# Capítulo 3: Camada de Transporte

## Metas do capítulo:

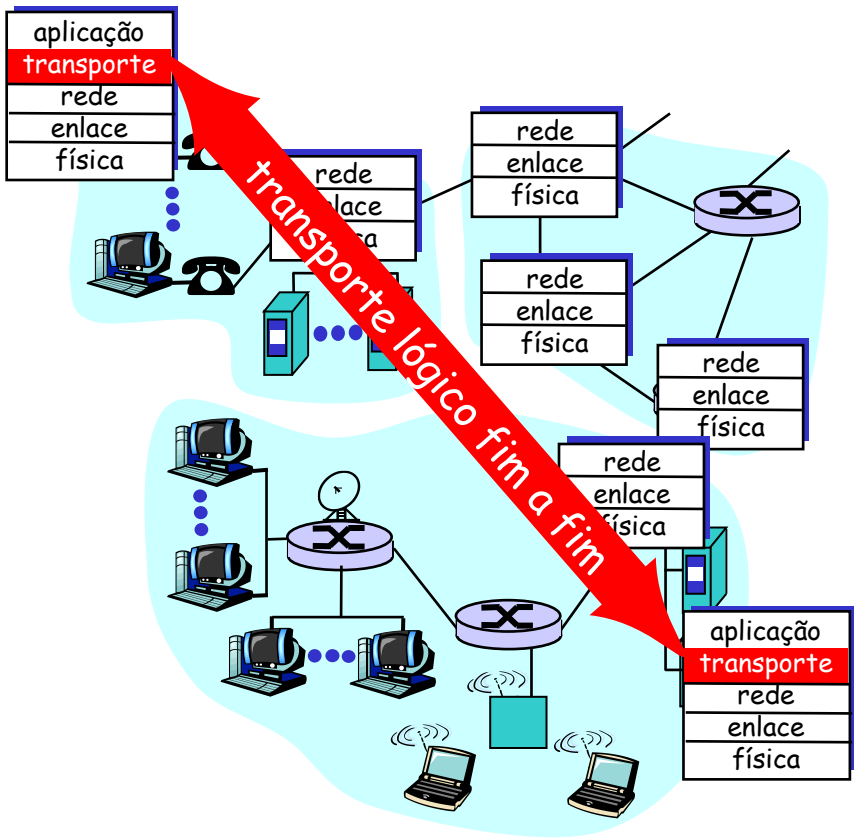
- compreender os princípios atrás dos serviços da camada de transporte:
  - multiplexação/ demultiplexação
  - transferência confiável de dados
  - controle de fluxo
  - controle de congestionamento
- instanciação e implementação na Internet dos protocolos de transporte
  - UDP
  - TCP

## Resumo do Capítulo:

- serviços da camada de transporte
- multiplexação/demultiplexação
- transporte sem conexão: UDP
- princípios de transferência confiável de dados
- transporte orientado a conexão: TCP
  - transferência confiável
  - controle de fluxo
  - gerenciamento de conexões
- princípios de controle de congestionamento
- controle de congestionamento em TCP

# Serviços e protocolos de transporte

- provê *comunicação lógica* entre processos de aplicação executando em hosts diferentes
- protocolos de transporte executam em sistemas terminais
  - ✓ emissor: fragmenta a mensagem da aplicação em segmentos e os envia para a camada de rede;
  - ✓ receptor: rearranja os segmentos em mesnagens e os transmite para a camada de aplicação;
- Mais de um protocolo de transporte disponível para as aplicações
  - ✓ Internet: TCP e UDP



# Camada de Rede versus Camada de Transporte

- *camada de rede :* comunicação lógica entre hosts ou sistemas;
- *camada de transporte:* comunicação lógica entre processos
  - ✓ depende dos serviços da camada de rede
  - ✓ extende os serviços da camada de rede

**Analogia de Casas:**

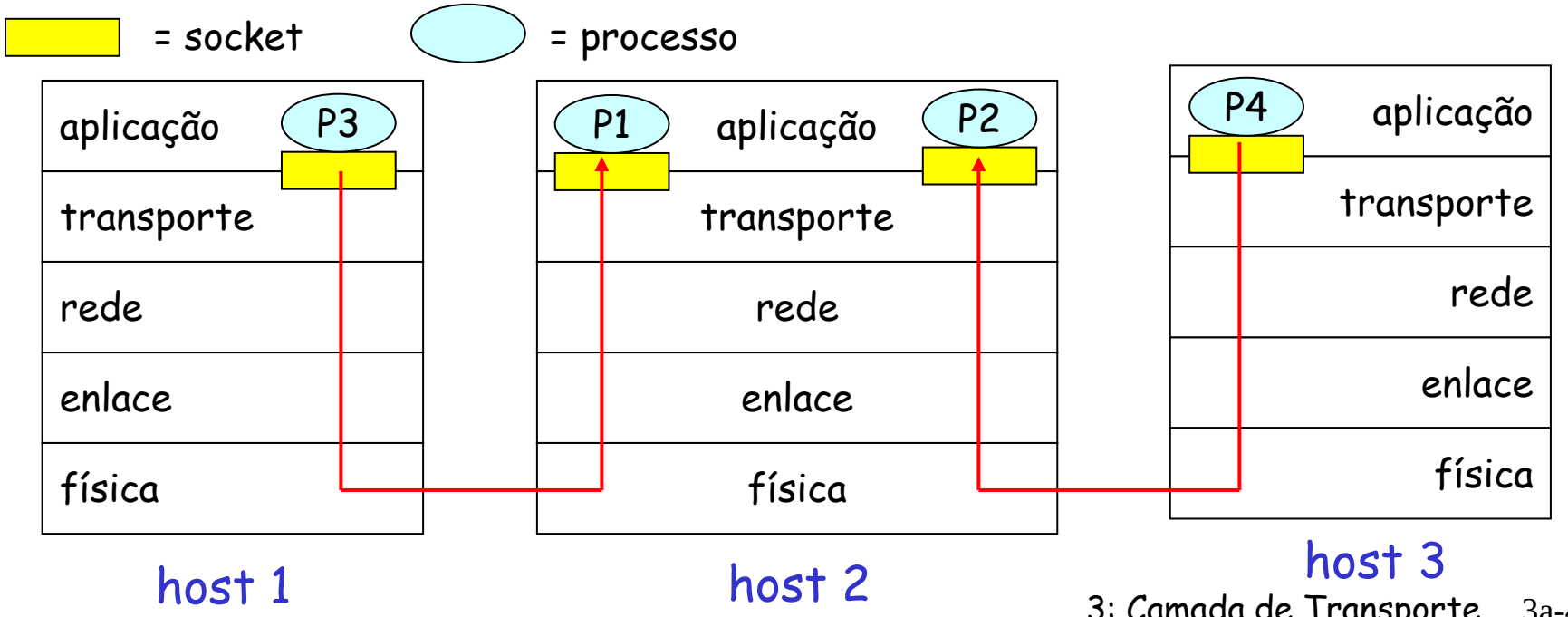
*12 crianças enviando cartas para 12 crianças*

- processos = crianças
- Mensagens da aplicação = cartas no envelope
- hosts = casas
- Protolo de transporte = Ann e Bill
- Protocolo da camada de rede = serviço de correio

# Multiplexação/demultiplexação

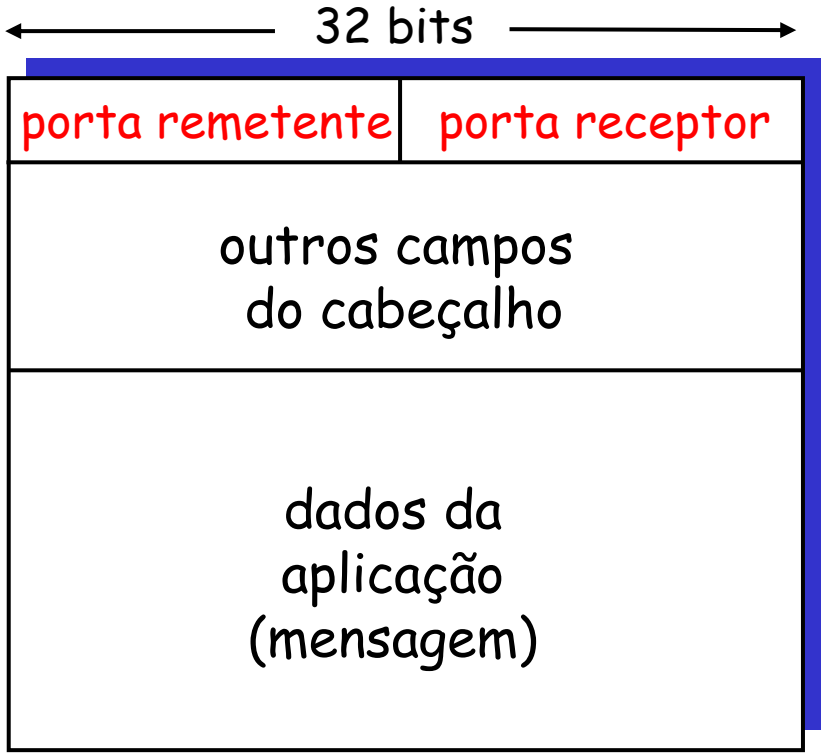
**Demultiplexação no receptor:**
entrega de segmentos recebidos para os processos da camada de apl corretos

**Multiplexação no emissor:**
juntar dados de múltiplos processos de apl, envelopando dados com cabeçalho (usado depois para demultiplexação)



# Como demultiplexação funciona

- host recebe os datagramas IP
  - ✓ Cada datagrama tem um endereço IP de origem e de destino;
  - ✓ Cada datagrama carrega 1 segmento da camada de transporte;
  - ✓ Cada segemento tem um número de porta de origem e um de destino
  - ✓ lembrete: número de porta bem conhecido para aplicações específicas
- host usa o endereço IP e o número de porta para direcionar o segmento para o socket apropriado;




formato de segmento TCP/UDP

# Demultiplexação não orientada a conexão

- Cria sockets com os números de porta

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

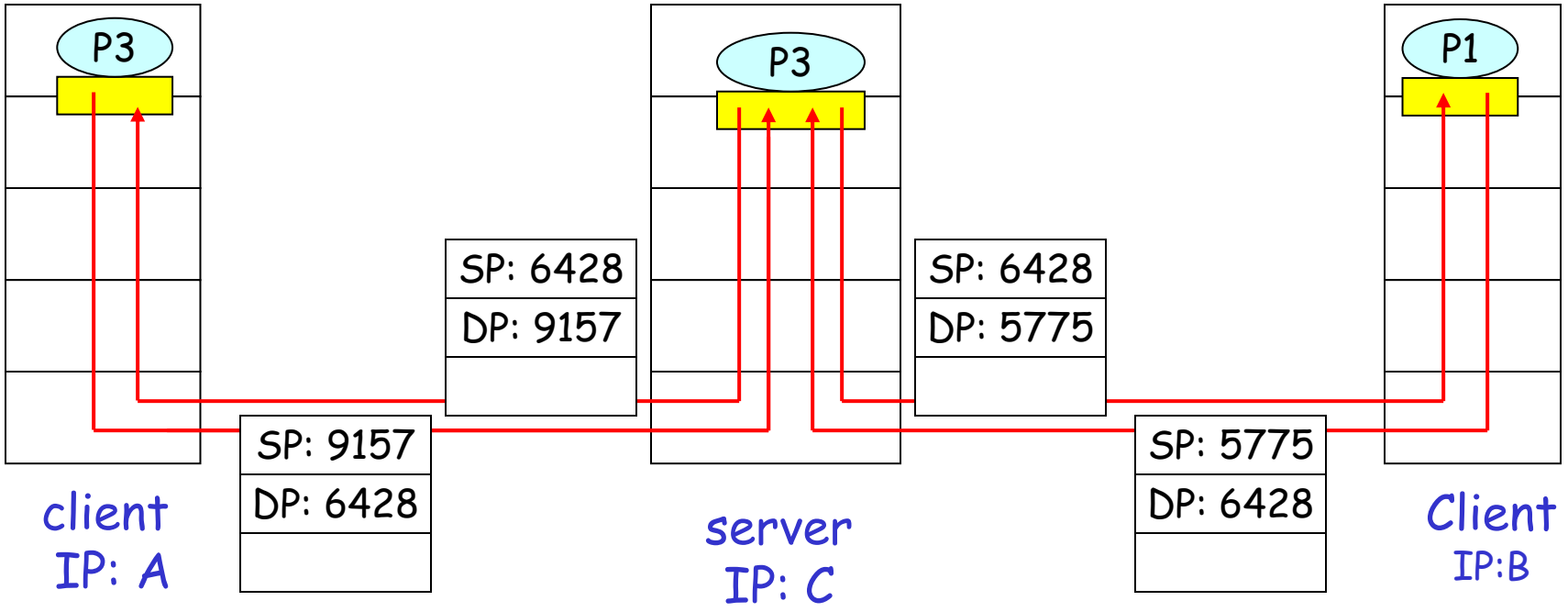
- Socket UDP identificado pela 2-tupla:

(endereço IP destino, no porta destino)

- Quando o host recebe o segmetno UDP:
  - ✓ Verifica o número da porta de destino no segmento
  - ✓ Direciona o segmento UDP para o socket correspondente ao número da porta;
- Datagramas IP com diferentes endereços IP de origem e/ou números de porta de origem são direcionados para o mesmo socket

# Demultiplexação não orientada a conexão (cont)

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

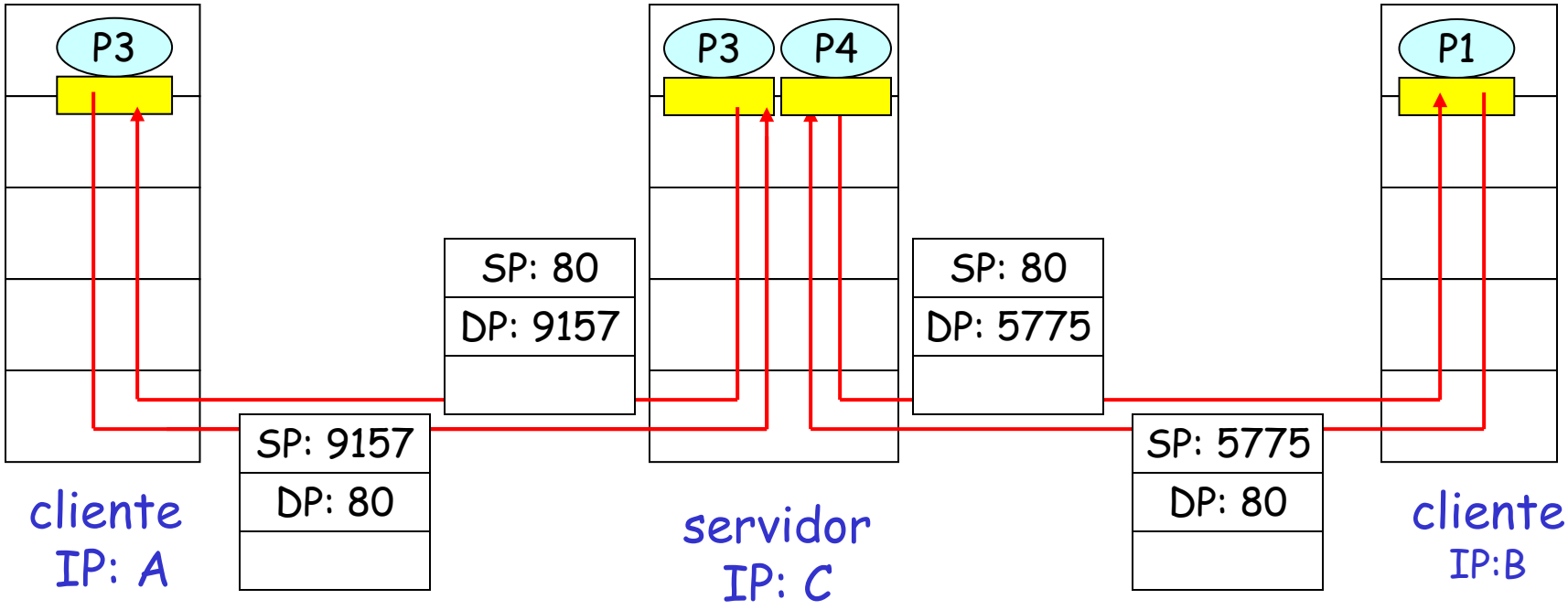



SP provê "endereço de retorno"

# Demultiplexação orientada a conexão

- Identificação do socket, 4-tupla:
  - endereço IP de origem
  - número da porta de origem
  - endereço IP de destino
  - número da porta de destino
- Hosts receptor usa estes valores da tupla para direcionar os segmentos para o socket apropriado
- host servidor deve suportar múltiplos sockets TCP simultaneamente:
  - Cada socket é identificado por sua 4-tupla
- servidores Web tem diferentes sockets para cada cliente
  - HTTP não persistente tem diferentes sockets para cada requisição

# Demultiplexação orientada a conexão (cont)



# UDP: User Datagram Protocol [RFC 768]

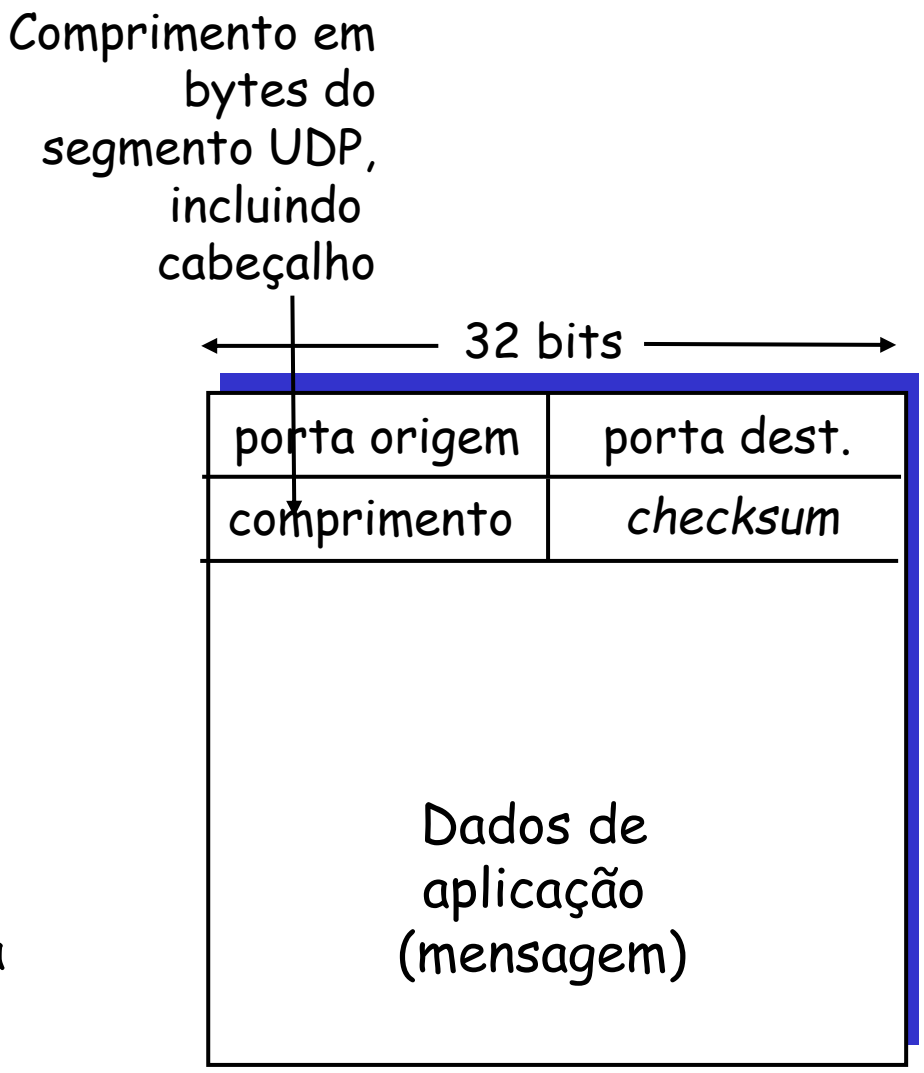
- Protocolo de transporte da Internet mínimo, "sem frescura",
- Serviço "melhor esforço", segmentos UDP podem ser:
  - ✓ perdidos
  - ✓ entregues à aplicação fora de ordem do remesso
- *sem conexão:*
  - ✓ não há "setup" UDP entre remetente, receptor
  - ✓ tratamento independente para cada segmento UDP

**Por quê existe um UDP?**

- elimina estabelecimento de conexão (o que pode causar retardo)
- simples: não se mantém "estado" da conexão no remetente/receptor
- pequeno cabeçalho de segmento
- sem controle de congestionamento: UDP pode transmitir o mais rápido possível

# Mais sobre UDP

- muito utilizado para apls. de meios contínuos (voz, vídeo)
  - ✓ tolerantes de perdas
  - ✓ sensíveis à taxa de transmissão
- outros usos de UDP (por quê?):
  - ✓ DNS (nomes)
  - ✓ SNMP (gerenciamento)
- transferência confiável com UDP: incluir confiabilidade na camada de aplicação
  - ✓ recuperação de erro específica à apl.!




UDP segment format

# Checksum UDP

**Meta:** detecta "erro" (e.g., bits invertidos) no segmento transmitido

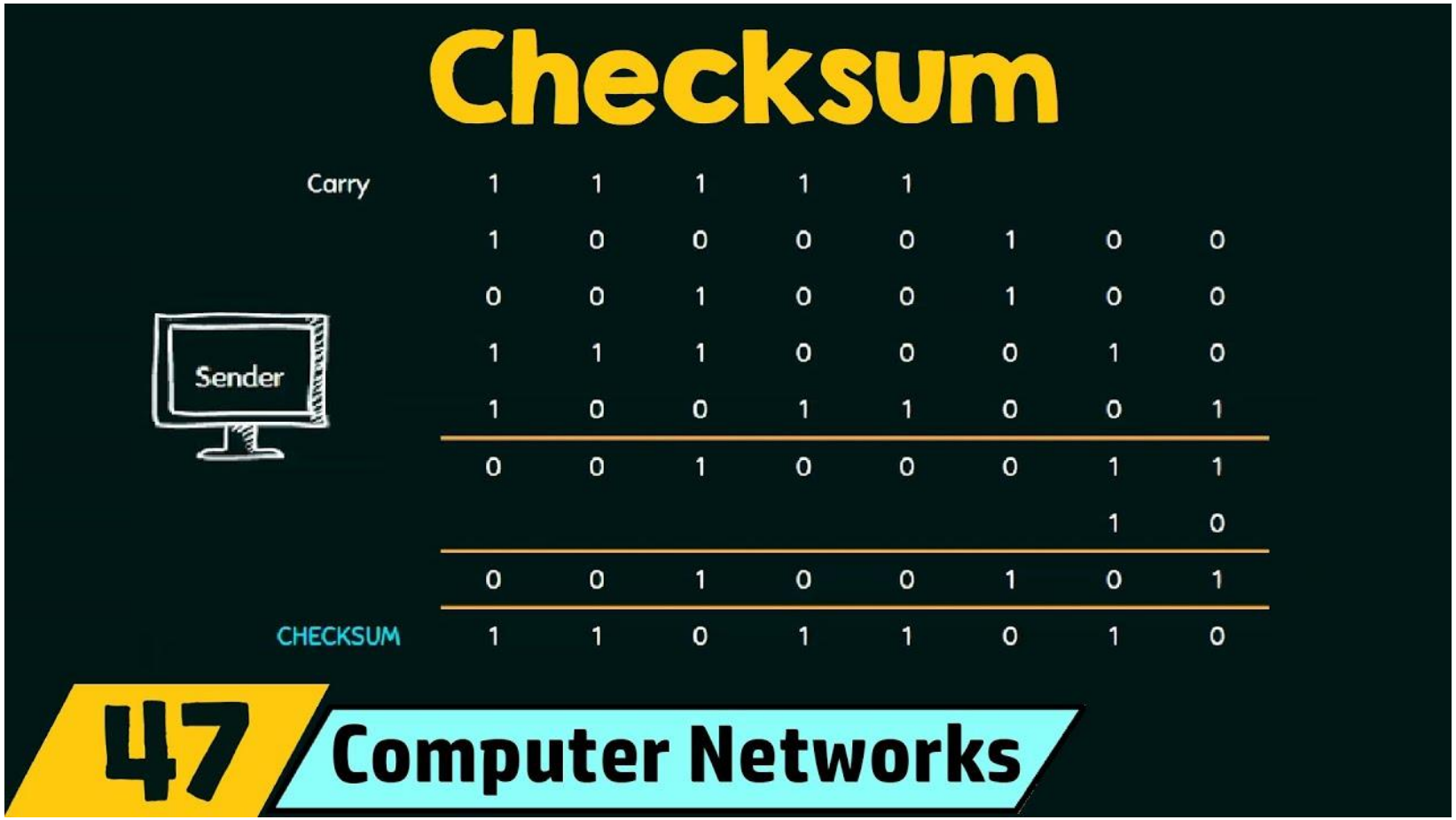
**Remetente:**

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

**Receptor:**

- calcula checksum do segmento recebido
- verifica se checksum computado é zero:
  - ✓ NÃO - erro detectado
  - ✓ SIM - nenhum erro detectado. *Mas ainda pode ter erros?* Veja depois ….

# Checksum

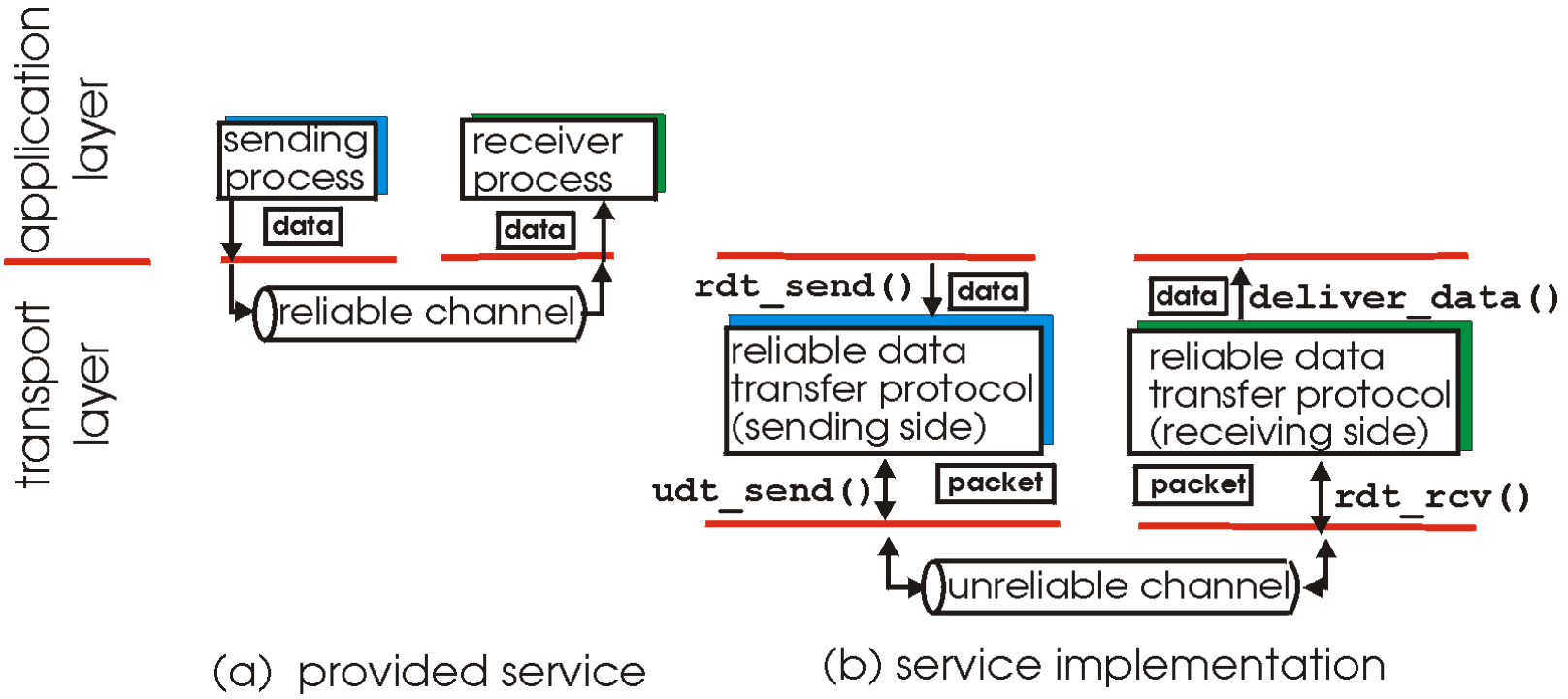

Checksum
Carry
1 1 1 1 1
1 0 0 0 0 1 0 0
0 0 1 0 0 1 0 0
1 1 1 0 0 0 1 0
1 0 0 1 1 0 0 1
0 0 1 0 0 0 1 1
1 0
0 0 1 0 0 1 0 1
CHECKSUM
1 1 0 1 1 0 1 0
Sender
47 Computer Networks

# UDP Lite


0
15
31
Porta de Origem
Porta de Destino
Checksum Coverage
Checksum
Área de Dados

# Princípios de Transferência confiável de dados (rdt)

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

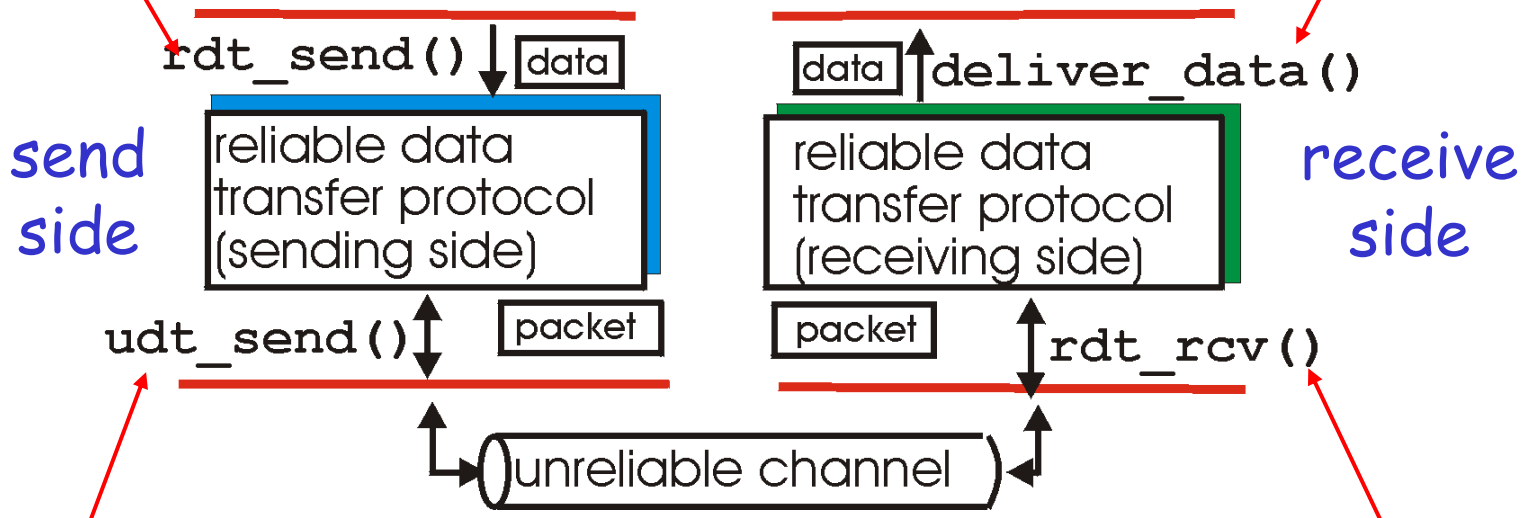



- características do canal não confiável determinam a complexidade de um protocolo de transferência confiável de dados (rdt)

# Transferência confiável de dados (rdt): como começar

**`rdt_send()`:** chamada de cima, (p.ex.,pela apl.). Dados recebidos p/ entregar à camada sup. do receptor

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




**`udt_send()`:** chamada por rdt, p/ transferir pacote pelo canal ñ confiável ao receptor

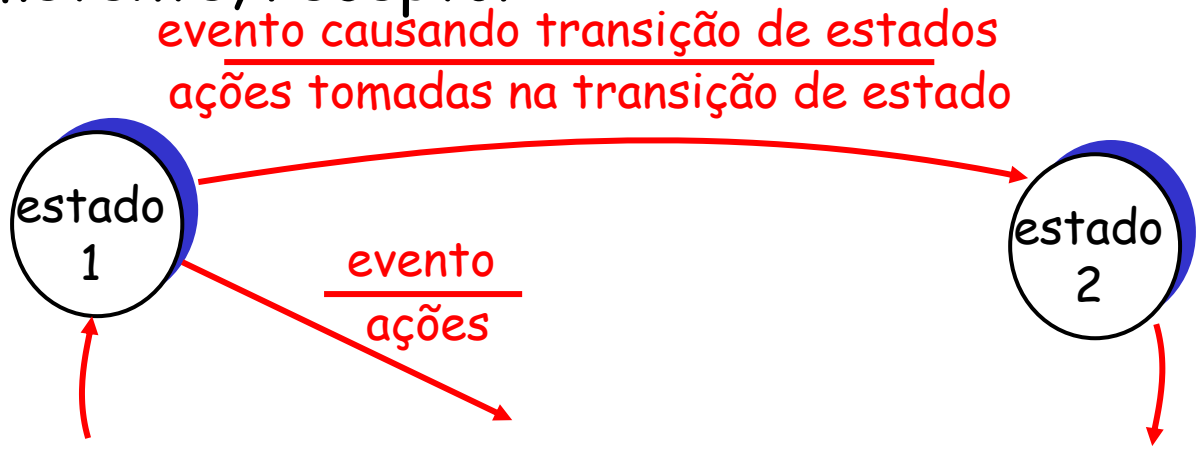
**`rdt_rcv()`:** chamada quando pacote chega chega no lado receptor do canal

# Transferência confiável de dados (rdt): como começar

Etapas:

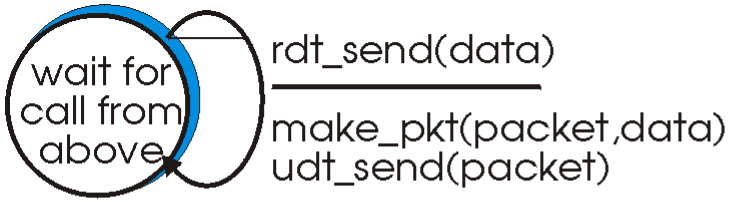
- desenvolver incrementalmente os lados remetente, receptor do protocolo RDT
- considerar apenas fluxo unidirecional de dados
  - ✓ mas info de controle flui em ambos os sentidos!
- Usar máquinas de estados finitos (FSM) p/ especificar remetente, receptor

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

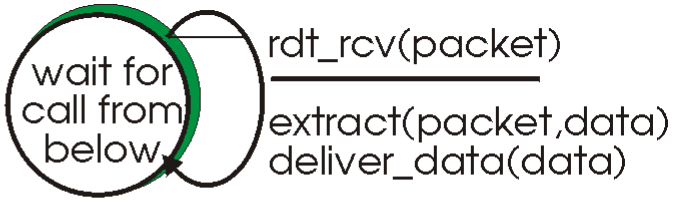


# Rdt1.0: transferência confiável usando um canal confiável

- canal subjacente perfeitamente confiável
  - ✓ não tem erros de bits
  - ✓ não tem perda de pacotes
- FSMs separadas para remetente, receptor:
  - ✓ remetente envia dados pelo canal subjacente
  - ✓ receptor recebe dados do canal subjacente




(a) rdt1.0: sending side

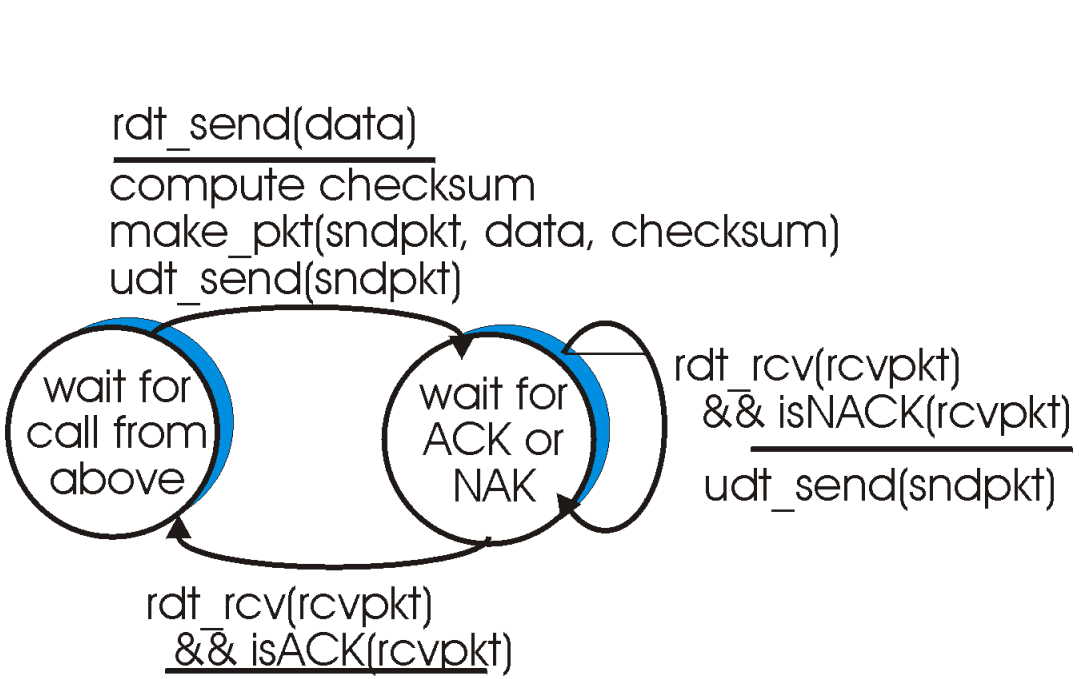



(b) rdt1.0: receiving side

# Rdt2.0: canal com erros de bits

- canal subjacente pode inverter bits no pacote
  - lembre-se: checksum UDP pode detectar erros de bits
- *a questão:* como recuperar dos erros?
  - *reconhecimentos (ACKs):* receptor avisa explicitamente ao remetente que pacote chegou bem
  - *reconhecimentos negativos (NAKs):* receptor avisa explicitamente ao remetente que pacote tinha erros
  - remetente retransmite pacote ao receber um NAK
  - cenários humanos usando ACKs, NAKs?
- novos mecanismos em **`rdt2.0`** (em relação ao **`rdt1.0`**):
  - deteção de erros
  - realimentação pelo receptor: msgs de controle (ACK,NAK) receptor->remetente

# rdt2.0: especificação da FSM

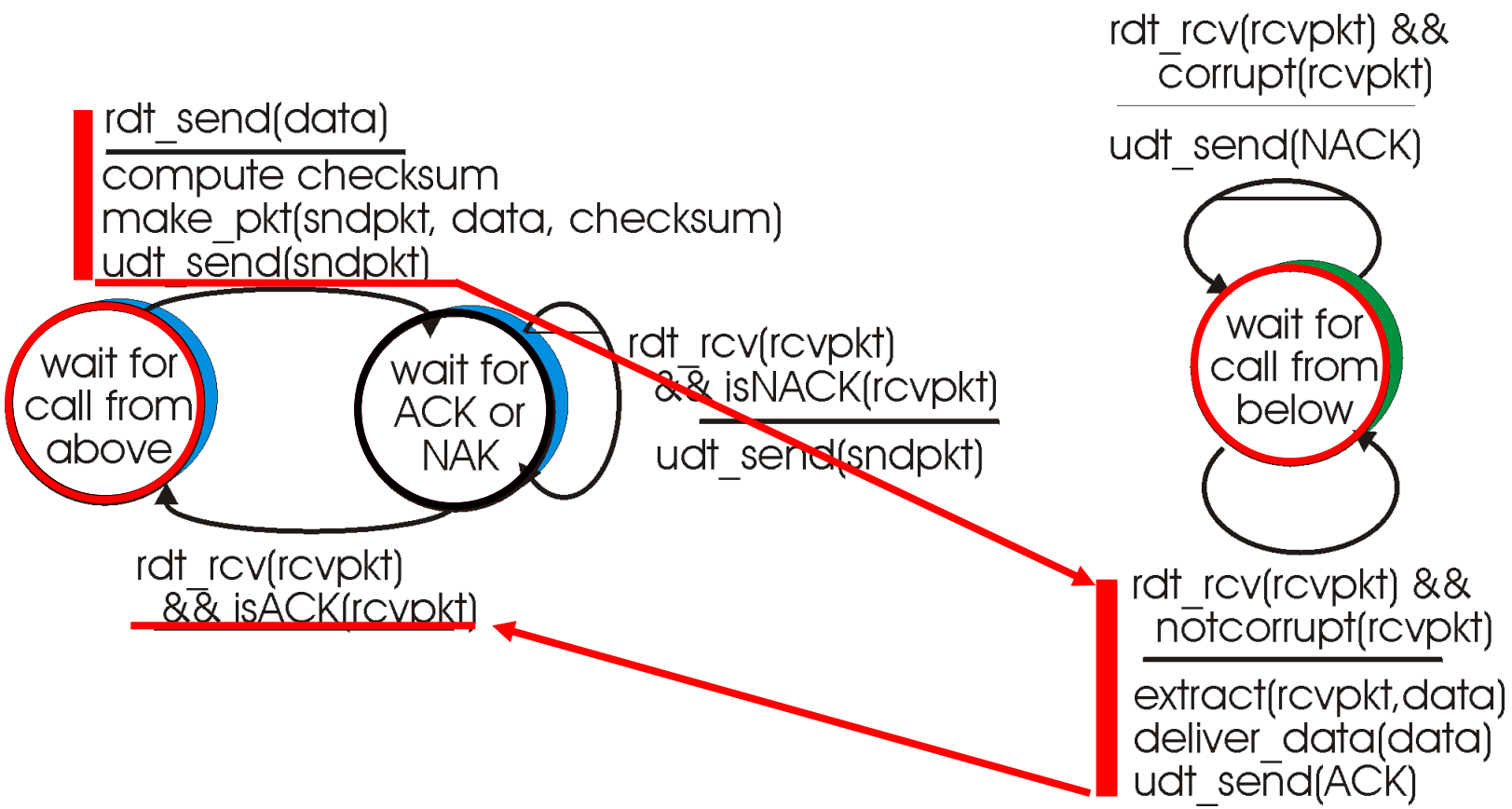




FSM do remetente

FSM do receptor

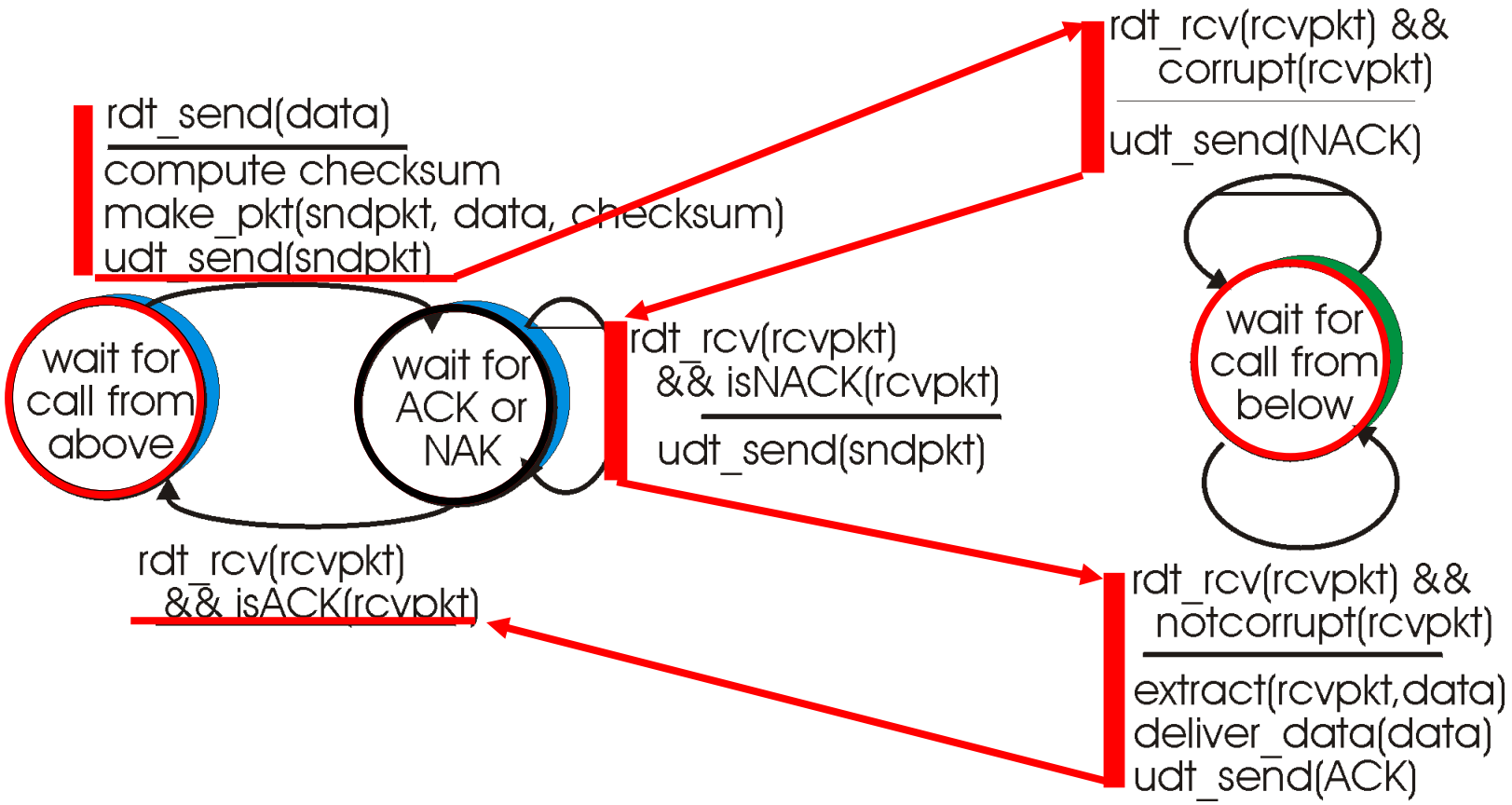
# rdt2.0: em ação (sem erros)




FSM do remetente

FSM do receptor

# rdt2.0: em ação (cenário de erro)




FSM do remetente

FSM do receptor

# rdt2.0 tem uma falha fatal!

## O que acontece se ACK/NAK com erro?

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

## O que fazer?

- remetente usa ACKs/NAKs p/ ACK/NAK do receptor? E se perder ACK/NAK do remetente?
- retransmitir, mas pode causar retransmissão de pacote recebido certo!

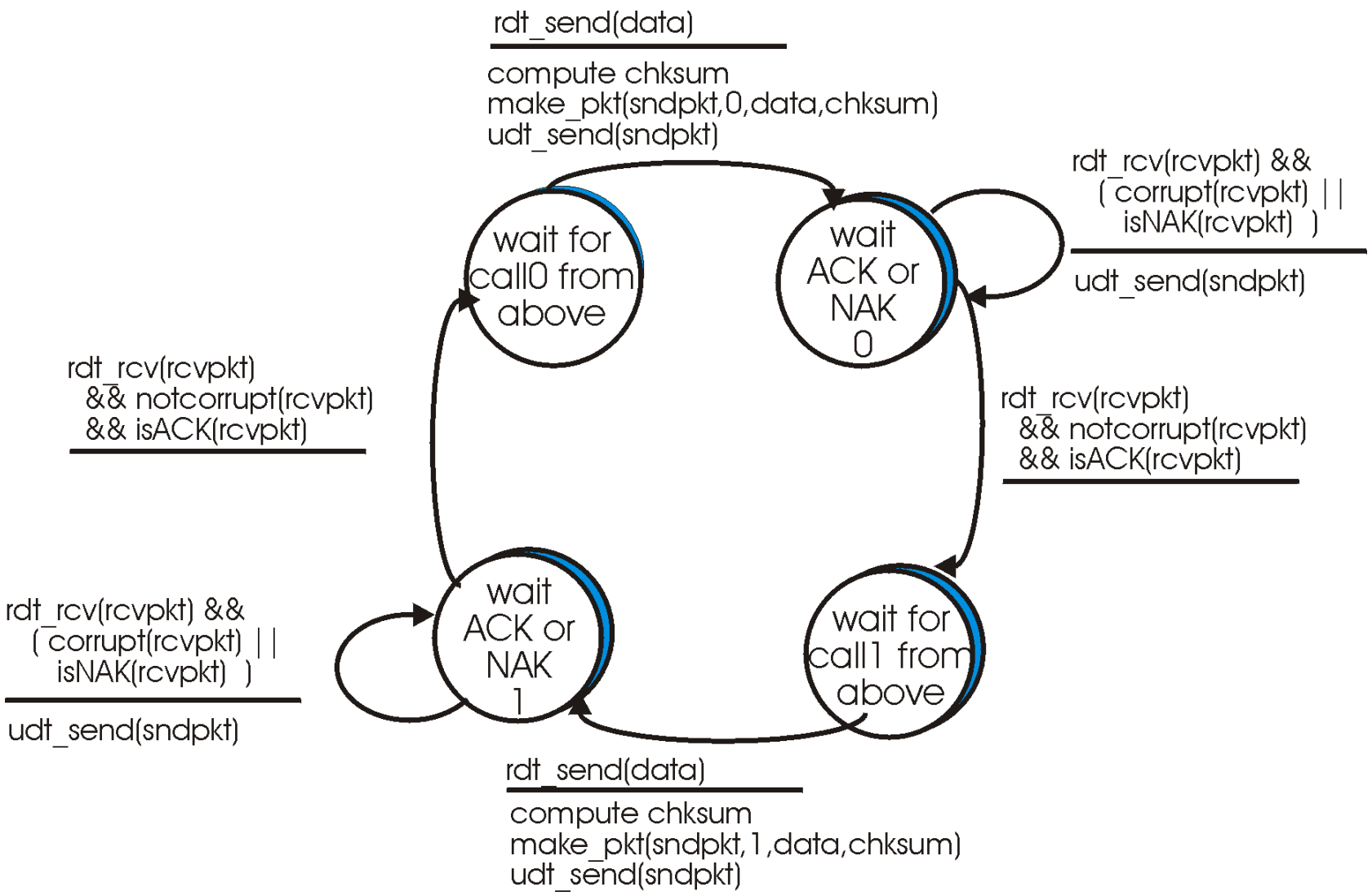
## Lidando c/ duplicação:

- remetente inclui *número de seqüência* p/ cada pacote
- remetente retransmite pacote atual se ACK/NAK recebido com erro
- receptor descarta (não entrega) pacote duplicado

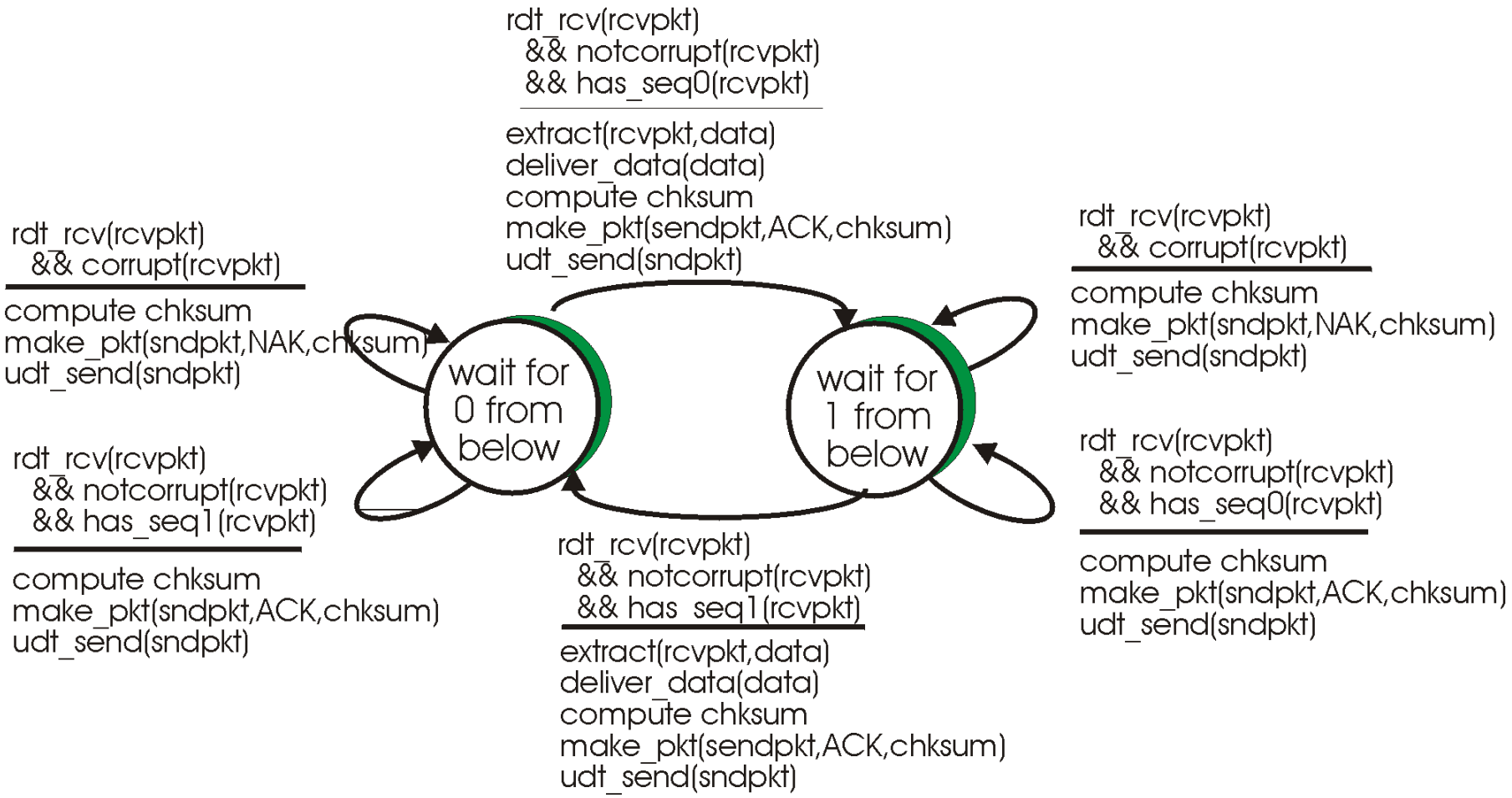
para e espera

Remetente envia um pacote, e então aguarda resposta do receptor

# rdt2.1: remetente, trata ACK/NAKs c/ erro



# rdt2.1: receptor, trata ACK/NAKs com erro



# rdt2.1: discussão

**Remetente:**

- no. de seq no pacote
- bastam dois nos. de seq. (0,1). Por quê?
- deve checar se ACK/NAK recebido tinha erro
- duplicou o no. de estados
  - estado deve "lembrar" se pacote "corrente" tem no. de seq. 0 ou 1

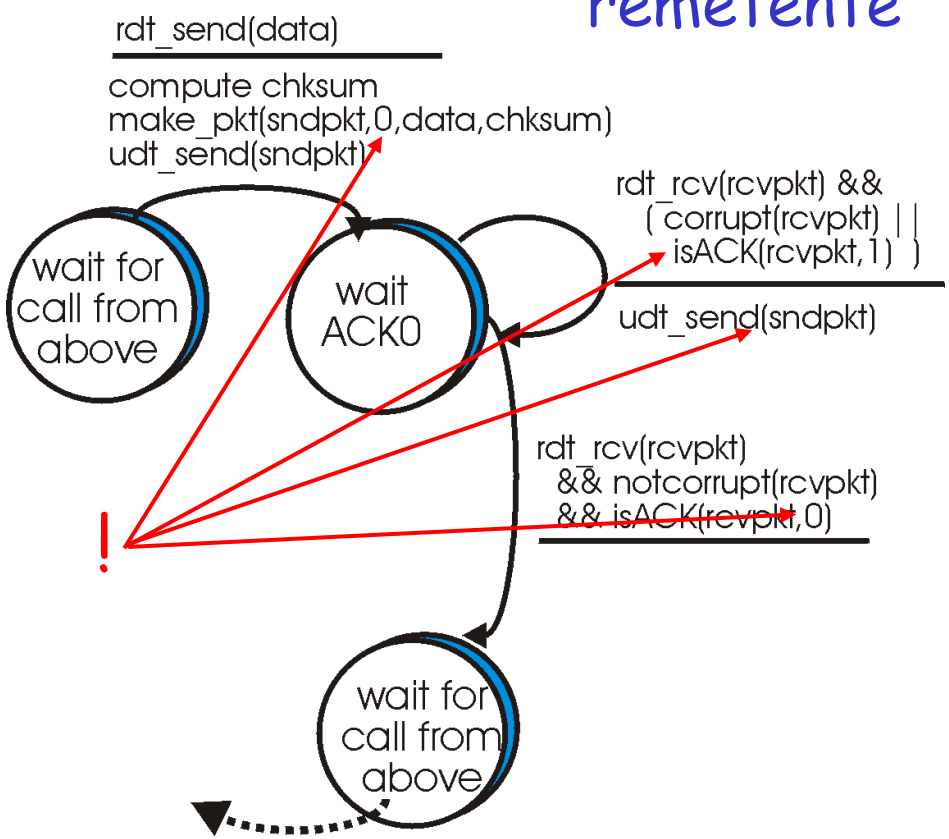
**Receptor:**

- deve checar se pacote recebido é duplicado
  - estado indica se no. de seq. esperado é 0 ou 1
- note: receptor não tem como saber se último ACK/NAK foi recebido bem pelo remetente

# rdt2.2: um protocolo sem NAKs

- mesma funcionalidade que rdt2.1, só com ACKs
- ao invés de NAK, receptor envia ACK p/ último pacote recebido bem
  - receptor deve incluir *explicitamente* no. de seq do pacote reconhecido
- ACK duplicado no remetente resulta na mesma ação que o NAK: *retransmite pacote atual*

FSM do remetente



# rdt3.0: canais com erros e perdas

Nova suposição: canal subjacente também pode perder pacotes (dados ou ACKs)

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

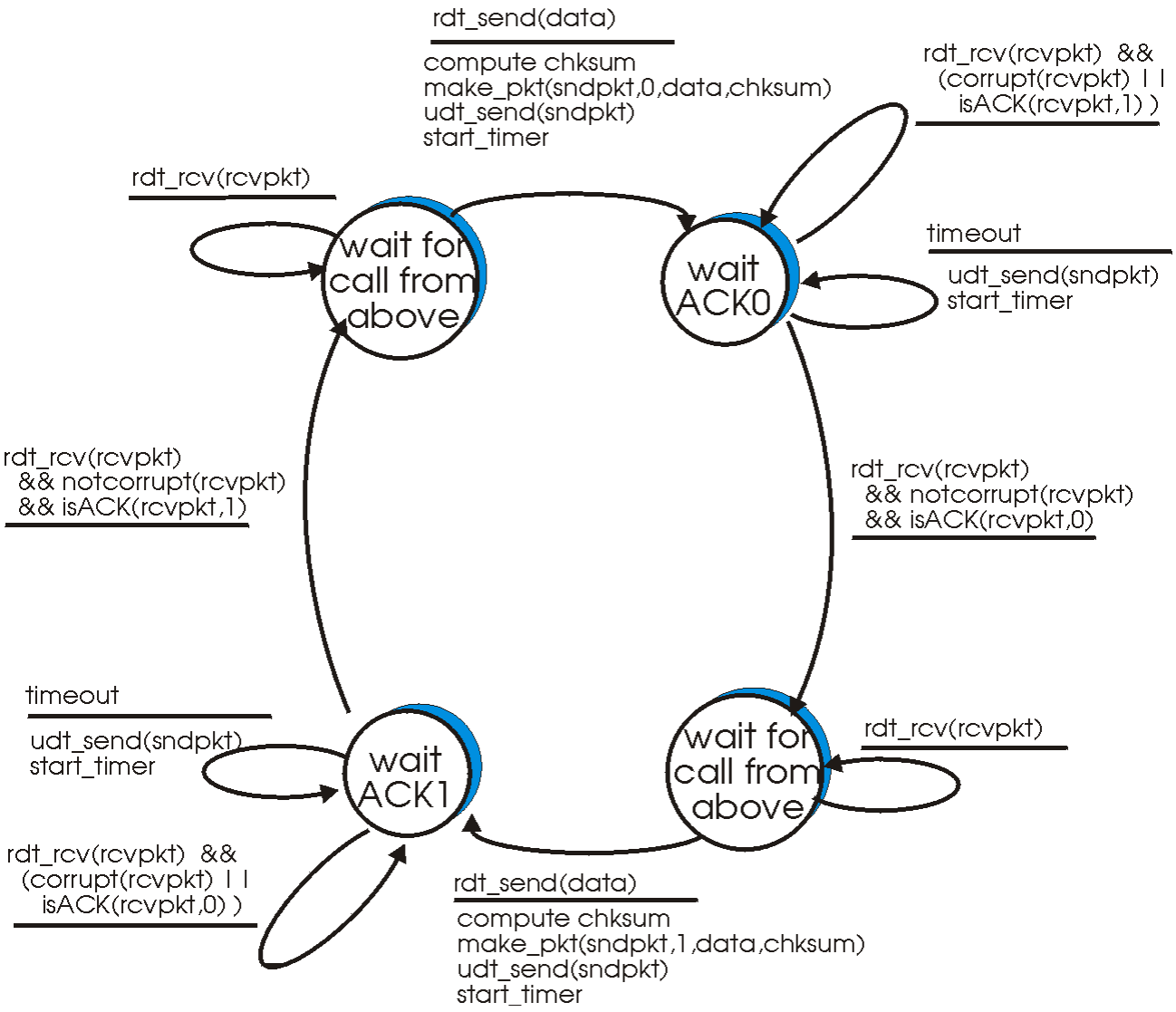
P: como lidar com perdas?

- ✓ remetente espera até ter certeza que se perdeu pacote ou ACK, e então retransmite
- ✓ eca!: desvantagens?

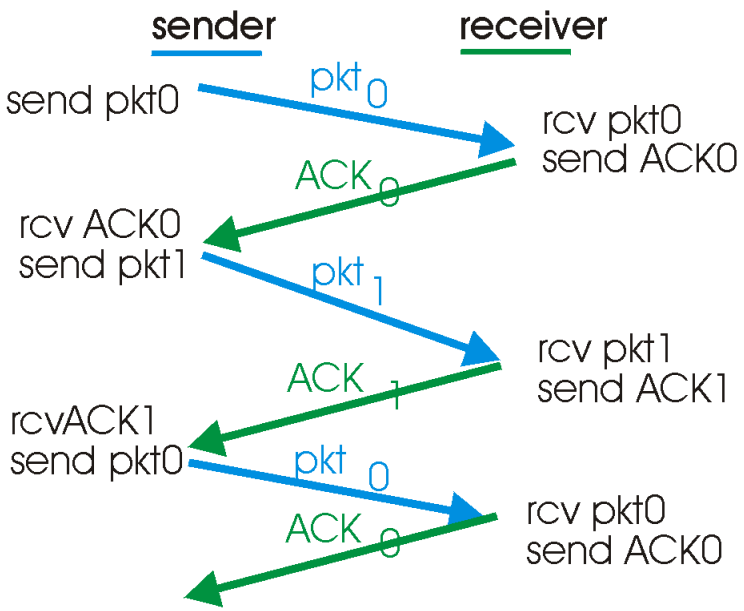
Abordagem: remetente aguarda um tempo "razoável" pelo ACK

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

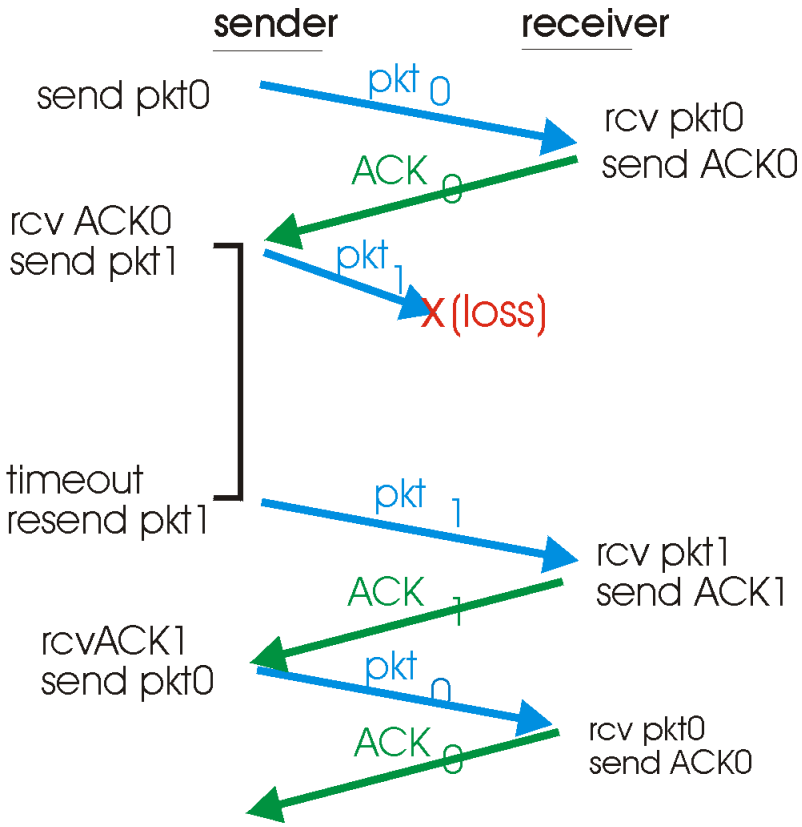
# rdt3.0: remetente



# rdt3.0 em ação

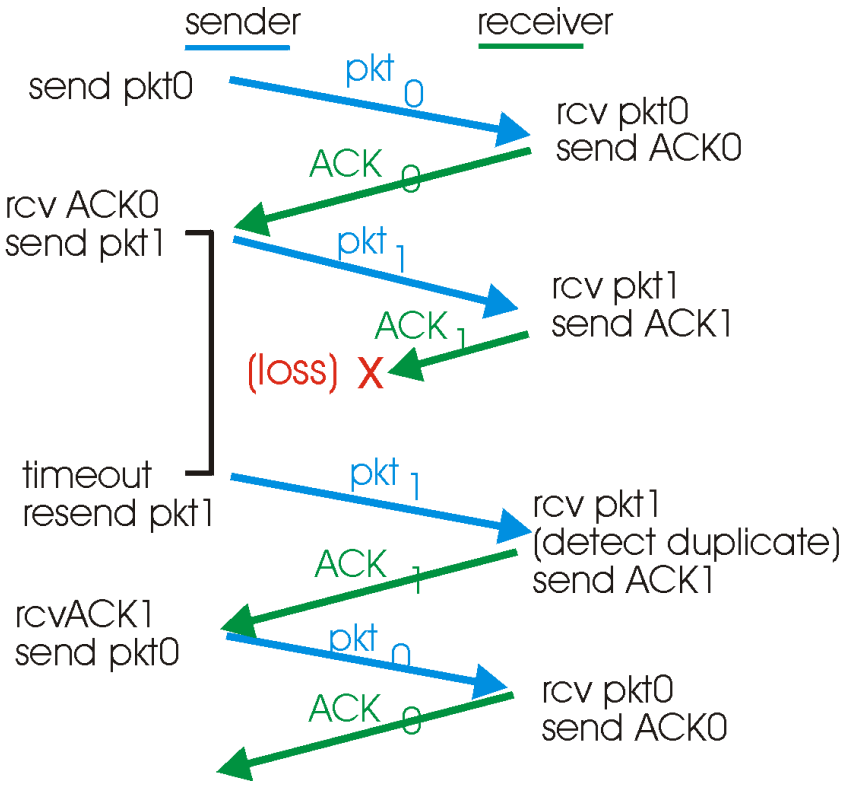



(a) operation with no loss

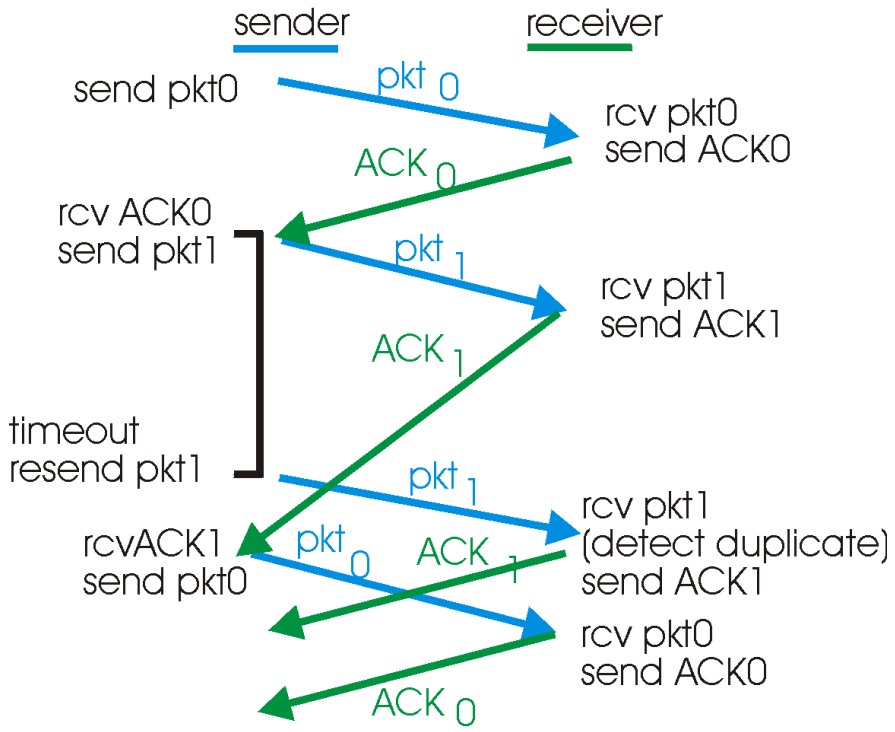



(b) lost packet

# rdt3.0 em ação




(c) lost ACK




(d) premature timeout

# Desempenho de rdt3.0

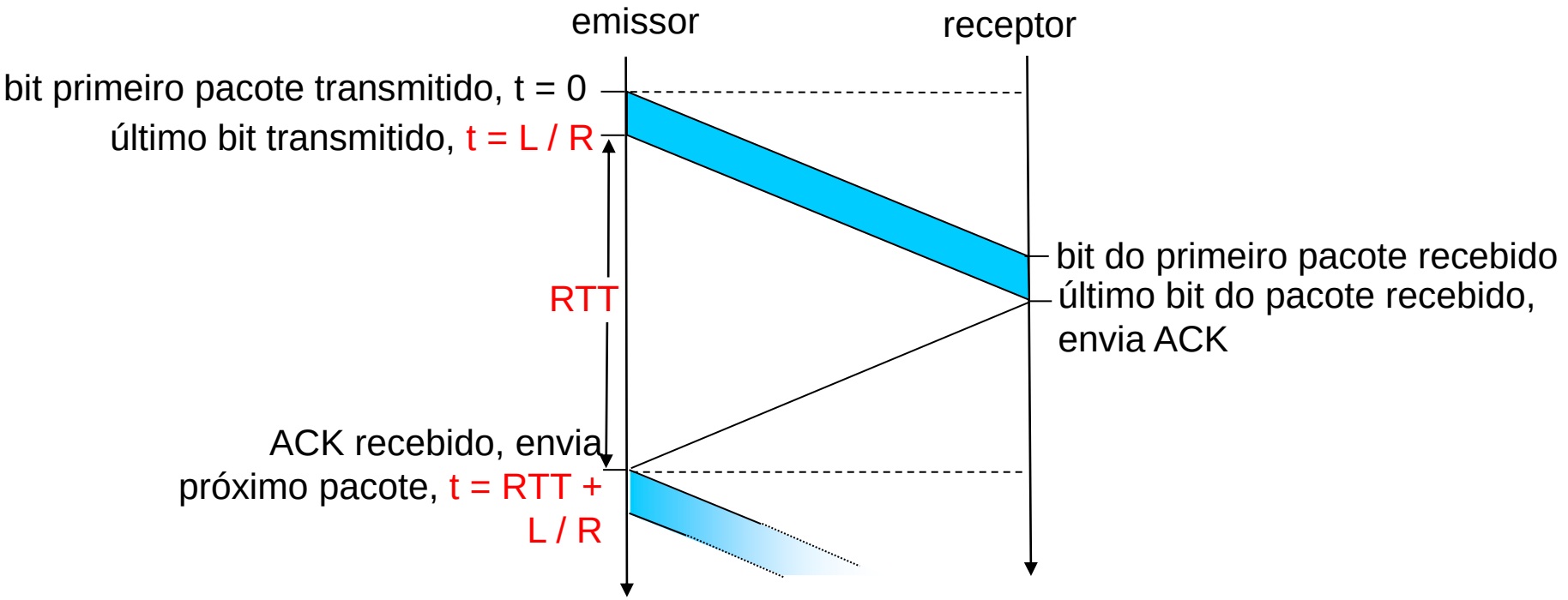
- rdt3.0 funciona, porém seu desempenho é muito ruim
- exemplo: enlace de 1 Gbps, retardo fim a fim de 15 ms, pacote de 1KB:

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

$$U_{emissor} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027 \text{ microsegundos}$$

- ✓ Utilização: fração do temporemetente ocupado
- ✓ pac. de 1KB a cada 30 mseg -> vazão de 33kB/seg num enlace de 1 Gbps
- ✓ protocolo limita uso dos recursos físicos!

# rdt3.0: operação para e espera

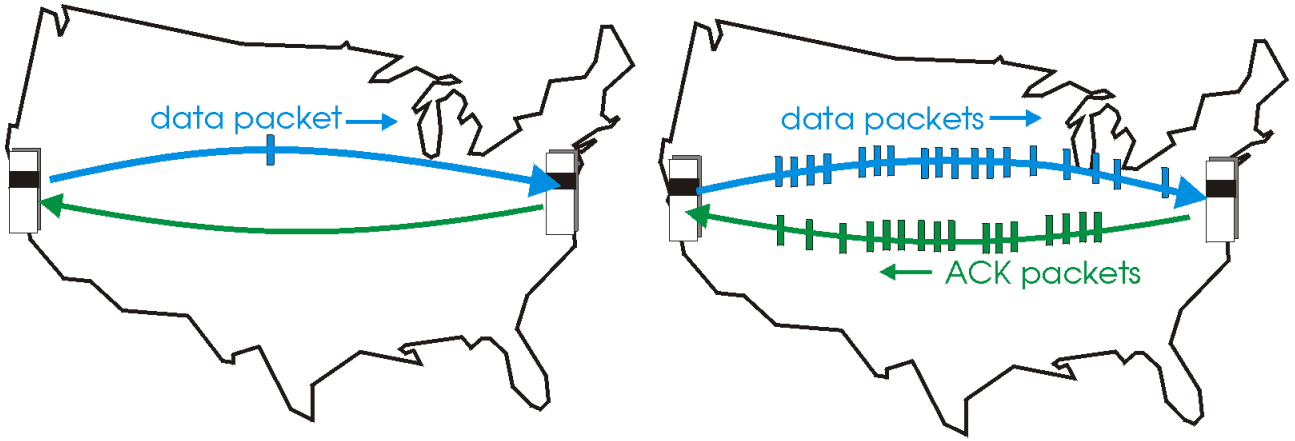



$$U_{emissor} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027 \text{ microsegundos}$$

# Protocolos "dutados" (*pipelined*)

Dutagem (pipelining): remetente admite múltiplos pacotes "em trânsito", ainda não reconhecidos

- ✓ faixa de números de seqüência deve ser aumentada
- ✓ buffers no remetente e/ou no receptor

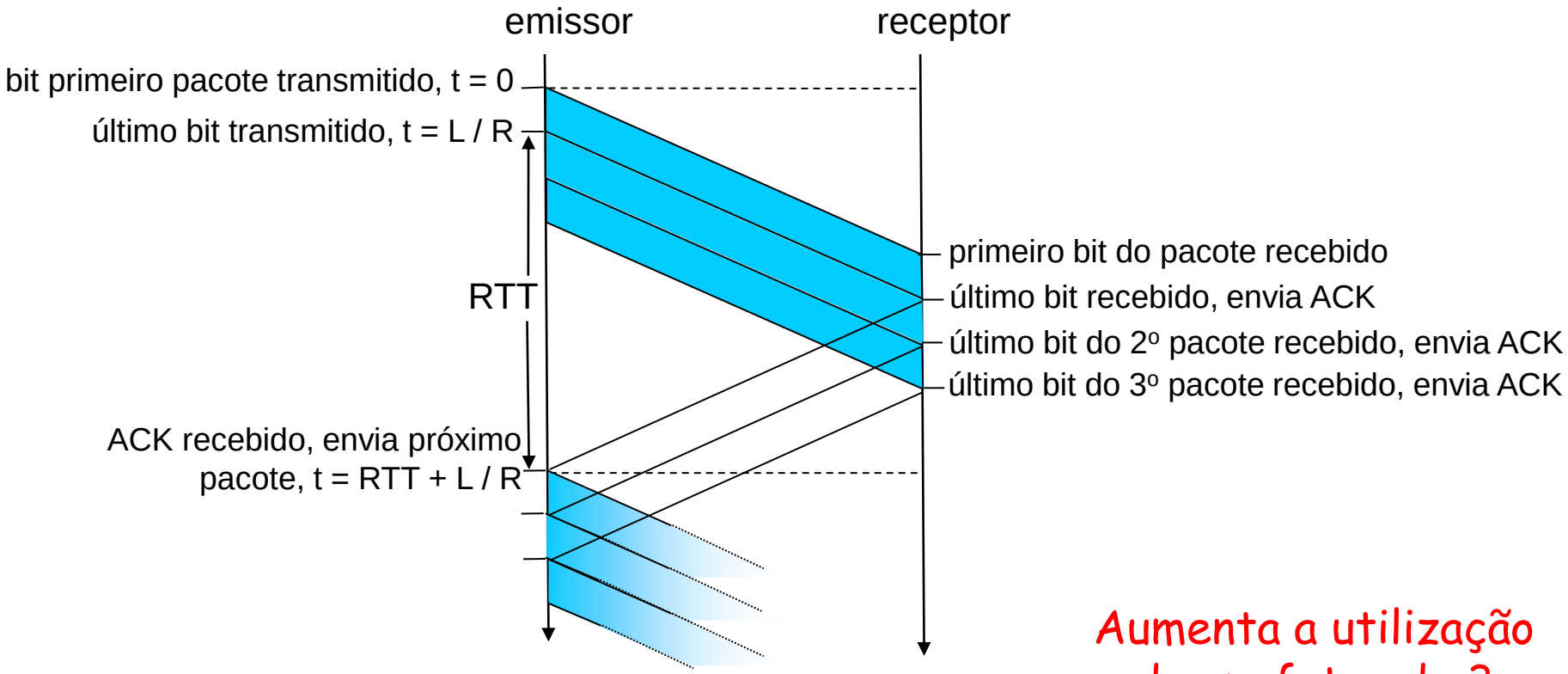



(a) a stop-and-wait protocol in operation

(b) a pipelined protocol in operation

- ➢ Duas formas genéricas de protocolos dutados: *volta-N, retransmissão seletiva*

# Pipelining: aumenta utilização



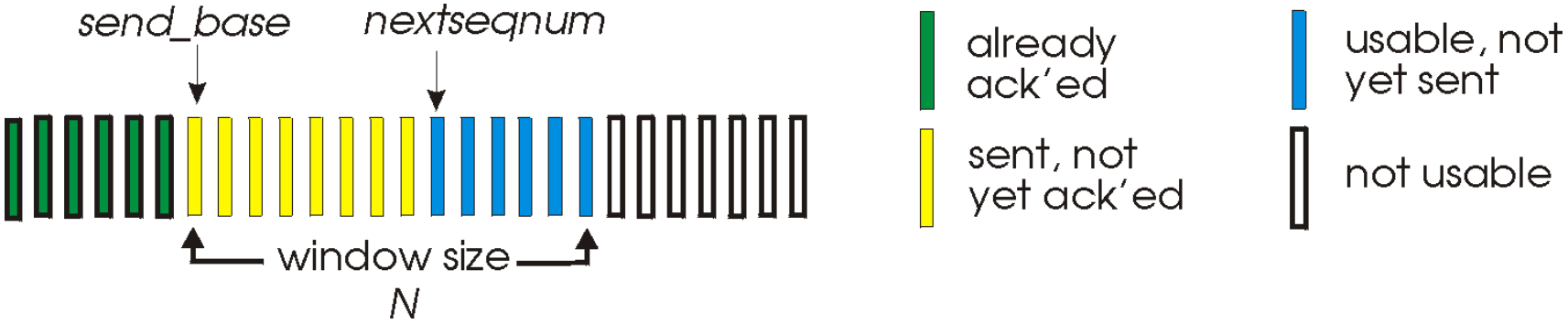

Aumenta a utilização de um fator de 3

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

# Volta-N

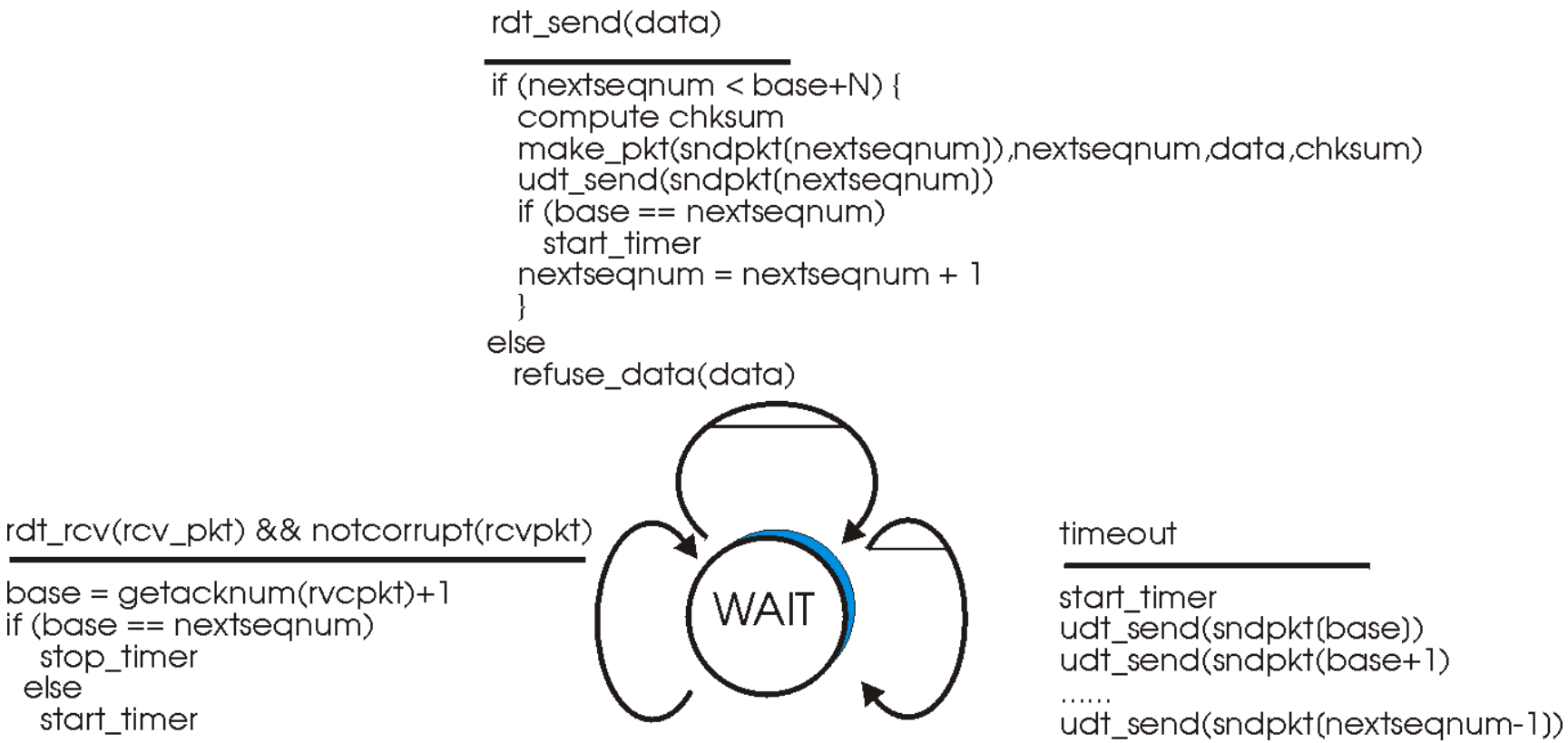
Remetente:

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

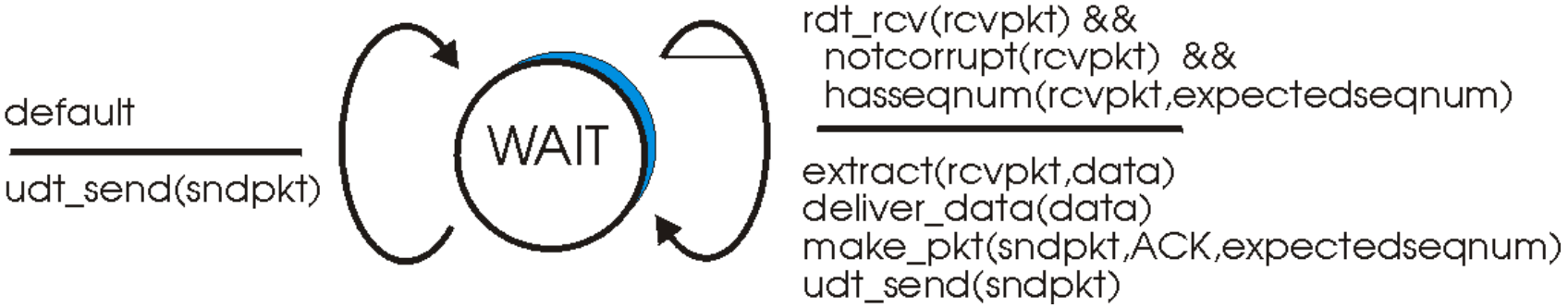



- ACK(n): reconhece todos pacotes, até e inclusive no. de seq n - "ACK cumulativo"
  - ✓ pode receber ACKs duplicados (veja receptor)
- temporizador para cada pacote em trânsito
- *timeout(n)*: retransmite pacote n e todos os pacotes com no. de seq maiores na janela

# Volta-N: FSM estendida do remetente



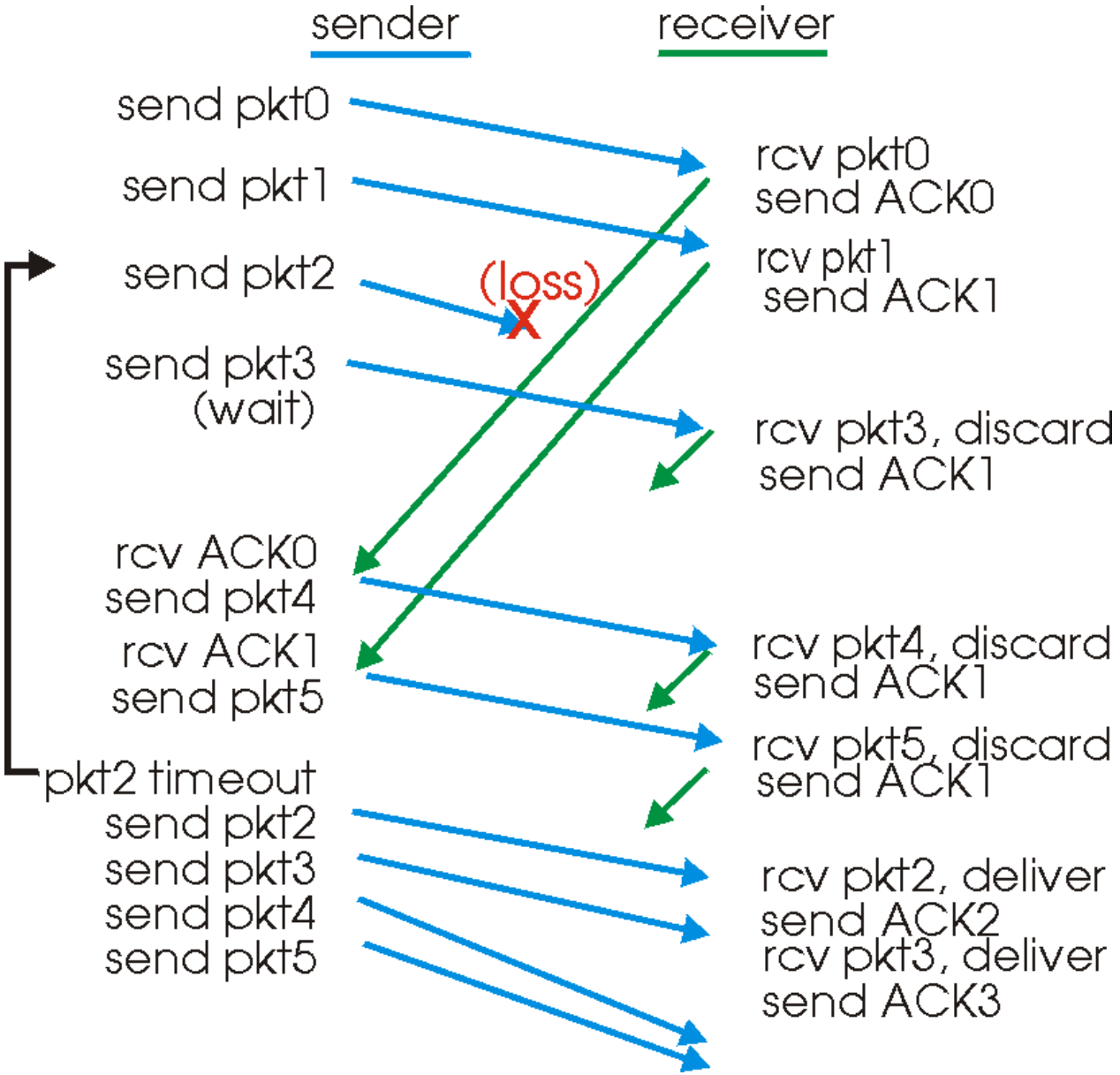
# Volta-N: FSM estendida do receptor




## receptor simples:

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

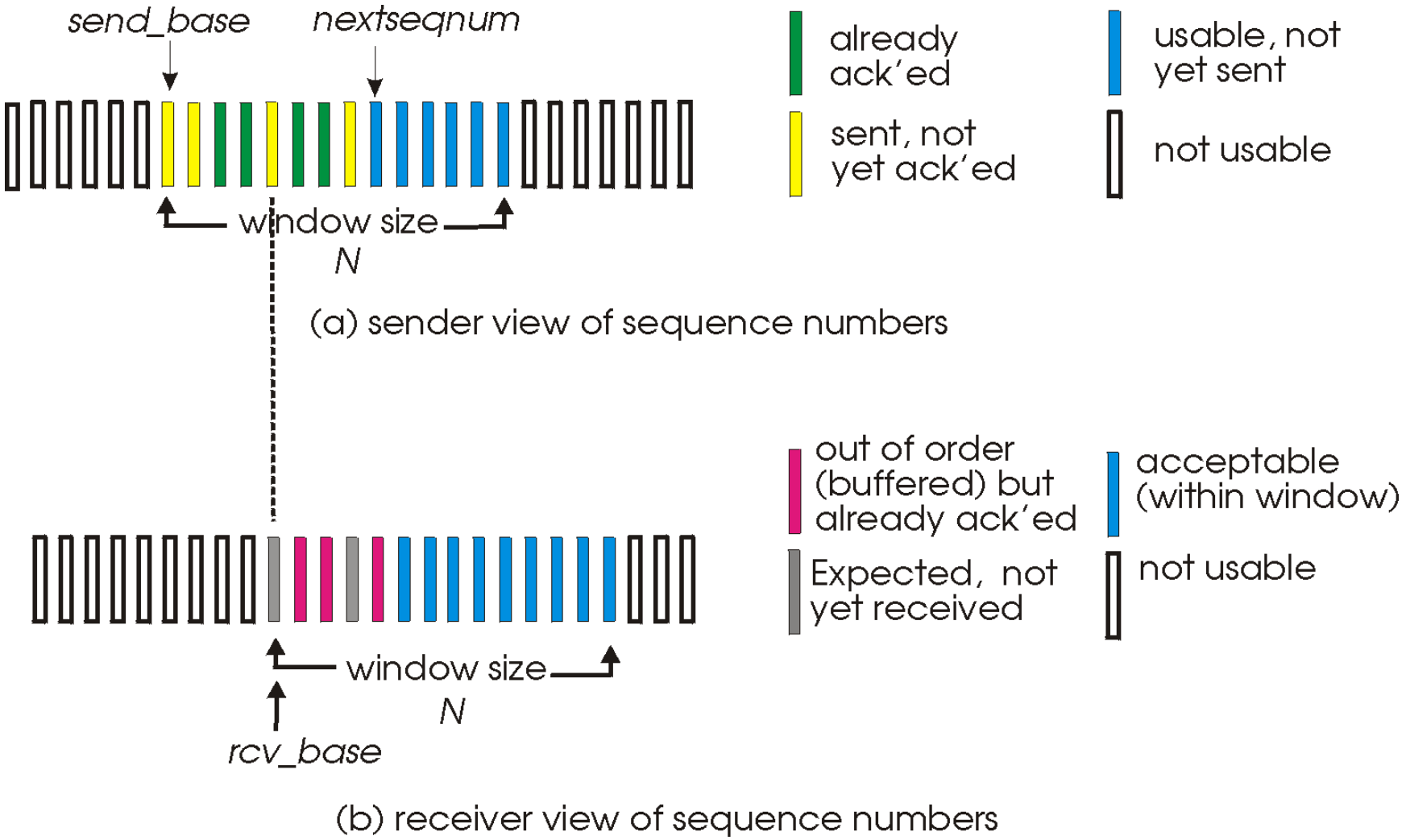
# Volta-N em ação



# Retransmissão seletiva

- receptor reconhece *individualmente* todos os pacotes recebidos corretamente
  - armazena pacotes no buffer, conforme precisa, para posterior entrega em-ordem à camada superior
- remetente apenas re-envia pacotes para os quais ACK não recebido
  - temporizador de remetente para cada pacote sem ACK
- janela do remetente
  - N nos. de seq consecutivos
  - outra vez limita nos. de seq de pacotes enviados, mas ainda não reconhecidos

# Retransmissão seletiva: janelas de remetente, receptor



# Retransmissão seletiva

**remetente**

dados de cima:

- se próx. no. de seq na janela, envia pacote

timeout(n):

- reenvia pacote n, reiniciar temporizador

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

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

**receptor**

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

- envia ACK(n)
- fora de ordem: buffer
- em ordem: entrega (tb. entrega pacotes em ordem no buffer), avança janela p/ próxima pacote ainda não recebido

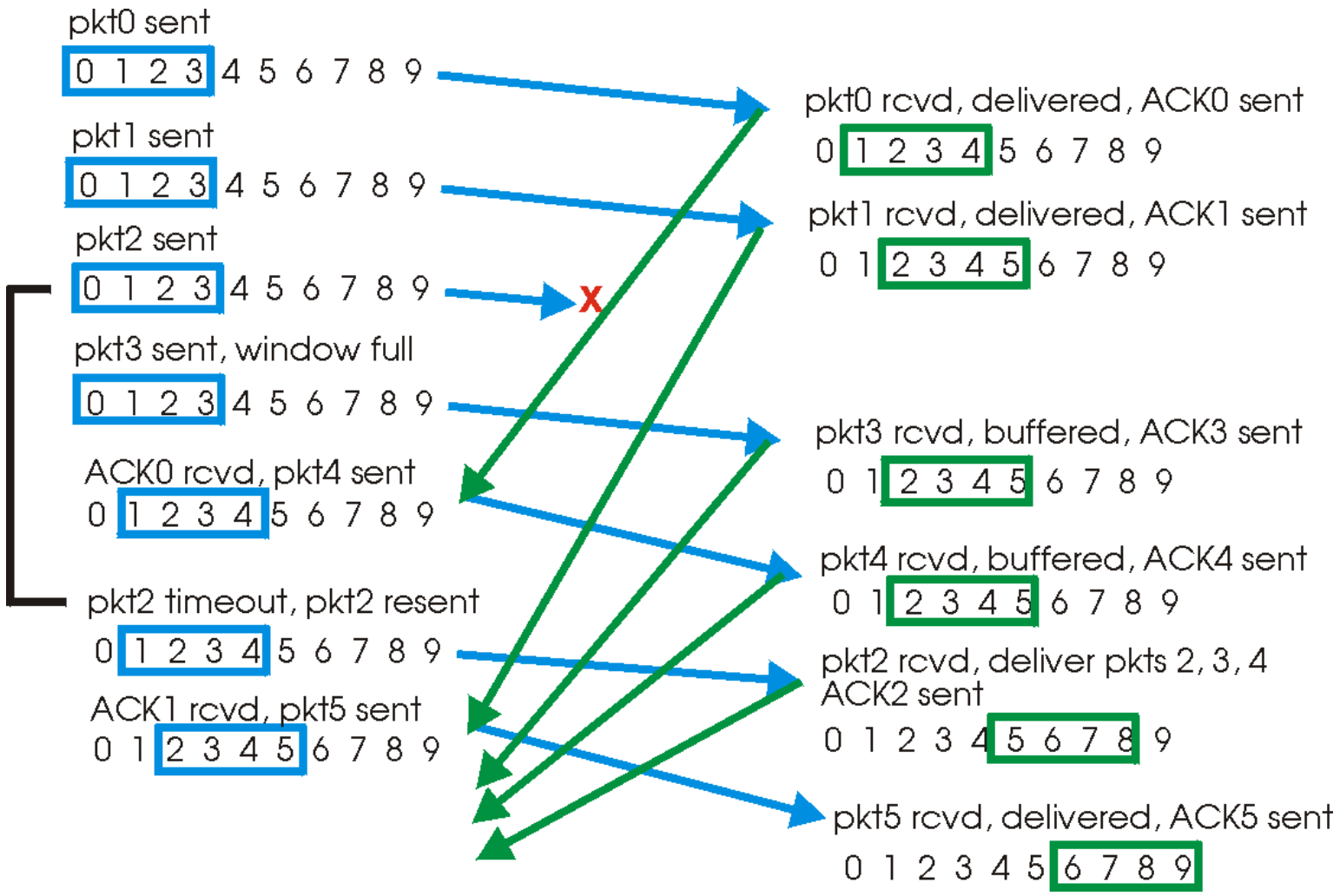
pacote n em [rcvbase-N,rcvbase-1]

- ACK(n)

senão:

- ignora

# Retransmissão seletiva em ação



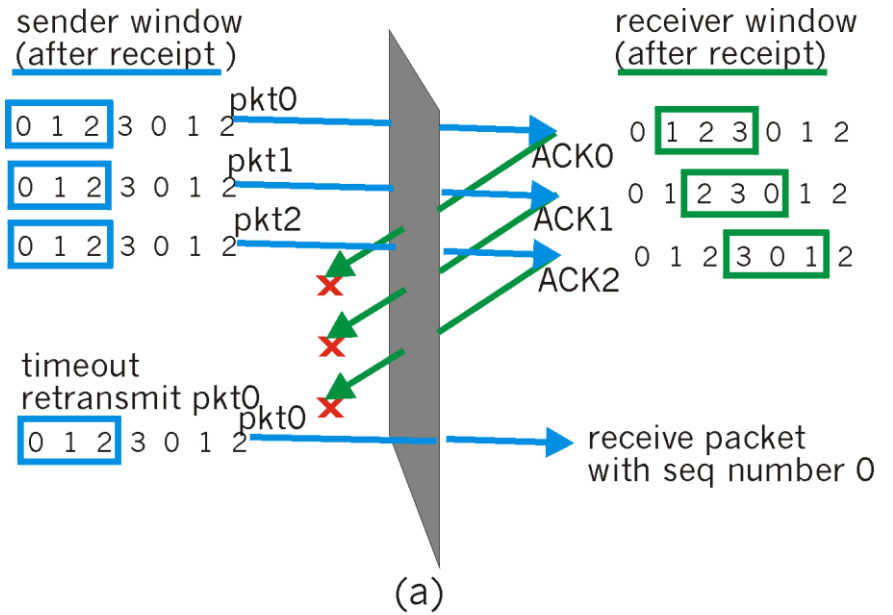
# Retransmissão seletiva: dilema

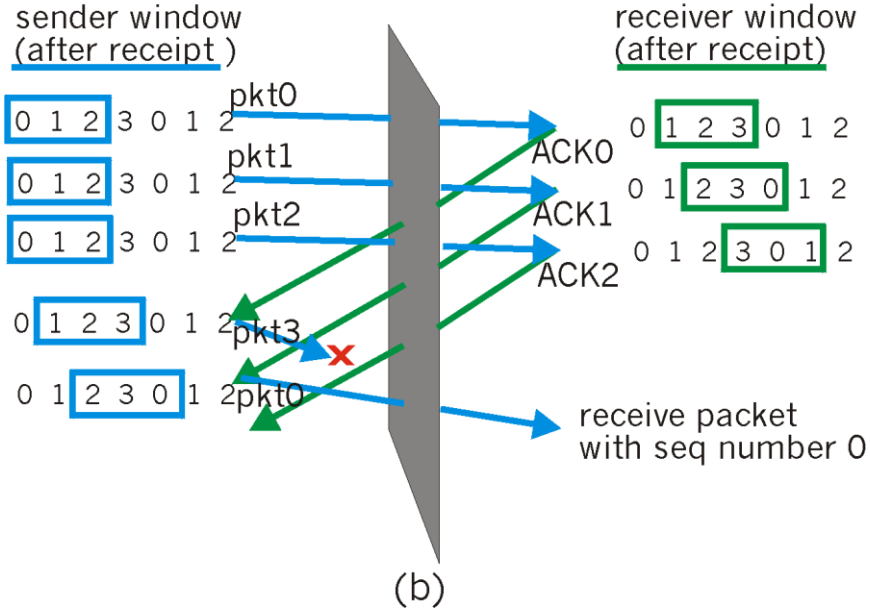
Exemplo:

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

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

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



# Transmission Control Protocol

**CASE HISTORY**

**VINTON CERF, ROBERT KAHN, AND TCP/IP**

In the early 1970s, packet-switched networks began to proliferate, with the ARPAnet—the precursor of the Internet—being just one of many networks. Each of these networks had its own protocol. Two researchers, Vinton Cerf and Robert Kahn, recognized the importance of interconnecting these networks and invented a cross-network protocol called TCP/IP, which stands for Transmission Control Protocol/Internet Protocol. Although Cerf and Kahn began by seeing the protocol as a single entity, it was later split into its two parts, TCP and IP, which operated separately. Cerf and Kahn published a paper on TCP/IP in May 1974 in *IEEE Transactions on Communications Technology* [Cerf 1974].

The TCP/IP protocol, which is the bread and butter of today's Internet, was devised before PCs, workstations, smartphones, and tablets, before the proliferation of Ethernet, cable, and DSL, WiFi, and other access network technologies, and before the Web, social media, and streaming video. Cerf and Kahn saw the need for a networking protocol that, on the one hand, provides broad support for yet-to-be-defined applications and, on the other hand, allows arbitrary hosts and link-layer protocols to interoperate.

In 2004, Cerf and Kahn received the ACM's Turing Award, considered the "Nobel Prize of Computing" for "pioneering work on internetworking, including the design and implementation of the Internet's basic communications protocols, TCP/IP, and for inspired leadership in networking."

# A Protocol for Packet Network Intercommunication

VINTON G. CERF AND ROBERT E. KAHN,
MEMBER, IEEE

*Abstract* — A protocol that supports the sharing of resources that exist in different packet switching networks is presented. The protocol provides for variation in individual network packet sizes, transmission failures, sequencing, flow control, end-to-end error checking, and the creation and destruction of logical process-to-process connections. Some implementation issues are considered, and problems such as internetwork routing, accounting, and timeouts are exposed.

## INTRODUCTION

IN THE LAST few years considerable effort has been expended on the design and implementation of packet switching networks [1]-[7],[14],[17]. A principle reason for developing such networks has been to facilitate the sharing of computer resources. A packet communication network includes a transportation mechanism for delivering data between computers or between computers and terminals. To make the data meaningful, computer and terminals share a common protocol (i.e, a set of agreed upon conventions). Several protocols have already been developed for this purpose [8]-[12],[16]. However, these protocols have addressed only the problem of communication on the same network. In this paper we present a protocol design and philosophy that supports the sharing of resources that exist in different packet switching networks.

After a brief introduction to internetwork protocol issues, we describe the function of a GATEWAY as an interface between networks and discuss its role in the protocol. We then consider the various details of the protocol, including addressing, formatting, buffering, sequencing, flow control, error control, and so forth. We close with a description of an interprocess communication mechanism and show how it can be supported by the internetwork protocol.

Even though many different and complex problems must be solved in the design of an individual packet switching network, these problems are manifestly compounded when dissimilar networks are interconnected. Issues arise which may have no direct counterpart in an individual network and which strongly influence the way in which internetwork communication can take place.

A typical packet switching network is composed of a set of computer resources called HOSTS, a set of one or more *packet switches*, and a collection of communication media that interconnect the packet switches. Within each HOST, we assume that there exist *processes* which must communicate with processes in their own or other HOSTS. Any current definition of a process will be adequate for our purposes [13]. These processes are generally the ultimate source and destination of data in the network. Typically, within an individual network, there exists a protocol for communication between any source and destination process. Only the source and destination processes require knowledge of this convention for communication to take place. Processes in two distinct networks would ordinarily use different protocols for this purpose. The ensemble of packet switches and communication media is called the *packet switching subnet*. Fig. 1 illustrates these ideas.

In a typical packet switching subnet, data of a fixed maximum size are accepted from a source HOST, together with a formatted destination address which is used to route the data in a store and forward fashion. The transmit time for this data is usually dependent upon internal network parameters such as communication media data rates, buffering and signalling strategies, routeing, propagation delays, etc. In addition, some mechanism is generally present for error handling and determination of status of the networks components.

Individual packet switching networks may differ in their implementations as follows.

1) Each network may have distinct ways of addressing the receiver, thus requiring that a uniform addressing scheme be created which can be understood by each individual network.

2) Each network may accept data of different maximum size, thus requiring networks to deal in units of the smallest maximum size (which may be impractically small) or requiring procedures which allow data crossing a network boundary to be reformatted into smaller pieces.

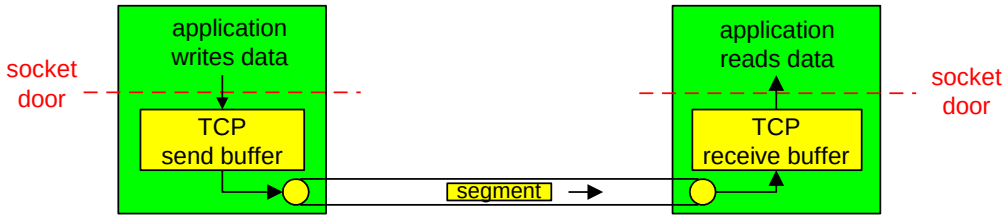
3) The success or failure of a transmission and its performance in each network is governed by different time delays in accepting, delivering, and transporting the data. This requires careful development of internetwork timing procedures to insure that data can be successfully delivered through the various networks.

4) Within each network, communication may be disrupted due to unrecoverable mutation of the data or missing data. End-to-end restoration procedures are desirable to allow complete recovery from these conditions.

Paper approved by the Associate Editor for Data Communications of the IEEE Communications Society for publications without oral presentation. Manuscript received November 5, 1973. The research reported in this paper was supported in part by the Advanced Research Projects Agency of the Department of Defense under Contract DAHC 15-73-C-0370.
V.G. Cerf is with the Department of Computer Science and Electrical Engineering, Standford University, Stanford, Calif.
R.E. Kahn is with the Information Processing Technology Office, Advanced Research Projects Agency, Department of Defense, Arlington, Va.

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

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




- transmissão full duplex:
  - fluxo de dados bi-direcional na mesma conexão
  - MSS: tamanho máximo de segmento
- orientado a conexão:
  - handshaking (troca de msgs de controle) inicia estado de remetente, receptor antes de trocar dados
- fluxo controlado:
  - receptor não será afogado

# TCP orientado a bytes

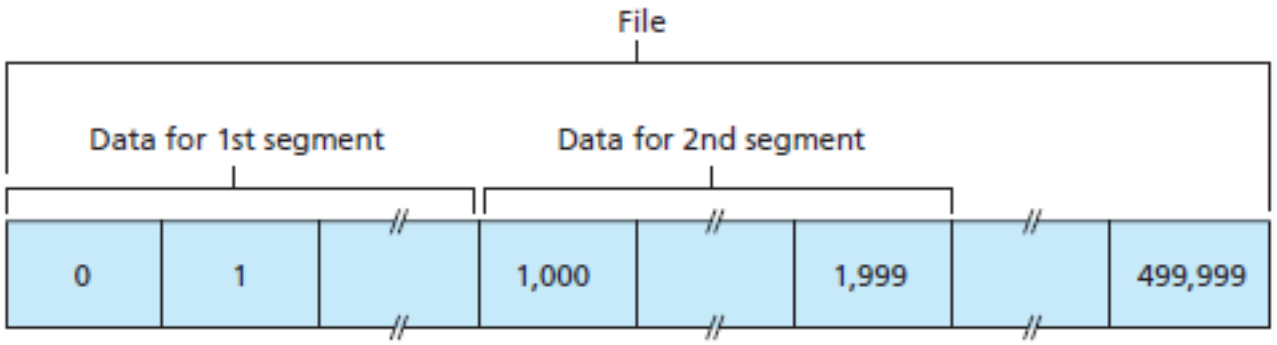
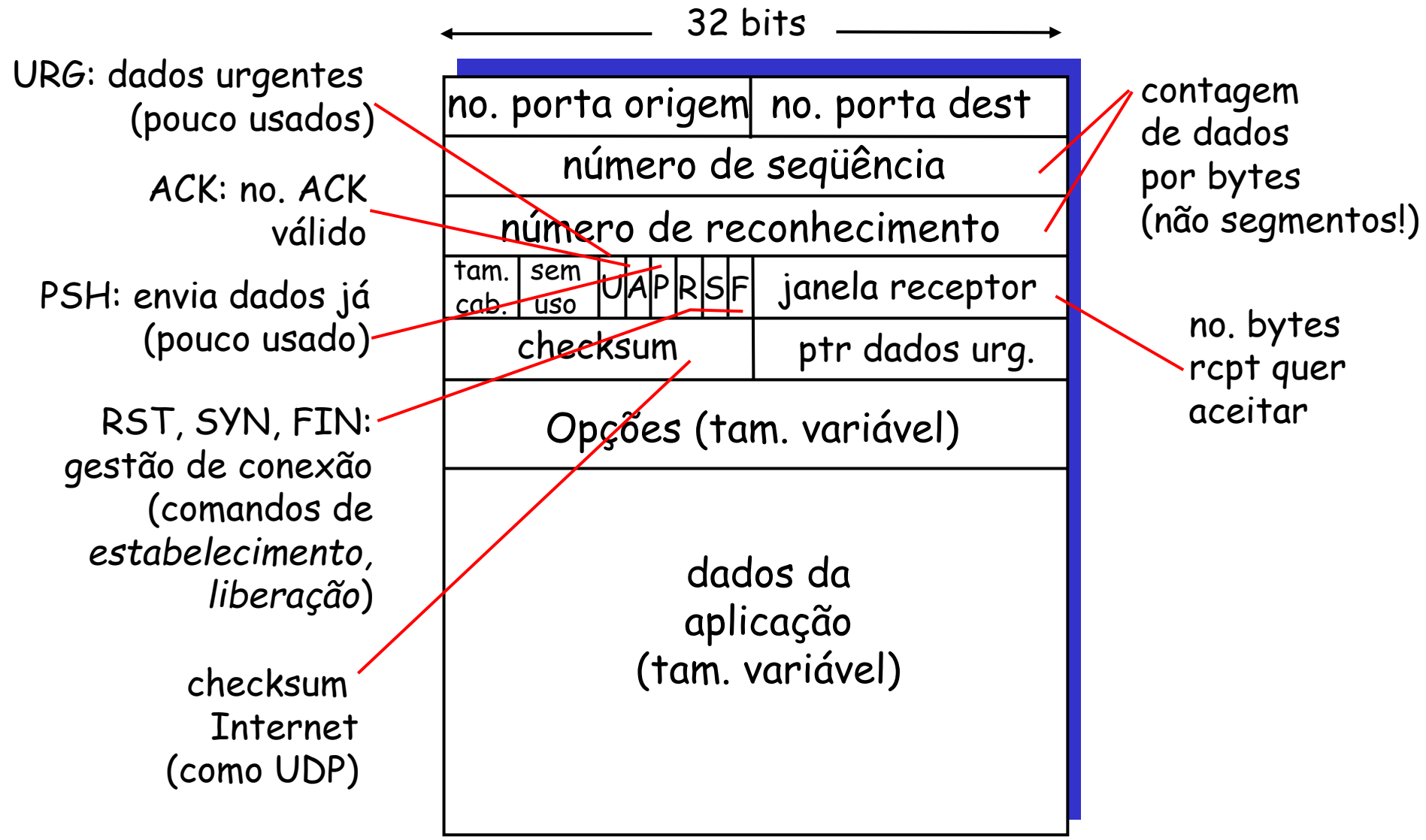



**Figure 3.30** ♦ Dividing file data into TCP segments

# TCP: estrutura do segmento



# TCP: nºs. de seq. e ACKs

Nºs. de seq.:

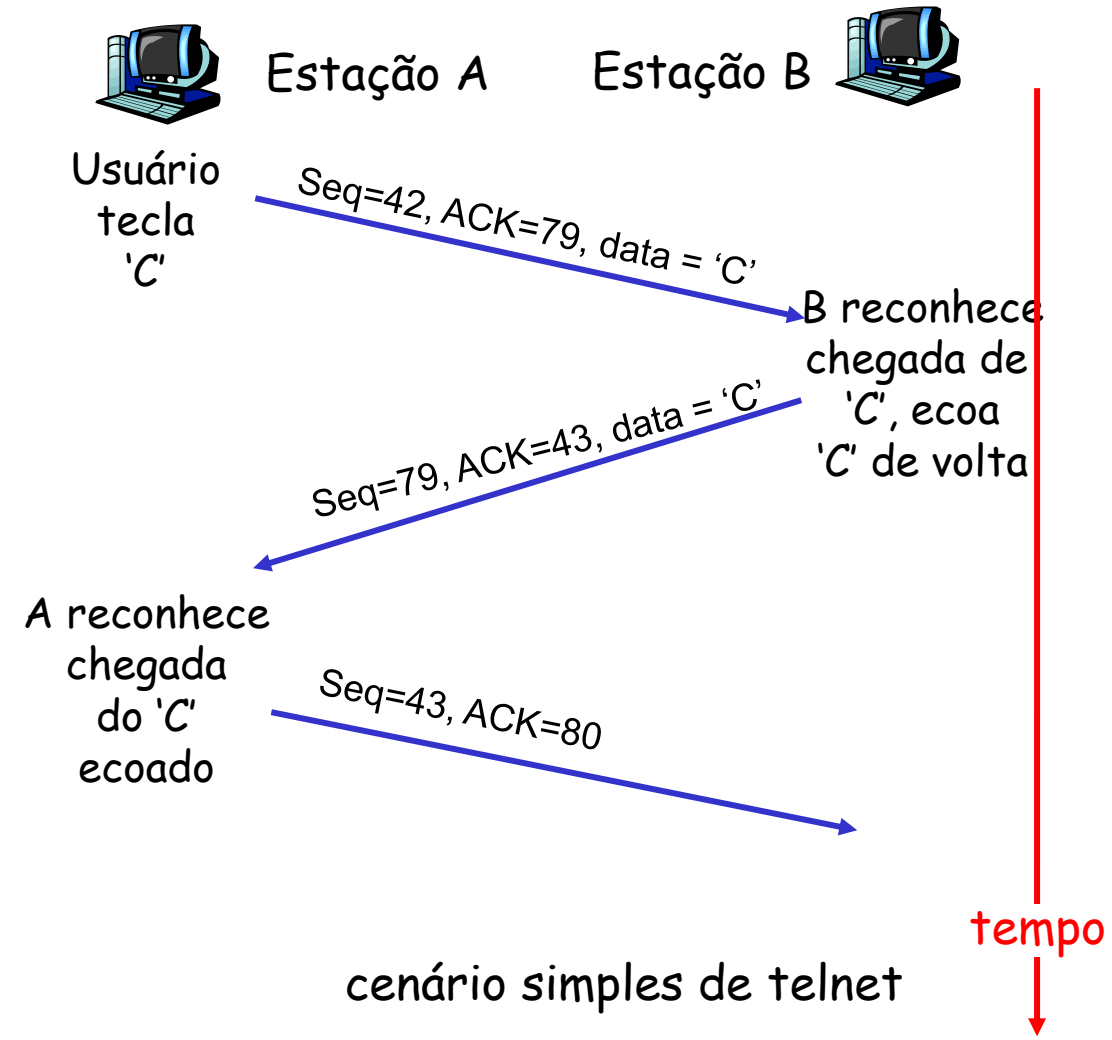
- ✓ "número"dentro do fluxo de bytes do primeiro byte de dados do segmento

ACKs:

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

P: como receptor trata segmentos fora da ordem?

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




cenário simples de telnet

# TCP: Tempo de Resposta (RTT) e Temporização

P: como escolher valor do temporizador TCP?

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

P: como estimar RTT?

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

# TCP: Tempo de Resposta (RTT) e Temporização

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

- média corrente exponencialmente ponderada
- influência de cada amostra diminui exponencialmente com o tempo
- valor típico de $\alpha$ : 0.125

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

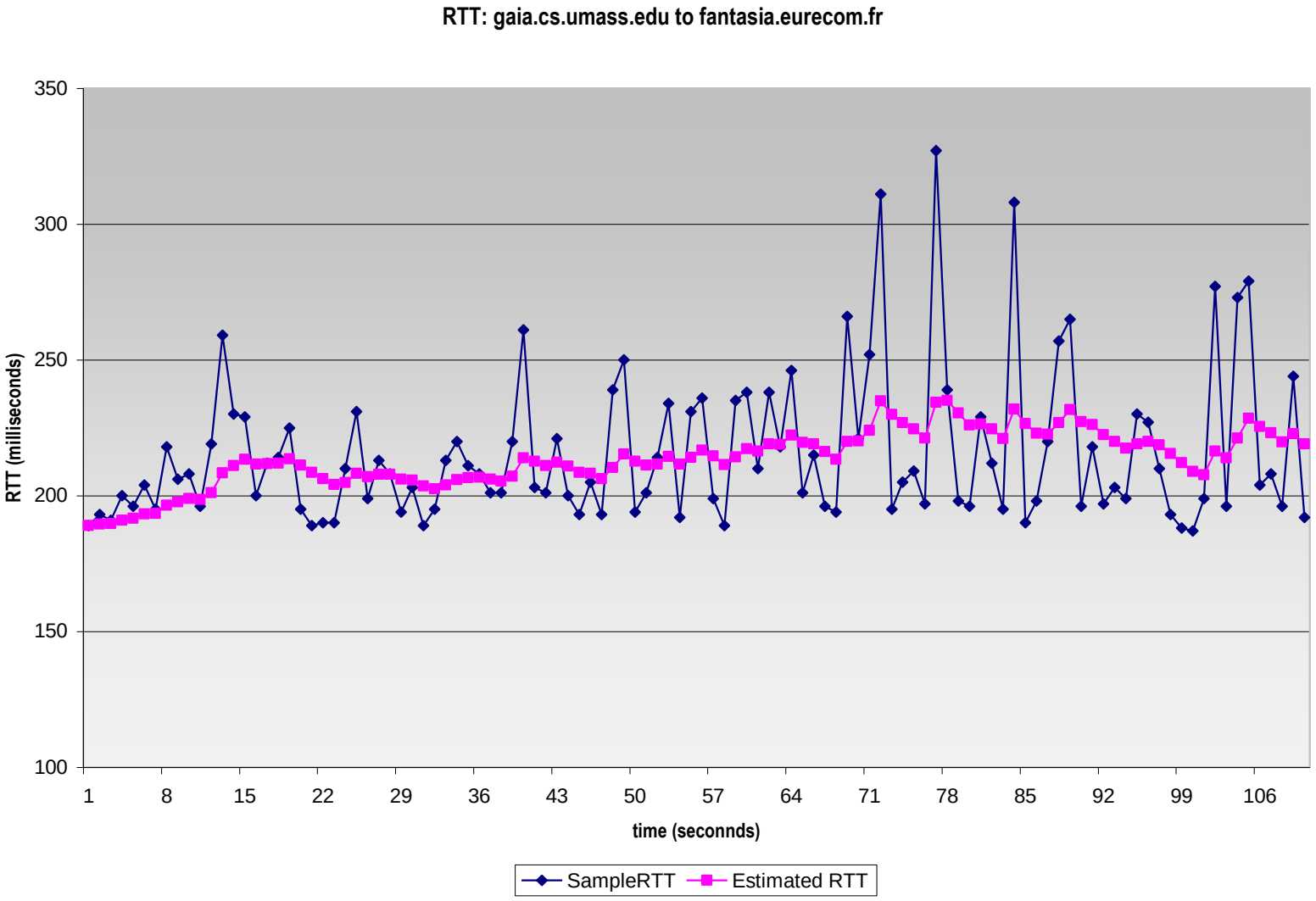
valor típico de $\beta$ : 0.25

## Escolhendo o intervalo de temporização

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

**Temporização = RTT_estimado + 4*Desvio**

# Exemplo estimativa RTT



# TCP: transferência confiável de dados

- TCP cria serviço rdt sobre o serviço não confiável IP
- Utiliza pipeline para enviar segmentos
- ACKS cumulativos
- TCP usa temporizador para retransmissão
- Retransmissões são disparadas por:
  - ✓ timeout
  - ✓ Acks duplicados
- Inicialmente considere um TCP emissor simplificado:
  - ✓ ignore acks duplicados
  - ✓ ignore controle de fluxo e controle de congestionamento;

# Eventos do TCP emissor:

## Dados recebidos da aplic:

- Cria o segmento com nº de seq X
- seq X é o número do primeiro byte do segmento
- Inicia o temporizador se ele não foi iniciado anteriormente
- Intervalo de expiração: `TimeOutInterval`

## timeout:

- Retransmite o segmento que causou o timeout
- Reinicializa o temporizador

## Ack recebido:

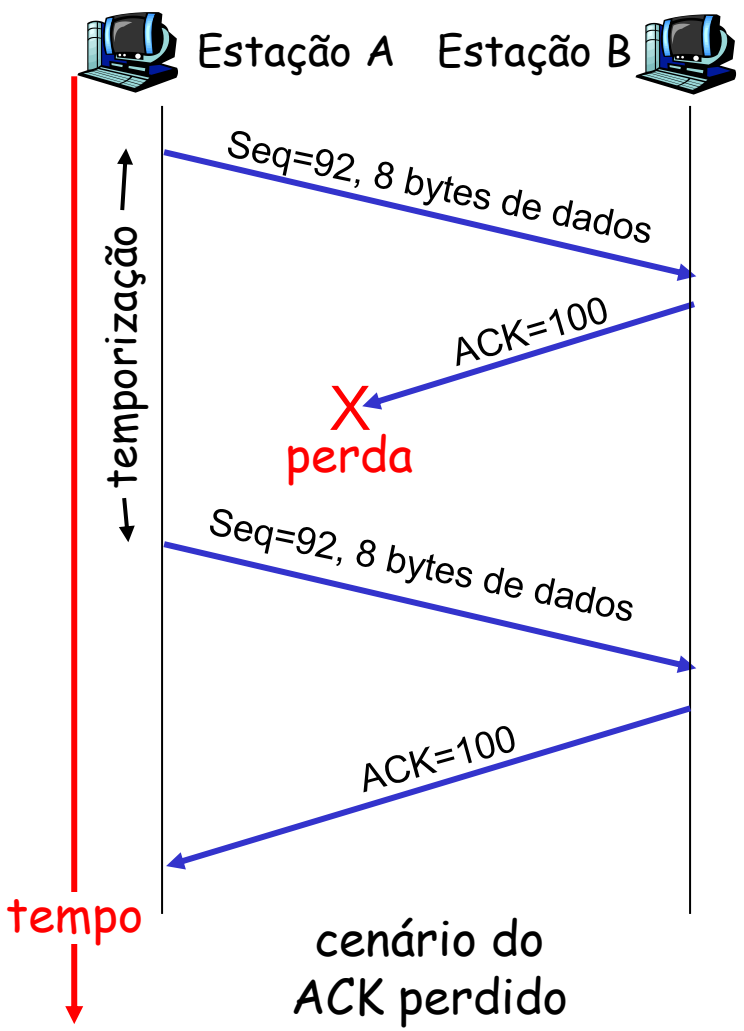
- Se reconhece segmentos não reconhecidos anteriormente
  - ✓ Atualiza a informação sobre pacotes reconhedidos
  - ✓ Inicia o temporizador se existem ainda segmentos não reconhecidos

# TCP: transferência confiável de dados

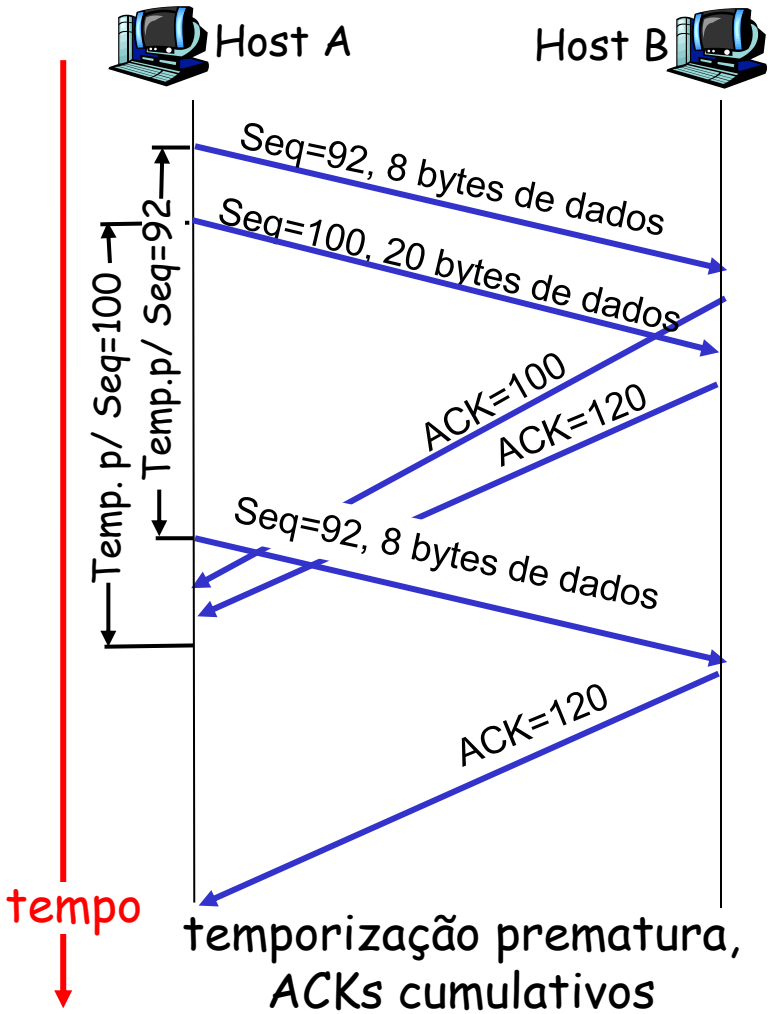
Remetente TCP simplificado

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

# TCP: cenários de retransmissão

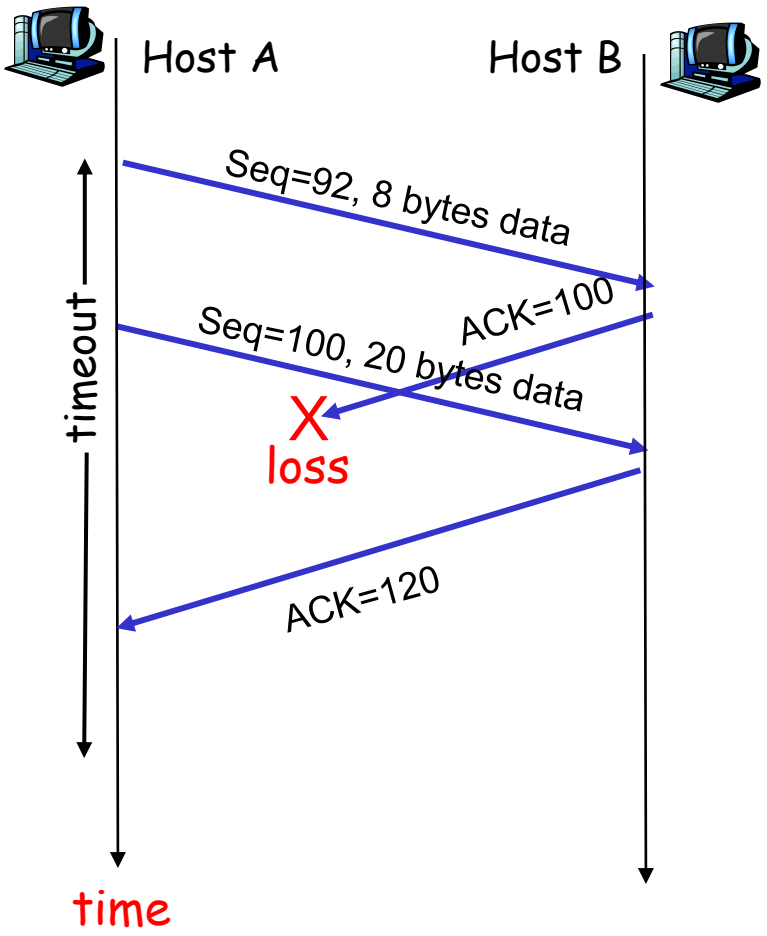



cenário do
ACK perdido




temporização prematura,
ACKs cumulativos

# TCP: cenários de retransmissão (cont)




Cenário de ACK cumulativo

# TCP geração de ACKs [RFCs 1122, 2581]

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

# Fast Retransmit

- Período de timeout é geralmente longo:
  - Longo atraso até a retransmissão do segmento perdido
- Detectar segmentos perdidos via ACKs duplicados
  - Emissores geralmente enviam vários segmentos
  - Se um segmento é perdido, provavelmente vão ser recebidos ACKs duplicados
- Se o emissor recebe 3 ACKs para o mesmo segmento, supõe que o segemtno subseqüente foi perdido:
  - fast retransmit: retransmite o segmetno antes que o temporizador expire

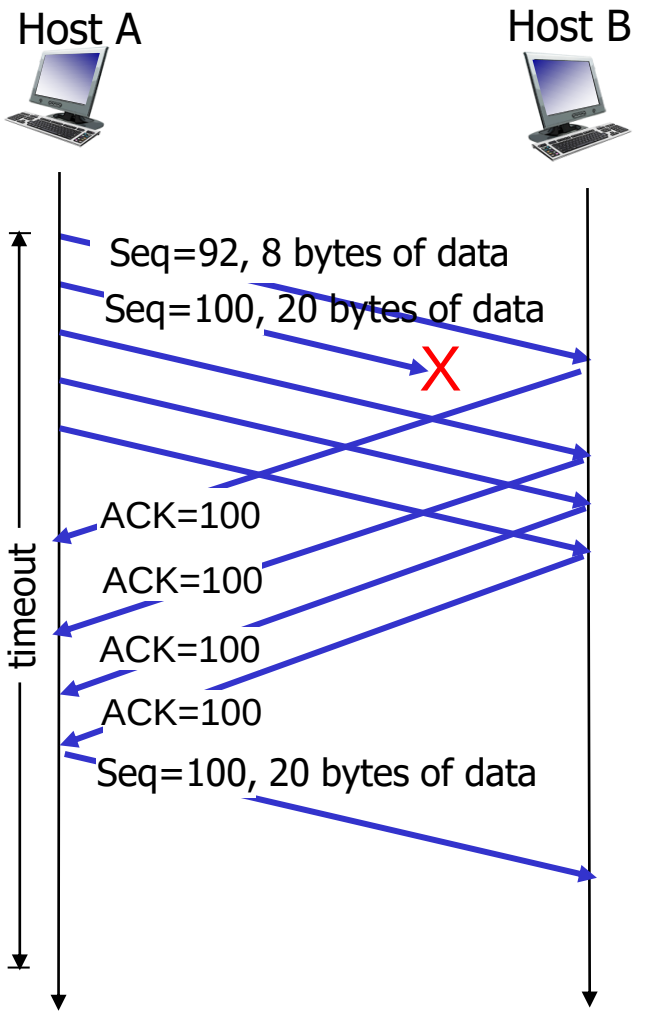
# Algoritmo Fast retransmit:

```
evento: ACK recebido, com valor de ACK igual a y
            if(y > SendBase) {
                SendBase = y
                if (existem segemtnos ainda não reconhecidos)
                    inicializa o temporizador
              }
            else {
                incrementa o contador de ACKs duplicados para y
                if (contador de  ACKs duplicados para y = 3) {
                    retransmite o segmento com nº seq y
                }
```

Um ACK duplicado para um segmento já reconhecido previamente

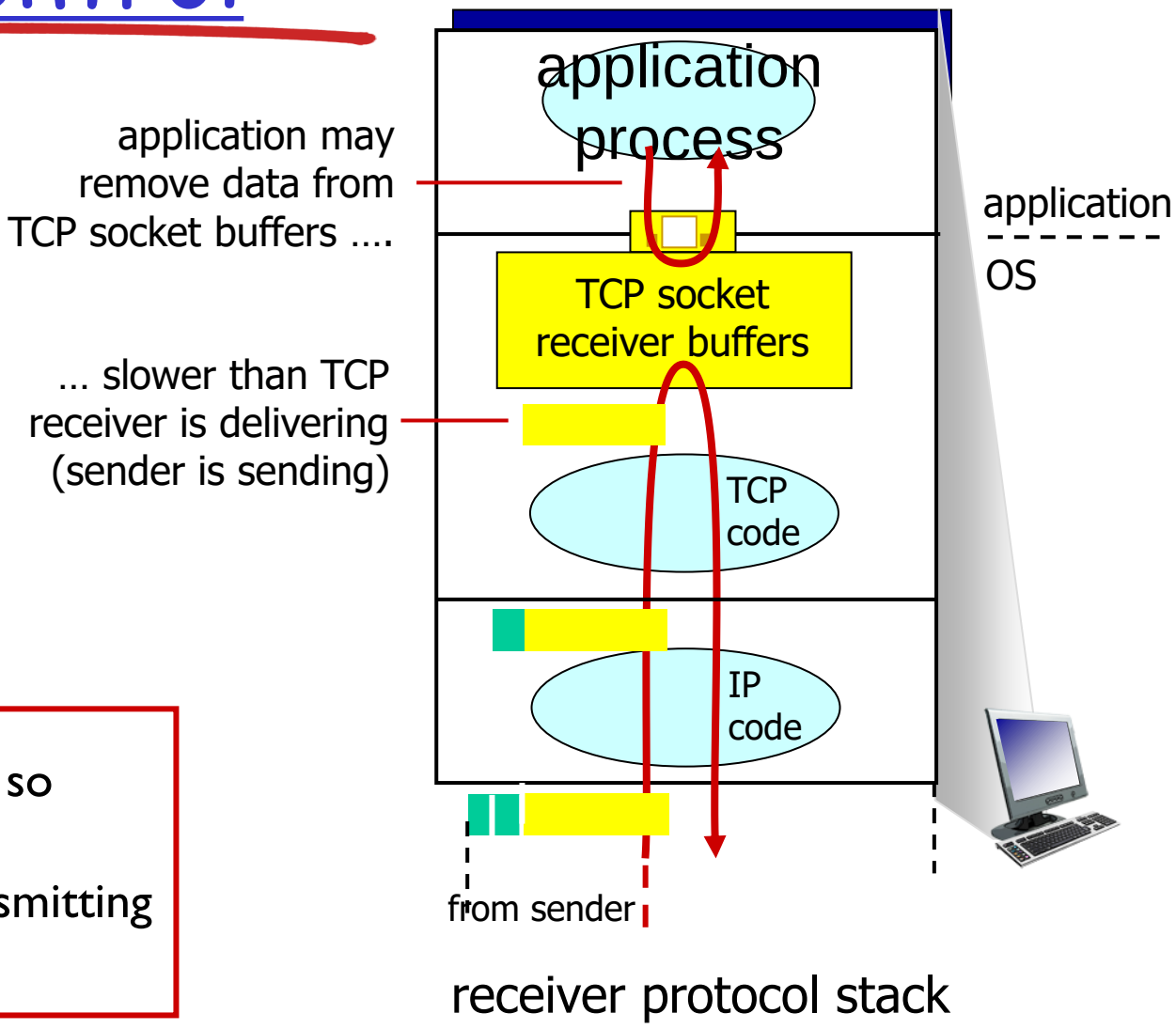
fast retransmit

# TCP fast retransmit




fast retransmit after sender receipt of triple duplicate ACK

# TCP flow control

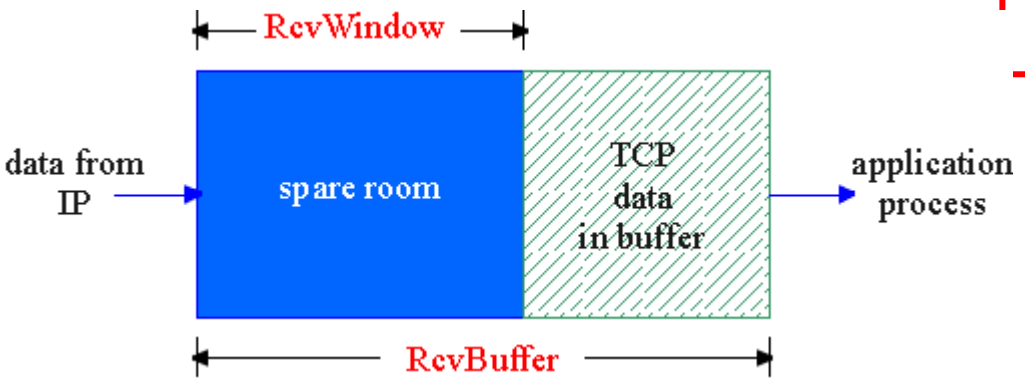



*flow control*

receiver controls sender, so sender won't overflow receiver's buffer by transmitting too much, too fast

# TCP: Controle de Fluxo

- Lado receptor de uma conexão TCP tem um buffer de recepção:



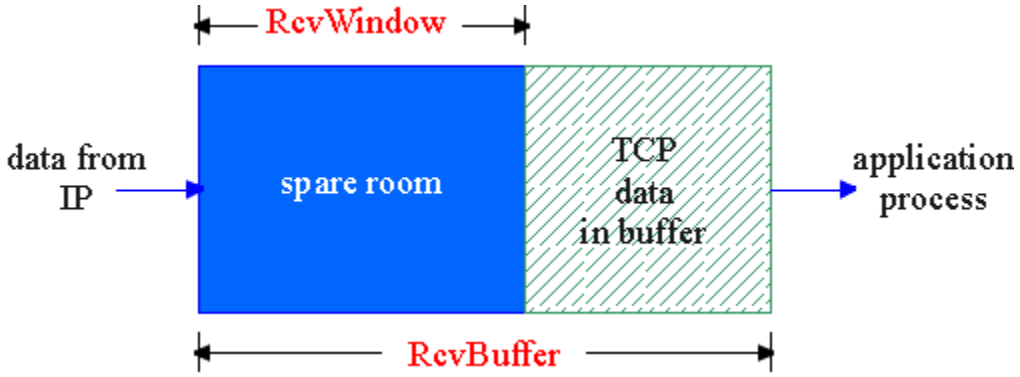

- Processo da aplicação pode ser lento para retirar os dados do buffer

**controle de fluxo**

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

- Serviço de compatibilização de velocidades: compatibilizar a taxa de envio do emissor com a taxa de recebimento dos dados da aplicação do receptor

# Controle de Fluxo TCP: como funciona ?




(Suponha que o TCP receptor discarte pacotes fora de ordem)

= **RcvWindow**

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

- receptor: explicitamente avisa o remetente da quantidade de espaço livre disponível (muda dinamicamente)
  - ✓ campo **RcvWindow no** segmento TCP
- remetente: mantém a quantidade de dados transmitidos, porém ainda não reconhecidos, menor que o valor mais recente de **RcvWindow**
  - ✓ Garante que não tenha overflow no buffer do receptor

# TCP: Gerenciamento de Conexões

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

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

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

- *servidor*: contactado por cliente

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

## Inicialização em 3 tempos:

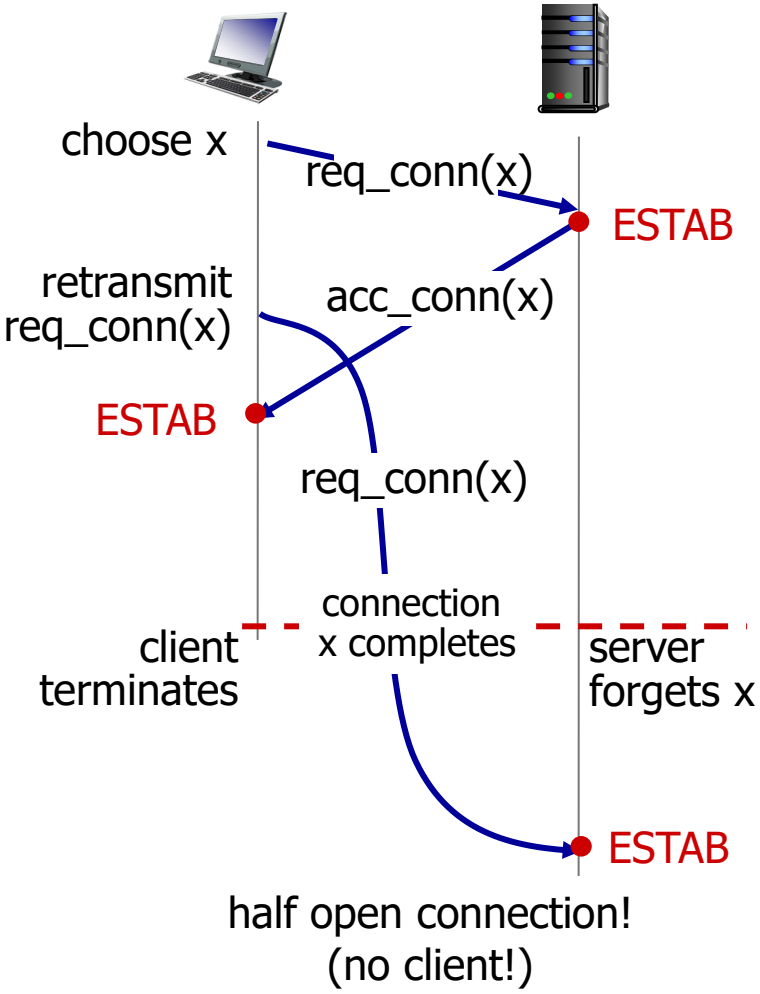
Passo 1: sistema cliente envia segmento de controle SYN do TCP ao servidor
- ✓ especifica no. inicial de seq
- ✓ sem dados

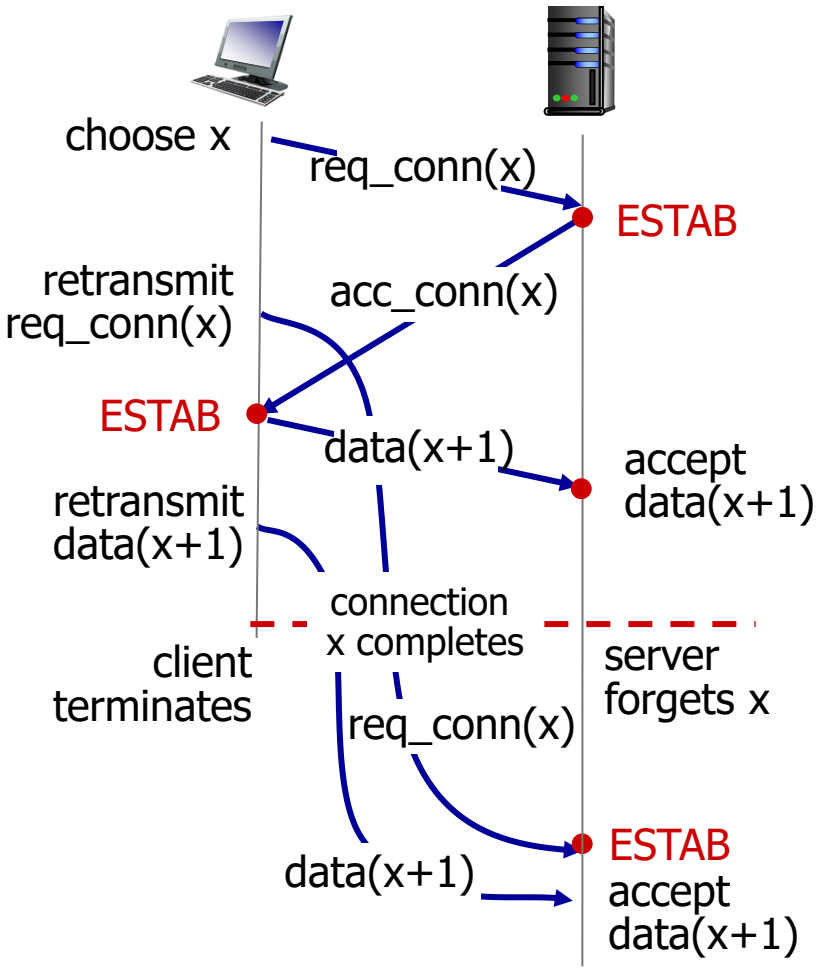
Passo 2: sistema servidor recebe SYN, responde com segmento de controle SYNACK
- ✓ aloca buffers
- ✓ especifica no. inicial de seq.

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

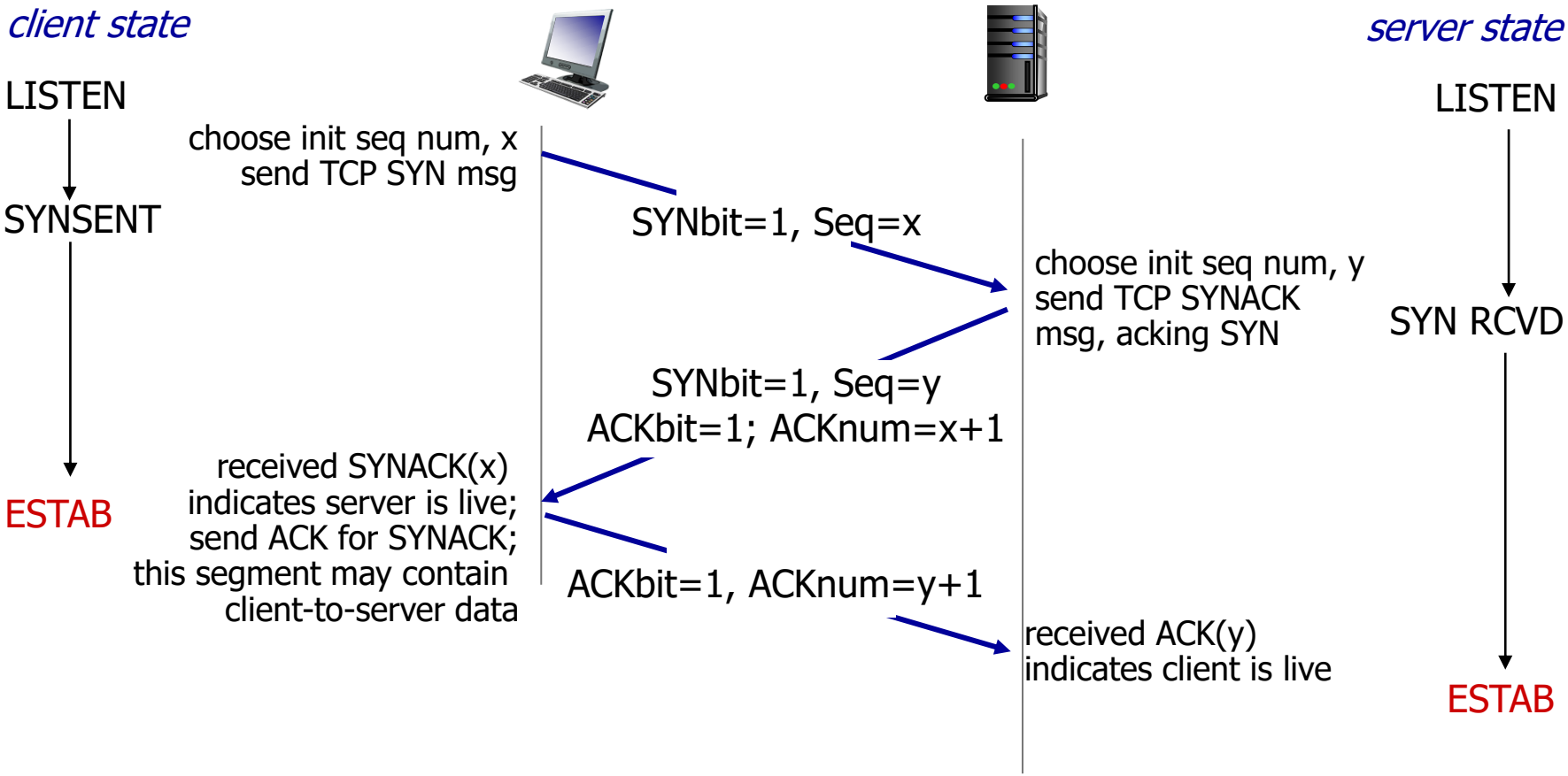
# Agreeing to establish a connection

2-way handshake failure scenarios:



# TCP 3-way handshake



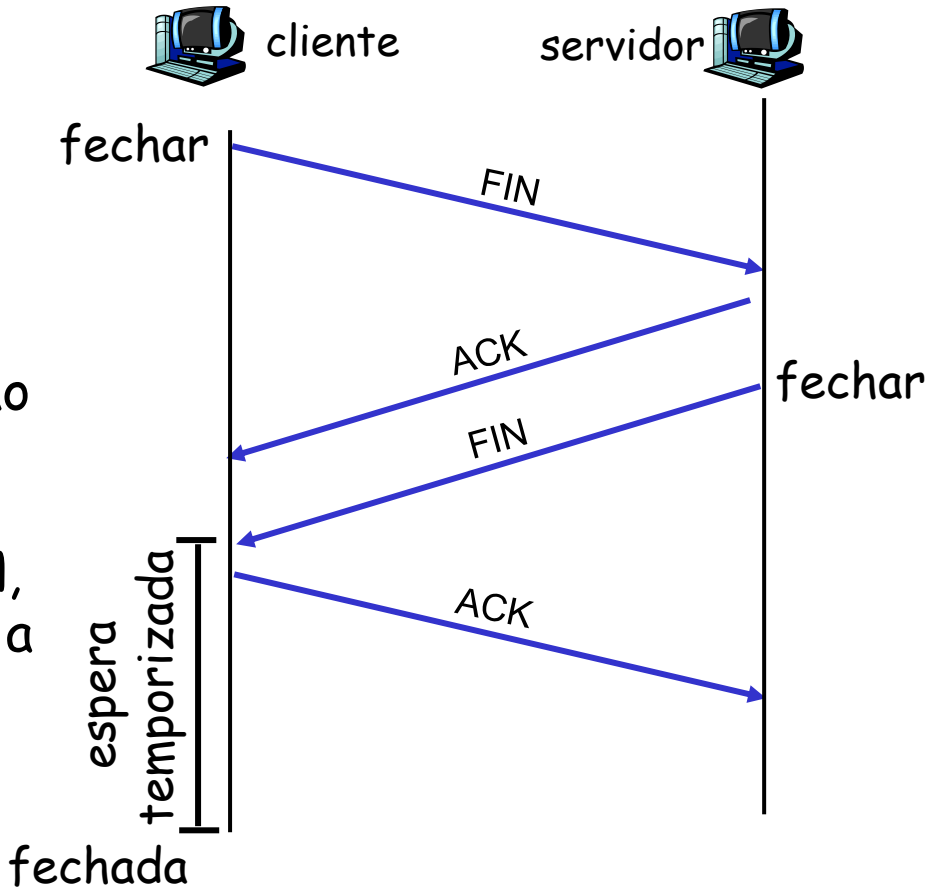
# TCP: Gerenciamento de Conexões (cont.)

## Encerrando uma conexão:

cliente fecha soquete:
`clientSocket.close();`

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

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



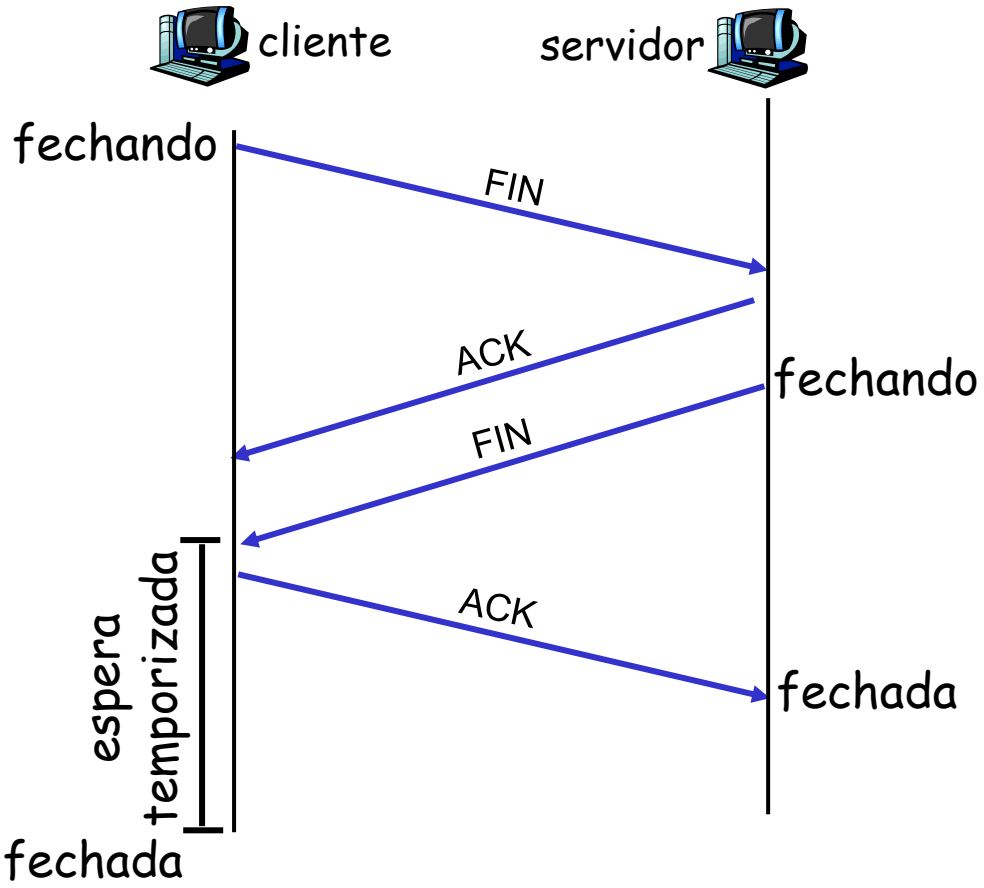
# TCP: Gerenciamento de Conexões (cont.)

**Passo 3:** cliente recebe FIN, responde com ACK.

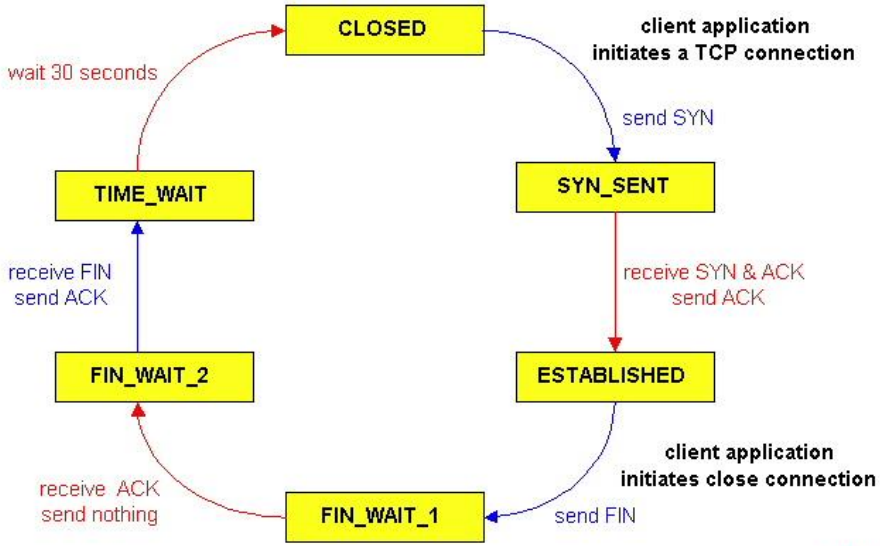
- ✓ Entre em "espera temporizada" - responderá com ACK a FINs recebidos

**Step 4:** servidor, recebe ACK. Conexão encerrada.

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

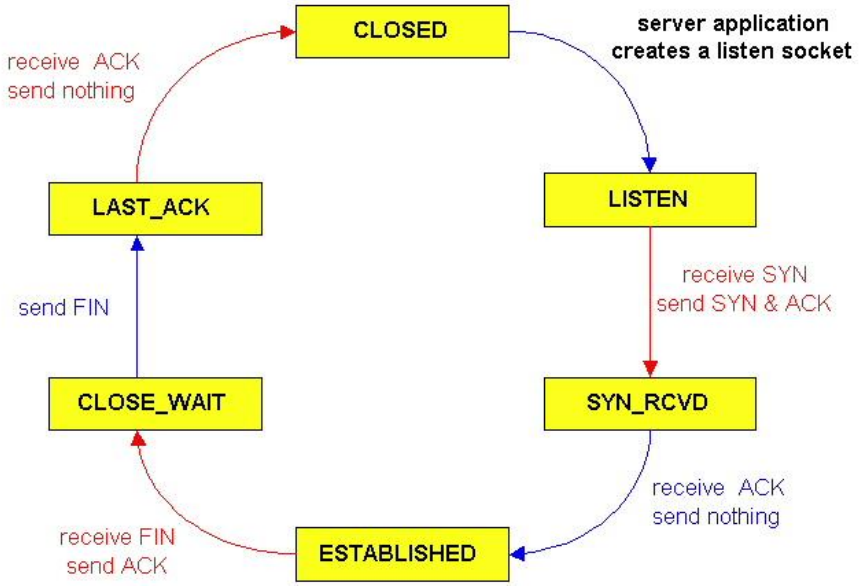


# TCP: Gerenciamento de Conexões (cont.)

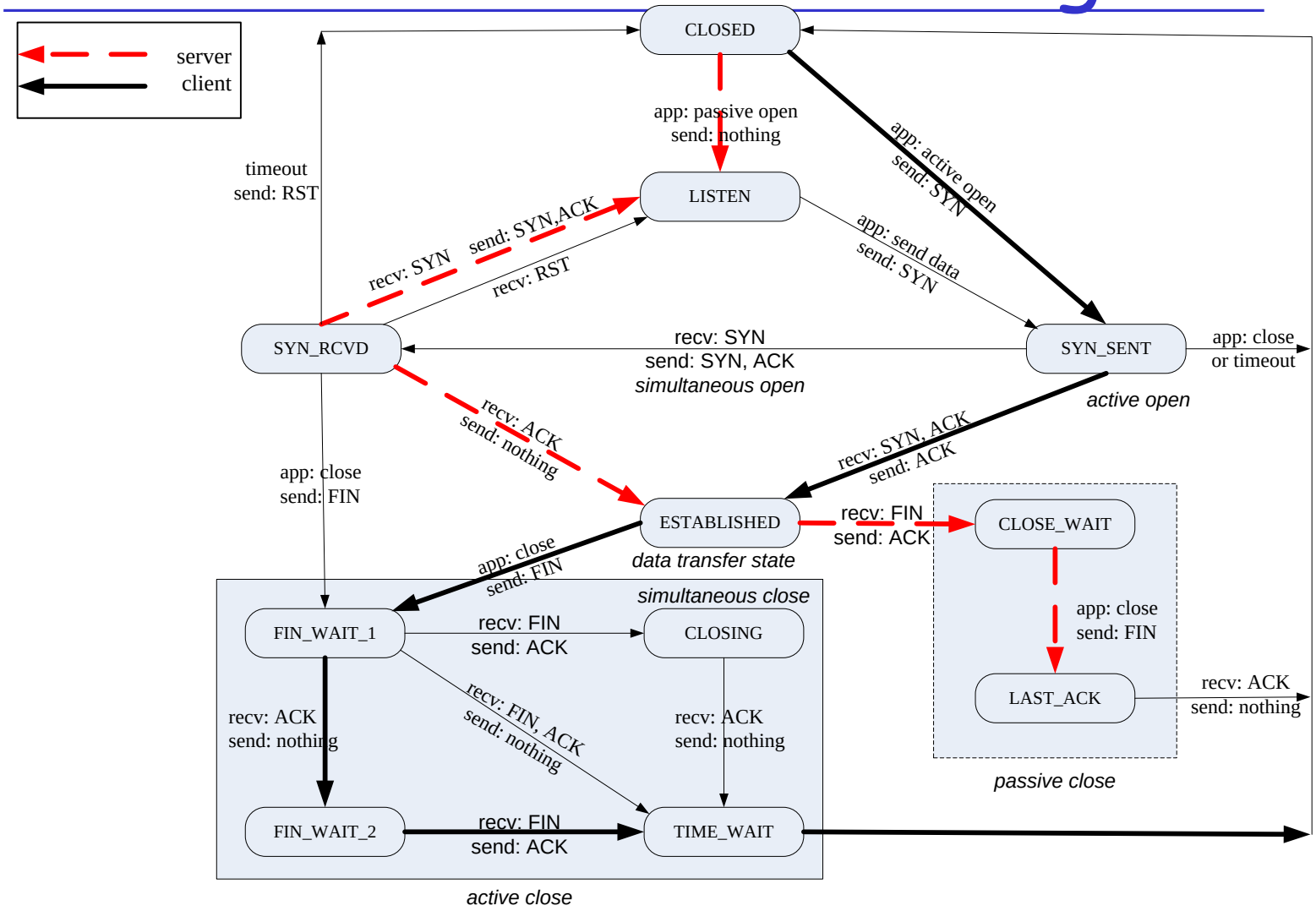



Ciclo de vida
de cliente TCP

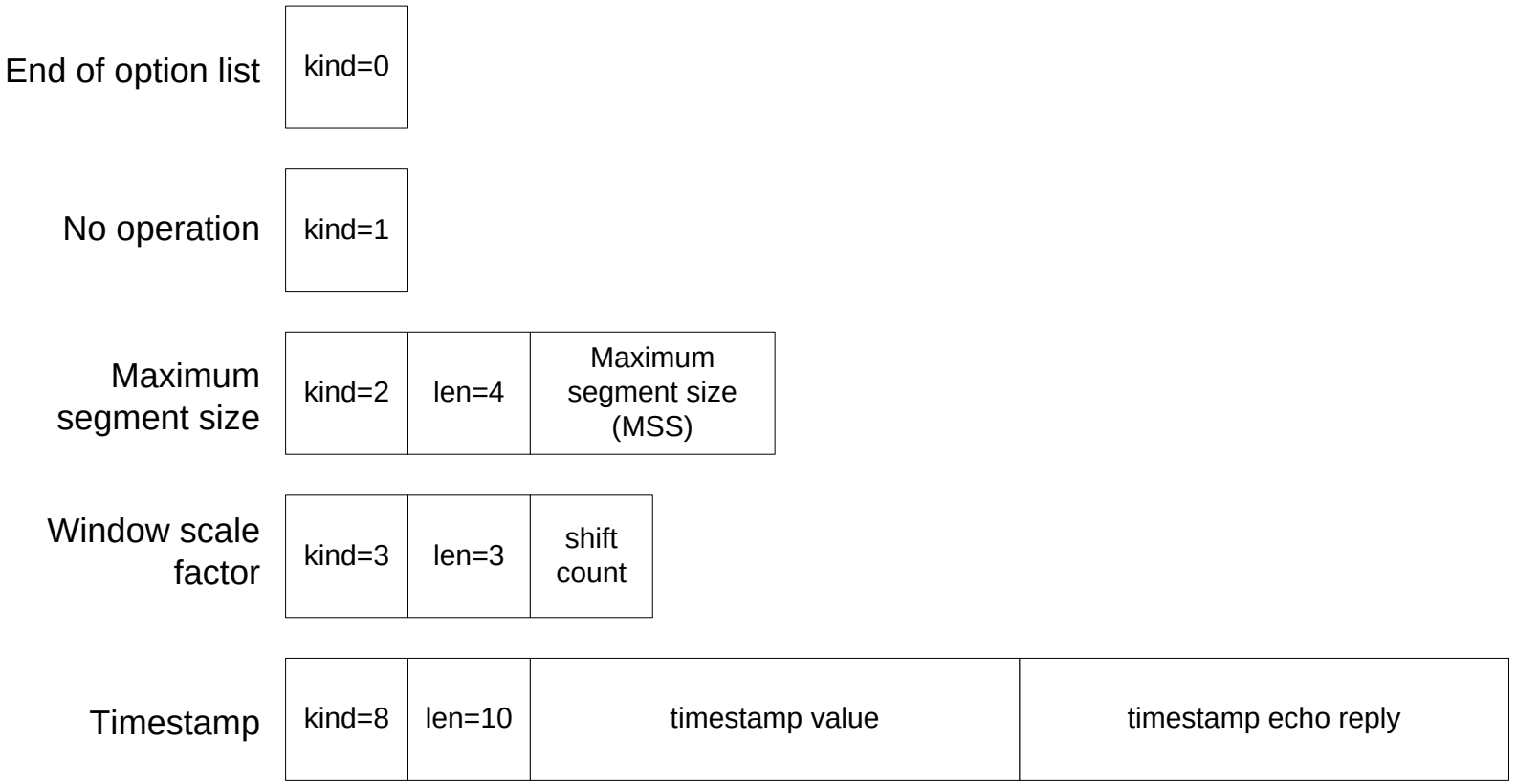
Ciclo de vida
de servidor TCP



# TCP State Transition Diagram

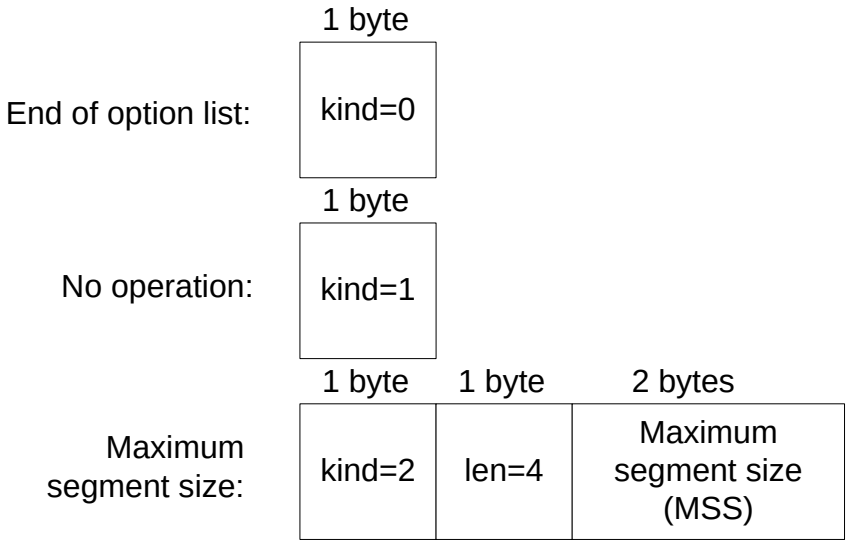


# TCP Options


End of option list
kind=0
No operation
kind=1
Maximum segment size
kind=2
len=4
Maximum segment size (MSS)
Window scale factor
kind=3
len=3
shift count
Timestamp
kind=8
len=10
timestamp value
timestamp echo reply

# TCP Options

- End of Option List
  - As name suggests
- No Operation
  - Padding fields to a multiple of 4 bytes
- Maximum Segment Size
  - Negotiating the max transfer unit at 3-way handshake



# TCP Options (Window Scale Factor, RFC 1323)

- Issue: window too small when in Gigabit networks, causing limited throughput
  - ✓ Solution: negotiate a shifting factor for window
    - Negotiate during 3-way handshaking
      - SYN with timestamp, then SYN+ACK with timestamp
    - Shift up to 14 bits (from $2^{16}$ to $2^{16} \times 2^{14}$)
    - When this option is not used:
      - Linux do not advertise window over $2^{15}$ to avoid other stack that uses signed bit (include/net/tcp.h)

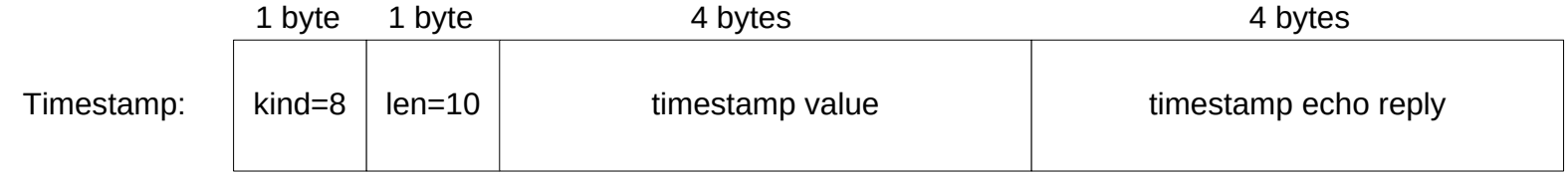
Window scale factor:

| 1 byte | 1 byte | 1 byte |
|---|---|---|
| kind=3 | len=3 | shift count |

```
/*
 * Never offer a window over 32767 without using window scaling. Some
 * poor stacks do signed 16bit maths!
 */
#define MAX_WINDOW      32767
```

# TCP Options – Timestamp

- Mission 1 – Improving RTT measurement
  - ✓ Receiver: copies & replies the timestamp
    - Delayed ACK
  - ✓ Sender: always update RTT when seeing timestamp
- Mission 2 – Protecting Wrapped SeqNum
  - ✓ Avoid receiving old segments in high speed network
- How to enable timestamp option?
  - ✓ 3-way handshake
    - Timestamp in SYN, timestamp in its ACK



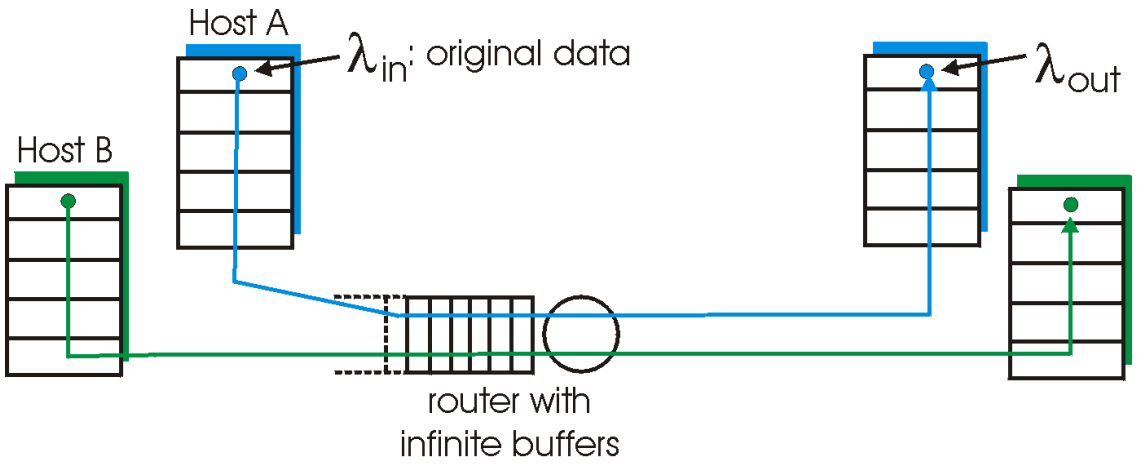
# Princípios de Controle de Congestionamento

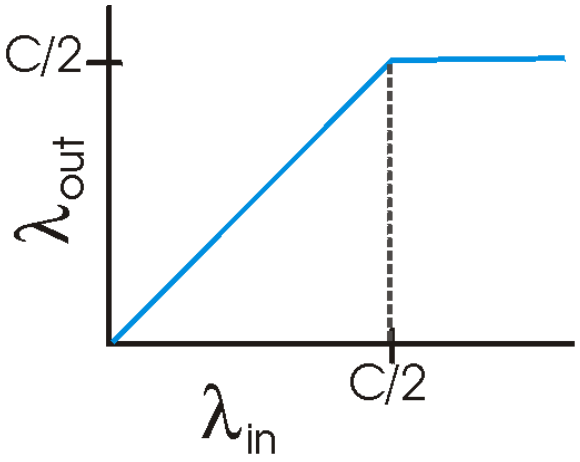
Congestionamento:

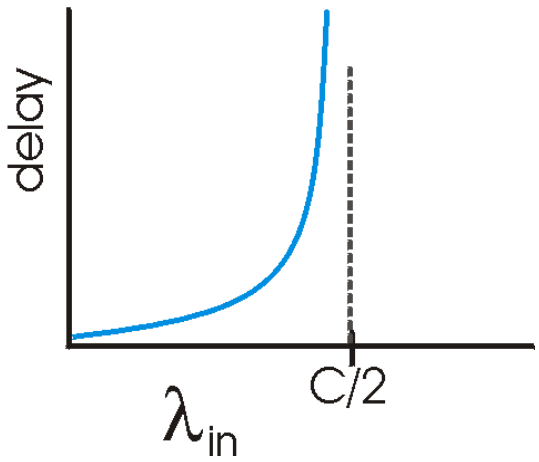
- informalmente: "muitas fontes enviando muitos dados muito rapidamente para a *rede* poder tratar"
- diferente de controle de fluxo!
- manifestações:
  - ✓ perda de pacotes (esgotamento de buffers em roteadores)
  - ✓ longos atrasos (enfileiramento nos buffers dos roteadores)
- um dos 10 problemas mais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

# Causas/custos de congestionamento: cenário 1

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



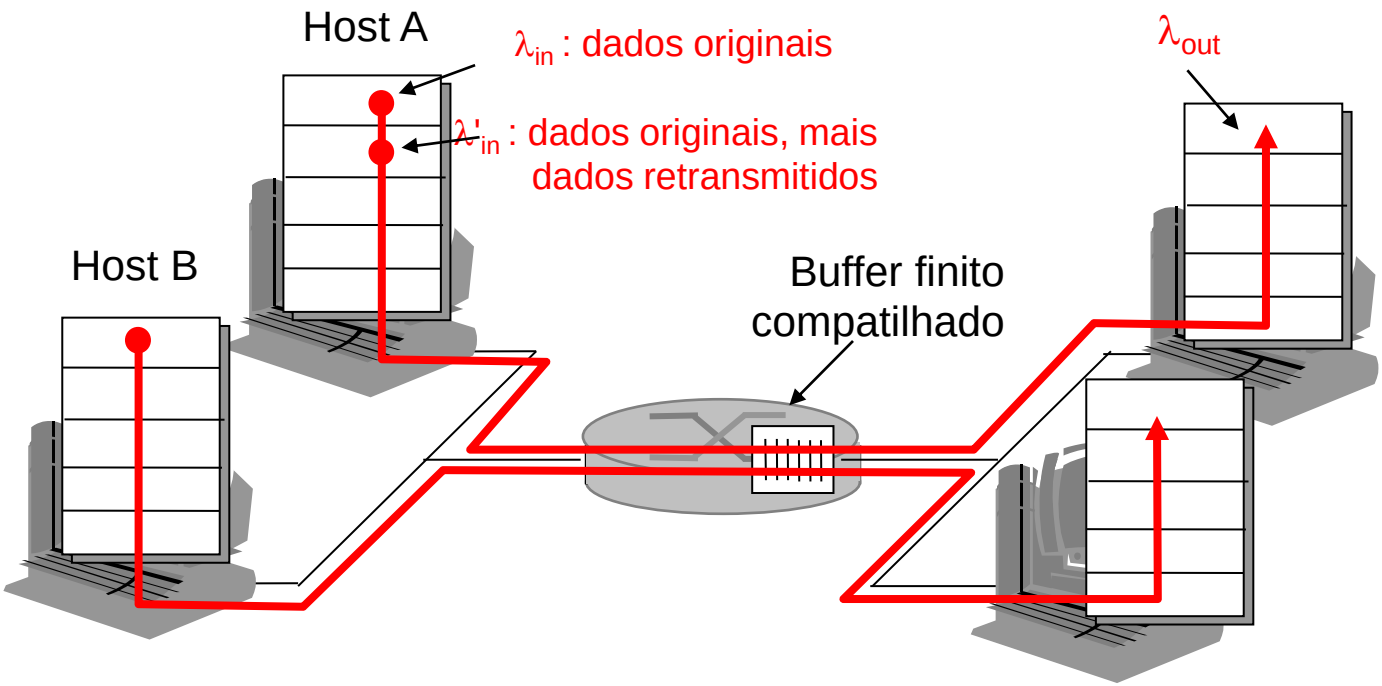




- grandes retardos qdo. congestionada
- vazão máxima alcançável

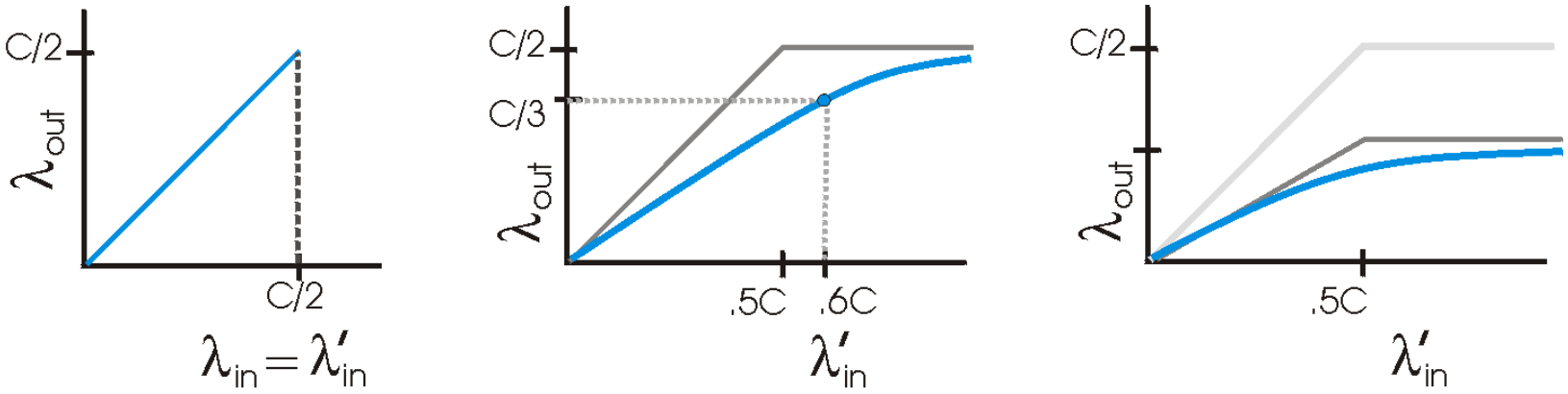
# Causas/custos de congestionamento: cenário 2

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



# Causas/custos de congestionamento: cenário 2

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



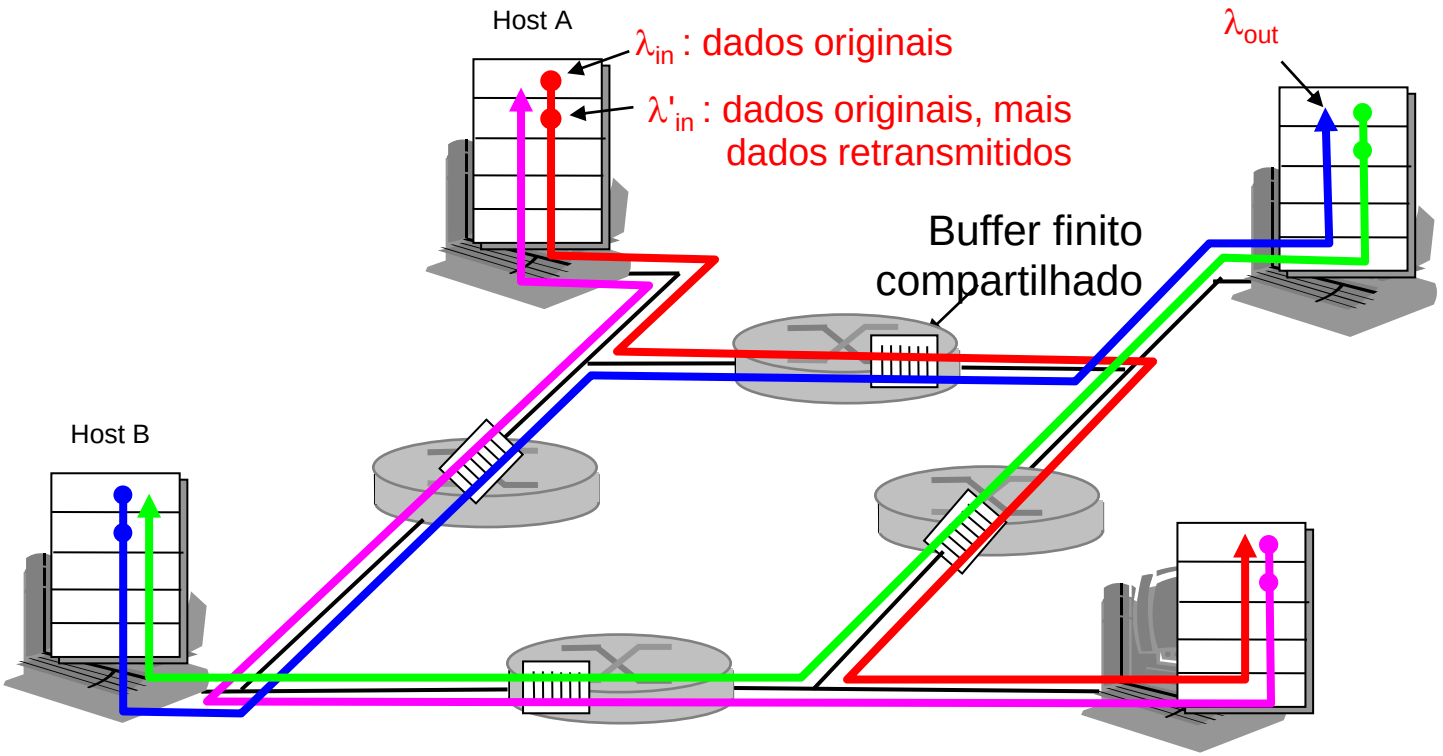

"custos" de congestionamento:

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

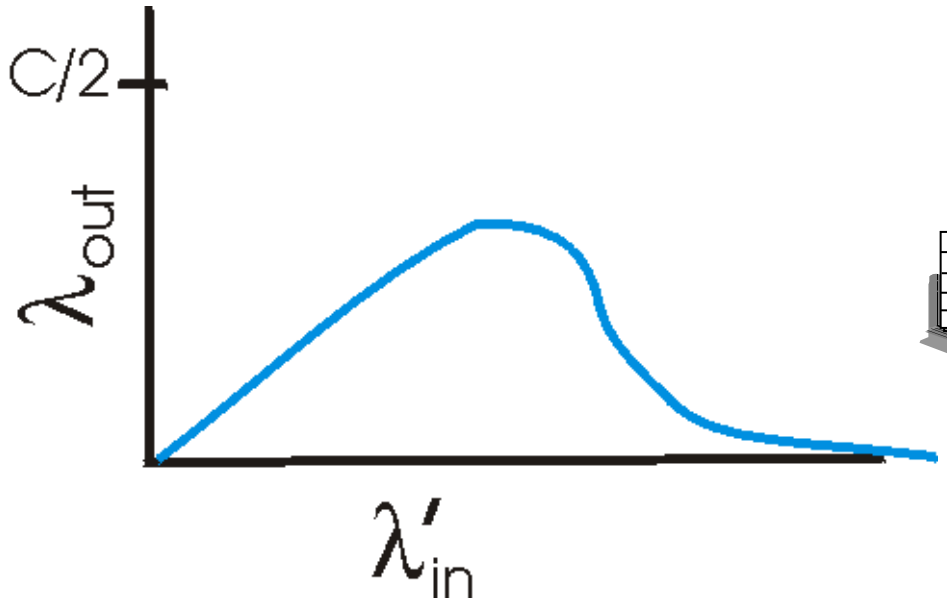
# Causas/custos de congestionamento: cenário 3

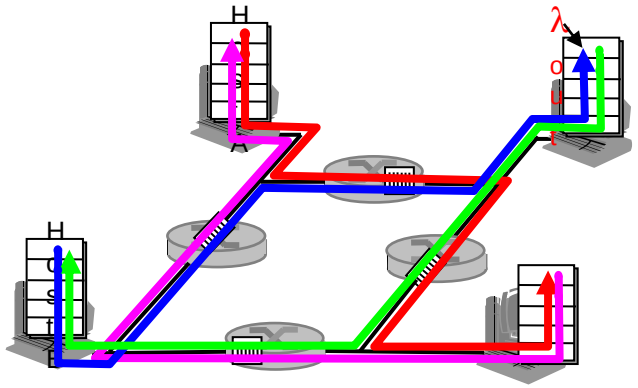
- quatro remetentes
- caminhos com múltiplos enlaces
- temporização/retransmissão

P: o que acontece à medida que $\lambda_{in}$ e $\lambda'_{in}$ crescem ?



# Causas/custos de congestionamento: cenário 3



Outro "custo" de congestionamento:

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

# Abordagens de controle de congestionamento

Duas abordagens amplas para controle de congestionamento:

## Controle de congestionamento fim a fim :

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

## Controle de congestionamento com apoio da rede:

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

# Estudo de caso: controle de congestionamento no ABR da ATM

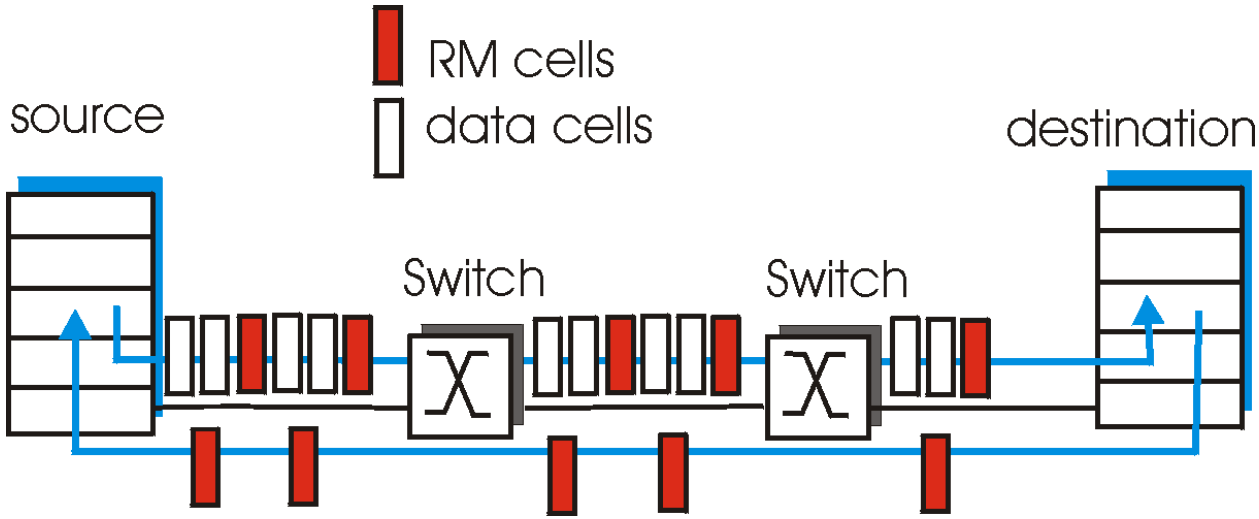
## ABR: available bit rate:

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

## células RM (resource management):

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

# Estudo de caso: controle de congestionamento em ABR da ATM




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

# TCP: Controle de Congestionamento

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

  **LastByteSent-LastByteAcked**

  **≤ CongWin**
- **CongWin** é dinâmica, e é função do congestionamento na rede;

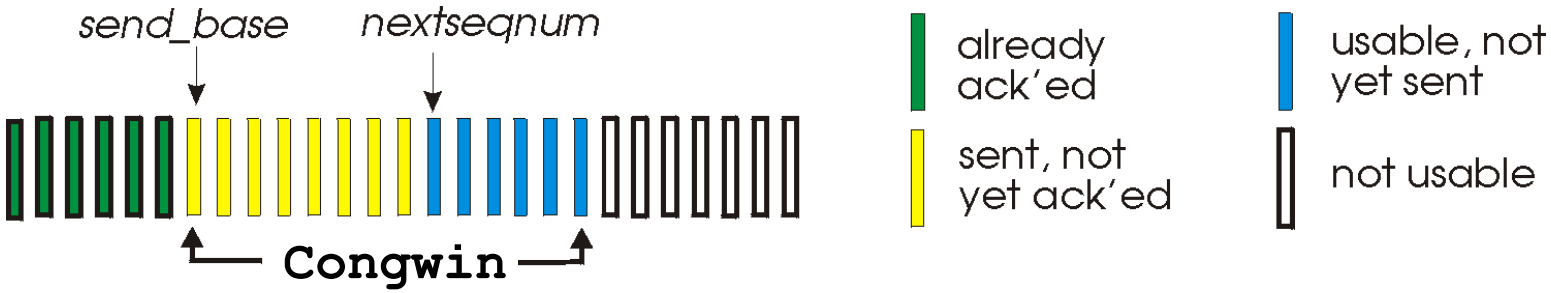
## Como TCP detecta congestionamento?

- Evento de perda = timeout *ou* 3 acks duplicados
- TCP emissor reduz taxa de transmissão (**CongWin**) depois de um evento de perda

## Três mecanismos:

- ✓ AIMD
- ✓ slow start
- ✓ conservative after timeout events

# TCP: Controle de Congestionamento




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

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

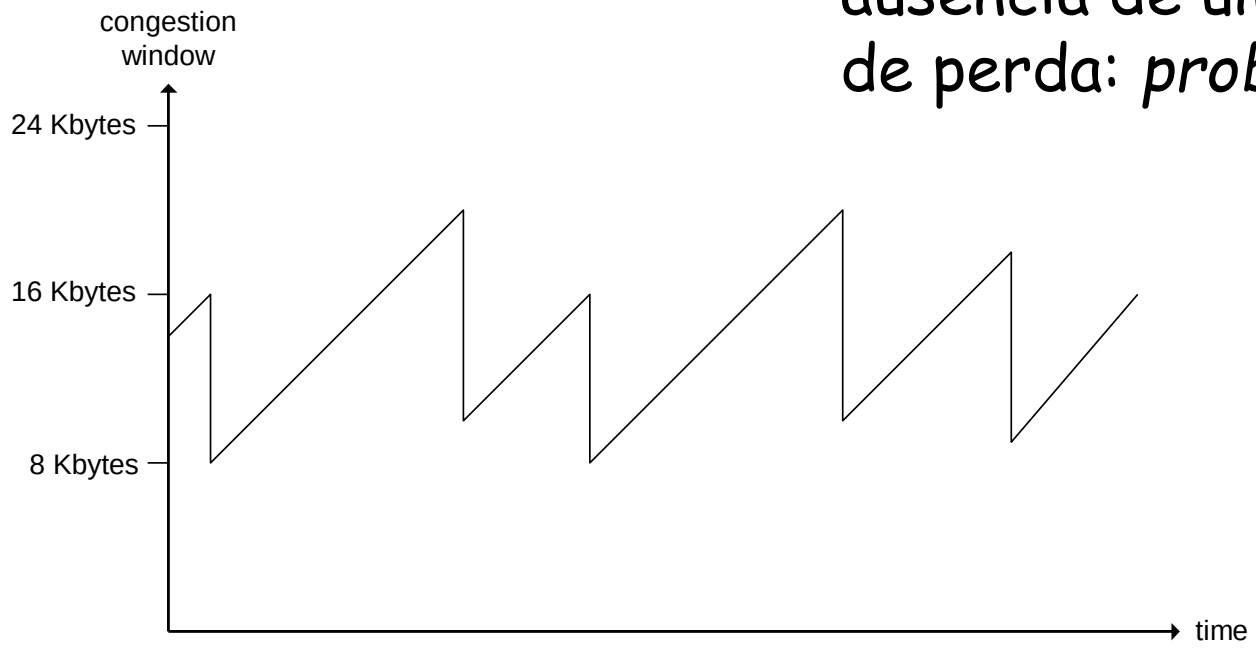
# TCP: Controle de Congestionamento

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

# TCP AIMD

multiplicative decrease (decréscimo multiplicativo: reduz **CongWin** pela metade depois de um evento de perda

additive increase (crescimento aditivo): aumenta **CongWin** de 1 MSS a cada RTT na ausência de um evento de perda: *probing*




Conexão TCP de longa duração

# TCP: Partida lenta (Slow Start)

- Quando a conexão começa, **`CongWin`** = 1 MSS
  - ✓ Exemplo: MSS = 500 bytes & RTT = 200 msec
  - ✓ Taxa inicial = 20 kbps
- A banda disponível deve ser >> MSS/RTT

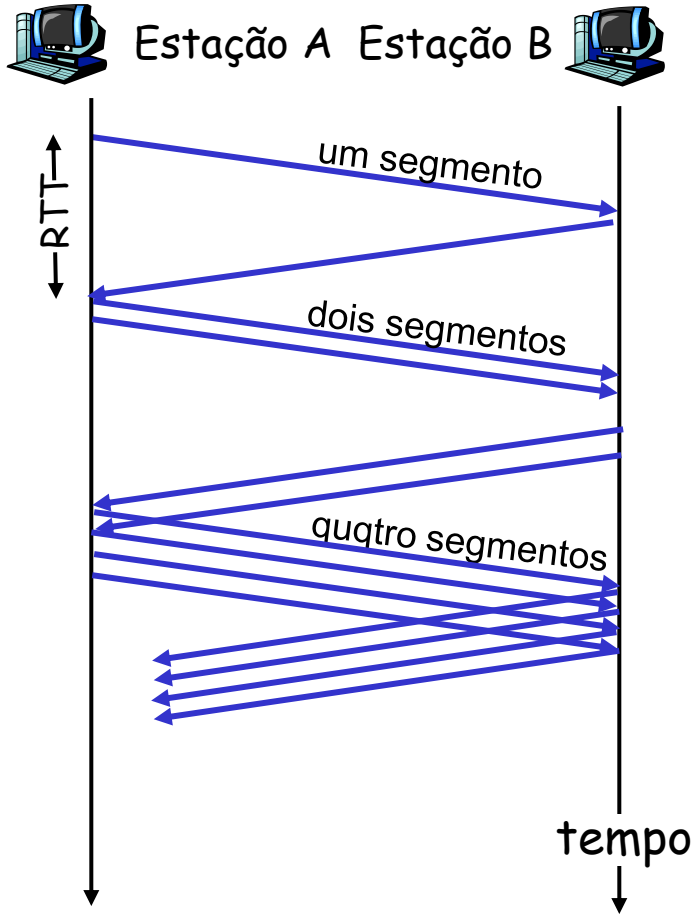
- Quando a conexão começa, aumenta a taxa exponencialmente até o primeiro evento de perda

- Resumo: taxa inicial baixa, mais cresce rapidamente (crescimento exponencial)

# TCP: Partida lenta (slow start)

**Algoritmo Partida Lenta**

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




- Quando a conexão começa, aumenta a taxa exponencialmente até que ococra uma perda
  - Dobra **CongWin** a cada RTT, através do incremento de **CongWin**, a cada ACK recebido

# Refinamento

- Depois de 3 DupACKs:
  - ✓ **CongWin** é reduzida a metade
  - ✓ Janela cresce linearmente
- Mas depois de um evento de timeout:
  - ✓ **CongWin** é reduzida a 1 MSS;
  - ✓ Janela cresce exponencialmente até o valor do threshold, e depois cresce linearmente

**Filosofia:**

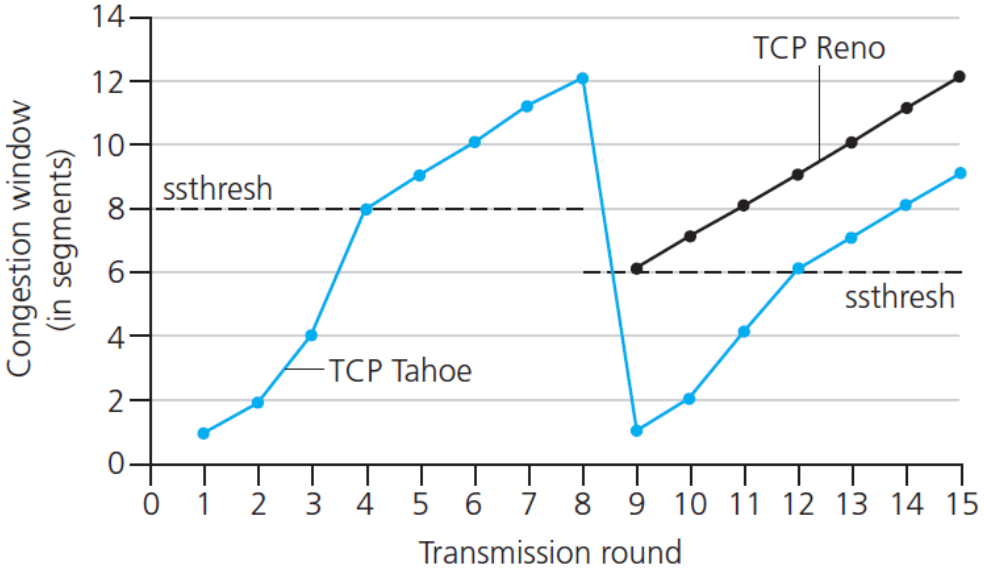
✓ o recebimento de 3 ACKs duplicados indica que a rede tem condição de transmitir alguns segmentos

✓Ocorrência de timeout antes de 3 ACKs duplicados é "mais alarmante"

# TCP: switching from slow start to CA

Q: when should the exponential increase switch to linear?

A: when **cwnd** gets to 1/2 of its value before timeout.



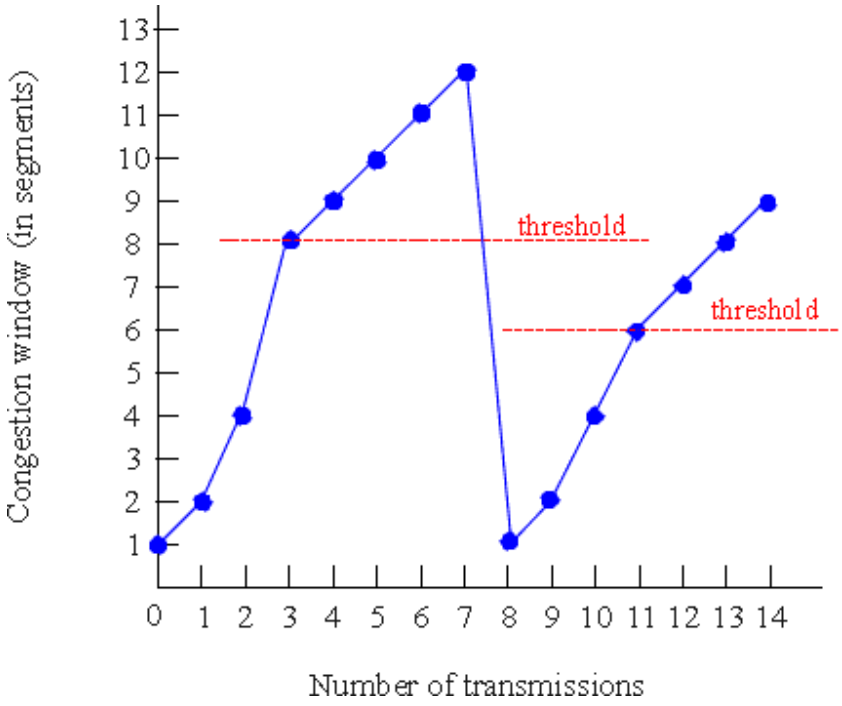

## Implementation:

- variable **ssthresh**
- on loss event, **ssthresh** is set to 1/2 of **cwnd** just before loss event

# TCP: Prevenção do Congestionamento (congestion Avoidance

**prevenção congestionamento**

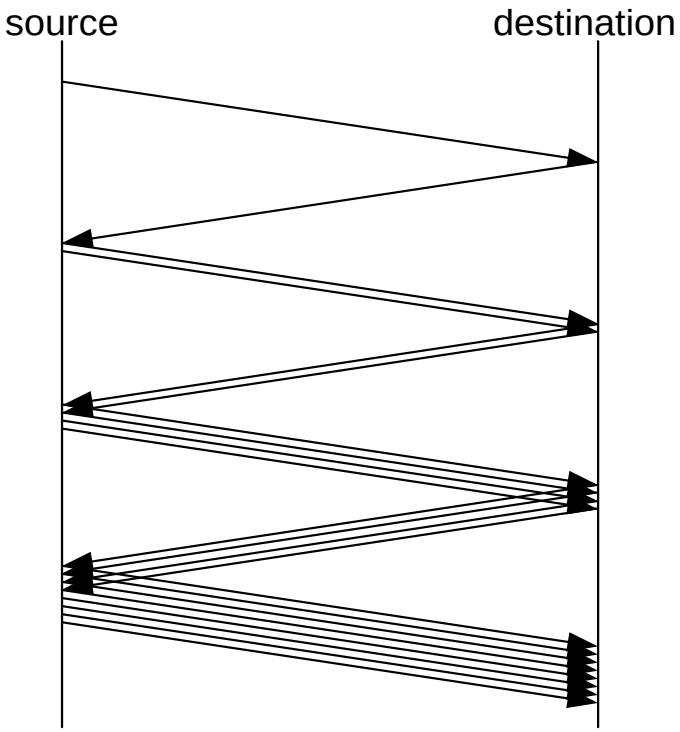
```
/* partida lenta acabou */
/* Congwin > threshold */
Until (event de perda) {
  cada w segmentos
reconhecidos:
      Congwin++
  }
threshold = Congwin/2
Congwin = 1
faça partida lenta      1
```



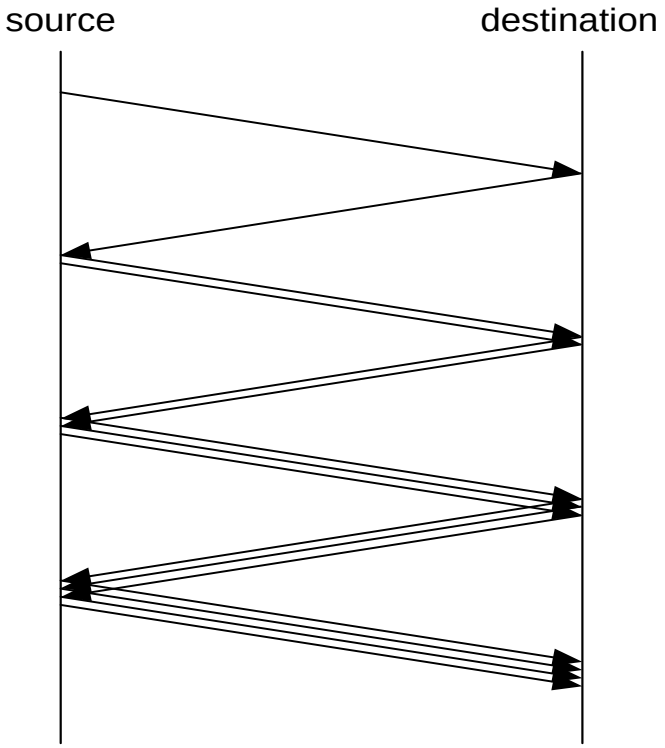

1: TCP Reno pula partida lenta (recuperação rápida) depois de três ACKs duplicados

# Slow Start & Congestion Avoidance

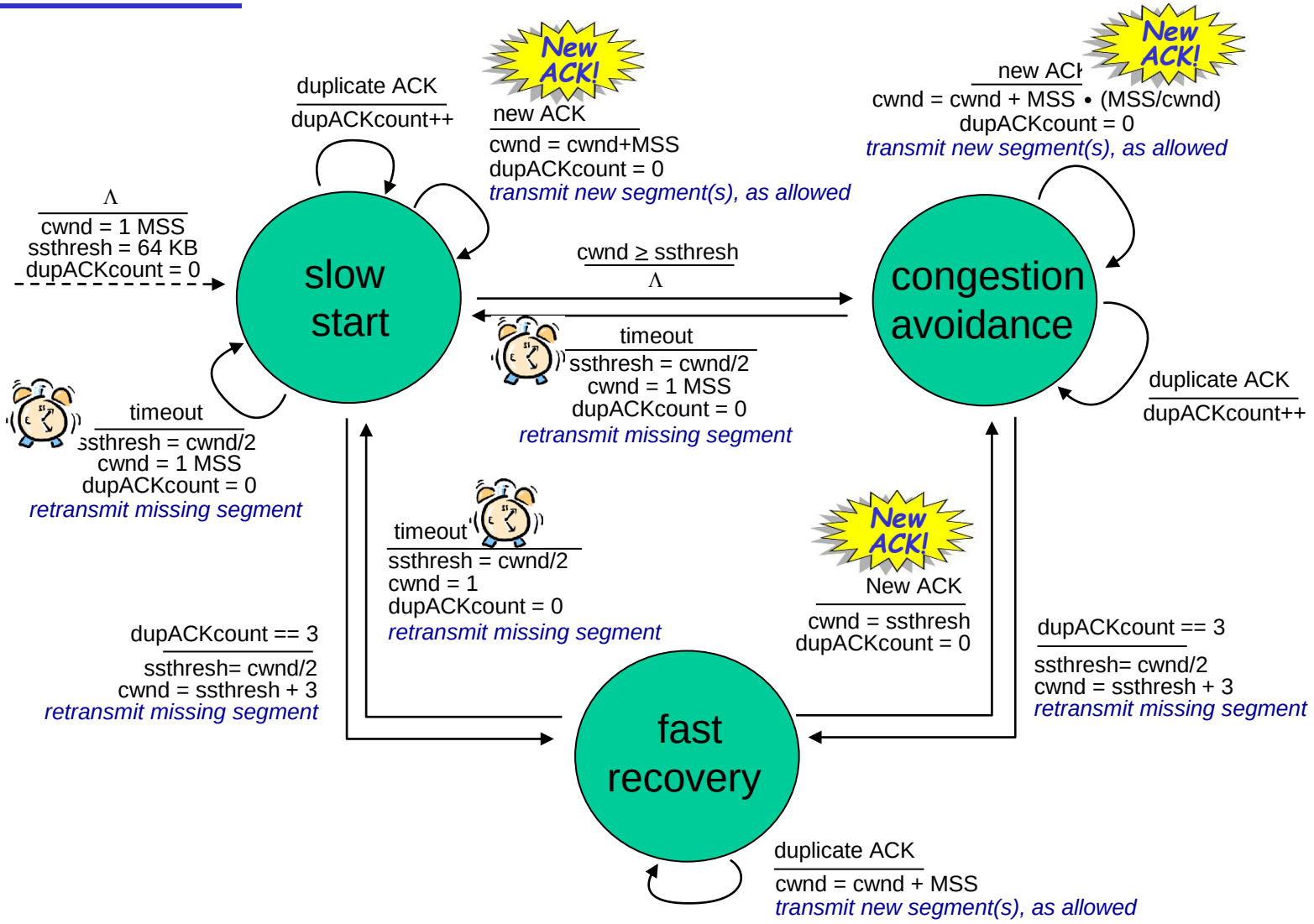
- Slow Start




- Congestion Aviodance



# Summary: TCP Congestion Control

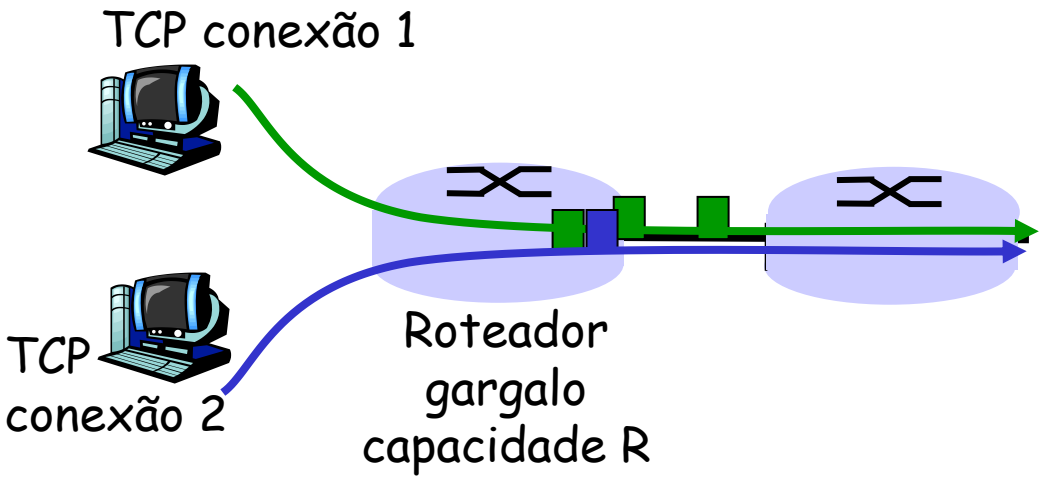


# Resumo: Controle de Congestionamento

- Quando **CongWin** está abaixo do **Threshold**, o emissor está na fase de slow-start, e a janela cresce de 1 MSS a cada ACK recebido
- Quando **CongWin** está acima do **Threshold**, o emissor está na fase de congestion-avoidance, e a janela cresce de 1/n, onde n é otamanho da janela, a cada ACK recebido
- Quando são recebidos três ACK duplicados, faz-se **Threshold = CongWin/2** e **CongWin = Threshold**.
- Quando ocorre um timeout, faz-se **Threshold = CongWin/2** e **CongWin** = 1 MSS.

# Eqüidade TCP

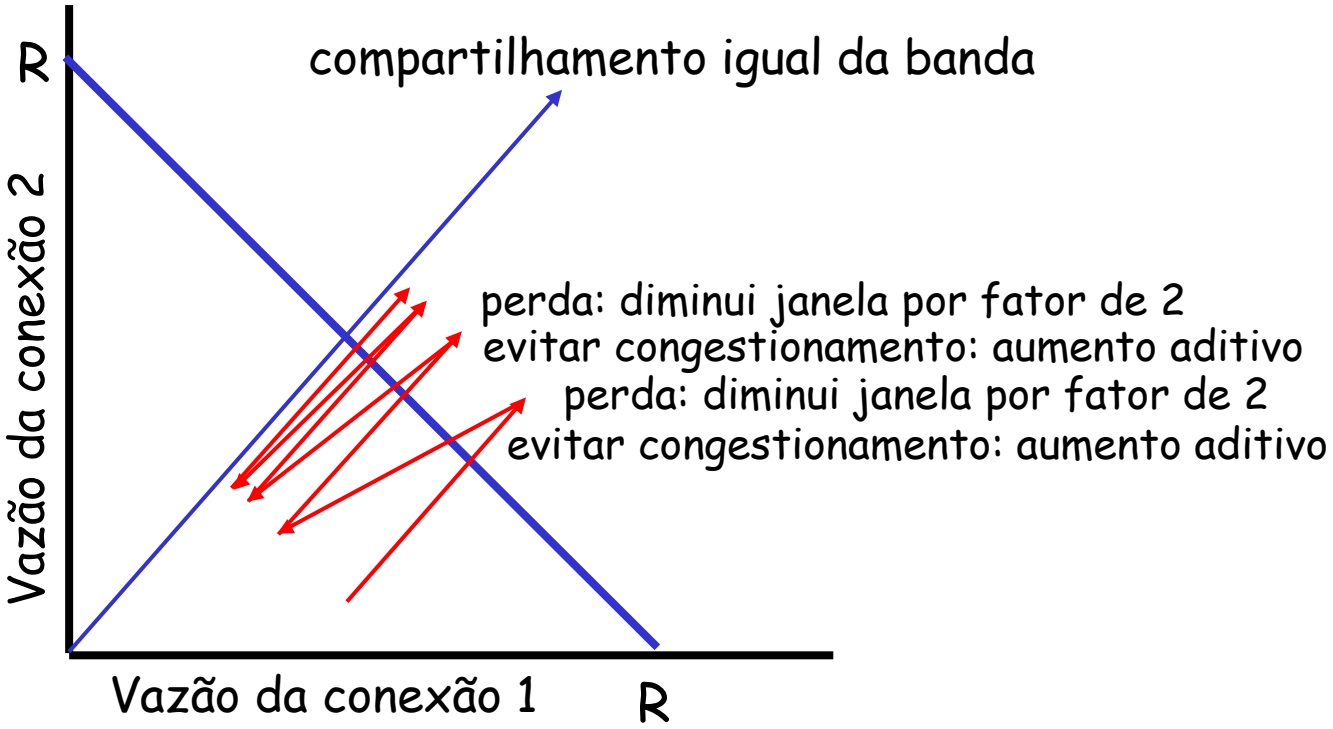
Meta de eqüidade: se N sessões TCP compartilham o mesmo enlace de gargalo, cada uma deve ganhar 1/N da capacidade do enlace



# Por quê TCP é justo?

Duas sessões concorrentes:

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



# Eqüidade (mais)

## Eqüidade e UDP

- Aplic. Multimídia geralmente não usam TCP
  - ✓ Não desejam que a taxa seja reduzida pelo controle de congestionamento
- Geralmente usam UDP:
  - ✓ audio/video a taxa constante, toleram perdas de pacotes
- Área de pesquisa: TCP friendly

## Eqüidade e conexões TCP paralelas

- Nada previne que aplic. abram várias conexões simultânceas entre os 2 hosts;
- Wrowsers fazem isto
- Exemplo: enlace com taxa igual a R, com 9 conexões;
  - ✓ Nova aplic. requer uma conexão TCP, recebe R/10 da taxa
  - ✓ Nova aplic. requer 11 conexões TCP, recebe R/2 da taxa

# Redes com Alto Produto Banda Atraso

Banda (passante) X Atraso

Redes de alta velocidade (alta capacidade)

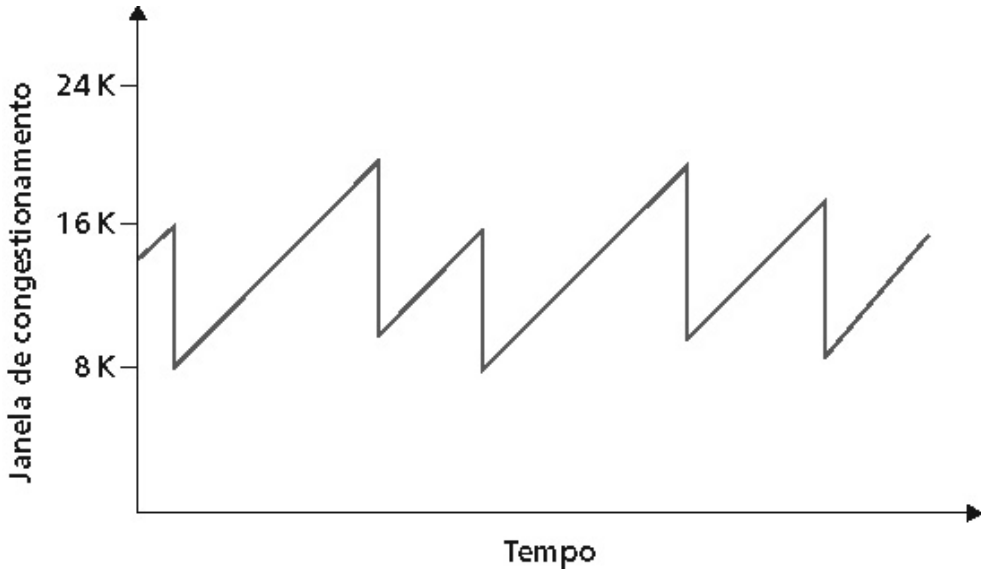
Satélite

# TCP para Redes de Alta Velocidade

- Janela de Congestionamento TCP Reno cresce lentamente e
- Seu tamanho e drasticamente reduzido em eventos de perda

- Ineficiência para operar em redes de alta velocidade

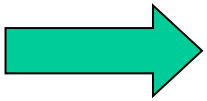
# TCP AIMD




- *Upon an ACK arrival:* $W \leftarrow W + 1/W$
- *Upon a loss detection:* $W \leftarrow W/2$

# TCP para Redes de Alta Velocidade

- Para que uma conexao do TCP Reno,
- segmentos de 1500 bytes em
- enlace de 10Gbps e 100 ms de atraso de propagacao,



- Tamanho de janela para utilizar toda a capcidade do enlace em uma hora e meia, o que implica
- taxa de perda de $10^{-11}$,

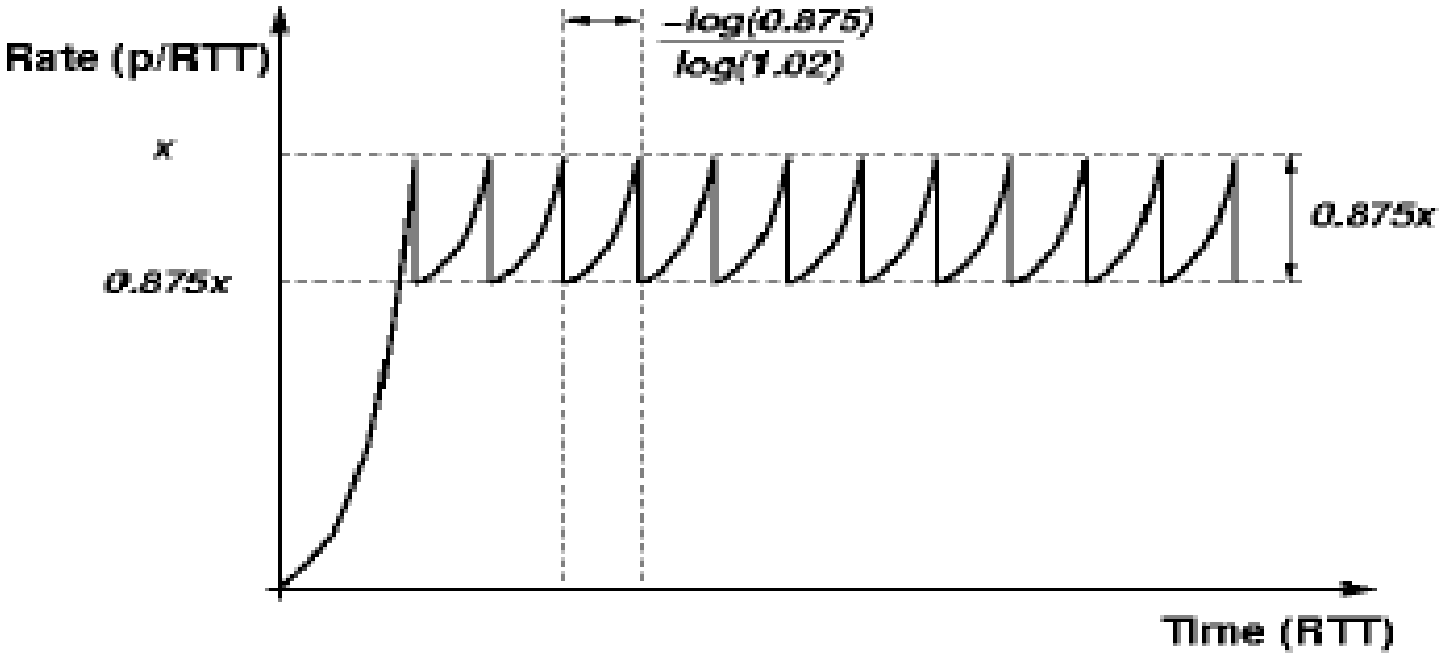
# TCP para Redes de Alta Velocidade

Propriedades Desejáveis:

- Justiça intra-protocolo;
- Amigável ao TCP-Reno;
- Rápido em reação;
- Estável;
- Facil difusão.

# Scalable TCP

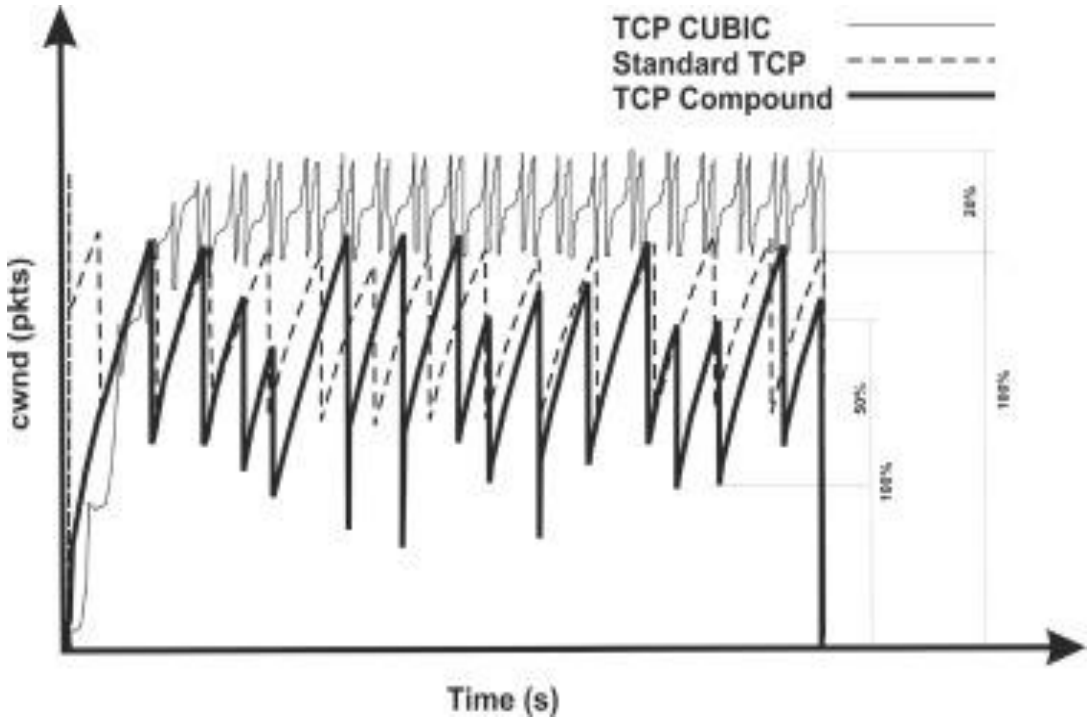
- *It is based on a MIMD algorithm:*
  - $W \leftarrow W + \alpha$ *upon an ACK arrival*
  - $W \leftarrow W - \beta W$ *upon a loss detection*



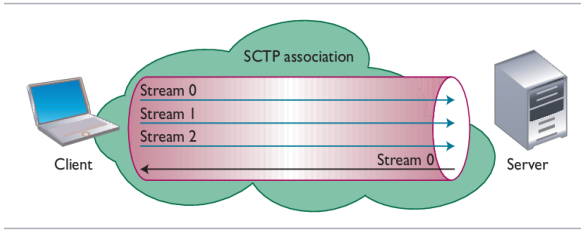
| Variante | Principal Característica | Mecanismo de controle de congestionamento |
|---|---|---|
| HS-TCP | Relação linear em uma escala log-log entre taxa de transmissão e evento denotando congestionamento | Dois modos: TCP-Reno AIMD e Crescimento Rapido (Fast Increase) |
| FAST-TCP | Quatro modos independentes: estimação, controle de dados, controle de janela e controle de rajadas | Baseado no Atraso |
| Scalable-TCP | Utilização eficiente independente de capacidade do canal | Multiplicative Increase Multiplicative Decrease (MIMD) |
| BIC | Busca dinâmica para tamanho de janela | Aditive Increase and Binary Search Increase, Multiplicative Descrease (AIBIMD) |

| Variante | Principal Característica | Mecanismo de controle de congestionamento |
|---|---|---|
| CUBIC | Busca dinâmica para estabelecr tamanho da janela | Additive increase and cubic decrease, multiplicative decrease (AICIMD) |
| Compound-TCP | Abordagem híbrida | AIMD e baseado no atraso |
| TCP-Africa | Baseado no Atraso | AIMD e Crescimento rápido |
| H-TCP | Rápida convergencia para equilíbrio de compartilhamento de banda | AIMD e Crescimento Rápido (Fast Increase) |
| TCP-Libra | Amigável a RTT, componentes para controle de: justiça, estimação capacidade, rajadas, escalabilidade e estabilidade | AIMD |

# TCP CUBIC



# SCTP




| Services/Features | SCTP | TCP | UDP |
|---|---|---|---|
| Full-duplex data transmission | yes | yes | yes |
| Connection-oriented | yes | yes | no |
| Reliable data transfer | yes | yes | no |
| Partially reliable data transfer | optional | no | no |
| Ordered data delivery | yes | yes | no |
| Unordered data delivery | yes | no | yes |
| Flow and congestion control | yes | yes | no |
| Explicit congestion notification support | yes | yes | no |
| Selective acks | yes | optional | no |
| Preservation of message boundaries | yes | no | yes |
| Path maximum transmission unit discovery | yes | yes | no |
| Application data fragmentation/bundling | yes | yes | no |
| Multistreaming | yes | no | no |
| Multihoming | yes | no | no |
| Protection against SYN flooding attack | yes | no | n/a |
| Half-closed connections | no | yes | n/a |

# DCCP

| Feature | UDP | TCP | DCCP |
|---|---|---|---|
| **Packet size** | 8 bytes | 20 bytes | **12 or 16 bytes** |
| **Transport layer packet entity** | Datagram | Segment | **Datagram** |
| **Port numbering** | Yes | Yes | Yes |
| **Error detection** | Optional | Yes | Yes |
| **Reliability: Error recovery by ARQ** | No | Yes | **No** |
| **Sequence numbering and reordering** | No | Yes | **Yes/No** |
| **Flow control** | No | Yes | Yes |
| **Congestion Control** | No | Yes | Yes |
| **ECN support** | No | Yes | Yes |

# Capítulo 3: Resumo

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

## Próximo capítulo:

- saímos da "borda" da rede (camadas de aplicação e transporte)
- entramos no "núcleo"da rede