Mạng máy tính

Bộ môn Kỹ thuật máy tính và Mạng
Khoa Công nghệ Thông tin
Đại học Sư phạm Hà Nội

Chương 4: Tầng giao vận

Mục đích:

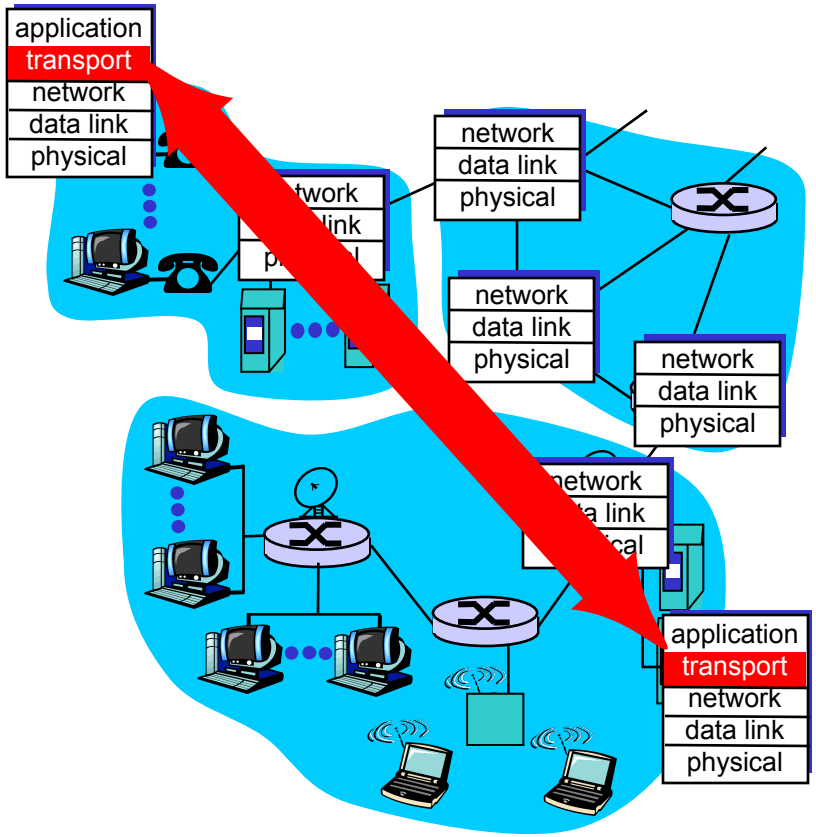
- Hiểu các nguyên tắc bên trong dịch vụ của tầng giao vận:
  - Multiplexing/Demultiplexing
  - Truyền dữ liệu tin cậy
  - Điều khiển luồng
  - Điều khiển tắc nghẽn
- Học về giao thức tầng giao vận trong Internet:
  - UDP: không hướng kết nối
  - TCP: hướng kết nối
  - Điều khiển tắc nghẽn của TCP

Chương 4: Tầng giao vận

- 4.1 Các dịch vụ tầng giao vận
- 4.2 Multiplexing và demultiplexing
- 4.3 Dịch vụ không hướng kết nối: UDP
- 4.4 Các nguyên tắc của truyền dữ liệu tin cậy
- 4.5 Dịch vụ hướng kết nối: TCP
  - Cấu trúc segment
  - Truyền dữ liệu tin cậy
  - Điều khiển luồng
  - Quản lý kết nối
- 4.6 Các nguyên tắc của điều khiển tắc nghẽn
- 4.7 Điều khiển tắc nghẽn của TCP

Các giao thức và dịch vụ tầng giao vận

- Cung cấp *truyền thông lô-gíc* giữa các tiến trình ứng dụng chạy trên các host khác nhau
- Các giao thức giao vận chạy trên các hệ thống cuối
  - Bên gửi: chia các bản tin ứng dụng thành các segment, chuyển tới tầng mạng
  - Bên nhận: ghép các segment thành bản tin, chuyển lên tầng ứng dụng
- Nhiều hơn một giao thức giao vận cho ứng dụng
  - Internet: TCP và UDP



Tầng giao vận và tầng mạng

- *Tầng mạng:* truyền thông lô-gíc giữa các host
- *Tầng giao vận:* truyền thông lô-gíc giữa các tiến trình
  - dựa trên dịch vụ của tầng mạng

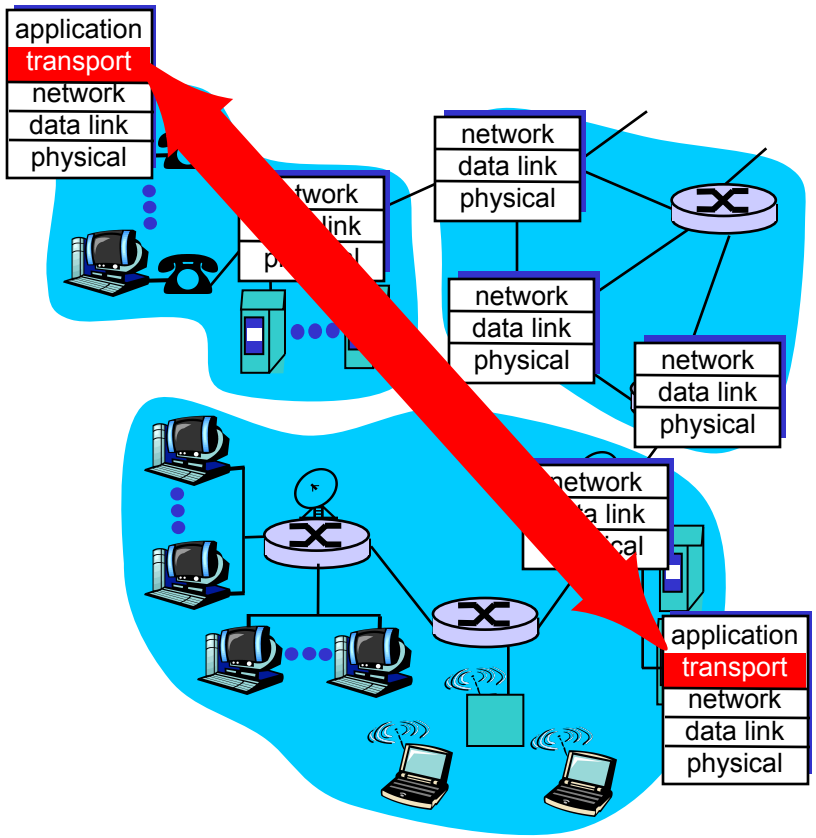
Tương tự hộ gia đình:

12 đứa trẻ gửi thư cho 12 đứa trẻ

- Các tiến trình = các đứa trẻ
- Các bản tin ứng dụng = các bức thư
- host = nhà
- Giao thức giao vận = Ann và Bill
- Giao thức tầng mạng = dịch vụ chuyển thư

Các giao thức tầng giao vận của Internet

- Truyền tin cậy, có thứ tự (TCP)
  - Điều khiển tắc nghẽn
  - Điều khiển luồng
  - Thiết lập kết nối
- Truyền không có thứ tự, không tin cậy: UDP
- Các dịch vụ không có:
  - Đảm bảo độ trễ
  - Đảm bảo băng thông



Chương 4: Tầng giao vận

- 4.1 Các dịch vụ tầng giao vận
- 4.2 Multiplexing và demultiplexing
- 4.3 Dịch vụ không hướng kết nối: UDP
- 4.4 Các nguyên tắc của truyền dữ liệu tin cậy
- 4.5 Dịch vụ hướng kết nối: TCP
  - Cấu trúc segment
  - Truyền dữ liệu tin cậy
  - Điều khiển luồng
  - Quản lý kết nối
- 4.6 Các nguyên tắc của điều khiển tắc nghẽn
- 4.7 Điều khiển tắc nghẽn của TCP

Multiplexing/demultiplexing

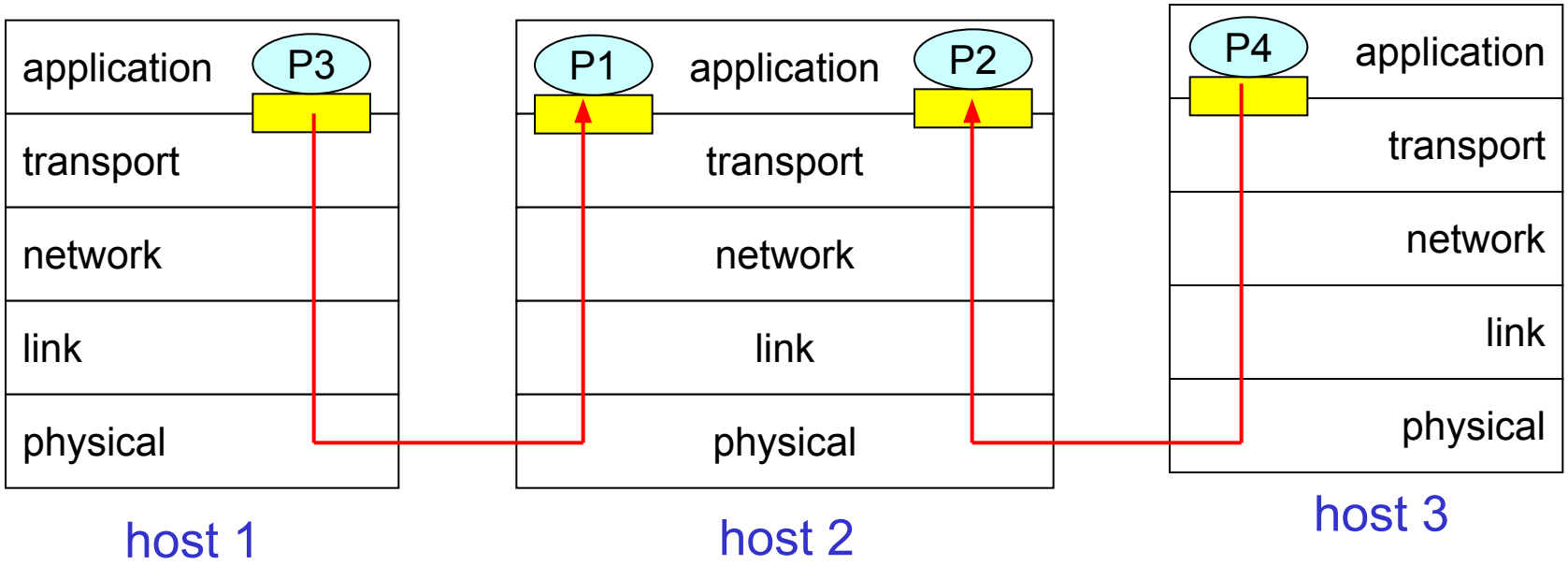
Demultiplexing tại host nhận:

Chuyển các segment đã nhận tới đúng socket

Multiplexing tại host gửi:

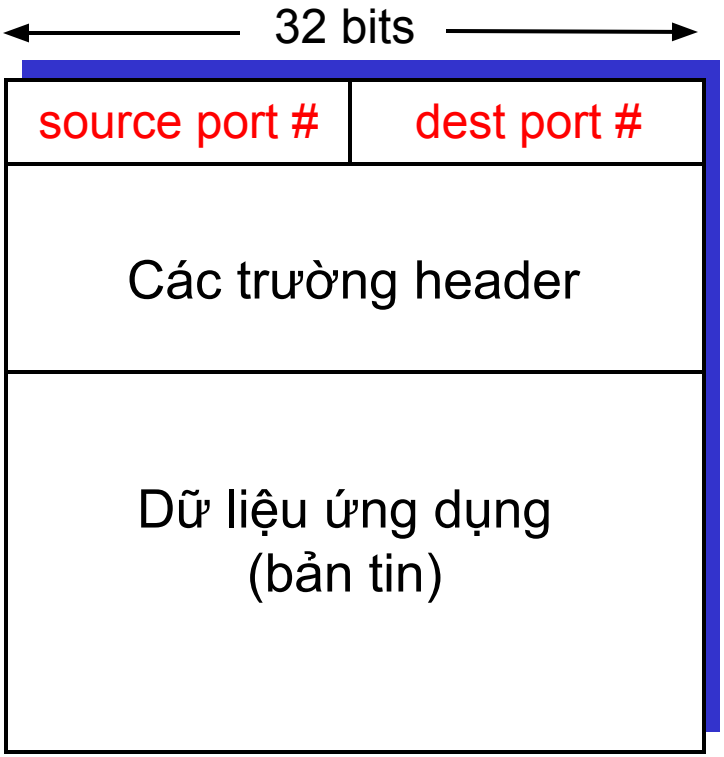
Thu thập dữ liệu từ các socket, đóng gói dữ liệu bởi header (sau đó sẽ dùng để demultiplexing)

= socket = tiến trình



Thực hiện demultiplexing

- Host nhận gói dữ liệu IP
  - Mỗi gói dữ liệu có địa chỉ IP nguồn, địa chỉ IP đích
  - Mỗi gói dữ liệu mang một segment của tầng giao vận
  - Mỗi segment có giá trị cổng nguồn và cổng đích (giá trị cổng cố định cho các kiểu ứng dụng cụ thể)
- Host sử dụng địa chỉ IP và giá trị cổng để chuyển segment tới socket thích hợp



Định dạng TCP/UDP segment

Chương 4: Tầng giao vận

- 4.1 Các dịch vụ tầng giao vận
- 4.2 Multiplexing và demultiplexing
- 4.3 Dịch vụ không hướng kết nối: UDP
- 4.4 Các nguyên tắc của truyền dữ liệu tin cậy
- 4.5 Dịch vụ hướng kết nối: TCP
  - Cấu trúc segment
  - Truyền dữ liệu tin cậy
  - Điều khiển luồng
  - Quản lý kết nối
- 4.6 Các nguyên tắc của điều khiển tắc nghẽn
- 4.7 Điều khiển tắc nghẽn của TCP

UDP: User Datagram Protocol [RFC 768]

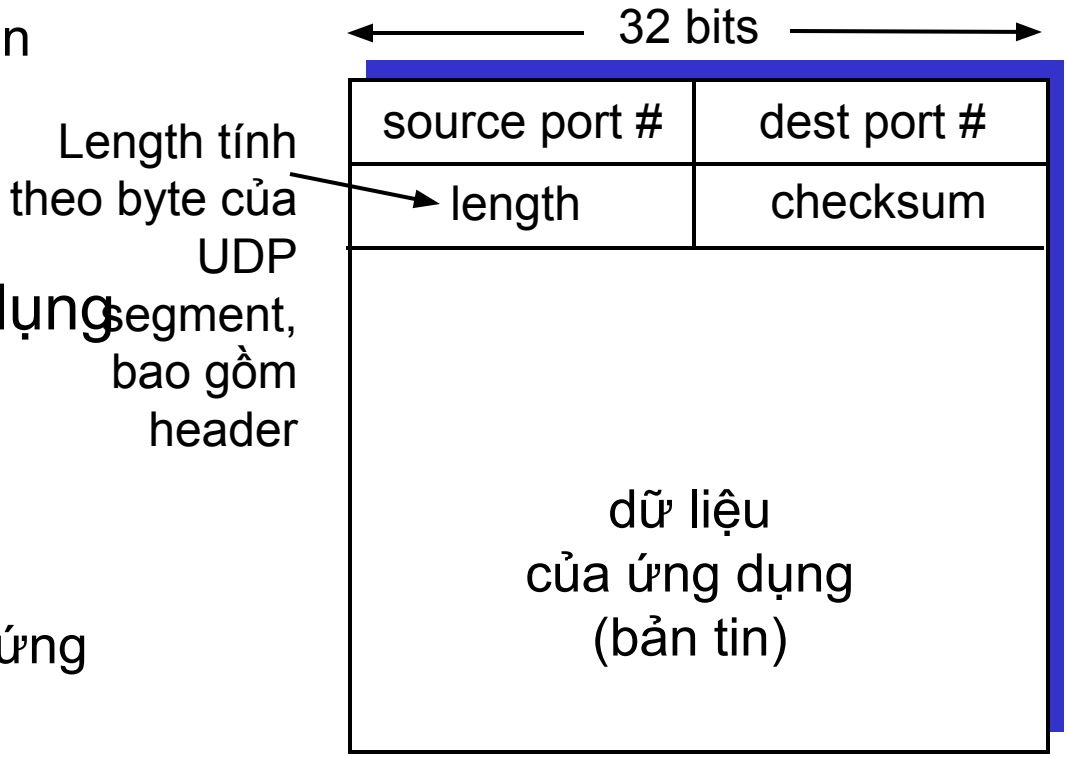
- Dịch vụ "best effort", UDP segment có thể:
  - mất
  - chuyển không theo thứ tự đến ứng dụng
- *Không hướng kết nối:*
  - Không có bắt tay giữa bên gửi và bên nhận
  - Mỗi UDP segment được điều khiển độc lập

Tại sao có UDP?

- Không thiết lập kết nối (thiết lập có thể tăng độ trễ)
- Đơn giản: không có trạng thái kết nối tại bên gửi, bên nhận
- Header của segment nhỏ
- Không điều khiển tắc nghẽn: UDP có thể gửi ra với tốc độ mong muốn

UDP (tiếp)

- Thường sử dụng cho các ứng dụng đa phương tiện truyền dòng
  - Chấp nhận mất gói
  - Nhạy cảm với tốc độ
- Ứng dụng khác sử dụng UDP
  - DNS
  - SNMP
- Truyền tin cậy qua UDP: thêm sự tin cậy tại tầng ứng dụng
  - Khôi phục lỗi do ứng dụng cụ thể



Định dạng của UDP segment

UDP checksum

Mục đích: phát hiện lỗi trong segment đã truyền

Bên gửi:

- Đối xử với nội dung các segment như chuỗi các số nguyên 16 bít
- checksum: cộng (tổng bù của 1) của nội dung segment
- Phía gửi đặt giá trị checksum trong trường checksum của UDP

Bên nhận:

- Tính toán checksum của segment nhận được
- Kiểm tra xem checksum đã tính có bằng giá trị trường checksum:
  - KHÔNG BẰNG– Phát hiện có lỗi
  - BẰNG – không phát hiện ra lỗi. Nhưng có thể có lỗi?

Ví dụ Checksum

- Chú ý
  - Khi cộng các số, giá trị bít nhớ cần thêm vào kết quả
- Ví dụ: cộng hai số nguyên 16 bít

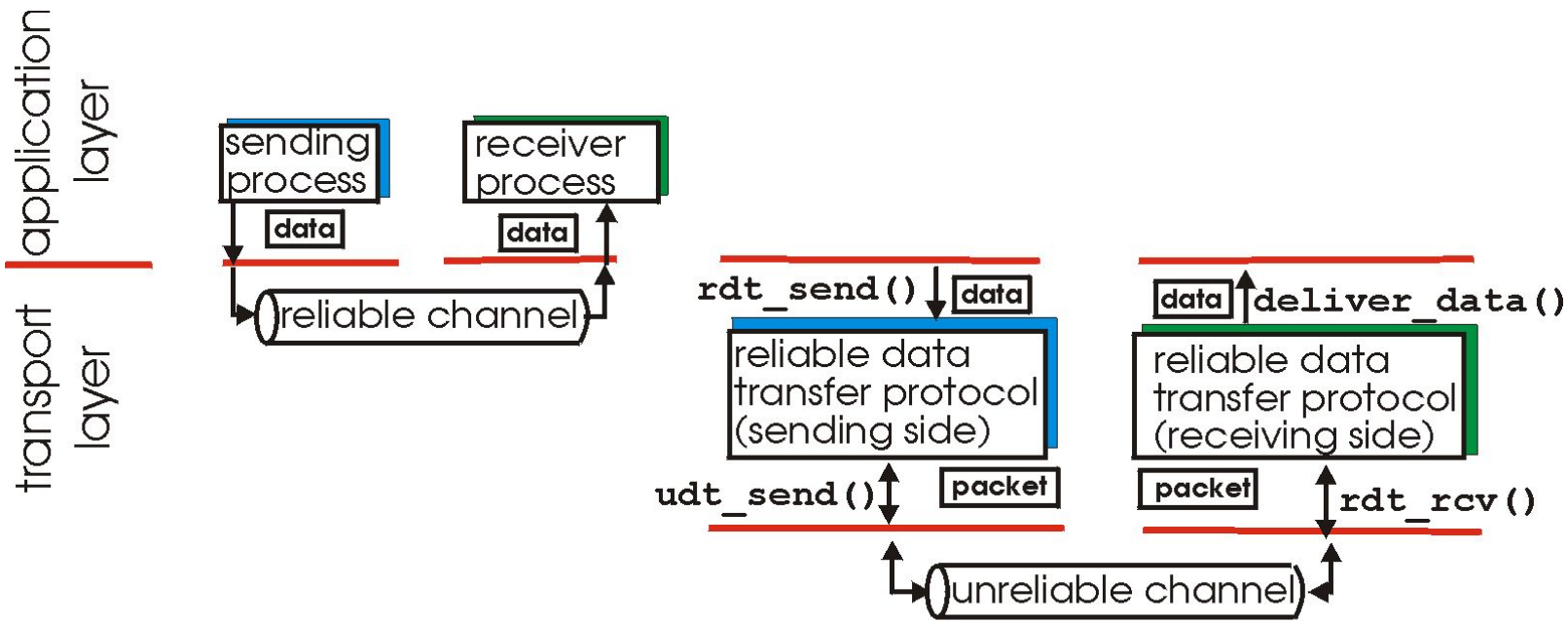
	1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
Tổng checksum	1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
	0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

Chương 4: Tầng giao vận

- 4.1 Các dịch vụ tầng giao vận
- 4.2 Multiplexing và demultiplexing
- 4.3 Dịch vụ không hướng kết nối: UDP
- 4.4 Các nguyên tắc của truyền dữ liệu tin cậy
- 4.5 Dịch vụ hướng kết nối: TCP
  - Cấu trúc segment
  - Truyền dữ liệu tin cậy
  - Điều khiển luồng
  - Quản lý kết nối
- 4.6 Các nguyên tắc của điều khiển tắc nghẽn
- 4.7 Điều khiển tắc nghẽn của TCP

Các nguyên tắc của truyền dữ liệu tin cậy

- Tầm quan trọng của tầng liên kết dữ liệu, tầng giao vận, tầng ứng dụng



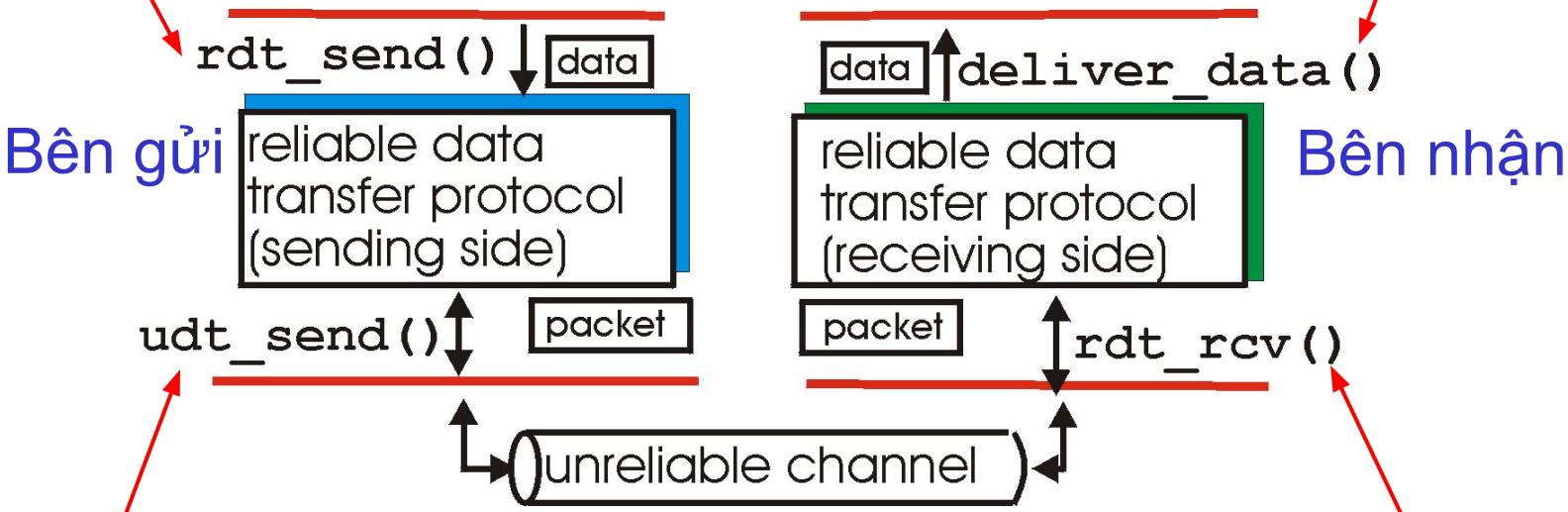
(a) Dịch vụ cung cấp (b) Cài đặt dịch vụ

- Đặc điểm của kênh truyền không tin cậy xác định sự phức tạp của giao thức truyền dữ liệu tin cậy (rdt)

Truyền dữ liệu tin cậy

rdt_send(): được gọi bởi tầng trên. Dữ liệu đã chuyển được chuyển tới tầng trên của bên nhận

deliver_data(): được gọi bởi **rdt** để truyền dữ liệu lên tầng trên



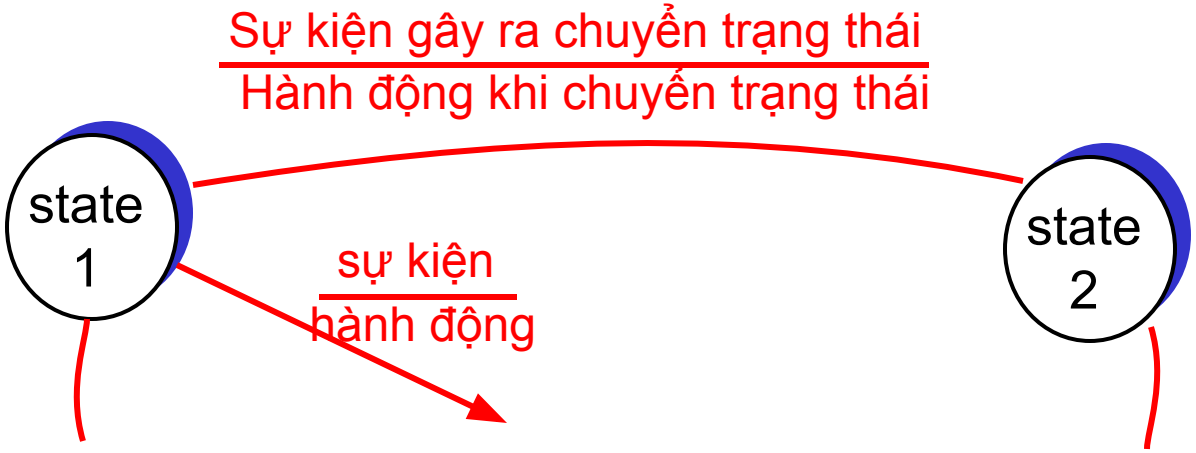
udt_send(): gọi bởi rdt, để truyền gói tin qua kênh không tin cậy tới bên nhận

rdt_rcv(): gọi khi gói tin đến phía bên nhận

Truyền dữ liệu tin cậy

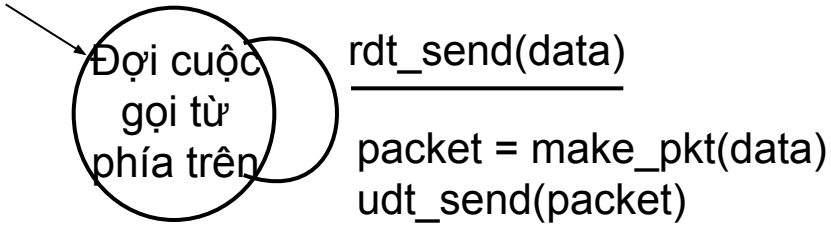
- Sử dụng máy trạng thái hữu hạn (FSM) để xử lý bên nhận và bên gửi

state: khi trong 1 trạng thái, trạng thái tiếp theo là duy nhất đối với 1 sự kiện

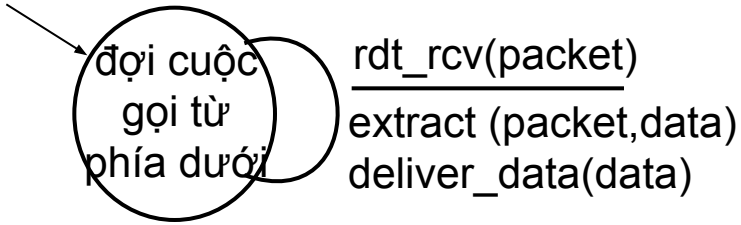


rdt1.0: Truyền tin cậy qua kênh tin cậy

- Tầng dưới là truyền tin cậy
  - Không có lỗi bít
  - Không mất gói tin
- FSM của bên gửi và bên nhận:
  - Bên gửi chuyển dữ liệu xuống kênh phía dưới
  - Bên nhận đọc dữ liệu từ kênh bên dưới



Bên gửi

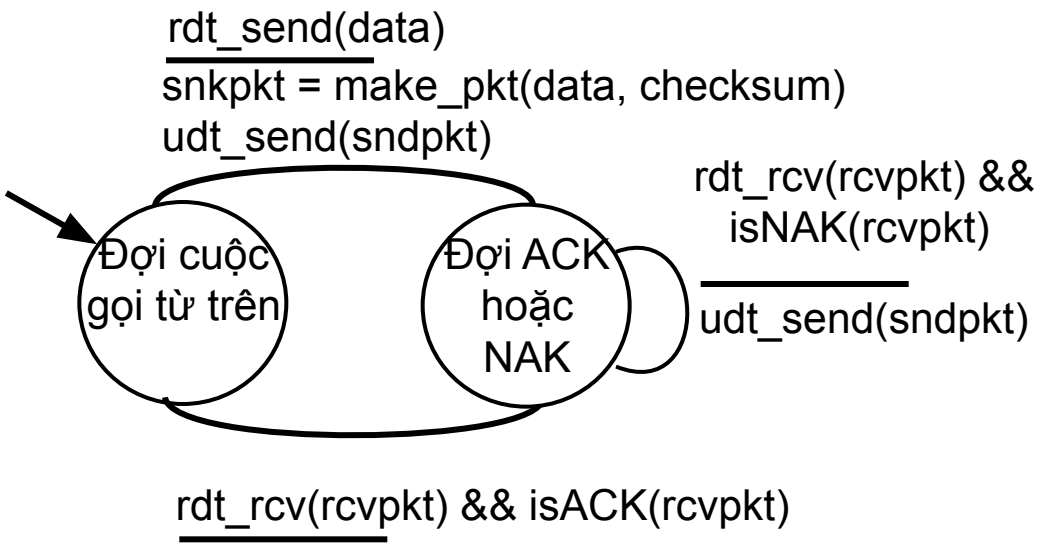


Bên nhận

Rdt2.0: kênh có lỗi bít

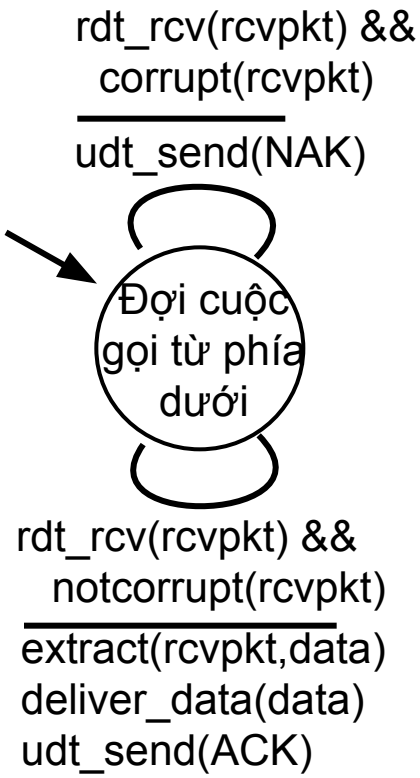
- Kênh phía dưới có thể có lỗi
  - checksum để phát hiện lỗi
- *Cách khôi phục lỗi*
  - *Báo nhận (ACK):* bên nhận chỉ rõ cho bên gửi gói tin nhận thành công
  - *Báo lỗi (NAK):* bên nhận chỉ rõ cho bên gửi gói tin có lỗi
  - Bên nhận truyền lại gói tin nếu nhận NAK
- Cơ chế **`rdt2.0`**:
  - Phát hiện lỗi
  - Phản hồi cho bên nhận: bản tin điều khiển (ACK, NAK: bên nhận -> bên gửi)

rdt2.0: Máy trạng thái

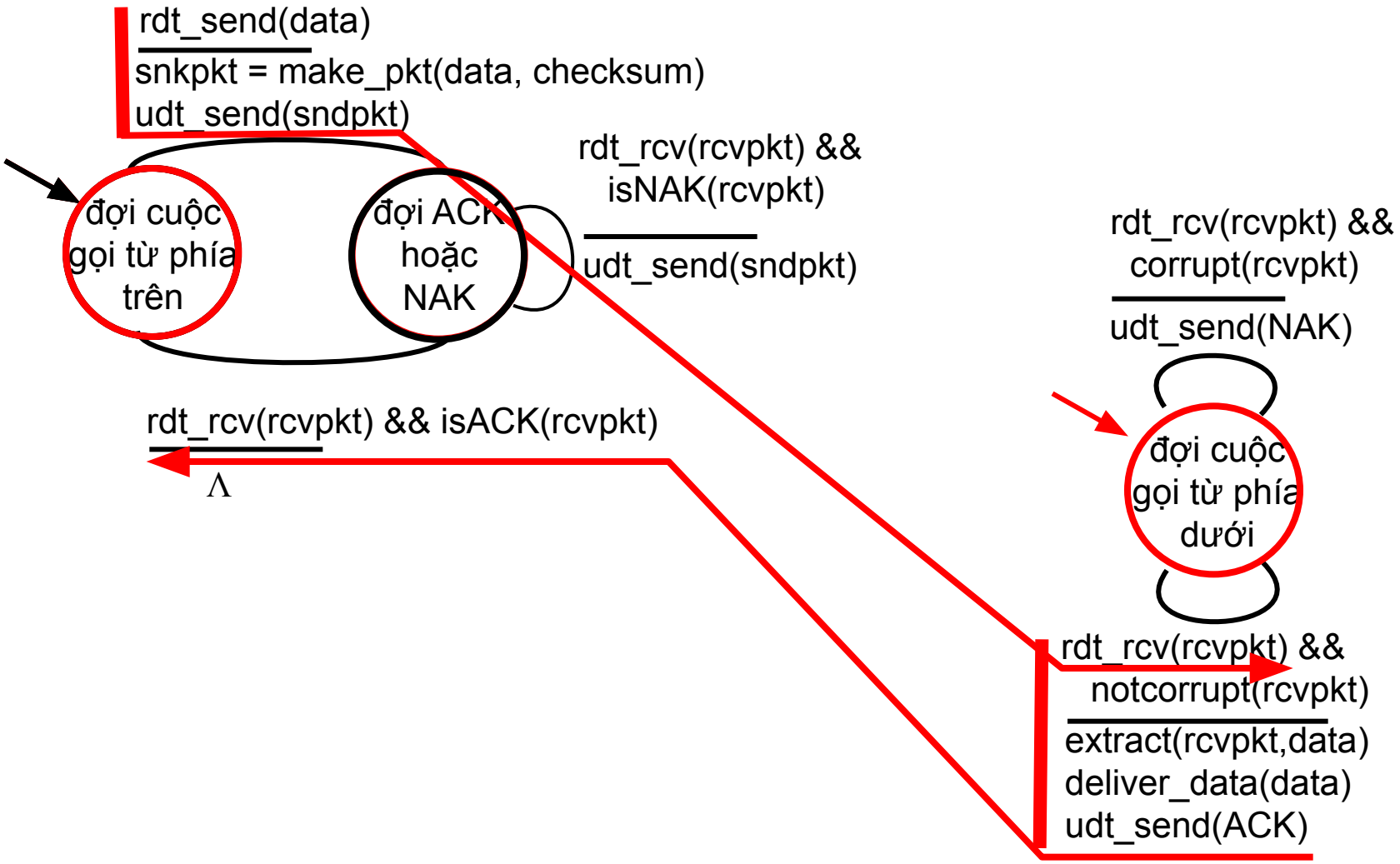


Bên gửi

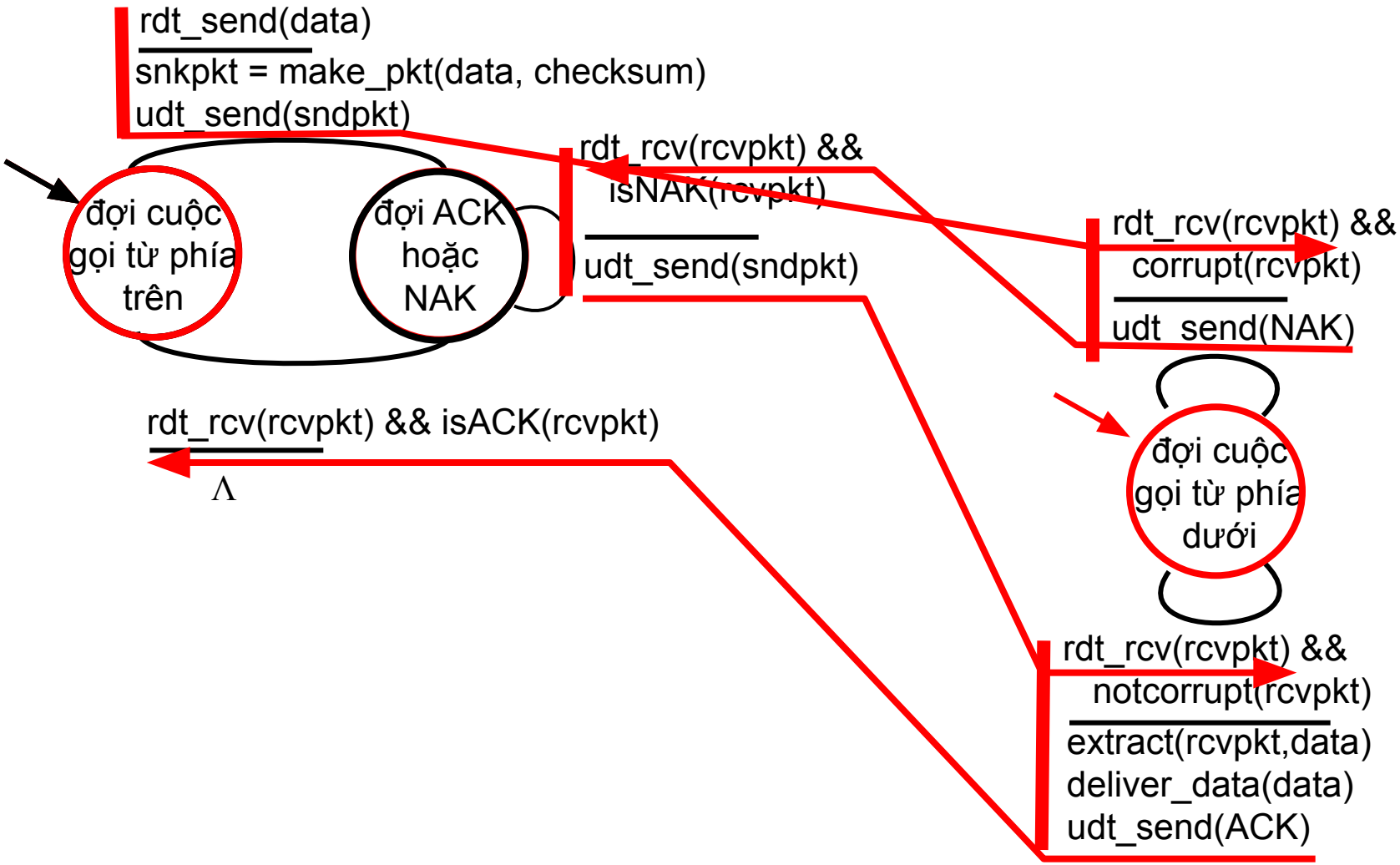
Bên nhận



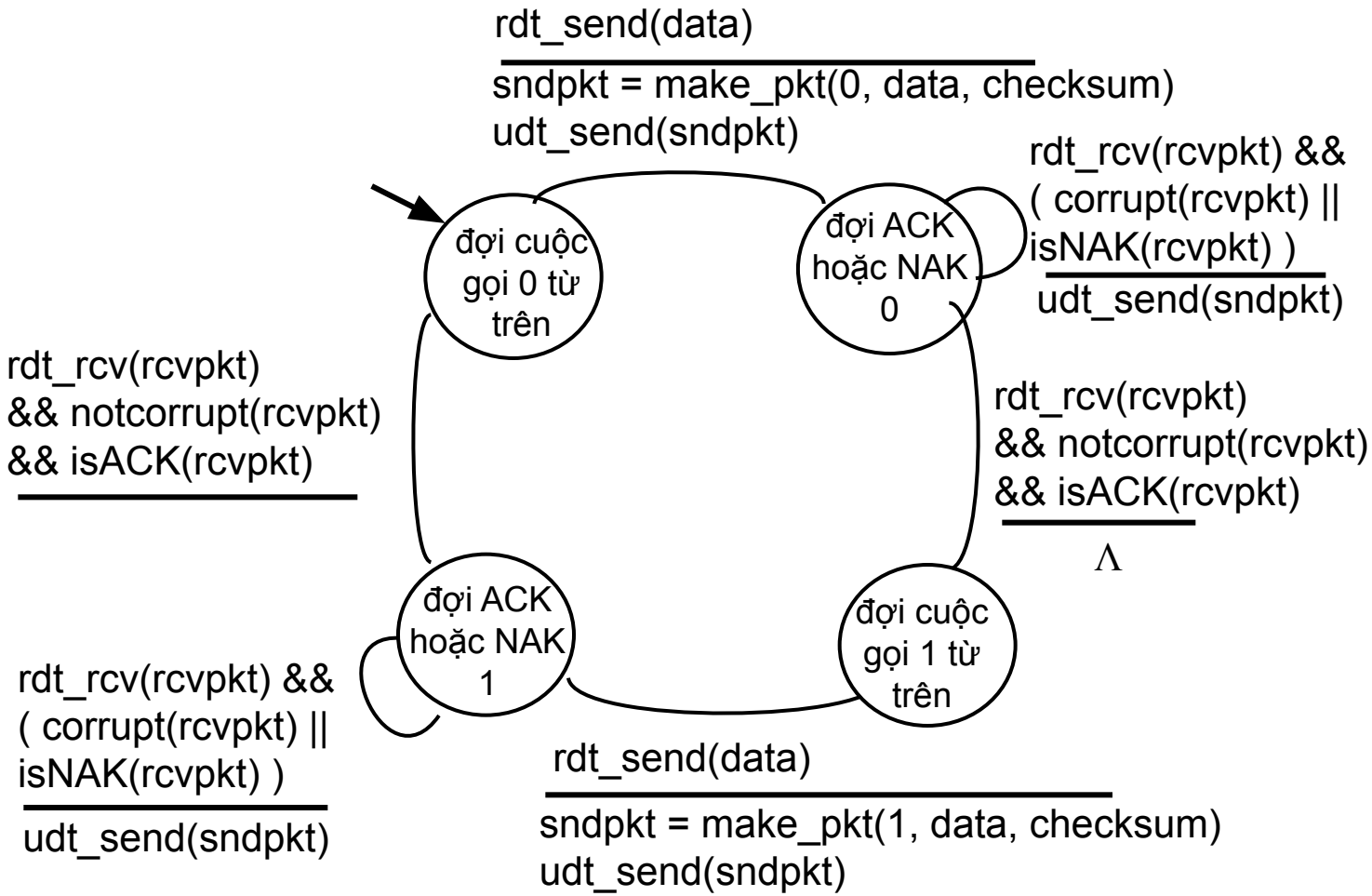
rdt2.0: Trường hợp không lỗi



rdt2.0: Trường hợp có lỗi



rdt2.1: Bên gửi, điều khiển ACK/NAK lỗi



rdt2.1: Bên nhận, điều khiển ACK/NAK lỗi

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq0(rcvpkt)

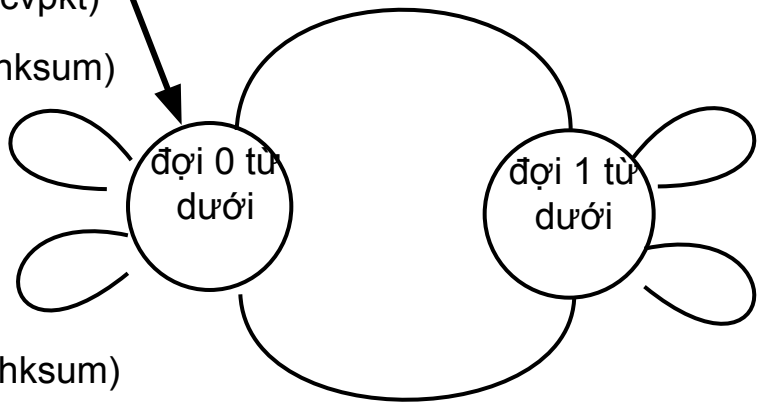
extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && (corrupt(rcvpkt)

sndpkt = make_pkt(NAK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
not corrupt(rcvpkt) &&
has_seq1(rcvpkt)

sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)



rdt_rcv(rcvpkt) && (corrupt(rcvpkt)

sndpkt = make_pkt(NAK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
not corrupt(rcvpkt) &&
has_seq0(rcvpkt)

sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq1(rcvpkt)

extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

rdt2.1

Bên gửi:

- seq # được thêm vào gói tin
- Hai seq. #'s (0,1)
- Phải kiểm tra nếu ACK/NAK đã nhận có lỗi
- Hai lần -> ổn định

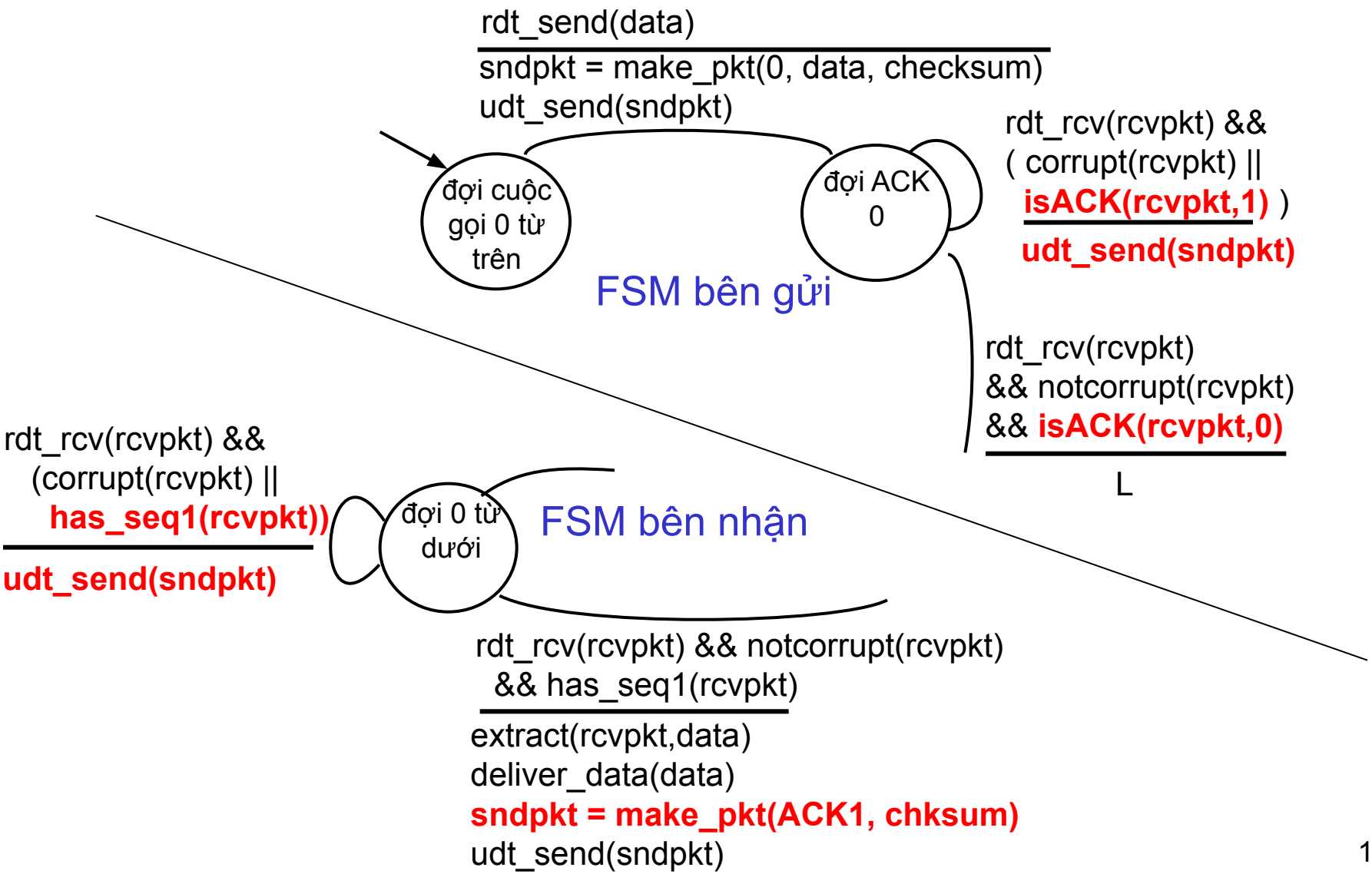
Bên nhận:

- Phải kiểm tra gói tin đã nhận có lặp không
  - Trạng thái chỉ định pkt seq # mong đợi là 0 hay 1

rdt2.2: Giao thức NAK-free

- Tương tự rdt2.1, chỉ sử dụng ACK
- Thay vì NAK, bên nhận phải gửi ACK cho pkt cuối đã nhận OK
- ACK lặp tại bên nhận sẽ như xử lý như nhận NAK: truyền lại pkt hiện tại

rdt2.2: Phân mảnh tại bên gửi và bên nhận



rdt3.0: kênh có lỗi và mất gói

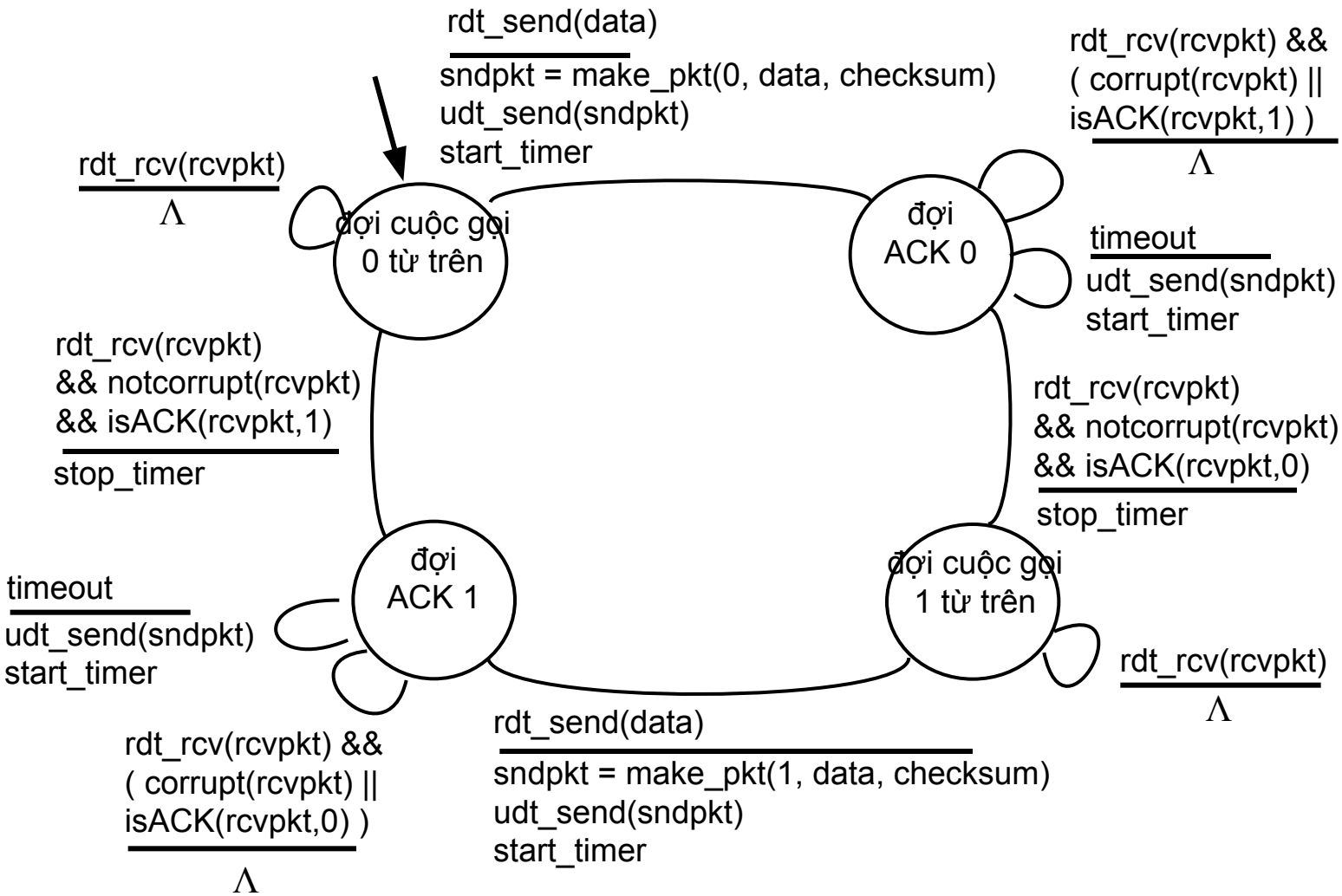
Giả sử: kênh phía dưới có thể mất gói (dữ liệu hoặc ACK)

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

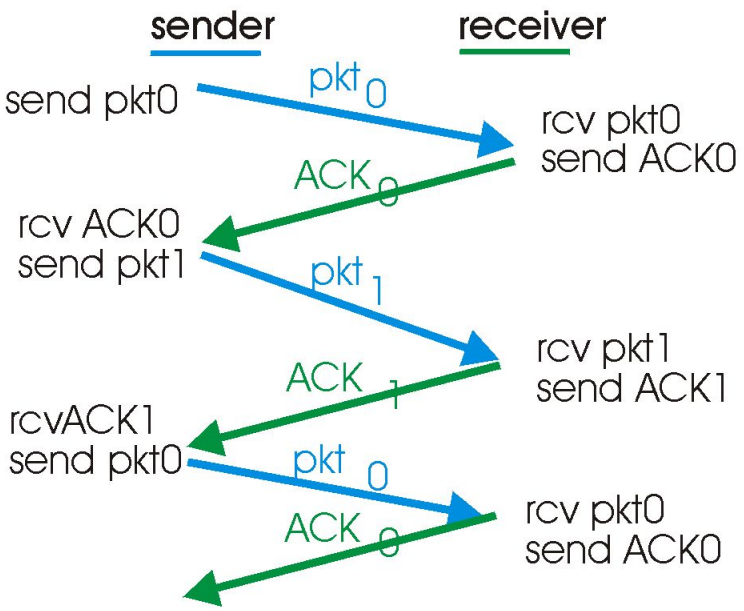
Cách tiếp cận: bên nhận đợi ACK một thời gian

- Truyền lại nếu không có ACK nhận tại thời điểm này
- Nếu gói tin (hoặc ACK) trễ (không mất):
  - Truyền lại -> lặp, sử dụng seq# để giải quyết
  - Bên nhận phải gán seq # của gói tin được ACK
- Đòi hỏi bộ đếm thời gian ngược

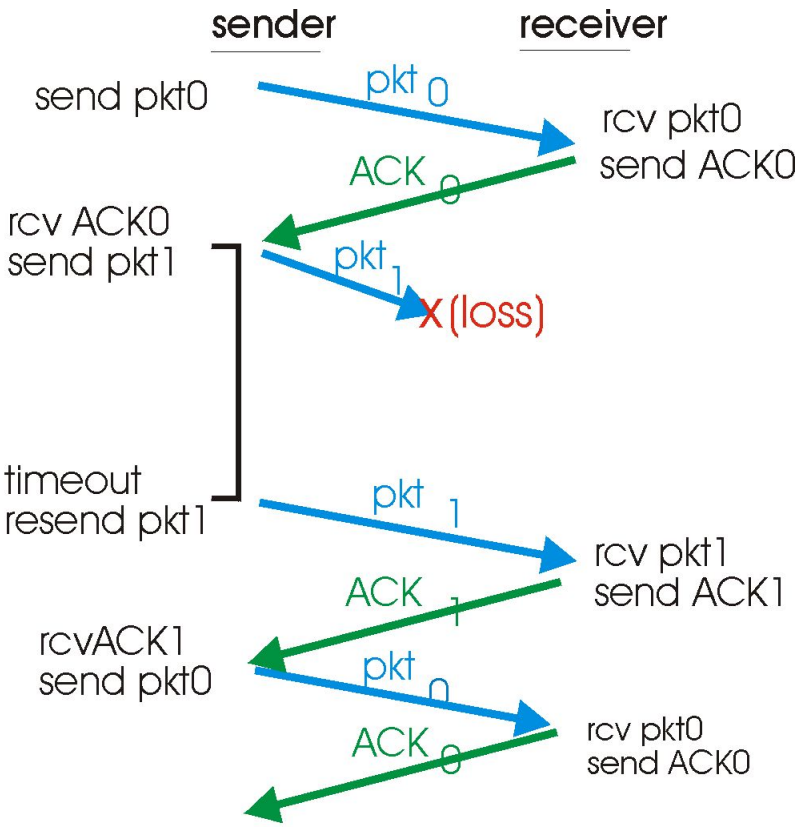
rdt3.0 Bên gửi



rdt3.0

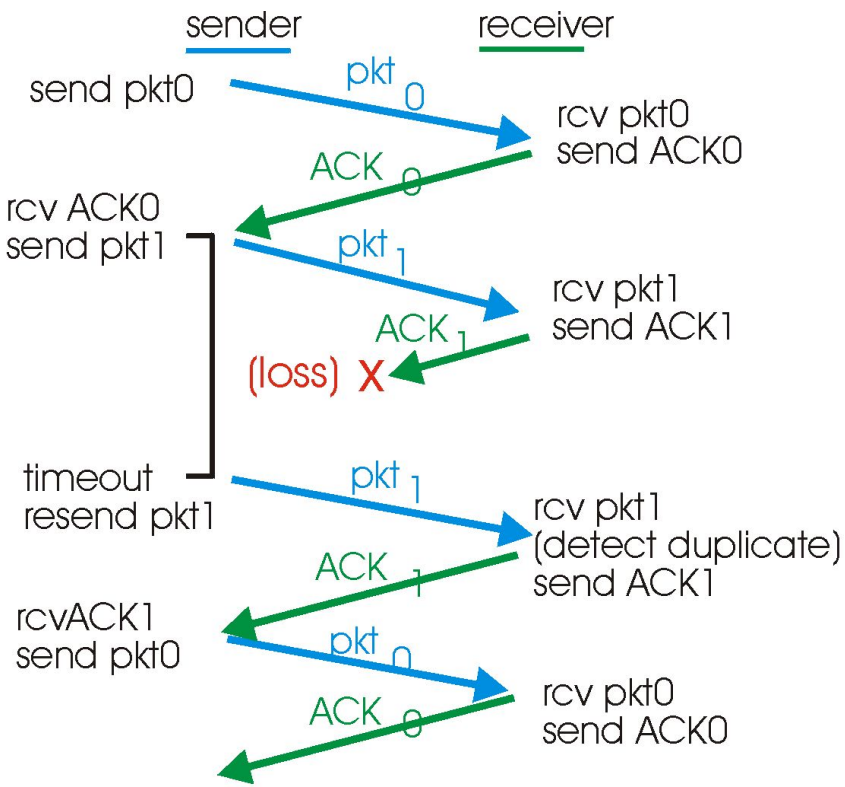


(a) Không mất gói

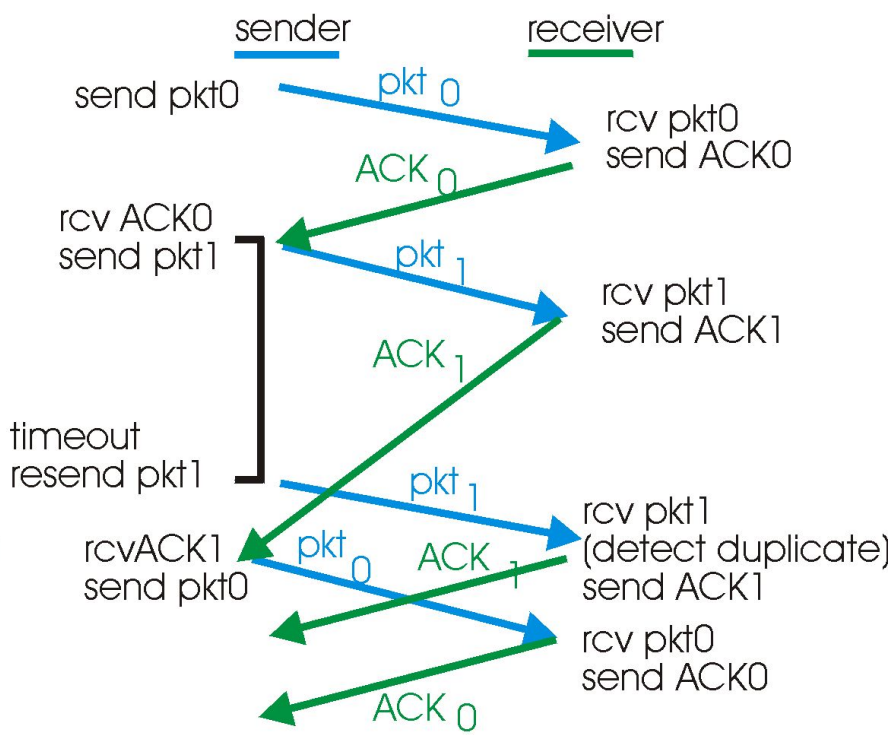


(b) Mất gói

rdt3.0



(c) Mất ACK



(c) Timeout

Hiệu năng của rdt3.0

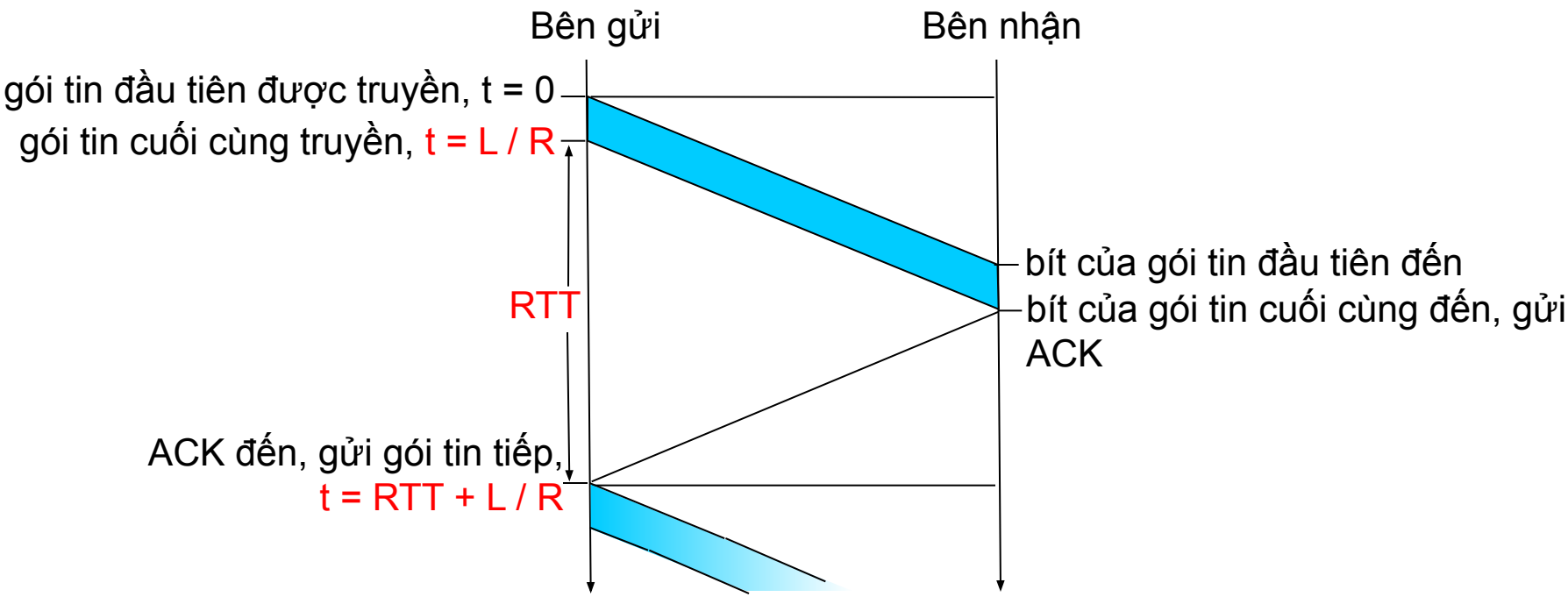
- Hiệu năng của rdt3.0 bị ảnh hưởng
- Ví dụ: đường truyền 1 Gbps, lan truyền 15 ms, gói tin 1KB:

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

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

  - U_{sender}: sự sử dụng – thời gian bên gửi bận gửi
  - 1KB pkt trong mỗi 30 msec -> 33kB/sec thông lượng qua đường truyền 1 Gbps
  - Giao thức mạng hạn chế sử dụng tài nguyên vật lý!

rdt3.0: Hoạt động stop-and-wait

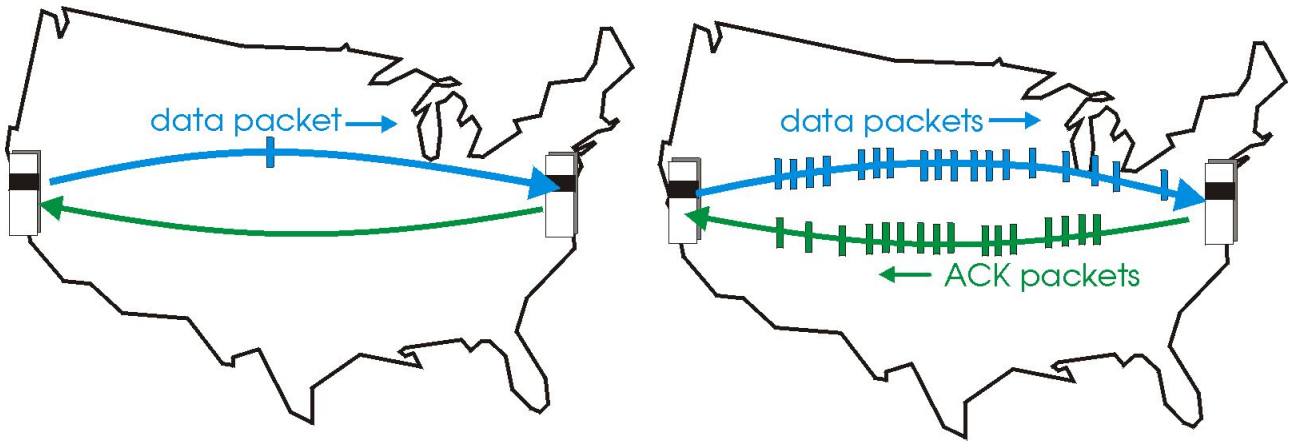


$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

Các giao thức Pipeline

Pipeline: Bên gửi cho phép nhiều, tới các gói tin được ack

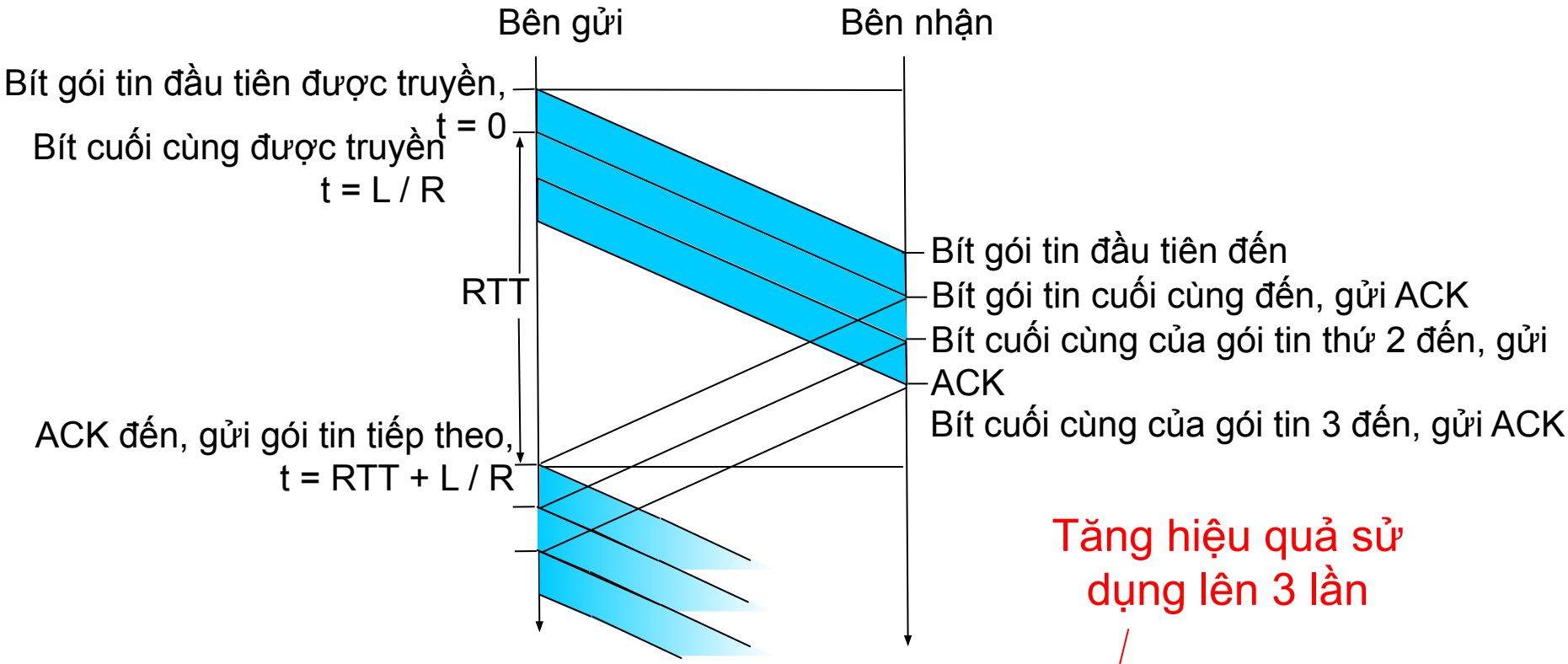
- Dải giá trị sequence phải tăng
- Vùng đệm tại bên gửi và bên nhận



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

- Hai hình thức chung của các giao thức pipeline: *go-Back-N, selective repeat*

Pipelining: Tăng hiệu quả sử dụng



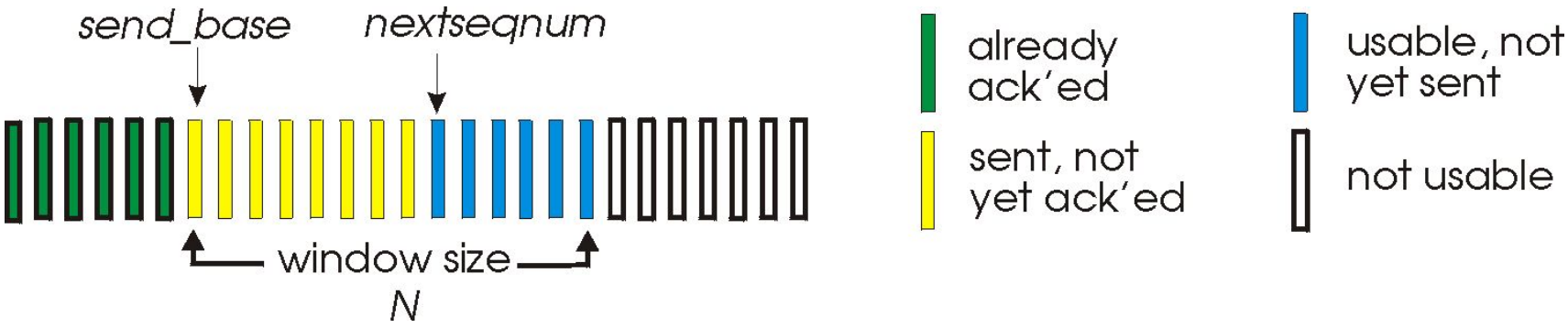
Tăng hiệu quả sử dụng lên 3 lần

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

Go-Back-N

Bên gửi:

- k-bit seq # trong pkt header
- "window" N, cho phép các gói tin không ack liên tiếp



- ACK(n): ACK mọi gói tin tới seq # n - "ACK tích lũy"
  - Có thể nhầm ACK lặp
- Thời gian cho mỗi gói tin
- *timeout(n):* truyền lại gói tin n và tất cả gói tin seq# lớn hơn n trong cửa sổ
- Lý do phải giới hạn N: điều khiển luồng, điều khiển tắc nghẽn

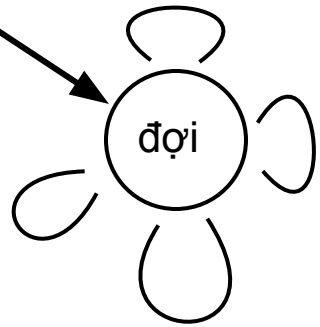
GBN: FSM mở rộng của bên gửi

rdt_send(data)

```
if (nextseqnum < base+N) {
   sndpkt[nextseqnum] = make_pkt(nextseqnum,data,chksum)
   udt_send(sndpkt[nextseqnum])
   if (base == nextseqnum) /* pkt chưa được ack đầu tiên */
      start_timer
   nextseqnum++
   }
else
  refuse_data(data)
```

Λ

base=1
nextseqnum=1



timeout

```
start_timer
udt_send(sndpkt[base])
udt_send(sndpkt[base+1])
…
udt_send(sndpkt[nextseqnum-1])
```

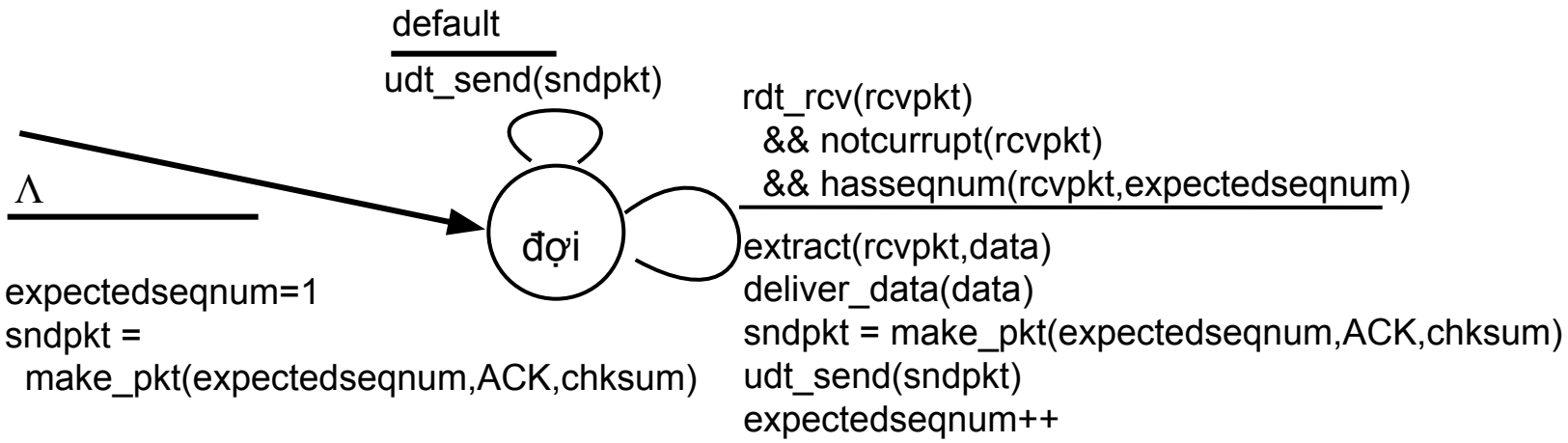
rdt_rcv(rcvpkt)
&& corrupt(rcvpkt)

rdt_rcv(rcvpkt) &&
notcorrupt(rcvpkt)

```
base = getacknum(rcvpkt)+1
If (base == nextseqnum) /* tất cả các pkt đều được ack */
   stop_timer
 else /* vẫn còn gói tin chưa được ack */
   start_timer
```

Sử dụng một timer cho gói tin đã gửi và chưa được ack và cũ nhất

GBN: FSM mở rộng của bên nhận

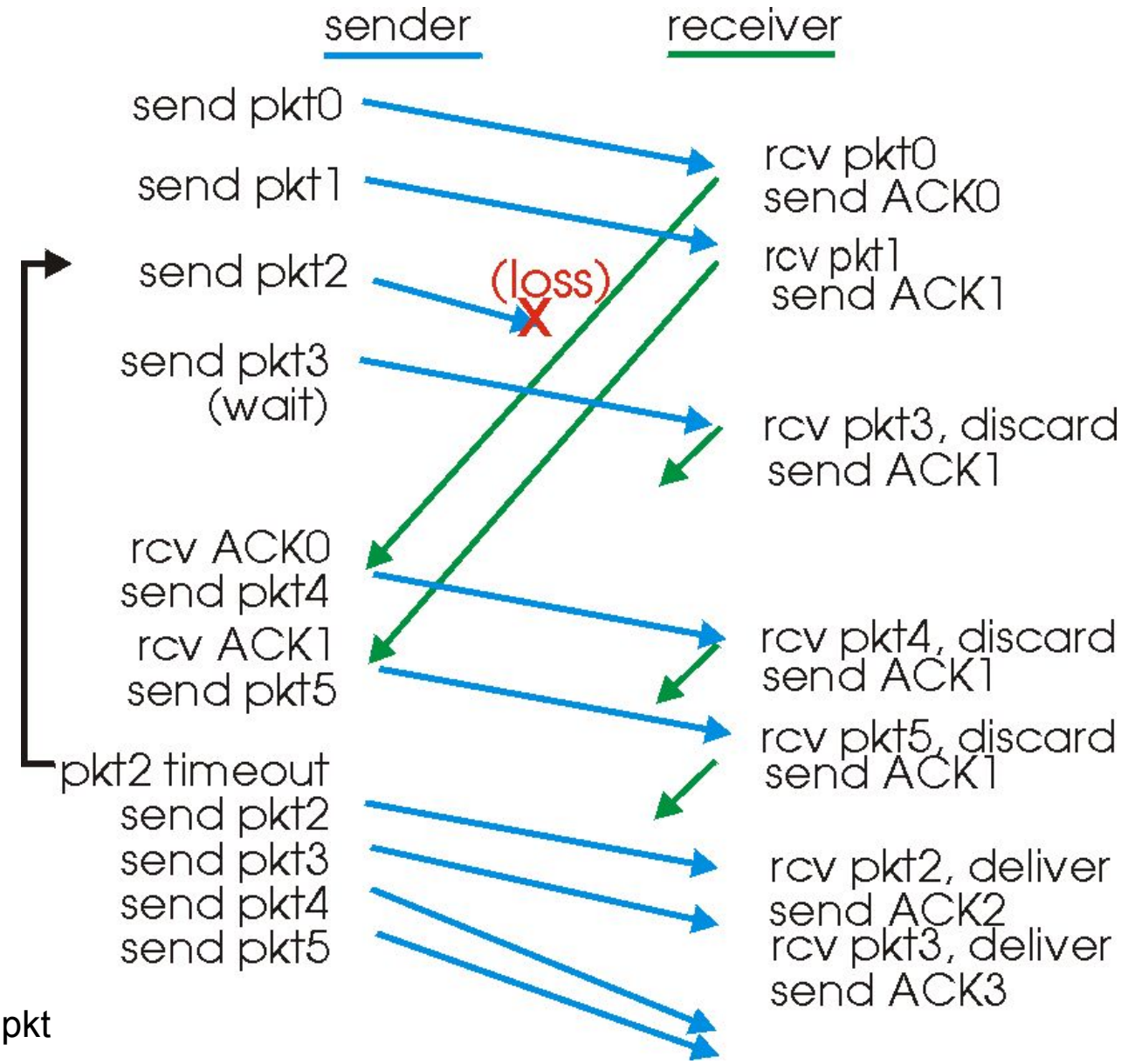


ACK: luôn gửi ACK cho gói tin đã nhận đúng với gói tin đúng thứ tự seq # nhất

- Có thể sinh ra ACK lặp
- Chỉ cần nhớ **`expectedseqnum`**

- Gói tin không đúng thứ tự:
  - Loại bỏ (không đưa vào bộ nhớ đệm)

GBN

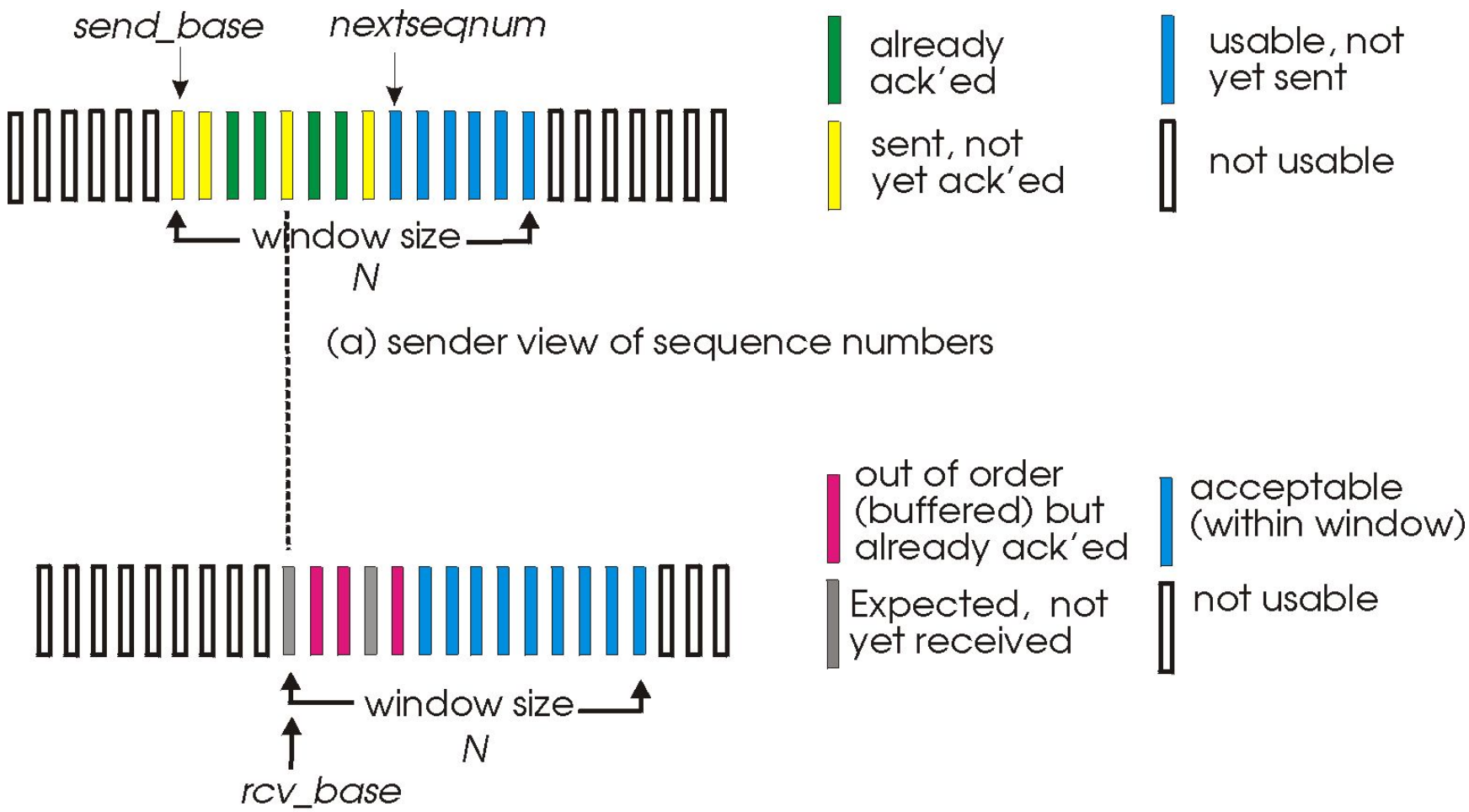


Window size = 4 pkt

Selective Repeat

- Bên nhận ack riêng cho mọi gói tin nhận đúng
  - Đưa gói tin vào vùng đệm nếu cầu, có thể sắp thứ tự chuyển lên lớp trên
- Bên gửi chỉ gửi lại gói tin khi không nhận ACK
  - Bộ đếm thời gian bên gửi cho mỗi gói tin không được ACK
- Cửa sổ bên nhận
  - N seq # liên tục
  - Giới hạn seq #s gửi, gói tin không ACK

Selective repeat: Cửa sổ bên gửi, bên nhận



Selective repeat

Bên gửi

Dữ liệu từ trên:

- Nếu có seq # tiếp trong cửa sổ, gửi gói tin

timeout(n):

- Gửi lại gói tin n, khởi tạo lại bộ đếm thời gian

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

- Đánh dấu gói tin n đã nhận
- Nếu n gói tin không được ACK nhỏ nhất, if n smallest unACKed pkt, chuyển cơ sở của cửa sổ tới seq # không được ACK tiếp

Bên nhận

Gói tin n [rcvbase, rcvbase+N-1]

- Gửi ACK(n)
- Không đúng thứ tự: vùng đệm
- Đúng thứ tự: chuyển lên (cũng có thể đưa vào vùng đệm, xếp thứ tự), cửa sổ chuyển tiếp tới gói tin đã nhận tiếp

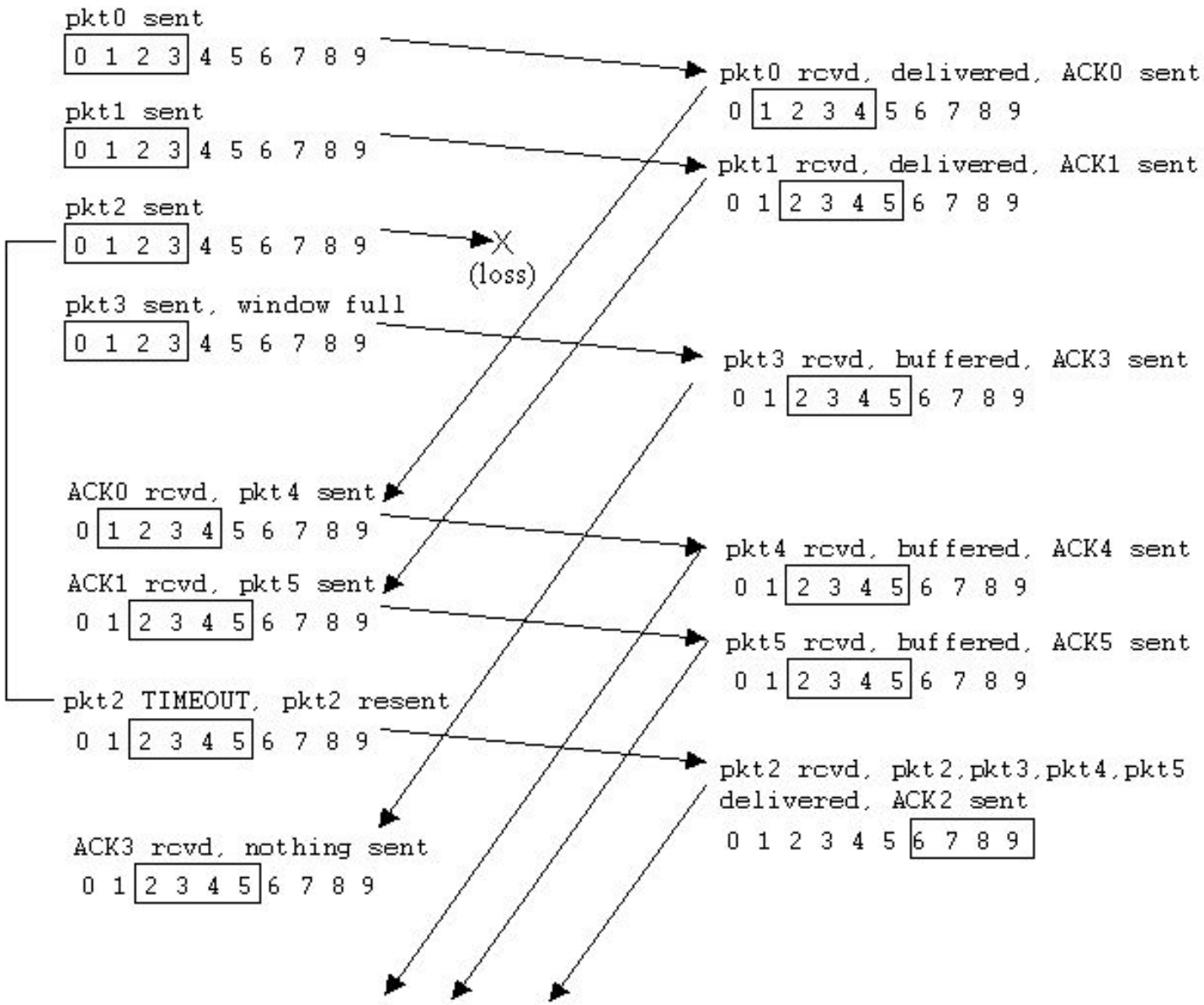
pkt n [rcvbase-N,rcvbase-1]

- ACK(n)

Nếu không:

- Bỏ qua

Selective repeat



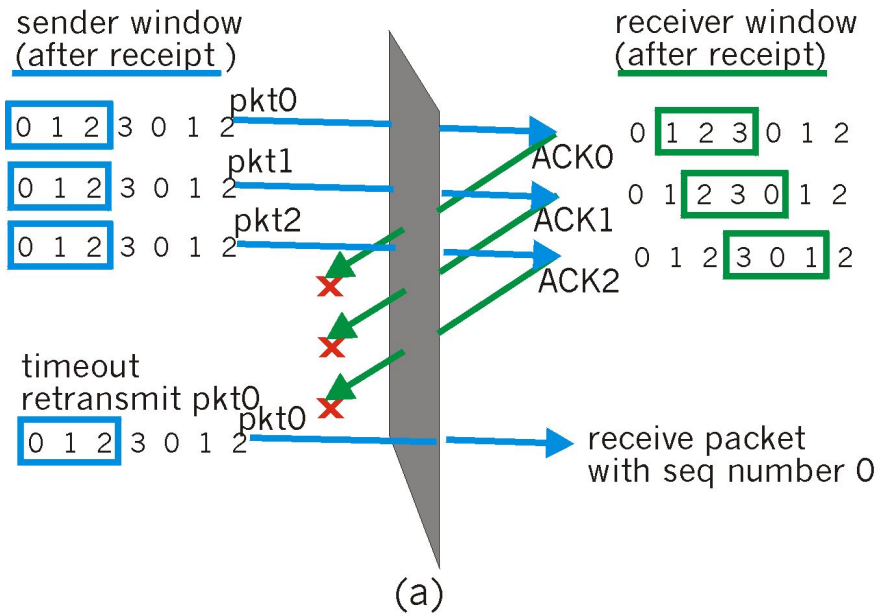
Selective repeat

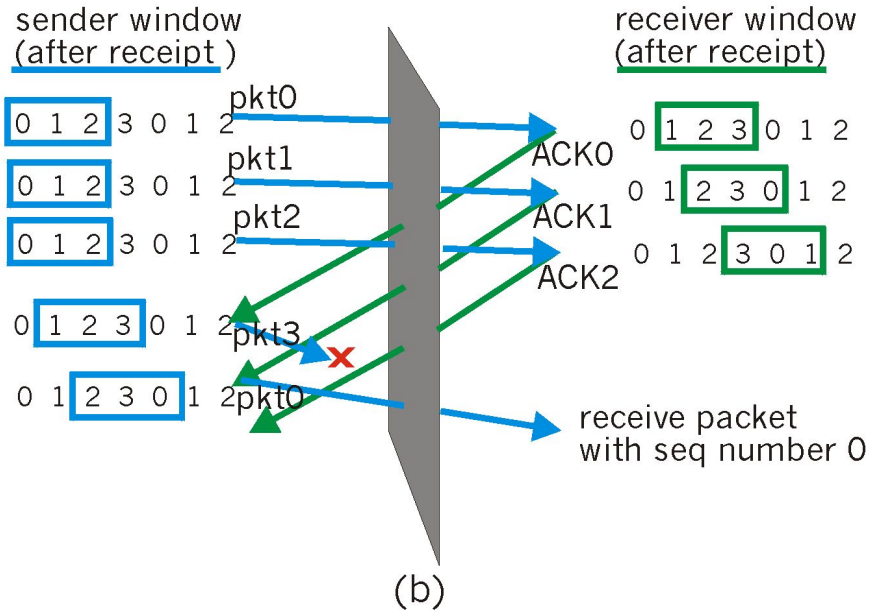
Ví dụ:

- seq #'s: 0, 1, 2, 3
- Kích thước cửa sổ=3

- Bên nhận thấy không có sự khác nhau trong 2 kịch bản
- Chuyển không đúng dữ liệu như mới (a)

Q: Quan hệ giữa kích thước seq# và kích thước cửa sổ?



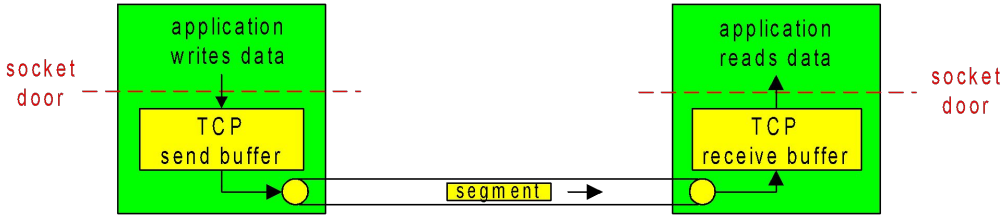


Chương 4: Tầng giao vận

- 4.1 Các dịch vụ tầng giao vận
- 4.2 Multiplexing và demultiplexing
- 4.3 Dịch vụ không hướng kết nối: UDP
- 4.4 Các nguyên tắc của truyền dữ liệu tin cậy
- 4.5 Dịch vụ hướng kết nối: TCP
  - Cấu trúc segment
  - Truyền dữ liệu tin cậy
  - Điều khiển luồng
  - Quản lý kết nối
- 4.6 Các nguyên tắc của điều khiển tắc nghẽn
- 4.7 Điều khiển tắc nghẽn của TCP

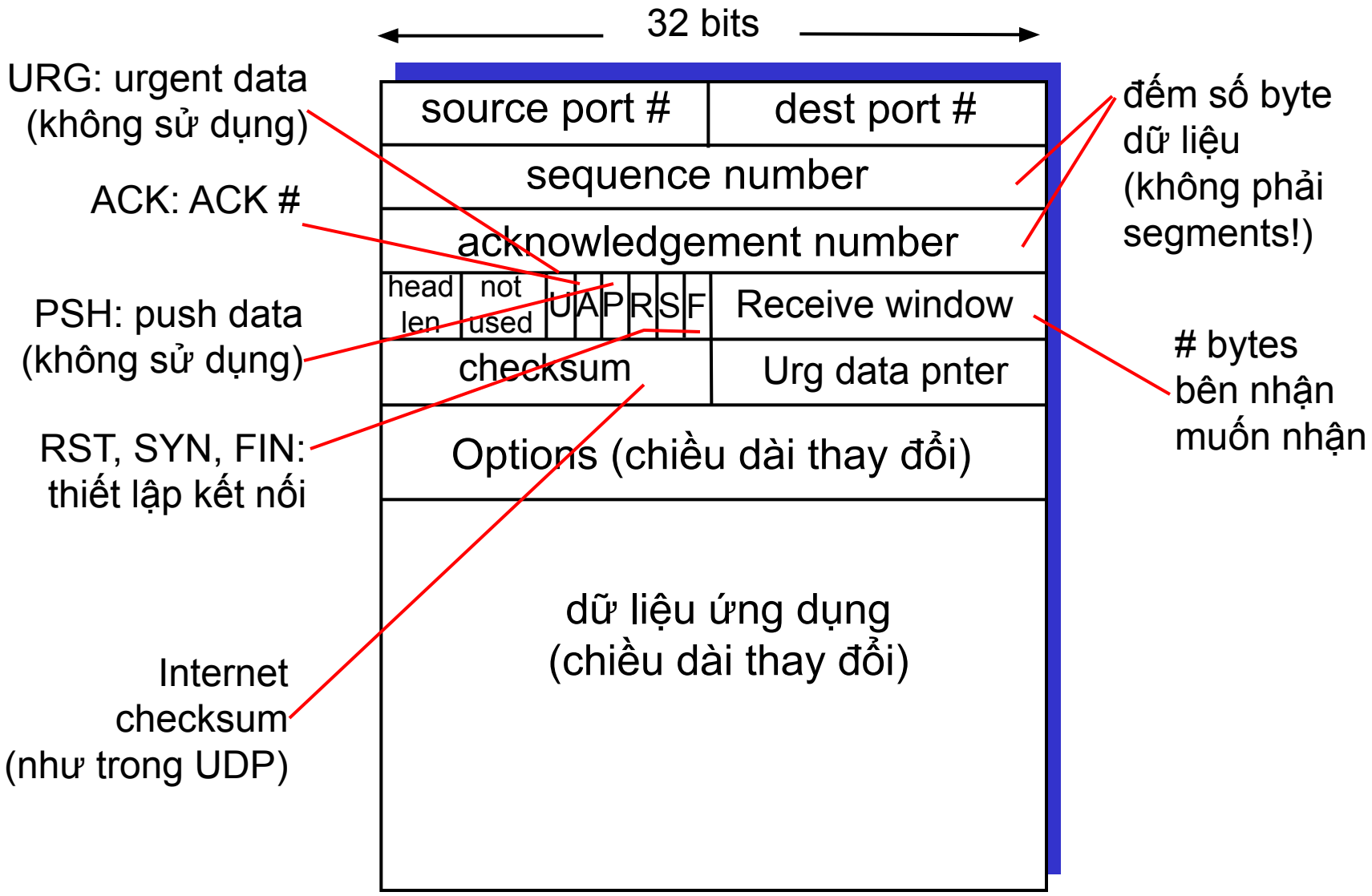
TCP: Tổng quan RFC: 793, 1122, 1323, 2018, 2581

- Point-to-point:
  - Một bên gửi, một bên nhận
- Tin cậy, có thứ tự
- Pipeline:
  - Điều khiển tắc nghẽn và điều khiển luồng của TCP thiết lập giá trị kích thước cửa sổ
- *Vùng đệm gửi và nhận*



- Dữ liệu truyền song công:
  - Luồng dữ liệu truyền hai chiều trên cùng một kết nối
  - MSS: maximum segment size
- Hướng kết nối:
  - Bắt tay (trao đổi các bản tin điều khiển), bên gửi khởi đầu
- Điều khiển luồng:
  - Bên gửi không gửi quá khả năng bên nhận

Cấu trúc của TCP segment



TCP seq # và ACK

Seq. #:

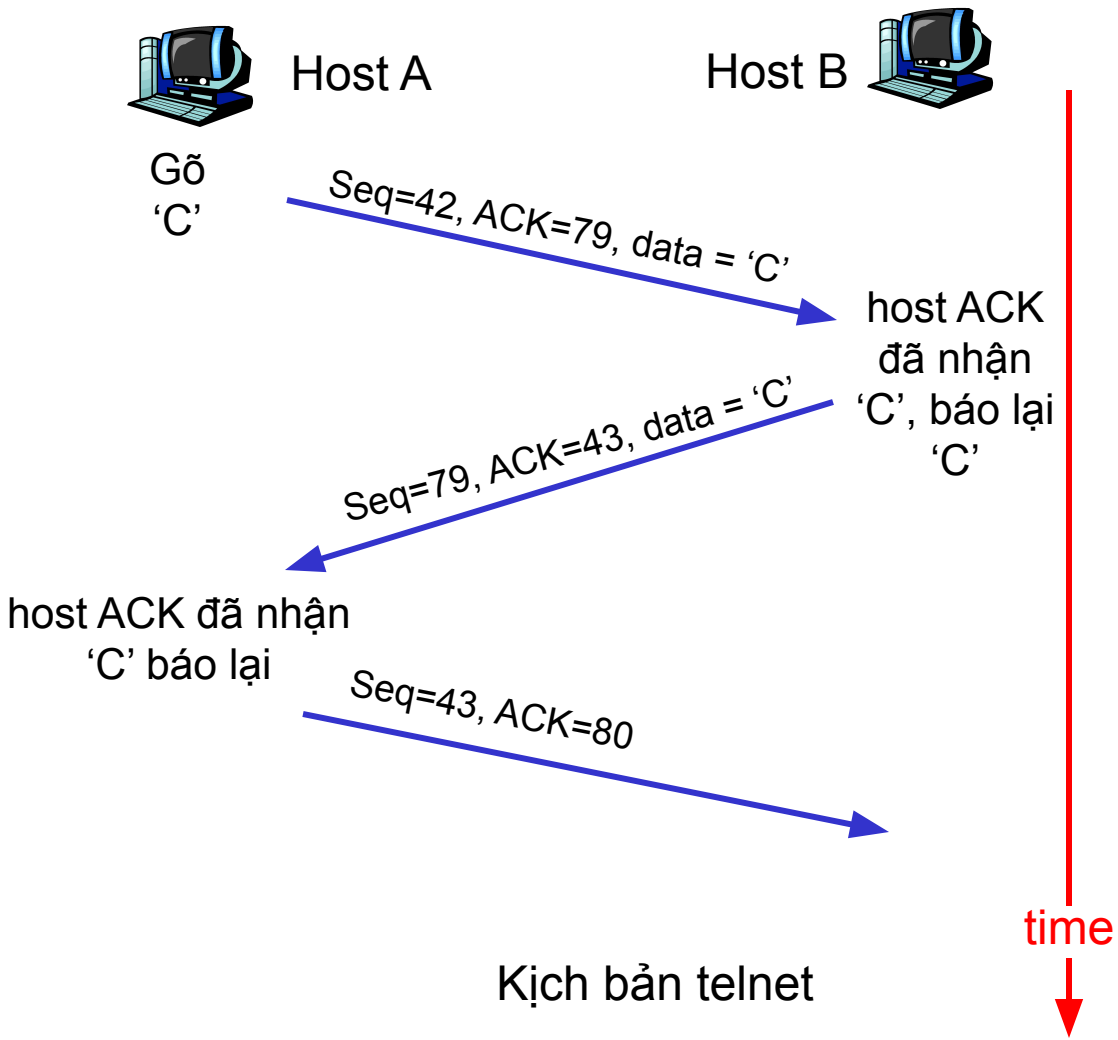
- Giá trị của luồng byte của byte đầu tiên trong dữ liệu của segment

ACK:

- seq # của byte tiếp theo mong nhận
- ACK tích lũy

Q: Bên nhận điều khiển các segment không đúng thứ tự

- A: Chuẩn không chỉ rõ, tùy thuộc vào cài đặt cụ thể



Kịch bản telnet

RTT và Timeout trong TCP

Q: Thiết lập giá trị timeout của TCP?

- Lớn hơn RTT
  - RTT thay đổi
- Quá nhỏ: timeout sớm
  - Không cần thiết truyền lại
- Quá lớn: xử lý chậm các segment bị mất

Q: Ước lượng RTT?

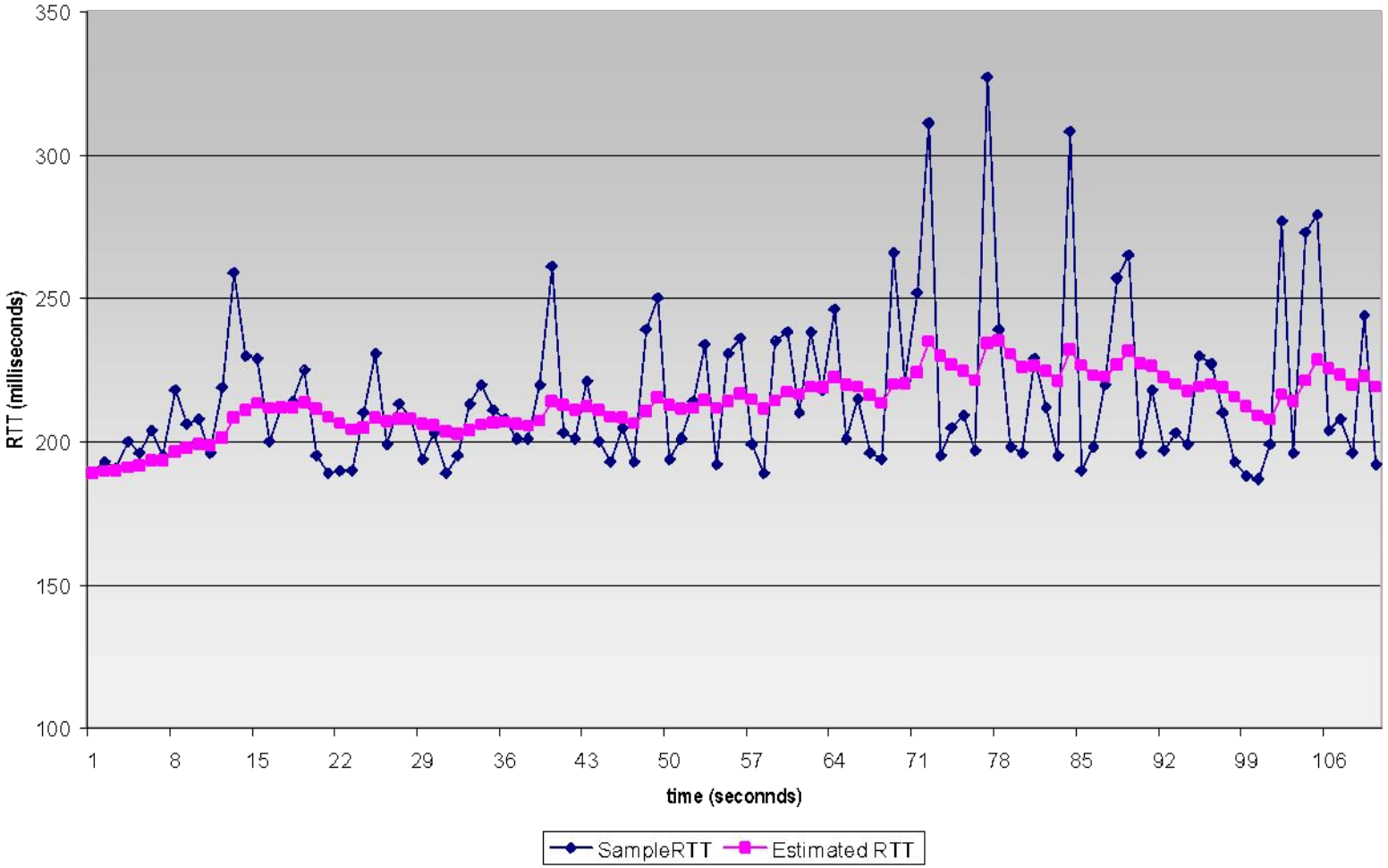
- **SampleRTT**: đo thời gian từ truyền segment tới khi ACK được nhận
  - Bỏ qua truyền lại
- **SampleRTT** thay đổi, ước lượng RTT chính xác hơn
  - Giá trị trung bình của nhiều giá trị đo gần đó

RTT và timeout trong TCP

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

- Giá trị thường dùng: **α** = 0.125

Ví dụ ước lượng RTT



RTT và timeout của TCP

Thiết lập timeout

- **EstimatedRTT** cộng giới hạn an toàn
  - Sự thay đổi lớn của **EstimatedRTT ->** giá trị lề an toàn lớn
- Ước lượng SampleRTT kế thừa từ EstimatedRTT:

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

(thường, β = 0.25)

Rồi thiết lập timeout:

TimeoutInterval = EstimatedRTT + 4*DevRTT

Chương 3: Tầng giao vận

- 3.1 Các dịch vụ tầng giao vận
- 3.2 Multiplexing và demultiplexing
- 3.3 Dịch vụ không hướng kết nối: UDP
- 3.4 Các nguyên tắc của truyền dữ liệu tin cậy
- 3.5 Dịch vụ hướng kết nối: TCP
  - Cấu trúc segment
  - Truyền dữ liệu tin cậy
  - Điều khiển luồng
  - Quản lý kết nối
- 3.6 Các nguyên tắc của điều khiển tắc nghẽn
- 3.7 Điều khiển tắc nghẽn của TCP

Truyền dữ liệu tin cậy của TCP

- TCP tạo dịch vụ rdt trên dịch vụ không tin cậy của IP
- Pipelined segment
- ACK tích lũy
- Truyền lại khi:
  - Có sự kiện timeout
  - Lặp ack
- Xét trường hợp bên gửi:
  - Bỏ qua điều khiển luồng, điều khiển tắc nghẽn

Các sự kiện của bên gửi TCP

Nhận dữ liệu từ ứng dụng:

- Tạo segment với seq #
- seq # là giá trị luồng byte của byte đầu tiên trong segment
- Khởi tạo bộ đếm thời gian
- Chuyển segment tới IP
- Tính NextSeqNum

Timeout:

- Truyền lại segment bị quá hạn
- Tính timeout interval cho segment truyền lại
- Khởi tạo lại bộ đếm thời gian

Nhận Ack:

- Nếu segment trước đó chưa được ACK
  - Cập nhật để biết đã ack
- Nếu segment trước đó đã ACK
  - Tăng bộ đếm ACK lặp, lặp 3 lần thì truyền lại

Bên gửi TCP (đơn giản)

```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum
loop (forever) {
  switch(event)
  event: Nhận được dữ liệu từ tầng ứng dụng
      Tạo TCP segment với giá trị sequence NextSeqNum
      Khởi động timer cho segment NextSegNum
      Chuyển segment tới IP
      NextSeqNum = NextSeqNum + length(data)

   event: timer quá hạn cho segment có sequence number = y
Truyền lại segment có sequence number = y
Tính timeout interval cho segment y
      Khởi động timer cho segment y

   event: Nhận được ACK, giá trị của trường ACK: y
      if (y > SendBase) {
     Bỏ timer của tất cả các segment có sequence number < y
          SendBase = y
} else { /* ACK lặp */
     Tăng số ACK lặp của segment y
          if (số lần ACK lặp của segment y == 3) {
      Truyền lại segment với sequence number = y
                Khởi động timer cho segment y
          }
 }
```

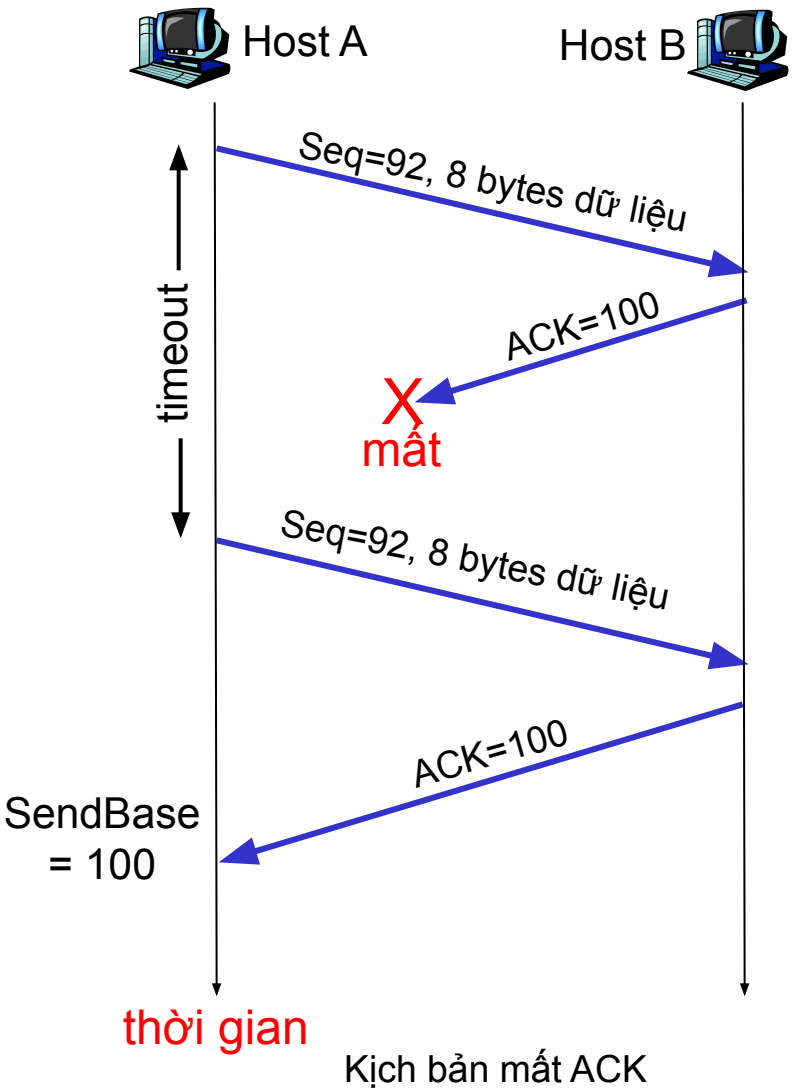
Giải thích:

- SendBase-1: byte được ack tích lũy cuối

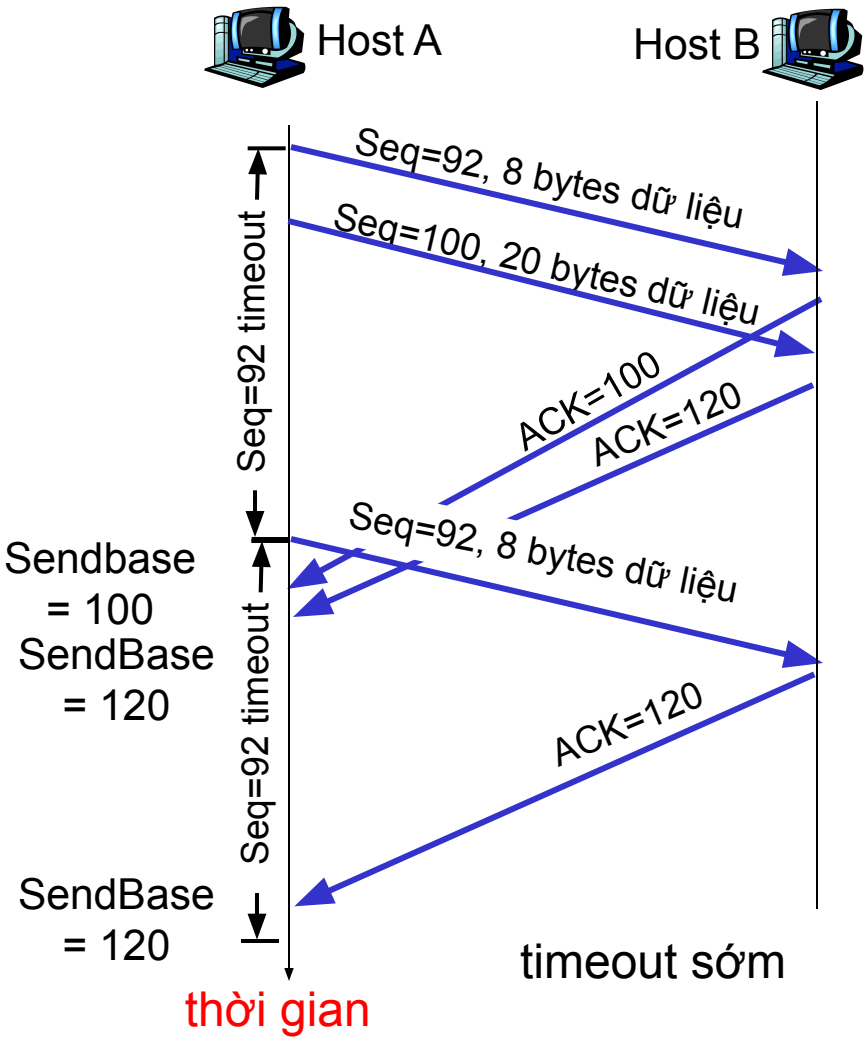
Ví du:

- SendBase-1 = 71; y= 73, rcvr muốn 73+ ; y > SendBase, vì thế dữ liệu mới được ack

TCP: Kịch bản truyền lại

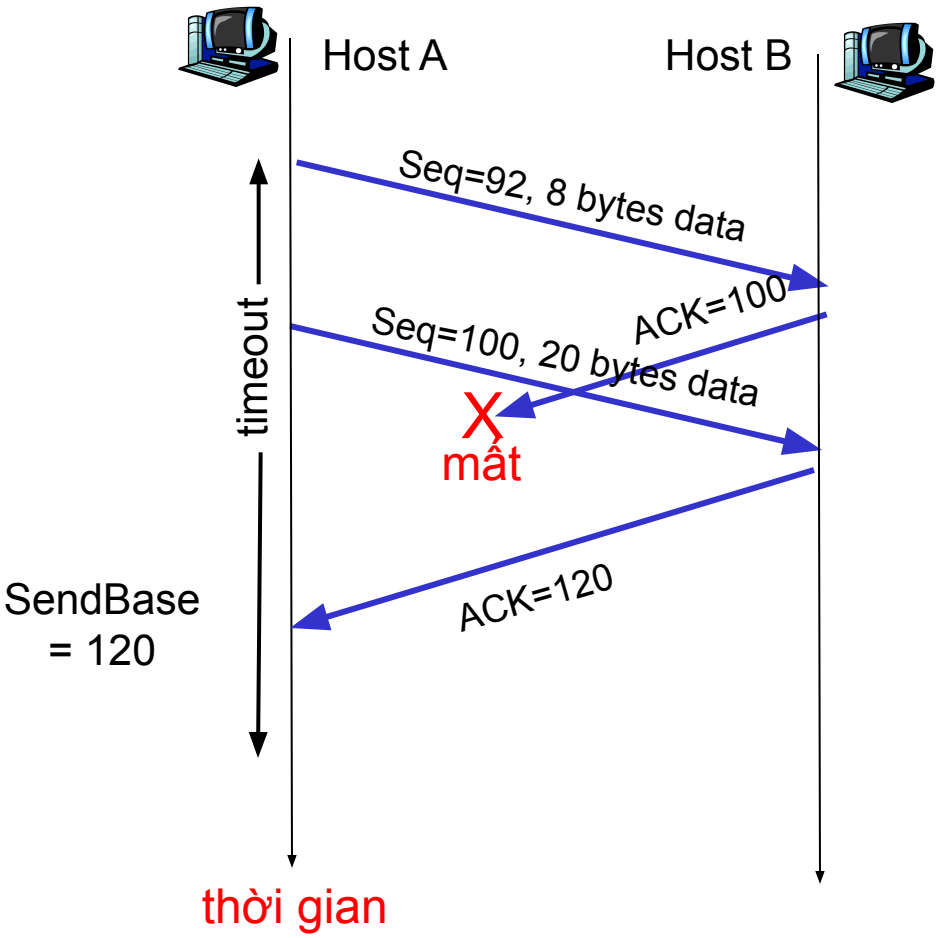


Kịch bản mất ACK



Kịch bản không truyền lại segment khi ACK đến trước timeout

Kịch bản truyền lại



Kịch bản ACK tích lũy

Truyền dữ liệu tin cậy của TCP: GBN hay Selective Repeat

Giống GBN:

- ACK tích lũy
- Bên gửi của TCP chỉ cần duy trì
  - Sequence number nhỏ nhất của gói tin đã gửi, chưa được ack (sendbase)
  - Sequence number của byte tiếp theo sẽ gửi (nextseqnum)

Khác GBN:

- Timeout của segment có sequence number là n chỉ gửi lại segment n
- RFC 2018 - TCP Selective Acknowledgment Options

Chương 4: Tầng giao vận

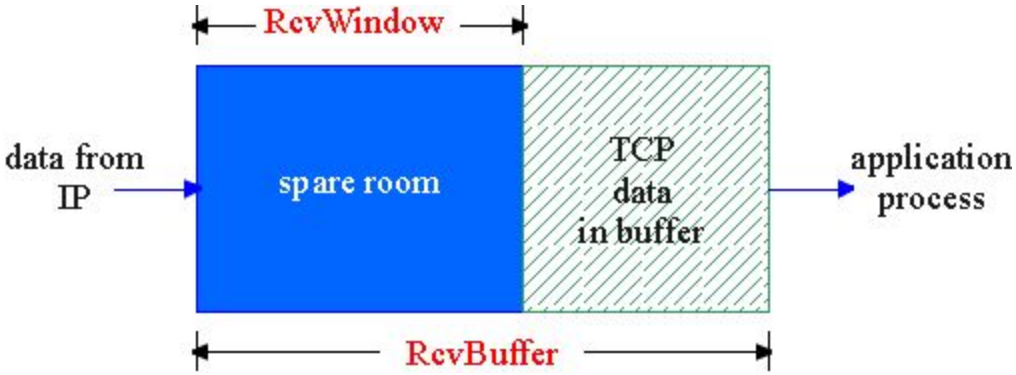
- 4.1 Các dịch vụ tầng giao vận
- 4.2 Multiplexing và demultiplexing
- 4.3 Dịch vụ không hướng kết nối: UDP
- 4.4 Các nguyên tắc của truyền dữ liệu tin cậy
- 4.5 Dịch vụ hướng kết nối: TCP
  - Cấu trúc segment
  - Truyền dữ liệu tin cậy
  - Điều khiển luồng
  - Quản lý kết nối
- 4.6 Các nguyên tắc của điều khiển tắc nghẽn
- 4.7 Điều khiển tắc nghẽn của TCP

Điều khiển luồng TCP

- Bên nhận của kết nối TCP có buffer nhận:

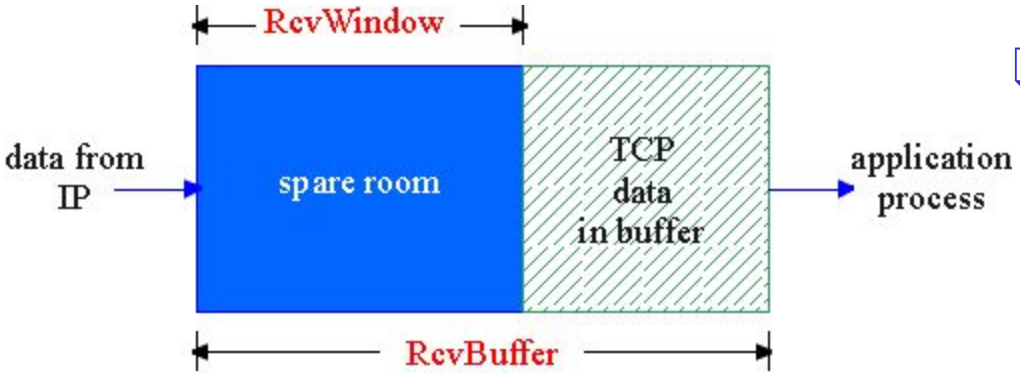
Điều khiển luồng

Bên gửi không gửi làm tràn vùng đệm bên nhận: truyền quá nhiều, quá nhanh



- Dịch vụ tương ứng tốc độ: tương ứng tốc độ gửi với tốc độ bên nhận

Điều khiển luồng của TCP



(Giả sử bên nhận bỏ segment không đúng thứ tự)

- Không gian còn thừa trong buffer
  - = RcvWindow
  - = RcvBuffer-(LastByteRcvd – LastByteRead)
- RcvWindow = 0 ?

- Bên nhận thông tin về không gian còn thừa trong giá trị của **`RcvWindow`** trong segment
- Bên gửi hạn chế dữ liệu chưa ACK theo **`RcvWindow`**
  - Đảm bảo buffer nhận không bị tràn
    - LastByteSent - LastByteAcked <= RcvWindow

Chương 4: Tầng giao vận

- 4.1 Các dịch vụ tầng giao vận
- 4.2 Multiplexing và demultiplexing
- 4.3 Dịch vụ không hướng kết nối: UDP
- 4.4 Các nguyên tắc của truyền dữ liệu tin cậy
- 4.5 Dịch vụ hướng kết nối: TCP
  - Cấu trúc segment
  - Truyền dữ liệu tin cậy
  - Điều khiển luồng
  - Quản lý kết nối
- 4.6 Các nguyên tắc của điều khiển tắc nghẽn
- 4.7 Điều khiển tắc nghẽn của TCP

Quản lý kết nối của TCP

Thiết lập kết nối:

Nhắc lại: Bên gửi, bên nhận của TCP thiết lập kết nối trước khi trao đổi dữ liệu

- Khởi tạo giá trị:
  - seq. #
  - buffer, thông tin điều khiển luồng (ví dụ: **RcvWindow**)
- *Client:* khởi tạo kết nối

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

- *Server:* liên lạc bởi client

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

Bắt tay 3 đường:

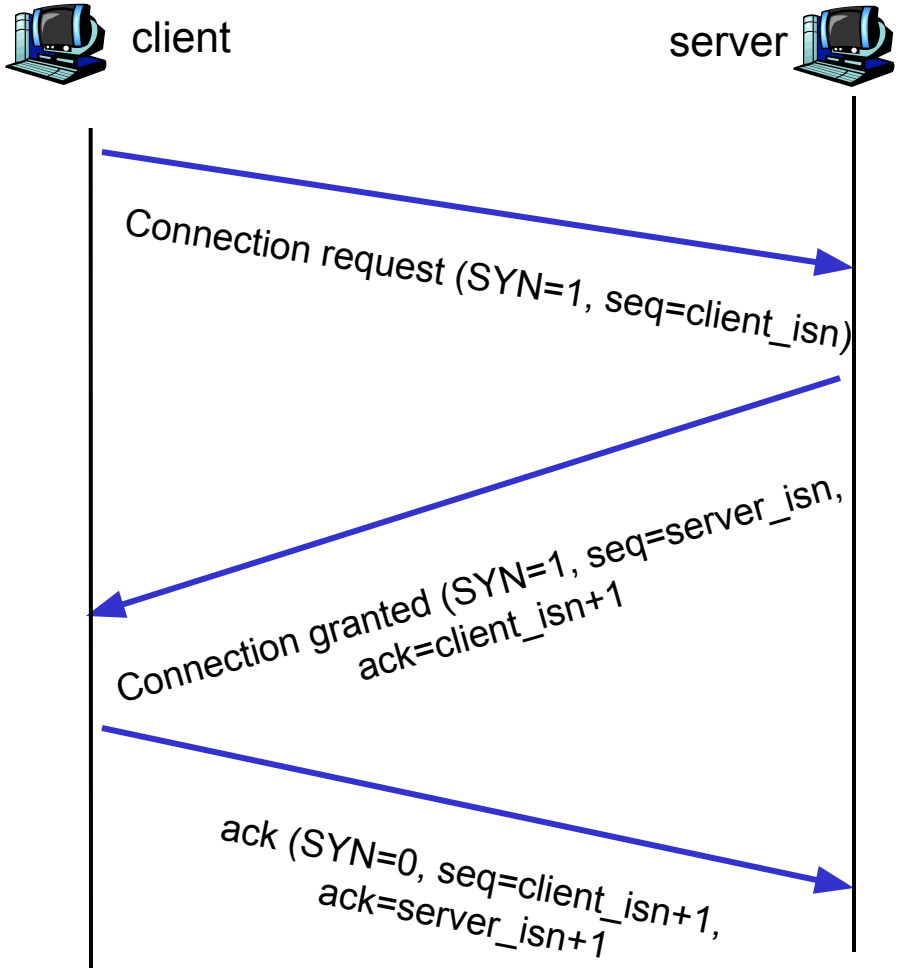
Bước 1: Client gửi TCP SYN segment tới server
- Chỉ định seq # ban đầu
- Không có dữ liệu

Bước 2: Server nhận SYN, trả lời với SYNACK segment
- Server cấp phát buffer
- Server khởi tạo seq. #

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

Quản lý kết nối của TCP

Thiết lập kết nối:



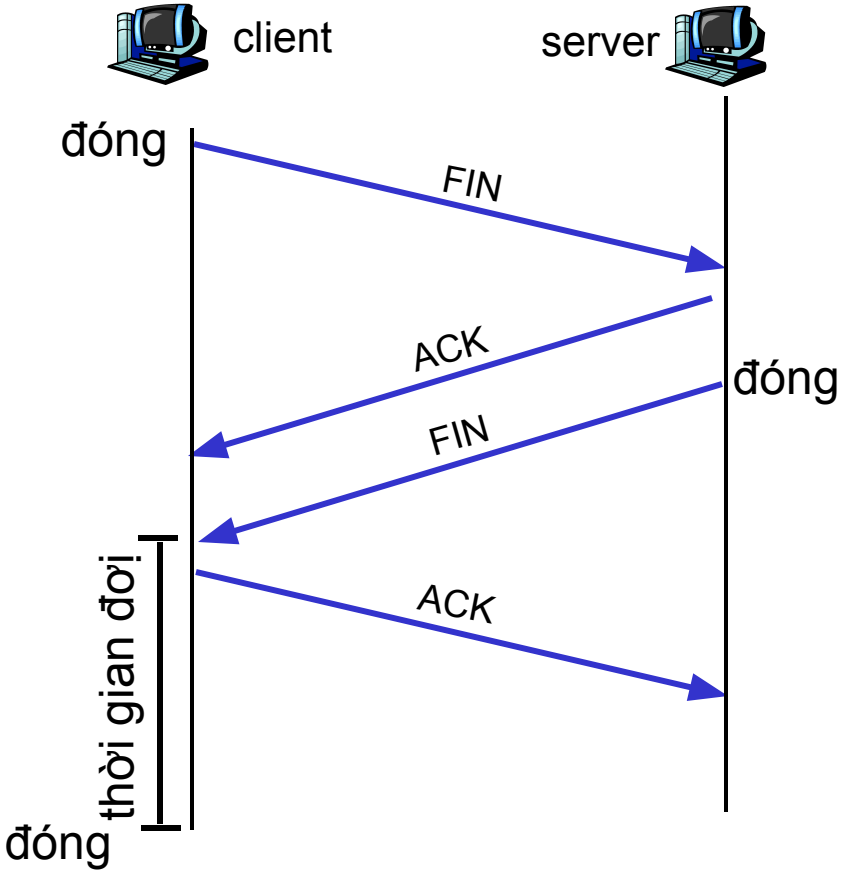
Quản lý kết nối của TCP

Đóng kết nối:

client đóng socket:
clientSocket.close();

Bước 1: client gửi TCP FIN tới server

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

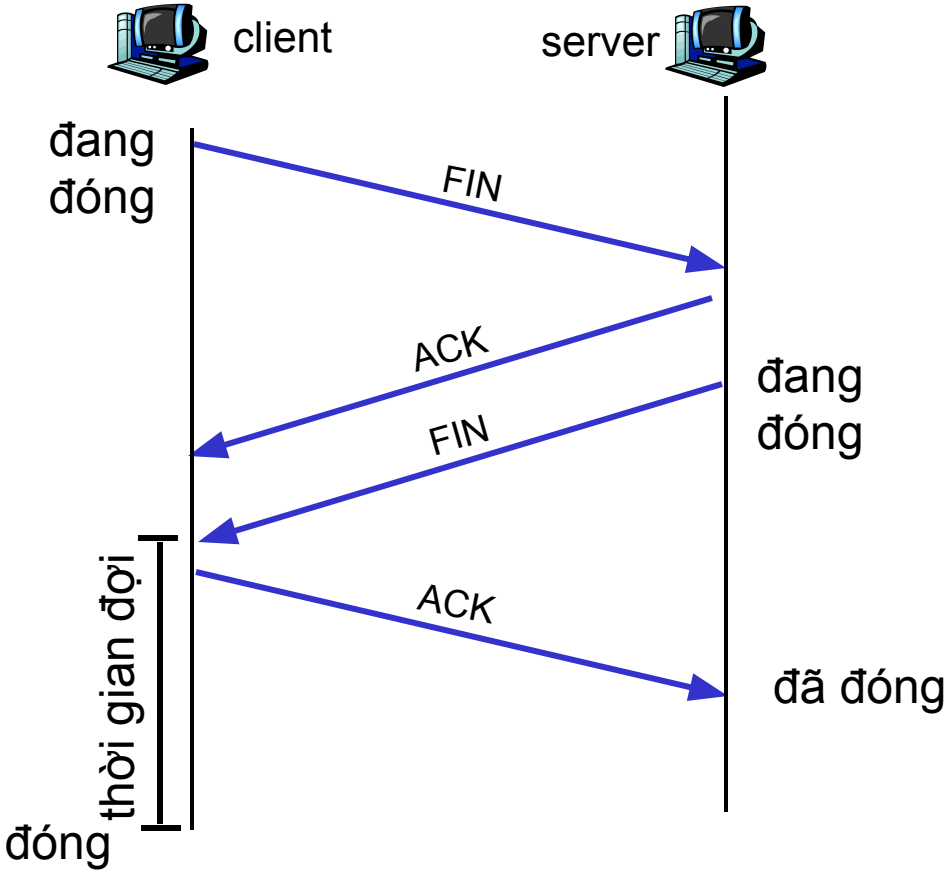


Quản lý kết nối của TCP

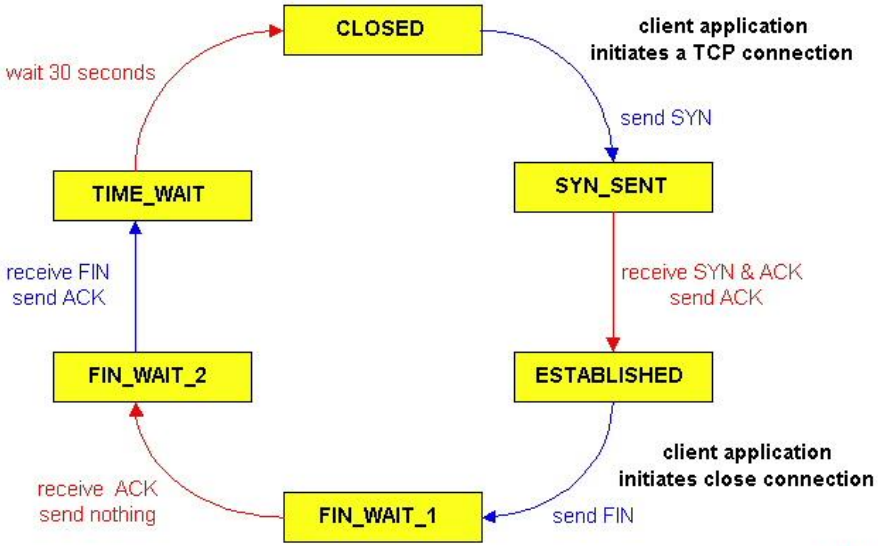
Bước 3: client nhận FIN, trả lời bằng ACK.

- Thời gian đợi: trả lời bằng ACK báo đã nhận FIN

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

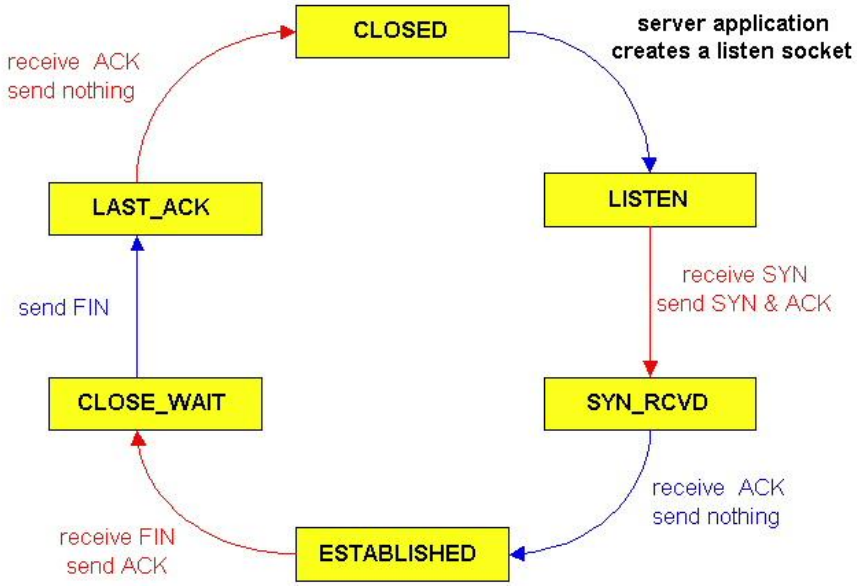


Quản lý kết nối của TCP



Chu kỳ hoạt động TCP client

Chu kỳ hoạt động TCP server



Chương 4: Tầng giao vận

- 4.1 Các dịch vụ tầng giao vận
- 4.2 Multiplexing và demultiplexing
- 4.3 Dịch vụ không hướng kết nối: UDP
- 4.4 Các nguyên tắc của truyền dữ liệu tin cậy
- 4.5 Dịch vụ hướng kết nối: TCP
  - Cấu trúc segment
  - Truyền dữ liệu tin cậy
  - Điều khiển luồng
  - Quản lý kết nối
- 4.6 Các nguyên tắc của điều khiển tắc nghẽn
- 4.7 Điều khiển tắc nghẽn của TCP

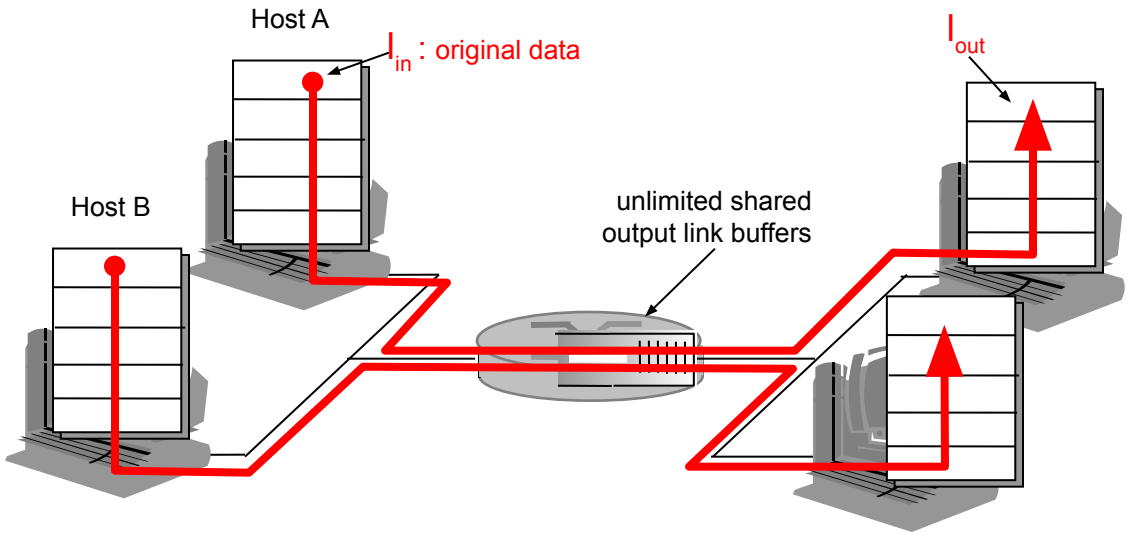
Nguyên tắc điều khiển tắc nghẽn

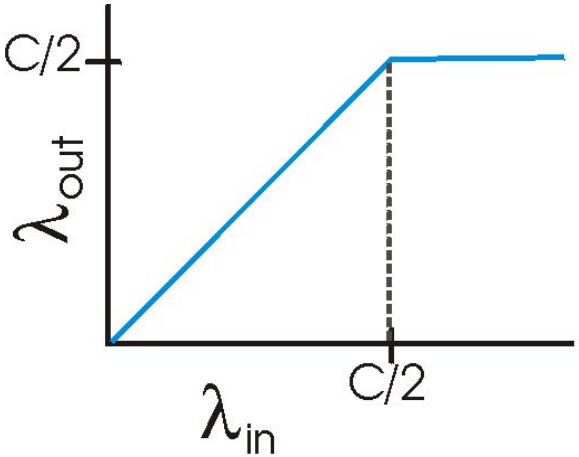
Tắc nghẽn:

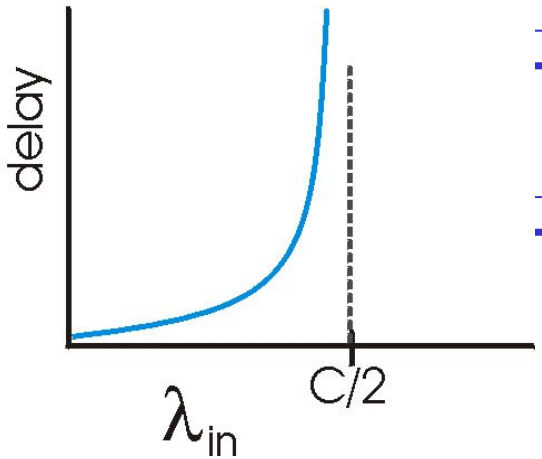
- Quá nhiều nguồn gửi quá nhiều dữ liệu nhanh quá khả năng điều khiển của mạng
- Khác điều khiển luồng
- Đặc điểm:
  - Mất gói tin (tràn buffer tại router)
  - Độ trễ tăng (xếp hàng tại buffer của router)

Nguyên nhân, tác hại của tắc nghẽn: Kịch bản 1

- Hai đối tượng gửi, hai đối tượng nhận
- Một router, vùng đệm không giới hạn
- Không truyền lại



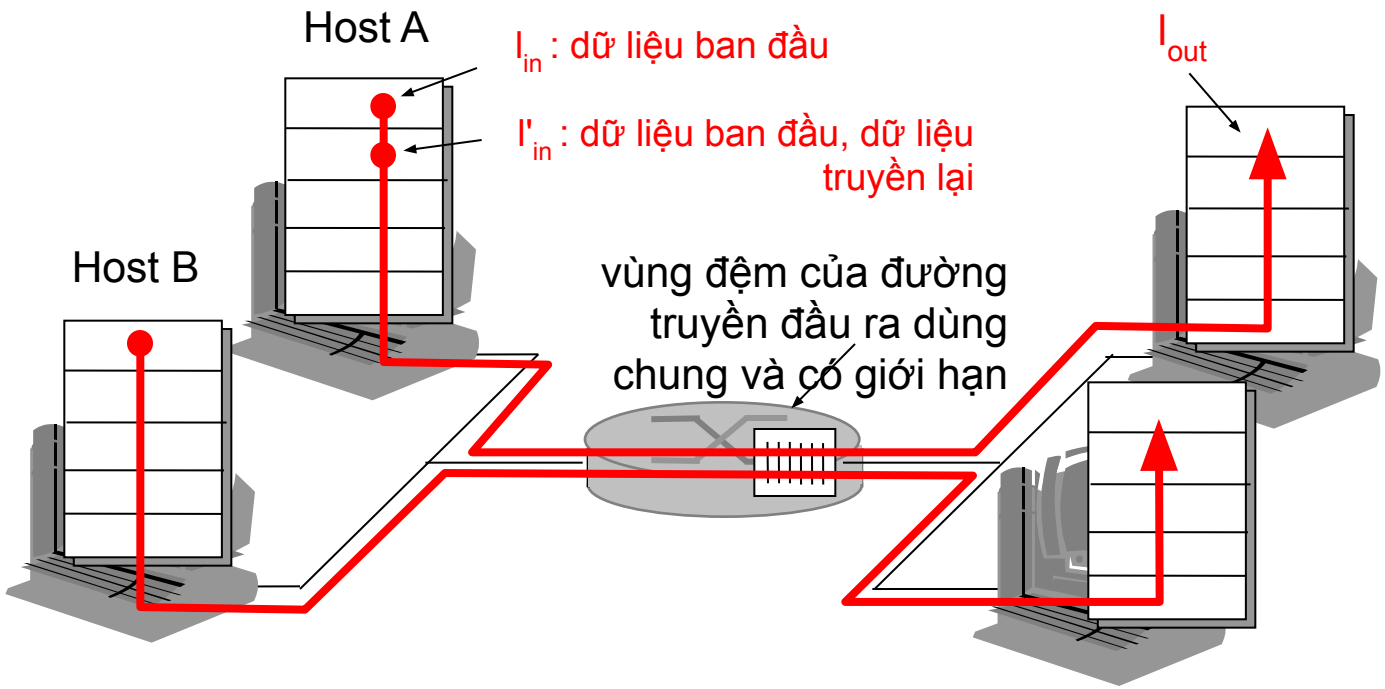




- Độ trễ lớn khi xảy ra tắc nghẽn
- Tối đa thông lượng có thể đạt được

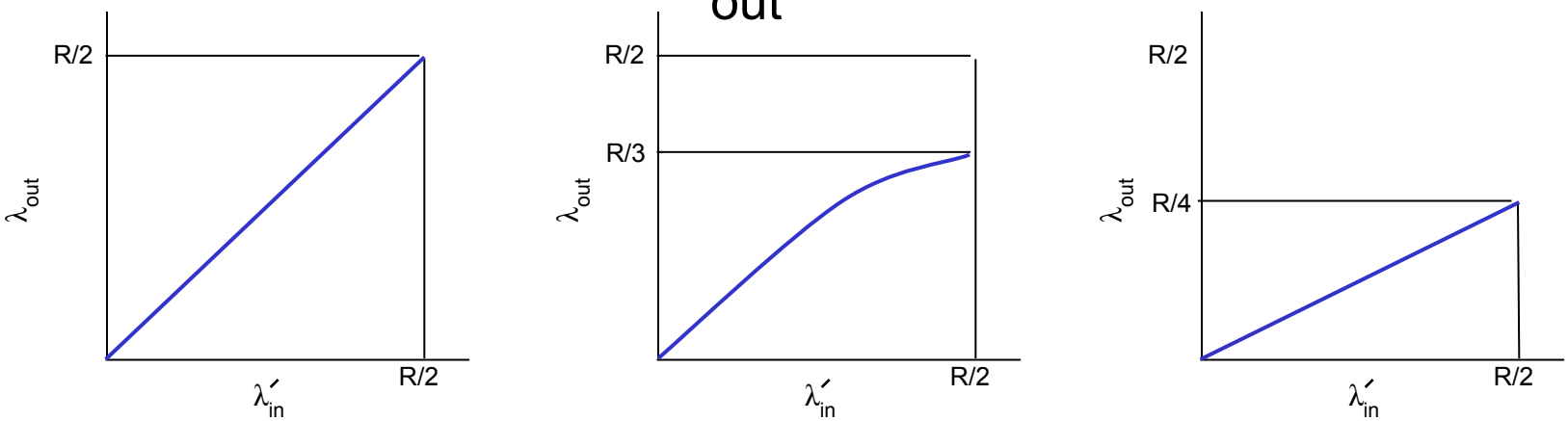
Nguyên nhân, tác hại của tắc nghẽn: Kịch bản 2

- Một router, vùng đệm *giới hạn*
- Bên gửi truyền lại gói tin mất



Nguyên nhân, tác hại của tắc nghẽn: Kịch bản 2

- Luôn luôn: $\lambda_{in} = \lambda_{out}$ (tốt)
- Truyền lại "hoàn hảo": truyền lại chỉ khi mất $\lambda'_{in} > \lambda_{out}$
- Sự truyền lại của gói tin bị trễ (không mất) làm λ'_{in} lớn hơn (trường hợp hoàn hảo) so với λ_{out}



a. b. c.

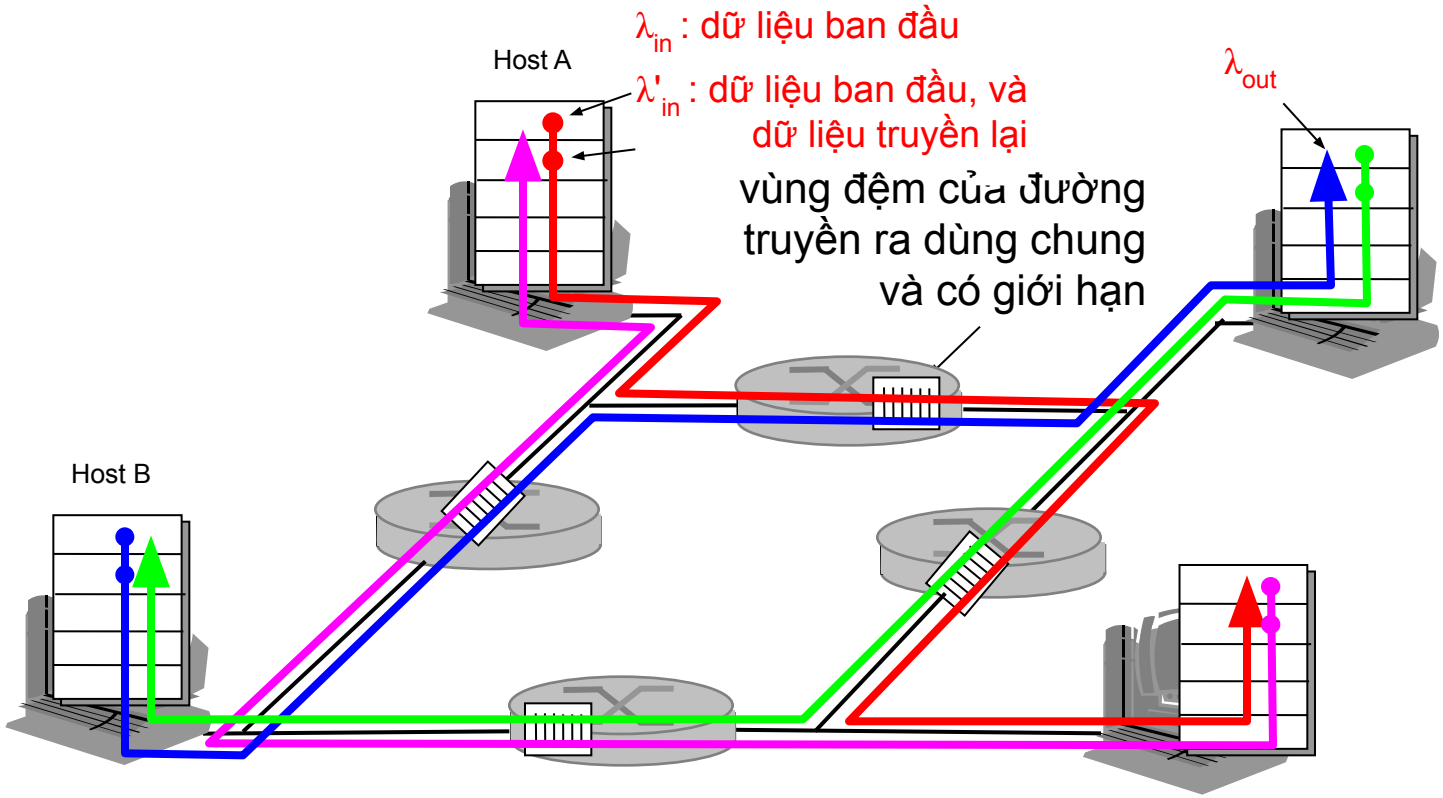
"Tác hại" của tắc nghẽn:

- Xử lý nhiều hơn (truyền lại)
- Truyền lại không cần thiết: đường truyền mang nhiều bản sao chép của gói tin

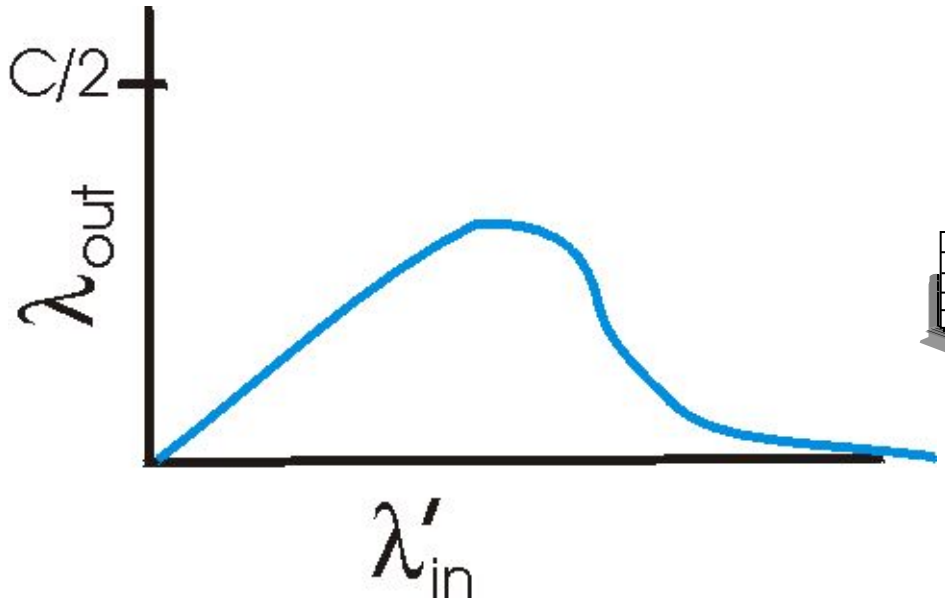
Nguyên nhân, tác hại của tắc nghẽn: Kịch bản 3

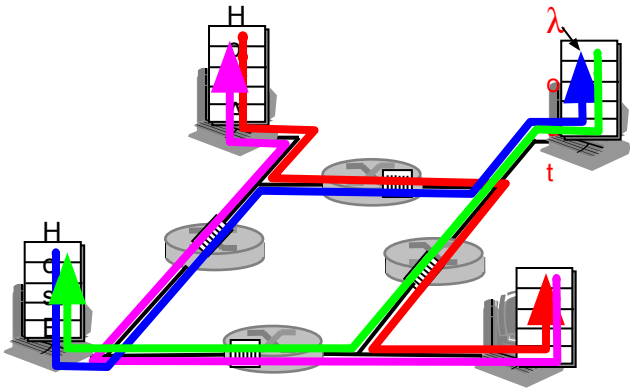
- Bốn đối tượng gửi
- Nhiều đường
- timeout/truyền lại

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



Nguyên nhân, tác hại của tắc nghẽn: Kịch bản 3





Tác hại khác của tắc nghẽn:

- Khi gói tin bị loại bỏ, khả năng truyền đường lên sử dụng cho gói tin đó đã lãng phí

Các cách tiếp cận để điều khiển tắc nghẽn

Hai cách tiếp cận chính để điều khiển tắc nghẽn:

Điều khiển tắc nghẽn cuối-cuối:

- Không có phản hồi chính thức từ mạng
- Tắc nghẽn suy ra từ hệ thống cuối theo dõi mất gói và độ trễ
- Cách tiếp cận sử dụng bởi TCP

Điều khiển tắc nghẽn với sự giúp đỡ của mạng:

- Router cung cấp phản hồi tới hệ thống cuối
  - Một bít chỉ ra tắc nghẽn (SNA, DECbit, TCP/IP, ECN, ATM)
  - Chỉ rõ tốc độ bên gửi nên gửi

Trường hợp nghiên cứu: Điều khiển tắc nghẽn ATM ABR

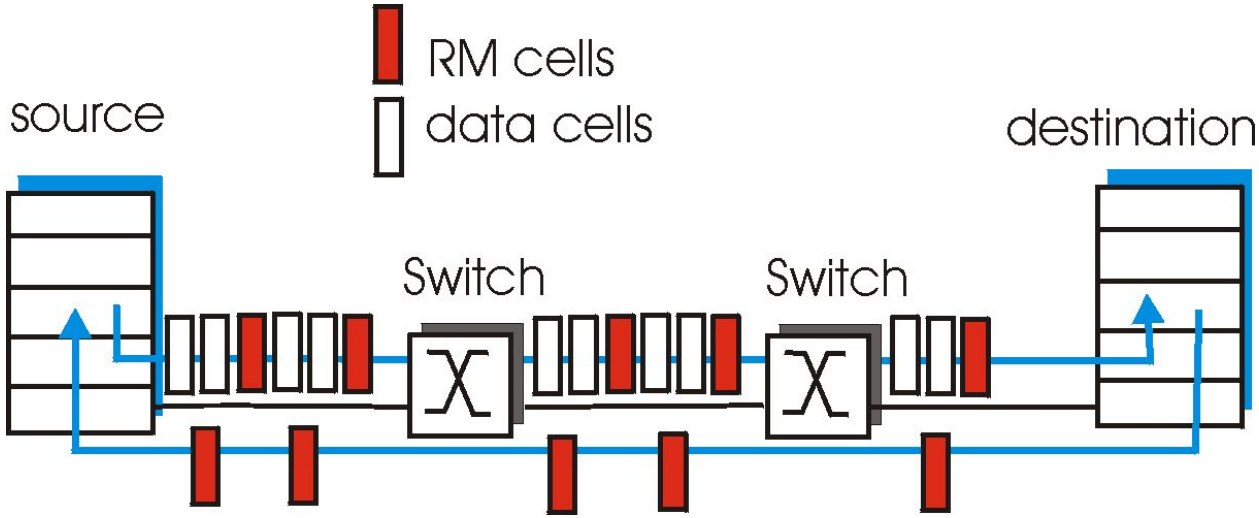
ABR: Available Bit Rate:

- "Dịch vụ co dãn"
- Nếu đường đi của bên gửi chưa đến giới hạn tải:
  - Bên gửi nên sử dụng băng thông khả dụng
- Nếu đường đi của bên gửi bị tắc nghẽn:
  - Bên gửi điều chỉnh tốc độ đảm bảo tối thiểu

RM cell (Resource Management):

- Được gửi bởi bên gửi, rải rác cùng với cell dữ liệu
- Các bít trong RM cell do switch thiết lập (có sự tham gia của mạng)
  - NI bit: không tăng tốc độ (tắc nghẽn nhẹ)
  - CI bit: tắc nghẽn
- RM cell trả về cho bên gửi bởi bên nhận với các bít không thay đổi

Trường hợp nghiên cứu: Điều khiển tắc nghẽn ATM ABR



- Hai byte ER (Explicit Rate) trong RM cell
  - Switch tắc nghẽn có thể giảm giá trị ER trong cell
  - Vì vậy, tốc độ gửi của bên gửi tối thiểu tốc độ hỗ trợ trên đường
- Bít EFCI trong cell dữ liệu: đặt bằng 1 trong switch bị tắc nghẽn
  - Nếu cell dữ liệu, trước cell RM, có EFCI thiết lập, bên gửi thiết lập bít CI trong cell RM trả về

Chương 4: Tầng giao vận

- 4.1 Các dịch vụ tầng giao vận
- 4.2 Multiplexing và demultiplexing
- 4.3 Dịch vụ không hướng kết nối: UDP
- 4.4 Các nguyên tắc của truyền dữ liệu tin cậy
- 4.5 Dịch vụ hướng kết nối: TCP
  - Cấu trúc segment
  - Truyền dữ liệu tin cậy
  - Điều khiển luồng
  - Quản lý kết nối
- 4.6 Các nguyên tắc của điều khiển tắc nghẽn
- 4.7 Điều khiển tắc nghẽn của TCP

Điều khiển tắc nghẽn của TCP

- Điều khiển end-end (không có hỗ trợ của mạng)
- Bên gửi giới hạn truyền:

LastByteSent-LastByteAcked ≤ CongWin

LastByteSent - LastByteAcked <= min{CongWin, RcvWin}

- Gần đúng,

$$\text{rate} = \frac{\text{CongWin}}{\text{RTT}} \text{ Bytes/sec}$$

- **CongWin** thay đổi động, chức năng nhận biết sự tắc nghẽn của mạng

Cách bên gửi nhận biết sự tắc nghẽn?

- Sự kiện mất gói = timeout hoặc lặp lại 3 ack
- Bên gửi TCP giảm tốc độ (**CongWin**) sau sự kiện mất gói

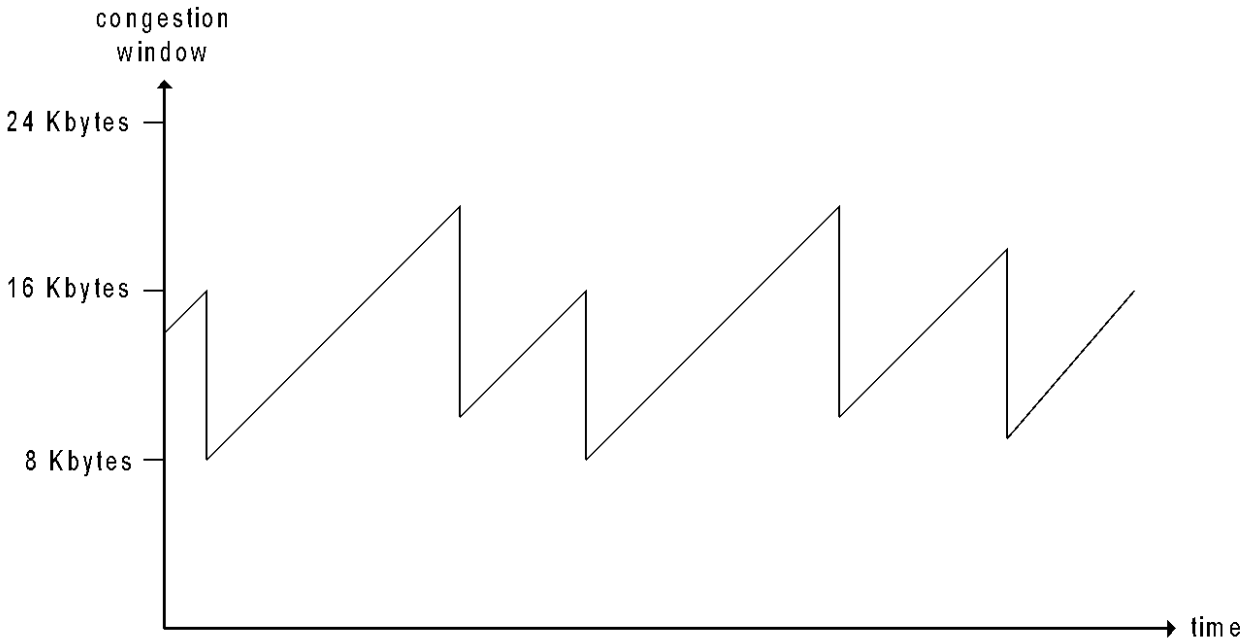
Ba cơ chế:

- AIMD
- slow start
- Không thay đổi sau sự kiện timeout

TCP AIMD

Giảm cấp số nhân: Giảm **CongWin** một nửa sau sự kiện mất gói

Tăng cấp số cộng: Tăng **CongWin** 1 MSS mỗi RTT khi không có sự kiện mất gói



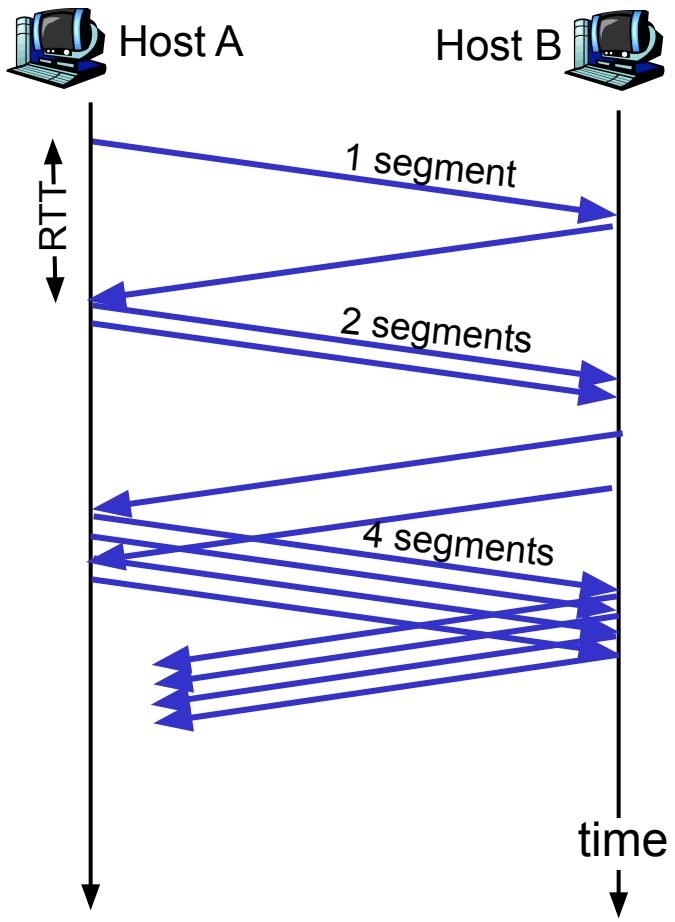
Kết nối TCP trong thời gian dài

TCP Slow Start

- Khi kết nối bắt đầu, **`CongWin`** = 1 MSS
  - Ví dụ: MSS = 500 byte & RTT = 200 msec
  - Tốc độ ban đầu = 20 kbps
- Băng thông khả dụng có thể >> MSS/RTT
  - Mong muốn nhanh tới tốc độ đáng kể
- Khi kết nối bắt đầu, tăng tốc độ nhanh theo hàm mũ cho đến khi có sự kiện mất gói đầu tiên

TCP Slow Start (tiếp)

- Khi kết nối bắt đầu, tăng tốc độ hàm mũ tới khi có sự kiện mất gói đầu tiên:
  - Gấp đôi **CongWin** mỗi RTT
  - Thự hiện bằng cách tăng **CongWin** cho mọi ACK nhận được
- Tóm lai: tốc độ ban đầu là chậm nhưng nhanh chóng tăng theo hàm mũ



Quá trình tinh chỉnh

- Sau 3 ACK lặp lại:
  - **CongWin** giảm một nửa
  - window tăng tuyến tính
- Nhưng sau sự kiện timeout:
  - **CongWin** thay vì thiết lập 1 MSS;
  - window tăng hàm mũ
  - Tới ngưỡng thì giảm tuyến tính

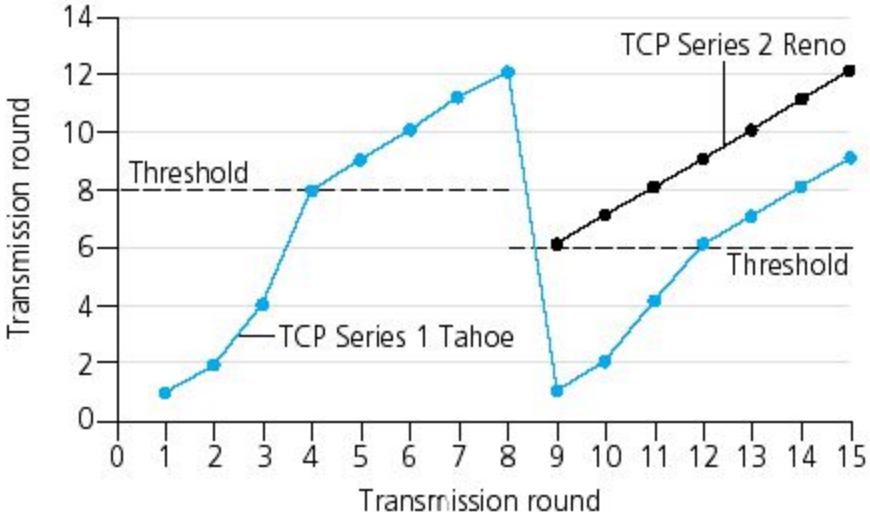
Triết lý:

- 3 ACK lặp chỉ rằng khả năng của mạng chuyển một số segment
- timeout trước 3 ACK lặp là đáng chú ý hơn

Quá trình tinh chỉnh (tiếp)

Q: Khi nào tăng theo số mũ chuyển thành theo tuyến tính?

A: Khi **`CongWin`** có giá trị bằng 1/2 giá trị của nó trước khi timeout.



Thực hiện:

- Variable Threshold
- Tại sự kiện mất gói, Threshold đặt bằng ½ CongWin ngay trước sự kiện mất gói

Tổng kết: Điều khiển tắc nghẽn của TCP

- Khi **`CongWin`** nhỏ hơn **`Threshold`**, bên gửi trong pha slow-start, window lớn theo hàm mũ.
- Khi **`CongWin`** lớn hơn **`Threshold`**, bên gửi trong pha congestion-avoidance, window lớn theo hàm tuyến tính.
- Khi xảy ra lặp 3 ACK, **`Threshold`** đặt bằng **`CongWin/2`** và **`CongWin`** đặt bằng **`Threshold`**.
- Khi timeout xảy ra, **`Threshold`** đặt bằng **`CongWin/2`** và **`CongWin`** đặt bằng 1 MSS.

Điều khiển tắc nghẽn bên gửi của TCP

Sự kiện	Trạng thái	Hành động bên gửi TCP	Giải thích
Nhận ACK cho dữ liệu không được ack trước đó	Slow Start (SS)	CongWin = CongWin + MSS, If (CongWin > Threshold) Đặt trạng thái thành "Congestion Avoidance"	Kết quả bởi gấp đôi CongWin mỗi RTT
Nhận ACK cho dữ liệu không được ack trước đó	Congestion Avoidance (CA)	CongWin = CongWin+MSS * (MSS/CongWin)	Tăng theo cấp số cộng, tăng CongWin lên 1 MSS mỗi RTT
Sự kiện mất gói phát hiện bởi 3 ACK lặp	SS hoặc CA	Threshold = CongWin/2, CongWin = Threshold, Đặt trạng thái thành "Congestion Avoidance"	Nhanh chóng phục hồi, thực hiện tăng cấp số nhân. CongWin sẽ không giảm dưới 1 MSS.
Timeout	SS hoặc CA	Threshold = CongWin/2, CongWin = 1 MSS, Đặt trạng thái thành "Slow Start"	Vào slow start
ACK lặp	SS hoặc CA	Tăng bộ đếm ACK lặp cho segment được ack	CongWin và Threshold không thay đổi

Thông lượng của TCP

- Thông lượng trung bình của TCP từ chức năng của window size và RTT?
  - Bỏ qua slow start
- Cho W là window size khi xảy ra mất gói.
- Khi window là W, thông lượng là W/RTT
- Ngay sau mất gói, window giảm tới W/2, thông lượng W/2RTT.
- Thông lượng trung bình: .75 W/RTT

Tương lai của TCP

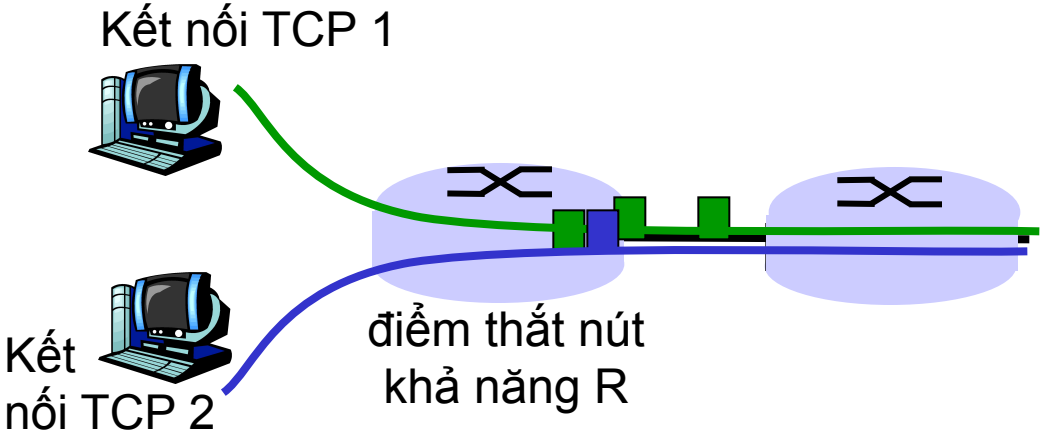
- Ví dụ: 1500 byte segment, 100ms RTT, muốn 10 Gbps thông lượng
- Đòi hỏi window size W = 83,333 segment
- Thông lượng theo tốc độ mất gói:

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

- ➜ $L = 2 \cdot 10^{-10}$
- Phiên bản mới của TCP cho tốc độ cao là cần thiết!

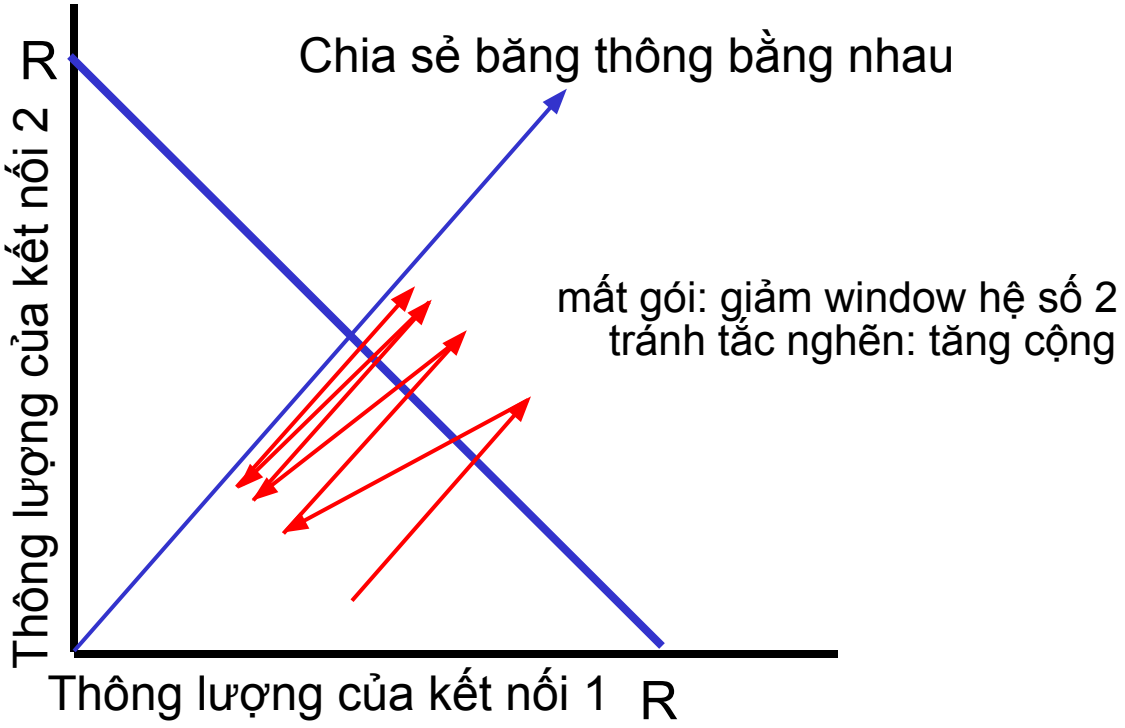
Sự công bằng của TCP

- Mục đích của sự công bằng: Nếu K phiên TCP dùng chung đường truyền thắt nút băng thông R, mỗi đường nên có tốc độ trung bình là R/K



Lý do TCP công bằng

Hai phiên cạnh tranh:



Sự công bằng (tiếp)

Sự công bằng và UDP

- Các ứng dụng đa phương tiện thường không sử dụng TCP
  - Không muốn tốc độ giảm bởi điều khiển tắc nghẽn
- Ứng dụng sử dụng UDP:
  - Đẩy dữ liệu audio/video ở tốc độ hằng số, chấp nhận mất gói

Sự công bằng và các kết nối TCP song song

- Không ngăn chặn ứng dụng mở song song các kết nối giữa 2 host
- Trình duyệt Web thực hiện như trên

Mô hình độ trễ

Q: Thời gian để nhận một đối tượng từ Web server sau khi gửi một yêu cầu?

Bỏ qua tắc nghẽn, độ trễ ảnh hưởng bởi:

- Thiết lập kết nối TCP
- Độ trễ truyền dữ liệu

Giả sử một đường truyền giữa client và server có tốc độ R

- S: MSS (bits)
- O: kích thước đối tượng (bit)
- Không truyền lại (không mất gói, không lỗi)

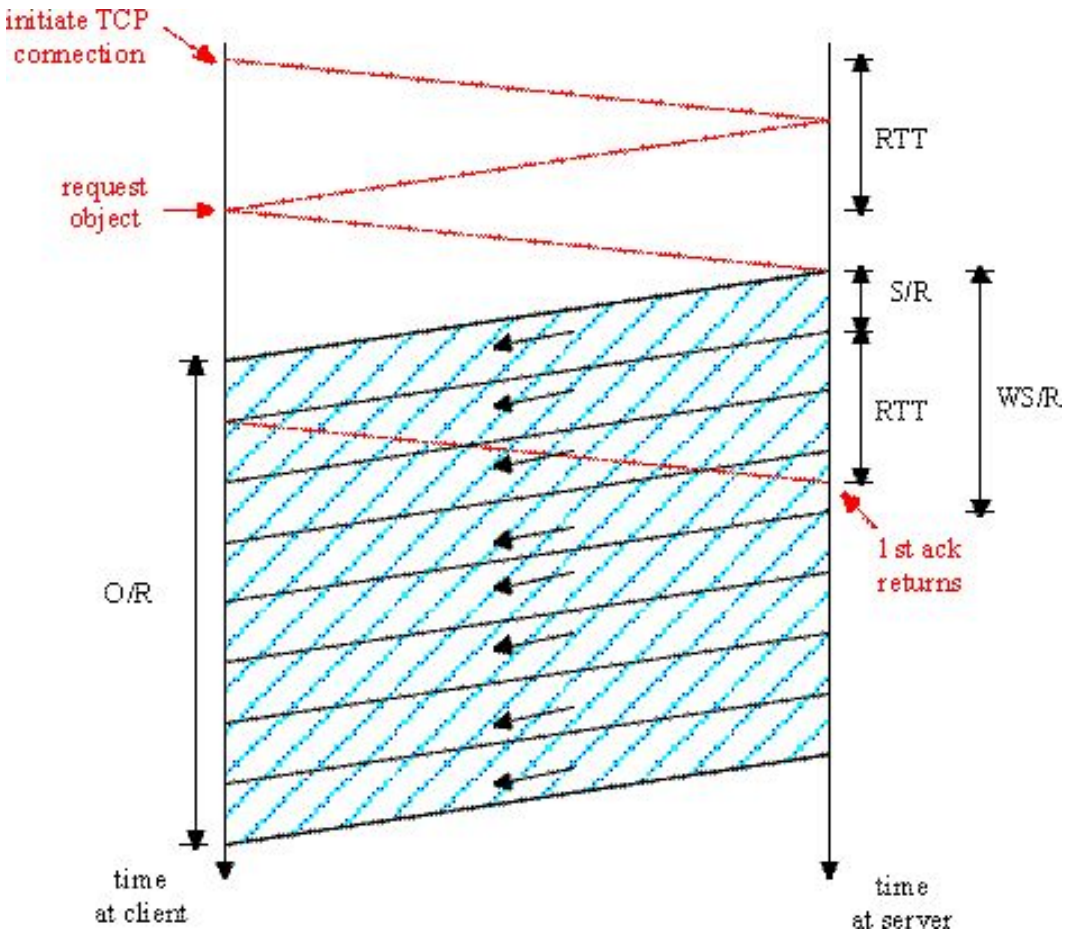
Kích thước cửa sổ:

- Cửa sổ tắc nghẽn cố định, W segment
- Cửa sổ động, mô hình slow start

Cửa sổ tắc nghẽn cố định

Trường hợp 1:

WS/R > RTT + S/R: ACK cho segment đầu tiên trong cửa sổ



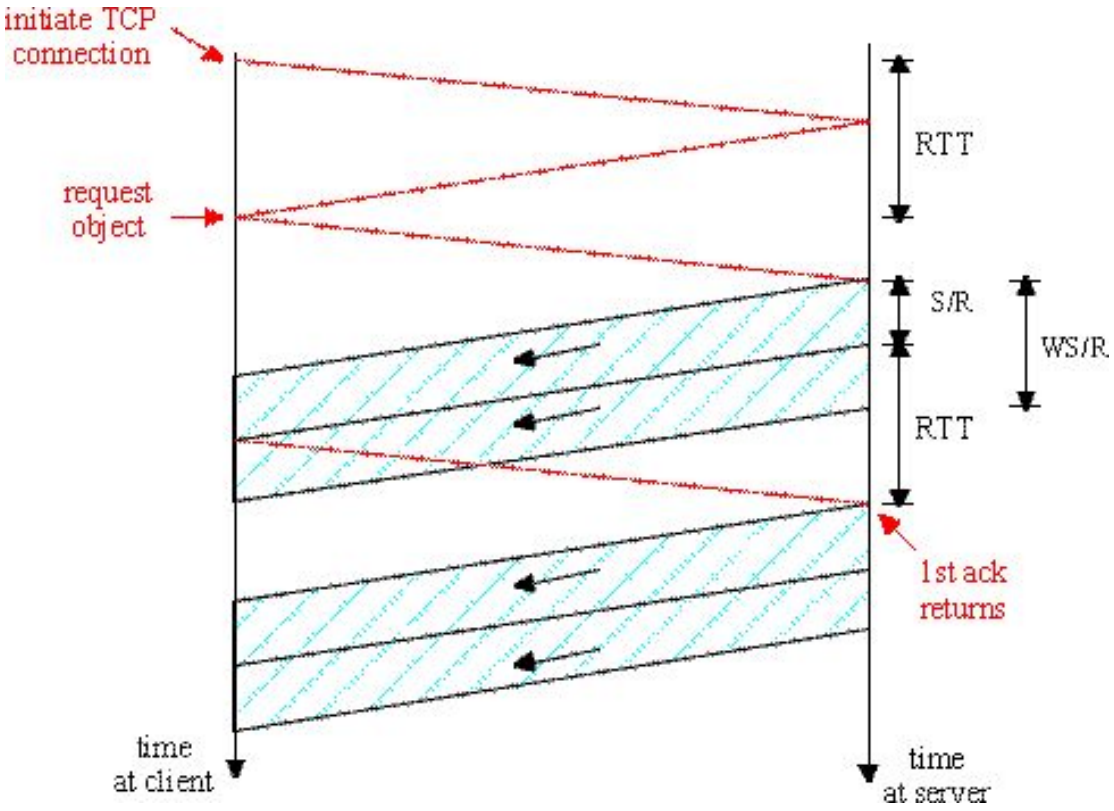
Độ trễ = 2RTT + O/R

Cửa sổ tắc nghẽn cố định

Trường hợp 2:

- WS/R < RTT + S/R: đợi ACK sau khi gửi lượng dữ liệu của cửa sổ

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



Mô hình độ trễ TCP: Slow Start (1)

Giả sử cửa sổ lớn theo slow start

Độ trễ cho một đối tượng là:

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

Trong đó, P là lượng thời gian TCP rỗi tại server

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

- Trong đó, Q là lượng thời gian server rỗi nếu đối tượng kích thước không giới hạn.
- Và K là số cửa sổ trùm qua đối tượng

Mô hình độ trễ TCP: Slow Start (2)

Các thành phần trễ:

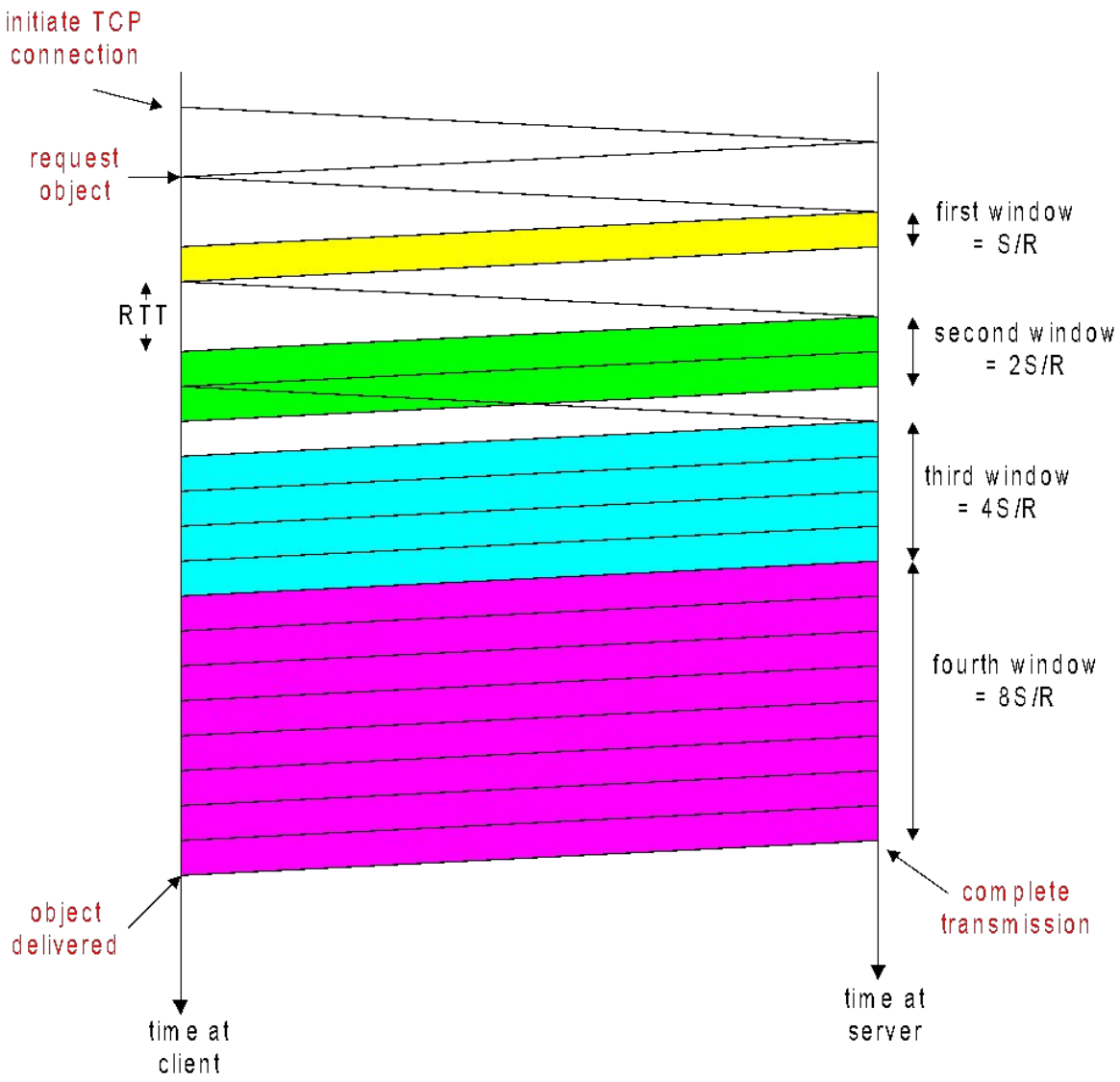
- 2 RTT để thiết lập kết nối và yêu cầu
- O/R để truyền đối tượng
- thời gian server rỗi vì slow start

Server rỗi:
P = min{K-1,Q} times

Ví dụ:

- O/S = 15 segment
- K = 4 windows
- Q = 2
- P = min{K-1,Q} = 2

Thời gian rỗi của server
P=2 times



Mô hình độ trễ TCP (3)

$$\frac{S}{R} + RTT =$$ thời gian khi server bắt đầu gửi segment tới khi server nhận ack segment

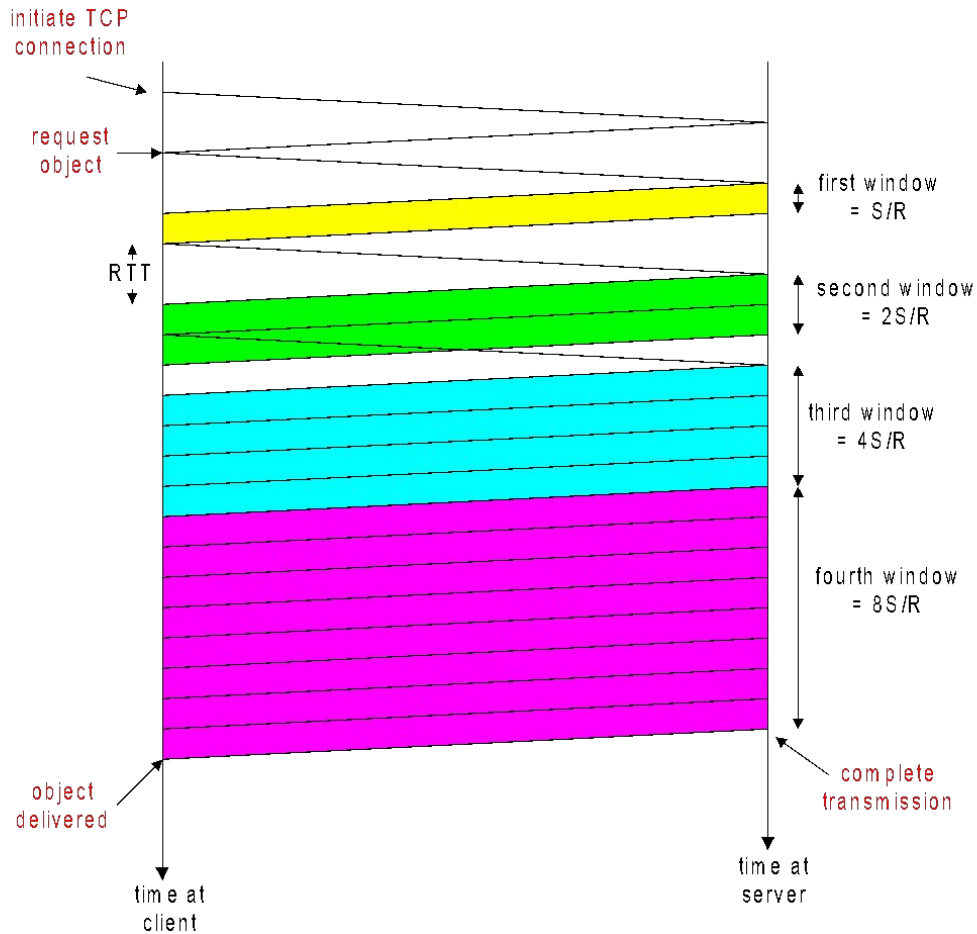
$$2^{k-1}\frac{S}{R} =$$ thời gian để truyền cửa sổ thứ k

$$\left[\frac{S}{R} + RTT - 2^{k-1}\frac{S}{R}\right]^{+} =$$ thời gian rỗi sau cửa sổ thứ k

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

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

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



Mô hình độ trễ TCP (4)

K = số cửa sổ bao đối tượng

Cách tính K?

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

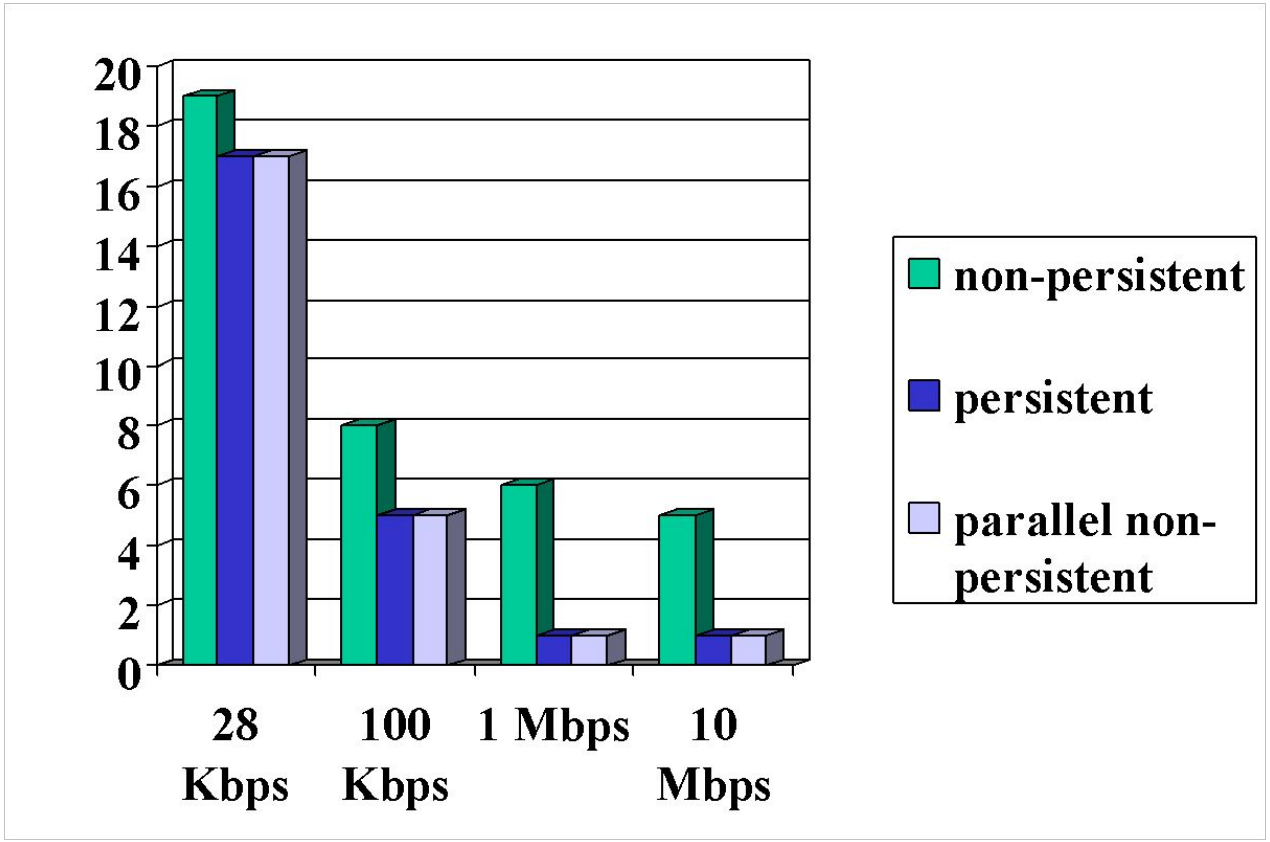
Tính Q, giá trị rỗi cho đối tượng có kích thước không giới hạn tương tự.

Mô hình hóa HTTP

- Giả sử trang Web chứa:
  - *1* trang HTML cơ sở (kích thước *O* bít)
  - *M* ảnh (mỗi ảnh kích thước *O* bít)
- Non-persistent HTTP:
  - *M+1* kết nối TCP
  - *Response time = (M+1)O/R + (M+1)2RTT + tổng thời gian rỗi*
- Persistent HTTP:
  - *2 RTT* để yêu cầu và nhận file HTML cơ sở
  - *1 RTT* để yêu cầu và nhận M ảnh
  - *Response time = (M+1)O/R + 3RTT + tổng thời gian rỗi*
- Non-persistent HTTP với X kết nối song song
  - Giả sử M/X nguyên
  - 1 kết nối TCP cho file cơ sở
  - M/X tập của các kết nối song song cho ảnh
  - *Response time = (M+1)O/R + (M/X + 1)2RTT + tổng thời gian rỗi*

Thời gian trả lời HTTP (giây)

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

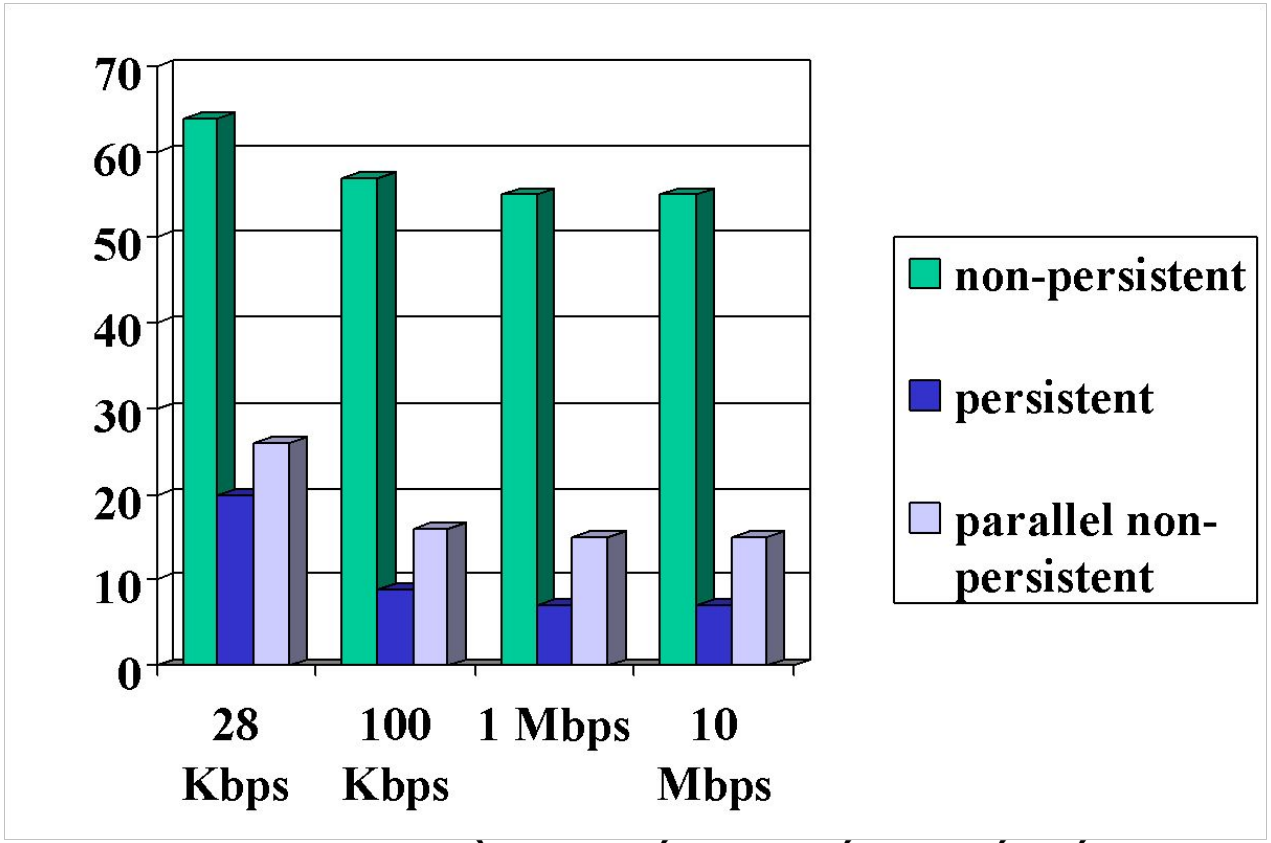


Đối với băng thông thấp, thời gian kết nối và thời gian phản hồi chi phối bởi thời gian truyền

Persistent connection chỉ mang lại cải tiến thêm qua các kết nối song song

Thời gian phản hồi (giây)

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



Đối với RTT lớn, thời gian phản hồi chi phối bởi thiết lập kết nối TCP và độ trễ slow start. Persistent connection đem lại cải tiến quan trọng: đặc biệt trong mạng delay, bandwidth cao.

Chương 4: Tổng kết

- Các nguyên tắc bên trong các dịch vụ tầng giao vận:
  - multiplexing, demultiplexing
  - Truyền dữ liệu tin cậy
  - Điều khiển luồng
  - Điều khiển tắc nghẽn
- Sự thực hiện trong Internet
  - UDP
  - TCP

Tiếp theo:

- Dừng tìm hiểu phần bên ngoài của mạng (tầng ứng dụng, tầng giao vận)
- Tìm hiểu vào trong lõi của mạng