Infra-Estrutura de Comunicação (IF678)

Módulo III

Fonte: kurose

Adaptações : Prof. Paulo Gonçalves
pasg@cin.ufpe.br
CIn/UFPE

Módulo 3: Camada Transporte

Nossos objetivos:

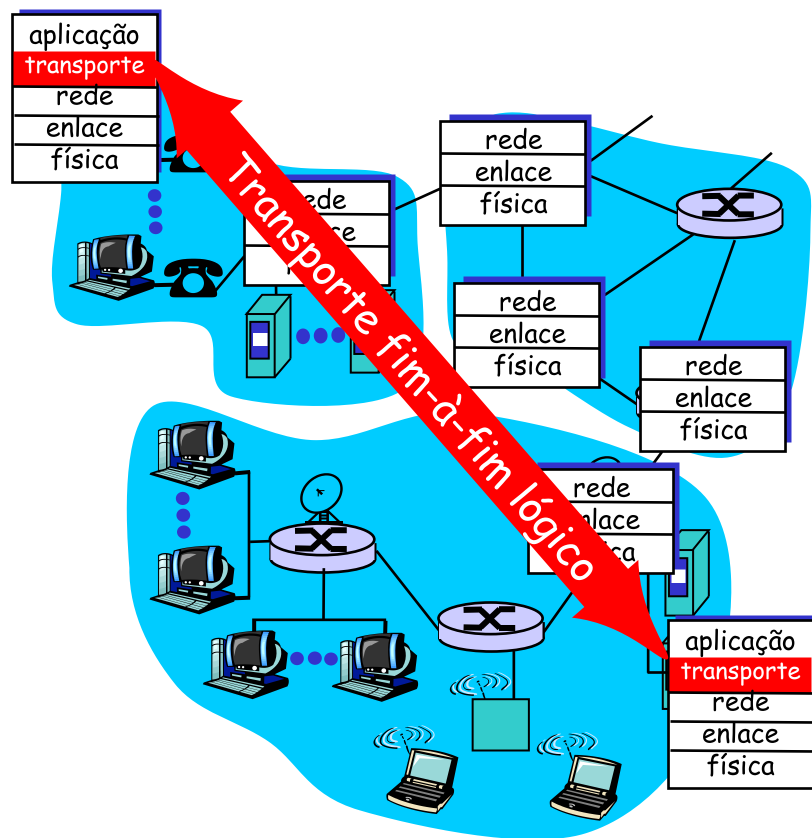
- Compreender os princípios por trás dos serviços de transporte:
  - multiplexação/demultiplexação
  - Transferência confiável de dados
  - Controle de fluxo
  - Controle de congestionamento
- Aprender sobre os protocolos da camada transporte na Internet:
  - UDP: transporte não-orientado à conexão
  - TCP: transporte orientado à conexão
  - Controle de congestionamento do TCP

Resumo do Módulo 3

- 3.1 Serviços da camada transporte
- 3.2 Multiplexação e demultiplexão
- 3.3 transporte não-orientado à conexão: UDP
- 3.4 Princípios da transferência confiável de dados
- 3.5 transporte orientado à conexão: TCP
  - Estrutura do segmento
  - Transferência de dados confiável
  - Controle de fluxo
  - Gerenciamento de conexão
- 3.6 Princípios do controle de congestionamento
- 3.7 controle de congestionamento TCP

Serviços e Protocolos de Transporte

- provêem *comunicação lógica* entre processos aplicativos executando em diferente hosts
- Protocolos de transporte "rodam" em end systems
  - Lado emissor: quebra mensagens da aplicação em segmentos que são passados à camada de rede
  - Lado receptor: remonta segmentos em mensagens e os passa à camada aplicação
- mais de um protocolo de transporte disponível para as aplicações
  - Internet: TCP e UDP



Camada Transporte vs. Camada de Rede

- *Camada de rede:* comunicação lógica entre hosts
- *Camada transporte:* comunicação lógica entre processos
  - Conta com (e melhora) serviços da camada de rede

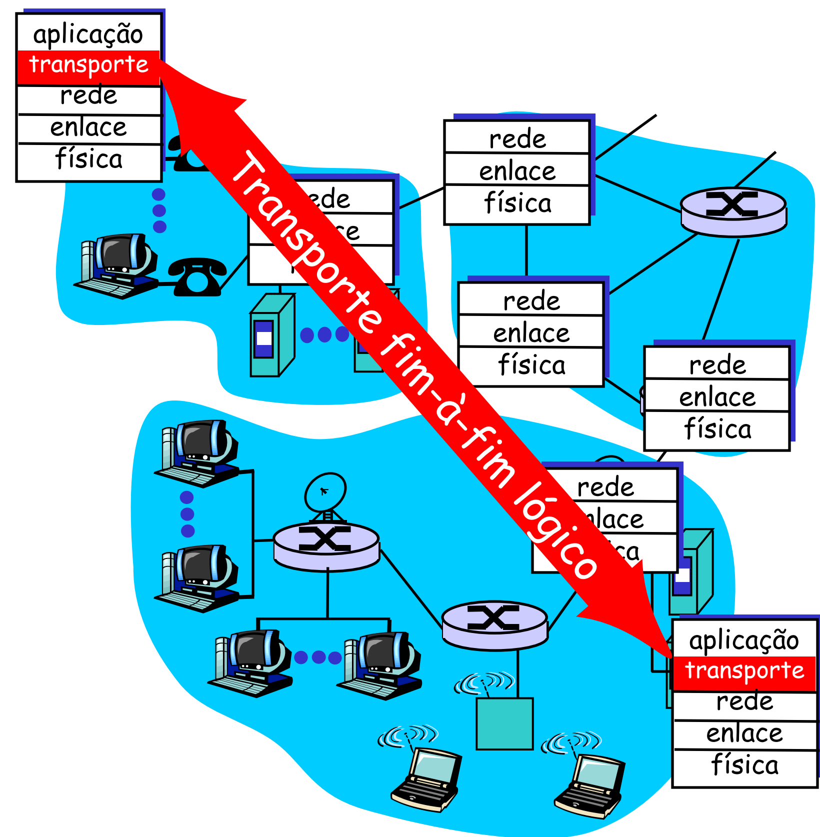
analogia:

12 crianças enviando cartas à 12 crianças

- processos = crianças
- mensagens da aplic.= cartas em envelopes
- hosts = casas
- Protocolo de transporte = *Ana* e Bill
- Protocolo da camada de rede = serviço postal

Protocolos da camada transporte da Internet

- Entrega confiável, em ordem (TCP)
  - Controle de congestionamento
  - Controle de fluxo
  - Estabelecimento de conexão
- Entrega não-confiável, sem garantias de ordenação: UDP
  - Extensões “sem ornamentos” ao serviço de melhor esforço (best-effort) IP
- serviços indisponíveis:
  - Garantias de atraso
  - Garantias de banda passante



Resumo do Módulo 3

- 3.1 Serviços da camada transporte
- 3.2 Multiplexação e demultiplexão
- 3.3 transporte não-orientado à conexão: UDP
- 3.4 Princípios da transferência confiável de dados
- 3.5 transporte orientado à conexão: TCP
  - Estrutura do segmento
  - Transferência de dados confiável
  - Controle de fluxo
  - Gerenciamento de conexão
- 3.6 Princípios do controle de congestionamento
- 3.7 controle de congestionamento TCP

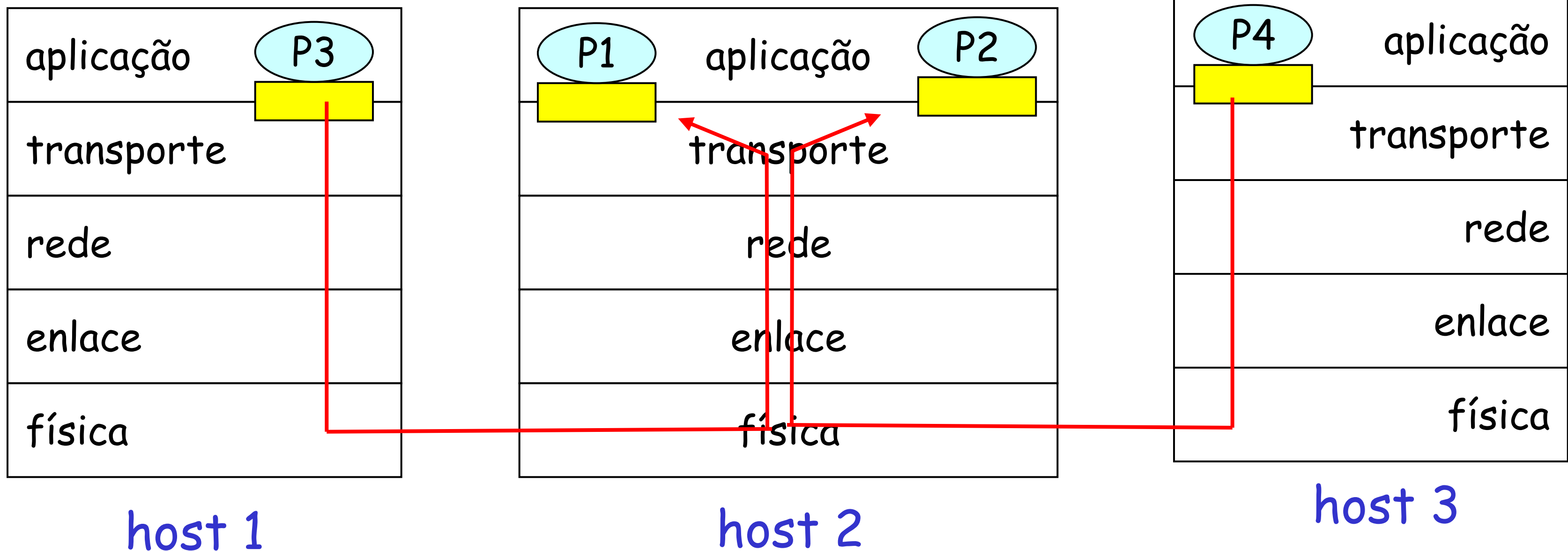
Multiplexação/demultiplexação

Demultiplexação no host receptor:

Entrega dos segmentos recebidos aos sockets corretos

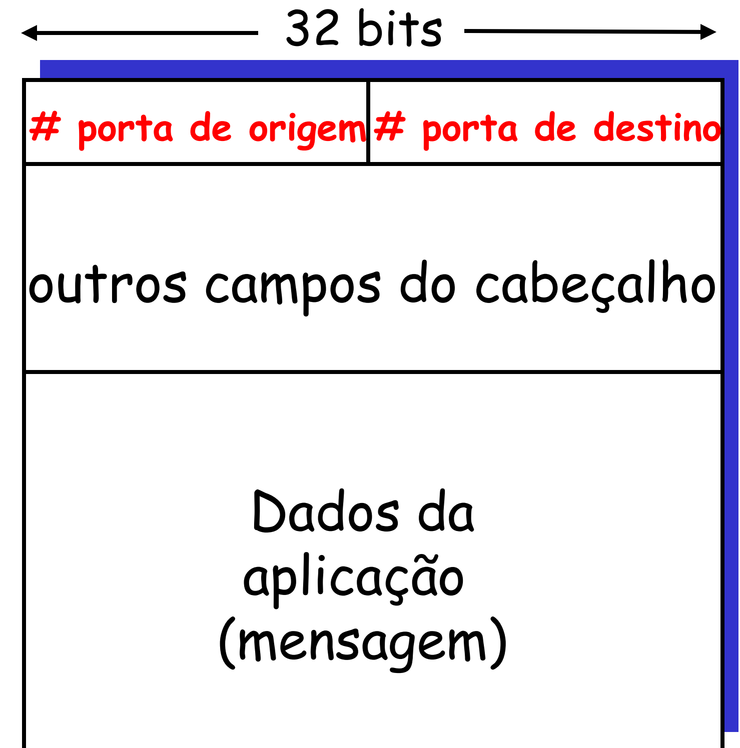
Multiplexação no host emissor:

Coletar dados dos vários sockets, adiciona cabeçalho aos dados (mais tarde usado para demultiplexação)



Como a demultiplexação funciona

- host recebe datagramas IP
  - cada datagrama possui endereço IP fonte, endereço IP de destino
  - cada datagrama carrega 1 segmento da camada transporte
  - cada segmento possui número de porta de origem e de porta de destino
- host usa os endereços IP & número das portas para enviar segmento ao socket adequado



Formato do segmento TCP/UDP

demultiplexação com UDP

- Criar sockets com portas respectivas:

```
DatagramSocket mySocket1 = new
   DatagramSocket(99111);
DatagramSocket mySocket2 = new
   DatagramSocket(99222);
```

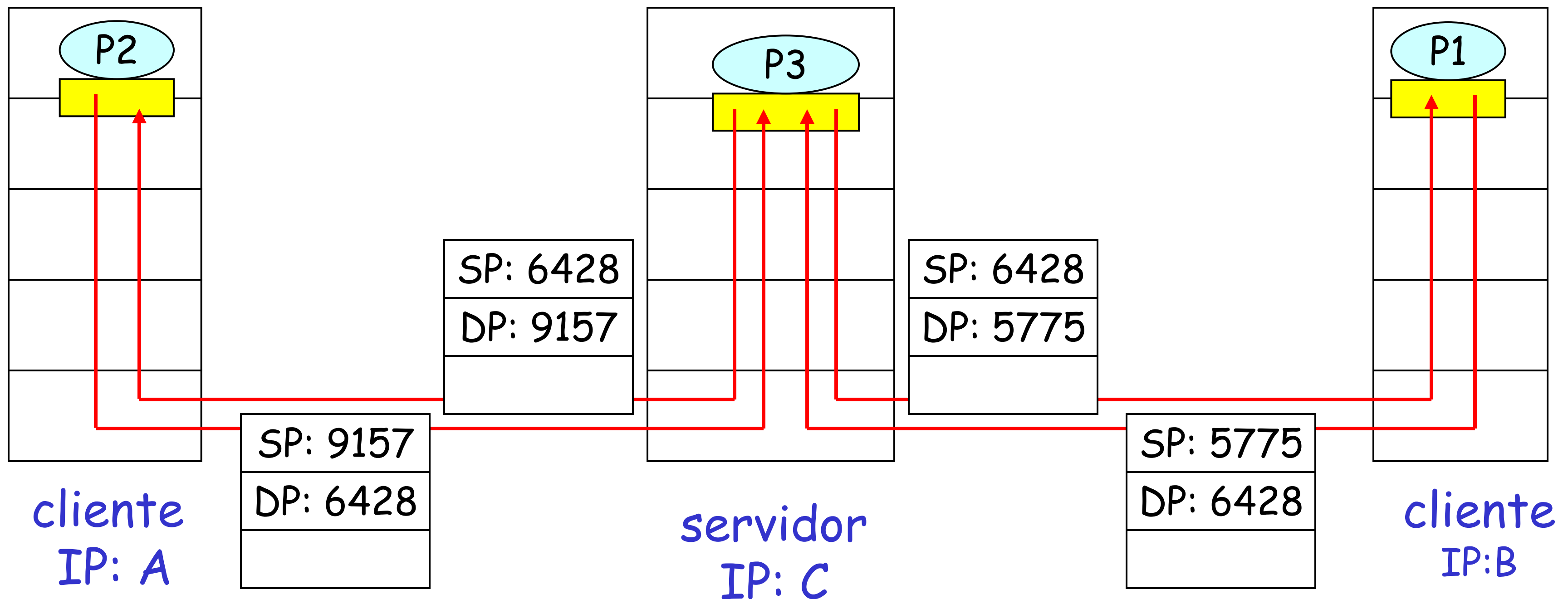
- Socket UDP identificado pela tupla:

(endereço IP de destino, número da porta de destino)

- Quando um host recebe um segmento UDP:
  - Verifica o número da porta de destino no segmento
  - direciona o segmento UDP para o socket com o número da porta especificado
- Datagramas IP com endereço IP fonte diferentes e/ou números de porta de origem diferentes são direcionados ao mesmo socket

demux com UDP (cont)

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



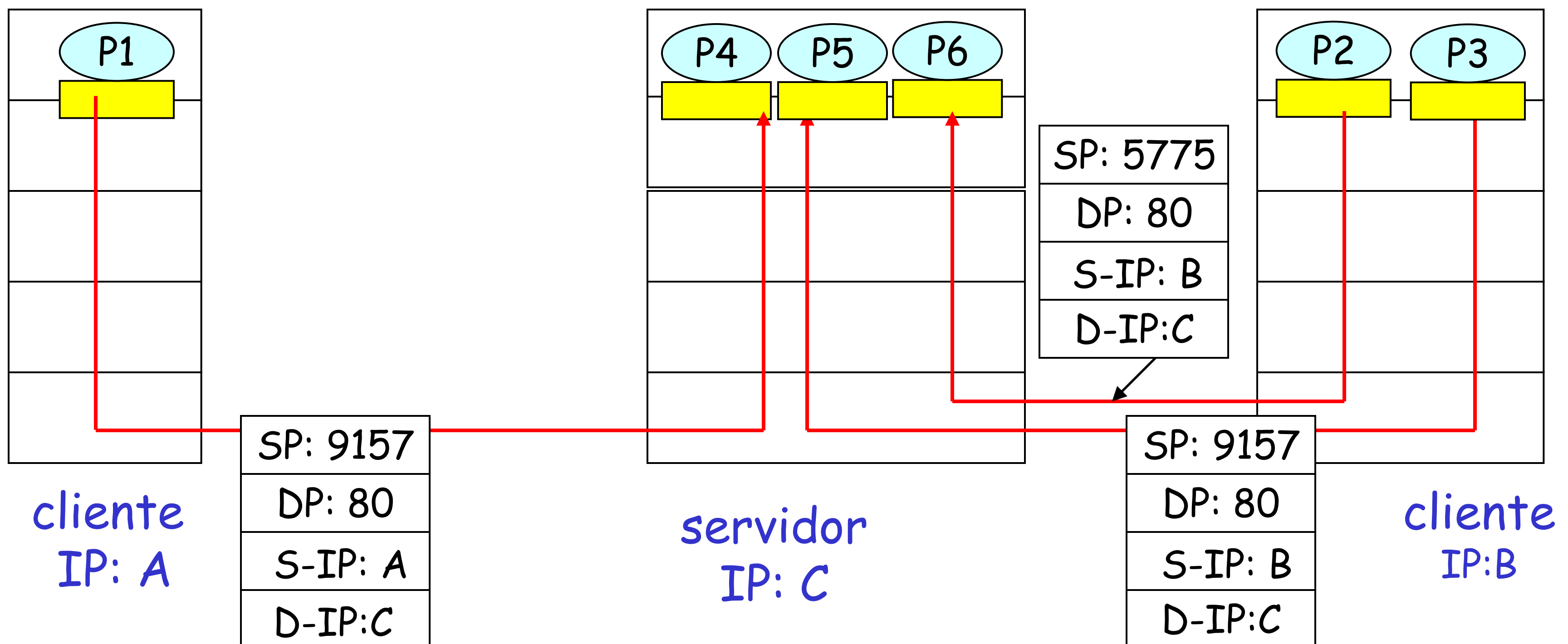
SP provê uma espécie de "endereço de retorno"

SP : source port (porta de origem) DP: destination port (porta de destino)

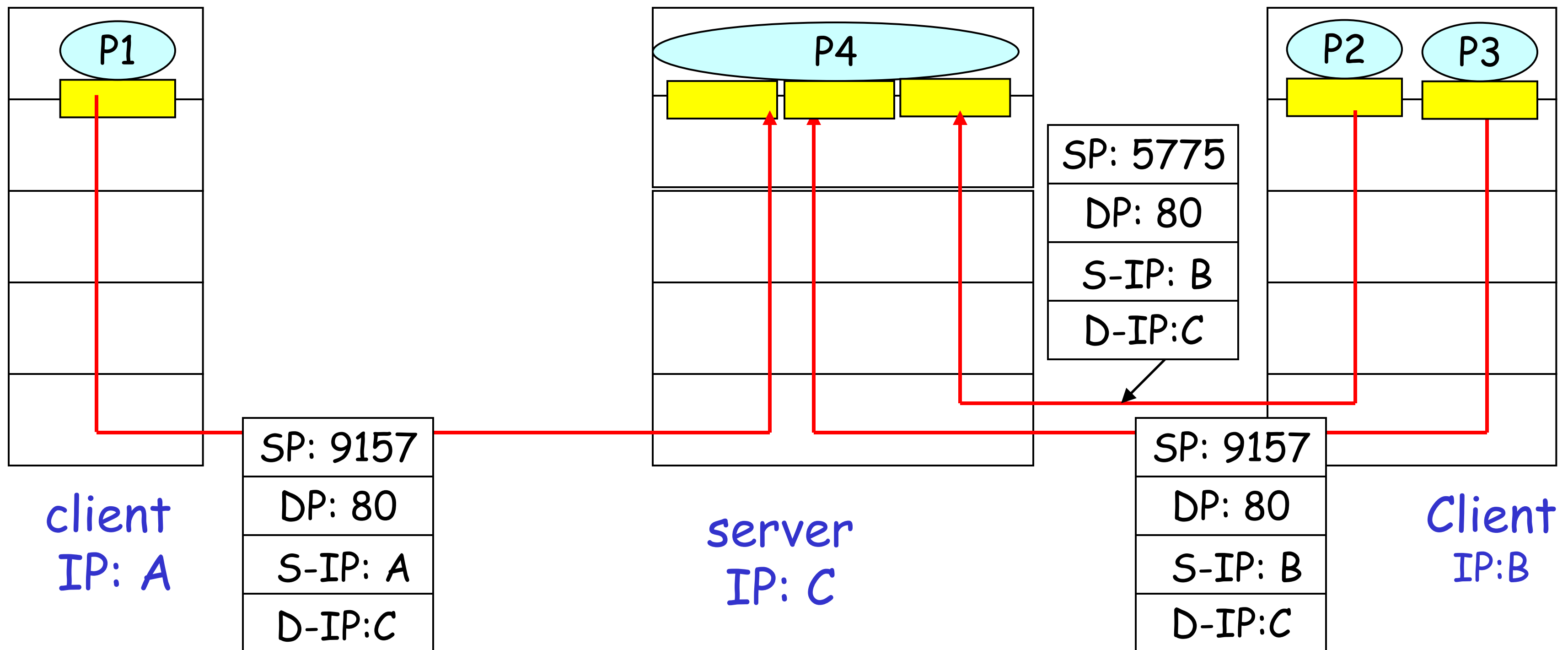
Demux com TCP

- Socket TCP identificado pela tupla quádrupla:
  - Endereço IP de origem
  - Número da porta de origem
  - Endereço IP de destino
  - Número da porta de destino
- Host receptor usa todos esses quatro valores para enviar o segmento ao socket apropriado
- host servidor pode suportar diversos sockets TCP simultaneamente:
  - cada socket é identificado por sua proópria tupla quádrupla
- Servidores Web possuem sockets diferentes para cada cliente conectado
  - HTTP não-persistente terá diferentes sockets para cada requisição

demux (cont) com TCP



Demux com TCP: Web Server com Threads



Resumo do Módulo 3

- 3.1 Serviços da camada transporte
- 3.2 Multiplexação e demultiplexão
- 3.3 transporte não-orientado à conexão: UDP
- 3.4 Princípios da transferência confiável de dados
- 3.5 transporte orientado à conexão: TCP
  - Estrutura do segmento
  - Transferência de dados confiável
  - Controle de fluxo
  - Gerenciamento de conexão
- 3.6 Princípios do controle de congestionamento
- 3.7 controle de congestionamento TCP

UDP: User Datagram Protocol [RFC 768]

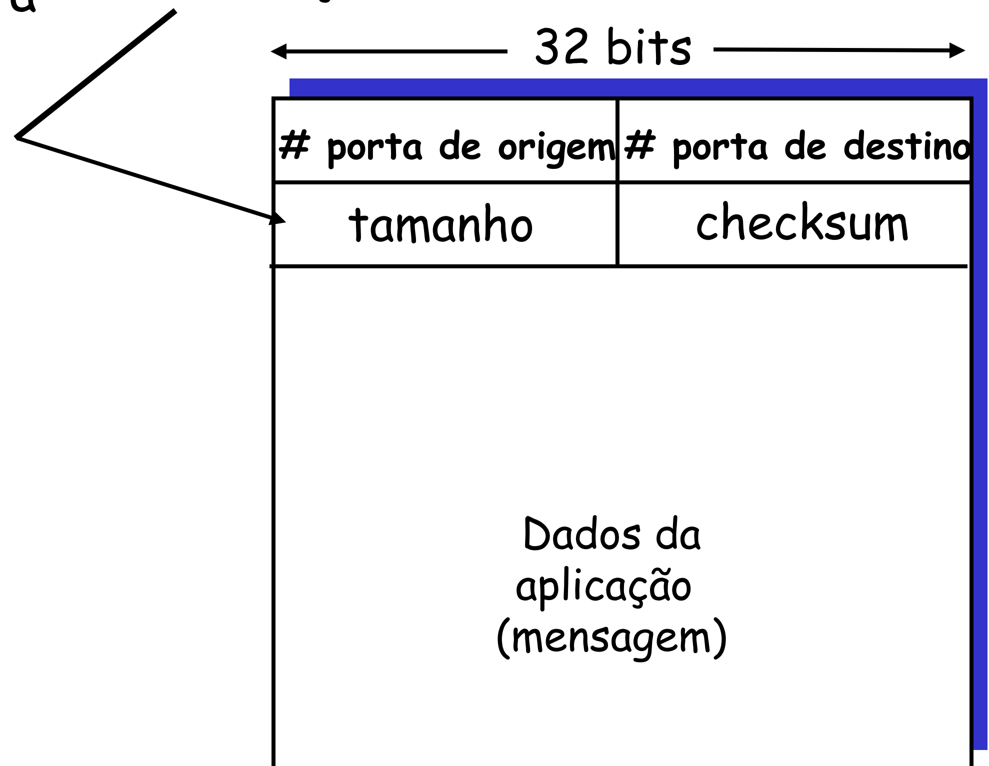
- Protocolo Internet de transporte "sem ornamentos" e com "elementos básicos"
- Serviço "best effort", segmentos UDP podem ser:
  - perdidos
  - entregues fora de ordem à aplicação
- *não-orientado à conexão:*
  - sem handshaking entre o emissor e receptor UDP
  - Cada segmento UDP é tratado de forma independente dos outros

Por que existe o UDP?

- Sem estabelecimento de conexão (que pode adicionar atraso)
- simples: sem estado de conexão no emissor nem no receptor
- Cabeçalho do segmento pequeno
- nenhum controle de congestionamento

UDP: mais ...

- Freqüentemente usado para aplicações multimídia de streaming
  - Tolerante à perdas
  - Sensível à taxa de dados
- Outros usos do UDP
  - DNS
  - SNMP
- Transferência confiável sobre UDP: confiabilidade adicionada na camada aplicação
  - Recuperação de erros específica da camada aplicação!



Formato do segmento UDP

Checksum UDP

Objetivo: detectar "erros" (e.g., bits trocados) no segmento transmitido

Emissor:
- Trata o conteúdo de segmentos como uma seqüência de inteiros de 16 bits
- checksum: adição (soma complemento 1) do conteúdo do segmento
- Emissor coloca o valor do checksum no campo checksum do UDP

Receptor:
- computa o checksum do segmento recebido
- Verifica se o checksum computado bate com o valor informado no campo checksum:
  - Não - erro detectado
  - Sim - nenhum erro detectado. *Mas pode haver erros?* Mais em breve ...

Exemplo de Checksum Internet

- Nota
  - Ao adicionar números, não esquecer do "vai um"
- Examplo: adição de dois inteiros de 16 bits

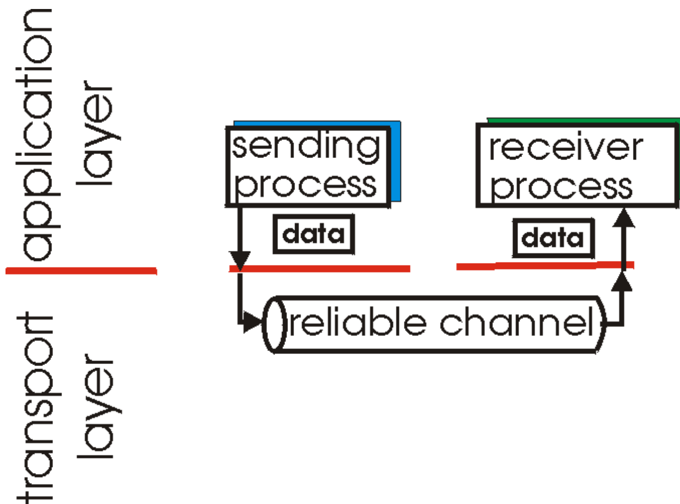
		1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
		1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
soma		1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum		0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

Resumo do Módulo 3

- 3.1 Serviços da camada transporte
- 3.2 Multiplexação e demultiplexão
- 3.3 transporte não-orientado à conexão: UDP
- 3.4 Princípios da transferência confiável de dados
- 3.5 transporte orientado à conexão: TCP
  - Estrutura do segmento
  - Transferência de dados confiável
  - Controle de fluxo
  - Gerenciamento de conexão
- 3.6 Princípios do controle de congestionamento
- 3.7 controle de congestionamento TCP

Princípios da transferência confiável de dados

- importante nas camadas aplicação, transporte e enlace
- Está na lista dos 10 tópicos mais importantes em redes!

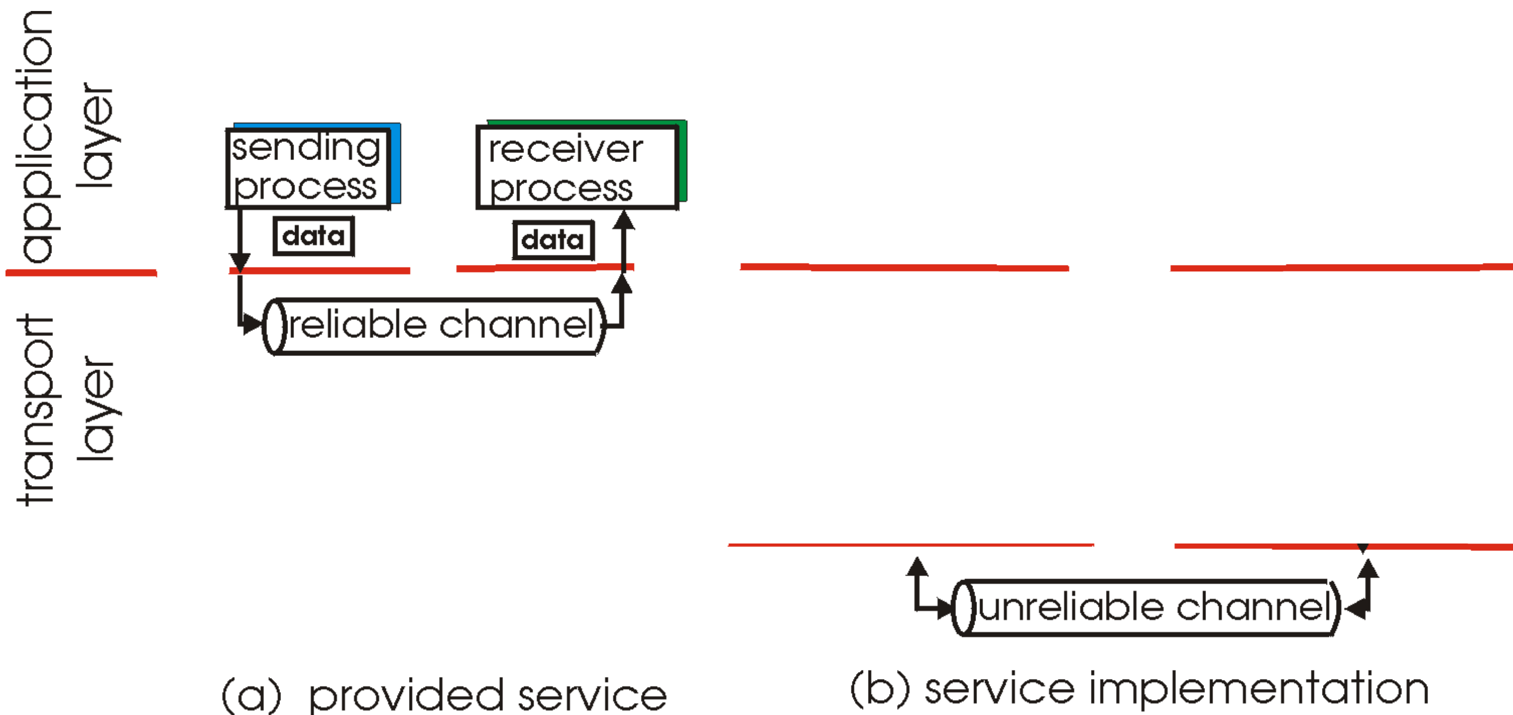


(a) provided service

- características do canal não confiável determinará a complexidade do protocolo de transferência confiável (rdt)

Princípios da transferência confiável de dados

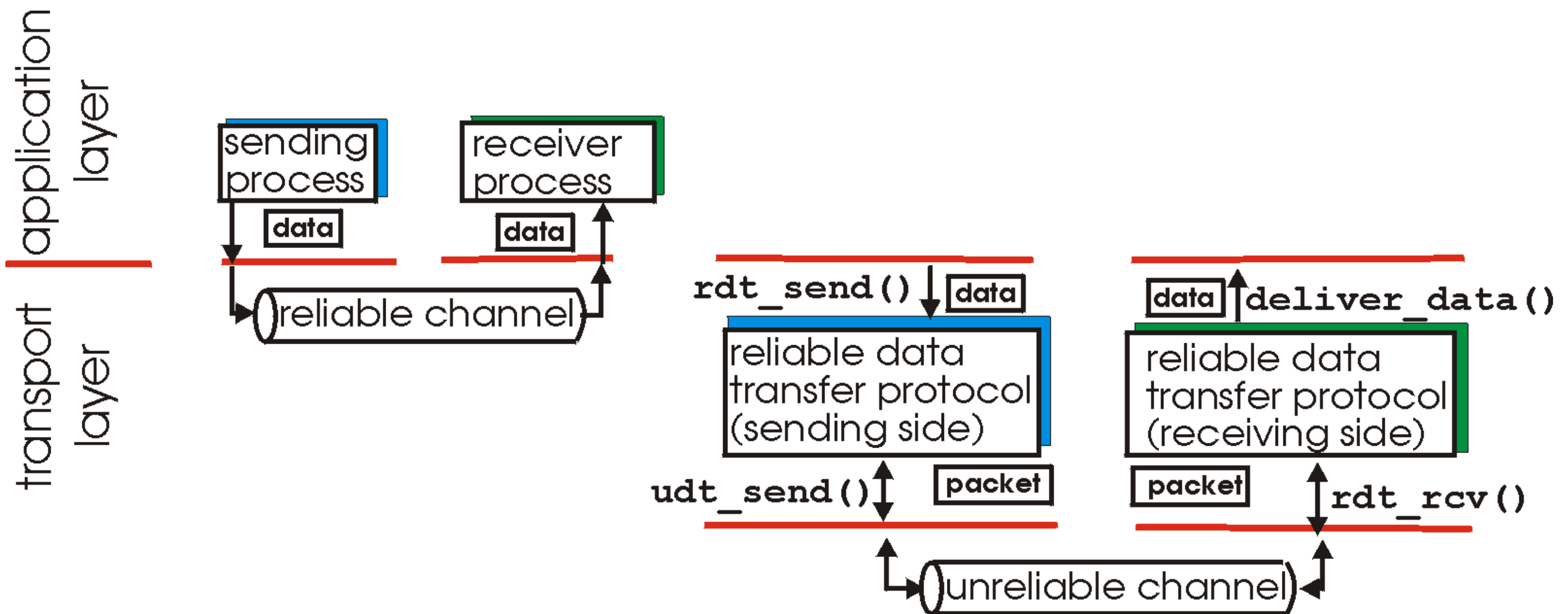
- importante nas camadas aplicação, transporte e enlace
- Está na lista dos 10 tópicos mais importantes em redes!



- características do canal não confiável determinará a complexidade do protocolo de transferência confiável (rdt)

Princípios da transferência confiável de dados

- importante nas camadas aplicação, transporte e enlace
- Está na lista dos 10 tópicos mais importantes em redes!

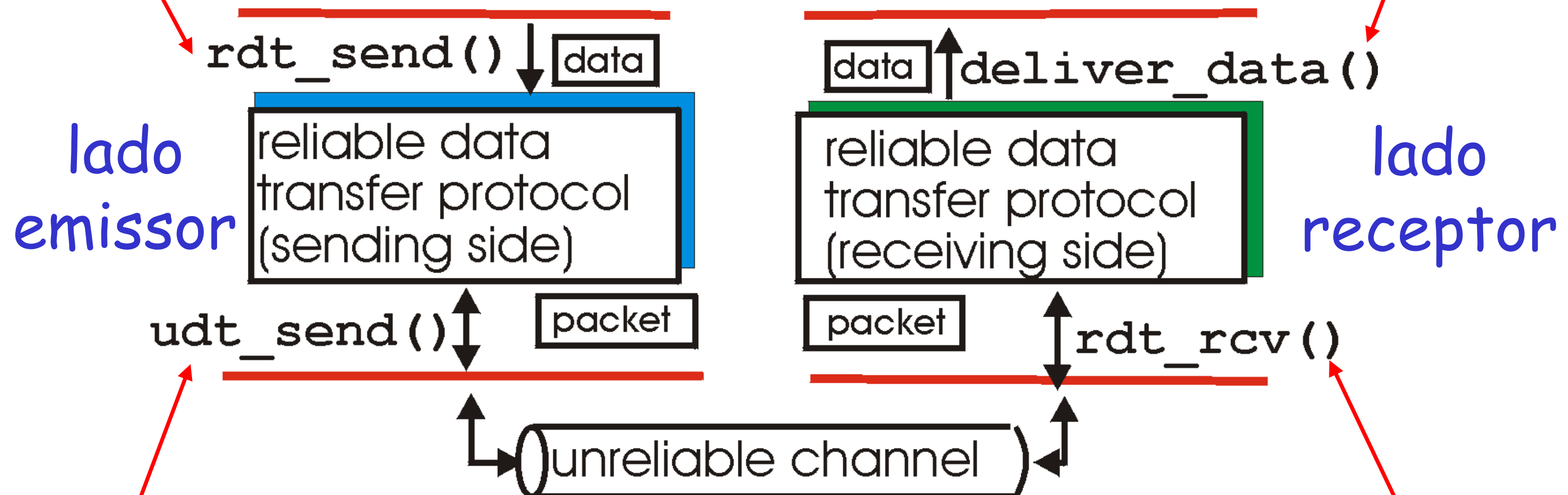


- características do canal não confiável determinará a complexidade do protocolo de transferência confiável (rdt)

Transferência confiável de dados: introdução ...

rdt_send() : chamado pela camada superior, (e.g., aplic.). Passagem de dados da camada superior

deliver_data() : chamado pelo **rdt** para enviar dados à camada superior



udt_send() : chamado por rdt para transferir pacote pelo canal não-confiável ao receptor

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

Transferência confiável de dados: introdução ...

Iremos:

- Desenvolver incrementalmente os lados receptor e emissor do protocolo de transferência confiável de dados (rdt)
- considere somente transferência de dados unidirecional
  - mas informações de controle transitarão em ambos sentidos!
- usar máquinas de estado finitas (FSM) para especificar o emissor e o receptor

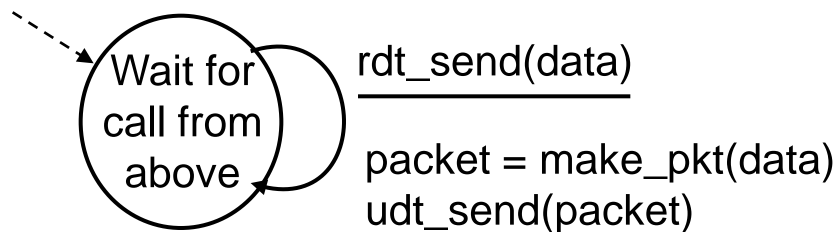
Evento causando transição de estado

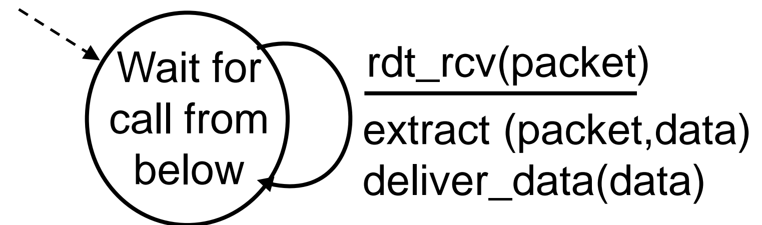
Ações tomadas na transição de estados

estado: quando estiver neste "estado", próximo estado é unicamente determinado pelo próximo evento

Rdt1.0: transferência confiável sobre um canal confiável

- Canal de base perfeitamente confiável
  - Nenhum erro nos bits
  - Nenhuma perda de pacotes
- FSMs separadas para emissor e receptor:
  - Emissor envia dados no canal
  - Receptor lê dados do canal





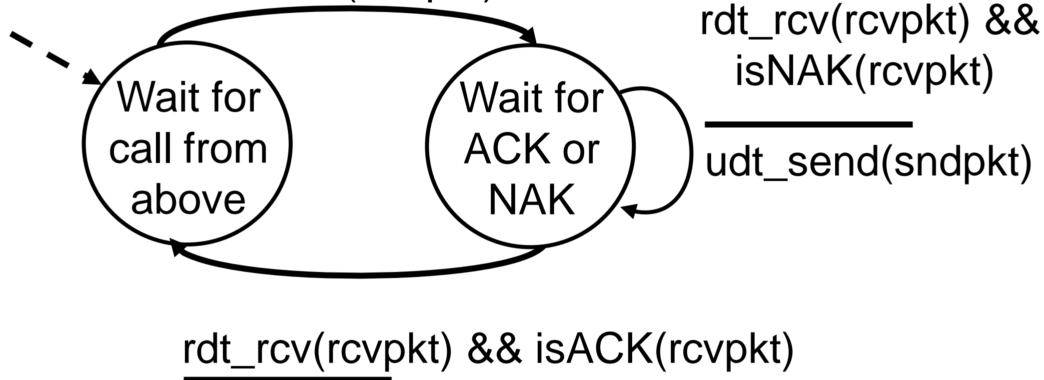
emissor

receptor

Rdt2.0: canal com erros em bits

- Canal de base pode trocar bits dos pacotes
  - checksum para detectar erros nos bits
- *a* questão: como se recuperar de erros:
  - *acknowledgements (ACKs):* receptor informa explicitamente ao emissor que pacote recebido está OK
  - *negative acknowledgements (NAKs):* receptor informa explicitamente ao emissor que o pacote possui erros
  - Emissor retransmite pacote ao receber um NAK
- novos mecanismos no **`rdt2.0`** (além de **`rdt1.0`**):
  - Detecção de erro
  - Feedback do receptor: msgs de controle (ACK,NAK) do recptor para o emissor

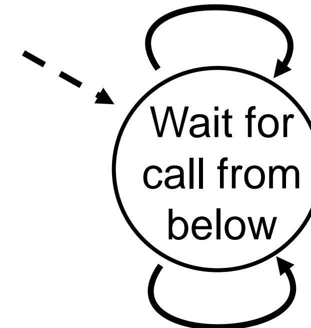
rdt2.0: especificação FSM



emissor

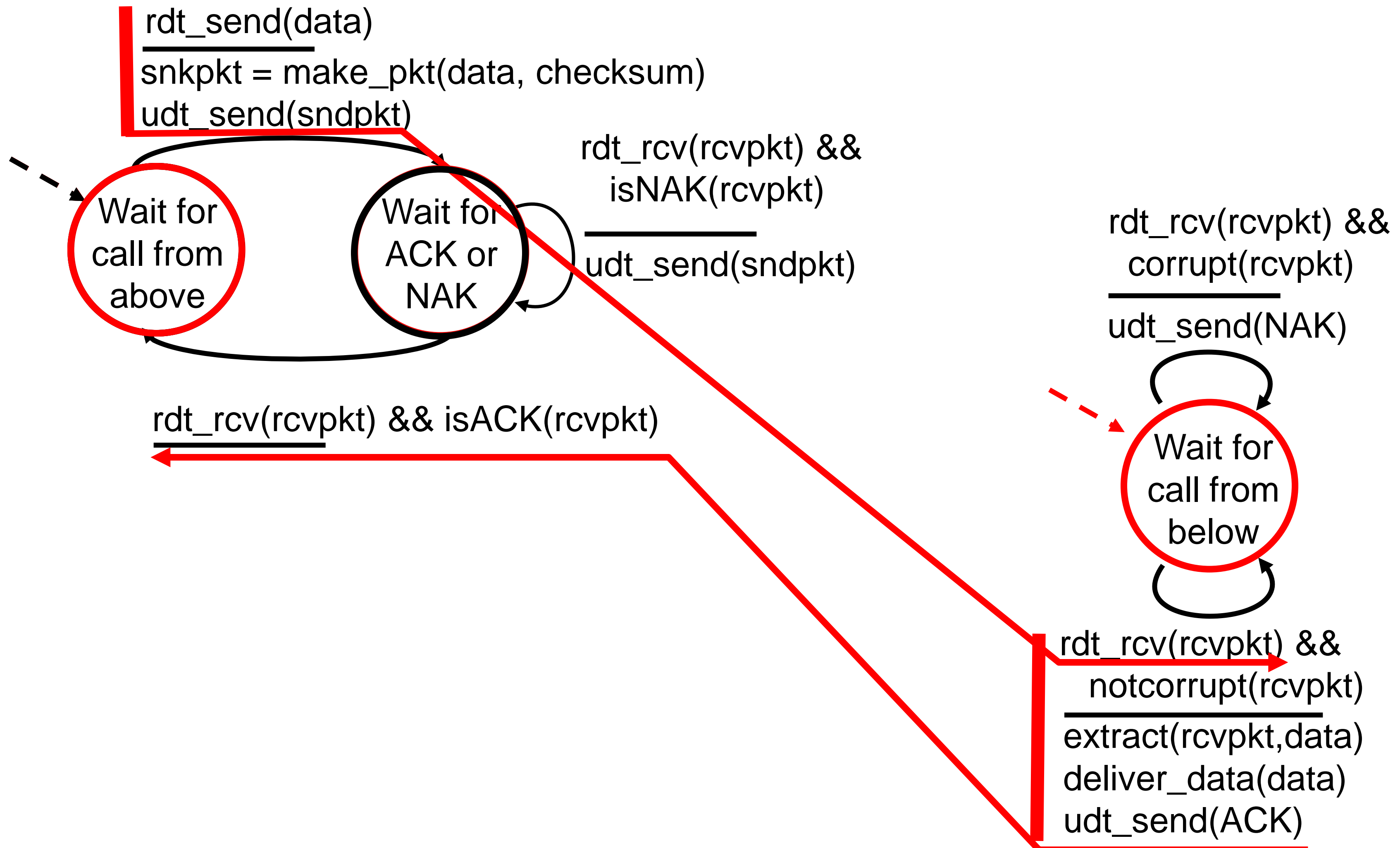
receptor

rdt_rcv(rcvpkt) && corrupt(rcvpkt)
udt_send(NAK)

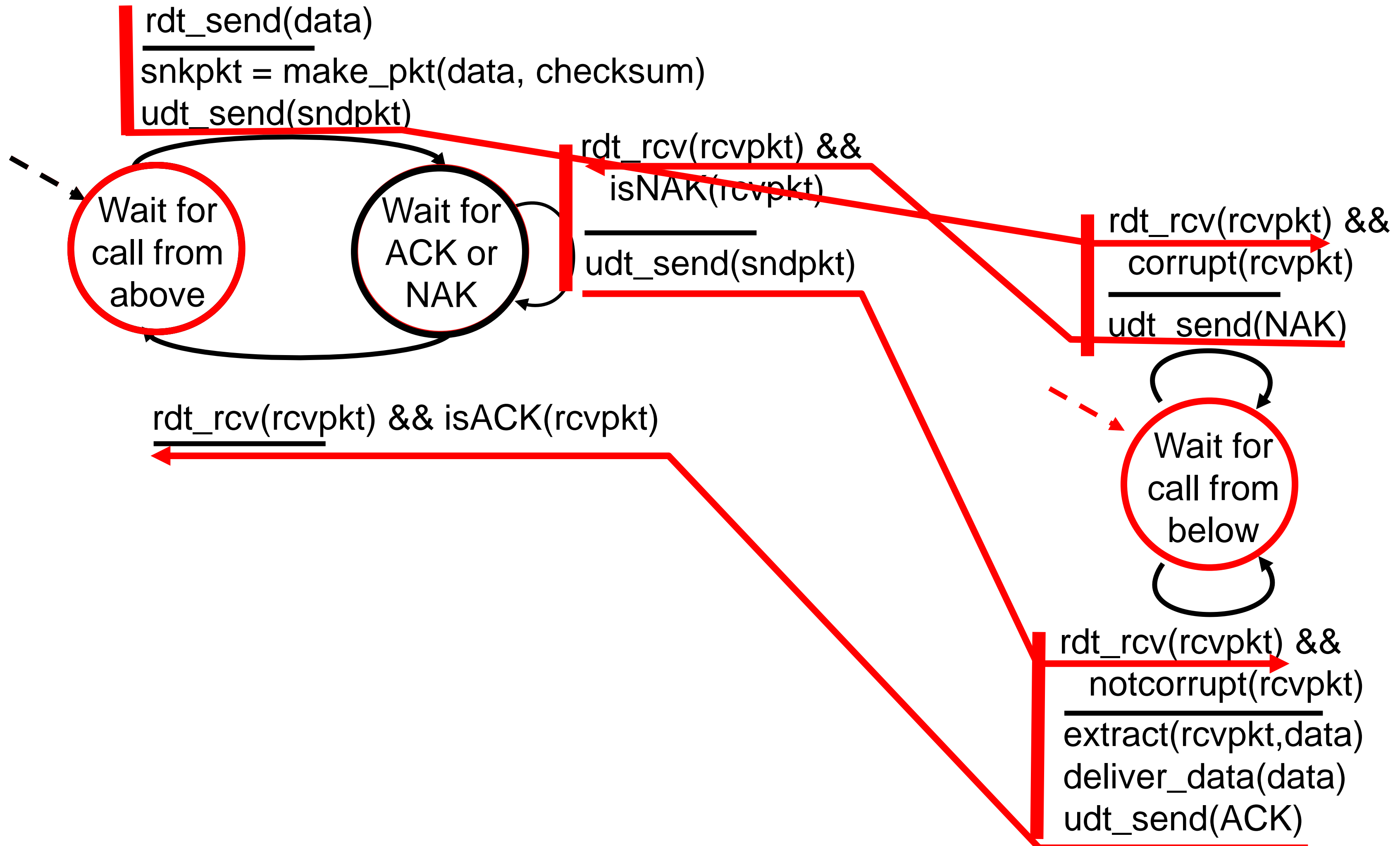


rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
udt_send(ACK)

rdt2.0: operação na ausência de erros



rdt2.0: cenário na presença de erros



rdt2.0 tem um problema fatal!

O que acontece se um ACK/NAK é corrompido?

- emissor não sabe o que aconteceu no receptor!
- Não pode somente retransmitir: possibilidade de duplicação de pacotes

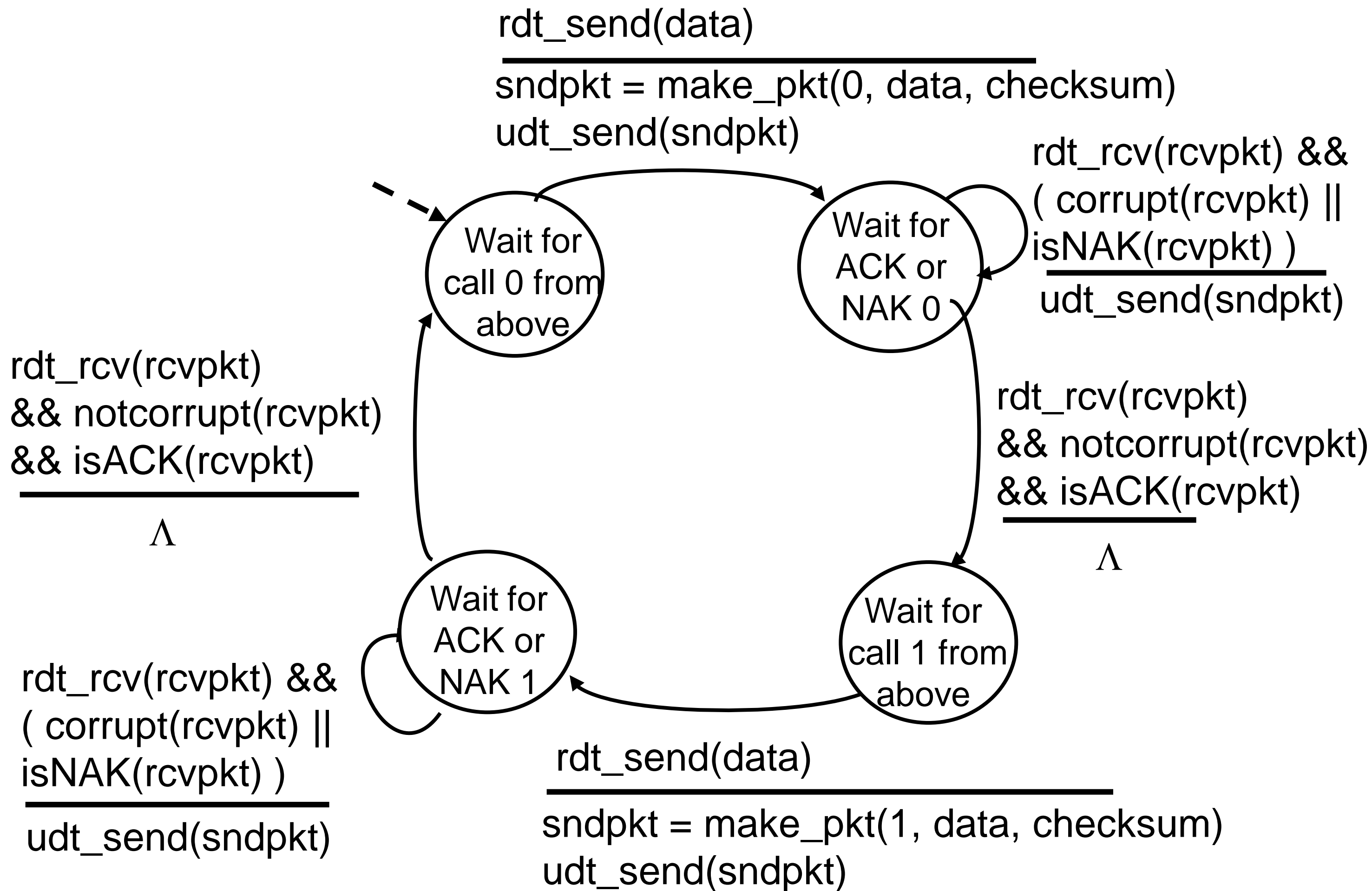
Tratando duplicações:

- Emissor retransmitepacote atual se ACK/NAK é corrompido
- Emissor adiciona *números de seqüência* a cada pacote
- Receptor descarta (não entrega para a camada superior) pacotes duplicados

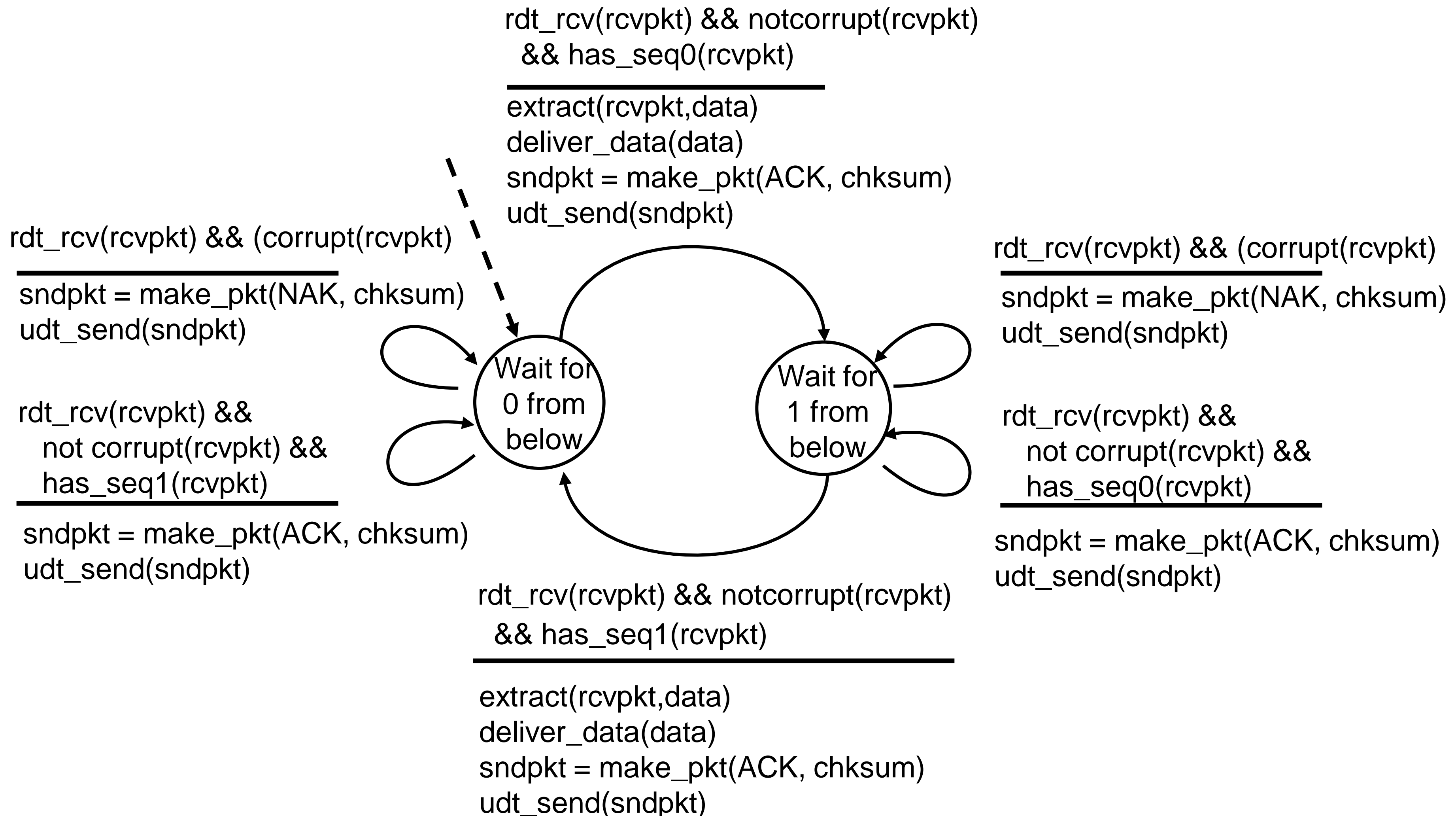
stop and wait

Emissor envia 1 pacote e então aguarda pela Resposta do receptor

rdt2.1: emissor trata ACK/NAKs corrompidos



rdt2.1: receptor trata ACK/NAKs corrompidos



rdt2.1: discussão

Emissor:

- # de seq. adicionado a pkt
- 2 #'s de seq. (0,1) são suficientes. Por que?
- Deve verificar se ACK/NAK recebido está corrompido
- 2x mais estados
  - state deve "recordar" se pacote "atual" possui # de seq. 0 ou 1

Receptor:

- Deve verificar se pacote recebido é um duplicado
  - estado indica se # de seq. do pacote esperado é 0 ou 1
- nota: receptor não pode saber se seu último ACK/NAK foi recebido OK no emissor

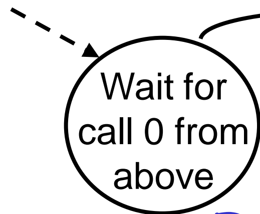
rdt2.2: um protocolo sem NAKs

- Igual ao rdt2.1 mas usando ACKs somente
- Ao invés de NAK, receptor envia ACK do último pacote recebido corretamente (OK)
  - Receptor deve incluir explicitamente o # de seq. do pacote sendo confirmado
- ACK duplicado no emissor resulta na mesma ação como para o NAK: *retransmição do pacote atual*

rdt2.2: fragmentos do emissor e receptor

rdt_send(data)

sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)



Wait for
ACK
0

rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isACK(rcvpkt,1))

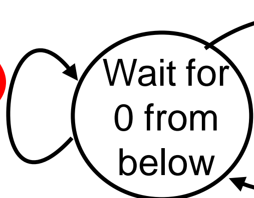
udt_send(sndpkt)

Fragmento da FSM do emissor

rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& **isACK(rcvpkt,0)**

rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
has_seq1(rcvpkt))

udt_send(sndpkt)



fragmento da FSM Do receptor

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq1(rcvpkt)

extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK1, chksum)
udt_send(sndpkt)

rdt3.0: canais com erros e perdas

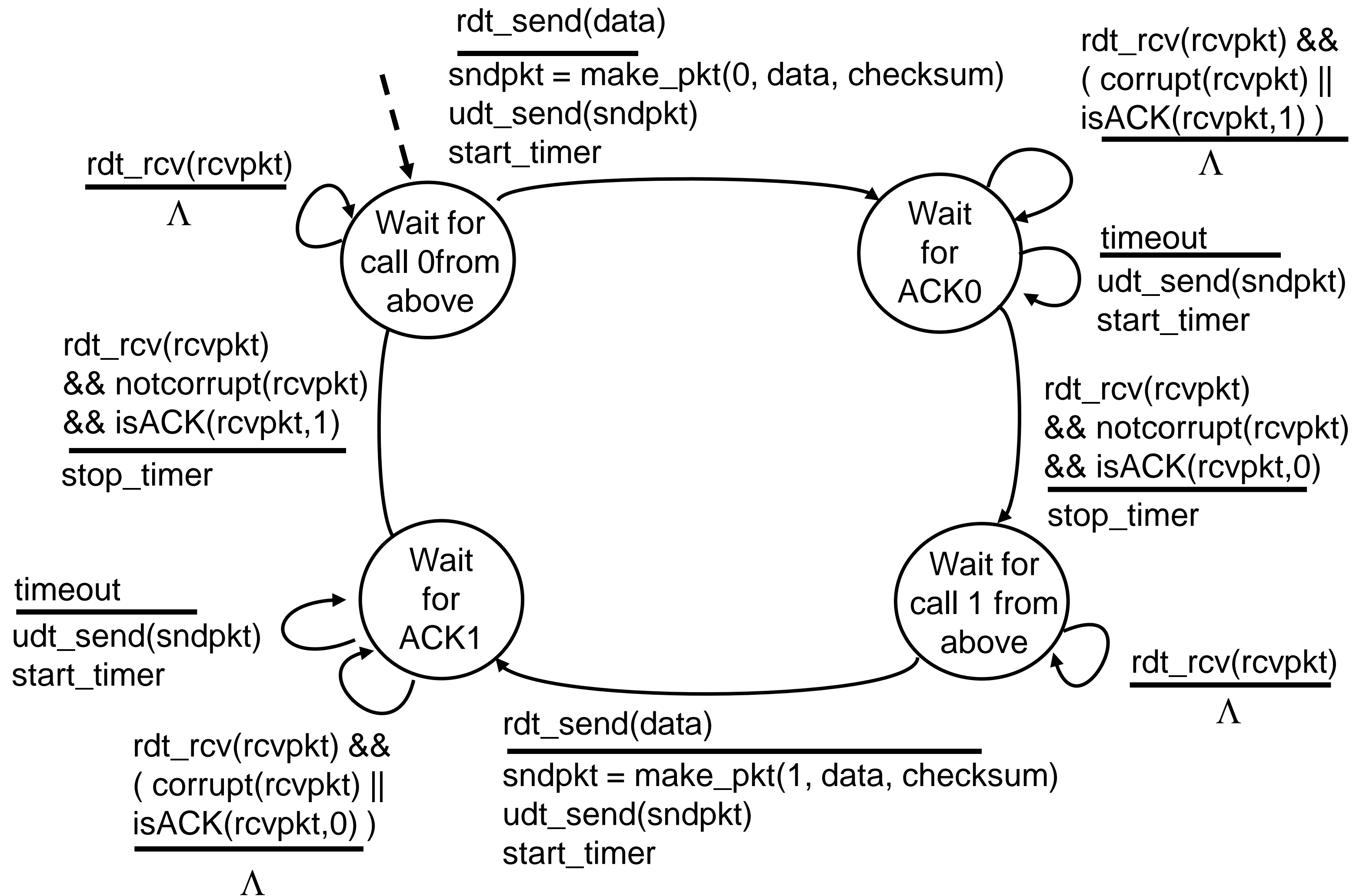
Nova hipótese: canal de base pode agora perder pacotes (dados ou ACKs)

- checksum, # de seq., ACKs, retransmissões ajudarão mas não serão suficientes

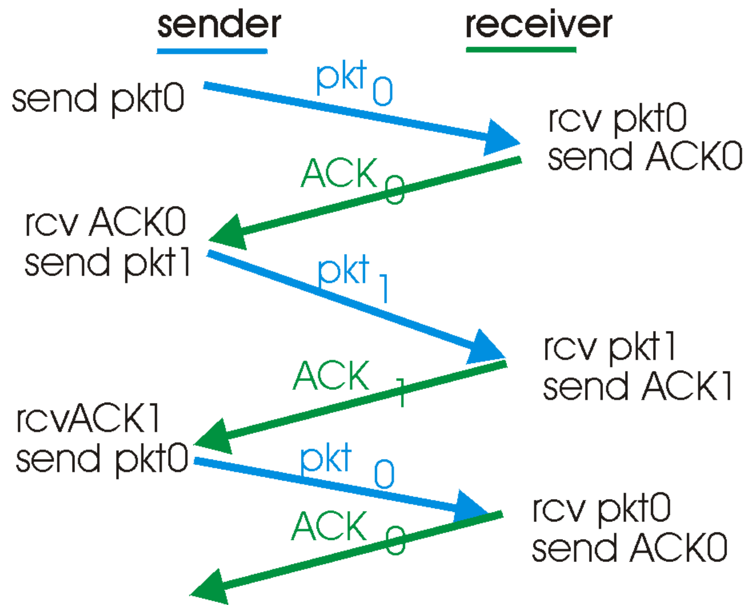
Abordagem: emissor aguarda um tempo "razoável" a recepção de ACKs

- retransmite se nenhum ACK é recebido neste tempo
- se pkt (ou ACK) apenas atrasado (não foi perdido):
  - retransmissão causará duplicação, mas uso de # de seq. tratam isso
  - receptor deve especificar # de seq. do pacote sendo confirmado
- requer temporizador

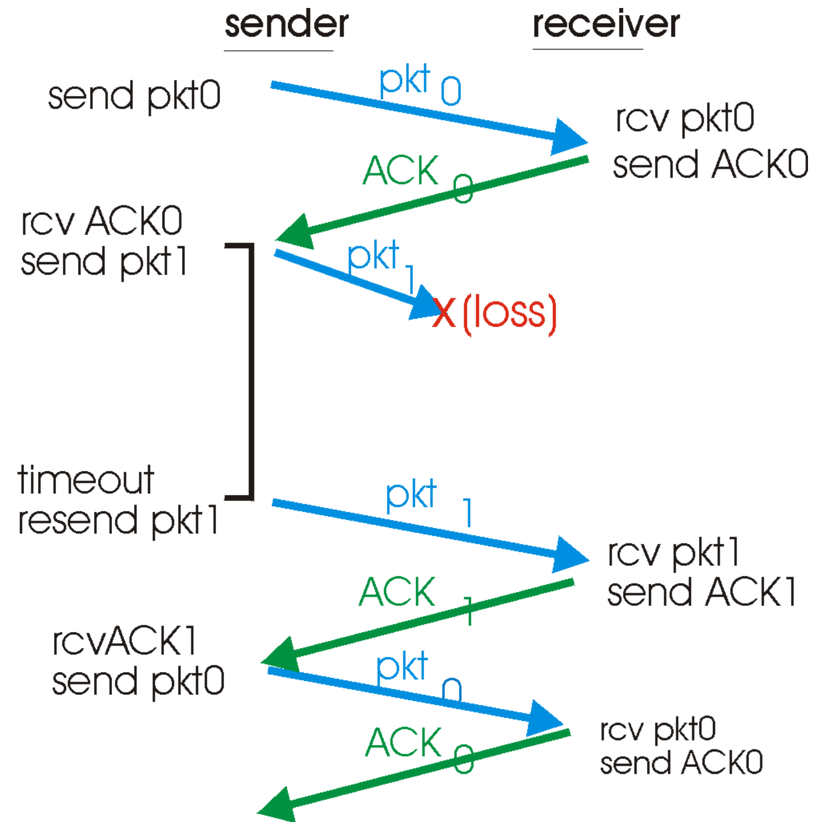
Emissor rdt3.0



rdt3.0 em ação

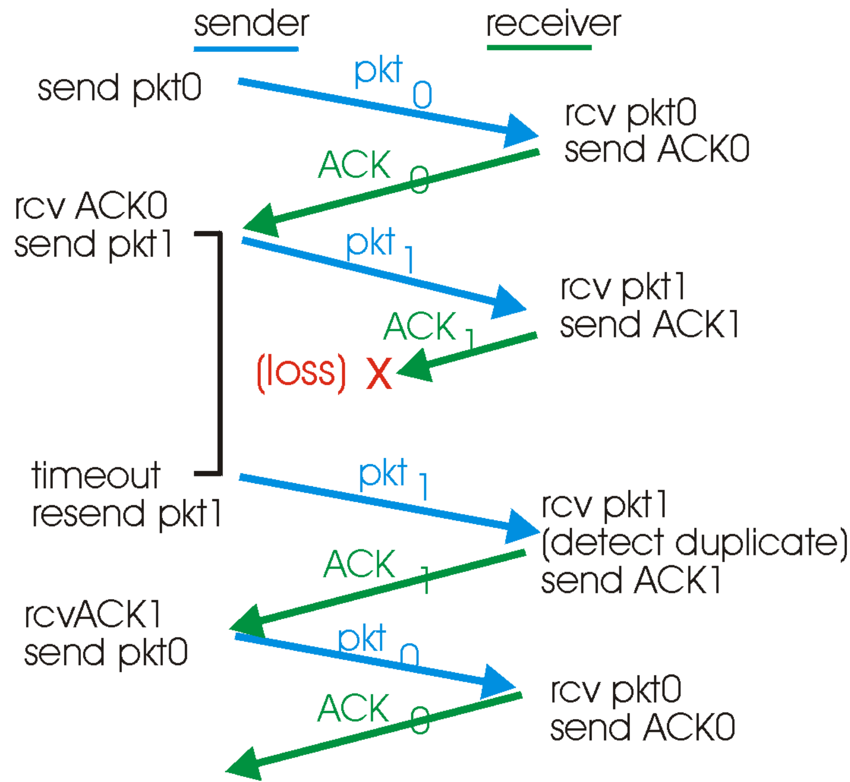


(a) operation with no loss

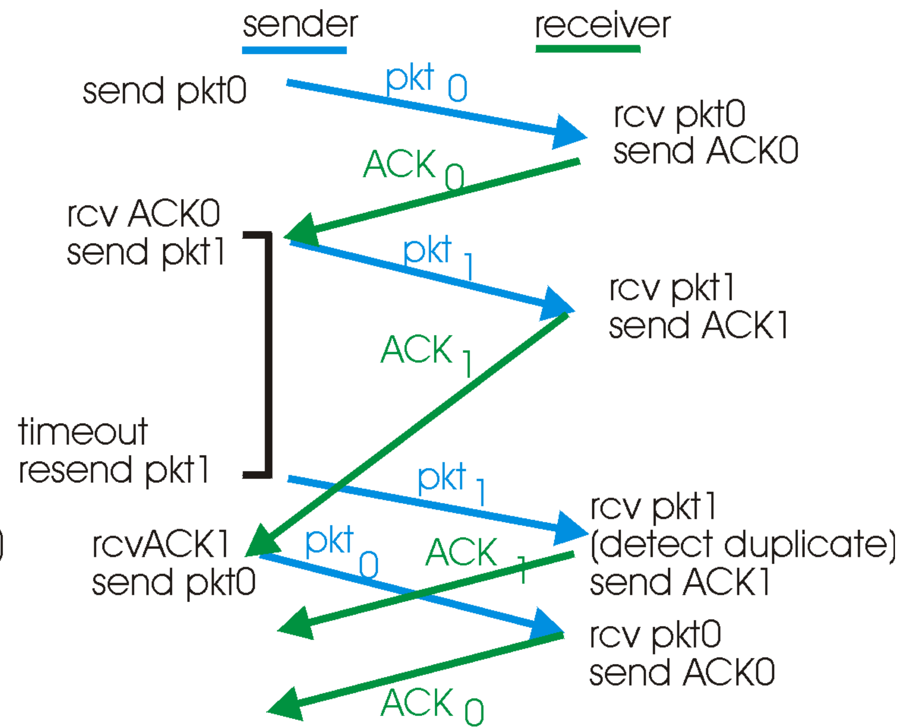


(b) lost packet

rdt3.0 em ação

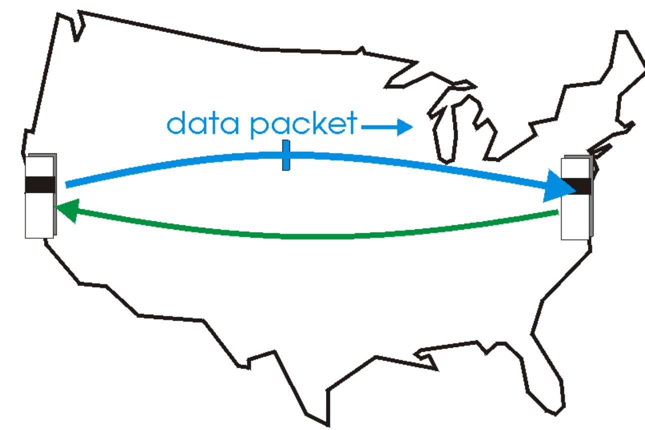


(c) lost ACK



(d) premature timeout

Desempenho do rdt3.0



(a) a stop-and-wait protocol in operation

- rdt3.0 funciona, mas desempenho é ruim
- examplo: enlace de 1 Gbps, 15 ms de atraso de propagação fim-a-fim, pacote de 1KB:

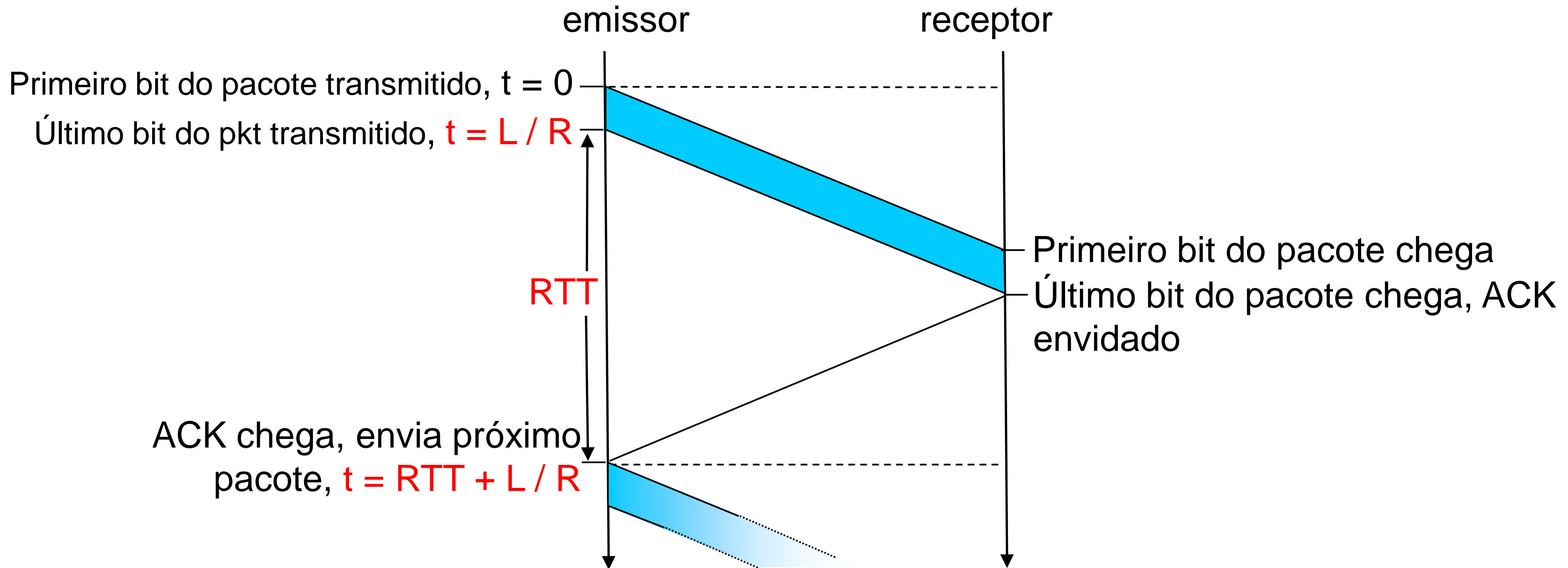
$$T_{transmit} = \frac{L \text{ (tamanho do pacote bits)}}{R \text{ (taxa de transmissão, bps)}} = \frac{8\text{kb/pkt}}{10^{**}9 \text{ b/sec}} = 8 \text{ microsec}$$

- U_{sender}: utilização : fração do tempo em que o emissor está ocupado enviando

$$U_{sender} = \frac{L/R}{RTT + L/R} = \frac{.008}{30.008} = 0.00027 = 0.027\%$$

- Pacote de 1KB a cada 30 msec -> 33kB/sec de vazão em um enlace de 1 Gbps !
- O protocolo de rede limita o uso de recursos físicos!

rdt3.0: funcionamento do stop-and-wait

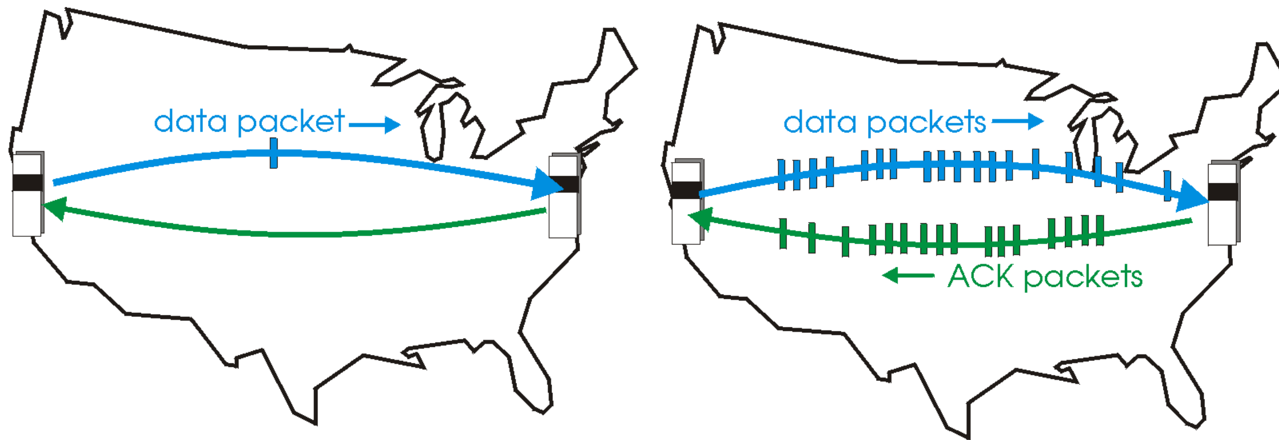


$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027 = 0.027\%$$

Protocolos com Pipeline

Pipelining: emissor permite múltiplos pacotes enviados e que ainda não foram confirmados

- range do número de seqüência deve ser aumentado
- "bufferização" no emissor e/ou receptor

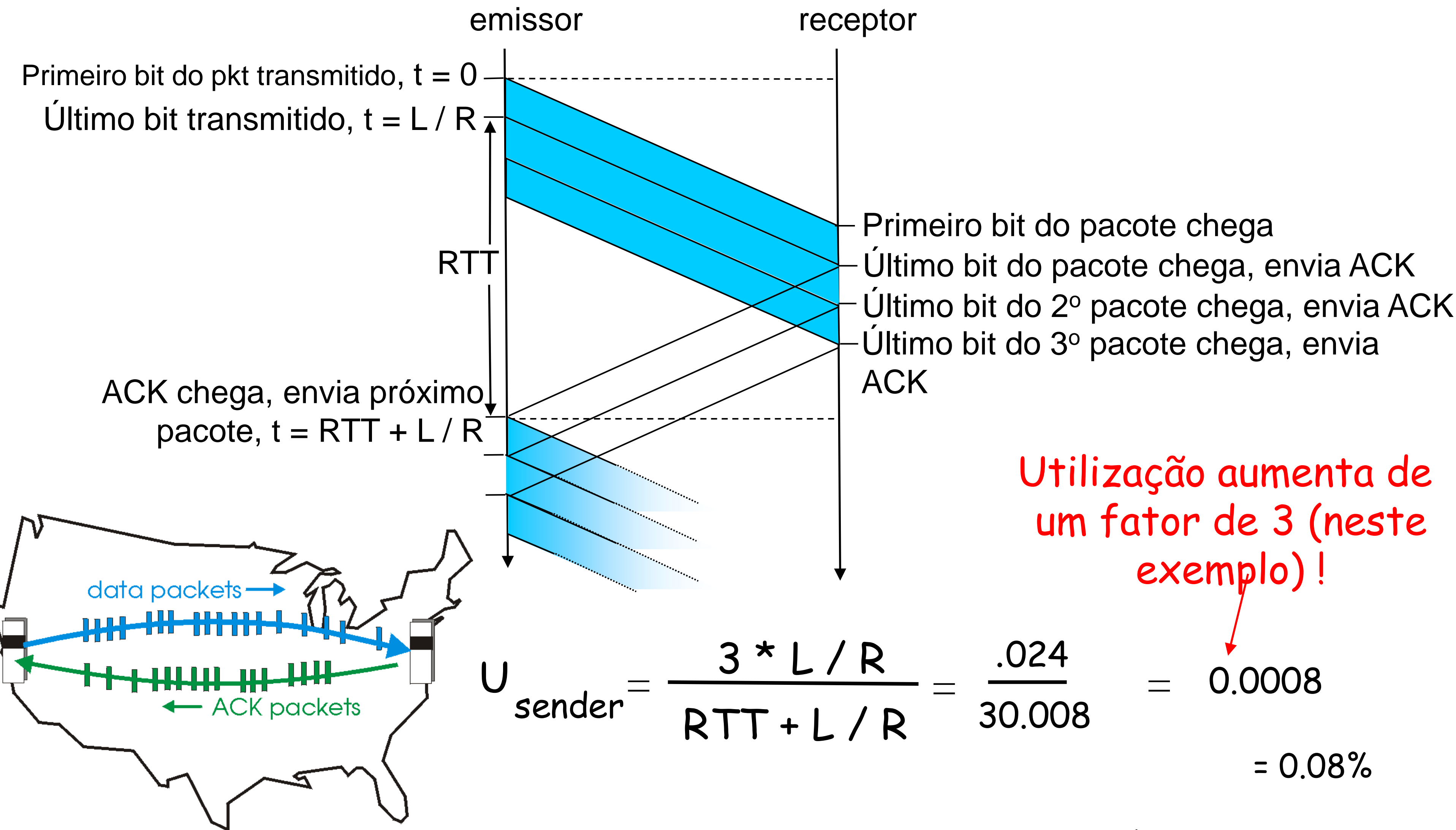


(a) a stop-and-wait protocol in operation

(b) a pipelined protocol in operation

- 2 formas genéricas de protocolos pipeline: *go-Back-N, selective repeat (SR)*

Pipelining: aumento da utilização



(b) a pipelined protocol in operation

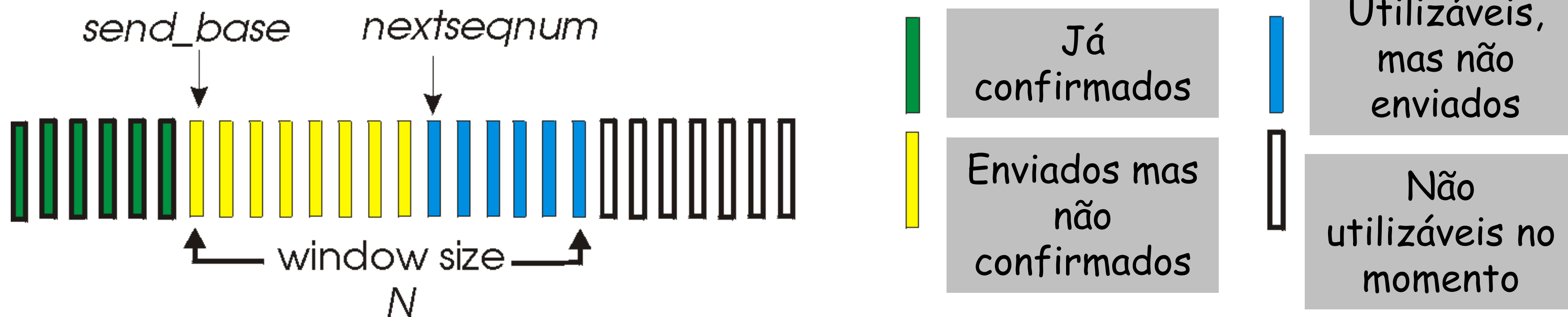
$$U_{sender} = \frac{3 * L / R}{RTT + L / R} = \frac{.024}{30.008} = 0.0008$$

= 0.08%

Go-Back-N

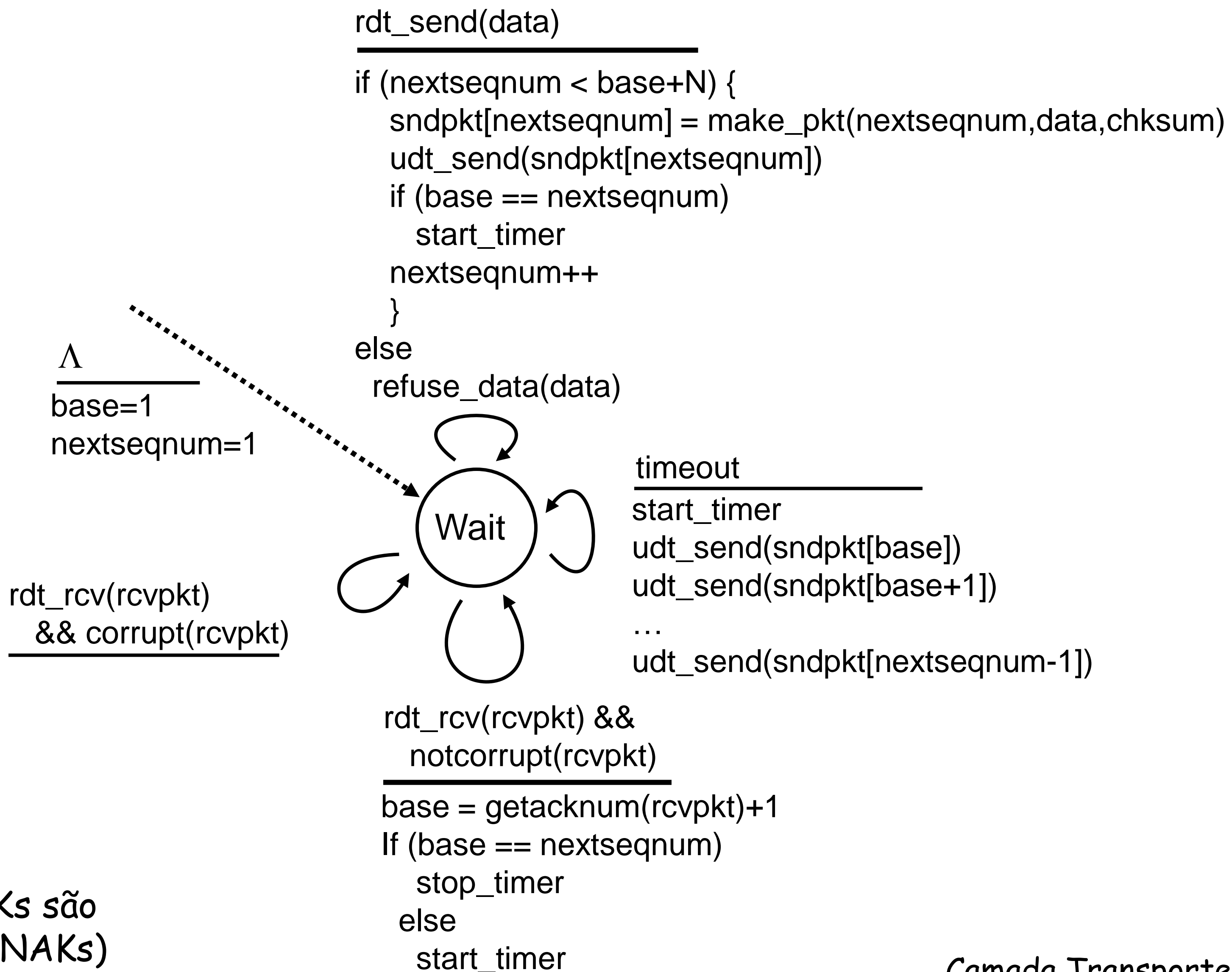
Emissor:

- # de seq. de k bits no cabeçalho do pacote
- "janela" de até N consecutivos pacotes não confirmados permitida



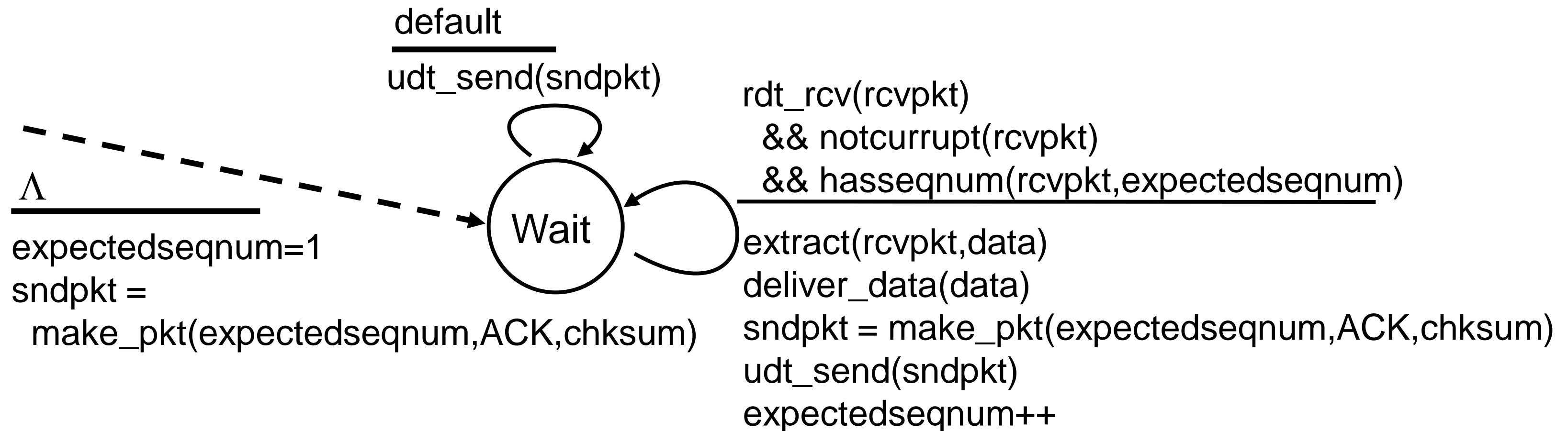
- ACK(n): confirma todos os pacotes até o # n - "ACK cumulativo"
  - Pode receber ACKs duplicados ACKs (veja receptor)
- Temporizador para cada pacote enviado
- *timeout(n):* retransmite pkt n e todos pacotes com #s de seq. maiores que estão dentro da janela

GBN: FSM do emissor estendida



Somente ACKs são usados (sem NAKs)

GBN: FSM do receptor estendida

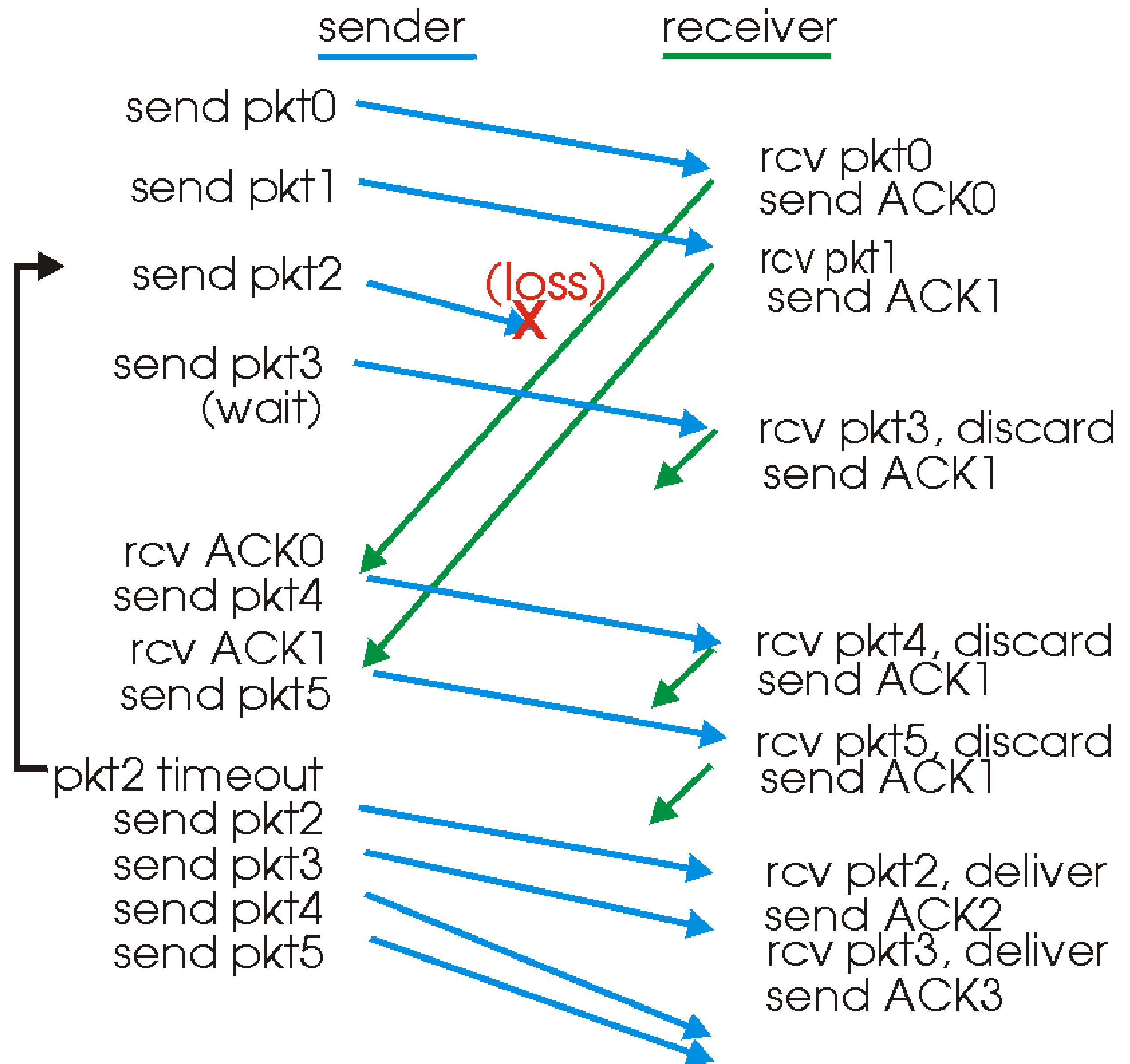


ACK-only: sempre envia ACK para pacote recebido "*na ordem*" corretamente com # de seq. mais elevado

- Pode gerar ACKs duplicados
- Precisa recordar somente o # de seq. esperado (**`expectedseqnum`**)

- Pkt fora de ordem:
  - descarta (não armazena) -> receptor não possui buffer!
  - Re-ACK pkt com o # de seq. mais elevado e em ordem

GBN em ação

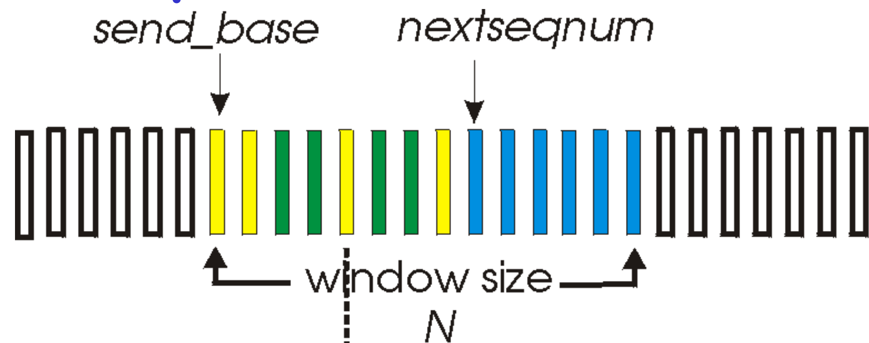


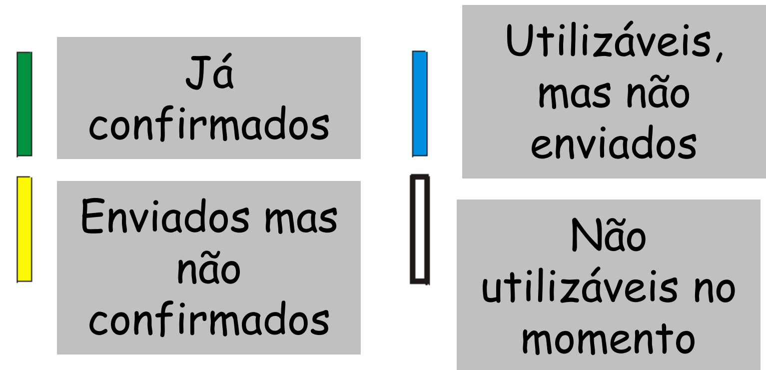
Tamanho da janela = 4

SR - Selective Repeat

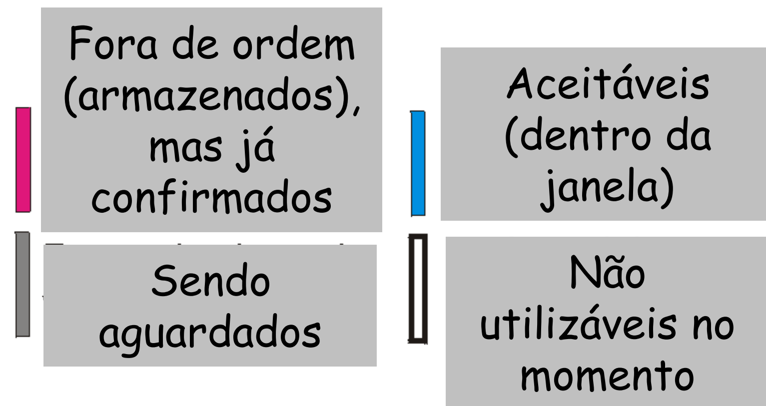
- Receptor confirma individualmente cada pkt recebido corretamente
  - armazena pkts, quando necessário, para eventual envio de pacotes ordenados à camada superior
- Emissor reenvia somente pkts para os quais o ACK não foi recebido
  - Emissor possui temporizador para cada pacote não confirmado (unACKed pkt)
- Janela do emissor
  - N consecutivos #s de seq.
  - Novamente limita #s de seq. de enviados, pkts não confirmados (unACKed pkts)

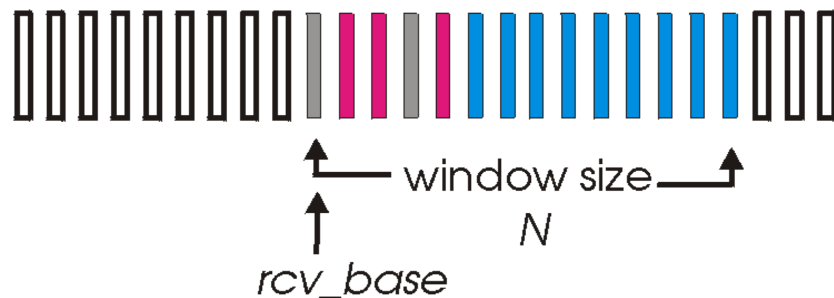
Selective repeat: janelas do emissor e receptor





Números de seq. do ponto de vista do emissor





Números de seq. do ponto de vista do receptor

Selective repeat

emissor

Dados da camada superior :

- Se próximo # de seq. disponível dentro da janela, envia pkt

timeout(n):

- reenvia pkt n, reinicia temporizador

ACK(n) no intervalo [sendbase,sendbase+N]:

- marca pkt n como recebido
- se n é o menor # de seq. de pacotes não confirmados, avança base da janela para o próximo # de seq. não confirmado

receptor

pkt n no intervalo [rcvbase, rcvbase+N-1]

- envia ACK(n)
- For a de ordem: buffer
- Em ordem: entrega (incluindo pkts ordenados no buffer), avança janela para o próximo pkt ainda não recebido

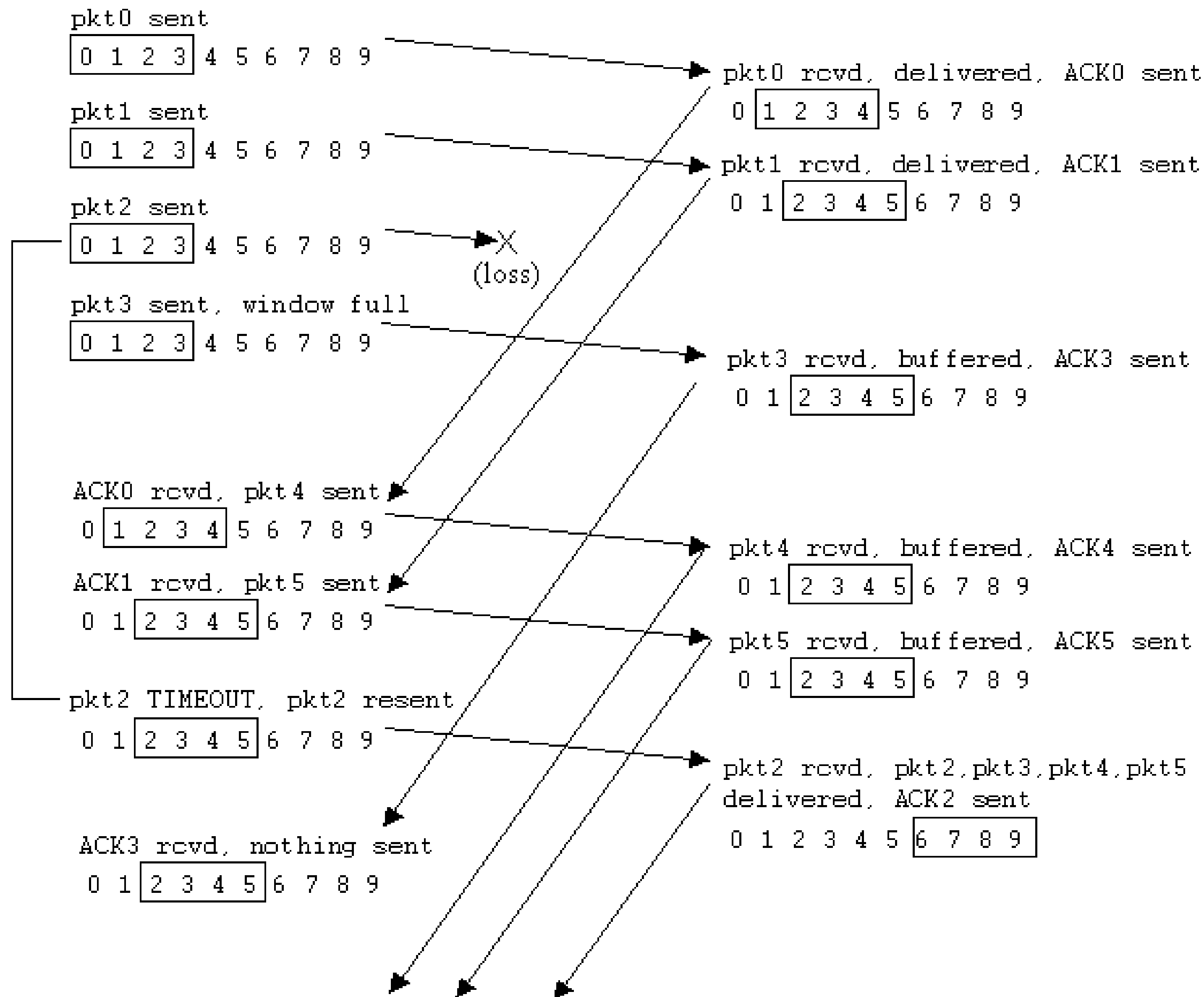
pkt n no intervalo [rcvbase-N,rcvbase-1]

- ACK(n)

Caso contrário:

- ignore

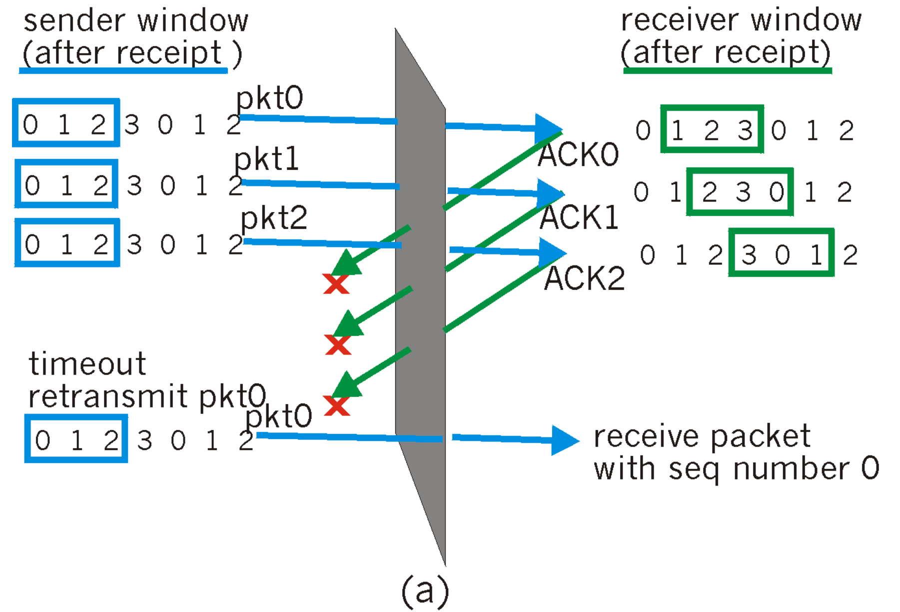
Selective repeat em ação



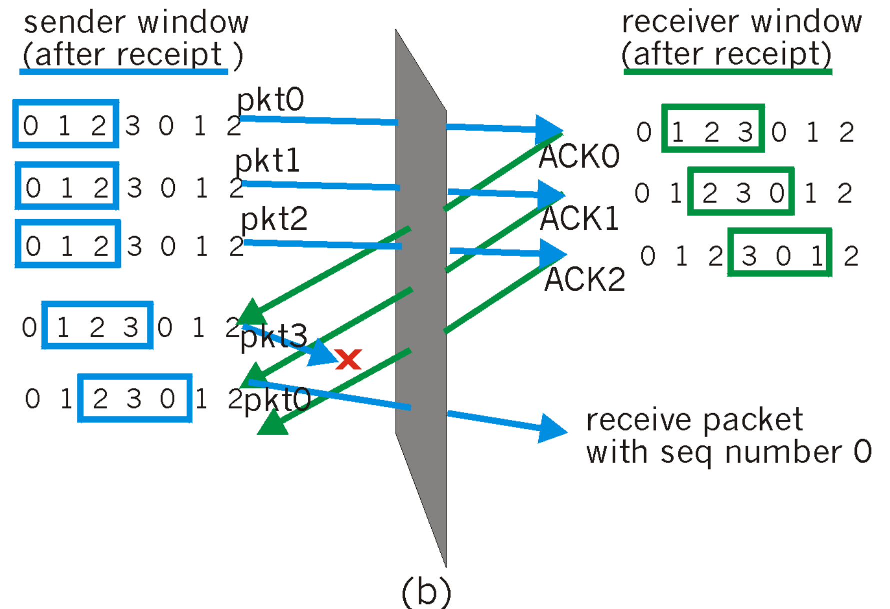
Selective repeat: dilema

Exemplo:

- #s de seq.: 0, 1, 2, 3
- Tamanho da janela =3



- Receptor não vê diferença nos dois cenários!
- Incorretamente passa dados duplicados como sendo novos no exemplo (a)



Q: qual a relação entre o o # de seq. máximo e o tamanho da janela?

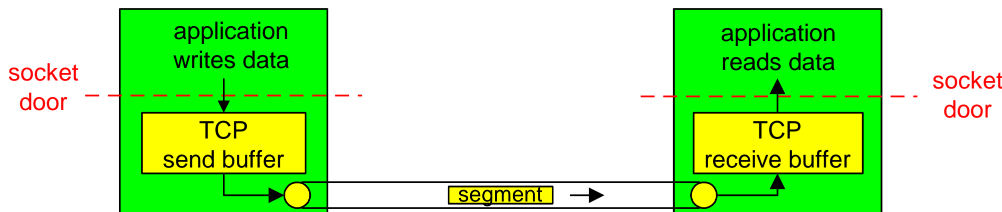
Resumo do Módulo 3

- 3.1 Serviços da camada transporte
- 3.2 Multiplexação e demultiplexão
- 3.3 transporte não-orientado à conexão: UDP
- 3.4 Princípios da transferência confiável de dados
- 3.5 transporte orientado à conexão: TCP
  - Estrutura do segmento
  - Transferência de dados confiável
  - Controle de fluxo
  - Gerenciamento de conexão
- 3.6 Princípios do controle de congestionamento
- 3.7 controle de congestionamento TCP

TCP: Overview

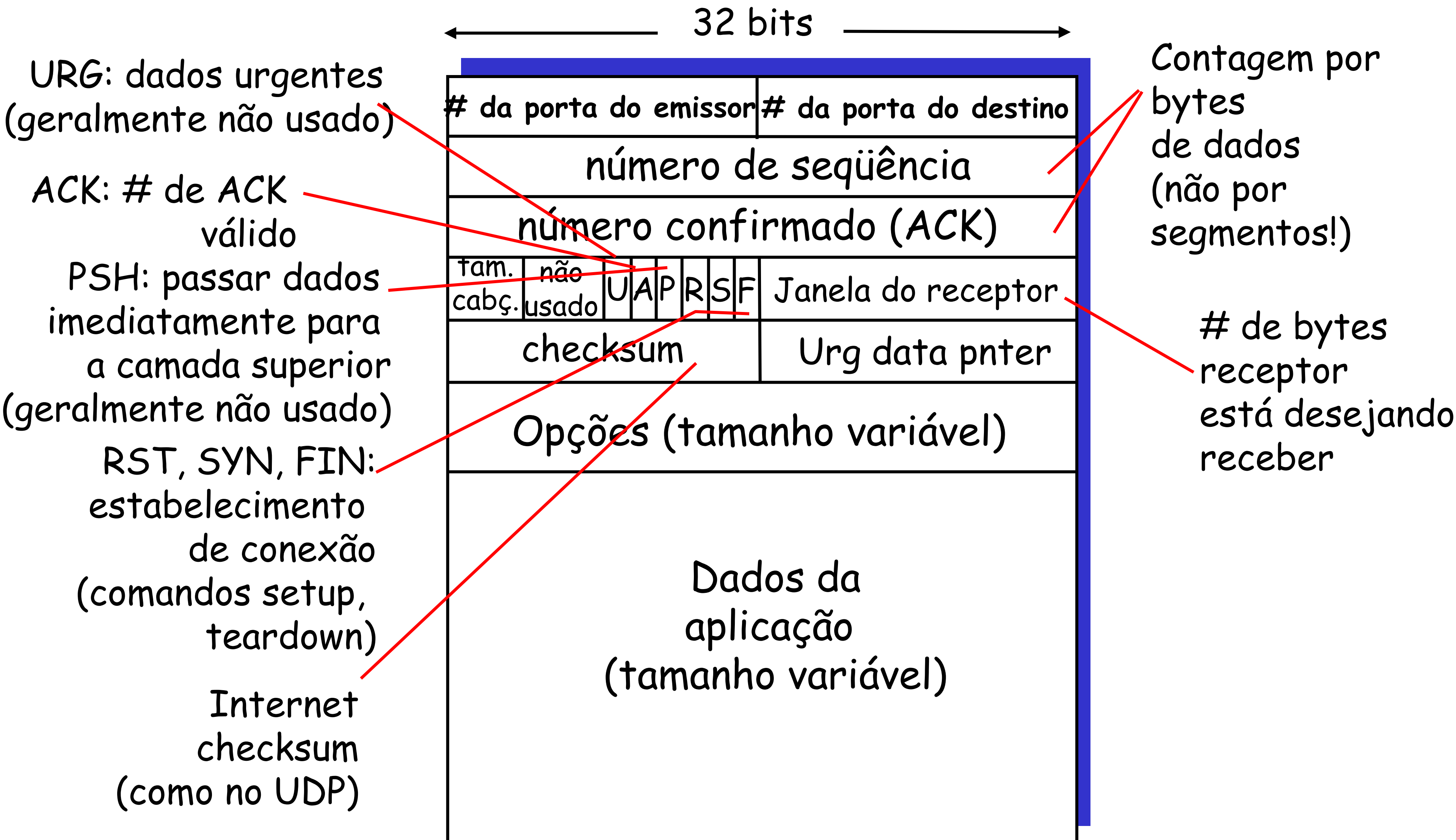
RFCs: 793, 1122, 1323, 2018, 2581

- ponto-a-ponto:
  - Um emissor, um receptor
- Stream de bytes confiável e em ordem*:*
  - "mensagens não possuem limites"
- pipelined:
  - Controle de fluxo e congestionamento do TCP setam tamanho da janela
- *Buffers nos emissor e receptor*



- Dados full duplex:
  - Fluxo de dados bidirecional sobre a mesma conexão
  - MSS: Maximum Segment Size (tamanho máximo de segmento)
- Orientado à conexão:
  - handshaking (troca de msgs de controle) inicia emissor e estado do receptor antes da troca de dados
- Fluxo controlado:
  - Emissor não vai enviar mais dados que o receptor possa receber

Estrutura do segmento TCP



de seq. e ACKs noTCP

#s de seq:

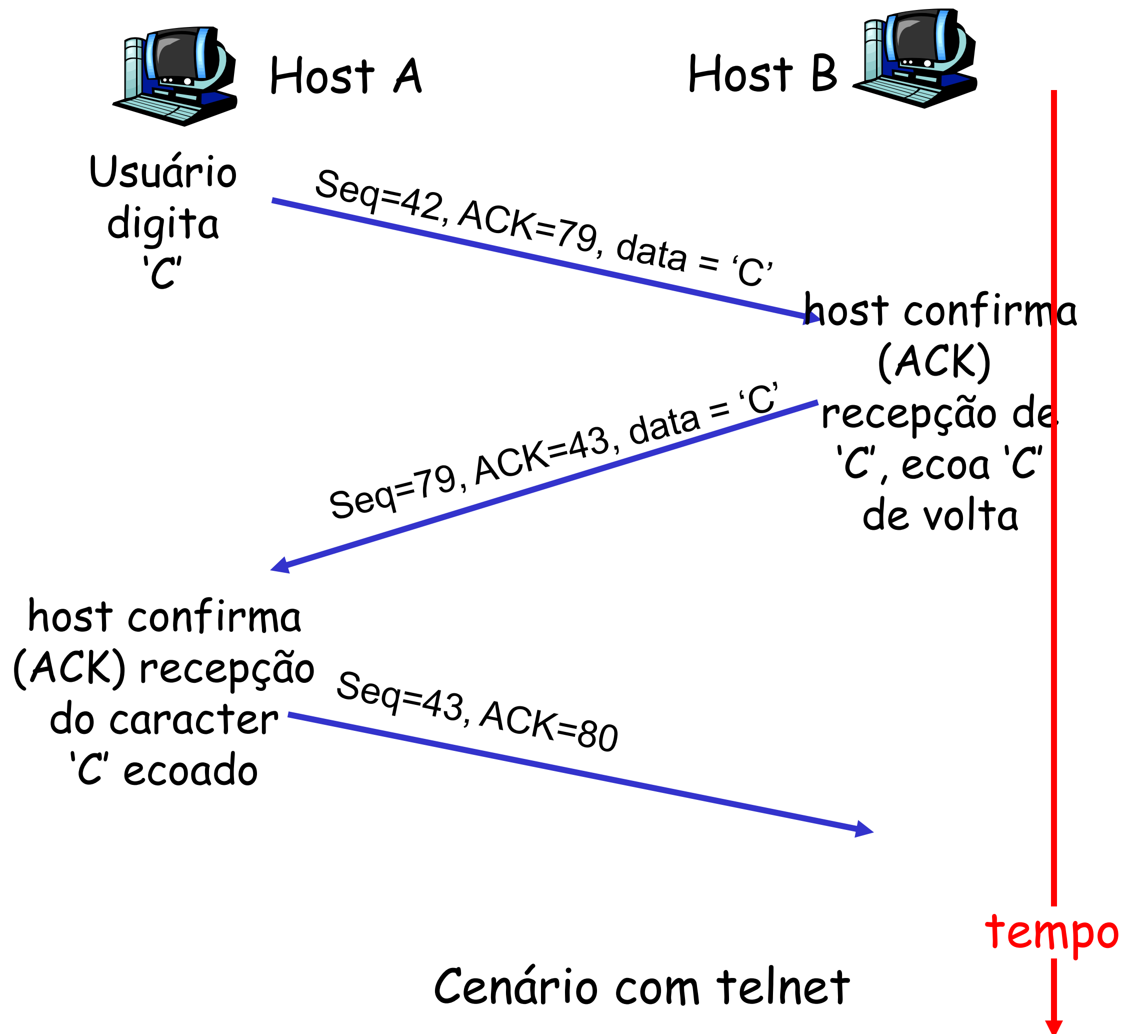
- byte stream "number" of first byte in segment's data

ACKs:

- # de seq do próximo byte aguardado pelo outro lado
- ACK cumulativo

Q: como receptor trata segmentos fora de ordem?

- R: especificação TCP não diz – deixada para o desenvolvedor



Round Trip Time (RTT) e Timeout no TCP

Q: como setar o valor do timeout do TCP?

- maior que o RTT
  - mas o RTT varia
- Muito curto: timeout prematuro
  - Retransmissões desnecessárias
- Muito longo: reação lenta à perda de segmentos

Q: como estimar o RTT?

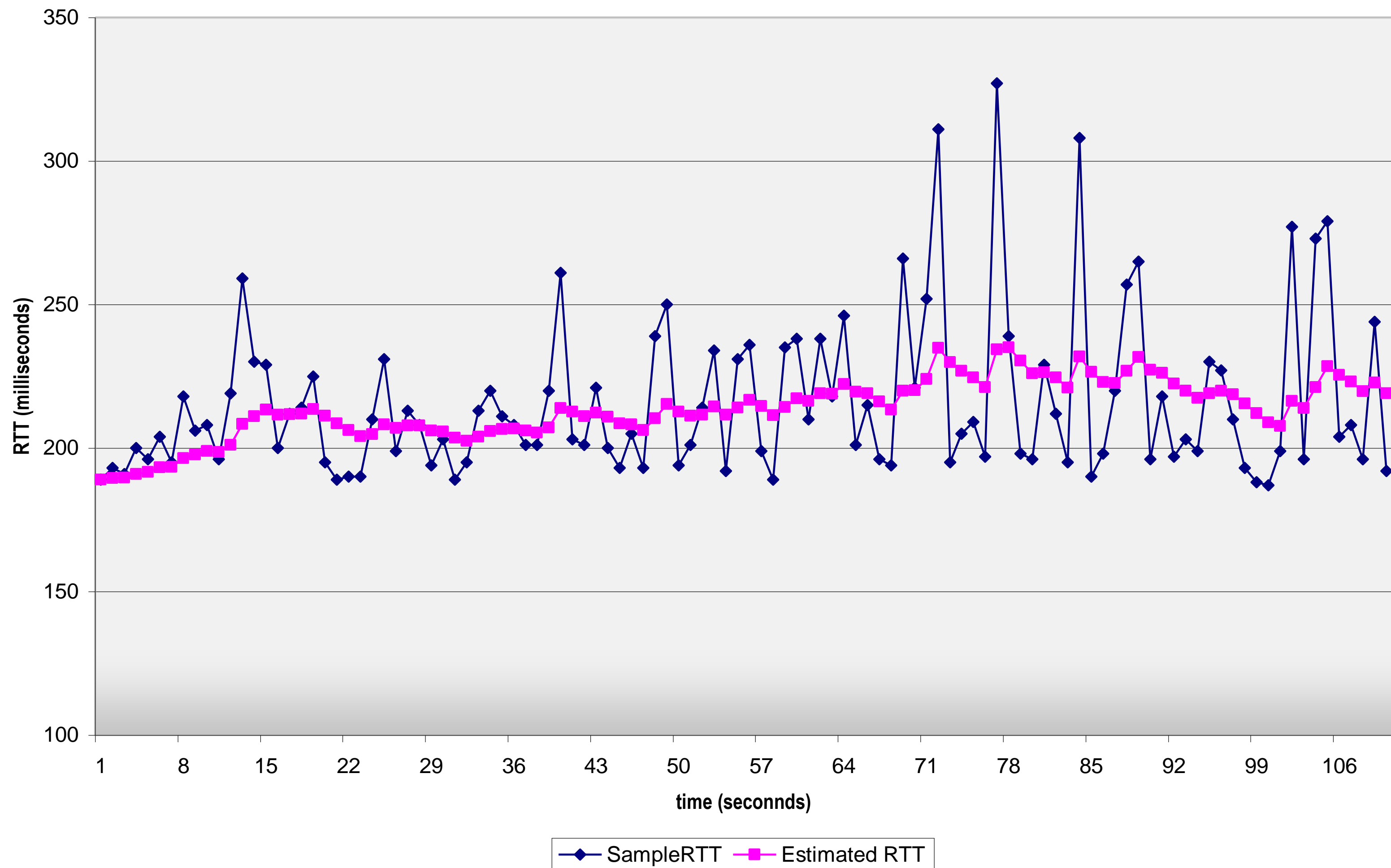
- `SampleRTT`: tempo medido a partir da transmissão do segmento até a recepção do ACK
  - ignora retransmissões
- `SampleRTT` variará, deseja-se uma estimativa de RTT "suavizada"
  - Faz-se uma média de várias medidas recentes, não se usa somente a atual (`SampleRTT`)

Round Trip Time (RTT) e Timeout no TCP

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

- Exponential Weighted Moving Average (EWMA)
- Influência de amostras do passado decresce exponencialmente de forma rápida
  - Peso para um dado SampleRTT decai exponencialmente rápido quando atualizações (novos SampleRTTs) são feitas
  - Maior peso para amostras recentes
- Valor típico: $\alpha = 0.125$

Exemplo de estimação do RTT :



Round Trip Time (RTT) e Timeout no TCP

Setando o timeout

- **EstimtedRTT** + "margem segura"
  - Quanto maior a variação no RTT estimado (**EstimatedRTT**)

  -> maior a margem de segurança
- Primeira estimativa de quanto a amostra do RTT (SampleRTT) desvia do RTT estimado (EstimatedRTT):

```
DevRTT = (1-β)*DevRTT +
             β*|SampleRTT-EstimatedRTT|

(tipicamente, β = 0.25)
```

Em seguida, o intervalo de timeout é setado para:

```
TimeoutInterval = EstimatedRTT + 4*DevRTT
```

Resumo do Módulo 3

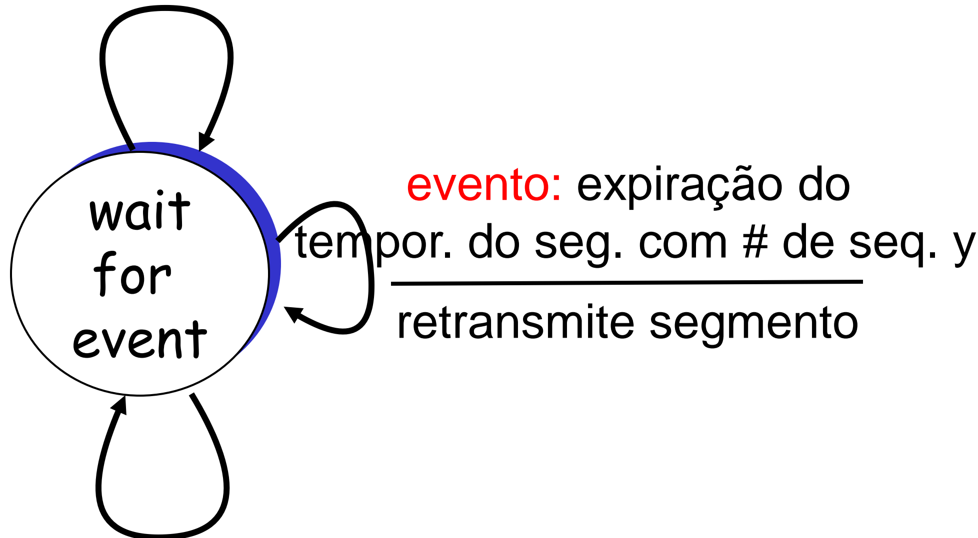
- 3.1 Serviços da camada transporte
- 3.2 Multiplexação e demultiplexão
- 3.3 transporte não-orientado à conexão: UDP
- 3.4 Princípios da transferência confiável de dados
- 3.5 transporte orientado à conexão: TCP
  - Estrutura do segmento
  - Transferência de dados confiável
  - Controle de fluxo
  - Gerenciamento de conexão
- 3.6 Princípios do controle de congestionamento
- 3.7 controle de congestionamento TCP

Transferência confiável de dados no TCP

- TCP cria serviço de transf. confiável de dados no topo do serviço não-confiável do IP
- Segmentos "Pipelined"
- Acks cumulativos
- Conceitualmente o TCP usa múltiplos temporizadores de transmissão

- Retransmissões são disparadas por:
  - Eventos de expiração (timeout)
  - Acks duplicados
- Inicialmente considere um emissor TCP simplificado:
  - ignore acks duplicados
  - ignore controle de fluxo, ignore controle de congestionamento

TCP: transferência confiável de dados



emissor simplificado, supondo:

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

TCP: transfe-rência confiável de dados

emissor 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 */
```

Eventos no emissor TCP:

dados receb. da aplic.:

- Cria segmento com # de seq.
- # de seq. é o número do primeiro byte de dados no segmento
- Inicia temporizador se já não estiver executando
- Intervalo de expiração: `TimeOutInterval`

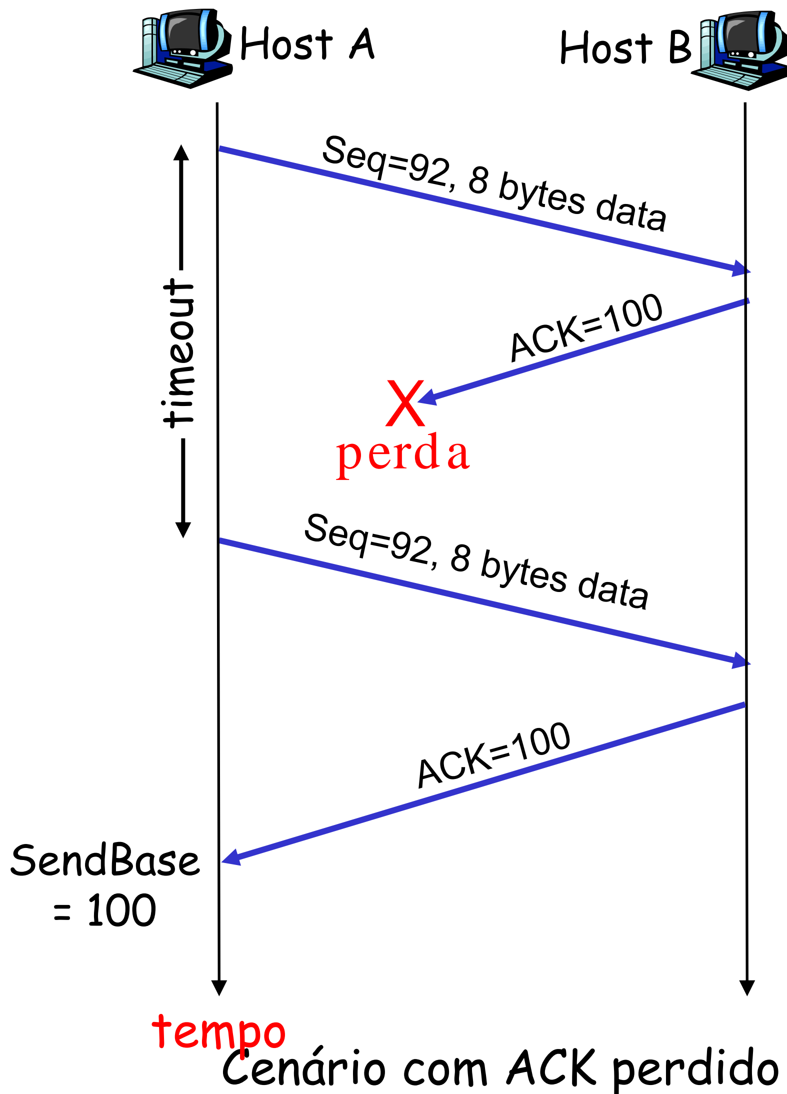
timeout:

- retransmite segmento que causou o timeout
- reinicia temporizador

Ack recebido:

- Se confirma segmentos prévios não-confirmados
  - Atualiza o que é conhecido a ser confirmado
  - Inicia temporizador se existem segmentos devidos

TCP: cenários com retransmissão

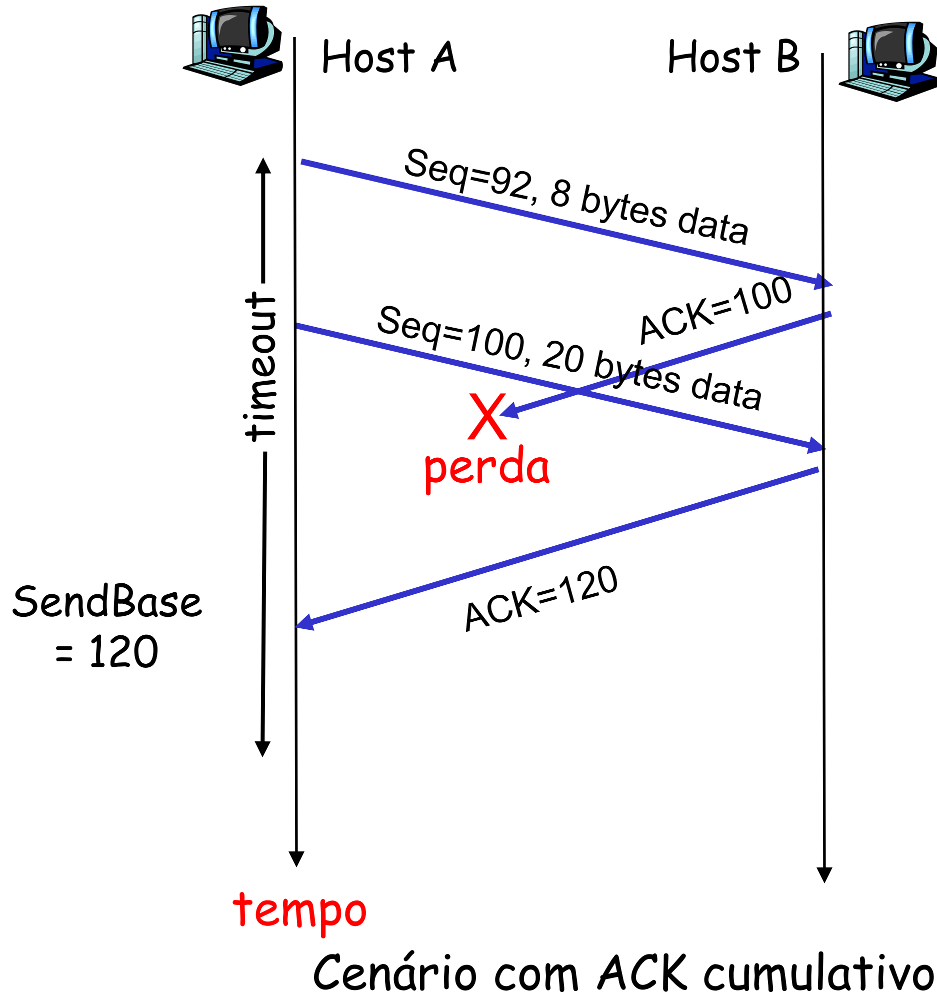


Cenário com ACK perdido

Host A
Host B
Seq=92 timeout
Seq=92, 8 bytes data
Seq=100, 20 bytes data
ACK=100
ACK=120
Sendbase
= 100
SendBase
= 120
Seq=92, 8 bytes data
Seq=92 timeout
ACK=120
SendBase
= 120
tempo

Timeout prematuro

TCP: + cenários com retransmissão



Cenário com ACK cumulativo

Geração de ACK no TCP [recomendações na RFC 1122 e na RFC 2581]

Evento no Receptor	Ação do Receptor TCP
Chegada “na ordem” de segmento com esperado # de seq. Todos os dados até o # de seq. esperado já confirmados. Sem lacunas	Atrasar ACK. Aguarde até 500ms pelo próximo segmento. Se não chegar, envie ACK
Chegada “na ordem” de segmento com # de seq. esperado. Um outro seg “em ordem” aguardando transmissão de ACK	Enviar imediatamente ACK cumulativo único, confirmando ambos segmentos “na ordem”
Chegada de segmento fora de ordem com # de seq. > que esperado. Lacuna detectada	Enviar imediatamente *ACK (duplicado)*, indicando # de seq. do próximo byte esperado
Chegada de segmento que parcialmente ou totalmente preenche a lacuna	Enviar imediatamente ACK, dado que segmento inicia no princípio da lacuna

Fast Retransmit - Retransmissão Rápida

- Período de expiração relativamente longo:
  - Grande atraso antes de reenviar pacote perdido
- Detecta segmentos perdidos via ACKs duplicados.
  - Emissor freqüentemente envia muitos segmentos sucessivos
  - Se segmento é perdido, haverá provavelmente muitos ACKs duplicados.
- Se emissor recebe 3 duplicados ACKs (4 ACKS consecutivos/idênticos/ sem outros pacotes no meio) para o mesmo dado, ele supõe que segmento após dado confirmado foi perdido:
  - fast retransmit: reenvia segmento antes do temporizador expirar

Algoritmo do Fast retransmit:

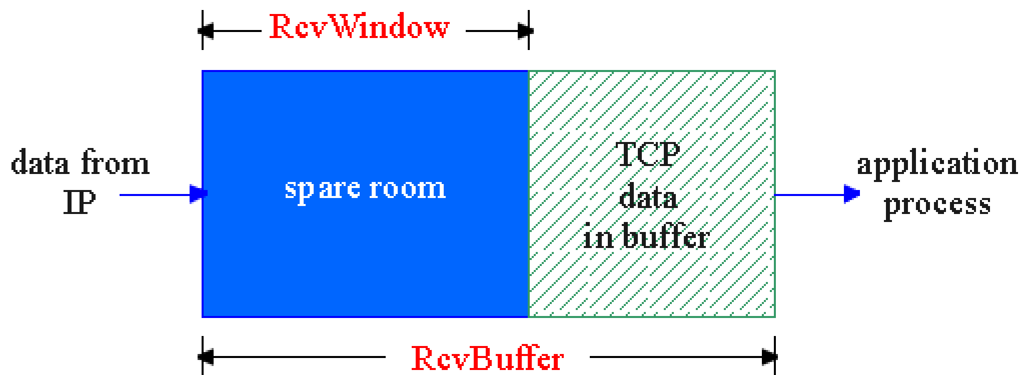
```
evento: ACK recebido, com campo ACK de valor igual a y
            if (y > SendBase) {
                SendBase = y
                if (há atualmente segmentos ainda não confirmados)
                    iniciar temporizador
              }
            else {
                 incrementar contador de ACKs duplicados recebidos para y
                 if (contador de ACKs duplicados recebidos para y = 3) {
                     reenviar segmento com # de seq. y
                 }
```

Resumo do Módulo 3

- 3.1 Serviços da camada transporte
- 3.2 Multiplexação e demultiplexão
- 3.3 transporte não-orientado à conexão: UDP
- 3.4 Princípios da transferência confiável de dados
- 3.5 transporte orientado à conexão: TCP
  - Estrutura do segmento
  - Transferência de dados confiável
  - Controle de fluxo
  - Gerenciamento de conexão
- 3.6 Princípios do controle de congestionamento
- 3.7 controle de congestionamento TCP

Controle de FluxoTCP

- Lado receptor da conexão TCP possui um buffer receptor:



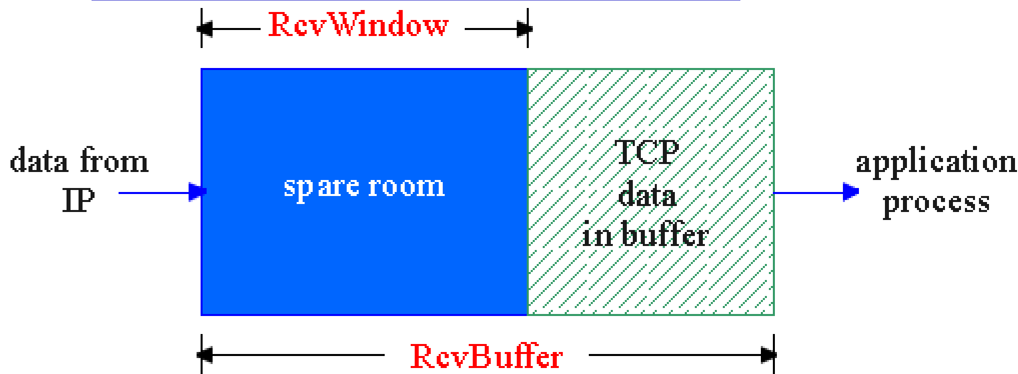
- Processo aplicação pode ser lento ao ler dados do buffer

Controle de fluxo

Emissor não estoura o buffer do receptor transmitindo demais ou muito rápido

- serviço de "casamento" de taxas: "casar" a taxa de envio com a taxa de drenagem de dados da aplicação receptora

Controle de fluxo TCP: funcionamento



(Suponha que receptor TCP descarte segmentos for a de ordem)

- espaço livre no buffer

= **RcvWindow**

= **RcvBuffer-[LastByteRcvd - LastByteRead]**

- Recept. informa espaço livre através da inclusão nos segmentos do valor da janela de recepção **RcvWindow**
- Emissor limita dados não confirmados ao tamanho da janela **RcvWindow**
  - guarante que buffer do receptor não "transborda"

Resumo do Módulo 3

- 3.1 Serviços da camada transporte
- 3.2 Multiplexação e demultiplexão
- 3.3 transporte não-orientado à conexão: UDP
- 3.4 Princípios da transferência confiável de dados
- 3.5 transporte orientado à conexão: TCP
  - Estrutura do segmento
  - Transferência de dados confiável
  - Controle de fluxo
  - Gerenciamento de conexão
- 3.6 Princípios do controle de congestionamento
- 3.7 controle de congestionamento TCP

Gerenciamento de conexão TCP

Lembre-se: estabelecimento de conexão entre o emissor e receptor TCP antes da troca de segmentos de dados

- inicializar variáveis TCP :
  - #s de seq.
  - buffers, informação de controle de fluxo (e.g. **`RcvWindow`**)
- *cliente:* iniciador da conexão

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

- *servidor:* contactado pelo cliente

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

Three-way handshake:

Passo 1: host cliente envia segmento SYN TCP para o servidor

- especifica # de seq. inicial
- sem dados

Passo 2: host servidor recebe SYN, responde com segmento SYNACK

- servidor aloca buffers
- especifica # de seq. inicial do servidor

Passo 3: cliente recebe SYNACK, responde com segmento ACK que pode conter dados

Gerenciamento de ConexãoTCP (cont.)

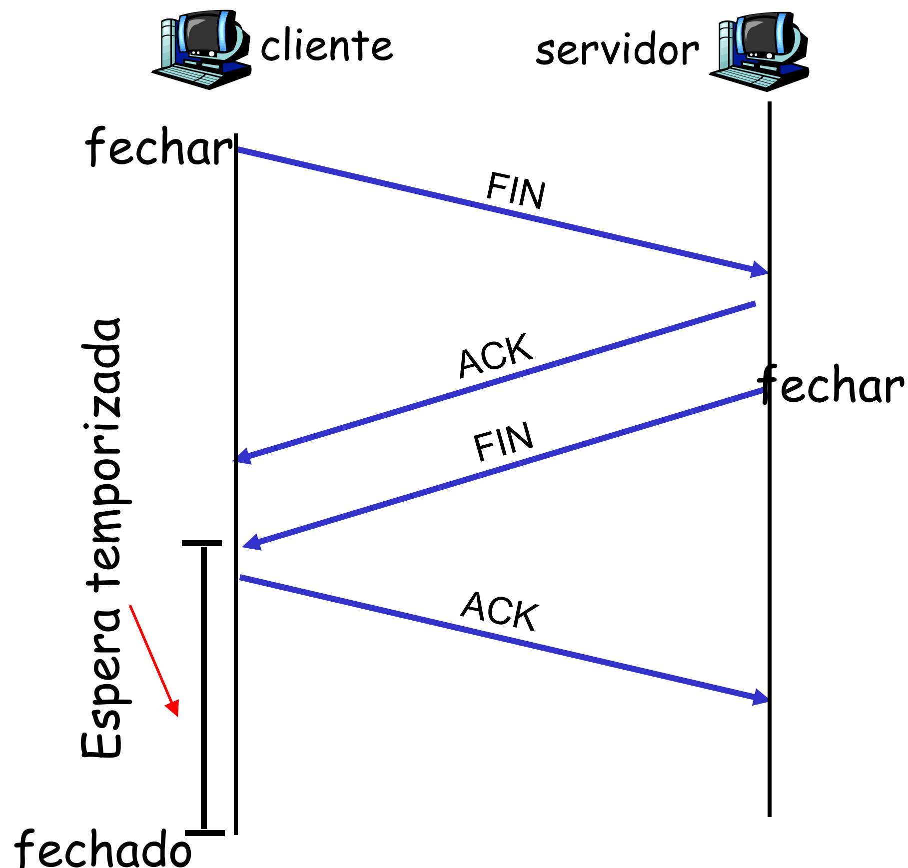
Fechando uma conexão:

cliente fecha socket:

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

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

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

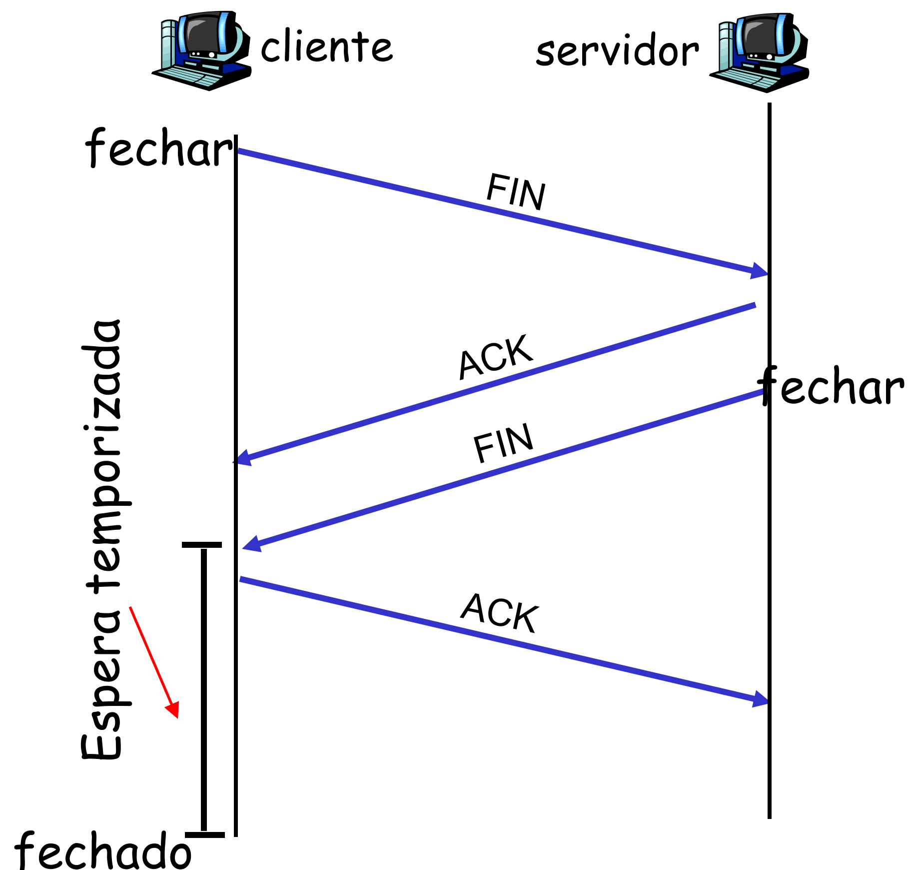


Gerenciamento de Conexão TCP (cont.)

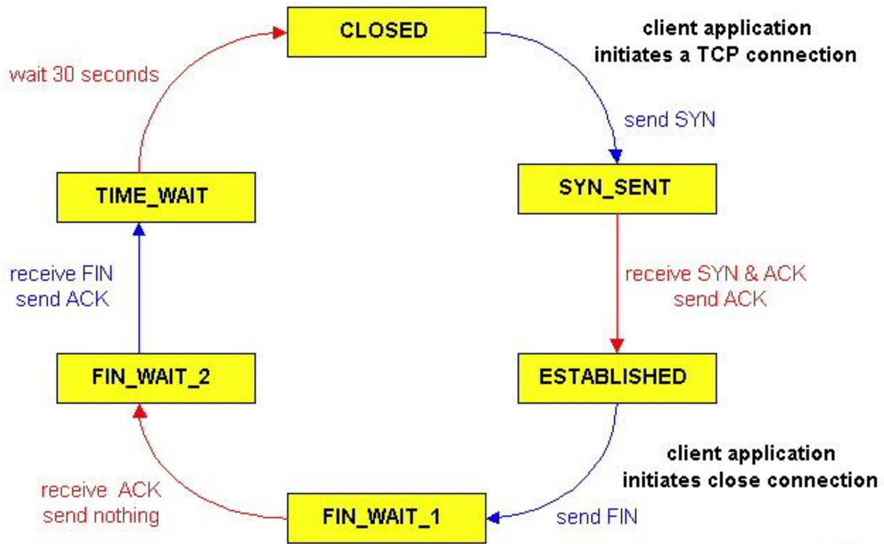
Passo 3: cliente recebe FIN, responde com ACK.

- Entra na "espera temporizada" – responderá com ACK aos FINs recebidos

Passo 4: servidor, recebe ACK. Conexão fechada.

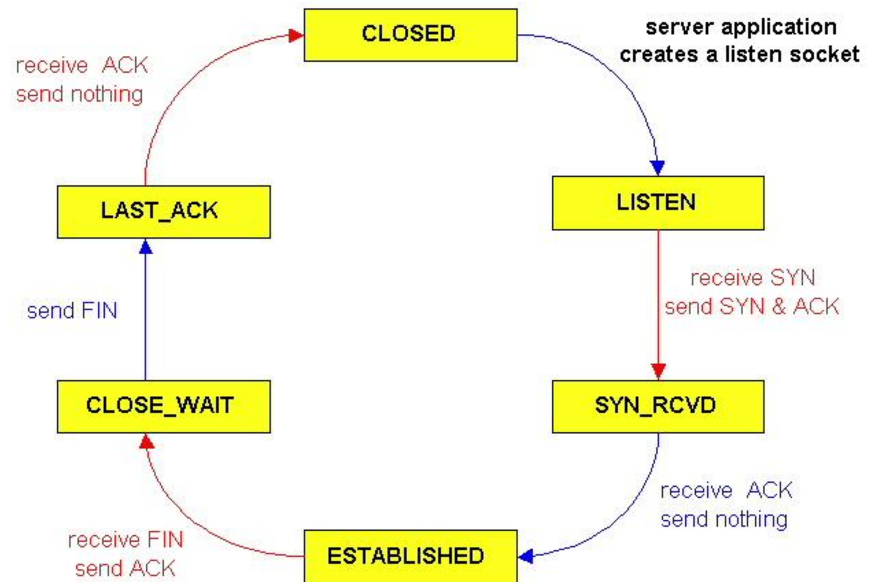


Gerenciamento de Conexão TCP (cont.)



Ciclo de vida do cliente TCP

Ciclo de vida do servidor TCP



Resumo do Módulo 3

- 3.1 Serviços da camada transporte
- 3.2 Multiplexação e demultiplexão
- 3.3 transporte não-orientado à conexão: UDP
- 3.4 Princípios da transferência confiável de dados
- 3.5 transporte orientado à conexão: TCP
  - Estrutura do segmento
  - Transferência de dados confiável
  - Controle de fluxo
  - Gerenciamento de conexão
- 3.6 Princípios do controle de congestionamento
- 3.7 controle de congestionamento TCP

Princípios do Controle de Congestionamento

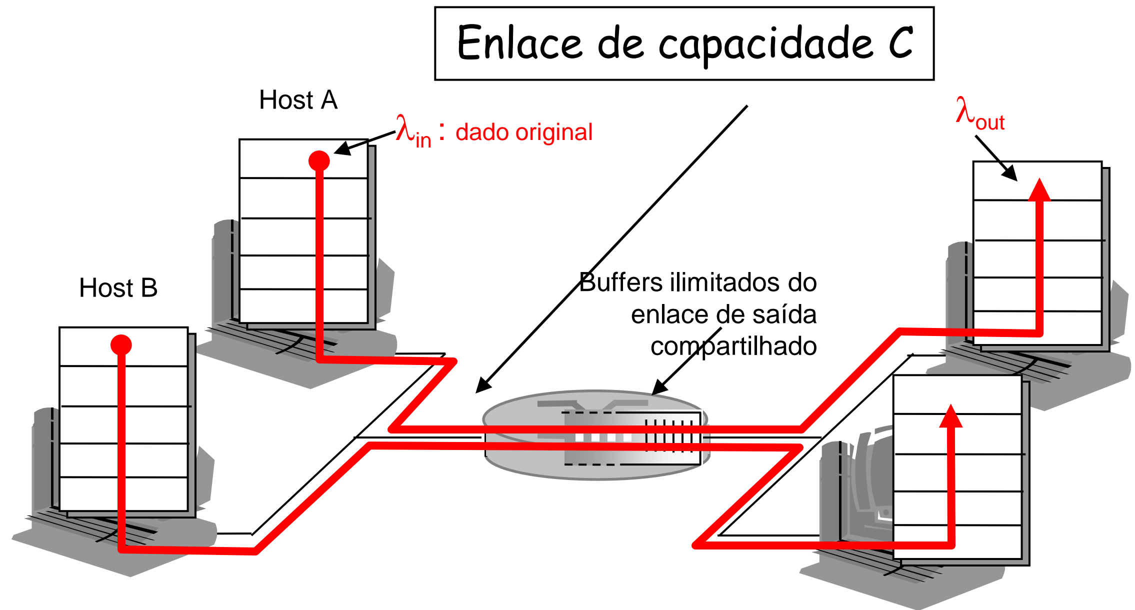
Congestionamento:

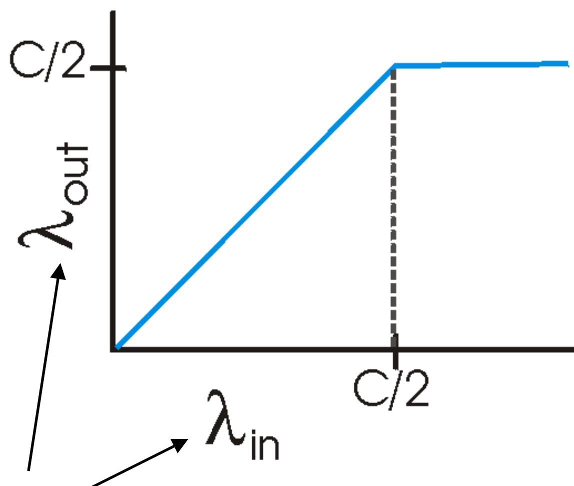
- informalmente: "fontes demais enviando dados demais, muito rápido ultrapassando a capacidade da rede"
- diferente de controle de fluxo!
- manifestações:
  - Pacotes perdidos ("estouro" de buffer nos roteadores)
  - Atrasos elevados ("enfileiramento" em buffers nos roteadores)
- Problema na lista dos top-10!

Causas/custos do congestionamento: cenário 1

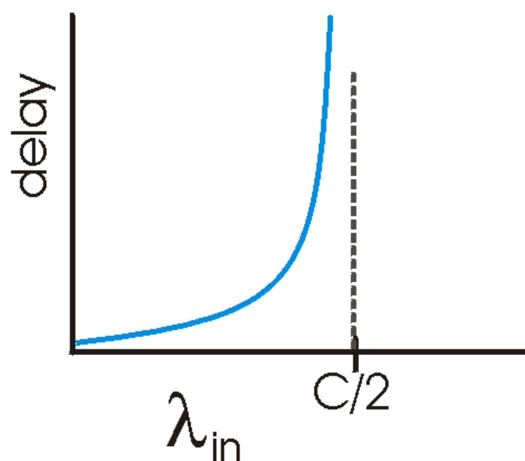
- 2 emissores, 2 receptores
- 1 roteador, buffers infinitos
- sem retransmissões

Enlace de capacidade C





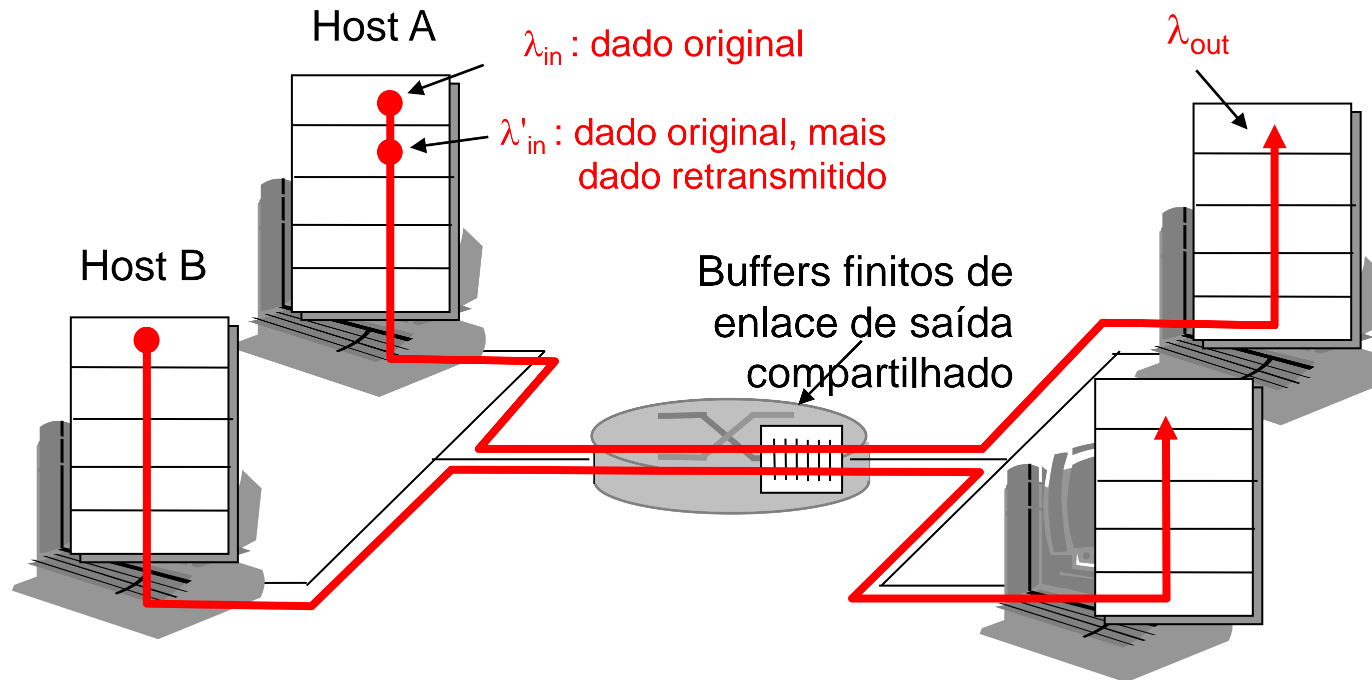
Em bytes/segundo



- Atraso (delay) muito elevado quando rede congestionada
- Vazão (throughput) máxima alcançável

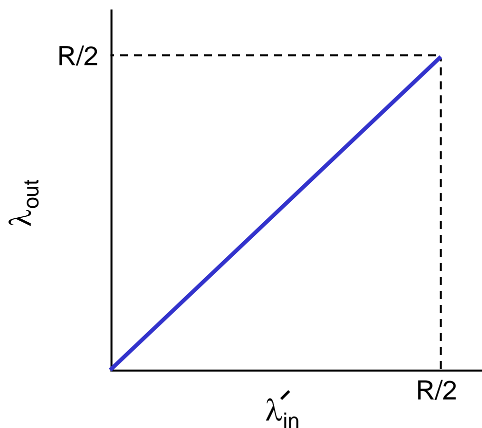
Causas/custos do congestionamento: cenário 2

- 1 roteador, buffers finitos
- emissor: retransmissão de pacote perdido

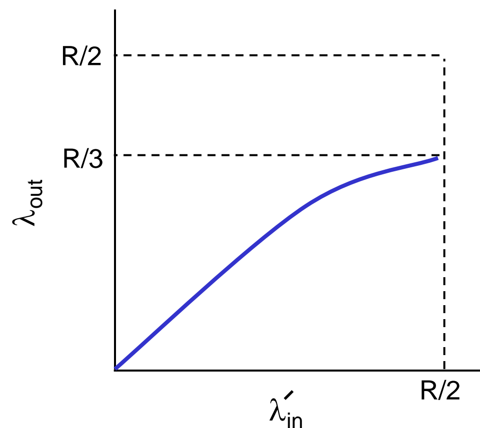


Causas/custos do congestionamento: cenário 2

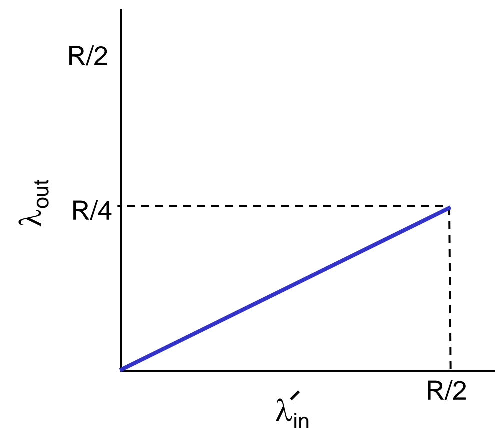
- (a) sempre: $\lambda_{in} = \lambda_{out}$ (goodput)
- (b) Retransmissões "perfeitas" somente quando perda: $\lambda'_{in} > \lambda_{out}$
- (c) Retransmissão de pacotes atrasados (não perdidos) faz λ'_{in} maior (que caso perfeito) para mesmo λ_{out}



a.



b.



c.

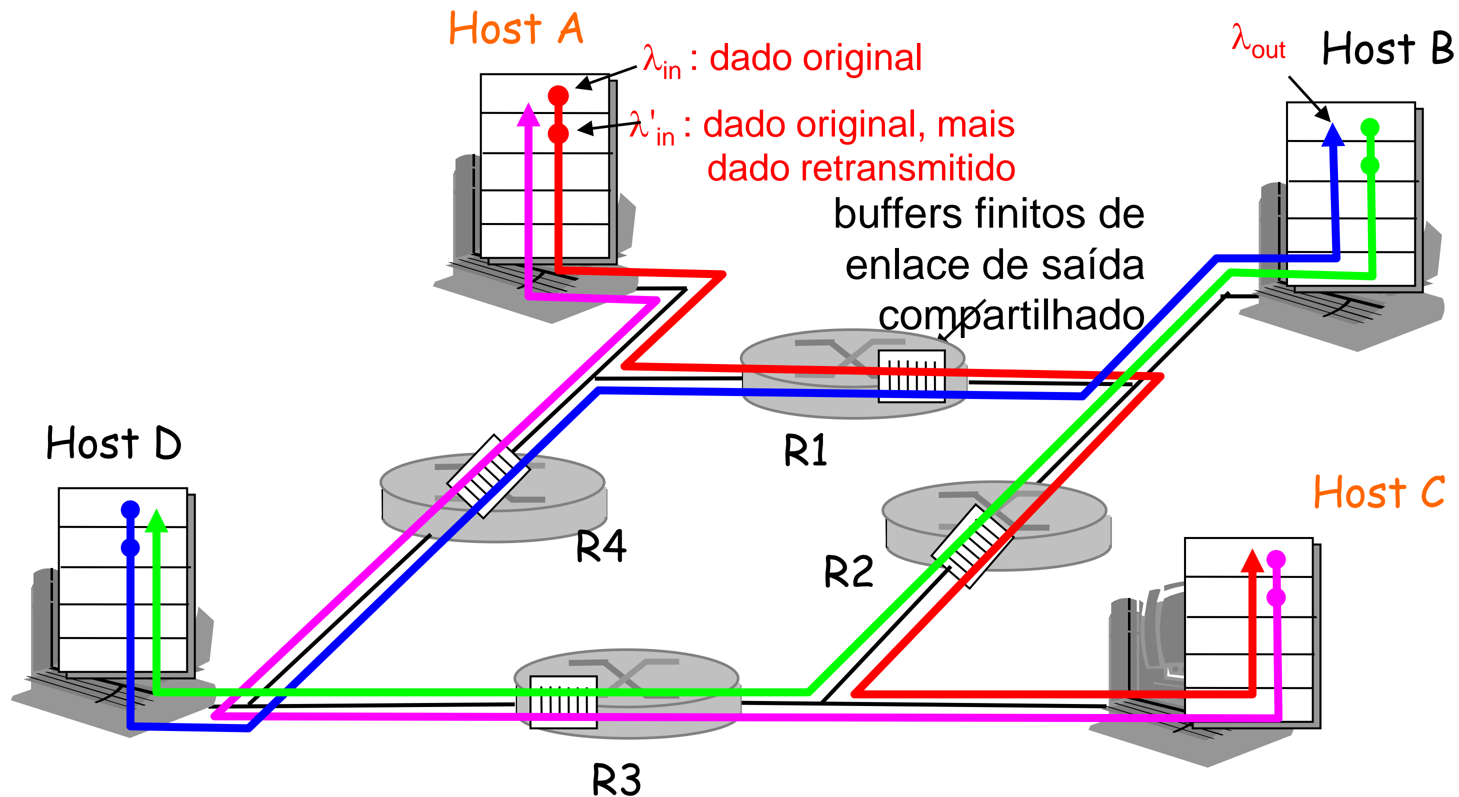
"custos" do congestionamento:

- (b) mais trabalho (retransmissões) para dada "goodput"
- (c) Retransmissões desnecessárias: enlace carrega múltiplas cópias do mesmo pacote

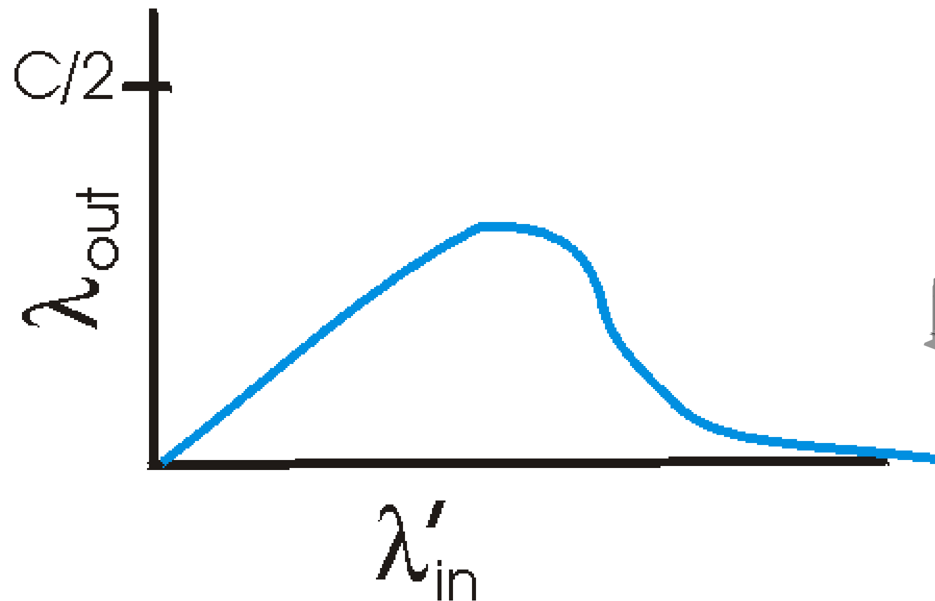
Causas/custos do congestionamento: cenário 3

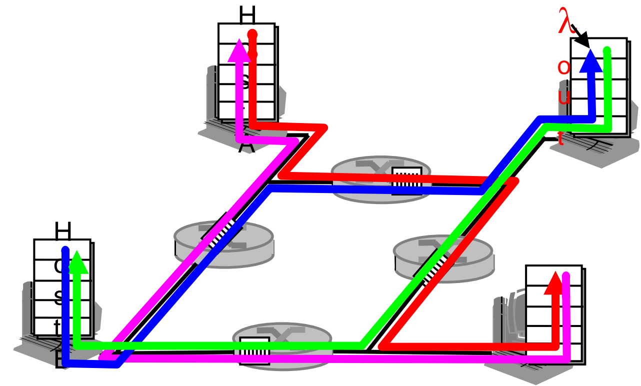
- 4 emissores
- Caminhos com múltiplos saltos
- timeout/retransmissão

Q: o que acontece quando λ_{in} e λ'_{in} aumentam ?



Causas/custos do congestionamento: cenário 3





Um outro "custo" do congestionamento:

- Quando pacote descartado, a capacidade de transmissão "upstream" usada para este pacote foi desperdiçada!

Abordagens para o controle de congestionamento

2 abordagens amplas para o controle de congestionamento:

Controle de congestionamento fim-à-fim:

- Sem feedback explícito da rede
- Congestionamento inferido pelos end-systems através das perdas e atrasos observados
- Abordagem usada pelo TCP

Controle de congestionamento assistido pela rede (Network-assisted congestion control):

- roteadores provêem feedback para os end systems
  - bit único indicando congestionamento (SNA, DECbit, TCP/IP ECN, ATM)
  - Taxa explícita que emissor deve usar

Estudo de caso: Controle de congestionamento ABR da rede ATM

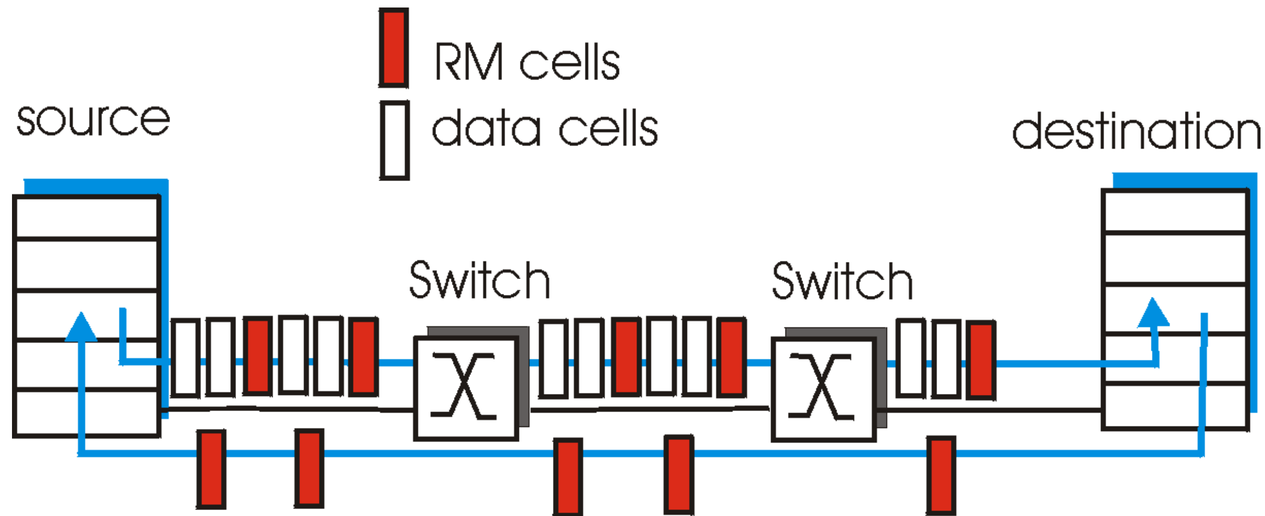
ABR: available bit rate:

- "serviço elástico"
- Se caminho emissor-receptor estiver "sub-utilizado" (underloaded):
  - Emissor deve usar banda disponível
- Se caminho emissor-receptor estiver congestionado:
  - Emissor reduz taxa para o menor valor garantido

Células RM (resource management):

- Enviadas pelo emissor, intercaladas com células de dados
- bits na célula RM cell setados pelos comutadores ("*network-assisted*")
  - Bit NI: nenhum incremento na taxa (mild congestion)
  - Bit CI: indicação de congestinamento
- Receptor retorna células RM ao emissor com os bits inalterados

Estudo de caso: Controle de congestionamento ABR da rede ATM



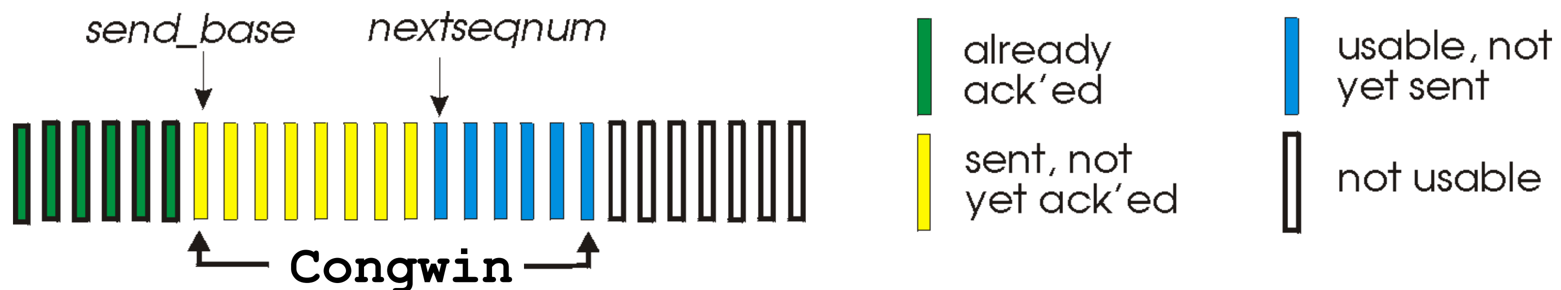
- Campo ER (explicit rate) de 2 bytes no cabeçalho da célula RM
  - Comutador congestionado pode reduzir valor ER na célula
  - Taxa do emissor ajustada para a menor taxa suportada pelo caminho até o receptor
- Bit EFCI em células de dados: setado para 1 em comutadores congestionados
  - Se célula de dados precedendo célula RM possui o bit RFCI setado, emissor seta bit CI na célula RM retornada

Resumo do Módulo 3

- 3.1 Serviços da camada transporte
- 3.2 Multiplexação e demultiplexão
- 3.3 transporte não-orientado à conexão: UDP
- 3.4 Princípios da transferência confiável de dados
- 3.5 transporte orientado à conexão: TCP
  - Estrutura do segmento
  - Tranferência de dados confiável
  - Controle de fluxo
  - Gerenciamento de conexão
- 3.6 Princípios do controle de congestionamento
- 3.7 controle de congestionamento TCP

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:

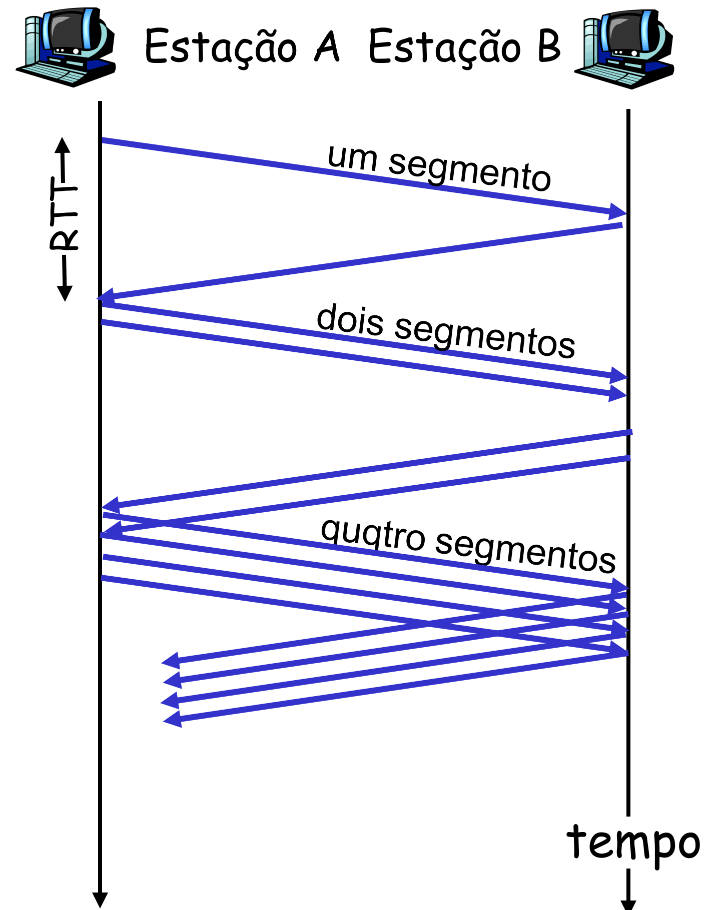
$$\text{throughput} = \frac{w * MSS}{RTT} \text{ Bytes/sec}$$

TCP: Partida lenta (slow start)

Algoritmo Partida Lenta

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

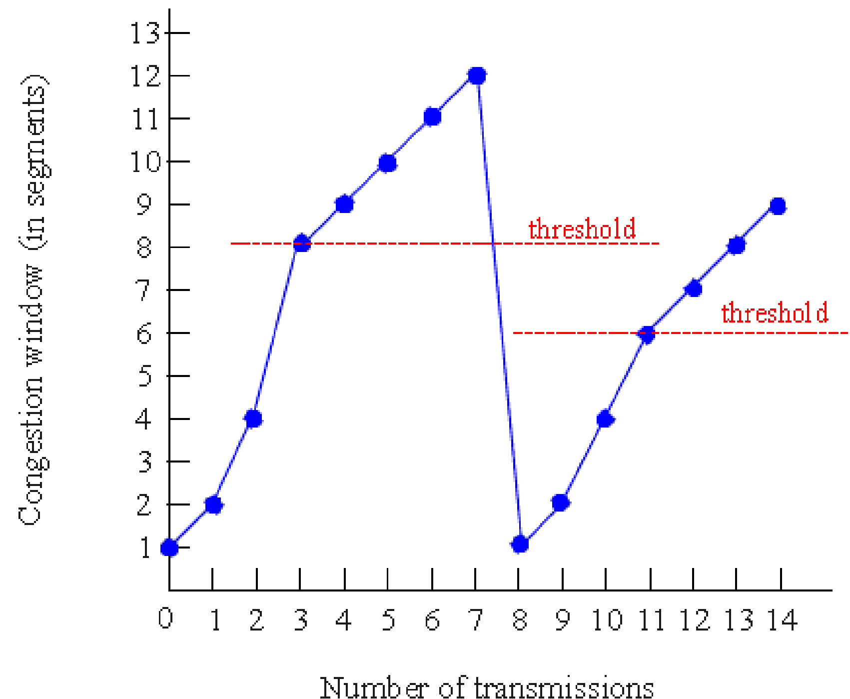
- aumento exponencial (por RTT) no tamanho da janela (não muito lenta!)
- evento de perda: temporizador (Tahoe TCP) e/ou três ACKs duplicados (Reno TCP)



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]

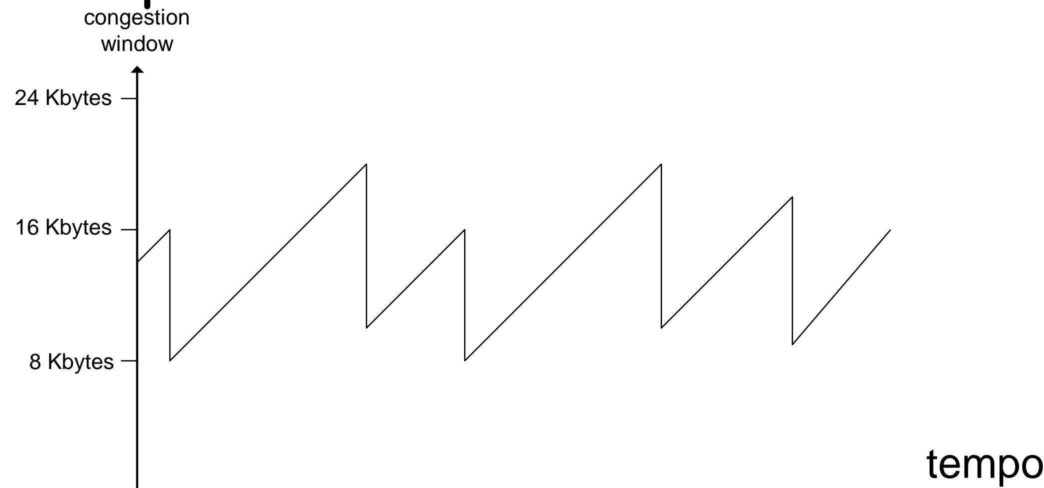


1: TCP Reno pula partida lenta (recuperação rápida) depois de três ACKs duplicados

Controle de congestionamento TCP: additive increase, multiplicative decrease (AIMD)

- *Abordagem:* aumentar taxa de transmissão (tamanho da janela), sondar bw ainda utilizável, até ocorrência de perda de pacote
  - *Aumento aditivo:* aumenta **CongWin** de 1 MSS a cada RTT até perda ser detectada
  - *redução multiplicativa*: reduz **CongWin** pela metade após detectar perda

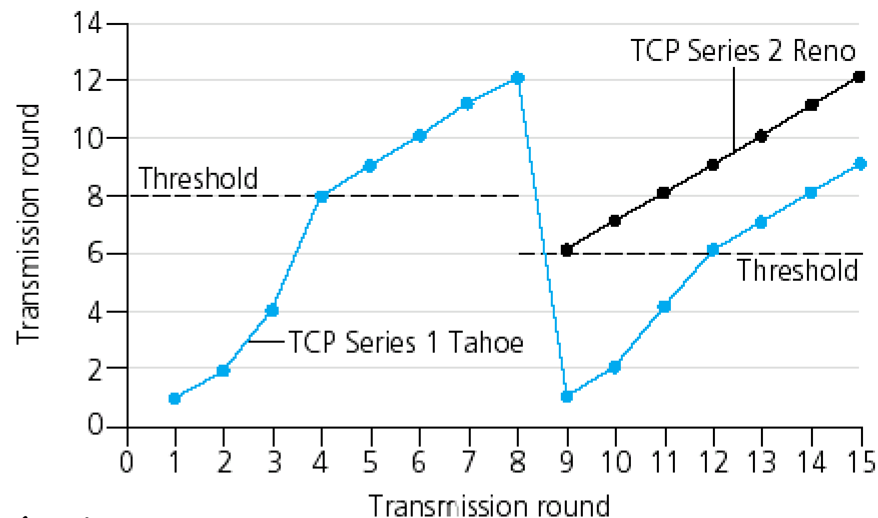
Comportamento dente-de-serra: Sondagem de bw



Refinamento

Q: Quando o aumento exponencial deve parar para se tornar linear?

A: Quando **CongWin** assume 1/2 do seu valor antes do timeout.



Implementação:

- Threshold (limiar) Variável
- No evento de perda, Threshold é setado para 1/2 da CongWin. Valor da janela antes do evento de perda

Refinamento: inferindo perdas

- Após 3 ACKs duplicados:
  - `CongWin` é reduzida pela metade
  - Janela cresce lineamente
- Mas após evento de timeout:
  - `CongWin` é setada para 1 MSS;
  - Janela então cresce exponencialmente
  - Até um threshold (limiar), e então volta a crescer linearmente

Filosofia:

- 3 ACKs dup. indica rede capaz de entregar alguns segmentos
- timeout indica um cenário "mais alarmante" de congestionamento

Sumário: Controle de Congestionamento

- Quando `CongWin` está abaixo de `Threshold`, emissor está na fase slow-start, janela cresce exponencialmente.
- Quando `CongWin` está acima de `Threshold`, emissor está na fase congestion-avoidance, janela cresce linearmente.
- Quando três ACKs duplicados ocorrem, `Threshold` é setado para `CongWin/2` e `CongWin` é setado para `Threshold`.
- Quando timeout ocorre, `Threshold` é setado para `CongWin/2` e `CongWin` é setado para 1 MSS.

Controle de congestionamento TCP

estado	Evento	Ação do emissor TCP	Comentário
Slow Start (SS)	ACK recebido para dado ainda não confirmado	CongWin = CongWin + MSS, If (CongWin > Threshold) estado atual setado para "Congestion Avoidance"	Dobra CongWin a cada RTT
Congestion Avoidance (CA)	ACK recebido para dado ainda não confirmado	CongWin = CongWin+MSS * (MSS/CongWin)	Aumento aditivo, CongWin aumenta de 1 MSS a cada RTT
SS ou CA	Perda detectada através de 3 ACKs duplicados	Threshold = CongWin/2, CongWin = Threshold, Estado atual setado para "Congestion Avoidance"	Recuperação rápida, implementando redução multiplicativa. CongWin não cairá abaixo de 1 MSS.
SS ou CA	Timeout	Threshold = CongWin/2, CongWin = 1 MSS, Estado atual setado para "Slow Start"	Entra no slow start
SS ou CA	ACK duplicado	Incrementar contador de ACK duplicado para o segmento sendo confirmado	CongWin e Threshold não mudam

Vazão do TCP

- Qual a vazão média do TCP em função do tamanho da janela e do RTT?
  - Ignore slow start
- Seja W o tamanho da janela quando uma perda ocorre.
- Quando janela é W, vazão é W/RTT
- Logo após perda, janela cai à W/2, e vazão à W/2RTT.
- Vazão média: .75 W/RTT

Características do TCP

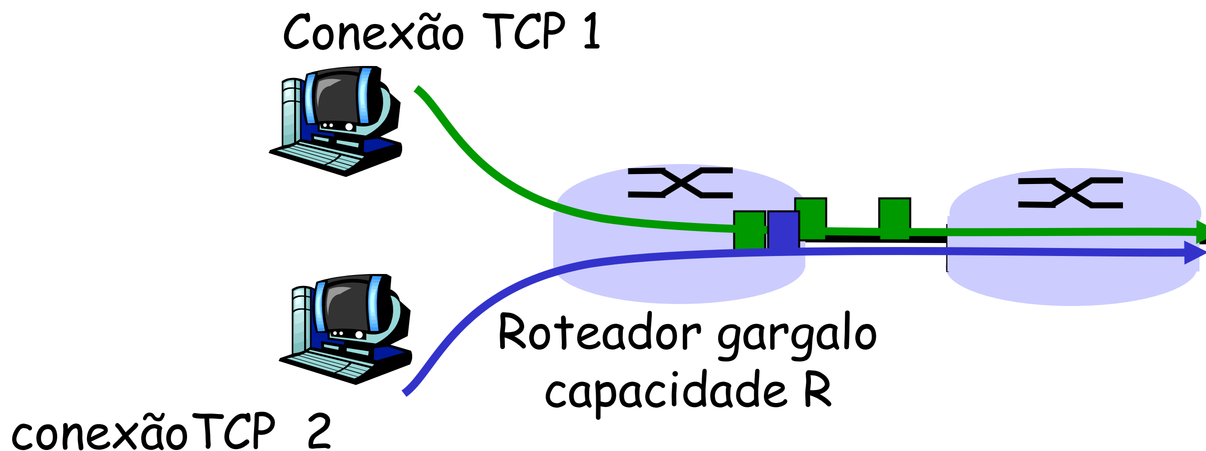
- Exemplo: segmentos de 1500 bytes, RTT = 100ms, deseja-se 10 Gbps de vazão
- Requer janela W = 83,333 segmentos sendo enviados
- Vazão em termos da taxa de perdas (L):

$$\frac{1.22 \cdot MSS}{RTT\sqrt{L}}$$

- ➜ $L = 2 \cdot 10^{-10}$ *Uau!*
- Novas versões do TCP para altas taxas de envio necessárias!

Justiça no TCP (TCP Fairness)

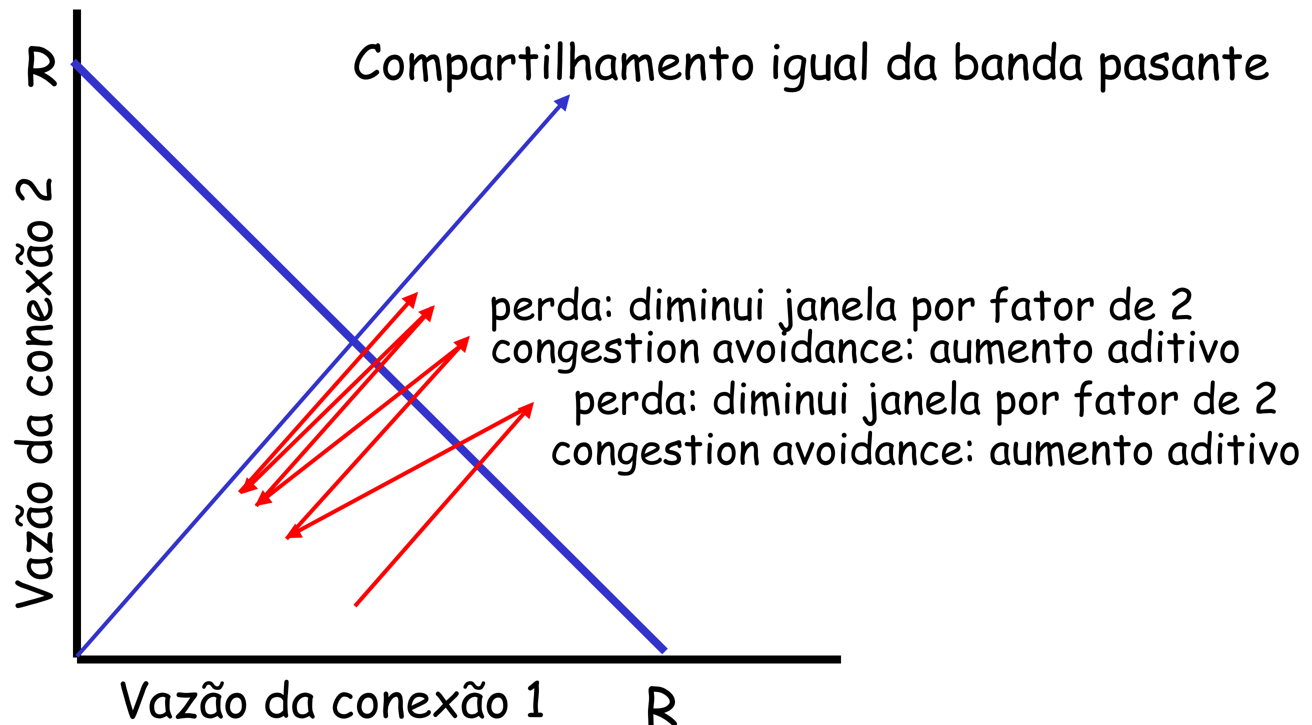
Objetivo da justiça: se K sessões TCP compartilham um mesmo enlace (de gargalo) de banda passante R, cada uma deve ter uma taxa média de R/K



Por que TCP é justo?

2 sessões competindo:

- Aumento aditivo - saltos de 1 (vazão aumenta)
- Redução multiplicativa dimuni vazão proporcionalmente



Justiça (mais)

Justiça e o UDP

- Apps multimídia geralmente não usam TCP
  - Para não restringir taxa devido ao controle de congestionamento
- Em vez do TCP, as apps multimídia usam UDP:
  - áudio/vídeo injetados à taxas constantes, toleram perdas de pacotes
- Área de pesquisa: TCP friendly (protocolos amigos do TCP)

Justiça e conexões TCP paralelas

- Nada previne apps de abrirem conexões paralelas entre dois hosts.
- Web browsers fazem isto
- Exemplo: enlace de taxa R suportando 9 conexões;
  - Nova app requisita 1 conex.TCP, recece taxa R/10
  - Nova app requisita 11 conex.TCPs, recebe R/2 !

Modelagem de Atraso

Q: Quanto tempo leva para receber um objeto de um servidor Web após o envio da requisição ?

Ignorando congestionamento, atraso é influenciado por:

- Estabelecimento de conexão TCP
- Atraso de transmissão de dados
- slow start (partida lenta)

Notação, hipóteses:

- Assuma 1 enlace de taxa R entre cliente e servidor
- S: MSS (bits)
- O: tamanho do objeto (bits)
- sem retransmissões (sem perdas, não há pkts corrompidos)

Tamanho da janela:

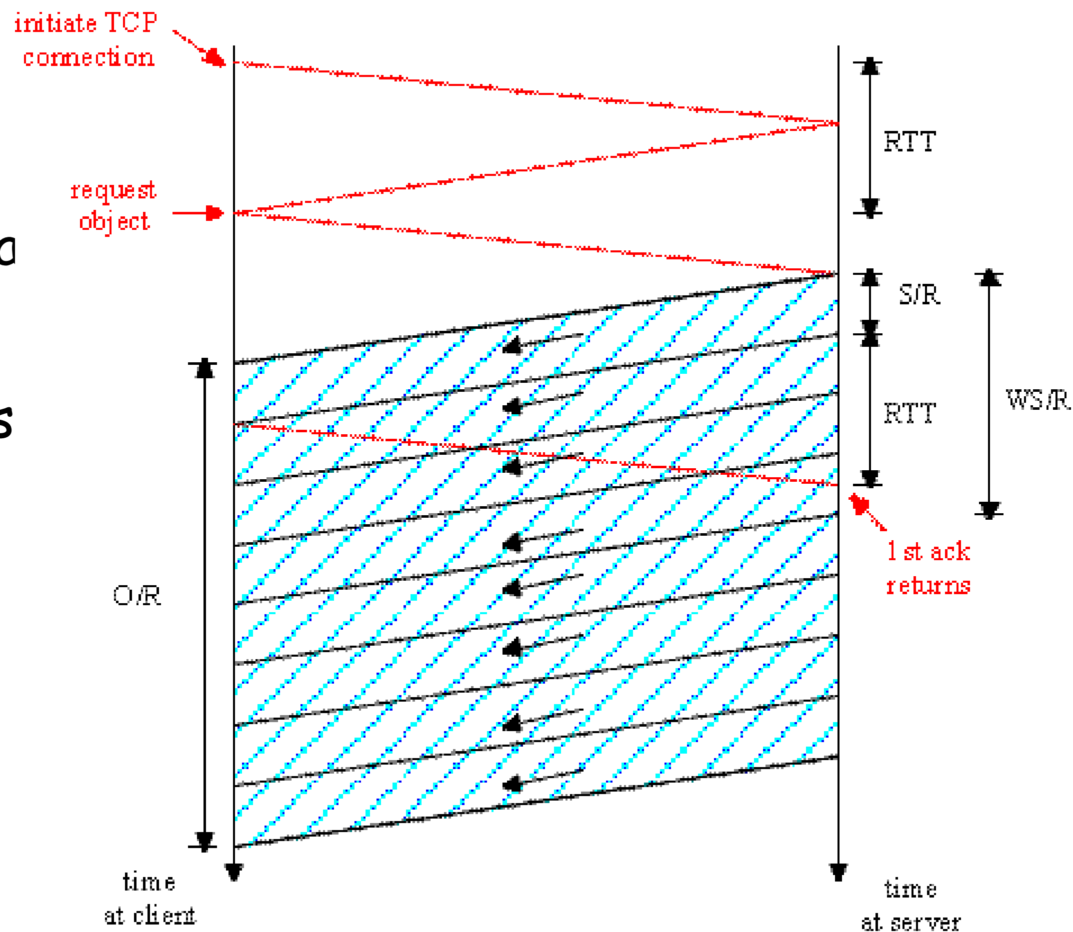
- Primeiro assuma: janela de congestionamento fixa, W segmentos
- Em seguida, assuma janela dinâmica com modelagem do slow start

Janela de congestionamento fixa (1)

Primeiro caso:

WS/R > RTT + S/R: ACK pa
primeiro segmento na janela retorna antes da transmissão de todos os segmentos da janela

delay = 2RTT + O/R

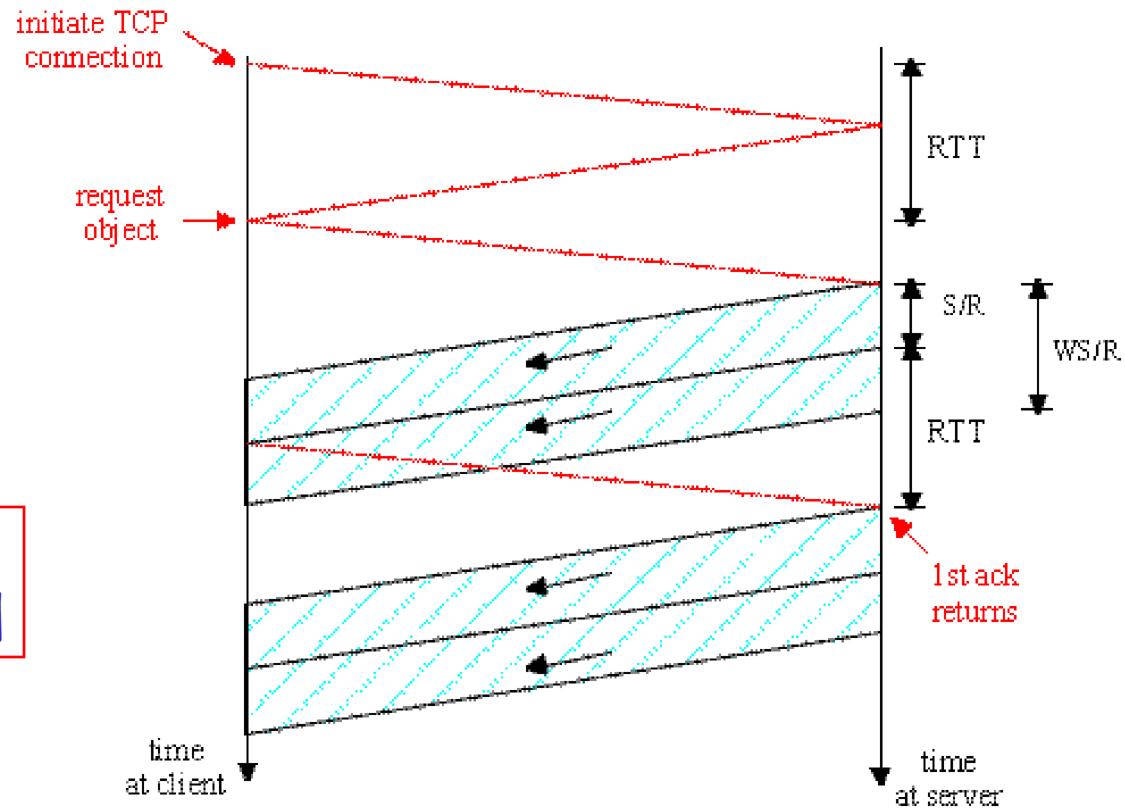


Janela de congestionamento fixa(2)

Segundo caso:

- WS/R < RTT + S/R: aguarda por ACK após enviar todos os segmentos previstos na janela

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



Modelagem do Atraso no TCP: Slow Start (1)

Agora suponha que janela cresce de acordo com o slow start

O atraso para um objeto será :

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

onde P é o número de vezes que o TCP fica esperando no servidor:

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

-onde Q é o número de vezes que o servidor espera caso o objeto fosse de tamanho infinito.

- e K é o número necessário de janelas para transmitir o objeto.

Modelagem do Atraso no TCP: Slow Start (2)

Componentes do atraso:

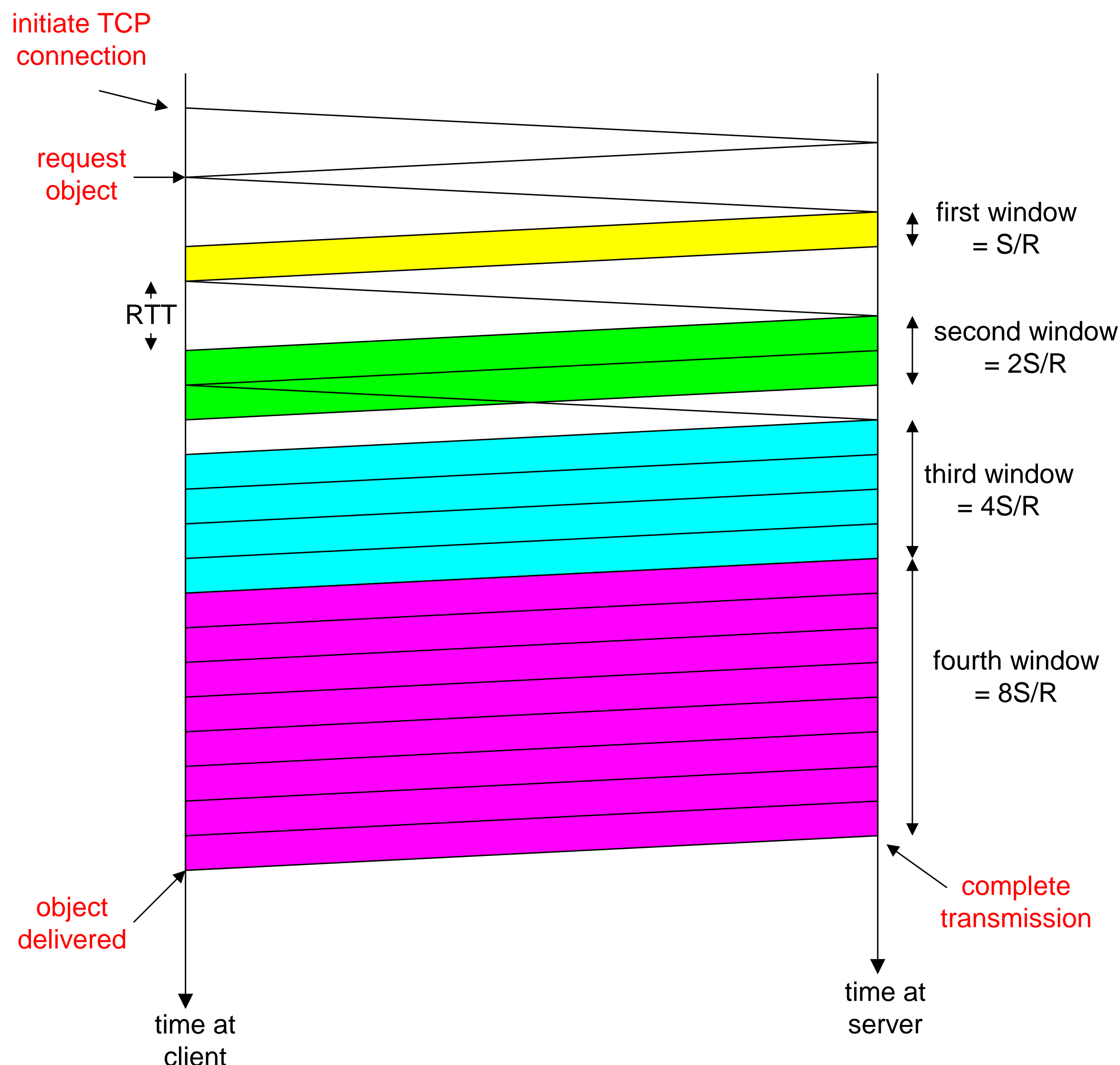
- 2 RTT para estabelecimento de conex. e requisição
- O/R para transmitir objeo
- tempo que o servidor espera devido ao slow start

Servidor espera:
P = min{K-1,Q} vezes

Exemplo:

- O/S = 15 segmentos
- K = 4 janelas
- Q = 2
- P = min{K-1,Q} = 2

Servidor espera P=2 vezes



Modelagem do Atraso no TCP (3)

$$\frac{S}{R} + RTT =$$ Tempo desde o início do envio de um segmento até o recebimento do ack

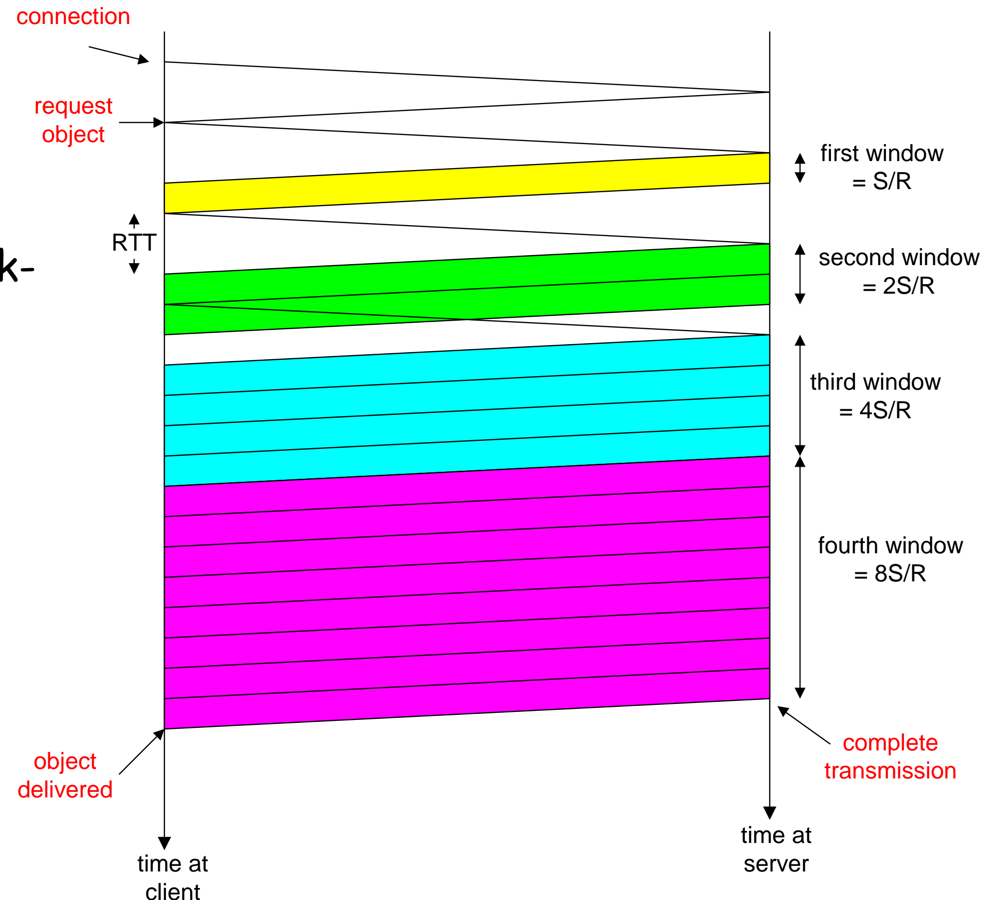
$$2^{k-1}\frac{S}{R} =$$ Tempo para transmitir a k-ésima janela

$$\left[\frac{S}{R} + RTT - 2^{k-1}\frac{S}{R}\right]^{+} =$$ Tempo de espera após k-ésima janela

$$\text{atraso} = \frac{O}{R} + 2RTT + \sum_{p=1}^{P} \text{Tempo de espera}_p$$

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

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



Modelagem do Atraso no TCP (4)

Lembre que K = número de janelas para transmitir objeto
Como calculamos K ?

$$
\begin{aligned}
K &= \min\{k : 2^0 S + 2^1 S + \cdots + 2^{k-1} S \geq O\} \\
&= \min\{k : 2^0 + 2^1 + \cdots + 2^{k-1} \geq O / S\} \\
&= \min\{k : 2^k - 1 \geq \frac{O}{S}\} \\
&= \min\{k : k \geq \log_2(\frac{O}{S} + 1)\} \\
&= \left\lceil \log_2(\frac{O}{S} + 1) \right\rceil
\end{aligned}
$$

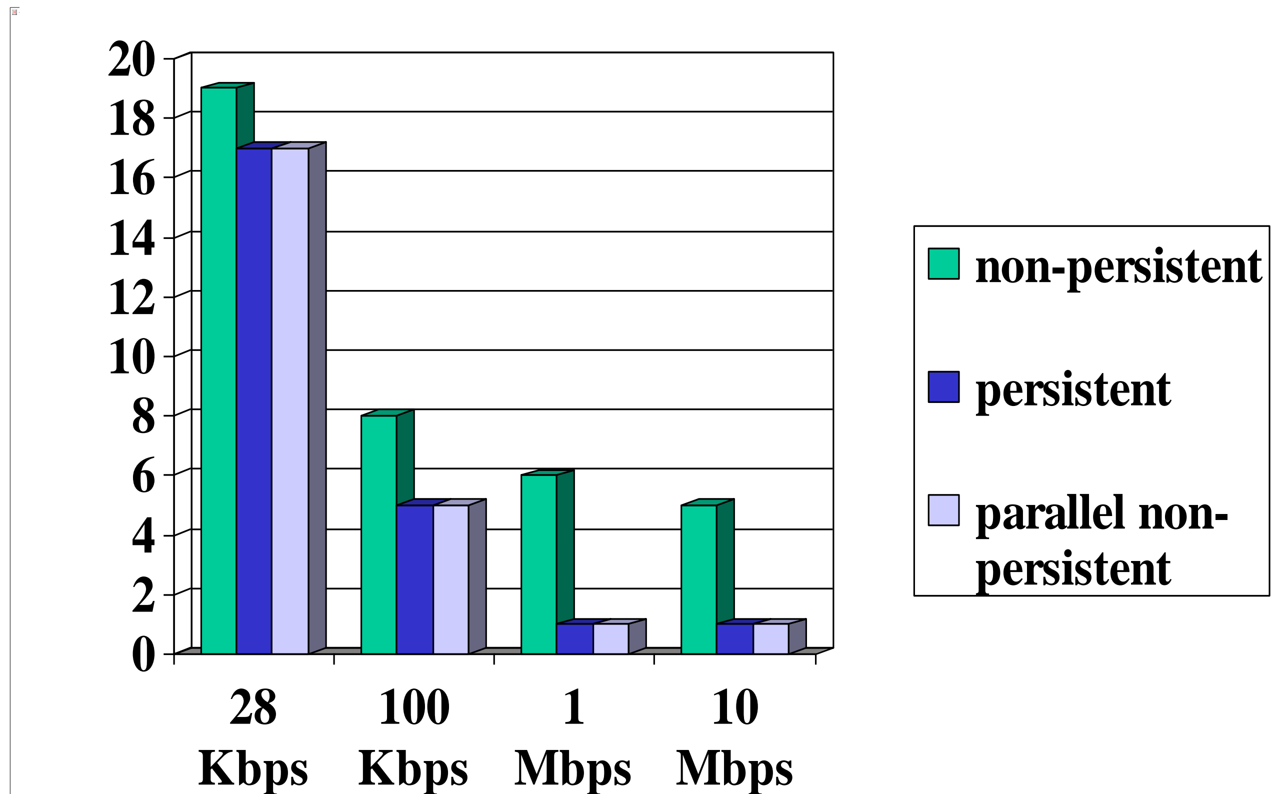
Cálculo de Q, número de “esperas” para objeto de tamanho infinito, é similar (veja o livro).

Modelagem do HTTP

- Assuma que uma página Web consista em:
  - *1* página-base em HTML (de tamanho *O* bits)
  - *M* imagens (cada uma de tamanho *O* bits)
- HTTP não-persistente:
  - *M+1* conexões TCP em série
  - *Tempo de Resposta = (M+1)O/R + (M+1)2RTT + soma de tempos de espera*
- HTTP Persistente:
  - *2 RTT para requição e recebimento da página-base* HTML
  - *1 RTT* para requisição e início de recebimento de M images
  - *Tempo de Resposta = (M+1)O/R + 3RTT + soma de tempos de espera*
- HTTP Não-persistente com X conexões paralelas
  - Suponha que M/X seja inteiro.
  - 1 conexão TCP para o aqrquivo de base
  - M/X conjuntos de conexões paralelas para imagens.
  - *Tempo de Resposta = (M+1)O/R + (M/X + 1)2RTT + soma de tempos de espera*

Tempo de resposta do HTTP (em seg.)

RTT = 100 ms, O = 5 Kbytes, M=10 e X=5

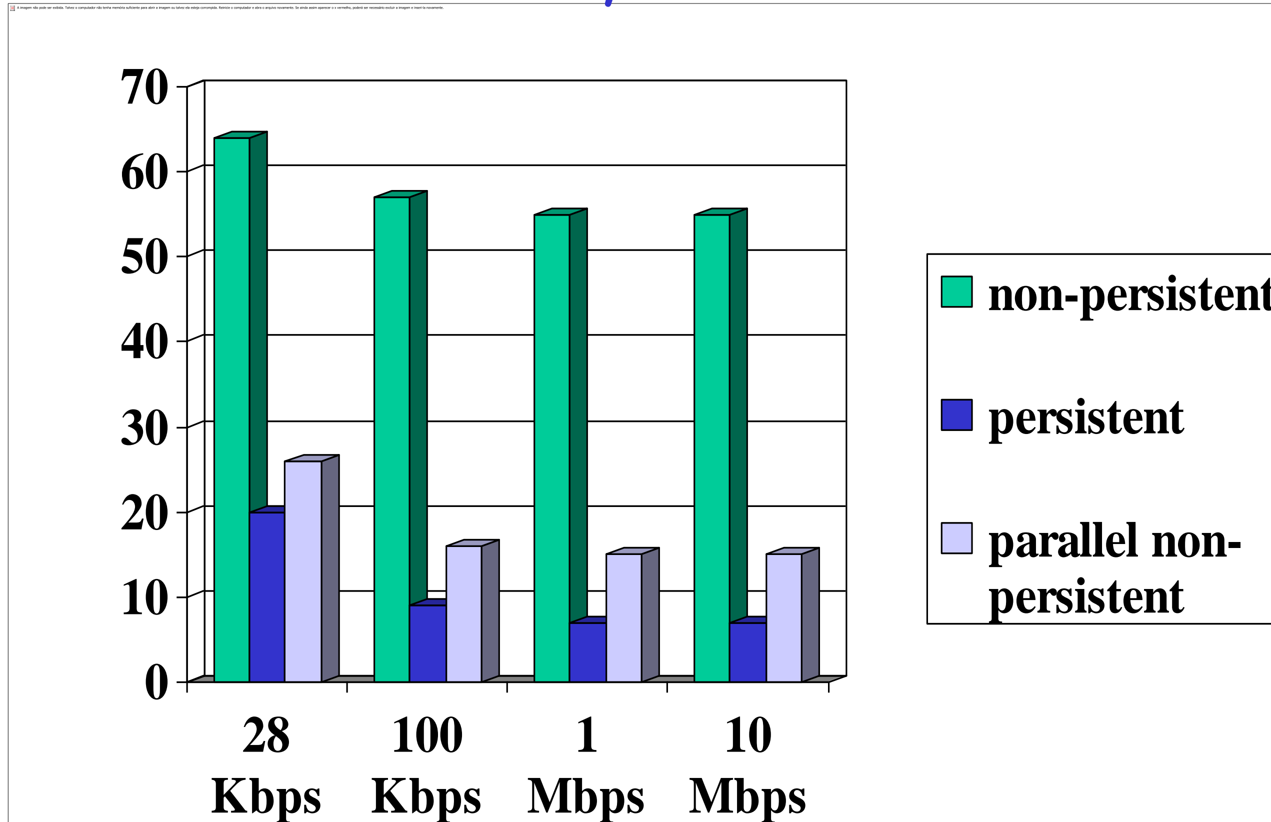


Para baixa BW, tempo de conexão e de resposta dominado pelo tempo de transmissão.

Conexões persistentes somente provêem pequena melhoria com relação ao uso de conexões paralelas.

Tempo de resposta do HTTP (em seg.)

RTT =1 sec, O = 5 Kbytes, M=10 and X=5



Para altos RTTs, tempo de resposta é dominado pelos atrasos de estabelecimento de conexão & do slow start. Conexões persistentes proporcionam agora melhoria importante de desempenho: particularmente em redes com alto produto atraso•bw.

Módulo 3: Sumário

- Princípios por trás dos serviços da camada transporte:
  - multiplexação, demultiplexação
  - Transferência confiável de dados
  - Controle de fluxo
  - Controle de congestionamento
- Instanciação e implementação na Internet
  - UDP
  - TCP

A seguir:

- Deixando a extremidade da rede (camadas aplicação, transporte)
- e entrando no núcleo da rede