

# Chương 3

## Tầng Transport

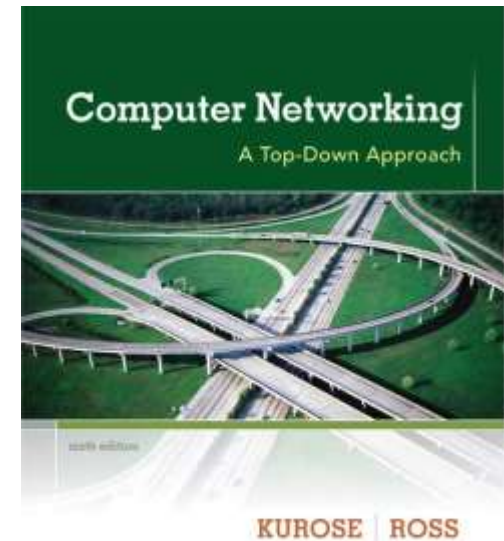
### A note on the use of these ppt slides:

We're making these slides freely available to all (faculty, students, readers). They're in PowerPoint form so you 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.

Thanks and enjoy! JFK/KWR

© All material copyright 1996-2012  
J.F Kurose and K.W. Ross, All Rights Reserved



**Computer  
Networking: A Top  
Down Approach**  
6<sup>th</sup> edition  
Jim Kurose, Keith Ross  
Addison-Wesley  
March 2012

# Chương 3: Tầng Transport

## Mục tiêu:

- ❖ Hiểu về các nguyên lý đằng sau các dịch vụ tầng transport:
  - multiplexing/demultiplexing
  - Truyền dữ liệu tin cậy
  - Điều khiển luồng (flow control)
  - Điều khiển tắc nghẽn (congestion control)
- ❖ Tìm hiểu về các giao thức tầng transport trên Internet:
  - UDP: vận chuyển phi kết nối
  - TCP: vận chuyển tin cậy hướng kết nối (connection-oriented reliable transport)
  - Điều khiển tắc nghẽn TCP

# Chương 3: Nội dung

## 3.1 các dịch vụ tầng Transport

## 3.2 multiplexing và demultiplexing

## 3.3 vận chuyển phi kết nối: UDP

## 3.4 các nguyên lý truyền dữ liệu tin cậy

## 3.5 vận chuyển hướng kết nối: TCP

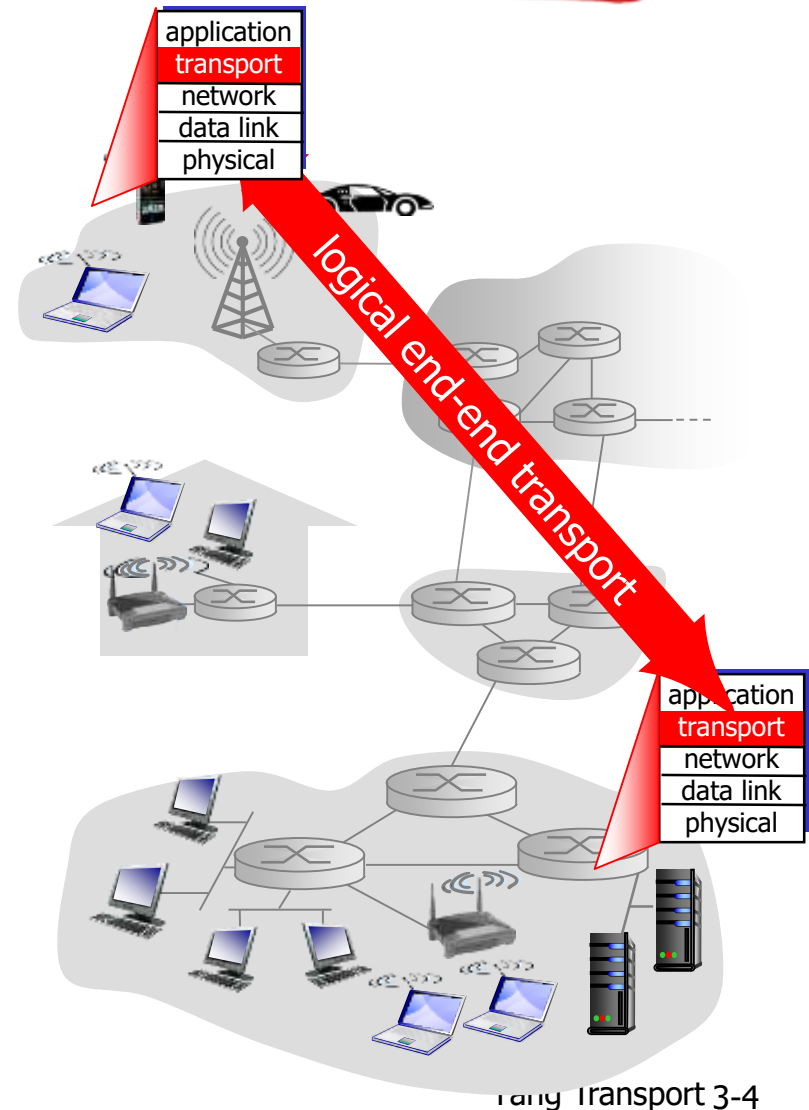
- Cấu trúc segment
- Truyền dữ liệu tin cậy
- Điều khiển luồng (flow control)
- Quản lý kết nối

## 3.6 các nguyên lý về điều khiển tắc nghẽn

## 3.7 điều khiển tắc nghẽn TCP

# Các giao thức và dịch vụ tầng Transport

- ❖ Cung cấp **truyền thông logic** giữa các tiến trình ứng dụng đang chạy trên có host khác nhau
- ❖ Các giao thức protocols chạy trên các hệ thống đầu cuối
  - Phía gửi: chia nhỏ các thông điệp (message) ứng dụng thành các **segments**, sau đó chuyển các segments này cho tầng network
  - Phía nhận: tái kết hợp các segments thành các thông điệp (message), các thông điệp này được chuyển lên tầng Application
- ❖ Có nhiều hơn 1 giao thức tầng transport dành cho các ứng dụng
  - Internet: TCP và UDP



# Tầng Transport với tầng network

- ❖ *Tầng network*: truyền thông logic giữa các host
- ❖ *Tầng transport* : truyền thông logic giữa các tiến trình
  - Dựa vào và tăng cường các dịch vụ tầng network

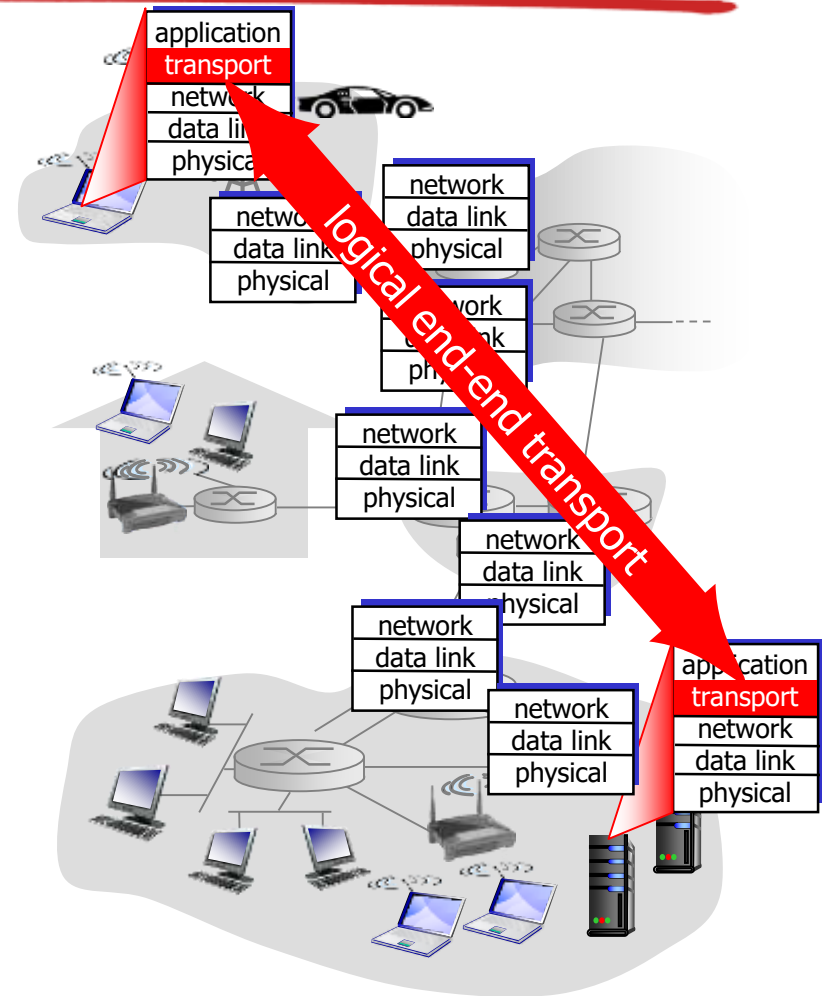
*Tình huống tương tự:*

12 đứa trẻ ở nhà Ann gửi những bức thư đến 12 đứa trẻ ở nhà Bill:

- ❖ hosts = nhà
- ❖ Các tiến trình (processes) = những đứa trẻ
- ❖ Thông điệp tầng ứng dụng = các bức thư trong các phong bì
- ❖ Giao thức tầng transport = Ann and Bill
- ❖ Giao thức tầng network = dịch vụ bưu điện

# Các giao thức tầng transport trên Internet

- ❖ Tin cậy, truyền theo thứ tự (TCP)
  - Điều khiển tắc nghẽn
  - Điều khiển luồng
  - Thiết lập kết nối
- ❖ Không tin cậy, truyền không theo thứ tự: UDP
  - Không rườm rà, mở rộng "nỗ lực tốt nhất" (best-effort) của IP
- ❖ Không có các dịch vụ:
  - Bảo đảm độ trễ
  - Bảo đảm băng thông



# Chương 3 Nội dung

3.1 các dịch vụ tầng  
Transport

3.2 multiplexing và  
demultiplexing

3.3 vận chuyển phi kết  
nối: UDP

3.4 các nguyên lý  
truyền dữ liệu tin  
cậy

3.5 vận chuyển hướng  
kết nối: TCP

- Cấu trúc segment
- Truyền dữ liệu tin cậy
- Điều khiển luồng (flow control)
- Quản lý kết nối

3.6 các nguyên lý về điều  
khiển tắc nghẽn

3.7 điều khiển tắc nghẽn  
TCP

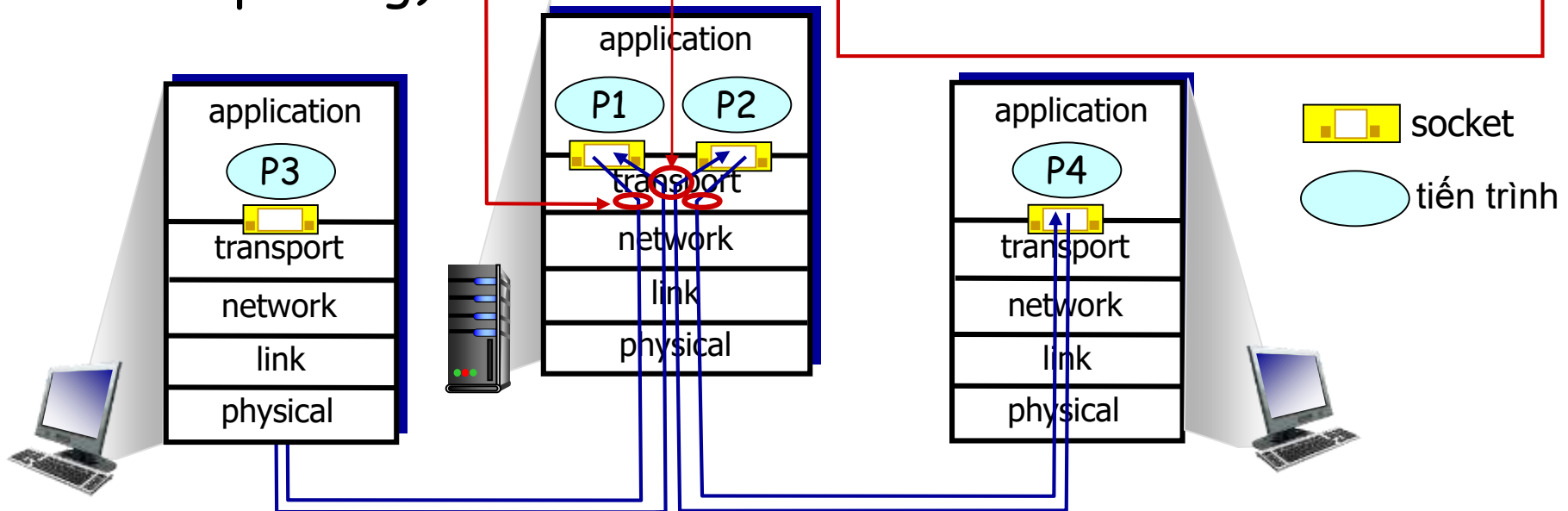
# Multiplexing/demultiplexing

*multiplexing tại bên gửi:*

xử lý dữ liệu từ nhiều socket, thêm thông tin header về tầng transport vào segment (được sử dụng sau cho demultiplexing)

*demultiplexing tại bên nhận:*

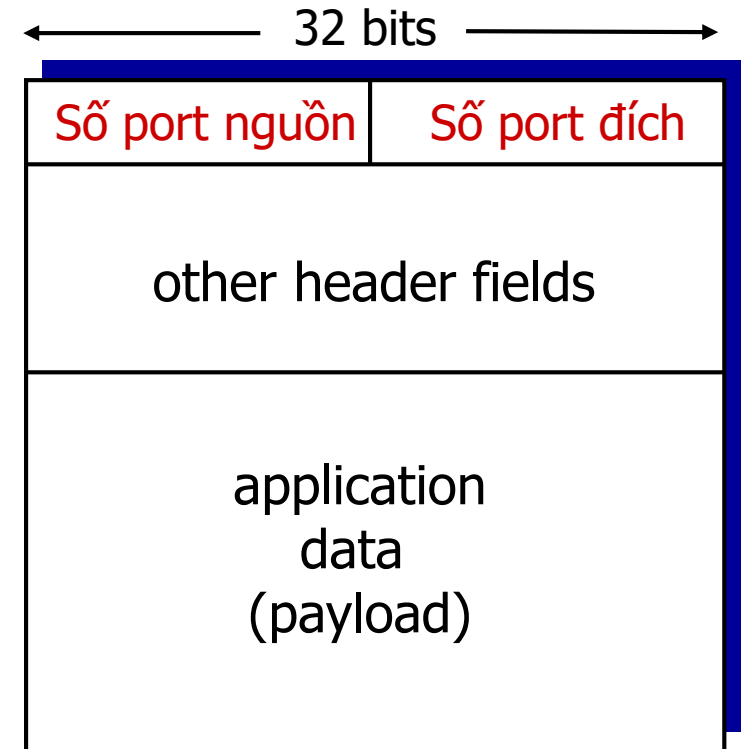
sử dụng thông tin trong header để chuyển segment vừa nhận vào đúng socket





# demultiplexing làm việc như thế nào

- ❖ host nhận các khung dữ liệu (datagram) IP
  - Mỗi khung dữ liệu có địa chỉ IP nguồn và đích
  - Mỗi khung dữ liệu mang một segment tầng transport
  - Mỗi segment có số port nguồn và đích
- ❖ host dùng các địa chỉ IP và số port để gửi segment đến socket thích hợp



Định dạng segment TCP/UDP

# Demultiplexing không kết nối

- ❖ Ôn lại: socket đã tạo có số port của host cục bộ (host-local port #) :  
`DatagramSocket mySocket1 = new DatagramSocket(12534);`
- ❖ Ôn lại: khi tạo khung dữ liệu (datagram) để gửi vào đến socket UDP socket, phải xác định
  - Địa chỉ IP đích
  - Số port đích

- ❖ Khi host nhận segment UDP :
  - Kiểm tra số port đích trong segment
  - Đưa segment UDP đến socket có số port đó



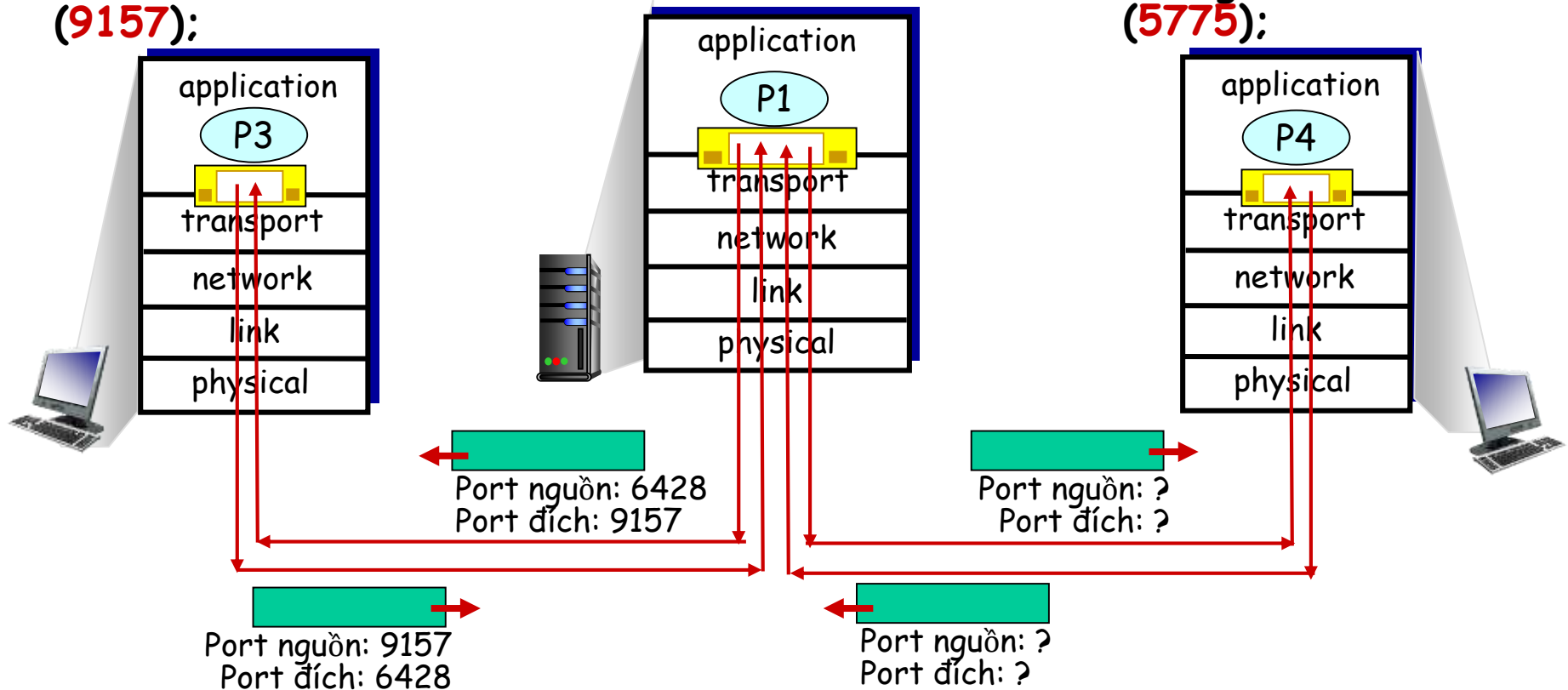
Các khung dữ liệu IP với số cùng số port đích, nhưng khác địa chỉ IP nguồn và/hoặc khác số port nguồn sẽ được chuyển đến cùng socket tại máy đích

# Demultiplexing không kết nối: ví dụ

DatagramSocket  
mySocket2 = new  
DatagramSocket  
(9157);

DatagramSocket  
serverSocket = new  
DatagramSocket  
(6428);

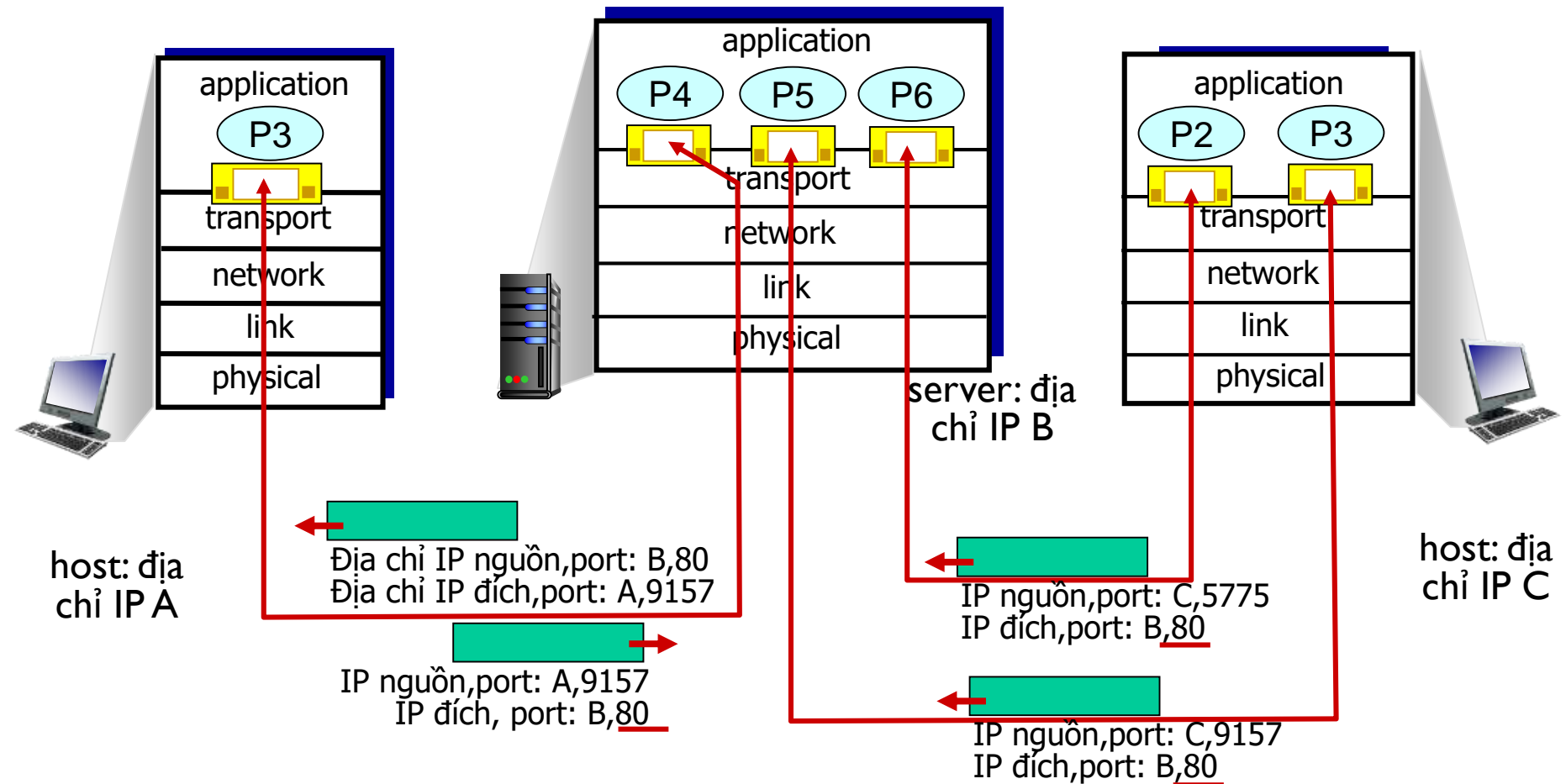
DatagramSocket  
mySocket1 = new  
DatagramSocket  
(5775);



# Demux hướng kết nối

- ❖ Socket TCP được xác định bởi 4 yếu tố:
  - Địa chỉ ip nguồn
  - Số port nguồn
  - Địa chỉ IP đích
  - Số port đích
- ❖ demux: nơi nhận dùng tất cả 4 giá trị trên để điều hướng segment đến socket thích hợp
- ❖ host server có thể hỗ trợ nhiều socket TCP đồng thời:
  - Mỗi socket được xác định bởi bộ 4 của nó
- ❖ Các web server có các socket khác nhau cho mỗi kết nối từ client
  - Kết nối HTTP không bền vững sẽ có socket khác nhau cho mỗi yêu cầu

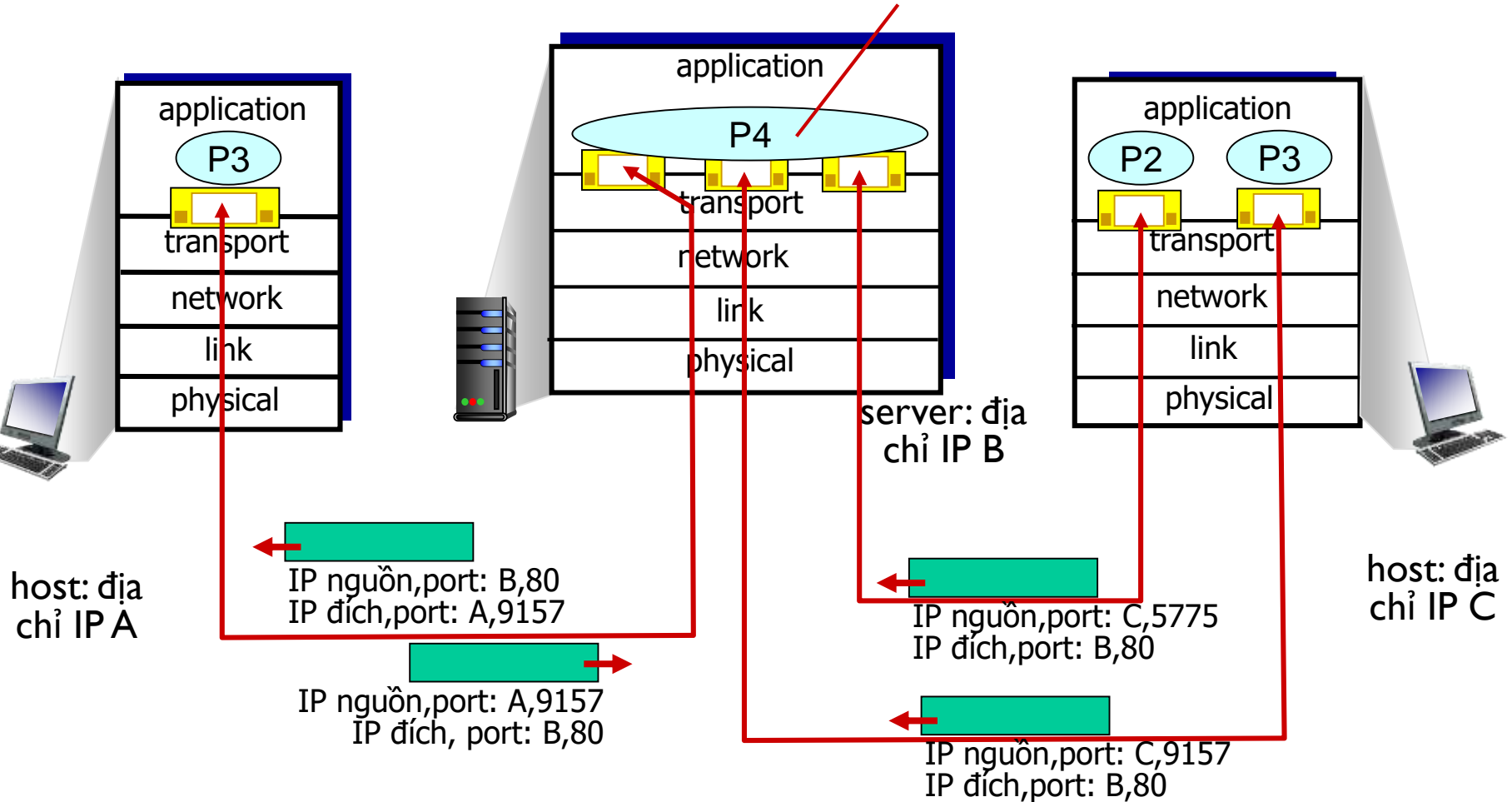
# Demultiplexing hướng kết nối: ví dụ



Ba segment, tất cả được đưa đến địa chỉ IP: B,  
Port đích: 80 được demultiplex đến các socket khác nhau

# Demultiplexing hướng kết nối: ví dụ

threaded server



# Chương 3 Nội dung

3.1 các dịch vụ tầng  
Transport

3.2 multiplexing và  
demultiplexing

3.3 vận chuyển phi kết  
nối: UDP

3.4 các nguyên lý  
truyền dữ liệu tin  
cậy

3.5 vận chuyển hướng  
kết nối: TCP

- Cấu trúc segment
- Truyền dữ liệu tin cậy
- Điều khiển luồng (flow control)
- Quản lý kết nối

3.6 các nguyên lý về điều  
khiển tắc nghẽn

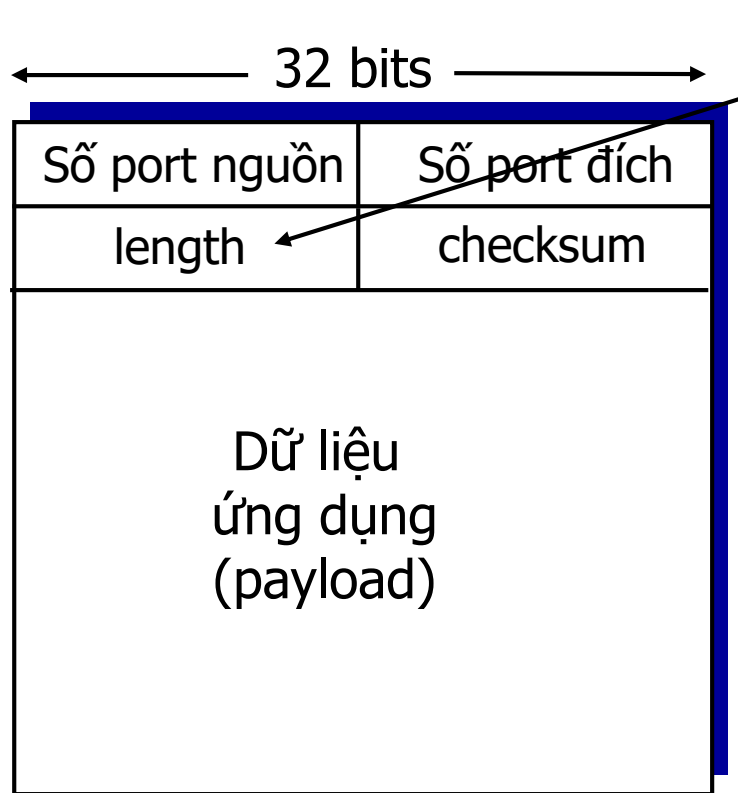
3.7 điều khiển tắc nghẽn  
TCP

# UDP: User Datagram Protocol [RFC 768]

- ❖ “đơn giản,” “bare bones” Internet transport protocol
- ❖ Dịch vụ “best effort” (“nỗ lực tốt nhất”), các segment UDP có thể bị:
  - Mất mát
  - Vận chuyển không theo thứ tự đến ứng dụng
- ❖ *Connectionless (phi kết nối):*
  - Không bắt tay giữa bên nhận và gửi UDP
  - Mỗi segment UDP được xử lý độc lập
- ❖ Ứng dụng UDP:
  - Các ứng dụng đa phương tiện trực tuyến chịu mất mát (loss tolerant), cần tốc độ (rate sensitive)
  - DNS
  - SNMP
- ❖ Truyền tin cậy trên UDP:
  - Thêm độ tin cậy tại tầng application
  - Phục hồi lỗi tại các ứng dụng cụ thể!



# UDP: segment header



Định dạng segment UDP

Độ dài được tính bằng byte của segment UDP, bao gồm cả header

## Tại sao có UDP?

- ❖ Không thiết lập kết nối (cái mà có thể gây ra độ trễ)
- ❖ Đơn giản: không trạng thái kết nối tại nơi gửi và nhận
- ❖ Kích thước header nhỏ
- ❖ Không điều khiển tắc nghẽn: UDP có thể gửi dữ liệu nhanh như mong muốn

# UDP checksum

*Mục tiêu:* dò tìm "các lỗi" (các bit cờ được bật) trong các segment đã được truyền

## *bên gửi:*

- ❖ Xét nội dung của segment, bao gồm các trường của header, là chuỗi các số nguyên 16-bit
- ❖ checksum: bổ sung (tổng bù 1) của các nội dung segment
- ❖ Bên gửi đặt giá trị checksum vào trường checksum UDP

## *bên nhận:*

- ❖ Tính toán checksum của segment đã nhận
- ❖ Kiểm tra giá trị trên có bằng với giá trị trong trường checksum hay không:
  - NO - có lỗi xảy ra
  - YES - không có lỗi. *Nhưng có thể còn lỗi khác nữa không? Xem phần sau....*

# Internet checksum: ví dụ

Ví dụ: cộng 2 số nguyên 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/>    |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| bit dư   | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 |
| <hr/>    |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| tổng     | 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 |

*Lưu ý:* khi cộng các số, bit nhớ ở phía cao nhất cần được thêm vào kết quả

# Chương 3 Nội dung

3.1 các dịch vụ tầng  
Transport

3.2 multiplexing và  
demultiplexing

3.3 vận chuyển phi kết  
nối: UDP

3.4 các nguyên lý  
truyền dữ liệu tin  
cậy

3.5 vận chuyển hướng  
kết nối: TCP

- Cấu trúc segment
- Truyền dữ liệu tin cậy
- Điều khiển luồng (flow control)
- Quản lý kết nối

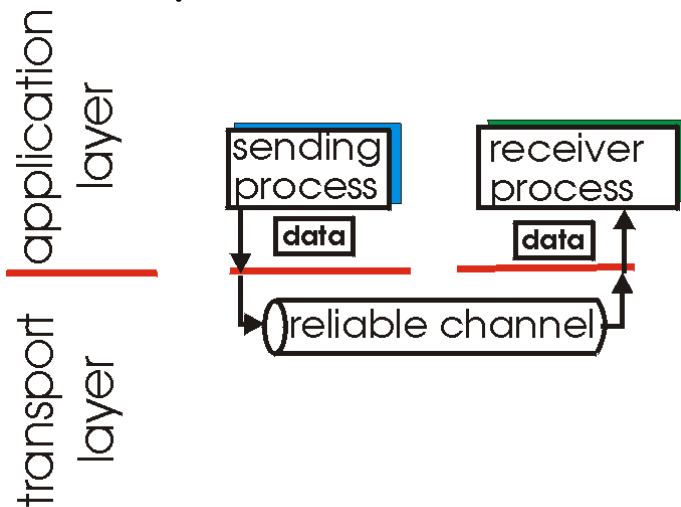
3.6 các nguyên lý về điều  
khiển tắc nghẽn

3.7 điều khiển tắc nghẽn  
TCP

# Các nguyên lý truyền dữ liệu tin cậy

Quan trọng trong các tầng application, transport và link

- Top 10 danh sách các chủ đề mạng quan trọng



(a) provided service

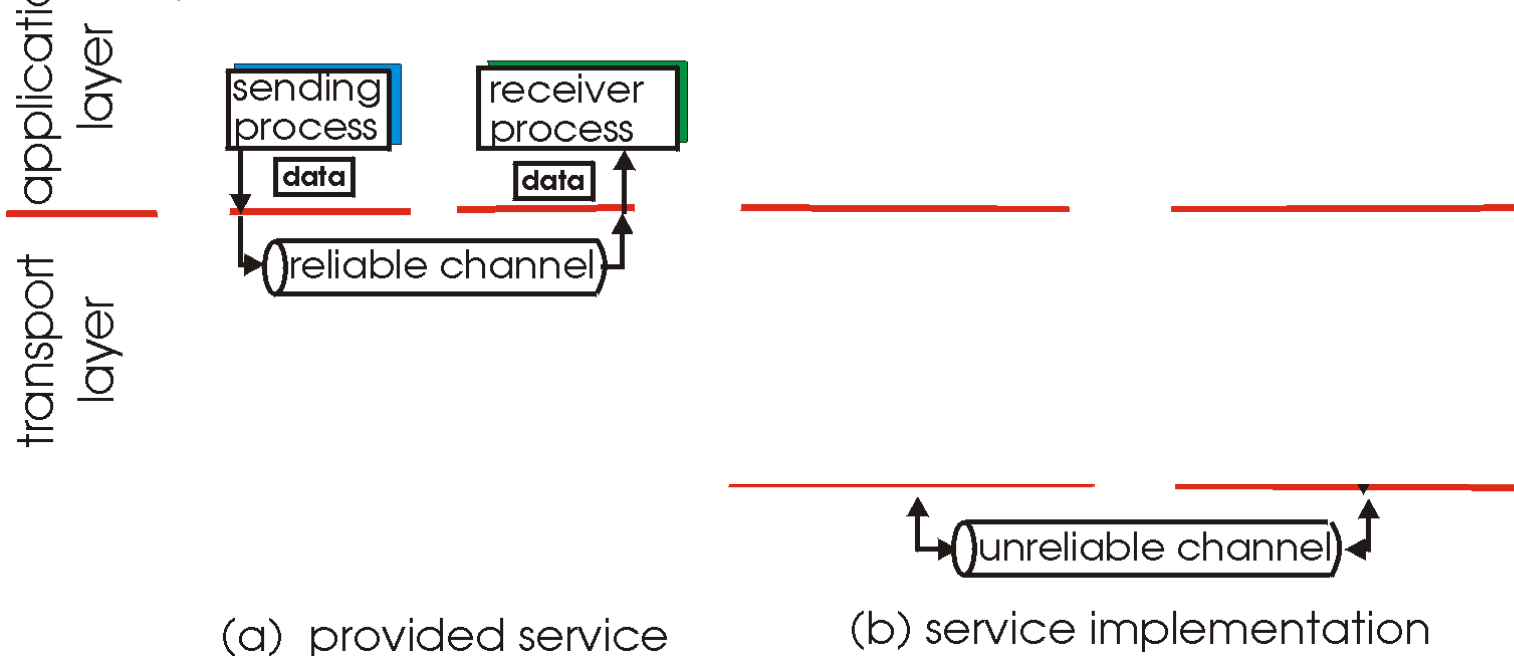
(b) service implementation

- Các đặc điểm của kênh truyền không tin cậy sẽ xác định sự phức tạp của giao thức truyền dữ liệu (data transfer protocol) (rdt)

# Các nguyên lý truyền dữ liệu tin cậy

❖ quan trọng trong các tầng application, transport và link

■ Top 10 danh sách các chủ đề mạng quan trọng

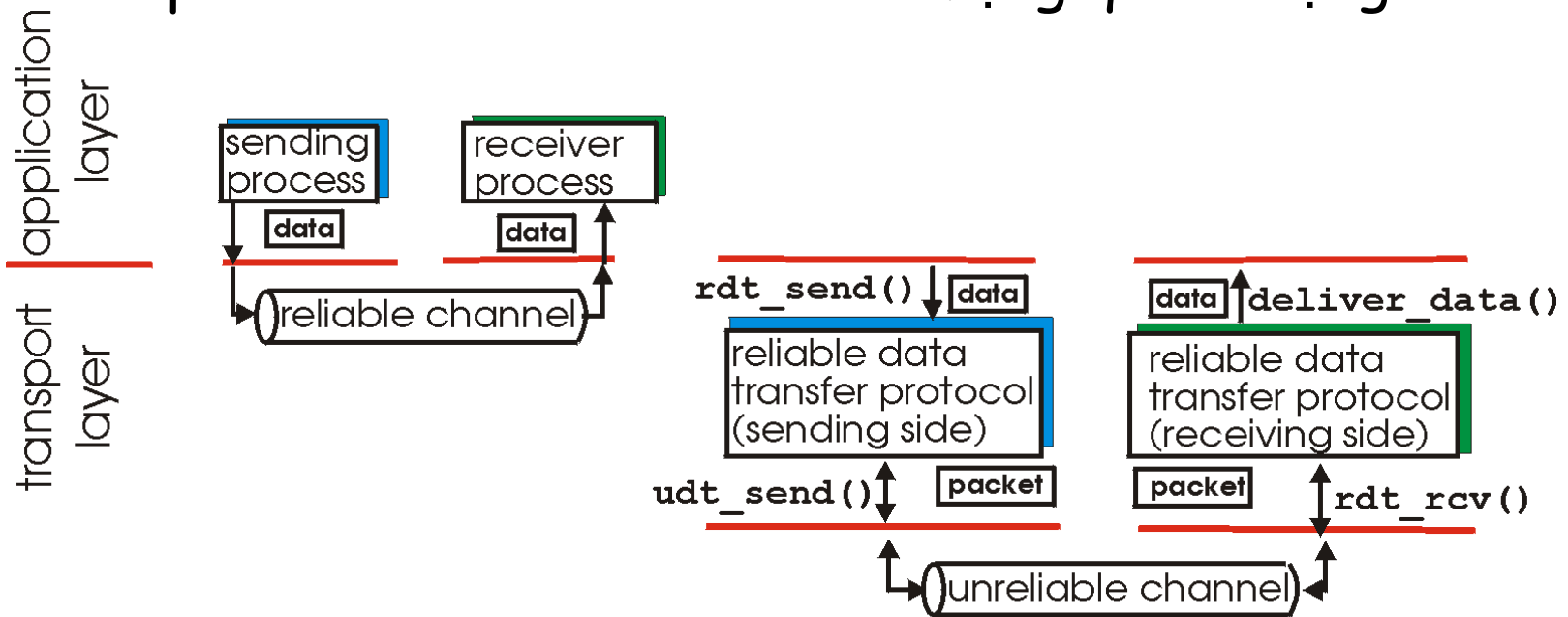


❖ Các đặc điểm của kênh truyền không tin cậy sẽ xác định sự phức tạp của giao thức truyền dữ liệu (data transfer protocol) (rdt)

# Các nguyên lý truyền dữ liệu tin cậy

quan trọng trong các tầng application, transport và link

- Top 10 danh sách các chủ đề mạng quan trọng



(a) provided service

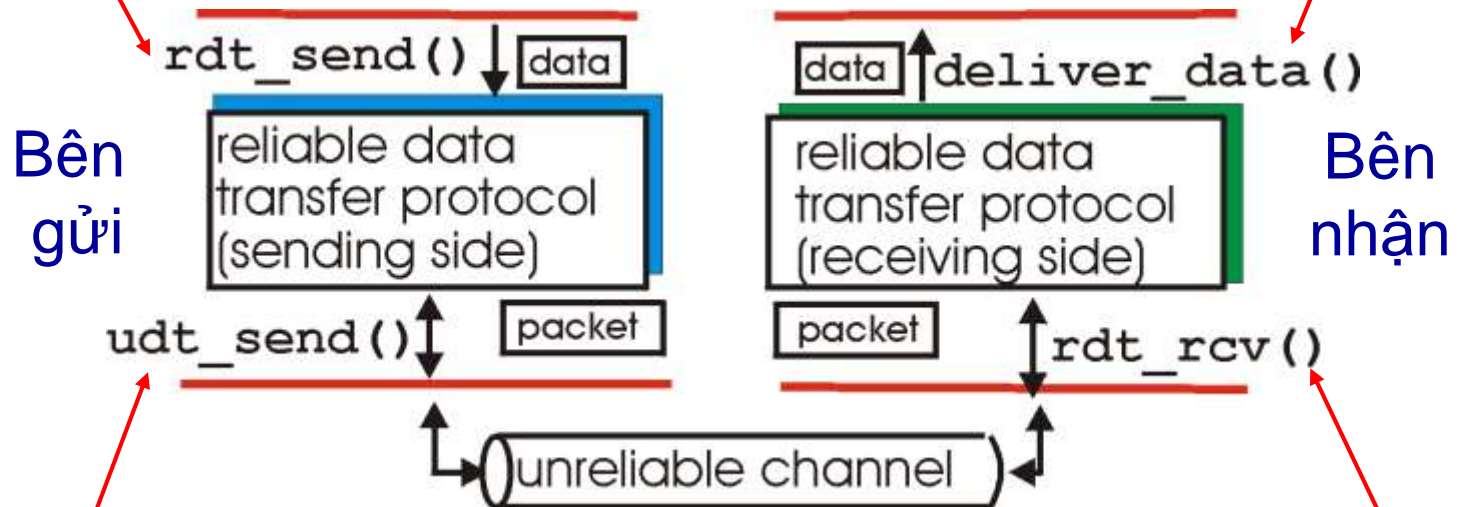
(b) service implementation

- Các đặc điểm của kênh truyền không tin cậy sẽ xác định sự phức tạp của giao thức truyền dữ liệu data transfer protocol (rdt)

# Truyền dữ liệu tin cậy: bắt đầu

**rdt\_send():** được gọi bởi tầng trên, (tầng application). Chuyển dữ liệu cần truyền đến lớp cao hơn ở bên nhận

**deliver\_data():** được gọi bởi rdt để chuyển dữ liệu đến tầng cao hơn



**udt\_send() :** được gọi bởi rdt, để truyền các gói trên kênh không tin cậy đến nơi nhận

**rdt\_rcv() :** được gọi khi gói dữ liệu đến kênh của bên nhận



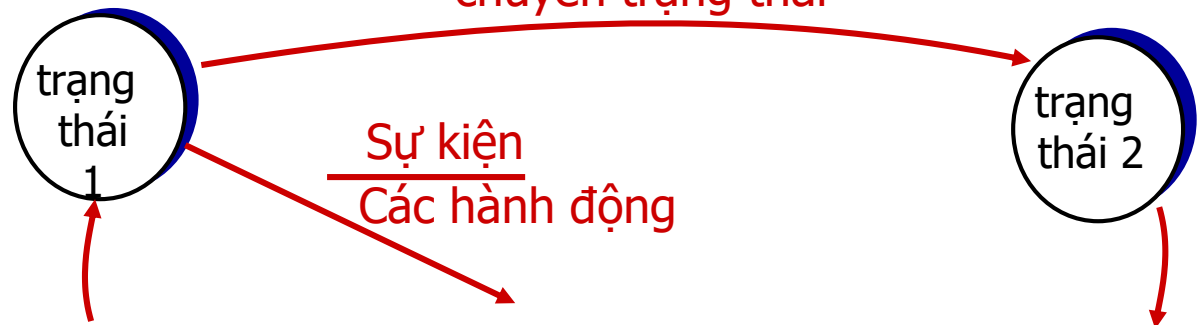
# Truyền dữ liệu tin cậy: bắt đầu

**Chúng ta sẽ:**

- ❖ Từng bước phát triển truyền dữ liệu tin cậy (rdt) bên phía người gửi và nhận
- ❖ Chỉ xem xét chuyển dữ liệu theo 1 hướng
  - Nhưng điều khiển thông tin sẽ theo cả 2 hướng!
- ❖ Sử dụng Máy trạng thái hữu hạn (finite state machines - FSM) để xác định bên gửi và nhận

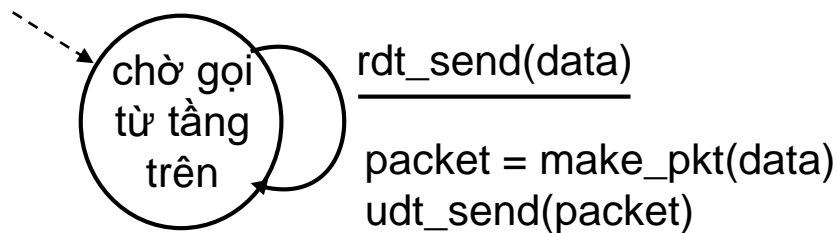
Sự kiện gây ra trạng thái truyền  
Các hành động được thực hiện trên  
chuyển trạng thái

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

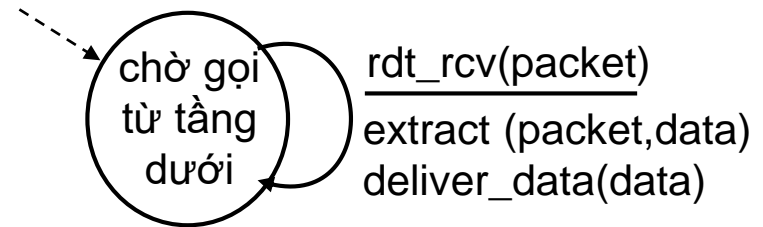


# rdt1.0: truyền tin cậy trên 1 kênh tin cậy

- ❖ Kênh cơ bản tin cậy hoàn toàn (underlying channel perfectly reliable)
  - không có bit lỗi
  - không mất mát gói
- ❖ Các FSMs riêng biệt cho bên gửi và nhận:
  - Bên gửi gửi dữ liệu vào kênh cơ bản (underlying channel)
  - Bên nhận đọc dữ liệu từ kênh cơ bản (underlying channel)



bên gửi



bên nhận

# rdt2.0: kênh với các lỗi

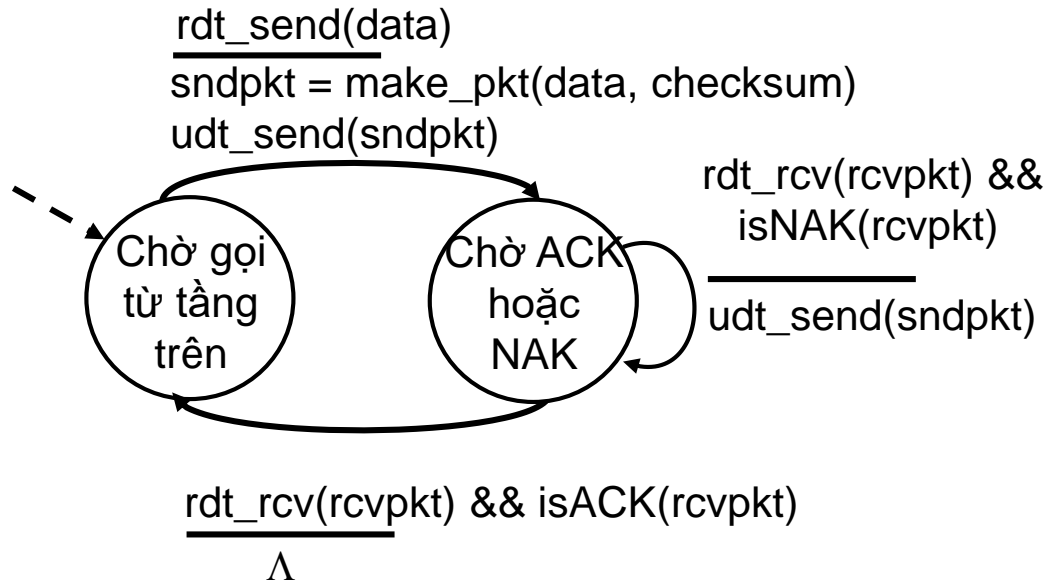
- ❖ Kênh cơ bản có thể đảo các bit trong packet
  - checksum để kiểm tra các lỗi
- ❖ *Câu hỏi: làm sao khôi phục các lỗi:*

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

# rdt2.0: kênh với các lỗi

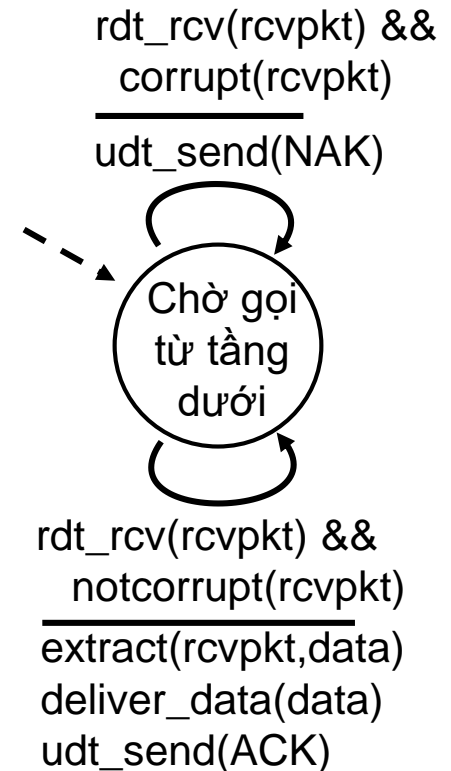
- ❖ Kênh cơ bản có thể đảo các bit trong packet
  - checksum để kiểm tra các lỗi
- ❖ *Câu hỏi*: làm sao khôi phục các lỗi:
  - *acknowledgements (ACKs)*: bên nhận thông báo rõ ràng cho bên gửi rằng packet được nhận thành công (OK)
  - *negative acknowledgements (NAKs)*: bên nhận thông báo rõ ràng cho bên gửi rằng packet đã bị lỗi
  - Bên gửi truyền lại gói nào được xác nhận là NAK
- ❖ Các cơ chế mới trong rdt2.0 (sau rdt1.0):
  - Phát hiện lỗi
  - Phản hồi: các thông điệp điều khiển (ACK, NAK) từ bên nhận đến bên gửi

# rdt2.0: đặc điểm kỹ thuật FSM

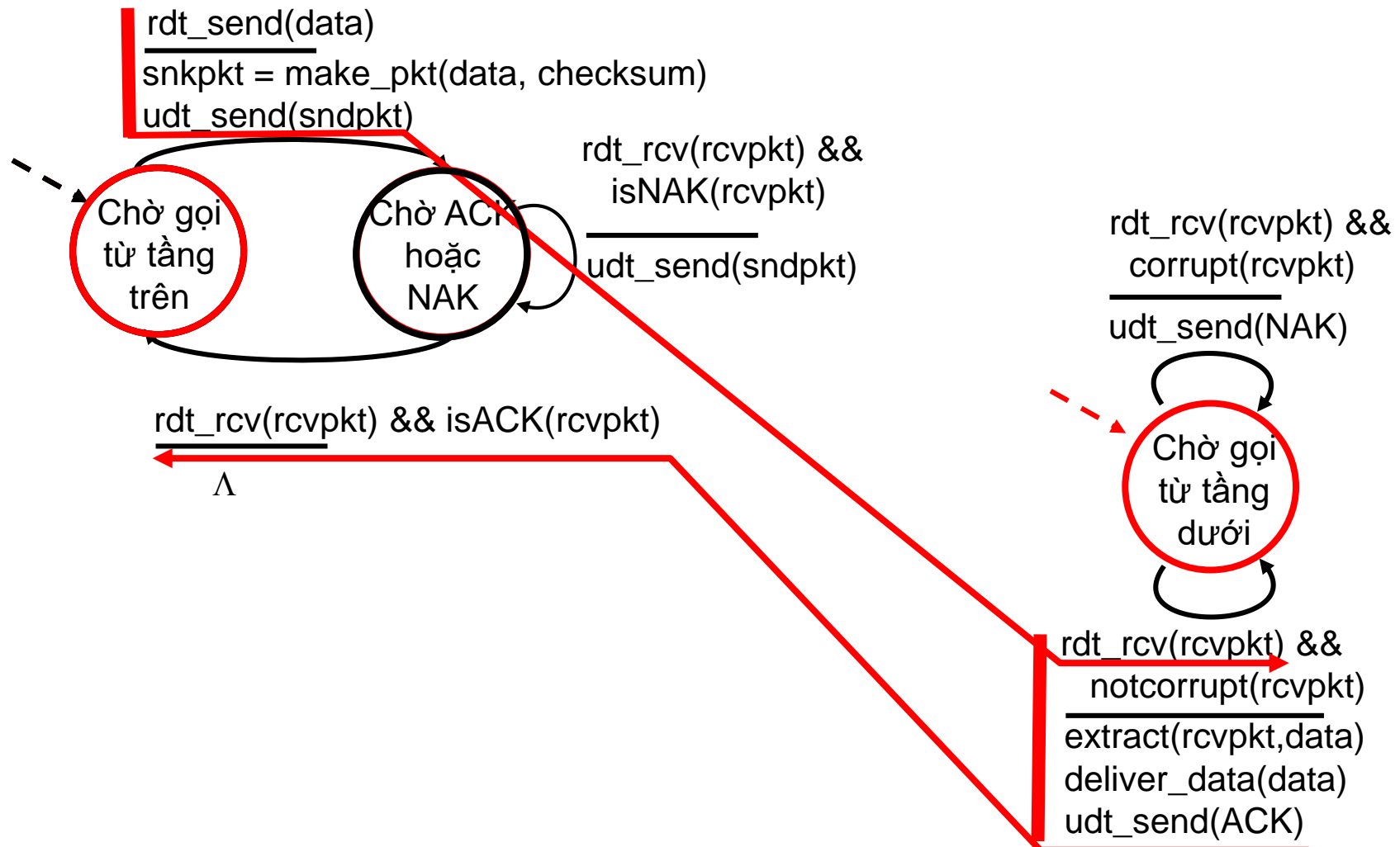


Bên gửi

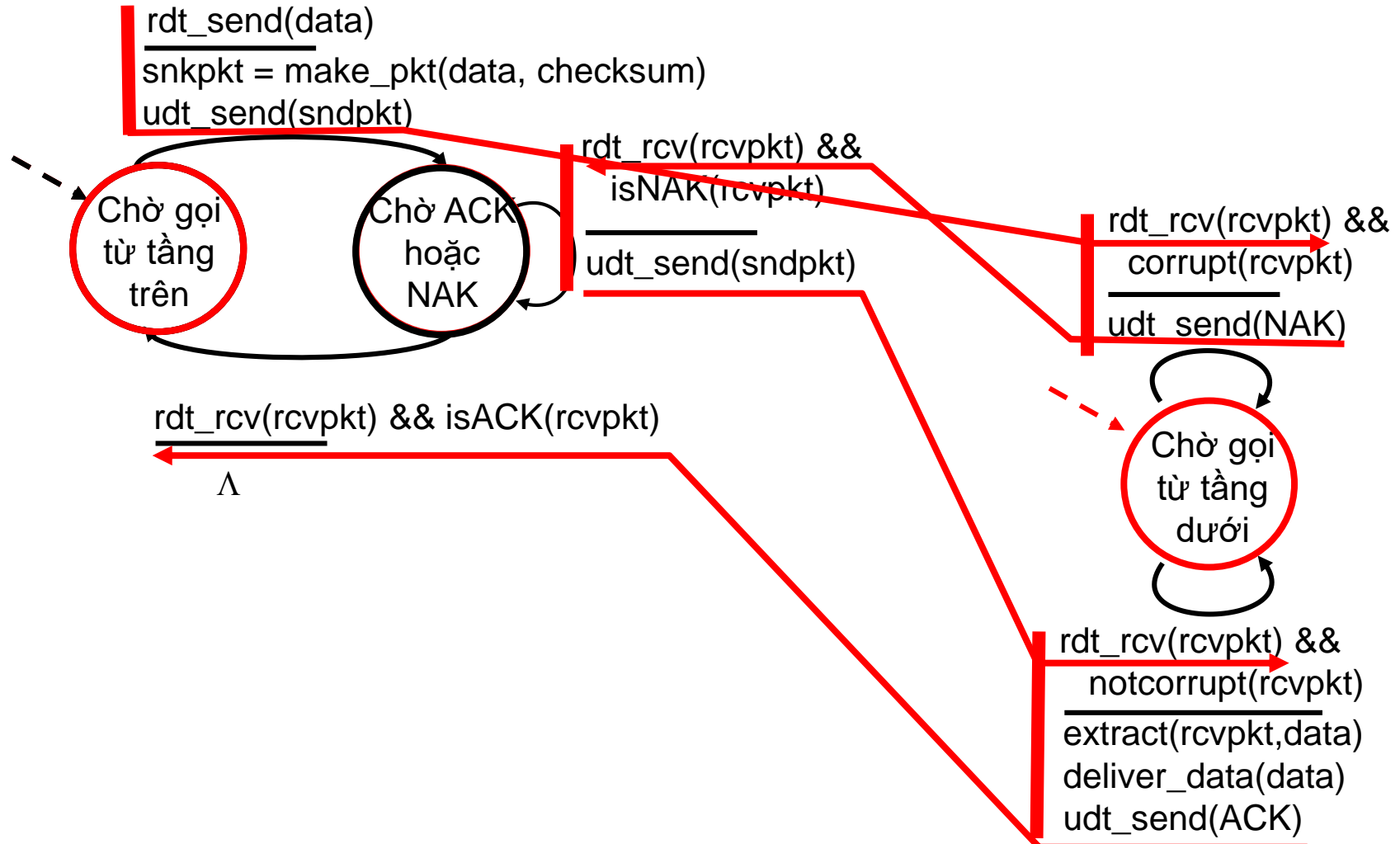
Bên nhận



# rdt2.0: hoạt động khi không lỗi



# rdt2.0: hoạt động khi có lỗi



# rdt2.0 có lỗi hỏng nghiêm trọng!

## Điều gì xảy ra nếu ACK/NAK bị hỏng?

- ❖ Bên gửi sẽ không biết điều gì đã xảy ra ở bên nhận!
- ❖ Không thể đơn phương truyền lại: có thể trùng lặp

## Xử lý trùng lặp:

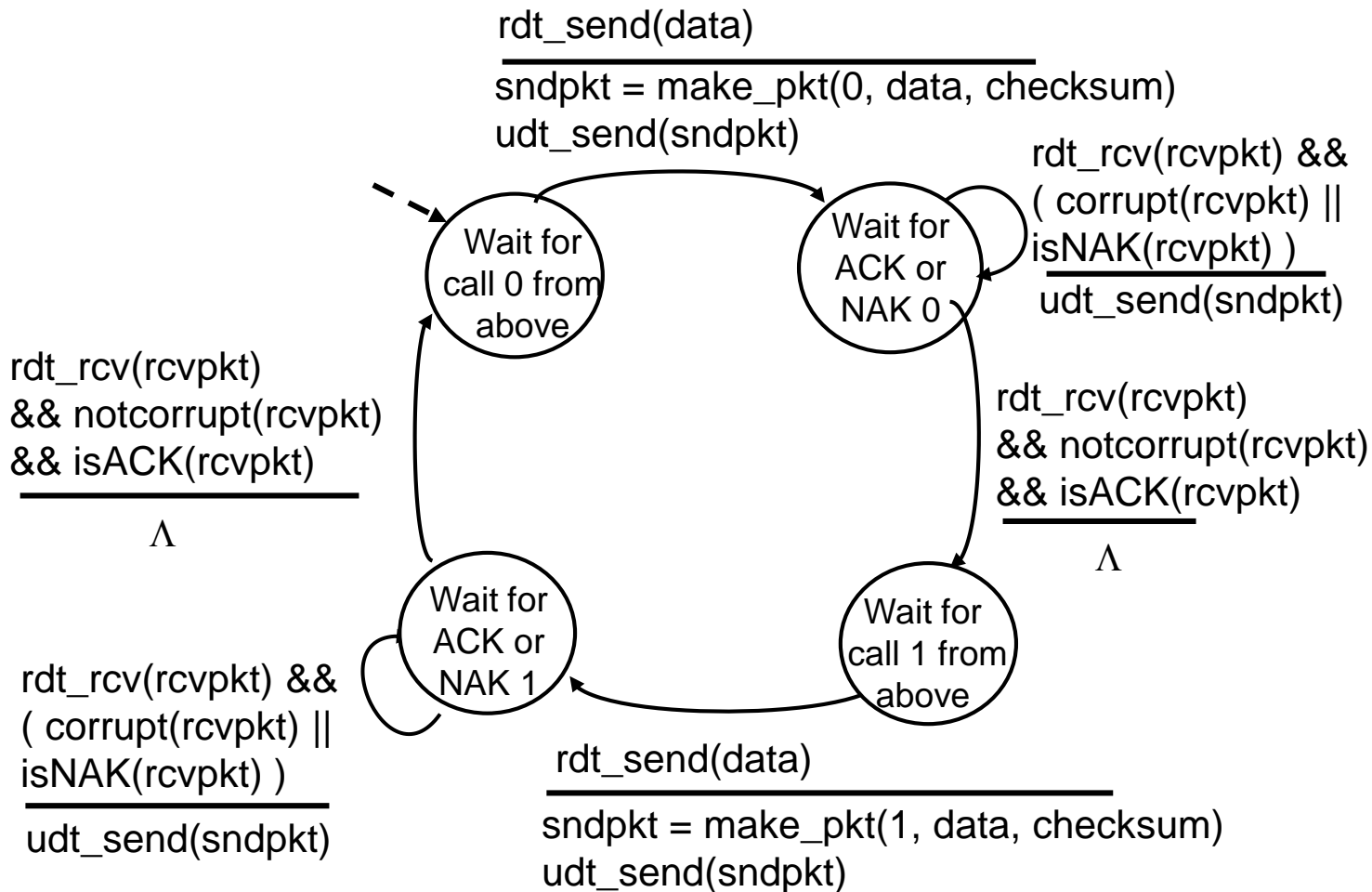
- ❖ Bên gửi truyền lại packet hiện thời nếu ACK/NAK bị hỏng
- ❖ Bên gửi thêm *số thứ tự* vào trong mỗi packet (*sequence number*)
- ❖ Bên nhận hủy packet bị trùng lặp

## dừng và chờ

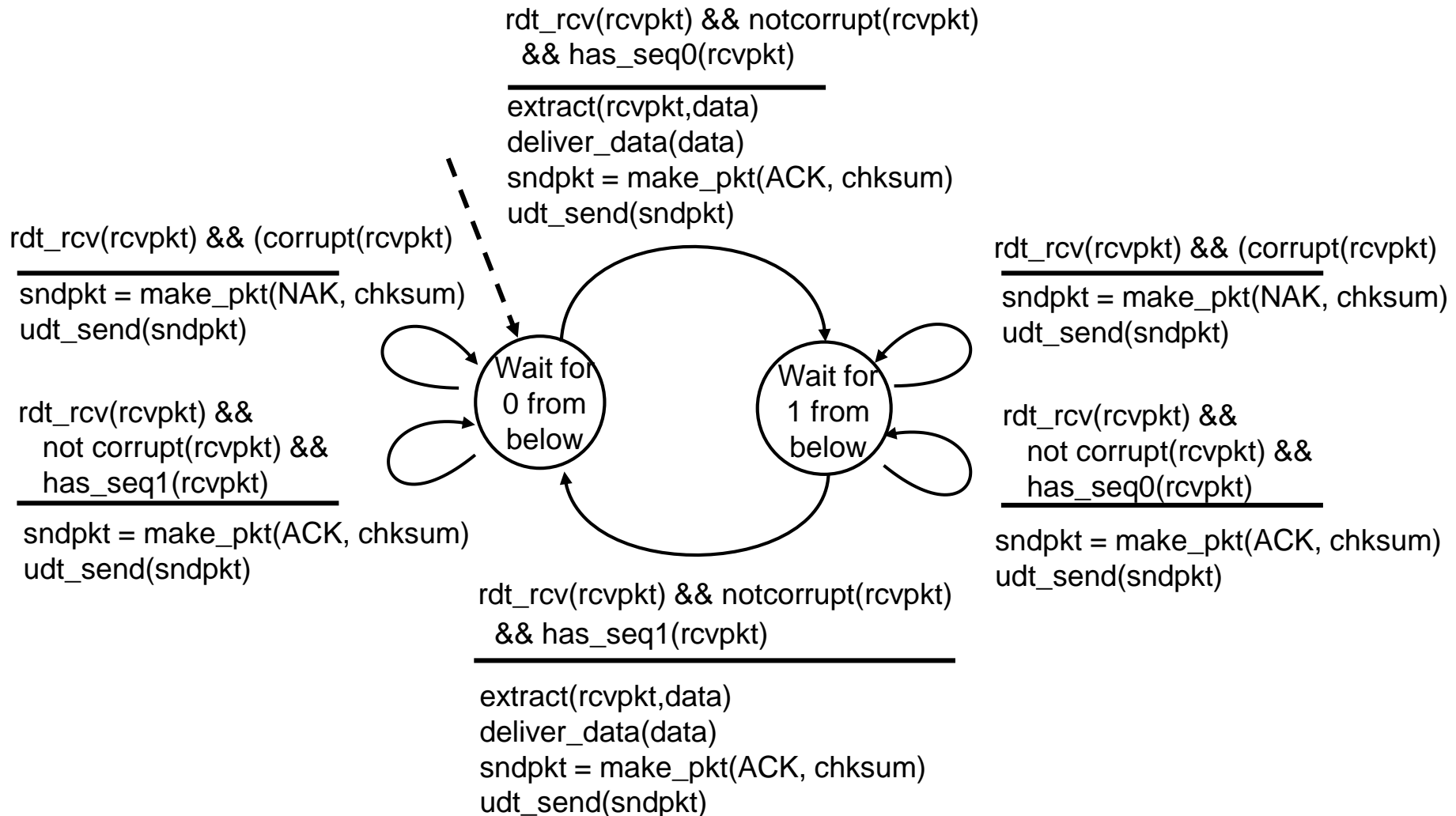
Bên gửi gửi một packet, sau đó chờ phản hồi từ bên nhận



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



# rdt2.1: bên nhận, xử lý các ACK/NAK bị hỏng



# rdt2.1: thảo luận

## Bên gửi:

- ❖ Số thứ tự (seq #) được thêm vào packet
- ❖ 2 số thứ tự (0,1) là đủ. Tại sao?
- ❖ Phải kiểm tra có hay không ACK/NAK vừa nhận bị hỏng
- ❖ Số trạng thái tăng lên 2 lần
  - Trạng thái phải "nhớ" xem packet "mong đợi" có số thứ tự là 0 hay 1

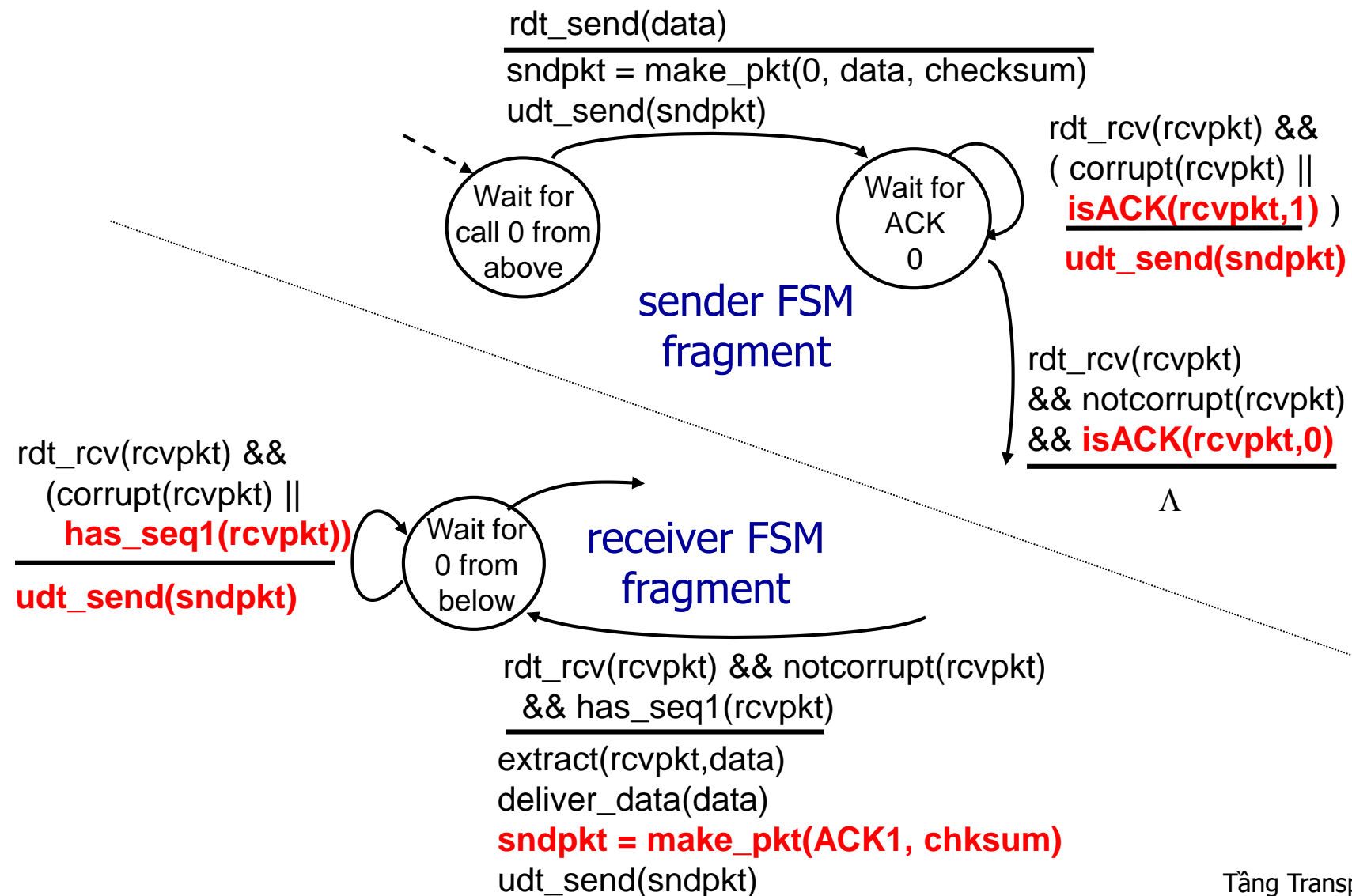
## Bên nhận:

- ❖ Phải kiểm tra có hay không gói vừa nhận trị trùng
  - Trạng thái chỉ rõ có hay không 0 hoặc 1 là số thứ tự của gói được mong chờ
- ❖ Chú ý: bên nhận có thể không biết ACK/NAK vừa rồi có được bên gửi nhận tốt hay không

# 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 rõ ràng chèn 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: các fragment bên nhận và gửi



# rdt3.0: các kênh với lỗi và mất mát

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

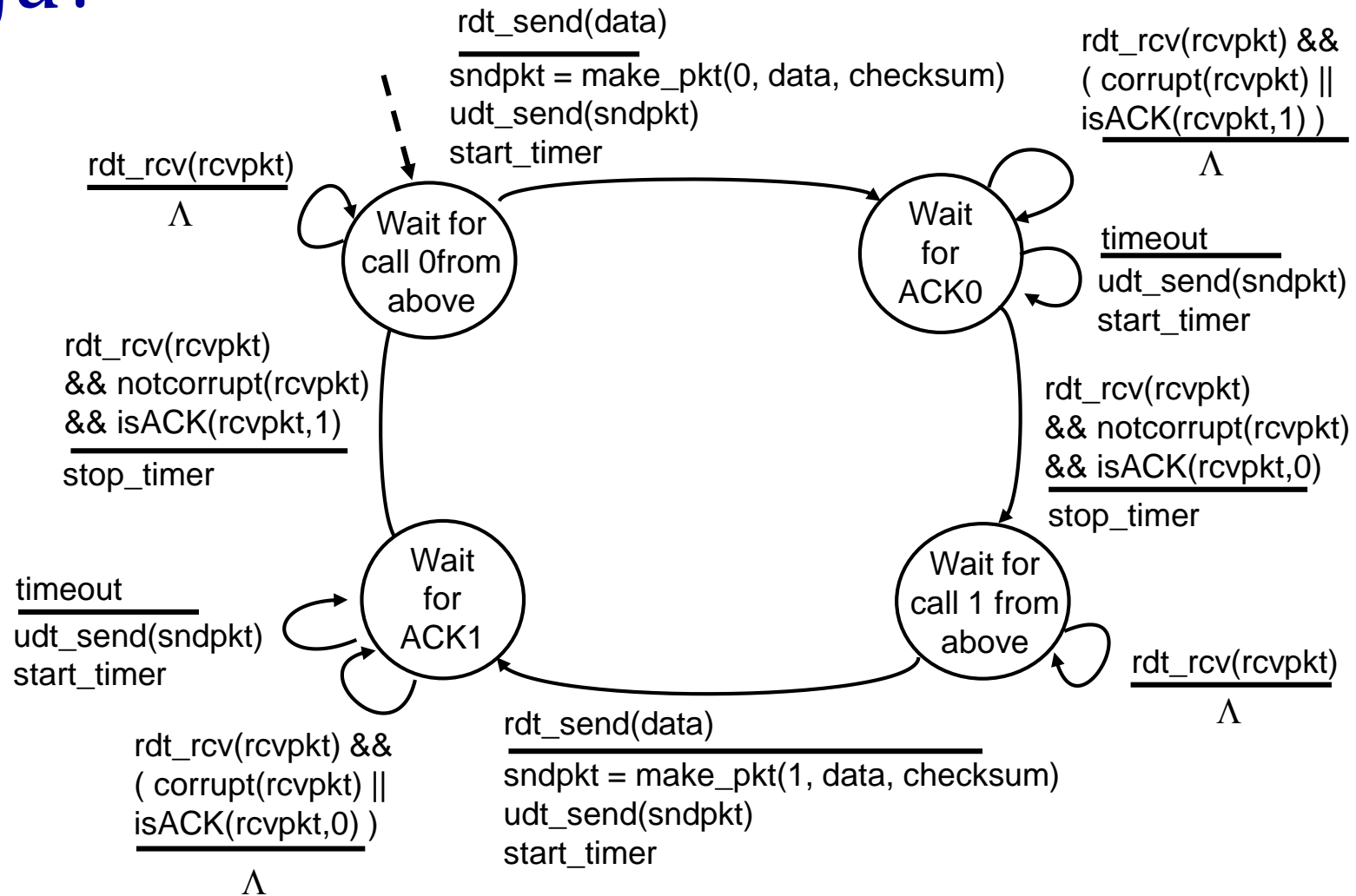
- checksum, số thứ tự, các ACK, việc truyền lại sẽ hỗ trợ...nhưng không đủ

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

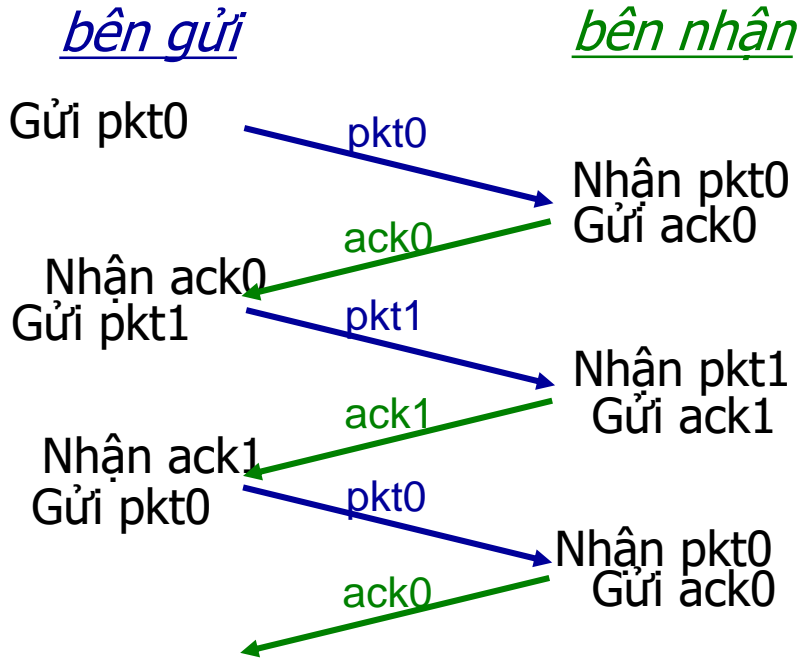
- ❖ Truyền lại nếu không có ACK được nhận trong khoảng thời gian này
- ❖ Nếu gói (hoặc ACK) chỉ trễ (không mất):
  - Việc truyền lại sẽ gây trùng, nhưng số thứ tự đã xử lý trường hợp này
  - Bên nhận phải xác định số thứ tự của gói vừa gửi ACK
- ❖ Yêu cầu bộ định thì đếm lùi

# rdt3.0 bên

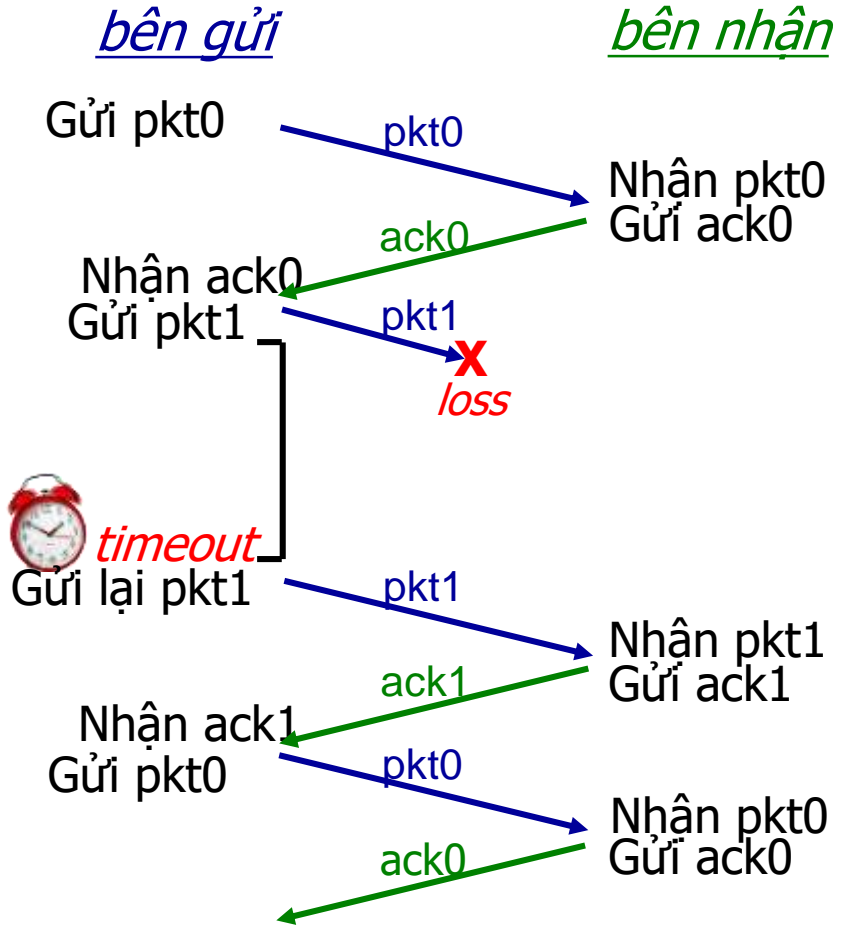
gửi



# Hành động của rdt3.0



(a) Không mất mát



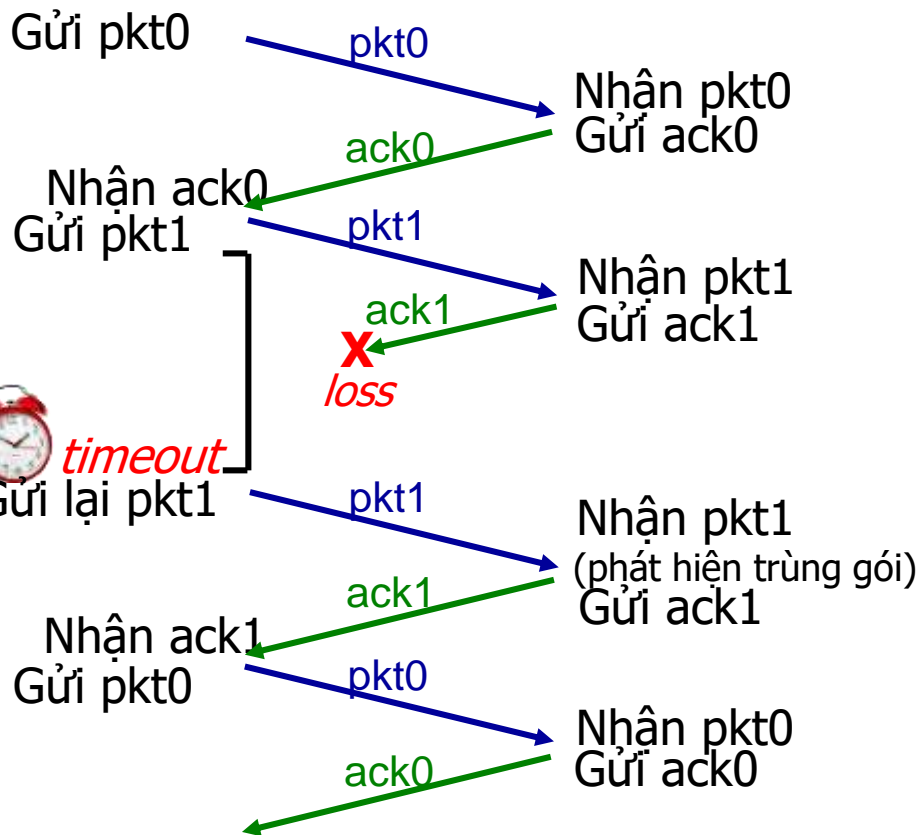
(b) **Mắt gói**



# Hành động của rdt3.0

bên gửi

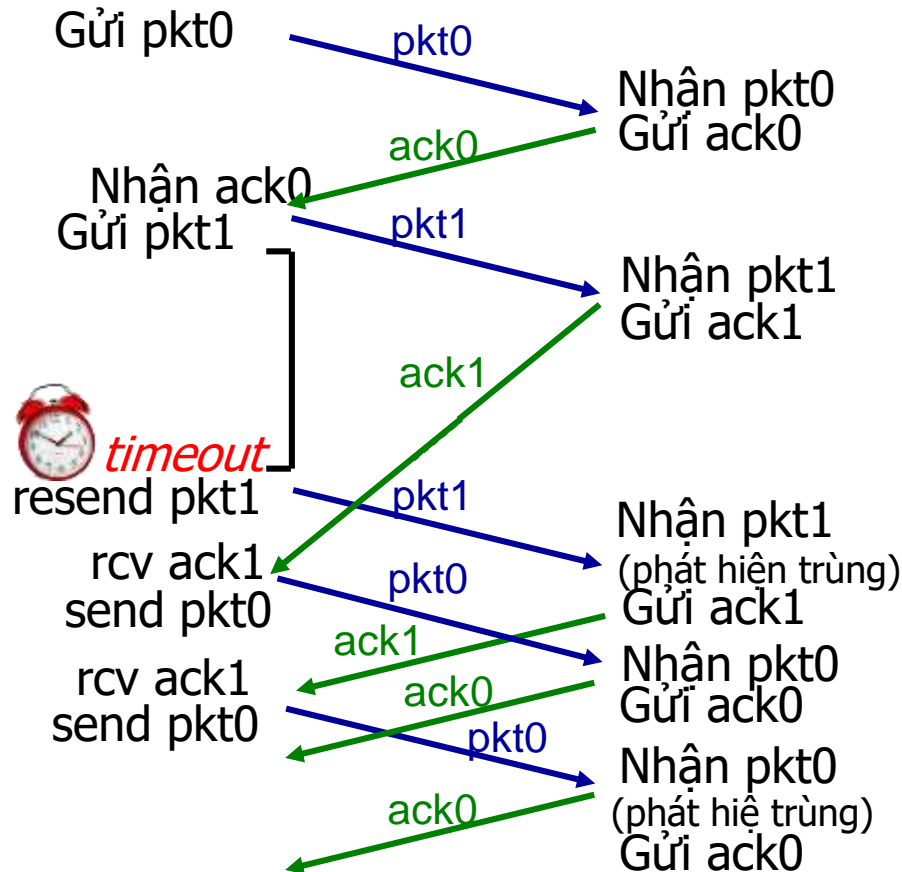
bên nhận



(c) Mất ACK

bên gửi

bên nhận



(d) premature timeout/ delayed ACK

# Hiệu suất của rdt3.0

- ❖ rdt3.0 làm việc được, nhưng đánh giá hiệu suất hơi rắc rối
- ❖ Ví dụ: đường link 1 Gbps, trễ lan truyền giữa 2 đầu cuối là 15 ms, gói 8000 bit:

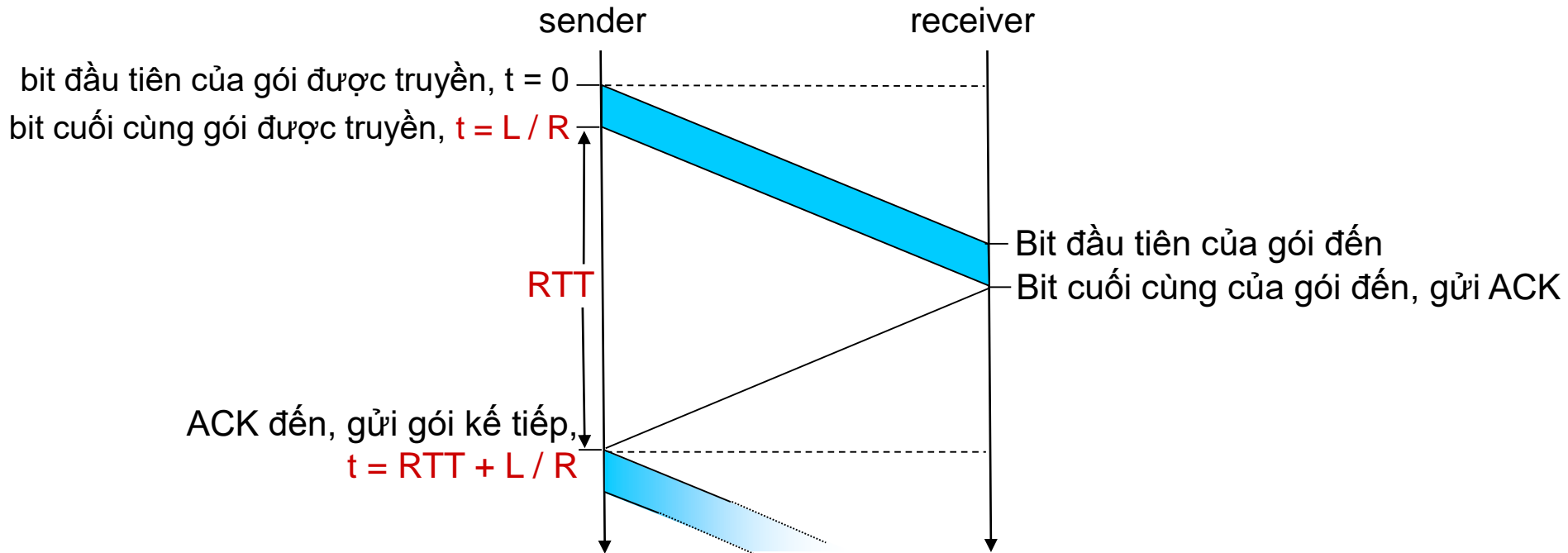
$$D_{\text{truyền}} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}$$

- $U_{\text{sender}}$ : **utilization** – fraction of time sender busy sending

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

- Nếu  $RTT=30$  msec, gói 1KB mỗi 30 msec: thông lượng 33kB/sec trên đường link 1 Gbps
- ❖ Giao thức network hạn chế việc sử dụng các tài nguyên vật lý!

# rdt3.0: hoạt động dừng-và-chờ

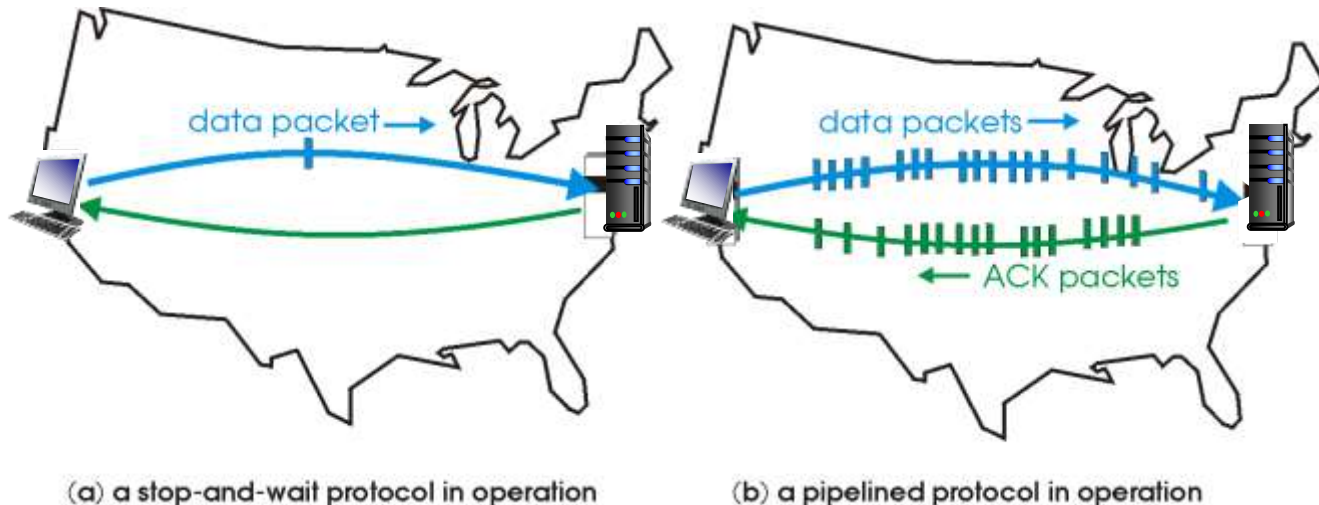


$$U_{\text{sender}} = \frac{L/R}{RTT + L/R} = \frac{.008}{30.008} = 0.00027$$

# Các giao thức Pipelined

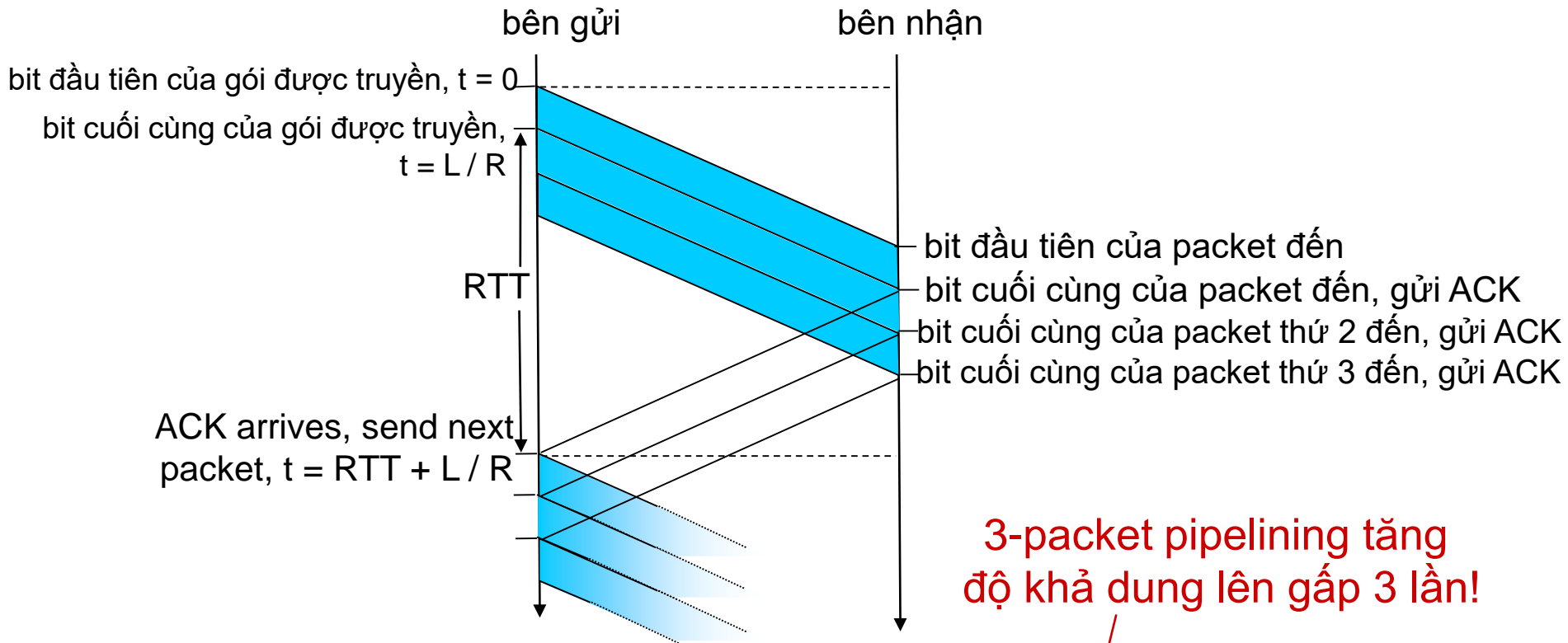
**pipelining:** bên gửi cho phép gửi nhiều gói đồng thời, không cần chờ báo nhận được

- Nhóm các số thứ tự phải được tăng dần
- Phải có bộ nhớ đệm tại nơi gửi và/hoặc nhận



❖ hai dạng phổ biến của các giao thức pipelined :  
*go-Back-N, lặp có lựa chọn*

# Pipelining: độ khả dụng tăng



$$U_{\text{sender}} = \frac{3L / R}{RTT + L / R} = \frac{.0024}{30.008} = 0.00081$$

# Pipelined protocols: tổng quan

## Go-back-N:

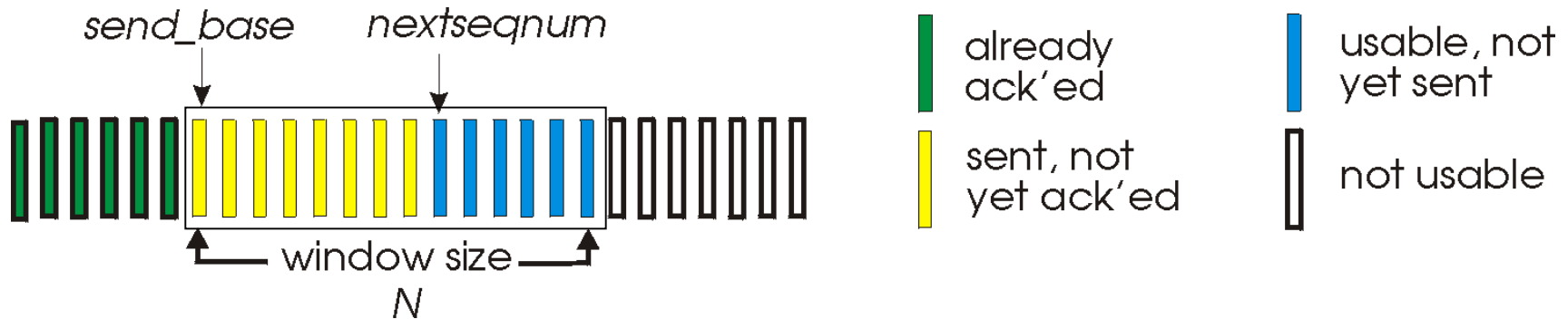
- ❖ Bên gửi có thể có đến N packet không cần ACK trong đường ống (pipeline)
- ❖ Bên nhận chỉ gửi *cumulative (tích lũy) ack*
  - Sẽ không thông báo nhận packet thành công nếu có một gián đoạn
- ❖ bên gửi có bộ định thì cho packet sớm nhất mà không cần ACK (oldest unacked packet)
  - Khi bộ định thì hết, truyền lại tất cả các packet mà không được ACK

## Lặp có lựa chọn (Selective Repeat):

- ❖ Bên gửi có thể có đến N packet không cần ACK trong đường ống (pipeline)
- ❖ Bên nhận gửi rcvr ack riêng biệt (*individual ack*) cho mỗi packet
- ❖ Bên nhận duy trì bộ định thì cho mỗi packet không được ACK
  - Khi bộ định thì của packet nào hết hạn, thì chỉ truyền lại packet không được ACK đó

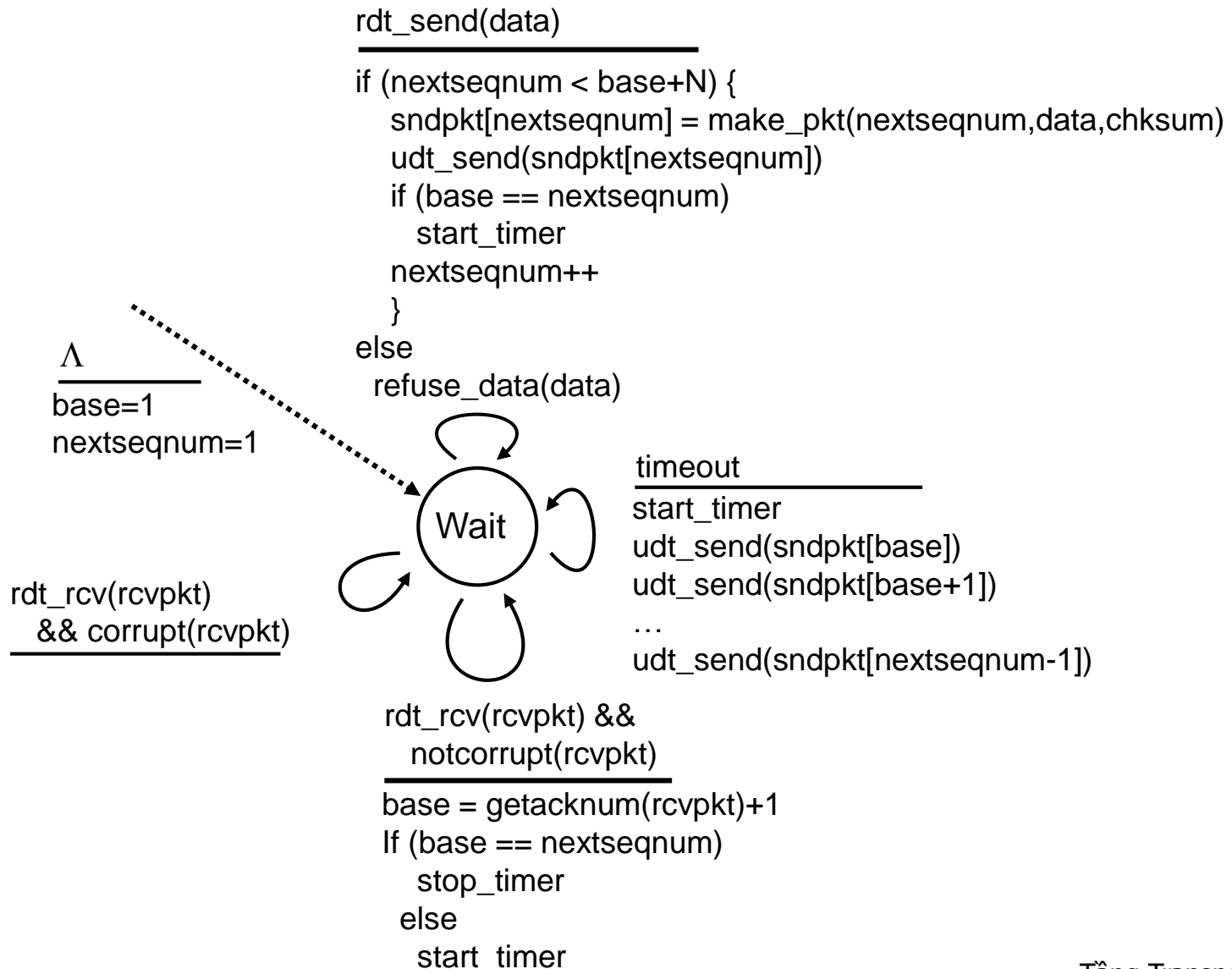
# Go-Back-N: bên gửi

- ❖ Số thứ tự k-bit trong header của packet
- ❖ "cửa sổ" ("window") lên đến N packet liên tiếp không cần ACK được cho phép



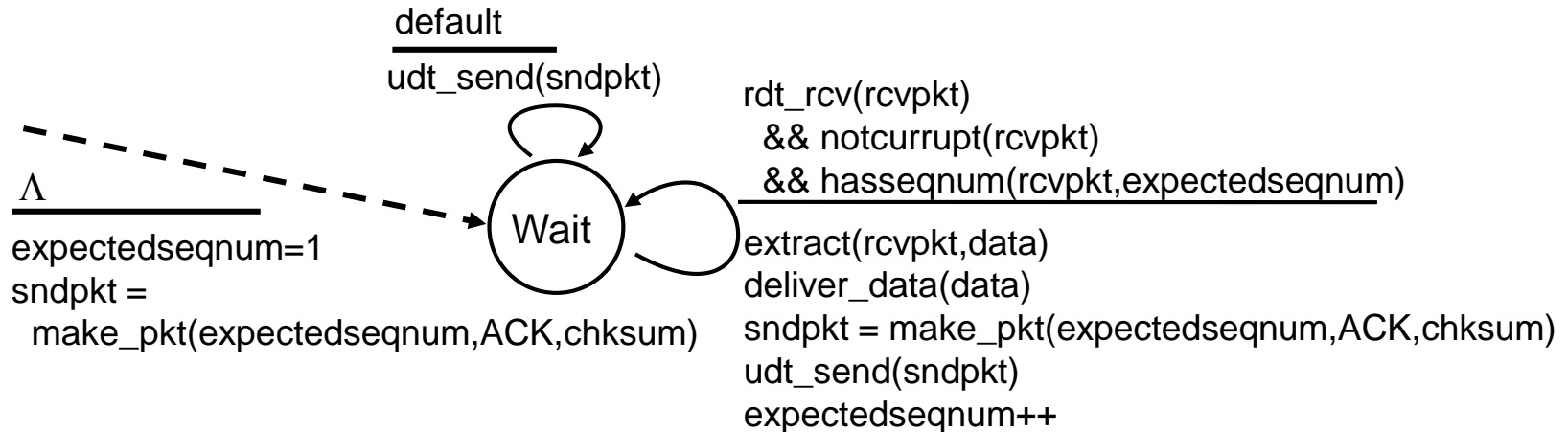
- ❖ ACK(n): thông báo nhận tất cả các packet lên đến n, bao gồm n số thứ tự - **"ACK tích lũy" ("cumulative ACK")**
  - Có thể nhận ACK trùng (xem bên nhận)
- ❖ Định thì cho packet sớm nhất đang trong tiến trình xử lý (oldest in-flight pkt)
- ❖ timeout(n): truyền lại packet n và tất cả các packet có số thứ tự cao hơn trong cửa sổ (window)

# GBN: sender extended FSM





# GBN: receiver extended FSM



ACK-duy nhất: luôn luôn gửi ACK cho gói đã nhận chính xác, với số thứ tự xếp hạng cao nhất (highest *in-order* seq #)

- Có thể sinh ra các ACK trùng nhau
- Chỉ cần nhớ **expectedseqnum**
- ❖ Packet không theo thứ tự(out-of-order pkt):
  - hủy discard (không đệm): *không bộ nhớ đệm bên nhận!*
  - Gửi lại với số thứ tự xếp hạng cao nhất

# Hoạt động GBN

sender window (N=4)

0 1 2 3 4 5 6 7 8  
 0 1 2 3 4 5 6 7 8  
 0 1 2 3 4 5 6 7 8  
 0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8  
 0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8  
 0 1 2 3 4 5 6 7 8  
 0 1 2 3 4 5 6 7 8  
 0 1 2 3 4 5 6 7 8

bên gửi

send pkt0  
 send pkt1  
 send pkt2  
 send pkt3  
 (wait)

rcv ack0, send pkt4  
 rcv ack1, send pkt5

ignore duplicate ACK



*pkt 2 timeout*

send pkt2  
 send pkt3  
 send pkt4  
 send pkt5

bên nhận

receive pkt0, send ack0  
 receive pkt1, send ack1

receive pkt3, discard,  
 (re)send ack1

receive pkt4, discard,  
 (re)send ack1

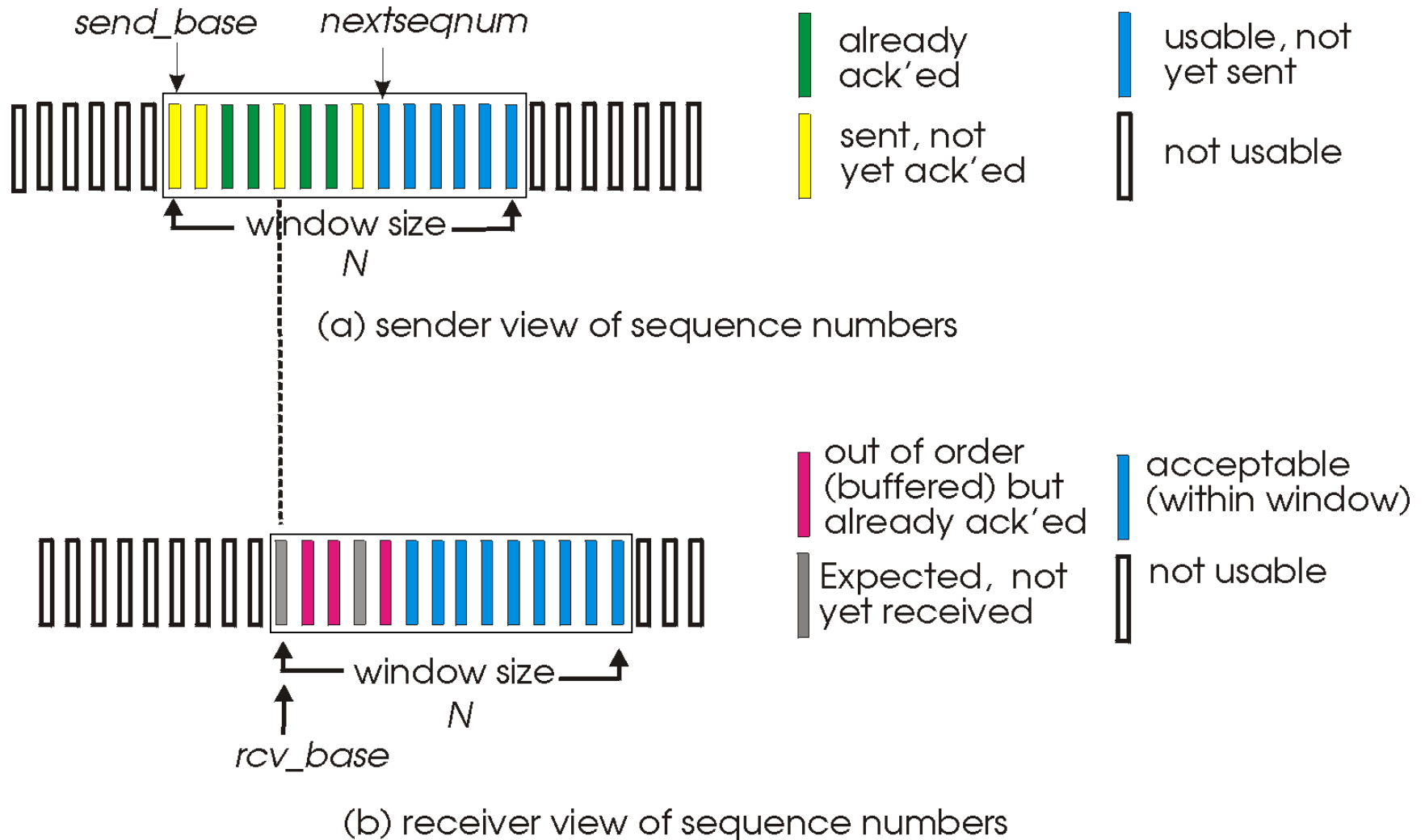
receive pkt5, discard,  
 (re)send ack1

rcv pkt2, deliver, send ack2  
 rcv pkt3, deliver, send ack3  
 rcv pkt4, deliver, send ack4  
 rcv pkt5, deliver, send ack5

# Lặp có lựa chọn (Selective repeat)

- ❖ Bên nhận thông báo đã nhận đúng tất cả từng gói một
  - Đệm các gói, khi cần thiết, cho sự vận chuyển trong thứ tự ngẫu nhiên đến tầng cao hơn
- ❖ Bên gửi chỉ gửi lại các packet nào mà ACK không được nhận
  - Bên gửi định thời cho mỗi packet không có gửi ACK
- ❖ Cửa sổ bên gửi (sender window)
  - $N$  số thứ tự liên tục
  - Hạn chế số thứ tự các gói không gửi ACK

# Lắp có lựa chọn: cửa sổ bên gửi và nhận



# Lắp có lựa chọn

## Bên gửi

Dữ liệu từ tầng trên:

- ❖ Nếu số thứ tự kế tiếp sẵn sàng trong cửa sổ, gửi packet

timeout(n):

- ❖ Gửi lại packet n, khởi động lại bộ định thì

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

- ❖ Đánh dấu packet n là đã được nhận
- ❖ Nếu gói không ACK có n nhỏ nhất, thì dịch chuyển cửa sổ base đến số thứ tự không ACK kế tiếp

## Bên nhận

Packet n trong [rcvbase, rcvbase+N-1]

- ❖ Gửi ACK(n)
- ❖ Không thứ tự: đệm
- ❖ Thứ tự: truyền (cũng truyền các gói đã đệm, có thứ tự), dịch chuyển cửa sổ đến gói chưa nhận kế tiếp

Packet n trong [rcvbase-N, rcvbase-1]

- ❖ ACK(n)

Ngược lại:

- ❖ Bỏ qua

# Hành động của lặp lại có lựa chọn

sender window (N=4)

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

Bên gửi

gửi pkt0

gửi pkt1

gửi pkt2

gửi pkt3

(đợi)

nhận ack0, gửi pkt4

nhận ack1, gửi pkt5

Ghi nhận ack3 đã đến



**pkt 2 timeout**

gửi pkt2

Ghi nhận ack4 đã đến

Ghi nhận ack5 đã đến

Bên nhận

nhận pkt0, gửi ack0

nhận pkt1, gửi ack1

nhận pkt3, buffer,  
gửi ack3

nhận pkt4, buffer,  
gửi ack4

nhận pkt5, buffer,  
gửi ack5

nhận pkt2; chuyển pkt2,  
pkt3, pkt4, pkt5; gửi ack2

*Q: việc gì xảy ra khi ack2 đến?*

# Lắp có lựa chọn: tình huống khó giải quyết

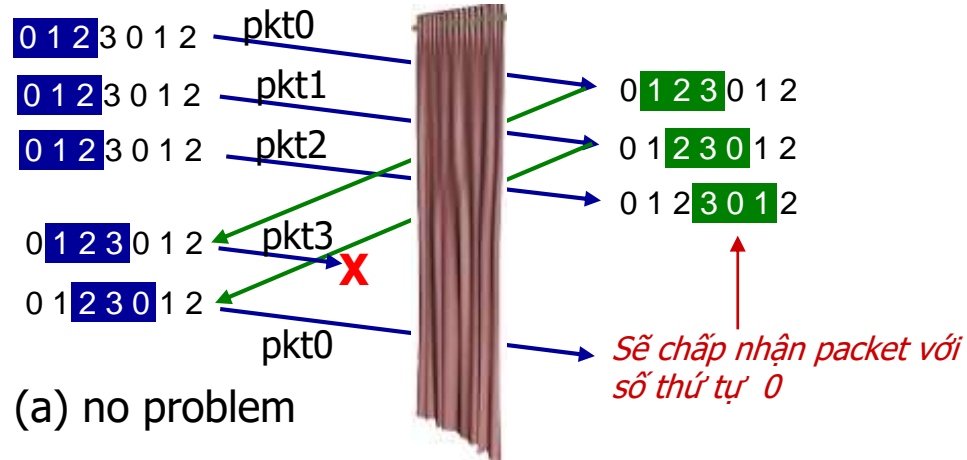
Ví dụ:

- ❖ Số thứ tự: 0, 1, 2, 3
- ❖ Kích thước cửa sổ=3
- ❖ Bên nhận không thấy sự khác nhau trong 2 tình huống!
- ❖ Dữ liệu trùng lặp được chấp nhận như dữ liệu mới (b)

**Q:** quan hệ giữa dãy số thứ tự và kích thước cửa sổ để tránh vấn đề (b)?

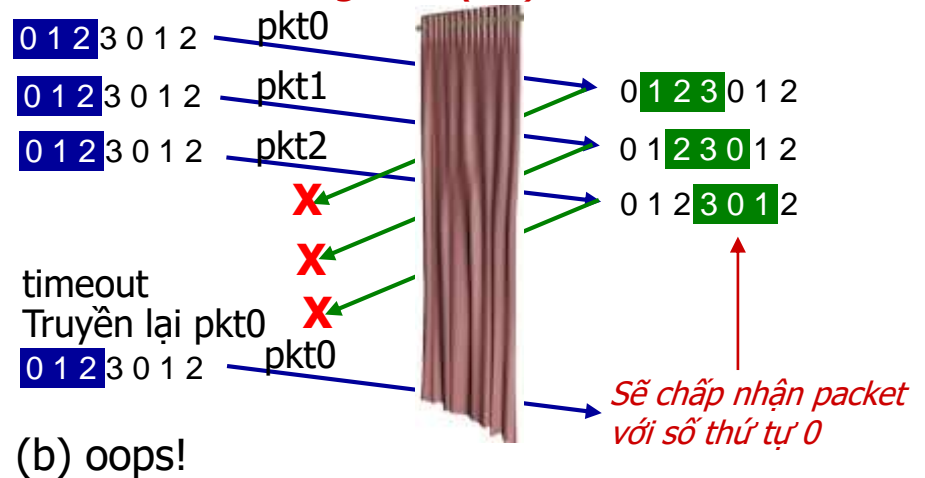
sender window  
(sau khi nhận)

receiver window  
(sau khi nhận)



Bên nhận không thể thấy phía bên gửi.  
Hành vi bên nhận như nhau trong cả 2 trường hợp!

*Có điều gì đó (rất) sai lầm!*



# Chương 3 Nội dung

3.1 các dịch vụ tầng  
Transport

3.2 multiplexing và  
demultiplexing

3.3 vận chuyển phi kết  
nối: UDP

3.4 các nguyên lý  
truyền dữ liệu tin  
cậy

3.5 vận chuyển hướng  
kết nối: TCP

- Cấu trúc segment
- Truyền dữ liệu tin cậy
- Điều khiển luồng (flow control)
- Quản lý kết nối

3.6 các nguyên lý về điều  
khiển tắc nghẽn

3.7 điều khiển tắc nghẽn  
TCP



# TCP: tổng quan RFCs: 793,1122,1323, 2018, 2581

## ❖ point-to-point:

- Một bên gửi, một bên nhận

## ❖ Tin cậy, dòng byte theo thứ tự (in-order byte stream):

- Không “ranh giới thông điệp” (“message boundaries”)

## ❖ pipelined:

- Điều khiển luồng và tắt nghẽn của TCP thiết lập kích thước cửa sổ (window size)

## ❖ Dữ liệu full duplex:

- Luồng dữ liệu đi 2 chiều trong cùng 1 kết nối
- MSS: kích thước tối đa của segment (maximum segment size)

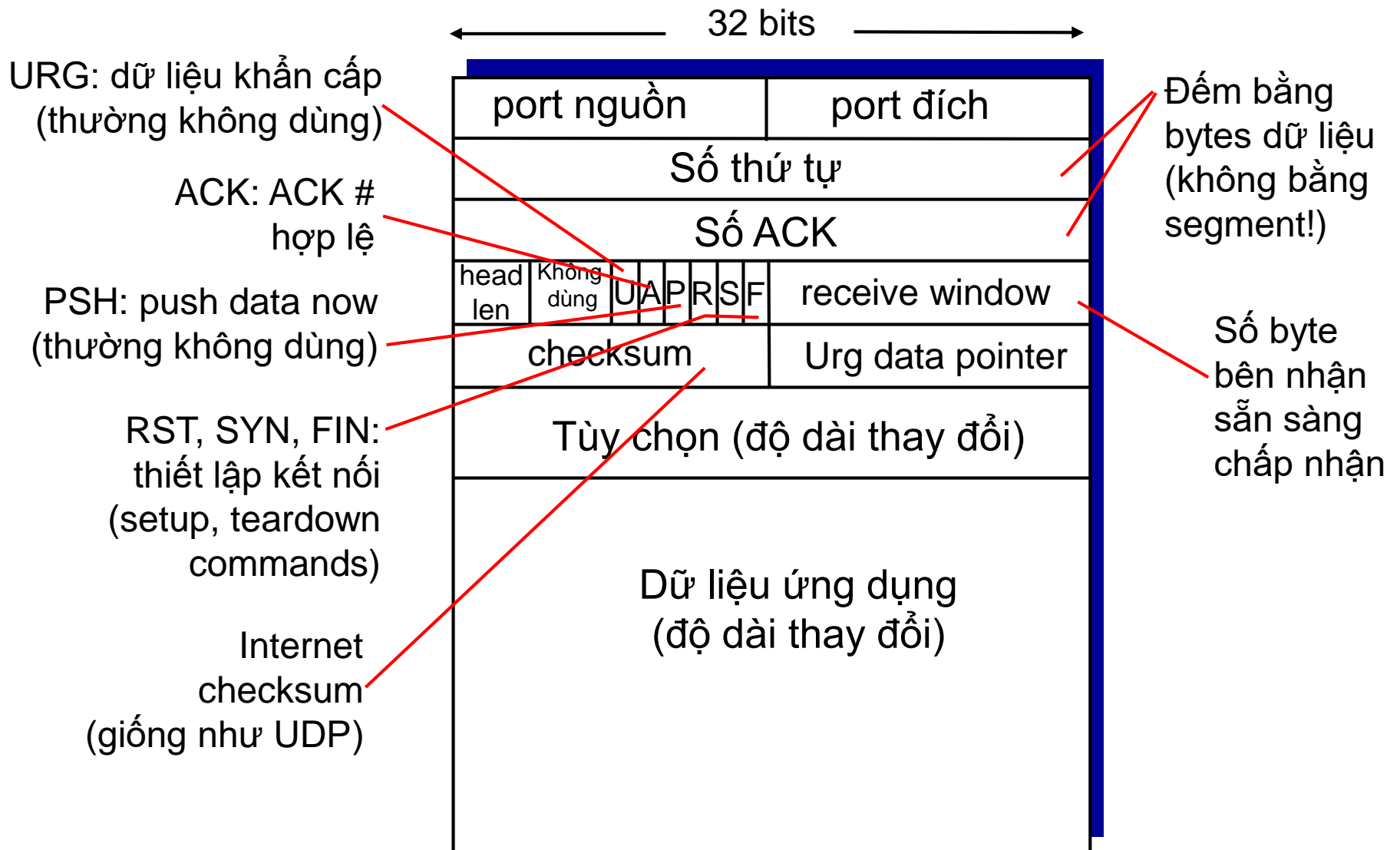
## ❖ Hướng kết nối:

- Bắt tay (trao đổi các thông điệp điều khiển) khởi tạo trạng thái bên gửi và nhận trước khi trao đổi dữ liệu

## ❖ Luồng được điều khiển:

- Bên gửi sẽ không áp đảo bên nhận

# Cấu trúc segment TCP segment



# Số thứ tự TCP và ACK

## Các số thứ tự:

- Dòng byte “đánh số” byte đầu tiên trong dữ liệu của segment

## Các ACK:

- số thứ tự của byte kế tiếp được mong đợi từ phía bên kia
- ACK tích lũy

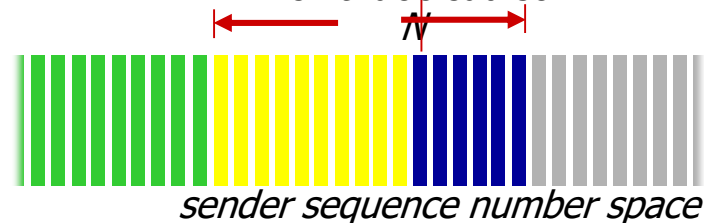
**Hỏi:** làm thế nào để bên nhận xử lý các segment không theo thứ tự

- Trả lời: TCP không đề cập, tùy thuộc người thực hiện

Segment đi ra từ bên gửi

|            |             |
|------------|-------------|
| port nguồn | port đích   |
| số thứ tự  |             |
| số ACK     |             |
|            | rwnd        |
| checksum   | urg pointer |

kích thước cửa sổ



sent  
ACKed

sent, not-  
yet ACKed  
("in-  
flight")

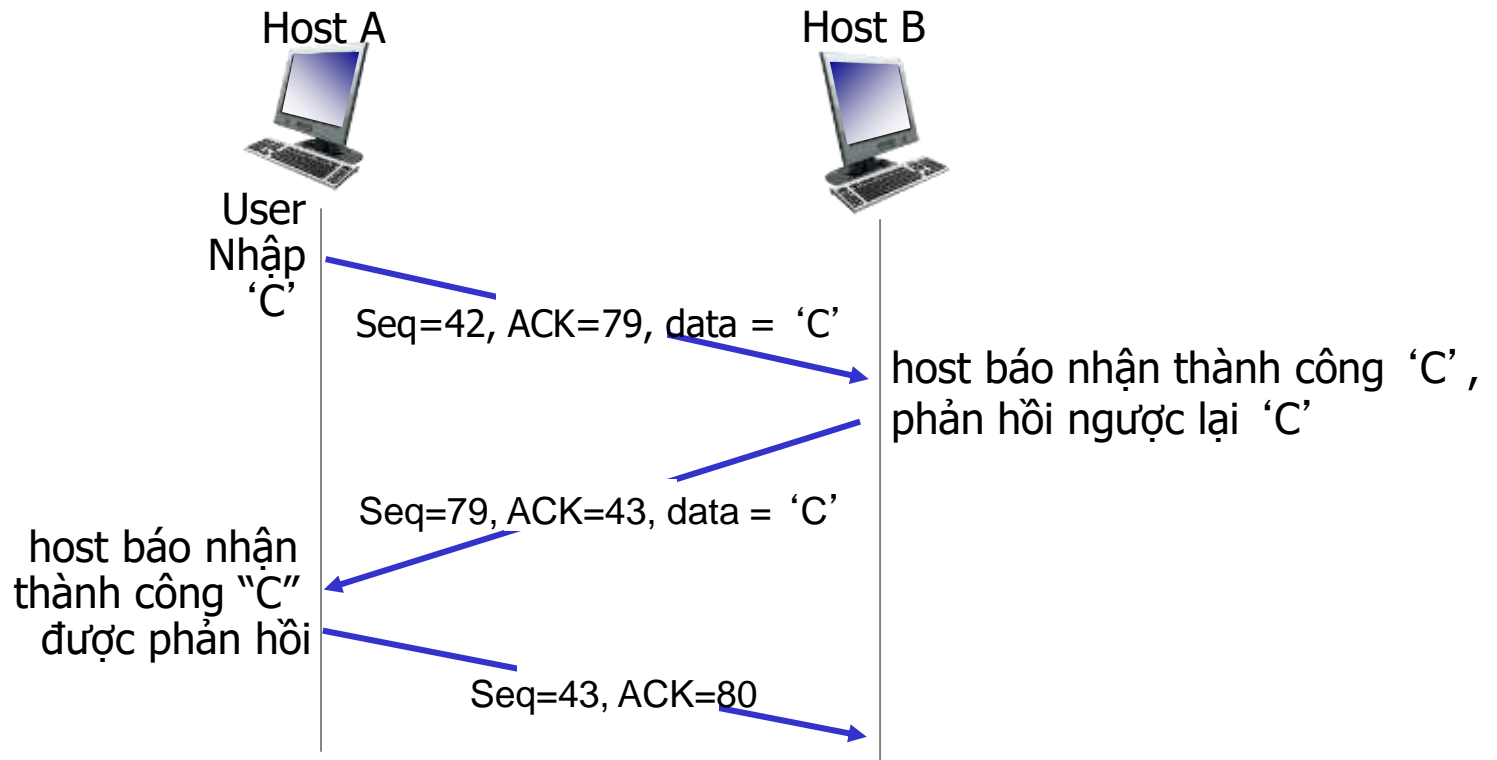
usable  
but not  
yet sent

not  
usable

Segment vào, đến bên gửi

|            |             |
|------------|-------------|
| port nguồn | port đích   |
| số thứ tự  |             |
| số ACK     |             |
|            | rwnd        |
| checksum   | urg pointer |

# Số thứ tự TCP và ACK



Tình huống telnet đơn giản

# TCP round trip time và timeout

**Hỏi:** làm cách nào để thiết lập giá trị TCP timeout?

- ❖ Dài hơn RTT
  - Nhưng RTT thay đổi
- ❖ Quá ngắn: timeout sớm, không cần thiết truyền lại
- ❖ Quá dài: phản ứng chậm đối với việc mất segment

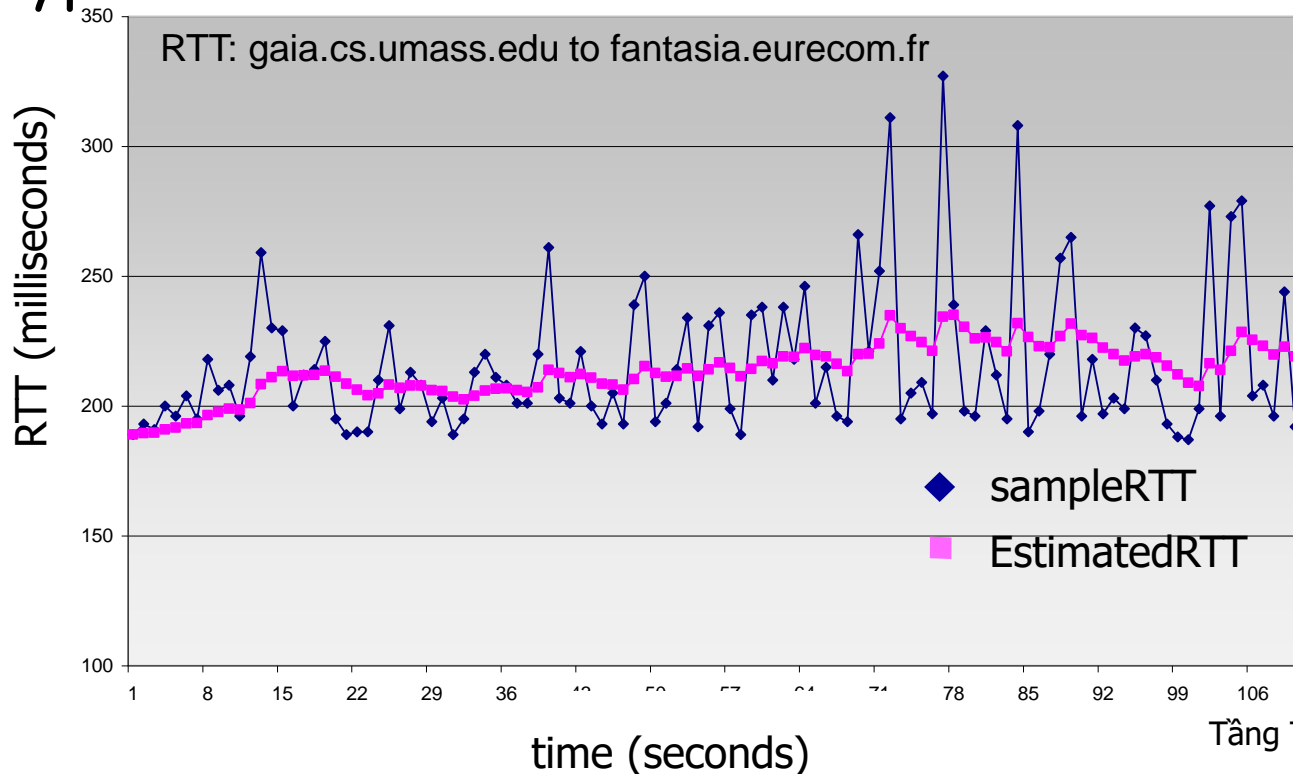
**Q:** làm cách nào để ước lượng RTT?

- ❖ **SampleRTT:** thời gian được đo từ khi truyền segment đến khi báo nhận ACK
  - Lờ đi việc truyền lại
- ❖ **SampleRTT** sẽ thay đổi, muốn RTT được ước lượng “mượt hơn”
  - Đo lường trung bình của một số giá trị vừa xảy ra, không chỉ **SampleRTT** hiện tại

# TCP round trip time và timeout

$$\text{EstimatedRTT} = (1 - \alpha) * \text{EstimatedRTT} + \alpha * \text{SampleRTT}$$

- ❖ Đường trung bình dịch chuyển hàm mũ (exponential weighted moving average)
- ❖ ảnh hưởng của mẫu đã xảy ra sẽ làm giảm tốc độ theo cấp số nhân
- ❖ typical value:  $\alpha = 0.125$



# TCP round trip time và timeout

- ❖ Khoảng thời gian timeout (timeout interval):  
**EstimatedRTT** cộng với “biên an toàn”
  - Sự thay đổi lớn trong **EstimatedRTT** -> an toàn biên lớn hơn
- ❖ Ước lượng độ lệch **SampleRTT** từ **EstimatedRTT**:

$$\text{DevRTT} = (1-\beta) * \text{DevRTT} + \beta * |\text{SampleRTT} - \text{EstimatedRTT}|$$

(typically,  $\beta = 0.25$ )

$$\text{TimeoutInterval} = \text{EstimatedRTT} + 4 * \text{DevRTT}$$



↑  
estimated RTT

↑  
“biên an toàn”

# Chương 3 Nội dung

3.1 các dịch vụ tầng  
Transport

3.2 multiplexing và  
demultiplexing

3.3 vận chuyển phi kết  
nối: UDP

3.4 các nguyên lý  
truyền dữ liệu tin  
cậy

3.5 vận chuyển hướng  
kết nối: TCP

- Cấu trúc segment
- Truyền dữ liệu tin cậy
- Điều khiển luồng (flow control)
- Quản lý kết nối

3.6 các nguyên lý về điều  
khiển tắc nghẽn

3.7 điều khiển tắc nghẽn  
TCP



# TCP truyền dữ liệu tin cậy

- ❖ TCP tạo dịch vụ rdt trên dịch vụ không tin cậy của IP
  - Các segment pipelined
  - Các ack tích lũy
  - Bộ định thì truyền lại đơn (single retransmission timer)
- ❖ Việc truyền lại được kích hoạt bởi:
  - Sự kiện timeout
  - Các ack bị trùng

Lúc đầu khảo sát TCP đơn giản ở bên gửi:

- Lờ đi các ack bị trùng
- Lờ đi điều khiển luồng và điều khiển tắt nghẽn

# TCP các sự kiện bên gửi:

*Dữ liệu được nhận từ ứng dụng:*

- ❖ Tạo segment với số thứ tự
- ❖ Số thứ tự là số byte-stream của byte dữ liệu đầu tiên trong segment
- ❖ Khởi động bộ định thì nếu chưa chạy
  - Xem bộ định thì như là đối với segment sớm nhất không được ACK
  - Khoảng thời gian hết hạn: **TimeoutInterval**

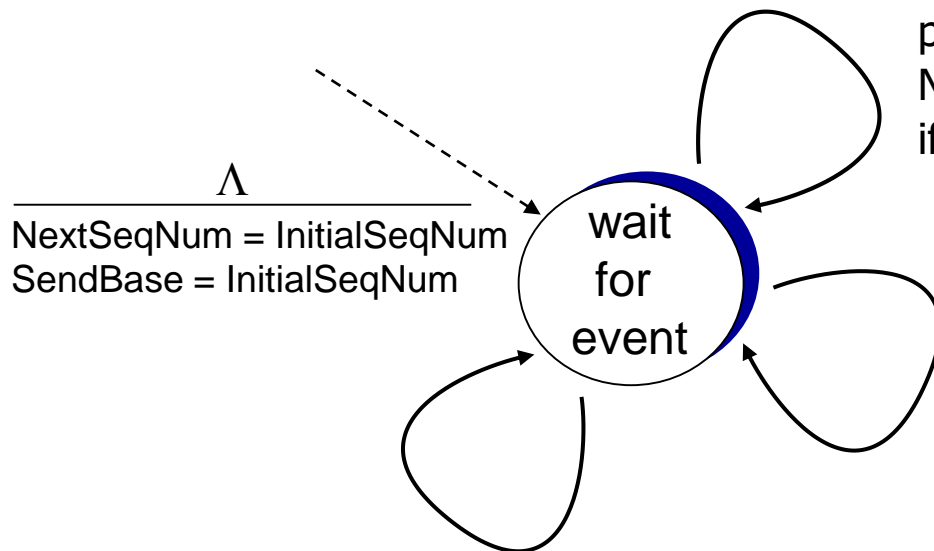
*timeout:*

- ❖ Gửi lại segment nào gây ra timeout
- ❖ Khởi động lại bộ định thì

*ack được nhận:*

- ❖ Nếu ack thông báo đã các segment không được ACK trước đó
  - Cập nhật những gì được biết là đã được nhận thành công
  - Khởi động lại bộ định thì nếu có các segment vẫn chưa được thông báo nhận thành công

# TCP bên gửi (đơn giản)



NextSeqNum = InitialSeqNum  
SendBase = InitialSeqNum

ACK received, with ACK field value y

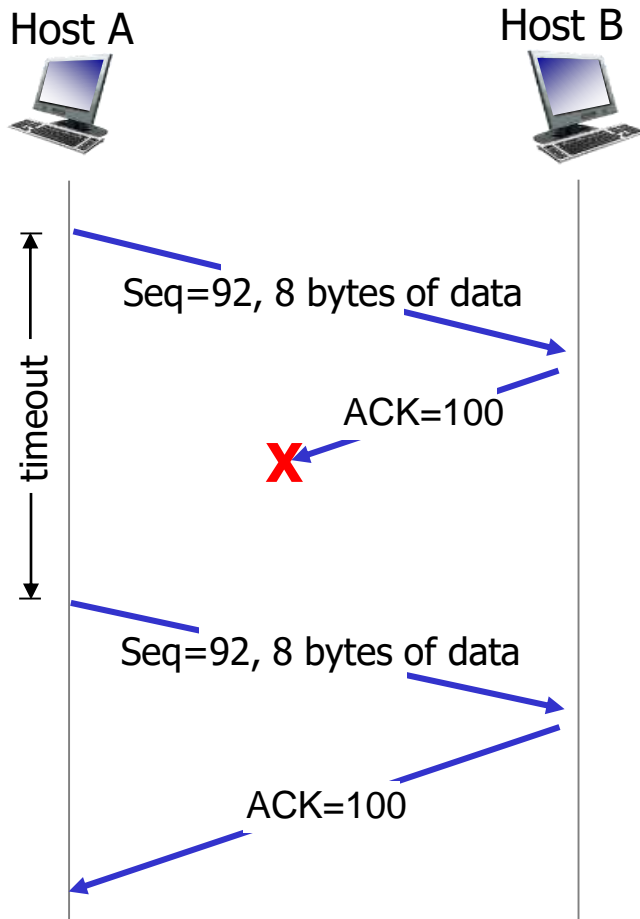
```
if (y > SendBase) {  
    SendBase = y  
    /* SendBase-1: last cumulatively ACKed byte */  
    if (there are currently not-yet-acked segments)  
        start timer  
    else stop timer  
}
```

Dữ liệu được nhận từ tầng application trên  
create segment, seq. #: NextSeqNum  
pass segment to IP (i.e., “send”)  
NextSeqNum = NextSeqNum + length(data)  
if (bộ định thì hiện thời không chạy)  
khởi động bộ định thì

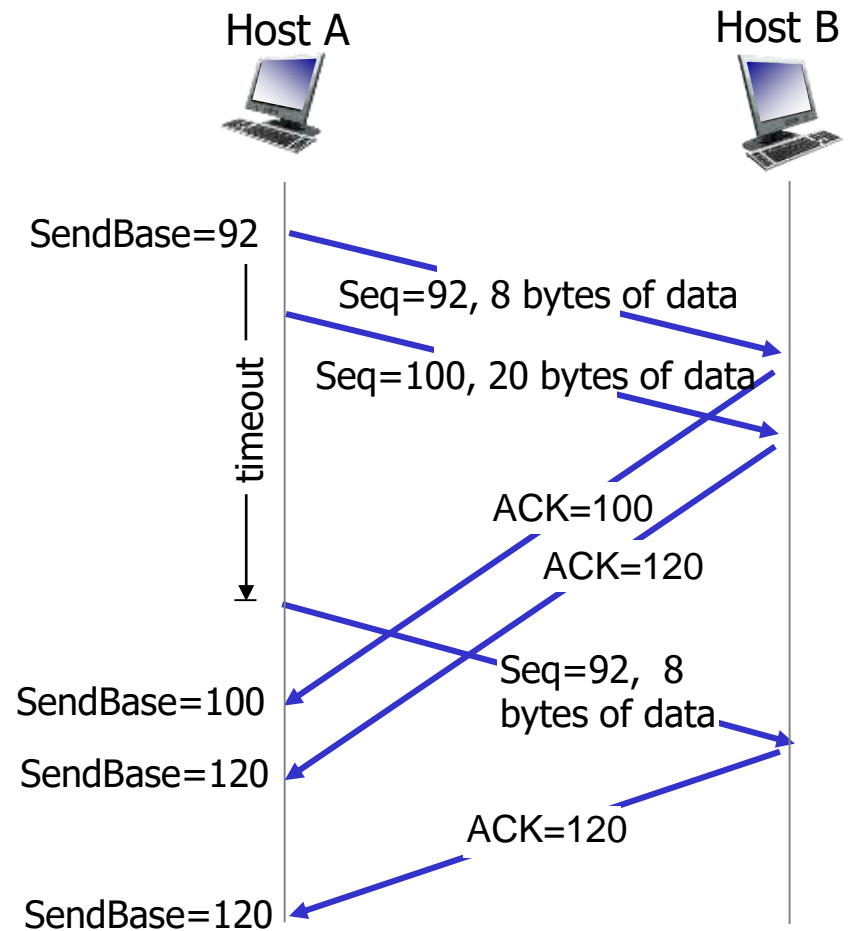
timeout

Truyền lại segment nào chưa được  
báo đã nhận thành công với số  
thứ tự nhỏ nhất.  
Khởi động bộ định thì

# TCP: tình huống truyền lại

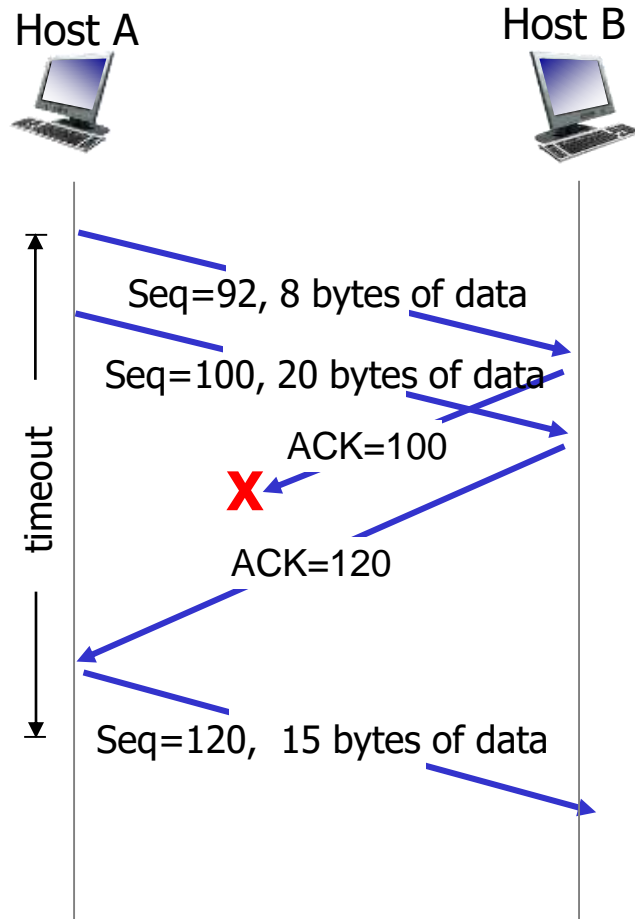


Tình huống mất ACK



Timeout sớm

# TCP: tình huống truyền lại



ACK tích lũy

# Sự phát sinh TCP ACK [RFC 1122, RFC 2581]

| <i>Sự kiện tại bên nhận</i>   | <i>Hành động bên nhận TCP</i>  |
|---|--|
| segment đến theo thứ tự với số thứ tự được mong đợi. Tất cả dữ liệu đến đã được ACK   | ACK bị trễ. Đợi đến 500ms cho segment kế tiếp. Nếu không có segment kế tiếp, gửi ACK   |
| segment đến theo thứ tự với số thứ tự mong muốn. 1 segment khác có ACK đang treo      | Lập tức gửi lại một ACK tích lũy, thông báo nhận thành công cho cả segment theo thứ tự |
| Segment đến không theo thứ tự với số thứ tự lớn hơn số được mong đợi. Có khoảng trống | Lập tức gửi lại ACK trùng, chỉ ra số thứ tự của byte được mong đợi kế tiếp             |
| segment đến lấp đầy từng phần hoặc toàn bộ khoảng trống                               | Lập tức gửi ACK, với điều kiện là segment đó bắt đầu ngay điểm có khoảng trống         |

# TCP truyền lại nhanh

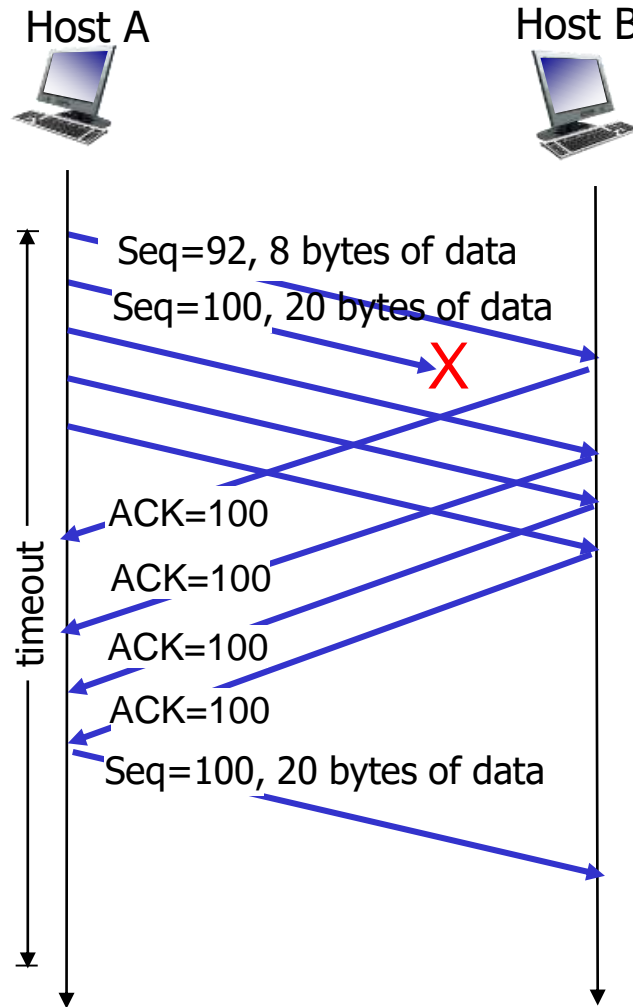
- ❖ Chu kỳ time-out thường tương đối dài:
  - Độ trễ dài trước khi gửi lại packet bị mất
- ❖ Phát hiện các segment bị mất thông qua các ACKs trùng.
  - Bên gửi thường gửi nhiều segment song song
  - Nếu segment bị mất, thì sẽ có khả năng có nhiều ACK trùng.

## *TCP truyền lại nhanh*

Nếu bên gửi nhận 3 ACK của cùng 1 dữ liệu ("3 ACK trùng"), thì gửi lại segment chưa được ACK với số thứ tự nhỏ nhất

- Có khả năng segment không được ACK đã bị mất, vì thế không đợi đến thời gian timeout

# TCP truyền lại nhanh



Truyền lại nhanh sau khi  
bên gửi nhận 3 lần ACK bị trùng



# Chương 3 Nội dung

3.1 các dịch vụ tầng  
Transport

3.2 multiplexing và  
demultiplexing

3.3 vận chuyển phi kết  
nối: UDP

3.4 các nguyên lý  
truyền dữ liệu tin  
cậy

3.5 vận chuyển hướng  
kết nối: TCP

- Cấu trúc segment
- Truyền dữ liệu tin cậy
- Điều khiển luồng (flow control)
- Quản lý kết nối

3.6 các nguyên lý về điều  
khiển tắc nghẽn

3.7 điều khiển tắc nghẽn  
TCP

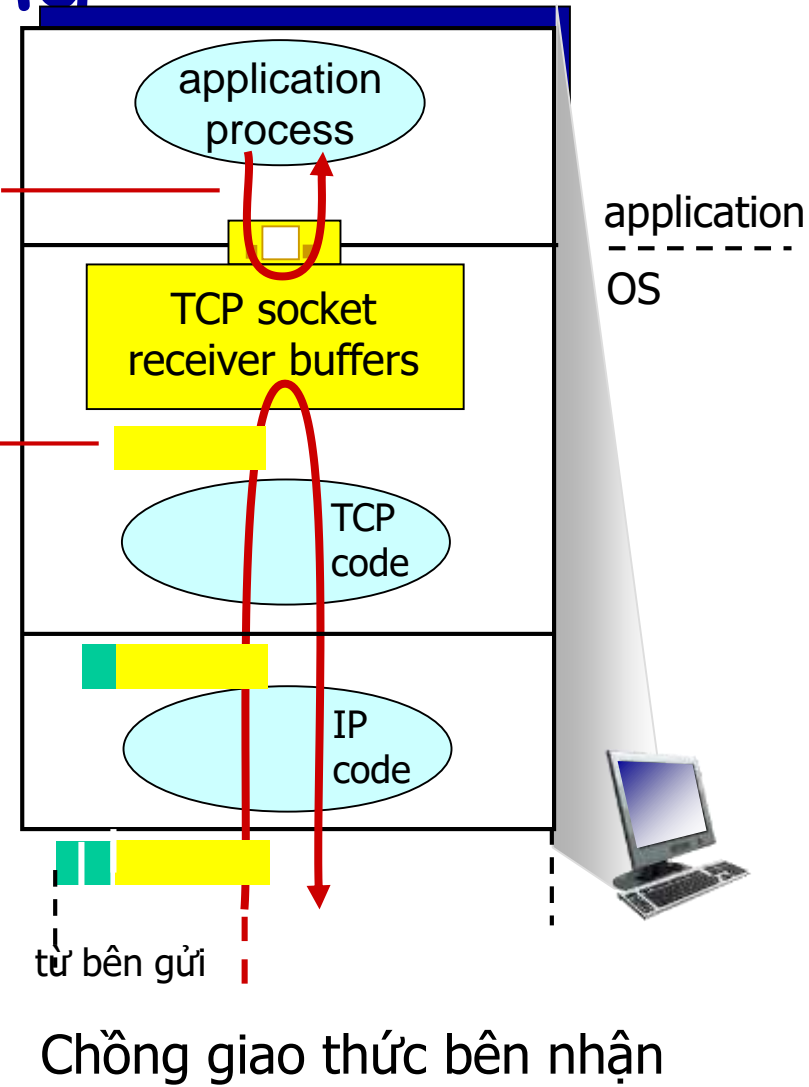
# TCP điều khiển luồng

application có thể loại bỏ dữ liệu  
từ các bộ nhớ đệm socket TCP  
....

... chậm hơn TCP bên  
nhận đang cung cấp  
(bên gửi đang gửi)

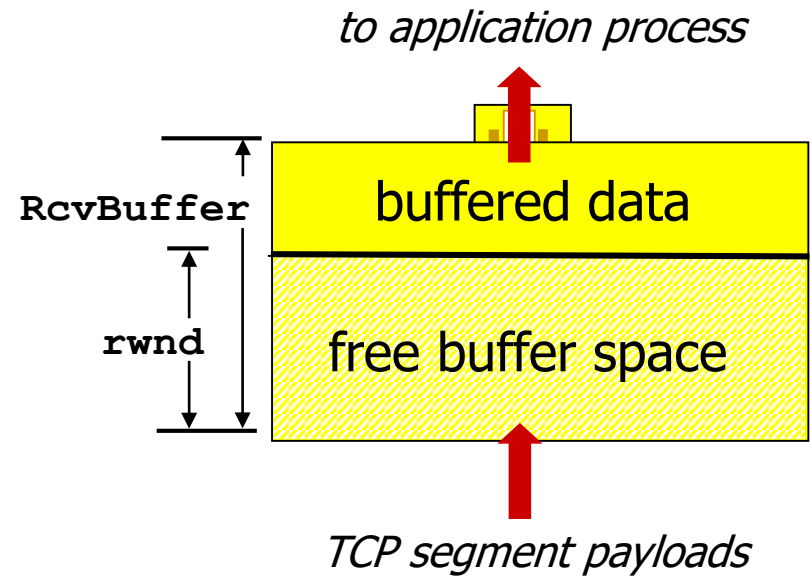
## Điều khiển luồng

bên nhận kiểm soát bên gửi, để  
bên gửi sẽ không làm tràn bộ  
nhớ đệm của bên nhận bởi  
truyền quá nhiều và quá nhanh



# TCP điều khiển luồng

- ❖ Bên nhận “quảng cáo” không gian bộ nhớ đệm còn trống bằng cách thêm giá trị **rwnd** trong TCP header của các segment từ bên nhận đến bên gửi
  - Kích thước của **RcvBuffer** được thiết đặt thông qua các tùy chọn của socket (thông thường mặc định là 4096 byte)
  - Nhiều hệ điều hành tự động điều chỉnh **RcvBuffer**
- ❖ Bên gửi giới hạn số lượng dữ liệu tới giá trị **rwnd** của bên nhận
- ❖ Bảo đảm bộ đệm bên nhận sẽ không bị tràn



*Bộ đệm phía bên nhận*

# Chương 3 Nội dung

3.1 các dịch vụ tầng  
Transport

3.2 multiplexing và  
demultiplexing

3.3 vận chuyển phi kết  
nối: UDP

3.4 các nguyên lý  
truyền dữ liệu tin  
cậy

3.5 vận chuyển hướng  
kết nối: TCP

- Cấu trúc segment
- Truyền dữ liệu tin cậy
- Điều khiển luồng (flow control)
- Quản lý kết nối

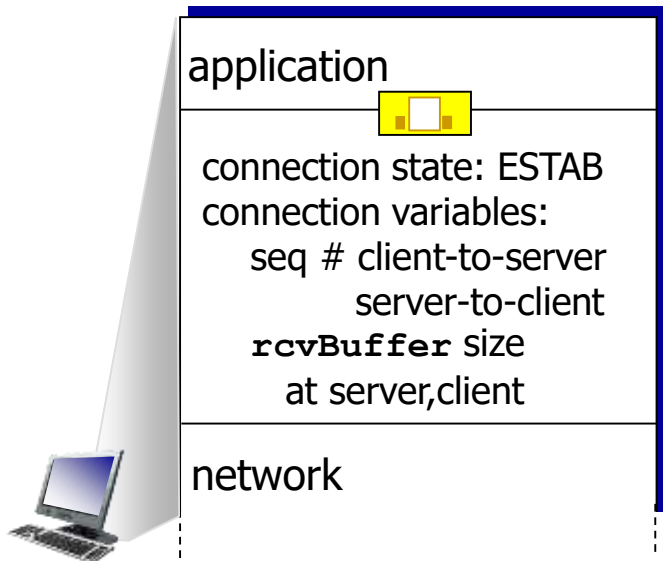
3.6 các nguyên lý về điều  
khiển tắc nghẽn

3.7 điều khiển tắc nghẽn  
TCP

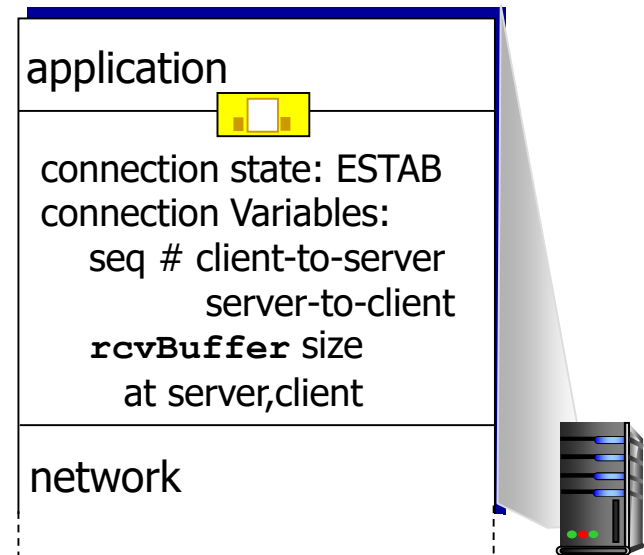
# Quản lý kết nối (Connection Management)

Trước khi trao đổi dữ liệu, bên gửi và nhận "bắt tay nhau" :

- ❖ Đồng ý thiết lập kết nối (mỗi bên biết bên kia sẵn sàng để thiết lập kết nối)
- ❖ Đồng ý các thông số kết nối



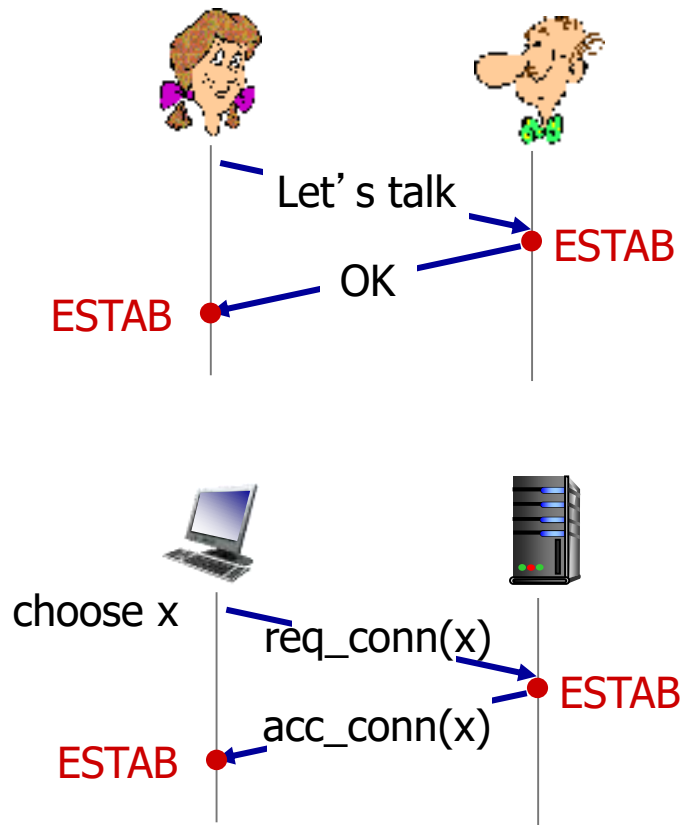
```
Socket clientSocket =  
    newSocket("hostname", "port  
    number");
```



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

# Đồng ý thiết lập kết nối

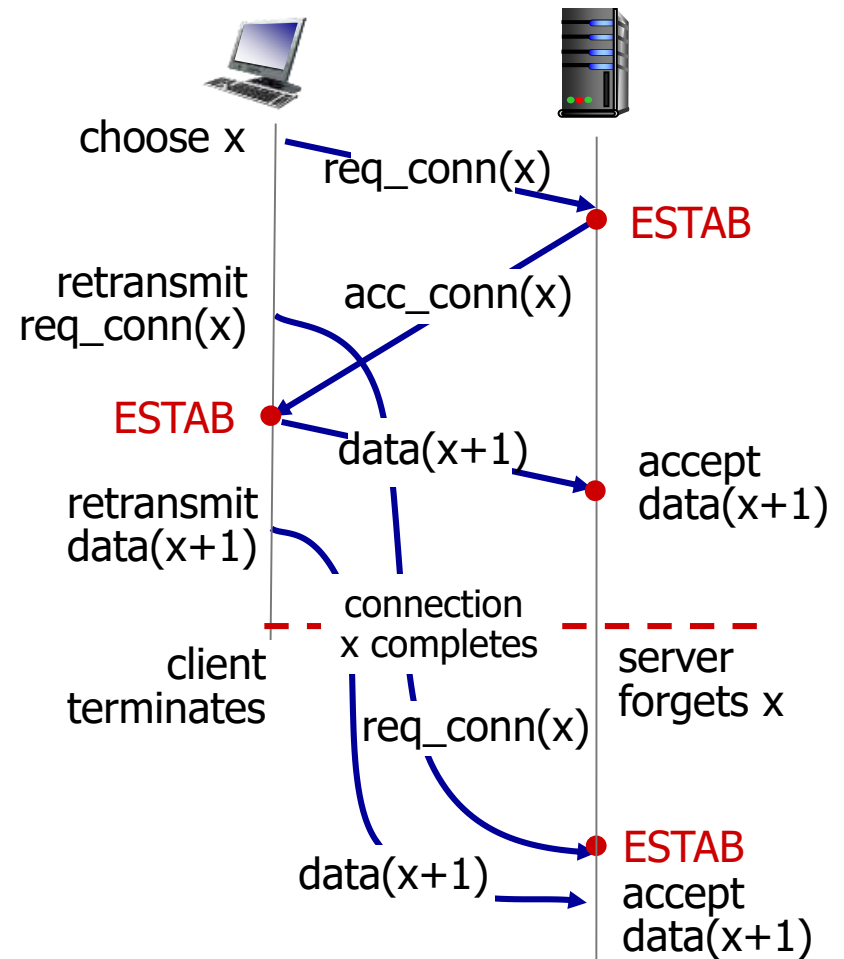
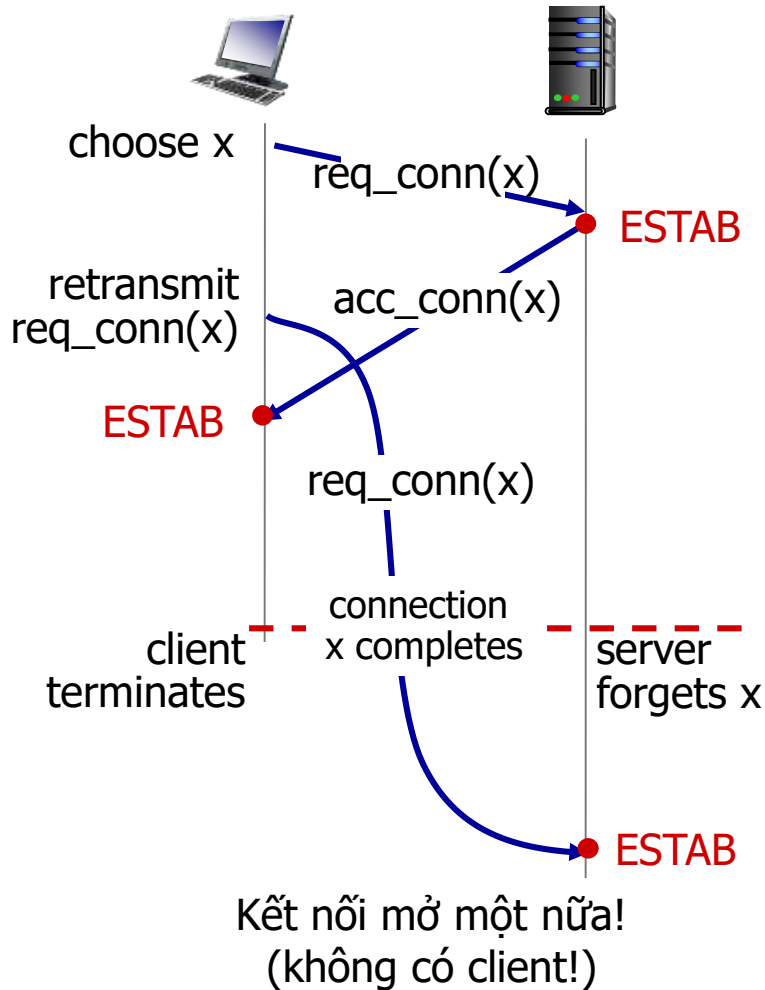
Bắt tay 2-way:



- Hỏi: bắt tay 2-way sẽ luôn luôn hoạt động trong mạng hay không?
- ❖ Độ chậm trễ biến thiên
  - ❖ Các thông điệp được truyền lại (như `req_conn(x)`) vì mất thông điệp
  - ❖ Sắp xếp lại thông điệp
  - ❖ Không thể "thấy" phía bên kia

# Đồng ý thiết lập kết nối

Các tình huống thất bại khi bắt tay 2-way:



# TCP bắt tay 3-way

*Trạng thái client*



LISTEN

SYNSENT

ESTAB

Chọn số thứ tự ban đầu, x  
Gửi TCP SYN msg

SYNbit=1, Seq=x

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

ACKbit=1, ACKnum=y+1

SYNACK(x) vừa được nhận  
cho hay server vẫn còn sống;  
send ACK for SYNACK;  
this segment may contain  
client-to-server data



*Trạng thái server*

LISTEN

SYN RCVD

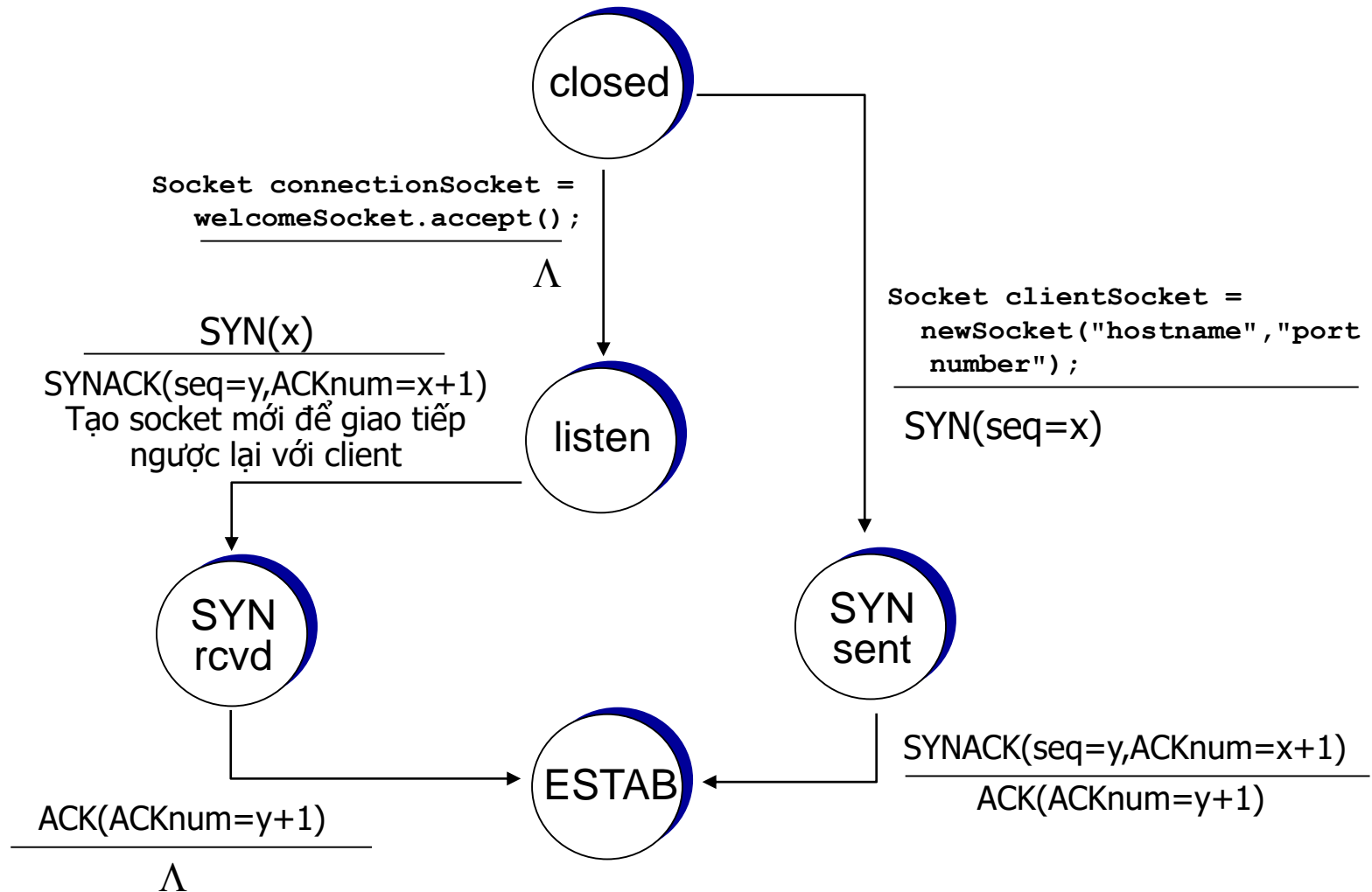
ESTAB

Chọn số thứ tự ban đầu, y  
gửi TCP SYNACK  
msg, acking SYN

ACK(y) vừa được nhận  
cho hay client vẫn sống



# TCP bắt tay 3-way: FSM

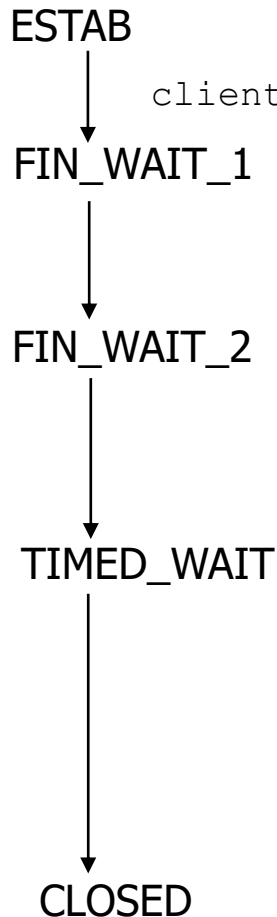


# TCP: đóng kết nối

- ❖ Mỗi bên client và server sẽ đóng kết nối bên phía của nó
  - Gửi TCP segment với FIN bit = 1
- ❖ Phản hồi bằng ACK cho FIN vừa được nhận
  - Khi nhận FIN, ACK có thể được kết hợp với FIN của nó
- ❖ Các trao đổi FIN đồng thời có thể được vận dụng

# TCP: đóng kết nối

## Trạng thái client



`clientSocket.close()`

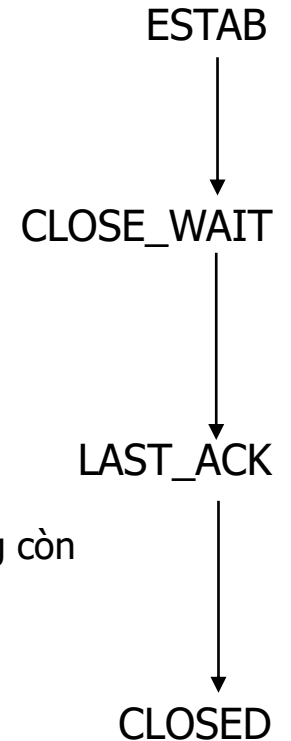
Có thể không  
còn gửi nhưng  
vẫn còn nhận  
dữ liệu

Chờ server  
đóng

timed wait  
for  $2 * \text{max}$   
segment lifetime



## Trạng thái server



FINbit=1, seq=x

ACKbit=1; ACKnum=x+1

Vẫn có thể  
gửi dữ liệu

FINbit=1, seq=y

ACKbit=1; ACKnum=y+1

Có thể không còn  
gửi dữ liệu

# Chương 3 Nội dung

3.1 các dịch vụ tầng  
Transport

3.2 multiplexing và  
demultiplexing

3.3 vận chuyển phi kết  
nối: UDP

3.4 các nguyên lý  
truyền dữ liệu tin  
cậy

3.5 vận chuyển hướng  
kết nối: TCP

- Cấu trúc segment
- Truyền dữ liệu tin cậy
- Điều khiển luồng (flow control)
- Quản lý kết nối

3.6 các nguyên lý về điều  
khiển tắc nghẽn

3.7 điều khiển tắc nghẽn  
TCP

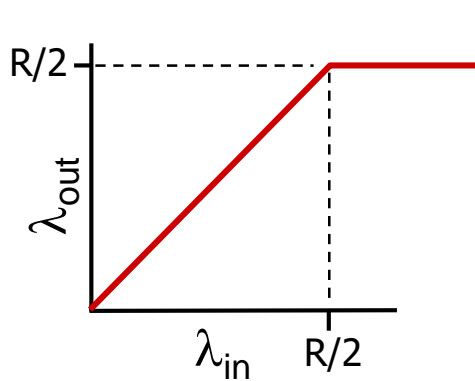
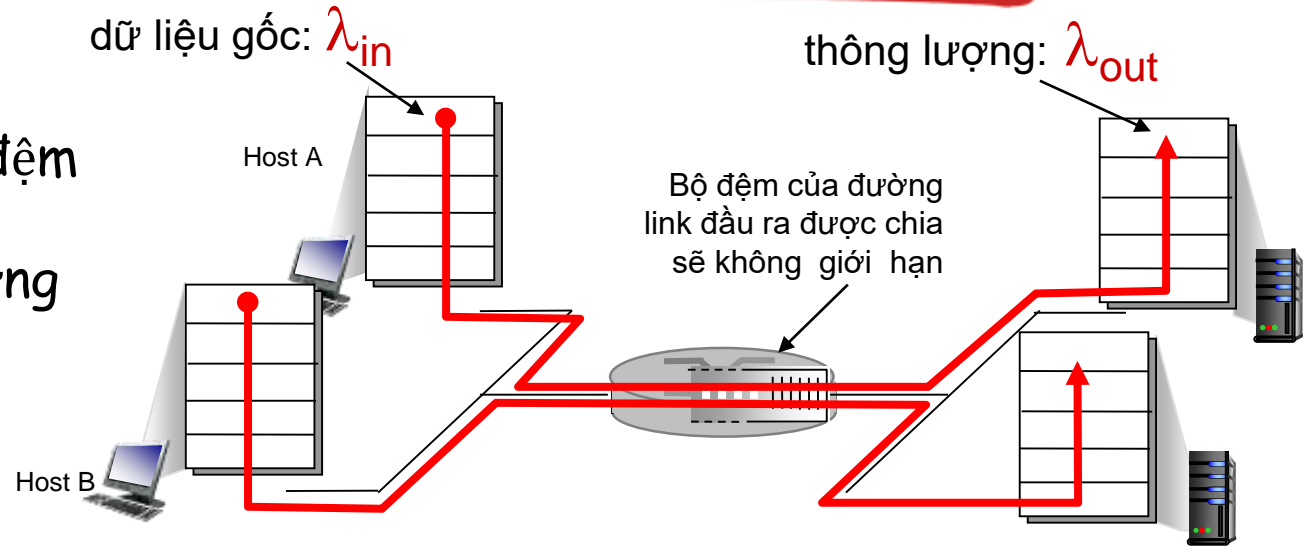
# Các nguyên lý điều khiển tắc nghẽn (congestion control)

## Tắc nghẽn:

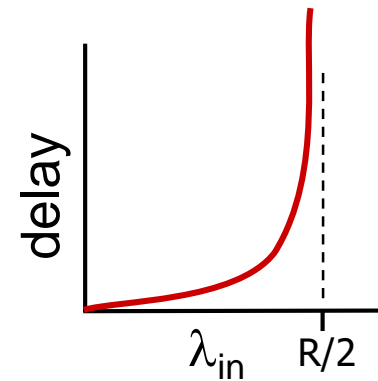
- ❖ “ quá nguyên nguồn gửi quá nhiều dữ liệu với tốc độ quá nhanh đến *mạng* để được xử lý”
- ❖ Khác với điều khiển luồng (flow control)!
- ❖ Các biểu hiện:
  - Mất gói (tràn bộ đệm tại các router)
  - Độ trễ lớn (xếp hàng trong các bộ đệm của router)
- ❖ 1 trong 10 vấn đề khó khăn!

# Nguyên nhân/Chi phí của tắc nghẽn: tình huống 1

- ❖ 2 gửi, 2 nhận
- ❖ 1 router, các bộ đệm không giới hạn
- ❖ Khả năng của đường link đầu ra:  $R$
- ❖ Không truyền lại



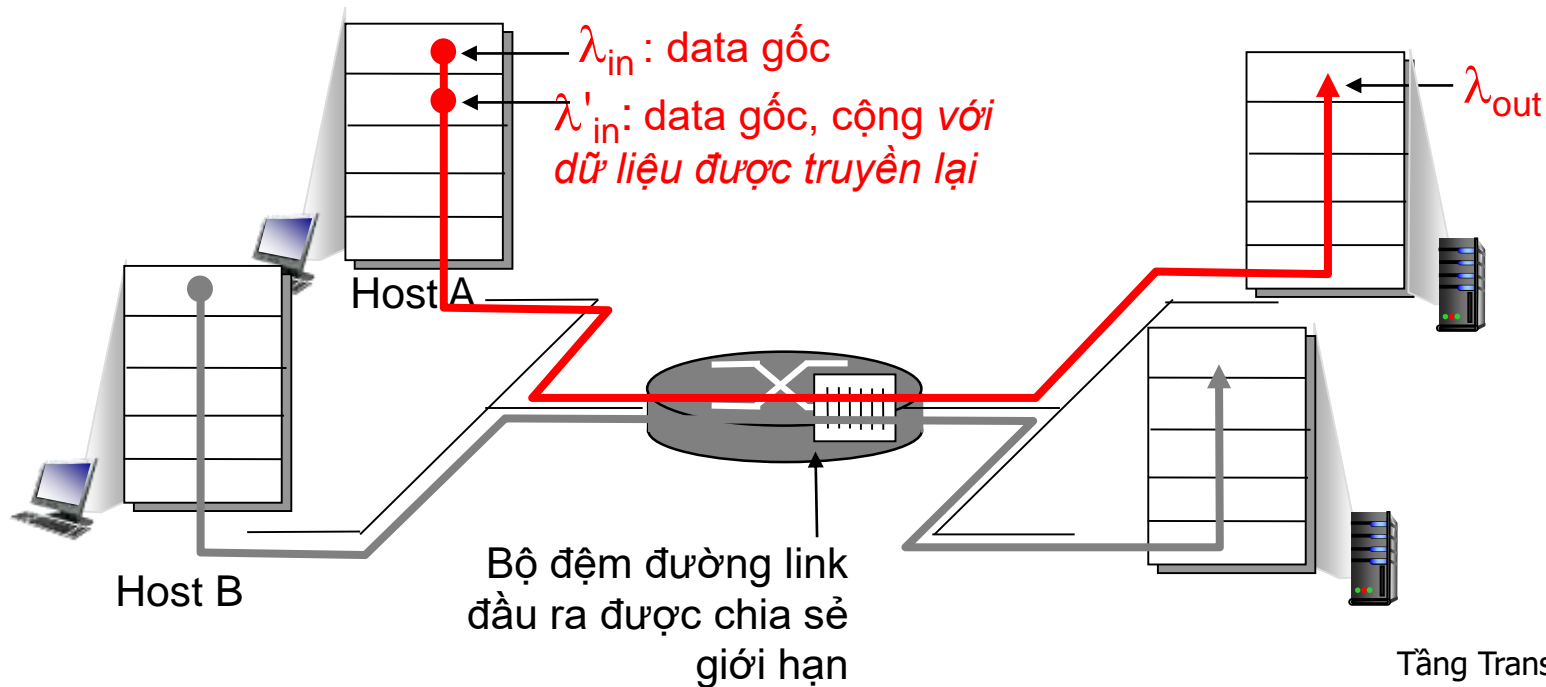
- ❖ Thông lượng lớn nhất của mỗi kết nối:  $R/2$



- ❖ Độ trễ lớn khi tốc độ đến,  $\lambda_{in}$ , vượt tới capacity

# Nguyên nhân/Chi phí của tắc nghẽn: tình huống 2

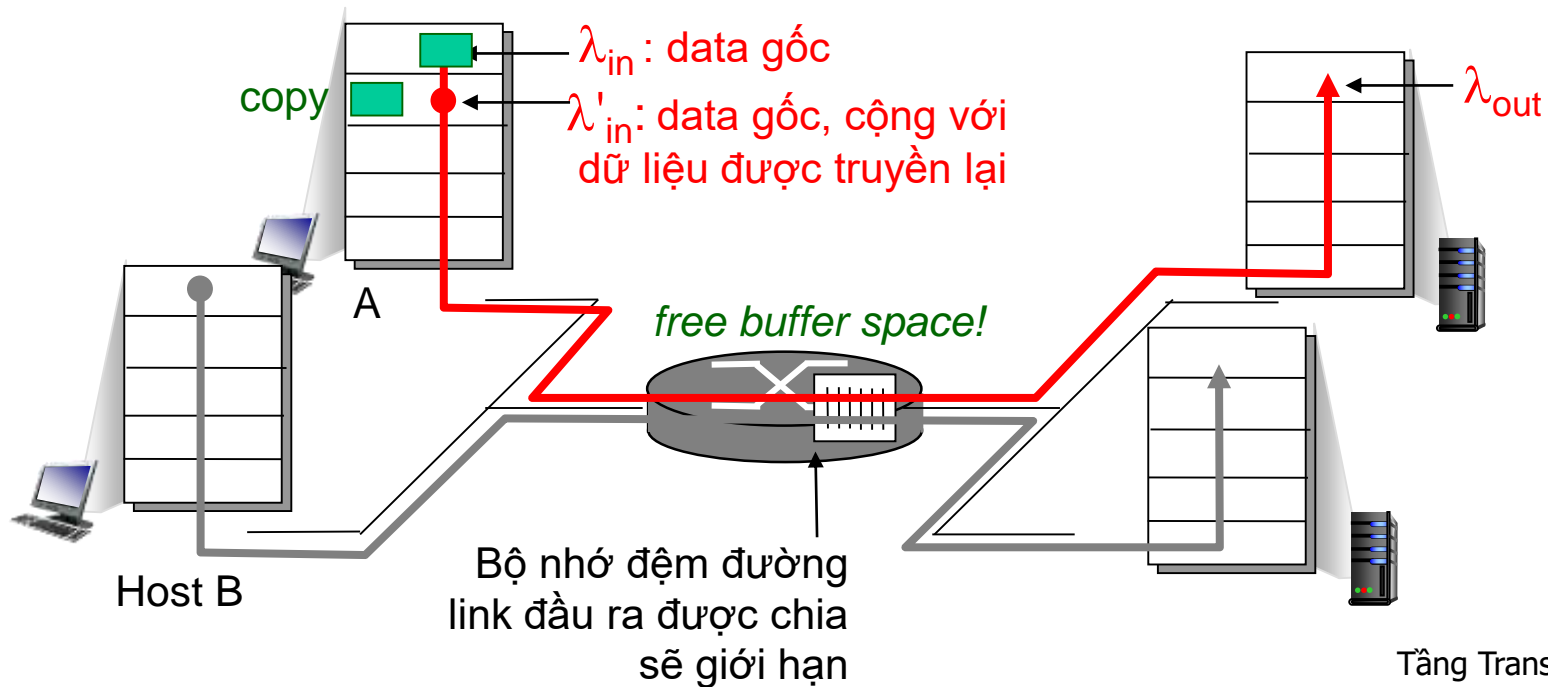
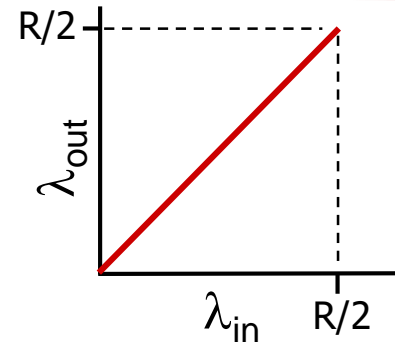
- ❖ 1 router, các bộ đệm có giới hạn
- ❖ bên gửi truyền lại các packet bị time-out
  - application-layer input = application-layer output:  $I_{in} = I_{out}$
  - transport-layer input bao gồm việc truyền lại:  $I'_{in} \geq I_{in}$



# Nguyên nhân/Chi phí của tắc nghẽn: tình huống 2

Lý tưởng hóa: kiến thức hoàn hảo

- ❖ Bên gửi chỉ gửi khi bộ nhớ đệm của router sẵn sàng

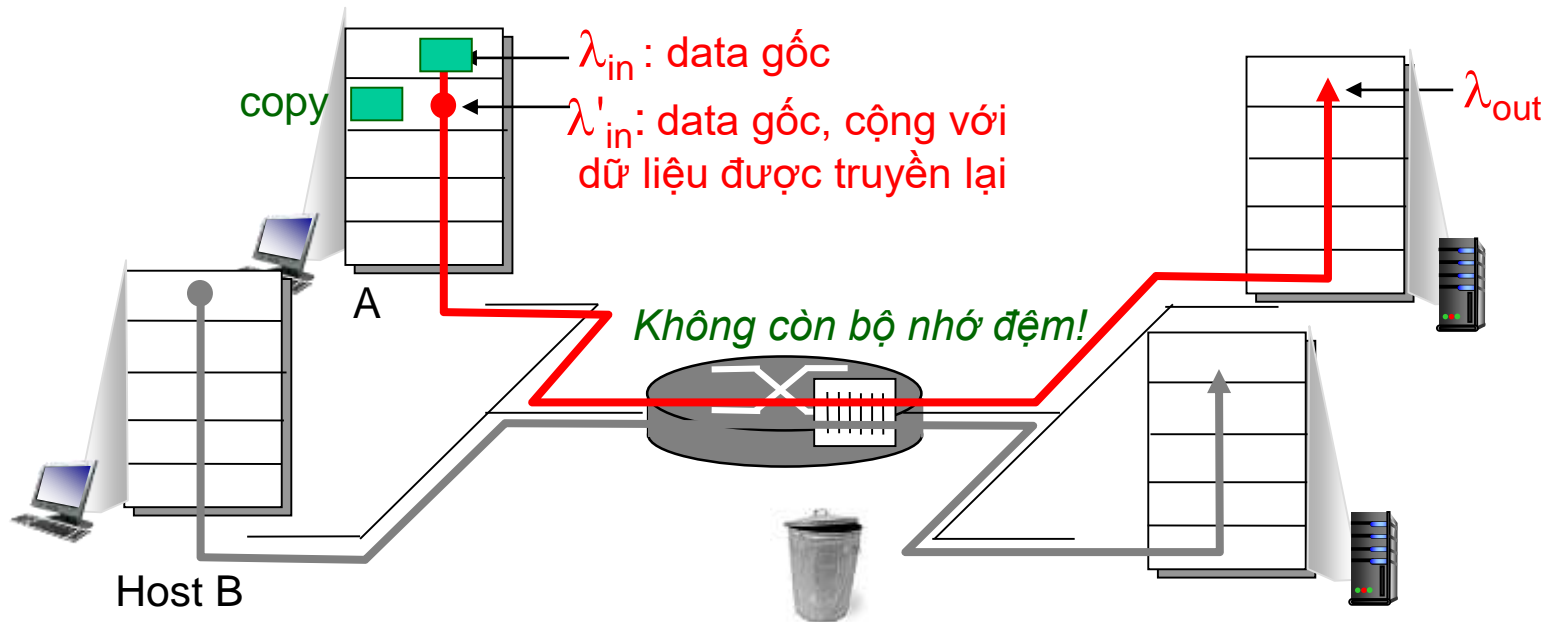




# Nguyên nhân/Chi phí của tắc nghẽn: tình huống 2

*Lý tưởng hóa:* các packet bị mất được biết đến có thể bị mất hoặc bị loại bỏ tại router bởi vì bộ nhớ đệm bị đầy

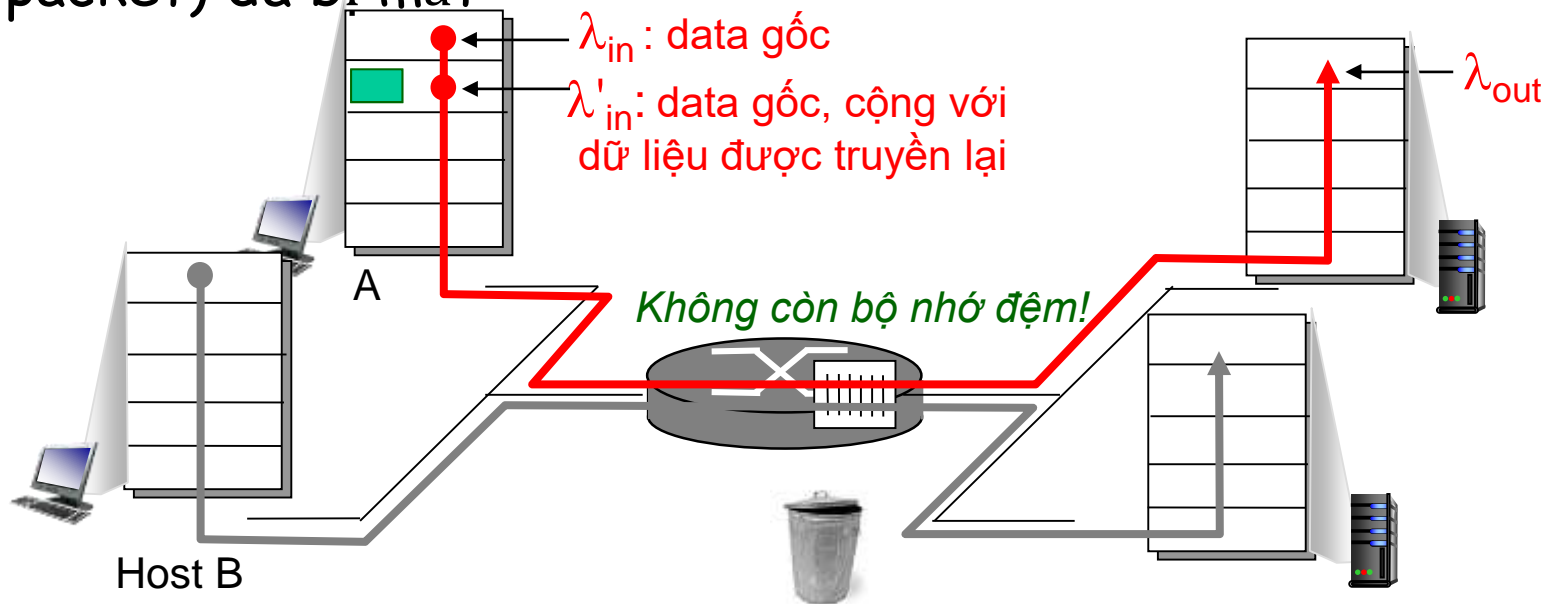
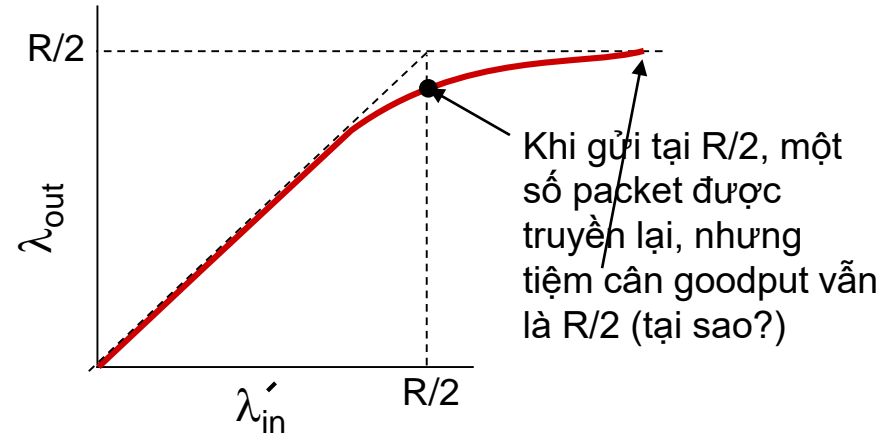
- ❖ Bên gửi chỉ gửi lại packet được biết đến (known packet) đã bị mất



# Nguyên nhân/Chi phí của tắc nghẽn: tình huống 2

**Lý tưởng hóa:** các packet bị mất được biết đến có thể bị mất hoặc bị loại bỏ tại router bởi vì bộ nhớ đệm bị đầy

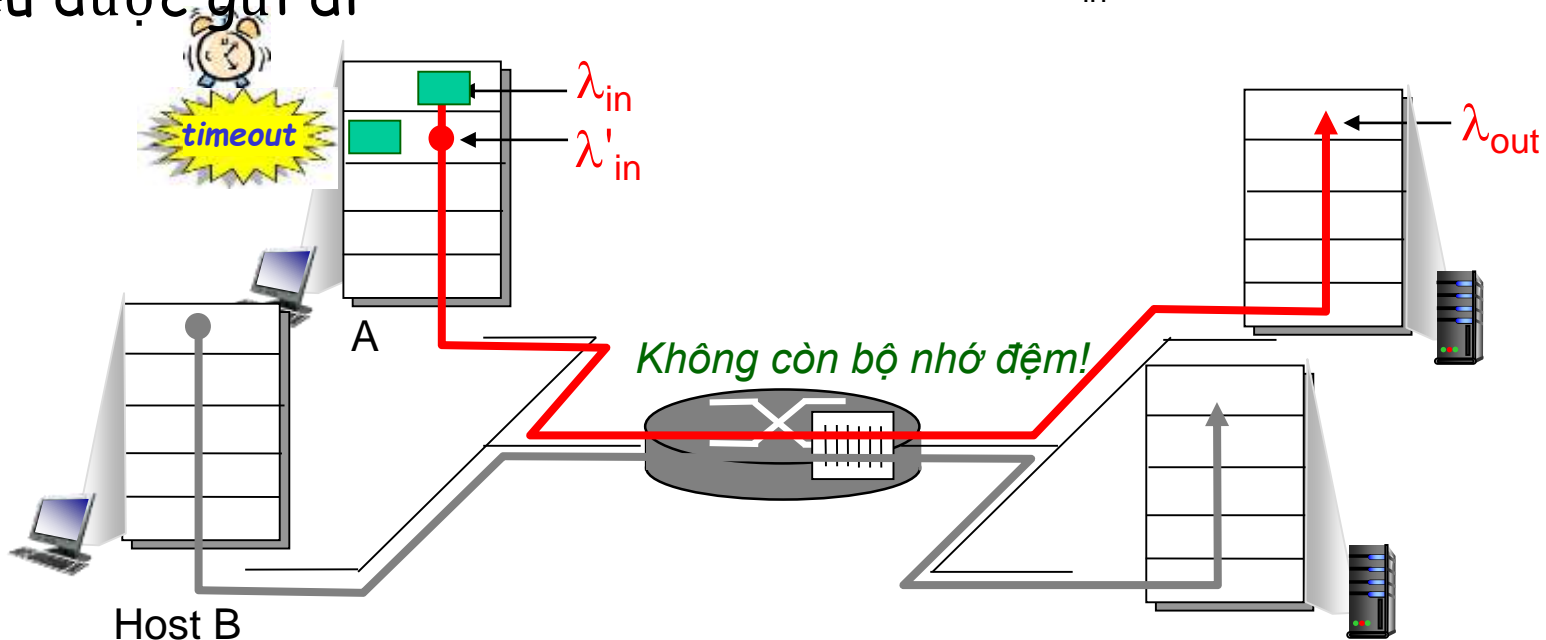
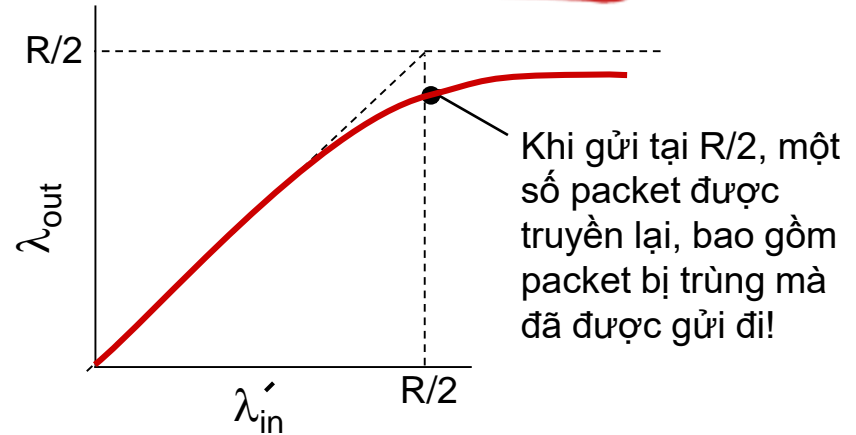
- ❖ Bên gửi chỉ gửi lại packet được biết đến (known packet) đã bị mất



# Nguyên nhân/Chi phí của tắc nghẽn: tình huống 2

## Thực tế: *trùng lặp*

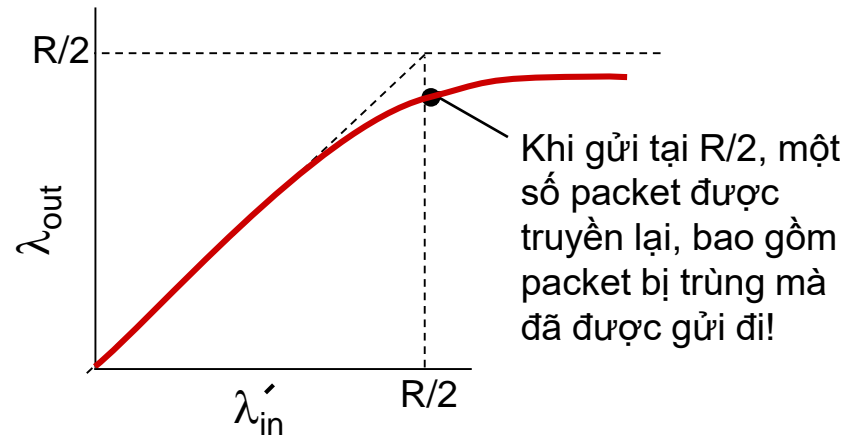
- ❖ Các packet có thể bị mất, bị bỏ tại router bởi vì bộ nhớ đệm đầy
- ❖ Thời gian time out bên gửi hết sớm, gửi 2 bản giống nhau, cả 2 đều được gửi đi



# Nguyên nhân/Chi phí của tắc nghẽn: tình huống 2

## Thực tế: trùng lặp

- ❖ Các packet có thể bị mất, bị bỏ tại router bởi vì bộ nhớ đệm đầy
- ❖ Thời gian time out bên gửi hết sớm, gửi 2 bản giống nhau, cả 2 đều được gửi đi



## “chi phí” của tắc nghẽn:

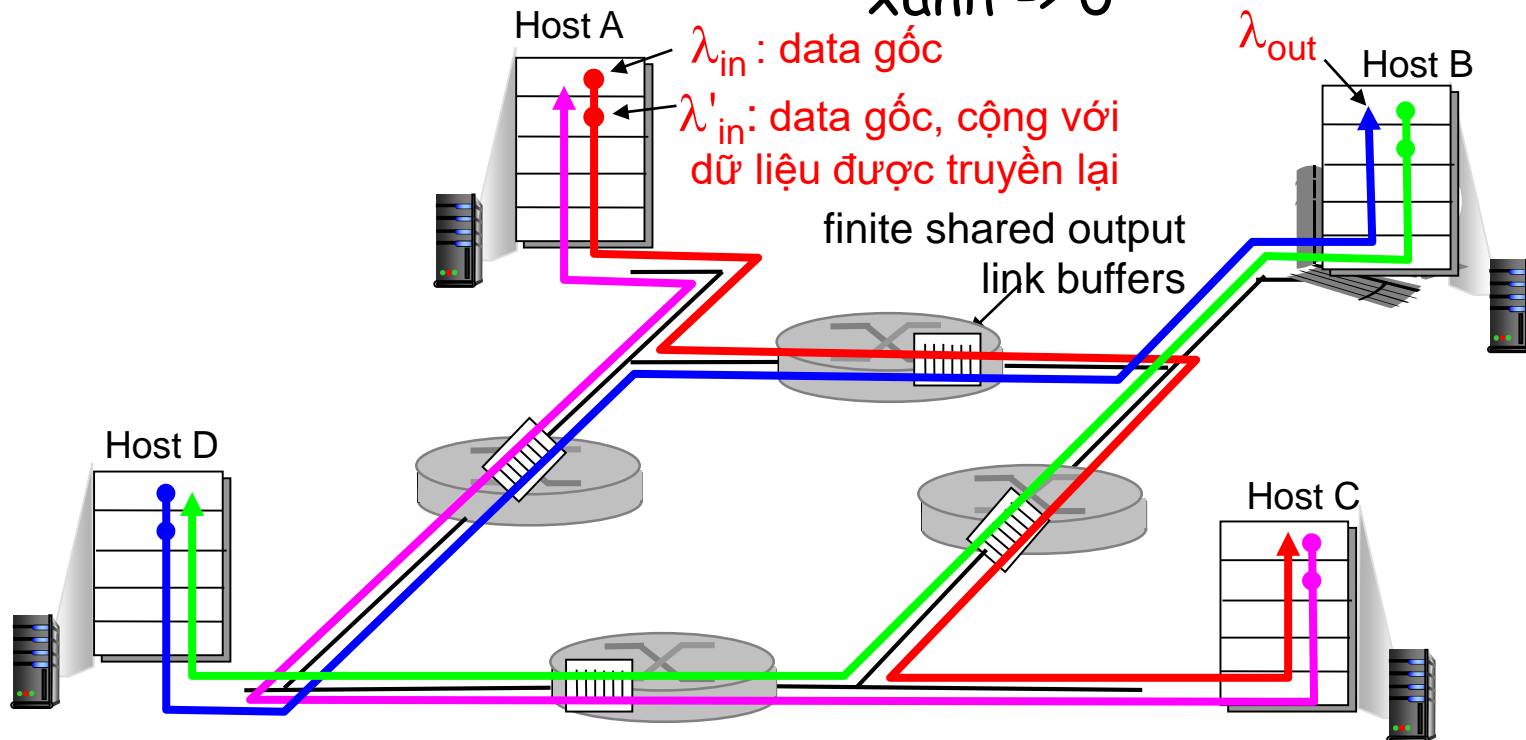
- ❖ Nhiều việc (truyền lại) cho given “goodput” được cho
- ❖ Truyền lại không cần thiết: đường link mang nhiều bản sao của packet
  - Giảm goodput

# Nguyên nhân/Chi phí của tắc nghẽn: tình huống 3

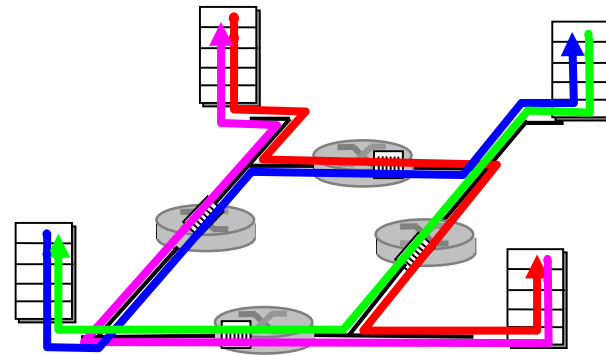
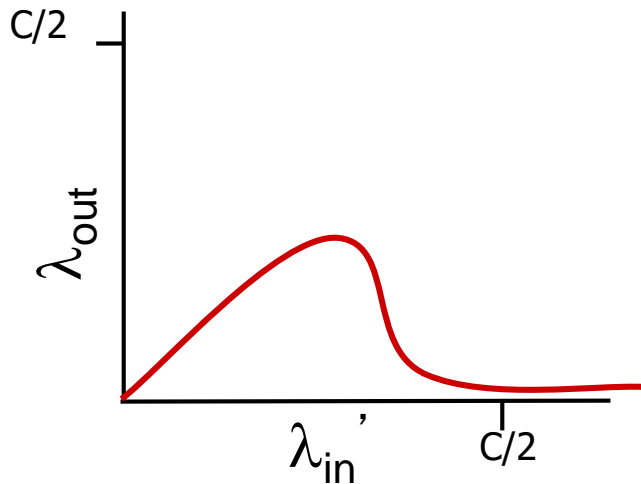
- ❖ 4 người gửi
- ❖ Các đường qua nhiều hop
- ❖ timeout/truyền lại

Hỏi: cái gì xảy ra khi  $I_{in}$  và  $I_{in}'$  tăng?

A: khi  $I_{in}'$  màu đỏ tăng, tất cả packet màu xanh đến tại hàng đợi phía trên bị loại bỏ, thông lượng màu xanh  $\rightarrow 0$



# Nguyên nhân/Chi phí của tắc nghẽn: tình huống 3



## “Chi phí” khác của tắc nghẽn

- ❖ Khi packet bị loại bỏ, bất kỳ “khả năng truyền upstream được sử dụng cho packet đó đều bị lãng phí!”

# Các phương pháp tiếp cận đối với điều khiển tắc nghẽn

---

2 phương pháp tiếp cận:

## Điều khiển tắc nghẽn end-end :

- ❖ Không phản hồi rõ ràng từ mạng
- ❖ Tắc nghẽn được suy ra từ việc quan sát hệ thống đầu cuối có mất mát hoặc bị trễ
- ❖ Tiếp cận được thực hiện bởi TCP

## Điều khiển tắc nghẽn có sự hỗ trợ của mạng (network-assisted) :

- ❖ Các router cung cấp phản hồi đến các hệ thống đầu cuối
  - Bit đơn chỉ ra tắc nghẽn (SNA, DECbit, TCP/IP ECN, ATM)
  - Tốc độ sẽ gửi của người gửi được xác định rõ ràng

# Case study: điều khiển tắt nghẽn ATM ABR

---

## ABR: available bit rate:

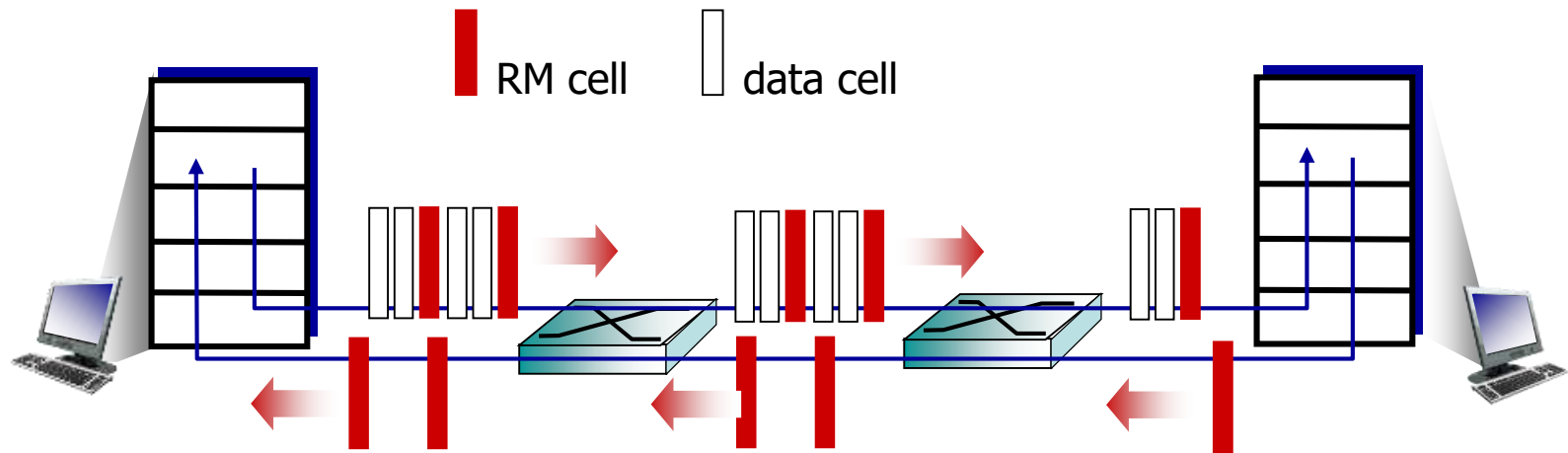
- ❖ “dịch vụ mềm dẻo”
- ❖ Nếu đường đi của bên gửi “chưa hết”:
  - Bên gửi sẽ dùng băng thông sẵn sàng
- ❖ Nếu đường đi của bên gửi bị tắt nghẽn:
  - Bên gửi sẽ điều tiết với tốc độ tối thiểu được bảo đảm

## Các cell RM (resource management):

- ❖ Được gởi bởi bên gửi, được xen kẽ với các cell dữ liệu
- ❖ Các bit trong RM cell được thiết lập bởi các switcher (“network-assisted”)
  - *NI bit*: không tăng tốc độ (tắt nghẽn nhẹ)
  - *CI bit*: tắt nghẽn rõ rệt
- ❖ Các cell RM được trả về bên gửi từ bên nhận với nguyên vẹn các bit



# Case study: điều khiển tắc nghẽn ATM ABR



- ❖ Trường 2 byte ER (tốc độ tường minh) trong cell RM
  - Switch bị tắc nghẽn có thể có giá trị ER thấp hơn trong cell
  - Do đó, tốc độ gửi của bên gửi được hỗ trợ tối đa trên đường đi
- ❖ Bit EFCI bit trong cell dữ liệu: được thiết lập đến 1 trong switch bị tắc nghẽn
  - Nếu cell dữ liệu đứng trước cell RM có thiết lập EFCI bên

# Chương 3 Nội dung

3.1 các dịch vụ tầng  
Transport

3.2 multiplexing và  
demultiplexing

3.3 vận chuyển phi kết  
nối: UDP

3.4 các nguyên lý  
truyền dữ liệu tin  
cậy

3.5 vận chuyển hướng  
kết nối: TCP

- Cấu trúc segment
- Truyền dữ liệu tin cậy
- Điều khiển luồng (flow control)
- Quản lý kết nối

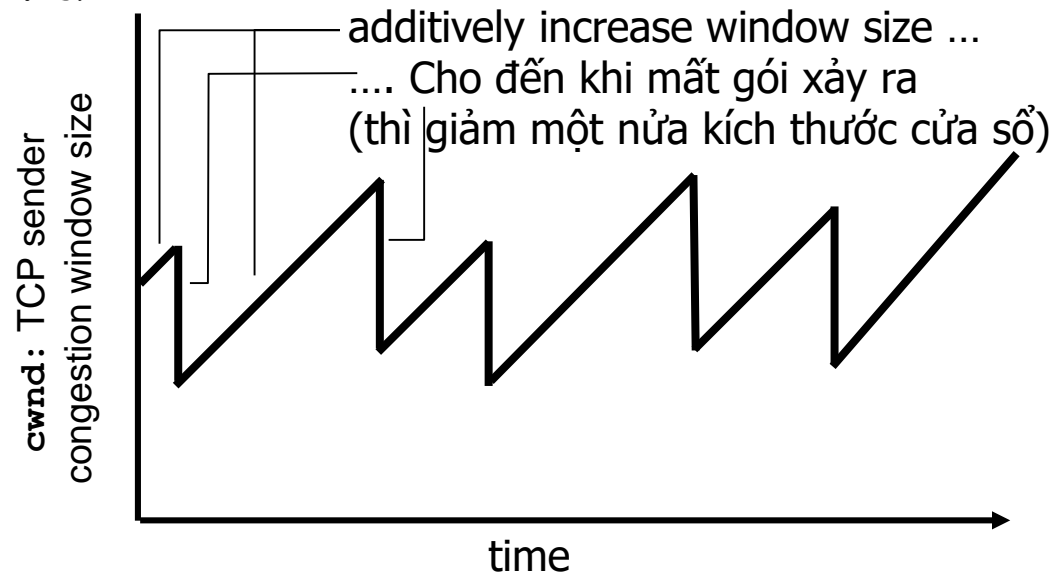
3.6 các nguyên lý về điều  
khiển tắc nghẽn

3.7 điều khiển tắc nghẽn  
TCP

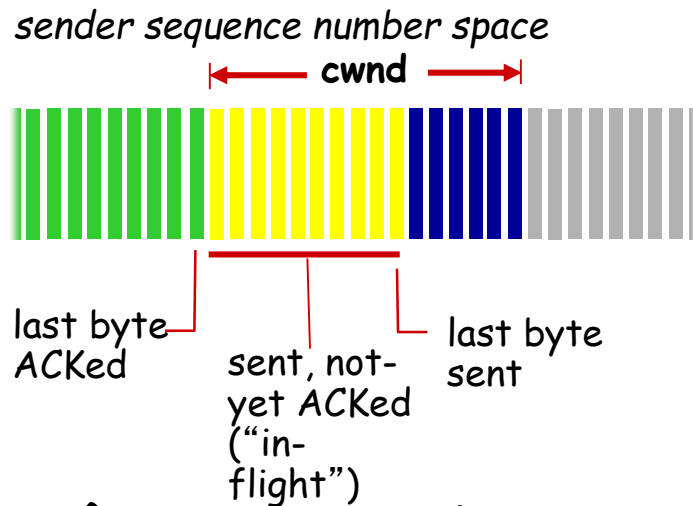
# TCP điều khiển tắc nghẽn: additive increase, multiplicative decrease

- ❖ *Hướng tiếp cận*: bên gửi tăng tốc độ truyền (kích thước cửa sổ), thăm dò băng thông có thể sử dụng, cho đến khi mất mát gói xảy ra
  - *additive increase*: tăng **cwnd** bởi 1 MSS (maximum segment size) mỗi RTT cho đến khi mất gói xảy ra
  - *multiplicative decrease*: giảm một nửa **cwnd** sau khi mất gói xảy ra

AIMD saw tooth behavior: thăm dò băng thông



# TCP điều khiển tắc nghẽn: chi tiết



- ❖ Bên gửi giới hạn truyền tại:

$$\text{LastByteSent} - \text{LastByteAcked} \leq \text{cwnd}$$

- ❖ **cwnd** thay đổi, chức năng nhận biết tắc nghẽn trên mạng

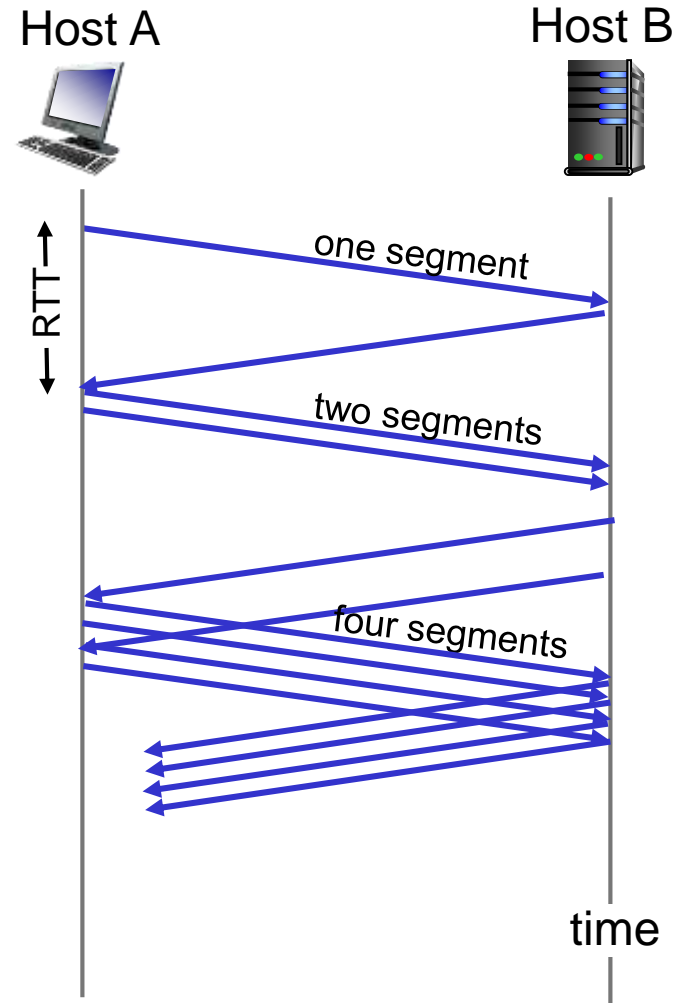
tốc độ gửi TCP:

- ❖ Ước lượng: gửi các byte cwnd, đợi ACK trong khoảng thời gian RTT, sau đó gửi thêm các byte

$$\text{rate} \approx \frac{\text{cwnd}}{\text{RTT}} \text{ bytes/sec}$$

# TCP Slow Start

- ❖ Khi kết nối bắt đầu, tăng tốc độ theo cấp số nhân cho đến sự kiện mất gói đầu tiên xảy ra:
  - initially **cwnd** = 1 MSS
  - Gấp đôi **cwnd** mỗi RTT
  - Được thực hiện bằng cách tăng **cwnd** cho mỗi ACK nhận được
- ❖ Tóm lại: tốc độ ban đầu chậm, nhưng nó sẽ tăng lên theo cấp số nhân



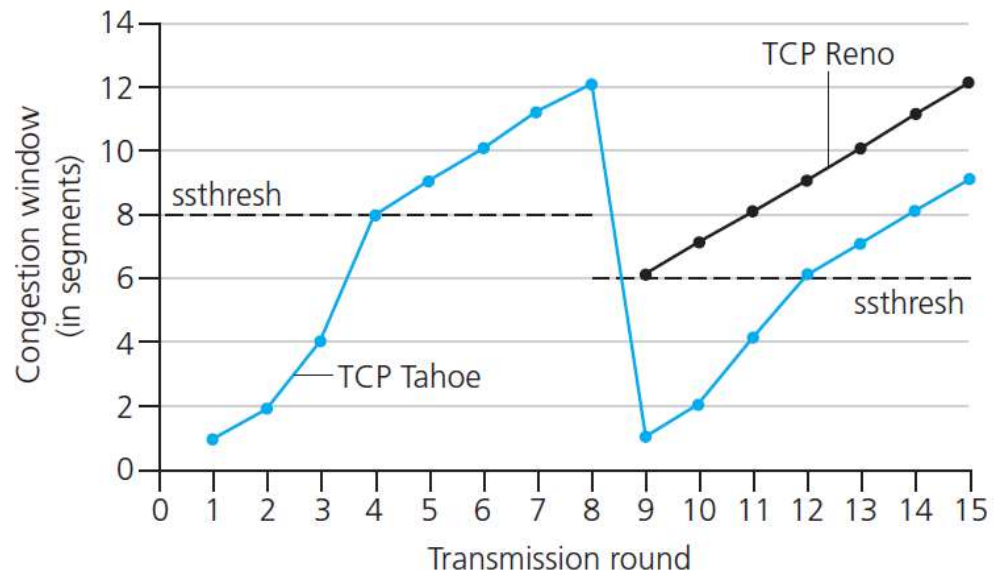
# TCP: phát hiện, phản ứng khi mất gói

- ❖ Mất gói được chỉ ra bởi timeout:
  - **cwnd** (congestion window) được thiết lập 1 MSS;
  - Sau đó kích thước cửa sổ sẽ tăng theo cấp số nhân (như trong slow start) đến ngưỡng, sau đó sẽ tăng tuyến tính
- ❖ Mất gói được xác định bởi 3 ACK trùng nhau: TCP RENO
  - Các ACK trùng lặp chỉ ra khả năng truyền của mạng
  - **cwnd** bị cắt một nửa sau đó tăng theo tuyến tính
- ❖ TCP Tahoe luôn luôn thiết lập **cwnd** bằng 1 (timeout hoặc 3 ack trùng nhau)

# TCP: chuyển từ slow start qua CA (congestion avoidance)

**Hỏi:** khi nào tăng cấp số nhân nên chuyển qua tuyến tính?

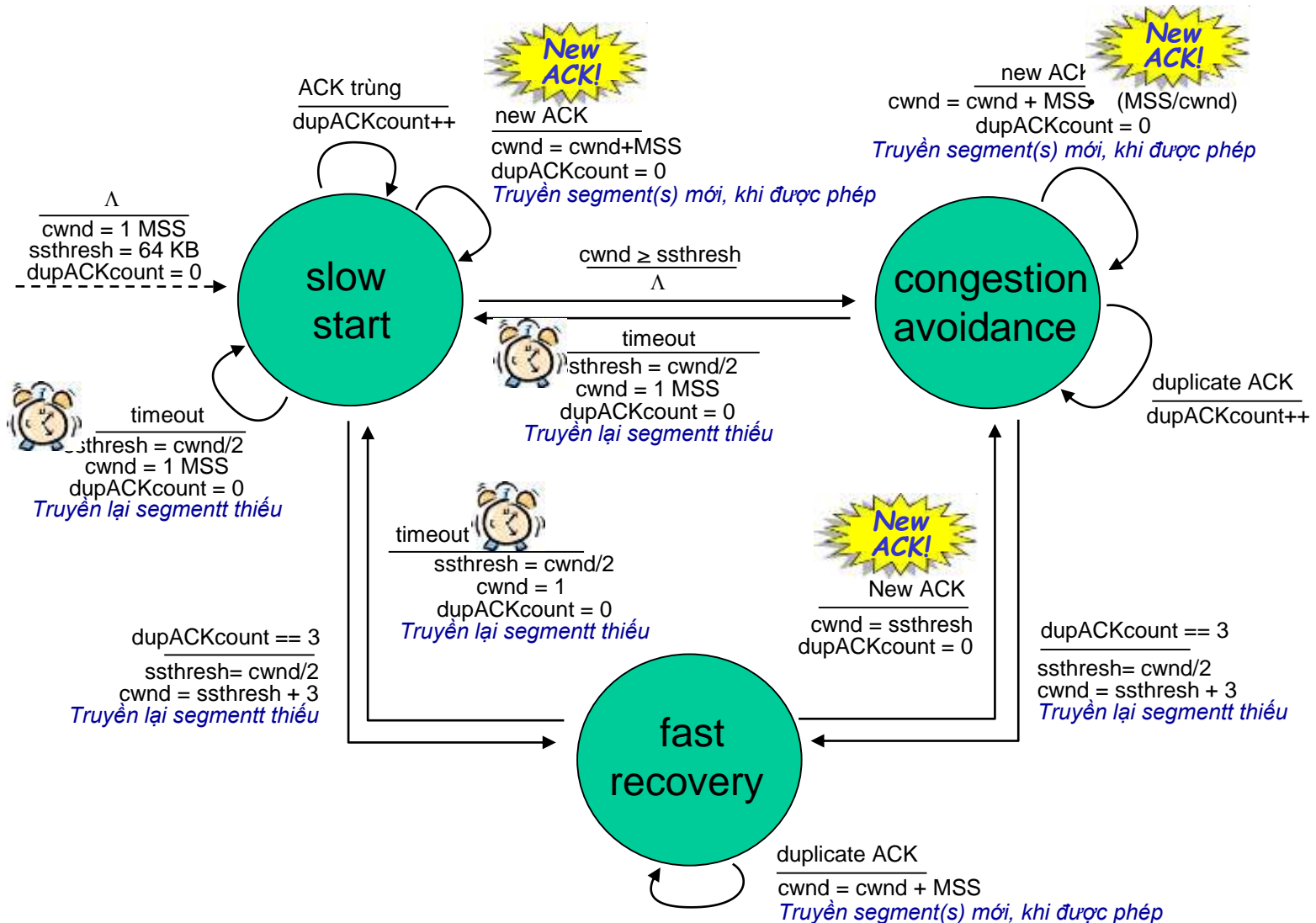
**Trả lời:** khi **cwnd** được  $1/2$  giá trị của nó trước thời gian timeout.



## Thực hiện:

- ❖ **ssthresh** (slow start threshold) thay đổi
- ❖ Khi mất gói, **ssthresh** được thiết lập về chỉ  $1/2$  của **cwnd** trước khi mất gói

# Tóm tắt: TCP điều khiển tắc nghẽn

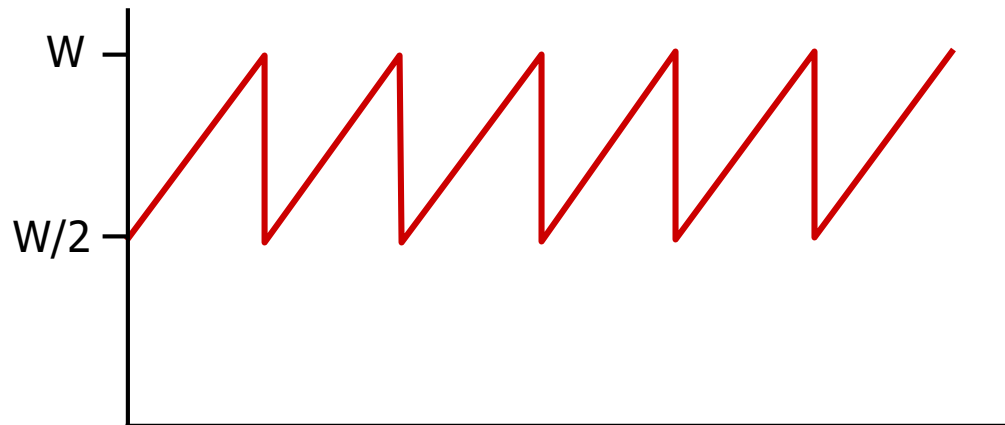




# TCP thông lượng (throughput)

- ❖ Thông lượng trung bình của TCP như là chức năng của kích thước cửa sổ và RTT?
  - Bỏ qua slow start, giả sử dữ liệu luôn luôn được gửi
- ❖  $W$ : kích thước cửa sổ (được đo bằng byte) khi mất gói xảy ra
  - Kích thước cửa sổ trung bình (# in-flight bytes) là  $\frac{3}{4} W$
  - Thông lượng trung bình là  $3/4W$  mỗi RTT

$$\text{avg TCP thrupt} = \frac{3}{4} \frac{W}{\text{RTT}} \text{ bytes/sec}$$



# TCP tương lai: TCP qua “ống lớn và dài”

- ❖ Ví dụ: segment 1500 byte, 100ms RTT, muốn thông lượng 10 Gbps
- ❖ Kích thước cửa sổ yêu cầu  $W = 83,333$  segment trên đường truyền
- ❖ Thông lượng trong các trường hợp mất gói,  $L$  [Mathis 1997]:

$$\text{TCP throughput} = \frac{1.22 \cdot \text{MSS}}{\text{RTT} \sqrt{L}}$$

→ để đạt thông lượng 10 Gbps, cần thì lệ mất gói là  $L = 2 \cdot 10^{-10}$  - *một tỷ lệ mất gói rất nhỏ!*

- ❖ Phiên bản mới của TCP cho tốc độ cao

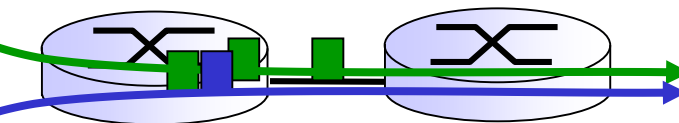
# TCP Công bằng

**Mục tiêu công bằng:** nếu có K session TCP chia sẻ cùng đường link bị bóp cổ chai của băng thông R, thì mỗi phiên nên có tốc độ trung bình là  $R/K$

Kết nối TCP 1



Kết nối TCP 2

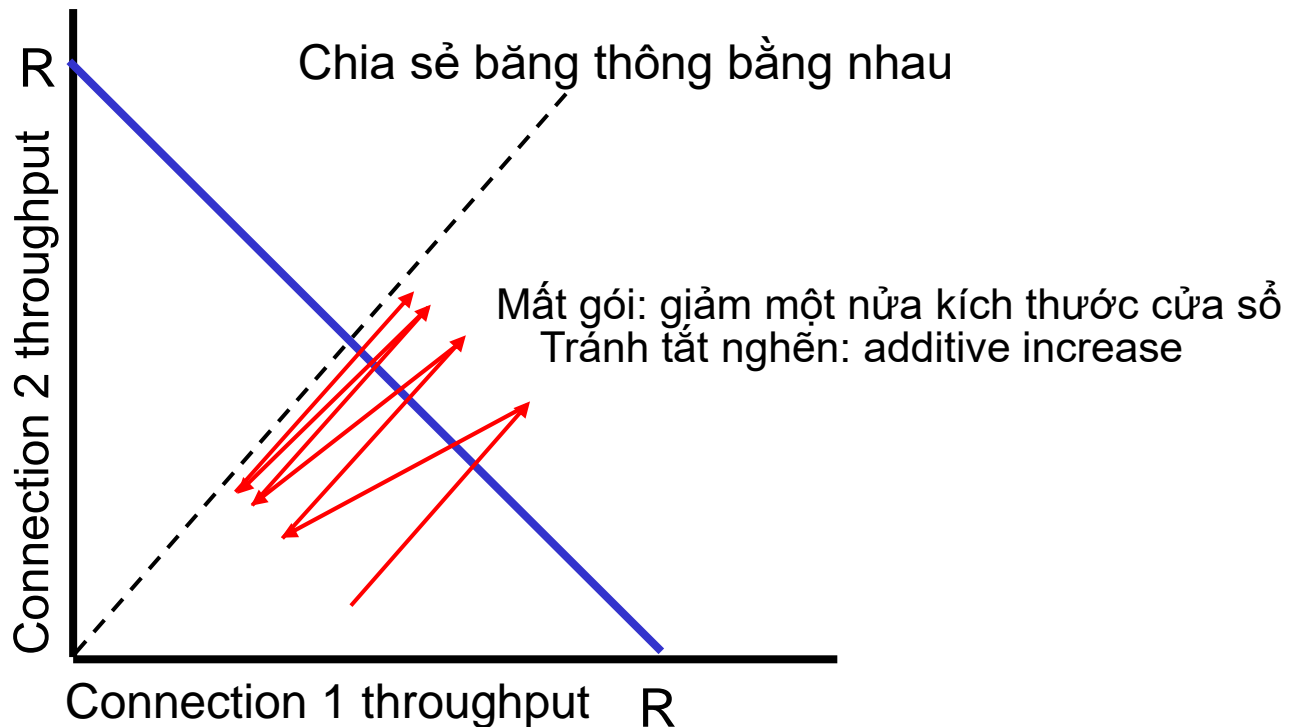


Router cổ chai  
Khả năng R

# Tại sao TCP là công bằng?

2 session cạnh tranh nhau:

- ❖ additive increase cho độ dốc tăng 1, khi thông lượng tăng
- ❖ multiplicative decrease giảm thông lượng tương ứng



# Công bằng (tt)

## *Công bằng và UDP*

- ❖ Nhiều ứng dụng thường không dùng TCP
  - Không muốn tốc độ bị điều tiết do điều khiển tắc nghẽn
- ❖ Thay bằng dùng UDP:
  - Truyền audio/video với tốc độ ổn định, chịu được mất gói

## *Công bằng, các kết nối TCP song song*

- ❖ ứng dụng có thể mở nhiều kết nối song song giữa 2 host
- ❖ Trình duyệt web làm điều này
- ❖ Ví dụ: đường link với tốc độ  $R$  hỗ trợ 9 kết nối:
  - ứng dụng mới yêu cầu 1 TCP, có tốc độ  $R/10$
  - ứng dụng mới yêu cầu 11 TCPs, có tốc độ  $R/2$

# Chương 3: Tóm tắt

- ❖ Các nguyên lý của các dịch vụ tầng transport layer :
  - multiplexing, demultiplexing
  - Truyền dữ liệu tin cậy
  - Điều khiển luồng (flow control)
  - Điều khiển tắc nghẽn (congestion control)
- ❖ Khởi tạo và thực hiện trên Internet
  - UDP
  - TCP

## Kế tiếp:

- ❖ Tìm hiểu xong các vấn đề mạng "biên" (các tầng application, transport)
- ❖ Chuẩn bị vào phần mạng "lõi"