



CHƯƠNG 3

TẦNG VẬN CHUYỂN

NHẬP MÔN MẠNG MÁY TÍNH



A note on the use of these PowerPoint slides:

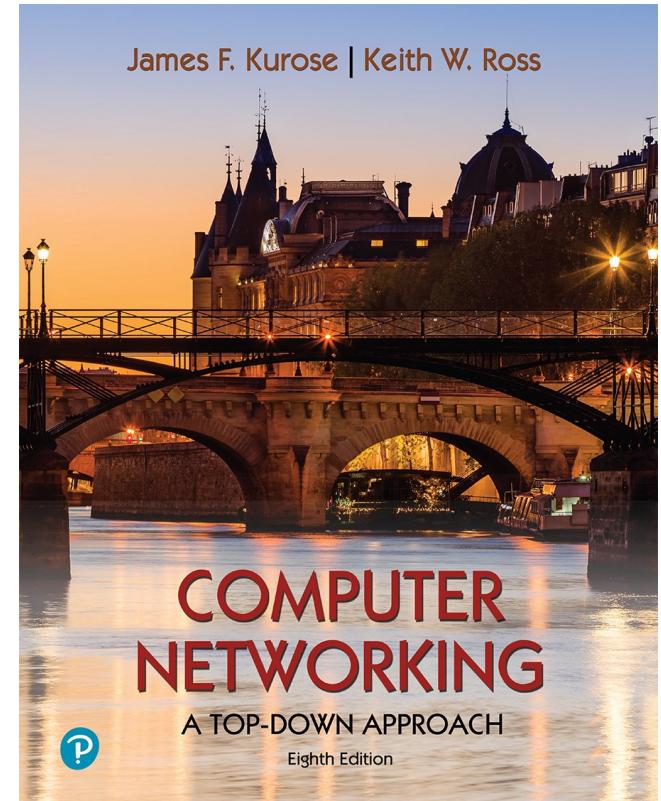
We're making these slides freely available to all (faculty, students, readers). They're in PowerPoint form so you see the animations; and can add, modify, and delete slides (including this one) and slide content to suit your needs. They obviously represent a *lot* of work on our part. In return for use,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

- If you use these slides (e.g., in a class) that you mention their source (after all, we'd like people to use our book!)
- If you post any slides 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.

For a revision history, see the slide note for this page.

Thanks and enjoy! JFK/KWR

All material copyright 1996-2020
J.F Kurose and K.W. Ross, All Rights Reserved



**Computer Networking:
A Top-Down Approach**
8th edition
Jim Kurose, Keith Ross
Pearson, 2020



Tổng quan

Mục đích:

- Hiểu được các nguyên tắc/cơ chế hoạt động:
 - multiplexing, demultiplexing
 - reliable data transfer
 - flow control
 - congestion control

- Hiểu các giao thức:
 - UDP: connectionless transport
 - TCP: connection-oriented reliable transport
 - TCP congestion control



Nội dung

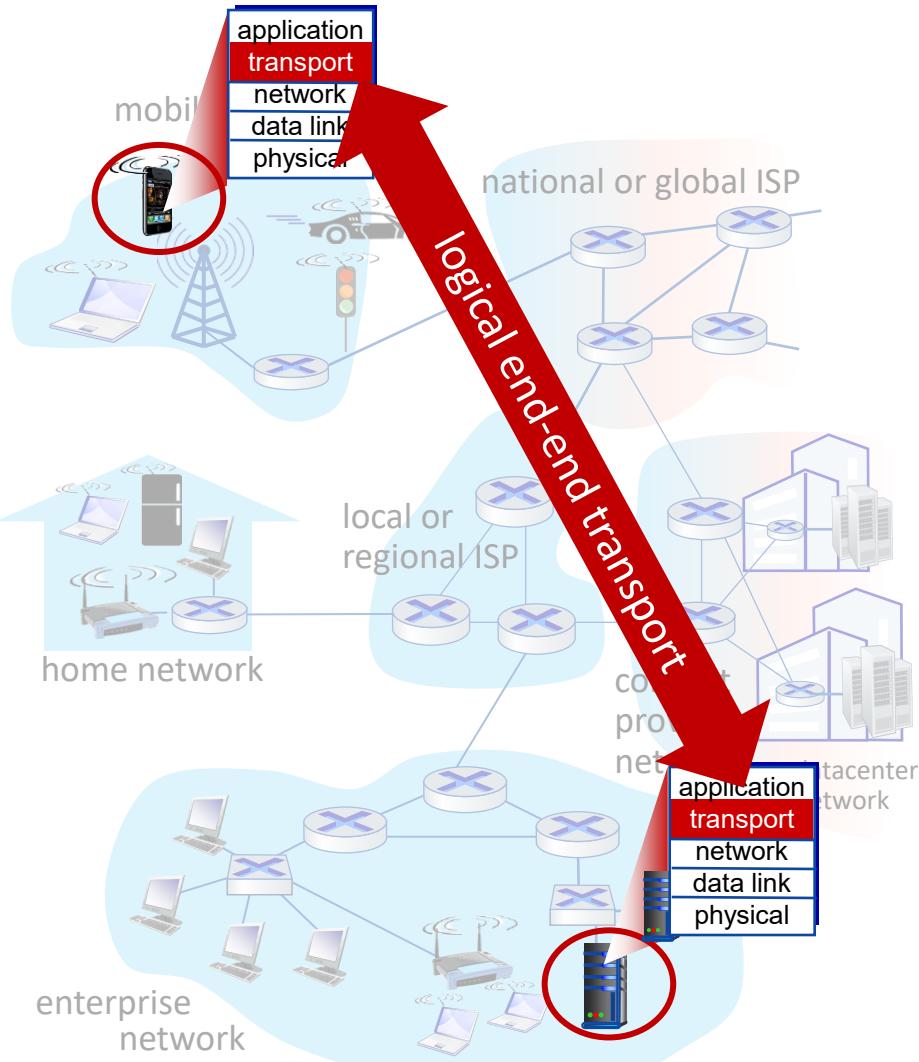
- Các dịch vụ tầng vận chuyển
- Multiplexing and demultiplexing
- UDP
- Nguyên lý truyền tin cậy
- TCP
- TCP - Điều khiển tắc nghẽn
- Sự phát triển của các tính năng của tầng vận chuyển





Giao thức và dịch vụ

- Cung cấp “*truyền thông luận lý* (*logical communication*)” giữa các tiến trình trên các “*host*” khác nhau
- Các giao thức vận chuyển hoạt động trên các thiết bị đầu cuối:
 - Bên gửi: chia các “*messages – thông điệp*” thành các “*segments*” và chuyển xuống tầng mạng.
 - Bên nhận: ghép các segment thành messages, chuyển đến tầng ứng dụng.
- 2 giao thức chính: TCP và UDP





Hoạt động của tầng vận chuyển

Bên gửi:

- Nhận message từ tầng ứng dụng (nhận thư)

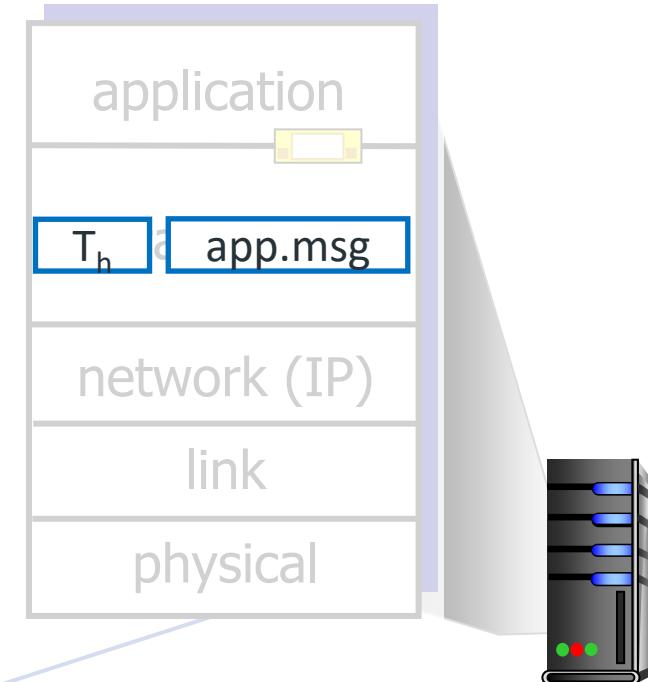
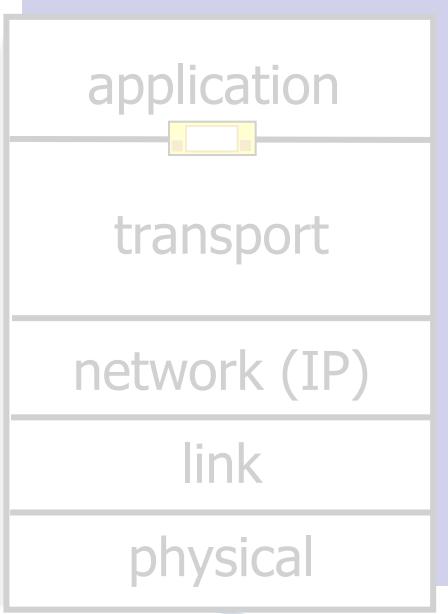




Hoạt động của tầng vận chuyển

Bên gửi:

- Nhận message từ tầng ứng dụng (nhận thư)
- Xác định giá trị header (thông tin phong bì)

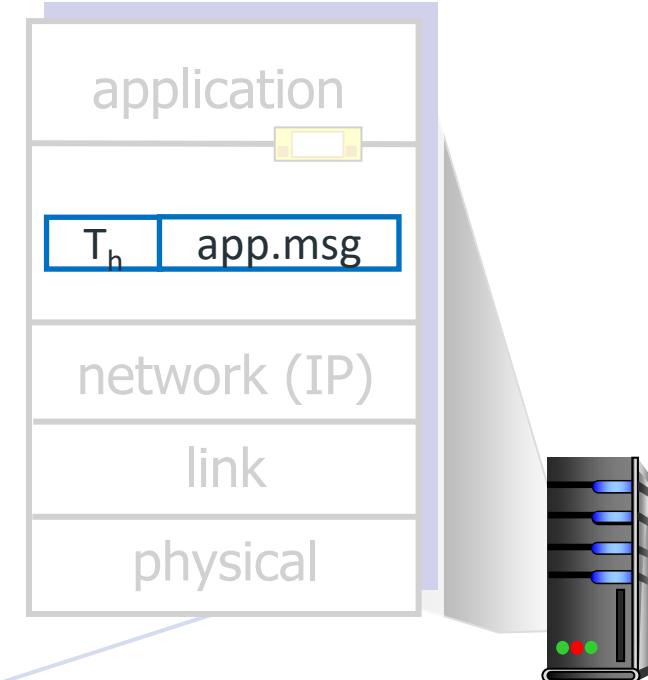
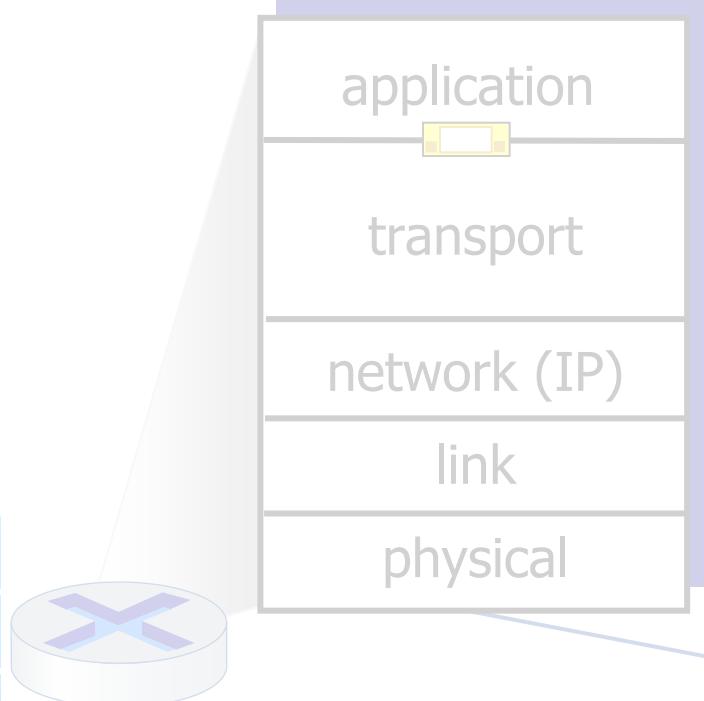




Hoạt động của tầng vận chuyển

Bên gửi:

- Nhận message từ tầng ứng dụng (nhận thư)
- Xác định giá trị header (thông tin phong bì)
- Tạo segment (bỏ thư vào phong bì)

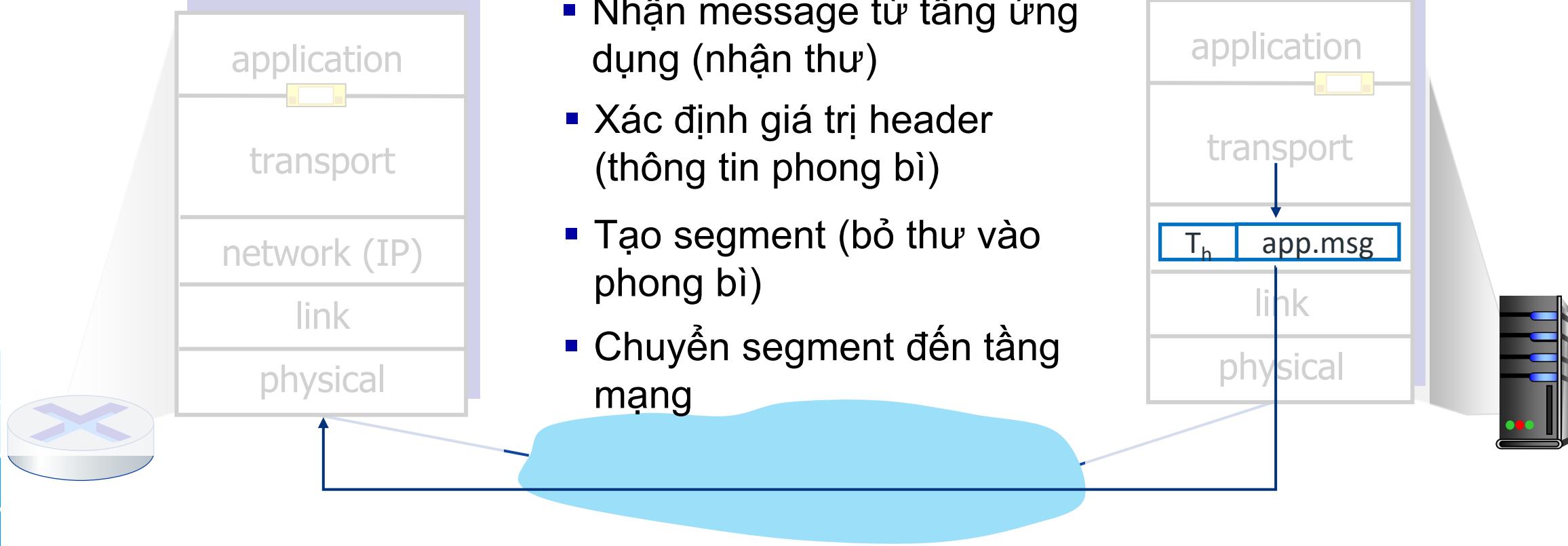




Hoạt động của tầng vận chuyển

Bên gửi:

- Nhận message từ tầng ứng dụng (nhận thư)
- Xác định giá trị header (thông tin phong bì)
- Tạo segment (bỏ thư vào phong bì)
- Chuyển segment đến tầng mạng

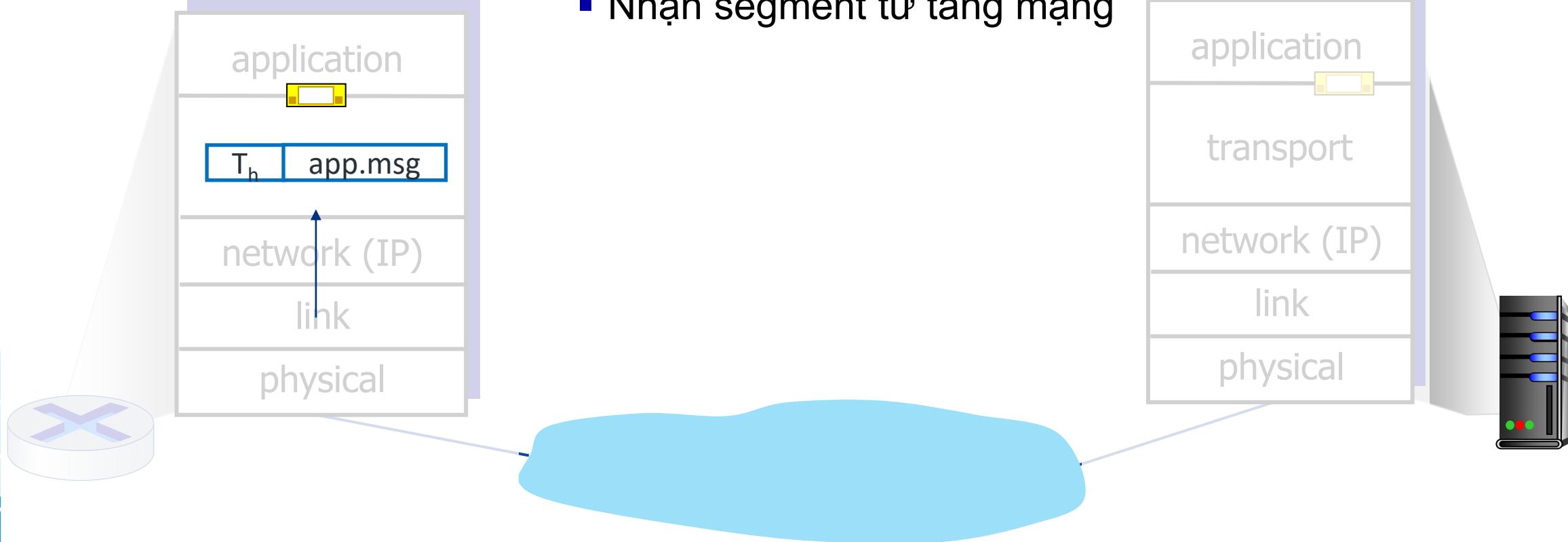




Transport Layer Actions

Bên nhận:

- Nhận segment từ tầng mạng

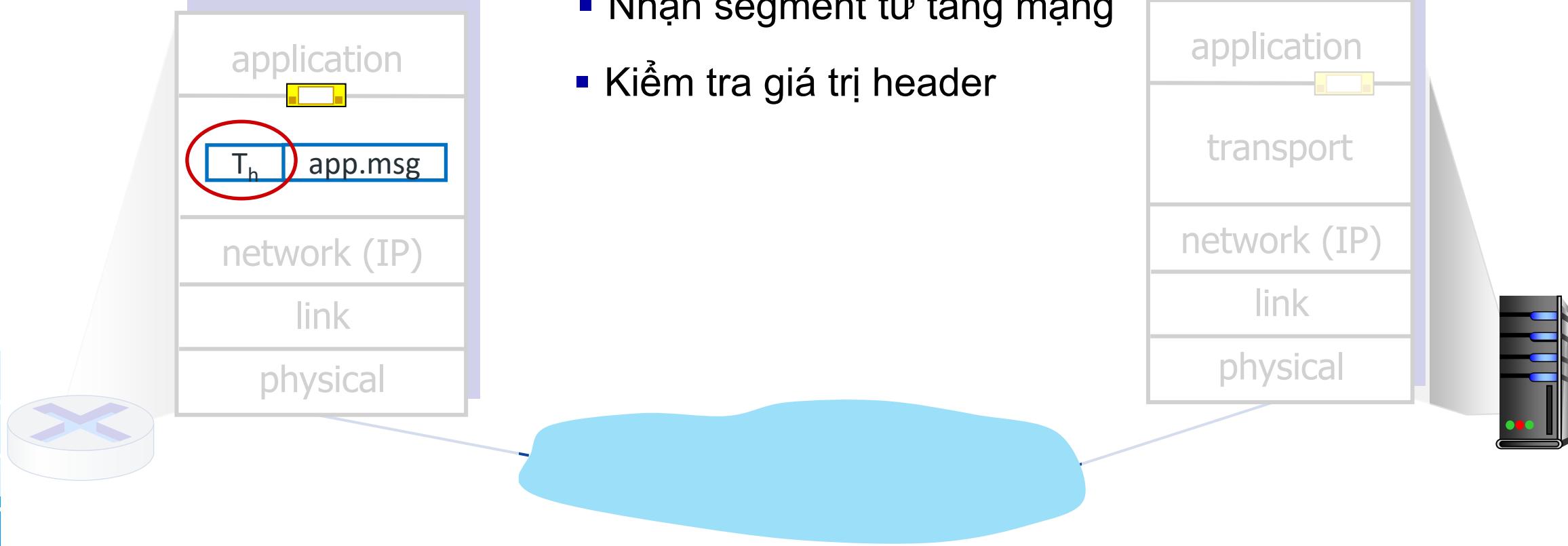




Transport Layer Actions

Bên nhận:

- Nhận segment từ tầng mạng
- Kiểm tra giá trị header

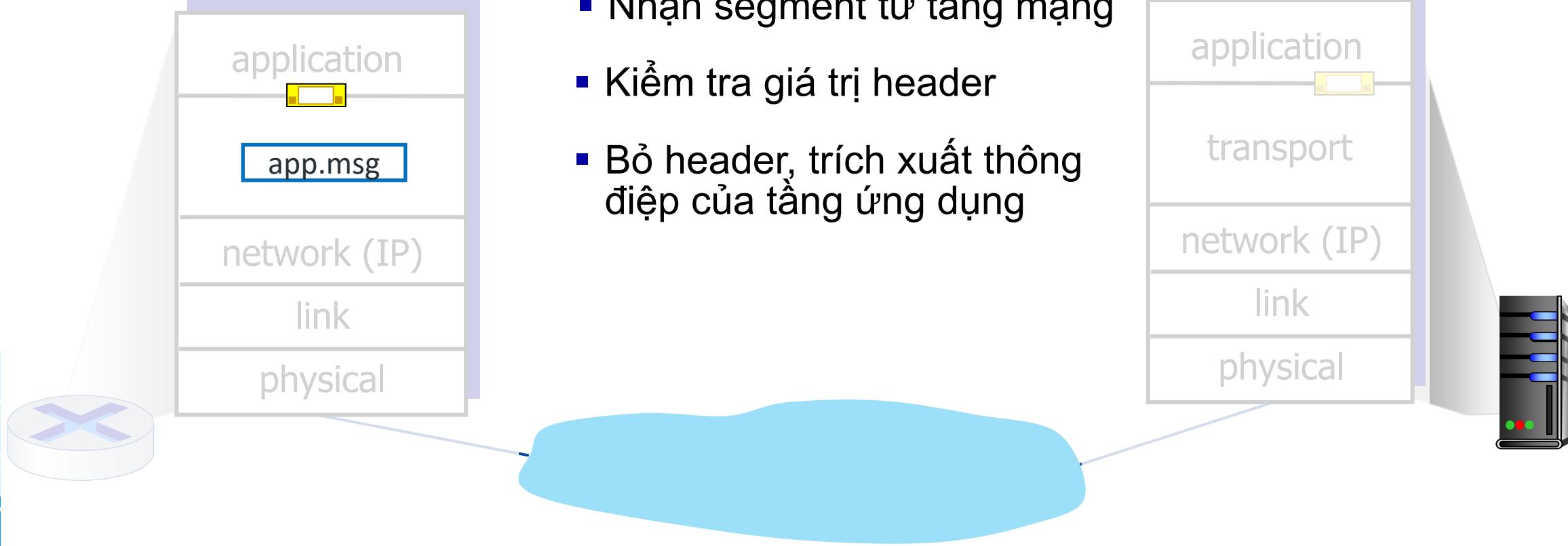




Transport Layer Actions

Bên nhận:

- Nhận segment từ tầng mạng
- Kiểm tra giá trị header
- Bỏ header, trích xuất thông điệp của tầng ứng dụng

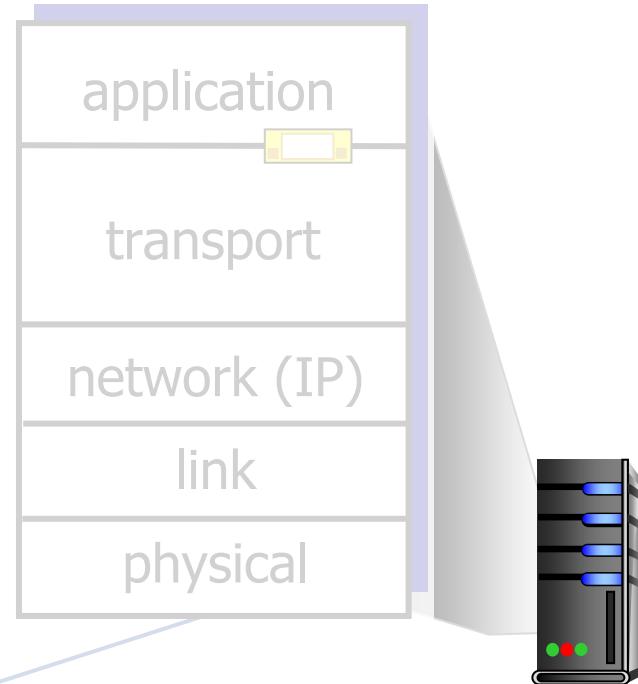
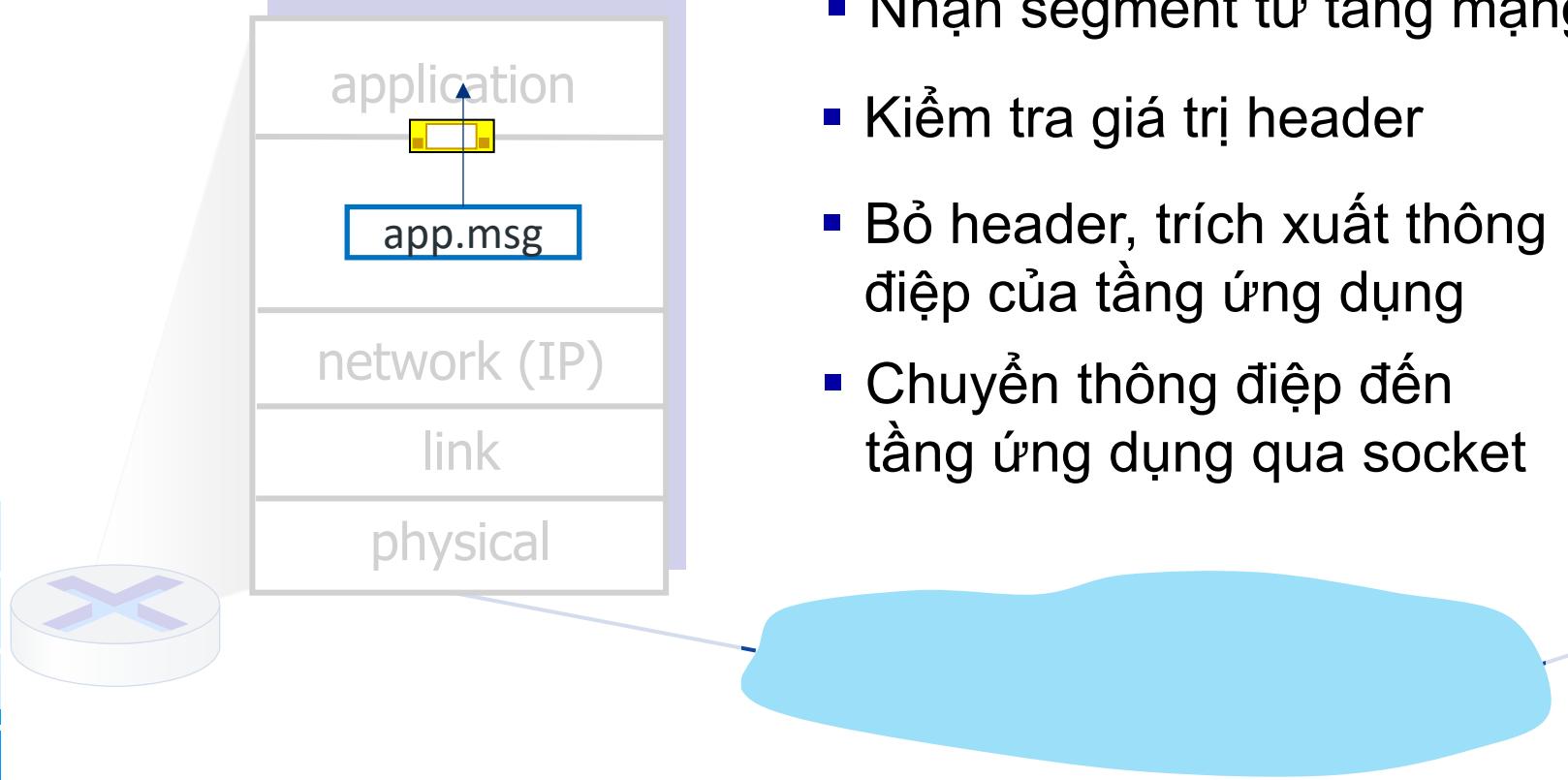




Transport Layer Actions

Bên nhận:

- Nhận segment từ tầng mạng
- Kiểm tra giá trị header
- Bỏ header, trích xuất thông điệp của tầng ứng dụng
- Chuyển thông điệp đến tầng ứng dụng qua socket





Giao thức tầng vận chuyển

○ **TCP:** Transmission Control Protocol

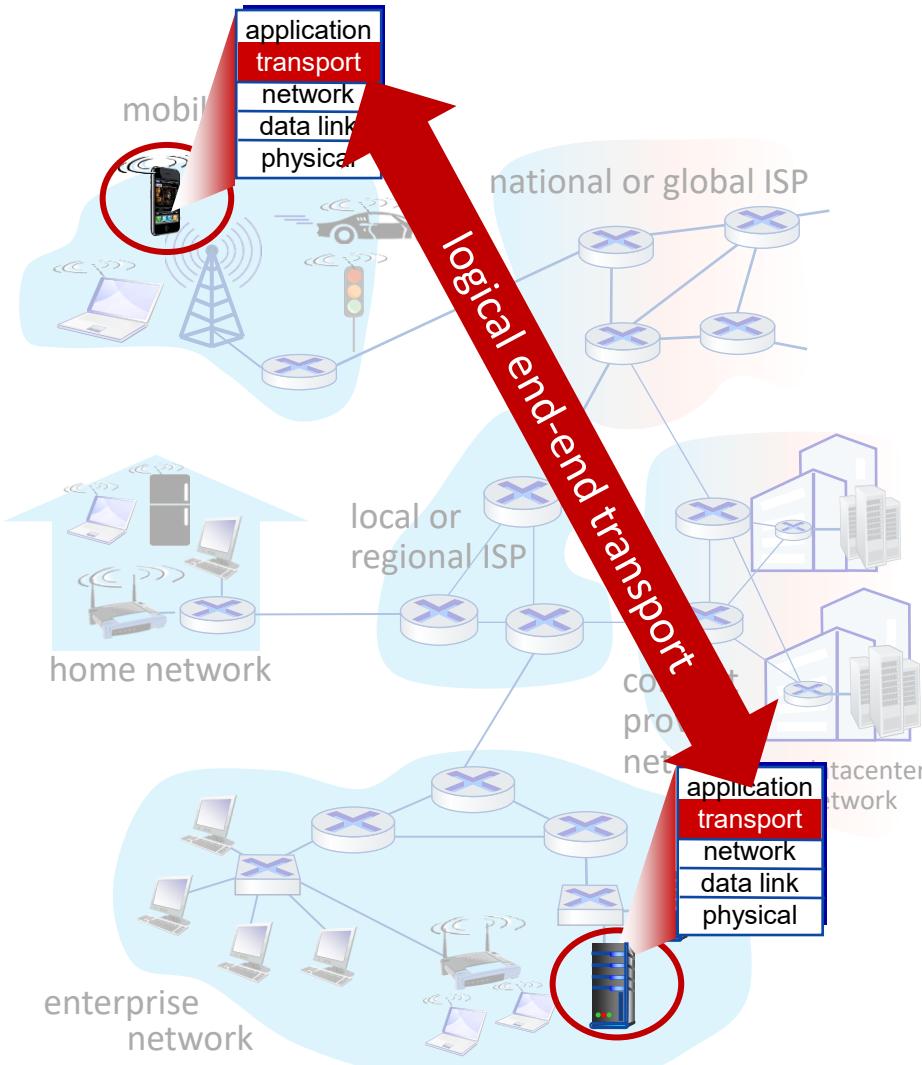
- Tin cậy, vận chuyển đúng thứ tự
- Điều khiển tắc nghẽn
- Điều khiển luồng
- Thiết lập kết nối

○ **UDP:** User Datagram Protocol

- Không tin cậy, truyền nhận không đúng thứ tự
- Phần mở rộng của giao thức IP “best-effort”

○ Không cung cấp các dịch vụ sau:

- Đảm bảo độ trễ
- Đảm bảo băng thông





Nội dung

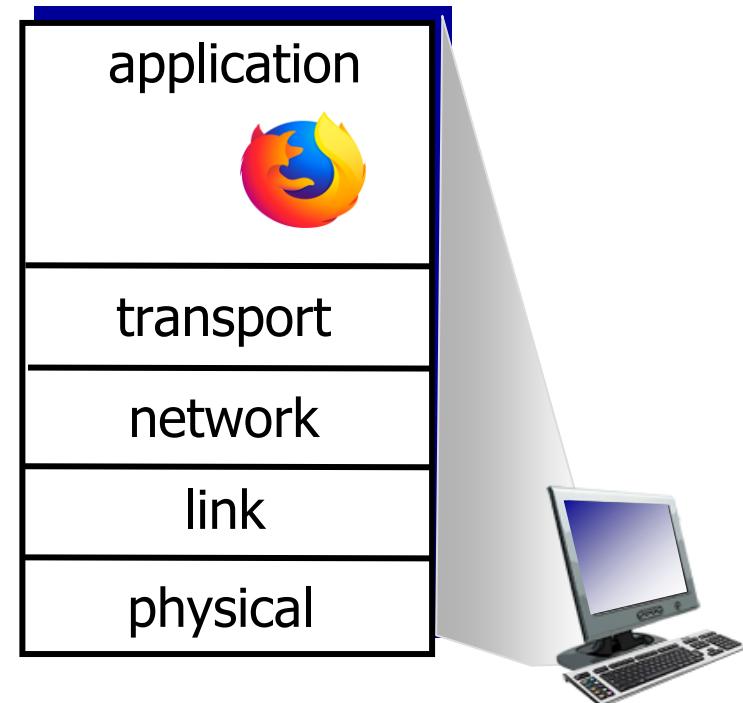
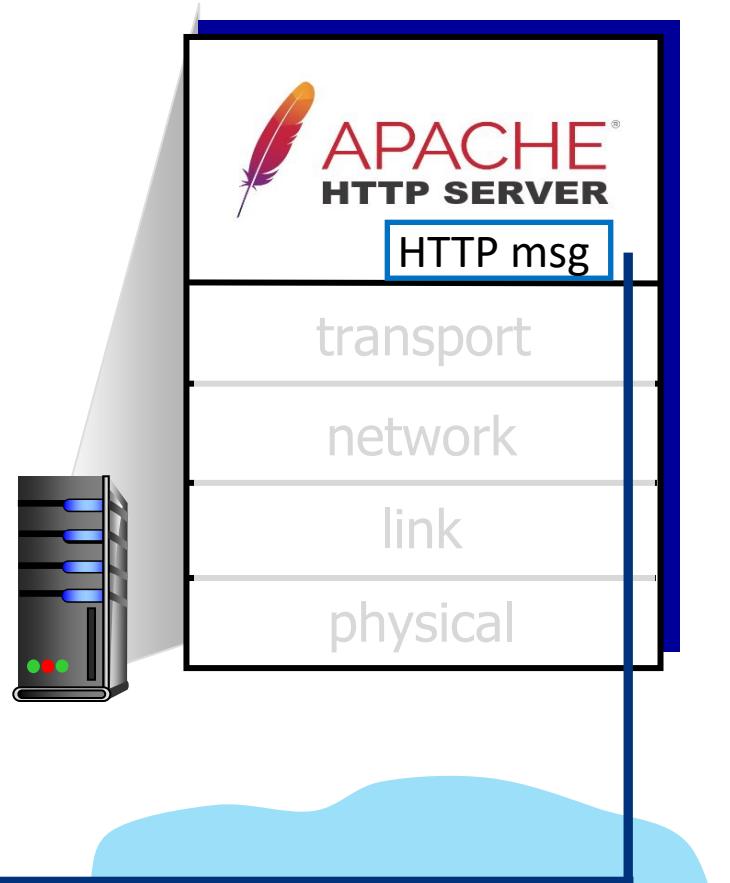
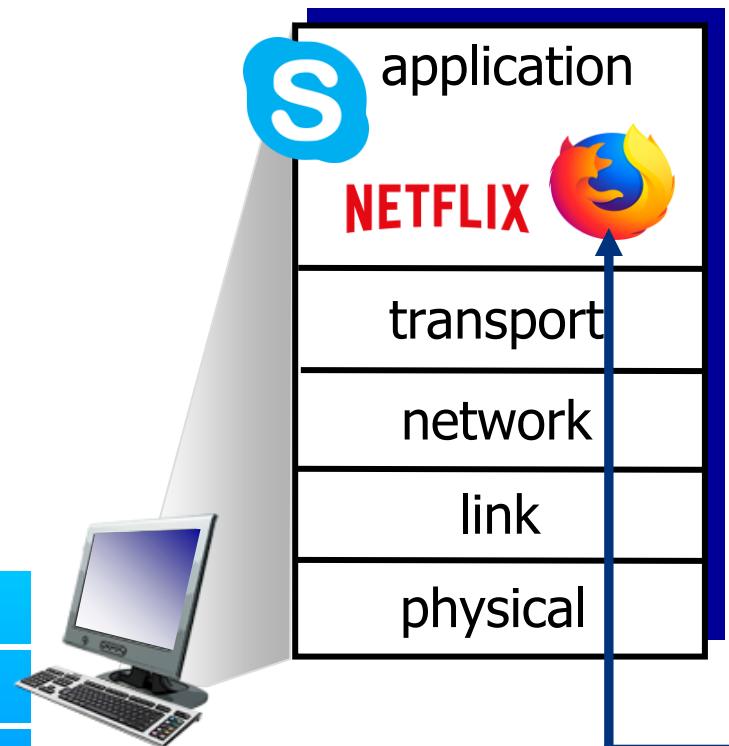
- Các dịch vụ tầng vận chuyển
- Multiplexing and demultiplexing
- UDP
- Nguyên lý truyền tin cậy
- TCP
- TCP - Điều khiển tắc nghẽn
- Sự phát triển của các tính năng của tầng vận chuyển





HTTP server

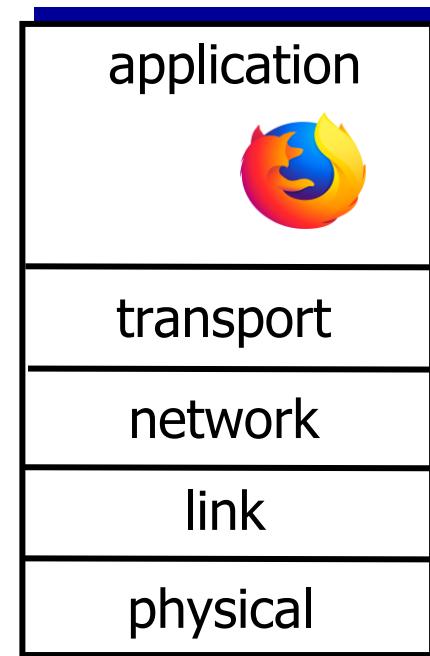
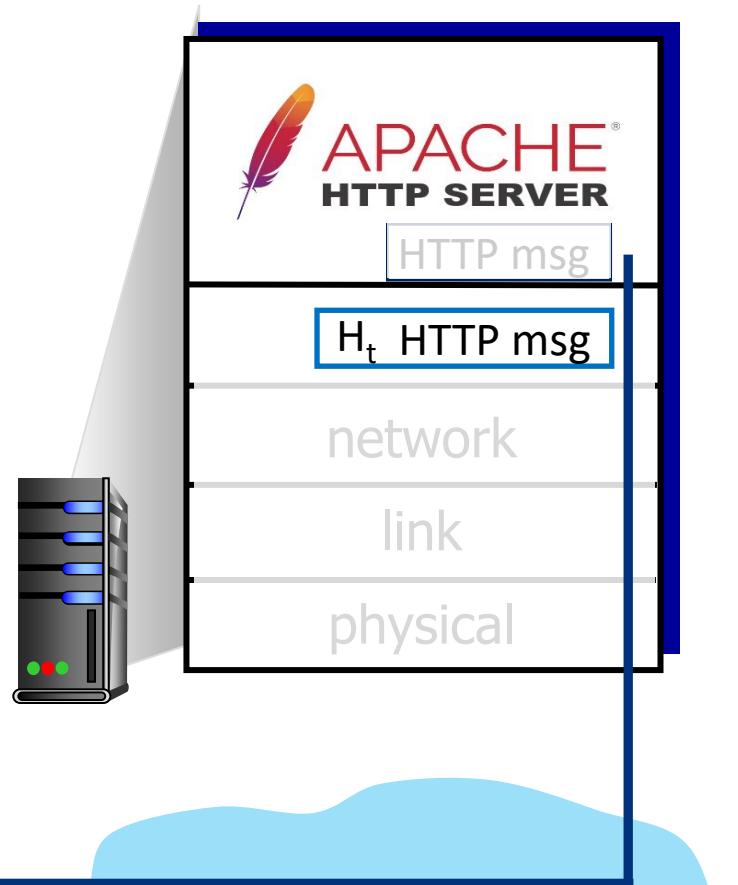
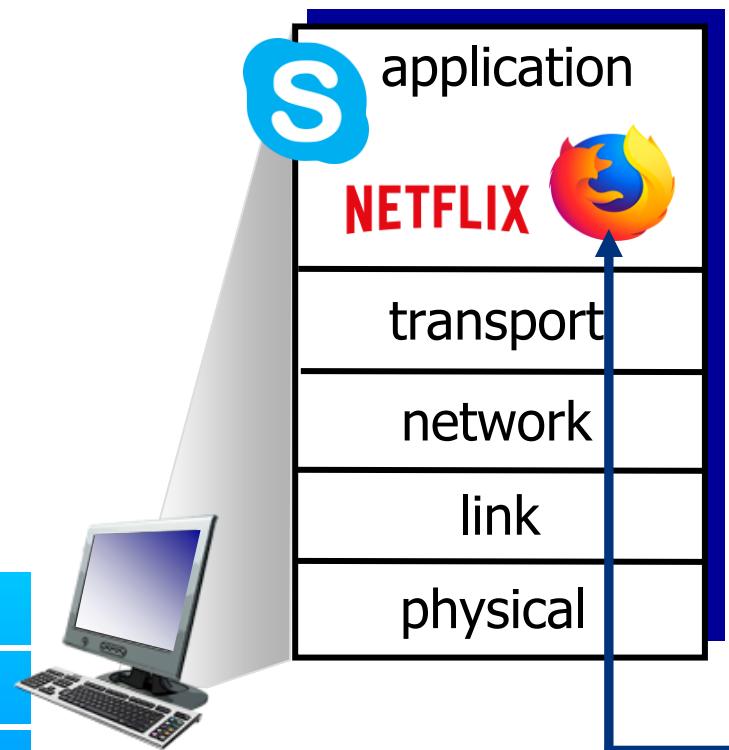
client





HTTP server

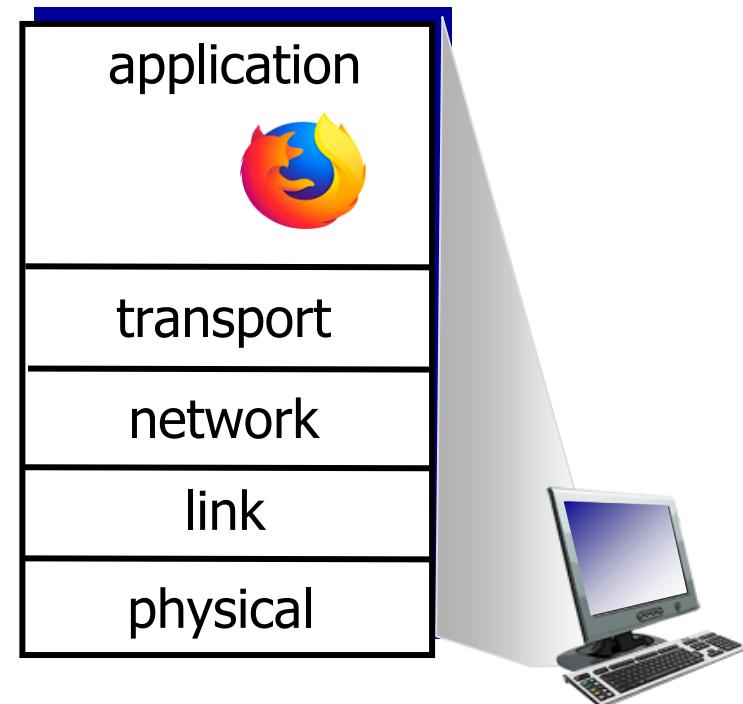
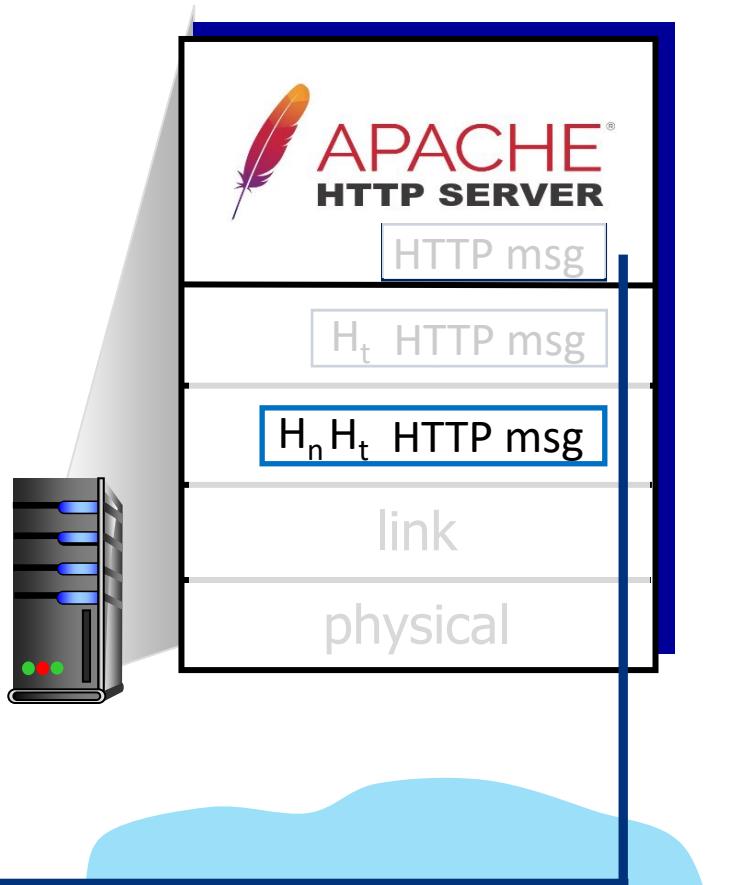
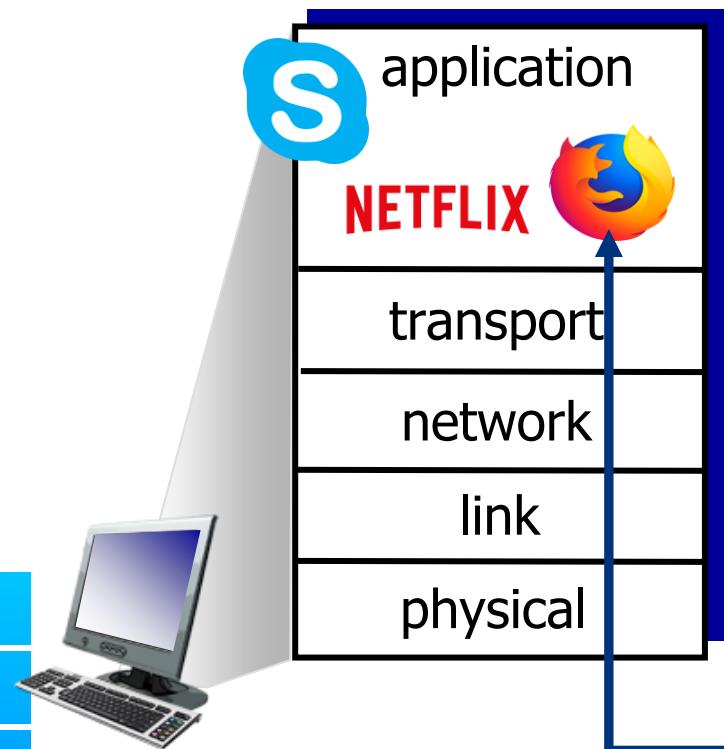
client





HTTP server

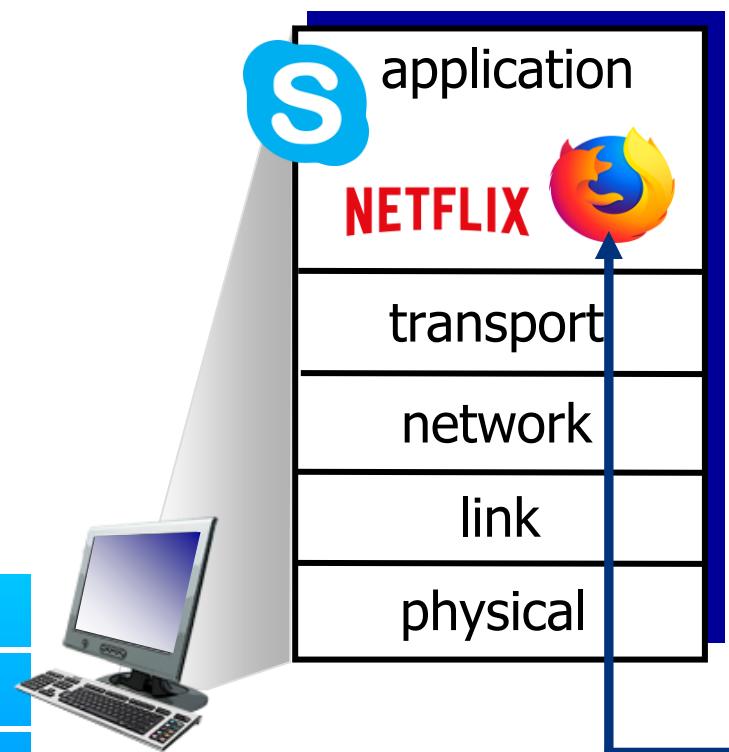
client



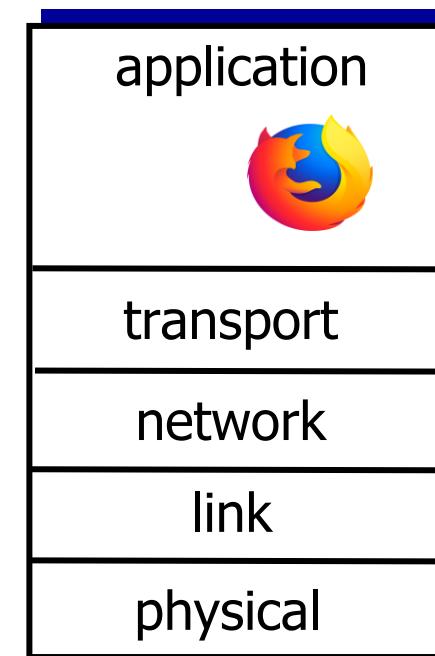
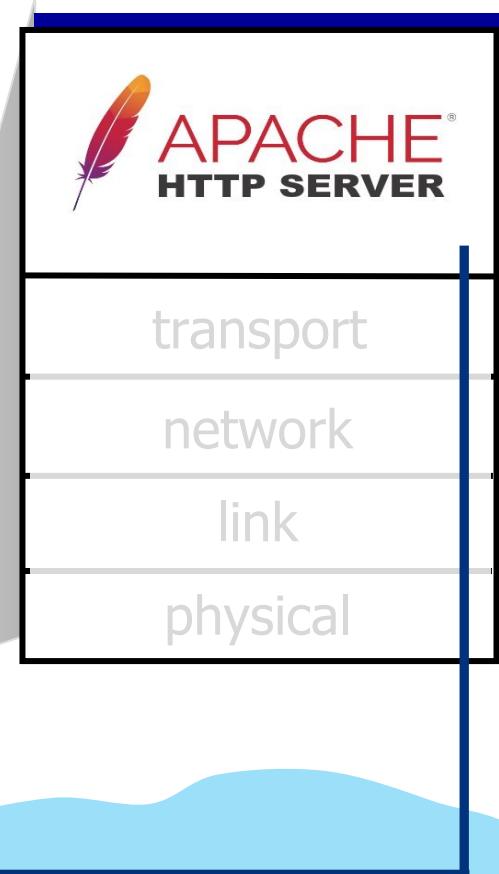


HTTP server

client



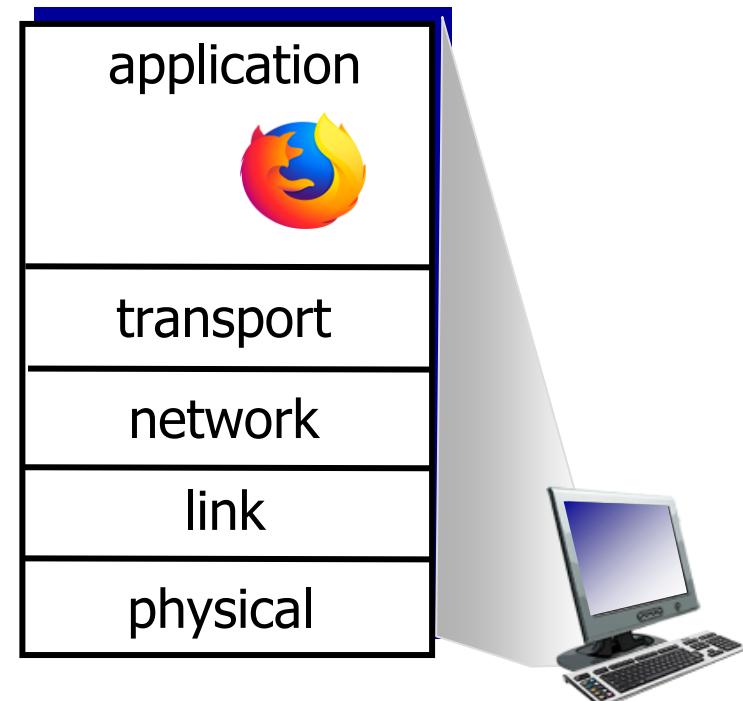
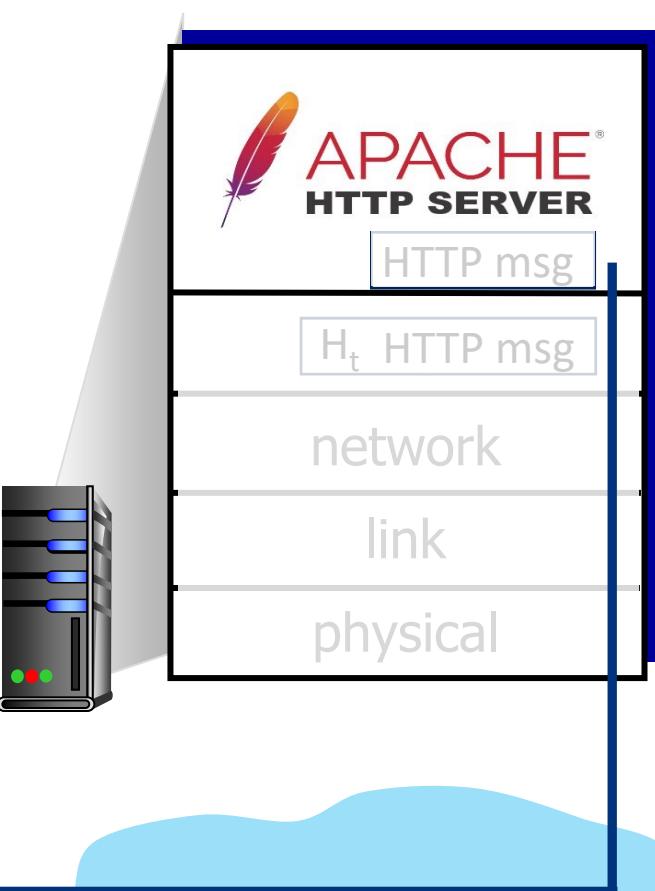
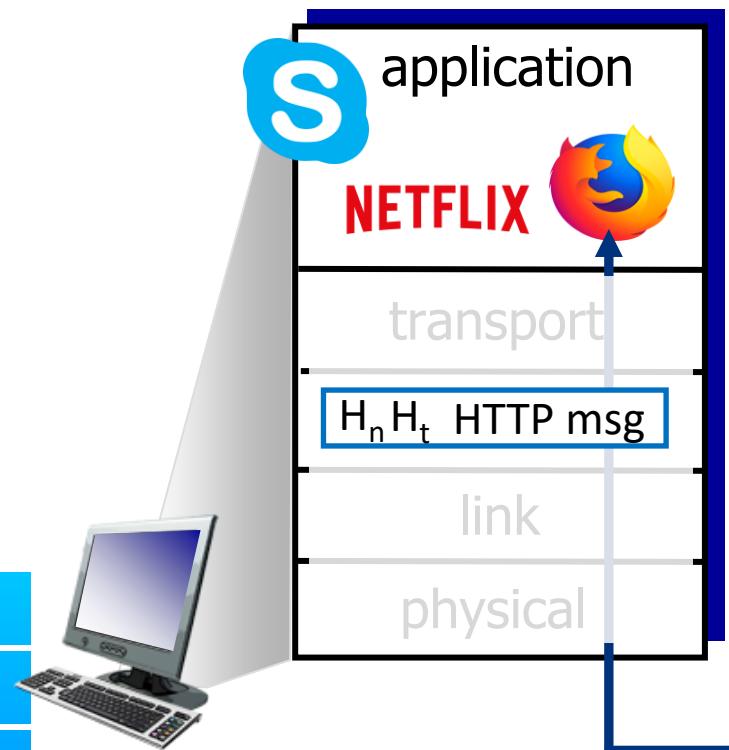
$H_n H_t$ HTTP msg





HTTP server

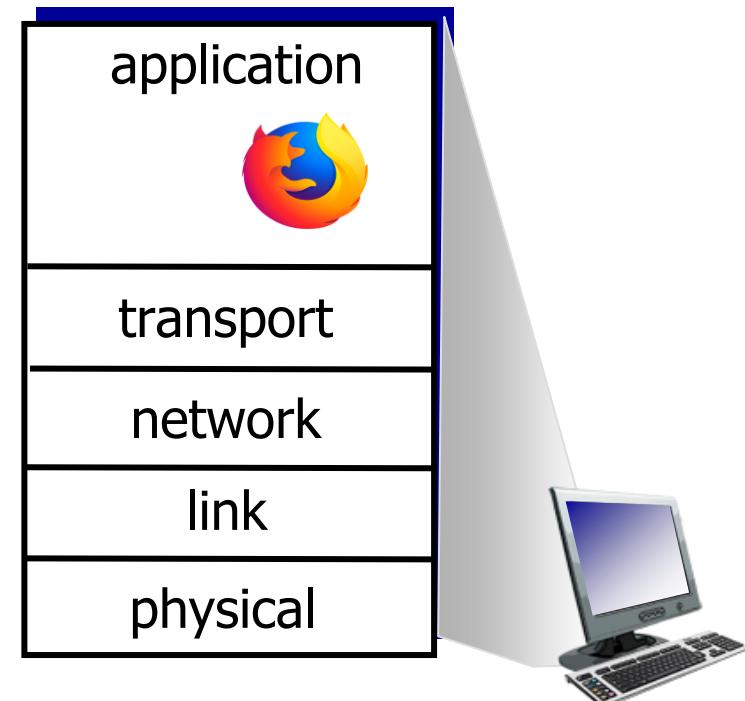
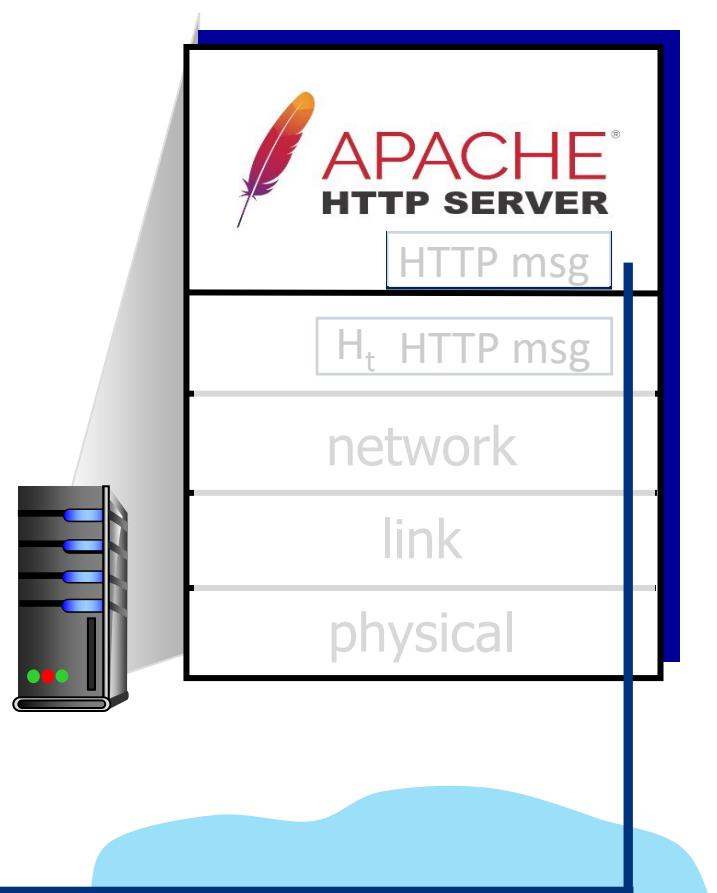
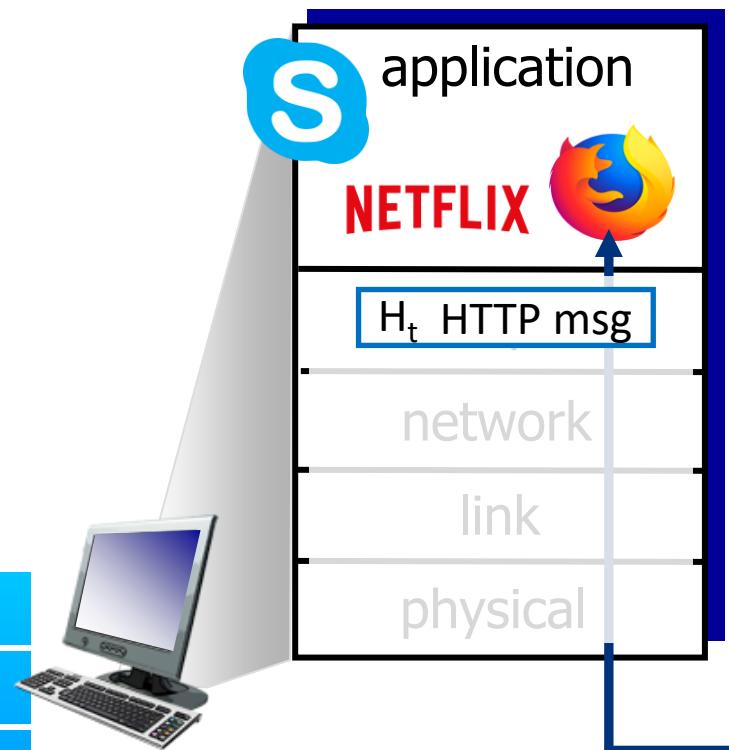
client





HTTP server

client

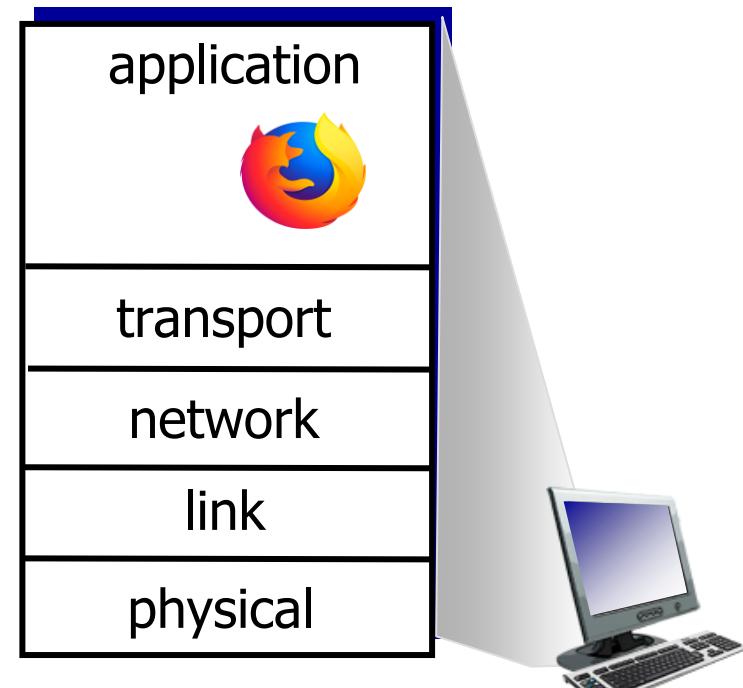
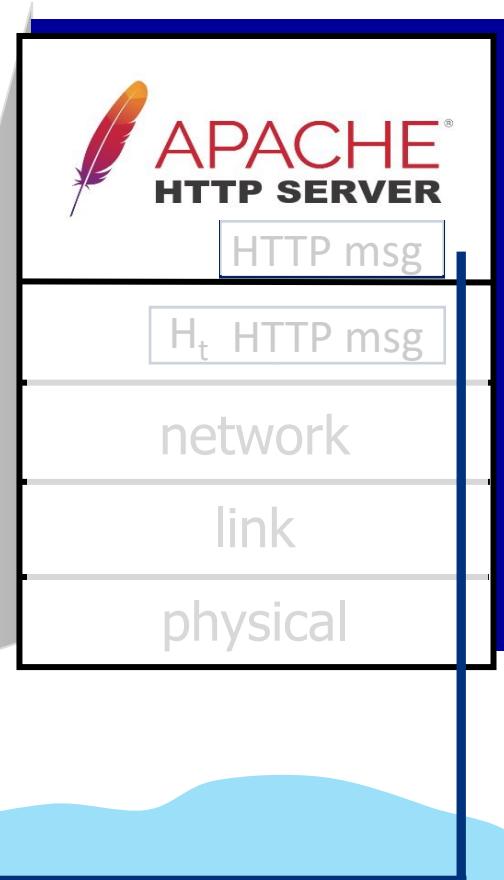
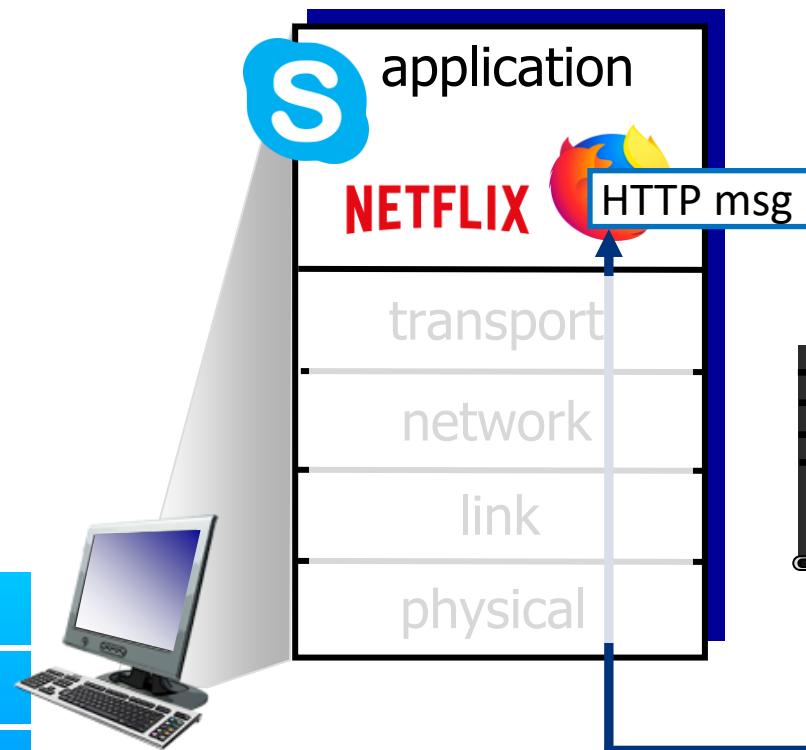




*Q: Làm thế nào mà tầng vận chuyển biết gửi đúng message
tới trình duyệt Firefox thay vì Netflix hoặc Skype?*



client



Multiplexing/demultiplexing

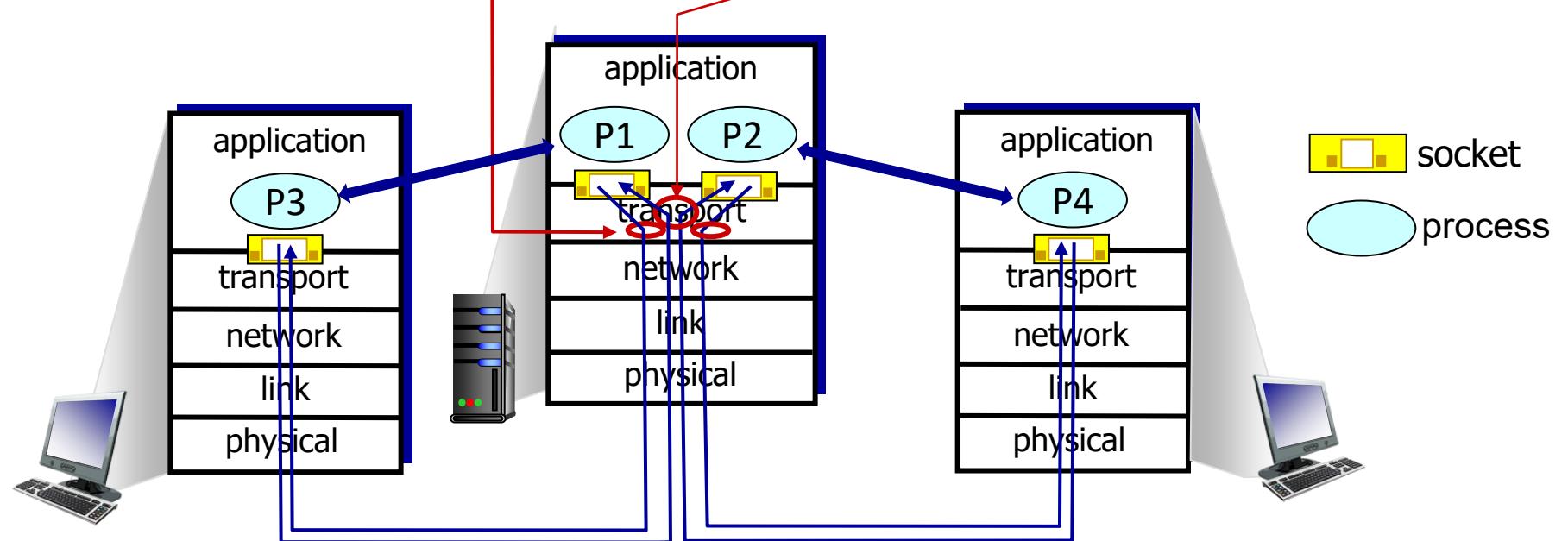


multiplexing tại bên gửi:

Nhận dữ liệu từ socket, thêm header của tầng vận chuyển

demultiplexing tại bên nhận:

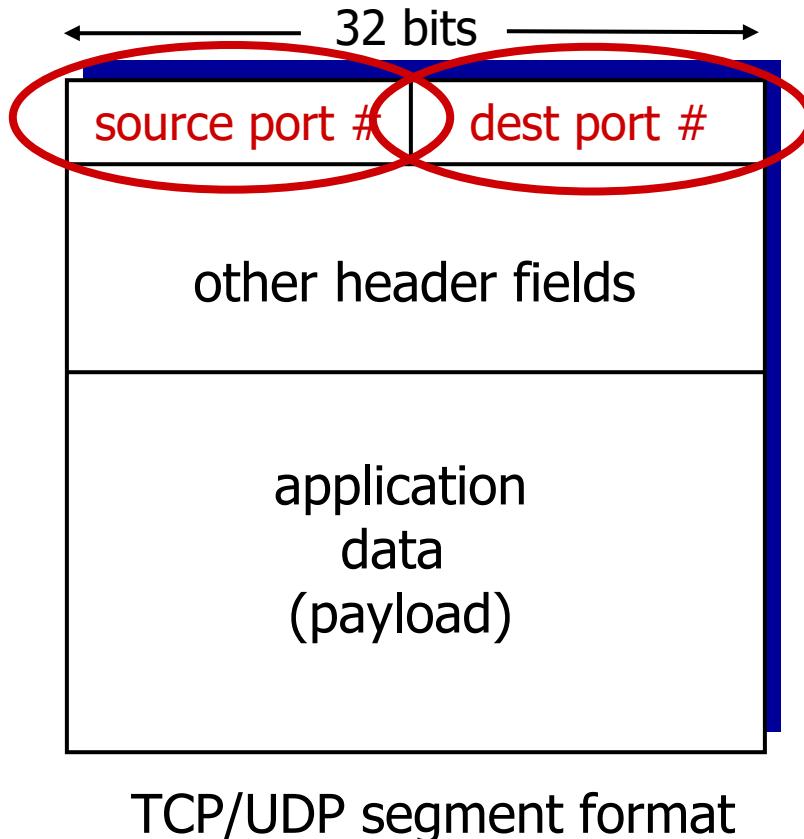
Sử dụng thông tin trong header để chuyển segment nhận được đến đúng socket.





Demultiplexing làm việc thế nào?

- Khi host nhận được một IP “datagram”
 - Mỗi datagram có địa chỉ IP nguồn và IP đích
 - Mỗi datagram này chứa 1 segment (đơn vị dữ liệu của tầng vận chuyển)
- Mỗi segment có port nguồn và port đích
 - “Host” dùng **địa chỉ IP & số port** để chuyển segment đến đúng socket tương ứng

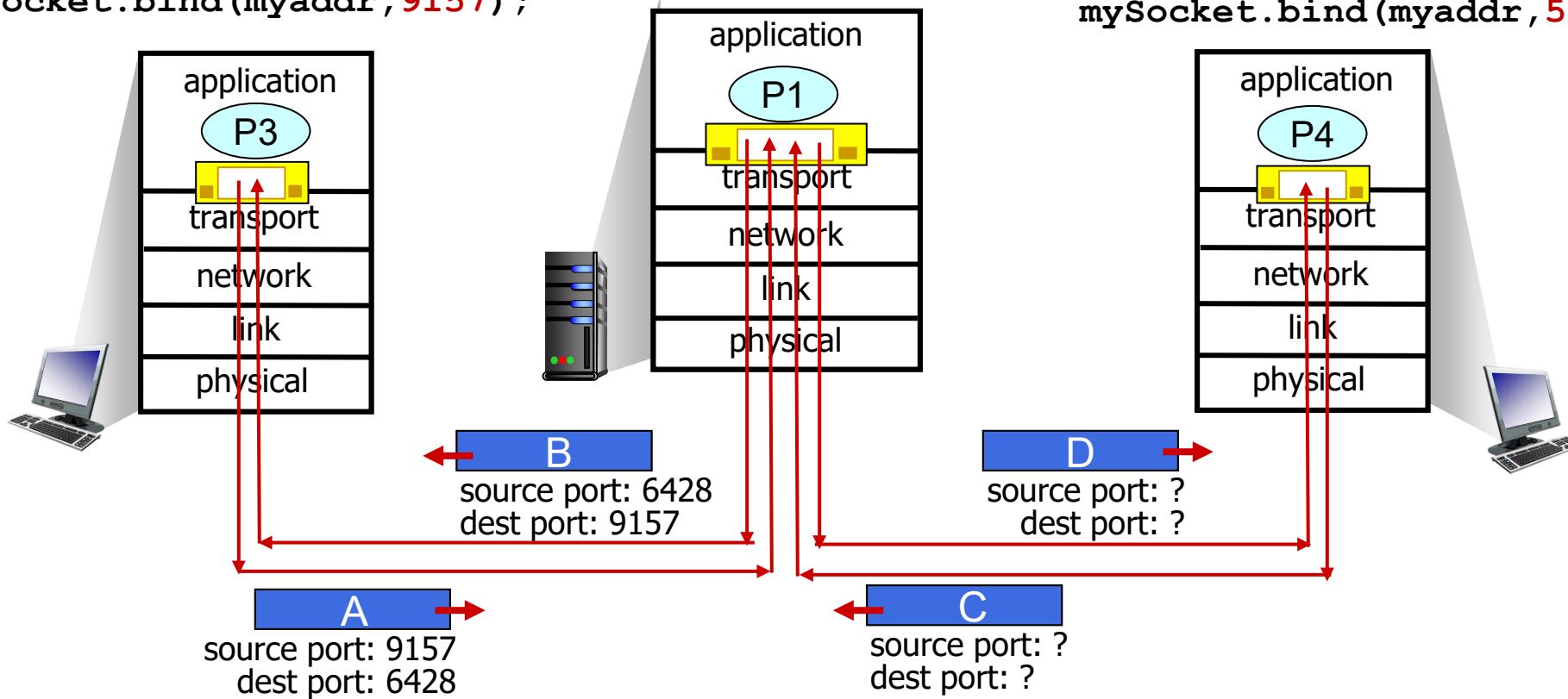


Ví dụ về “connectionless - không kết nối”

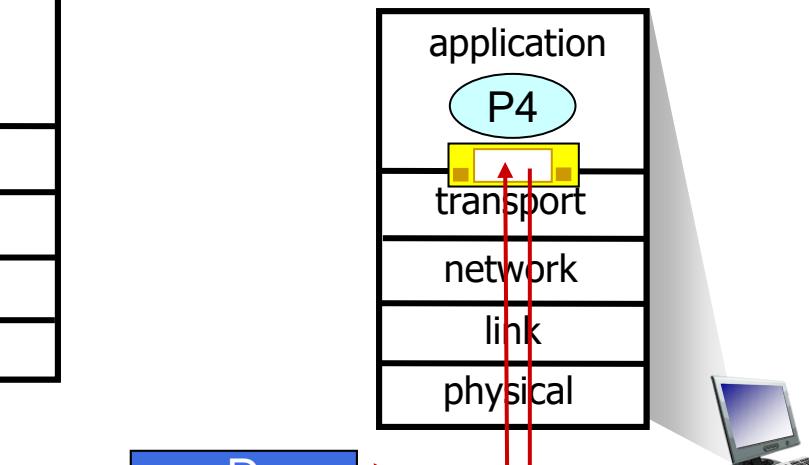


```
mySocket =  
    socket(AF_INET, SOCK_DGRAM)  
mySocket.bind(myaddr, 6428);
```

```
mySocket =  
    socket(AF_INET, SOCK_STREAM)  
mySocket.bind(myaddr, 9157);
```



```
mySocket =  
    socket(AF_INET, SOCK_STREAM)  
mySocket.bind(myaddr, 5775);
```



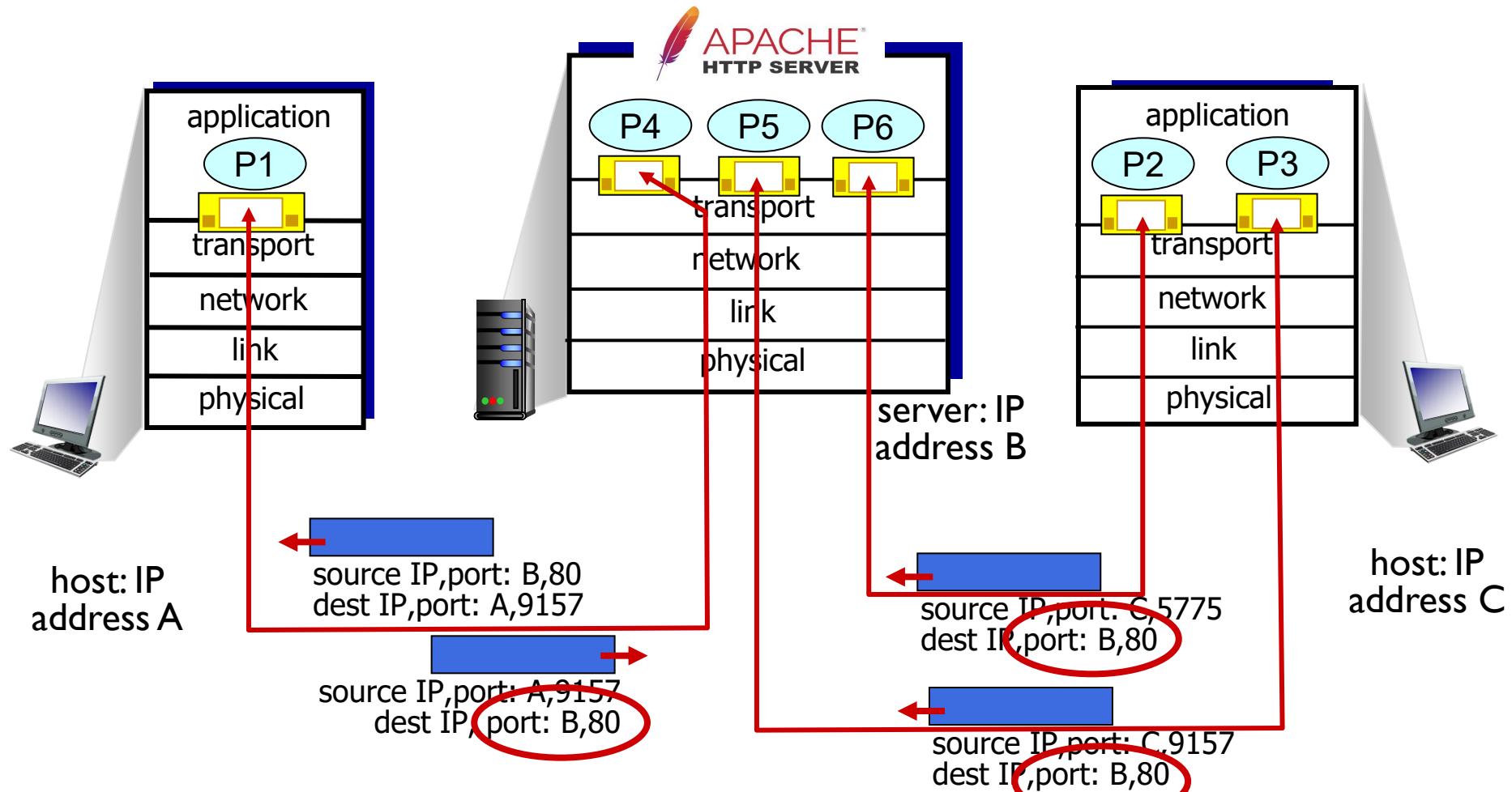


Hướng kết nối

- TCP socket được xác định bởi **4-tuple** (4 thông tin chính):
 - source IP address
 - source port number
 - dest IP address
 - dest port number
- demux: bên nhận sử dụng cả 4 thông tin chính để chuyển segment đến đúng socket
- Server có thể hỗ trợ nhiều TCP socket cùng lúc:
 - Mỗi socket được xác định bởi 4 thông tin
 - Mỗi socket tương ứng với một client đang kết nối với nó



Ví dụ về “hướng kết nối”



3 segments, đều có địa chỉ IP đích là B,
dest port: 80 được demultiplexed đến các socket *khác nhau*



Tổng kết về Multiplexing, demultiplexing

- Multiplexing, demultiplexing: dựa trên giá trị trong header của segment, datagram
 - **UDP**: demultiplexing chỉ sử dụng destination port number
 - **TCP**: demultiplexing sử dụng 4 thông tin: source và destination IP addresses, và port numbers
- Multiplexing/demultiplexing hoạt động ở tất cả các tầng



Nội dung

- Các dịch vụ tầng vận chuyển
- Multiplexing and demultiplexing
- UDP
- Nguyên lý truyền tin cậy
- TCP
- TCP - Điều khiển tắc nghẽn
- Sự phát triển của các tính năng của tầng vận chuyển





UDP: User Datagram Protocol

- Một giao thức đơn giản
- Cung cấp dịch vụ “**best effort**”
 - Segment có thể mất
 - Segment có thể đến không đúng thứ tự
- “**connectionless**”:
 - Không có bước thiết lập kết nối
 - Mỗi segment được xử lý độc lập

Tại sao vẫn dùng UDP?

- Không có bước thiết lập kết nối (bước này làm tăng độ trễ)
- Đơn giản: không lưu trữ trạng thái kết nối
- Header nhỏ gọn
- Không điều khiển tắc nghẽn
 - Segment đi nhanh nhất có thể



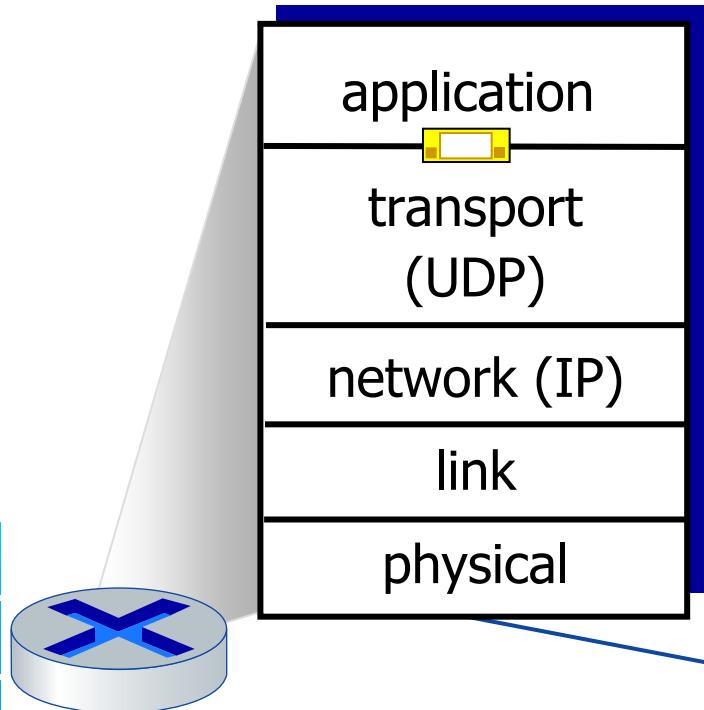
UDP: User Datagram Protocol

- UDP được sử dụng bởi:
 - streaming multimedia apps (cho phép mất mát, cần tốc độ)
 - DNS
 - SNMP
 - HTTP/3
- Nếu cần truyền tin cậy qua UDP (e.g., HTTP/3):
 - Thêm xử lý tin cậy ở tầng ứng dụng
 - Thêm điều khiển tắc nghẽn ở tầng ứng dụng

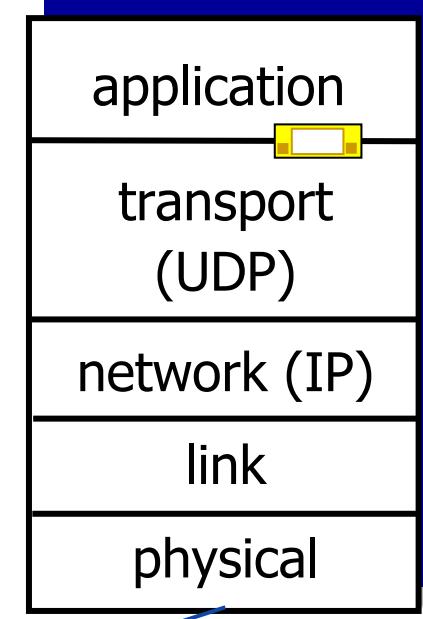
UDP: Hoạt động



SNMP client



SNMP server



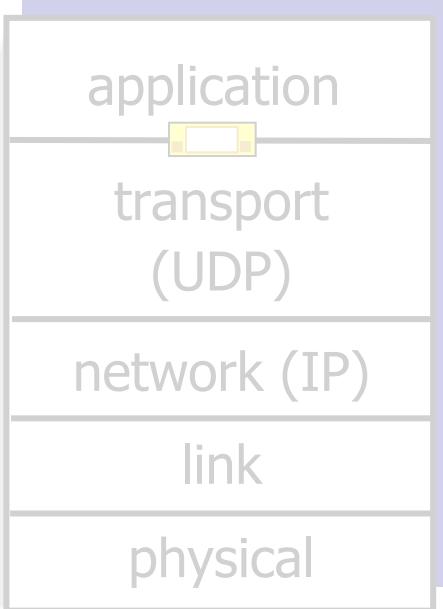


UDP: Hoạt động

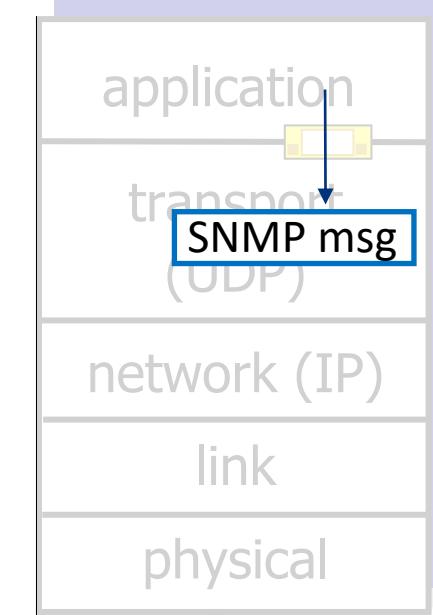
Bên gửi:

- Nhận message từ tầng ứng dụng

SNMP client



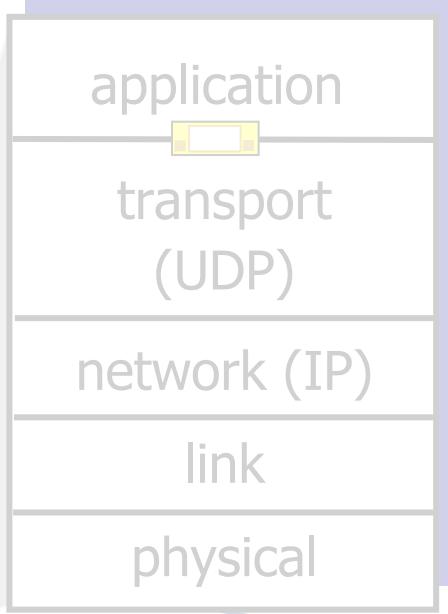
SNMP server



UDP: Hoạt động



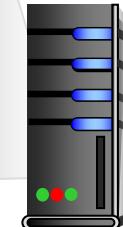
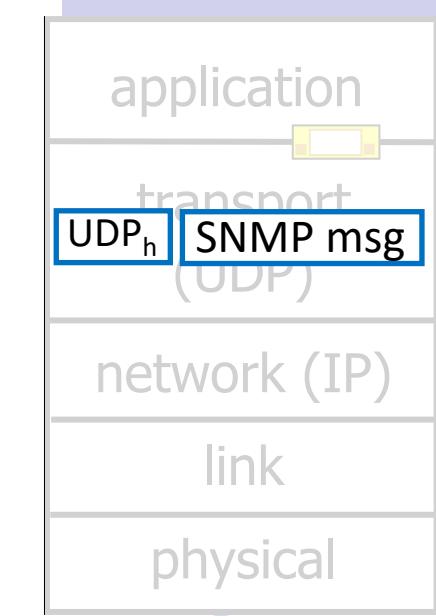
SNMP client



Bên gửi:

- Nhận message từ tầng ứng dụng
- Xác định giá trị header

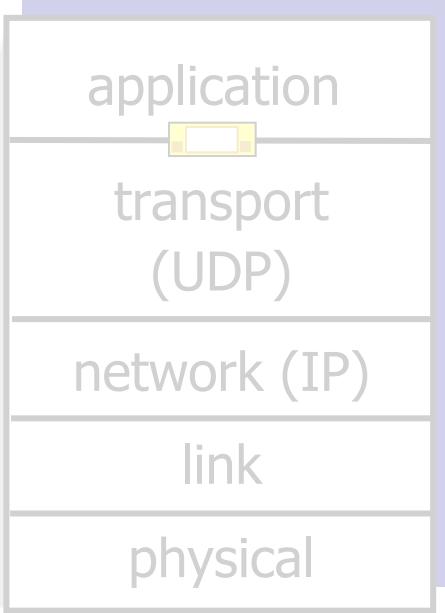
SNMP server



UDP: Hoạt động



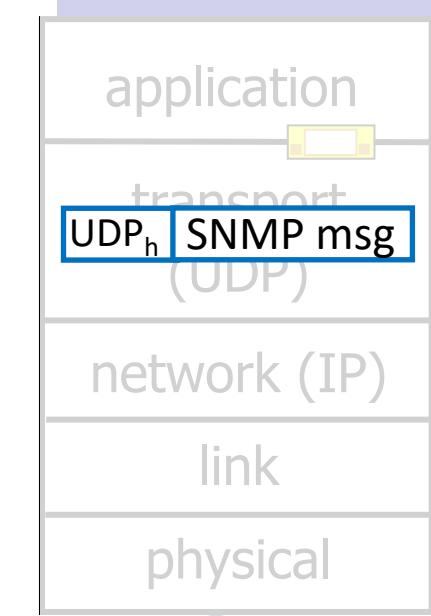
SNMP client



Bên gửi:

- Nhận message từ tầng ứng dụng
- Xác định giá trị header
- Tạo segment

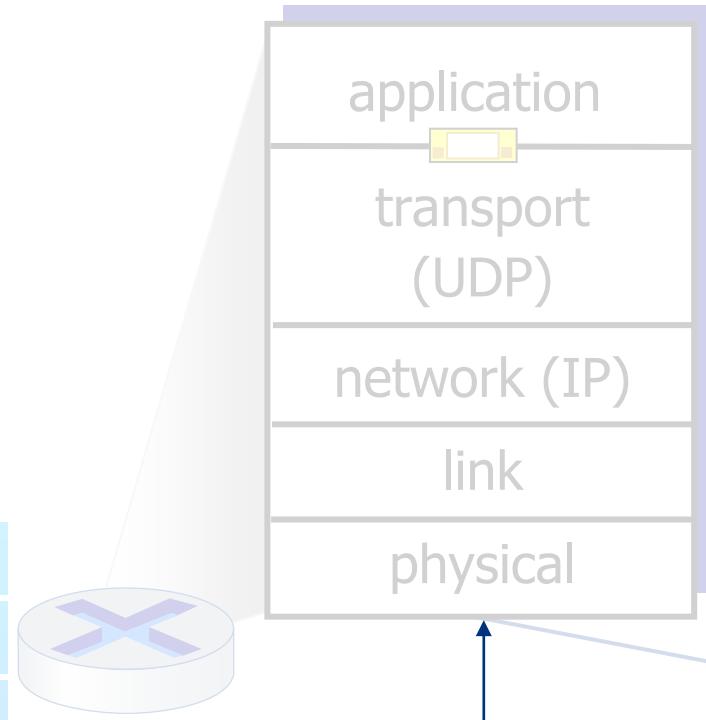
SNMP server



UDP: Hoạt động



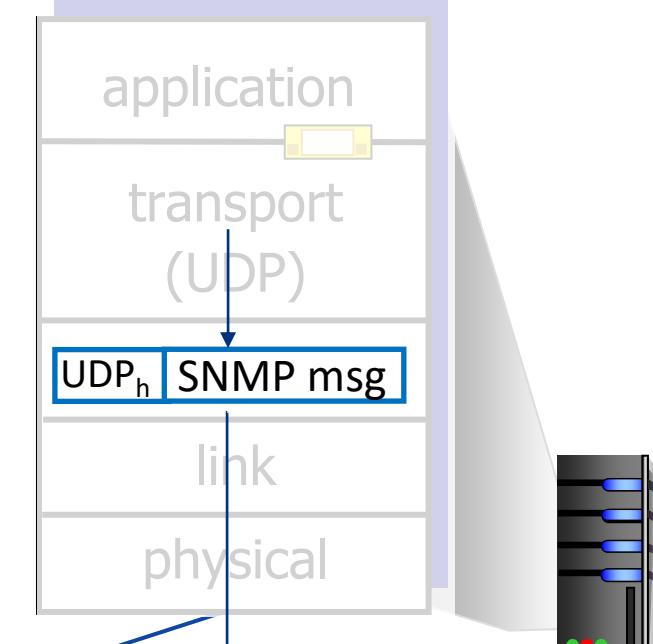
SNMP client



Bên gửi:

- Nhận message từ tầng ứng dụng
- Xác định giá trị header
- Tạo segment
- Chuyển segment đến tầng mạng

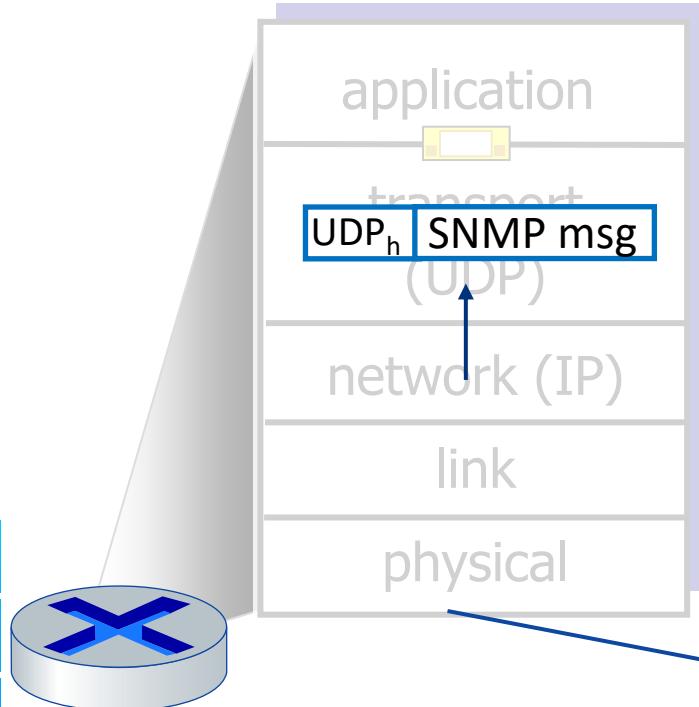
SNMP server



UDP: Hoạt động



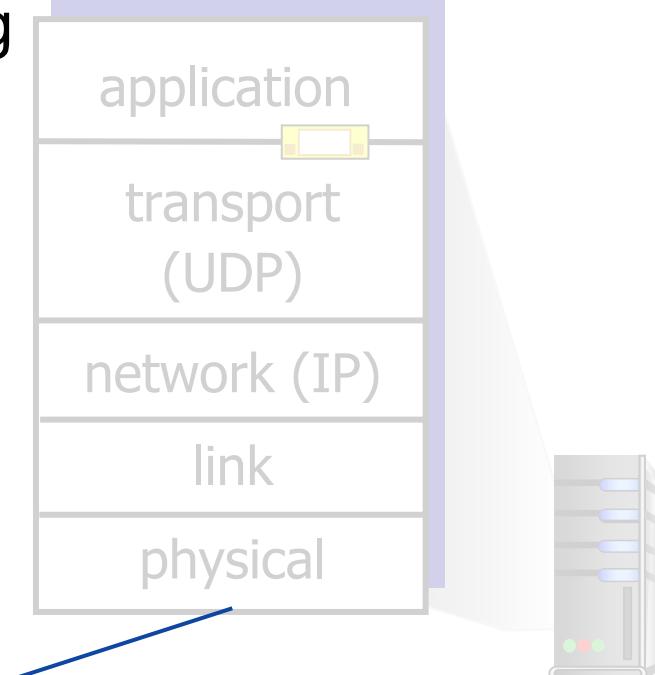
SNMP client



Bên nhận:

- Nhận segment từ tầng mạng

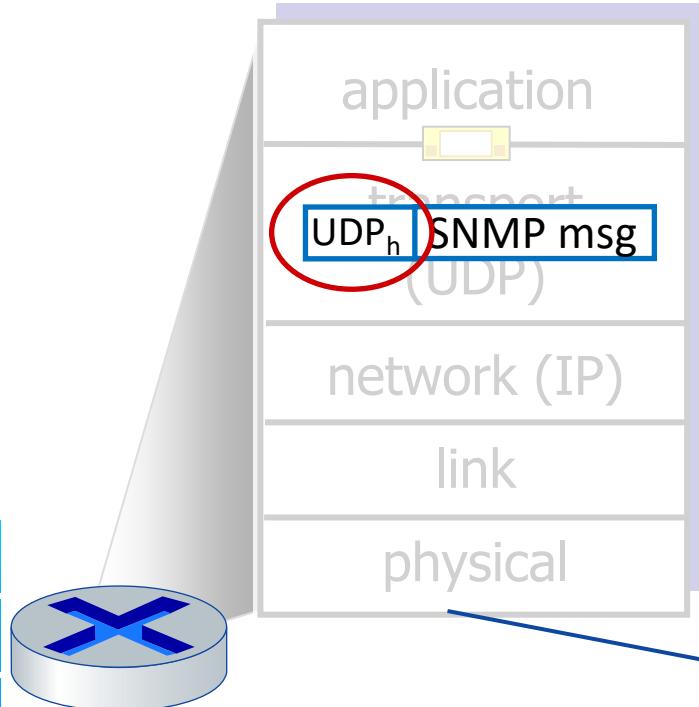
SNMP server



UDP: Hoạt động



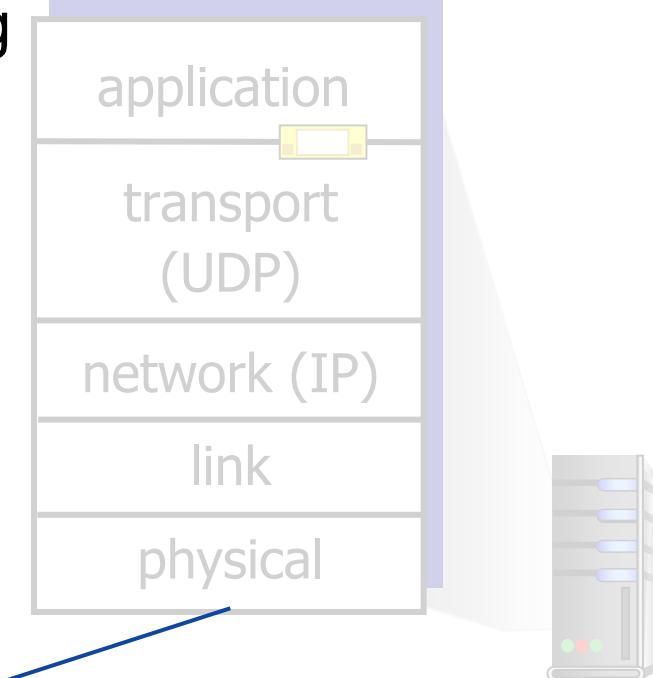
SNMP client



Bên nhận:

- Nhận segment từ tầng mạng
- Kiểm tra lỗi với giá trị checksum

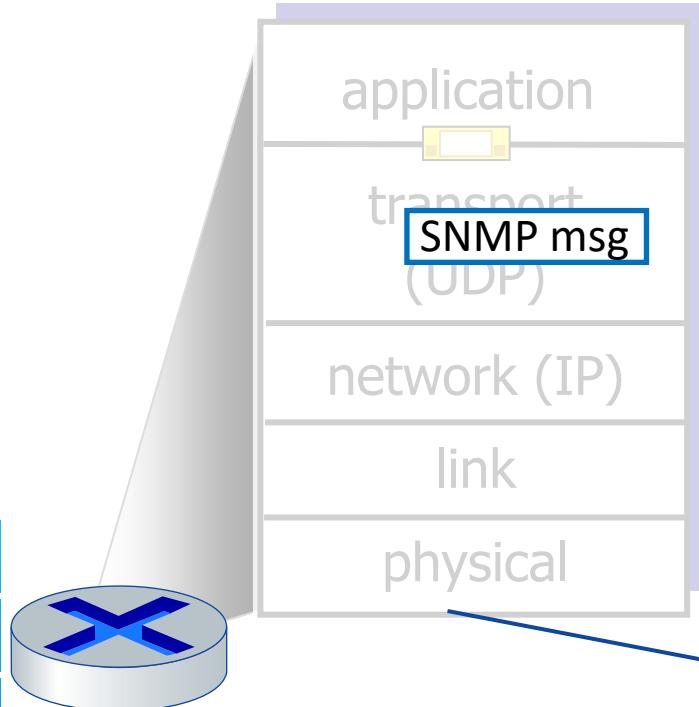
SNMP server



UDP: Hoạt động



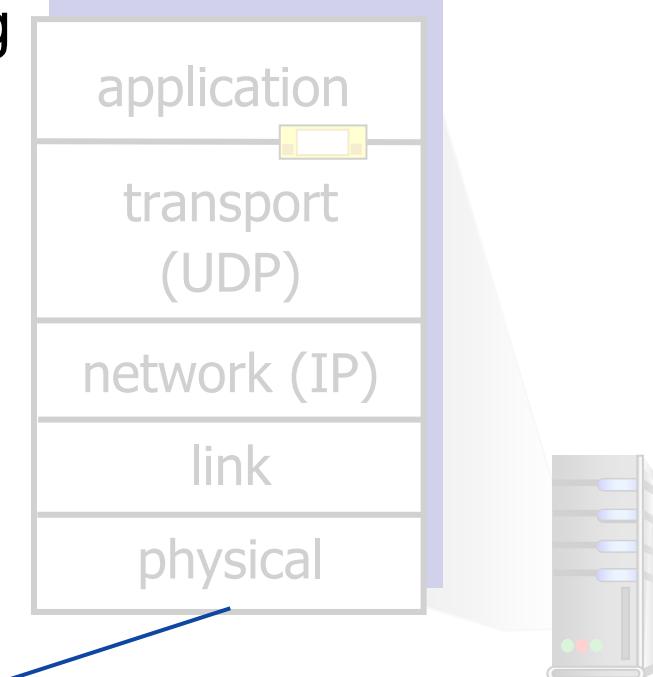
SNMP client



Bên nhận:

- Nhận segment từ tầng mạng
- Kiểm tra lỗi với giá trị checksum
- Bỏ header, lấy messages

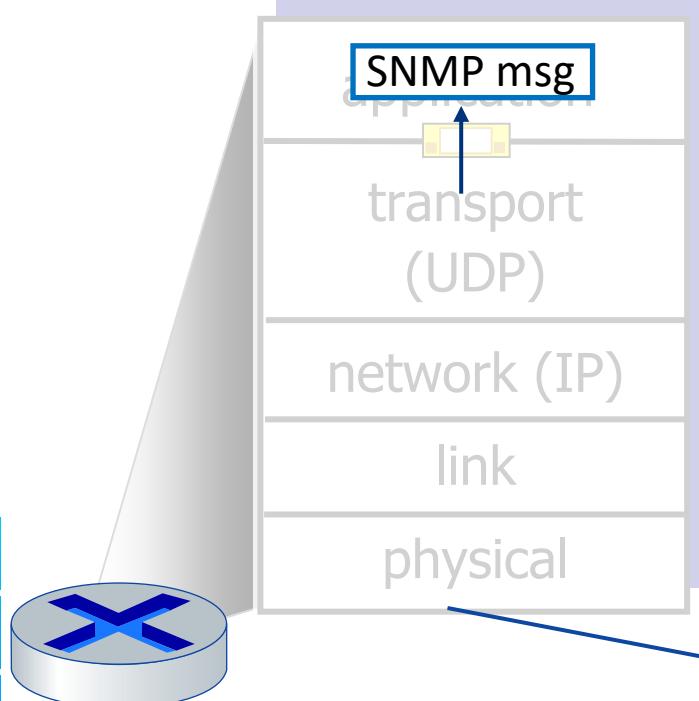
SNMP server



UDP: Hoạt động



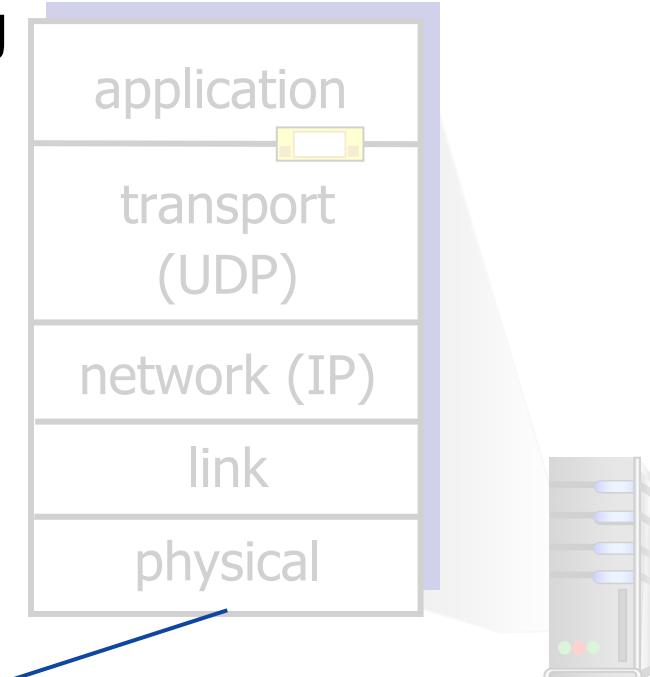
SNMP client



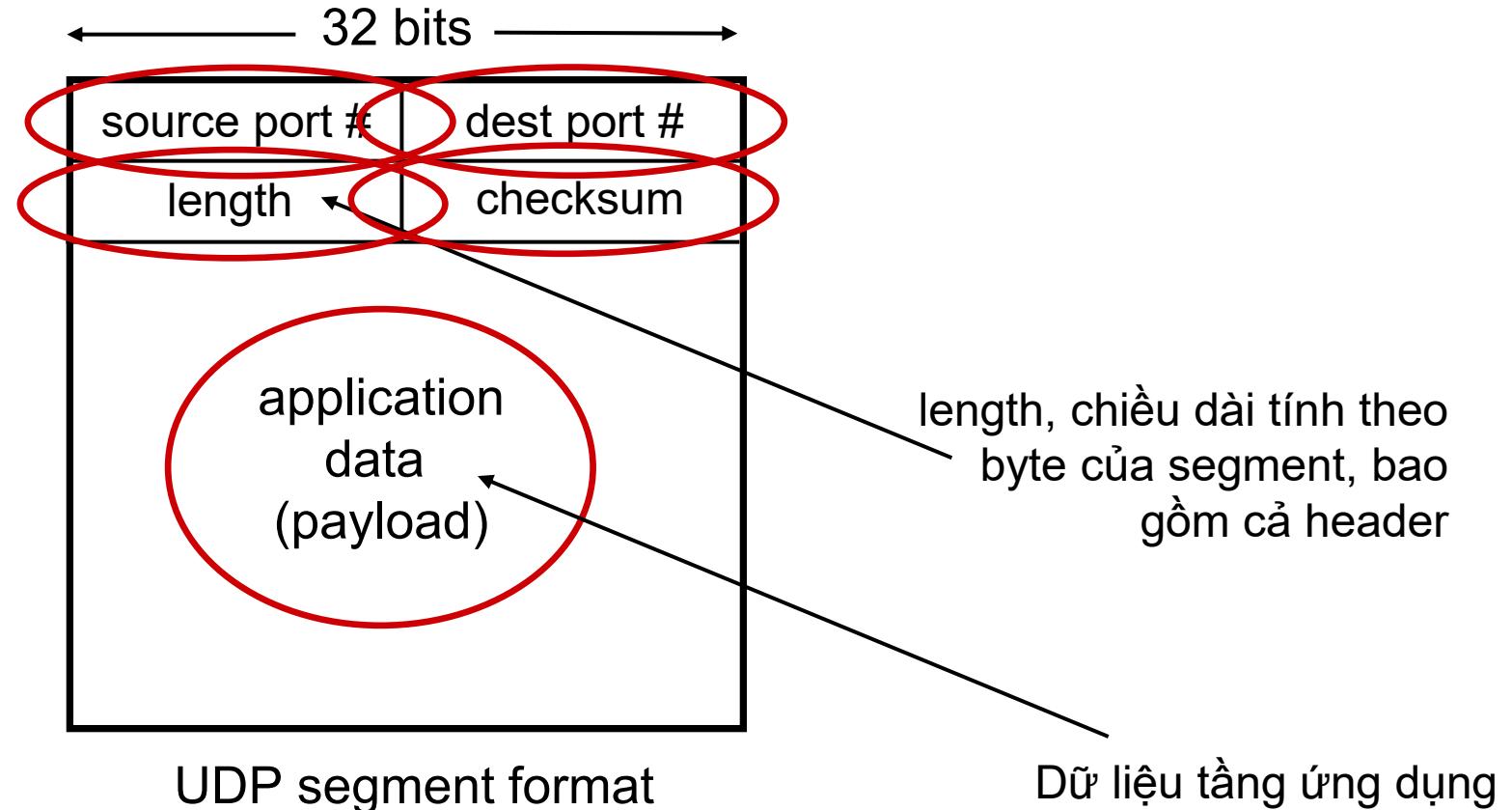
Bên nhận:

- Nhận segment từ tầng mạng
- Kiểm tra lỗi với giá trị checksum
- Bỏ header, lấy messages
- Chuyển message đến tầng ứng dụng qua socket

SNMP server



UDP header





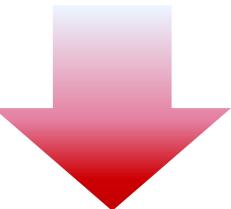
UDP checksum

Mục tiêu: phát hiện lỗi trong quá trình truyền (các bit bị lỗi, từ 0 thành 1 hoặc ngược lại)

Gửi:

1 st number	2 nd number	sum
------------------------	------------------------	-----

5	6	11
---	---	----



Nhận:

4	6
---	---

receiver-computed
checksum

11

sender-computed
checksum (as received)





Internet checksum

- **Mục tiêu:** phát hiện lỗi trong quá trình truyền (các bit bị lỗi, từ 0 thành 1 hoặc ngược lại)

bên gửi:

- Chia nội dung của segment (bao gồm các trường tiêu đề UDP) dưới dạng các chuỗi 16 bit
- **checksum:** Tính tổng bù 1 (one's complement sum)
- Đặt kết quả vào phần UDP checksum trong header

bên nhận:

- Tính checksum của segment nhận được
- So sánh kết quả tính được và nội dung phần checksum
 - Không bằng nhau – có lỗi
 - Bằng nhau – không phát hiện lỗi.
Tuy nhiên...



Internet checksum: ví dụ

- Ví dụ: cộng 2 chuỗi 16-bit

	1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
<hr/>																
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1
<hr/>																
sum	1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum	0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

Note: nếu kết quả là 17 bits, lấy bit cuối cùng bên trái cộng vào kết quả.



Internet checksum: điểm yếu!

- Ví dụ: cộng 2 chuỗi 16-bit

	1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0	0 1
	1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1	1 0
wraparound	1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1	
sum	1 0 1 1 1 0 1 1 1 0 1 1 1 1 0 0	
checksum	0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 1	

Khi có 2 bit bị thay đổi, checksum không thay đổi



Tổng kết: UDP

- Giao thức đơn giản:
 - Segment có thể bị mất, sai thứ tự
 - “Best-effort”: gửi và hy vọng có kết quả tốt nhất
- UDP vẫn có điểm cộng:
 - Không có quá trình thiết lập kết nối
 - Có checksum để kiểm tra lỗi
- Các tính năng bổ sung thì đưa lên tầng ứng dụng (e.g., HTTP/3)



Nội dung

- Các dịch vụ tầng vận chuyển
- Multiplexing and demultiplexing
- UDP
- Nguyên lý truyền tin cậy
- TCP
- TCP - Điều khiển tắc nghẽn
- Sự phát triển của các tính năng của tầng vận chuyển





Nguyên lý truyền tin cậy



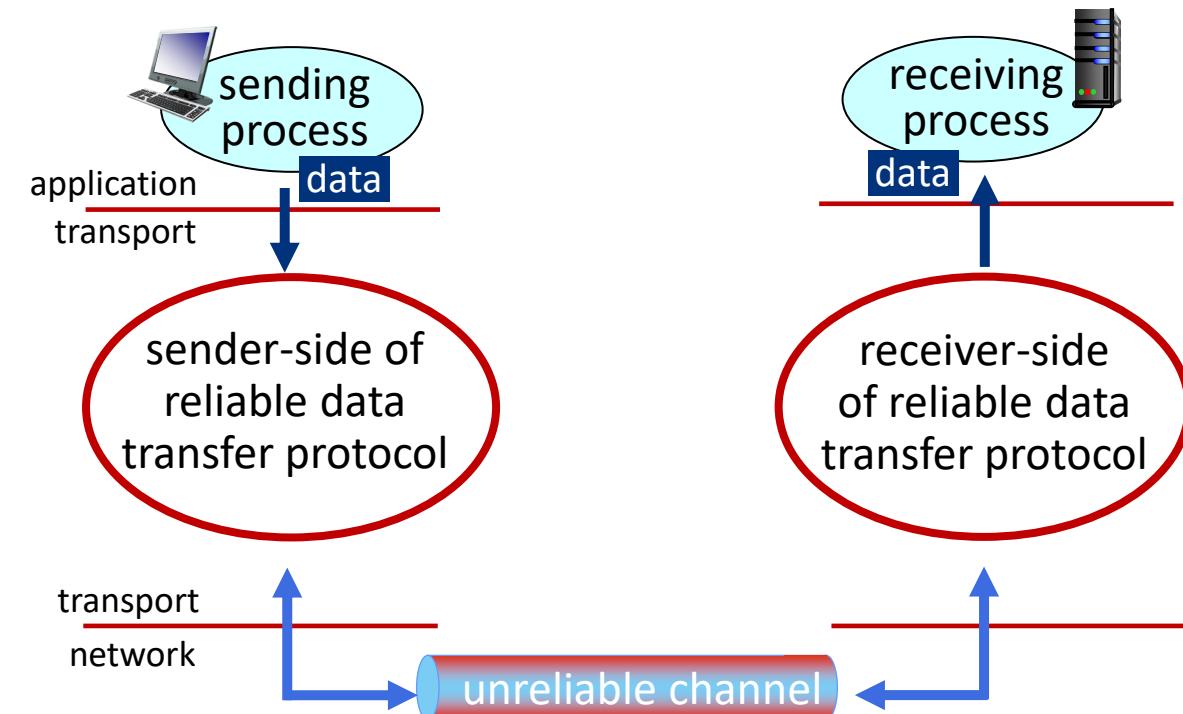
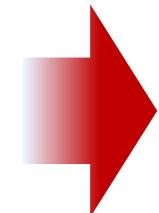
Tổng quát dịch vụ truyền tin cậy



Nguyên lý truyền tin cậy



Tổng quát dịch vụ truyền tin cậy

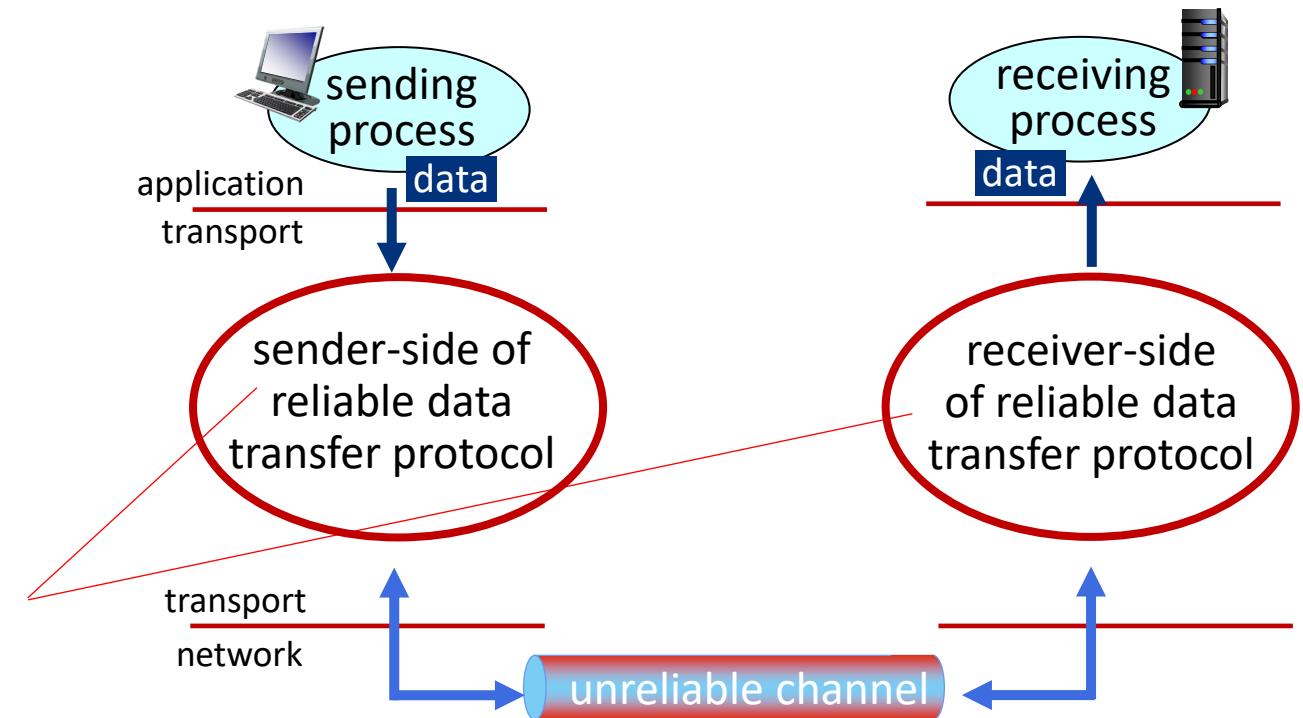


Triển khai cụ thể của dịch vụ truyền tin cậy

Nguyên lý truyền tin cậy



Độ phức tạp của giao thức
truyền dữ liệu đáng tin cậy sẽ
phụ thuộc (mạnh mẽ) vào các
đặc điểm của kênh không đáng
tin cậy (mất, hỏng, sắp xếp lại dữ
liệu?)

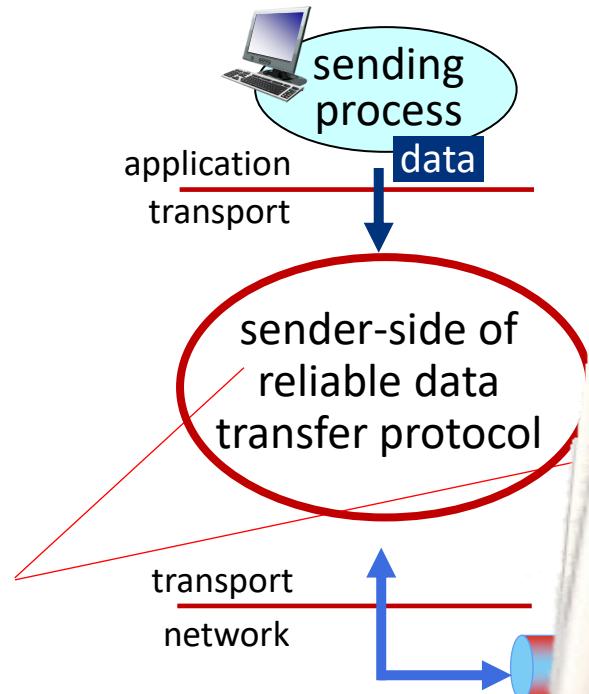


Triển khai cụ thể của dịch vụ truyền tin cậy

Nguyên lý truyền tin cậy



- Bên gửi, bên nhận không biết “trạng thái” của nhau, ví dụ: đã nhận được thông điệp chưa?
 - trừ khi được liên lạc qua các thông điệp

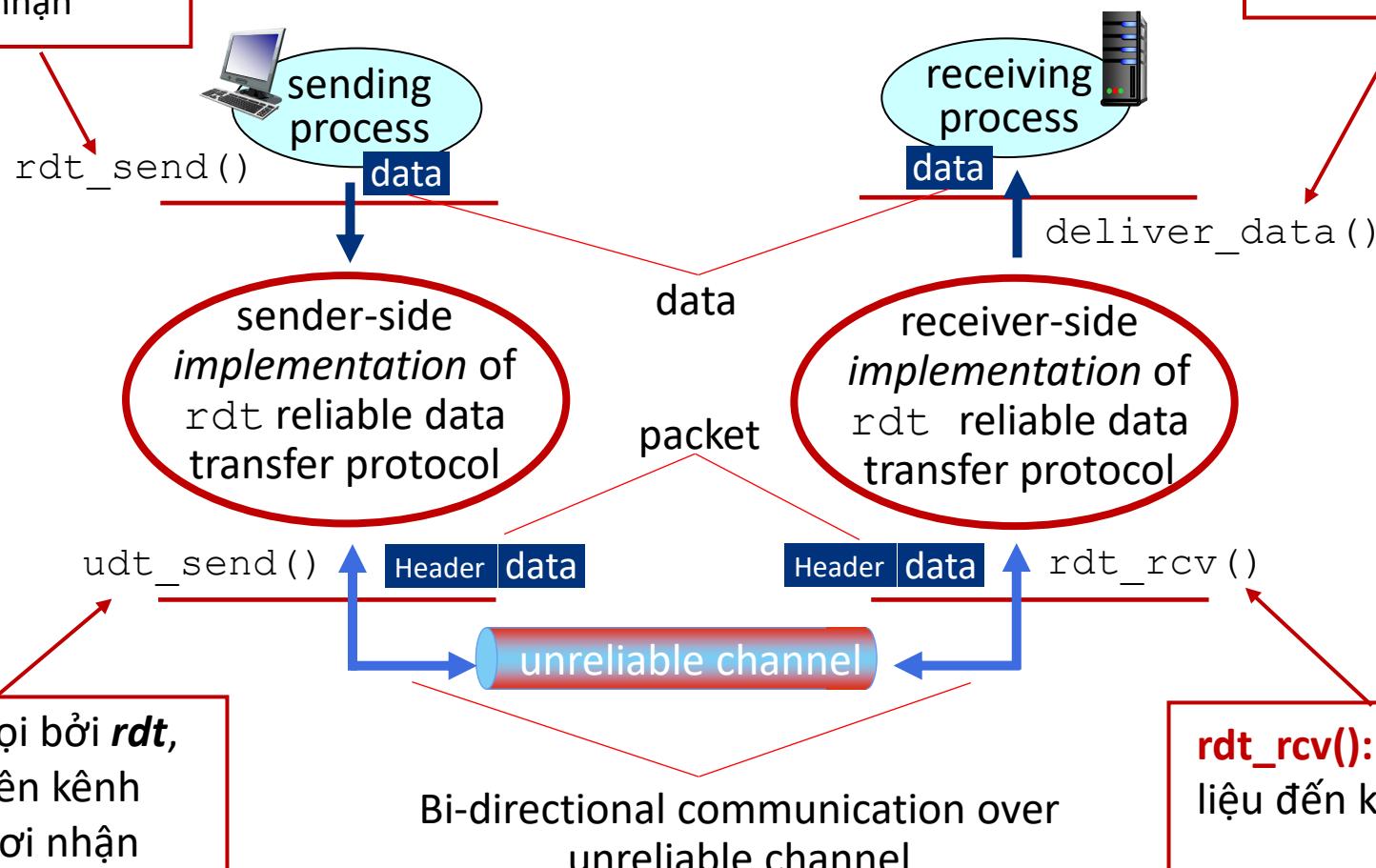


Triển khai cụ thể của dịch vụ truyền tin cậy



Reliable data transfer protocol (rdt): interfaces

rdt_send(): được gọi bởi tầng ứng dụng). Chuyển dữ liệu cần truyền đến tầng Ứng dụng bên nhận





Truyền dữ liệu tin cây: bắt đầu

Chúng ta sẽ:

- Từng bước phát triển truyền dữ liệu tin cây (rdt) bên phía người gửi và nhận
- Chỉ xem xét việc truyền dữ liệu theo một chiều (bên gửi đến bên nhận)
- Sử dụng finite state machines (FSM) để xác định bên gửi và nhận

Trạng thái: khi ở “trạng thái” này thì trạng thái kế tiếp được xác định duy nhất bởi sự kiện kế tiếp



Tổng quan

RDT1.0

RDT2.0

RDT2.1

RDT2.2

RDT3.0



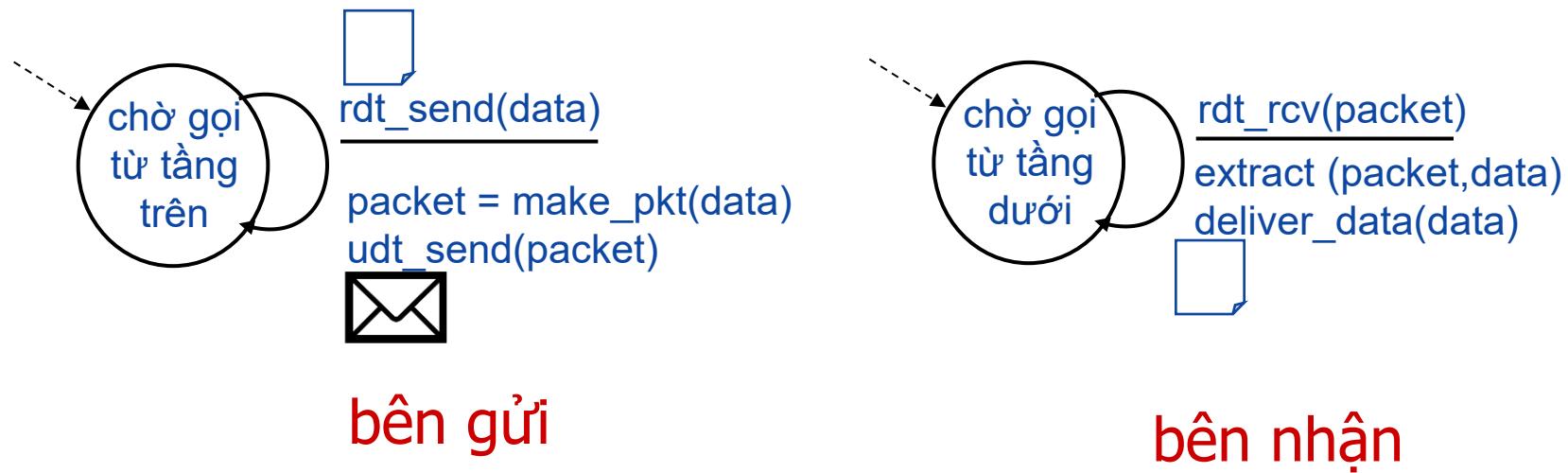


Truyền dữ liệu tin cậy

RDT1.0

- Kênh truyền tin cậy hoàn toàn

rdt1.0 - Truyền dữ liệu tin cậy hoàn toàn





Truyền dữ liệu tin cậy

RDT1.0

- Kênh truyền tin cậy hoàn toàn

RDT2.0

- Vấn đề: Kênh truyền có lỗi



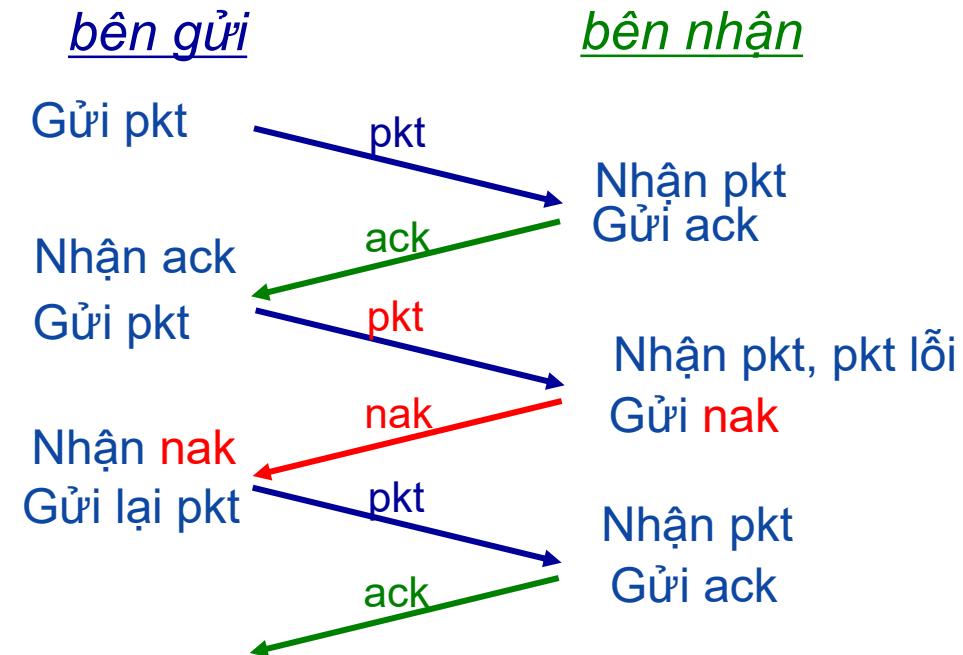
rdt2.0: Kênh truyền xảy ra lỗi

- Kênh cơ bản có thể đảo các bit trong packet
 - checksum để kiểm tra các lỗi
- *Câu hỏi:* làm sao khôi phục các lỗi:
 - *acknowledgements (ACKs)*: bên nhận báo bên gửi là gói tin đã được nhận thành công.
 - *negative acknowledgements (NAKs)*: bên nhận báo bên gửi là gói tin đã bị lỗi
 - Bên gửi sẽ gửi lại gói tin nếu nhận được NAK
- Cơ chế mới trong rdt2.0:
 - Phát hiện lỗi
 - Phản hồi từ bên nhận: ACK, NAK từ bên nhận gửi lại cho bên gửi

*Làm thế nào để con người phục hồi
“lỗi” trong cuộc trò chuyện?*

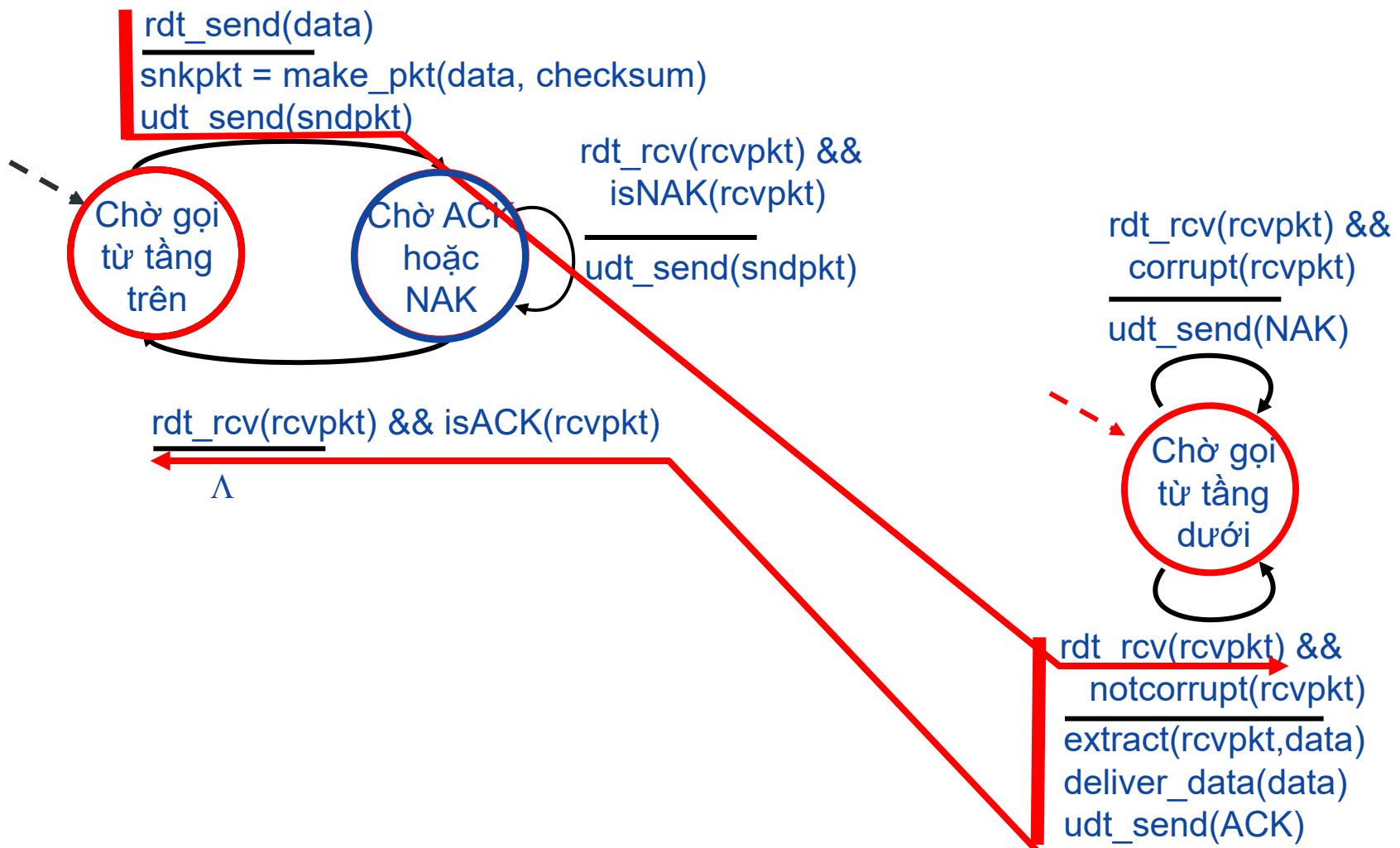


rdt2.0: Kênh truyền xảy ra lỗi



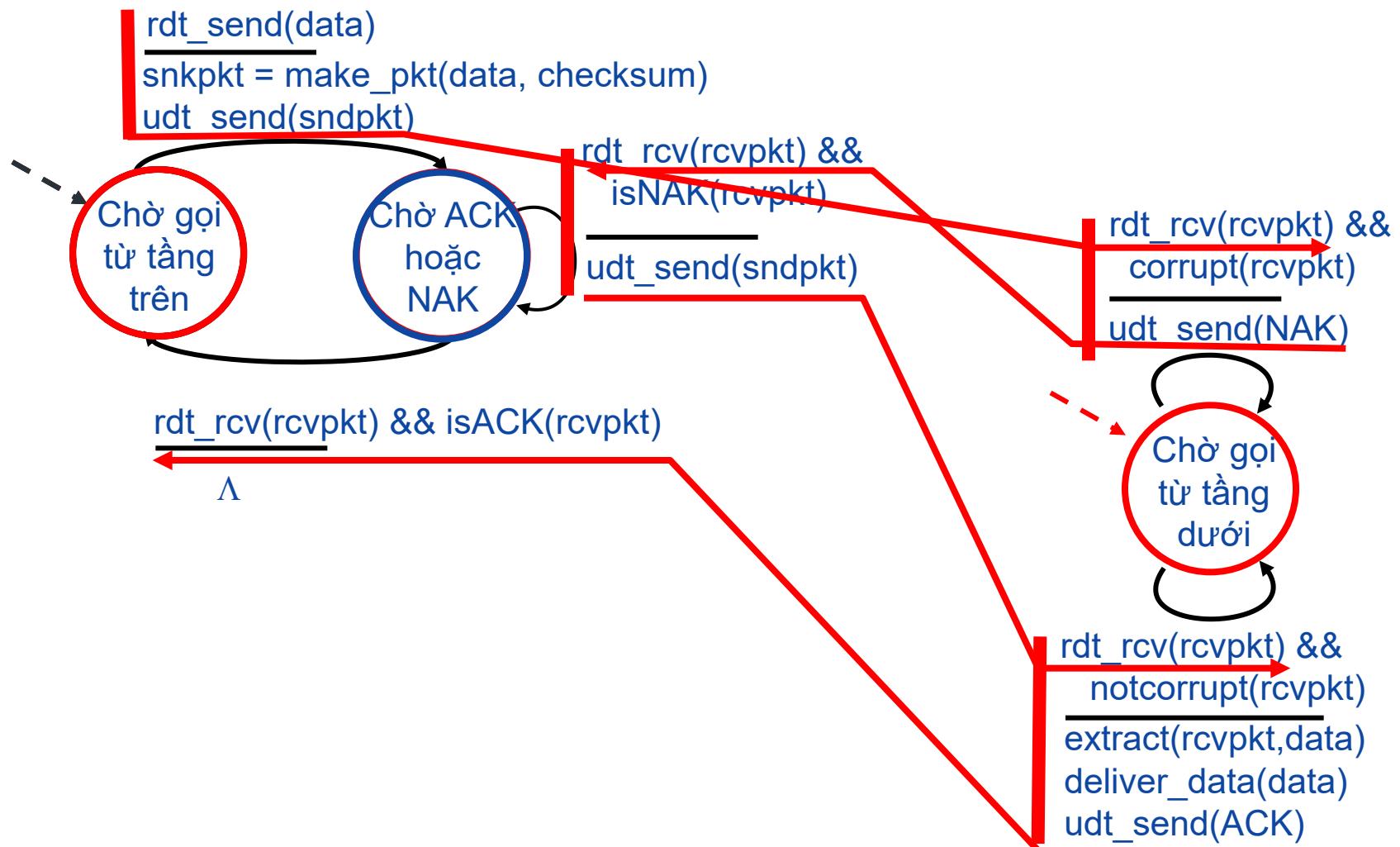


rdt2.0: hoạt động khi không lỗi





rdt2.0: hoạt động khi có lỗi





Truyền dữ liệu tin cây

RDT1.0

- Kênh truyền tin cây hoàn toàn

RDT2.0

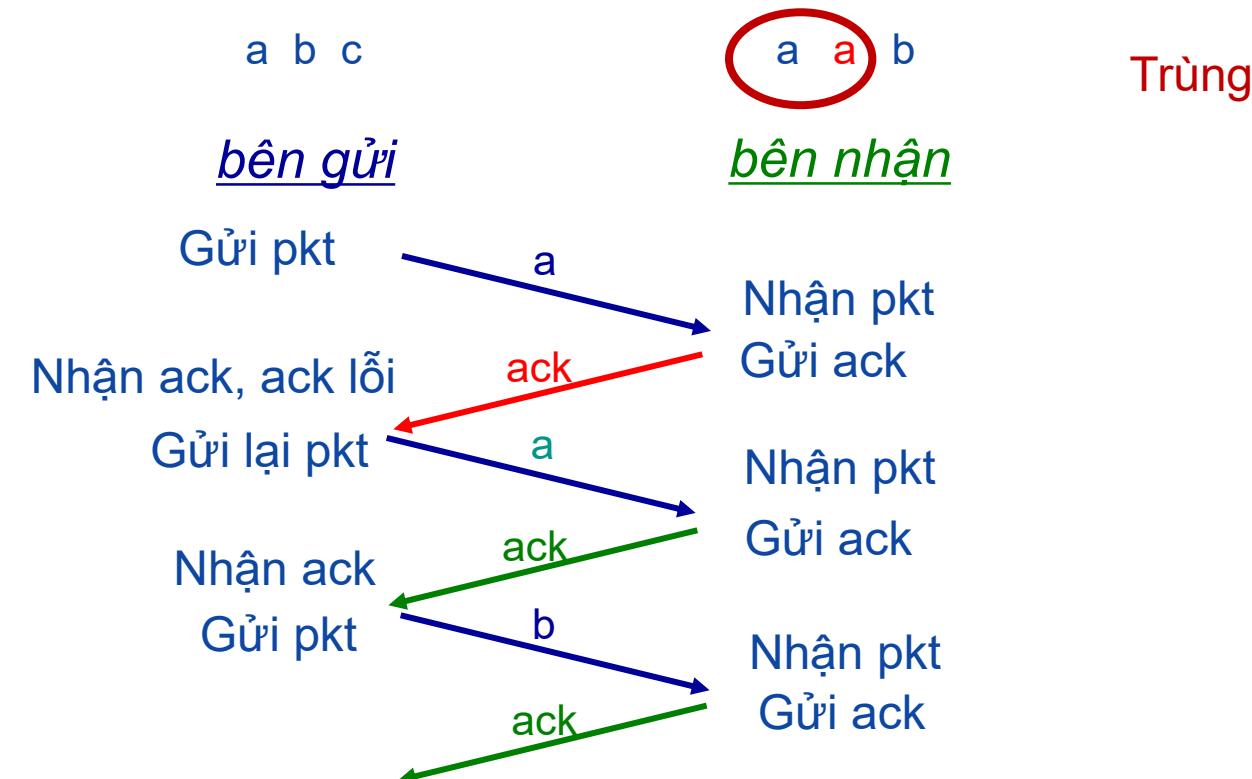
- Vấn đề: Kênh truyền có lỗi
- Giải quyết: ACK và NAK

RDT2.1

- Vấn đề: ACK/NAK lỗi

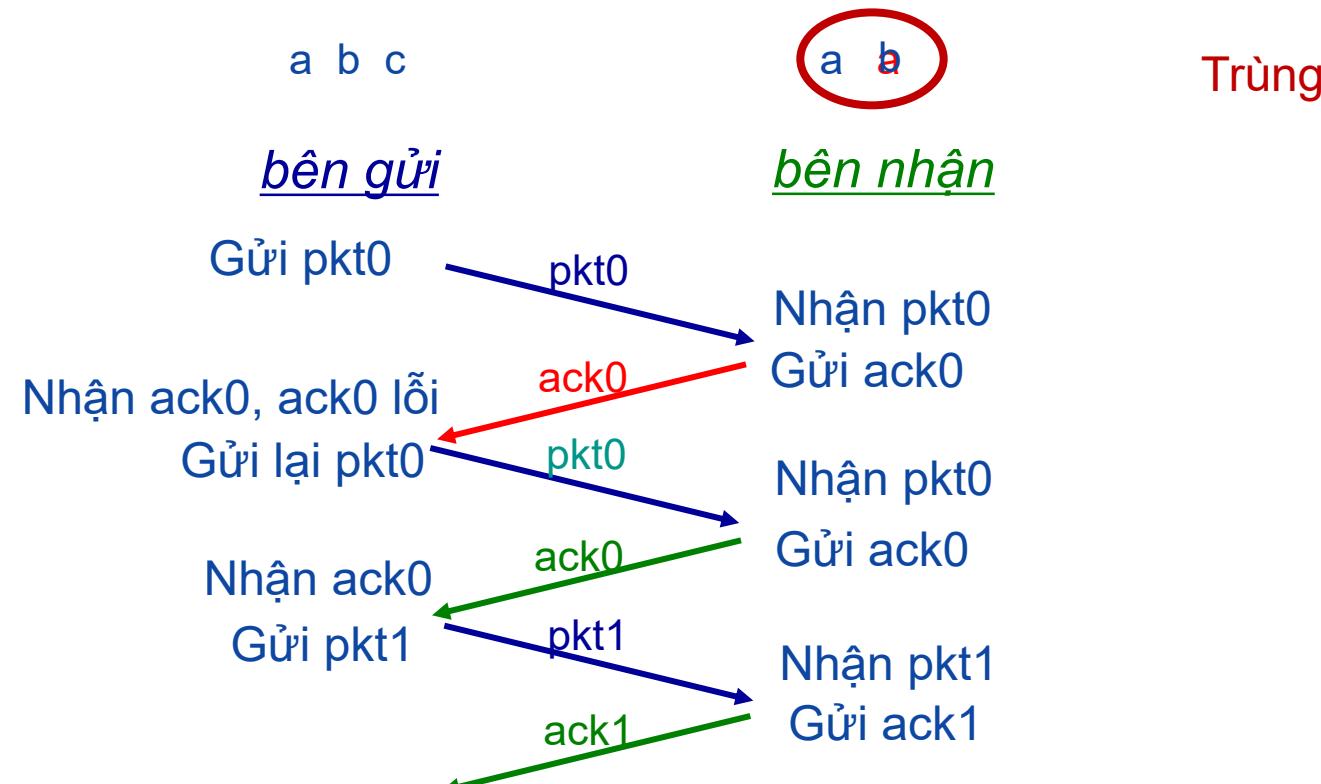


rdt2.0 có lỗi hỏng nghiêm trọng!



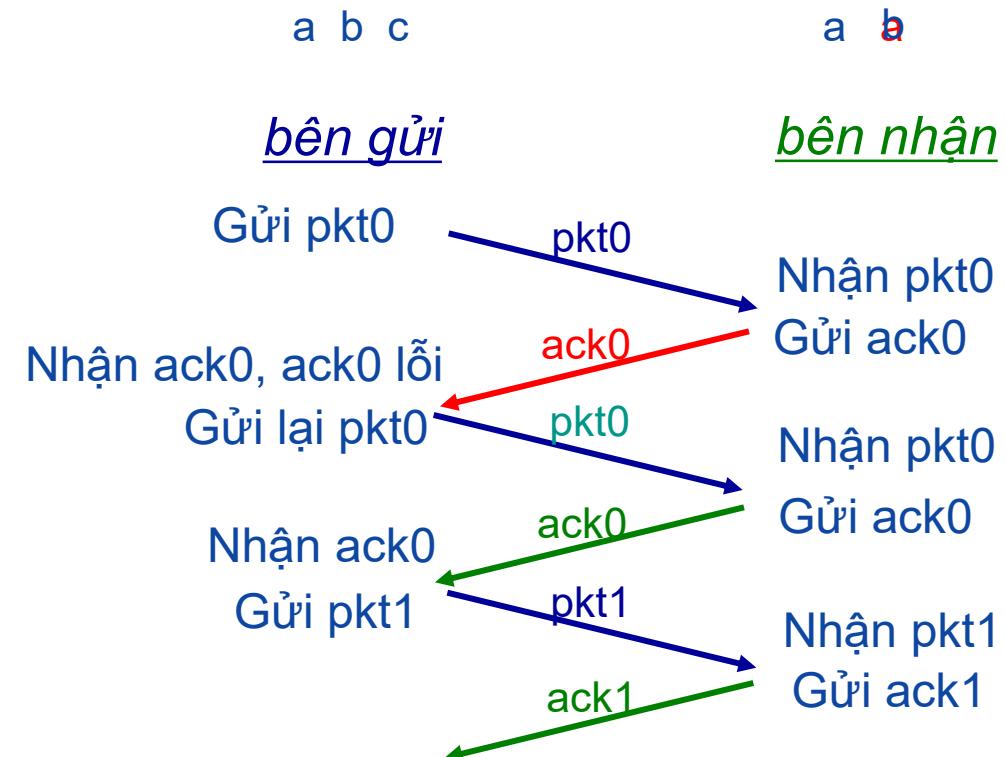
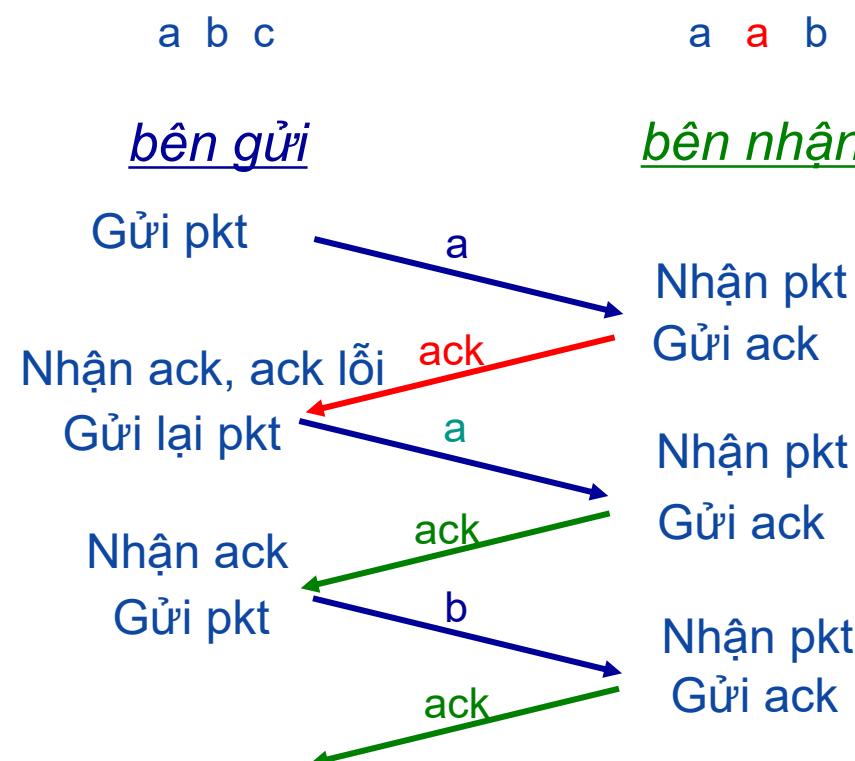


rdt2.1 bên gửi, xử lý các ACK/NAK bị hỏng





rdt2.1 bên gửi, xử lý các ACK/NAK bị hỏng



Truyền dữ liệu tin cậy



RDT1.0

- Kênh truyền tin cậy hoàn toàn

RDT2.0

- Vấn đề: Kênh truyền có lỗi
- Giải quyết: ACK và NAK

RDT2.1

- Vấn đề: ACK/NAK lỗi
- Giải quyết: Đánh số gói tin {0, 1}

RDT2.2

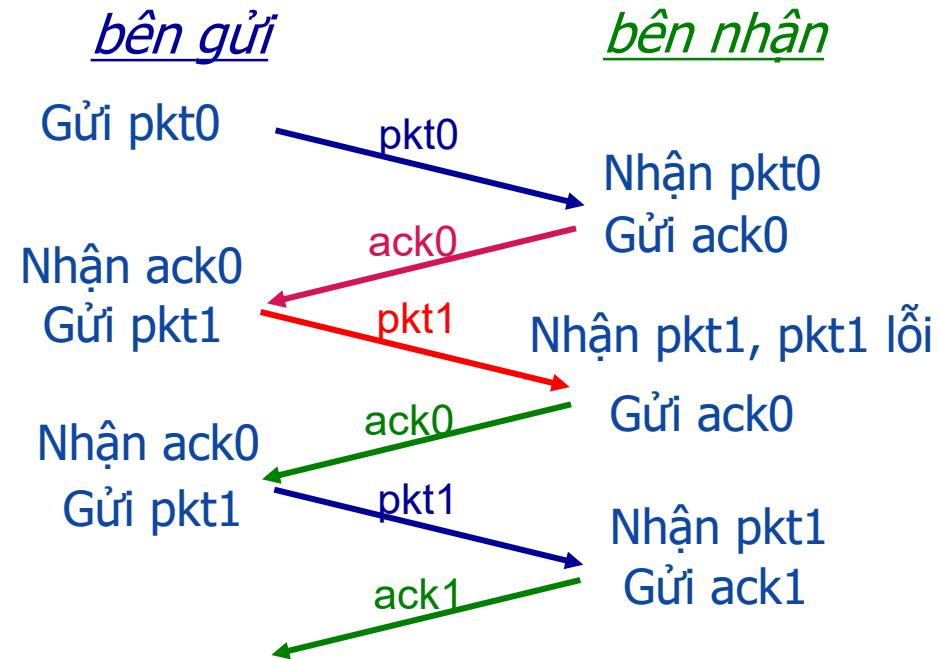
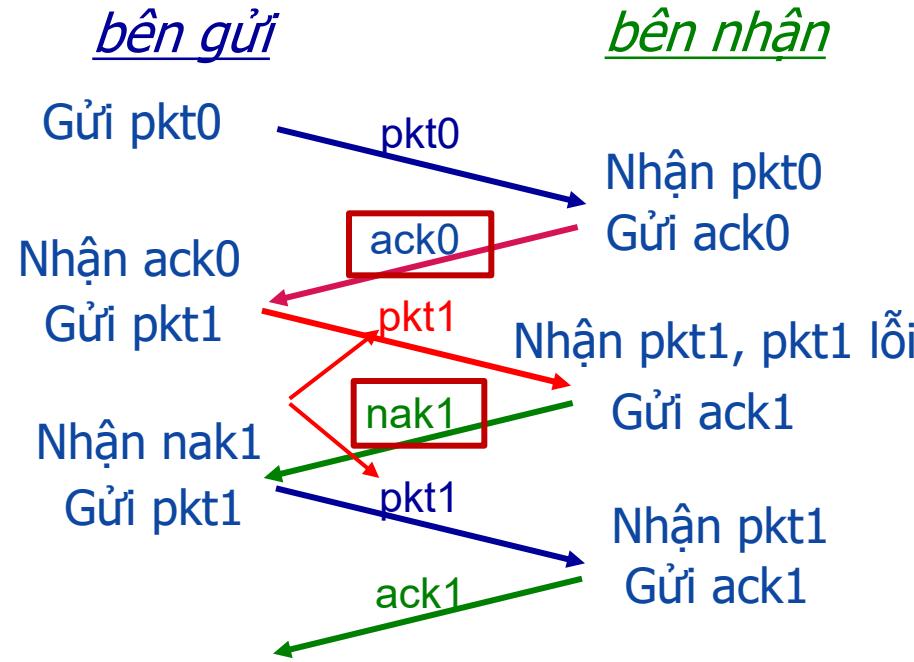


rdt2.2: một giao thức không cần NAK

- Chức năng giống như rdt2.1, chỉ dùng các ACK
- Thay cho NAK, bên nhận gửi ACK cho gói cuối cùng được nhận thành công
 - Bên nhận phải ghi rõ số thứ tự của gói vừa được ACK
- ACK bị trùng tại bên gửi dẫn tới kết quả giống như hành động của NAK: *truyền lại gói vừa rồi*



rdt2.2: một giao thức không cần NAK



Truyền dữ liệu tin cậy



RDT1.0

- Kênh truyền tin cậy hoàn toàn

RDT2.0

- Vấn đề: Kênh truyền có lỗi
- Giải quyết: ACK và NAK

RDT2.1

- Vấn đề: ACK/NAK lỗi
- Giải quyết: Đánh số gói tin {0,1}

RDT2.2

- Không cần NAK

RDT3.0

- Vấn đề: Mất gói tin



rdt3.0: các kênh với lỗi và mất mát

Cách tiếp cận: bên gửi chờ ACK trong khoảng thời gian “hợp lý”

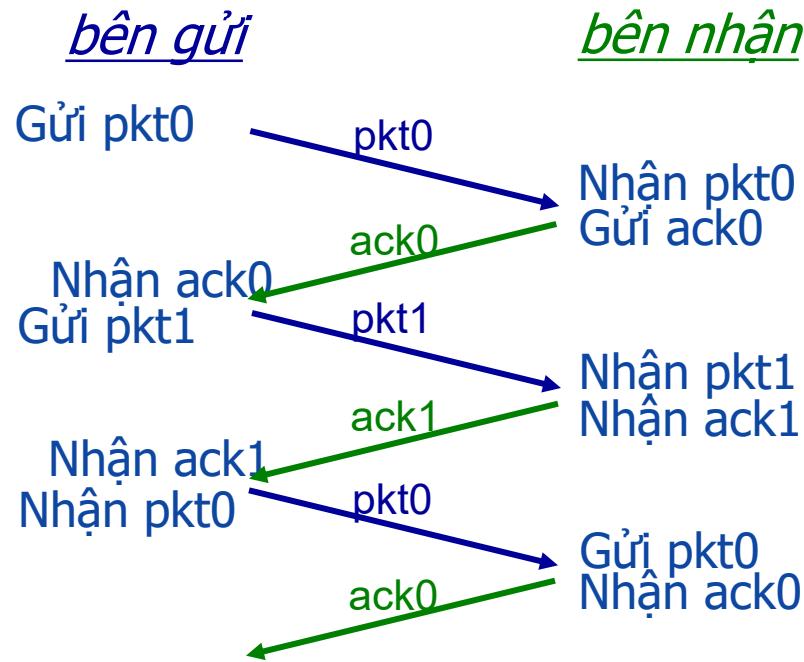
- Truyền lại nếu không nhận được ACK trong khoảng thời gian này
- Nếu gói (hoặc ACK) chỉ trễ (không mất):
 - Việc truyền lại sẽ gây trùng, nhưng số thứ tự đã xử lý trường hợp này
 - Bên nhận phải xác định số thứ tự của gói vừa gửi ACK
- Yêu cầu bộ định giờ (timer)

Giả định mới: kênh truyền cũng có thể làm mất gói (dữ liệu, các ACK)

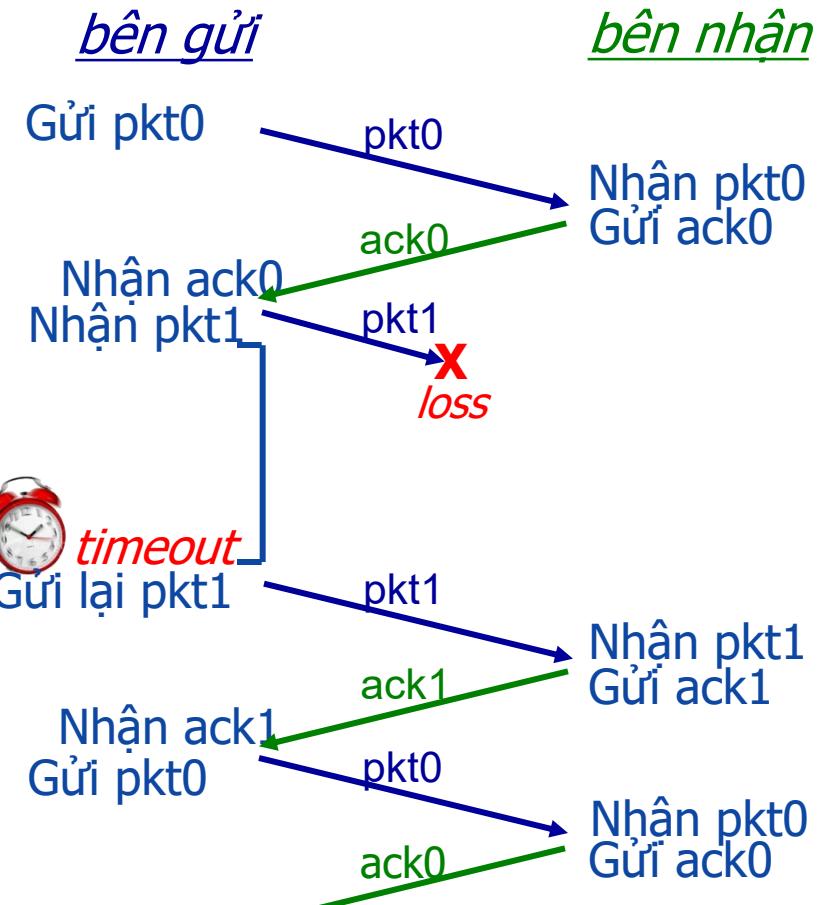
- checksum, số thứ tự, các ACK, việc truyền lại sẽ hỗ trợ... nhưng không đủ



Hoạt động của rdt3.0



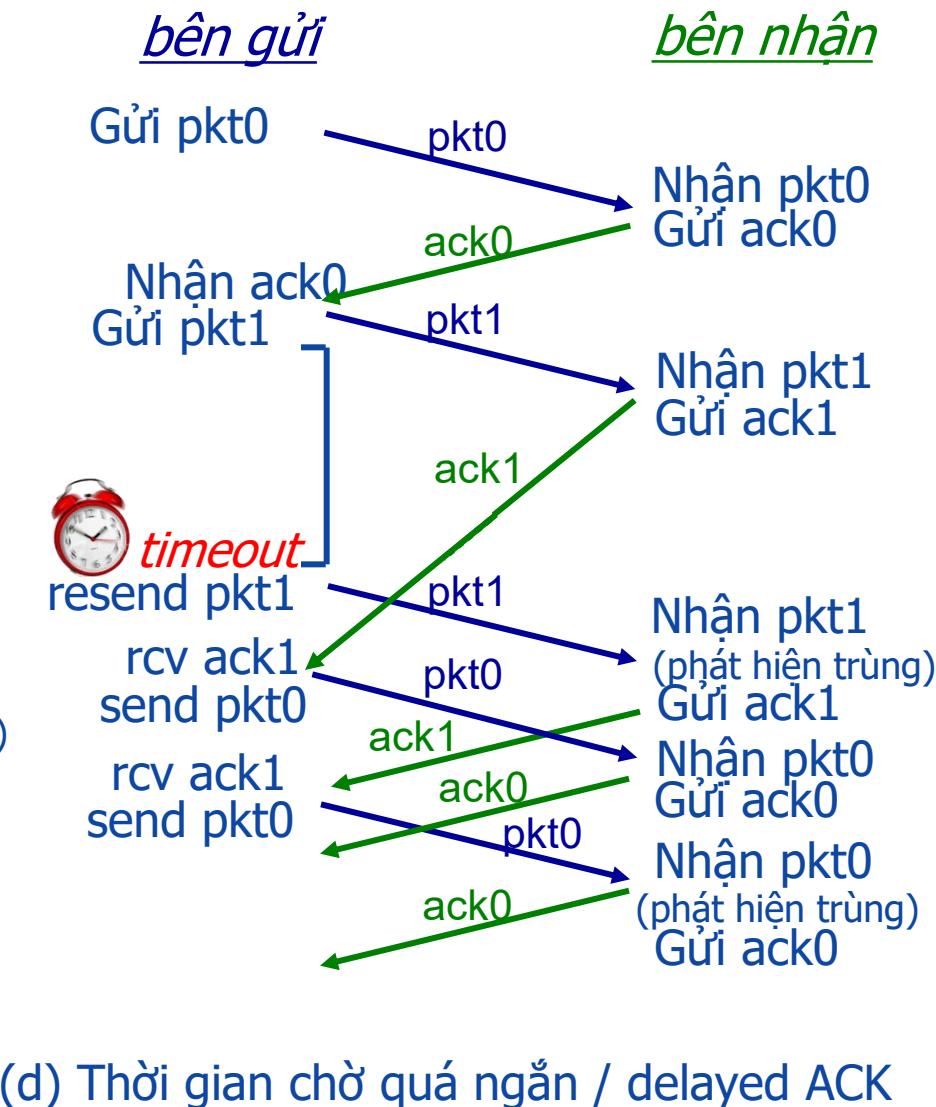
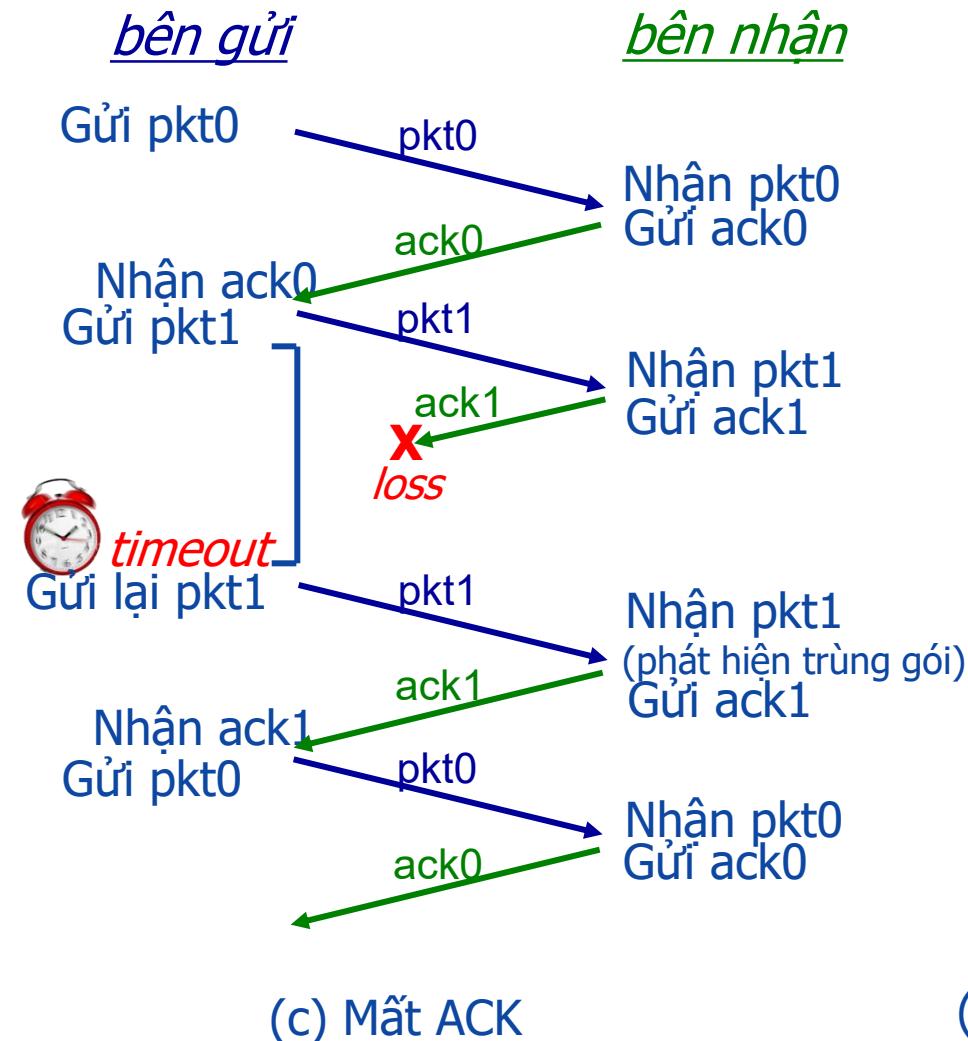
(a) Không mất mát



(b) Mất gói



Hoạt động của rdt3.0



Tổng kết về Truyền dữ liệu tin cậy



RDT1.0

- Kênh truyền tin cậy hoàn toàn

RDT2.0

- Vấn đề: Kênh truyền có lỗi
- Giải quyết: ACK và NAK

RDT2.1

- Vấn đề: ACK/NAK lỗi
- Giải quyết: Đánh số gói tin {0,1}

RDT2.2

- Không cần NAK

RDT3.0

- Vấn đề: Mất gói tin
- Giải quyết: Thêm biến timer



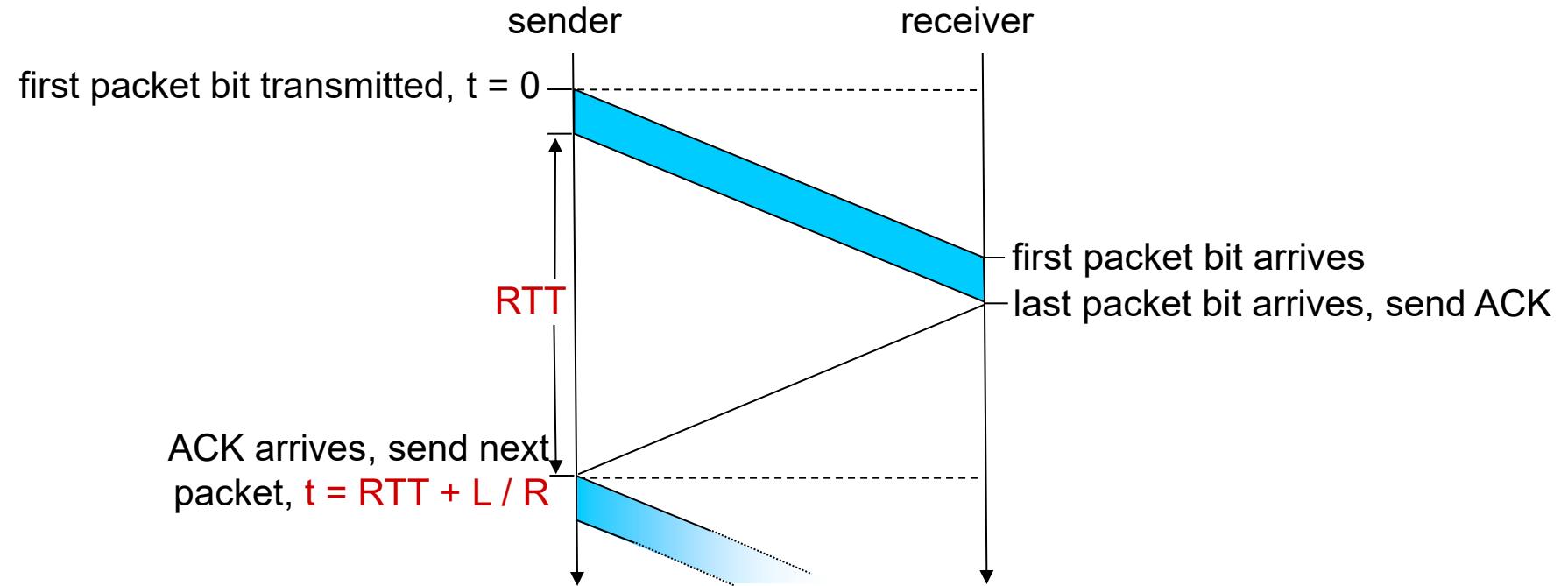
Hiệu suất của rdt3.0 (stop-and-wait)

- U_{sender} : *utilization* – tỉ lệ thời gian truyền của bên gửi
- Ví dụ: 1 Gbps link, truyền một gói tin 8000 bit trên kênh truyền có độ trễ lan truyền 15 ms
 - Thời gian truyền gói tin đến kênh truyền:

$$D_{\text{trans}} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}$$



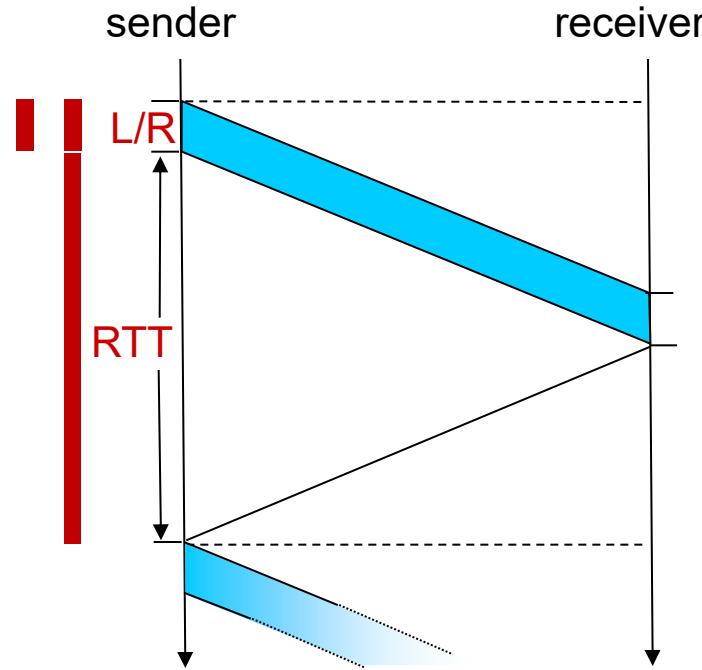
rdt3.0: hoạt động stop-and-wait





rdt3.0: hoạt động stop-and-wait

$$\begin{aligned}U_{\text{sender}} &= \frac{L / R}{RTT + L / R} \\&= \frac{.008}{30.008} \\&= 0.00027\end{aligned}$$



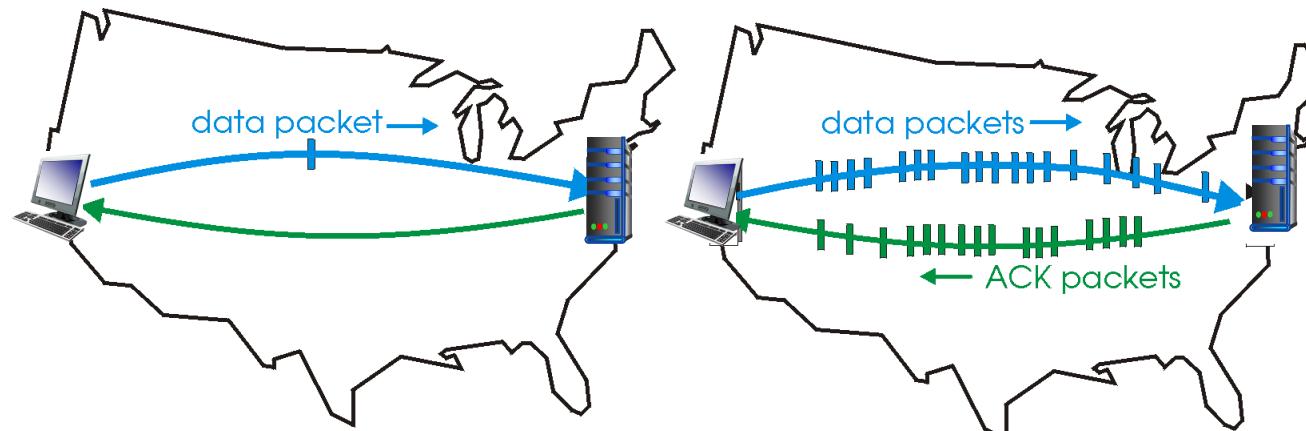
- Hiệu suất giao thức rdt 3.0 kém
- Giao thức giới hạn hiệu suất của cơ sở hạ tầng cơ bản (kênh truyền)

rdt3.0: hoạt động giao thức “pipelined”



pipelining: bên gửi gửi nhiều packet cùng một lúc mà không cần chờ ACK

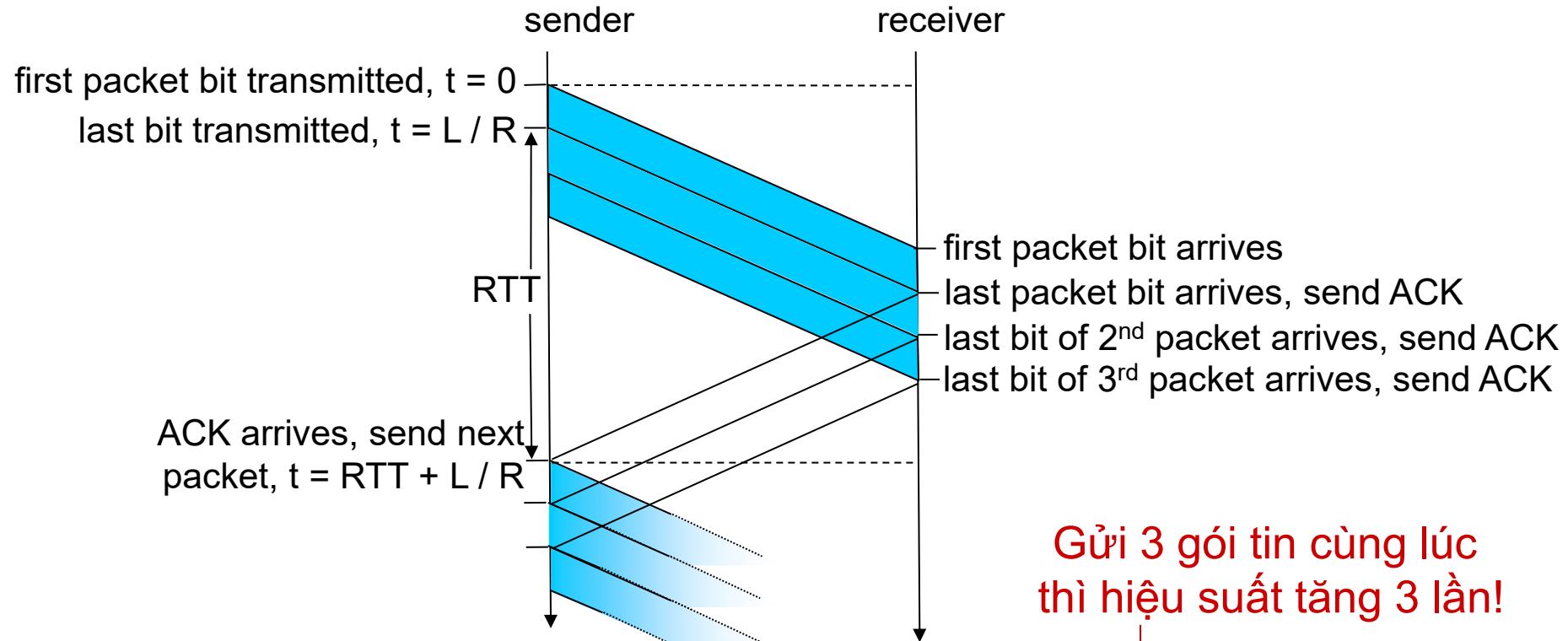
- Dãy số thứ tự phải được tăng lên
- Đưa vào bộ đếm tại bên gửi hoặc bên nhận



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

(b) a pipelined protocol in operation

Pipelining: tăng hiệu suất



$$U_{\text{sender}} = \frac{3L/R}{RTT + L/R} = \frac{.0024}{30.008}$$

Gửi 3 gói tin cùng lúc
thì hiệu suất tăng 3 lần!

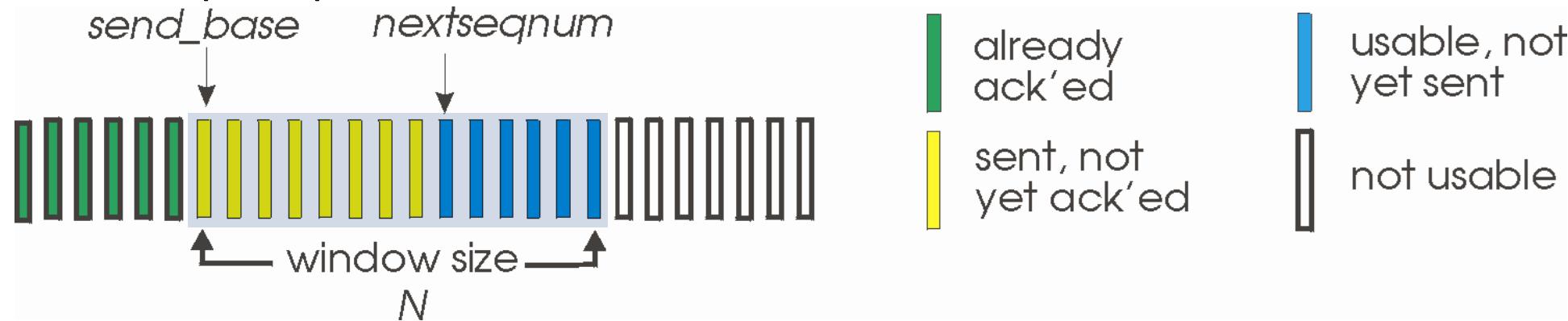
$$= 0.00081$$



Go-Back-N: bên gửi

- Bên gửi: gửi tối đa N packets liên tiếp mà không cần chờ ACK (“window”)

- k-bit seq # in pkt header



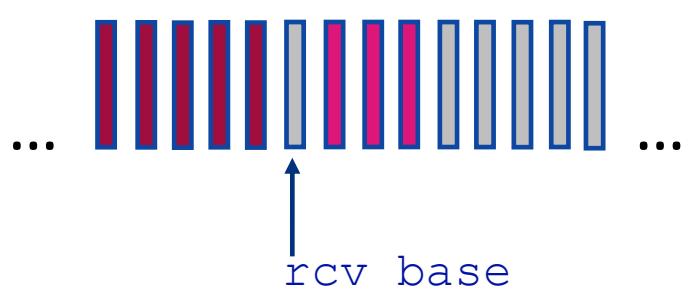
- Khi nhận ACK(n) => ACK cho tất cả các packet đã gửi tính tới gói tin có seq là n (*cumulative ACK* – ACK tích lũy)
- Tính timer cho packet gửi sớm nhất nhưng chưa được ACK
- $timeout(n)$: gửi lại tất cả các packet trong “window” từ vị trí (n)



Go-Back-N: bên nhận

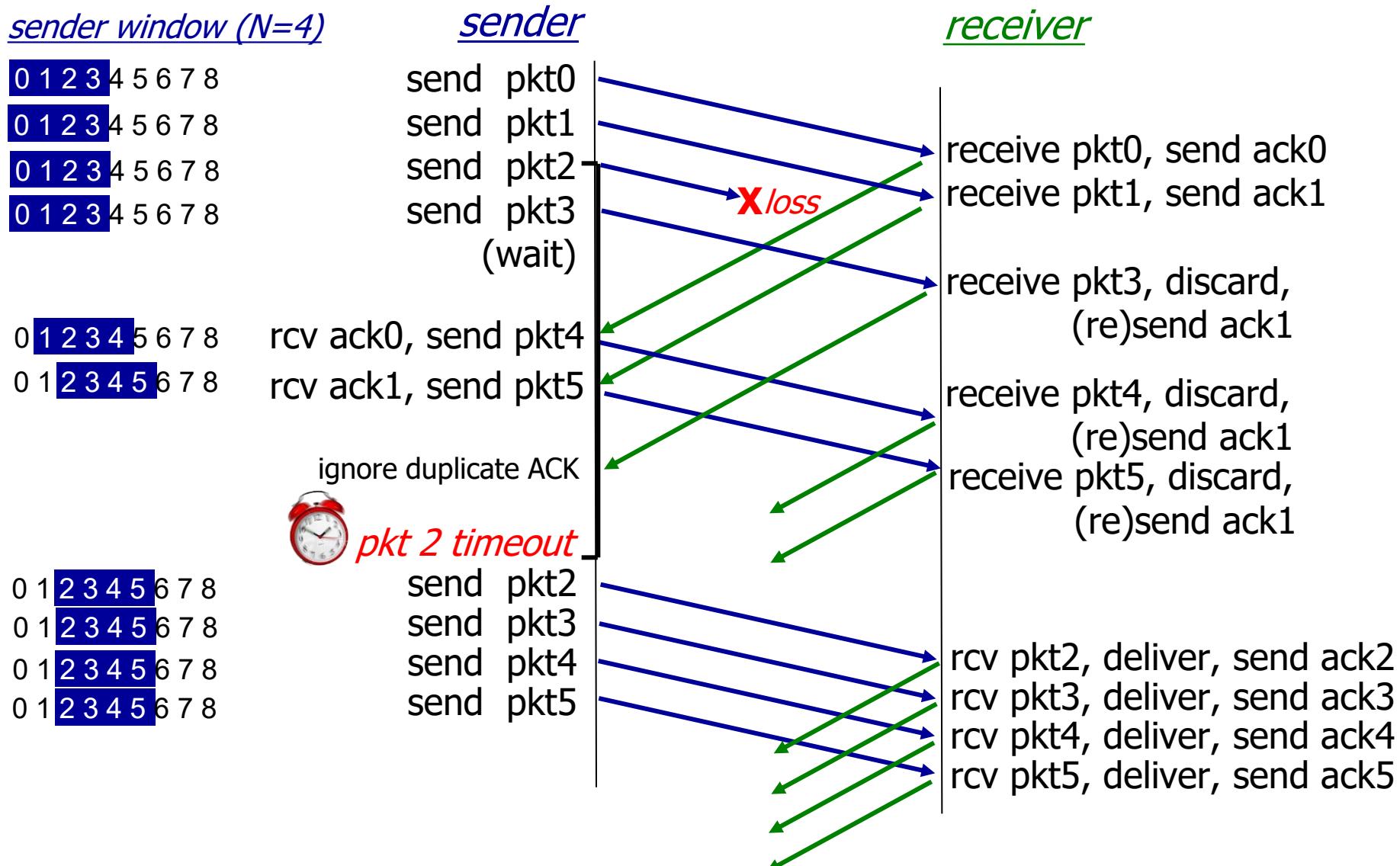
- Nhận packet đúng thứ tự: gửi ACK tương ứng với packet đúng thứ tự có STT cao nhất
- Nhận packet không đúng thứ tự:
 - Có thể hủy hoặc đưa vào buffer: tùy vào cách triển khai
 - Gửi lại ACK tương ứng với packet đúng thứ tự có số TT cao nhất

Receiver view of sequence number space:



	received and ACKed
	Out-of-order: received but not ACKed
	Not received

Go-Back-N: các hành động





Go-Back-N in action

- Tham khảo ví dụ về Go-Back-N:

Minh họa Go-Back-N

- Thủ nghiệm và quan sát các kịch bản với window size = 5, gửi lần lượt các gói 0, 1, 2, 3, 4
 - TH1: Các gói tin gửi bình thường và không mất mát
 - TH2: Packet 2 bị mất
 - TH3: ACK2 bị mất



Selective repeat

- Bên nhận gửi ACK cho từng packet nhận thành công (cơ chế buffer tại bên gửi)
- Bên gửi bật cơ chế timeout/gửi lại các packet không được ACK
- Cơ chế window tại bên gửi
 - Cho phép gửi liên tiếp N packet có số thứ tự tăng dần
 - Giới hạn số packet được gửi nhưng không được ACKs



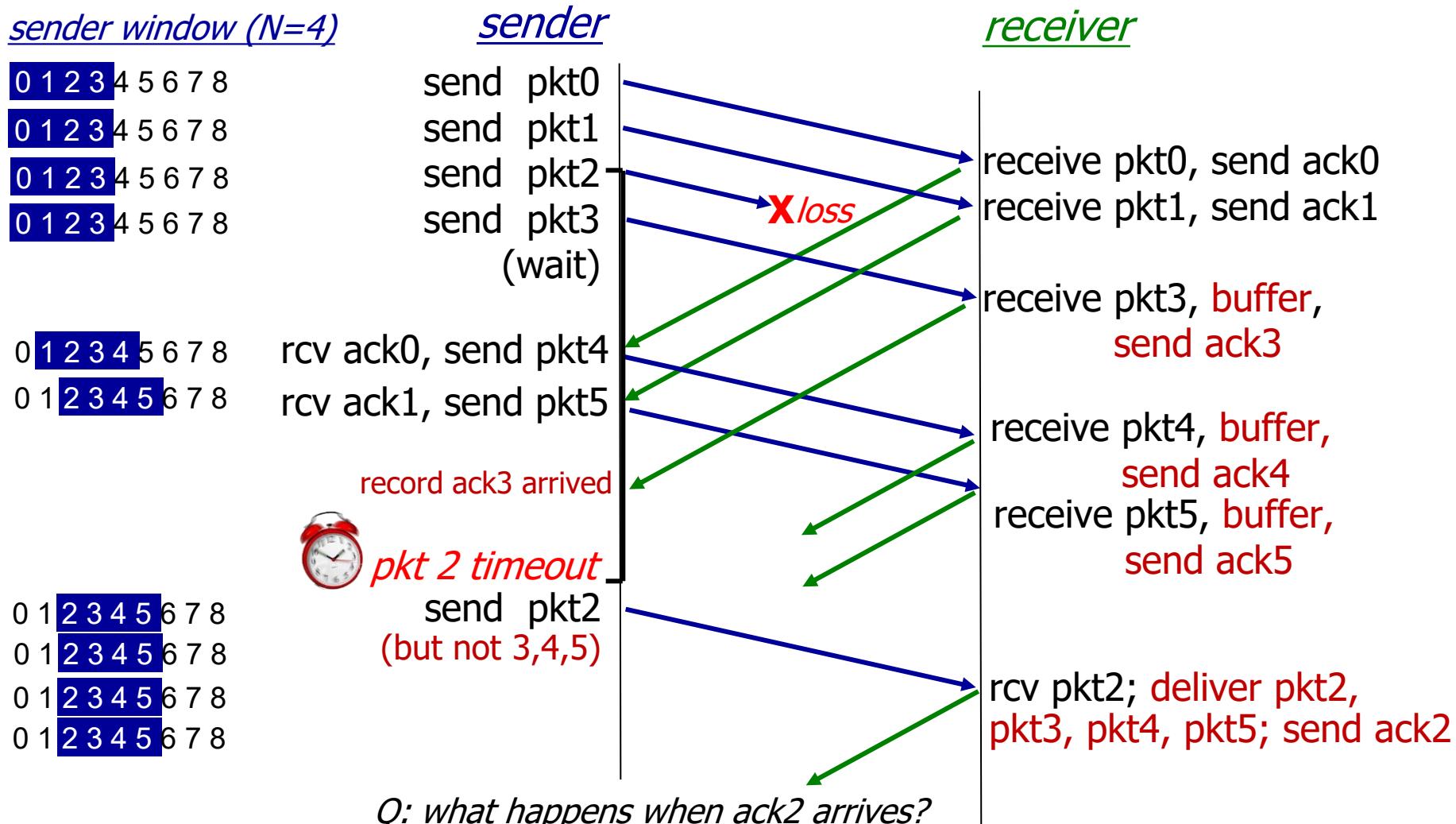
Selective repeat

- Tham khảo ví dụ về Selective repeat

Minh họa về Selective Repeat

- Thủ nghiệm và quan sát các kịch bản với window size = 5, gửi lần lượt các gói 0, 1, 2, 3, 4
 - TH1: Các gói tin gửi bình thường và không mất mát
 - TH2: Packet 2 bị mất
 - TH3: ACK2 bị mất

Selective Repeat: các hành động





Nội dung

- Các dịch vụ tầng vận chuyển
- Multiplexing and demultiplexing
- UDP
- Nguyên lý truyền tin cậy
- TCP
- TCP - Điều khiển tắc nghẽn
- Sự phát triển của các tính năng của tầng vận chuyển



TCP: tổng quan

RFCs: 793, 1122, 2018, 5681, 7323



- point-to-point:
 - 1 bên gửi, 1 bên nhận
- Tin cậy, dữ liệu đúng thứ tự theo byte
- “full duplex data”:
 - Luồng dữ liệu 2 chiều trong cùng một kết nối
 - MSS: maximum segment size (độ dài tối đa của 1 segment)
- ACKs tích lũy
- Cơ chế pipelining:
 - Điều khiển tắc nghẽn và điều khiển luồng với “window size”
- Hướng kết nối:
 - handshaking (trao đổi các thông điệp thiết lập và đóng kết nối)
- “flow controlled”:
 - Bên gửi không gửi nhiều làm quá tải bên nhận



Cấu trúc của TCP segment

ACK: seq # of next expected byte; A bit: this is an ACK

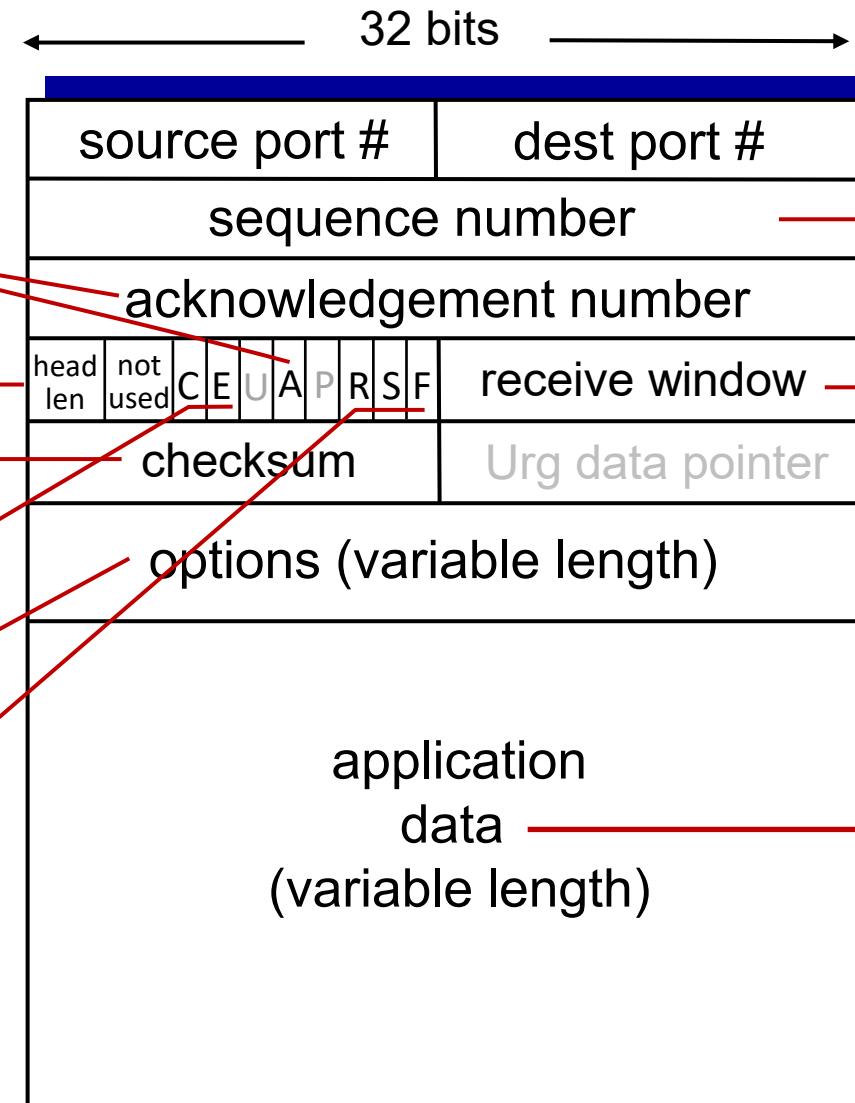
length (of TCP header)

Internet checksum

C, E: congestion notification

TCP options

RST, SYN, FIN: connection management



segment seq #: counting bytes of data into bytearray (not segments!)

flow control: # bytes receiver willing to accept

data sent by application into TCP socket



TCP: Reliable data transfer

- Truyền theo thứ tự: Sequence number, ACK
- Truyền lại:
 - Segment lỗi: ACK
 - Segment bị mất: timeout



TCP sequence numbers, ACKs: ví dụ

- Thông điệp tầng ứng dụng: 9000 bytes (byte #0 -> byte #8999)
- Mỗi segment chứa tối đa 2000 bytes
- => 5 segment kích thước 2000 byte



Sender - Sequence number

Seq = **0**, length = 2000B

You need byte
from 2000, I'll
send you from
#2000

S = **2000**, length = 2000B

S = **4000**, length = 2000B

S = **6000**, length = 2000B

S = **8000**, length = 1000B

Receiver ACK

ACK = **2000**

ACK = **4000**

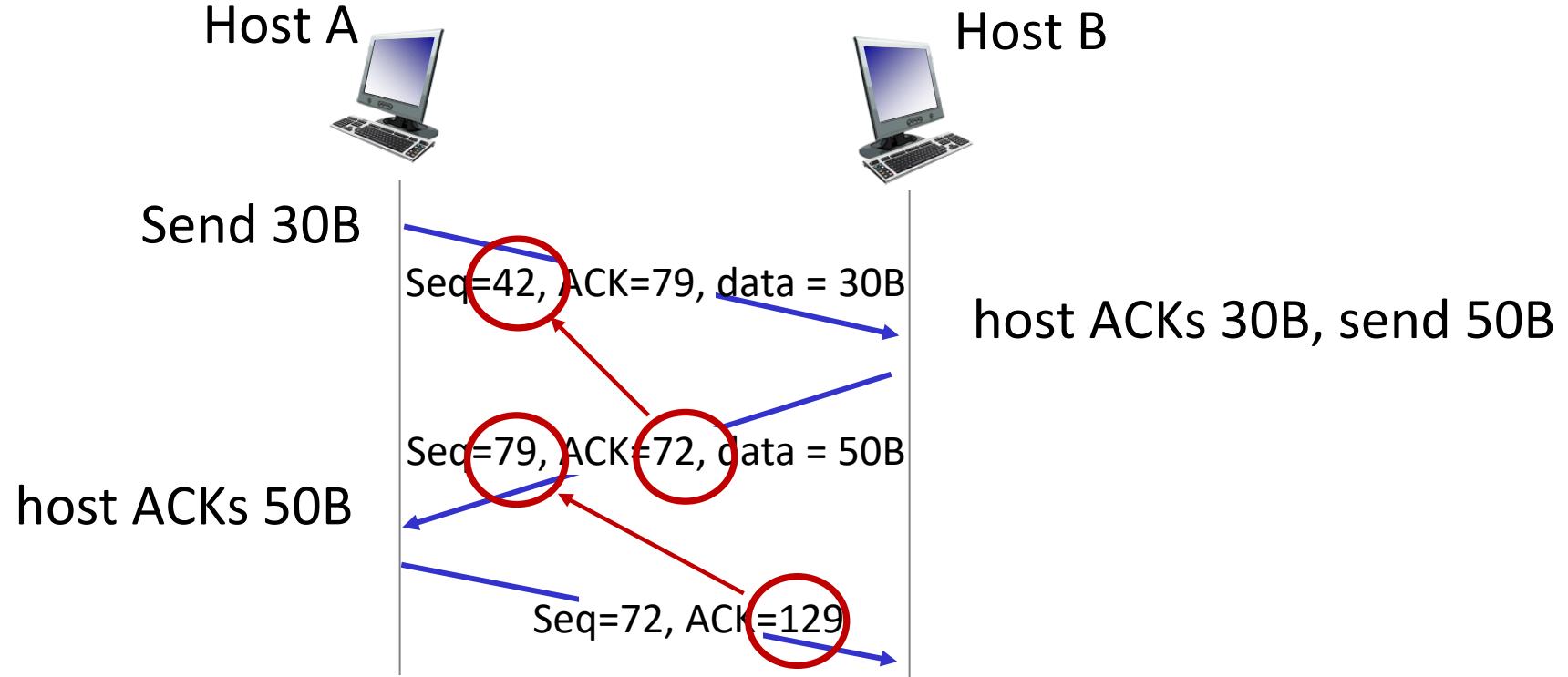
ACK = **6000**

ACK = **8000**

ACK = **9000**

I receive until #1999, I
want byte from #2000

TCP sequence numbers, ACKs





TCP Sender (tinh giản)

Sự kiện: Nhận dữ liệu từ tầng ứng dụng

- Tạo segment với STT seq #
- seq #: STT của byte đầu tiên trong segment
- Bắt đầu tính thời gian với segment cũ nhất

Sự kiện: *timeout*

- Gửi lại segment bị timeout
- Bắt đầu tính lại thời gian

Sự kiện: nhận được ACK

- Nếu ACK xác nhận cho các segment chưa được ACK
 - Cập nhật trạng thái
 - Tính lại thời gian với các segment chưa được ACK

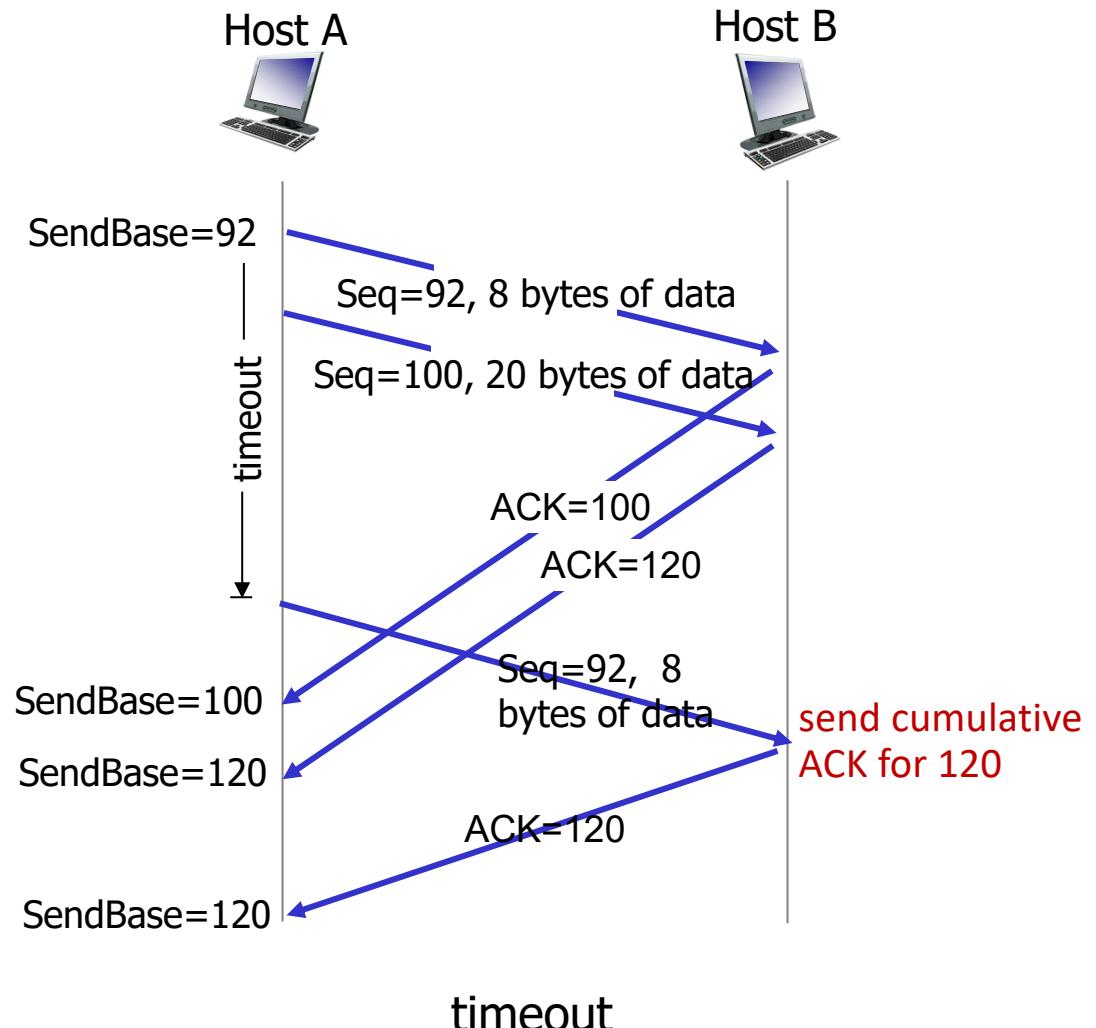
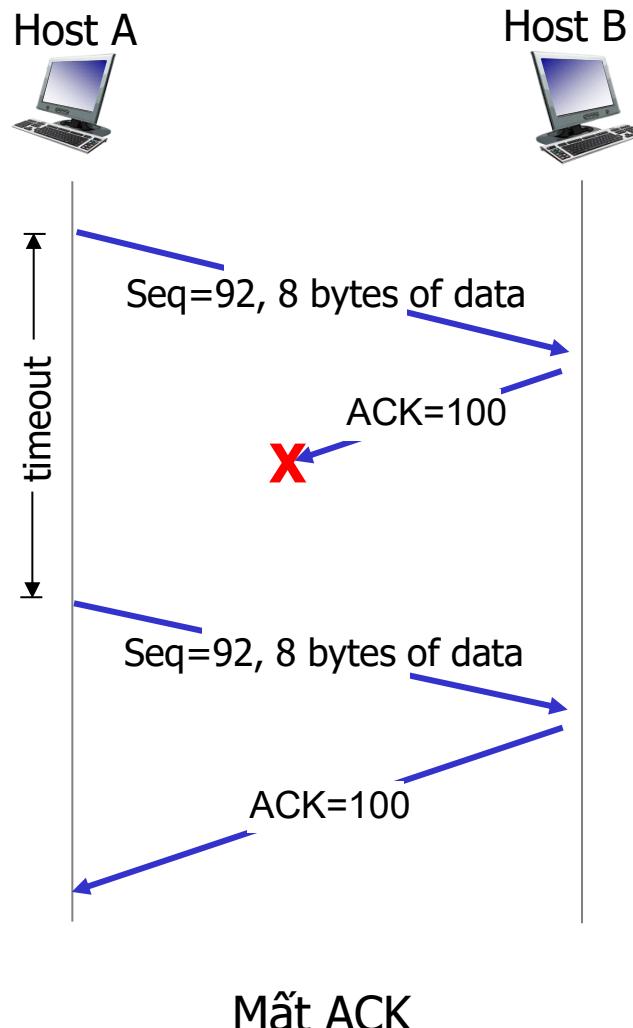


TCP Receiver: ACK generation [RFC 5681]

Sự kiện tại bên nhận	Hành động
Nhận được segment đúng với STT đang chờ. Tất cả segment trước đó đã được ACK.	Chờ segment kế tiếp trong 500ms, nếu không nhận được thì sẽ ACK.
Nhận được segment đúng với STT đang chờ, một segment chưa được ACK.	Gửi ACK tích lũy để ACK cho 2 segment.
Nhận được segment không đúng thứ tự (STT cao hơn).	Gửi ACK trùng ngay lập tức, để chỉ cho bên gửi biết đang chờ segment nào.
Nhận được segment trong khoảng bị trống (giữa STT đang chờ và STT nhận được trước đó).	ACK ngay lập tức cho segment có thứ tự thấp nhất trong khoảng bị trống

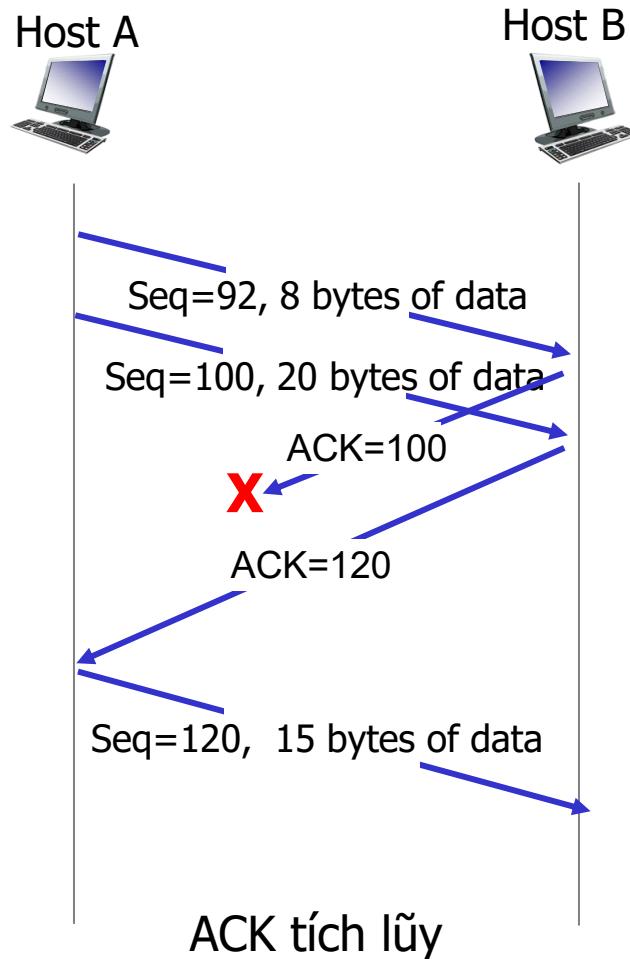


TCP: các trường hợp truyền lại





TCP: các trường hợp truyền lại





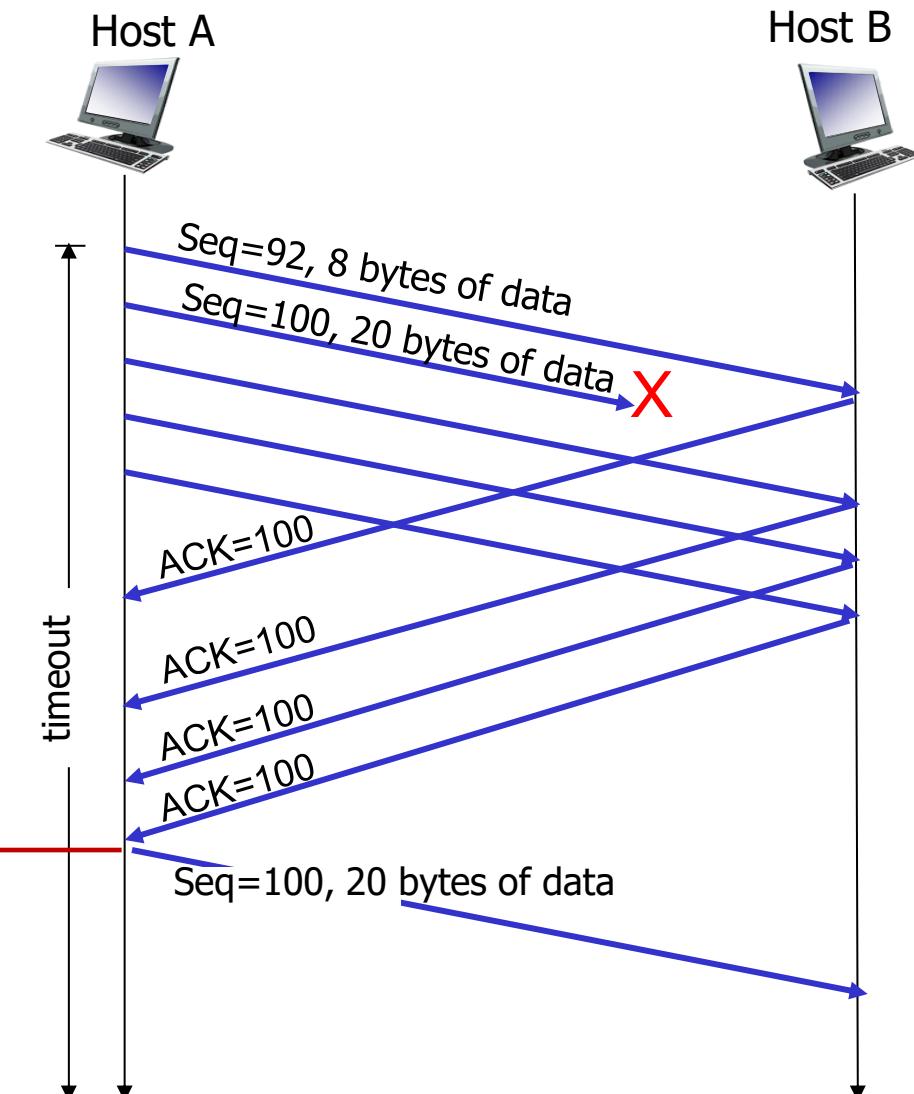
TCP: truyền lại nhanh

TCP: truyền lại nhanh

Khi bên gửi nhận được 3 ACK yêu cầu cùng segment, bên gửi gửi lại segment được yêu cầu mà không cần chờ timeout.



Việc nhận 3 ACK trùng lặp cho biết đã nhận được 3 segment sau một segment bị thiếu – có khả năng bị mất phân đoạn.
=> Truyền lại!





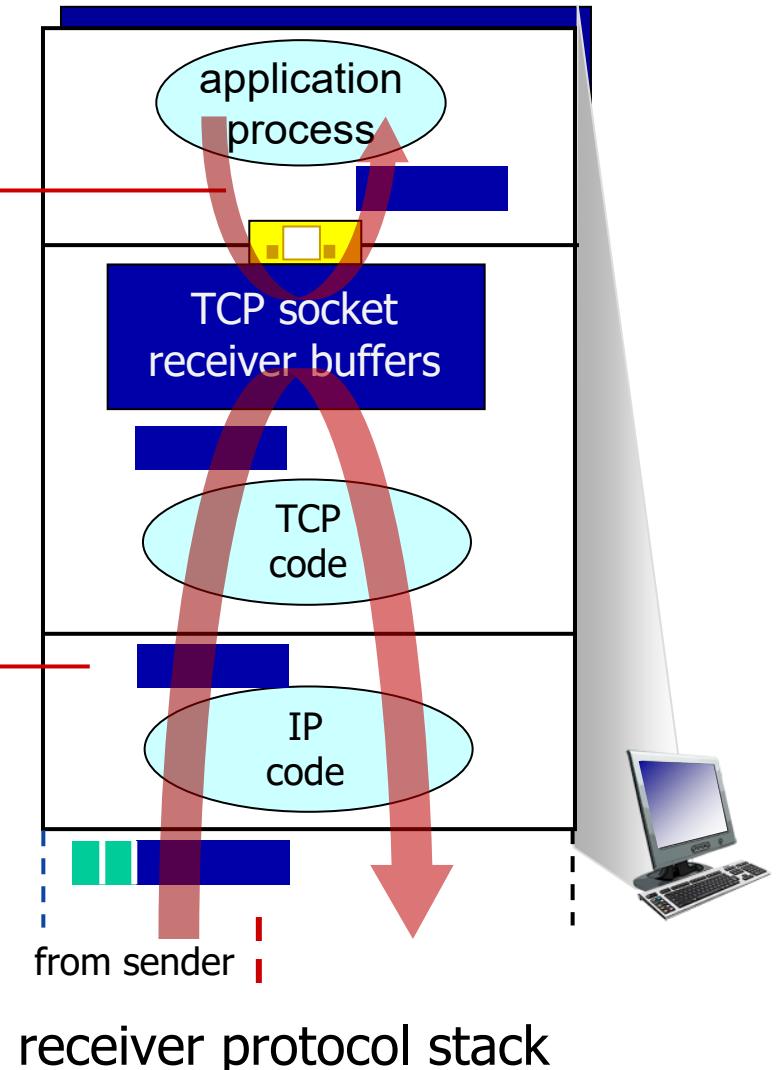
TCP: flow control – điều khiển luồng

Q: Điều gì xảy ra nếu tầng mạng truyền dữ liệu nhanh hơn tầng ứng dụng xóa dữ liệu khỏi bộ đệm của socket?



Application removing data from TCP socket buffers

Network layer delivering IP datagram payload into TCP socket buffers

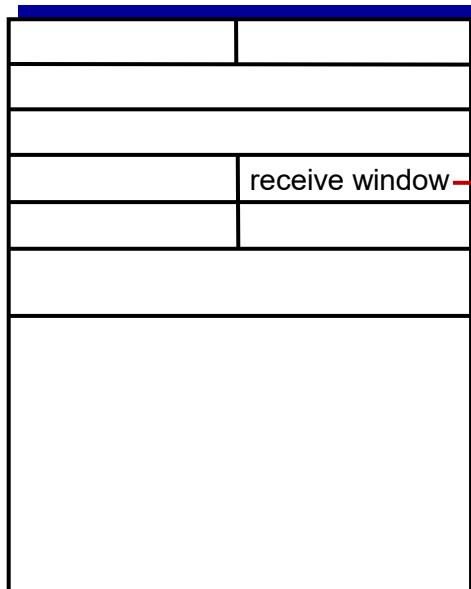


receiver protocol stack



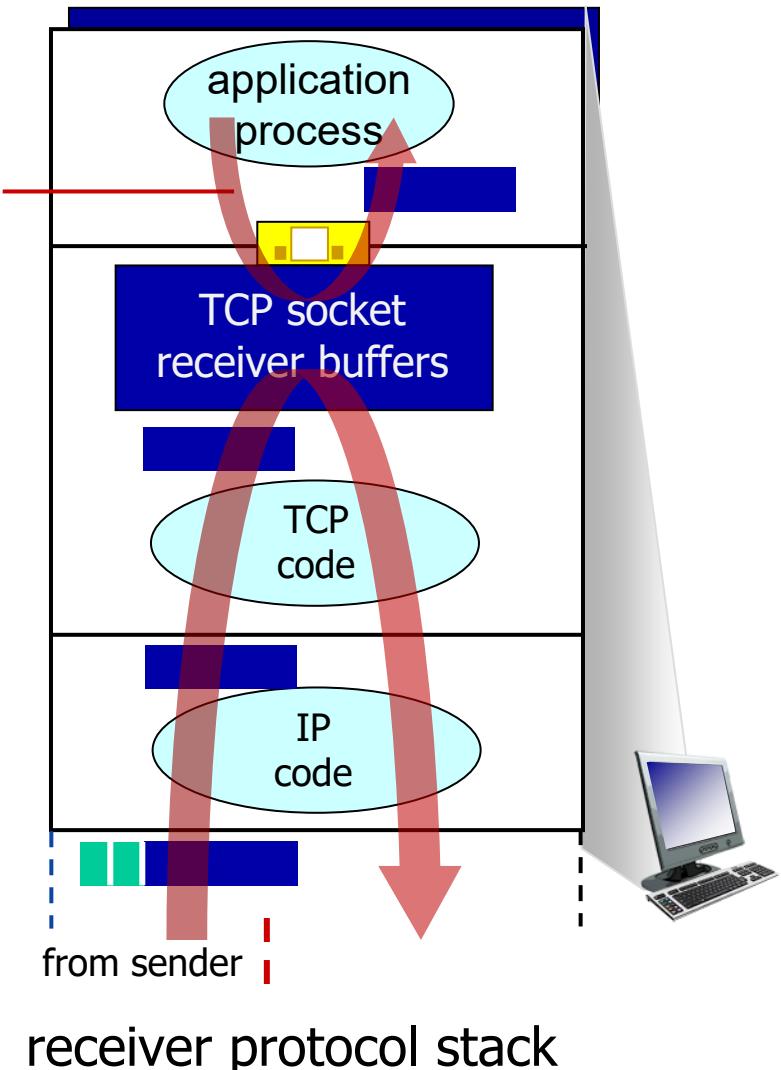
TCP: điều khiển luồng

Q: Điều gì xảy ra nếu tầng mạng truyền dữ liệu nhanh hơn tầng ứng dụng xóa dữ liệu khỏi bộ đệm của socket?



flow control: # bytes
receiver willing to accept

Application removing
data from TCP socket
buffers



receiver protocol stack



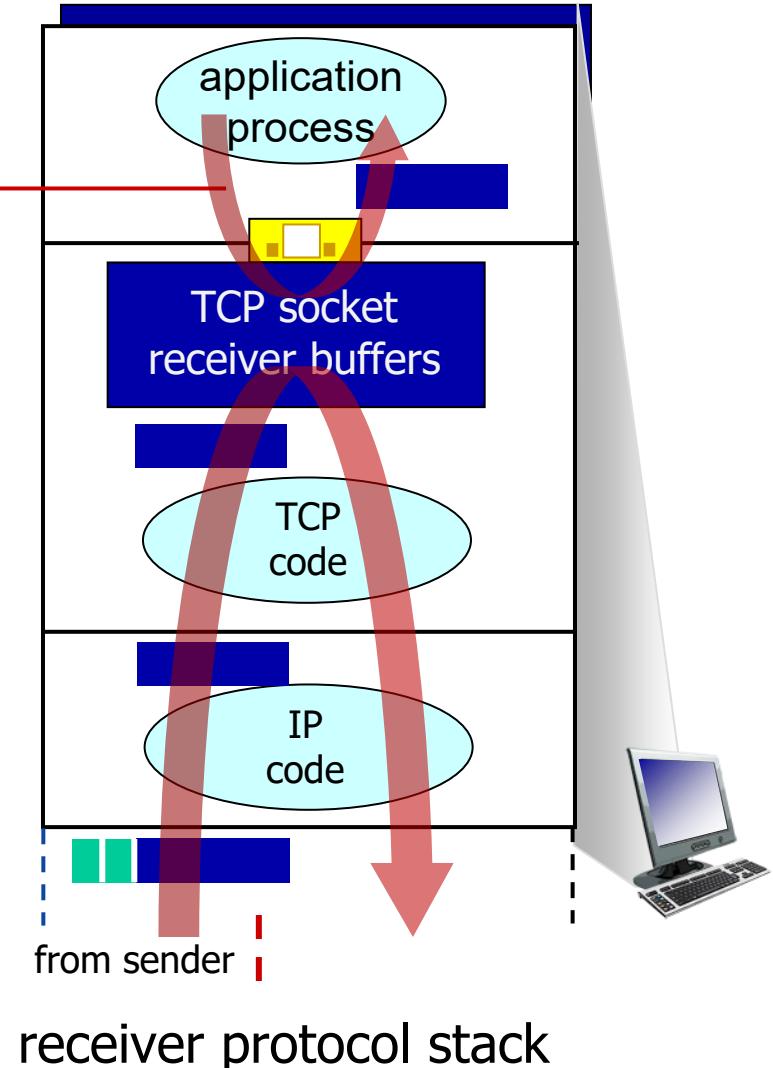
TCP: điều khiển luồng

Q: Điều gì xảy ra nếu tầng mạng truyền dữ liệu nhanh hơn tầng ứng dụng xóa dữ liệu khỏi bộ đếm của socket?

flow control

bên nhận kiểm soát bên gửi, vì vậy bên gửi sẽ không làm tràn bộ đếm của bên nhận bằng cách truyền quá nhiều, quá nhanh

Application removing data from TCP socket buffers

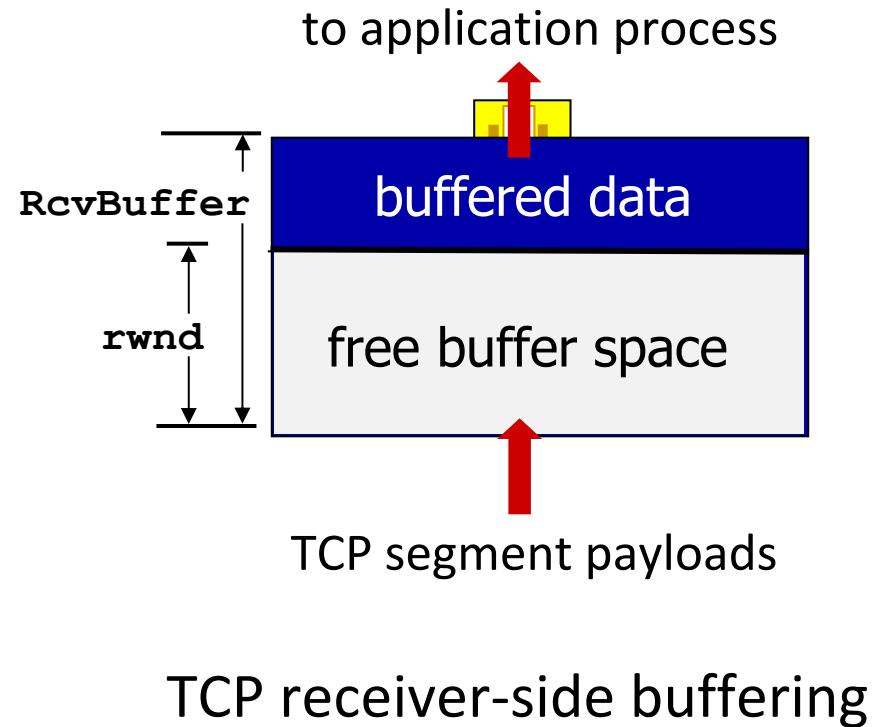


receiver protocol stack



TCP: điều khiển luồng

- Bên gửi thông báo bộ đệm trống bằng thông tin rwnd trong TCP header
- Bên gửi giới hạn lượng dữ liệu được gửi khi nhận được rwnd để đảm bảo bên nhận không bị quá tải

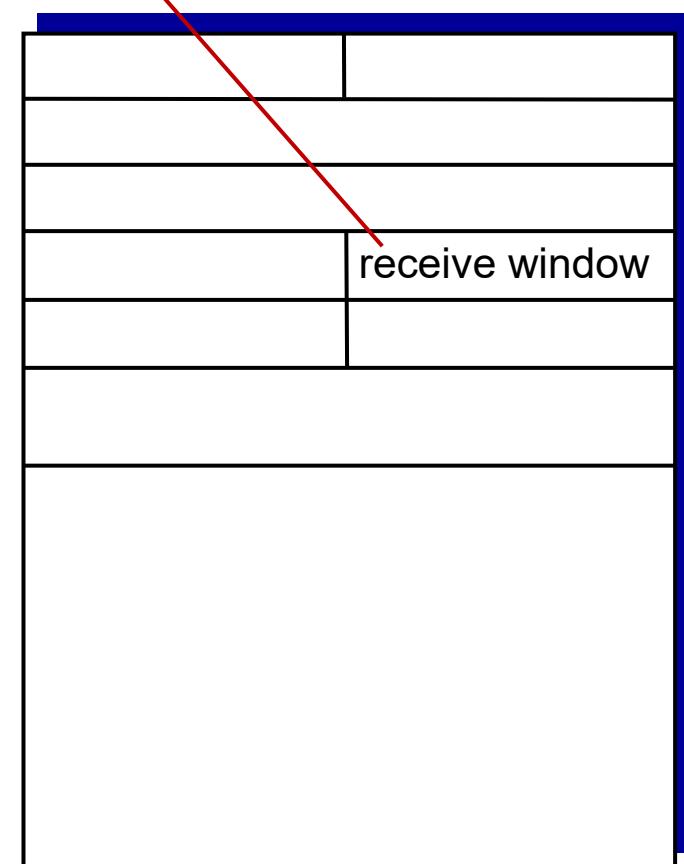




TCP: điều khiển luồng

- Bên gửi thông báo bộ đệm trống bằng thông tin rwnd trong TCP header
- Bên gửi giới hạn lượng dữ liệu được gửi khi nhận được rwnd để đảm bảo bên nhận không bị quá tải

flow control: # bytes receiver willing to accept

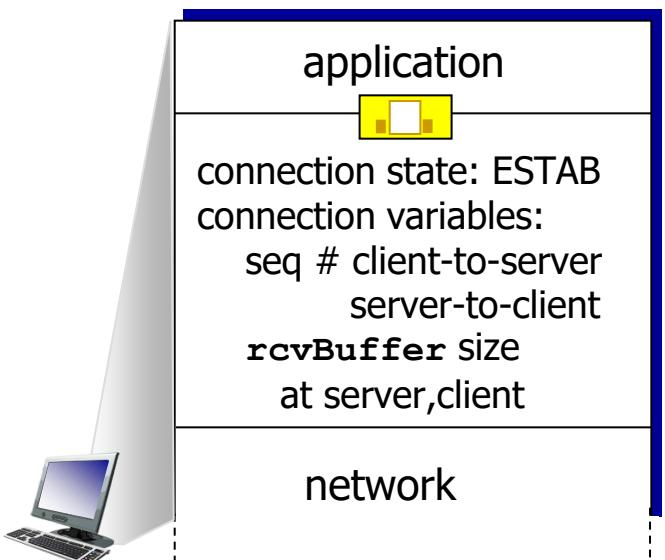


TCP segment format

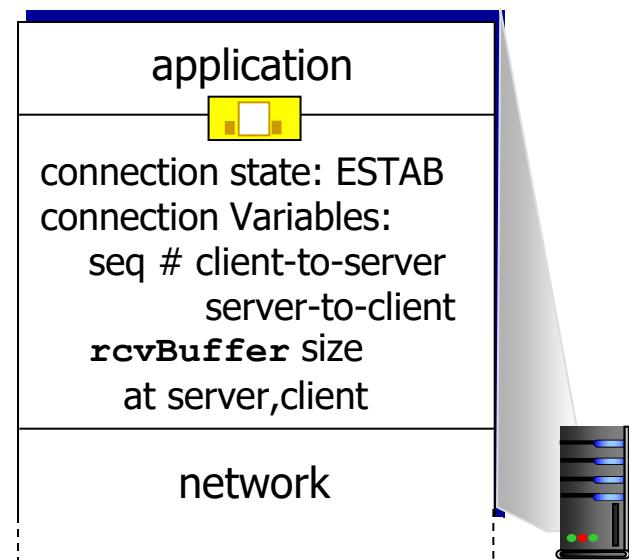


TCP: quản lý kết nối

- Trước khi trao đổi dữ liệu, bên gửi/bên nhận “handshake – bắt tay”:
 - đồng ý thiết lập kết nối (mỗi bên đều biết bên kia sẵn sàng thiết lập kết nối)
 - đồng ý về các thông số kết nối (ví dụ: bắt đầu từ seq #s)



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



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

TCP 3-way handshake: bắt tay 3 bước



Client state

```
clientSocket = socket(AF_INET, SOCK_STREAM)
```

LISTEN

```
clientSocket.connect((serverName, serverPort))
```

SYNSENT

choose init seq num, x
send TCP SYN msg



SYNbit=1, Seq=x

ESTAB

received SYNACK(x)
indicates server is live;
send ACK for SYNACK;
this segment may contain
client-to-server data

SYNbit=1, Seq=y
ACKbit=1; ACKnum=x+1

ACKbit=1, ACKnum=y+1

Server state

```
serverSocket = socket(AF_INET, SOCK_STREAM)  
serverSocket.bind(('', serverPort))  
serverSocket.listen(1)  
connectionSocket, addr = serverSocket.accept()
```

LISTEN

SYN RCVD



choose init seq num, y
send TCP SYNACK
msg, acking SYN

received ACK(y)
indicates client is live

ESTAB



Ví dụ thực tế

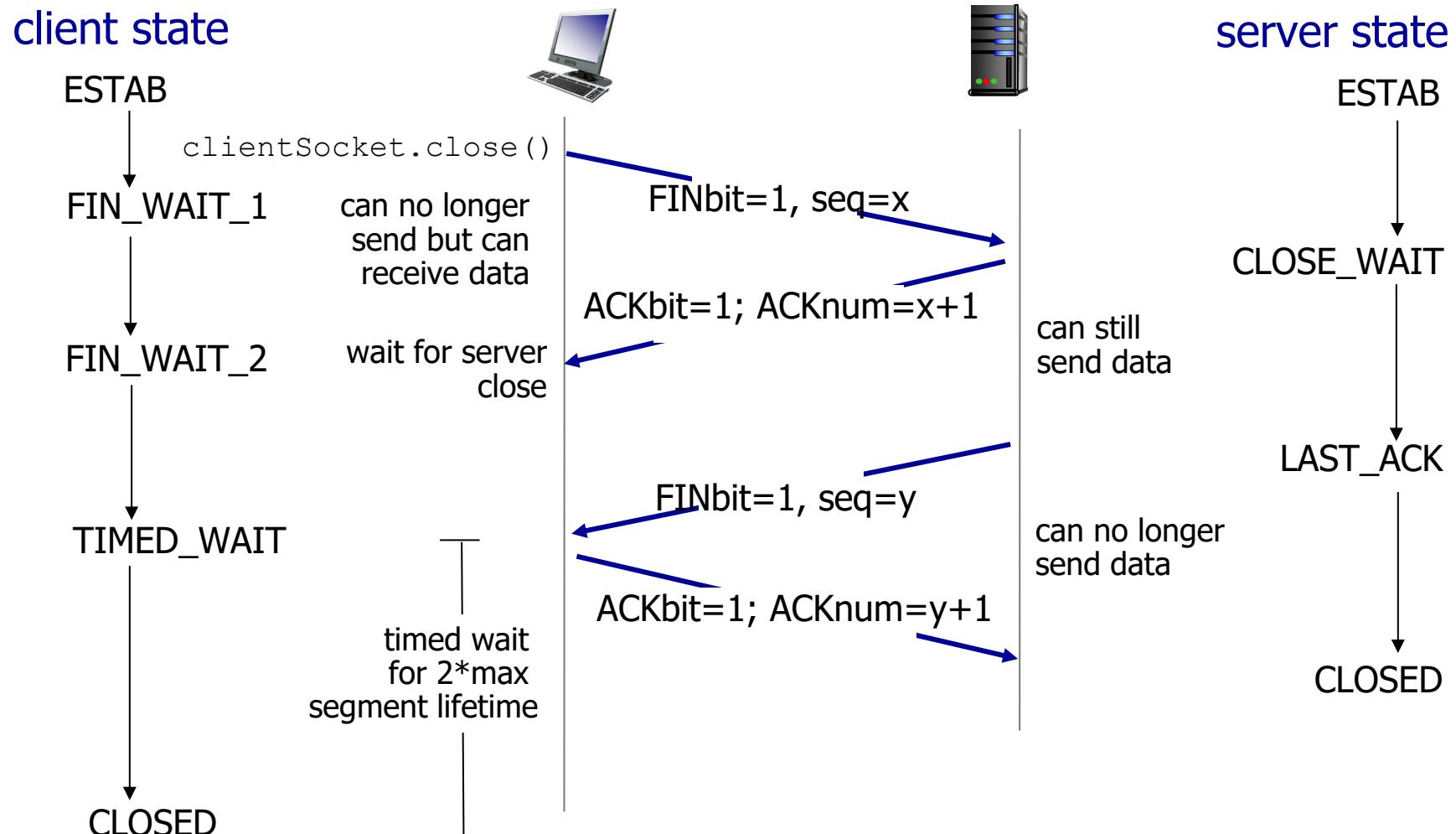




Đóng kết nối

- Client, server đóng kết nối cho mỗi bên
 - Gửi TCP segment có FIN bit = 1
- Phản hồi khi nhận segment có FIN với ACK
- Khi nhận FIN, ACK có thể kết hợp với FIN của nó
 - có thể xử lý các trao đổi FIN đồng thời

Đóng kết nối





Nội dung

- Các dịch vụ tầng vận chuyển
- Multiplexing and demultiplexing
- UDP
- Nguyên lý truyền tin cậy
- TCP
- TCP - Điều khiển tắc nghẽn
- Sự phát triển của các tính năng của tầng vận chuyển





Quản lý tắc nghẽn của TCP: AIMD

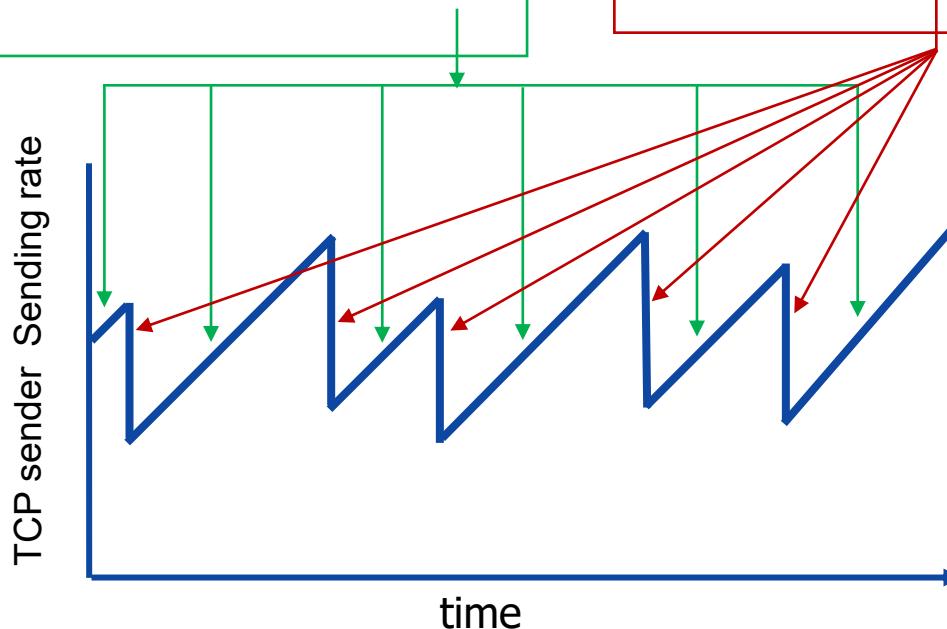
- **Cách tiếp cận:** bên gửi liên tục tăng tốc độ truyền cho đến khi việc mất gói tin xảy ra (tắc nghẽn – congestion), sau đó giảm tốc độ truyền dựa trên sự kiện mất.

Tăng cấp số cộng

Tăng tốc độ gửi lên 1 MSS cho đến khi phát hiện sự mất mát

Giảm cấp số nhân

Giảm tốc độ gửi 1 nửa khi mất mát xảy ra



AIMD : thăm dò băng thông



TCP: Điều khiển tắc nghẽn

○ Ký hiệu

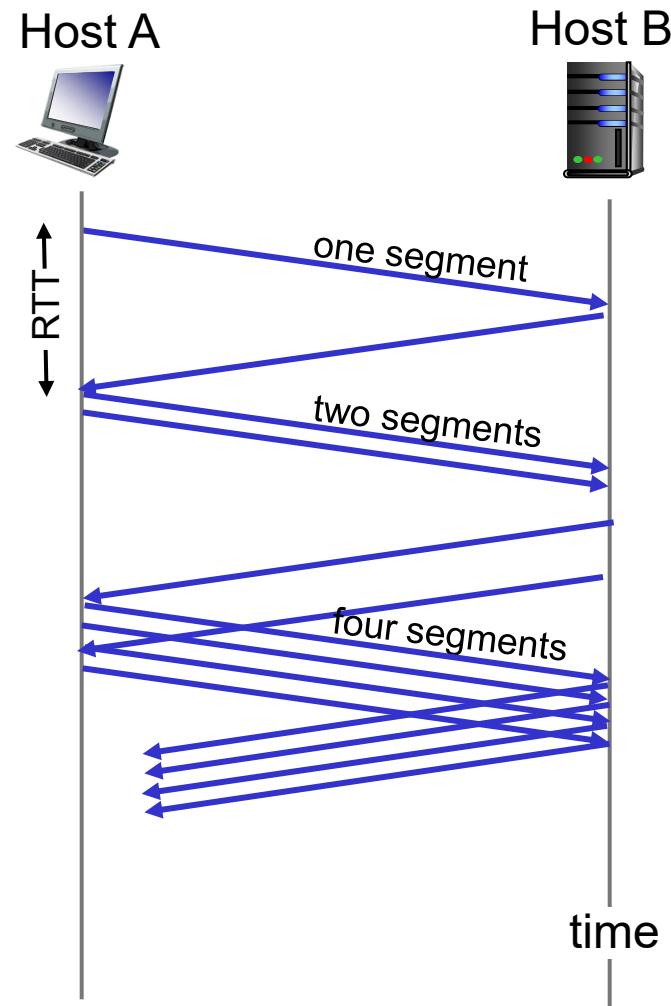
- Cwnd(n) - congestion windows: số segment được gửi trong cùng một window
- Ssthresh (slowstart threshold): ngưỡng kết thúc giai đoạn slowstart và chuyển sang congestion avoidance



TCP slow start

- Khi bắt đầu, tăng tốc độ theo cấp số nhân cho đến khi xảy ra sự kiện mất đầu tiên :
 - Ban đầu **cwnd** = 1 MSS
 - Nhân đôi **cwnd** sau mỗi RTT
 - Được thực hiện bằng cách tăng **cwnd** cho mỗi ACK nhận được

Tóm tắt: tốc độ ban đầu chậm, nhưng tăng nhanh theo cấp số nhân



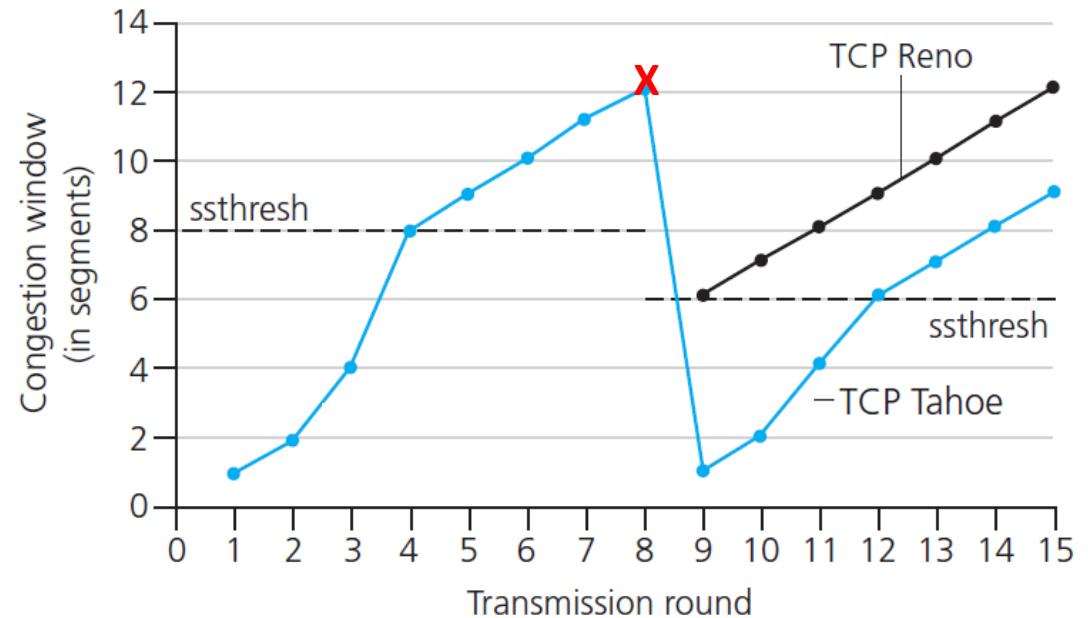


TCP: từ slow start đến congestion avoidance

- Q: khi nào nên tăng theo cấp số nhân chuyển sang tuyến tính?

O A:

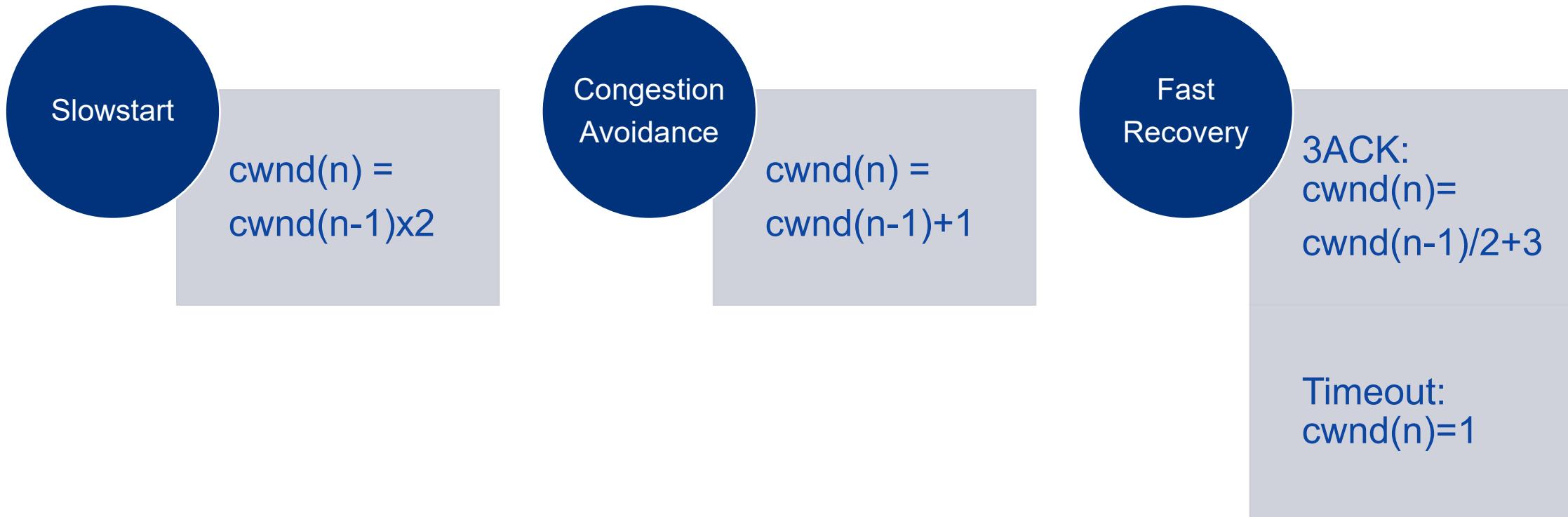
- **Ssthresh**: ngưỡng kết thúc Slowstart
- Khi sự kiện mất xảy ra, **ssthresh** bằng $1/2$ giá trị của **cwnd** trước khi xảy ra mất mát



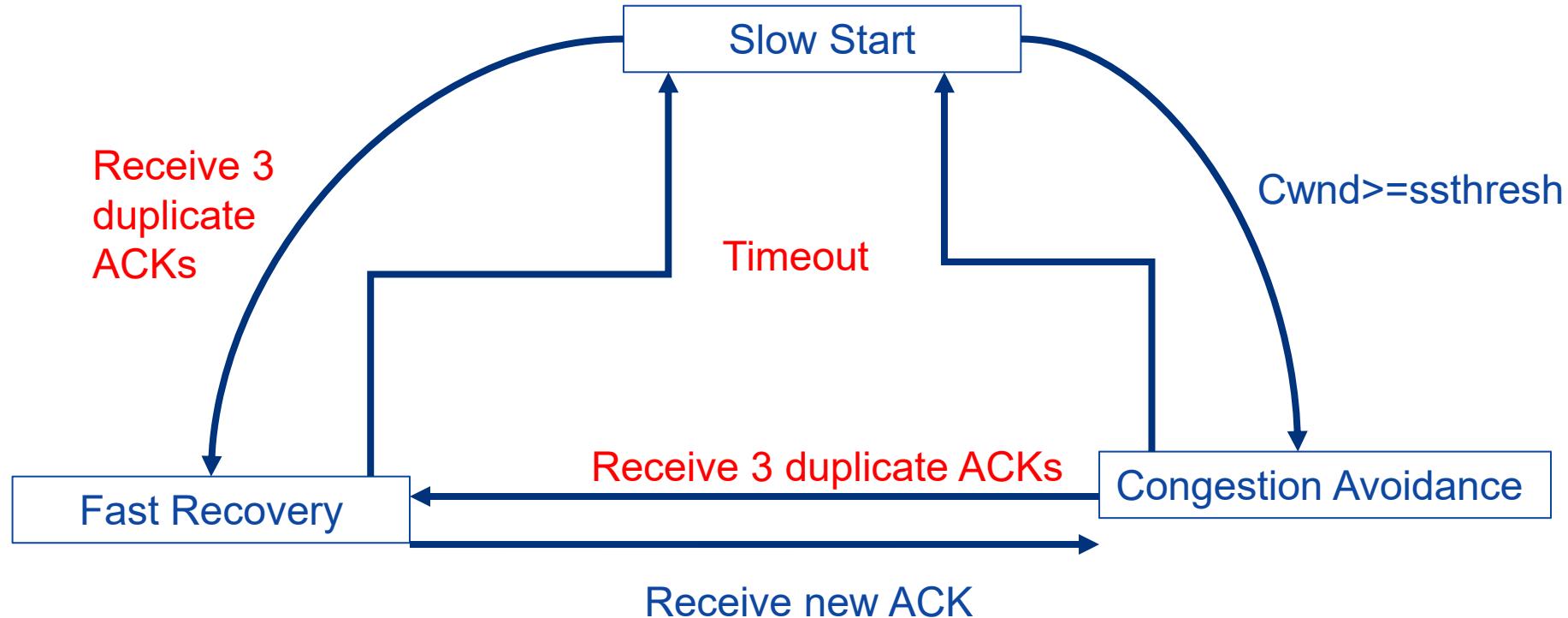
* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/



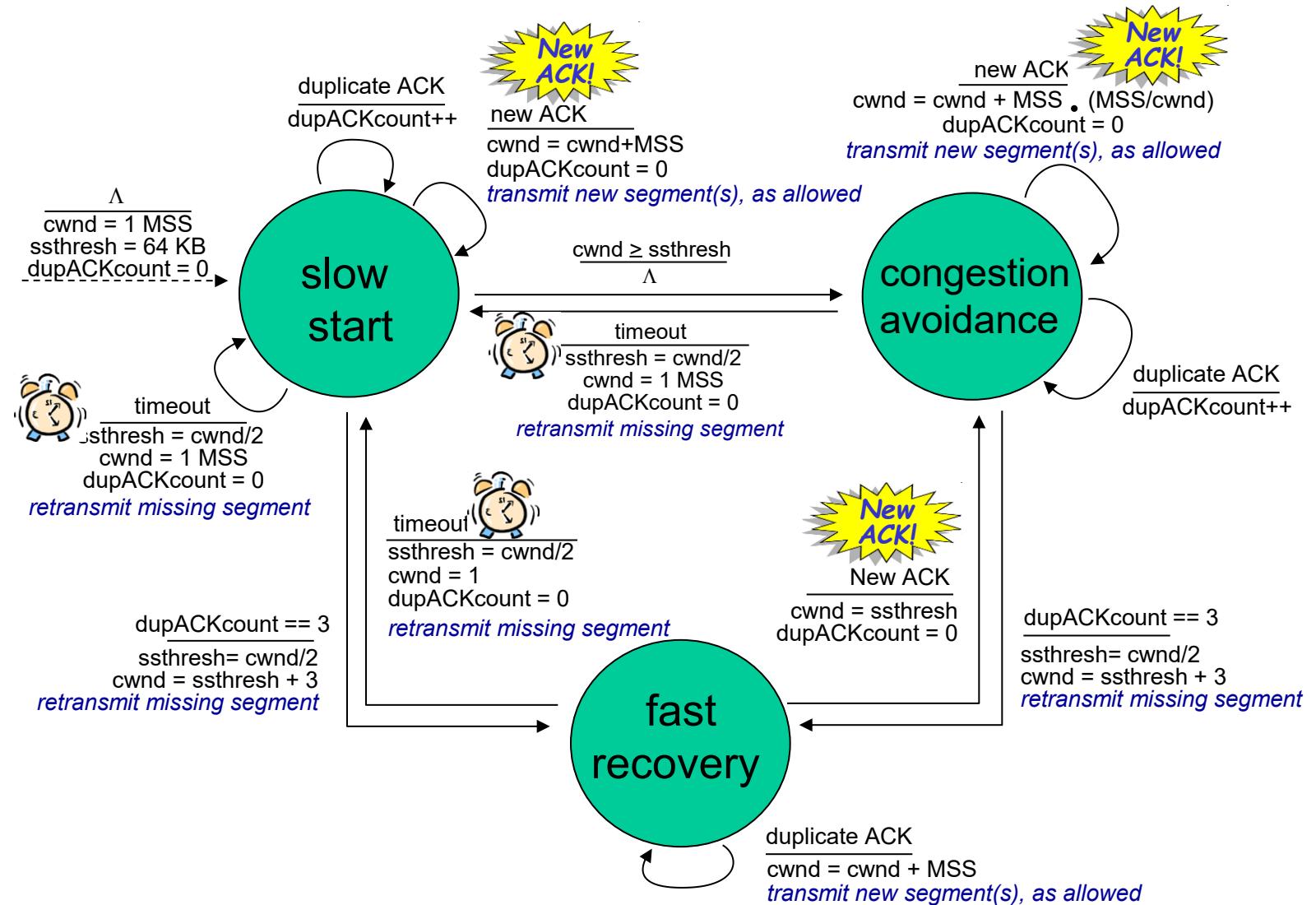
TCP: Điều khiển tắc nghẽn – Các giai đoạn



TCP: Điều khiển tắc nghẽn – Chuyển giai đoạn



Tổng kết





Nội dung

- Các dịch vụ tầng vận chuyển
- Multiplexing and demultiplexing
- UDP
- Nguyên lý truyền tin cậy
- TCP
- TCP - Điều khiển tắc nghẽn
- Sự phát triển của các tính năng của tầng vận chuyển





Evolving transport-layer functionality

- TCP, UDP: principal transport protocols for 40 years
- different “flavors” of TCP developed, for specific scenarios:

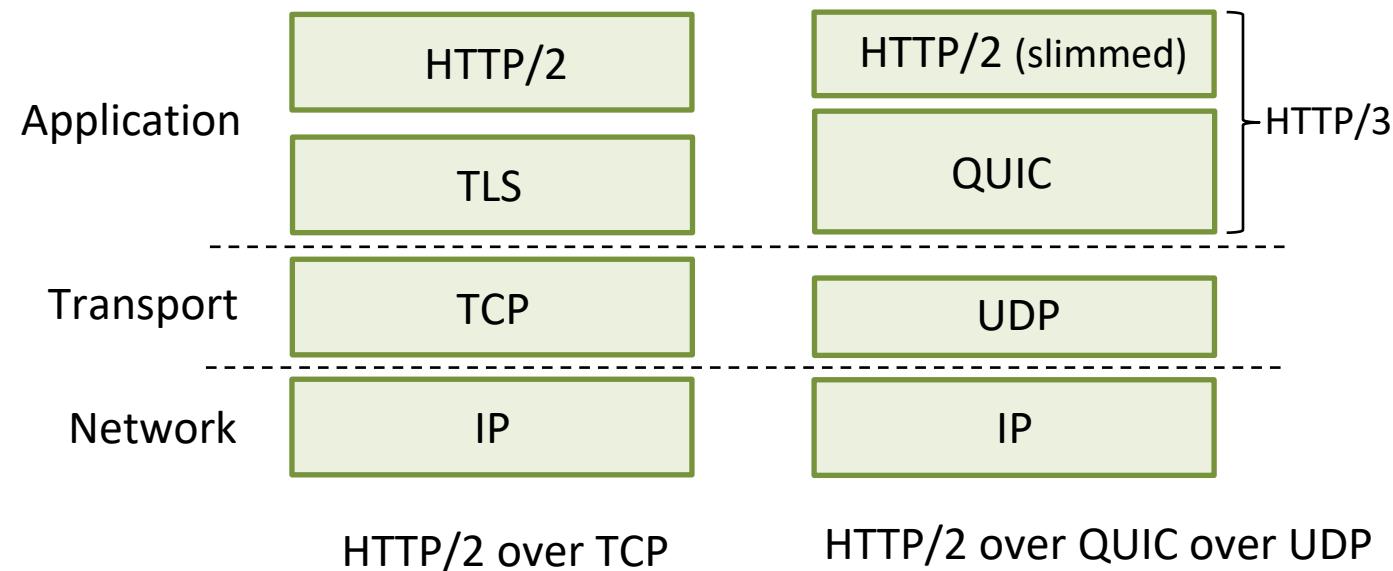
Scenario	Challenges
Long, fat pipes (large data transfers)	Many packets “in flight”; loss shuts down pipeline
Wireless networks	Loss due to noisy wireless links, mobility; TCP treat this as congestion loss
Long-delay links	Extremely long RTTs
Data center networks	Latency sensitive
Background traffic flows	Low priority, “background” TCP flows

- moving transport–layer functions to application layer, on top of UDP
 - HTTP/3: QUIC



QUIC: Quick UDP Internet Connections

- application-layer protocol, on top of UDP
 - increase performance of HTTP
 - deployed on many Google servers, apps (Chrome, mobile YouTube app)



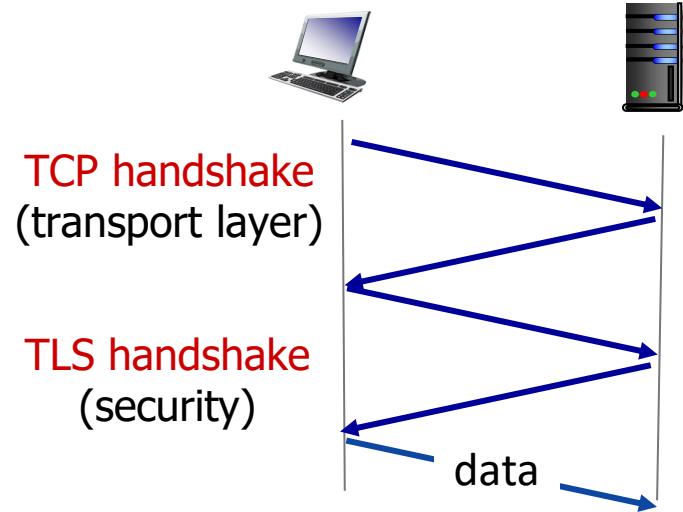
QUIC: Quick UDP Internet Connections



- multiple application-level “streams” multiplexed over single QUIC connection
 - separate reliable data transfer, security
 - common congestion control
 - **error and congestion control:** “Readers familiar with TCP’s loss detection and congestion control will find algorithms here that parallel well-known TCP ones.” [from QUIC specification]
 - **connection establishment:** reliability, congestion control, authentication, encryption, state established in one RTT

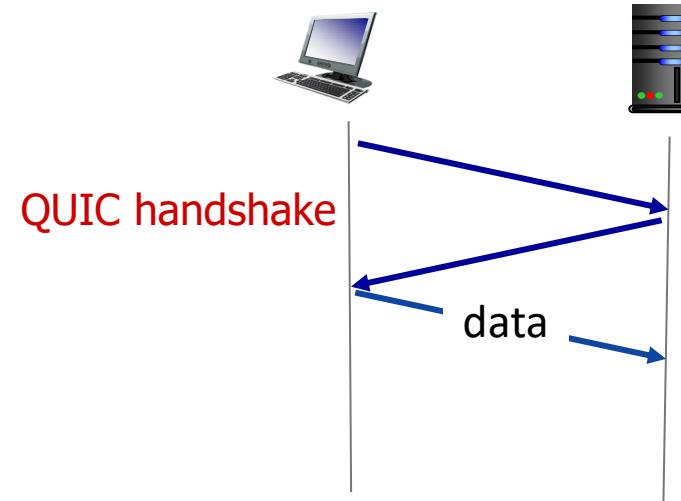
adopts approaches we’ve studied in this chapter for connection establishment, error control, congestion control

QUIC: Connection establishment



TCP (reliability, congestion control state) + TLS (authentication, crypto state)

- 2 serial handshakes



QUIC: reliability, congestion control, authentication, crypto state

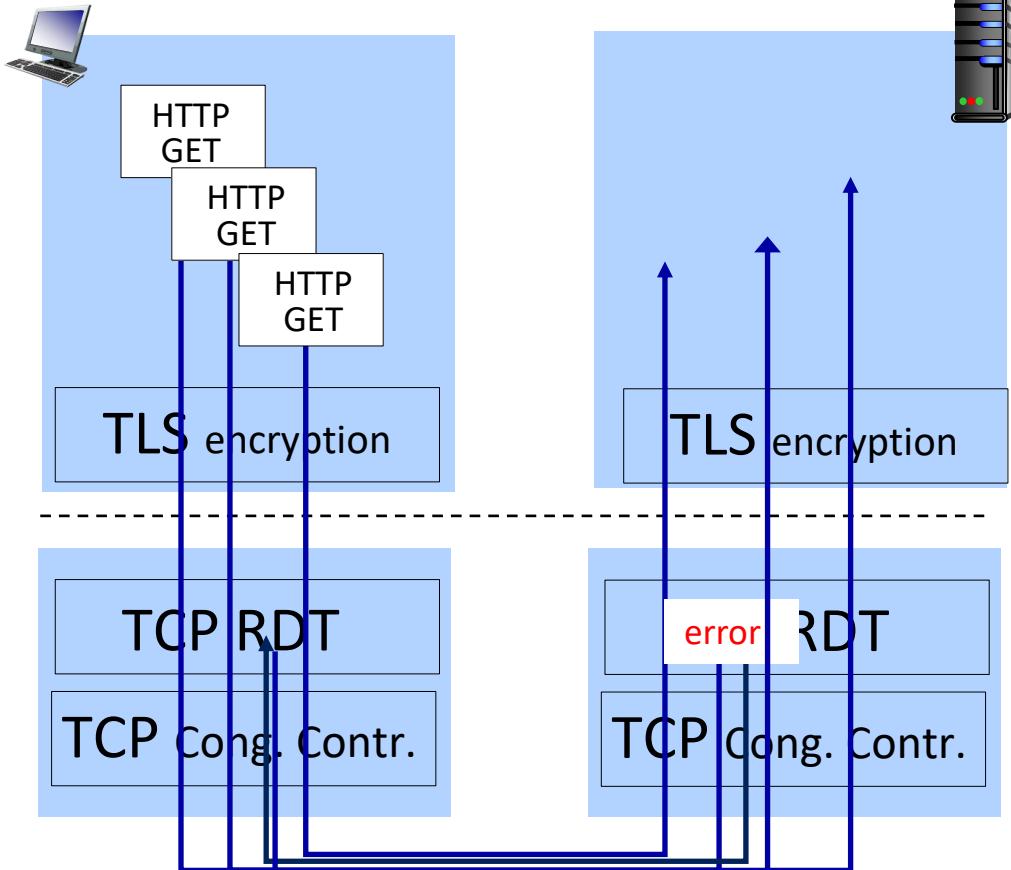
- 1 handshake

QUIC: streams: parallelism, no HOL blocking

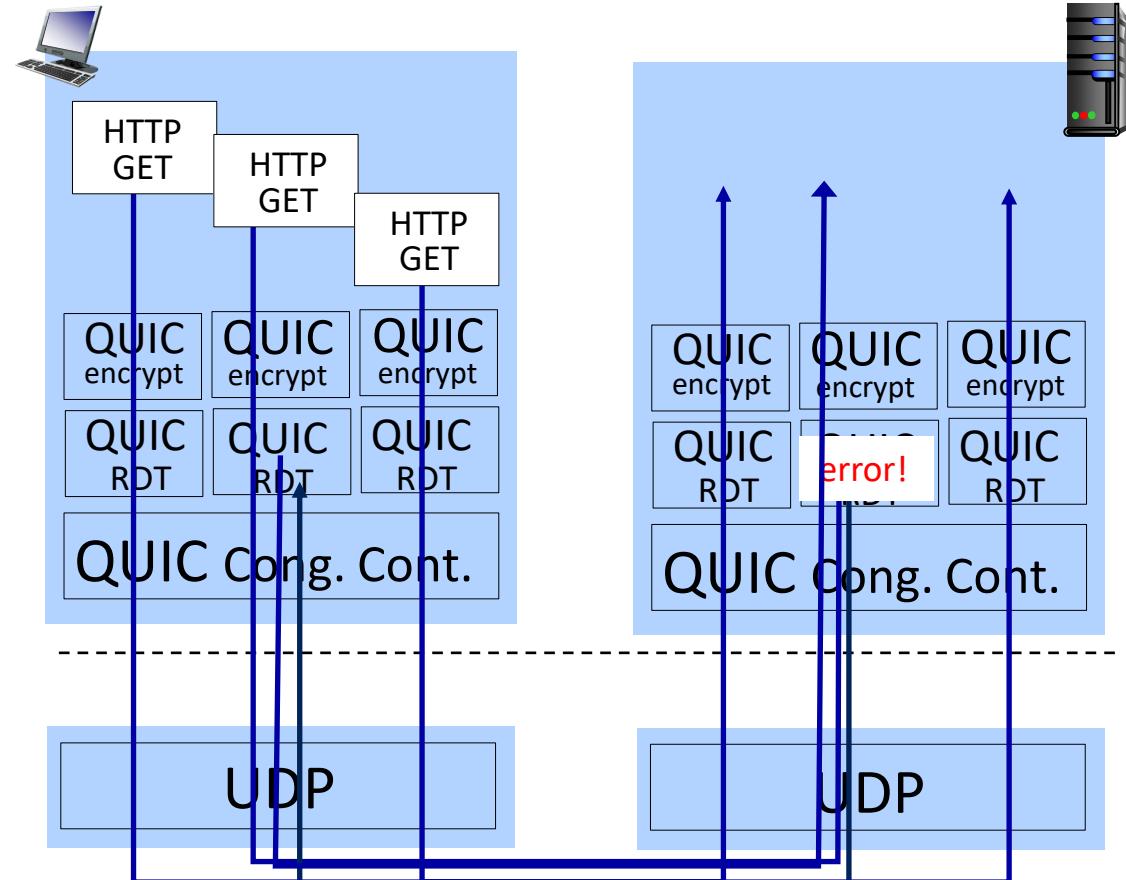


application

transport



(a) HTTP 1.1



(b) HTTP/2 with QUIC: no HOL blocking



Chương 3: Tổng kết

- Các nguyên lý của tầng vận chuyển:
 - multiplexing, demultiplexing
 - Truyền tin cây
 - Điều khiển luồng
 - Điều khiển tắc nghẽn
 - UDP
 - TCP

Tiếp theo:

- Tầng network