PTC 3450 - Aula 15

3.5 Transporte orientado para conexão: TCP

(Kurose, p. 177 - 190)
(Peterson, p. 105-124 e 242-264)

23/05/2017

Capítulo 3: conteúdo

3.1 serviços da camada de transporte

3.2 multiplexação e desmultiplexação

3.3 transporte sem conexão: UDP

3.4 princípios da transferência de dados confiável

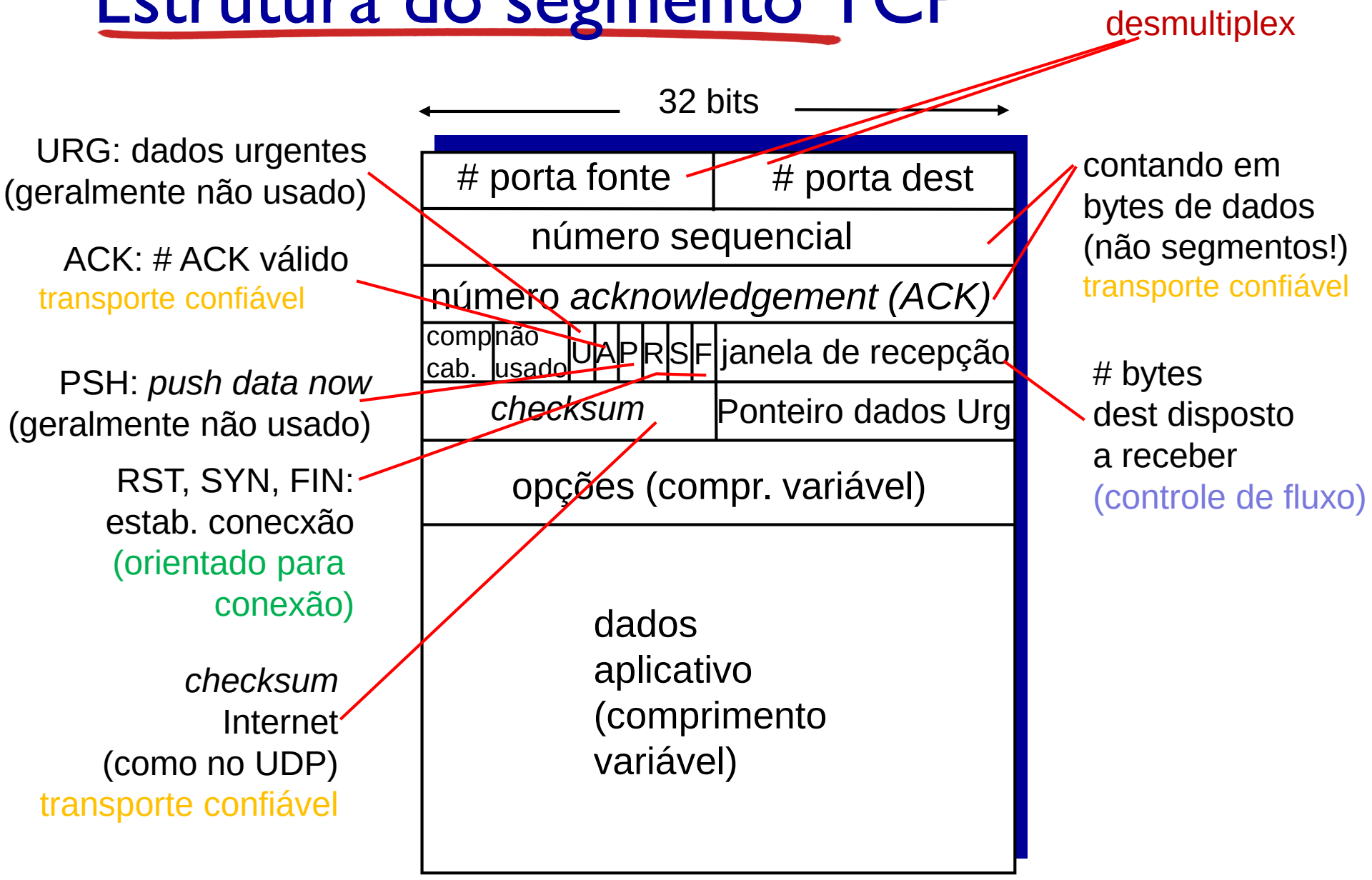
3.5 transporte orientado para conexão: TCP

- estrutura dos segmentos
- transferência de dados confiável
- controle de fluxo
- gerenciamento de conexão

3.6 princípios do controle de congestionamento

3.7 controle de congestionamento no TCP

Estrutura do segmento TCP



TCP: transferência de dados confiável

- TCP cria serviço `rdt` sobre serviço não confiável do IP
  - segmentos paralelizados (*pipelined*)
  - *acks* acumulativos
  - *único* temporizador de retransmissão
  - *(parece GBN)*
- retransmissões disparadas por:
  - eventos *timeout*
  - *acks* duplicados

vamos inicialmente considerar remetente TCP simplificado:

- ignorar *acks* duplicados
- ignorar controle de fluxo e congestionamento
- Dados enviados apenas em 1 direção
- Remetente enviando arquivo grande

Eventos no remetente TCP

dados recebidos da aplicação:

- cria segmento com # seq
- # seq é número na cadeia de bytes do primeiro byte de dado no segmento
- inicializa temporizador se já não está rodando
  - temporizador para o segmento mais antigo *unacked*
  - intervalo de expiração: `TimeOutInterval`

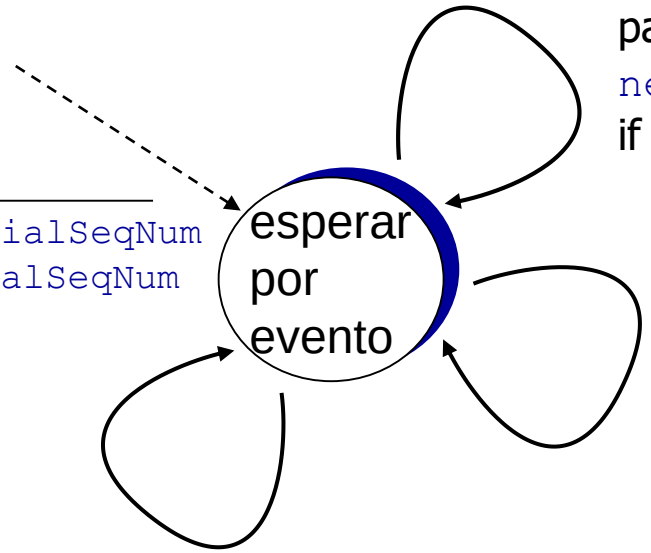
timeout:

- retransmite *apenas* segmento que causou *timeout (parece SR)*
- reinicializa temporizador

ack recebido (acumulativo):

- se *ack* reconhece segmentos previamente *unacked*
  - atualizar o que sabe-se ACKed
  - reiniciar temporizador se ainda existem segmentos *unacked*

Remetente TCP (simplificado)



Λ

nextseqnum = InitialSeqNum
send_base = InitialSeqNum

dados recebidos de aplicativo acima

cria segmento, # seq. : nextseqnum
passa segmento para IP (i.e., "envia")
nextseqnum = nextseqnum + length(data)
if (temporizador não rodando)
    inicializa temporizador

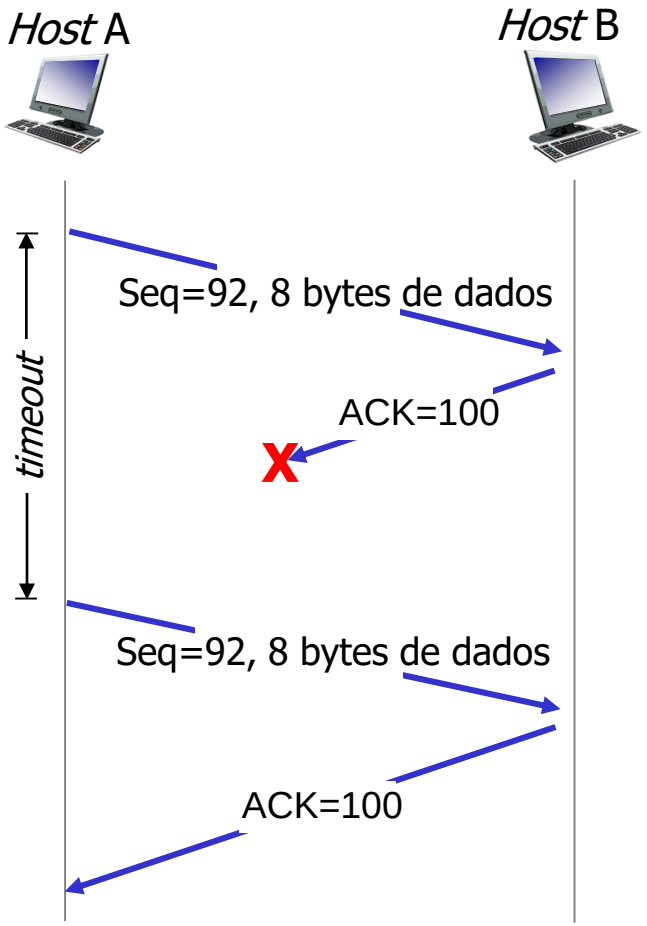
timeout

retransmite segmento ainda não ACKed com menor # seq.
inicializa temporizador

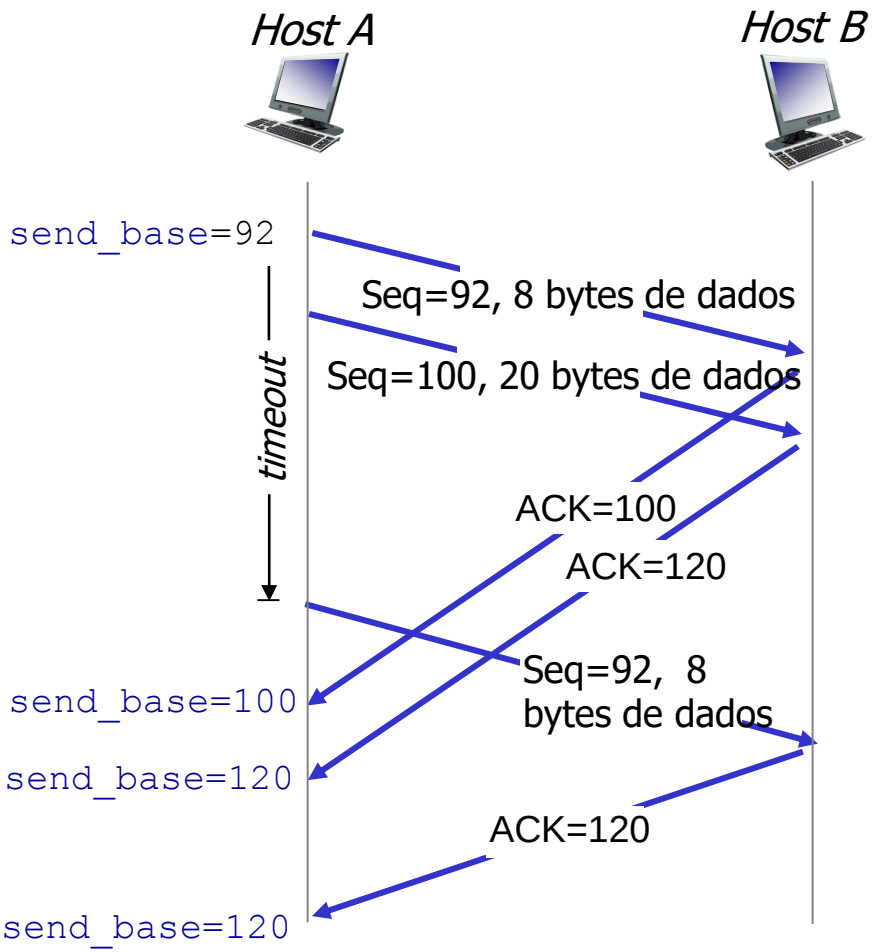
ACK recebido, com valor de campo ACK y

```
if (y > send_base) {
 send_base = y
   /* send_base–1: último byte acumulativamente ACKed */
   if (existem segmentos ainda não ACKed)
       inicializa temporizador
     else para temporizador
   }
```

TCP: cenários de retransmissão

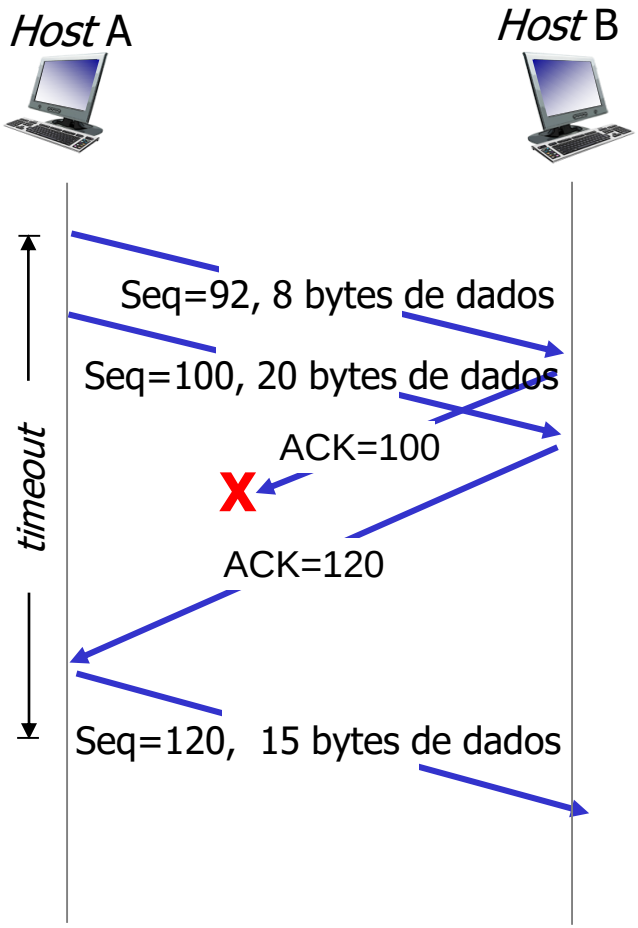


cenário ACK perdido



cenário *timeout* prematuro

TCP: cenários de retransmissão



cenário ACK cumulativo

Interessante: mesmo com ACK perdido não foi necessário retransmissão.

Ideias para modificações que melhoram nosso TCP simplificado?

Objetivo: economizar ACKs e reenvios

- ACK retardado
- Retransmissão rápida
- Duplicar `TimeOutInterval`

TCP: ACK retardado [Última versão RFC 5681 (2009)]

evento no destinatário	*ação do TCP do destinatário*
chegada de segmento em ordem com # seq esperado. Todos os dados até # seq esperado já *ACKed*	*ACK retardado*. Espera até 500 ms pelo próximo segmento. Se não chega o próximo segmento, envia ACK
chegada de segmento em ordem com # seq esperado. Outro segmento esperando por transmissão de *ACK*	imediatamente envia ACK acumulativo único, *validando* ambos segmentos em ordem
chegada de segmento fora de ordem com # seq maior do que o esperado. Lacuna detectada	imediatamente envia *ACK duplicado*, indicando # seq. do próximo byte esperado
chegada de segmento que preenche a lacuna parcialmente ou completamente	imediatamente envia *ACK*, desde que segmento inicie na extremidade inferior da lacuna

TCP: retransmissão rápida

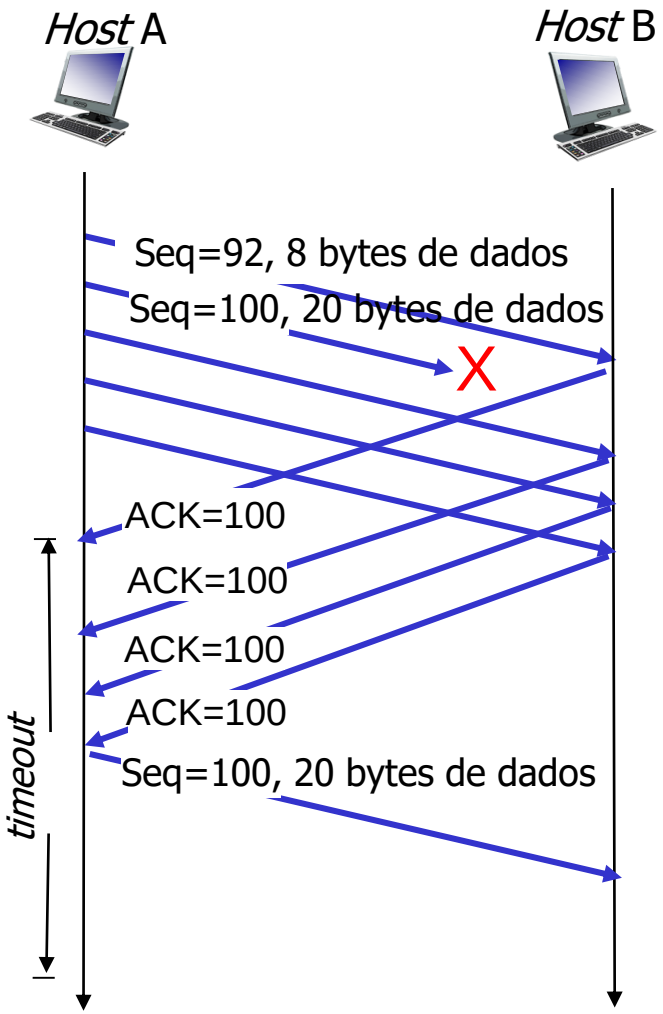
- período *timeout* muitas vezes é relativamente longo:
  - longo atraso antes de reenviar pacote perdido
- detectar segmentos perdidos via ACKs duplicados
  - remetente usualmente envia muitos segmentos
  - se segmento é perdido, provavelmente haverão muitos ACKs duplicados.

Retransmissão rápida TCP

se remetente recebe 3 *ACK*s para mesmos dados ("triplo *ACK* duplicado"), reenvia segmento *unacked* com menor # seq

- provavelmente aquele segmento *unacked* está perdidos, assim não esperar por *timeout*

TCP: retransmissão rápida



retransmissão rápida depois que remetente recebeu triplo ACK duplicado

TCP: duplicando o `TimeOutInterval`

- Quando ocorre *timeout*, TCP retransmite segmento não-ack com número sequencial mais baixo (como antes)

- Mas faz

  `TimeOutInterval(t)= 2*TimeOutInterval(t-1)`

ao invés de usar **`EstimatedRTT`**

  - Aumento exponencial de `TimeOutInterval` a cada *timeout* seguido

- Ideia: se houve *timeout* é porque a rede deve estar congestionada – melhor esperar mais para evitar que novos pacotes se percam
  - Controle de congestionamento simples

- Assim que recebe ACK válido, volta a usar **`EstimatedRTT`** para calcular `TimeOutInterval`

TCP: transferência de dados confiável

- Discussão: TCP é GBN ou SR?

- Resposta: Versão híbrida
  - Um só temporizador e ACK cumulativo como GBN
  - Retransmissão apenas de pacote não-ACK como SR
  - Mais uma série de "truques" aprendidos ao longo de mais de 40 anos de pesquisas e experiências...

Capítulo 3: conteúdo

3.1 serviços da camada de transporte

3.2 multiplexação e desmultiplexação

3.3 transporte sem conexão: UDP

3.4 princípios da transferência de dados confiável

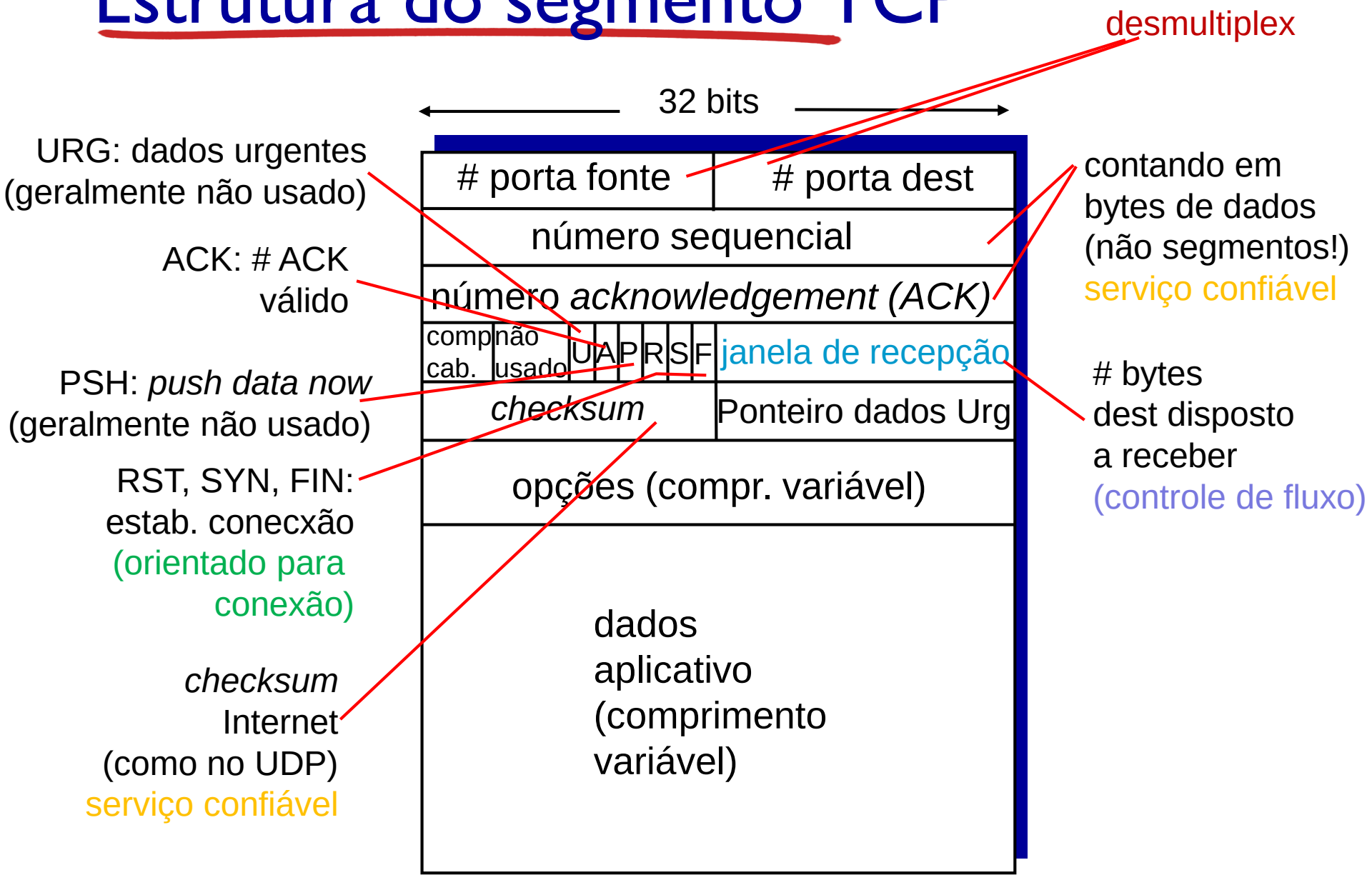
3.5 transporte orientado para conexão: TCP

- estrutura dos segmentos
- transferência de dados confiável
- controle de fluxo
- gerenciamento de conexão

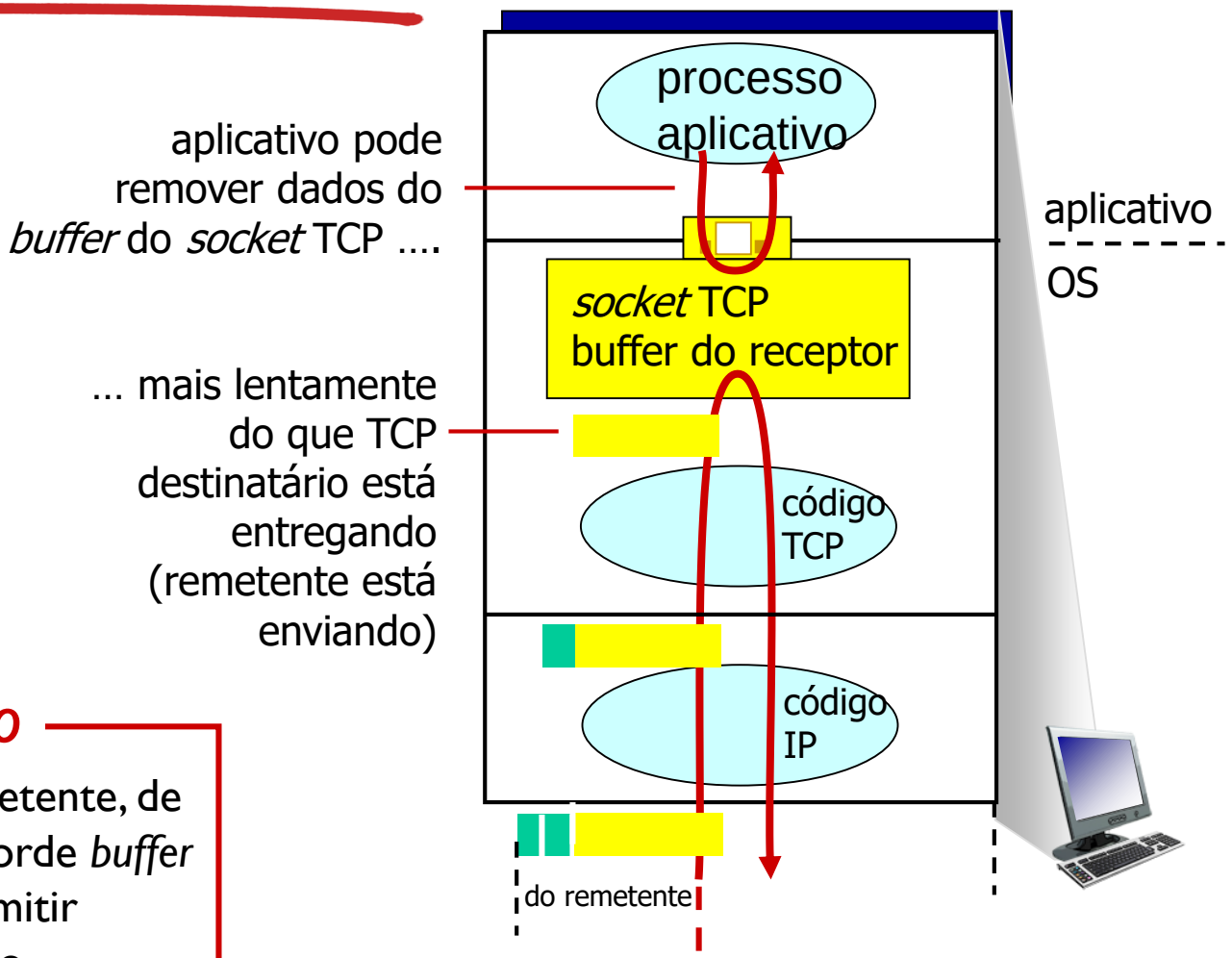
3.6 princípios do controle de congestionamento

3.7 controle de congestionamento no TCP

Estrutura do segmento TCP



TCP: controle de fluxo



pilha de protocolos do destinatário

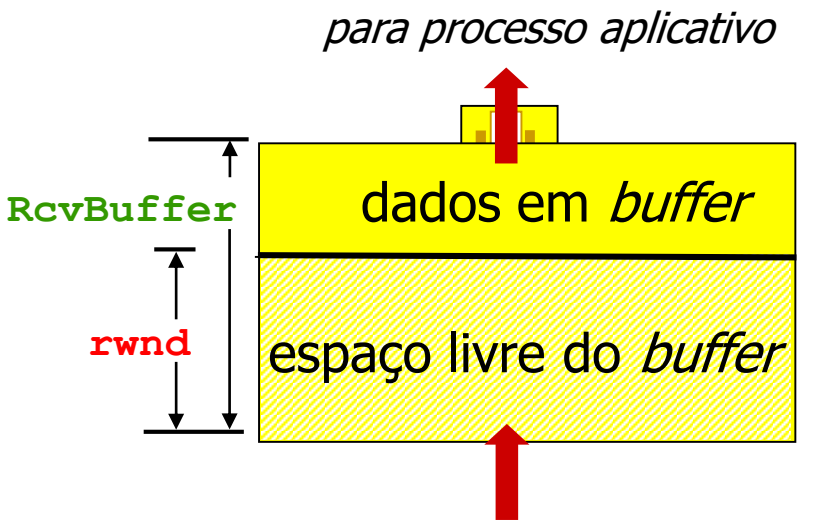
controle de fluxo

destinatário controla remetente, de forma que ele não transborde *buffer* do destinatário por transmitir muitos dados muito rápido

Veja: http://media.pearsoncmg.com/aw/aw_kurose_network_4/applets/flow/FlowControl.htm

TCP: controle de fluxo

- destinatário "informa" espaço de *buffer* livre incluindo o valor **`rwnd`** no campo "janela de recepção" do cabeçalho TCP dos segmentos destinatário-para-remetente
  - **`RcvBuffer`** determinado via opção na criação do *socket* (default típico é 4096 bytes)
  - muitos sistemas operacionais autoajustam **`RcvBuffer`**
- remente limita quantidade de dados *unacked* ("*in-flight*") ao valor **`rwnd`** do destinatário
- garante que *buffer do destinatário não* transborda



buffer do lado destinatário

- *Lembrando que TCP é full-duplex, ambos os lados terão buffer de recepção...*
- *OBS: cuidado com* **`rwnd = 0`**

Capítulo 3: conteúdo

3.1 serviços da camada de transporte

3.2 multiplexação e desmultiplexação

3.3 transporte sem conexão: UDP

3.4 princípios da transferência de dados confiável

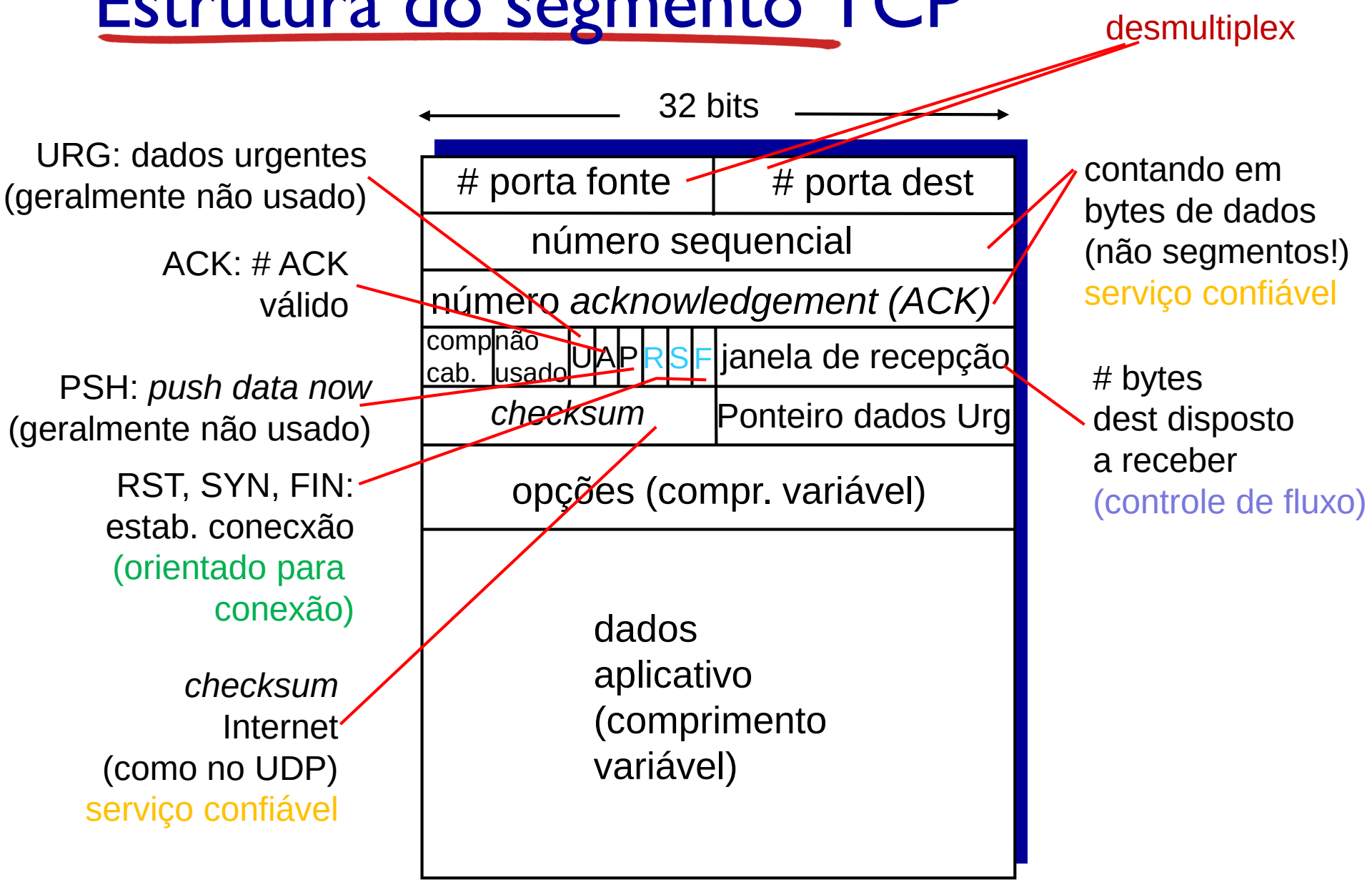
3.5 transporte orientado para conexão: TCP

- estrutura dos segmentos
- transferência de dados confiável
- controle de fluxo
- gerenciamento de conexão

3.6 princípios do controle de congestionamento

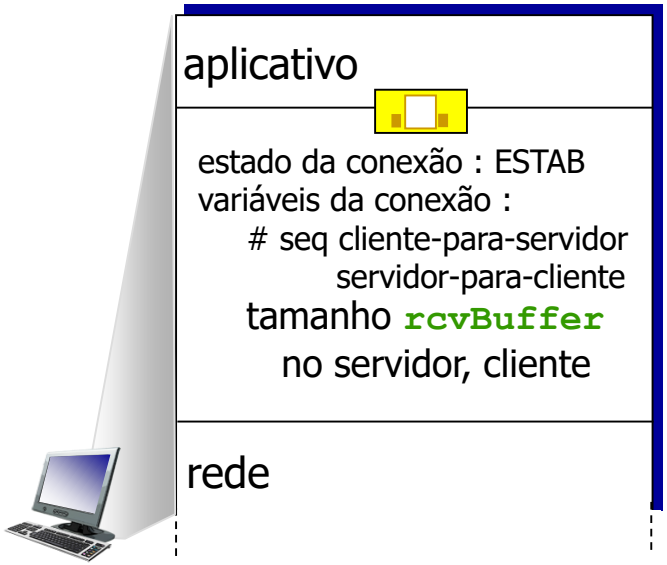
3.7 controle de congestionamento no TCP

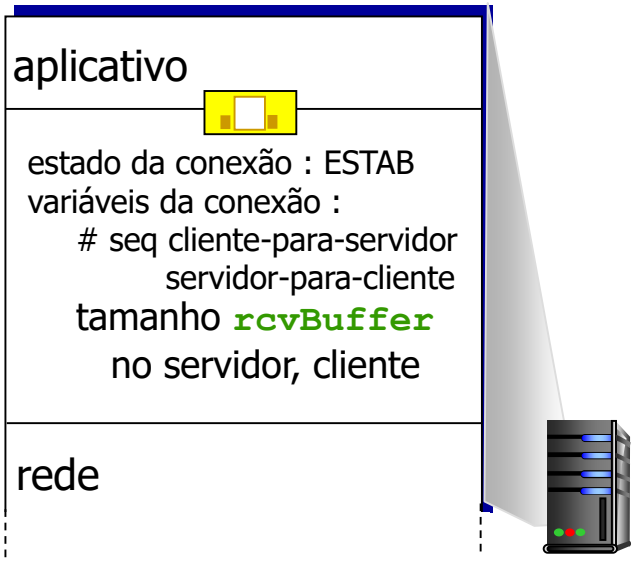
Estrutura do segmento TCP



Gerenciamento de Conexão

- Papel significativo nos atrasos (*web*) e ataques DDoS *(SYN flood)*
- antes de trocar dados, remetente/destinatário fazem *handshake*:
  - concordam em estabelecer conexão (cada um toma conhecimento do outro)
  - concordam com parâmetros da conexão



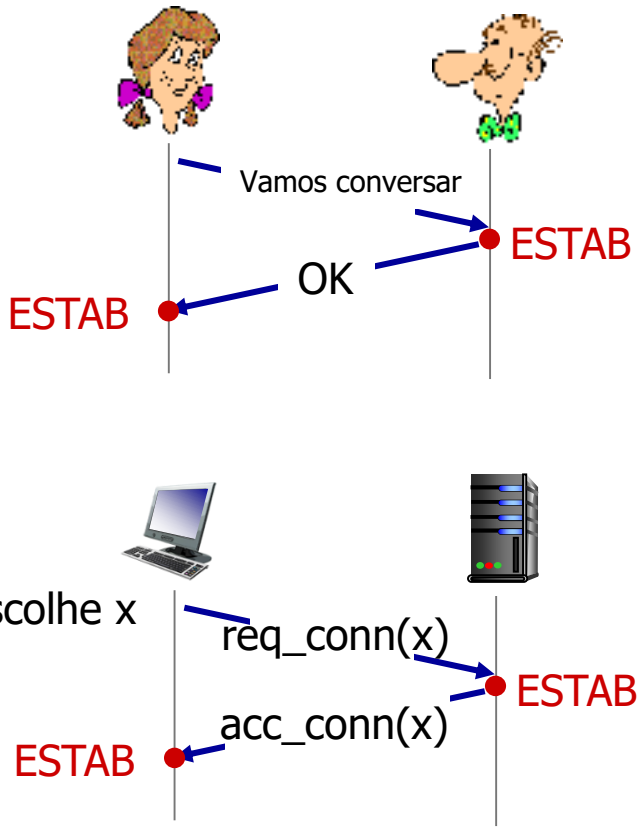



```
clienteSocket.connect((Nomeservidor, Portaservidor))
```

```
conexaoSocket, addr = servidorSocket.accept()
```

Concordando em estabelecer uma conexão

handshake 2 vias:

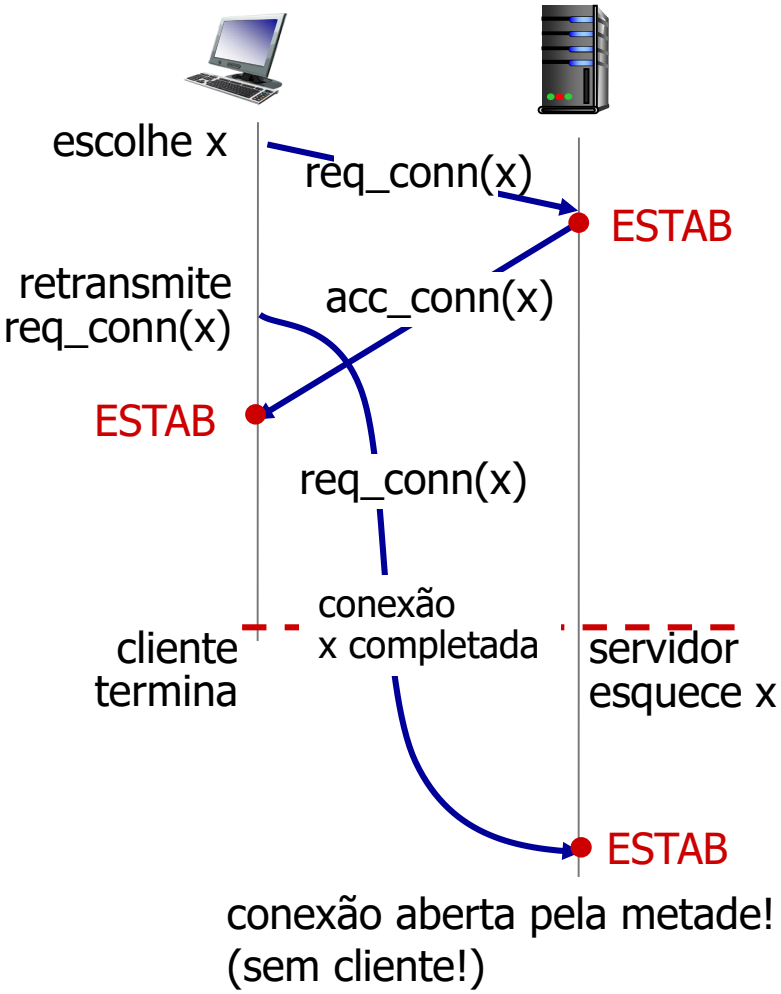


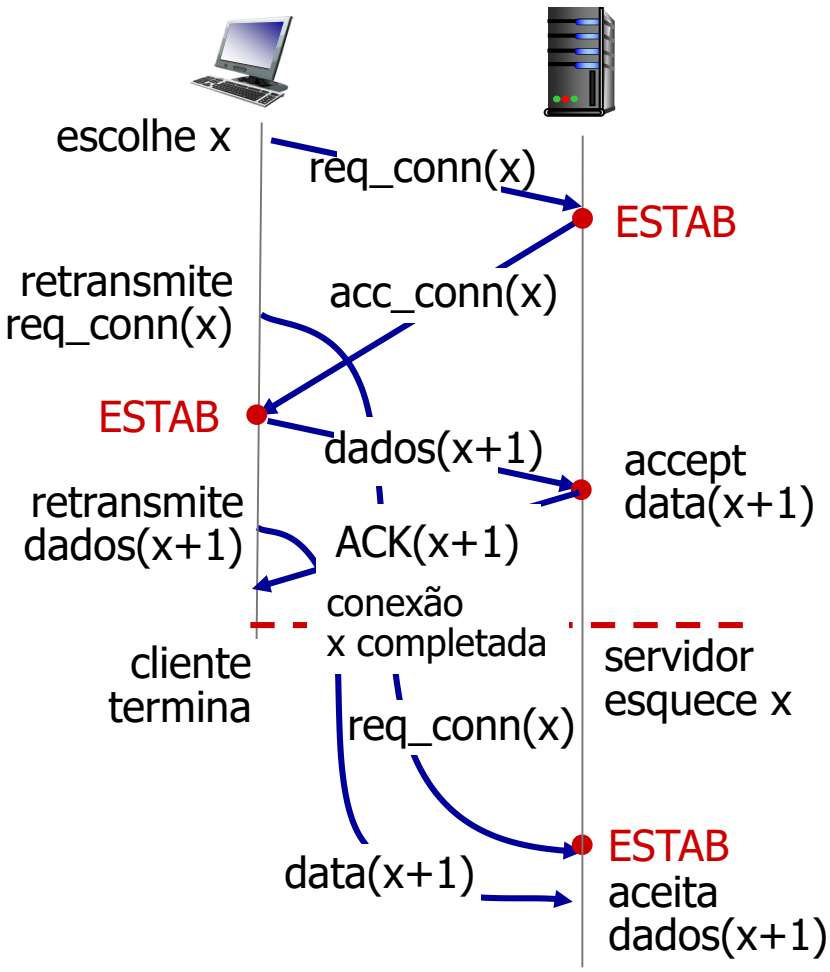
Q: *handshake* de 2 vias sempre funciona em rede?

- atrasos variáveis
- mensagens retransmitidas (e.g. req_conn(x)) devido a perdas de mensagens
- reordenamento de mensagens
- não pode-se "ver" o outro lado

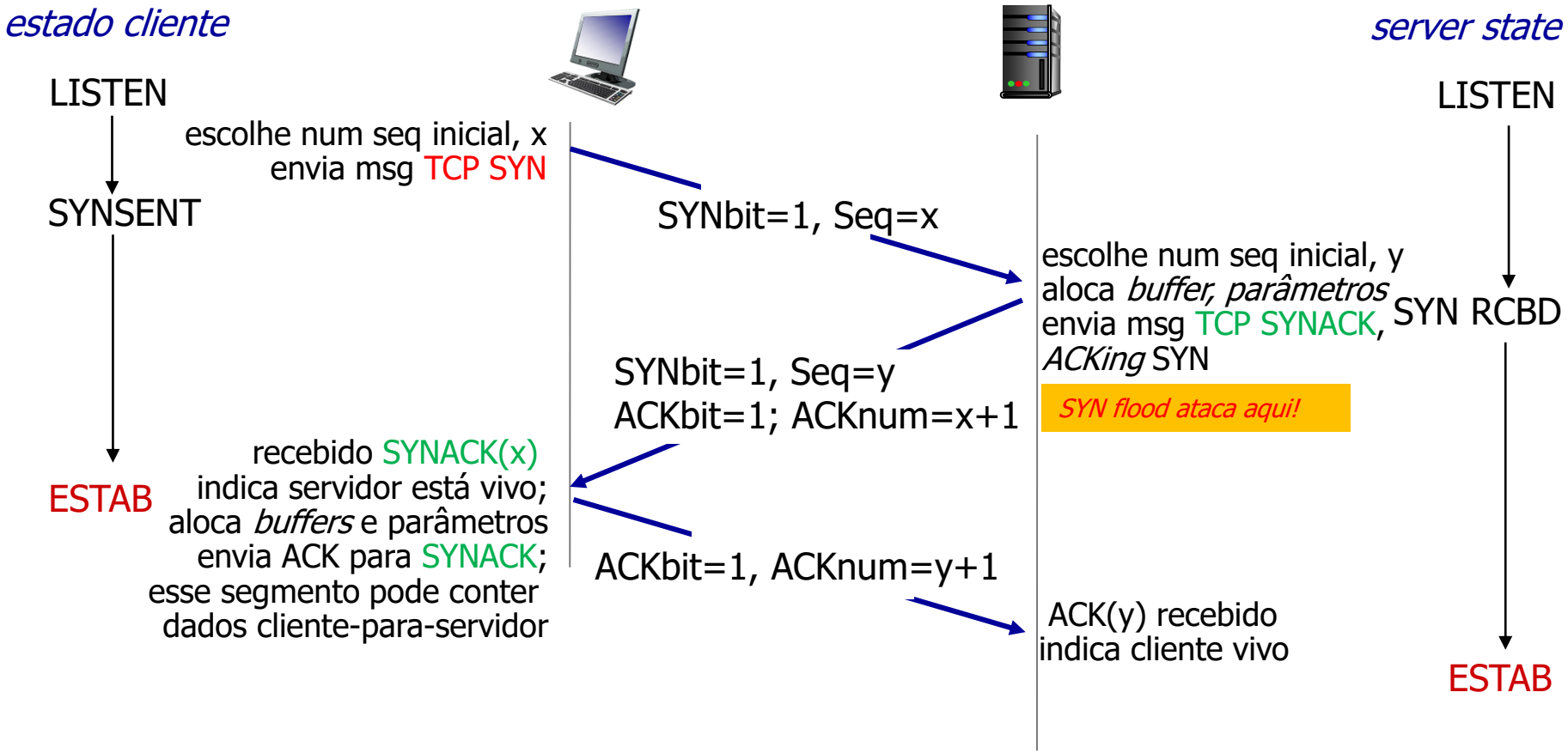
Concordando em estabelecer uma conexão

cenários de falha em *handshake* de 2 vias:

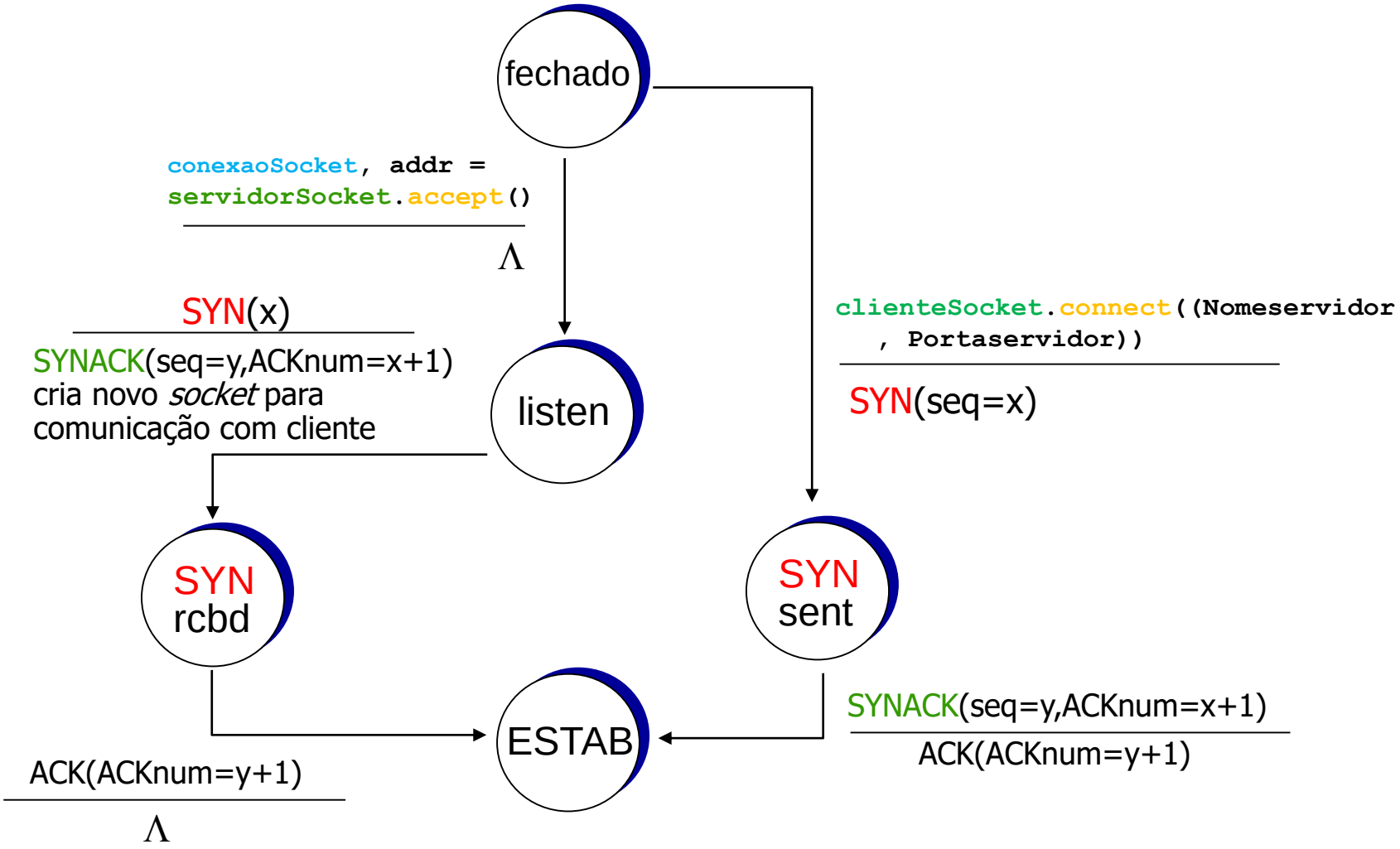




TCP: *Handshake* em 3 vias



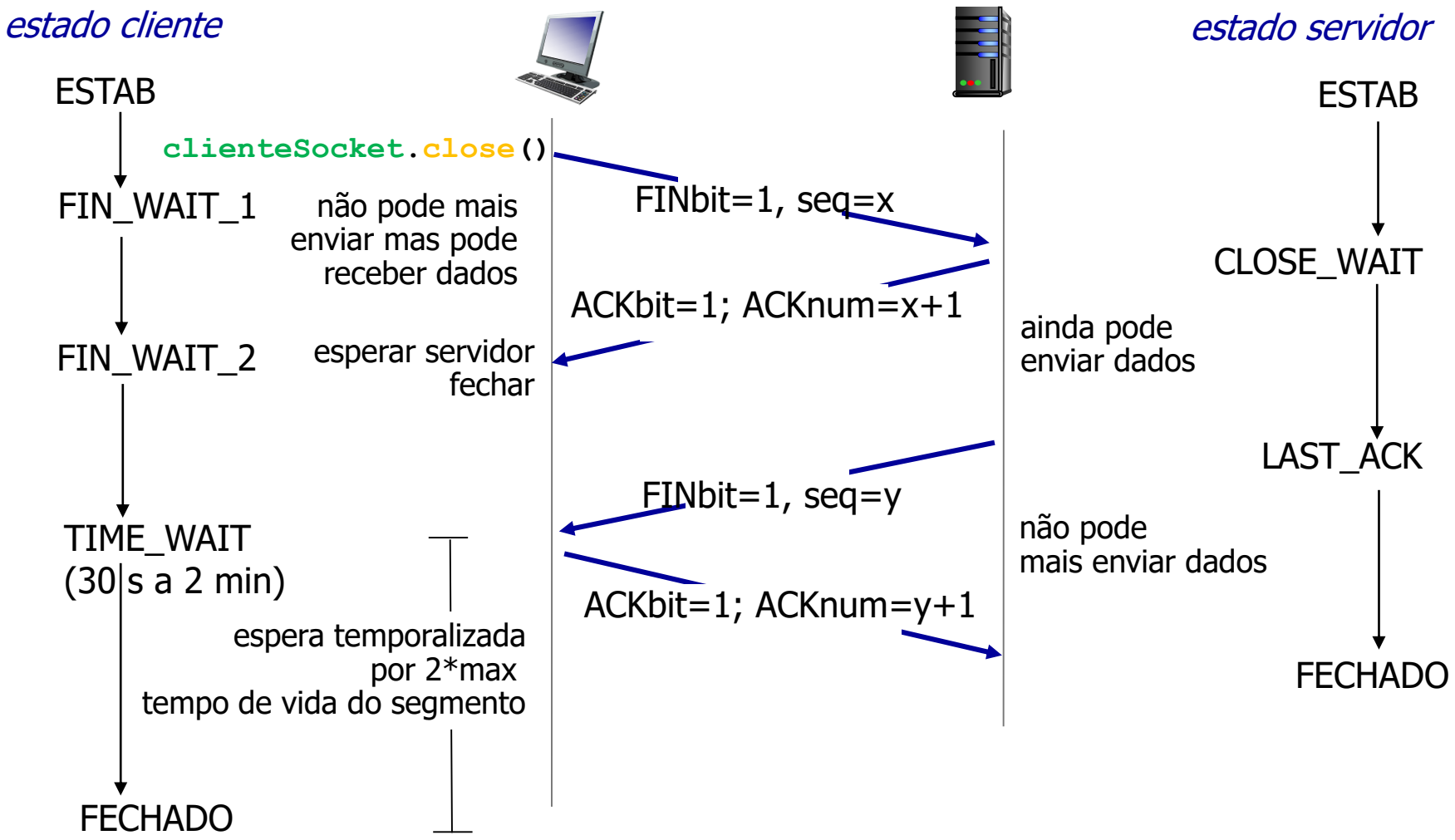
TCP: *Handshake* em 3 vias - FSM



TCP: fechando uma conexão

- cliente ou servidor cada um fecha seu lado da conexão
  - enviar segmento TCP com bit FIN = 1
- responder a FIN recebido com ACK
  - ao receber FIN, ACK pode ser combinado com FIN próprio
- trocas de FIN simultâneos pode ser tratada

TCP: fechando uma conexão



Veja: http://www.nmap.org