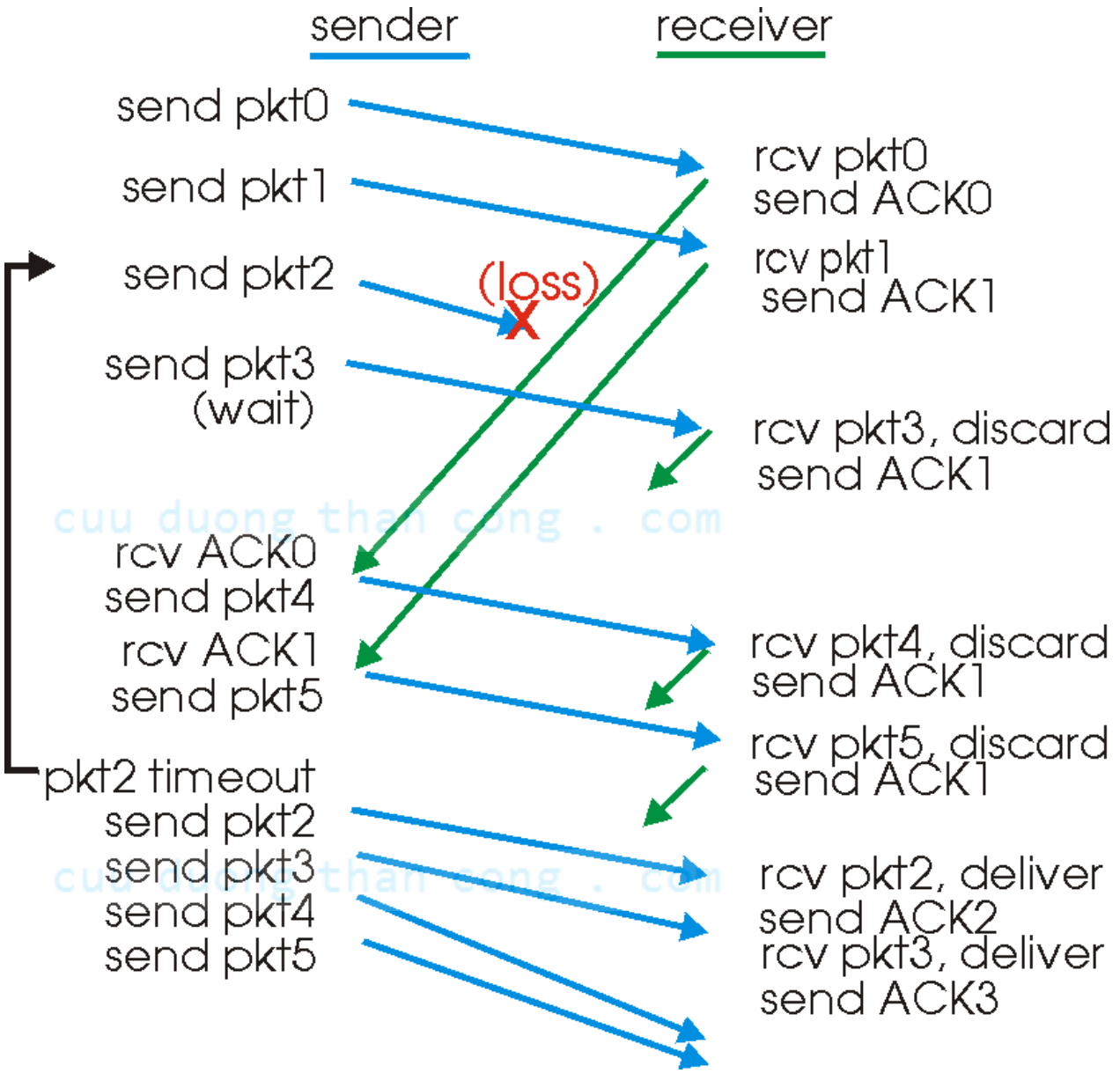
GBN: sender extended FSM



GBN: receiver extended FSM



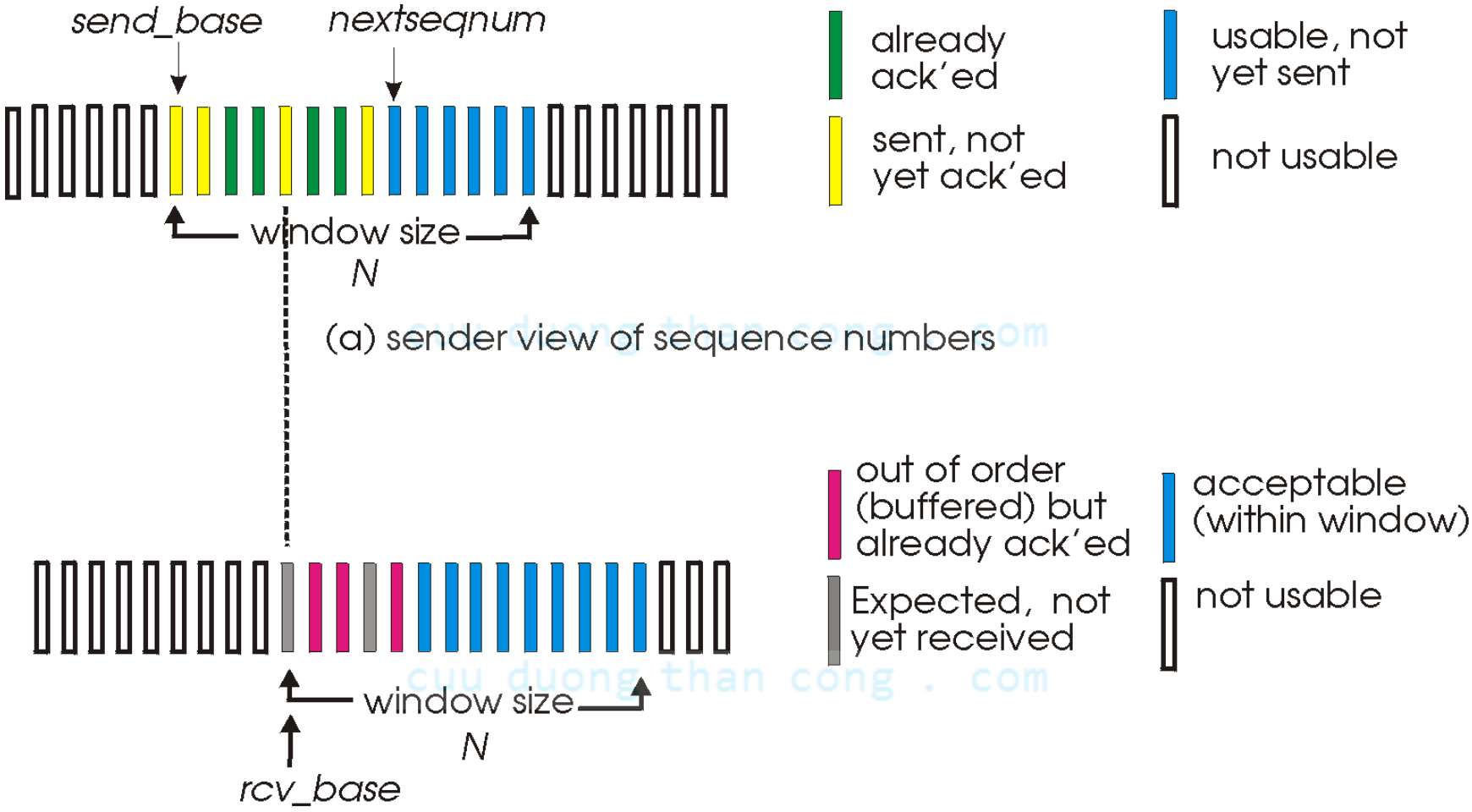
- ACK-only: luôn gửi ACK cho gói tin nhận được ko bị lỗi và có số thứ tự đúng trật tự truyền cao nhất
  - Có thể phát sinh trùng lắp các gói tin ACK
  - Chỉ cần nhớ biến **`expectedseqnum`**
- Với các gói tin đến sai thứ tự:
  - Vứt bỏ (không lưu vào vùng đệm) -> no receiver buffering!
  - Gửi lại gói tin ACK cho gói tin đến đúng thứ tự và có số thứ tự cao nhất.



Selective Repeat

- Bên nhận xác nhận cho từng gói tin mà nó nhận được chính xác.
  - Sử dụng buffer.
- Bên gửi chỉ gửi lại các gói tin mà chưa được xác nhận.
  - Sử dụng bộ đếm giờ cho mỗi gói tin chưa được xác nhận.
- Cửa sổ bên gửi
  - Kích thước N
  - Cũng giới hạn số gói tin đã gửi nhưng đang chờ xác nhận

Selective repeat: sender, receiver windows



Selective repeat

Bên gửi

Khi có data đến từ tầng trên:

- Nếu số thứ tự kế tiếp nằm trong cửa sổ, gửi gói tin.

timeout(n):

- Gửi lại gói tin n, khởi động lại bộ đếm giờ

ACK(n) trong cửa sổ [sendbase,sendbase+N]:

- Đánh dấu gói tin n là đã nhận được
- Nếu n là nhỏ nhất trong số các gói tin chưa được ACK, dịch chuyển cửa sổ sang phải.

Bên nhận

Gói tin n trong cửa sổ [rcvbase, rcvbase+N-1]

- Gửi ACK(n)
- Nếu gói tin đến sai thứ tự: đưa vô buffer
- Nếu gt đến đúng thứ tự: chuyển giao lên tầng trên (và cũng chuyển giao luôn các gói tin nằm trong buffer và đúng thứ tự), dịch chuyển cửa sổ sang phải.

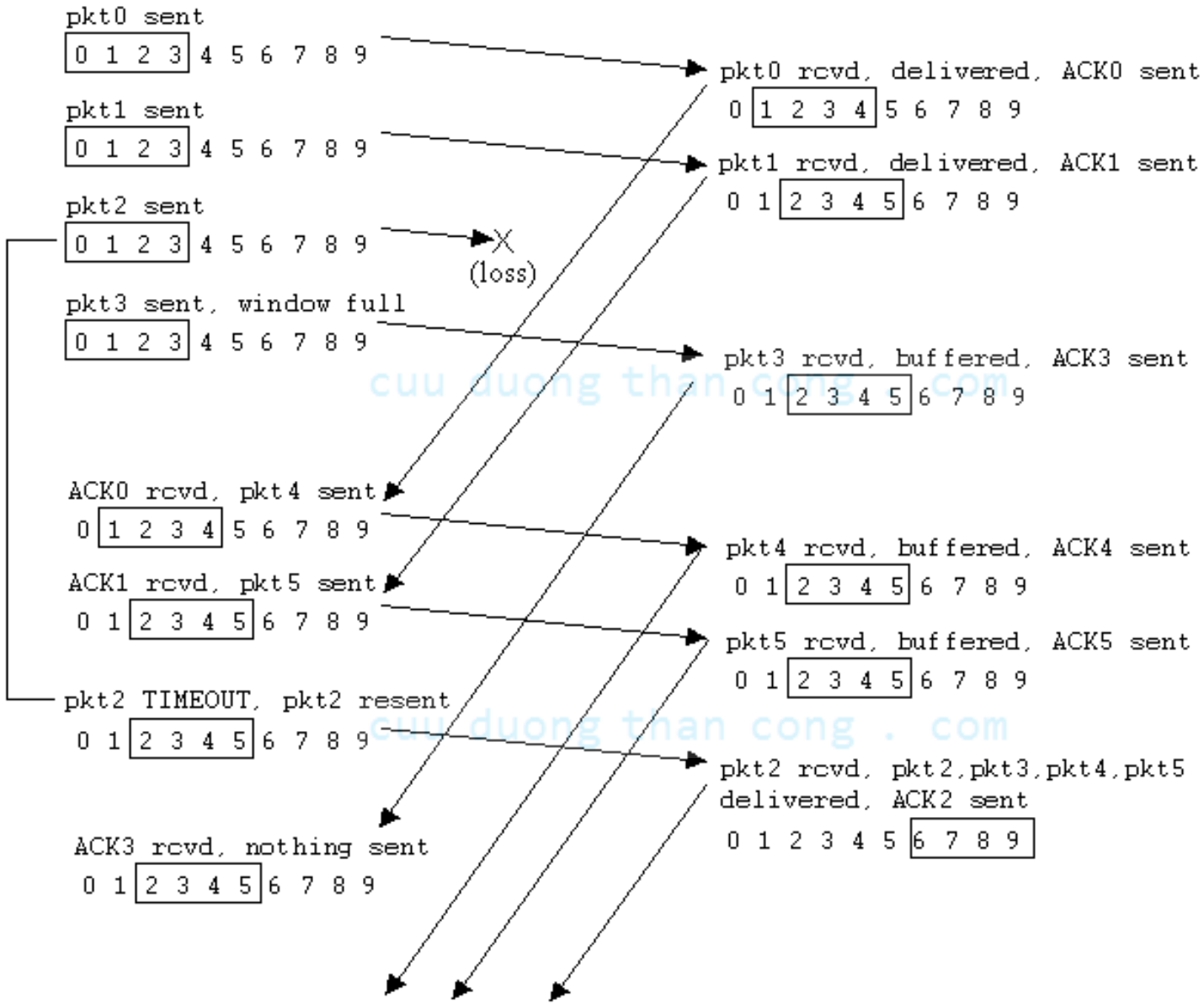
Gói tin n trong [rcvbase-N,rcvbase-1]

- Xác nhận ACK(n)

Trường hợp khác

- Bỏ qua

Selective repeat in action



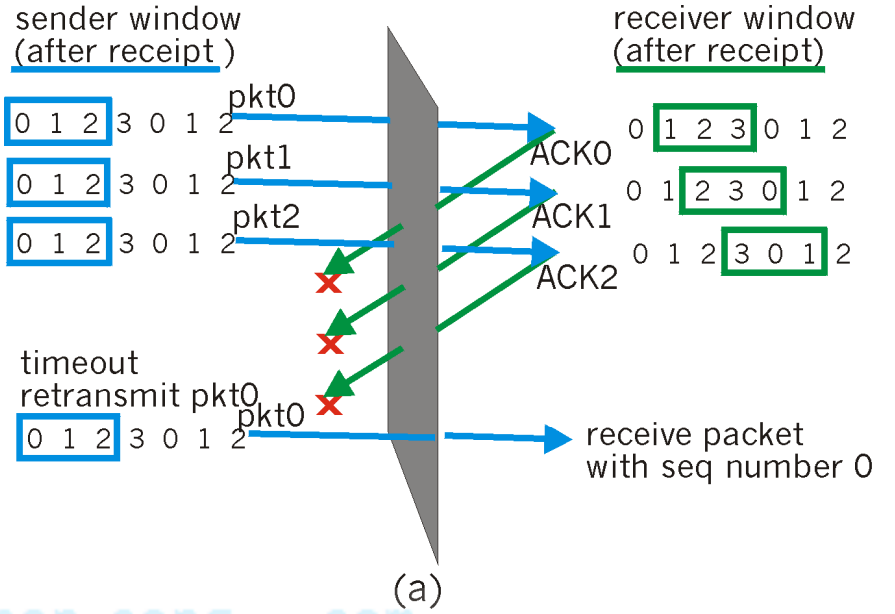
Selective repeat: dilemma

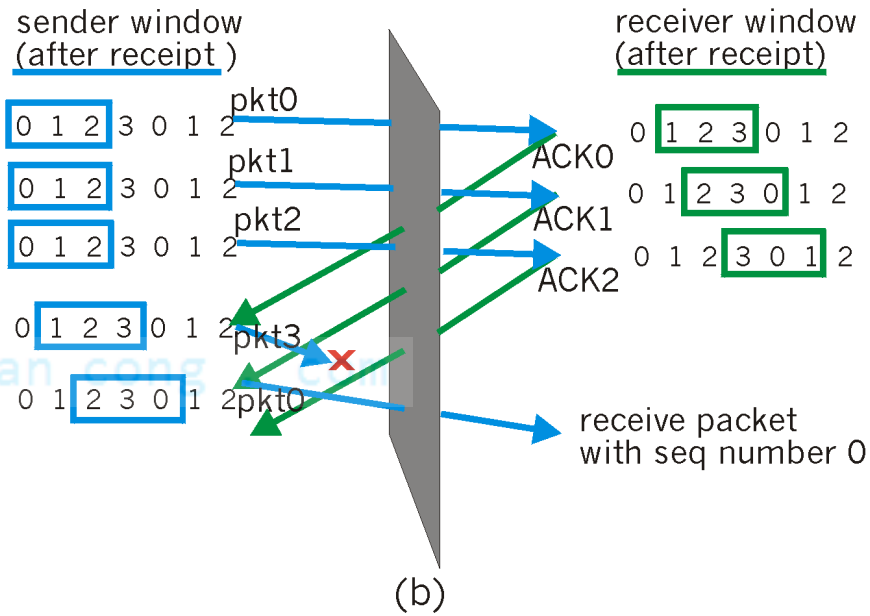
Ví dụ:

- Vùng các stt: 0, 1, 2, 3
- Kích thước cửa sổ=3

- Bên nhận không thấy gì khác giữa 2 kịch bản!

Q: giữa kích thước cửa sổ và kích thước vùng số thứ tự có mối quan hệ gì?





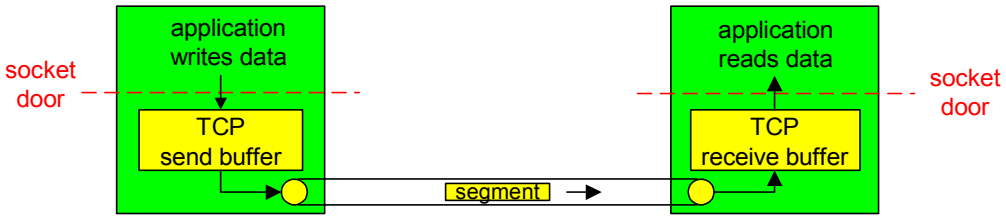
Chapter 3 outline

- 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

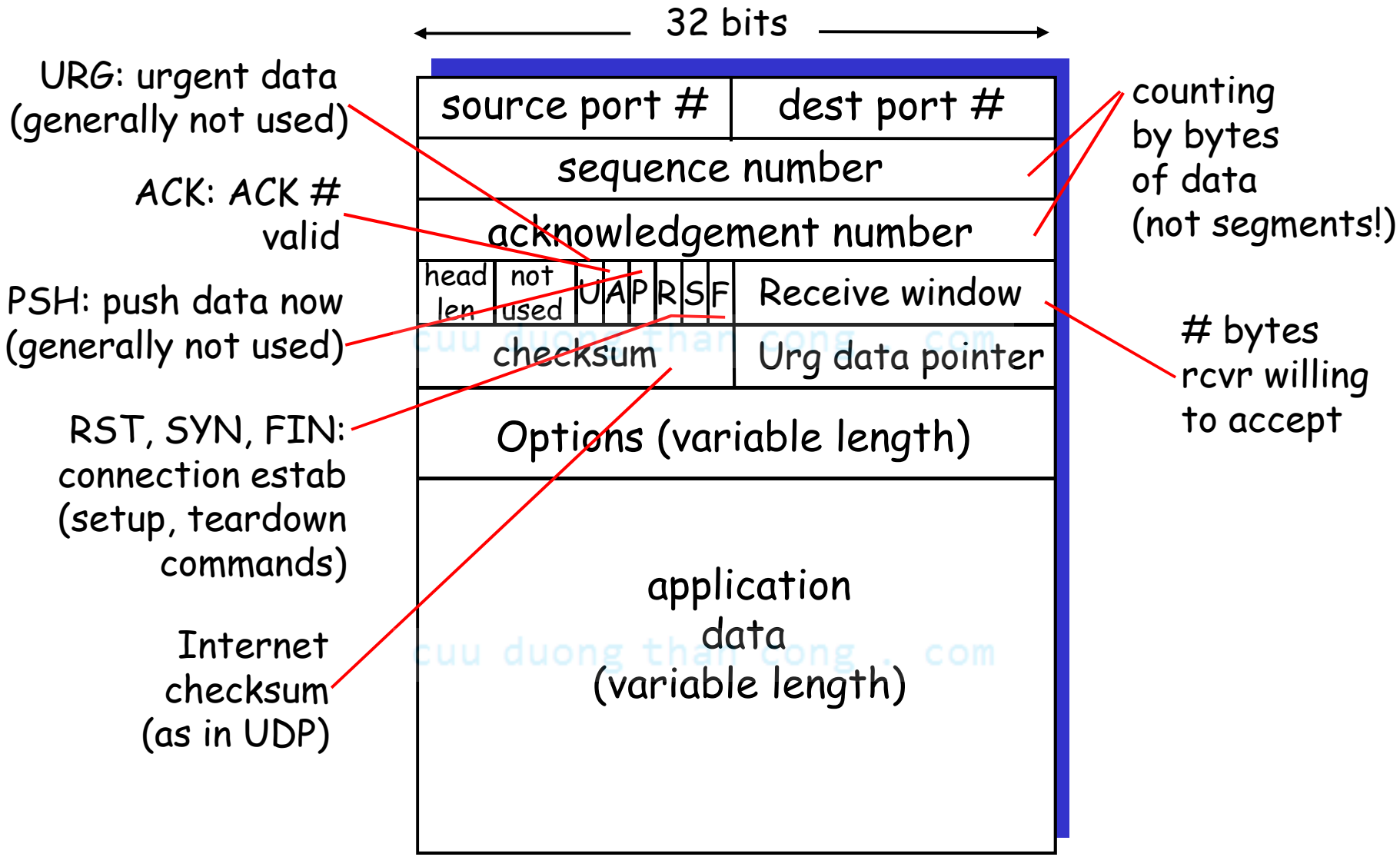
RFCs: 793, 1122, 1323, 2018, 2581

- point-to-point:
  - Một gửi, một nhận
- reliable, in-order *byte stream:*
  - Không thể hiện được ranh giới các gói tin ("message boundaries")
- pipelined:
  - Cơ chế congestion and flow control thiết lập kích thước cửa sổ truyền dữ liệu
- *send & receive buffers*



- full duplex data:
  - Truyền nhận dữ liệu 2 chiều trên cùng kết nối
  - MSS: maximum segment size
- connection-oriented:
  - Nghi thức bắt tay (handshaking) để trao đổi các msg điều khiển (thông số khởi tạo bên gửi tình trạng bên nhận) trước khi truyền
- flow controlled:
  - Bên gửi sẽ ko làm "ngập úng" bên nhận

TCP segment structure



TCP seq. #'s and ACKs

Seq. #'s:

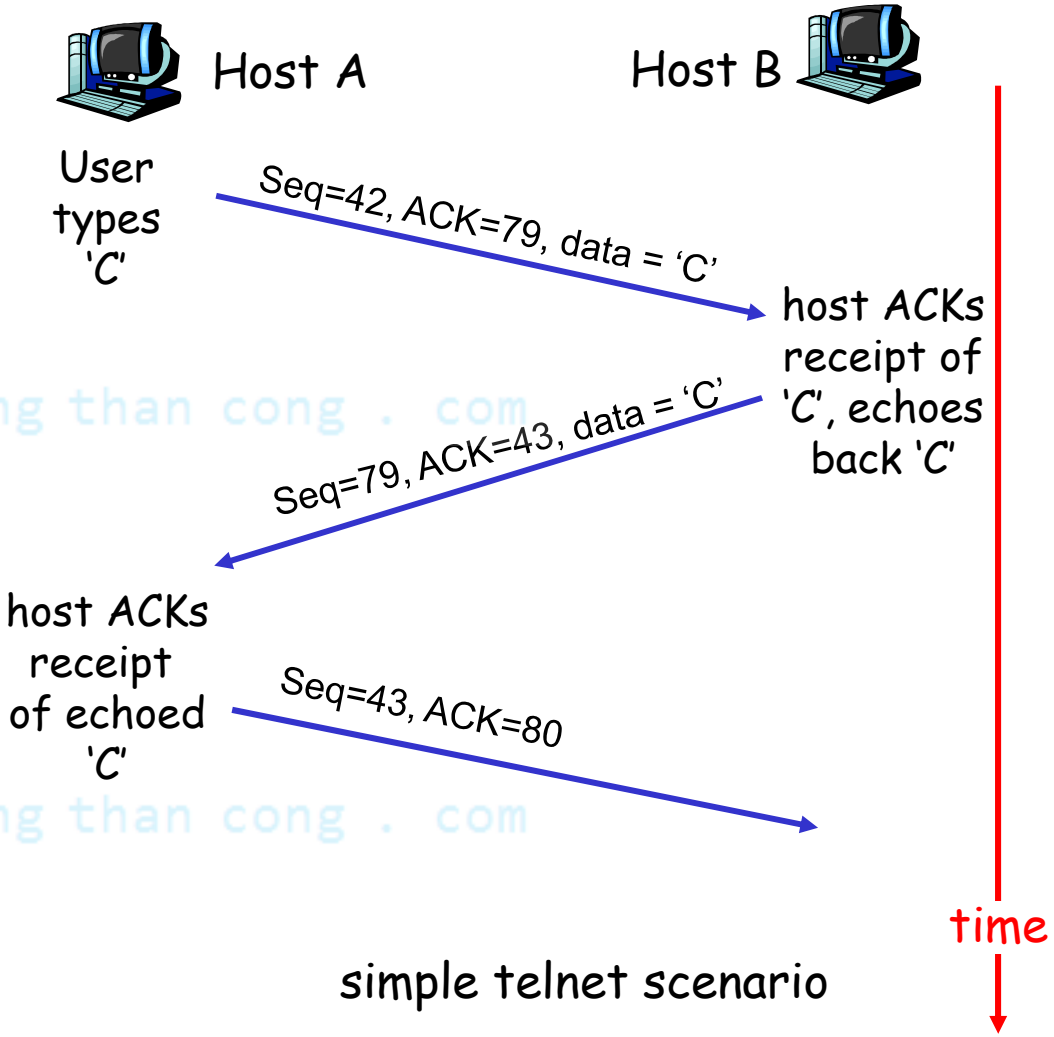
- STTcủa byte dữ liệu đầu tiên nằm trong segment (xét trên chuỗi byte)

ACKs:

- STT của byte kế tiếp được chờ đợi gửi sang từ bên gửi.
- cumulative ACK

Q: Bên nhận xử lý các segments sai thứ tự như thế nào?

- A: Đặc tả TCP ko nói, - tùy thuộc vào bản cài đặt.



simple telnet scenario

TCP Round Trip Time and Timeout

Q: thiết đặt giá trị TCP timeout như thế nào?

- Dài hơn RTT
  - nhưng RTT không cố định
- Quá ngắn: premature timeout
  - Truyền lại một cách ko cần thiết
- Quá dài: phản ứng ko kịp thời khi có segment bị mất

Q: Ước lượng RTT như thế nào?

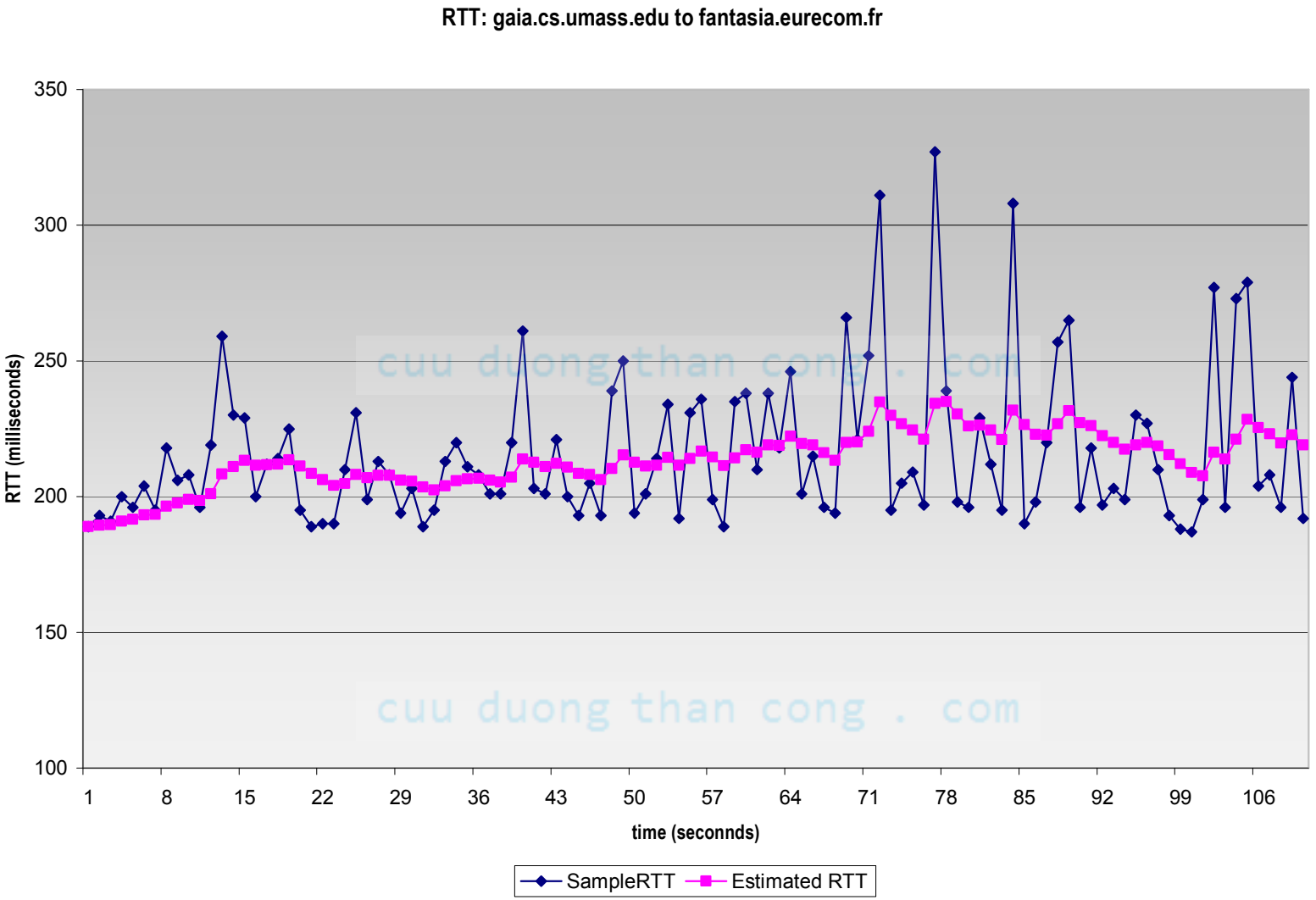
- **SampleRTT:** thời gian đo được từ lúc truyền segment cho đến khi nhận được ACK.
  - Bỏ qua việc truyền lại
- **SampleRTT** sẽ biến động, muốn RTT được ước lượng phải "trơn tru hơn"
  - Tính giá trị trung bình của nhiều lần đo gần đó, chứ không chỉ là giá trị **SampleRTT** hiện hành

TCP Round Trip Time and Timeout

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

- Exponential weighted moving average
- influence of past sample decreases exponentially fast
- Thông thường, α = 0.125

Example RTT estimation:



TCP Round Trip Time and Timeout

Thiết lập giá trị cho timeout

- **Là EstimtedRTT** cộng với "biên độ an toàn"
  - **EstimatedRTT dao động lớn ->** biên độ an toàn phải lớn hơn
- trước tiên, ước lượng giá trị SampleRTT lệch cỡ bao nhiêu so với EstimatedRTT

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

(thông thường, β = 0.25)

sau đó thiết lập giá trị cho timeout theo công thức

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
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

TCP reliable data transfer

- TCP tạo rdt service trên nền tảng các dịch vụ "ko đảm bảo tin cậy" IP
- Segment được truyền đi mà ko chờ ACK của segment trước (pipelined segments)
- cumulative ACKs
- TCP chỉ sử dụng 1 bộ đếm thời gian để quyết định truyền lại hay ko.

- Việc truyền lại được kích hoạt nhờ:
  - Sự kiện "timeout"
  - Trùng lắp ACK
- Khởi đầu, chỉ xem xét trường hợp đơn giản:

Bên gửi:

  - Bỏ qua các ACK trùng lặp
  - Bỏ qua flow control, congestion control

TCP sender events:

Nhận được dữ liệu từ app:

- Tạo ra segment từ dữ liệu của app và gửi xuống tầng IP
- Field "seq #" chứa STT của byte đầu tiên của segment, xét trên chuỗi byte.
- Khởi động timer nếu nó chưa chạy (xem như timer là để tính giờ cho segment "cũ nhất" chưa-được-xác-nhận
- Hết giờ: `TimeOutInterval`

Khi xảy ra timeout:

- Truyền lại segment gây ra sự kiện timeout
- Khởi động lại timer

Nhận được ACK:

- Nếu đúng là để xác nhận cho các segment vốn chưa được xác nhận trước đó
  - Cập nhật lại cửa sổ
  - Khởi động timer nếu hiện thời có ít nhất một segment chưa được xác nhận

```
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)

Ghi chú:

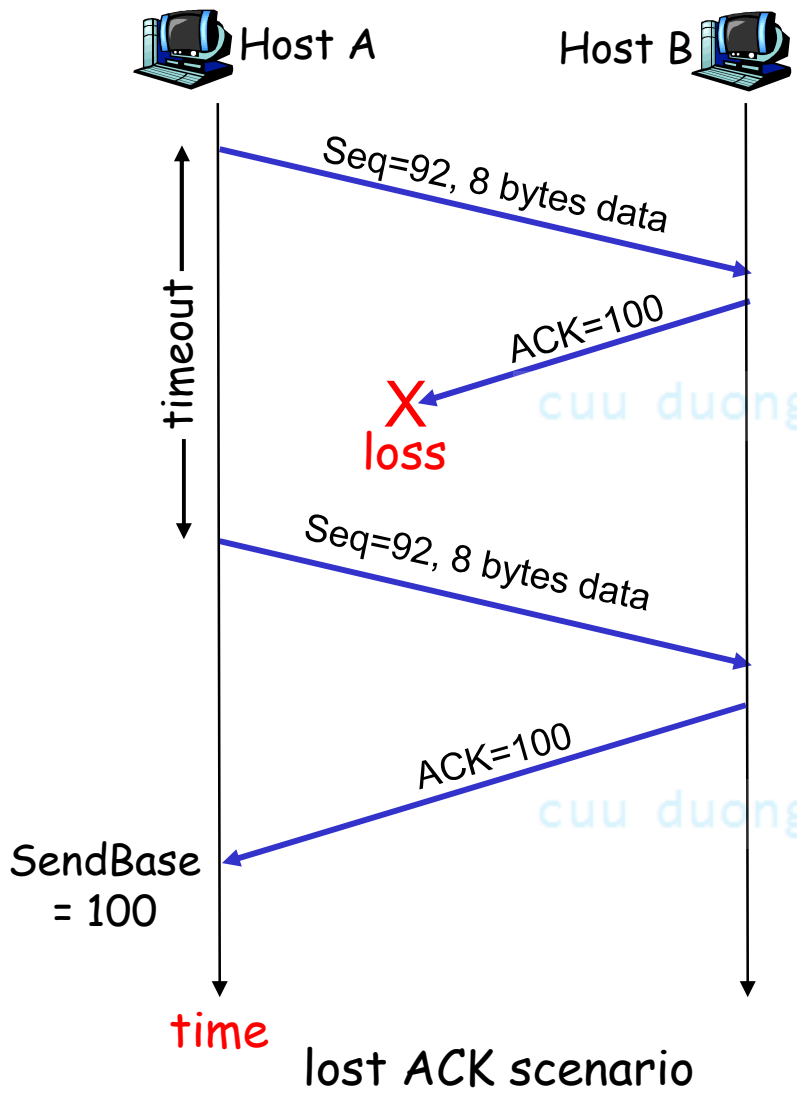
- SendBase-1: last cumulatively ACKed byte

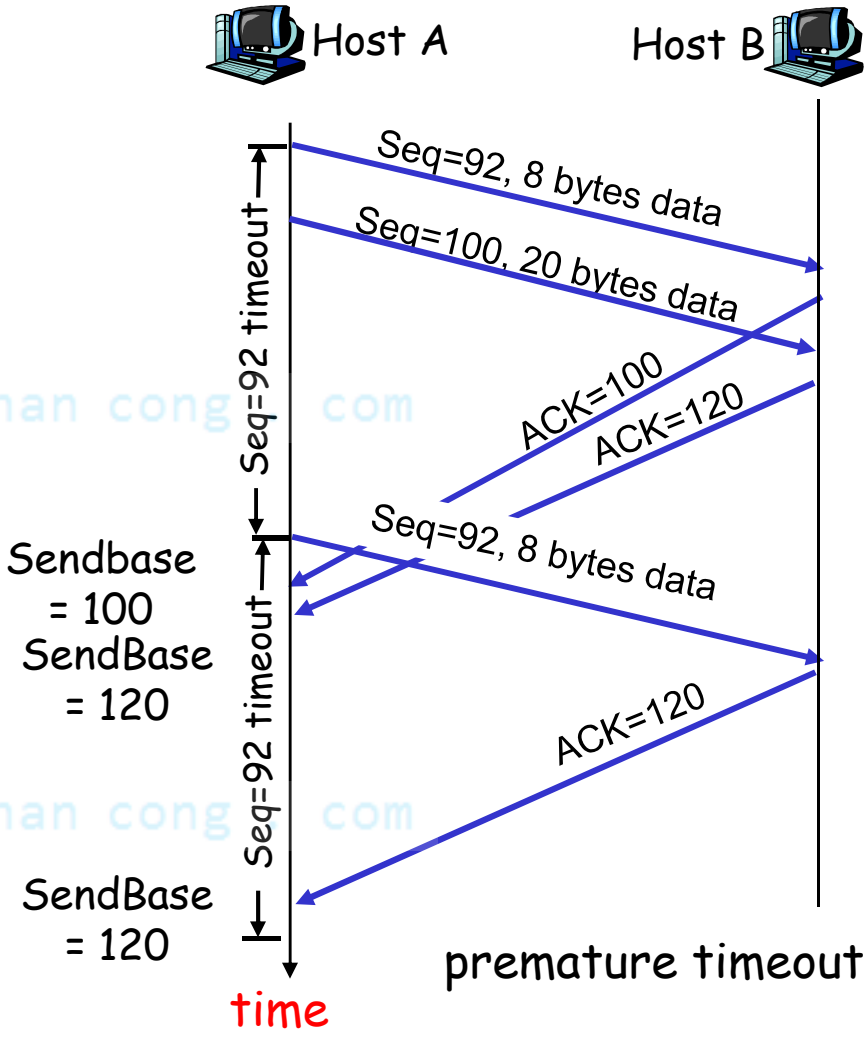
Example:

- SendBase-1 = 71; y= 73, so the rcvr wants 73+ ;

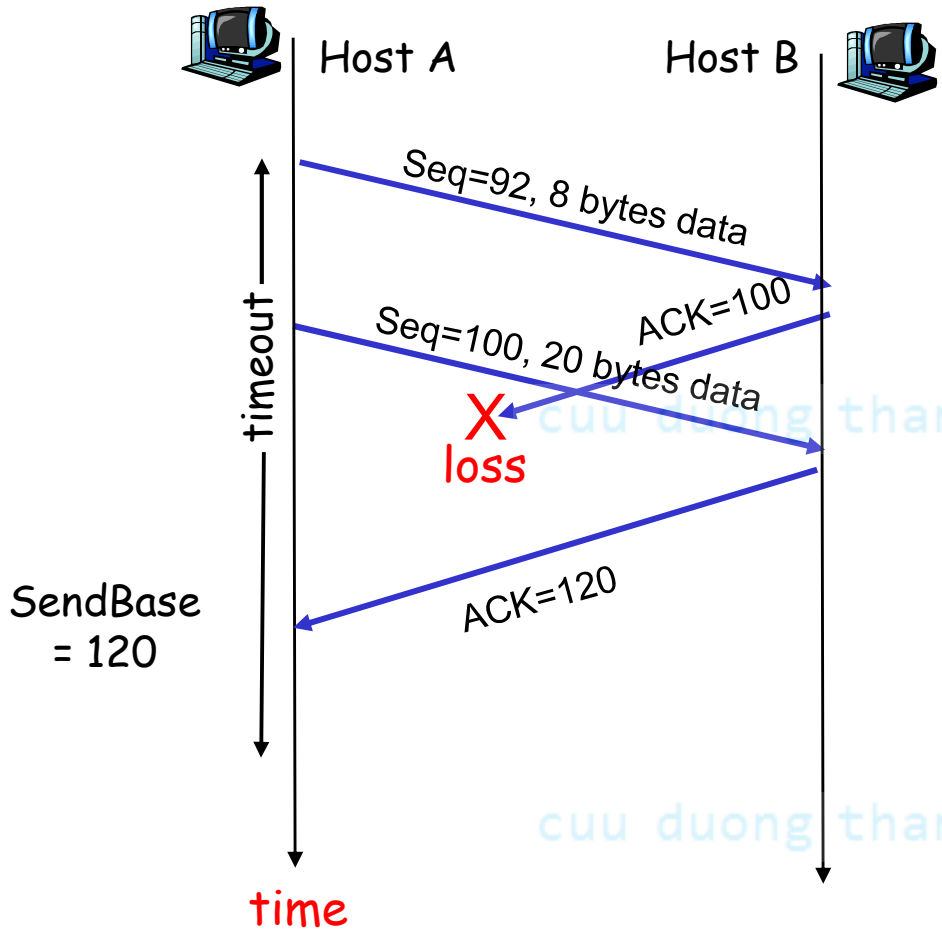
y > SendBase, so that new data is ACKed

TCP: retransmission scenarios





TCP retransmission scenarios (more)



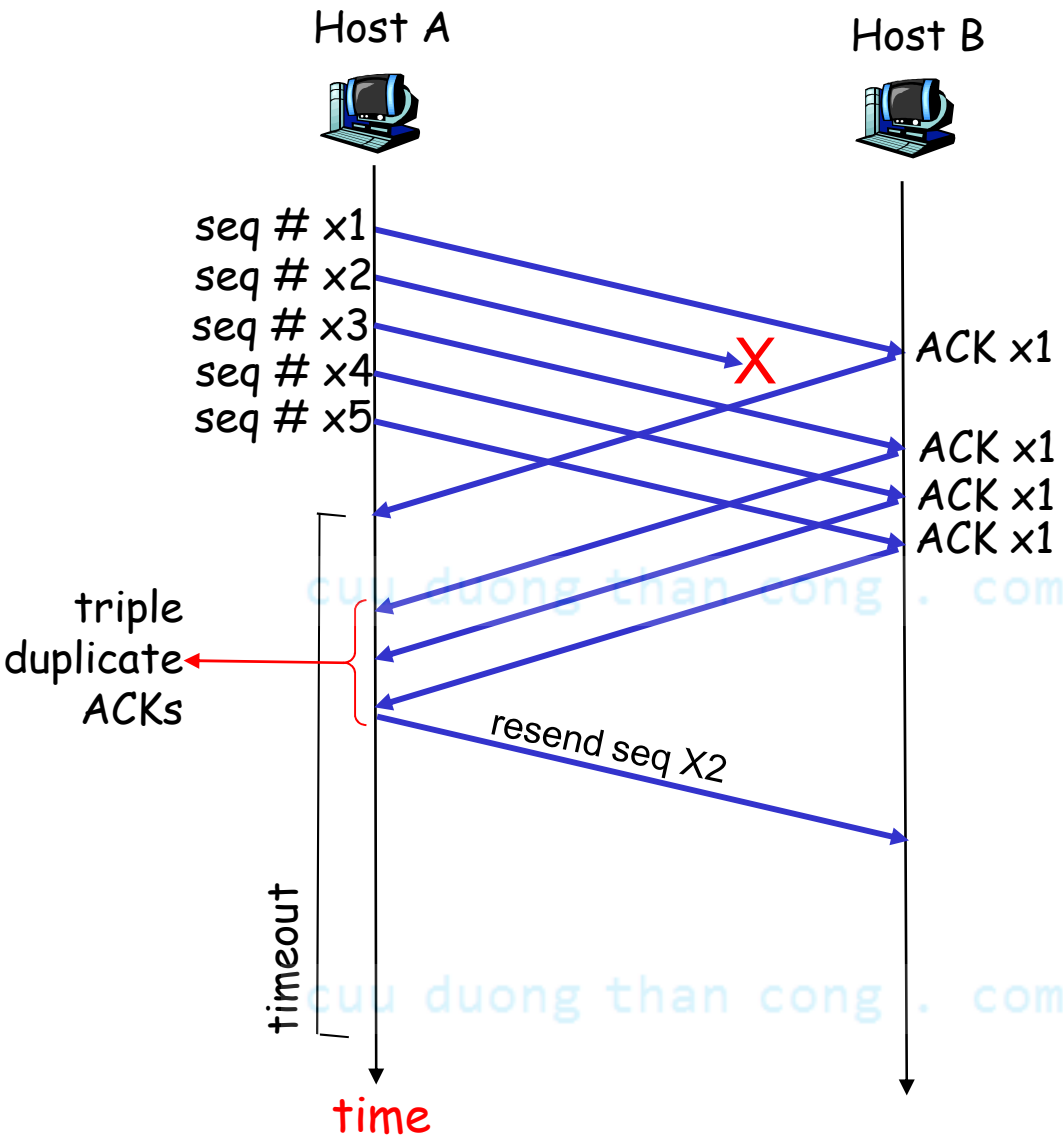
Cumulative ACK scenario

TCP ACK generation [RFC 1122, RFC 2581]

Sự kiện ở bên nhận	Hành động ở bên nhận
Có segment đến với số STT đúng như mong đợi. Tất cả dữ liệu trước segment này đều đã được xác nhận	Trì hoãn gửi ACK. Chờ segment kế tiếp trong 500 ms. Nếu ko có segment nào đến, thì mới gửi ACK.
Có segment đến với số STT đúng như mong đợi. Còn có 1 segment khác đang chờ xác nhận.	Gửi ngay 1 ACK ứng với segment vừa đến (cumulative ACK).
Có segment đến với số STT lớn hơn STT mong đợi. Bên nhận phát hiện trình trạng "không liên tục"	Gửi ngay 1 ACK trong đó nêu STT của byte kế tiếp được chờ đợi (duplicate ACK)
Có segment đến để "bít khoảng hở" (một phần hay toàn bộ)	Gửi ngay 1 ACK trong đó nêu STT của byte kế tiếp được chờ đợi

Fast Retransmit

- Khoảng thời gian time-out thường hơi dài:
  - Chờ lâu (hơn cần thiết) trước khi gửi lại gói tin bị mất
- Phát hiện segment bị mất thông qua việc nhận trùng lặp ACK
  - Bên gửi thường gửi nhiều segment theo loạt
  - Nếu 1 segment bị mất, có khả năng có nhiều ACK trùng lặp dành cho segment ngay trước segment bị mất.
- Nếu bên gửi nhận được 3 ACK cho cùng dữ liệu, nó giả sử rằng "segment đi sau dữ liệu đã được xác nhận" bị mất:
  - fast retransmit: gửi lại segment mà không chờ đến khi time-out.



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

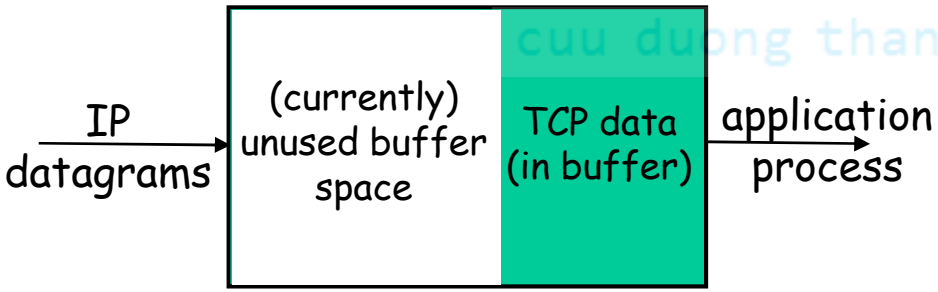
fast retransmit

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
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

TCP Flow Control

- Bên nhận của kết nối TCP, có một vùng đệm (receive buffer):



- Các process của ứng dụng bên nhận có thể đọc dữ liệu từ vùng đệm chậm hơn tốc độ dữ liệu đến.

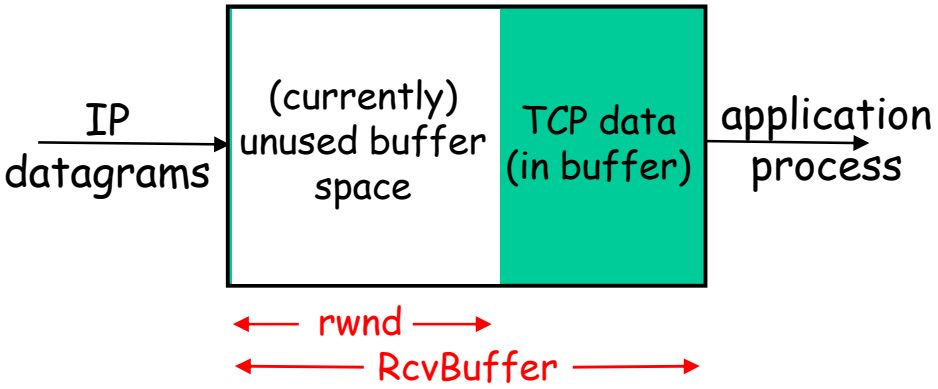
flow control

Bên gửi không gửi quá nhiều, quá nhanh, gây nên tình trạng tràn vùng đệm bên nhận

- *speed-matching service:*

Điều chỉnh tốc độ gửi phù hợp với tốc độ "tiêu thoát" dữ liệu (của ứng dụng) ở bên nhận.

TCP Flow control: how it works



(giả sử bên nhận vứt bỏ các segment đến sai thứ tự)

- Kích thước vùng đệm chưa sử dụng:

= **rwnd**

= **RcvBuffer-[LastByteRcvd - LastByteRead]**

- Bên nhận: báo cho bên gửi biết *kích thước vùng đệm chưa sử dụng* bằng cách đặt giá trị `rwnd hiện hành` vào trong segment header của mọi segment gửi cho bên gửi
- Bên gửi: giới hạn số bytes chưa-được-xác-nhận nhỏ hơn `rwnd`
  - Đảm bảo vùng đệm bên nhận không bị tràn

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 Connection Management

Nhớ lại: Trong TCP, bên gửi và bên nhận **thiết lập kết nối** trước khi trao đổi các segments

- Khởi tạo các biến của TCP:
  - seq. #s
  - buffers, flow control info (e.g. **`RcvWindow`**)
- *client:* bên khởi tạo kết nối đến server

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

- *server:* chờ client liên lạc

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

Thủ tục "Bắt-tay-3-bước" (three-way handshake)

Bước 1: client gửi TCP segment SYN đến server
- Chỉ định số thứ tự (seq.#) cho SYN segment
- Không có dữ liệu

Bước 2: server nhận segment SYN, và trả lời bằng segment SYNACK
- server cấp phát vùng buffer
- Chỉ định số thứ tự (seq.#) cho SYNACK segment

Bước 3: client nhận SYNACK, trả lời bằng segment ACK, trong segment này có thể chứa dữ liệu

TCP Connection Management (cont.)

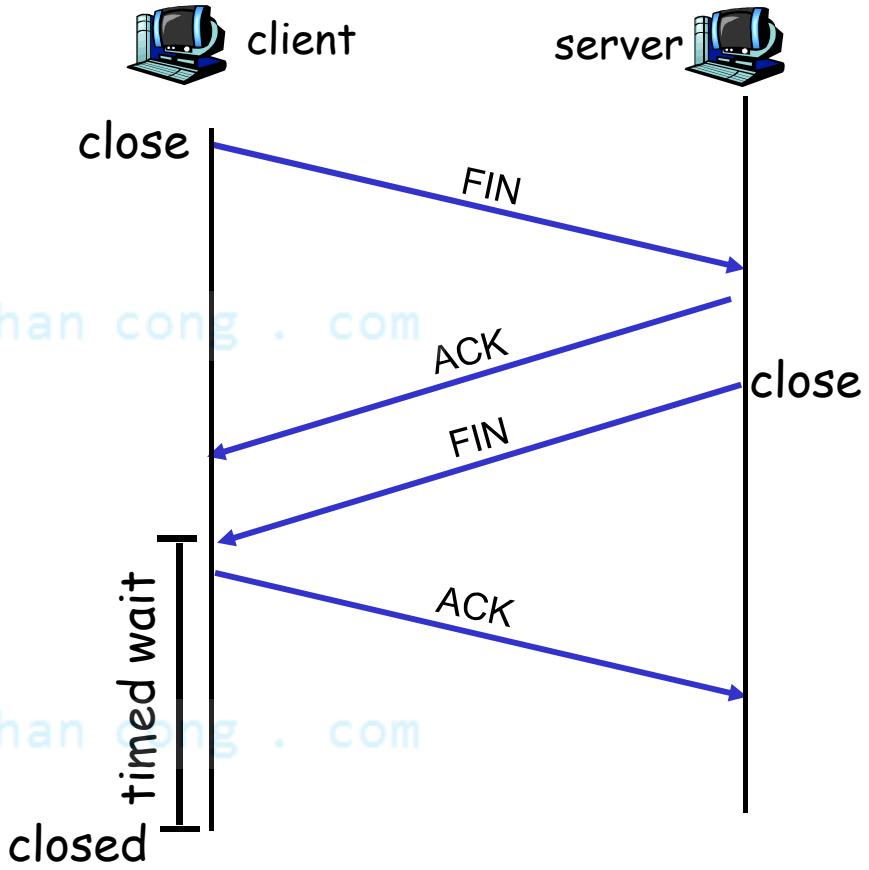
Thủ tục "đóng kết nối"

client đóng socket:

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

Bước 1: client gửi segment FIN đến server

Bước 2: server nhận FIN, trả lời bằng ACK. Sau đó server báo hiệu đóng kết nối từ phía mình bằng cách gửi FIN.



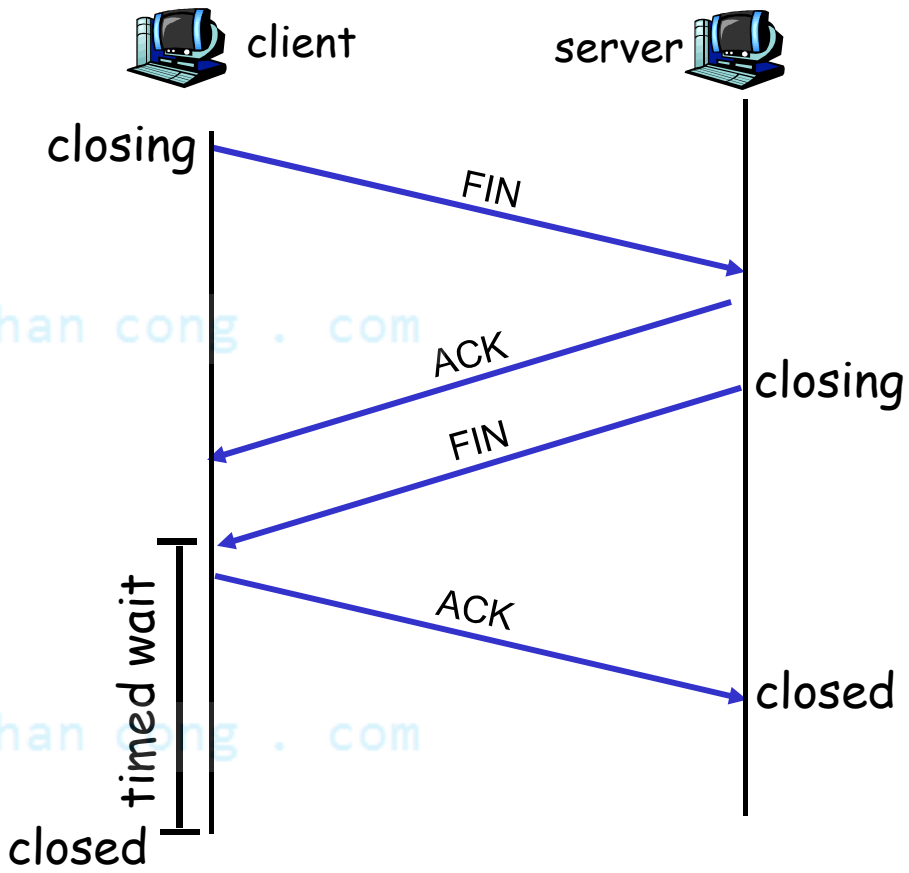
TCP Connection Management (cont.)

Bước 3: client nhận FIN, trả lời bằng ACK.

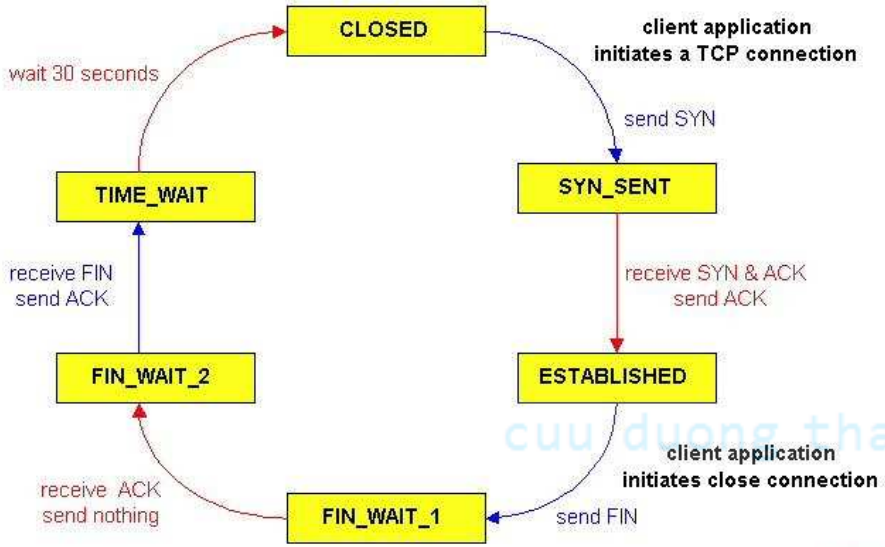
- Vào trạng thái "timed wait": chờ thêm 1 thời gian (VD 30s) trước khi "đóng" . Trong thời gian này có thể gửi lại ACK nếu ACK đã gửi bị mất.

Bước 4: server nhận ACK. Kết nối được đóng.

Lưu ý: với một thay đổi nhỏ, server sẽ xử lý được tình huống có nhiều FIN đến đồng thời

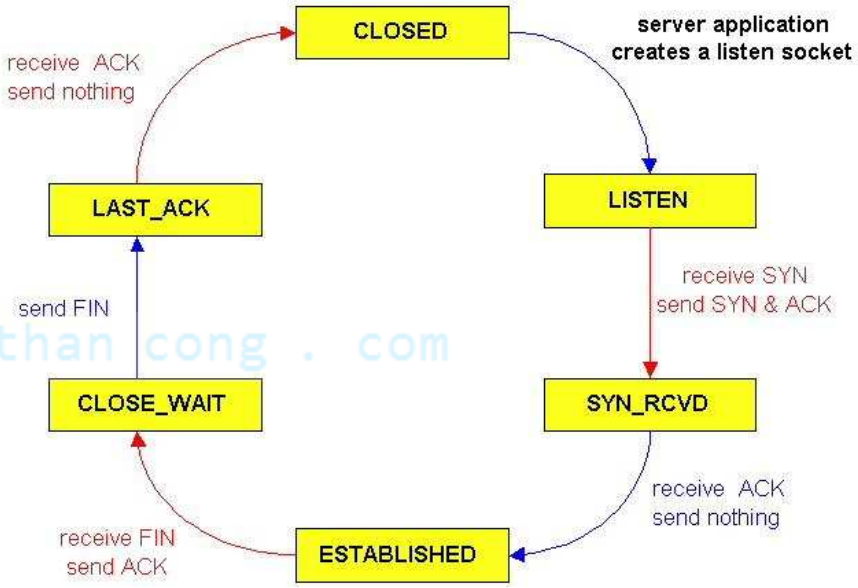


TCP Connection Management (cont)



TCP client lifecycle

TCP server lifecycle



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

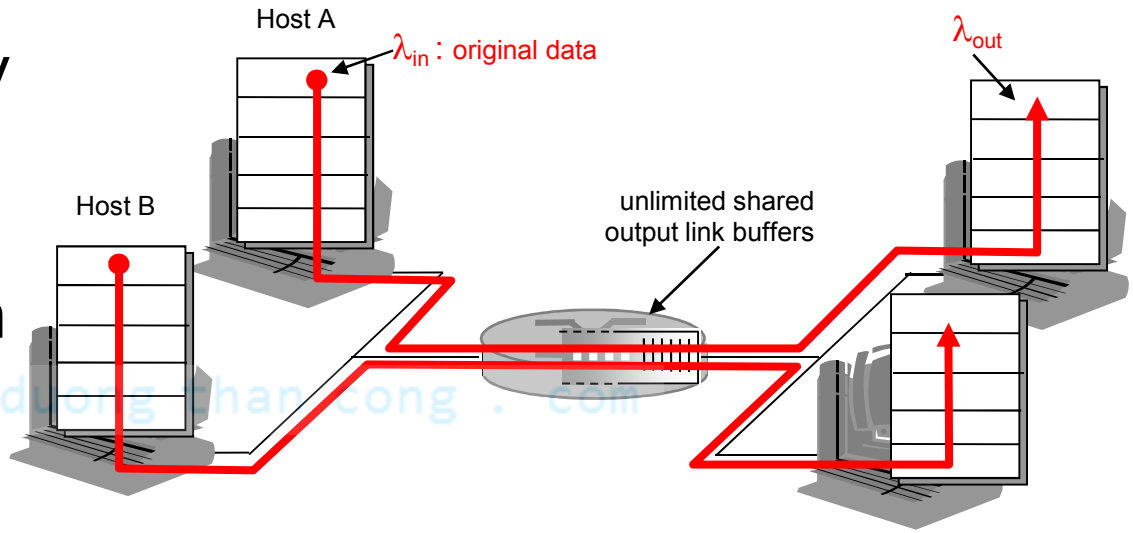
Principles of Congestion Control

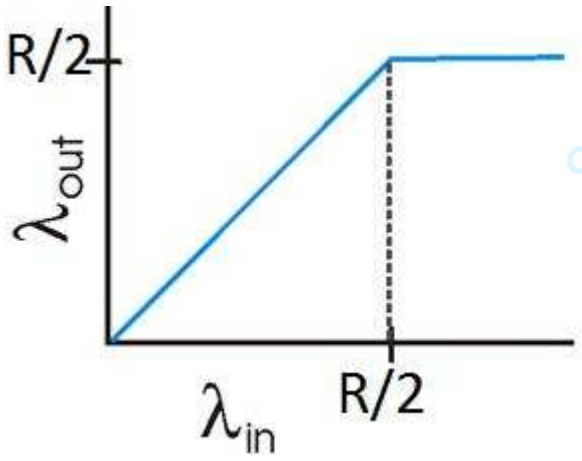
Congestion:

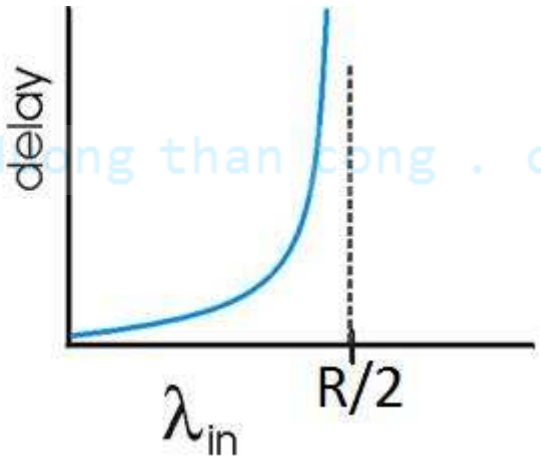
- Nôm na: "Có quá nhiều máy, gửi quá nhiều dữ liệu, gửi nhanh quá khả năng xử lý của mạng.
- Khác với flow control!
- Biểu hiện:
  - Mất gói tin (tràn vùng đệm của router)
  - Kéo dài thời gian (do xếp hàng chờ tại vùng đệm router)
- Là 1 trong 10 vấn đề được quan tâm nhất!

Causes/costs of congestion: scenario 1

- 02 máy gửi, 02 máy nhận
- Một router với bộ đệm không giới hạn
- Ko truyền lại gói tin bị hỏng



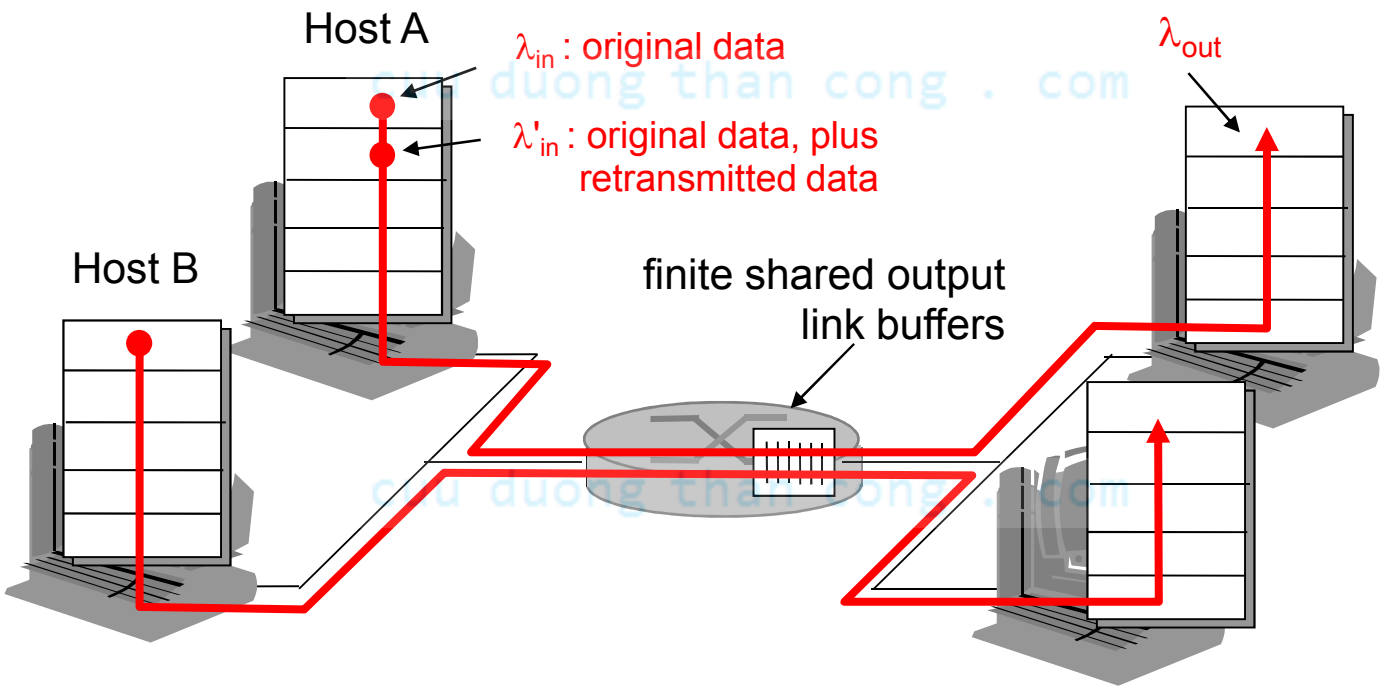




- Khi truyền "hết ga", băng thông có thể được khai thác tối đa nhưng độ trễ rất lớn.
- Đây là cái giá của sự ùn tắt.

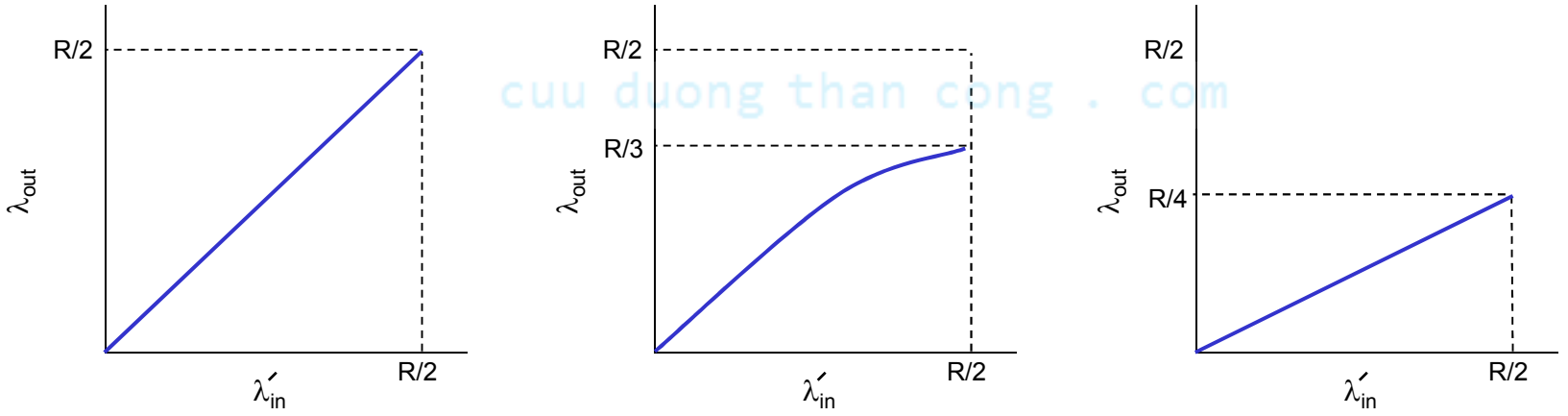
Causes/costs of congestion: scenario 2

- Một router, bộ đệm có kích thước giới hạn
- Bên gửi truyền lại gói tin bị mất



Causes/costs of congestion: scenario 2

- $\lambda_{out}=\lambda'_{in}$: tình huống lý tưởng
- $\lambda_{out}< \lambda'_{in}$: truyền lại hiệu quả
- Việc truyền lại các gói tin bị trễ (nhưng ko phải mất) làm cho λ'_{in} trở nên lớn hơn so với trường hợp truyền lại hiệu quả, với cùng λ_{out}



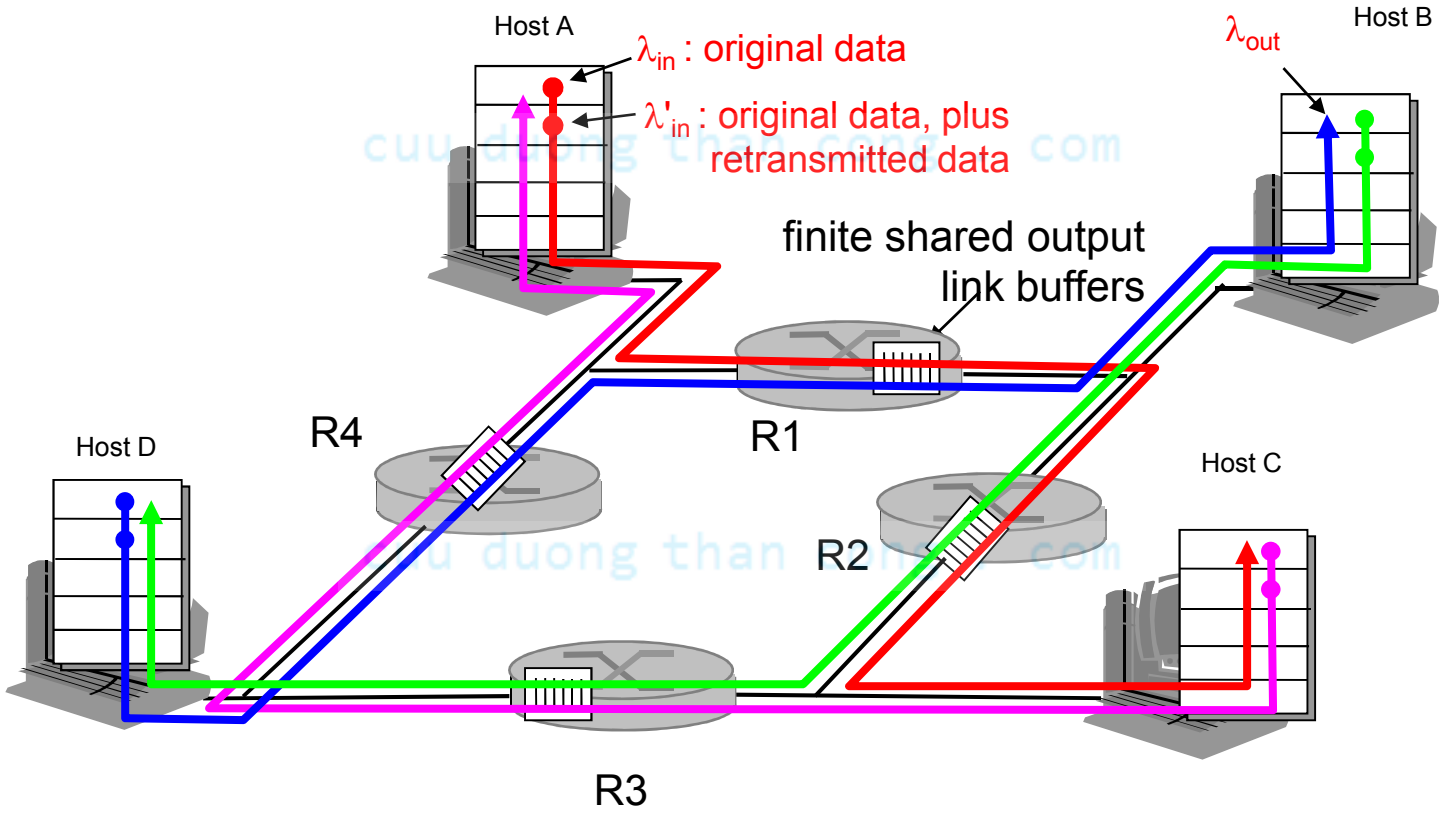
"cái giá" của sự ùn tắt:

- Truyền lại dữ liệu bị mất do tràn bộ đệm
- Truyền lại **ko cần thiết** các gói tin mà bên gửi "tưởng bị mất"

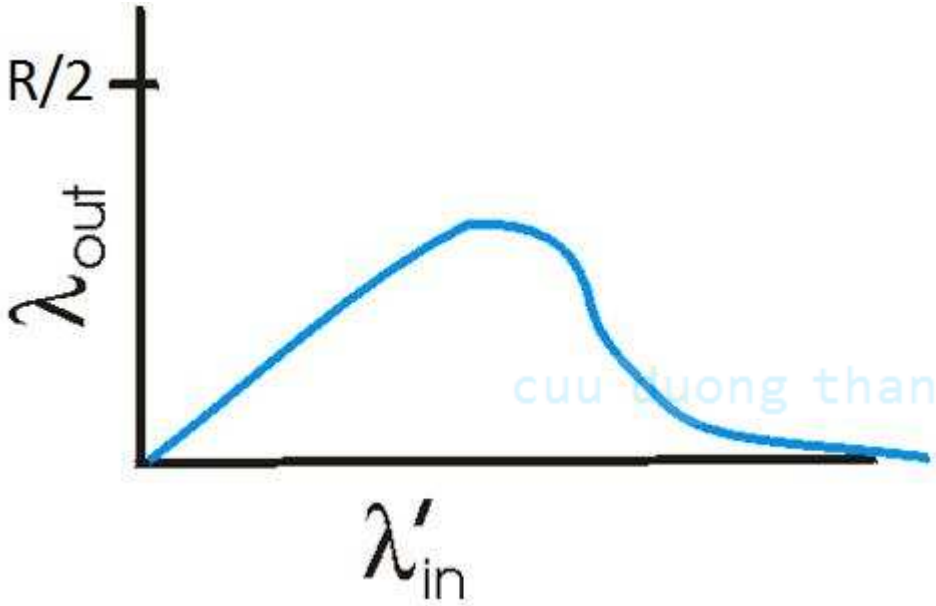
Causes/costs of congestion: scenario 3

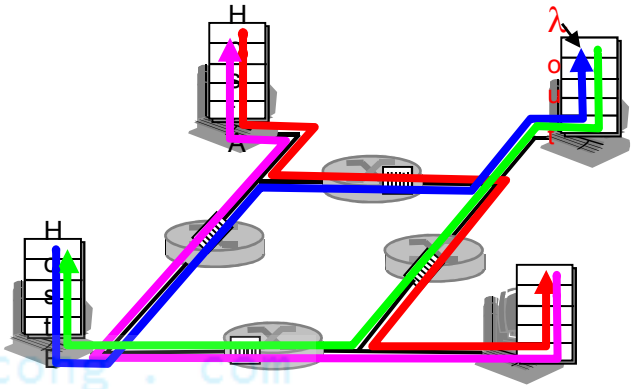
- 04 host gửi dữ liệu
- Tuyến truyền băng qua nhiều router
- Sử dụng cơ chế timeout/retransmit

Q: điều gì sẽ xảy ra khi λ_{in} và λ'_{in} gia tăng ?



Causes/costs of congestion: scenario 3





"Cái giá" của sự ùn tắt:

- Khi 1 gói tin bị mất (do tràn vùng đệm của router), bất kỳ công sức truyền gói tin đó ở các router nằm trước nào cũng là vô ích!

Approaches towards congestion control

Có 2 cách tiếp cận để kiểm soát ùn tắt:

end-end congestion control:

- Tầng mạng ko cung cấp thông tin nào về tình trạng ùn tắt.
- Các host suy diễn ra sự ùn tắt do quan sát các dấu hiệu mất gói tin và thời gian bị kéo dài.
- TCP chọn cách tiếp cận này.

network-assisted congestion control:

- Tầng mạng (cụ thể là router) cung cấp thông tin feedback cho các host.
  - Sử dụng 1 bit trong các gói tin để biểu thị trạng thái ùn tắt (VD trong mạng SNA, DECnet, TCP/IP RFC 3168, ATM ABR)
- Thông tin có thể đến sender bằng 1 trong 2 cách: trực tiếp từ router->sender, gián tiếp router->reveiver->sender

Case study: ATM ABR congestion control

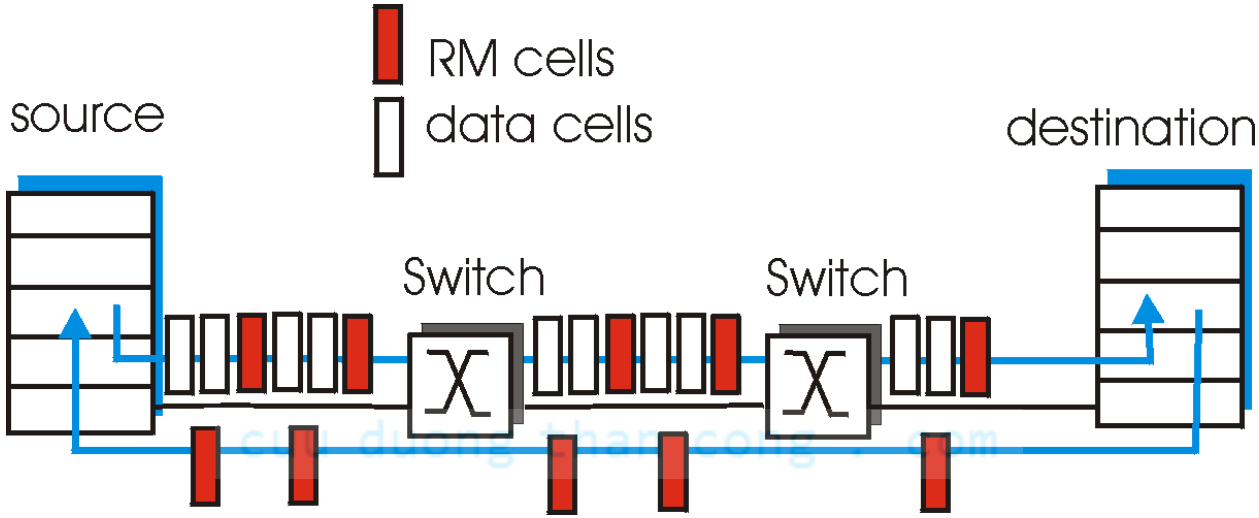
ABR: available bit rate:

- "dịch vụ truyền dữ liệu linh hoạt"
- Nếu đường truyền bên gửi không bị quá tải:
  - Bên gửi nên sử dụng toàn bộ băng thông dự phòng.
- Ngược lại:
  - Bên gửi điều chỉnh mức độ gửi tối thiểu.

RM (resource management) cells:

- Cell~ Gói tin trong mạng ATM
- Bên gửi gửi RM cells, đan xen với các data cells.
- Các switch thiết lập giá trị cho các bit trong gói tin RM (có bàn tay của tầng mạng!)
  - NI bit: no increase in rate (ko ùn tắt)
  - CI bit: congestion indication
- Bên nhận gửi nguyên RM cells lại cho bên gửi. Bên gửi sẽ biết là có ùn tắt hay không.

Case study: ATM ABR congestion control



- two-byte ER (explicit rate) field in RM cell
  - Switch bị ùn tắt, có thể hạ thấp giá trị ER của RM cell khi đi ngang qua switch
  - Theo cách này, ER có thể được đặt thành giá trị nhỏ nhất mà tất cả các switch trên con đường từ nguồn đến đích cho phép.
- EFCI bit in data cells: set to 1 in congested switch
  - Nếu phần lớn các data cell đi trước RM cell có bit EFCI được bật lên 1, bên nhận sẽ bật CI bit của RM cell đó lên 1 trước khi gửi RM Cell về cho bên gửi, báo hiệu có ùn tắt.

Chapter 3 outline

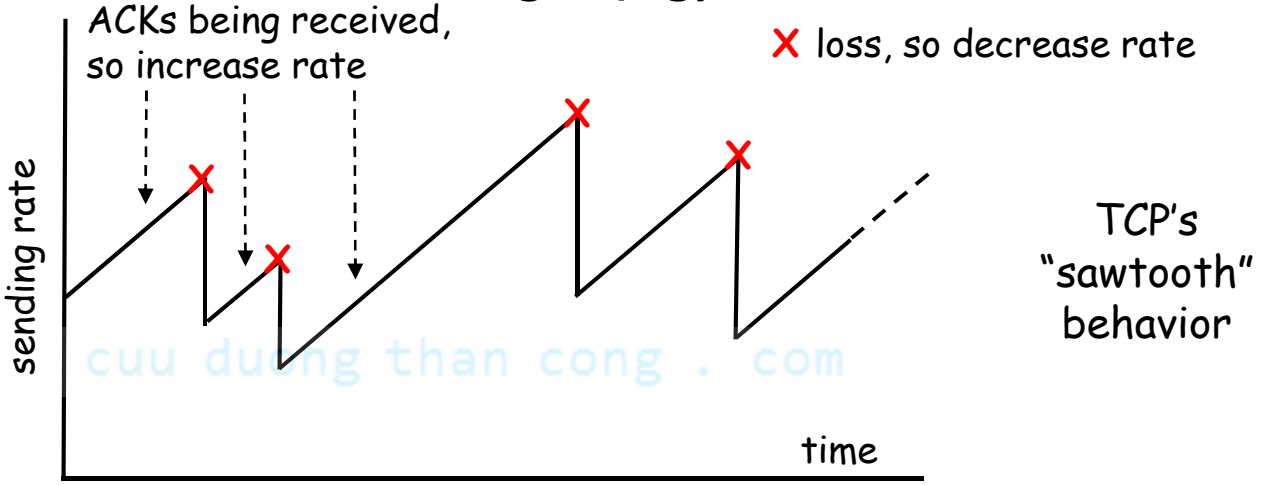
- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

TCP congestion control:

- *Mục tiêu:* Trong TCP, bên gửi cần truyền dữ liệu nhanh nhất có thể, nhưng ko được gây ùn tắt mạng
  - Q: làm thế nào để tìm ra mức truyền vừa dưới mức có thể gây ùn tắt.
- Mỗi bên gửi tự thiết lập tốc độ truyền, dựa trên các dấu hiệu phản hồi:
  - *ACK:* dấu hiệu này cho biết rằng đã được nhận, chứng tỏ mạng không bị ùn tắt. Do đó bên gửi gia tăng tốc độ truyền.
  - Mất segment: bên gửi giả định rằng nguyên nhân mất là do mạng bị ùn tắt, do đó nó giảm tốc độ truyền.

TCP congestion control: bandwidth probing

- Thăm dò băng thông mạng ("probing for bandwidth"): gia tăng tốc độ truyền khi còn nhận được ACK, đến khi có sự kiện mất gói tin xảy ra thì giảm tốc độ truyền.
  - Tiếp tục gia tăng tốc độ truyền khi nhận được ACK và giảm khi thấy mất gói tin (vì bằng thông của mạng có thể thay đổi, phụ thuộc vào các kết nối khác trong mạng)



- Q: how fast to increase/decrease?
  - details to follow

TCP Congestion Control: details

- Bên gửi giới hạn tốc độ truyền bằng việc giới hạn lượng dữ liệu chưa-được-xác-nhận:

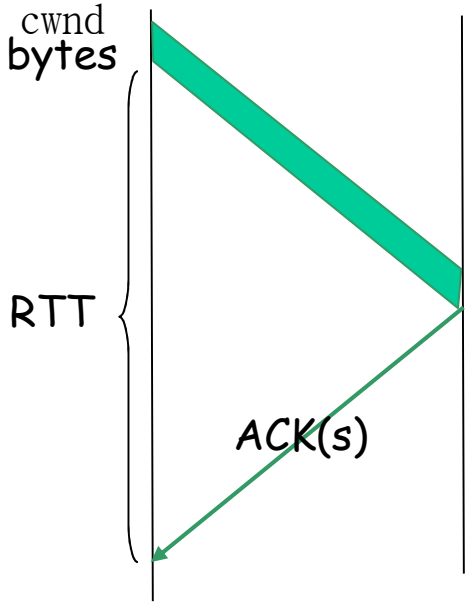
  LastByteSent-LastByteAcked ≤
  min (cwnd, rwnd)

  - **cwnd: congestion window**
  - **rwnd: receive window**

- Một cách gần đúng:

$$\text{Tốc độ truyền} = \frac{\text{cwnd}}{\text{RTT}} \text{ bytes/sec}$$

- **cwnd** là thường xuyên thay đổi, phụ thuộc vào mức độ ùn tắt của mạng mà bên gửi "nhận thức" được.



TCP Congestion Control: more details

Bên gửi giám cwnd khi có sự kiện "mất segment", xảy ra khi:

- Đã timeout mà không nhận được phản hồi
  - Giảm cwnd về 1
- Nhận được 3 ACK trùng nhau
  - Giảm cwnd một nửa

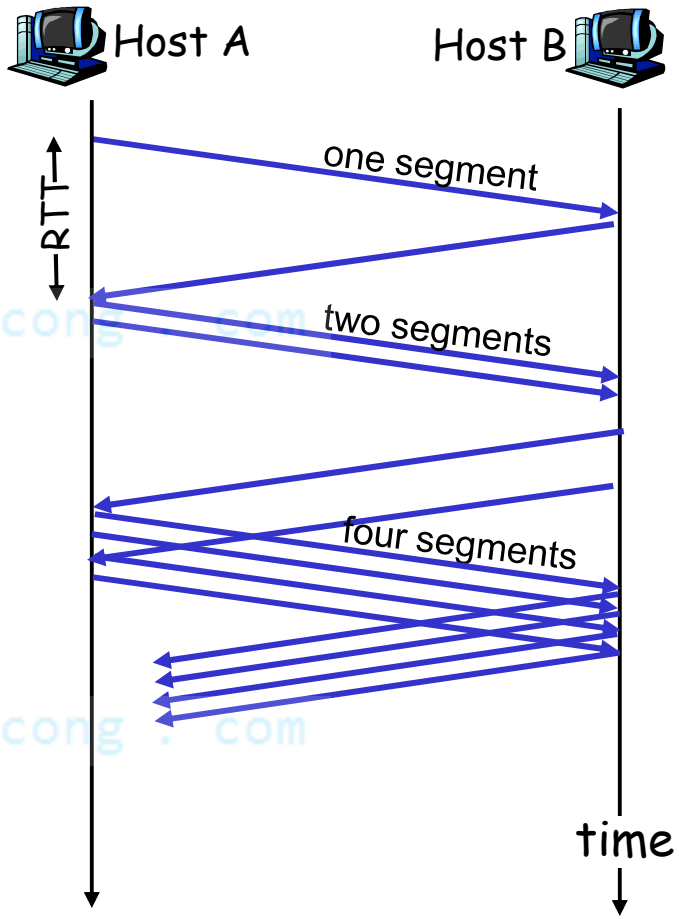
Bên gửi tăng cwnd khi nhận được ACK

- Giai đoạn slow-start
  - Gia tăng theo hàm mũ
- Giai đoạn tránh tắc nghẽn
  - Gia tăng tuyến tính

TCP Slow Start

- Khi kết nối bắt đầu, **`cwnd`** = 1 MSS
  - VD: MSS = 500 bytes & RTT = 200 msec

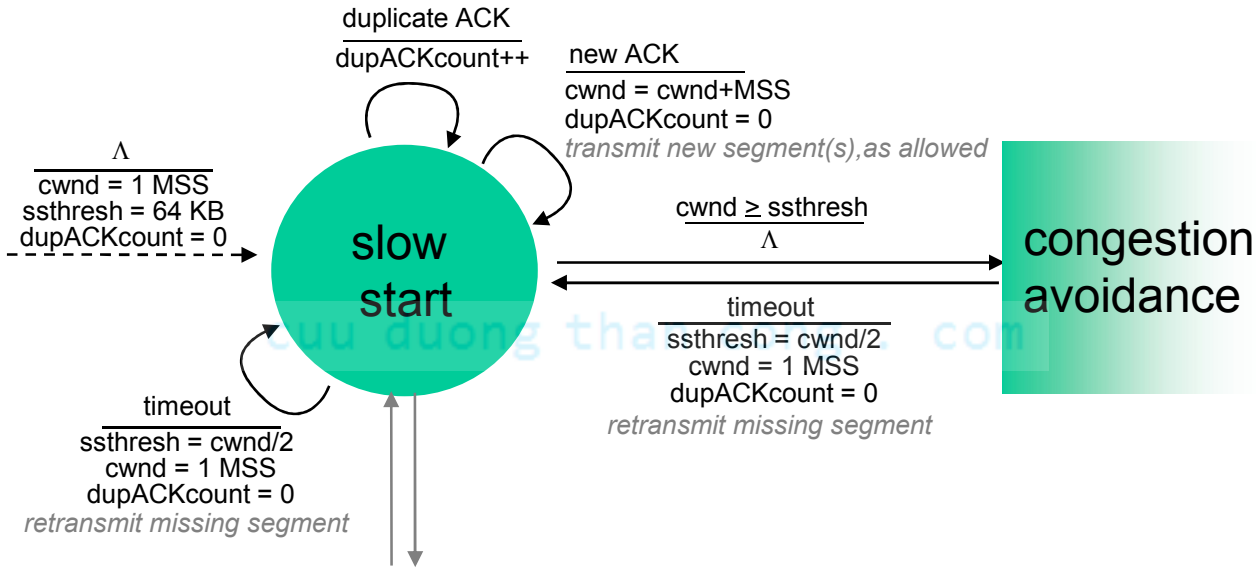
  =>Tốc độ truyền = 20 kbps
- Năng lực của đường truyền có thể lớn hơn MSS/RTT rất nhiều lần
  - Có nhu cầu tăng nhanh đến tốc độ truyền phù hợp.
- Gia tăng tốc độ truyền theo hàm mũ, cho tới khi có sự kiện "mất segment" xảy ra hoặc khi chạm ngưỡng.
  - Gấp đôi **`cwnd`** sau mỗi RTT
  - Làm điều này bằng cách tăng cwnd lên 1 sau mỗi lần nhận được ACK.



Transitioning into/out of slowstart

`ssthresh`: là biến lưu giá trị ngưỡng của **`cwnd`**, do TCP quản lý

- Khi "mất segment": đặt **`ssthresh`** bằng **`cwnd/2`**
  - ghi nhận lại 1 nửa giá trị cwnd vào thời điểm ùn tắt trước đó
- Khi **`cwnd`** >= **`ssthresh`**: chuyển từ slowstart sang giai đoạn tránh ùn tắt.



TCP: congestion avoidance

- Khi **cwnd > ssthresh** gia tăng **cwnd** tuyến tính
  - Tăng **cwnd thêm** 1 MSS sau mỗi RTT
  - Đạt tới điểm ùn tắt khả dĩ chậm hơn slowstart
  - Cài đặt: với mỗi ACK nhận được

cwnd = cwnd + MSS/cwnd

AIMD

- ACKs: increase **cwnd** by 1 MSS per RTT: additive increase
- loss: cut **cwnd** in half (non-timeout-detected loss): multiplicative decrease

AIMD: Additive Increase Multiplicative Decrease

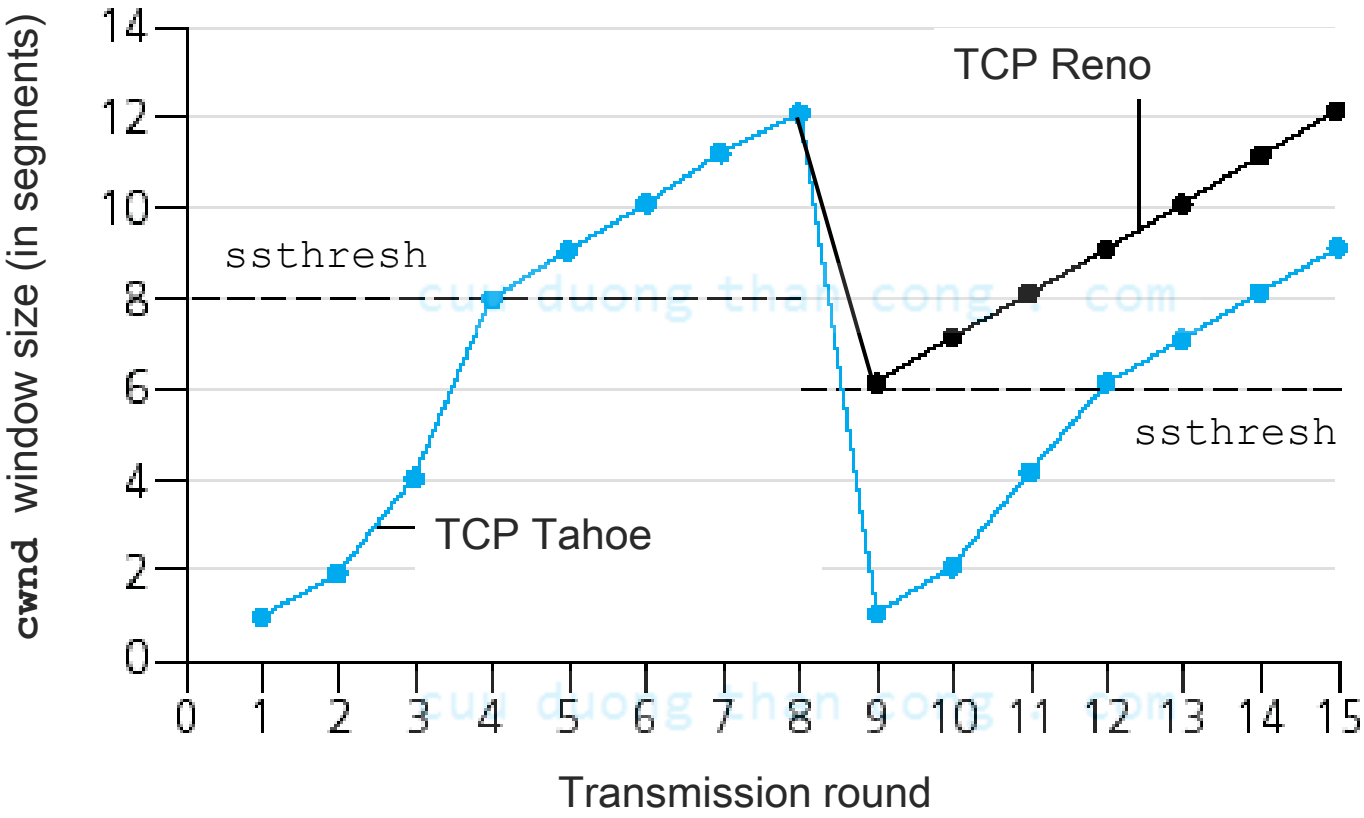
Summary: congestion control algorithm

ssthresh = ? (e,g, 64KB)
cwnd = 1 MSS
/* slow start or exponential increase */
While (No Packet Loss and cwnd < ssthresh) {
    send cwnd TCP segments
    for each ACK, cwnd = cwnd + 1 MSS
}

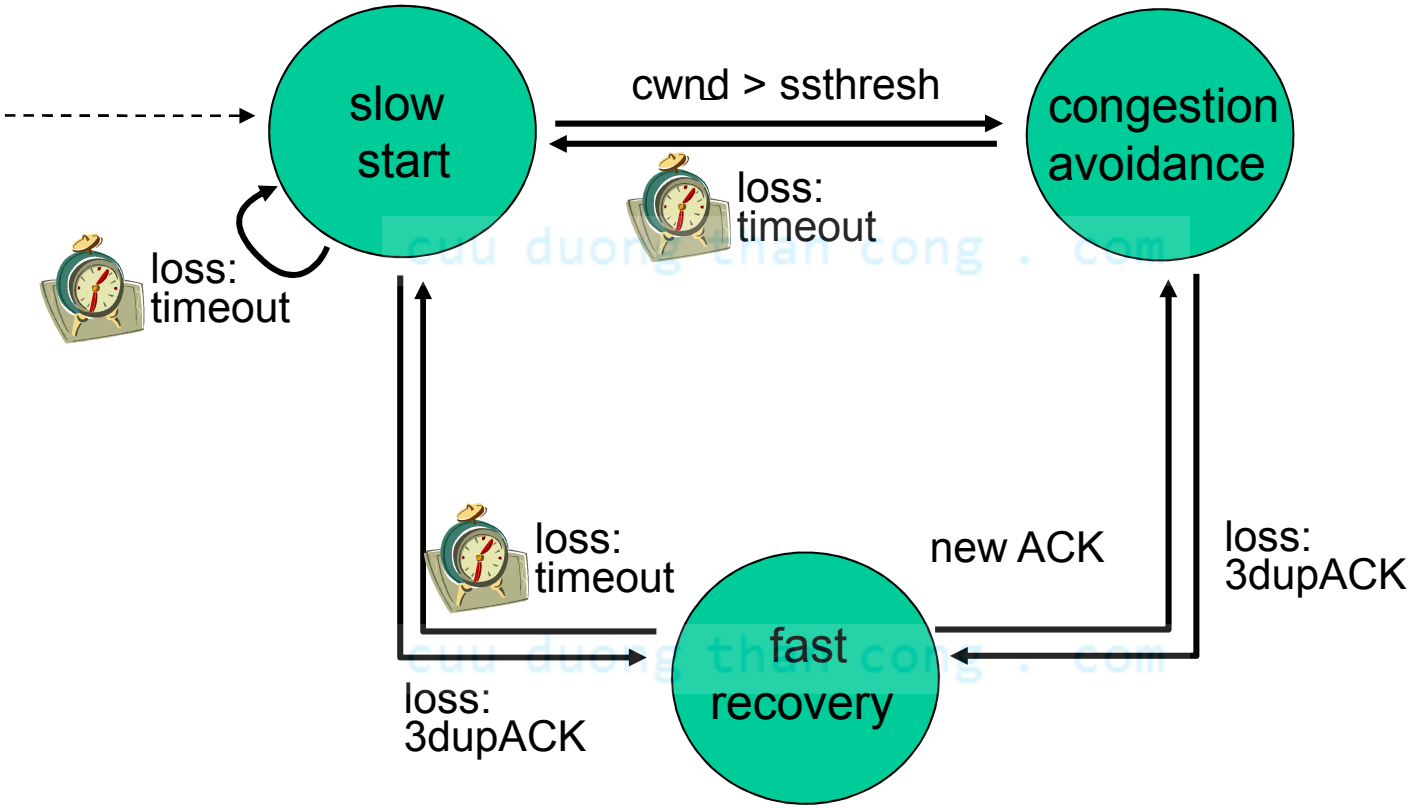
/* congestion avoidance or linear increase */
While (No Packet Loss) {
        send cwnd TCP segments
        /* increase cwnd by 1 MSS when cwnd ACKs are received */
        for each ACK, cwnd = cwnd + MSS/cwnd
}

ssthresh = cwnd/2
If (3 Dup ACKs) cwnd = ssthresh;
Else If (timeout) cwnd=1 MSS;

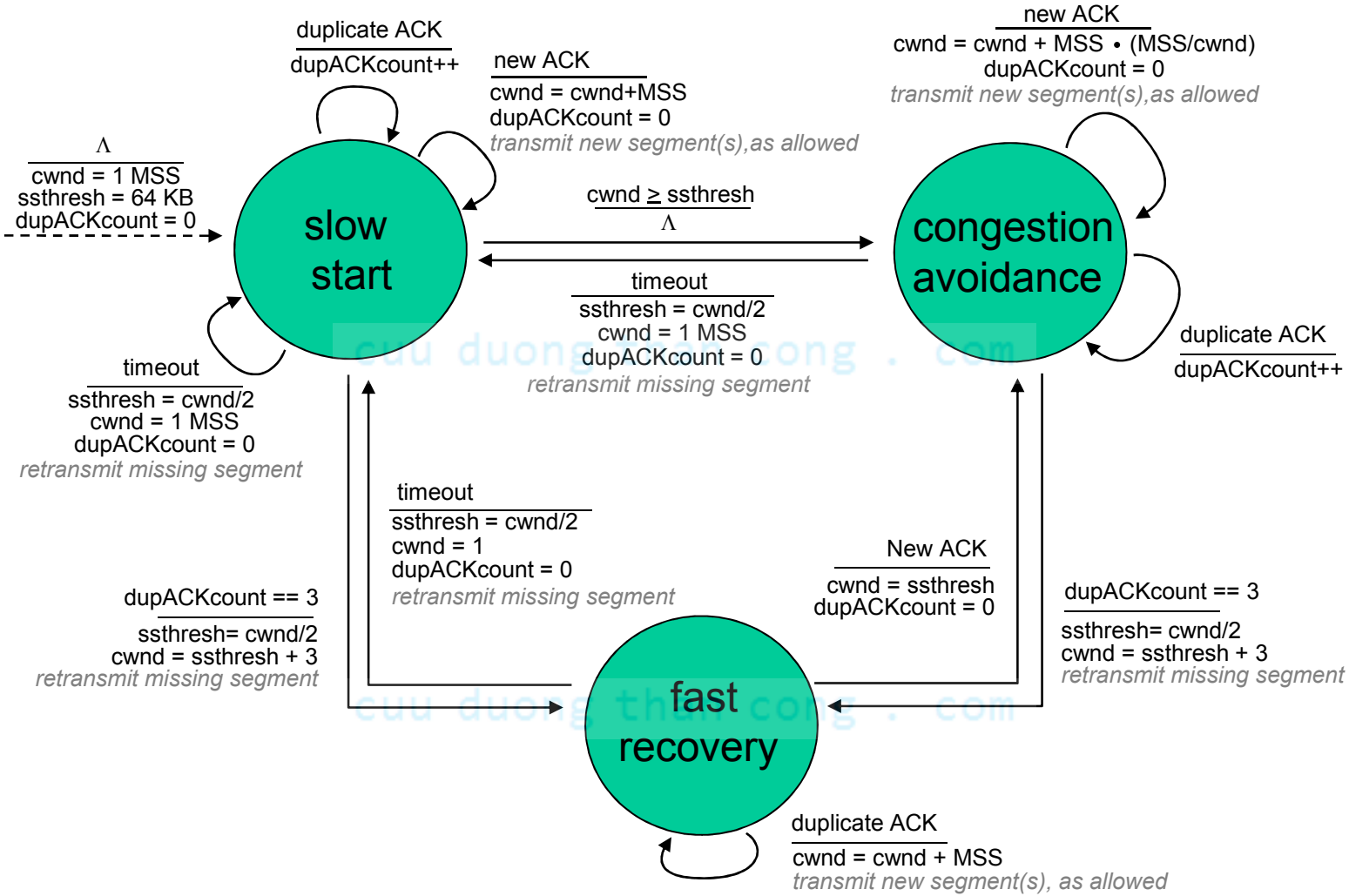
Popular “flavors” of TCP



TCP congestion control FSM: overview



TCP congestion control FSM: details



Summary: TCP Congestion Control

- Khi **`cwnd < ssthresh`**, bên gửi làm việc theo slow-start, cửa sổ cwnd tăng nhanh theo hàm mũ.
- Khi **`cwnd >= ssthresh`**, bên gửi làm việc theo congestion-avoidance, cửa sổ cwnd tăng tuyến tính.
- Khi nhận được 3 ACK trùng nhau,
  `ssthresh := cwnd/2, cwnd := ssthresh`
- Khi timeout xảy ra,
  `ssthresh` := **`cwnd/2, cwnd`** := 1 MSS.

TCP throughput

- *Q:* Thông lượng trung bình của TCP là bao nhiêu, tính theo kích thước cửa sổ và RTT?
  - Bỏ qua giai đoạn slow start
- Gọi W là kích thước của sổ khi có sự kiện "mất segment".
  - Khi kích thước cửa sổ là W, thông lượng là W/RTT
  - Ngay sau khi mất segment, cửa sổ giảm xuống W/2, thông lượng giảm thành W/2RTT.
  - Thông lượng trung bình: 0.75 W/RTT

TCP Futures: TCP over "long, fat pipes"

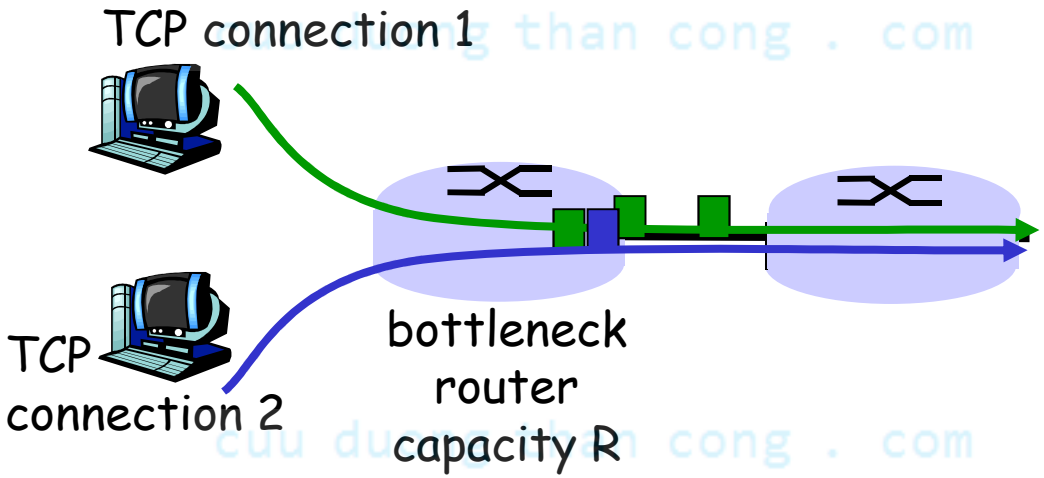
- VD: Xét kết nối TCP với segment kích thước 1500 byte, RTT= 100ms, chúng ta muốn thông lượng 10 Gbps trên kết nối TCP này.
- Cần phải có cửa sổ với kích thước W = 83,333 in-flight segments
- Thông lượng tính theo tỷ lệ mất gói tin (loss rate L):

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

- ➜ L = $2 \cdot 10^{-10}$ *Wow*
- Phiên bản mới của TCP dành cho mạng tốc độ cao.

TCP Fairness

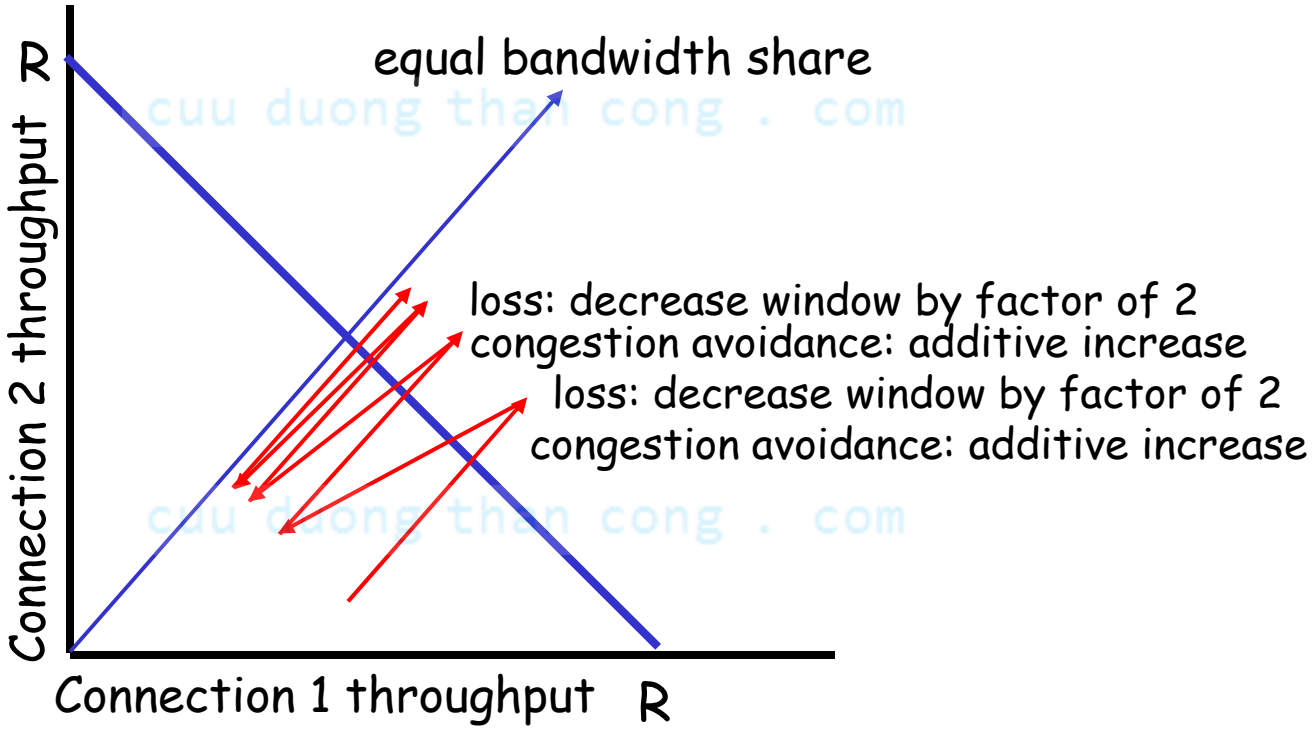
fairness goal: nếu có K phiên truyền TCP chia sẻ cùng một kết nối TCP, mỗi phiên truyền có tốc độ truyền trung bình R/K



Why is TCP fair?

Two competing sessions:

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



Fairness (more)

Fairness and UDP

- Các ứng dụng multimedia thường ko sử dụng TCP
  - Ko muốn tốc độ truyền bị hạn chế do cơ chế congestion control
- Thay bằng UDP:
  - Truyền audio/video ở một tốc độ truyền không đổi, chấp nhận mất gói tin chút đỉnh.

Fairness and parallel TCP connections

- Ko có gì ngăn cản các ứng dụng mở nhiều kết nối song song giữa 2 host.
- web browsers làm điều này
- VD: một đường truyền băng thông R đang phục vụ 9 kết nối;
  - Nếu 1 ứd dùng 1 kết nối TCP, nhận được bằng thông R/10
  - Còn nếu ứd mở cùng lúc 11 kết nối TCPs, nhận được R/2 !

Chapter 3: Summary

- principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- instantiation and implementation in the Internet
  - UDP
  - TCP

Next:

- leaving the network "edge" (application, transport layers)
- into the network "core"