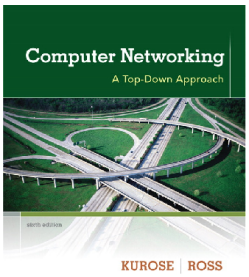
Chương 3
Tầng giao vận



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

Người dịch: Nguyễn Thanh Thủy

Tài liệu được dịch cho mục đích giảng dạy (được sự đồng ý của tác giả).

Chương 3: Tầng giao vận

Mục đích:

- Hiểu được các nguyên lý đằng sau các dịch vụ tầng giao vận:
  - Ghép kênh/phân kênh (multiplexing, demultiplexing)
  - Truyền dữ liệu tin cậy
  - Điều khiển luồng
  - Điều khiển tắc nghẽn
- Nghiên cứu về các giao thức tầng giao vận trong mạng Internet:
  - UDP: vận chuyển không kết nối
  - TCP: Vận chuyển tin cậy, hướng kết nối
  - Điều khiển tắc nghẽn trong TCP

Chương 3: Nội dung

3.1 Các dịch vụ tầng giao vận

3.2 Ghép kênh và phân kênh

3.3 Vận chuyển không kết nối: UDP

3.4 Các nguyên lý truyền dữ liệu tin cậy

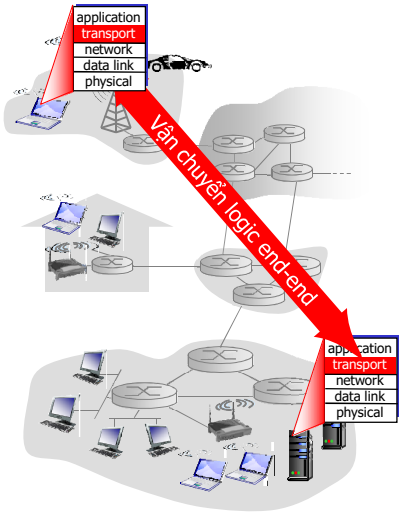
3.5 Vận chuyển hướng kết nối: TCP
- Cấu trúc đoạn dữ liệu (segment)
- Truyền dữ liệu tin cậy
- Điều khiển luồng
- Quản lý kết nối

3.6 Các nguyên lý điều khiển tắc nghẽn

3.7 Điều khiển tắc nghẽn TCP

Các dịch vụ và giao thức tầng giao vận

- Cung cấp *truyền thông logic* giữa các tiến trình ứng dụng chạy trên các host khác nhau.
- Giao thức tầng giao vận chạy trên các hệ thống đầu cuối
  - Phía gửi: cắt các thông điệp ứng dụng thành các *đoạn (segment)*, chuyển xuống tầng mạng
  - Phía nhận: Tập hợp lại các đoạn thành các thông điệp, chuyển lên tầng ứng dụng.
- Có nhiều hơn một giao thức tầng giao vận dành cho các ứng dụng
  - Internet: TCP và UDP



Tầng giao vận và tầng mạng

- *Tầng mạng:* truyền thông logic giữa các host
- *Tầng giao vận:* truyền thông logic giữa các tiến trình
  - Dựa vào và nâng cao các dịch vụ tầng mạng

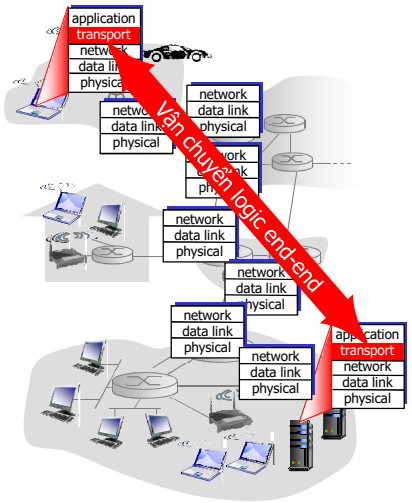
Tình huống tương tự:

12 em bé nhà Ann gửi thư đến 12 em bé nhà Bill:

- Các host = Các ngôi nhà
- Các tiến trình = các em bé
- Thông điệp ứng dụng = Nội dung bức thư (trong bì thư)
- Giao thức giao vận = Quy ước giữa các em bé nhà Ann và nhà Bill
- Giao thức tầng mạng = Dịch vụ bưu điện

Các giao thức tầng giao vận trên Internet

- Truyền tin cậy, theo thứ tự: TCP
  - Điều khiển tắc nghẽn
  - Điều khiển luồng
  - Thiết lập kết nối
- Truyền không tin cậy, không theo thứ tự: UDP
  - Mở rộng của giao thức IP
- Không có các dịch vụ:
  - Đảm bảo trễ
  - Đảm bảo băng thông



Chương 3: Nội dung

3.1 Các dịch vụ tầng giao vận

3.2 Ghép kênh và phân kênh

3.3 Vận chuyển không kết nối: UDP

3.4 Các nguyên lý truyền dữ liệu tin cậy

3.5 Vận chuyển hướng kết nối: TCP
- Cấu trúc đoạn dữ liệu (segment)
- Truyền dữ liệu tin cậy
- Điều khiển luồng
- Quản lý kết nối

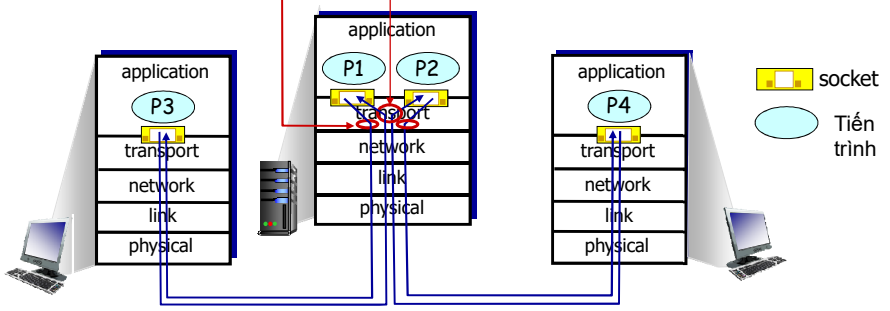
3.6 Các nguyên lý điều khiển tắc nghẽn

3.7 Điều khiển tắc nghẽn TCP

Ghép kênh/Phân kênh

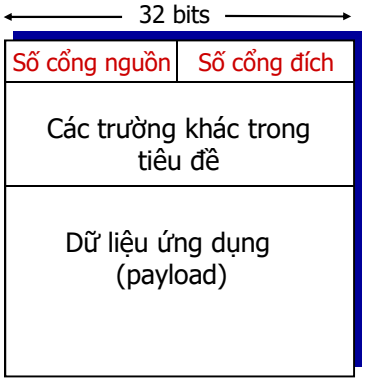
Ghép kênh tại phía gửi:
Xử lý dữ liệu từ nhiều socket, thêm phần tiêu đề tầng giao vận (sau này dùng cho việc phân kênh)

Phân kênh tại phía nhận:
Sử dụng thông tin trong phần tiêu đề để phân phối các đoạn dữ liệu (segment) đã nhận được đến đúng socket



Việc phân kênh được thực hiện như thế nào?

- Host nhận các IP datagram
  - Mỗi datagram có địa chỉ nguồn IP và địa chỉ IP đích
  - Mỗi datagram mang một đoạn dữ liệu của tầng giao vận
  - Mỗi segment có số hiệu cổng nguồn và số hiệu cổng đích
- Host sử dụng *địa chỉ IP & số hiệu cổng* để định hướng đoạn đến socket phù hợp



Định dạng TCP/UDP segment

Phân kênh hướng không kết nối

- Tạo các socket có số hiệu cổng cục bộ của host:

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

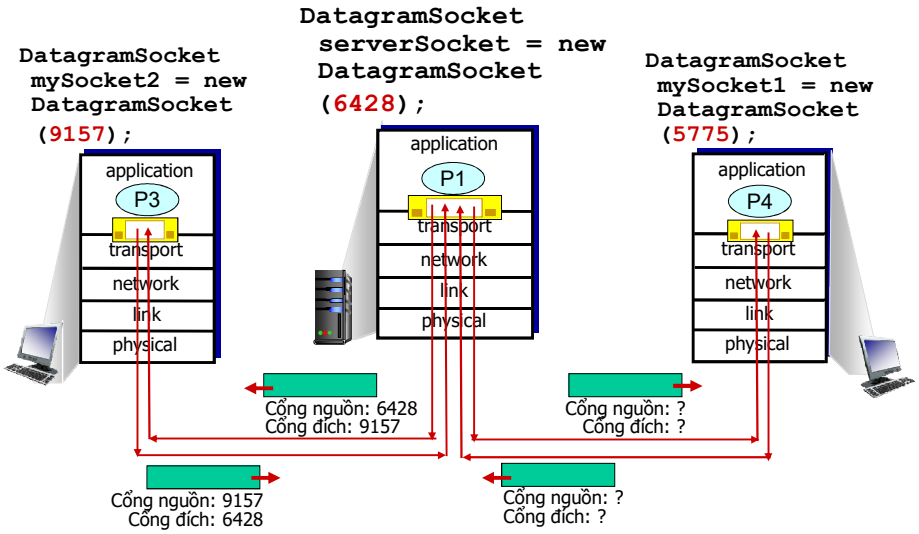
- Khi tạo datagram để gửi vào trong UDP socket, cần phải xác định:
  - Địa chỉ IP đích
  - Số hiệu cổng đích

- Khi host nhận UDP segment:
  - Kiểm tra số hiệu cổng đích trong segment
  - Định hướng UDP segment tới socket tương ứng với số hiệu cổng đó

➡

Các IP datagram với *cùng số hiệu cổng đích,* nhưng có địa chỉ IP nguồn và/hoặc các số hiệu cổng nguồn khác nhau sẽ được định hướng tới *cùng socket* tại đích

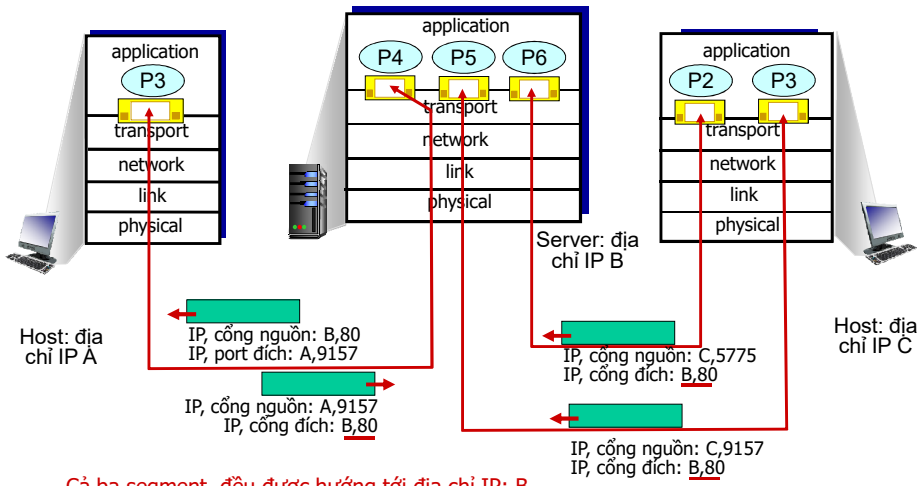
Ví dụ phân kênh hướng không kết nối



Phân kênh hướng kết nối

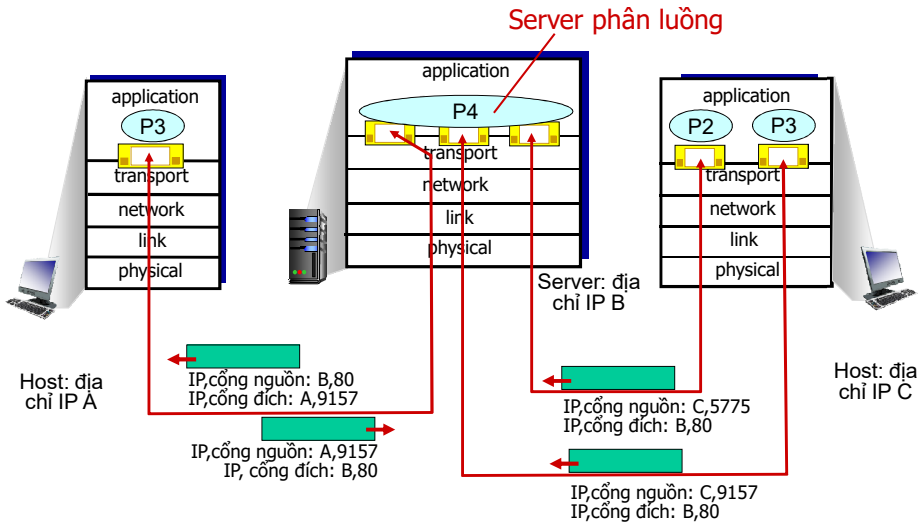
- TCP socket được xác định bởi bộ-4 giá trị:
  - Địa chỉ IP nguồn
  - Số hiệu cổng nguồn
  - Địa chỉ IP đích
  - Số hiệu cổng đích
- Phân kênh: Phía nhận sử dụng cả bốn giá trị này để định hướng segment tới socket phù hợp
- Host server có thể hỗ trợ nhiều TCP socket đồng thời:
  - Mỗi socket được xác định bởi bộ-4 giá trị của nó
- Web server có các socket khác nhau cho mỗi kết nối từ client
  - Kết nối HTTP không bền vững sẽ có các socket khác nhau cho mỗi yêu cầu.

Ví dụ phân kênh hướng kết nối



Cả ba segment, đều được hướng tới địa chỉ IP: B, cổng đích: 80, nhưng được phân kênh tới các socket *khác nhau*

Ví dụ phân kênh hướng kết nối



Chương 3: Nội dung

3.1 Các dịch vụ tầng giao vận

3.2 Ghép kênh và phân kênh

3.3 Vận chuyển không kết nối: UDP

3.4 Các nguyên lý truyền dữ liệu tin cậy

3.5 Vận chuyển hướng kết nối: TCP
- Cấu trúc đoạn dữ liệu (segment)
- Truyền dữ liệu tin cậy
- Điều khiển luồng
- Quản lý kết nối

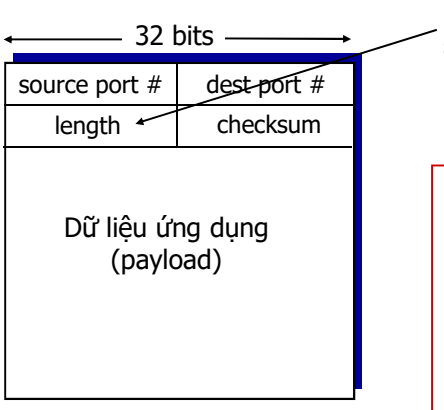
3.6 Các nguyên lý điều khiển tắc nghẽn

3.7 Điều khiển tắc nghẽn TCP

UDP: User Datagram Protocol [RFC 768]

- Là giao thức tầng giao vận của mạng Internet
- Dịch vụ "best effort", các UDP segment có thể:
  - Bị mất
  - Được vận chuyển không đúng thứ tự tới ứng dụng
- *Hướng không kết nối:*
  - Không có giai đoạn bắt tay giữa bên gửi và bên nhận của UDP
  - Mỗi UDP segment được xử lý độc lập với các segment khác
- UDP được dùng trong:
  - Các ứng dụng streaming multimedia (chịu mất mát dữ liệu, bị ảnh hưởng bởi tốc độ)
  - DNS
  - SNMP
- Truyền tin cậy trên UDP:
  - Bổ sung đặc tính tin cậy vào tầng ứng dụng
  - Khôi phục lỗi cụ thể của ứng dụng

UDP: Tiêu đề segment



Định dạng UDP segment

Chiều dài, được tính theo số byte của UDP segment, bao gồm cả phần tiêu đề

Tại sao lại dùng UDP?

- Không cần thiết lập kết nối (vì việc này có thể làm tăng độ trễ)
- Đơn giản: không lưu trạng thái kết nối tại bên gửi, bên nhận
- Kích thước tiêu đề nhỏ
- Không điều khiển tắc nghẽn: UDP có thể gửi nhanh theo mong muốn

UDP checksum

Mục tiêu: Phát hiện các "lỗi" (ví dụ: các bit bị bật lên) trong các segment được truyền đến

Bên gửi:

- Xử lý nội dung các đoạn, bao gồm cả các trường trong tiêu đề, như là chuỗi các số nguyên 16-bit
- checksum: bổ sung thêm (tổng bù của 1) vào nội dung segment
- Bên gửi đặt giá trị checksum vào trong trường checksum của UDP

Bên nhận:

- Tính toán checksum của segment đã nhận được
- Kiểm tra xem checksum đã tính có bằng giá trị của trường checksum hay không:
  - KHÔNG – phát hiện có lỗi
  - CÓ – không phát hiện lỗi. *Nhưng vẫn có thể có lỗi mà chưa được phát hiện?* Xem thêm phần sau ….

Ví dụ: checksum trên Internet

Ví dụ: Cộng hai số nguyên 16-bit

		1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
		1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
Bit dư	(1)	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
Tổng		1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum		0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

Chú ý: Khi cộng các số nguyên, một bit nhớ ở phía cao nhất cần phải được thêm vào kết quả

Chương 3: Nội dung

3.1 Các dịch vụ tầng giao vận

3.2 Ghép kênh và phân kênh

3.3 Vận chuyển không kết nối: UDP

3.4 Các nguyên lý truyền dữ liệu tin cậy

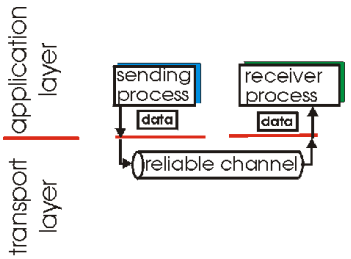
3.5 Vận chuyển hướng kết nối: TCP
- Cấu trúc đoạn dữ liệu (segment)
- Truyền dữ liệu tin cậy
- Điều khiển luồng
- Quản lý kết nối

3.6 Các nguyên lý điều khiển tắc nghẽn

3.7 Điều khiển tắc nghẽn TCP

Các nguyên lý của truyền dữ liệu tin cậy

- Quan trọng trong các tầng ứng dụng, giao vận và liên kết
  - Thuộc danh sách 10 vấn đề quan trọng nhất của mạng!

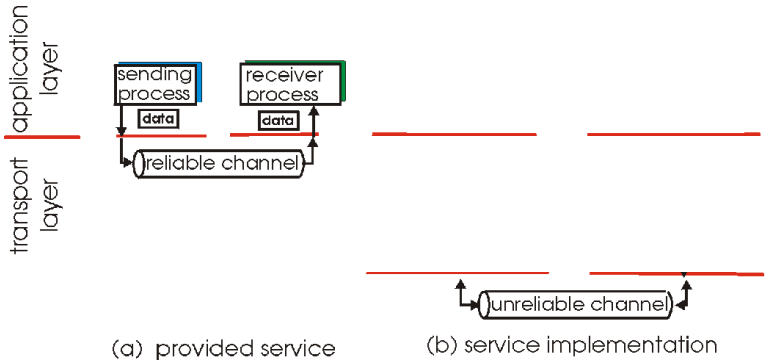


(a) provided service

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

Các nguyên lý của truyền dữ liệu tin cậy

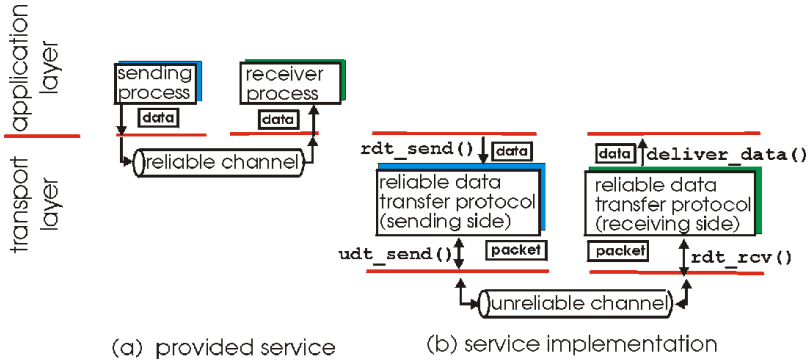
- Quan trọng trong các tầng ứng dụng, giao vận và liên kết
  - Thuộc danh sách 10 vấn đề quan trọng nhất của mạng!



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

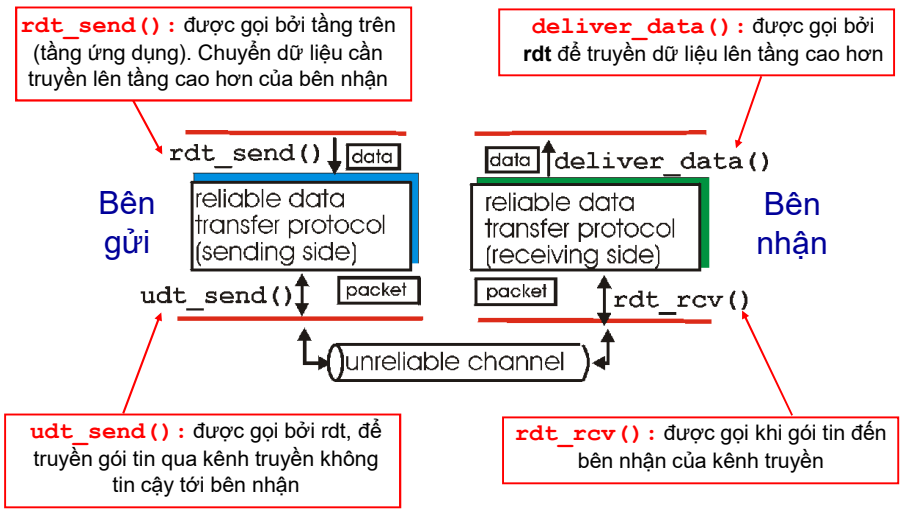
Các nguyên lý của truyền dữ liệu tin cậy

- Quan trọng trong các tầng ứng dụng, giao vận và liên kết
  - Thuộc danh sách 10 vấn đề quan trọng nhất của mạng!



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

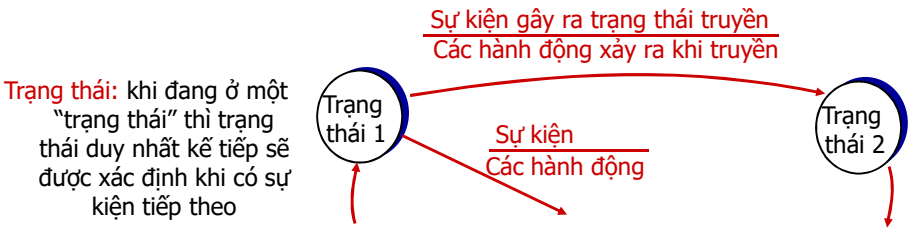
Truyền dữ liệu tin cậy



Truyền dữ liệu tin cậy

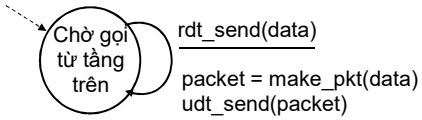
Việc cần làm:

- Phát triển dần giao thức truyền dữ liệu tin cậy (reliable data transfer protocol - rdt) cho cả bên gửi và bên nhận
- Chỉ xem xét truyền dữ liệu theo một hướng
  - Nhưng thông tin điều khiển vẫn được truyền theo cả hai hướng
- Dùng máy trạng thái hữu hạn (finite state machines - FSM) để xác định bên gửi, bên nhận

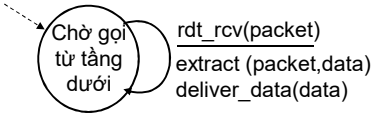


rdt1.0: truyền dữ liệu tin cậy qua một kênh truyền tin cậy

- Kênh truyền cơ bản hoàn toàn tin cậy
  - Không có lỗi bit
  - Không có mất mát gói tin
- Phân biệt các FSM cho bên gửi, bên nhận:
  - Bên gửi gửi dữ liệu vào kênh truyền cơ bản
  - Bên nhận đọc dữ liệu từ kênh truyền cơ bản



Bên gửi



Bên nhận

rdt2.0: Kênh truyền có lỗi bit

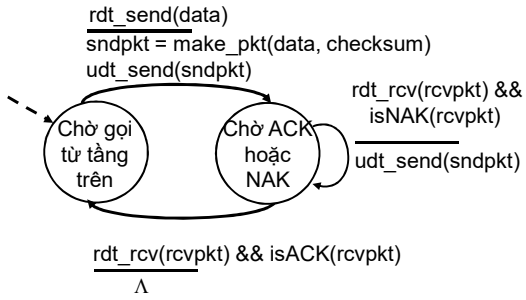
- Kênh cơ bản có thể bật một vài bit trong gói tin
  - Kiểm tra (checksum) để phát hiện các lỗi bit
- Câu hỏi: Làm thế nào để khôi phục lại các lỗi?

Làm thế nào con người khôi phục được "lỗi" trong suốt quá trình thực hiện cuộc hội thoại?

rdt2.0: Kênh truyền có lỗi bit

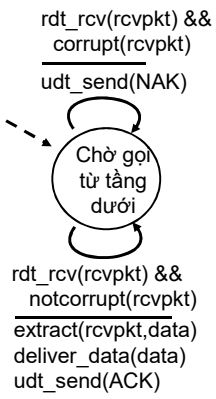
- Kênh truyền cơ bản có thể bật một vài bit trong gói tin
  - Kiểm tra (checksum) để phát hiện các lỗi bit
- Câu hỏi: Làm thế nào để khôi phục lại các lỗi?
  - *Báo nhận ACK (acknowledgement):* bên nhận thông báo rõ cho cho bên gửi là gói tin nhận được tốt
  - *Báo nhận NAK (negative acknowledgement):* bên nhận thông báo rõ cho bên gửi là gói tin nhận được có lỗi
  - Bên gửi truyền lại gói tin có báo nhận là NAK
- Các cơ chế mới trong rdt2.0 (ngoài rdt1.0):
  - Phát hiện lỗi
  - Phản hồi: các thông điệp điều khiển (ACK,NAK) từ bên nhận gửi về bên gửi

rdt2.0: Đặc tả FSM

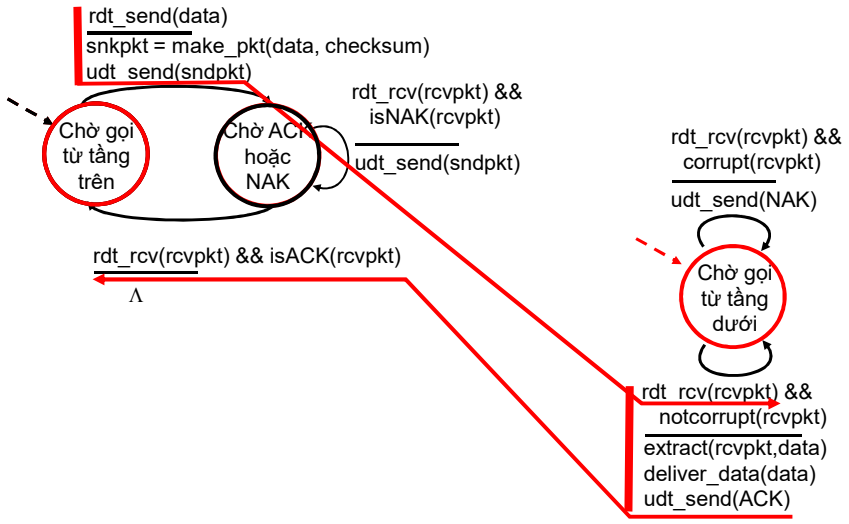


Bên gửi

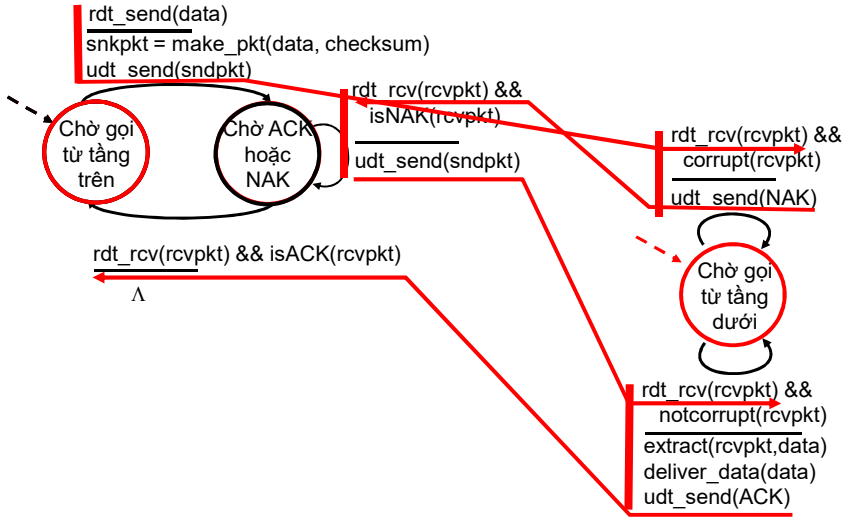
Bên nhận



rdt2.0: Hoạt động khi không có lỗi



rdt2.0: Kịch bản khi có lỗi



rdt2.0 có lỗ hổng nghiêm trọng!

Điều gì xảy ra khi ACK/NAK bị hỏng?

- Bên gửi không biết được điều gì đã xảy ra tại bên nhận!
- Không thể đơn phương truyền lại: có thể bị trùng lặp

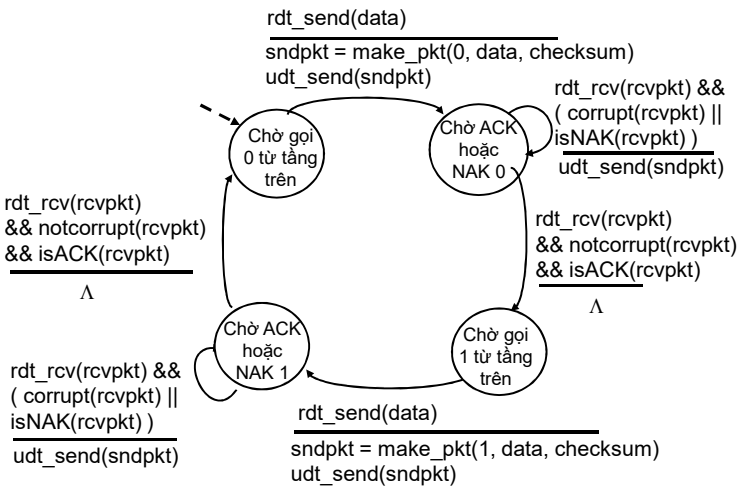
Xử lý trùng lặp:

- Bên gửi truyền lại gói tin hiện tại nếu ACK/NAK bị hỏng
- Bên gửi thêm *số thứ tự* vào trong mỗi gói tin
- Bên nhận bỏ qua (không nhận) gói bị trùng lặp

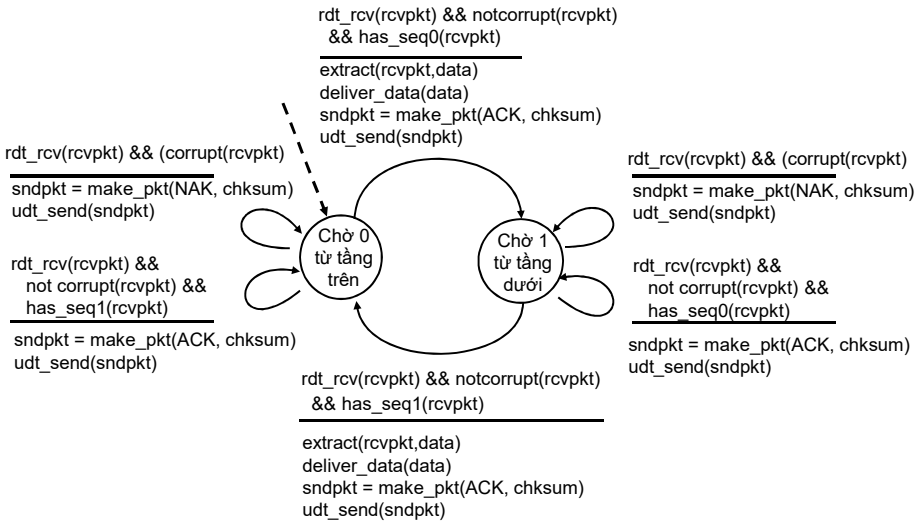
Dừng và chờ

Bên gửi gửi một gói tin, sau đó dừng lại chờ bên nhận phản hồi

rdt2.1: Bên gửi xử lý các ACK/NAK bị hỏng



rdt2.1: Bên nhận xử lý các ACK/NAK bị hỏng



rdt2.1: Thảo luận

Bên gửi:

- Số thứ tự được bổ sung vào gói tin
- Chỉ cần hai số thứ tự (0,1) là đủ. Vì sao?
- Phải kiểm tra lại nếu việc nhận ACK/NAK bị hỏng
- Số trạng thái tăng lên 2 lần
  - Trạng thái phải "nhớ" xem gói tin đang "dự kiến" đến sẽ có số thứ tự là 0 hay 1

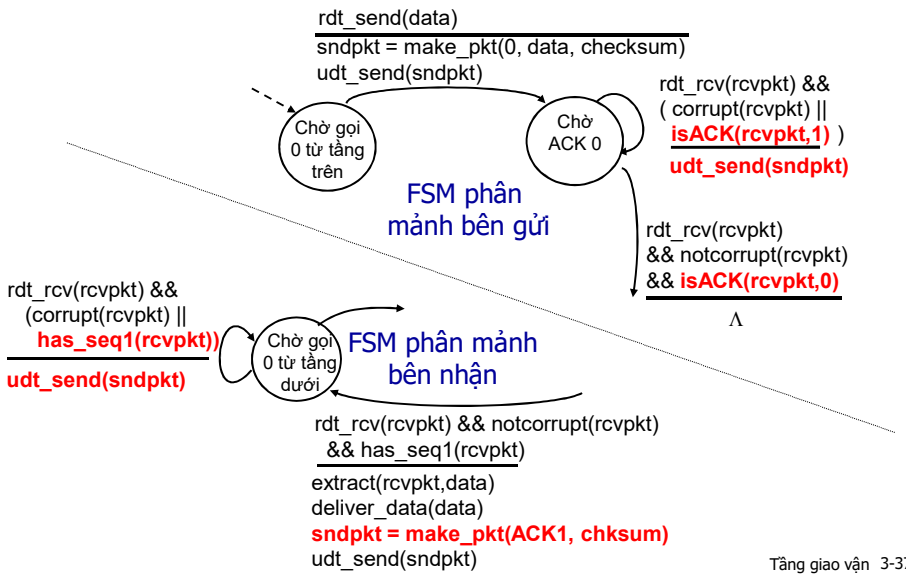
Bên nhận:

- Phải kiểm tra xem gói tin nhận được có bị trùng lặp hay không
  - Trạng thái chỉ rõ gói tin đang chờ đến có số thứ tự là 0 hay 1
- Chú ý: bên nhận *không thể* biết được ACK/NAK cuối cùng gửi đi có được nhận tốt hay không tại bên gửi

rdt2.2: Một giao thức không cần NAK

- Chức năng giống như trong rdt2.1, nhưng chỉ dùng báo nhận ACK
- Thay vì sử dụng NAK, bên nhận sẽ gửi ACK cho gói tin cuối cùng nhận tốt
  - Bên nhận phải thêm số thứ tự của gói tin đang được báo nhận
- ACK bị trùng lặp tại bên gửi sẽ dẫn đến cùng hành động như NAK: *truyền lại gói tin hiện tại*

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



rdt3.0: Kênh truyền có lỗi và mất mát

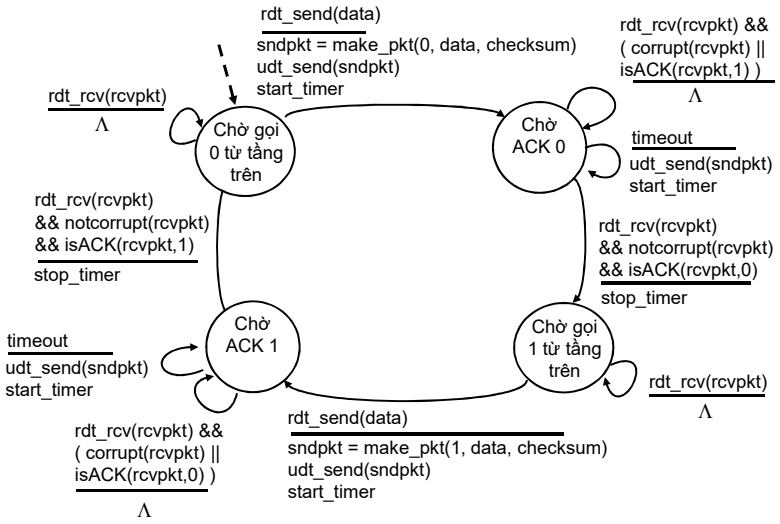
Giả thiết mới: Kênh cơ bản cũng có thể làm mất các gói tin (dữ liệu, ACK)

- checksum, số thứ tự, báo nhận ACK, truyền lại sẽ hỗ trợ… nhưng chưa đủ

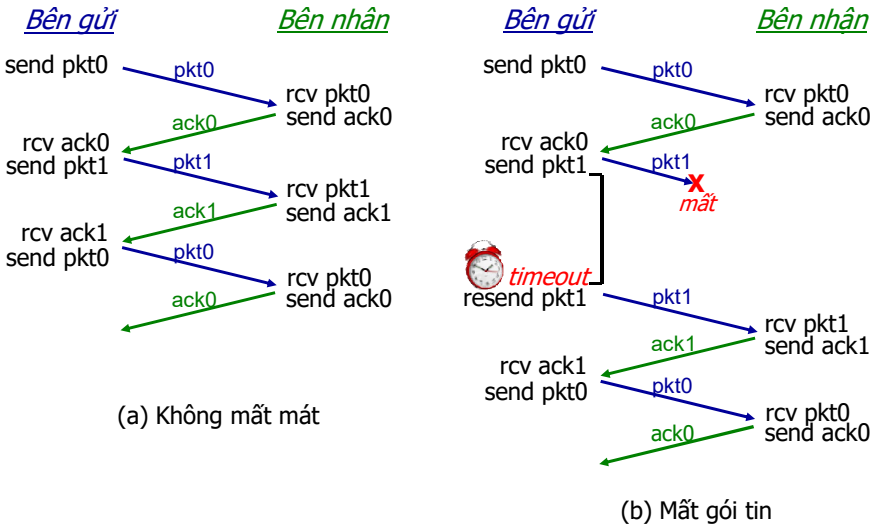
Tiếp cận: Bên gửi chờ ACK trong khoảng thời gian "chấp nhận được"

- Truyền lại nếu không nhận được ACK trong khoảng thời gian này
- Nếu gói tin (hoặc ACK) chỉ đến trễ (chứ không bị mất):
  - Việc truyền lại sẽ gây trùng lặp, nhưng số thứ tự sẽ xử lý việc này
  - Bên nhận phải chỉ rõ số thứ tự của gói tin đang được báo nhận
- Cần bộ định thời đếm ngược

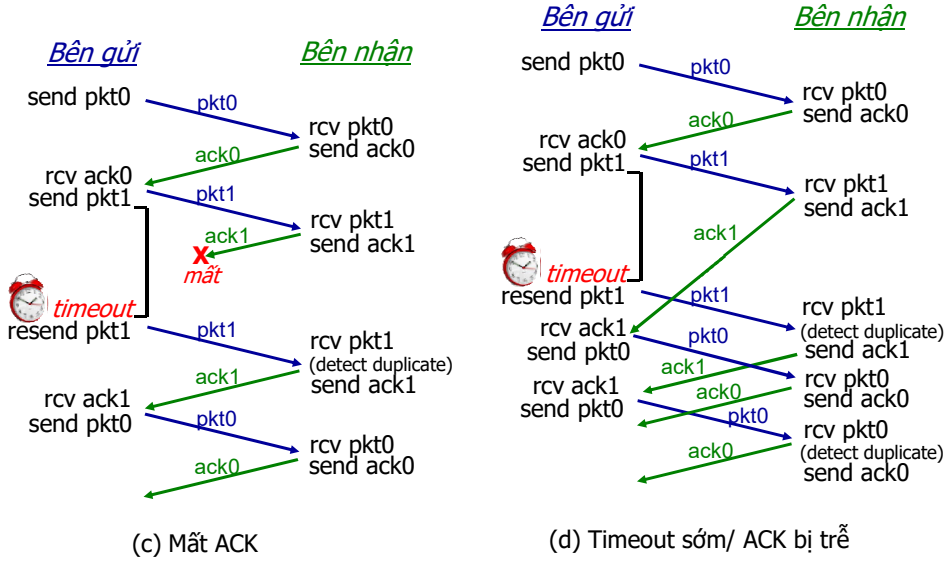
rdt3.0 bên gửi



Hoạt động của rdt3.0



Hoạt động của rdt3.0



Hiệu suất của rdt3.0

- rdt3.0 hoạt động tốt, nhưng không hiệu quả
- Ví dụ: Liên kết 1 Gbps, trễ lan truyền 15 ms, gói tin 8000 bit:

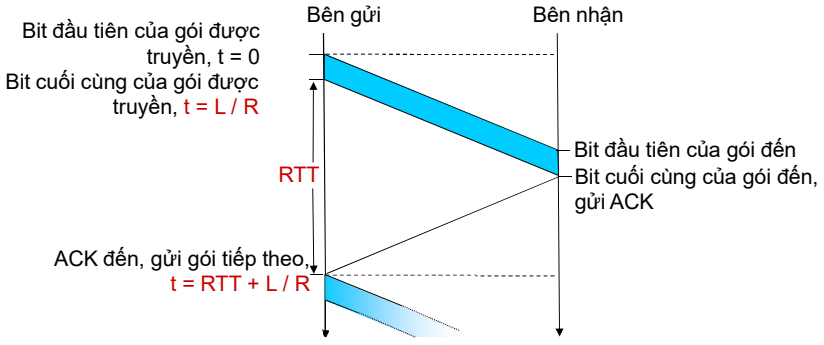
$$D_{trans} = \frac{L}{R} = \frac{8000\ bit}{10^9\ bit/sec} = 8\ microsecs$$

  - U sender: độ khả dụng – tỷ lệ về mặt thời gian bên gửi liên tục phải gửi

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

    - Nếu RTT=30 msec, gói tin 1KB được truyền sau mỗi 30 msec: thông lượng trên liên kết 1 Gbps là 33kB/sec
- Giao thức mạng giới hạn việc sử dụng các tài nguyên vật lý!

rdt3.0: Hoạt động dừng-và-chờ

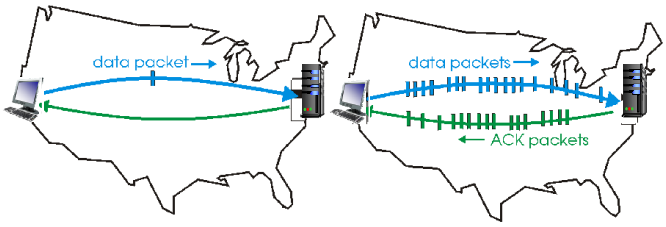


$$U_{sender} = \frac{L/R}{RTT + L/R} = \frac{.008}{30.008} = 0.00027$$

Các giao thức Pipeline

Pipelining: bên gửi cho phép gửi nhiều gói "đồng thời", mà không cần chờ gói báo nhận

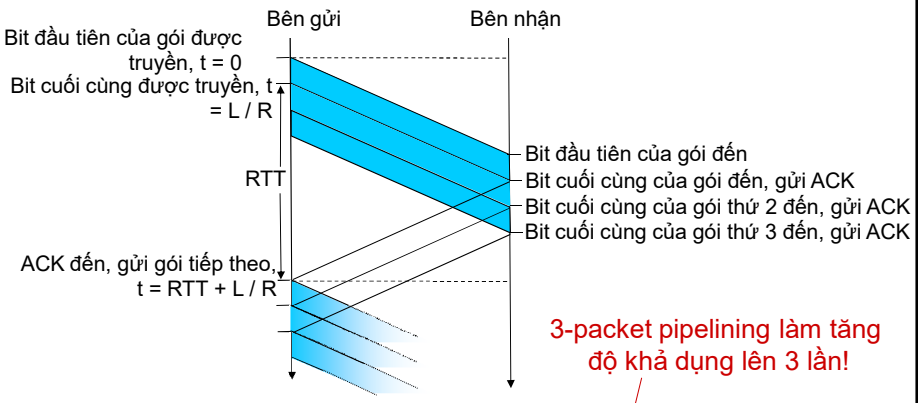
- Dãy các số thứ tự sẽ được tăng dần
- Cần có bộ đệm tại bên gửi và/hoặc bên nhận



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

- Hai dạng thức chung của các giao thức pipeline : *go-Back-N, lặp có lựa chọn (selective repeat)*

Pipelining: tăng độ khả dụng



3-packet pipelining làm tăng độ khả dụng lên 3 lần!

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

Các giao thức pipeline

Go-back-N:

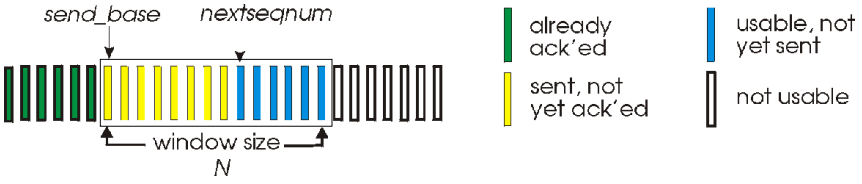
- Bên gửi có thể có đến N gói chưa được báo nhận trong pipeline
- Bên nhận chỉ gửi *ack tích lũy*
  - Không báo nhận cho gói tin cho đến khi có một khoảng trống
- Bên gửi có bộ định thời cho các gói tin gửi đi mà chưa được báo nhận
  - Khi bộ định thời hết hạn, truyền lại tất cả các gói tin chưa được báo nhận

Lặp có lựa chon:

- Bên gửi có thể có đến N gói chưa được báo nhận trong pipeline
- Bên nhận gửi *ack riêng* cho mỗi gói tin
- Bên gửi duy trì bộ định thời cho mỗi gói tin chưa được báo nhận
  - Khi bộ định thời hết hạn, chỉ truyền lại gói tin chưa được báo nhận

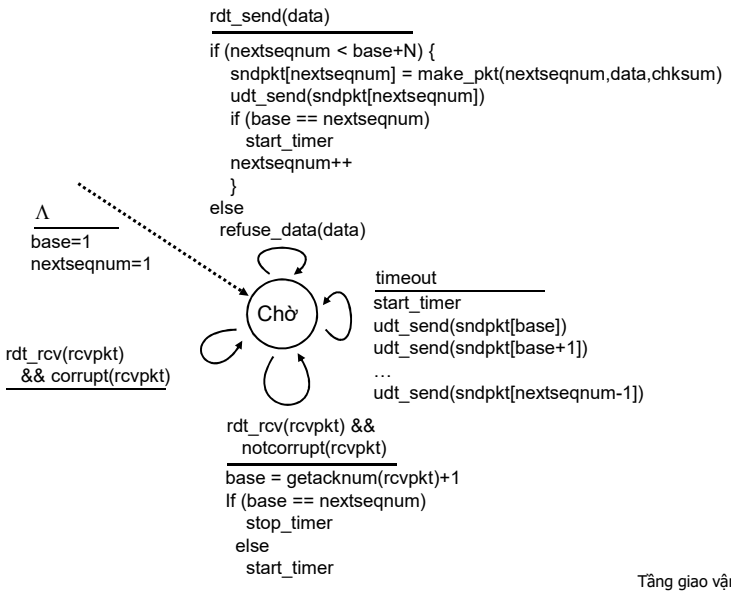
Go-Back-N: bên gửi

- k-bit số thứ tự trong phần tiêu đề của gói tin
- "Cửa sổ" tăng lên đến N, cho phép gửi gói liên tục không cần báo nhận

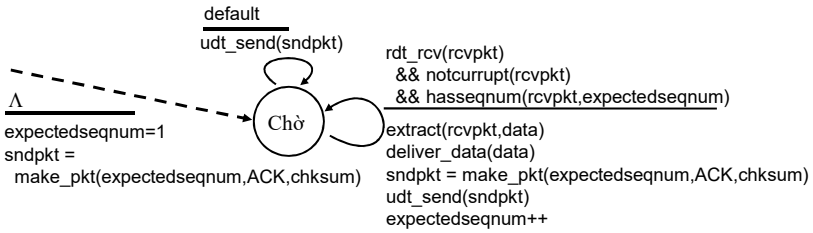


- ACK(n): báo nhận ACK cho tất cả các gói đến, chứa số thứ tự n- *"ACK tích lũy"*
  - Có thể nhận được ACK trùng lặp (xem bên nhận)
- Đặt bộ định thời cho các gói tin truyền đi
- *timeout(n):* truyền lại gói n và tất cả các gói có số thứ tự lớn hơn trong cửa sổ

GBN: FSM mở rộng tại bên gửi



GBN: FSM mở rộng tại bên nhận



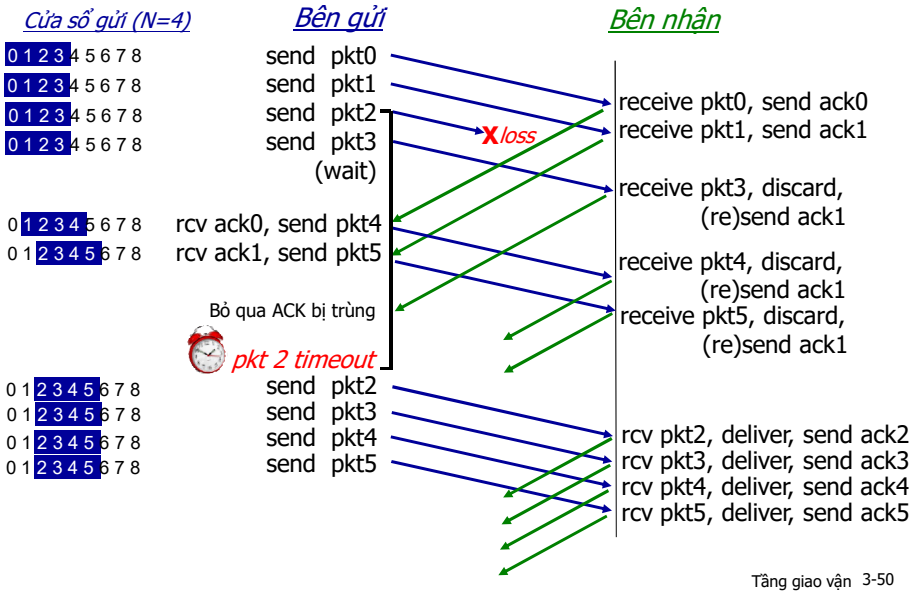
ACK-duy nhất: luôn gửi ACK cho gói đã nhận đúng với số thứ tự xếp hạng cao nhất

- Có thể sinh ra ACK trùng nhau
- Chỉ cần nhớ số thứ tự của gói dự kiến đến (**expectedseqnum**)

❖ Gói không theo đúng thứ tự:

- Hủy: *không nhận vào vùng đệm!*
- Gửi lại ACK với số thứ tự (xếp hạng) cao nhất

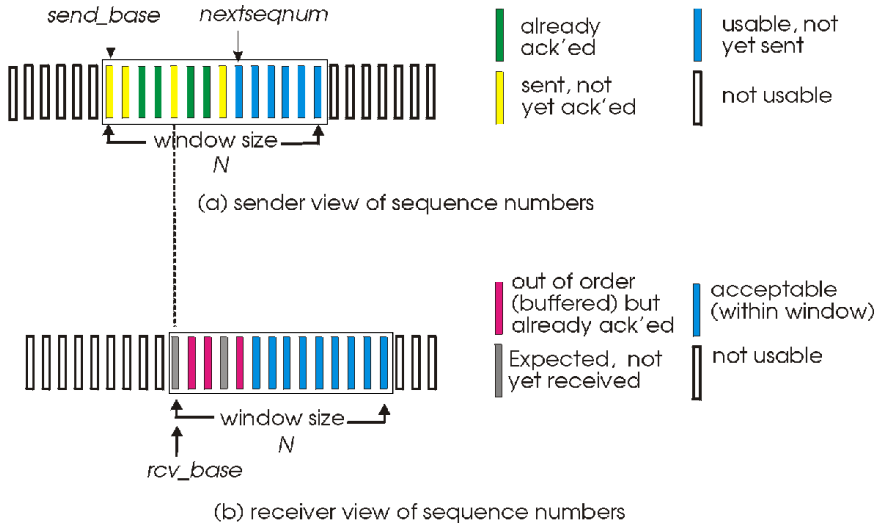
Hoạt động của GBN



Lặp có lựa chọn

- Bên nhận báo nhận *riêng* cho tất cả các gói tin đã nhận đúng.
  - Đặt các gói vào bộ đệm (nếu cần), cho đúng thứ tự để chuyển lên tầng cao hơn
- Bên gửi chỉ gửi lại các gói tin nào mà không nhận được ACK
  - Có bộ định thời bên gửi cho mỗi gói tin 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: cửa sổ bên gửi, bên nhận



Lặp có lựa chọn

Bên gửi

Dữ liệu từ tầng trên:

- Nếu số thứ tự kế tiếp sẵn sàng trong cửa sổ, thì gửi gói tin

timeout(n):

- Gửi lại gói n, khởi tạo lại bộ định thời

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

- Đánh dấu gói n là đã nhận
- Nếu gói có số thứ tự n thấp nhất mà chưa được ACK, thì dịch chuyển cửa sổ cơ sở đến số thứ tự kế tiếp chưa được ACK.

Bên nhận

Gói n trong [rcvbase, rcvbase+N-1]

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

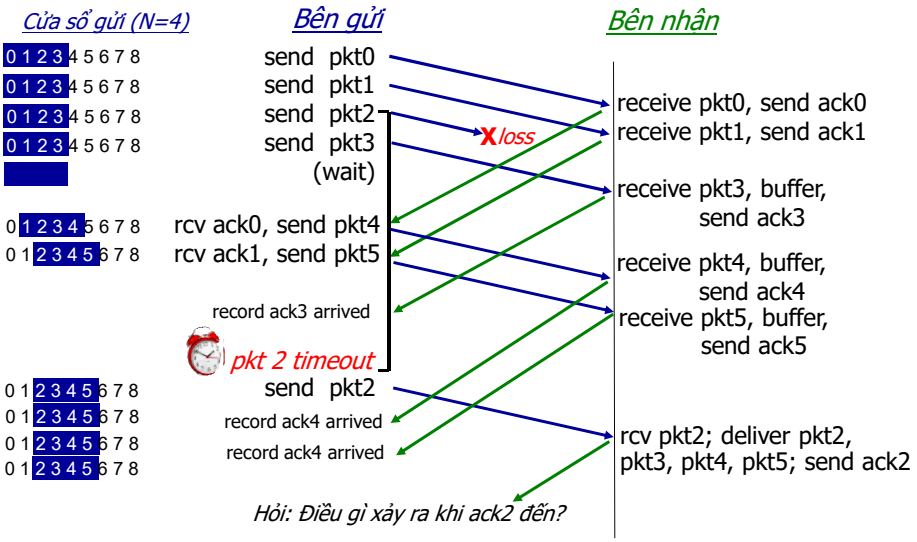
Gói n trong [rcvbase-N,rcvbase-1]

- ACK(n)

Ngược lại:

- Bỏ qua

Hoạt động trong lặp có lựa chọn

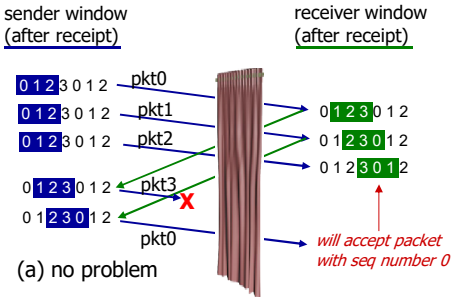


Lặp có lựa chọn: tình trạng khó giải quyết

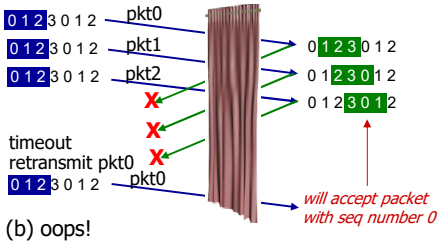
Ví dụ:

- Các số thứ tự: 0, 1, 2, 3
- Kích thước cửa sổ =3
- Bên nhận không nhận ra sự khác biệt giữa 2 kịch bản!
- Chấp nhận dữ liệu bị trùng lặp như là dữ liệu mới trong (b)

Hỏi: Quan hệ giữa kích thước số thứ tự và kích thước cửa sổ như thế nào để tránh vấn đề như trong (b)?



receiver can't see sender side.
receiver behavior identical in both cases!
something's (very) wrong!



Chương 3: Nội dung

3.1 Các dịch vụ tầng giao vận

3.2 Ghép kênh và phân kênh

3.3 Vận chuyển không kết nối: UDP

3.4 Các nguyên lý truyền dữ liệu tin cậy

3.5 Vận chuyển hướng kết nối: TCP
- Cấu trúc đoạn dữ liệu (segment)
- Truyền dữ liệu tin cậy
- Điều khiển luồng
- Quản lý kết nối

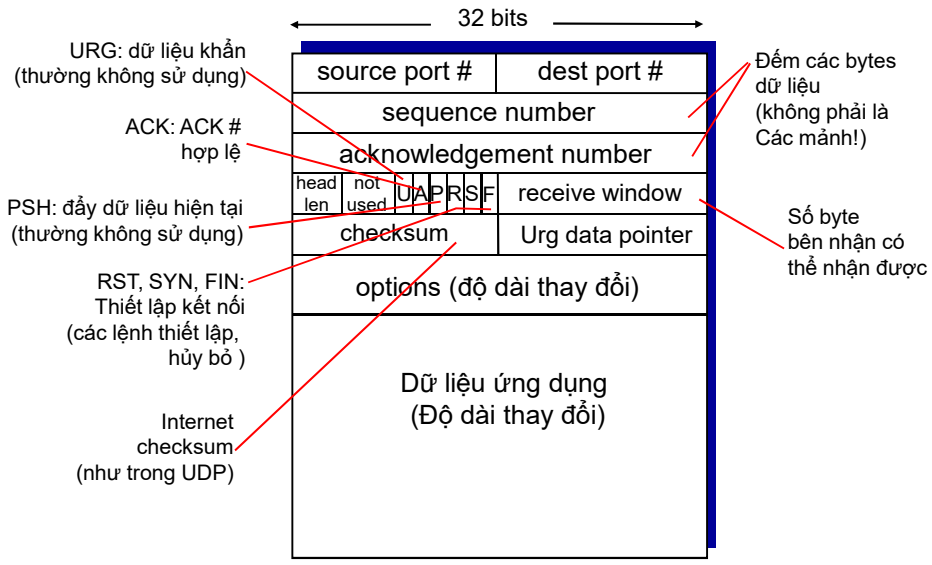
3.6 Các nguyên lý điều khiển tắc nghẽn

3.7 Điều khiển tắc nghẽn TCP

Khái quát TCP RFCs: 793,1122,1323, 2018, 2581

- Điểm-tới-điểm:
  - Một bên gửi, một bên nhận
- Truyền *dòng byte* theo đúng thứ tự và truyền tin cậy:
  - Không có "ranh giới thông điệp"
- pipeline:
  - Điều khiển tắc nghẽn và điều khiển luồng TCP thiết lập kích thước cửa sổ
- Truyền dữ liệu song công (full duplex):
  - Luồng dữ liệu đi theo 2 hướng trên cùng một kết nối
  - MSS: maximum segment size (kích thước đoạn lớn nhất)
- Hướng kết nối:
  - Bắt tay (trao đổi các thông điệp điều khiển) khởi tạo trạng thái cho bên gửi và bên nhận trước khi trao đổi dữ liệu
- Điều khiển luồng:
  - Bên gửi không lấn át bên nhận

Cấu trúc TCP segment



Số thứ tự và báo nhận ACK trong TCP

Số thứ tự:

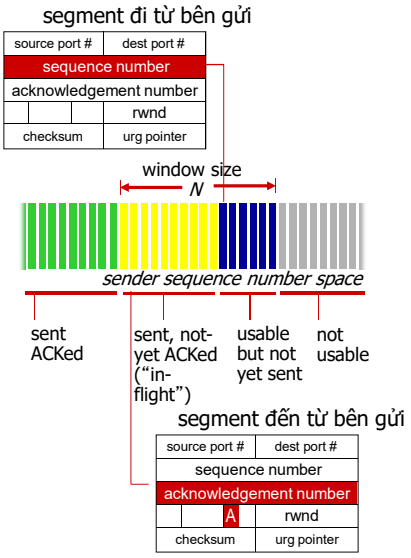
- "Số" dòng byte của byte đầu tiên trong đoạn (segment) dữ liệu

Báo nhận:

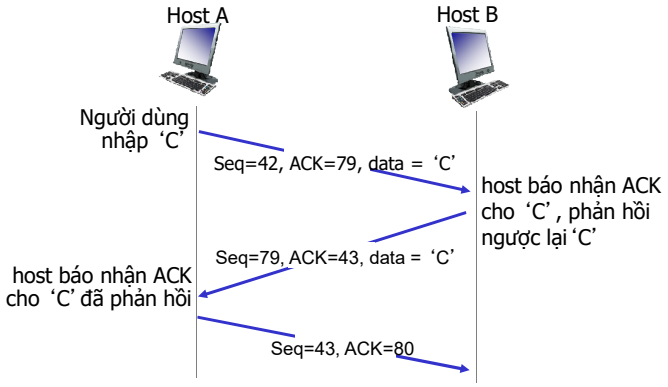
- Số thứ tự của byte tiếp theo được mong đợi từ phía bên kia
- ACK tích lũy

Hỏi: Làm thế nào bên nhận xử lý được các segment không đúng thứ tự?

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



Số thứ tự và báo nhận ACK trong TCP



Kịch bản telnet đơn giản

TCP round trip time và timeout

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

- Dài hơn RTT
  - nhưng RTT thay đổi
- *Quá ngắn:* timeout sớm, không cần truyền lại
- *Quá dài:* phản ứng chậm với các segment bị mất

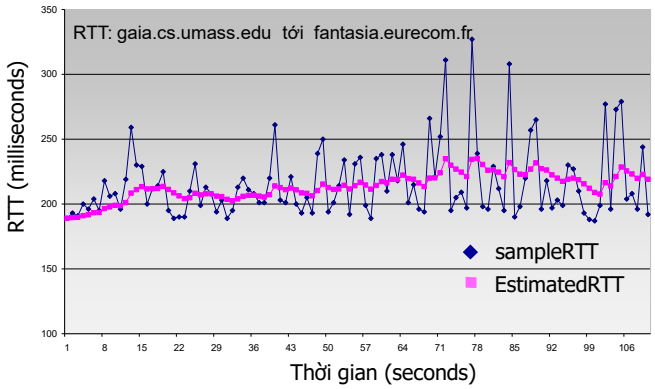
Hỏi: Ước lượng RTT như nào?

- **SampleRTT**: thời gian đo được từ khi truyền segment đến khi nhận được ACK
  - Bỏ qua việc truyền lại
- **SampleRTT** có thể thay đổi, cần giá trị RTT ước lượng "mượt hơn"
  - Tính trung bình một vài độ đo gần đây, không chỉ **SampleRTT** hiện tại

TCP round trip time và timeout

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

- Giá trị đặc trưng: α = 0.125



TCP round trip time và timeout

- Khoảng thời gian timeout: **EstimatedRTT** cộng với "hệ số dự trữ an toàn"
  - Nếu có biến thiên lớn trong **EstimatedRTT**, thì hệ số dự trữ an toàn phải lớn hơn
- Ước lượng sự biến thiên của SampleRTT từ EstimatedRTT:

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

(Giá trị đặc trưng: β = 0.25)

TimeoutInterval = EstimatedRTT + 4*DevRTT

RTT ước lượng

"hệ số dự trữ an toàn"

Chương 3: Nội dung

3.1 Các dịch vụ tầng giao vận

3.2 Ghép kênh và phân kênh

3.3 Vận chuyển không kết nối: UDP

3.4 Các nguyên lý truyền dữ liệu tin cậy

3.5 Vận chuyển hướng kết nối: TCP
- Cấu trúc đoạn dữ liệu (segment)
- Truyền dữ liệu tin cậy
- Điều khiển luồng
- Quản lý kết nối

3.6 Các nguyên lý điều khiển tắc nghẽn

3.7 Điều khiển tắc nghẽn TCP

Truyền dữ liệu tin cậy trong TCP

- TCP tạo dịch vụ rdt trên dịch vụ không tin cậy của IP
  - Truyền segment theo kiểu pipelining
  - ACK tích lũy
  - Dùng bộ định thời cho việc truyền lại
- Việc truyền lại được kích hoạt bởi:
  - Các sự kiện timeout
  - ACK bị trùng lặp

Hãy bắt đầu xem xét bên gửi TCP theo cách đơn giản:
- Bỏ qua trùng lặp ACK
- Bỏ qua điều khiển luồng, điều khiển tắc nghẽn

Các sự kiện của TCP bên gửi:

Dữ liệu nhận từ ứng dụng:

- Tạo segment với số thứ tự
- Số thứ tự là số dòng byte của byte dữ liệu đầu tiên trong segment
- Khởi tạo bộ định thời nếu chưa chạy:
  - Chú ý bộ định thời của segment chưa được báo nhận muộn nhất
  - Hết thời hạn: **TimeOutInterval**

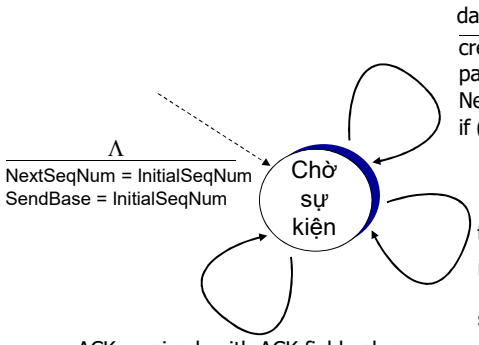
Timeout:

- Truyền lại segment bị timeout
- Khởi tạo lại bộ định thời

ACK đã nhận:

- Nếu ACK báo nhận cho các segment chưa được báo nhận trước đó, thì:
  - Cập nhật lại các segment đã được báo nhận
  - Khởi tạo bộ định thời nếu vẫn còn các segment chưa được báo nhận

TCP bên gửi (đơn giản hóa)



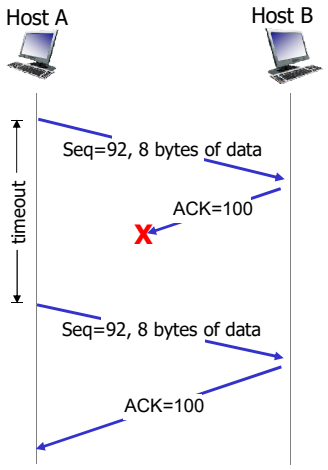
data received from application above
create segment, seq. #: NextSeqNum
pass segment to IP (i.e., "send")
NextSeqNum = NextSeqNum + length(data)
if (timer currently not running)
start timer

timeout
retransmit not-yet-acked segment with smallest seq. #
start timer

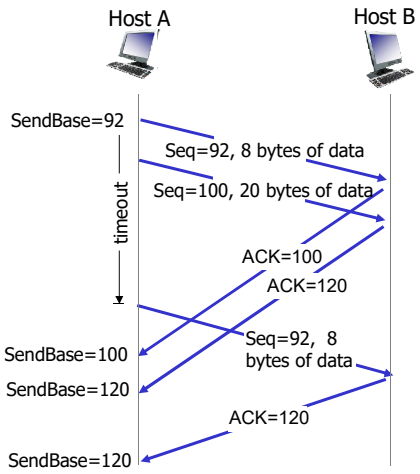
ACK received, with ACK field value y

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

TCP: Các tình huống phải truyền lại

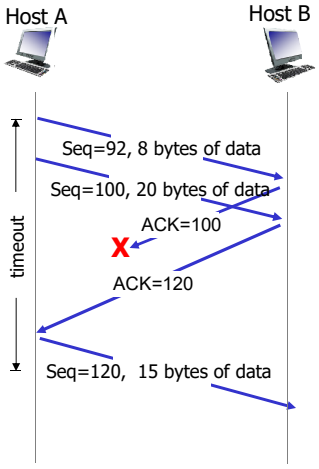


Tình huống mất ACK



Tình huống timeout sớm

TCP: Các tình huống phải truyền lại



Tình huống ACK tích lũy

Tạo ACK trong TCP [RFC 1122, RFC 2581]

Sự kiện tại bên nhận	*Hành động của TCP tại bên nhận*
Segment đến đúng thứ tự với số thứ tự mong muốn. Tất cả dữ liệu đến đã được báo nhận	ACK bị trễ. Chờ 500ms cho segment tiếp theo. Nếu không có segment tiếp theo thì gửi ACK
Segment đến đúng thứ tự với số thứ tự mong muốn. Một segment khác đang chờ ACK	Gửi ngay một ACK tích lũy, báo nhận ACK cho cả hai segment đến đúng thứ tự
Segment đến không đúng số thứ tự, số thứ tự lớn hơn mong đợi. Phát hiện có khoảng trống	Gửi ngay *ACK trùng lặp*, chỉ ra số thứ tự của byte mong đợi tiếp theo
Segment đến lấp đầy hoặc một phần khoảng trống	Gửi ngay ACK, với điều kiện là segment bắt đầu ngay tại điểm có khoảng trống

Truyền lại nhanh trong TCP

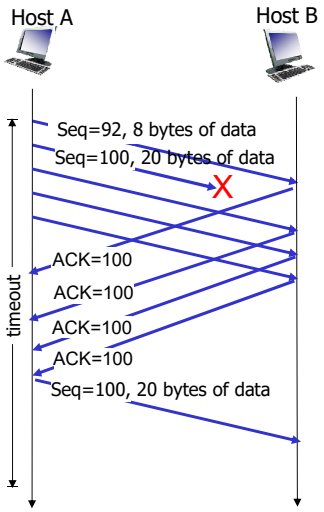
- Chu kỳ time-out thường tương đối dài:
  - Trễ dài trước khi gửi lại gói tin đã bị mất
- Phát hiện các segment bị mất qua các ACK bị trùng lặp.
  - Bên gửi thường gửi nhiều segment song song
  - Nếu segment, có thể sẽ có nhiều ACK bị trùng lặp.

Truyền lại nhanh trong TCP

Nếu bên gửi nhận được 3 ACK trùng lặp cho cùng một dữ liệu ("Ba ACK trùng lặp"), thì sẽ gửi lại segment chưa được báo nhận có số thứ tự nhỏ nhất

- Có thể là đã bị mất segment chưa được báo nhận, nên không cần phải đợi đến timeout

Truyền lại nhanh trong TCP



Truyền lại nhanh sau khi bên gửi nhận được 3 ACK trùng lặp

Chương 3: Nội dung

3.1 Các dịch vụ tầng giao vận

3.2 Ghép kênh và phân kênh

3.3 Vận chuyển không kết nối: UDP

3.4 Các nguyên lý truyền dữ liệu tin cậy

3.5 Vận chuyển hướng kết nối: TCP

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

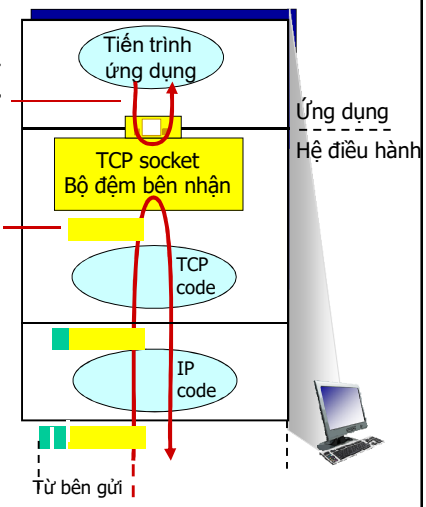
3.6 Các nguyên lý điều khiển tắc nghẽn

3.7 Điều khiển tắc nghẽn TCP

Điều khiển luồng trong TCP

Ứng dụng có thể di chuyển dữ liệu từ bộ đệm TCP socket ....

... chậm hơn so với bên nhận TCP (trong khi bên gửi vẫn đang gửi)



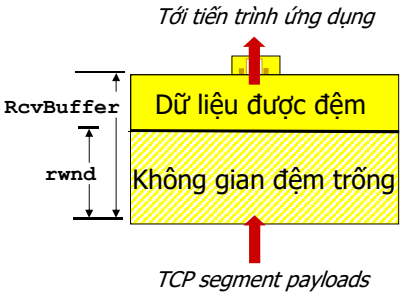
Điều khiển luồng

Bên nhận điều khiển bên gửi, để bên gửi sẽ không làm tràn bộ đệm của bên nhận do việc gửi quá nhanh và quá nhiều

Chồng giao thức phía bên nhận

Điều khiển luồng trong TCP

- Bên nhận "thông báo" không gian đệm còn trống bởi giá trị **rwnd** trong TCP header của các segment gửi-nhận
  - Kích thước **RcvBuffer** được thiết lập qua tùy chọn (option) của socket (thường mặc định là 4096 byte)
  - Nhiều hệ điều hành tự động điều chỉnh **RcvBuffer**
- Bên gửi giới hạn tổng số dữ liệu chưa được báo nhận đến bên nhận theo giá trị **rwnd**
- Đảm bảo vùng đệm nhận không bị tràn



Đệm tại phía nhận

Chương 3: Nội dung

3.1 Các dịch vụ tầng giao vận

3.2 Ghép kênh và phân kênh

3.3 Vận chuyển không kết nối: UDP

3.4 Các nguyên lý truyền dữ liệu tin cậy

3.5 Vận chuyển hướng kết nối: TCP

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

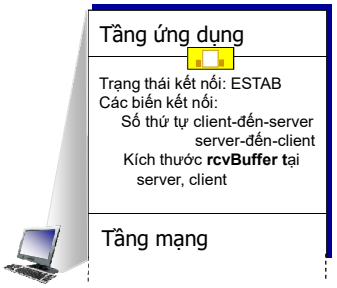
3.6 Các nguyên lý điều khiển tắc nghẽn

3.7 Điều khiển tắc nghẽn TCP

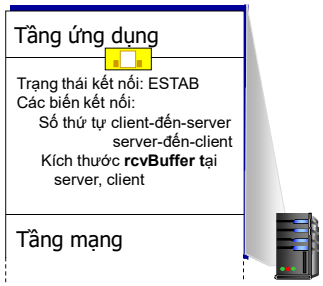
Quản lý kết nối

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

- Đồng ý thiết lập kết nối
- Đồng ý các tham số kết nối



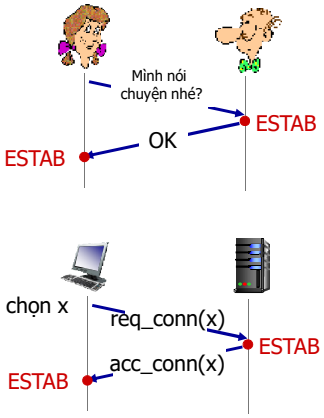

```
Socket clientSocket =
  newSocket("hostname","port
   number");
```




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

Đồng ý thiết lập kết nối

Bắt tay hai bước

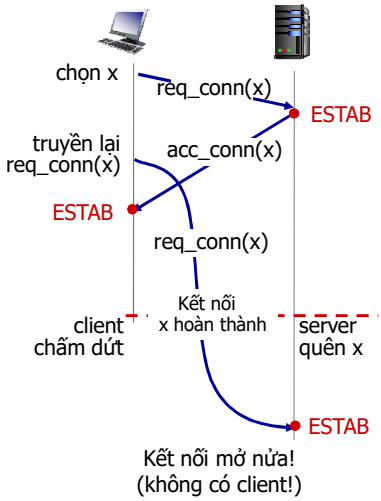


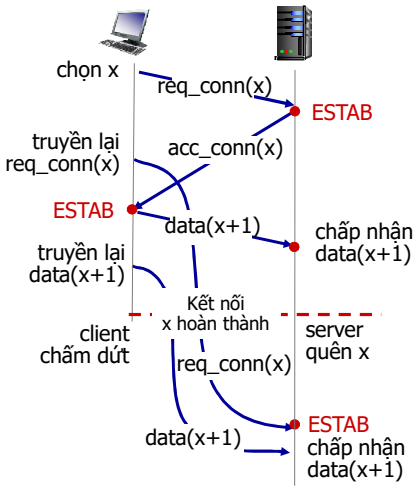
Hỏi: Bắt tay hai bước có luôn được thực hiện trên mạng?

- Chậm trễ liên quan tới các biến
- Các thông điệp được truyền lại (ví dụ, req_conn(x)) là do mất thông điệp
- Sắp xếp lại thông điệp
- Không thể "nhìn thấy" phía bên kia

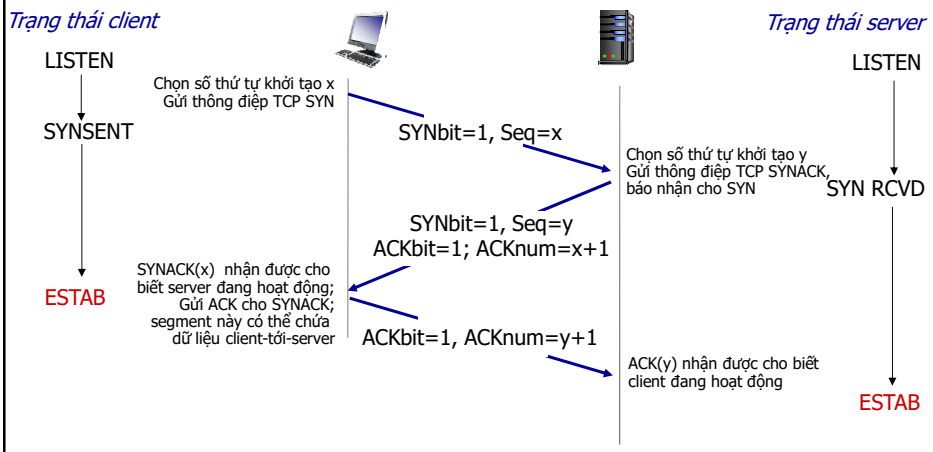
Đồng ý thiết lập kết nối

Các kịch bản có lỗi khi thực hiện bắt tay 2 bước:

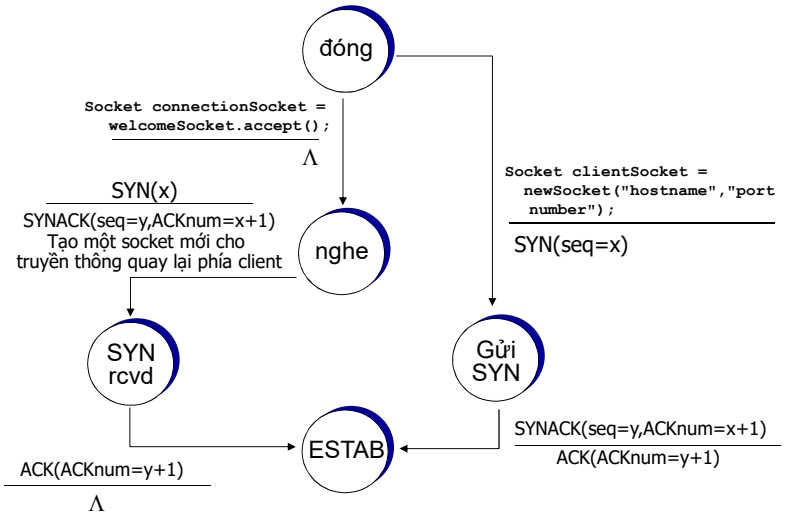




Bắt tay 3 bước trong TCP



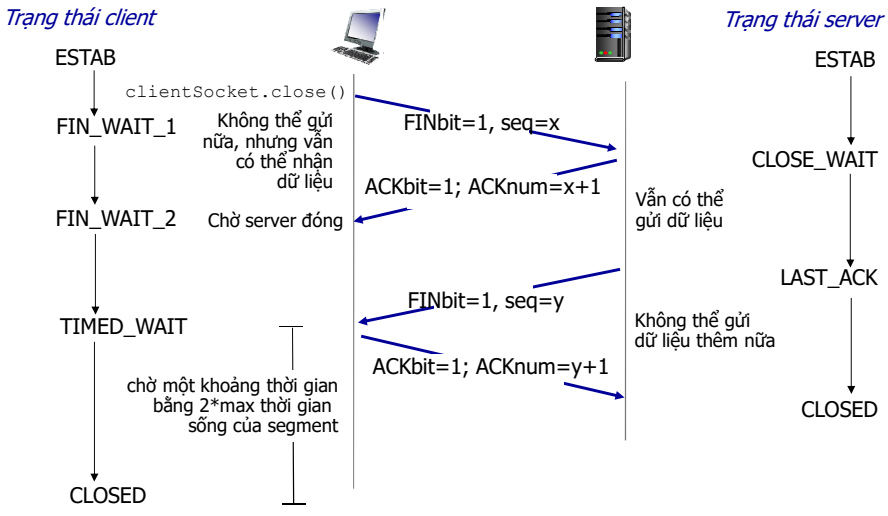
Bắt tay 3 bước trong TCP: FSM



TCP: đóng kết nối

- Mỗi bên client và server thực hiện đóng kết nối
  - Gửi TCP segment với bit FIN = 1
- Đáp ứng lại FIN nhận được bằng ACK
  - FIN, ACK đang nhận có thể được kết nối với FIN của nó
- Có thể thực hiện đồng bộ trao đổi FIN

TCP: đóng kết nối



Chương 3: Nội dung

3.1 Các dịch vụ tầng giao vận

3.2 Ghép kênh và phân kênh

3.3 Vận chuyển không kết nối: UDP

3.4 Các nguyên lý truyền dữ liệu tin cậy

3.5 Vận chuyển hướng kết nối: TCP

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

3.6 Các nguyên lý điều khiển tắc nghẽn

3.7 Điều khiển tắc nghẽn TCP

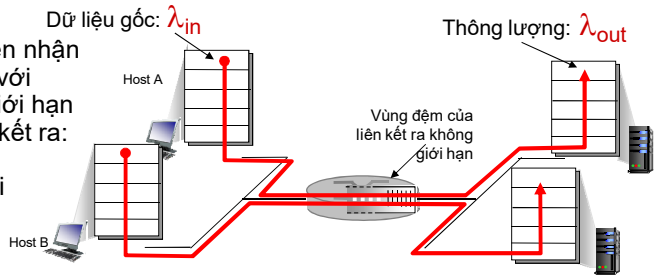
Các nguyên lý điều khiển tắc nghẽn

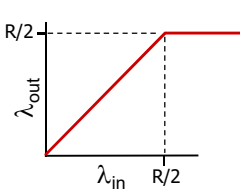
Tắc nghẽn:

- Có thể hiểu là: "quá nhiều nguồn cùng gửi quá nhiều dữ liệu với tốc độ quá nhanh tới *mạng*"
- Khác điều khiển luồng dữ liệu!
- Các biểu hiện chính:
  - Mất các gói tin (tràn bộ đệm tại các bộ định tuyến)
  - Trễ quá lâu (hàng đợi dài trong vùng đệm của bộ định tuyến)
- Là một trong mười vấn đề nan giải nhất của mạng!

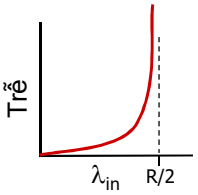
Các nguyên nhân/chi phí của tắc nghẽn: tình huống 1

- Hai bên gửi, hai bên nhận
- Một bộ định tuyến với vùng đệm không giới hạn
- Khả năng của liên kết ra: R
- Không có truyền lại





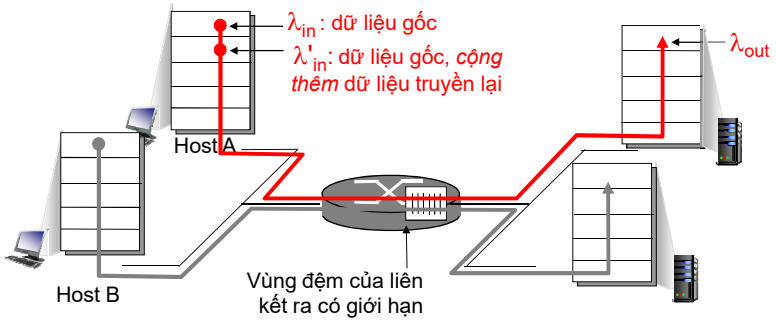
- Thông lượng lớn nhất trên mỗi kết nối: R/2



- Trễ lớn do tốc độ đến, λ_{in}, tiệm cận đến thông lượng tối đa

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

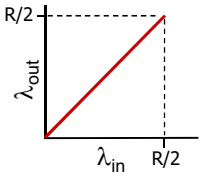
- Một bộ định tuyến, vùng đệm có *giới hạn*
- Bên gửi truyền lại gói tin bị timeout
  - Đầu vào tầng ứng dụng = đầu ra tầng ứng dụng: $\lambda_{in} = \lambda_{out}$
  - Đầu vào tầng giao vận bao gồm việc truyền lại $\lambda'_{in} \geq \lambda_{in}$

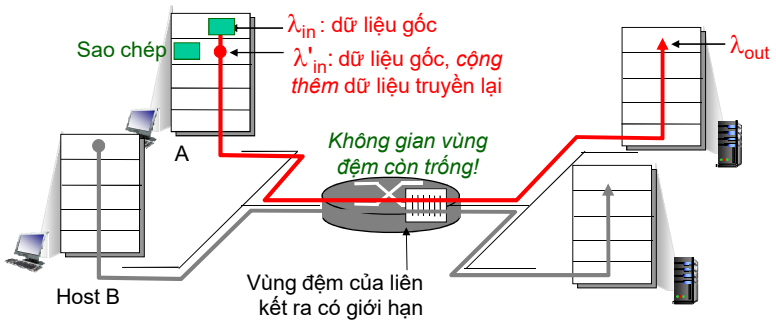


Các nguyên nhân/chi phí của tắc nghẽn: tình huống 2

Lý tưởng hóa: hiểu biết hoàn hảo

- Bên gửi chỉ gửi khi vùng đệm của bộ định tuyến sẵn sàng.

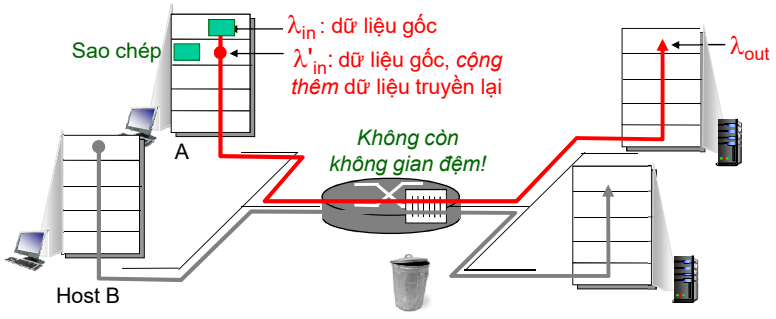




Các nguyên nhân/chi phí của tắc nghẽn: tình huống 2

Lý tưởng hóa: biết về sự mất mát

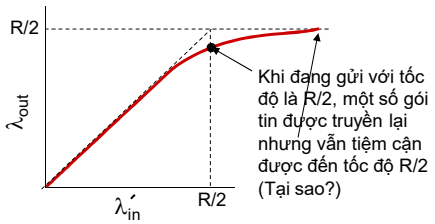
- Các gói tin có thể bị mất, bị bỏ rơi tại bộ định tuyến nếu vùng đệm của nó bị đầy
- Bên gửi chỉ gửi lại nếu gói tin *được biết* là đã bị mất

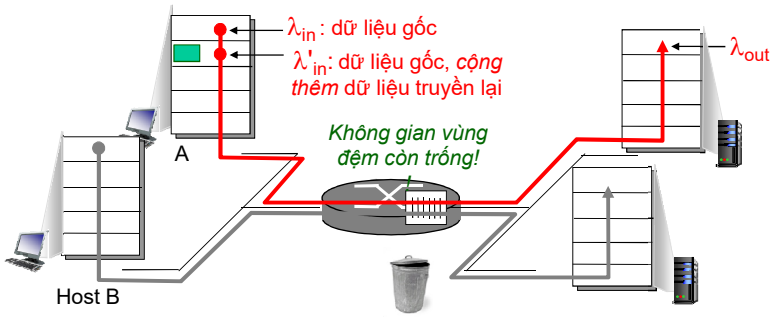


Các nguyên nhân/chi phí của tắc nghẽn: tình huống 2

Lý tưởng hóa: biết về sự mất mát

- Các gói tin có thể bị mất, bị bỏ rơi tại bộ định tuyến nếu vùng đệm của nó bị đầy
- Bên gửi chỉ gửi lại nếu gói tin *được biết* là đã bị mất

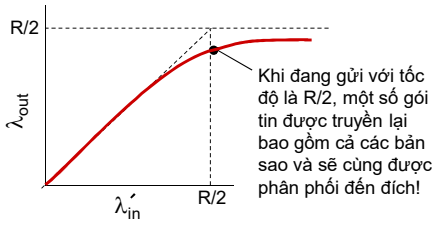


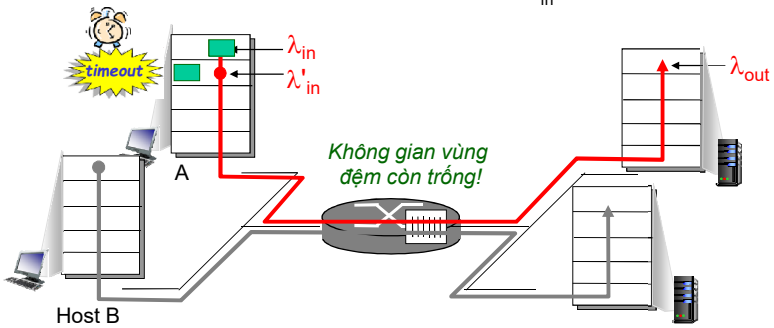


Các nguyên nhân/chi phí của tắc nghẽn: tình huống 2

Thực tế: các bản sao

- Các gói tin có thể bị mất, bị bỏ rơi tại bộ định tuyến nếu vùng đệm của nó bị đầy
- Nếu bên gửi timeout sớm, thì sẽ gửi đi *hai* bản sao của gói tin, và cả hai đều được phân phối đến đích

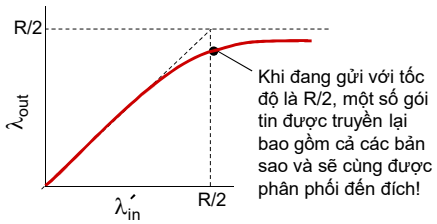




Các nguyên nhân/chi phí của tắc nghẽn: tình huống 2

Thực tế: các bản sao

- Các gói tin có thể bị mất, bị bỏ rơi tại bộ định tuyến nếu vùng đệm của nó bị đầy
- Nếu bên gửi timeout sớm, thì sẽ gửi đi *hai* bản sao của gói tin, và cả hai đều được phân phối đến đích



"Chi phí" của tắc nghẽn:

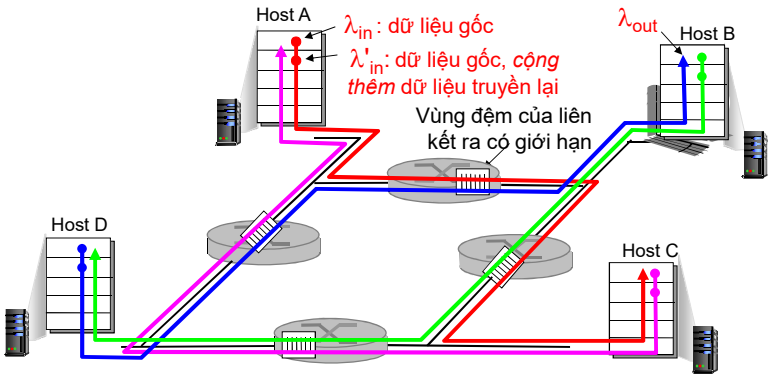
- Nhiều việc (truyền lại), với lưu lượng xác định
- Không cần thiết phải truyền lại: liên kết mang nhiều bản sao của gói tin
  - Làm giảm lưu lượng

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

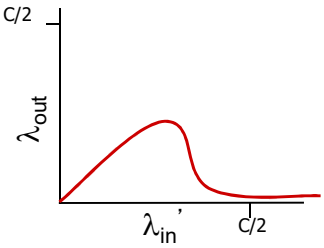
- Bốn bên gửi
- Nhiều đường đến đích
- timeout/truyền lại

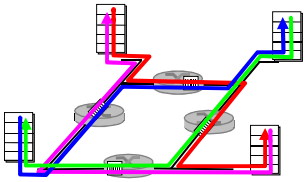
Hỏi: Điều gì sẽ xảy ra khi λ_{in} và λ_{in}' tăng lên?

Trả lời: Nếu λ_{in}' (đỏ) tăng lên, thì tất cả gói tin màu xanh nước biển đang đến tại hàng đợi phía trên sẽ bị bỏ rơi, thông lượng màu xanh nước biển sẽ tiến đến 0



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





"Chi phí" khác của tắc nghẽn:

- Khi gói tin bị bỏ rơi, thì bất kỳ luồng lưu lượng truyền nào cho gói tin đều là lãng phí!

Phương pháp tiếp cận hướng tới điều khiển tắc nghẽn

Hai cách tiếp cận chính hướng tới điều khiển tắc nghẽn:

Điều khiển tắc nghẽn end-end:

- Không có phản hồi rõ ràng từ mạng
- Tắc nghẽn được suy ra từ hiện tượng mất mát hoặc trễ quan sát được tại hệ thống đầu cuối
- Cách tiếp cận này được thực hiện bởi TCP

Điều khiển tắc nghẽn có hỗ trợ từ mạng:

- Các bộ định tuyến cung cấp phản hồi tới các hệ thống đầu cuối.
  - bit đơn chỉ thị tắc nghẽn (SNA, DECbit, TCP/IP ECN, ATM)
  - Tốc độ gửi được xác định rõ ràng

Case study: điều khiển tắc nghẽn trong ATM ABR

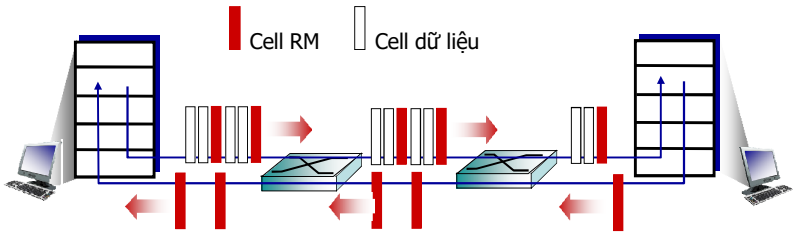
ABR: tốc độ bit có sẵn:

- "Dịch vụ mềm dẻo"
- Nếu đường dẫn phía bên gửi "dưới tải" thì:
  - Bên gửi nên dùng băng thông có sẵn
- Nếu đường dẫn bên gửi bị tắc nghẽn thì:
  - Bên gửi nên giảm để đảm bảo tốc độ là tối thiểu

Các cell RM (quản lý tài nguyên):

- Được gửi bởi bên gửi, xen kẽ với các cell dữ liệu
- Các bit trong cell RM được thiết lập bởi các switch ("*có hỗ trợ từ mạng*")
  - *bit NI:* không tăng theo tốc độ (tắc nghẽn nhẹ)
  - *bit CI:* xác định tắc nghẽn
- Các cell RM được trả lại bên gửi từ bên nhận, với các bit còn nguyên vẹn

Case study: điều khiển tắc nghẽn trong ATM ABR



- Hai byte trường ER (explicit rate) trong cell RM
  - Switch bị tắc nghẽn có thể có giá trị ER thấp hơn trong cell
  - Bên gửi gửi với tốc độ được hỗ trợ lớn nhất trên đường truyền
- Bit EFCI trong các cell dữ liệu: được thiết lập là 1 trong switch bị tắc nghẽn
  - Nếu cell dữ liệu trước cell RM có EFCI được thiết lập, thì bên nhận thiết lập bit CI trong cell RM được trả về

Chương 3: Nội dung

3.1 Các dịch vụ tầng giao vận

3.2 Ghép kênh và phân kênh

3.3 Vận chuyển không kết nối: UDP

3.4 Các nguyên lý truyền dữ liệu tin cậy

3.5 Vận chuyển hướng kết nối: TCP

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

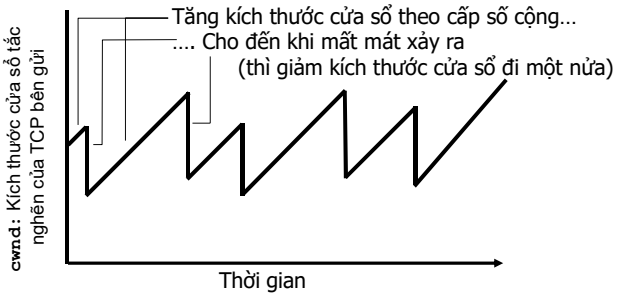
3.6 Các nguyên lý điều khiển tắc nghẽn

3.7 Điều khiển tắc nghẽn TCP

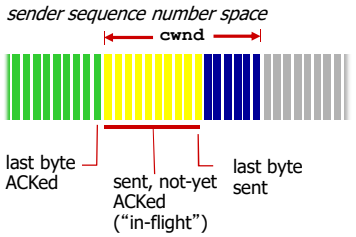
Điều khiển tắc nghẽn trong TCP: Tăng theo cấp số cộng Giảm theo cấp số nhân

- *Cách tiếp cận:* Bên gửi tăng tốc độ truyền (kích thước cửa sổ), thăm dò băng thông sử dụng, cho đến khi có mất mát xảy ra
  - *Tăng theo cấp số cộng:* tăng **cwnd** theo 1 MSS mỗi RTT cho đến khi phát hiện mất mát
  - *Giảm theo cấp số nhân*: giảm **cwnd** đi một nửa sau khi phát hiện có mất mát

Thăm dò băng thông



Chi tiết điều khiển tắc nghẽn trong TCP



- Bên gửi giới hạn việc truyền:

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

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

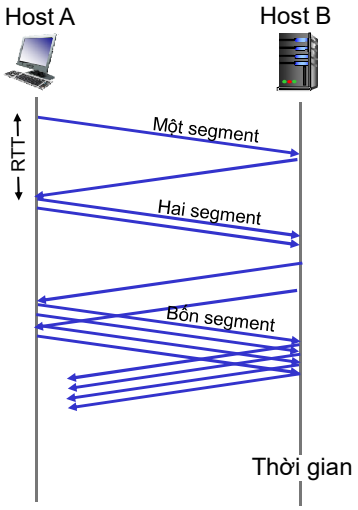
Tốc độ gửi của TCP:

- *Được hiểu là:* gửi cwnd byte, chờ một RTT cho ACK, sau đó gửi nhiều byte hơn

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

TCP khởi động chậm

- Khi kết nối bắt đầu, tăng tốc độ lên theo cấp số nhân cho đến khi có sự kiện mất mát đầu tiên xảy ra:
  - Khởi tạo **cwnd** = 1 MSS
  - Tăng gấp đôi **cwnd** cho mỗi RTT
  - Thực hiện tăng **cwnd** cho mỗi ACK nhận được
- *Tổng kết:* tốc độ khởi đầu là chậm nhưng sau đó tăng lên theo cấp số nhân



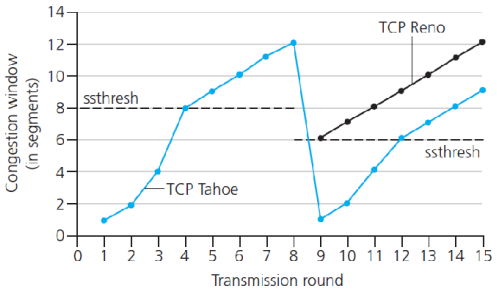
Phát hiện và phản ứng lại khi có mất mát

- Mất mát được xác định khi bị timeout:
  - **cwnd** được thiết lập lại là 1 MSS;
  - Cửa sổ sau đó sẽ tăng theo cấp số nhân (như trong khởi động chậm) tới ngưỡng, thì sẽ tăng tuyến tính
- Mất mát được xác định khi thấy 3 ACK trùng lặp: TCP RENO
  - Các ACK trùng lặp xác định khả năng truyền các segment của mạng
  - **cwnd** giảm đi một nửa kích thước cửa sổ, sau đó tăng tuyến tính
- TCP Tahoe luôn đặt **cwnd** là 1 (khi có timeout hoặc 3 ACK trùng lặp)

Hiện thực trong TCP

Hỏi: Khi nào nên chuyển từ tăng theo cấp số nhân sang tăng tuyến tính?

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



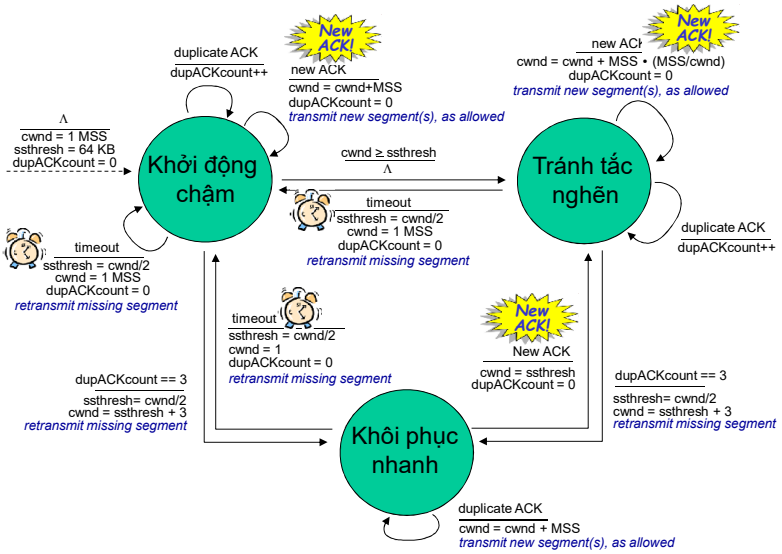
Cài đặt:

- Biến **ssthresh**
- Với mỗi sự kiện mất mát, **ssthresh** sẽ được đặt bằng 1/2 **cwnd** ngay trước khi có mất mát xảy ra

Tổng kết điều khiển tắc nghẽn trong TCP

- Khi **cwnd** dưới **ssthresh**, bên gửi đang trong giai đoạn khởi động chậm, kích thước cửa sổ tăng nhanh theo cấp số nhân.
- Khi **cwnd** trên **ssthresh**, 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, **ssthresh** = **cwnd/2** và **cwnd** = **ssthresh**.
- Khi timeout xảy ra, **ssthresh** = **cwnd/2** và **cwnd=1 MSS**.

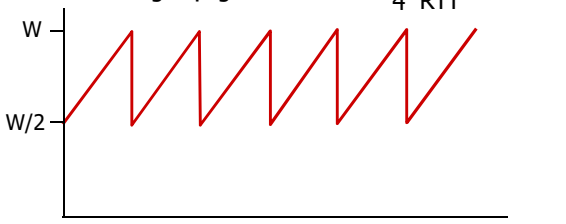
Tổng kết điều khiển tắc nghẽn trong TCP



Thông lượng của TCP

- Thông lượng trung bình của TCP được xác định qua kích thước cửa sổ và RTT như thế nào?
  - Bỏ qua khởi động chậm, giả sử dữ liệu luôn luôn được gửi
- W: kích thước cửa sổ (được tính bằng byte) khi có mất mát xảy ra
  - Kích thước cửa sổ trung bình (số byte trong lưu lượng) là ¾ W
  - Thông lượng trung bình là ¾ W trên RTT

$$\text{Thông lượng TB của TCP} = \frac{3}{4}\ \frac{W}{RTT}\ \text{bytes/sec}$$



TCP trong tương lai: TCP qua "đường truyền rộng và dài"

- Ví dụ: Các segment dài 1500 byte, RTT là 100ms, muốn đạt được thông lượng là10 Gbps
- Yêu cầu lưu lượng với kích thước cửa sổ là W = 83,333 segment
- Thông lượng của xác suất mất đoạn là L [Mathis 1997]:

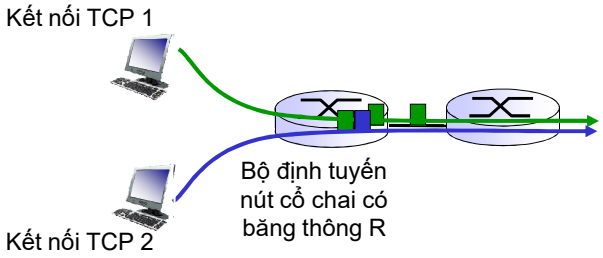
$$\text{Thông lượng TCP} = \frac{1.22 \cdot MSS}{RTT\ \sqrt{L}}$$

➜ Để có được thông lượng là 10 Gbps, cần tỷ lệ mất mát là $L = 2 \cdot 10^{-10}$ *– một tỷ lệ mất mát rất nhỏ!*

- Các phiên bản mới của TCP dành cho tốc độ cao

Tính công bằng trong TCP

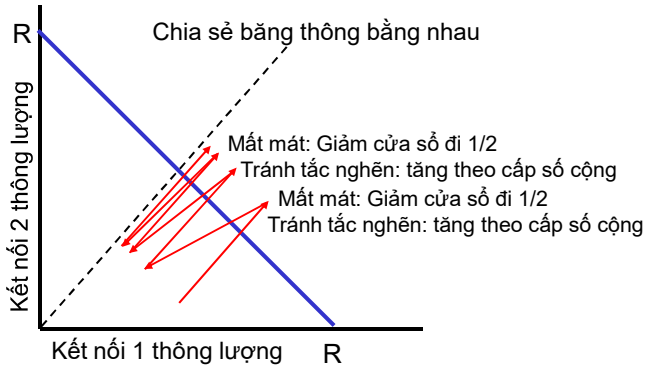
Mục tiêu: Nếu K phiên làm việc trong TCP chia sẻ cùng liên kết nút cổ chai có băng thông là R, thì mỗi phiên nên có tốc độ trung bình là R/K



Tại sao TCP là công bằng?

Hai phiên làm việc cạnh tranh nhau:

- Tăng theo cấp số cộng làm tăng lưu lượng liên tục
- Giảm theo cấp số nhân làm giảm lưu lượng tương ứng



Tính công bằng (tiếp)

Tính công bằng và UDP

- Các ứng dụng đa phương tiện thường không dùng TCP
  - Không muốn tốc độ bị chặn do điều khiển tắc nghẽn
- Thay bằng dùng UDP:
  - Gửi audio/video với tốc độ ổn định, chịu mất mát gói tin

Tính công bằng và kết nối song song trong TCP

- Ứng dụng có thể mở nhiều kết nối song song giữa hai host
- Các trình duyệt web làm theo cách này
- Ví dụ: liên kết có 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

Chương 3: Tổng kết

- Các nguyên lý của các dịch vụ tầng giao vận:
  - Ghép kênh, phân kênh
  - Truyền dữ liệu tin cậy
  - Điều khiển luồng
  - Điều khiển tắc nghẽn
- Hiện thực trên mạng Internet:
  - UDP
  - TCP

Tiếp theo:

- Kết thúc các vấn đề liên quan đến "phần cạnh" của mạng (tầng ứng dụng và tầng giao vận)
- Chuẩn bị đi vào "phần lõi" của mạng