

UDP - User Datagram Protocol

- Protocolo de transporte da Internet “cru”
- Serviço de “melhor esforço”, segmentos UDP podem ser:
 - Perdidos
 - Entregues fora de ordem
- **Sem conexão:**
 - Sem *handshaking* entre o remetente e o receptor UDP
 - Cada segmento UDP tratado independente dos outros

UDP - User Datagram Protocol

- Por que existe UDP?
 - Sem estabelecimento de conexão (adiciona atraso RTT)
 - Simples: sem estado de conexão no remetente e no receptor
 - Cabeçalho pequeno
 - Sem controle de congestionamento
 - Pode viajar tão rápido quanto desejado!
 - Pode funcionar mesmo em congestionamento

UDP - User Datagram Protocol

- Uso do UDP:
 - Streaming de aplicações multimídia (tolerantes a perdas, sensível a taxas de transmissão)
 - DNS
 - SNMP
 - HTTP/3
- Se for necessária uma transferência confiável UDP (e.g., HTTP/3)
 - Adicione a confiabilidade necessária na camada de aplicação
 - Adicione controle de congestionamento na camada de aplicação

UDP - User Datagram Protocol

- RFC 768

```
INTERNET STANDARD

RFC 768                                J. Postel
                                         ISI
                                         28 August 1980
```

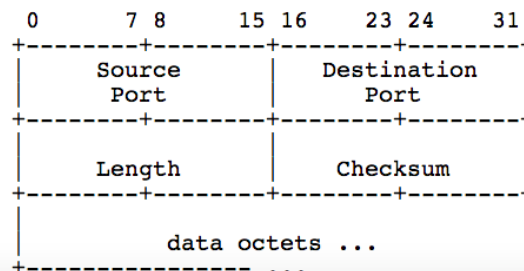
User Datagram Protocol

Introduction

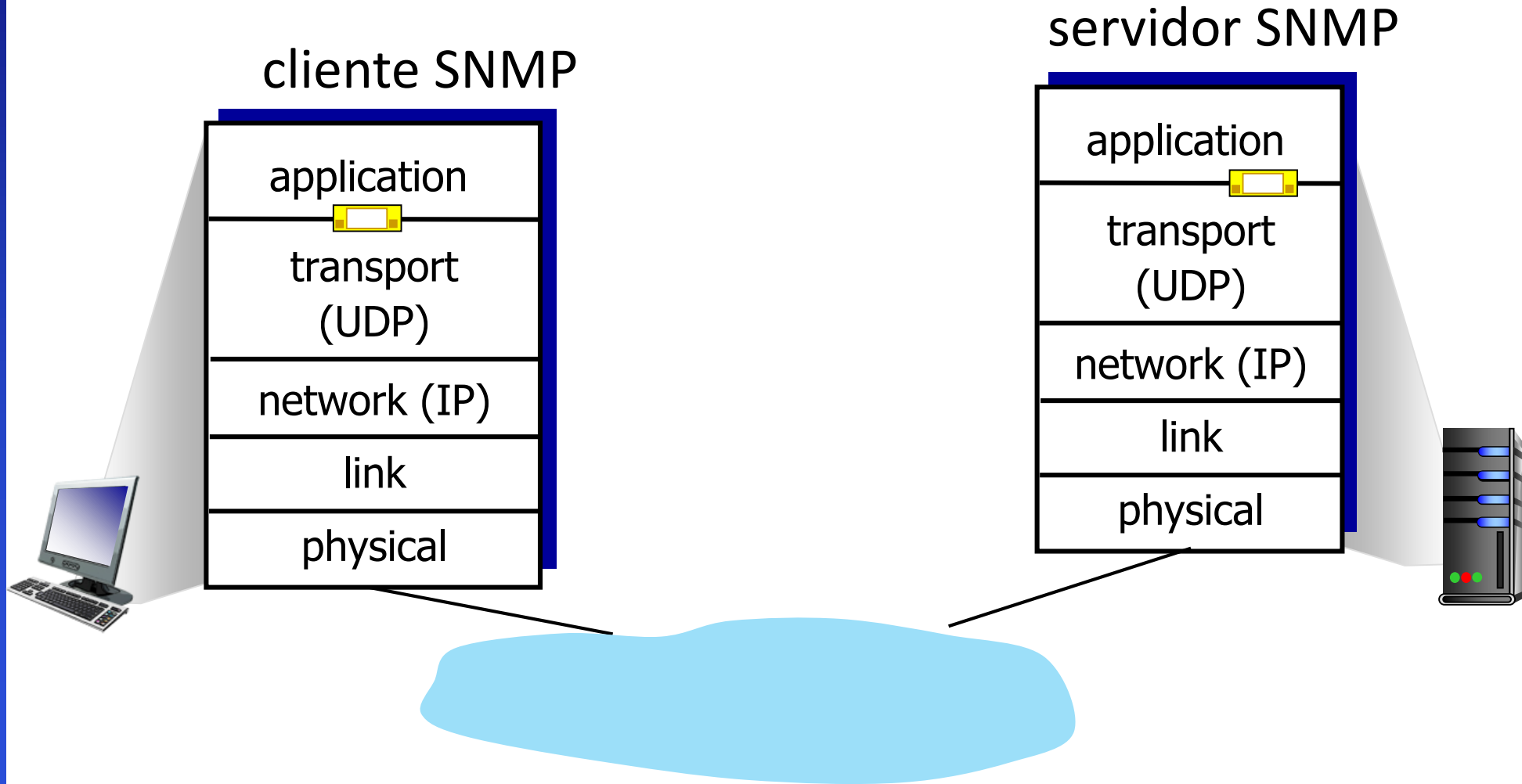
This User Datagram Protocol (UDP) is defined to make available a datagram mode of packet-switched computer communication in the environment of an interconnected set of computer networks. This protocol assumes that the Internet Protocol (IP) [1] is used as the underlying protocol.

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].

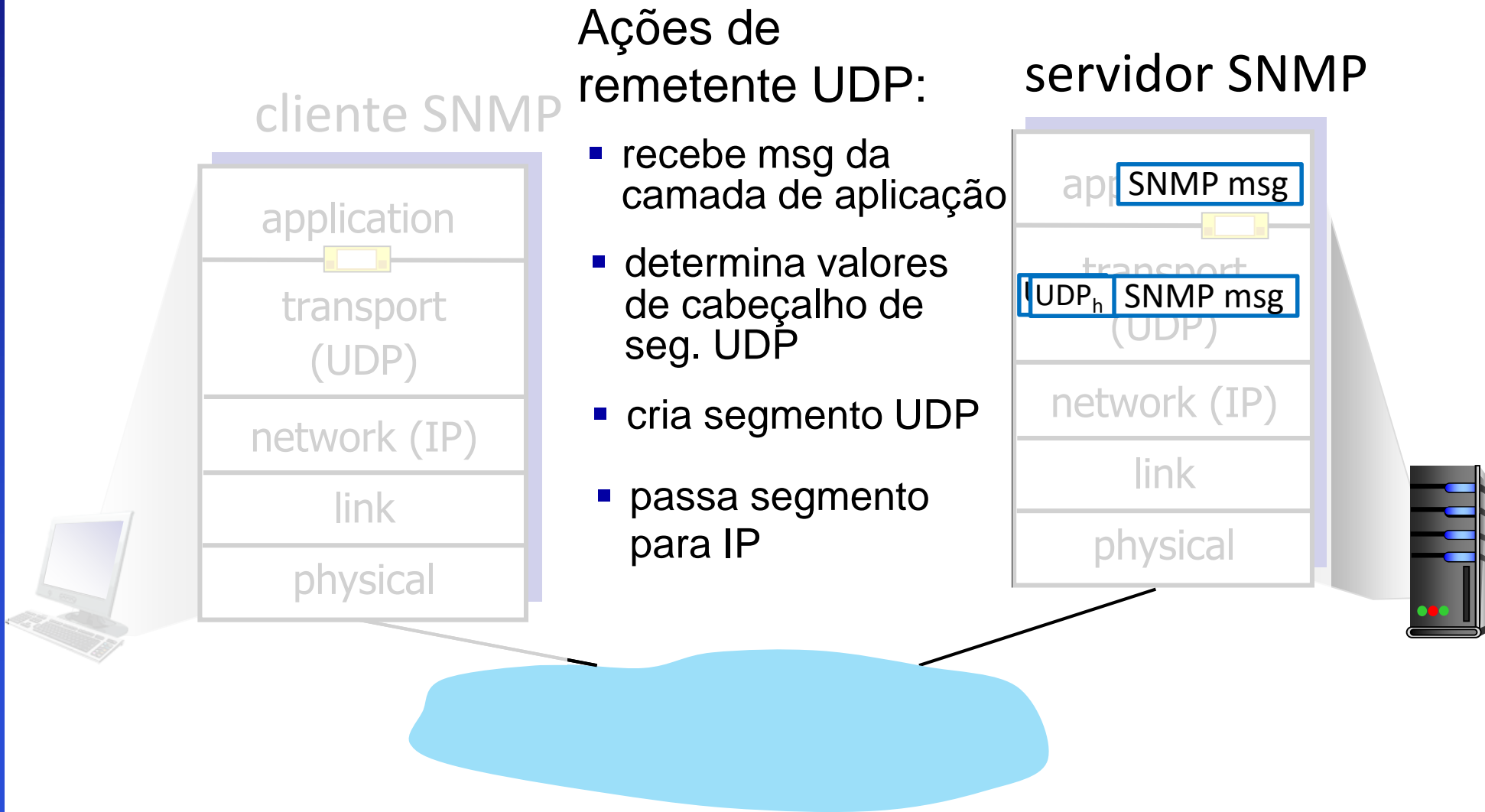
Format



UDP – Ações da camada de transporte



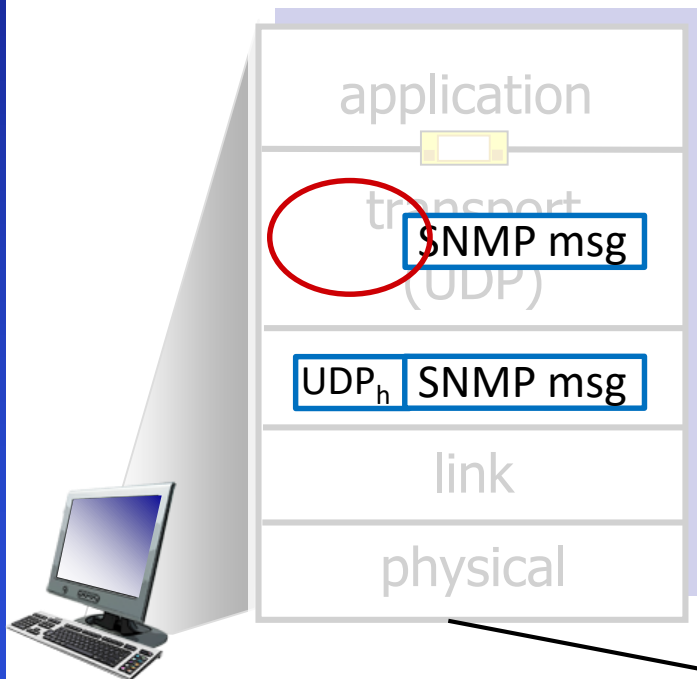
UDP – Ações da camada de transporte



UDP – Ações da camada de transporte

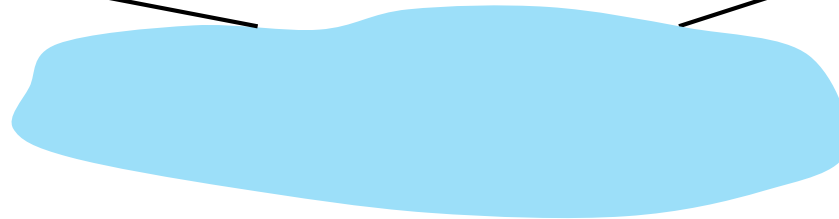
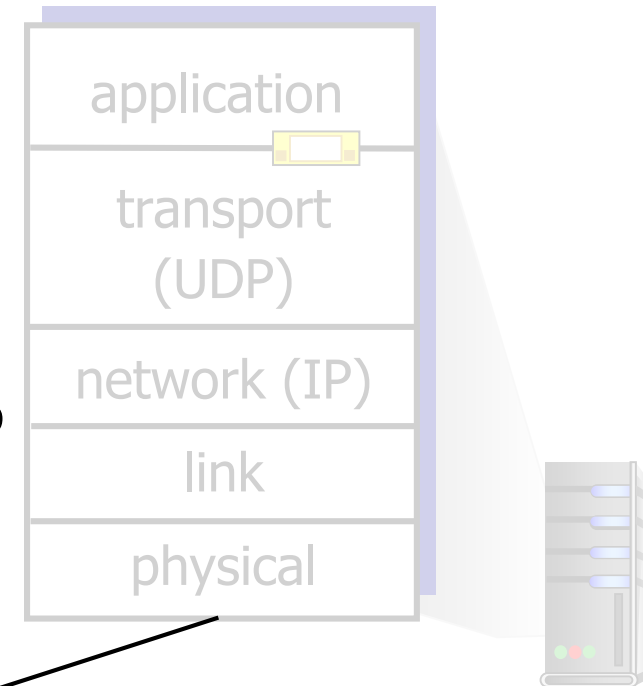
Ações do receptor UDP:

cliente SNMP

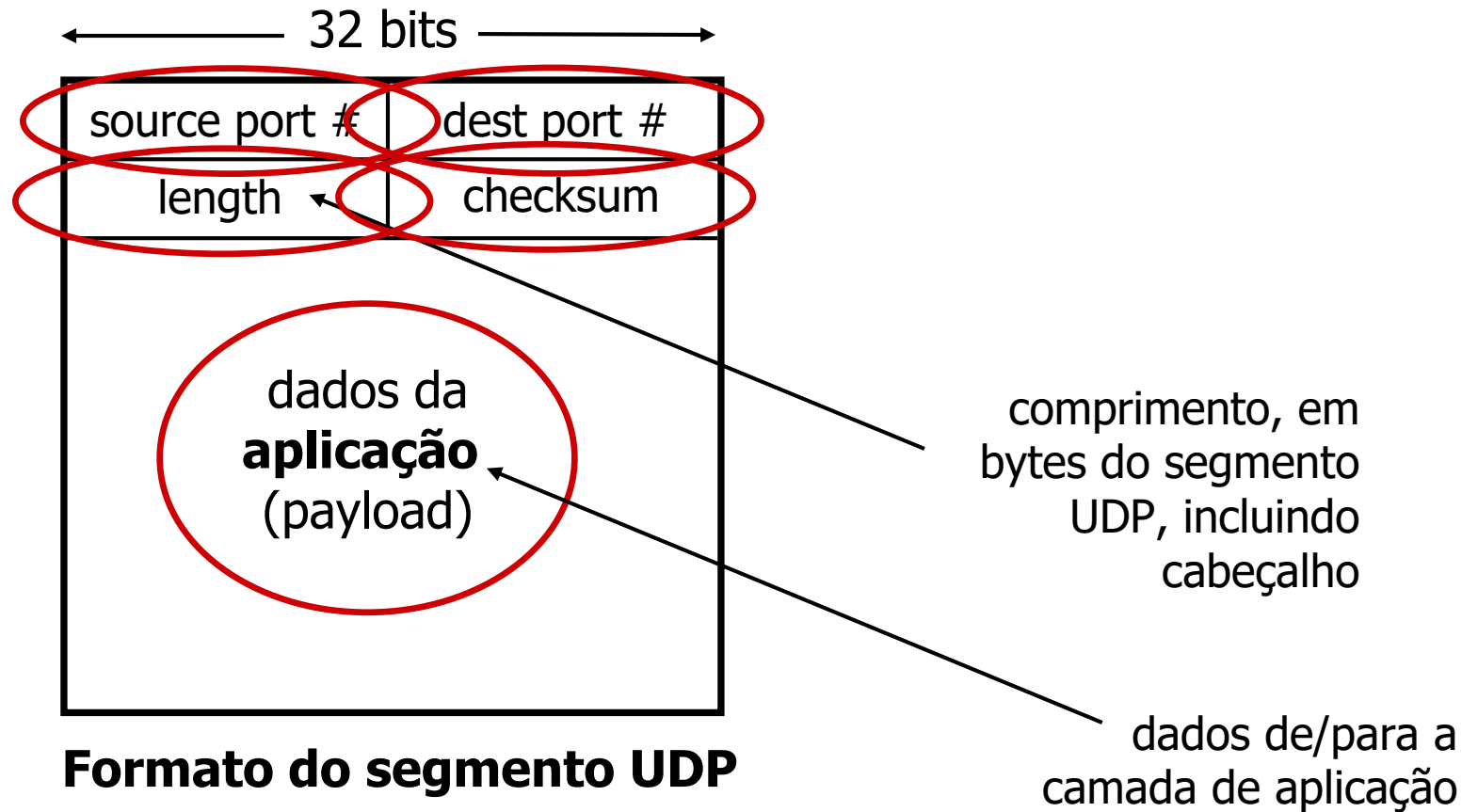


- recebe segmento de IP
- verifica o valor de *checksum* UDP
- Extrai a msg para a camada de aplicação
- demultiplexa msg até a aplicação via socket

servidor SNMP

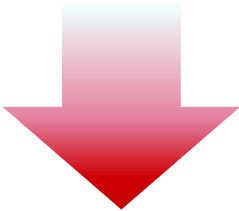





Cabeçalho do Segmento UDP



Checksum UDP

Objetivo: detectar erros (ex: bits invertidos) no segmento transmitido

	1º número	2º número	soma
Transmitido:	5	6	11
			
Recebido:	4	6	11
			
	soma calculada no receptor		soma calculada no transmissor (recebida)
	\neq		
			

Checksum na Internet

Objetivo: detectar erros (ex: bits invertidos) no segmento transmitido

Remetente:

- Trata o conteúdo do segmento UDP (incluindo campos de cabeçalho UDP e endereços IP) como sequência de inteiros de 16 bits
- **Checksum:** adição (soma do complemento) de conteúdo do segmento
- Valor da soma de verificação colocado no campo checksum UDP

Receptor:

- Calcula o checksum do segmento recebido
- Verifica se o checksum calculado é igual ao valor do campo de checksum:
- Diferente - erro detectado
- Igual - nenhum erro detectado. Mas talvez haja erros mesmo assim? Mais adiante

Checksum na Internet: um exemplo

Exemplo: somar 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/>																	
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
<hr/>																	
sum		1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum		0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

Nota: Ao adicionar números, um **carryout** do bit mais significativo precisa ser adicionado ao resultado

Checksum na Internet: proteção fraca!

Exemplo: somar 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
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1
sum	1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum	0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

Mesmo que os números tenham mudado (bit flips), nenhuma mudança na soma de verificação!

Resumo: UDP

- Protocolo "cru":
 - Segmentos podem ser perdidos, entregues fora de ordem
 - Melhor esforço de serviço: "enviar e esperar pelo melhor"
- UDP tem suas vantagens:
 - Nenhuma configuração/handshake necessário (sem RTT)
 - Pode funcionar quando o serviço de rede é comprometido
 - Ajuda com a confiabilidade (soma de verificação)
- Criar funcionalidade adicional sobre UDP na camada de aplicação (ex: HTTP/3)

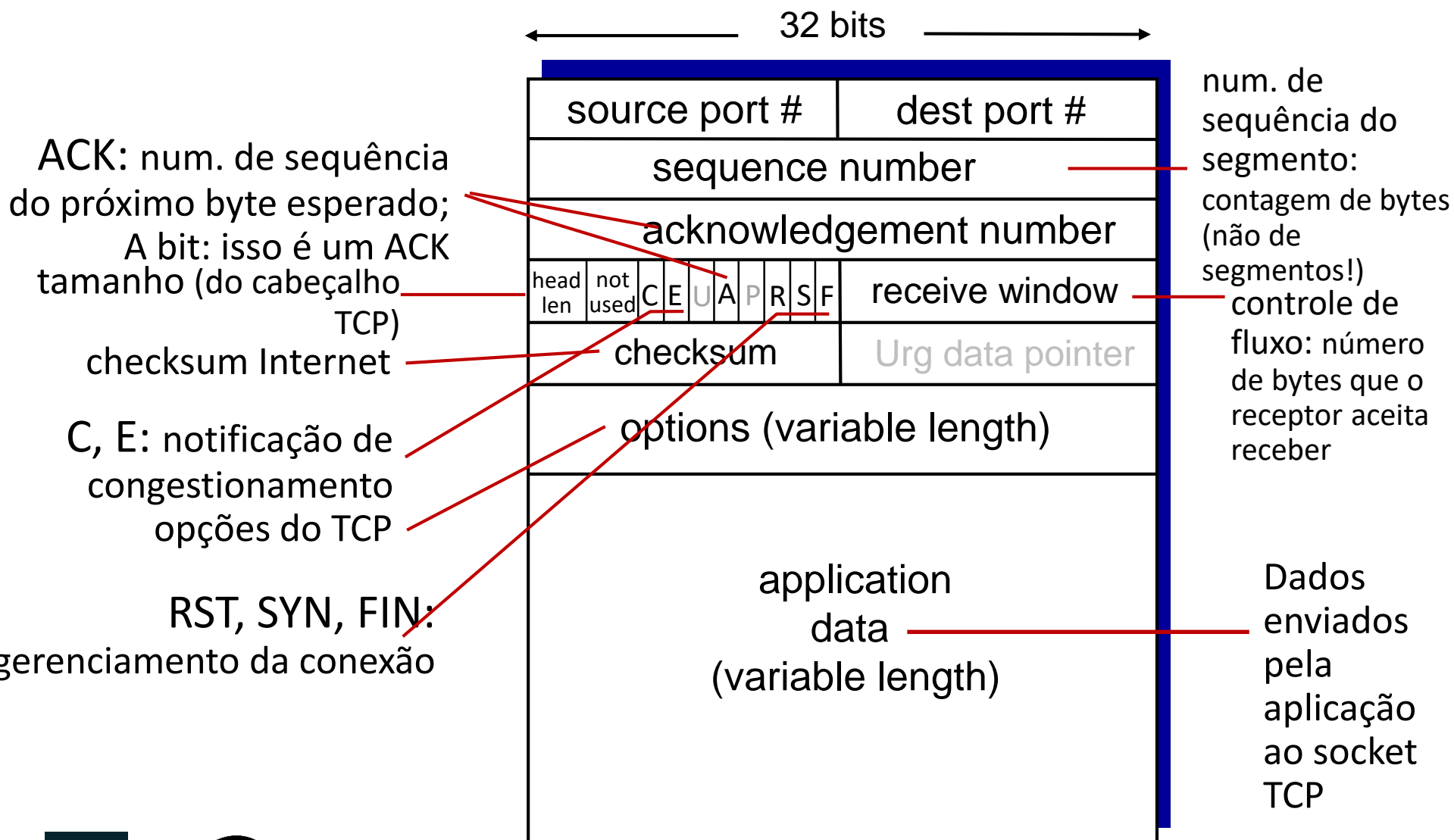
TCP: Visão Geral

- RFCs 793, 1122, 2018, 5681, 7323
- Ponto-a-ponto
 - Um emissor, um receptor
- Fluxo confiável de **bytes** ordenados
 - Sem delimitação por “mensagens”
- Dados full-duplex:
 - Fluxo de dados bidirecional na mesma conexão
 - MSS: Maximum Segment Size

TCP: Visão Geral

- ACKs cumulativos
- Pipelining:
 - Controle de fluxo e de congestionamento do TCP definem um tamanho de janela
- Orientado a conexão:
 - Handshaking (troca de mensagens de controle) inicializa estado do emissor e do receptor antes das trocas de mensagens
- Fluxo controlado:
 - O emissor não irá sobrecarregar o receptor

Estrutura do segmento TCP



Estrutura do segmento TCP

Números de sequência:

- “numeração” do primeiro byte contido no segmento de dados

Acknowledgements:

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

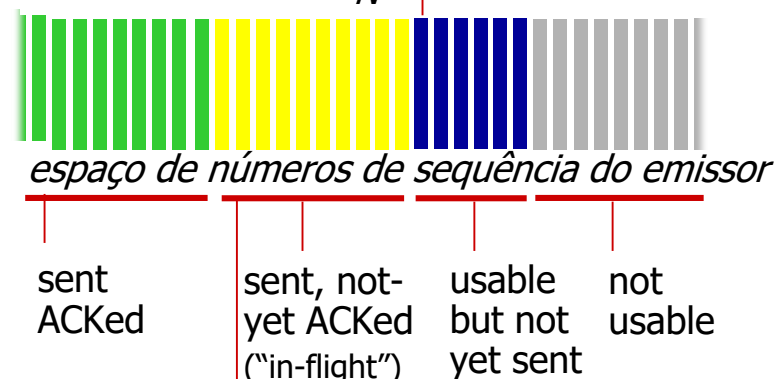
Q: como o receptor lida com segmentos fora de ordem

- A: A especificação do TCP não diz nada a respeito, - a critério do implementador

segmento que sai do emissor

source port #	dest port #
sequence number	
acknowledgement number	
	rwnd
checksum	urg pointer

tamanho da janela
 N



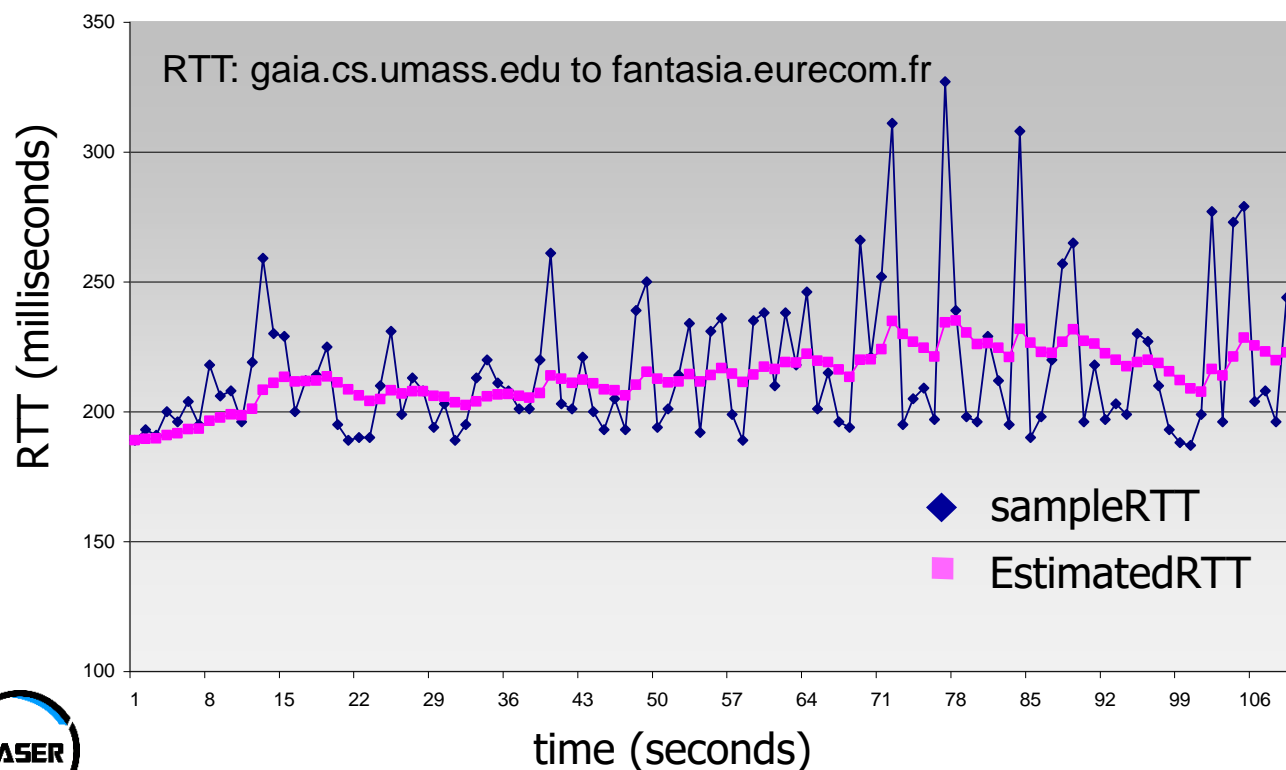
segmento que sai do receptor

source port #	dest port #
sequence number	
acknowledgement number	
	A rwnd
checksum	urg pointer

TCP round trip time (RTT), timeout

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

- exponential weighted moving average (EWMA)
- Influência de amostra anterior decresce exponencialmente rápido
- Valor típico: $\alpha = 0.125$



TCP round trip time (RTT), timeout

- Intervalo do timeout: **EstimatedRTT** mais “margem de segurança”
 - se for grande a variação em **EstimatedRTT**: iremos querer uma margem de segurança maior

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



RTT estimado

“margem de segurança”

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

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

(typically, $\beta = 0.25$)

Emissor TCP (simplificado)

evento: recebidos dados da aplicação

- criar segmento com número de sequência
- Número de sequência é o número do primeiro byte do fluxo de bytes que aparece no segmento
- Iniciar o temporizador caso ainda não esteja rodando
 - Considere o timer como sendo para o segmento mais velho “unACKed”
 - Intervalo de expiração: **TimeOutInterval**

evento: timeout

- retransmitir o segmento que causou o timeout
- reiniciar o timer

evento: ACK recebido

- se o ACK se referir a segmentos “unACKed”
 - atualizar o que se sabe que foi “ACKed”
 - iniciar timer se ainda houver segmentos “unACKed”

Receptor TCP: geração de ACK

Evento no receptor

Ação TCP no receptor

chegada de segmento em ordem com número de sequência esperado. Todos os dados até o número de sequência esperado já foram “ACKed”

ACK atrasado. Aguardar até 500ms pelo próximo segmento. Se não houver próximo segmento, enviar ACK,

chegada de segmento em ordem com número de sequência esperado. Um outro segmento possui ACK pendent

enviar imediatamente um ACK único cumulativo, reconhecendo ambos os segmentos em ordem

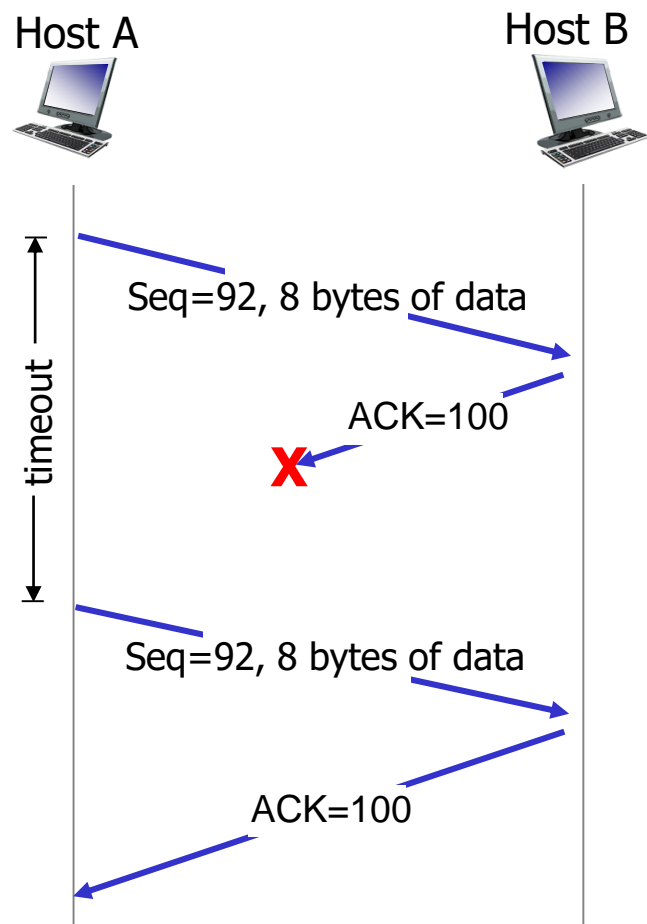
chegada de segmento fora de ordem com número de sequência maior do que o esperado. Lacuna detectada.

enviar imediatamente um *ACK duplicado*, indicando o número de sequência do próximo byte esperado

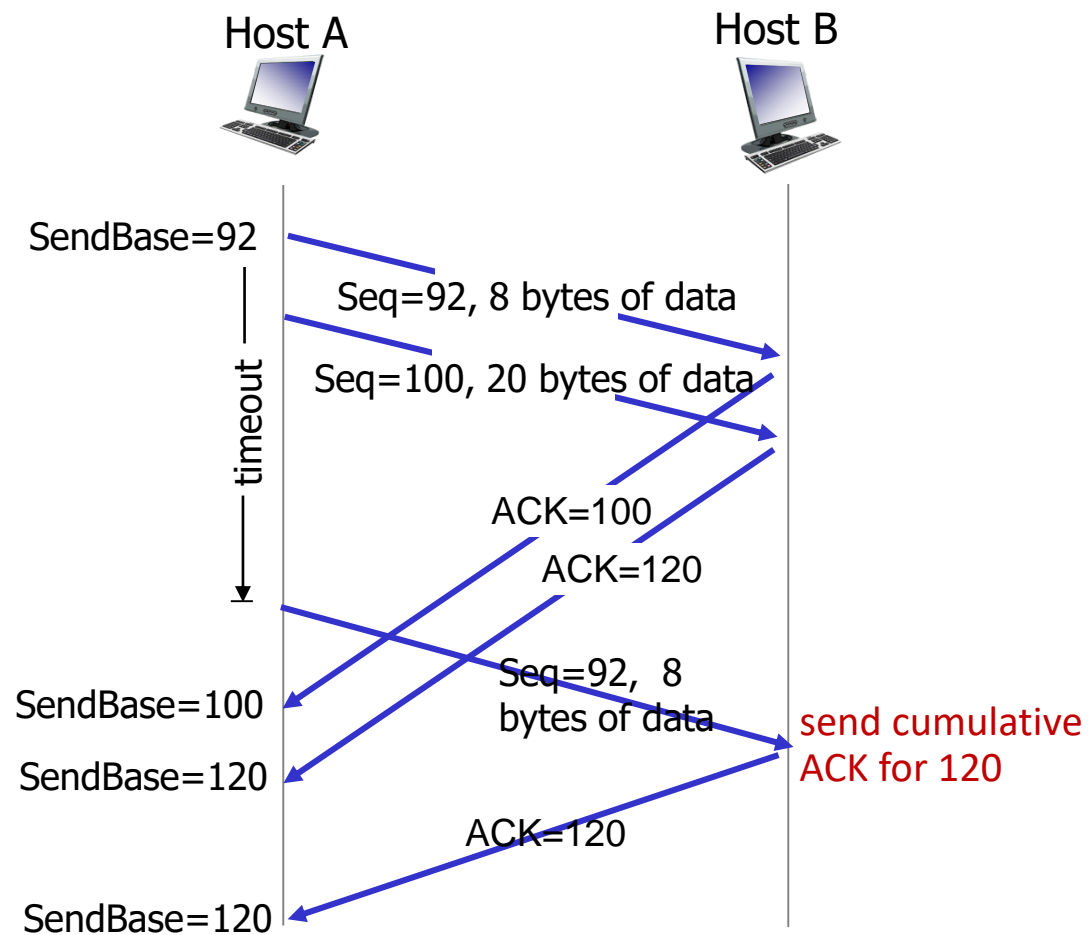
chegada de segmento que preenche uma lacuna de forma parcial ou de forma completa

enviar imediatamente ACK, considerando que o segmento começa na parte inferior da lacuna

TCP: cenários de retransmissão

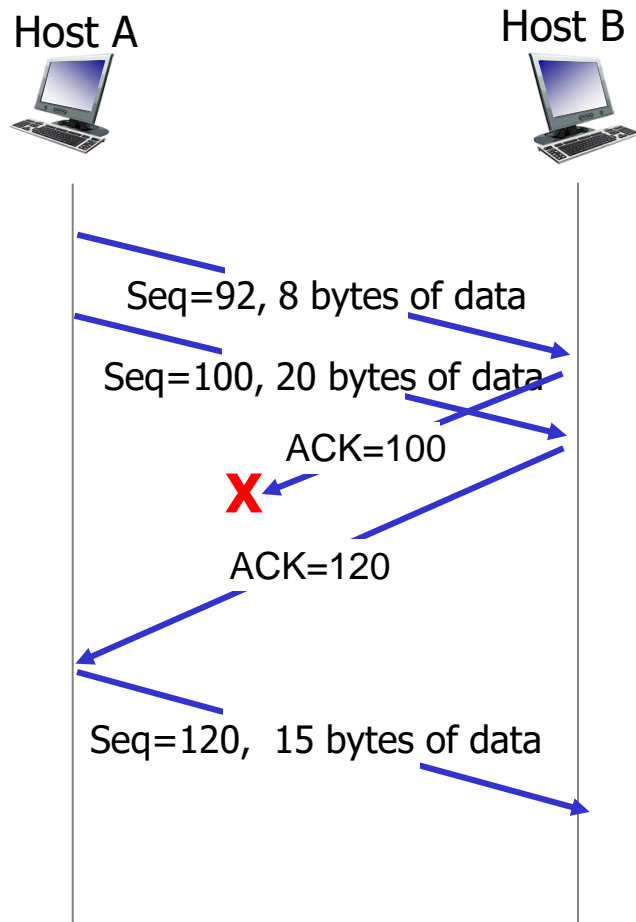


cenário de ACK perdido



Timeout prematuro

TCP: cenários de retransmissão



ACK cumulative abrange
ACK perdido anteriormente

Retransmissão rápida do TCP

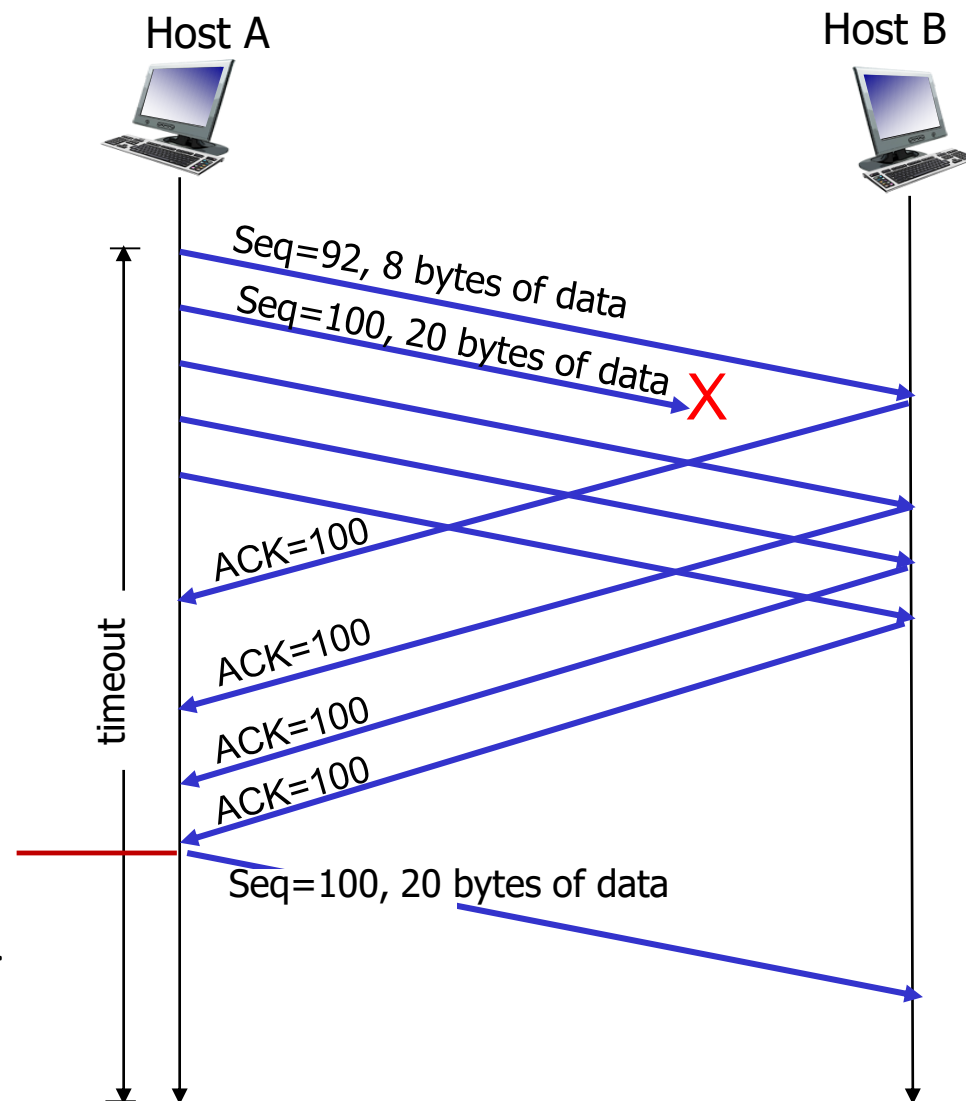
Retransmissão rápida do TCP

se o emissor receber 3 ACKs adicionais para o mesmo dado (“ACK duplicado triplo”), reenvia segmento “unACKed” com o menor número de sequência

- É provável que o segmento “unACKed” foi perdido, por isso não espere por um timeout

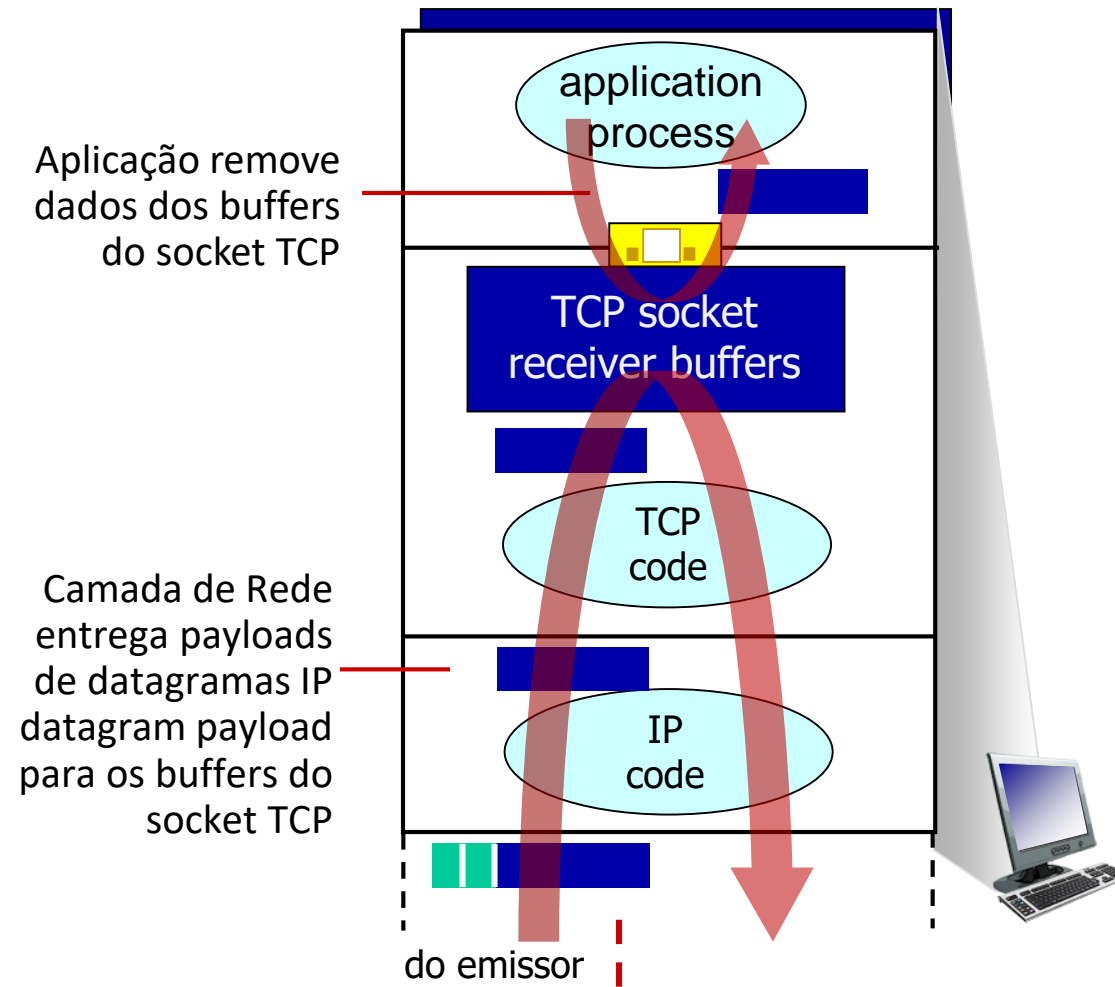


Recepção de ACKs duplicados indica 3 segmentos recebidos após um segmento perdido – provável perda de segmento. Então retransmita!



Controle de fluxo TCP

Pergunta: O que acontece se a Camada de Rede entrega dados mais rápido do que a Camada de Aplicação remove dados dos buffers do socket?



pilha de protocolo do receptor

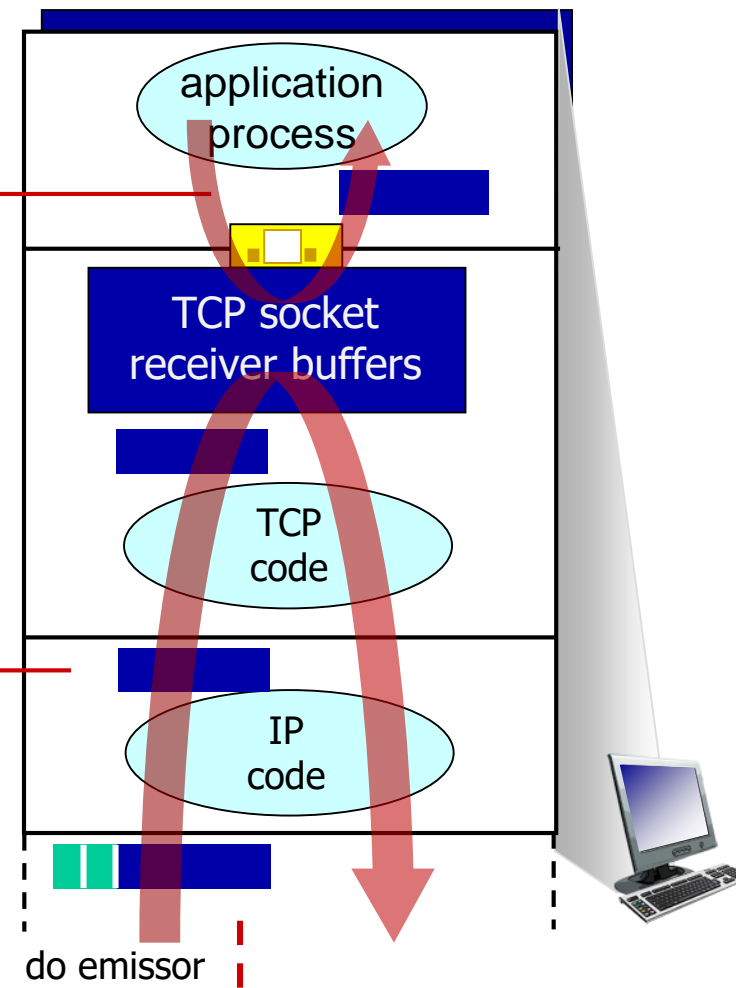
Controle de fluxo TCP

Pergunta: O que acontece se a Camada de Rede entrega dados mais rápido do que a Camada de Aplicação remove dados dos buffers do socket?



Aplicação remove dados dos buffers do socket TCP

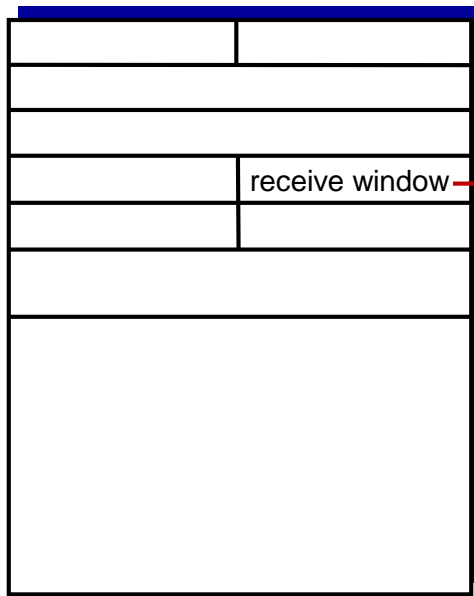
Camada de Rede entrega payloads de datagramas IP datagram payload para os buffers do socket TCP



pilha de protocolo do receptor

Controle de fluxo TCP

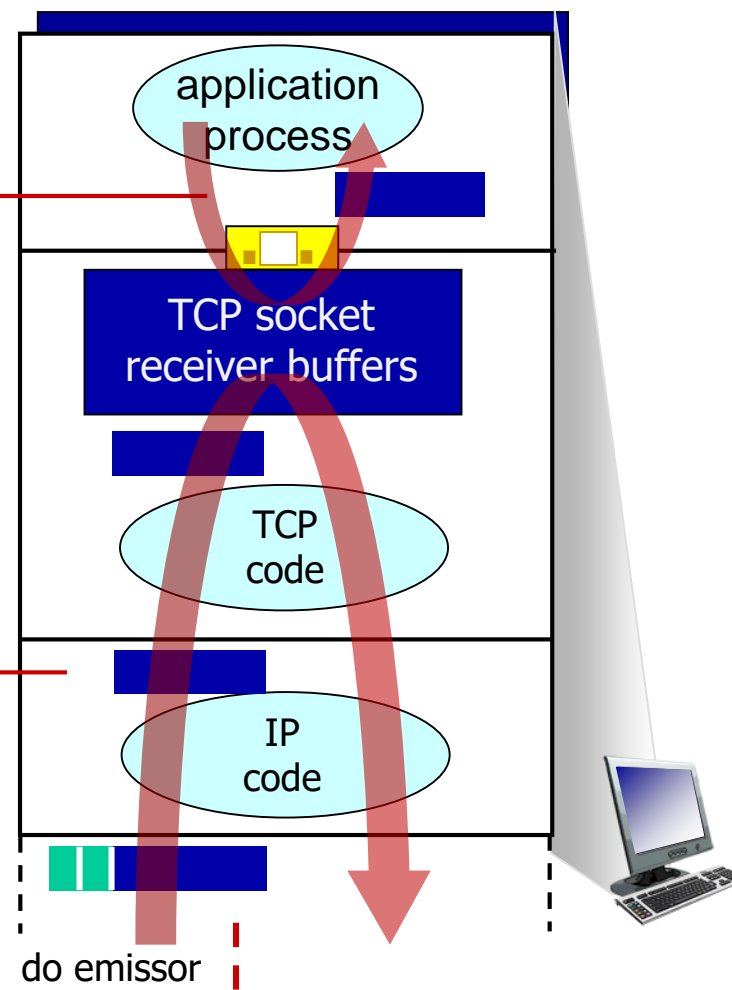
Pergunta: O que acontece se a Camada de Rede entrega dados mais rápido do que a Camada de Aplicação remove dados dos buffers do socket?



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

Aplicação remove dados dos buffers do socket TCP

Camada de Rede entrega payloads de datagramas IP datagram payload para os buffers do socket TCP



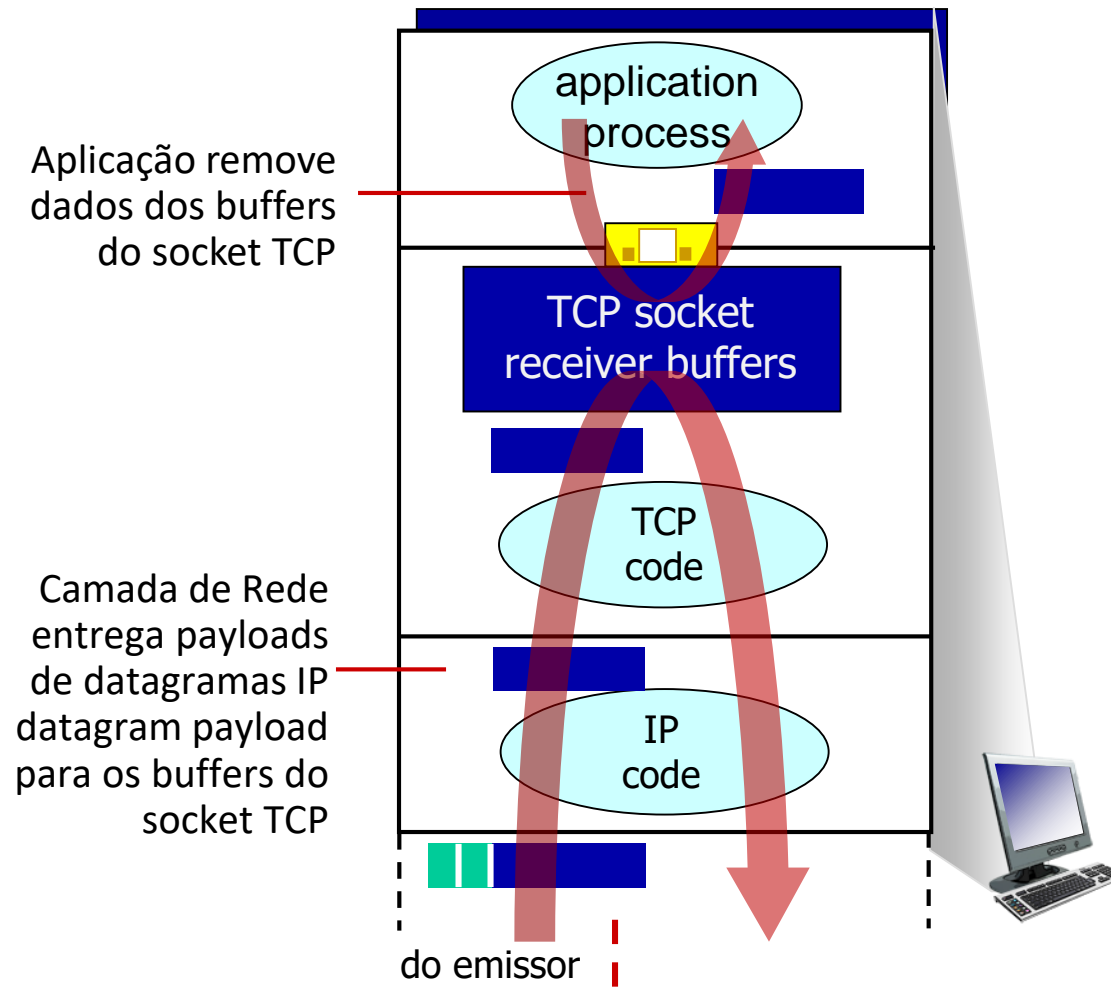
pilha de protocolo do receptor

Controle de fluxo TCP

Pergunta: O que acontece se a Camada de Rede entrega dados mais rápido do que a Camada de Aplicação remove dados dos buffers do socket?

controle de fluxo

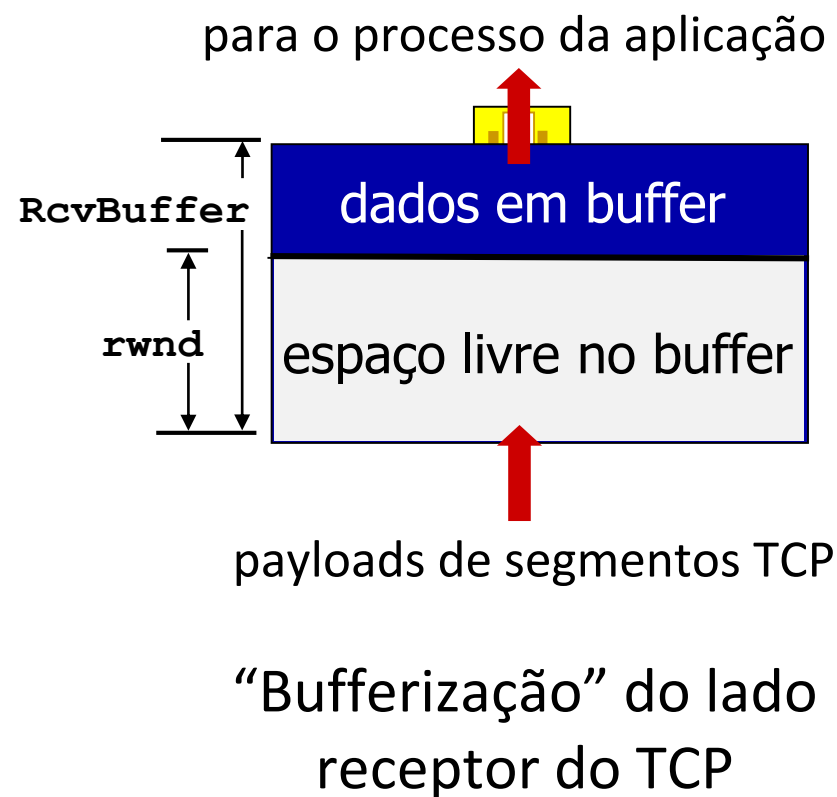
O receptor controla o emissor, de modo que o emissor não sobrecarregue o buffer do receptor ao transmitir muitos dados muito rapidamente



pilha de protocolo do receptor

Controle de fluxo TCP

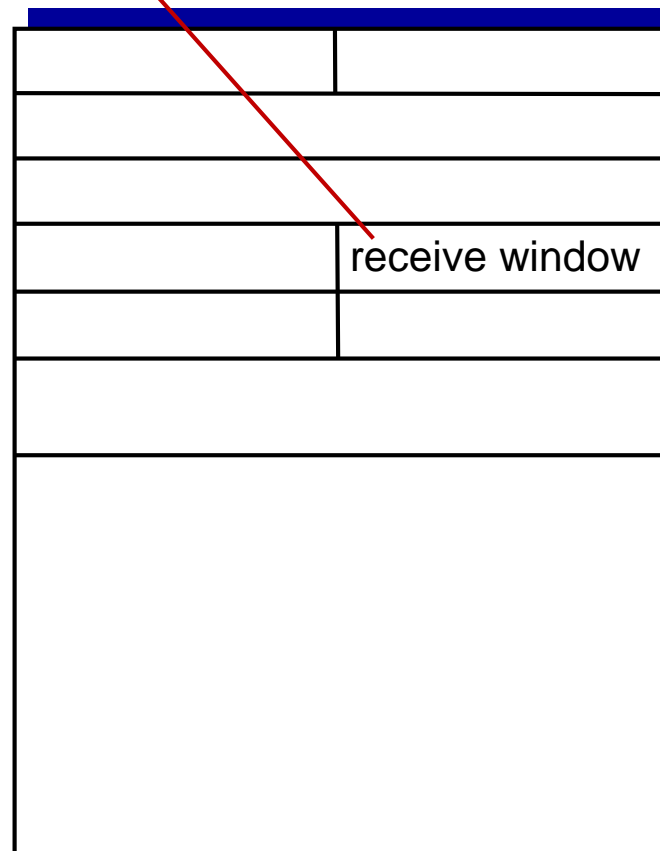
- Receptor TCP “anuncia” o espaço livre em buffer no campo **rwnd** do cabeçalho TCP
 - Tamanho de **RcvBuffer** é configurado através das opções de sockets (valor default típico é 4096 bytes)
 - Vários sistemas operacionais fazem ajustes automáticos em **RcvBuffer**
- Emissor limita a quantidade de dados unACKed ao valor de **rwnd** recebido
- Garante que o buffer do receptor não vai transbordar



Controle de fluxo TCP

- Receptor TCP “anuncia” o espaço livre em buffer no campo **rwnd** do cabeçalho TCP
 - Tamanho de **RcvBuffer** é configurado através das opções de sockets (valor default típico é 4096 bytes)
 - Vários sistemas operacionais fazem ajustes automáticos em **RcvBuffer**
- Emissor limita a quantidade de dados unACKed ao valor de **rwnd** recebido
- Garante que o buffer do receptor não vai transbordar

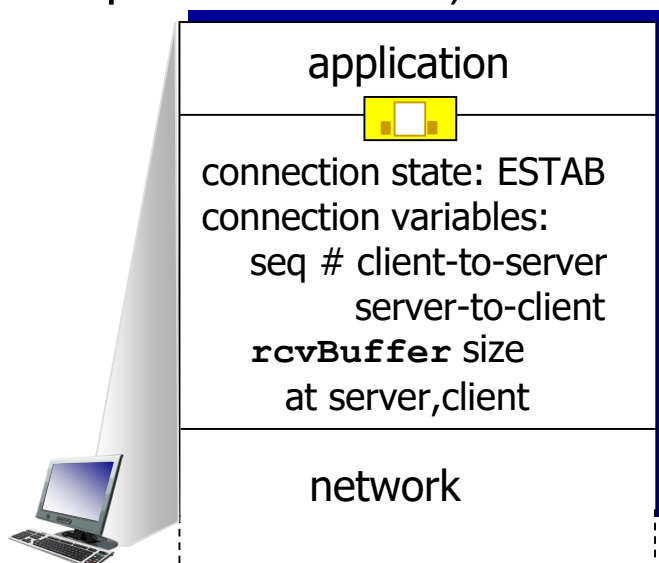
controle de fluxo: número de bytes que o receptor está disposto a aceitar



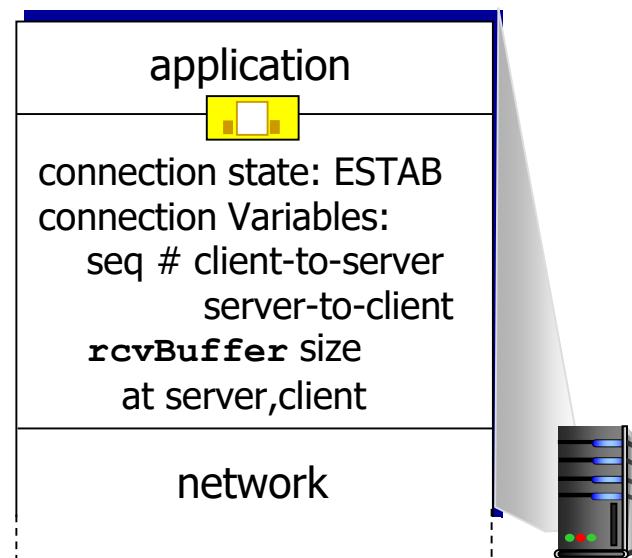
formato de segmento TCP

Gerenciamento de conexão TCP

- Antes de trocar dados, emissor/receptor fazem o “aperto de mãos” (“handshake”)
 - Concordam em estabelecer a conexão (cada um ciente que o outro deseja estabelecer a conexão)
 - Concordam com parâmetros da conexão (e.g., números de sequência iniciais)



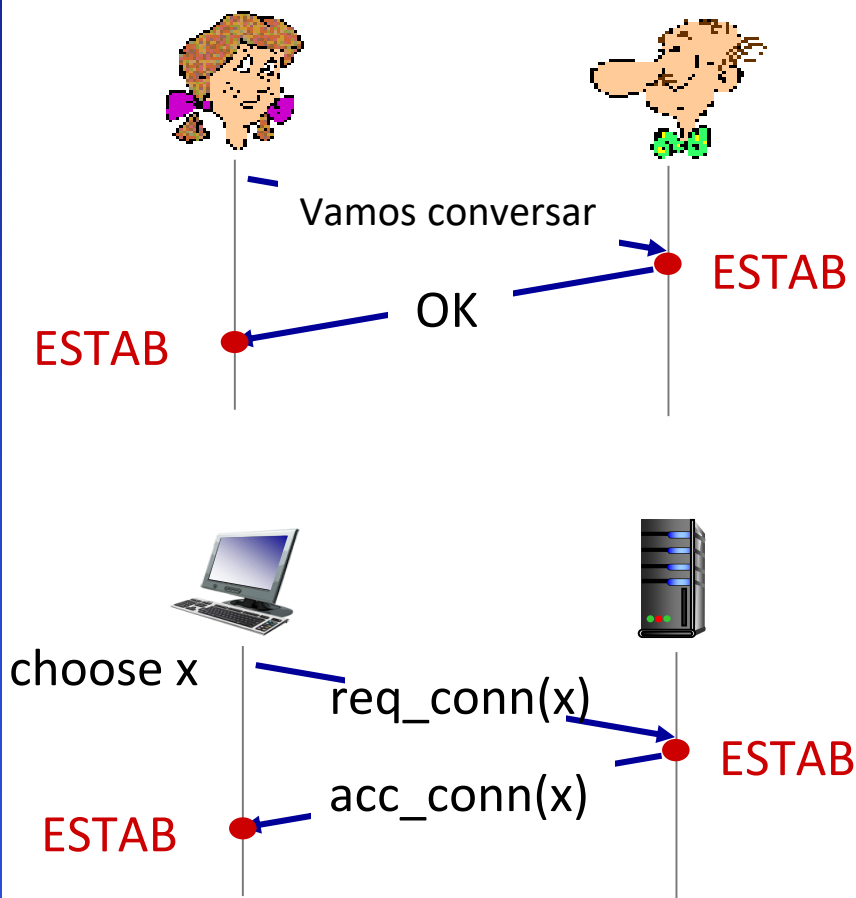
```
Socket clientSocket =  
newSocket("hostname", "port number");
```



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

Concordando em estabelecer conexão

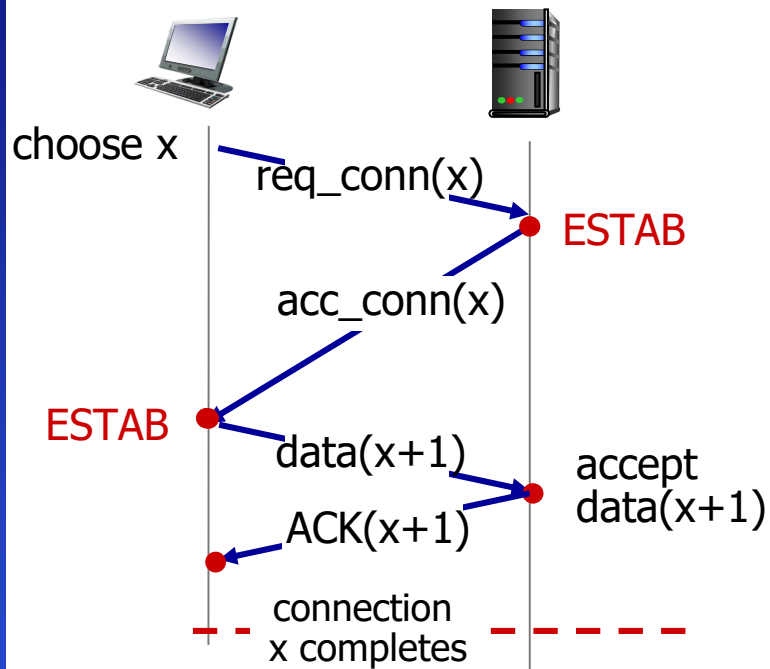
2-way handshake:



Pergunta: 2-way handshake irá sempre funcionar em redes?

- atrasos variáveis
- Mensagens retransmitidas (e.g. req_conn(x)) devido a perda de mensagens
- reordenação de mensagens
- não pode “enxergar” o outro lado

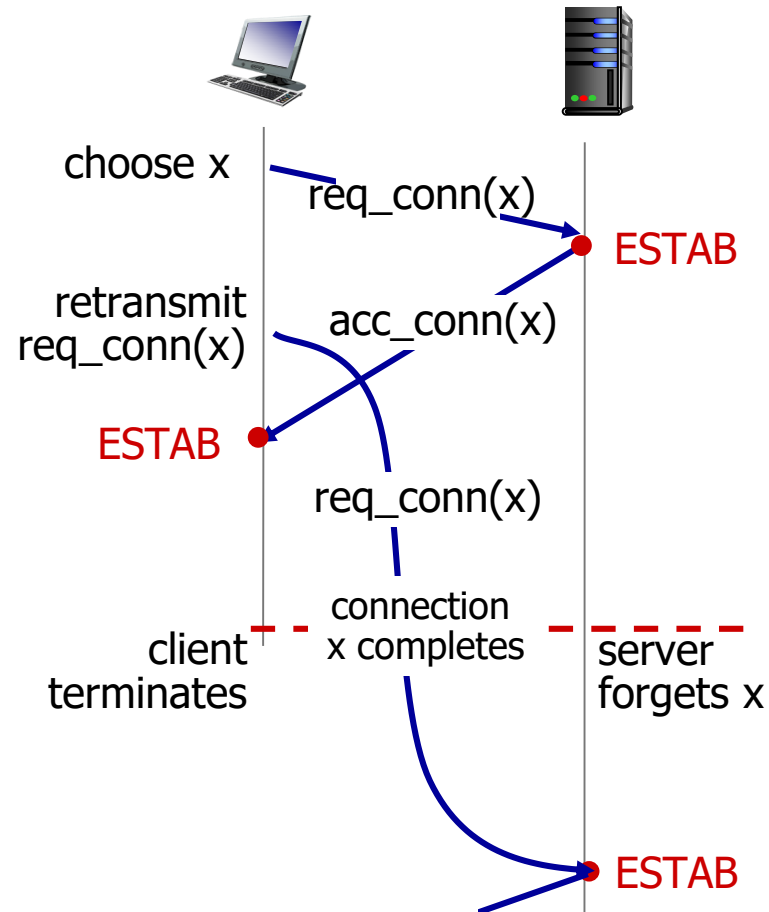
Cenários de 2-way handshake




Sem problema!

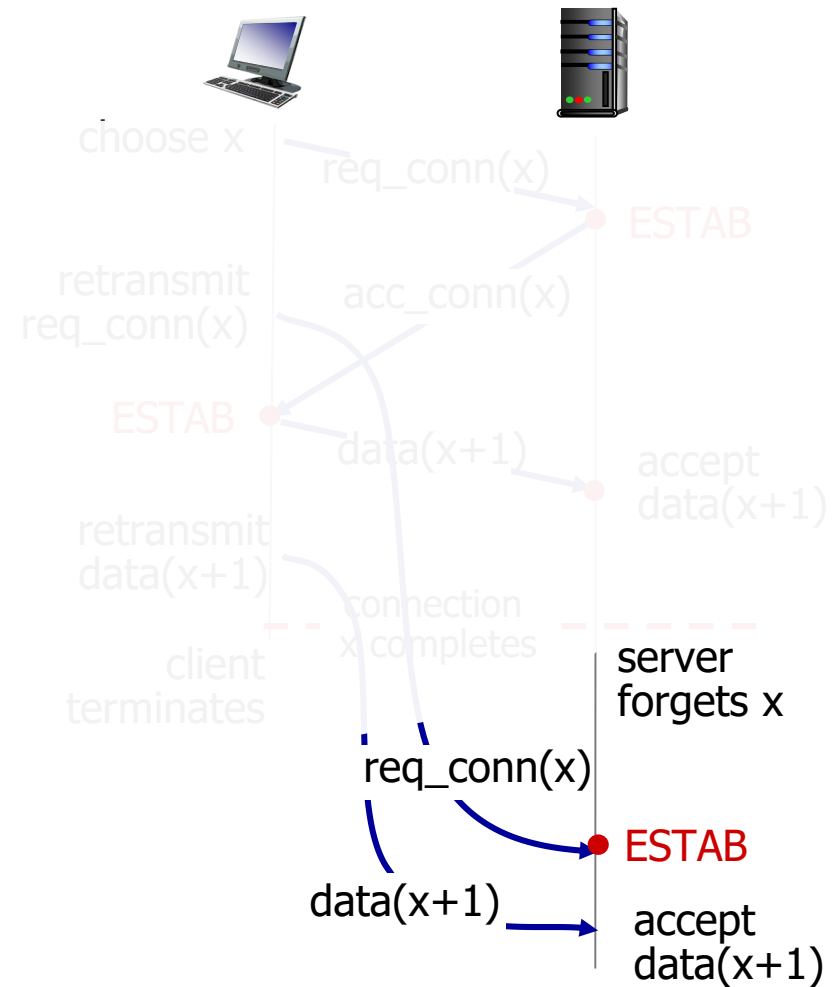


Cenários de 2-way handshake



 Problema: conexão aberta pela metade! (sem cliente)

Cenários de 2-way handshake



Problema: dados duplicados aceitos!

3-way handshake do TCP

Estado do Cliente

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

LISTEN

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

SYNSENT

ESTAB

choose init seq num, x
send TCP SYN msg

received SYNACK(x)
indicates server is live;
send ACK for SYNACK;
this segment may contain
client-to-server data

SYNbit=1, Seq=x

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

ACKbit=1, ACKnum=y+1

Estado do Servidor

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

LISTEN

SYN RCVD

ESTAB

choose init seq num, y
send TCP SYNACK
msg, acking SYN

received ACK(y)
indicates client is live

3-way handshake do TCP



Encerrando uma conexão TCP

- Cliente e servidor fecham seus respectivos lados da conexão
 - Envio de segmento TCP com o bit $FIN = 1$
- Cada um responde ao FIN recebido com um ACK
 - Ao receber FIN, ACK pode ser combinado com o próprio FIN
- Troca simultânea de FIN pode ocorrer

Encerrando uma conexão TCP

- Além do encerramento “gracioso” da conexão TCP, também existem casos em que a conexão precisa encerrar imediatamente
 - Falta de comunicação (segmento retransmitido cinco vezes)
 - Erros de sistema ou de protocolo (e.g., receber pacote de um host para o qual não existe conexão)
- Nesses casos, é encaminhado uma mensagem com a flag RST = 1
 - O outro lado responde com um ACK para essa flag RST