

# Capítulo 3

## Camada de transporte

Uma observação sobre o uso desses slides do PowerPoint:

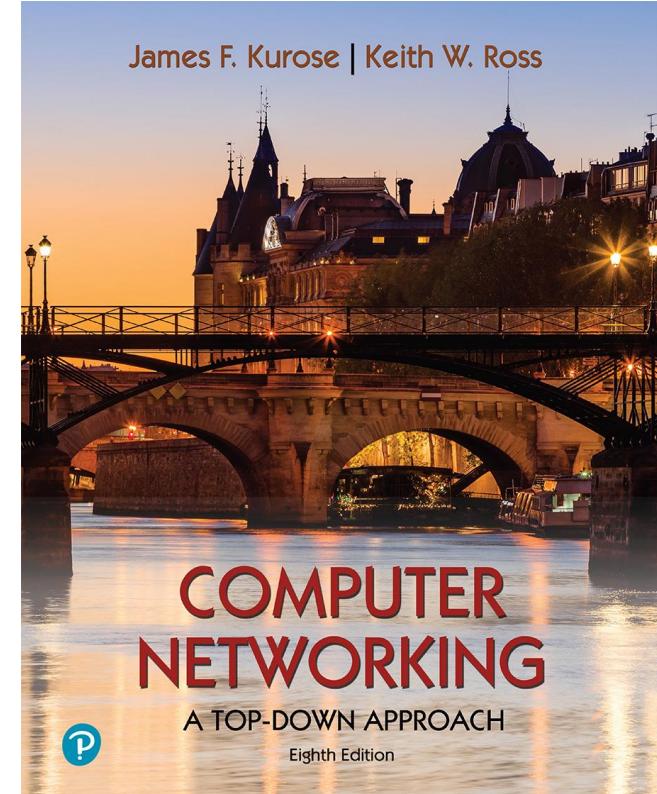
Estamos disponibilizando esses slides gratuitamente para todos (professores, alunos, leitores). Eles estão no formato PowerPoint para que você veja as animações e possa adicionar, modificar e excluir slides (inclusive este) e o conteúdo dos slides para atender às suas necessidades. Obviamente, eles representam *muito* trabalho de nossa parte. Em troca do uso, pedimos apenas o seguinte:

- Se você usar esses slides (por exemplo, em uma aula), mencione a fonte (afinal, gostaríamos que as pessoas usassem nosso livro!)
- Se você publicar algum slide em um site www, informe que ele foi adaptado de nossos slides (ou talvez idêntico a eles) e informe nossos direitos autorais sobre esse material.

Para obter um histórico de revisões, consulte a nota do slide desta página.

Obrigado e divirta-se! JFK/KWR

Todos os materiais têm direitos autorais de 1996 a 2023  
J.F. Kurose e K.W. Ross, Todos os direitos reservados



*Redes de computadores: A Top-Down Approach (Uma abordagem de cima para baixo)*

8<sup>th</sup> edition  
Jim Kurose, Keith Ross  
Pearson, 2020

# Camada de transporte: visão geral

*Nosso objetivo:*

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

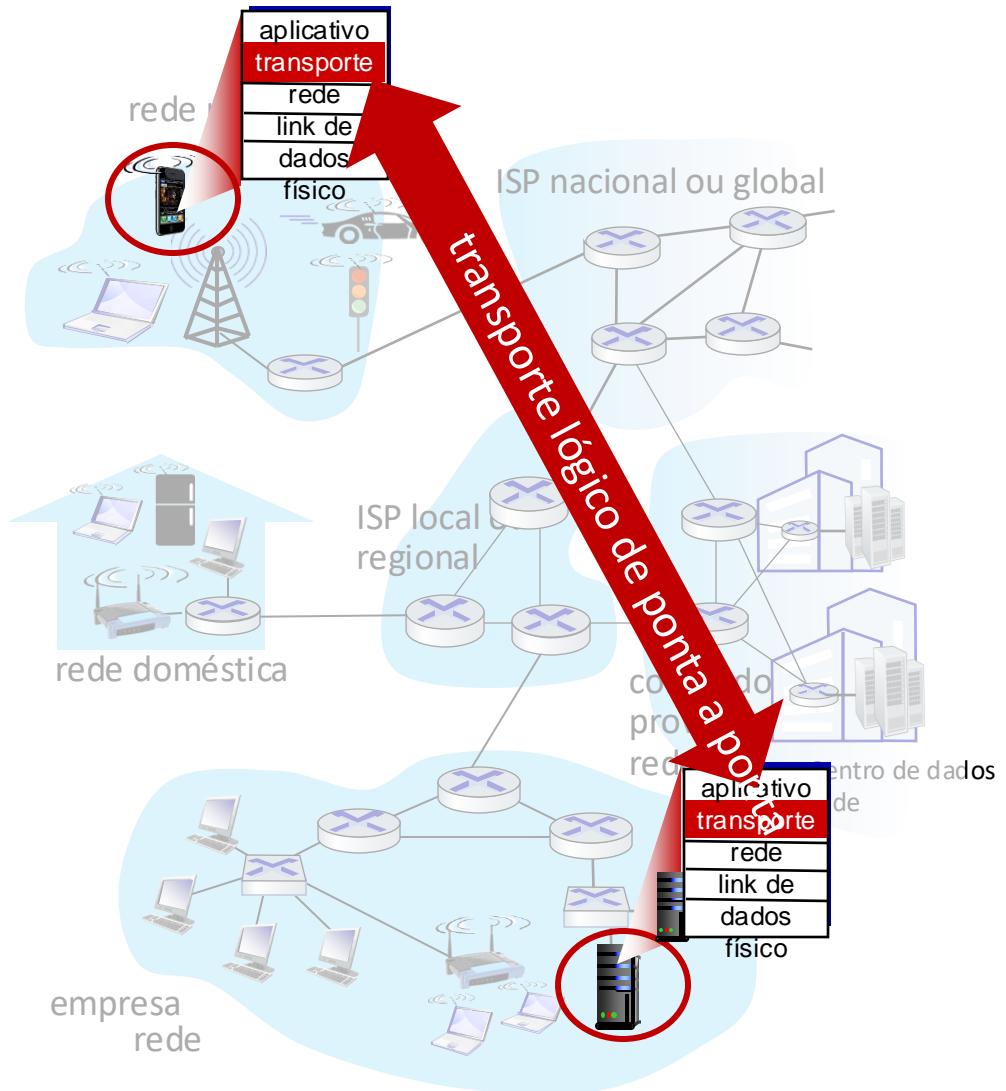
# Camada de transporte: roteiro

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



# Serviços e protocolos de transporte

- fornecer *comunicação lógica* entre processos de aplicativos executados em hosts diferentes
- ações de protocolos de transporte em sistemas finais:
  - remetente: divide as mensagens do aplicativo em *segmentos* e passa para a camada de rede
  - receptor: reagrupa os segmentos em mensagens e passa para a camada de aplicativos
- dois protocolos de transporte disponíveis para aplicativos da Internet
  - TCP, UDP



# Serviços e protocolos de transporte versus camada de rede



*analogia doméstica:*

*12 crianças na casa de Ann  
enviando cartas para 12  
crianças na casa de Bill:*

- hosts = casas
- processos = crianças
- mensagens de aplicativos = cartas em envelopes

**serviço postal**

# Serviços e protocolos de transporte versus camada de rede

- **camada de transporte:**  
comunicação entre processos
  - depende de, aprimora,  
serviços de camada de rede
  
- **camada de rede:**  
comunicação entre *hosts*

*analogia doméstica:*

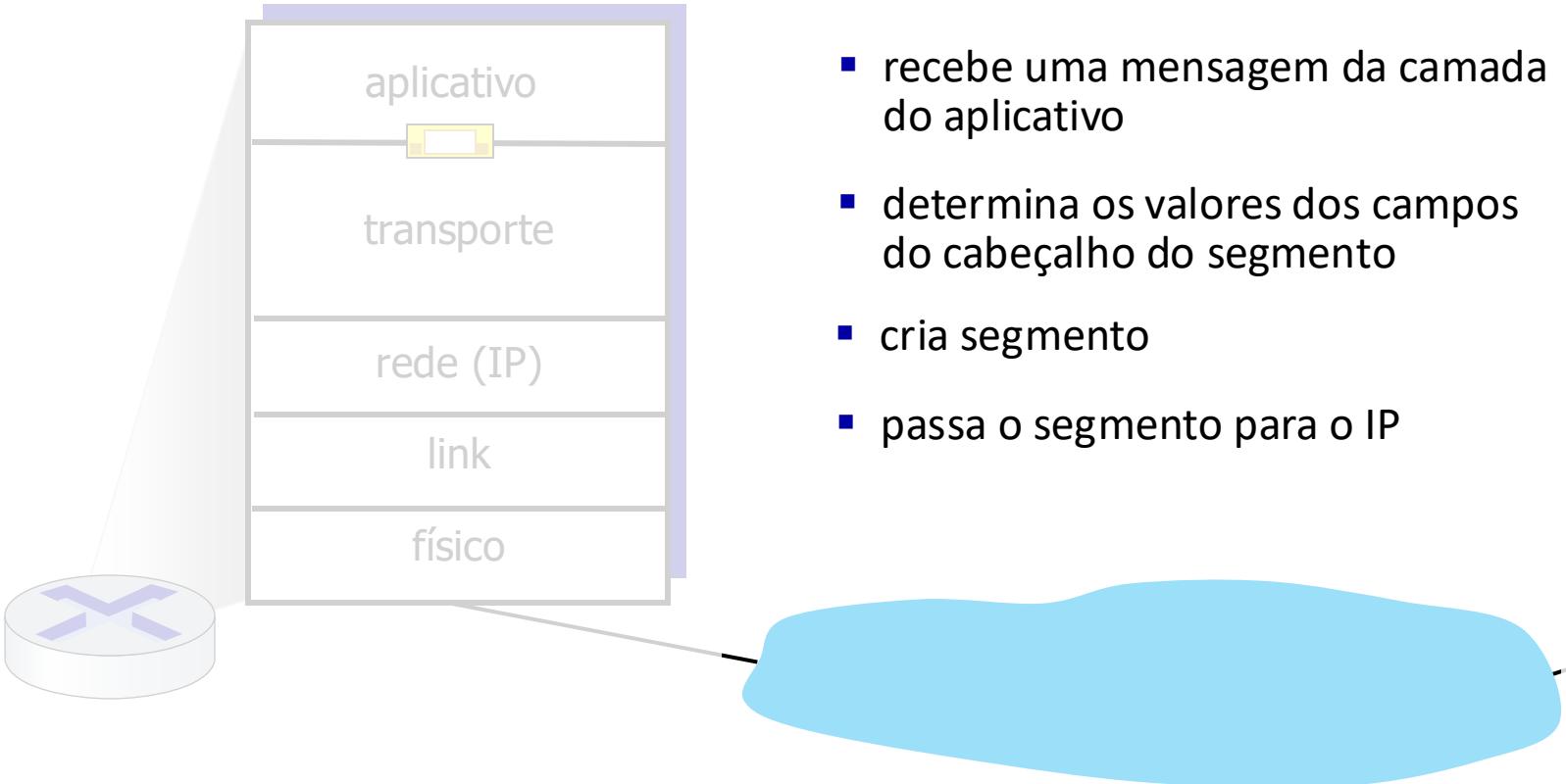
- 12 crianças na casa de Ann  
enviando cartas para 12  
crianças na casa de Bill:*
- hosts = casas
  - processos = crianças
  - mensagens de aplicativos =  
cartas em envelopes

*serviço postal*

# Ações da camada de transporte

Remetente:

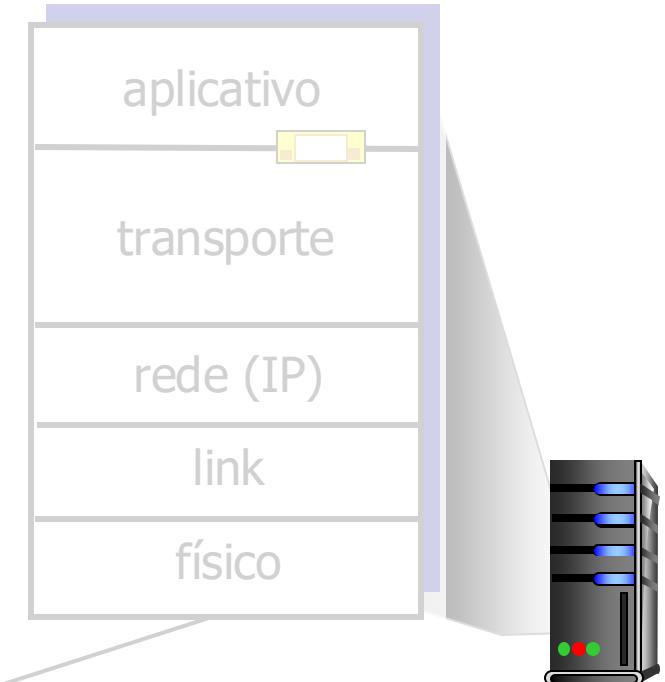
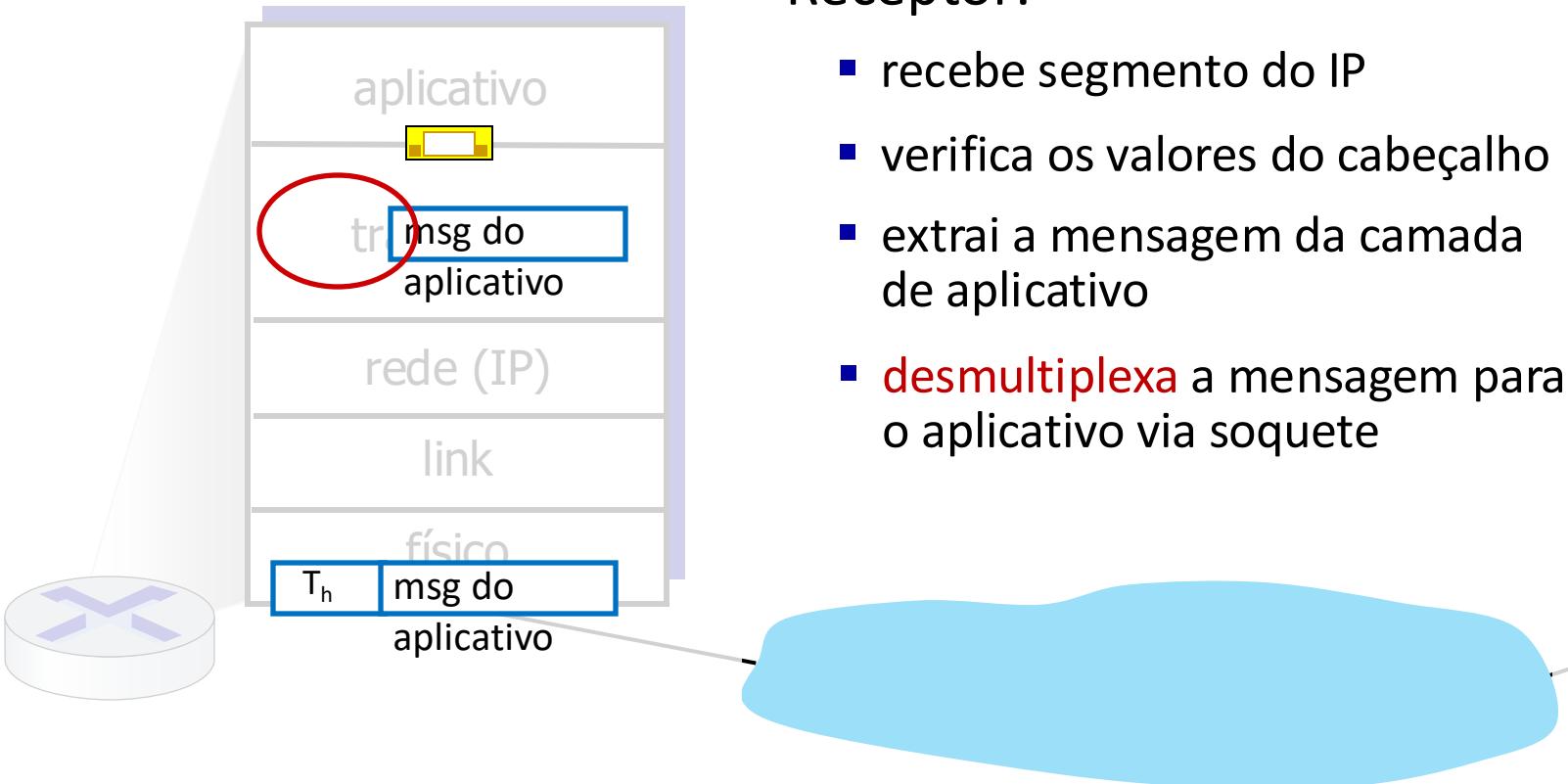
- recebe uma mensagem da camada do aplicativo
- determina os valores dos campos do cabeçalho do segmento
- cria segmento
- passa o segmento para o IP



# Ações da camada de transporte

## Receptor:

- recebe segmento do IP
- verifica os valores do cabeçalho
- extrai a mensagem da camada de aplicativo
- **desmultiplexa** a mensagem para o aplicativo via soquete



# Dois principais protocolos de transporte da Internet

## ■ **TCP:** Protocolo de Controle de Transmissão

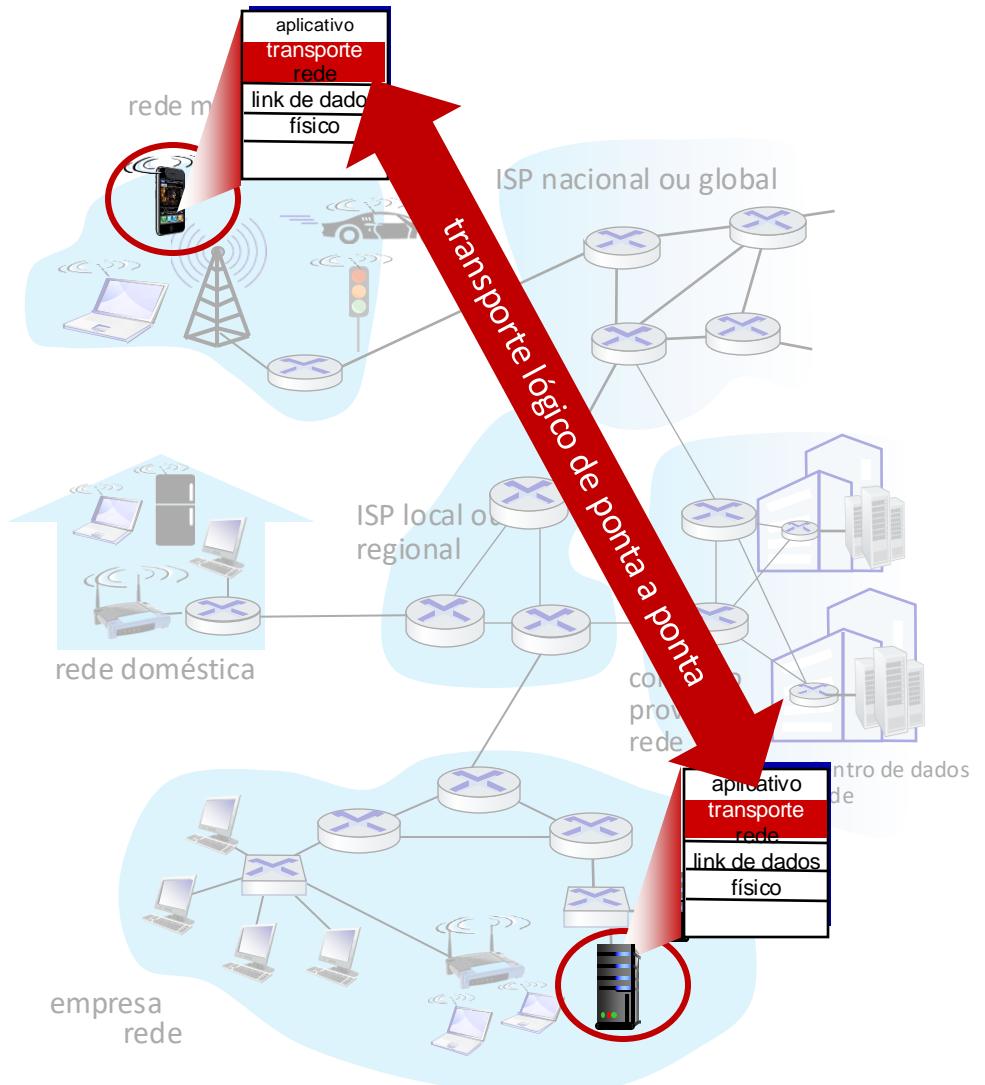
- entrega confiável e dentro do pedido
- controle de congestionamento
- controle de fluxo
- Configuração da conexão

## ■ **UDP:** Protocolo de datagrama do usuário

- entrega não confiável e não ordenada
- extensão simples do IP de "melhor esforço"

## ■ serviços *não* disponíveis:

- garantias de atraso
- garantias de largura de banda



# Capítulo 3: roteiro

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



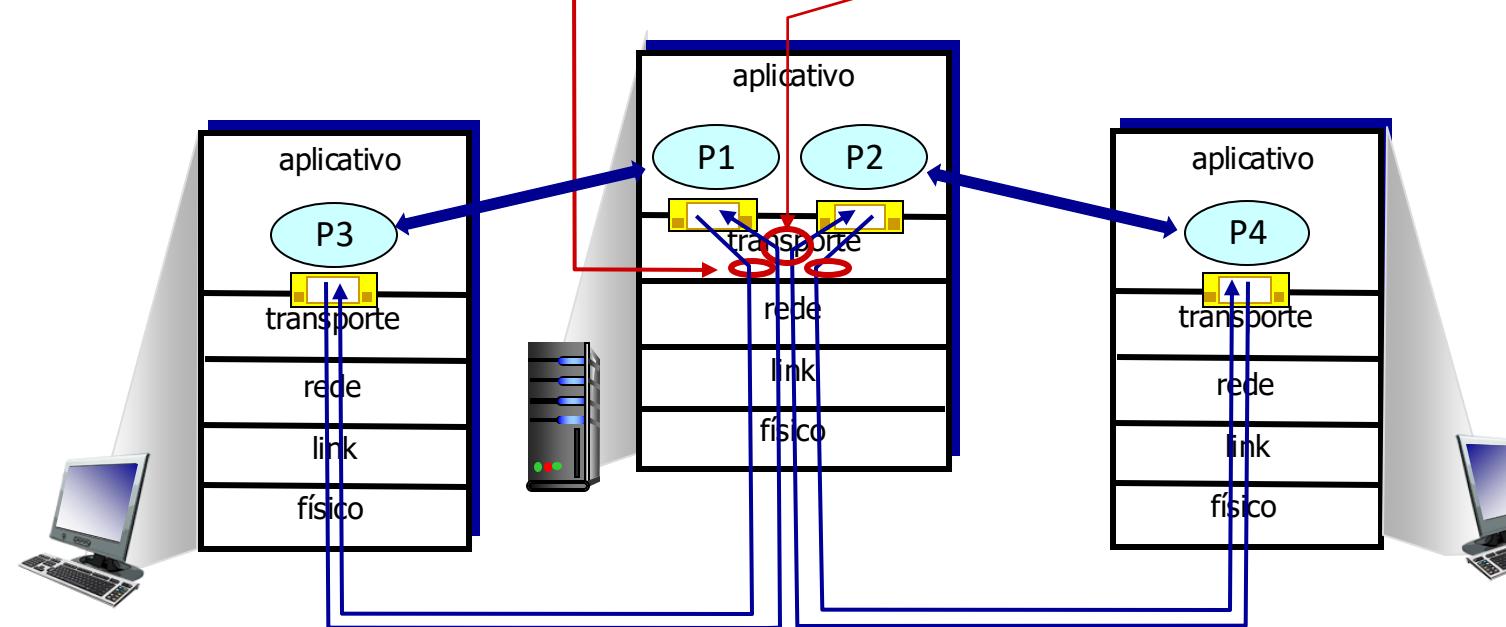
# Multiplexação/demultiplexação

*multiplexação como remetente:*

manipular dados de vários soquetes, adicionar cabeçalho de transporte (usado posteriormente para demultiplexação)

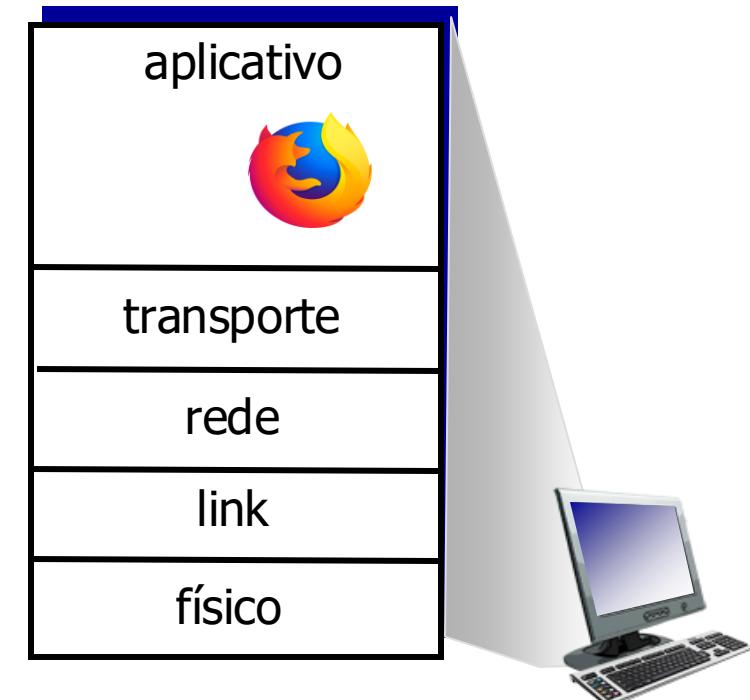
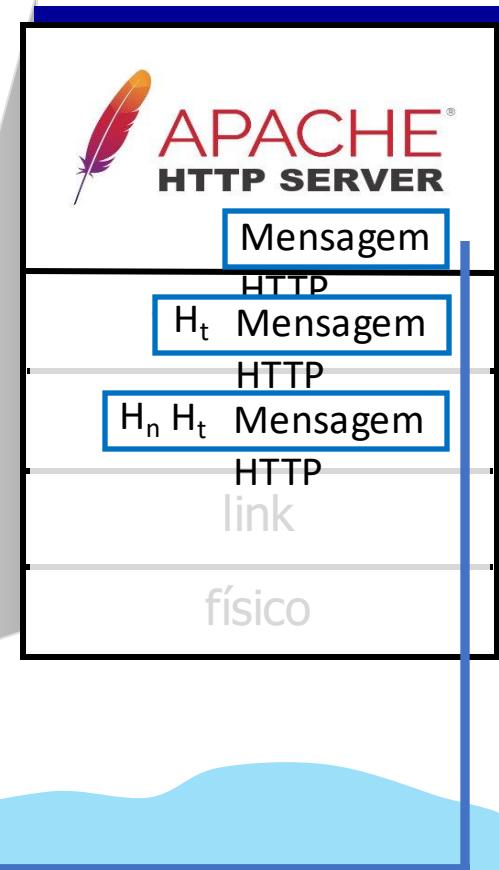
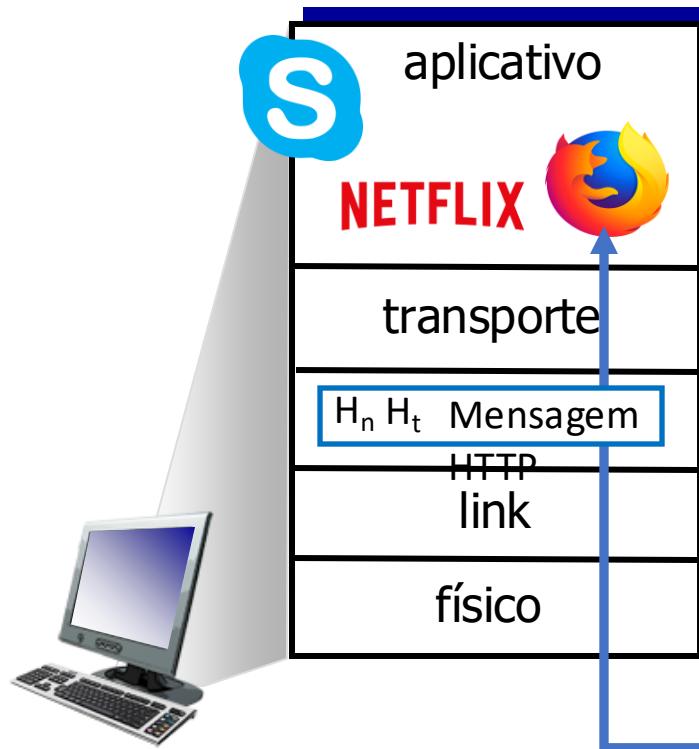
*demultiplexação como receptor:*

usar informações de cabeçalho para entregar segmentos recebidos para corrigir soquete



## Servidor HTTP

cliente

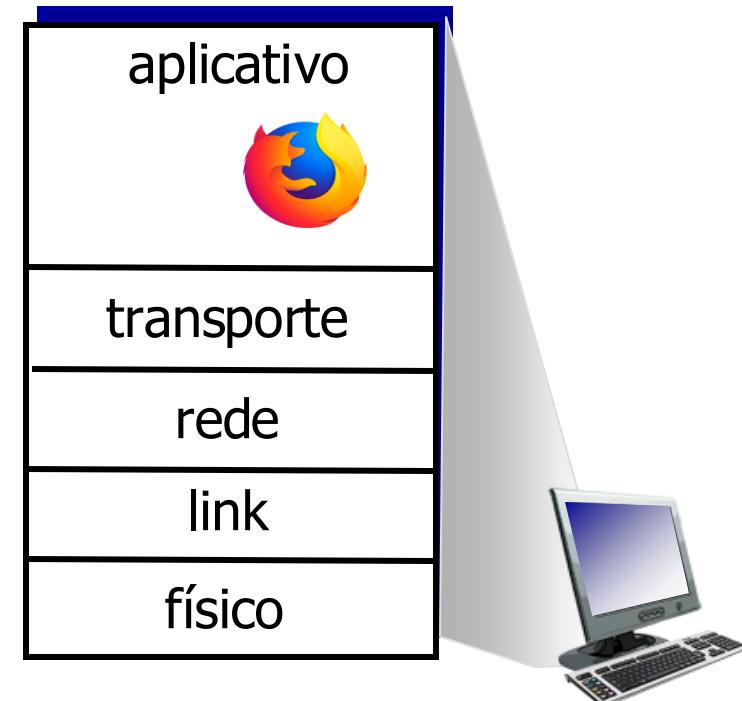
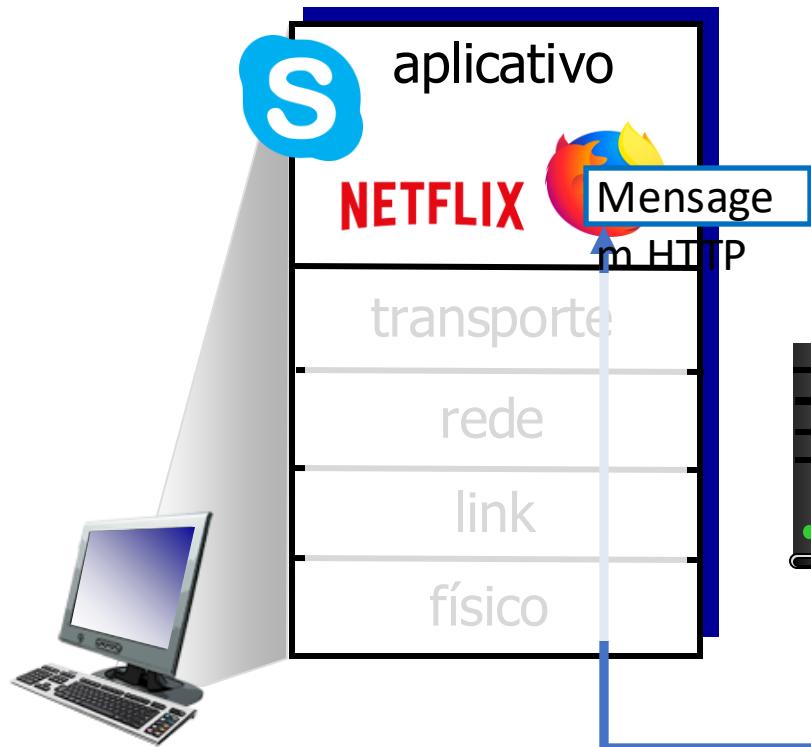


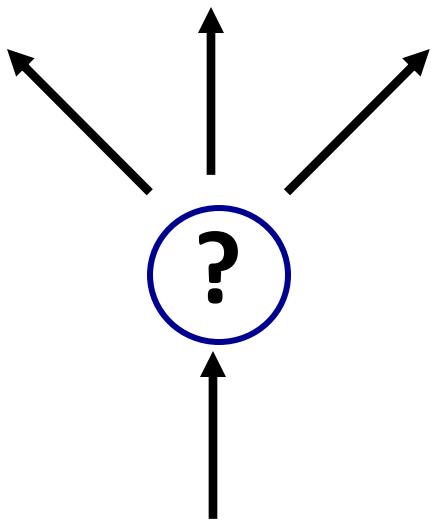
$H_n H_t$  Mensagem  
HTTP



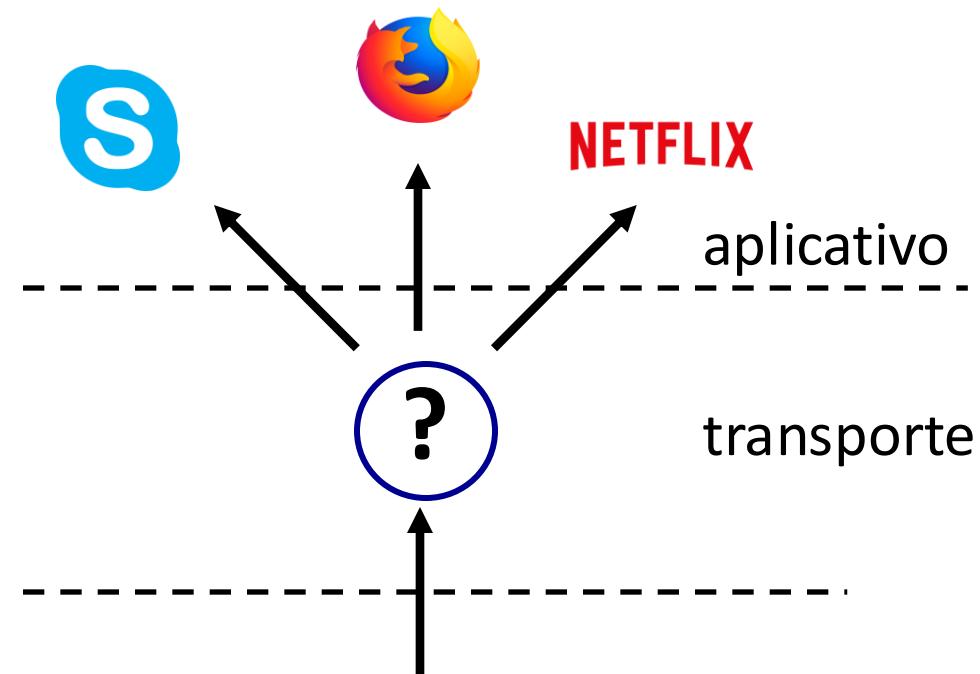
*P: Como a camada de transporte sabia que deveria entregar a mensagem ao processo do navegador Firefox em vez de ao processo da Netflix ou do Skype?*

cliente





de-multiplexação



de-multiplexação



# Demultiplexação

AIRFRANCE /

ECONOMY /



SKYTEAM

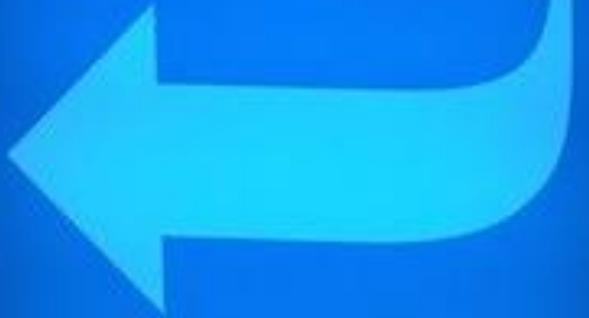
AIRFRANCE /

SKY  
PRIORITY™



SKYTEAM

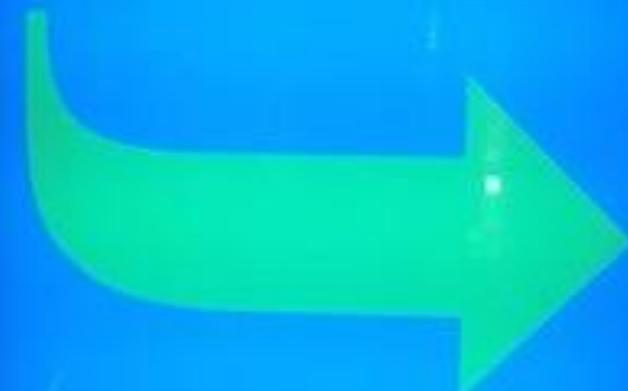
TSA Pre✓



Transportation  
Security  
Administration

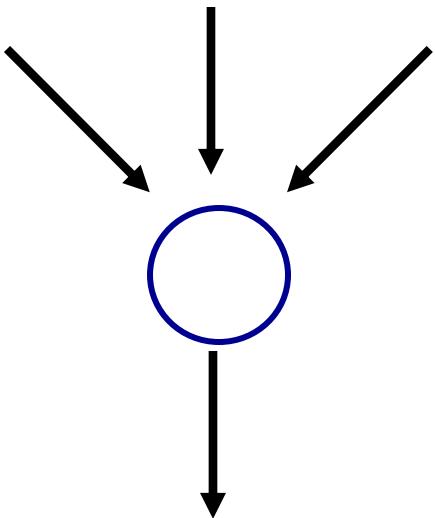
tsa.gov

Main  
Checkpoint

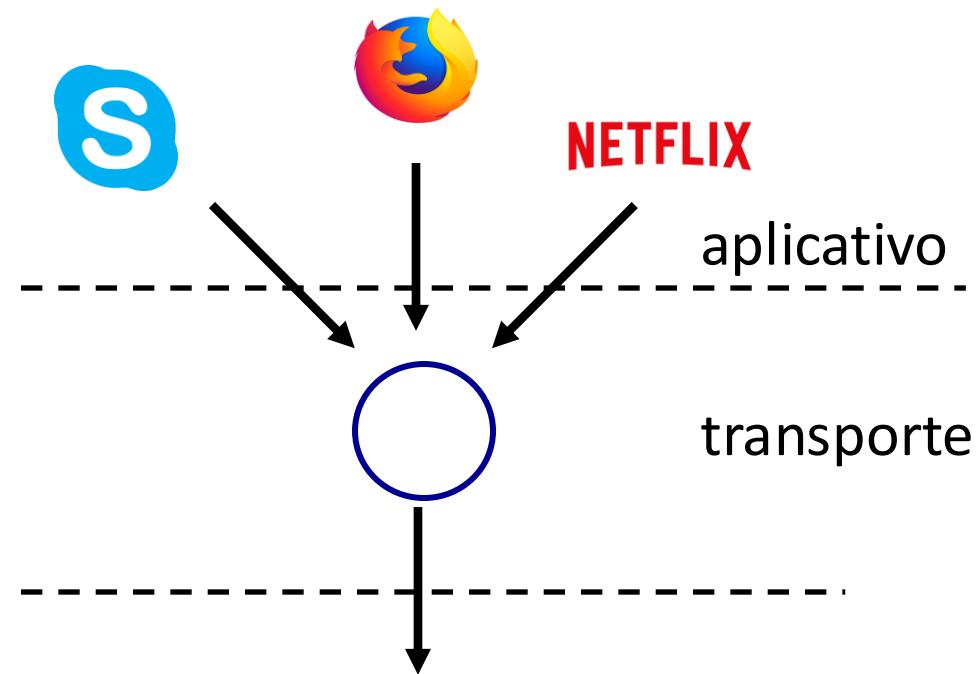


Transportation  
Security  
Administration

tsa.gov



multiplexação



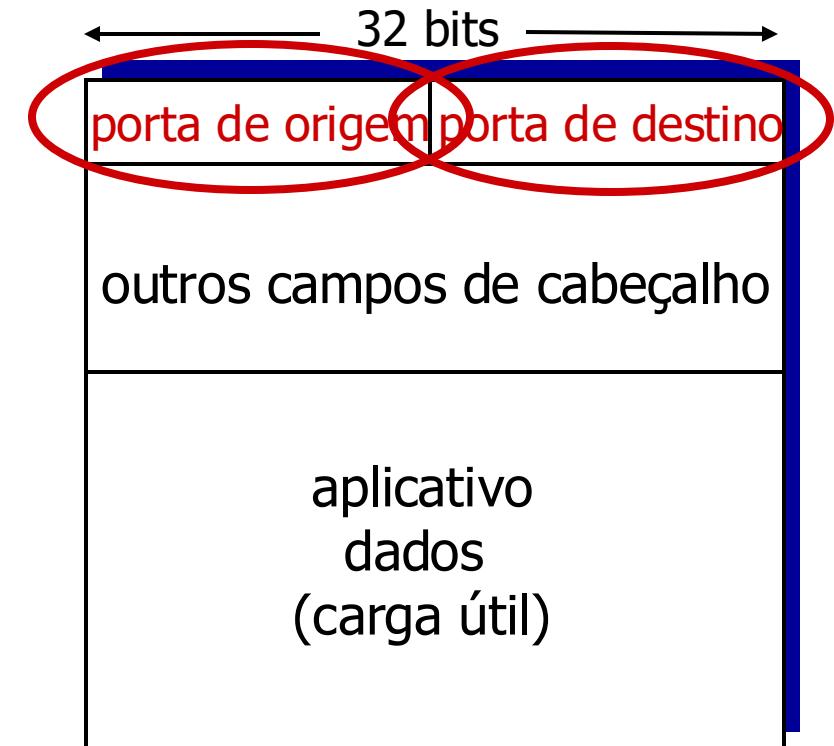
multiplexação



# Multiplexação

# Como funciona a demultiplexação

- o host recebe datagramas IP
  - Cada datagrama tem endereço IP de origem, endereço IP de destino
  - cada datagrama carrega um segmento da camada de transporte
  - cada segmento tem um número de porta de origem e de destino
- O host usa *endereços IP e números de porta* para direcionar o segmento para o soquete apropriado



Formato do segmento TCP/UDP

# Demultiplexação sem conexão

*Recall:*

- ao criar o soquete, deve especificar o número da porta *local do host*:  


```
DatagramSocket mySocket1 =  
    novo DatagramSocket(1234);
```
- ao criar um datagrama para enviar para um soquete UDP, deve especificar
  - endereço IP de destino
  - porta de destino #

quando o host receptor recebe o segmento *UDP*:

- verifica a porta de destino nº no segmento
- direciona o segmento UDP para o soquete com essa porta nº.



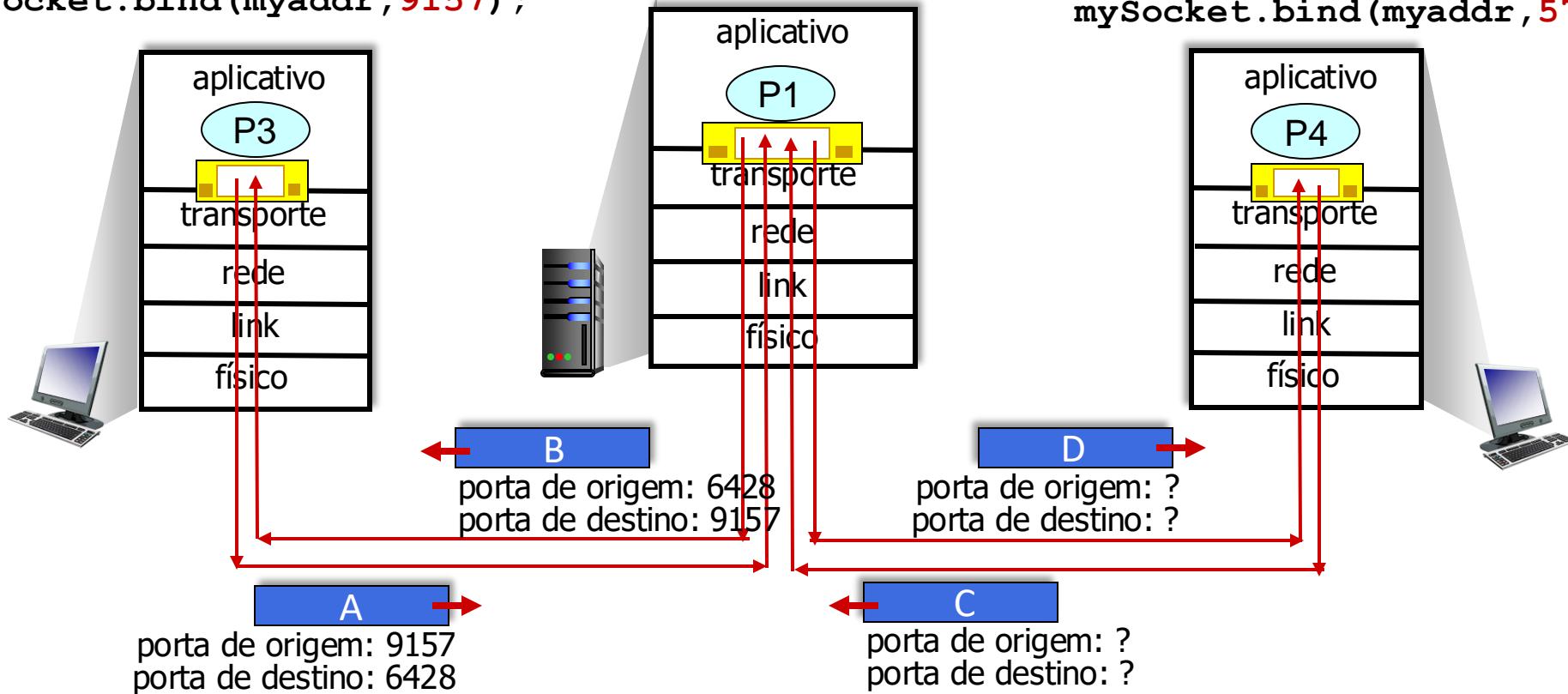
Os datagramas IP/UDP com o *mesmo número de porta de destino*, mas com endereços IP de origem e/ou números de porta de origem diferentes, serão direcionados para o *mesmo soquete* no host receptor

# Demultiplexação sem conexão: um exemplo

```
mySocket =  
    socket(AF_INET, SOCK_DGRAM)  
mySocket.bind(myaddr, 6428);
```

```
mySocket =  
    socket(AF_INET, SOCK_STREAM)  
mySocket.bind(myaddr, 9157);
```

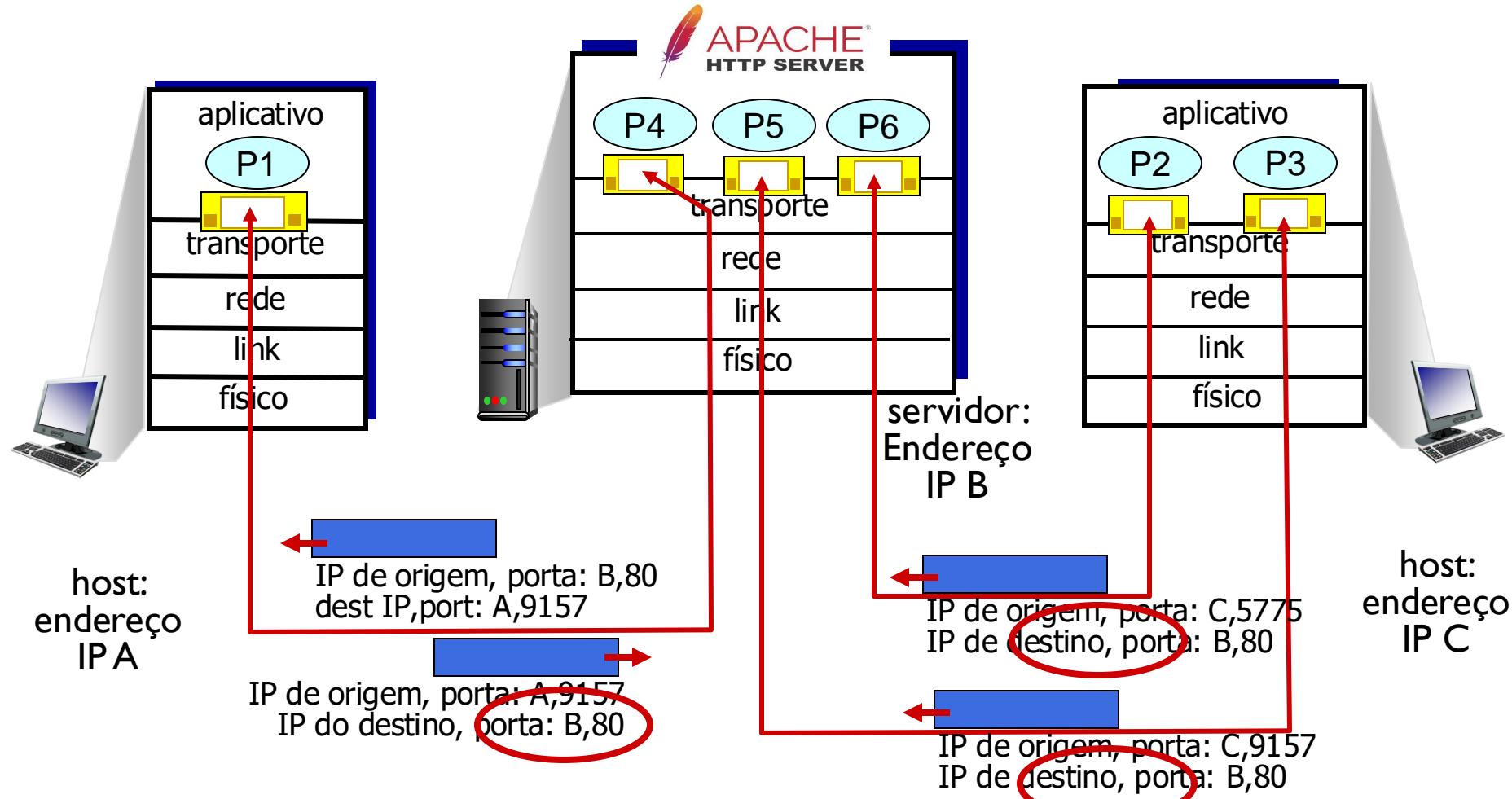
```
mySocket =  
    socket(AF_INET, SOCK_STREAM)  
mySocket.bind(myaddr, 5775);
```



# Demultiplexação orientada por conexão

- Soquete TCP identificado por **4 tuplas**:
  - endereço IP de origem
  - número da porta de origem
  - endereço IP de destino
  - número da porta de destino
- demux: o receptor usa ***todos os quatro valores (4 tuplas)*** para direcionar o segmento para o soquete apropriado
- pode suportar muitos soquetes TCP simultâneos:
  - cada soquete identificado por sua própria 4-tupla
  - cada soquete associado a um cliente de conexão diferente

# Demultiplexação orientada por conexão: exemplo



Três segmentos, todos destinados ao endereço IP: B,  
dest port: 80 são demultiplexados para *diferentes* soquetes

# Resumo

- Multiplexação, demultiplexação: com base no segmento, valores de campo do cabeçalho do datagrama
- **UDP:** demultiplexação usando o número da porta de destino (somente)
- **TCP:** demultiplexação usando 4 tuplas: endereços IP de origem e destino e números de porta
- A multiplexação/demultiplexação ocorre em *todas as camadas*

# Capítulo 3: roteiro

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



# UDP: Protocolo de datagrama do usuário

- Protocolo de transporte da Internet "sem frescuras", "básico"
- serviço de "melhor esforço", os segmentos UDP podem ser:
  - perdido
  - entregues fora de ordem no aplicativo
- *sem conexão:*
  - sem handshaking entre o remetente e o receptor UDP
  - cada segmento UDP é tratado independentemente dos outros

## Por que existe um UDP?

- sem estabelecimento de conexão (o que pode aumentar o atraso de RTT)
- simples: sem estado de conexão no remetente e no receptor
- tamanho pequeno do cabeçalho
- sem controle de congestionamento
  - O UDP pode ser lançado tão rápido quanto desejado!
  - pode funcionar em caso de congestionamento

# UDP: Protocolo de datagrama do usuário

- Uso de UDP:
  - aplicativos multimídia de streaming (tolerante a perdas, sensível à taxa)
  - DNS
  - SNMP
  - HTTP/3
- se for necessária uma transferência confiável por UDP (por exemplo, HTTP/3):
  - adicionar a confiabilidade necessária na camada de aplicativos
  - adicionar controle de congestionamento na camada de aplicativos

# UDP: Protocolo de datagrama do usuário [RFC 768]

INTERNET STANDARD  
RFC 768 J. Postel  
ISI  
28 August 1980

## User Datagram Protocol

---

### Introduction

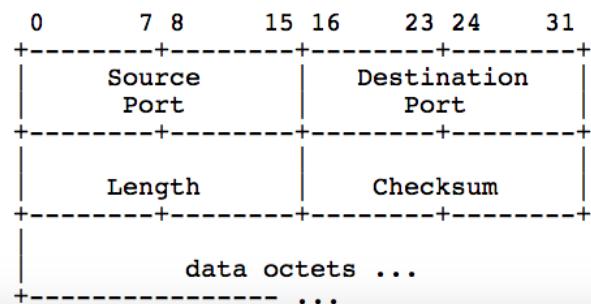
---

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.

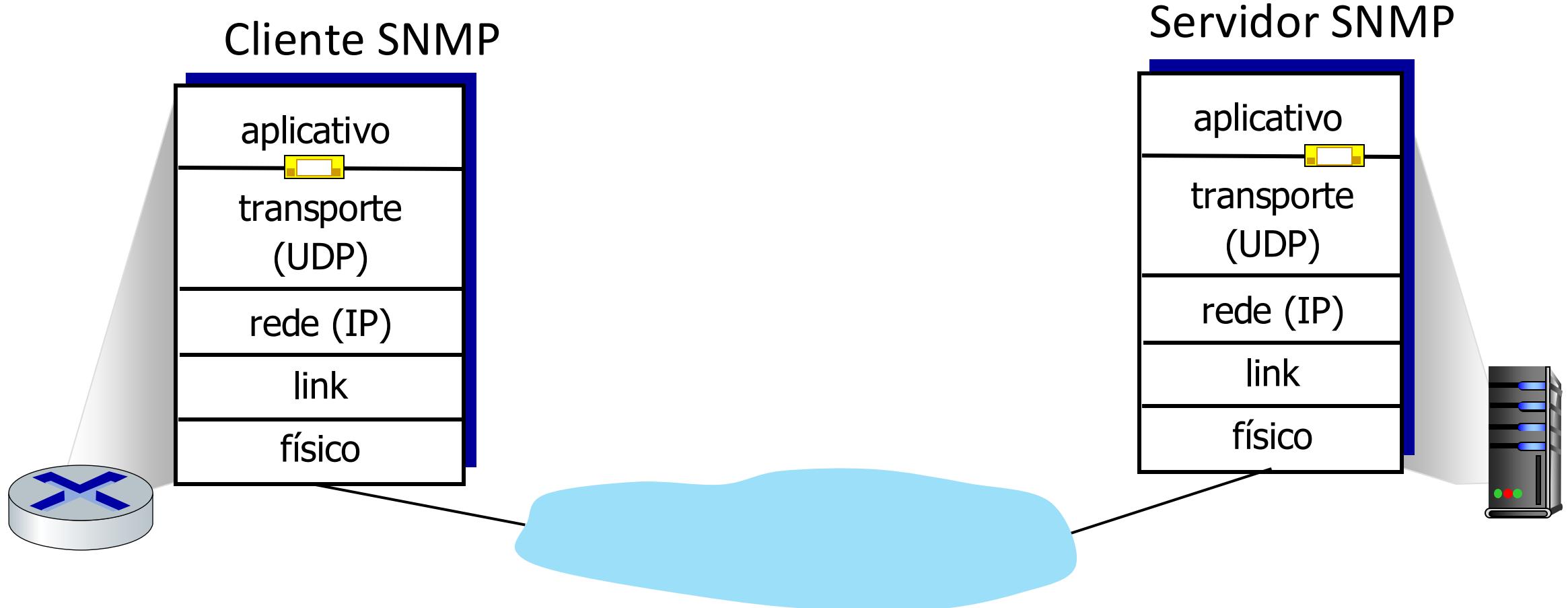
This protocol provides a procedure for application programs to send messages to other programs with a minimum of protocol mechanism. The protocol is transaction oriented, and delivery and duplicate protection are not guaranteed. Applications requiring ordered reliable delivery of streams of data should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

### Format

---



# UDP: Ações da camada de transporte



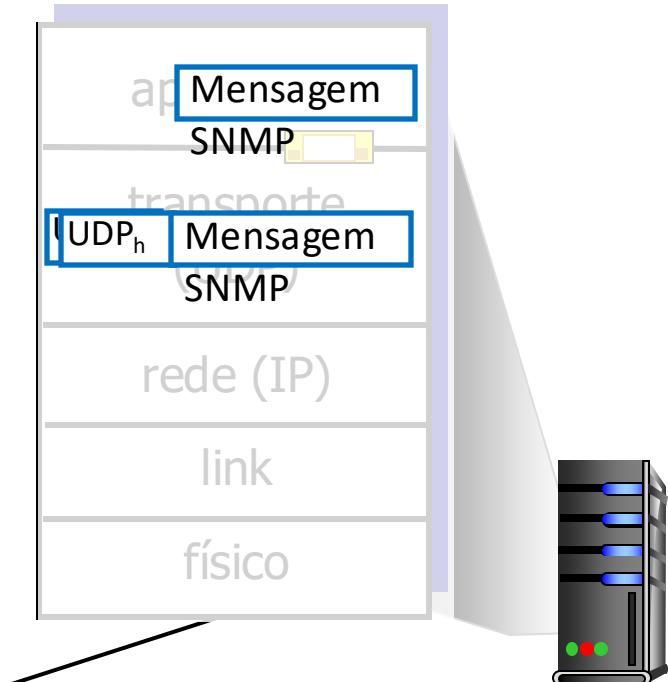
# UDP: Ações da camada de transporte



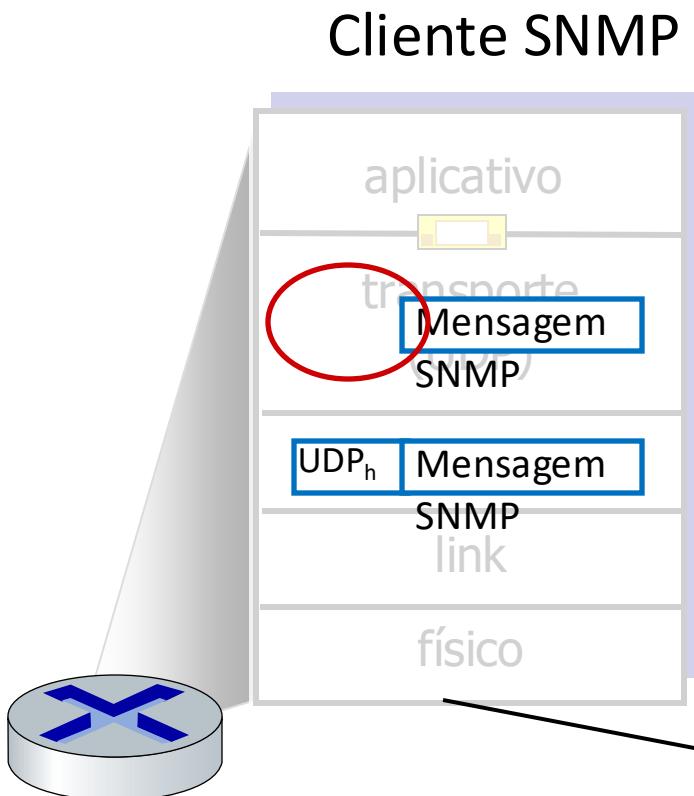
## Ações do remetente UDP:

- recebe uma mensagem da camada do aplicativo
- determina os valores dos campos do cabeçalho do segmento UDP
- cria segmento UDP
- passa o segmento para o IP

## Servidor SNMP



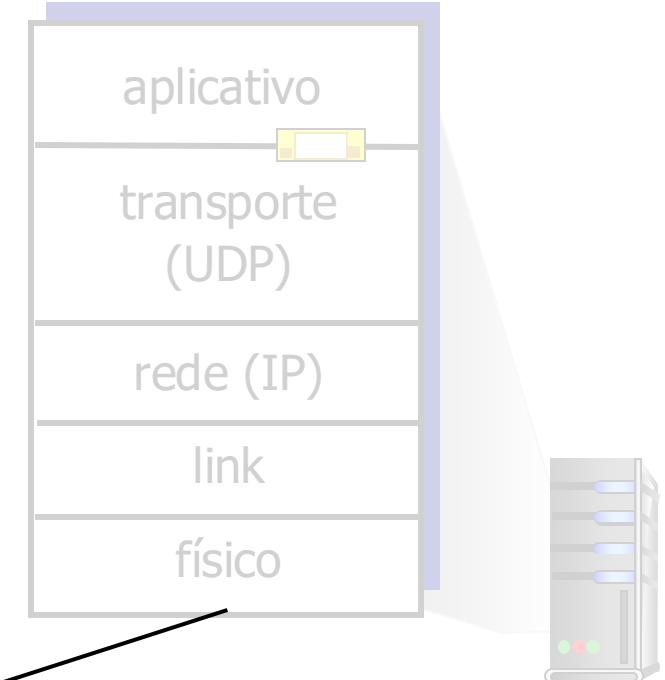
# UDP: Ações da camada de transporte



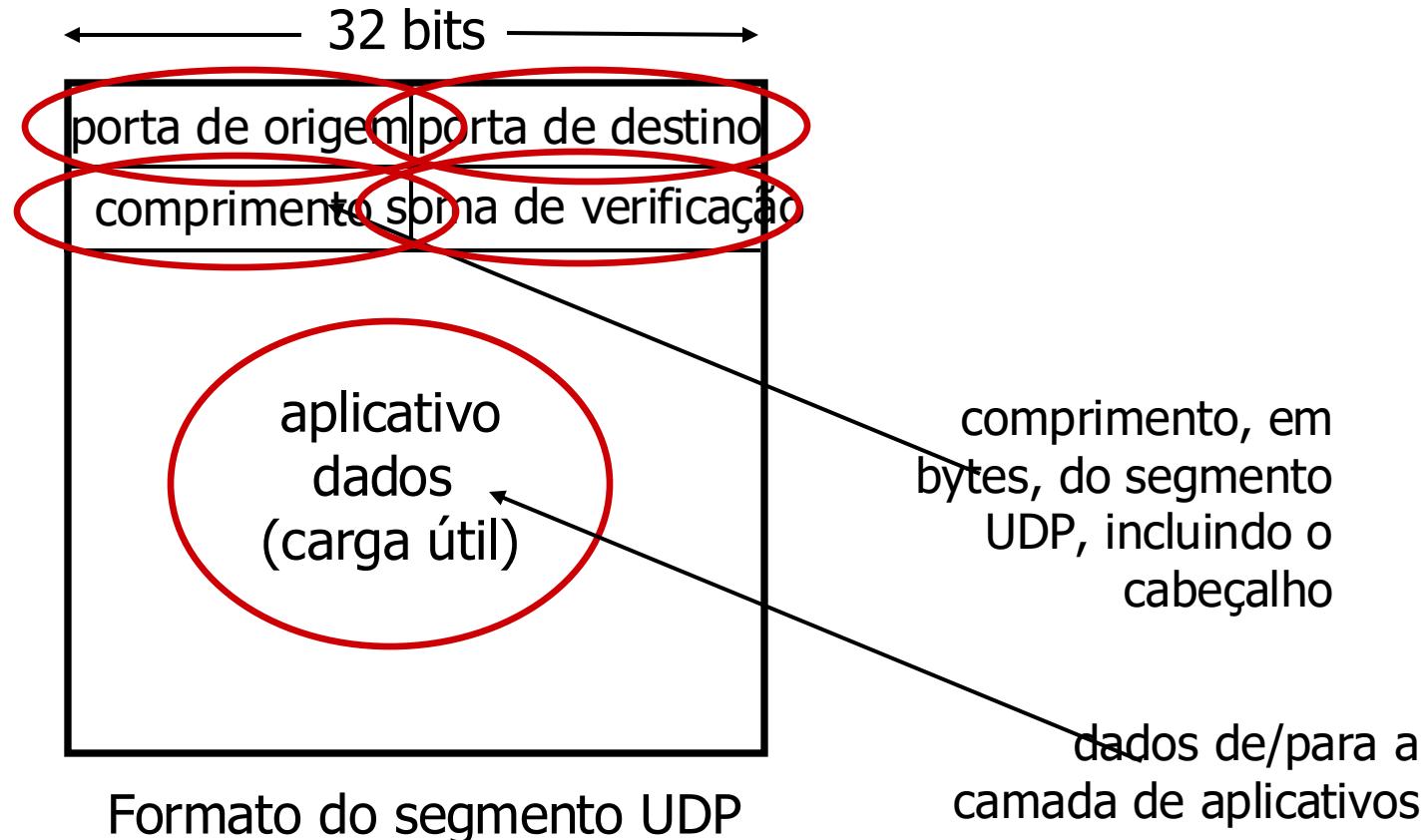
## Ações do receptor UDP:

- recebe segmento do IP
- verifica o valor do cabeçalho da soma de verificação UDP
- extrai a mensagem da camada de aplicativo
- demultiplexa a mensagem para o aplicativo via soquete

## Servidor SNMP



# Cabeçalho do segmento UDP

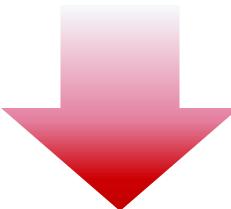


# Soma de verificação UDP

**Objetivo:** detectar erros (ou seja, bits invertidos) no segmento transmitido

1<sup>st</sup> number      2<sup>nd</sup> number      soma

Transmitido: 5 6 11



Recebido:

4 6 11

calculado pelo receptor  
soma de verificação



computado pelo remetente  
soma de verificação (conforme recebido)

# Soma de verificação da Internet

**Objetivo:** detectar erros (ou seja, bits invertidos) no segmento transmitido

**remetente:**

- tratar o conteúdo do segmento UDP (incluindo campos de cabeçalho UDP e endereços IP) como uma sequência de inteiros de 16 bits
- **soma de verificação:** adição (soma do complemento de um) do conteúdo do segmento
- valor de soma de verificação colocado no campo de soma de verificação UDP

**receptor:**

- computar a soma de verificação do segmento recebido
- verifica se a soma de verificação computada é igual ao valor do campo de soma de verificação:
  - não é igual - erro detectado
  - igual - nenhum erro detectado. *Mas, mesmo assim, talvez haja erros? Mais informações em ....*

# checksum 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
<hr/>																
envolvente	1	1	0	1	1	1	0	1	1	1	0	1	1	0	1	1
<hr/>																
soma	1	0	1	1	1	0	1	1	1	0	1	1	1	0	0	
soma de verificação	0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

*Observação:* ao somar números, um carryout do bit mais significativo precisa ser adicionado ao resultado

\* Confira os exercícios interativos on-line para obter mais exemplos: [http://gaia.cs.umass.edu/kurose\\_ross/interactive/](http://gaia.cs.umass.edu/kurose_ross/interactive/)

# Checksum 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	0 1
	1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1	1 0
	—————	
envolvente	1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 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 0 1 0 0 0 1 1	

Mesmo que os números tenham mudado (inversão de bits), **não há** alteração na soma de verificação!

# Resumo: UDP

- Protocolo "sem frescuras":
  - os segmentos podem ser perdidos, entregues fora de ordem
  - serviço de melhor esforço: "enviar e esperar o melhor"
- O UDP tem suas vantagens:
  - não é necessária nenhuma configuração/handshaking (não há RTT incorrido)
  - pode funcionar quando o serviço de rede está comprometido
  - ajuda na confiabilidade (checksum)
- criar funcionalidade adicional sobre o UDP na camada de aplicativos (por exemplo, HTTP/3)

# Capítulo 3: roteiro

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

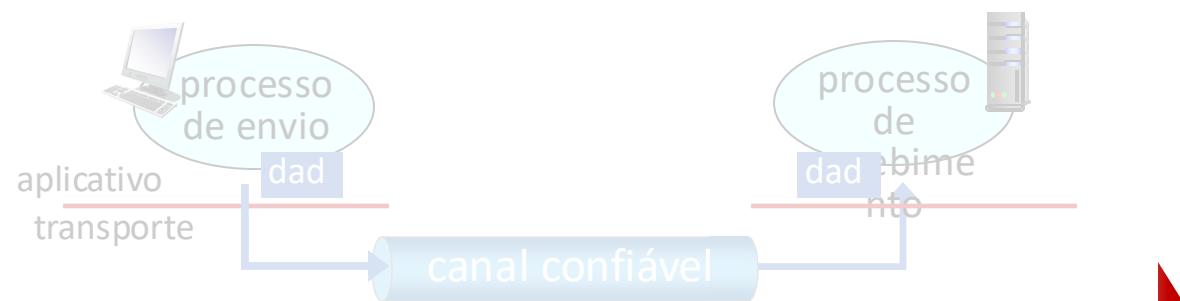


# Princípios de transferência confiável de dados

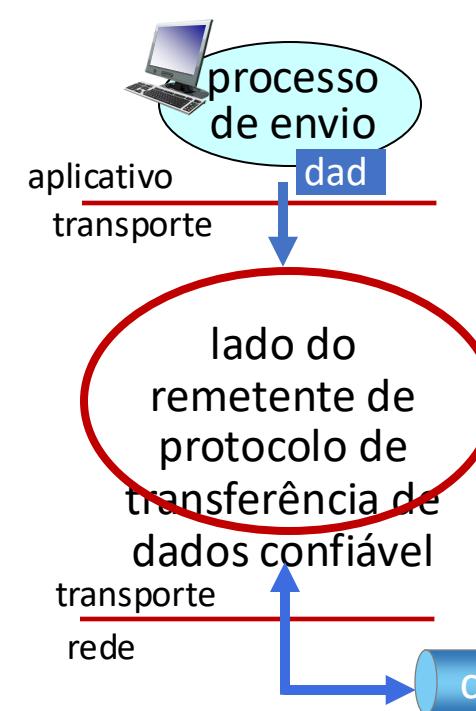
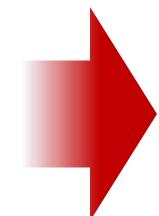


*abstração* de serviços  
confiáveis

# Princípios da transferência confiável de dados



*abstração de serviços  
confiáveis*

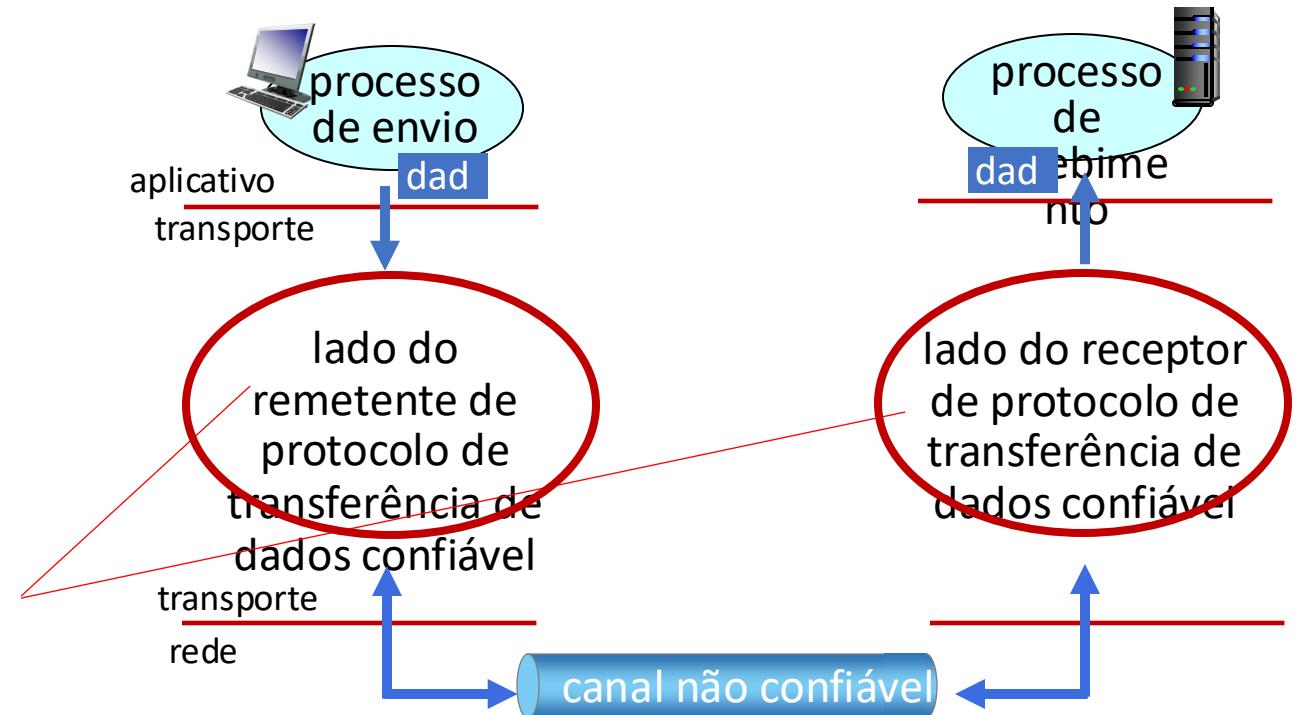


*implementação de serviços  
confiáveis*



# Princípios de transferência confiável de dados

A complexidade do protocolo de transferência de dados confiável dependerá (fortemente) das características do canal não confiável (perder, corromper, reordenar dados?)

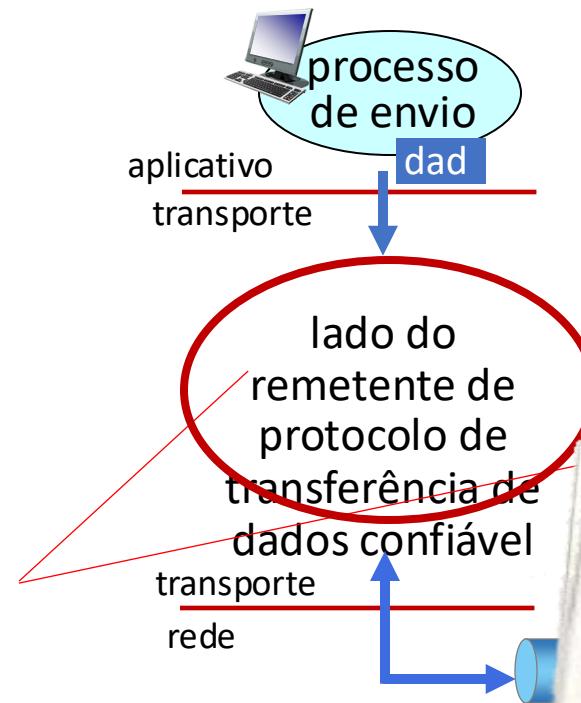


*implementação* de serviços confiáveis

# Princípios de transferência confiável de dados

O remetente e o destinatário *não sabem* o "estado" um do outro, por exemplo, se uma mensagem foi recebida?

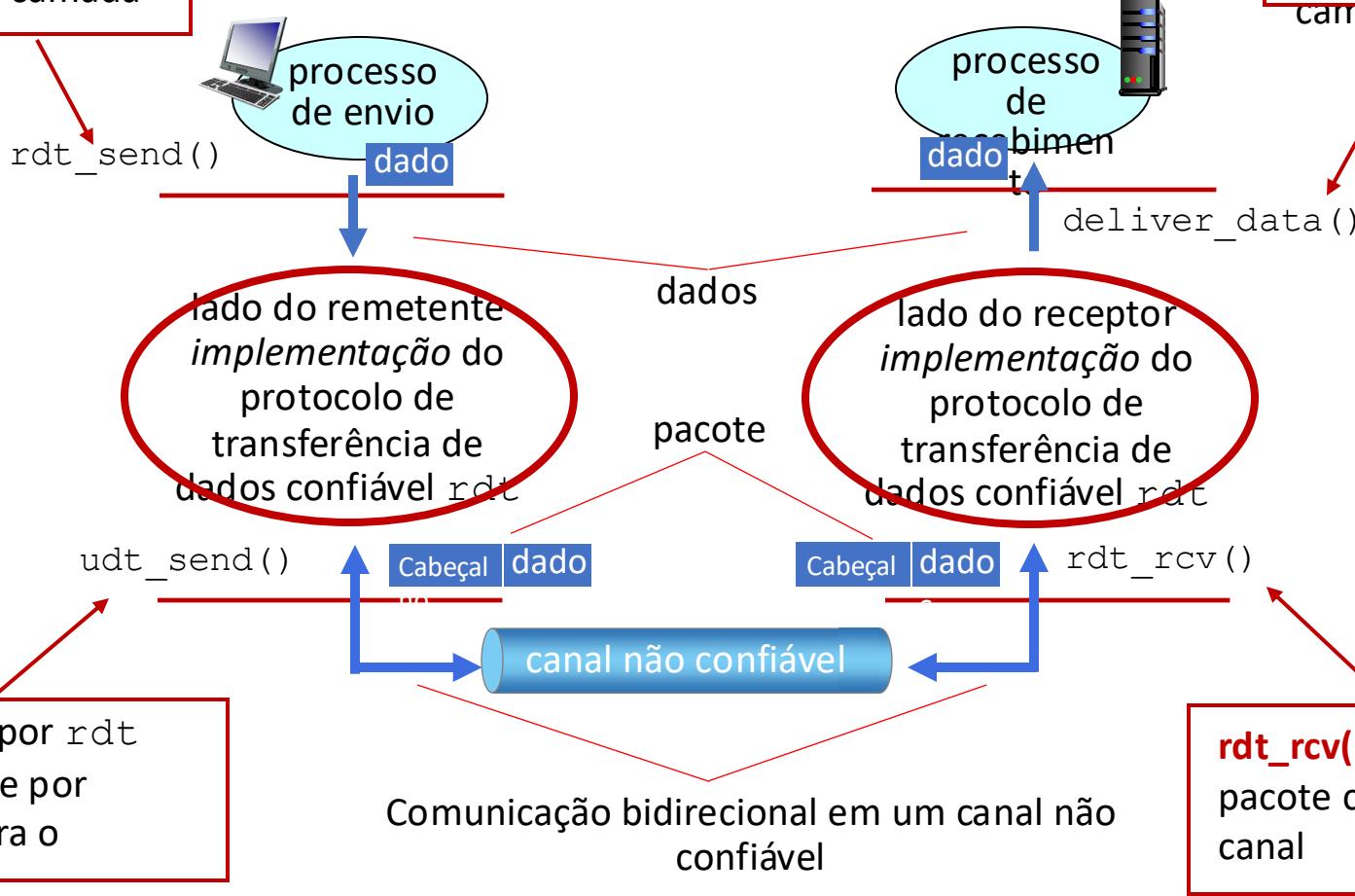
- a menos que seja comunicado por meio de uma mensagem



*implementação* de serviços  
confiáveis

# Protocolo de transferência confiável de dados (rdt): interfaces

**rdt\_send()**: chamada de cima (por exemplo, pelo aplicativo). Dados passados para entregar à camada superior do receptor

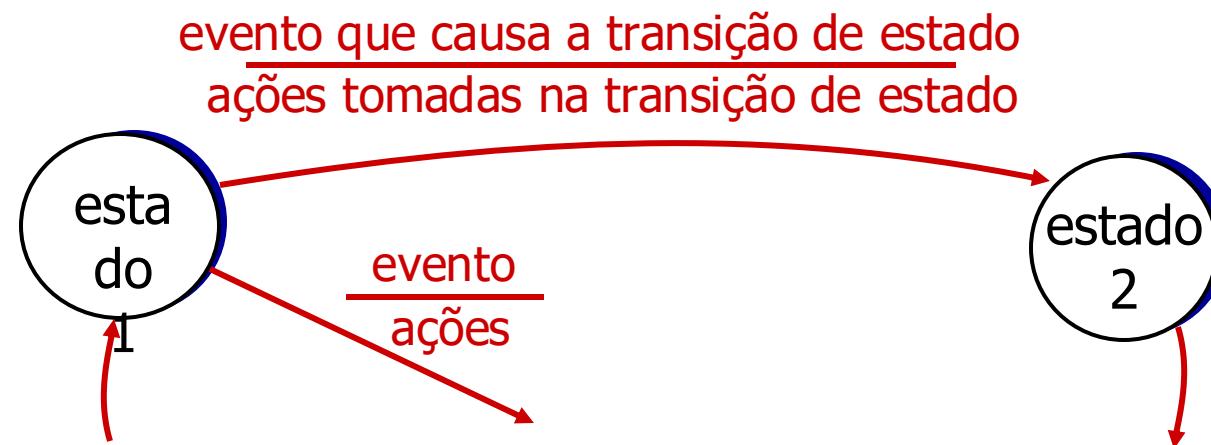


# Transferência confiável de dados: primeiros passos

Nós o faremos:

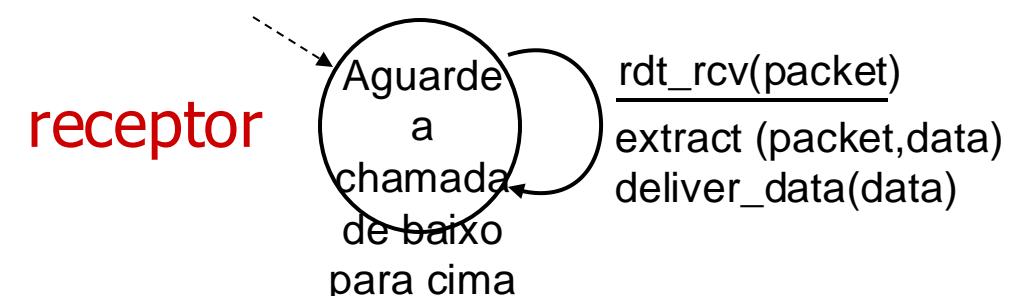
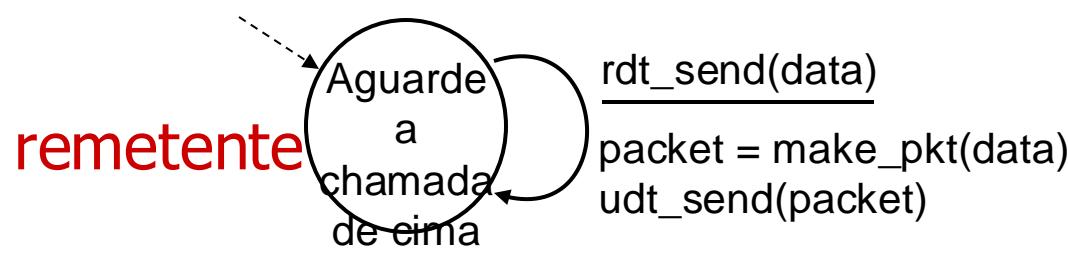
- desenvolver de forma incremental os lados do remetente e do receptor do protocolo de transferência de dados confiável (rdt)
- considerar apenas a transferência unidirecional de dados
  - mas as informações de controle fluirão em ambas as direções!
- usar máquinas de estado finito (FSM) para especificar o remetente, o receptor

**estado:** quando estiver nesse "estado", o próximo estado será determinado exclusivamente pelo próximo evento



# rdt1.0: transferência confiável em um canal confiável

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



# rdt2.0: canal com erros de bit

- o canal subjacente pode inverter os bits do pacote
  - soma de verificação (por exemplo, soma de verificação da Internet) para detectar erros de bits
- A pergunta: como se recuperar de erros?

*Como os seres humanos se recuperam de "erros" durante uma conversa?*

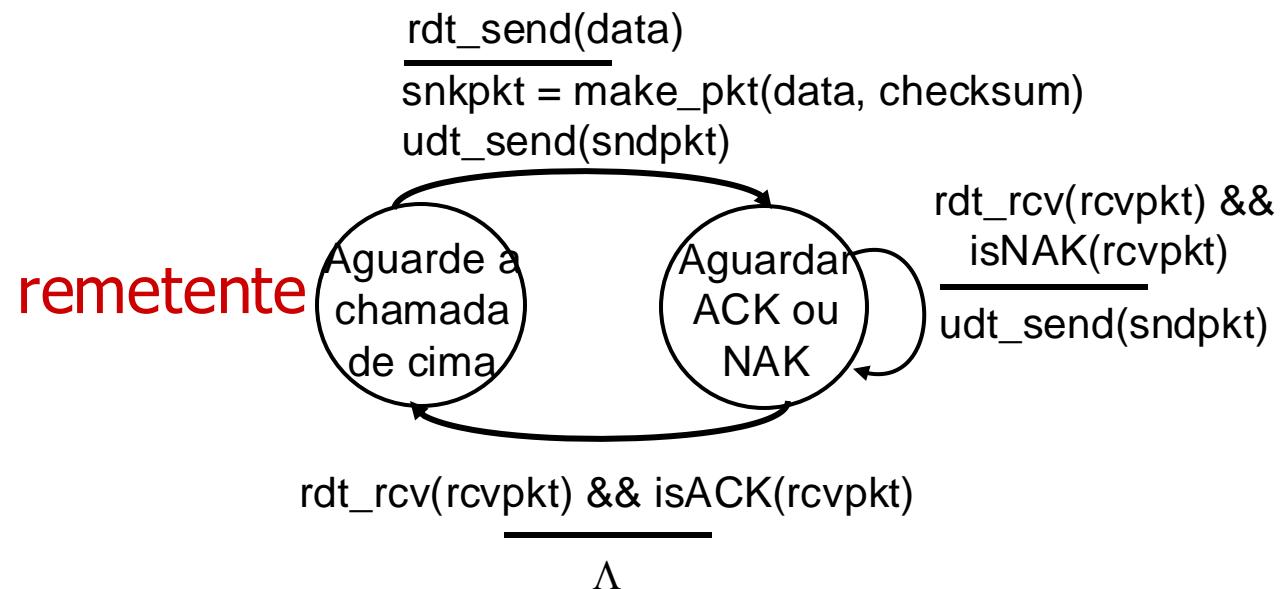
# rdt2.0: canal com erros de bit

- o canal subjacente pode inverter os bits do pacote
  - soma de verificação para detectar erros de bits
- A pergunta: como se recuperar de erros?
  - *confirmações (ACKs)*: o receptor informa explicitamente ao remetente que o pkt foi recebido OK
  - *confirmações negativas (NAKs)*: o receptor informa explicitamente ao remetente que o pkt tinha erros
  - o remetente *retransmite* o pkt ao receber o NAK

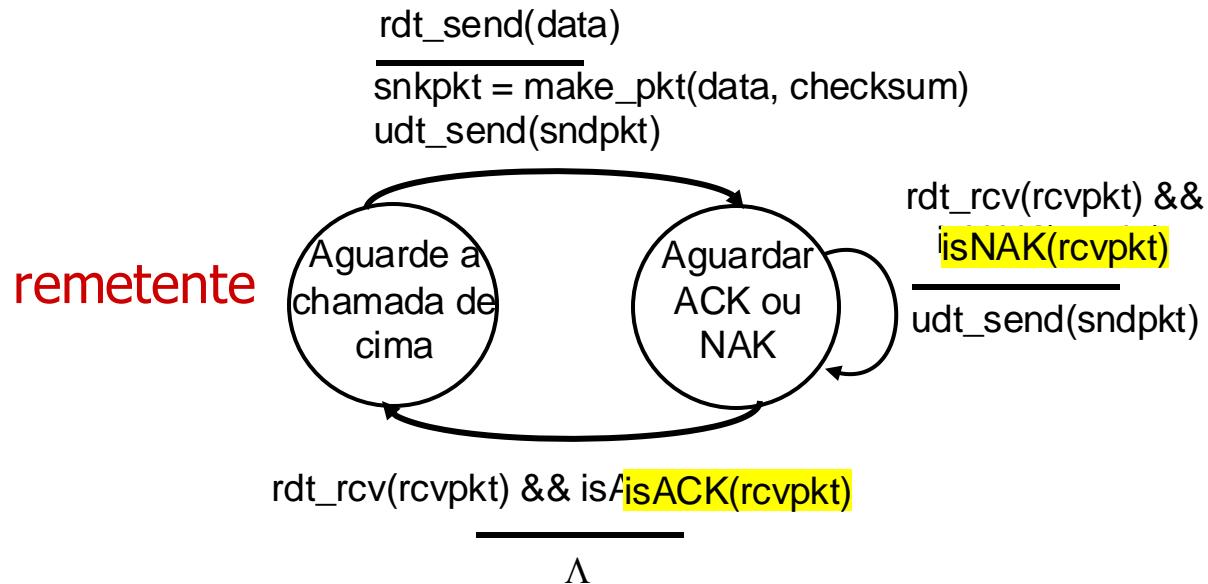
parar e esperar

o remetente envia um pacote e aguarda a resposta do receptor

# rdt2.0: Especificações do FSM



# rdt2.0: Especificação de FSM

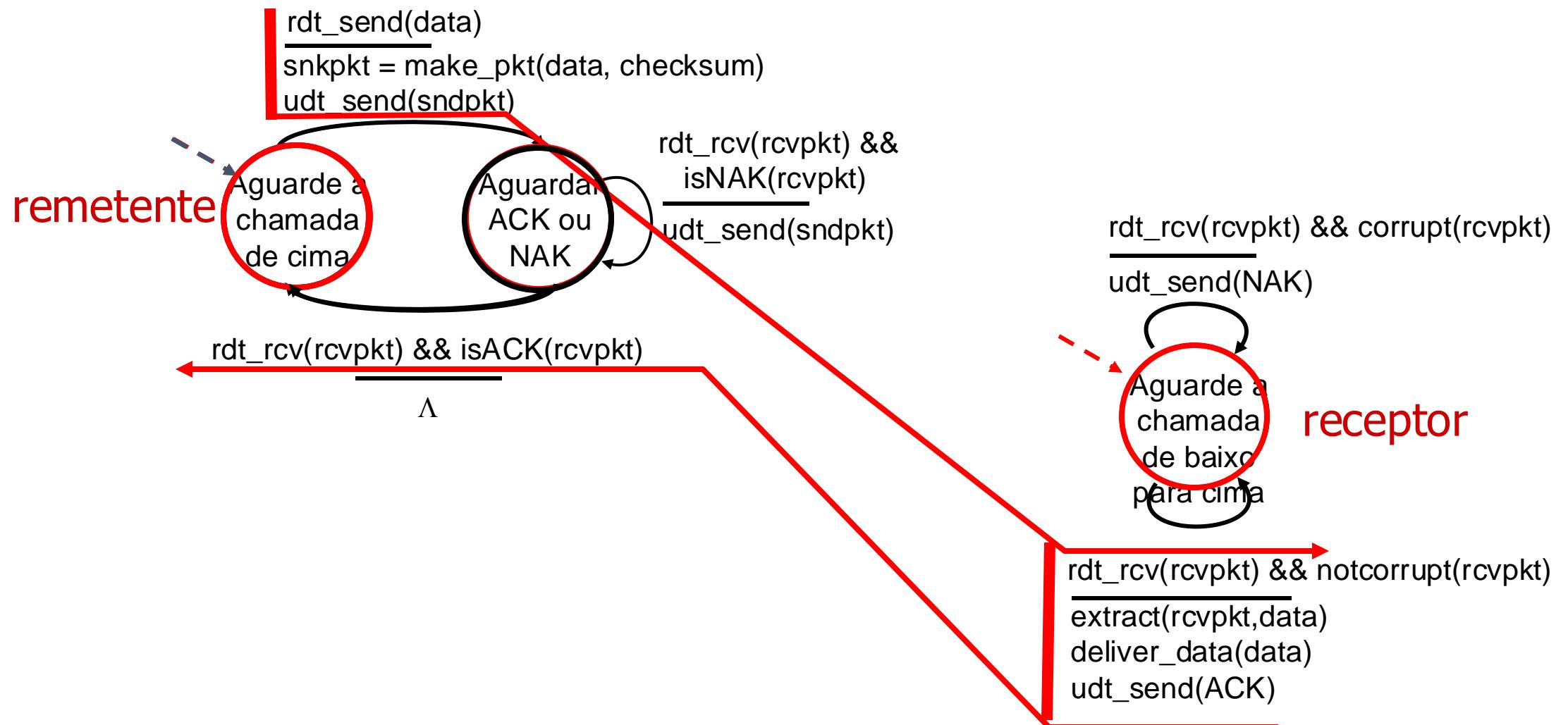


**Observação:** o "estado" do receptor (o receptor recebeu minha mensagem corretamente?) não é conhecido pelo remetente, a menos que seja comunicado de alguma forma do receptor para o remetente

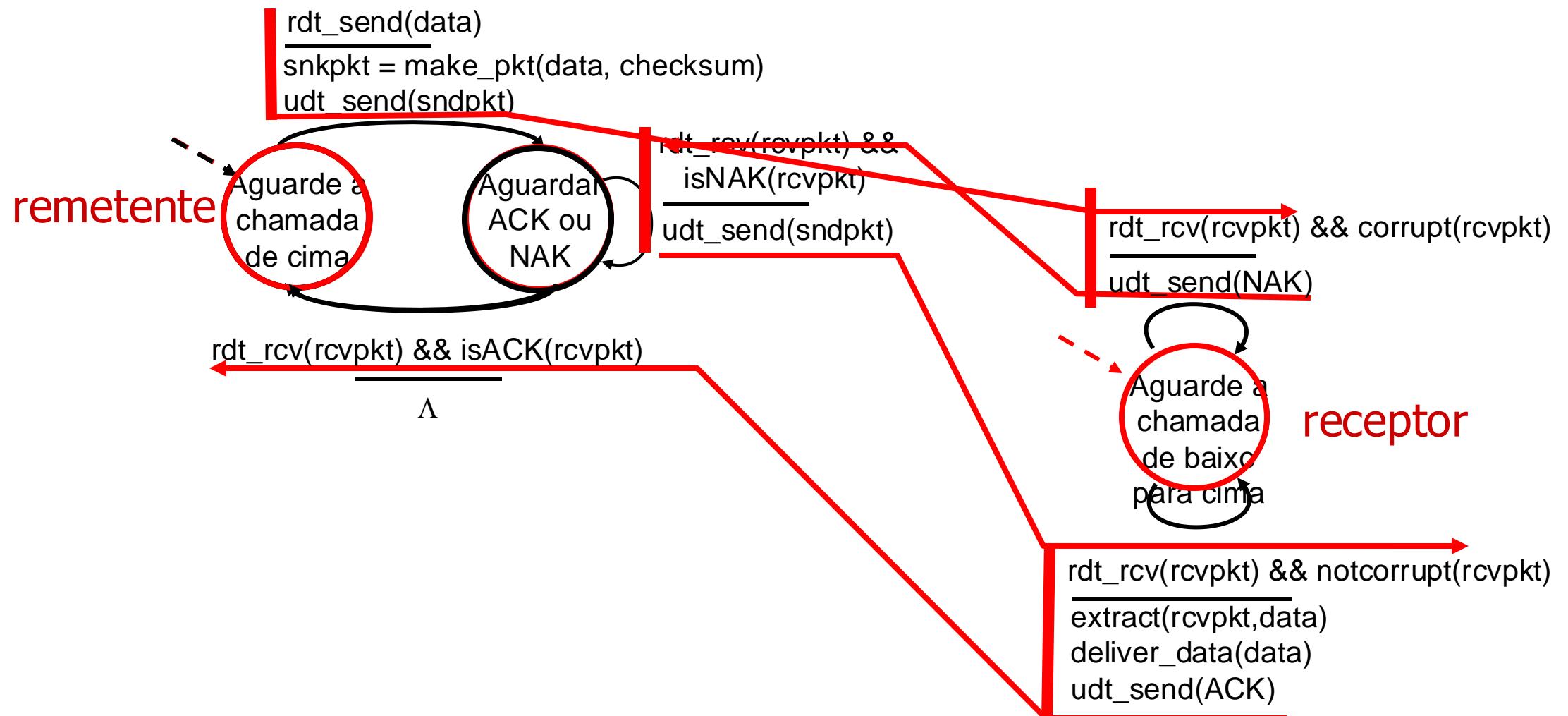
- É por isso que precisamos de um protocolo!



# rdt2.0: operação sem erros



# rdt2.0: cenário de pacote corrompido



# O rdt2.0 tem uma falha fatal!

O que acontece se o ACK/NAK for corrompido?

- o remetente não sabe o que aconteceu no destinatário!
- não pode simplesmente retransmitir: possível duplicata

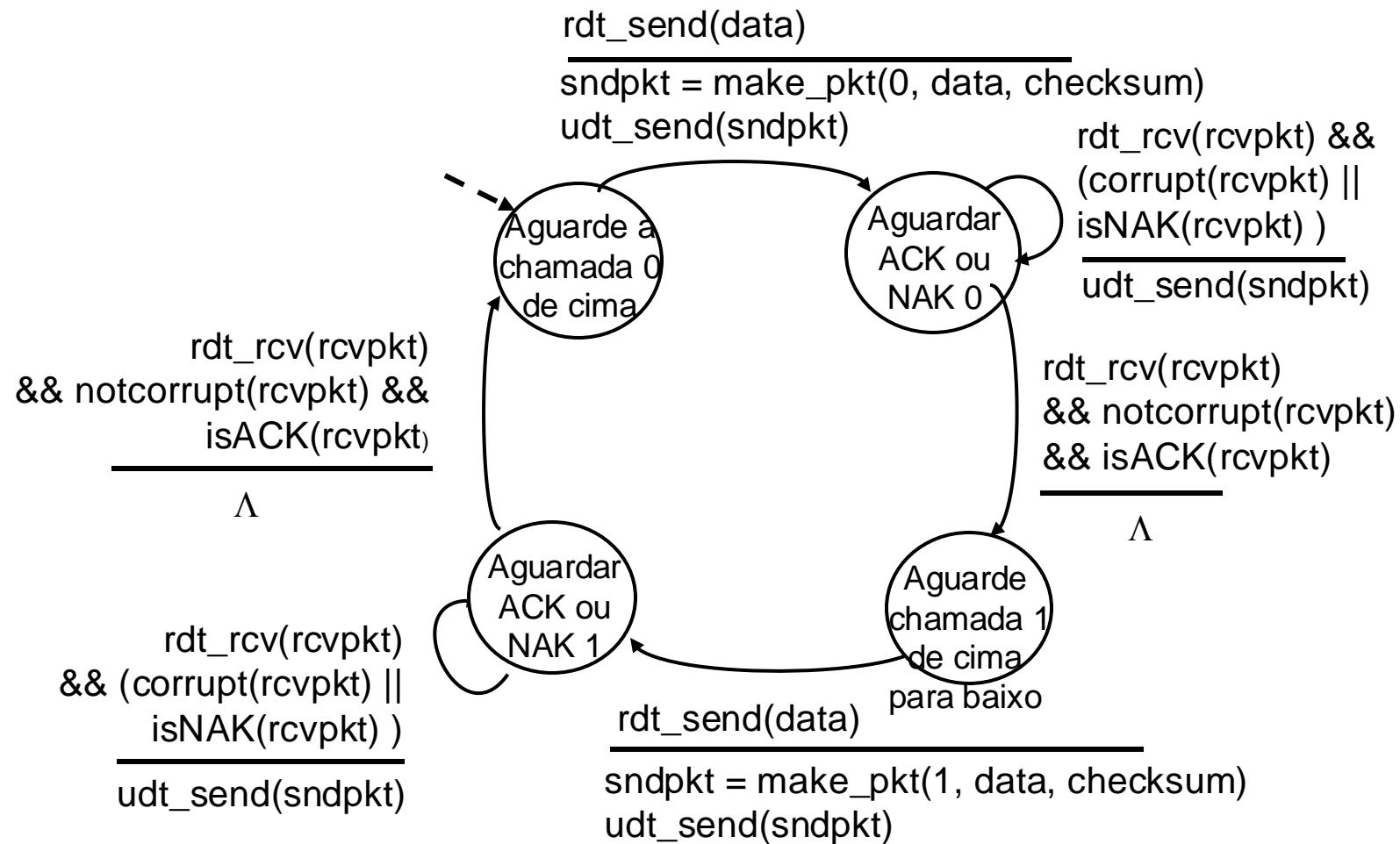
manuseio de duplicatas:

- o remetente retransmite o pkt atual se o ACK/NAK for corrompido
- o remetente adiciona *um número de sequência* a cada pkt
- o receptor descarta (não entrega) o pkt duplicado

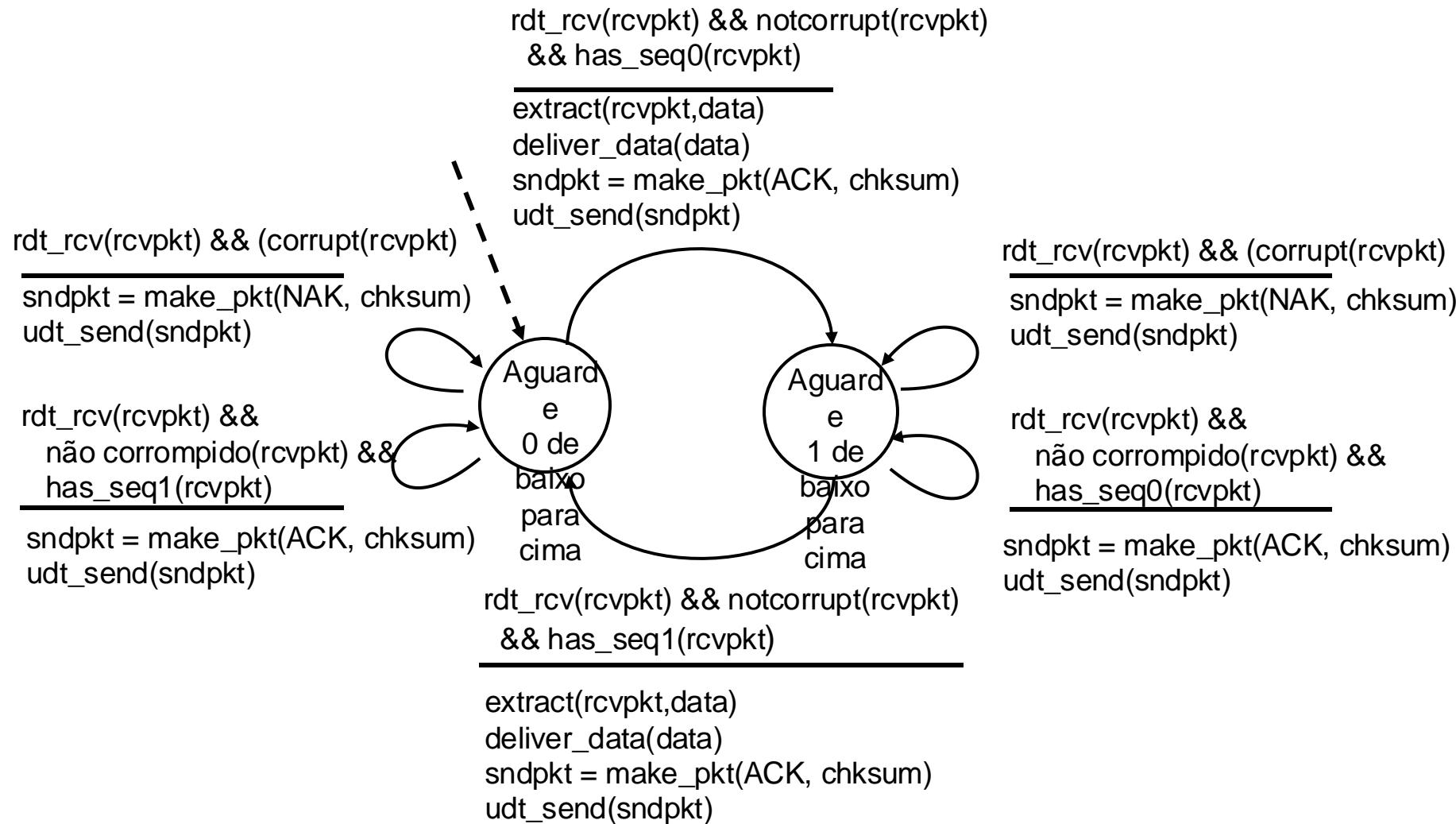
parar e esperar

o remetente envia um pacote e aguarda a resposta do receptor

# rdt2.1: remetente, manipulando ACK/NAKs distorcidos



# rdt2.1: receptor, manuseio de ACK/NAKs distorcidos



# rdt2.1: discussão

## remetente:

- seq # adicionado ao pkt
- dois seq. #s (0,1) serão suficientes. Por quê?
- deve verificar se o ACK/NAK recebido está corrompido
- duas vezes mais estados
  - o estado deve "lembrar" se o pkt "esperado" deve ter o número de seq. 0 ou 1

## receptor:

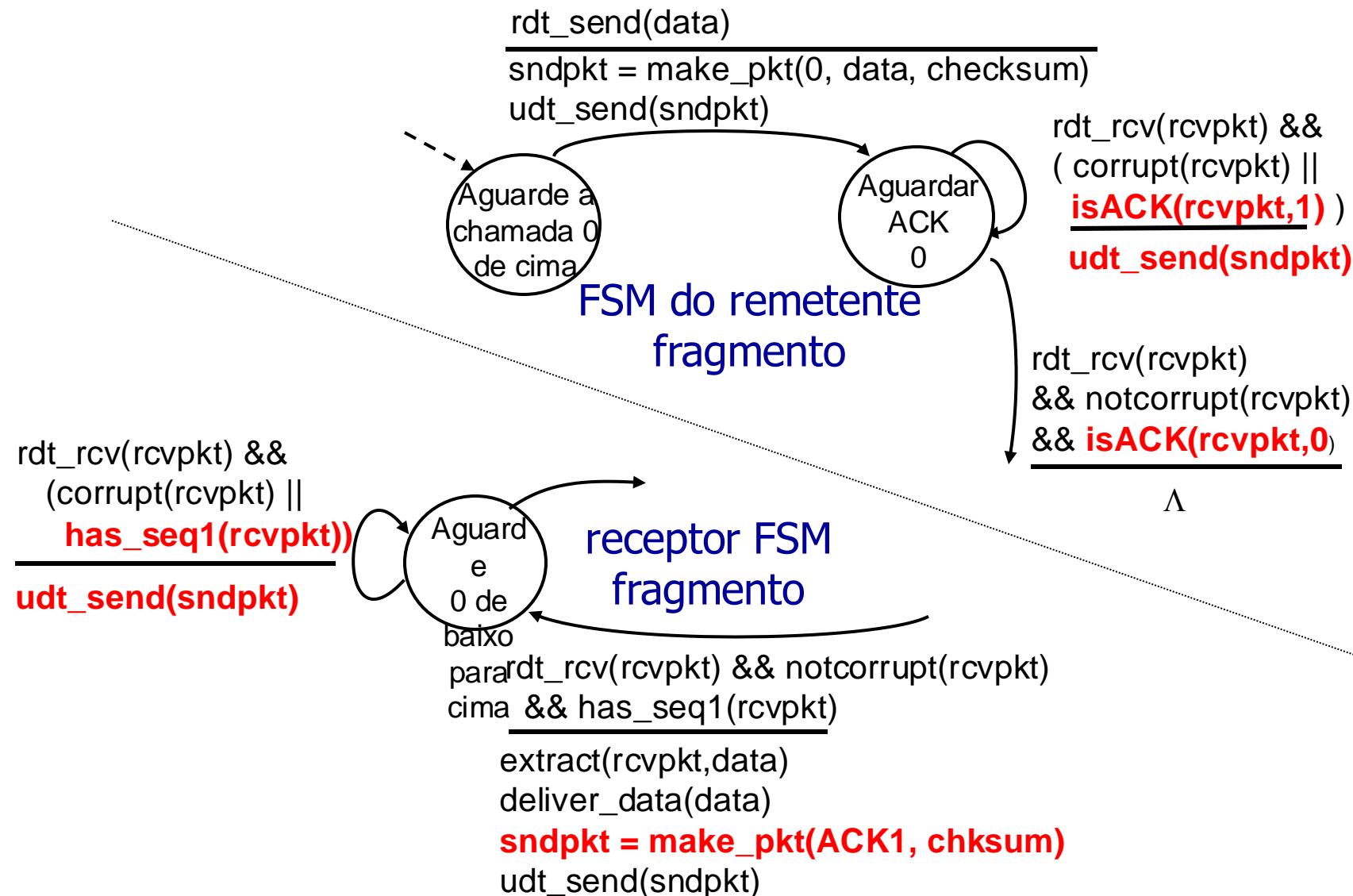
- deve verificar se o pacote recebido é duplicado
  - O estado indica se é esperado 0 ou 1 pkt seq #
- Observação: o receptor *não* pode saber se seu último ACK/NAK foi recebido com OK pelo remetente

# rdt2.2: um protocolo sem NAK

- a mesma funcionalidade do rdt2.1, usando apenas ACKs
- em vez de NAK, o receptor envia ACK para o último pkt recebido OK
  - o receptor deve incluir *explicitamente* o número de seq. do pkt que está sendo devolvido
- ACK duplicado no remetente resulta na mesma ação que o NAK:  
*retransmissão do pkt atual*

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

# rdt2.2: fragmentos de emissor e receptor



# rdt3.0: canais com erros e perdas

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

- soma de verificação, números de sequência, ACKs, retransmissões serão úteis... mas não o suficiente

*P:* Como os seres humanos lidam com a perda de palavras do remetente para o destinatário em uma conversa?

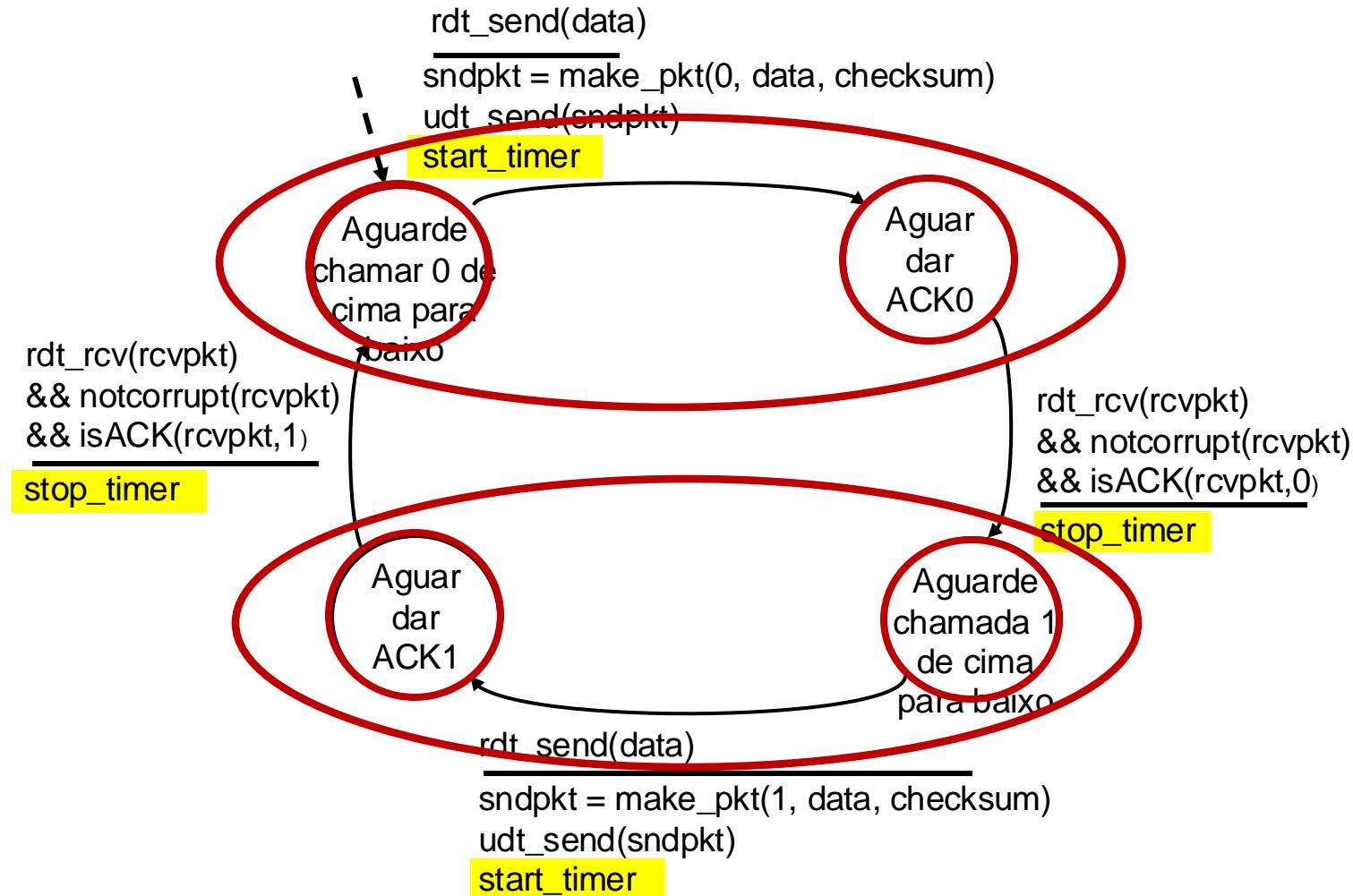
# rdt3.0: canais com erros e perdas

*Abordagem:* o remetente aguarda um período de tempo "razoável" pelo ACK

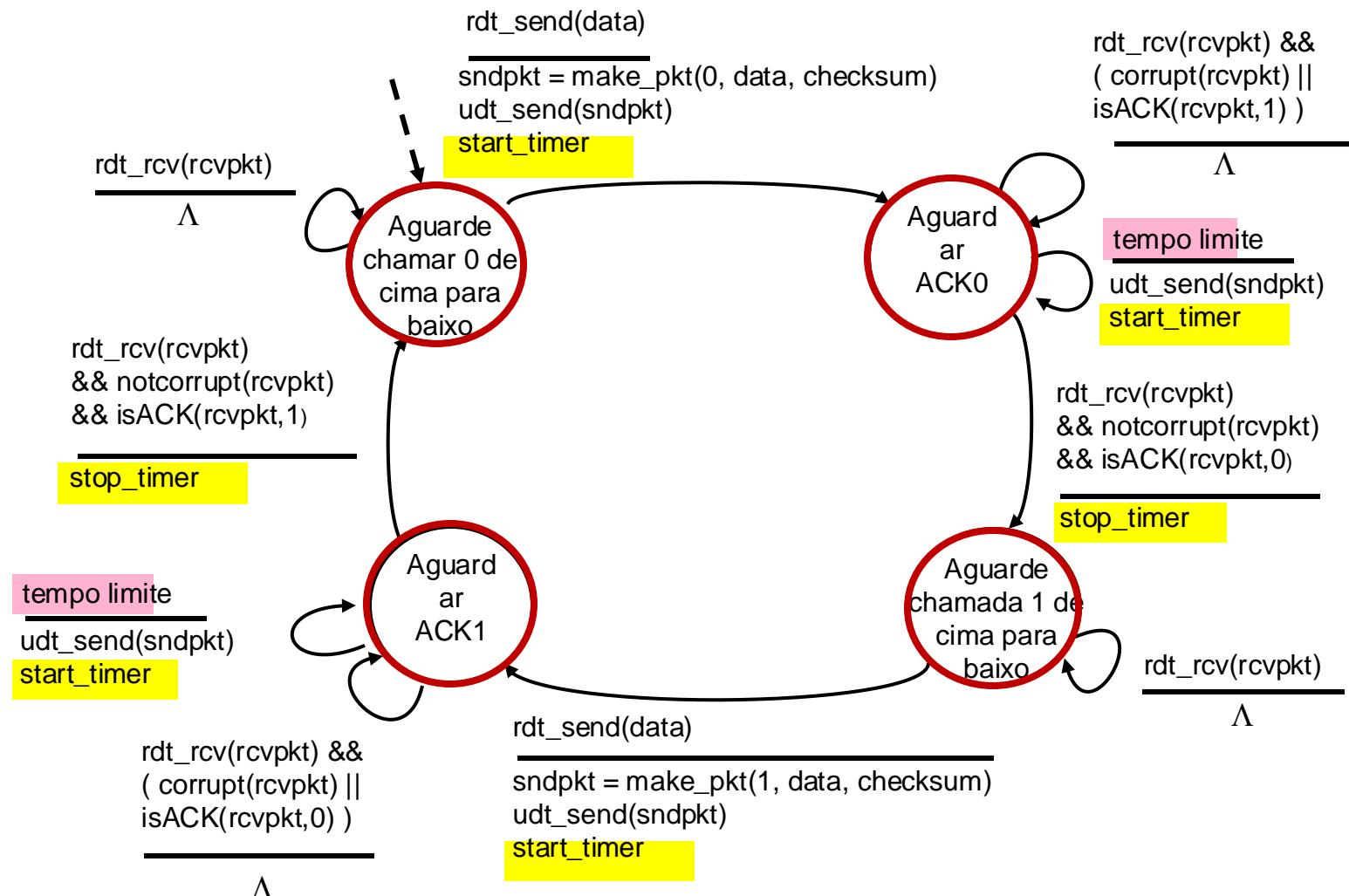
- retransmite se nenhum ACK for recebido nesse período
- se o pkt (ou ACK) apenas atrasou (não foi perdido):
  - a retransmissão será duplicada, mas o seq #s já lida com isso!
  - o receptor deve especificar o número da sequência do pacote que está sendo devolvido
- usar o cronômetro de contagem regressiva para interromper após um período de tempo "razoável"



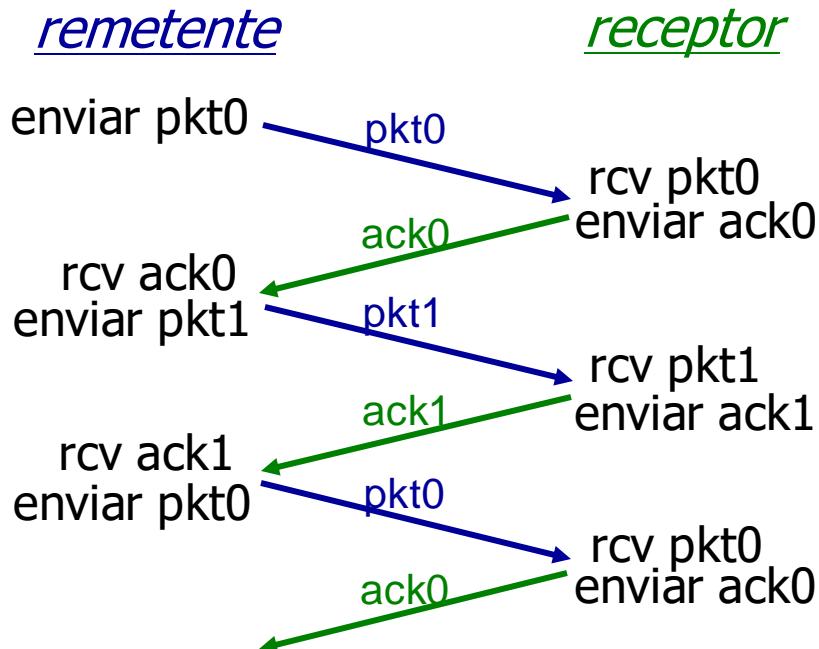
# remetente rdt3.0



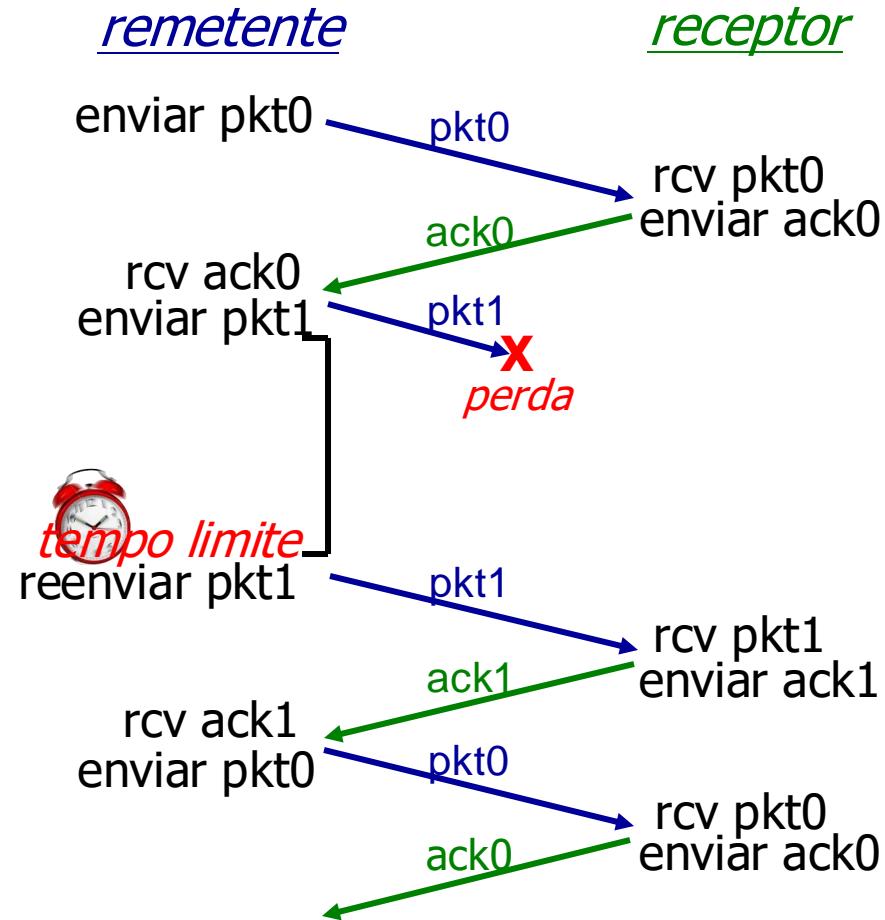
# remetente rdt3.0



# rdt3.0 em ação

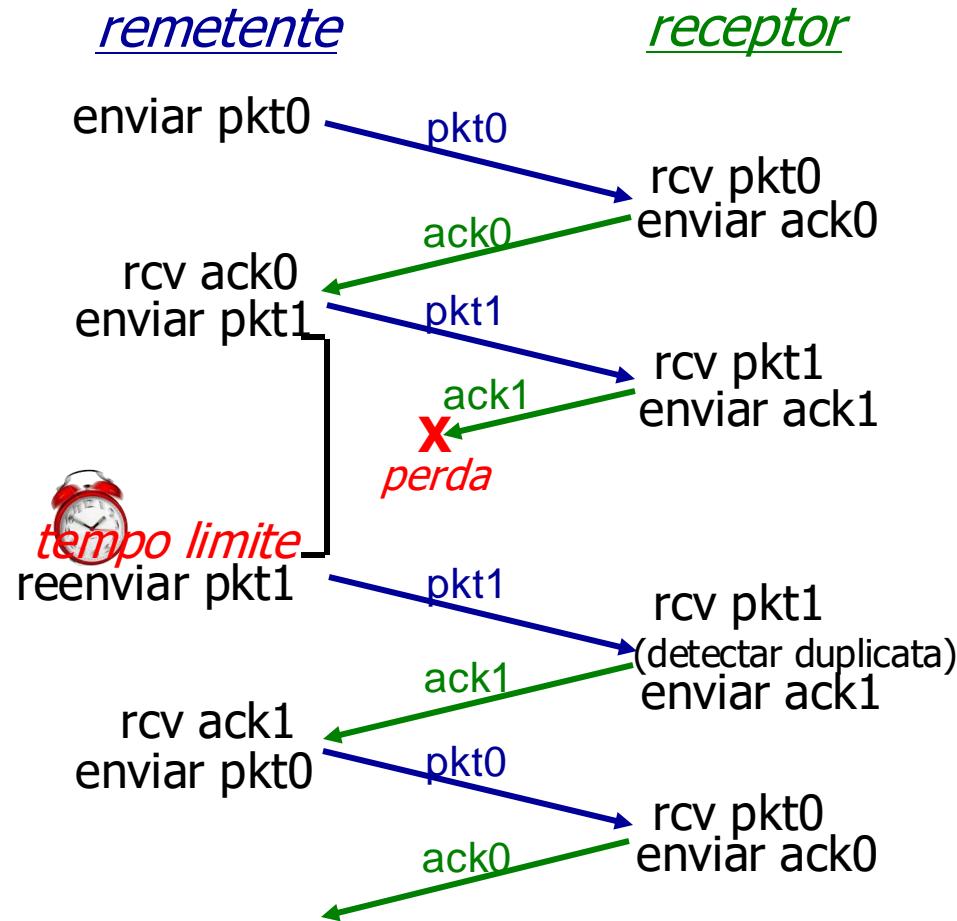


(a) nenhuma perda



(b) perda de pacotes

# rdt3.0 em ação

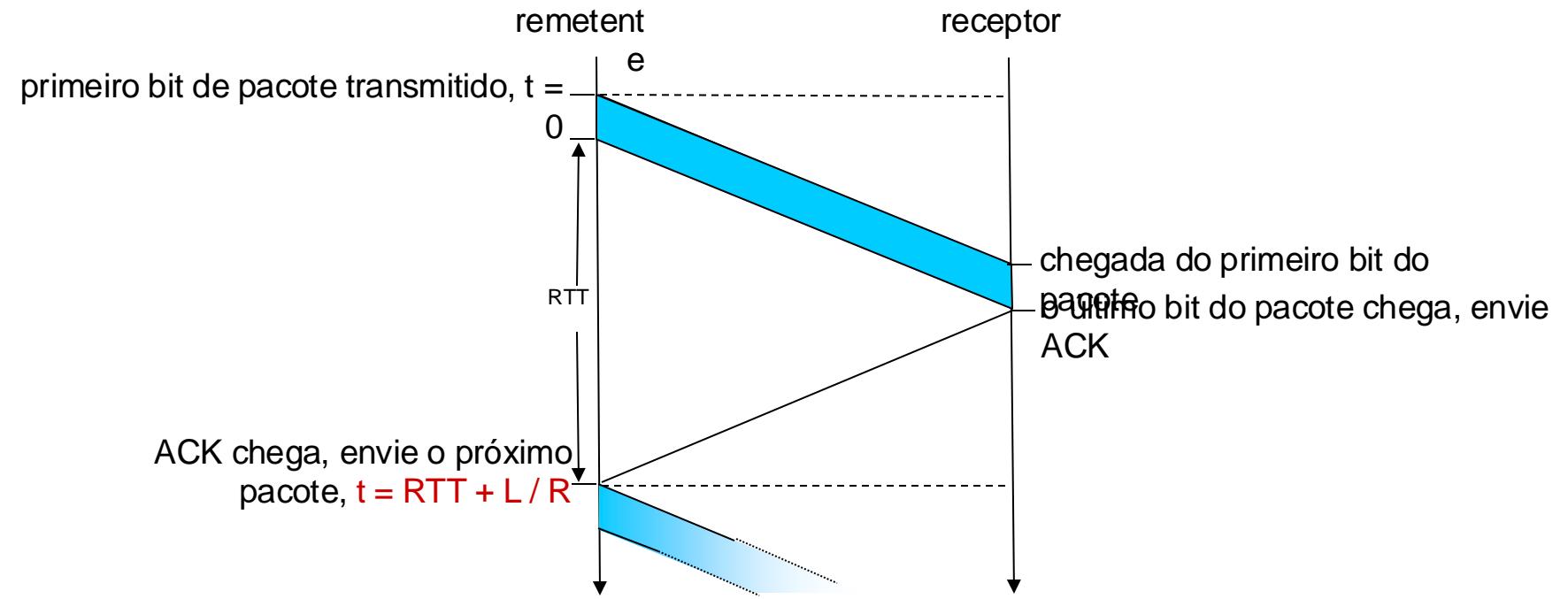


# Desempenho do rdt3.0 (stop-and-wait)

- $U_{remetente}$ : *utilização* - fração do tempo em que o remetente está ocupado enviando
- Exemplo: link de 1 Gbps, atraso de 15 ms, pacote de 8000 bits
  - tempo para transmitir o pacote para o canal:

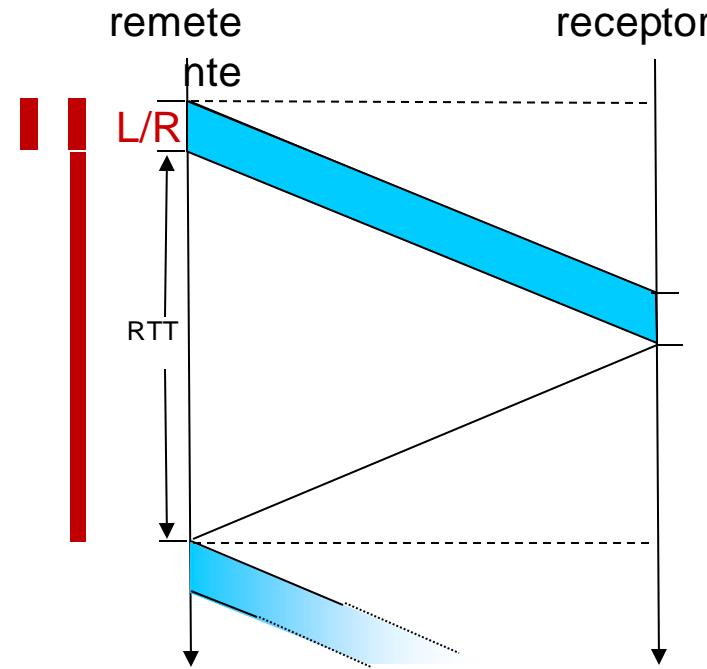
$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsegundos}$$

# rdt3.0: operação stop-and-wait



# rdt3.0: operação stop-and-wait

$$\begin{aligned} u_{\text{sender}} &= \frac{L / R}{RTT + L / R} \\ &= \frac{.008}{30.008} \\ &= 0.00027 \end{aligned}$$

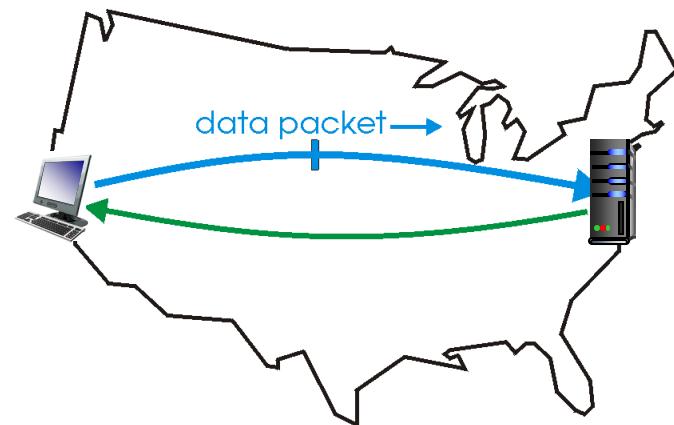


- O desempenho do protocolo rdt 3.0 é péssimo!
- O protocolo limita o desempenho da infraestrutura subjacente (canal)

# rdt3.0: operação de protocolos em pipeline

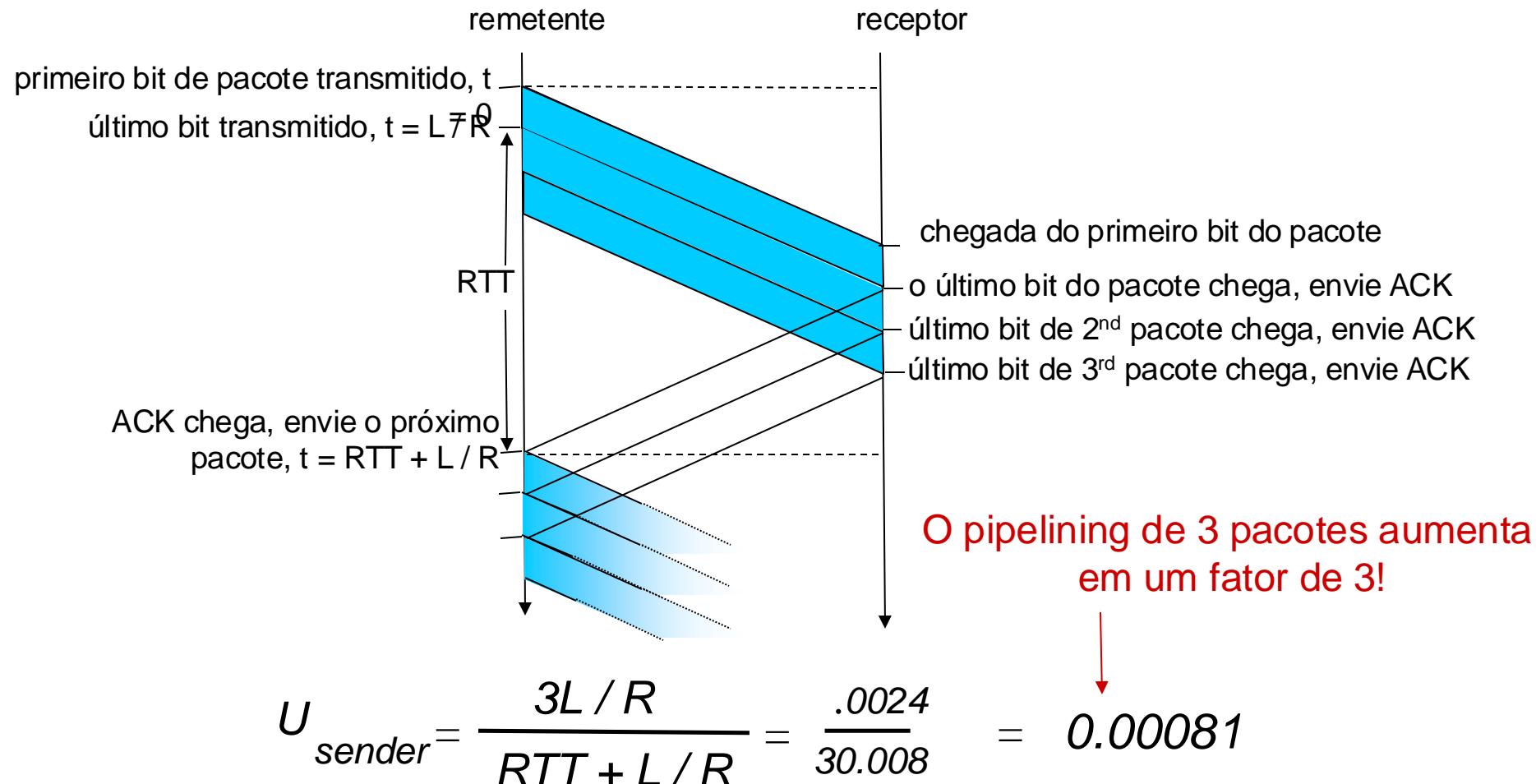
**pipelining:** o remetente permite vários pacotes "em trânsito", que ainda não foram reconhecidos

- o intervalo de números de sequência deve ser aumentado
- buffering no remetente e/ou no receptor



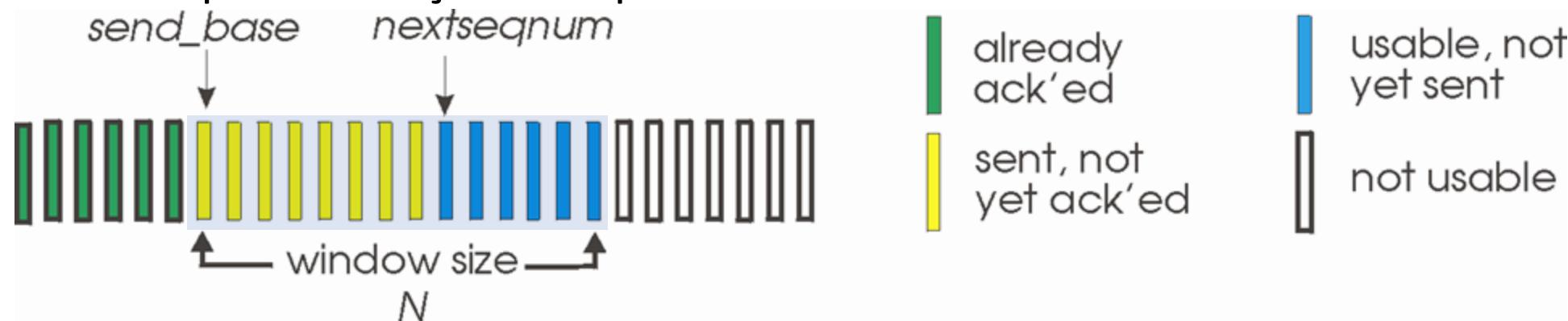
(a) a stop-and-wait protocol in operation

# Pipelining: maior utilização



# Go-Back-N: remetente

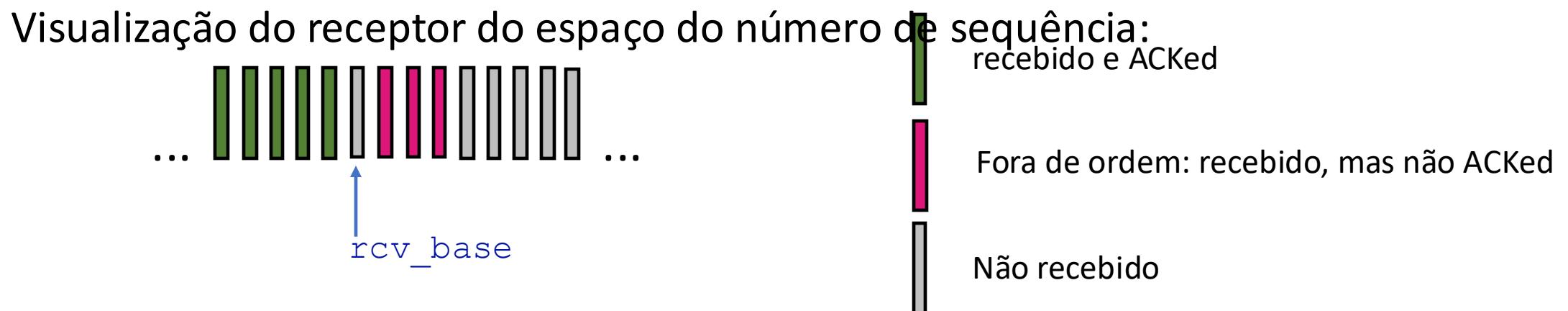
- remetente: "janela" de até N pkts consecutivos transmitidos, mas não enviados (unACKed)
  - k-bit seq # no cabeçalho do pkt



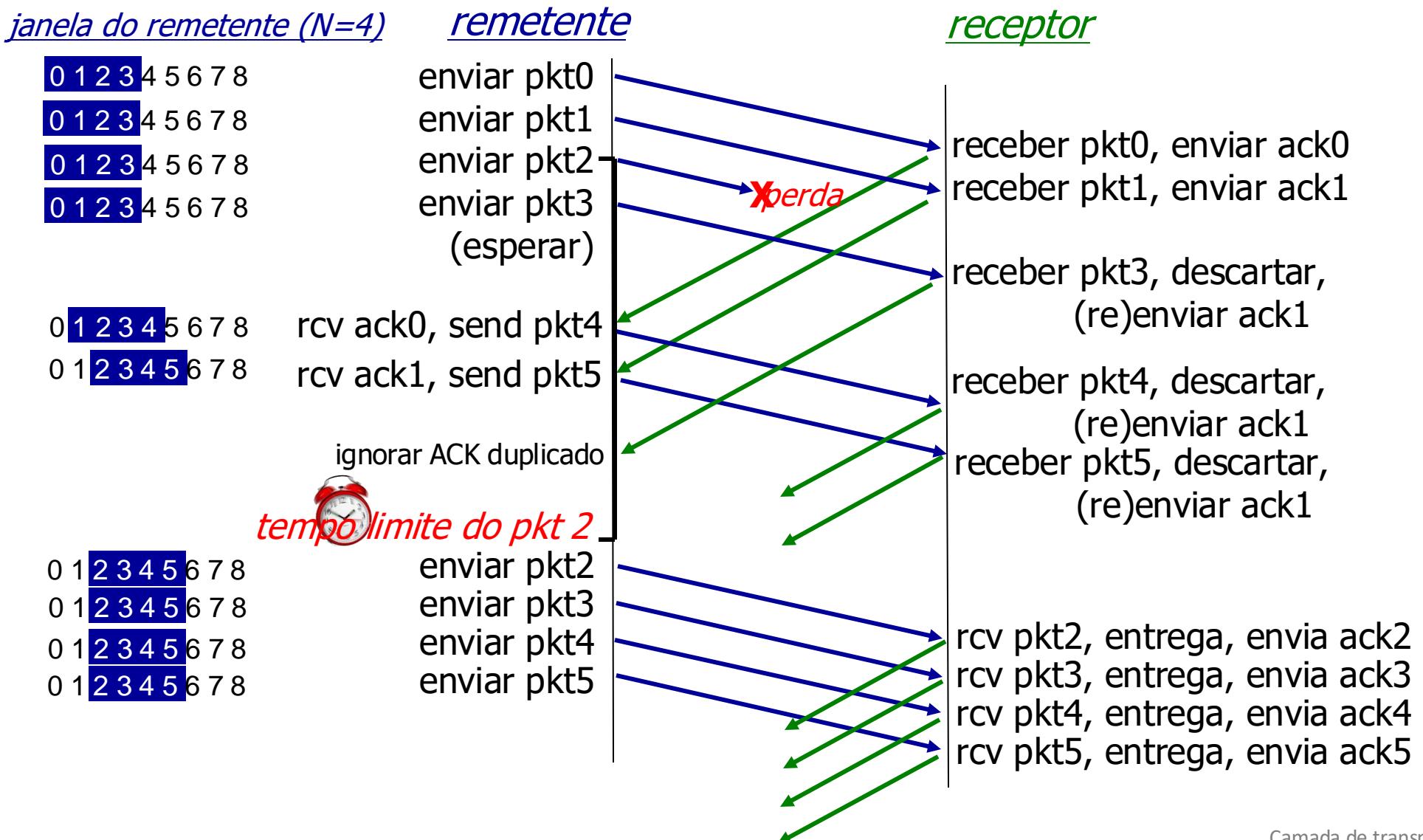
- *ACK cumulativo*:  $ACK(n)$ : ACKs todos os pacotes até a seq. n, inclusive
  - ao receber  $ACK(n)$ : move a janela para frente para começar em  $n+1$
- temporizador para o pacote mais antigo em trânsito
- $timeout(n)$ : retransmite o pacote n e todos os pacotes de seq. superior na janela

# Go-Back-N: receptor

- ACK-only: sempre envia ACK para o pacote recebido corretamente até o momento, com o número de sequência mais alto *na ordem*
  - pode gerar ACKs duplicados
  - só precisa se lembrar de `rcv_base`
- no recebimento de um pacote fora de ordem:
  - pode descartar (não armazenar em buffer) ou armazenar em buffer: uma decisão de implementação
  - reenviar o pkt com a seq. mais alta na ordem



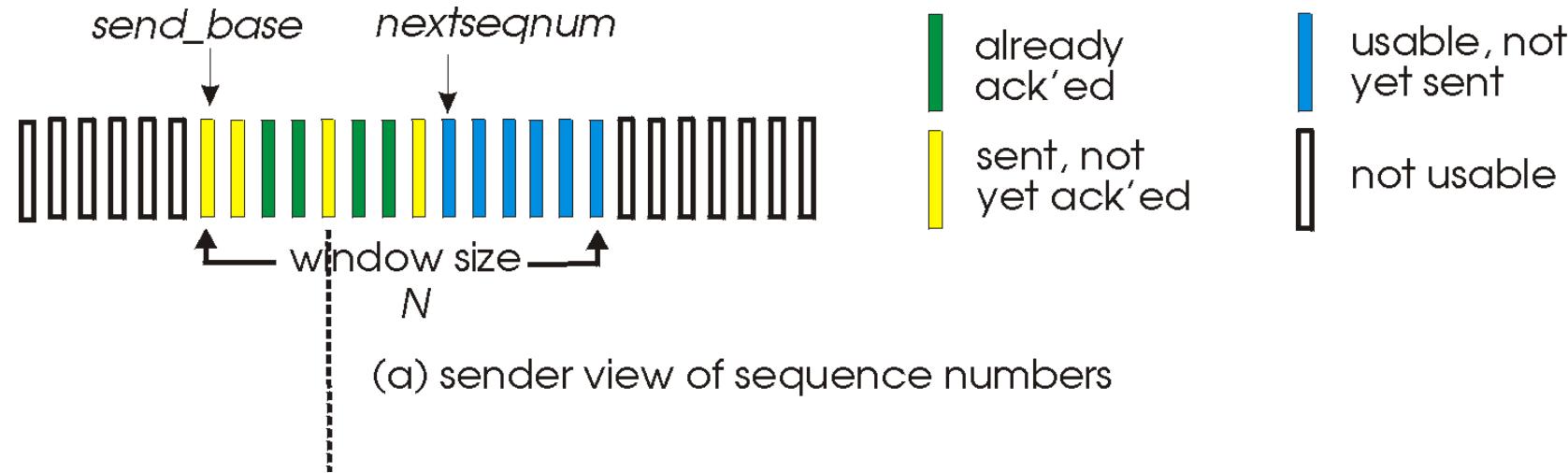
# Go-Back-N em ação



# Repetição seletiva: a abordagem

- *pipelining*: vários pacotes em voo
- *o receptor ACKs individualmente* todos os pacotes recebidos corretamente
  - armazena os pacotes em buffer, conforme necessário, para que sejam entregues em ordem à camada superior
- remetente:
  - mantém (conceitualmente) um cronômetro para cada pkt não aceito
    - timeout: retransmite um único pacote unACKed associado ao timeout
  - mantém (conceitualmente) uma "janela" sobre *N* seq #s consecutivos
    - limita os pacotes em pipeline e "em voo" a estarem dentro dessa janela

# Repetição seletiva: janelas do emissor e do receptor



# Repetição seletiva: emissor e receptor

remetente

dados acima:

- Se a próxima seq # disponível na janela, enviar o pacote

**timeout( $n$ ):**

- reenviar o pacote  $n$ , reiniciar o cronômetro

**ACK( $n$ )** em [sendbase,sendbase+N-1]:

- marcar o pacote  $n$  como recebido
- se o n menor pacote não aceito, avance a base da janela para a próxima sequência não aceita

receptor

**pacote  $n$  em [rcvbase, rcvbase+N-1]**

- enviar ACK( $n$ )
- fora de ordem: buffer
- in-order: entrega (também entrega pacotes em buffer, em ordem), avança a janela para o próximo pacote ainda não recebido

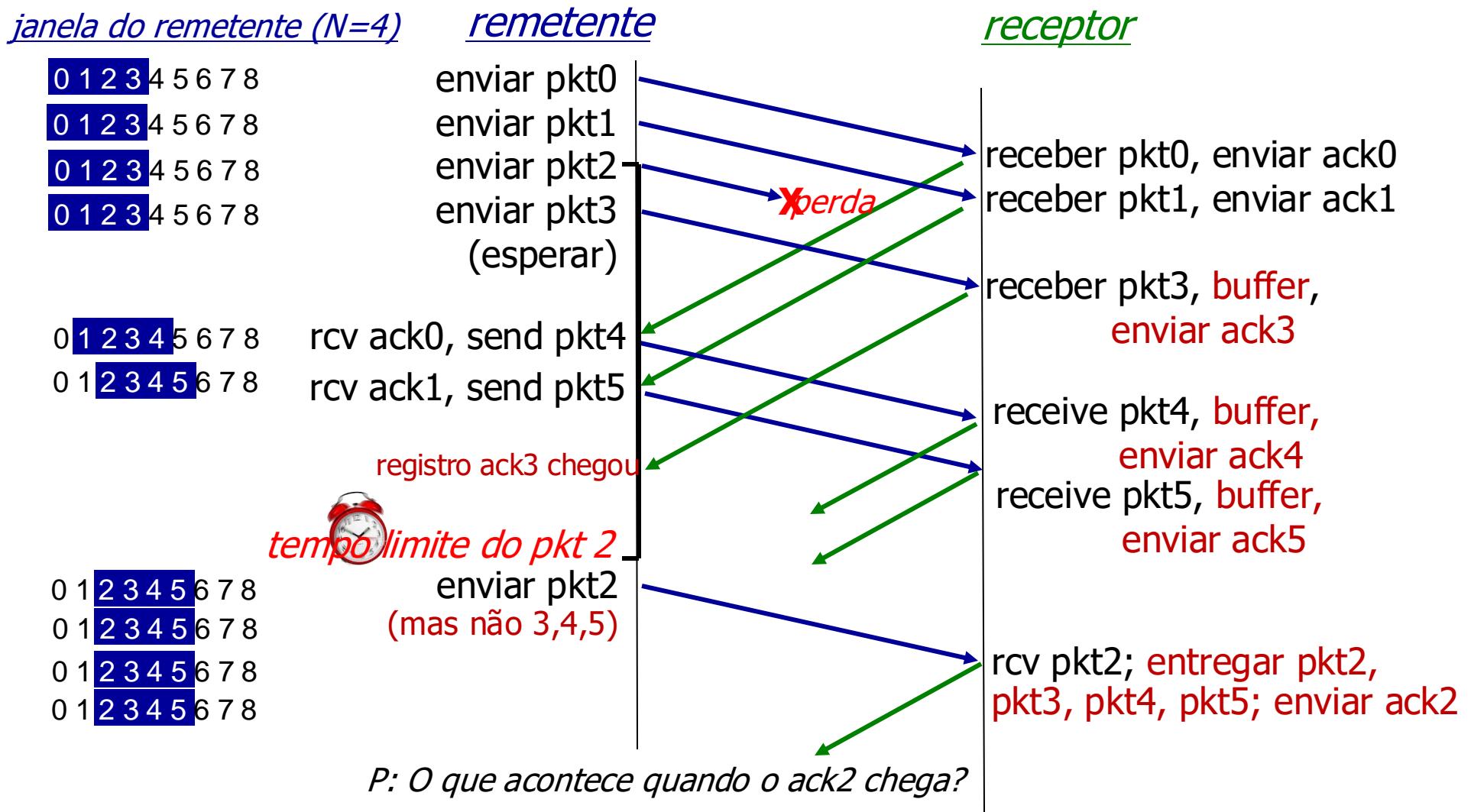
**pacote  $n$  em [rcvbase-N,rcvbase-1]**

- ACK( $n$ )

**caso contrário:**

- ignorar

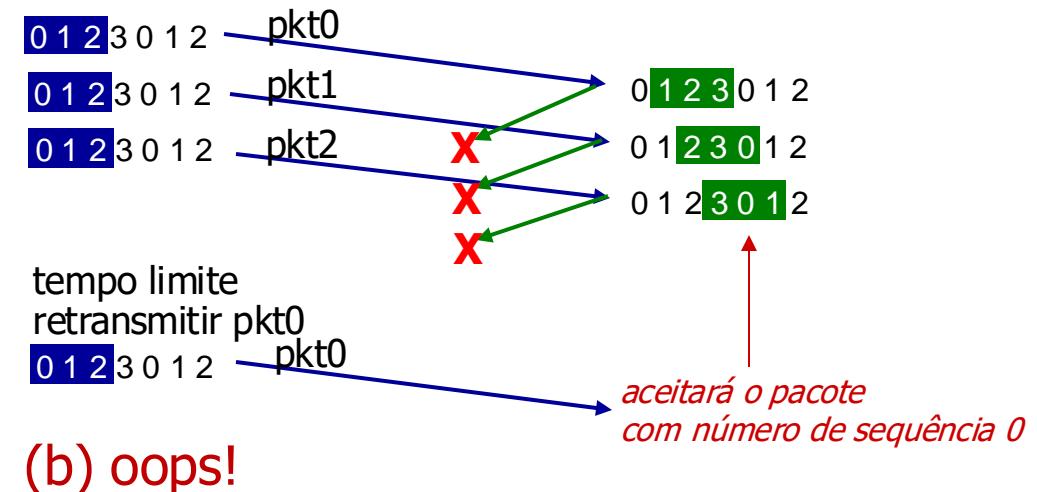
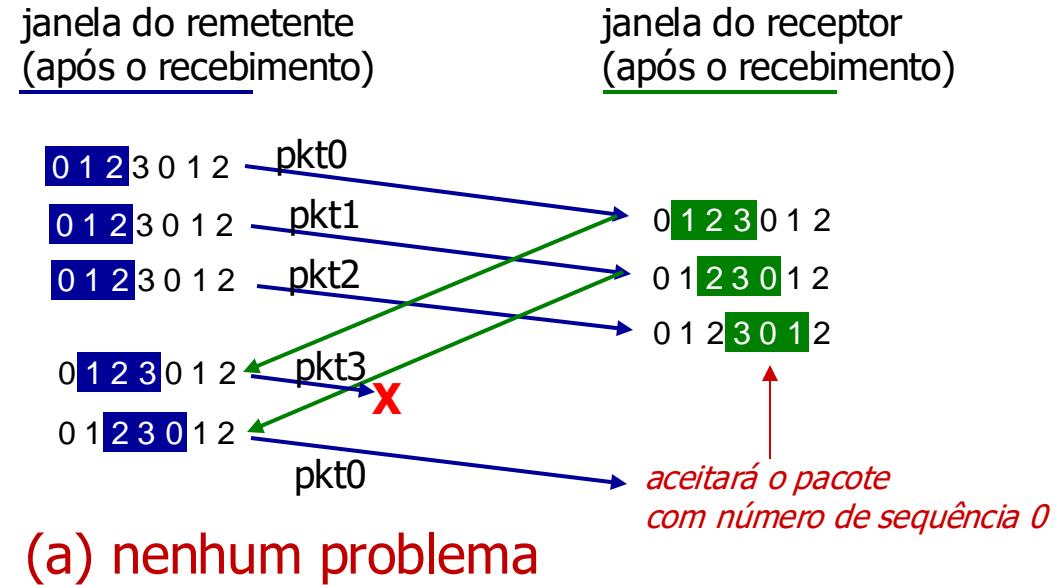
# Repetição seletiva em ação



# Repetição seletiva: um dilema!

exemplo:

- seq #s: 0, 1, 2, 3 (contagem de base 4)
- tamanho da janela=3



# Repetição seletiva: um dilema!

exemplo:

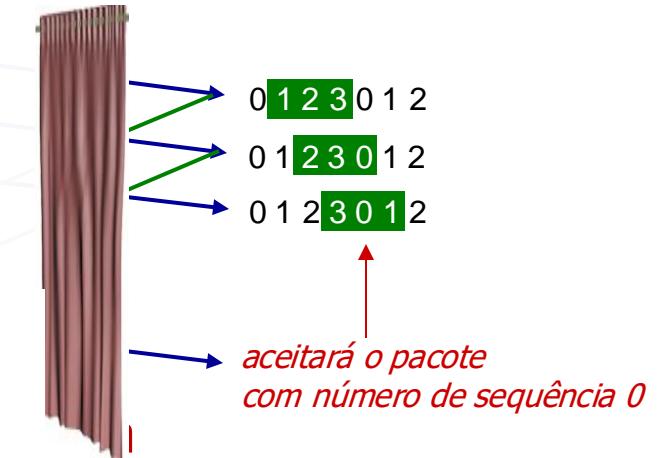
- seq #s: 0, 1, 2, 3 (contagem de base 4)
- tamanho da janela=3

P: Que relação é necessária entre o tamanho do número da sequência e o tamanho da janela para evitar o problema no cenário (b)?

janela do remetente  
(após o recebimento)

0 1 2 3 0 1 2 — pkt0  
0 1 2 3 0 1 2 — pkt1  
0 1 2 3 0 1 2 — pkt2  
0 1 2 3 0 1 2 < pkt3

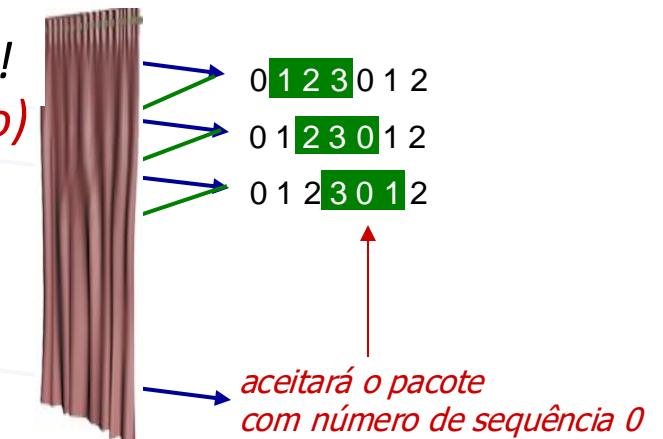
janela do receptor  
(após o recebimento)



- *o receptor não consegue ver o lado do remetente*
- *comportamento do receptor idêntico em ambos os casos!*
- *algo está (muito) errado!*

tempo limite  
retransmitir pkt0  
0 1 2 3 0 1 2 — pkt0

(b) oops!



# Capítulo 3: roteiro

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

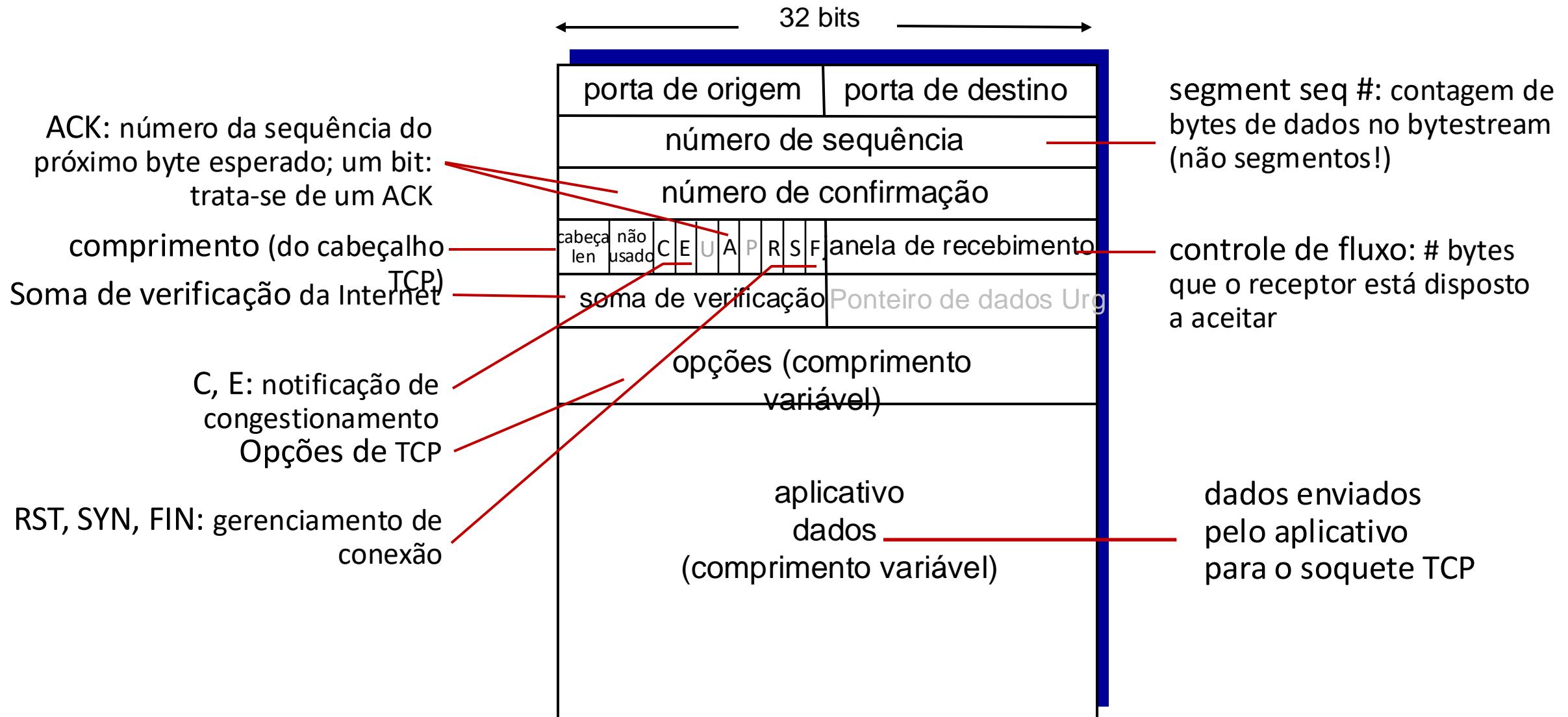


# TCP: visão geral

RFCs: 793, 1122, 2018, 5681, 7323

- ponto a ponto:
  - um remetente, um receptor
- *vapor de bytes* confiável e em ordem:
  - sem "limites de mensagem"
- dados full duplex:
  - fluxo de dados bidirecional na mesma conexão
  - MSS: tamanho máximo do segmento
- ACKs cumulativos
- pipelining:
  - Tamanho da janela do conjunto de controle de fluxo e congestionamento TCP
- orientado para a conexão:
  - handshaking (troca de mensagens de controle) inicializa o estado do emissor e do receptor antes da troca de dados
- controle de fluxo:
  - o remetente não sobrecarregará o receptor

# Estrutura do segmento TCP



# Números de sequência TCP, ACKs

## Números de sequência:

- "Número" do fluxo de bytes do primeiro byte nos dados do segmento

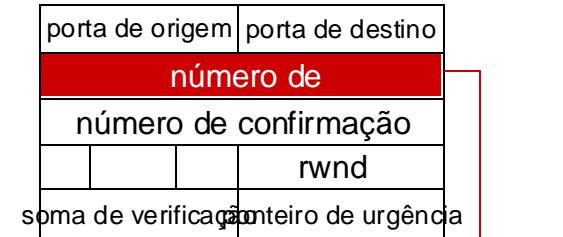
## Agradecimentos:

- seq # do próximo byte esperado do outro lado
- ACK cumulativo

P: como o receptor lida com segmentos fora de ordem

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

segmento de saída do remetente



tamanho da janela  
 $N$

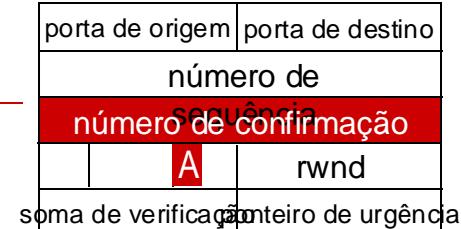


enviado  
ACKed

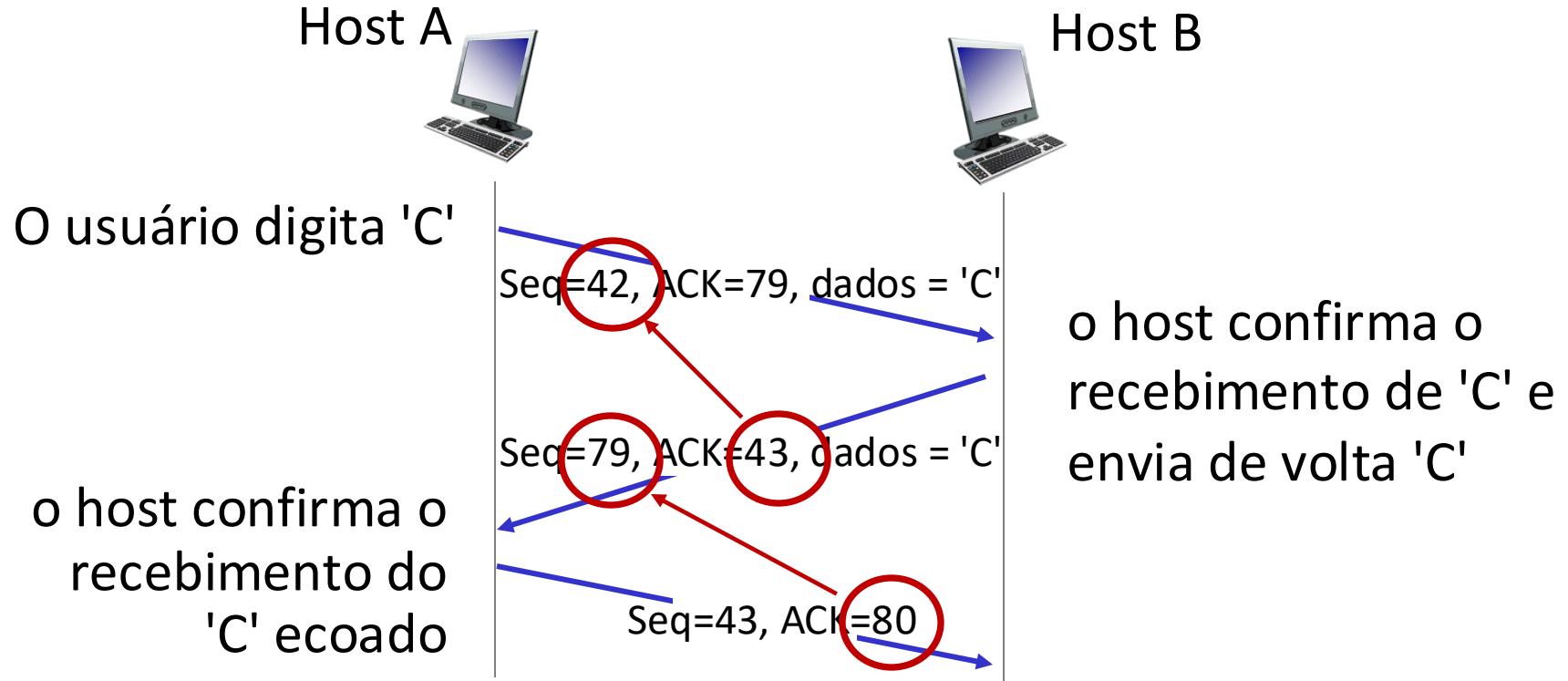
enviado,  
ainda não  
ACKed  
("em voo")

utilizável  
mas não utilizáv  
ainda el  
enviado

segmento de saída do receptor



# Números de sequência TCP, ACKs



cenário simples de telnet

# Tempo de ida e volta do TCP, tempo limite

P: Como definir o valor de tempo limite do TCP?

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

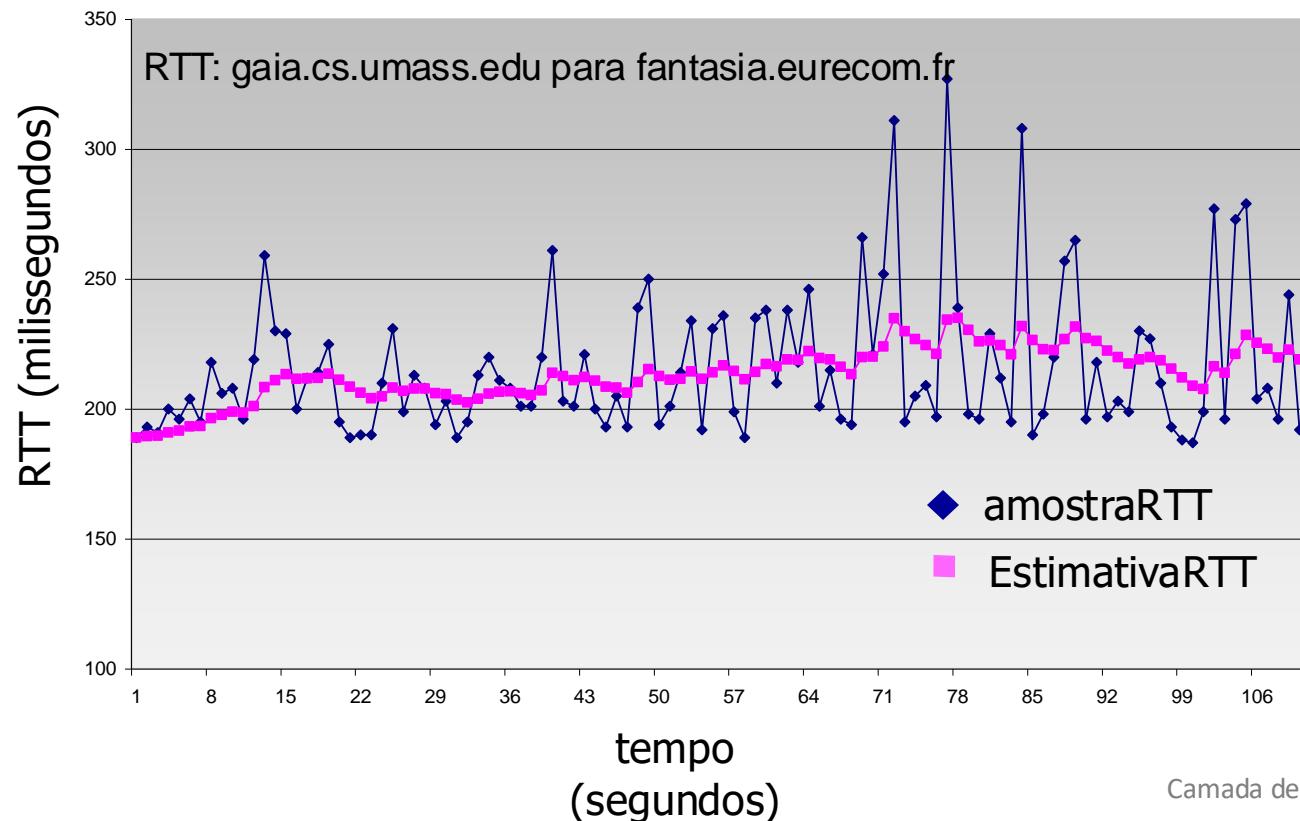
P: Como estimar o RTT?

- *SampleRTT*: tempo medido da transmissão do segmento até o recebimento do ACK
  - ignorar retransmissões
- O *SampleRTT* varia, queremos um RTT estimado mais "suave"
  - média de várias medições *recentes*, não apenas do *SampleRTT* atual

# Tempo de ida e volta do TCP, tempo limite

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

- média móvel ponderada exponencial (EWMA)
- a influência da amostra anterior diminui de forma exponencialmente rápida
- Valor típico:  $\alpha = 0,125$



# Tempo de ida e volta do TCP, tempo limite

- intervalo de tempo limite: **EstimatedRTT** mais "margem de segurança"
  - grande variação no **EstimatedRTT**: deseja uma margem de segurança maior

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



RTT estimado

"margem de segurança"

- DevRTT**: EWMA do desvio do **SampleRTT** em relação ao **EstimatedRTT**:

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

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

\* Confira os exercícios interativos on-line para obter mais exemplos: [http://gaia.cs.umass.edu/kurose\\_ross/interactive/](http://gaia.cs.umass.edu/kurose_ross/interactive/)

# Remetente TCP (simplificado)

*evento: dados recebidos do aplicativo*

- criar segmento com seq #
- seq # é o número do fluxo de bytes do primeiro byte de dados no segmento
- iniciar o cronômetro se ainda não estiver em execução
  - pense no cronômetro como o do segmento mais antigo nãoACKed
  - intervalo de expiração:  
TimeOutInterval

*evento: timeout*

- retransmitir o segmento que causou o tempo limite
- reiniciar o cronômetro

*evento: ACK recebido*

- se o ACK reconhecer segmentos anteriormente não reconhecidos
  - atualizar o que é conhecido como ACKed
- iniciar o cronômetro se ainda houver segmentos nãoACKed

# Receptor TCP: Geração de ACK [RFC 5681]

*Evento no receptor*

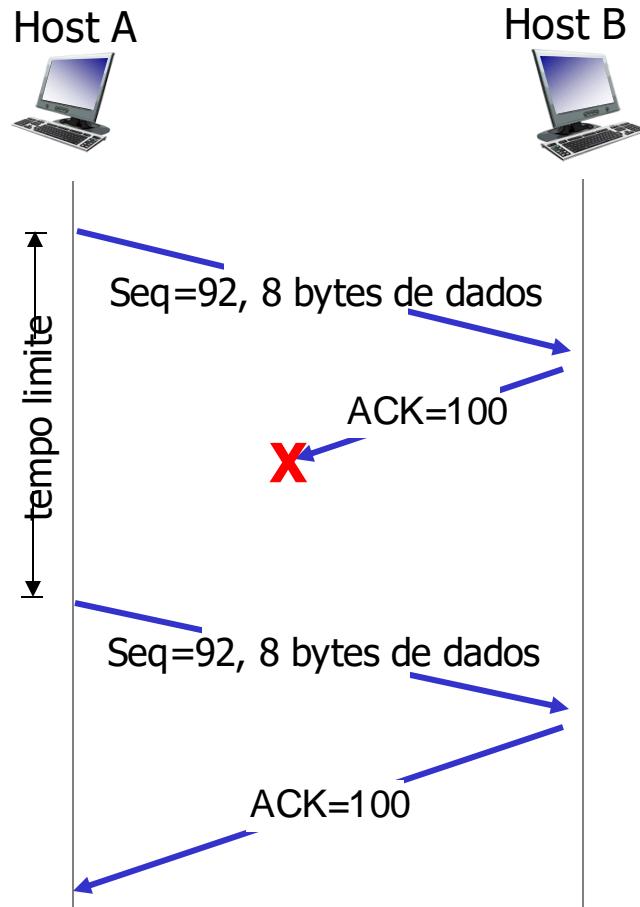
*Ação do receptor TCP*

chegada do segmento em ordem com | enviar imediatamente um único acumulado  
dem

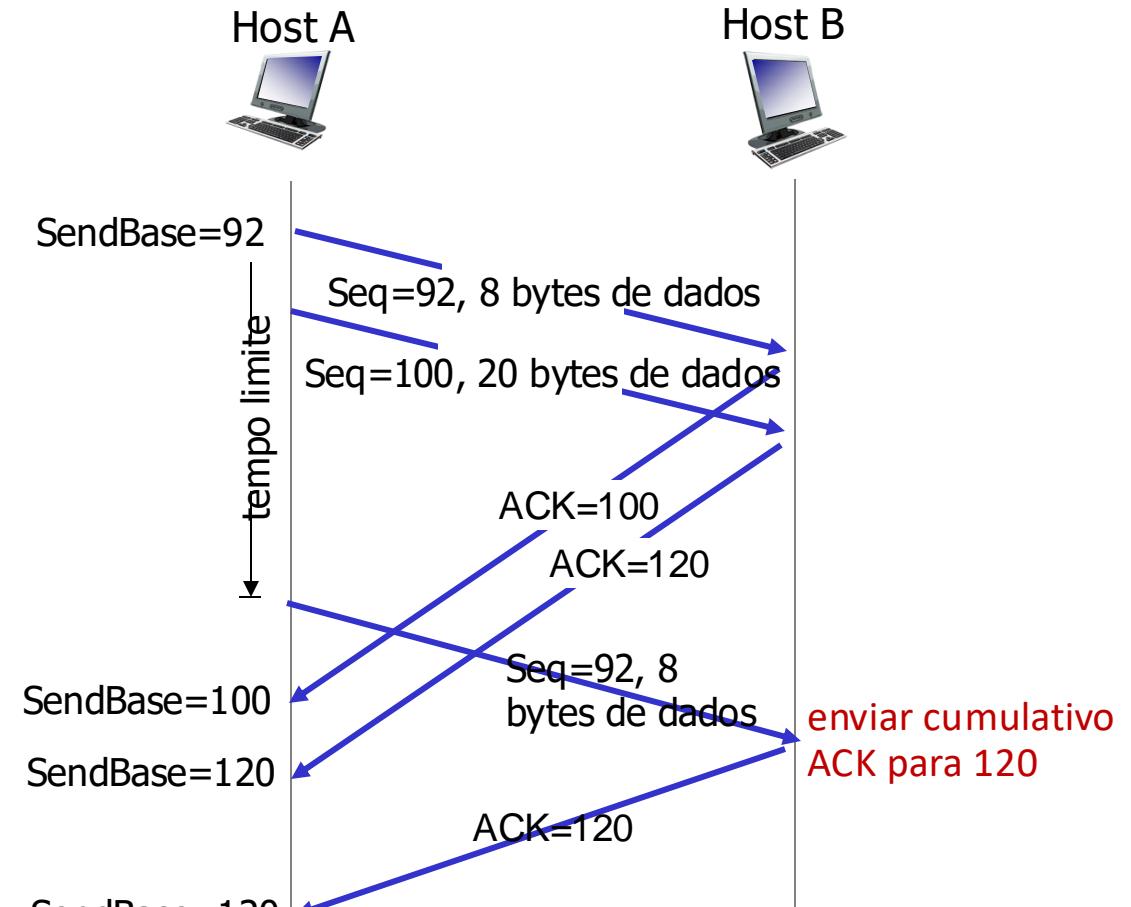
chegada de segmento fora de ordem | enviar imediatamente **um ACK duplicado**,  
)

... | enviar imediatamente, ... o menor ACK maior do intervalo

# TCP: cenários de retransmissão

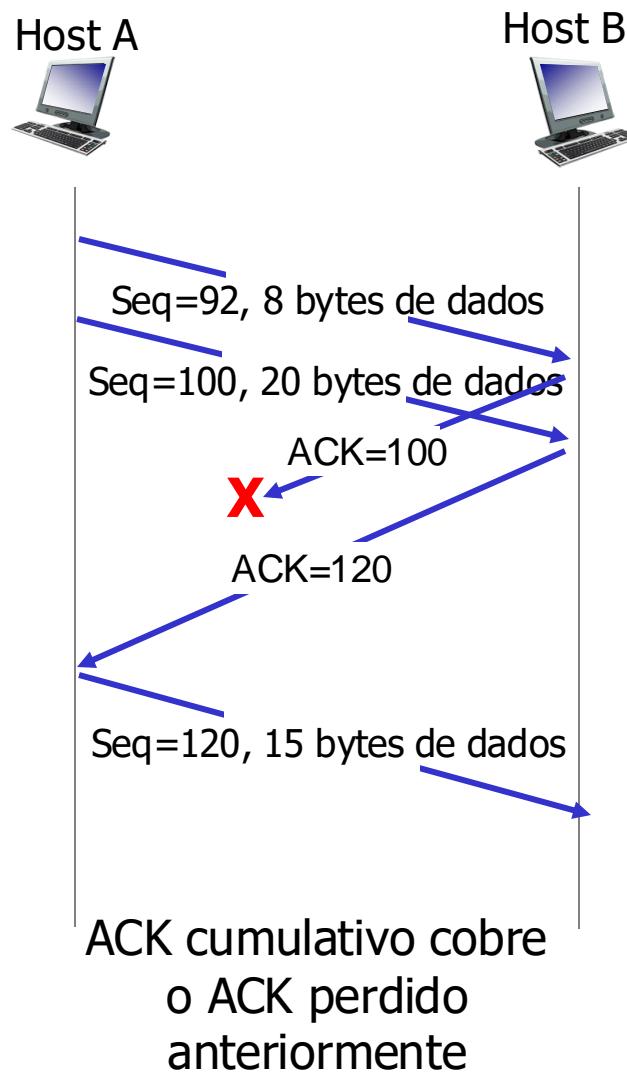


cenário de ACK perdido



tempo limite prematuro

# TCP: cenários de retransmissão



# Retransmissão rápida de TCP

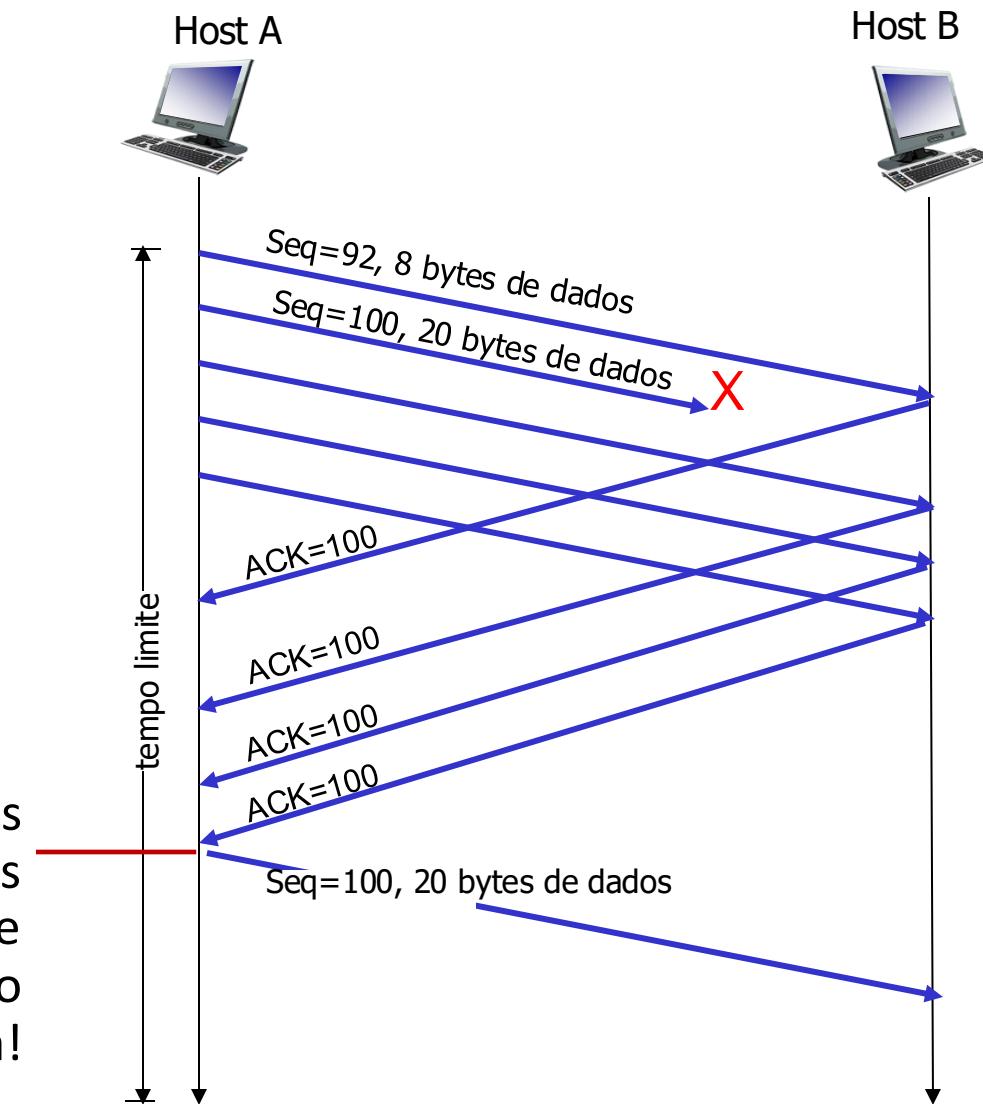
## *Retransmissão rápida de TCP*

Se o remetente receber 3 ACKs adicionais para os mesmos dados ("ACKs triplos duplicados"), reenvie o segmento sem ACK com o menor número de seq.

- é provável que o segmento unACKed tenha sido perdido, portanto, não espere pelo tempo limite



O recebimento de três ACKs duplicados indica 3 segmentos recebidos após um segmento ausente  
- é provável que haja um segmento perdido. Portanto, retransmita!



# Capítulo 3: roteiro

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

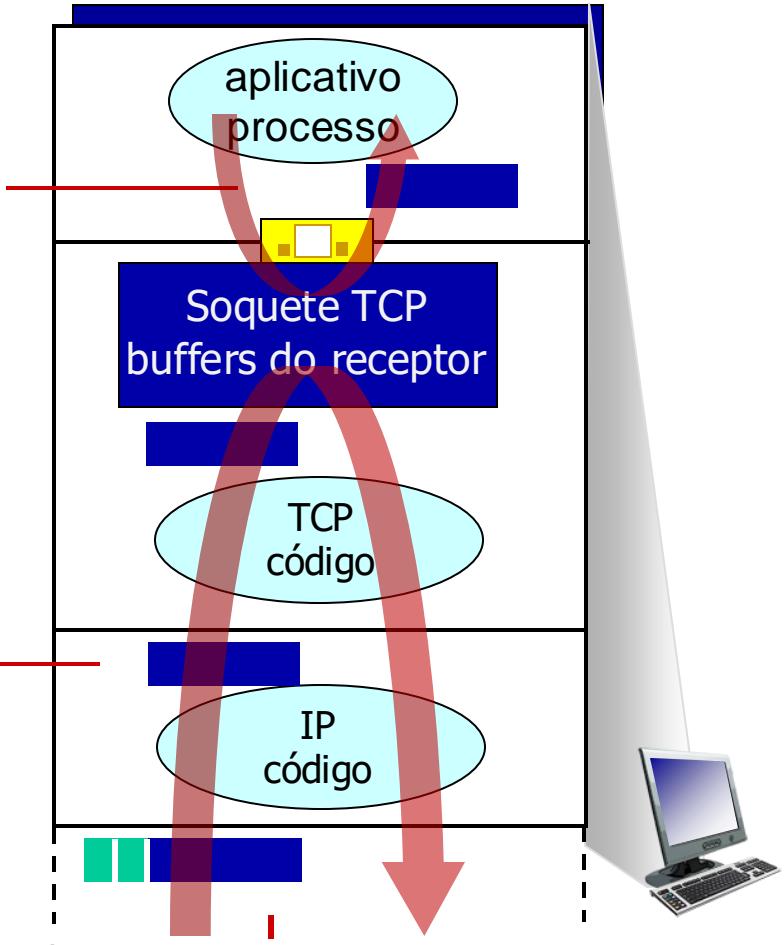


# Controle de fluxo TCP

**P:** O que acontece se a camada de rede fornecer dados mais rapidamente do que a camada de aplicativos remover os dados dos buffers de soquete?

Aplicativo que remove dados dos buffers de soquete TCP

Camada de rede que entrega carga útil de datagrama IP em buffers de soquete TCP



pilha de protocolos do receptor

# Controle de fluxo TCP

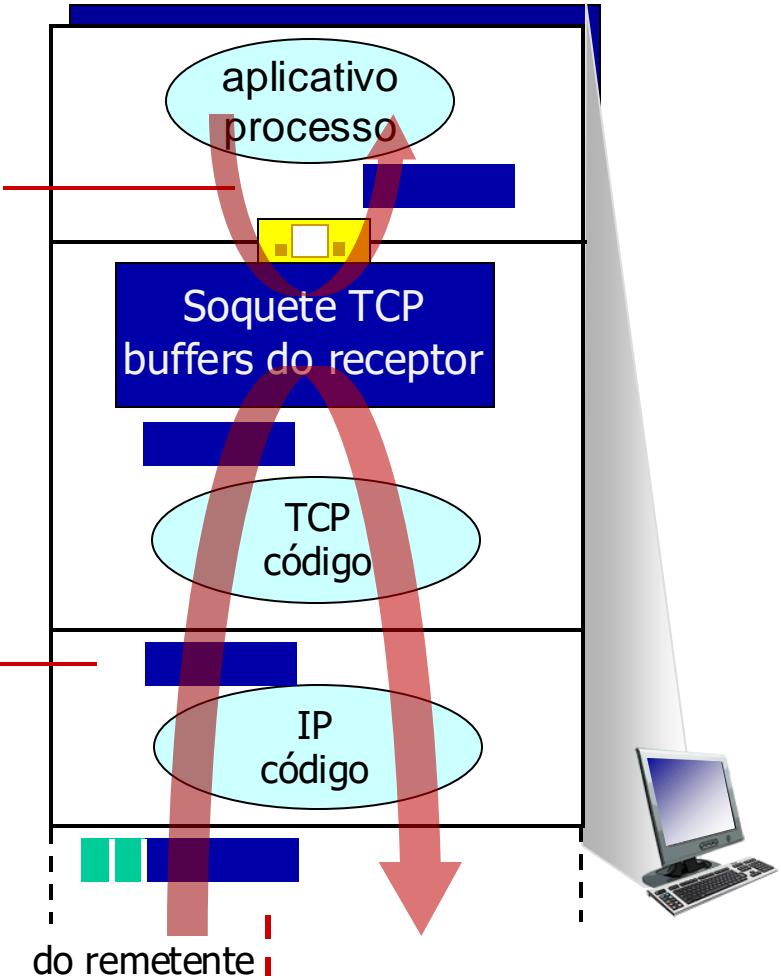
P: O que acontece se a camada de rede fornecer dados mais rapidamente do que a camada de aplicativos remover os dados dos buffers



Aplicativo que remove dados dos buffers de soquete TCP

Camada de rede que entrega carga útil de datagrama IP em buffers de soquete TCP

pilha de protocolos do receptor

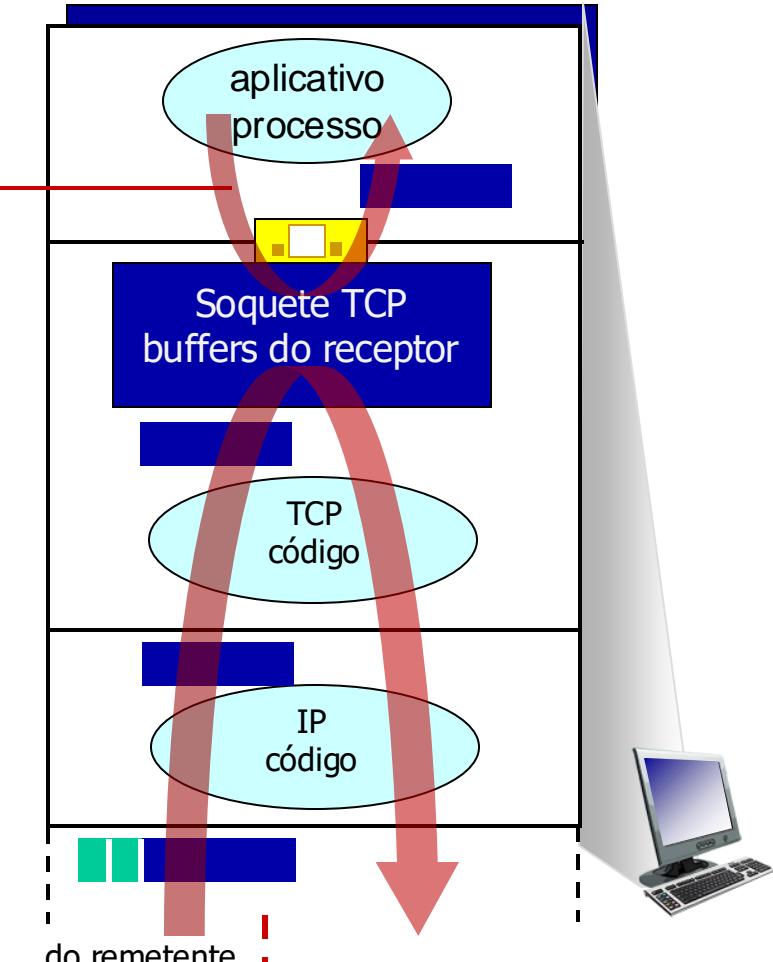


# Controle de fluxo TCP

**P:** O que acontece se a camada de rede fornecer dados mais rapidamente do que a camada de aplicativos remover os dados dos buffers de soquete?



Aplicativo que remove dados dos buffers de soquete TCP



pilha de protocolos do receptor

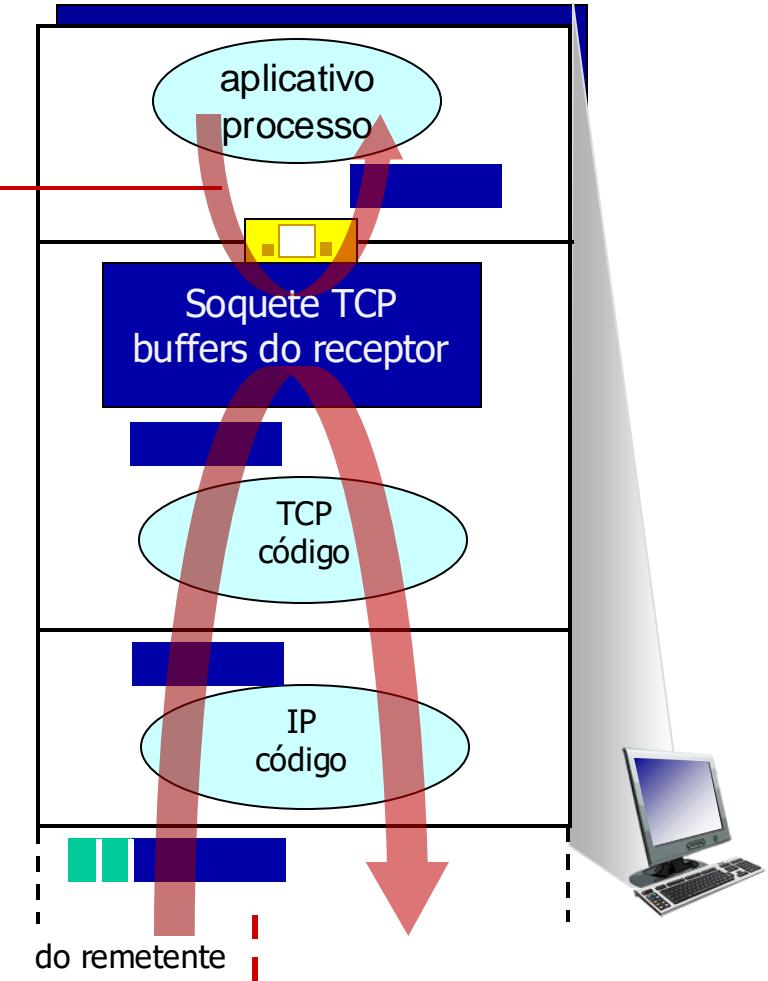
# Controle de fluxo TCP

**P:** O que acontece se a camada de rede fornecer dados mais rapidamente do que a camada de aplicativos remover os dados dos buffers de soquete?

## controle de fluxo

o receptor controla o remetente, para que o remetente não sobrecarregue o buffer do receptor transmitindo muito, muito rápido

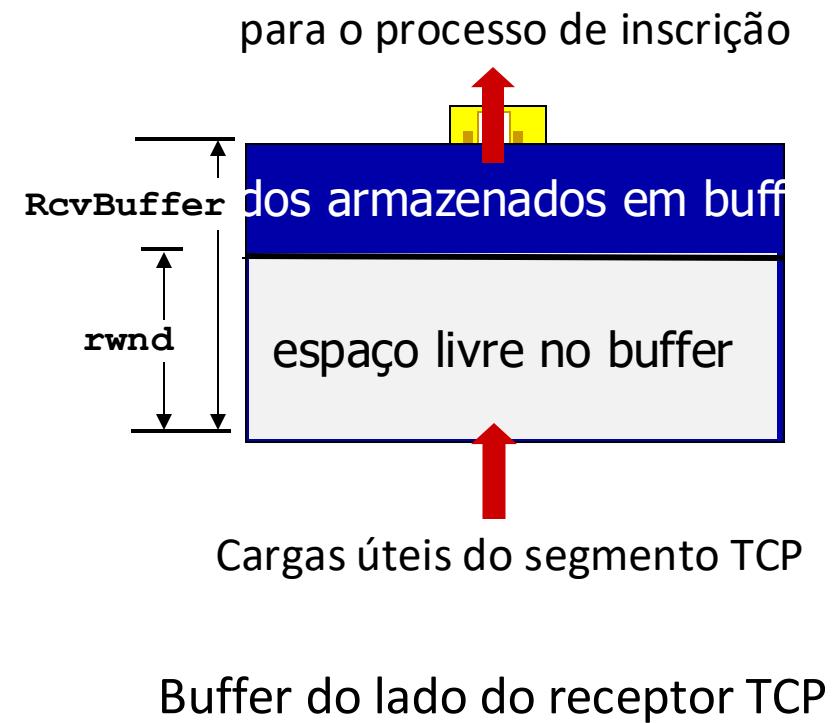
Aplicativo que remove dados dos buffers de soquete TCP



pilha de protocolos do receptor

# Controle de fluxo TCP

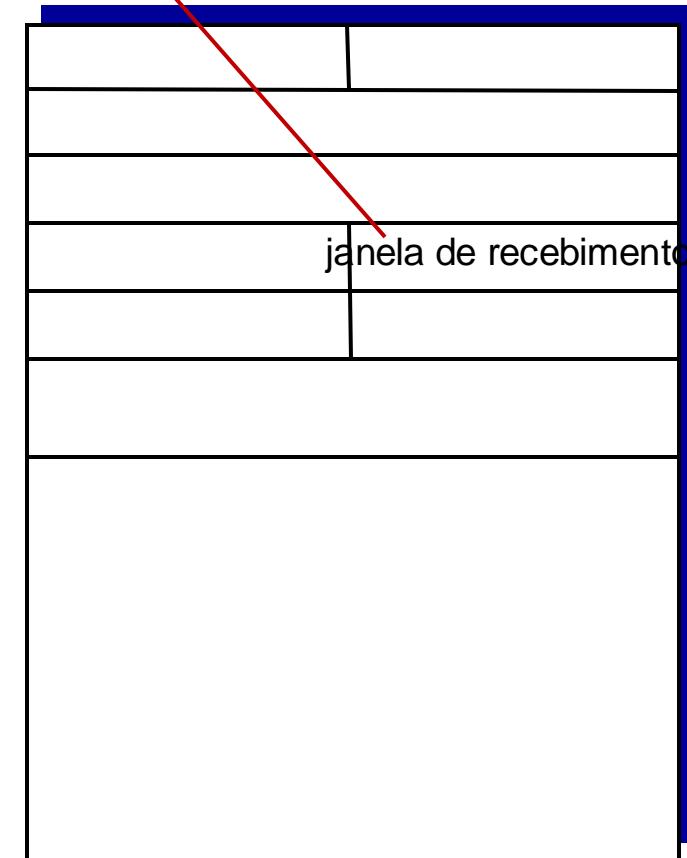
- O receptor TCP "anuncia" o espaço livre do buffer no campo **rwnd** do cabeçalho TCP
  - Tamanho do **RcvBuffer** definido por meio de opções de soquete (o padrão típico é 4096 bytes)
  - Muitos sistemas operacionais ajustam automaticamente o **RcvBuffer**
- o remetente limita a quantidade de dados não embalados ("em andamento") à **rwnd** recebida
- garante que o buffer de recepção não transbordará



# Controle de fluxo TCP

- O receptor TCP "anuncia" o espaço livre do buffer no campo **rwnd** do cabeçalho TCP
  - Tamanho do **RcvBuffer** definido por meio de opções de soquete (o padrão típico é 4096 bytes)
  - Muitos sistemas operacionais ajustam automaticamente o **RcvBuffer**
- o remetente limita a quantidade de dados não embalados ("em andamento") à **rwnd** recebida
- garante que o buffer de recepção não transbordará

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



Formato do segmento TCP

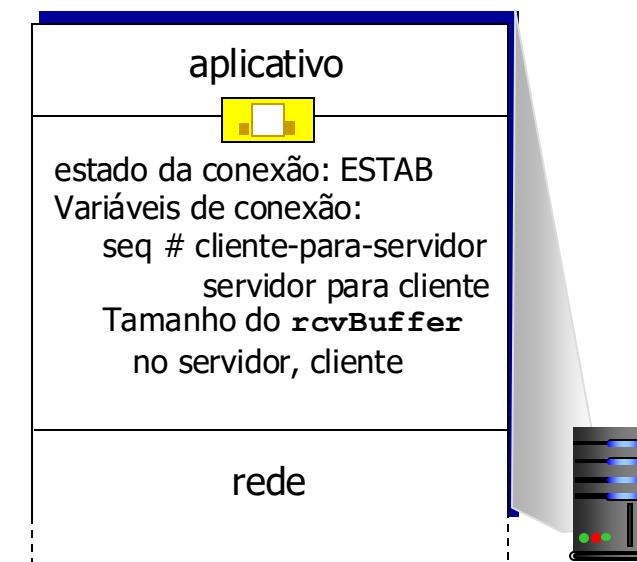
# Gerenciamento de conexão TCP

antes de trocar dados, o "handshake" do remetente/receptor:

- concordar em estabelecer conexão (cada um sabendo que o outro está disposto a estabelecer conexão)
- concordar com os parâmetros de conexão (por exemplo, números de seq. iniciais)



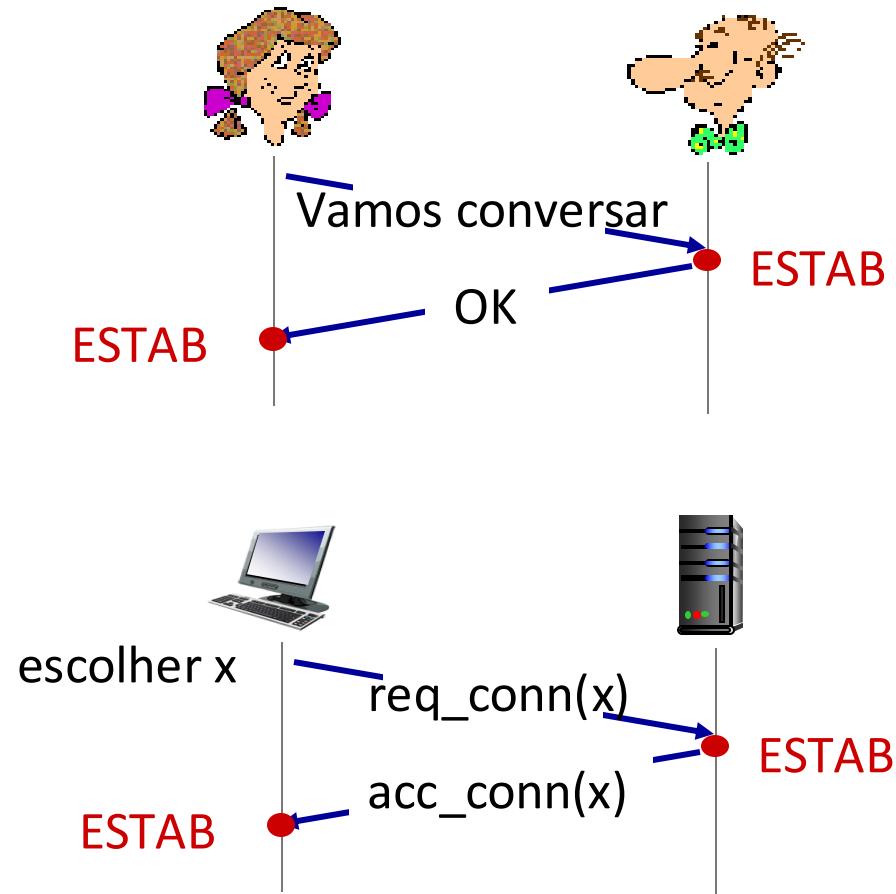
```
Socket clientSocket =  
    newSocket("hostname", "port number");
```



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

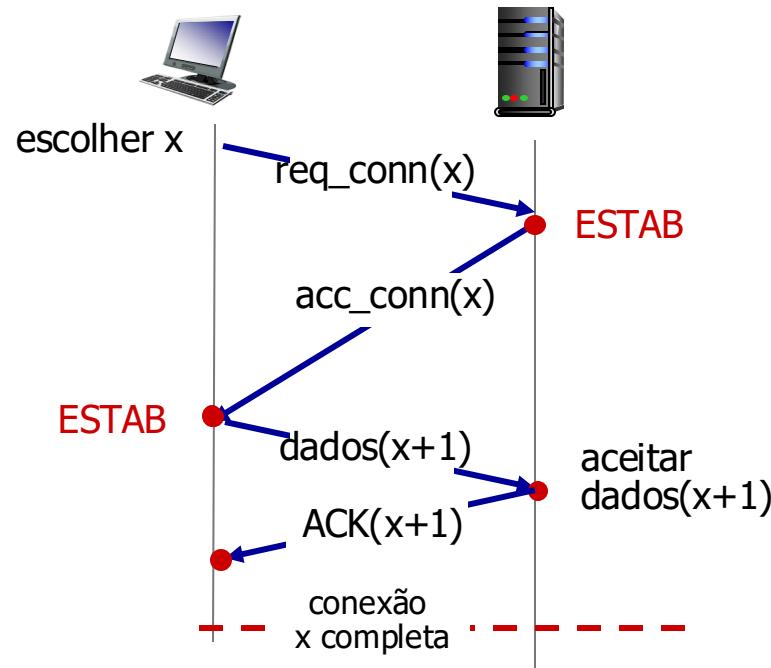
# Concordar em estabelecer uma conexão

Aperto de mão bidirecional:



- P: O handshake bidirecional sempre funcionará na rede?
- atrasos variáveis
  - mensagens retransmitidas (por exemplo,  $\text{req\_conn}(x)$ ) devido à perda de mensagens
  - reordenação de mensagens
  - não consegue "ver" o outro lado

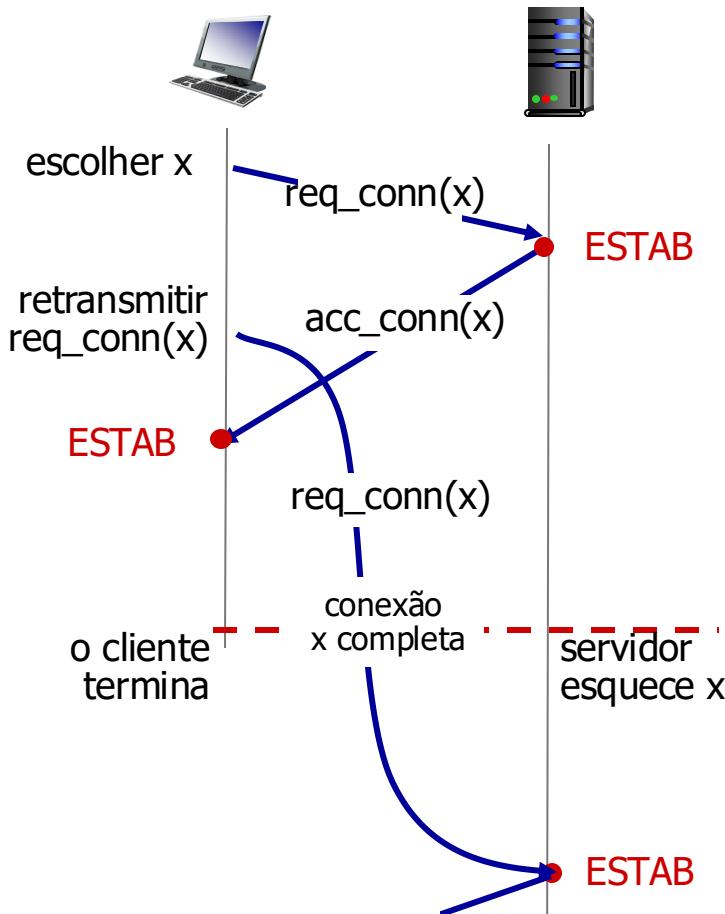
# Cenários de handshake bidirecional



Não tem problema!

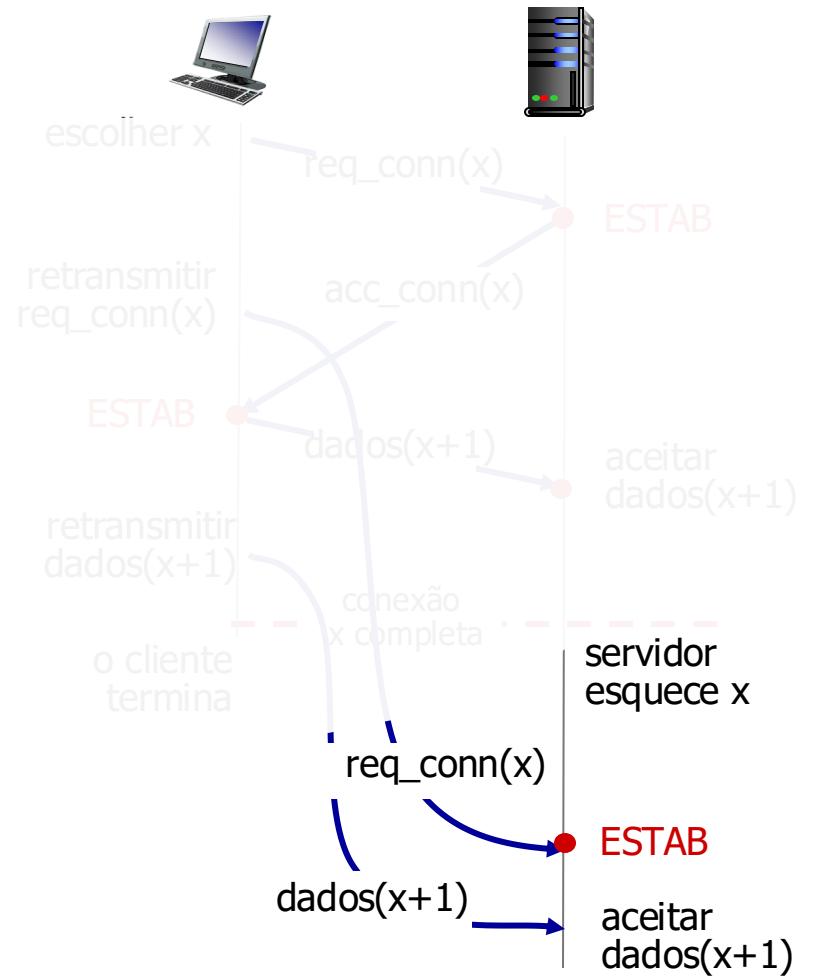


# Cenários de handshake bidirecional



Problema: conexão meio aberta! (sem cliente)

# Cenários de handshake bidirecional



Problema: dados  
duplicados  
aceito!

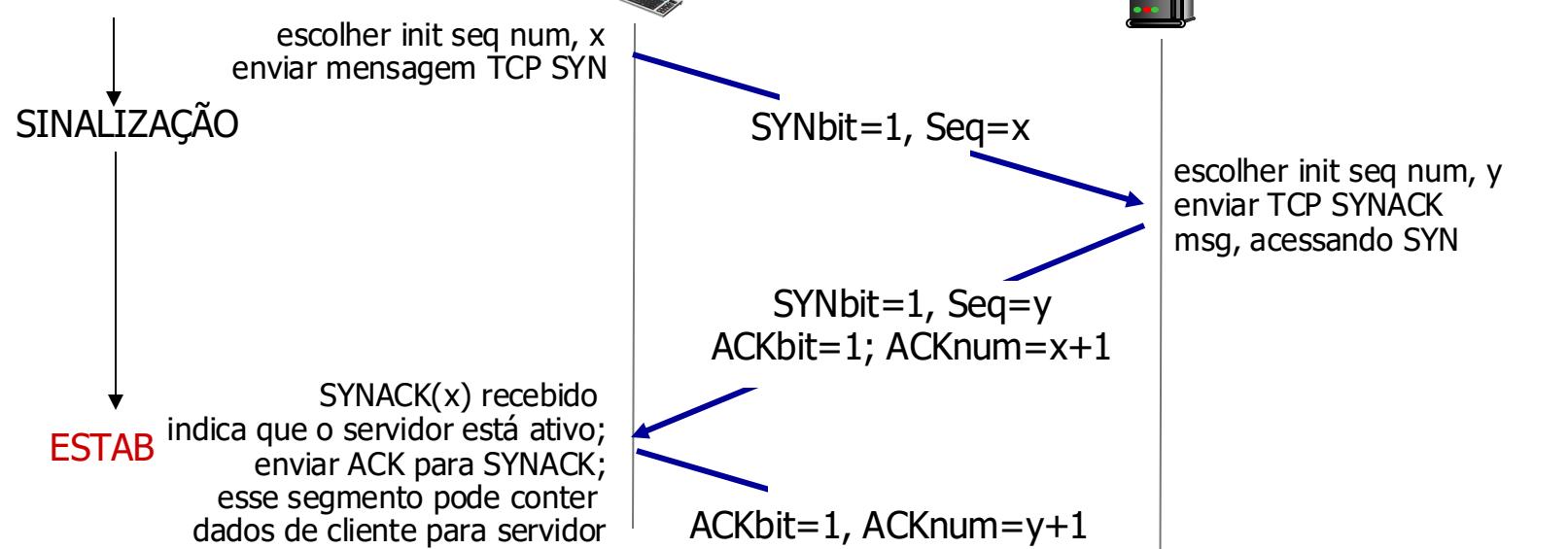
# Aperto de mão de 3 vias do TCP

## Estado do cliente

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

OUVIR

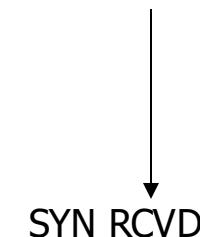
```
clientSocket.connect((serverName, serverPort))
```



## Estado do servidor

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

OUVIR



# Um protocolo de handshake humano de 3 vias



# Fechamento de uma conexão TCP

- cliente e servidor fecham cada um seu lado da conexão
  - enviar segmento TCP com bit FIN = 1
- responder ao FIN recebido com ACK
  - Ao receber o FIN, o ACK pode ser combinado com o próprio FIN
- trocas FIN simultâneas podem ser tratadas

# Capítulo 3: roteiro

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



# Princípios do controle de congestionamento

## Congestionamento:

- informalmente: "muitas fontes enviando muitos dados muito rapidamente para a *rede* suportar"
- manifestações:
  - longos atrasos (enfileiramento nos buffers do roteador)
  - perda de pacotes (estouro de buffer nos roteadores)
- diferente do controle de fluxo!
- um problema entre os 10 melhores!

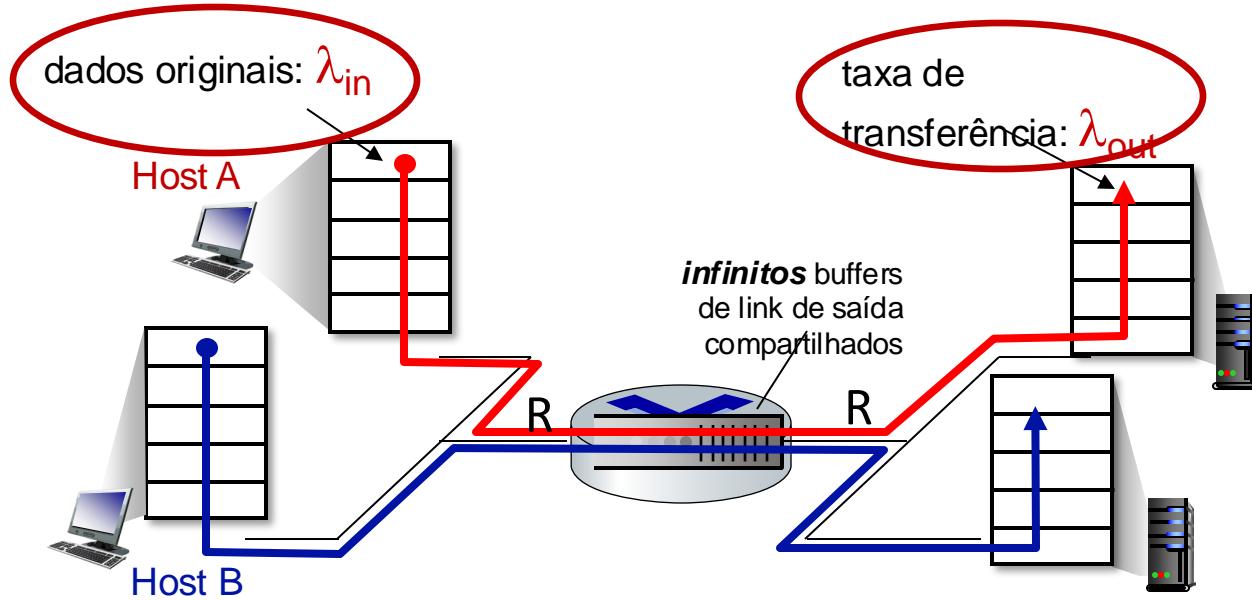


**controle de congestionamento**: muitos remetentes, envio muito rápido  
**controle de fluxo**: um remetente muito rápido para um receptor

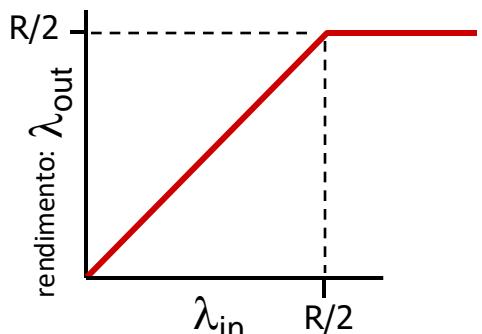
# Causas/custos do congestionamento: cenário 1

Cenário mais simples:

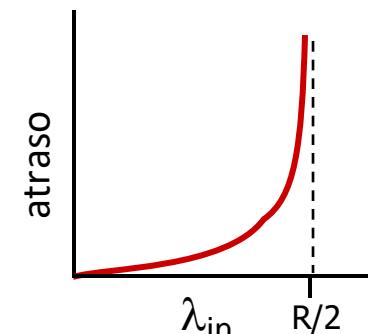
- um roteador, buffers infinitos
- capacidade do link de entrada e saída:  $R$
- dois fluxos
- não são necessárias retransmissões



P: O que acontece quando a taxa de chegada  $\lambda_{in}$  se aproxima de  $R/2$ ?



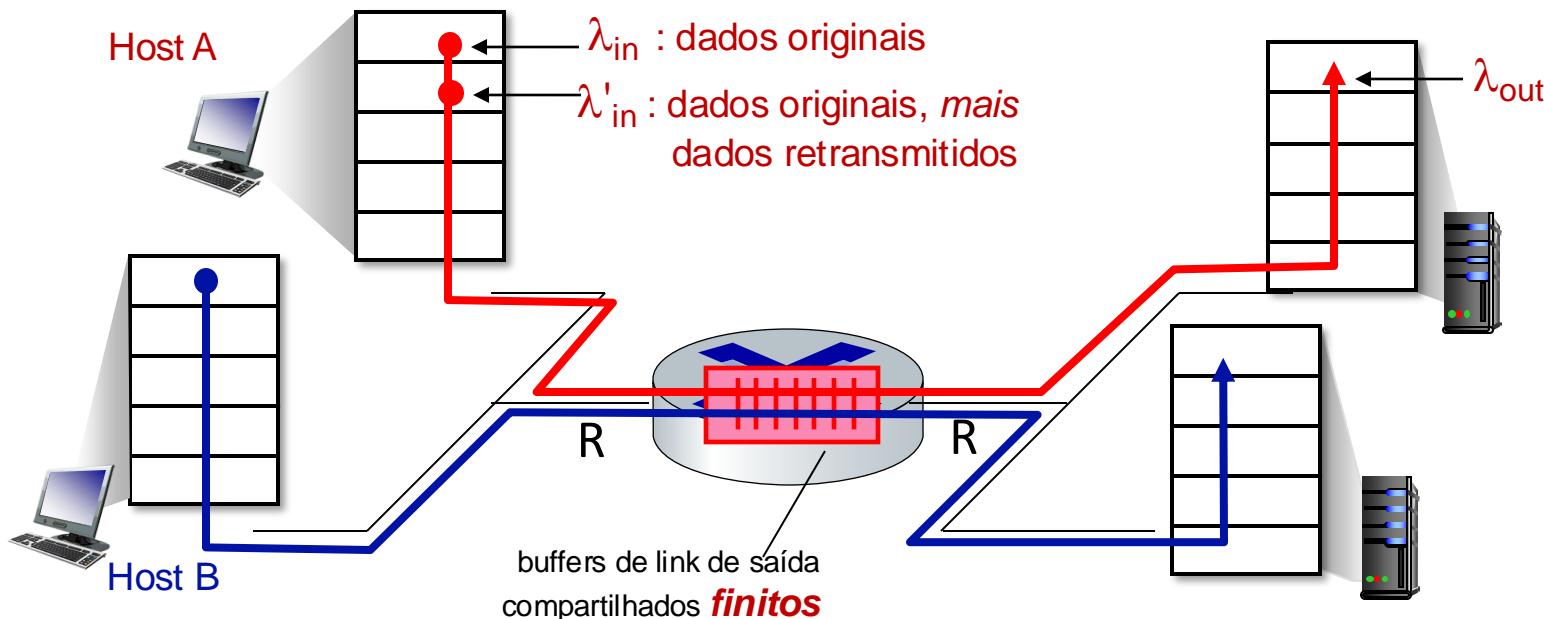
taxa de transferência máxima por conexão:  $R/2$



grandes atrasos à medida que a λinjia se aproxima da capacidade

# Causas/custos do congestionamento: cenário 2

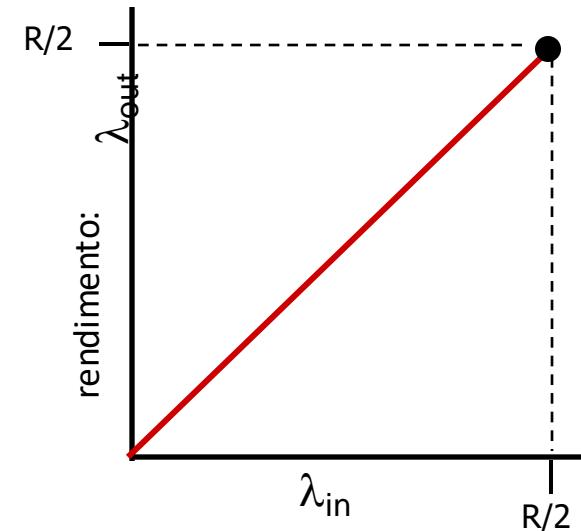
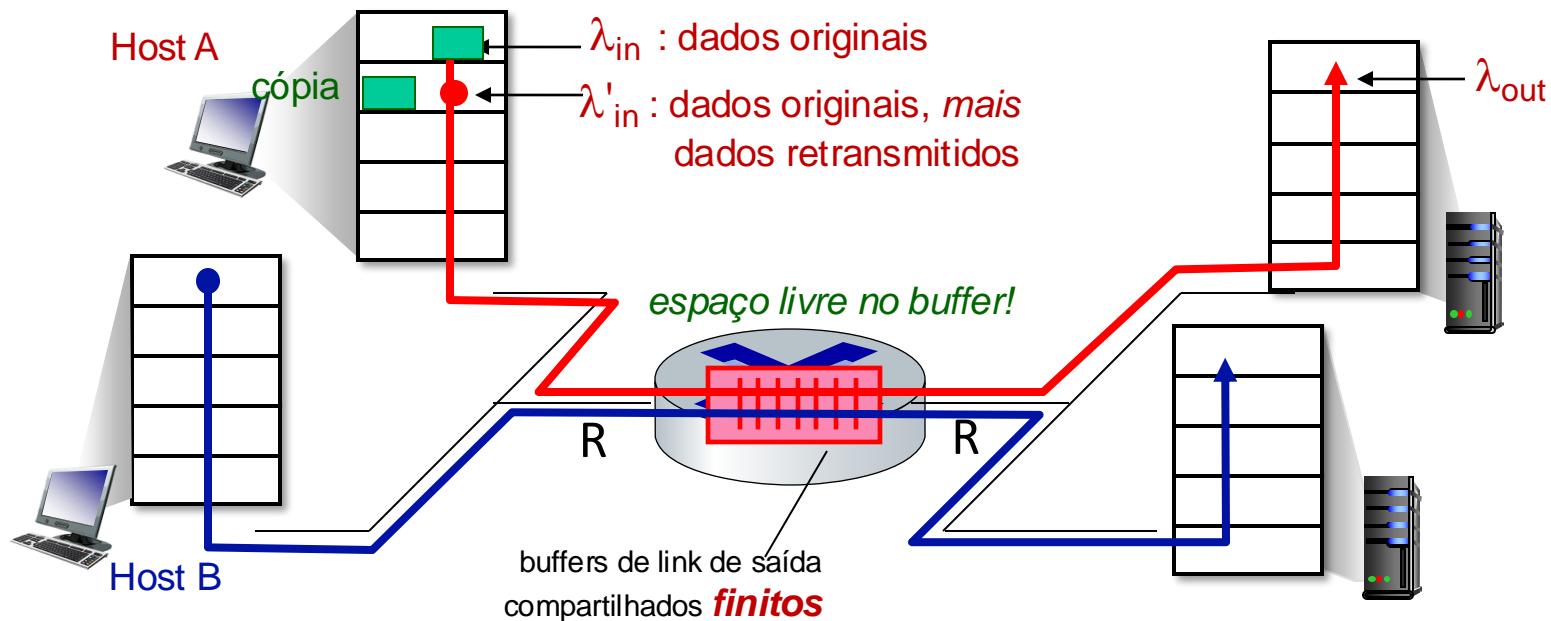
- um roteador, buffers *finitos*
- o remetente retransmite o pacote perdido e com tempo limite
  - entrada da camada de aplicativos = saída da camada de aplicativos:  $\lambda_{in} = \lambda_{out}$
  - a entrada da camada de transporte inclui *retransmissões*:  $\lambda'_{in} \geq \lambda_{in}$



# Causas/custos do congestionamento: cenário 2

Idealização: conhecimento perfeito

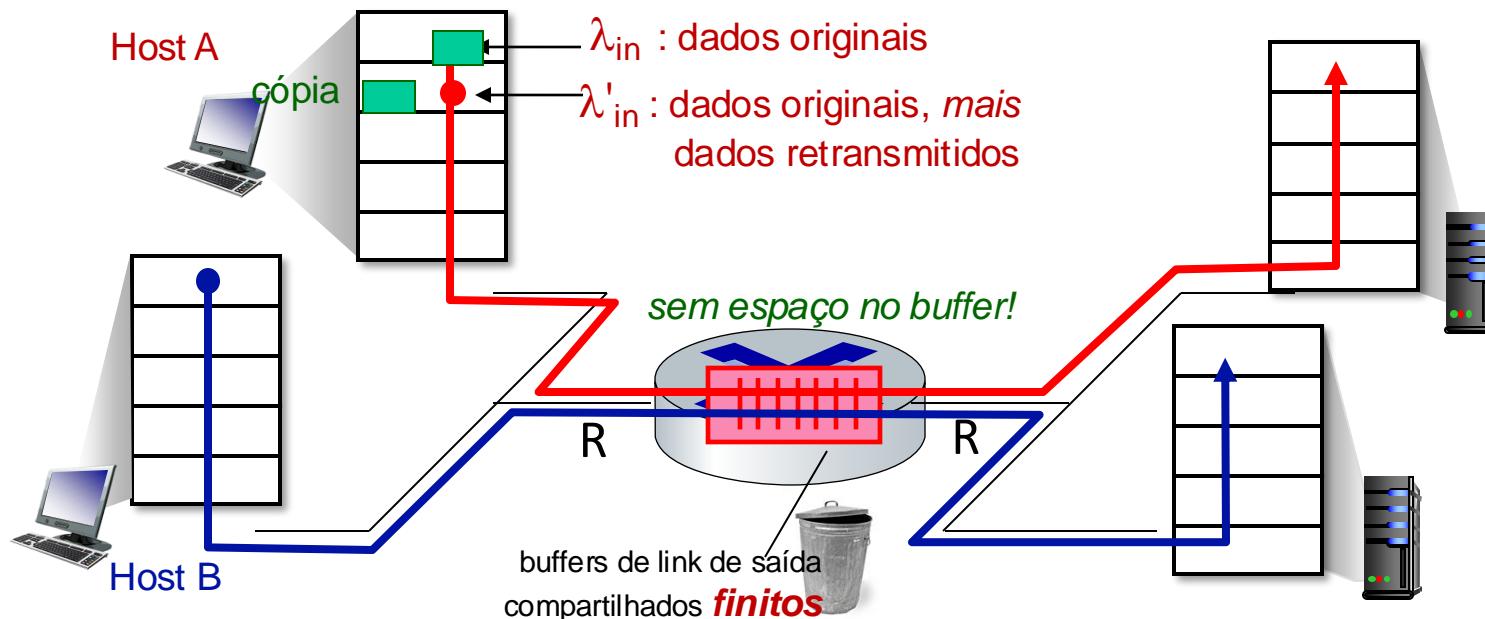
- o remetente envia somente quando os buffers do roteador estão disponíveis



# Causas/custos do congestionamento: cenário 2

Idealização: *algum* conhecimento perfeito

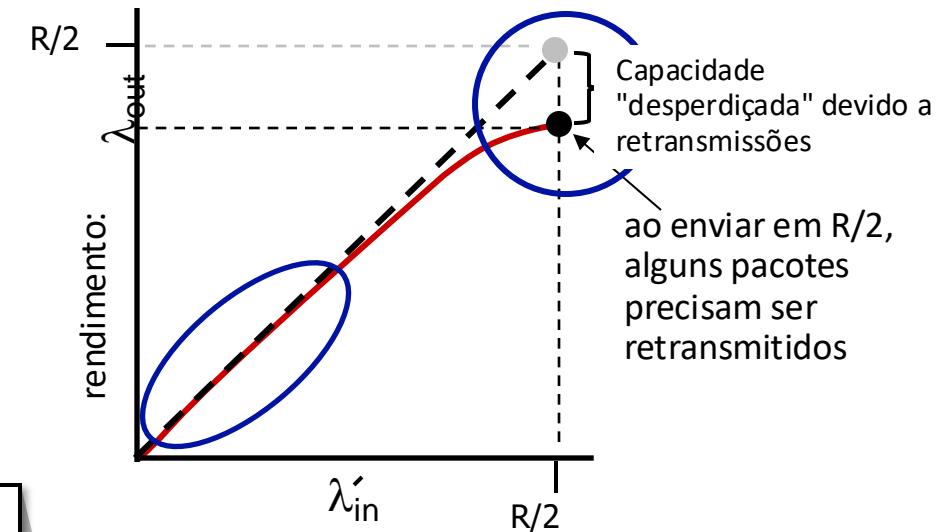
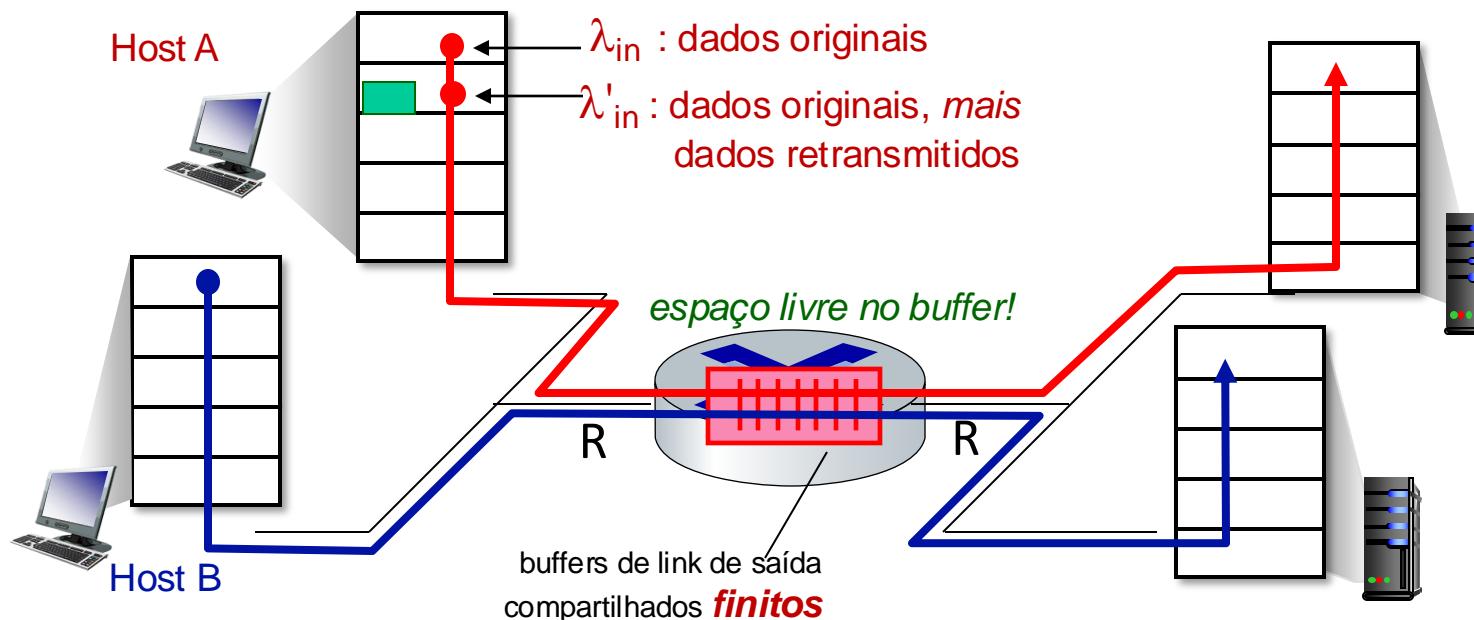
- os pacotes podem ser perdidos (descartados no roteador) devido a buffers cheios
- o remetente sabe quando o pacote foi descartado: só reenvia se o pacote *for sabidamente* perdido



# Causas/custos do congestionamento: cenário 2

Idealização: *algum* conhecimento perfeito

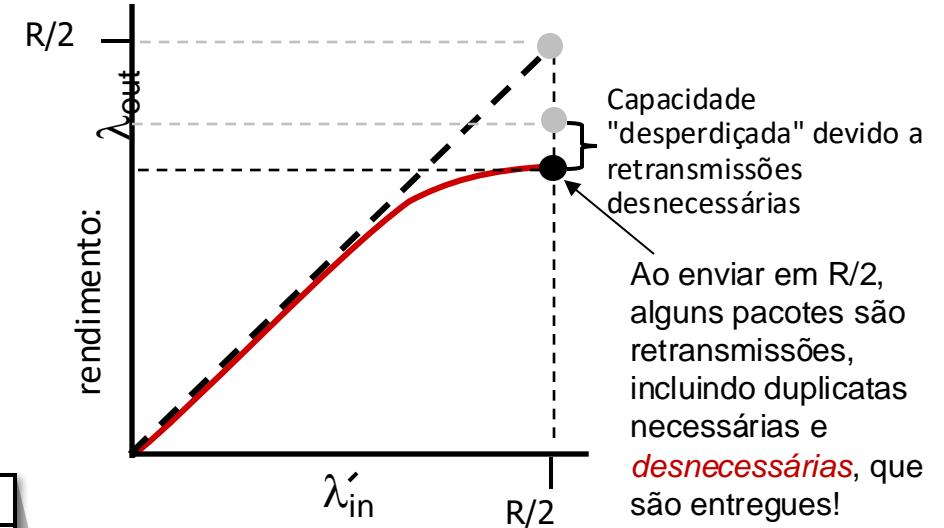
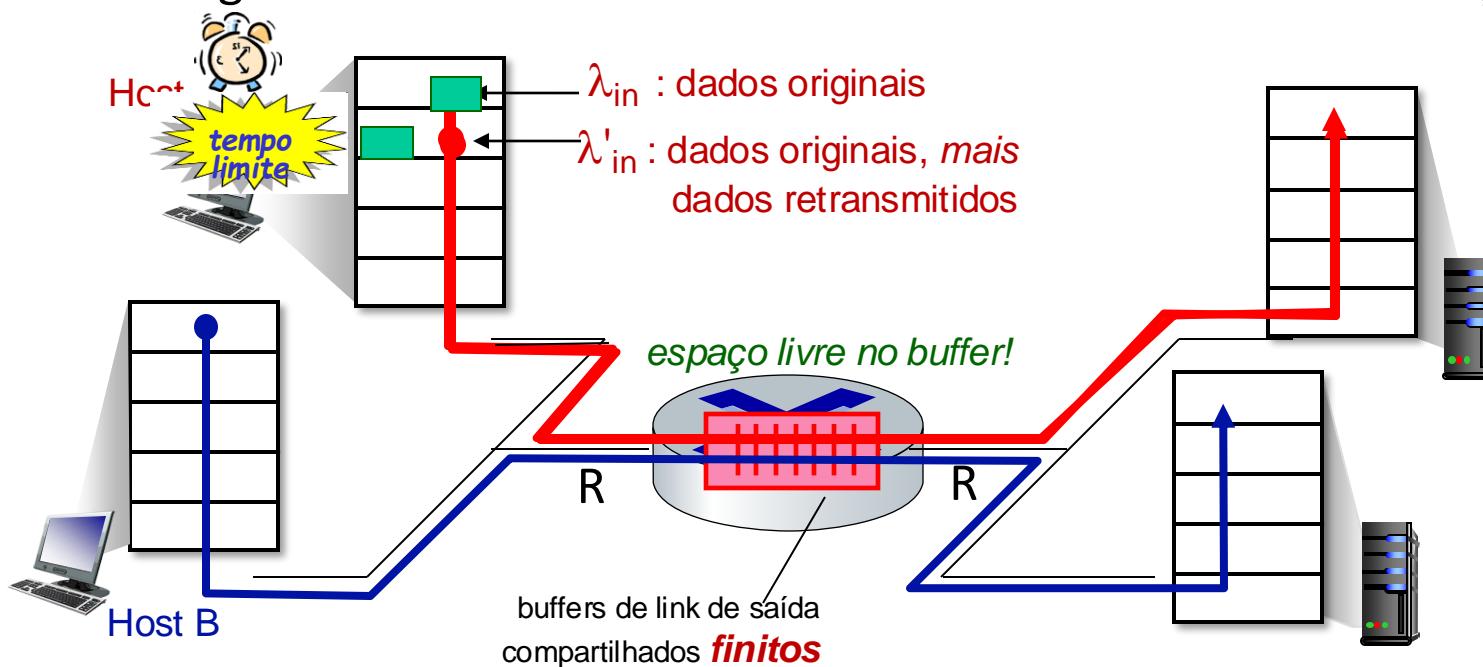
- os pacotes podem ser perdidos (descartados no roteador) devido a buffers cheios
- o remetente sabe quando o pacote foi descartado: só reenvia se o pacote *for sabidamente* perdido



# Causas/custos do congestionamento: cenário 2

## Cenário realista: *duplicatas desnecessárias*

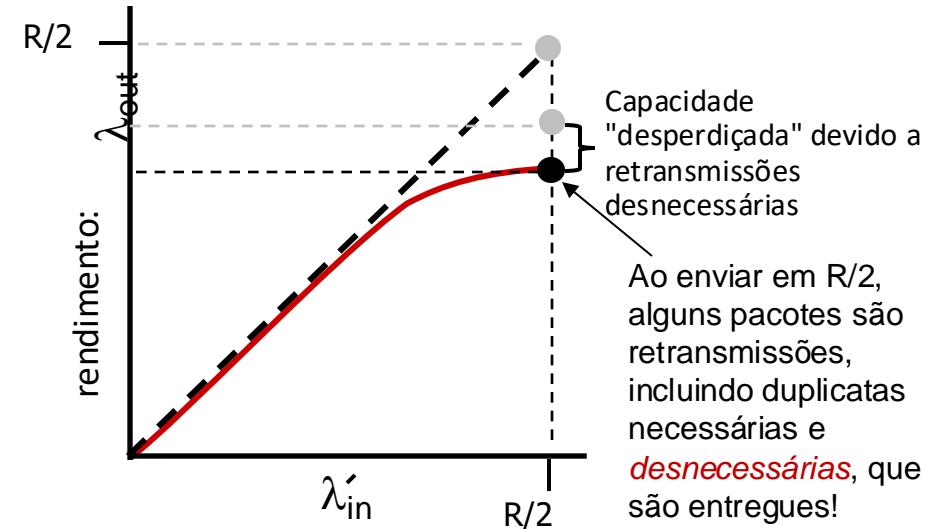
- os pacotes podem ser perdidos, descartados no roteador devido a buffers cheios, exigindo retransmissões
- mas o tempo do remetente pode expirar prematuramente, enviando *duas* cópias, *ambas* entregues



# Causas/custos do congestionamento: cenário 2

## Cenário realista: *duplicatas desnecessárias*

- os pacotes podem ser perdidos, descartados no roteador devido a buffers cheios, exigindo retransmissões
- mas o tempo do remetente pode expirar prematuramente, enviando *duas* cópias, *ambas* entregues



## "custos" do congestionamento:

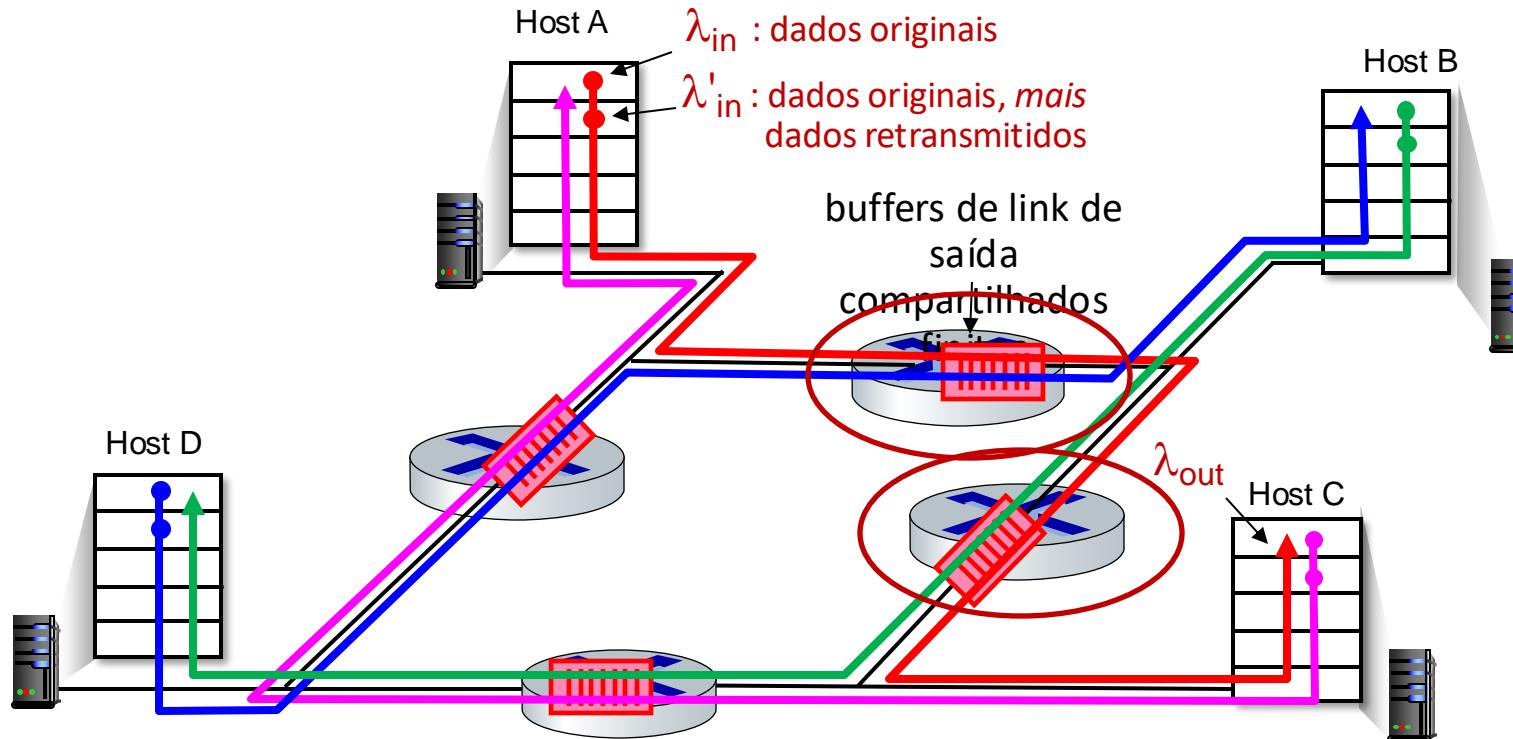
- mais trabalho (retransmissão) para uma determinada taxa de transferência do receptor
- retransmissões desnecessárias: o link carrega várias cópias de um pacote
  - diminuição da taxa de transferência máxima alcançável

# Causas/custos do congestionamento: cenário 3

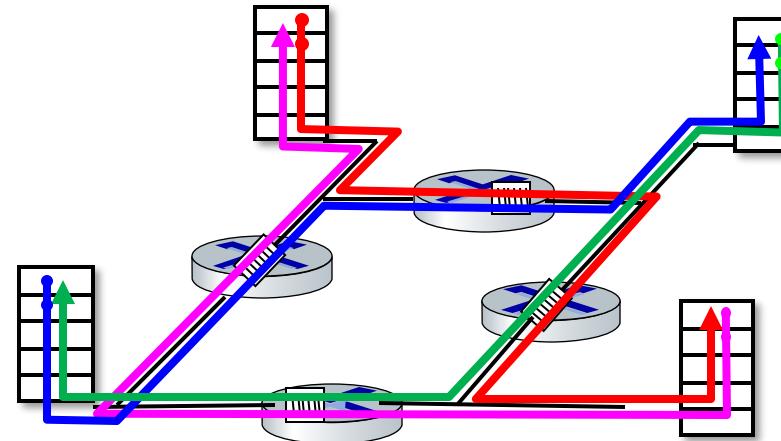
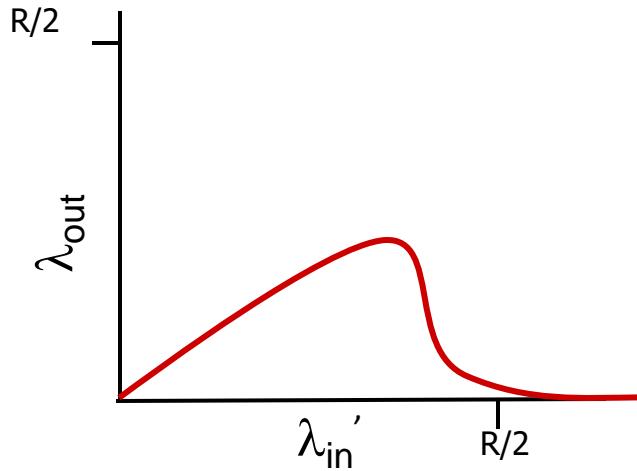
- quatro remetentes
- caminhos de múltiplos saltos
- timeout/retransmissão

P: O que acontece quando  $\lambda_{in}$  e  $\lambda'_{in}$  aumentam?

A: à medida que a  $\lambda_{in}'$  vermelha aumenta, todos os pkts azuis que chegam na fila superior são descartados, a taxa de transferência azul  $\rightarrow 0$



# Causas/custos do congestionamento: cenário 3

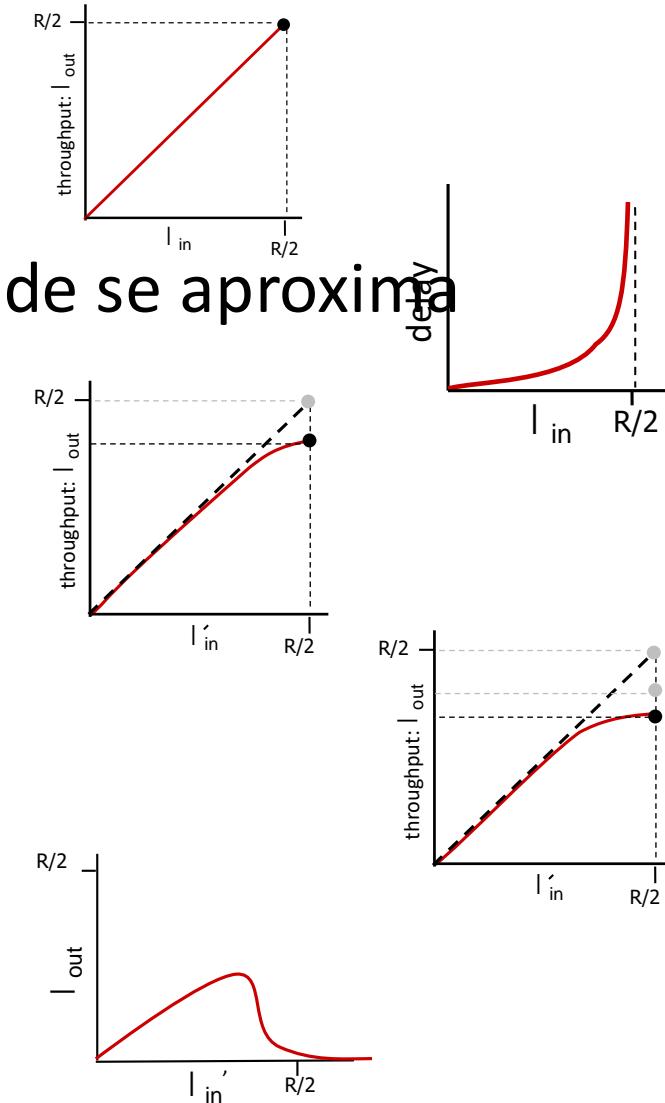


outro "custo" do congestionamento:

- Quando o pacote é descartado, toda a capacidade de transmissão upstream e o buffer usado para esse pacote são desperdiçados!

# Causas/custos do congestionamento: percepções

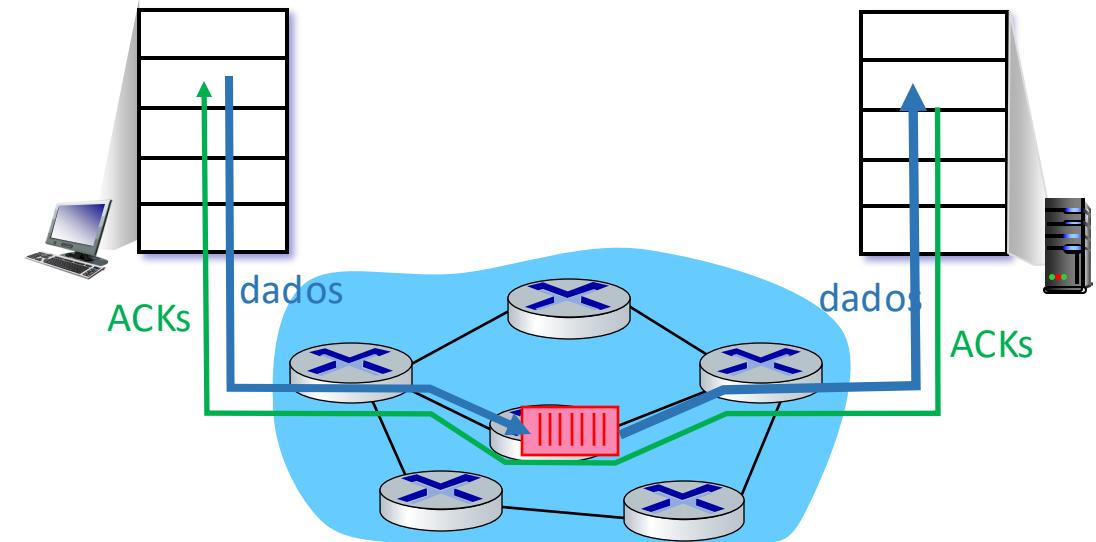
- a taxa de transferência nunca pode exceder a capacidade
- o atraso aumenta à medida que a capacidade se aproxima
- a perda/transmissão diminui a taxa de transferência efetiva
- duplicatas desnecessárias diminuem ainda mais a taxa de transferência efetiva
- capacidade de transmissão upstream / buffering desperdiçado para pacotes perdidos downstream



# Abordagens para o controle de congestionamento

Controle de congestionamento de extremidade:

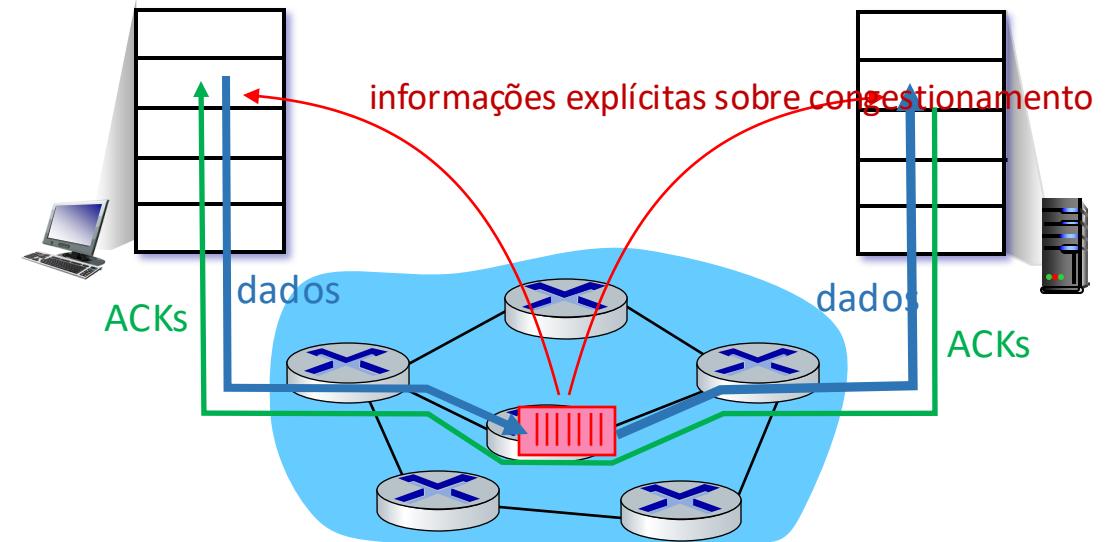
- nenhum feedback explícito da rede
- congestionamento *inferido* a partir da perda observada, atraso
- abordagem adotada pelo TCP



# Abordagens para o controle de congestionamento

## Controle de congestionamento assistido por rede:

- os roteadores fornecem feedback *direto* aos hosts de envio/recebimento com fluxos que passam pelo roteador congestionado
- pode indicar o nível de congestionamento ou definir explicitamente a taxa de envio
- Protocolos TCP ECN, ATM, DECbit



# Capítulo 3: roteiro

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



# Controle de congestionamento TCP: AIMD

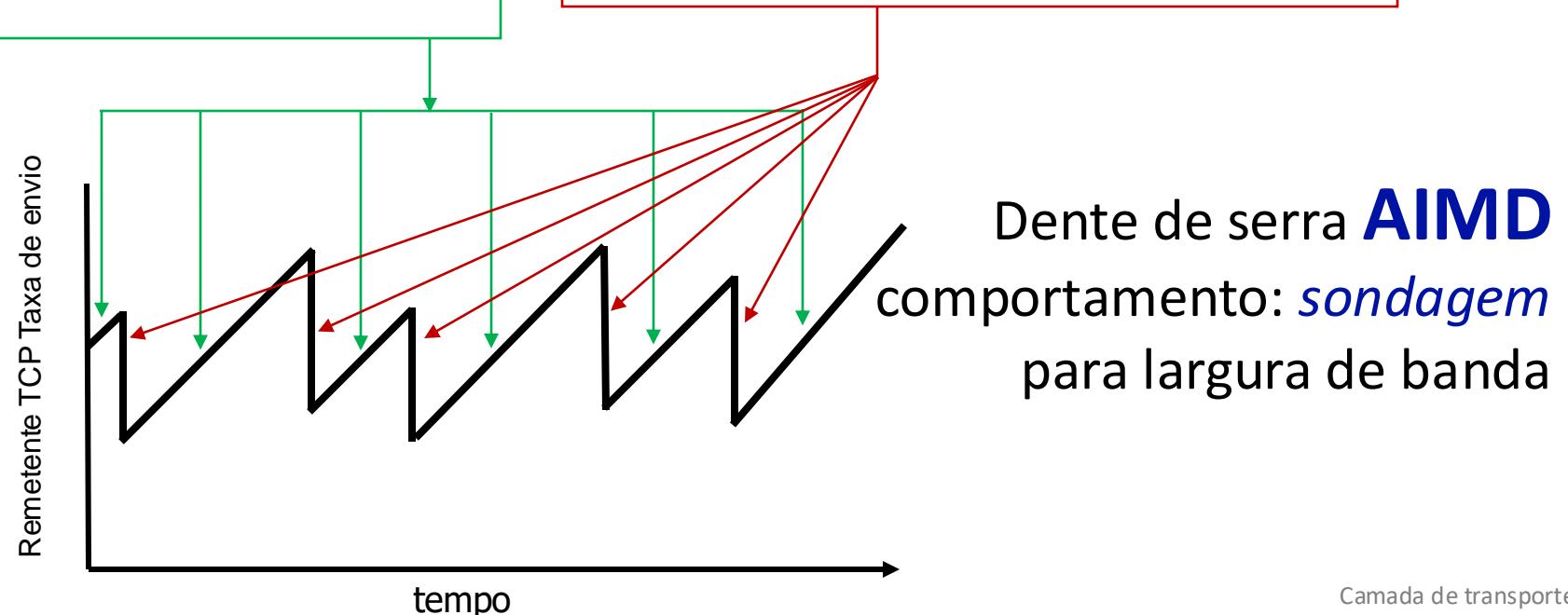
- *abordagem*: os remetentes podem aumentar a taxa de envio até que ocorra a perda de pacotes (congestionamento) e, em seguida, diminuir a taxa de envio em caso de perda

*Aumento de aditivos*

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

*Diminuição multiplicativa*

reduzir a taxa de envio pela metade a cada evento de perda



# TCP AIMD: mais

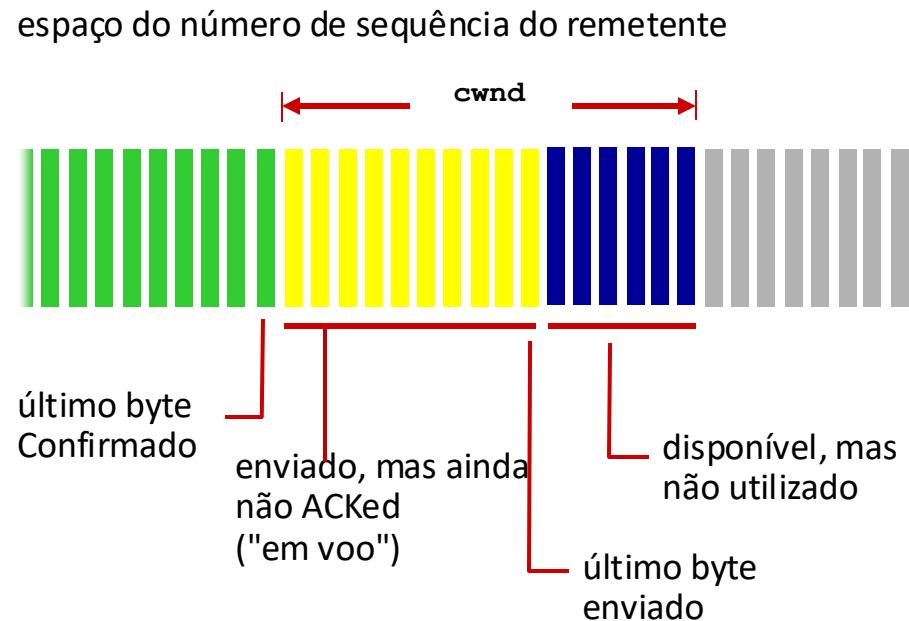
Detalhe *da redução multiplicativa*: a taxa de envio é

- Corte pela metade na perda detectada por ACK triplo duplicado (TCP Reno)
- Corte para 1 MSS (tamanho máximo do segmento) quando a perda for detectada por tempo limite (TCP Tahoe)

Por que a AIMD?

- Foi demonstrado que o AIMD - um algoritmo distribuído e assíncrono - é capaz de:
  - otimize as taxas de fluxo congestionado em toda a rede!
  - têm propriedades de estabilidade desejáveis

# Controle de congestionamento TCP: detalhes



Comportamento de envio de TCP:

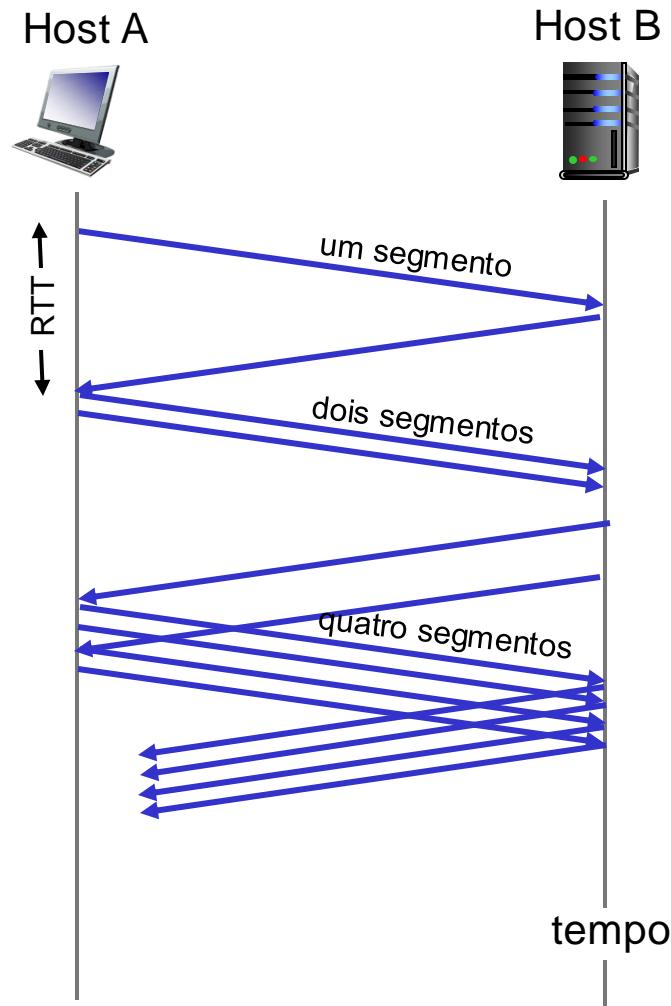
- *Grosso modo*: enviar bytes de cwnd, aguardar RTT para ACKS e, em seguida, enviar mais bytes

$$\text{Taxa TCP} \approx \frac{\text{cwnd}}{\text{RTT}} \text{ bytes/segundo}$$

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

# Início lento do TCP

- Quando a conexão começar, aumente a taxa exponencialmente até o primeiro evento de perda:
  - inicialmente **cwnd** = 1 MSS
  - double **cwnd** a cada RTT
  - feito pelo incremento do **cwnd** para cada ACK recebido
- *Resumo:* a taxa inicial é lenta, mas aumenta exponencialmente de forma rápida



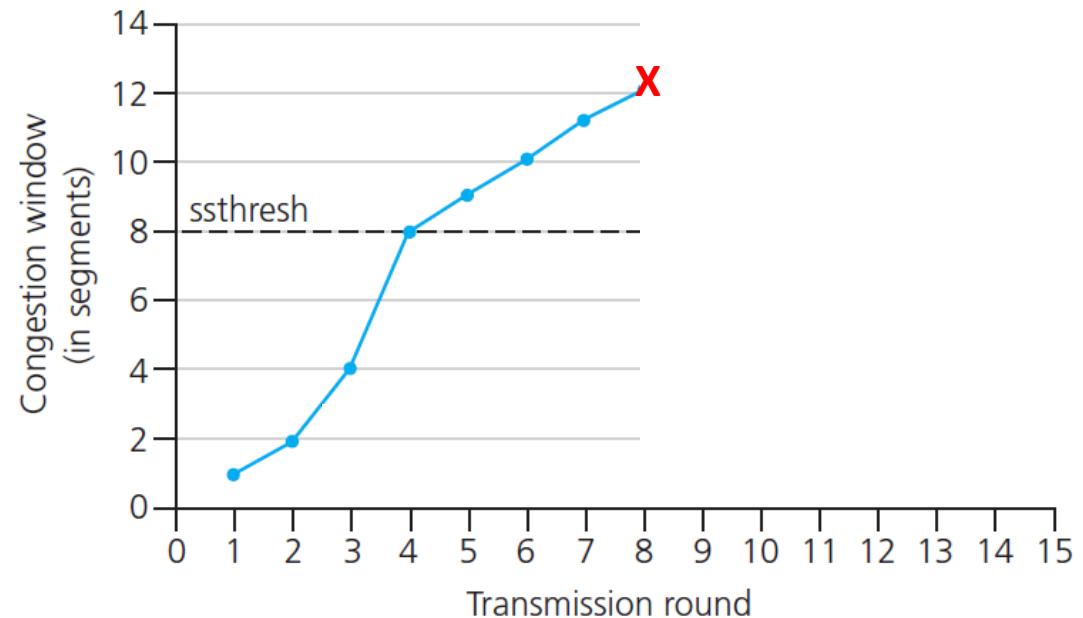
# TCP: do início lento à prevenção de congestionamento

*P:* Quando o aumento exponencial deve mudar para linear?

*R:* Quando o **cwnd** atinge  $1/2$  de seu valor antes do tempo limite

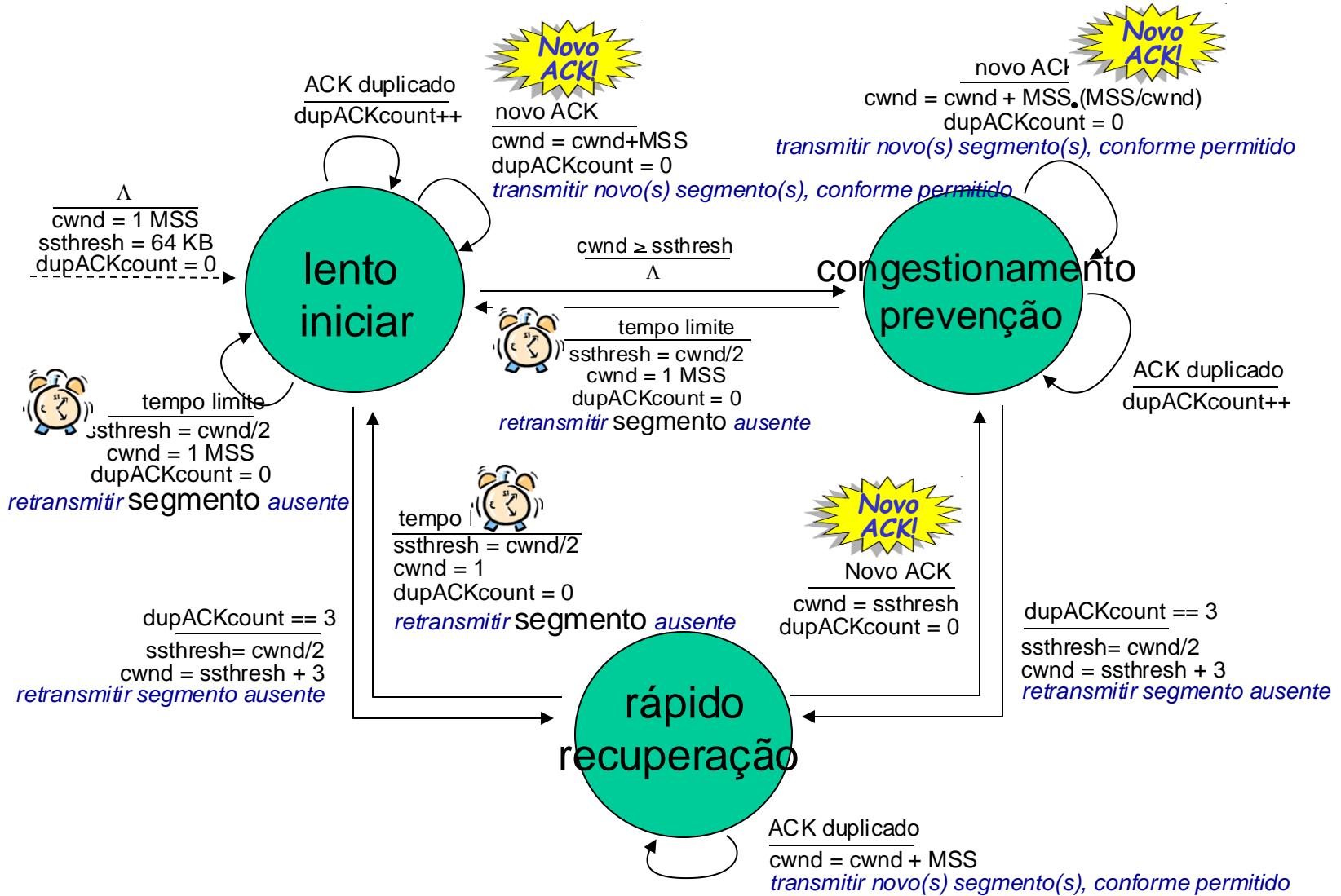
## Implementação:

- variável **ssthresh**
- no evento de perda, o **ssthresh** é definido como  $1/2$  do **cwnd** imediatamente antes do evento de perda



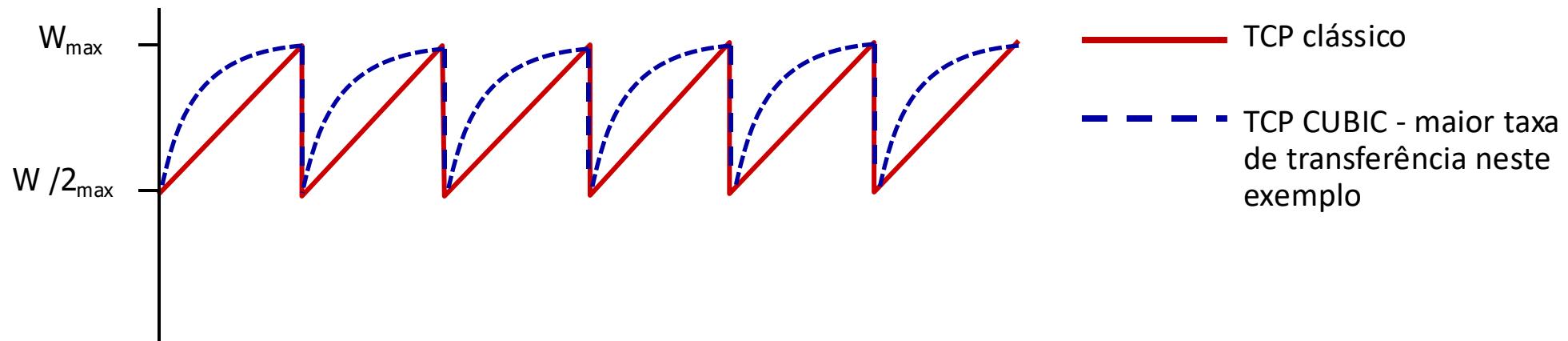
\* Confira os exercícios interativos on-line para obter mais exemplos: [http://gaia.cs.umass.edu/kurose\\_ross/interactive/](http://gaia.cs.umass.edu/kurose_ross/interactive/)

# Resumo: controle de congestionamento do TCP



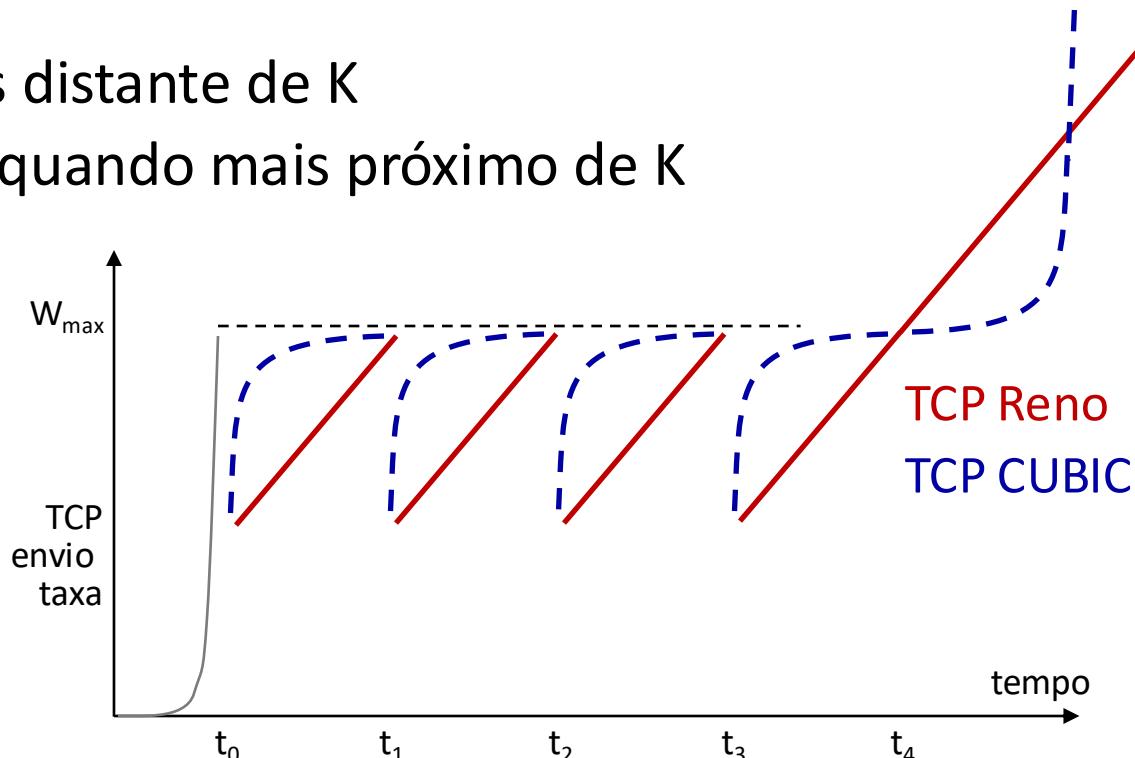
# TCP CUBIC

- Existe uma maneira melhor do que o AIMD para "sondar" a largura de banda
- Utilizável?
- Insight/intuição:
  - $W_{\max}$  : taxa de envio na qual a perda de congestionamento foi detectada
  - O estado de congestionamento do link de gargalo provavelmente (?) não mudou muito
  - Depois de reduzir a taxa/janela pela metade na perda, inicialmente aumente para  $W_{\max}$  *mais rápido*, mas depois aproxime-se de  $W_{\max}$  mais *lentamente*



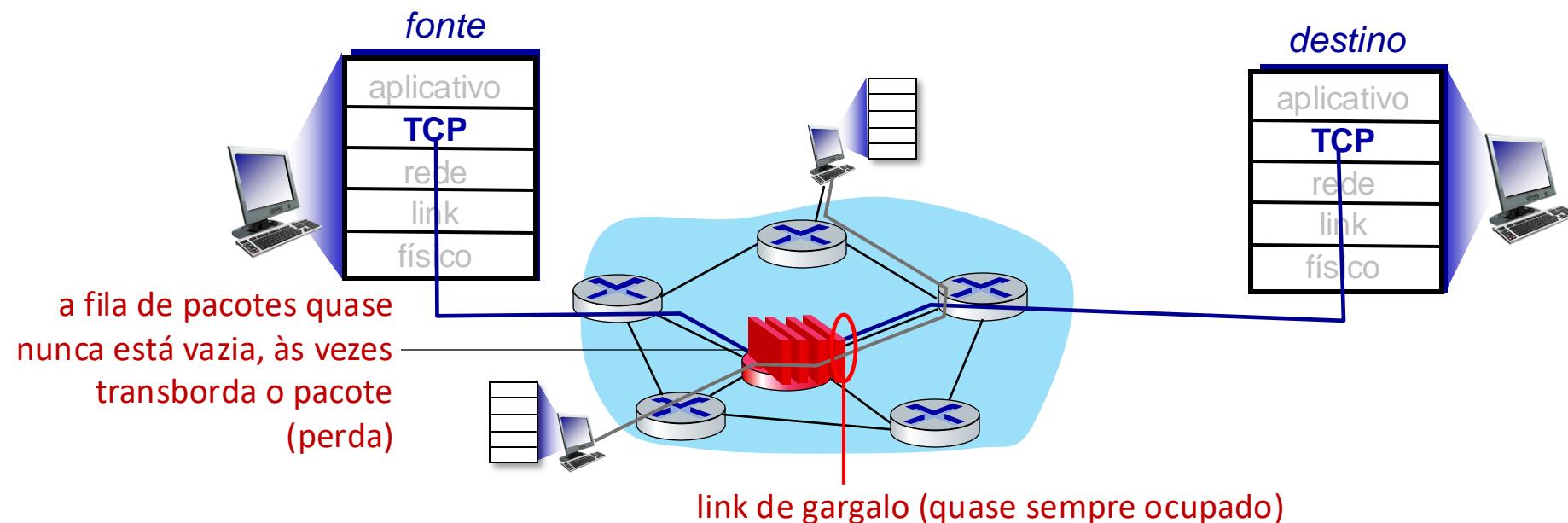
# TCP CUBIC

- K: momento em que o tamanho da janela TCP atingirá  $W_{\max}$ 
  - O próprio K é ajustável
- aumentam W como uma função do *cubo* da distância entre o tempo atual e K
  - maior aumenta quando mais distante de K
  - menor aumenta (cauteloso) quando mais próximo de K
- TCP CUBIC padrão no Linux, o TCP mais popular para servidores da Web populares



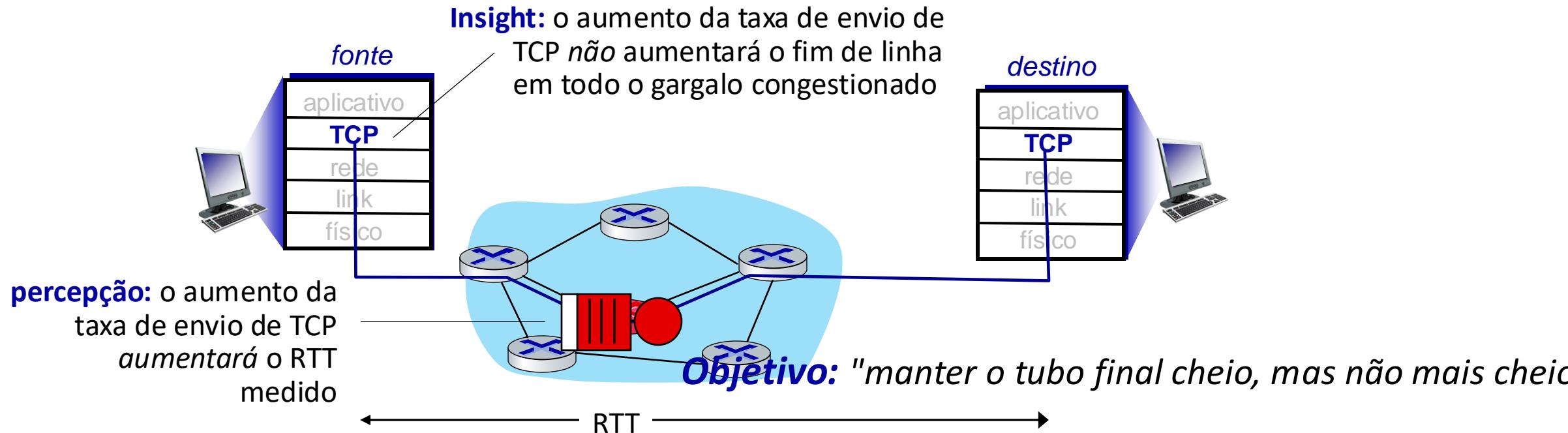
# TCP e o "link de gargalo" congestionado

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



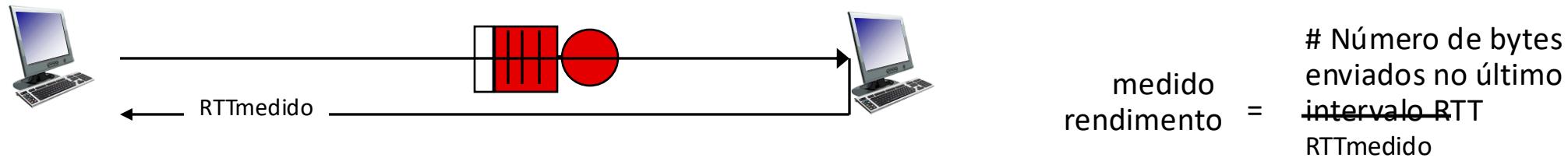
# TCP e o "link de gargalo" congestionado

- TCP (clássico, CUBIC) aumenta a taxa de envio do TCP até que ocorra perda de pacotes na saída de algum roteador: o *link de gargalo*
- Compreensão do congestionamento: é útil concentrar-se no link de gargalo congestionado



# Controle de congestionamento TCP baseado em atraso

Manter o tubo do remetente para o receptor "cheio o suficiente, mas não mais cheio": manter o link de gargalo ocupado com a transmissão, mas evitar grandes atrasos/buffering



## Abordagem baseada em atrasos:

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

se a taxa de transferência medida for "muito próxima" da taxa de transferência não congestionada

aumentar o  $cwnd$  linearmente /\* já que o caminho não está congestionado \*/

caso contrário, se a taxa de transferência medida for "muito inferior" à não congestionada durante todo o período

diminuir o  $cwnd$  linearmente /\* já que o caminho está congestionado \*/

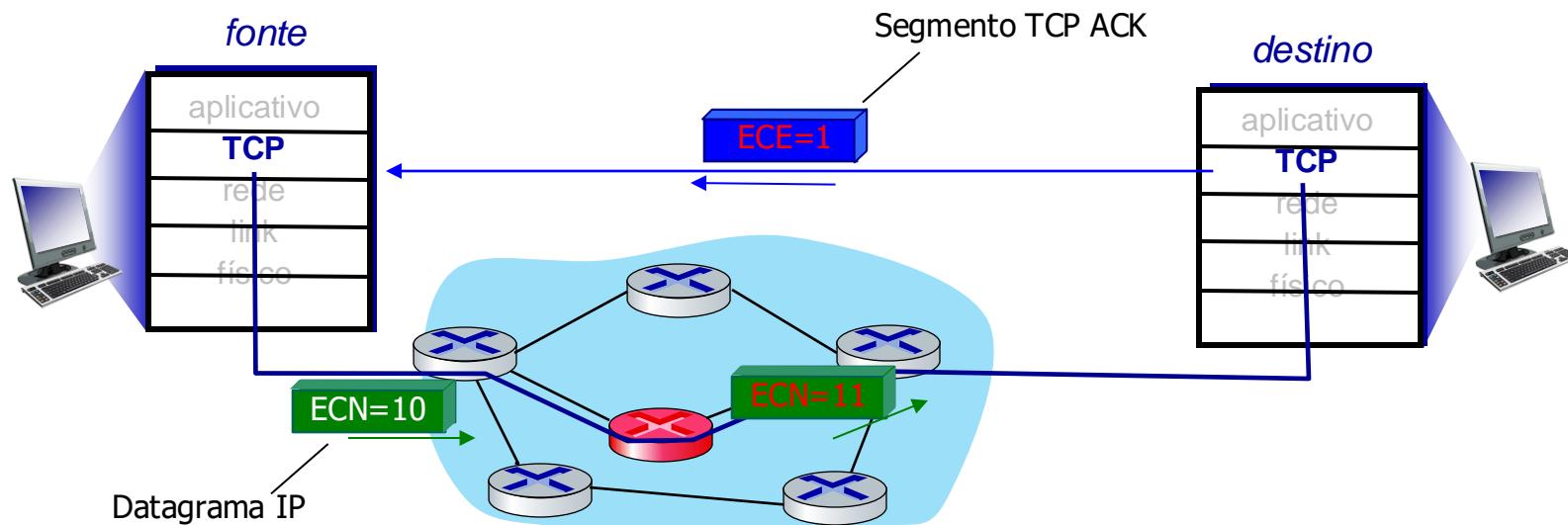
# Controle de congestionamento TCP baseado em atraso

- controle de congestionamento sem induzir/forçar a perda
- maximização de todo o processo ("manter o tubo justo cheio...") e, ao mesmo tempo, manter o atraso baixo ("...mas não mais cheio")
- vários TCPs implementados adotam uma abordagem baseada em atraso
  - BBR implementado na rede de backbone (interna) do Google

# Notificação explícita de congestionamento (ECN)

As implantações de TCP geralmente implementam o controle de congestionamento *assistido pela rede*:

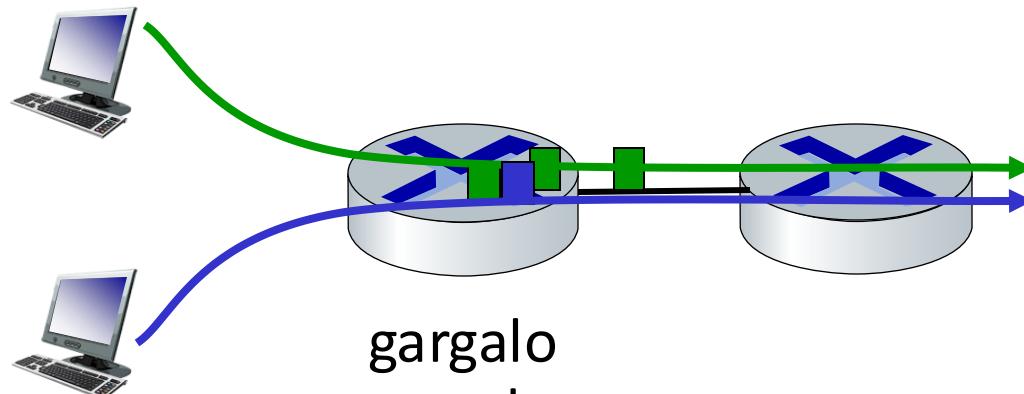
- dois bits no cabeçalho IP (campo ToS) marcados *pelo roteador de rede* para indicar congestionamento
  - *política* para determinar a marcação escolhida pela operadora de rede
- indicação de congestionamento levada ao destino
- O destino define o bit ECE no segmento ACK para notificar o remetente sobre o congestionamento
- envolve tanto IP (marcação de bit ECN do cabeçalho IP) quanto TCP (marcação de bit C,E do cabeçalho TCP)



# Equidade do TCP

**Meta de equidade:** se  $K$  sessões TCP compartilharem o mesmo link de gargalo com largura de banda  $R$ , cada uma deverá ter uma taxa média de  $R/K$

Conexão TCP 1



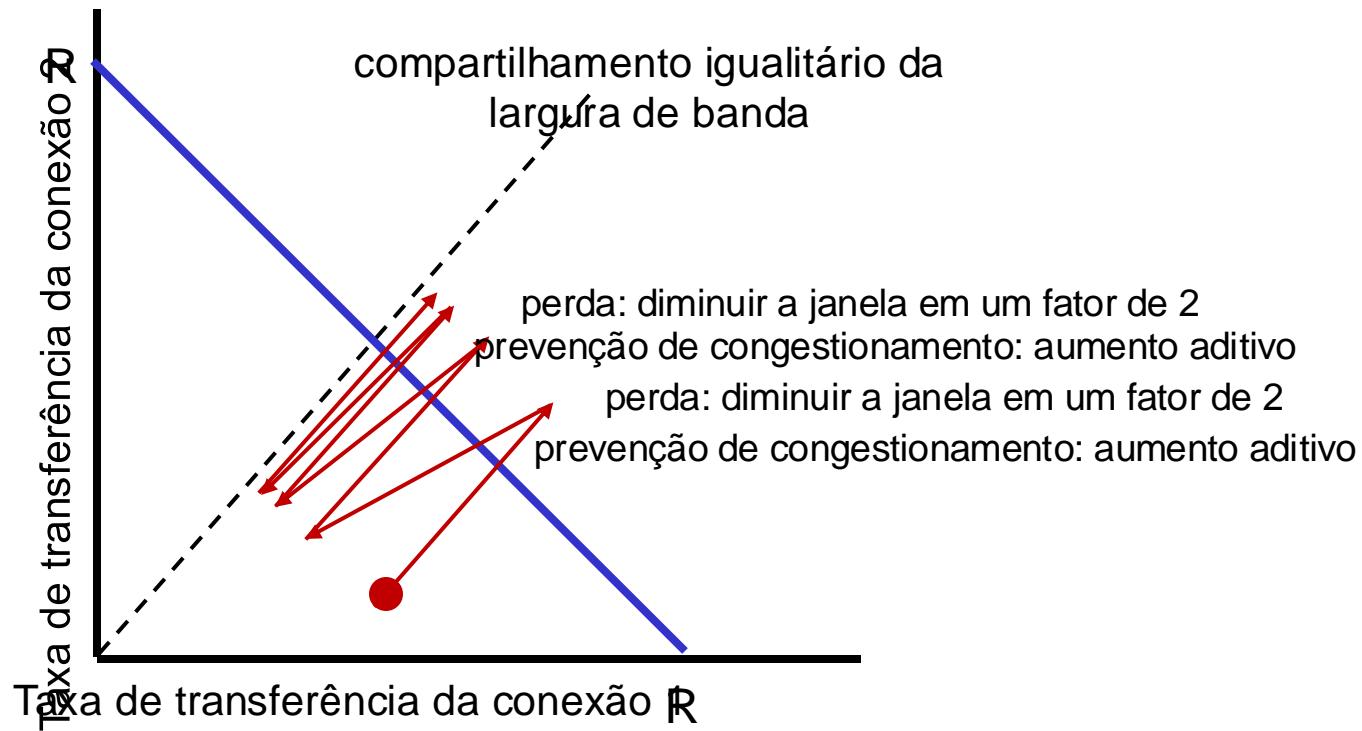
Conexão TCP 2

gargalo  
roteador  
capacidade R

# P: O TCP é justo?

Exemplo: duas sessões TCP concorrentes:

- o aumento aditivo dá uma inclinação de 1, à medida que aumenta
- a redução multiplicativa diminui a taxa de transferência proporcionalmente



O TCP é justo?

**R:** Sim, sob suposições idealizadas:

- mesmo RTT
- número fixo de sessões somente para evitar congestionamentos

# Equidade: todos os aplicativos de rede devem ser "justos"?

## Equidade e UDP

- os aplicativos multimídia geralmente não usam TCP
  - não querem que a taxa seja estrangulada pelo controle de congestionamento
- em vez disso, use UDP:
  - enviar áudio/vídeo a uma taxa constante, tolerar perda de pacotes
- não existe uma "polícia da Internet" que policie o uso do controle de congestionamento

## Equidade, conexões TCP paralelas

- O aplicativo pode abrir *várias* conexões paralelas entre dois hosts
- Os navegadores da Web fazem isso, por exemplo, link de taxa R com 9 conexões existentes:
  - O novo aplicativo solicita 1 TCP, obtém a taxa  $R/10$
  - O novo aplicativo solicita 11 TCPs e recebe  $R/2$

# Camada de transporte: roteiro

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



# Evolução da funcionalidade da camada de transporte

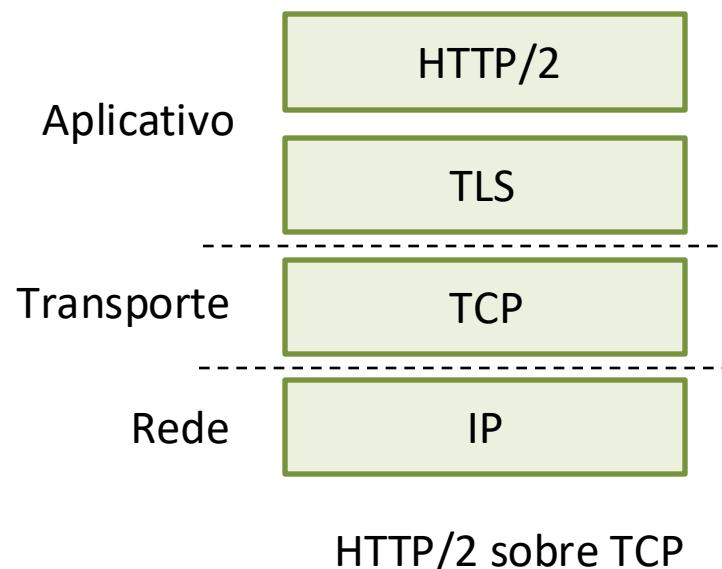
- TCP, UDP: principais protocolos de transporte há 40 anos
- diferentes "sabores" de TCP desenvolvidos para cenários específicos:

Cenário	Desafios
Tubos longos e gordos (grandes transferências de dados)	Muitos pacotes "em voo"; a perda interrompe o pipeline
Redes sem fio	Perda devido a links sem fio com ruído, mobilidade; o TCP trata isso como perda de congestionamento
Links de longo prazo	RTTs extremamente longos
Redes de data center	Sensível à latência
Fluxos de tráfego de fundo	Fluxos TCP de baixa prioridade e "em segundo plano"

- mover as funções da camada de transporte para a camada de aplicativos, sobre o UDP
  - HTTP/3: QUIC

# QUIC: Conexões de Internet UDP rápidas

- protocolo de camada de aplicativo, sobre o UDP
  - aumentar o desempenho do HTTP
  - implantado em muitos servidores e aplicativos do Google (Chrome, aplicativo móvel do YouTube)

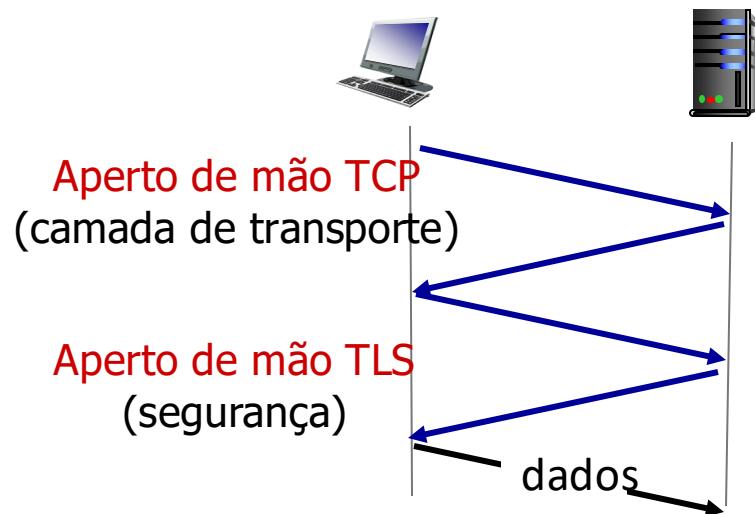


# QUIC: Conexões de Internet UDP rápidas

adota as abordagens que estudamos neste capítulo para estabelecimento de conexão, controle de erros, controle de congestionamento

- **controle de erros e congestionamento:** "Os leitores familiarizados com a detecção de perdas e o controle de congestionamento do TCP encontrarão aqui algoritmos paralelos aos conhecidos do TCP." [da especificação QUIC]
- **estabelecimento de conexão:** confiabilidade, controle de congestionamento, autenticação, criptografia, estado estabelecido em um RTT
- vários "fluxos" em nível de aplicativo multiplexados em uma única conexão QUIC
  - transferência de dados confiável separada, segurança
  - controle de congestionamento comum

# QUIC: Estabelecimento de conexão



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

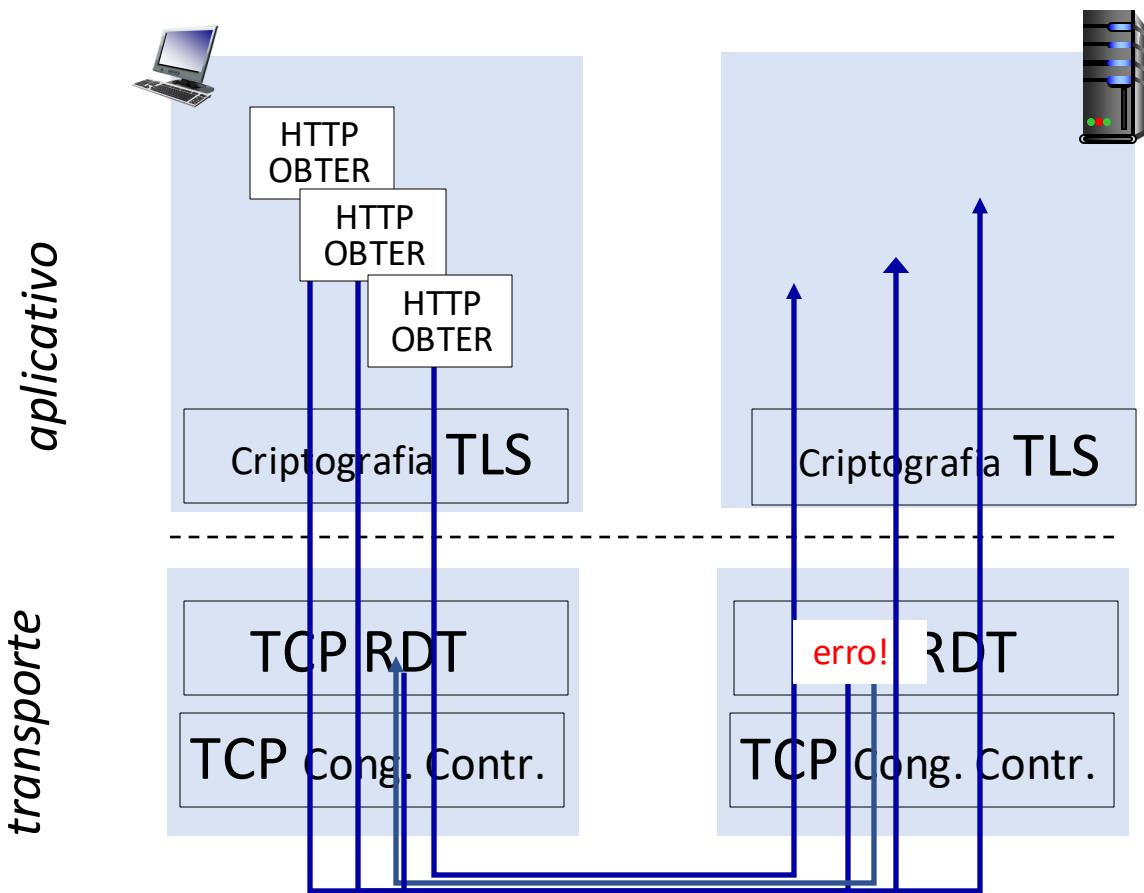
- 2 handshakes seriais



QUIC: confiabilidade, controle de congestionamento, autenticação, estado da criptografia

- 1 aperto de mão

# QUIC: fluxos: paralelismo, sem bloqueio de HOL



(a) HTTP 1.1

# Capítulo 3: resumo

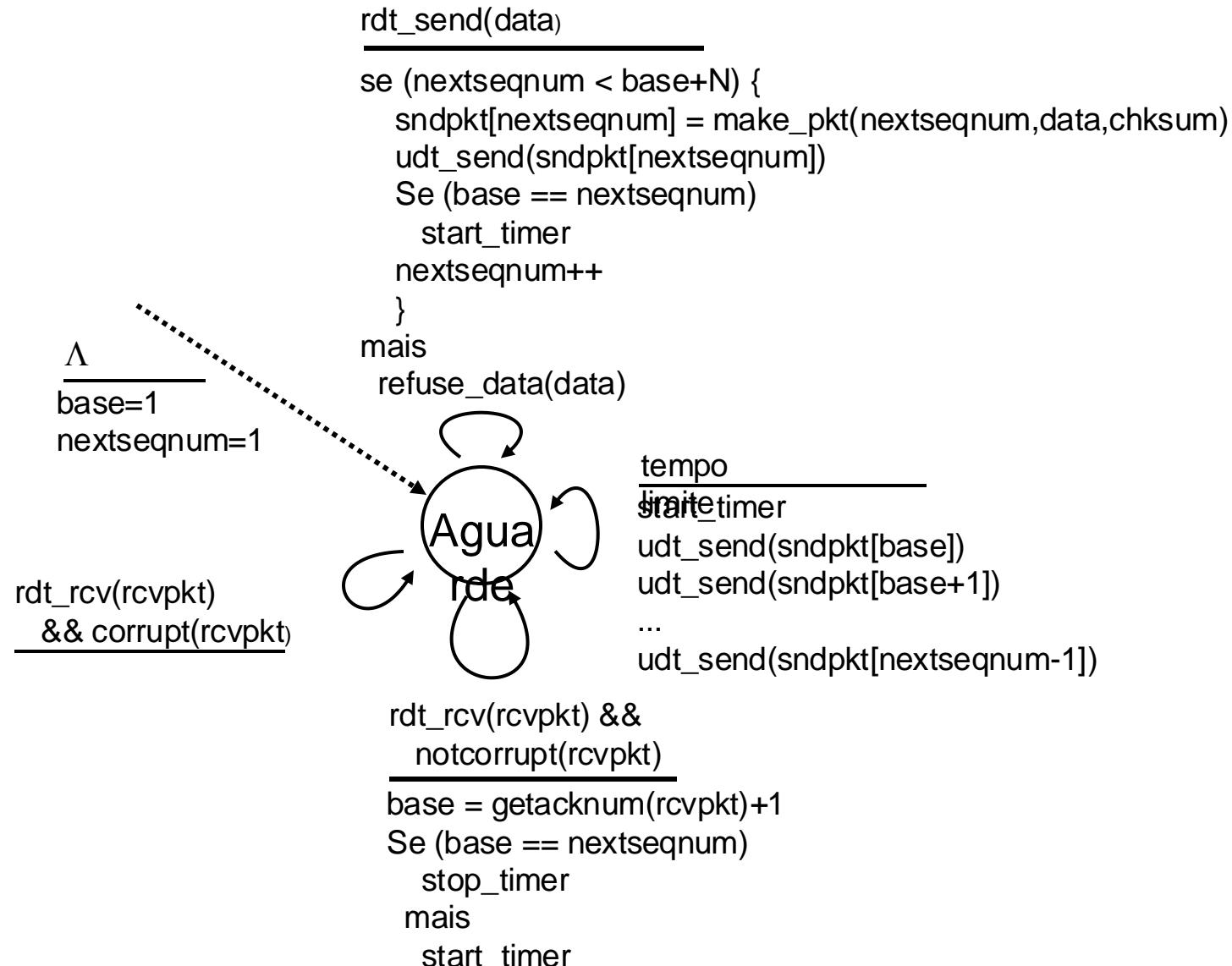
- princípios por trás dos serviços da camada de transporte:
  - multiplexação, demultiplexação
  - transferência de dados confiável
  - controle de fluxo
  - controle de congestionamento
- instanciação, implementação na Internet
  - UDP
  - TCP

## A seguir:

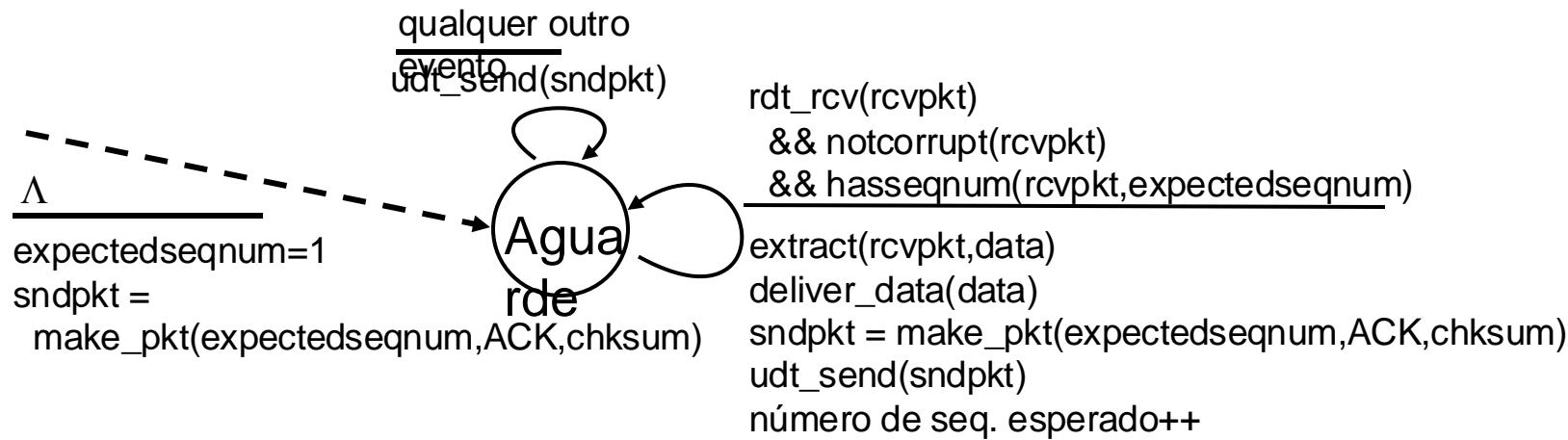
- saindo da "borda" da rede (camadas de aplicativo e transporte)
- no "núcleo" da rede
- dois capítulos da camada de rede:
  - plano de dados
  - plano de controle

# Slides adicionais do Capítulo 3

# Go-Back-N: FSM estendido do remetente



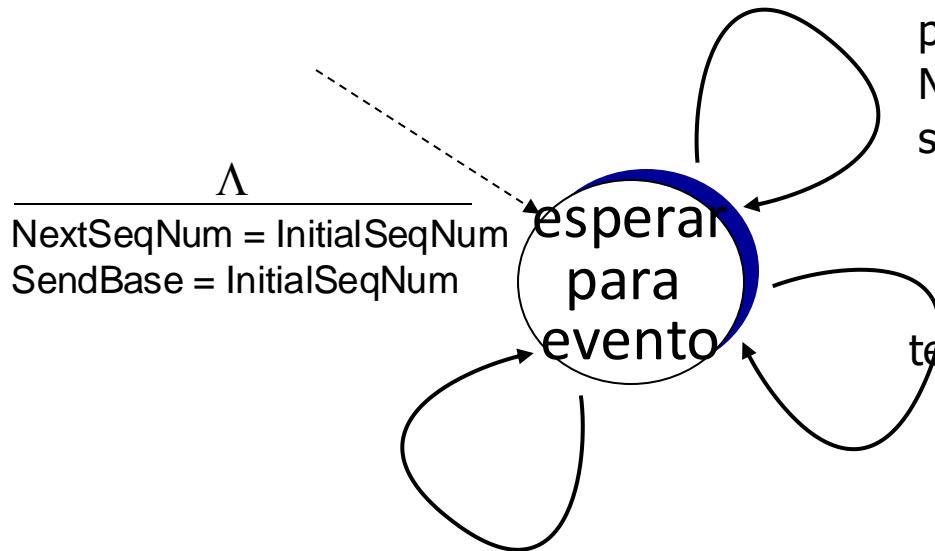
# Go-Back-N: FSM estendido do receptor



ACK-only: sempre envia ACK para o pacote recebido corretamente com o número de seq. mais alto *na ordem*

- pode gerar ACKs duplicados
  - só precisa se lembrar do **expectedseqnum**
- pacote fora de ordem:
- descartar (não armazenar em buffer): *nenhum receptor armazenando em buffer!*
  - reenviar o pkt com a seq. mais alta na ordem

# Remetente TCP (simplificado)



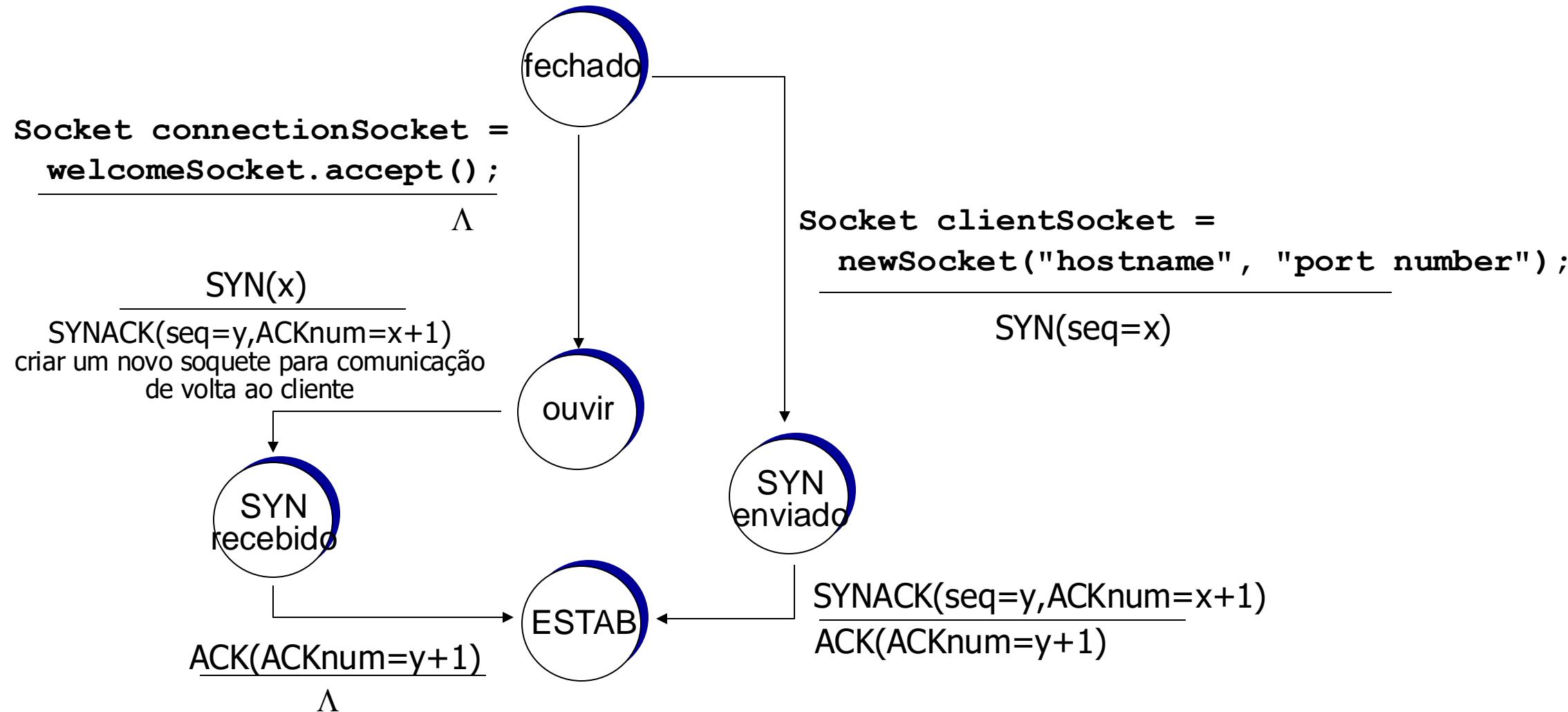
ACK recebido, com o valor do campo ACK y

```
Se (y > SendBase) {  
    SendBase = y  
    /* SendBase-1: último byte ACKed cumulativo */  
    se (houver segmentos que ainda não foram empacotados)  
        iniciar o cronômetro  
    senão, pare o cronômetro  
}
```

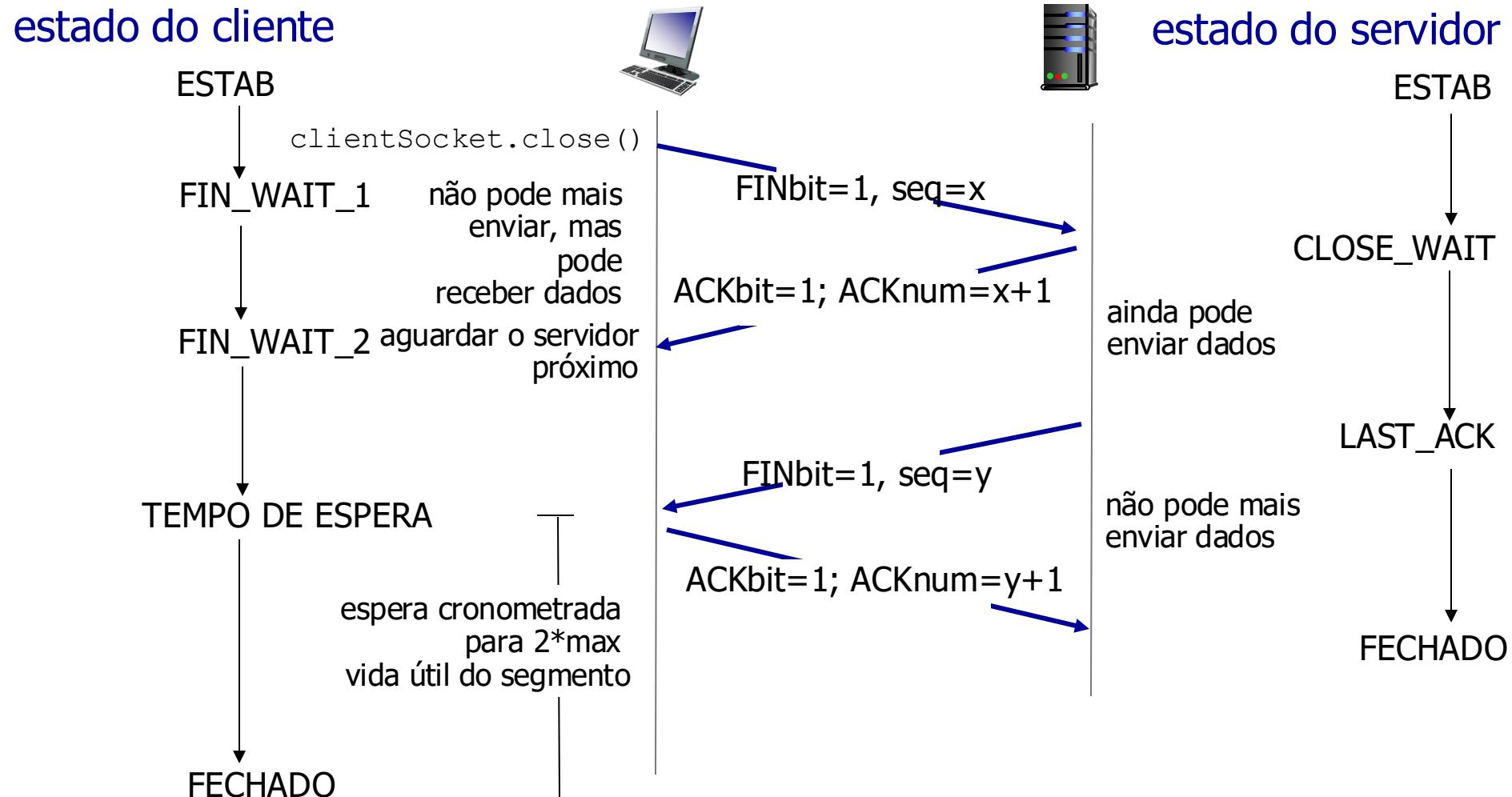
dados recebidos do aplicativo acima  
criar segmento, seq. #: NextSeqNum  
passar o segmento para o IP (ou seja, "enviar")  
NextSeqNum = NextSeqNum + length(data)  
se (o cronômetro não estiver em execução no momento)  
 iniciar o cronômetro

tempo limite  
retransmitir o segmento ainda não  
empacotado com a menor seq. #  
iniciar o cronômetro

# FSM de handshake de 3 vias do TCP



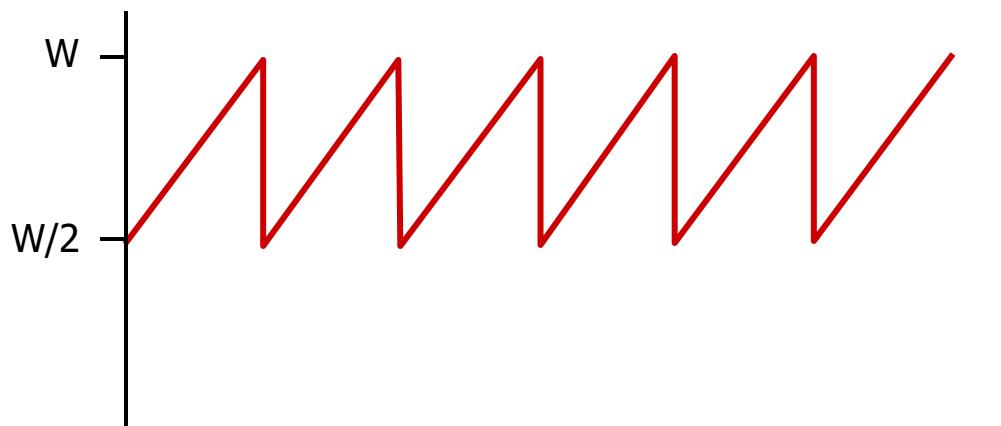
# Fechamento de uma conexão TCP



# Taxa de transferência de TCP

- Taxa de transferência média de TCP como função do tamanho da janela, RTT?
  - ignorar o início lento, presumir que sempre há dados para enviar
- W: tamanho da janela (medido em bytes) em que ocorre a perda
  - O tamanho médio da janela (número de bytes em voo) é  $\frac{3}{4}W$
  - A potência média é de  $\frac{3}{4}W$  por RTT

$$\text{taxa de transferência média de TCP} = \frac{3}{4} \frac{W}{RTT} \text{ bytes/segundo}$$



# TCP sobre "canos longos e gordos"

- Exemplo: segmentos de 1.500 bytes, RTT de 100 ms, desejo de taxa de transferência de 10 Gbps
- requer  $W = 83.333$  segmentos de voo
- rendimento em termos de probabilidade de perda de segmento,  $L$  [Mathis 1997]:

$$\text{Taxa de transferência TCP} = \frac{1.22 \cdot \text{MSS}}{\text{RTT} \sqrt{L}}$$

→ para atingir uma taxa de transferência de 10 Gbps, é necessária uma taxa de perda de  $L = 2 \cdot 10^{-10}$  - *uma taxa de perda muito pequena!*

- versões do TCP para cenários longos e de alta velocidade