

Redes de Computadores

Camada de Transporte

Nota: a maioria dos slides dessa apresentação são traduzidos ou adaptados dos slides disponibilizados gratuitamente pelos autores do livro [KUROSE, James F. e ROSS, Keith W. *Computer Networking: A Top-Down Approach*. 8th Edition](#). Pearson, 2020. Todo o material pertencente aos seus respectivos autores está protegido por direito autoral.

Camada de Transporte: visão geral

Nossos objetivos:

- entender os princípios por trás dos serviços da camada de transporte:
 - multiplexação, demultiplexação
 - transferência de dados confiável
 - controle de fluxo
 - controle de congestionamento
- aprender sobre protocolos da camada de transporte da Internet:
 - UDP: transporte sem conexão
 - TCP: transporte confiável orientado a conexão
 - controle de congestionamento do TCP

Camada de Transporte: roteiro

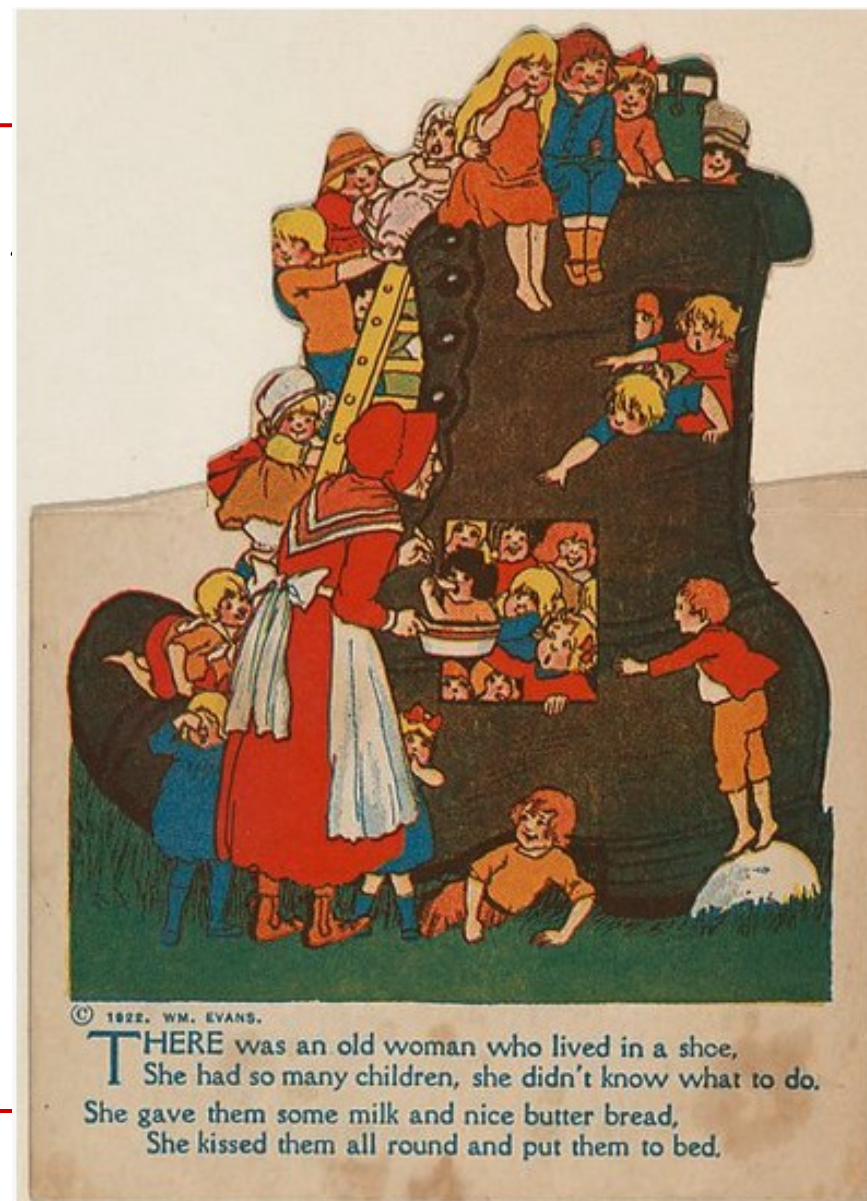
Serviços da camada de transporte

- Multiplexação e demultiplexação
- Transporte sem conexão: UDP
- Princípios da transferência confiável de dados
- Transporte orientado a conexão: TCP
- Princípios de controle de congestionamentos
- Controle de congestionamento do TCP
- Evolução da funcionalidade da camada de transporte



Serviços e protocolos: transporte versus rede

- **camada de transporte:**
comunicação entre *processos*
 - depende de e melhora os serviços da camada de rede
- **camada de rede:**
comunicação entre *hospedeiros*



Serviços e protocolos: transporte versus rede

- **camada de transporte:**
comunicação entre *processos*
 - depende de e melhora os serviços da camada de rede
- **camada de rede:**
comunicação entre *hospedeiros*

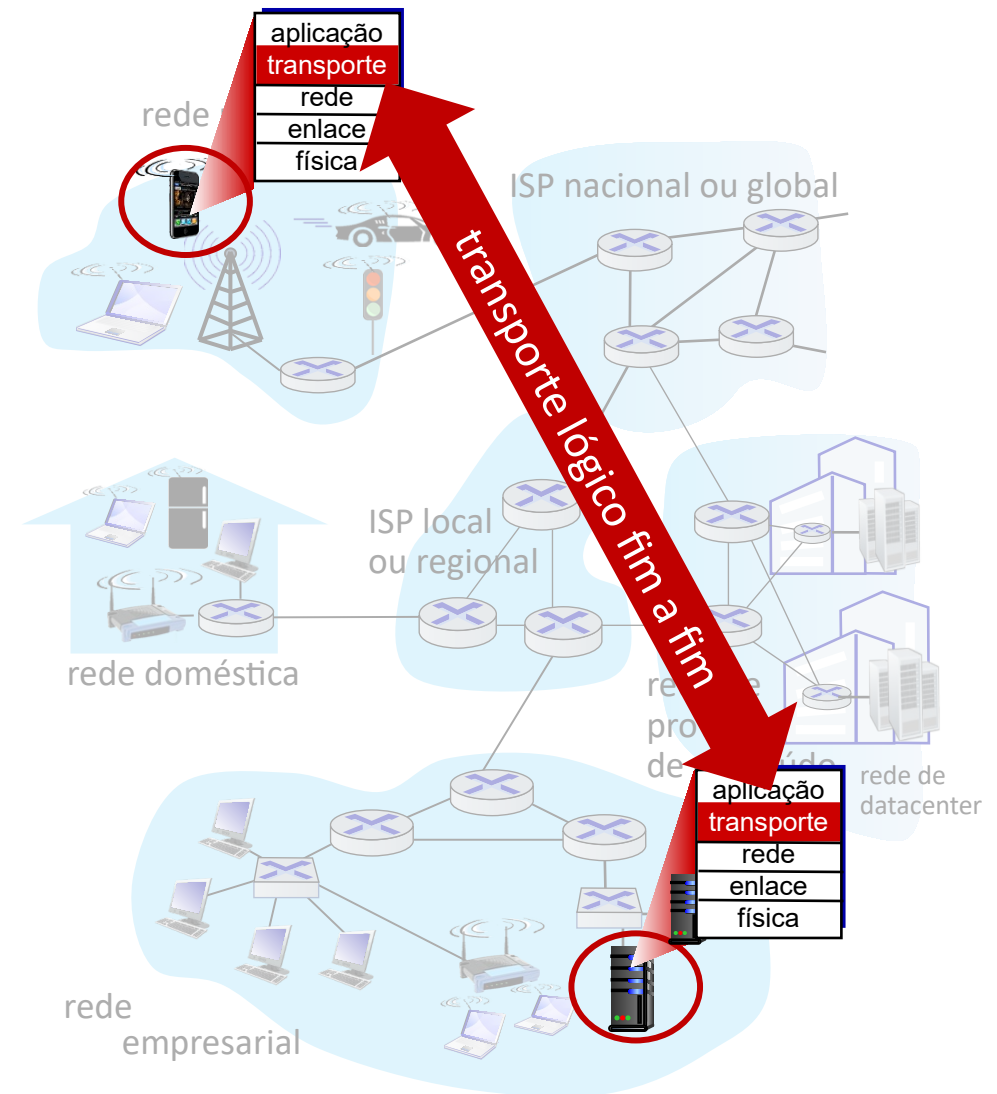
analogia da família:

12 crianças na casa de Ana enviando cartas para 12 crianças na casa de Bill:

- hospedeiros = casas
- processos = crianças
- mensagens de aplicação = cartas em envelopes
- **protocolo de transporte** = Ann e Bill que demultiplexam para irmãos em casa
- protocolo da camada de rede = serviço postal

Serviços e protocolos de transporte

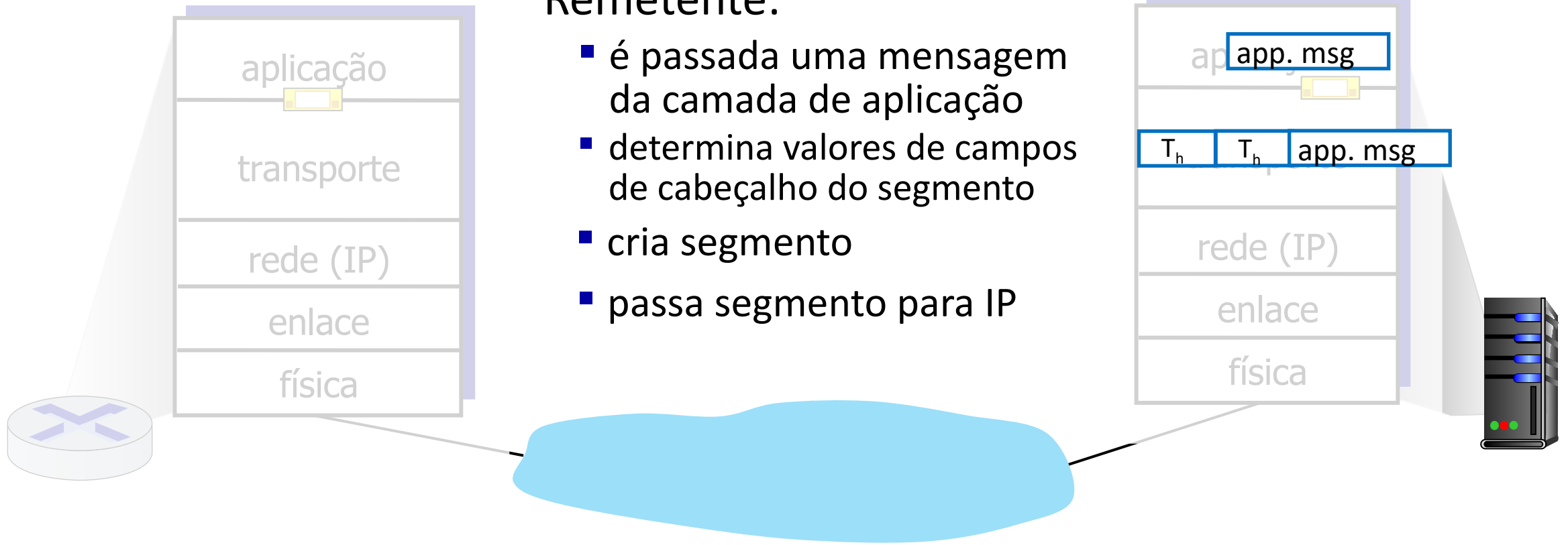
- fornece *comunicação lógica* entre processos de aplicação em execução em diferentes hospedeiros
- ações de protocolos de transporte em sistemas finais:
 - remetente: quebra mensagens de aplicação em *segmentos* e os passa para a camada de rede
 - destinatário: remonta segmentos em mensagens e as passa para a camada de aplicação
- dois protocolos de transporte disponíveis para aplicações na Internet
 - TCP, UDP



Ações da Camada de Transporte

Remetente:

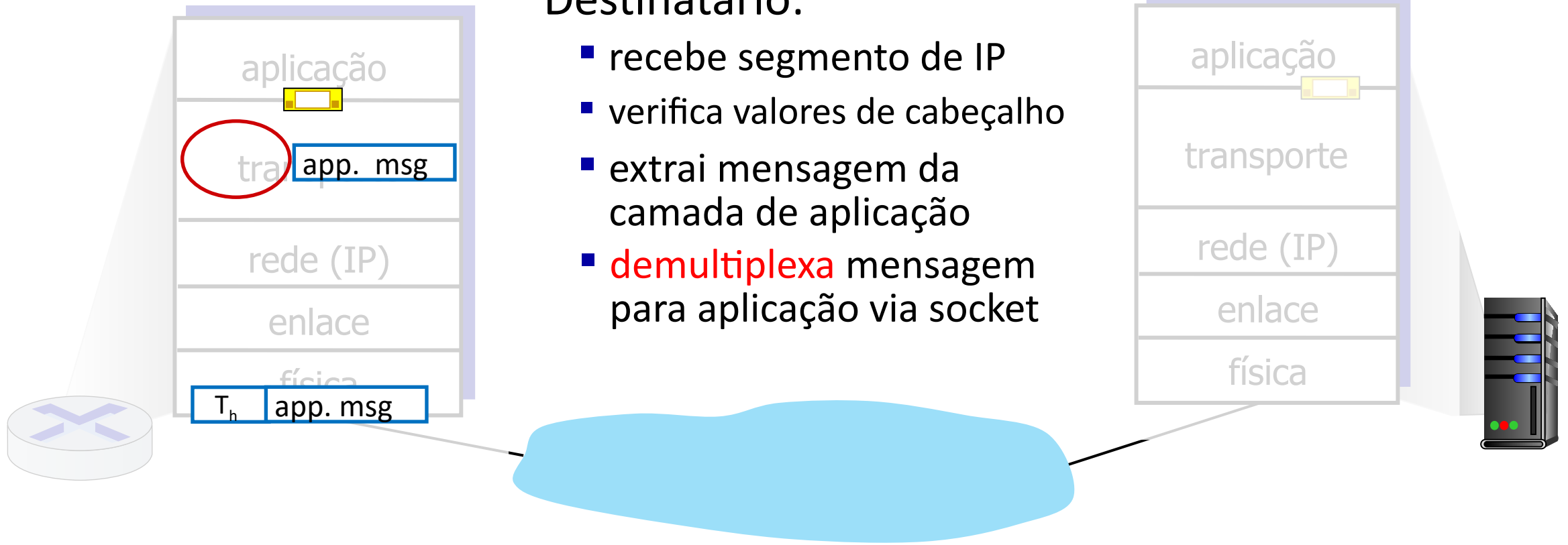
- é passada uma mensagem da camada de aplicação
- determina valores de campos de cabeçalho do segmento
- cria segmento
- passa segmento para IP



Ações da Camada de Transporte

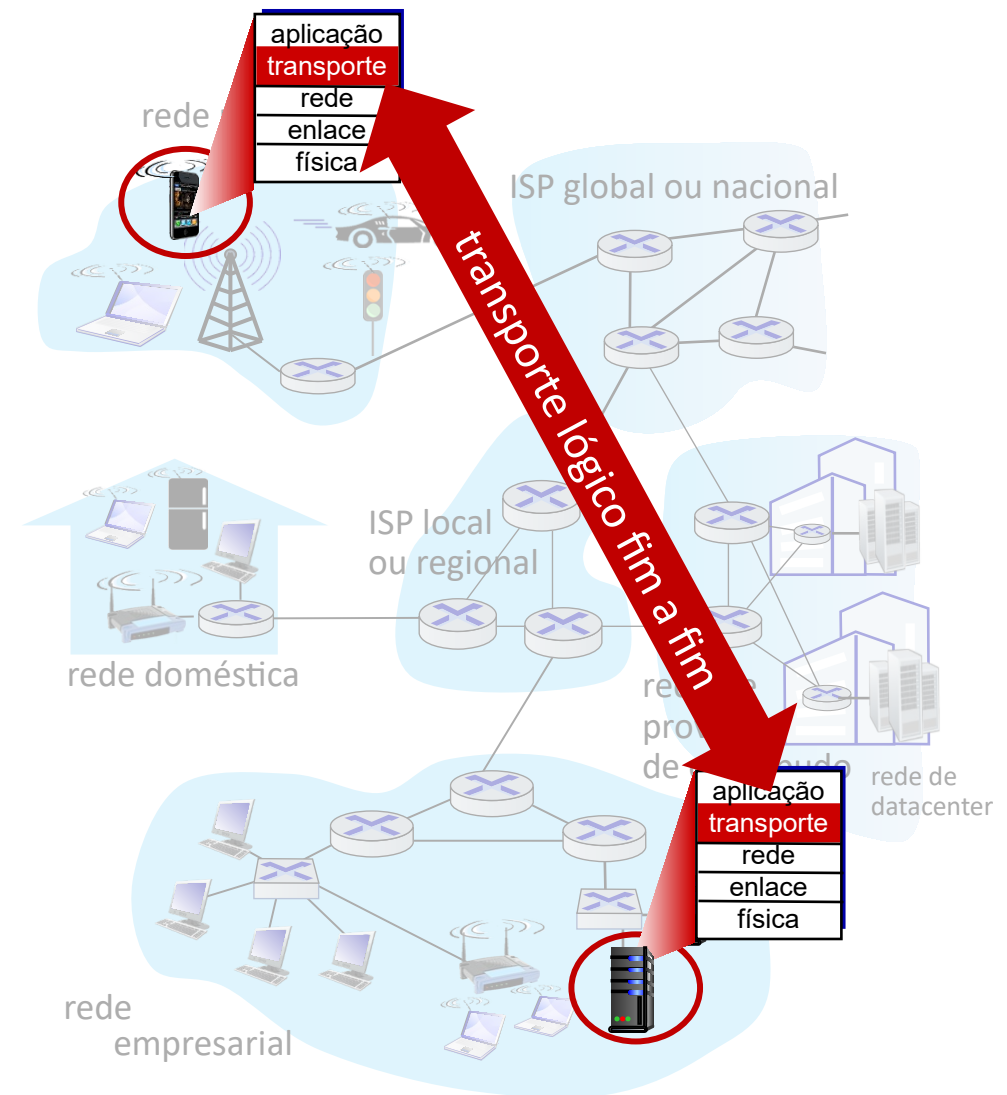
Destinatário:

- recebe segmento de IP
- verifica valores de cabeçalho
- extrai mensagem da camada de aplicação
- **demultiplexa** mensagem para aplicação via socket



Dois principais protocolos de transporte da Internet

- **TCP:** Transmission Control Protocol
 - entrega confiável e em ordem
 - controle de congestionamento
 - controle de fluxo
 - configuração de conexão
- **UDP:** User Datagram Protocol
 - entrega não confiável e não ordenada
 - extensão simples do “melhor esforço” do IP
- serviços não disponíveis:
 - garantias de atraso
 - garantias de largura de banda



Camada de transporte: roteiro

- Serviços da camada de transporte
- Multiplexação e demultiplexação
- Transporte sem conexão: UDP
- Princípios da transferência confiável de dados
- Transporte orientado a conexão: TCP
- Princípios de controle de congestionamentos
- Controle de congestionamento do TCP
- Evolução da funcionalidade de camada de transporte



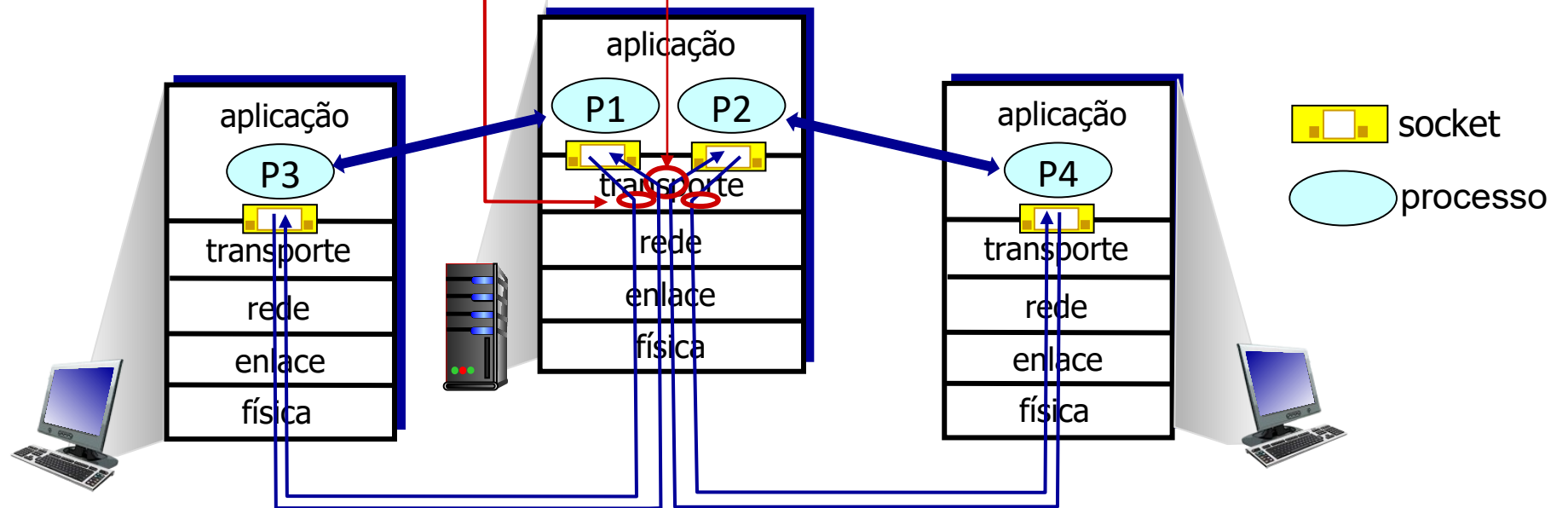
Multiplexação / demultiplexação

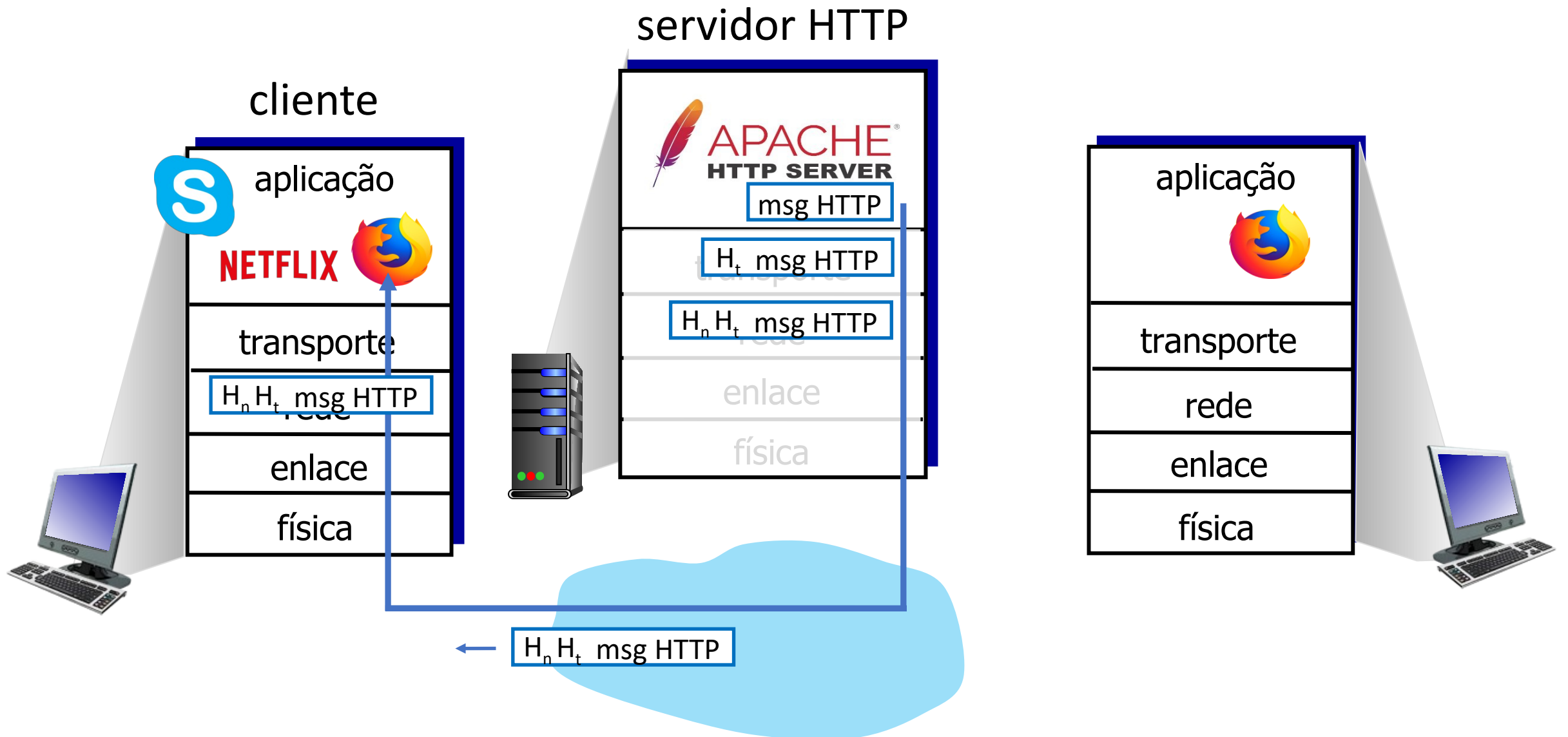
multiplexação no remetente:

manipula dados de vários sockets, adiciona cabeçalho de transporte (posteriormente usado para demultiplexação)

demultiplexação no destinatário:

usa informações de cabeçalho para entregar segmentos recebidos para o socket correto

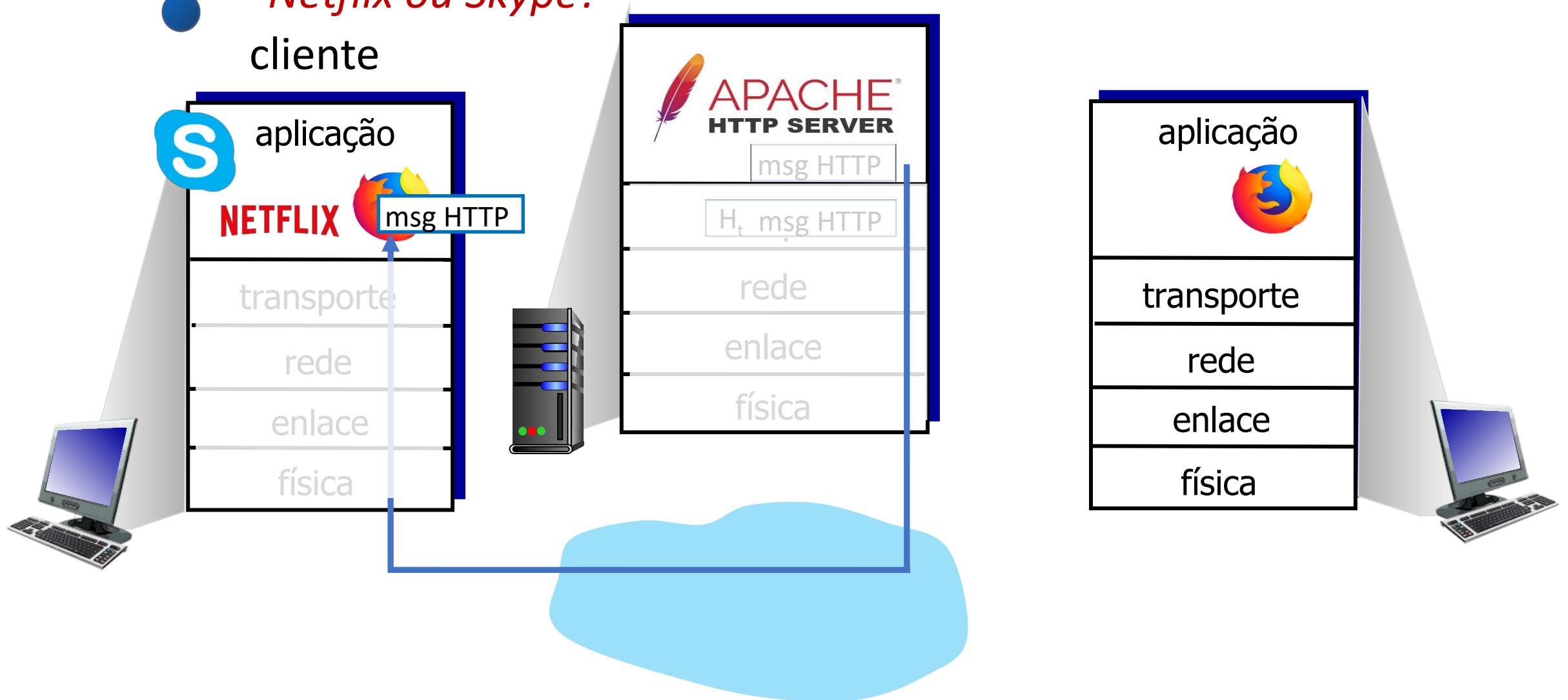


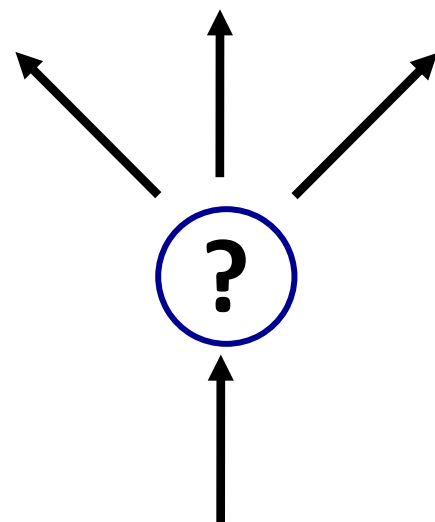




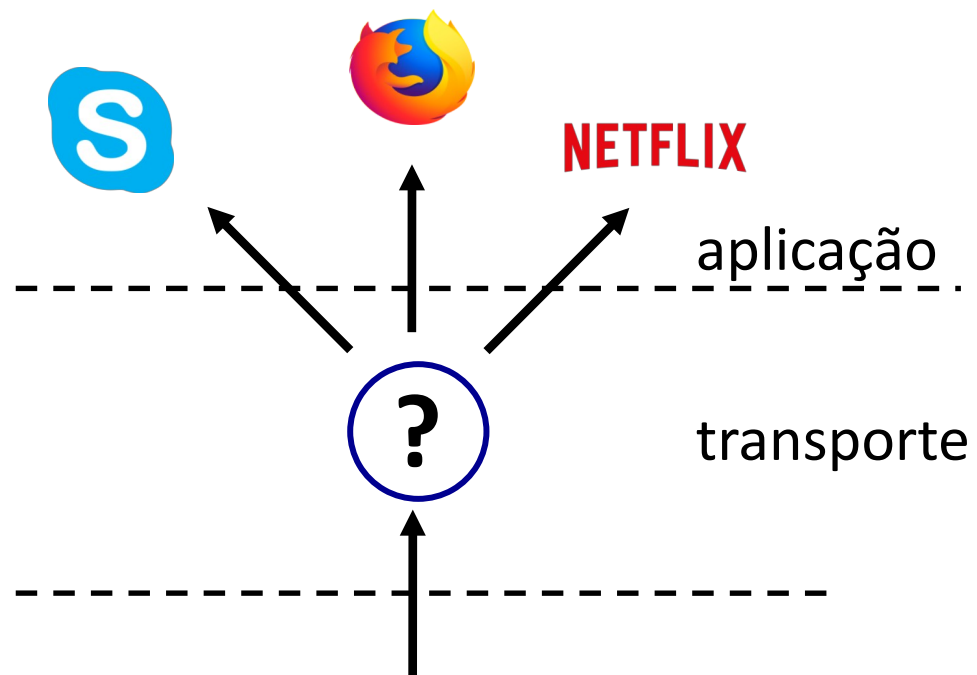
Q: como a camada de transporte sabe que deve entregar a mensagem ao processo do navegador Firefox em vez do processo Netflix ou Skype?

cliente





de-multiplexação



de-multiplexação



Demultiplexação

AIRFRANCE 

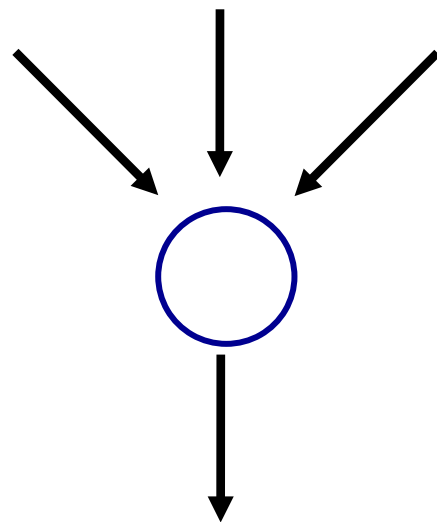
ECONOMY 



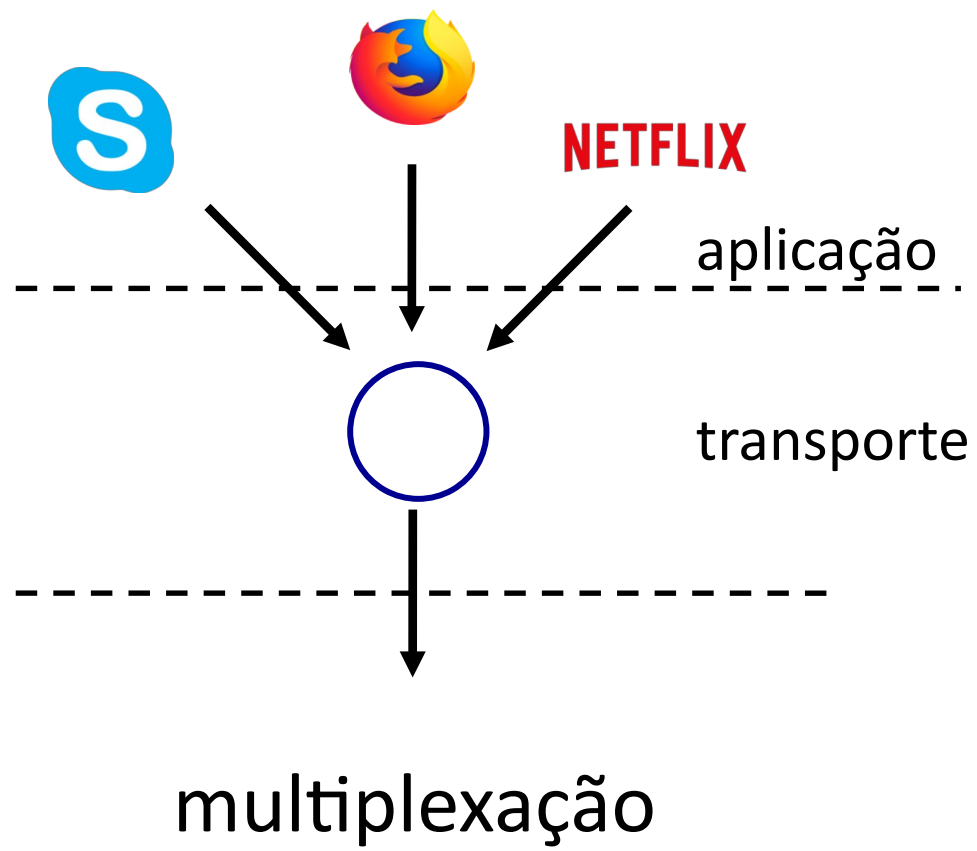
AIRFRANCE 

SKY
PRIORITY™





multiplexação

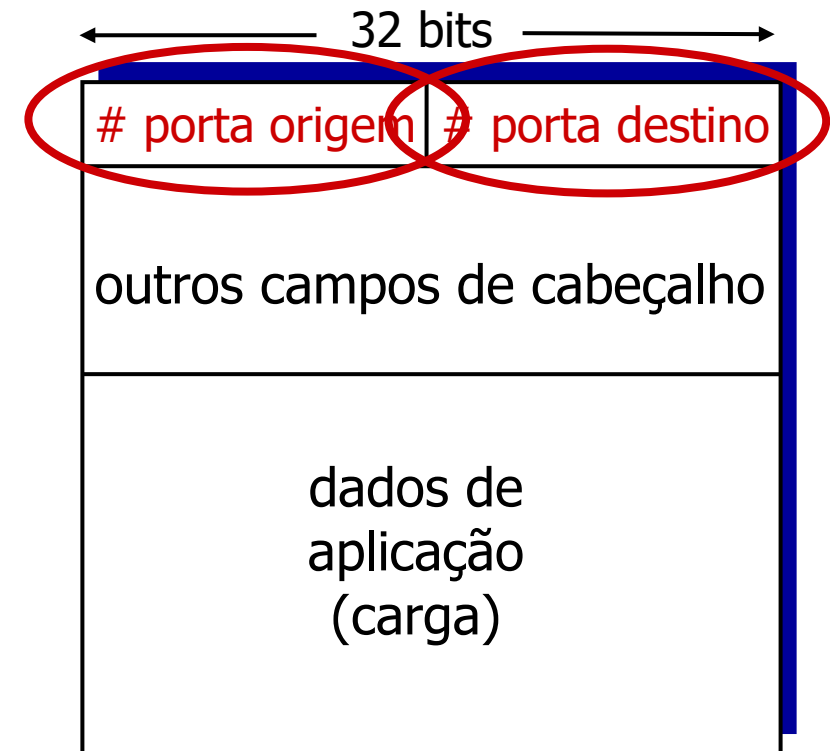




Multiplexação

Como funciona a demultiplexação

- hospedeiro recebe datagramas IP
- cada datagrama tem endereço IP de origem e endereço IP de destino
- cada datagrama carrega um segmento da camada de transporte
 - cada segmento tem números de porta de origem e de destino
- hospedeiro usa *endereços IP e números de portas* para direcionar segmento para o socket apropriado



formato de segmento TPC/UDP

Demultiplexação sem conexão

Lembre-se:

- ao criar um socket, é preciso especificar número de porta no *hospedeiro local*:

```
DatagramSocket mySocket1  
DatagramSocket(12534);
```



- ao criar datagrama para enviar para o socket UDP, deve-se especificar
 - endereço IP de destino
 - número de porta de destino

quando o hospedeiro receptor recebe um segmento *UDP*:

- verifica o número da porta de destino no segmento
- direciona o segmento UDP para o socket com aquele número de porta



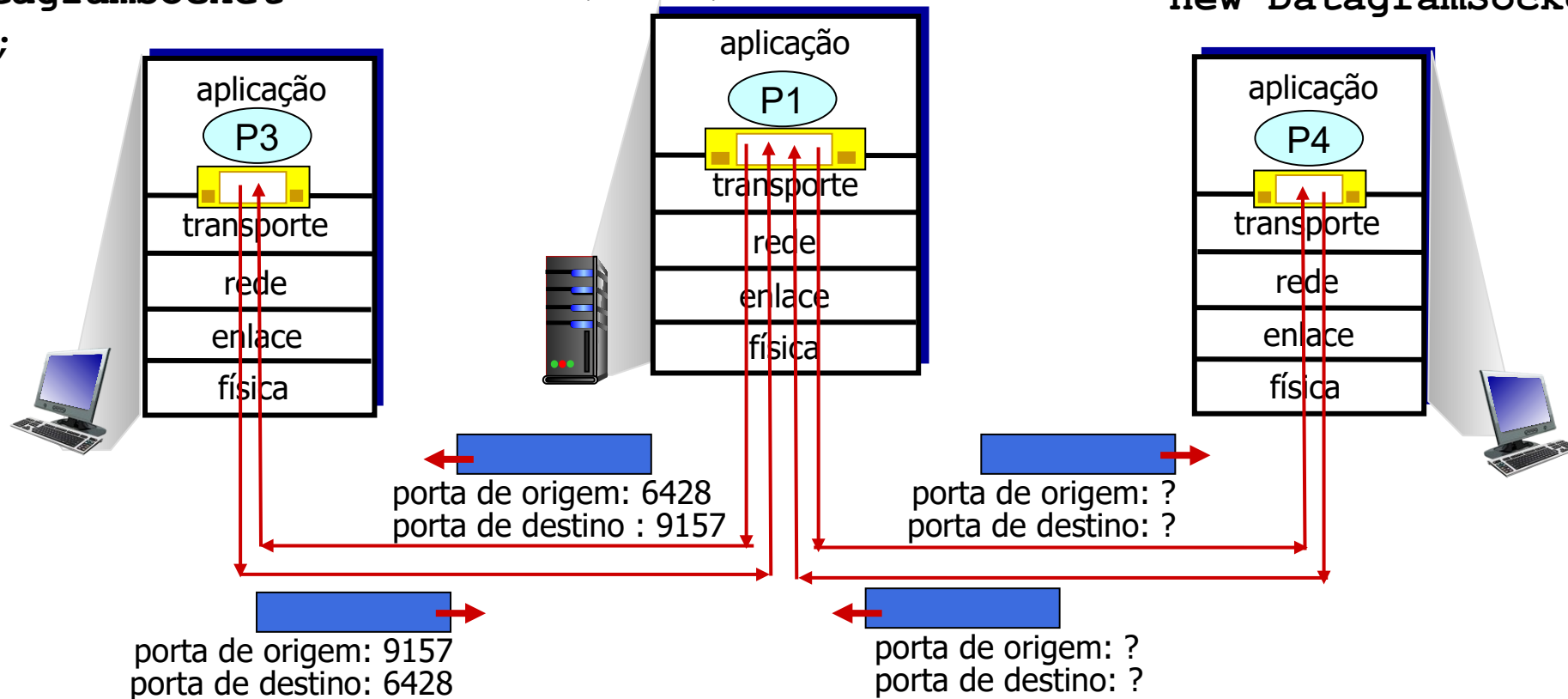
datagramas IP/UDP com o *mesmo número de porta de destino*, mas diferentes endereços IP de origem e/ou números de porta de origem serão direcionados para o *mesmo socket* no hospedeiro receptor

Demultiplexação sem conexão: um exemplo

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

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

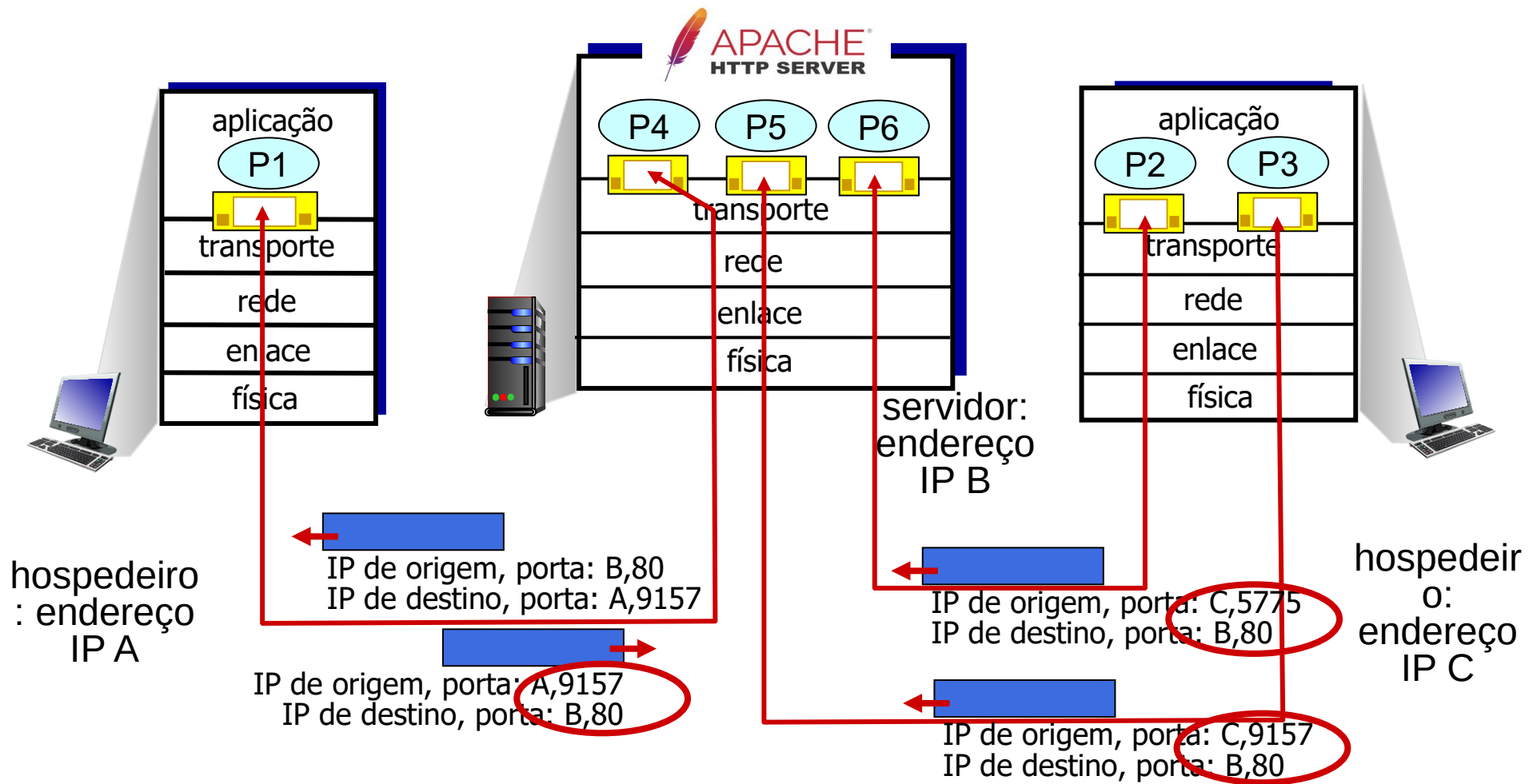
```
DatagramSocket mySocket1 =  
new DatagramSocket (5775) ;
```



Demultiplexação orientada a conexão

- socket TCP identificado por uma tupla de **4 elementos**:
 - endereço IP de origem
 - número de porta de origem
 - endereço IP de destino
 - número de porta de destino
- demultiplexação:
destinatário usa ***todos os quatro valores*** (*tupla de 4*)
para direcionar o segmento para o socket apropriado
- servidor pode suportar muitos sockets TCP simultâneos:
 - cada socket identificado por sua própria tupla de 4
 - cada socket associado com uma conexão de cliente diferente

Demultiplexação orientada a conexão: exemplo



Três segmentos, todos destinados ao endereço IP: B, porta de destino: 80 são demultiplexados para *diferentes* sockets

Resumo

- Multiplexação e demultiplexação: baseadas em valores de campos de cabeçalho do segmento e do datagrama
- **UDP:** demultiplexação usando (apenas) endereço IP e número da porta de destino
- **TCP:** demultiplexação usando tupla de 4 elementos: endereços IP e números de portas de origem e destino
- Multiplexação/demultiplexação acontece em *todas* as camadas

Camada de transporte: roteiro

- Serviços da camada de transporte
- Multiplexação e demultiplexação
- Transporte sem conexão: UDP
- Princípios da transferência confiável de dados
- Transporte orientado a conexão: TCP
- Princípios de controle de congestionamentos
- Controle de congestionamento do TCP
- Evolução da funcionalidade da camada de transporte



UDP: User Datagram Protocol

- protocolo de transporte da Internet “sem luxo”, só o básico
- serviço de “melhor esforço”, segmentos UDP podem ser:
 - perdidos
 - entregues fora de ordem à aplicação
- *sem conexão*:
 - sem *handshaking* entre emissor e receptor UDP
 - cada segmento UDP é manipulado independentemente dos outros

Por que existe UDP?

- nenhum estabelecimento de conexão (que pode adicionar atraso de RTT)
- simples: nenhum estado de conexão nos remetente e destinatário
- tamanho de cabeçalho pequeno
- nenhum controle de congestionamento
 - UDP pode disparar tão rápido quanto desejado!
 - pode funcionar na presença de congestionamento

UDP: User Datagram Protocol

- usos do UDP:
 - aplicações multimídia de streaming (tolerante à perda, sensível à taxa)
 - DNS
 - SNMP
 - HTTP/3
- se for necessária uma transferência confiável sobre o UDP (por exemplo, HTTP/3):
 - adicionar confiabilidade necessária na camada de aplicação
 - adicionar controle de congestionamento na camada de aplicação

UDP: User Datagram Protocol [RFC 768]

INTERNET STANDARD

RFC 768

J. Postel

ISI

28 August 1980

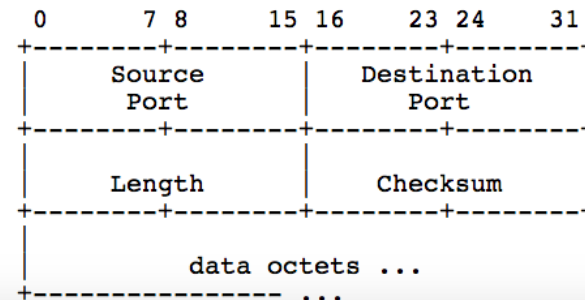
User Datagram Protocol

Introduction

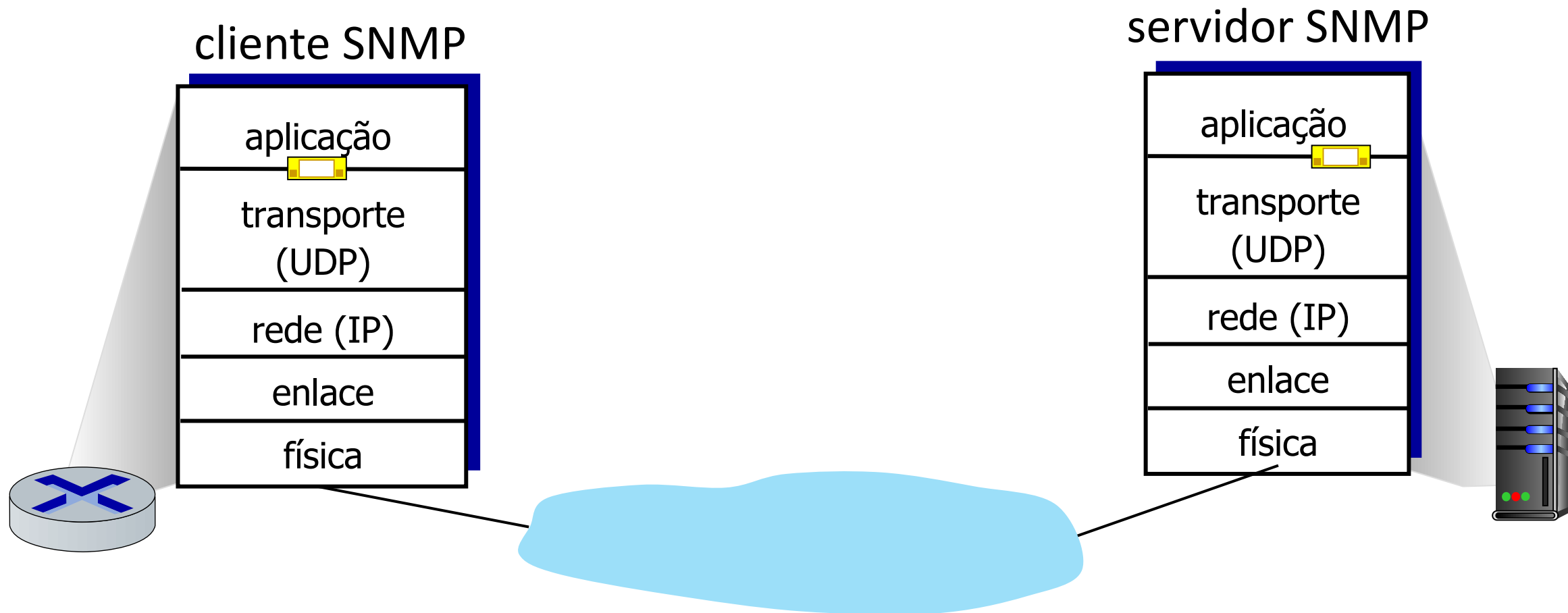
This User Datagram Protocol (UDP) is defined to make available a datagram mode of packet-switched computer communication in the environment of an interconnected set of computer networks. This protocol assumes that the Internet Protocol (IP) [1] is used as the underlying protocol.

This protocol provides a procedure for application programs to send messages to other programs with a minimum of protocol mechanism. The protocol is transaction oriented, and delivery and duplicate protection are not guaranteed. Applications requiring ordered reliable delivery of streams of data should use the Transmission Control Protocol (TCP) [2].

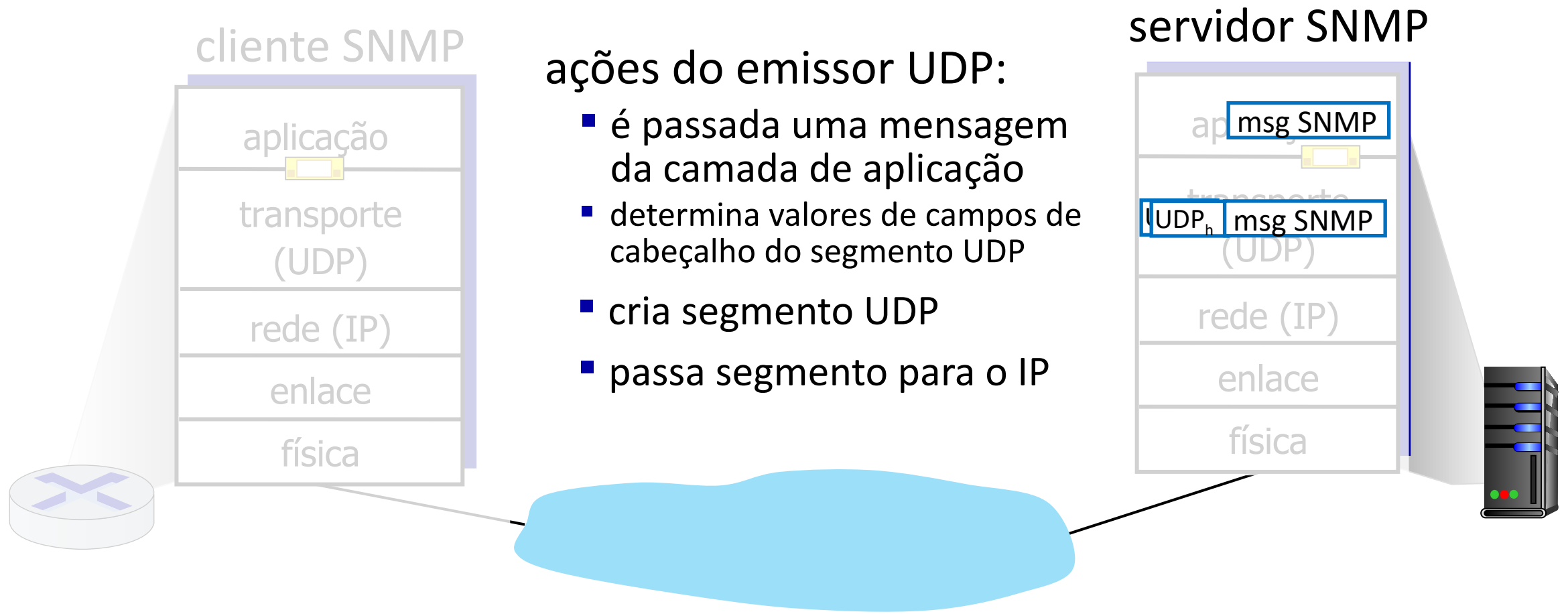
Format



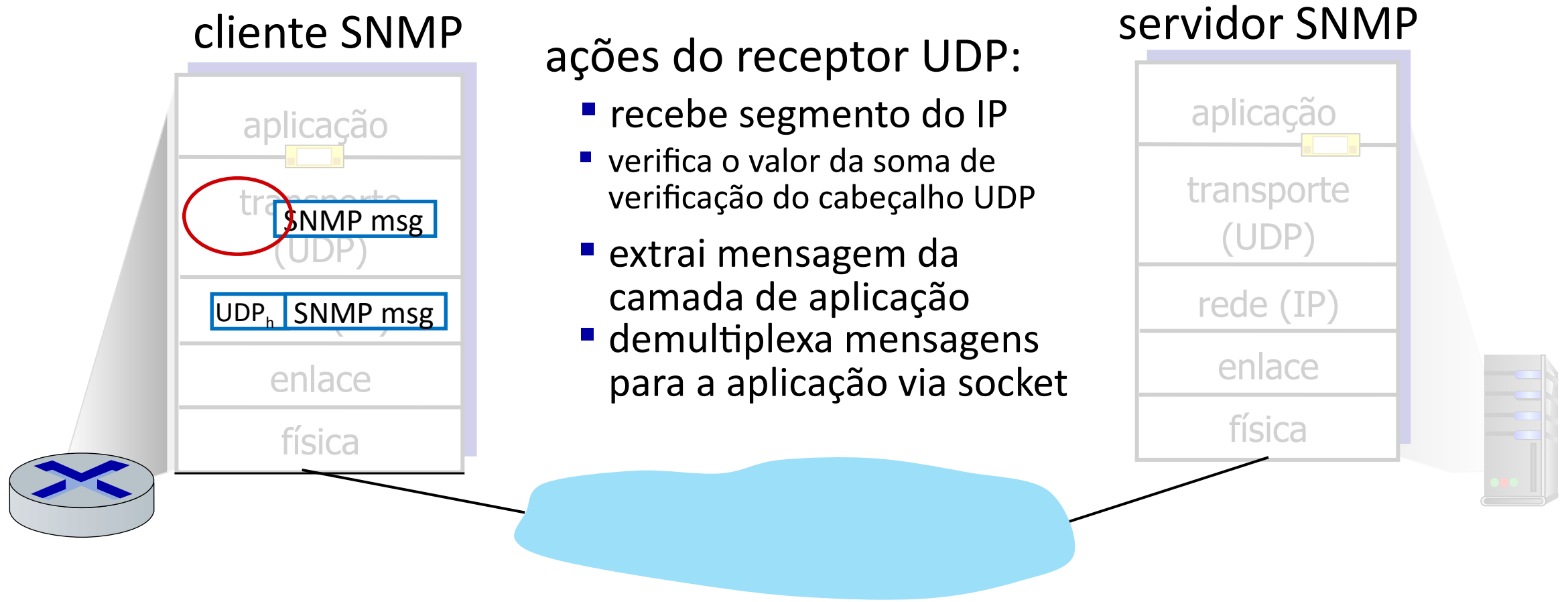
UDP: Ações da Camada de Transporte



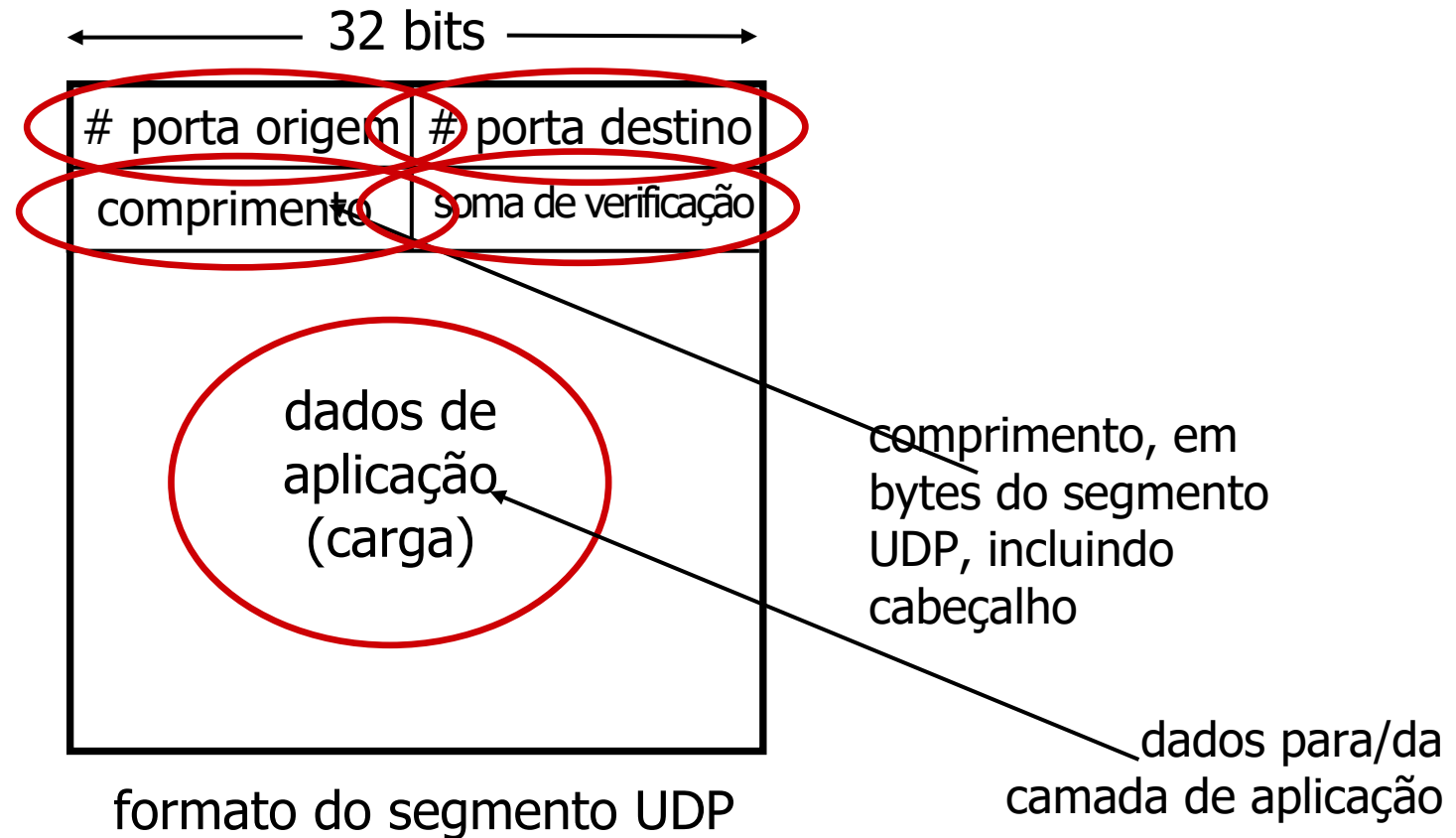
UDP: Ações da Camada de Transporte



UDP: Ações da Camada de Transporte



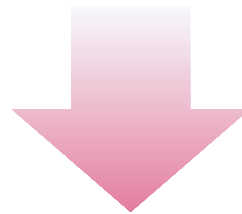
Cabeçalho do segmento UDP



Soma de verificação do UDP

Objetivo: detectar erros (ou seja, bits invertidos) em segmentos transmitidos

	1º número	2º número	soma
Transmitido:	5	6	11



Recebido:	4	6	11
-----------	---	---	----

soma de verificação
computada pelo receptor

≠

soma de verificação computada
pelo emissor (como recebida)



Soma de verificação do UDP

Objetivo: detectar erros (ou seja, bits invertidos) em segmentos transmitidos

emissor:

- trata conteúdo do segmento UDP (incluindo campos de cabeçalhos do UDP e endereços IP) como sequências de inteiros de 16 bits
- **soma de verificação:** adição (soma com complemento de um) do conteúdo do segmento
- valor da soma de verificação colocado no campo de soma de verificação do UDP

receptor:

- computa a soma de verificação dos segmentos recebidos
- verifica se a soma de verificação é igual ao valor no campo da soma de verificação:
 - não é igual - erro detectado
 - igual - sem erros detectados. *Mas talvez hajam erros mesmo assim? Mais a seguir*

Soma de verificação da Internet: um exemplo

exemplo: adicionar dois inteiros de 16 bits

	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
retorna	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1
soma	1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
soma de verificação	0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

Nota: ao adicionar números, um “vai um” do bit mais significativo precisa ser adicionado ao resultado

* Confira os exercícios interativos online para mais exemplos: http://gaia.cs.umass.edu/kurose_ross/interactive/

Soma de verificação da Internet: proteção fraca!

exemplo: adicionar dois inteiros de 16 bits

	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
retorna	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1
soma	1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
soma de verificação	0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

Apesar de os números terem mudado (bits invertidos), *nenhuma* mudança ocorre na soma de verificação!

Resumo: UDP

- protocolo “sem luxo”:
 - segmentos podem ser perdidos, entregues fora de ordem
 - serviço de melhor esforço: “envie e espere pelo melhor”
- UDP tem seus pontos positivos:
 - sem configuração/*handshaking* necessários (nenhum RTT a mais)
 - pode funcionar quando o serviço de rede está comprometido
 - ajuda com a confiabilidade (soma de verificação)
- funcionalidade adicional em cima do UDP pode ser criada na camada de aplicação (por exemplo, HTTP/3)

Camada de transporte: roteiro

- Serviços da camada de transporte
- Multiplexação e demultiplexação
- Transporte sem conexão: UDP
- **Princípios da transferência confiável de dados**
- Transporte orientado a conexão: TCP
- Princípios de controle de congestionamentos
- Controle de congestionamento do TCP
- Evolução da funcionalidade da camada de transporte



Princípios de transferência confiável de dados

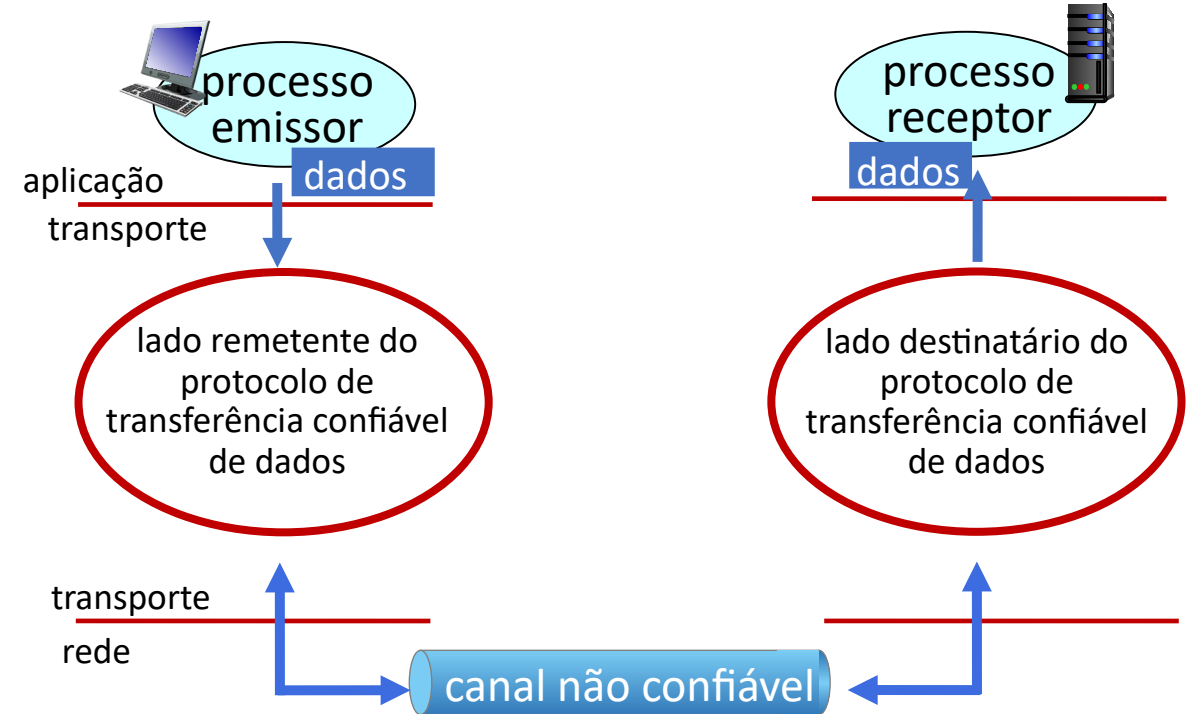
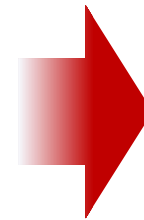


abstração de serviço confiável

Princípios de transferência confiável de dados



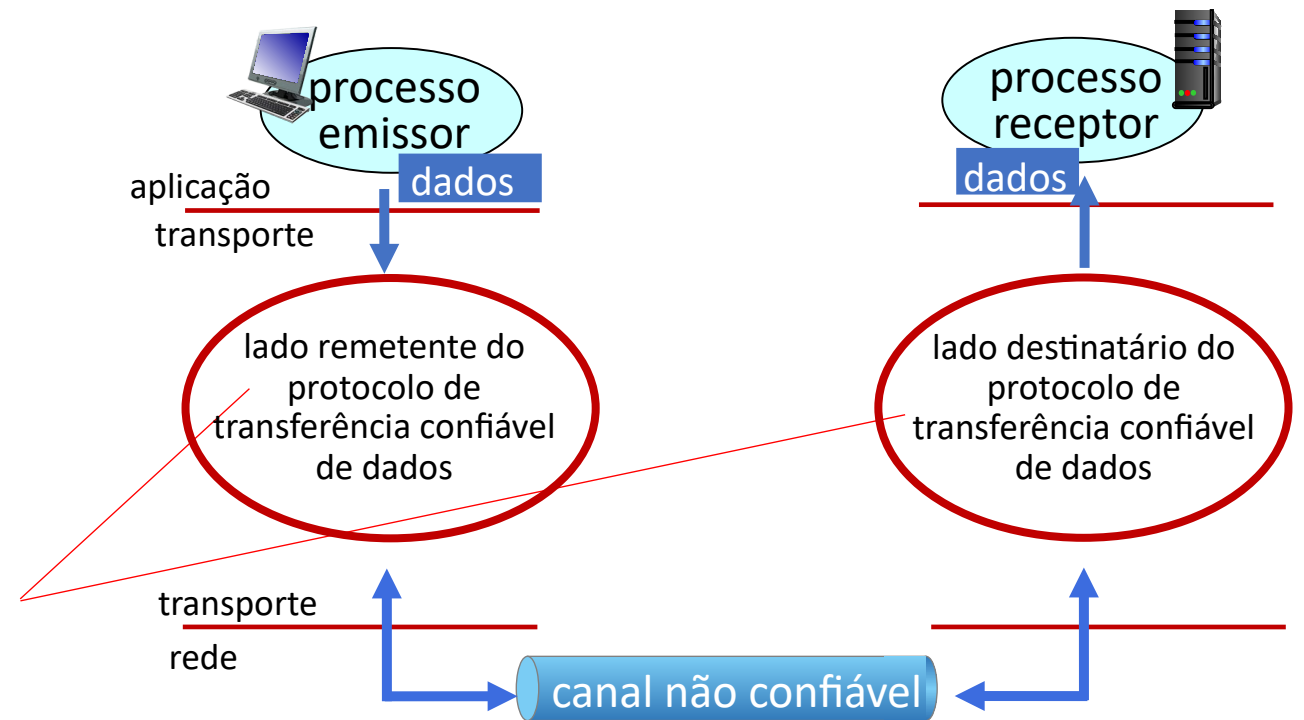
abstração de serviço confiável



Implementação de serviço confiável

Princípios de transferência confiável de dados

A complexidade do protocolo de transferência confiável de dados dependerá (fortemente) das características do canal não confiável (perde, corrompe, reordena dados?)

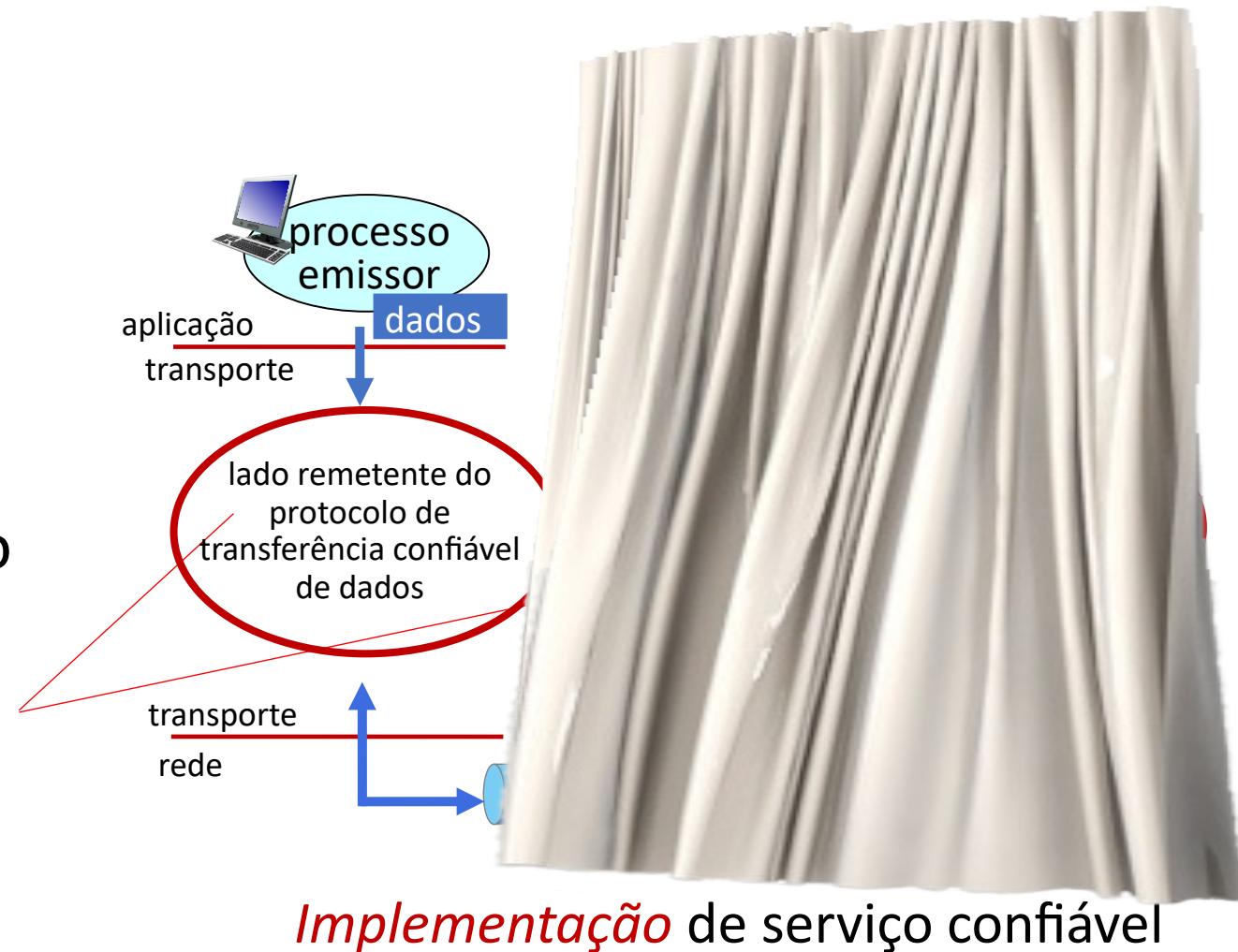


Implementação de serviço confiável

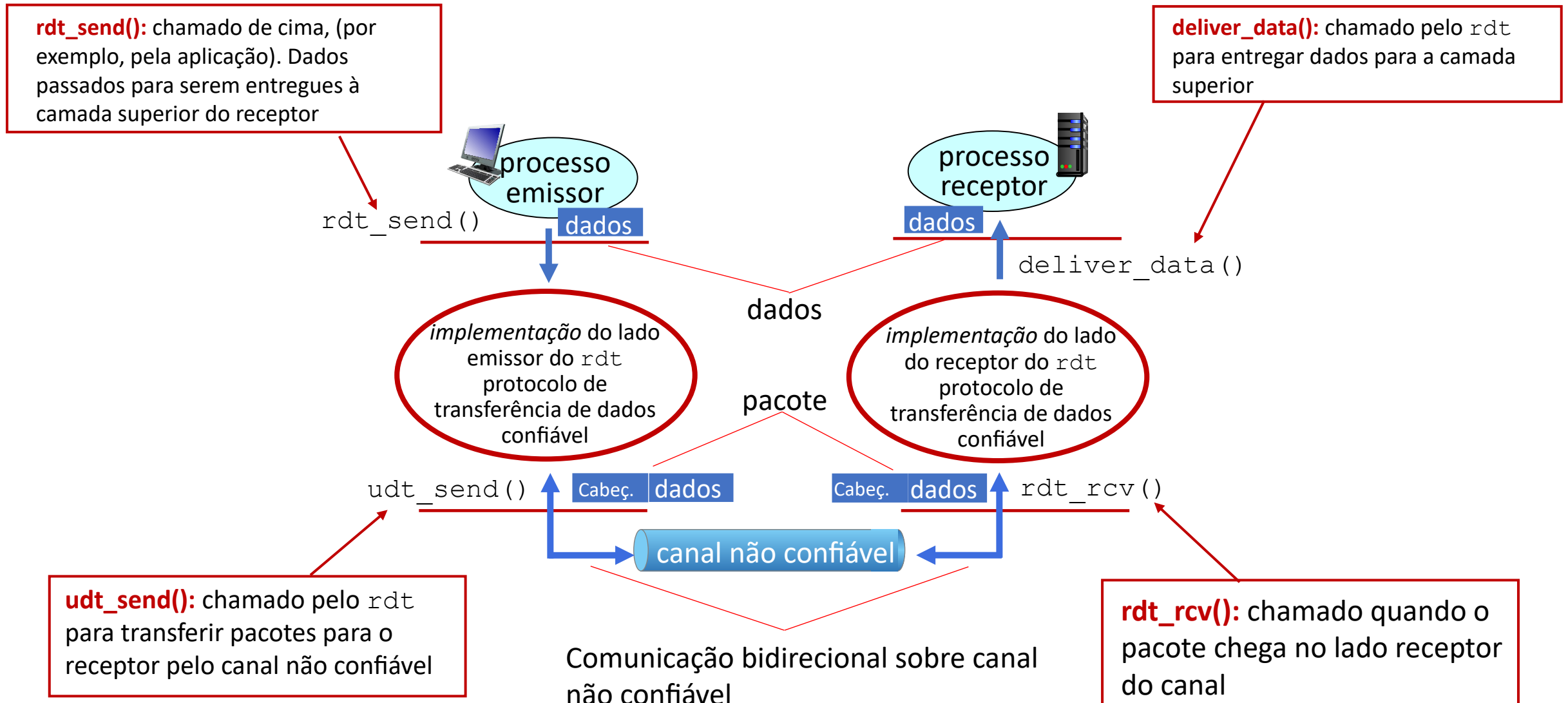
Princípios de transferência confiável de dados

Emissor e receptor *não* sabem o “estado” do outro, ex: uma mensagem foi recebida?

- a menos que comunicado através de uma mensagem



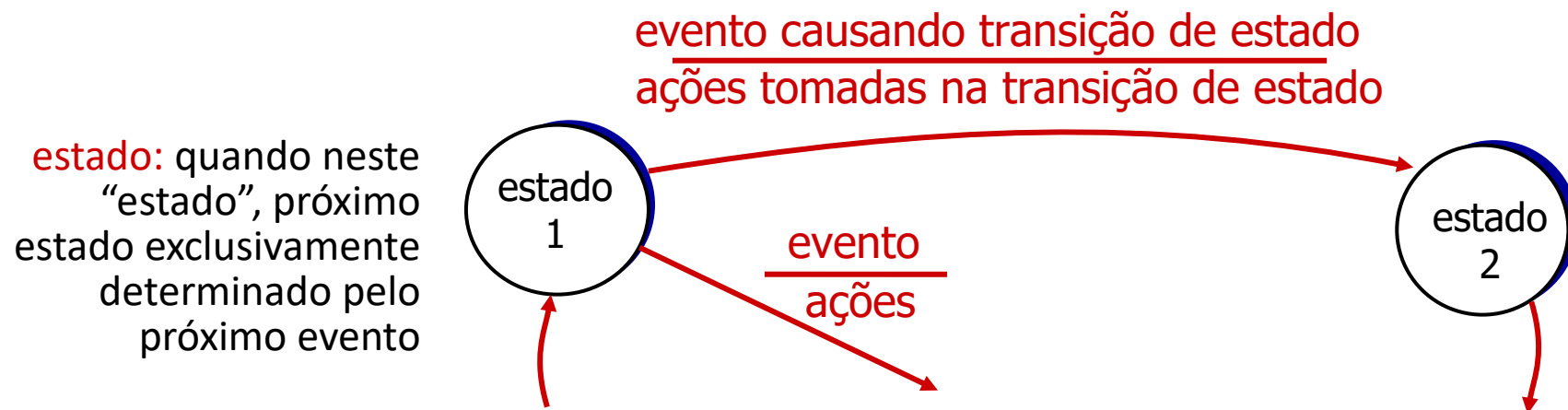
Protocolo de transferência confiável de dados (rdt): interfaces



Transferência confiável de dados: começando

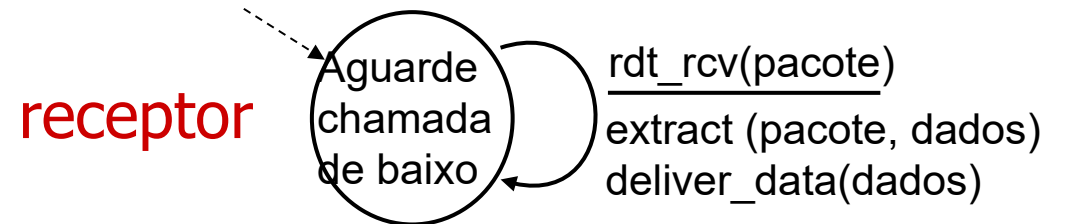
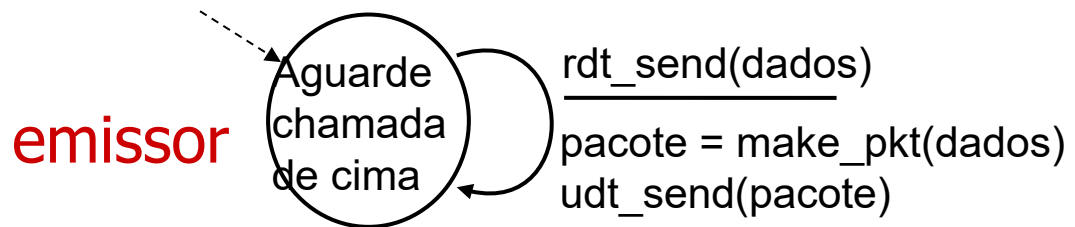
Nós vamos:

- desenvolver incrementalmente os lados emissor e receptor do protocolo de transferência confiável de dados (rdt)
- considerar apenas transferência de dados unidirecional
 - mas as informações de controle irão fluir em ambas as direções!
- usar máquinas de estado finito (FSM) para especificar emissor e receptor



rdt1.0: transferência confiável por um canal confiável

- canal subjacente perfeitamente confiável
 - sem erros de bit
 - sem perda de pacotes
- FSMs *separadas* para emissor e receptor:
- emissor envia dados para o canal subjacente
 - receptor lê dados do canal subjacente



rdt2.0: canal com erros de bit

- canal subjacente pode inverter bits no pacote
 - soma de verificação (ex.: soma de verificação da Internet) pode detectar erros de bit
- a questão: como se recuperar de erros?

Como os humanos se recuperam de “erros” durante uma conversa?

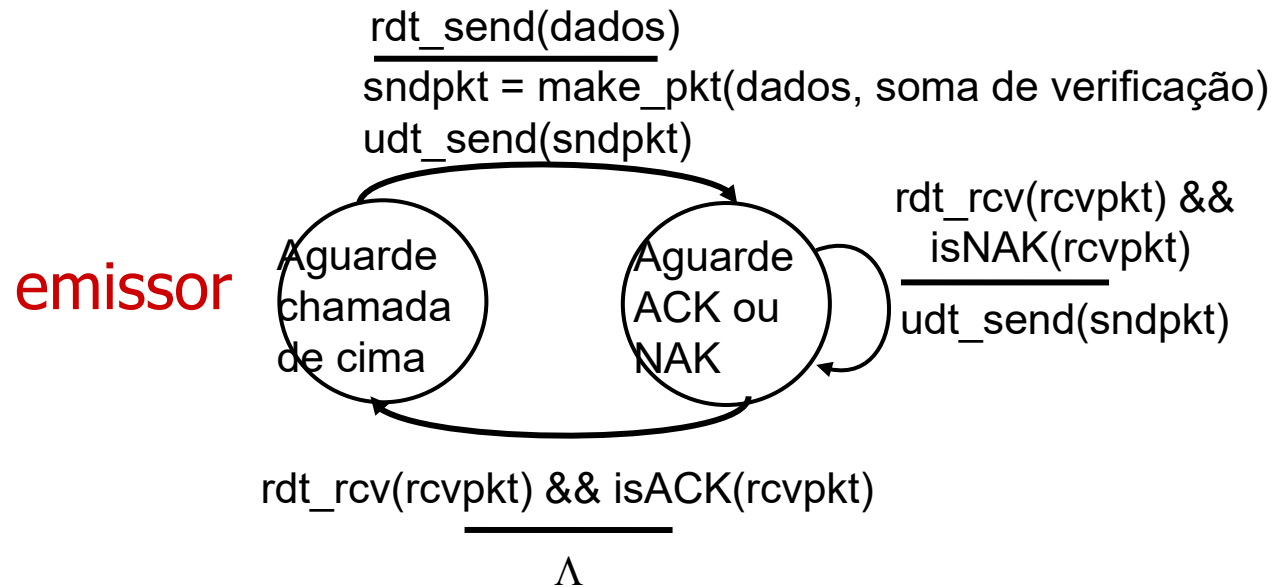
rdt2.0: canal com erros de bit

- canal subjacente pode inverter bits no pacote
 - soma de verificação para detectar erros de bit
- a questão: como se recuperar de erros?
 - *confirmação (ACKs)*: receptor explicitamente diz ao remetente que pacote recebido está OK
 - *confirmação negativa (NAKs)*: receptor explicitamente diz ao remetente que pacote tinha erros
 - emissor *retransmite* pacote ao receber um NAK

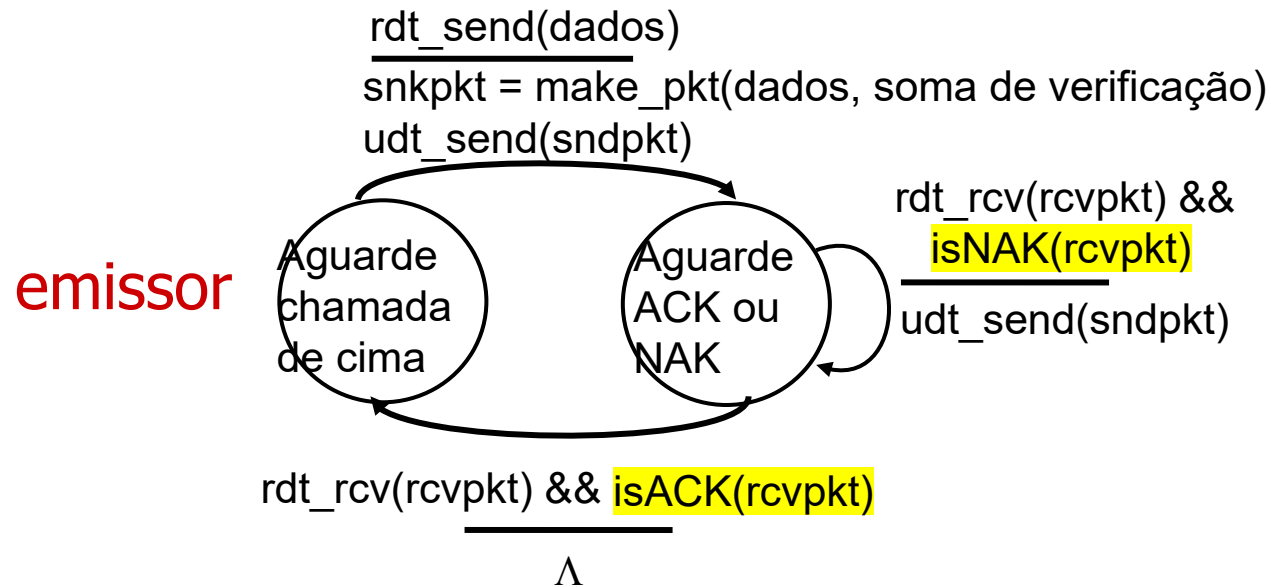
pare e espere

remetente envia um pacote, em seguida, espera por resposta do receptor

rdt2.0: especificações da FSM



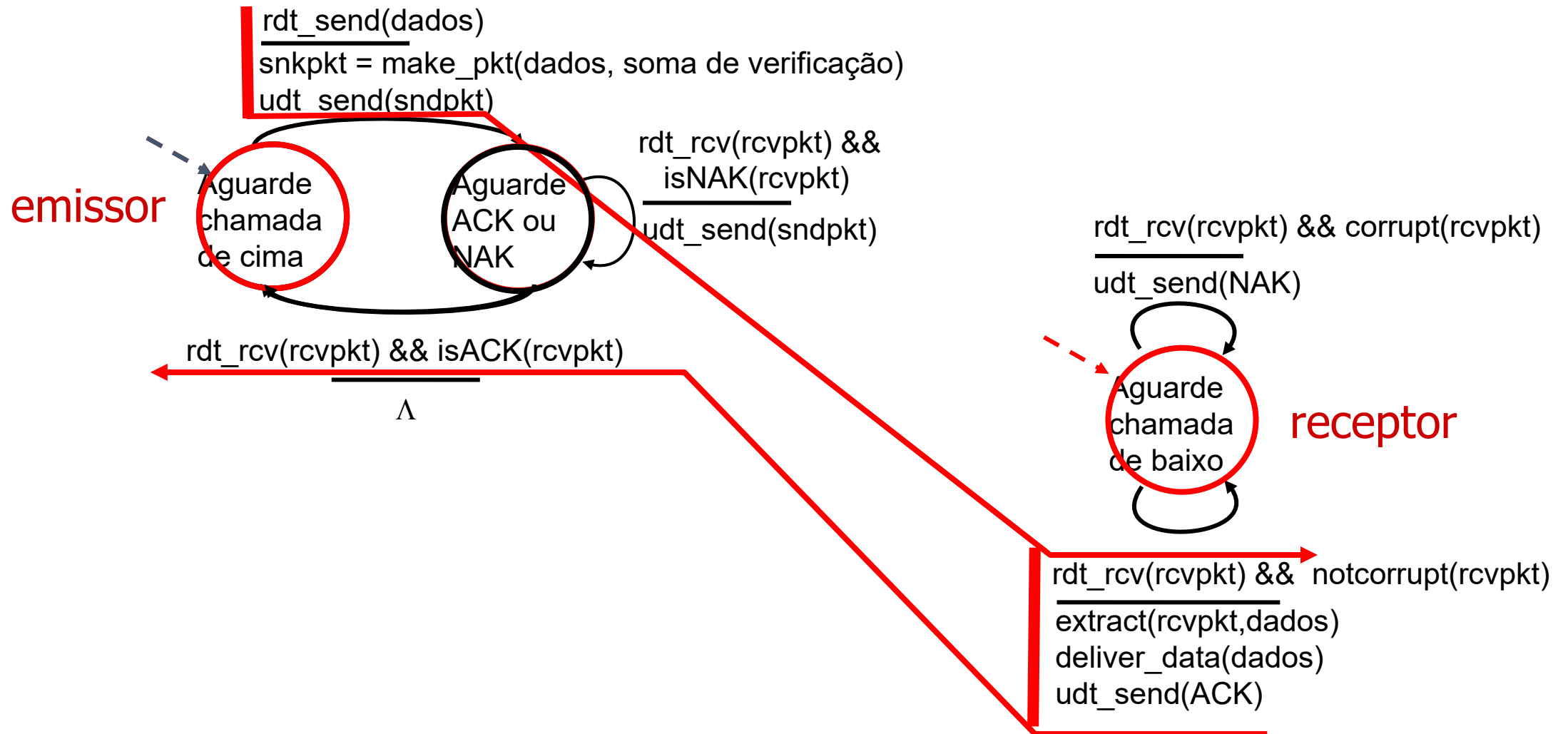
rdt2.0: especificações da FSM



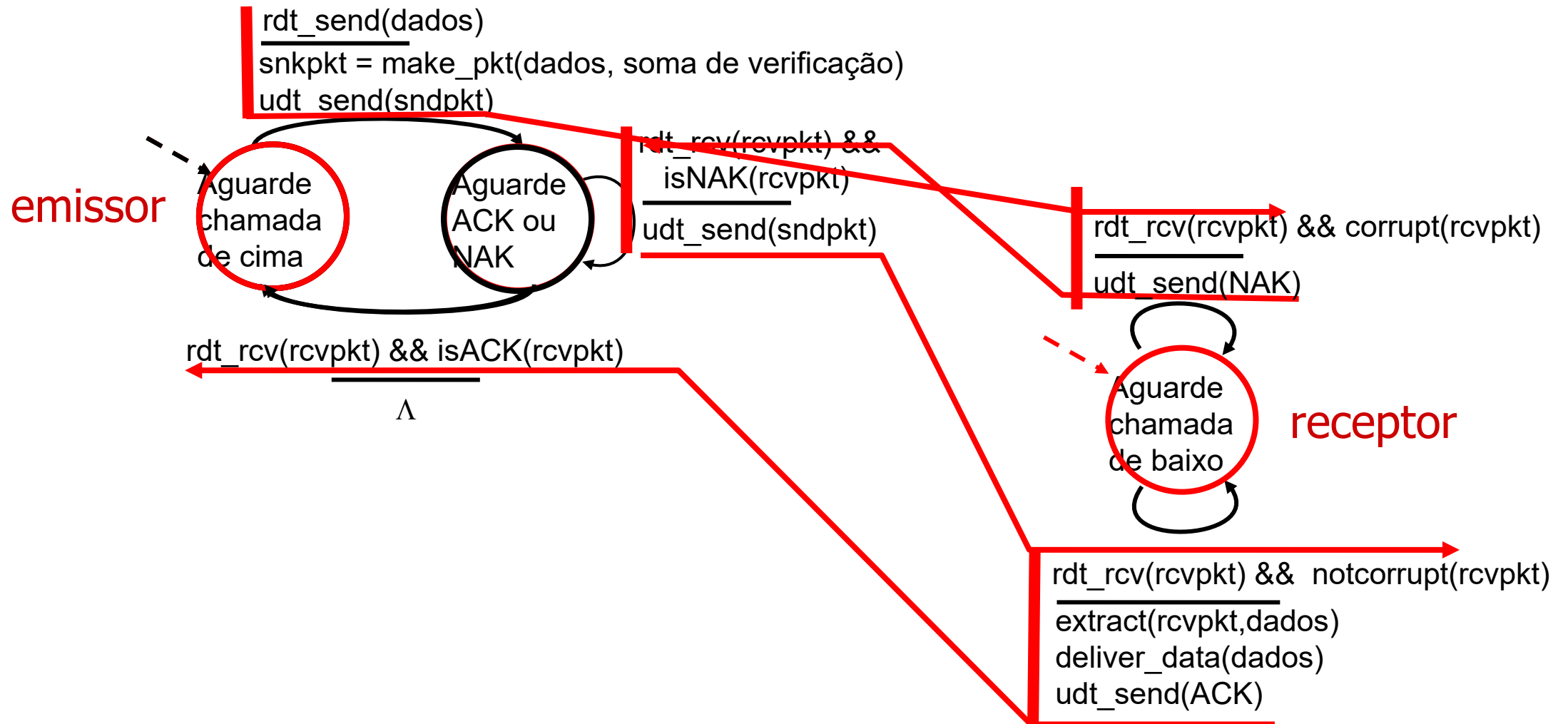
- Nota:** “estado” do receptor (o receptor entendeu minha mensagem corretamente?) não é conhecido pelo emissor a menos que seja comunicado de alguma forma do receptor para o transmissor
- É por isso que precisamos de um protocolo!



rdt2.0: operação sem erros



rdt2.0: cenário de pacote corrompido



rdt2.0 tem uma falha fatal!

o que acontece se ACK/NAK for corrompido?

emissor não sabe o que aconteceu no receptor!

- não pode simplesmente retransmitir: possível duplicata

manipulação de duplicatas:

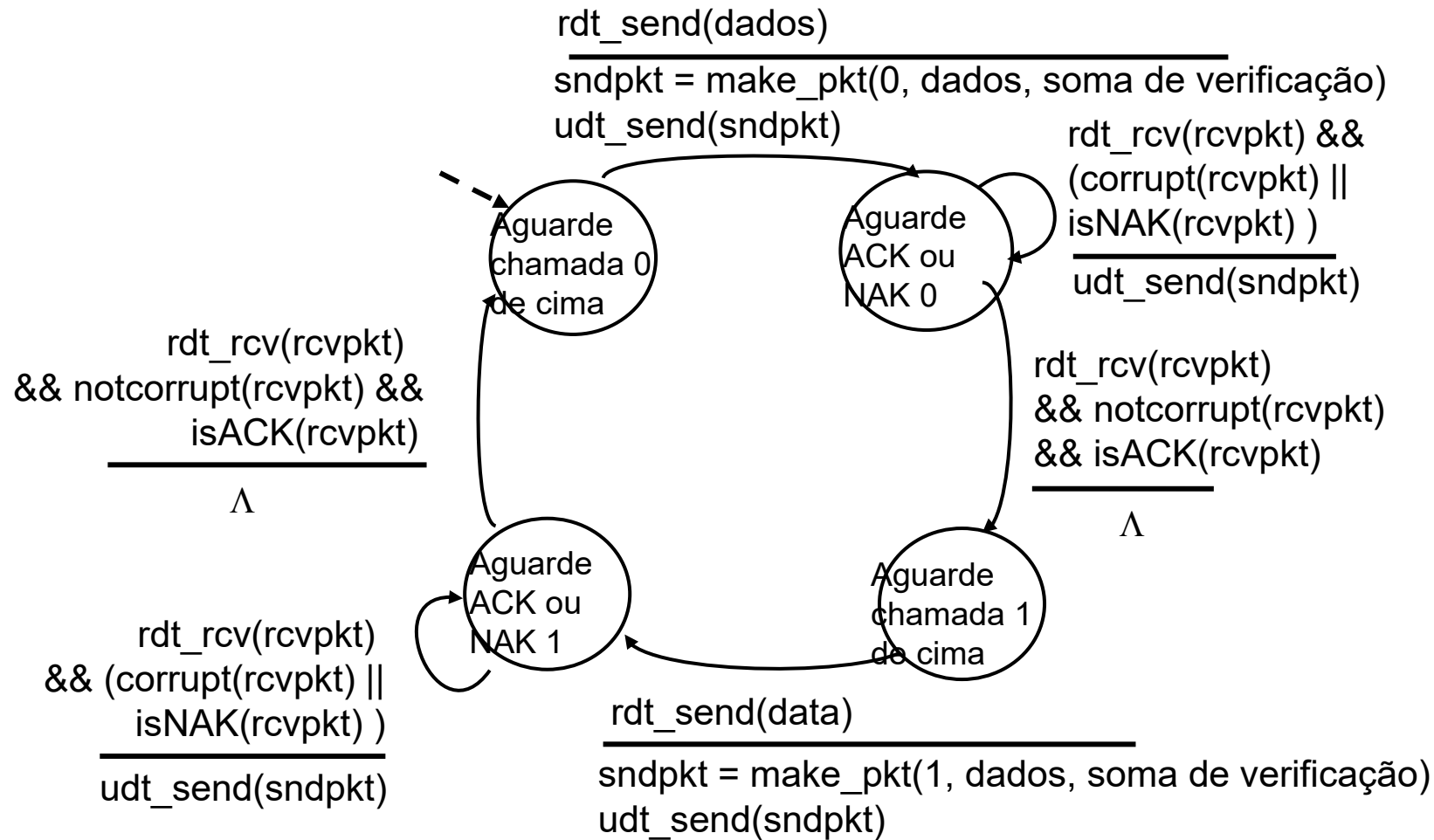
remetente retransmite pacote atual se ACK/NAK estiver corrompido

- emissor adiciona *número de sequência* para cada pacote
- receptor descarta (não entrega) pacote duplicado

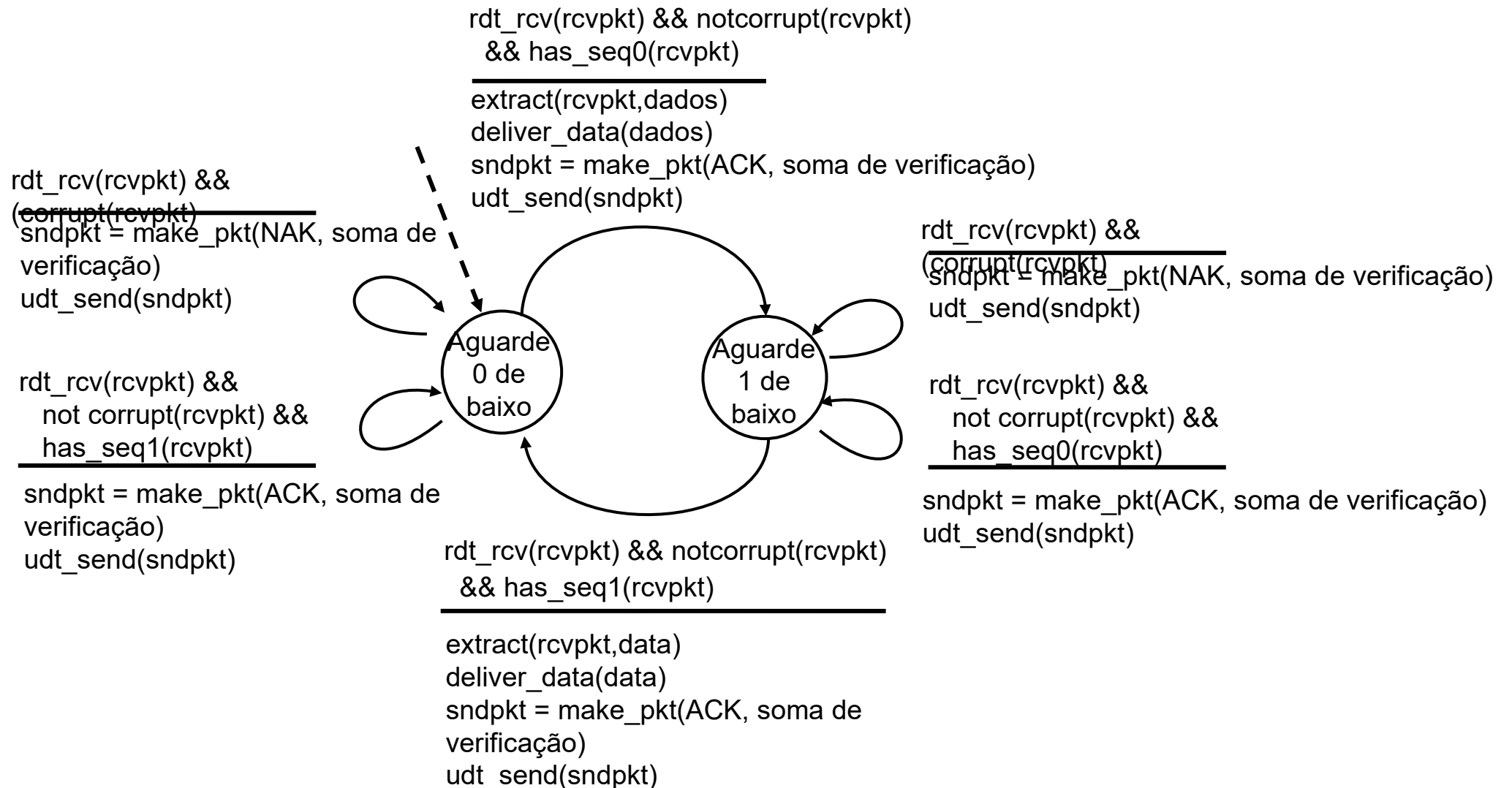
pare e espere

remetente envia um pacote, em seguida, espera por resposta do receptor

rdt2.1: emissor, manipulando ACK/NAKs corrompidos



rdt2.1: receptor, manipulando ACK/NAKs corrompidos



rdt2.1: discussão

emissor:

- número de sequência adicionado a pacote
- dois números de sequência (0,1) são suficientes. Por que?
- deve verificar se recebeu ACK/NAK corrompido
- duas vezes mais estados
 - estado deve “lembrar” se pacote “esperado” deve ter número de sequência 0 ou 1

receptor:

- deve verificar se o pacote recebido é duplicado
 - estado indica se 0 ou 1 é o número de sequência do pacote esperado
- nota: o receptor *não* tem como saber se o último ACK/NAK foi recebido ok no emissor

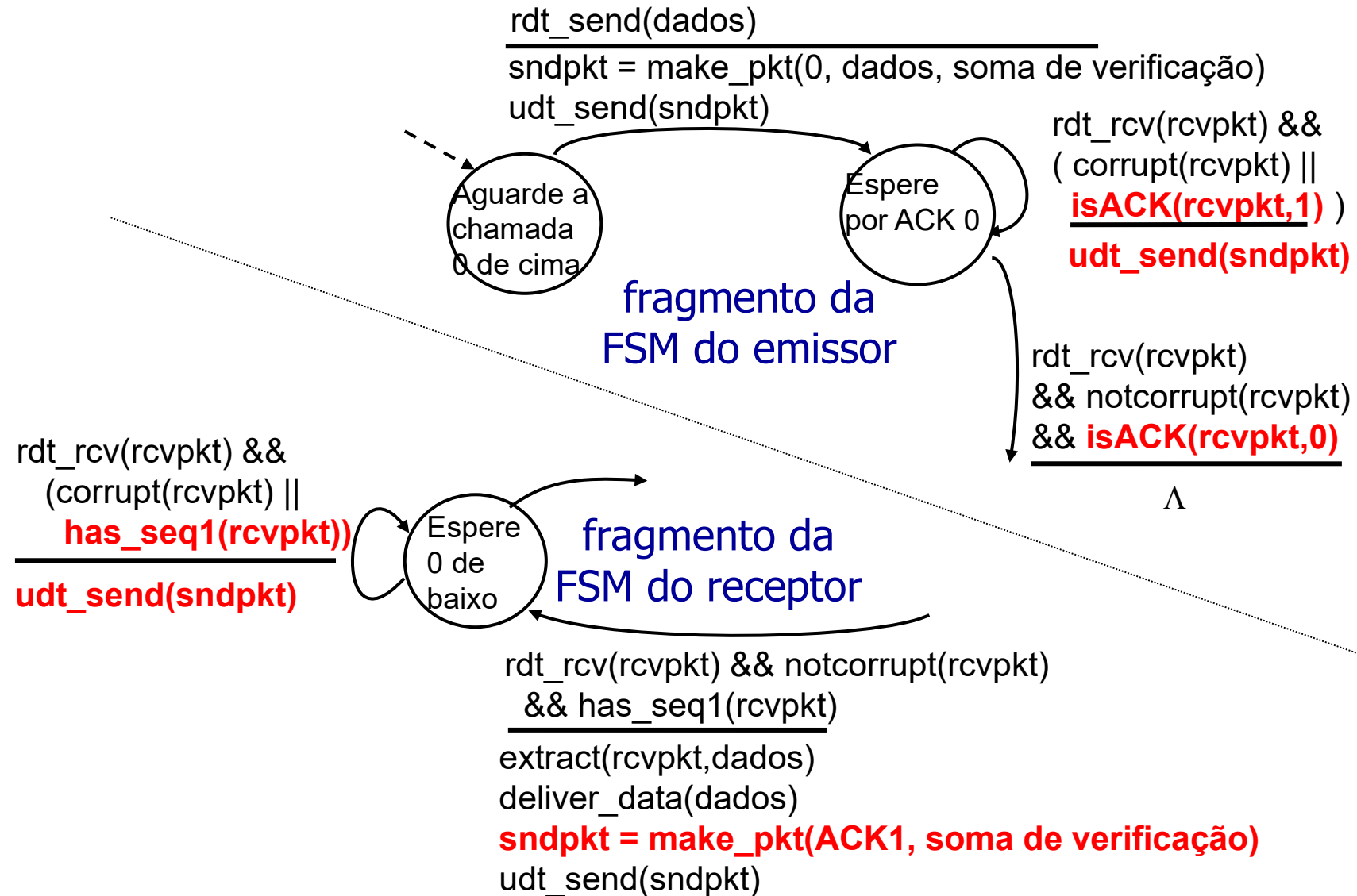
rdt2.2: um protocolo sem NAK

mesma funcionalidade do rdt2.1, usando apenas ACKs

- em vez de NAK, receptor envia ACK para último pacote recebido OK
 - receptor deve incluir *explicitamente* número de sequência do pacote sendo reconhecido
- ACK duplicado no remetente resulta na mesma ação que o NAK: *retransmitir pacote atual*

Como veremos, o TCP usa essa abordagem para ser livre de NAK

rdt2.2: fragmentos do emissor e do receptor



rdt3.0: canais com erros e perdas

Nova suposição do canal: canal subjacente também pode *perder* pacotes (dados, ACKs)

- soma de verificação, números de sequência, ACKs, retransmissões ajudarão ... mas não são o suficiente

Q: Como os *humanos* lidam com palavras perdidas de emissor para receptor na conversa?

rdt3.0: canais com erros e perdas

Abordagem: emissor espera quantidade “razoável” de tempo por ACK

- retransmite se nenhum ACK foi recebido neste tempo
- se pacote (ou ACK) está apenas atrasado (não perdido):
- a retransmissão será duplicada, mas os números de sequência já lidam com isso!
 - receptor deve especificar número de sequência do pacote sendo reconhecido
- usa temporizador de contagem regressiva para interromper após quantidade “razoável” de tempo



timeout

Camada de transporte: roteiro

- Serviços da camada de transporte
- Multiplexação e demultiplexação
- Transporte sem conexão: UDP
- Princípios da transferência confiável de dados
- Transporte orientado a conexão: TCP
 - estrutura do segmento
 - transferência confiável de dados
 - controle de fluxo
 - gerenciamento de conexão
 - Princípios de controle de congestionamentos
 - Controle de congestionamento do TCP
- Evolução da funcionalidade da camada de transporte

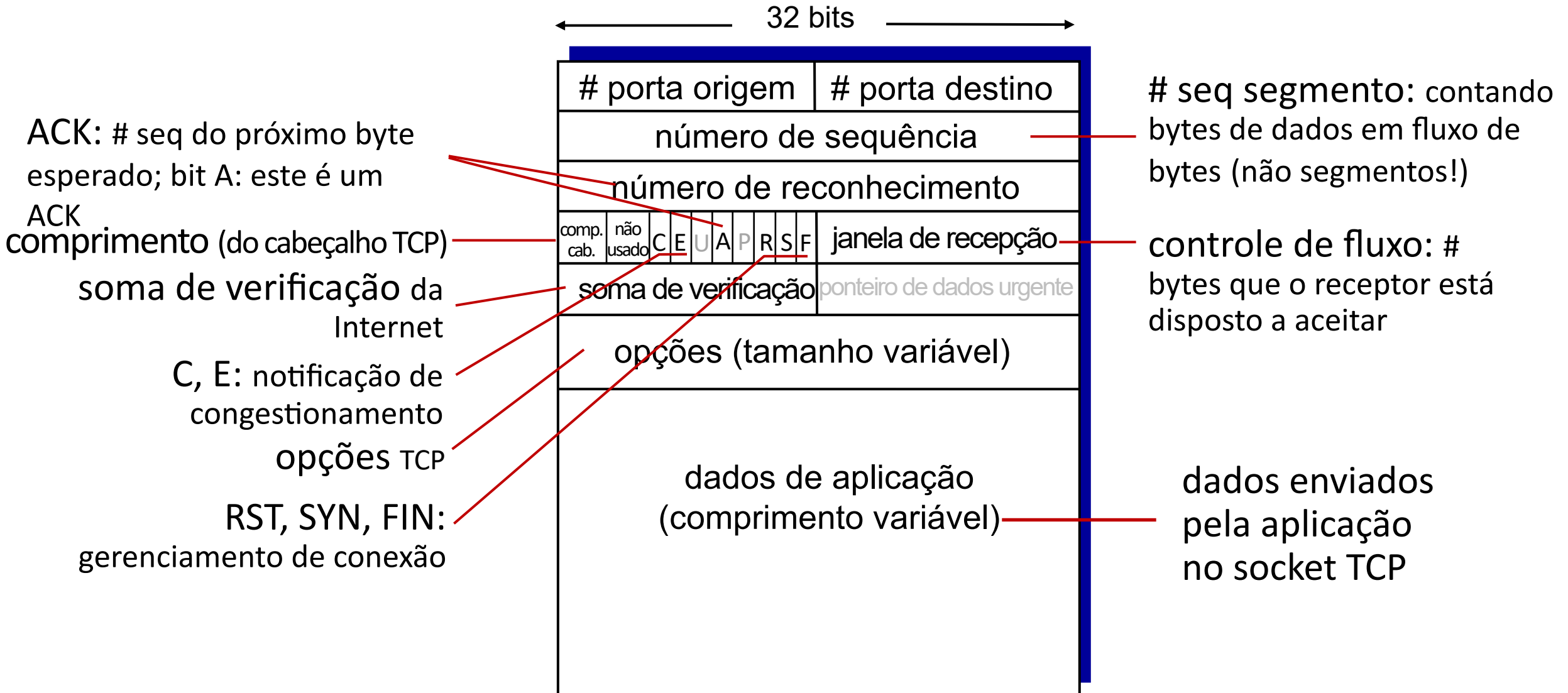


TCP: visão geral

RFCs: 793, 1122, 2018, 5681, 7323

- **ponto a ponto:**
 - um emissor, um receptor
- ***fluxo de bytes confiável e em ordem:***
 - sem “fronteira de mensagem”
- ***dados full duplex:***
 - fluxos de dados bidirecionais na mesma conexão
 - MSS: maximum segment size
- ***ACKs cumulativos***
- ***pipelining:***
 - controle de congestionamento e de fluxo do TCP ajustam tamanho da janela
- ***orientado a conexão:***
 - handshaking (troca de mensagens de controle) inicializa estados do emissor e do receptor antes da troca de dados
- ***fluxo controlado:***
 - emissor não sobrecarrega receptor

Estrutura do segmento TCP



Números de sequência do TCP, ACKs

Números de sequência:

- “número” no fluxo de bytes do primeiro byte nos dados do segmento

Reconhecimentos:

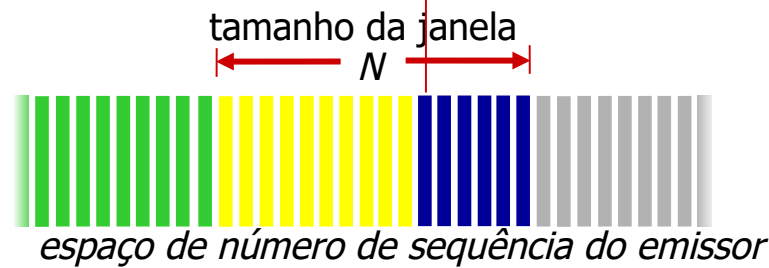
- número de sequência do próximo byte esperado do outro lado
- ACK cumulativo

Q: como o receptor lida com segmentos fora de ordem

- R: a especificação TCP não diz, - depende do implementador

segmento de saída do emissor

# porta origem		# porta destino	
número de sequência			
número de reconhecimento			
			janela recep.
soma verif.		ponteiro urg.	

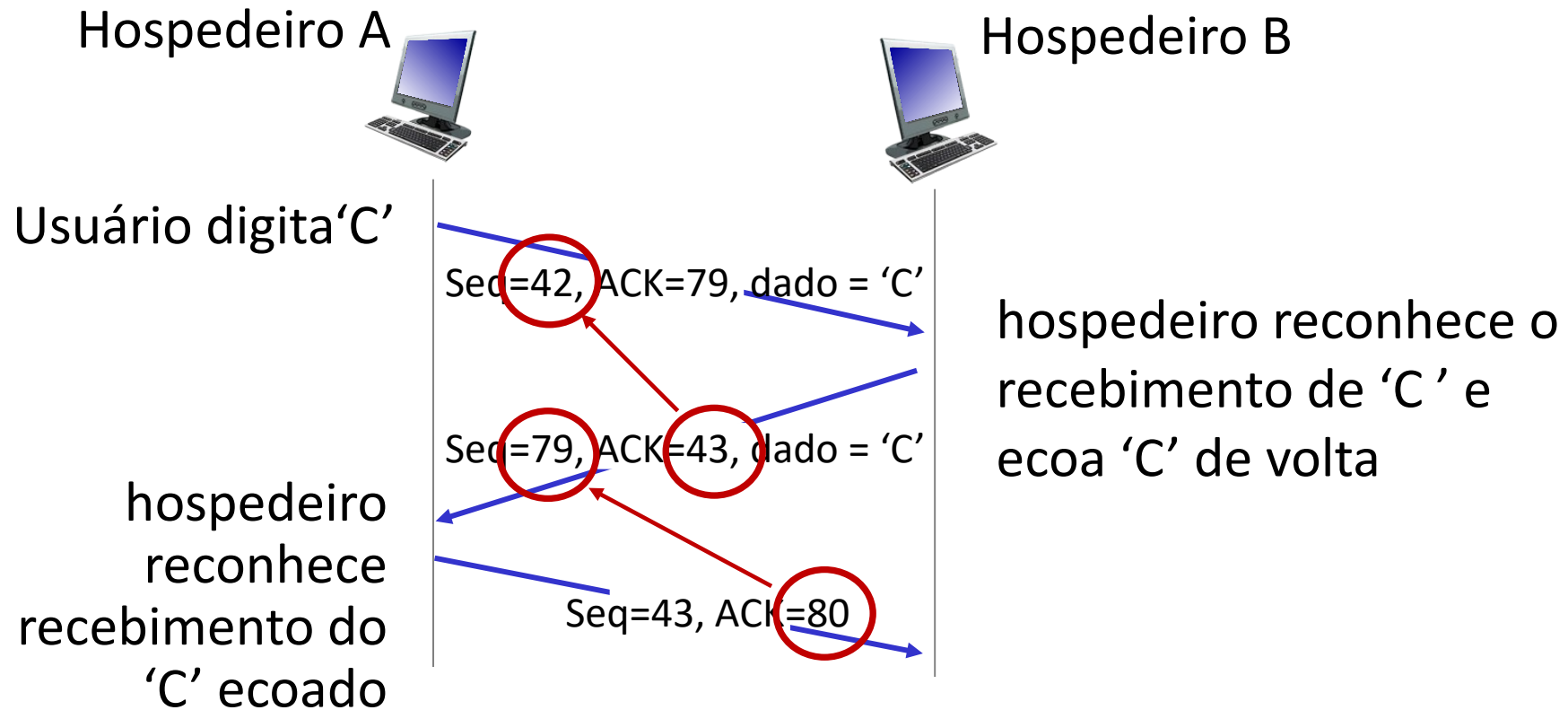


enviado, reconhecido enviado, ainda não reconhecido ("em voo") usável mas ainda não enviado não usável

segmento de saída do receptor

# porta origem		# porta destino	
número de sequência			
número de reconhecimento			
		A	janela recep.
soma verif.		ponteiro urg.	

Números de sequência do TCP, ACKs



cenário de telnet simples

Tempo e ida e volta e tempo limite do TCP

Q: como definir o valor de tempo limite do TCP?

- mais longo que o RTT, mas o RTT varia!
- *muito curto: timeout* prematuro, retransmissões desnecessárias
- *muito longo: reação* lenta à perda de segmento

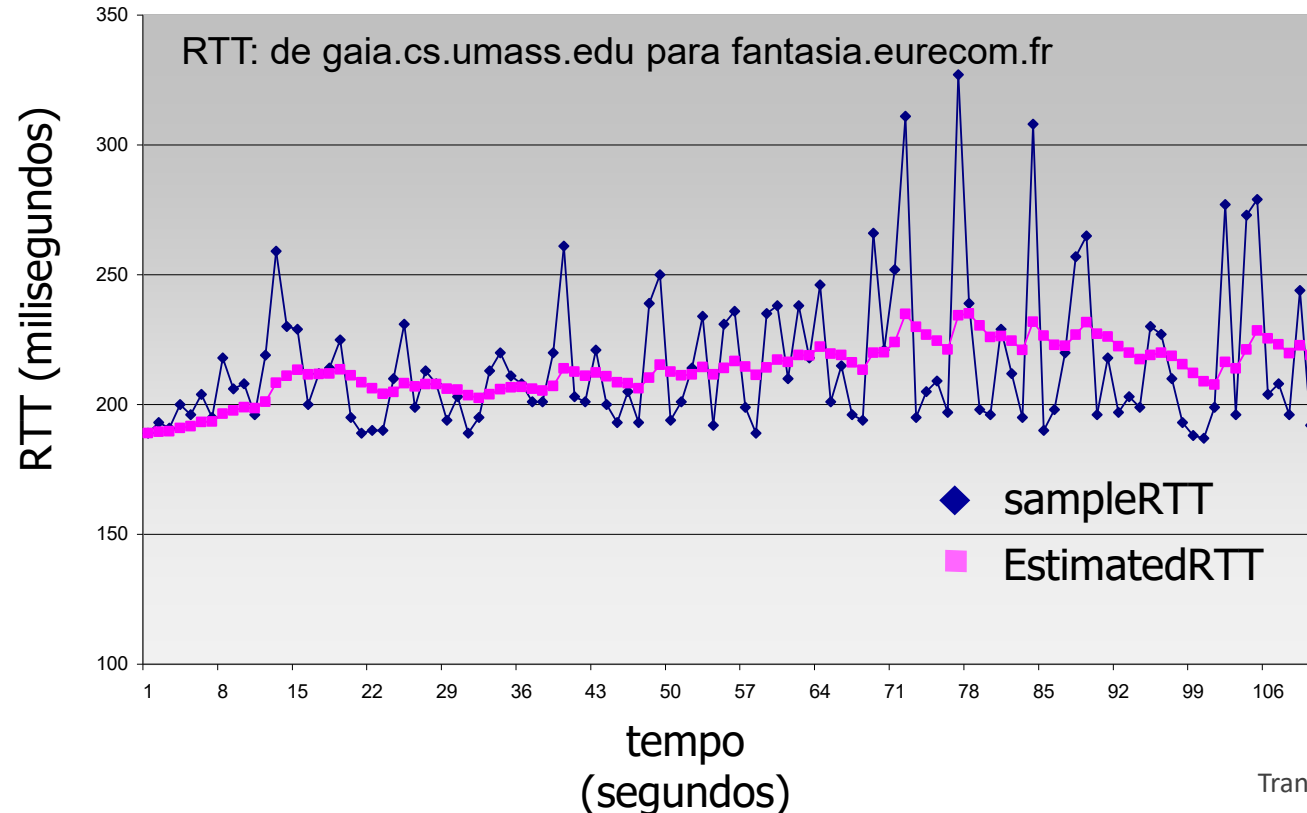
Q: como estimar o RTT?

- `SampleRTT`: tempo medido desde a transmissão do segmento até o recebimento do ACK
 - ignora retransmissões
- `SampleRTT` vai variar, queremos um RTT estimado “mais suave”
 - média de várias *medições recentes*, não apenas `SampleRTT` atual

Tempo e ida e volta e tempo limite do TCP

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

- exponential weighted moving average (EWMA) - média móvel exponencial ponderada
- influência da amostra passada diminui exponencialmente rápido
- valor típico: $\alpha = 0,125$



Tempo e ida e volta e tempo limite do TCP

- intervalo de *timeout*: **EstimatedRTT** mais “margem de segurança”
 - grande variação em **EstimatedRTT**: requer uma margem de segurança maior

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



↑
RTT estimado

↑
“margem de segurança”

- **DevRTT**: EWMA do desvio de **SampleRTT** com relação a **EstimatedRTT**:

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

(tipicamente, $\beta = 0,25$)

Transmissor TCP (simplificado)

evento: dados recebidos da aplicação

- cria segmento com número de sequência
- número de sequência é o número no fluxo de bytes do primeiro byte de dados no segmento
- inicia temporizador se já não estiver executando
 - o temporizador corresponde ao segmento mais antigo não reconhecido
 - intervalo de expiração: **TimeoutInterval**

evento: timeout

- retransmite segmento que causou o timeout
- reinicia o temporizador

evento: ACK recebido

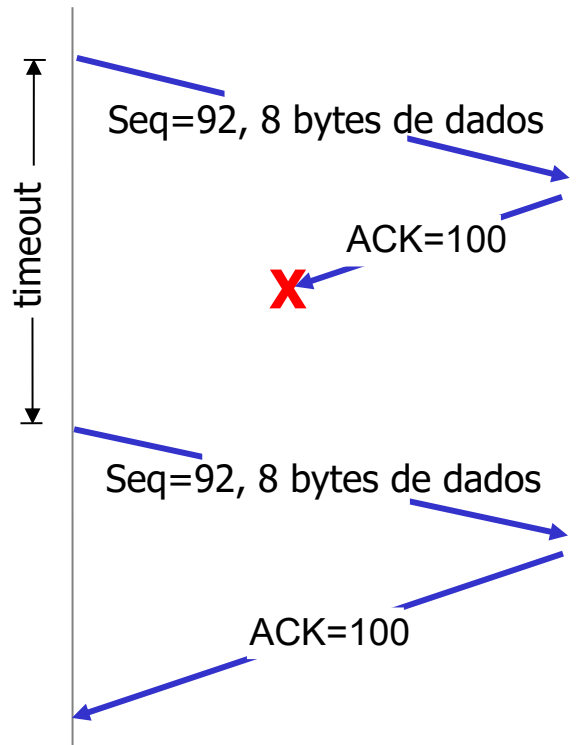
- se o ACK reconhece segmentos previamente não reconhecidos
 - atualiza o que já se sabe que foi reconhecido
 - inicia temporizador se ainda existem segmentos não reconhecidos

Receptor TCP: geração de ACK [RFC 5681]

<i>Evento no receptor</i>	<i>Ação do receptor TCP</i>

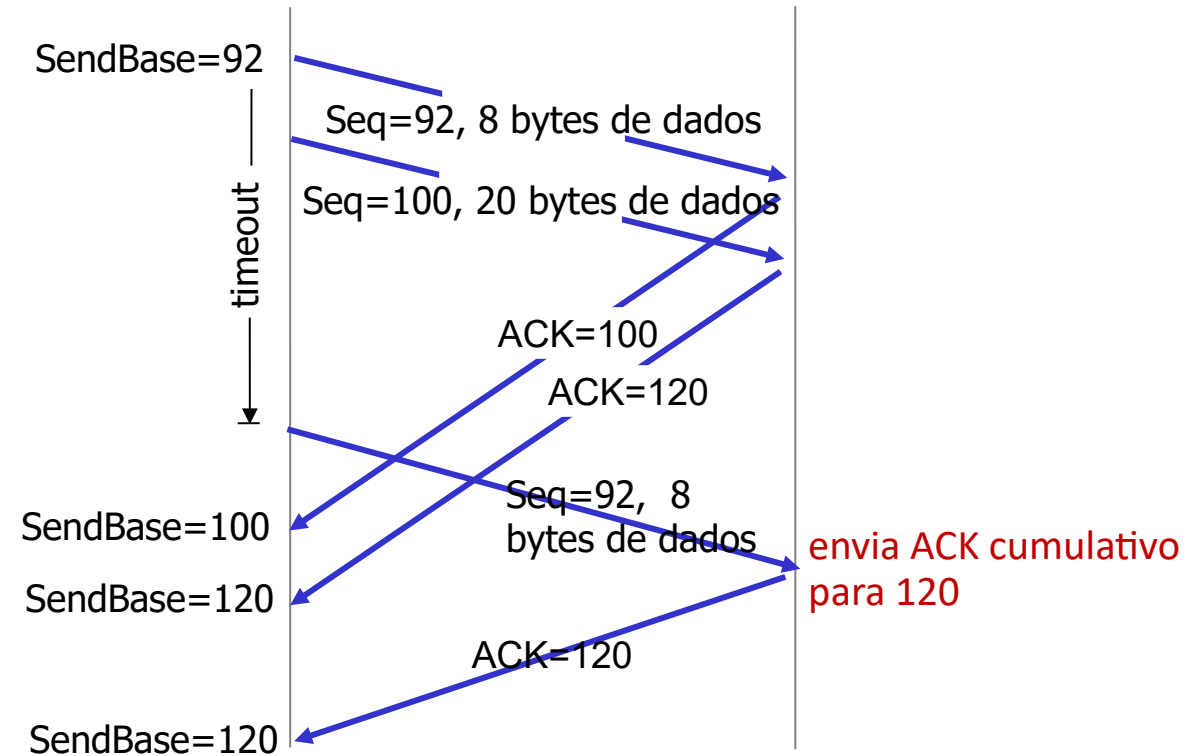
TCP: cenários de retransmissão

Hospedeiro A Hospedeiro B



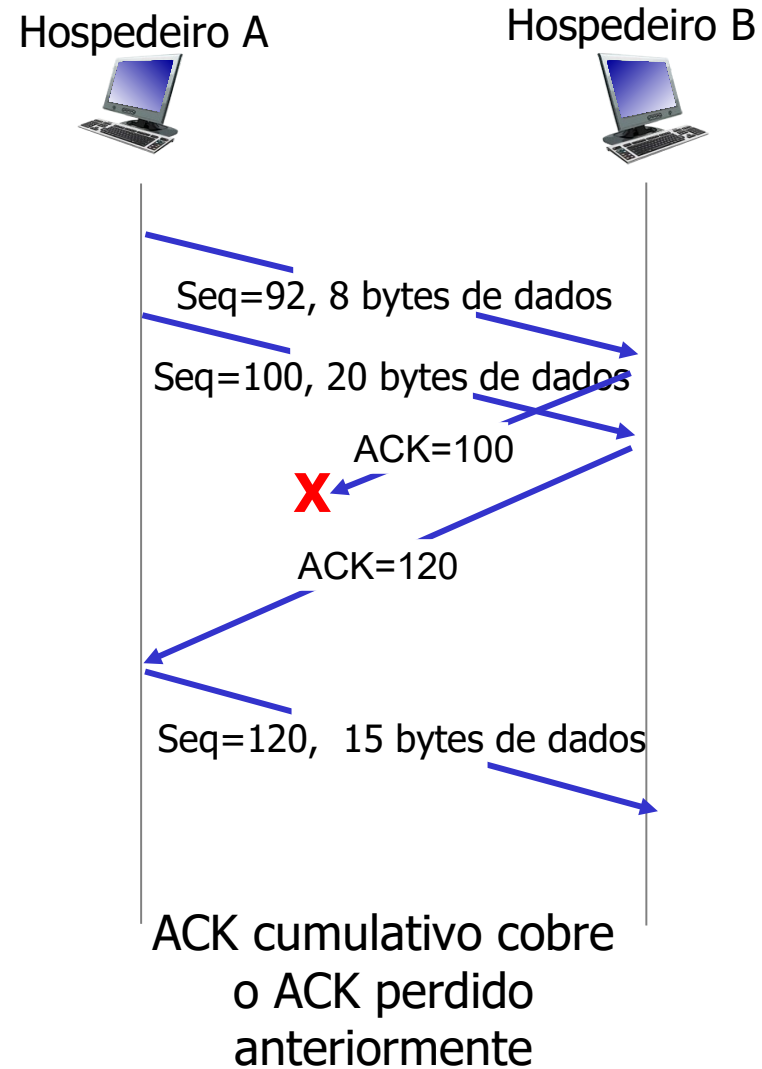
cenário de ACK perdido

Hospedeiro A Hospedeiro B



timeout prematuro

TCP: cenários de retransmissão



Retransmissão rápida do TCP

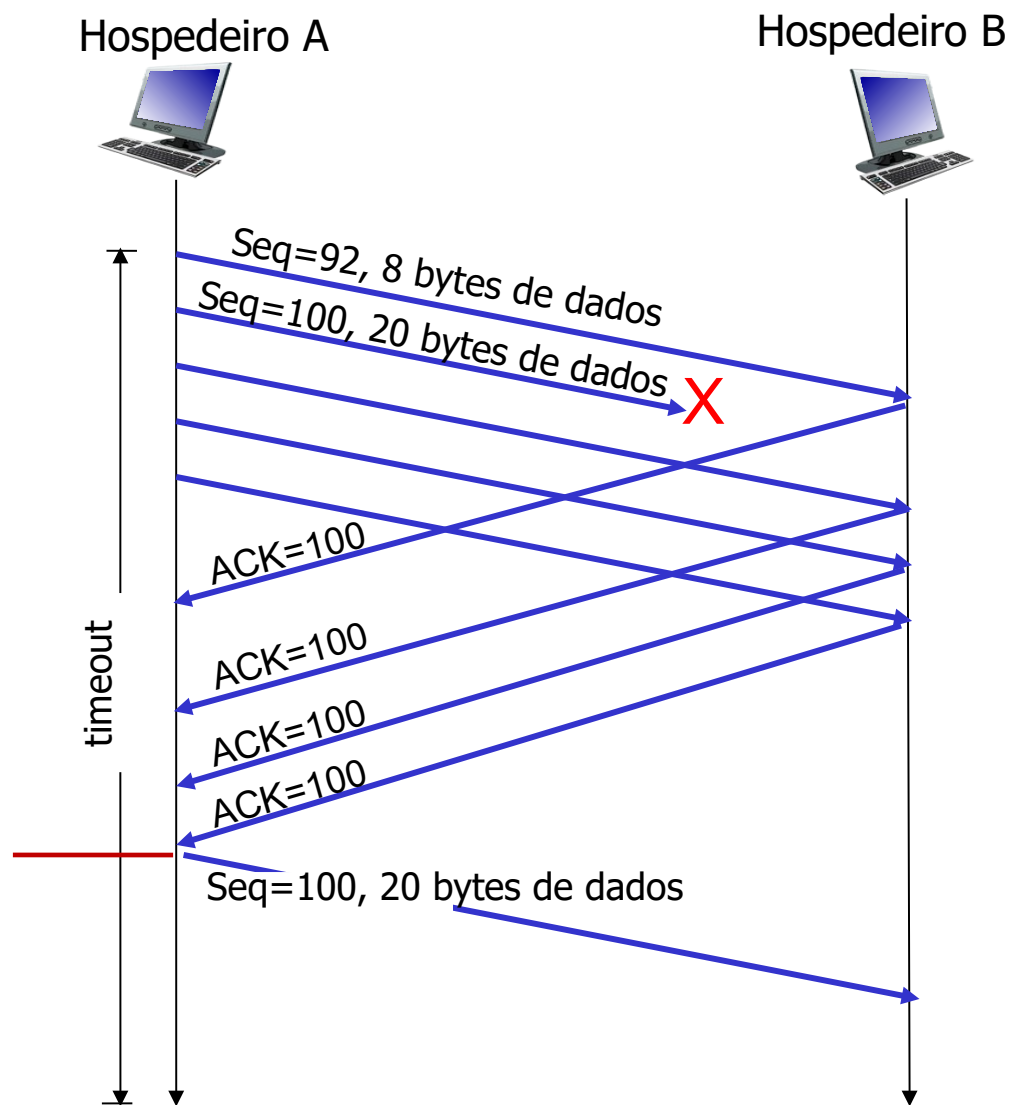
retransmissão rápida do TCP

se o remetente receber 3 ACKs adicionais para os mesmos dados (“ACK quadruplicado”), reenvia segmento não reconhecido com menor número de sequência

- é provável que o segmento não reconhecido foi perdido, portanto não espera pelo timeout



O recebimento de três ACKs duplicados indica que 3 segmentos foram recebidos após um segmento ausente – é provável que o segmento esteja perdido. Então, retransmita!



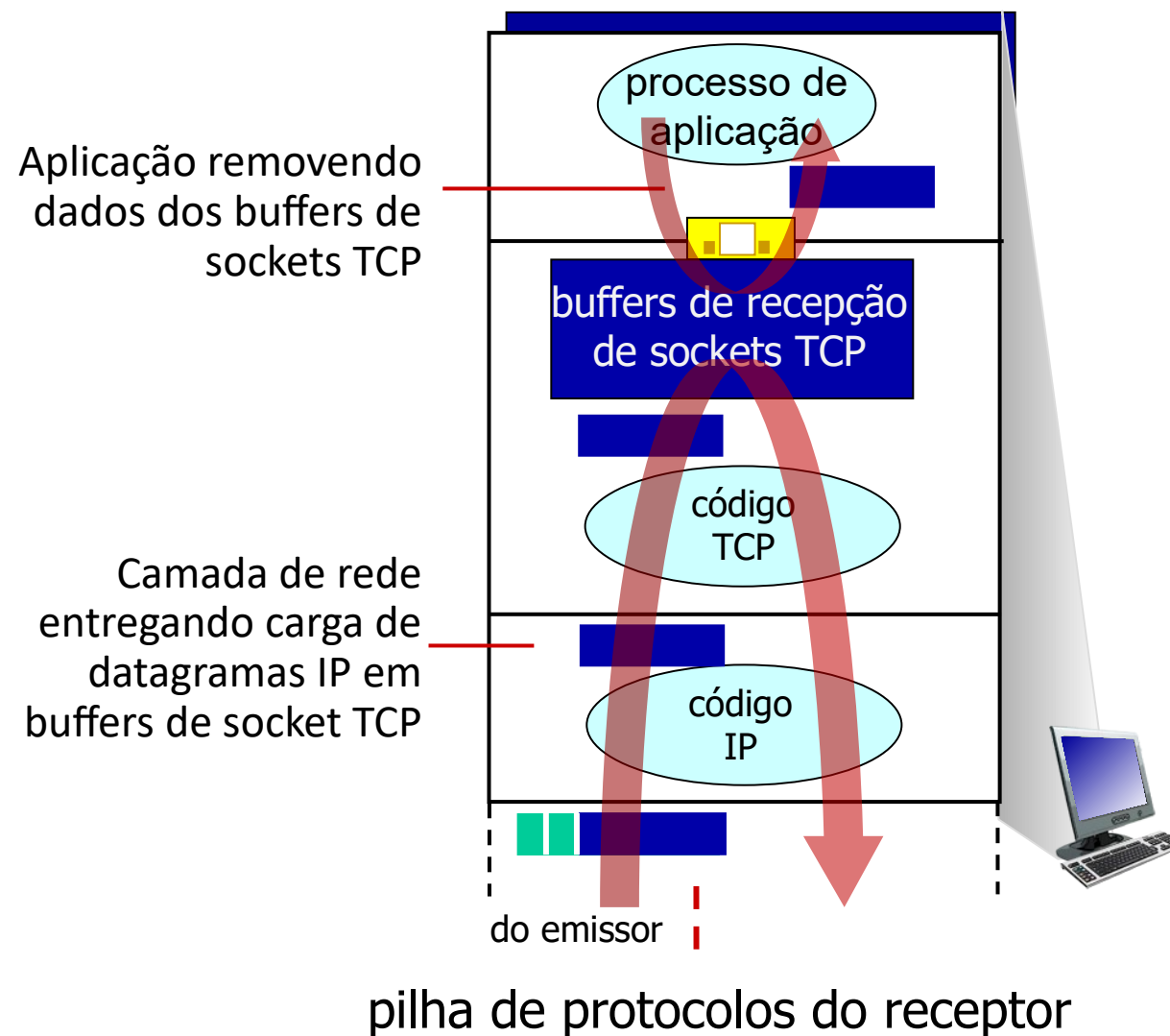
Camada de transporte: roteiro

- Serviços da camada de transporte
- Multiplexação e demultiplexação
- Transporte sem conexão: UDP
- Princípios da transferência confiável de dados
- **Transporte orientado a conexão: TCP**
 - estrutura do segmento
 - transferência confiável de dados
 - controle de fluxo
 - gerenciamento de conexão
 - Princípios de controle de congestionamentos
 - Controle de congestionamento do TCP
- Evolução da funcionalidade da camada de transporte



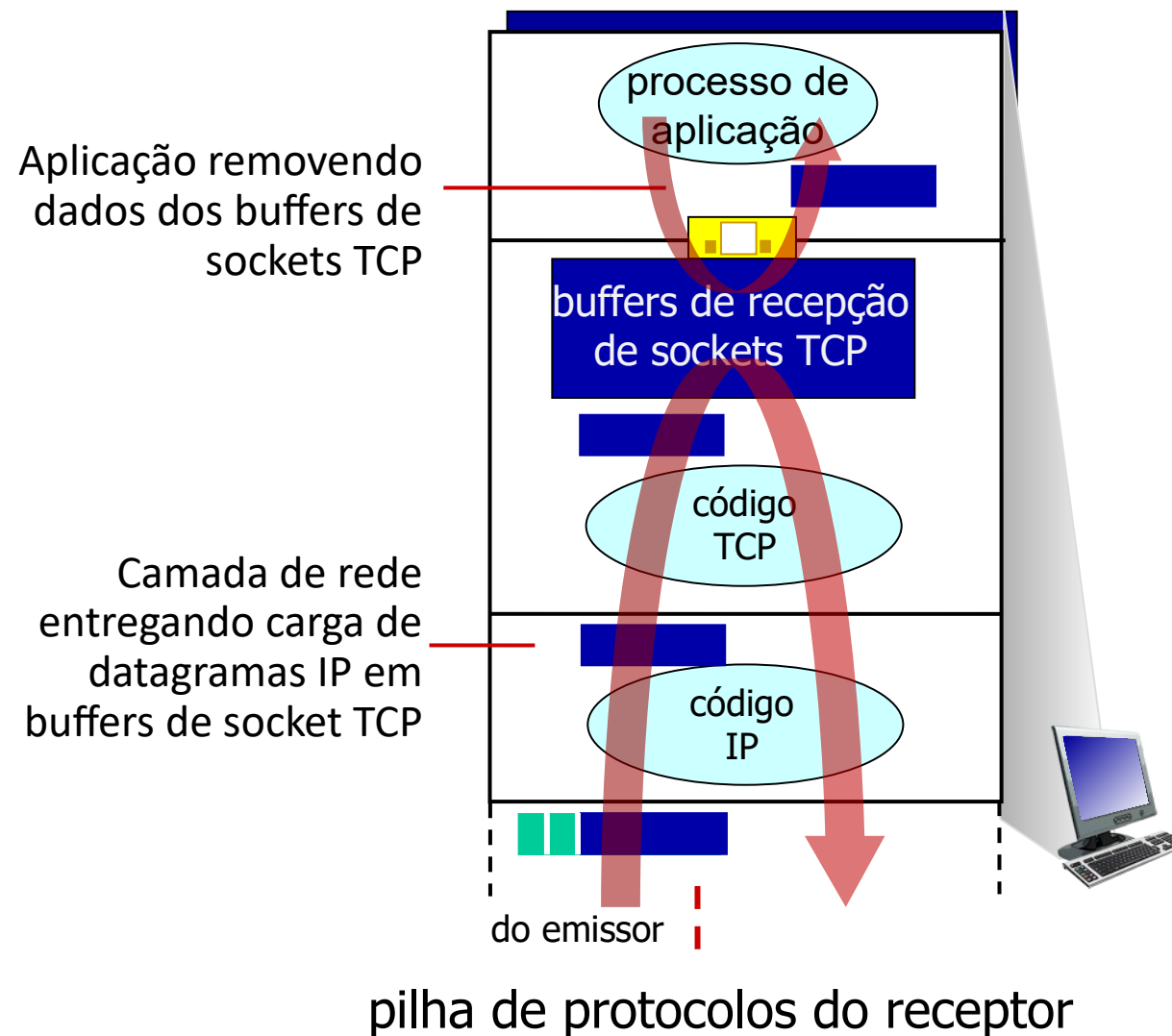
Controle de fluxo do TCP

Q: O que acontece se a camada de rede fornecer dados mais rápido do que a camada de aplicação remove dados dos buffers de socket?



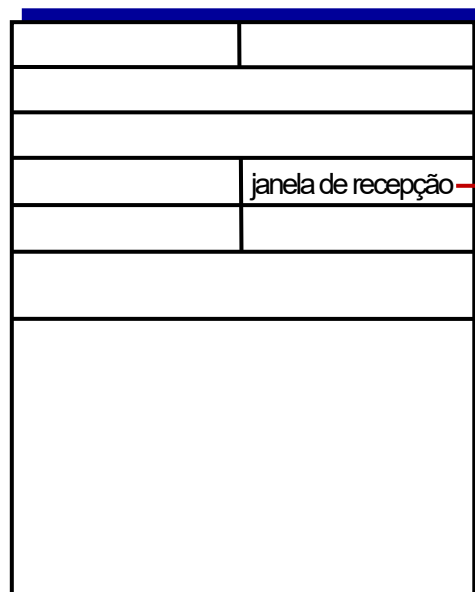
Controle de fluxo do TCP

Q: O que acontece se a camada de rede fornecer dados mais rápido do que a camada de aplicação remove dados dos buffers de socket?



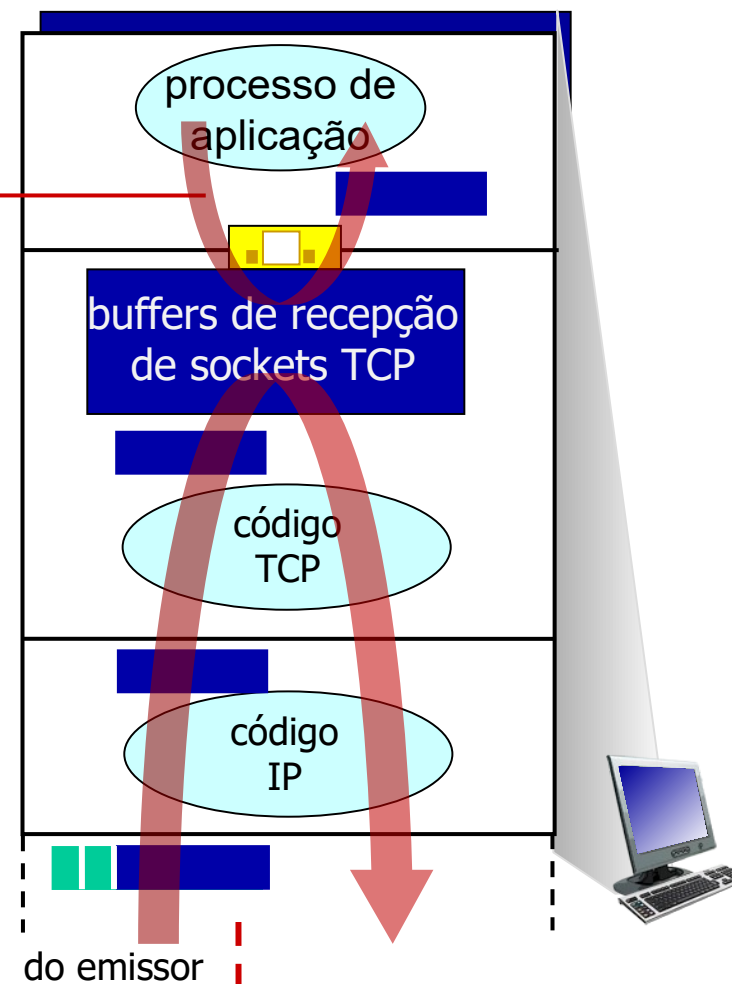
Controle de fluxo do TCP

Q: O que acontece se a camada de rede fornecer dados mais rápido do que a camada de aplicação remove dados dos buffers de socket?



controle de fluxo: número de bytes que o receptor está disposto a aceitar

Aplicação removendo dados dos buffers de sockets TCP



pilha de protocolos do receptor

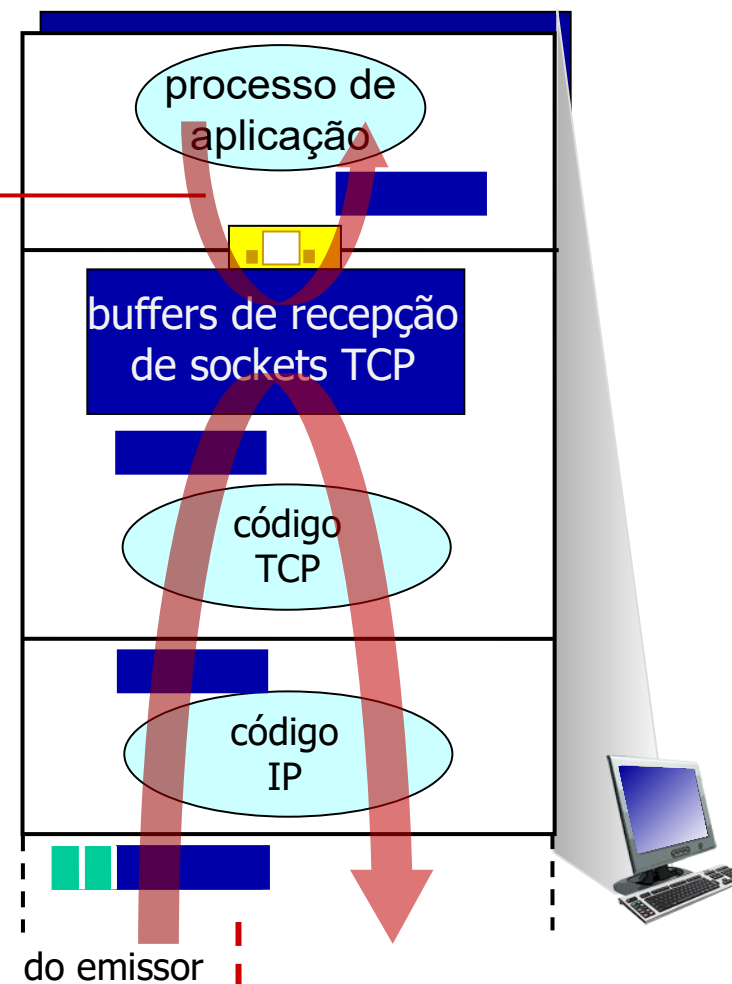
Controle de fluxo do TCP

Q: O que acontece se a camada de rede fornecer dados mais rápido do que a camada de aplicação remove dados dos buffers de socket?

controle de fluxo

receptor controla emissor, para que o emissor não transborde o buffer do receptor transmitindo muito e muito rápido

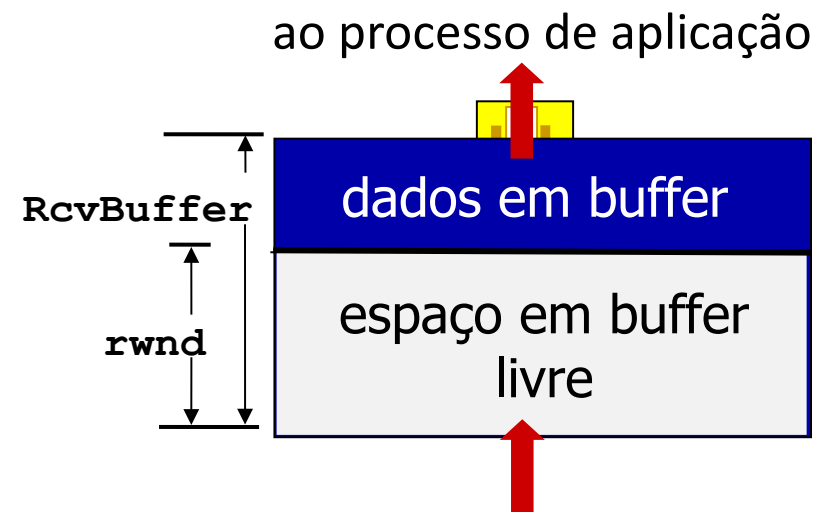
Aplicação removendo dados dos buffers de sockets TCP



pilha de protocolos do receptor

Controle de fluxo do TCP

- receptor TCP “anuncia” espaço livre em buffer no campo **rwnd** do cabeçalho do TCP
 - tamanho de **RcvBuffer** é configurado via opções de socket (padrão típico é 4096 bytes)
 - muitos sistemas operacionais autoajustam o **RcvBuffer**
- emissor limita a quantidade de dados não reconhecidos (“em voo”) para **rwnd**
- garante que o buffer de recepção não irá transbordar

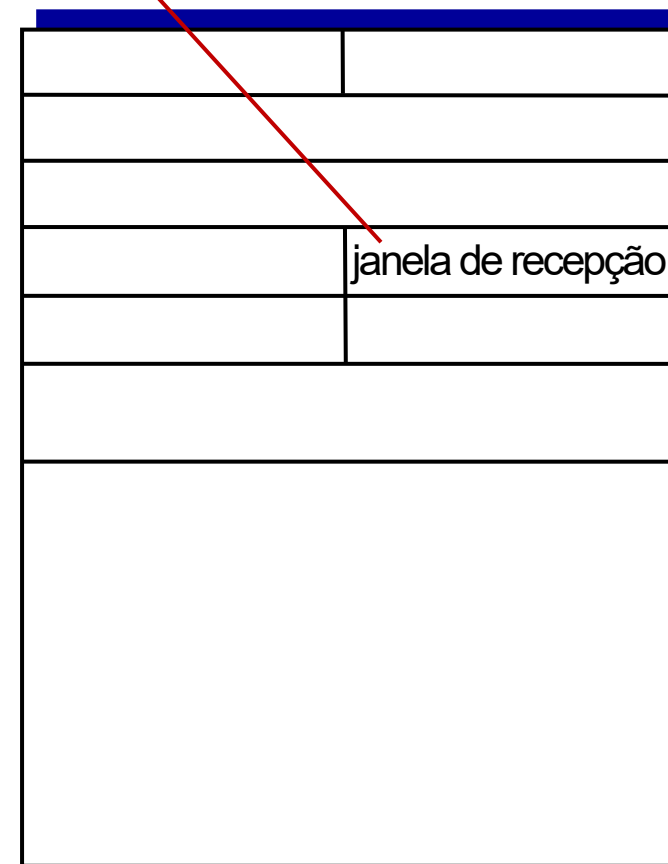


Buffer do lado do receptor TCP

Controle de fluxo do TCP

- receptor TCP “anuncia” espaço livre em buffer no campo **rwnd** do cabeçalho do TCP
 - tamanho de **RcvBuffer** é configurado via opções de socket (padrão típico é 4096 bytes)
 - muitos sistemas operacionais autoajustam o **RcvBuffer**
- emissor limita a quantidade de dados não reconhecidos (“em voo”) para **rwnd**
- garante que o buffer de recepção não irá transbordar

controle de fluxo: número de bytes que o receptor está disposto a aceitar



formato de segmento TCP

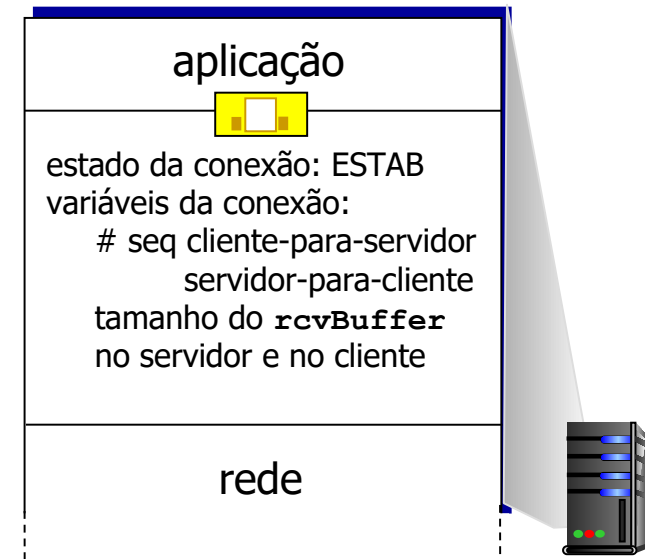
Gerenciamento de conexão TCP

antes de trocar dados, emissor e receptor fazem o “handshake”:

- concordam em estabelecer conexão (cada um conhecendo o desejo do outro de estabelecer uma conexão)
- concordam com parâmetros de conexão (ex.: números de sequência iniciais)



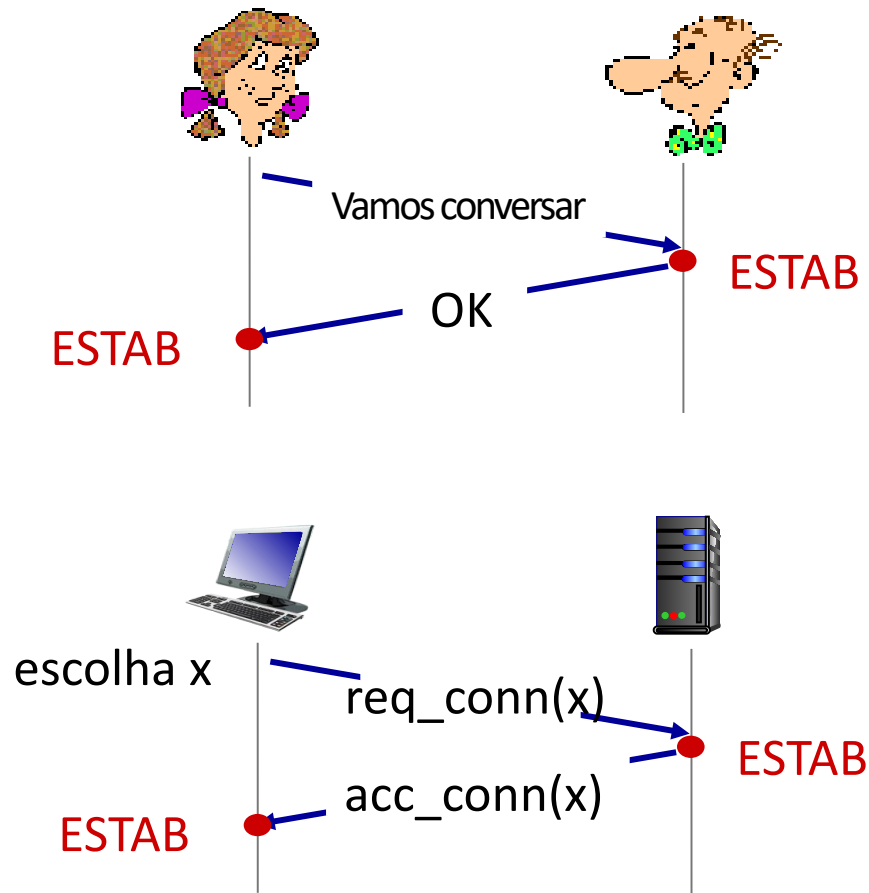
```
Socket clientSocket =  
    newSocket("hostname", "port number");
```



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

Concordando em estabelecer uma conexão

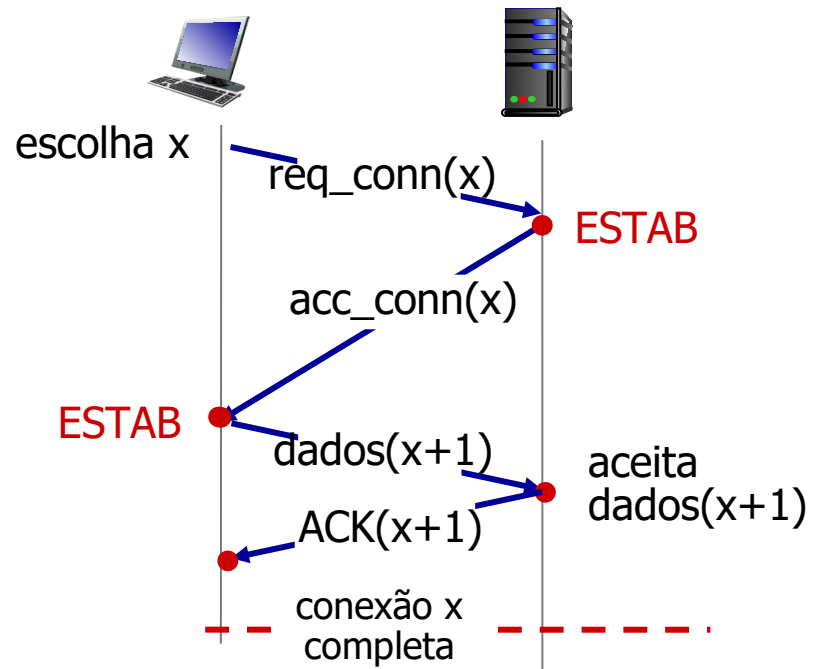
Handshake de 2 vias:



Q: o handshake de 2 vias
sempre funciona em rede?
atrasos variáveis
mensagens retransmitidas (por
exemplo, req_conn(x)) devido à
perda de mensagens
reordenação de mensagens

- não pode “ver” o outro lado

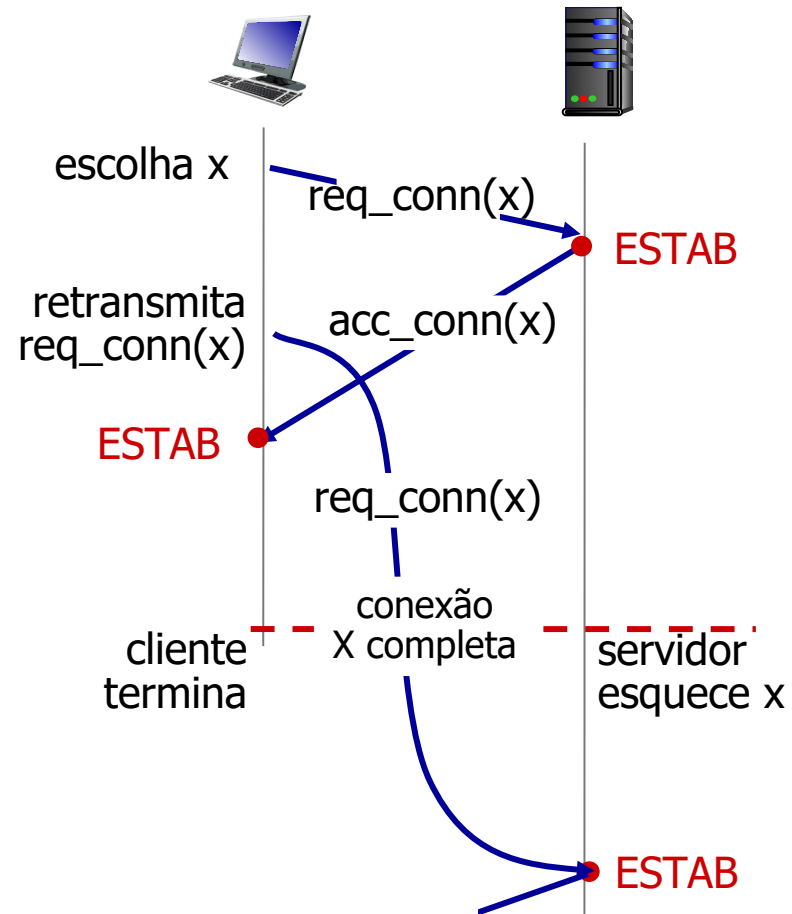
Cenários de handshake de 2 vias



Sem problema!

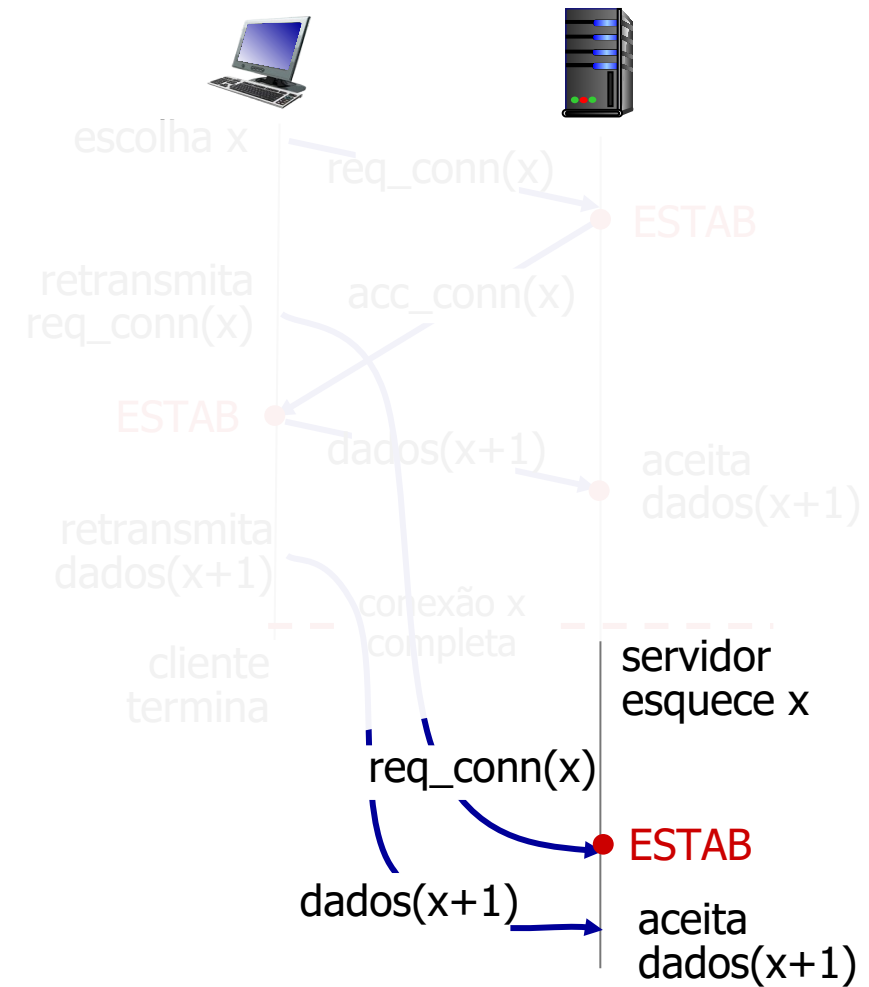



Cenários de handshake de 2 vias



Problema: conexão meio aberta! (sem cliente)

Cenários de handshake de 2 vias



 Problema: dados duplicados aceitos!

Handshake de 3 vias do TCP

Estado do cliente

```
clientSocket = socket(AF_INET, SOCK_STREAM)
```

LISTEN

```
clientSocket.connect((serverName, serverPort))
```

SYNSENT

ESTAB

SYNACK(x) recebido indica que servidor está vivo; envia ACK para SYNACK; este segmento pode conter dados cliente-para-servidor

escolhe número de sequência inicial, x
envia mensagem TCP SYN

SYNbit=1, Seq=x

SYNbit=1, Seq=y
ACKbit=1; ACKnum=x+1

ACKbit=1, ACKnum=y+1

ACK(y) recebido indica que cliente está vivo

Estado do servidor

```
serverSocket = socket(AF_INET, SOCK_STREAM)  
serverSocket.bind('', serverPort)  
serverSocket.listen(1)  
connectionSocket, addr = serverSocket.accept()
```

LISTEN

SYN RCVD

ESTAB

escolhe número de sequência inicial, y
envia mensagem TCP SYNACK, reconhecendo SYN

Camada de transporte: roteiro

- Serviços da camada de transporte
- Multiplexação e demultiplexação
- Transporte sem conexão: UDP
- Princípios da transferência confiável de dados
- Transporte orientado a conexão: TCP
- Princípios de controle de congestionamentos
- Controle de congestionamento do TCP
- Evolução da funcionalidade da camada de transporte



Princípios de Controle de Congestionamentos

Congestionamento:

- informalmente: “muitas fontes enviando muitos dados mais rápido do que a *rede* consegue manipular”
- manifestações:
 - longos atrasos (filas em buffers de roteadores)
 - perda de pacotes (estouro de buffer em roteadores)
- diferente do controle de fluxo!
- um problema top-10!



controle de congestionamento:
muitos emissores, enviando
muito rápido

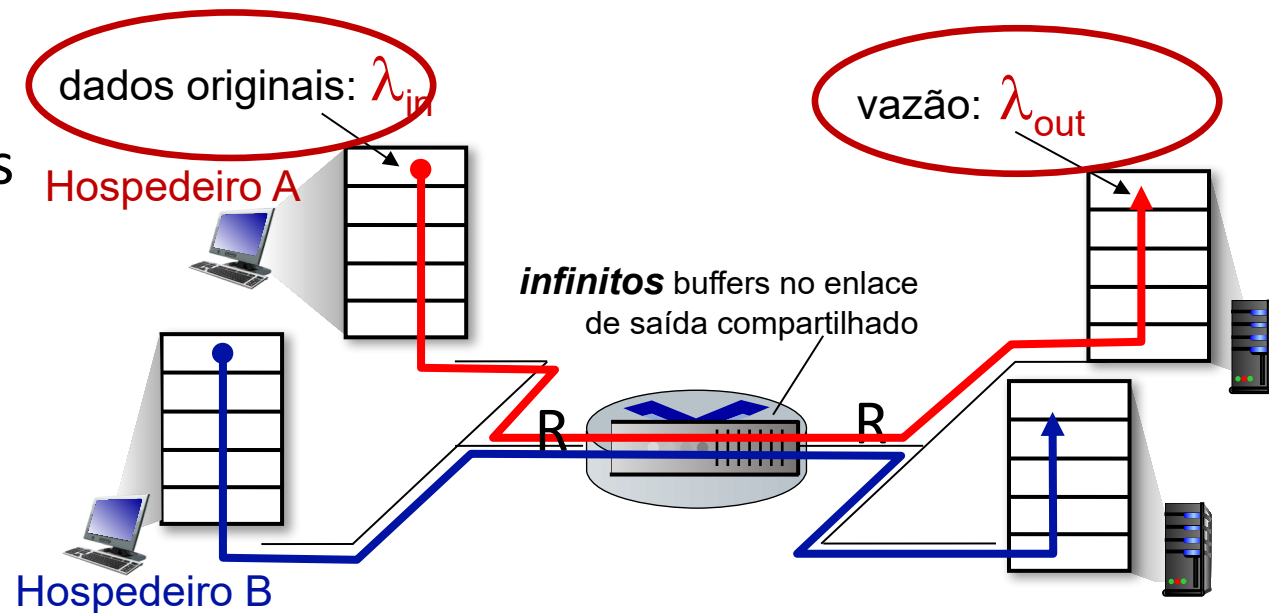
controle de fluxo: um
emissor muito rápido para um
receptor



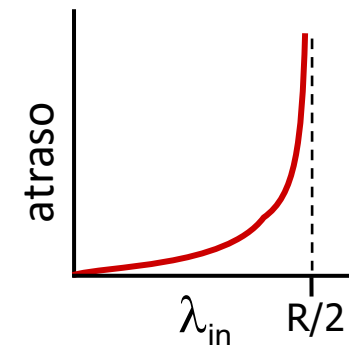
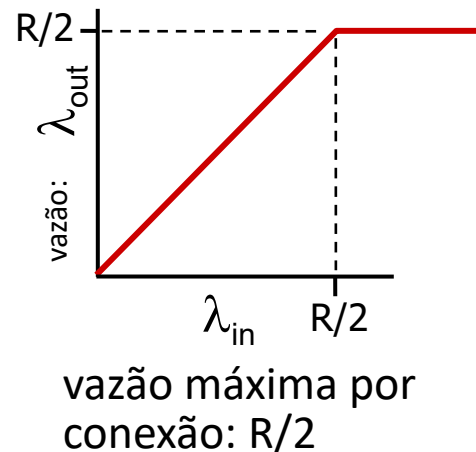
Causas/custos de congestionamento: cenário 1

Cenário mais simples:

- um roteador, buffers infinitos
- capacidade do enlace de entrada e saída: R
- dois fluxos
- não são necessárias retransmissões



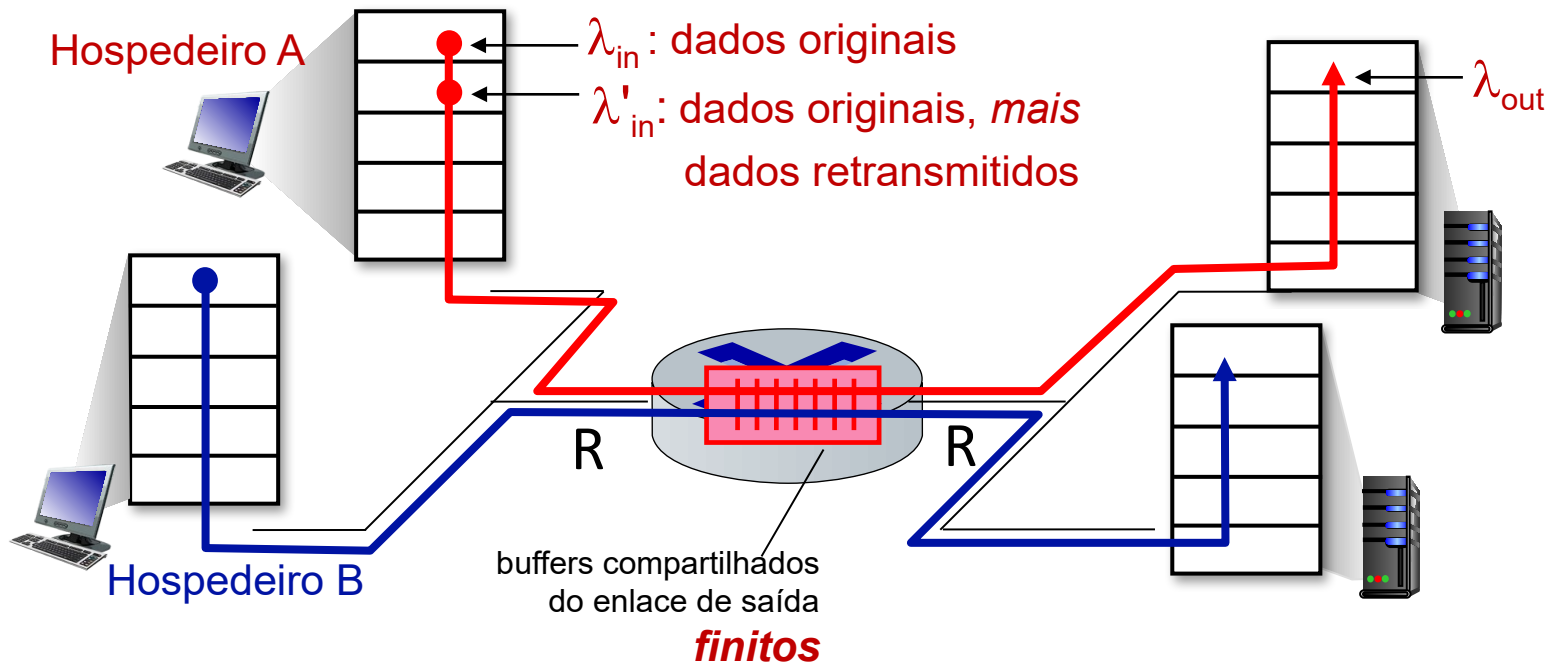
Q: O que acontece conforme a taxa de chegada λ_{in} se aproxima de $R/2$?



grandes atrasos conforme a taxa de chegada λ_{in} aproxima-se da capacidade máxima

Causas/custos de congestionamento: cenário 2

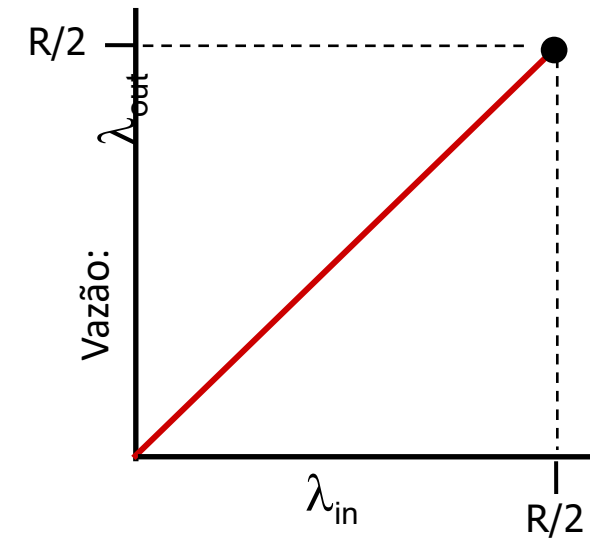
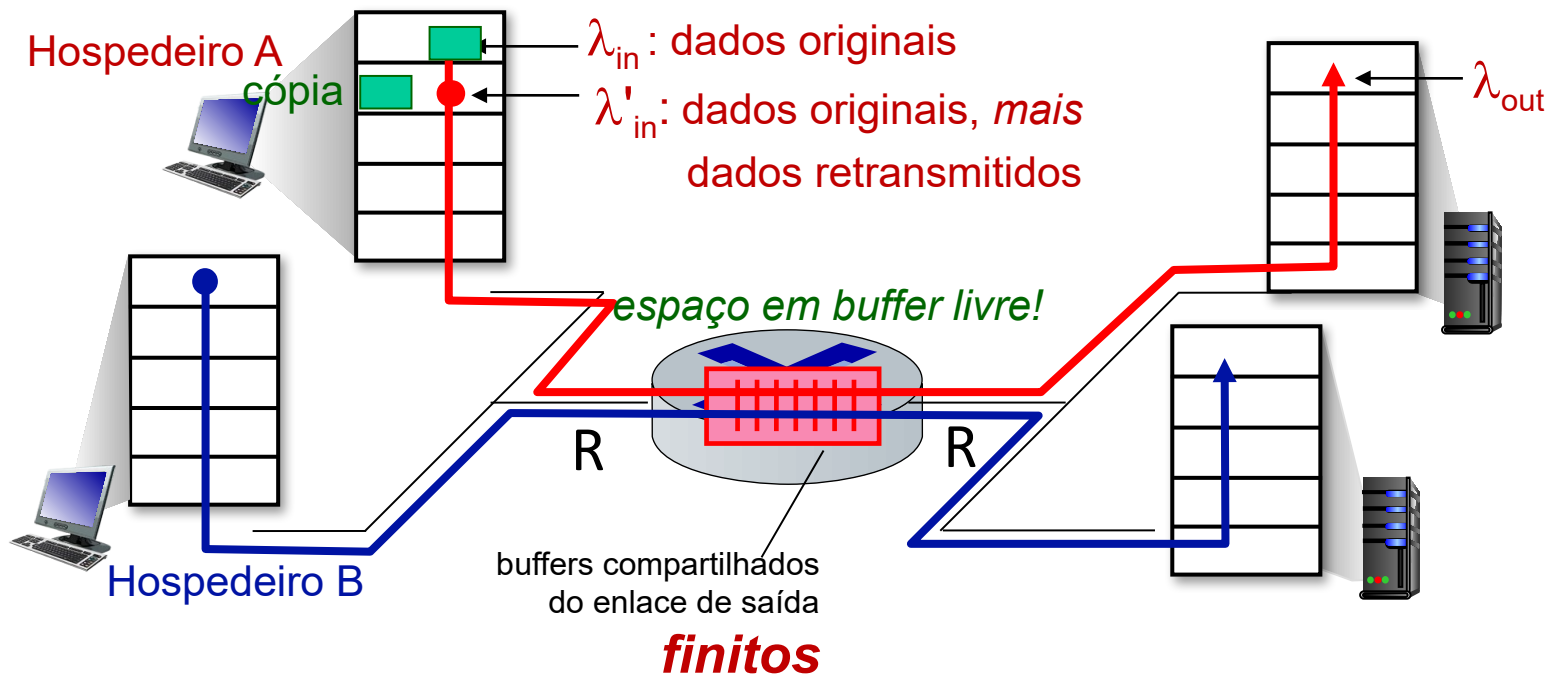
- um roteador, buffers *finitos*
- emissor retransmite pacotes perdidos e expirados
 - entrada da camada de aplicação = saída da camada de aplicação: $\lambda_{in} = \lambda_{out}$
 - entrada da camada de transporte inclui *retransmissões*: $\lambda'_{in} \geq \lambda_{in}$



Causas/custos de congestionamento: cenário 2

Idealização: conhecimento perfeito

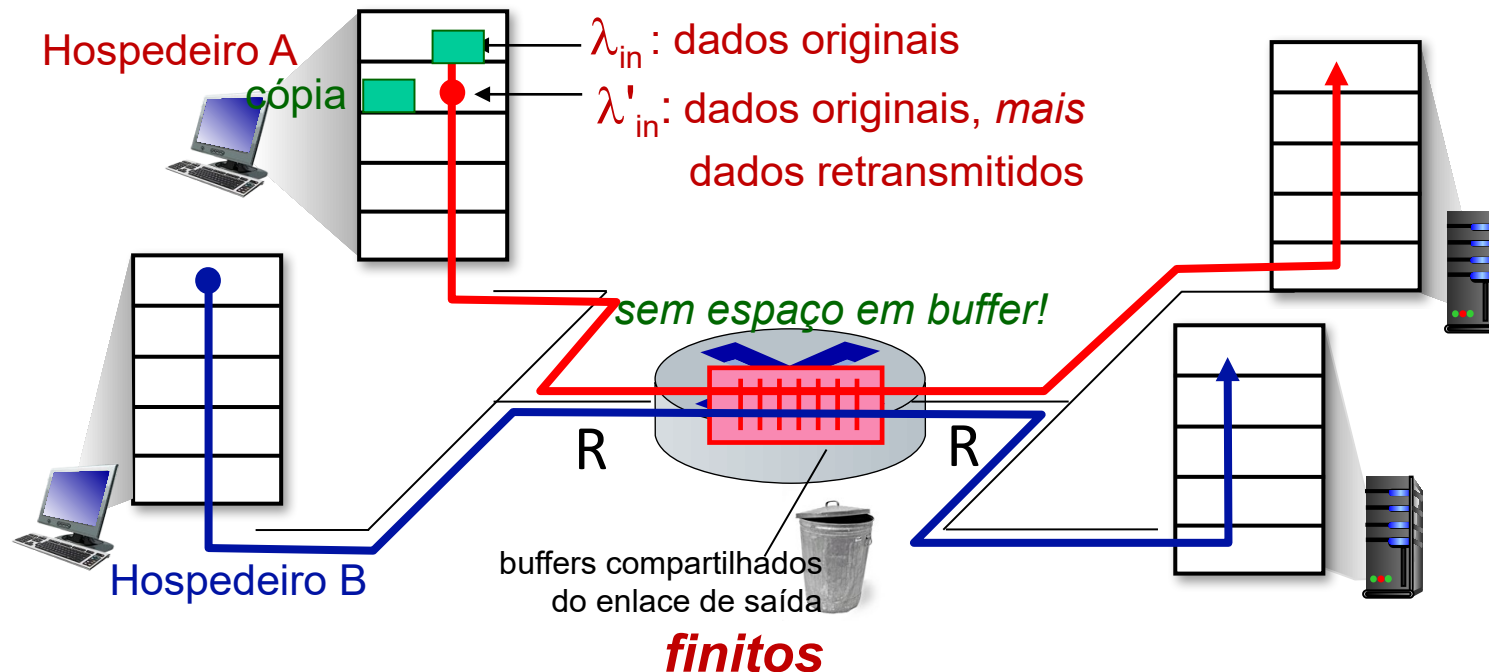
- emissor envia apenas quando há buffer disponível no roteador



Causas/custos de congestionamento: cenário 2

Idealização: *algum* conhecimento perfeito

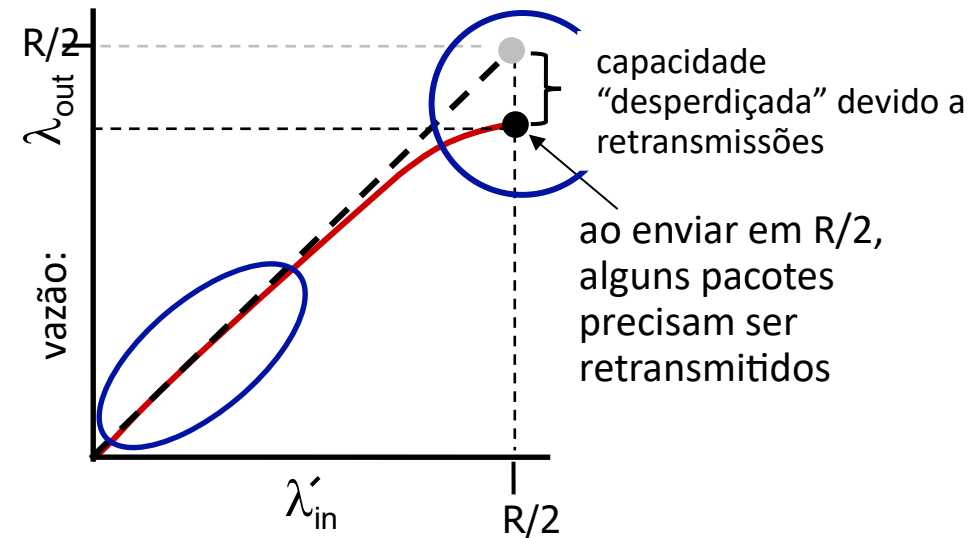
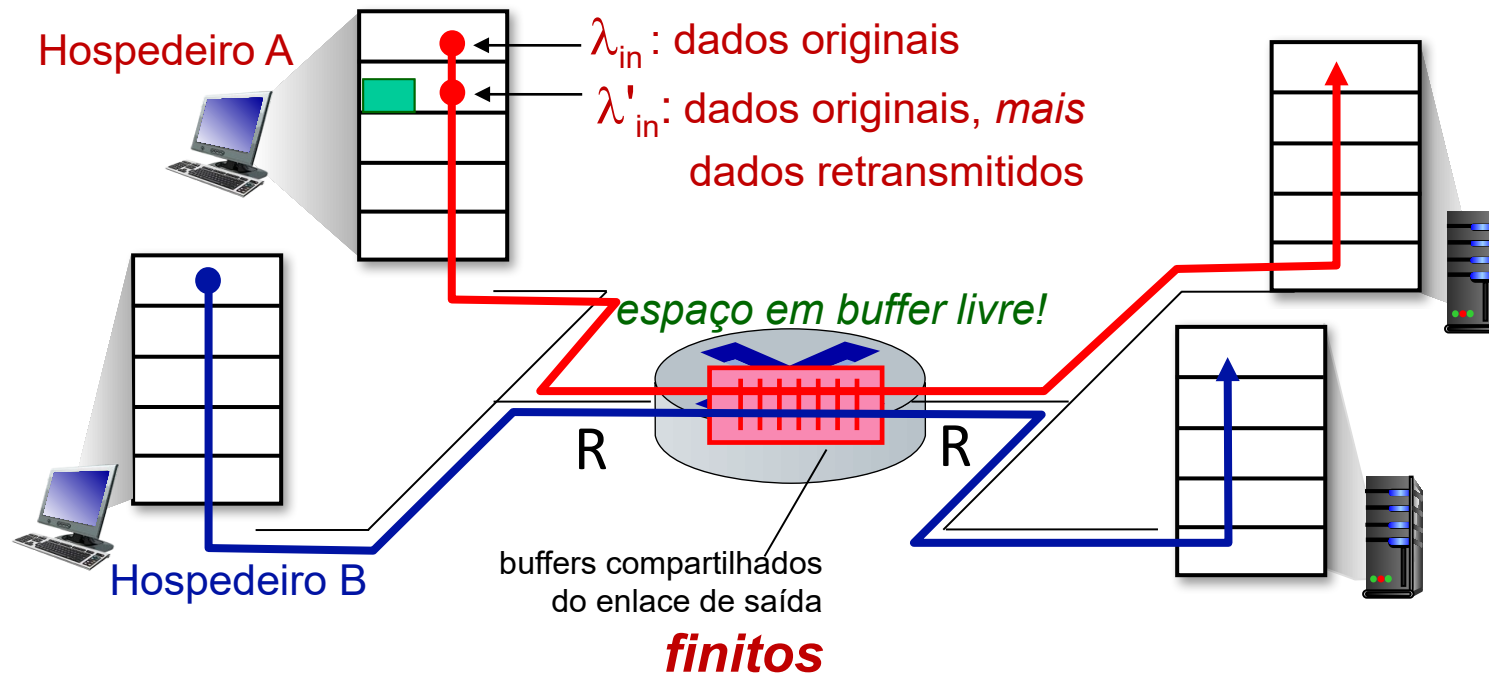
- pacotes podem ser perdidos (descartados no roteador) devido a buffers cheios
- remetente sabe quando o pacote foi descartado: só reenvia se *sabe* que o pacote foi perdido



Causas/custos de congestionamento: cenário 2

Idealização: *algum* conhecimento perfeito

- pacotes podem ser perdidos (descartados no roteador) devido a buffers cheios
- remetente sabe quando o pacote foi descartado: só reenvia se *sabe* que o pacote foi perdido

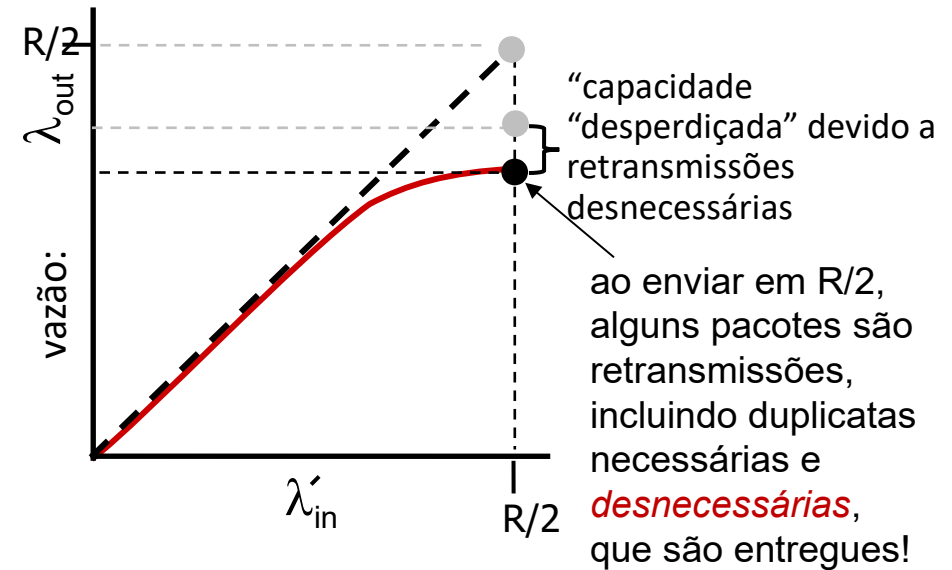
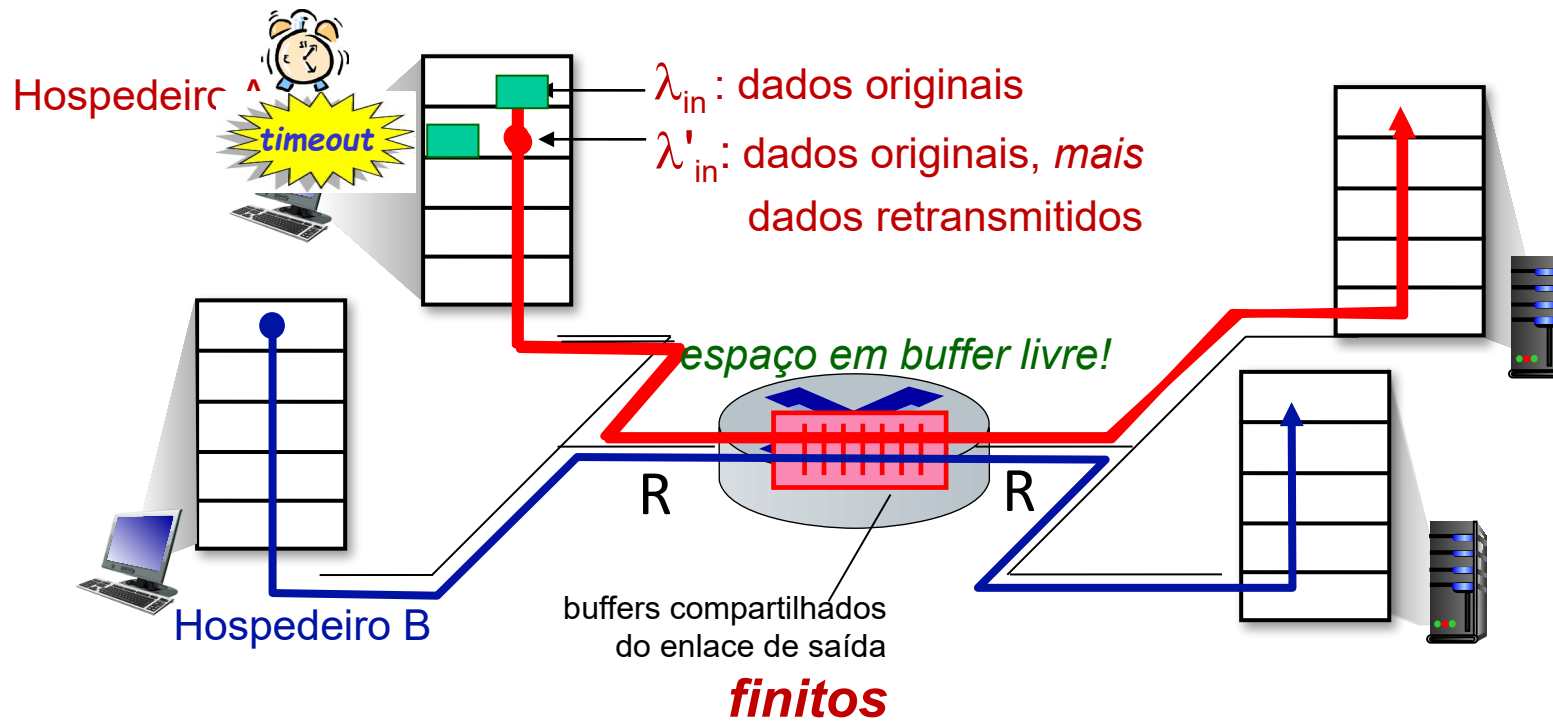


Causas/custos de congestionamento: cenário 2

Cenário realista: *duplicatas não necessárias*

pacotes podem ser perdidos, descartados no roteador devido a buffers cheios – exigindo retransmissões

- mas os temporizadores do emissor podem se esgotar prematuramente, enviando *duas* cópias, sendo *ambas* entregues

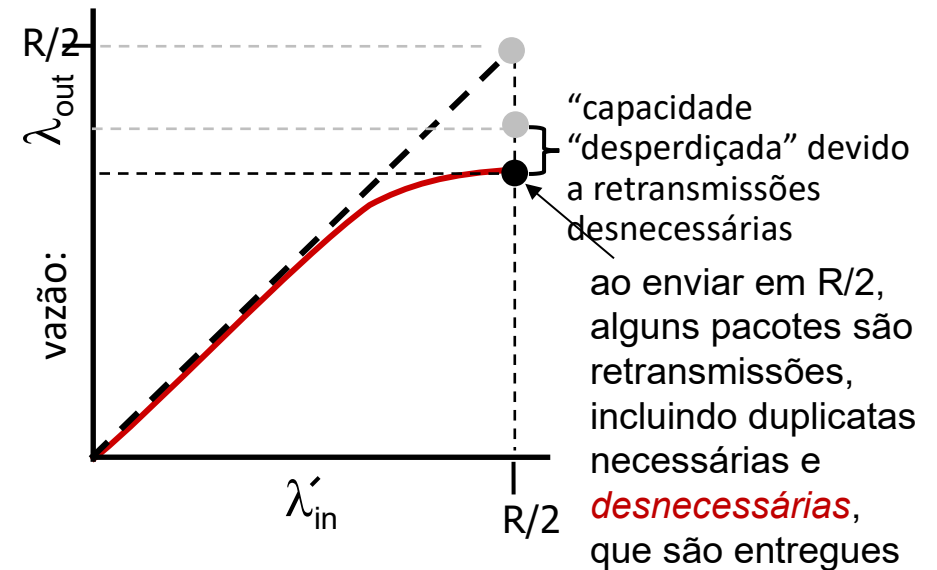


Causas/custos de congestionamento: cenário 2

Cenário realista: *duplicatas não necessárias*

pacotes podem ser perdidos, descartados no roteador devido a buffers cheios – exigindo retransmissões

- mas os temporizadores do emissor podem se esgotar prematuramente, enviando *duas* cópias, sendo *ambas* entregues



"custos" do congestionamento:

mais trabalho (retransmissões) para dada vazão do receptor

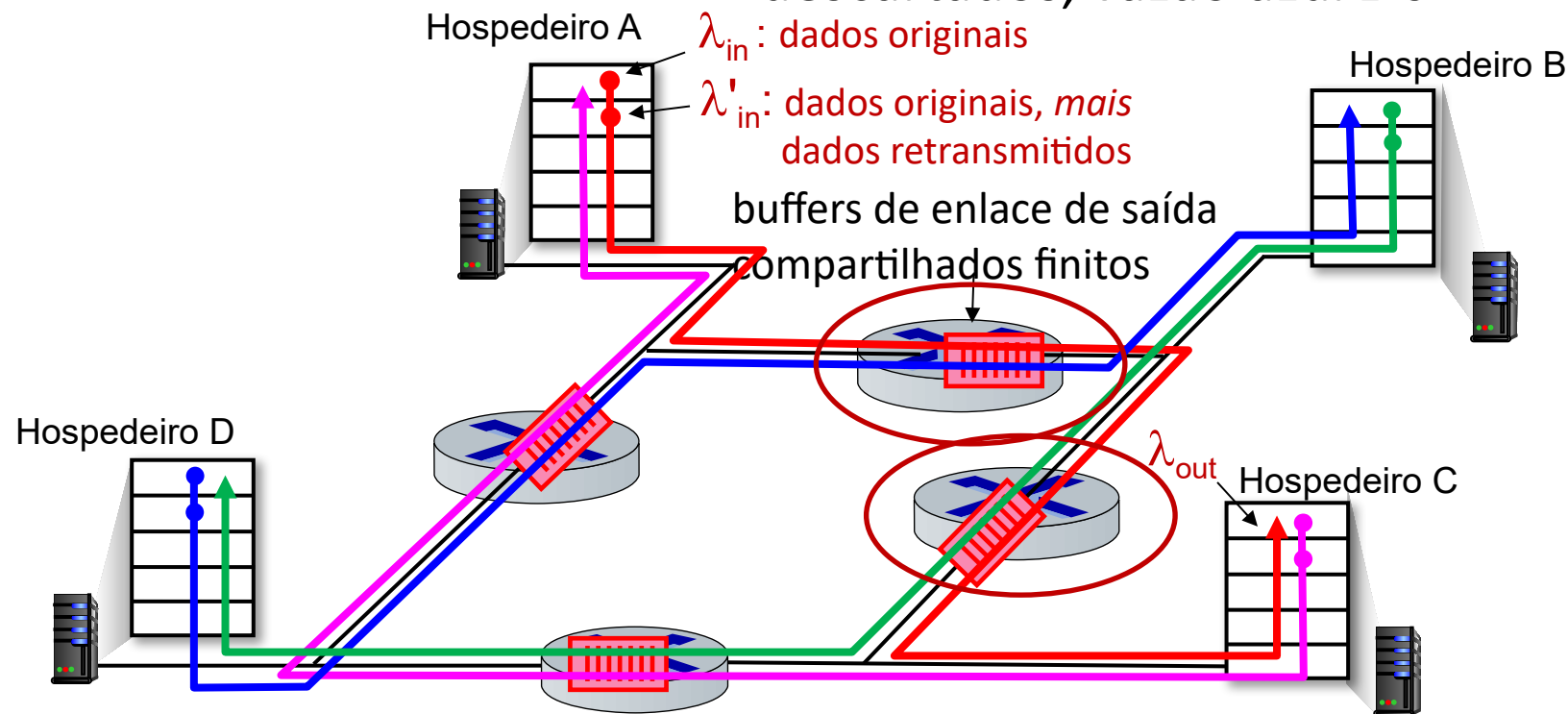
- retransmissões desnecessárias: enlace carrega várias cópias de um pacote
 - diminuindo a vazão máxima alcançável

Causas/custos de congestionamento: cenário 3

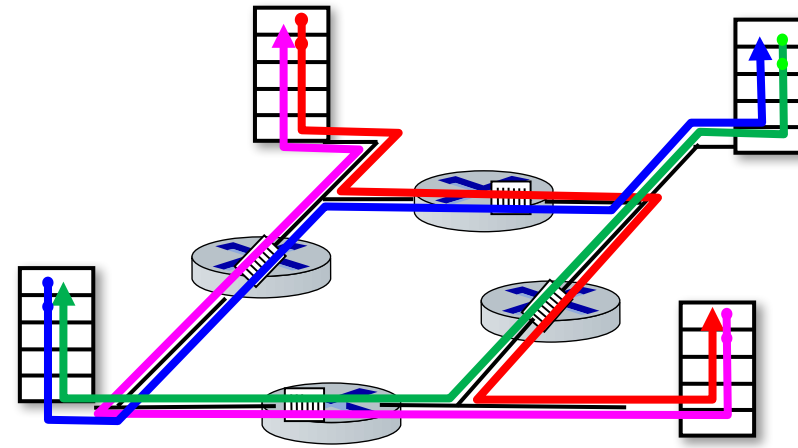
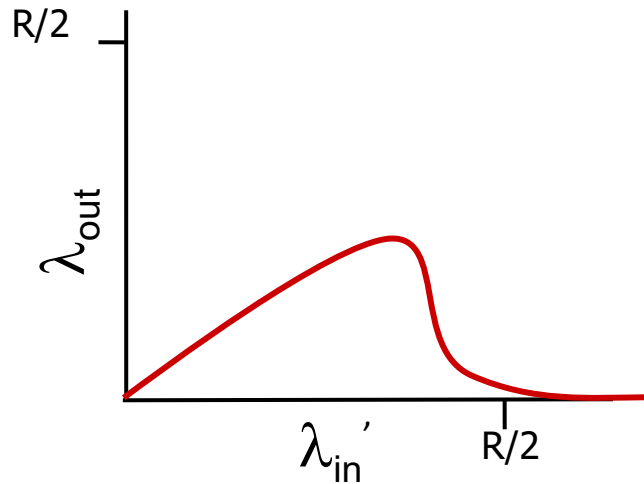
- *quatro* emissores
- caminhos *multi-salto*
- timeout/retransmissão

Q: o que acontece conforme λ_{in} e λ'_{in} aumentam?

R: conforme o λ'_{in} vermelho aumenta, todos os pacotes azuis chegando na fila superior são descartados, vazão azul $\rightarrow 0$



Causas/custos de congestionamento: cenário 3

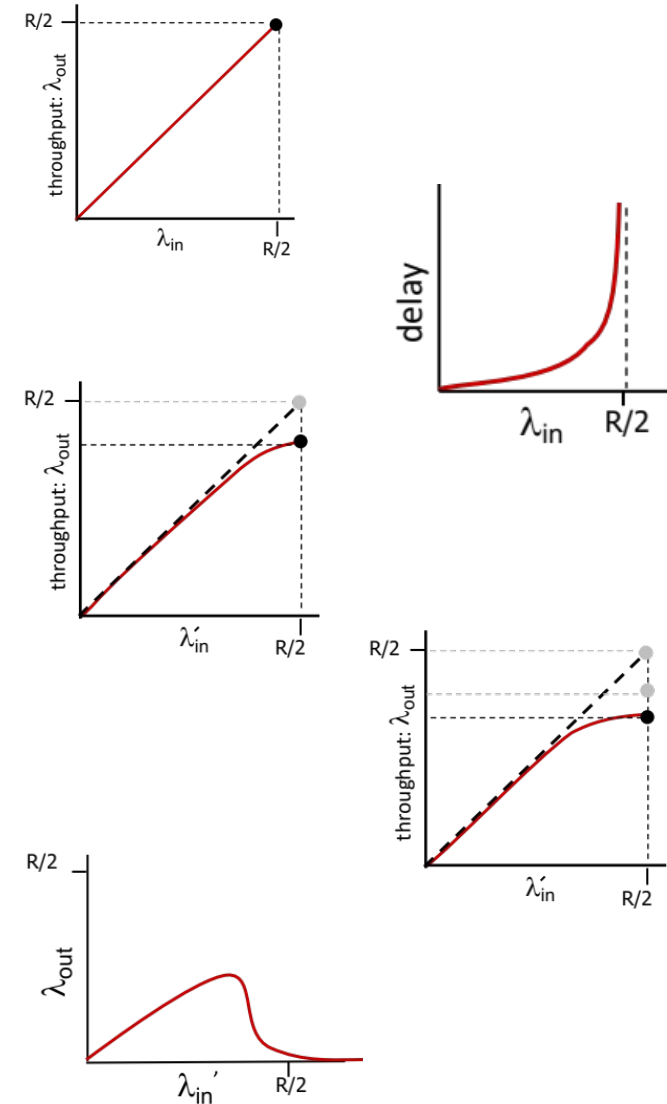


outro “custo” do congestionamento:

- quando um pacote é perdido, qualquer capacidade de transmissão upstream e de buffer utilizadas para aquele pacote foram desperdiçadas!

Causas/custos de congestionamento: insights

- vazão nunca pode exceder a capacidade máxima
- atraso aumenta à medida que a capacidade máxima se aproxima
- perda/retransmissão diminui vazão efetiva
- duplicações desnecessárias diminuem ainda mais a vazão efetiva
- capacidade de transmissão upstream e buffers são desperdiçados para pacotes perdidos no downstream

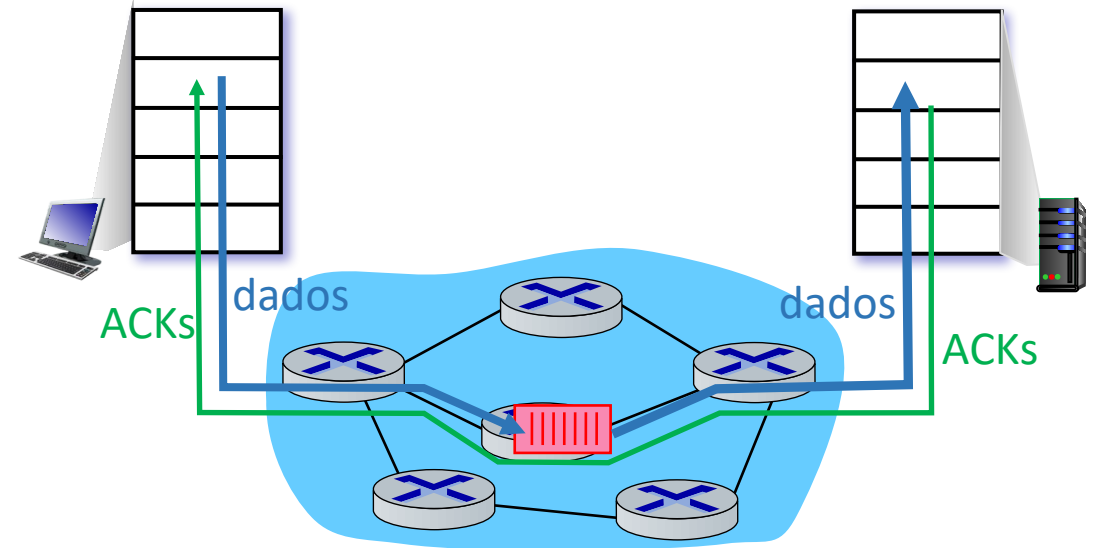


Abordagens para o controle de congestionamento

Controle de congestionamento fim-a-fim:

nenhum retorno explícito da rede

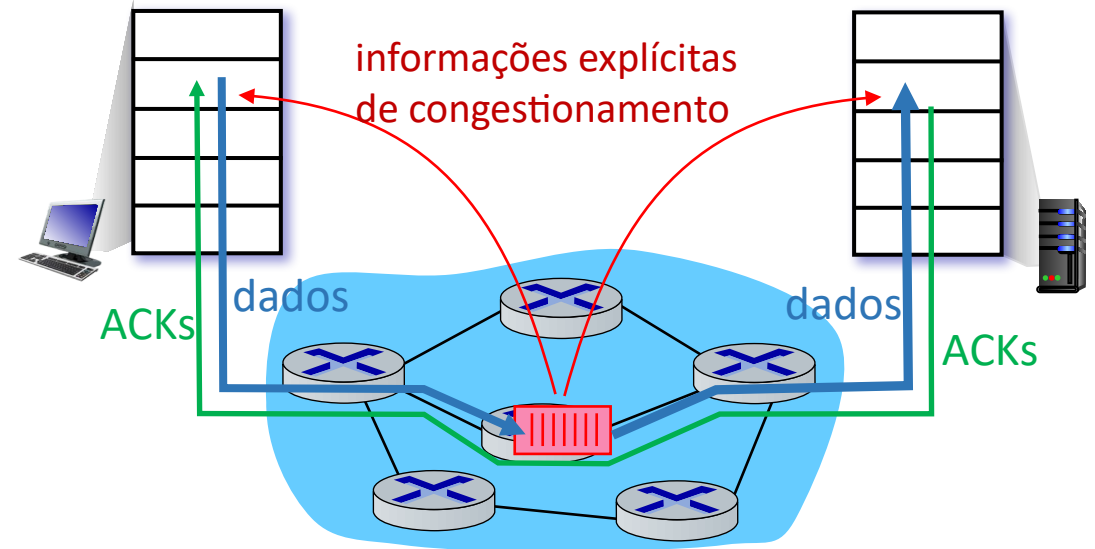
- congestionamento *inferido* das perdas e atrasos observados
- abordagem tomada pelo TCP



Abordagens para o controle de congestionamento

Controle de congestionamento assistido por rede:

- roteadores fornecem retorno *direto* para hospedeiros enviando/recebendo com fluxos passando por um roteador congestionado
- pode indicar nível de congestionamento ou definir explicitamente taxa de envio
- protocolos TCP ECN, ATM, DECbit



Camada de transporte: roteiro

- Serviços da camada de transporte
- Multiplexação e demultiplexação
- Transporte sem conexão: UDP
- Princípios da transferência confiável de dados
- Transporte orientado a conexão: TCP
- Princípios de controle de congestionamentos
- **Controle de congestionamento do TCP**
- Evolução da funcionalidade da camada de transporte



Controle de Congestionamento do TCP: AIMD

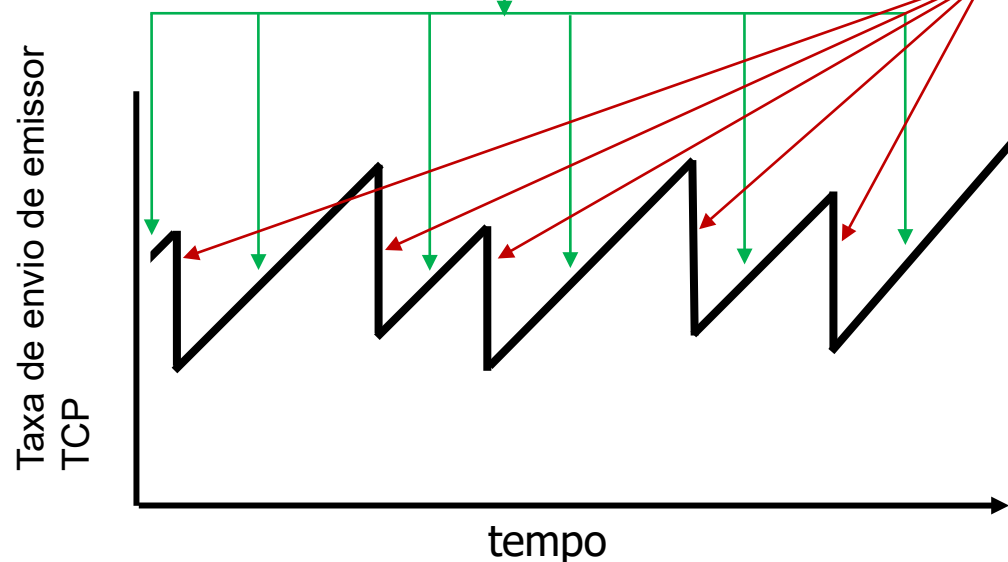
- *abordagem*: os emissores podem aumentar a taxa de envio até que ocorra perda de pacote (congestionamento) e, em seguida, diminuir a taxa de envio no evento de perda

Additive Increase
(incremento aditivo)

aumentar a taxa de envio em 1 tamanho máximo do segmento a cada RTT até que uma perda seja detectada

Multiplicative Decrease
(decremento multiplicativo)

reduzir taxa de envio pela metade em cada evento de perda



comportamento de
dente de serra do
AIMD: *sondagem*
por largura de
banda

AIMD do TCP : mais

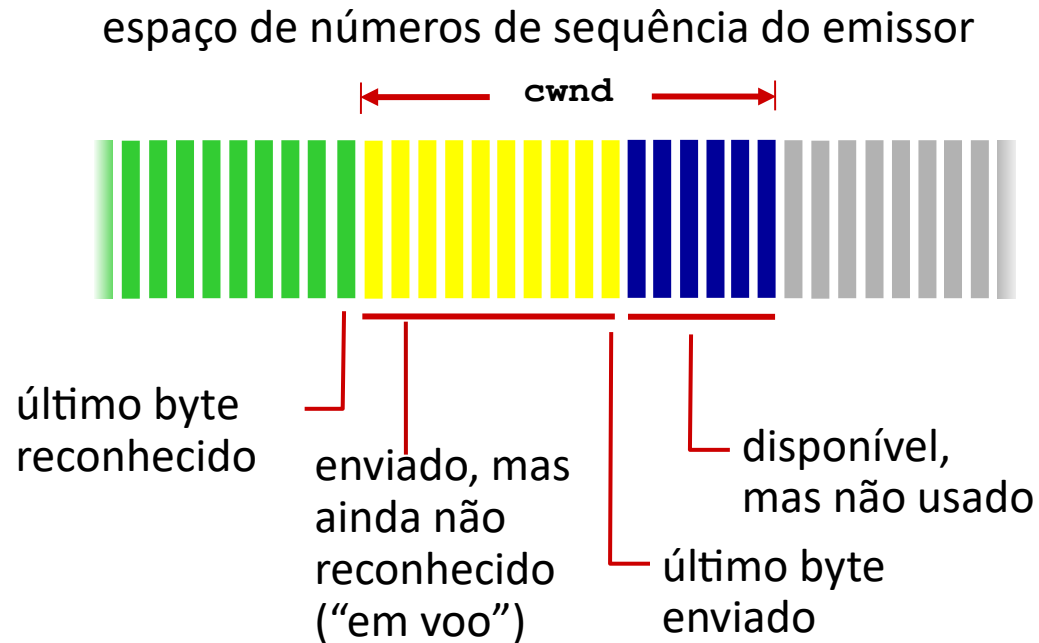
Detalhe do *decremento multiplicativo*: taxa de envio é Cortada pela metade em perda detectada pelo ACK triplamente duplicado (TCP Reno)

- Cortada para 1 MSS (tamanho máximo do segmento) quando uma perda é detectada por tempo limite (TCP Tahoe)

Por que AIMD?

- AIMD – um algoritmo distribuído e assíncrono – tem sido mostrado que:
 - otimiza taxas de fluxo congestionadas em toda a rede!
 - têm propriedades de estabilidade desejáveis

Controle de Congestionamento do TCP: detalhes



Comportamento de envio do TCP:

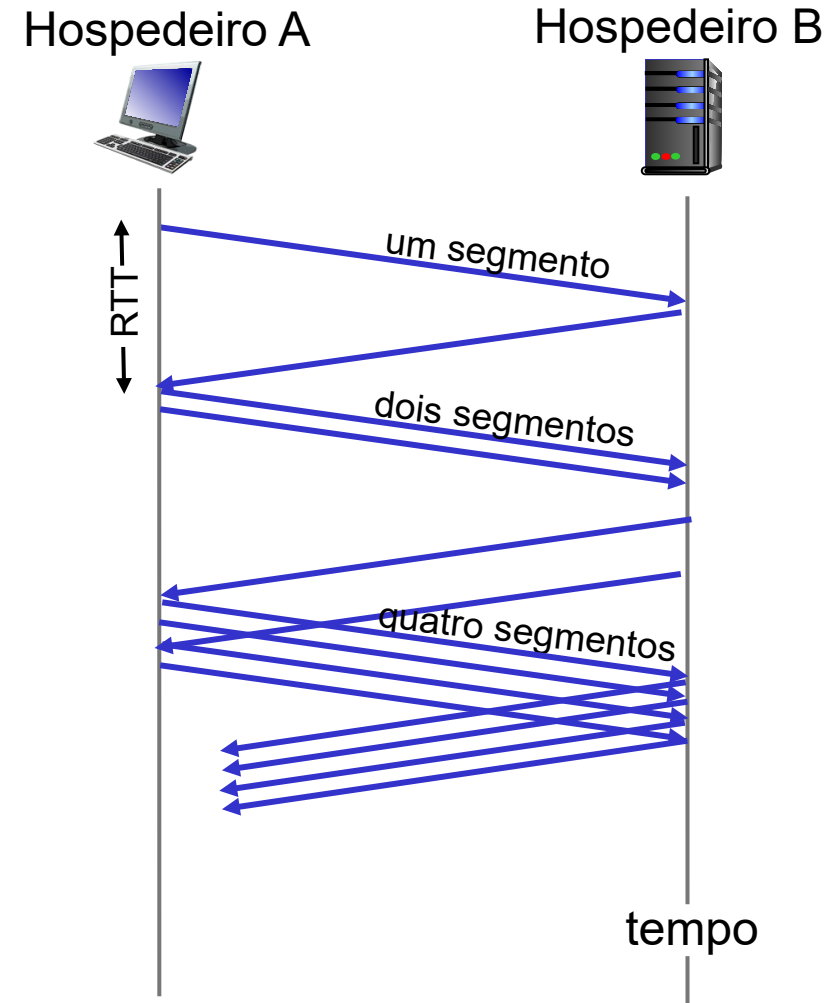
- *aproximadamente*: envia `cwnd` bytes, espera RTT por ACKs, então envia mais bytes

$$\text{taxa TCP} \approx \frac{\text{cwnd}}{\text{RTT}} \text{ bytes/seg}$$

- emissor TCP limita transmissão: $\text{LastByteSent} - \text{LastByteAcked} \leq \text{cwnd}$
- `cwnd` é ajustado dinamicamente em resposta ao congestionamento observado na rede (implementando o controle de congestionamento do TCP)

Partida lenta do TCP

- quando a conexão começa, aumenta a taxa exponencialmente até o primeiro evento de perda:
 - inicialmente **cwnd** = 1 MSS
 - dobra **cwnd** a cada RTT
 - feito incrementando **cwnd** para cada ACK recebido
- *resumo*: taxa inicial é lenta, mas aumenta exponencialmente rápido



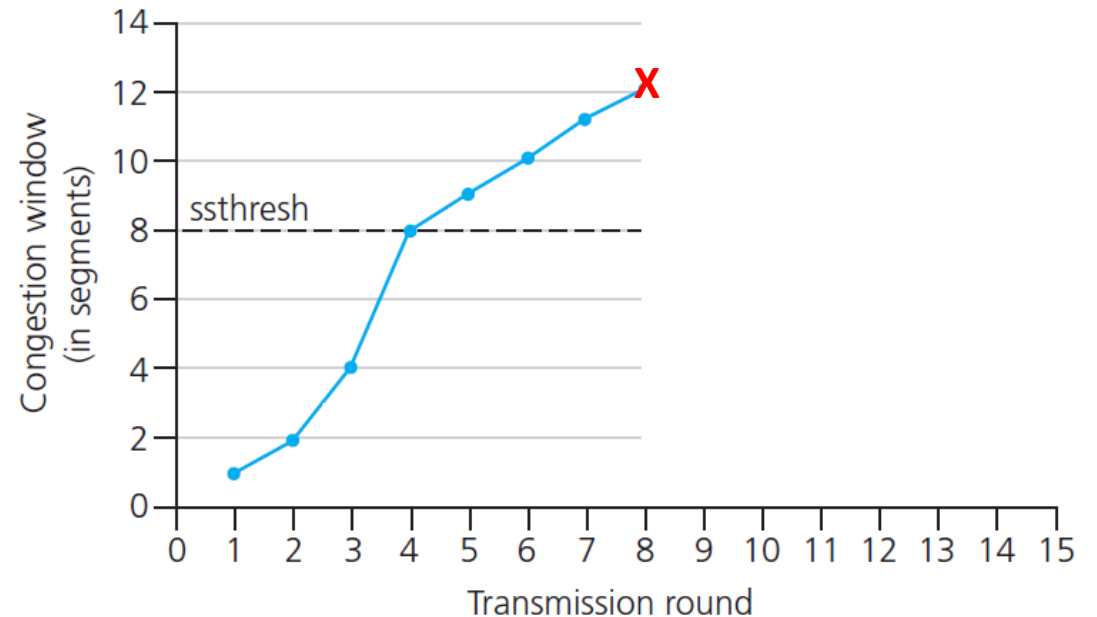
TCP: da partida lenta para a prevenção de congestionamento

Q: quando o aumento exponencial deve mudar para linear?

R: quando **cwnd** chega a 1/2 de seu valor antes do timeout.

Implementação:

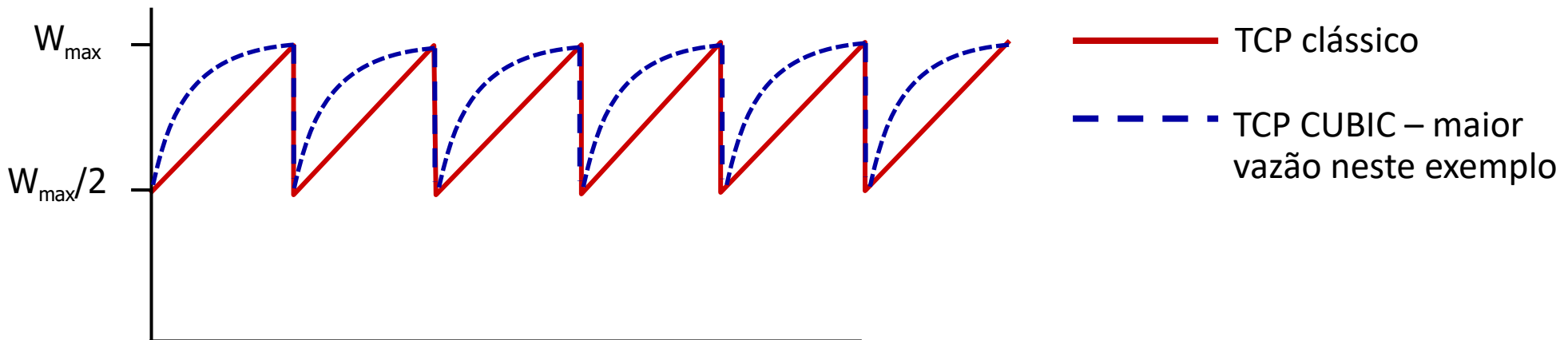
- variável **ssthresh**
- em caso de perda, **ssthresh** é ajustado para 1/2 do **cwnd** um pouco antes do evento de perda



* Confira os exercícios interativos online para mais exemplos: http://gaia.cs.umass.edu/kurose_ross/interactive/

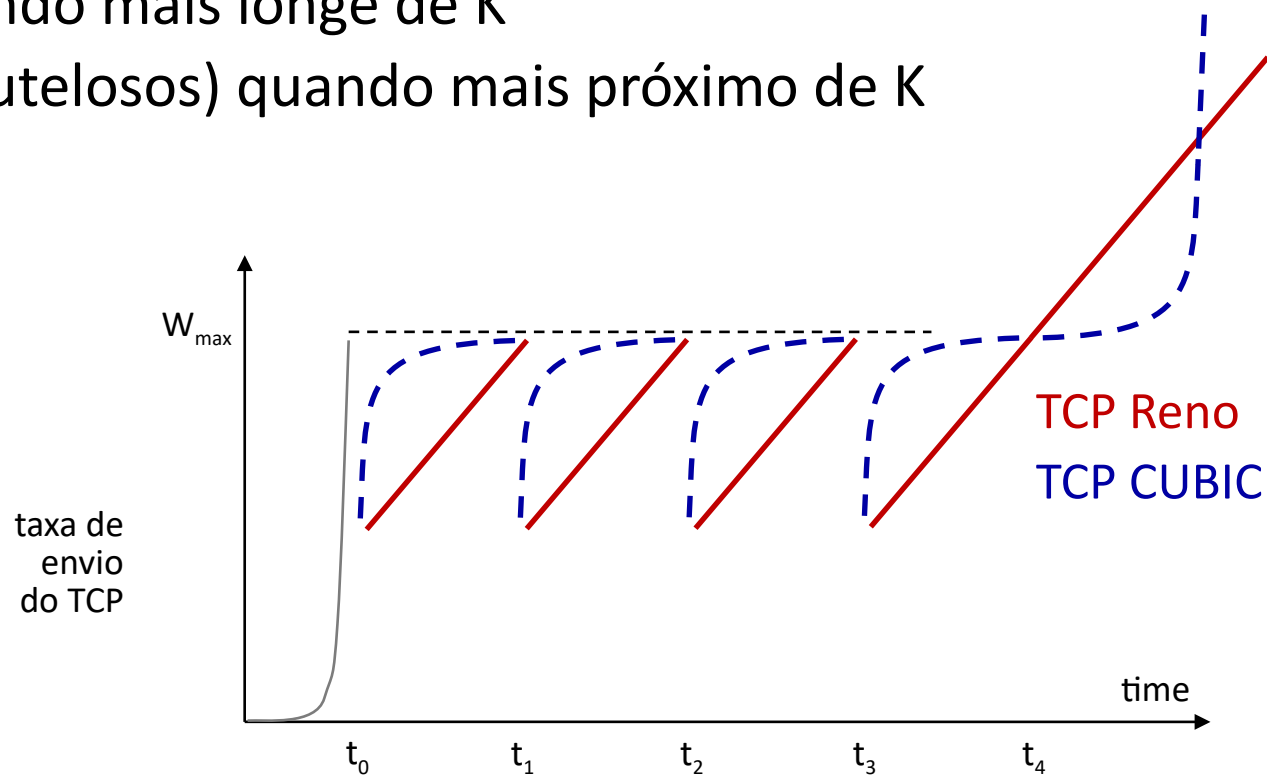
TCP CUBIC

- Existe uma maneira melhor do que o AIMD de “sondar” a largura de banda utilizável?
- Insight/intuição:
 - W_{\max} : taxa de envio em que a perda por congestionamento foi detectada
 - estado de congestionamento do enlace de gargalo provavelmente (?) não mudou muito
 - depois de cortar taxa/janela ao meio na perda, inicialmente sobe em direção a W_{\max} *mais rápido*, mas em seguida se aproxima de W_{\max} mais *devagar*



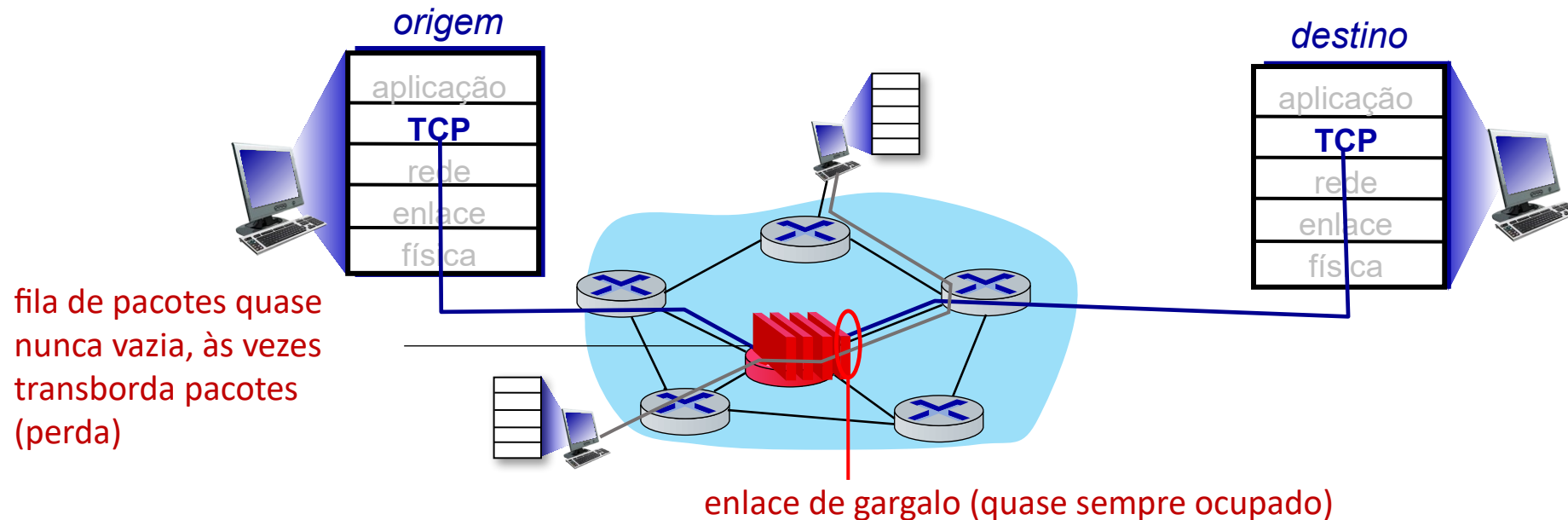
TCP CUBIC

- K: ponto no tempo em que o tamanho da janela do TCP atingirá W_{\max}
 - K em si é ajustável
- aumenta W em função do *cu*bo da distância entre o tempo atual e K
 - aumentos maiores quando mais longe de K
 - aumentos menores (cautelosos) quando mais próximo de K
- TCP CUBIC é padrão no Linux, e o TCP mais popular em servidores Web populares



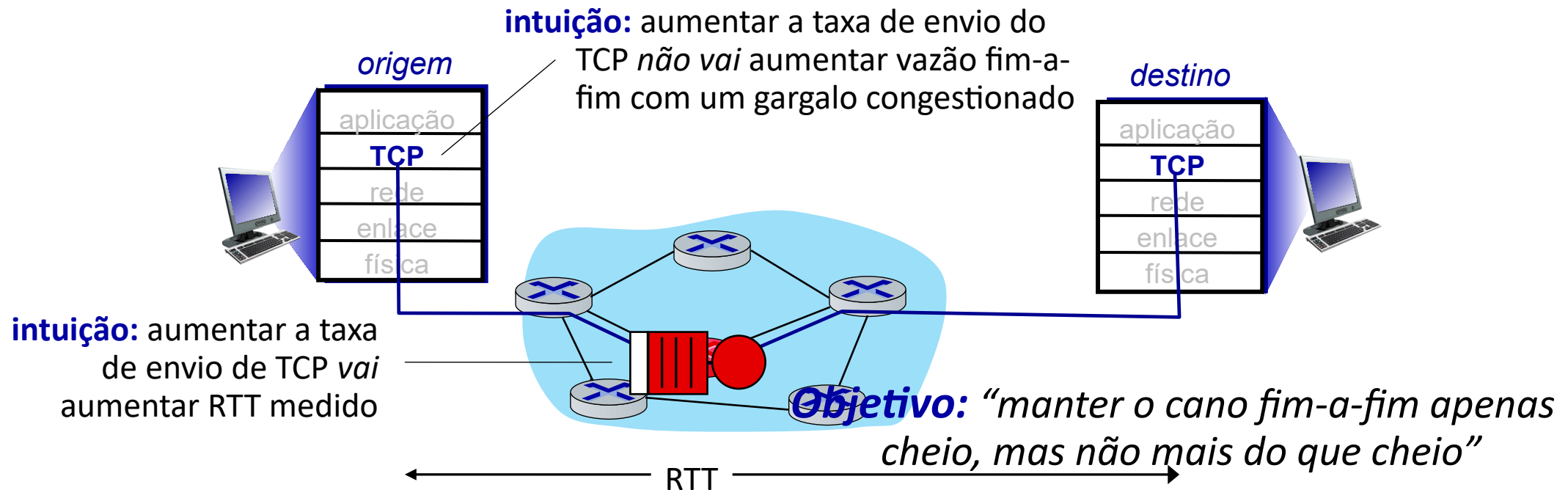
TCP e o “enlace de gargalo” congestionado

- TCP (clássico, CUBIC) aumenta a taxa de envio do TCP até que uma perda de pacote ocorra na saída de algum roteador: o *enlace de gargalo*



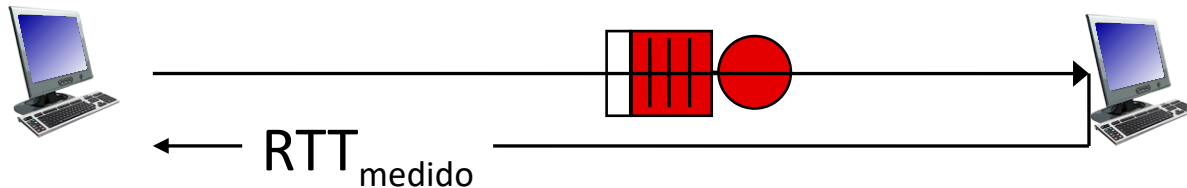
TCP e o “enlace de gargalo” congestionado

- TCP (clássico, CUBIC) aumenta a taxa de envio do TCP até que uma perda de pacote ocorra na saída de algum roteador: o *enlace de gargalo*
- entendendo o congestionamento: é útil focar no enlace de gargalo congestionado



Controle de congestionamento do TCP baseado em atraso

Mantendo o tubo emissor-para-receptor “apenas cheio o suficiente, mas não mais que isso”: manter o enlace de gargalo ocupado transmitindo, mas evitar altos atrasos e uso de buffer



$$\text{vazão medida} = \frac{\text{\# bytes enviados no último intervalo de RTT}}{RTT_{medido}}$$

Abordagem baseada em atraso:

- RTT_{min} - RTT mínimo observado (caminho não congestionado)
- vazão não congestionada quando janela de congestionamento $cwnd$ é $cwnd/RTT_{min}$

se vazão medida “muito perto” da vazão sem congestionamento

 aumentar $cwnd$ linearmente /* pois o caminho não está congestionado */

senão se vazão medida “muito abaixo” da vazão sem congestionamento

 diminuir $cwnd$ linearmente /* pois o caminho está congestionado */

Controle de congestionamento do TCP baseado em atraso

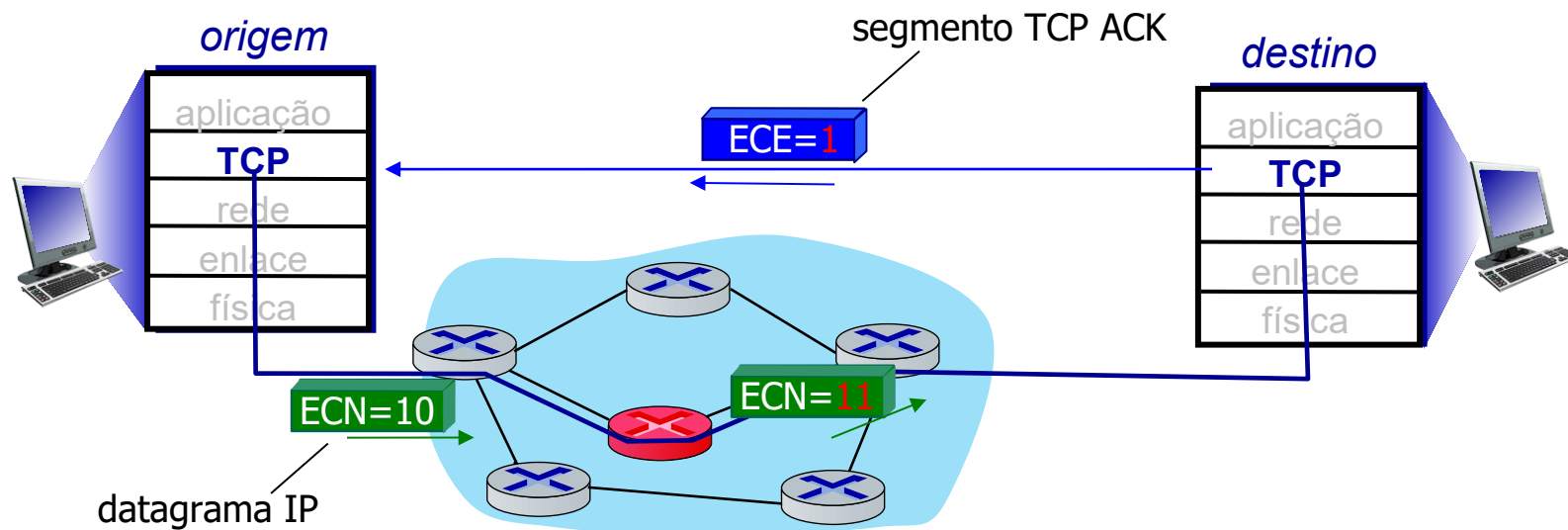
controle de congestionamento sem induzir/forçar perda

- maximizando vazão (“mantendo o tubo apenas cheio...”) enquanto mantém o atraso baixo (“...mas não mais do que cheio”)
- uma série de TCPs implantados usam uma abordagem baseada em atraso
 - BBR implantado na rede de backbone (interna) do Google

Notificação Explícita de Congestionamento- Explicit congestion notification (ECN)

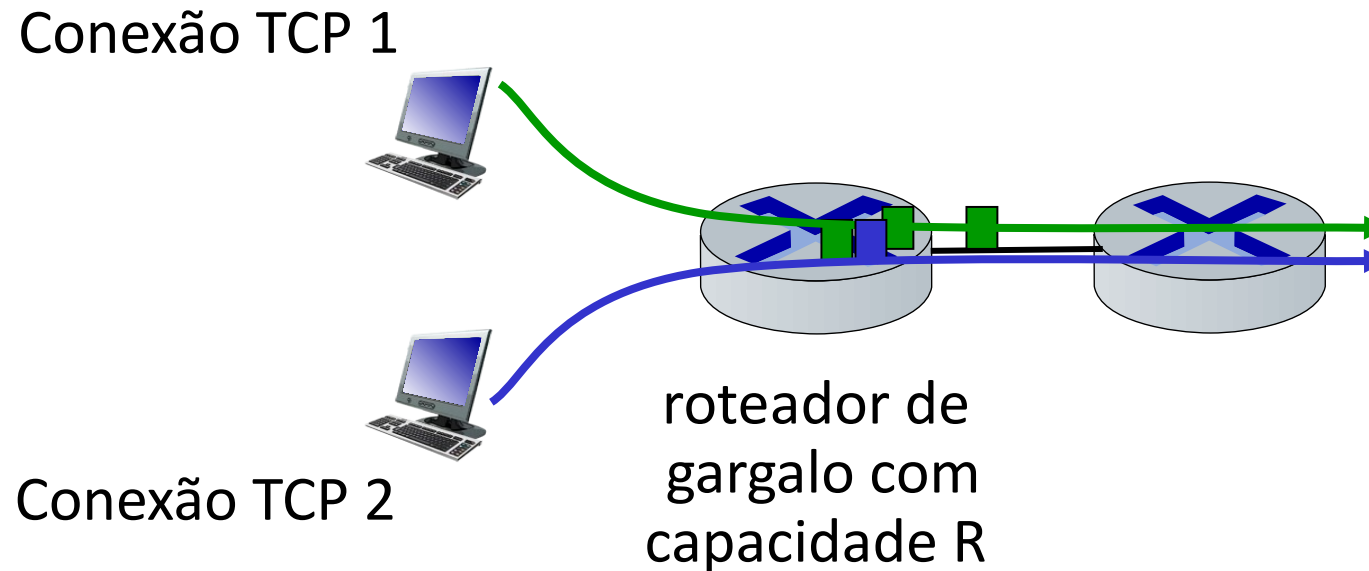
Implantações TCP muitas vezes implementam controle de congestionamento *assistido por rede*:

- dois bits no cabeçalho IP (campo ToS) marcado *por roteador de rede* para indicar congestionamento
 - *política* para determinar a marcação escolhida pelo operador de rede
- indicação de congestionamento levada para o destino
- destino define o bit ECE no segmento ACK para notificar o remetente do congestionamento
- envolve tanto IP (marcação de bit ECN do cabeçalho IP) e TCP (marcação de bits C,E no cabeçalho do TCP)



Equidade do TCP

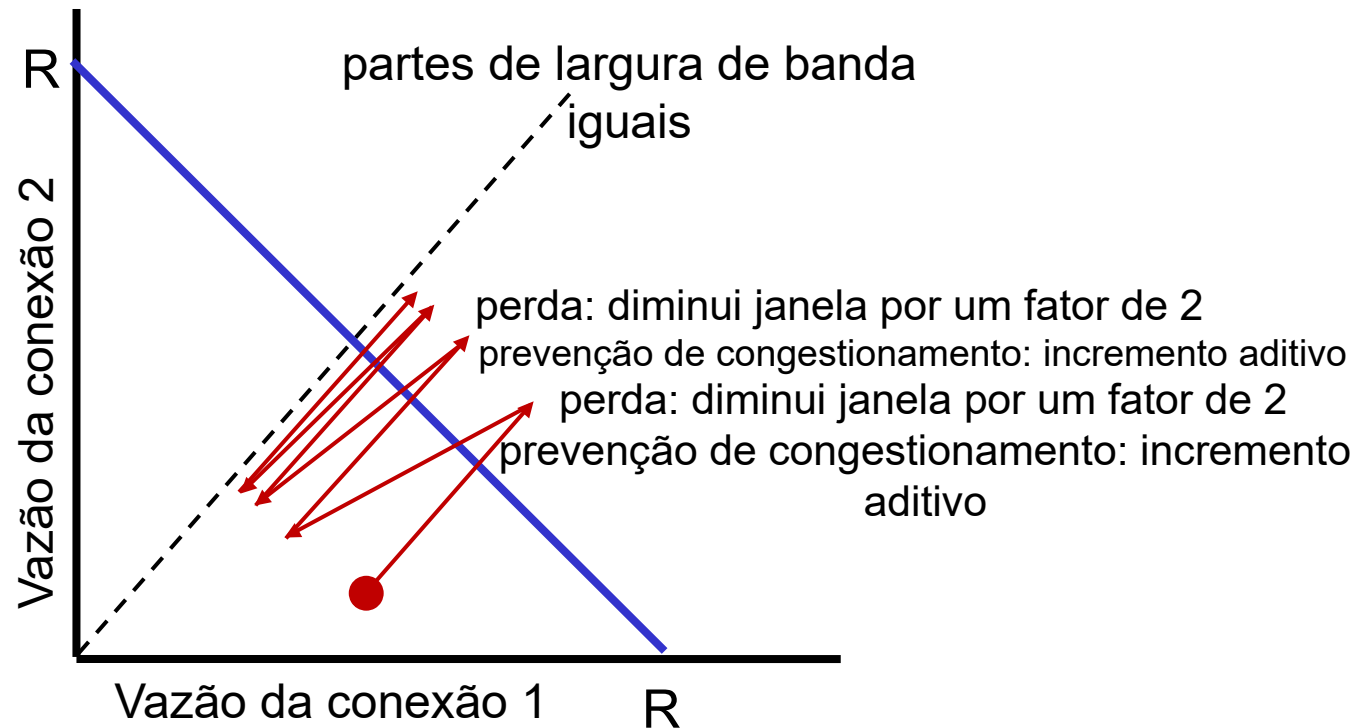
Objetivo de equidade: se K sessões TCP compartilham o mesmo enlace de gargalo de largura de banda R , cada um deve ter taxa média de R/K



Q: o TCP é justo?

Exemplo: duas sessões TCP competindo:

- aumento aditivo dá inclinação de 1, conforme a vazão aumenta
- decremento multiplicativo diminui vazão proporcionalmente



O TCP é justo?

R: Sim, sob suposições idealizadas:

- mesmo RTT
- número fixo de sessões apenas para evitar congestionamento

Equidade: todas as aplicações de rede devem ser “justas”?

Equidade e UDP

- aplicações multimídia muitas vezes não usam TCP
 - não querem taxa estrangulada pelo controle de congestionamento
- em vez disso usam UDP:
 - enviam áudio/vídeo em taxa constante, tolerando perda de pacotes
- não há uma “polícia da Internet” policiando o uso do controle de congestionamento

Equidade, conexões TCP paralelas

- aplicação pode abrir *múltiplas* conexões paralelas entre dois hospedeiros
- navegadores fazem isso, ex.: enlace de taxa R com 9 conexões existentes:
 - nova aplicação pede 1 conexão TCP, recebe taxa $R/10$
 - nova aplicação pede 11 conexões TCP, recebe $R/2$

Camada de transporte: roteiro

- Serviços da camada de transporte
- Multiplexação e demultiplexação
- Transporte sem conexão: UDP
- Princípios da transferência confiável de dados
- Transporte orientado a conexão: TCP
- Princípios de controle de congestionamentos
- Controle de congestionamento do TCP
- **Evolução da funcionalidade da camada de transporte**



Funcionalidade em evolução da camada de transporte

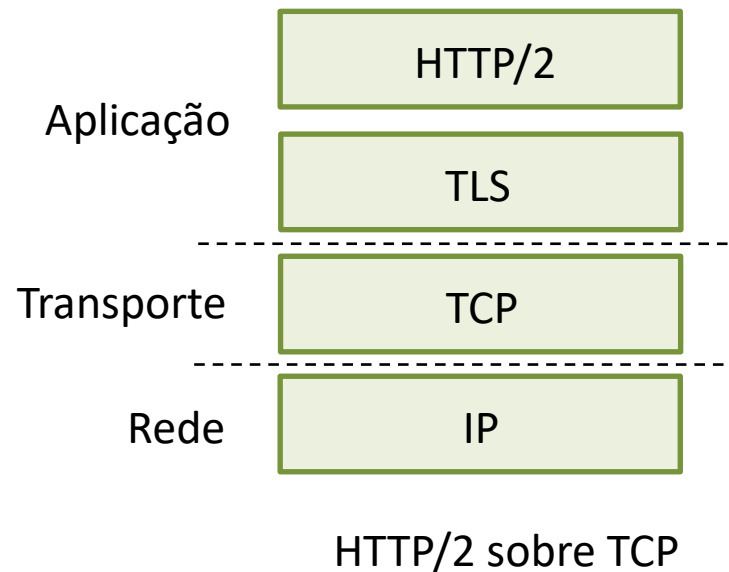
- TCP, UDP: principais protocolos de transporte por 40 anos
- diferentes “sabores” do TCP desenvolvidos para cenários específicos:

Cenário	Desafios
Pipes longos e gordos (grandes transferências de dados)	Muitos pacotes “em voo”; perda fecha o pipeline
Redes sem fio	Perdas devido a enlaces sem fio ruidosos, mobilidade; TCP trata isto como perda por congestionamento
Enlaces de longo atraso	RTTs extremamente longos
Redes de data center	Sensibilidade à latência
Fluxos de tráfego de fundo	Fluxos TCP de baixa prioridade, “de fundo”

- movendo funções da camada de transporte para a camada de aplicação, em cima do UDP
 - HTTP/3: QUIC

QUIC: Quick UDP Internet Connections – Conexões Rápidas de Internet UDP

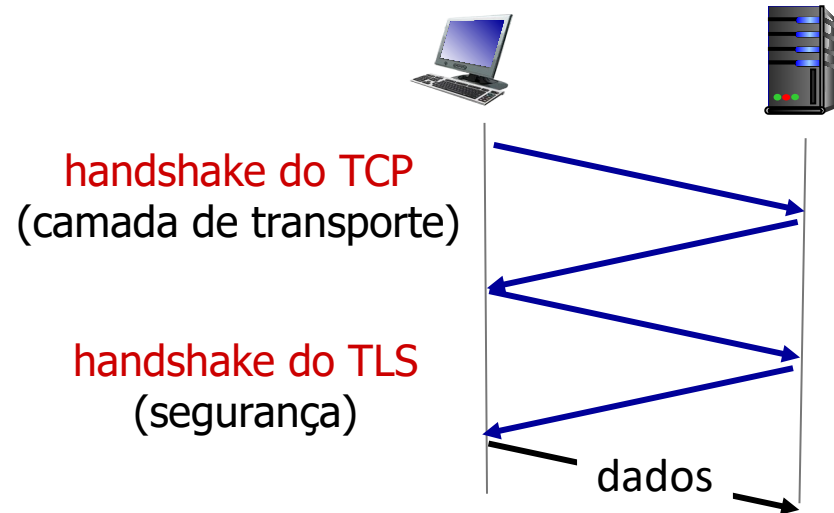
- protocolo da camada de aplicação, em cima do UDP
 - aumenta o desempenho do HTTP
 - implantado em muitos servidores do Google e aplicativos (Chrome, aplicativo móvel do YouTube)



QUIC: Quick UDP Internet Connections

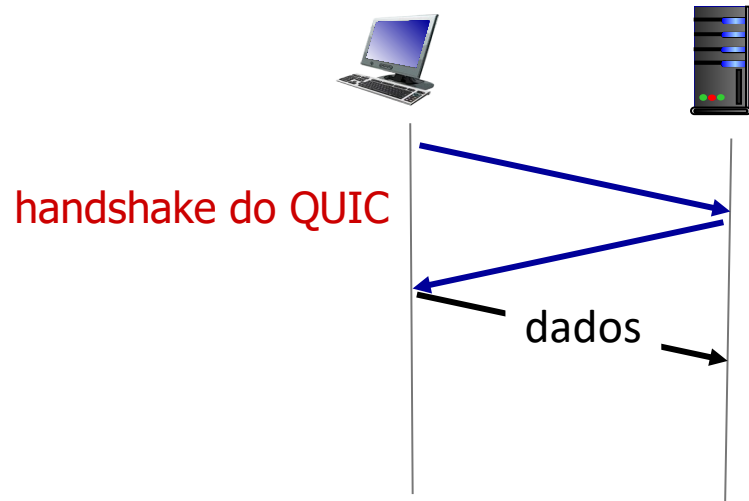
- adota abordagens que estudamos neste capítulo para estabelecimento de conexão, controle de erros, controle de congestionamento
 - **controle de erros e congestionamento:** “Leitores familiarizados com a detecção de perda e controle de congestionamento do TCP encontrarão algoritmos aqui paralelos aos bem conhecidos do TCP.” [da especificação do QUIC]
 - **estabelecimento de conexão:** confiabilidade, controle de congestionamento, autenticação, criptografia e estado estabelecidos em um RTT
- múltiplos “fluxos” em nível de aplicação multiplexados sobre uma única conexão QUIC
 - transferência de dados confiável e segurança separados para cada fluxo
 - controle de congestionamento comum para todos os fluxos

QUIC: Estabelecimento de conexão



TCP (confiabilidade, controle de congestionamento) + TLS (autenticação, criptografia)

- 2 *handshakes* em série



QUIC: confiabilidade, controle de congestionamento, autenticação, criptografia

- 1 *handshake*