Mạng máy tính

TS. Phạm Tuấn Minh

Khoa Công nghệ Thông tin, Đại học Thủy lợi

minhpt@tlu.edu.vn

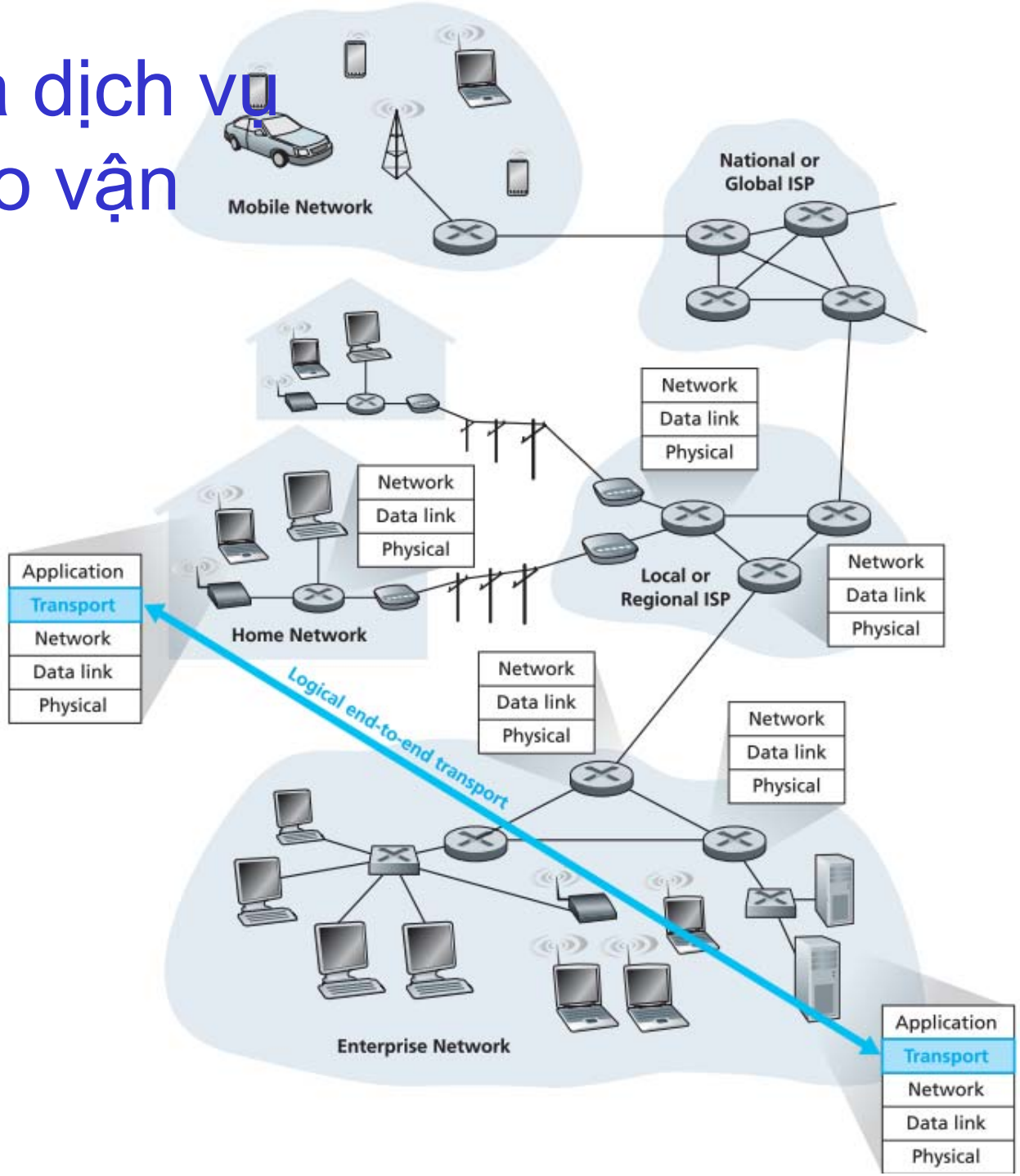
http://netlab.tlu.edu.vn/~minhpt/

Chương 3: Tầng giao vận

- Dịch vụ của tầng giao vận
- Multiplexing và Demultiplexing
- Truyền không kết nối: UDP
- Nguyên tắc của truyền dữ liệu tin cậy
- Truyền hướng kết nối: TCP
- Nguyên tắc của điều khiển tắc nghẽn
- Điều khiển tắc nghẽn của TCP

Giao thức và dịch vụ của tầng giao vận

- cung cấp *truyền thông lô-gic* giữa các tiến trình ứng dụng chạy trên các host khác nhau
- giao thức giao vận chạy trong các end system
  - phía gửi: chia app message thành các *segment*, chuyển tới tầng mạng
  - phía nhận: ghép các segment lại thành message, chuyển tới tầng ứng dụng
- có nhiều hơn một giao thức giao vận cho các ứng dụng
  - Internet: TCP và UDP



So sánh tầng giao vận và tầng mạng

- *tầng mạng:* truyền thông lô-gic giữa các host
- *tầng giao vận:* truyền thông lô-gic giữa các tiến trình
  - dựa vào, nâng cao, các dịch vụ của tầng mạng

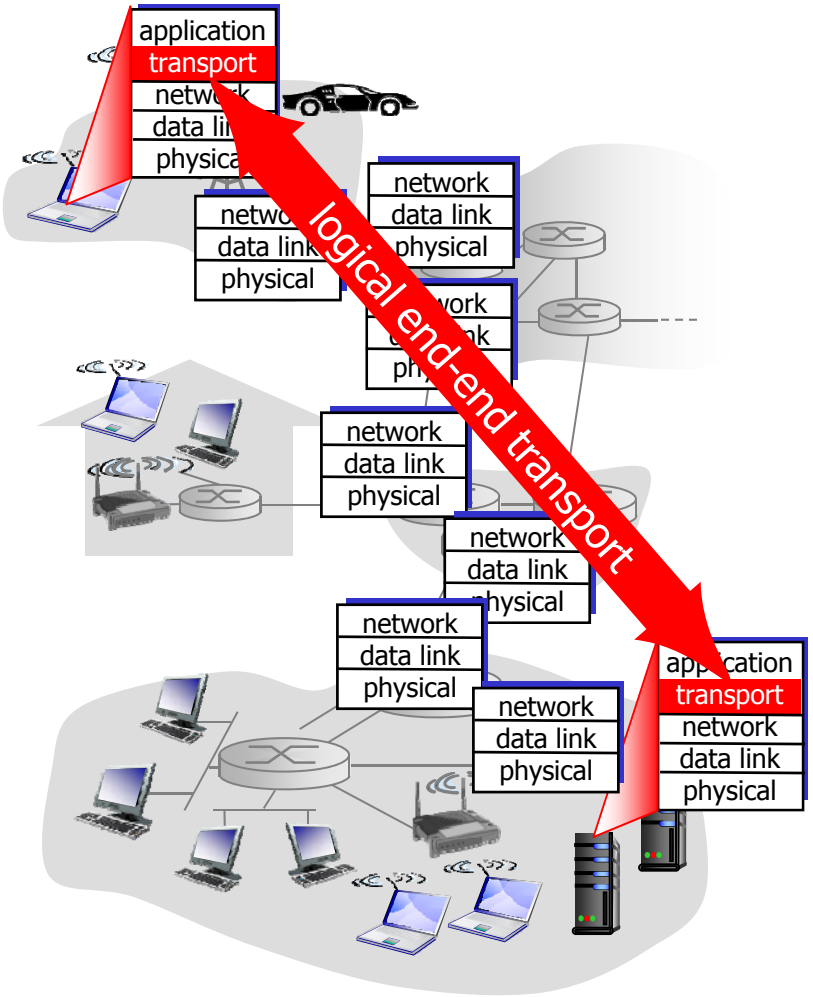
một so sánh:

12 trẻ trong nhà của Ann gửi thư cho 12 trẻ trong nhà của Bill:

- host = nhà
- tiến trình = trẻ
- app message = letter trong phong bì thư
- giao thức giao vận = Ann và Bill
- giao thức tầng mạng = dịch vụ thư bưu điện

Giao thức tầng giao vận của Internet

- tin cậy, chuyển đảm bảo thứ tự (TCP)
  - điều khiển tắc nghẽn
  - điều khiển luồng
  - thiết lập kết nối
- không tin cậy, chuyển không đảm bảo thứ tự: UDP
- các dịch vụ không cung cấp:
  - đảm bảo độ trễ
  - đảm bảo băng thông



Chương 3: Tầng giao vận

- Dịch vụ của tầng giao vận
- Multiplexing và Demultiplexing
- Truyền không kết nối: UDP
- Nguyên tắc của truyền dữ liệu tin cậy
- Truyền hướng kết nối: TCP
- Nguyên tắc của điều khiển tắc nghẽn
- Điều khiển tắc nghẽn của TCP

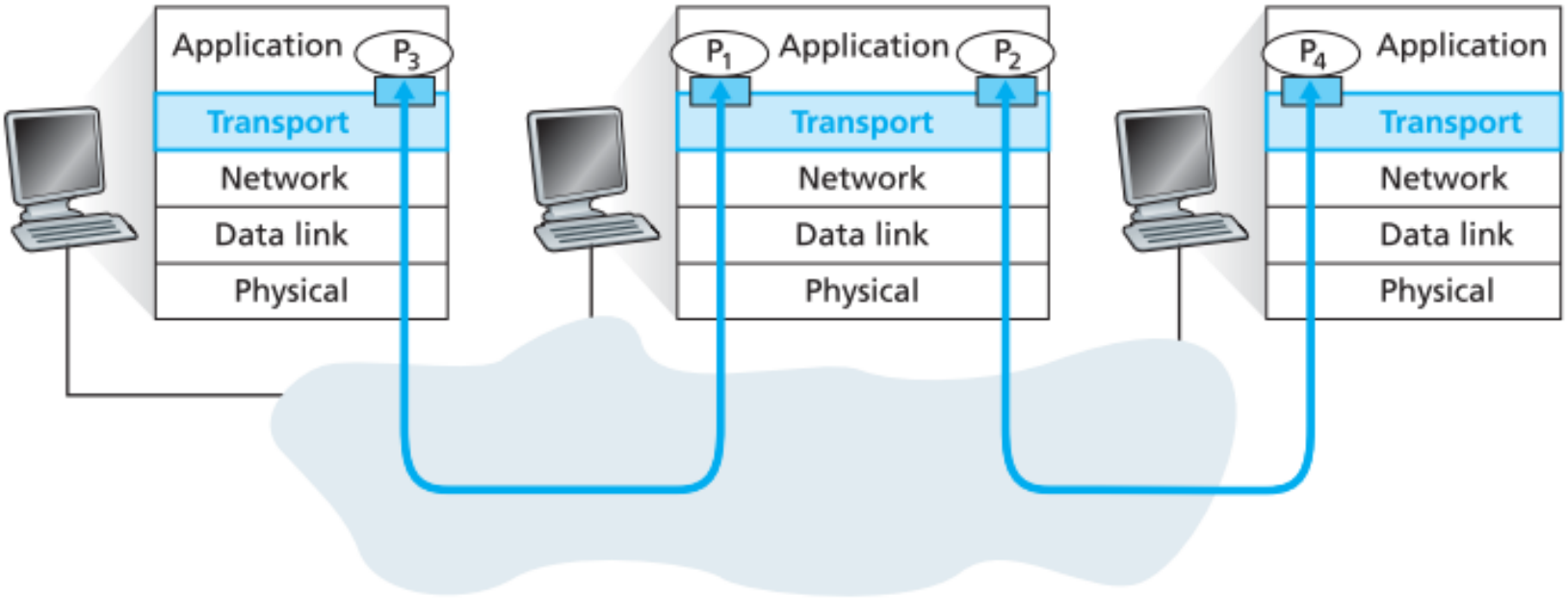
Multiplexing/demultiplexing

ghép kênh (multiplexing) tại nút gửi

chuyển dữ liệu từ nhiều socket, thêm transport header (sau đó sẽ sử dụng để demultiplexing)

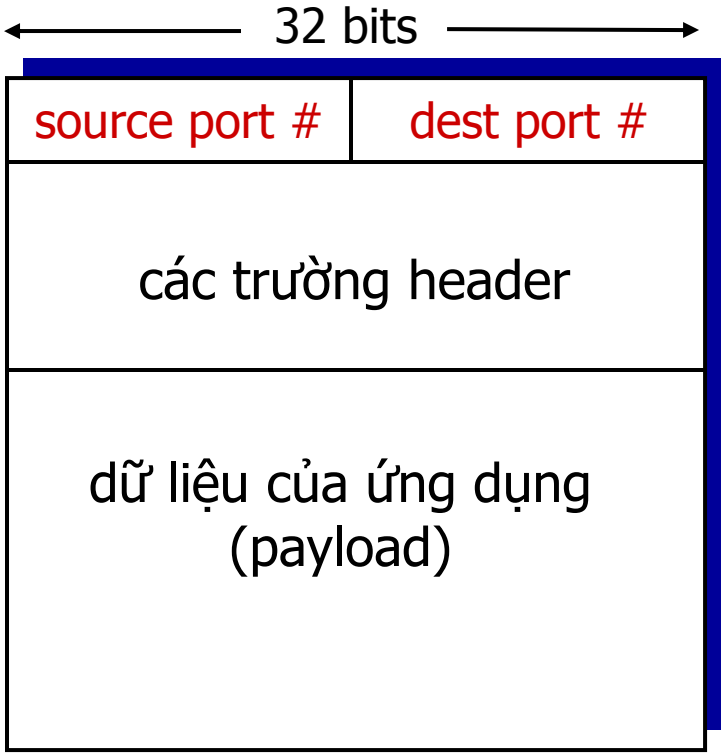
Tách kênh (demultiplexing) tại nút nhận

sử dụng thông tin header để chuyển các segment đã nhận tới đúng socket



Demultiplexing

- host nhận IP datagrams
  - mỗi datagram có source IP address, destination IP address
  - mỗi datagram mạng một transport-layer segment
  - mỗi segment có source port number và destination port number
- host sử dụng *IP address và port number* để chuyển segment tới socket thích hợp



cấu trúc TCP/UDP segment

Connectionless Multiplexing, Connectionless Demultiplexing

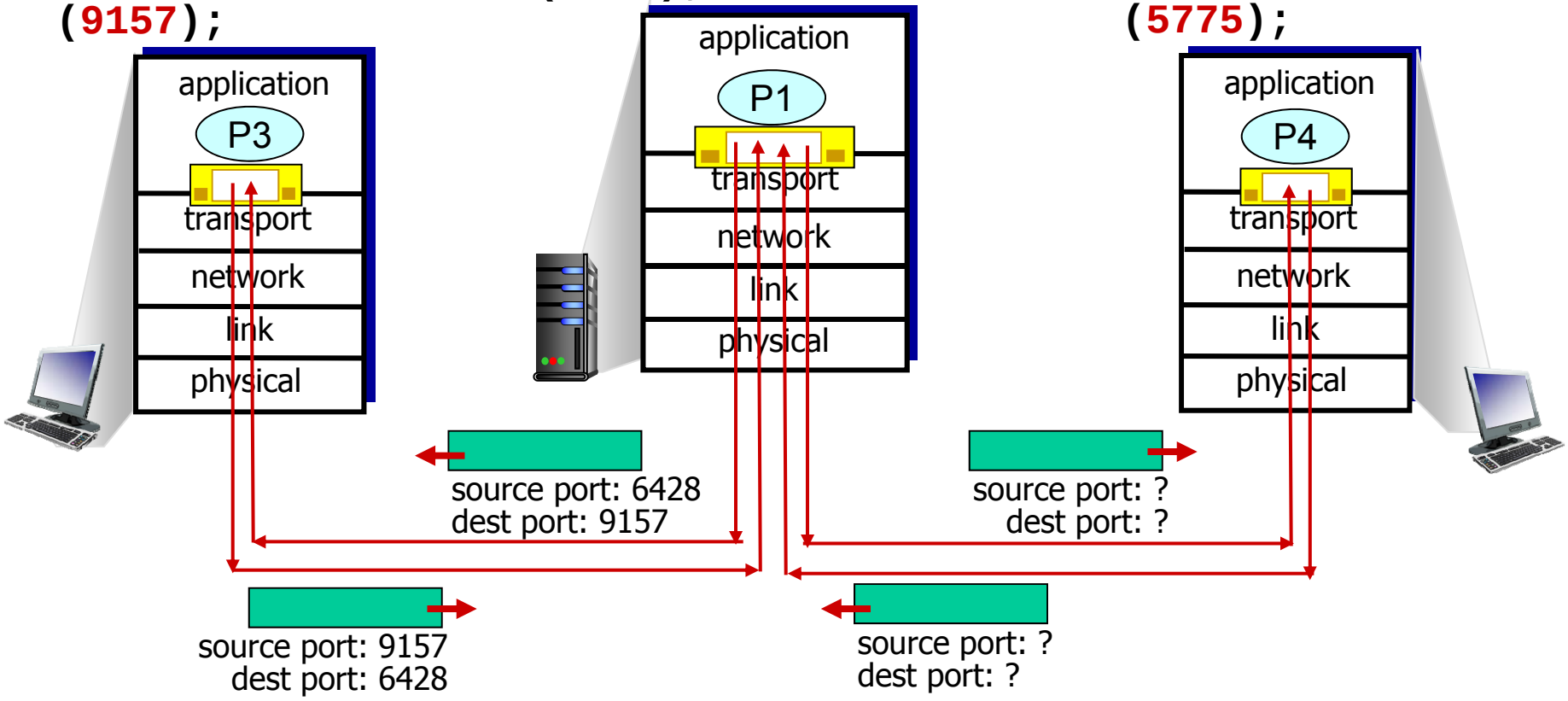
- Chương trình Java chạy trong một host có thể tạo một UDP socket như sau
  - DatagramSocket mySocket = new DatagramSocket( ) ;
  - DatagramSocket mySocket = new DatagramSocket(19157);
- UDP socket được xác định bởi two-tuple
  - a destination IP address
  - a destination port number
  - kết quả ?
- Khi UDP segment tới từ mạng
  - Host B chuyển (demultiplex) từng segment tới socket thích hợp bằng cách kiểm tra destination port number của segment

Connectionless Multiplexing, Connectionless Demultiplexing

```
DatagramSocket
 mySocket2 = new
 DatagramSocket
 (9157);
```

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

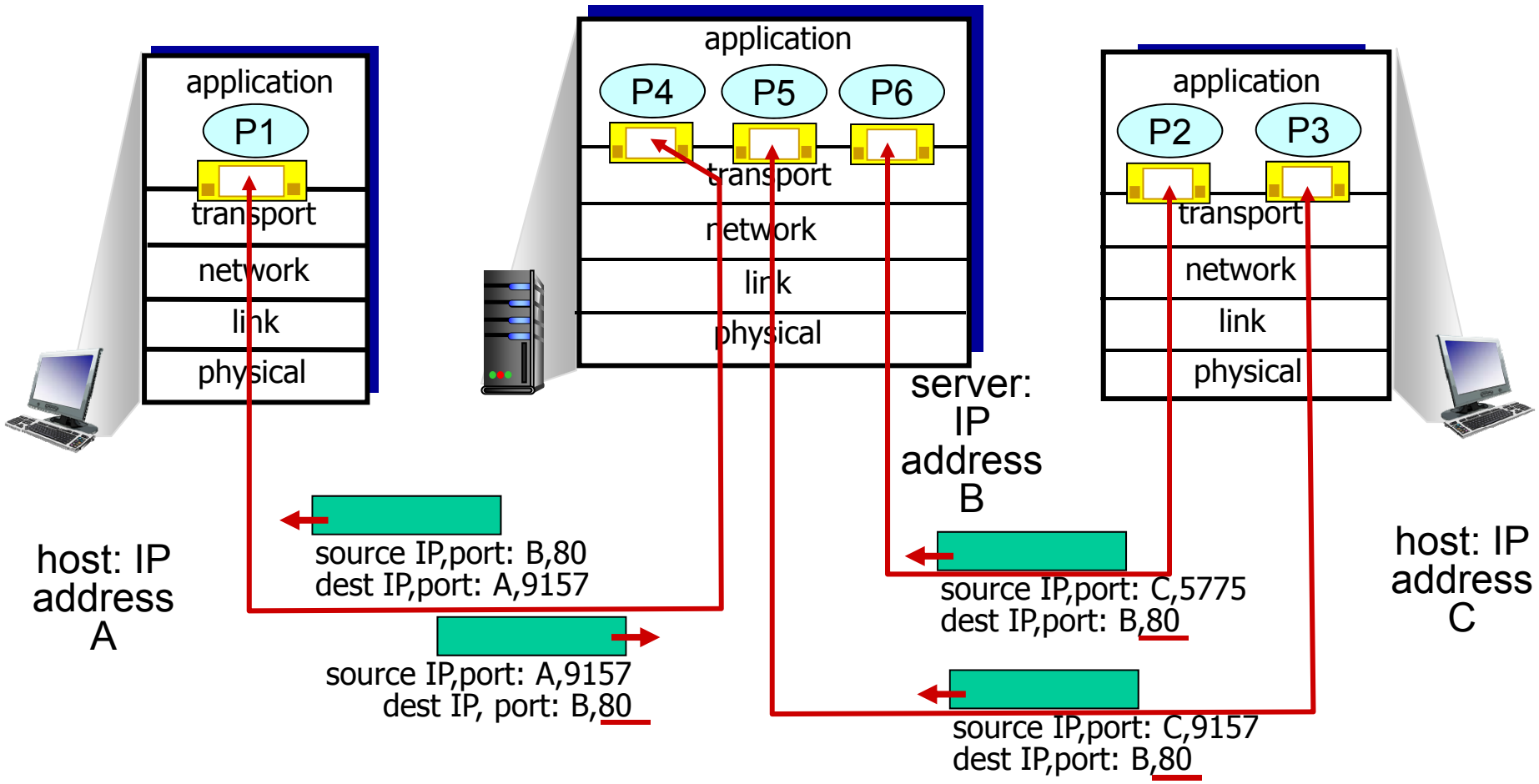
```
DatagramSocket
 mySocket1 = new
 DatagramSocket
 (5775);
```



Connection-Oriented Multiplexing, Connection-Oriented Demultiplexing

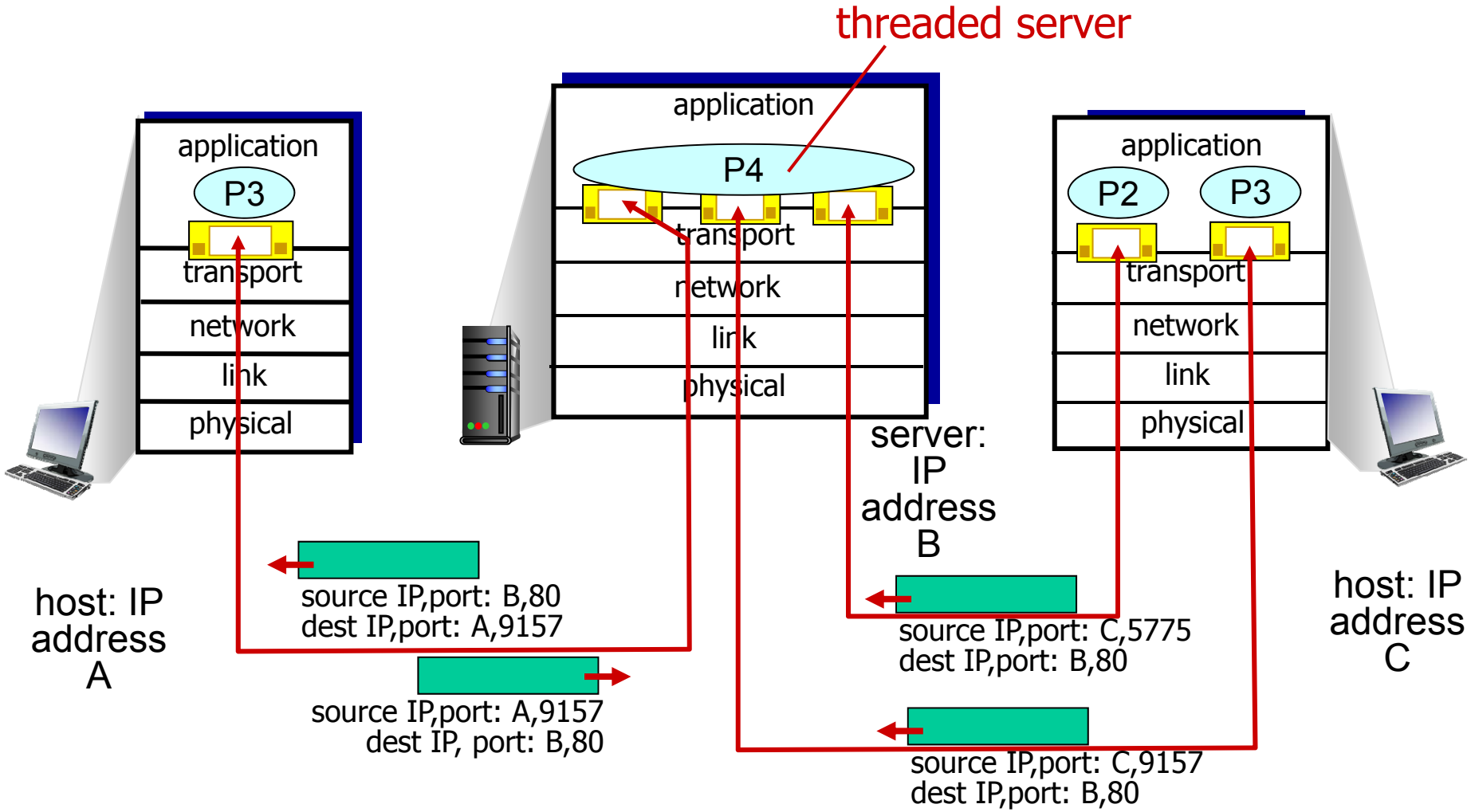
- TCP socket xác định bởi 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- demux: bên nhận sử dụng 4 giá trị để chuyển segment tới đúng socket
- server host có thể hỗ trợ đồng thời nhiều TCP sockets:
  - mỗi socket được xác định bởi 4-tuple của nó

Connection-Oriented Multiplexing, Connection-Oriented Demultiplexing



3 segments, gửi tới IP address: B,
dest port: 80 được tách kênh tới các socket khác nhau

Connection-Oriented Multiplexing and Demultiplexing



Chương 3: Tầng giao vận

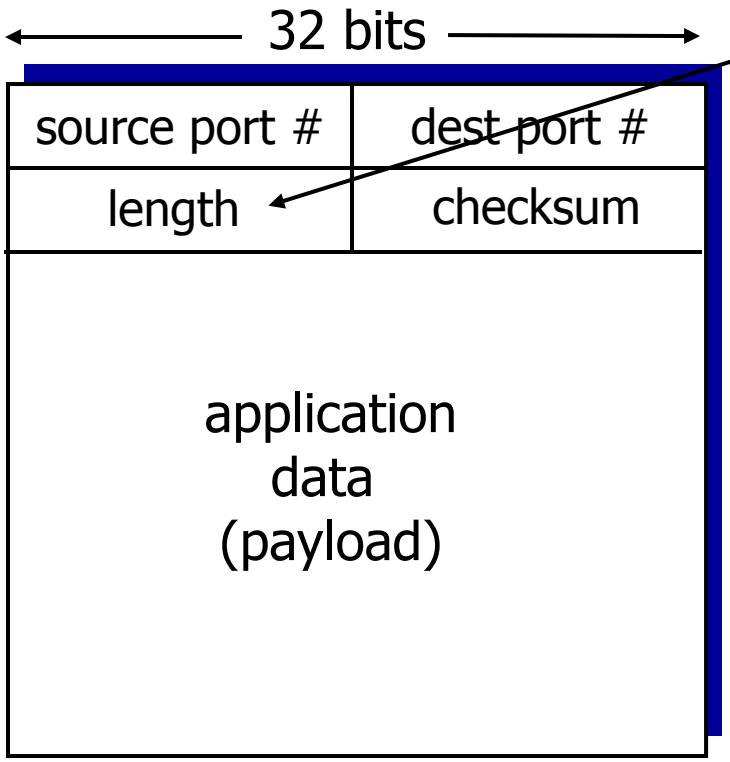
- Dịch vụ của tầng giao vận
- Multiplexing và Demultiplexing
- Truyền không kết nối: UDP
- Nguyên tắc của truyền dữ liệu tin cậy
- Truyền hướng kết nối: TCP
- Nguyên tắc của điều khiển tắc nghẽn
- Điều khiển tắc nghẽn của TCP

UDP: User Datagram Protocol [RFC 768]

- dịch vụ “best effort”, UDP segment có thể:
  - không tới đích
  - tới ứng dụng đích không theo thứ tự gửi
- *không hướng kết nối:*
  - không bắt tay (handshaking) giữa UDP sender và UDP receiver
  - Các UDP segment được xử lý độc lập với nhau

- Sử dụng UDP?
- truyền tin cậy qua UDP?

UDP: segment header



length, tính theo bytes của UDP segment, bao gồm cả header

Tại sao có UDP?

Cấu trúc UDP segment

UDP checksum

Mục đích: phát hiện lỗi (ví dụ đảo bít) trong segment đã gửi

Bên gửi:

- coi dữ liệu của segment, bao gồm cả header, như chuỗi các số nguyên 16 bít
- checksum: cộng (tổng bù 1) toàn bộ các số nguyên này
- bên gửi ghi giá trị checksum vào trường checksum

Bên nhận:

- tính checksum của segment đã nhận được
- kiểm tra xem checksum đã tính có bằng vois giá trị của trường checksum không:
  - Không: phát hiện lỗi
  - Có - không phát hiện lỗi. *Nhưng có thể có lỗi?*

Ví dụ

cộng hai số nguyên 16 bít

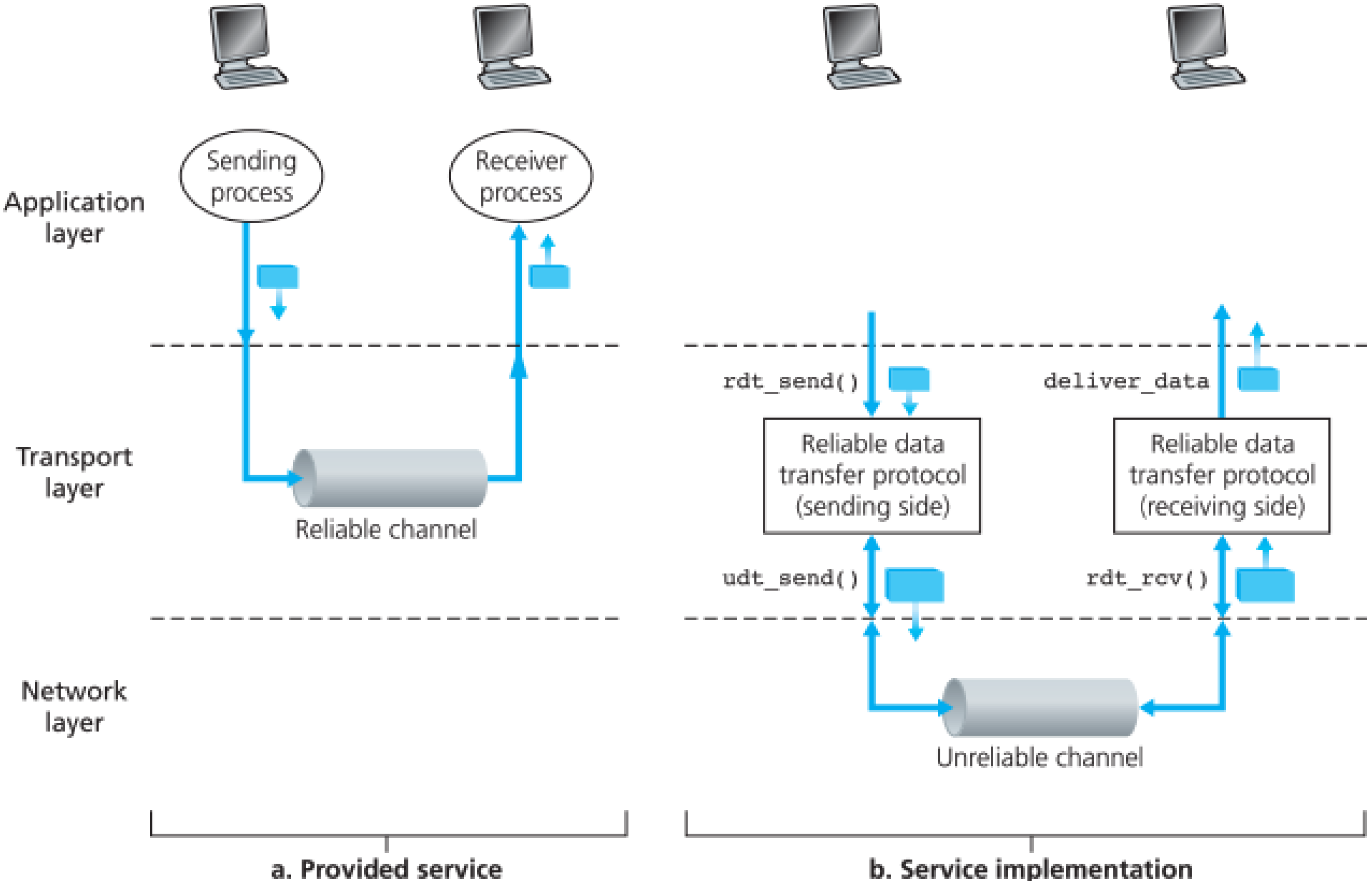
	1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
wraparound	(1)	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
sum	1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum	0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

Tại sao UDP cung cấp checksum khi nhiều giao thức của tầng liên kết (ví dụ giao thức Ethernet) cũng cung cấp kiểm tra lỗi?

Chương 3: Tầng giao vận

- Dịch vụ của tầng giao vận
- Multiplexing và Demultiplexing
- Truyền không kết nối: UDP
- Nguyên tắc của truyền dữ liệu tin cậy
- Truyền hướng kết nối: TCP
- Nguyên tắc của điều khiển tắc nghẽn
- Điều khiển tắc nghẽn của TCP

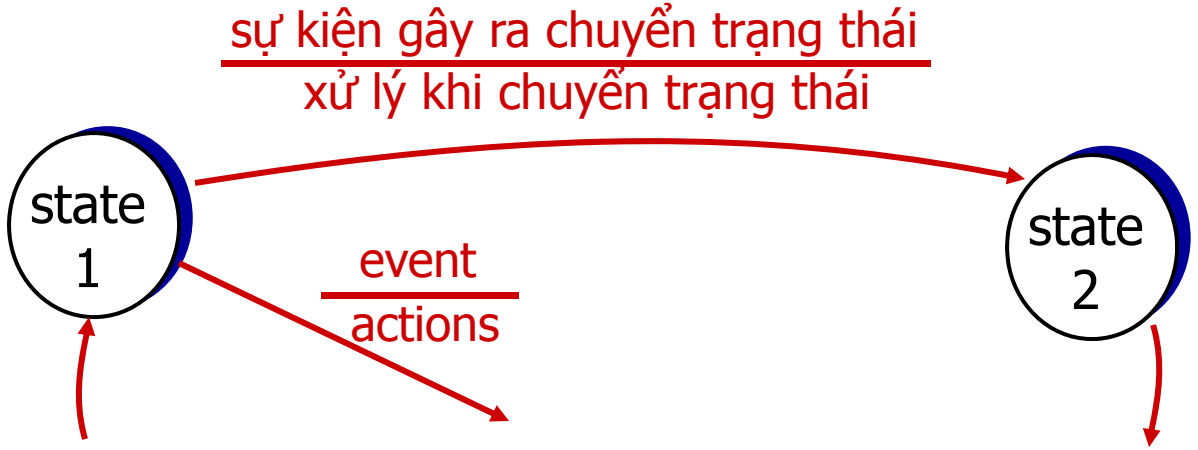
Nguyên tắc của truyền dữ liệu tin cậy



Cách tiếp cận

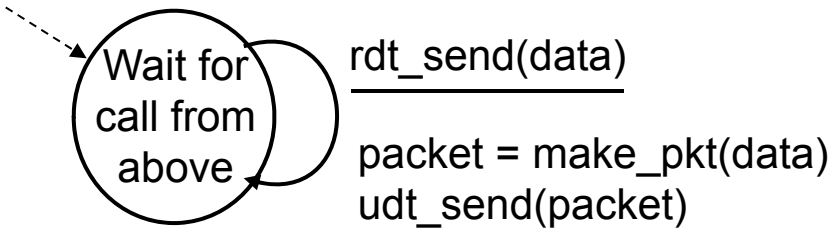
- từng bước phát triển giao thức truyền dữ liệu tin cậy giữa bên gửi và bên nhận (rdt - reliable data transfer protocol)
- xem xét một hướng truyền dữ liệu
  - thông tin điều khiển sẽ truyền theo cả 2 hướng
- sử dụng máy trạng thái hữu hạn (FSM - finite state machines) để mô tả

trạng thái (state): khi ở trong một trạng thái, trạng thái tiếp theo được xác định duy nhất bởi sự kiện

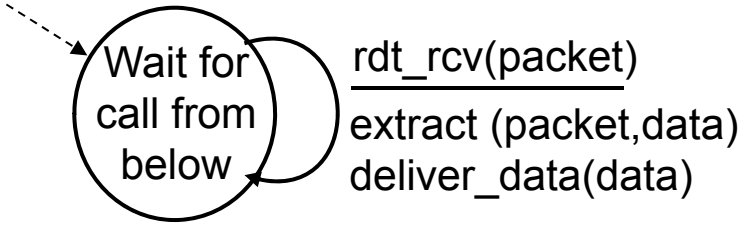


rdt1.0: Truyền dữ liệu tin cậy qua kênh tinh cậy

- kênh truyền phía dưới là tin cậy
  - không có lỗi bít
  - không mất gói tin
- máy trạng thái cho bên gửi và bên nhận
  - bên gửi truyền dữ liệu vào kênh truyền
  - bên nhận đọc dữ liệu từ kênh truyền



sender

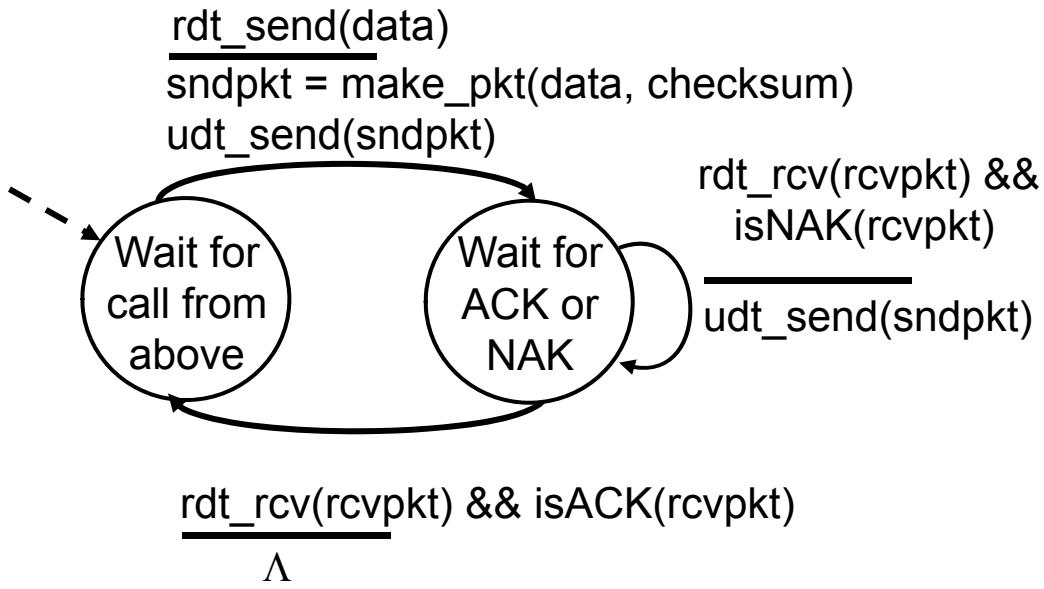


receiver

rdt2.0: Kênh truyền có xảy ra lỗi

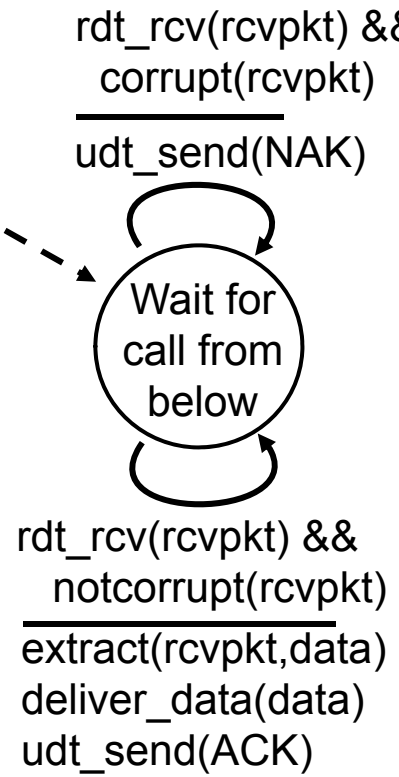
- kênh truyền có thể đảo bít trong gói tin
  - dùng checksum để phát hiện bít bị lỗi
- *cách thức để sửa lỗi?*
  - *acknowledgements (ACKs):* bên nhận thông báo cho bên gửi là gói tin đã nhận không có lỗi
  - *negative acknowledgements (NAKs):* bên nhận thông báo cho bên gửi là gói tin có lỗi
  - bên gửi truyền lại gói tin khi nhận được NAK
- cơ chế mới trong **`rdt2.0`**:
  - phát hiện lỗi
  - phản hồi: gói tin điều khiển (ACK, NAK) từ bên nhận tới bên gửi

Máy trạng thái của rdt2.0

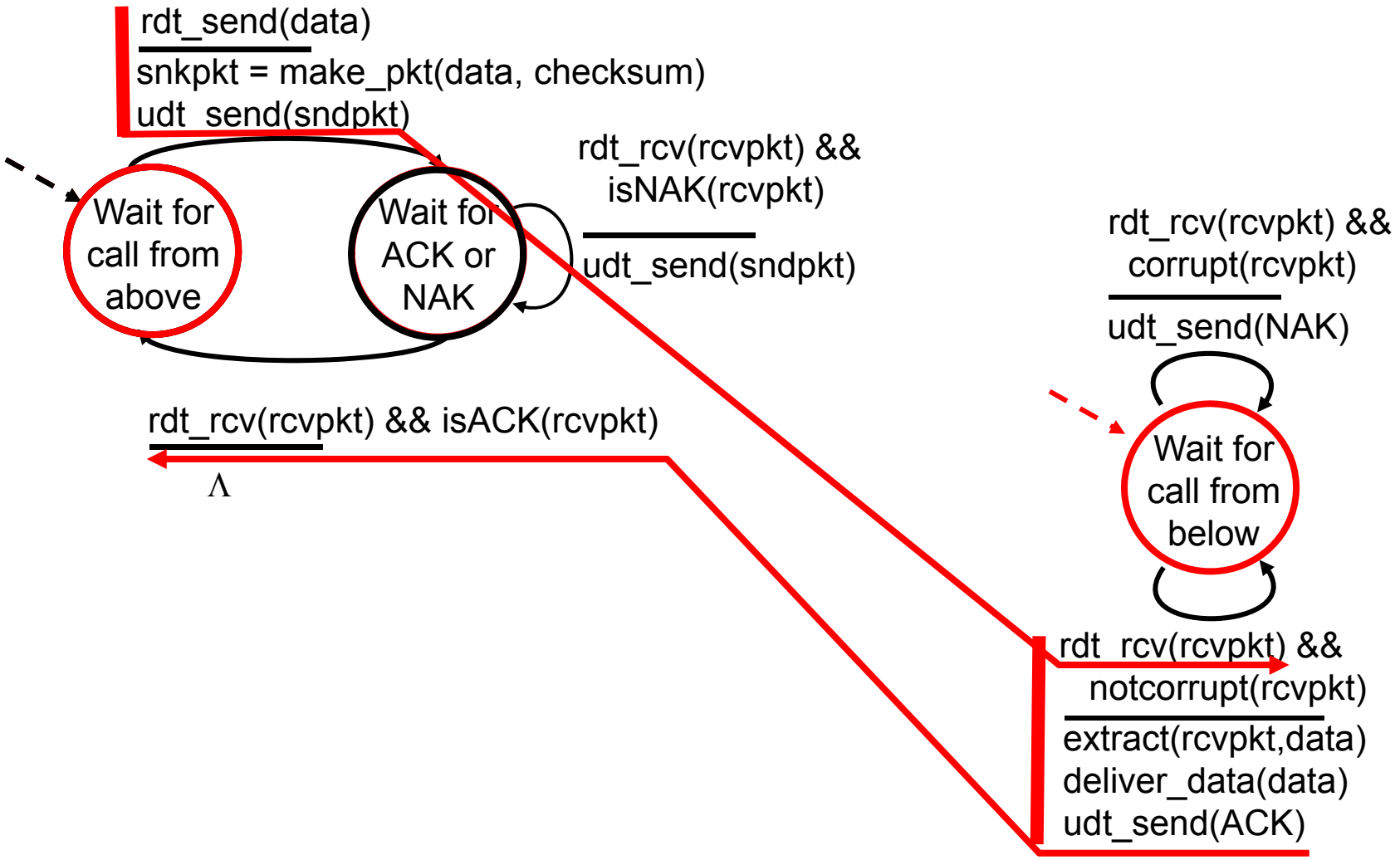


bên gửi

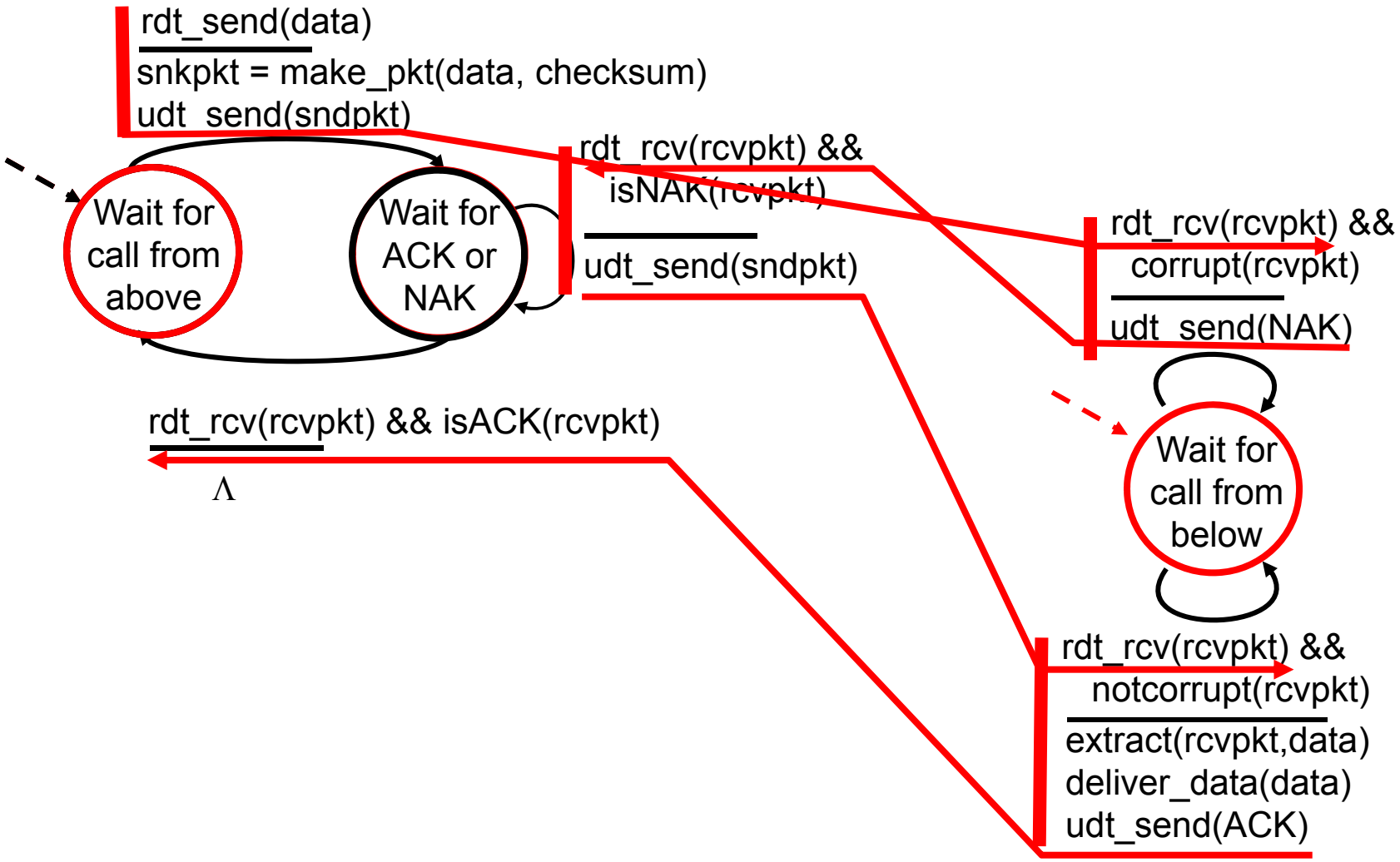
bên nhận



rdt2.0: Hoạt động không có lỗi



rdt2.0: Khi có lỗi xảy ra



Hạn chế của rdt2.0

Điều gì xảy ra khi ACK/NAK có lỗi?

- bên gửi không biết tình trạng của gói tin bên nhận ra sao
- không thể đơn giản là truyền lại: xảy ra trùng lặp

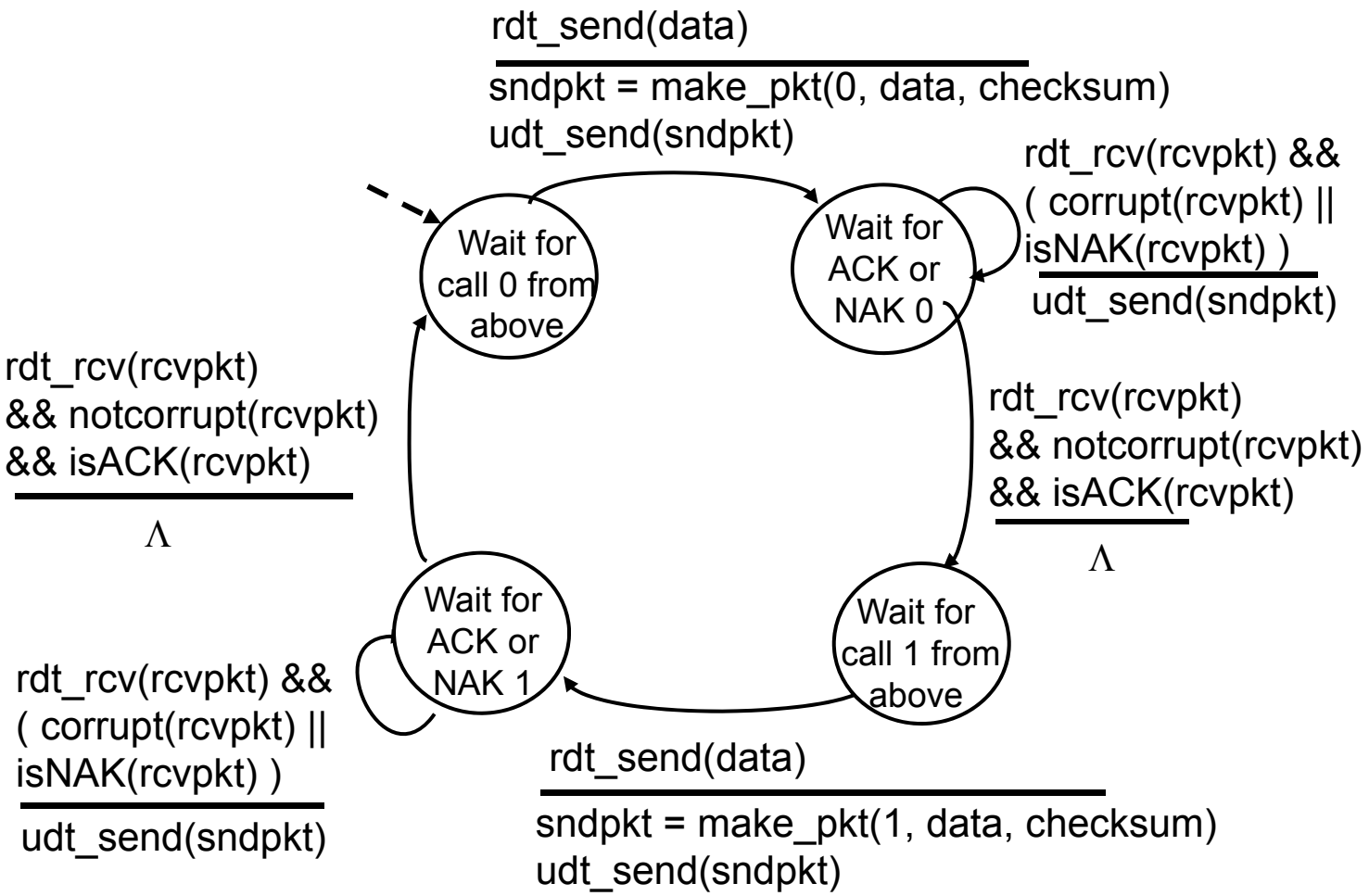
xử lý trùng lặp:

- bên gửi truyền lại gói tin nếu ACK/NAK bị lỗi
- bên gửi thêm *sequence number* vào mỗi gói tin
- bên nhận loại bỏ gói tin lặp (không chuyển lên tầng trên)

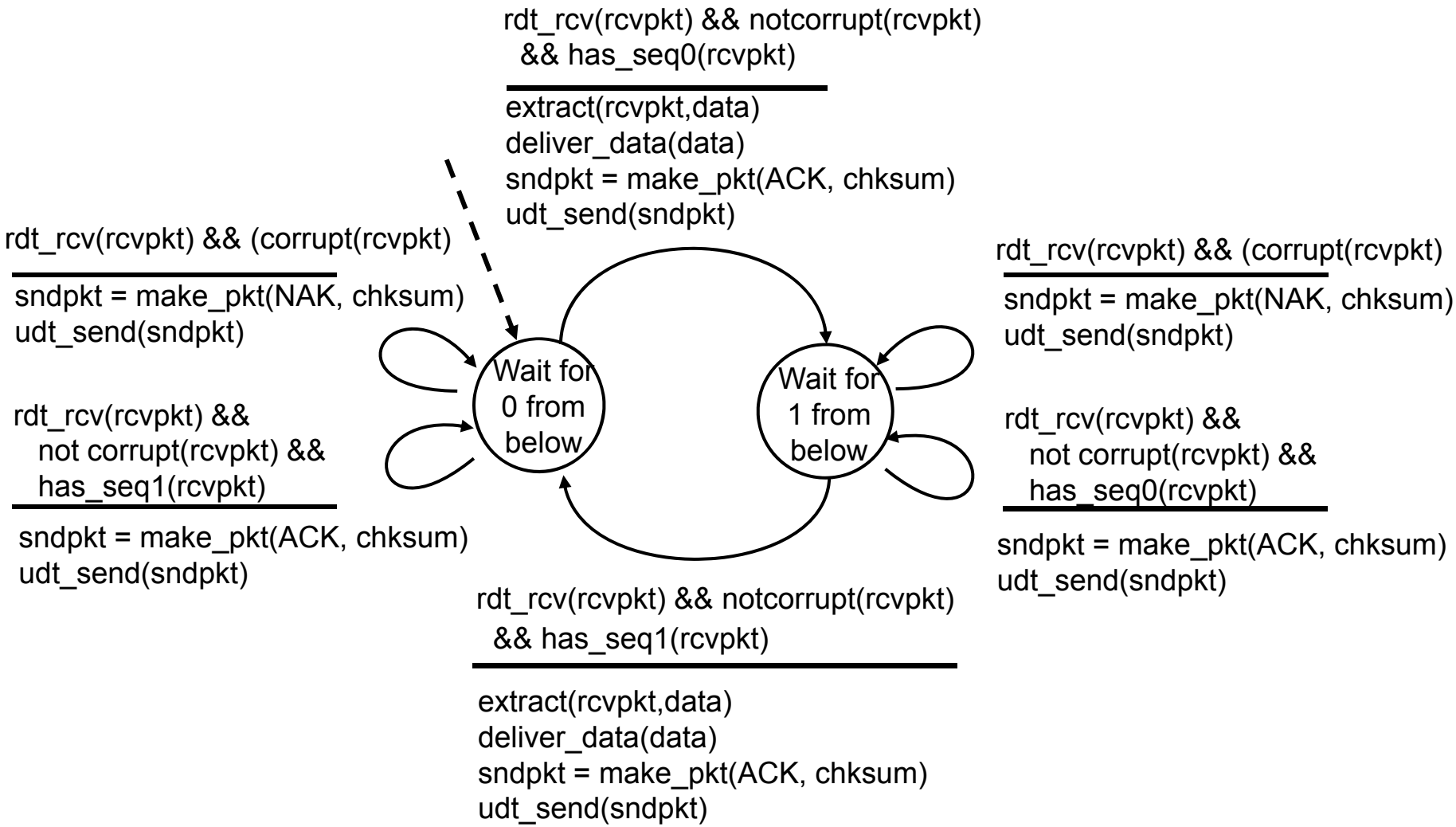
stop and wait

bên gửi gửi một gói tin, rồi đợi trả lời của bên nhận

rdt2.1: Bên gửi, xử lý lý ACK/NAK lỗi



rdt2.1: Bên nhận, xử lý ACK/NAK lỗi



rdt2.1: Thảo luận

Bên gửi:

- seq # thêm vào gói tin
- 2 seq. # (0,1) là đủ. Tại sao?
- phải kiểm tra ACK/NAK có lỗi không
- gấp đôi số trạng thái
  - trạng thái phải nhớ gói tin chờ nhận có seq # 0 hay 1

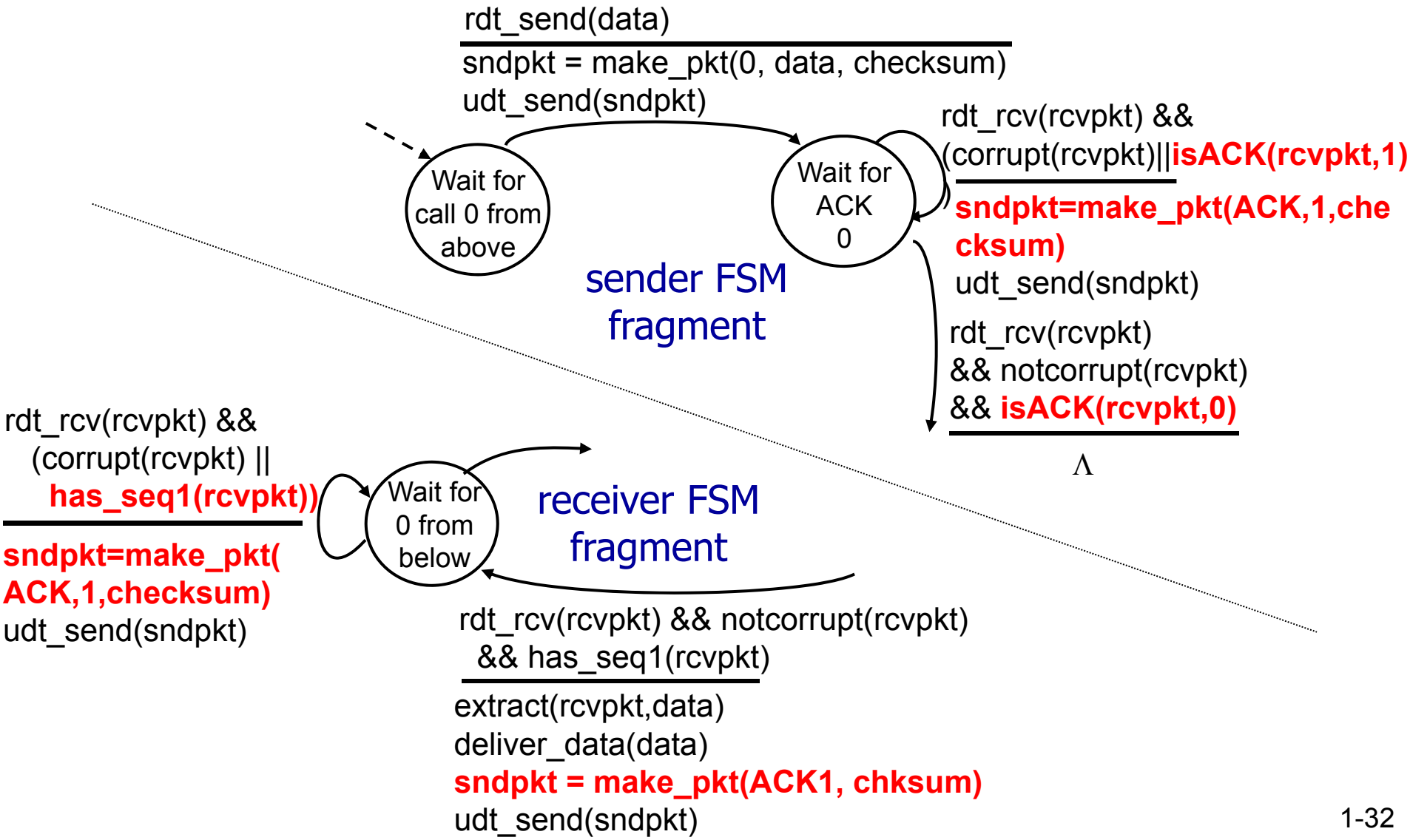
Bên nhận:

- phải kiểm tra xem gói tin đã nhận có bị lặp không
  - trạng thái chỉ ra có phải 0 hay 1 là seq # của gói tin chờ nhận
- lưu ý: bên nhận không thể biết ACK/NAK cuối cùng của nó có nhận được đúng tại bên gửi hay không

rdt2.2: Giao thức không dùng NAK

- Tương tự rdt2.1, nhưng chỉ dùng ACK
- Thay vì NAK, bên nhận gửi ACK cho gói tin cuối cùng mà nó nhận được
  - bên nhận phải chứa seq # của gói tin được ACK
- ACK trùng lặp tại bên nhận xử lý như NAK: truyền lại gói tin hiện tại

rdt2.2: Một phần của bên gửi và bên nhận



rdt3.0: Kênh có lỗi và mất gói

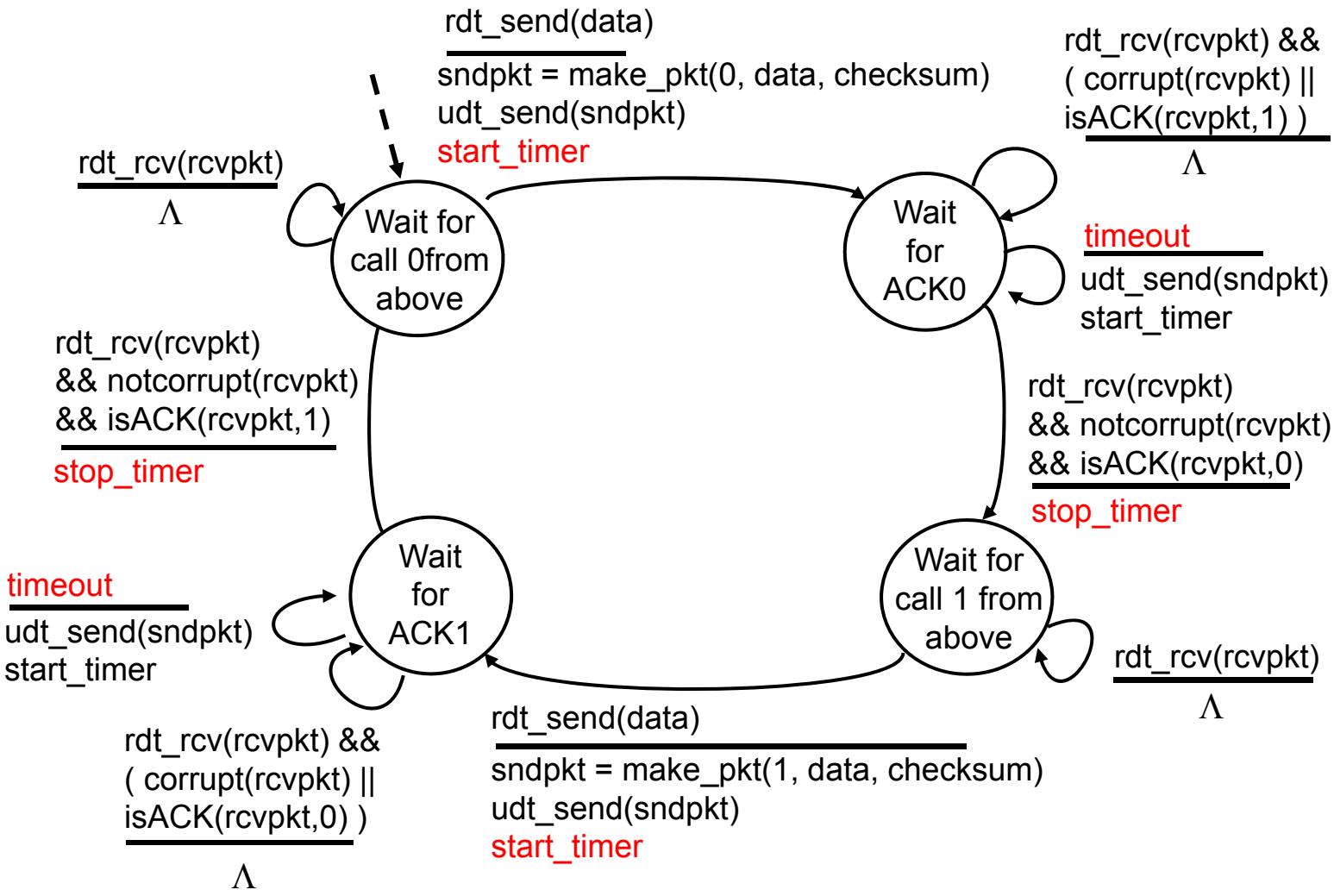
Tình huống: kênh truyền có thể làm mất gói tin (data, ACK)

- checksum, seq. #, ACK, truyền lại là không đủ

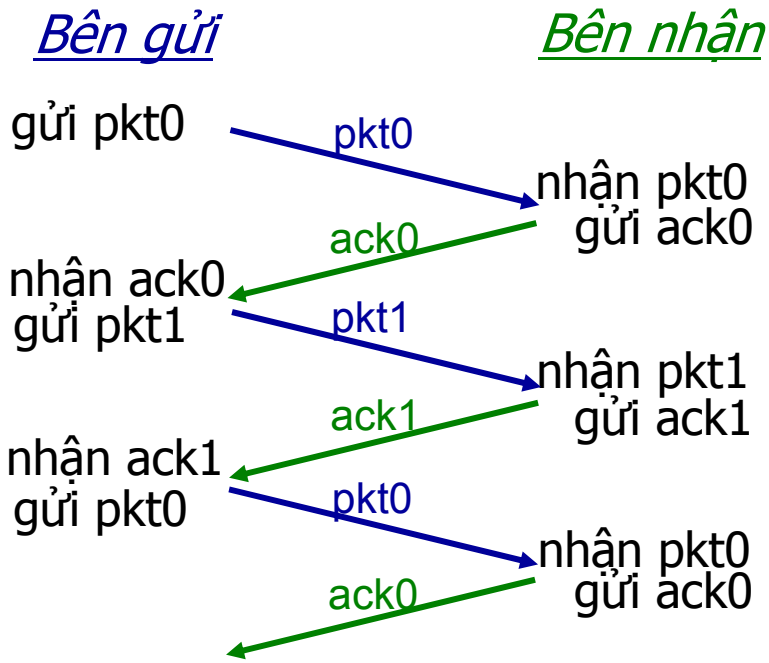
Giải quyết: bên gửi đợi một khoảng thời gian hợp lý cho ACK

- truyền lại nếu không có ACK tới trong thời gian này
- Nếu gói tin (hoặc ACK) đến trễ (không phải là bị mất) :
  - truyền lại sẽ gây trùng lặp nhưng được xử lý dựa vào seq. #
  - bên nhận phải chỉ ra seq # của gói tin được ACK
- cần bộ đếm thời gian

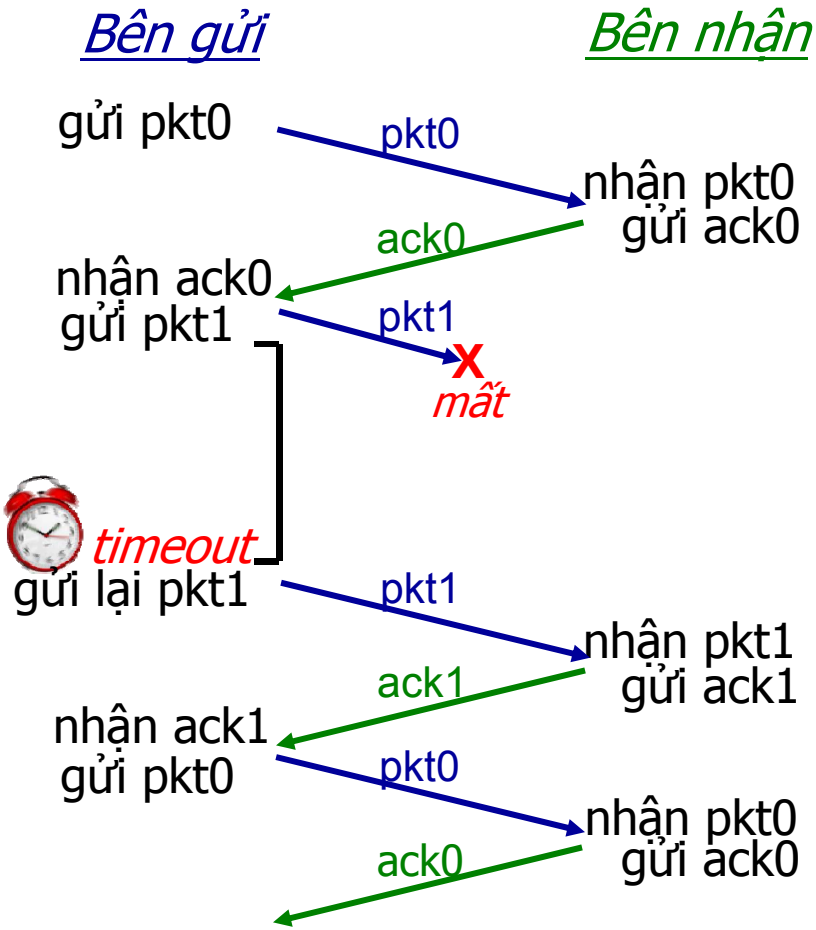
rdt3.0: Bên gửi



rdt3.0: Ví dụ

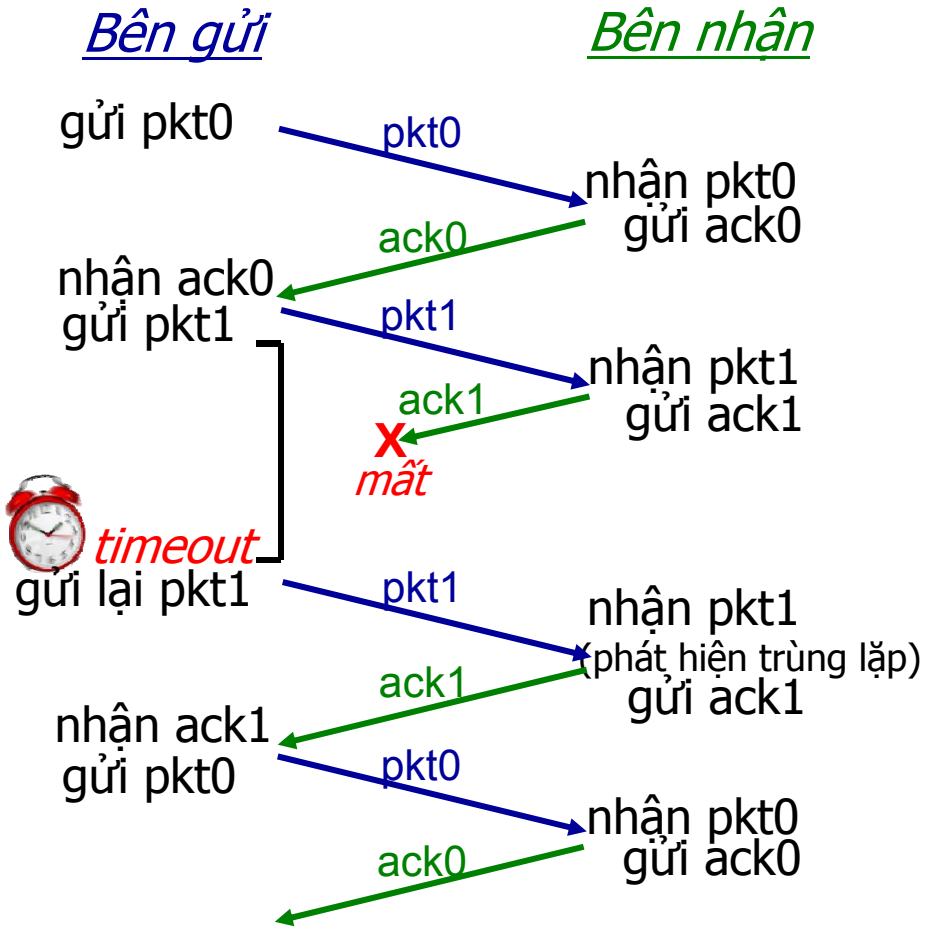


(a) không mất gói

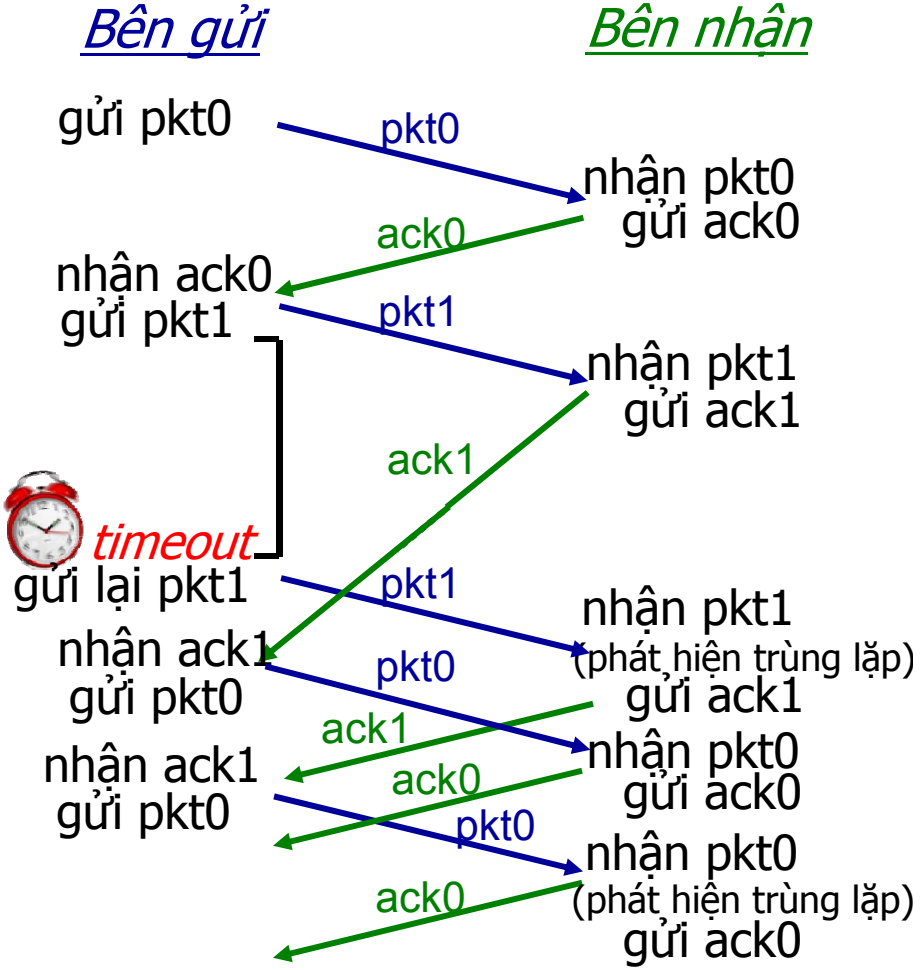


(b) có mất gói

rdt3.0: Ví dụ



(c) mất ACK



(d) timeout sớm/ ACK tới trễ

Hiệu năng của rdt3.0

- Liên kết 1 Gbps, độ trễ lan truyền 15 ms, gói tin 1KB

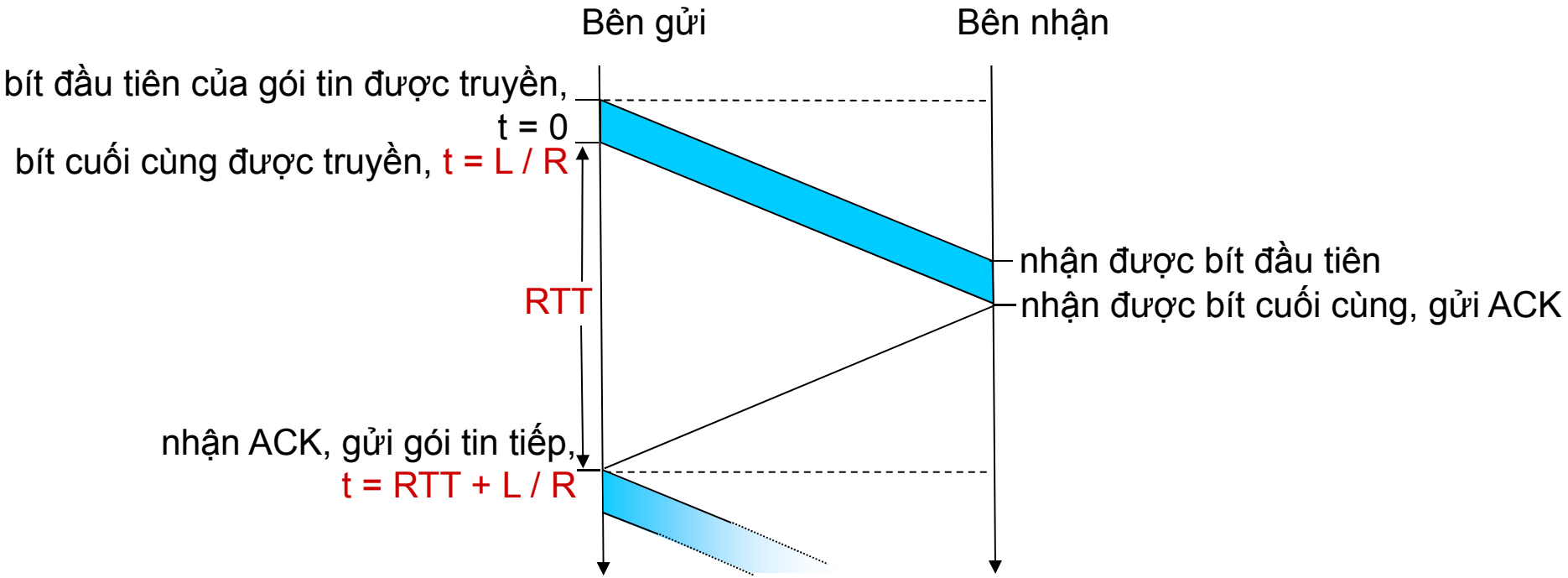
$$D_{trans} = \frac{L}{R} = \frac{8000\ bits}{10^9\ bits/sec} = 8\ microsecs$$

- U_{sender}: *utilization* – phần thời gian bên gửi thực hiện gửi

$$U_{sender} = \frac{L/R}{RTT + L/R} = \frac{0{,}008}{30{,}008} = 0{,}00027$$

  - nếu RTT=30 ms, gói tin 1KB mất 30 ms: thông lượng 267 kilobits / giây trên liên kết 1 Gbps

rdt3.0: xử lý stop-and-wait

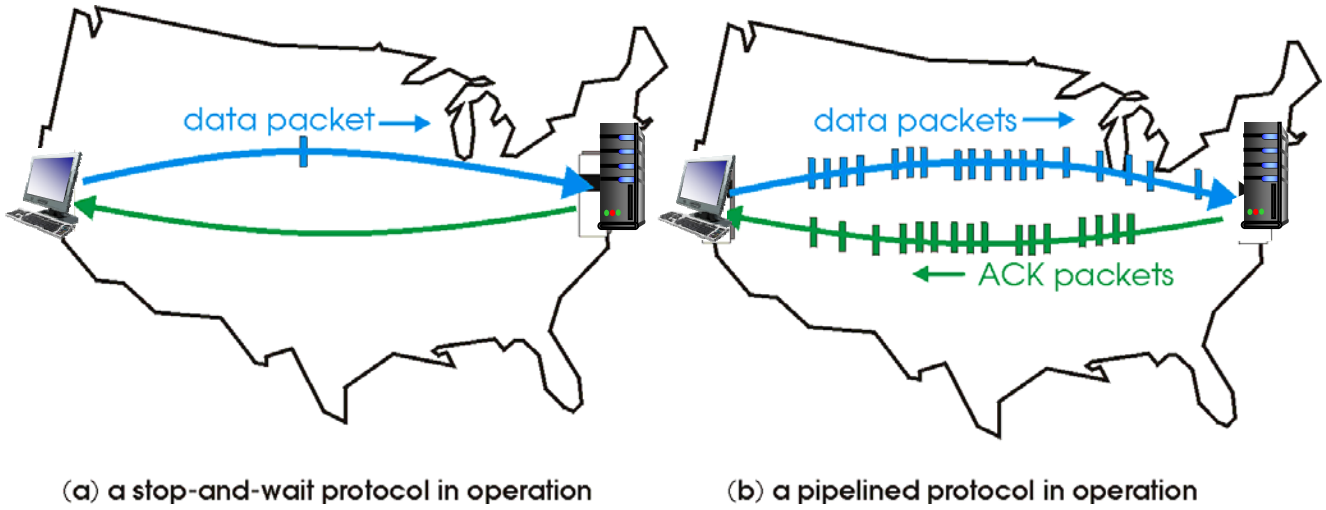


$$U_{sender} = \frac{L/R}{RTT + L/R} = \frac{.008}{30.008} = 0.00027$$

Giao thức được xử lý liên tục

Giao thức được xử lý liên tục (Pipelined protocols):
bên gửi cho phép nhiều gói tin chưa được ack

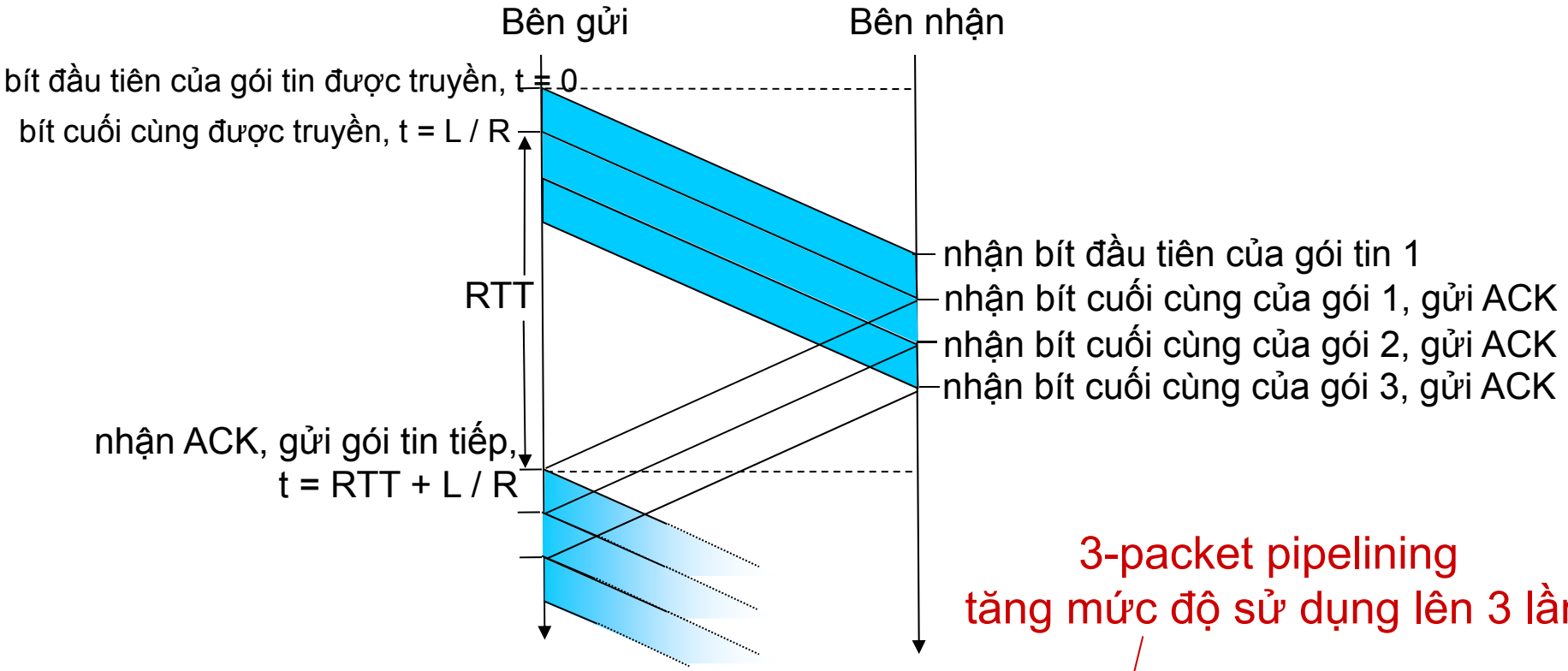
- cần tang dải sequence number
- vùng đệm tại bên gửi và bên nhận



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

- Hai dạng của pipelined protocols: *go-Back-N, selective repeat*

Pipelining: Tăng mức độ sử dụng



3-packet pipelining
tăng mức độ sử dụng lên 3 lần

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

Pipelined protocols

Go-back-N:

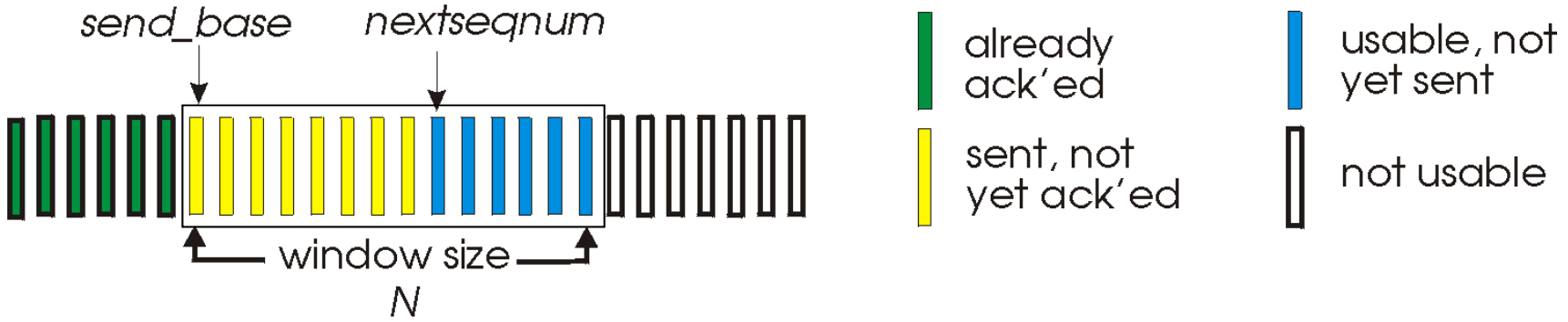
- Bên gửi có thể có tới N gói tin chưa được ACK
- Bên nhận gửi ack cho nhóm gói tin (*cumulative ack)*
  - không ack gói tin nếu có khoảng trống
- Bên gửi có đồng hồ cho gói tin chưa ack lâu nhất
  - khi hết thời gian, gửi lại mọi gói tin chưa ack

Selective Repeat:

- Bên gửi có thể có N gói tin chưa ack
- Bên nhận gửi ack cho mỗi gói tin (*individual ack*)
- Bên gửi có đồng hồ cho mỗi gói tin chưa ack
  - khi hết thời gian, chỉ truyền lại gói tin chưa ack

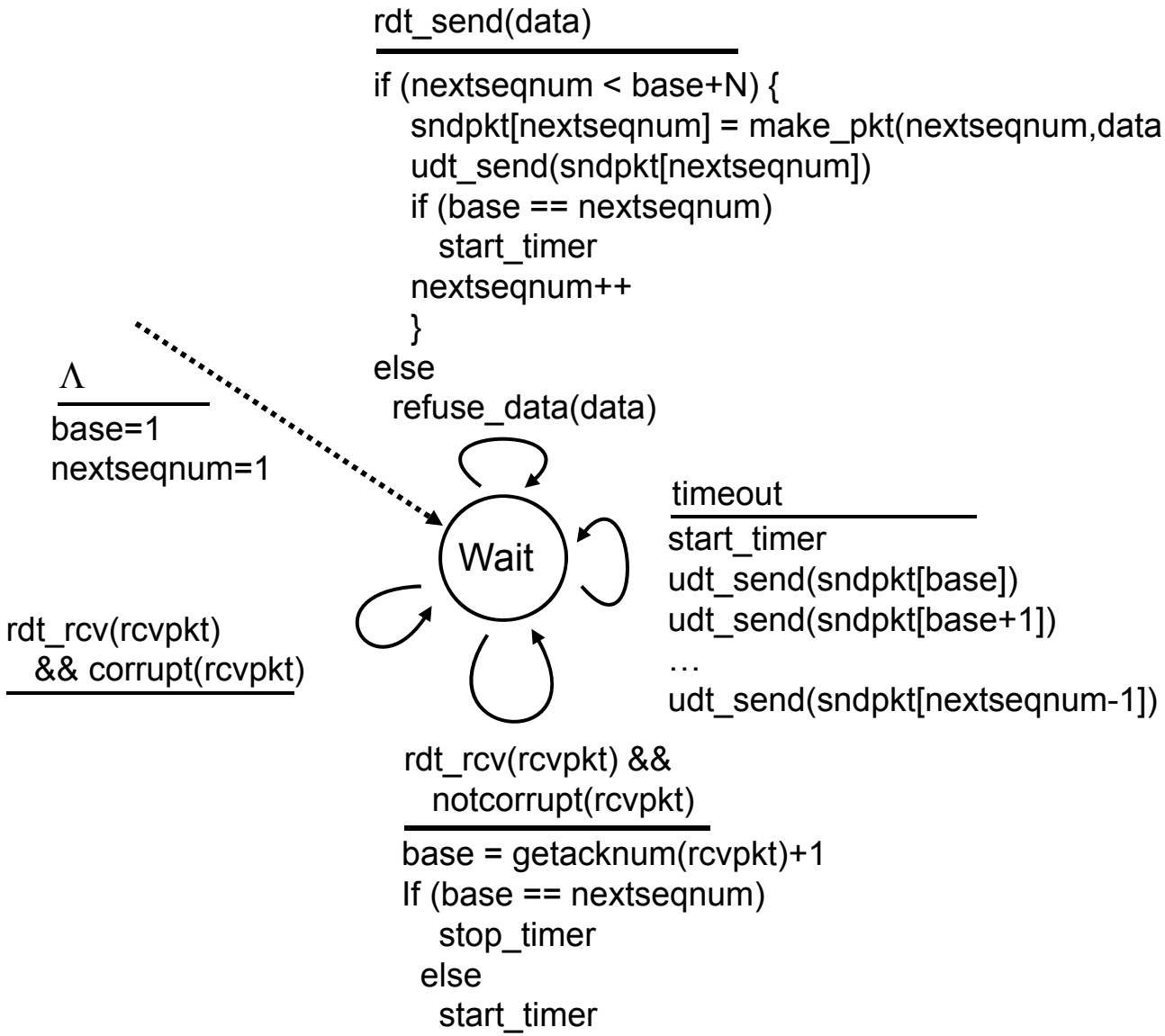
Go-Back-N: Bên gửi

- k-bit seq # trong header của gói tin
- "window" kích thước N, số gói tin liên tục chưa được ack

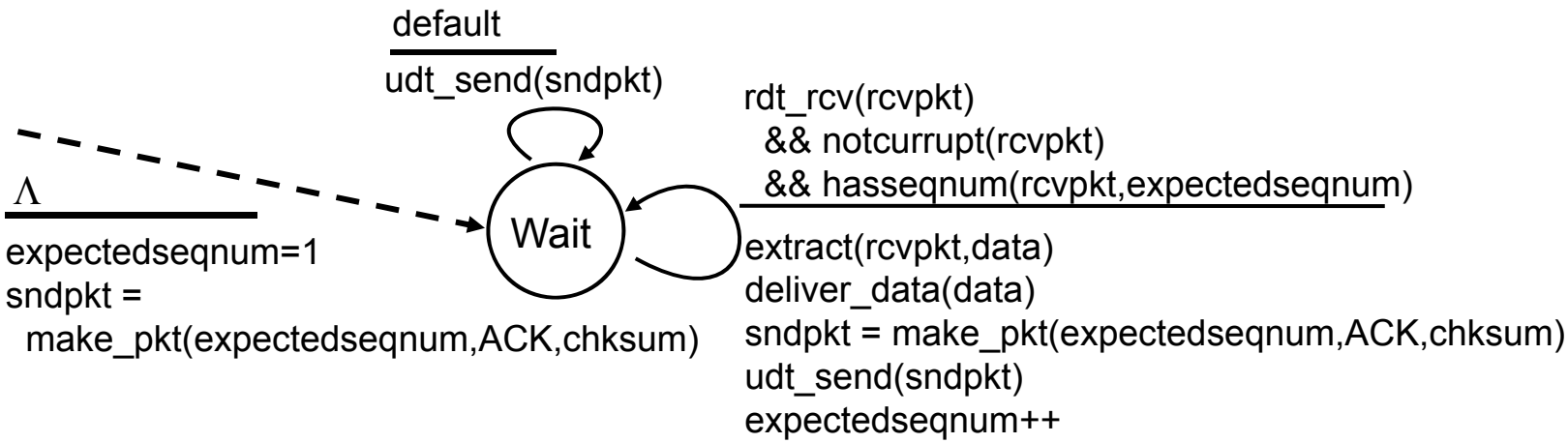


- ACK(n): ACK tất cả các gói tin tới seq # n - *"cumulative ACK"*
  - có thể nhận ACK trùng lặp (xem bên nhận)
- đồng hồ cho gói tin cũ nhất chờ ack
- *timeout(n):* gửi lại gói tin n và tất cả gói tin có seq # lớn hơn trong window

GBN: FSM bên gửi



GBN: FSM bên nhận

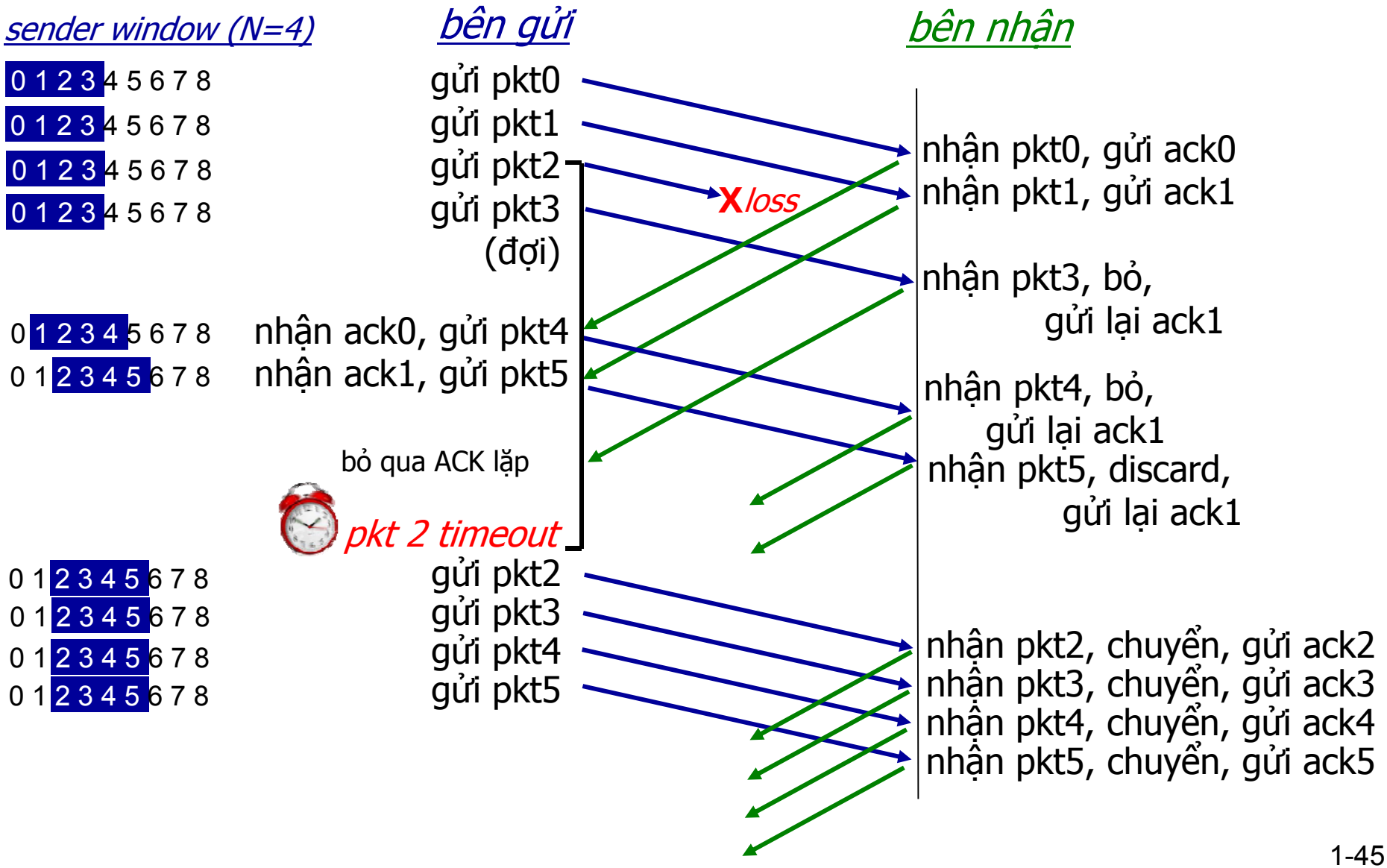


ACK-only: luôn gửi ACK cho gói tin nhận đúng với seq # đúng thứ tự lớn nhất

- có thể gây trùng lặp ACK
- chỉ cần nhớ **`expectedseqnum`**

- gói tin không đúng thứ tự
  - bỏ (không lưu vào vùng đệm): *bên nhận không cần vùng đệm*
  - ack lại gói tin với seq # đúng thứ tự cao nhất

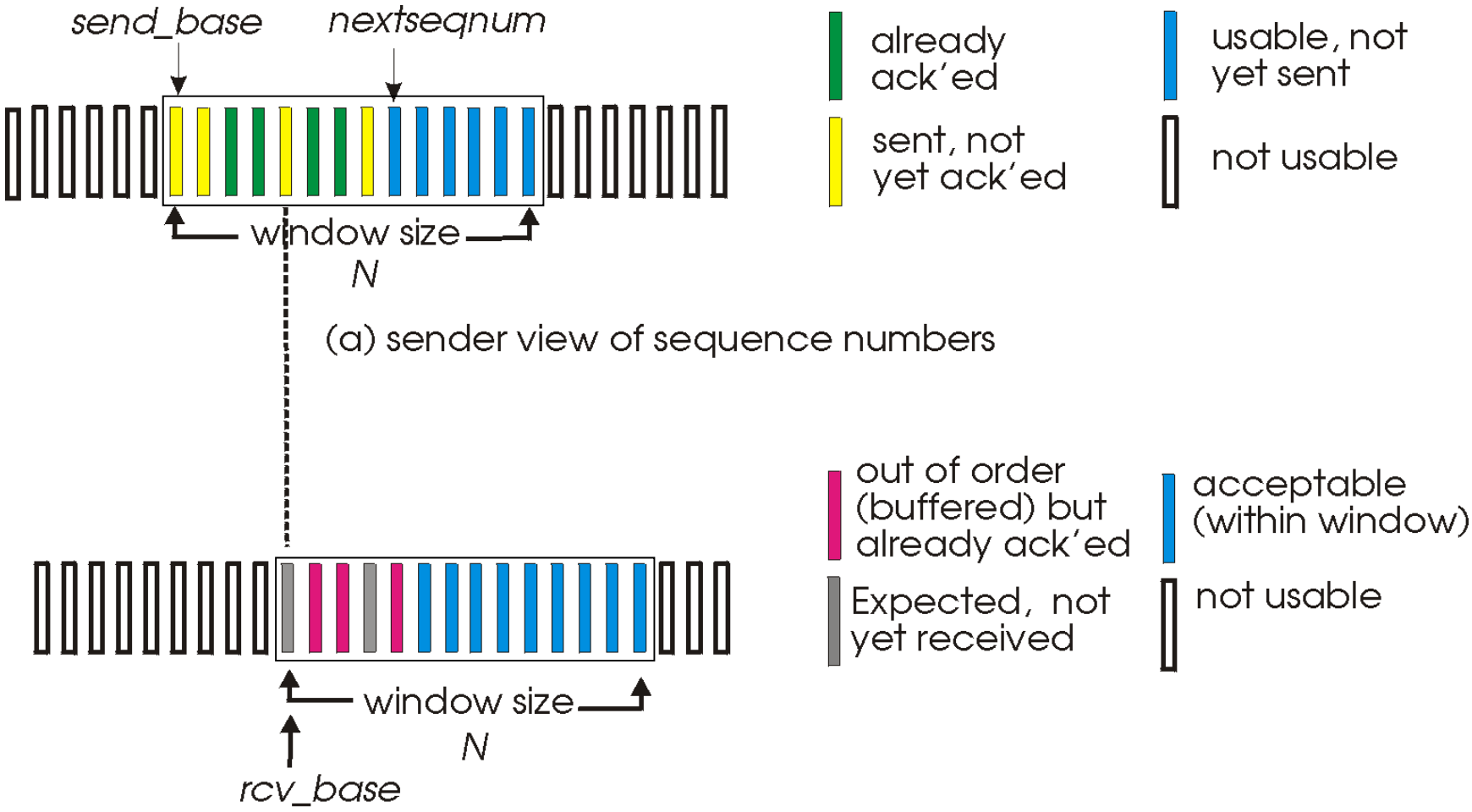
GBN: Ví dụ


sender window (N=4)
bên gửi
bên nhận
0 1 2 3 4 5 6 7 8
gửi pkt0
gửi pkt1
gửi pkt2
gửi pkt3
(đợi)
nhận pkt0, gửi ack0
nhận pkt1, gửi ack1
X loss
nhận pkt3, bỏ,
gửi lại ack1
0 1 2 3 4 5 6 7 8
nhận ack0, gửi pkt4
0 1 2 3 4 5 6 7 8
nhận ack1, gửi pkt5
nhận pkt4, bỏ,
gửi lại ack1
bỏ qua ACK lặp
nhận pkt5, discard,
gửi lại ack1
pkt 2 timeout
0 1 2 3 4 5 6 7 8
gửi pkt2
gửi pkt3
gửi pkt4
gửi pkt5
nhận pkt2, chuyển, gửi ack2
nhận pkt3, chuyển, gửi ack3
nhận pkt4, chuyển, gửi ack4
nhận pkt5, chuyển, gửi ack5

Selective repeat

- bên nhận ack riêng lẻ cho từng gói tin nhận đúng
  - chứa gói tin vào vùng đệm, khi cần, để chuyển đảm bảo thứ tự cho tầng trên
- bên gửi chỉ gửi lại gói tin mà nó không nhận được ACK
  - đồng hồ của bên gửi cho từng gói tin chưa được ack
- sender window
  - *N* seq # liên tục
  - giới hạn seq # được gửi, gói tin chưa được ack

Selective repeat



Selective repeat

Bên gửi

dữ liệu từ trên:

- nếu có seq # khả dụng trong window, gửi gói tin

timeout(n):

- gửi lại gói tin n, khởi tạo lại đồng hồ

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

- đánh dấu gói tin n đã nhận
- nếu n gói tin chưa ack nhỏ nhất, chuyển window base tới seq # chưa ack tiếp theo

Bên nhận

gói tin n trong [rcvbase, rcvbase+N-1]

- gửi ACK(n)
- không đúng thứ tự: đưa vào vùng đệm
- đúng thứ tự: chuyển (đồng thời chuyển các gói tin đúng thứ tự đã chứa trong vùng đệm), chuyển window tới gói tin chưa nhận được ack tiếp

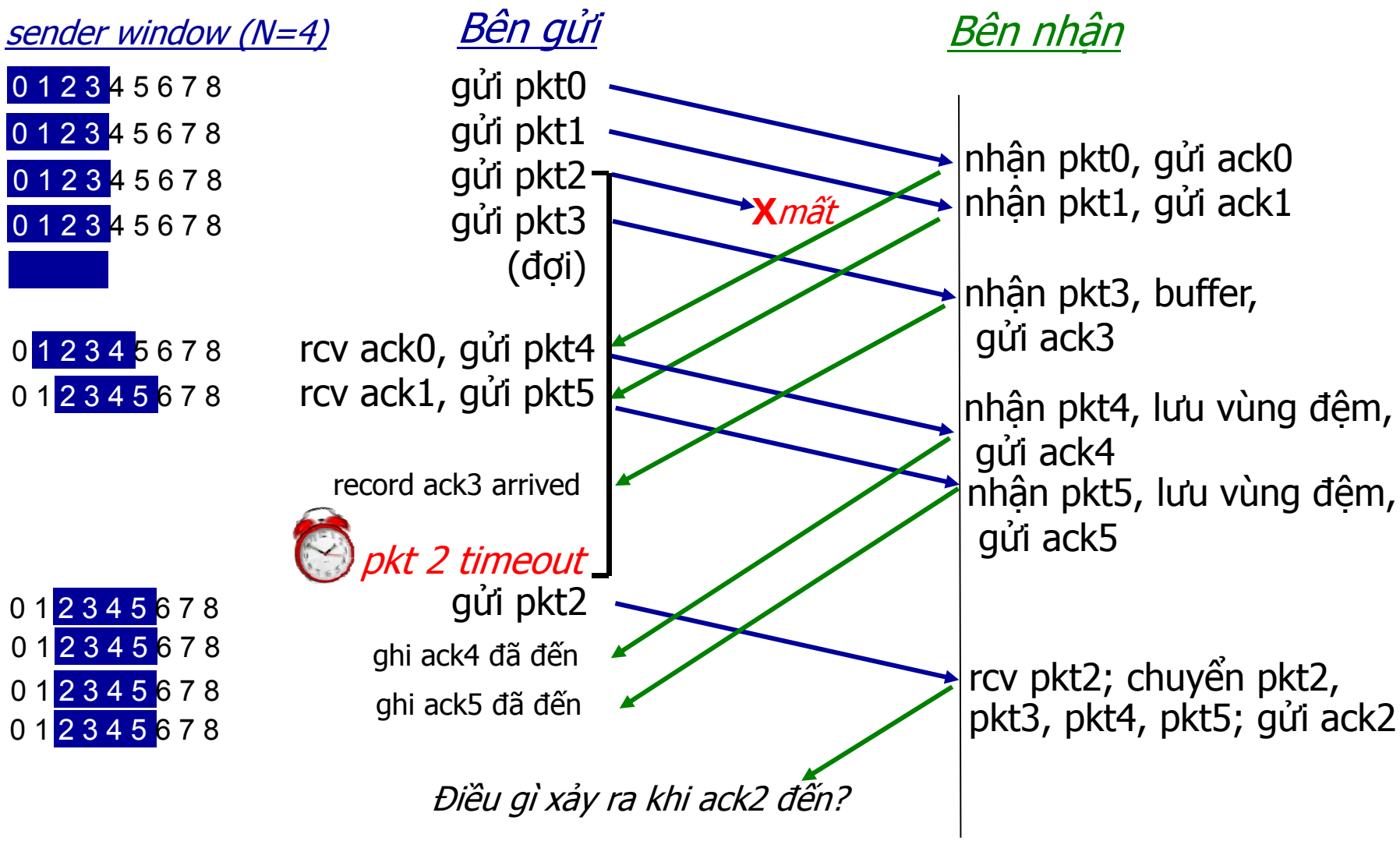
pkt n trong [rcvbase-N,rcvbase-1]

- ACK(n)

nếu không:

- bỏ qua

Selective repeat: Ví dụ



Chương 3: Tầng giao vận

- Dịch vụ của tầng giao vận
- Multiplexing và Demultiplexing
- Truyền không kết nối: UDP
- Nguyên tắc của truyền dữ liệu tin cậy
- Truyền hướng kết nối: TCP
- Nguyên tắc của điều khiển tắc nghẽn
- Điều khiển tắc nghẽn của TCP

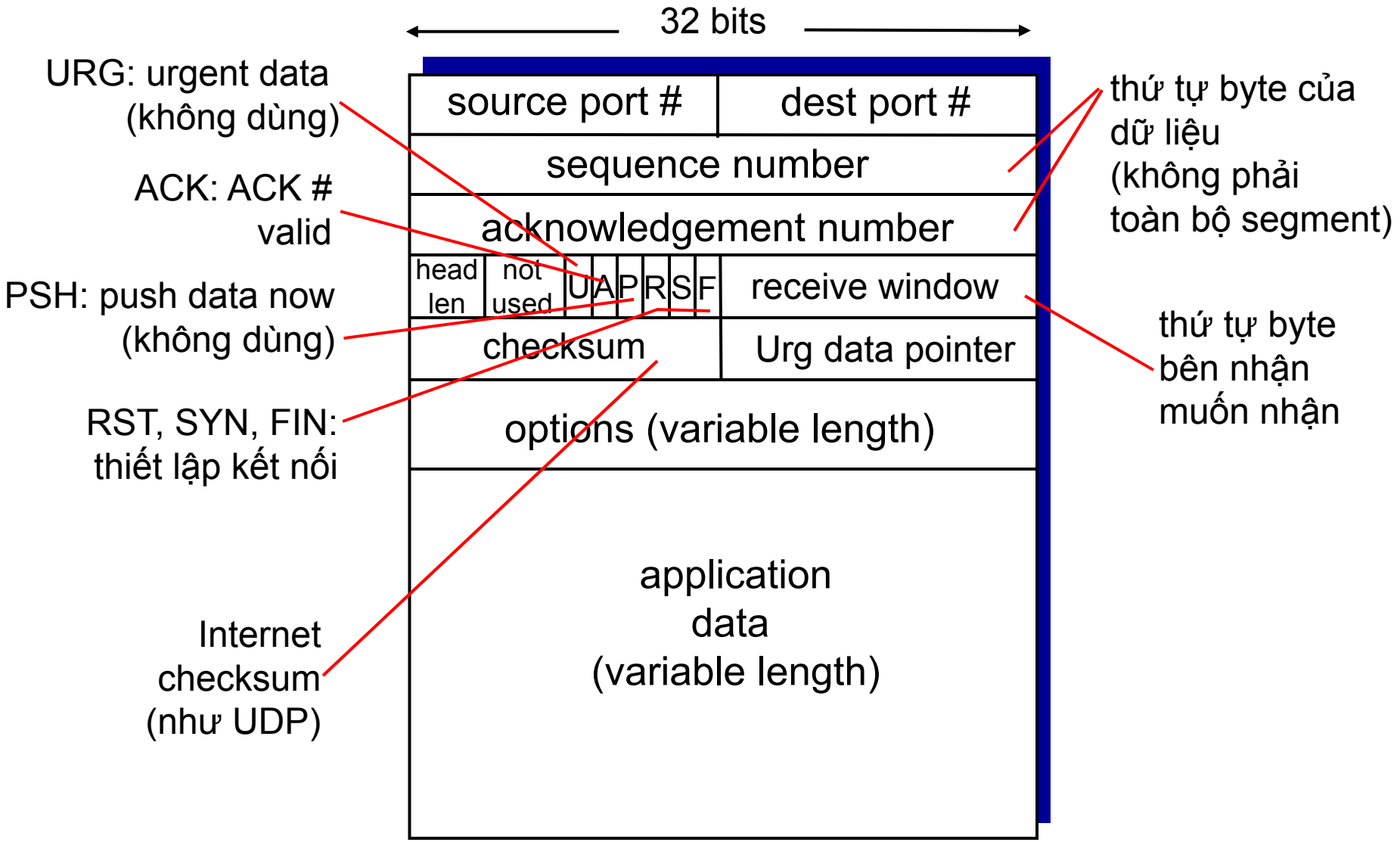
Connection-Oriented Transport: Cấu trúc của TCP Segment

Một số đặc điểm TCP

RFCs: 793,1122,1323, 2018, 2581

- điểm tới điểm (point-to-point):
  - một nút gửi, một nút nhận
- tin cậy, chuỗi byte đảm bảo thứ tự:
  - không phân tách bản tin
- xử lý liên tục (pipelined):
  - Điều khiển luồng và điều khiển tắc nghẽn của TCP gán window size
- dữ liệu hai chiều (full duplex):
  - luồng dữ liệu hai chiều trong cùng kết nối
  - MSS: maximum segment size
- hướng kết nối (connection-oriented):
  - handshaking (exchange of control msgs) inits sender, receiver state before data exchange
- điều khiển luồng:
  - bên gửi không làm quá tải bên nhận

Cấu trúc của TCP segment



Sequence number, ACK

sequence number:

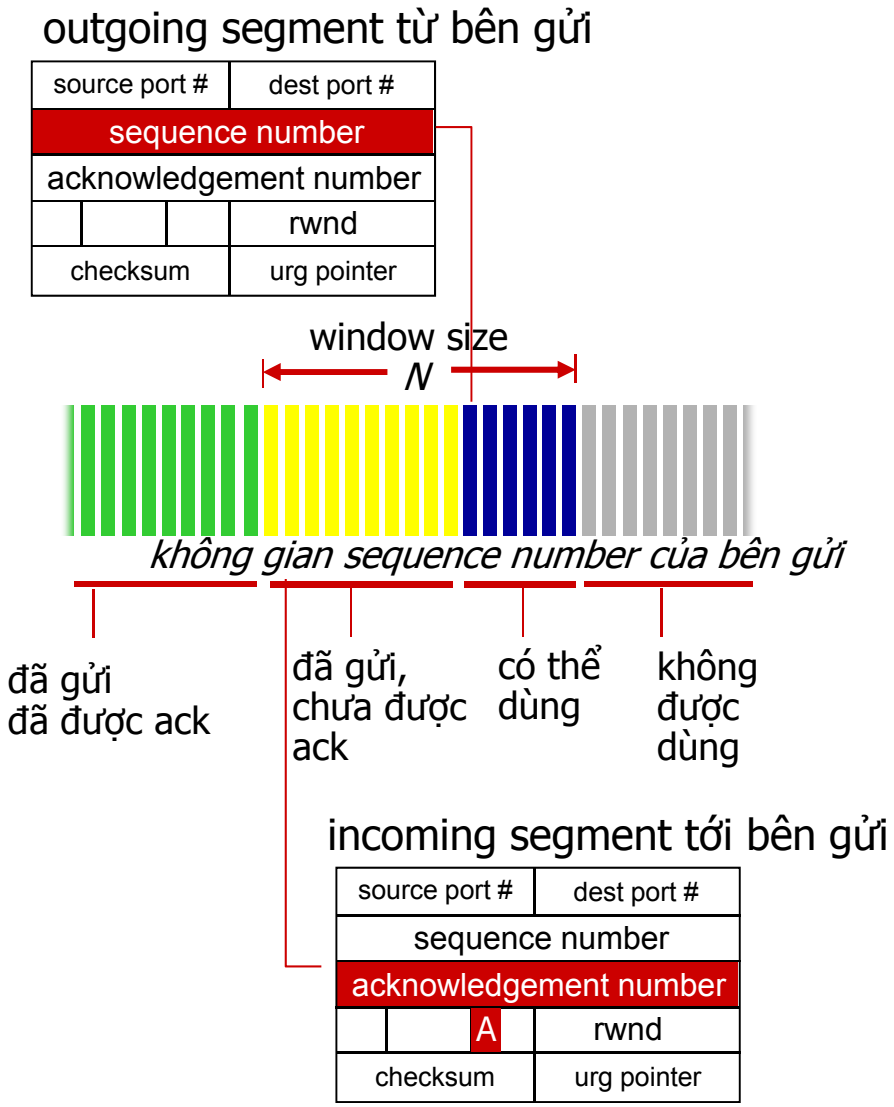
- thứ tự dòng byte của byte đầu tiên trong phần dữ liệu của segment

acknowledgement:

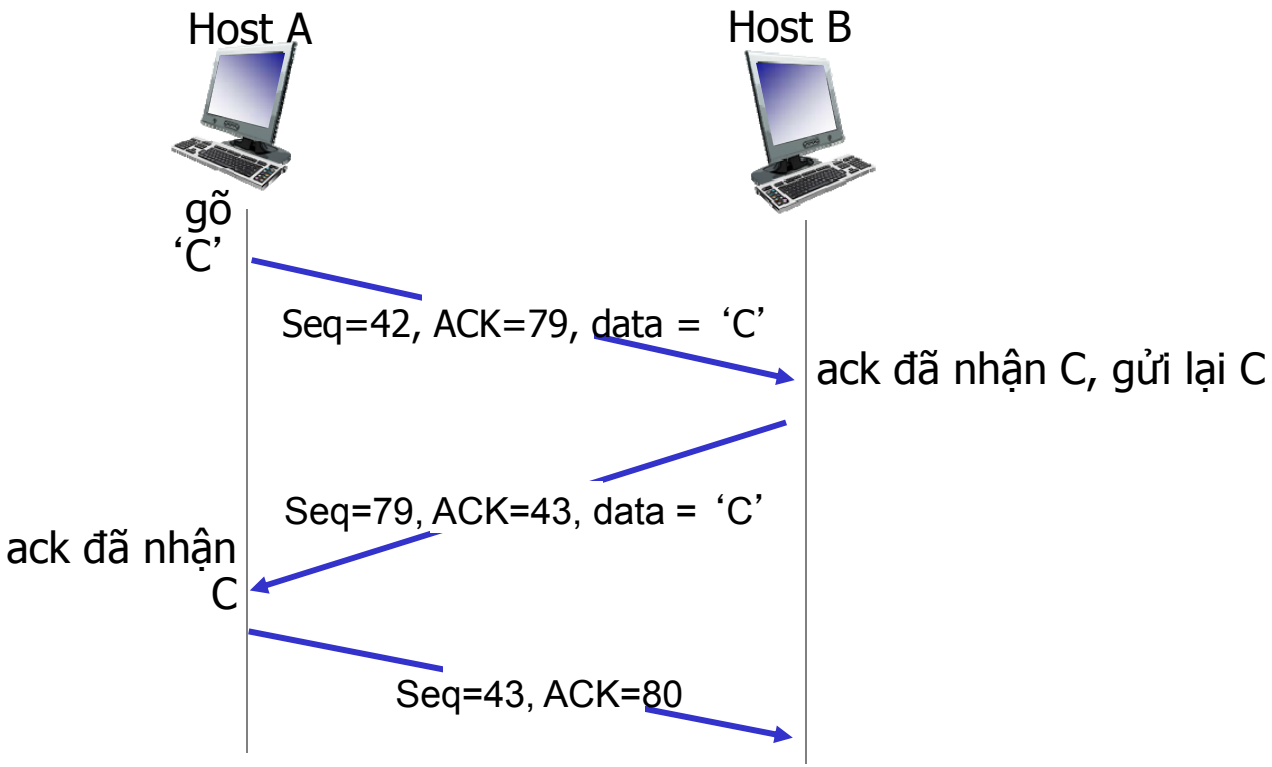
- seq # của byte tiếp theo muốn nhận từ phía bên kia
- cumulative ACK

Bên gửi xử lý segment không đúng thứ tự như thế nào?

- TCP không mô tả



Sequence number, ACK



kịch bản telnet

TCP round trip time, timeout

Cách xác định giá trị TCP timeout?

- lớn hơn RTT
  - nhưng RTT thay đổi
- *quá nhỏ:* timeout sớm, gây truyền lại không cần thiết
- *quá lớn:* chậm xử lý mất gói

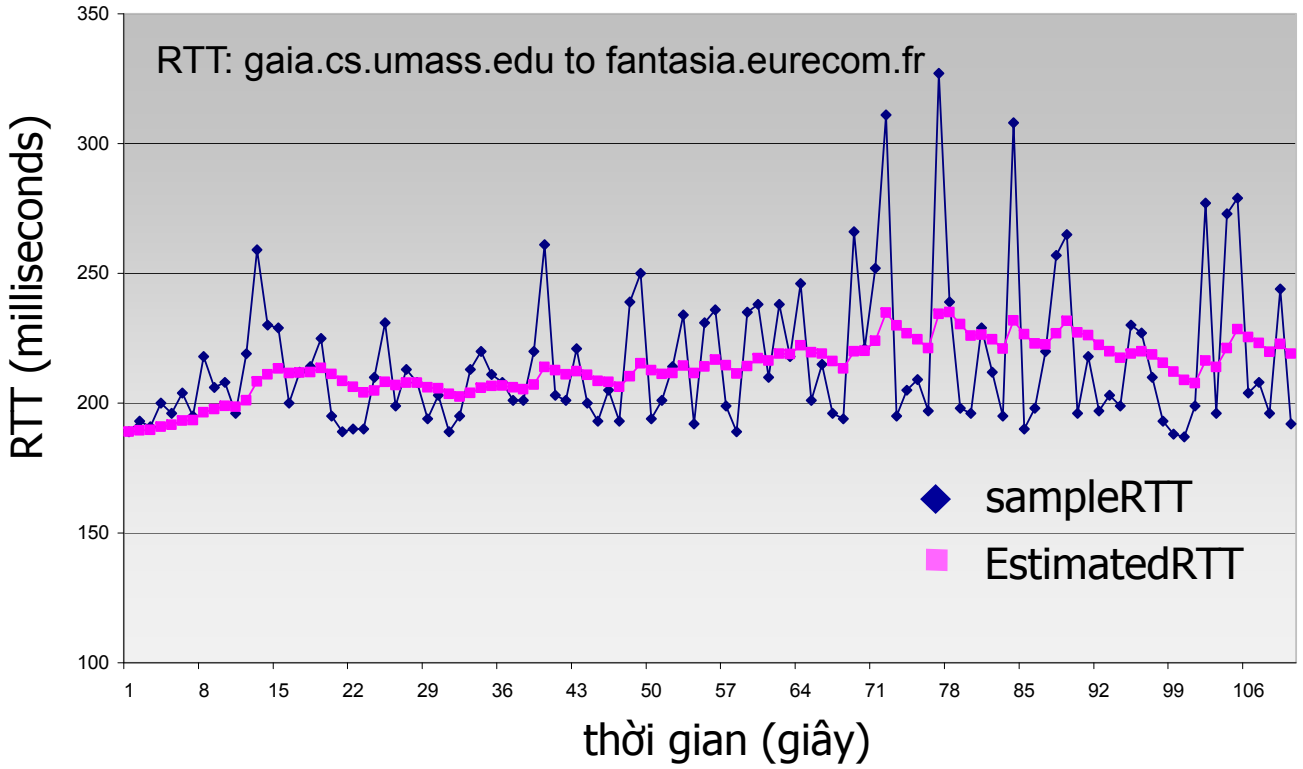
Ước tính RTT?

- **`SampleRTT`**: đo thời gian từ segment gửi tới nhận ACK
  - bỏ qua truyền lại
- **`SampleRTT`** sẽ thay đổi
  - trung bình của một số giá trị đo gần nhất, không chỉ giá trị hiện tại của **`SampleRTT`**

TCP round trip time, timeout

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

- thường chọn: α = 0.125



TCP round trip time, timeout

- timeout interval:

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

(typically, $\beta = 0.25$)

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

estimated RTT

"safety margin"

Connection-Oriented Transport: Truyền tin cậy

Truyền dữ liệu tin cậy của TCP

- TCP tạo dịch vụ rdt trên dịch vụ không tin cậy của IP
  - pipelined segment
  - cumulative ack
  - một đồng hồ truyền lại
- truyền lại khi:
  - có timeout
  - trùng lặp ack

Xem xét trường hợp đơn giản của nút gửi trong TCP:

- bỏ qua trùng lặp ack
- bỏ qua điều khiển luồng, điều khiển tắc nghẽn

Sự kiện của nút gửi:

Dữ liệu nhận được từ ứng dụng:

- tạo segment với seq #
- seq # là thứ tự của dòng byte của byte đầu tiên trong segment
- bật đồng hồ nếu chưa chạy
  - đồng hồ cho segment chưa được ack cũ nhất
  - khoảng thời gian quá hạn: **`TimeOutInterval`**

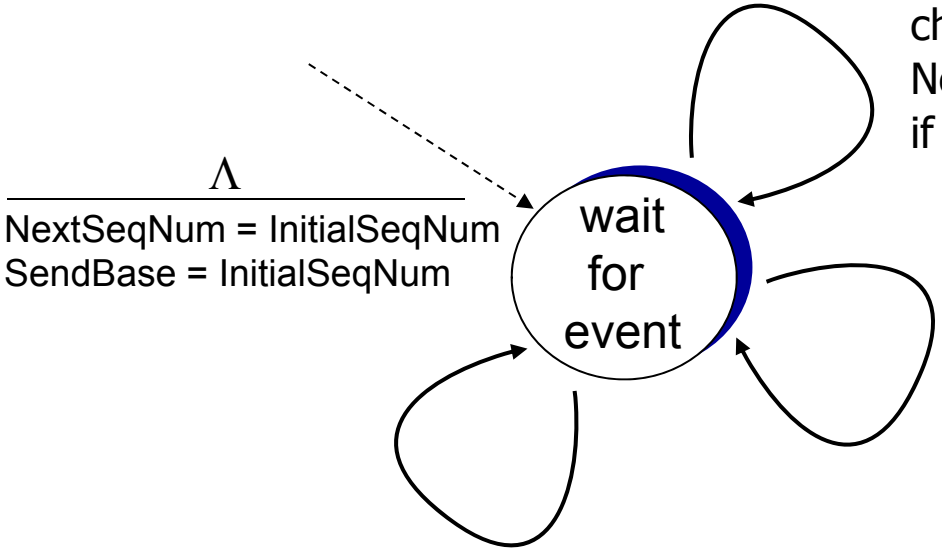
timeout:

- truyền lại segment đã gây ra timeout
- khởi tạo lại đồng hồ

nhận ack:

- nếu ack xác nhận segment chưa ack trước đó
  - cập nhật segment được ack
  - bật đồng hồ nếu vẫn còn gói tin chưa ack

TCP sender (đơn giản)



Λ

NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

dữ liệu nhận từ ứng dụng

tạo segment, seq. #: NextSeqNum
chuyển segment tới IP (để gửi)
NextSeqNum = NextSeqNum + length(data)
if (timer chưa chạy)
  start timer

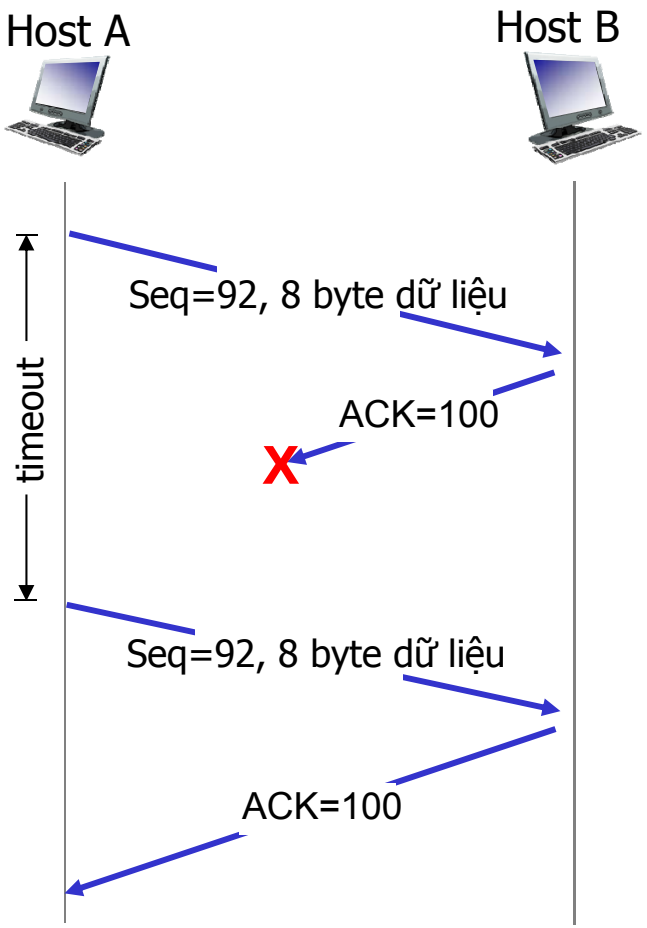
timeout

truyền lại segment chưa được ack
với seq. # nhỏ nhất
bật timer

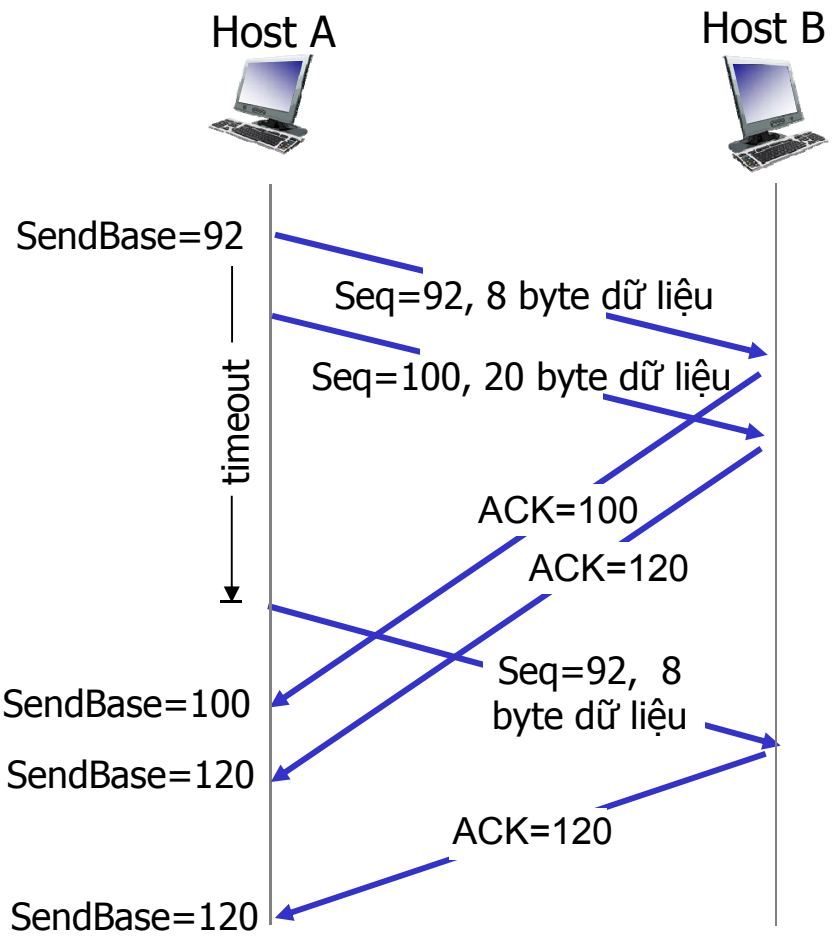
nhận ACK, giá trị trường ACK là y

```
if (y > SendBase) {
   SendBase = y
   /* SendBase–1: last cumulatively ACKed byte */
   if (có segment chưa được ack)
       bật timer
     else dừng timer
    }
```

TCP: Kịch bản truyền lại

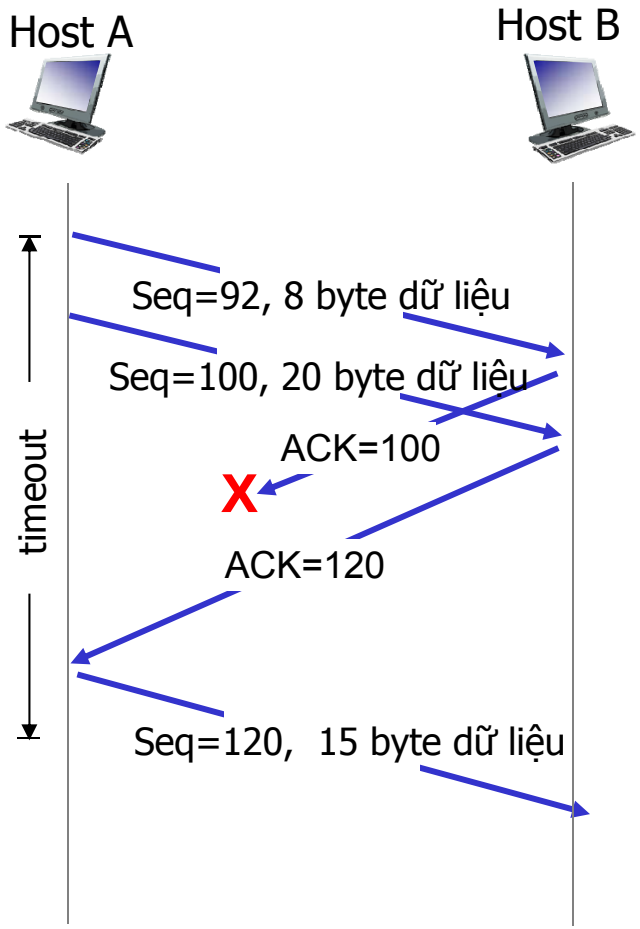


kịch bản mất ACK



timeout sớm

TCP: Kịch bản truyền lại



cumulative ACK

TCP ACK generation [RFC 1122, RFC 2581]

Sự kiện tại bên nhận	*Xử lý của bên nhận*
Nhận segment đúng thứ tự với seq # đang chờ. Toàn bộ dữ liệu cho tới seq # đang chờ được ack	ACK trễ. Đợi 500ms cho segment tiếp. Nếu không có segment tiếp, gửi ACK
Nhận segment đúng thứ tự với seq # đang chờ. Một segment khác chờ ACK	Gửi một cumulative ACK, cho hai segment đảm bảo thứ tự
Nhận segment không đúng thứ tư có seq. # lớn hơn seq # đang chờ. Phát hiện khoảng trống.	Gửi *duplicate ACK*, chỉ seq. # của byte đang chờ
Nhận segment trong khoảng trống	Gửi ACK, send ACK, nếu segment bắt đầu tại phần cuối của khoảng trống

TCP fast retransmit

- Khoảng thời gian time-out thường lớn:
  - độ trễ dài trước khi gửi gói tin bị mất
- Phát hiện segment mất dựa vào duplicate ACK

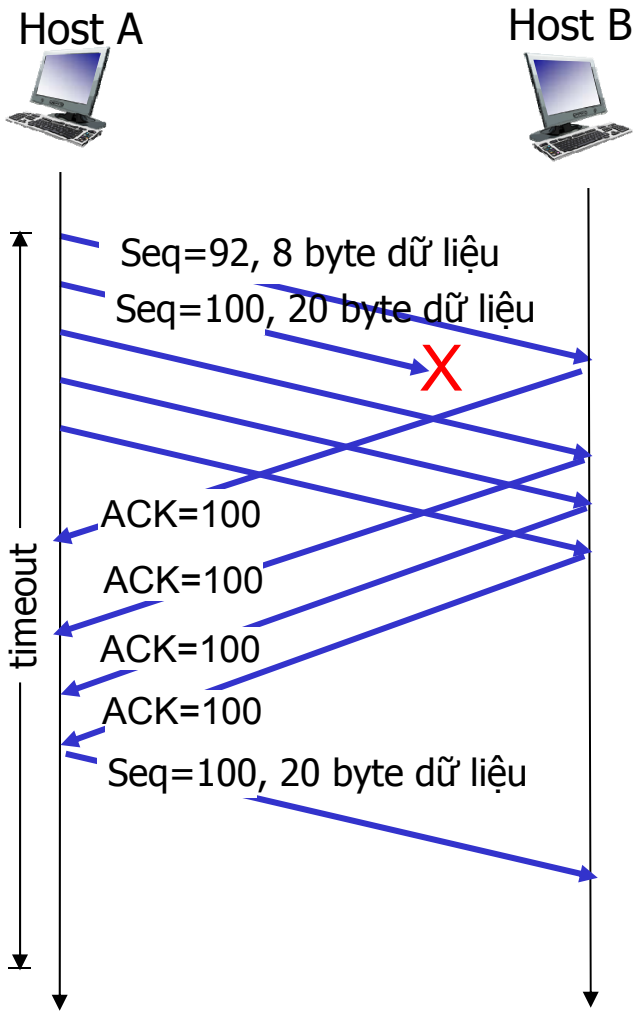
TCP fast retransmit

Nếu nút gửi nhận 3 ACK cho cùng dữ liệu

("triple duplicate ACKs"), gửi lại segment chưa được ack với seq # nhỏ nhất

- có khả năng segment chưa ack bị mất, vì vậy không đợi đến khi timeout

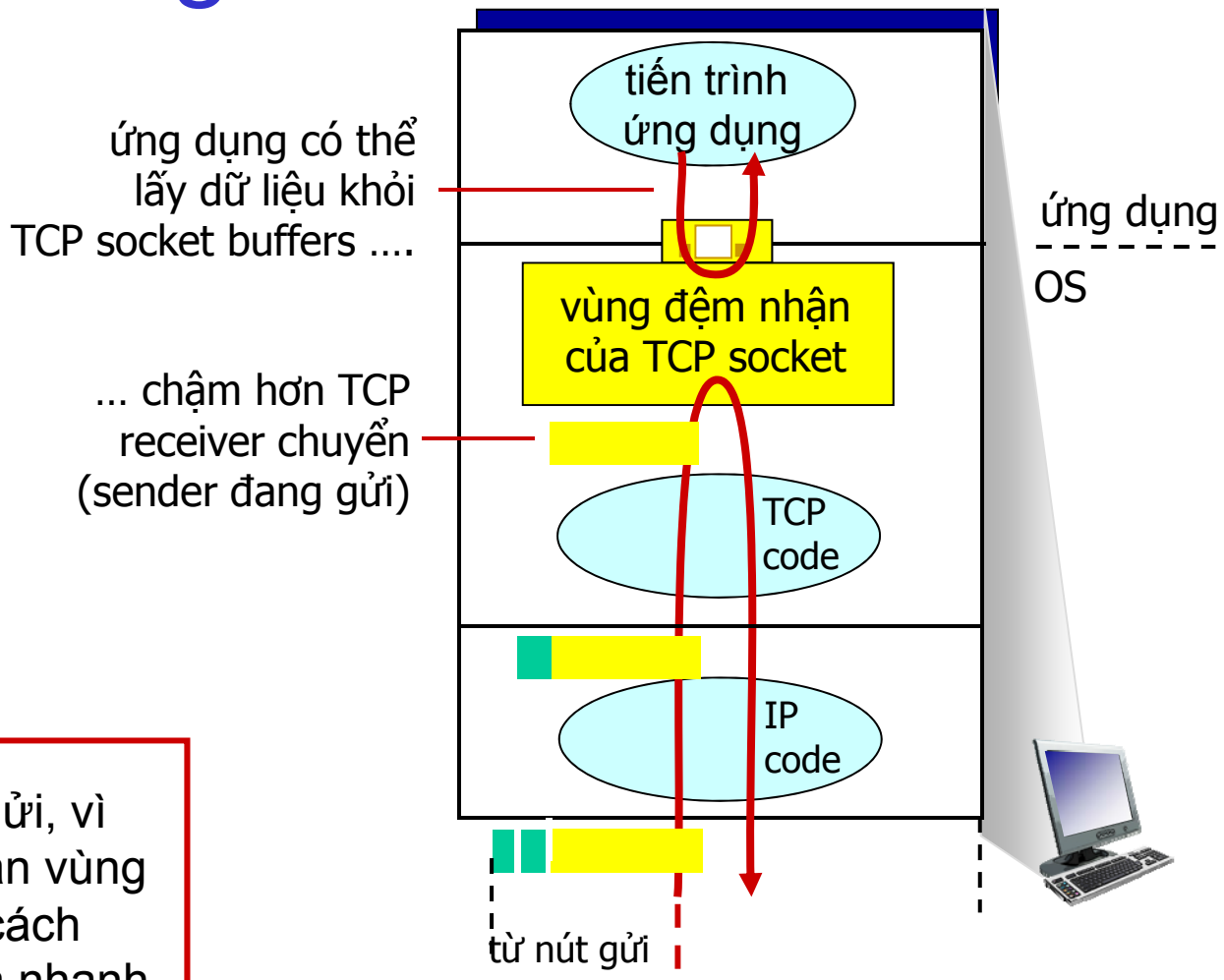
TCP fast retransmit



fast retransmit sau khi nút gửi nhận triple-duplicate-ACK

Connection-Oriented Transport: Điều khiển luồng của TCP

Điều khiển luồng của TCP



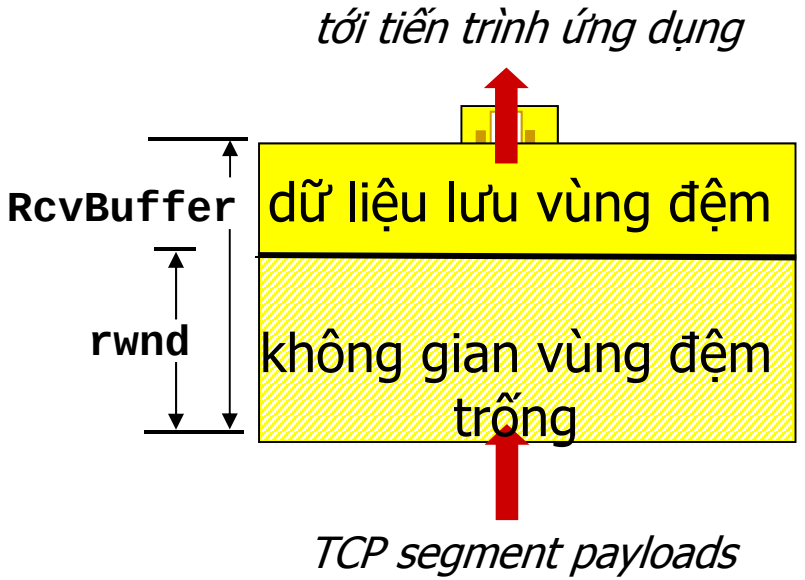
ngăn xếp giao thức tại nút nhận

điều khiển luồng

nút nhận điều khiển nút gửi, vì vậy nút gửi không làm tràn vùng đệm của nút nhận bằng cách truyền quá nhiều hay quá nhanh

Điều khiển luồng của TCP

- Nút nhận thông báo không gian vùng đệm còn trống bằng cách đưa giá trị **`rwnd`** trong TCP header của segment gửi từ nút nhận tới nút gửi
  - **`RcvBuffer`** size gán qua tùy chọn socket (thường mặc định 4096 byte)
  - nhiều hệ điều hành tự động điều chỉnh **`RcvBuffer`**
- Nút gửi giới hạn dữ liệu chưa được ack bằng giá trị **`rwnd`** của nút nhận
- Đảm bảo vùng đệm nhận không bị tràn



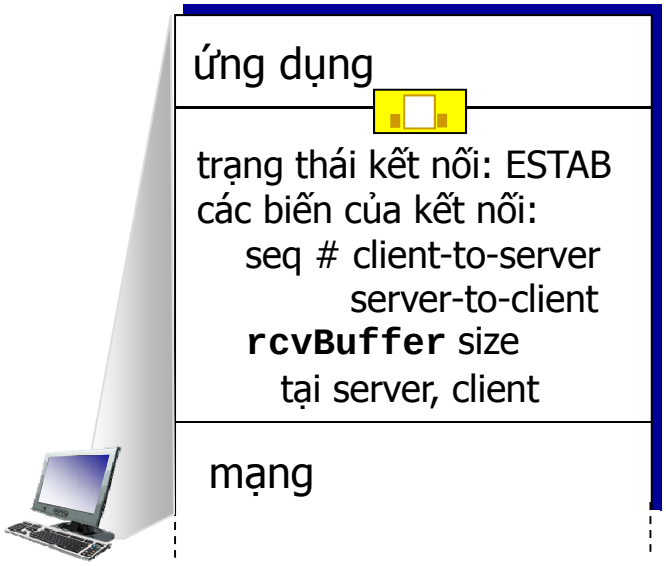
lưu vùng đệm phía nút nhận

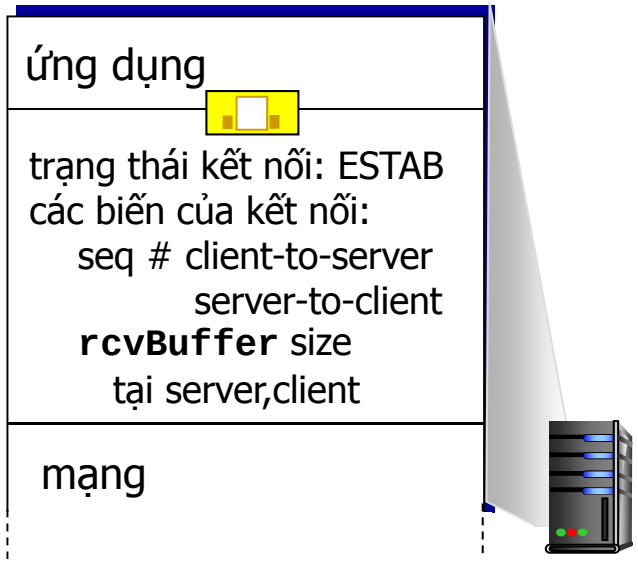
Connection-Oriented Transport: Quản lý kết nối của TCP

Quản lý kết nối

Trước khi trao đổi dữ liệu, nút gửi và nhận thực hiện bắt tay (handshake):

- chấp nhận thiết lập kết nối (mỗi nút biết nút kia muốn thiết lập kết nối)
- chấp nhận tham số kết nối



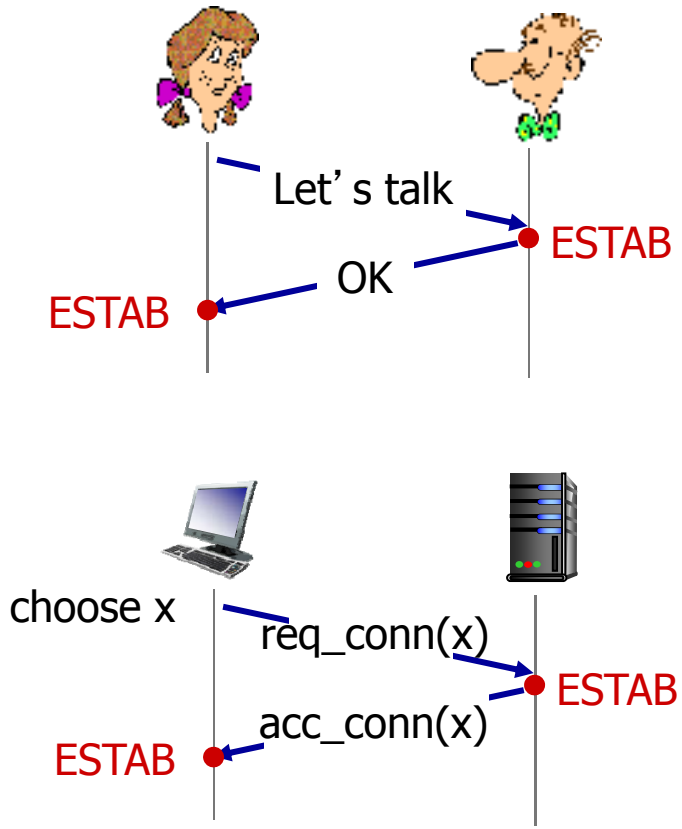



```
Socket clientSocket =
  newSocket("hostname","port
   number");
```

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

Chấp nhận thiết lập kết nối

2-way handshake:

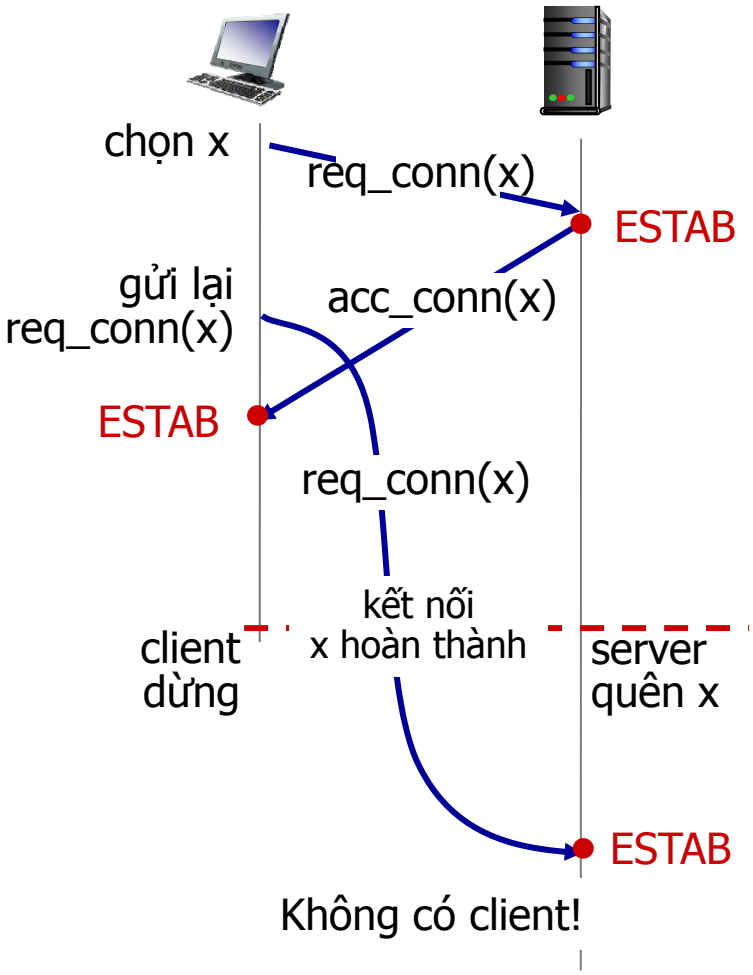


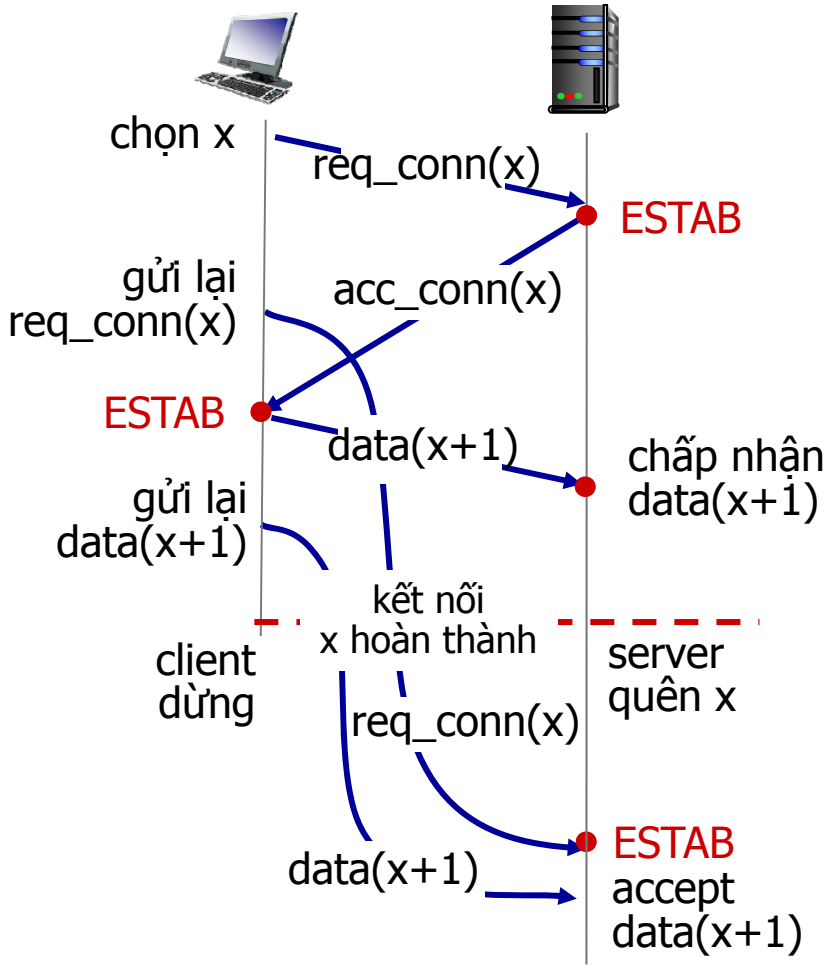
2-way handshake có dùng được trong mạng không?

- độ trễ thay đổi
- bản tin truyền lại (ví dụ req_conn(x)) do mất gói tin
- thứ tự bản tin
- không thấy phía kia

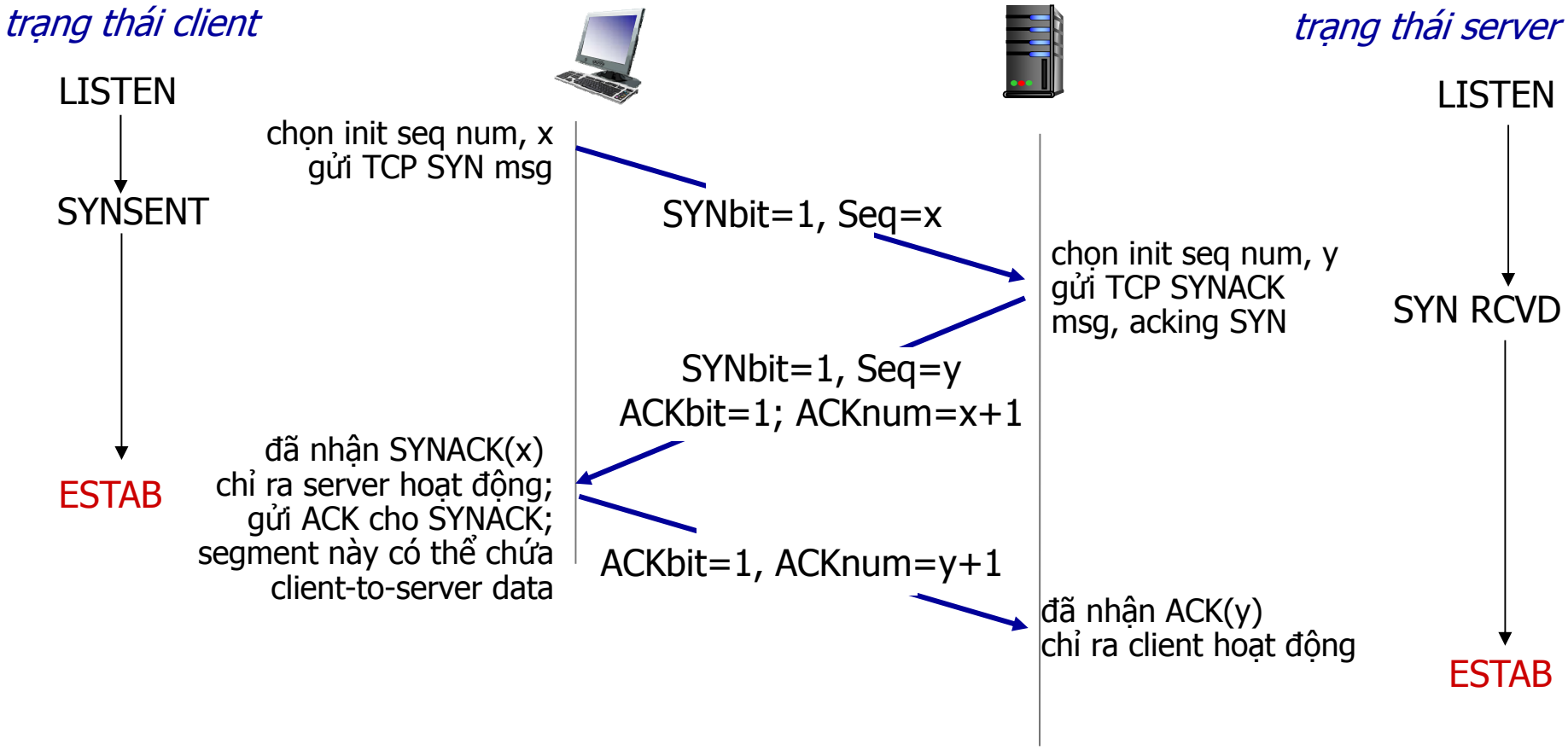
Chấp nhận thiết lập kết nối

Ví dụ lỗi trong 2-way handshake:

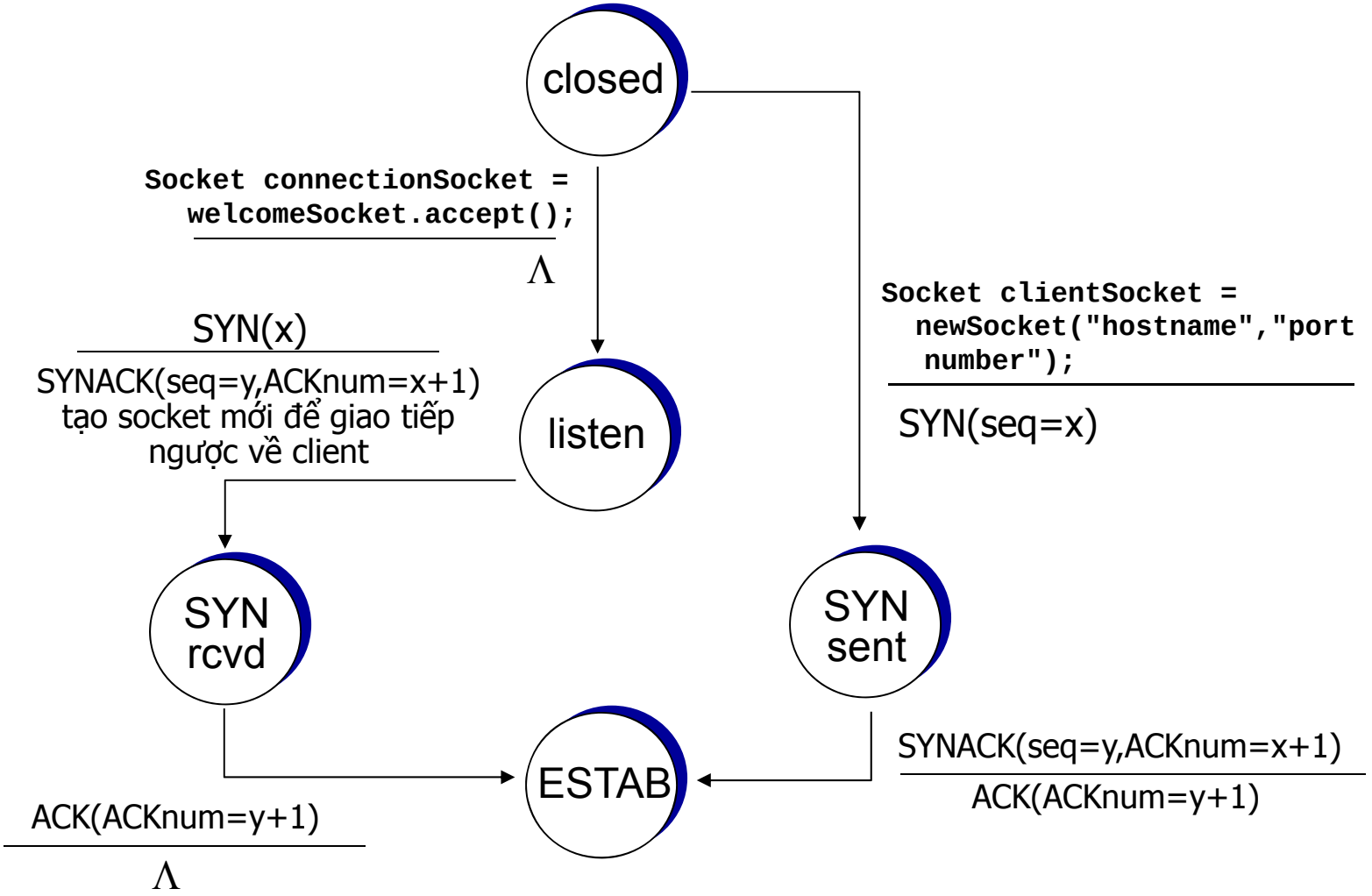




TCP 3-way handshake



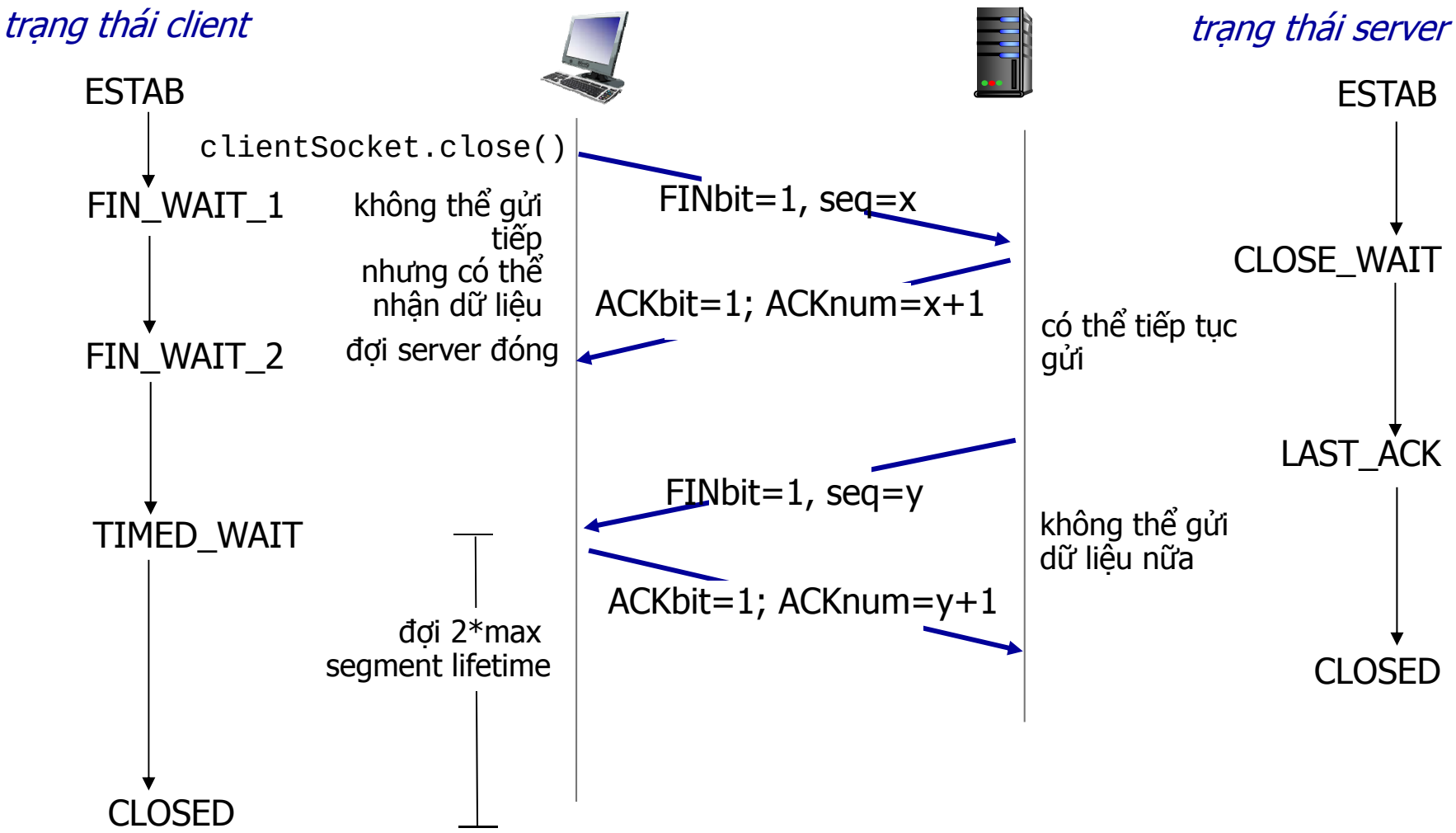
TCP 3-way handshake: FSM



Chấm dứt kết nối

- Client, server đóng kết nối ở phía mình
  - gửi TCP segment có FIN bit = 1
- Trả lời FIN đã nhận bằng ACK
  - khi nhận FIN, ACK có thể kèm với FIN của nó
- Có thể xử lý trao đổi FIN đồng thời

Chấm dứt kết nối



Chương 3: Tầng giao vận

- Dịch vụ của tầng giao vận
- Multiplexing và Demultiplexing
- Truyền không kết nối: UDP
- Nguyên tắc của truyền dữ liệu tin cậy
- Truyền hướng kết nối: TCP
- Nguyên tắc của điều khiển tắc nghẽn
- Điều khiển tắc nghẽn của TCP

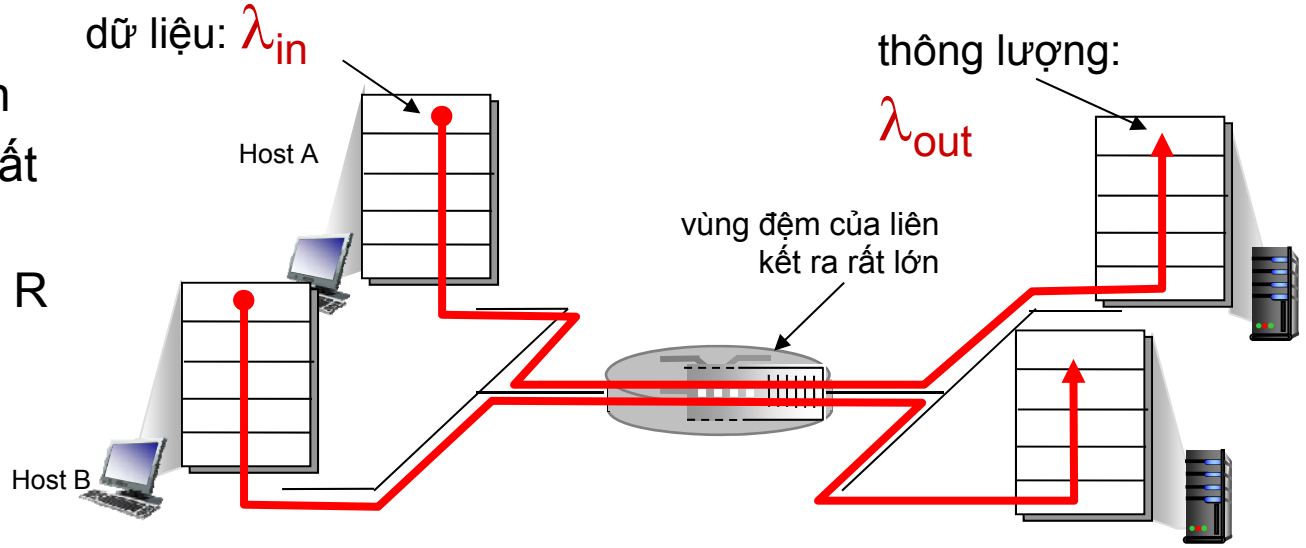
Nguyên tắc của điều khiển tắc nghẽn

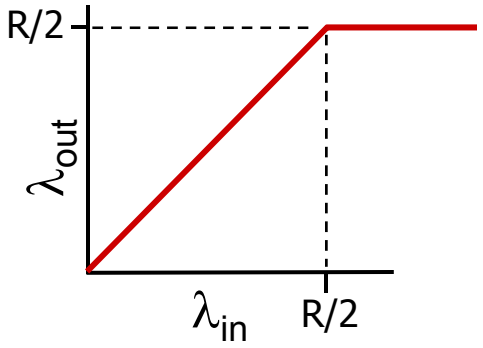
Tắc nghẽn:

- Quá nhiều nguồn gửi quá nhiều dữ liệu và quá nhanh vào mạng
- Khác với điều khiển luồng!
- Thể hiện qua:
  - mất gói tin (tràn vùng đệm tại router)
  - độ trễ lớn (đợi trong vùng đệm của router)
- Một trong 10 vấn đề của mạng!

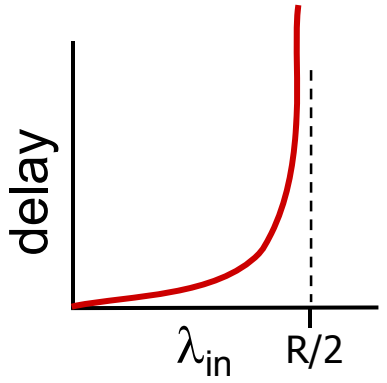
Hậu quả của tắc nghẽn: Kịch bản 1

- 2 nút gửi, 2 nút nhận
- 1 router, vùng đệm rất lớn
- khả năng liên kết ra: R
- không truyền lại





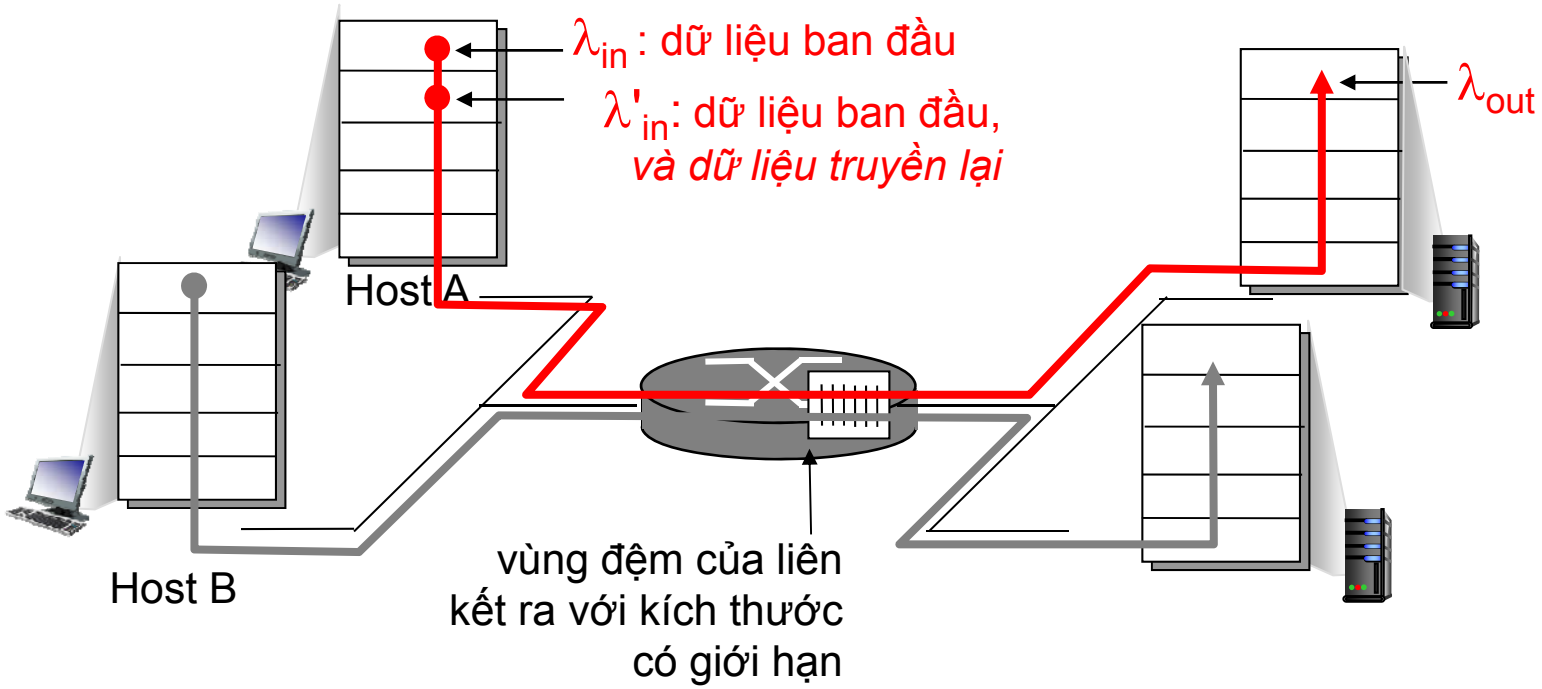
- maximum per-connection throughput: R/2



- delays lớn theo arrival rate, λ_{in}, tiến dần tới capacity

Hậu quả của tắc nghẽn: Kịch bản 2

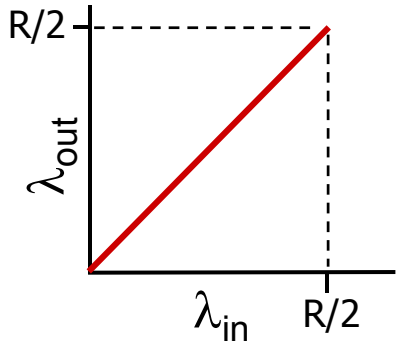
- 1 router, vùng đệm có giới hạn
- Truyền lại timed-out packet của nút gửi
  - application-layer input = application-layer output: $\lambda_{in} = \lambda_{out}$
  - transport-layer input tính cả *truyền lại*: $\lambda_{in} \geq \lambda'_{in}$

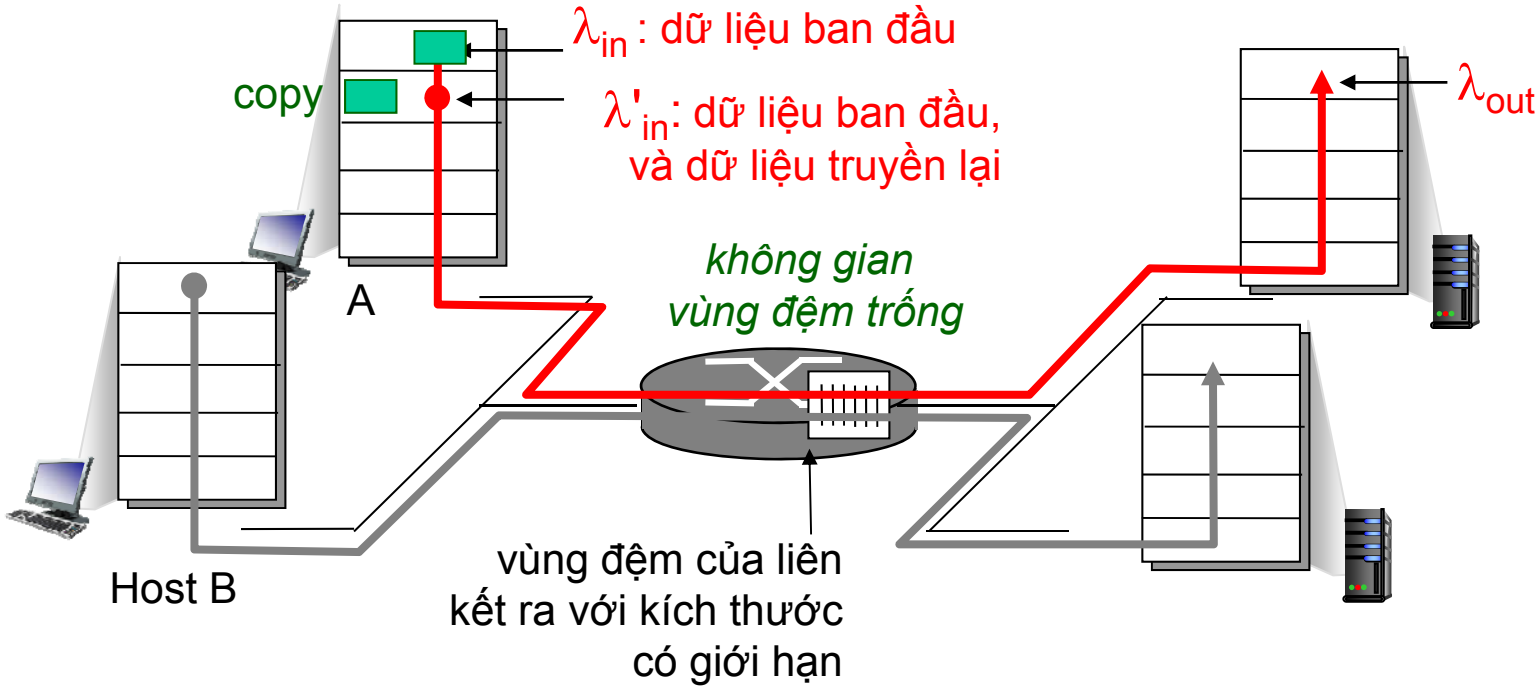


Hậu quả của tắc nghẽn: Kịch bản 2

Lý tưởng: Có đủ thông tin

- nút gửi chỉ gửi khi vùng đệm của router còn trống



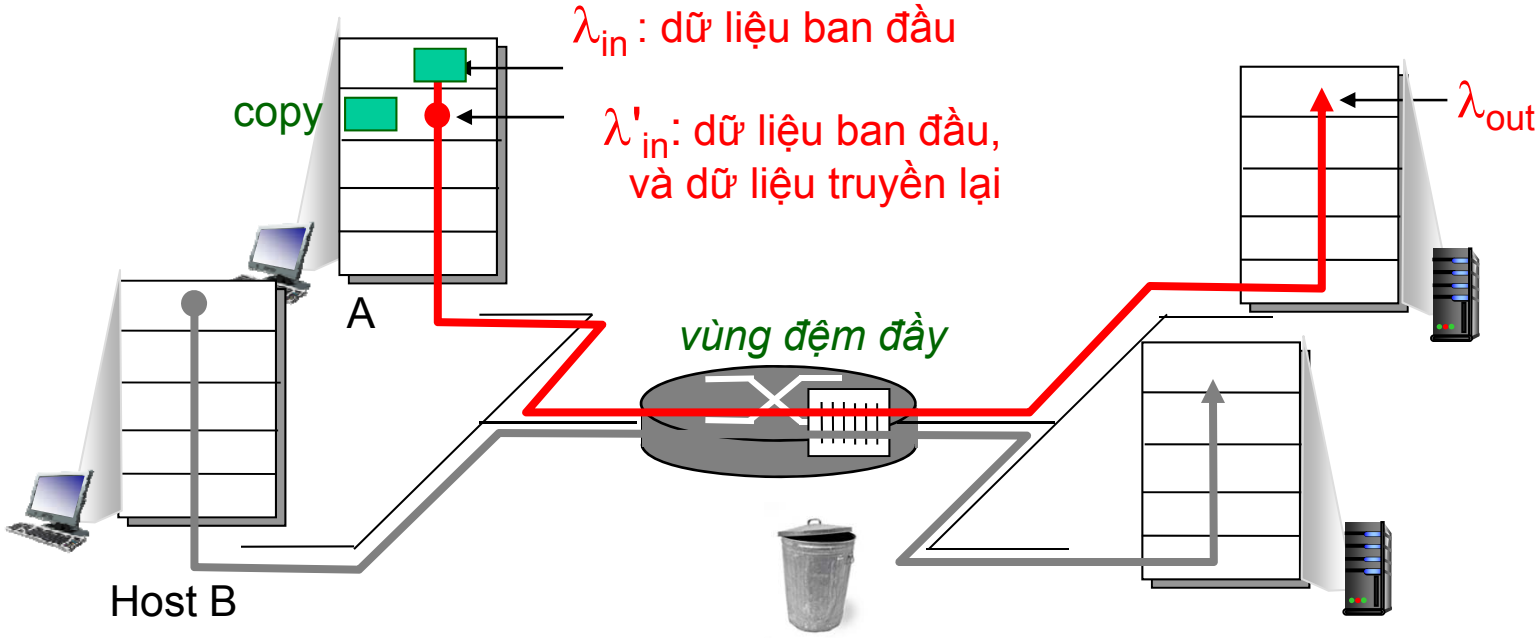


Hậu quả của tắc nghẽn: Kịch bản 2

Lý tưởng: biết mất gói

gói tin có thể mất, bị bỏ tại router do tràn vùng đệm

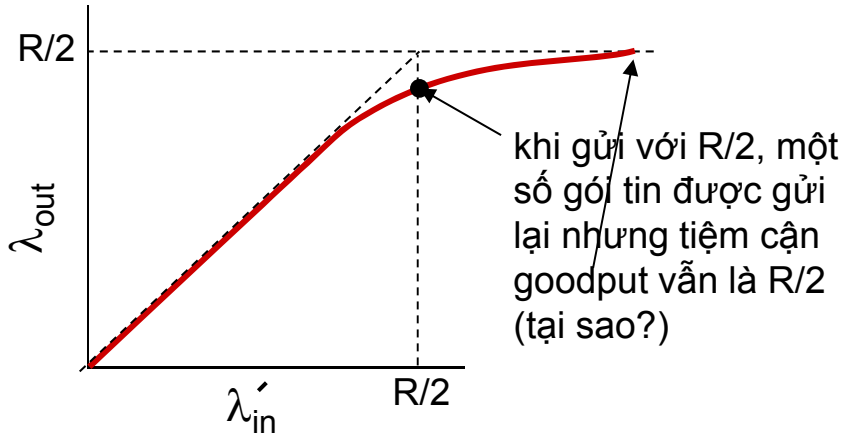
- nút gửi chỉ gửi lại nếu gói tin biết là bị mất

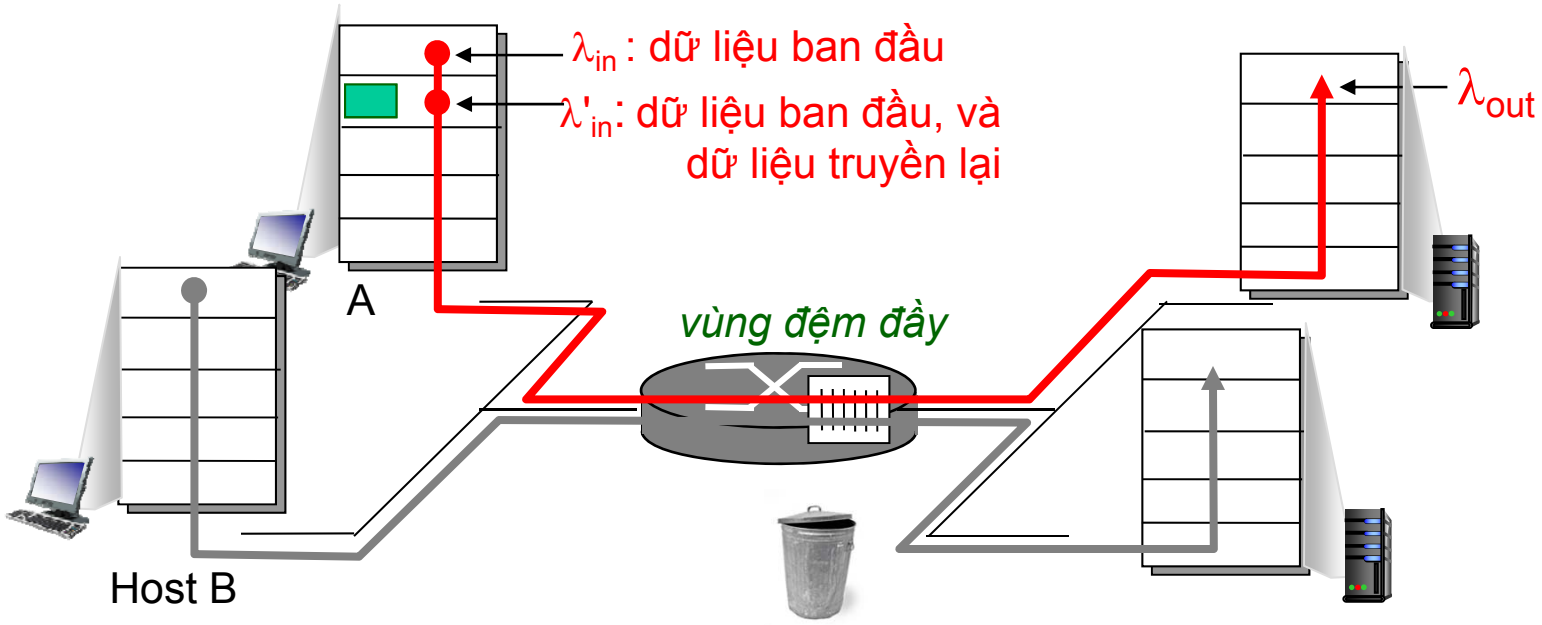


Hậu quả của tắc nghẽn: Kịch bản 2

Lý tưởng: biết mất gói gói tin có thể mất, bị bỏ tại router do tràn vùng đệm

- nút gửi chỉ gửi lại nếu gói tin biết là bị mất

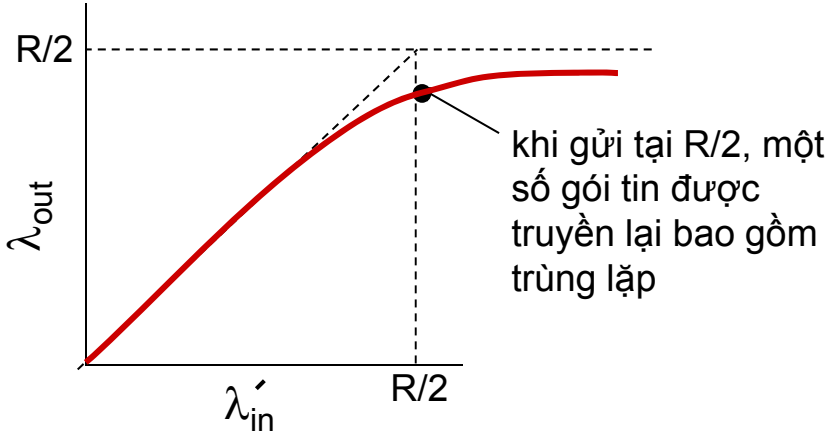


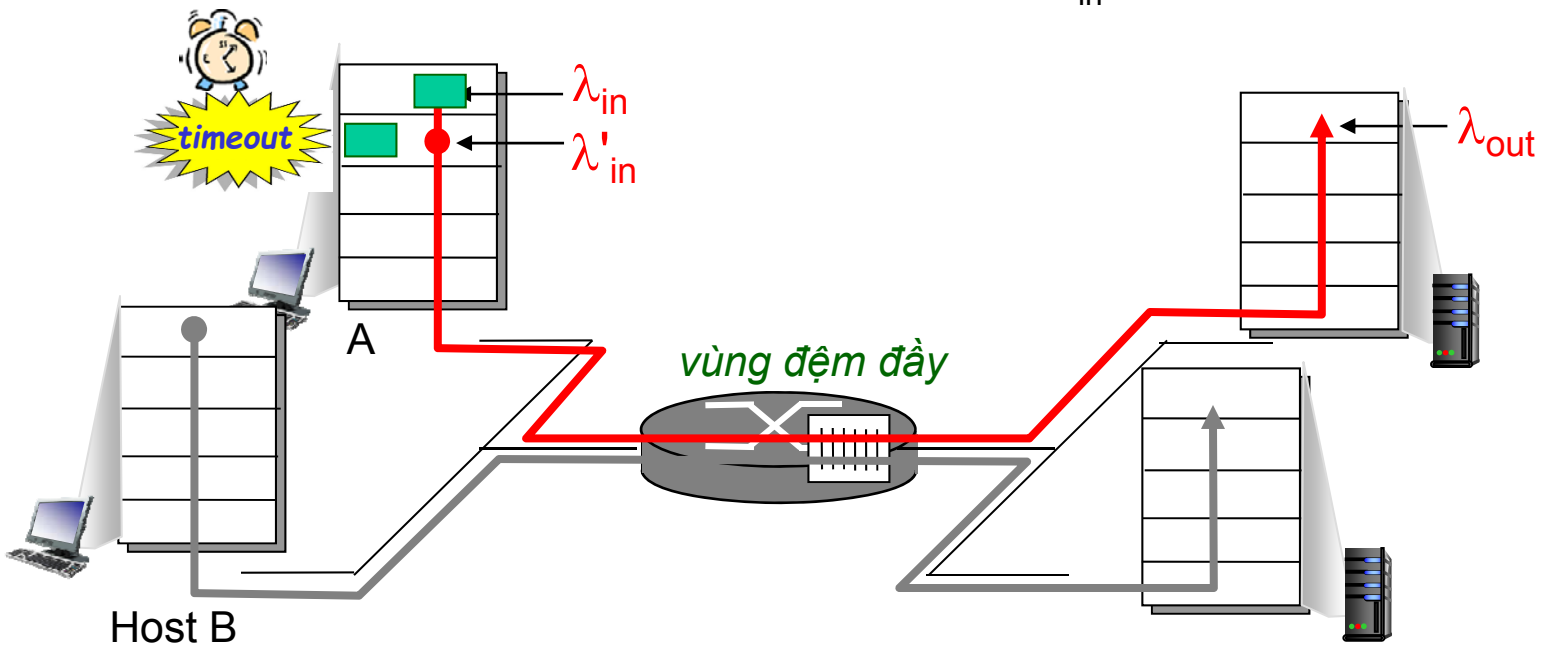


Hậu quả của tắc nghẽn: Kịch bản 2

Thực tế: trùng lặp

- gói tin có thể bị mất, hủy bỏ tại router do vùng đệm đầy
- nút gửi có timeout sớm, gửi 2 bản sao

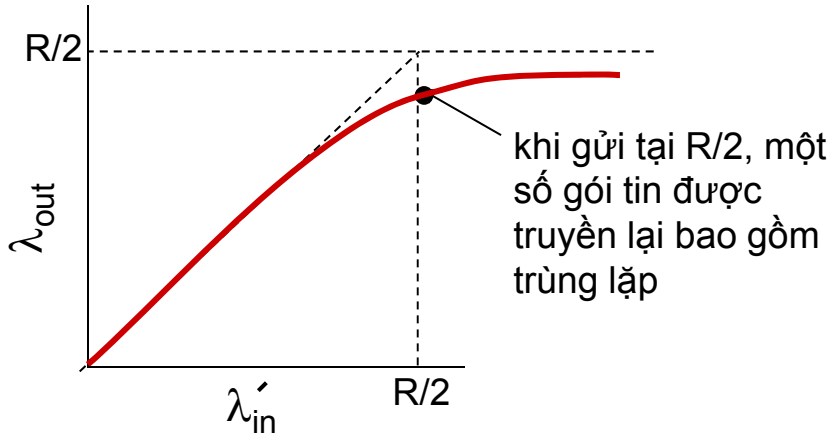




Hậu quả của tắc nghẽn: Kịch bản 2

Thực tế: trùng lặp

- gói tin có thể bị mất, hủy bỏ tại router do vùng đệm đầy
- nút gửi có timeout sớm, gửi 2 bản sao



Chi phí của tắc nghẽn:

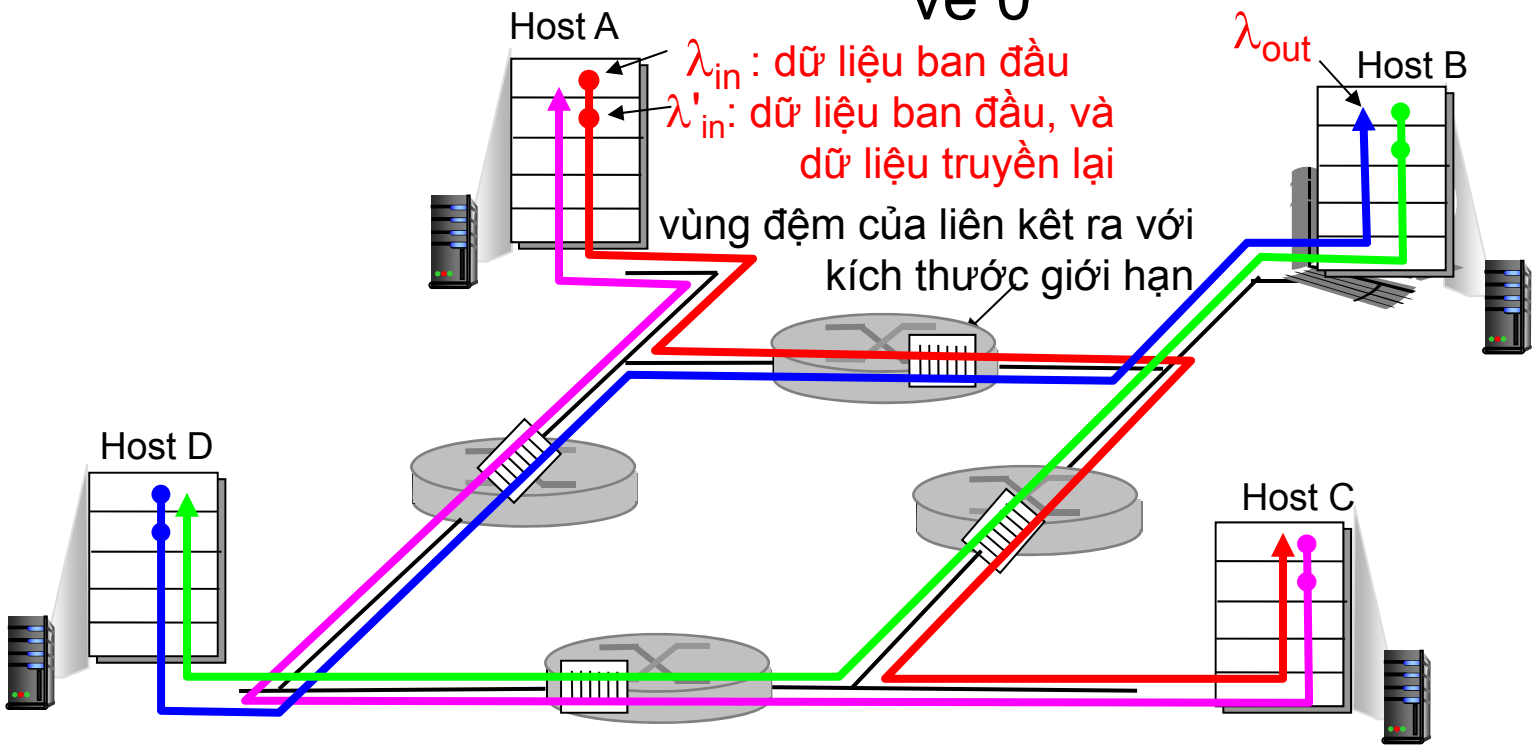
- nhiều việc (gửi lại) cho một mức "goodput"
- truyền lại không cần thiết: liên kết có nhiều bản sao của cùng một gói tin
  - làm giảm goodput

Hậu quả của tắc nghẽn: Kịch bản 3

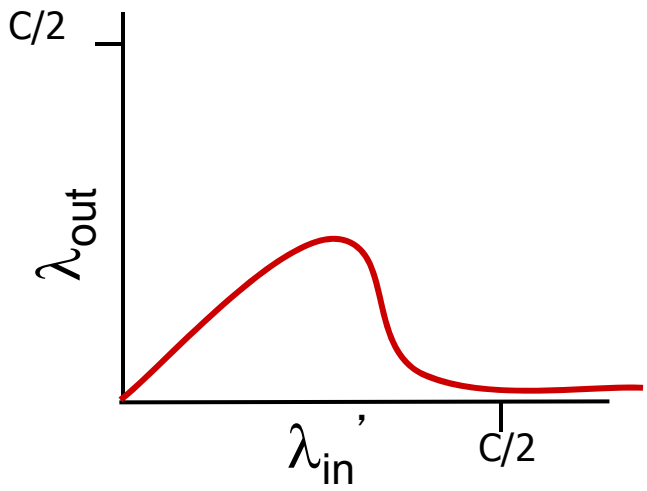
- 4 nút gửi
- đường đi qua nhiều nút
- timeout/truyền lại

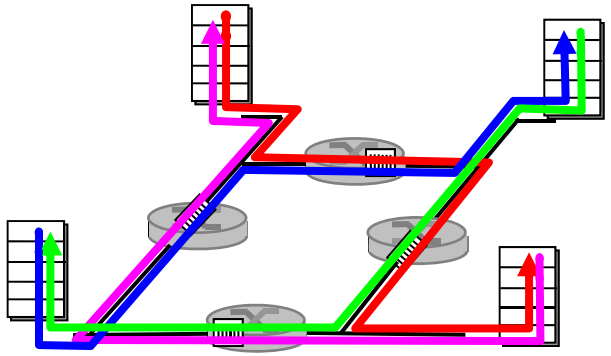
Vấn đề gì xảy ra khi λ_{in} và λ_{in}' tăng ?

Khi λ_{in}' (đỏ) tang, mọi gói tin (xanh) đến tầng trên bị hủy bỏ, thông lượng (xanh) giảm về 0



Hậu quả của tắc nghẽn: Kịch bản 3





Chi phí khác của tắc nghẽn:

- khi gói tin bị hủy bỏ, bất kì upstream transmission capacity dùng cho gói tin là lãng phí

Giải pháp điều khiển tắc nghẽn

Hai cách tiếp cận

Điều khiển tắc nghẽn cuối-cuối (end-end congestion control):

- không có phản hồi trực tiếp mạch từ mạng
- thông tin tắc nghẽn được suy ra từ end-system khi quan sát độ trễ, mất gói
- TCP sử dụng

Điều khiển tắc nghẽn có trợ giúp của mạng (network-assisted congestion control):

- router cung cấp phản hồi cho end-system
  - 1 bít chỉ ra tắc nghẽn (SNA, DECbit, TCP/IP ECN, ATM)

Điều khiển tắc nghẽn ATM ABR

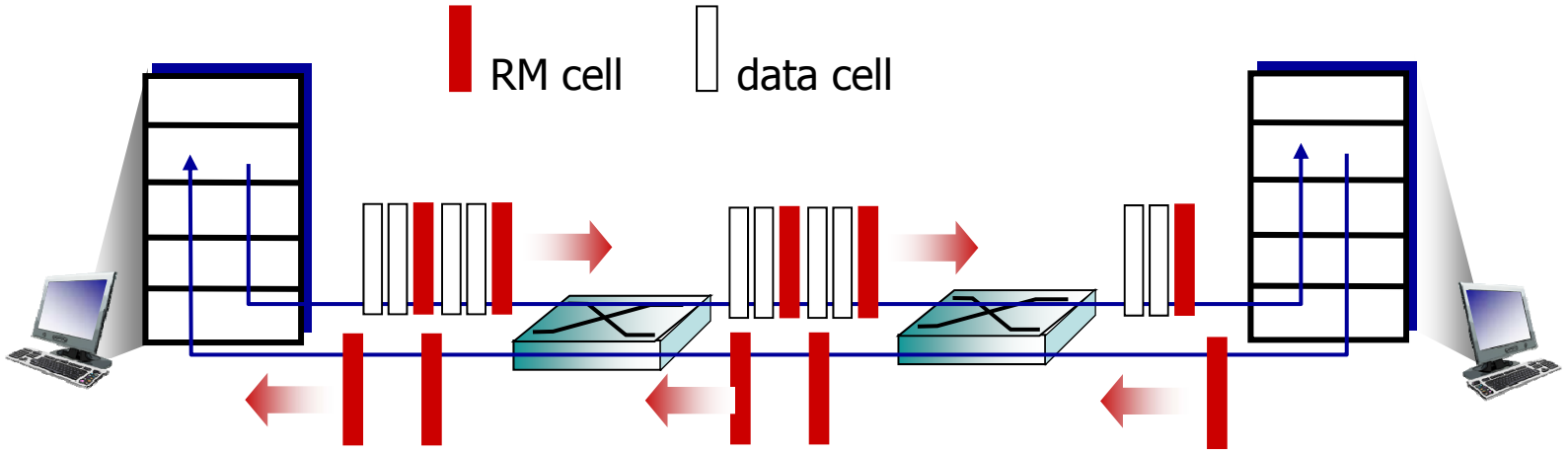
ABR: available bit rate:

- "elastic service"
- nếu đường đi của nút gửi là dưới tải:
  - nút gửi sử dụng bang thông còn
- nếu đường đi của nút gửi bị tắc nghẽn:
  - nút gửi sender điều khiển minimum guaranteed rate

RM (resource management) cells:

- gửi bởi nút gửi, rải rác cùng với data cell
- bít trong RM cell gán bởi switch ("*network-assisted*")
  - *NI bit:* không tăng tốc độ (tắc nghẽn nhẹ)
  - *CI bit:* dấu hiệu tắc nghẽn
- RM cell do nút nhận trả về nút gửi có bít giữ nguyên

Điều khiển tắc nghẽn ATM ABR



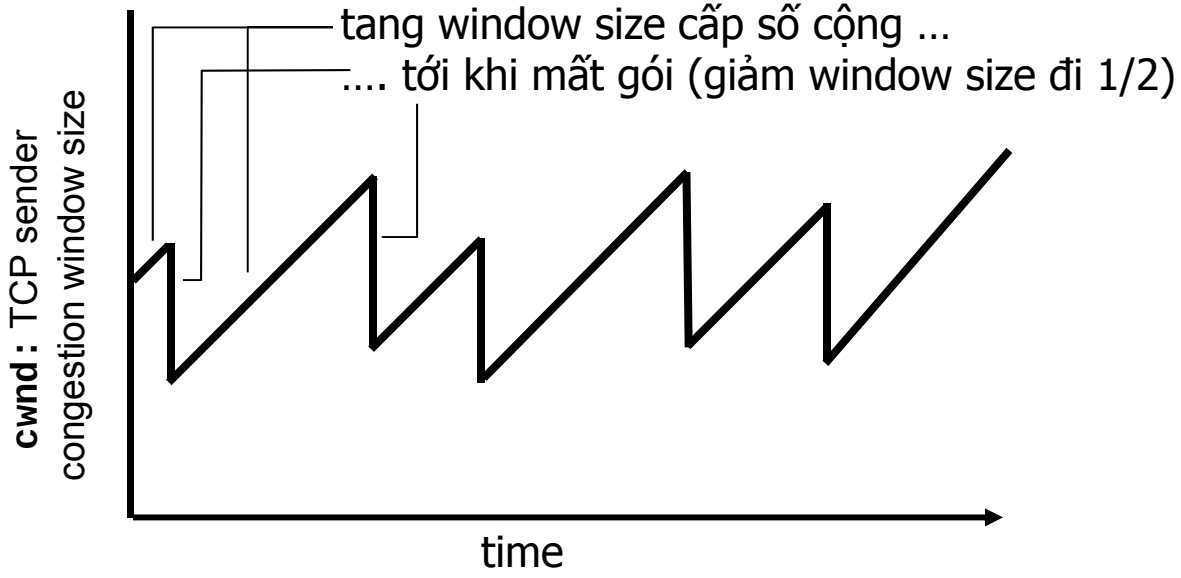
- Trường ER (explicit rate) (2 byte) trong RM cell
  - congested switch có thể giảm giá trị ER trong cell
  - tốc độ gửi của nút gửi là tốc độ tối đa được hỗ trợ của đường đi
- Bít EFCI trong data cell: gán 1 trong congested switch
  - nếu data cell trước RM cell có EFCI là 1, nút nhận gán CI trong RM cell gửi trở lại

Chương 3: Tầng giao vận

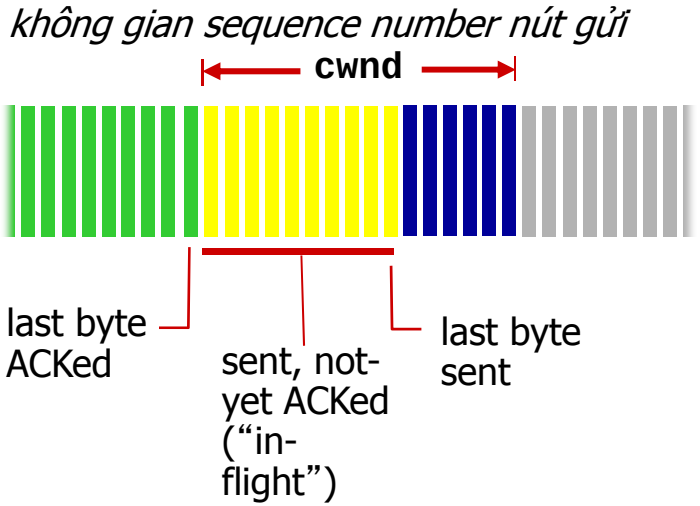
- Dịch vụ của tầng giao vận
- Multiplexing và Demultiplexing
- Truyền không kết nối: UDP
- Nguyên tắc của truyền dữ liệu tin cậy
- Truyền hướng kết nối: TCP
- Nguyên tắc của điều khiển tắc nghẽn
- Điều khiển tắc nghẽn của TCP

Điều khiển tắc nghẽn của TCP:
additive increase multiplicative decrease

- Nút gửi tăng tốc độ truyền (window size), thăm dò băng thông sử dụng được, tới khi xảy ra mất gói
  - *additive increase:* tăng **cwnd** 1 MSS với mỗi RTT tới khi phát hiện mất gói
  - *multiplicative decrease*: giảm **cwnd** đi 1/2 sau khi mất gói



Điều khiển tắc nghẽn của TCP



- Nút gửi giới hạn việc truyền:

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

- **cwnd** thay đổi động

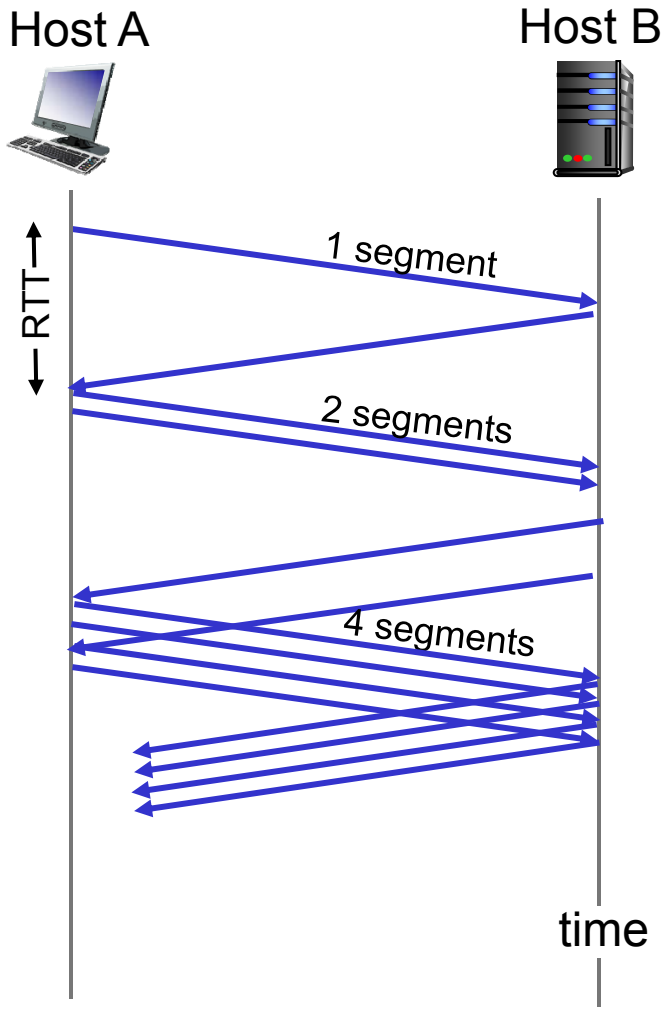
Tốc độ gửi của TCP:

- *gần đúng:* gửi cwnd byte, đợi RTT chờ ACK, rồi gửi byte tiếp

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

TCP Slow Start

- Khi kết nối bắt đầu, tang tốc độ theo hàm mũ tới khi mất gói xảy ra:
  - ban đầu **cwnd** = 1 MSS
  - gấp đô **cwnd** với mỗi RTT
  - kết thúc bằng cách tang **cwnd** cho mỗi ACK nhận được
- Tốc độ ban đầu là chậm nhưng tăng nhanh



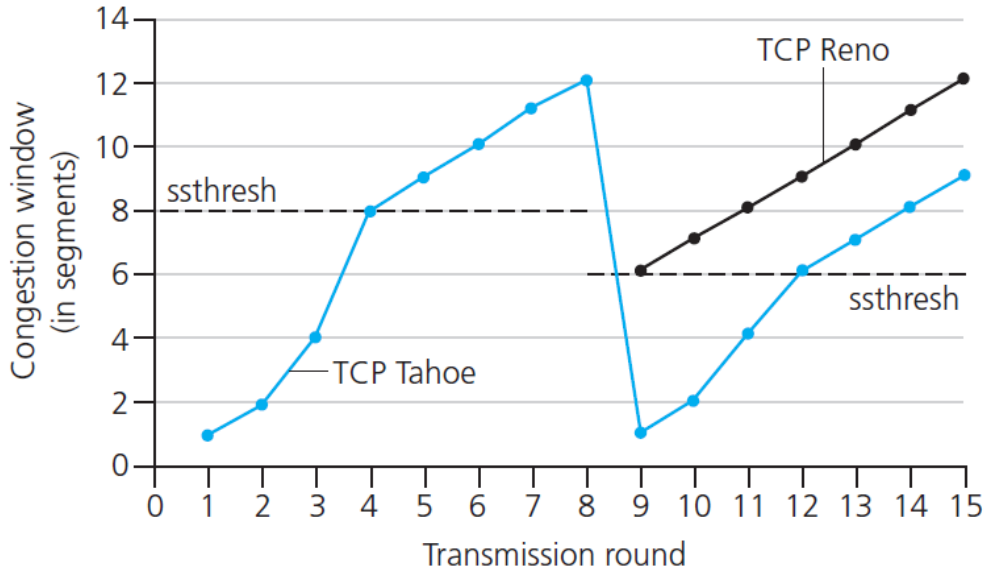
TCP: Phát hiện, xử lý đối với mất gói

- Xác định mất gói bằng timeout:
  - **cwnd** gán bằng 1 MSS;
  - window tăng số mũ (slow start) tới ngưỡng, rồi tăng tuyến tính
- Xác đinh mất gói bằng 3 duplicate ACKs: TCP RENO
  - dup ACKs chỉ khả năng của mạng trong chuyển segment
  - **cwnd** giảm 1/2, rồi tăng tuyến tính
- TCP Tahoe luôn gán **cwnd** bằng 1 (timeout hoặc 3 duplicate acks)

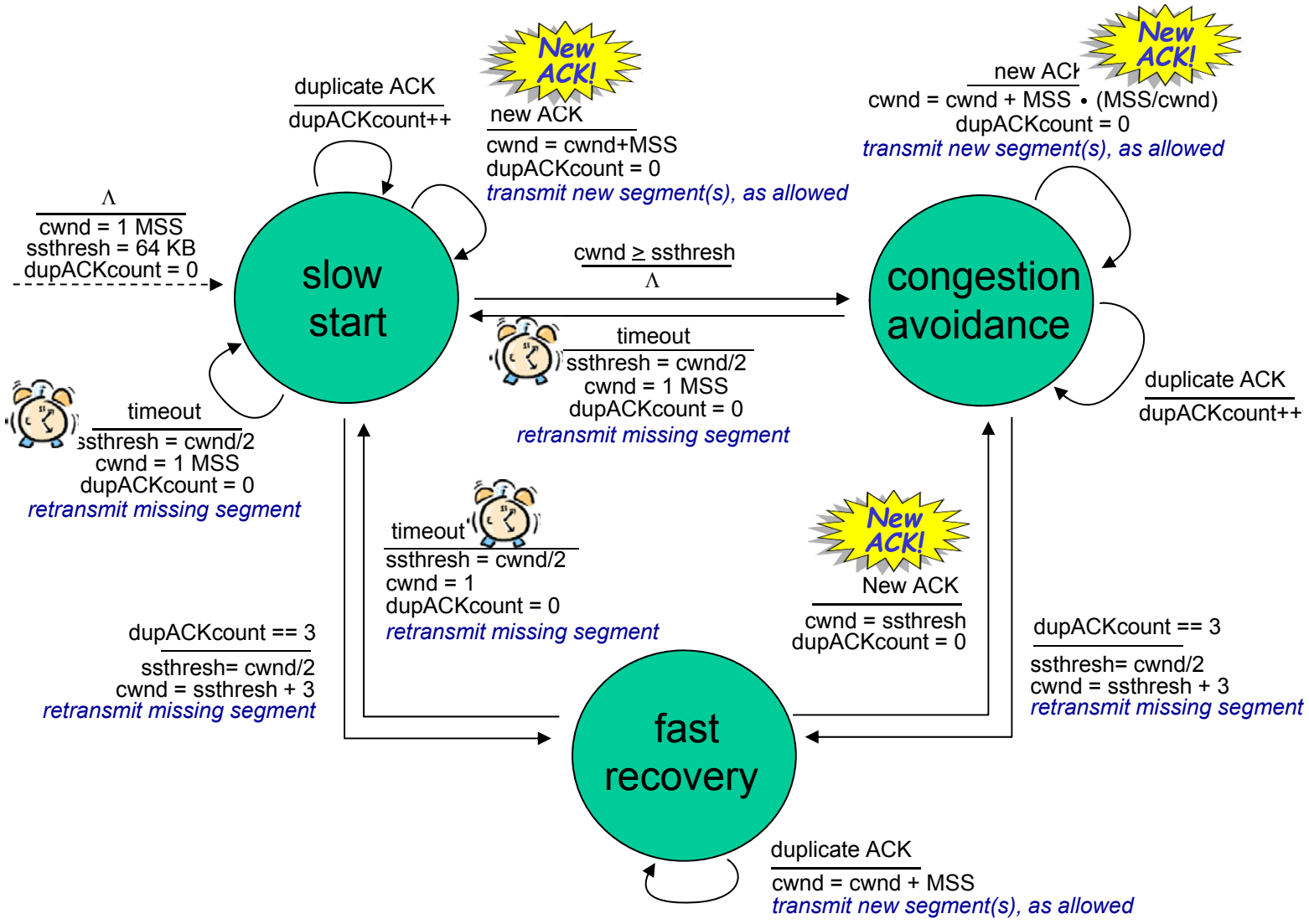
TCP: Chuyển từ slow start sang CA

Khi nào nên chuyển từ tăng số mũ sang tăng tuyến tính?

Khi **cwnd** bằng 1/2 giá trị của nó trước khi timeout



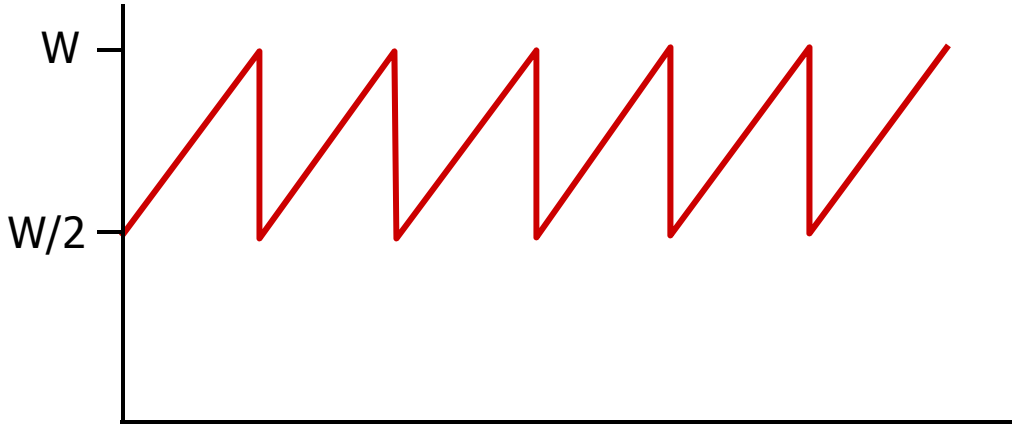
Tóm tắt: Điều khiển tắc nghẽn của TCP



TCP throughput

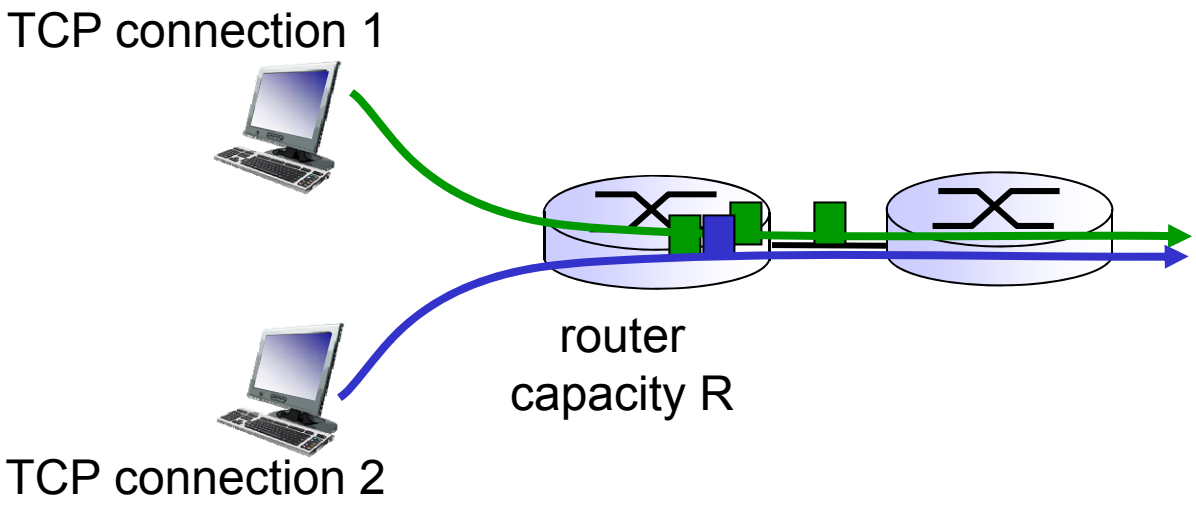
- Giá trị trung bình của TCP throughput là hàm của window size, RTT?
  - bỏ qua slow start, giả sử luôn có dữ liệu để gửi
- W: window size (bytes) khi mất gói xảy ra
  - window size trung bình (# in-flight bytes): ¾ W
  - throughput trung bình : 3/4W mỗi RTT

$$\text{TCP throughput trung bình} = \frac{3}{4}\frac{W}{RTT} \text{ bytes/sec}$$



TCP Fairness

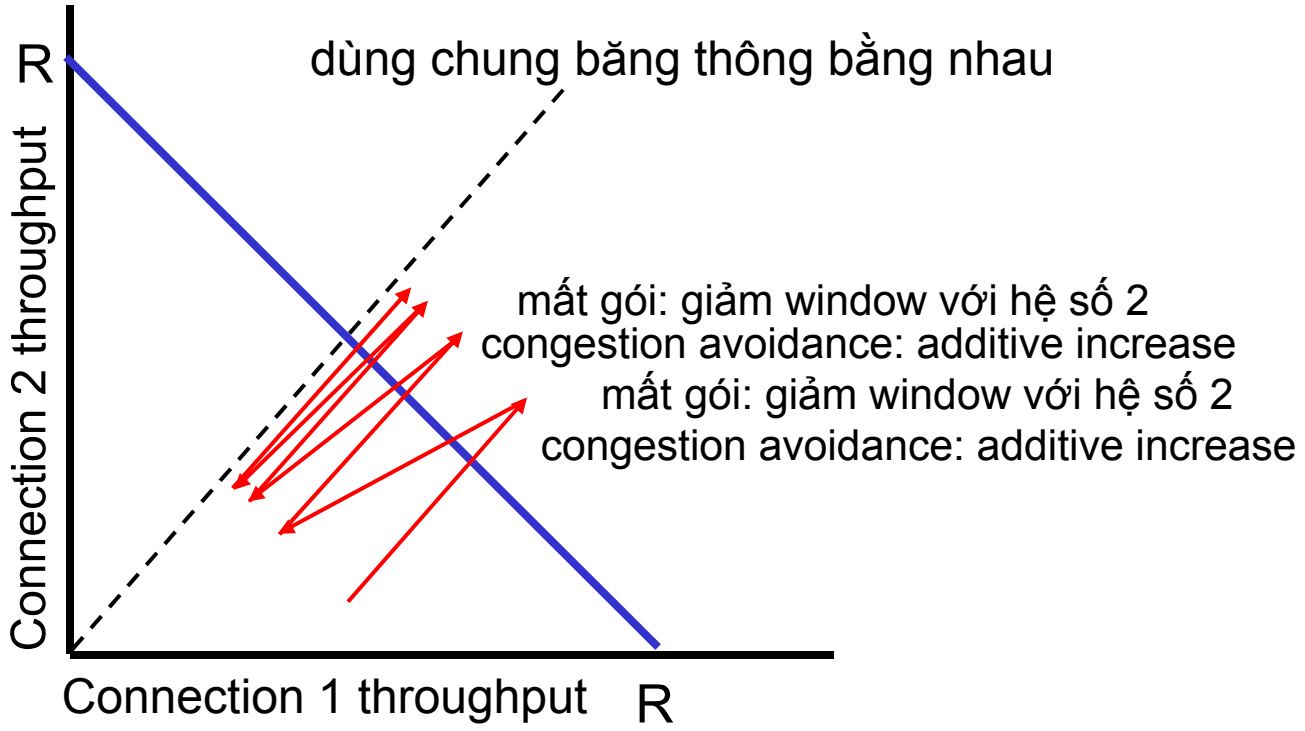
K phiên TCP dùng chung một liên kết có băng thông R, mỗi phiên nên có tốc độ trung bình R/K



Tại sao TCP đảm bảo fairness?

Hai phiên TCP:

- additive increase có độ dốc 1, khi throughput tăng
- multiplicative decrease giảm throughput tỷ lệ



Fairness

Fairness và UDP

- ứng dụng đa phương tiện thường không dùng TCP
  - không muốn tốc độ bị điều khiển bởi điều khiển tắc nghẽn
- dùng UDP:
  - gửi audio/video với tốc độ cố định, có thể mất gói

Fairness và kết nối TCP song song

- ứng dụng có thể mở đồng thời nhiều kế nối giữa 2 host
- ví dụ như web browser
- ví dụ: liên kết có tốc độ R với 9 kết nối:
  - ứng dụng có 1 kết nối TCP, tốc độ R/10
  - ứng dụng có 11 kết nối TCP, tốc độ R/2

Chương 3: Tóm tắt

- Các nguyên tắc bên trong các dịch vụ của tầng giao vận:
  - multiplexing, demultiplexing
  - truyền dữ liệu tin cậy
  - điều khiển luồng
  - điều khiển tắc nghẽn
- Cài đặt trong Internet
  - UDP
  - TCP

Tiếp:

- chuyển từ phần biên mạng (network edge) (tầng giao vận, tầng ứng dụng)
- vào phần lõi mạng (network core)

An interview with Van Jacobson

- Van Jacobson received the ACM SIGCOMM Award in 2001 for outstanding lifetime contribution to the field of communication networks and the IEEE Kobayashi Award in 2002 for "contributing to the understanding of network congestion and developing congestion control mechanisms that enabled the successful scaling of the Internet". He was elected to the U.S. National Academy of Engineering in 2004.
- Please describe one or two of the most exciting projects you have worked on during your career. What were the biggest challenges?
  - School teaches us lots of ways to find answers. In every interesting problem I've worked on, the challenge has been finding the right question. When Mike Karels and I started looking at TCP congestion, we spent months staring at protocol and packet traces asking "Why is it failing?". One day in Mike's office, one of us said "The reason I can't figure out why it fails is because I don't understand how it ever worked to begin with." That turned out to be the right question and it forced us to figure out the "ack clocking" that makes TCP work. After that, the rest was easy.

Mạng máy tính

- Hình ảnh và nội dung trong bài giảng này có tham khảo từ sách và bài giảng của TS. J.F. Kurose and GS. K.W. Ross