Capítulo 3: Camada de Transporte

Metas do capítulo:

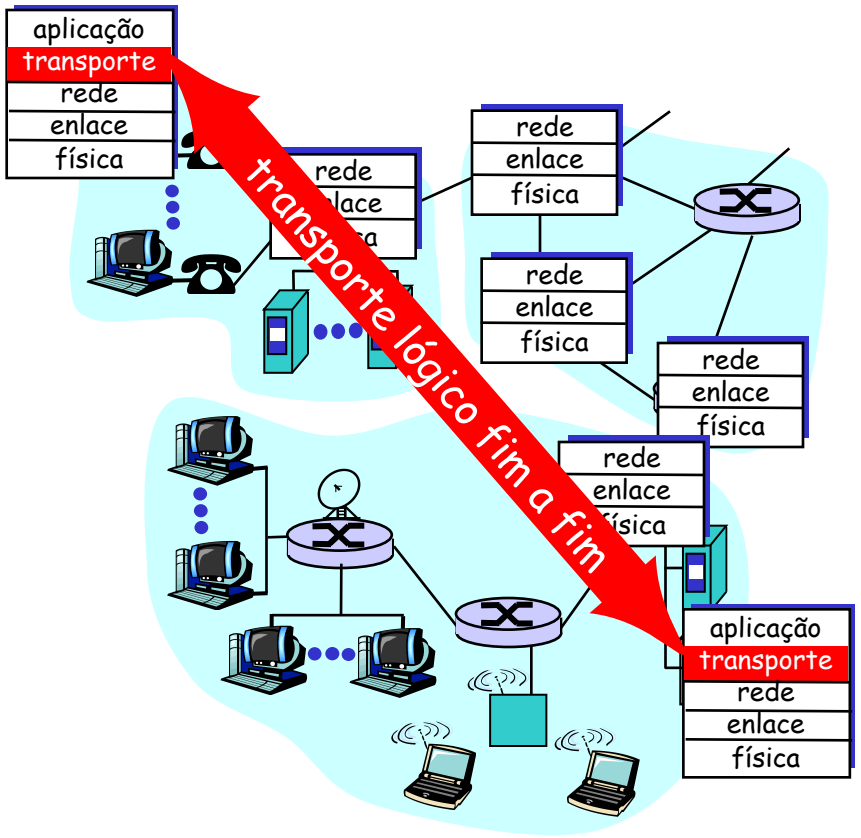
- entender os princípios atrás dos serviços da camada de transporte:
  - multiplexação/ demultiplexação
  - transferência confiável de dados
  - controle de fluxo
  - controle de congestionamento
- aprender sobre os protocolos da camada de transporte da Internet:
  - UDP: transporte não orientado a conexões
  - TCP: transporte orientado a conexões
  - Controle de congestionamento do TCP

Conteúdo do Capítulo 3

- 3.1 Introdução e serviços de camada de transporte
- 3.2 Multiplexação e demultiplexação
- 3.3 Transporte não orientado para conexão: UDP
- 3.4 Princípios da transferência confiável de dados
- 3.5 Transporte orientado para conexão: TCP
- 3.6 Princípios de controle de congestionamento
- 3.7 Controle de congestionamento no TCP

Serviços e protocolos de transporte

- fornecem *comunicação lógica* entre processos de aplicação executando em diferentes hospedeiros
- os protocolos de transporte são executados nos sistemas finais:
  - lado transmissor: quebra as mensagens da aplicação em segmentos, repassa-os para a camada de rede
  - lado receptor: remonta as mensagens a partir dos segmentos, repassa-as para a camada de aplicação
- existe mais de um protocolo de transporte disponível para as aplicações
  - Internet: TCP e UDP



Camadas de Transporte x rede

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

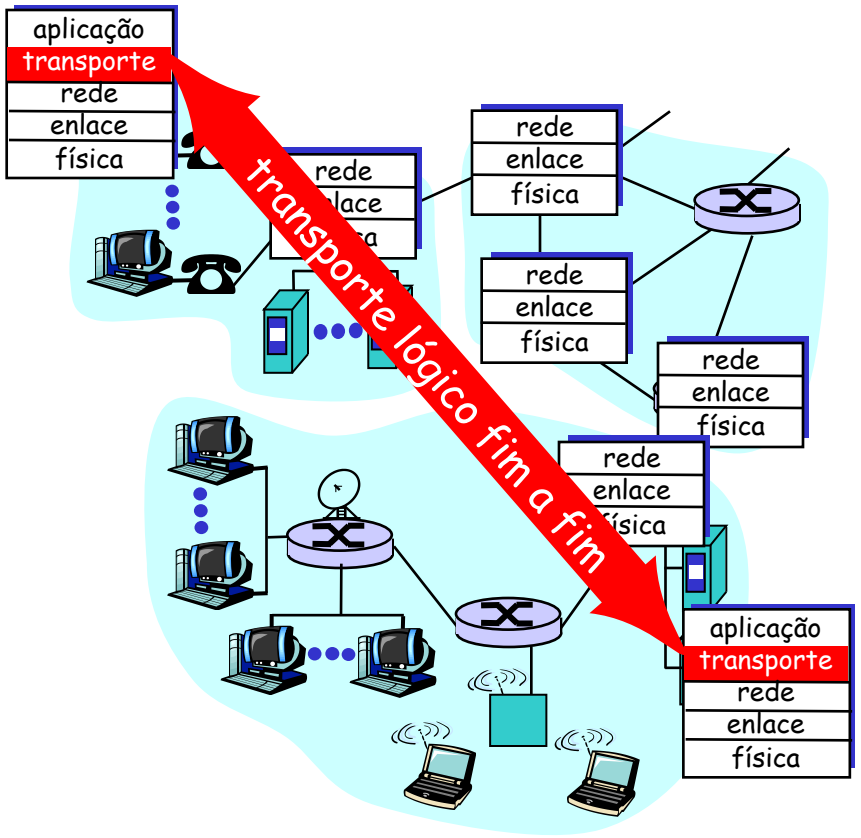
Analogia doméstica:

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

- hospedeiros = casas
- processos = crianças
- mensagens da apl. = cartas nos envelopes
- protocolo de transporte = Ana e Bill que demultiplexam para suas crianças
- protocolo da camada de rede = serviço postal

Protocolos da camada de transporte Internet

- entrega confiável, ordenada (TCP)
  - controle de congestionamento
  - controle de fluxo
  - estabelecimento de conexão ("setup")
- entrega não confiável, não ordenada: UDP
  - extensão sem "gorduras" do "melhor esforço" do IP
- serviços não disponíveis:
  - garantias de atraso máximo
  - garantias de largura de banda mínima



Conteúdo do Capítulo 3

- 3.1 Introdução e serviços de camada de transporte
- 3.2 Multiplexação e demultiplexação
- 3.3 Transporte não orientado para conexão: UDP
- 3.4 Princípios da transferência confiável de dados
- 3.5 Transporte orientado para conexão: TCP
- 3.6 Princípios de controle de congestionamento
- 3.7 Controle de congestionamento no TCP

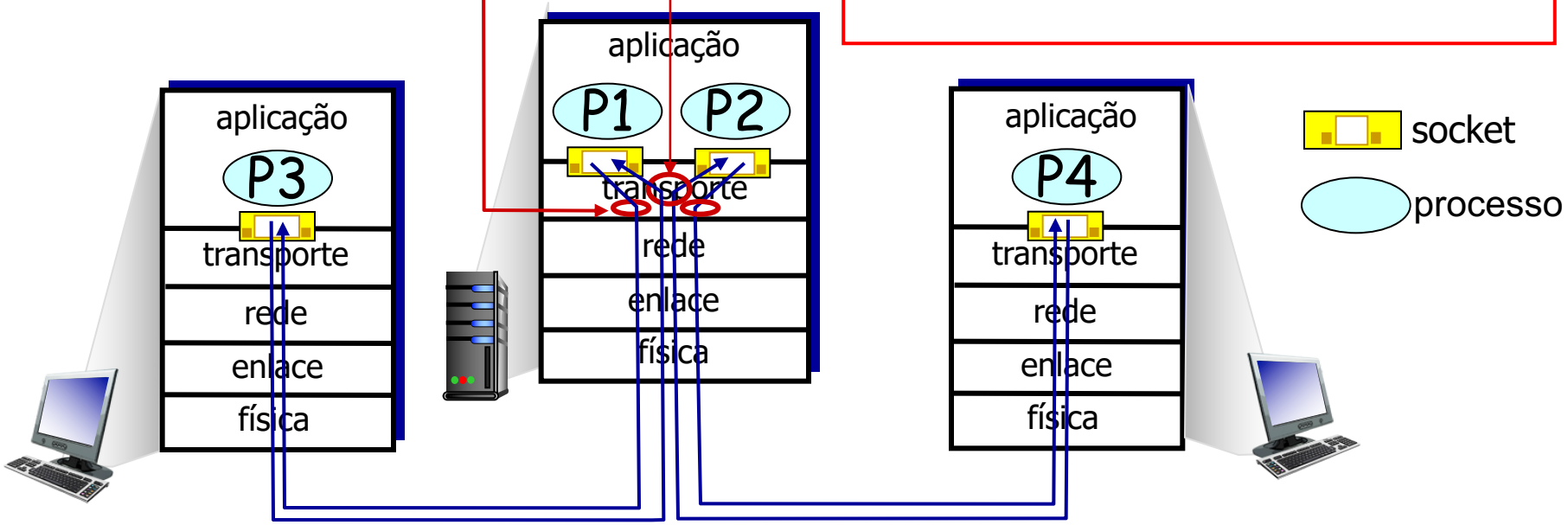
Multiplexação/demultiplexação

Multiplexação no transm.:

reúne dados de muitos sockets, adiciona o cabeçalho de transporte (usado posteriormente para a demultiplexação)

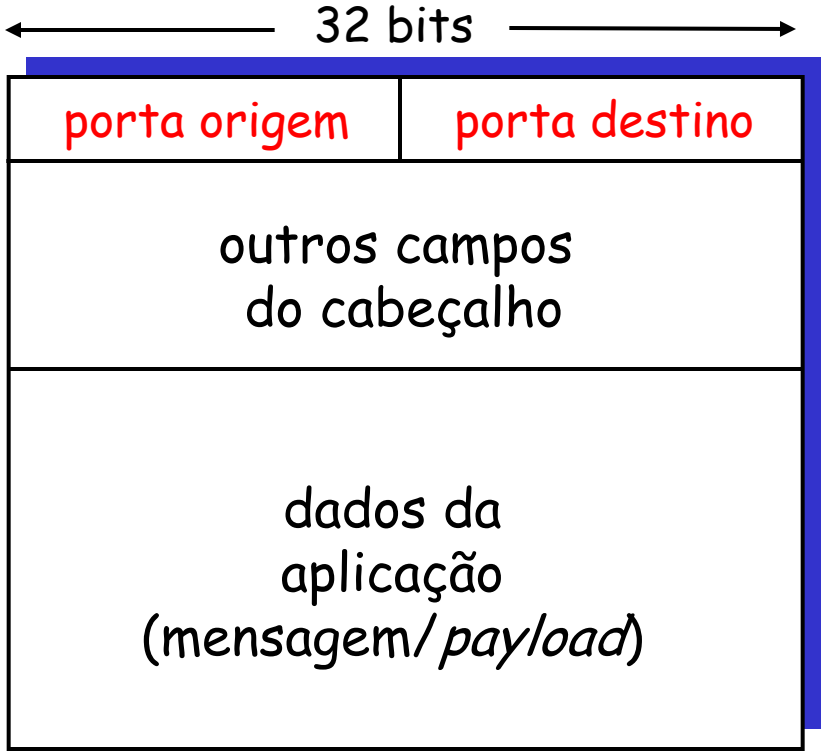
Demultiplexação no receptor:

Usa info do cabeçalho para entregar os segmentos recebidos aos sockets corretos



Como funciona a demultiplexação

- computador recebe os datagramas IP
  - cada datagrama possui os endereços IP da origem e do destino
  - cada datagrama transporta um segmento da camada de transporte
  - cada segmento possui números das portas origem e destino
- O hospedeiro usa os endereços IP e os números das portas para direcionar o segmento ao socket apropriado



formato de segmento TCP/UDP

Demultiplexação não orientada a conexões

- *Lembrete*: socket criado possui número de porta local ao host:

```
DatagramSocket mySocket1 = new
   DatagramSocket(12534);
```

- *Lembrete*: ao criar um datagrama para enviar para um socket UDP, deve especificar:
  - Endereço IP de destino
  - Número da porta de destino

- Quando o hospedeiro recebe o segmento UDP:
  - verifica no. da porta de destino no segmento
  - encaminha o segmento UDP para o socket com aquele no. de porta

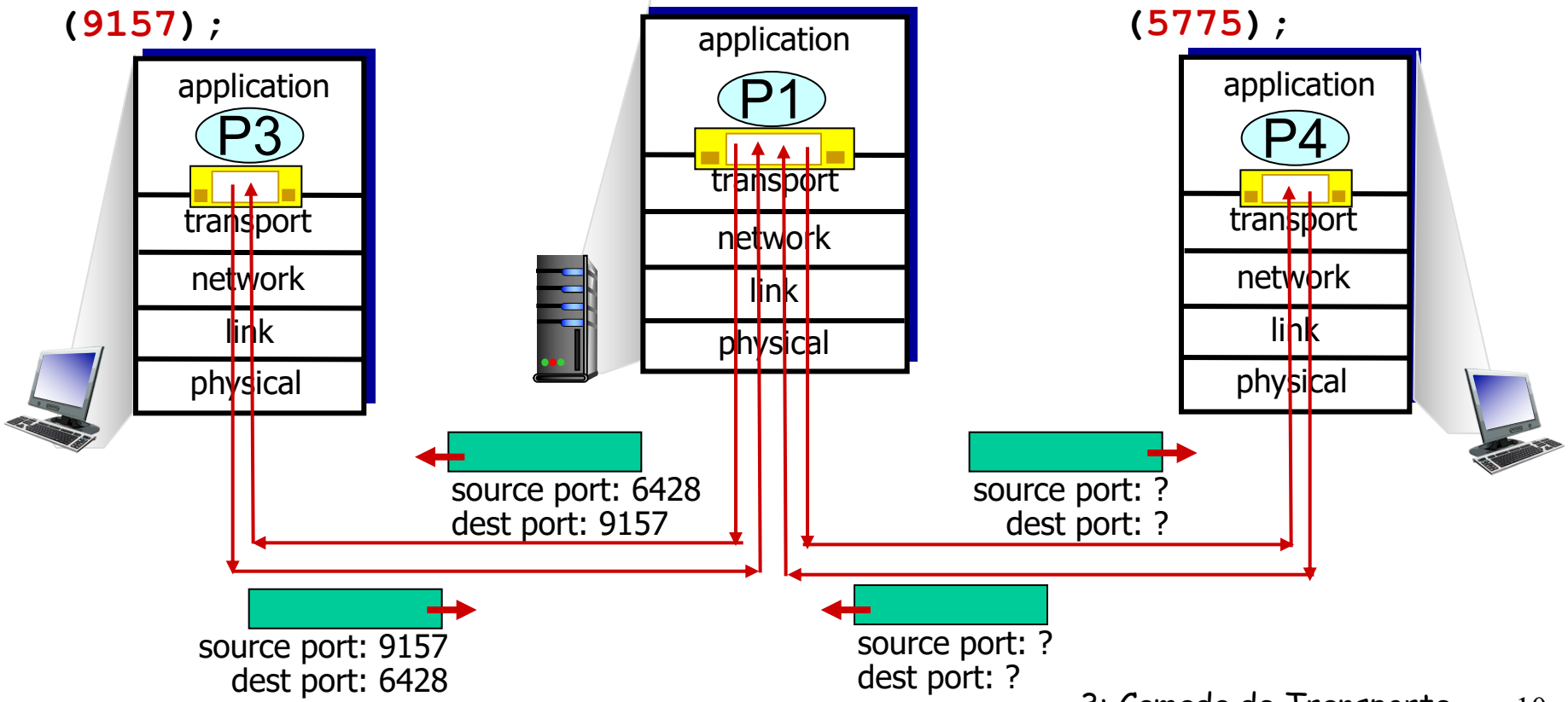
- Datagramas IP com mesmo no. de porta destino, mas diferentes endereços IP origem e/ou números de porta origem podem ser encaminhados para o mesmo socket no destino

Demultiplexação não orientada a conexões: exemplo

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

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

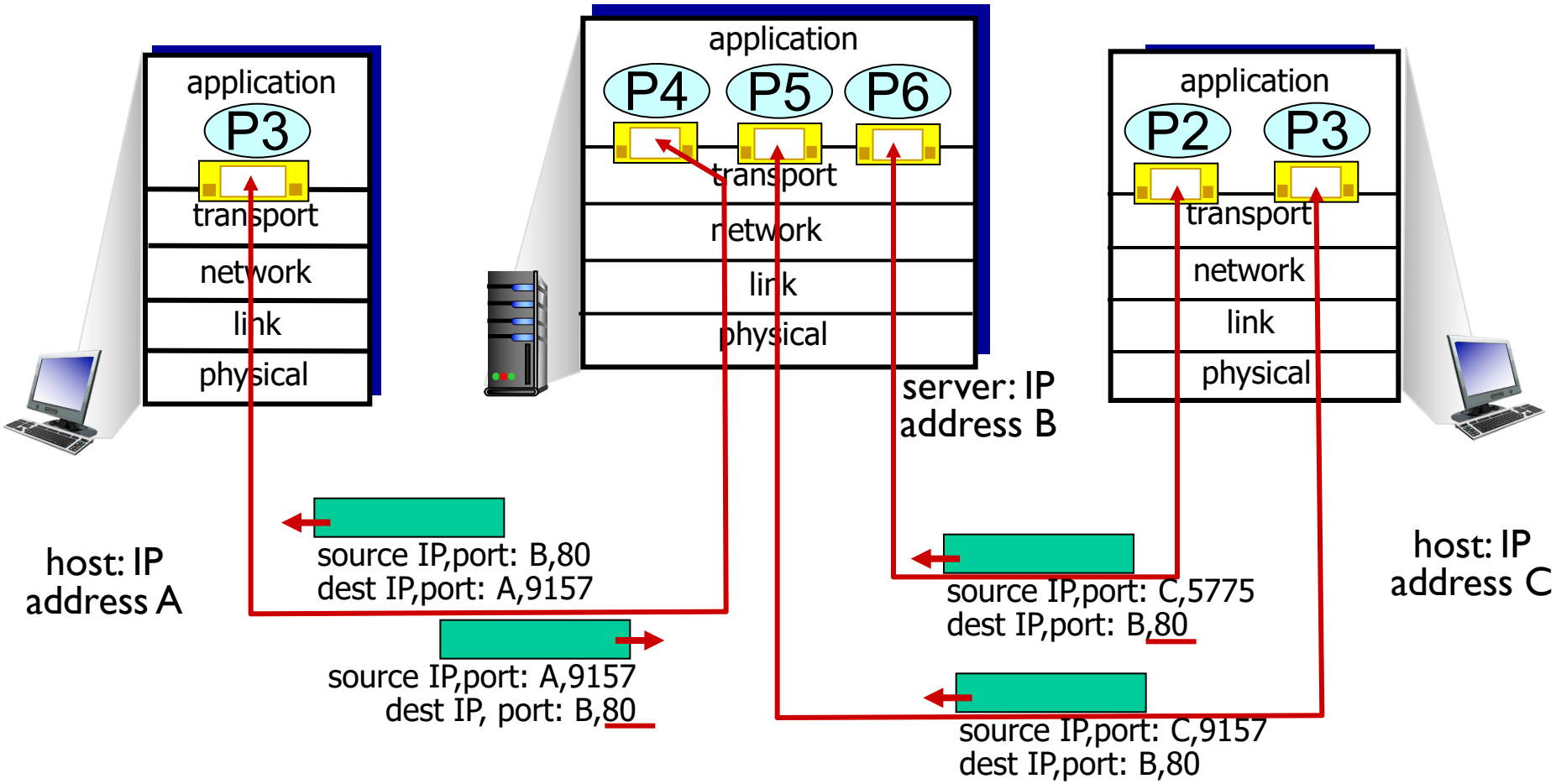
```
DatagramSocket
 mySocket1 = new
 DatagramSocket
 (5775);
```



Demultiplexação Orientada a Conexões

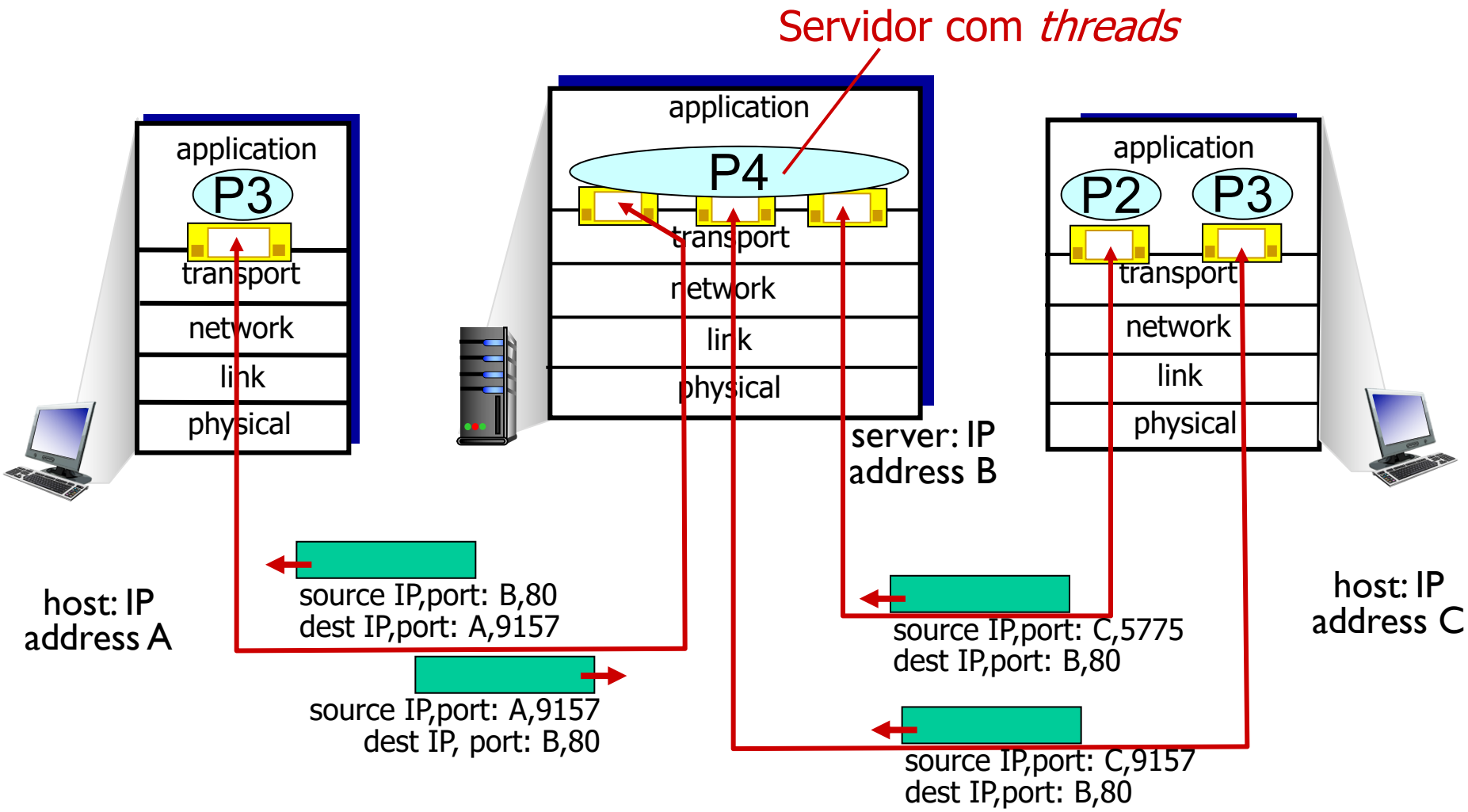
- Socket TCP identificado pela quádrupla:
  - endereço IP origem
  - número da porta origem
  - endereço IP destino
  - número da porta destino
- Demultiplexação: receptor usa todos os quatro valores para direcionar o segmento para o socket apropriado
- Servidor pode dar suporte a muitos sockets TCP simultâneos:
  - cada socket é identificado pela sua própria quádrupla
- Servidores Web têm sockets diferentes para cada conexão de cliente
  - HTTP não persistente terá sockets diferentes para cada pedido

Demultiplexação Orientada a Conexões: exemplo



três segmentos, todos destinados ao endereço IP: B, dest port: 80 são demultiplexados para *sockets* distintos

Demultiplexação Orientada a Conexões: Servidor Web com Threads



Conteúdo do Capítulo 3

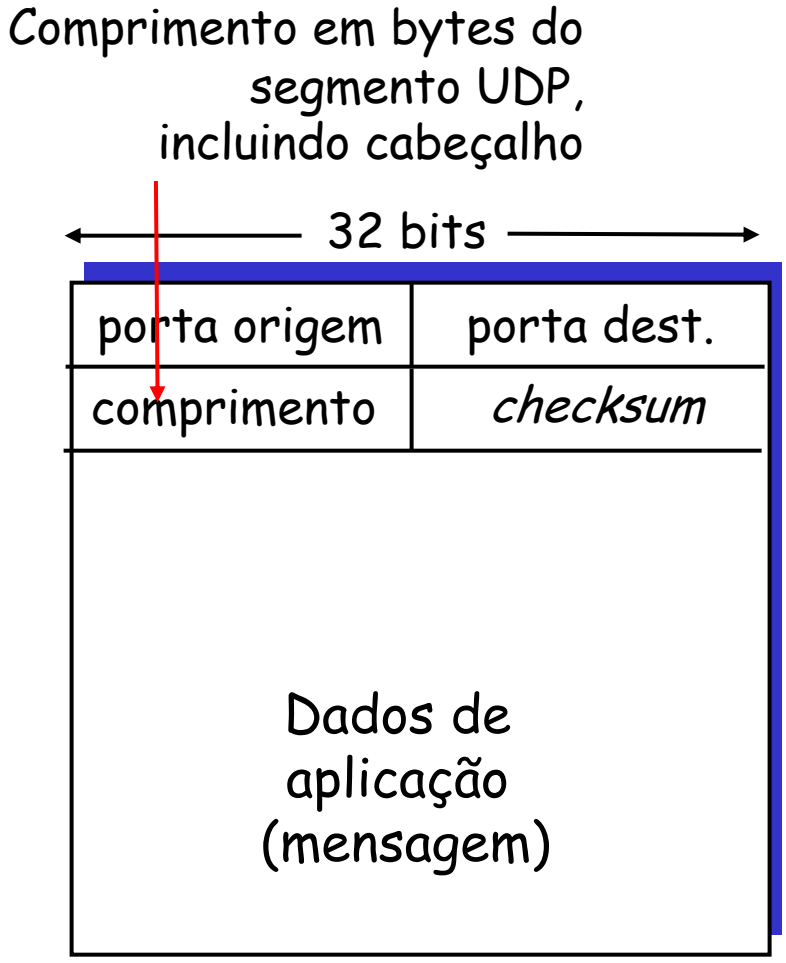
- 3.1 Introdução e serviços de camada de transporte
- 3.2 Multiplexação e demultiplexação
- 3.3 Transporte não orientado para conexão: UDP
- 3.4 Princípios da transferência confiável de dados
- 3.5 Transporte orientado para conexão: TCP
- 3.6 Princípios de controle de congestionamento
- 3.7 Controle de congestionamento no TCP

UDP: User Datagram Protocol [RFC 768]

- Protocolo de transporte da Internet mínimo, "sem gorduras",
- Serviço "melhor esforço", segmentos UDP podem ser:
  - perdidos
  - entregues à aplicação fora de ordem
- *sem conexão:*
  - não há saudação inicial entre o remetente e o receptor UDP
  - tratamento independente para cada segmento UDP

- Uso do UDP:
  - aplicações de *streaming* multimídia (tolerante a perdas, sensível a taxas)
  - DNS
  - SNMP
- transferência confiável sobre UDP:
  - adiciona confiabilidade na camada de aplicação
  - recuperação de erros específica da aplicação

UDP: Cabeçalho do segmento



Formato do segmento UDP

Por quê existe um UDP?

- elimina estabelecimento de conexão (que pode causar retardo)
- simples: não mantém "estado" da conexão nem no remetente, nem no receptor
- cabeçalho de segmento reduzido
- Não há controle de congestionamento: UDP pode transmitir tão rápido quanto desejado (e possível)

Soma de Verificação (*checksum*) UDP

Objetivo: detectar "erros" (ex.: bits trocados) no segmento transmitido

Transmissor:

- trata conteúdo do segmento como sequência de inteiros de 16-bits
- checksum: soma (adição usando complemento de 1) do conteúdo do segmento
- transmissor coloca *complemento do valor da soma* no campo *checksum* do UDP

Receptor:

- calcula *checksum* do segmento recebido
- verifica se o *checksum* calculado bate com o valor recebido:
  - NÃO - erro detectado
  - SIM - nenhum erro detectado. *Mas ainda pode ter erros?* Veja depois ....

Exemplo do *Checksum* Internet

- Exemplo: adição de 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
transbordo	(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	1	0	0	0	0	1	1

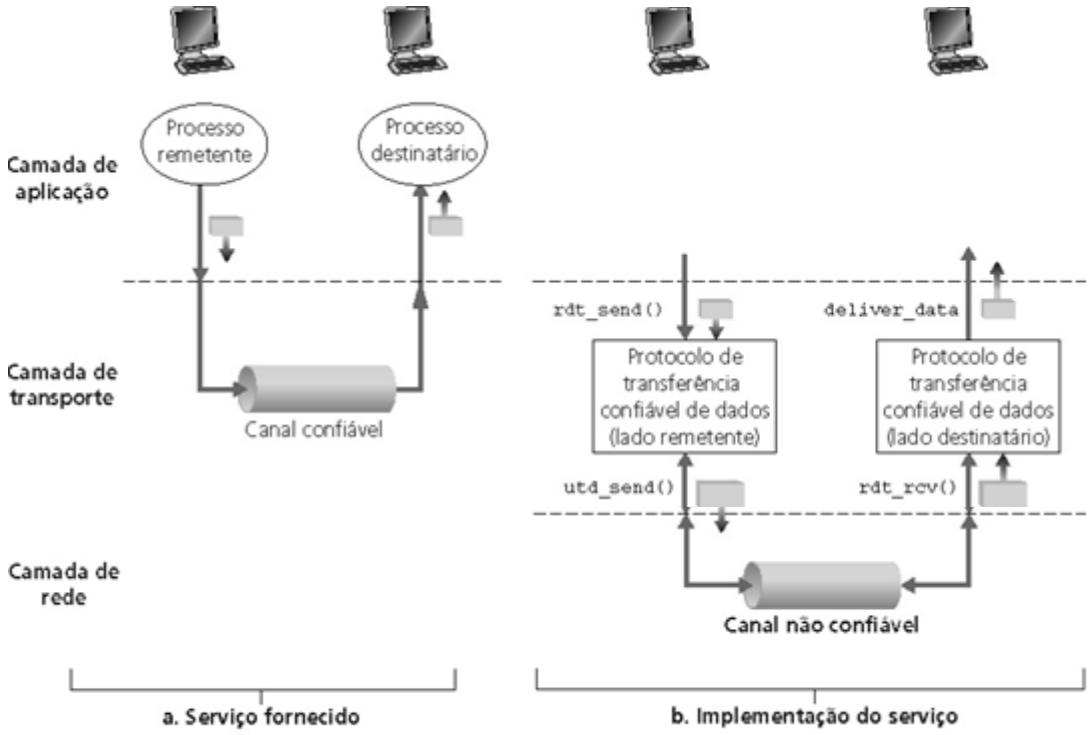
Note que: ao adicionar números, o transbordo (vai um) do bit mais significativo deve ser adicionado ao resultado

Conteúdo do Capítulo 3

- 3.1 Introdução e serviços de camada de transporte
- 3.2 Multiplexação e demultiplexação
- 3.3 Transporte não orientado para conexão: UDP
- 3.4 Princípios da transferência confiável de dados
- 3.5 Transporte orientado para conexão: TCP
- 3.6 Princípios de controle de congestionamento
- 3.7 Controle de congestionamento no TCP

Princípios de Transferência confiável de dados (rdt)

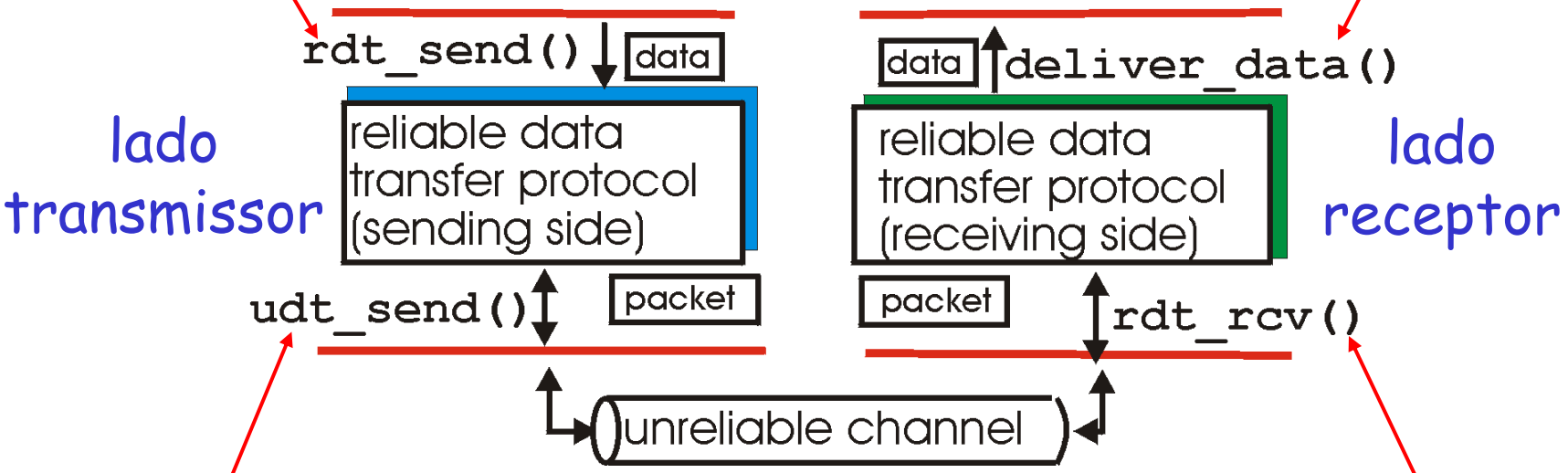
- importante nas camadas de transporte e de enlace
- na lista dos 10 tópicos mais importantes em redes!
- características do canal não confiável determinam a complexidade de um protocolo de transferência confiável de dados (rdt)



Transferência confiável: o ponto de partida

`rdt_send()`: chamada de cima, (ex.: pela apl.). Passa dados p/ serem entregues à camada sup. do receptor

`deliver_data()`: chamada pelo **rdt** para entregar dados p/ camada superior



`udt_send()`: chamada pelo **rdt** para transferir um pacotes para o receptor sobre um canal não confiável

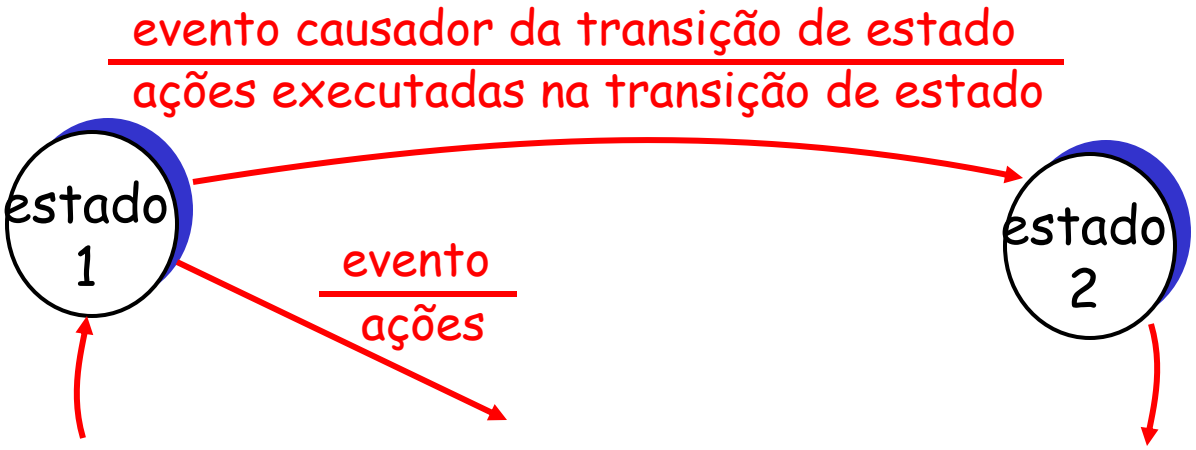
`rdt_rcv()`: chamada quando pacote chega no lado receptor do canal

Transferência confiável: o ponto de partida

Iremos:

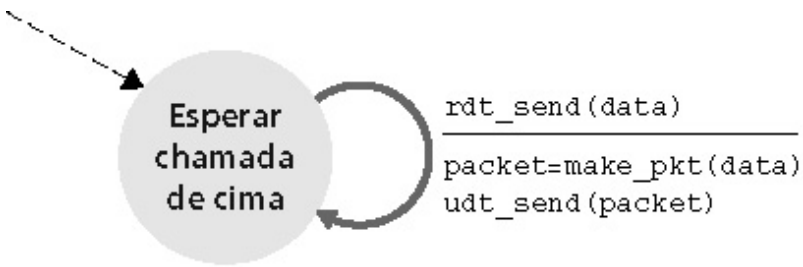
- desenvolver incrementalmente os lados transmissor e receptor de um protocolo confiável de transferência de dados (rdt)
- considerar apenas fluxo unidirecional de dados
  - mas info de controle flui em ambos os sentidos!
- Usar máquinas de estados finitos (FSM) p/ especificar os protocolos transmissor e receptor

estado: neste "estado" o próximo estado é determinado unicamente pelo próximo evento

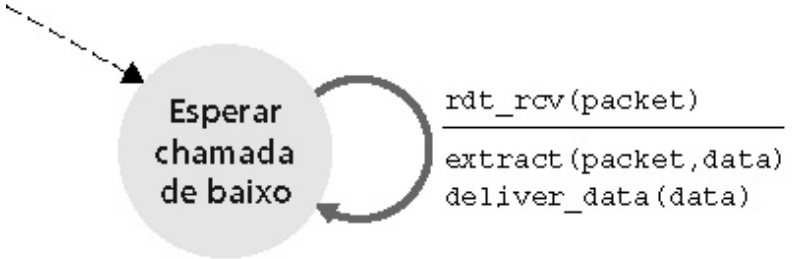


rdt1.0: transferência confiável sobre canais confiáveis

- canal de transmissão perfeitamente confiável
  - não há erros de bits
  - não há perda de pacotes
- FSMs separadas para transmissor e receptor:
  - transmissor envia dados pelo canal subjacente
  - receptor lê os dados do canal subjacente



a. rdt1.0: lado remetente



b. rdt1.0: lado destinatário

rdt2.0: canal com erros de bits

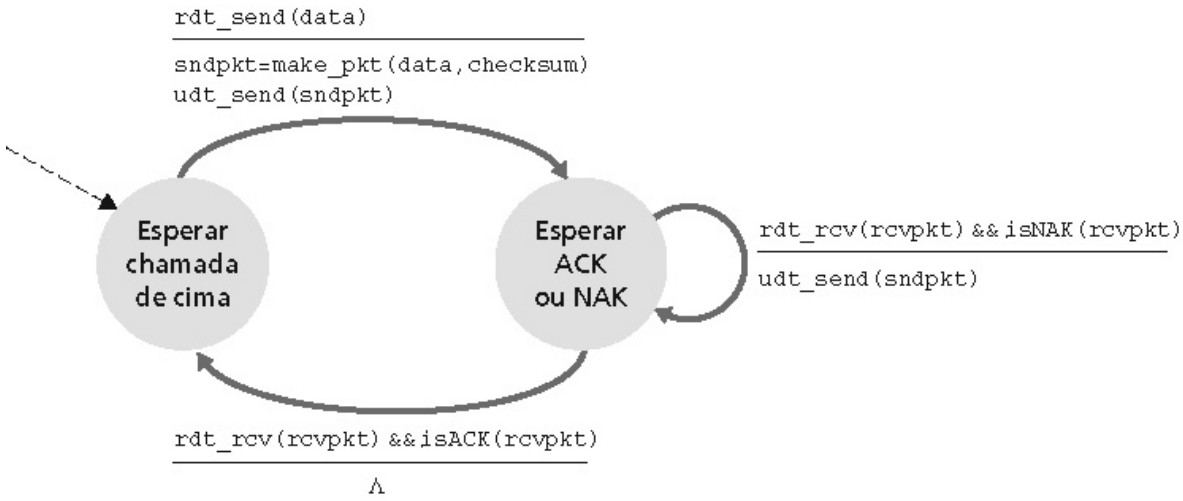
- canal subjacente pode trocar valores dos bits num pacote
  - lembrete: *checksum* UDP pode detectar erros de bits
- *a* questão: como recuperar esses erros?

Como as pessoas recuperam "erros" durante uma conversa?

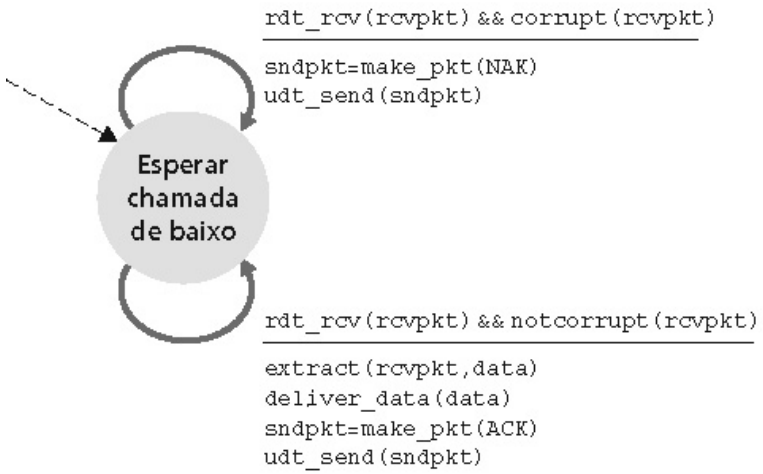
rdt2.0: canal com erros de bits

- canal subjacente pode trocar valores dos bits num pacote
  - lembre-se: checksum UDP pode detectar erros de bits
- *a* questão: como recuperar esses erros?
  - reconhecimentos *(ACKs):* receptor avisa explicitamente ao transmissor que o pacote foi recebido corretamente
  - reconhecimentos negativos *(NAKs):* receptor avisa explicitamente ao transmissor que o pacote tinha erros
  - transmissor reenvia o pacote ao receber um NAK
- novos mecanismos no `rdt2.0` (em relação ao `rdt1.0`):
  - detecção de erros
  - Realimentação (*feedback*): mensagens de controle (ACK,NAK) do receptor para o transmissor

rdt2.0: especificação da FSM



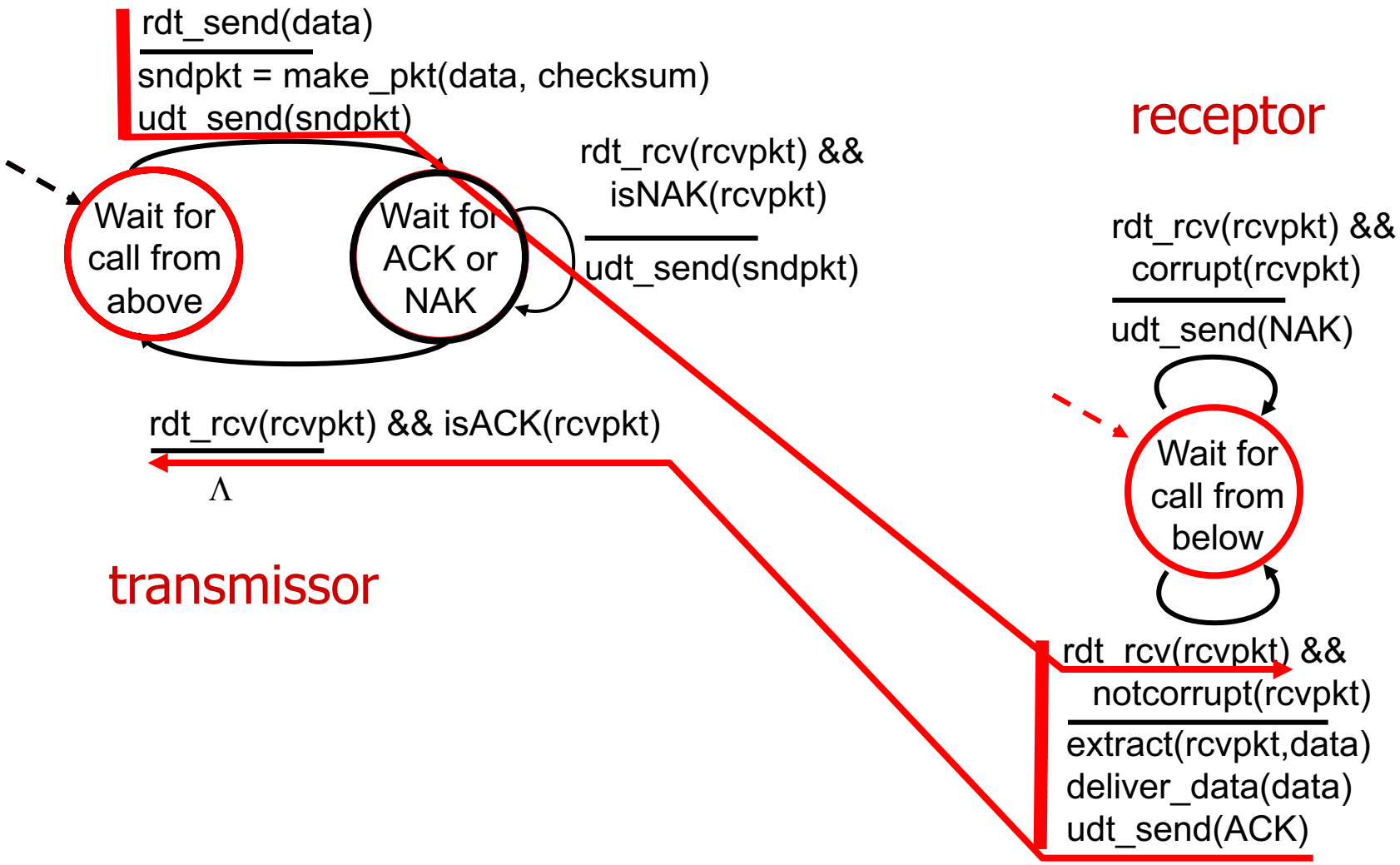
a. rdt2.0: lado remetente



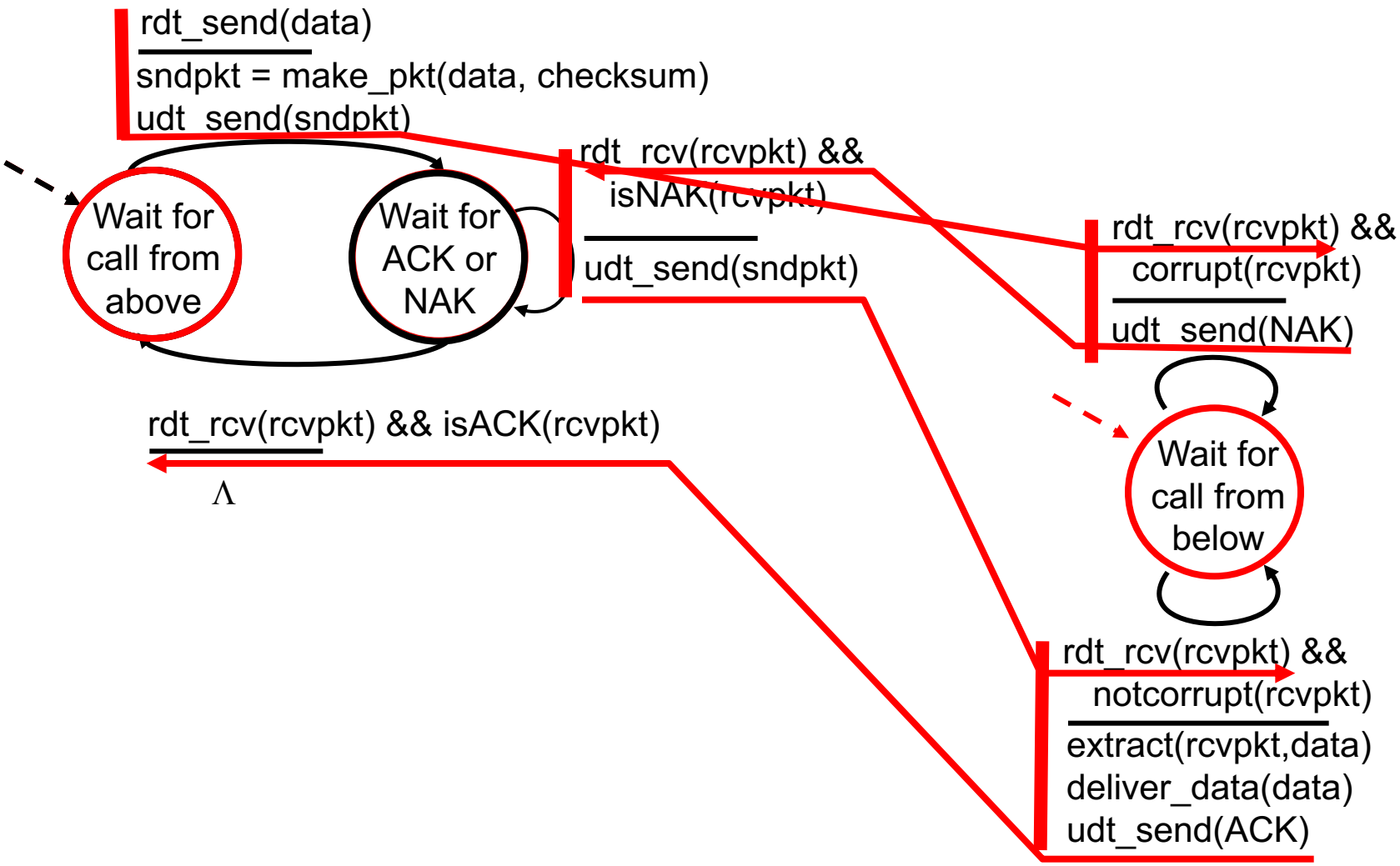
b. rdt2.0: lado destinatário

Animação no slide seguinte!

rdt2.0: operação com ausência de erros



rdt2.0: cenário de erro



rdt2.0 tem uma falha fatal!

O que acontece se o ACK/NAK for corrompido?

- Transmissor não sabe o que se passou no receptor!
- não pode apenas retransmitir: possibilidade de pacotes duplicados

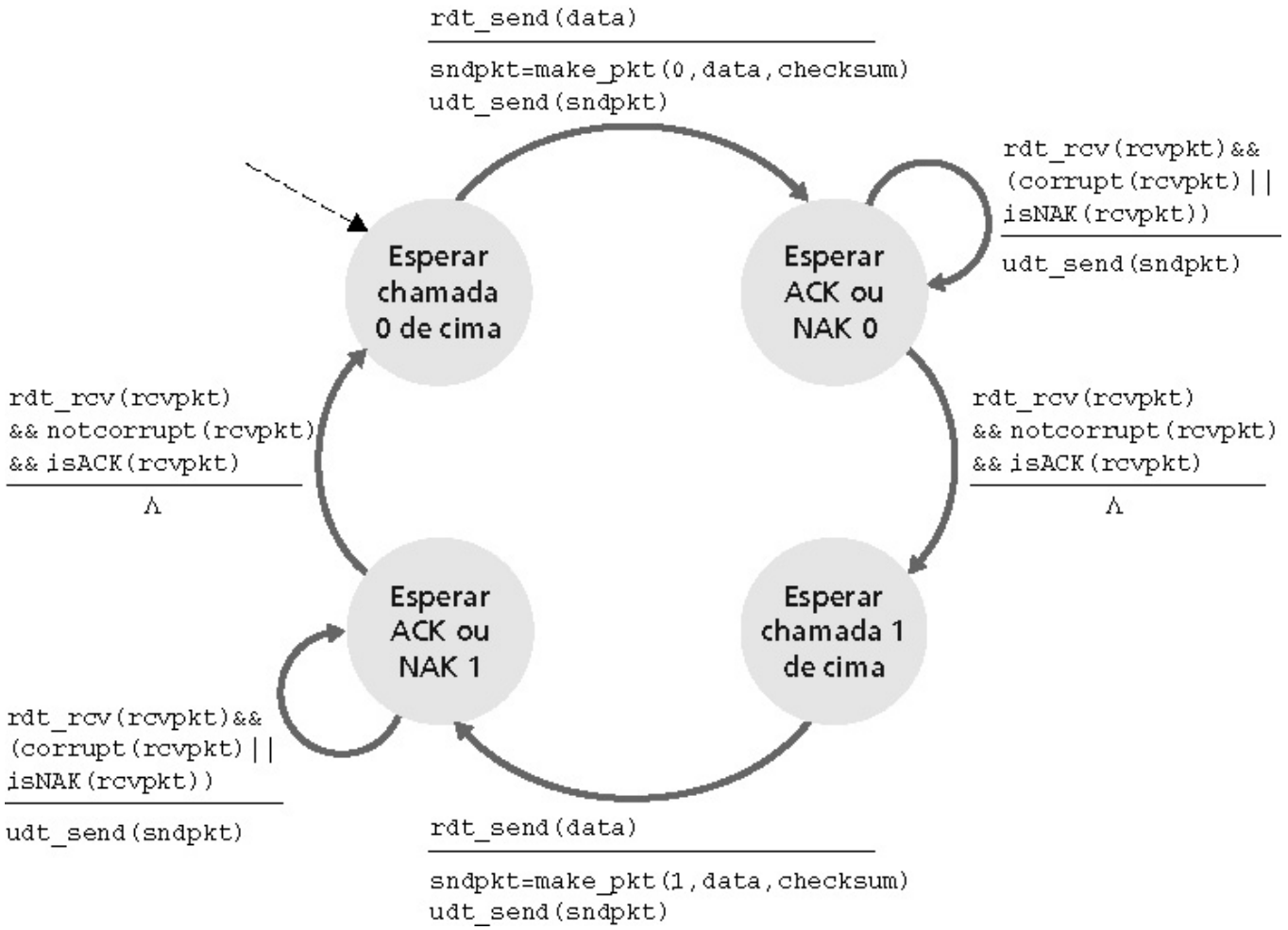
Lidando c/ duplicatas:

- transmissor retransmite o último pacote se ACK/NAK chegar com erro
- transmissor inclui *número de sequência* em cada pacote
- receptor descarta (não entrega a aplicação) pacotes duplicados

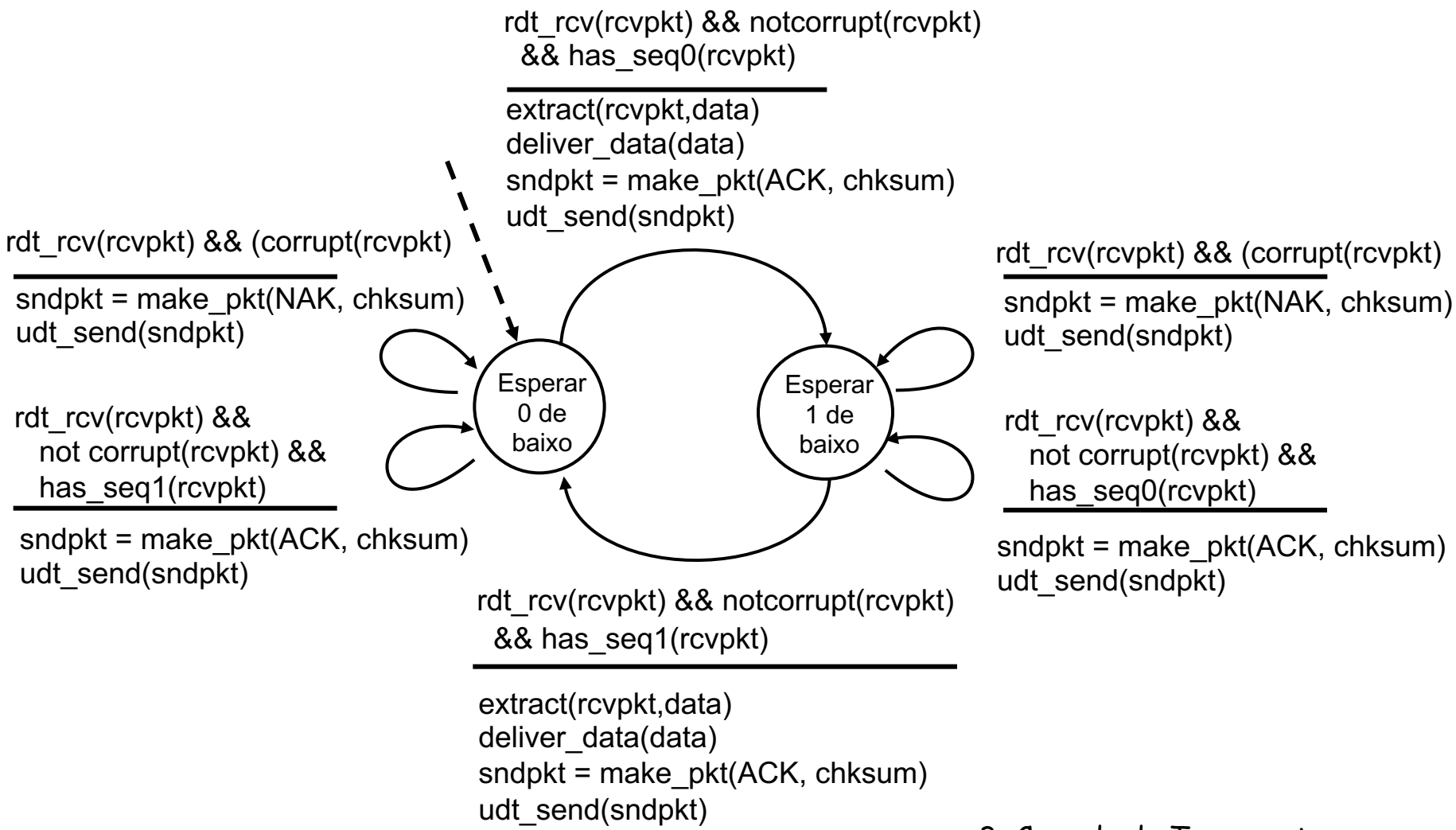
pare e espera

Transmissor envia um pacote, e então aguarda resposta do receptor

rdt2.1: transmissor, trata ACK/NAKs corrompidos



rdt2.1: receptor, trata ACK/NAKs corrompidos



rdt2.1: discussão

Transmissor:

- no. de seq no pacote
- bastam dois nos. de seq. (0,1). Por quê?
- deve verificar se ACK/NAK recebidos estão corrompidos
- duplicou o no. de estados
  - estado deve "lembrar" se pacote "esperado" deve ter no. de seq. 0 ou 1

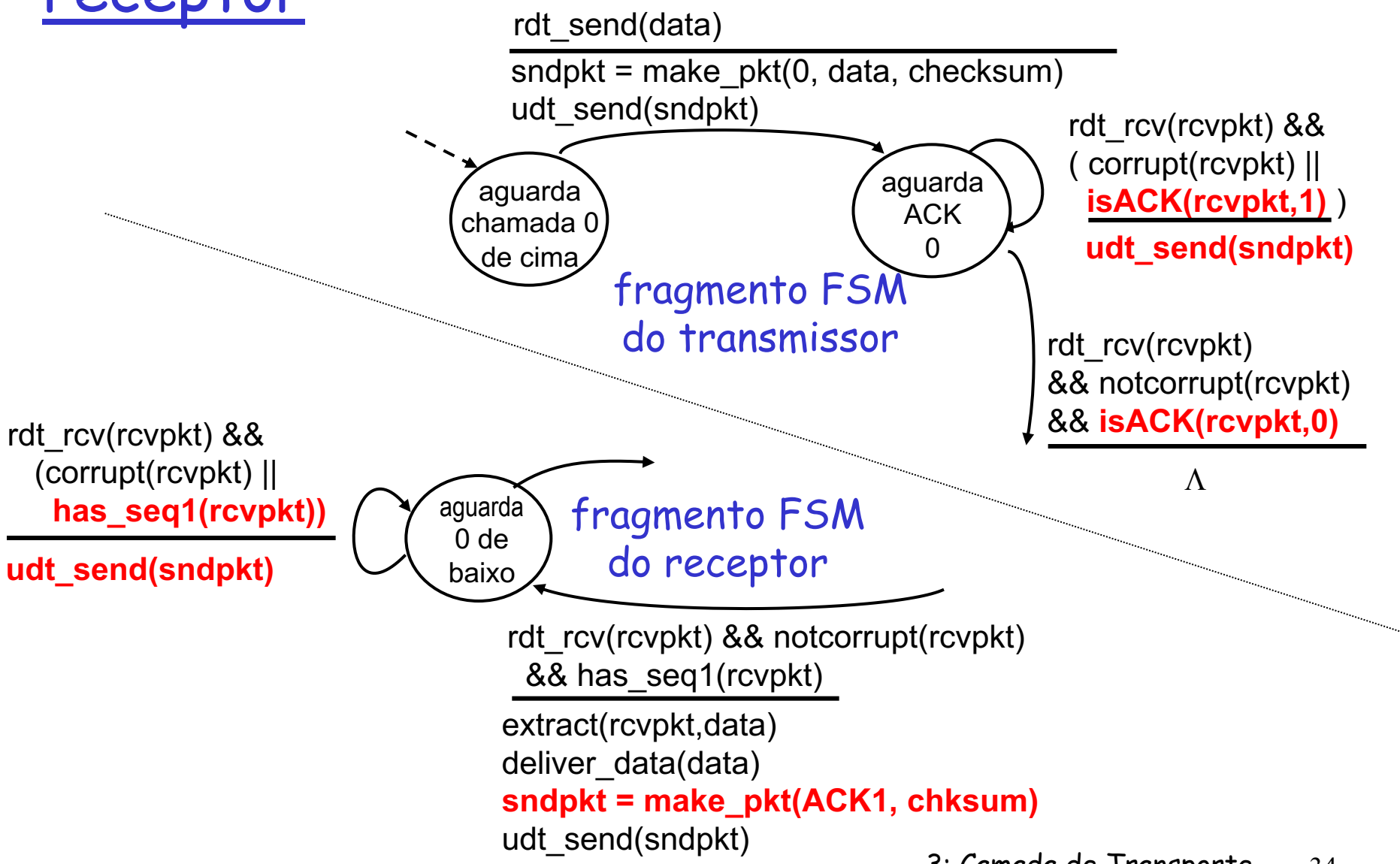
Receptor:

- deve verificar se o pacote recebido é uma duplicata
  - estado indica se no. de seq. esperado é 0 ou 1
- nota: receptor não tem como saber se último ACK/NAK foi recebido bem pelo transmissor

rdt2.2: um protocolo sem NAKs

- mesma funcionalidade do rdt2.1, usando apenas ACKs
- ao invés de NAK, receptor envia ACK para último pacote recebido sem erro
  - receptor deve incluir *explicitamente* no. de seq do pacote reconhecido
- ACKs duplicados no transmissor resultam na mesma ação do NAK: *retransmissão do pacote atual*

rdt2.2: fragmentos do transmissor e receptor



rdt3.0: canais com erros *e* perdas

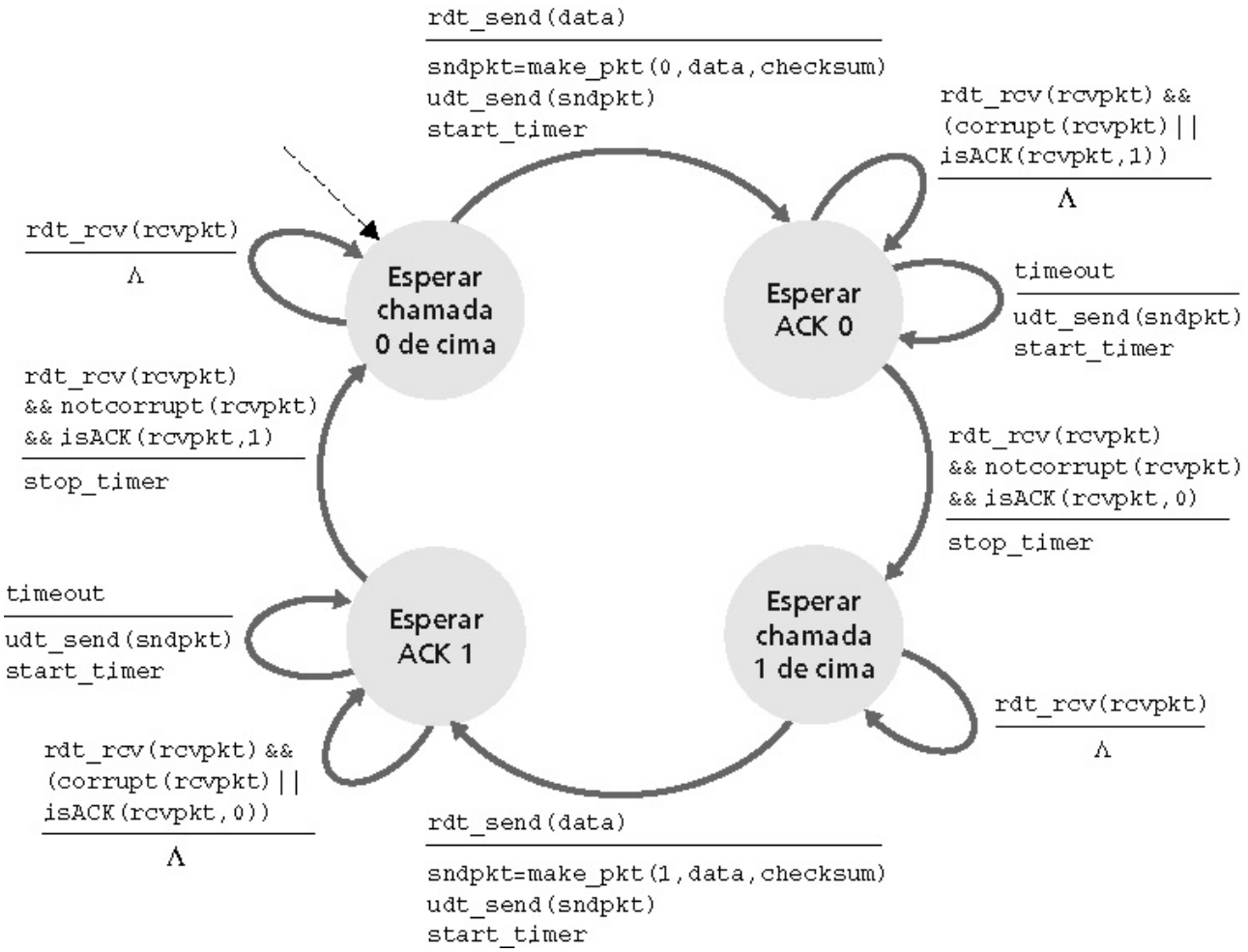
Nova hipótese: canal de transmissão também pode perder pacotes (dados ou ACKs)

- checksum, no. de seq., ACKs, retransmissões podem ajudar, mas não são suficientes

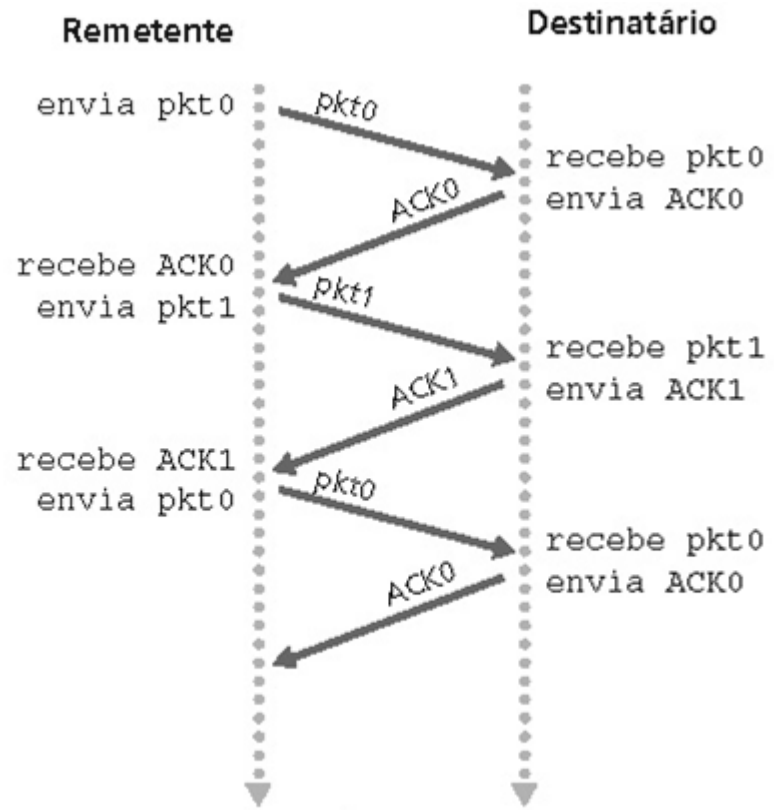
Abordagem: transmissor aguarda um tempo "razoável" pelo ACK

- retransmite se nenhum ACK for recebido neste intervalo
- se pacote (ou ACK) estiver apenas atrasado (e não perdido):
  - retransmissão será duplicata, mas uso de no. de seq. já cuida disto
  - receptor deve especificar no. de seq do pacote sendo reconhecido
- requer temporizador

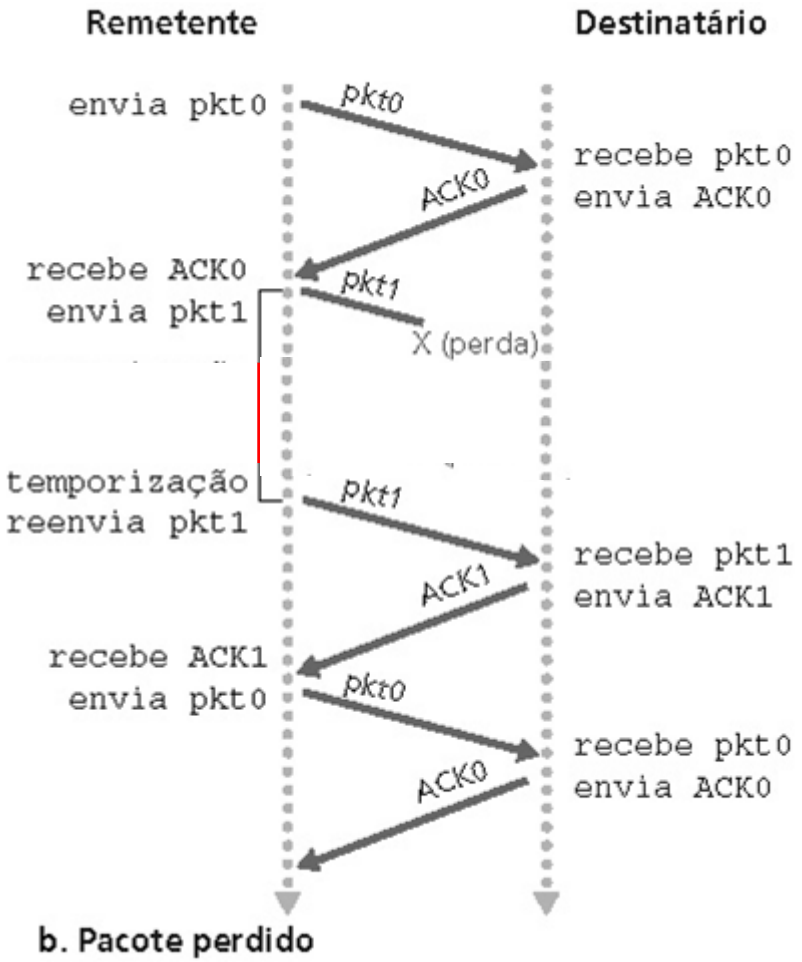
Transmissor rdt3.0



rdt3.0 em ação

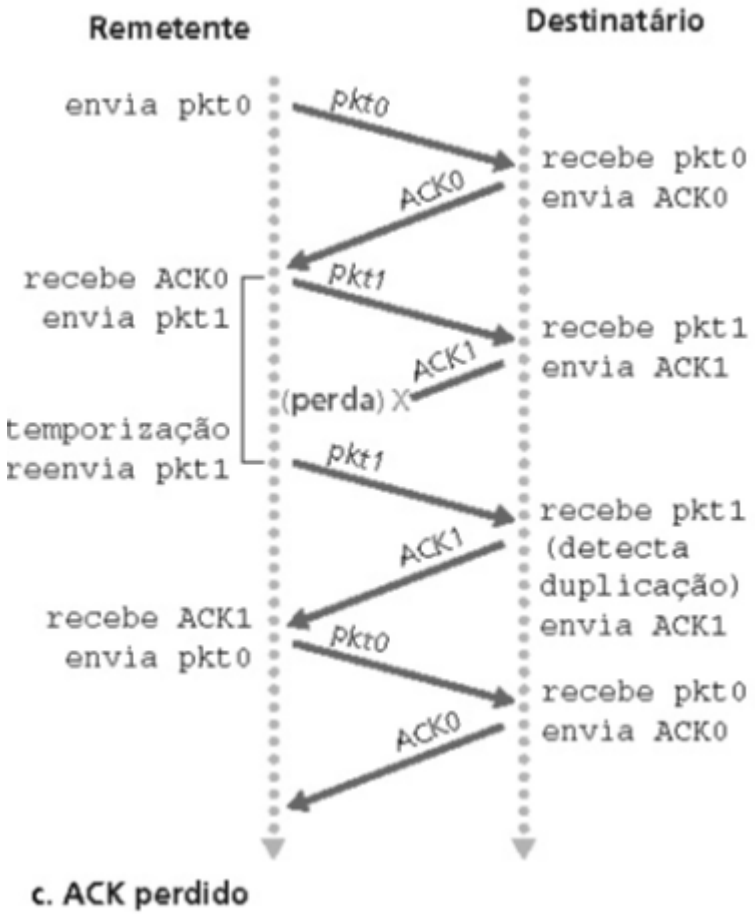


a. Operação sem perda

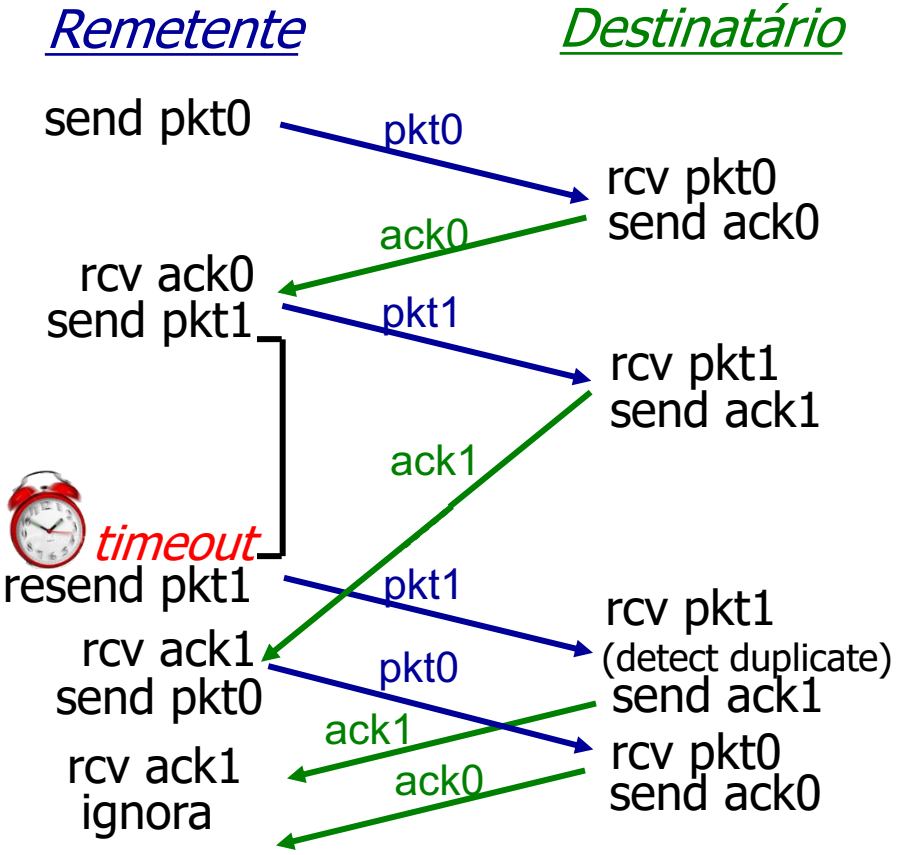


b. Pacote perdido

rdt3.0 em ação



c. ACK perdido



(d) retransmissão prematura

Desempenho do rdt3.0

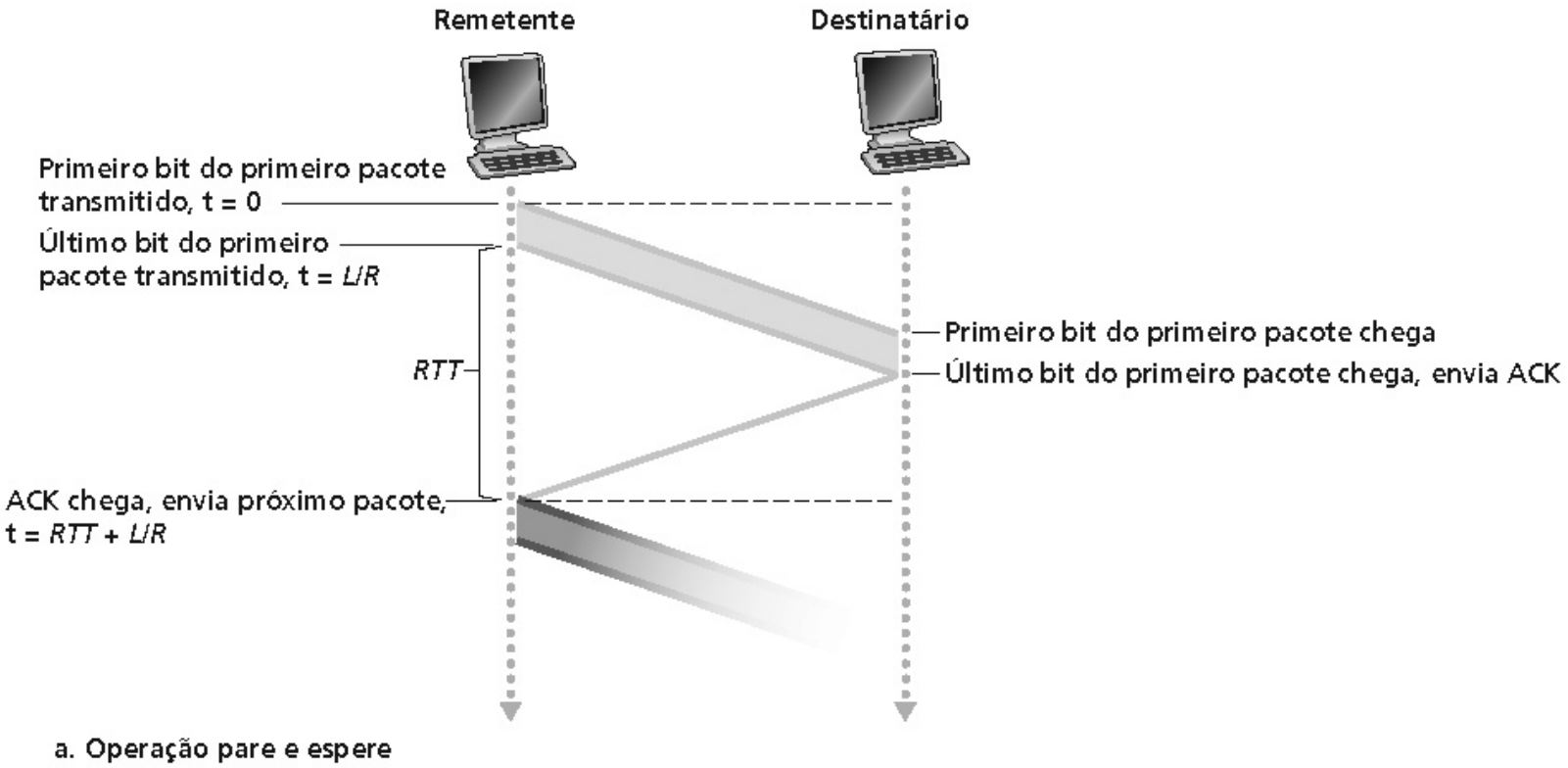
- rdt3.0 funciona, porém seu desempenho é sofrível
- Exemplo: enlace de 1 Gbps, retardo fim a fim de 15 ms, pacote de 8000 bits:

$$d_{trans} = \frac{L}{R} = \frac{8000\text{bits}}{10^9\,\text{bps}} = 8\,\text{microsegundos}$$

$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{0{,}008}{30{,}008} = 0{,}00027$$

- pac. de 1KB a cada 30 mseg -> vazão de 33kB/seg num enlace de 1 Gbps
- protocolo limita uso dos recursos físicos!

rdt3.0: operação *pare e espere*



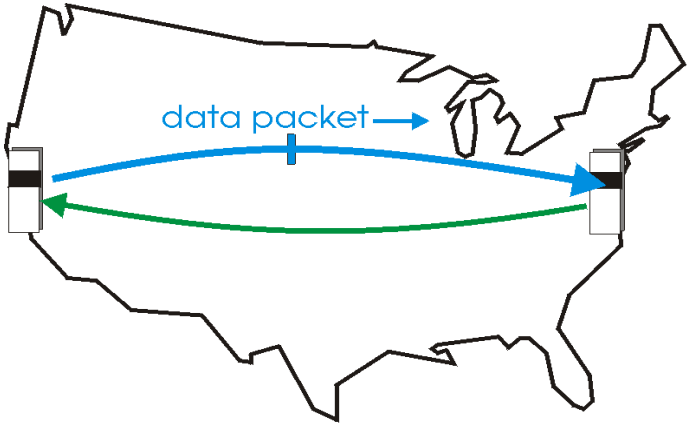
a. Operação pare e espere

$$U_{\text{tx}} = \frac{L/R}{\text{RTT} + L/R} = \frac{0{,}008}{30{,}008} = 0{,}00027$$

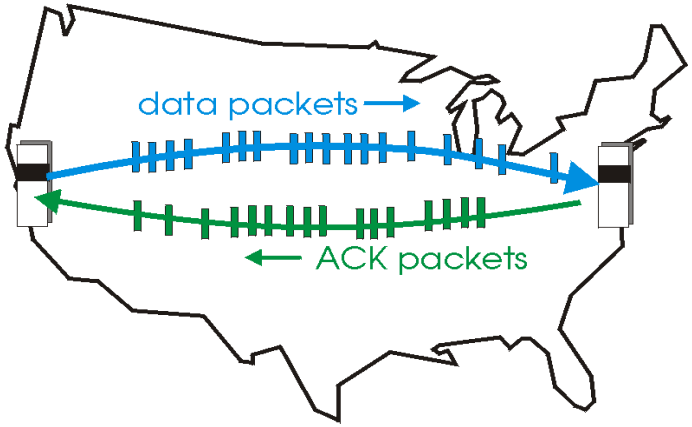
Protocolos com paralelismo (*pipelining*)

Paralelismo (*pipelining*): transmissor envia vários pacotes em sequência, todos esperando para serem reconhecidos

- faixa de números de sequência deve ser aumentada
- Armazenamento no transmissor e/ou no receptor



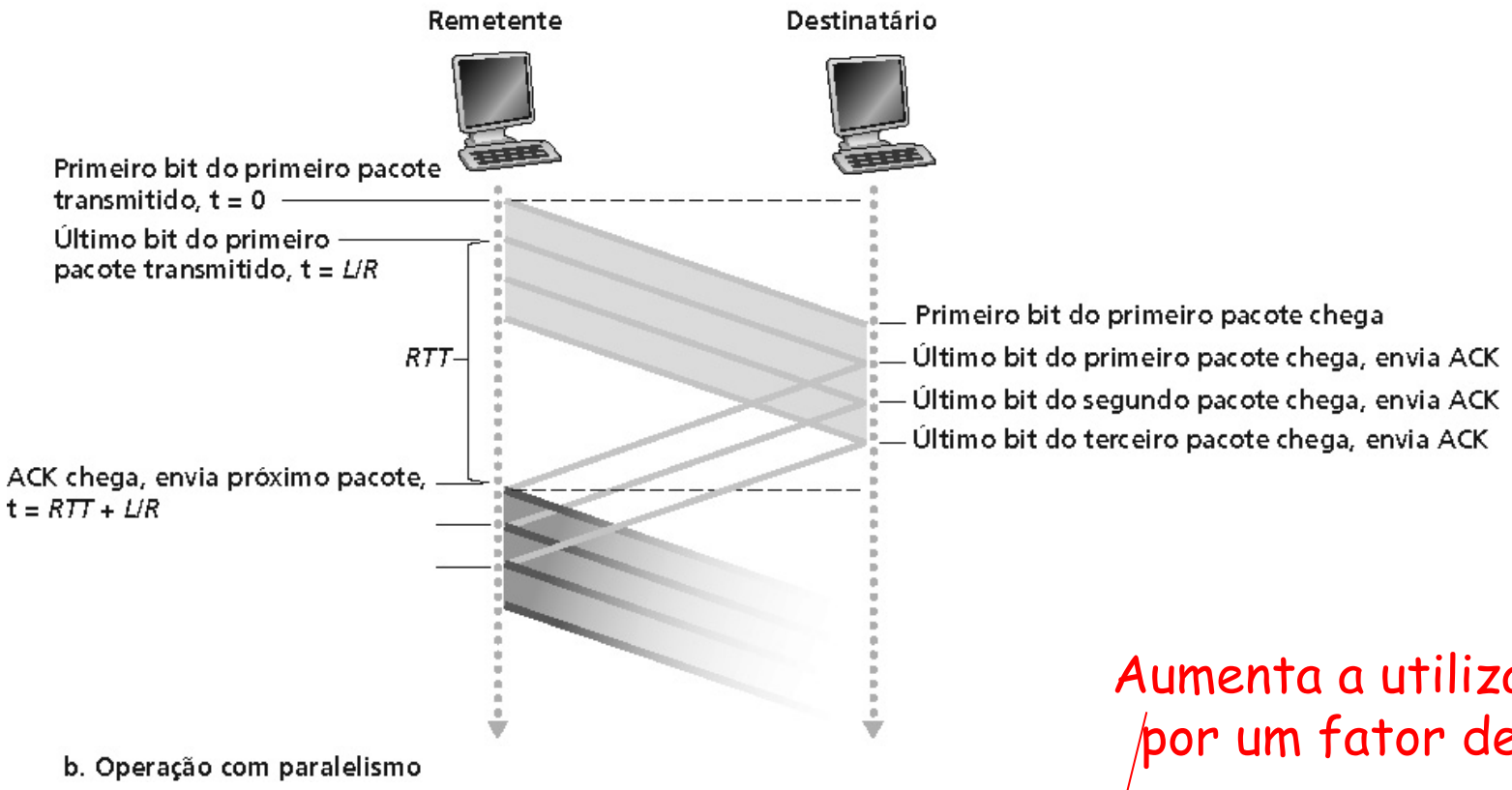
(a) operação do protocolo pare e espere



(a) operação do protocolo com paralelismo

- Duas formas genéricas de protocolos com paralelismo: *Go-back-N, retransmissão seletiva*

Paralelismo: aumento da utilização



Aumenta a utilização por um fator de 3!

$$U_{\text{tx}} = \frac{3 \times L/R}{\text{RTT} + L/R} = \frac{0{,}024}{30{,}008} = 0{,}00081$$

Protocolos com Paralelismo

Go-back-N:

- O transmissor pode ter até N pacotes não reconhecidos no "tubo"
- Receptor envia apenas acks cumulativos
  - Não reconhece pacote se houver falha de seq.
- Transmissor possui um temporizador para o pacote mais antigo ainda não reconhecido
  - Se o temporizador estourar, retransmite *todos* os pacotes ainda não reconhecidos.

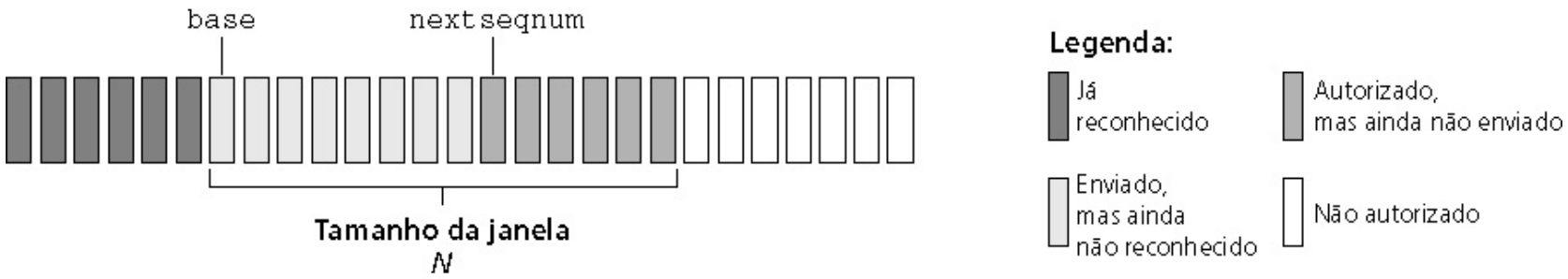
Retransmissão seletiva:

- O transmissor pode ter até N pacotes não reconhecidos no "tubo"
- Receptor envia acks individuais para cada pacote
- Transmissor possui um temporizador para cada pacote ainda não reconhecido
  - Se o temporizador estourar, retransmite apenas o pacote correspondente.

Go-back-N (GBN)

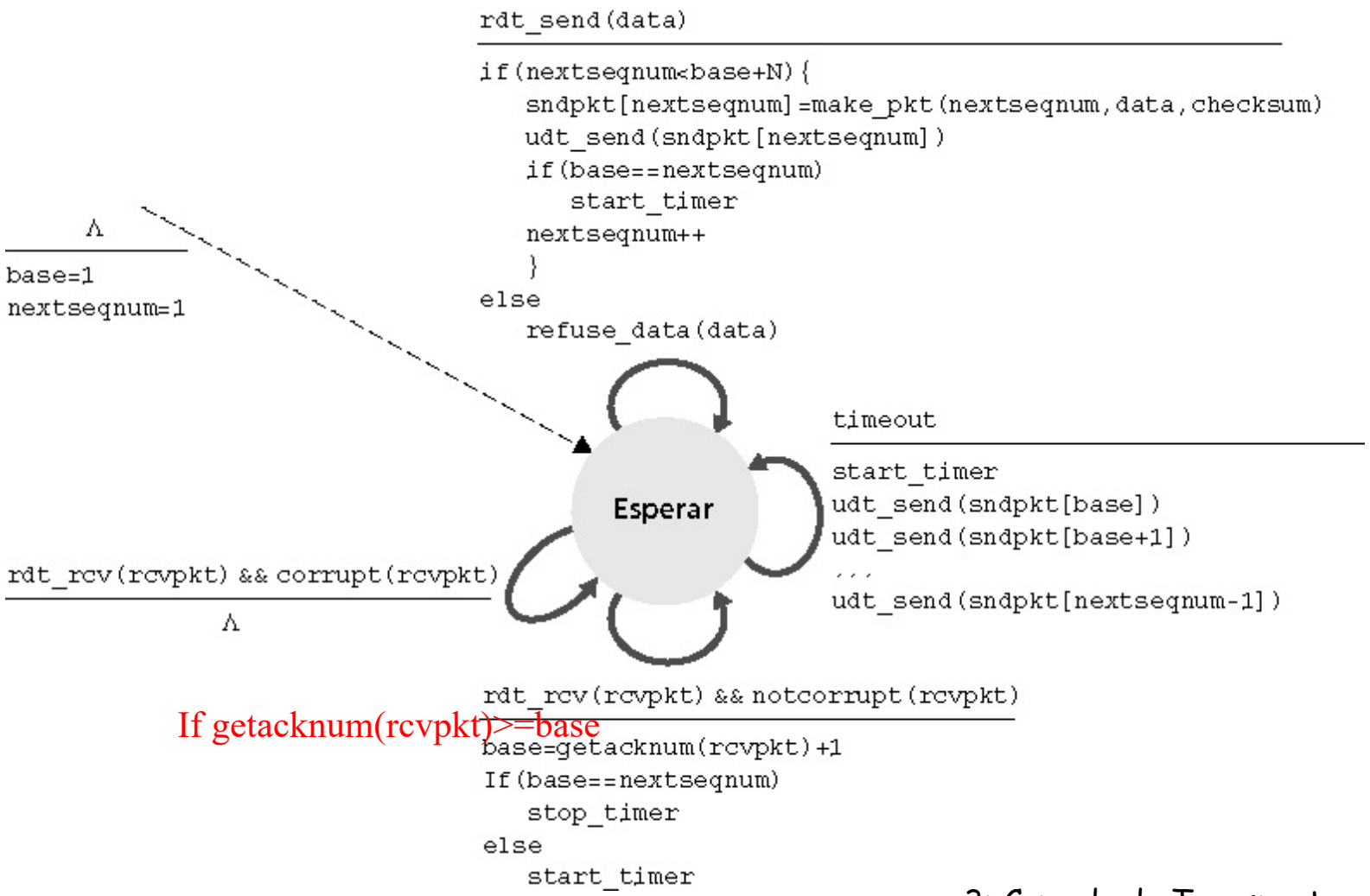
Transmissor:

- no. de seq. de k-bits no cabeçalho do pacote
- admite "janela" de até N pacotes consecutivos não reconhecidos

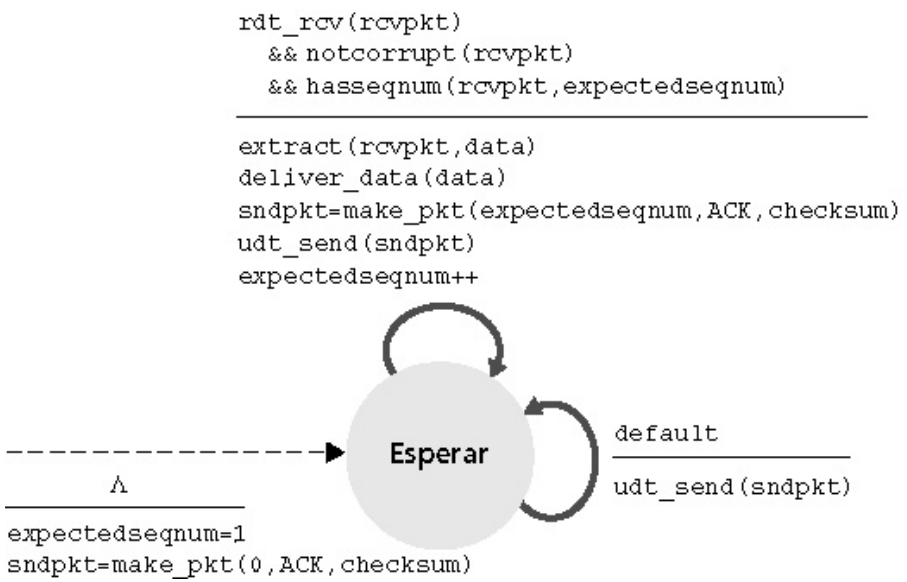


- ACK(n): reconhece todos pacotes, até e inclusive no. de seq n - "ACK/reconhecimento cumulativo"
  - pode receber ACKs duplicados (veja receptor)
- temporizador para o pacote mais antigo ainda não confirmado
- *Estouro do temporizador:* retransmite todos os pacotes pendentes.

GBN: FSM estendida para o transmissor

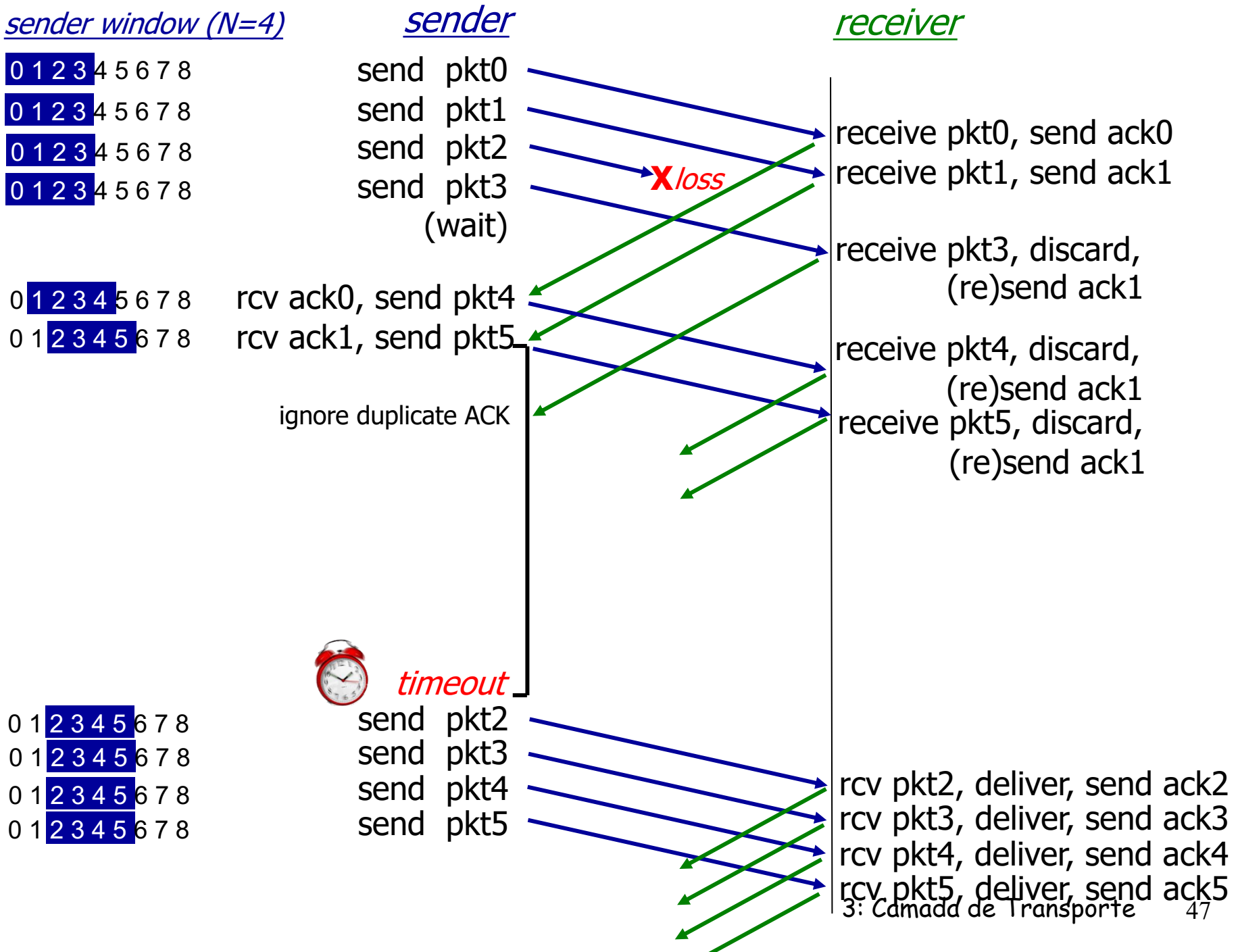


GBN: FSM estendida para o receptor



receptor simples:

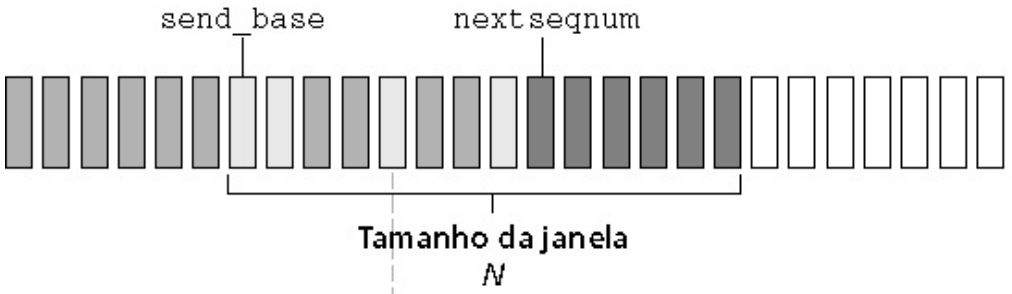
- usa apenas ACK: sempre envia ACK para pacote recebido corretamente com o maior no. de seq. *em-ordem*
  - pode gerar ACKs duplicados
  - só precisa se lembrar do **`expectedseqnum`**
- pacotes fora de ordem:
  - descarta (não armazena) -> receptor não usa buffers!
  - reconhece pacote com o número de sequência mais alto em-ordem


sender window (N=4)
sender
receiver
0 1 2 3 4 5 6 7 8
send pkt0
send pkt1
send pkt2
send pkt3
(wait)
loss
receive pkt0, send ack0
receive pkt1, send ack1
receive pkt3, discard,
(re)send ack1
rcv ack0, send pkt4
rcv ack1, send pkt5
receive pkt4, discard,
(re)send ack1
ignore duplicate ACK
receive pkt5, discard,
(re)send ack1
timeout
send pkt2
send pkt3
send pkt4
send pkt5
rcv pkt2, deliver, send ack2
rcv pkt3, deliver, send ack3
rcv pkt4, deliver, send ack4
rcv pkt5, deliver, send ack5

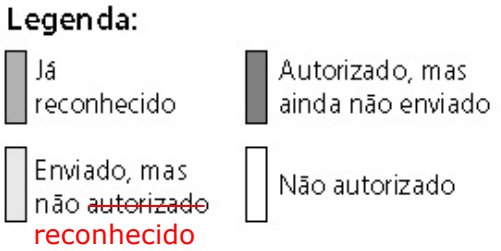
Retransmissão seletiva

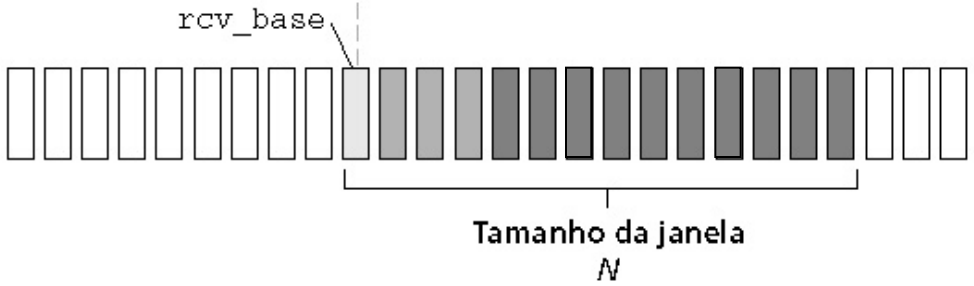
- receptor reconhece *individualmente* todos os pacotes recebidos corretamente
  - armazena pacotes no buffer, conforme necessário, para posterior entrega em-ordem à camada superior
- transmissor apenas reenvia pacotes para os quais um ACK não foi recebido
  - temporizador de remetente para cada pacote sem ACK
- janela do transmissão
  - N números de sequência consecutivos
  - outra vez limita números de sequência de pacotes enviados, mas ainda não reconhecidos

Retransmissão seletiva: janelas do transmissor e do receptor

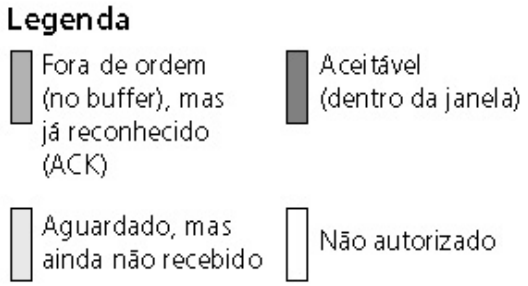


a. Visão que o remetente tem dos números de seqüência





b. Visão que o destinatário tem dos números de seqüência



Retransmissão seletiva

transmissor

dados de cima:

- se próx. no. de seq (n) disponível estiver na janela, envia o pacote e liga temporizador(n)

estouro do temporizador(n):

- reenvia pacote n, reinicia temporizador(n)

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

- marca pacote n como "recebido"
- se n for menor pacote não reconhecido, avança base da janela ao próx. no. de seq não reconhecido

receptor

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

- envia ACK(n)
- fora de ordem: armazena
- em ordem: entrega (tb. entrega pacotes armazenados em ordem), avança janela p/ próxima pacote ainda não recebido

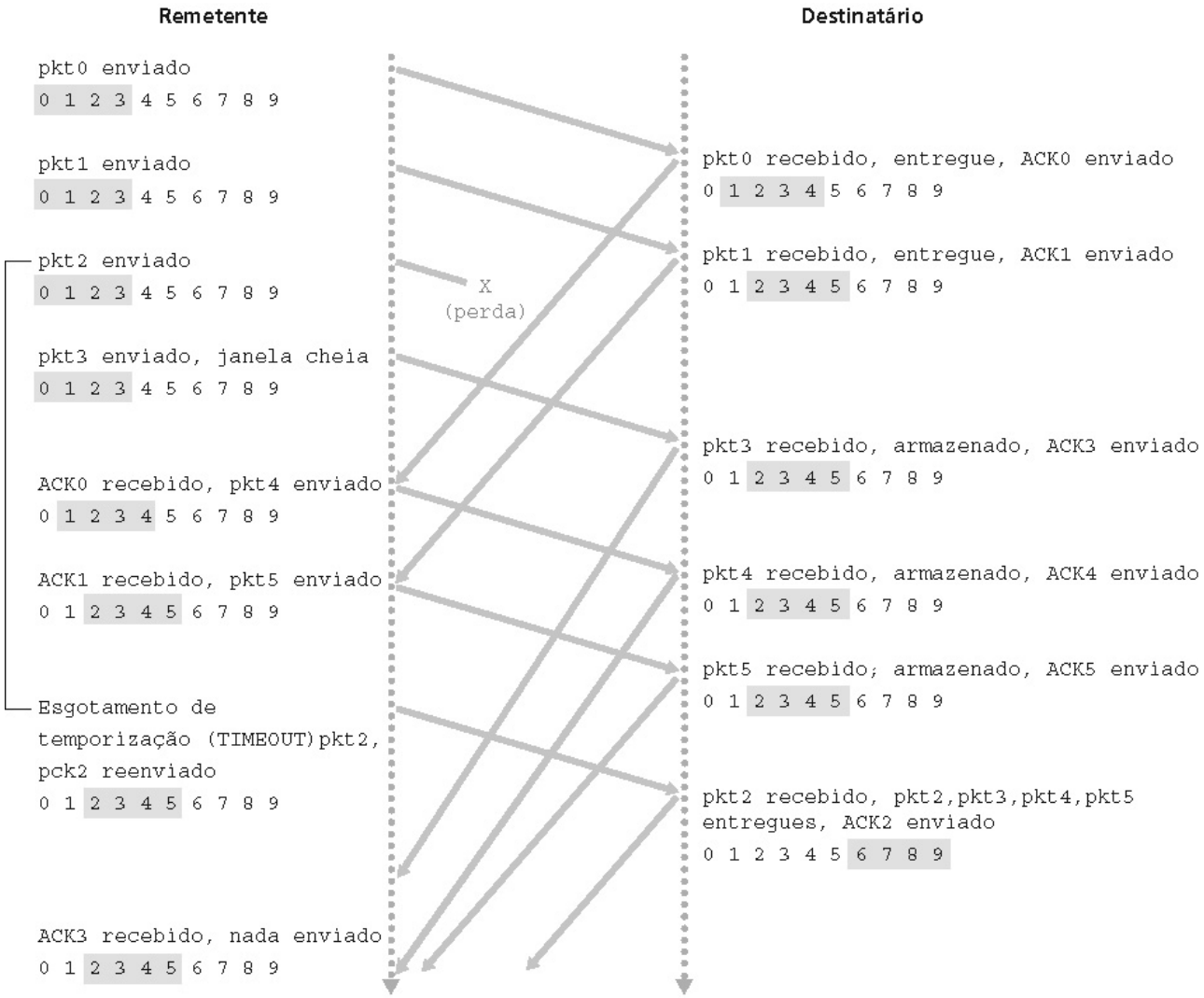
pacote n em [rcvbase-N,rcvbase-1]

- ACK(n)

senão:

- ignora

Retransmissão seletiva em ação



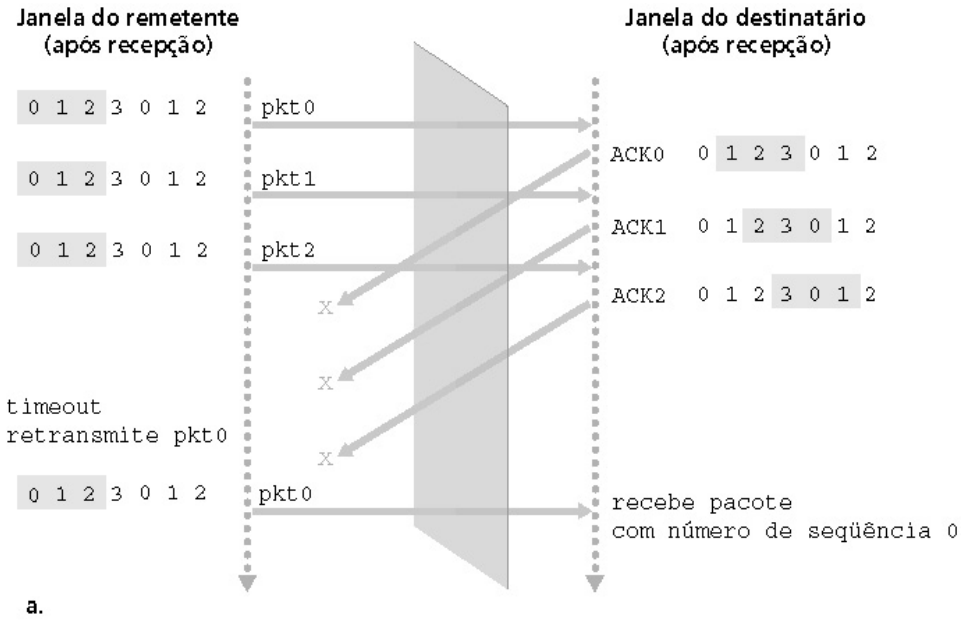
Retransmissão seletiva: dilema

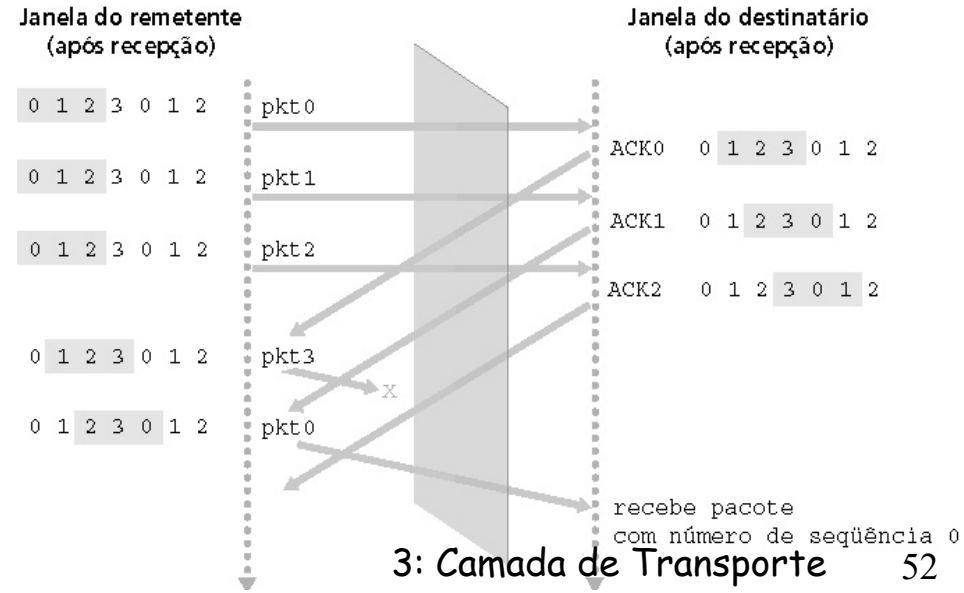
Exemplo:

- nos. de seq : 0, 1, 2, 3
- tam. de janela =3

- receptor não vê diferença entre os dois cenários!
- incorretamente passa dados duplicados como novos em (a)

P: qual a relação entre tamanho de no. de seq e tamanho de janela?



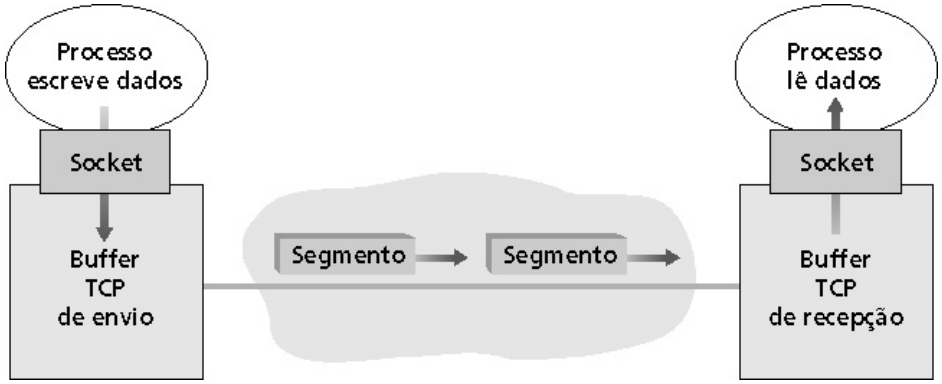


Conteúdo do Capítulo 3

- 3.1 Introdução e serviços de camada de transporte
- 3.2 Multiplexação e demultiplexação
- 3.3 Transporte não orientado para conexão: UDP
- 3.4 Princípios da transferência confiável de dados
- 3.5 Transporte orientado para conexão: TCP
  - estrutura do segmento
  - transferência confiável de dados
  - controle de fluxo
  - gerenciamento da conexão
- 3.6 Princípios de controle de congestionamento
- 3.7 Controle de congestionamento no TCP

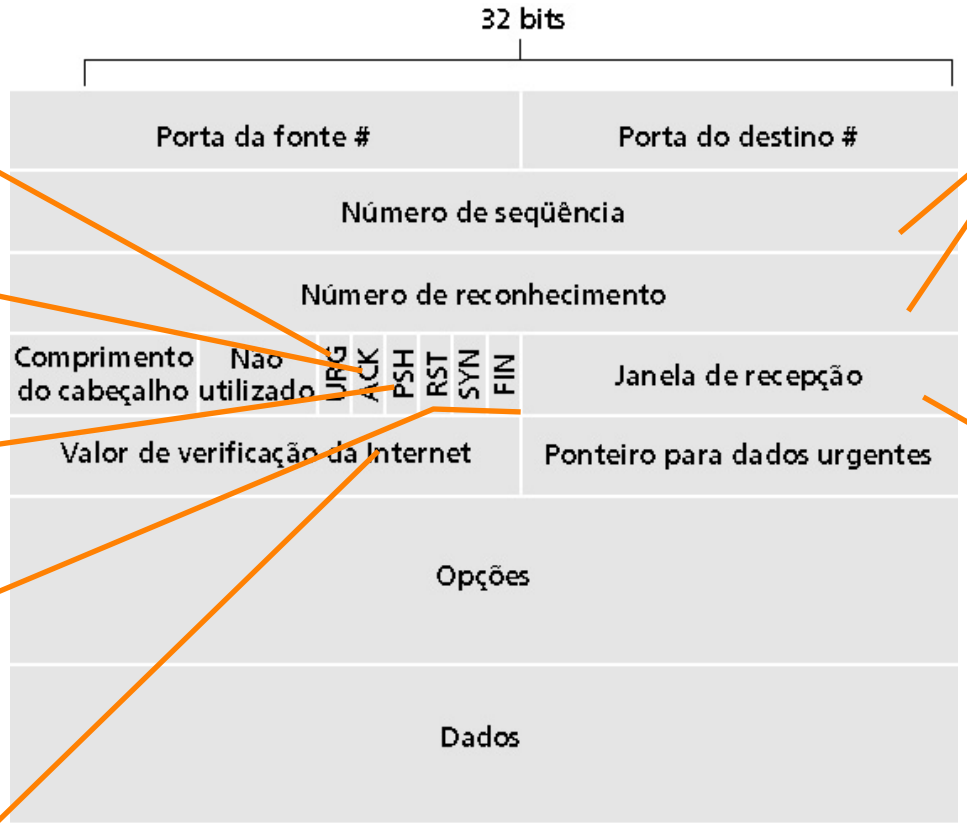
TCP: Visão geral RFCs: 793, 1122, 1323, 2018, 2581

- ponto a ponto:
  - um transmissor, um receptor
- fluxo de bytes, ordenados, confiável:
  - não estruturado em msgs
- com paralelismo *(pipelined)*:
  - tam. da janela ajustado por controle de fluxo e congestionamento do TCP



- transmissão full duplex:
  - fluxo de dados bi-direcional na mesma conexão
  - MSS: tamanho máximo de segmento
- orientado a conexão:
  - *handshaking* (troca de msgs de controle) inicia estado do transmissor e do receptor antes da troca de dados
- fluxo controlado:
  - receptor não será afogado pelo transmissor

Estrutura do segmento TCP



URG: dados urgentes (pouco usado)

ACK: campo de ACK é válido

PSH: produz envio de dados (pouco usado)

RST, SYN, FIN: estabelec. de conexão (comandos de criação e término)

Internet checksum (como no UDP)

contagem por bytes de dados (não segmentos!)

número de bytes receptor está pronto para aceitar

TCP: nos. de seq. e ACKs

Nos. de seq.:

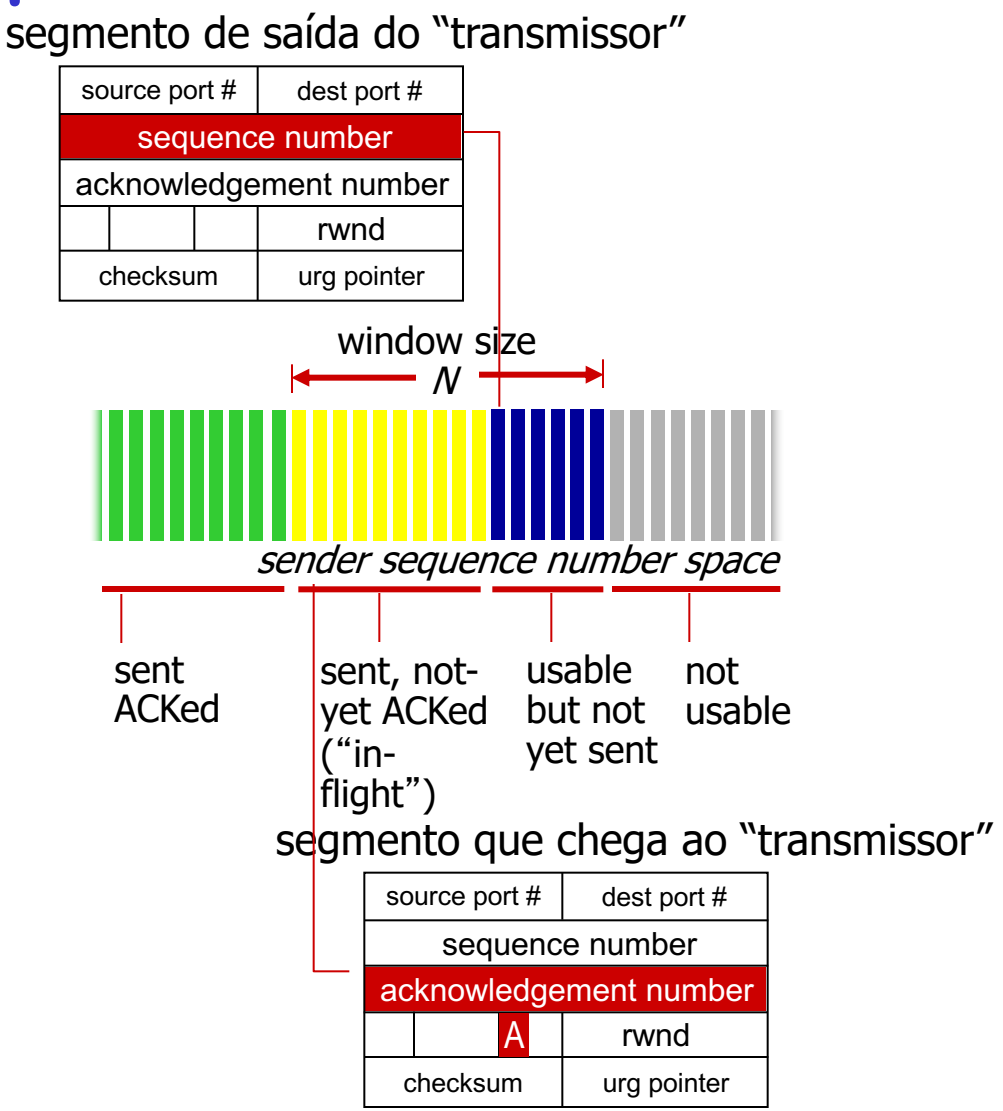
- "número"dentro do fluxo de bytes do primeiro byte de dados do segmento

ACKs:

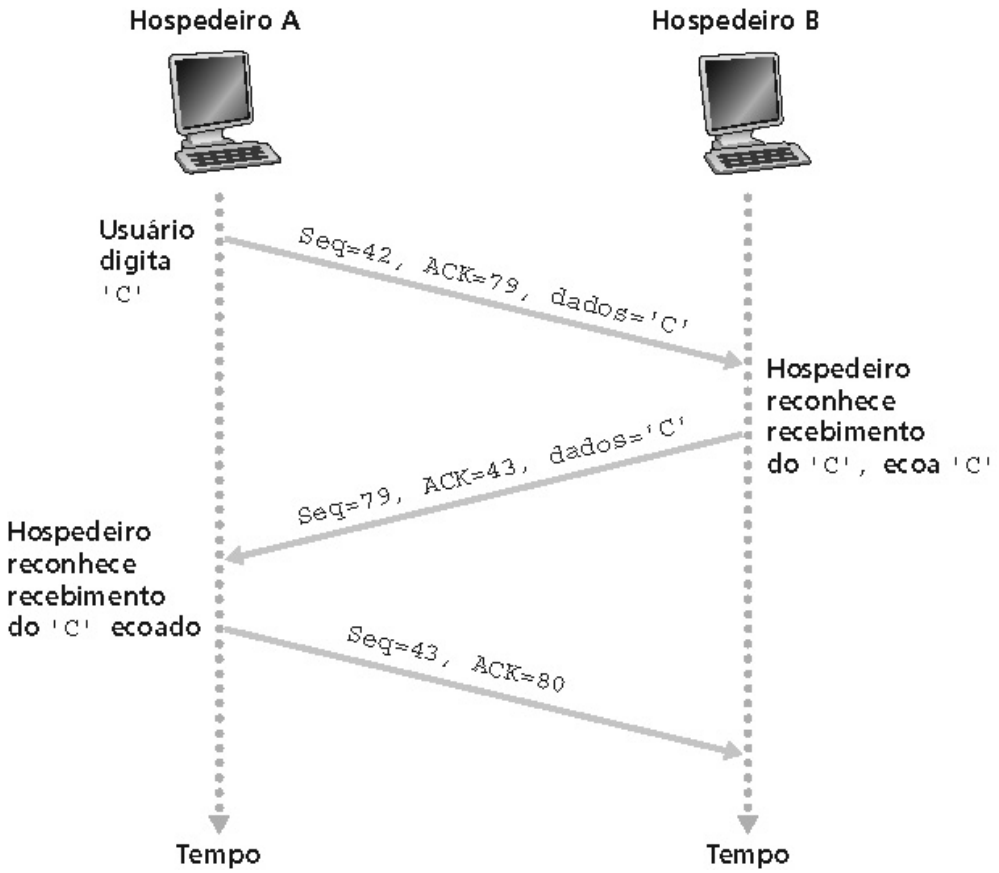
- no. de seq do próx. byte esperado do outro lado
- ACK cumulativo

P: como receptor trata segmentos fora da ordem?

- R: espec do TCP omissa - deixado ao implementador



TCP: nos. de seq. e ACKs



cenário telnet simples

TCP: tempo de viagem de ida e volta (RTT – *Round Trip Time*) e Temporização

P: como escolher o valor do temporizador TCP?

- maior que o RTT
  - mas o RTT varia
- *muito curto:* temporização prematura
  - retransmissões desnecessárias
- *muito longo:* reação demorada à perda de segmentos

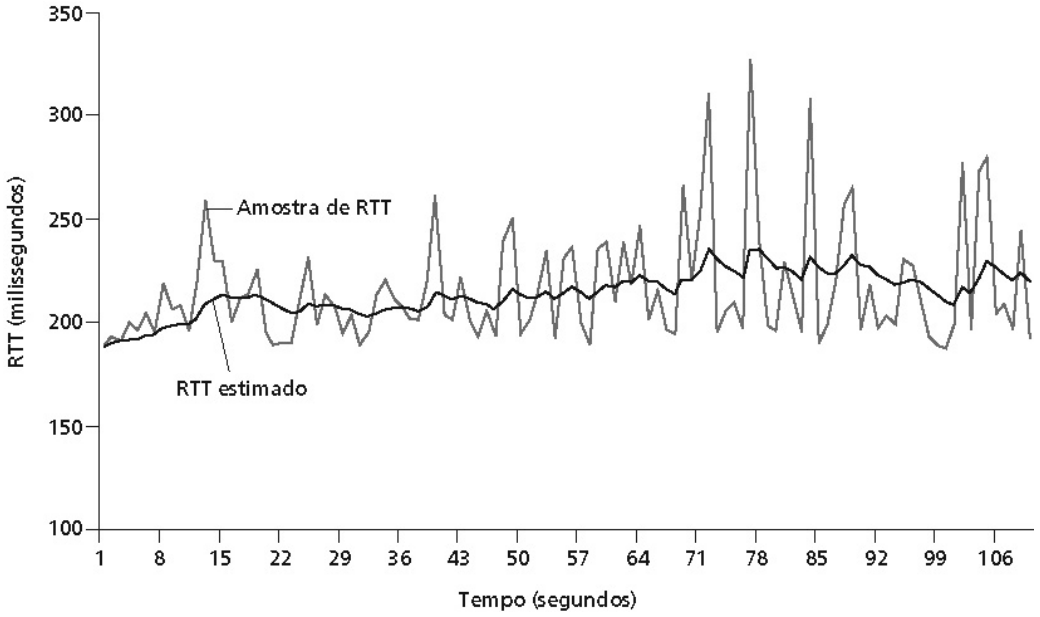
P: como estimar RTT?

- **SampleRTT:** tempo medido entre a transmissão do segmento e o recebimento do ACK correspondente
  - ignora retransmissões
- **SampleRTT** varia de forma rápida, é desejável um "amortecedor" para a estimativa do RTT
  - usa várias medições recentes, não apenas o último **SampleRTT** obtido

TCP: Tempo de Resposta (RTT) e Temporização

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

- média móvel exponencialmente ponderada
- influência de cada amostra diminui exponencialmente com o tempo
- valor típico de α = 0,125



TCP: Tempo de Resposta (RTT) e Temporização

Escolhendo o intervalo de temporização

- **EstimatedRTT** mais uma "margem de segurança"
  - grandes variações no **EstimatedRTT**
    -> maior margem de segurança
- primeiro estimar o quanto a SampleRTT se desvia do EstimatedRTT:

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

(valor típico de $\beta = 0{,}25$)

- Então, ajusta o temporizador para:

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

RTT estimado

"margem de segurança"

Conteúdo do Capítulo 3

- 3.1 Introdução e serviços de camada de transporte
- 3.2 Multiplexação e demultiplexação
- 3.3 Transporte não orientado para conexão: UDP
- 3.4 Princípios da transferência confiável de dados
- 3.5 Transporte orientado para conexão: TCP
  - estrutura do segmento
  - transferência confiável de dados
  - controle de fluxo
  - gerenciamento da conexão
- 3.6 Princípios de controle de congestionamento
- 3.7 Controle de congestionamento no TCP

Transferência de dados confiável do TCP

- O TCP cria um serviço rdt sobre o serviço não confiável do IP
  - Segmentos transmitidos em "paralelo" (*pipelined)*
  - Acks cumulativos
  - O TCP usa um único temporizador para retransmissões
- As retransmissões são disparadas por:
  - estouros de temporização
  - acks duplicados
- Considere inicialmente um transmissor TCP simplificado:
  - ignore acks duplicados
  - ignore controles de fluxo e de congestionamento

Eventos do transmissor TCP

Dados recebidos da aplicação:

- Cria segmento com no. de sequência (nseq)
- nseq é o número de sequência do primeiro byte de dados do segmento
- Liga o temporizador se já não estiver ligado (temporização do segmento mais antigo ainda não reconhecido)
- Valor do temporizador: calculado anteriormente

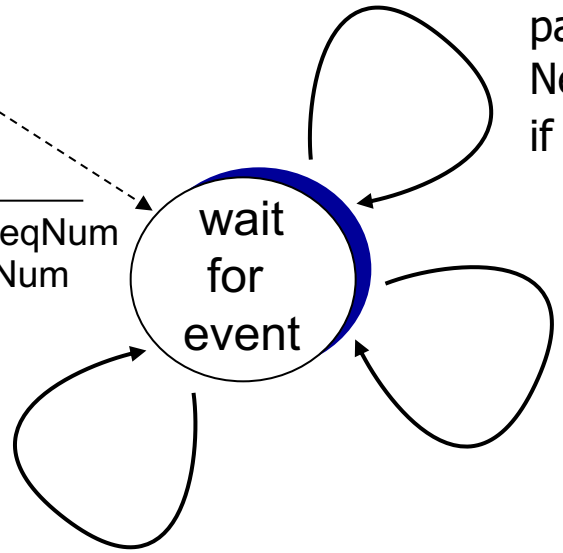
Estouro do temporizador:

- Retransmite o segmento que causou o estouro do temporizador
- Reinicia o temporizador

Recepção de Ack:

- Se reconhecer segmentos ainda não reconhecidos
  - atualizar informação sobre o que foi reconhecido
  - religa o temporizador se ainda houver segmentos pendentes (não reconhecidos)

Transmissor TCP (simplificado)



Λ

NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

data received from application above

create segment, seq. #: NextSeqNum
pass segment to IP (i.e., "send")
NextSeqNum = NextSeqNum + length(data)
if (timer currently not running)
    start timer

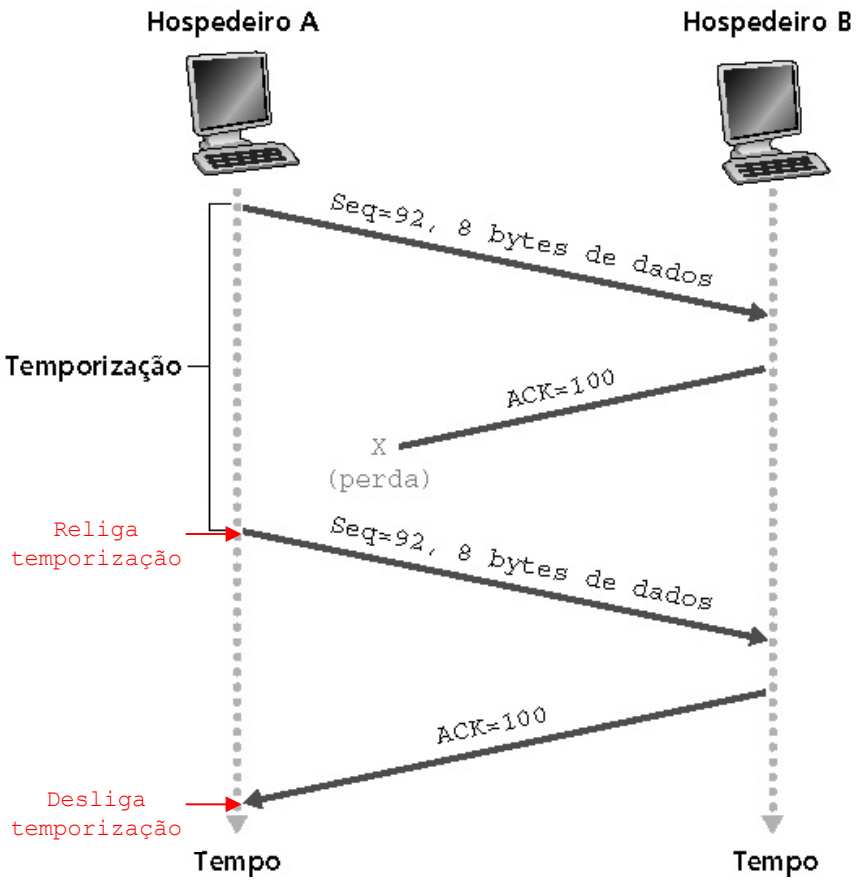
timeout

retransmit not-yet-acked segment
        with smallest seq. #
start timer

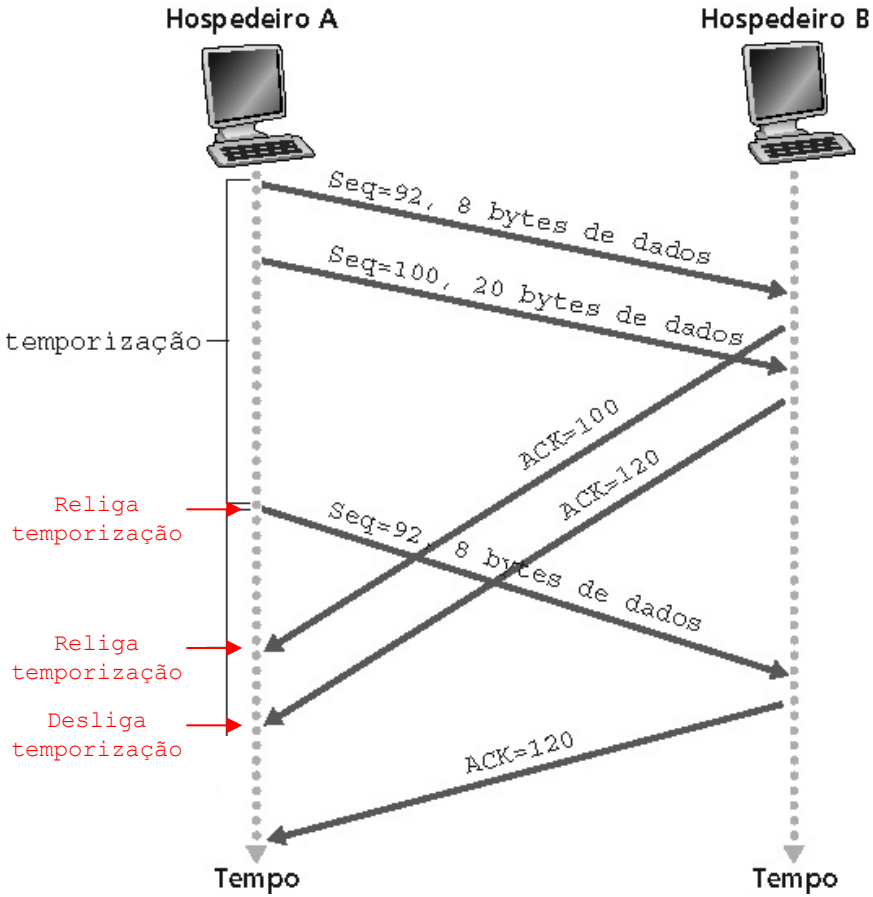
ACK received, with ACK field value y

```
if (y > SendBase) {
   SendBase = y
   /* SendBase–1: last cumulatively ACKed byte */
   if (there are currently not-yet-acked segments)
       start timer
     else stop timer
    }
```

TCP: cenários de retransmissão

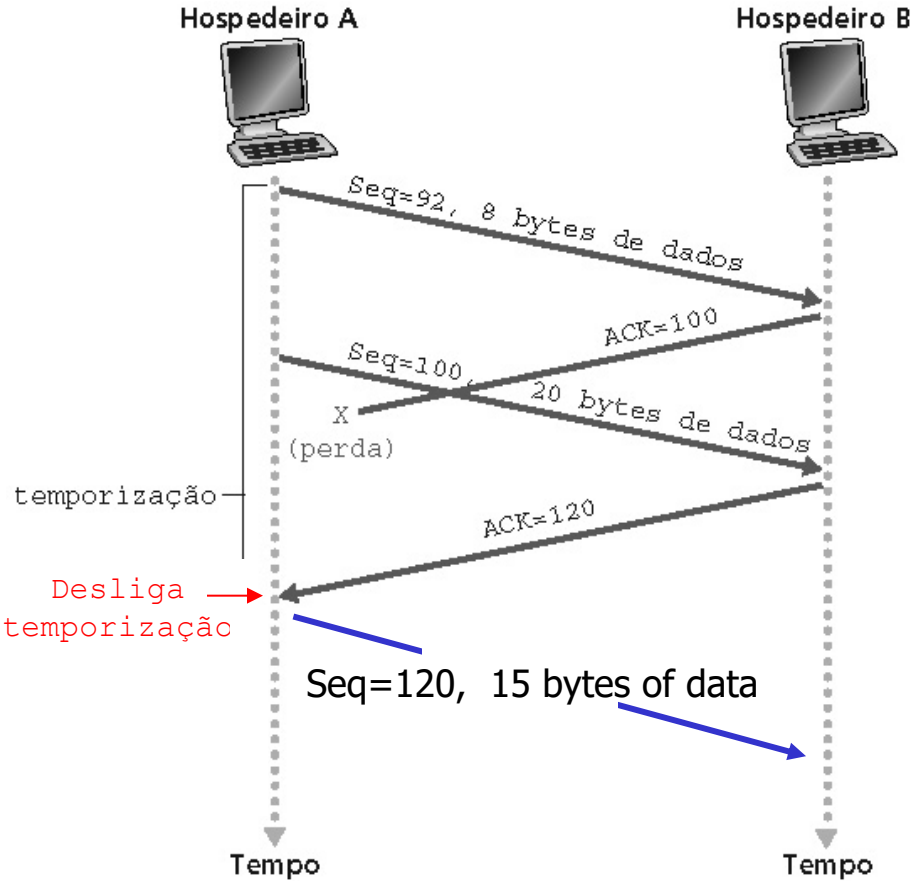


Cenário com perda do ACK



Temporização prematura, ACKs cumulativos

TCP: cenários de retransmissão (mais)



Cenário de ACK cumulativo

TCP geração de ACKs [RFCs 1122, 2581]

Evento no Receptor	Ação do Receptor TCP
chegada de segmento em ordem sem lacunas, anteriores já reconhecidos	ACK retardado. Espera até 500ms pelo próx. segmento. Se não chegar segmento, envia ACK
chegada de segmento em ordem sem lacunas, um ACK retardado pendente	envia imediatamente um único ACK cumulativo
chegada de segmento fora de ordem, com no. de seq. maior que esperado -> lacuna	envia ACK duplicado, indicando no. de seq.do próximo byte esperado
chegada de segmento que preenche a lacuna parcial ou completamente	ACK imediato se segmento começa no início da lacuna

Retransmissão rápida do TCP

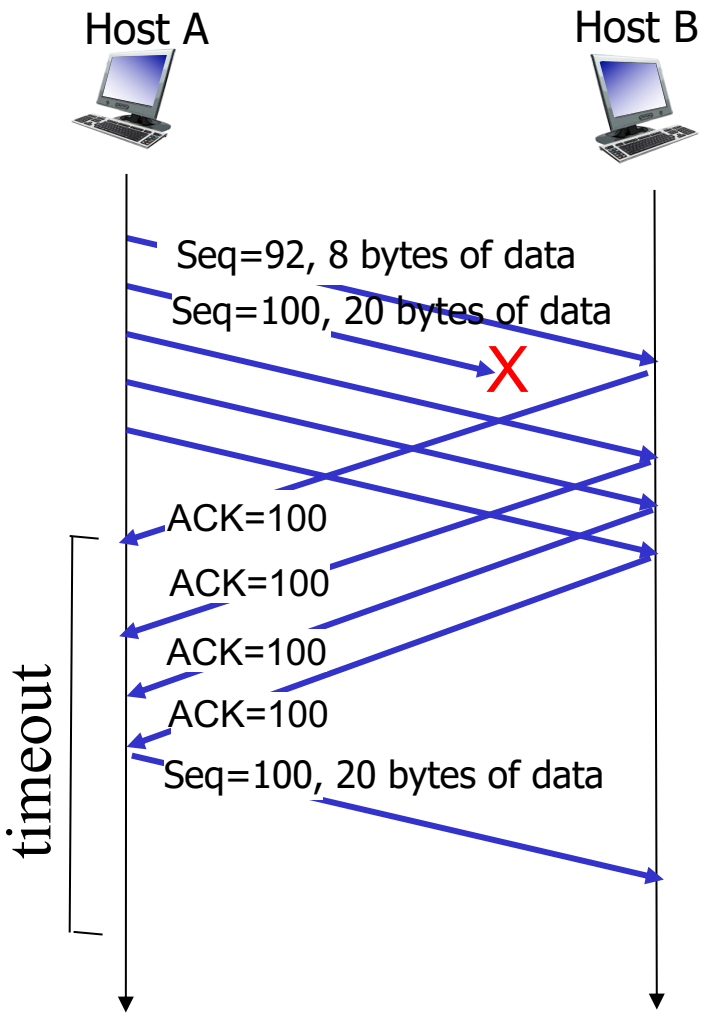
- O intervalo do temporizador é frequentemente bastante longo:
  - longo atraso antes de retransmitir um pacote perdido
- Detecta segmentos perdidos através de ACKs duplicados.
  - O transmissor normalmente envia diversos segmentos
  - Se um segmento se perder, provavelmente haverá muitos ACKs duplicados.

retx rápida do TCP

se o transmissor receber 3 ACKs para os mesmos dados

("três ACKs duplicados"), retransmite segmentos não reconhecidos com menores nos. de seq.

- provavelmente o segmento não reconhecido se perdeu, não é preciso esperar o temporizador.



Retransmissão de um segmento após três ACKs duplicados

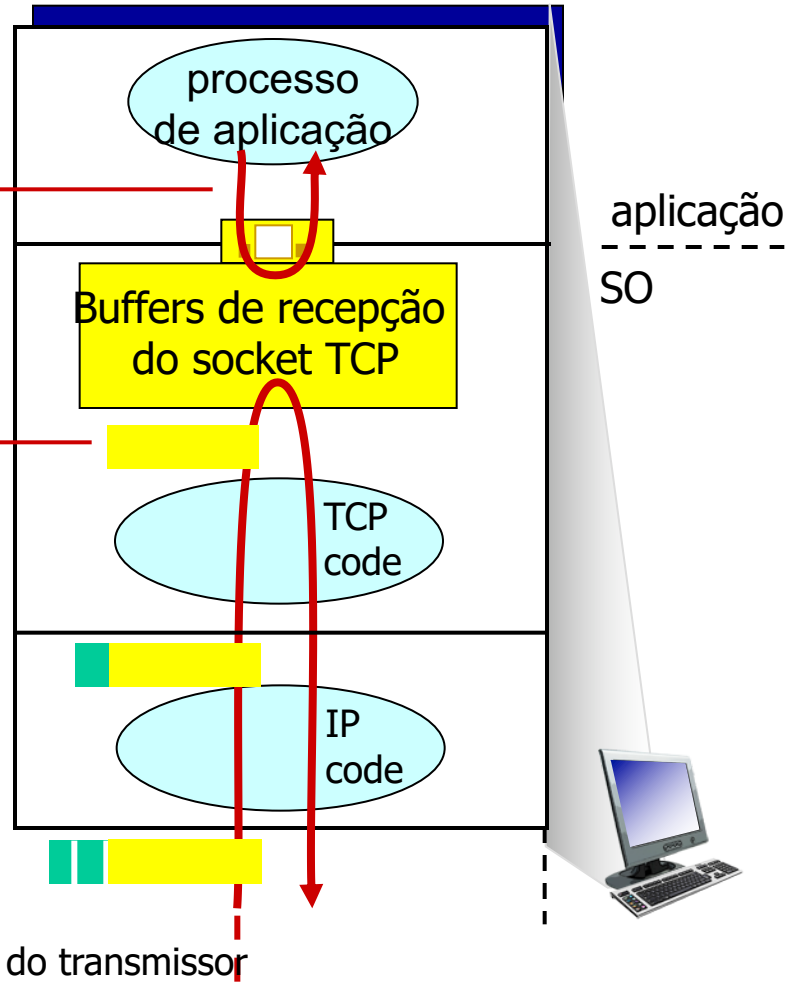
Conteúdo do Capítulo 3

- 3.1 Introdução e serviços de camada de transporte
- 3.2 Multiplexação e demultiplexação
- 3.3 Transporte não orientado para conexão: UDP
- 3.4 Princípios da transferência confiável de dados
- 3.5 Transporte orientado para conexão: TCP
  - estrutura do segmento
  - transferência confiável de dados
  - controle de fluxo
  - gerenciamento da conexão
- 3.6 Princípios de controle de congestionamento
- 3.7 Controle de congestionamento no TCP

Controle de Fluxo do TCP

a aplicação pode remover dados dos buffers do socket TCP ....

... mais devagar do que o receptor TCP está entregando (transmissor está enviando)



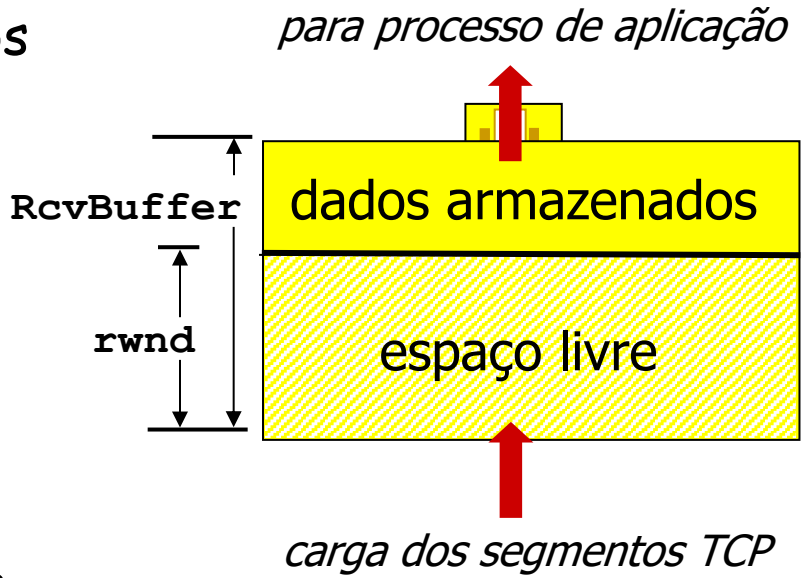
pilha de protocolos no receptor

Controle de fluxo

o receptor controla o transmissor, de modo que este não inunde o buffer do receptor transmitindo muito e rapidamente

Controle de Fluxo do TCP: como funciona

- O receptor "anuncia" o espaço livre do buffer incluindo o valor da **`rwnd`** nos cabeçalhos TCP dos segmentos que saem do receptor para o transmissor
  - Tamanho do **`RcvBuffer`** é configurado através das opções do socket (o valor default é de 4096 bytes)
  - muitos sistemas operacionais ajustam **`RcvBuffer`** automaticamente.
- O transmissor limita a quantidade os dados não reconhecidos ao tamanho do **`rwnd`** recebido.
- Garante que o buffer do receptor não transbordará



armazenamento no lado do receptor

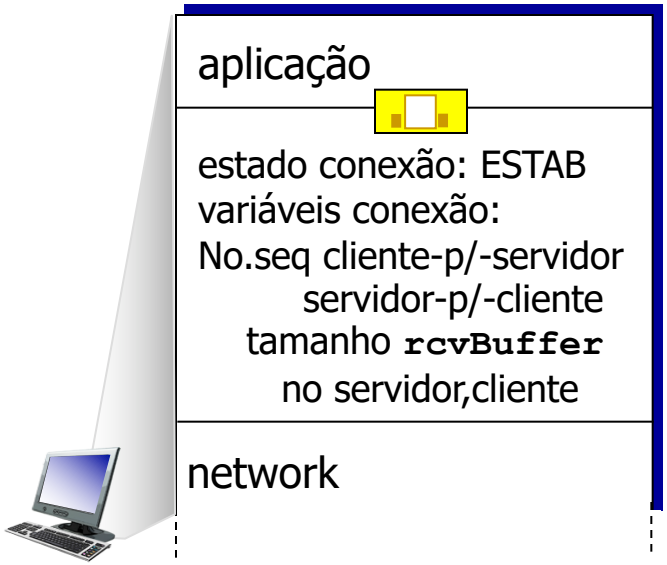
Conteúdo do Capítulo 3

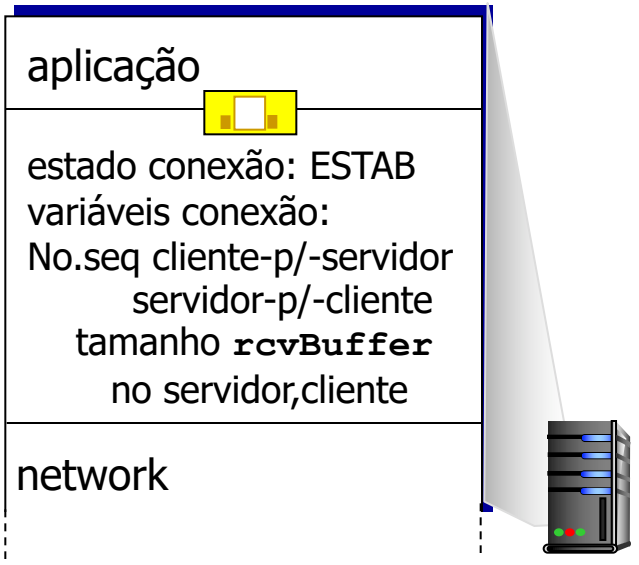
- 3.1 Introdução e serviços de camada de transporte
- 3.2 Multiplexação e demultiplexação
- 3.3 Transporte não orientado para conexão: UDP
- 3.4 Princípios da transferência confiável de dados
- 3.5 Transporte orientado para conexão: TCP
  - estrutura do segmento
  - transferência confiável de dados
  - controle de fluxo
  - gerenciamento da conexão
- 3.6 Princípios de controle de congestionamento
- 3.7 Controle de congestionamento no TCP

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.



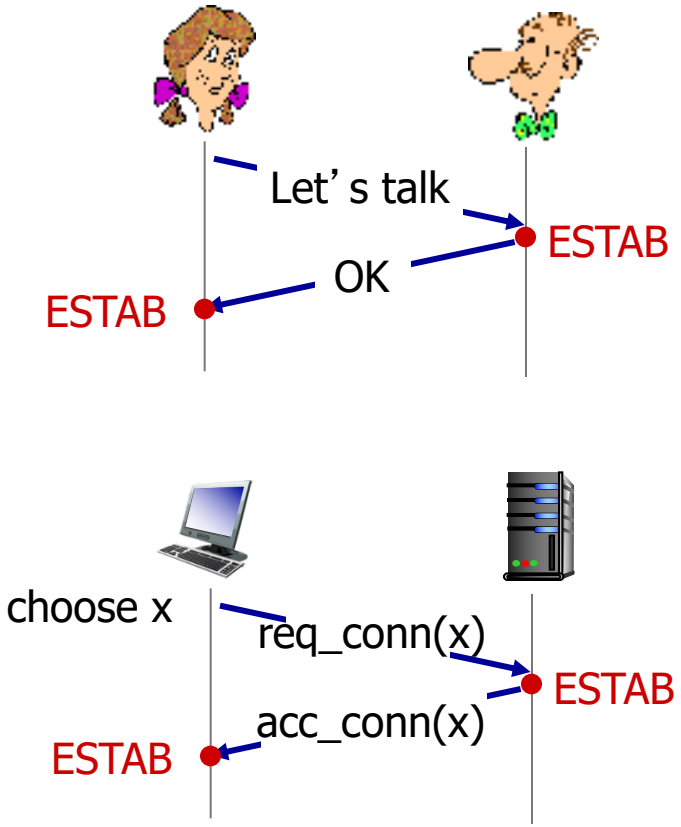



```
Socket clientSocket =
  newSocket("hostname","port
   number");
```

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

Concordando em estabelecer uma conexão

Apresentação de duas vias (*2-way handshake*):

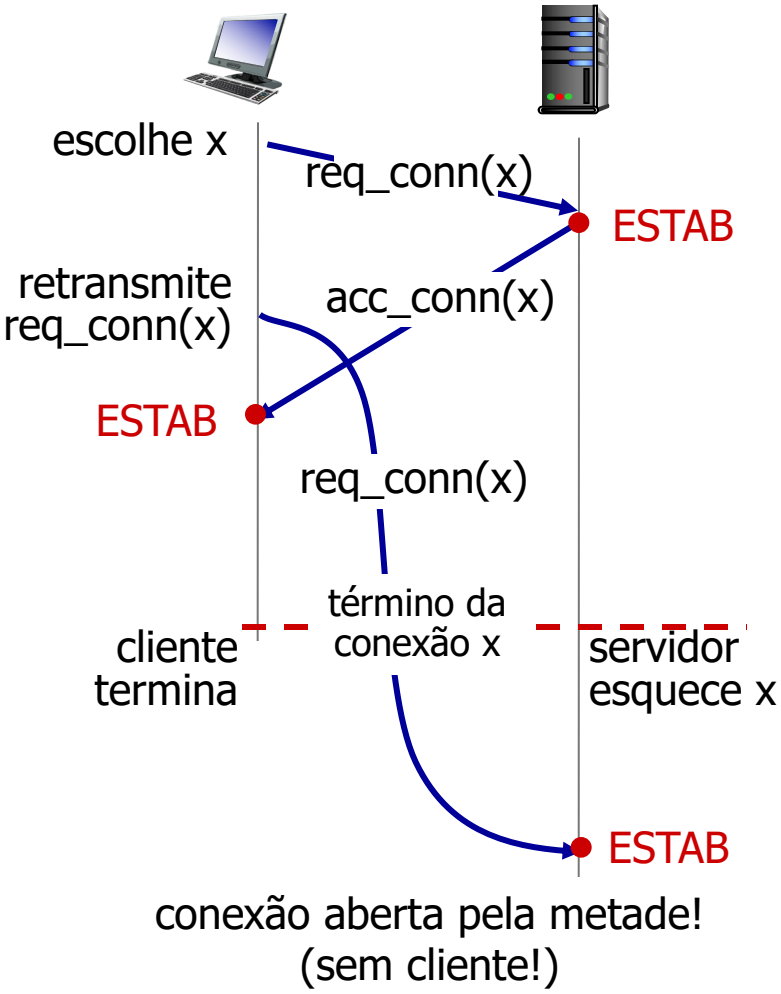


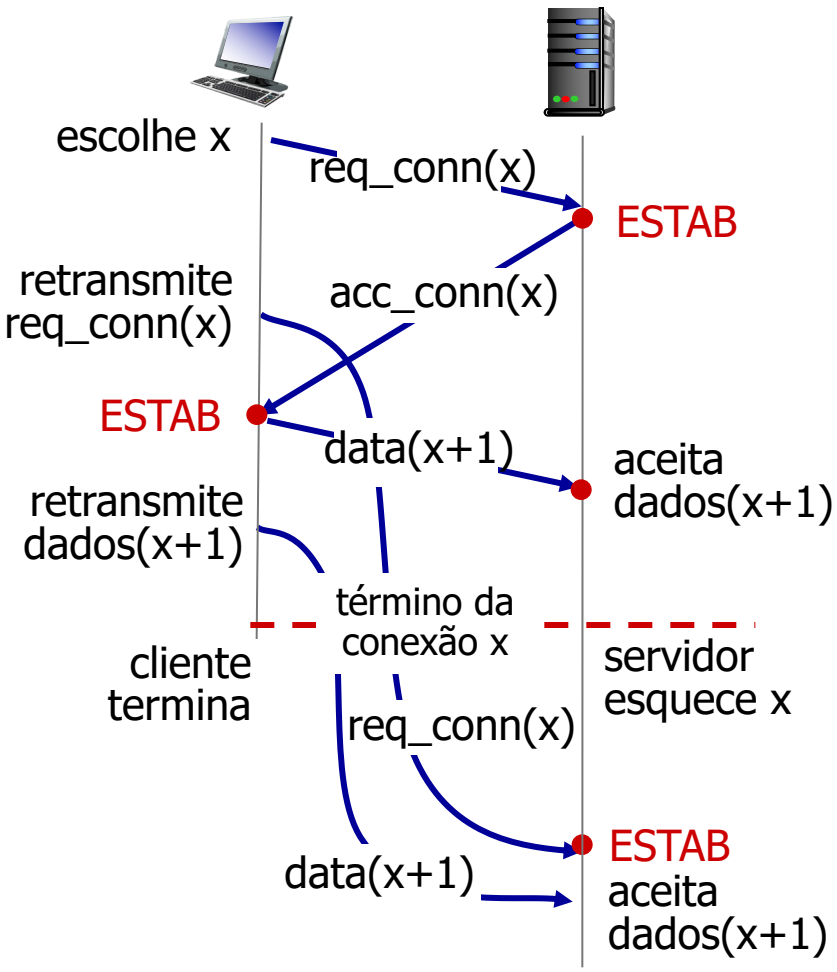
P: a apresentação em duas vias sempre funciona em redes?

- atrasos variáveis
- mensagens retransmitidas (ex: req_conn(x)) devido à perda de mensagem
- reordenação de mensagens
- não consegue ver o outro lado

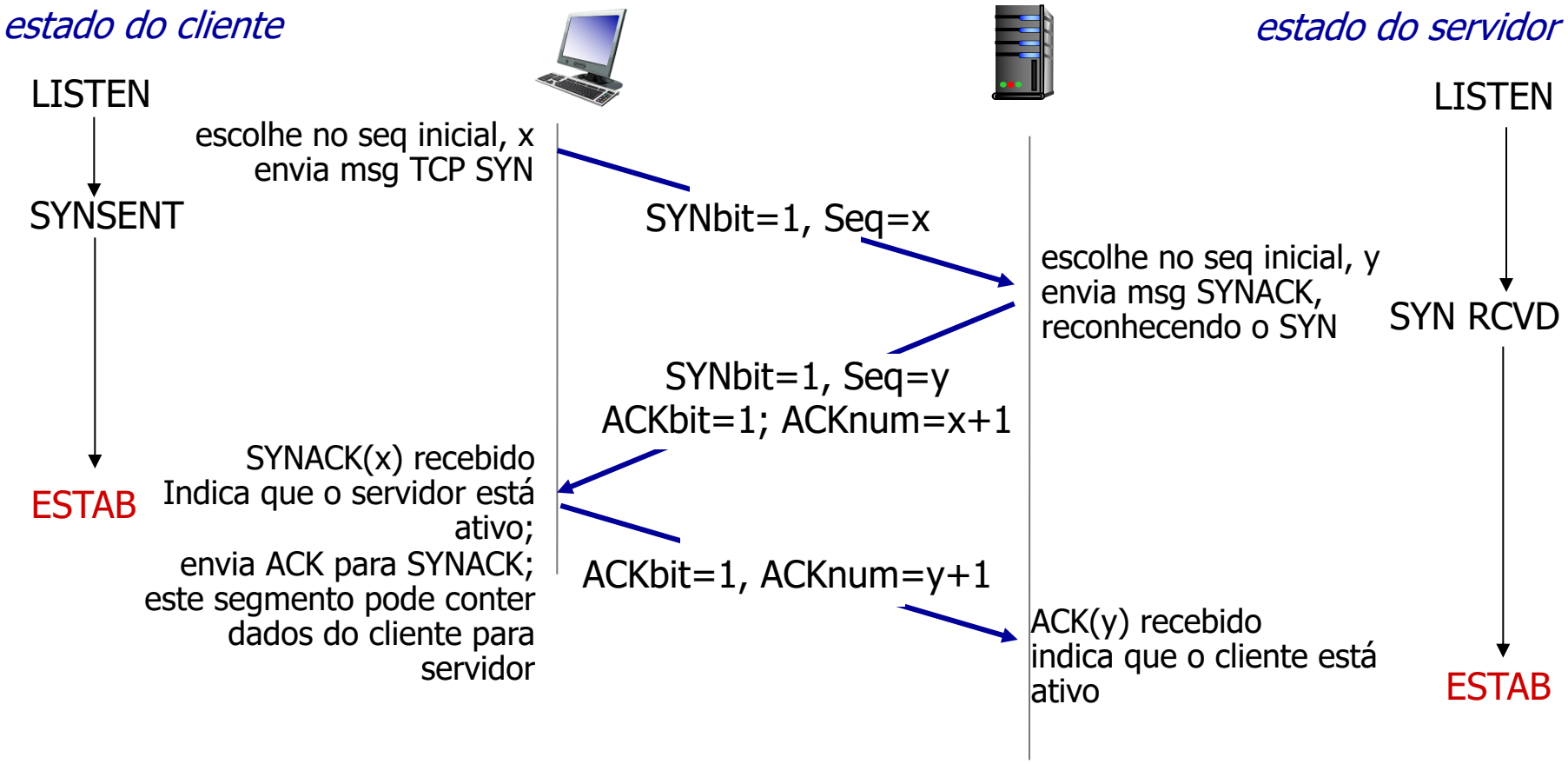
Concordando em estabelecer uma conexão

cenários de falha da apresentação de duas vias:

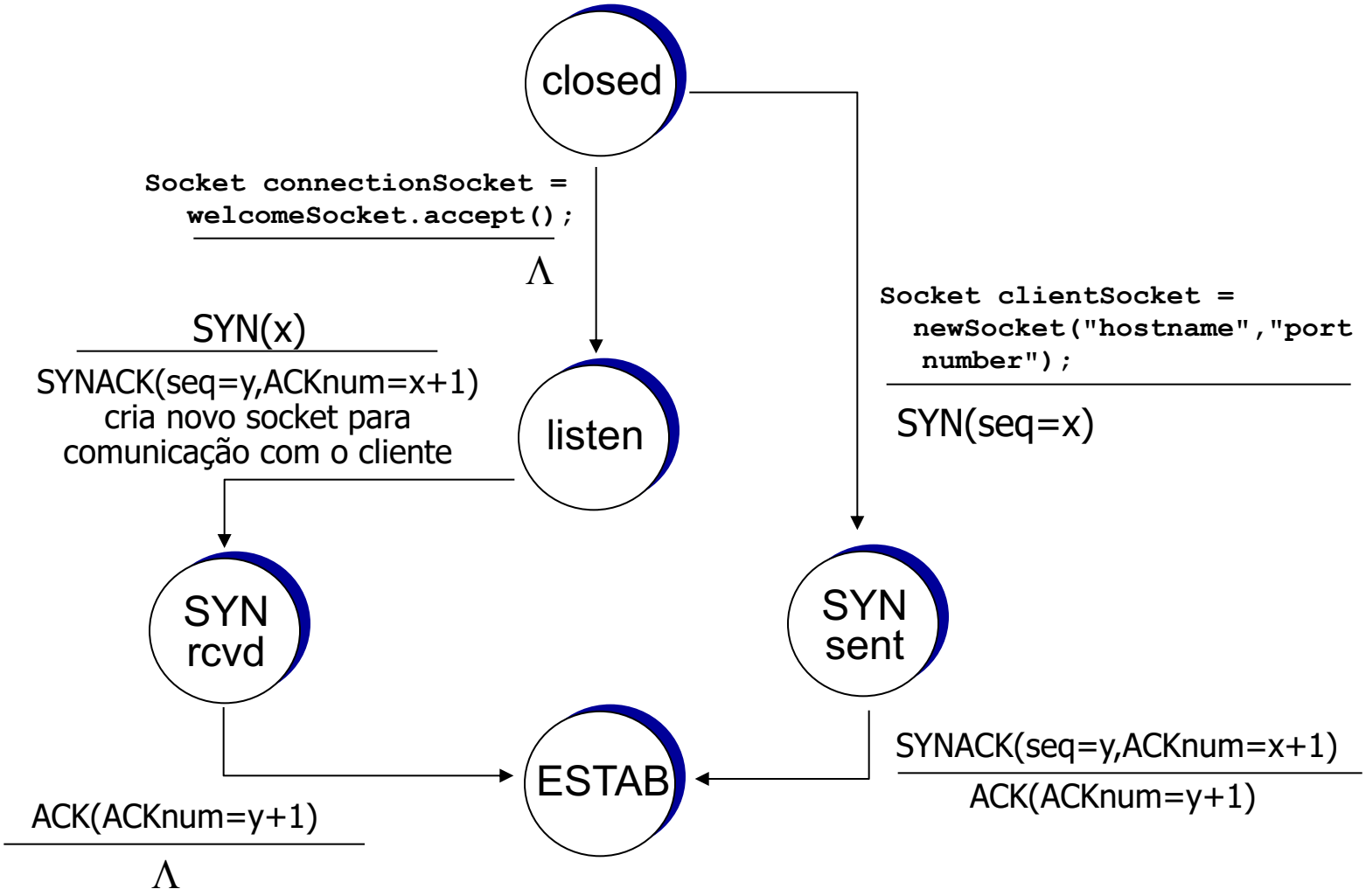




Apresentação de três vias do TCP



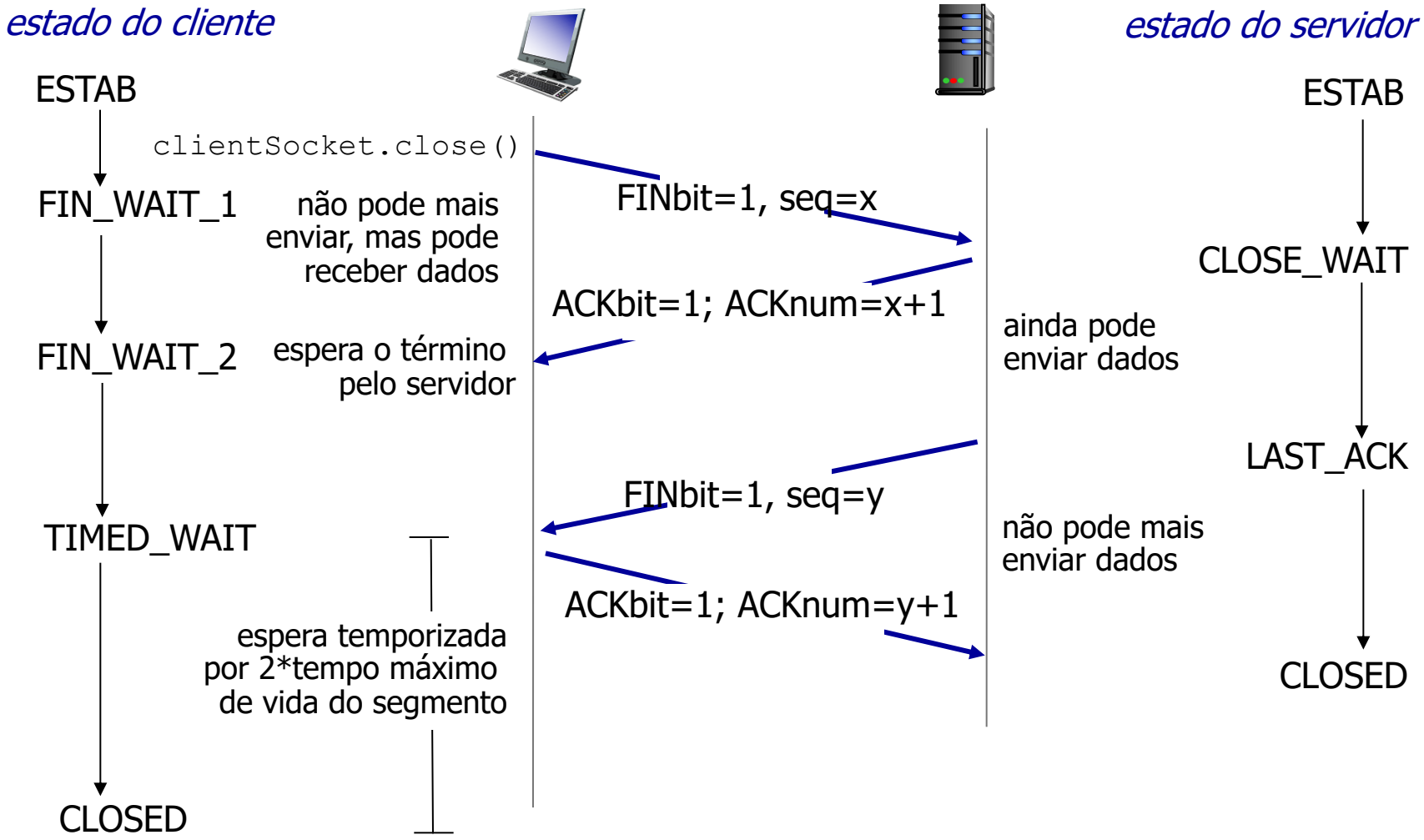
Apresentação de três vias do TCP



TCP: Encerrando uma conexão

- seja o cliente que o servidor fecham cada um o seu lado da conexão
  - enviam segmento TCP com bit FIN = 1
- respondem ao FIN recebido com um *ACK*
  - ao receber um FIN, *ACK* pode ser combinado com o próprio FIN
- lida com trocas de FIN simultâneos

TCP: Encerrando uma conexão



Conteúdo do Capítulo 3

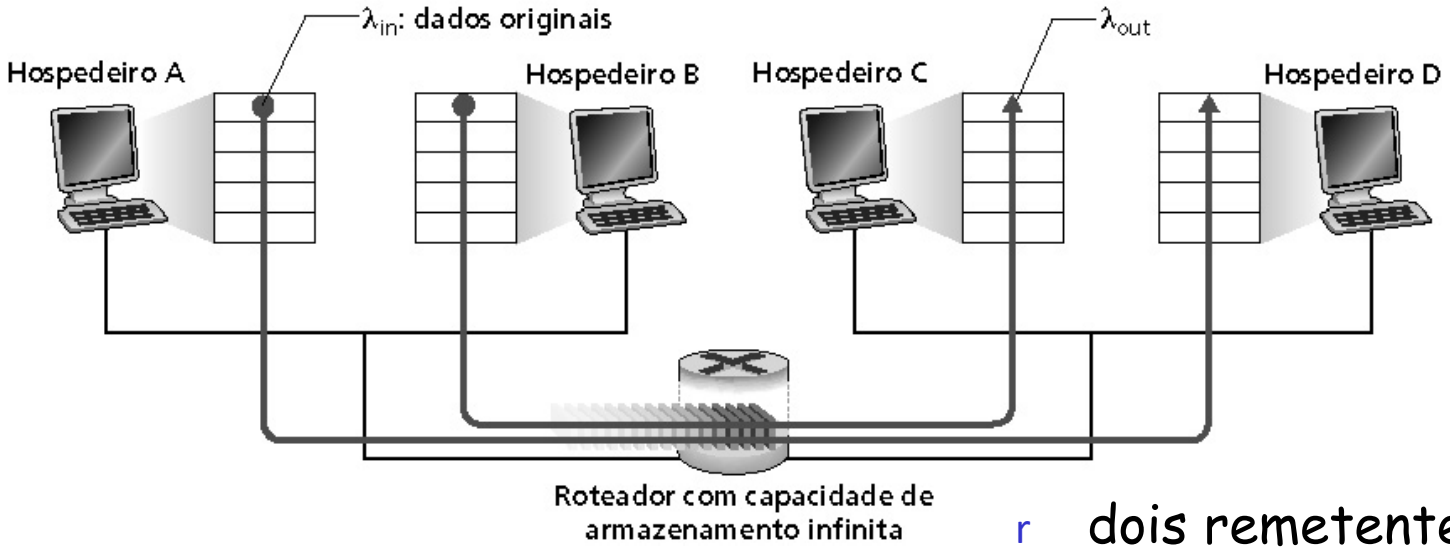
- 3.1 Introdução e serviços de camada de transporte
- 3.2 Multiplexação e demultiplexação
- 3.3 Transporte não orientado para conexão: UDP
- 3.4 Princípios da transferência confiável de dados
- 3.5 Transporte orientado para conexão: TCP
- 3.6 Princípios de controle de congestionamento
- 3.7 Controle de congestionamento no TCP

Princípios de Controle de Congestionamento

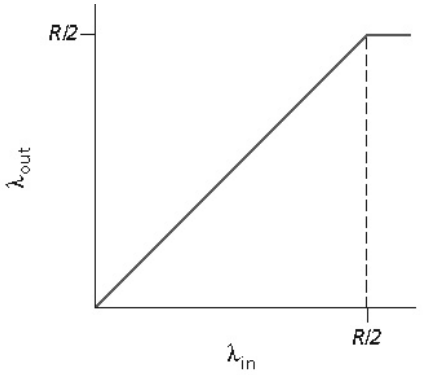
Congestionamento:

- informalmente: "muitas fontes enviando dados acima da capacidade da *rede* de tratá-los"
- diferente de controle de fluxo!
- Sintomas:
  - perda de pacotes (saturação de buffers nos roteadores)
  - longos atrasos (enfileiramento nos buffers dos roteadores)
- um dos 10 problemas mais importantes em redes!

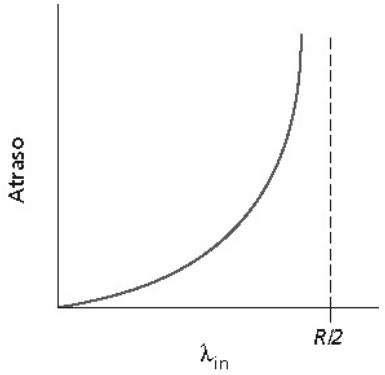
Causas/custos de congestionamento: cenário 1



- dois remetentes, dois receptores
- um roteador, *buffers* infinitos
- sem retransmissão
- capacidade do link de saída: R



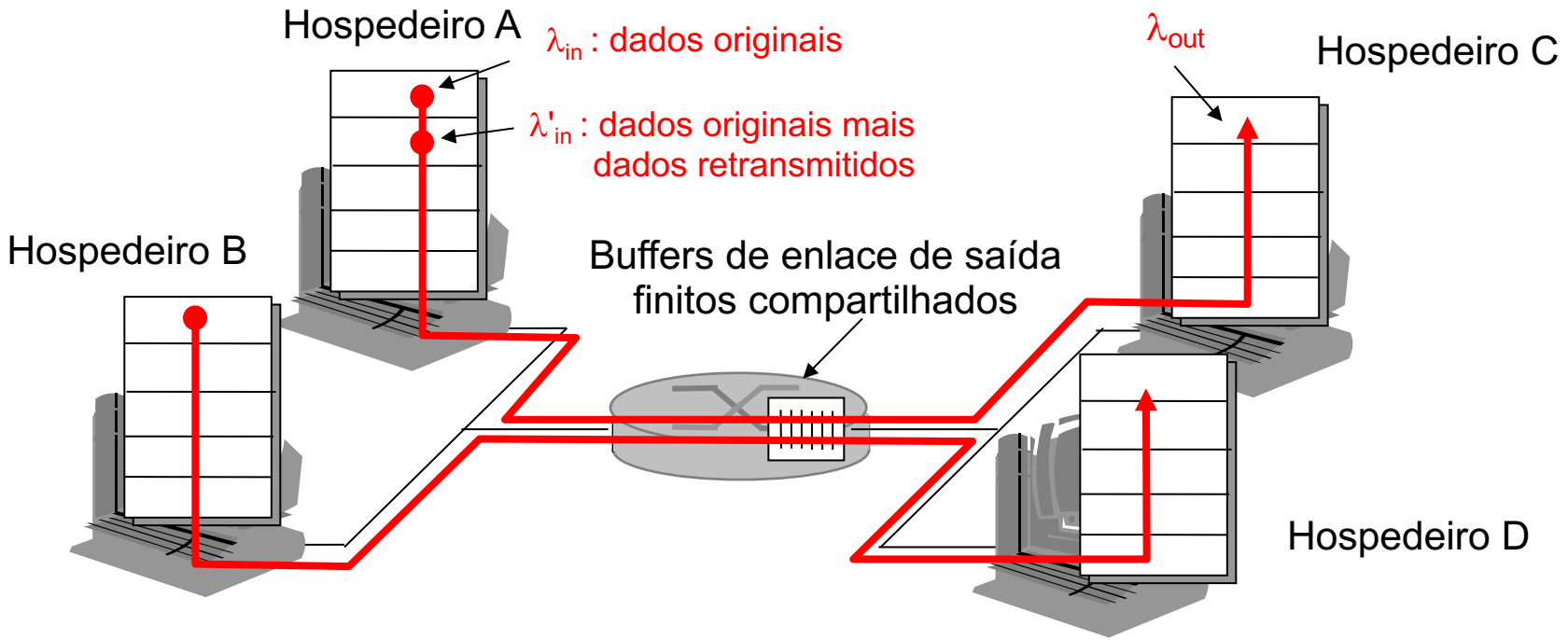
Vazão máxima por conexão: R/2



Grandes atrasos qdo. taxa de chegada se aproxima da capacidade

Causas/custos de congest.: cenário 2

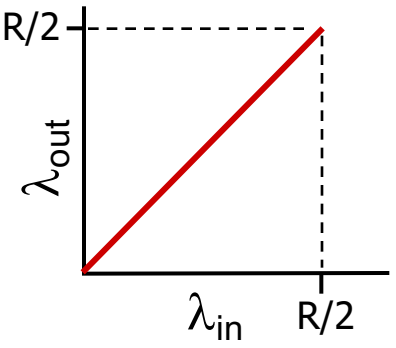
- Um roteador, buffers *finitos*
- retransmissão pelo remetente de pacote perdido
  - entrada camada apl. = saída camada apl.: $\lambda_{in} = \lambda_{out}$
  - entrada camada transp. inclui retransmissões.: $\lambda'_{in} \geq \lambda_{out}$

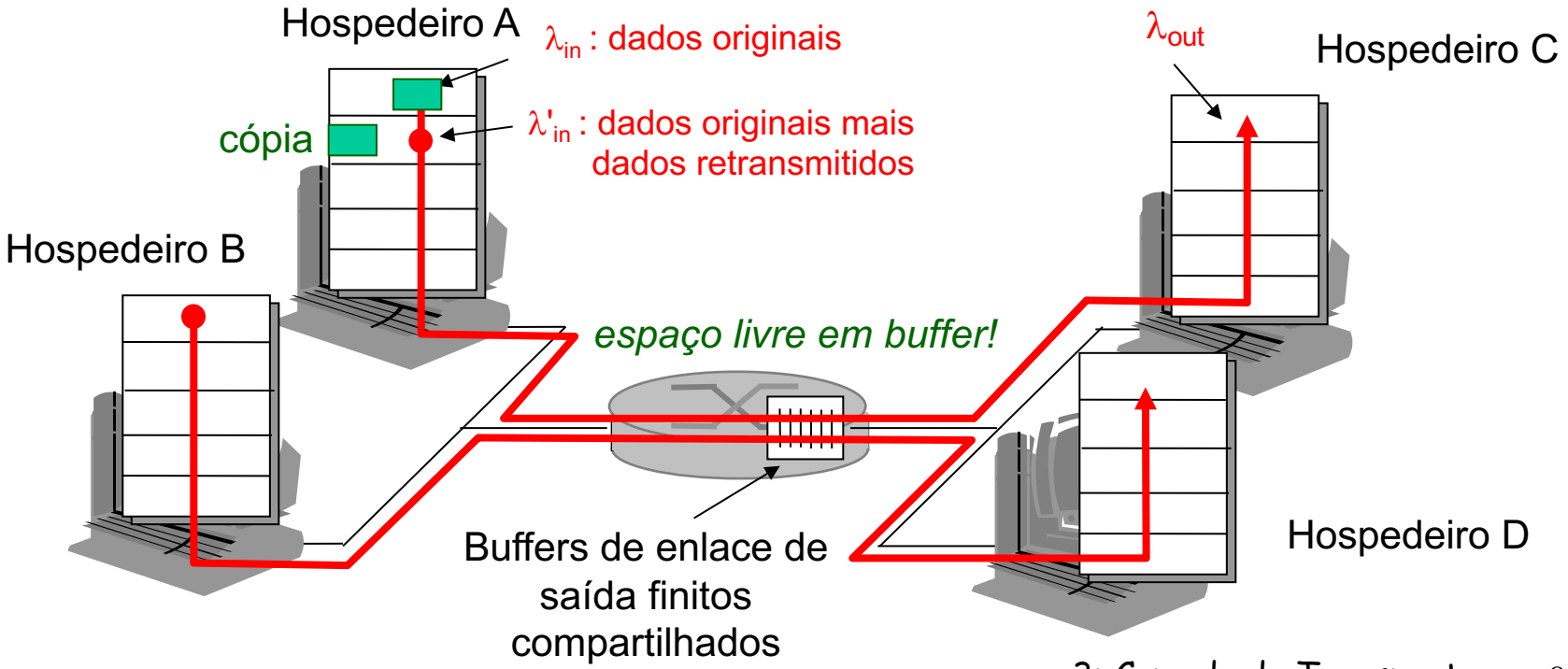


Causas/custos de congest.: cenário 2

Idealização: conhecimento perfeito

- transmissor envia apenas quando houver buffer disponível no roteador

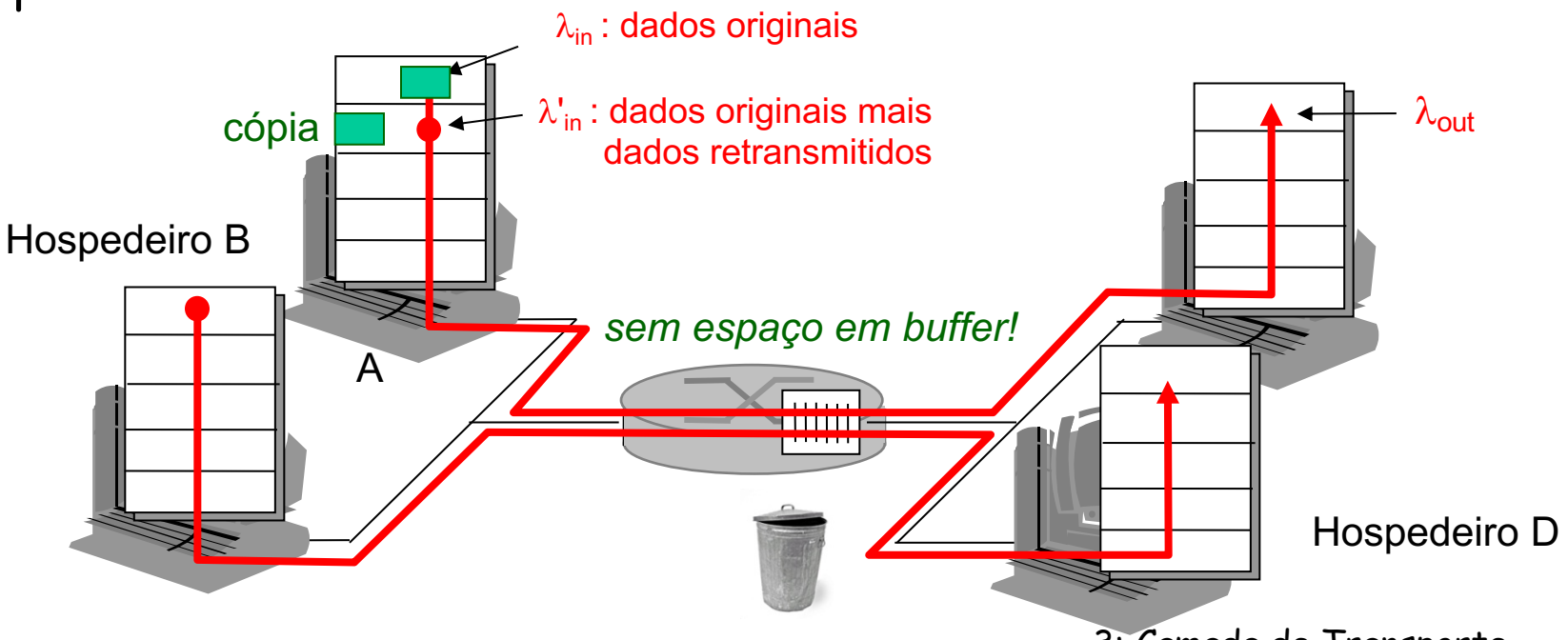




Causas/custos de congest.: cenário 2

Idealização: *perda conhecida.*
pacotes podem ser perdidos, descartados no roteador devido a buffers cheios

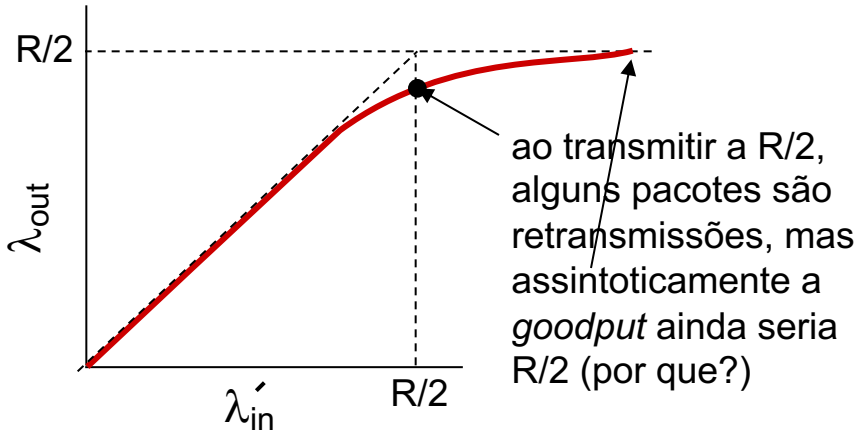
- transmissor apenas retransmite se o pacote *sabidamente* se perdeu.

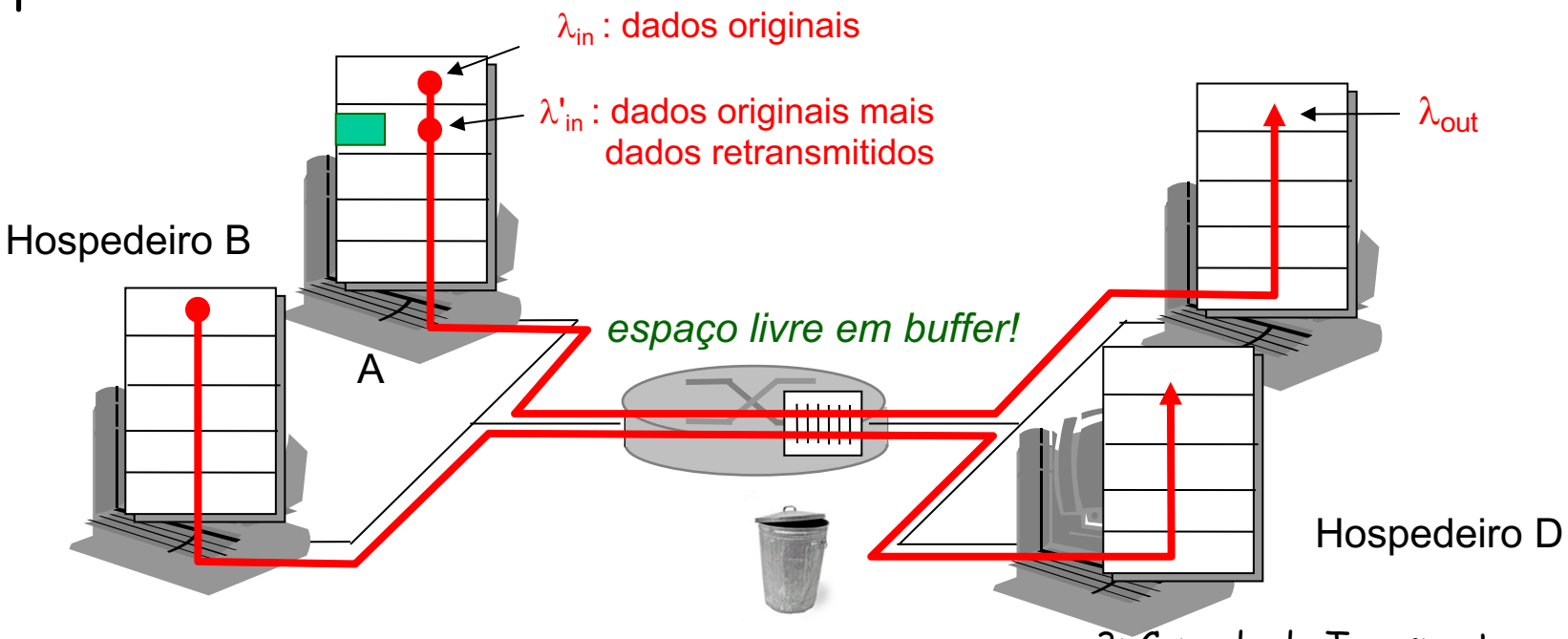


Causas/custos de congest.: cenário 2

Idealização: *perda conhecida.* pacotes podem ser perdidos, descartados no roteador devido a buffers cheios

- transmissor apenas retransmite se o pacote *sabidamente* se perdeu.

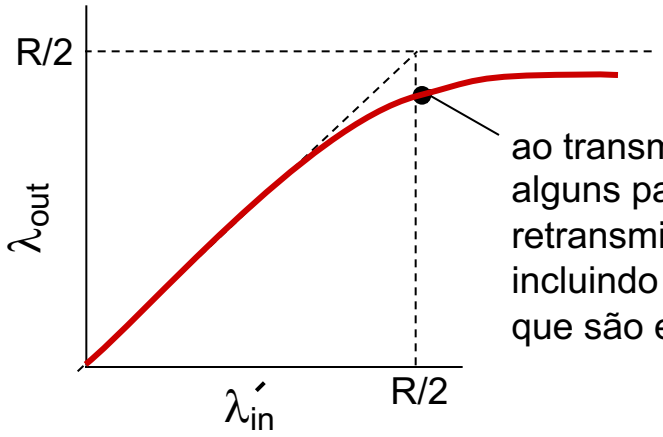


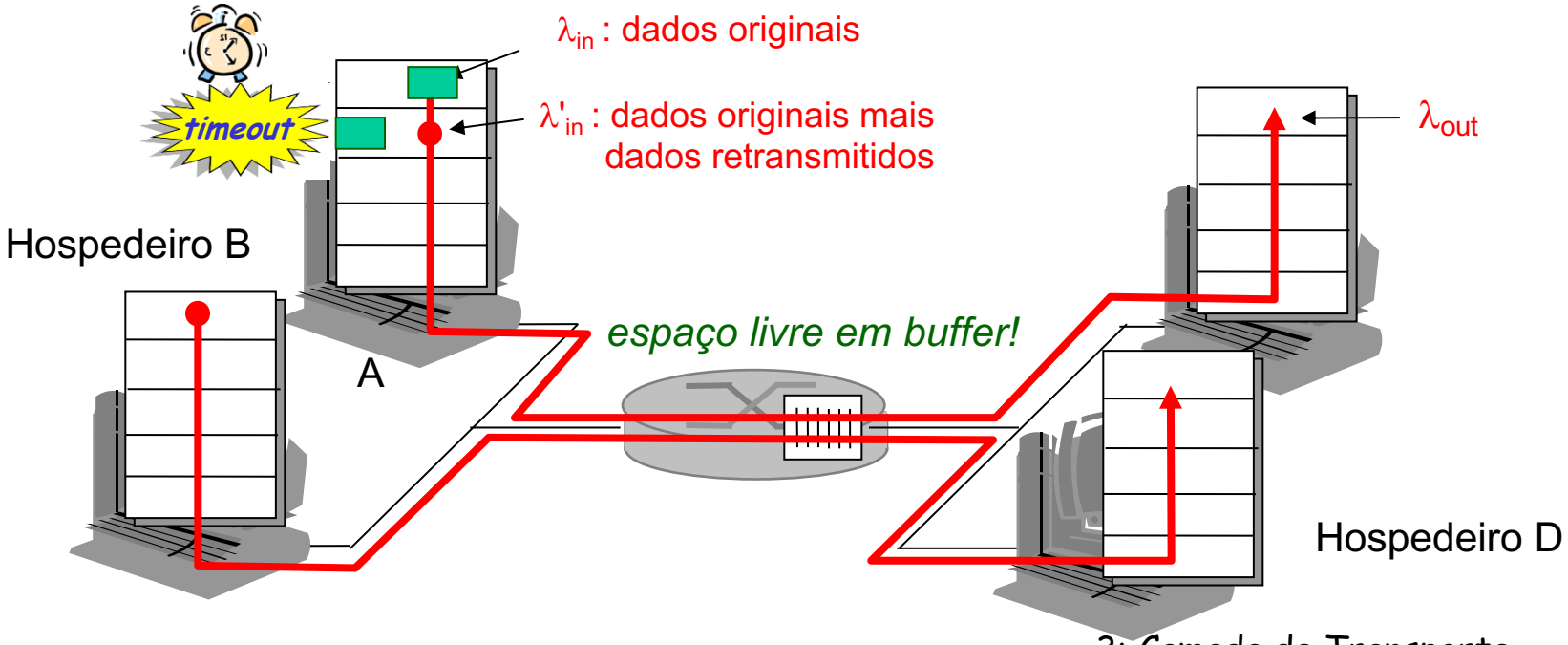


Causas/custos de congest.: cenário 2

Realidade: *duplicatas*

- pacotes podem ser perdidos, descartados no roteador devido a buffers cheios
- retransmissão prematura, envio de duas cópias, ambas entregues.

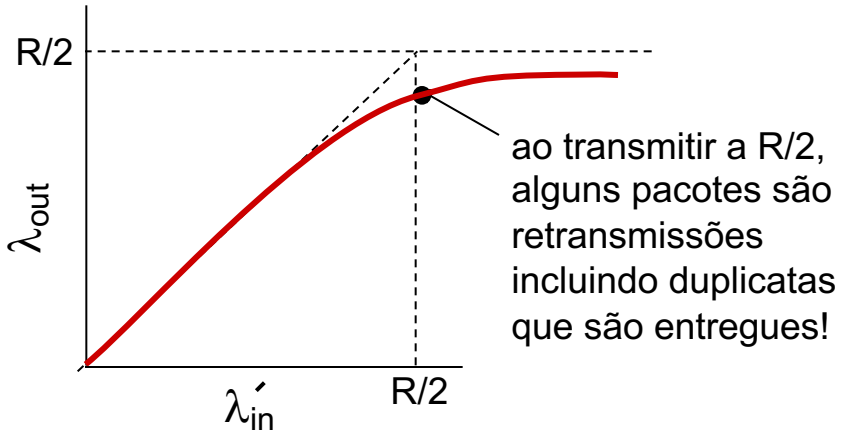




Causas/custos de congest.: cenário 2

Realidade: *duplicatas*

- pacotes podem ser perdidos, descartados no roteador devido a buffers cheios
- retransmissão prematura, envio de duas cópias, ambas entregues.



"custos" do congestionamento:

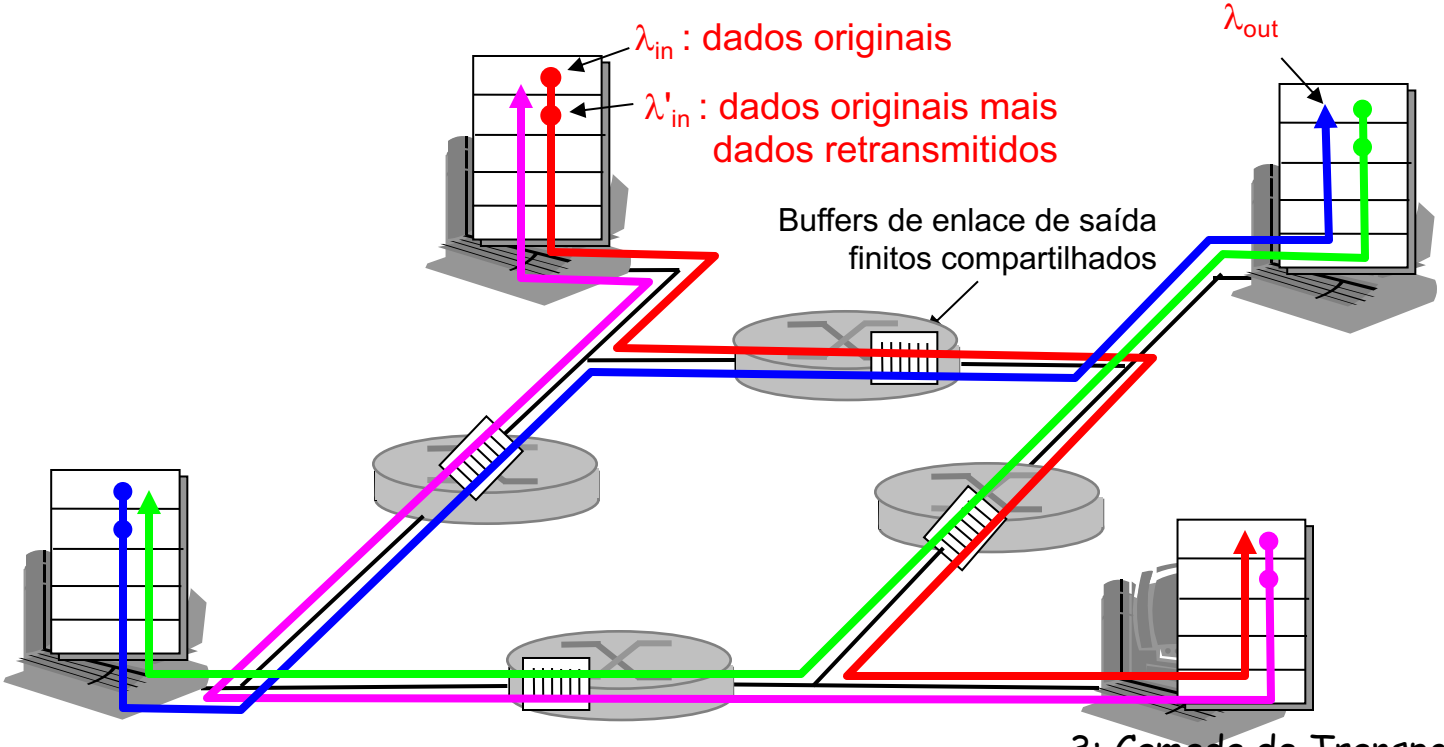
- mais trabalho (retransmissões) para uma dada "*goodput*"
- Retransmissões desnecessárias: link transporta múltiplas cópias do pacote
  - diminuindo a "goodput"

Causas/custos de congestionamento: cenário 3

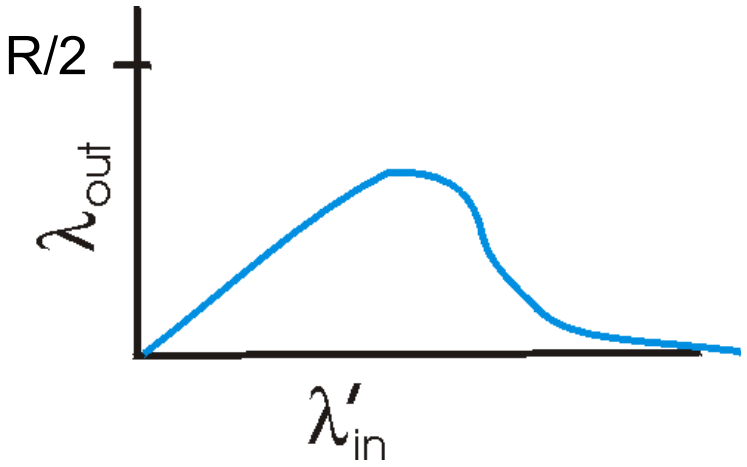
- quatro remetentes
- caminhos com múltiplos enlaces
- temporização/ retransmissão

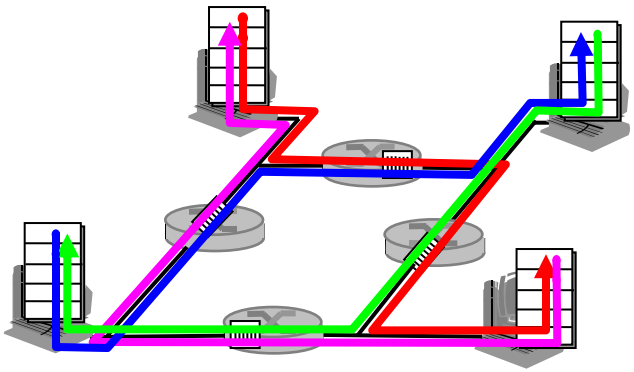
P: o que acontece à medida que λ_{in} e λ'_{in} crescem ?

R: à medida que λ'_{in} vermelho cresce, todos os pacotes azuis que chegam à fila superior são descartados, vazão azul -> 0



Causas/custos de congestionamento: cenário 3





Outro "custo" de congestionamento:

- quando pacote é descartado, qq. capacidade de transmissão já usada (antes do descarte) para esse pacote foi desperdiçada!

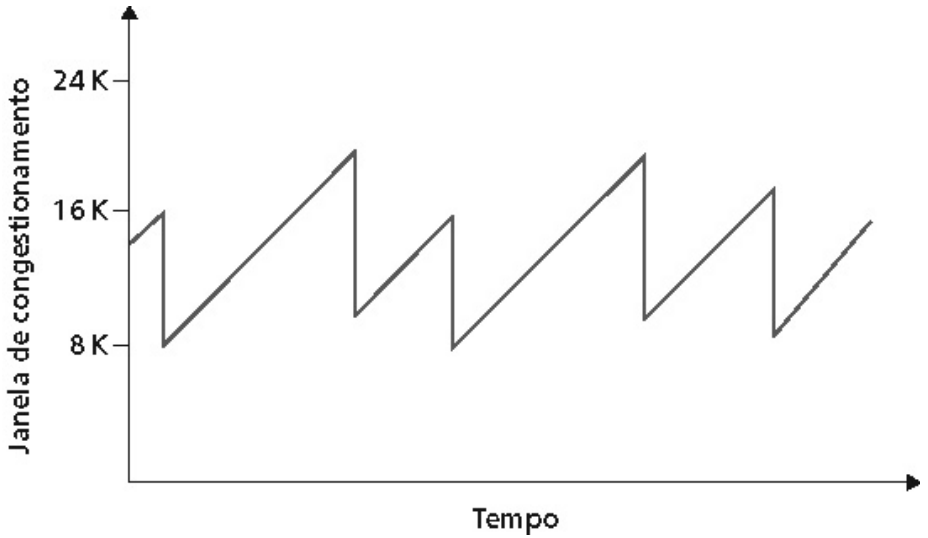
Conteúdo do Capítulo 3

- 3.1 Introdução e serviços de camada de transporte
- 3.2 Multiplexação e demultiplexação
- 3.3 Transporte não orientado para conexão: UDP
- 3.4 Princípios da transferência confiável de dados
- 3.5 Transporte orientado para conexão: TCP
- 3.6 Princípios de controle de congestionamento
- 3.7 Controle de congestionamento no TCP

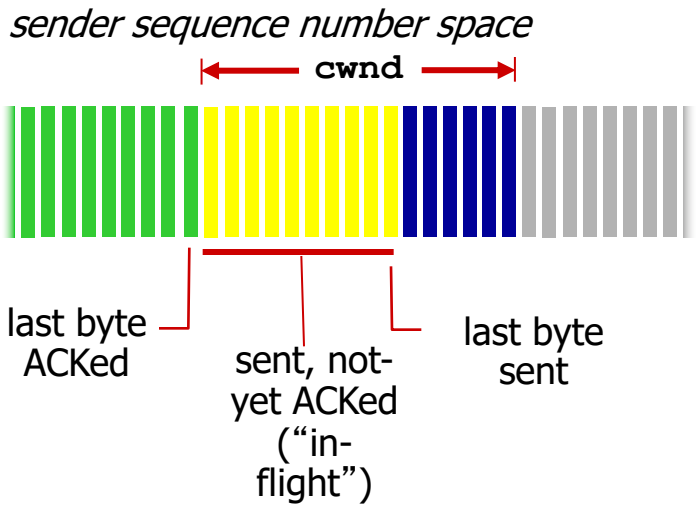
Controle de Congestionamento do TCP: aumento aditivo, diminuição multiplicativa

- Abordagem: aumentar a taxa de transmissão (tamanho da janela), testando a largura de banda utilizável, até que ocorra uma perda
  - aumento aditivo: incrementa **cwnd** de 1 MSS a cada RTT até detectar uma perda
  - diminuição multiplicativa: corta **cwnd** pela metade após evento de perda

Comportamento de dente de serra: testando a largura de banda



Controle de Congestionamento do TCP: detalhes



- transmissor limita a transmissão:

  LastByteSent-LastByteAcked

  ≤ cwnd

- **cwnd** é dinâmica, em função do congestionamento detectado na rede

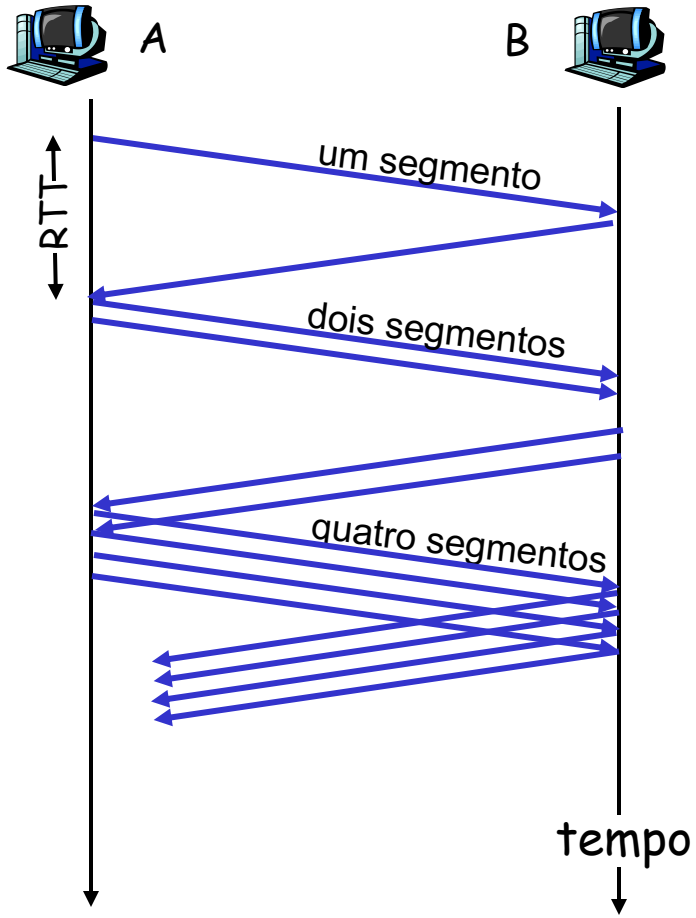
Taxa de transmissão do TCP:

- *aproximadamente*: envia uma janela (cwnd), espera RTT para os ACKs, depois envia mais bytes

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

TCP: Partida lenta

- no início da conexão, aumenta a taxa exponencialmente até o primeiro evento de perda:
  - inicialmente **cwnd** = 1 MSS
  - duplica **cwnd** a cada RTT
  - através do incremento da **cwnd** para cada ACK recebido
- *resumo*: taxa inicial é baixa mas cresce rapidamente de forma exponencial



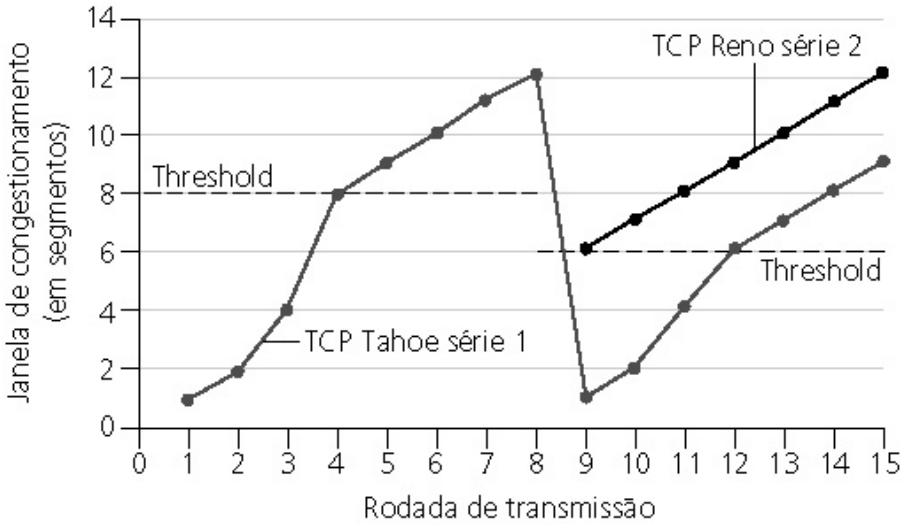
TCP: detectando, reagindo a perdas

- perda indicada pelo estouro de temporizador:
  - **cwnd** é reduzida a 1 MSS;
  - janela cresce exponencialmente (como na partida lenta) até um limiar, depois cresce linearmente
- perda indicada por ACKs duplicados: TCP RENO
  - ACKs duplicados indicam que a rede é capaz de entregar alguns segmentos
  - corta **cwnd** pela metade depois cresce linearmente
- O TCP Tahoe sempre reduz a **cwnd** para 1 (seja por estouro de temporizador que três ACKS duplicados)

TCP: mudando da partida lenta para a CA

P: Quando o crescimento exponencial deve mudar para linear?

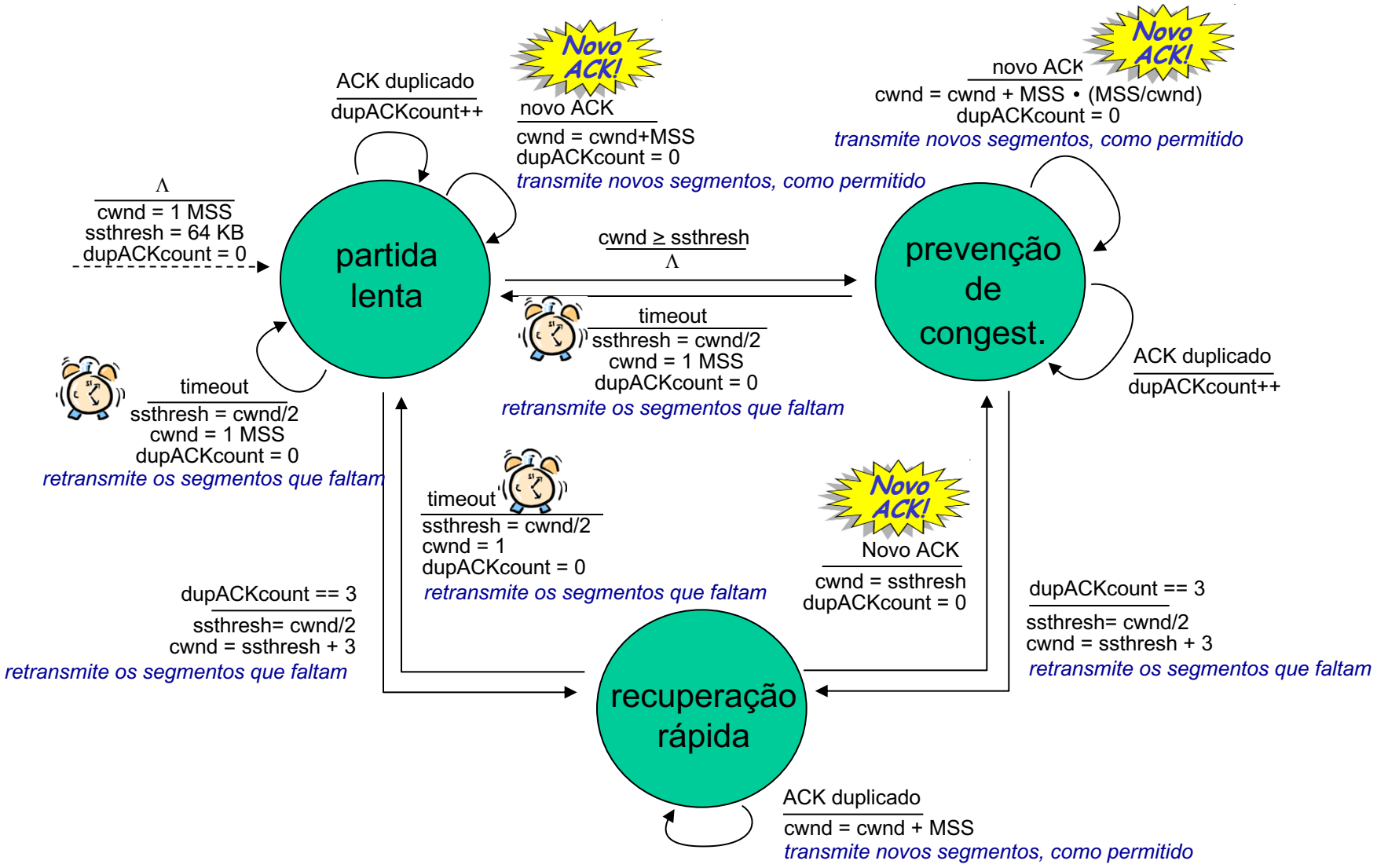
R: Quando **cwnd** atingir 1/2 do seu valor antes da detecção de perda.



Implementação:

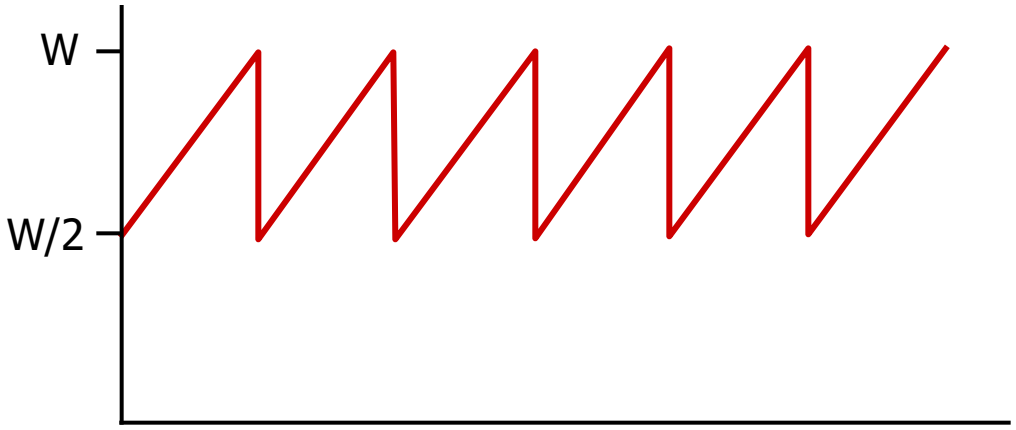
- Limiar (*Threshold*) variável (**ssthresh**)
- Com uma perda o limiar (**ssthresh**) é ajustado para 1/2 da **cwnd** imediatamente antes do evento de perda.

Controle de congestionamento do transmissor TCP



Vazão (*throughput*) do TCP

- Qual é a vazão média do TCP em função do tamanho da janela e do RTT?
  - Ignore a partida lenta, assuma que sempre haja dados a serem transmitidos
- Seja W o tamanho da janela (medida em bytes) quando ocorre uma perda
  - Tamanho médio da janela é $\frac{3}{4}$ W
  - Vazão média é de $\frac{3}{4}$ W por RTT



Futuro do TCP: TCP em "tubos longos e largos"

- exemplo: segmentos de 1500 bytes, RTT de 100ms, deseja vazão de 10 Gbps
- Requer janela de W = 83.333 segmentos em trânsito
- Vazão em termos de taxa de perdas (L) [Mathis 1997]:

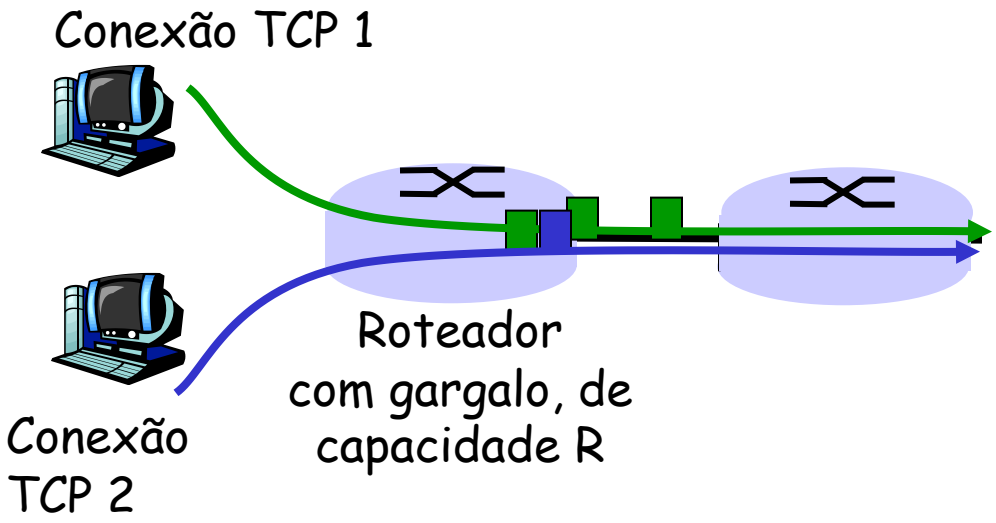
$$\text{vazão do TCP} = \frac{1{,}22 \cdot MSS}{RTT\sqrt{L}}$$

➜ para atingir uma vazão de 10Gbps, seria necessária uma taxa de perdas $L = 2 \cdot 10^{-10}$ *demasiado baixa!!!*

- São necessárias novas versões do TCP para altas velocidades!

Equidade (*Fairness*) do TCP

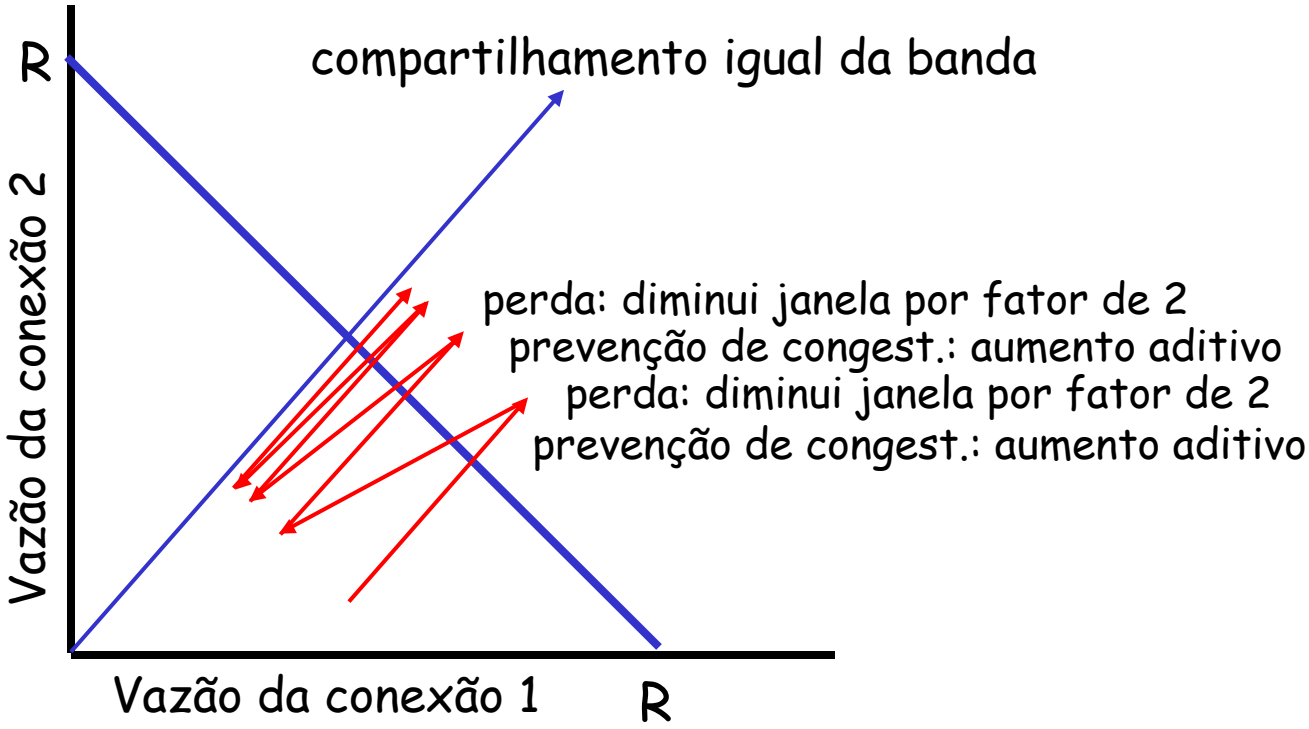
objetivo de equidade: se K sessões TCP compartilham o mesmo enlace de gargalo com largura de banda R, cada uma deve obter uma taxa média de R/K



Por que o TCP é justo?

Duas sessões competindo pela banda:

- Aumento aditivo dá gradiente de 1, enquanto vazão aumenta
- Redução multiplicativa diminui vazão proporcionalmente



Equidade (mais)

Equidade e UDP

- aplicações multimídia frequentemente não usam TCP
  - não querem a taxa estrangulada pelo controle de congestionamento
- preferem usar o UDP:
  - injetam áudio/vídeo a taxas constantes, toleram perdas de pacotes

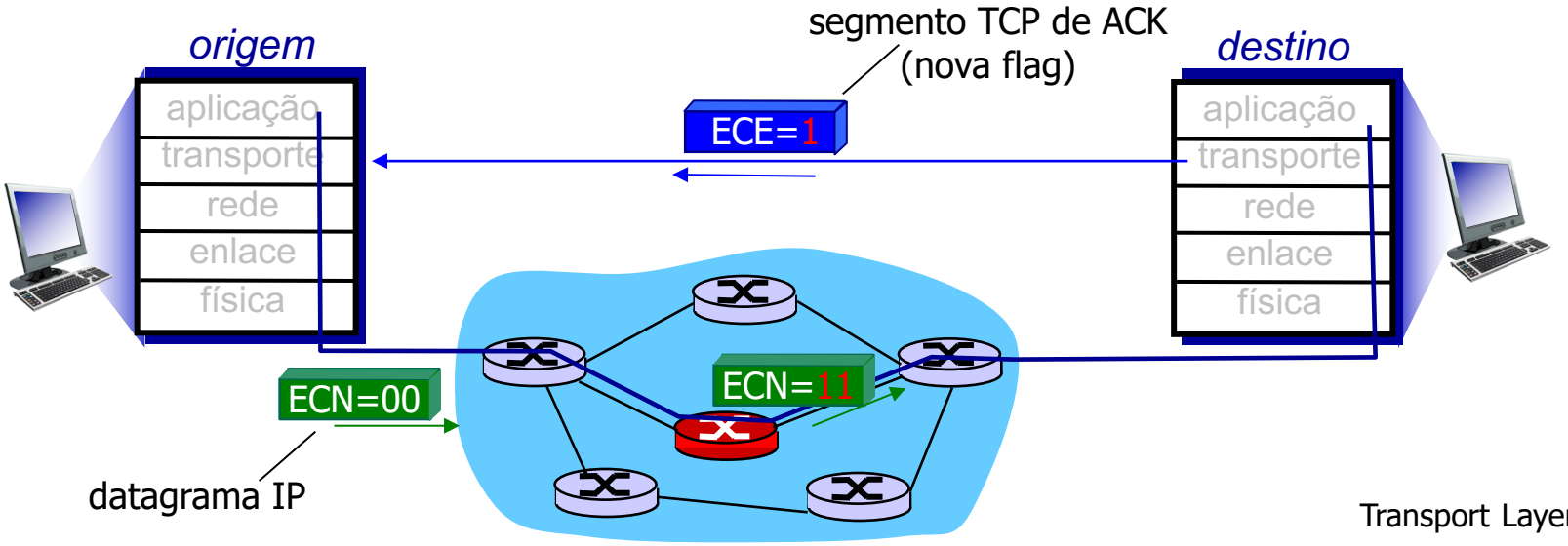
Equidade e conexões TCP em paralelo

- nada impede que as apls. abram conexões paralelas entre 2 hosts
- os *browsers* Web fazem isto
- exemplo: canal com taxa R compartilhado por 9 conexões;
  - novas aplicações pedem 1 TCP, obtém taxa de R/10
  - novas aplicações pedem 11 TCPs, obtém taxa R/2 !

Notificação Explícita de Congestionamento (ECN)

controle de congestionamento assistido pela rede:

- dois bits no cabeçalho IP (campo ToS) são marcados *pelo roteador de rede* para indicar o congestionamento
- indicação de congestionamento é levada até o receptor
- o receptor (vendo a indicação de congestionamento) seta o bit ECE no segmento de reconhecimento para notificar o transmissor sobre o congestionamento.



Capítulo 3: Resumo

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

Próximo capítulo:

- saímos da "borda" da rede (camadas de aplicação e transporte)
- entramos no "núcleo" da rede
- dois capítulos sobre a camada de rede:
  - plano de dados
  - plano de controle