

HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG
Posts and Telecommunications Institute of Technology

MẠNG MÁY TÍNH

Chương 3: Tầng giao vận – Transport Layer

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

Nội dung

- Giới thiệu các dịch vụ tầng giao vận
- Ghép kênh và phân kênh
- Vận chuyển không hướng kết nối: UDP
- Các nguyên tắc truyền dữ liệu tin cậy
- Vận chuyển hướng kết nối: TCP
- Các nguyên tắc điều khiển tắc nghẽn
- Điều khiển tắc nghẽn TCP

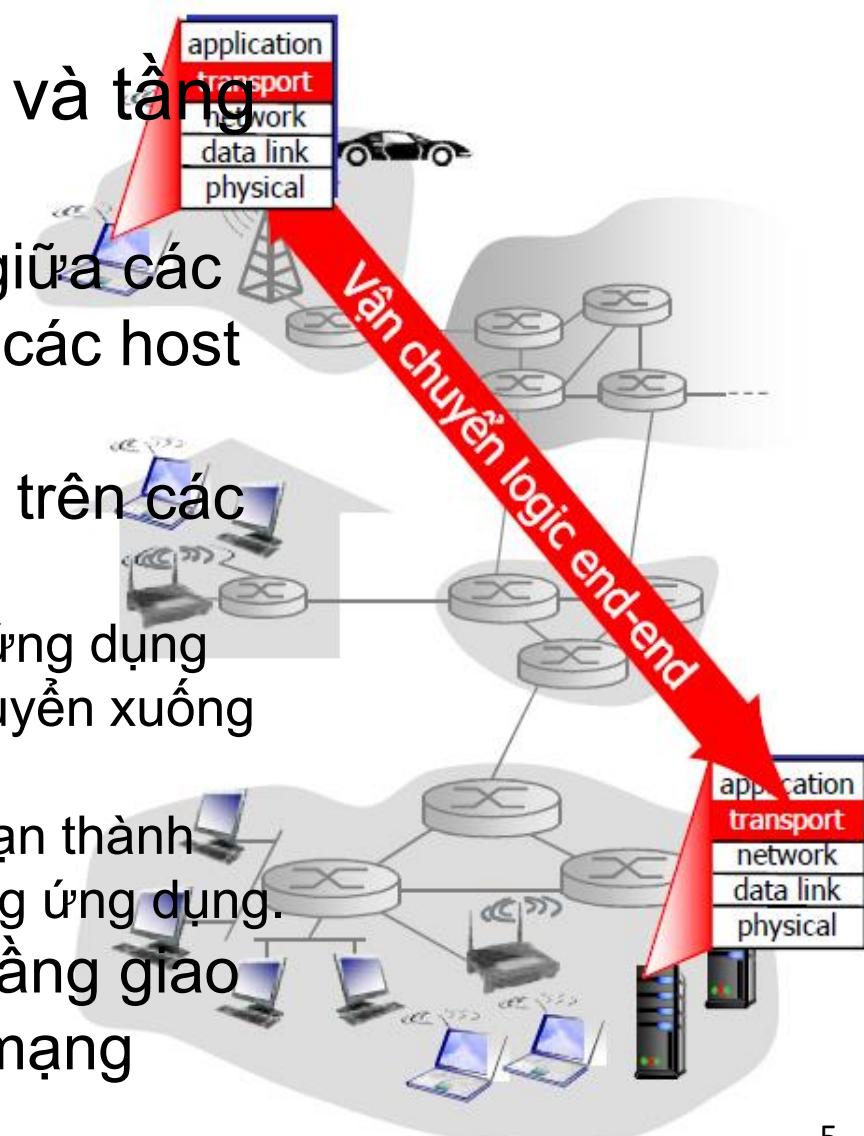
Nội dung

- Giới thiệu các dịch vụ tầng giao vận
- Ghép kênh và phân kênh
- Vận chuyển không kết nối: UDP
- Các nguyên tắc truyền dữ liệu tin cậy
- Vận chuyển hướng kết nối: TCP
- Các nguyên lý điều khiển tắc nghẽn
- Điều khiển tắc nghẽn TCP

Các dịch vụ và giao thức tầng giao vận

- Quan hệ giữa tầng giao vận và tầng mạng

- 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: chia 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 mạng
 - ✓ Internet: TCP và UDP



Tầng giao vận và tầng mạng

- Quan hệ giữa 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

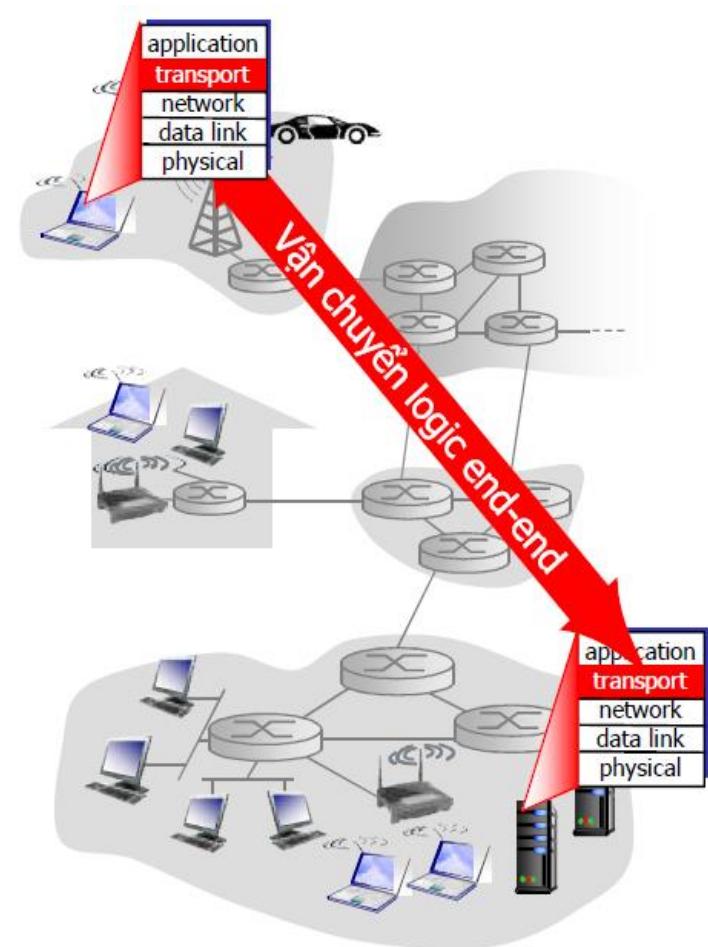
Tình huống tương tự: _____

- 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

• Tầng giao vận trong Internet

- Nhiệm vụ chính của UDP và TCP là mở rộng dịch vụ phân phối của IP giữa 2 host thành dịch vụ phân phối giữa hai tiến trình chạy trên 2 host đó
- Truyền tin cậy, theo trật tự: TCP
 - ✓ Điều khiển tắc nghẽn
 - ✓ Điều khiển luồng
 - ✓ Thiết lập kết nối
- Truyền không tin cậy, không theo thứ tự: UDP
 - ✓ Truyền dữ liệu giữa các tiến trình
 - ✓ Kiểm tra lỗi
- Không có các dịch vụ:
 - ✓ Đảm bảo trễ
 - ✓ Đảm bảo băng thông



Nội dung

- Các dịch vụ tầng giao vận
- **Ghép kênh và phân kênh**
- Vận chuyển không kết nối: UDP
- Các nguyên tắc truyền dữ liệu tin cậy
- Vận chuyển hướng kết nối: TCP
- Các nguyên lý điều khiển tắc nghẽn
- Điều khiển tắc nghẽn TCP

- LLC: **Multiplexing/De-Multiplexing.**



**MULTIPLEXING
and
DEMULTIPLEXING**

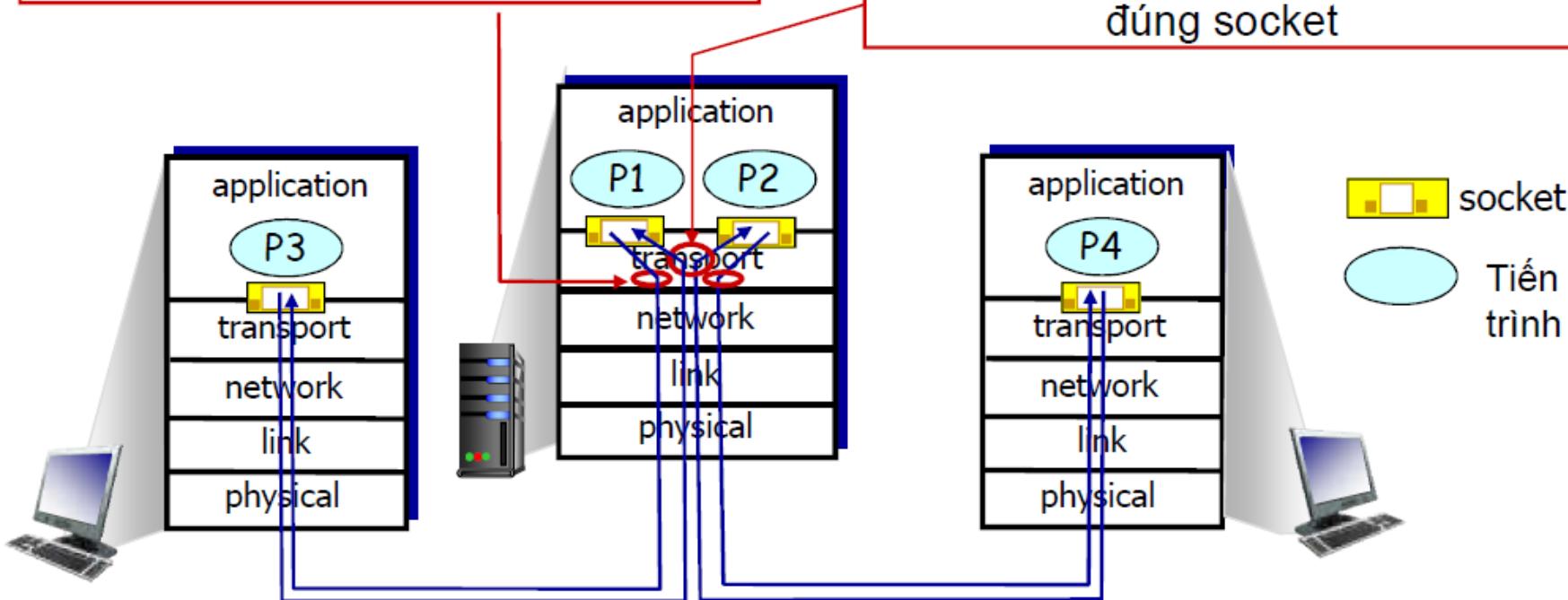
Ghép kênh/Phân kênh

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)

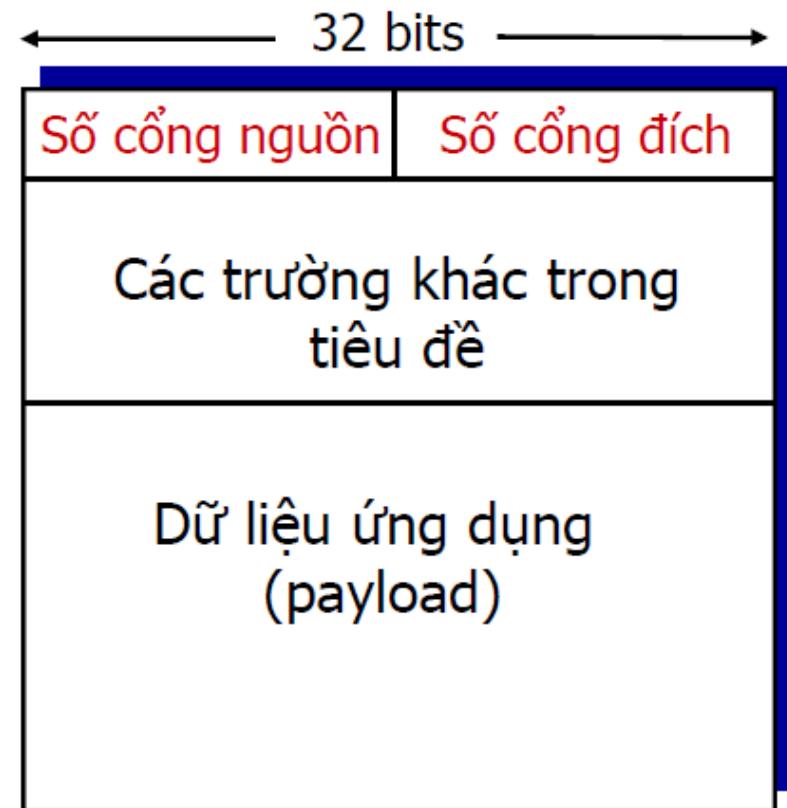
Phân kênh tại phía nhận:

Sử dụng thông tin trong phân tiêu đề để phân phối các đoạn dữ liệu (segment) đã nhận được đến đúng socket



Ghép kênh/Phân kênh

- 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

Ghép kênh/Phân kênh

- Ghép kênh và phân kênh không hướng kết nối (UDP)
 - Một host tạo một UDP socket
clientSocket = socket(AF_INET, SOCK_DGRAM)
 - Tầng giao vận sẽ gán một port number cho socket trên (từ 1024 đến 65535)
 - Hoặc tự gán *clientSocket.bind(("", 19157))*

Ghép kênh/Phân kênh

- Ghép kênh và phân kênh không hướng kết nối (UDP)
 - Host A (UDP port 1957) muốn gửi một chunk của application data tới một tiến trình UDP port 46428 ở Host B
 - Tầng giao vận ở host A tạo một segment (gồm data, **source port number 1957, destination port number 46428**) và 2 giá trị khác (đề cập sau)
 - Tầng giao vận chuyển segment tới tầng mạng -> tầng mạng đóng gói segment thành datagram và chuyển sang đầu nhận

Ghép kênh/Phân kênh

- Ghép kênh và phân kênh không hướng kết nối (UDP)
 - Host B giả sử nhận được segment
 - Tầng giao vận kiểm tra destination port number trong segment (46428) và chuyển segment này tới socket được định danh bởi port 46428
 - Host B có thể chạy nhiều tiến trình, mỗi tiến trình có socket UDP của chính nó với port number tương ứng. Khi các segment UDP đến host B từ tầng mạng, Host B sẽ chuyển (phân kênh) từng segment này tới socket tương ứng bằng cách kiểm tra destination port number của segment này.

Ghép kênh/Phân kênh

- Ghép kênh và phân kênh không hướng kết nối (UDP)
 - UDP socket được định danh đầy đủ bởi 2 thông tin
 - Destination IP address
 - Destination port number
 - Do đó, nếu 2 UDP segment có **IP nguồn** và **port number nguồn khác nhau** nhưng có **cùng destination IP** và **destination port number** thì 2 segment này sẽ được chuyển tới **cùng tiến trình destination** qua **cùng destination socket**

Ghép kênh/Phân kênh

- Ghép kênh và 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
 - Ngược với UDP, 2 TCP segment tới từ các IP nguồn và port nguồn khác nhau sẽ được chuyển tới 2 socket khác nhau

Ghép kênh/Phân kênh

- Ghép kênh và phân kênh hướng kết nối
 - Client

serverName = 'servername'

serverPort = 12000

clientSocket = socket(AF_INET, SOCK_STREAM)

clientSocket.connect((serverName, serverPort))

- Server

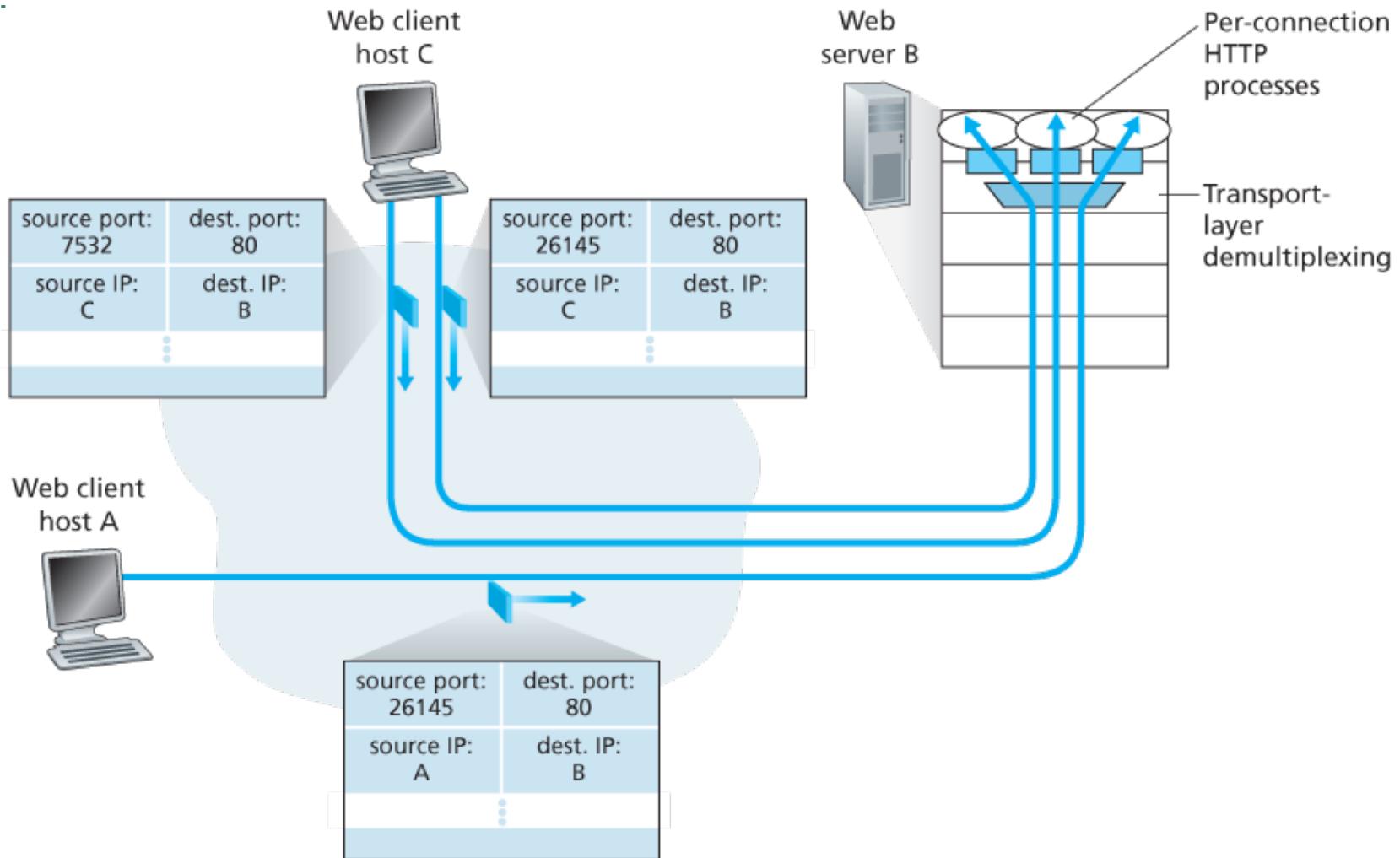
connectionSocket, addr = serverSocket.accept()

Ghép kênh/Phân kênh

- Phân kênh hướng kết nối
 - Host server có thể hỗ trợ nhiều TCP socket đồng thời:
 - Mỗi socket tương ứng với một tiến trình và được định danh bởi bộ-4 giá trị

Ghép kênh/Phân kênh

- Web Servers và TCP



Nội dung

- Các dịch vụ tầng giao vận
- Ghép kênh và phân kênh
- **Vận chuyển không kết nối: UDP**
- Các nguyên tắc truyền dữ liệu tin cậy
- **Vận chuyển hướng kết nối: TCP**
- Các nguyên lý điều khiển tắc nghẽn
- 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

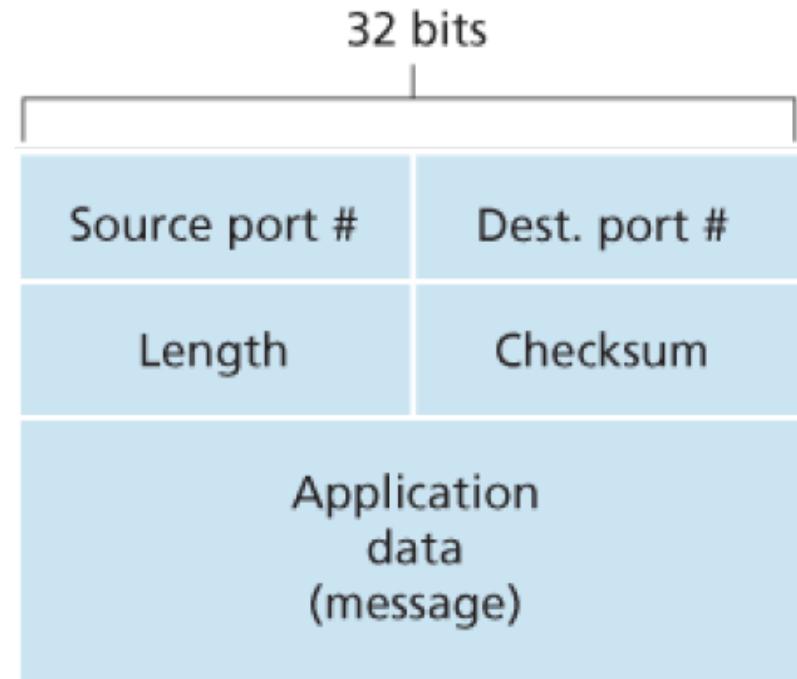
- UDP: User Datagram Protocol [RFC 768]
 - UDP được dùng trong:
 - Các ứng dụng streaming multimedia (chịu mất mát dữ liệu, bị ảnh hưởng bởi tốc độ)
 - DNS
 - SNMP
 - Truyền tin cậy trên UDP:
 - Bổ sung đặc tính tin cậy vào tầng ứng dụng
 - Khôi phục lỗi cụ thể của ứng dụng

- Cấu trúc UDP segment

- Tiêu đề Header

- Số hiệu cổng nguồn
 - Số hiệu cổng đích
 - Độ dài
 - Mã kiểm tra

- Data



- UDP checksum
 - *Mục tiêu:* Phát hiện các “lỗi”
 - Xác định xem liệu các bít trong UDP segment có bị thay đổi trong quá trình truyền từ nguồn tới đích (do nhiễu trong đường truyền hay khi lưu ở router)
 - Bên gửi:
 - Xử lý nội dung các segment, bao gồm cả các trường trong tiêu đề, dưới dạng chuỗi các số nguyên 16-bit
 - checksum: bổ sung thêm (tổng bù 1) vào nội dung segment
 - Bên gửi đặt giá trị checksum vào trong trường checksum của UDP

- UDP checksum
 - Bên nhận:
 - Tính tổng các số nguyên 16 bit nhận được, gồm cả checksum
 - Tính toán checksum của segment đã nhận được
 - Nếu không có lỗi, tổng sẽ là 1111111111111111
 - Nếu một trong số 16 bit của tổng = 0 => có lỗi.

- 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
<hr/>																
Bit dư	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1
<hr/>																
Tổng checksum	1	0	1	1	1	0	1	1	1	0	1	1	1	0	0	
	0	1	0	0	0	1	0	0	0	1	0	0	0	1	1	

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

Review

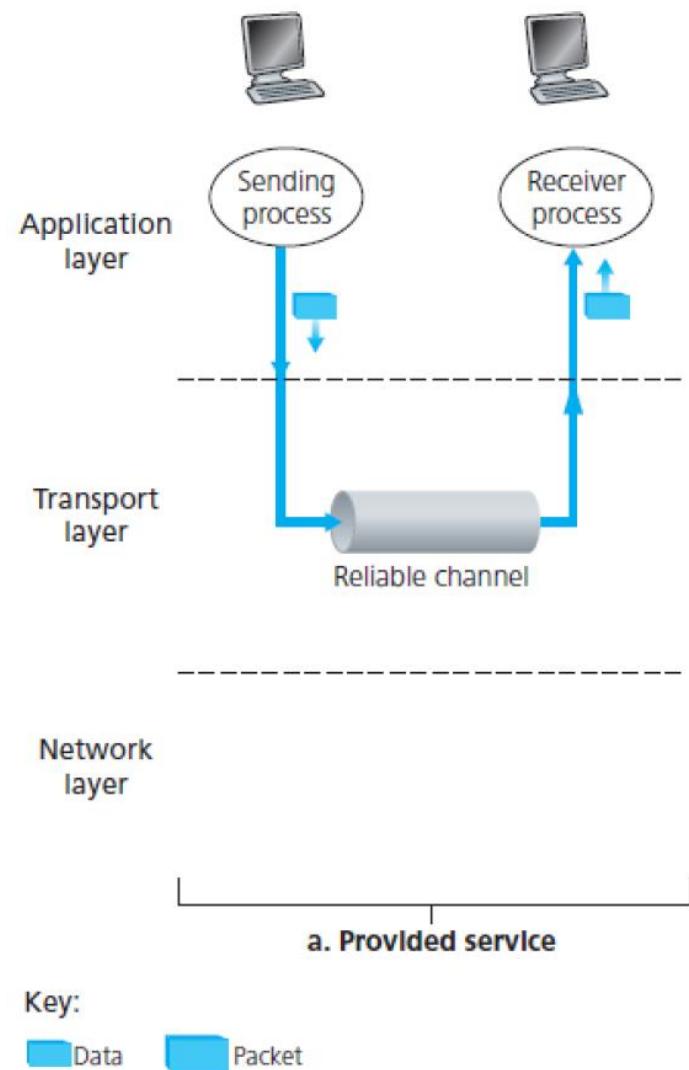
- Phân biệt truyền thông logic ở tầng giao vận và truyền thông logic ở tầng mạng?
- Tên gọi dữ liệu ở tầng giao vận?
- Ý nghĩa của việc dồn kênh và phân kênh ở tầng giao vận?
- Dồn kênh là gì? Được thực hiện ở đầu gửi hay nhận?
- Phân kênh là gì? Được thực hiện ở đầu gửi hay nhận.
- Hai giao thức quan trọng ở tầng giao vận Internet? Phân biệt chúng

Nội dung

- Các dịch vụ tầng giao vận
- Ghép kênh và phân kênh
- Vận chuyển không kết nối: UDP
- **Các nguyên tắc truyền dữ liệu tin cậy**
- Vận chuyển hướng kết nối: TCP
- Các nguyên lý điều khiển tắc nghẽn
- Điều khiển tắc nghẽn TCP

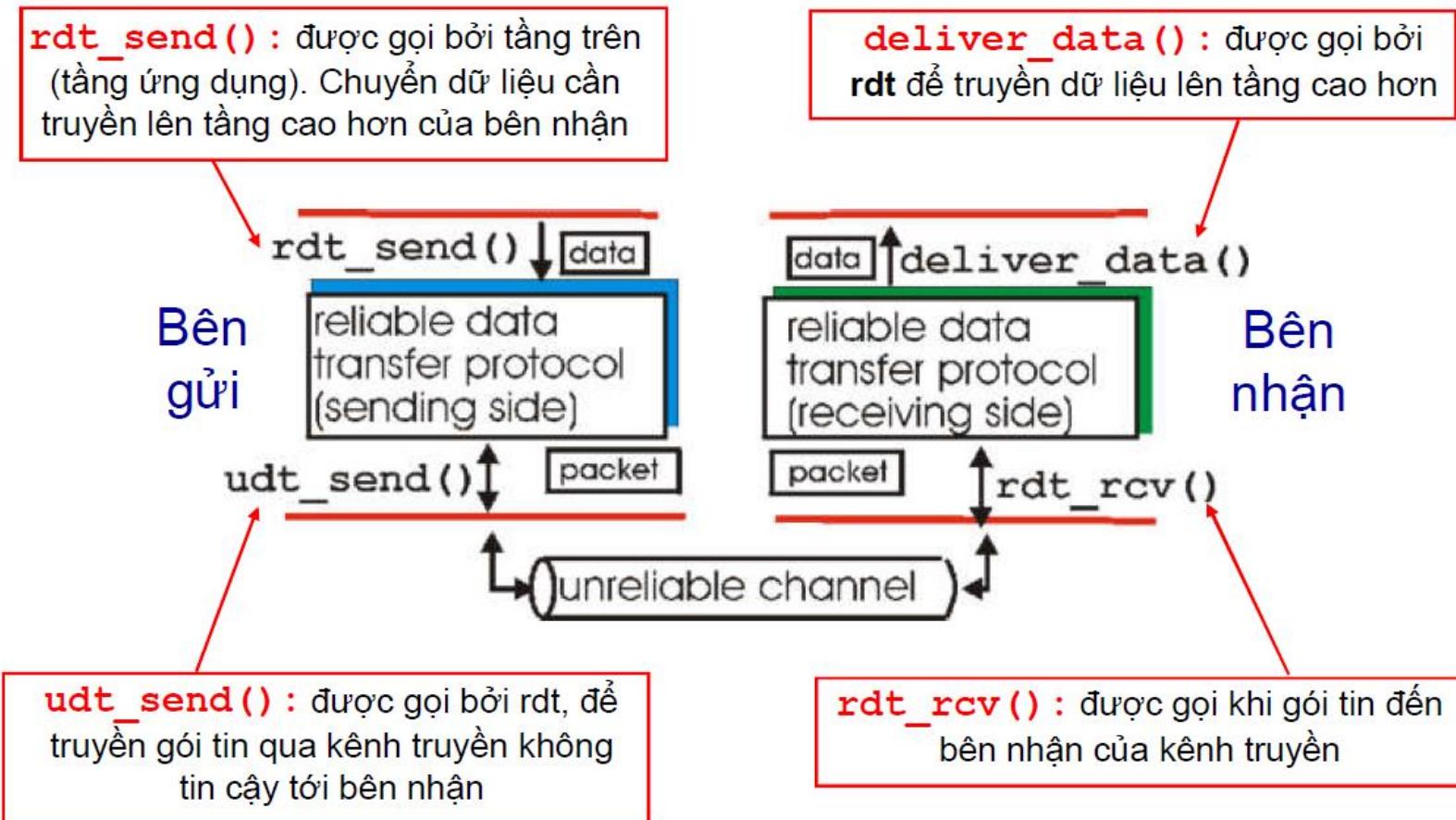
Các nguyên tắc của truyền DL tin cậy

- Quan trọng trong các tầng ứng dụng, giao vận và liên kết
 - Thuộc danh sách 10 vấn đề quan trọng nhất của mạng!
- Giao thức truyền dữ liệu tin cậy (reliable data transfer protocol – rdt)
 - 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 DL tin cậy

- Các giao diện cho rdt



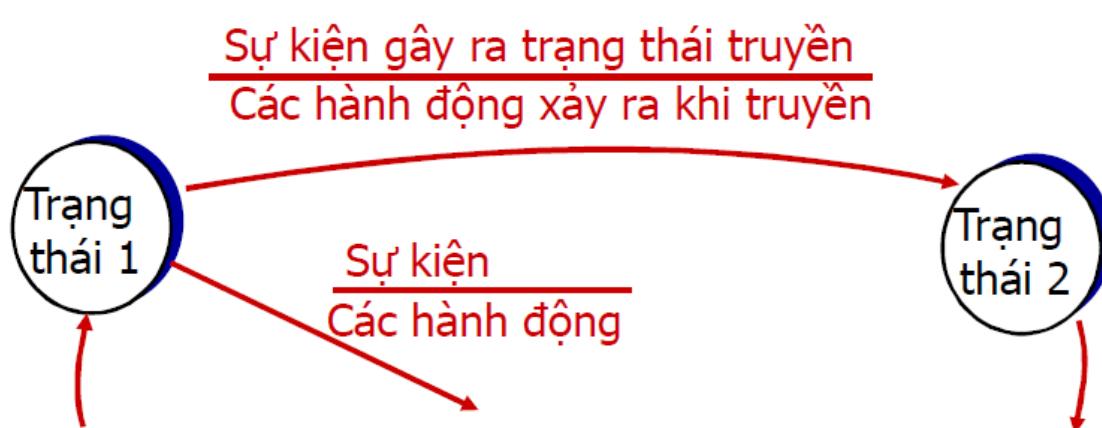
Các nguyên lý của truyền DL tin cậy

• Truyền dữ liệu tin cậy

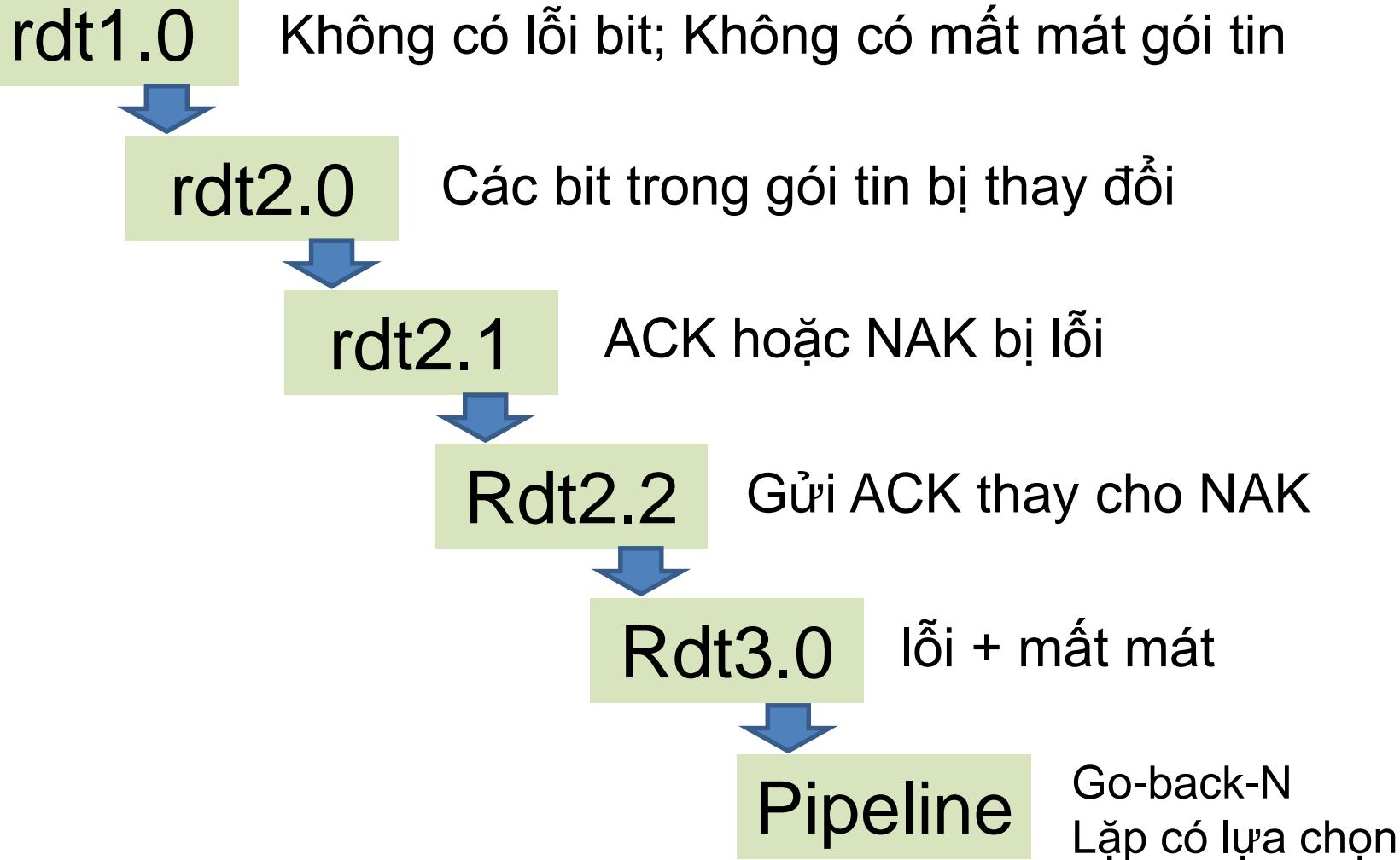
– Việc cần làm:

- Phát triển 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 hoạt động bên gửi, bên nhận.

Trạng thái: khi đang ở một “trạng thái” thì trạng thái duy nhất kế tiếp sẽ được xác định khi có sự kiện tiếp theo

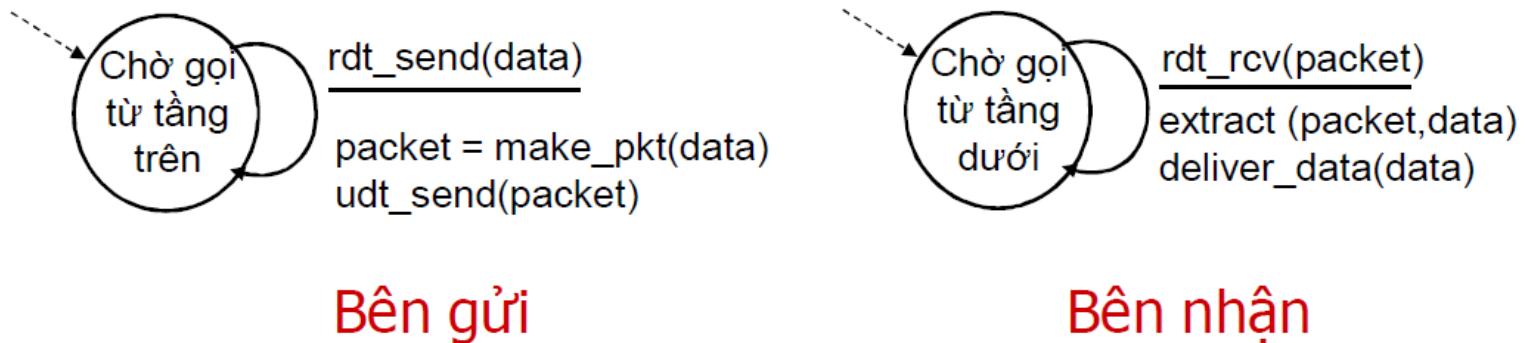


Review: Các nguyên lý của truyền DL tin cậy



Các nguyên lý của truyền DL tin cậy

- 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



Các nguyên lý của truyền DL tin cậy

- rdt2.0: Kênh truyền có lỗi bit
 - Các bit trong gói tin bị thay đổi
 - Lỗi thường xảy ra khi truyền trên đường truyền vật lý hoặc khi lưu ở bộ nhớ đệm (router)
 - Giả thiết tất cả các gói tin được truyền (dù có một số bit bị lỗi) vẫn được nhận theo đúng trật tự ban đầu

Làm thế nào con người khắc phục được “lỗi” trong suốt quá trình thực hiện cuộc nói chuyện qua điện thoại?

ARQ (Automatic Repeat reQuest) protocols

Các nguyên lý của truyền DL tin cậy

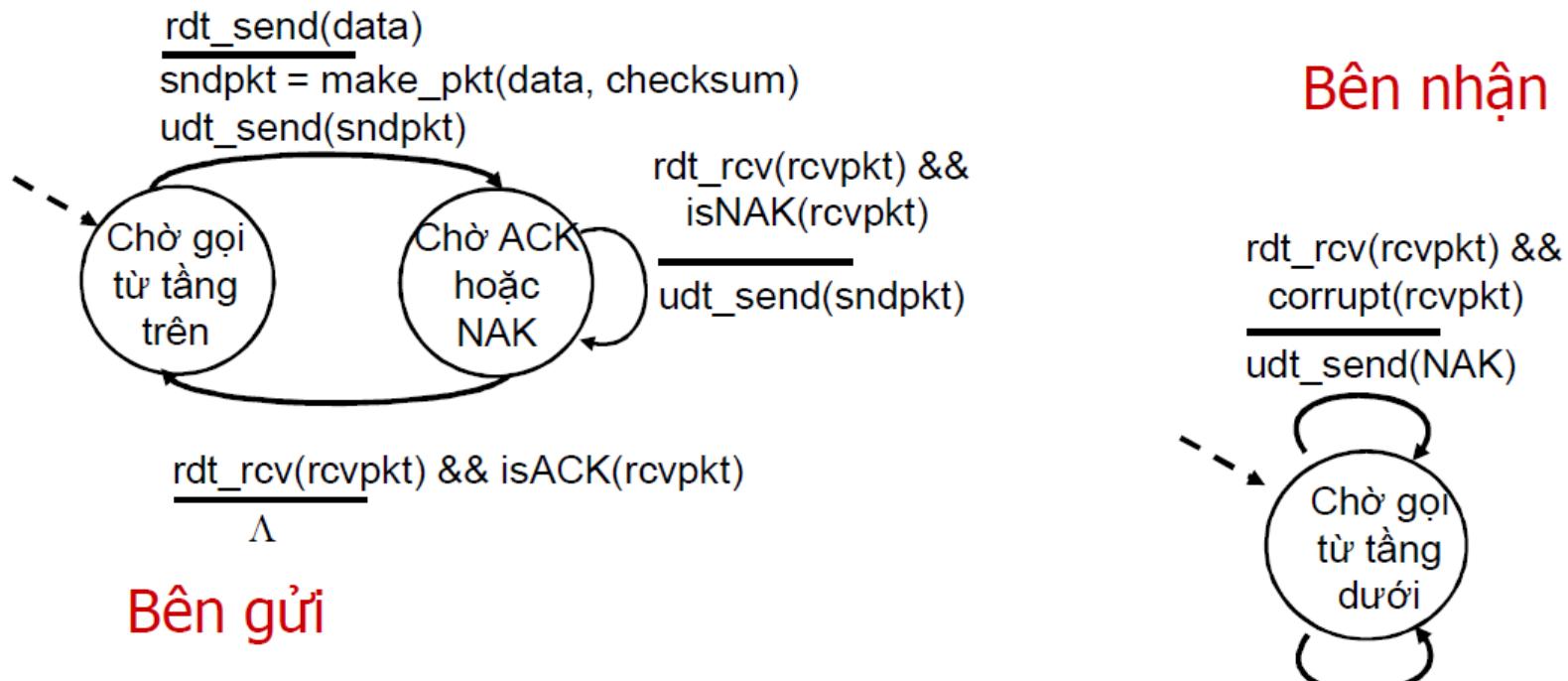
- rdt2.0: Kênh truyền có lỗi bit
 - Để xử lý bít lỗi, về cơ bản có 3 tính năng được bổ sung trong ARQ
 - Bên nhận phát hiện lỗi
 - Bên nhận biết được bít nào mình nhận được bị lỗi
 - » UDP sử dụng checksum
 - Bên nhận gửi phản hồi (Receiver feedback)
 - Báo nhận ACK (acknowledgement): bên nhận thông báo rõ 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
 - Các gói ACK hay NAK chỉ có độ dài 1 bit: 0 - NAK và 1 – ACK
 - Bên gửi truyền lại gói tin có báo nhận là NAK

Các nguyên lý của truyền DL tin cậy

- rdt2.0: Kênh truyền có lỗi bit
 - Các cơ chế mới trong rdt2.0 (so với rdt1.0)
 - Phát hiện lỗi
 - Phản hồi: các thông điệp điều khiển (ACK, NAK) từ bên nhận gửi về cho bên gửi

Các nguyên lý của truyền DL tin cậy

- rdt2.0: Đặc tả FSM



rdt2.0 còn được gọi là giao thức stop-and-wait

Các nguyên lý của truyền DL tin cậy

- rdt2.0: Lỗi hỏng
 - Gói ACK hay NAK bị lỗi => người gửi sẽ không biết được bên nhận đã nhận đúng đoạn dữ liệu cuối cùng hay chưa
 - Khắc phục lỗi này như thế nào
 - Thêm gói tin báo hiệu mới
 - Cuộc hội thoại của con người
 - » Người gửi không hiểu câu nói “OK” hay “xin nhắc lại” của người nhận
 - » Người gửi sẽ hỏi lại “Bạn đã nói gì vậy” \Leftrightarrow một kiểu gói tin mới được tạo ra ngoài ACK và NAK
 - » Nếu gói “Bạn đã nói gì vậy” cũng bị lỗi => chuyển cái khó này sang cái khác

Các nguyên lý của truyền DL tin cậy

- rdt2.0: Lỗi hỏng-khắc phục
 - 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 (sequence number) vào trong mỗi gói tin. Số thứ tự có độ dài 1 bit.
 - Bên nhận bỏ qua (không nhận) gói bị trùng lặp

Các nguyên lý của truyền DL tin cậy

- rdt2.1

Bên gửi:

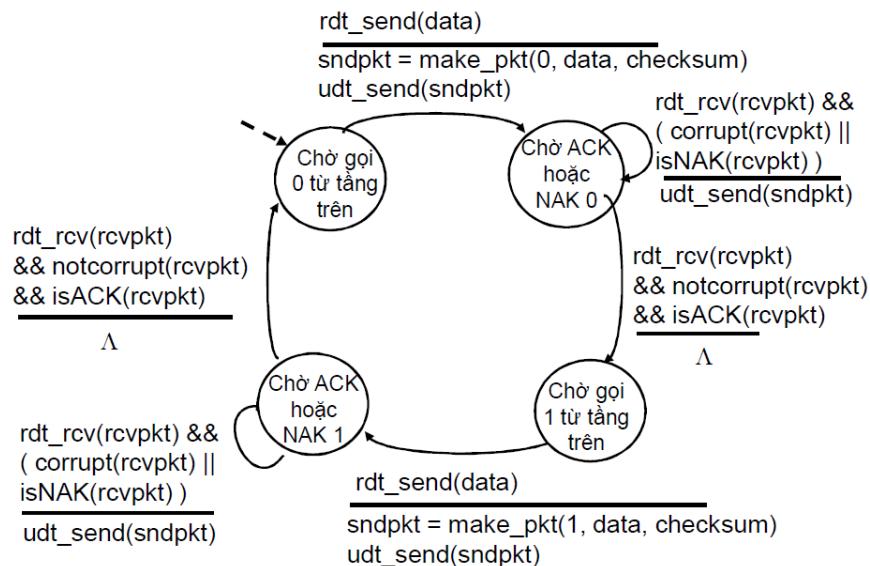
- ❖ Số thứ tự được bổ sung vào gói tin
- ❖ Chỉ cần hai số thứ tự (0,1) là đủ. Vì sao?
- ❖ Phải kiểm tra lại nếu việc nhận ACK/NAK bị hỏng
- ❖ Số trạng thái tăng lên 2 lần
 - Trạng thái phải “nhớ” xem gói tin đang “dự kiến” đến sẽ có số thứ tự là 0 hay 1

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

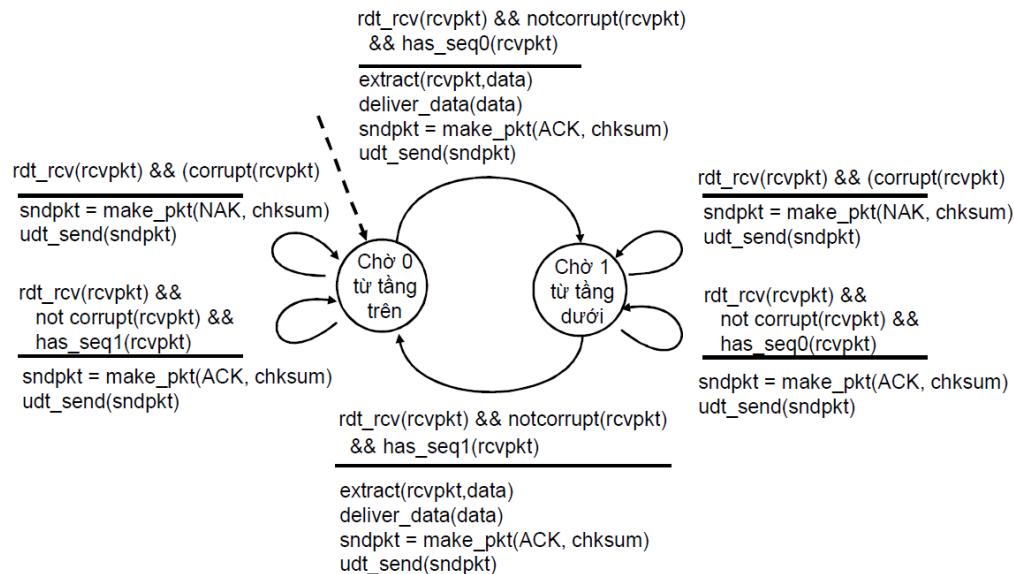
Các nguyên lý của truyền DL tin cậy

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



Bên gửi

Bên nhận

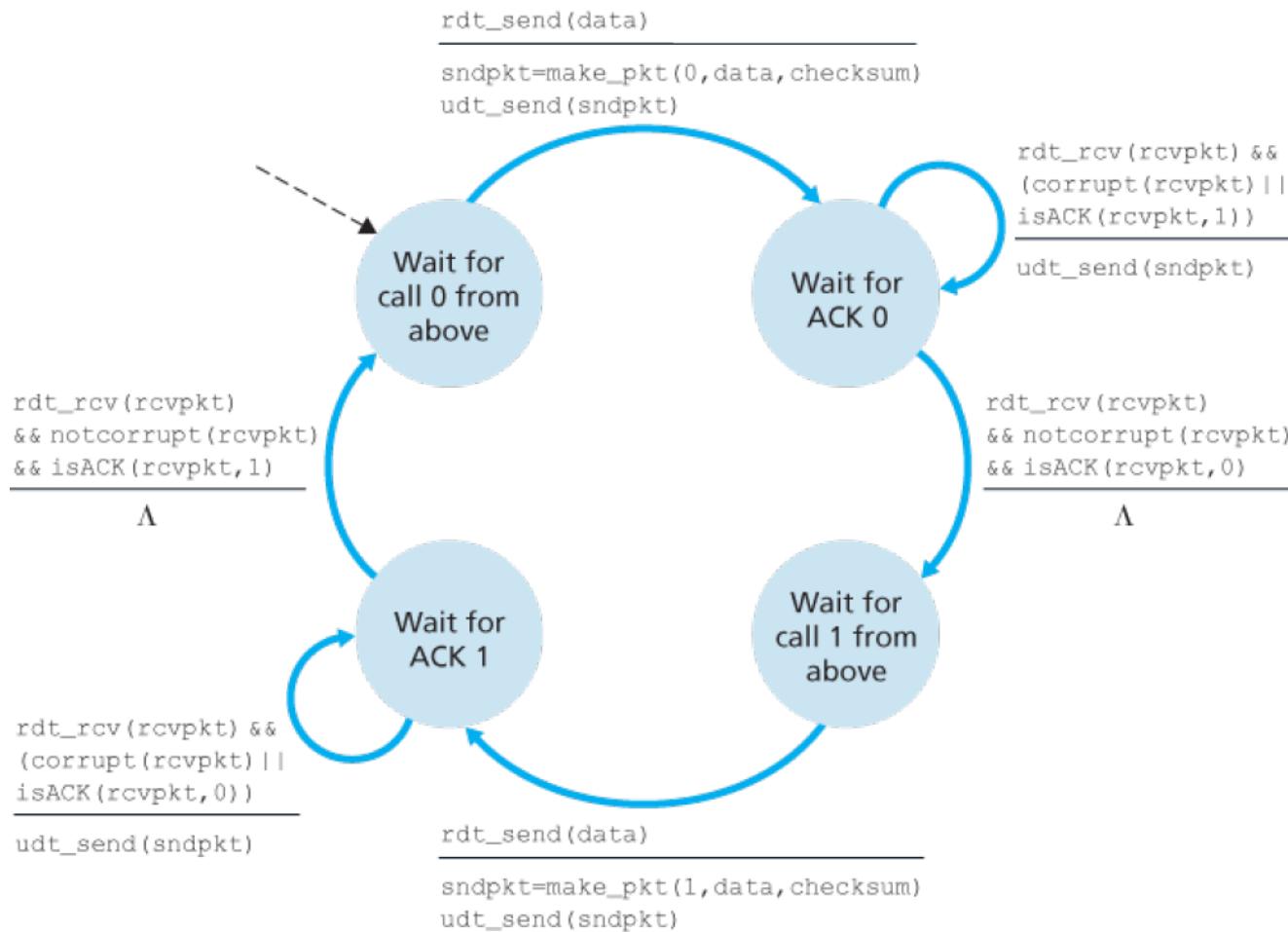


Các nguyên lý của truyền DL tin cậy

- 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

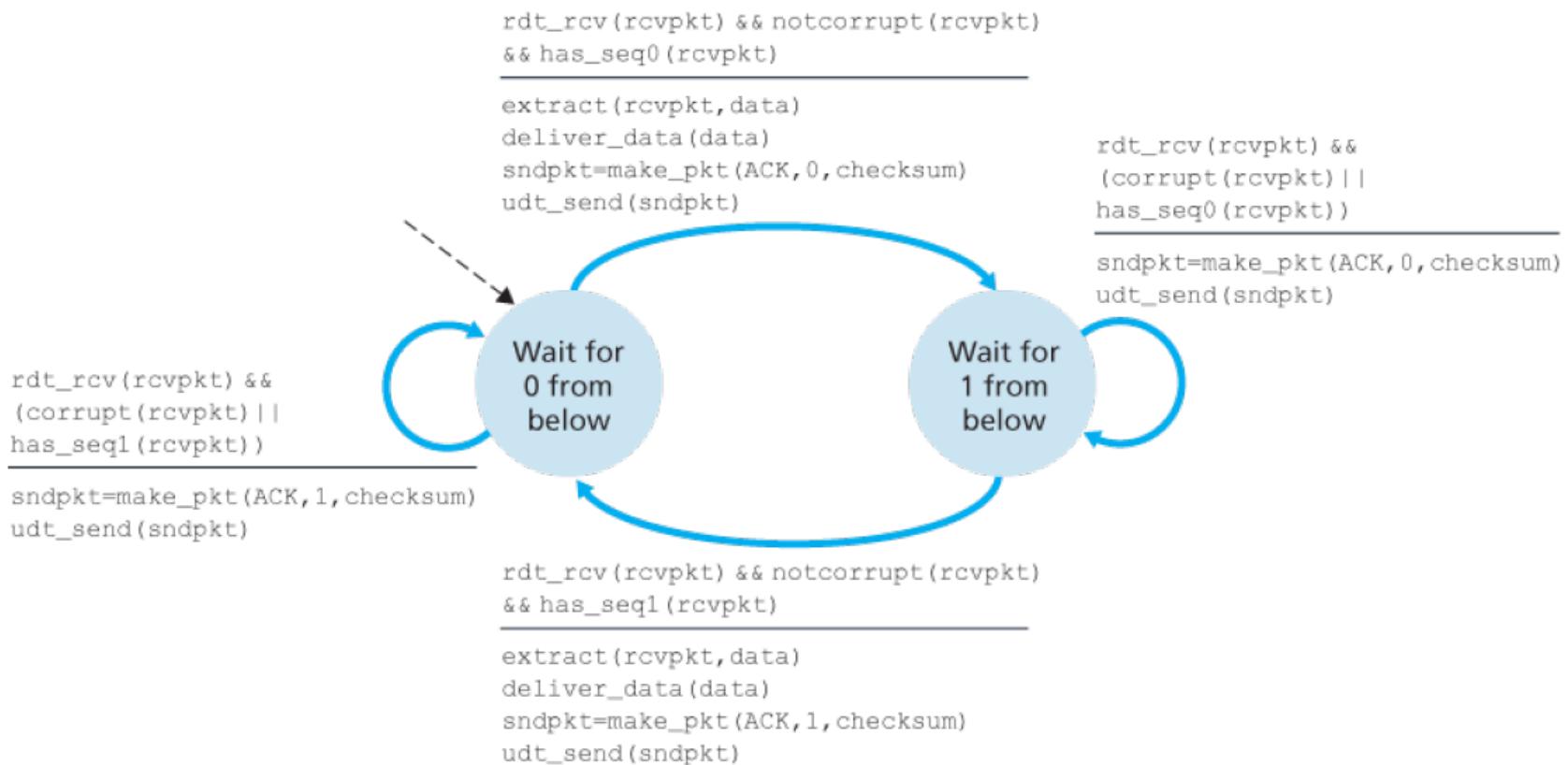
Các nguyên lý của truyền DL tin cậy

- rdt2.2: Bên gửi



Các nguyên lý của truyền DL tin cậy

- rdt2.2: Bên nhận



Các nguyên lý của truyền DL tin cậy

• 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)

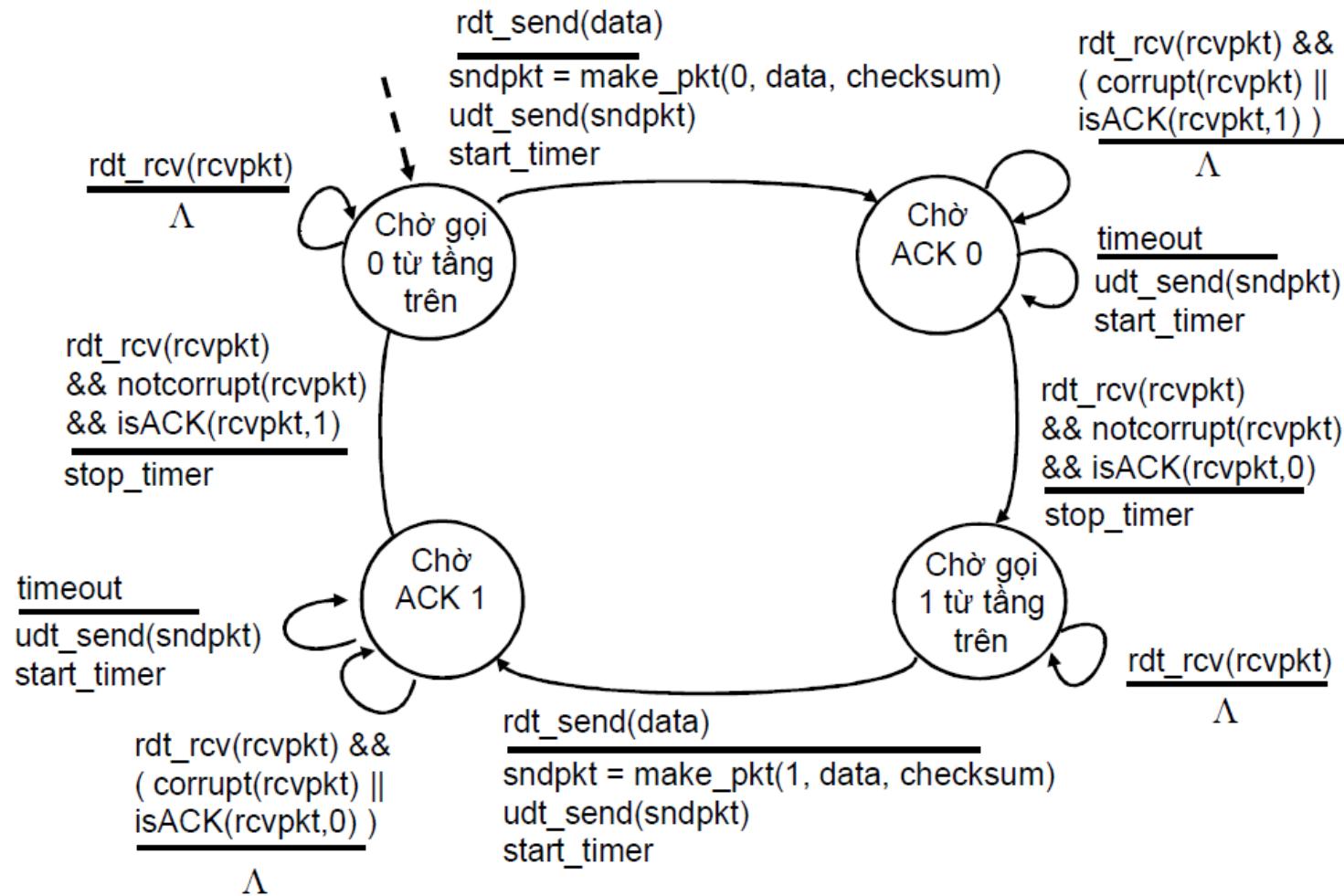
- checksum, số thứ tự, báo nhận ACK, truyền lại sẽ hỗ trợ... nhưng chưa đủ

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

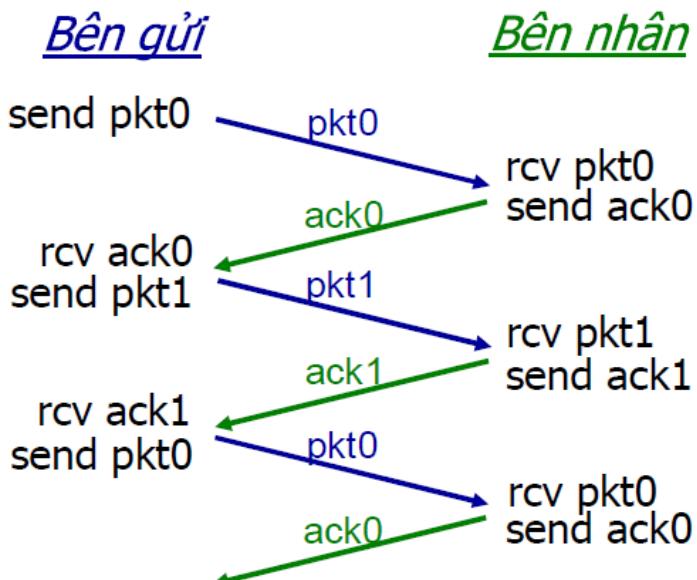
Các nguyên lý của truyền DL tin cậy

- Rdt3.0: Bên gửi

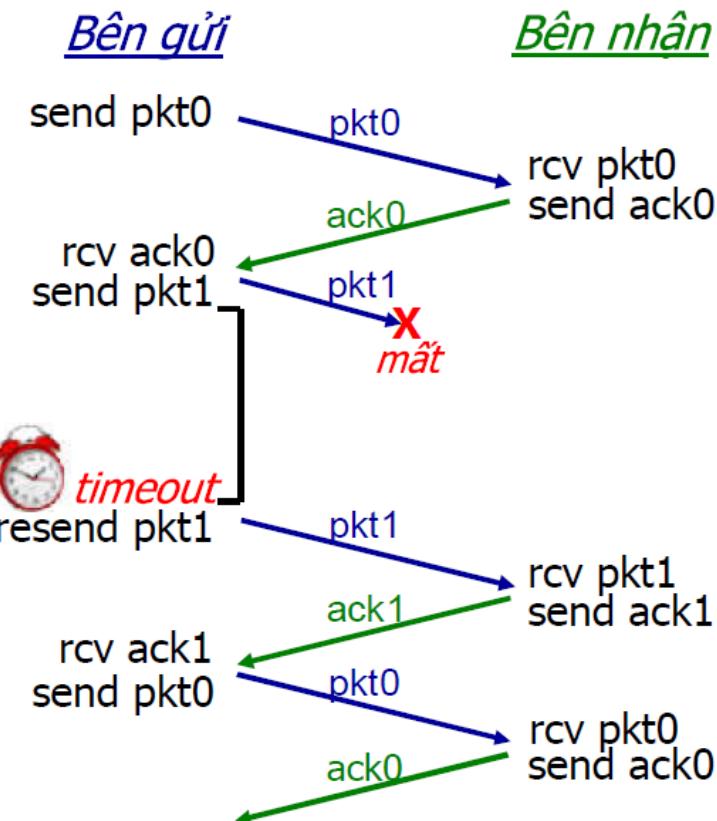


Các nguyên lý của truyền DL tin cậy

- Hoạt động của Rdt3.0



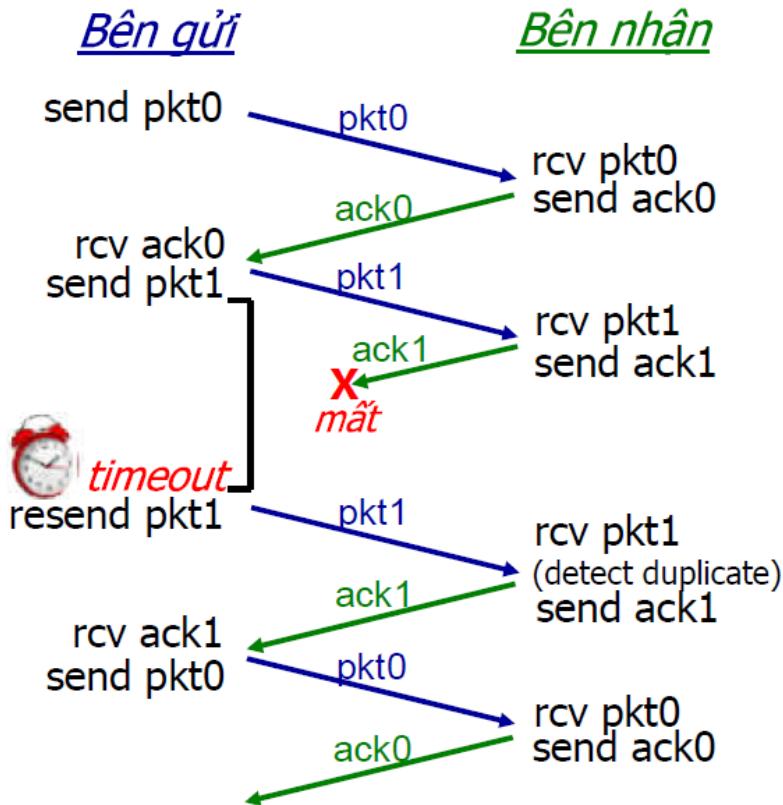
(a) Không mất mát



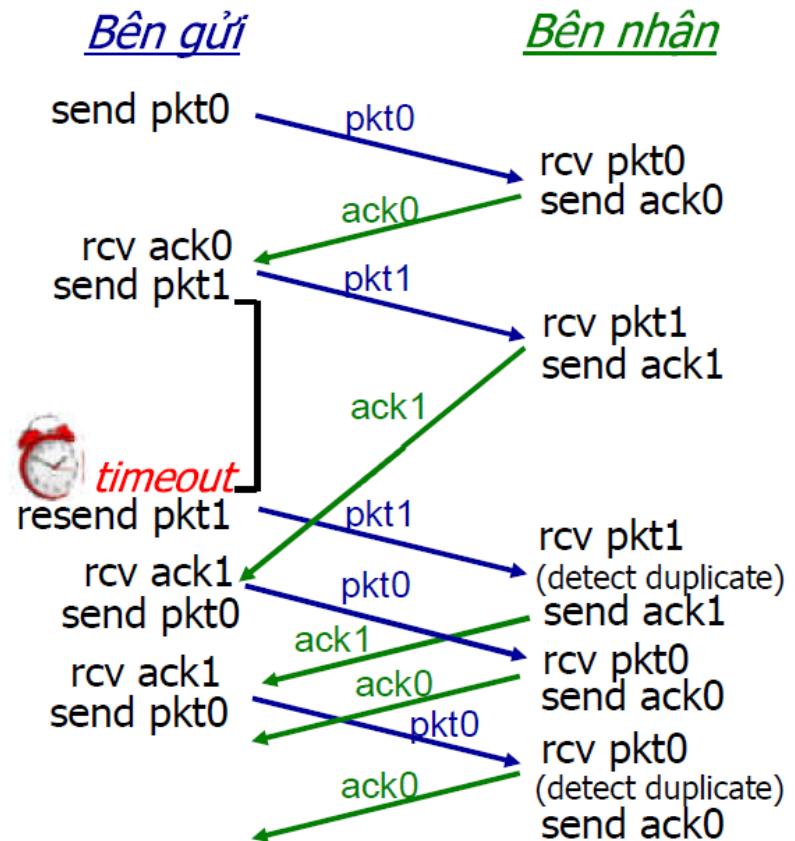
(b) Mất gói tin

Các nguyên lý của truyền DL tin cậy

- Hoạt động của Rdt3.0



(c) Mất ACK



(d) Timeout sớm/ ACK bị trễ

Các nguyên lý của truyền DL tin cậy

- Hiệu suất của Rdt3.0

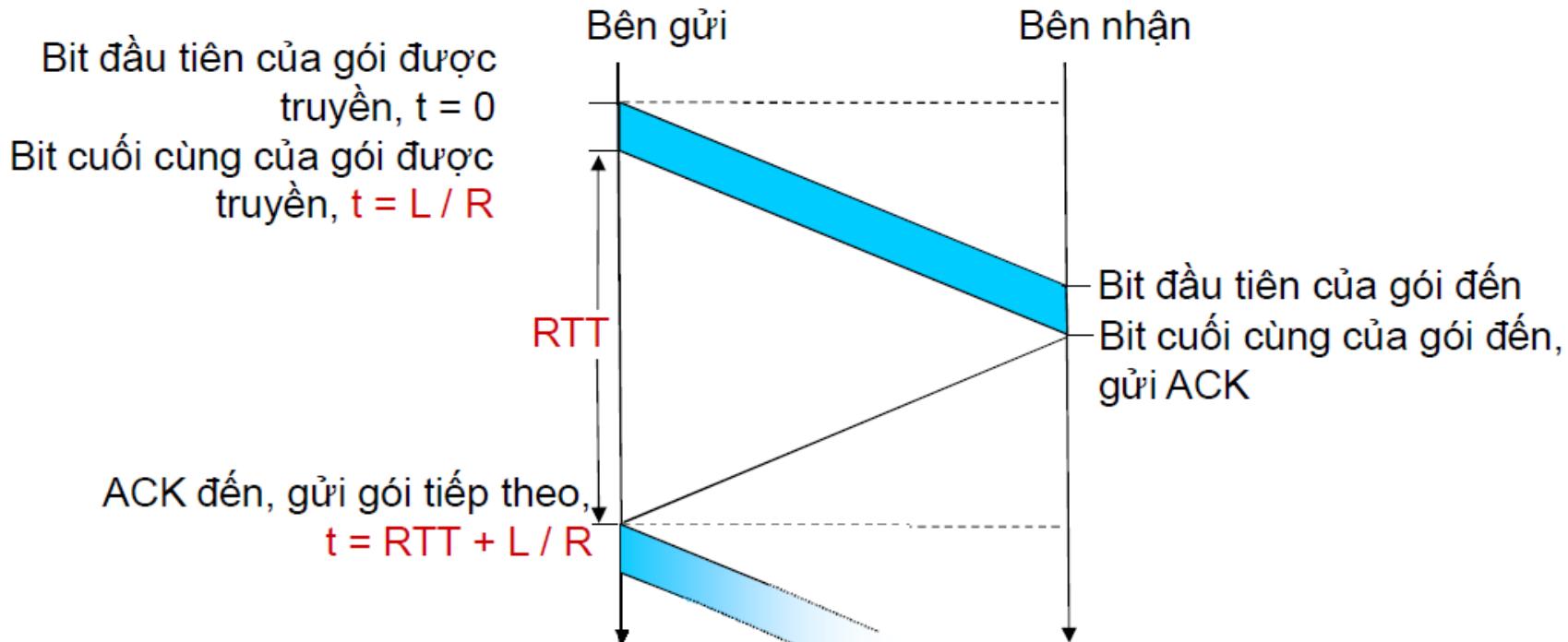
- rdt3.0 hoạt động tốt, nhưng không hiệu quả do cơ chế **stop and wait**
- Ví dụ: Liên kết 1 Gbps, trễ lan truyền 15 ms, gói tin 8000 bit:
 - Thời gian cần để truyền gói tin lên link:

$$D_{trans} = \frac{L}{R} = \frac{8000}{10^9 \text{ bít/sec}} = 8 \text{ microsec} = 0.008 \text{ ms}$$

- Để đến được đích, gói tin mất $15+0,008=15,008$ ms
- Giả sử gói ACK rất nhỏ (bỏ qua thời gian truyền ACK) và bên nhận gửi ACK ngay sau khi nhận được bít dữ liệu cuối cùng của gói tin; Nếu RTT=30 msec. Khi đó ACK xuất hiện tại đầu gửi sau khoảng thời gian: **t=RTT+L/R=30.008** ms

Các nguyên lý của truyền DL tin cậy

- rdt3.0: Hoạt động dừng-và-chờ



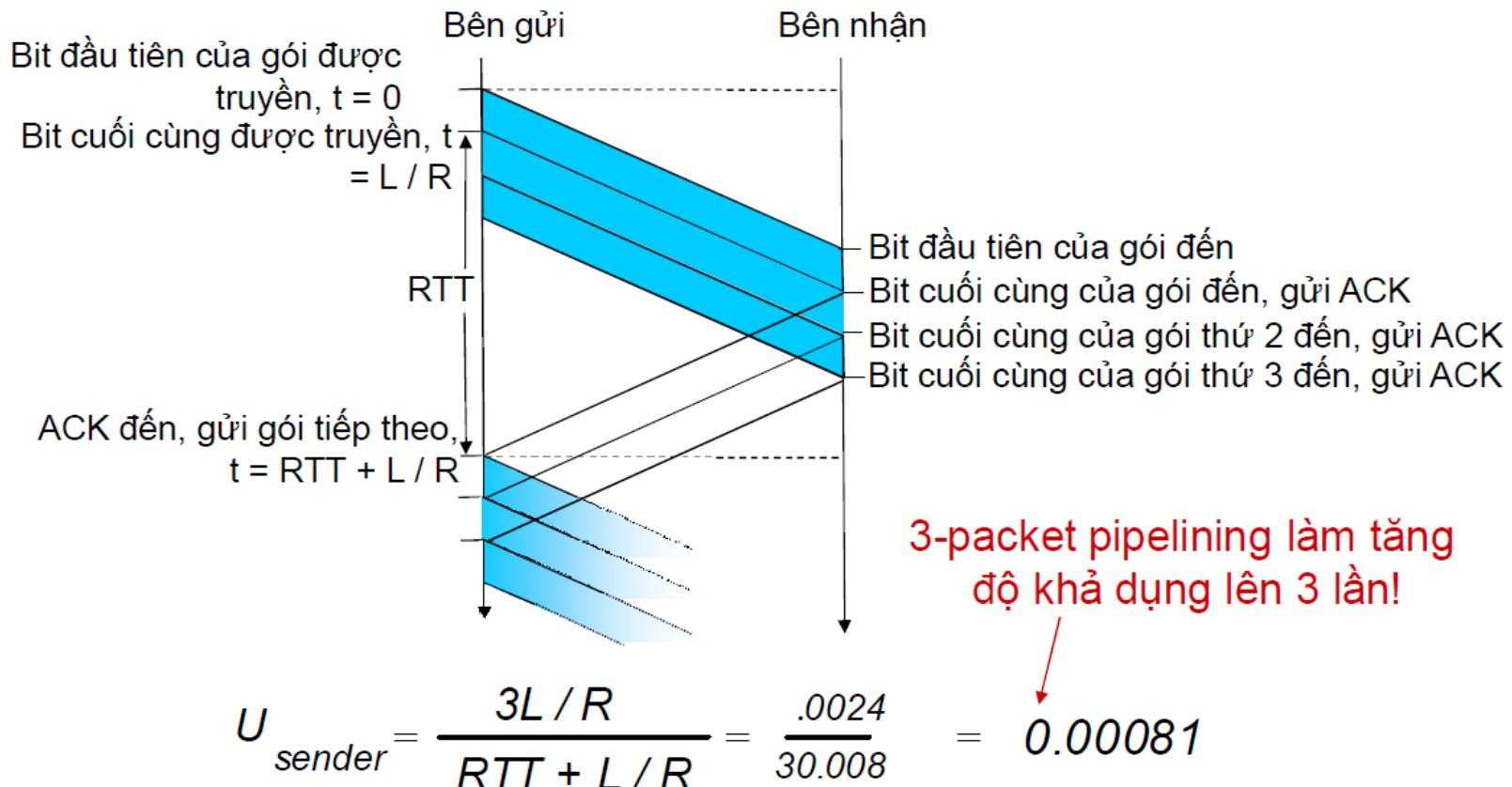
$$U_{\text{sender}} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

Các nguyên lý của truyền DL tin cậy

- Hiệu suất của Rdt3.0
 - 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!

Các nguyên lý của truyền DL tin cậy

- Pipeline: tăng độ khả dụng



Các nguyên lý của truyền DL tin cậy

- Các giao thức pipeline
 - Pipelining: bên gửi sẽ gửi nhiều gói tin “đồng thời”, mà không cần chờ nhận được gói ACK từ bên nhận
 - Dãy các số thứ tự sẽ được tăng dần
 - Cần có bộ đệm tại bên gửi và/hoặc bên nhận
 - Hai dạng thức chung của các giao thức pipeline
 - go-Back-N
 - Lặp có lựa chọn (selective repeat)

Các nguyên lý của truyền DL tin cậy

- Các giao thức Pipeline

Go-back-N:

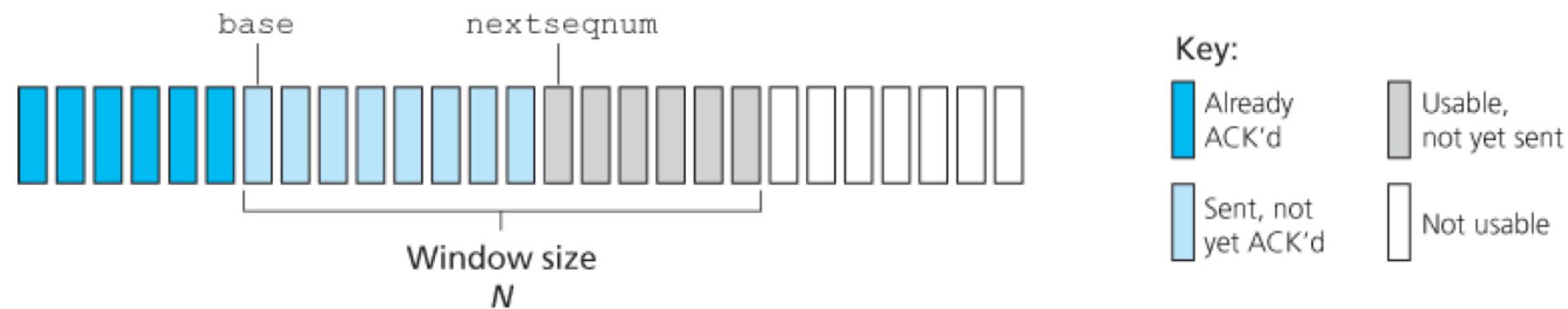
- Bên gửi có thể có đến N gói chưa được báo nhận trong pipeline
- Bên nhận chỉ gửi *ack tích lũy*
 - Không báo nhận cho gói tin cho đến khi có một khoảng trống
- Bên gửi có bộ định thời cho các gói tin gửi đi mà chưa được báo nhận
 - Khi bộ định thời hết hạn, **truyền lại tất cả các gói tin chưa được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

Lặp có lựa chọn:

- 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**

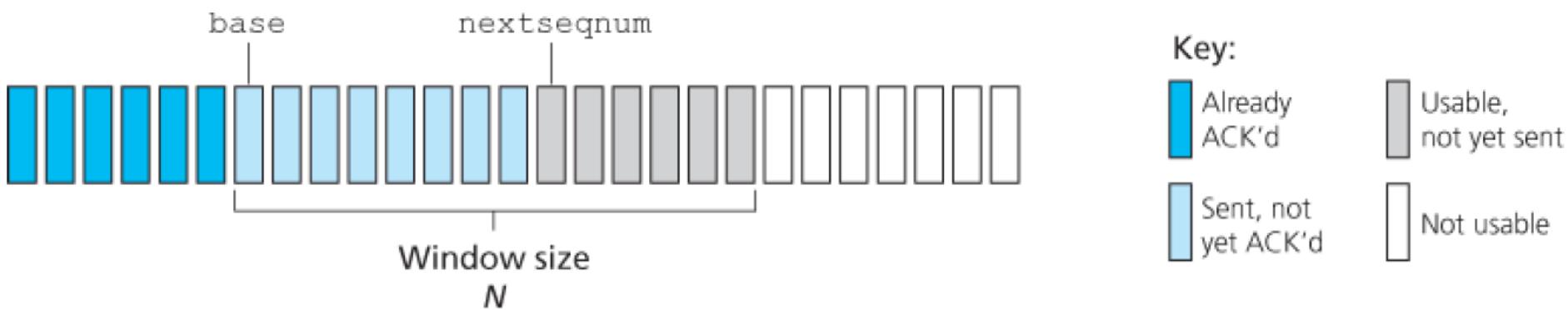
Các nguyên lý của truyền DL tin cậy

- Go-Back-N: bên gửi
 - base: số thứ tự của gói lâu nhất chưa được ACK
 - Nextseqnum: số thứ tự của gói tin tiếp theo sẽ được gửi



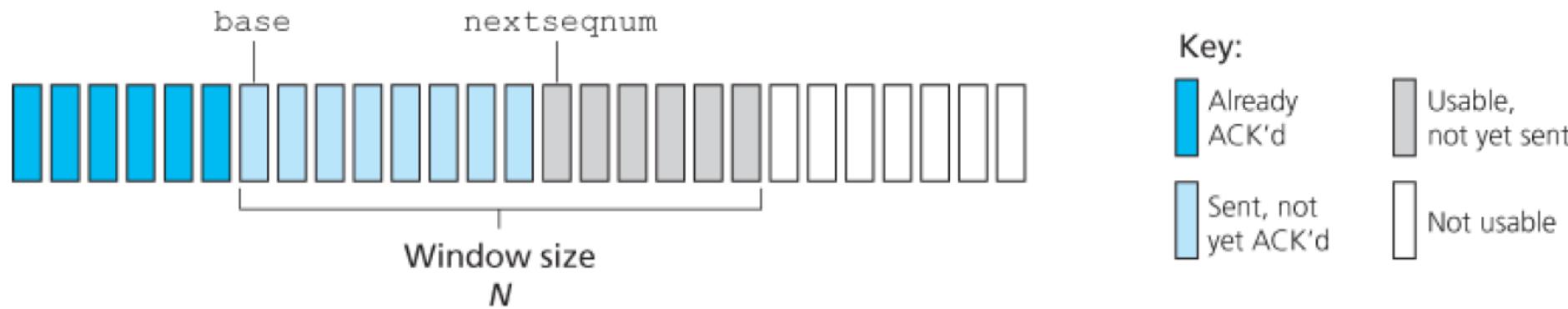
Các nguyên lý của truyền DL tin cậy

- Go-Back-N: bên gửi
 - Các số thứ tự nằm trong khoảng $[0, \text{base}-1]$ là các gói đã được truyền đi và đã được ACK
 - Các số thứ tự nằm trong khoảng $[\text{base}, \text{nextseqnum}-1]$ sẽ là các gói đã gửi nhưng chưa được ACK.



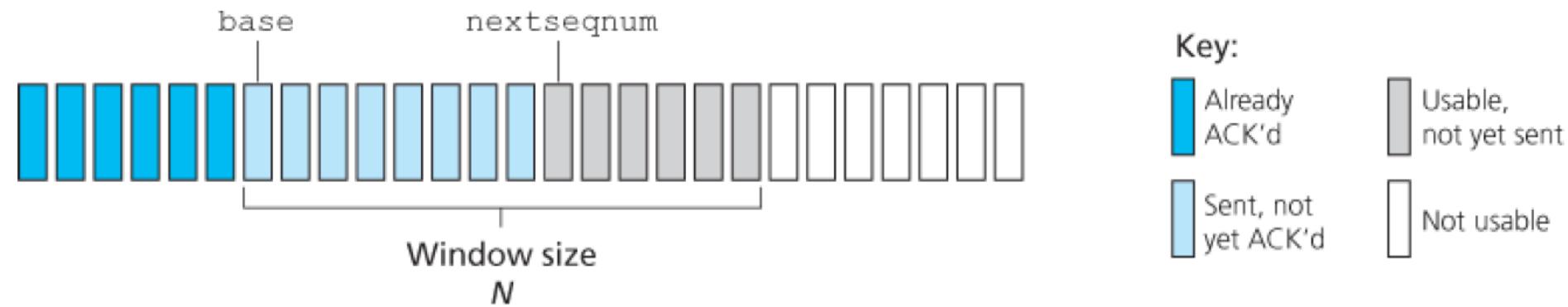
Các nguyên lý của truyền DL tin cậy

- Go-Back-N: bên gửi
 - Các số thứ tự nằm trong khoảng $[nextseqnum, base+N-1]$ dùng cho các gói tin có thể gửi ngay
 - Các số thứ tự $\geq (base+N)$ không thể sử dụng khi gói NAK hiện đang trong pipeline (cụ thể, gói với số thứ tự = base đã được ACK)



Các nguyên lý của truyền DL tin cậy

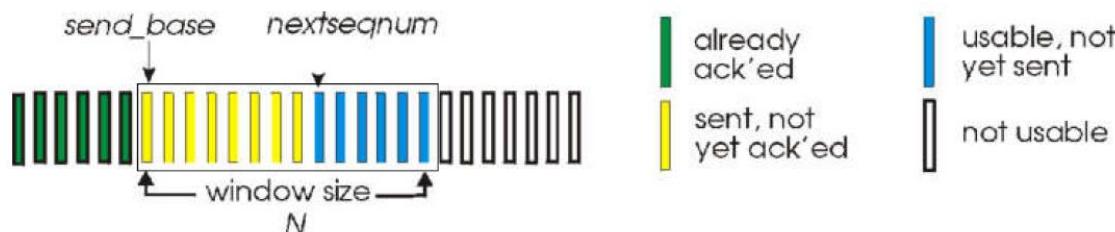
- Go-Back-N: bên gửi
 - Cửa sổ có kích cỡ N: gồm các số thứ tự được phép cho các gói truyền nhưng chưa có ACK
 - N = window size, giao thức GBN là giao thức cửa sổ trượt.



Các nguyên lý của truyền DL tin cậy

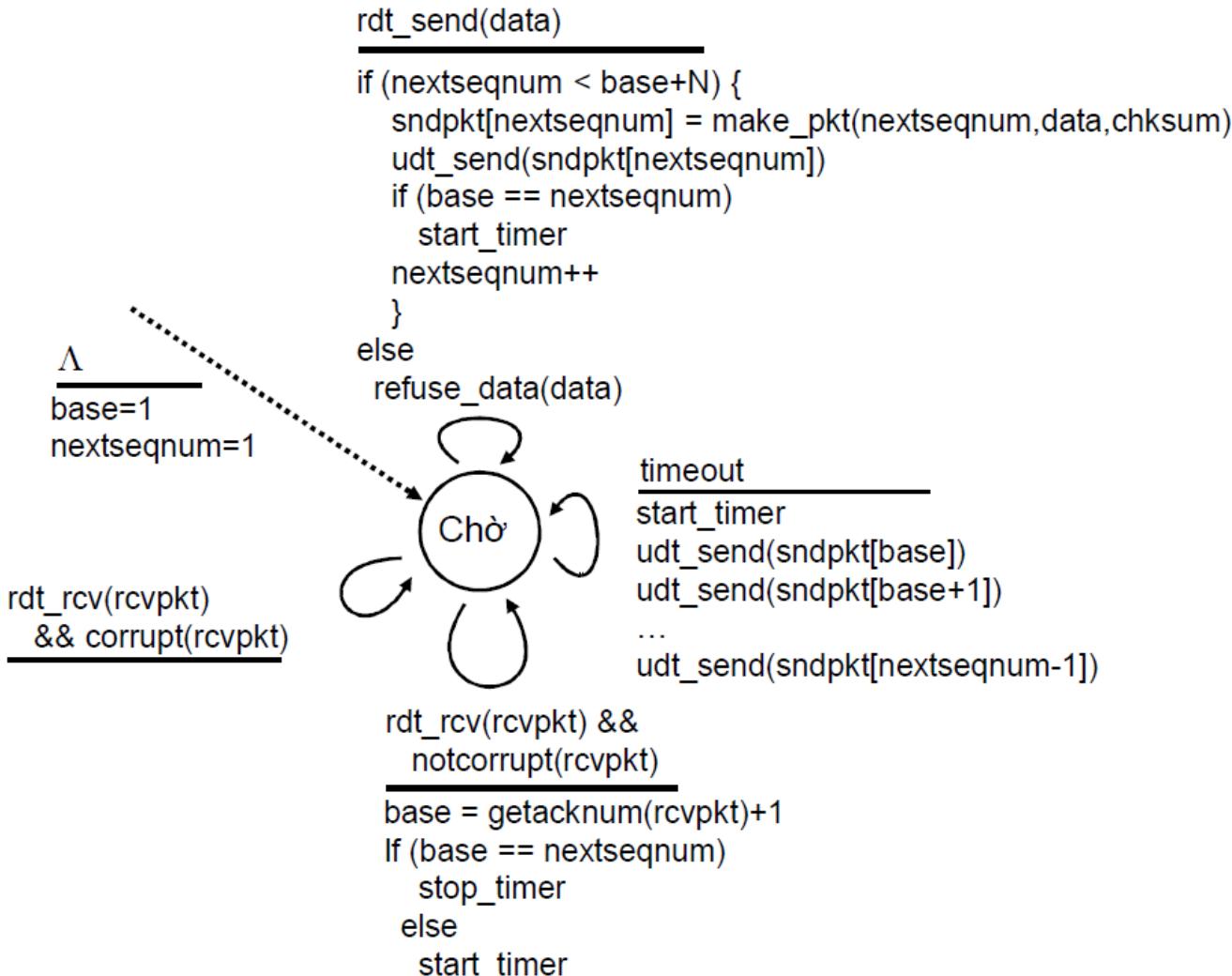
- Go-Back-N: bên gửi

- k-bit trong trường số thứ tự ở phần tiêu đề của gói tin
 - Phạm vi các số thứ tự là $[0, 2k-1]$
- “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



Các nguyên lý của truyền DL tin cậy

- GBN: FSM mở rộng tại bên gửi



Các nguyên lý của truyền DL tin cậy

- GBN: FSM mở rộng tại bên gửi
 - Phải phản hồi 3 kiểu sự kiện
 - Lời mời từ tầng trên
 - Khi ***rdt_send ()*** được gọi từ tầng trên, trước tiên người gửi sẽ kiểm tra xem cửa sổ đã đầy chưa, tức là có N gói chưa ACK hay không.
 - Nếu cửa sổ không đầy, một gói tin được tạo và gửi đi, và các biến được cập nhật tương ứng.
 - Nếu cửa sổ đã đầy, người gửi chỉ cần trả lại dữ liệu trả lại tầng trên (một dấu hiệu ngầm rằng cửa sổ đã đầy). Tầng trên có lẽ sẽ phải thử lại sau.
 - Trong triển khai thực tế, người gửi có nhiều khả năng đã lưu vào bộ đệm (nhưng không được gửi ngay lập tức) dữ liệu này hoặc sẽ có cơ chế đồng bộ hóa (ví dụ, một semaphore hoặc một cờ) sẽ cho phép tầng trên chỉ gọi ***rdt_send ()*** khi cửa sổ chưa đầy.

Các nguyên lý của truyền DL tin cậy

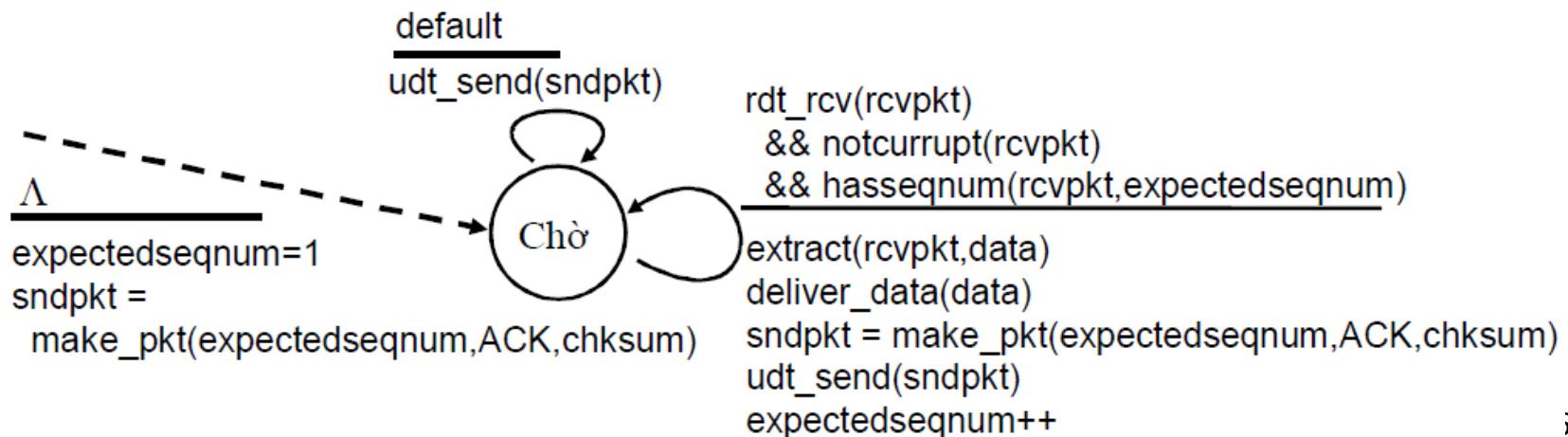
- GBN: FSM mở rộng tại bên gửi
 - Phải phản hồi 3 kiểu sự kiện
 - Nhận ACK
 - Trong giao thức GBN, một ACK cho một gói tin có số thứ tự n sẽ được coi là một ACK tích lũy, có nghĩa là tất cả các gói với một số thứ tự tăng dần tới n đã được nhận ở đầu nhận

Các nguyên lý của truyền DL tin cậy

- GBN: FSM mở rộng tại bên gửi
 - Phải phản hồi 3 kiểu sự kiện
 - Sự kiện hết thời gian
 - Tên của giao thức, “Go-Back-N”, bắt nguồn từ hành vi của người gửi trong sự hiện diện của các gói bị mất hoặc quá trễ. Như trong giao thức dừng và chờ, bộ đếm thời gian sẽ được sử dụng để khôi phục dữ liệu hoặc các gói ACK bị mất.
 - Nếu hết thời gian chờ xảy ra, người gửi sẽ gửi lại tất cả các gói đã được gửi trước đó nhưng chưa có ACK.
 - Người gửi chỉ sử dụng một bộ đếm thời gian duy nhất, có thể được coi là bộ định thời cho các gói tin được truyền lâu nhất và chưa có ACK.
 - Nếu nhận được ACK nhưng vẫn có thêm các gói tin được truyền và chưa có ACK, bộ đếm thời gian được khởi động lại.
 - Nếu không gói tin nào chưa có ACK, bộ đếm thời gian bị dừng.

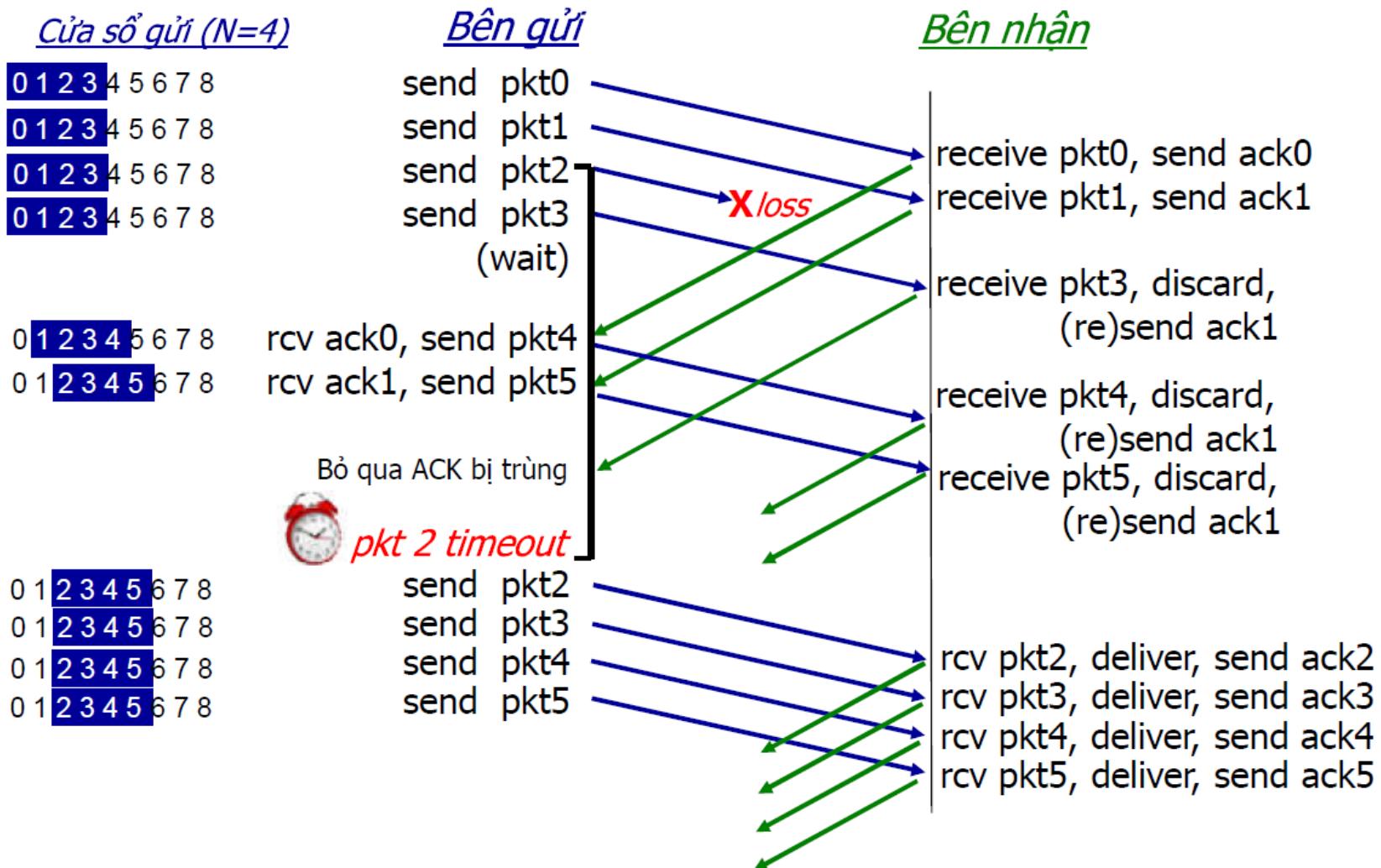
Các nguyên lý của truyền DL tin cậy

- 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



Các nguyên lý của truyền DL tin cậy

• Hoạt động của GBN



Các nguyên lý của truyền DL tin cậy

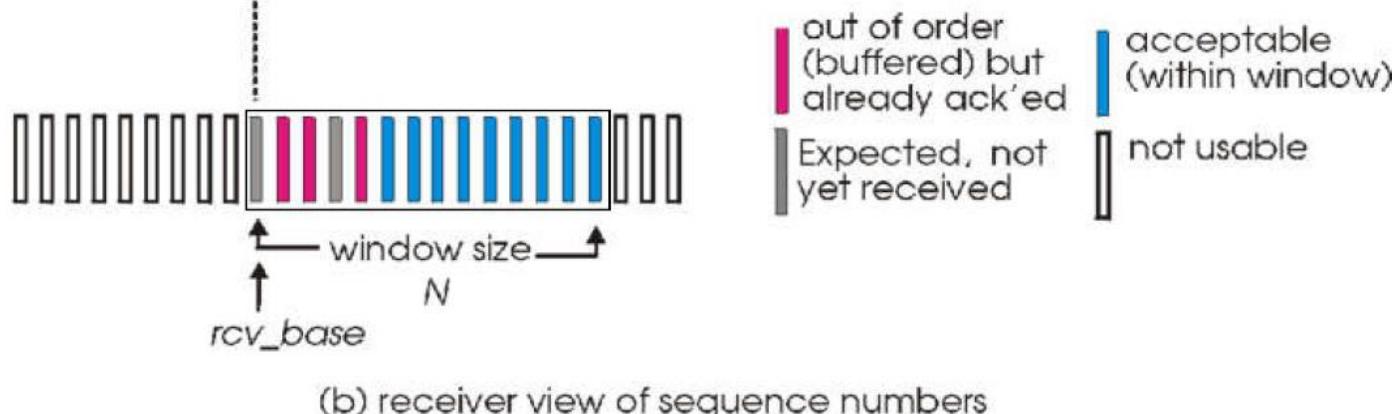
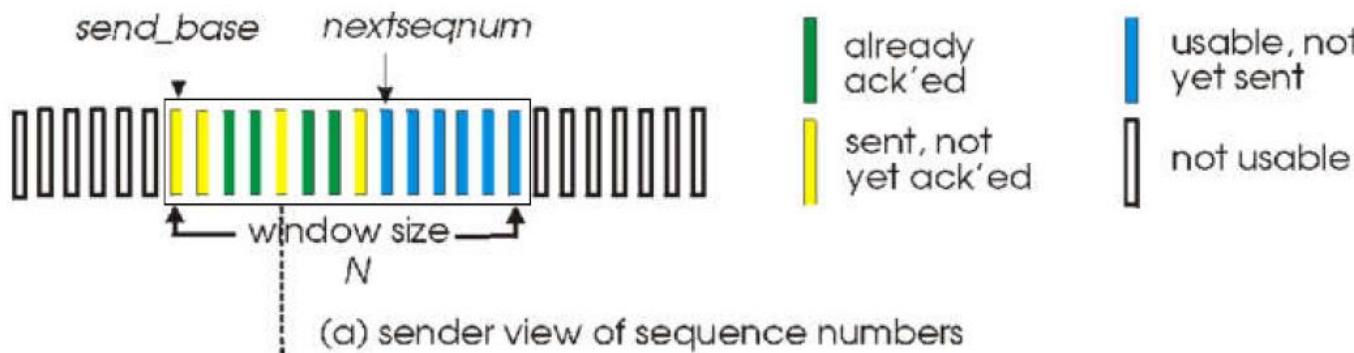
- Lắp có lựa chọn
 - Giao thức GBN
 - Cho phép người gửi có khả năng “lắp đầy đường ống” với các gói => khắc phục vấn đề sử dụng kênh đã lưu ý với các giao thức dừng và chờ.
 - Gặp sự cố về hiệu suất. Đặc biệt, khi kích thước cửa sổ và độ trễ băng thông lớn, nhiều gói có thể nằm trong đường ống.
 - Một lỗi lõi có thể khiến GBN truyền lại một số lượng lớn các gói, nhiều gói không cần thiết => khi xác suất lỗi kênh tăng lên, đường ống có thể bị lắp đầy bởi những lần truyền lại không cần thiết.

Các nguyên lý của truyền DL tin cậy

- Lắp 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

Các nguyên lý của truyền DL tin cậy

- Lắp có lựa chọn: cửa sổ bên gửi, bên nhận



Các nguyên lý của truyền DL tin cậy

- Lắp có lựa chọn
 - Các sự kiện và hoạt động của bên gửi
 - Dữ liệu nhận được từ tầng trên
 - Khi nhận được data từ tầng trên, bên gửi kiểm tra số thứ tự tiếp theo sẵn có cho gói tin. Nếu số thứ tự nằm trong cửa sổ của bên gửi, data sẽ được đóng gói và gửi đi, nếu không nó sẽ không được lưu vào bộ đệm hoặc chuyển trả lại cho tầng trên cho lần truyền sau (như trong GBN)
 - Bộ đếm thời gian
 - Xử lý với trường hợp gói tin bị mất. Tuy nhiên, mỗi gói tin phải có bộ đếm thời gian logic của chính nó, bởi vì chỉ có duy nhất một gói tin sẽ được truyền vào thời gian chờ. Có thể dùng một timer hardware để bắt chước hoạt động của nhiều bộ đếm logic.

Các nguyên lý của truyền DL tin cậy

- Lắp có lựa chọn
 - Các sự kiện và hoạt động của bên gửi
 - Nhận ACK
 - Khi nhận được một ACK, bên gửi sẽ đánh dấu gói tin này đã được nhận, miễn là nó ở trong cửa sổ. Nếu như số thứ tự = base, base cửa sổ sẽ được dịch về phía trước cho gói NAK với số thứ tự nhỏ nhất. Nếu cửa sổ di chuyển và có các gói chưa được truyền với các số thứ tự nằm trong cửa sổ, thì các gói này sẽ được truyền

Các nguyên lý của truyền DL tin cậy

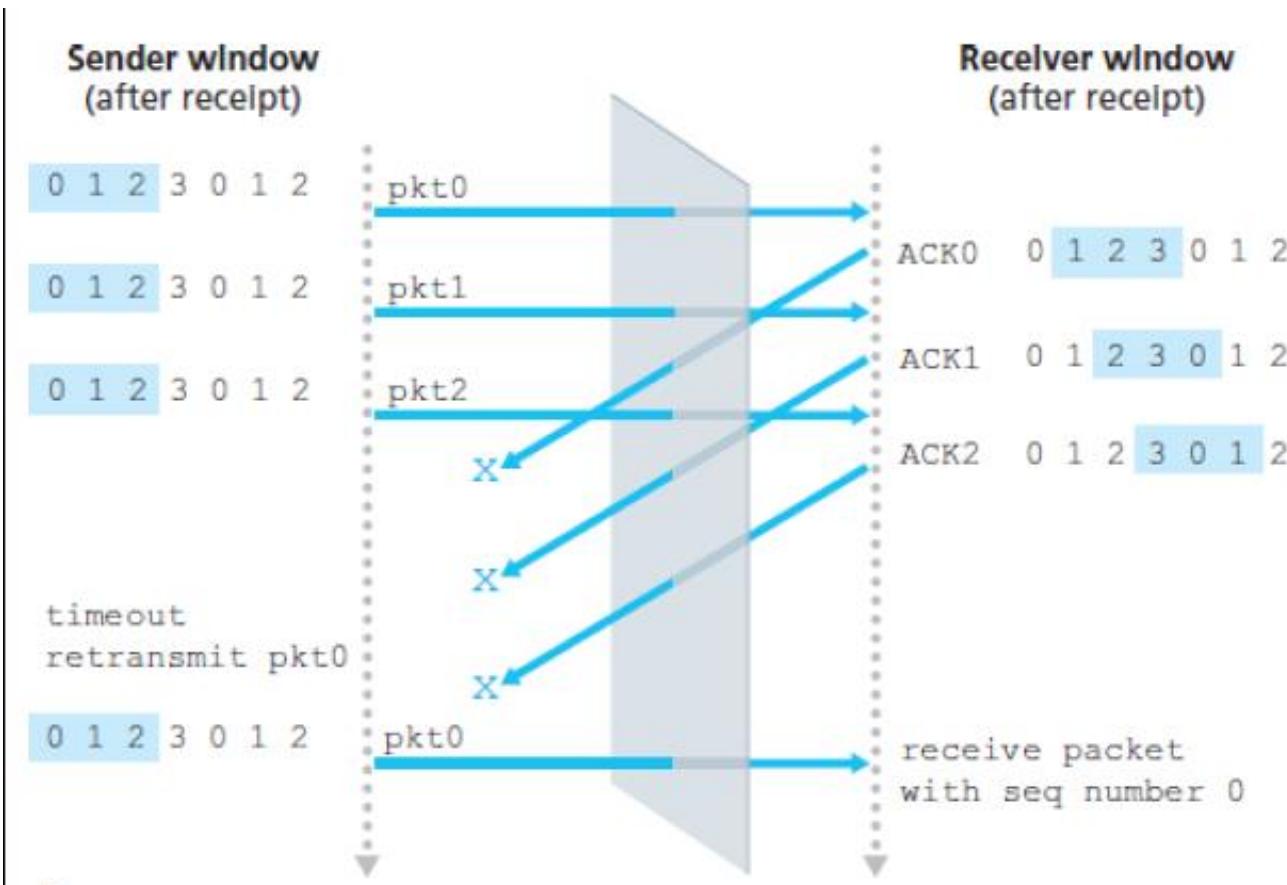
- Lắp có lựa chọn
 - Các sự kiện và hoạt động của bên nhận
 - Gói tin có số thứ tự trong khoảng [rcvbase, rcvbase+N-1] được nhận đúng
 - Trong trường hợp này, gói nhận được sẽ nằm trong cửa sổ của bên gửi và một gói ACK được trả về cho bên gửi.
 - Nếu như gói tin này không được nhận trước đó, nó sẽ được lưu vào bộ nhớ đệm
 - Nếu gói tin này có số thứ tự = base của cửa sổ nhận, khi đó gói tin này và bất kỳ gói tin nào được lưu trước đó trong bộ đệm và các gói tin được đánh số liên tiếp (bắt đầu bằng rcv_base) được truyền tới tầng cao hơn.
 - Cửa sổ trượt lên phía trước theo số gói tin chuyển tới tầng cao hơn

Các nguyên lý của truyền DL tin cậy

- Lắp có lựa chọn
 - Các sự kiện và hoạt động của bên nhận
 - Gói tin với số thứ tự trong khoảng [rcvbase-N, rcvbase-1] được nhận đúng
 - Trong trường hợp này, phải tạo lại ACK cho gói tin đã được bên nhận trước đó xác nhận ACK
 - Ngoài 2 trường hợp trên
 - Bỏ qua gói tin

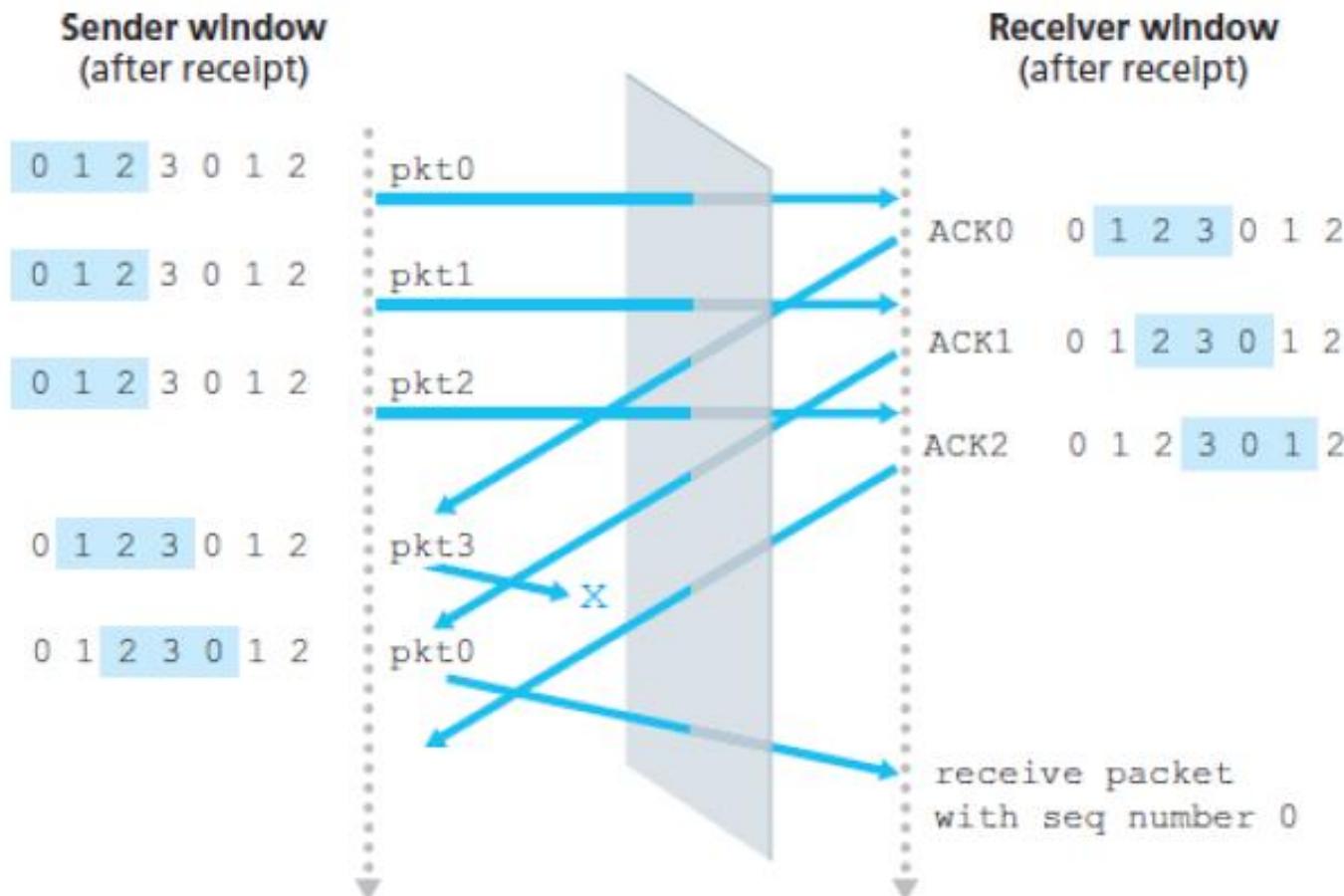
Các nguyên lý của truyền DL tin cậy

- Hoạt động trong lặp có lựa chọn
 - 2 kịch bản: Kịch bản 1



Các nguyên lý của truyền DL tin cậy

- Hoạt động trong lặp có lựa chọn
 - 2 kịch bản: Kịch bản 2

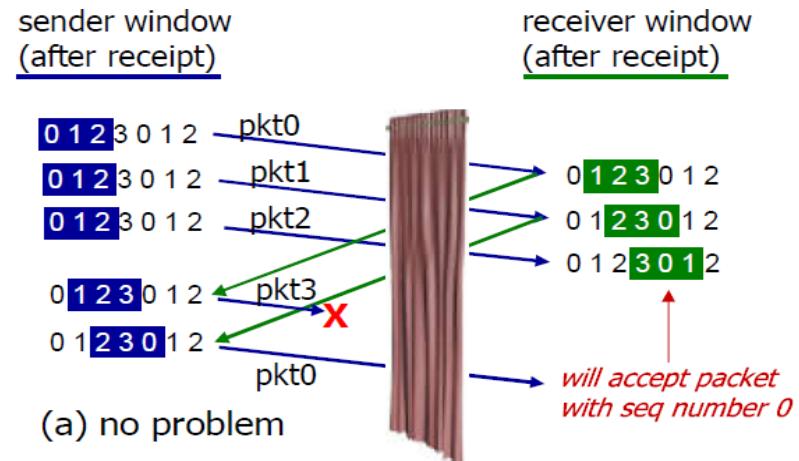


Các nguyên lý của truyền DL tin cậy

- Lắp có lựa chọn: tình trạng khó giải quyết

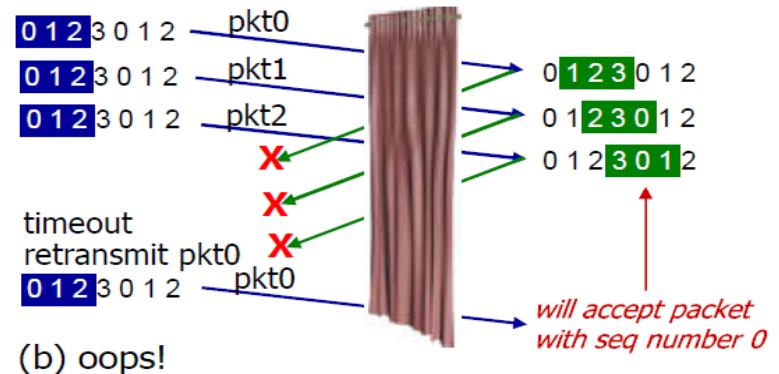
Ví dụ:

- 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 lắp như là dữ liệu mới trong kịch bản 2
- Hỏi:** Quan hệ giữa phạm vi số thứ tự và kích thước cửa sổ như thế nào để tránh vấn đề như trong kịch bản 2?



(a) no problem

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



(b) oops!

Các nguyên lý của truyền DL tin cậy

- Tóm tắt các cơ chế truyền dữ liệu tin cậy
 - Checksum
 - Dùng để phát hiện các lỗi bit trong một gói tin truyền
 - Timer
 - Định thời/truyền lại gói tin có thể do gói tin hay ACK của nó bị mất trong quá trình truyền trên kênh. Do timeouts có thể xuất hiện khi một gói tin bị trễ nhưng không mất, hay khi gói tin nhận được bởi đầu nhận nhưng ACK giữa bên gửi-nhận bị mất, duplicate copies của gói tin có thể được nhận bởi đầu nhận

Các nguyên lý của truyền DL tin cậy

- Tóm tắt các cơ chế truyền dữ liệu tin cậy
 - Sequence number
 - Sử dụng cho việc đánh số thứ tự các gói tin của dữ liệu truyền từ bên gửi sang bên nhận.
 - Khoảng cách giữa các số thứ tự của các gói nhận được cho phép bên nhận phát hiện ra gói tin bị mất.
 - Các gói có 2 số thứ tự trùng nhau cho phép bên nhận phát hiện các bản copy trùng nhau của một gói tin.

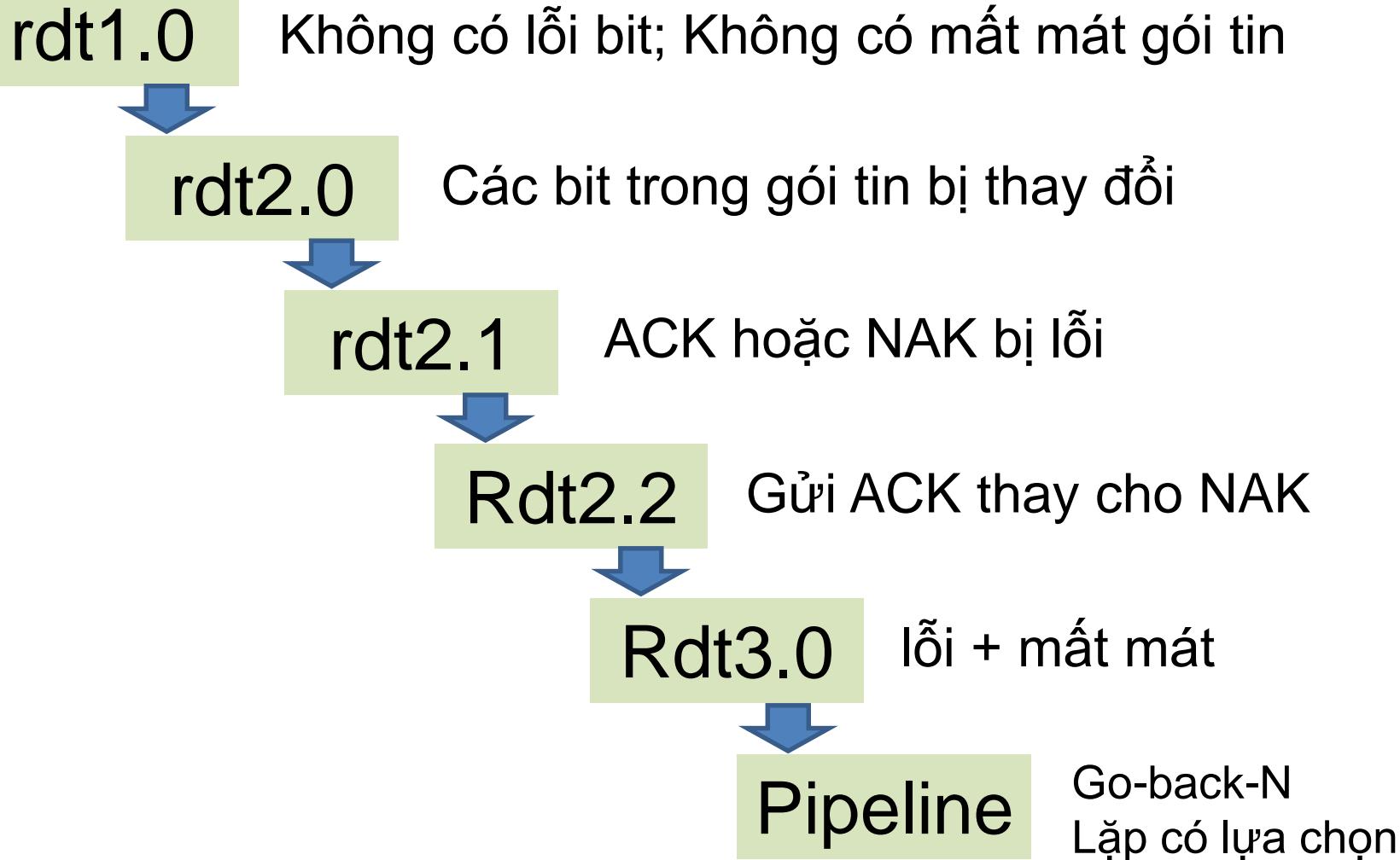
Các nguyên lý của truyền DL tin cậy

- Tóm tắt các cơ chế truyền dữ liệu tin cậy
 - Acknowledgment - ACK
 - Được bên nhận sử dụng để nói với bên gửi rằng gói tin hay một tập gói tin được nhận đúng.
 - Các ACK sẽ mang số thứ tự của gói tin hay các gói tin được ACK.
 - Các ACK có thể là riêng biệt hoặc tích tủy, phụ thuộc vào giao thức
 - Negative Acknowledgment – NAK
 - Được bên nhận sử dụng để nói với bên gửi rằng một gói tin chưa được nhận đúng.
 - NAK sẽ mang số thứ tự của gói tin chưa được nhận đúng.

Các nguyên lý của truyền DL tin cậy

- Tóm tắt các cơ chế truyền dữ liệu tin cậy
 - Window, pipelining
 - Bên gửi có thể bị hạn chế chỉ gửi các gói tin với các số thứ tự nằm trong phạm vi nào đó. Bằng cách cho phép nhiều gói tin được truyền mà chưa được xác nhận ACK, bằng thông bên gửi sử dụng có thể tăng hơn so với chế độ stop-and-wait
 - Kích cỡ cửa sổ có thể được thiết lập dựa trên cơ sở khả năng của bên nhận và các gói tin lưu trong bộ đệm, hoặc mức độ tắc nghẽn trong mạng hoặc cả hai.

Review: Các nguyên lý của truyền DL tin cậy



Nội dung

- Các dịch vụ tầng giao vận
- Ghép kênh và phân kênh
- Vận chuyển không kết nối: UDP
- Các nguyên lý truyền dữ liệu tin cậy
- **Vận chuyển hướng kết nối: TCP**
- Các nguyên lý điều khiển tắc nghẽn
- Điều khiển tắc nghẽn TCP

Vận chuyển hướng kết nối: 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

Vận chuyển hướng kết nối: TCP

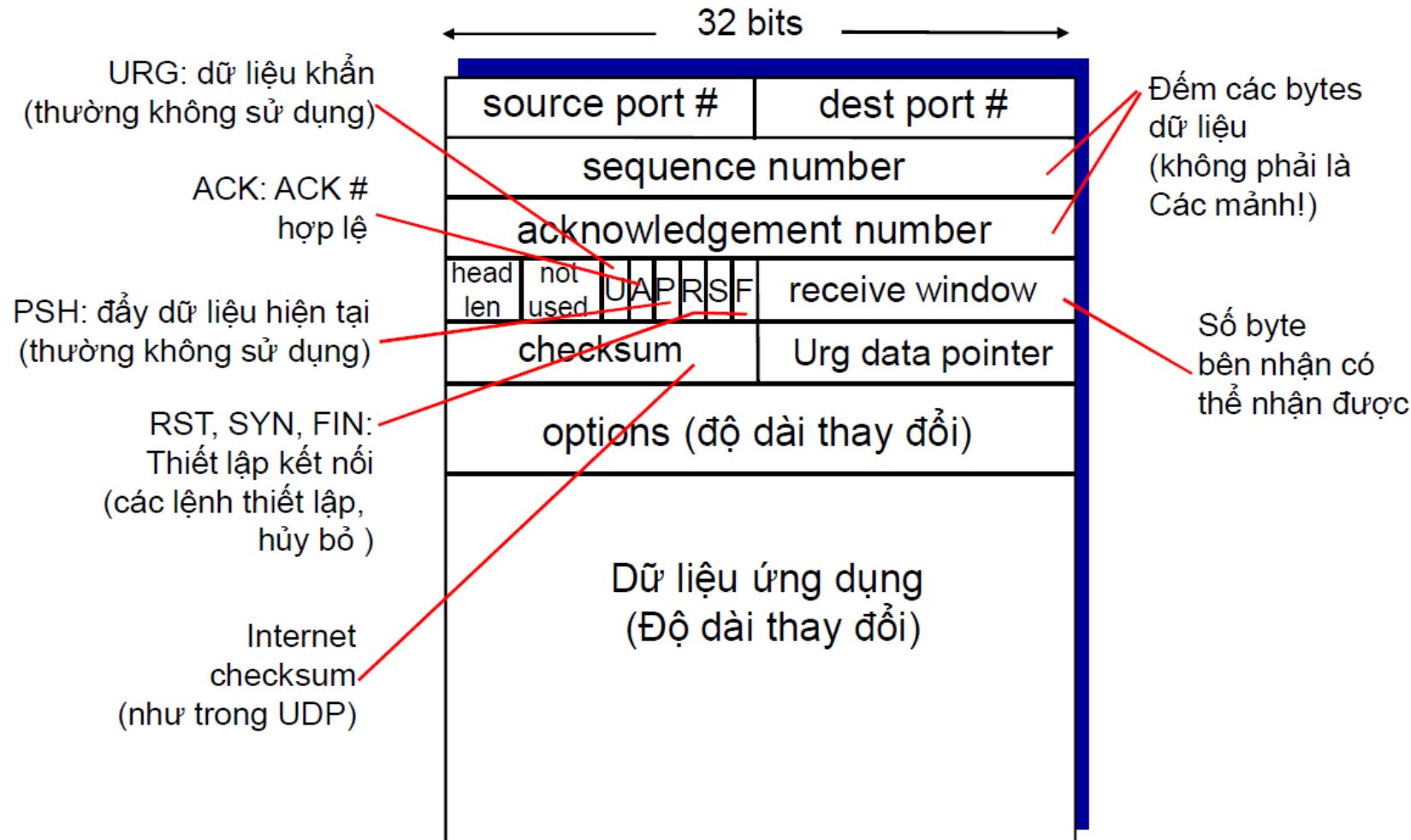
- Khái quát TCP
 - Truyền dữ liệu song công (full duplex)
 - Luồng dữ liệu đi theo 2 hướng trên cùng một kết nối
 - MSS: maximum segment size
 - Hướng kết nối:
 - Bắt tay (trao đổi các thông điệp điều khiển) khởi tạo trạng thái cho bên gửi và bên nhận trước khi trao đổi dữ liệu
 - Điều khiển luồng:
 - Bên gửi không lấn át bên nhận

Vận chuyển hướng kết nối: TCP

- Cấu trúc TCP segment
 - Gồm các trường headers và 1 trường data
 - Trường data chứa đoạn dữ liệu ứng dụng
 - MSS giới hạn kích cỡ tối đa của trường data trong một segment.
 - Khi TCP gửi một file lớn (file ảnh), nó chia file đó thành các đoạn có kích cỡ MSS (ngoại trừ đoạn cuối cùng có kích cỡ nhỏ hơn MSS)
 - Các ứng dụng tương tác thường truyền các đoạn dữ liệu nhỏ hơn MSS
 - » Ví dụ các ứng dụng đăng nhập từ xa (Telnet), trường dữ liệu trong segment thường là 1 byte. TCP header thường là 20 byte (lớn hơn 12 byte so với UDP header), các segment được gửi bởi Telnet có thể chỉ dài 21 byte.

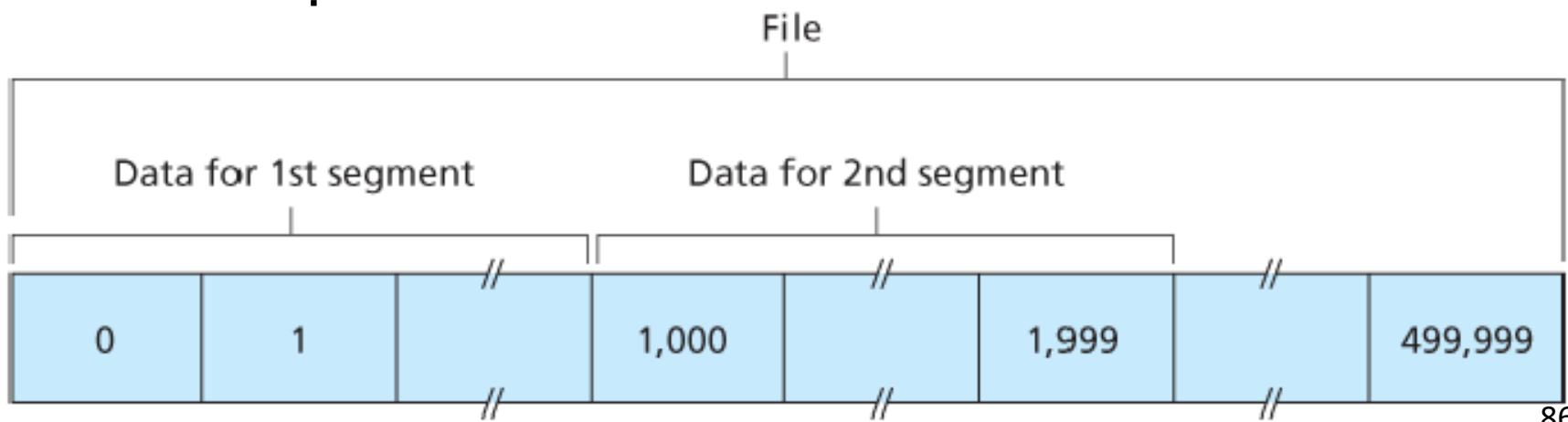
Vận chuyển hướng kết nối: TCP

• Cấu trúc TCP segment



Vận chuyển hướng kết nối: TCP

- Số thứ tự và số báo nhận ACK trong TCP
 - 2 trường quan trọng nhất trong phần TCP header, là phần quan trọng của dịch vụ truyền dữ liệu tin cậy của TCP
 - Số thứ tự
 - Số thứ tự byte của byte đầu tiên trong segment
 - Ví dụ



Vận chuyển hướng kết nối: TCP

- Số thứ tự và số báo nhận ACK trong TCP
 - Số ACK
 - Số thứ tự của byte tiếp theo được mong đợi từ phía bên kia
 - Ví dụ 1
 - Host A nhận được tất cả các byte có số thứ tự từ 0 đến 535 từ host B và giả sử nó sẽ gửi một segment tới host B
 - Host A đang chờ byte số 536 và tất cả các byte tiếp theo trong luồng dữ liệu từ host B. Do đó, host A đặt 536 vào trong trường số ACK của segment mà nó sẽ gửi sang cho B.

Vận chuyển hướng kết nối: TCP

- Số thứ tự và số báo nhận ACK trong TCP
 - Số ACK
 - Ví dụ 2
 - Giả sử host A đã nhận được một segment từ host B có chứa các byte từ 0 đến 535 và một segment khác có chứa các byte từ 900 đến 1000.
 - Vì một lý do nào đó Host A chưa nhận được các byte từ 536 đến 899.
 - Trong trường hợp này, host A vẫn đang đợi byte 536 (và các byte khác) để tái tạo lại luồng dữ liệu của B. Do đó, segment tiếp theo của A sẽ **chứa 536 trong trường số ACK**
 - Do TCP chỉ xác nhận các byte tối đa là byte bị thiếu đầu tiên trong luồng dữ liệu, nên TCP được cho là cung cấp **các ACK tích lũy**

Vận chuyển hướng kết nối: TCP

- Số thứ tự và số báo nhận ACK trong TCP
 - Số ACK
 - Ví dụ 3
 - Host A nhận được segment thứ 3 (các bytes từ 900 tới 1000) trước khi nhận được segment thứ 2 (các byte từ 536 tới 899). Do đó segment thứ 3 đến không theo trật tự.
 - Một host làm gì khi nó nhận được các segment không theo thứ tự trong một kết nối TCP?
 - » Điều thú vị là TCP RFCs không áp đặt bất kỳ nguyên tắc nào ở đây và để quyền quyết định cho các lập trình viên triển khai TCP.

Vận chuyển hướng kết nối: TCP

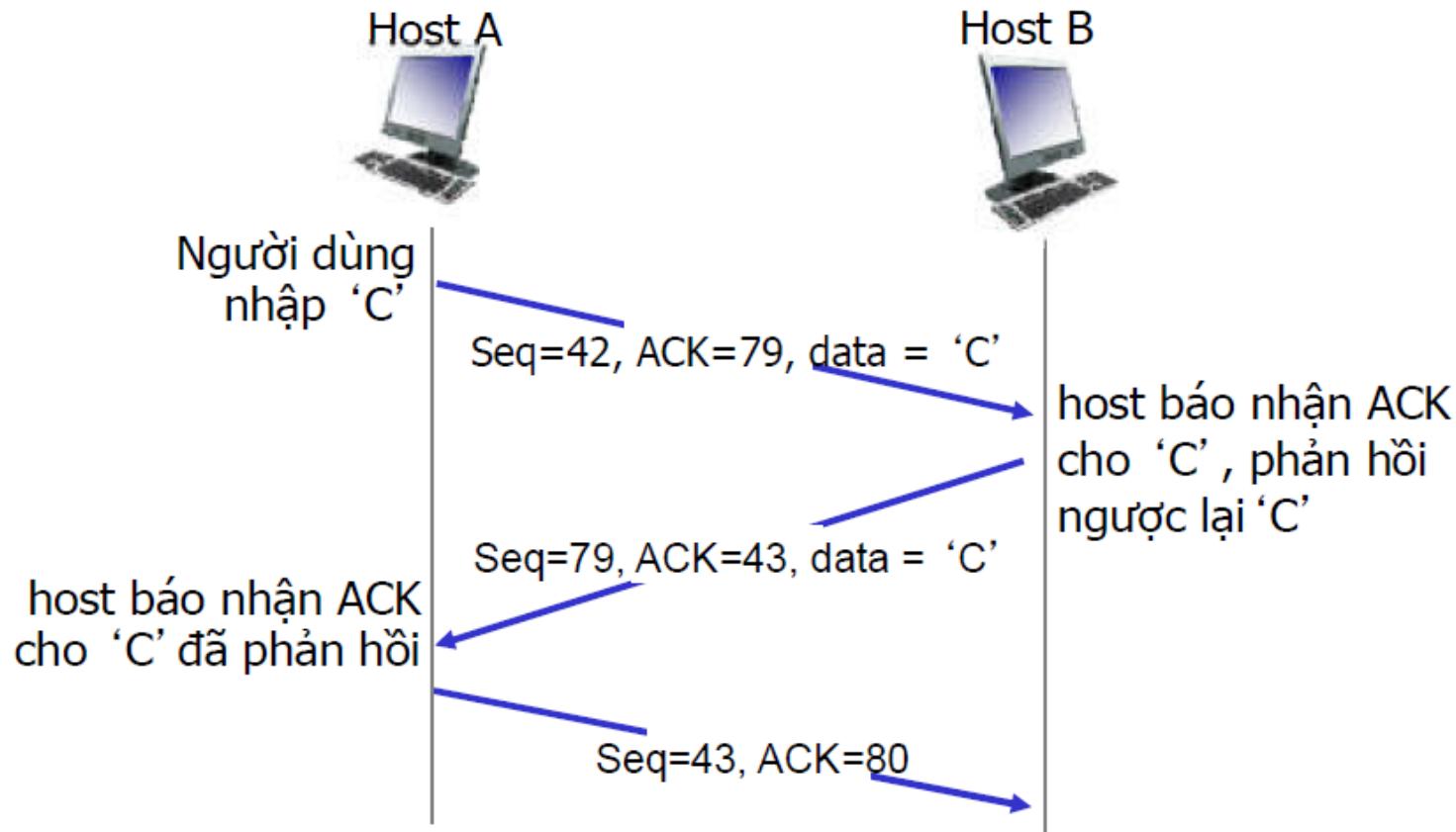
- Số thứ tự và số báo nhận ACK trong TCP
 - Số ACK
 - Ví dụ 3
 - Một host làm gì khi nó nhận được các segment không theo thứ tự trong một kết nối TCP?
 - Về cơ bản có 2 lựa chọn
 - » Bên nhận ngay lập tức loại bỏ các segment không theo thứ tự
 - » Bên nhận giữ lại các byte không theo thứ tự và chờ các byte còn thiếu đến để lấp đầy khoảng trống. Đây là lựa chọn hiệu quả hơn về băng thông mạng và là cách tiếp cận được áp dụng trong thực tế

Vận chuyển hướng kết nối: TCP

- Số thứ tự và số báo nhận ACK trong TCP
 - Kịch bản telnet đơn giản
 - Telnet là giao thức tầng ứng dụng được dùng cho đăng nhập từ xa.
 - Nó chạy trên TCP và được thiết kế để làm việc giữa bất kỳ cặp host nào.
 - Telnet là ứng dụng tương tác

Vận chuyển hướng kết nối: TCP

- Số thứ tự và số báo nhận ACK trong TCP
 - Kịch bản telnet đơn giản



Vận chuyển hướng kết nối: TCP

- TCP round trip time và timeout
 - Giống giao thức rdt, TCP sử dụng cơ chế timeout/retransmit để khôi phục các segment bị mất
 - Khoảng thời gian timeout?
 - Nên lớn hơn RTT của kết nối (thời gian từ khi một segment được gửi tới khi nó được xác nhận ACK), nếu không thì sẽ gửi lại gói tin không cần thiết.
 - Lớn hơn bao nhiêu? RTT nên được ước lượng như thế nào?

Vận chuyển hướng kết nối: TCP

- Ước lượng RTT

- SampleRTT

- Thời gian gửi segment và nhận ACK cho segment đó.
 - Thay vì đo SampleRTT cho từng segment được truyền, hầu hết thực thi TCP thực hiện một phép đo SampleRTT tại một thời điểm nào đó
 - Điều này có nghĩa là ở bất kỳ thời điểm nào, SampleRTT chỉ được ước lượng cho một trong các segment đã truyền nhưng hiện chưa được ACK => mỗi lần có một giá trị SampleRTT mới xấp xỉ cho từng RTT.
 - TCP không tính SampleRTT cho segment truyền lại, nó chỉ đo SampleRTT cho các segment đã được truyền một lần

Vận chuyển hướng kết nối: TCP

- Ước lượng RTT
 - SampleRTT và EstimatedRTT
 - Các giá trị SampleRTT sẽ khác nhau với các segment khác nhau do tắc nghẽn trong router và tải khác nhau ở các hệ thống cuối.
⇒ Cần có một giá trị RTT đại diện được tính bằng trung bình một vài SampleRTT \Leftrightarrow **EstimatedRTT**
 - Khi nhận được một SampleRTT mới, TCP cập nhật EstimatedRTT theo công thức

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

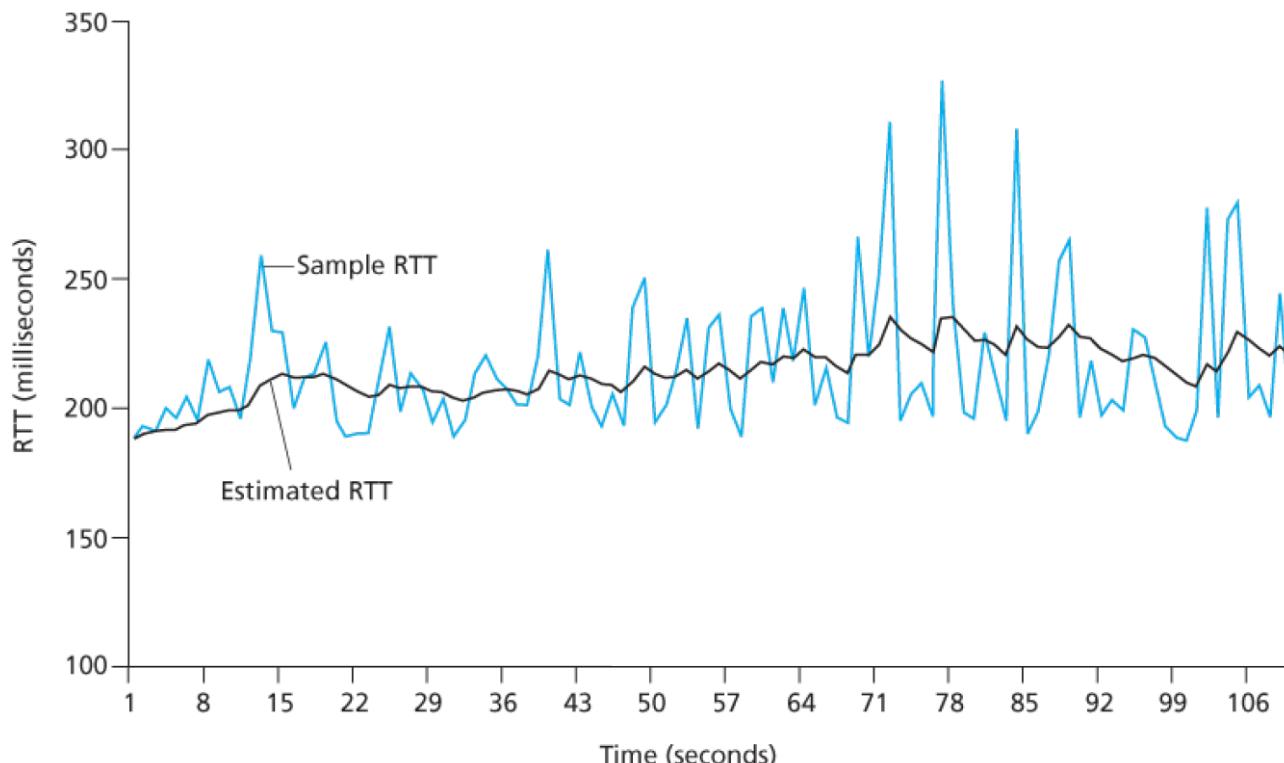
$$\alpha = 0.125$$

exponential weighted moving average (EWMA).

Vận chuyển hướng kết nối: TCP

- **Ước lượng RTT**

- SampleRTT và EstimatedRTT cho kết nối TCP giữa gaia.cs.umass.edu (ở Amherst, Massachusetts) và fantasia.eurecom.fr (ở miền nam nước Pháp).



Vận chuyển hướng kết nối: TCP

- Ước lượng RTT
 - Ngoài EstimatedRTT, cần có một phép đo về sự biến thiên của RTT, kí hiệu **DevRTT**
 - DevRTT là ước lượng SampleRTT sai lệnh bao nhiêu so với EstimatedRTT

$$DevRTT = (1 - \beta) \cdot DevRTT + \beta \cdot (SampleRTT - EstimatedRTT)$$

- DevRTT là một EWMA của sự khác nhau giữa SampleRTT và EstimatedRTT
 - Nếu SampleRTT biến thiên ít, DevRTT sẽ nhỏ
 - Nếu SampleRTT biến thiên nhiều, DevRTT sẽ lớn
 - $\beta = 0.25$.

Vận chuyển hướng kết nối: TCP

- Khoảng thời gian Timeout

- Biết EstimatedRTT và DevRTT => khoảng thời gian timeout nên \geq EstimatedRTT, nếu không phải thực hiện việc truyền lại gói tin không cần thiết.
- Tuy nhiên khoảng thời gian timeout không nên lớn hơn quá nhiều EstimatedRTT, nếu không segment sẽ bị mất, TCP sẽ nhanh chóng truyền lại segment, dẫn tới trễ truyền lớn.

\Rightarrow Nên thiết lập timeout = EstimatedRTT + margin

- Margin lớn khi các giá trị SampleRTT biến thiên nhiều
- Margin nhỏ khi các giá trị SampleRTT biến thiên ít
- DevRTT cũng phát huy tác dụng khi tính timeout

$$\text{Timeout Interval} = \text{Estimated RTT} + 4 \cdot \text{DevRTT}$$

Vận chuyển hướng kết nối: TCP

- Truyền dữ liệu tin cậy
 - 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

Xem xét bên gửi TCP theo cách đơn giản:

- Bỏ qua trùng lặp ACK
- Bỏ qua điều khiển luồng, điều khiển tắc nghẽn

Vận chuyển hướng kết nối: TCP

- Truyền dữ liệu tin cậy
 - Các sự kiện của TCP bên gửi
 - Dữ liệu nhận từ ứng dụng
 - TCP nhận dữ liệu từ ứng dụng, đóng gói dữ liệu thành segment, chuyển segment sang tầng IP. Mỗi segment có một số thứ tự là số thứ tự byte của byte dữ liệu đầu tiên trong segment.
 - TCP khởi tạo bộ định thời khi segment được chuyển xuống tầng IP
 - Thời gian hết hạn cho bộ định thời là **TimeOutInterval** được tính từ EstimatedRTT và DevRTT

Vận chuyển hướng kết nối: TCP

- Truyền dữ liệu tin cậy
 - Các sự kiện của TCP bên gửi
 - Timeout
 - TCP đáp ứng với sự kiện timeout bằng cách truyền lại segment bị timeout.
 - TCP sau đó khởi tạo lại bộ định thời

Vận chuyển hướng kết nối: TCP

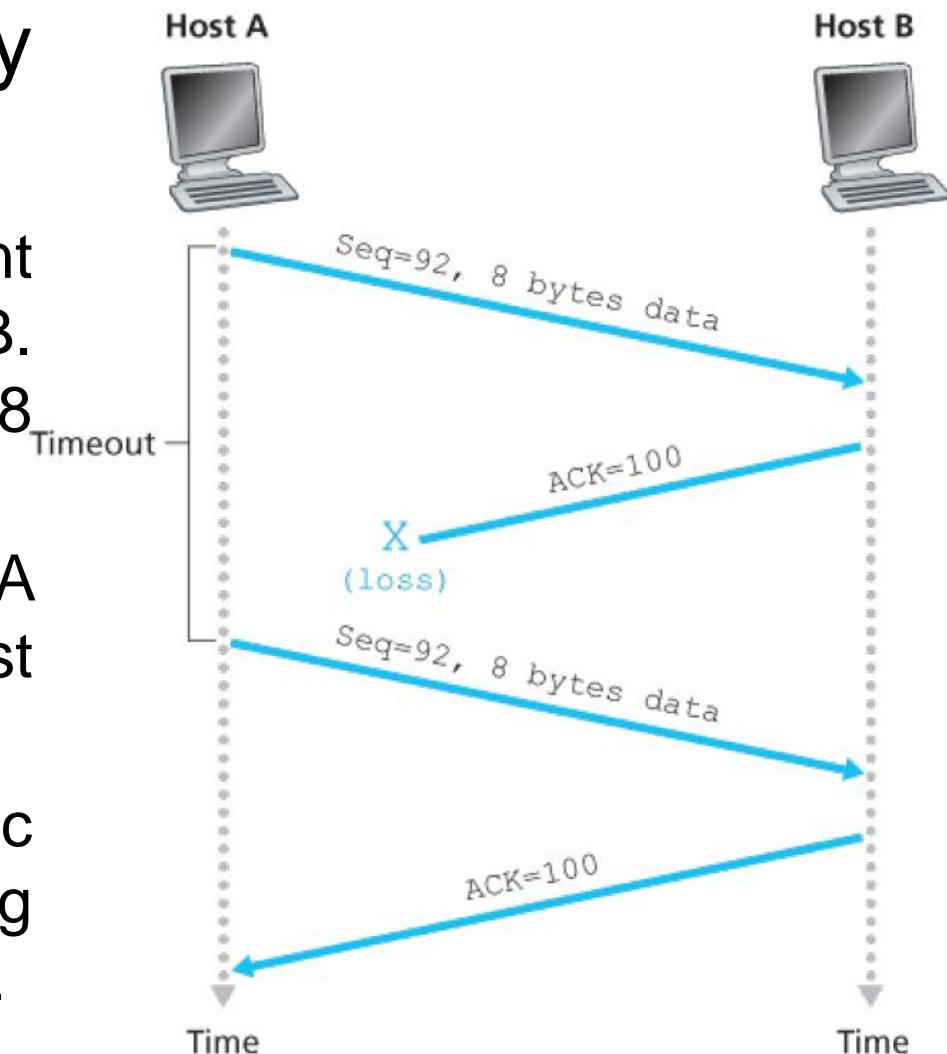
- Truyền dữ liệu tin cậy
 - Các sự kiện của TCP bên gửi
 - ACK đã nhận
 - Nếu ACK báo nhận cho các segment chưa được báo nhận trước đó, thì:
 - » Cập nhật lại các segment đã được báo nhận
 - » Khởi tạo bộ định thời nếu vẫn còn các segment chưa được báo nhận
 - TCP so sánh giá trị ACK y với biến **SendBase** của nó
 - TCP sử dụng ACK tích lũy, để cho y xác nhận việc nhận tất cả các byte trước byte số y .
 - » Nếu $y > \text{SendBase}$, khi đó ACK đang xác nhận một hoặc nhiều segment chưa được xác nhận trước đó. Khi đó, bên gửi sẽ cập nhật biến SendBase của mình.
 - Khởi động bộ định thời nếu hiện tại không có bất kỳ segment nào chưa được xác nhận.

Vận chuyển hướng kết nối: TCP

- Truyền dữ liệu tin cậy

- Kịch bản 1

- Host A gửi 1 segment (92) tới host B. Segment này chứa 8 byte dữ liệu.
 - Sau khi gửi, Host A chờ ACK (100) từ host B.
 - Host B nhận được segment từ A nhưng ACK từ B tới A bị mất.



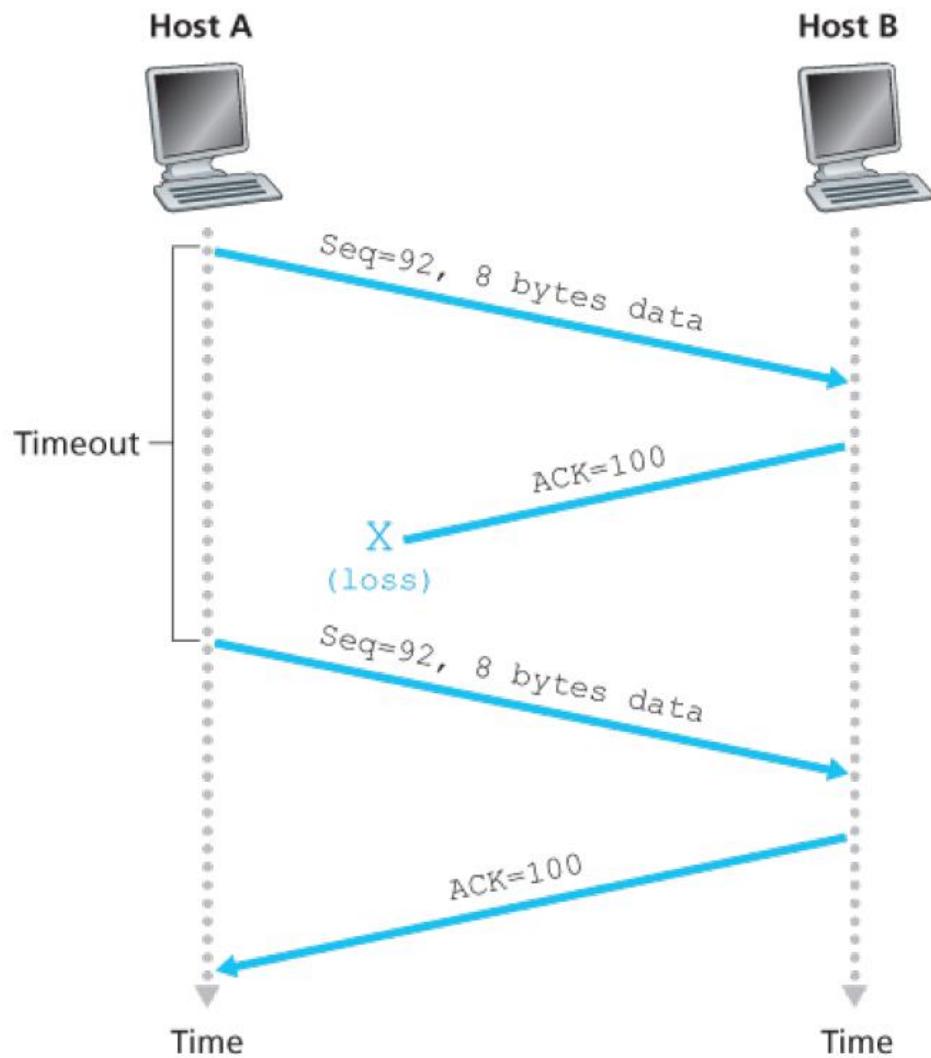
Vận chuyển hướng kết nối: TCP

- Truyền dữ liệu tin cậy

- Kịch bản 1

⇒ Trong trường hợp này sự kiện timeout xuất hiện và host A truyền lại segment.

- Khi host B nhận được segment truyền lại từ A, nó quan sát số thứ tự segment chứa data đã được truyền lại. Khi đó, TCP ở host B sẽ loại bỏ các byte trong segment truyền lại



Vận chuyển hướng kết nối: TCP

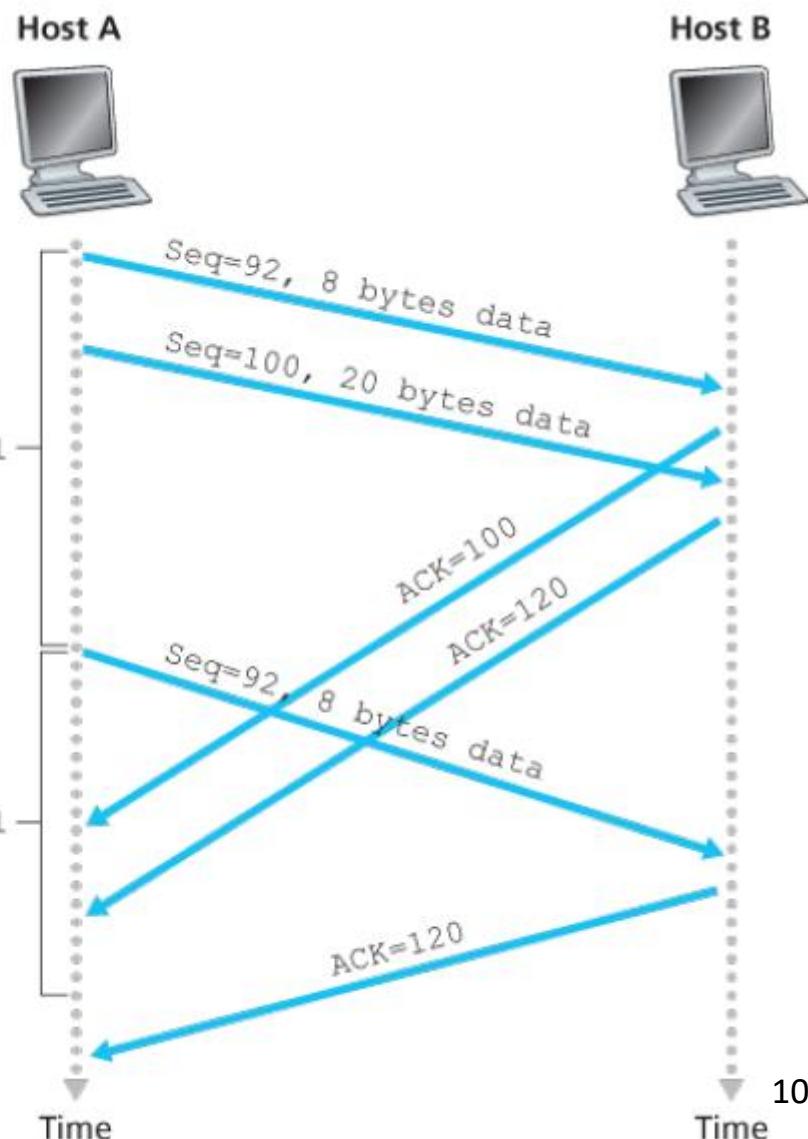
- Truyền dữ liệu tin cậy

- Kịch bản 2

- Host A gửi 2 segment cho host B

- Segment 1: 92, 8 bytes
 - Segment 2: 100, 20 bytes

- Giả sử 2 segments này đều đến B nguyên vẹn và B gửi 2 ACK (100, 120) tương ứng với 2 segment này sang cho host A



Vận chuyển hướng kết nối: TCP

- Truyền dữ liệu tin cậy

- Lịch bản 2

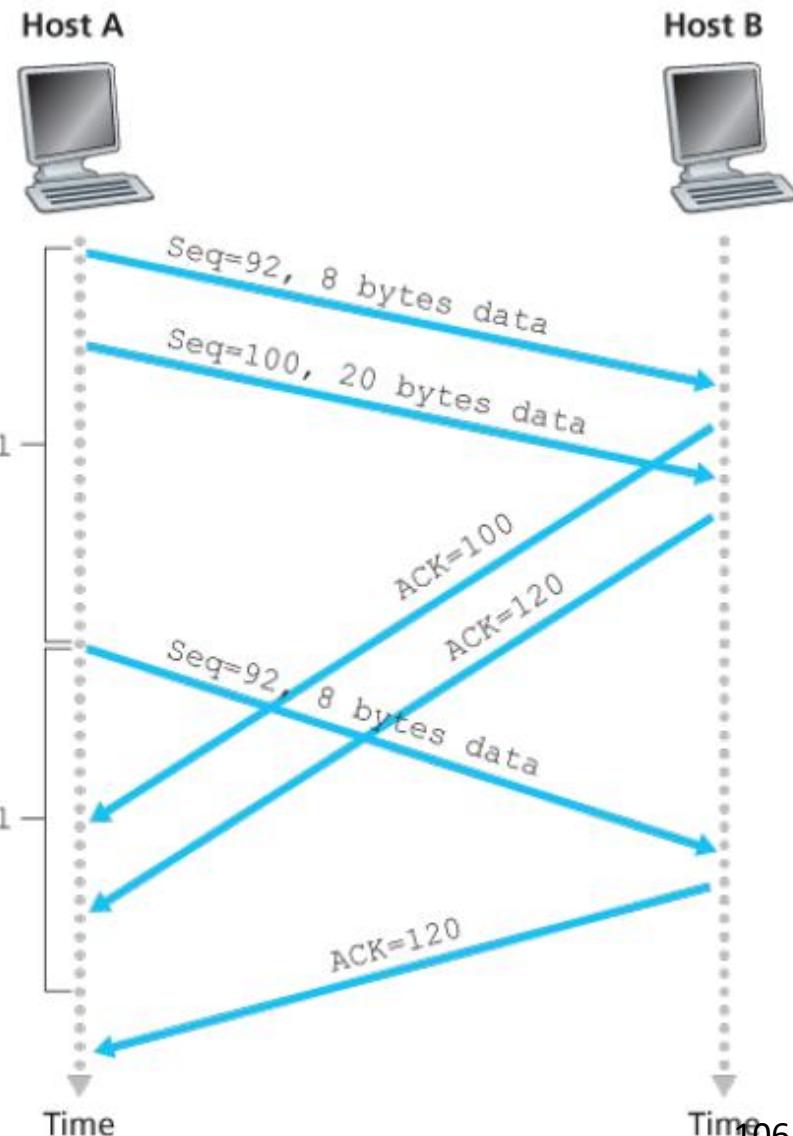
- Giả sử không có ACK nào đến trước thời gian timeout.

seq=92 timeout interval

- Khi sự kiện timeout xuất hiện, host A gửi lại segment đầu tiên (92) và khởi động lại bộ định thời.

seq=92 timeout interval

- Ngay khi ACK cho segment thứ 2 tới trước timeout mới, segment 2 sẽ không được truyền lại.

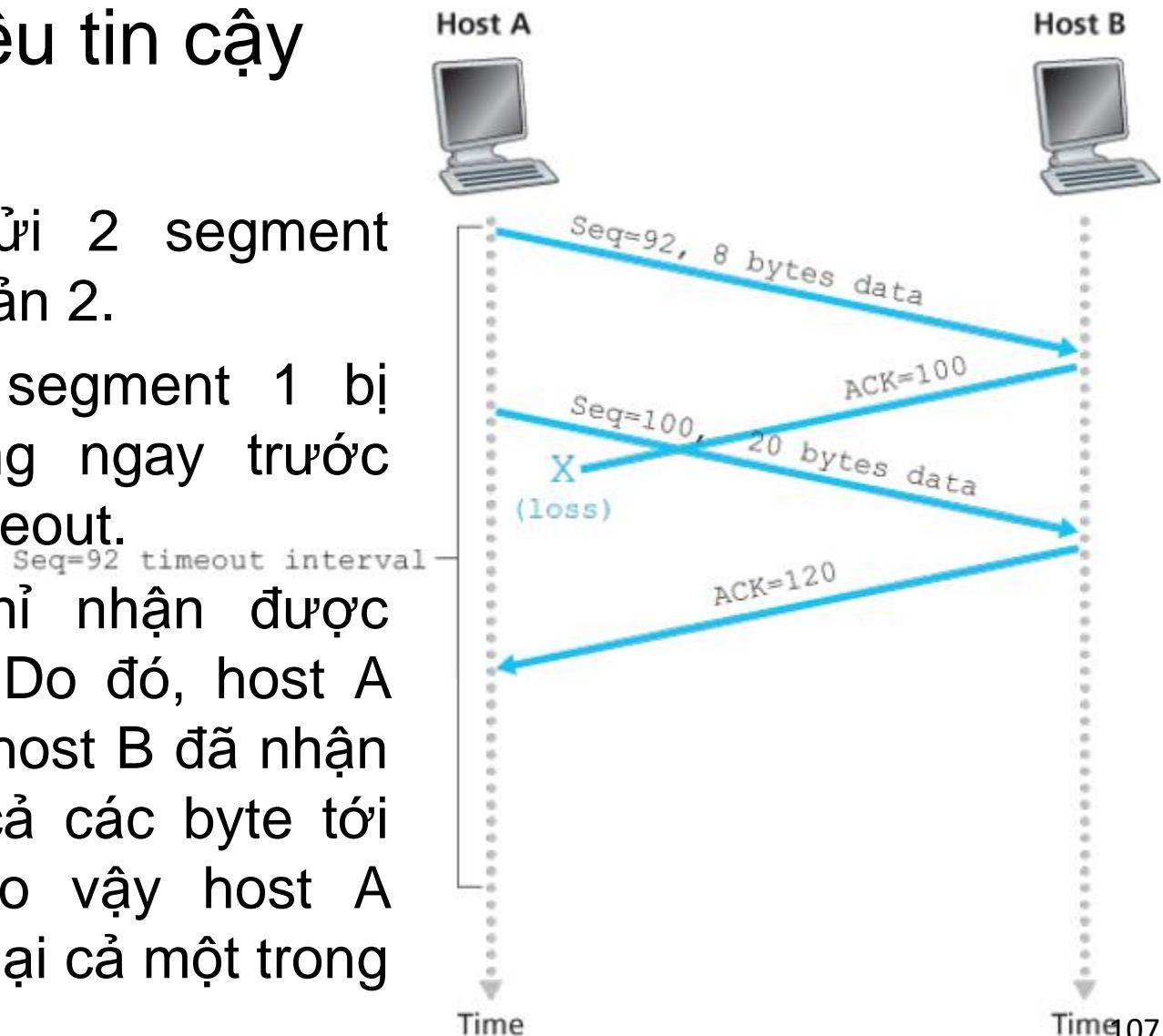


Vận chuyển hướng kết nối: TCP

- Truyền dữ liệu tin cậy

- Kịch bản 3

- Host A gửi 2 segment như kịch bản 2.
 - ACK của segment 1 bị mất, nhưng ngay trước sự kiện timeout.
 - Host A chỉ nhận được ACK 120. Do đó, host A biết được host B đã nhận được tất cả các byte tới số 119, do vậy host A không gửi lại cả một trong 2 segment



Vận chuyển hướng kết nối: TCP

- Truyền dữ liệu tin cậy
 - Tăng gấp đôi thời gian timeout
 - Bất kể khi nào sự kiện timeout xuất hiện, TCP truyền lại gói tin chưa có ACK với số thứ tự nhỏ nhất. Nhưng mỗi khi TCP truyền lại, nó thiết lập khoảng thời gian timeout tiếp theo gấp 2 lần giá trị trước đó thay vì tính toán từ EstimatedRTT và DevRTT.

Vận chuyển hướng kết nối: TCP

- Truyền dữ liệu tin cậy
 - Tăng gấp đôi thời gian timeout
 - Ví dụ
 - **TimeoutInterval** cho segment chưa được ACK muộn nhất = 0.75 giây khi bộ định thời đầu tiên hết hạn.
 - TCP khi đó sẽ truyền lại segment này và thiết lập thời hạn mới = 1.5 giây.
 - Nếu bộ định thời lại hết hạn sau 1.5 giây, TCP sẽ lại truyền lại segment này, và thiết lập thời hạn = 3.0 giây
 - ⇒ Các khoảng thời gian này tăng theo số mũ sau mỗi lần truyền lại. Tuy nhiên, bất cứ khi nào bộ định thời bắt đầu sau một trong 2 sự kiện (nhận dữ liệu từ ứng dụng, nhận ACK), TimeoutInterval được lấy từ các giá trị EstimatedRTT và DevRTT gần nhất.

Vận chuyển hướng kết nối: TCP

- Truyền dữ liệu tin cậy
 - Truyền lại nhanh
 - Một trong những vấn đề với việc truyền lại dựa trên kích hoạt timeout đó là chu kỳ timeout có thể dài. Khi một segment bị mất, chu kỳ timeout dài sẽ buộc người gửi phải trì hoãn việc gửi gói tin bị mất, do đó tăng trễ đầu cuối.
 - May mắn, bên gửi thường có thể phát hiện gói tin bị mất trước khi sự kiện timeout xuất hiện bằng cách ghi nhận cái gọi là **ACK trùng lặp**
 - **ACK trùng lặp** là một ACK xác nhận lại một segment mà bên gửi đã nhận được ACK trước đó

Vận chuyển hướng kết nối: TCP

- Truyền dữ liệu tin cậy
 - Tạo ACK trong TCP

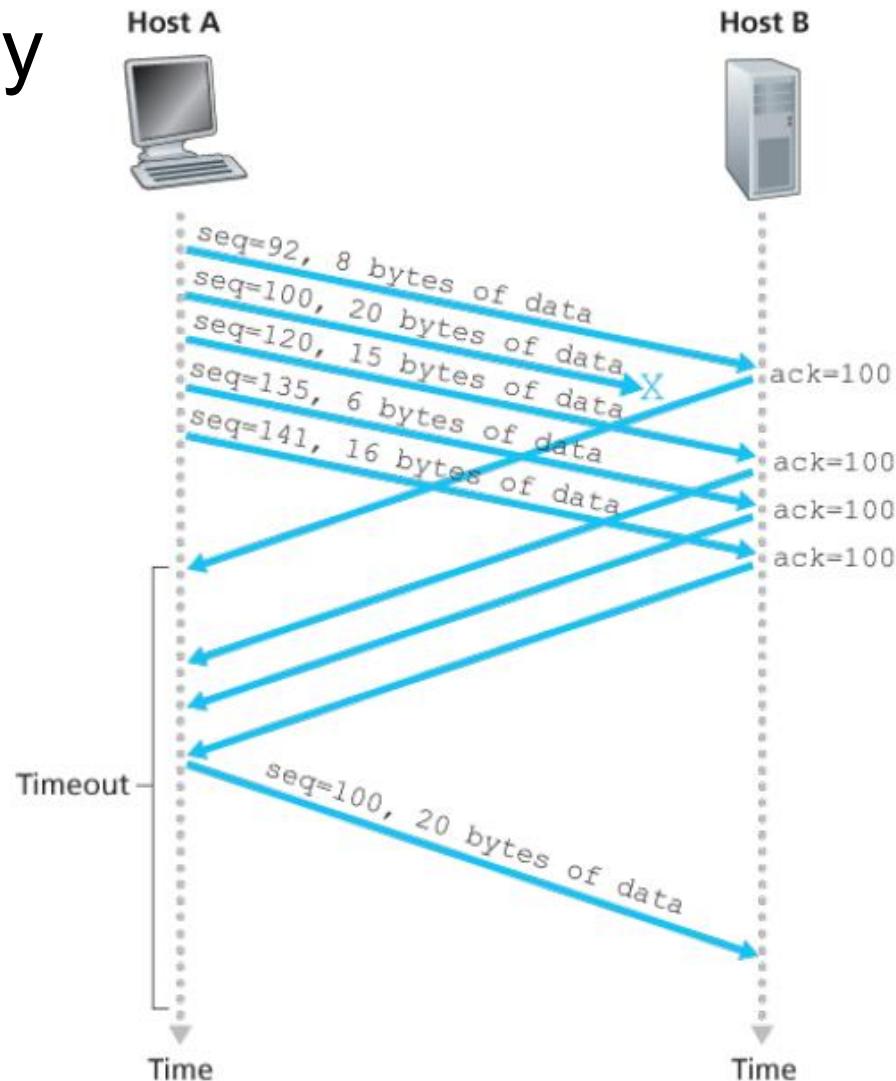
Sự kiện tại bên nhận	Hành động của TCP tại bên nhận
Segment đến đúng thứ tự với số thứ tự mong muốn. Tất cả dữ liệu đến đã được báo nhận	ACK bị trễ. Chờ 500ms cho segment tiếp theo. Nếu không có segment tiếp theo thì gửi ACK
Segment đến đúng thứ tự với số thứ tự mong muốn. Một segment khác đang chờ ACK	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

Vận chuyển hướng kết nối: TCP

- Truyền dữ liệu tin cậy
 - Tạo ACK trong TCP
 - Do bên gửi thường gửi một lượng lớn các segment, nếu một segment bị mất, sẽ có thể có nhiều ACK trùng lặp.
 - Nếu TCP bên gửi nhận được 3 ACK trùng lặp cho cùng dữ liệu, nó coi đây là một dấu hiệu cho thấy rằng segment theo sau segment đã được ACK 3 lần đã bị mất
 - Trong trường hợp nhận được 3 ACK trùng lặp, TCP bên gửi thực hiện **fast retransmit**, truyền lại segment bị mất trước khi bộ định thời của segment đó quá hạn.

Vận chuyển hướng kết nối: TCP

- Truyền dữ liệu tin cậy
 - Tạo ACK trong TCP



Vận chuyển hướng kết nối: TCP

- Truyền dữ liệu tin cậy
 - Go-Back-N hay Selective Repeat
 - Các TCP ACK tích lũy và các segment được nhận đúng nhưng không theo trật tự không được bén nhận ACK riêng cho từng segment. Do đó, TCP bén gửi chỉ cần duy trì số thứ tự nhỏ nhất của một byte đã truyền nhưng chưa được ACK (SendBase) và số thứ tự của byte tiếp theo sẽ được gửi (NextSeqNum). **Trong trường hợp này TCP ≈ GBN**

Vận chuyển hướng kết nối: TCP

- Truyền dữ liệu tin cậy
 - Go-Back-N hay Selective Repeat
 - Điểm khác
 - Bên gửi gửi 1,2,...,N segments và tất cả đến nơi nhận theo đúng thứ tự, không lỗi. Giả sử ACK cho gói $n < N$ bị mất, ($N-1$) ACK đến đầu nhận trước các timeouts tương ứng.
 - GBN truyền lại n và các gói sau n ($n+1, n+2, \dots, N$)
 - TCP truyền lại nhiều nhất 1 segment n. Hơn nữa, TCP thậm chí không truyền lại segment n nếu ACK cho segment ($n+1$) đến trước timeout của segment n

Vận chuyển hướng kết nối: TCP

- Truyền dữ liệu tin cậy
 - Go-Back-N hay Selective Repeat
 - ACK có lựa chọn (selective ACK) cho phép TCP bên nhận xác nhận các segments không theo thứ tự một cách có chọn lọc thay vì báo nhận tích lũy segment theo thứ tự, được nhận đúng cuối cùng.
 - Khi kết hợp với truyền lại có chọn lọc – bỏ qua việc truyền lại các segments đã được người nhận ghi nhận một cách có chọn lọc – **Lúc này TCP giống như giao thức SR**. Do đó, cơ chế khắc phục lỗi của TCP có lẽ tốt nhất nên được phân là loại lai giữa giao thức GBN và SR.

Vận chuyển hướng kết nối: TCP

- Điều khiển luồng
 - Các host ở mỗi bên gửi/nhận của kết nối TCP dành riêng một **bộ đệm nhận** cho kết nối.
 - Khi kết nối TCP nhận được các byte đúng và theo trình tự, nó sẽ đặt dữ liệu vào bộ đệm nhận.

Vận chuyển hướng kết nối: TCP

- Điều khiển luồng
 - Tiến trình ứng dụng liên quan sẽ đọc dữ liệu từ bộ đệm, nhưng không nhất thiết phải đọc ngay khi dữ liệu đến. Ứng dụng nhận có thể đang bận với một số tác vụ khác và thậm chí có thể không đọc dữ liệu khi nó đến.
 - Nếu ứng dụng đọc dữ liệu chậm, bên gửi rất dễ làm tràn bộ đệm nhận của kết nối bằng cách gửi quá nhiều và quá nhanh dữ liệu
=> TCP cung cấp dịch vụ **flow-control service** cho các ứng dụng để loại trừ khả năng bên gửi làm tràn bộ đệm của bên nhận.

Vận chuyển hướng kết nối: TCP

- Điều khiển luồng
 - Bên gửi TCP cũng có thể bị điều chỉnh do tắc nghẽn trong mạng IP \leftrightarrow **congestion control**
 - Mặc dù các hành động được thực hiện bởi điều khiển luồng và điều khiển tắc nghẽn là giống nhau (đều điều chỉnh người gửi) nhưng lý do thực hiện chúng là khác nhau

Vận chuyển hướng kết nối: TCP

- Điều khiển luồng
 - Cửa sổ nhận - **receive window**
 - TCP cung cấp dịch vụ điều khiển luồng bằng cách yêu cầu bên gửi duy trì một biến **receive window**.
 - Cửa sổ nhận được sử dụng để cung cấp cho người gửi thông tin về không gian bộ nhớ đệm còn trống sẵn có ở bên nhận là bao nhiêu.
 - Vì TCP là song công, người gửi ở mỗi phía của kết nối duy trì một cửa sổ nhận khác nhau.

Vận chuyển hướng kết nối: TCP

- Điều khiển luồng
 - Cửa sổ nhận - **receive window**
 - Ví dụ trong ngữ cảnh truyền file
 - Host A đang gửi 1 file lói tới host B qua kết nối TCP.
 - Host B phân bổ bộ nhớ đệm nhận cho kết nối này – **RcvBuffer**
 - Tiến trình ứng dụng ở host B đọc dữ liệu từ bộ nhớ đệm. Có một số biến sau:
 - » **LastByteRead**: số của byte cuối cùng trong dòng dữ liệu đọc từ bộ nhớ đệm bởi tiến trình ứng dụng trong host B
 - » **LastByteRcvd**: số byte cuối cùng trong dòng dữ liệu đã đến từ mạng và đã được đặt trong bộ nhớ đệm nhận tại host B

Vận chuyển hướng kết nối: TCP

- Điều khiển luồng
 - Cửa sổ nhận - **receive window**
 - Ví dụ trong ngữ cảnh truyền file
 - Do TCP không được phép làm tràn bộ nhớ đệm đã được phân bổ, ta phải có

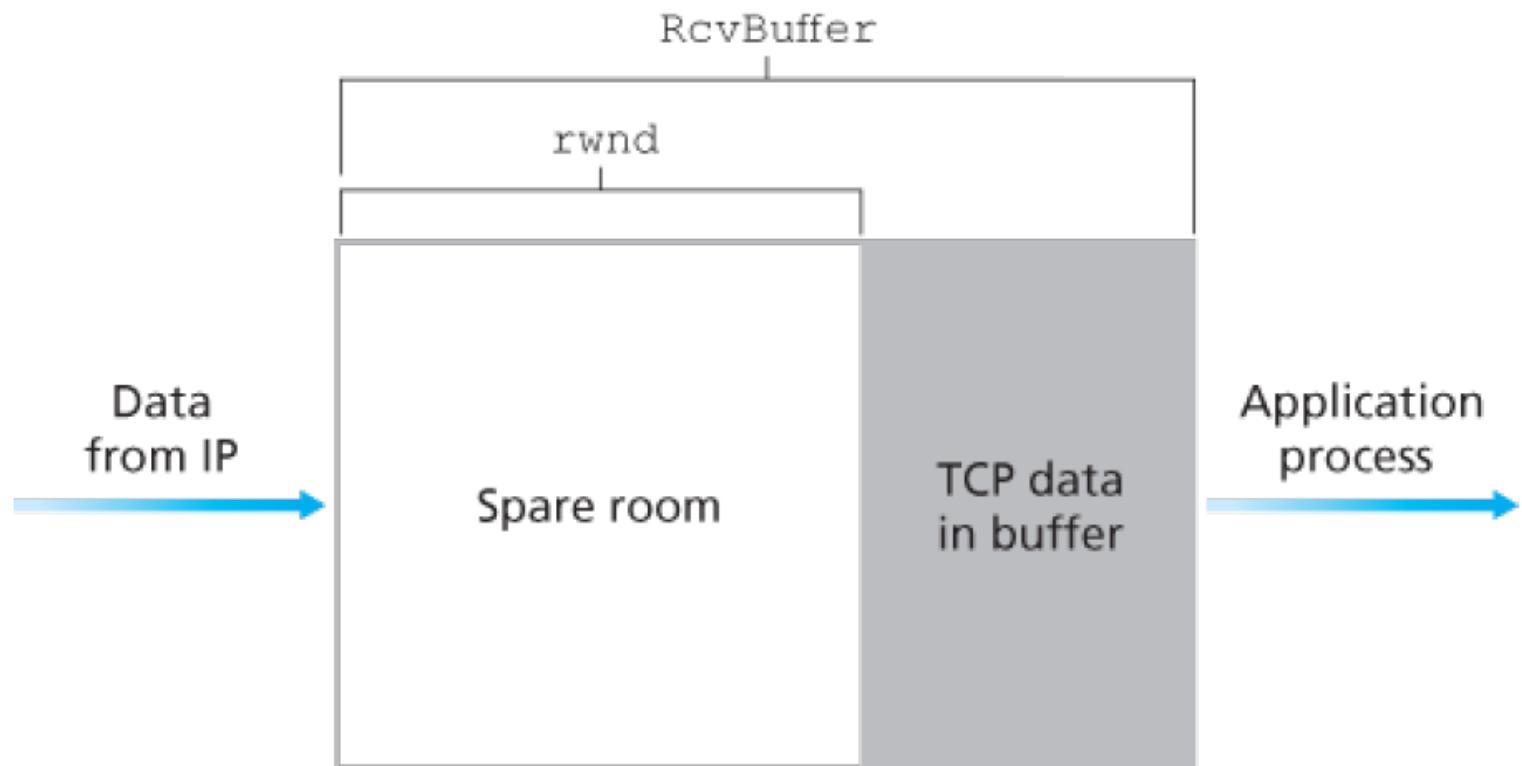
LastByteRcvd – LastByteRead ≤ RcvBuffer

- Cửa sổ nhận **rwnd** được thiết lập = dung lượng không gian trống trong bộ nhớ đệm. Do không gian trống thay đổi theo thời gian nên **rwnd** cũng thay đổi

$rwnd=RcvBuffer-[LastByteRcvd-LastByteRead]$

Vận chuyển hướng kết nối: TCP

- Điều khiển luồng
 - Cửa sổ nhận - **receive window**



Vận chuyển hướng kết nối: TCP

- Điều khiển luồng với biến **rwnd**
 - Host B nói với host A không gian trống nó có trong bộ nhớ đệm là bao nhiêu bằng cách đặt giá trị hiện tại của **rwnd** của nó trong trường cửa sổ nhận của từng segment nó gửi tới host A.
 - Ban đầu, host B đặt **rwnd = RcvBuffer**
 - Host A theo dõi 2 biến **LastByteSent** và **LastByteAcked**
 - LastByteSent – LastByteAcked**: là lượng dữ liệu chưa được báo nhận mà host A đã gửi lên kết nối
 - Duy trì lượng dữ liệu chưa được báo nhận $< \text{rwnd}$. Host A được đảm bảo nó không làm tràn bộ nhớ đệm nhận ở Host B. Do đó, host A đảm bảo trong suốt quá trình kết nối rằng $\text{LastByteSent} - \text{LastByteAcked} \leq \text{rwnd}$

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

Vận chuyển hướng kết nối: TCP

- Điều khiển luồng với biến **rwnd**
 - Vấn đề
 - Giả sử bộ nhớ đệm nhận của Host B đầy => **rwnd** = 0.
 - Sau khi báo **rwnd** = 0 cho Host A, và giả sử host B không có gì để gửi cho A.
 - Khi tiến trình ứng dụng ở B làm trống bộ nhớ đệm, TCP không gửi các segments mới với các giá trị **rwnd** mới tới host A. Do đó, host A không được thông báo về một không gian đã được mở trong bộ đệm nhận của host B. Host A bị chặn và không thể truyền thêm dữ liệu.

Vận chuyển hướng kết nối: TCP

- Điều khiển luồng với biến **rwnd**
 - Vấn đề
 - Khắc phục
 - Đặc tả TCP yêu cầu host A tiếp tục gửi các segments với một byte dữ liệu khi cửa sổ nhận của B = 0. Các segments này sẽ được báo nhận bởi bên nhận.
 - Khi bộ nhớ đệm sẽ bắt đầu trống và các ACK sẽ chứa một giá trị rwnd khác 0.

Vận chuyển hướng kết nối: TCP

- Quản lý kết nối TCP
 - Kết nối TCP được thiết lập như thế nào?
 - Giả sử một tiến trình đang chạy trong một host (client) muốn khởi tạo kết nối với tiến trình khác trong host khác (server).
 - Tiến trình ứng dụng client trước tiên thông báo cho client TCP rằng nó muốn thiết lập một kết nối tới một tiến trình trong server
 - Client TCP khi đó tiến hành thiết lập một kết nối TCP với server TCP theo cách sau:

Vận chuyển hướng kết nối: TCP

- Quản lý kết nối TCP
 - Kết nối TCP được thiết lập như thế nào?
 - Bước 1:
 - Client TCP trước tiên gửi một segment TCP đặc biệt tới server TCP. Segment đặc biệt này không chứa dữ liệu tầng ứng dụng. Nhưng một trong các bit cờ trong phần header của segment – bit SYN được thiết lập = 1 (segment đặc biệt được gọi là **SYN segment**).
 - Ngoài ra, client chọn ngẫu nhiên một số thứ tự ban đầu (client_isn) và đặt số này vào trong trường số thứ tự của segment TCP SYN đầu tiên. Segment này được đóng gói với IP datagram và gửi tới cho server.

Vận chuyển hướng kết nối: TCP

- Quản lý kết nối TCP
 - Kết nối TCP được thiết lập như thế nào?
 - Bước 2:
 - Khi IP datagram chứa TCP SYN segment tới server, server lấy ra TCP SYN segment từ datagram, phân bổ các biến và các bộ đệm TCP cho kết nối, và gửi một segment đồng ý kết nối tới client.
 - Segment đồng ý kết nối cũng không chứa dữ liệu tầng ứng dụng. Tuy nhiên, nó chứa 3 thông tin quan trọng trong segment header: bit **SYN = 1**, trường **ACK = client_isn+1**
 - Cuối cùng, server chọn số thứ tự bắt đầu của chính nó (**server_isn**) và đặt giá trị này vào trường số thứ tự của TCP segment header

Vận chuyển hướng kết nối: TCP

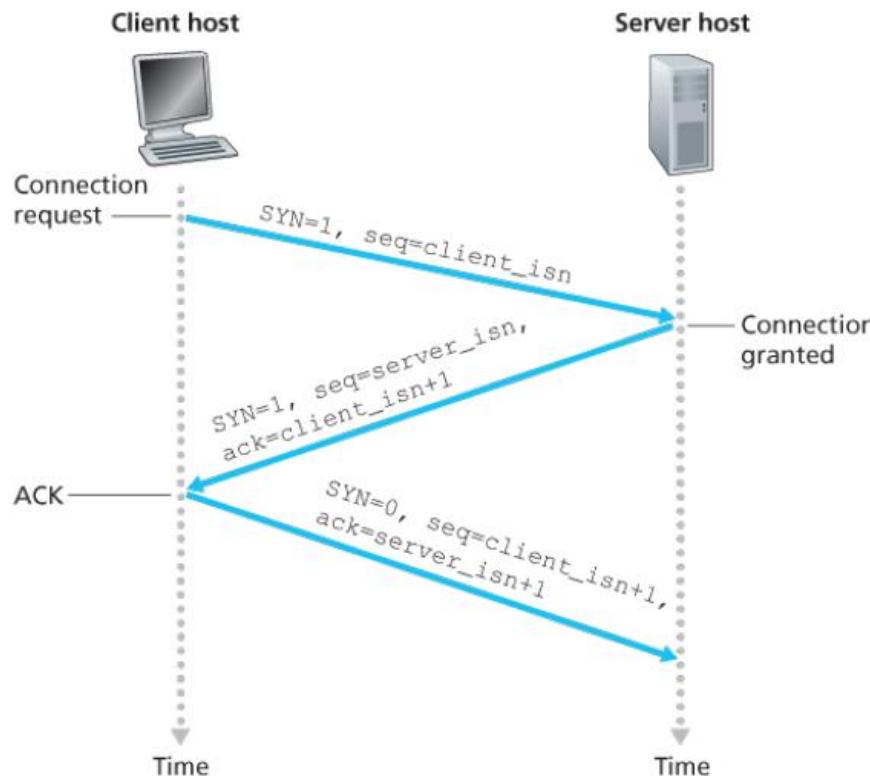
- Quản lý kết nối TCP
 - Kết nối TCP được thiết lập như thế nào?
 - Bước 2:
 - Thực tế segment đồng ý kết nối nói rằng “tôi đã nhận được gói SYN của bạn để bắt đầu một kết nối với số thứ tự ban đầu của bạn là **client_isn**, tôi đồng ý thiết lập kết nối này. Số thứ tự bắt đầu của tôi là **server_isn**”
 - Segment đồng ý kết nối gọi là **SYNACK segment**

Vận chuyển hướng kết nối: TCP

- Quản lý kết nối TCP
 - Kết nối TCP được thiết lập như thế nào?
 - Bước 3:
 - Khi nhận được **SYNACK segment** client cũng phân bổ các bộ nhớ đệm và các biến cho kết nối.
 - Client sau đó sẽ gửi cho server một segment khác, segment cuối cùng này báo đã nhận được segment đồng ý kết nối của server (client làm như vậy bằng cách đặt giá trị **server_isn+1** vào trong trường ACK của TCP segment header)
 - Bit SYN được thiết lập = 0, bởi vì kết nối đã được thiết lập
 - Giai đoạn 3 của quá trình bắt tay 3 bước này có thể chứa dữ liệu client gửi tới server trong segment payload.

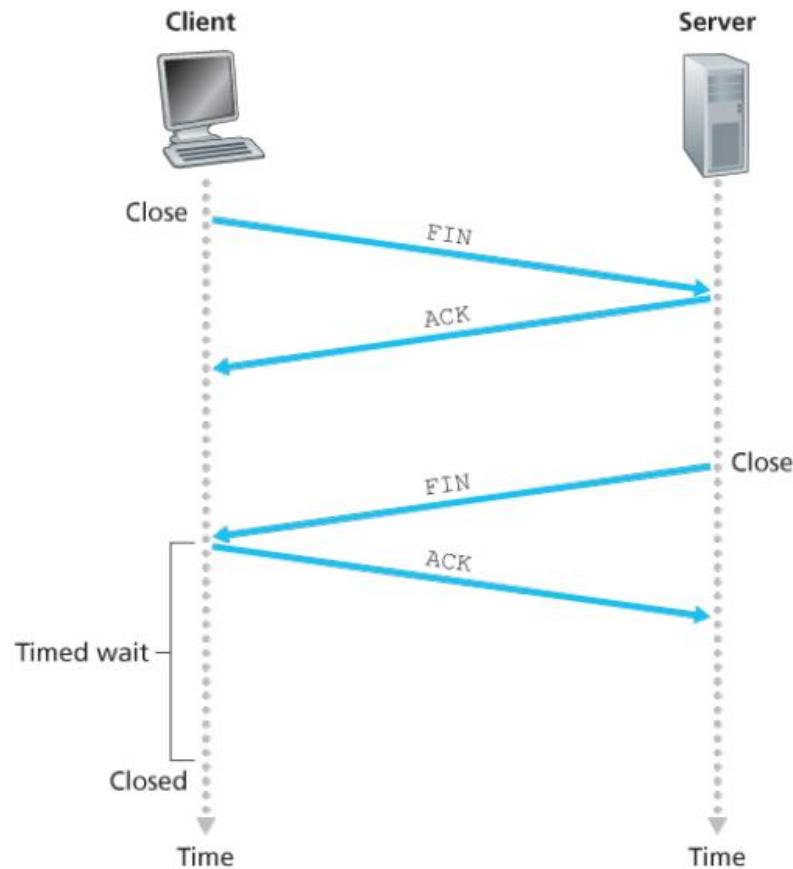
Vận chuyển hướng kết nối: TCP

- Quản lý kết nối TCP
 - Kết nối TCP được thiết lập như thế nào?
 - Thủ tục bắt tay 3 bước: three-way handshake



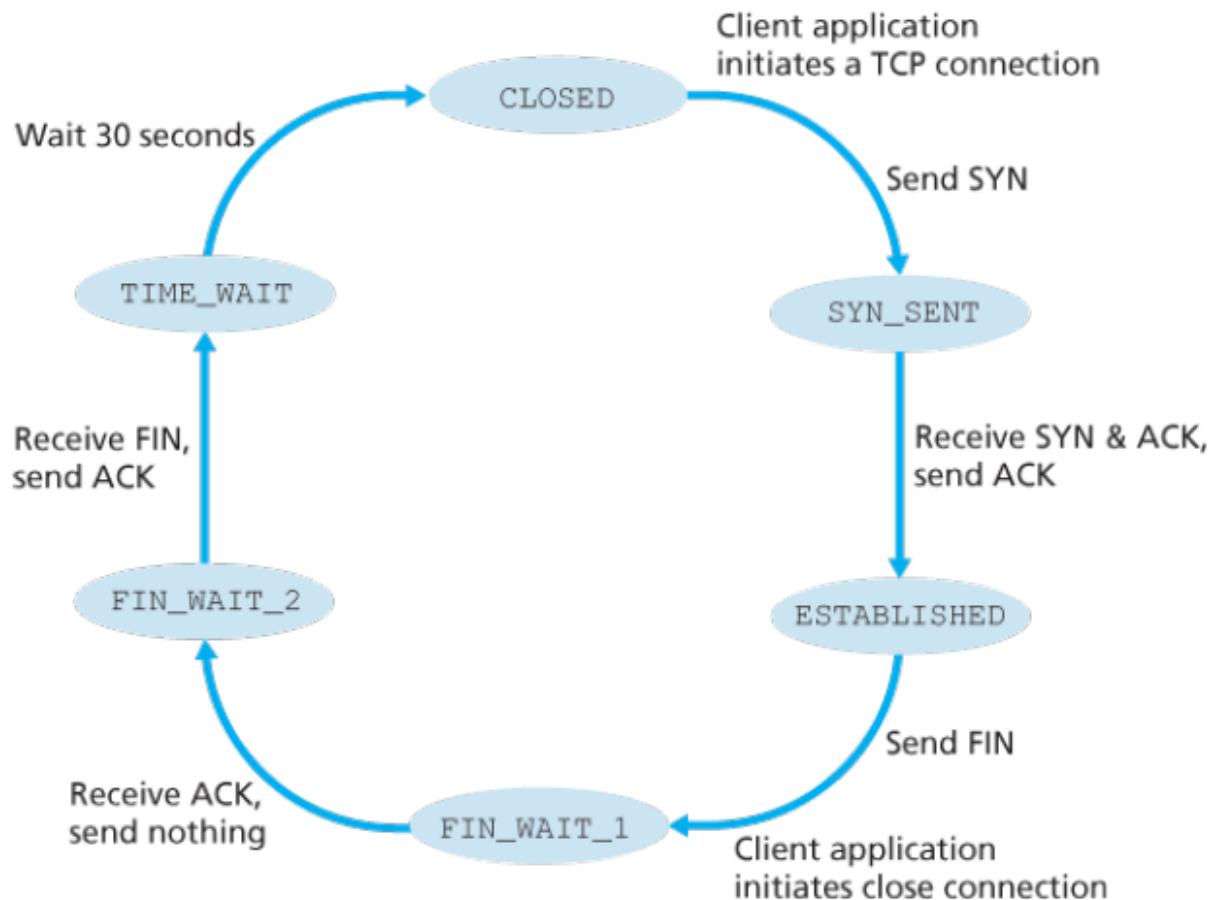
Vận chuyển hướng kết nối: TCP

- Quản lý kết nối TCP
 - Đóng kết nối TCP
- Giả sử client quyết định đóng kết nối. Khi đó tiến trình ứng dụng client đưa ra một lệnh đóng <=> TCP client gửi một TCP segment đặc biệt tới tiến trình server
- Segment đặc biệt có 1 bít cờ trong header là **FIN = 1**
- Khi server nhận được segment này, nó gửi cho client một segment ACK và gửi segment đóng kết nối của nó **FIN = 1**
- Client báo nhận segment đóng kết nối của server



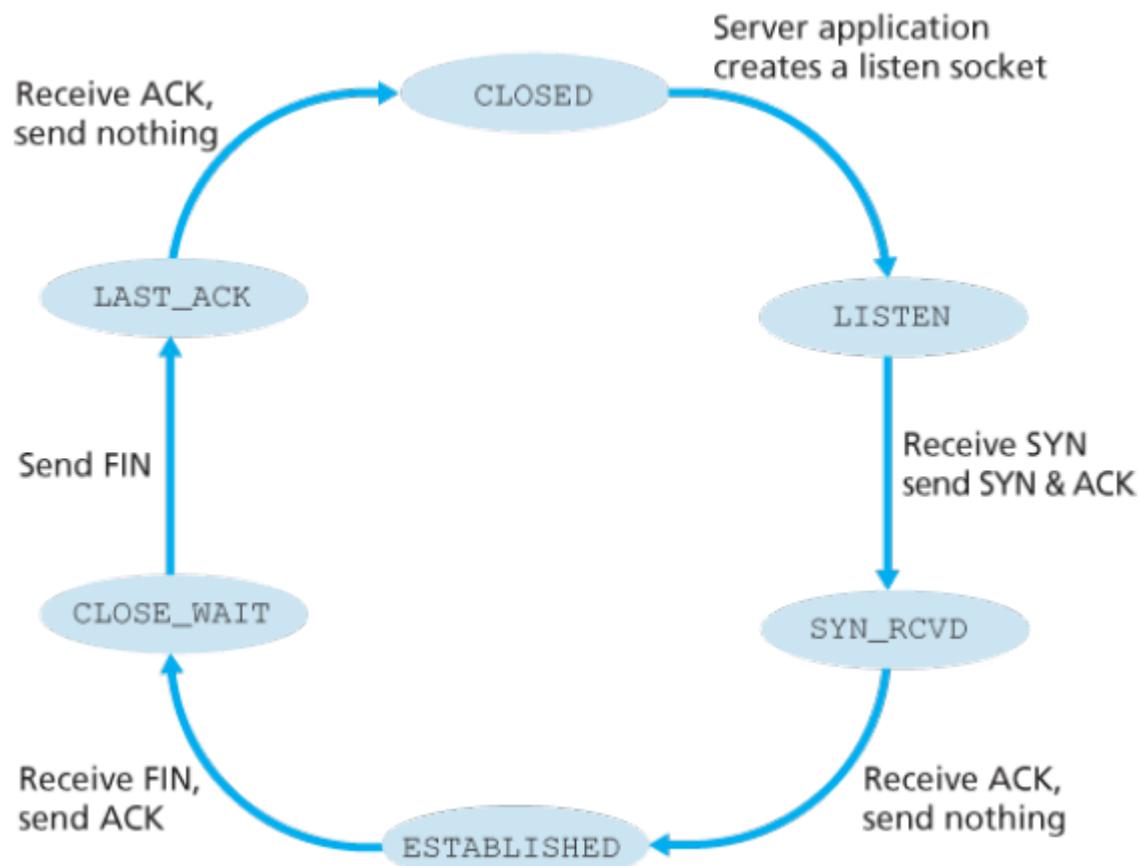
Vận chuyển hướng kết nối: TCP

- Quản lý kết nối TCP
 - Các trạng thái TCP phía client



Vận chuyển hướng kết nối: TCP

- Quản lý kết nối TCP
 - Các trạng thái TCP phía server

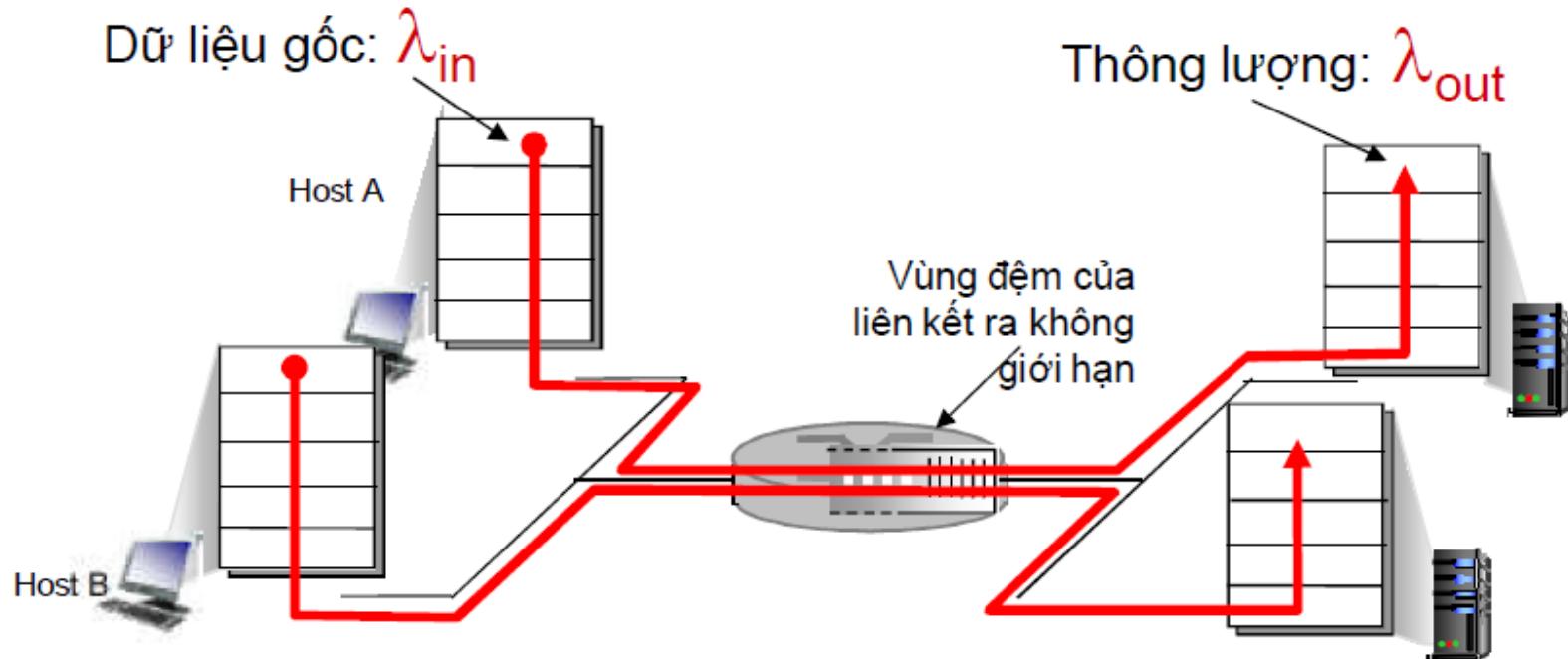


Nội dung

- Các dịch vụ tầng giao vận
- Ghép kênh và phân kênh
- Vận chuyển không kết nối: UDP
- Các nguyên tắc truyền dữ liệu tin cậy
- Vận chuyển hướng kết nối: TCP
- **Các nguyên lý điều khiển tắc nghẽn**
- Điều khiển tắc nghẽn TCP

Các nguyên lý điều khiển tắc nghẽn

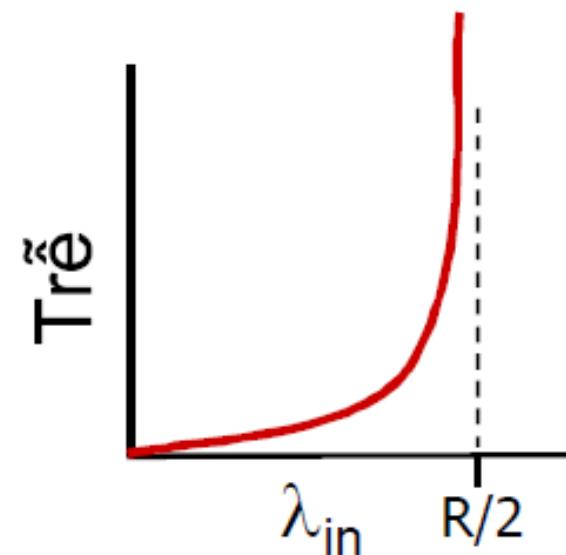
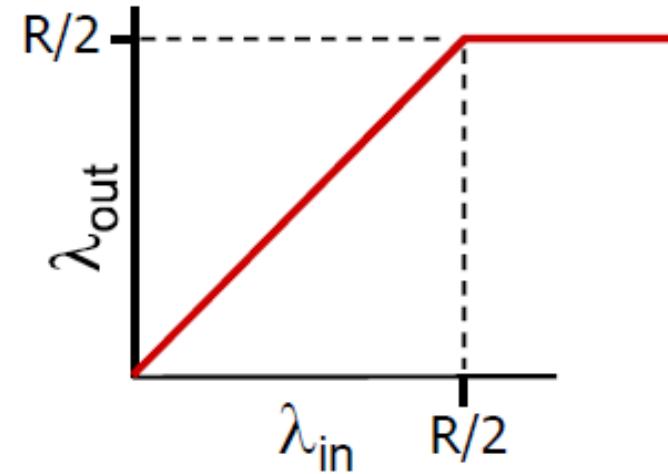
- Nguyên nhân và chi phí của tắc nghẽn
 - Tình huống 1: hai đối tượng gửi, một router với bộ nhớ đệm không giới hạn



Tốc độ lưu lượng tới router: λ_{in} bytes/sec
Tốc độ liên kết ra: R

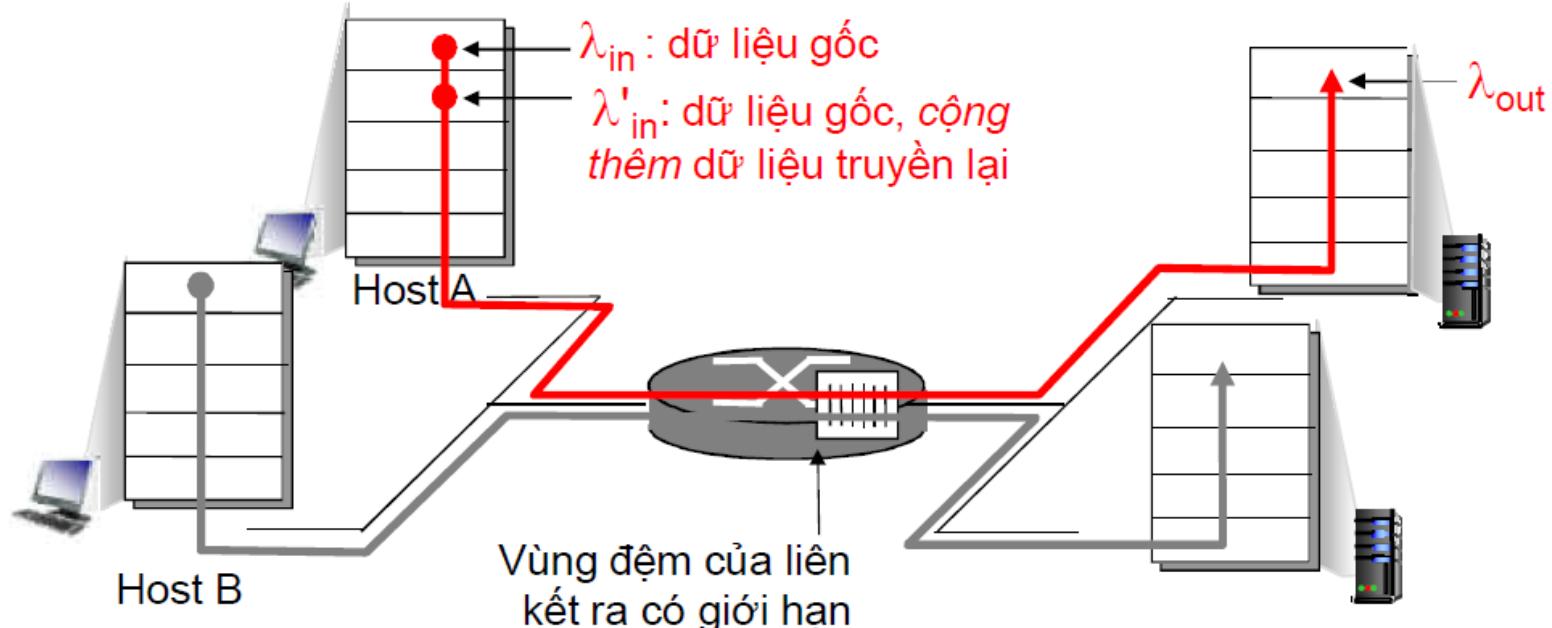
Các nguyên lý điều khiển tắc nghẽn

- Nguyên nhân và chi phí của tắc nghẽn
 - Tình huống 1: hai đối tượng gửi, một router với bộ nhớ đệm không giới hạn
 - **Thông lượng lớn nhất trên mỗi kết nối là $R/2$**
 - **Trễ lớn do tốc độ đến, λ_{in} tiệm cận đến thông lượng tối đa**



Các nguyên lý điều khiển tắc nghẽn

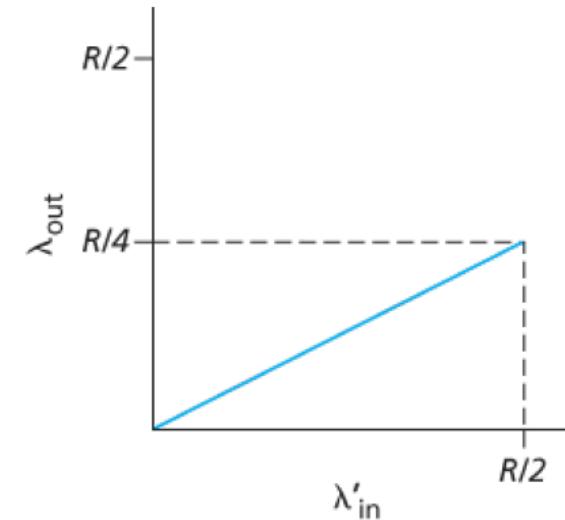
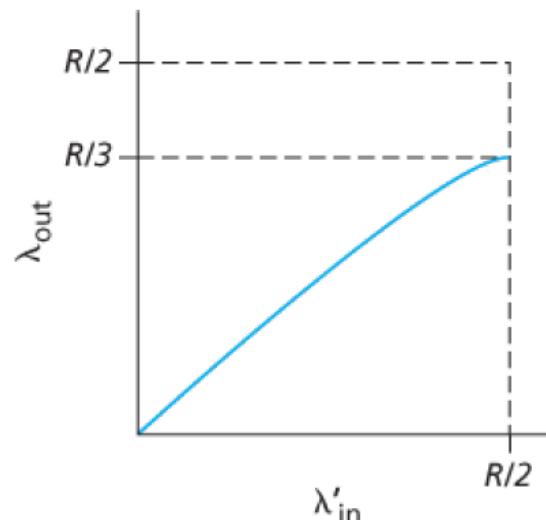
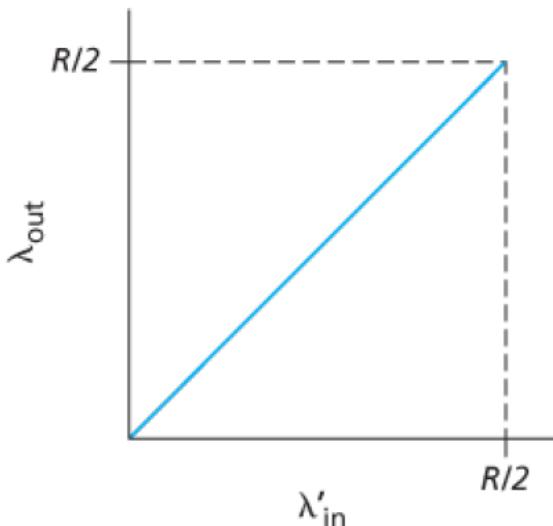
- Nguyên nhân và chi phí của tắc nghẽn
 - Tình huống 2: 2 bên gửi và một router với bộ nhớ hữu hạn



- Đầu vào tầng ứng dụng = đầu ra tầng ứng dụng: $\lambda_{in} = \lambda_{out}$
- Đầu vào tầng giao vận bao gồm việc truyền lại $\lambda'_{in} \geq \lambda_{in}$

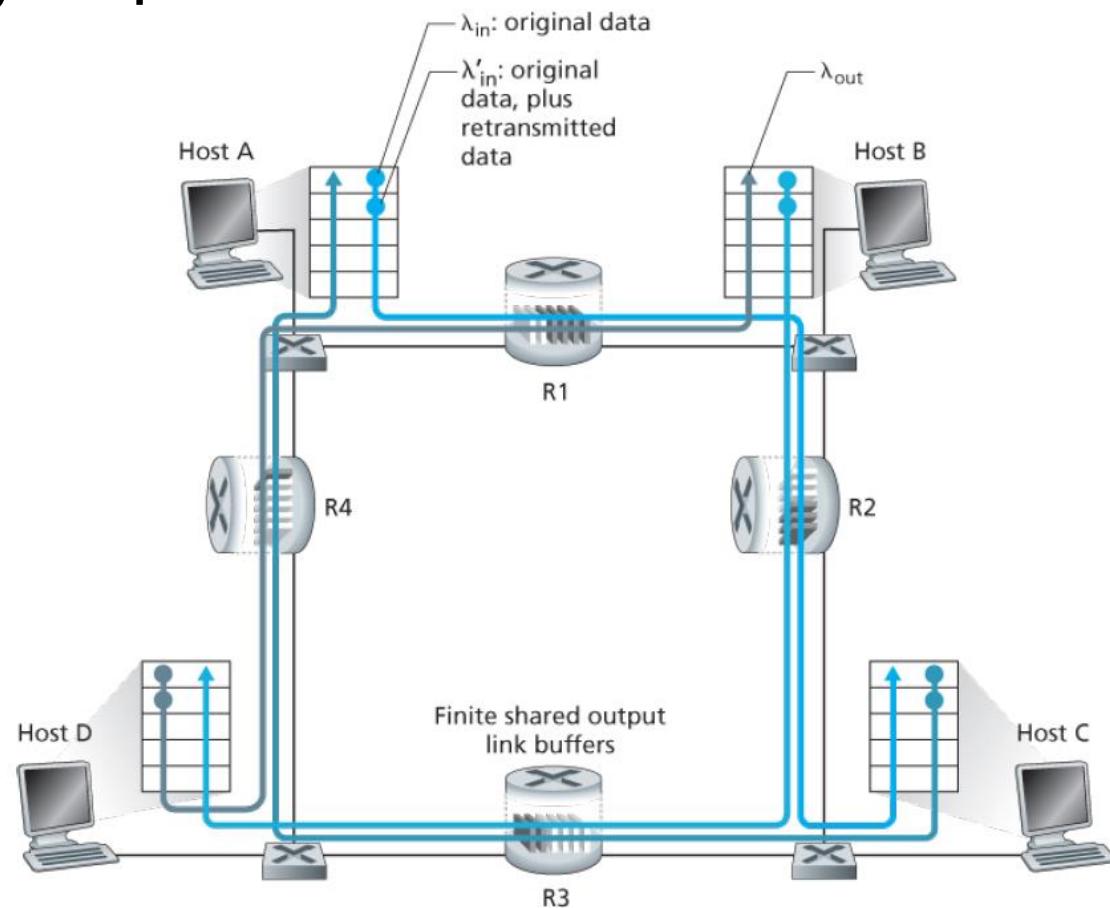
Các nguyên lý điều khiển tắc nghẽn

- Nguyên nhân và 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 lý điều khiển tắc nghẽn

- Nguyên nhân và chi phí của tắc nghẽn
 - Tình huống 3: 4 bên gửi; nhiều đường đến đích; timeout/truyền lại



Các nguyên lý điều khiển tắc nghẽn

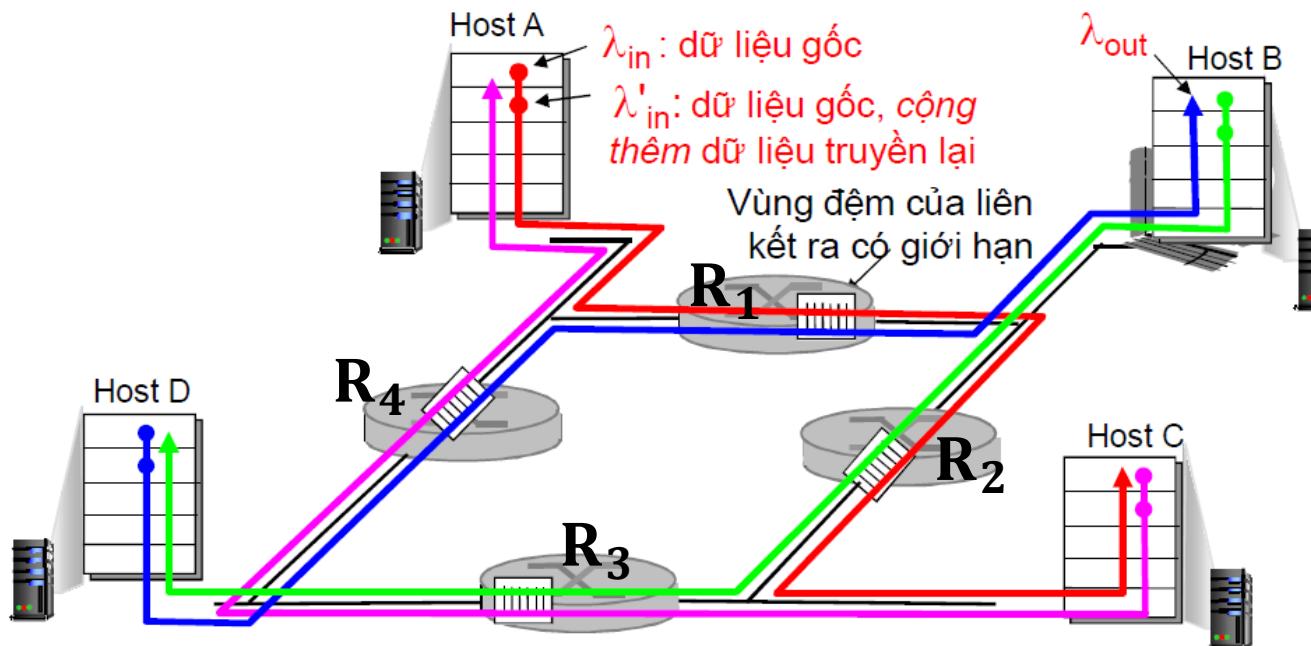
- Nguyên nhân và chi phí của tắc nghẽn

- Tình huống 3:

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

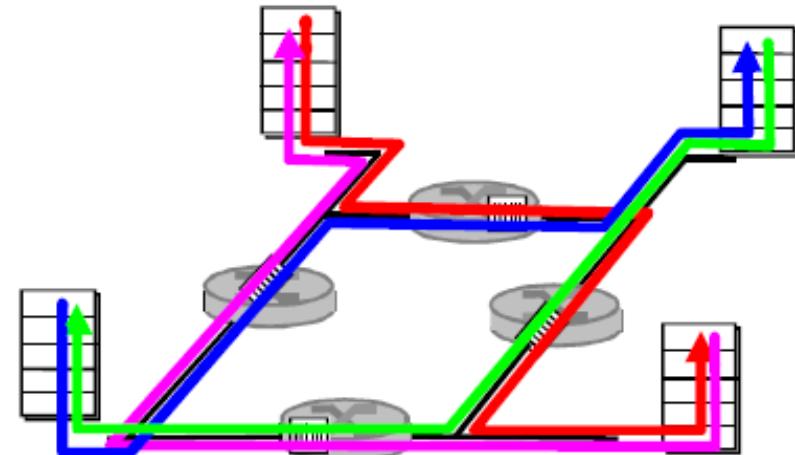
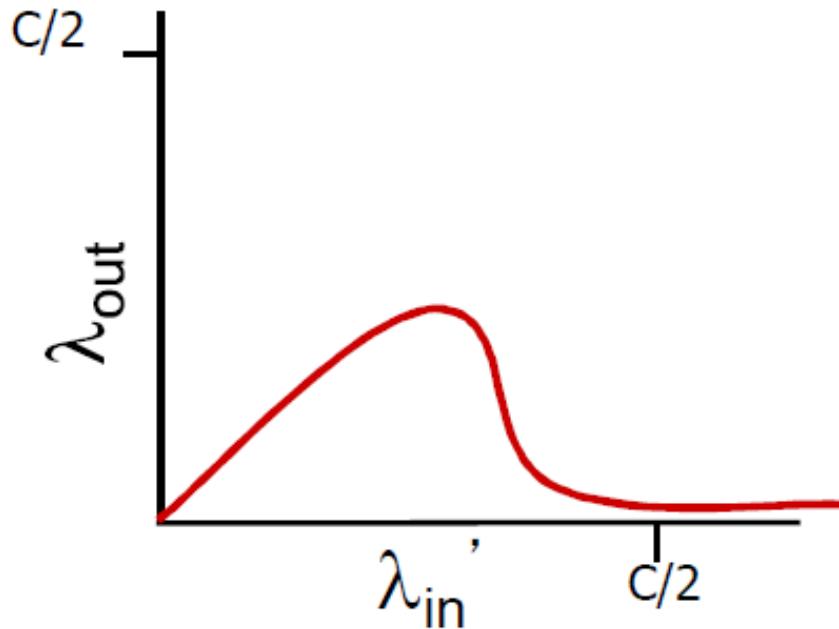
Hỏi: Điều gì sẽ xảy ra khi λ_{in} và λ'_{in} tăng lên?

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 lý điều khiển tắc nghẽn

- Nguyên nhân và 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!



Các nguyên lý điều khiển tắc nghẽn

- Các hướng tiếp cận 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

Các nguyên lý điều khiển tắc nghẽn

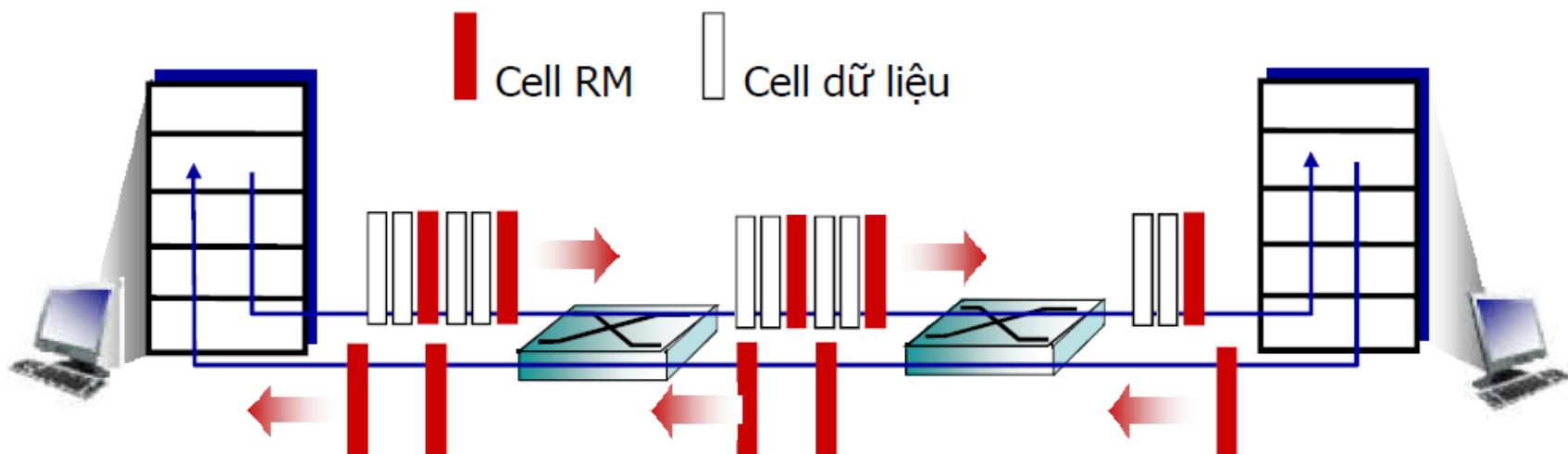
- Điều khiển tắc nghẽn trong ATM ABR
 - ABR (Available Bite Rate): tốc độ bit có sẵn
 - “Dịch vụ mềm dẻo”
 - Nếu đường truyền phía bên gửi “dưới tải” thì
 - Bên gửi nên dùng băng thông có sẵn
 - Nếu đường truyền bên gửi bị tắc nghẽn thì
 - Bên gửi nên giảm để đảm bảo tốc độ là tối thiểu

Các nguyên lý điều khiển tắc nghẽn

- Điều khiển tắc nghẽn trong ATM ABR
 - 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

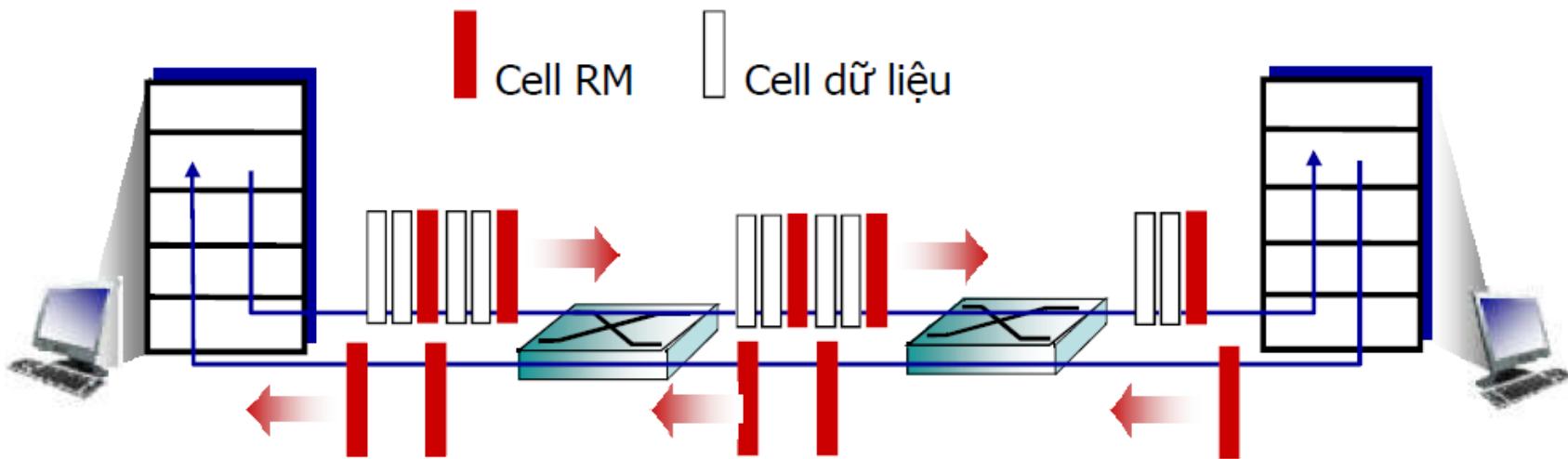
Các nguyên lý điều khiển tắc nghẽn

- 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



Các nguyên lý điều khiển tắc nghẽn

- Điều khiển tắc nghẽn trong ATM ABR
 - 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



Nội dung

- Các dịch vụ tầng giao vận
- Ghép kênh và phân kênh
- Vận chuyển không kết nối: UDP
- Các nguyên tắc truyền dữ liệu tin cậy
- Vận chuyển hướng kết nối: TCP
- Các nguyên lý điều khiển tắc nghẽn
- Điều khiển tắc nghẽn TCP

Điều khiển tắc nghẽn TCP

- Cách tiếp cận
 - Điều chỉnh tốc độ gửi dữ liệu lên kết nối theo mức độ tắc nghẽn mạng
 - Nếu TCP đầu gửi nhận thấy ít tắc nghẽn trên đường từ bên gửi tới bên nhận thì nó sẽ tăng tốc độ gửi. Ngược lại, nếu thấy tắc nghẽn lâu, nó sẽ giảm tốc độ gửi.
 - Đặt ra 3 câu hỏi
 - TCP đầu gửi điều chỉnh tốc độ gửi lên kết nối như thế nào?
 - TCP đầu gửi cảm nhận có tắc nghẽn trên đường truyền như thế nào?
 - Đầu gửi nên sử dụng giải thuật gì để thay đổi tốc độ gửi của nó theo tắc nghẽn đầu cuối cảm nhận được?

Điều khiển tắc nghẽn TCP

- Cách tiếp cận
 - TCP bên gửi điều chỉnh tốc độ gửi dữ liệu?
 - Lượng dữ liệu chưa được ACK ở bên gửi không vượt quá $\min\{\text{cwnd}, \text{rwnd}\}$

LastByteSent – LastByteAcked $\leq \min\{\text{cwnd}, \text{rwnd}\}$

- Giả thiết bộ đệm TCP nhận (RcvBuffer) rất lớn để có thể bỏ qua ràng buộc về cửa sổ nhận rwnd => lượng dữ liệu chưa được ACK ở đầu gửi hoàn toàn bị giới hạn bởi cwnd.
- Giả thiết bên gửi luôn có dữ liệu để gửi \Leftrightarrow tất cả các segment trong cửa sổ tắc nghẽn được gửi

Điều khiển tắc nghẽn TCP

- Cách tiếp cận
 - TCP bên gửi hạn chế tốc độ gửi dữ liệu?
 - Các ràng buộc nêu trên hạn chế lượng dữ liệu chưa được ACK ở đầu gửi, do đó hạn chế tốc độ gửi của người gửi.
 - Xem xét kết nối bở qua trễ truyền và mất mát gói tin => Ở đầu mỗi RTT, cho phép bên gửi gửi các bytes **cwnd** lên kết nối, cuối của RTT bên gửi nhận ACK cho dữ liệu
 - Do đó tốc độ gửi của bên gửi

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

- Bằng cách điều chỉnh giá trị **cwnd**, bên gửi điều chỉnh được tốc độ gửi lên kết nối

Điều khiển tắc nghẽn TCP

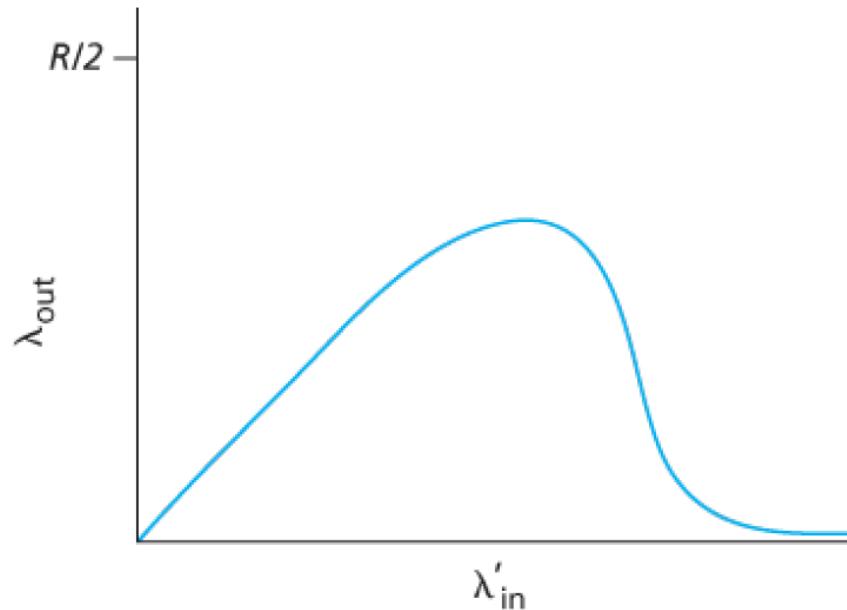
- Cách tiếp cận
 - TCP cảm nhận tắc nghẽn trên đường truyền?
 - Sự kiện mất mát do timeout hoặc nhận 3 ACK trùng nhau từ đầu nhận
 - Khi có tắc nghẽn quá mức, một hoặc nhiều bộ đệm router bị tràn khiến cho datagram bị mất. Datagram bị mất này lại gây ra sự kiện mất mát ở đầu gửi – cả **timeout** hoặc **nhận 3 ACK trùng nhau** – được bên gửi coi là dấu hiệu của tắc nghẽn trên đường dẫn từ đầu gửi tới đầu nhận.

Điều khiển tắc nghẽn TCP

- Cách tiếp cận
 - Phát hiện tắc nghẽn như thế nào?
 - Trường hợp lạc quan hơn: mạng không tắc nghẽn và không có sự kiện mất mát
 - Các ACK cho các segment chưa được ACK trước đó sẽ được nhận ở TCP gửi. Chứng tỏ các segment được truyền lên mạng tới đầu nhận thành công.
 - Sử dụng các ACK này để **tăng kích cỡ cửa sổ tắc nghẽn cwnd, từ đó tăng tốc độ truyền.**
 - Nếu ACK đến với tốc độ chậm thì cwnd cũng tăng chậm, nếu ACK đến với tốc độ cao, cwnd tăng nhanh hơn.
 - **TCP sử dụng các ACK để kích hoạt tăng cwnd** nên nó được gọi là **seft-clocking**

Điều khiển tắc nghẽn TCP

- Cách tiếp cận
 - TCP bên gửi xác định tốc độ nó gửi như thế nào?
 - Nếu TCP bên gửi gửi với tốc độ quá nhanh, nó có thể gây tắc nghẽn mạng, dẫn đến sụp đổ tắc nghẽn như hình:



Điều khiển tắc nghẽn TCP

- Cách tiếp cận
 - TCP bên gửi xác định tốc độ nó gửi như thế nào?
 - Nếu TCP bên gửi gửi với tốc độ quá chậm, dưới mức băng thông mạng. Khi đó, nó có thể gửi với tốc độ cao hơn mà không gây tắc nghẽn mạng.
- ⇒ Câu hỏi:
1. **Tốc độ cao hơn** này là bao nhiêu để vừa tận dụng băng thông mạng mà không gây tắc nghẽn?
 2. Những người gửi TCP có được phối hợp với nhau một cách rõ ràng không, hay có cách tiếp cận phân tán trong đó người gửi TCP có thể đặt tốc độ gửi của họ chỉ dựa trên thông tin cục bộ?

Điều khiển tắc nghẽn TCP

- Cách tiếp cận
 - TCP bên gửi xác định tốc độ nó gửi như thế nào?
 - Các nguyên tắc
 - Segment bị mất có nghĩa là có tắc nghẽn => tốc độ TCP đầu gửi nên giảm khi segment bị mất
 - Một segment ACK cho thấy mạng đang truyền các segment của bên gửi sang cho bên nhận => tốc độ gửi có thể tăng khi ACK đến cho một segment chưa được ACK trước đó.
 - Thăm dò băng thông
 - » Các ACK cho thấy không có tắc nghẽn từ nguồn tới đích và các sự kiện mất mát cho thấy có tắc nghẽn.
 - » **Chiến lược điều chỉnh tốc độ truyền của TCP là tăng tốc độ tương ứng với các ACK đang đến cho đến khi sự kiện mất mát xuất hiện thì giảm tốc độ.**

Điều khiển tắc nghẽn TCP

- Giải thuật điều khiển tắc nghẽn TCP
 - Gồm 3 thành phần
 - Slow start: khởi đầu chậm
 - Congestion avoidance: tránh tắc nghẽn
 - Fast recovery: khôi phục nhanh

Điều khiển tắc nghẽn TCP

- Giải thuật điều khiển tắc nghẽn TCP
 - Slow start: khởi đầu chậm, nhưng tăng theo cấp số nhân
 - Khi tắc nghẽn TCP bắt đầu, giá trị cwnd khởi tạo = 1 MSS (giá trị nhỏ)
=> tốc độ gửi ban đầu $\approx \text{MSS}/\text{RTT}$
 - Ví dụ: nếu MSS = 500 bytes, RTT = 200 msec => tốc độ gửi ban đầu khoảng 20 kbps
 - Do băng thông sẵn có cho TCP bên gửi có thể lớn hơn **MSS/RTT** nên TCP bên gửi muốn tìm lượng băng thông khả dụng một cách nhanh chóng.
 - Do đó ở trạng thái **slow-start**, giá trị cwnd lúc đầu = 1mss, sau đó tăng lên thêm 1 MSS mỗi lần một segment truyền đi được ACK lần đầu tiên.

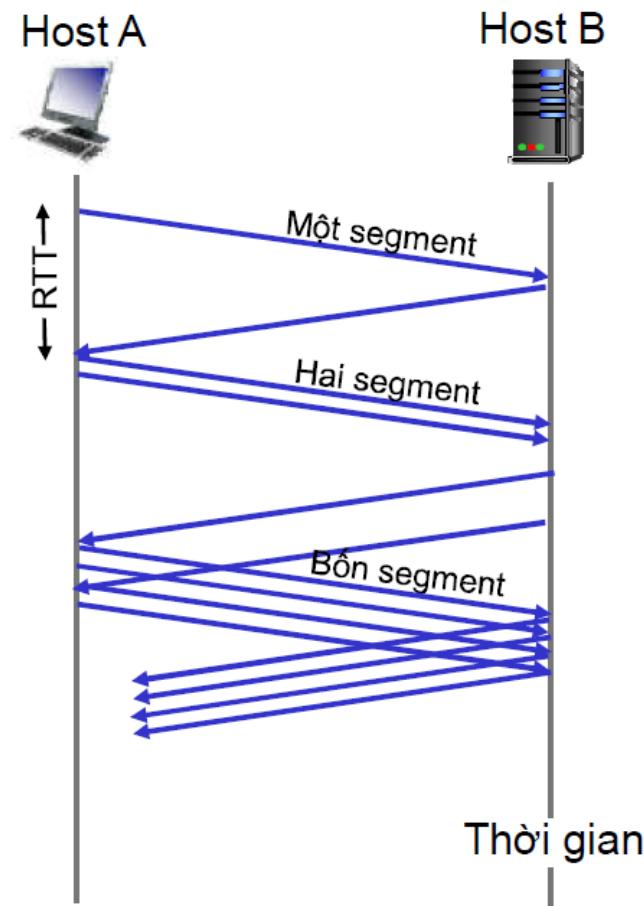
Điều khiển tắc nghẽn TCP

- Giải thuật điều khiển tắc nghẽn TCP

- Slow start: khởi đầu chậm, nhưng tăng theo cấp số nhân

- TCP gửi segment đầu tiên lên mạng và đợi ACK
 - Khi ACK tới, TCP bên gửi tăng cwnd thêm 1 MSS và gửi ra 2 segment có kích thước tối đa. Các segment này sau đó được ACK, với bên gửi tăng cwnd thêm 1 MSS cho từng segment được ACK, tạo ra một **cwnd = 4 MSS**,...

=> Tốc độ gửi TCP bắt đầu thì chậm nhưng tăng theo cấp số nhân trong slow-start.



Điều khiển tắc nghẽn TCP

- Giải thuật điều khiển tắc nghẽn TCP
 - Slow start: khởi đầu chậm, nhưng tăng theo cấp số nhân. Tăng đến bao giờ thì kết thúc?
 - Cách 1
 - » Nếu có sự kiện loss (tắc nghẽn) được chỉ ra bởi timeout, TCP bên gửi thiết lập cwnd = 1 và bắt đầu tiến trình slow start mới.
 - » TCP bên gửi cũng thiết lập giá trị của biến trạng thái thứ 2 **ssthresh (slow start threshold) = cwnd/2**

Điều khiển tắc nghẽn TCP

- Giải thuật điều khiển tắc nghẽn TCP
 - Slow start: khởi đầu chậm, nhưng tăng theo cấp số cộng. Tăng đến bao giờ thì kết thúc?
 - Cách 2
 - » Kết thúc trực tiếp bằng **ssthresh**.
 - » Do $ssthresh = \frac{1}{2} cwnd$ khi lần cuối tắc nghẽn được phát hiện. Có thể hơi liều lĩnh nếu tiếp tục nhân đôi $cwnd$ khi nó đạt hoặc vượt qua giá trị của **ssthresh**.
 - » Do đó, khi giá trị của **cwnd = ssthresh**, slow start kết thúc và TCP chuyển sang chế độ tránh tắc nghẽn. TCP tăng $cwnd$ cẩn trọng hơn khi ở chế độ tránh tắc nghẽn.

Điều khiển tắc nghẽn TCP

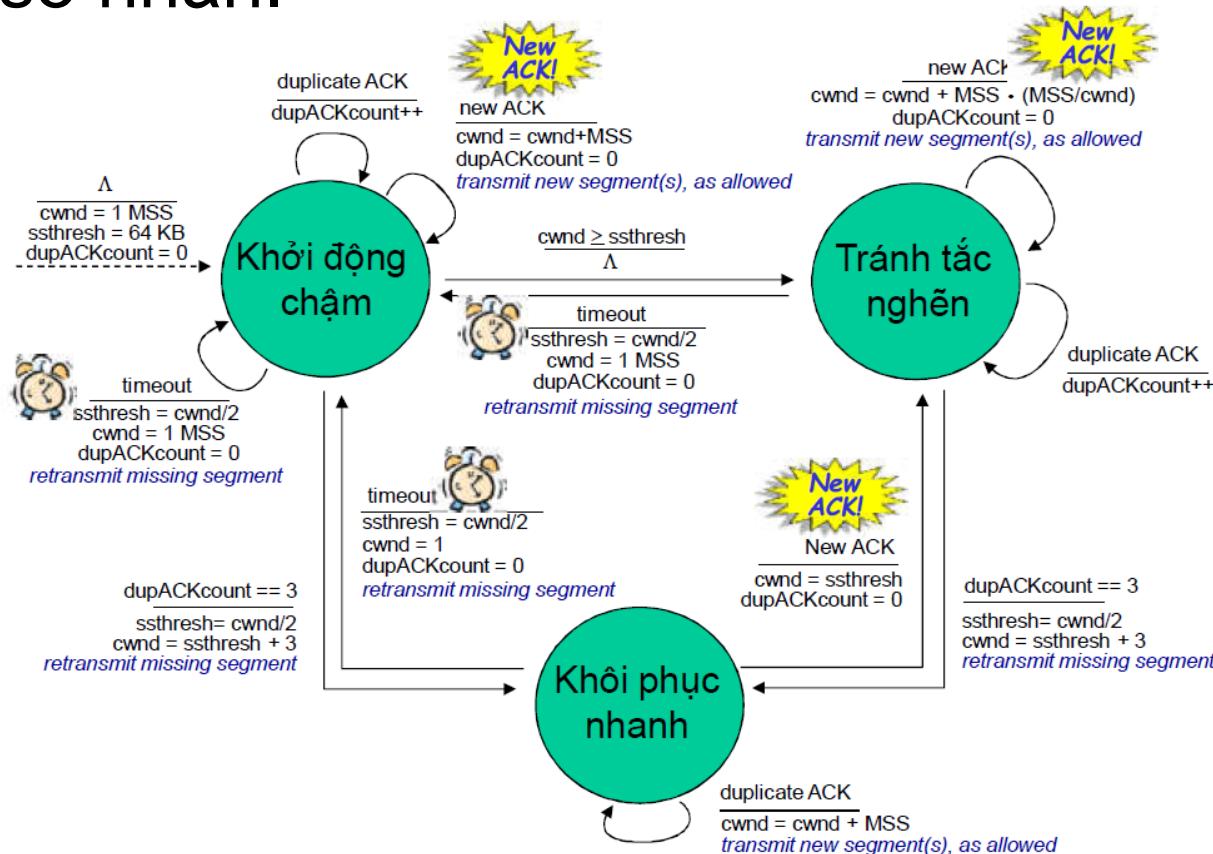
- Giải thuật điều khiển tắc nghẽn TCP
 - Slow start: khởi đầu chậm, nhưng tăng theo cấp số cộng. Tăng đến bao giờ thì kết thúc?
 - Cách 3
 - » Kết thúc khi phát hiện 3 ACK trùng lặp. Trong trường hợp này, TCP thực hiện truyền lại nhanh (fast retransmit) và chuyển sang trạng thái khôi phục nhanh (fast recovery).

Điều khiển tắc nghẽn TCP

- Tổng kết điều khiển tắc nghẽn trong TCP
 - Khi **cwnd < ssthresh**, bên gửi đang trong giai đoạn slow start, kích thước cửa sổ tăng nhanh theo cấp số nhân.
 - Khi **cwnd > ssthresh**, bên gửi đang trong giai đoạn tránh tắc nghẽn, kích thước cửa sổ tăng nhanh 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**

Điều khiển tắc nghẽn TCP

- Giải thuật điều khiển tắc nghẽn TCP
 - Slow start: khởi đầu chậm, nhưng tăng theo cấp số nhân.



Điều khiển tắc nghẽn TCP

- Tránh tắc nghẽn
 - Tăng giá trị **cwnd** = 1 MSS mỗi RTT [RFC 5681]. Điều này có thể được thực hiện bằng một số cách
 - Cách thông dụng nhất: TCP bên gửi tăng cwnd lên MSS bytes bất kỳ khi nào một ACK đến.
 - Ví dụ nếu MSS = 1.460 bytes và cwnd = 14.600 bytes => có 10 segments được gửi trong 1 RTT.
 - Mỗi ACK đến (giả sử 1 ACK/segment) sẽ tăng cwnd thêm 1/10 MSS, do đó, giá trị cwnd sẽ tăng thêm 1 MSS sau khi tất cả 10 segments được nhận.

Điều khiển tắc nghẽn TCP

- Tránh tắc nghẽn
 - Khi nào việc tăng tuyển tính của tránh tắc nghẽn kết thúc?
 - Giải thuật tránh tắc nghẽn của TCP giống nhau khi timeout xuất hiện. Như trong trường hợp của slow start: giá trị **cwnd = 1 MSS**, và giá trị **ssthresh = $\frac{1}{2}$ cwnd** khi sự kiện mất mát xuất hiện.

Điều khiển tắc nghẽn TCP

- Tránh tắc nghẽn
 - Khi nào việc tăng tuyến tính của tránh tắc nghẽn kết thúc?
 - Sự kiện mất mát cũng có thể được kích hoạt bởi sự kiện 3 ACK trùng lặp. Trong trường hợp này, mạng đang tiếp tục chuyển các segments từ bên gửi sang bên nhận.
 - Do đó, hành vi của TCP đối với loại sự kiện mất mát này sẽ ít nghiêm trọng hơn so với mất mát với timeout: TCP **giảm $\frac{1}{2}$ cwnd** và **ssthresh = $\frac{1}{2}$ cwnd** khi nhận được 3 ACK trùng lặp.
 - Sau đó vào trạng thái phục hồi nhanh

Điều khiển tắc nghẽn TCP

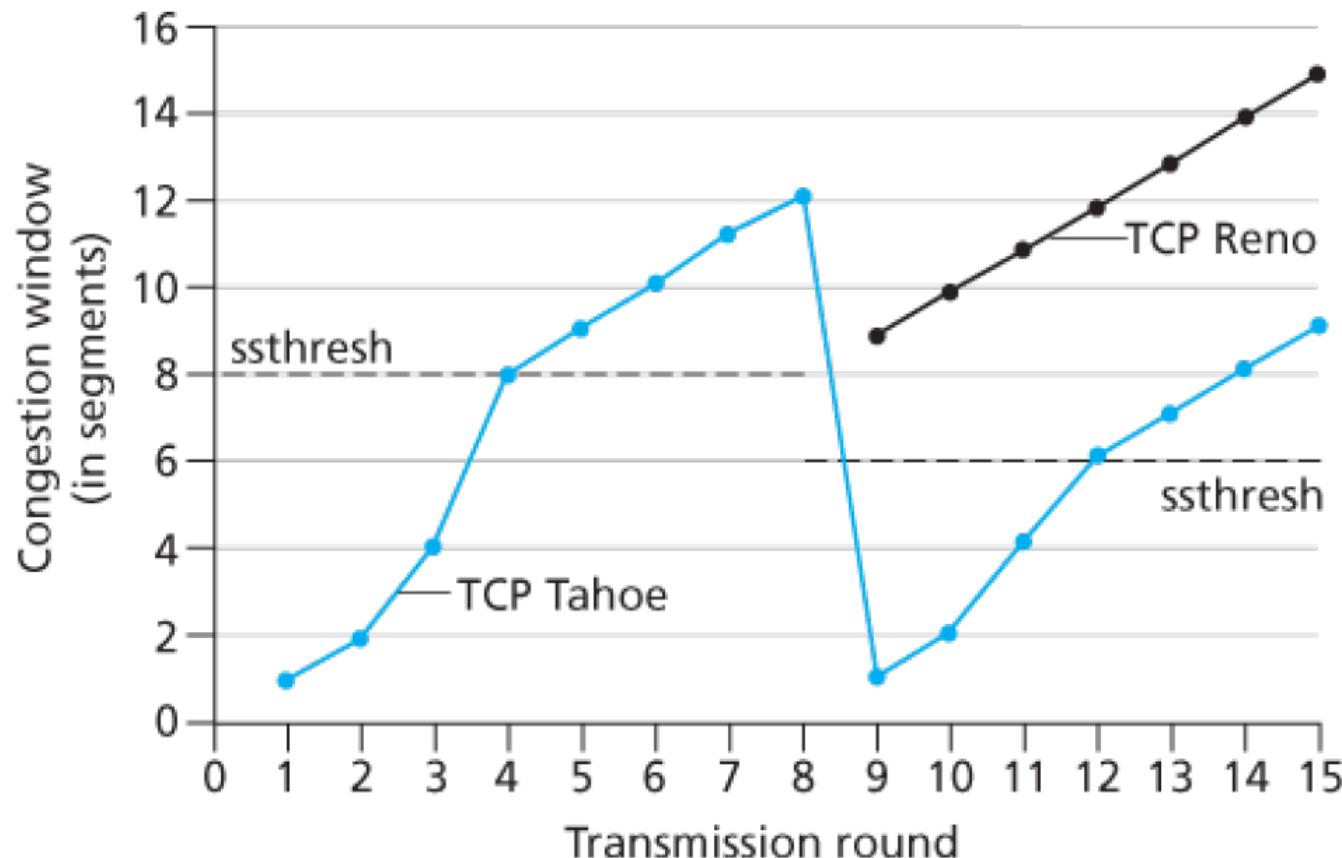
- Phục hồi nhanh
 - Trong phục hồi nhanh, **cwnd** được **tăng lên 1 MSS** cho mỗi ACK trùng lặp nhận được cho segment bị mất khiến cho TCP chuyển sang trạng thái phục hồi nhanh.
 - Cuối cùng, khi một ACK cho segment bị mất đến, TCP vào trạng thái tránh tắc nghẽn sau khi giảm cwnd.

Điều khiển tắc nghẽn TCP

- Phục hồi nhanh
 - Nếu sự kiện timeout xuất hiện, trạng thái phục hồi nhanh chuyển sang trạng thái slow-start sau khi thực hiện các hành động tương tự như trong slow start và tránh tắc nghẽn: giá trị cwnd = 1 MSS, và giá trị ssthresh = $\frac{1}{2}$ cwnd khi sự kiện mất mát xuất hiện
 - Phục hồi nhanh là một thành phần được khuyến nghị nhưng không yêu cầu của TCP
 - Phiên bản TCP thời kỳ đầu (**TCP Tahoe**) có cwnd = 1 MSS và chuyển sang pha slow-start sau khi sự kiện mất mát xảy ra khi timeout hay 3 ACK trùng lặp.
 - Phiên bản TCP mới hơn (TCP Reno) tích hợp phục hồi nhanh

Điều khiển tắc nghẽn TCP

- Phục hồi nhanh

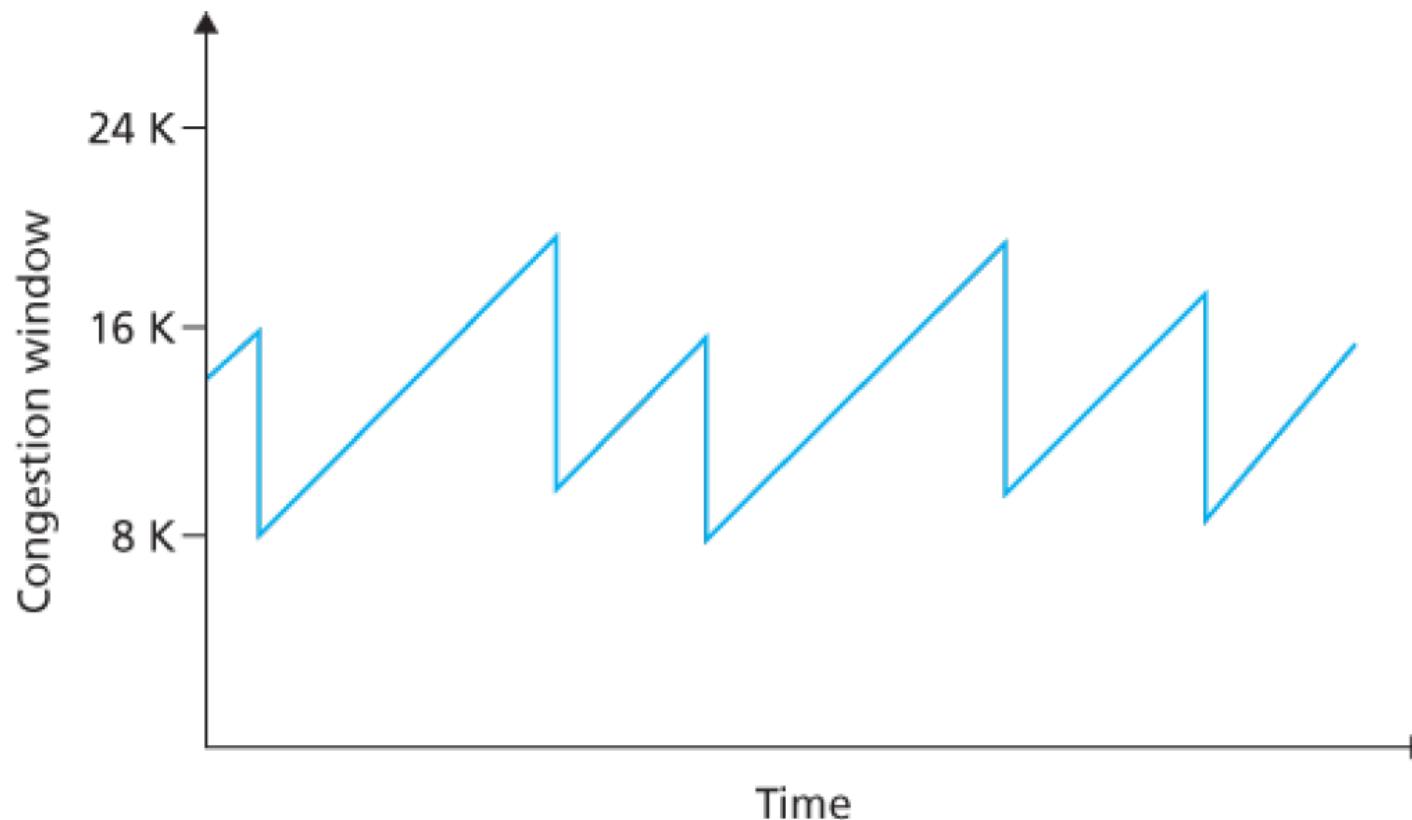


Điều khiển tắc nghẽn TCP

- Kiểm soát tắc nghẽn: tóm tắt lại
 - Bỏ qua giai đoạn slow-start ban đầu khi kết nối bắt đầu và giả sử các mất mát xảy ra với 3 ACK trùng lặp hơn là timeouts, kiểm soát tắc nghẽn của TCP gồm tăng tuyến tính theo $cwnd = 1 \text{ MSS}$ trên từng RTT và sau đó giảm $\frac{1}{2} cwnd$ với sự kiện 3 ACK trùng nhau.
 - Do đó, kiểm soát tắc nghẽn TCP còn được coi như AIMD (additive-increase, multiplicative-decrease) của kiểm soát tắc nghẽn

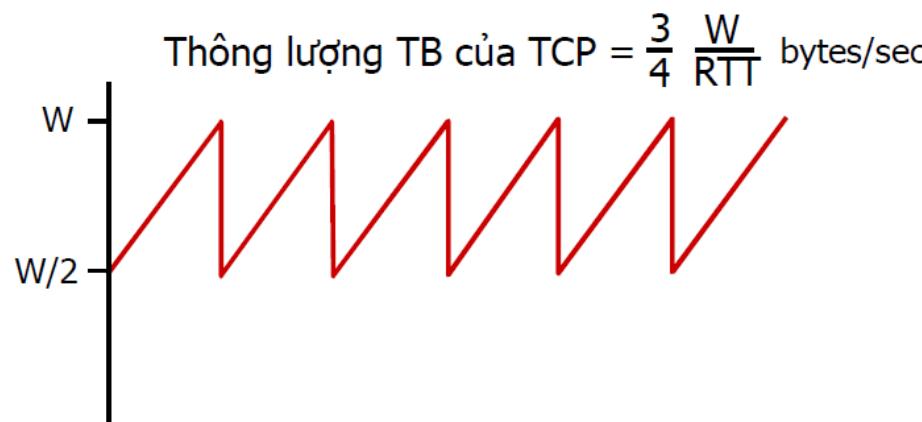
Điều khiển tắc nghẽn TCP

- Kiểm soát tắc nghẽn: tóm tắt lại



Điều khiển tắc nghẽn 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 slow-start, 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 $\frac{3}{4} W$
 - Thông lượng trung bình là $\frac{3}{4} W/RTT$



Điều khiển tắc nghẽn TCP

- TCP trong tương lai: TCP qua “đường truyền rộng và dài”
 - Ví dụ: Các segment dài 1500 byte, RTT=100ms, muốn đạt được thông lượng là 10 Gbps
 - Yêu cầu lưu lượng với kích thước cửa sổ là $W = 83,333$ segment
 - Thông lượng của xác suất mất segment là L

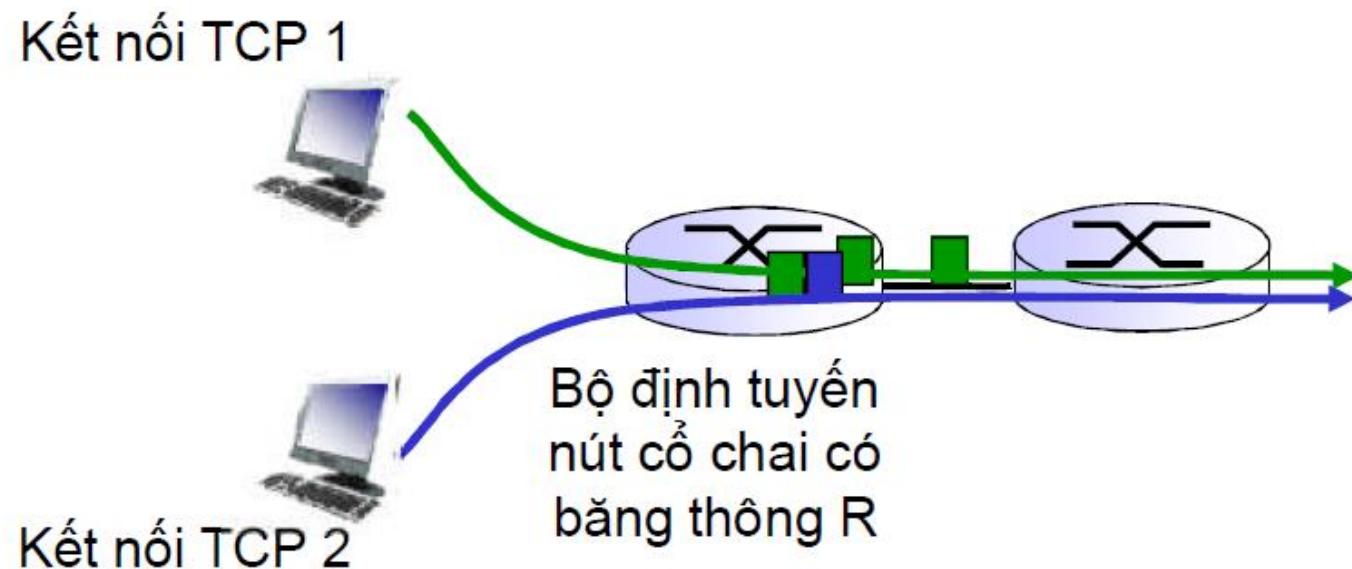
$$\text{Thông lượng TCP} = \frac{1.22 \cdot \text{MSS}}{\text{RTT} \sqrt{L}}$$

⇒ Để có được thông lượng 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 truyền tốc độ cao

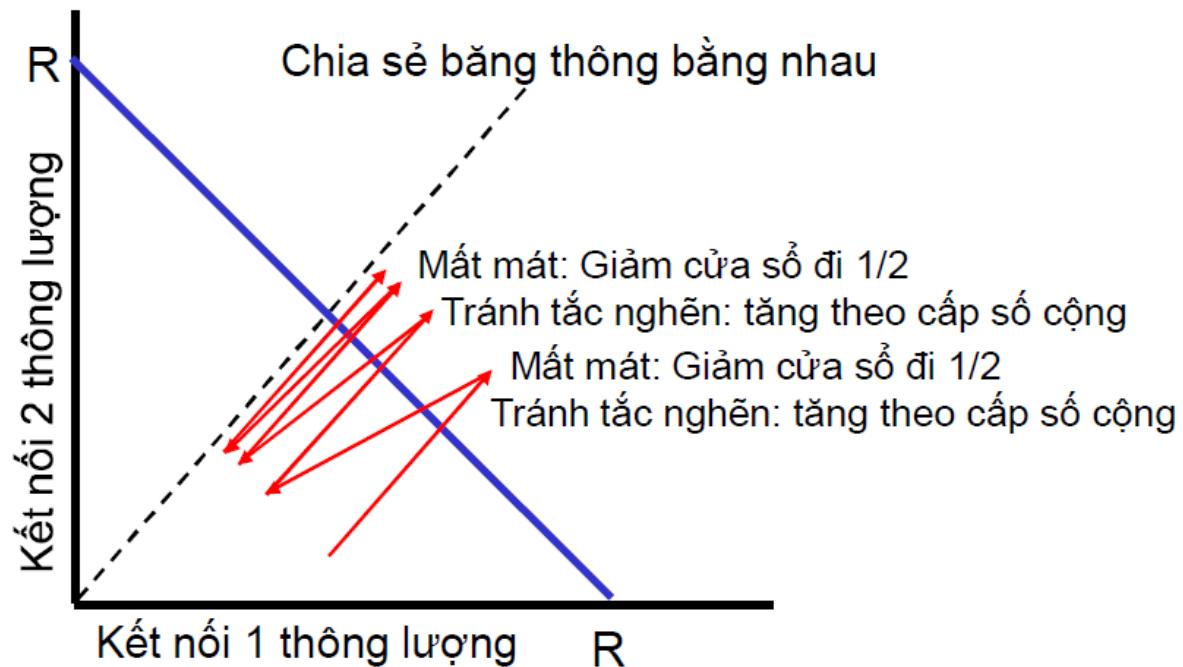
Điều khiển tắc nghẽn TCP

- 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



Điều khiển tắc nghẽn TCP

- Tại sao TCP 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



Điều khiển tắc nghẽn TCP

- 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

Điều khiển tắc nghẽn TCP

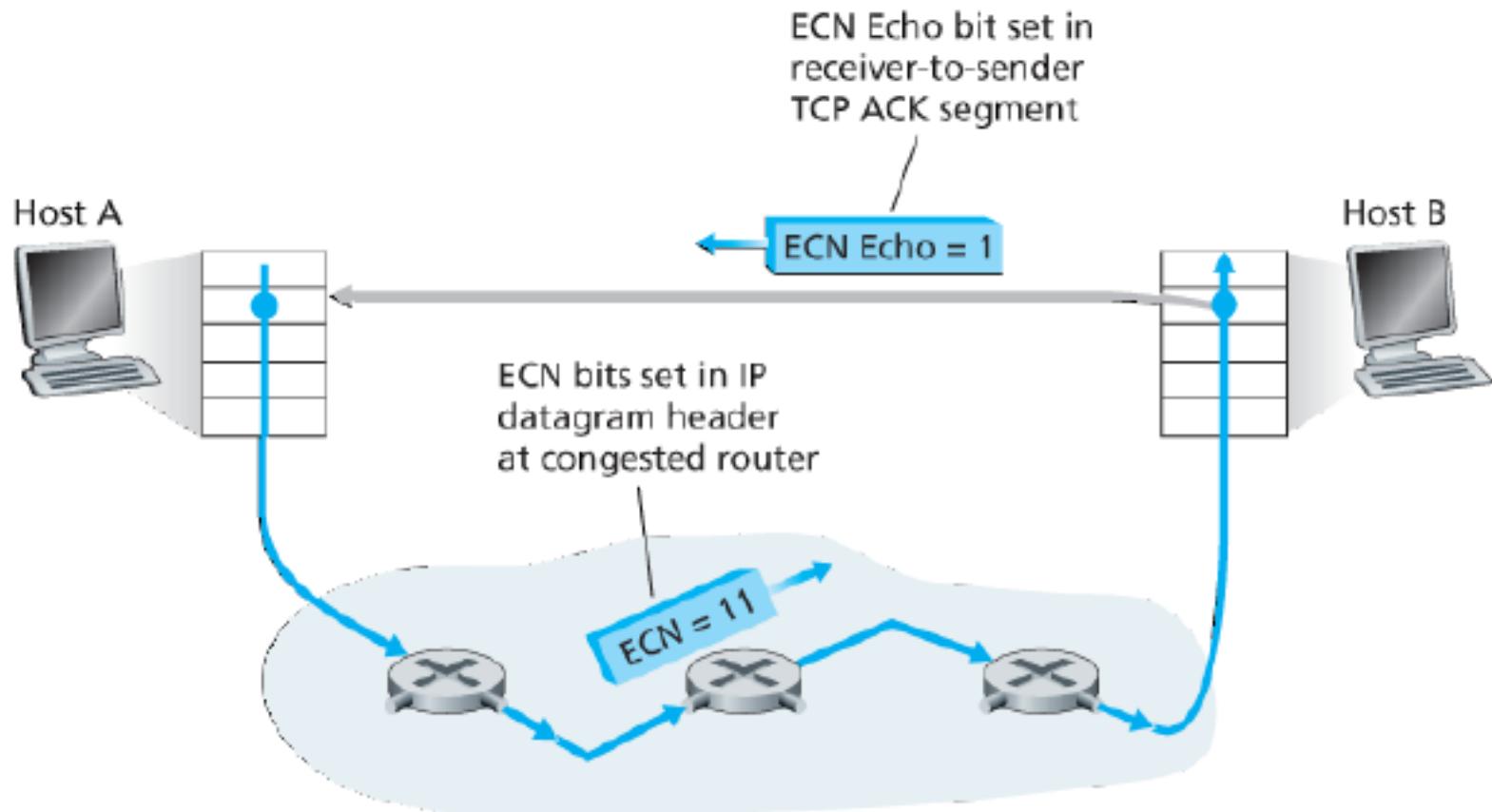
- Tính công bằng và kết nối song song trong TCP
 - Ứng dụng có thể mở nhiều kết nối song song giữa hai host
 - Các trình duyệt web làm theo cách này
 - Ví dụ: liên kết có băng thông R hỗ trợ 9 kết nối:
 - Ứng dụng mới yêu cầu 1 TCP, có băng thông R/10
 - Ứng dụng mới yêu cầu 11 TCP, có băng thông R/2

Điều khiển tắc nghẽn TCP

- Thông báo tắc nghẽn rõ ràng (ECN-Explicit Congestion Notification): Kiểm soát tắc nghẽn do mạng hỗ trợ
 - 2 bit trong trường Type of Service của gói tin IP (tầng mạng)

Điều khiển tắc nghẽn TCP

- Thông báo tắc nghẽn rõ ràng



DCTCP (Data Center TCP)