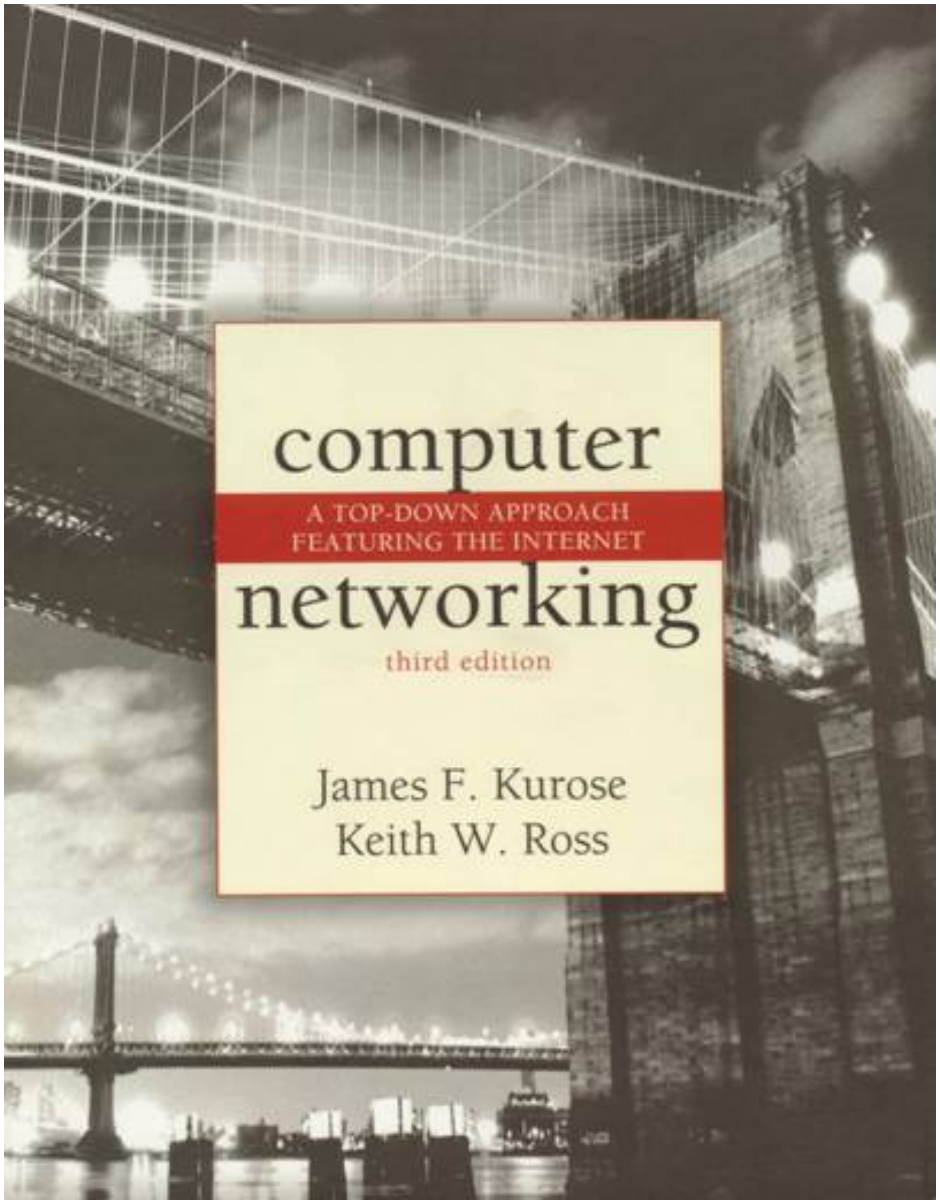


Chương 3
Lớp Transport

Computer Networking: A Top Down Approach Featuring the Internet, 3rd edition.
Jim Kurose, Keith Ross
Addison-Wesley, July 2004.

Slide này được biên dịch sang tiếng Việt theo sự cho phép của các tác giả

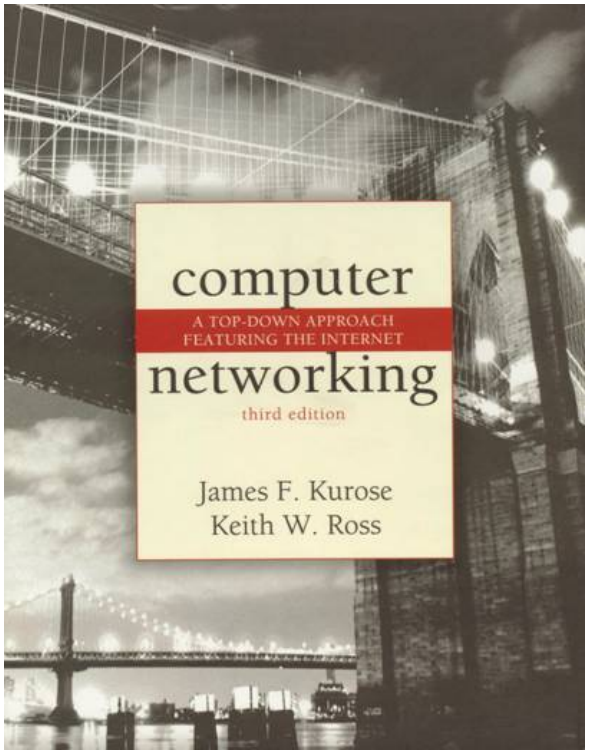
Chương 3: Lớp Transport

Mục tiêu:

- hiểu các nguyên tắc đằng sau các dịch vụ lớp transport:
  - multiplexing/demultiplexing
  - truyền dữ liệu tin cậy
  - điều khiển luồng
  - điều khiển tắc nghẽn
- nghiên cứu về các giao thức lớp Transport trên Internet:
  - UDP: vận chuyển không kết nối (connectionless)
  - TCP: vận chuyển hướng kết nối (connection-oriented)
  - điều khiển tắc nghẽn TCP

Chương 3: Nội dung trình bày

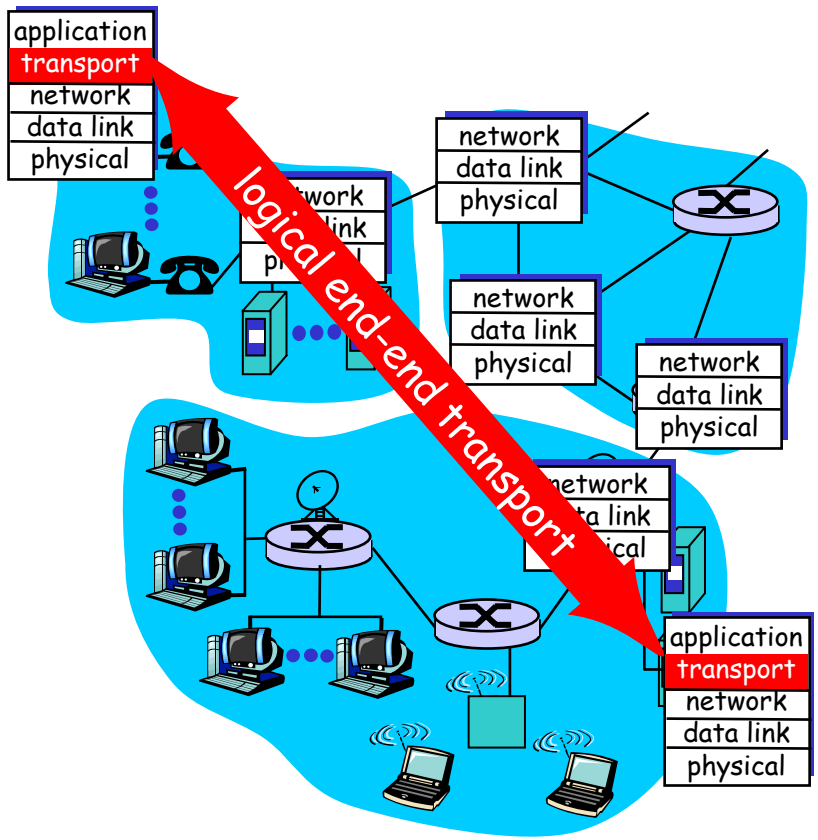
- 3.1 Các dịch vụ lớp Transport
- 3.2 Multiplexing và demultiplexing
- 3.3 Vận chuyển không kết nối: UDP
- 3.4 Các nguyên lý của việc truyền dữ liệu tin cậy
- 3.5 Vận chuyển hướng kết nối: TCP
  - cấu trúc phân đoạn
  - truyền dữ liệu tin cậy
  - điều khiển luồng
  - quản lý kết nối
- 3.6 Các nguyên lý của điều khiển tắc nghẽn
- 3.7 Điều khiển tắc nghẽn TCP



3.1 Các dịch vụ lớp Transport

Các dịch vụ và giao thức Transport

- cung cấp *truyền thông logic* chạy trên các host khác nhau
- các giao thức transport chạy trên các hệ thống đầu cuối
  - phía gửi: cắt các thông điệp ứng dụng thành các đoạn, chuyển cho lớp network
  - phía nhận: tái kết hợp các đoạn thành các thông điệp, chuyển cho lớp application
- có nhiều hơn 1 giao thức transport dành cho các ứng dụng
  - Internet: TCP và UDP



Lớp Transport với lớp network

- *lớp network:* truyền thông logic giữa các host
- *lớp transport:* truyền thông logic giữa các tiến trình
  - dựa vào và làm nổi bật các dịch vụ lớp network

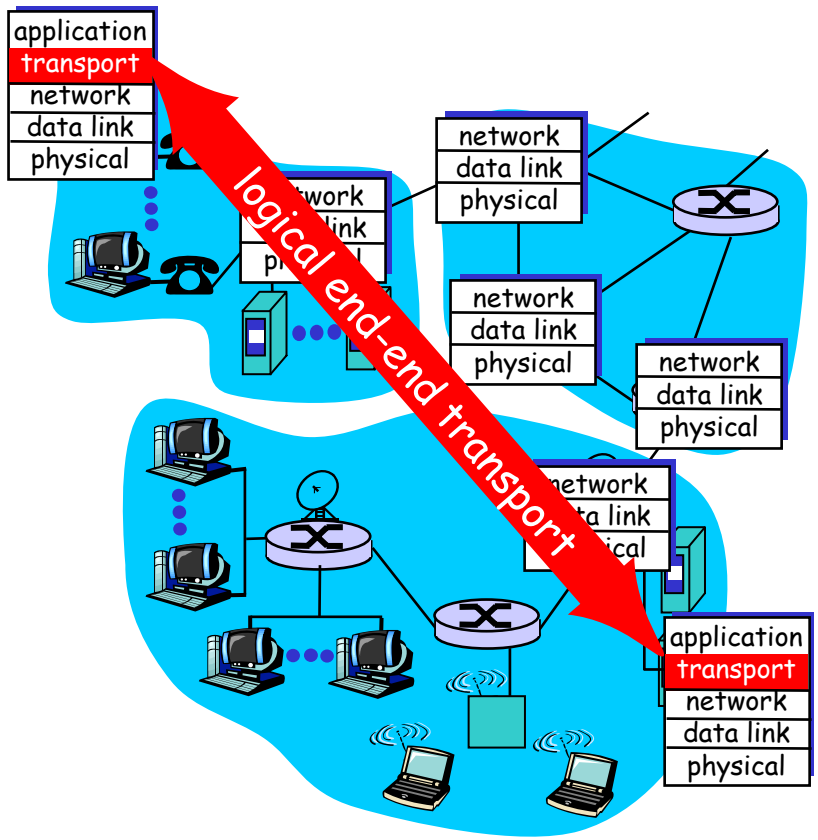
Tình huống tự nhiên tương tự:

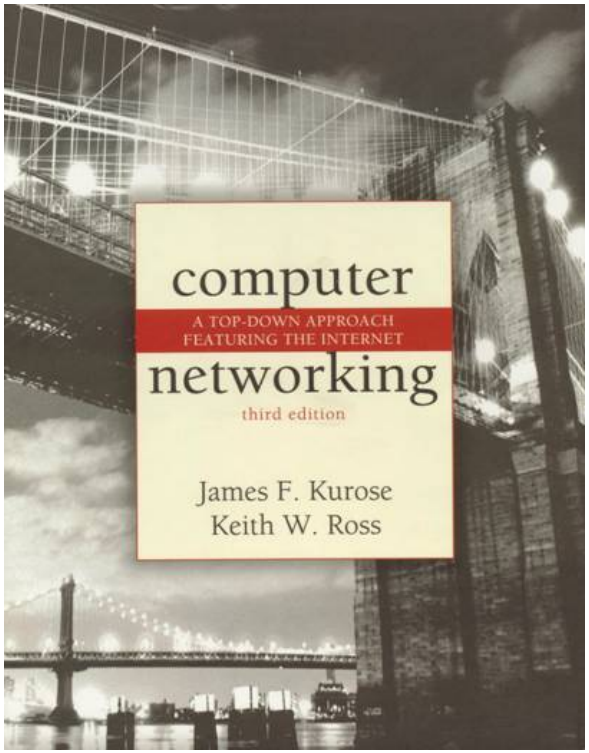
12 đứa trẻ gửi thư đến 12 đứa trẻ khác

- các tiến trình = các đứa trẻ
- các thông điệp = thư trong bao thư
- các host = các gia đình
- giao thức transport = Ann và Bill
- giao thức lớp network = dịch vụ bưu điện

Các giao thức lớp 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
  - mở rộng của giao thức IP
- không có các dịch vụ:
  - bảo đảm trễ
  - bảo đảm bandwidth





3.2 Multiplexing và demultiplexing

Multiplexing/demultiplexing

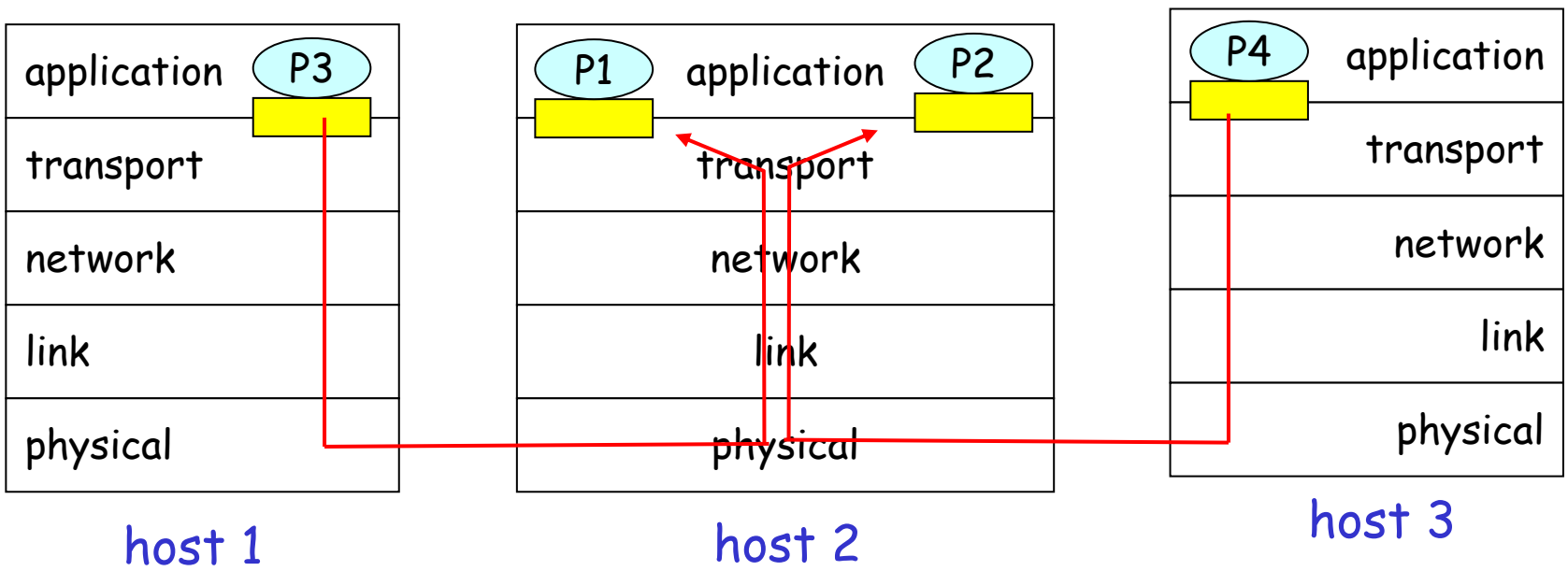
Demultiplexing tại host nhận:

vận chuyển các đoạn đã nhận được đến đúng socket

Multiplexing tại host gửi:

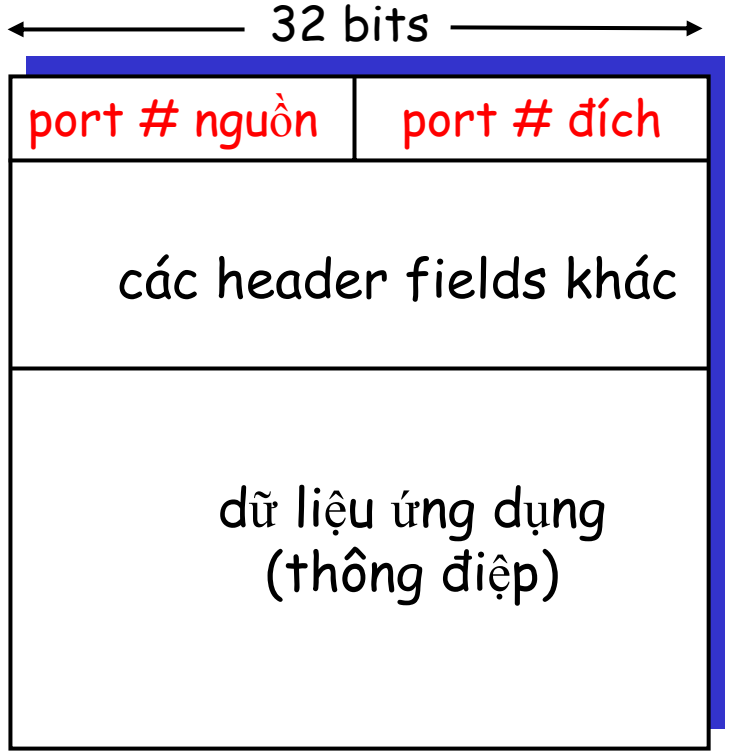
thu nhặt dữ liệu từ nhiều socket, đóng gói dữ liệu với header (sẽ dùng sau đó cho demultiplexing)

= socket = tiến trình



Demultiplexing làm việc như thế nào

- host nhận các IP datagrams
  - mỗi datagram có địa chỉ IP nguồn và IP đích
  - mỗi datagram mang 1 đoạn của lớp transport
  - mỗi đoạn có số port nguồn và đích
- host dùng địa chỉ IP & số port để điều hướng đoạn đến socket thích hợp



dạng thức đoạn TCP/UDP

Demultiplexing không kết nối

- Tạo các sockets với các số port:

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

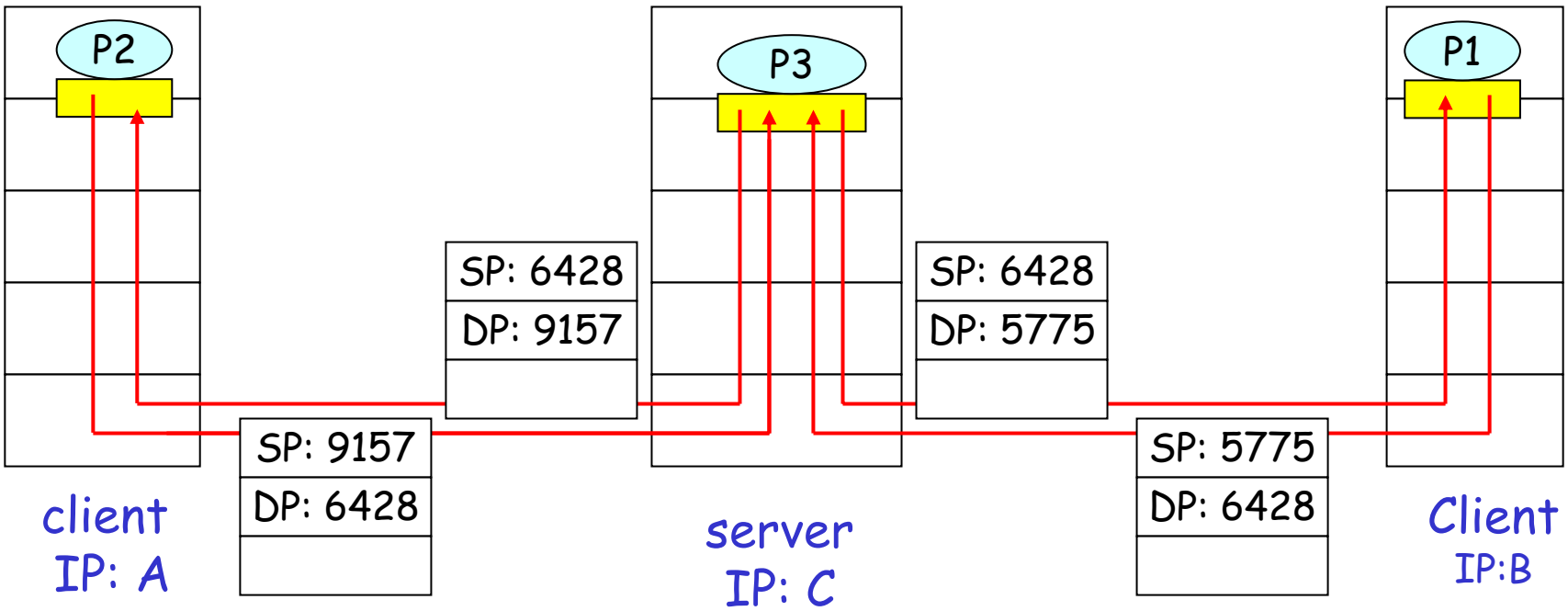
- UDP socket được xác định bởi bộ 2:

(địa chỉ IP, số port đích)

- Khi host nhận đoạn UDP:
  - kiểm tra port đích trong đoạn
  - điều hướng đoạn UDP đến socket nào phù hợp với số port đó
- IP datagrams với địa chỉ IP nguồn và/hoặc số port khác nhau có thể được điều hướng đến cùng socket

Demultiplexing không kết nối (tt)

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



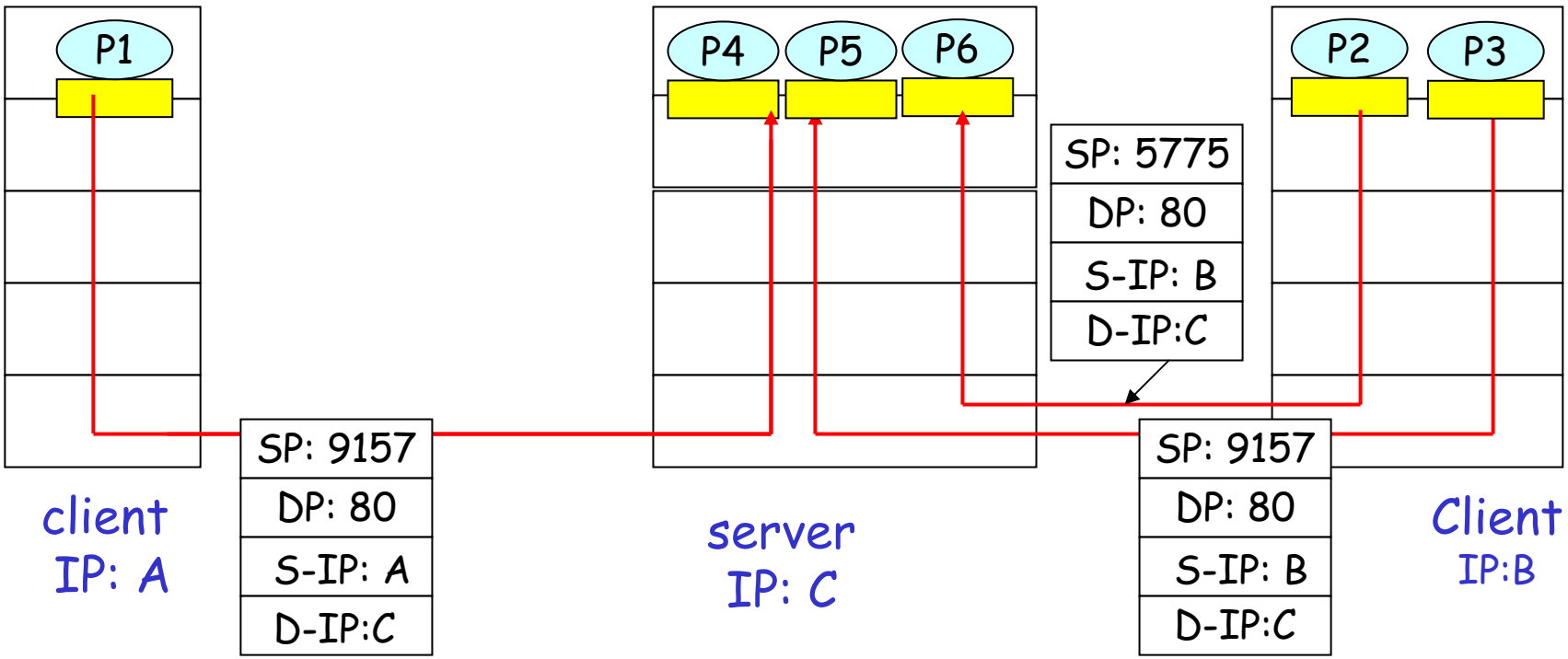
SP cung cấp "địa chỉ trở về"

Demultiplexing hướng kết nối

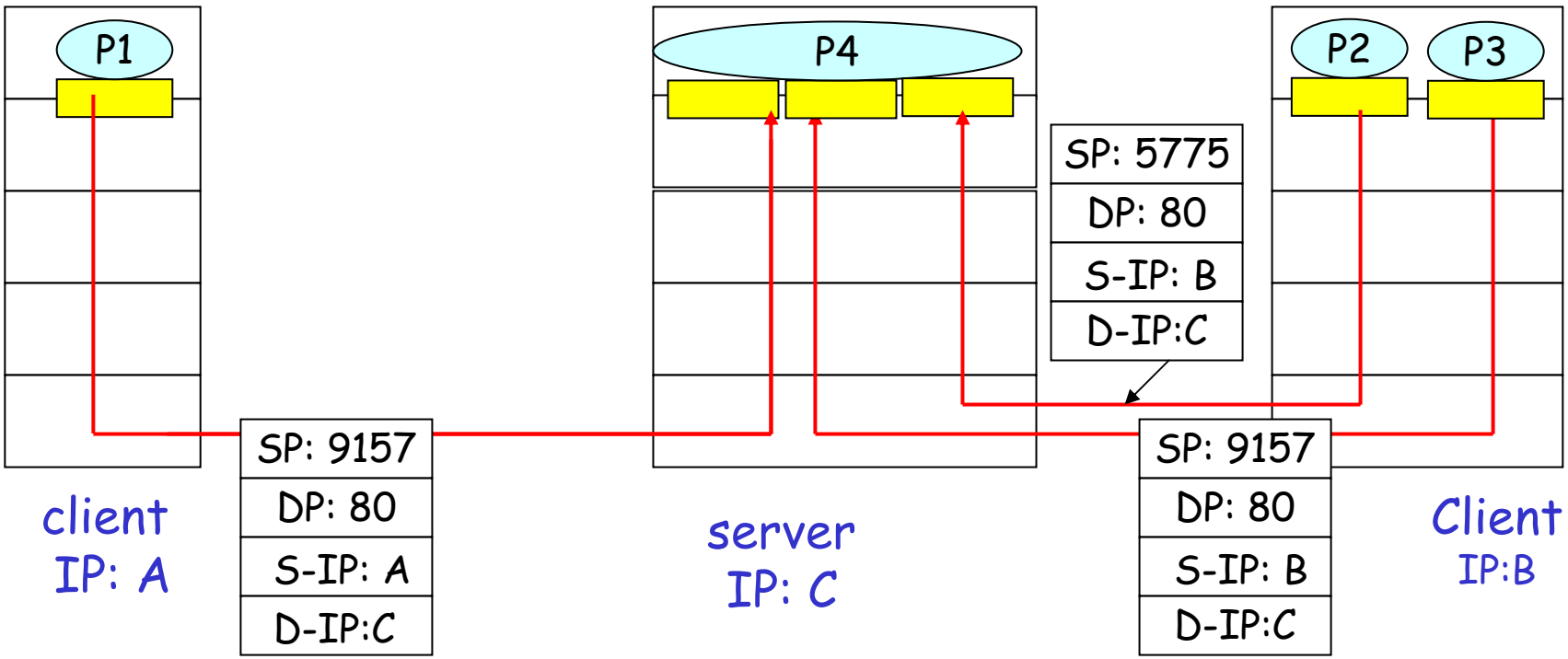
- TCP socket được xác định bởi bộ 4:
  - địa chỉ IP nguồn
  - số port nguồn
  - địa chỉ IP đích
  - số port đích
- host nhận dùng cả 4 giá trị trên để điều hướng đoạn đến socket thích hợp

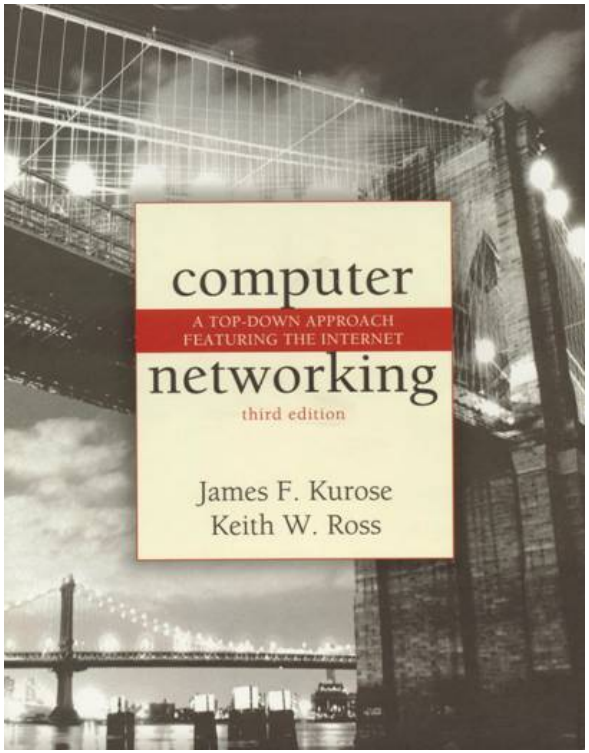
- Host server có thể hỗ trợ nhiều TCP socket đồng thời:
  - mỗi socket được xác định bởi bộ 4 của nó
- Web server có các socket khác nhau cho mỗi kết nối từ client
  - kết nối HTTP không bền vững sẽ có socket khác nhau cho mỗi yêu cầu

Demultiplexing hướng kết nối (tt)



Demultiplexing hướng kết nối: Threaded Web Server





3.3 Vận chuyển không kết nối: UDP

UDP: User Datagram Protocol [RFC 768]

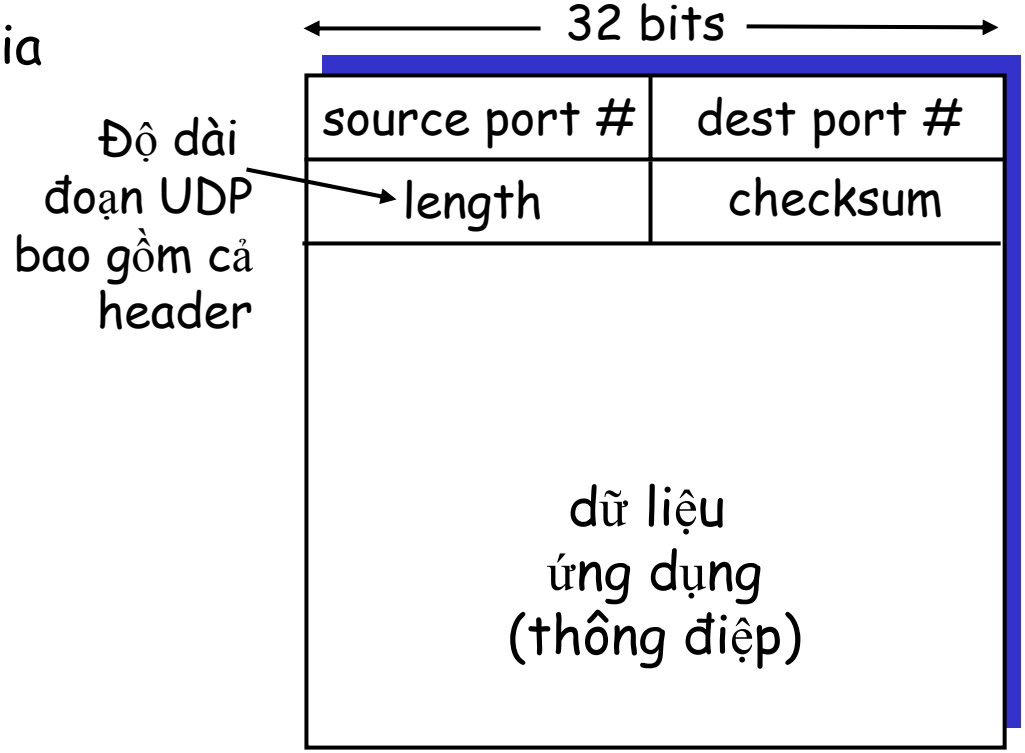
- giao thức Internet transport "đơn giản hóa"
- dịch vụ "best effort", các đoạn UDP có thể:
  - mất mát
  - vận chuyển không thứ tự đến ứng dụng
- *connectionless (không kết nối):*
  - không bắt tay giữa bên nhận và bên gửi UDP
  - mỗi đoạn UDP được quản lý độc lập

Có UDP để làm gì?

- không thiết lập kết nối (giúp có thể thêm delay)
- đơn giản: không trạng thái kết nối tại nơi gửi, nơi nhận
- header của đoạn nhỏ
- không điều khiển tắc nghẽn: UDP có thể gửi nhanh nhất theo mong muốn

UDP: (tt)

- thường dùng cho các ứng dụng streaming multimedia
  - chịu mất mát
  - cảm nhận tốc độ
- ngoài ra, UDP dùng
  - DNS
  - SNMP
- truyền tin cậy trên UDP: thêm khả năng này tại lớp application
  - sửa lỗi



dạng thức đoạn UDP

UDP checksum

Mục tiêu: kiểm tra các "lỗi" (các bit cờ đã bật lên) trong các đoạn đã truyền

bên gửi:

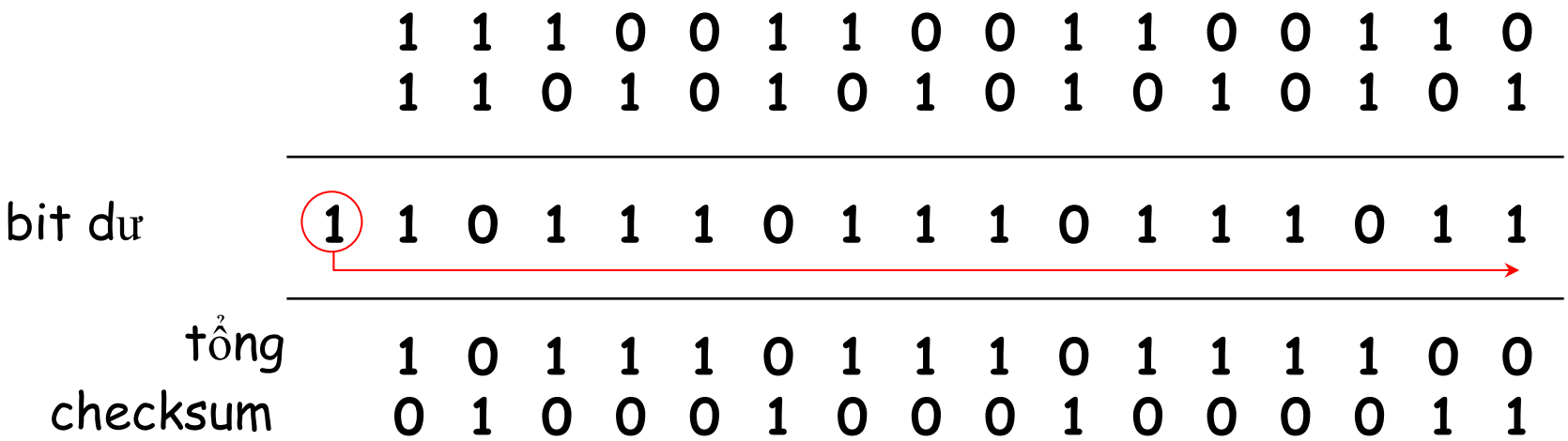
- đối xử các nội dung đoạn như một chuỗi các số nguyên 16-bit
- checksum: bổ sung(tổng bù 1) của các nội dung đoạn
- đặt giá trị checksum vào trường UDP checksum

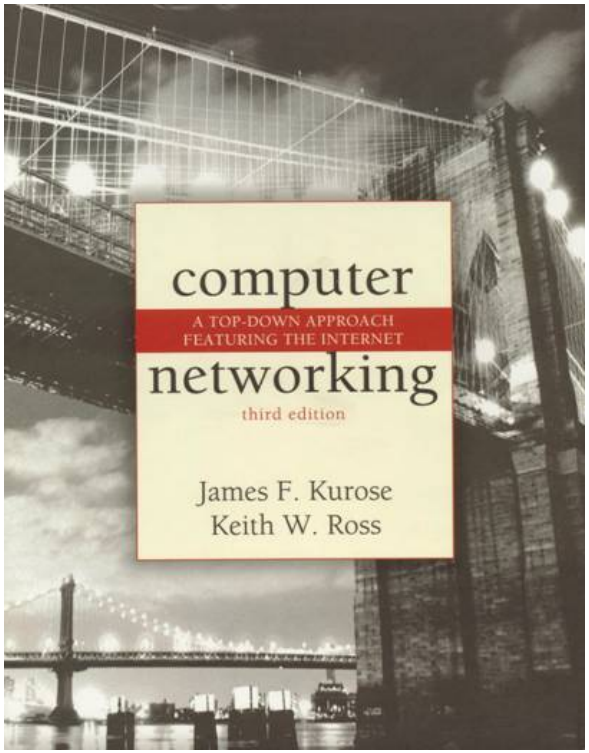
bên nhận:

- tính toán checksum của đoạn đã nhận
- kiểm tra giá trị trên có bằng với giá trị trong trường checksum:
  - NO – có lỗi xảy ra
  - YES – không có lỗi.
  - *Nhưng có thể còn lỗi khác nữa?* Xem tiếp phần sau ….

Ví dụ Checksum

- Lưu ý
  - Khi cộng các số, một bit nhớ ở phía cao nhất có thể sẽ phải thêm vào kết quả
- Ví dụ: cộng hai số nguyên 16-bit

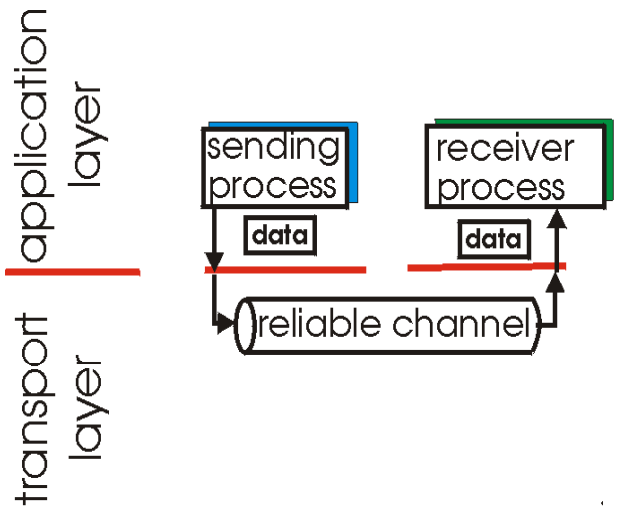




3.4 Các nguyên lý của việc truyền dữ liệu tin cậy

Các nguyên lý truyền dữ liệu tin cậy

- quan trọng trong các lớp application, transport, link
- là danh sách 10 vấn đề quan trọng nhất của mạng

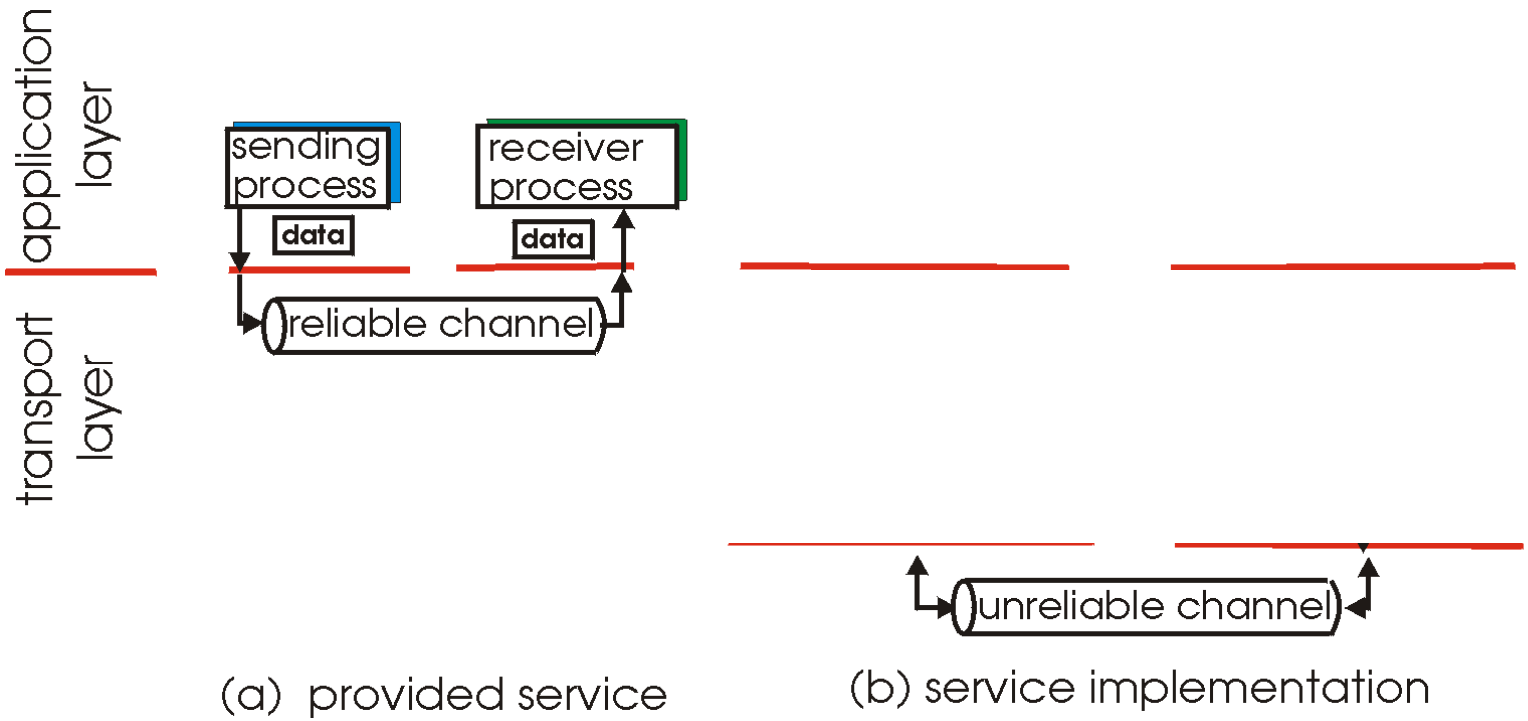


(a) provided service

- các đặc thù 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

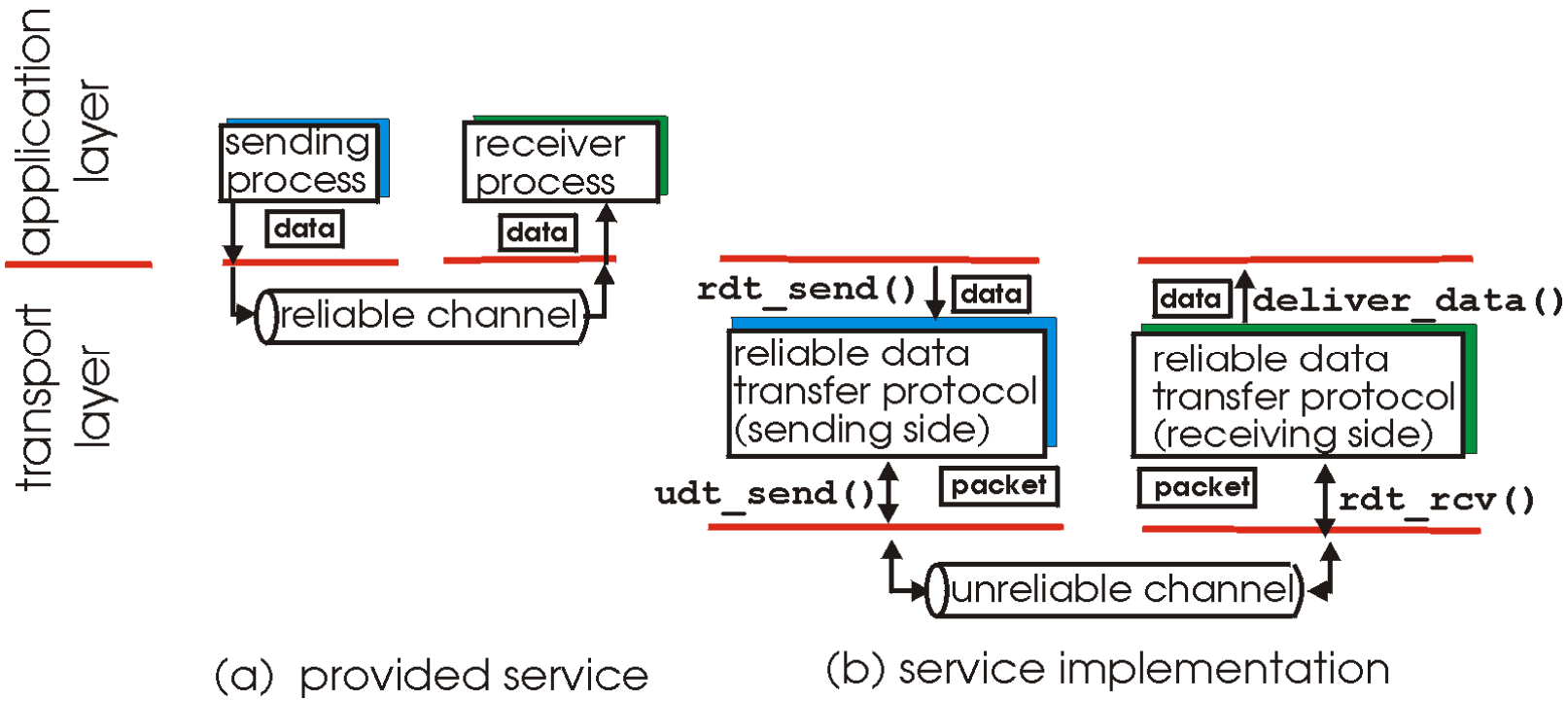
- quan trọng trong các lớp application, transport, link
- là danh sách 10 vấn đề quan trọng nhất của mạng



- các đặc thù 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 lớp application, transport, link
- là danh sách 10 vấn đề quan trọng nhất của mạng

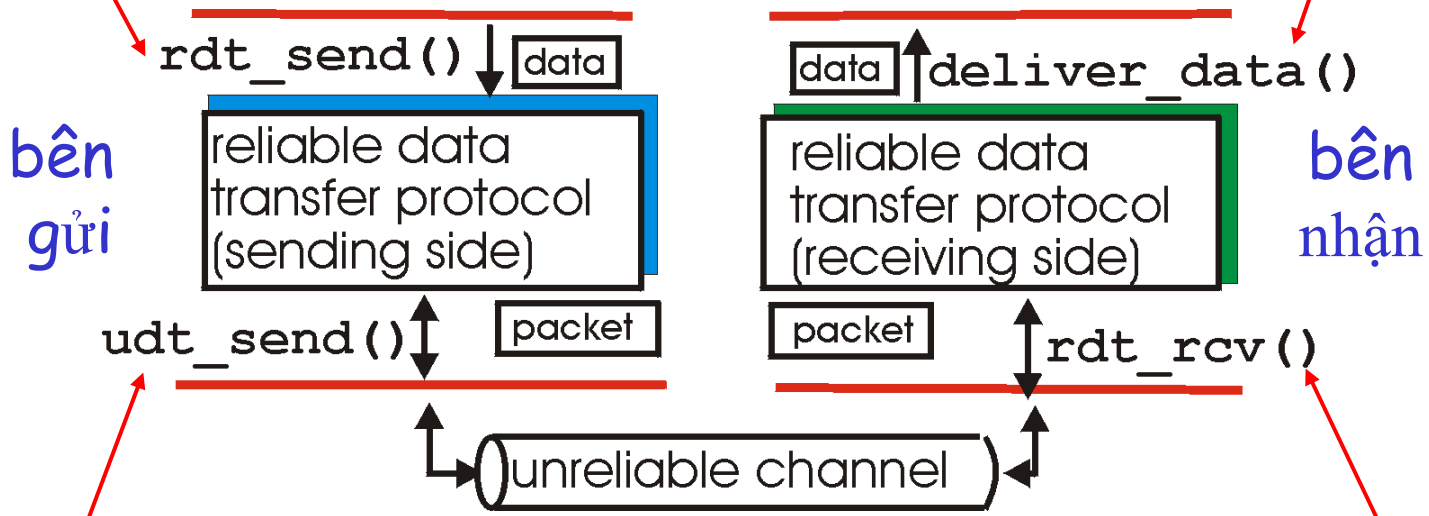


- các đặc thù của kênh truyền không tin cậy sẽ xác định sự phức tạp của giao thức truyền dữ liệu data transfer protocol (rdt)

Truyền dữ liệu tin cậy

rdt_send(): được gọi bởi lớp app. Chuyển dữ liệu cần truyền đến lớp cao hơn bên nhận

deliver_data(): được gọi bởi **rdt** để truyền dữ liệu đến lớp cao hơn



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

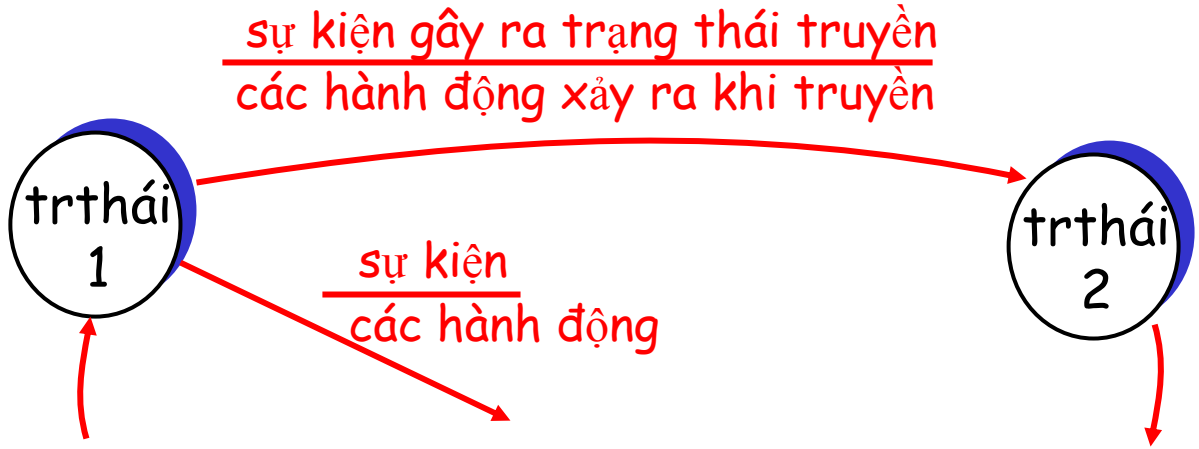
rdt_rcv(): được gọi khi gói đến kênh bên nhận

Truyền dữ liệu tin cậy

Sẽ:

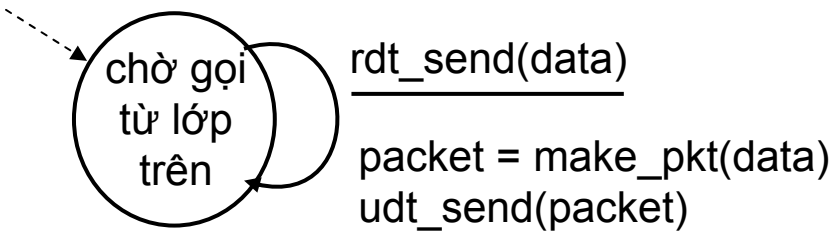
- chỉ xem xét truyền dữ liệu theo 1 hướng duy nhất
  - nhưng điều khiển luồng thông tin sẽ theo cả 2 chiều!
- dùng máy trạng thái hữu hạn (finite state machines-FSM) để xác định bên gửi, bên nhận

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

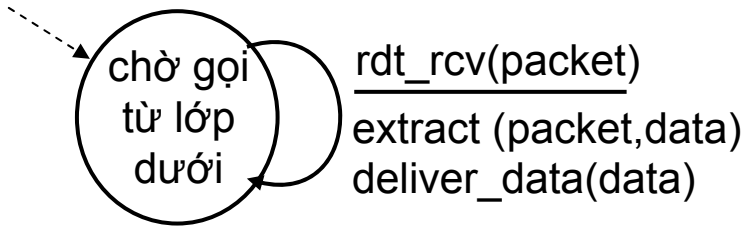


Rdt1.0: truyền dữ liệu tin cậy trên 1 kênh truyền tin cậy

- kênh ưu tiên tin cậy hoàn toàn
  - không có các lỗi
  - không mất mát các gói
- các FSM phân biệt cho bên gửi, bên nhận:
  - bên gửi gửi dữ liệu vào kênh ưu tiên
  - bên nhận nhận dữ liệu từ kênh ưu tiên



bên gửi

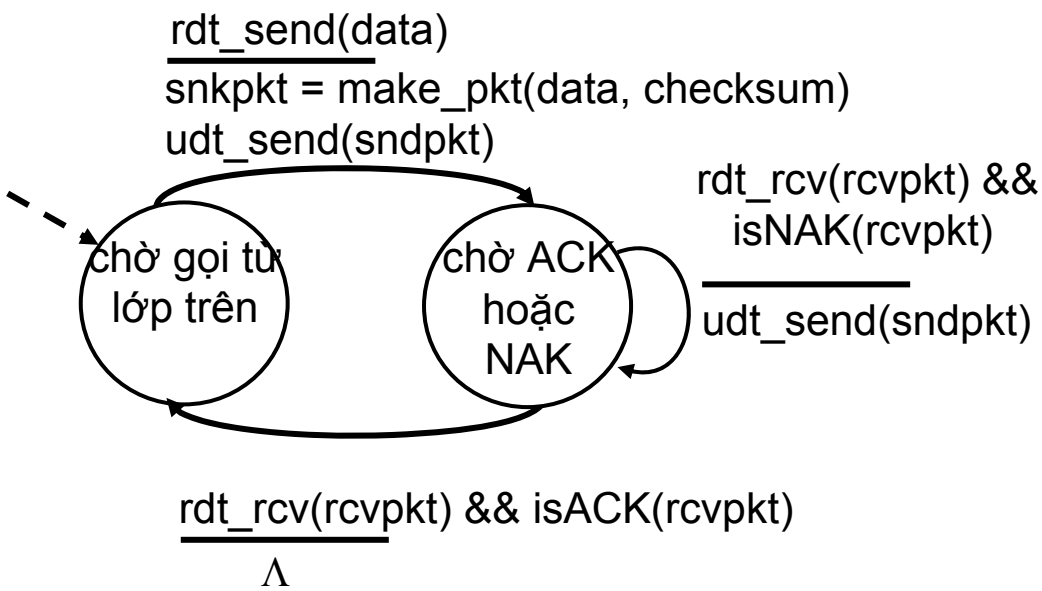


bên nhận

Rdt2.0: kênh với các lỗi

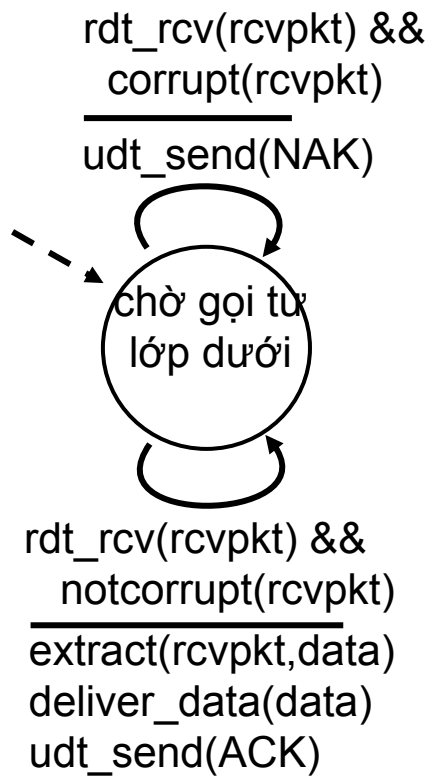
- kênh ưu tiên có thể bật lên một số bit trong gói
  - checksum để kiểm tra các lỗi
- *Hỏi*: làm sao khôi phục các lỗi?
  - *các acknowledgements (ACK):* bên nhận rõ ràng thông báo cho bên gửi rằng quá trình nhận gói tốt
  - *các negative acknowledgements (NAK):* bên nhận rõ ràng thông báo cho bên gửi rằng quá trình nhận gói có lỗi
  - bên gửi gửi lại gói nào được xác nhận là NAK
- **các cơ chế mới trong rdt2.0** (sau **rdt1.0**):
  - kiểm tra lỗi
  - nhận phản hồi: các thông điệp điều khiển (ACK,NAK) bên nhận → bên gửi

rdt2.0: đặc tả FSM

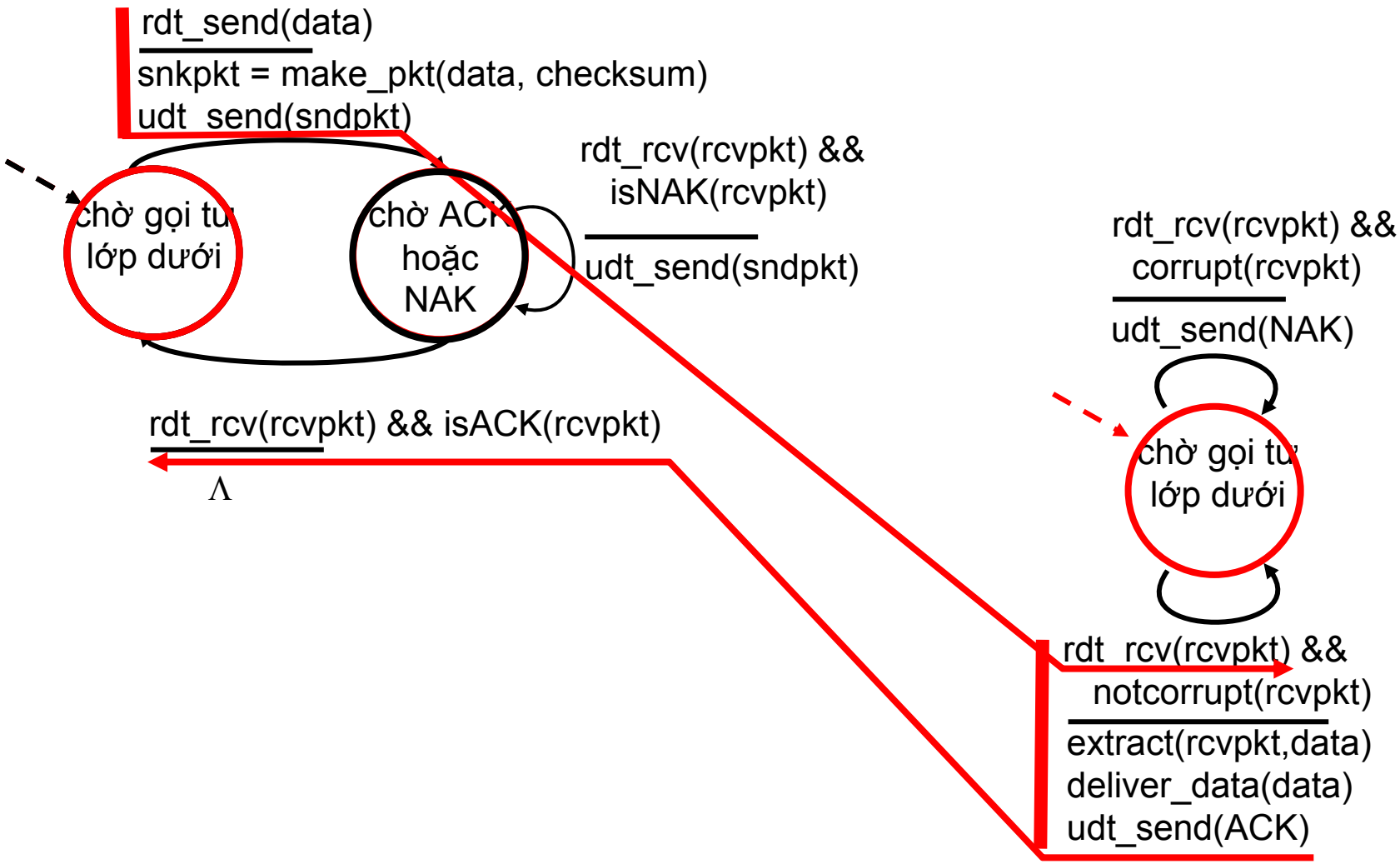


bên gửi

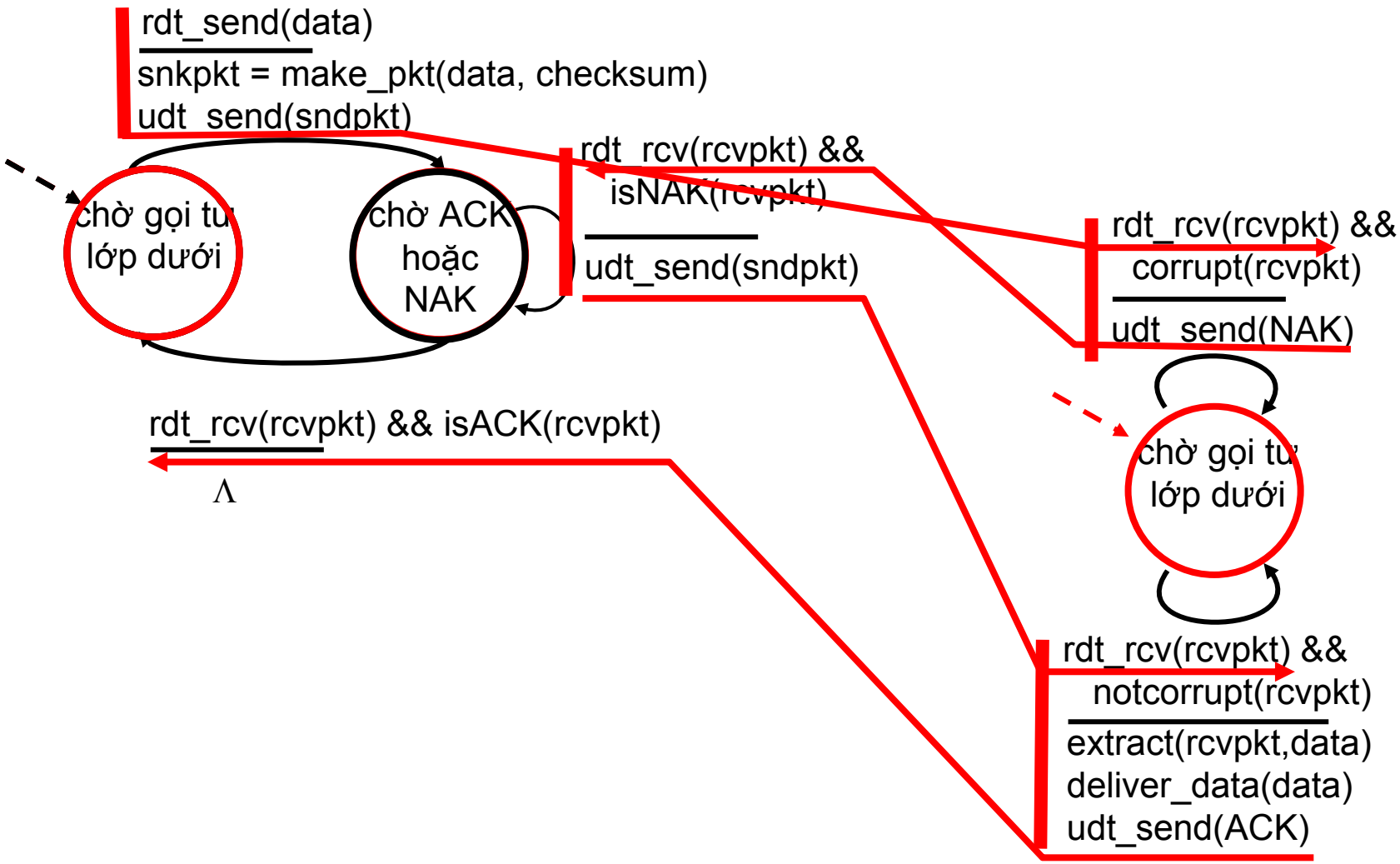
bên nhận



rdt2.0: hoạt động khi không lỗi



rdt2.0: hoạt động khi có lỗi



rdt2.0 có lỗ hổng nghiêm trọng!

Điều gì xảy ra nếu ACK/NAK bị hỏng?

- bên gửi không biết điều gì đã xảy ra tại bên nhận!
- không thể đơn phương truyền lại: khả năng trùng lặp

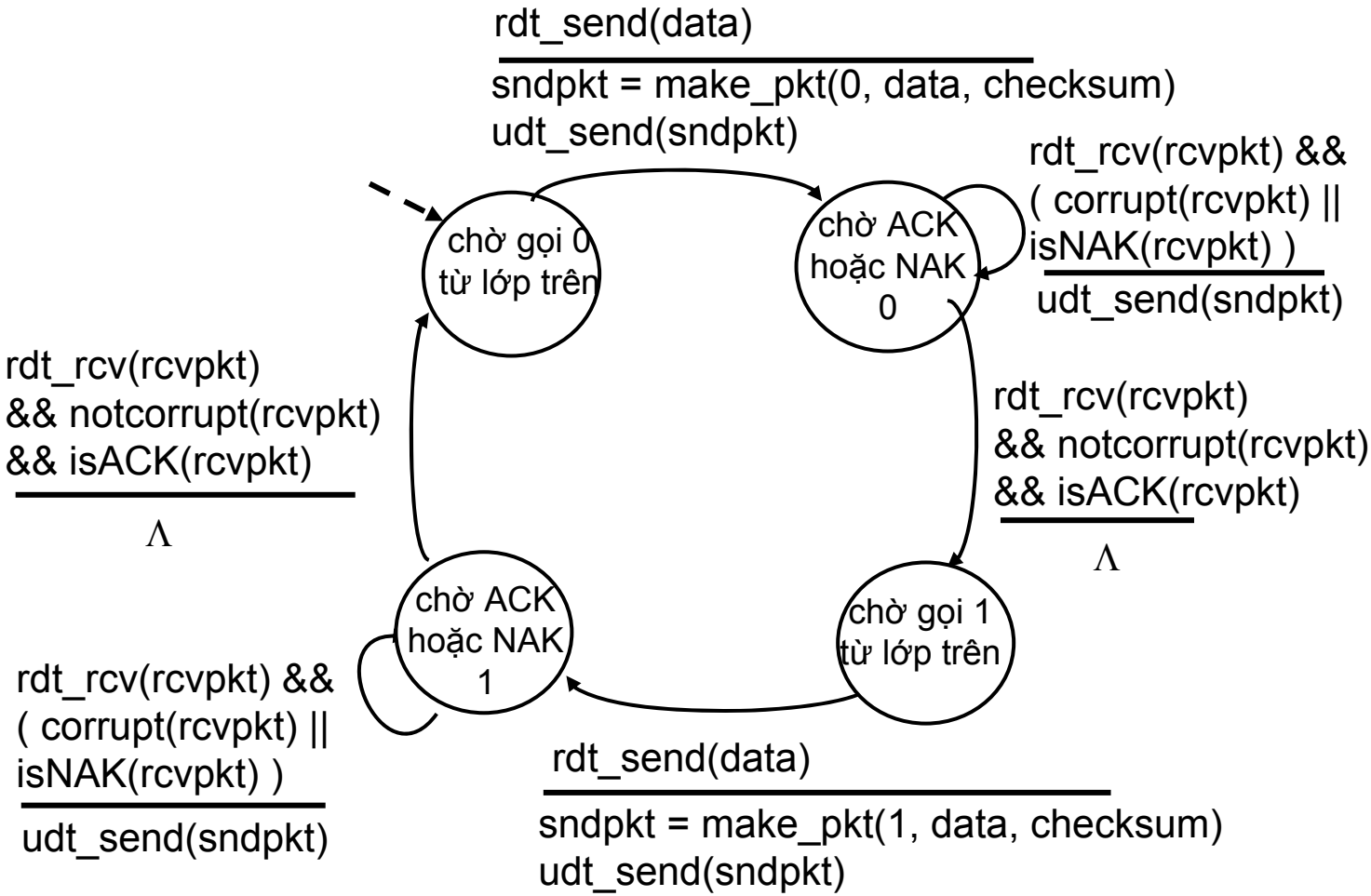
Quản lý trùng lặp:

- bêngửi truyền lại gói hiện tại nếu ACK/NAK bị hỏng
- bên gửi *thêm số thứ tự* vào mỗi gói
- bên nhận hủy (không nhận) gói trùng lặp

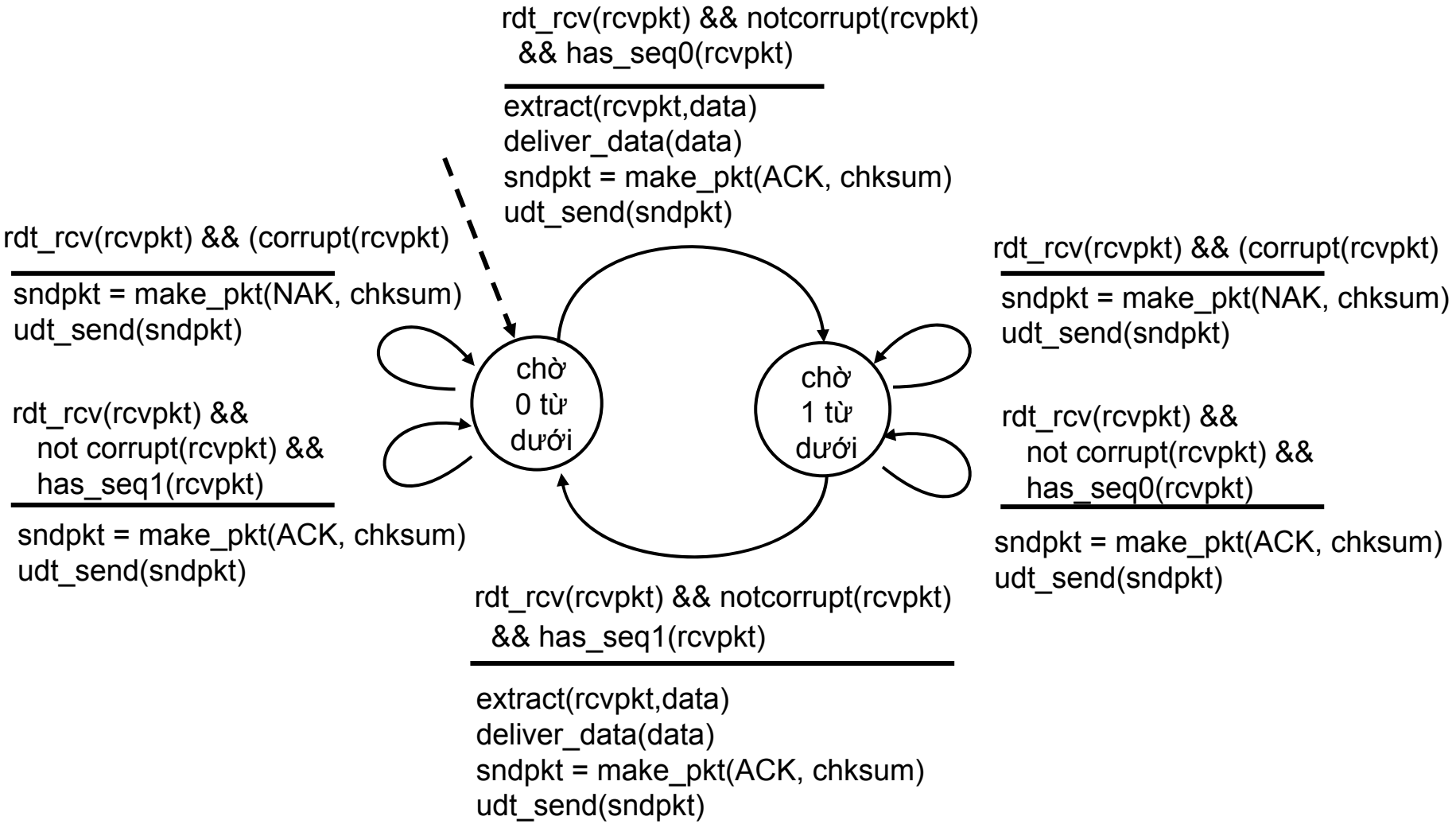
dừng và chờ

bên gửi gửi 1 gói,
sau đó dừng lại chờ phản hồi từ bên nhận

rdt2.1: bên gửi quản lý các ACK/NAK bị hỏng



rdt2.1: bên gửi quản lý các ACK/NAK bị hỏng



rdt2.1: thảo luận

bên gửi:

- số thứ tự # thêm vào gói
- chỉ cần hai số thứ tự (0,1) là đủ. Tại sao?
- phải kiểm tra nếu việc nhận ACK/NAK bị hỏng
- số trạng thái tăng lên 2 lần
  - trạng thái phải "nhớ" gói "hiện tại" có số thứ tự là 0 hay 1

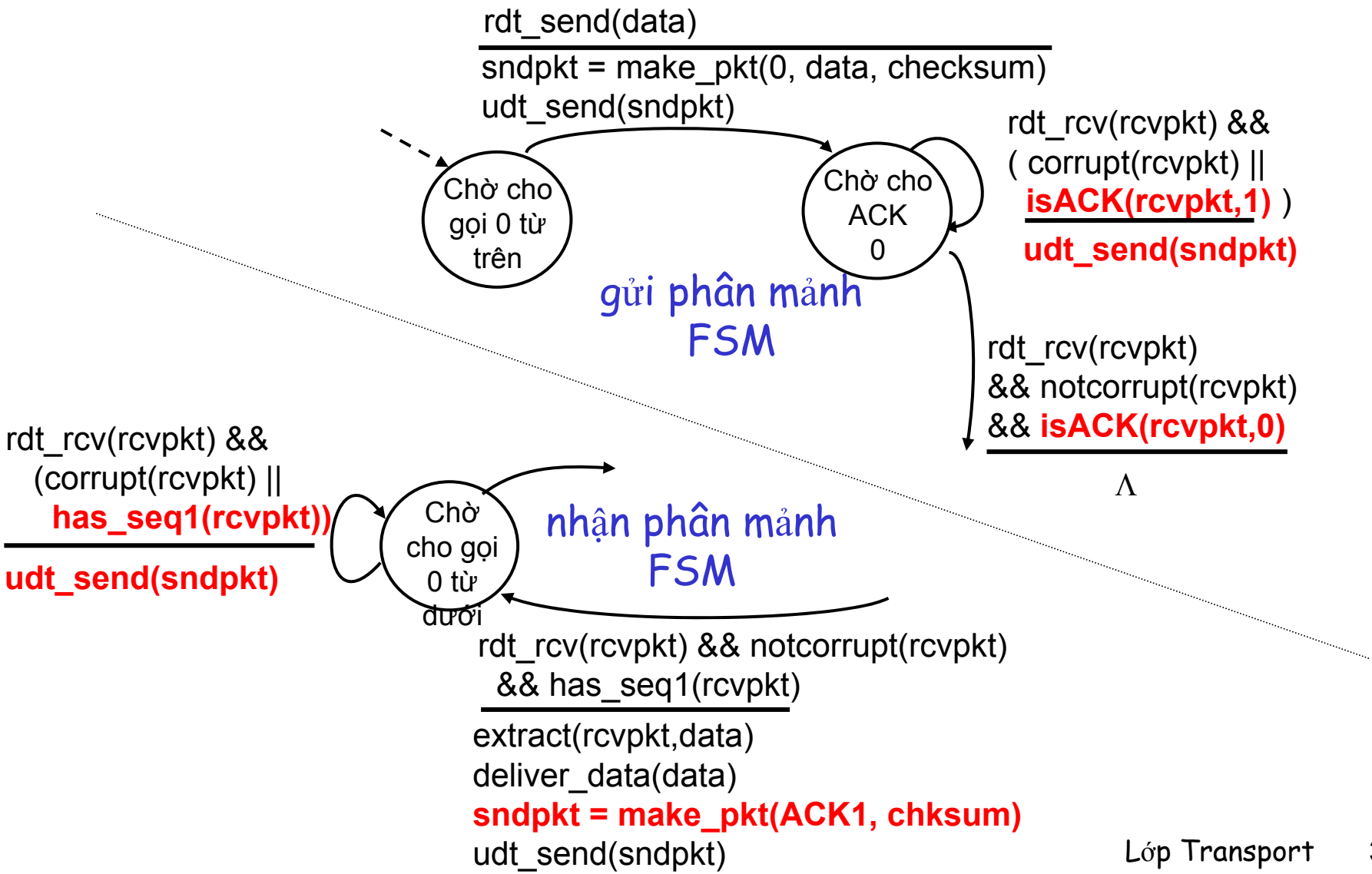
bên nhận:

- phải kiểm tra có nhận trùng gói không
  - trạng thái chỉ rõ có hay không mong chờ số thứ tự 0 hoặc 1
- chú ý: bên nhận không biết ACK/NAK vừa rồi của nó 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 vừa rồi đã nhận tốt
  - bên nhận phải *rõ ràng* chèn số thứ tự của gói vừa ACK
- trùng ACK tại bên gửi hậu quả giống như hành động của NAK: *truyền lại gói vừa rồi*

rdt2.2: gửi, nhận các mảnh



rdt3.0: các kênh với lỗi *và* mất mát

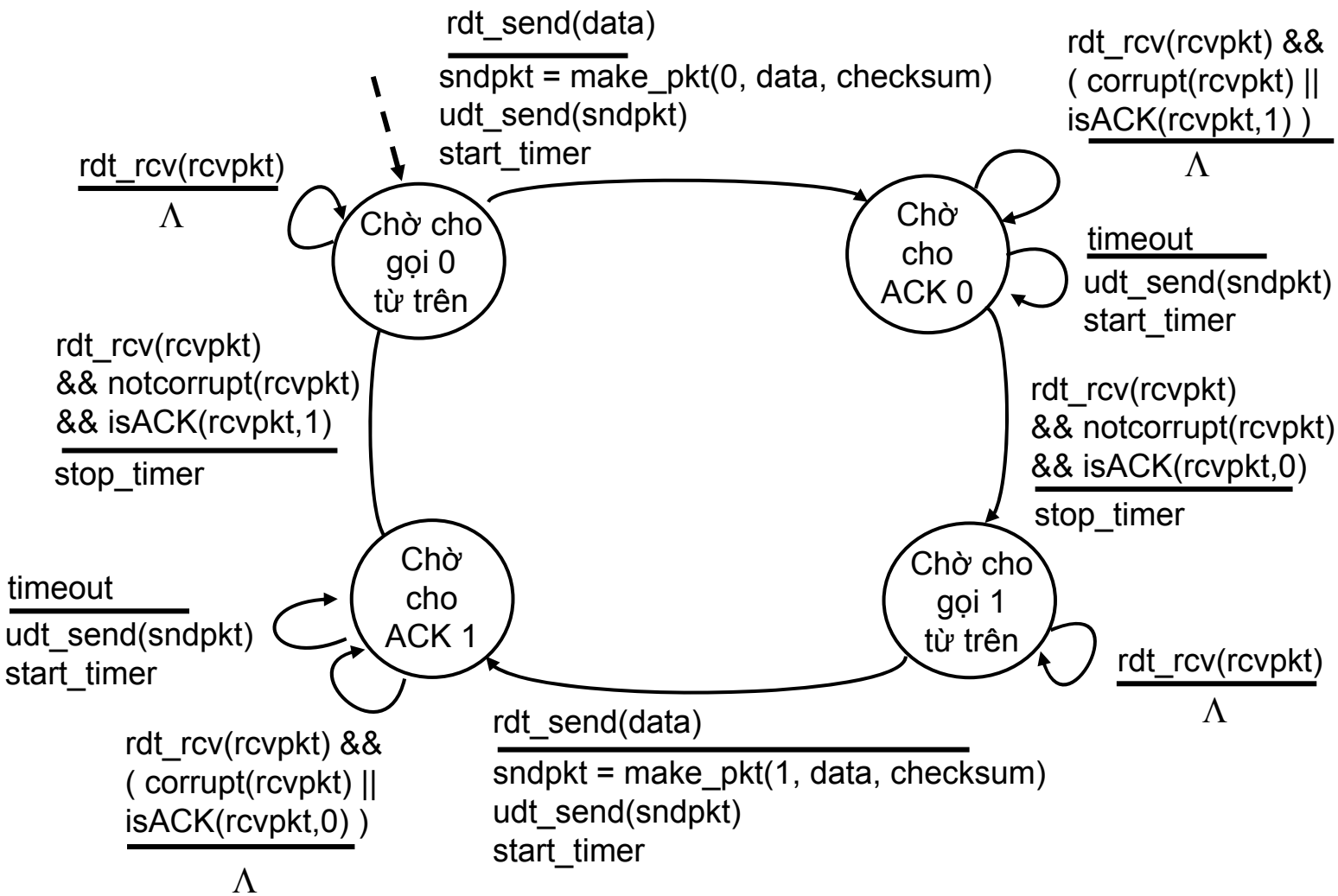
Giả định mới: kênh ưu tiên cũng có thể làm mất các gói (dữ liệu hoặc các ACK)

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

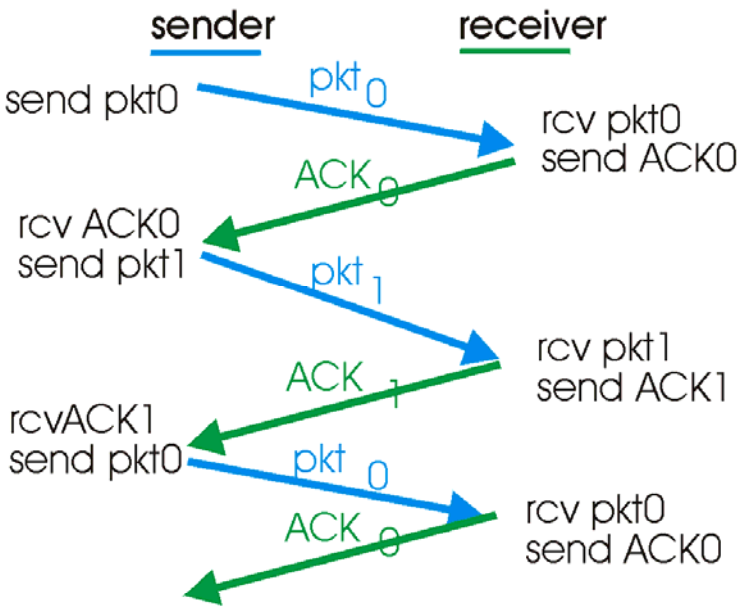
Cách tiếp cận: bên gửi chờ ACK trong khoảng thời gian "chấp nhận được"

- truyền lại nếu không nhận ACK trong khoảng thời gian này
- nếu gói (hoặc ACK) chỉ trễ (không mất):
  - truyền lại sẽ gây trùng, nhưng dùng số thứ tự sẽ giải quyết được
  - bên nhận phải xác định số thứ tự của gói vừa gửi ACK
- cần bộ định thì đếm lùi

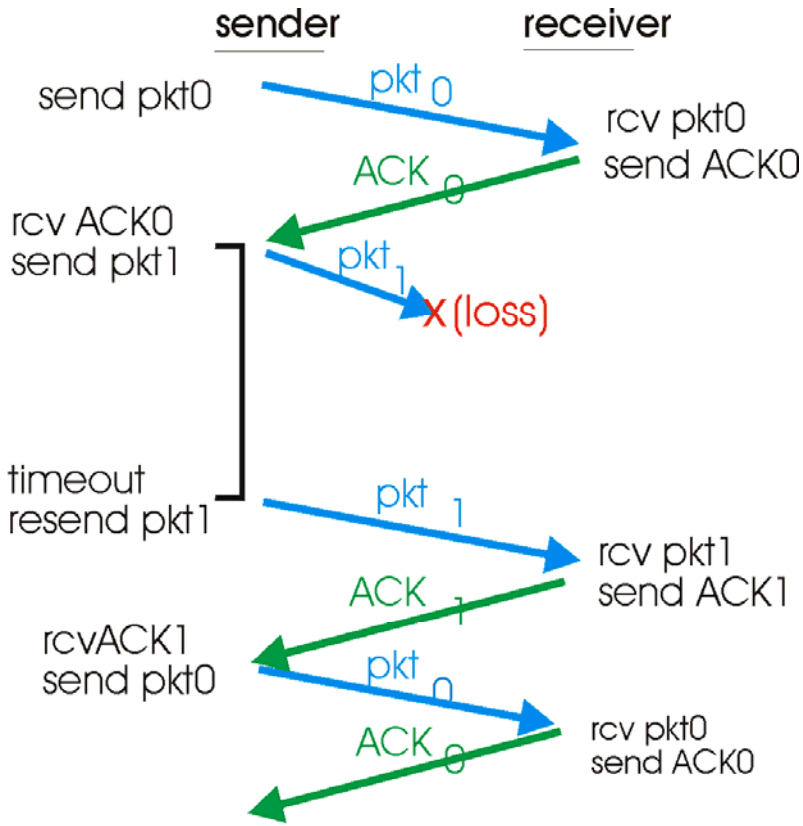
rdt3.0 gửi


rdt_send(data)
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isACK(rcvpkt,1))
Λ
rdt_rcv(rcvpkt)
Λ
Chờ cho gọi 0 từ trên
Chờ cho ACK 0
timeout
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt,1)
stop_timer
rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt,0)
stop_timer
Chờ cho ACK 1
Chờ cho gọi 1 từ trên
timeout
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt)
Λ
rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isACK(rcvpkt,0))
Λ
rdt_send(data)
sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)
start_timer

hành động của rdt3.0

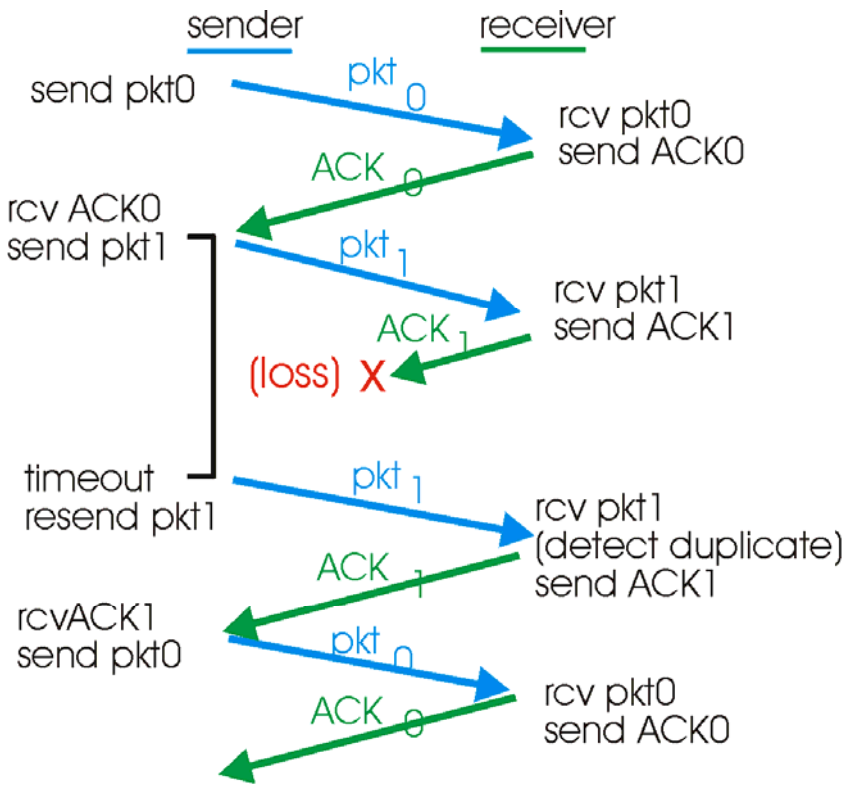


(a) operation with no loss

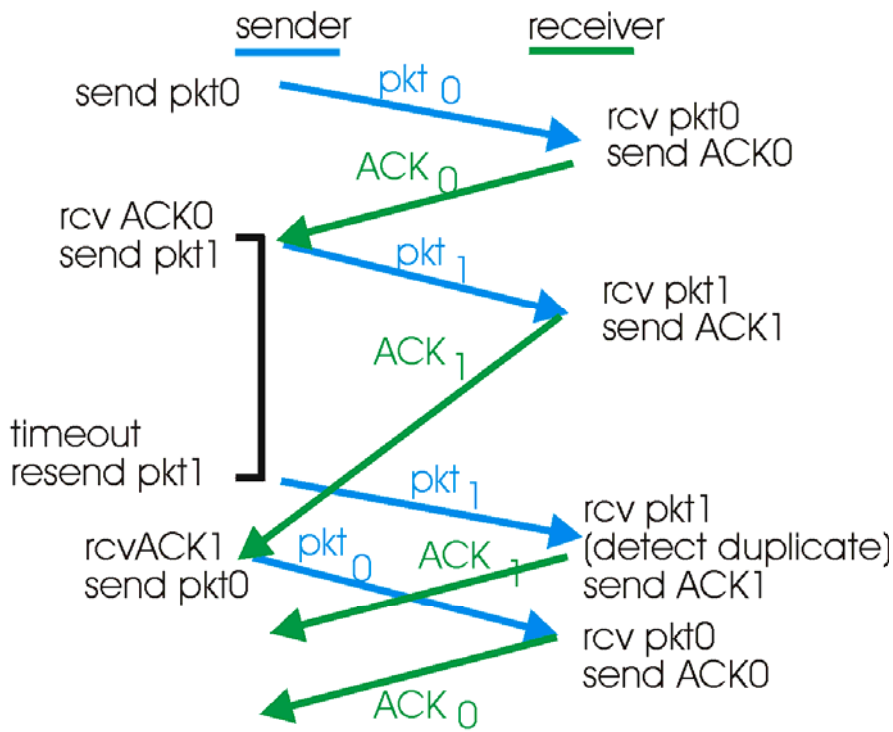


(b) lost packet

hành động của rdt3.0



(c) lost ACK



(d) premature timeout

Hiệu suất của rdt3.0

- rdt3.0 làm việc được, nhưng đánh giá hiệu suất hơi rắc rối
- ví dụ: liên kết 1 Gbps, trễ lan truyền giữa hai đầu cuối là 15 ms, gói 1KB:

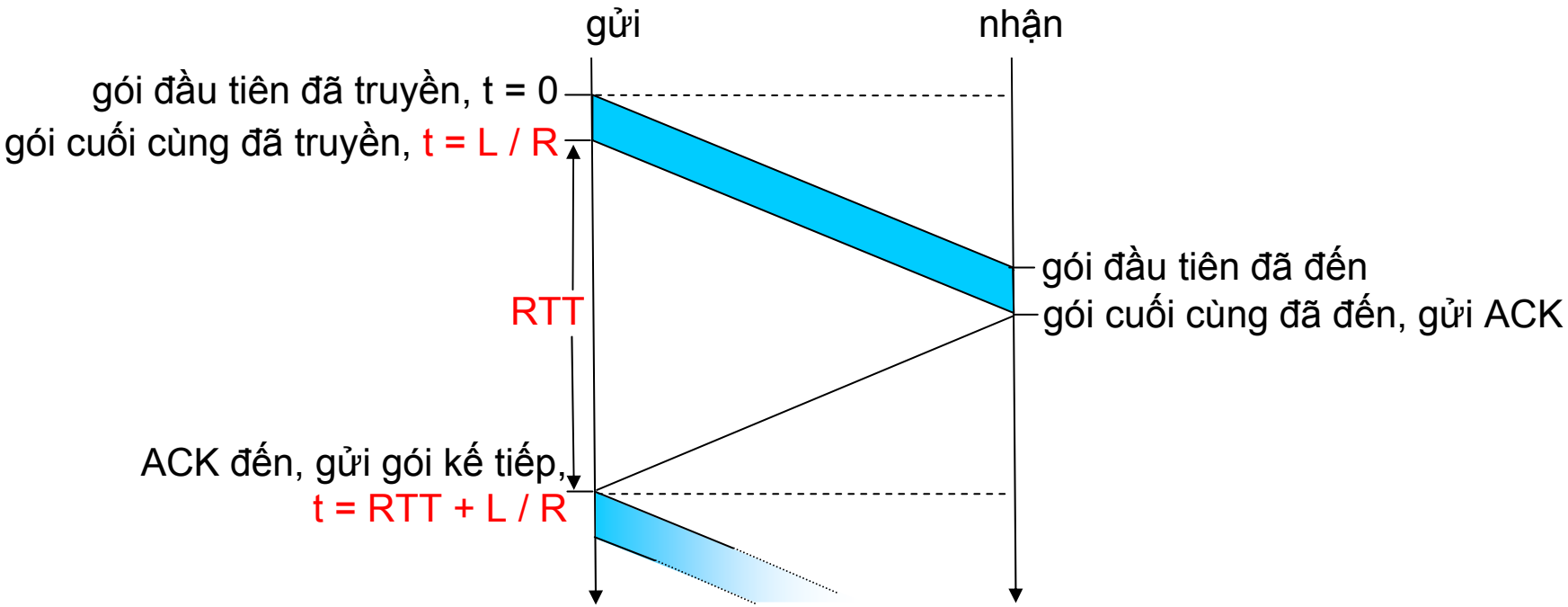
$$T_{truyền} = \frac{L \text{ (độ dài gói tính bằng bits)}}{R \text{ (tốc độ truyền, bps)}} = \frac{8kb/pkt}{10**9\ b/sec} = 8 \text{ microsec}$$

  - U_{sender}: độ khả dụng -

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

  - gói 1KB mỗi 30 msec -> 33kB/s trên đường truyền 1 Gbps
  - giao thức network hạn chế việc dùng các tài nguyên vật lý!

rdt3.0: hoạt động dừng-và-chờ

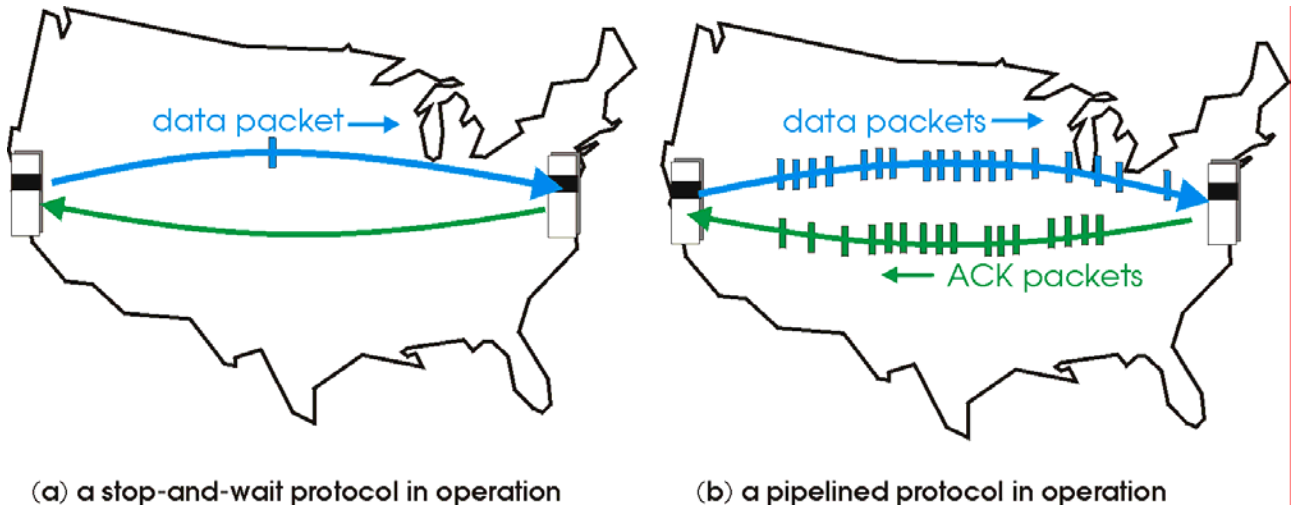


$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

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 tăng dần
- phải có bộ nhớ đệm tại nơi gửi và/hoặc nơi nhận

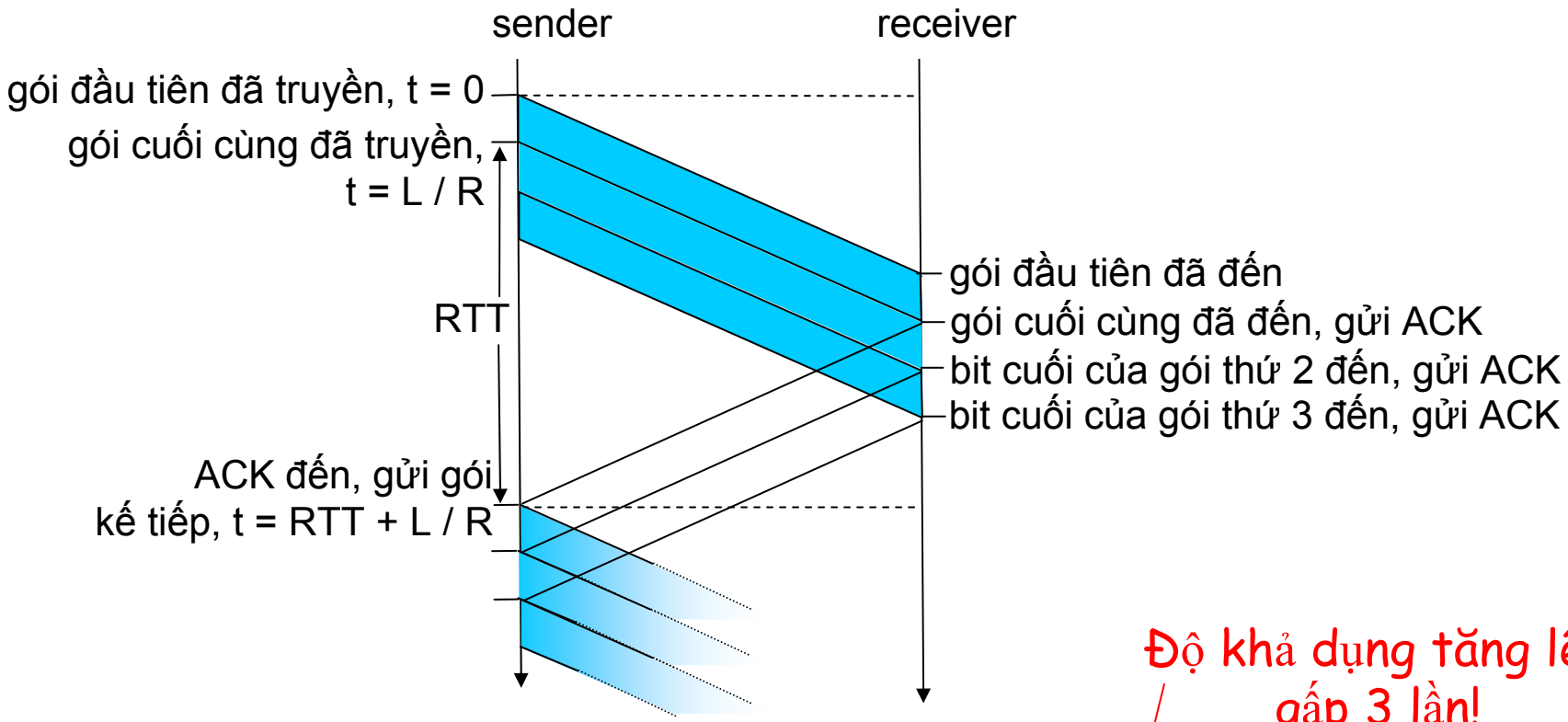


(a) a stop-and-wait protocol in operation

(b) a pipelined protocol in operation

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

Pipelining: độ khả dụng tăng



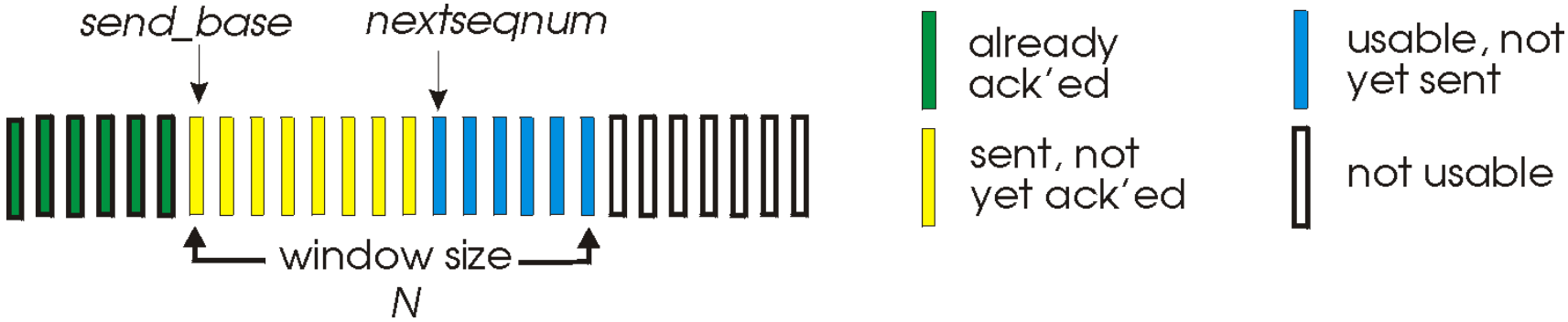
Độ khả dụng tăng lên gấp 3 lần!

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

Go-Back-N

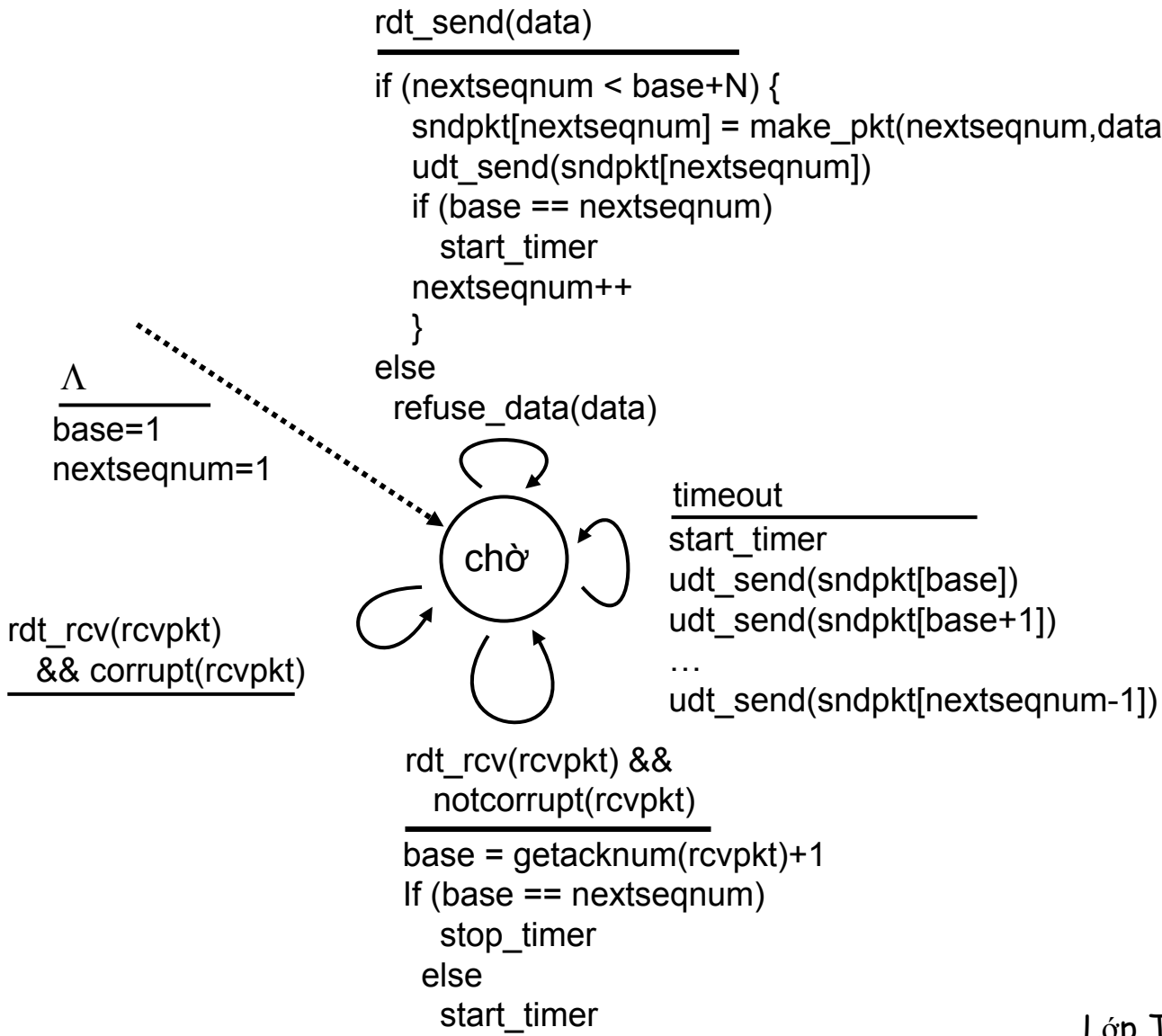
Bên gửi:

- k-bit số thứ tự trong header của gói
- "cửa sổ" tăng lên đến N, cho phép gửi các gói liên tục không cần ACK

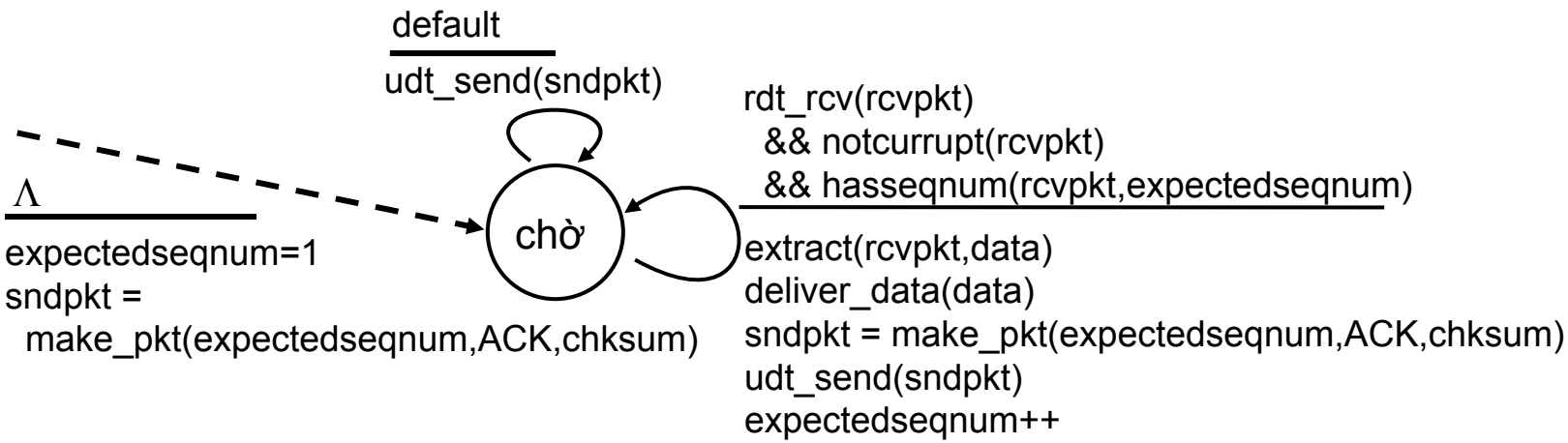


- ACK(n): ACKs tất cả các gói đến, chứa số thứ tự n – "ACK tích lũy"
  - có thể nhận các ACK trùng lặp (xem bên nhận)
- định thì cho mỗi gói trên đường truyền
- *timeout(n):* gửi lại gói n và tất cả các gói có số thứ tự cao hơn trong cửa sổ

GBN: bên gửi mở rộng FSM



GBN: bên nhận mở rộng FSM

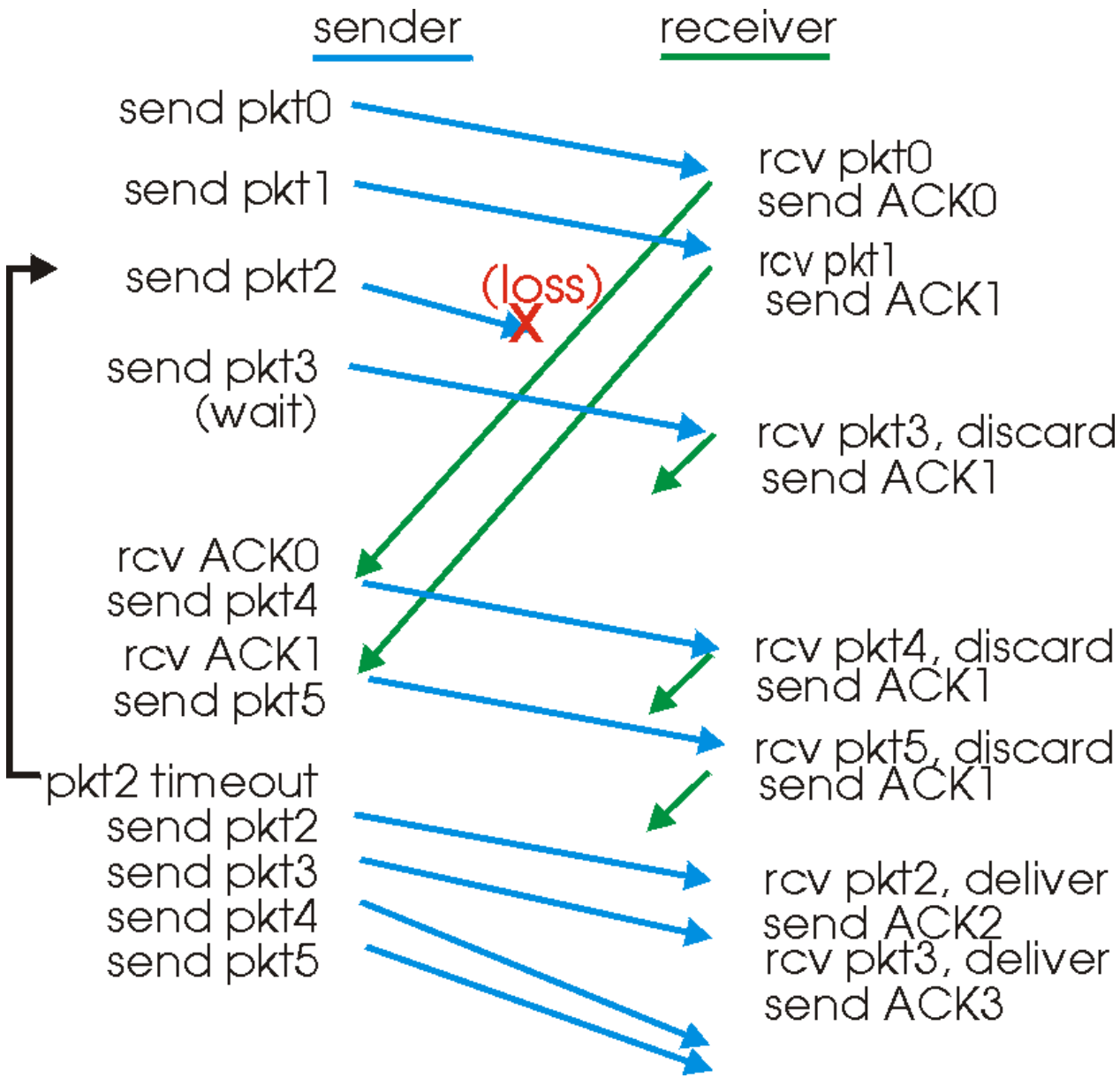


ACK-duy nhất: luôn luôn gửi ACK cho gói đã nhận đúng, với số thứ tự xếp hạng cao nhất

- có thể sinh ra các ACK trùng nhau
- chỉ cần nhớ **`expectedseqnum`**

- gói không theo thứ tự:
  - hủy -> không nhận vào bộ đệm!
  - gửi lại ACK với số thứ tự xếp hạng cao nhất

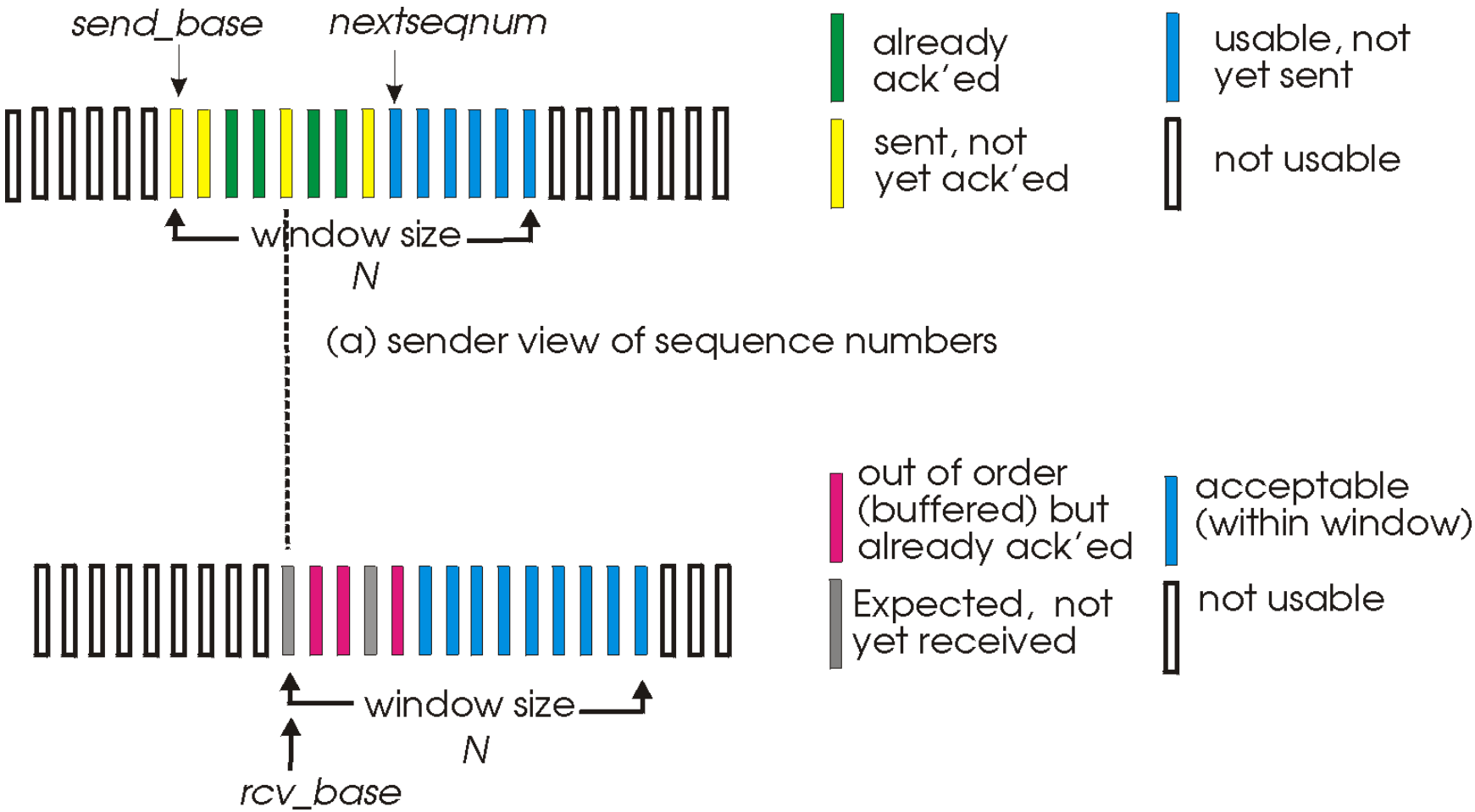
GBN hoạt động



Lặp có lựa chọn

- bên nhận thông báo đã nhận đúng tất cả từng gói một
  - đệm (buffer) các gói nếu cần thiết
- bên gửi chỉ gửi lại các gói nào không nhận được ACK
  - bên gửi định thì đối với mỗi gói không gửi ACK
- cửa sổ bên gửi
  - N số thứ tự liên tục
  - hạn chế số thứ tự các gói không gửi ACK

Lặp có lựa chọn: các cửa sổ gửi, nhận



Lặp có lựa chọn

Gửi

dữ liệu từ lớp trên:

- nếu số thứ tự kế tiếp sẵn sàng trong cửa sổ, gửi gói

timeout(n):

- gửi lại gói n, tái khởi tạo bộ định thì

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

- đánh dấu gói n là đã nhận
- nếu gói không ACK có n nhỏ nhất, dịch chuyển cửa sổ base đến số thứ tự không ACK kế tiếp

Nhận

gói n trong [rcvbase, rcvbase+N-1]

- gửi ACK(n)
- không thứ tự: đệm (buffer)
- có thứ tự: truyền (cũng truyền các gói đã đệm, có thứ tự), dịch chuyển cửa sổ đến gói chưa nhận kế tiếp

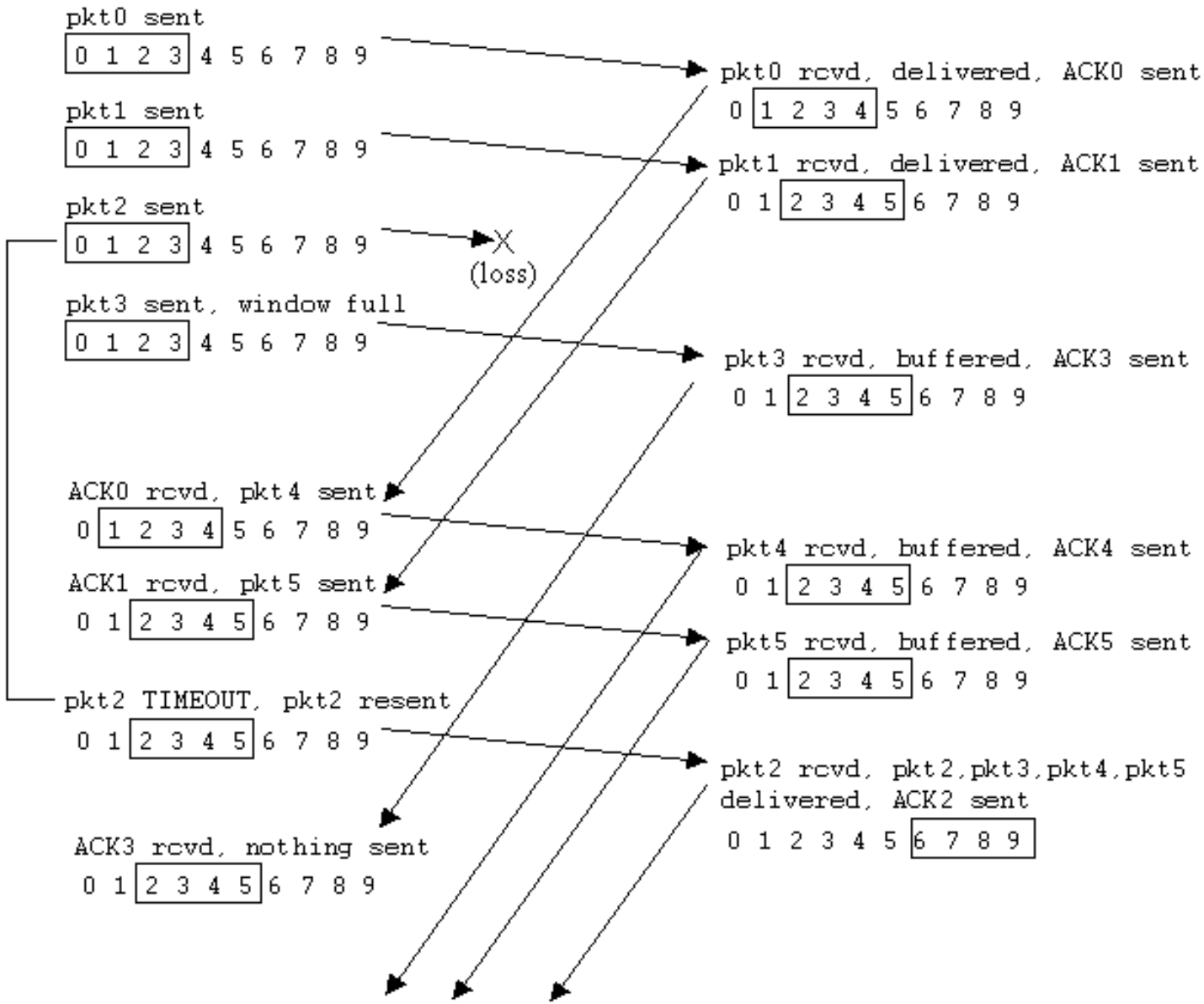
gói n trong [rcvbase-N,rcvbase-1]

- ACK(n)

ngược lại:

- lờ đi

Hoạt động của lặp có lựa chọn



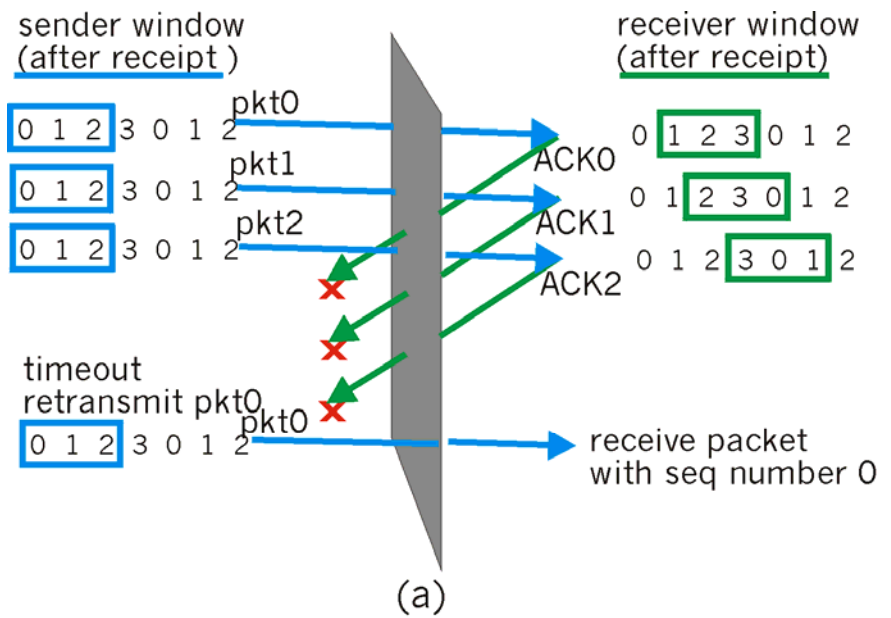
Lặp có lựa chọn: tình trạng khó giải quyết

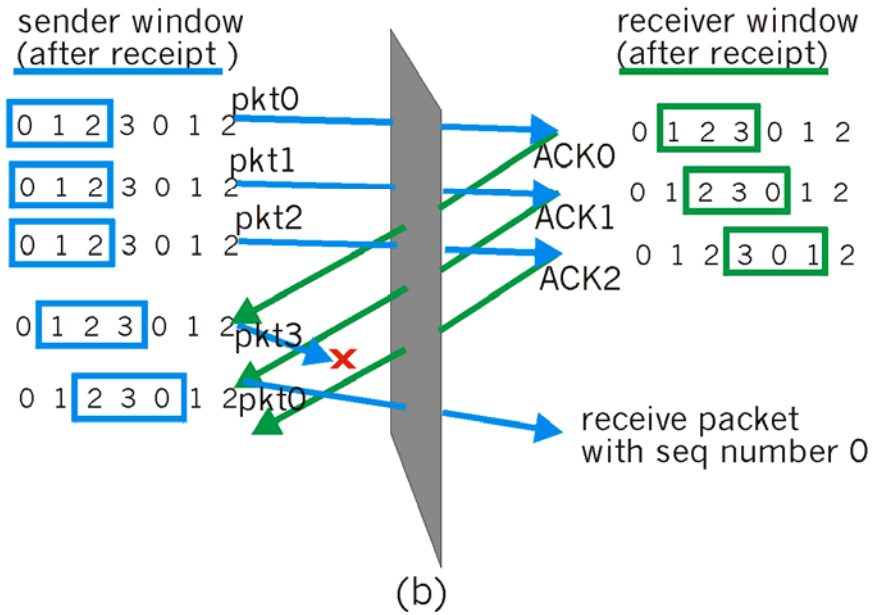
Ví dụ:

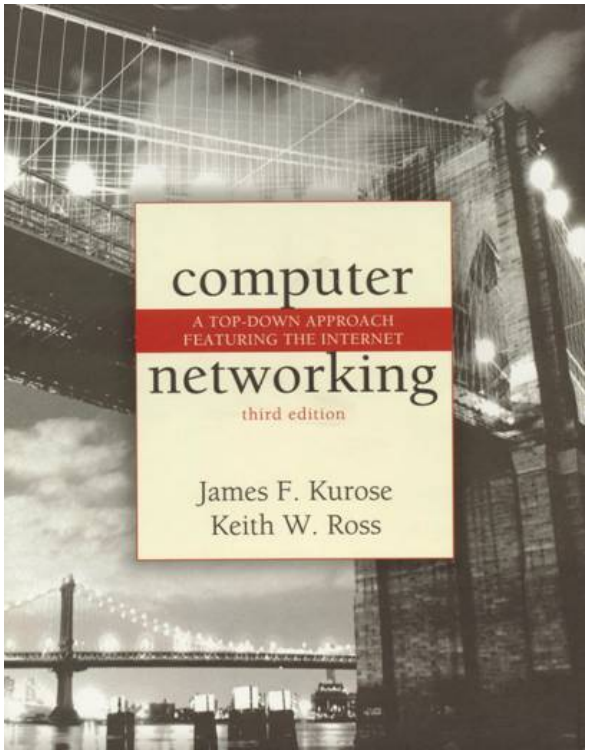
- Số thứ tự: 0, 1, 2, 3
- Kích thước cửa sổ = 3

- bên nhận không thấy sự khác nhau trong 2 tình huống
- chuyển không chính xác dữ liệu trùng lặp như dữ liệu mới trong (a)

Hỏi: quan hệ giữa dãy số thứ tự và kích thước cửa sổ?



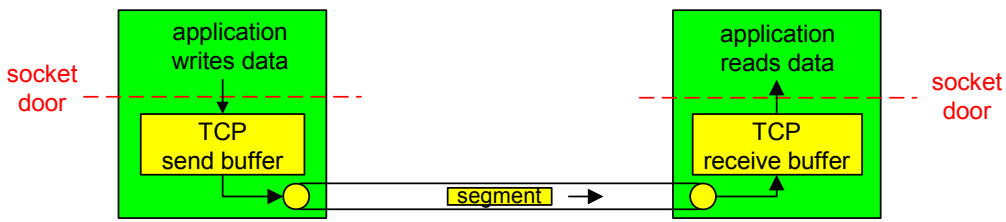




3.5 Vận chuyển hướng kết nối: TCP

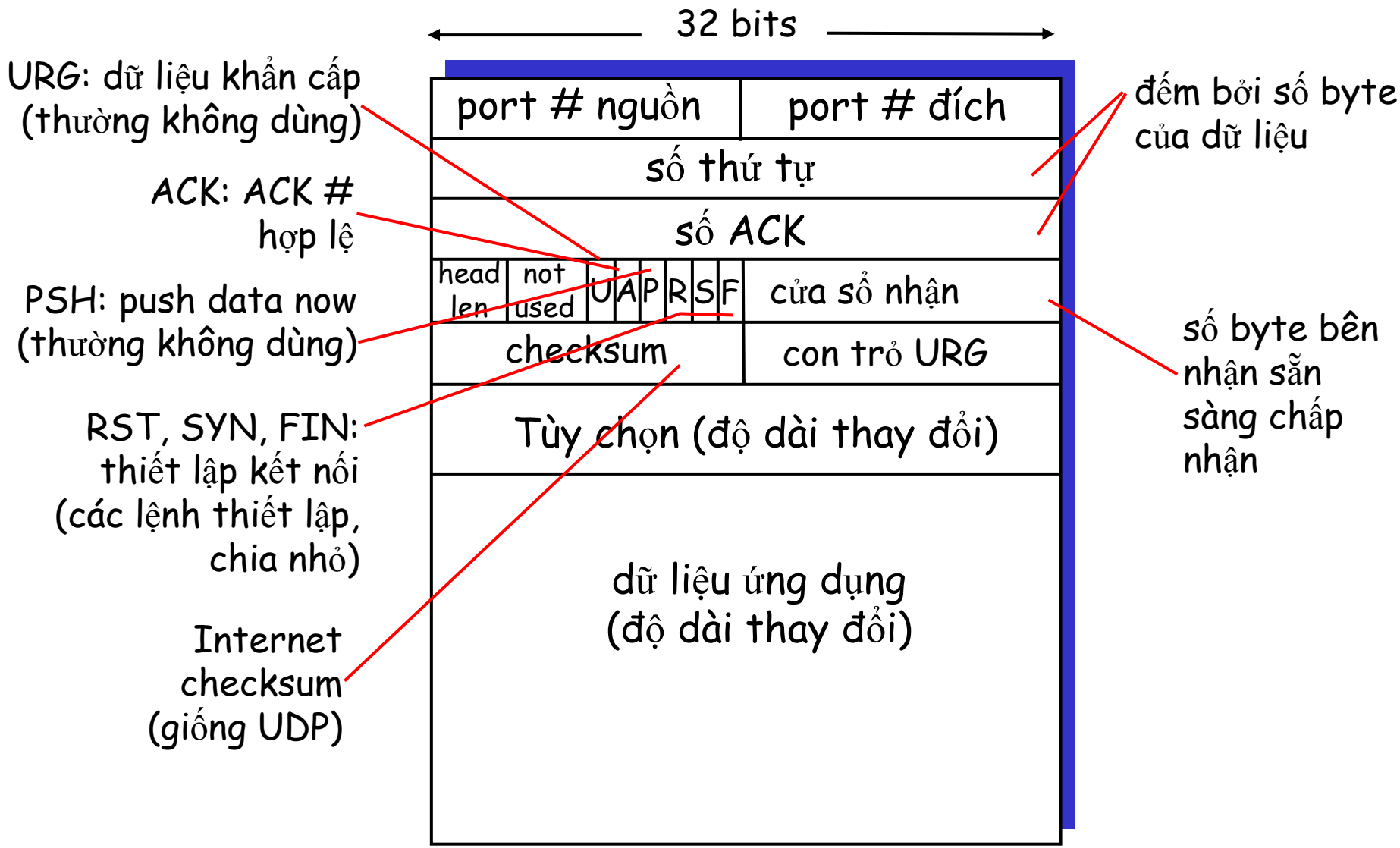
TCP: Tổng quan RFCs: 793, 1122, 1323, 2018, 2581

- point-to-point:
  - một bên gửi, một bên nhận
- tin cậy, *dòng byte có thứ tự:*
  - không "ranh giới thông điệp"
- kênh liên lạc:
  - TCP điều khiển luồng và tắc nghẽn, thiết lập kích thước cửa sổ
- *các bộ đệm gửi & nhận*



- dữ liệu full duplex:
  - luồng dữ liệu đi 2 chiều trong cùng một kết nối
  - MSS: maximum segment size – kích thước đoạn tối đa
- hướng kết nối:
  - bắt tay (trao đổi các thông điệp điều khiển) trạng thái bên gửi, bên nhận trước khi trao đổi dữ liệu
- điều khiển luồng:
  - bên gửi sẽ không lấn át bên nhận

TCP: cấu trúc đoạn



Các số thứ tự TCP và ACK

Các số thứ tự:

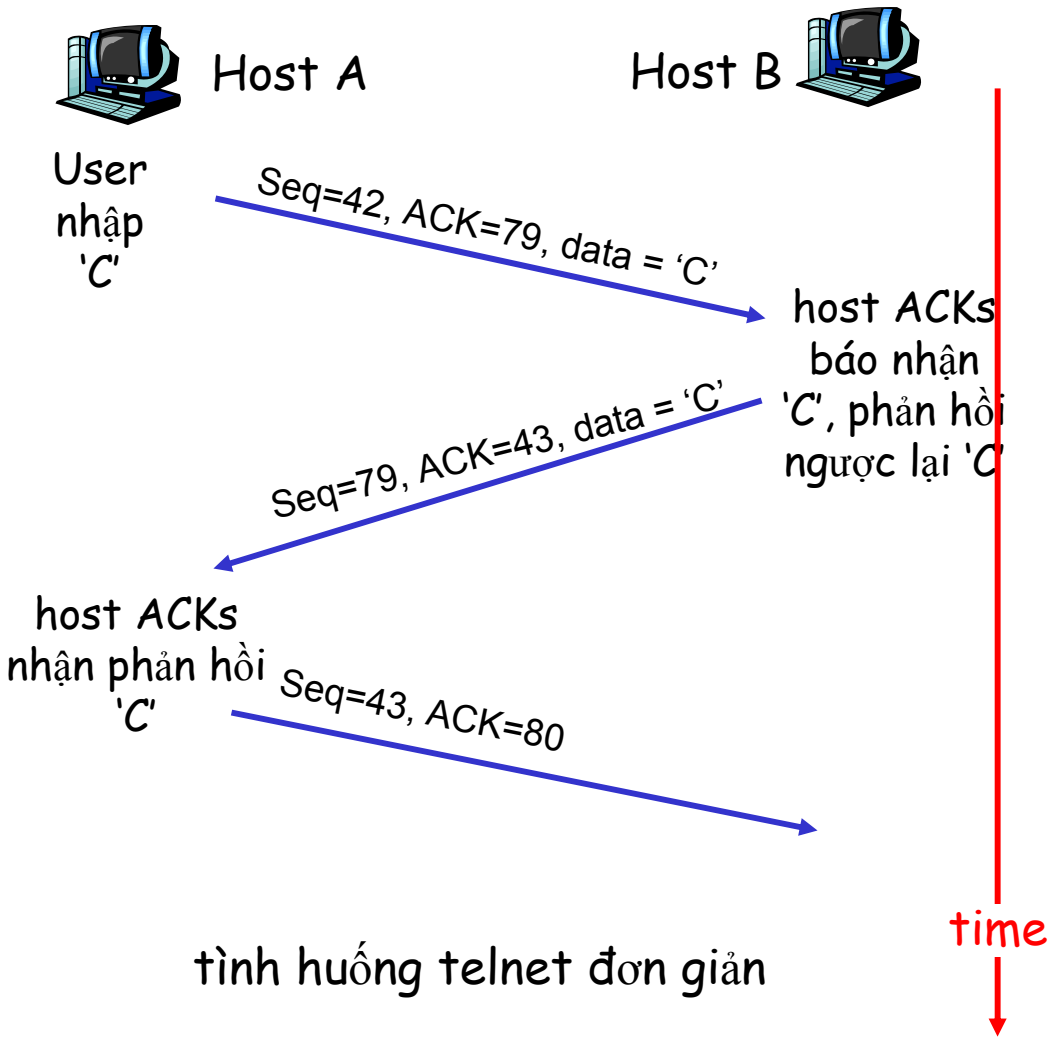
- dòng byte "đánh số" byte đầu tiên trong dữ liệu của đoạn

các ACK:

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

Hỏi: làm thế nào bên nhận quản lý các đoạn không thứ tự

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



tình huống telnet đơn giản

TCP Round Trip Time và Timeout

Hỏi: Làm thế nào để thiết lập giá trị TCP timeout?

- dài hơn RTT
  - khác với RTT
- quá ngắn: timeout sớm
  - truyền lại không cần thiết
- quá dài: phản ứng chậm đối với việc mất mát gói

Hỏi : Làm thế nào để thiết lập RTT?

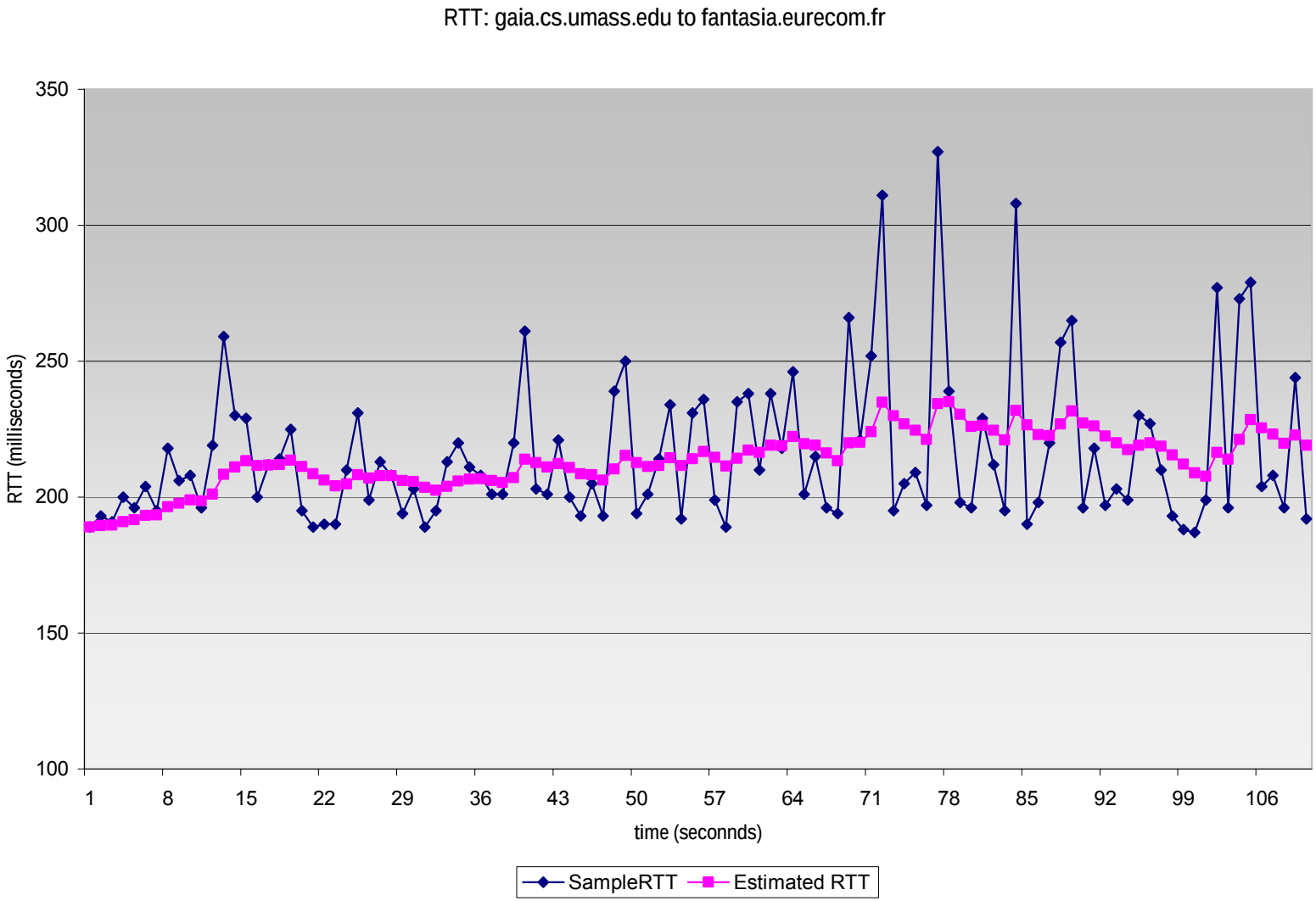
- **SampleRTT**: thời gian được đo từ khi truyền đoạn đến khi báo nhận ACK
  - lờ đi việc truyền lại
- **SampleRTT** sẽ thay đổi, muốn ước lượng RTT "mượt hơn"
  - tính trung bình một số giá trị đo được gần đó, không chỉ **SampleRTT** hiện tại

TCP Round Trip Time và Timeout

EstimatedRTT = (α)*EstimatedRTT + α*SampleRTT

- giá trị đặc trưng: α = 0.125

Ví dụ đánh giá RTT:



TCP Round Trip Time và Timeout

Thiết lập timeout

- **EstimtedRTT** cộng "hệ số dự trữ an toàn"
  - sự biến thiên lớn trong **EstimatedRTT** -> hệ số dự trữ an toàn lớn hơn
- ước lượng đầu tiên về sự biến thiên của SampleRTT từ EstimatedRTT :

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

(tiêu biểu β = 0.25)

Sau đó thiết lập timeout interval:

TimeoutInterval = EstimatedRTT + 4*DevRTT

TCP: truyền dữ liệu tin cậy

- TCP tạo dịch vụ rdt trên dịch vụ không tin cậy IP
- các đoạn Pipelined
- các ACK tích lũy
- TCP dùng bộ định thì truyền lại đơn
- Truyền lại được kích hoạt bởi:
  - các sự kiện timeout
  - các ack trùng lặp
- lúc đầu khảo sát các bên gửi TCP đơn giản:
  - lờ đi các ack trùng lặp
  - lờ đi điều khiển luồng, điều khiển tắc nghẽn

TCP các sự kiện:

dữ liệu đã nhận từ ứng dụng:

- tạo đoạn với số thứ tự của nó
- số thứ tự là số theo byte dữ liệu đầu tiên
- khởi động bộ định thì nếu chưa chạy
- khoảng thời gian hết hạn: `TimeOutInterval`

timeout:

- gửi lại đoạn nào gây ra timeout
- khởi động lại bộ định thì

Ack đã nhận:

- cập nhật cái gì sẽ được ACK
- khởi động bộ định thì nếu có các đoạn còn chờ

```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

loop (forever) {
  switch(event)

  event: data received from application above
      create TCP segment with sequence number NextSeqNum
      if (timer currently not running)
          start timer
      pass segment to IP
      NextSeqNum = NextSeqNum + length(data)

  event: timer timeout
      retransmit not-yet-acknowledged segment with
          smallest sequence number
      start timer

  event: ACK received, with ACK field value of y
      if (y > SendBase) {
          SendBase = y
          if (there are currently not-yet-acknowledged segments)
              start timer
          }

} /* end of loop forever */
```

TCP bên gửi (đơn giản)

Chú thích:

- SendBase-1: byte vừa được ACK tích lũy

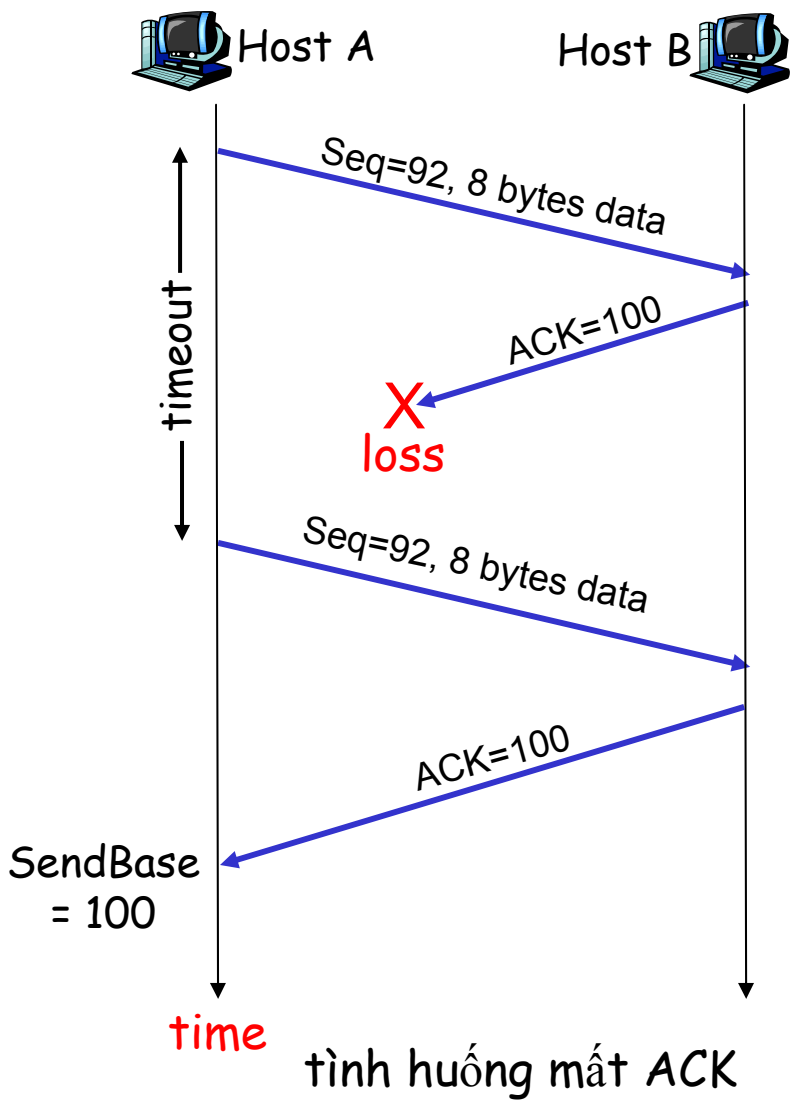
Ví dụ:

- SendBase-1 = 71;

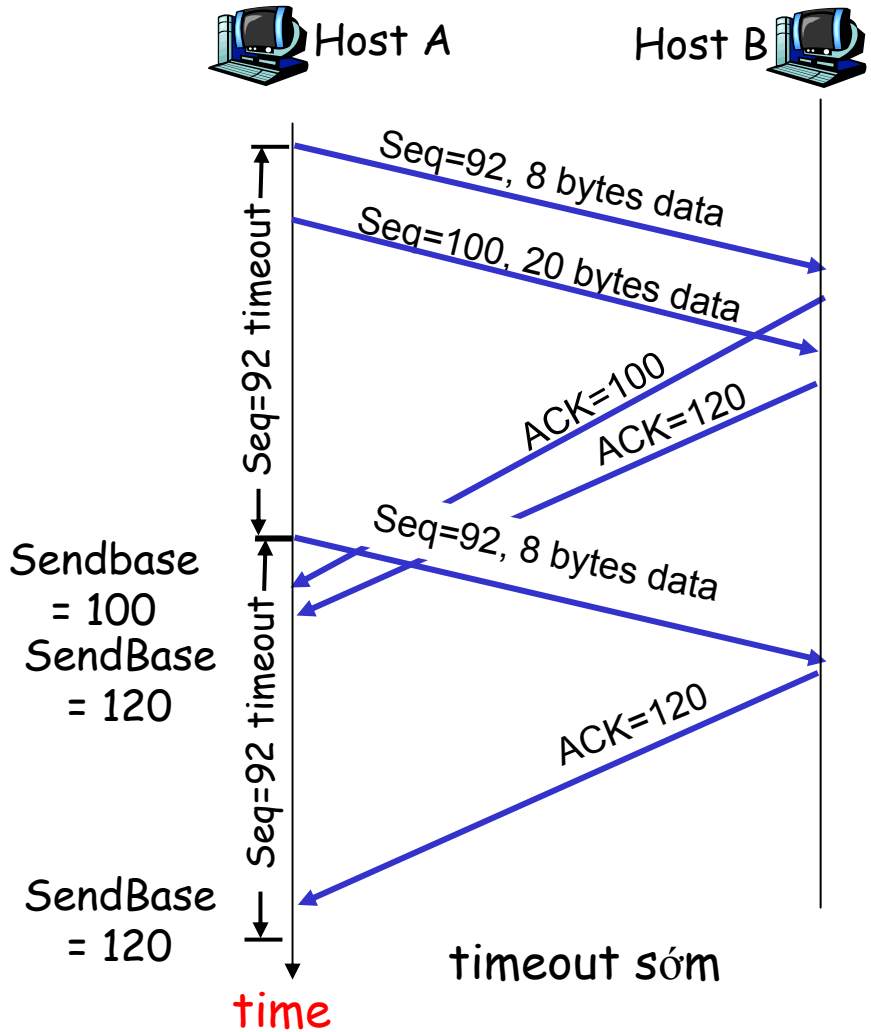
y= 73, vì thế bên nhận muốn 73+ ;

y > SendBase, vì thế dữ liệu mới được chấp nhận

TCP: các tình huống truyền lại

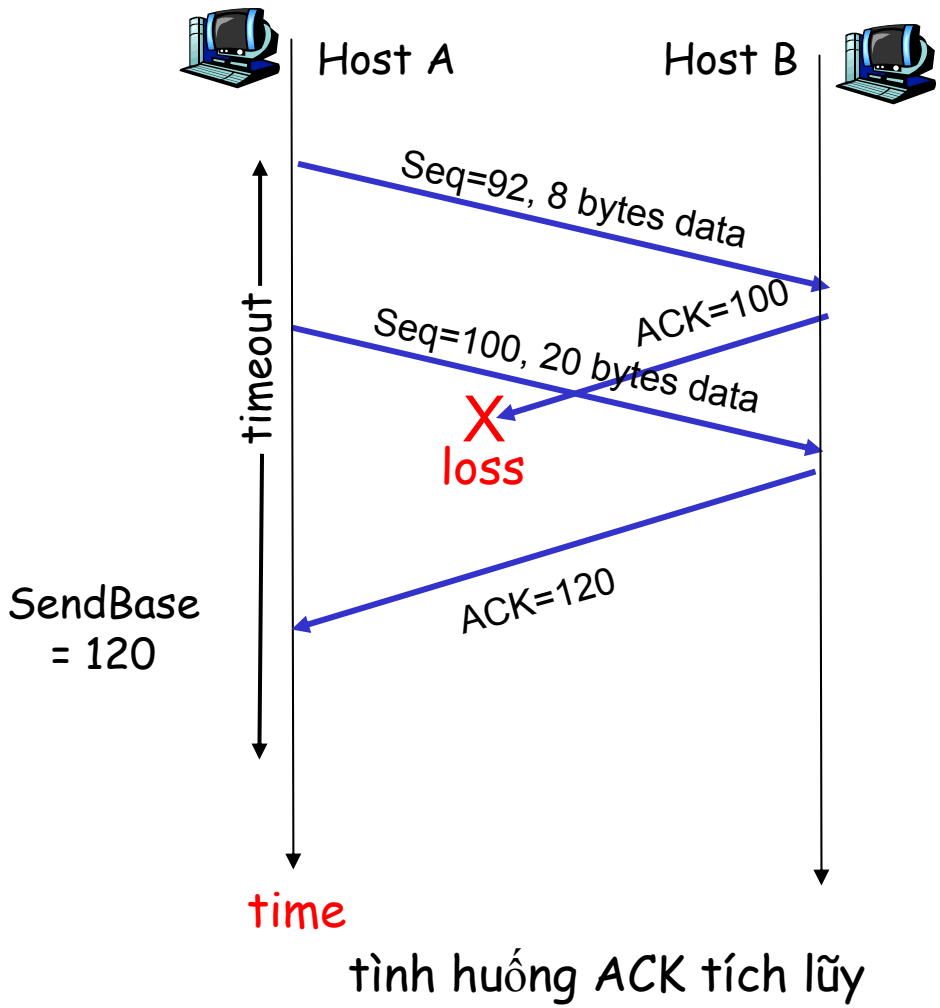


tình huống mất ACK



timeout sớm

TCP: các tình huống truyền lại (tt)



tình huống ACK tích lũy

TCP sinh ra ACK [RFC 1122, RFC 2581]

Sự kiện tại bên nhận	TCP bên nhận hành động
Đoạn đến với đúng số thứ tự mong muốn. Tất cả dữ liệu đến đã được ACK	ACK trễ. Chờ đến 500ms cho đoạn kế tiếp. Nếu không có đoạn kế tiếp, gửi ACK
Đoạn đến với đúng số thứ tự mong muốn. Một đoạn khác đang chờ ACK	Gửi ngay một ACK tích lũy, chấp nhận cho cả các đoạn theo thứ tự
Các đoạn đến không thứ tự lớn hơn số thứ tự đoạn mong muốn. Có khoảng trống	Gửi ngay *ACK trùng lặp*, chỉ thị số thứ tự đoạn của byte kế tiếp đang mong chờ
Đoạn đến lấp đầy từng phần hoặc toàn bộ khoảng trống	Gửi ngay ACK, với điều kiện là đoạn bắt đầu ngay điểm có khoảng trống

Truyền lại nhanh

- Chu kỳ Time-out thường tương đối dài:
  - độ trễ dài trước khi gửi lại gói đã mất
- Xác nhận các đoạn đã mất bằng các ACK trùng lặp.
  - bên gửi thường gửi nhiều đoạn song song
  - Nếu đoạn bị mất, sẽ xảy ra tình trạng giống như nhiều ACK trùng nhau
- Nếu bên gửi nhận 3 ACK của cùng một dữ liệu, nó cho là đoạn sau dữ liệu đã ACK bị mất:
  - Truyền lại nhanh: gửi lại đoạn trước khi bộ định thì hết hạn

Giải thuật truyền lại nhanh:

```
sự kiện: ACK đã nhận, với trường là y
            if (y > SendBase) {
                SendBase = y
                if (there are currently not-yet-acknowledged segments)
                    start timer
              }
            else {
                increment count of dup ACKs received for y
                if (count of dup ACKs received for y = 3) {
                    resend segment with sequence number y
                }
```

một ACK trùng lặp cho đoạn đã được ACK

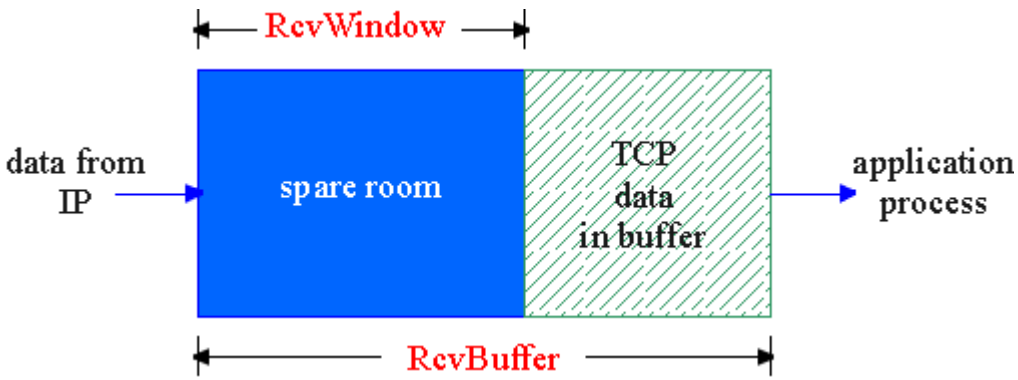
Truyền lại nhanh

TCP điều khiển luồng

điều khiển luồng

bên gửi sẽ không làm tràn bộ đệm vì truyền quá nhiều và quá nhanh

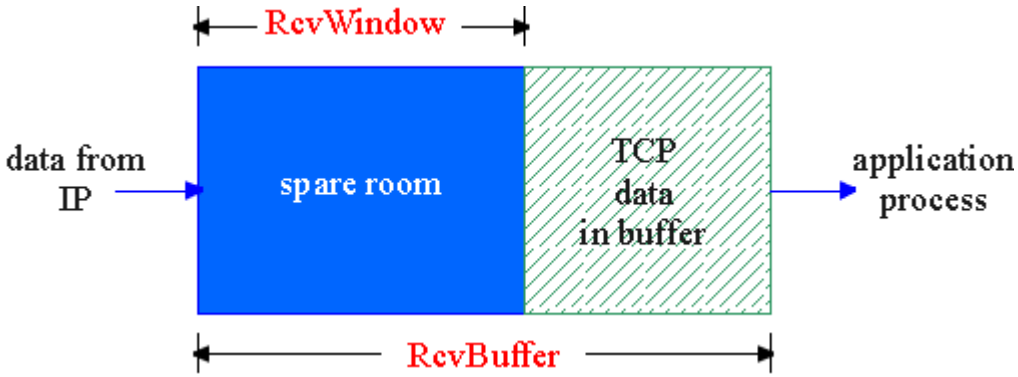
- bên nhận của kết nối TCP có một bộ đệm nhận:



- tiến trình ứng dụng có thể chậm tại lúc đọc bộ đệm

- dịch vụ so trùng tốc độ: so trùng tốc độ gửi với tốc độ nhận của ứng dụng

TCP điều khiển luồng: cách làm?



(Giả sử bên nhận TCP loại bỏ các đoạn không có thứ tự)

- dự phòng trong bộ đệm

= **RcvWindow**

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

- Bên nhận thông báo khoảng dự trữ nhờ giá trị **RcvWindow** trong các đoạn
- Bên gửi hạn chế dữ liệu không được ACK vào **RcvWindow**
  - bảo đảm bộ đệm nhận không tràn

TCP quản lý kết nối

Chú ý: Bên gửi và bên nhận TCP thiết lập "kết nối" trước khi trao đổi dữ liệu

- khởi tạo các biến TCP:
  - các số thứ tự đoạn
  - thông tin các bộ đệm, điều khiển luồng (như **RcvWindow**)
- *client:* người khởi xướng kết nối

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

- *server:* được tiếp xúc bởi client

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

3 phương pháp bắt tay:

Bước 1: client host gửi đoạnTCP SYN đến server
- xác định số thứ tự khởi đầu
- không phải dữ liệu

Bước 2: server host nhận SYN, trả lời với đoạn SYNACK
- server cấp phát các bộ đệm
- xác định số thứ tự khởi đầu

Bước 3: client nhận SYNACK, trả lời với đoạn ACK (có thể chứa dữ liệu)

TCP quản lý kết nối (tt)

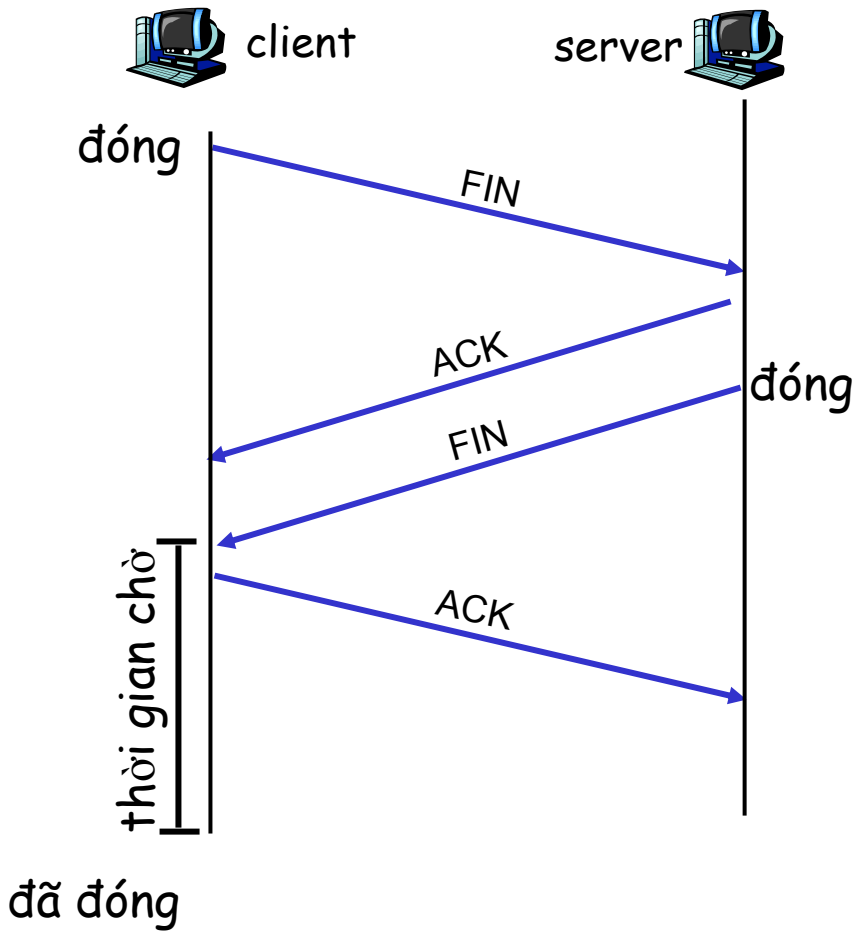
Đóng một kết nối:

client đóng socket:

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

Bước 1: client gửi đoạn điều khiển TCP FIN đến server

Bước 2: server nhận FIN, trả lời với ACK. Đóng kết nối, gửi FIN.



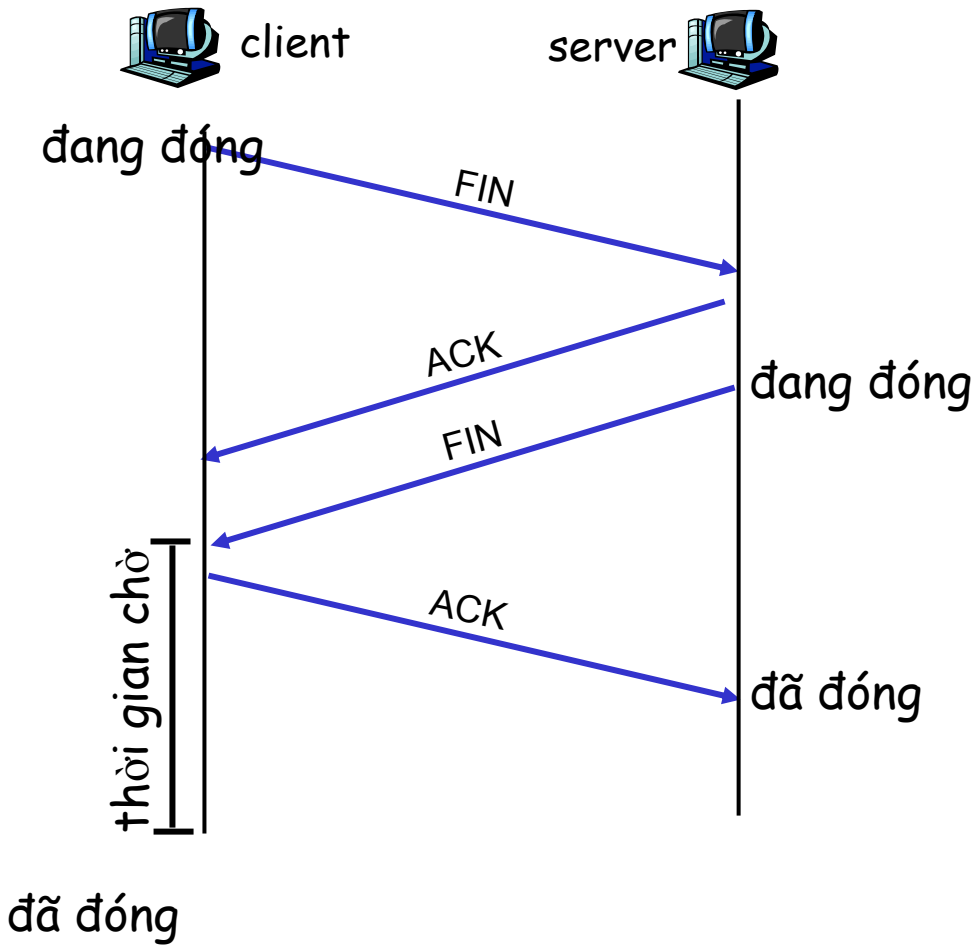
TCP quản lý kết nối (tt)

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

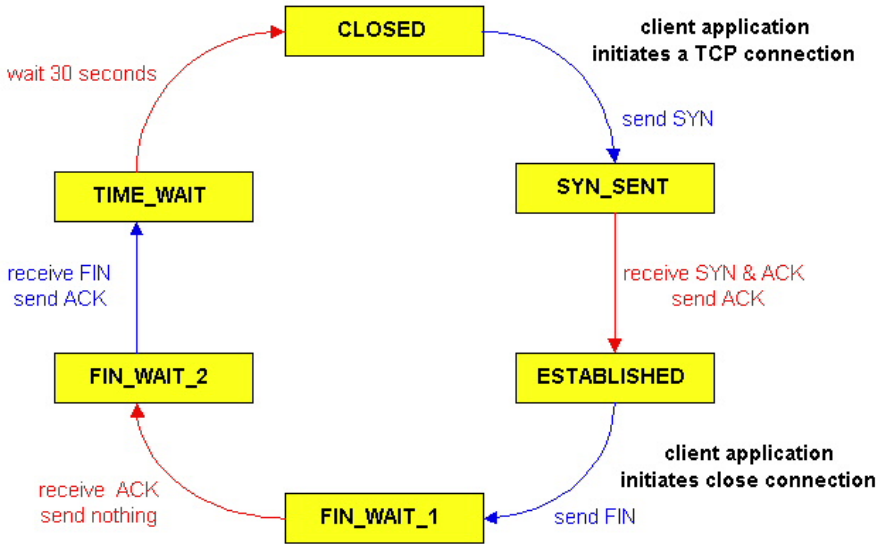
- Trong khoảng "thời gian chờ" – sẽ phản hồi với ACK để nhận các FIN

Bước 4: server, nhận ACK. Kết nối đã đóng.

Chú ý: với một sửa đổi nhỏ, có thể quản lý nhiều FIN đồng thời.

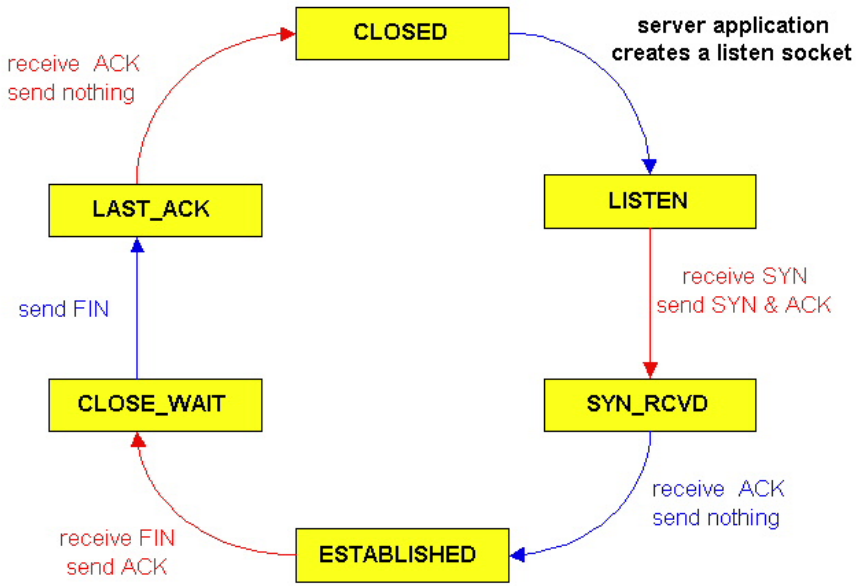


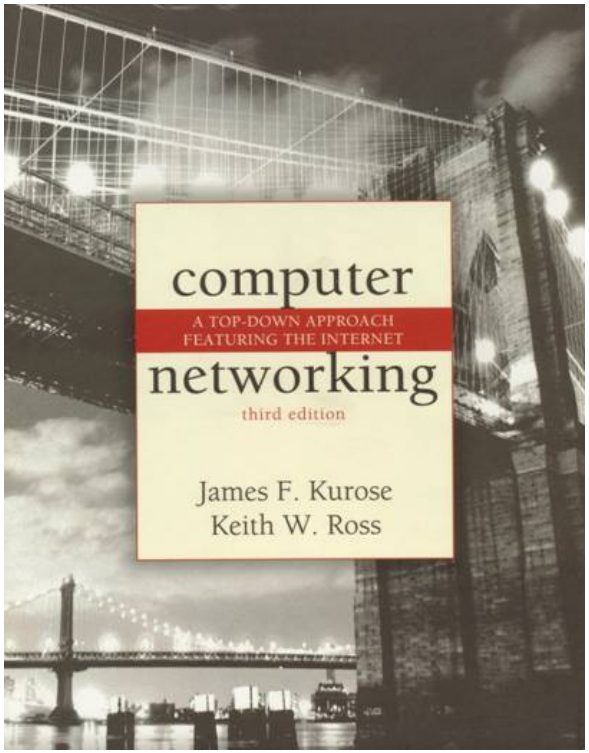
TCP quản lý kết nối (tt)



chu kỳ sống của
TCP client

chu kỳ sống của
TCP server





3.6 Các nguyên lý của điều khiển tắc nghẽn

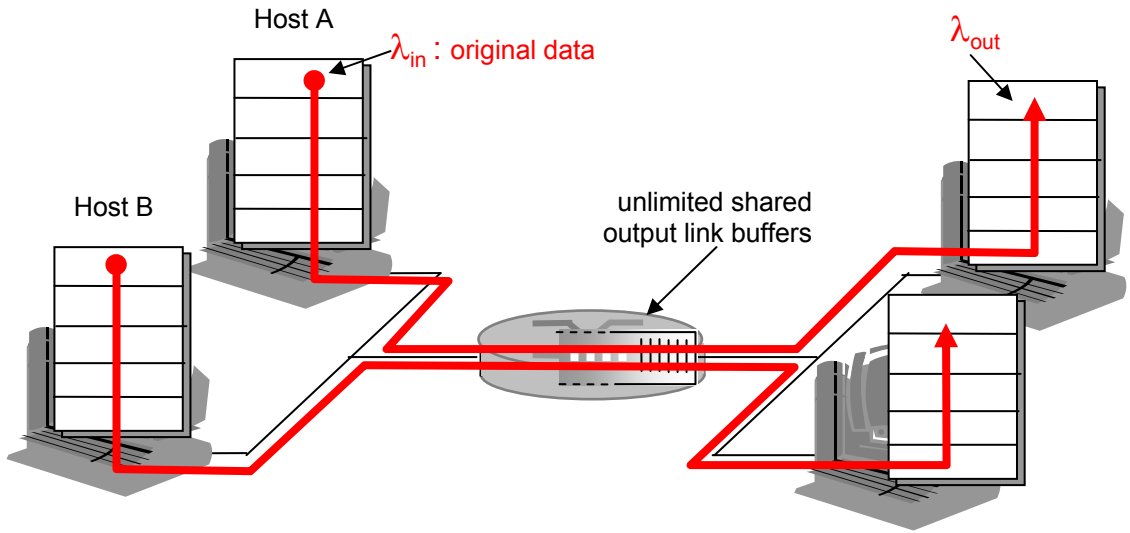
Các nguyên lý điều khiển tắc nghẽn

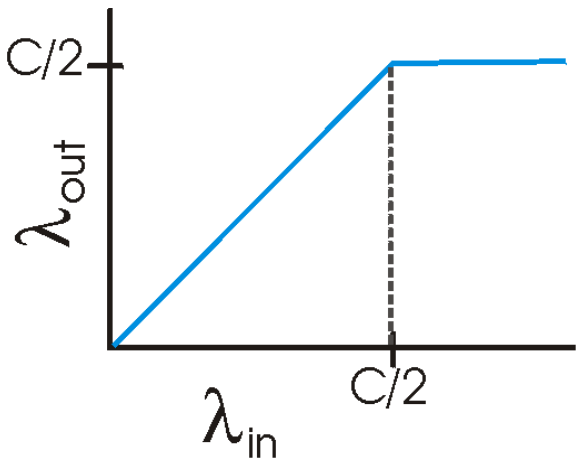
Tắc nghẽn:

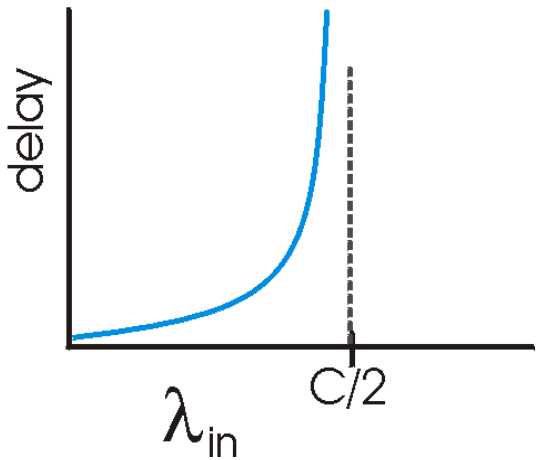
- "quá nhiều nguồn gửi quá nhanh và quá nhiều dữ liệu đến *mạng*"
- khác với điều khiển luồng!
- các biểu hiện:
  - các gói bị mất (tràn bộ đệm tại các router)
  - các độ trễ quá dài (xếp hàng trong bộ đệm của router)
- là 1 trong 10 vấn đề nan giải nhất!

Các nguyên nhân/chi phí của tắc nghẽn: tình huống 1

- 2 gửi, 2 nhận
- 1 router, các bộ đệm không giới hạn
- không có truyền lại



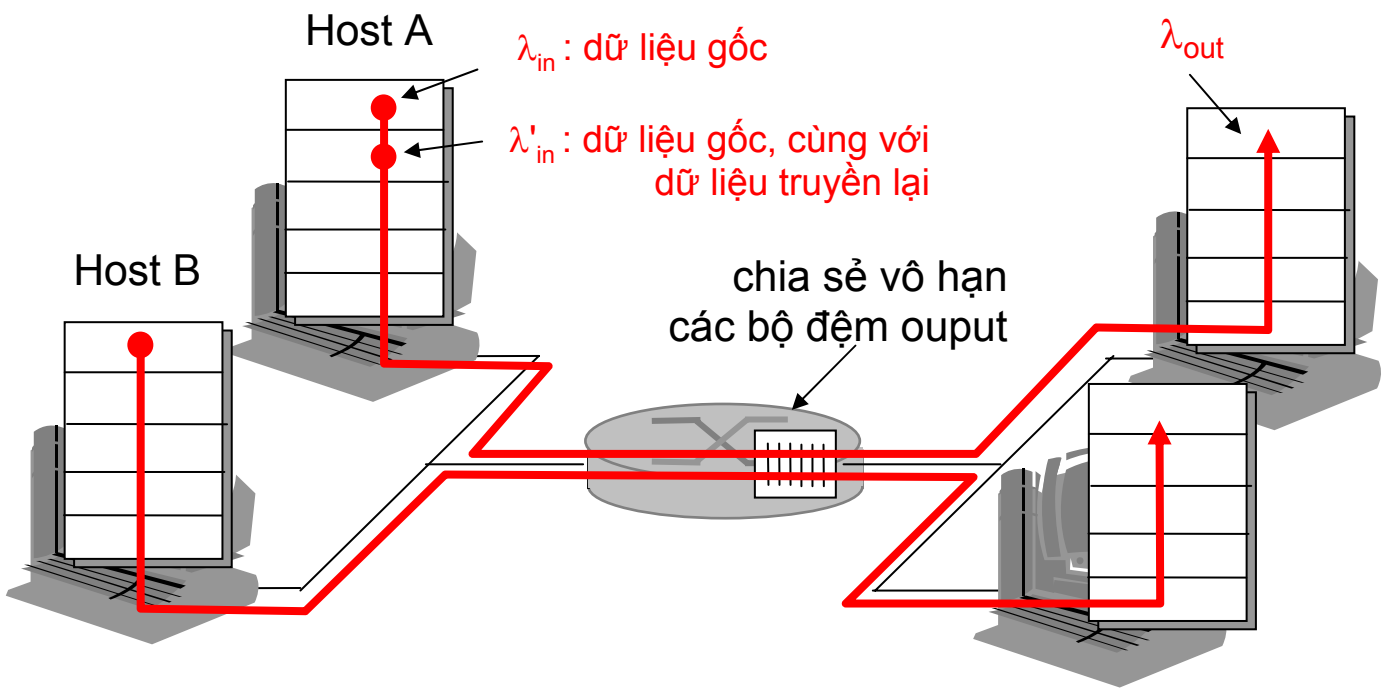




- các độ trễ lớn hơn khi tắc nghẽn
- năng suất có thể đạt tối đa

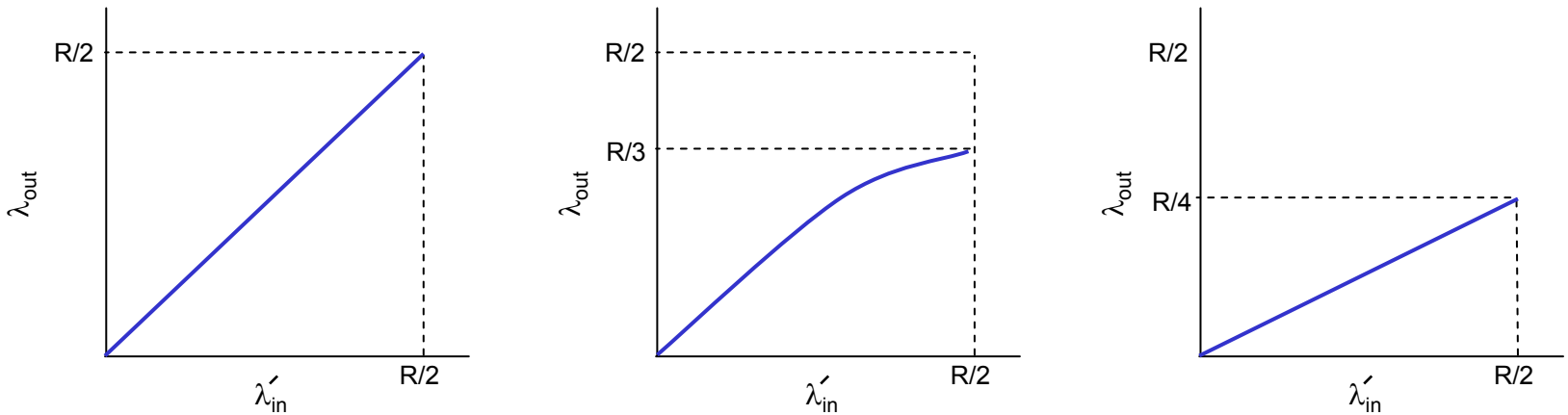
Các nguyên nhân/chi phí của tắc nghẽn: tình huống 2

- 1 router, các bộ đệm *có giới hạn*
- bên gửi truyền lại các gói đã mất



Các nguyên nhân/chi phí của tắc nghẽn: tình huống 2

- luôn luôn: $\lambda_{in} = \lambda_{out}$
- truyền lại "hoàn toàn" chỉ khi mất mát: $\lambda'_{in} > \lambda_{out}$
- truyền lại vì trễ (không mất) làm cho λ'_{in} lớn hơn với cùng λ_{out}



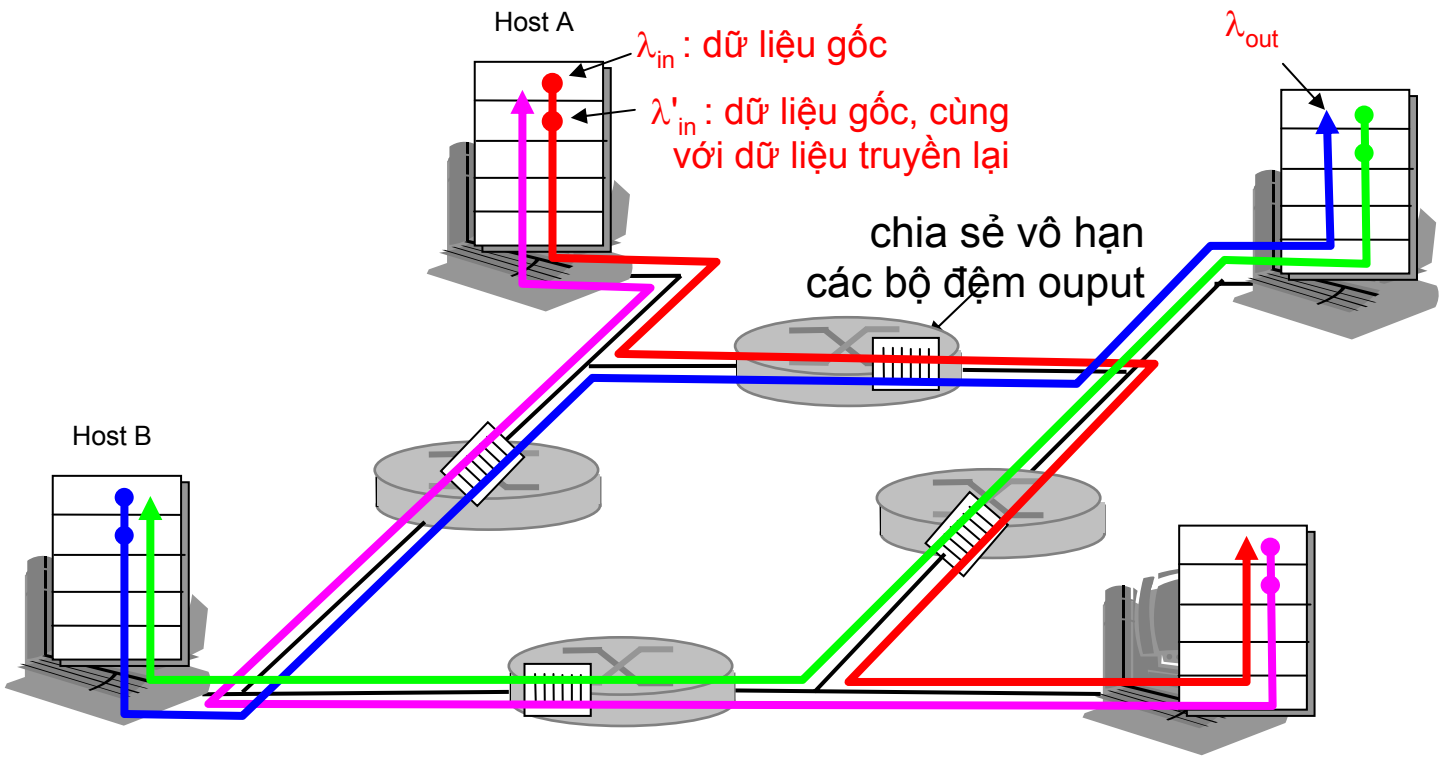
"các chi phí" của tắc nghẽn:

- nhiều việc (truyền lại)
- các truyền lại không cần thiết: liên kết nhiều bản sao của gói

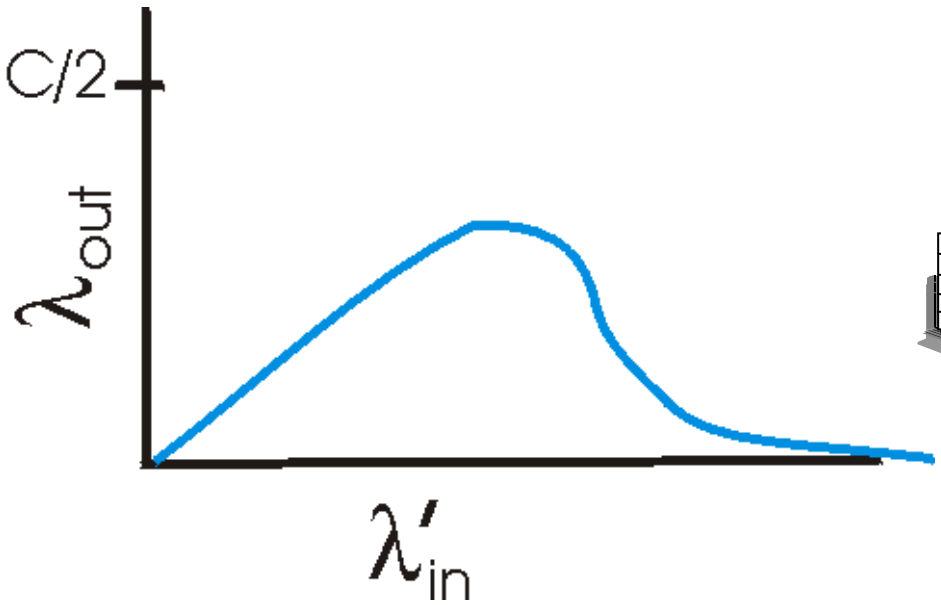
Các nguyên nhân/chi phí của tắc nghẽn: tình huống 3

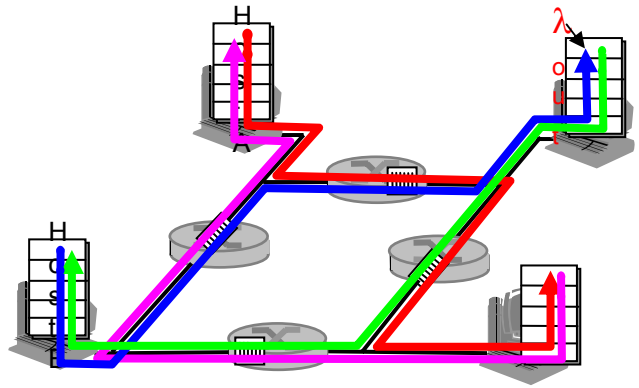
- 4 người gửi
- các đường qua nhiều hop
- timeout/truyền lại

Hỏi: điều gì xảy ra nếu λ_{in} và λ'_{in} tăng lên?



Các nguyên nhân/chi phí của tắc nghẽn: tình huống 3





"chi phí" khác của tắc nghẽn:

- khi các gói bị bỏ, bất kỳ "khả năng truyền upstream dùng cho gói đó sẽ bị lãng phí!"

Các cách tiếp cận đối với điều khiển tắc nghẽn

2 cách:

điều khiển tắc nghẽn end-end:

- không có phản hồi rõ ràng từ mạng
- tắc nghẽn được suy ra từ việc quan sát các hệ thống đầu cuối có mất mát, trễ
- tiếp cận được quản lý bởi TCP

điều khiển tắc nghẽn có sự hỗ trợ của mạng:

- các router cung cấp phản hồi về các hệ thống đầu cuối
  - 1 bit duy nhất chỉ thị tắc nghẽn (SNA, DECbit, TCP/IP ECN, ATM)
  - tốc độ sẽ gửi được xác định rõ ràng

Ví dụ: điều khiển tắc nghẽn ATM ABR

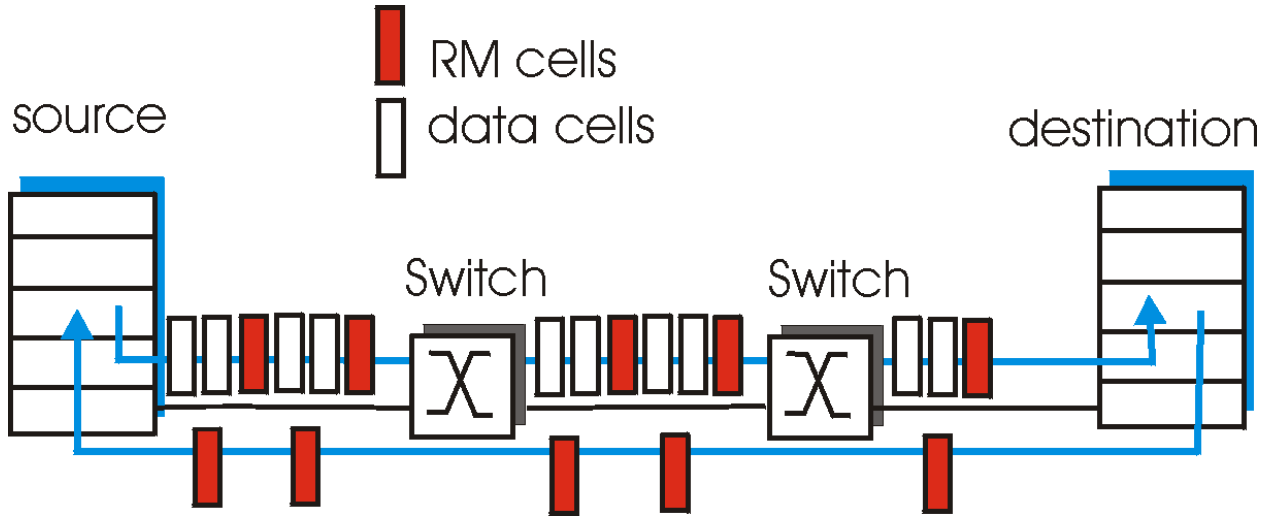
ABR: tốc độ bit sẵn sàng:

- "dịch vụ mềm dẻo"
- nếu đường gửi "chưa hết":
  - bên gửi sẽ dùng băng thông sẵn sàng
- nếu đường gửi tắc nghẽn:
  - bên gửi điều tiết với tốc độ tối thiểu

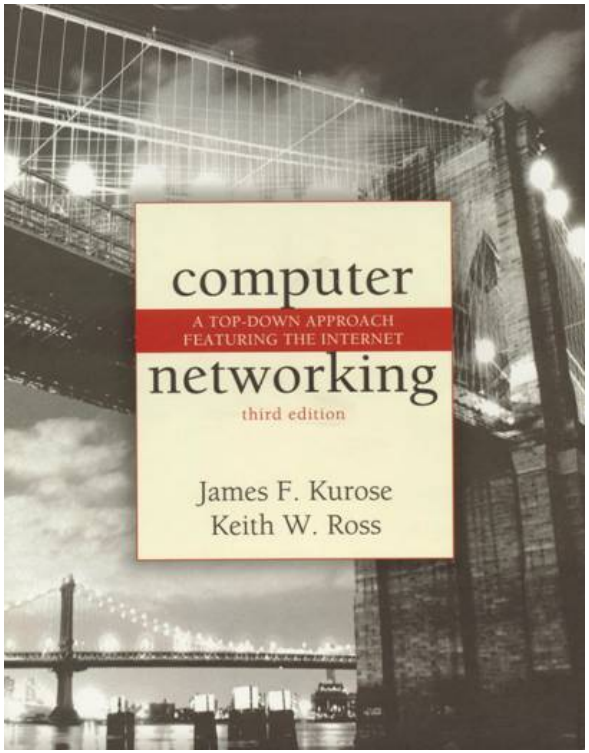
RM (resource management):

- gửi bởi bên gửi, rải rác với các ô dữ liệu
- các bit trong ô thiết lập bởi các switch
  - bit NI : không tăng tốc độ (tắc nghẽn nhẹ)
  - bit CI : tắc nghẽn rõ rệt
- Các ô RM được trả về bên gửi từ bên nhận với nguyên vẹn các bit

Ví dụ: điều khiển tắc nghẽn ATM ABR



- trường 2-byte ER trong ô RM
  - switch đã tắc nghẽn có thể có giá trị ER thấp hơn trong ô
  - tốc độ gửi do đó có thể được hỗ trợ tối đa trên đường
- EFCI bit trong các ô dữ liệu: được cài giá trị 1 trong switch đã tắc nghẽn
  - nếu ô dữ liệu đứng trước ô RM có cài EFCI, bên gửi sẽ cài bit CI trong ô RM trả về

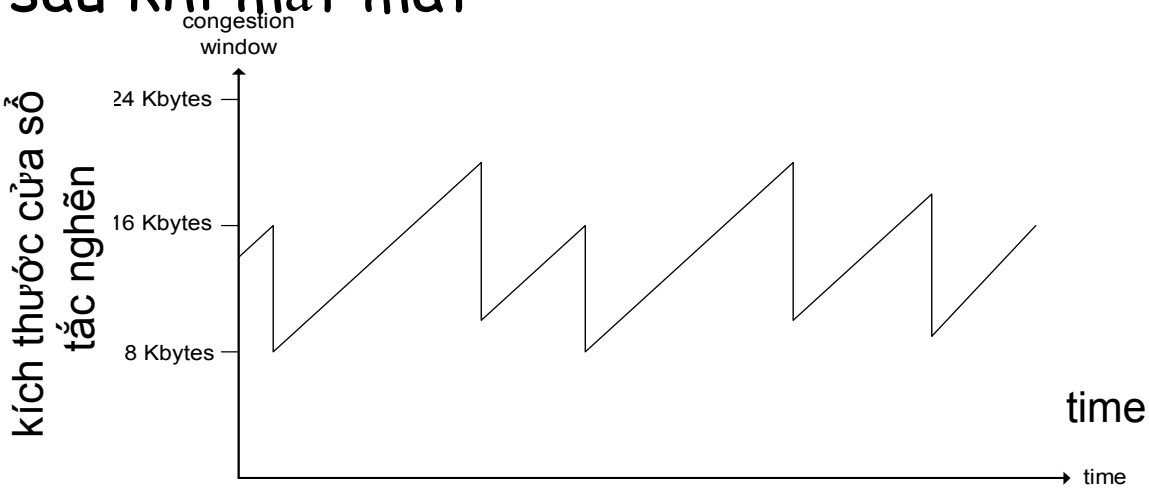


3.7 Điều khiển tắc nghẽn TCP

TCP điều khiển tắc nghẽn: additive tăng lên, multiplicative giảm xuống

- *Cách tiếp cận:* tăng tốc độ truyền (kích thước cửa sổ), tìm khả năng băng thông có thể, cho đến khi có mất mát xảy ra
  - *additive tăng lên:* tăng **CongWin** bởi 1 MSS mỗi RTT cho đến khi có mất mát xảy ra
  - *multiplicative giảm xuống:* bỏ **CongWin** trong nửa giai đoạn sau khi mất mát

Tìm kiếm băng thông



TCP điều khiển tắc nghẽn: chi tiết

- bên gửi hạn chế việc truyền:

  LastByteSent-LastByteAcked

  ≤ CongWin

- Công thức,

$$\text{tốc độ} = \frac{\text{CongWin}}{\text{RTT}} \text{ Bytes/s}$$

- **CongWin** thay đổi, chức năng là nhận biết tắc nghẽn trên mạng

Làm thế nào bên gửi nhận biết tắc nghẽn?

- mất mát xảy ra = timeout *hoặc* 3 ack trùng lặp
- bên gửi giảm tốc độ (**CongWin**) sau khi mất mát xảy ra

3 cơ chế:

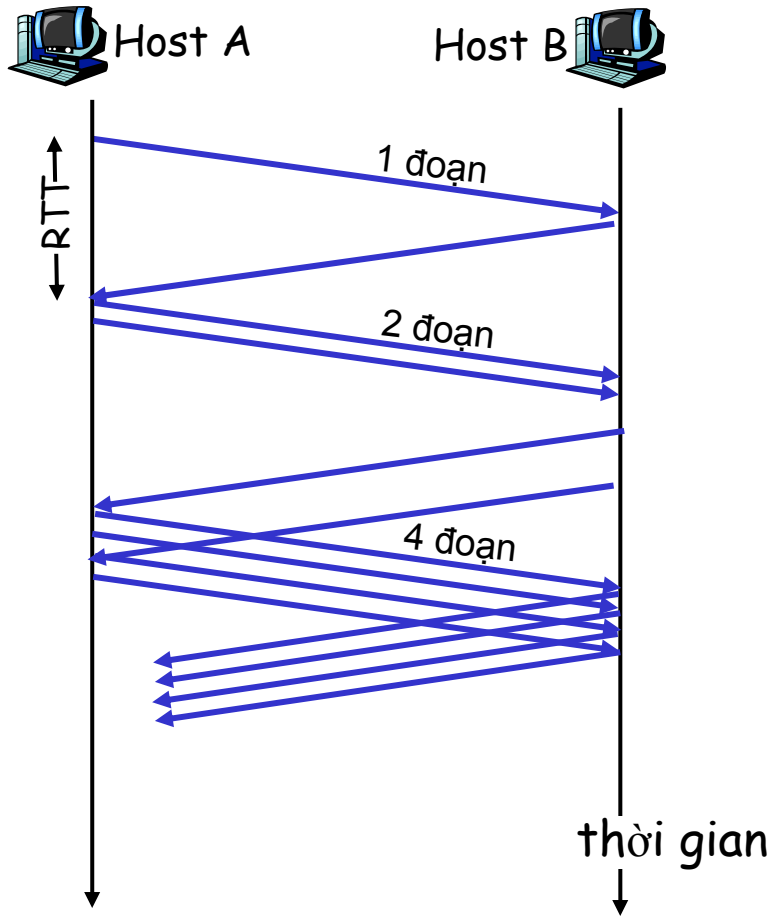
- AIMD
- khởi động chậm
- thận trọng sau khi có các sự kiện timeout

TCP khởi động chậm

- Khi kết nối bắt đầu, **CongWin** = 1 MSS
  - Ví dụ: MSS = 500 bytes & RTT = 200 ms
  - tốc độ khởi tạo = 20 kbps
- băng thông sẵn sàng có thể >> MSS/RTT
  - mong muốn nhanh chóng tăng tốc lên tốc độ có thể đáp ứng
- Khi kết nối bắt đầu, tăng tốc lên rất nhanh cho đến khi sự cố mất mát xảy ra đầu tiên

TCP khởi động chậm (tt)

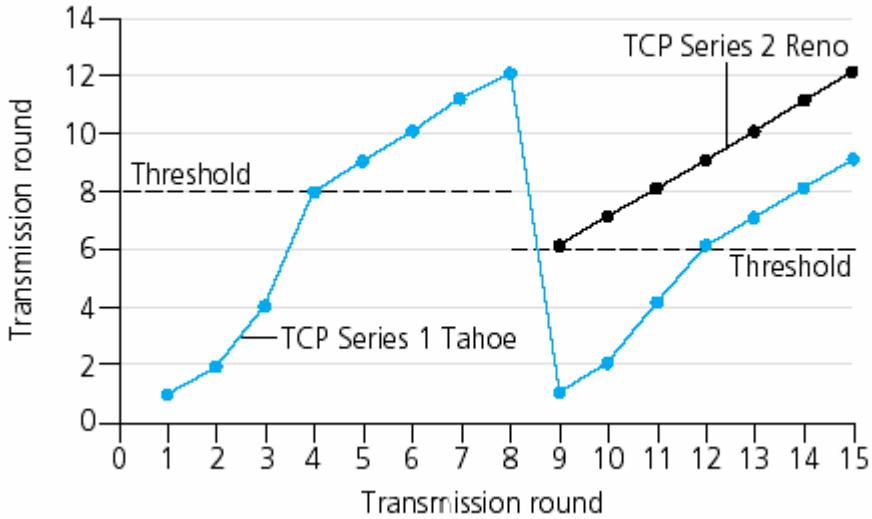
- Khi kết nối bắt đầu, tăng tốc lên rất nhanh cho đến khi sự cố mất mát xảy ra đầu tiên:
  - nhân đôi **CongWin** mỗi RTT
  - hoàn thành nhờ tăng **CongWin** ứng với mỗi ACK đã nhận
- Tổng kết: tốc độ khởi đầu là chậm nhưng sau đó tăng tốc rất nhanh



Tinh chế

Hỏi: Khi nào việc tăng tốc trở thành tuyến tính?

Trả lời: Khi **CongWin** đạt đến 1/2 giá trị của nó trước khi timeout.



Hiện thực:

- Ngưỡng thay đổi
- Tại thời điểm có sự cố mất mát, ngưỡng được cài giá trị bằng $\frac{1}{2}$ của CongWin ngay trước đó

Tinh chế: nhận biết mất mát

- Sau 3 ACK trùng lặp:
  - **CongWin** sẽ giảm 1/2
  - kích thước cửa sổ tăng tuyến tính
- nhưng sau sự cố timeout:
  - **CongWin** thay giá trị bằng 1 MSS;
  - kích thước cửa sổ tăng cấp lũy thừa
  - khi đến một ngưỡng thì tăng tuyến tính

Nguyên lý:

- 3 ACK trùng nhau chỉ ra khả năng truyền của mạng
- timeout chỉ thị "nhiều cảnh báo" về tình huống tắc nghẽn

Tổng kết: TCP điều khiển tắc nghẽn

- Khi **CongWin** dưới **Threshold**, bên gửi đang trong giai đoạn khởi động chậm, kích thước cửa sổ tăng nhanh theo cấp lũy thừa.
- Khi **CongWin** trên **Threshold**, bên gửi đang trong giai đoạn tránh tắc nghẽn, kích thước cửa sổ tăng nhanh theo cấp tuyến tính.
- Khi có 3 ACK trùng lặp xảy ra, **Threshold** = **CongWin/2** và **CongWin** = **Threshold**.
- Khi timeout xảy ra, **Threshold** = **CongWin/2** và **CongWin** = 1 MSS.

TCP điều khiển tắc nghẽn bên gửi

Trạng thái	Sự kiện	TCP bên gửi hành động	Diễn giải
Slow Start (SS)-Khởi động chậm	ACK báo nhận cho dữ liệu chưa ACK trước đó	CongWin = CongWin + MSS, If (CongWin > Threshold) cài đặt trạng thái "Tránh tắc nghẽn"	Hậu quả làm tăng gấp đôi CongWin mỗi RTT
Congestion Avoidance (CA) –Tránh tắc nghẽn	ACK báo nhận cho dữ liệu chưa ACK trước đó	CongWin = CongWin+MSS * (MSS/CongWin)	Additive tăng lên, làm tăng CongWin lên 1 MSS mỗi RTT
SS hoặc CA	Sự cố mất mát xảy ra khi thấy có 3 ACK trùng lặp	Threshold = CongWin/2, CongWin = Threshold, cài đặt trạng thái "Tránh tắc nghẽn"	Khôi phục nhanh, hiện thực giảm xuống multiplicative. CongWin sẽ không giảm xuống dưới 1 MSS.
SS hoặc CA	Timeout	Threshold = CongWin/2, CongWin = 1 MSS, cài đặt trạng thái "Khởi động chậm"	Vào chế độ "Khởi động chậm"
SS hoặc CA	ACK trùng lặp	Đếm ACK tăng lên cho đoạn vừa được ACK	CongWin và Threshold không thay đổi

TCP throughput

- Throughout trung bình của TCP biểu diễn qua kích thước của sổ và RTT?
  - Bỏ qua trạng thái "Khởi động chậm"
- Cho W là kích thước cửa sổ khi có mất mát xảy ra.
- Khi kích thước cửa sổ = W, lưu lượng = W/RTT
- Chỉ ngay sau khi mất mát, cửa sổ giảm xuống = W/2, lưu lượng = W/2RTT.
- throughout trung bình: 0.75 W/RTT

TCP tương lai

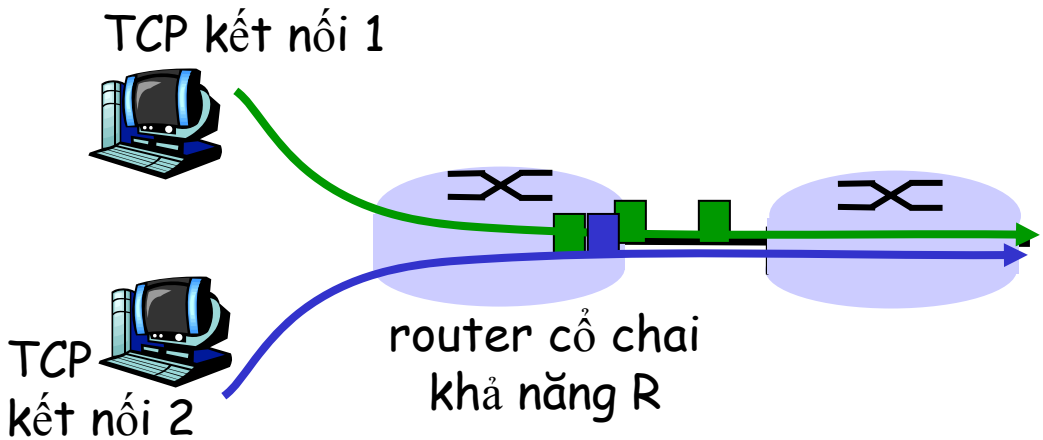
- Ví dụ: các đoạn dài 1500 byte, RTT 100ms, lưu lượng 10 Gbps
- Kích thước cửa sổ yêu cầu W = 83,333 đoạn trên đường truyền
- Lưu lượng trong các trường hợp mất mát:

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

- → L = $2 \cdot 10^{-10}$
- Phiên bản mới của TCP dành cho nhu cầu tốc độ cao!

TCP: tính công bằng

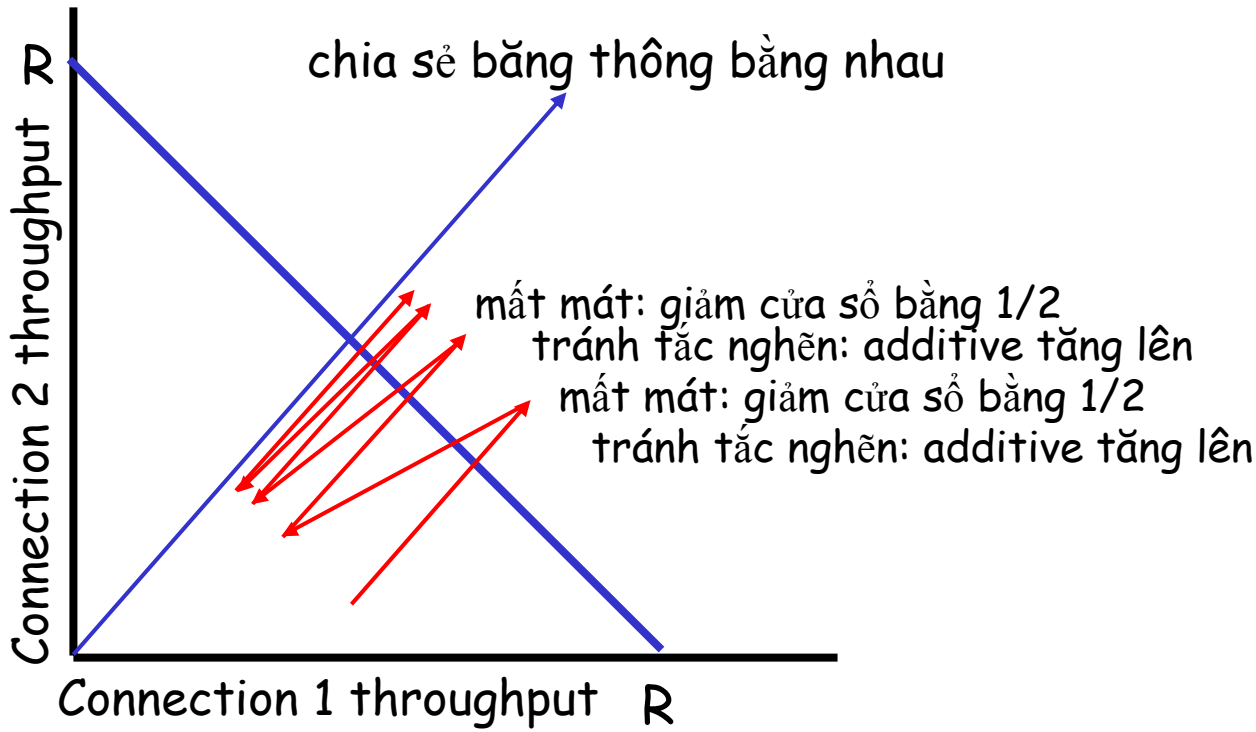
Mục tiêu: nếu K phiên làm việc TCP chia sẻ kết nối cổ chai của băng thông là R, mỗi phiên có tốc độ trung bình là R/K



Tại sao phải TCP công bằng?

2 phiên làm việc cạnh tranh nhau:

- Additive tăng, lưu lượng tăng
- multiplicative giảm lưu lượng tương xứng



TCP: tính công bằng (tt)

Tính công bằng & 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 mát
- Nghiên cứu: giao thức thân thiện với TCP

Tính công bằng & các kết nối TCP song song

- không có gì ngăn cản việc ứng dụng mở các kết nối song song giữa 2 host.
- Trình duyệt Web làm giống như thế
- Ví dụ: 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 TCP, có tốc độ R/2 !

Mô hình trễ

Hỏi: Mất bao lâu để nhận 1 đối tượng từ Web server sau khi gửi yêu cầu?

Bỏ qua tắc nghẽn, trễ bị ảnh hưởng bởi:

- thiết lập kết nối TCP
- trễ truyền dữ liệu
- khởi động chậm

Notation, các giả định:

- Giả sử một kết nối giữa client và server có tốc độ R
- S: MSS (bits)
- O: kích thước đối tượng (bits)
- không truyền lại (không mất mát, không hỏng)

Kích thước cửa sổ:

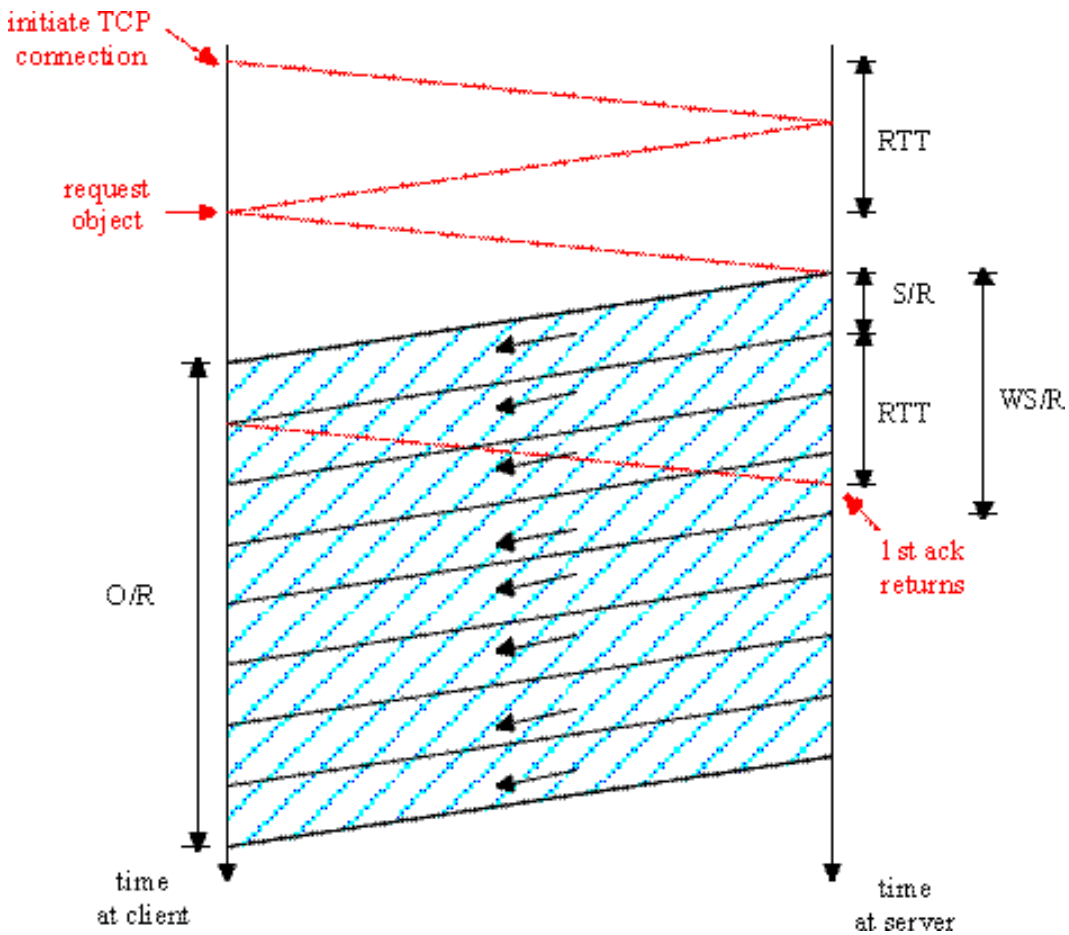
- Giả định 1: cửa sổ tắc nghẽn cố định, có W đoạn
- Sau đó cửa sổ thay đổi, mô hình khởi động chậm

Cửa sổ tắc nghẽn cố định (1)

Trường hợp đầu tiên:

WS/R > RTT + S/R: cho đoạn đầu tiên trong cửa sổ trả về trước khi cửa sổ dữ liệu gửi ACK

trễ = 2RTT + O/R

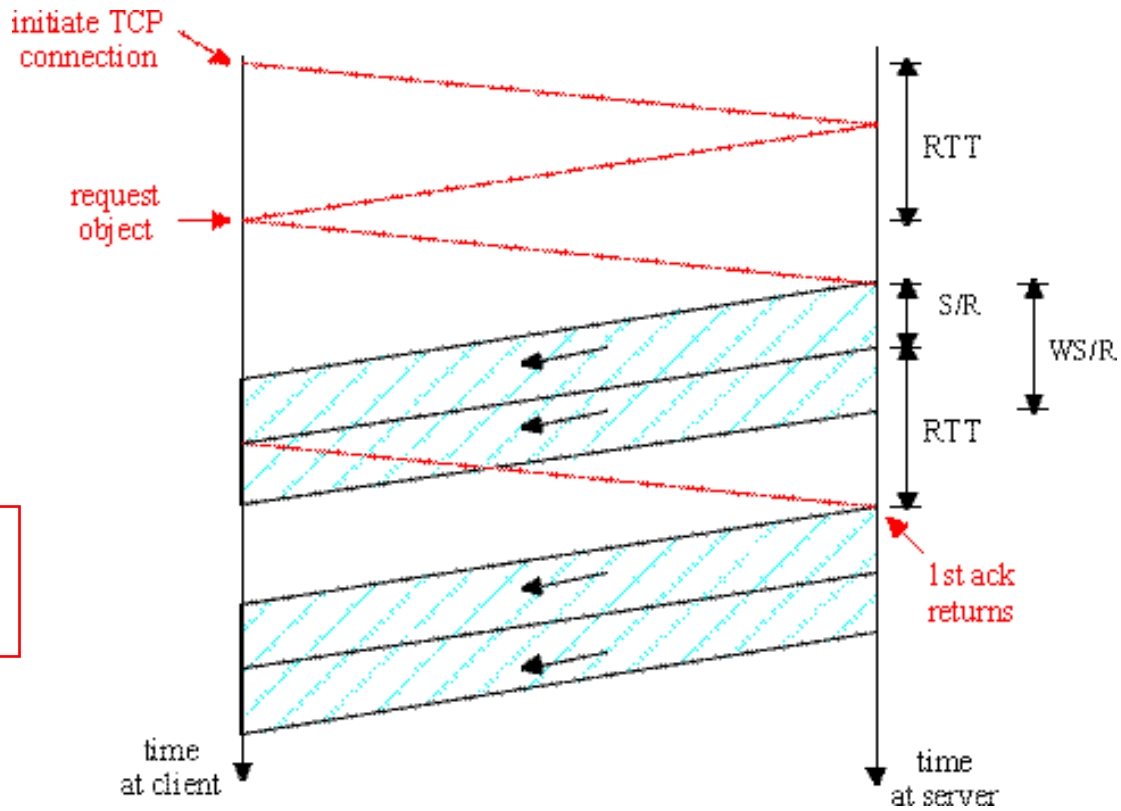


Cửa sổ tắc nghẽn cố định (2)

Trường hợp thứ hai:

- WS/R < RTT + S/R: sent chờ cho ACK sau khi gửi dữ liệu

trễ = 2RTT + O/R
+ (K-1)[S/R + RTT - WS/R]



TCP Mô hình trễ: Khởi động chậm (1)

Bây giờ giả sử kích thước cửa sổ tăng lên tùy theo quá trình khởi động chậm

Độ trễ của một đối tượng sẽ là:

$$Latency = 2RTT + \frac{O}{R} + P\left[RTT + \frac{S}{R}\right] - (2^P - 1)\frac{S}{R}$$

trong đó P là số lần TCP rảnh ở tại server:

$$P = \min\{Q, K-1\}$$

- trong đó Q là số lần server rảnh nếu đối tượng đã khởi tạo kích thước
- và K là số lượng cửa sổ bao trùm đối tượng

TCP Mô hình trễ: Khởi động chậm (2)

Các thành phần trễ:

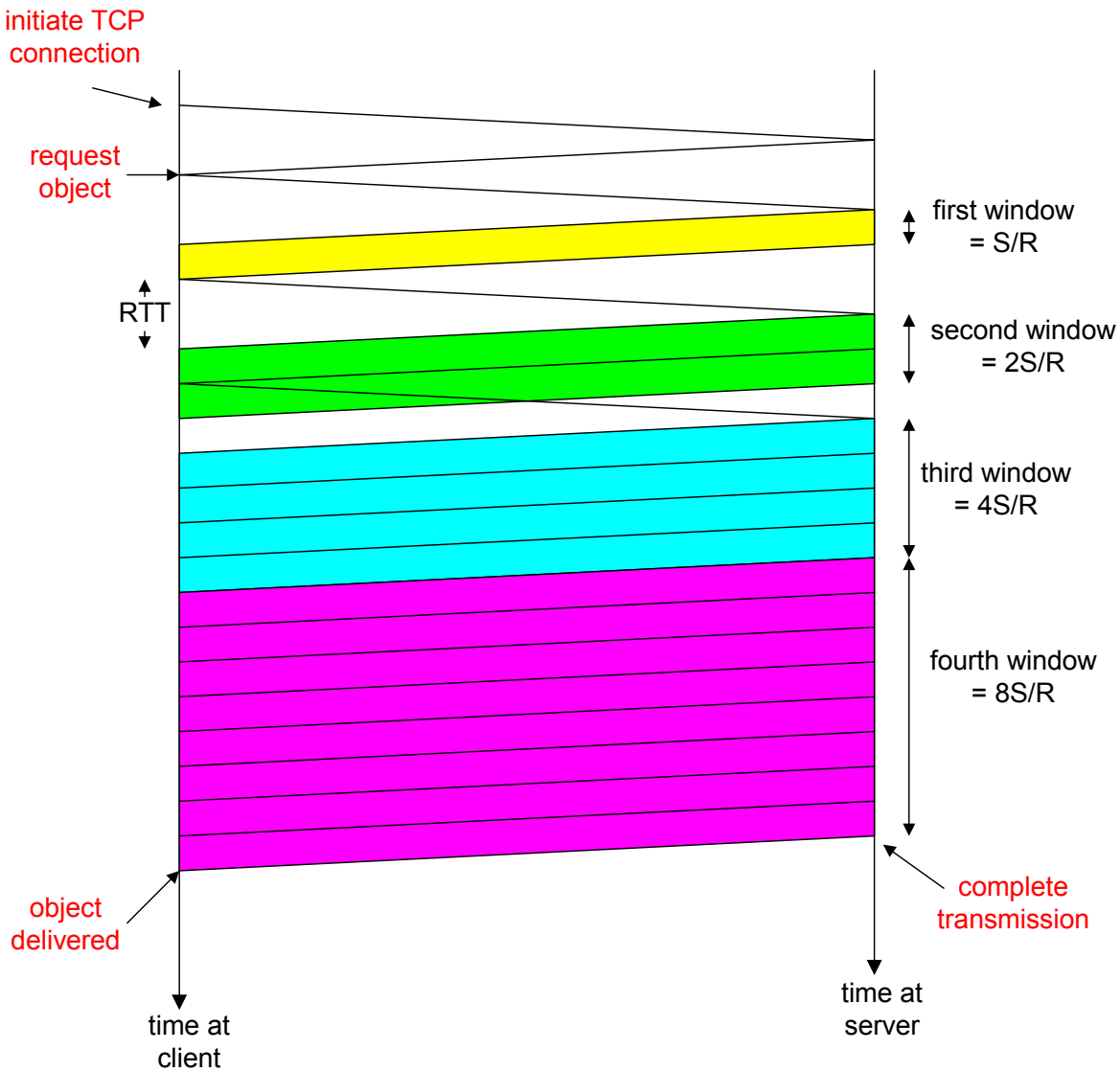
- 2 RTT dành cho thiết lập kết nối và yêu cầu
- O/R để truyền đối tượng
- thời gian server rảnh bởi vì khởi động chậm

Server rảnh:
P = min{K-1,Q} lần

Ví dụ:

- O/S = 15 đoạn
- K = 4 cửa sổ
- Q = 2
- P = min{K-1,Q} = 2

Server rảnh P=2 lần



TCP Mô hình trễ(3)

$\frac{S}{R} + RTT =$ time from when server starts to send segment

until server receives acknowledgement

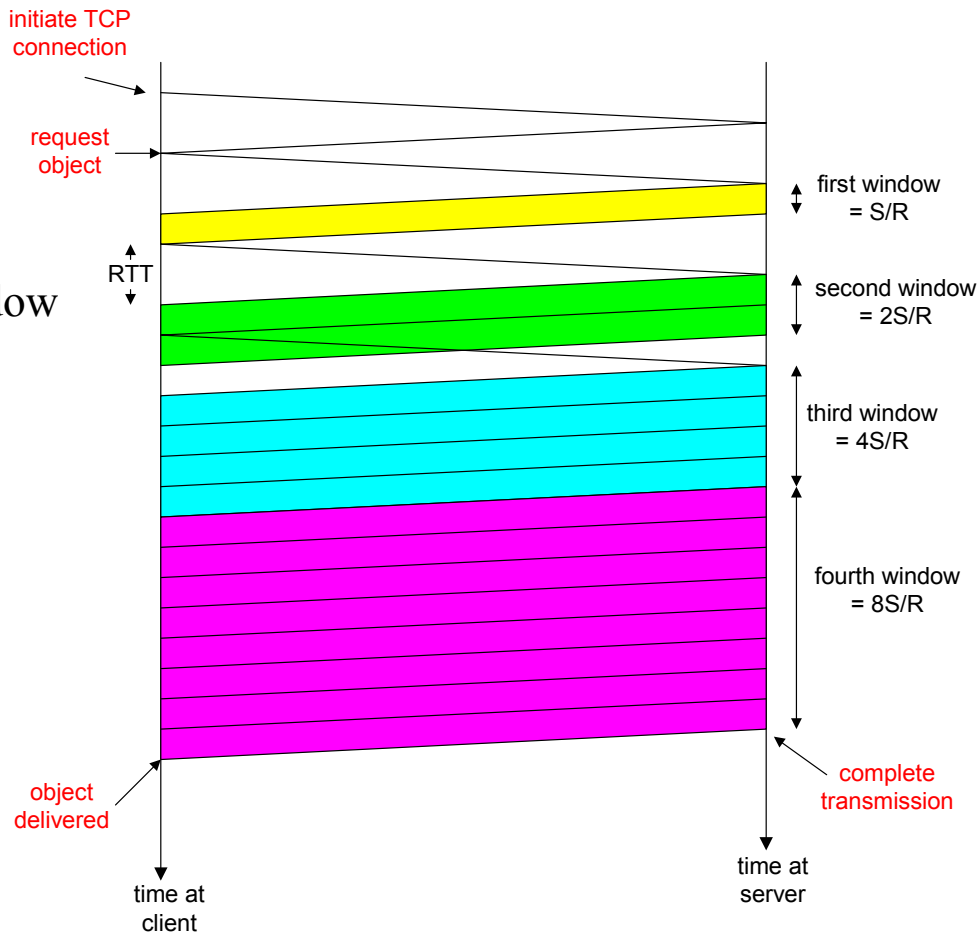
$2^{k-1}\frac{S}{R} =$ time to transmit the kth window

$\left[\frac{S}{R} + RTT - 2^{k-1}\frac{S}{R}\right]^{+} =$ idle time after the *k*th window

$$\text{delay} = \frac{O}{R} + 2RTT + \sum_{p=1}^{P} idleTime_p$$

$$= \frac{O}{R} + 2RTT + \sum_{k=1}^{P}\left[\frac{S}{R} + RTT - 2^{k-1}\frac{S}{R}\right]$$

$$= \frac{O}{R} + 2RTT + P\left[RTT + \frac{S}{R}\right] - (2^{P} - 1)\frac{S}{R}$$



TCP Mô hình trễ (4)

K = số lượng cửa sổ bao trùm đối tượng
Làm thế nào tính được K ?

$$
\begin{aligned}
K &= \min\{k : 2^0 S + 2^1 S + \cdots + 2^{k-1} S \geq O\} \\
&= \min\{k : 2^0 + 2^1 + \cdots + 2^{k-1} \geq O / S\} \\
&= \min\{k : 2^k - 1 \geq \frac{O}{S}\} \\
&= \min\{k : k \geq \log_2(\frac{O}{S} + 1)\} \\
&= \left\lceil \log_2(\frac{O}{S} + 1) \right\rceil
\end{aligned}
$$

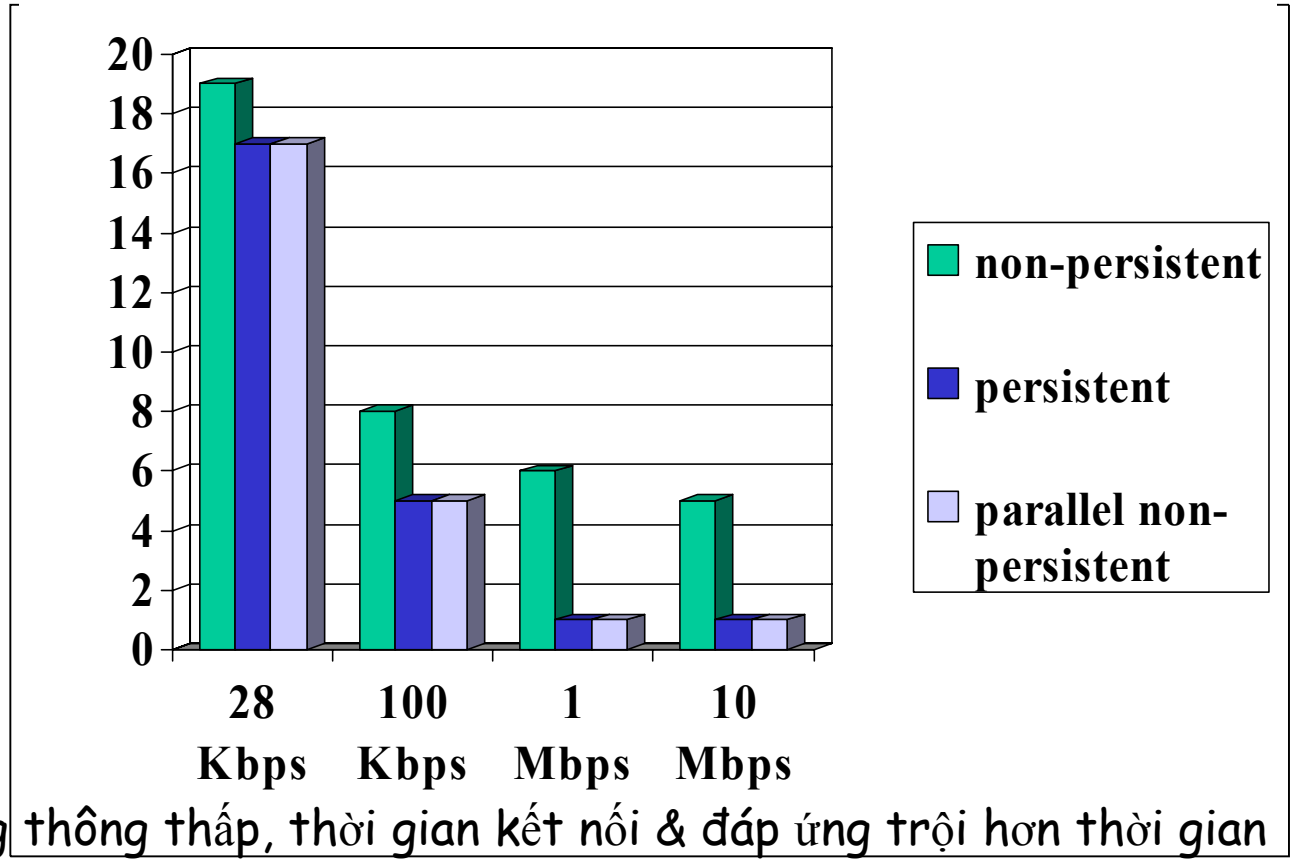
cách tính toán Q tương tự (xem HW).

HTTP Mô hình

- Giả sử trang Web chứa:
  - *1* trang HTML (kích thước *O* bits)
  - *M* hình ảnh (mỗi cái kích thước *O* bits)
- HTTP không bền vững:
  - *M+1* TCP kết nối
  - *Thời gian đáp ứng = (M+1)O/R + (M+1)2RTT + tổng số thời gian rảnh*
- HTTP bền vững:
  - *2 RTT* để yêu cầu và nhận file HTML
  - *1 RTT* để yêu cầu và nhận M hình ảnh
  - *Thời gian đáp ứng = (M+1)O/R + 3RTT + tổng số thời gian rảnh*
- HTTP không bền vững với X kết nối song song
  - Giả sử M/X là số nguyên.
  - 1 TCP kết nối cho file
  - M/X thiết lập các kết nối song song cho các hình ảnh
  - *Thời gian đáp ứng = (M+1)O/R + (M/X + 1)2RTT + tổng số thời gian rảnh*

HTTP thời gian đáp ứng (giây)

RTT = 100 msec, O = 5 Kbytes, M=10 và X=5

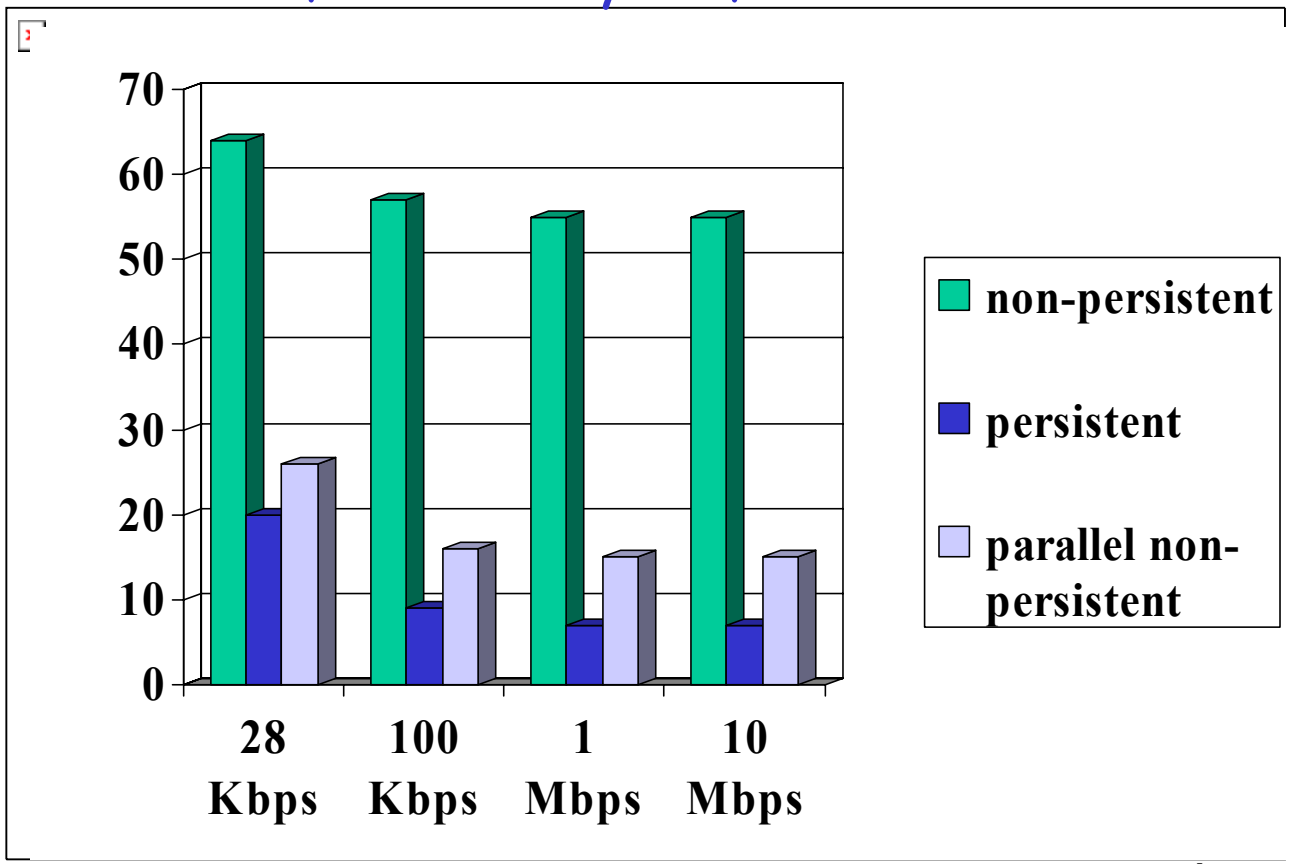


Với băng thông thấp, thời gian kết nối & đáp ứng trội hơn thời gian truyền

Các kết nối bền vững chỉ cho sự cải thiện không đáng kể trên các kết nối song song

HTTP thời gian đáp ứng (giây)

RTT =1 sec, O = 5 Kbytes, M=10 and X=5



Với RTT lớn hơn, thời gian đáp ứng trội hơn thời gian trễ chờ thiết lập kết nối TCP & khởi động chậm. Các kết nối bền vững bây giờ cho thấy cải thiện rõ rệt: đặc biệt với các mạng băng thông cao

Chương 3: Tổng kết

- các nguyên lý của các dịch vụ lớp transport
  - multiplexing, demultiplexing
  - truyền dữ liệu tin cậy
  - điều khiển luồng
  - điều khiển tắc nghẽn
- khởi tạo và hiện thực trong Internet
  - UDP
  - TCP

Tiếp theo:

- nghiên cứu xong các vấn đề "ngoài biên" (các lớp application, transport)
- chuẩn bị vào phần "lõi" của mạng