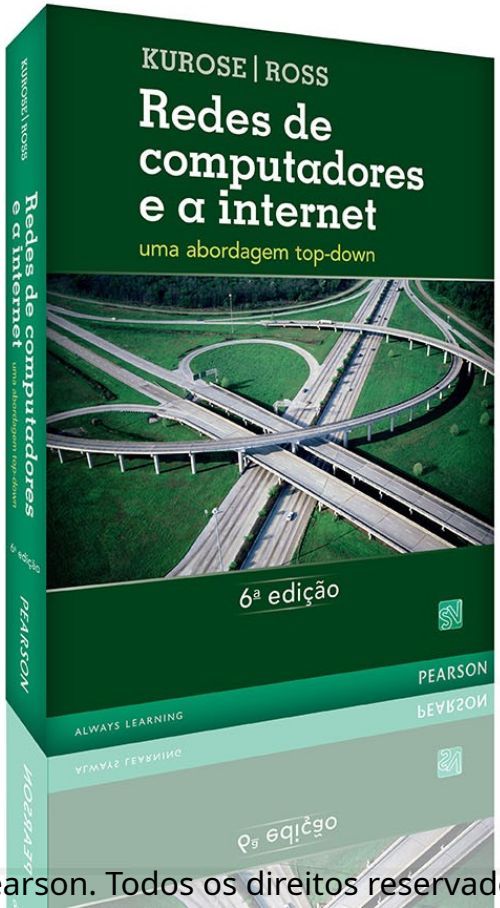
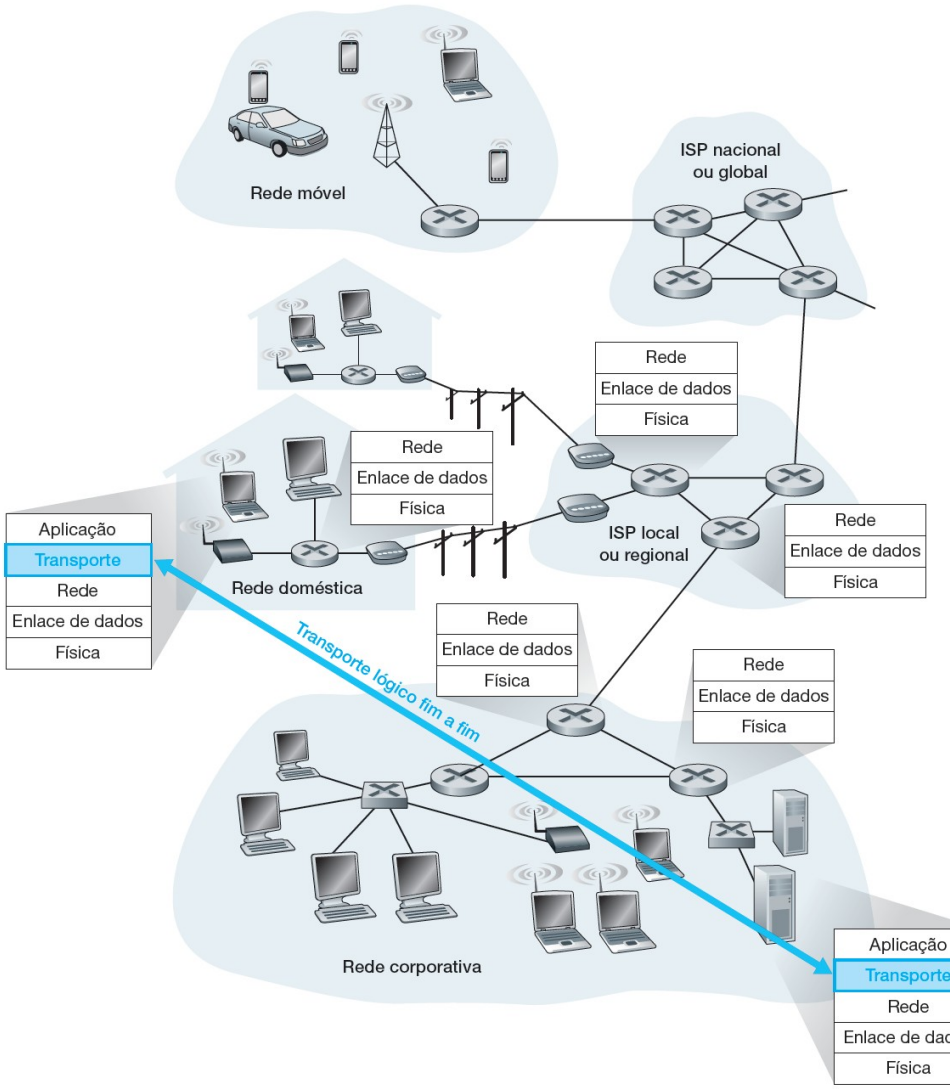
Capítulo 3

Camada de transporte



Introdução e serviços de camada de transporte



- A camada de transporte fornece comunicação lógica (e não física) diretamente entre processos de aplicações.
- Comunicação lógica: tudo se passa como se os processos dos hospedeiros estivessem conectados diretamente

Relação entre as camadas de transporte e de rede

- Um protocolo de camada de transporte fornece comunicação lógica entre *processos* que rodam em hospedeiros diferentes. **Segmentos** da camada de transporte.
- Um protocolo de camada de rede fornece comunicação lógica entre *hospedeiros*. **Datagrama** da camada de rede.
- Uma rede de computadores pode disponibilizar vários protocolos de transporte.
- Os serviços que um protocolo de transporte pode fornecer são muitas vezes limitados pelo modelo de serviço do protocolo subjacente da camada de rede.

Visão geral da camada de transporte na Internet

- A responsabilidade fundamental do UDP (*User Datagram Protocol*) e do TCP (*Transmission Control Protocol*) é ampliar o serviço de entrega IP entre dois sistemas finais para um serviço de entrega entre dois processos que rodam nos sistemas finais.

- A ampliação da entrega hospedeiro a hospedeiro para entrega processo a processo é denominada **multiplexação/demultiplexação de camada de transporte**.

- O UDP e o TCP também fornecem verificação de integridade ao incluir campos de detecção de erros nos cabeçalhos de seus segmentos.

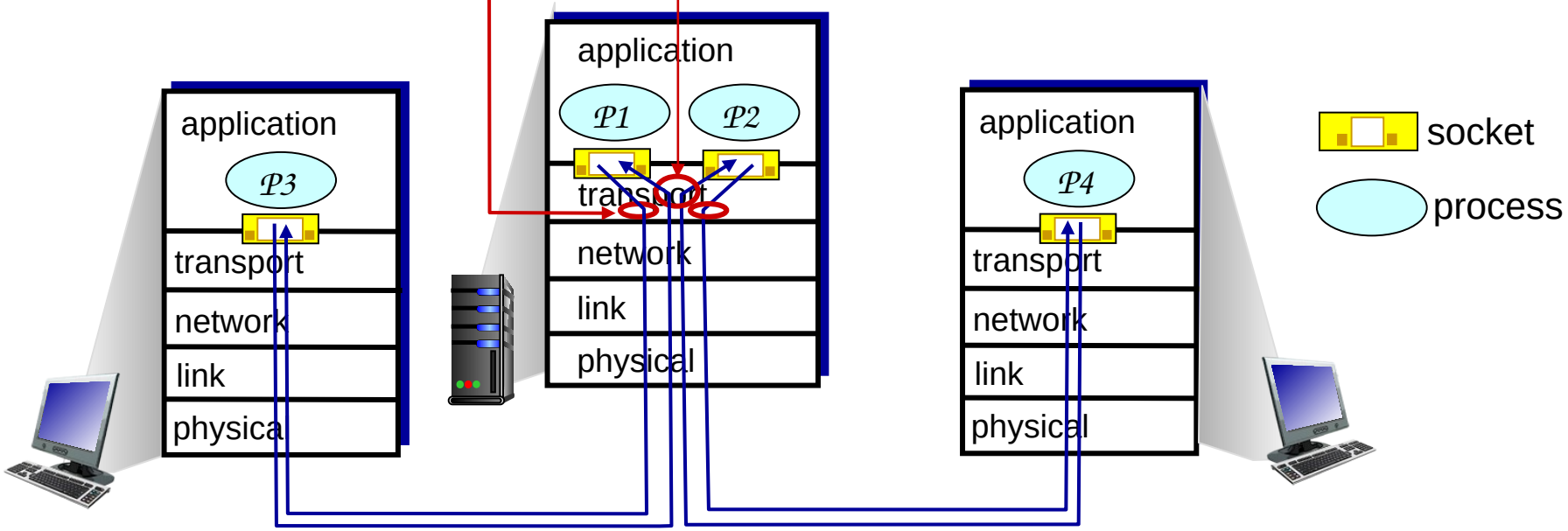
Multiplexing/demultiplexing

multiplexing at sender:

handle data from multiple sockets, add transport header (later used for demultiplexing)

demultiplexing at receiver:

use header info to deliver received segments to correct socket



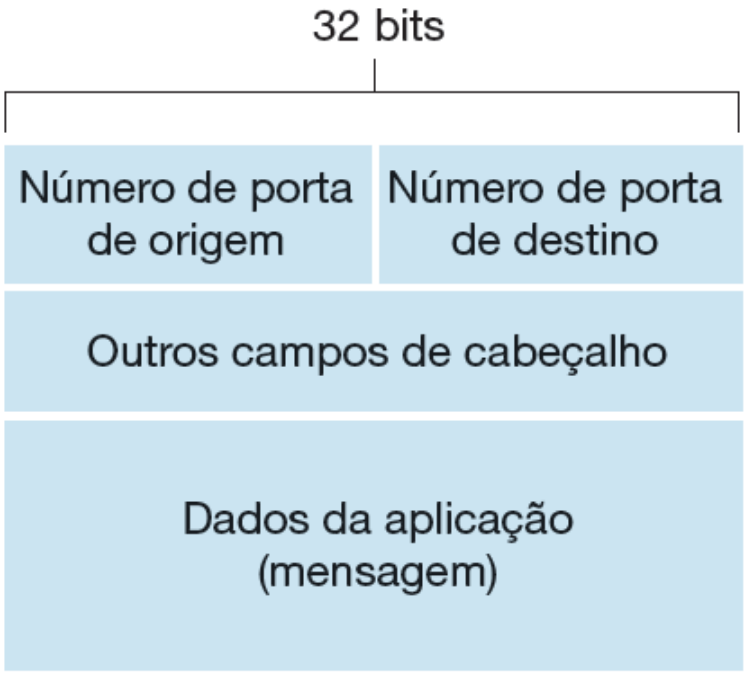
Multiplexação e demultiplexação

- A tarefa de entregar os dados contidos em um segmento da camada de transporte ao *socket* correto é denominada **demultiplexação**.

- O trabalho de reunir, no hospedeiro de origem, partes de dados provenientes de diferentes *sockets*, encapsular cada parte de dados com informações de cabeçalho para criar segmentos, e passar esses segmentos para a camada de rede é denominada **multiplexação**.

Multiplexação e demultiplexação

- Campos de número de porta de origem e de destino em um segmento de camada de transporte:



Demultiplexação não orientada para conexão

- cria *sockets* com número de porta automático:

```
clietSocket =  socket(AF_INET,
   SOCK_DGRAM);
```

- Alternativamente podemos fixar a porta, acrescentando:

```
clientSocket.bind(('', 12535));
```

- *socket* UDP identificado por tupla de dois elementos:

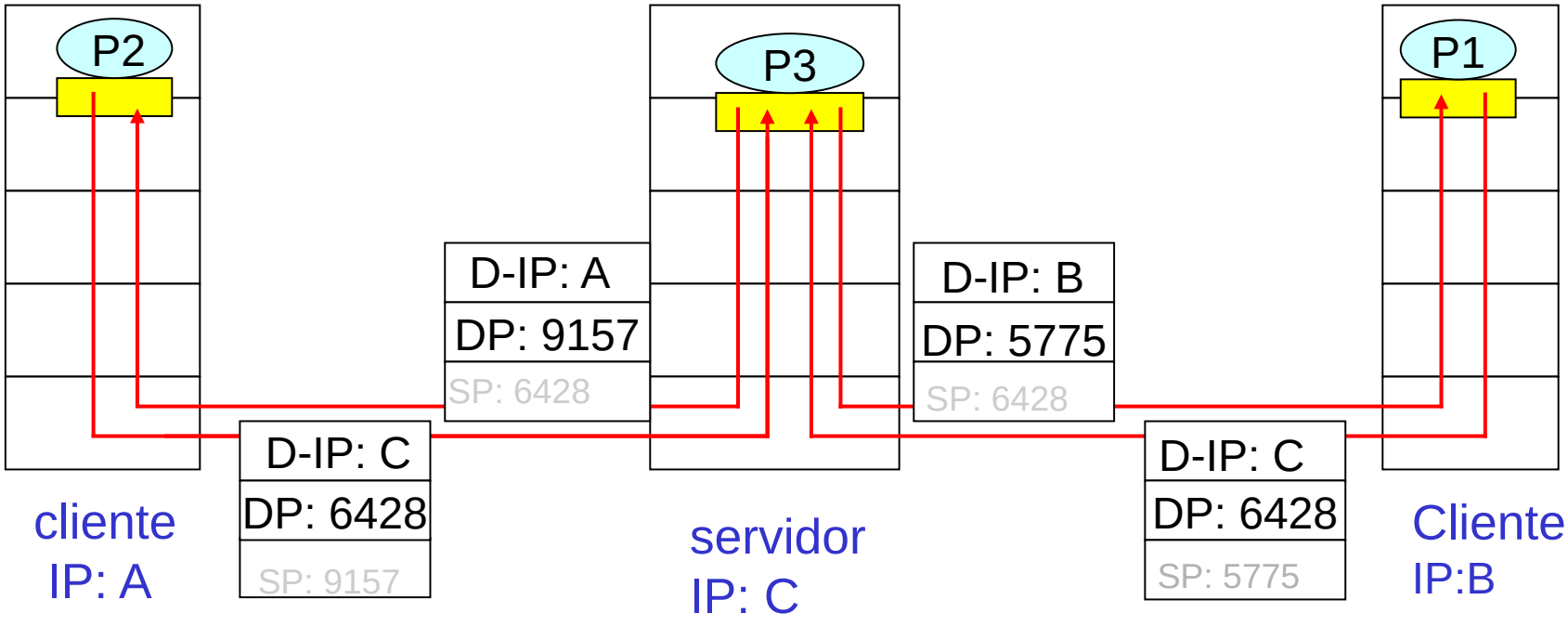
(endereço IP destino, número porta destino)

- quando hospedeiro recebe segmento UDP:
  - verifica número de porta de destino no segmento
  - direciona segmento UDP para *socket* com esse número de porta
- datagramas IP com diferentes endereços IP de origem e/ou números de porta de origem direcionados para o mesmo *socket*

Demultiplexação não orientada para conexão

socket UDP identificado por tupla de dois elementos:

(endereço IP destino, número porta destino);

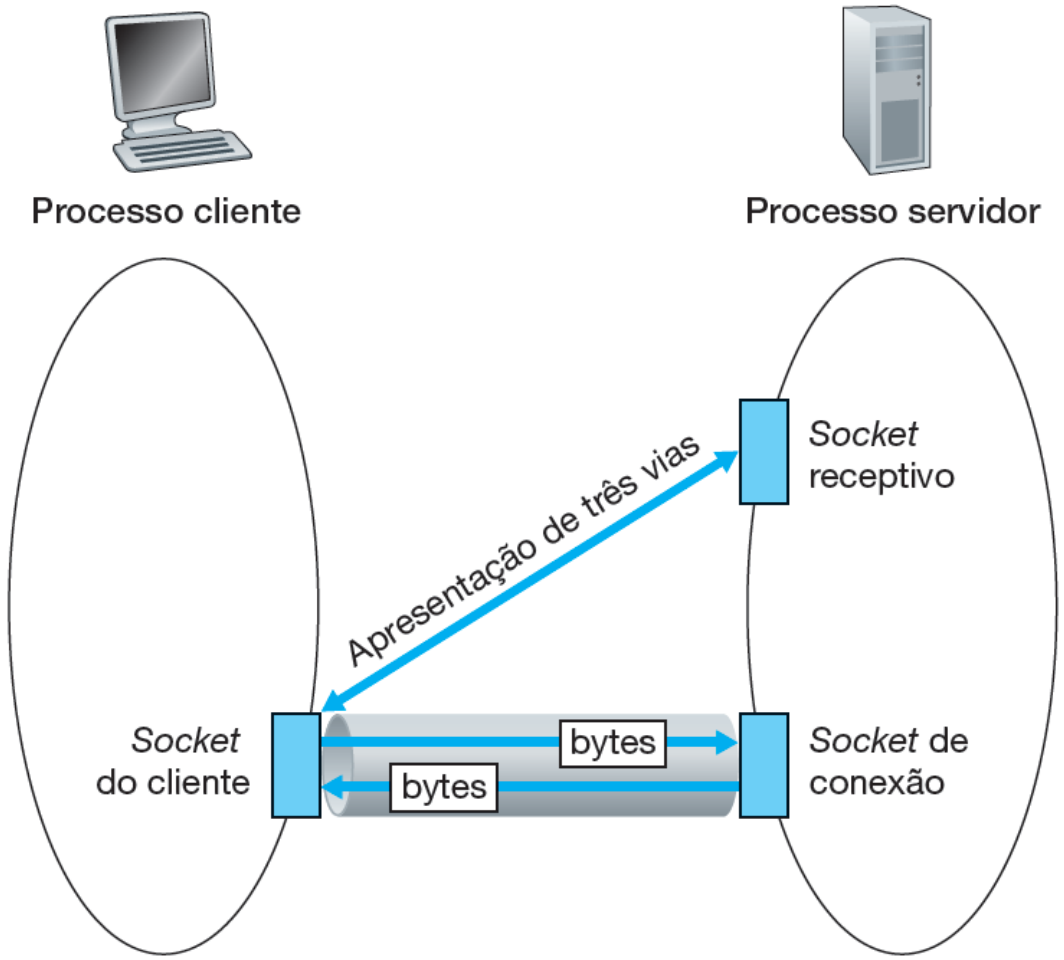


SP oferece “endereço de retorno”, caso necessário

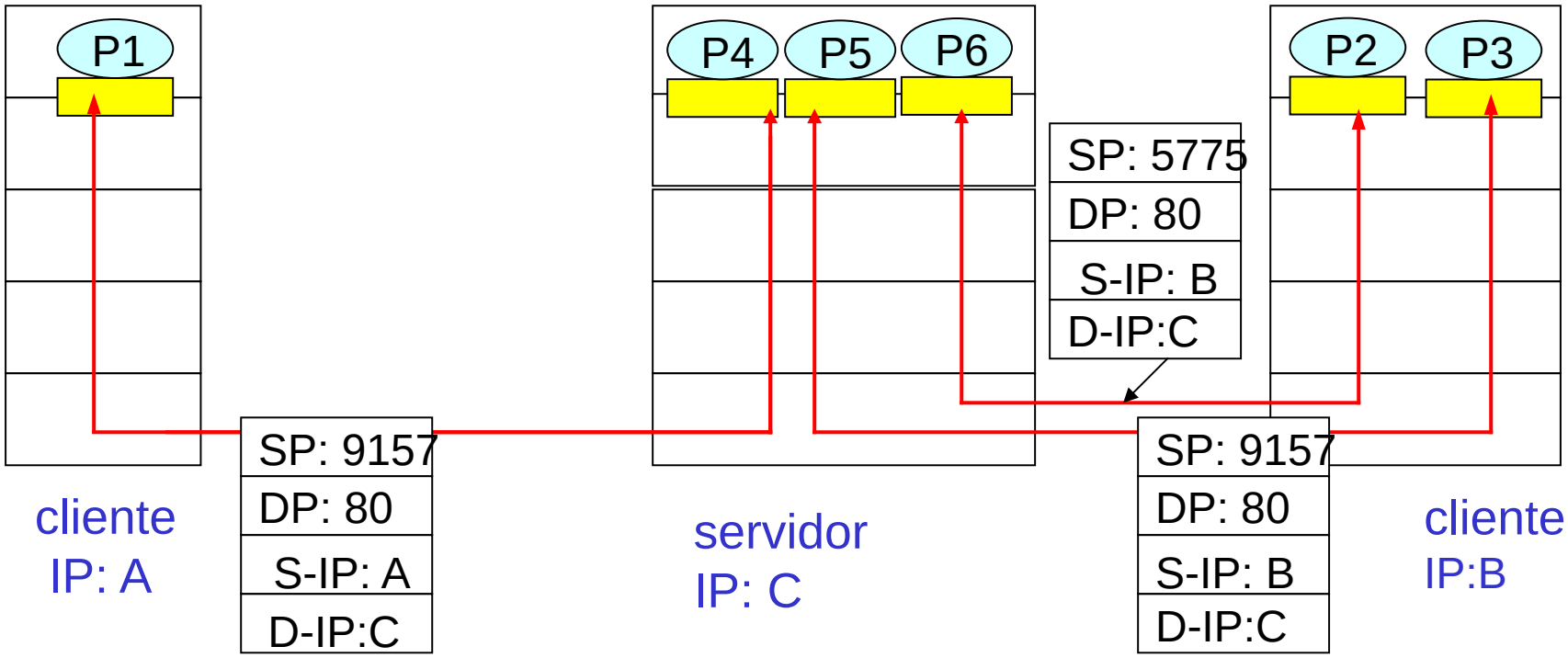
Demultiplexação orientada para conexão

- *socket* TCP identificado por tupla de 4 elementos:
  - endereço IP de origem
  - número de porta de origem
  - endereço IP de destino
  - número de porta de destino
- hospedeiro destinatário usa todos os quatro valores para direcionar segmento ao *socket* apropriado
- hospedeiro servidor pode admitir muitos *sockets* TCP simultâneos:
  - cada *socket* identificado por sua própria tupla de 4
- servidores Web têm diferentes *sockets* para cada cliente conectando
  - HTTP não persistente terá diferentes *sockets* para cada requisição

Demultiplexação orientada para conexão



Demultiplexação orientada para conexão



Transporte não orientado para conexão: UDP

- O UDP, definido no [RFC 768], faz apenas quase tão pouco quanto um protocolo de transporte pode fazer.
- À parte sua função de multiplexação/demultiplexação e de alguma verificação de erros simples, ele nada adiciona ao IP.
- Se o desenvolvedor de aplicação escolher o UDP, em vez do TCP, a aplicação estará “falando” quase diretamente com o IP.
- O UDP é *não orientado para conexão*. Não há a apresentação entre as entidades remetentes e destinatárias.

Por que usa-se UDP

- Melhor controle no nível de aplicação sobre quais dados são enviados e quando: menor atraso.
- Não há estabelecimento de conexão: menor atraso.
- Não há estados de conexão: *buffers*, parâmetros de controle de congestionamento, número de sequência. Menor "custo".
- Pequeno excesso de cabeçalho de pacotes: **8 bytes** de excesso, TCP 20 bytes. Menor "custo".
- Sem controle de congestionamento do TCP ==> isso pode implicar em "colapso" na rede.
- É possível uma aplicação ter transferência confiável de dados usando UDP?

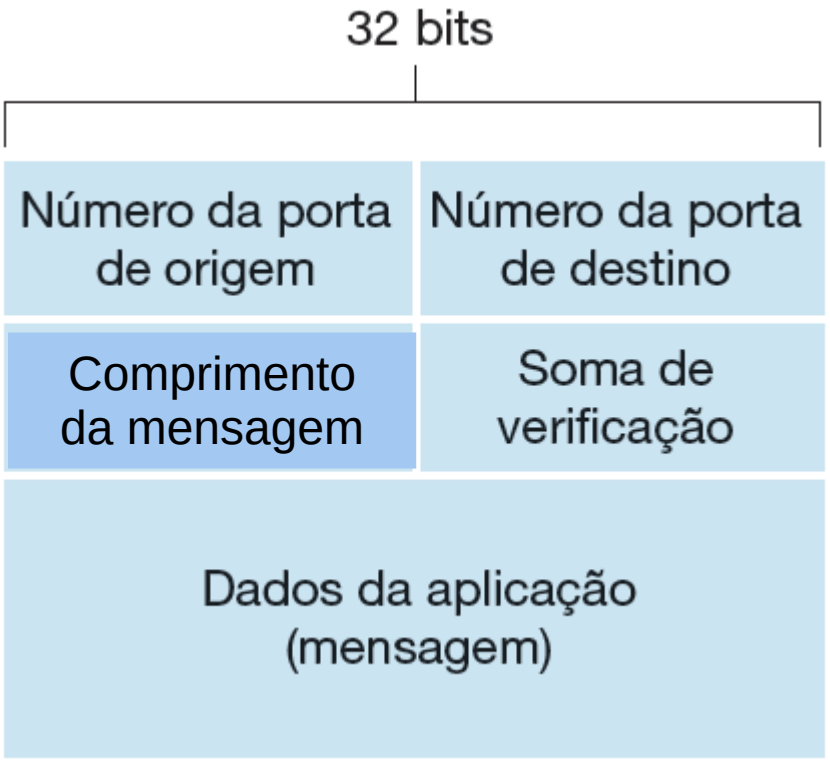
Transporte não orientado para conexão: UDP

- Aplicações populares da Internet e seus protocolos de transporte subjacentes:

Aplicação	Protocolo da camada de aplicação	Protocolo de transporte subjacente
Correio eletrônico	SMTP	TCP
Acesso a terminal remoto	Telnet	TCP
Web	HTTP	TCP
Transferência de arquivo	FTP	TCP
Servidor de arquivo remoto	NFS	Tipicamente UDP
Recepção de multimídia	Tipicamente proprietário	UDP ou TCP
Telefonia por Internet	Tipicamente proprietário	UDP ou TCP
Gerenciamento de rede	SNMP	Tipicamente UDP
Protocolo de roteamento	RIP	Tipicamente UDP
Tradução de nome	DNS	Tipicamente UDP

Estrutura do segmento UDP

- Estrutura do segmento UDP



Soma de verificação (*checksum*) UDP

- A soma de verificação (*checksum*) UDP serve para detectar erros em todo o segmento.

- Suponha que tenhamos as seguintes três palavras de 16 bits:

```
0110011001100000
0101010101010101
1000111100001100
```

- A soma das duas primeiras é:

```
 0110011001100000
 0101010101010101
------------------
 1011101110110101
```

- Adicionando a terceira palavra à soma anterior, temos:

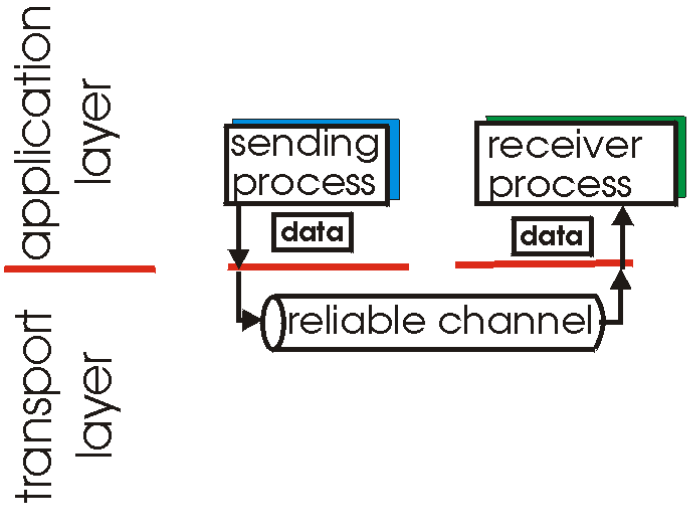
```
 1011101110110101
 1000111100001100
------------------
 0100101011000010
```

Soma de verificação UDP

- Note que a última adição teve um “vai um” no bit mais significativo que foi somado ao bit menos significativo.
- Complemento de 1: 0 ==> 1, 1 ==> 0
- Fazendo o complemento de 1 da soma final: 1011010100111101. Esta é a soma de verificação.
- No destinatário todas as 4 palavras recebidas de 16 bits são somadas: 1111... ==> Pacote correto; algum bit em 0 ==> algum erro foi introduzido no pacote.
- Por que soma de verificação no UDP?
  - Algumas camadas de enlace, por exemplo Ethernet, fornecem soma de verificação,
  - mas nem todas e, além disso, um roteador poderia introduzir algum erro.
  - UDP verifica erros fim a fim.

Principles of reliable data transfer

- important in application, transport, link layers
  - top-10 list of important networking topics!

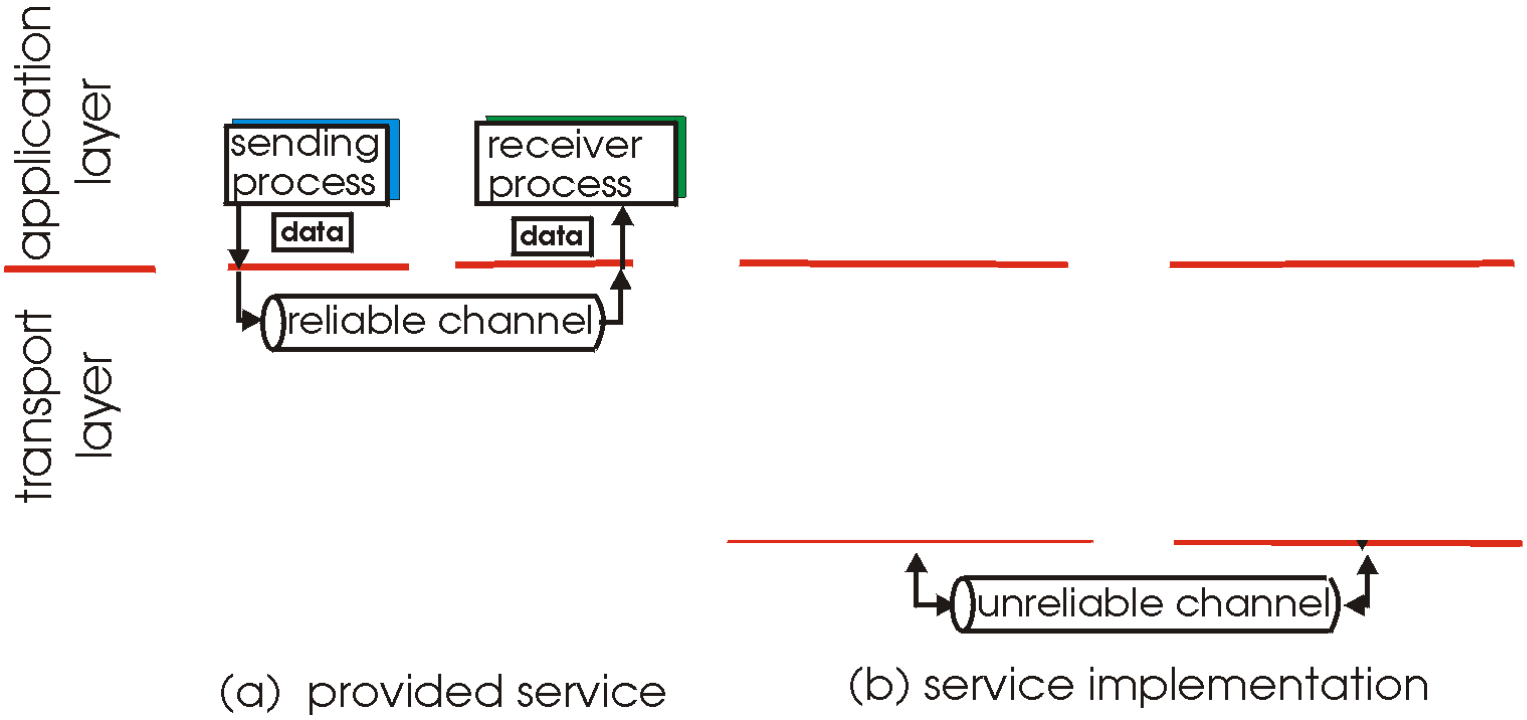


(a) provided service

- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Principles of reliable data transfer

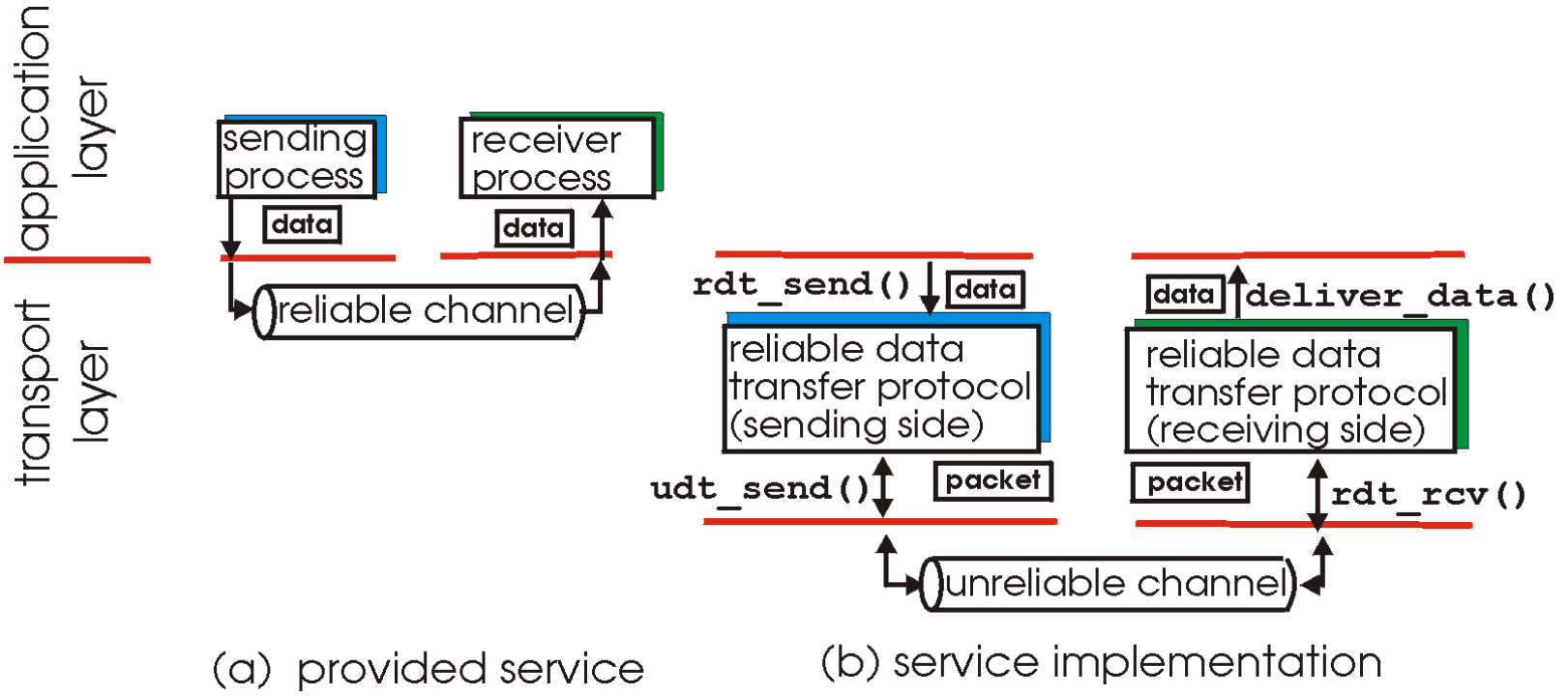
- important in application, transport, link layers
  - top-10 list of important networking topics!



- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Principles of reliable data transfer

- important in application, transport, link layers
  - top-10 list of important networking topics!

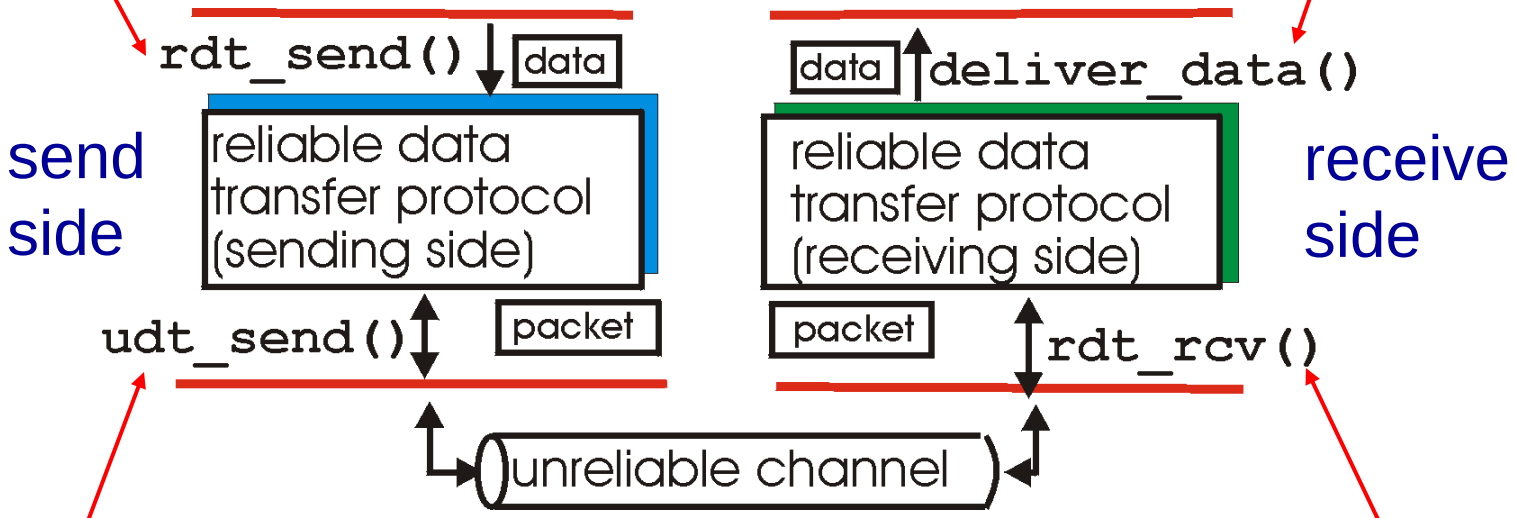


- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Principles of reliable data transfer

rdt_send(): called from above, (e.g., by app.). Passed data to deliver to receiver upper layer

deliver_data(): called by **rdt** to deliver data to upper



udt_send(): called by rdt, to transfer packet over unreliable channel to receiver

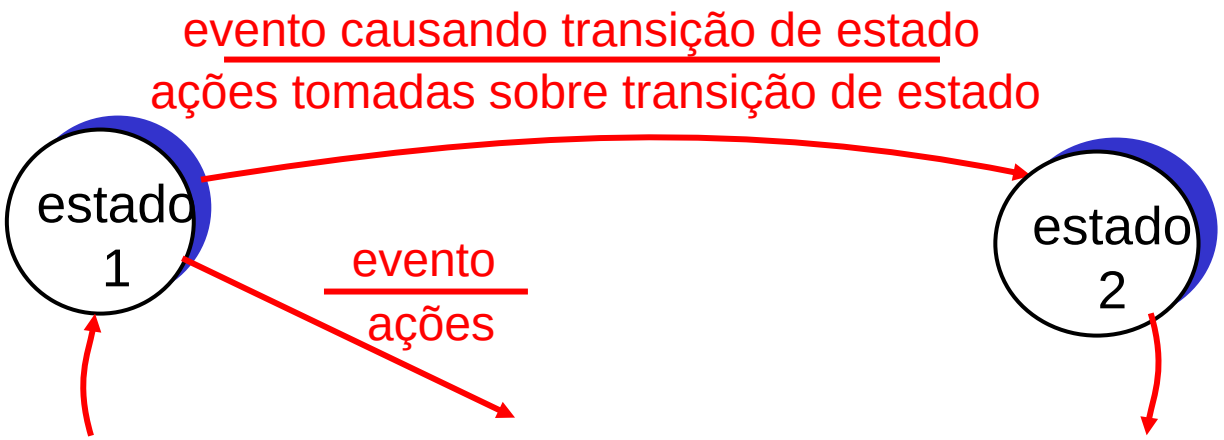
rdt_rcv(): called when packet arrives on rcv-side of channel

Transferência confiável de dados: introdução

vamos:

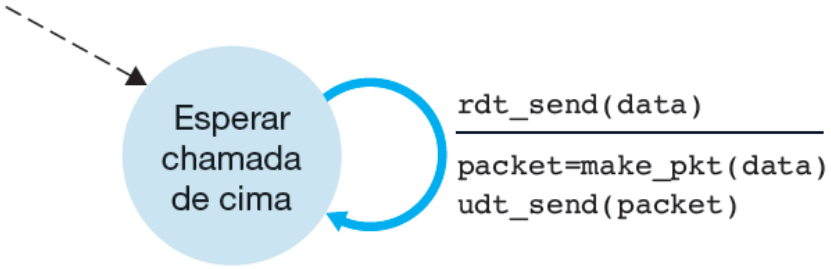
- desenvolver de forma incremental os lados remetente e destinatário do protocolo de transferência confiável de dados (rdt)
- considerar apenas a transf. de dados unidirecional. **Bidirecional**: é só duplicar a estrutura
  - mas informações de controle fluirão nas duas direções!
- usar máquinas de estado finito (FSM) para especificar remetente e destinatário

estado: quando neste "estado", próximo estado determinado exclusivamente pelo próximo evento

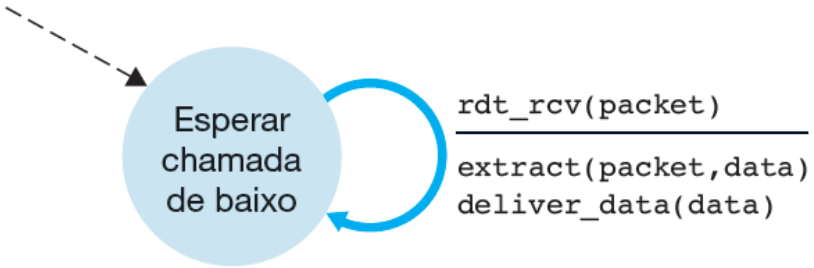


Transferência confiável de dados sobre um canal perfeitamente confiável: rdt1.0

- rdt1.0 – Um protocolo para um canal completamente confiável

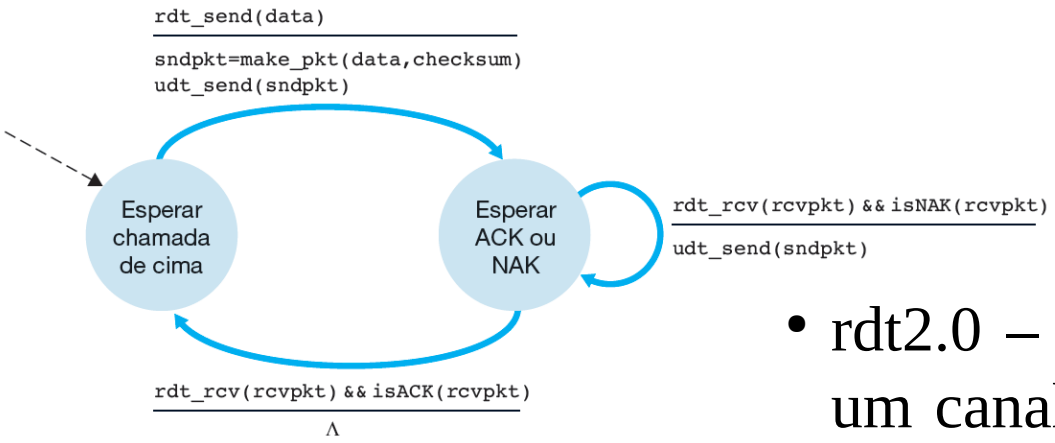


a. rdt1.0: lado remetente

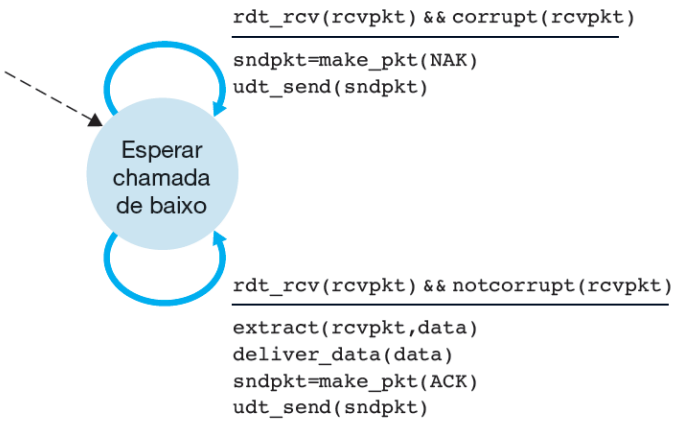


b. rdt1.0: lado destinatário

Transferência confiável de dados sobre um canal com erros de bits: rdt2.0



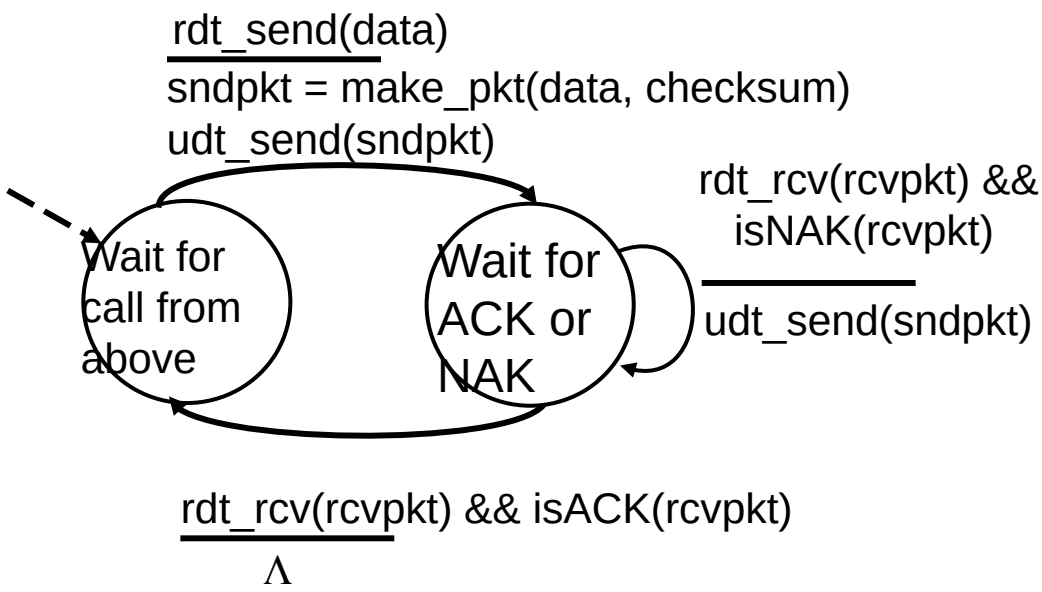
a. rdt2.0: lado remetente

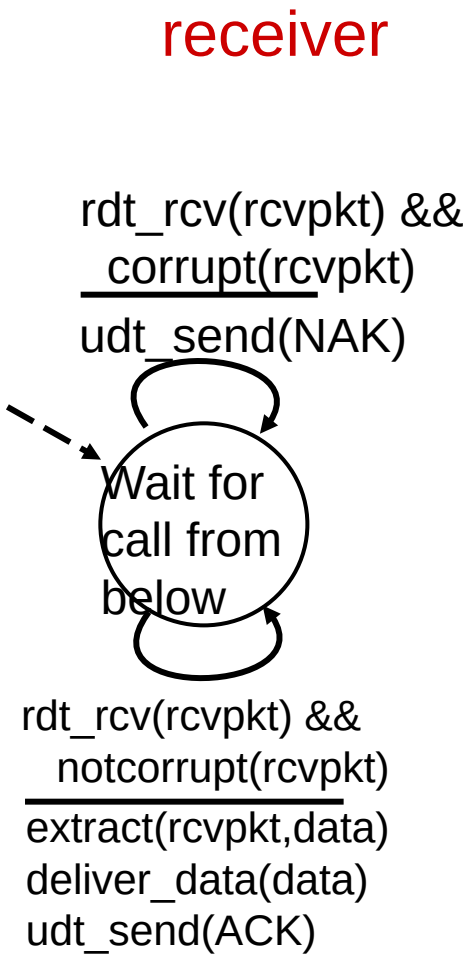


b. rdt2.0: lado destinatário

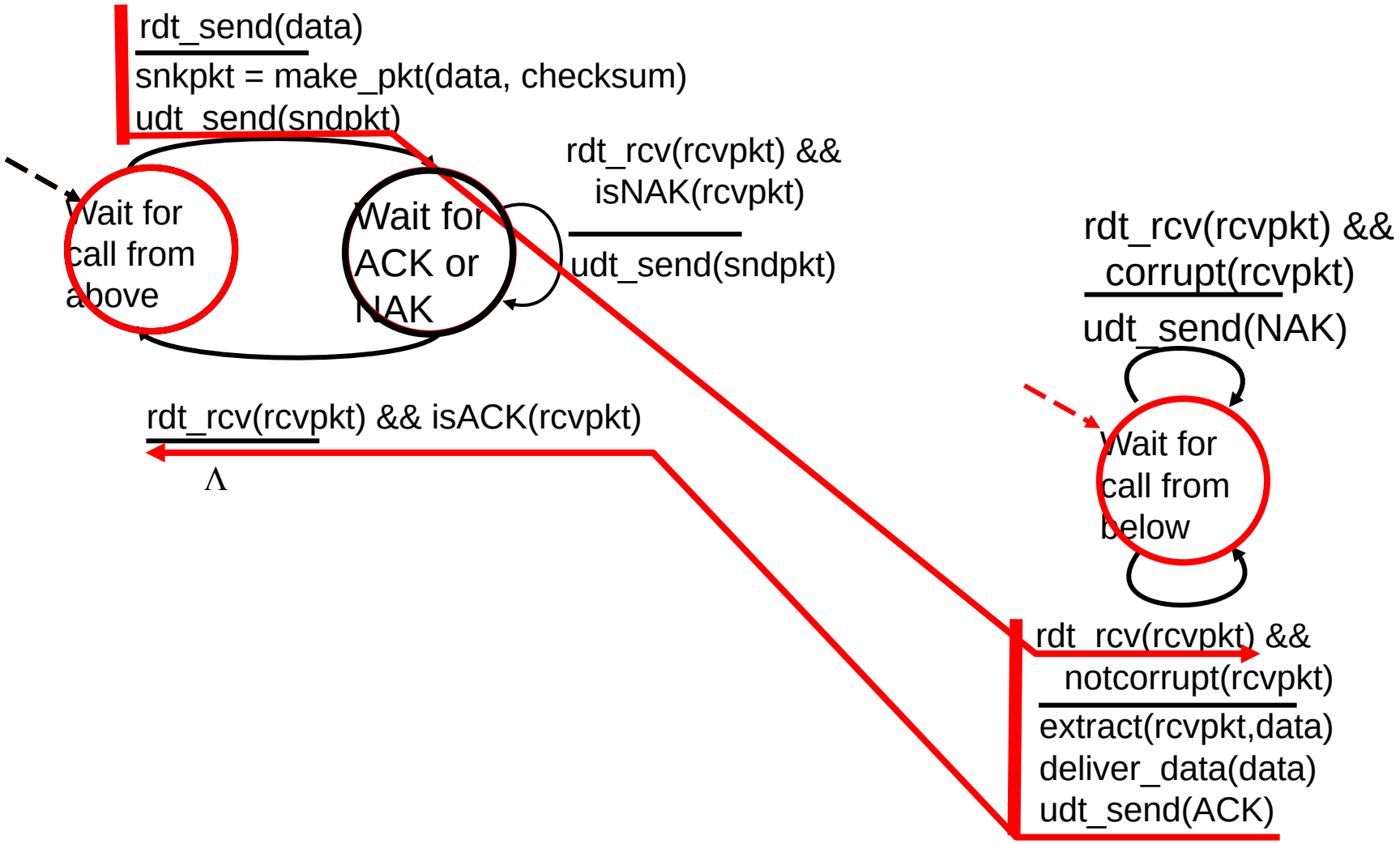
- rdt2.0 – Um protocolo para um canal com erros de bits. Base dos protocolos ARQ (*Automatic Repeat Request*)
  - ✔ Detecção de erros (mecanismo do UDP)
  - ✔ Realimentação do destinatário (ACK ou NACK)
  - ✔ Retransmissão
- Pare e espere

rdt2.0: FSM specification

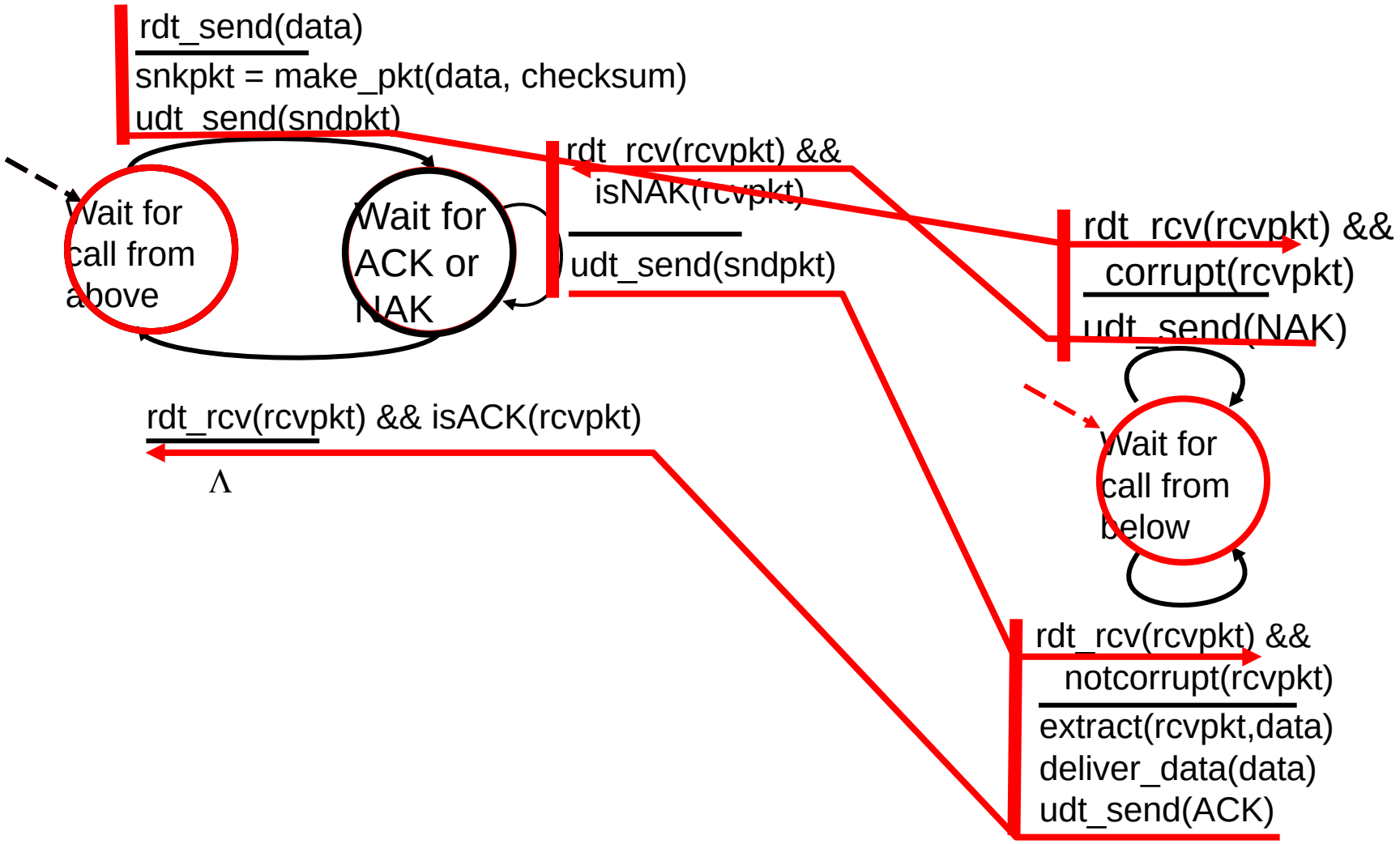




rdt2.0: operation with no errors



rdt2.0: error scenario

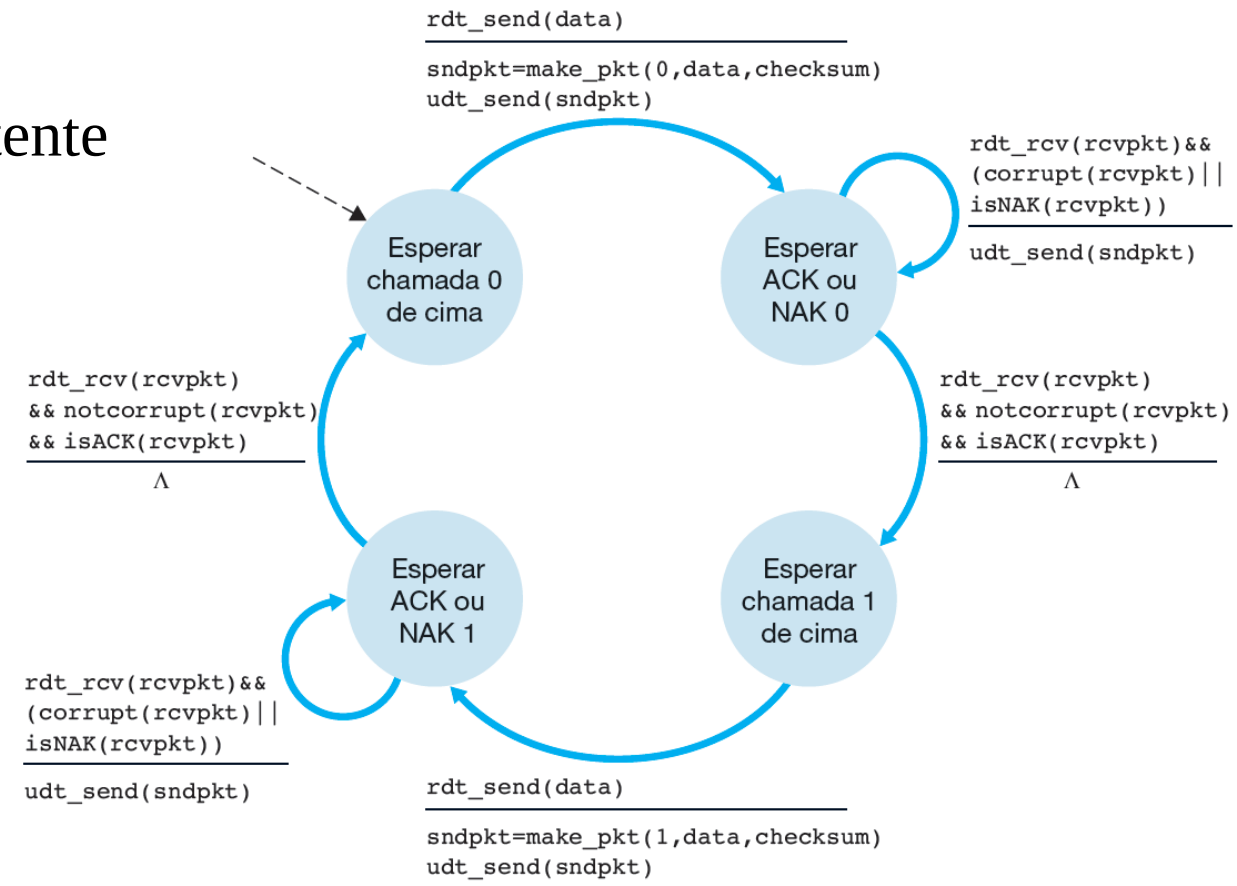


Transferência confiável de dados sobre um canal com erros de bits: rdt2.0

- O que acontece se o pacote ACK ou NAK estiver corrompido?
- Vamos considerar três possibilidade para ACKs ou NAKs corrompidos:
  1. Ser humano -- A: “...ditado....”, B: “OK” ou “Repita por favor” ==> A não entende: “O que foi que você disse?”... infinito. Péssimo!
  2. Bits de soma de verificação de tal modo que permita ao remetente a recuperação de erros de bits. Muito bom, mas serve somente para canais com erro de bits e não perda de pacotes.
  3. O Remetente reenvia o pacote de dados corrente quando receber um pacote ACK ou NAK truncado ==> pode gerar **pacotes duplicados.** Como o destinatário sabe se o pacote contém novos dados ou é uma retransmissão? ==> **número de sequência**. Este é o método utilizado.

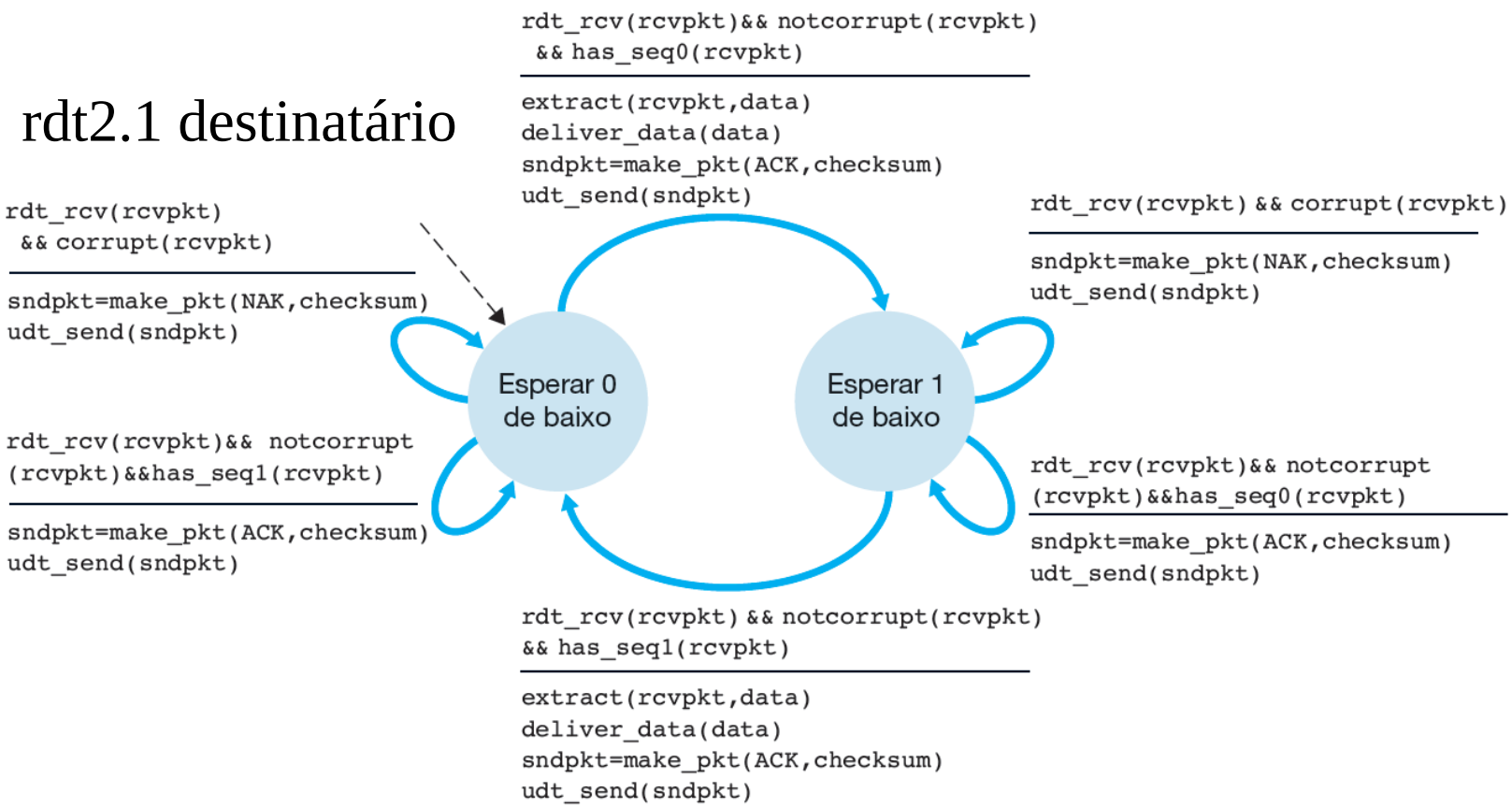
Transferência confiável de dados sobre um canal com erros de bits: rdt2.1

- rdt2.1 remetente



Transferência confiável de dados sobre um canal com erros de bits: rdt2.1

- rdt2.1 destinatário

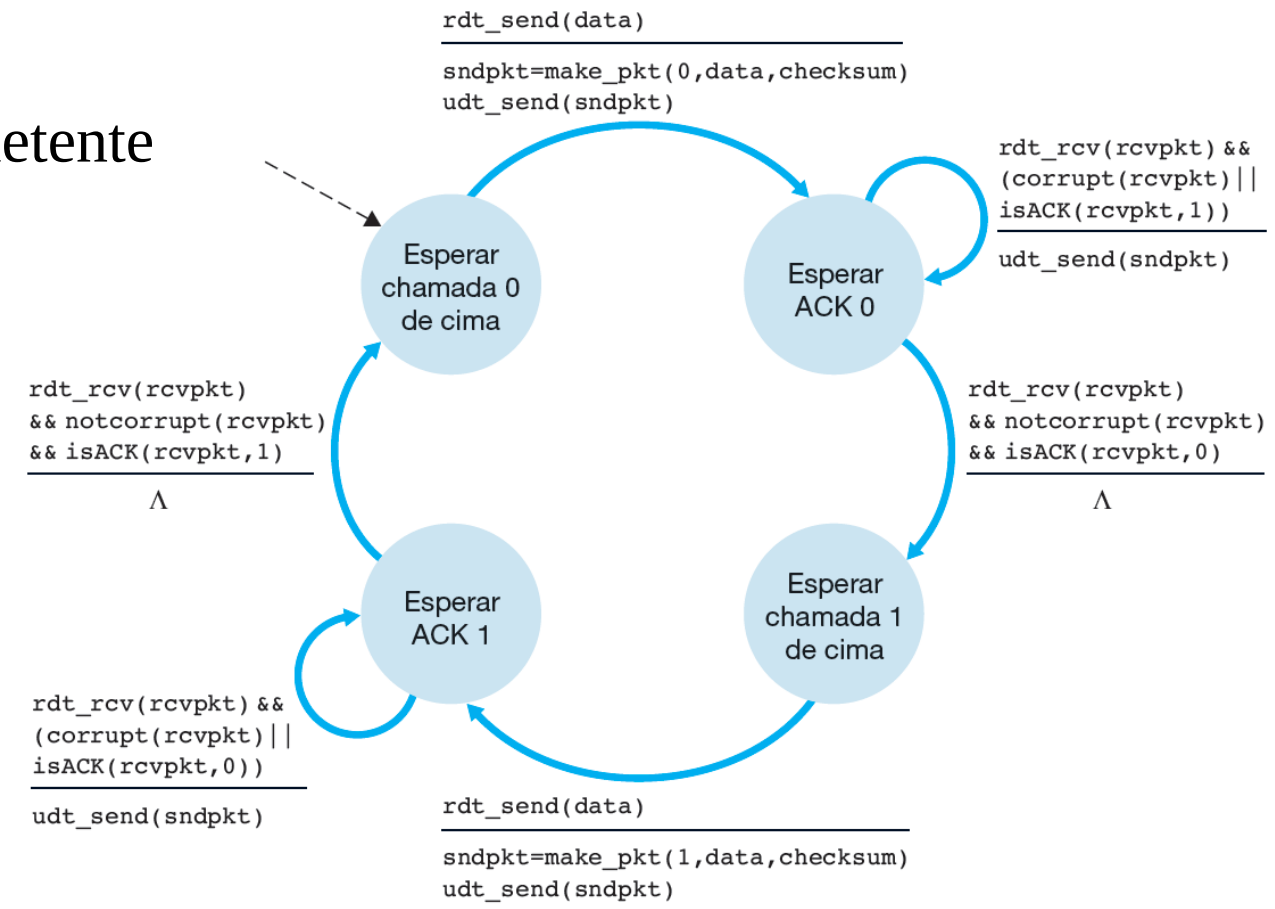


Transferência confiável de dados sobre um canal com erros de bits: rdt2.1-2

- O protocolo rdt2.1 usa tanto o reconhecimento positivo como negativo do remetente ao destinatário: pacote em ordem ==> ACK; pacote corrompido ==> NAK
- Podemos conseguir o mesmo efeito se, ao invés de enviarmos um NAK, enviarmos um ACK do último corretamente recebido. Ou seja, **ACK também numerado**.
- **ACKs duplicados** indicam ao remetente que o destinatário não recebeu corretamente o pacote seguinte àquele do ACK duplo.
- rdt2.2 deve enviar o número de sequência no ACK.

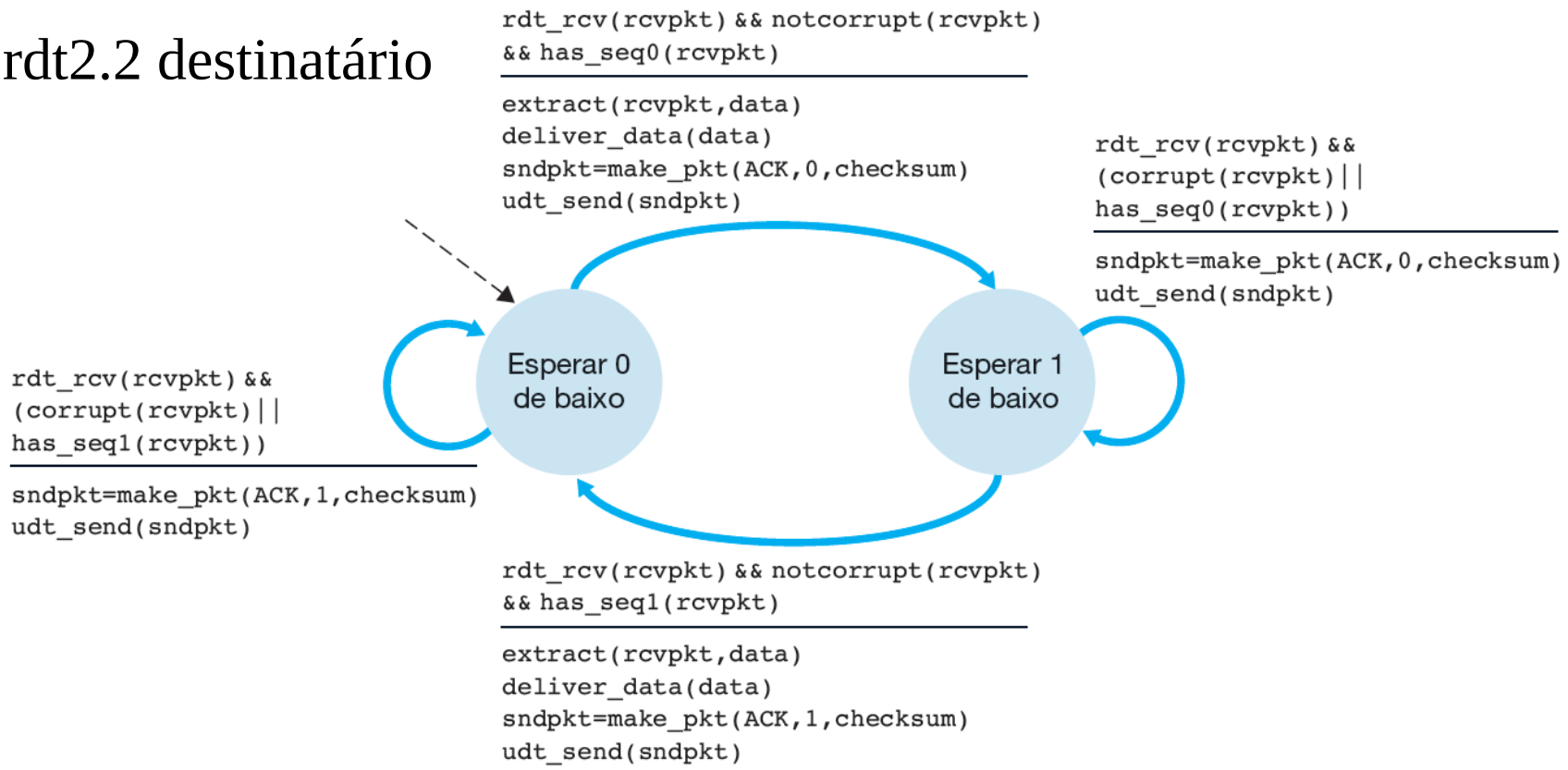
Transferência confiável de dados sobre um canal com erros de bits: rdt2.2

- rdt2.2 remetente



Transferência confiável de dados sobre um canal com erros de bits: rdt2.2

- rdt2.2 destinatário



Transferência confiável de dados sobre um canal com perda de pacotes e com erros de bits: rdt3.0

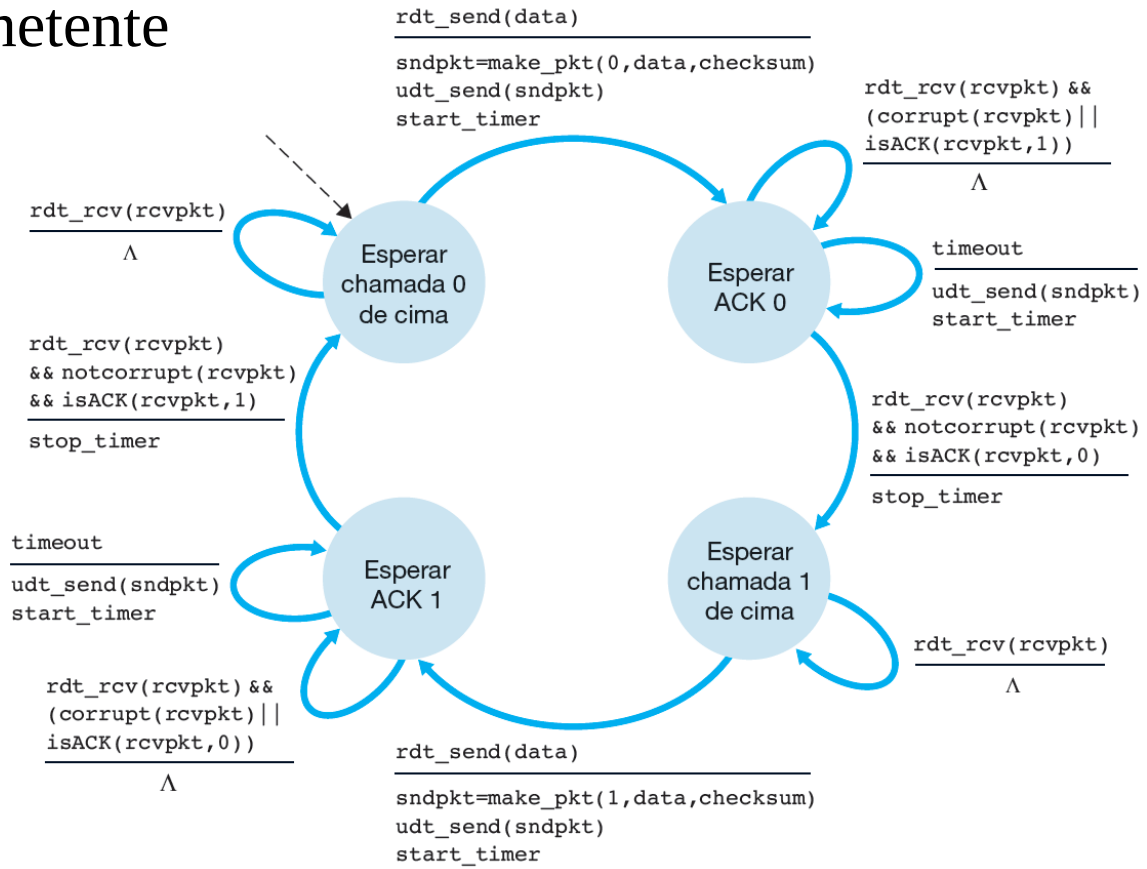
- Duas preocupações adicionais devem ser tratadas pelo protocolo:
  1. Como detectar perda de pacotes?
  2. O que fazer quando há perdas?
- Com ACK, ou falta dele, pode-se detectar a perda de pacotes.
- E para a segunda questão?
- Suponha que o remetente transmita um pacote e que esse pacote ou seu ACK seja perdido:
  - Nenhuma resposta chegará ao remetente
  - Se o remetente não quiser esperar ==> retransmite o pacote
- Mas quanto tempo o remetente precisa esperar para ter certeza que algo foi perdido?
  - Esperar um RTT
  - É muito difícil estimar o RTT, quanto mais ter certeza.

Transferência confiável de dados sobre um canal com perda e com erros de bits: rdt3.0

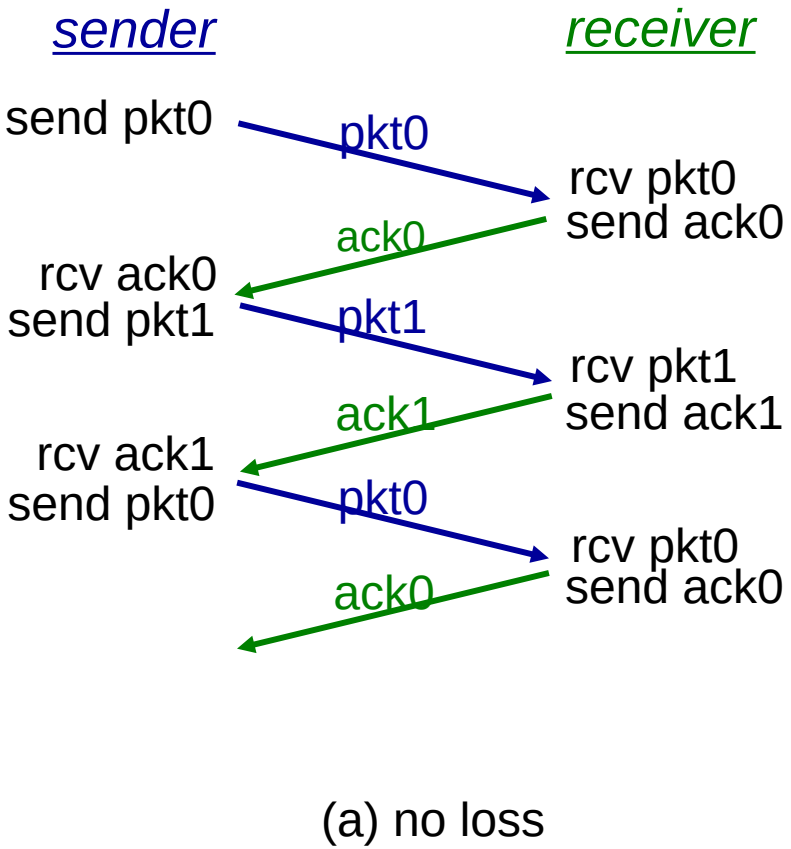
- Além disso, o ideal seria que o protocolo se recuperasse da perda assim que possível
- Na prática
  - O remetente faz uma escolha ponderada de um RTT provável, mas não garantido, que a perda tivesse acontecido. Se não receber um ACK nesse período retransmite o pacote.
- Se o pacote sofrer um atraso longo, pode haver retransmissão mesmo que o pacote ou ACK não tenham sido perdidos.
  - Possibilidade de **pacotes de dados duplicados**
- Para implementar um mecanismo de retransmissão com base no tempo é necessário um **temporizador de contagem regressiva.** Ações: iniciar, atender, parar.

Transferência confiável de dados sobre um canal com perda e com erros de bits: rdt3.0

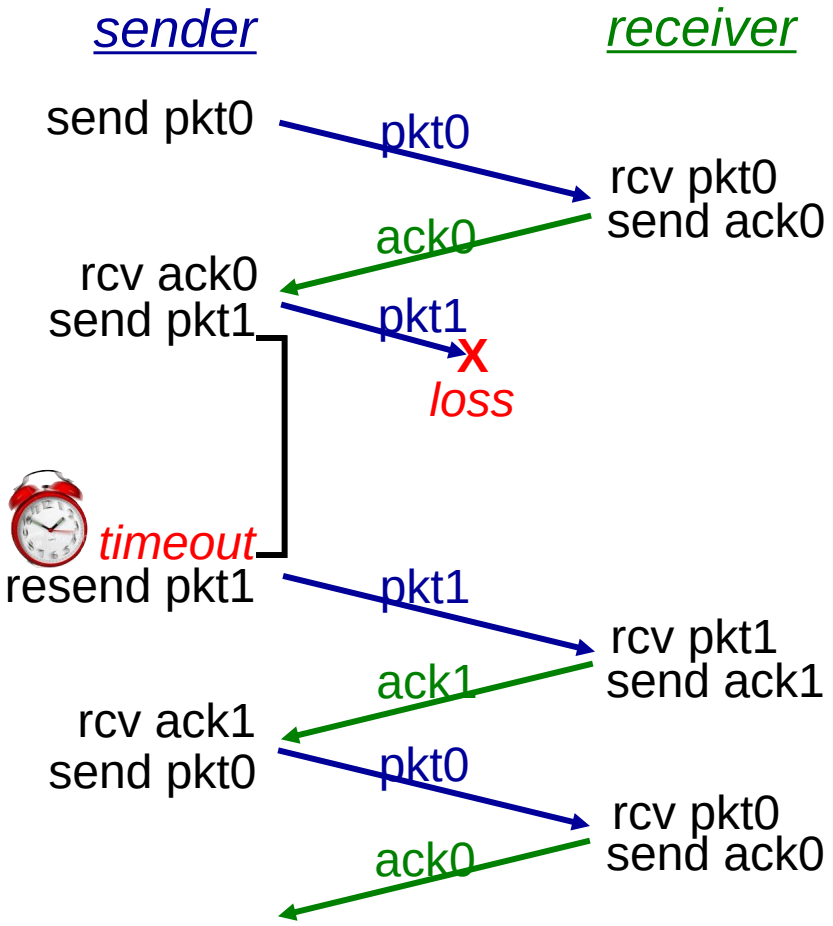
- rdt3.0 remetente



rdt3.0 in action

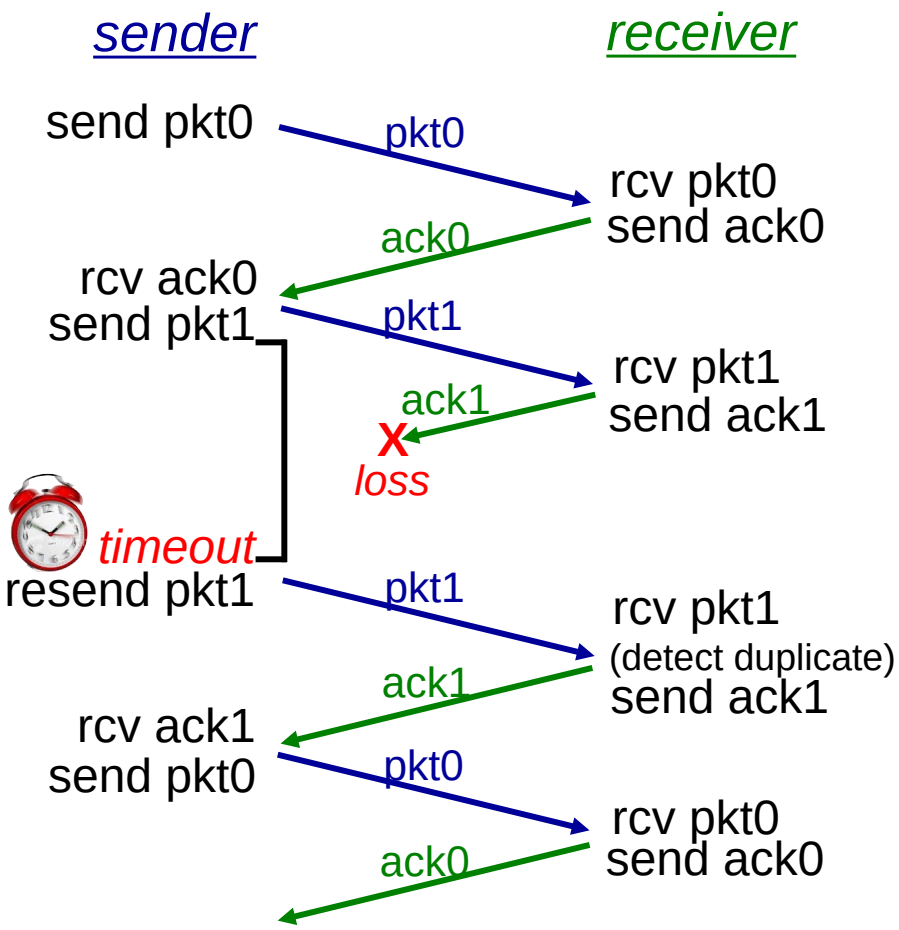


(a) no loss

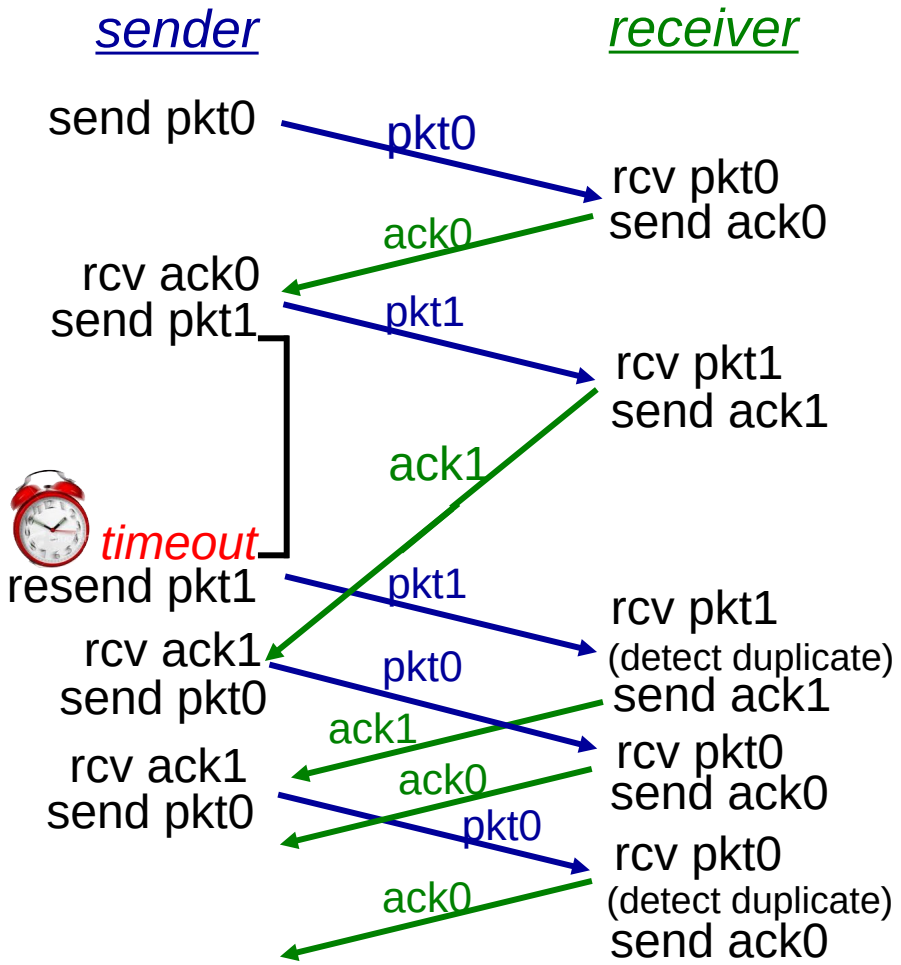


(b) packet loss

rdt3.0 in action



(c) ACK loss

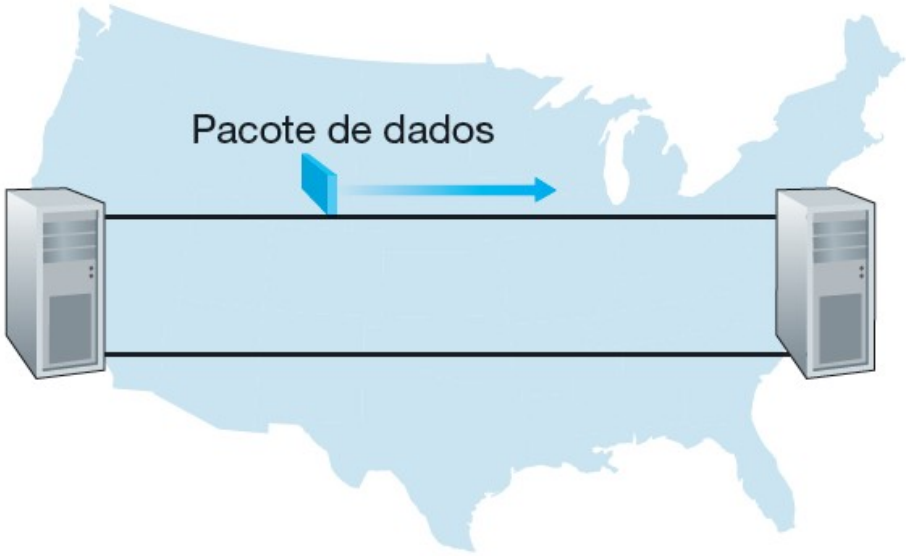


(d) premature timeout/ delayed ACK

Protocolos de transferência confiável de dados com paralelismo

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

- Um protocolo pare e espere em operação



Desempenho do rdt3.0

- rdt3.0 funciona, mas com desempenho ruim
- ex.: enlace 1 Gbps, 15 ms atraso, pacote 8000 bits:

$$t_{trans} = \frac{L}{R} = \frac{8000 \text{ bits/pacote}}{10^9 \text{ bps}} = 8\,us$$

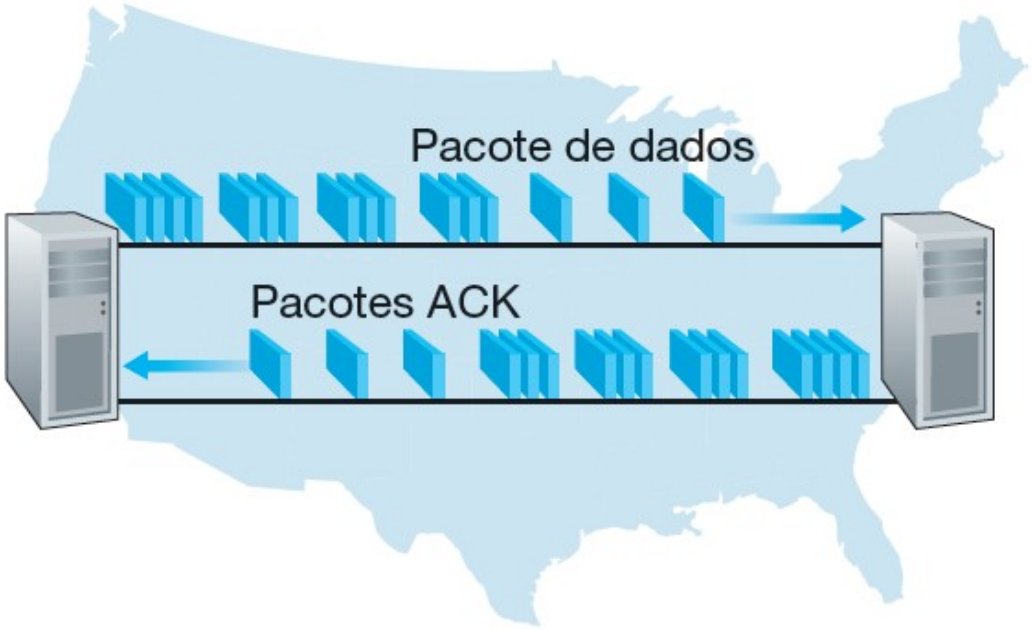
- U_{remet}: *utilização – fração do tempo remet. ocupado enviando*

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

- Pct. 1 KB cada 30 ms -> 267 kB/s vazão em enlace de 1 Gbps
- Protocolo de rede limita uso de recursos físicos!
- Solução?????

Protocolos de transferência confiável de dados com paralelismo

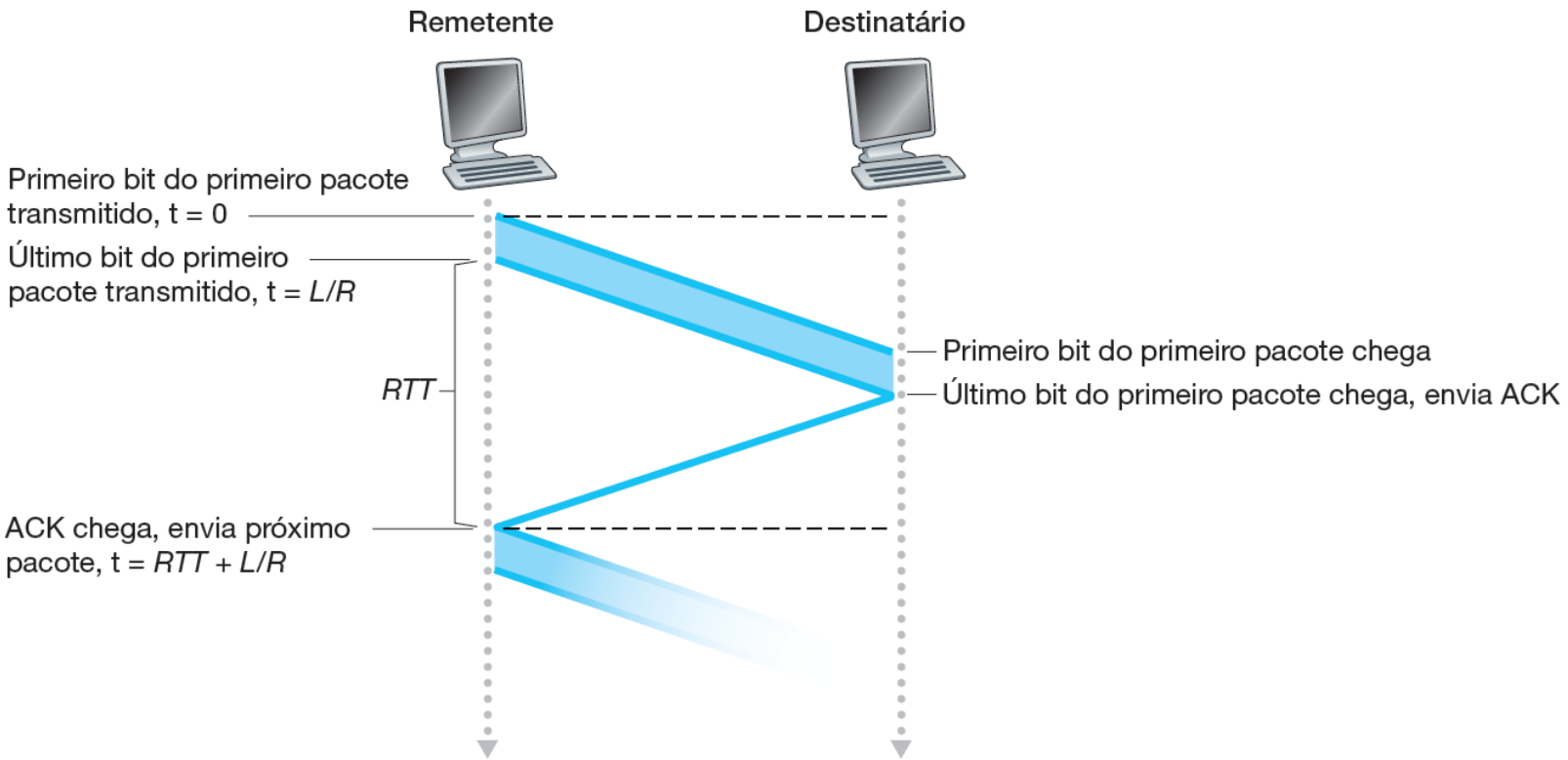
Um protocolo com paralelismo em operação



- O ACK recebido é de que pacote?

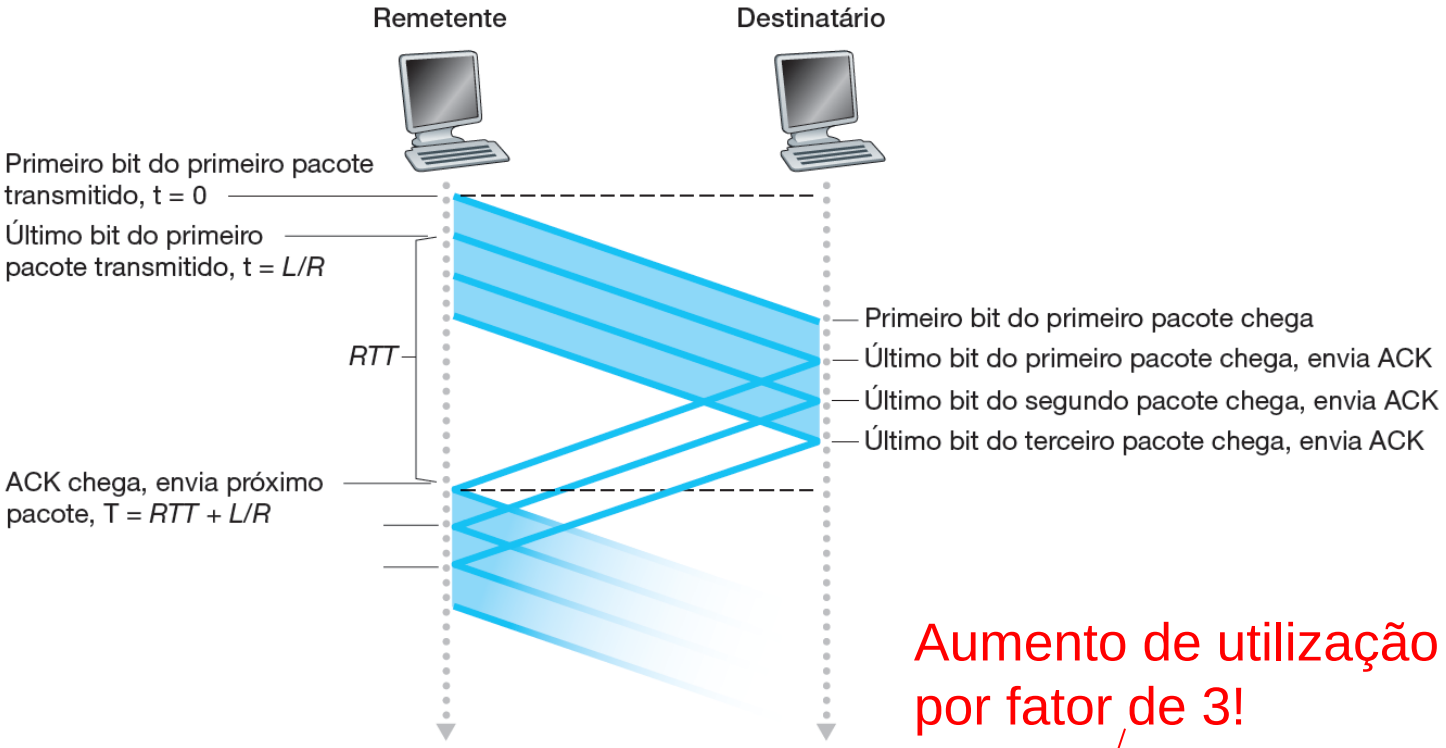
Protocolos de transferência confiável de dados com paralelismo

- Envio com pare e espere



Protocolos de transferência confiável de dados com paralelismo

- Envio com paralelismo



Aumento de utilização por fator de 3!

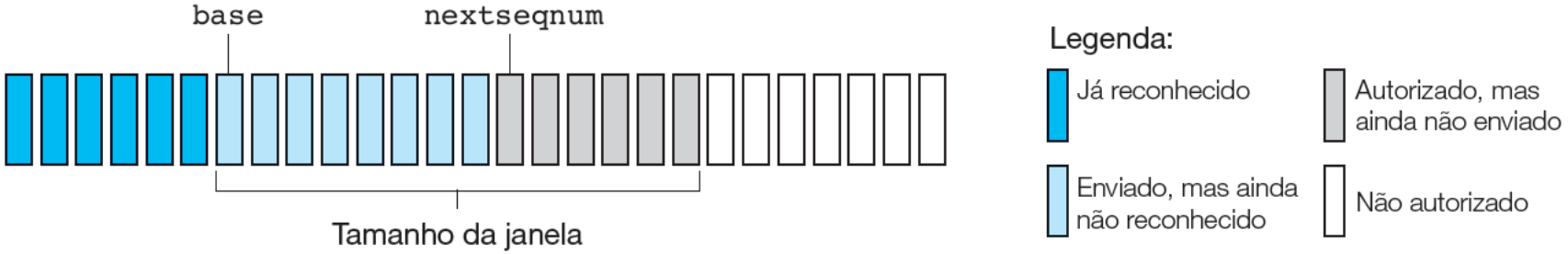
$$U_{remet} = \frac{3 * L/R}{RTT + L/R} = \frac{0{,}024}{30{,}008} = 0{,}0008$$

Go-Back-N (GBN)

- Em um **protocolo *Go-Back-N*** (**GBN**), o remetente é autorizado a transmitir múltiplos pacotes sem esperar por um reconhecimento.

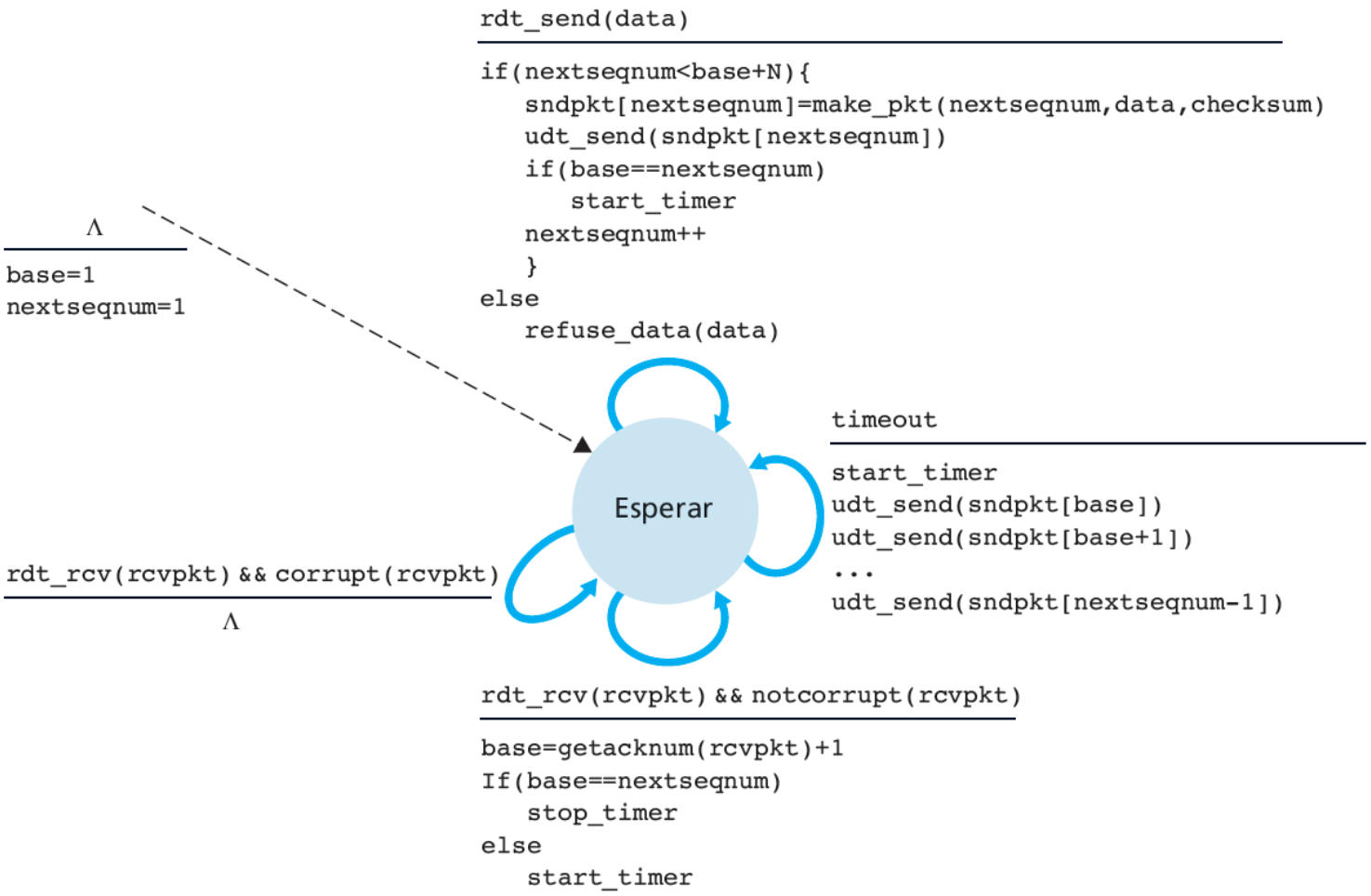
- Porém, fica limitado a ter não mais do que algum número máximo permitido, **N** (tamanho da janela), de pacotes não reconhecidos na "tubulação".

- Visão do remetente para os números de sequência no protocolo Go-Back-N:



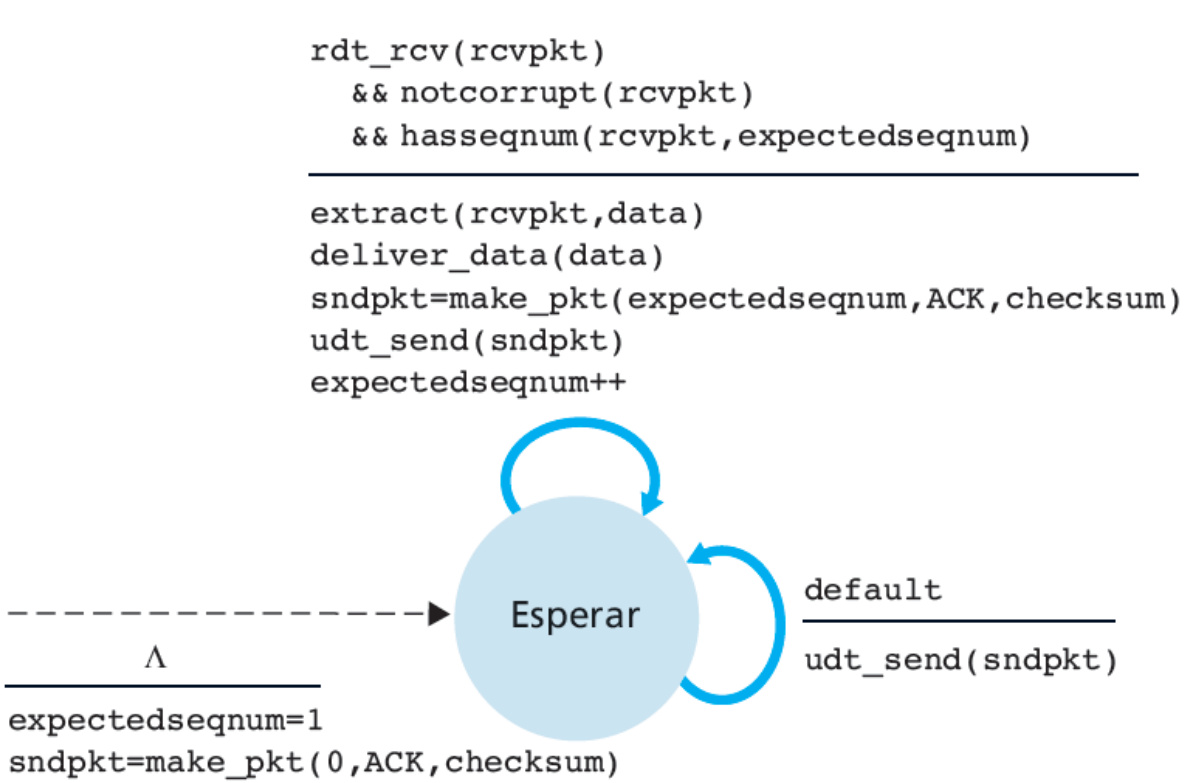
Go-Back-N (GBN)

- Descrição da FSM estendida do remetente GBN
- Quem consegue descrever o funcionamento?

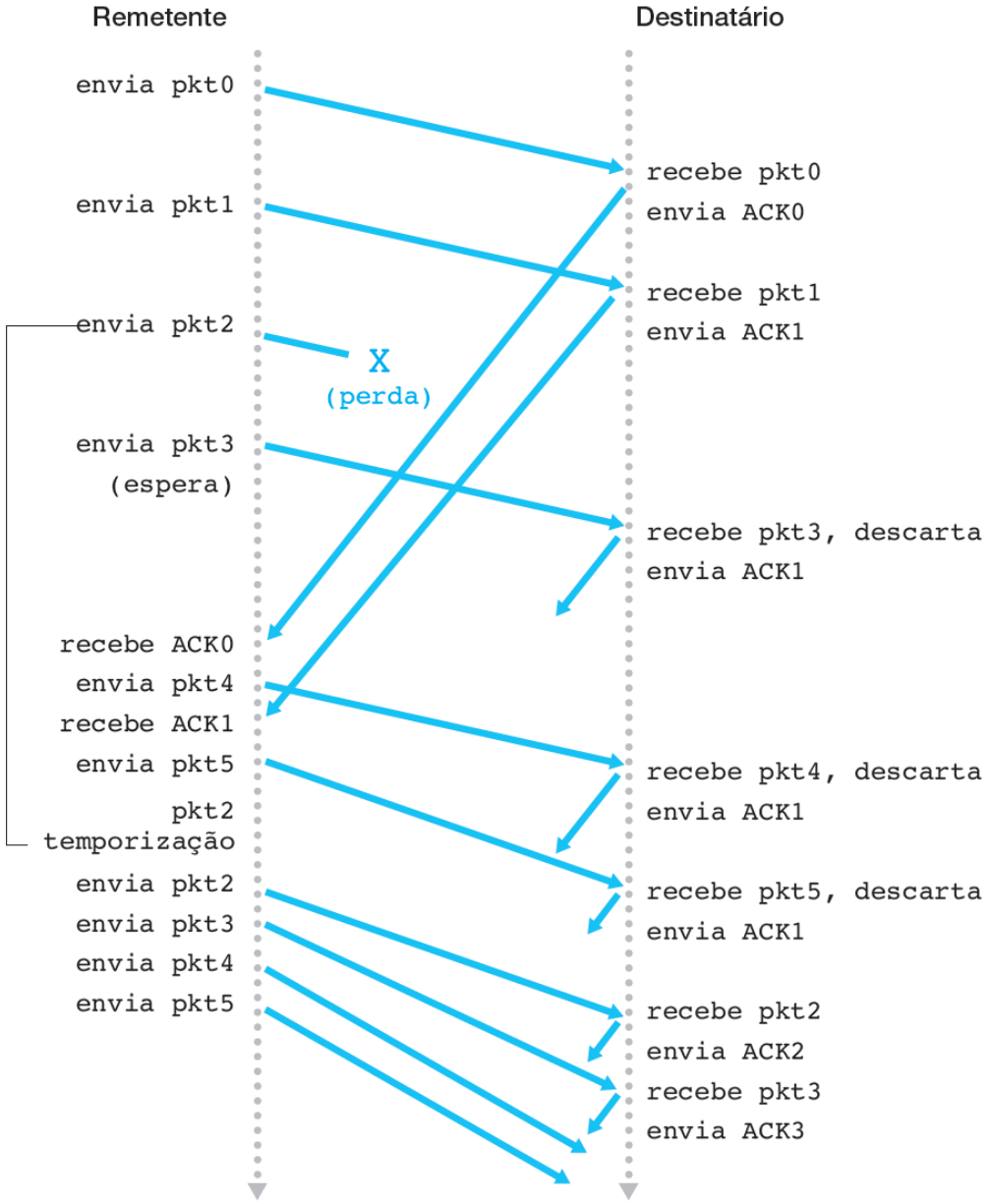


Go-Back-N (GBN)

- Descrição da FSM estendida do destinatário GBN



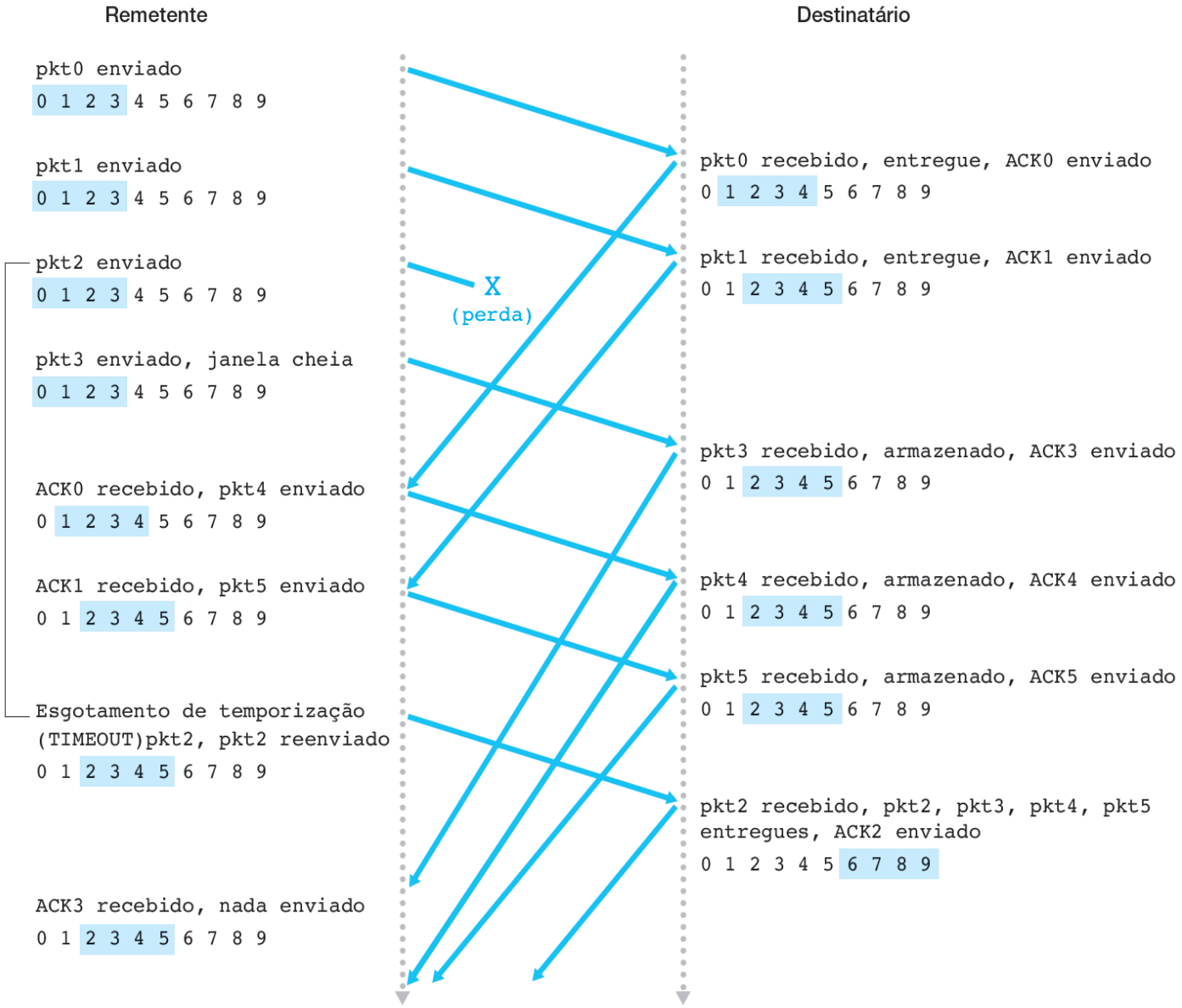
Go-Back-N (GBN) em operação – N = 4



Repetição seletiva (SR)

- Protocolos de repetição seletiva (SR) evitam retransmissões desnecessárias.
- Eles fazem o remetente retransmitir apenas os pacotes suspeitos de terem sido perdidos ou recebidos com erro no destinatário.
- Essa retransmissão individual, só quando necessária, exige que o destinatário reconheça individualmente os pacotes recebidos de modo correto armazene em *buffer* os pacotes recebidos fora de ordem.

Repetição seletiva (SR) – N = 4

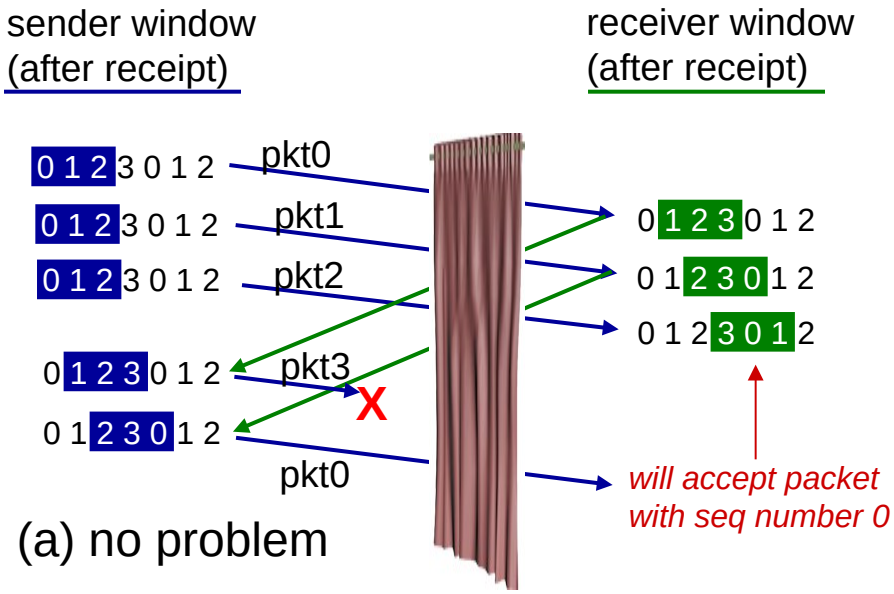


Selective repeat: dilemma

example:

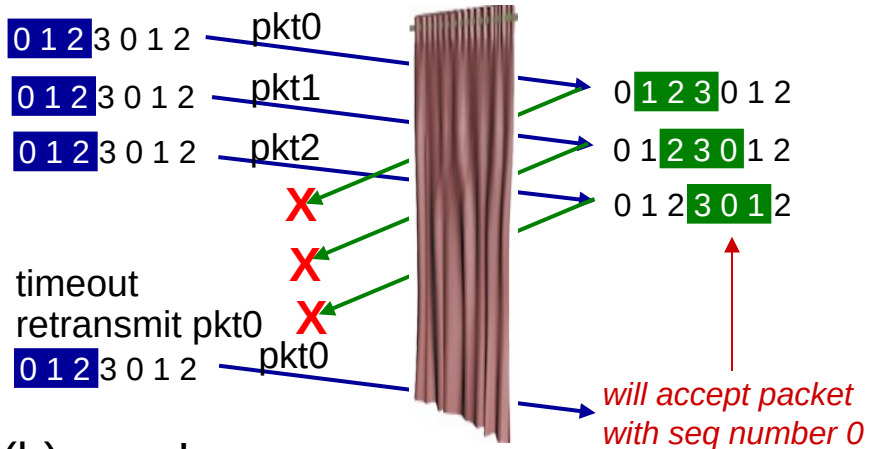
- seq #'s: 0, 1, 2, 3
- window size=3
- receiver sees no difference in two scenarios!
- duplicate data accepted as new in (b)

Q: what relationship between seq # size and window size to avoid problem in (b)?



(a) no problem

receiver can't see sender side.
receiver behavior identical in both cases!
something's (very) wrong!



(b) oops!

Transporte orientado para conexão: TCP

Transferência confiável

Canal pode ter erros de bits e perder pacotes

- Como lidar com erros de bits?
  1. Com *checksum* nas mensagens.
- Como saber se o pacote foi entregue?
  1. ACK - Com confirmação de recebimento.
- Como detectar perda de pacotes?
  1. ACK não recebido - Relógios temporizadores: passado determinado tempo sem reconhecimento, retransmite.
- Como lidar com pacotes duplicados?
  1. Com número de sequência em cada segmento.
- Como lidar com a lentidão do protocolo pare e espere?
  1. Enviar vários segmentos pela “tubulação”, cada um com seu número de sequência, armazenando em memória.
  2. Aguardar o reconhecimento (ACK) individual de cada segmento, apagando da memória.

Transporte orientado para conexão: TCP

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

- **Soma de verificação -** Usada para detectar erros de bits em um pacote transmitido.
- **Temporizador** - Usado para controlar a temporização/retransmissão de um pacote, possivelmente porque o pacote (ou seu ACK) foi perdido dentro do canal.

Transporte orientado para conexão: TCP

- **Número de sequência** - Usado para numeração sequencial de segmentos de dados que transitam do remetente ao destinatário. Ou seja, pode-se ter vários segmentos transitando simultaneamente no canal de comunicação, maximizando o uso do canal.

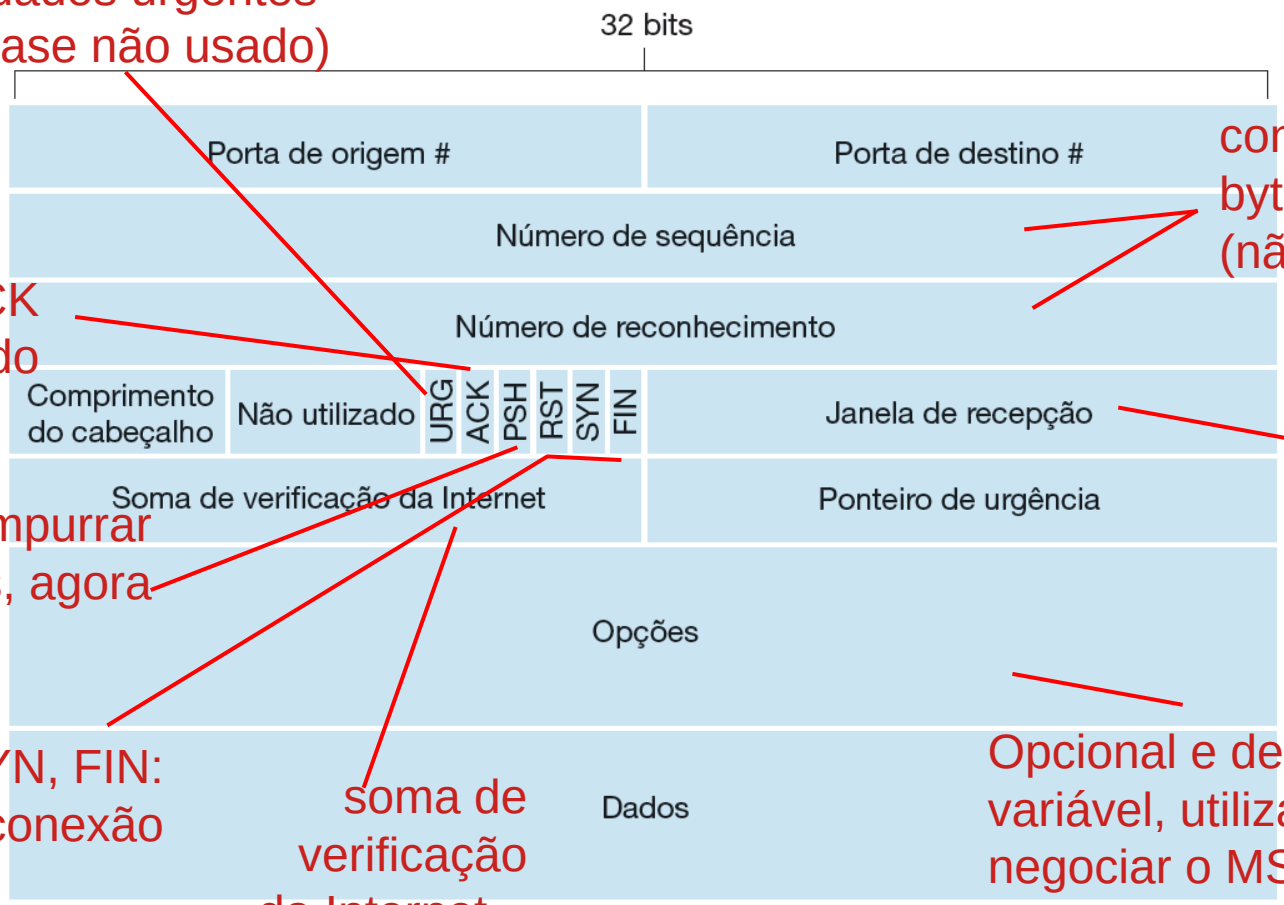
- **Reconhecimento** - Usado pelo destinatário para avisar o remetente que um pacote ou conjunto de pacotes foi recebido corretamente.

- **Janela, paralelismo** - O remetente fica restrito a enviar somente pacotes com números de sequência que pertençam a uma determinada faixa/janela.

A conexão TCP

- Uma conexão TCP provê um **serviço *full-duplex***.
- A conexão TCP é sempre **ponto a ponto**.
- Uma vez estabelecida uma conexão TCP, dois processos de aplicação podem enviar dados um para o outro. 3-*way handshake*, já estabelecendo o MSS (*Maximum Segment Size*).
- O TCP combina cada porção de dados do cliente com um cabeçalho TCP, formando, assim, **segmentos TCP**.

Estrutura do segmento TCP



URG: dados urgentes (quase não usado)

contagem por bytes de dados (não segmentos!)

ACK: # ACK válido

bytes destinatário pode aceitar

PSH: empurrar dados, agora

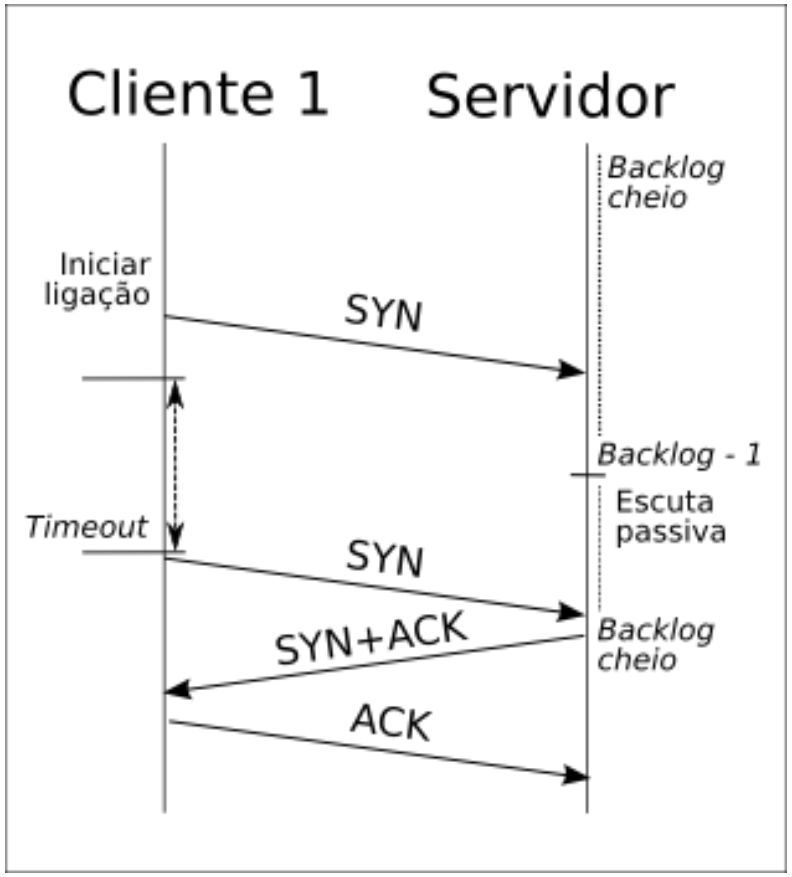
RST, SYN, FIN: estab. conexão

soma de verificação da Internet – todo o segmento (como em UDP)

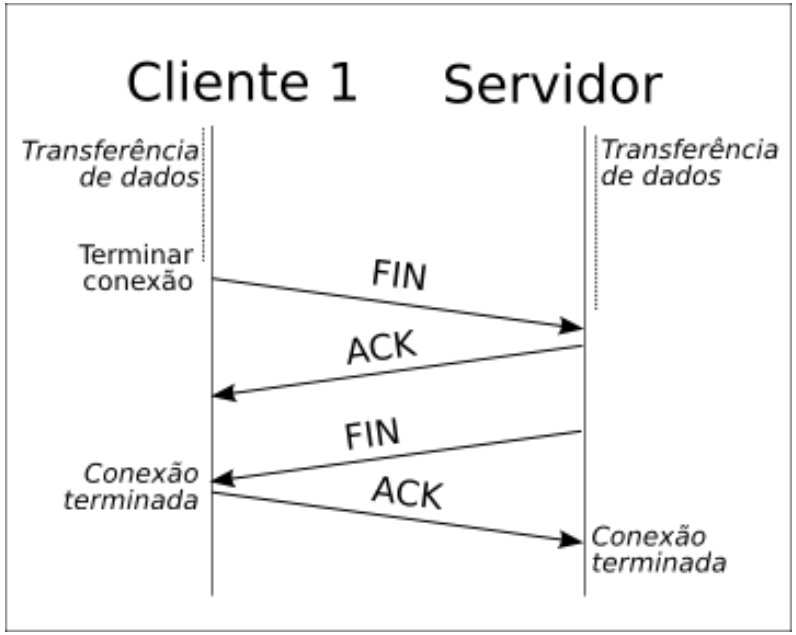
Opcional e de comprimento variável, utilizado para negociar o MSS ou fator de escala em redes de alta velocidade

TCP: Estabelecendo e fechando conexões

3-way handshake



Finalizando



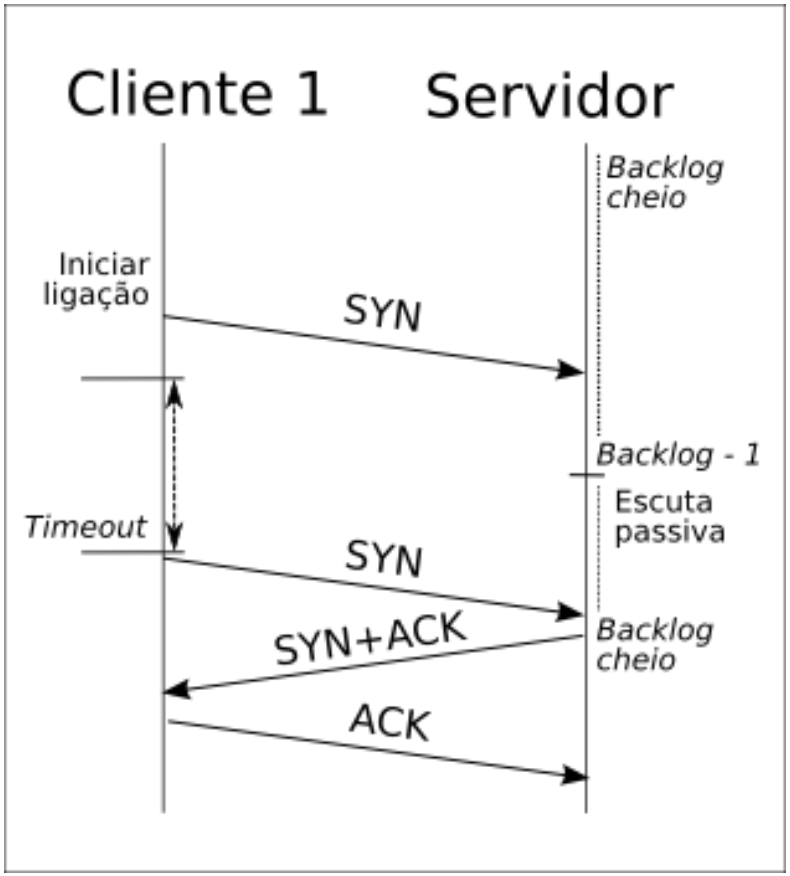
Gerenciamento da conexão TCP

apresentação de 3 vias:

etapa 1: hosp. cliente envia segmento SYN do TCP ao servidor
- especifica # seq. inicial
- sem dados

etapa 2: hosp. servidor recebe SYN, responde com segmento SYNACK
- servidor aloca *buffers*
- especifica # seq. inicial do servidor

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



Gerenciamento da conexão TCP

fechando uma conexão:

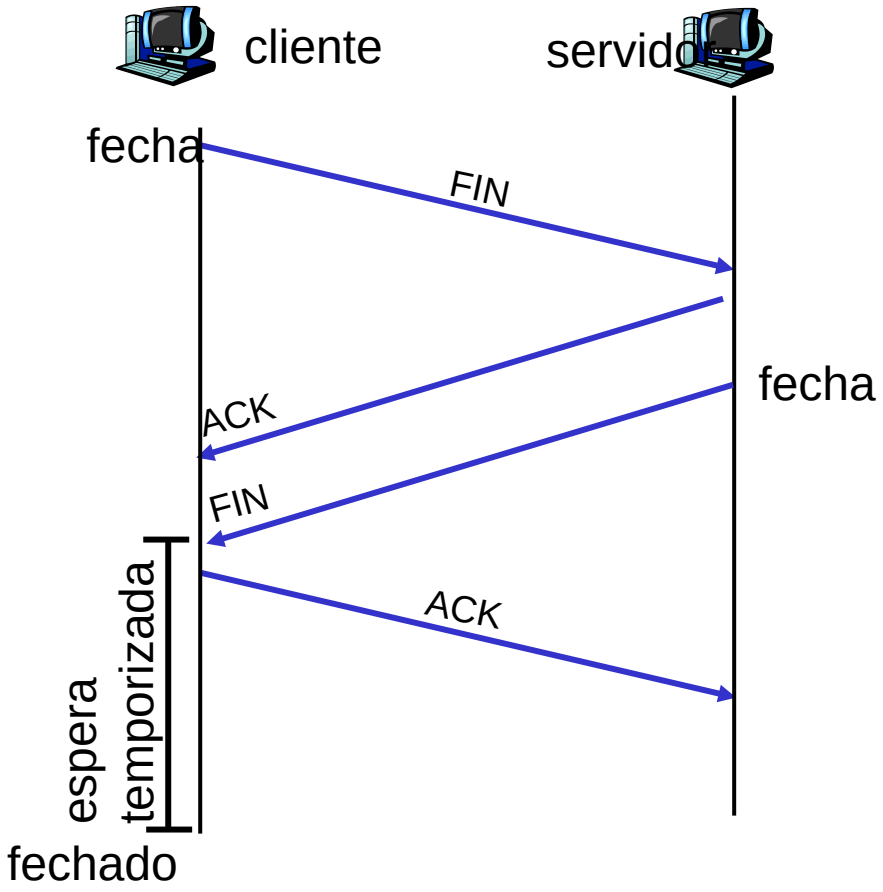
cliente fecha socket:

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

etapa 1: sistema final do cliente envia segmento de controle TCP FIN ao servidor

etapa 2: servidor recebe FIN, responde com ACK. Fecha conexão, envia FIN. (FINACK)

etapa 3: cliente recebe FINACK, responde com ACK.



#s sequência e ACKs do TCP

Redes de Computadores

#’s de sequência:

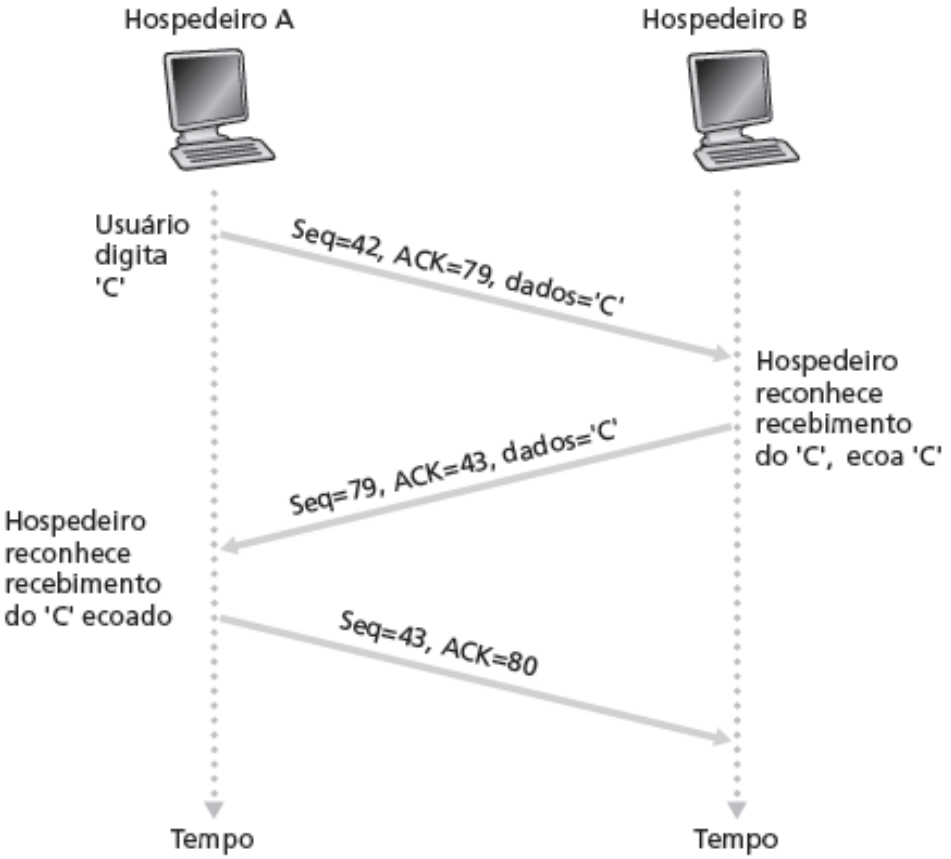
- “número” na cadeia de bytes do 1º byte nos dados do segmento

ACKs:

- # seq do **próximo** byte esperado do outro lado
- ACK cumulativo

P: como o destinatário trata segmentos fora de ordem

- R: TCP não diz – a critério do implementador



Cenário preestabelecido: telnet simples

Números de sequência e números de reconhecimento

- O **número de sequência** para um segmento é o número do primeiro byte do segmento.

- O **número de reconhecimento** que o hospedeiro A atribui a seu segmento é o número de sequência do próximo byte que ele estiver aguardando do hospedeiro B.

- Como o TCP somente reconhece bytes até o primeiro byte que estiver faltando na cadeia, dizemos que o TCP provê **reconhecimentos cumulativos**.

Tempo de ida e volta e *timeout* do TCP

P: Como definir o valor de *timeout* do TCP?

- maior que RTT
  - mas RTT varia
- muito curto: *timeout* prematuro
  - retransmissões desnecessárias
- muito longo: baixa reação a perda de segmento

P: Como estimar o RTT?

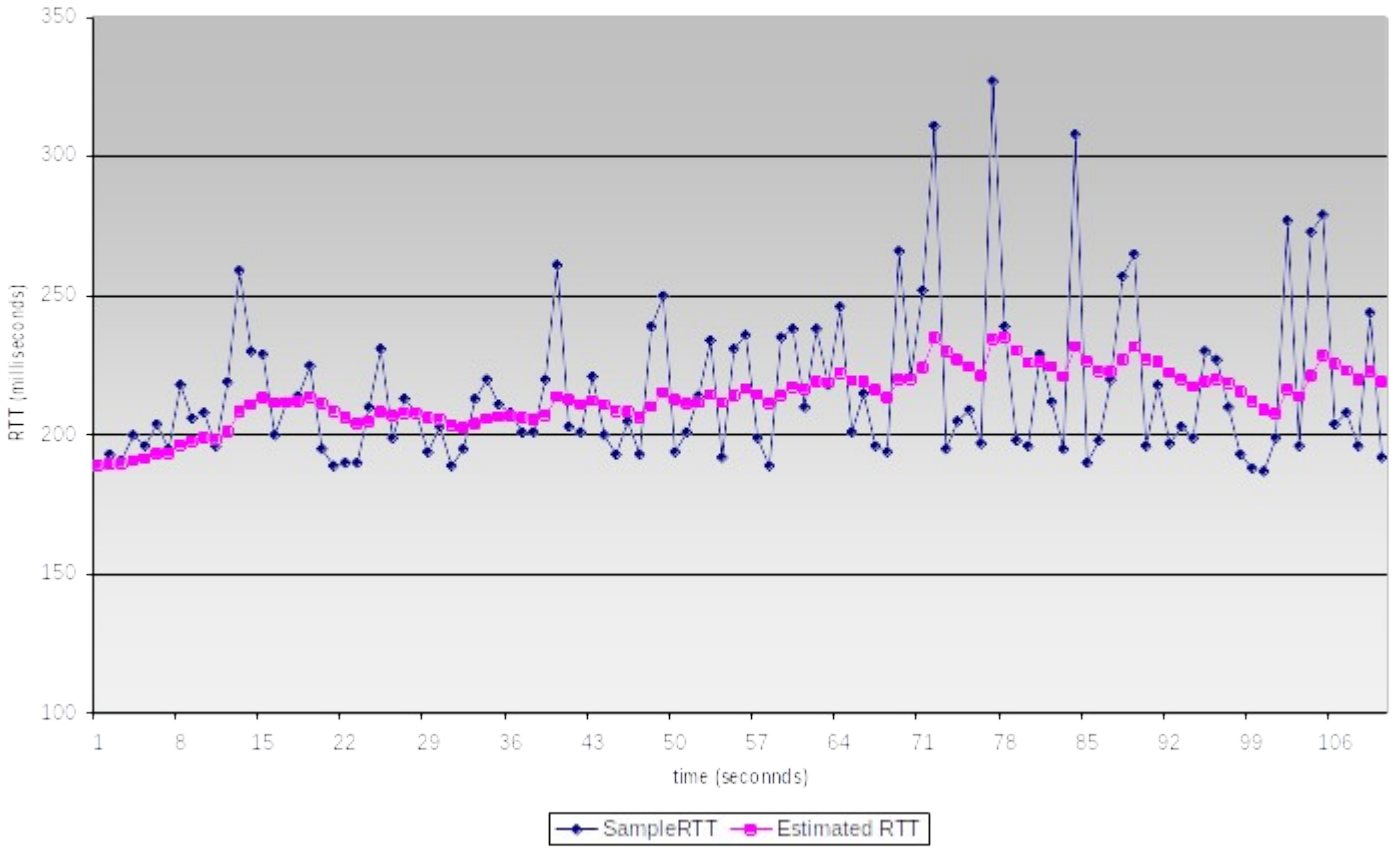
- **`SampleRTT`**: tempo medido da transmissão do segmento até receber o ACK
  - ignora retransmissões
- **`SampleRTT`** variará; queremos RTT estimado "mais estável"
  - média de várias medições recentes, não apenas **`SampleRTT`** atual

Tempo de ida e volta e *timeout* do TCP

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

- média móvel exponencial ponderada
- influência da amostra passada diminui exponencialmente rápido
- valor típico: α = 0,125

Amostras de RTTs estimados:



Tempo de ida e volta e *timeout* do TCP

Redes de Computadores

definindo o *timeout*

- **EstimtedRTT** mais “margem de segurança”
  - grande variação em **EstimatedRTT ->** maior margem de seg.
- primeira estimativa do quanto SampleRTT se desvia de EstimatedRTT:

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

(geralmente, $\beta = 0,25$)

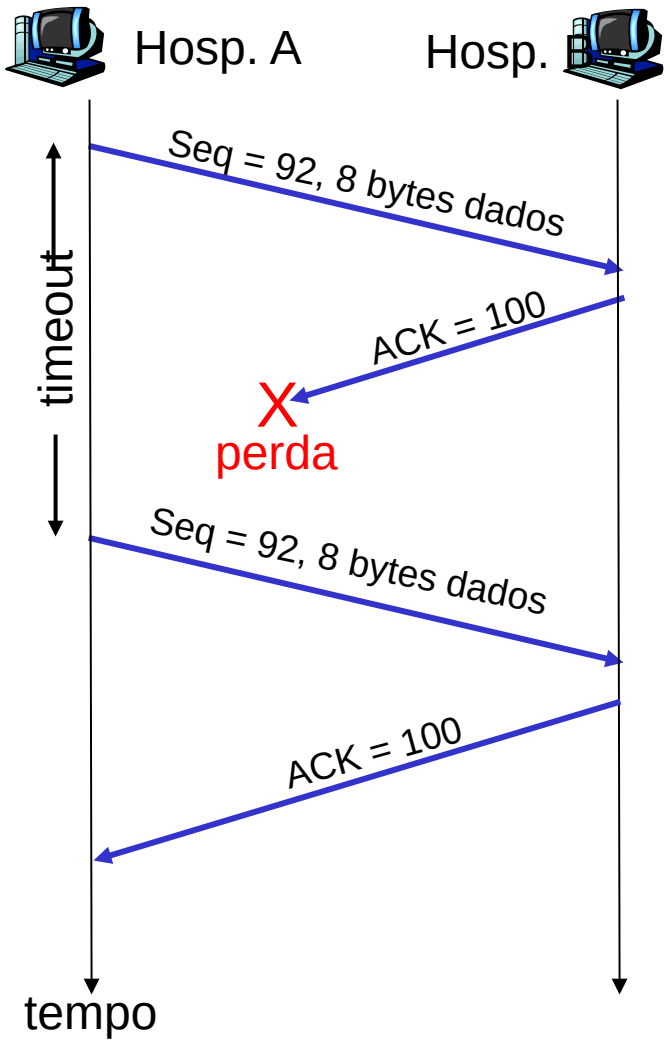
depois definir intervalo de *timeout*

$$TimeoutInterval = EstimatedRTT + 4*DevRTT$$

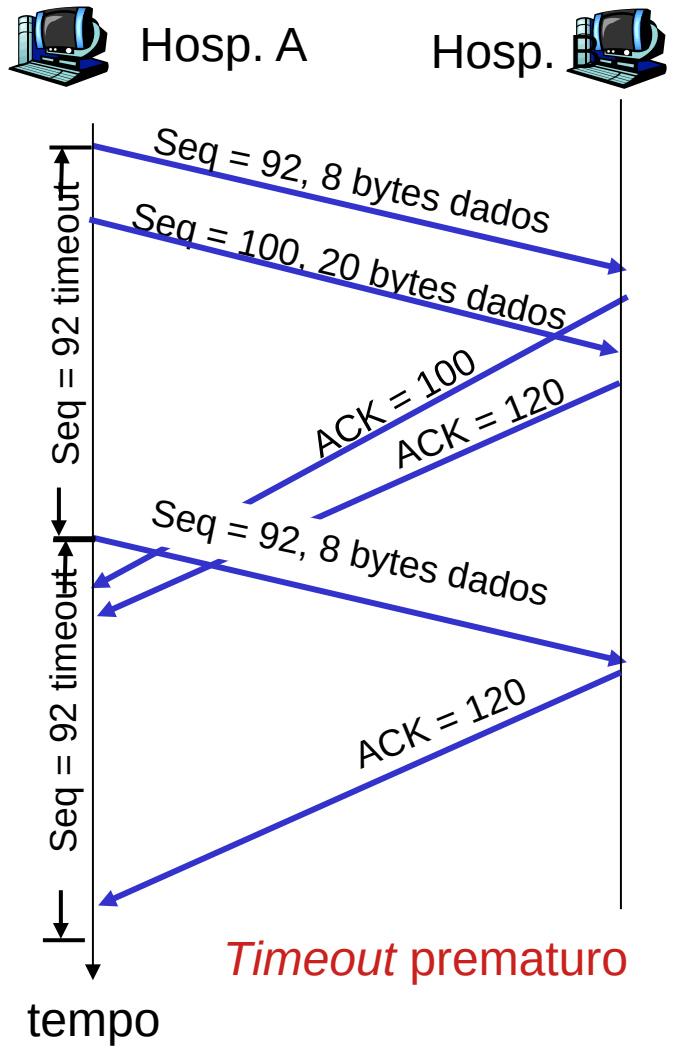
$$TimeoutInterval_{inicial} = 1\ s$$

“margem de segurança”

TCP: cenários de retransmissão

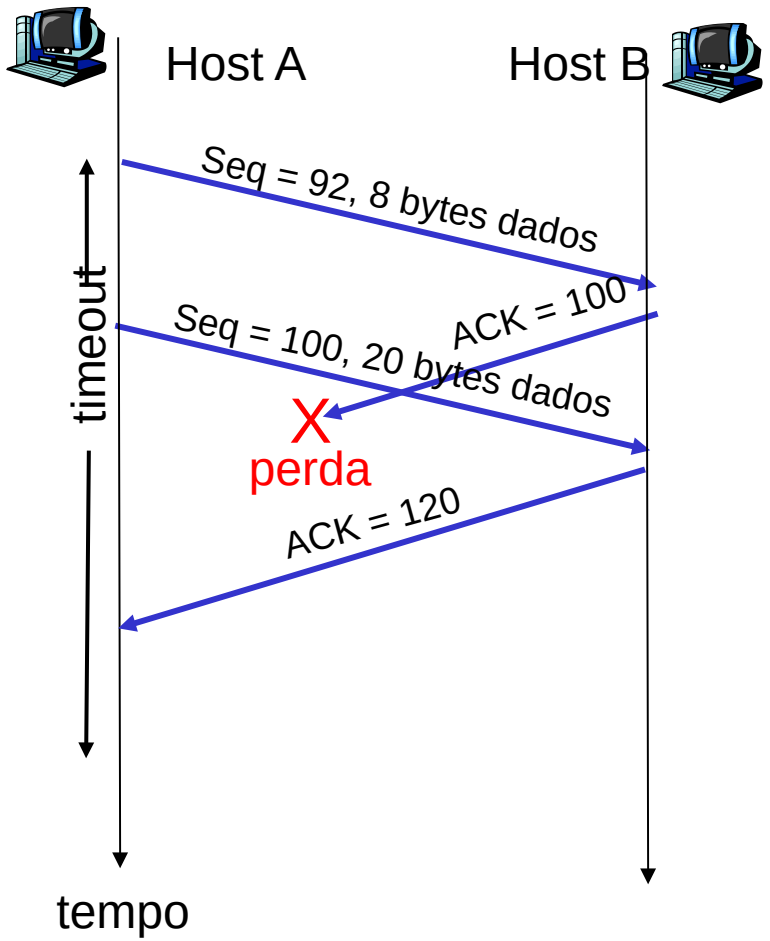


Cenário de ACK perdido



Timeout prematuro

TCP: cenários de retransmissão



Cenário ACK cumulativo

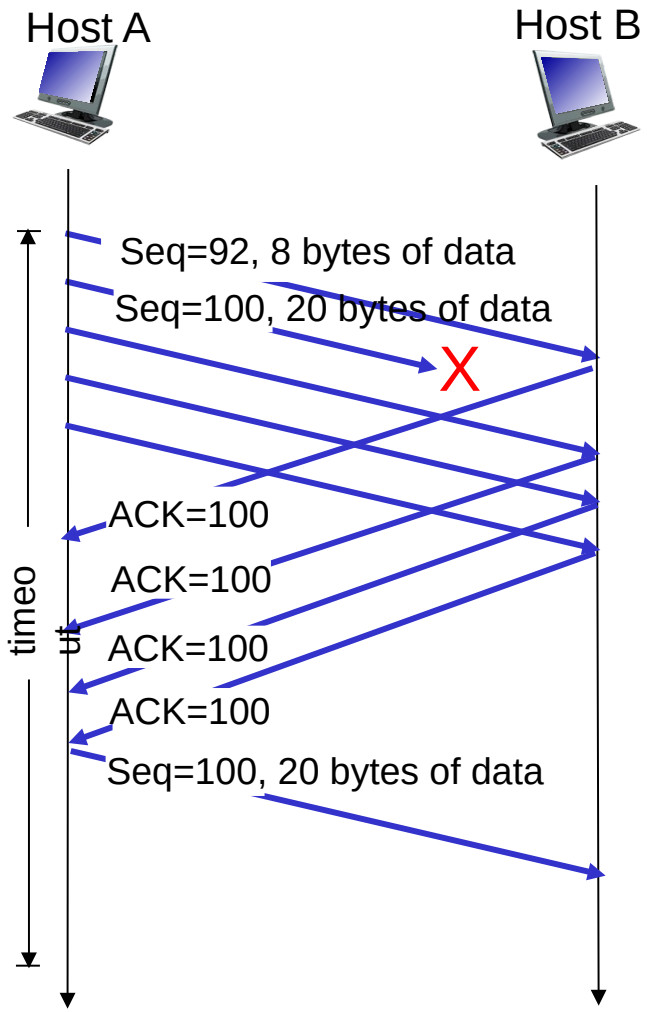
TCP fast retransmit

- O *timeout* é frequentemente relativamente longo:
  - muito atraso antes de reenviar o pacote perdido
- Detectar segmentos perdidos por meio de ACKs duplicados.
  - remetente muitas vezes envia muitos segmentos consecutivos
  - se o segmento for perdido, provavelmente haverá muitos ACKs duplicados.

TCP fast retransmit

- se o remetente receber 3 ACKs para os mesmos dados ("ACKs duplicados triplos"), reenvie o segmento com o menor número de seq #
  - é provável que o segmento não confirmado tenha se perdido, então não espere pelo tempo limite

TCP fast retransmit



Retransmissão rápida após o transmissor receber 3 ACKs duplicados

Geração de ACK [RFC 1122, RFC 2581]

Evento no receptor	Ação do receptor TCP
Segmento chega em ordem, não há lacunas, segmentos anteriores já aceitos	ACK retardado. Espera até 500 ms pelo próximo segmento. Se não chegar, envia ACK
Segmento chega em ordem, não há lacunas, um ACK atrasado pendente	Envia imediatamente um ACK cumulativo
Segmento chega fora de ordem, número de sequência chegou maior: *gap* detectado	Envia ACK duplicado, indicando número de sequência do próximo byte esperado
Chegada de segmento que parcial ou completamente preenche o *gap*	Reconhece imediatamente se o Segmento começa na borda inferior do *gap*

Resumo: Transferência confiável de dados

- O TCP cria um **serviço de transferência confiável de dados** sobre o serviço de melhor esforço do IP.

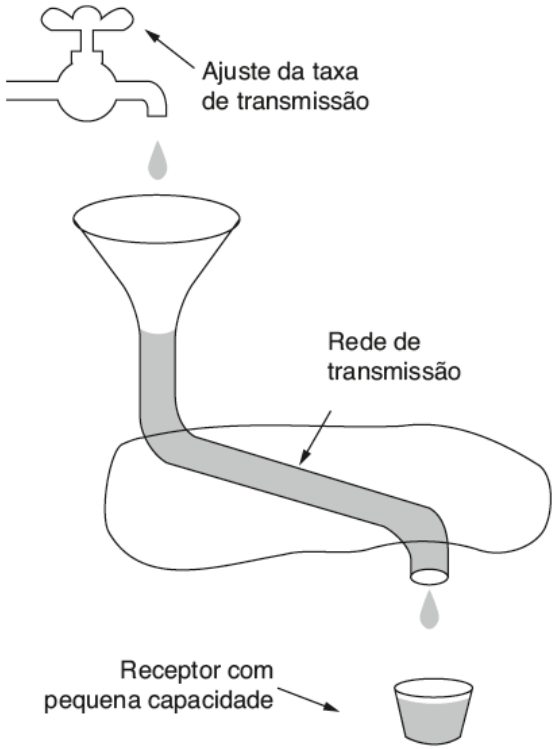
- O serviço de transferência garante que a cadeia de bytes é idêntica à cadeia de bytes enviada pelo sistema final que está do outro lado da conexão.

- Os procedimentos recomendados no [RFC 6298] para gerenciamento de temporizadores TCP utilizam apenas um único temporizador de retransmissão.

Controle de fluxo

- O TCP provê um **serviço de controle de fluxo** às suas aplicações, para eliminar a possibilidade de o remetente estourar o buffer do destinatário.
- **Controle de fluxo** é um serviço de compatibilização de velocidades.
- O TCP oferece serviço de controle de fluxo fazendo que o remetente mantenha uma variável denominada **janela de recepção**.

Regulação da taxa de envio



Uma rede de transmissão rápida e um receptor de baixa capacidade.

Controle de fluxo TCP

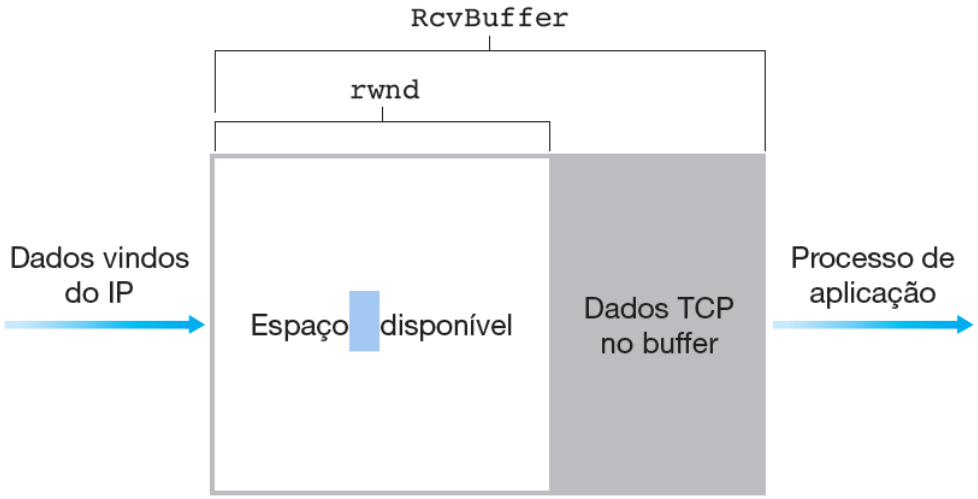
- Lado receptor da conexão TCP tem um *buffer* de recepção:
  - Espaço limitado em memória.
- Processo da aplicação pode ser lento na leitura do *buffer*
  - O remetente pode ser muito mais rápido

Controle de fluxo

remetente não estourará *buffer* do destinatário transmitindo muitos dados muito rapidamente

- *serviço de compatibilização de velocidades:* compatibiliza a taxa de envio do remetente com a de leitura da aplicação receptora

Controle de fluxo TCP: como funciona



(suponha que destinatário TCP não descarte segmentos fora de ordem)

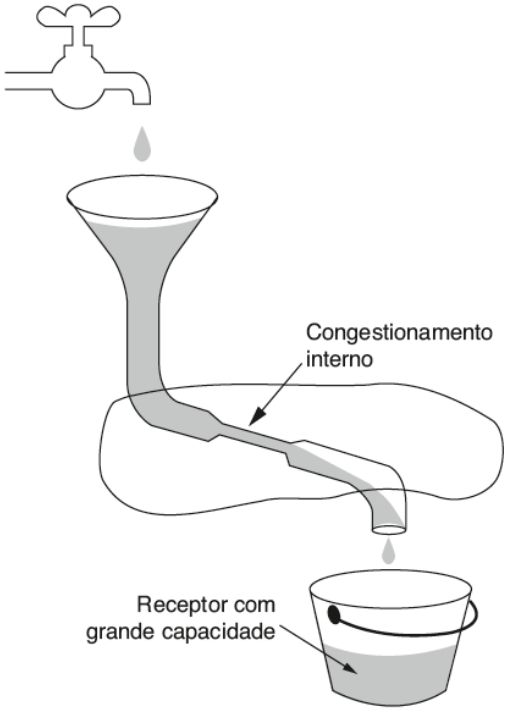
- espaço de buffer não usado:

= **rwnd** (slide 48)

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

- destinatário: anuncia espaço de buffer não usado incluindo valor de `rwnd` no cabeçalho do segmento
- remetente: limita # de bytes com ACK a `rwnd`
  - garante que buffer do destinatário não estoure

Controle de congestionamento

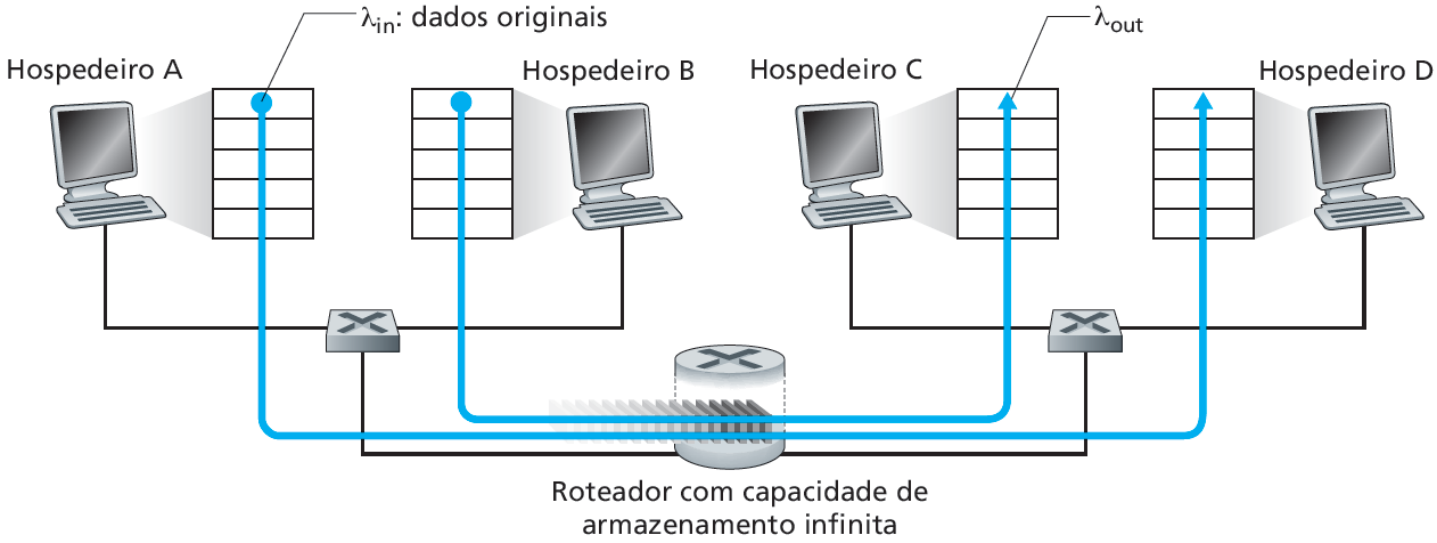


Uma rede de transmissão lenta e um receptor de alta capacidade.

Princípios de controle de congestionamento

- As causas e os custos do congestionamento:

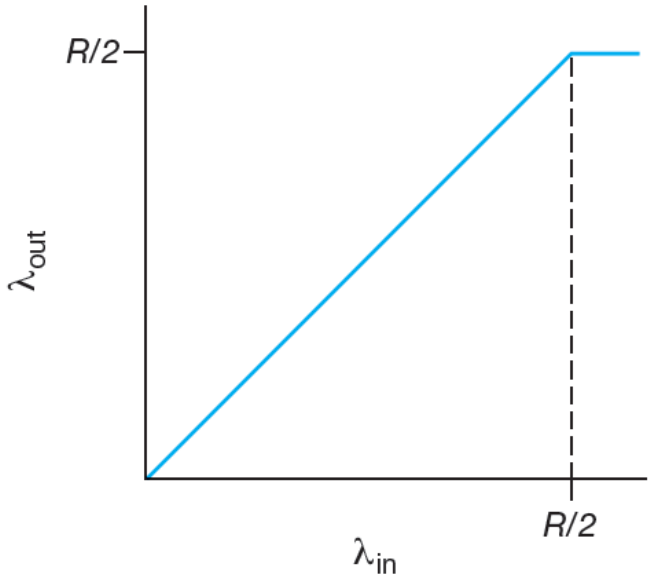
Cenário de congestionamento 1: duas conexões compartilhando um único roteador com número infinito de *buffers*.

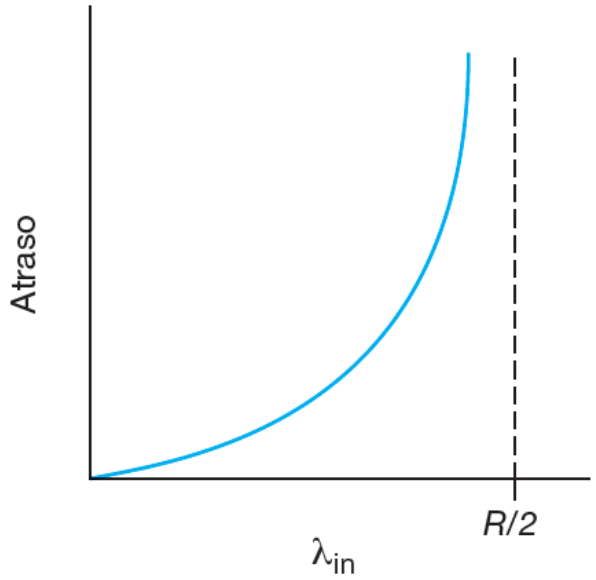


Princípios de controle de congestionamento

- As causas e os custos do congestionamento:

Cenário de congestionamento 1: vazão e atraso em função da taxa de envio do hospedeiro.





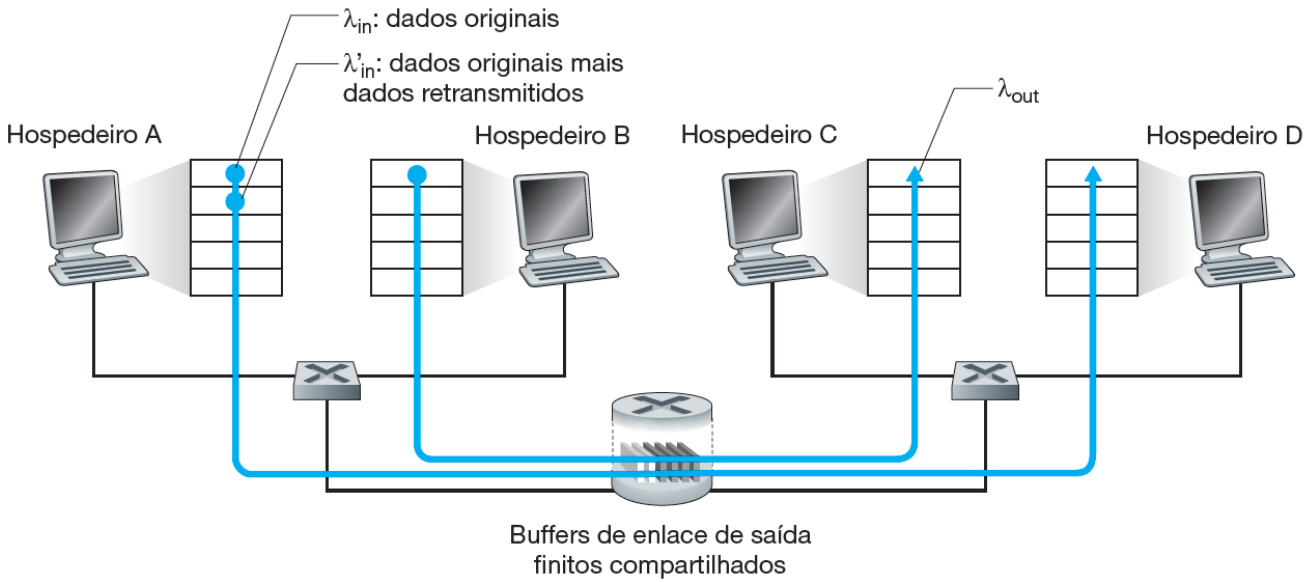
"custos" do congestionamento:

- Aumento de atrasos

Princípios de controle de congestionamento

- As causas e os custos do congestionamento:

Cenário 2: dois hospedeiros (com retransmissões) e um roteador com *buffers* finitos.

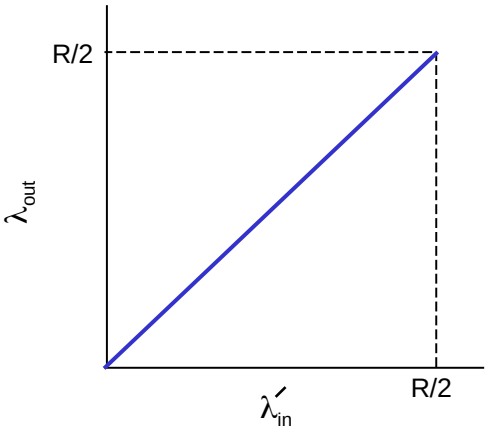


Princípios de controle de congestionamento

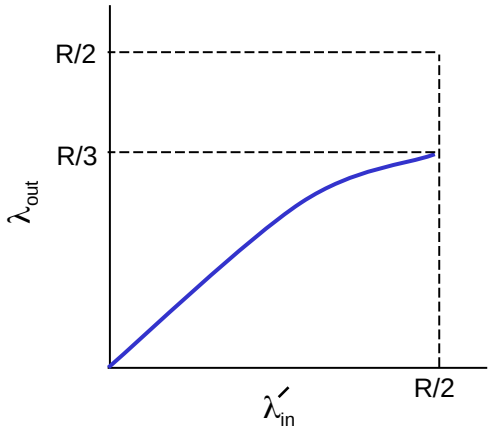
a) Caso ideal: $\lambda_{in} = \lambda_{out}$ (vazão)

b) retransmissão "perfeita" apenas quando há perda: $\lambda'_{in} > \lambda_{out}$

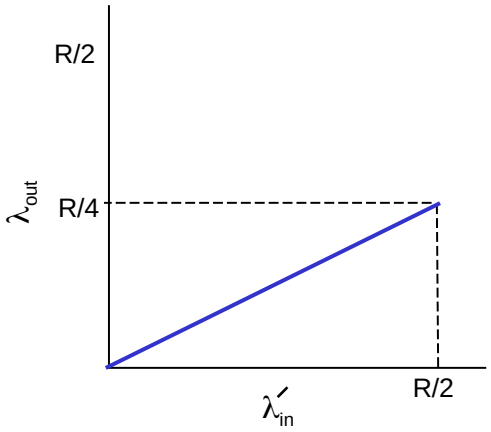
c) Pacote retransmitido sofre grandes atrasos (não perdido) torna λ'_{in} maior (que o caso perfeito) para o mesmo λ_{out}



a.



b.



c.

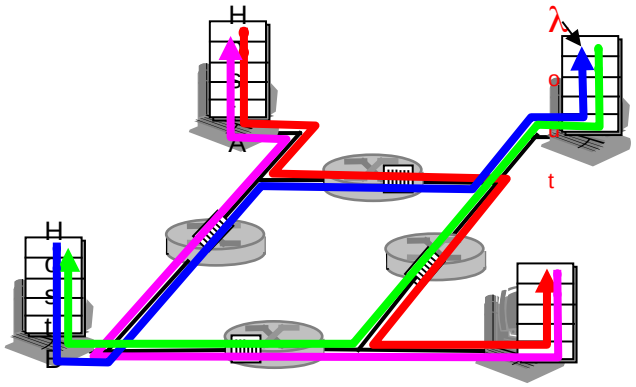
"custos" do congestionamento:

- mais trabalho (retransmissão) para determinada "vazão"
- retransmissões desnecessárias: enlace transporta várias cópias do pacote

Princípios de controle de congestionamento

As causas e os custos do congestionamento:

Cenário 3: quatro remetentes, roteadores com buffers finitos e trajetos com múltiplos roteadores.



outro “custo” do congestionamento:

- quando pacote é descartado, qualquer capacidade de transmissão anterior usada para esse pacote foi desperdiçada!

Mecanismos de controle de congestionamento

Controle de congestionamento fim a fim:

- A camada de rede não fornece nenhum suporte explícito à camada de transporte com a finalidade de controle de congestionamento.

Controle de congestionamento assistido pela rede:

- Os componentes da camada de rede (isto é, roteadores) fornecem retroalimentação específica de informações ao remetente a respeito do estado de congestionamento na rede. Não usual!

Controle de congestionamento no TCP

- A abordagem adotada pelo TCP é obrigar cada remetente a limitar a taxa à qual enviam tráfego para sua conexão como uma função do congestionamento de rede percebido:
  - Se um remetente TCP perceber que há pouco congestionamento no caminho entre ele e o destinatário, aumentará sua taxa de envio.
  - Se perceber que há congestionamento, reduzirá sua taxa de envio.

Controle de congestionamento no TCP

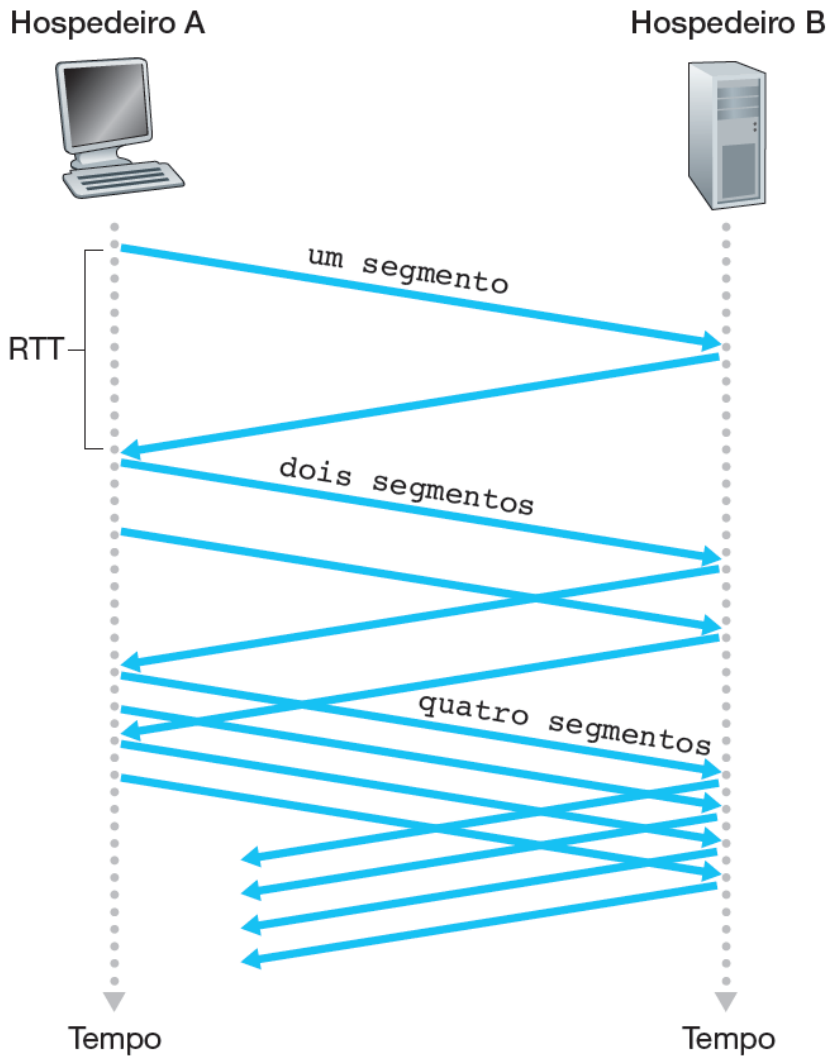
- Mas essa abordagem levanta três questões:

1. Como um remetente TCP limita a taxa pela qual envia tráfego para sua conexão?

2. Como um remetente TCP percebe que há congestionamento entre ele e o destinatário?

3. Que algoritmo o remetente deve utilizar para modificar sua taxa de envio como uma função do congestionamento fim a fim percebido?

Partida lenta



- Partida lenta TCP

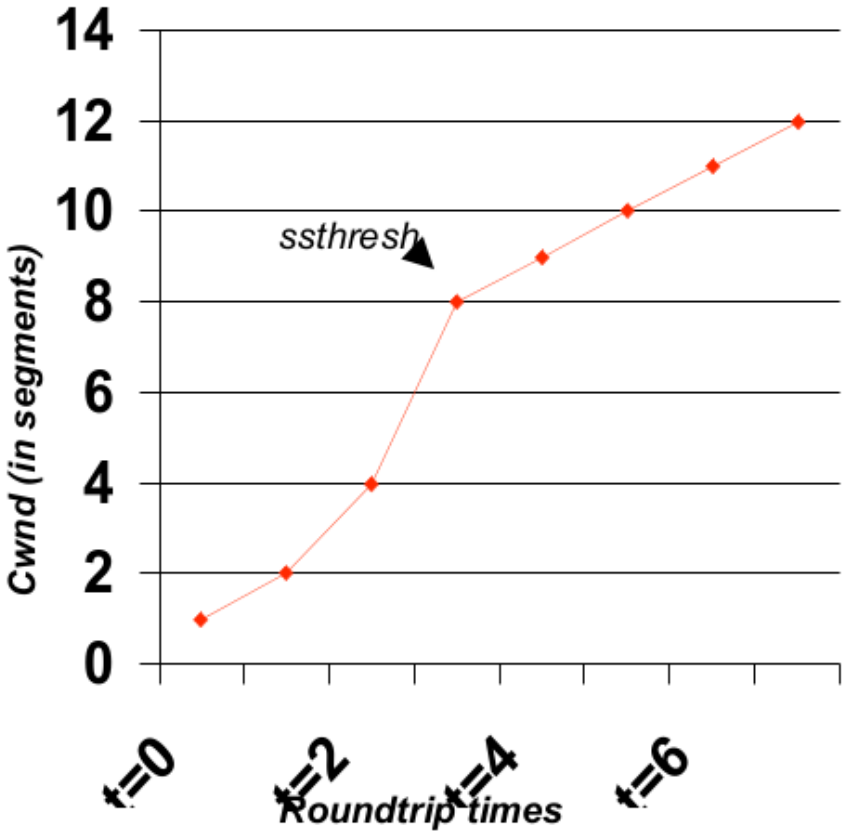
Controle de congestionamento TCP

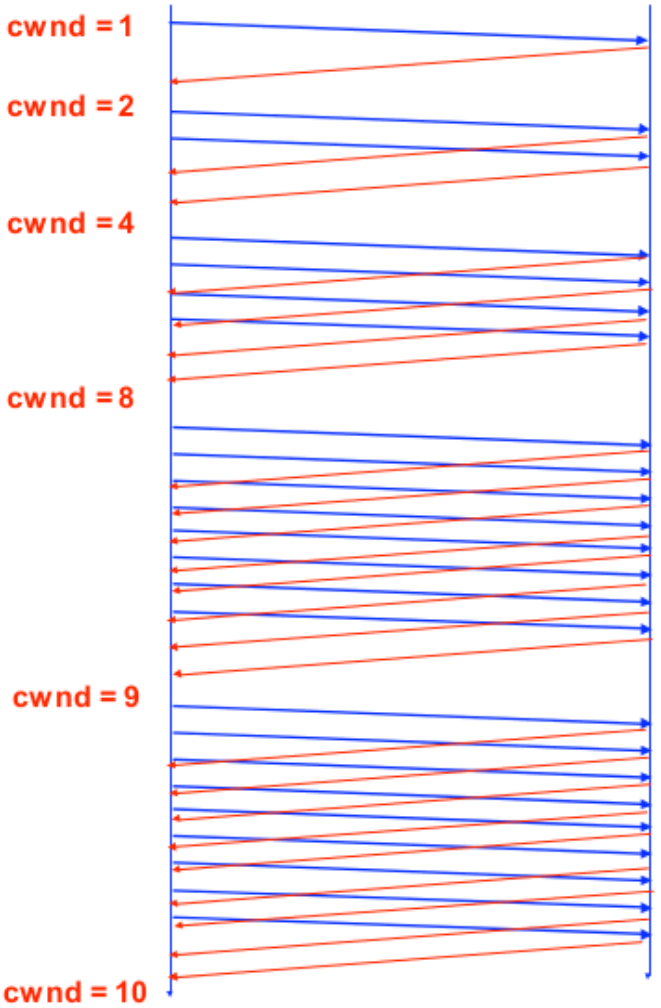
1) Cwnd = 1 MSS inicialmente
2) Partida lenta, a cada rodada redefine **cwnd = cwnd * 2**.
3) Perda de pacotes (esgotamento de temporização – problema grave) ==> **cwnd = 1** (Tahoe ou Reno)
4) Prevenção de congestionamento, após atingir sstresh: **cwnd = cwnd + 1**
5) Três acks duplicados (buraco no “fluxo” – problema brando), entra no estado prevenção de congestionamento e redefine **cwnd = 1** (Tahoe) ou **cwnd = sstresh** (Reno)
6) **Sstresh = cwnd/2**, quando são detectados problemas.

- **Quantidade de dados não reconhecidas no TCP**
- **= min(cwnd, RecWin)**

Example of Slow Start/Congestion Avoidance

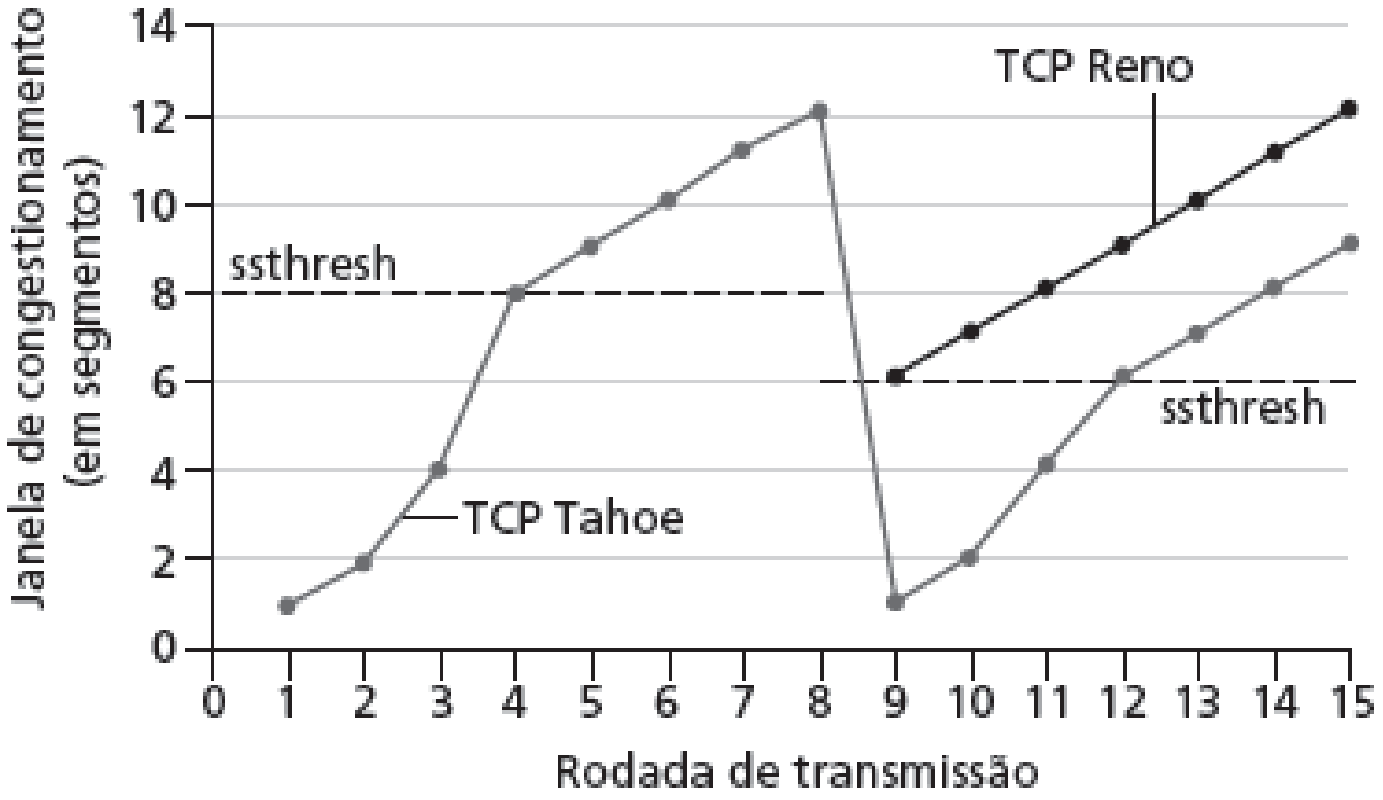
Assume that *ssthresh = 8*



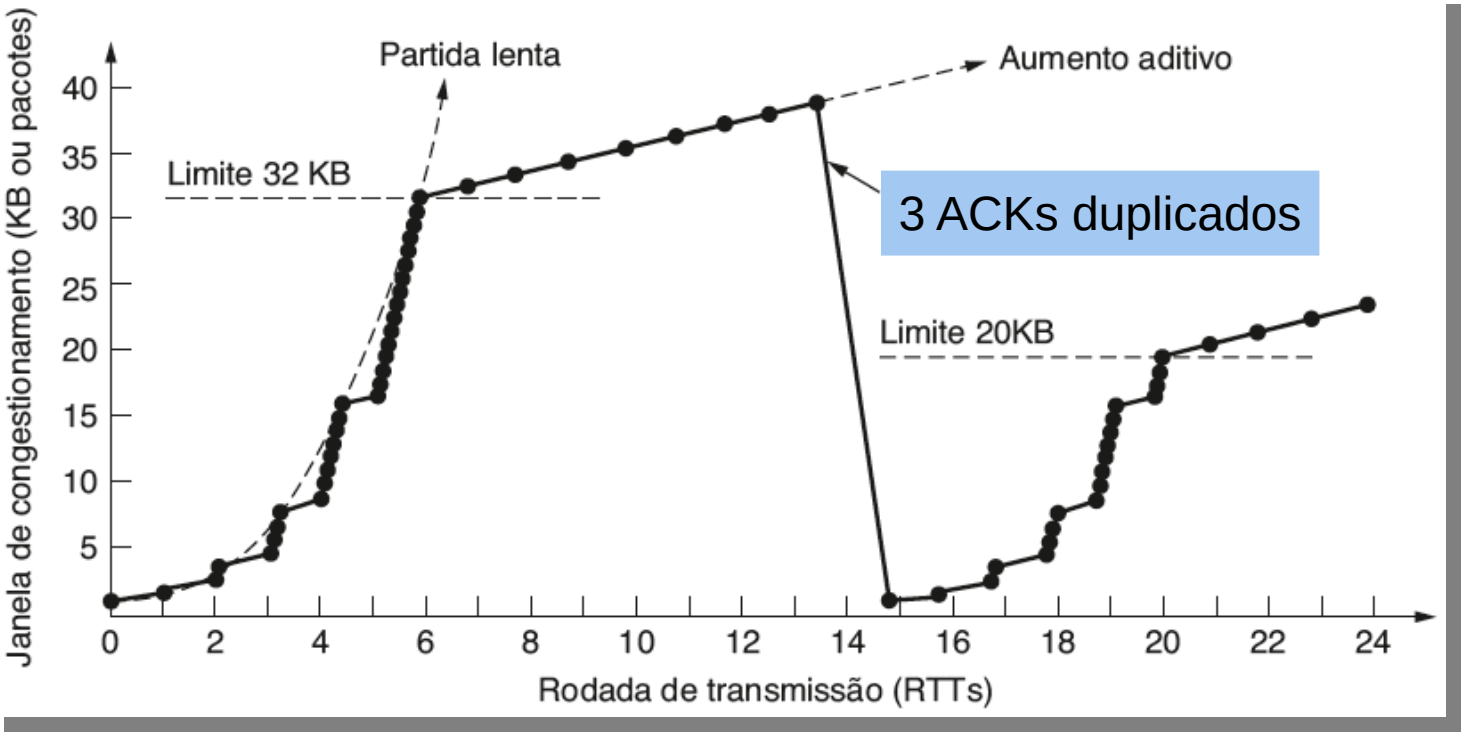


Recuperação rápida

- Evolução da janela de congestionamento do TCP (O antigo Tahoe e o moderno Reno). Considerando detecção de 3 ACKs duplicados na rodada 8.

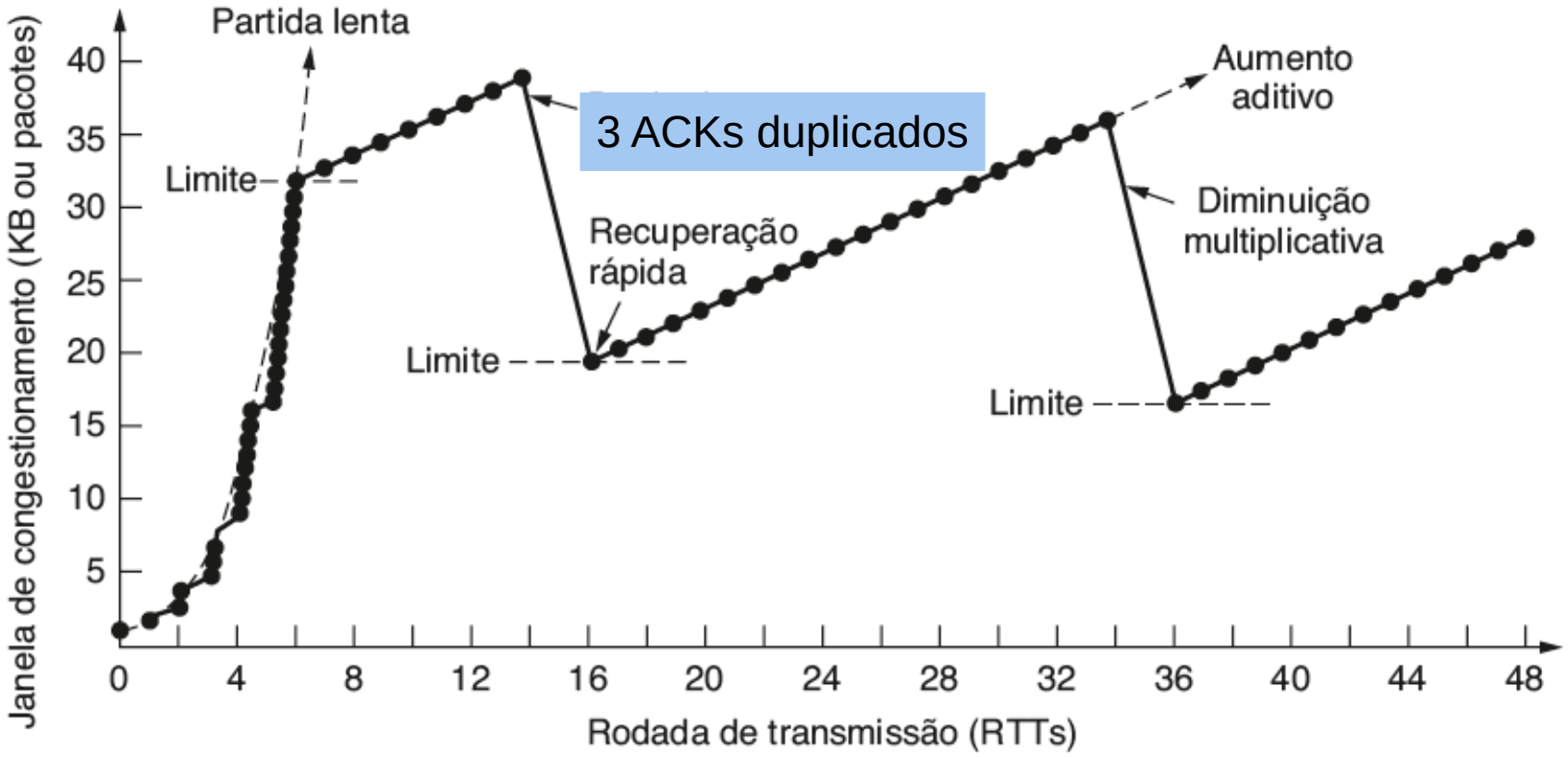


Controle de congestionamento TCP



Partida lenta seguida por aumento aditivo no **TCP Tahoe**.

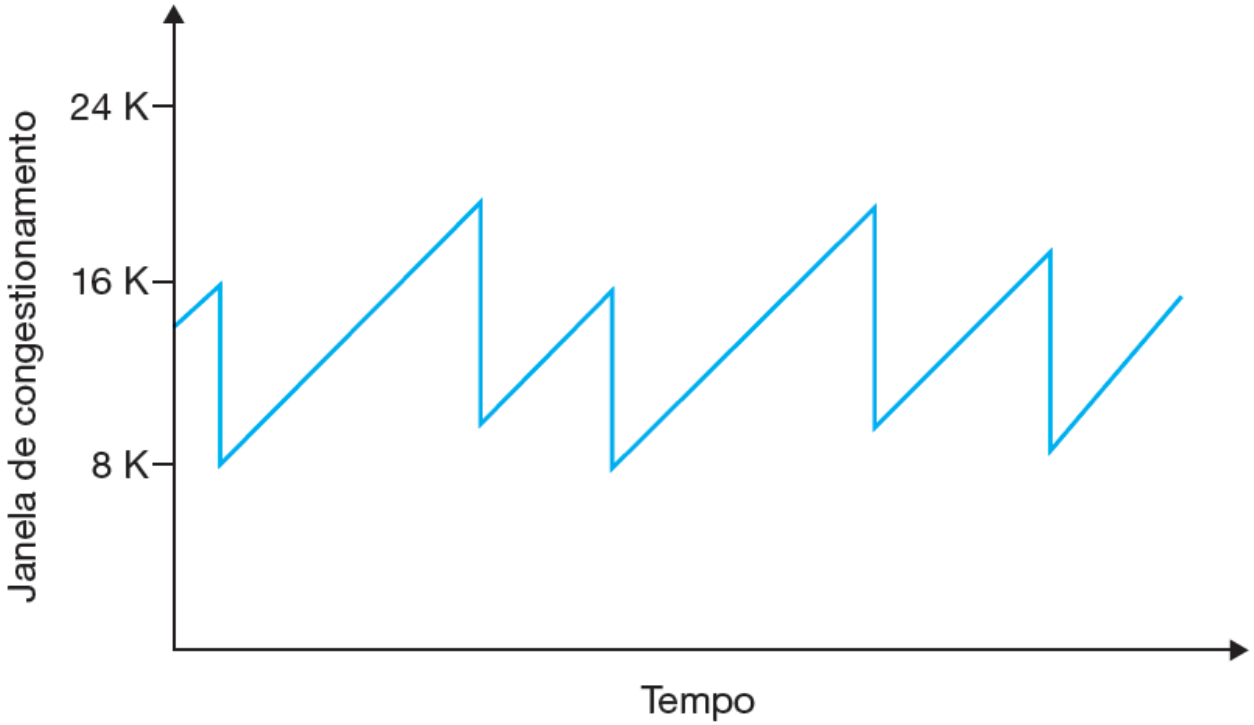
Controle de congestionamento TCP



Recuperação rápida e modelo dente serra do **TCP Reno**.

Controle de congestionamento no TCP: retrospectiva

- O controle de congestionamento AIMD (*Additive-increase*, *Multiplicative-Decrease*) faz surgir o comportamento semelhante a “dentes de serra”:
- https://media.pearsoncmg.com/aw/ecs_kurose_compnetwork_7/cw/content/interactiveanimations/tcp-congestion/index.html



TCP throughput

- O que é **throughput** médio do TCP como uma função do tamanho da janela e do RTT? ignore a partida lenta
- Deixe W ser o tamanho da janela quando ocorre perda.
- Quando a janela é **W**, o **throughput** é W/RTT
- Logo após a perda, a janela cai para W/2, e o **throughput** para W/2RTT
- **Throughput** médio: 0.75 W/RTT (75 % W/RTT)
- https://media.pearsoncmg.com/aw/ecs_kurose_compnetwork_7/cw/content/interactiveanimations/tcp-congestion/index.html

Futuro do TCP

- Exemplo: segmento de 1500 bytes, RTT de 100 ms, deseja 10 Gbps de *throughput*
- Requer tamanho de janela W = 83333 para os segmentos em trânsito
- Throughput em termos da taxa de perda:

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

- → L (Probabilidade de perda de segmentos) = $2 \cdot 10^{-10}$ Uau!
- São necessárias novas versões de TCP para alta velocidade!

• TCP: modelagem de latência

P.: Quanto tempo demora para receber um objeto de um servidor Web após enviar um pedido?

Ignorando o congestionamento, o atraso é influenciado por:

- Estabelecimento de conexão TCP
- Atraso de transferência de dados
- Partida lenta

Notação, hipóteses:

- Suponha um enlace entre o cliente e o servidor com taxa de dados R
- S: MSS (bits)
- O: tamanho do objeto (bits)
- Não há retransmissões (sem perdas e corrupção de dados)

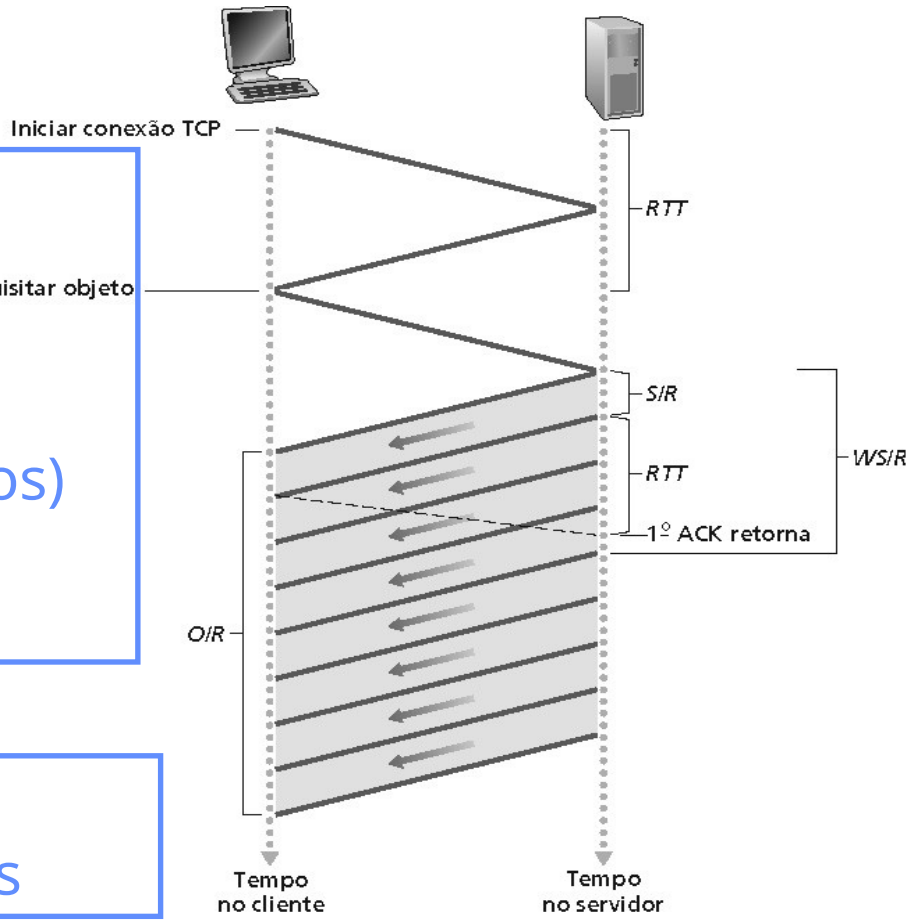
Tamanho da janela:

- Primeiro suponha: janela de congestionamento fixa, W segmentos
- Então janela dinâmica, modelagem *partida lenta*

• Janela de congestionamento fixa (1)

atraso = 2xRTT + O/R

Exemplo numérico:
R = 10Gbps
RTT = 0,3 s
S = 10000 bits
O = 100000 bits (10 segmentos)
atraso = 2xRTT + O/R
Atraso = 0,60001 s

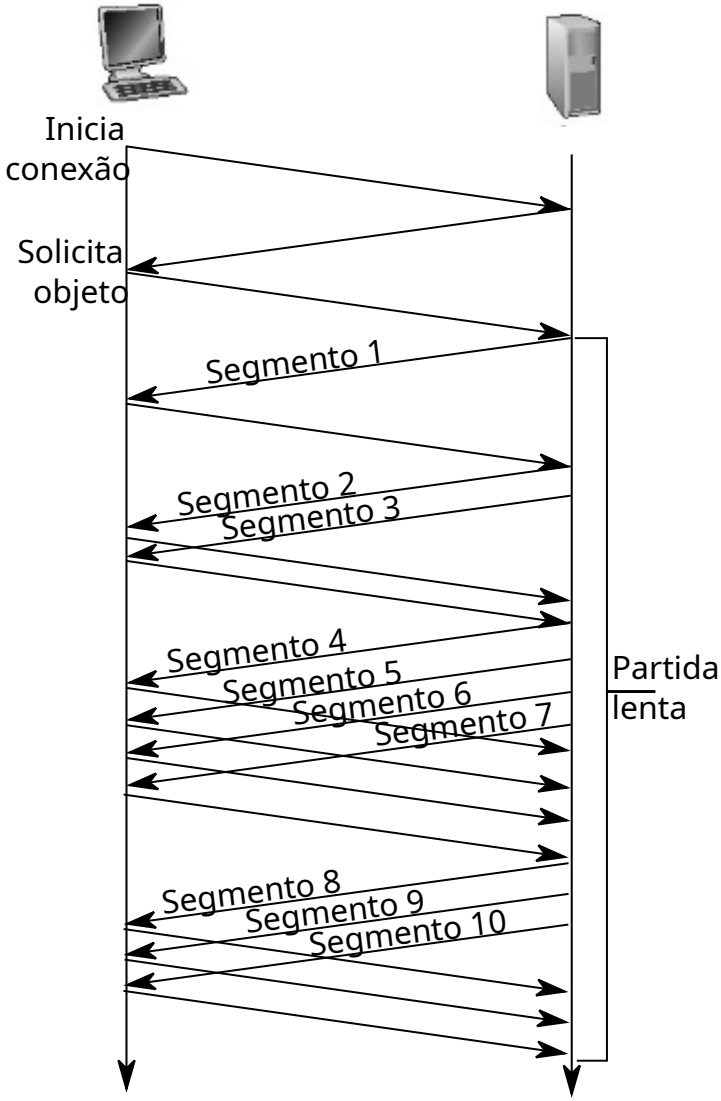


Portanto:
Vazão média = 166,67 kbps

• Considerando partida lenta

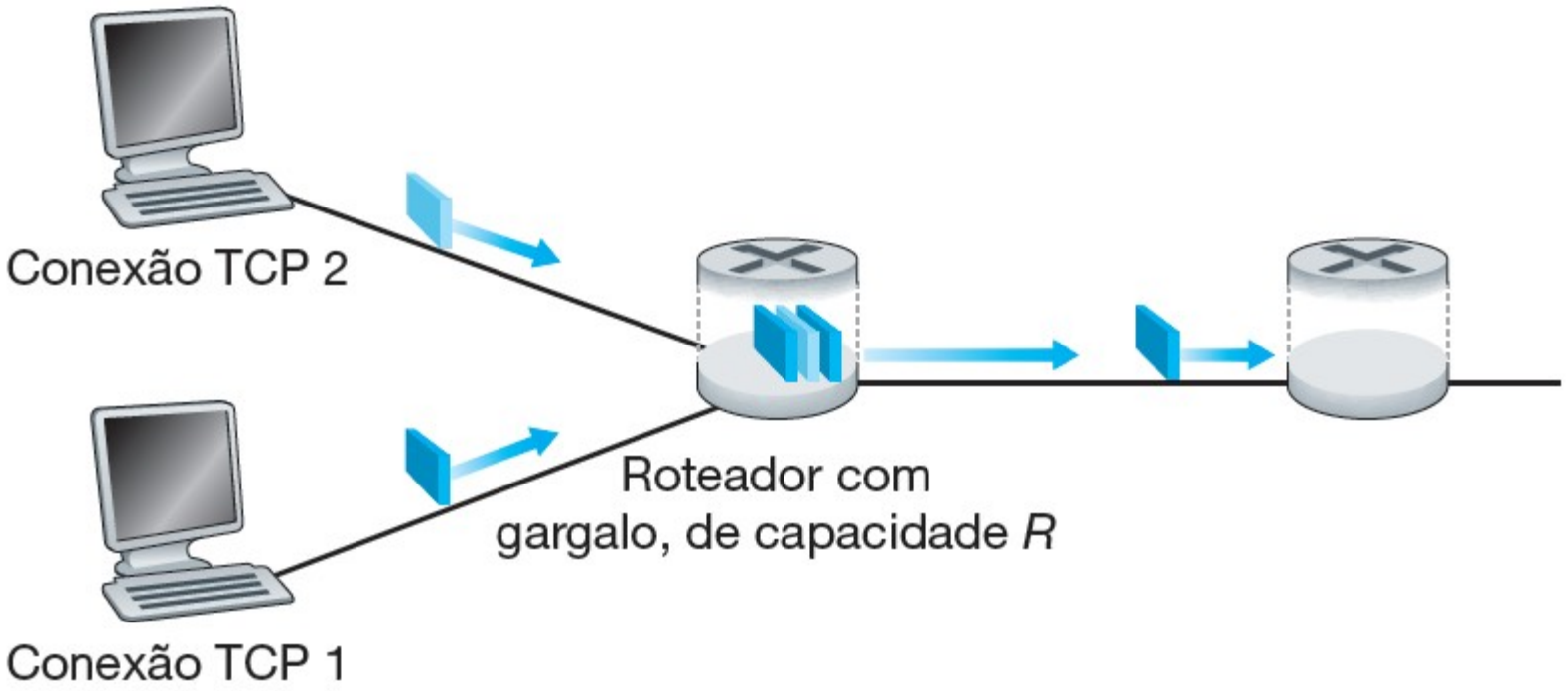
Exemplo numérico (cálculo aproximado):
R = 10Gbps
RTT = 0,3 s
S = 10000 bits
O = 100000 bits (10 segmentos)
Atraso = 5,5xRTT + O/R
Atraso = 1,65001 s

Portanto:
Vazão média = 60,6 kbps



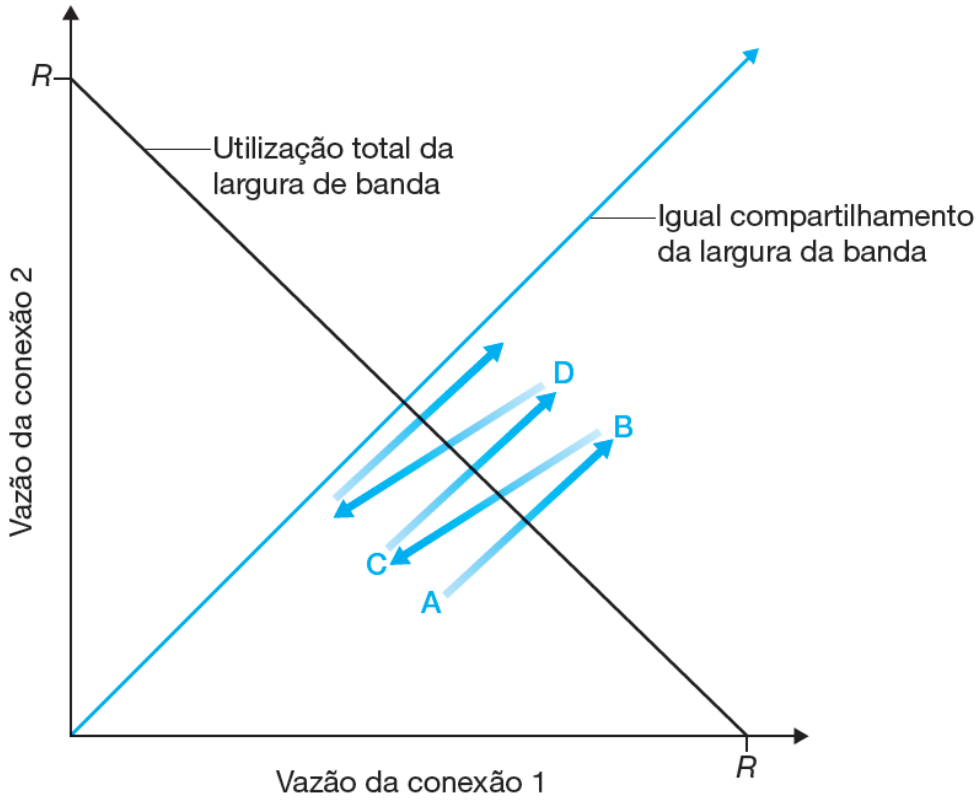
Equidade

- Duas conexões TCP compartilhando um único enlace congestionado



Equidade

- Vazão alcançada pelas conexões TCP 1 e TCP 2
  - Ambas conexões em algum momento estão com vazão **A**
  - Ambas vão aumentando a vazão até ocorrer perda – **B**
  - Sstresh cai pela metade e ambas migram para **C…..**



• Equidade

Eqüidade e UDP

- Aplicações multimedia normalmente não usam TCP
  - Não querem a taxa estrangulada pelo controle de congestionamento
- Em vez disso, usam UDP:
  - Trafega áudio/vídeo a taxas constantes, toleram perda de pacotes

Eqüidade e conexões TCP paralelas

- Nada previne as aplicações de abrirem conexões paralelas entre 2 hospedeiros.
  - Consequência?
- Web browsers fazem isso
- Exemplo: enlace de taxa R suportando 9 conexões;
  - Nova aplicação pede 1 TCP, obtém taxa de R/10
  - Outra aplicação pede 11 TCPs, obtém R/2!