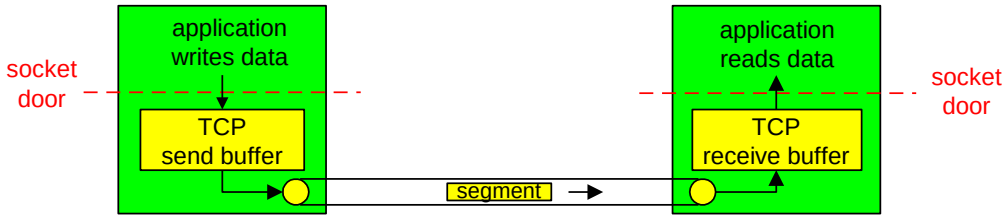
TCP: Visão geral

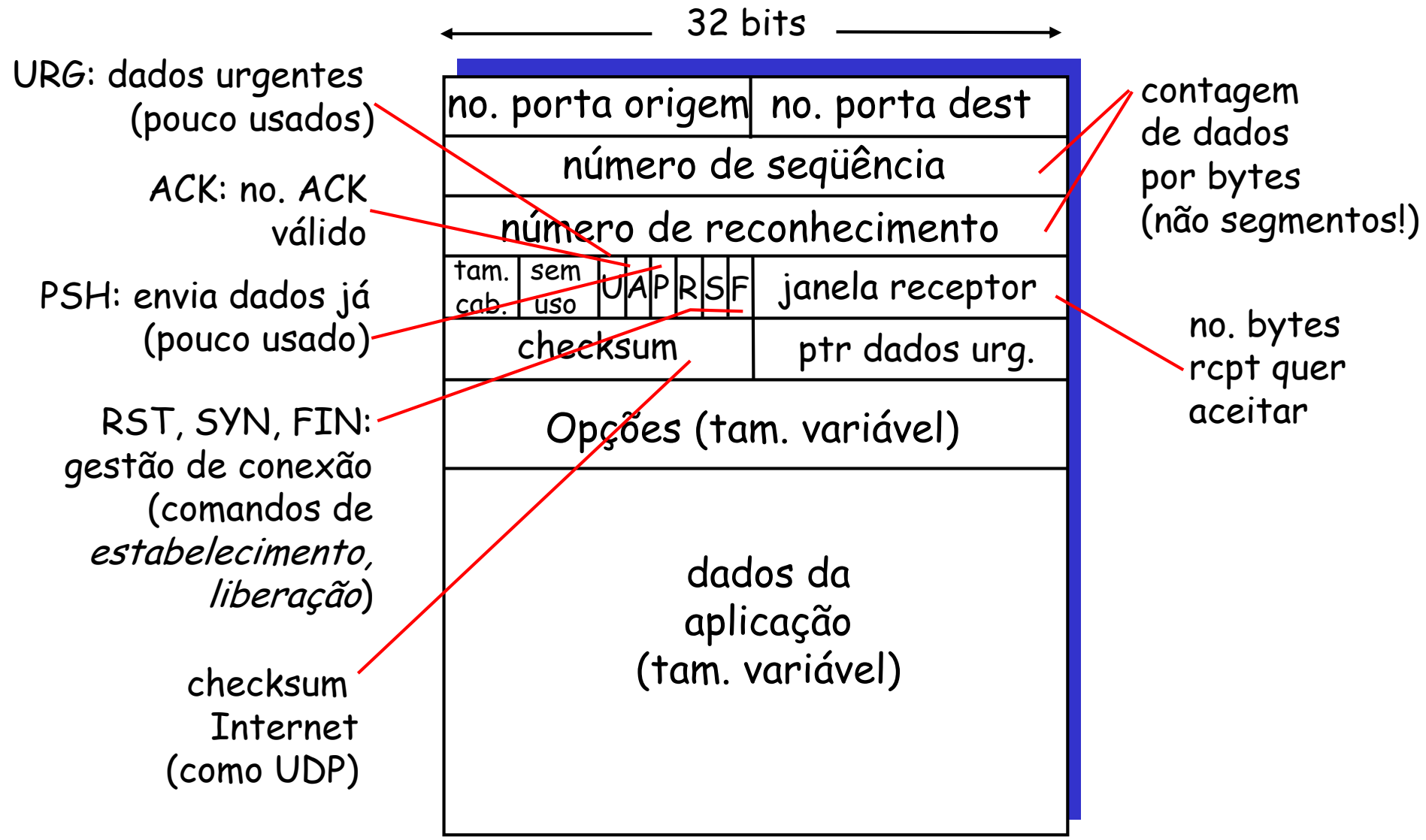
RFCs: 793, 1122, 1323, 2018, 2581

- ponto a ponto:
  - 1 remetente, 1 receptor
- fluxo de bytes, ordenados, confiável:
  - não estruturado em msgs
- dutado:
  - tam. da janela ajustado por controle de fluxo e congestionamento do TCP
- *buffers de envio e recepção*



- 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 de remetente, receptor antes de trocar dados
- fluxo controlado:
  - receptor não será afogado

TCP: estrutura do segmento



TCP: nos. de seq. e ACKs

Nos. de seq.:

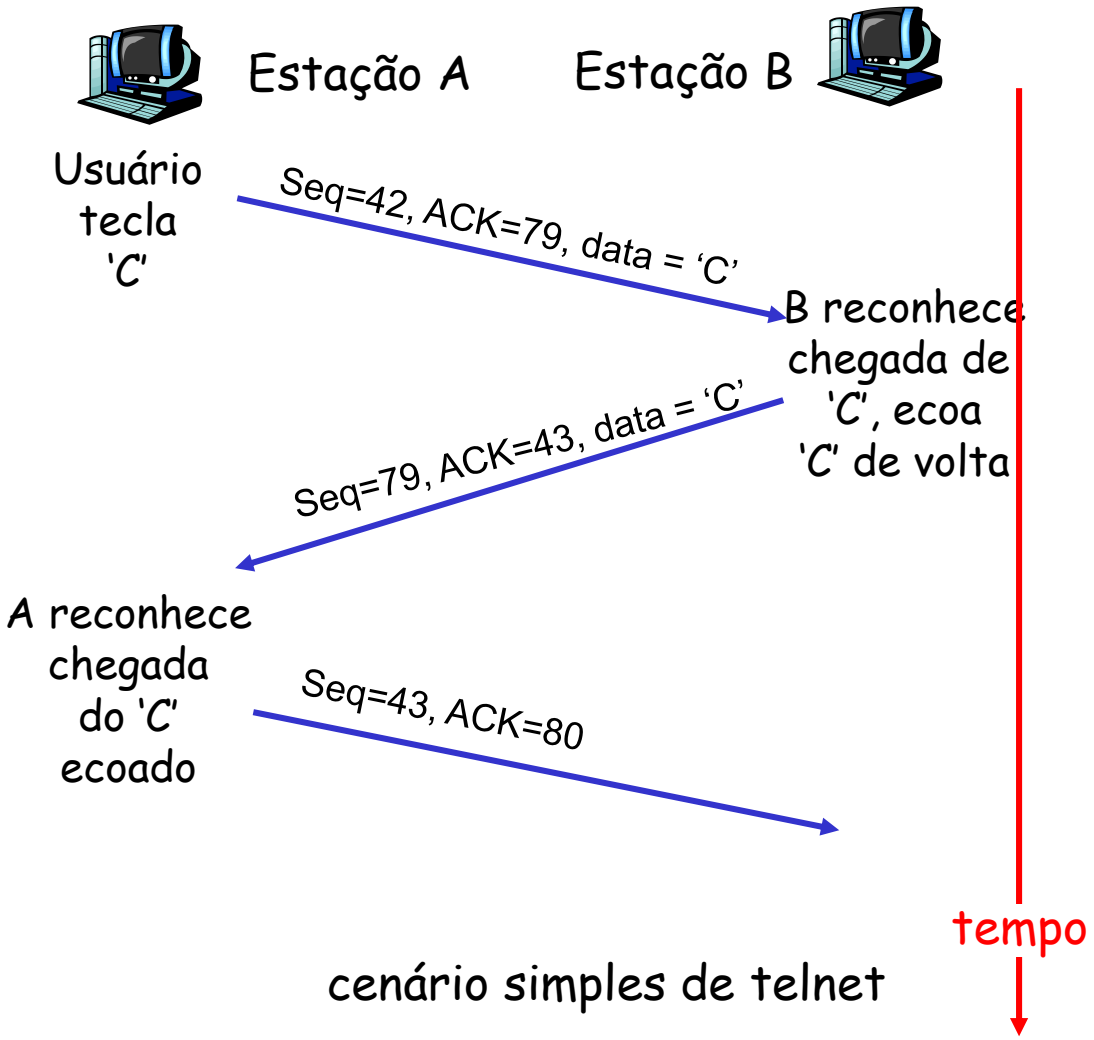
- ✓ "número"dentro do fluxo de bytes do primeiro byte de dados do segmento

ACKs:

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

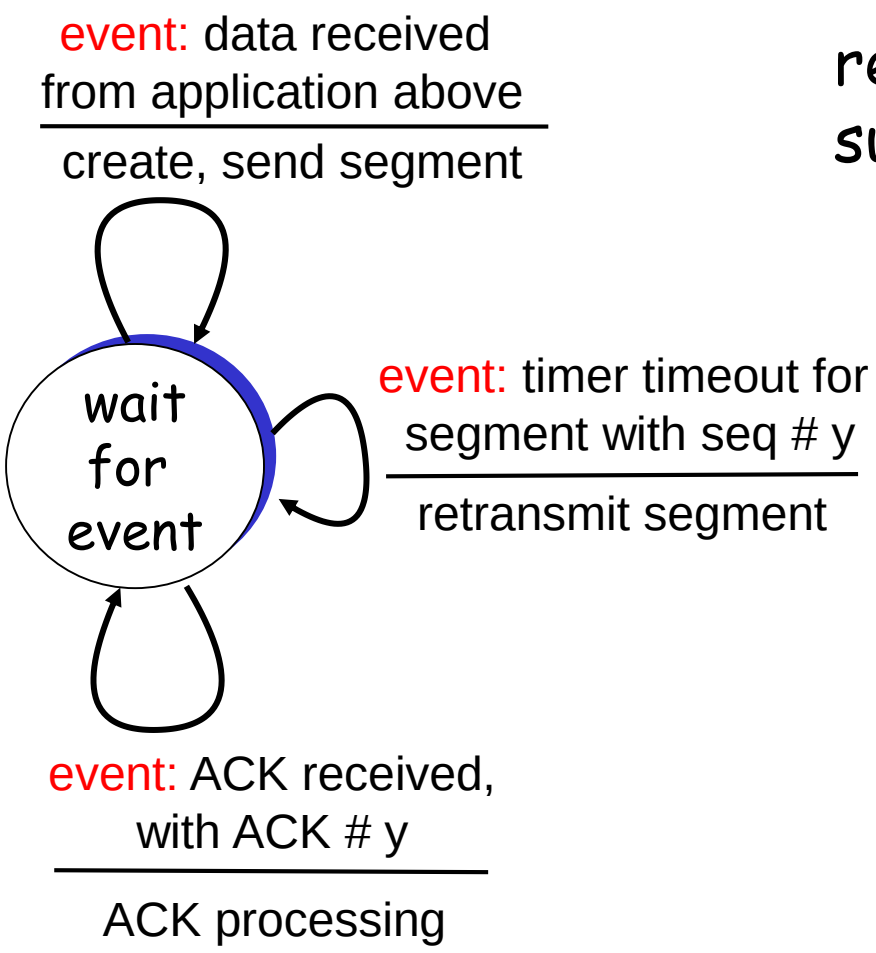
P: como receptor trata segmentos fora da ordem?

- ✓ R: espec do TCP omissa - deixado ao implementador



cenário simples de telnet

TCP: transferência confiável de dados



remetente simplificado, supondo:

- fluxo de dados uni-direcional
- sem controle de fluxo, congestionamento

TCP: transfe-rência confiável de dados

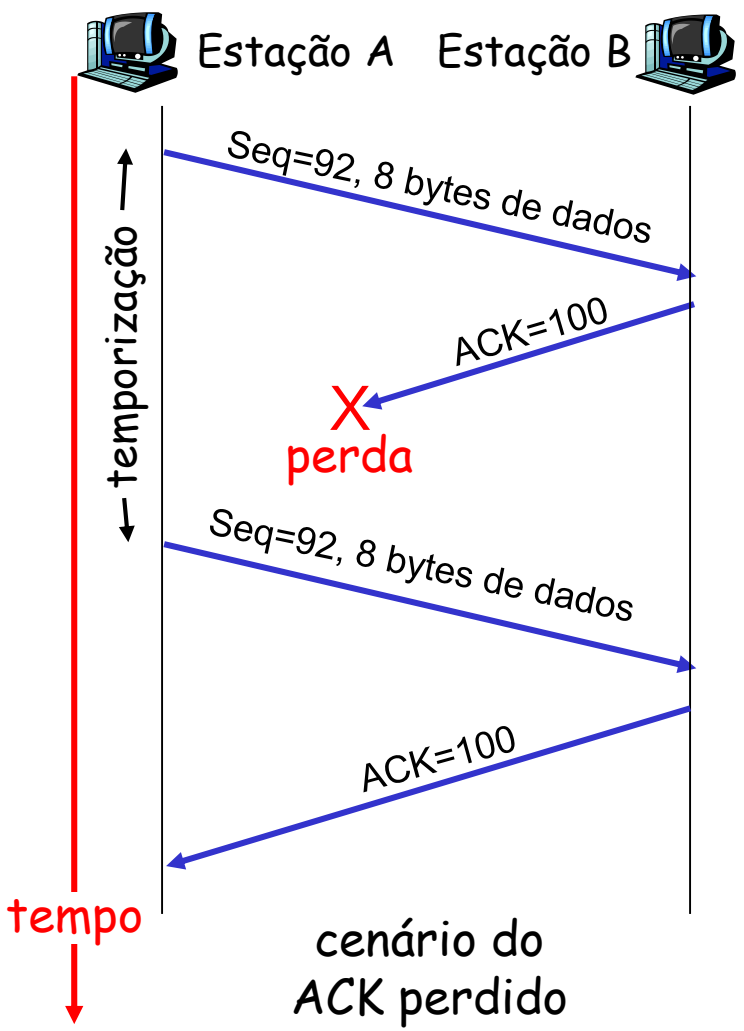
Remetente TCP simplificado

```
00  sendbase = número de seqüência inicial
01   nextseqnum = número de seqüência inicial
02
03    loop (forever) {
04      switch(event)
05      event: dados recebidos da aplicação acima
06           cria segmento TCP com número de seqüência nextseqnum
07           inicia temporizador para segmento nextseqnum
08           passa segmento para IP
09           nextseqnum = nextseqnum + comprimento(dados)
10       event: expirado temporizador de segmento c/ no. de seqüência y
11           retransmite segmento com número de seqüência y
12           calcula novo intervalo de temporização para segmento y
13           reinicia temporizador para número de seqüência y
14       event: ACK recebido, com valor de campo ACK de y
15           se (y > sendbase) { /* ACK cumulativo de todos dados até y */
16              cancela temporizadores p/ segmentos c/ nos. de seqüência < y
17               sendbase = y
18               }
19           senão { /* é ACK duplicado para segmento já reconhecido */
20               incrementa número de ACKs duplicados recebidos para y
21               if (número de ACKs duplicados recebidos para y == 3) {
22                   /* TCP:  retransmissão rápida */
23                  reenvia segmento com número de seqüência y
24                   reinicia temporizador para número de seqüência y
25              }
26      }  /* fim de loop forever */
```

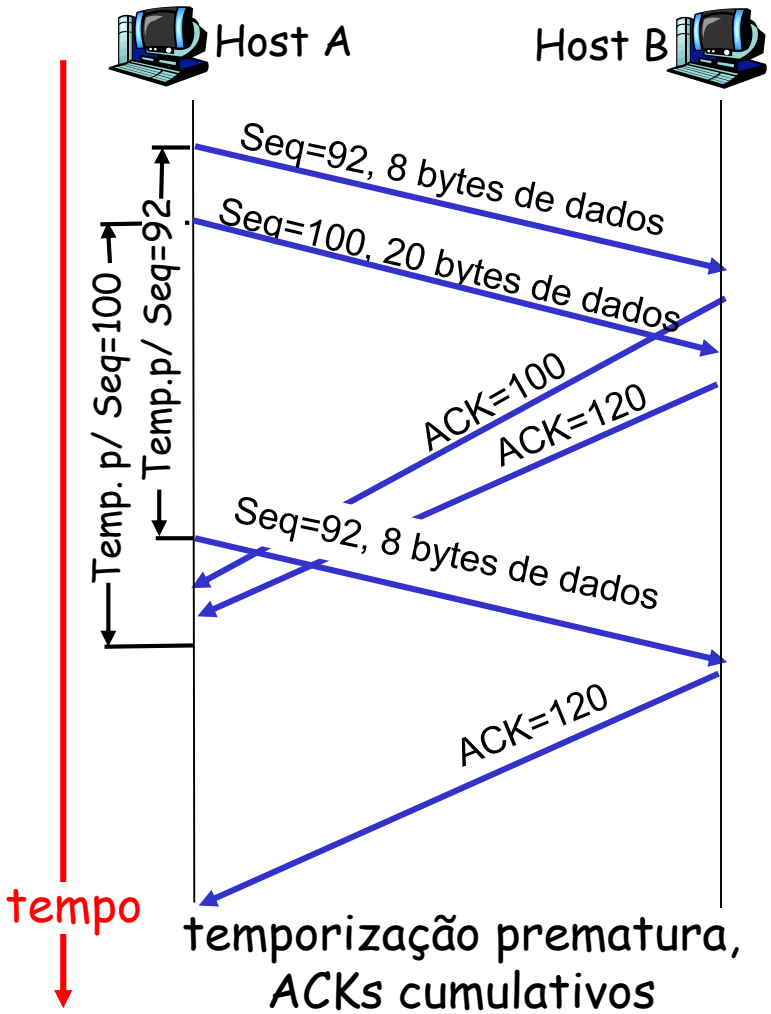
TCP geração de ACKs [RFCs 1122, 2581]

Evento	Ação do receptor TCP
chegada de segmento em ordem sem lacunas, anteriores já reconhecidos	ACK retardado. Espera até 500ms p/ 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 no início da lacuna

TCP: cenários de retransmissão



cenário do
ACK perdido



temporização prematura,
ACKs cumulativos

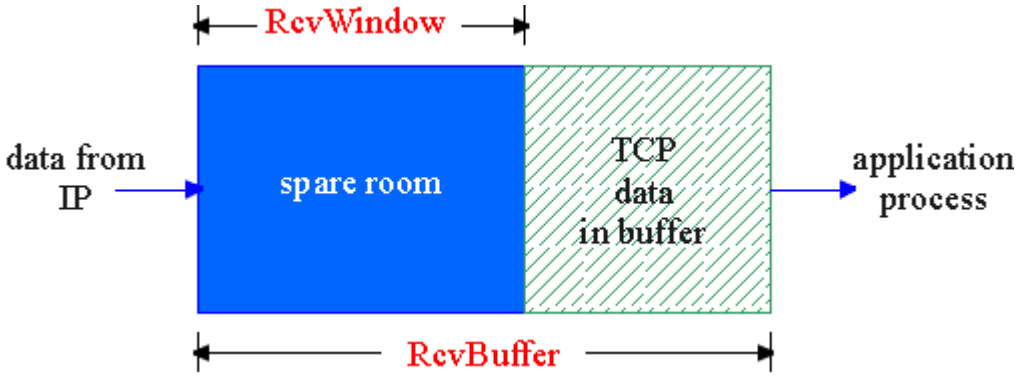
TCP: Controle de Fluxo

controle de fluxo

remetente não esgotaria buffers do receptor por transmitir muito, ou muito rápidamente

`RcvBuffer` = tamanho do Buffer de recepção

`RcvWindow` = espaço vazio no Buffer



buffering pelo receptor

receptor: explicitamente avisa o remetente da quantidade de espaço livre disponível (muda dinamicamente)

- ✓ **campo `RcvWindow` no** segmento TCP

remetente: mantém a quantidade de dados transmitidos, porém ainda não reconhecidos, menor que o valor mais recente de **`RcvWindow`**

TCP: Tempo de Resposta (RTT) e Temporização

P: como escolher valor do temporizador TCP?

- maior que o RTT
  - note: RTT pode variar
- muito curto: temporização prematura
  - retransmissões são desnecessárias
- muito longo: reação demorada à perda de segmentos

P: como estimar RTT?

- **RTTamostra**: tempo medido entre a transmissão do segmento e o recebimento do ACK correspondente
  - ignora retransmissões, segmentos com ACKs cumulativos
- **RTTamostra** vai variar, queremos "amaciador" de RTT estimado
  - usa várias medições recentes, não apenas o valor corrente (**RTTamostra**)

TCP: Tempo de Resposta (RTT) e Temporização

RTT_estimado = (1-x)* RTT_estimado + x*RTT_amostra

- média corrente exponencialmente ponderada
- influência de cada amostra diminui exponencialmente com o tempo
- valor típico de x: 0.1

Escolhendo o intervalo de temporização

- **RTT_estimado** mais uma "margem de segurança"
- variação grande em **RTT_estimado**
  -> margem de segurança maior

Temporização = RTT_estimado + 4*Desvio

Desvio = (1-x)* Desvio +
x*|RTT_amostra - RTT_estimado|

TCP: Gerenciamento de Conexões

Lembrete: Remetente, receptor TCP estabelecem "conexão" antes de trocar segmentos de dados

- inicializam variáveis TCP:
  - ✓ nos. de seq.
  - ✓ buffers, info s/ controle de fluxo (p.ex. **RcvWindow**)
- *cliente:* iniciador de conexão

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

- *servidor:* contactado por cliente

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

Inicialização em 3 tempos:

Passo 1: sistema cliente envia segmento de controle SYN do TCP ao servidor

- ✓ especifica no. inicial de seq

Passo 2: sistema servidor recebe SYN, responde com segmento de controle SYNACK

- ✓ reconhece SYN recebido
- ✓ aloca buffers
- ✓ especifica no. inicial de seq. servidor-> receptor

TCP: Gerenciamento de Conexões (cont.)

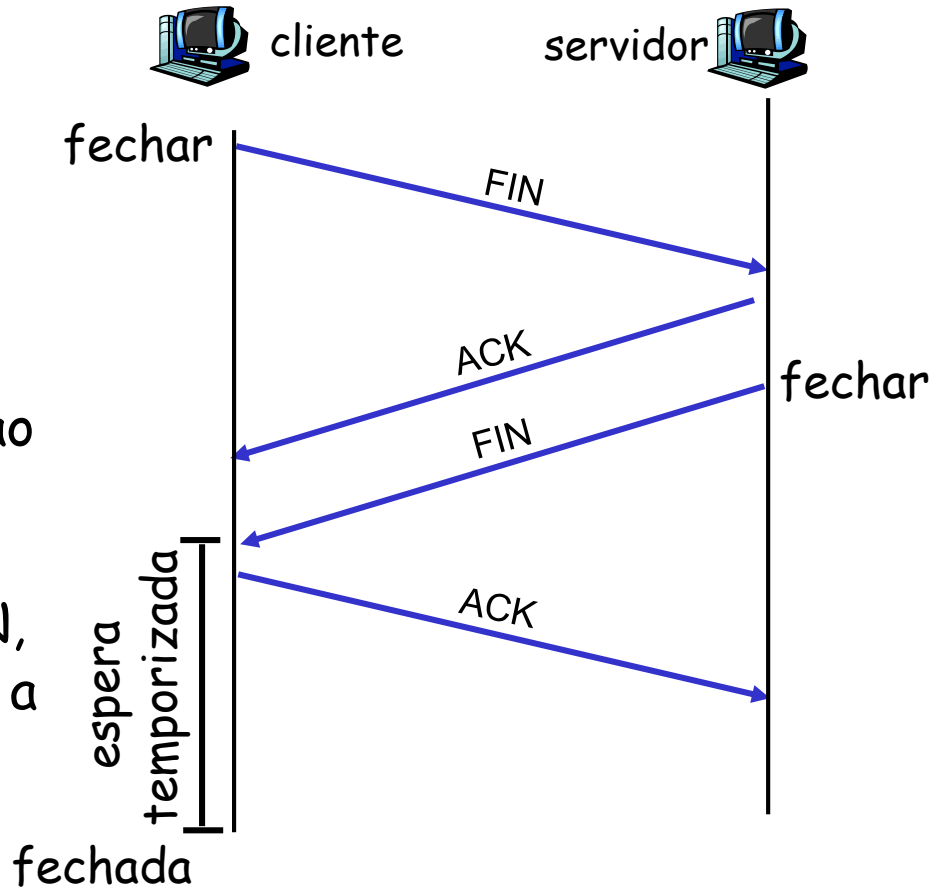
Encerrando uma conexão:

cliente fecha soquete:

```
clientSocket.close();
```

Passo 1: sistema cliente envia segmento de controle FIN ao servidor

Passo 2: servidor recebe FIN, responde com ACK. Encerra a conexão, enviando FIN.



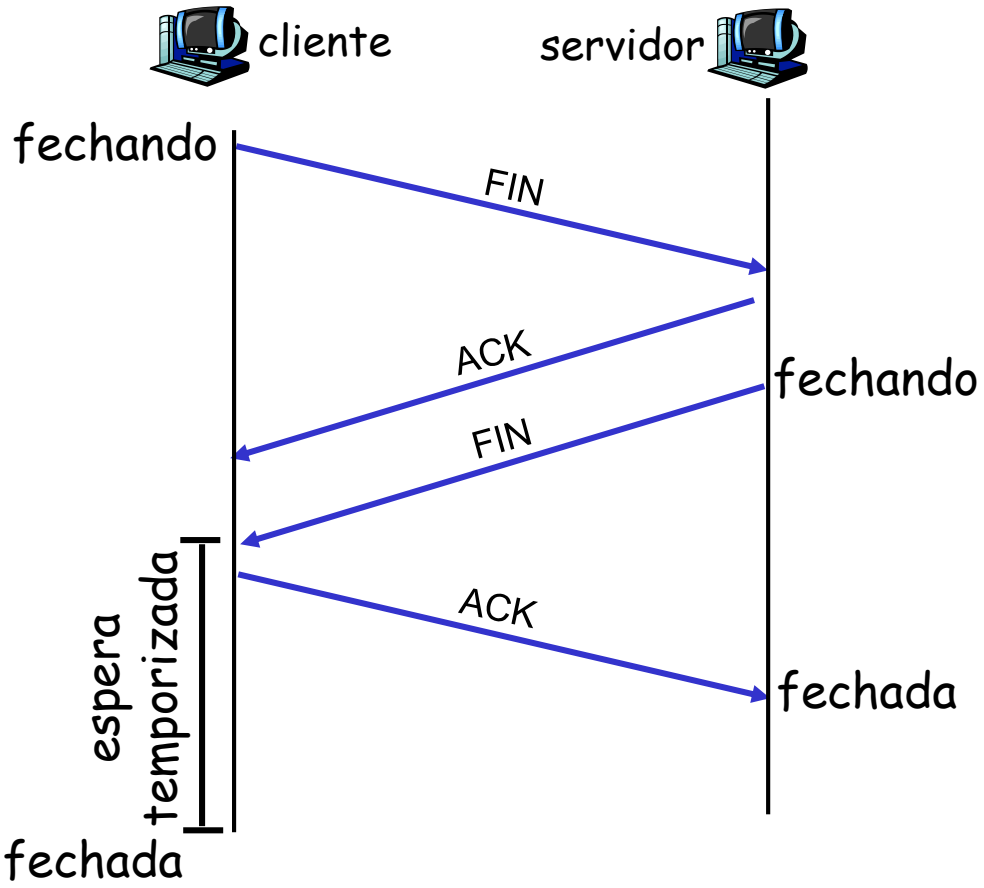
TCP: Gerenciamento de Conexões (cont.)

Passo 3: cliente recebe FIN, responde com ACK.

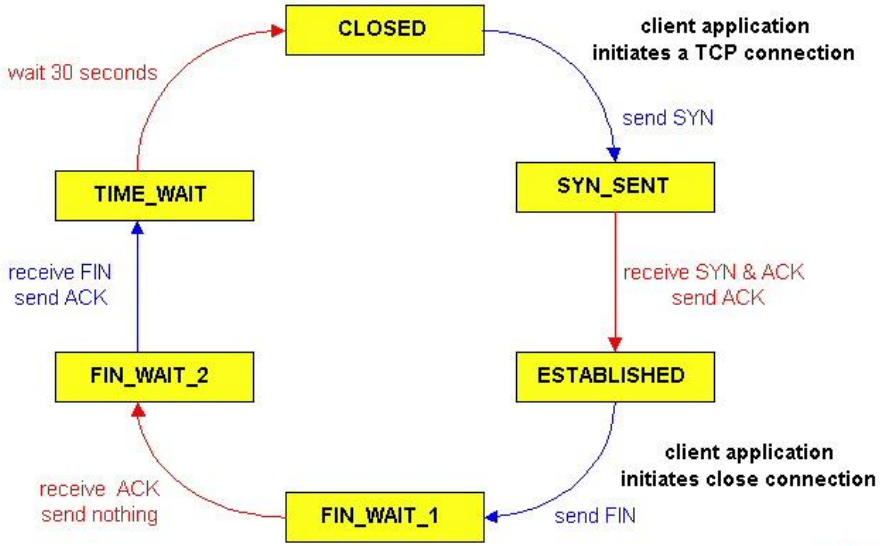
- ✓ Entre em "espera temporizada" - responderá com ACK a FINs recebidos

Step 4: servidor, recebe ACK. Conexão encerrada.

Note: com pequena modificação, consegue tratar de FINs simultâneos.

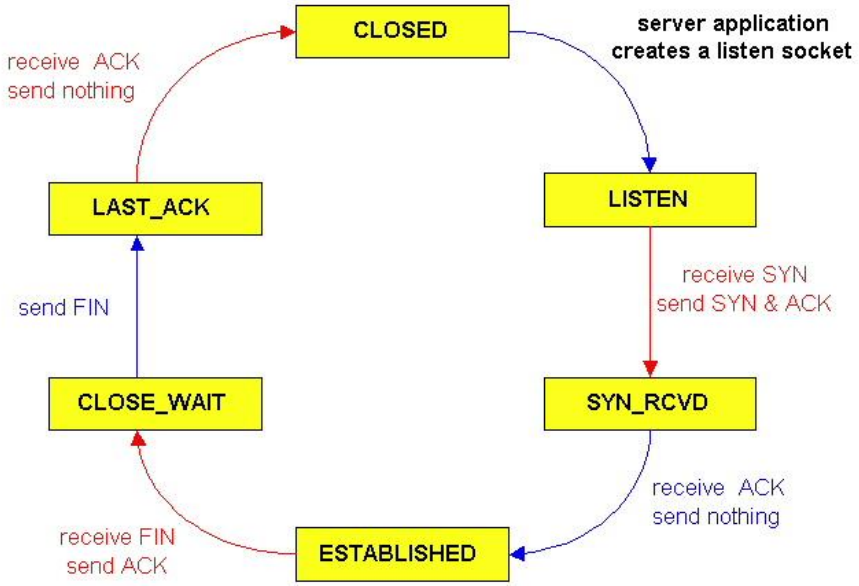


TCP: Gerenciamento de Conexões (cont.)



Ciclo de vida
de cliente TCP

Ciclo de vida
de servidor TCP



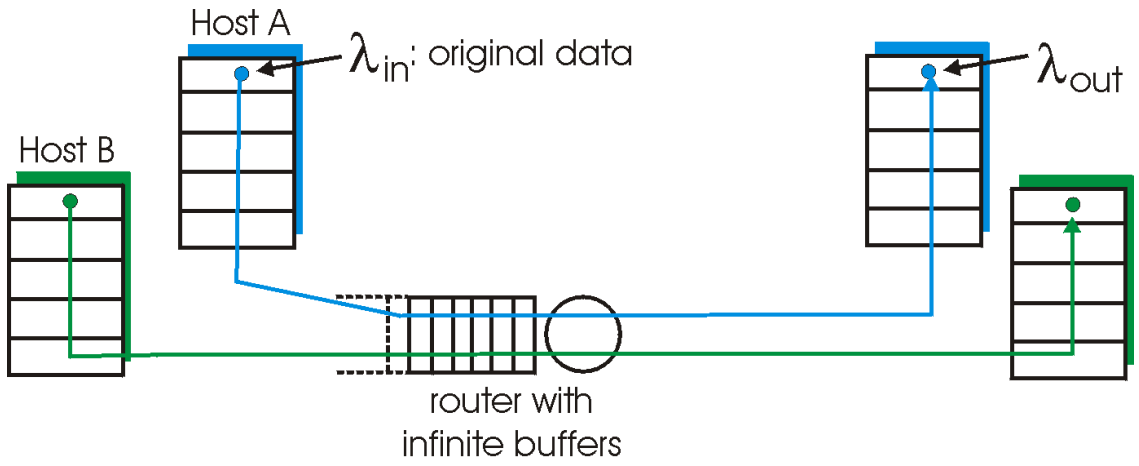
Princípios de Controle de Congestionamento

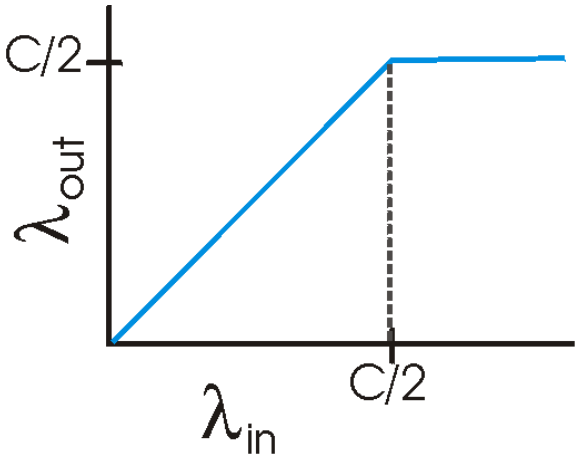
Congestionamento:

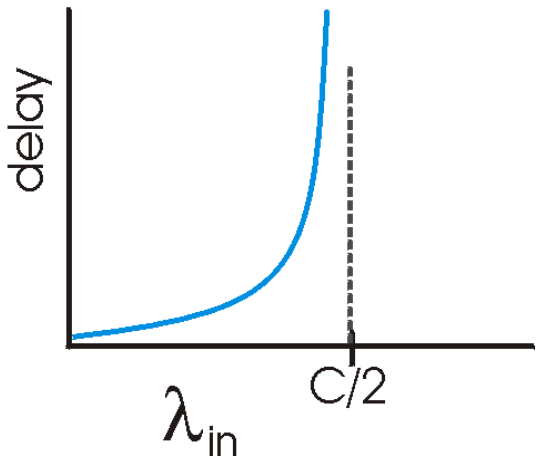
- informalmente: "muitas fontes enviando muitos dados muito rapidamente para a *rede* poder tratar"
- diferente de controle de fluxo!
- manifestações:
  - ✓ perda de pacotes (esgotamento de buffers em roteadores)
  - ✓ longos atrasos (enfileiramento nos buffers dos roteadores)
- um dos 10 problemas mais importantes em redes!

Causas/custos de congestionamento: cenário 1

- dois remetentes, dois receptores
- um roteador, buffers infinitos
- sem retransmissão



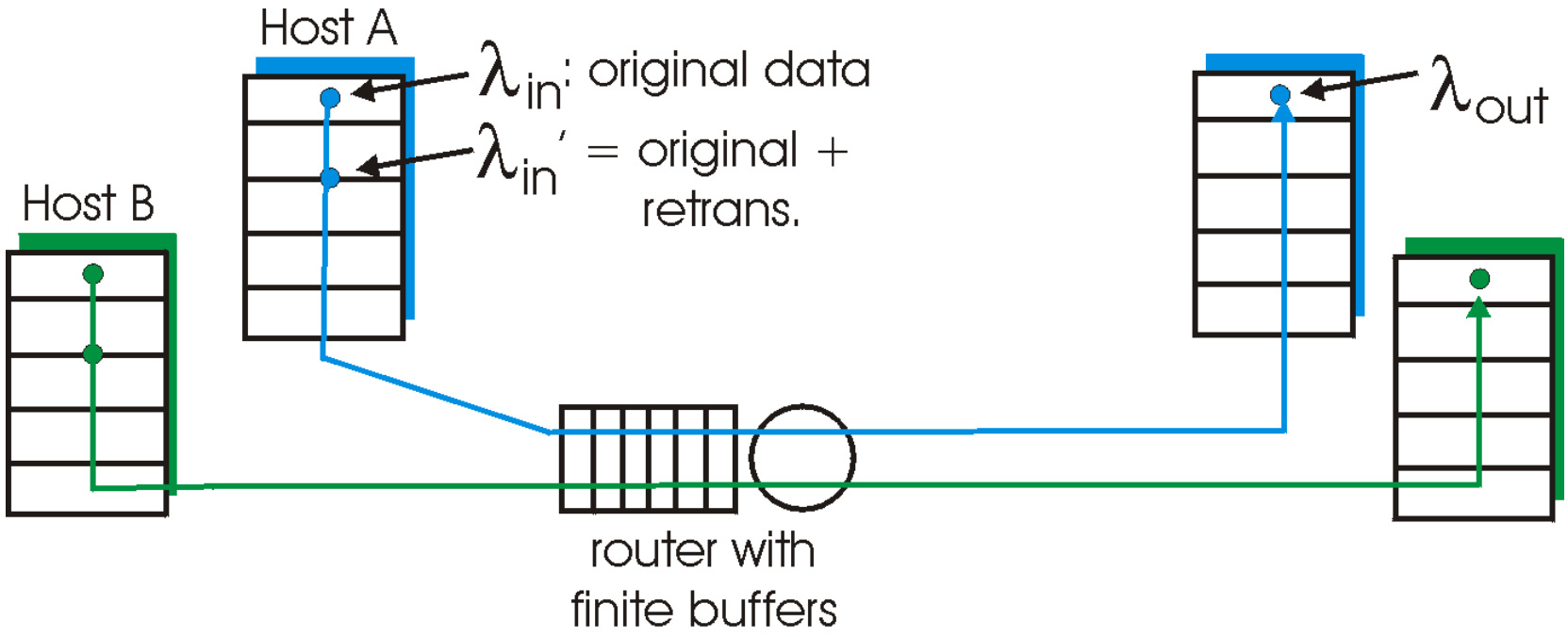




- grandes retardos qdo. congestionada
- vazão máxima alcançável

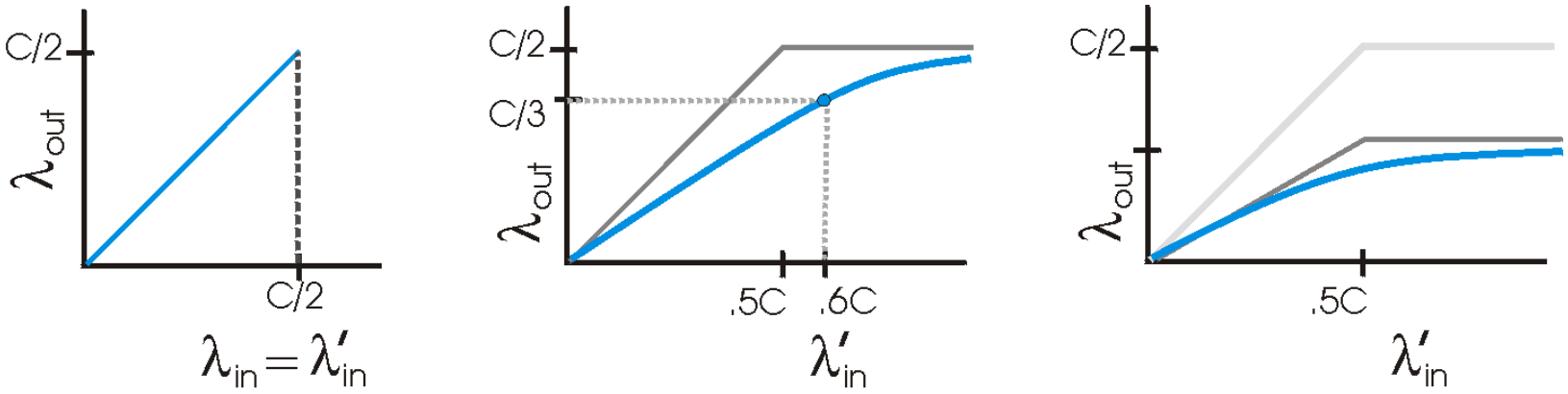
Causas/custos de congestionamento: cenário 2

- Um roteador, buffers *finitos*
- retransmissão pelo remetente de pacote perdido



Causas/custos de congestionamento: cenário 2

- sempre: $\lambda_{in} = \lambda_{out}$ ("goodput")
- retransmissão "perfeito" apenas quando perda: $\lambda'_{in} > \lambda_{out}$
- retransmissão de pacote atrasado (não perdido) faz λ'_{in} maior (que o caso perfeito) para o mesmo λ_{out}



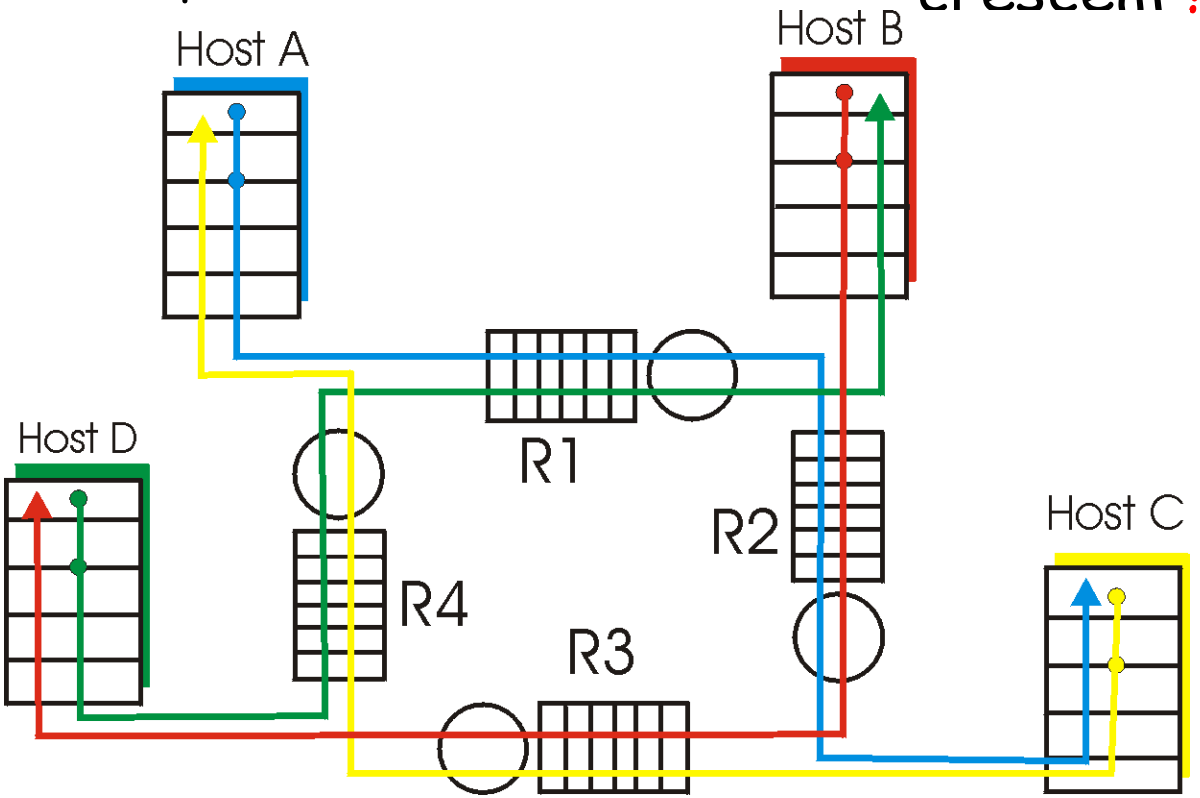
"custos" de congestionamento:

- mais trabalho (retransmissão) para dado "goodput"
- retransmissões desnecessárias: enviadas múltiplas cópias do pacote

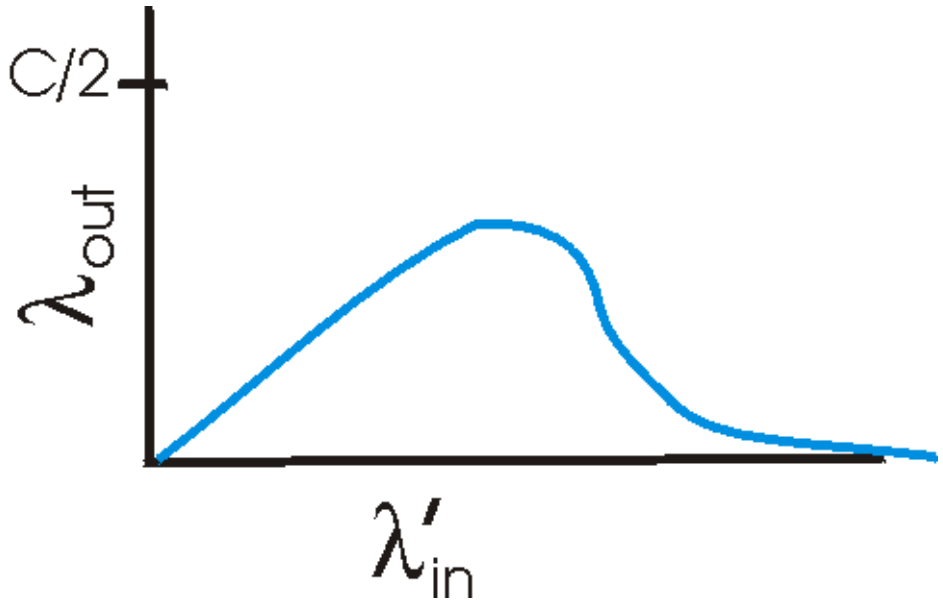
Causas/custos de congestionamento: cenário 3

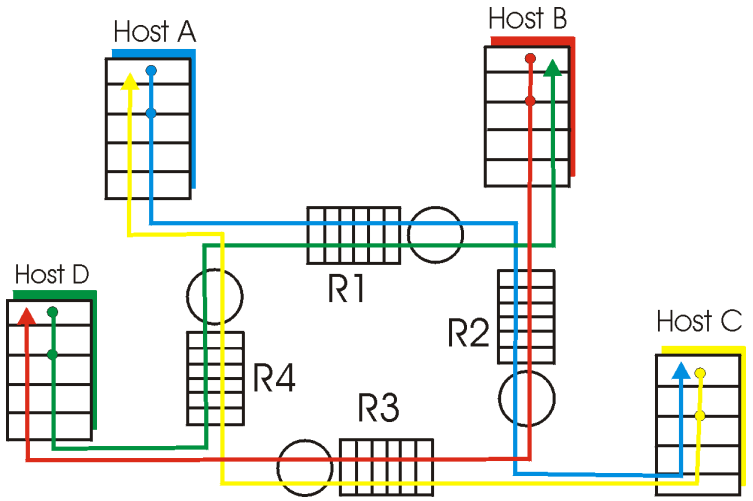
- quatro remetentes
- caminhos com múltiplos enlaces
- temporização/retransmissão

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



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 desperdiçado!

Abordagens de controle de congestionamento

Duas abordagens amplas para controle de congestionamento:

Controle de congestionamento fim a fim :

- não tem realimentação explícita pela rede
- congestionamento inferido das perdas, retardo observados pelo sistema terminal
- abordagem usada pelo TCP

Controle de congestionamento com apoio da rede:

- roteadores realimentam os sistemas terminais
  - ✓ bit único indicando congestionamento (SNA, DECbit, TCP/IP ECN, ATM)
  - ✓ taxa explícita p/ envio pelo remetente

Estudo de caso: controle de congestionamento no ABR da ATM

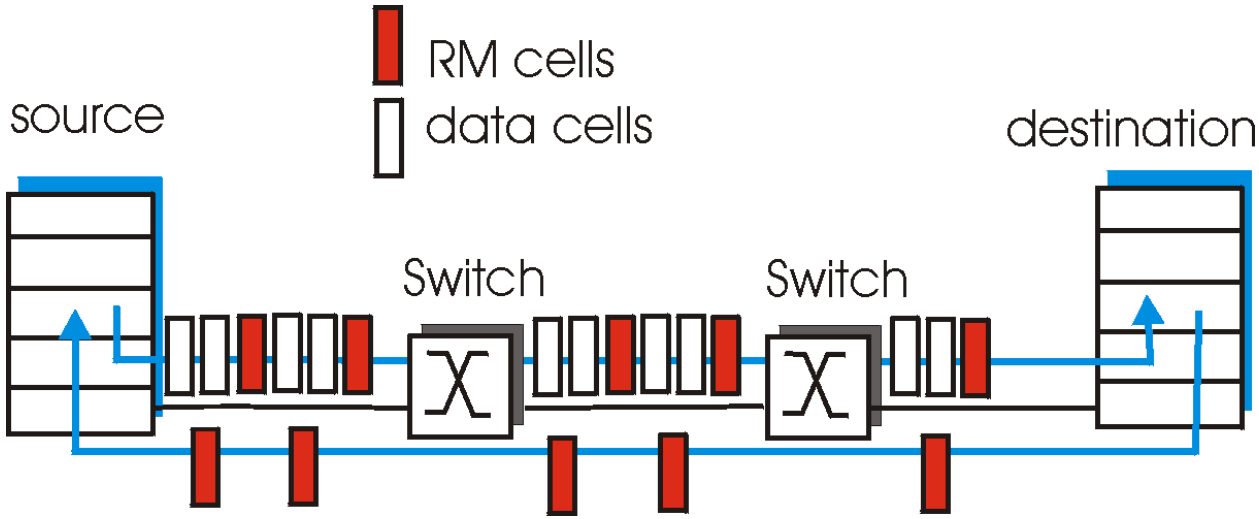
ABR: available bit rate:

- "serviço elástico"
- se caminho do remetente "sub-carregado":
  - ✓ remetente deveria usar banda disponível
- se caminho do remetente congestionado:
  - ✓ remetente reduzido à taxa mínima garantida

células RM (resource management):

- enviadas pelo remetente, intercaladas com células de dados
- bits na célula RM iniciados por comutadores ("*apoio da rede*")
  - ✓ bit NI: não aumente a taxa (congestionamento moderado)
  - ✓ bit CI: indicação de congestionamento
- células RM devolvidos ao remetente pelo receptor, sem alteração dos bits

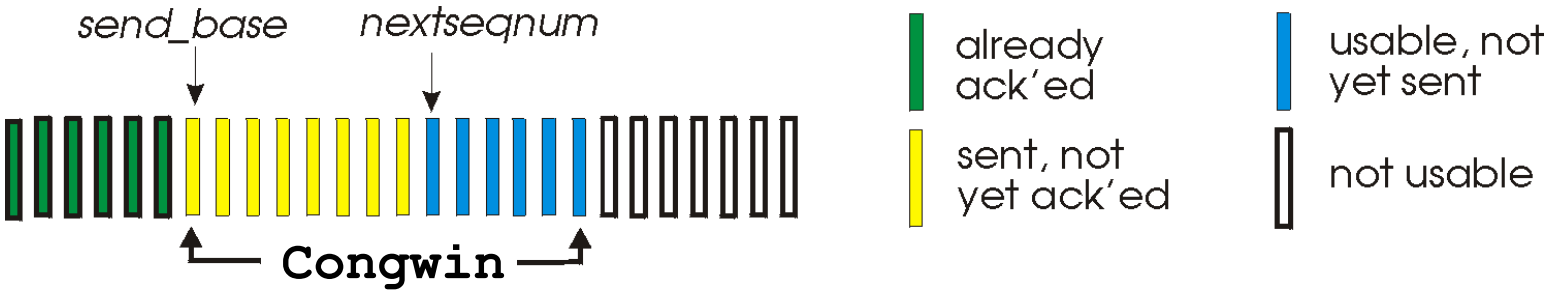
Estudo de caso: controle de congestionamento em ABR da ATM



- Campo ER (explicit rate) de 2 bytes na célula RM
  - ✓ comutador congestionado pode diminuir valor ER na célula
  - ✓ taxa do remetente assim ajustada p/ menor valor possível entre os comutadores do caminho
- bit EFCI em células de dados ligado por comutador congestionado
  - ✓ se EFCI ligado na célula de dados antes da célula RM, receptor liga bit CI na célula RM devolvida

TCP: Controle de Congestionamento

- controle fim a fim (sem apoio da rede)
- taxa de transmissão limitada pela tamanho da janela de congestionamento, **Congwin**:



- w segmentos, cada um c/ MSS bytes, enviados por RTT:

$$\text{throughput} = \frac{w * MSS}{RTT} \text{ Bytes/sec}$$

TCP: Controle de Congestionamento

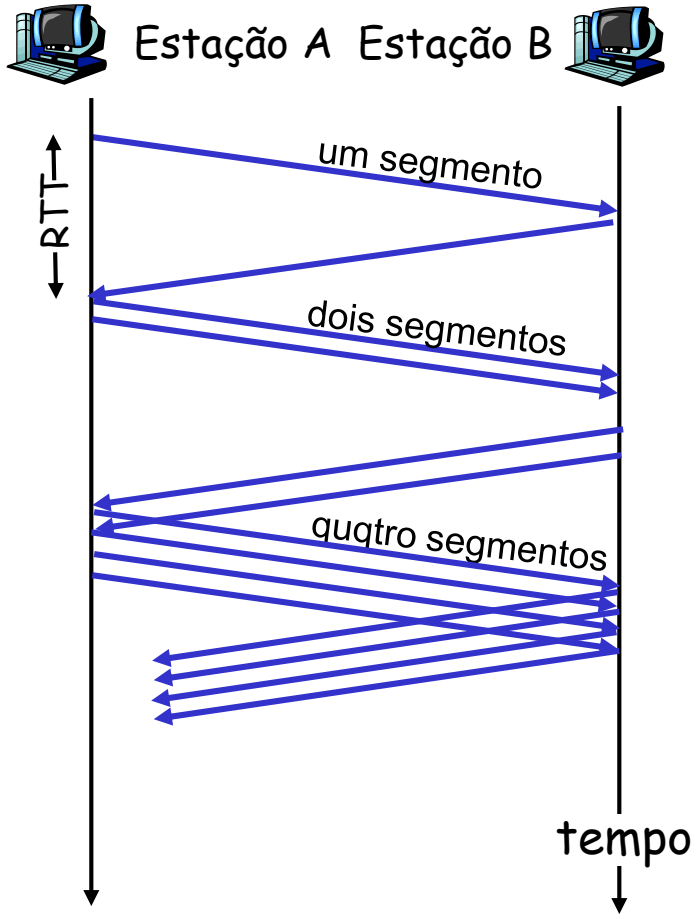
- "sondagem" para banda utilizável:
  - idealmente: transmitir o mais rápido possível (**Congwin** o máximo possível) sem perder pacotes
  - *aumentar* **Congwin** até perder pacotes (congestionamento)
  - perdas: *diminui* **Congwin**, depois volta a à sondagem (aumento) novamente
- duas "fases"
  - partida lenta
  - evitar congestionamento
- variáveis importantes:
  - **Congwin**
  - **threshold**: define limiar entre fases de partida lenta, controle de congestionamento

TCP: Partida lenta

Algoritmo Partida Lenta

inicializa: Congwin = 1
for (cada segmento c/ ACK)
    Congwin++
until (evento de perda OR
    CongWin > threshold)

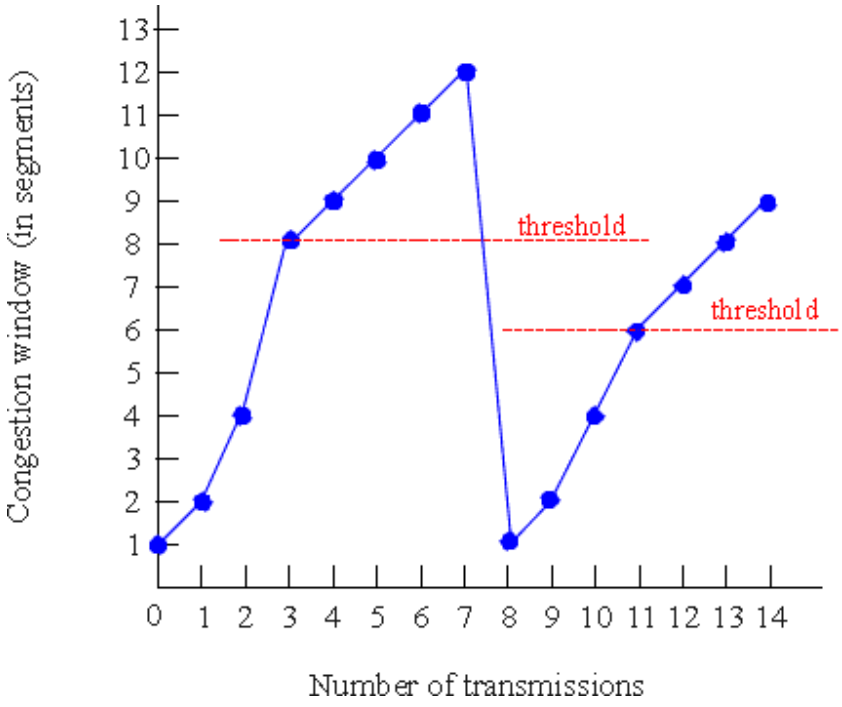
- aumento exponencial (por RTT) no tamanho da janela (não muito lenta!)
- evento de perda: temporizador (Tahoe TCP) e/ou três ACKs duplicados (Reno TCP)



TCP: Evitar Congestionamento

Evitar congestionamento

```
/* partida lenta acabou */
/* Congwin > threshold */
Until (event de perda) {
  cada w segmentos
reconhecidos:
      Congwin++
  }
threshold = Congwin/2
Congwin = 1
faça partida lenta
```



1: TCP Reno pula partida lenta (recuperação rápida) depois de três ACKs duplicados

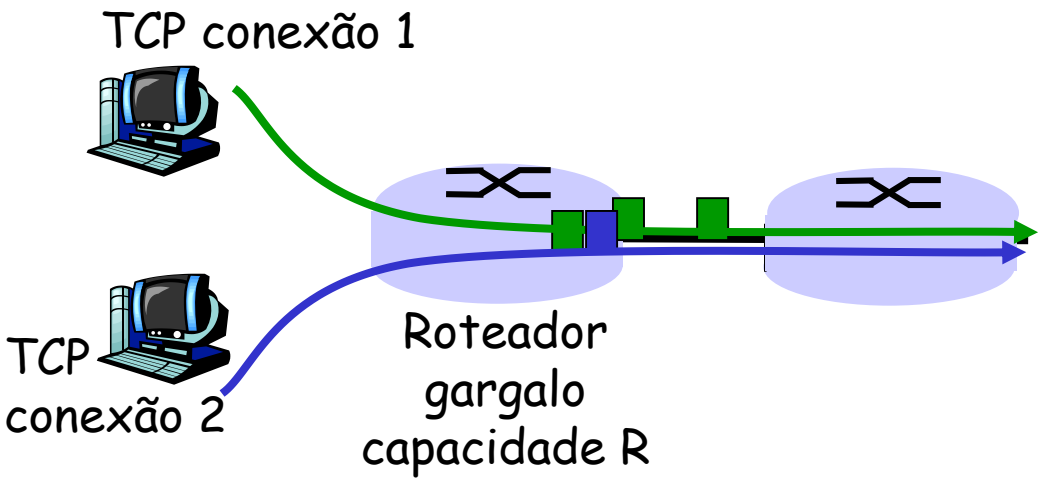
AADM

TCP congestion avoidance:

- AADM: *aumento aditivo, decremento multiplicativo*
  - ✓ aumenta janela em 1 por cada RTT
  - ✓ diminui janela por fator de 2 num evento de perda

Justeza do TCP

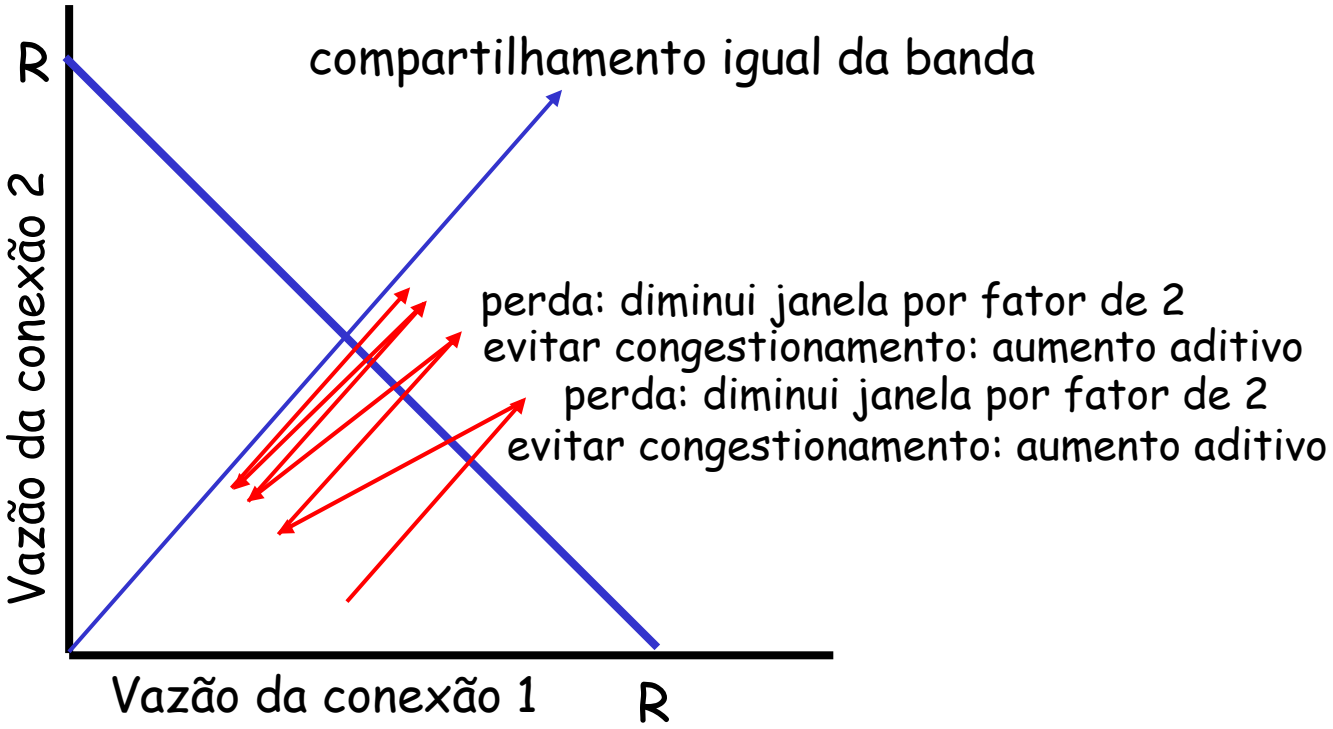
Meta de justeza: se N sessões TCP compartilham o mesmo enlace de gargalo, cada uma deve ganhar 1/N da capacidade do enlace



Por quê TCP é justo?

Duas sessões concorrentes:

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



TCP: modelagem de latência

P: Quanto tempo custa para receber um objeto de um servidor WWW depois de enviar o pedido?

- Estabelecimento de conexão TCP
- retardo de transferência de dados

Notação, suposições:

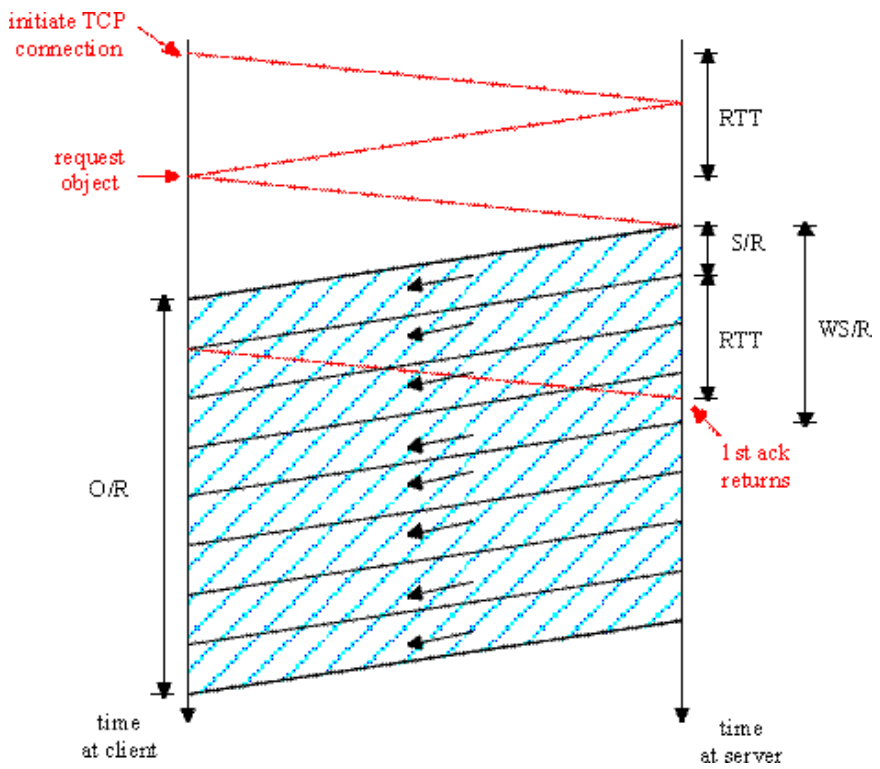
- Supomos um enlace entre cliente e servidor de taxa R
- Supomos: janela de congestionamento fixo, W segmentos
- S: MSS (bits)
- O: tamanho do objeto (bits)
- sem retransmissões (sem perdas, sem erros)

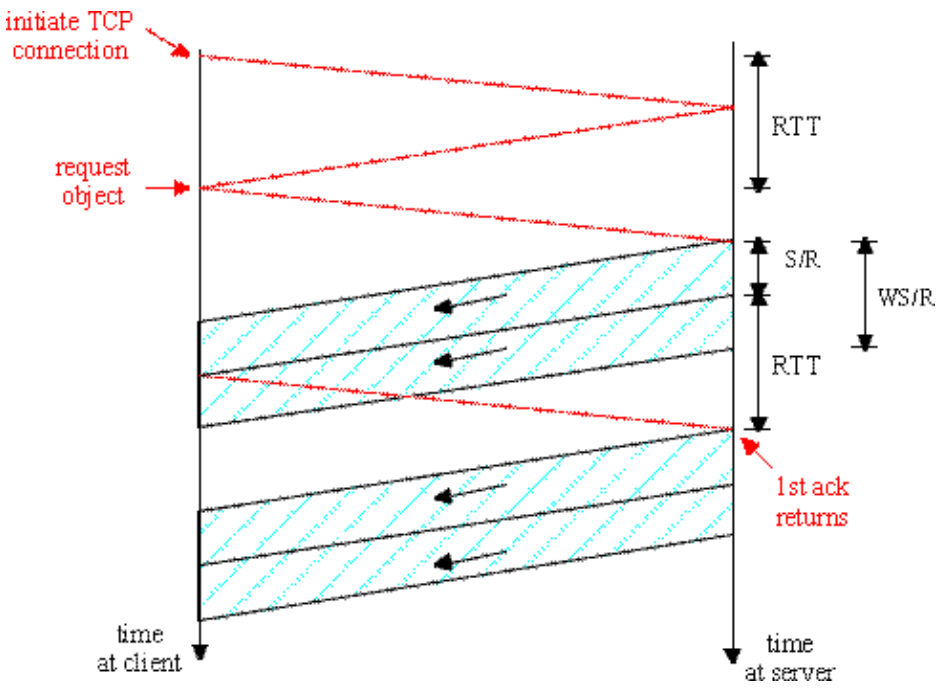
Dois casos a considerar:

- WS/R > RTT + S/R: ACK do primeiro segmento na janela chega antes de enviar todos dados na janela
- WS/R < RTT + S/R: aguarda ACK depois de enviar todos os dados na janela

TCP: modelagem de latência

K:= O/WS





Caso 1: latência = 2RTT + O/R

Caso 2: latência = 2RTT + O/R + (K-1)[S/R + RTT - WS/R]

TCP: modelagem de latência: partida lenta

- Agora supomos que a janela cresce à la partida lenta.
- Mostramos que a latência de um objeto de tamanho O é:

$$Latency = 2RTT + \frac{O}{R} + P\left[RTT + \frac{S}{R}\right] - (2^P - 1)\frac{S}{R}$$

onde *P* é o número de vezes TCP para no servidor:

$$P = \min\{Q, K-1\}$$

- onde Q é o número de vezes que o servidor pararia se o objeto fosse de tamanho infinito.

- e K é o número de janelas que cobrem o objeto.

TCP: modelagem de latência: partida lenta (cont.)

Exemplo:

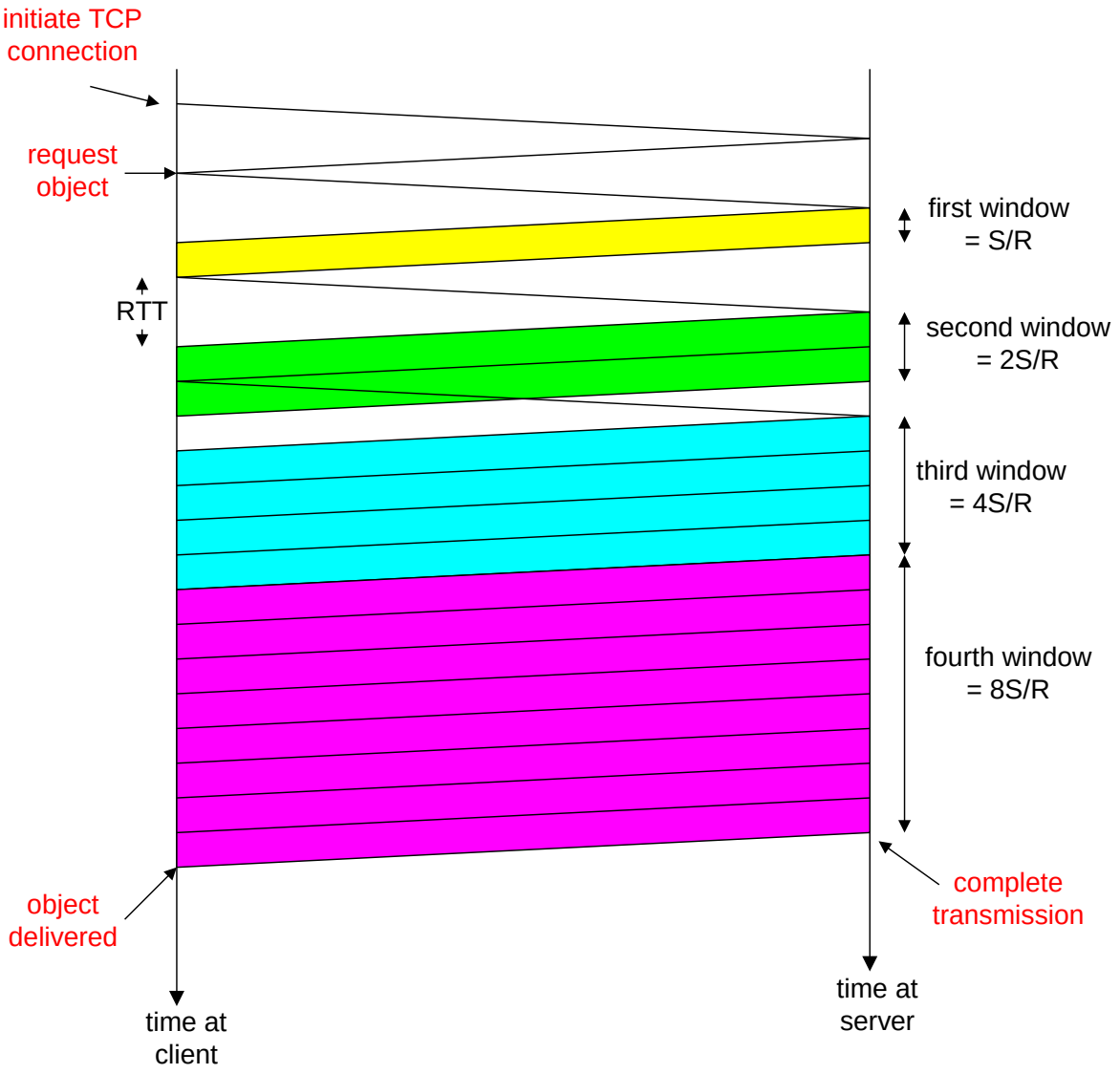
O/S = 15 segmentos

K = 4 janelas

Q = 2

P = mín{K-1,Q} = 2

Servidor para P=2 vezes.



TCP: modelagem de latência: partida lenta (cont.)

$$\frac{S}{R} + RTT = \text{time from when server starts to send segment}$$

$$\text{until server receives acknowledgement}$$

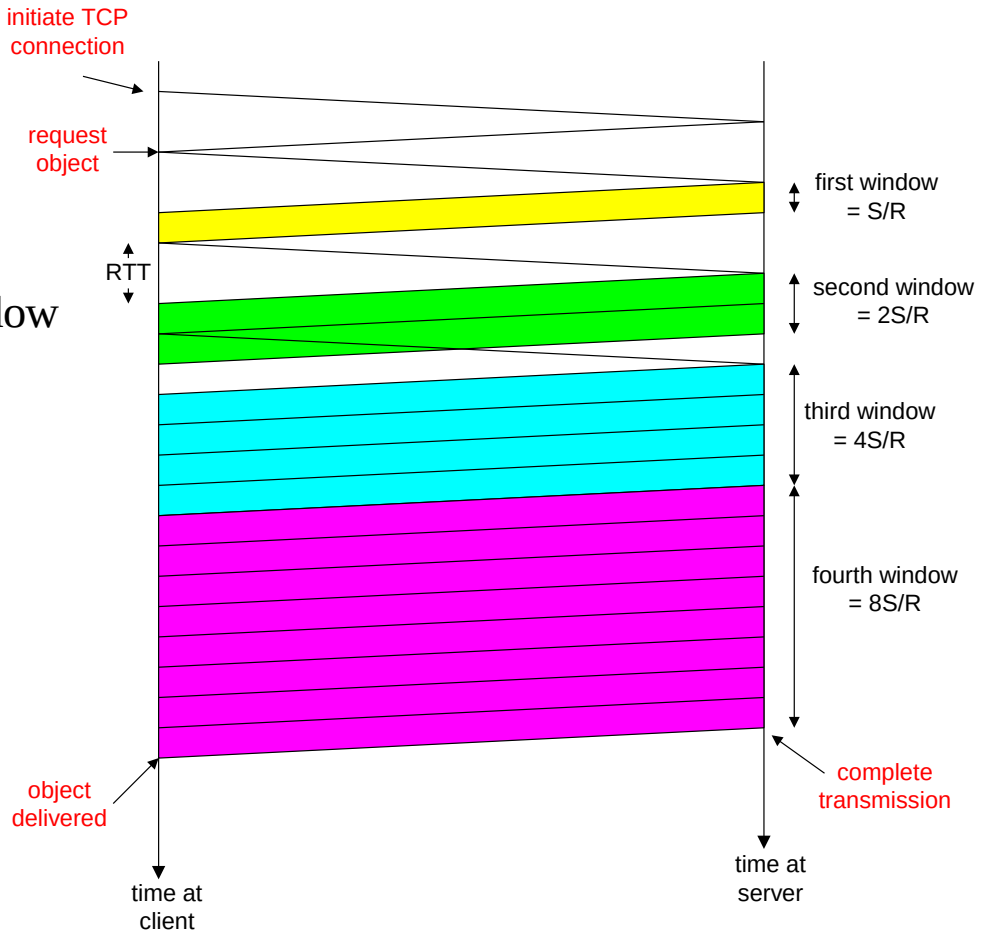
$$2^{k-1}\frac{S}{R} = \text{time to transmit the kth window}$$

$$\left[\frac{S}{R} + RTT - 2^{k-1}\frac{S}{R}\right]^{+} = \text{stall time after the } k\text{th window}$$

$$\text{latency} = \frac{O}{R} + 2RTT + \sum_{p=1}^{P} stallTime_p$$

$$= \frac{O}{R} + 2RTT + \sum_{k=1}^{P}\left[\frac{S}{R} + RTT - 2^{k-1}\frac{S}{R}\right]$$

$$= \frac{O}{R} + 2RTT + P\left[RTT + \frac{S}{R}\right] - (2^{P} - 1)\frac{S}{R}$$



Capítulo 3: Resumo

- Princípios atrás 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