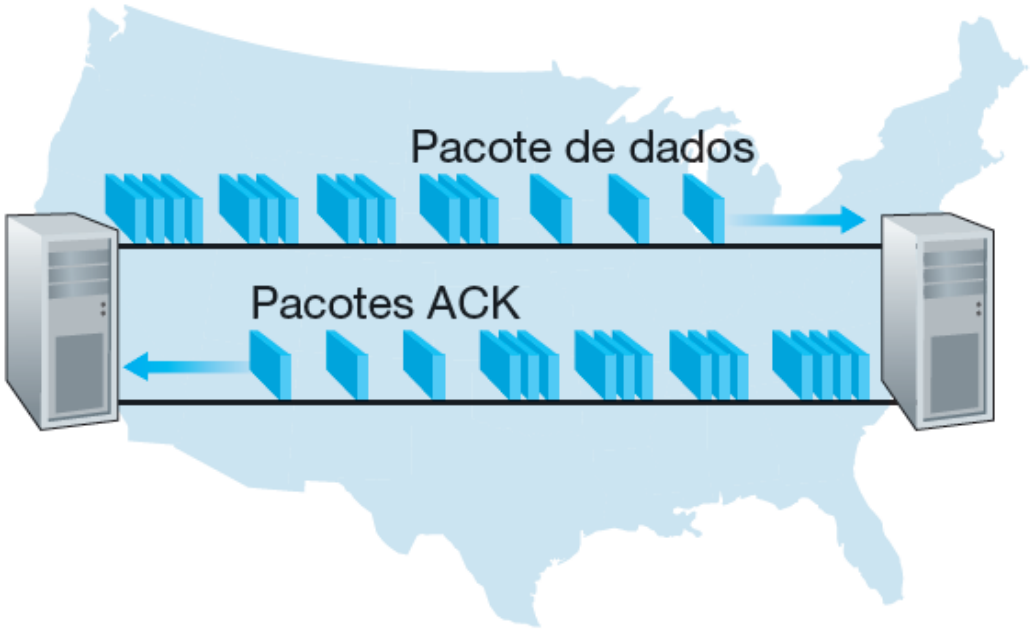
Protocolos de transferência confiável de dados c/ paralelismo

- No coração do problema do desempenho do rdt3.0 está o fato de ele ser um protocolo do tipo pare e espere.

- Um protocolo pare e espere em operação

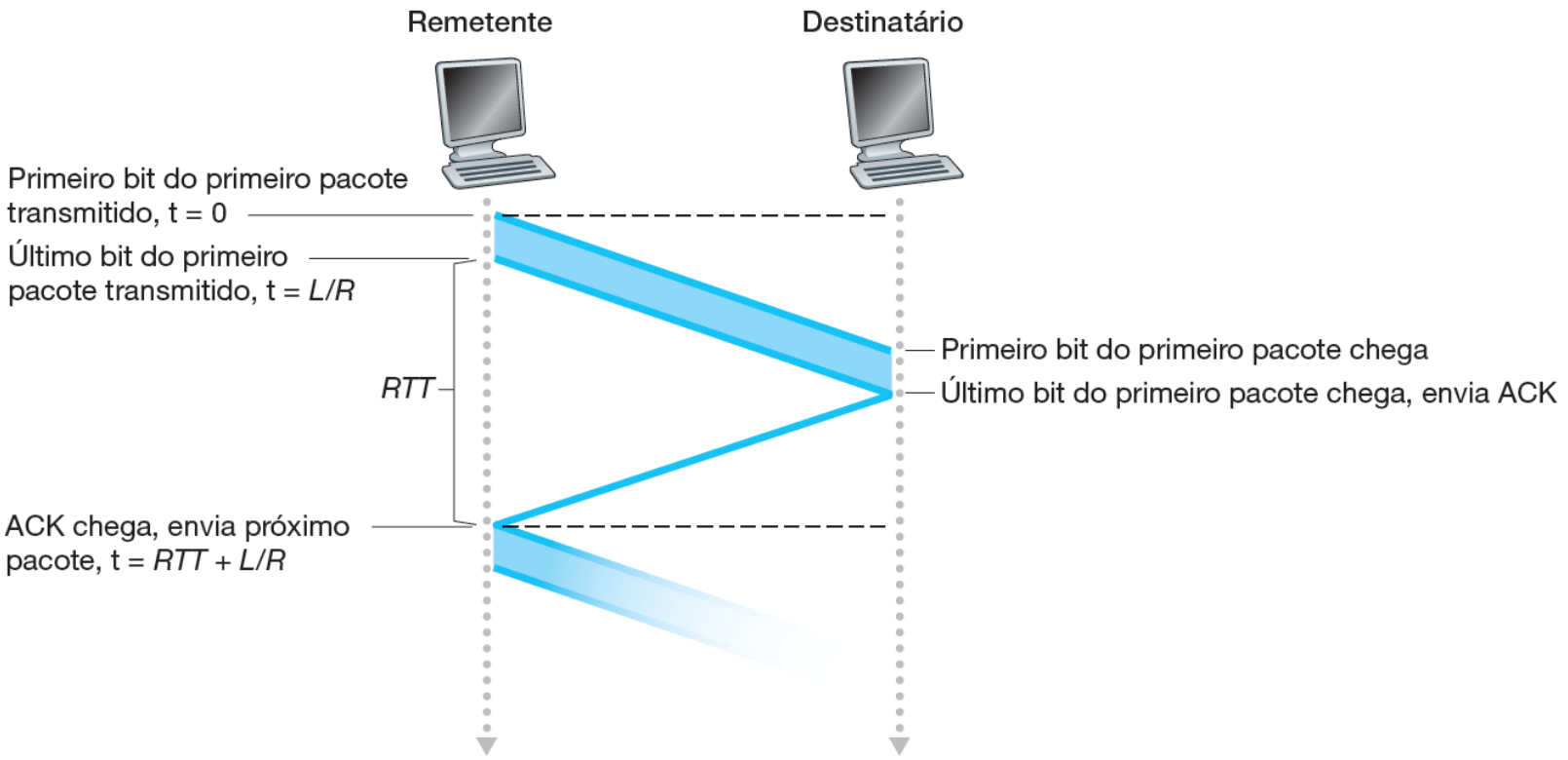
Protocolos de transferência confiável de dados c/ paralelismo

Um protocolo com paralelismo em operação

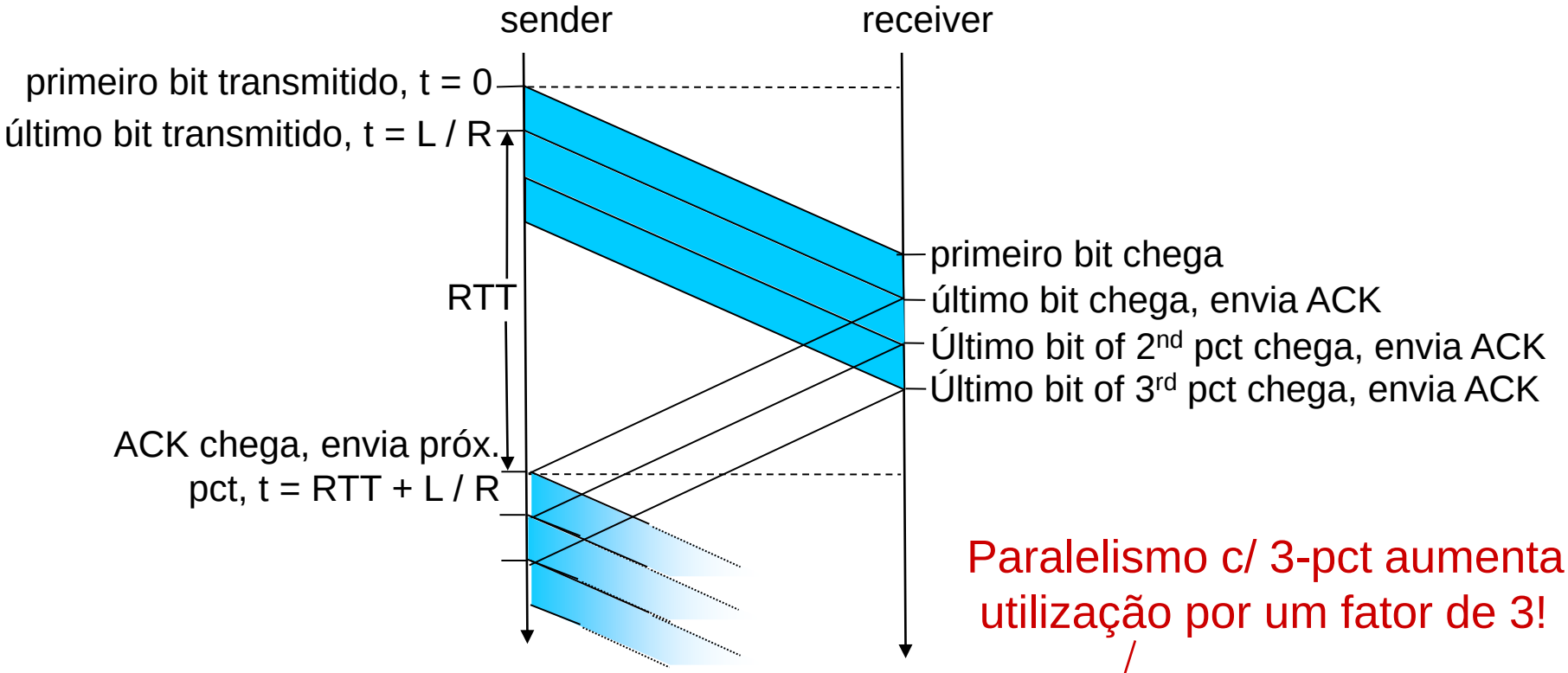


Protocolos de transferência confiável de dados c/ paralelismo

Envio com pare e espere



Paralelismo: aumenta utilização



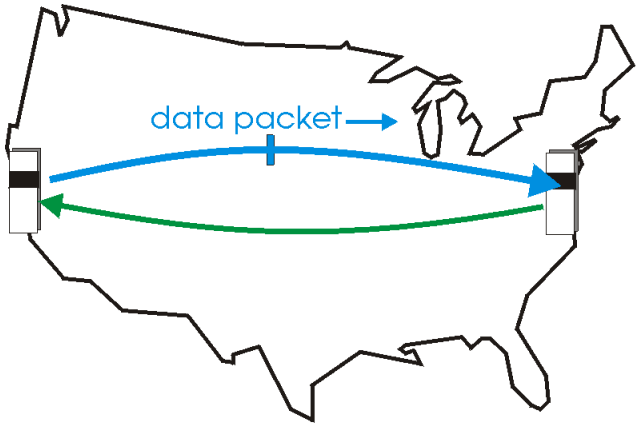
Paralelismo c/ 3-pct aumenta utilização por um fator de 3!

$$U_{sender} = \frac{3L / R}{RTT + L / R} = \frac{.0024}{30.008} = 0.00081$$

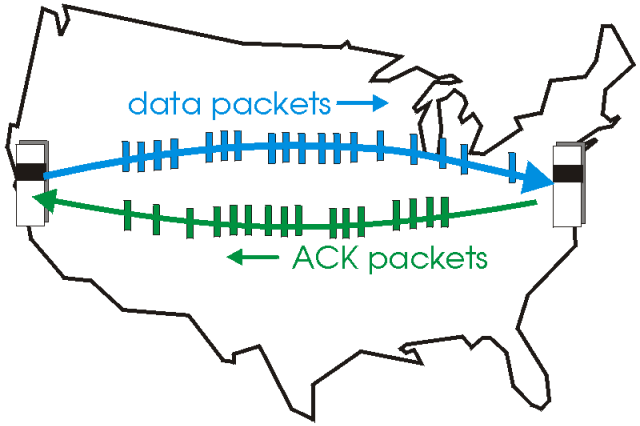
Protocolos de transferência confiável de dados c/ paralelismo

Paralelismo: transmissor envia vários pacotes ao mesmo tempo, todos esperando para serem reconhecidos

- faixa de números de sequência deve ser aumentada
- armazenamento dos pacotes no transmissor e/ou no receptor



(a) operação do protocolo stop-and-wait



(a) operação do protocolo com paralelismo

Duas formas genéricas de protocolos com paralelismo:
Go-Back-N e Selective Repeat

Protocolos de transferência confiável de dados c/ paralelismo

Go-back-N:

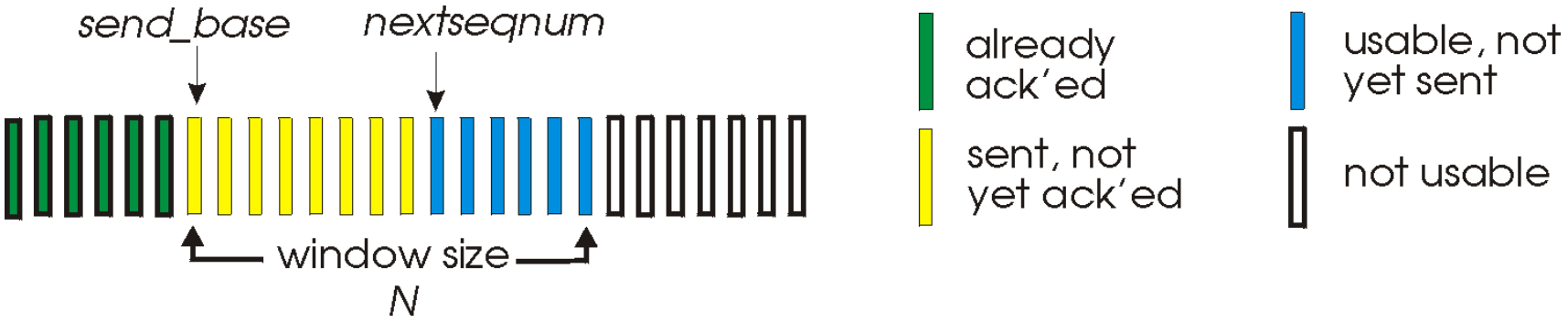
- transmissor pode ter até N pcts não reconhecidos
- receptor não envia ack se existe um GAP
- transmissor tem temporizador para o pct mais antigo não reconhecido
- qdo o temporizador expira retransmite ***todos*** os pacotes a partir deste

Selective Repeat:

- transmissor pode ter até N pcts não reconhecidos
- receptor envia ack *individual* para cada pacote
- transmissor mantém um temporizador para cada pct não reconhecido
- qdo o temporizador expira, retransmite somente o pacote não reconhecido

Go-Back-N (*sliding-window protocol*)

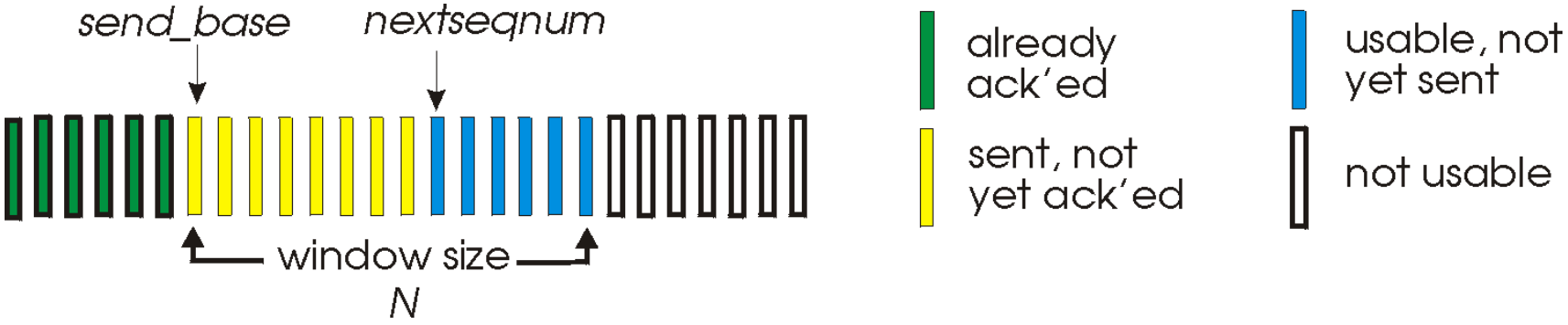
- Em um **protocolo *Go-Back-N*** (**GBN**), o remetente é autorizado a transmitir múltiplos pacotes sem esperar por um reconhecimento.
- Porém, fica limitado a ter não mais do que algum número máximo permitido, **N**, de pacotes não reconhecidos na "tubulação".
- Visão do remetente para os números de sequência no protocolo Go-Back-N:



Go-Back-N: transmissor

Transmissor:

- Coloca número de sequência com k bits no cabeçalho do pacote
- "janela" de até N pacotes não reconhecidos, consecutivos, são permitidos (N é limitado por causa do controle de fluxo e de congestion.)



- ACK(n): reconhece todos os pacotes até o número de sequência n (incluindo este limite). "ACK cumulativo"
  - pode receber ACKs duplicados (veja receptor)
- temporizador para o pacote ***mais antigo*** enviado e não confirmado
- Timeout(n): retransmite pacote mais antigo e todos os demais pacotes já transmitidos que estejam dentro da janela

GBN: FSM estendida para o transmissor

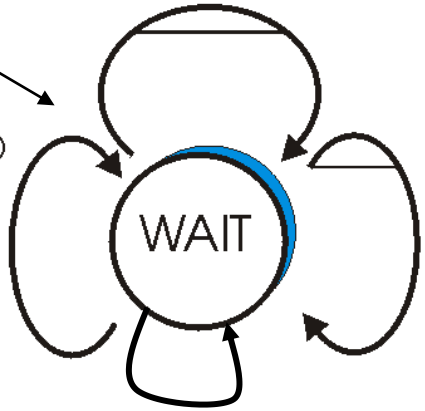
Inicialização:
base = 1
nextseqnum = 1

Dados a enviar

```
rdt_send(data)
if (nextseqnum < base+N) {
   compute chksum
   make_pkt(sndpkt(nextseqnum)),nextseqnum,data,chksum)
   udt_send(sndpkt(nextseqnum))
   if (base == nextseqnum)
      start_timer
   nextseqnum = nextseqnum + 1
   }
else
   refuse_data(data)
```

Avança a janela

```
rdt_rcv(rcv_pkt) && notcorrupt(rcvpkt)
base = getacknum(rvcpkt)+1
if (base == nextseqnum)
   stop_timer
 else
   start_timer
```



Ocorreu timeout

```
timeout
start_timer
udt_send(sndpkt(base))
udt_send(sndpkt(base+1)
.....
udt_send(sndpkt(nextseqnum-1))
```

Recebeu ACK

```
rdt_rcv(rcvpkt) && corrupt(rcvpkt)
Λ
```

Recebeu ACK com erro

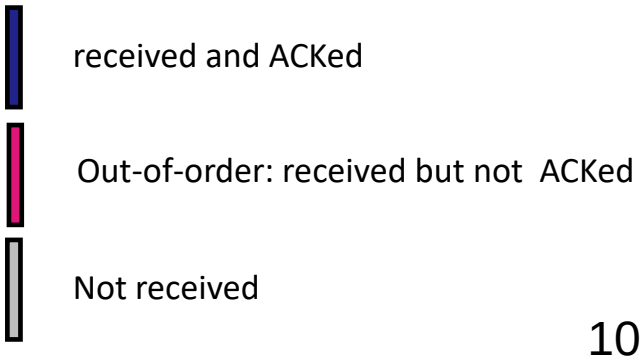
Go-Back-N: receptor

receptor simples:

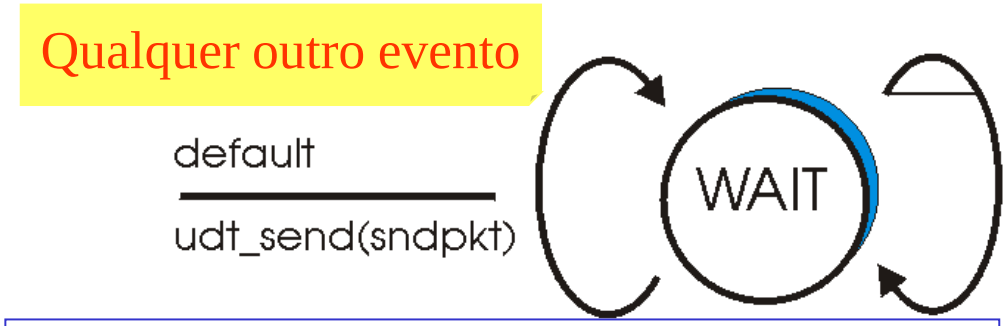
- somente ACK: sempre envia ACK para pacotes corretamente recebidos com o mais alto número de seqüência *em ordem* → pode gerar ACKs duplicados (vide exemplo)
  - precisa lembrar apenas do número de seqüência esperado **(expectedseqnum)**

- pacotes fora de ordem (pacote anterior foi perdido) ⟶ default:
  - descarte (não armazena) ⟶ não há necessidade de buffer de recepção! Dependendo da implementação, pode armazenar
  - reconhece pacote com o mais alto número de seqüência em ordem

Visão do receptor do espaço do número de sequência :

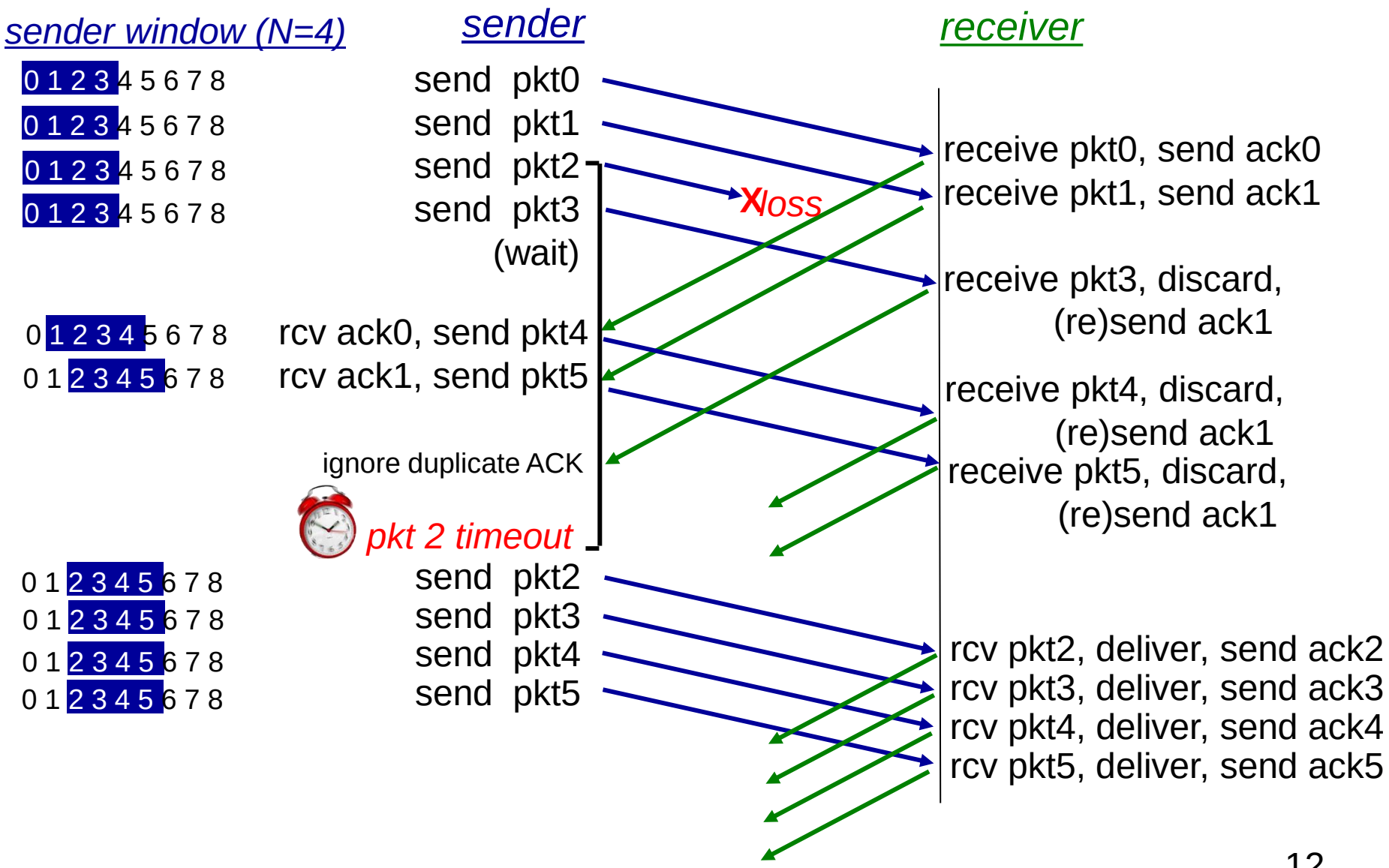


GBN: FSM estendida para o receptor



Inicialização:
- expectedseqnum = 1
- sndpkt = make_pkt(expectedseqnum, ACK, checksum)

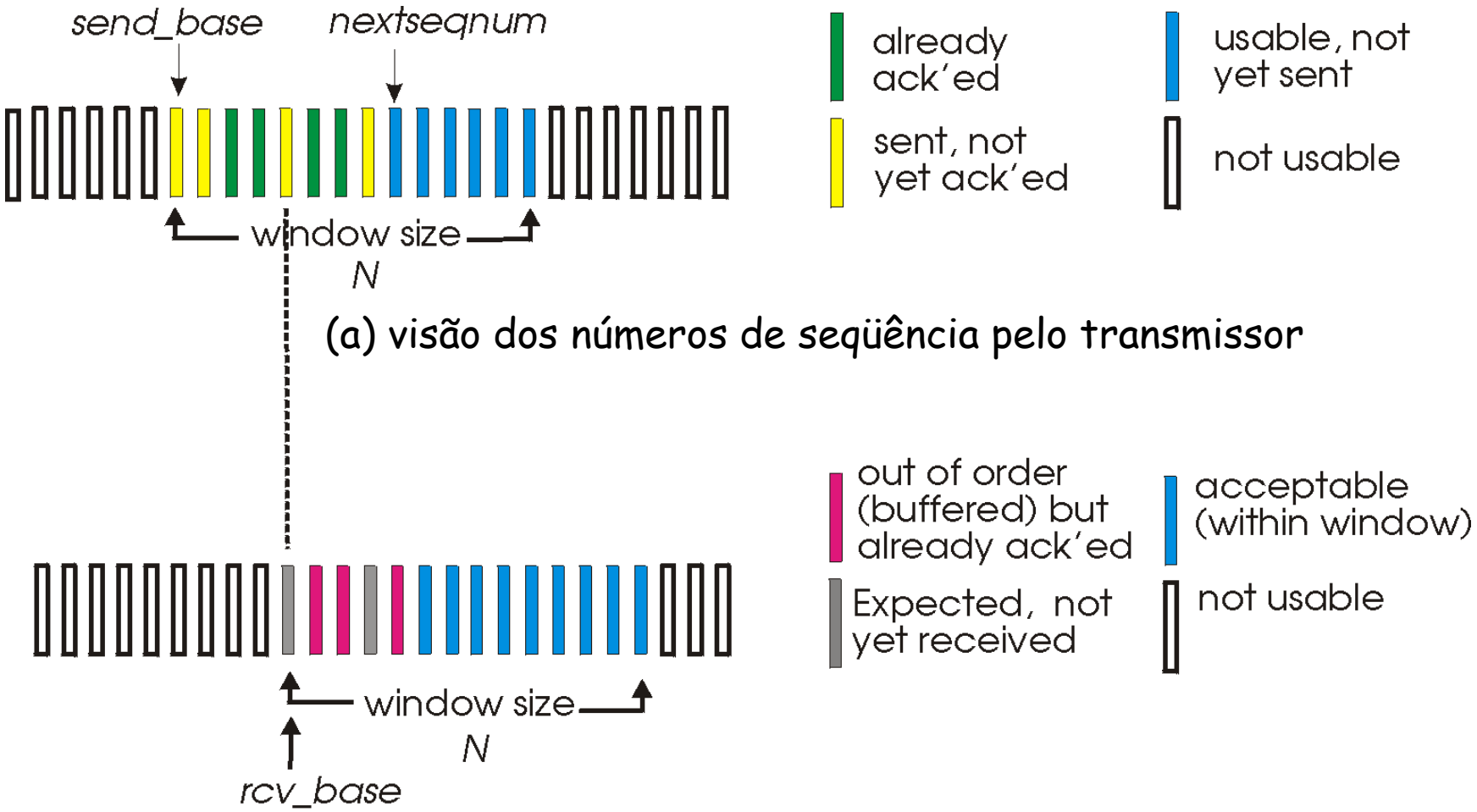
Go-Back-N em ação (janela: 4 pacotes)



Retransmissão seletiva (Selective Repeat - SR)

- Transmissor somente retransmite os pacotes para os quais um ACK não foi recebido
  - evita retransmissões desnecessárias
  - transmissor temporiza cada pacote não reconhecido (sem ACK)
- Receptor reconhece *individualmente* todos os pacotes recebidos corretamente
  - armazena pacotes, quando necessário, para posterior entrega em ordem para a camada superior
- Janela de transmissão
  - N números de seqüência consecutivos
  - novamente limita a quantidade de pacotes enviados, mas não reconhecidos (i.e., com ACK pendente)

Selective Repeat: janelas do transmissor e do receptor



(a) visão dos números de seqüência pelo transmissor

(b) visão dos números de seqüência pelo receptor

Selective Repeat

transmissor

dados da camada superior :

- se o próximo número de seqüência disponível está na janela, envia o pacote e dispara o temp. p/ ele

timeout(n):

- reenvia pacote n, e redispara seu temporizador de timeout

ACK(n) em [sendbase,sendbase+N]:

- marca pacote n como recebido
- se n é o menor pacote não reconhecido, avança a base da janela para o próximo número de seqüência não reconhecido

receptor

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

- envia ACK(n)
- fora de ordem: armazena
- em ordem: entrega n (e demais em ordem já ack´ed) para a camada superior, avança janela para o próximo pacote ainda não recebido

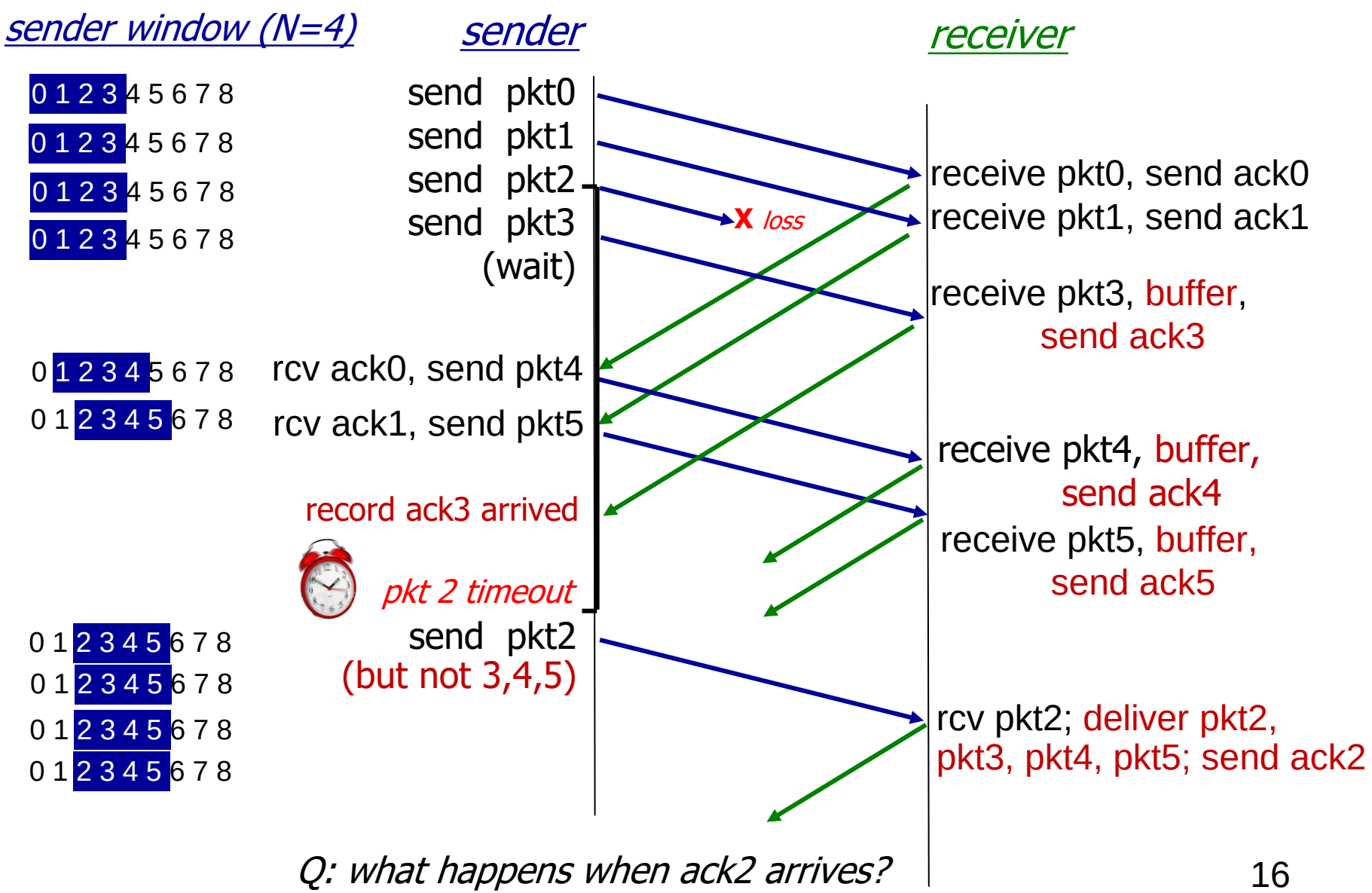
pkt n em [rcvbase-N,rcvbase-1]

- ACK(n)

caso contrário:

- ignora

Selective Repeat em ação



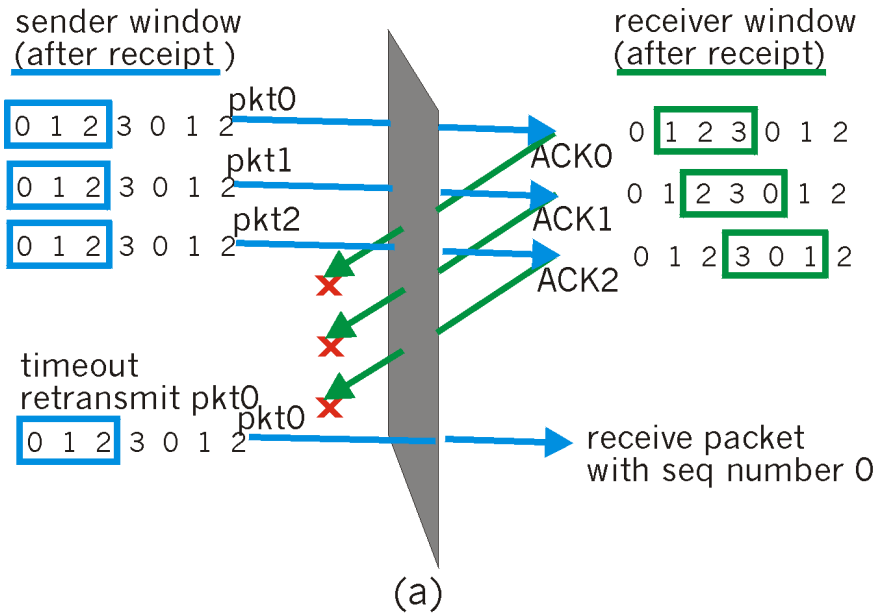
Selective repeat: dilema

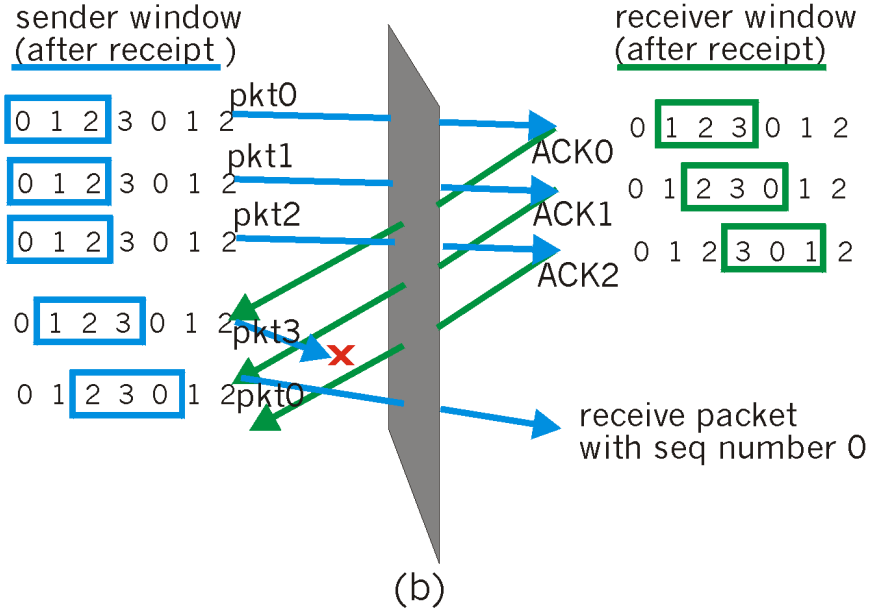
Exemplo:

- núms. de seqüência: 0,1,2,3
- tamanho da janela=3
- receptor não vê diferença nos dois cenários!
- incorretamente passa dados duplicados como novos (figura a)

Q: qual a relação entre o espaço de numeração seqüencial e o tamanho da janela?

Fazer problema 23, cap3!





Transferência confiável de dados: resumo

Resumo de mecanismos de transferência confiável de dados e sua utilização:

- **Soma de verificação** - Usada para detectar erros de bits em um pacote transmitido.
- **Temporizador** - Usado para controlar a temporização/retransmissão de um pacote, possivelmente porque o pacote (ou seu ACK) foi perdido dentro do canal.
- **Número de sequência** - Usado para numeração seqüencial de pacotes de dados que transitam do remetente ao destinatário.

Transferência confiável de dados: resumo

- **Reconhecimento** - Usado pelo destinatário para avisar ao remetente que um pacote ou conjunto de pacotes foi recebido corretamente.

- **Reconhecimento negativo** - Usado pelo destinatário para avisar ao remetente que um pacote não foi recebido corretamente.

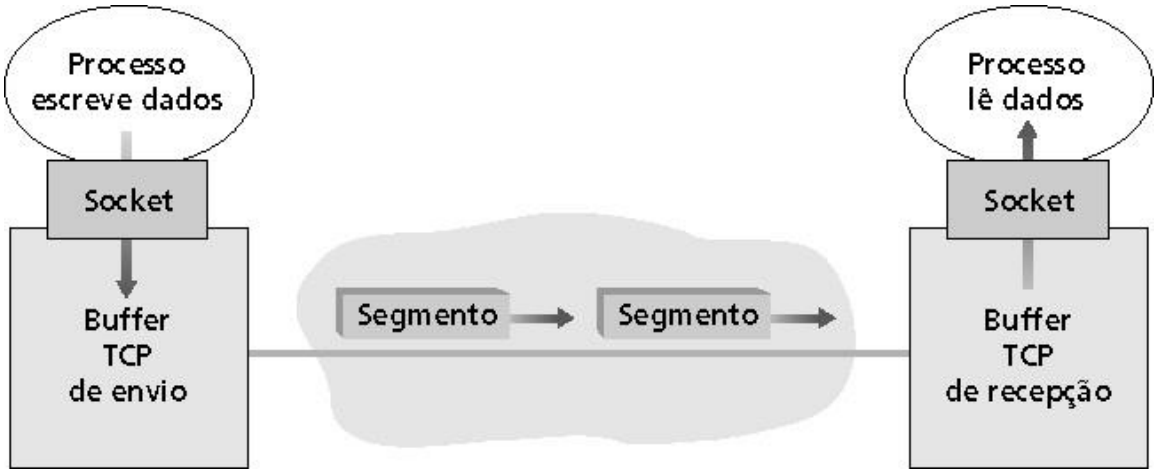
- **Janela, paralelismo** - O remetente pode ficar restrito a enviar somente pacotes com números de sequência que caiam dentro de uma determinada faixa.

TCP: visão geral (RFCs: 793, 1122, 2018, 5681, 7323)

- transmissão ponto a ponto:
  - 1 transmissor, 1 receptor
- fluxo de bytes, ordenados, confiável e seqüencial
- orientado a conexão:
  - *handshaking* (troca de msgs de controle) inicia estado do transmissor e do receptor antes de trocar dados
- transmissão full duplex:
  - fluxo de dados bi-direcional na mesma conexão
  - MSS: tamanho máximo do segmento
  - conexão fim-a-fim: reserva de recursos somente nos sistemas finais, diferentemente do FDM, TDM e circuitos virtuais

TCP: visão geral (RFCs: 793, 1122, 2018, 5681, 7323)

- paralelismo: transmissão de vários pacotes sem confirmação (ACK)
  - controle de congestionamento e de fluxo definem o tamanho da janela de transmissão
- controle de fluxo:
  - transmissor não esgota a capacidade do receptor
- *buffers* de envio e recepção



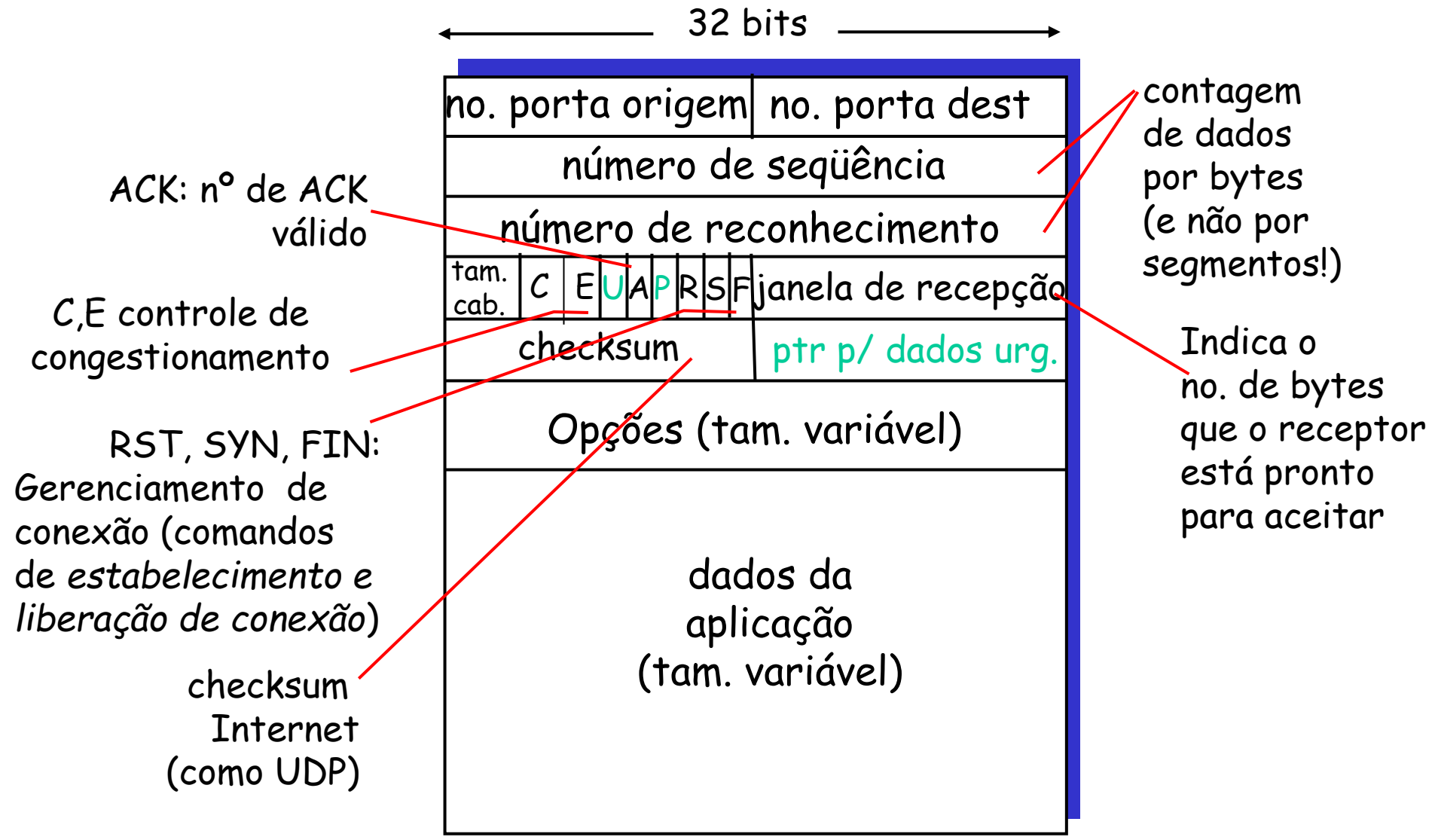
A conexão TCP

- Uma vez estabelecida uma conexão TCP, dois processos de aplicação podem enviar dados um para o outro.

- O TCP combina cada porção de dados do cliente com um cabeçalho TCP, formando, assim, **segmentos TCP**.

  MSS – tamanho máximo do segmento

TCP: estrutura do segmento

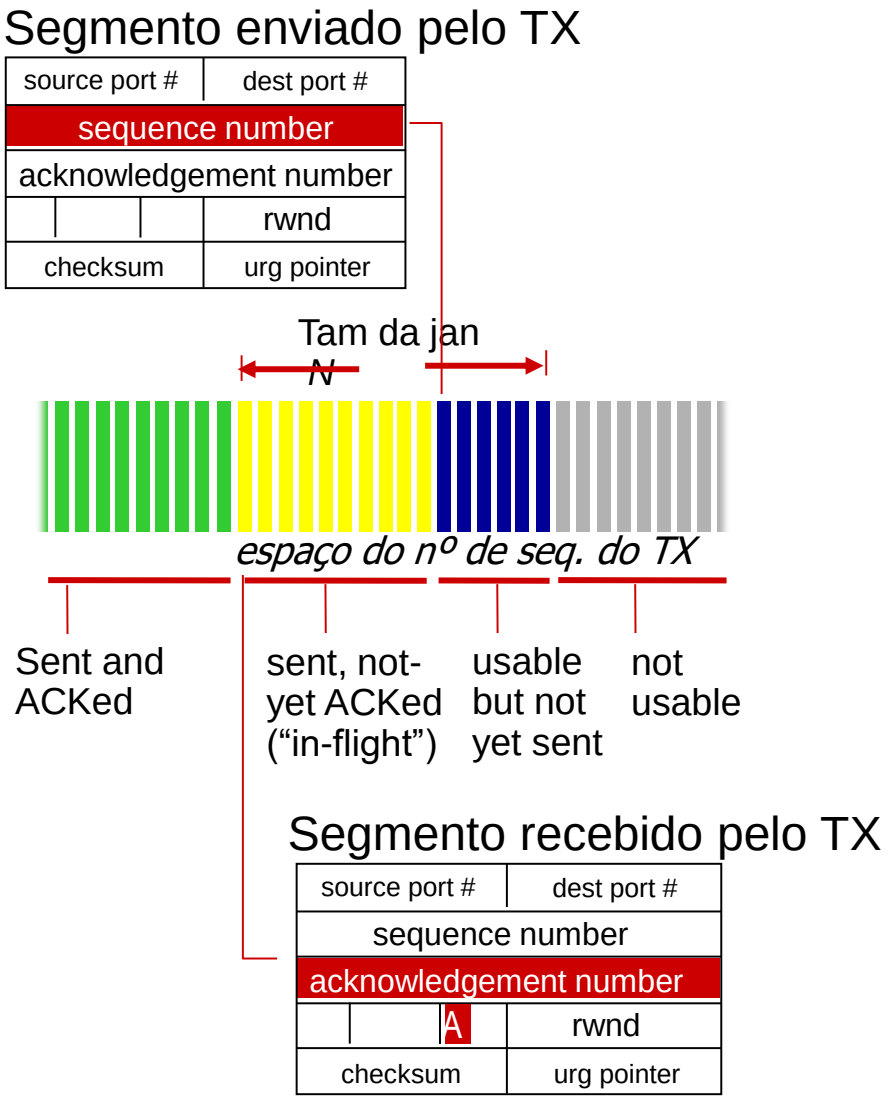


Números de sequência e números de ACKs

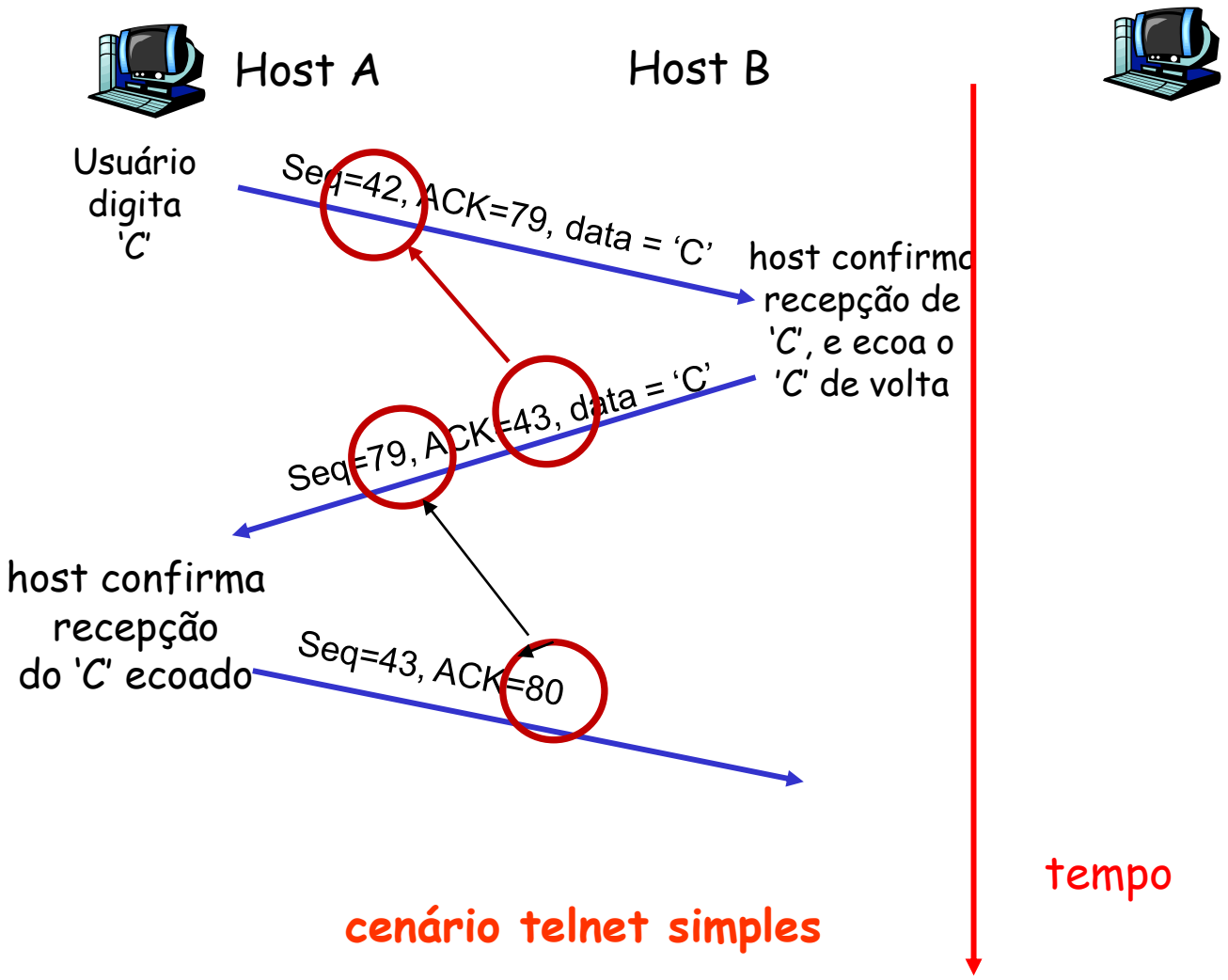
- O **número de sequência** para um segmento é o número do primeiro byte de dados do segmento TCP.
- O **número de reconhecimento** que o hospedeiro A atribui a seu segmento é o número de sequência do próximo byte que ele estiver aguardando do hospedeiro B.
- O TCP reconhece bytes até o primeiro byte que estiver faltando na cadeia
- Dizemos que o TCP provê **reconhecimentos cumulativos**.

TCP: nº de sequência e nº de ACK

- P: como o receptor trata segmentos foram de ordem?
  - descarta?
  - bufferiza para entrega posterior em ordem?
- R: A especificação do TCP não define. Fica a critério do implementador!



Números de sequência e ACKs do TCP (ex: conexão *telnet*)



TCP: *Round Trip Time* e temporização

Q: como escolher o valor da temporização *(timeout)* do TCP?

- maior que o RTT
  - nota: RTT é variável
- **muito curto:** temporização prematura
  - retransmissões desnecessárias
- **muito longo:** a reação à perda de segmento fica lenta

Q: Como estimar o RTT?

- **SampleRTT:** último tempo medido da transmissão de um segmento até a respectiva confirmação
  - ignora retransmissões e segmentos reconhecidos de forma cumulativa
- **SampleRTT** pode variar de forma rápida, portanto é desejável um "amortecedor" para a estimativa do RTT
  - Ideia: usar várias medidas recentes e não apenas o último **SampleRTT** obtido

TCP: *Round Trip Time* e temporização

EstimatedRTT = (1-x) * EstimatedRTT + x * SampleRTT

- Média ponderada
- valor típico de x = 0.125: história (representada pela estimativa anterior) tem mais peso que o último RTT medido
- influência de uma dada amostra decresce de forma exponencial

Definindo a temporização da retransmissão

- **Temporização = EstimatedRTT** + uma "margem de segurança"
- grandes variações no **EstimatedRTT** → maior margem de segurança, mas reação à perda demora mais

Temporização = EstimatedRTT + 4*Desvio

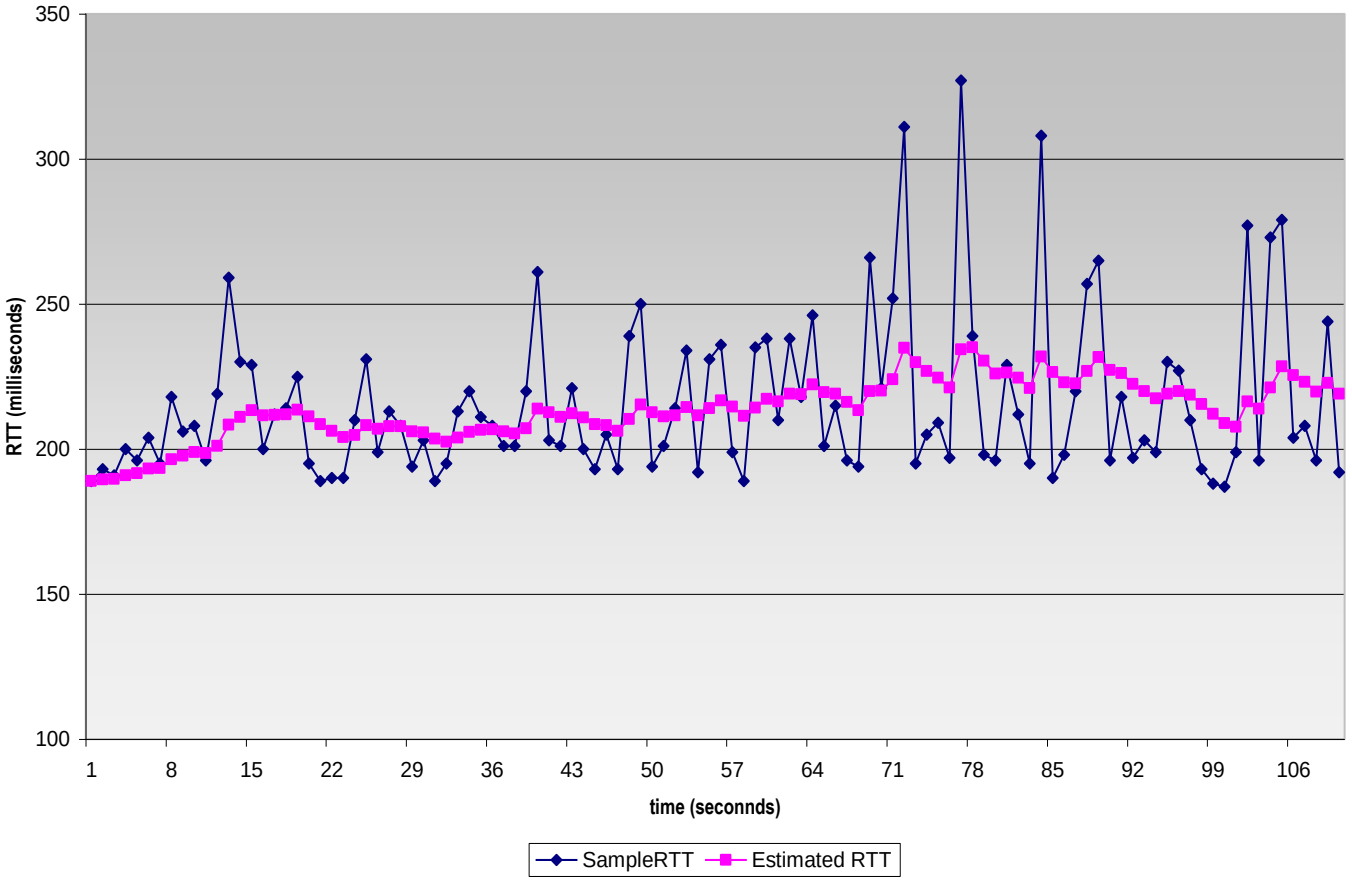
Desvio = (1-y) * Desvio +
y * |SampleRTT - EstimatedRTT|

"margem de segurança"

Valor típico de y = 0.25

TCP: *Round Trip Time* e temporização

RTT (ms)



TCP: transferência confiável de dados

Retransmissões podem ser disparadas por:

- ✓ eventos de timeout
- ✓ acks duplicados

Inicialmente vamos considerar um transmissor TCP simplificado:

- ✓ ignorar acks duplicados
- ✓ ignorar o controle de fluxo e o controle de congestionamento

TCP: eventos no transmissor

dados receb. da app:

- cria segmento c/ #seq

- #seq é o nº do 1º byte do fluxo de dados no segmento

- dispara o temporizador se este ainda estiver parado

  - temporizador se refere ao mais antigo segmento não reconhecido

  - intervalo até expirar:

    TimeOutInterval

timeout:

- retransmite segmento que causou o *timeout*

- redispara o temporizador

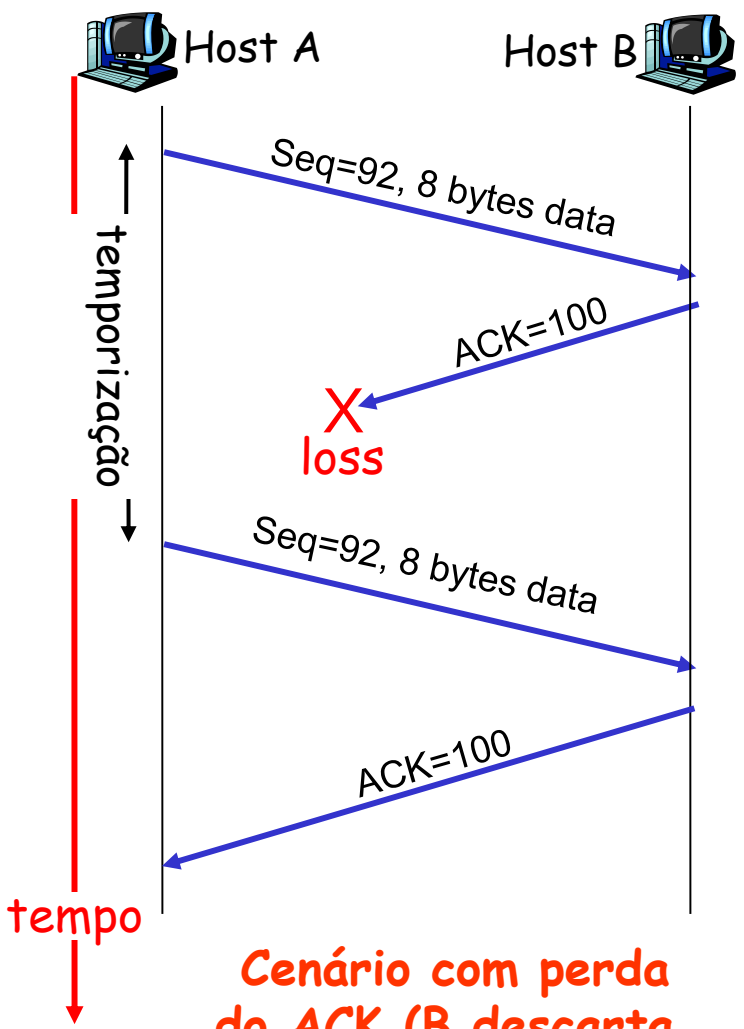
ack recebido:

- se *ack* reconhece segmentos anteriormente não reconhecidos (cumulativo)

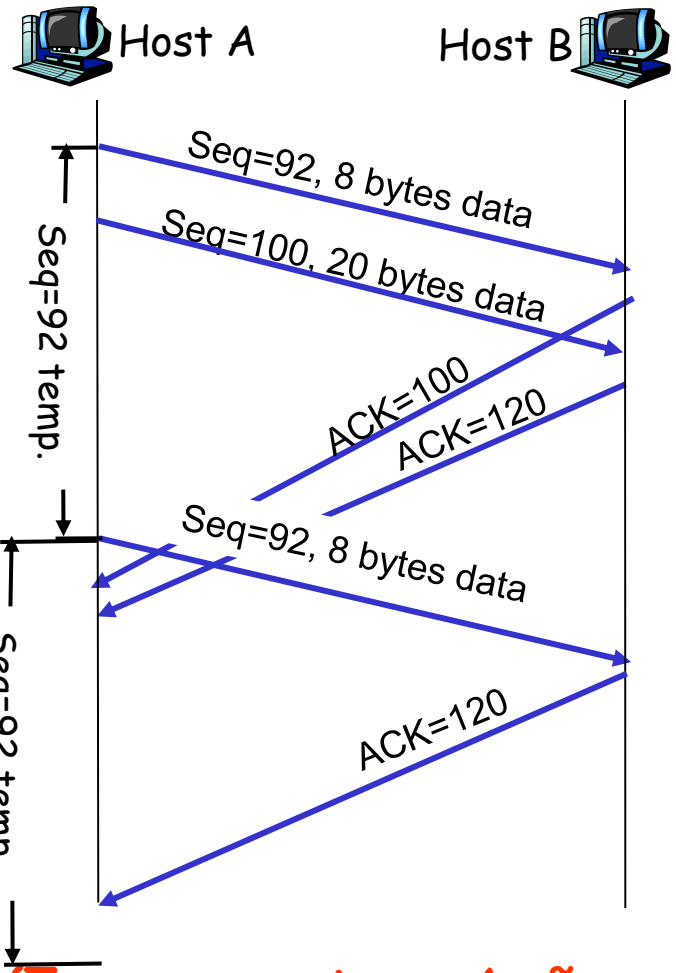
  -- atualiza o SendBase com o nº do ACK

  -- dispara temporizador se ainda existem segmentos não reconhecidos

TCP: cenários de retransmissão

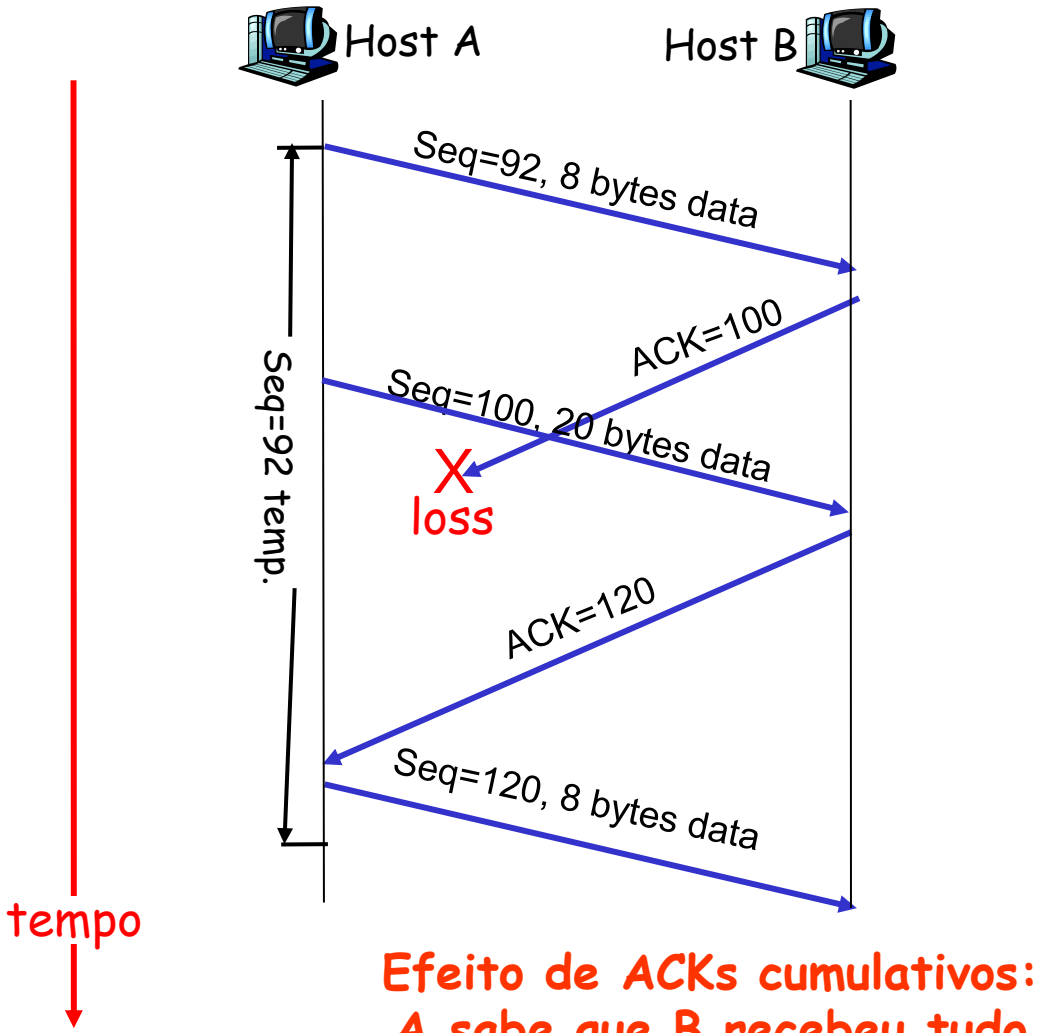


Cenário com perda do ACK (B descarta segmento duplicado)



ACK atrasado (Temp. prematura; A não reenvia segundo segmento desde que o ACK120 chegue antes do temporizador expirar)

TCP: cenários de retransmissão



Efeito de ACKs cumulativos:
A sabe que B recebeu tudo
até o *byte* 119

TCP *fast retransmit*

Obs: período de timeout normalmente é longo

⇒ **longo atraso antes de reenviar o pct perdido**

- Detecta segmentos perdidos via ACKs duplicados.

– transmissor normalmente envia muitos segmentos antes de ocorrer um timeout

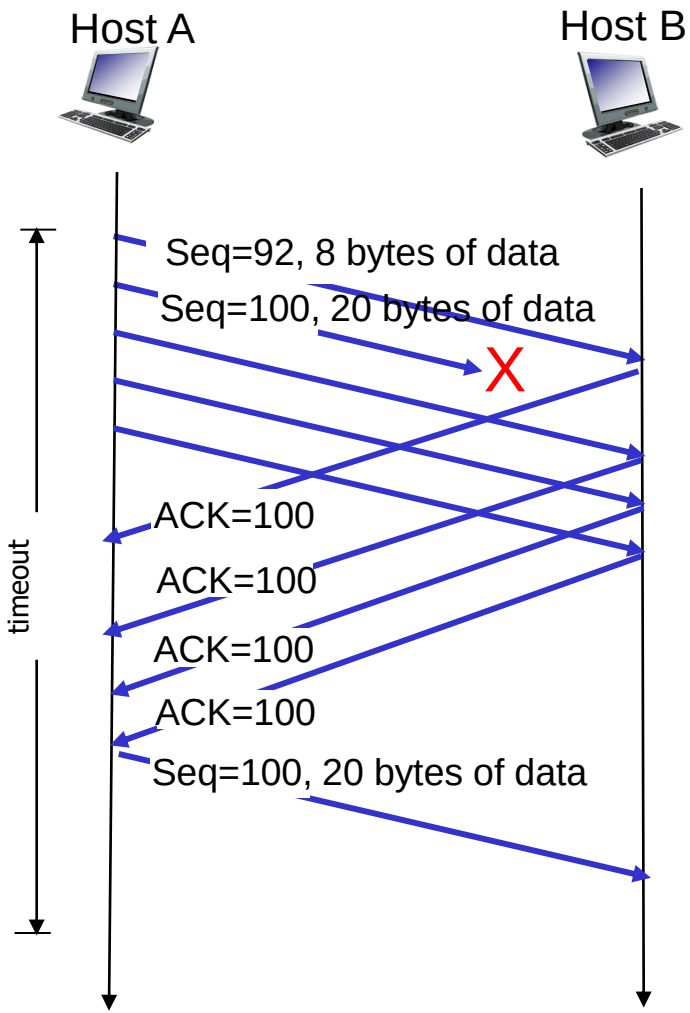
– se segmento é perdido, é provável que haja muitos ACKs duplicados

TCP fast retransmit

Se transmissor recebe 3 ACKs duplicados para o mesmo dado, reenvia o segmento não reconhecido com o menor nº de sequência

- Provavelmente o segmento não reconhecido foi perdido, portanto não espere pelo timeout

TCP *fast retransmit*



fast retransmit após transmissor receber 3 ACKs duplicados

TCP: controle de fluxo

- O TCP provê um **serviço de controle de fluxo** às suas aplicações, para eliminar a possibilidade de o remetente estourar o buffer do destinatário.

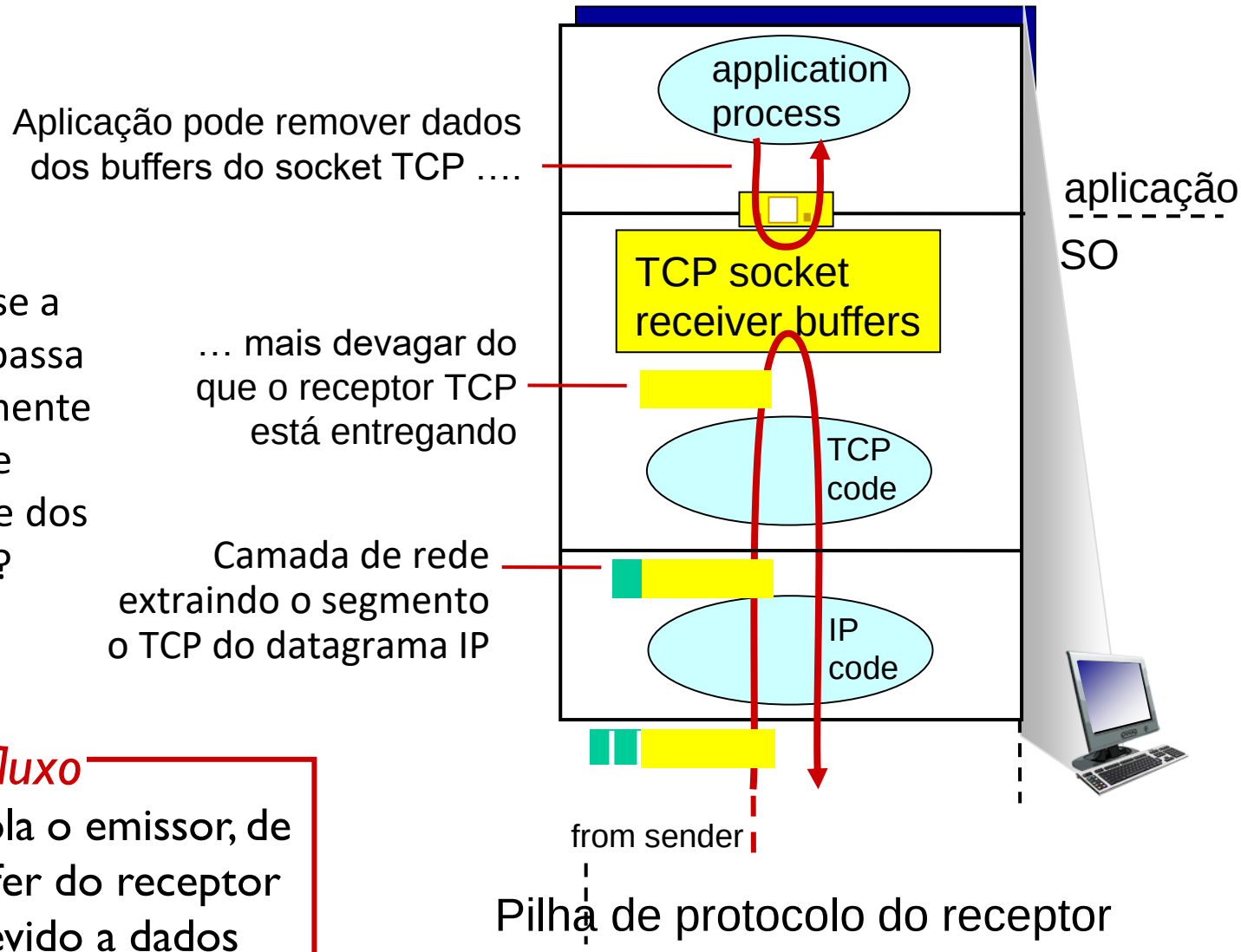
- **O controle de fluxo** é um serviço de compatibilização de velocidades.

- O TCP oferece serviço de controle de fluxo fazendo com que o remetente mantenha uma variável denominada **janela de recepção**.

TCP: controle de fluxo

Questão:

O que acontece se a camada de rede repassa dados mais rapidamente do que a camada de aplicação os remove dos buffers dos sockets?



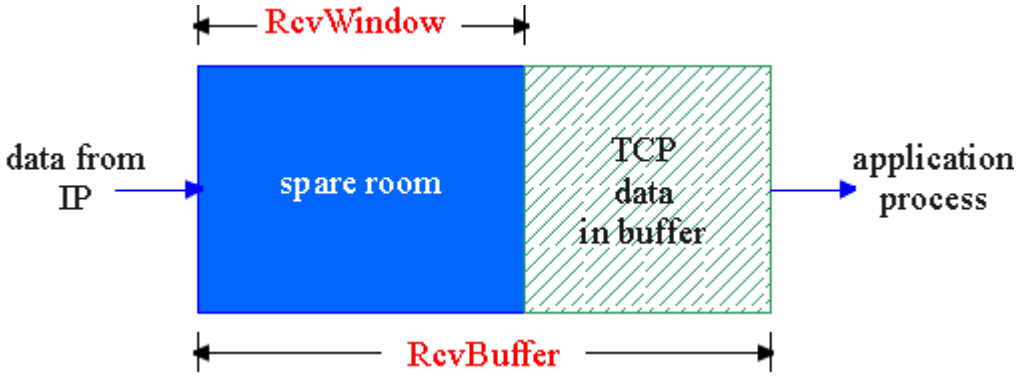
controle de fluxo

Receptor controla o emissor, de forma que o buffer do receptor não se esgote devido a dados enviados muito rapidamente

TCP: controle de fluxo

RcvBuffer = tamanho do Buffer de recepção do TCP

RcvWindow = total de espaço livre no buffer



armazenamento no lado do receptor

receptor: explicitamente informa ao transmissor sobre a quantidade de área livre no buffer, que varia dinamicamente (**campo RcvWindow** no cabeçalho do segmento TCP)

- tam. do RcvBuffer é ajustado através das opções do socket (default: 4096 bytes)
- Muitos sistemas operacionais ajustam o RcvBuffer automaticamente

transmissor: mantém a quantidade de dados transmitidos mas ainda não reconhecidos menor que a quantidade expressa no último **RcvWindow** recebido

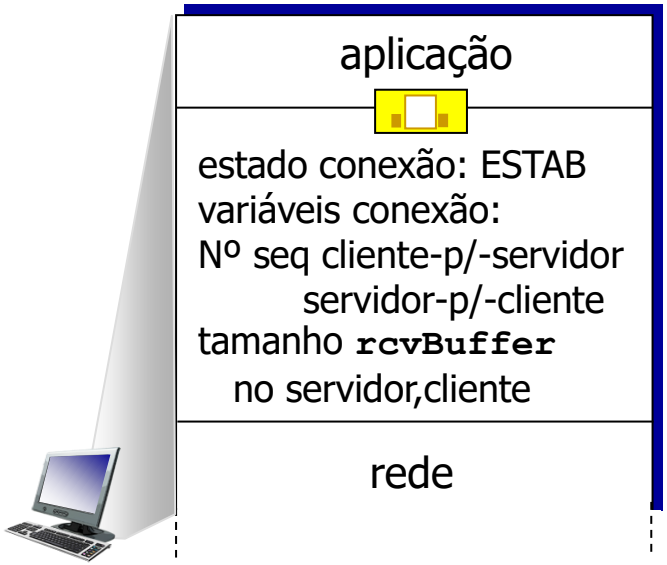
⇒ garante que não vai exceder a capacidade do buffer do receptor

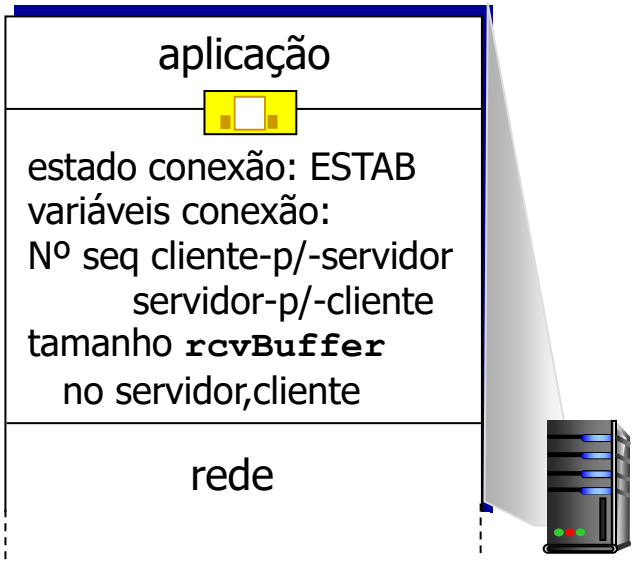
TCP: gerenciamento de conexões

antes de trocar dados, transmissor e receptor TCP dialogam:

concordam em estabelecer uma conexão (cada um sabendo que o outro quer estabelecer a conexão)

concordam com os parâmetros da conexão.






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

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

TCP: estabelecimento de conexão

TCP transmissor estabelece conexão com o receptor antes de trocar segmentos de dados

- inicializar variáveis:
  - números de sequência
  - buffers, controle de fluxo (ex. **`RcvWindow`**)
- ***cliente:*** iniciador da conexão
- ***servidor:*** chamado pelo cliente

Apresentação em 3 vias:

Passo 1: cliente envia TCP SYN ao servidor

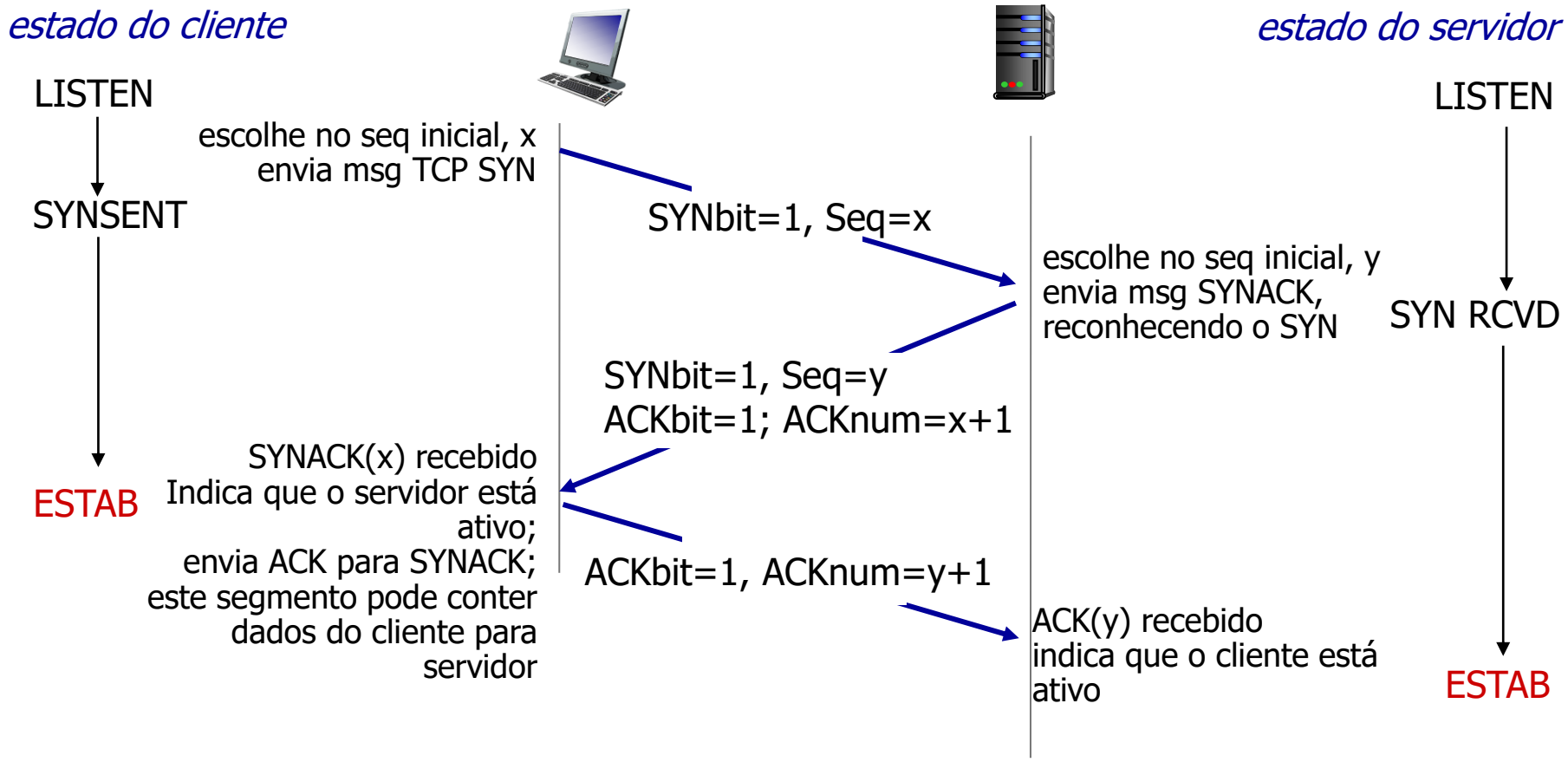
- especifica número de sequência inicial do cliente

Passo 2: servidor que recebe o SYN, responde com segmento SYNACK

- reconhece o SYN recebido
- aloca buffers
- especifica o número de sequência inicial do servidor

Passo 3: cliente reconhece o SYNACK

TCP: apresentação de três vias



Analogia com uma apresentação em 3 vias humana

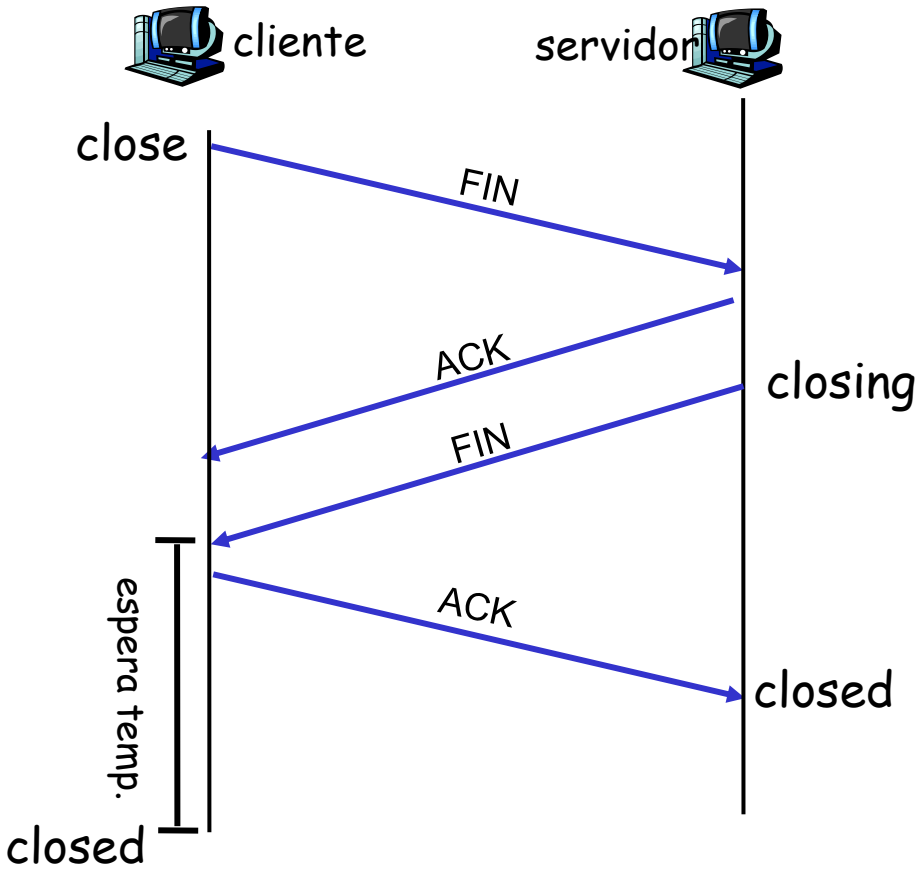
TCP: término de conexão

Fechando uma conexão:

Aplicação cliente fecha o socket:
`clientSocket.close();`

Passo 1: o cliente envia o segmento TCP FIN (bit FIN=1) ao servidor

Passo 2: servidor recebe FIN, responde com ACK. Fecha a conexão, envia FIN.

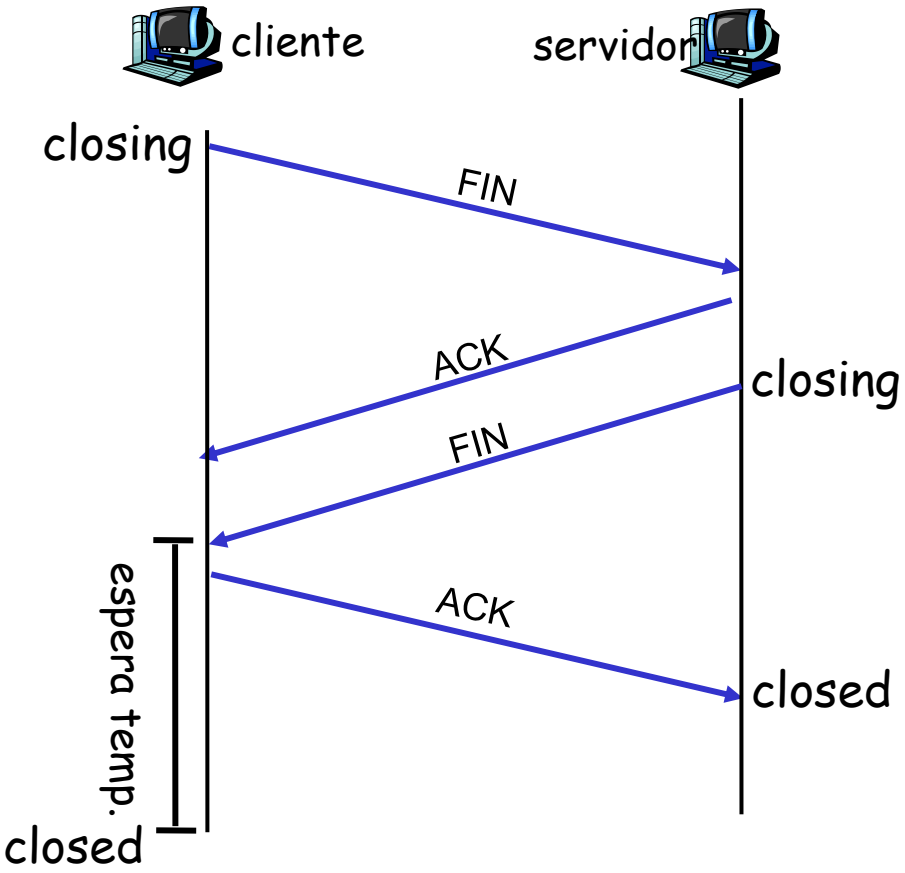


TCP: término de conexão

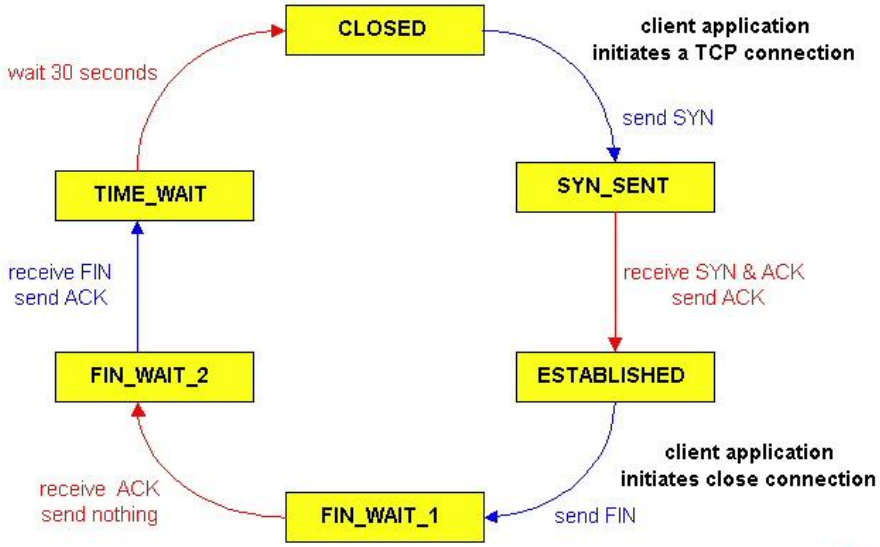
Passo 3: cliente recebe FIN, responde com ACK.

- Entra em "espera temporizada" - vai responder com ACK a eventuais FINs recebidos
  - se o ACK original do cliente se perder

Passo 4: servidor, recebe ACK. Conexão fechada.



TCP: controle de conexão



Estados do Cliente

Estados do Servidor