# SSC0641
# Redes de Computadores

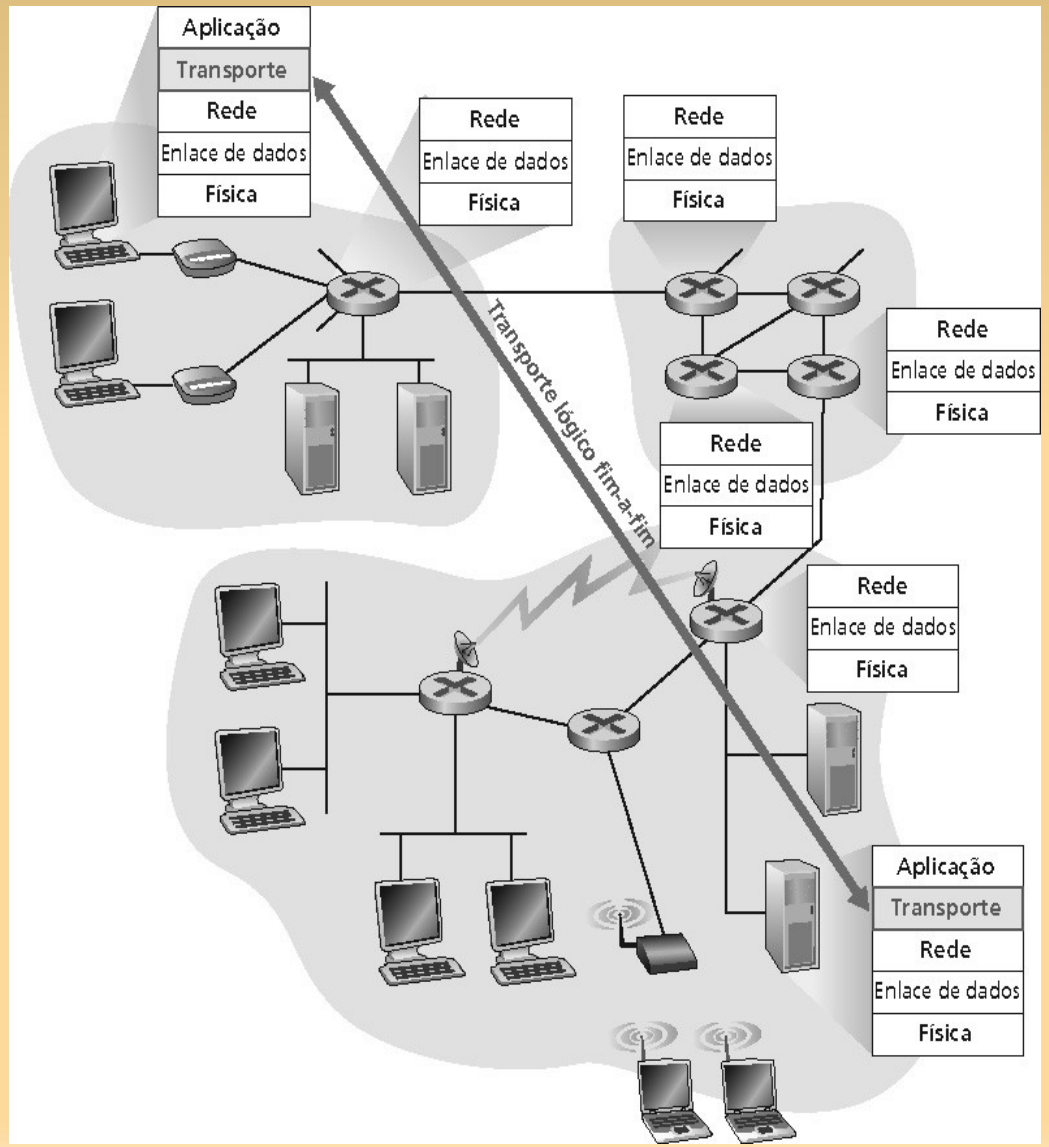
## Capítulo 3 - Camada de Transporte

Prof. Jó Ueyama

Abril/2013

# Serviços da Camada de Transporte

# Protocolos e Serviços de Transporte



# Serviços de Transporte

- Fornecem **comunicação lógica** entre processos de aplicação em diferentes hospedeiros:
  - protocolos de transporte são executados nos sistemas finais;
  - Lado emissor: quebra as mensagens da aplicação em segmentos e envia para a camada de rede;
  - Lado receptor: remonta os segmentos em mensagens e passa para a camada de aplicação.

# Protocolos de Transporte

- Há mais de um protocolo de transporte disponível para as aplicações na Internet:
  - TCP;
  - UDP.

# Protocolos da Camada de Transporte

- TCP: confiável, garante ordem de entrega.
  - controle de congestionamento;
  - controle de fluxo;
  - orientado à conexão.
- UDP: não-confiável, sem ordem de entrega.
  - extensão do “melhor esforço” do IP.
  - Nenhum QoS; nehuma prealocação de recursos
- Serviços não disponíveis:
  - garantia a atrasos;
  - garantia de banda.

# Camada de transporte *versus* rede

- *camada de rede:* comunicação lógica entre hospedeiros
- *camada de transporte:* comunicação lógica entre processos
  - conta com e amplia os serviços da camada de rede

analogia com a família:

*12 crianças mandando carta a 12 crianças*

- processos = crianças
- msgs da aplicação = cartas nos envelopes
- hospedeiros = casas
- protocolo de transporte = Ana e Bill
- protocolo da camada de rede = serviço postal

# Multiplexação / Demultiplexação

# Multiplexação / Demultiplexação

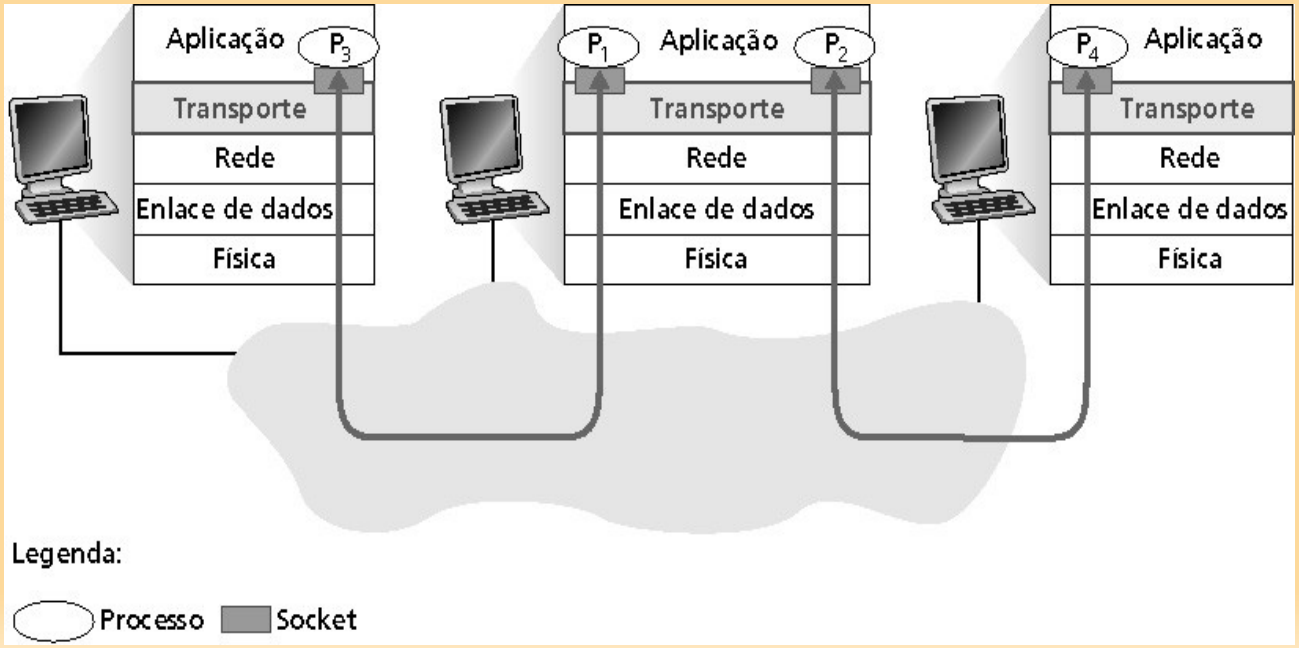
**Demultiplexação no receptor:**

• entrega os segmentos recebidos ao socket correto.

**Multiplexação no emissor:**

• coleta dados de múltiplos sockets;

• envelopa os dados com cabeçalho (usado depois para demultiplexação).



# Como funciona a demultiplexação

- Computador recebe datagramas IP:
  - cada datagrama possui endereço IP de origem e IP de destino;
  - cada datagrama carrega 1 segmento da camada de transporte;
  - cada segmento possui números de porta de origem e destino.
- O hospedeiro usa endereços IP (32 bits) e números de porta (16 bits) para direcionar o segmento ao socket apropriado.

# Demultiplexação não orientada à conexão

- Cria sockets com números de porta:

```
DatagramSocket mySocket1 = new
  DatagramSocket(99111); // no servidor

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

- Socket UDP identificado por dois valores:
  - endereço IP de destino,
  - número da porta de destino.

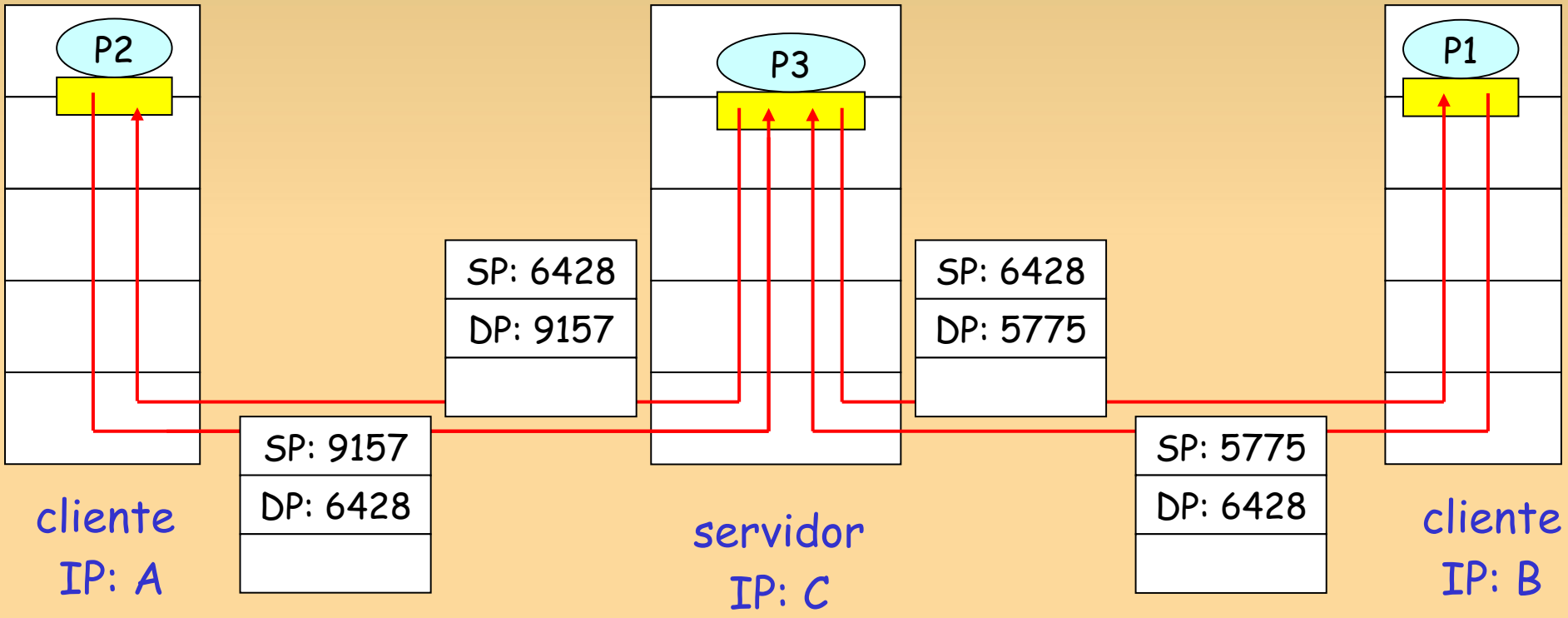
```
DatagramPacket sendpacket = new
  DatagramPaket(sendData, sendData.length,
  IPAddress, 99222);
DatagramSocket clientSocket = new
  DatagramSocket();
clientSocket.send(sendpacket);
```

# Demultiplexação não orientada à conexão (2)

- Quando o hospedeiro recebe o segmento UDP:
  - verifica o número da porta de destino no segmento;
  - direciona o segmento UDP para o socket com este número de porta.
- Datagramas com IP de origem diferentes e/ou portas de origem diferentes são direcionados para o mesmo socket.

# Demultiplexação não orientada à conexão

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



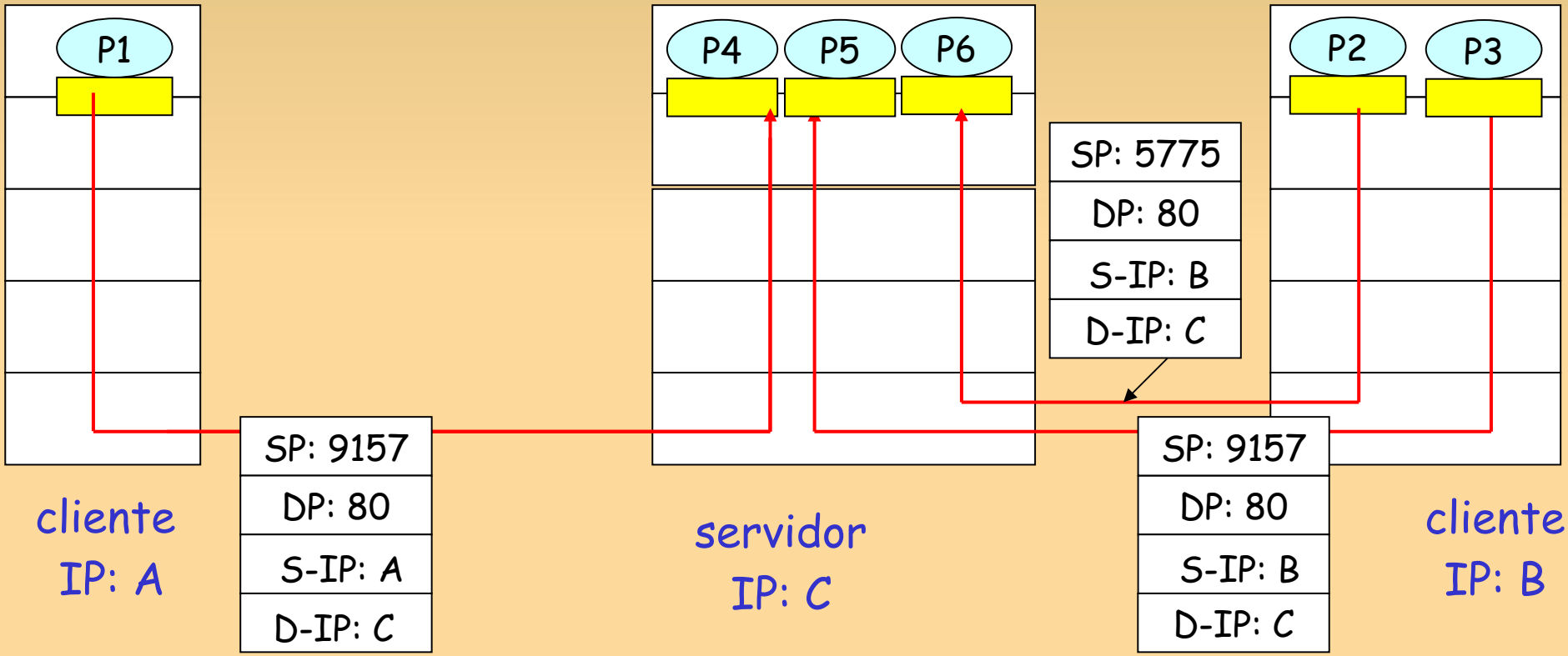
# Demultiplexação orientada à conexão

- TCP estabelece conexão.
- Conexão TCP (ou seja sockets servidor e cliente) identificada por 4 valores:
  - Endereço IP de origem
  - Porta de origem
  - Endereço IP de destino
  - Porta de destino
- Host receptor usa os quatro valores para direcionar o segmento ao socket apropriado.

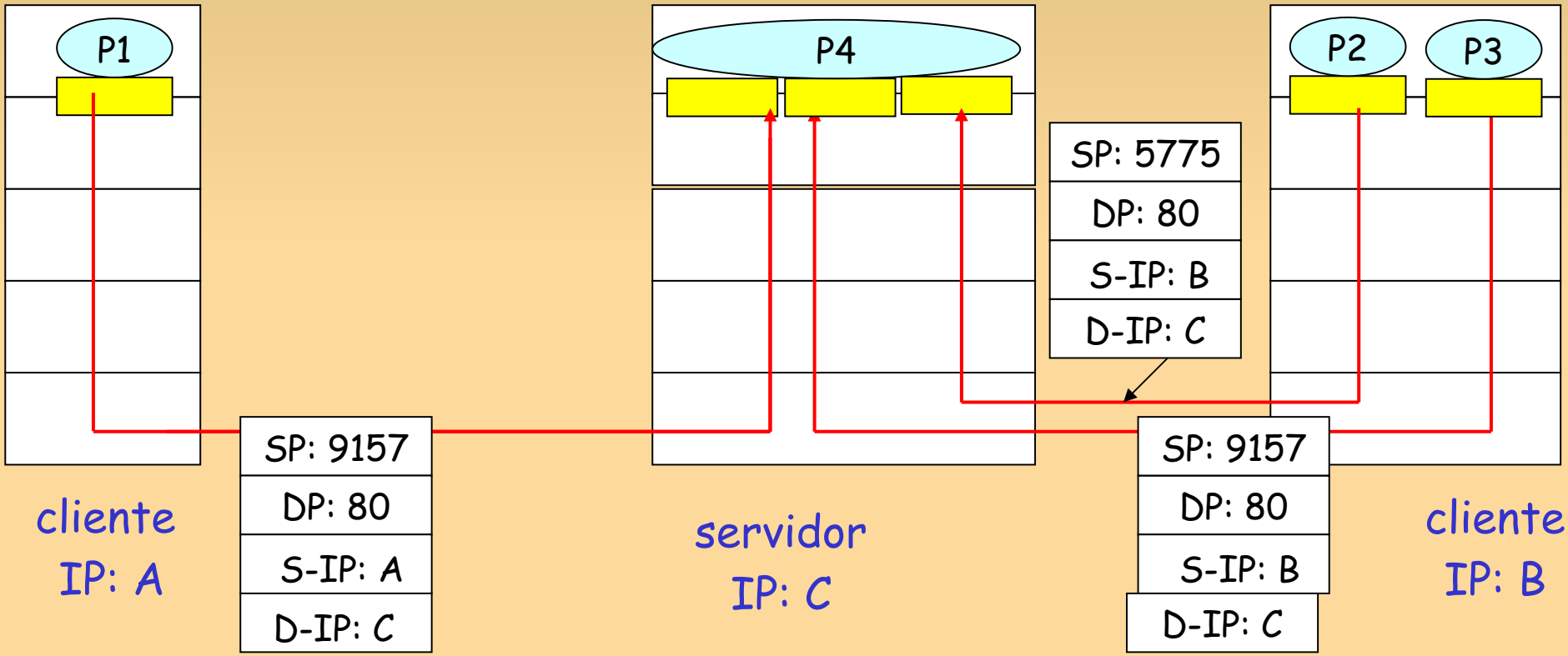
# Demultiplexação orientada à conexão

- Host servidor pode suportar vários sockets TCP simultâneos:
  - cada socket é identificado pelos seus próprios 4 valores;
  - servidores Web possuem sockets diferentes para cada cliente conectado;
  - HTTP não persistente terá um socket diferente para cada requisição.

# Demultiplexação orientada à conexão



# Servidor Web multi-threaded



# Transporte não orientado à conexão: UDP

# UDP (User Datagram Protocol)

- Definido pela RFC 768.
- Serviço de melhor esforço (*best effort*), sem *overheads*.
- Segmentos UDP podem ser:
  - perdidos;
  - entregues fora de ordem para a aplicação.
- Sem conexão:
  - não há apresentação entre transmissor e o receptor UDP;
  - cada segmento UDP é tratado de forma independente dos outros.

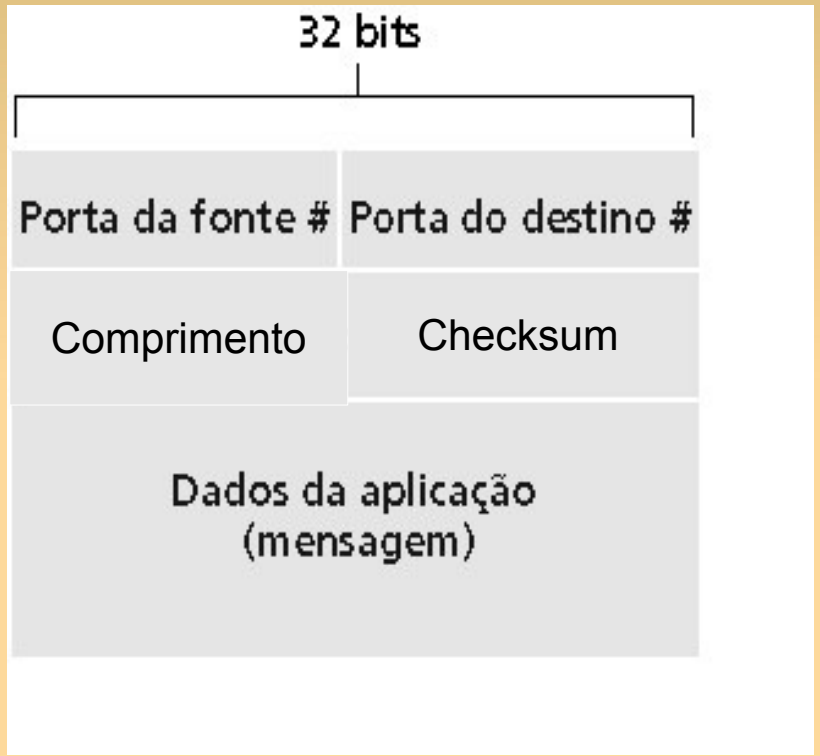
# Por que UDP?

- Não há estabelecimento de conexão (que possa redundar em atrasos).
- Simples:
  - não há estado de conexão nem no transmissor, nem no receptor.
- Cabeçalho de segmento reduzido.
- Não há controle de congestionamento:
  - UDP pode enviar segmentos tão rápido quanto desejado (e possível).

# Uso do UDP

- Muito usado por aplicações de multimídia contínua (streaming):
  - tolerantes à perda;
  - sensíveis à taxa.
- DNS.
- SNMP.
- Transferência confiável sobre UDP:
  - acrescentar confiabilidade na camada de aplicação;
  - recuperação de erro específica de cada aplicação.

# Segmento UDP



# Soma de Verificação (*Checksum*) UDP

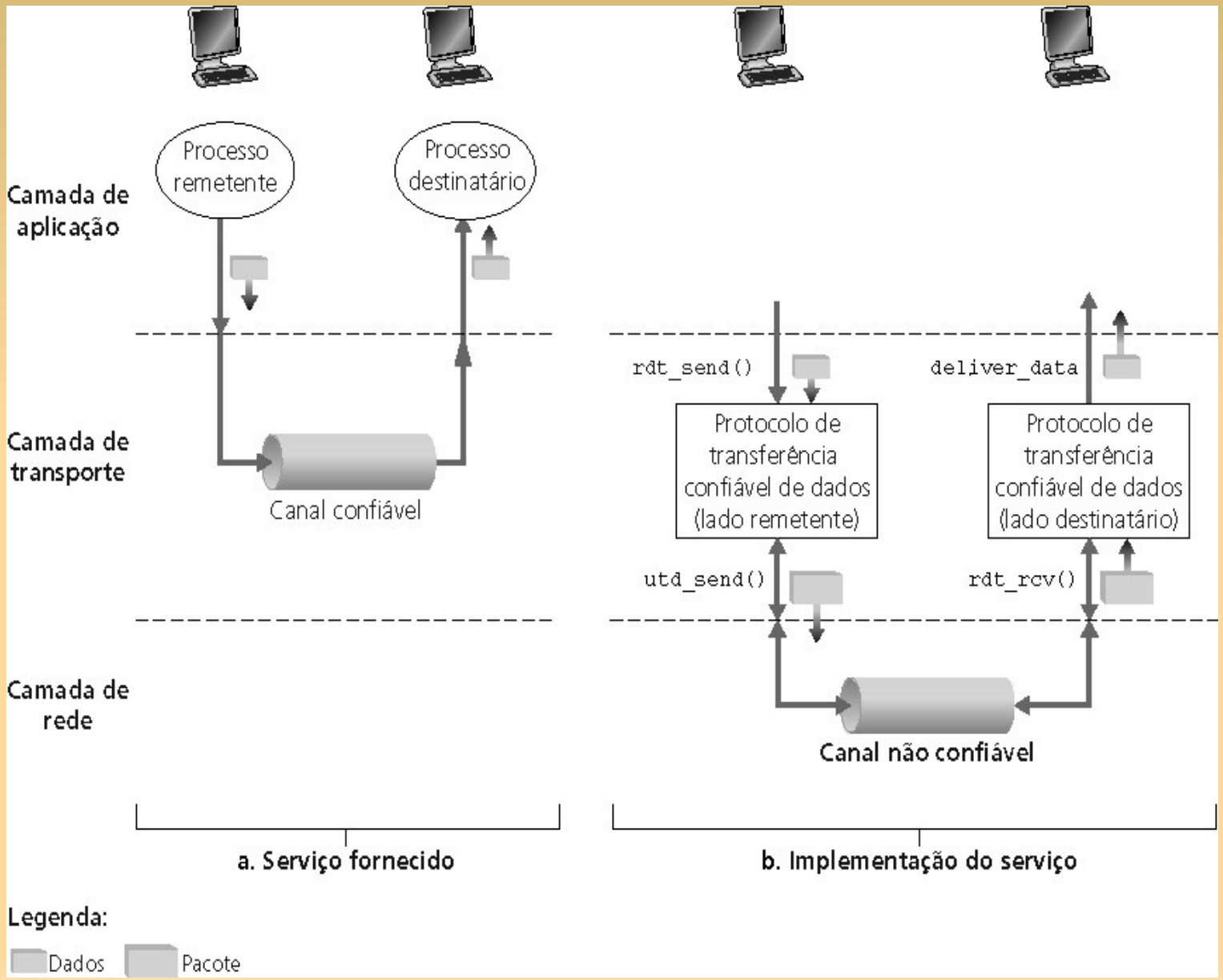
- Objetivo:
  - detectar “erros” (ex.: bits trocados) no segmento transmitido.
- Transmissor:
  - trata o conteúdo do segmento como seqüência de inteiros de 16 bits;
  - checksum: soma (complemento de 1 da soma) do conteúdo do segmento;
  - transmissor coloca o valor do checksum no campo de checksum do UDP.

# Soma de Verificação (*Checksum*) UDP (cont.)

- Receptor:
  - Calcula o checksum do segmento recebido.
  - Verifica se o checksum calculado é igual ao valor do campo checksum:
    - NÃO - erro detectado.
    - SIM - não há erros detectados.

# Transferência confiável de dados

# Transferência confiável de dados



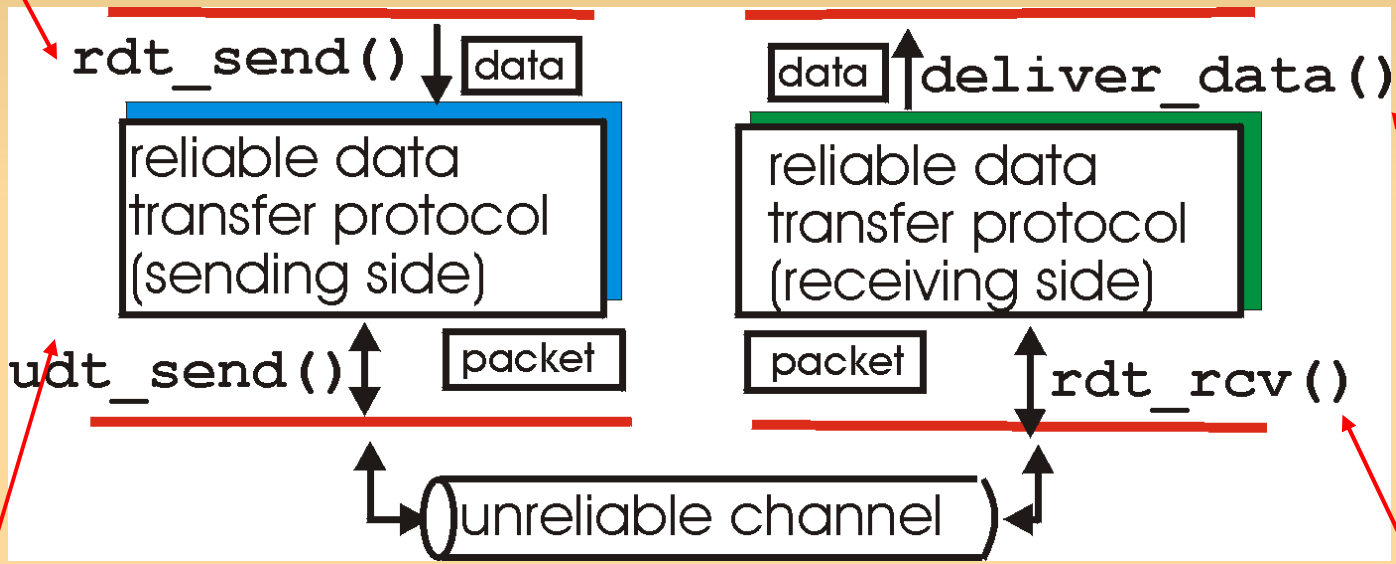
# Transferência confiável de dados

- “Um dos 10 maiores problemas em redes”.
- Características dos canais não confiáveis determinarão a complexidade dos protocolos confiáveis de transferência de dados (rdt – *reliable data transfer*).

# Transferência confiável: ponto de partida

`rdt_send()`: chamada da camada superior, (ex., pela aplicação). Passa dados para entregar à camada superior receptora

`deliver_data()`: chamada pela entidade de transporte para entregar dados para cima




lado transmissor

lado receptor

`udt_send()`: chamada pela entidade de transporte, para transferir pacotes para o receptor sobre o canal não confiável

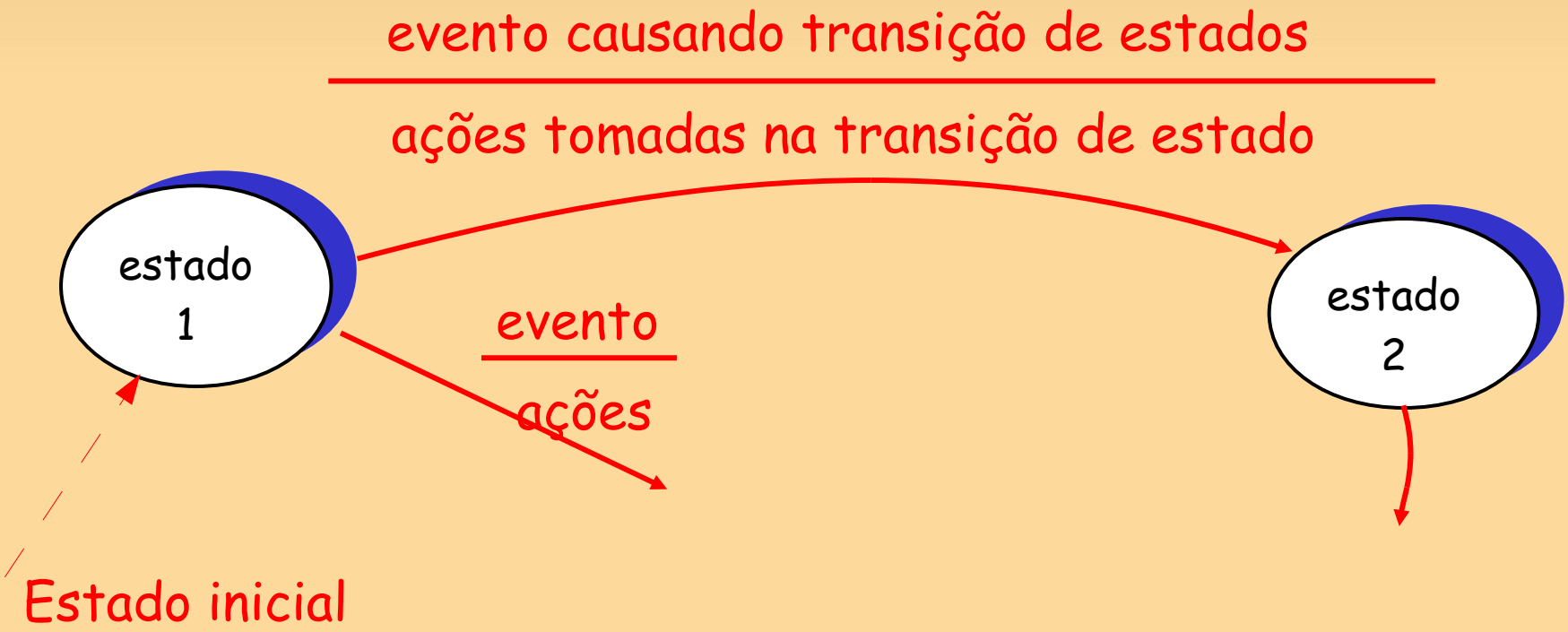
`rdt_rcv()`: chamada quando o pacote chega ao lado receptor do canal

# Etapas

- Desenvolver incrementalmente o transmissor e o receptor de um protocolo confiável de transferência de dados (rdt).
- Considerar apenas transferências de dados unidirecionais:
  - mas informação de controle deve fluir em ambas as direções!
- Usar máquinas de estados finitos (FSM) para especificar o protocolo transmissor e o receptor.

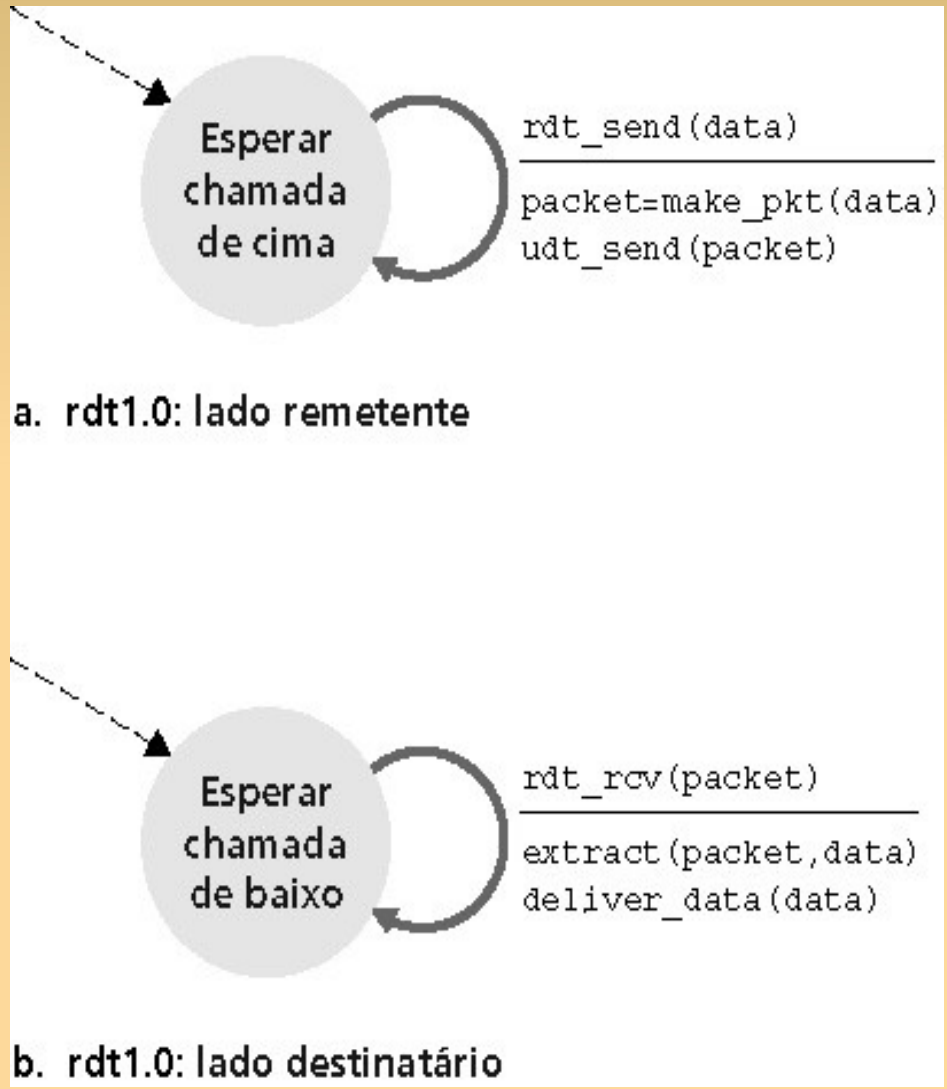
# Etapas: FSM

Estado: quando neste "estado" o próximo estado fica unicamente determinado pelo próximo evento.



# rdt 1.0

- Canal de transmissão perfeitamente confiável:
  - não há erros de bits;
  - não há perdas de pacotes;
  - não há necessidade de pacotes de controle.
- FSMs separadas para transmissor e receptor:
  - remetente envia dados para o canal subjacente;
  - destinatário lê os dados do canal subjacente.

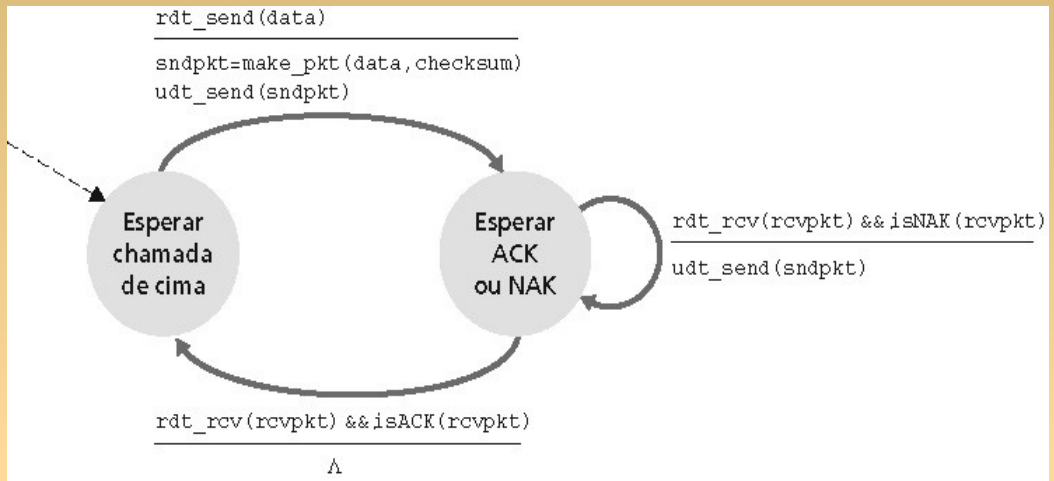


# rdt 2.0: canal com erro de bits

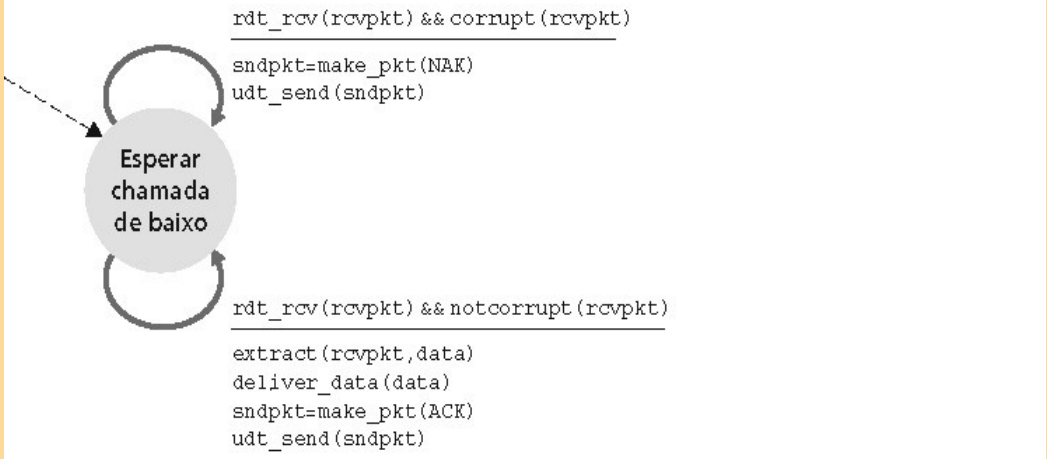
- Canal subjacente pode trocar valores dos bits num pacote!
  - *Checksum* para detectar erros de bits.
- Como recuperar esses erros?
  - **Reconhecimentos (ACKs):** destinatário avisa explicitamente ao remetente que o pacote foi recebido corretamente.
  - **Reconhecimentos negativos (NAKs):** destinatário avisa explicitamente ao remetente que o pacote tem erros, e este reenvia o pacote.

# rdt 2.0

- Novos mecanismos no **rdt2.0** (além do **rdt1.0**):
  - detecção de erros;
  - realimentação do destinatário: mensagens de controle:
    - (ACK, NAK) rcvr->sender
  - retransmissão.
- Protocolos ARQ: Automatic Repeat reQuest (solicitação automática de repetição).

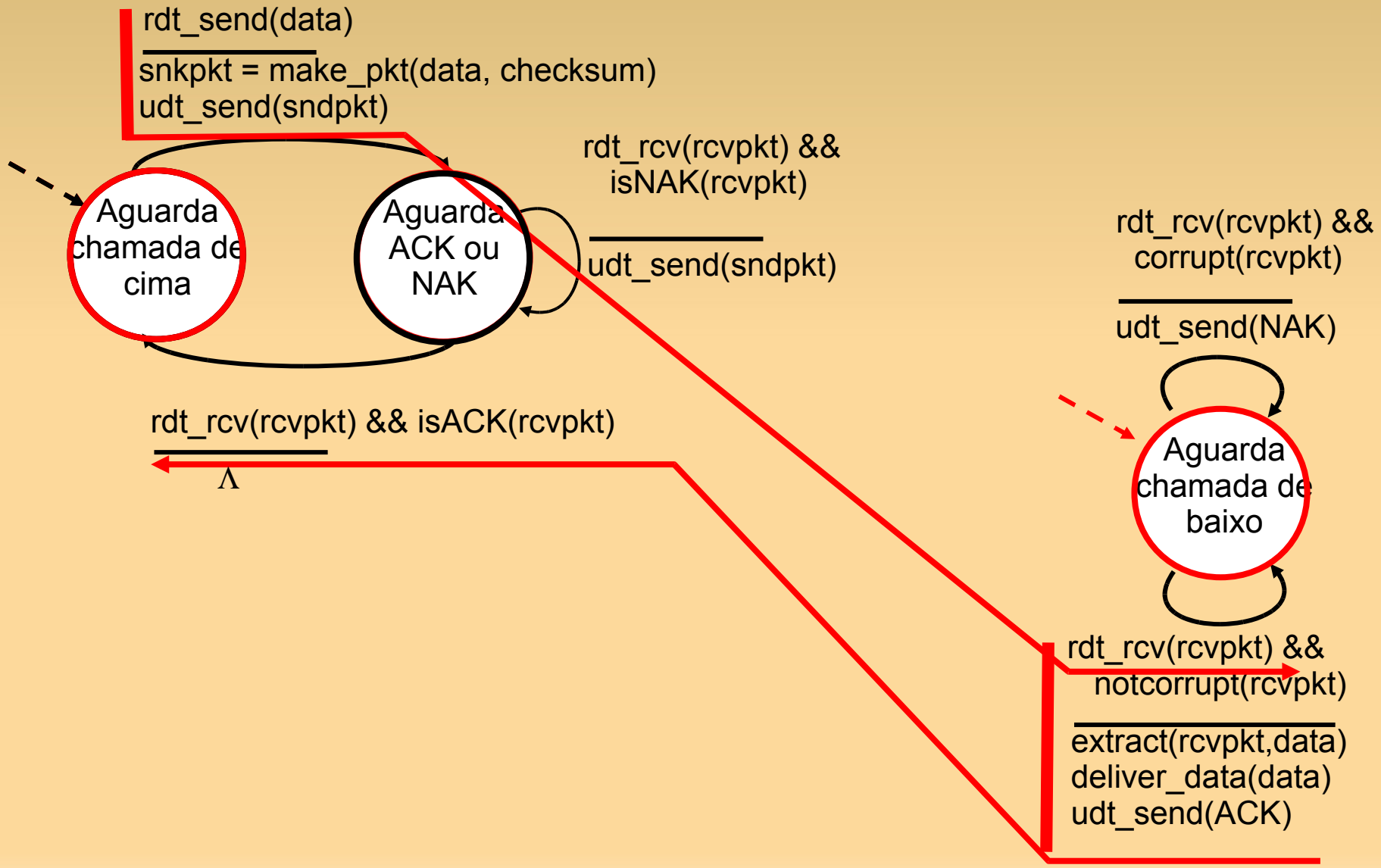



a. rdt2.0: lado remetente

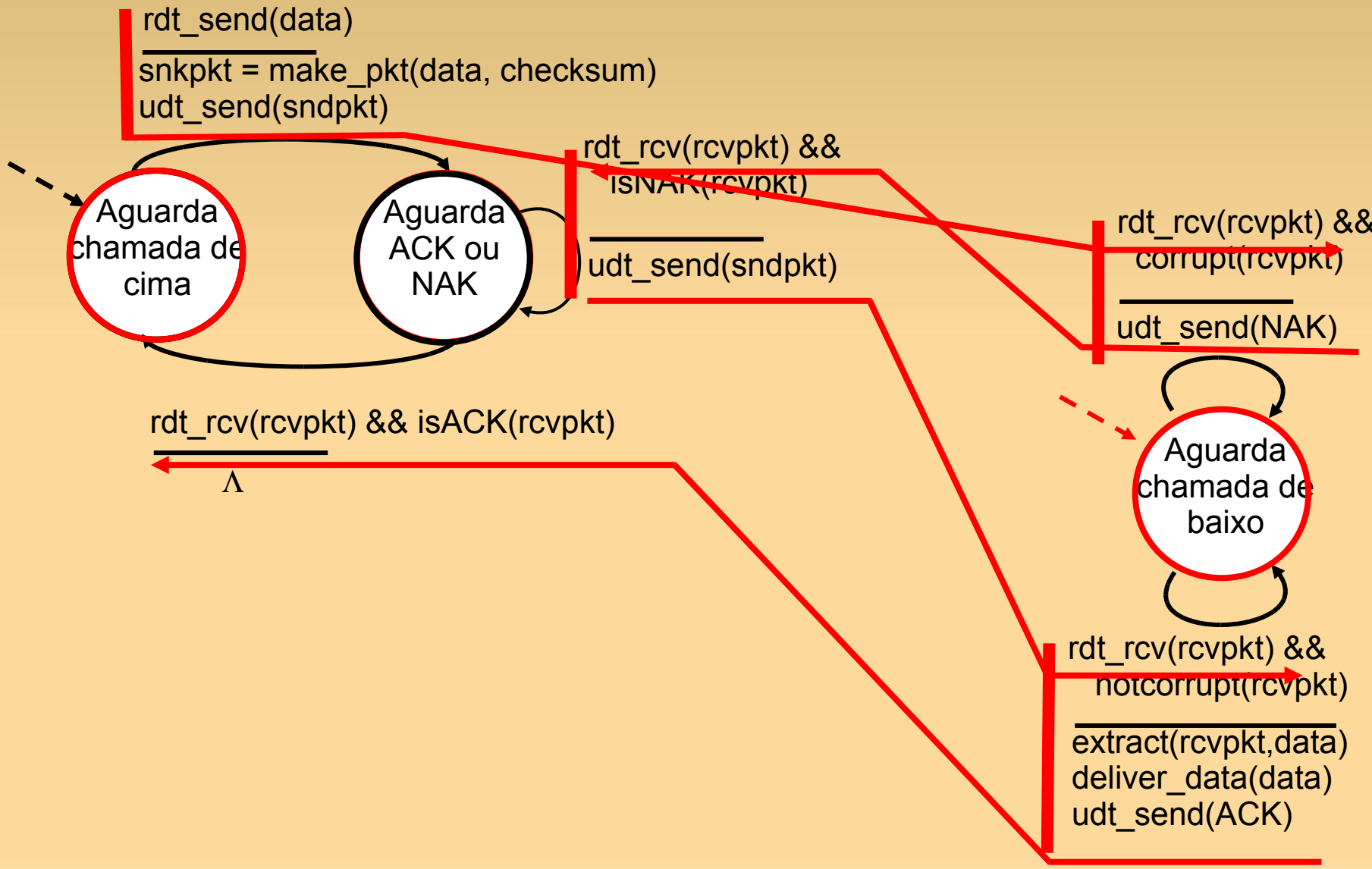



b. rdt2.0: lado destinatário

# rdt 2.0: operação sem ocorrência de erros



# rdt 2.0: operação com detecção de erros



# rdt 2.0: Problemas?

∀ O que acontece se o ACK/NAK é corrompido?

- Transmissor não sabe o que aconteceu no receptor!
- Não pode apenas retransmitir: possível duplicata!

# Tratando duplicatas

- Transmissor acrescenta número de sequência em cada pacote.
- Transmissor reenvia o último pacote se ACK/NAK for perdido.
- Receptor descarta (não passa para a aplicação) pacotes duplicados.

- Pare e espere (stop-and-go):
  - Transmissor envia um pacote e então espera pela resposta do receptor.

# rdt2.1: remetente, trata ACK/NAKs perdidos

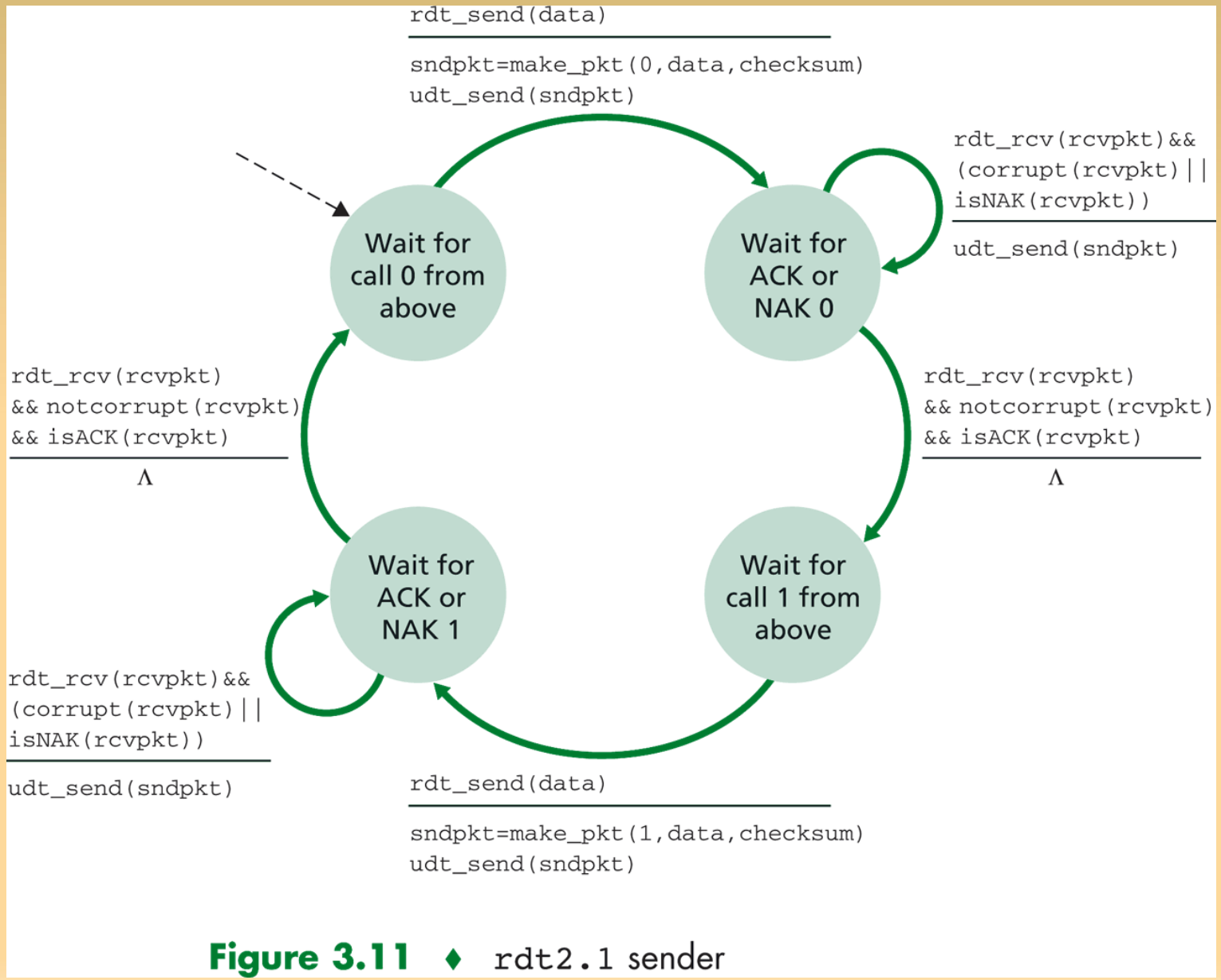



**Figure 3.11** ♦ rdt2.1 sender

# rdt2.1: destinatário, trata ACK/NAKs perdidos

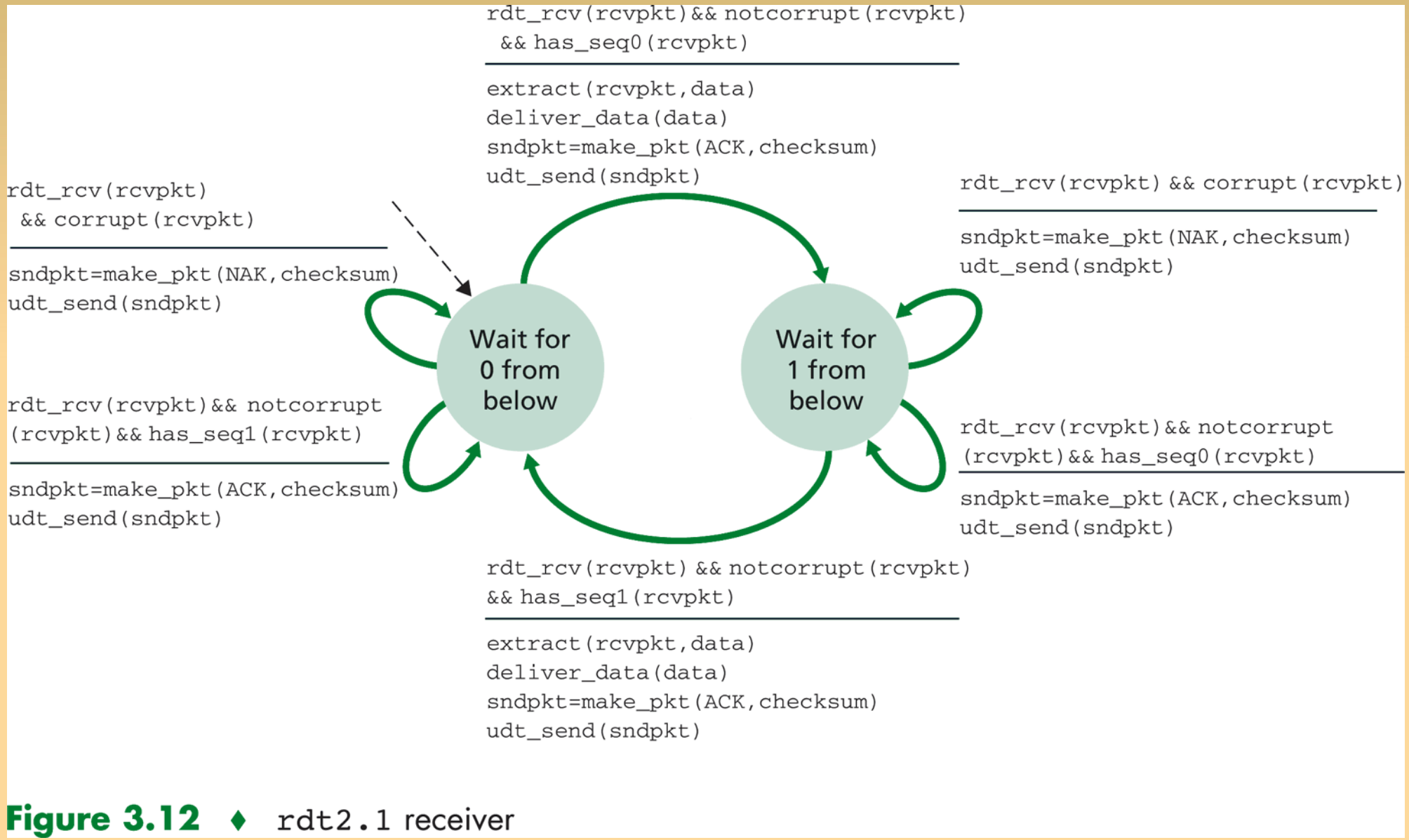



**Figure 3.12** ♦ rdt2.1 receiver

# rdt2.1: Discussão

**Transmissor:**

- Adiciona número de sequência ao pacote.
- Dois números (0 e 1) bastam. Por quê?
- Deve verificar se os ACK/NAK recebidos estão corrompidos.
- Duas vezes o número de estados.
  - O estado deve "lembrar" se o pacote "corrente" tem número de sequência 0 ou 1.

**Receptor:**

- Deve verificar se o pacote recebido é duplicado:
  - Estado indica se o pacote 0 ou 1 é esperado.
- Nota: receptor pode não saber se seu último ACK/NAK foi recebido pelo transmissor.

# rdt2.2: protocolo sem NACK

- Mesma funcionalidade do rdt2.1, usando somente ACKs.
- Em vez de enviar NAK, o receptor envia ACK para o último pacote recebido sem erro.
  - Receptor deve incluir explicitamente o número de sequência do pacote sendo reconhecido.
- ACKs duplicados no transmissor resultam na mesma ação do NAK: **retransmissão do pacote corrente.**

# rdt2.2: remetente

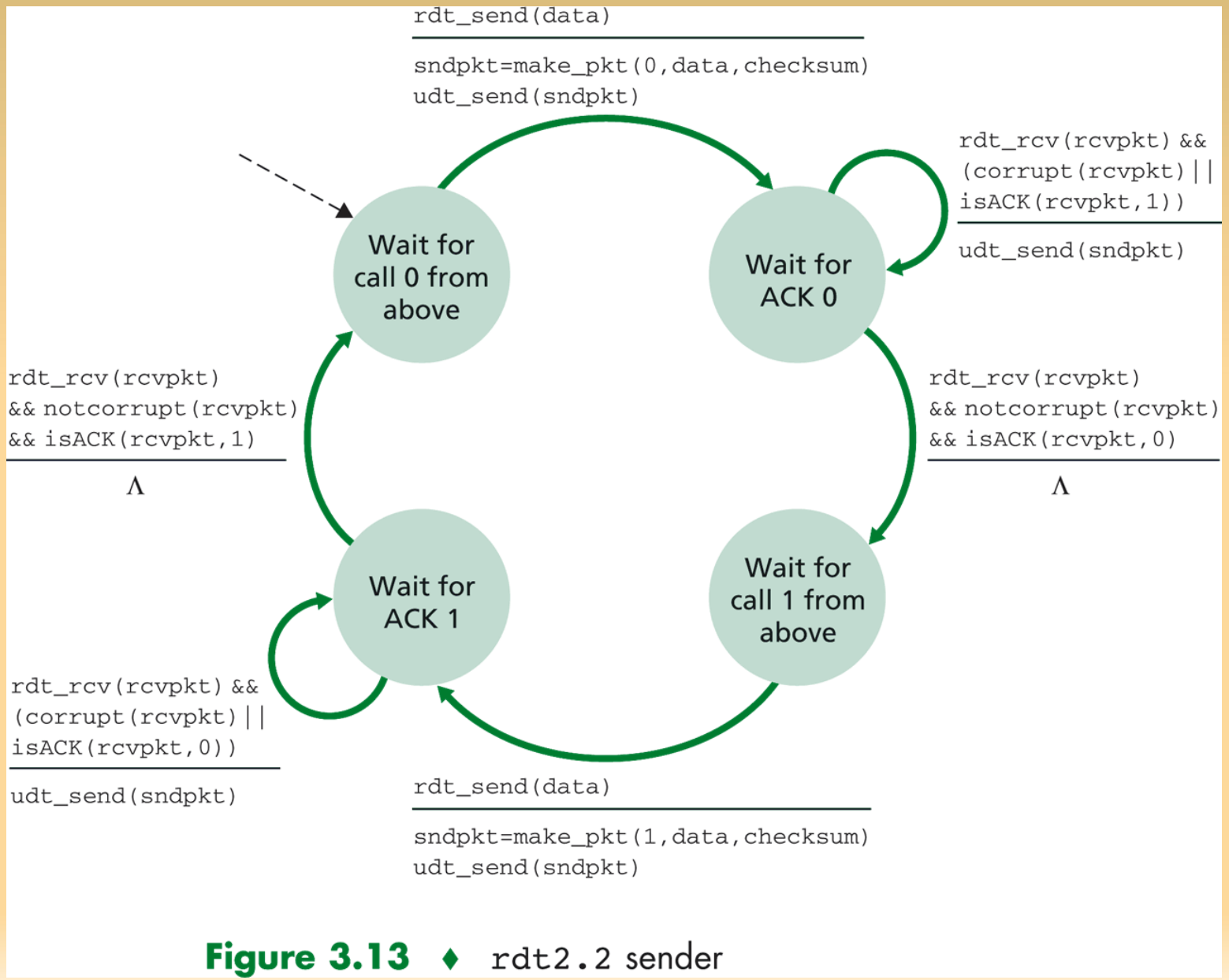



**Figure 3.13** ♦ rdt2.2 sender

# rdt2.2: destinatário

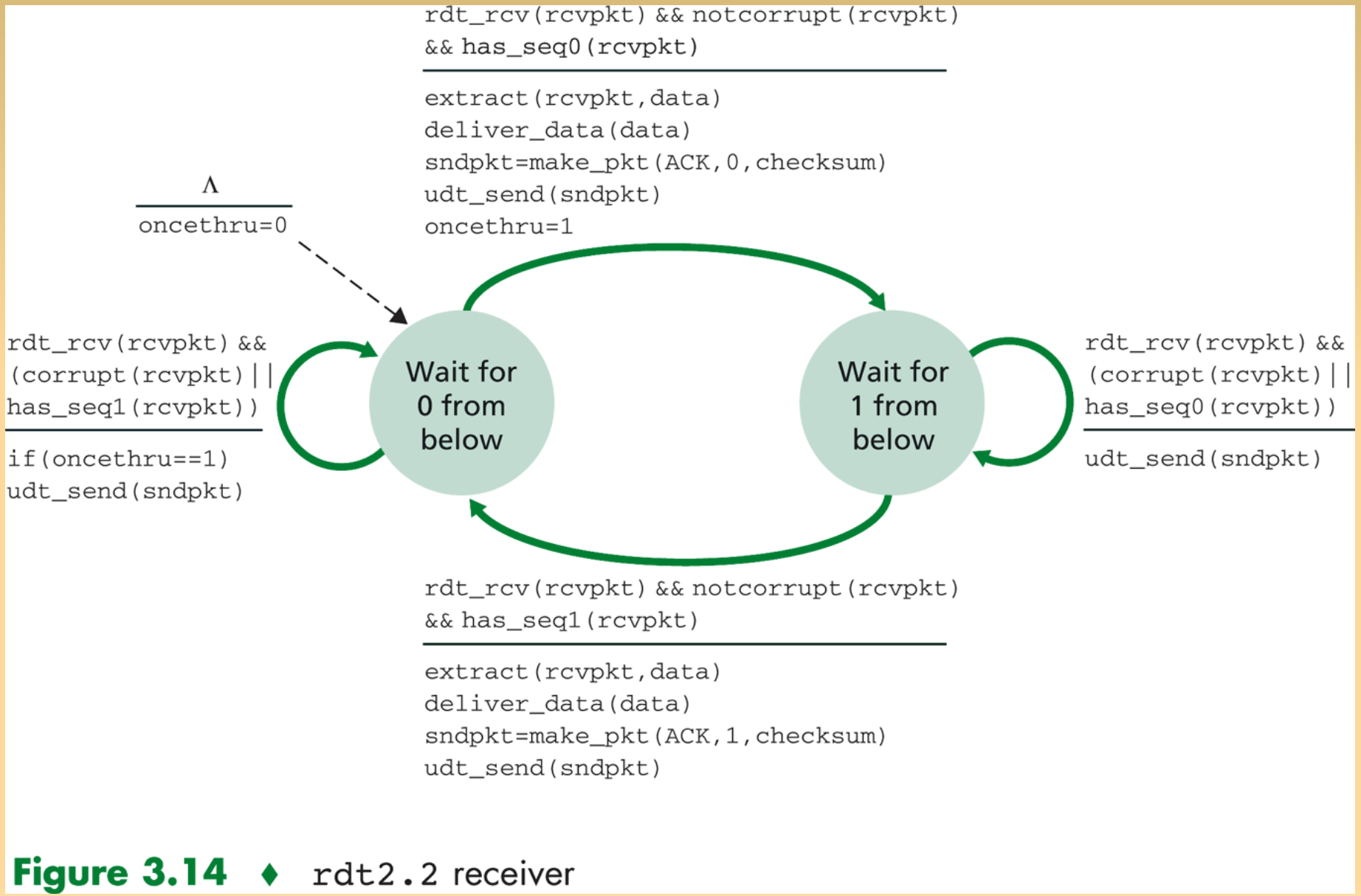



**Figure 3.14** ♦ rdt2.2 receiver

# rdt3.0: canal com erros e perdas

∀ Nova hipótese: canal de transmissão pode também perder pacotes!

∀ Checksum, números de sequência, ACKs, retransmissões serão de ajuda, mas não o bastante.

∀ Abordagem: transmissor espera um tempo "razoável" pelo ACK.

- Retransmite se nenhum ACK for recebido nesse tempo.

# rdt3.0: canal com erros e perdas (cont.)

- Se o pacote (ou ACK) estiver apenas atrasado (não perdido):
  - Retransmissão será duplicata, mas os números de sequência já tratam com isso.
- Receptor deve especificar o número de sequência do pacote sendo reconhecido.
- Exige um temporizador decrescente.

# rdt3.0: remetente

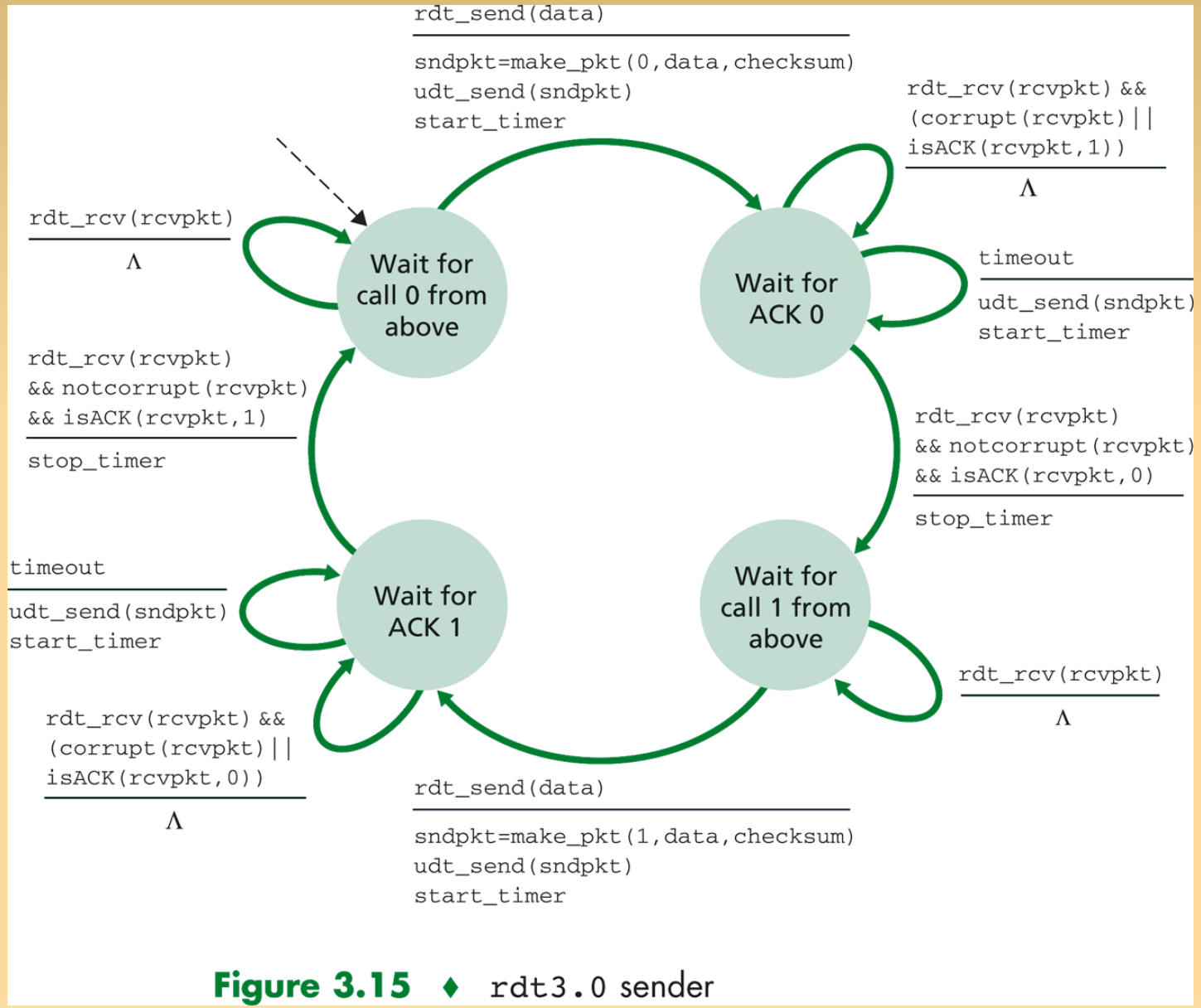
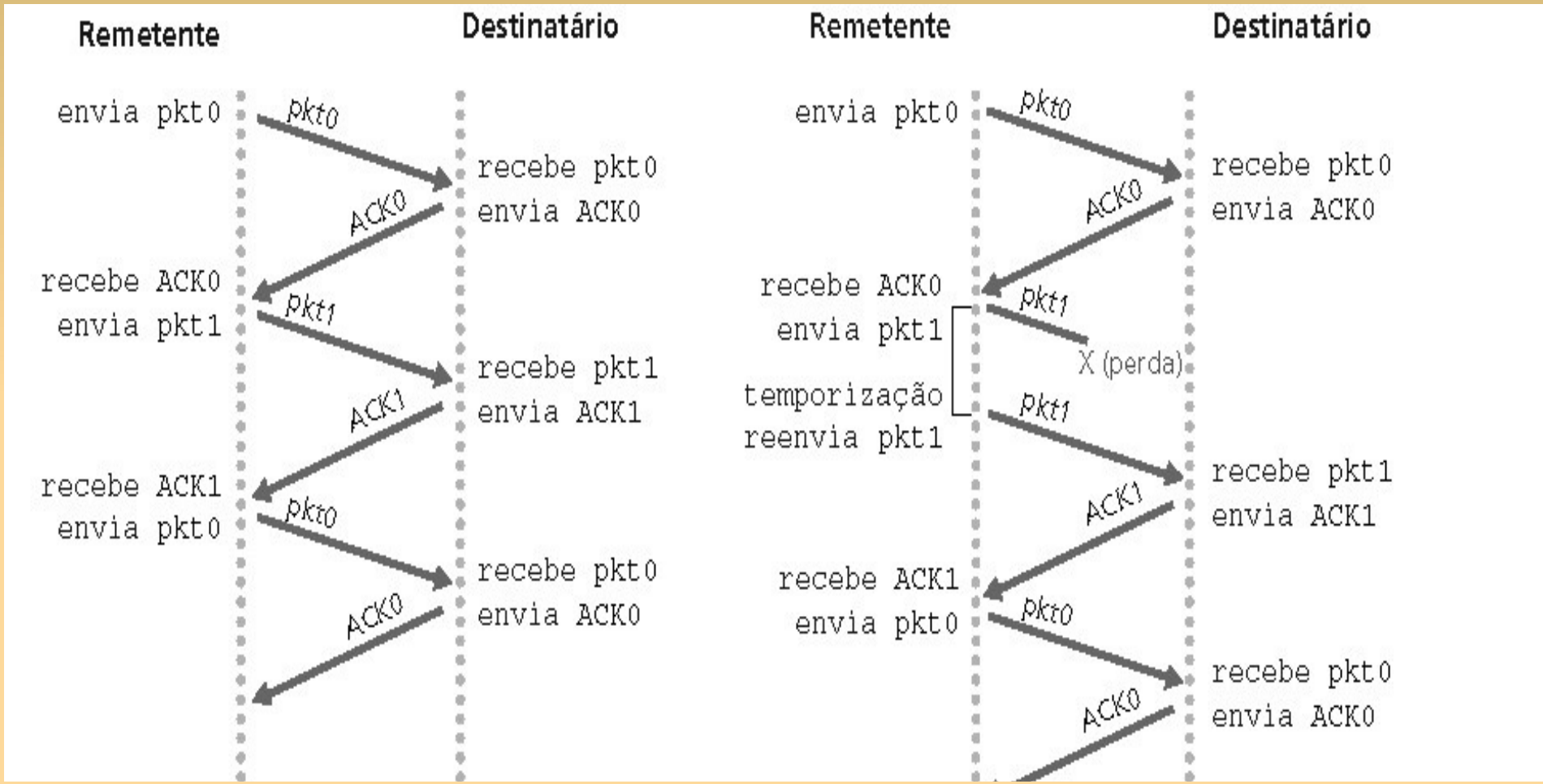
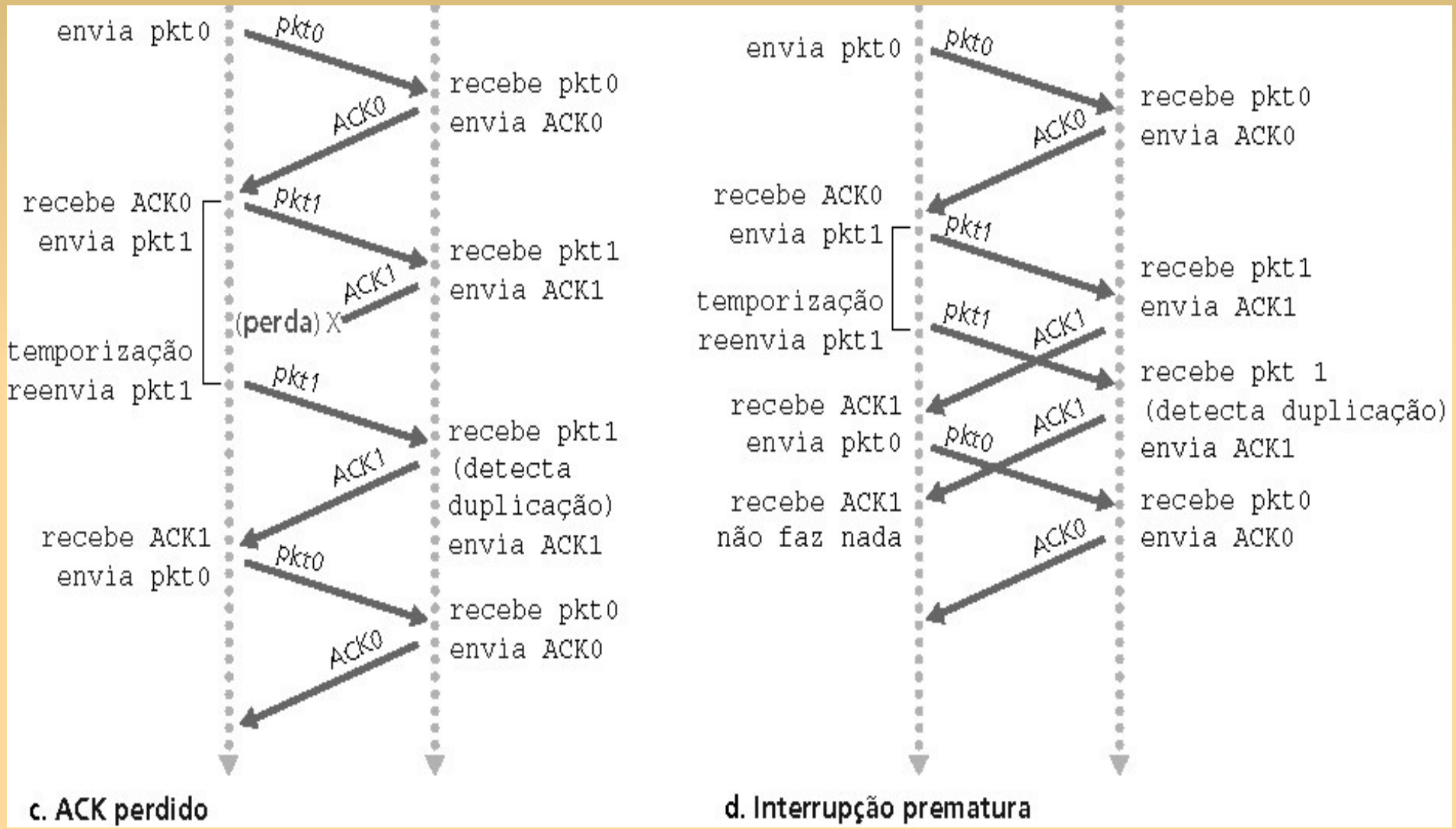



**Figure 3.15** ♦ rdt3.0 sender

# rdt3.0 - Operação



# rdt3.0 - Operação



# rdt3.0: desempenho

- ∀ rdt3.0 funciona, mas o desempenho é sofrível
- ∀ Exemplo: enlace de 1 Gbps, 15 ms de atraso de propagação, pacotes de 1 KB:

$$\text{Transmissão} = \frac{\text{L (tamanho do pacote em bits)}}{\text{R (taxa de transmissão, bps)}} = \frac{\text{8 kb/pkt}}{\text{10^9 b/s}} = \text{8 microsseg}$$

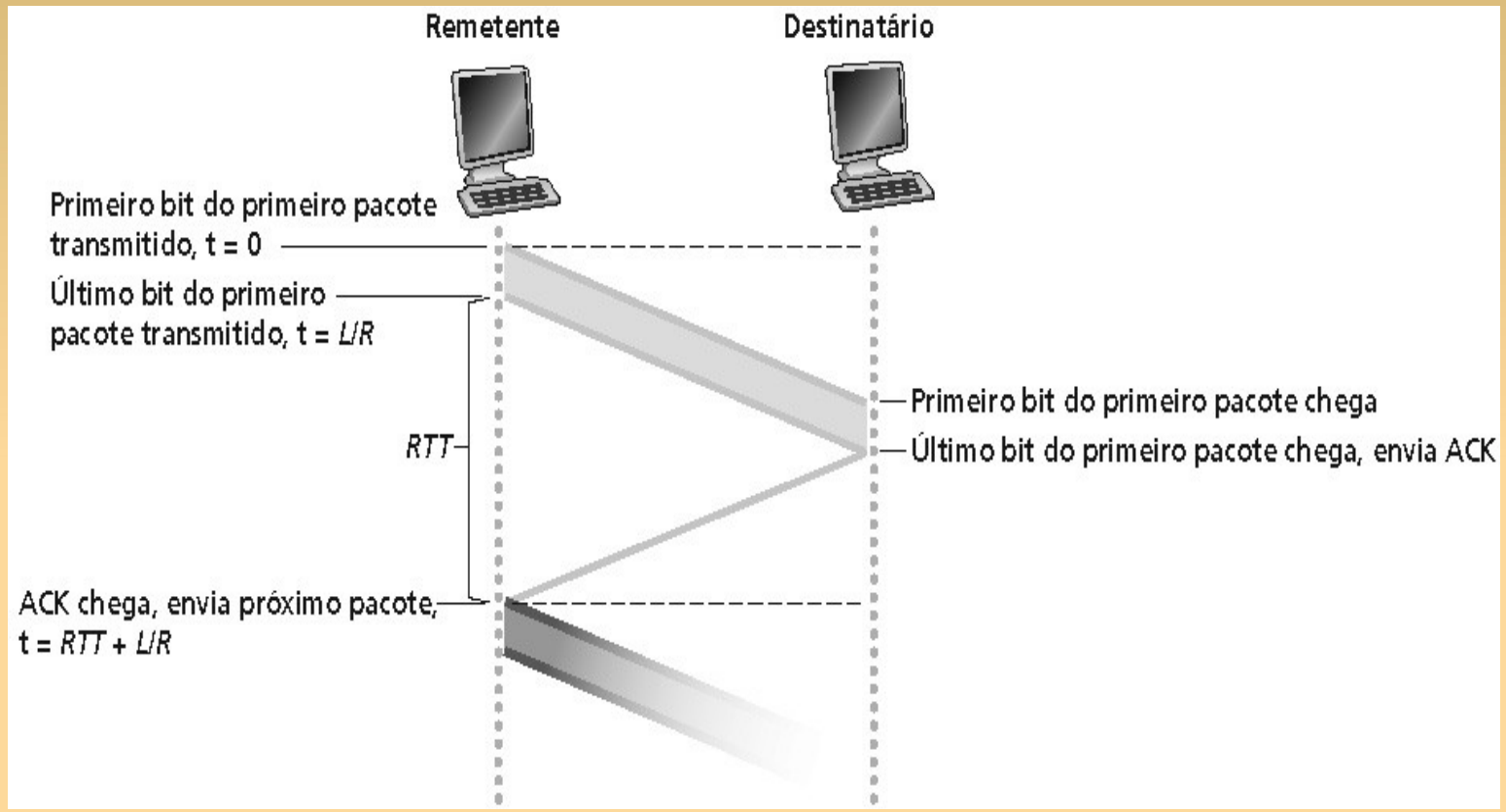
$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{0{,}008}{30{,}008} = 0{,}00027$$

- $U_{sender}$: utilização — fração de tempo do transmissor ocupado

Um pacote de 1 KB cada 30 ms -> 33 kB/s de vazão sobre um canal de 1 Gbps.

O protocolo de rede limita o uso dos recursos físicos!

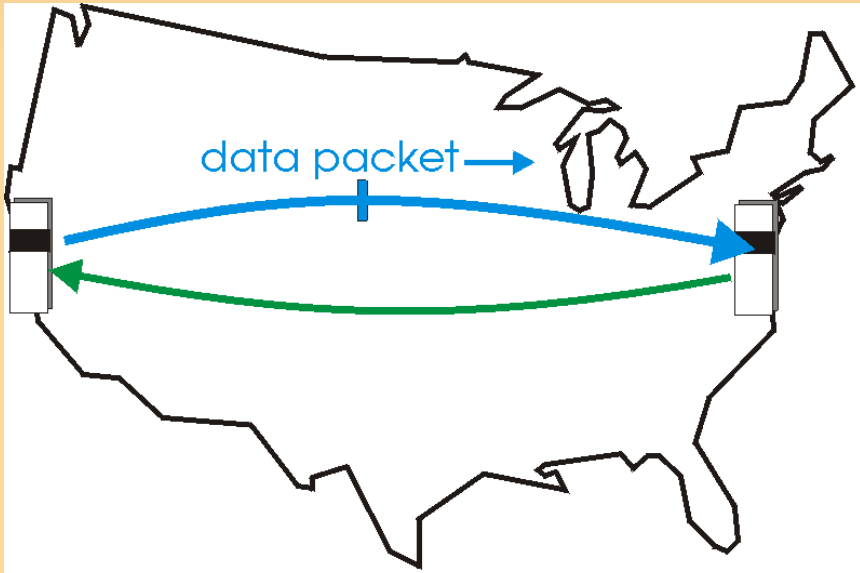
# rdt3.0 – “Pare e Espere”



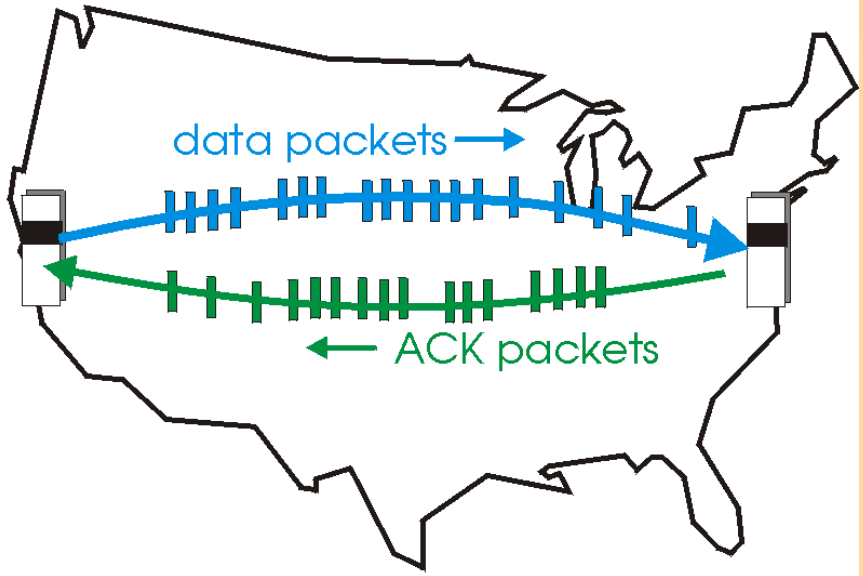

$$\text{sender} = \frac{L/R}{RTT + L/R} = \frac{0{,}008}{30{,}008} = 0{,}00027$$

# Protocolos com paralelismo

- Paralelismo: transmissor envia vários pacotes ao mesmo tempo, todos esperando para serem reconhecidos.

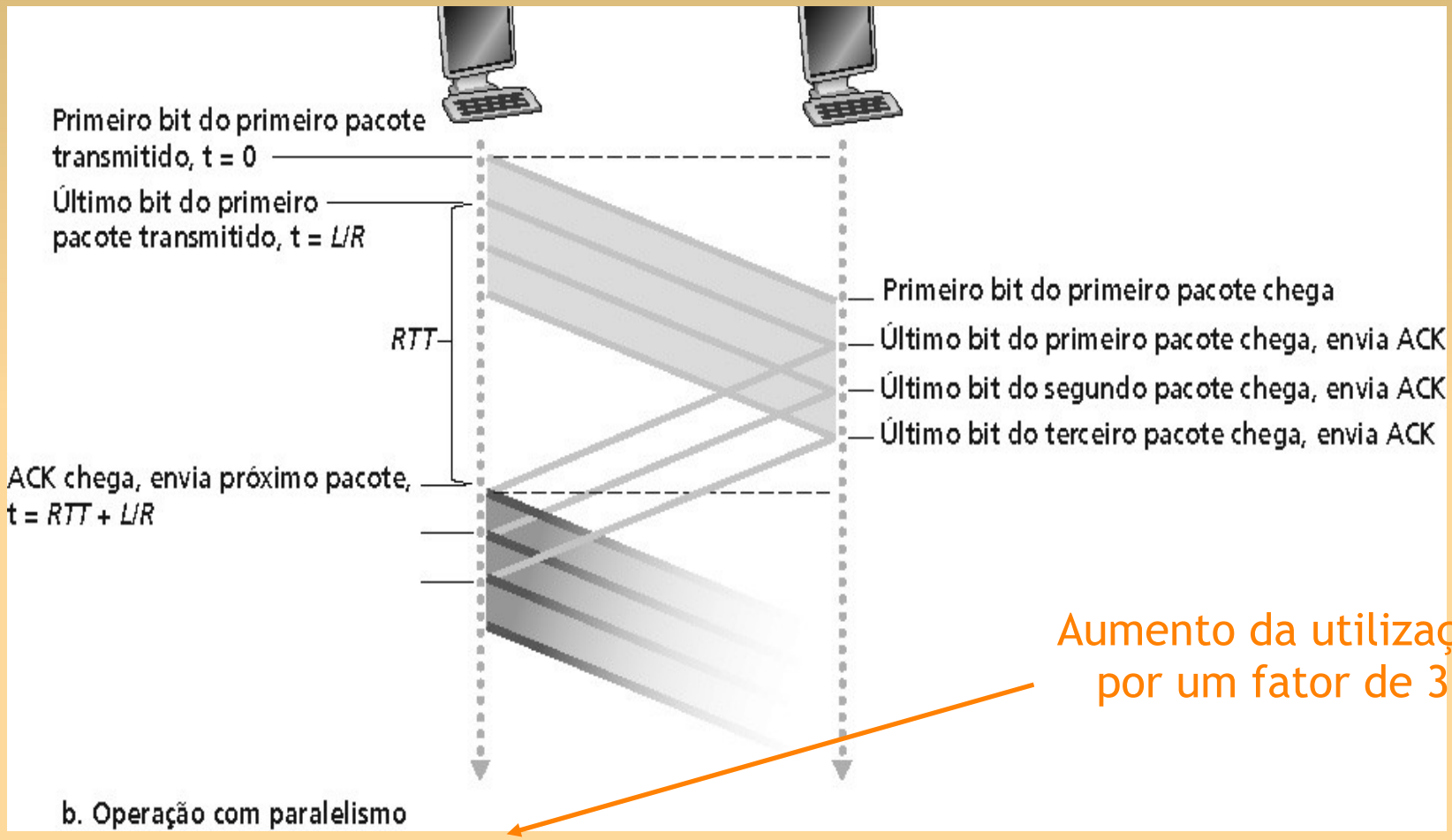



(a) operação do protocolo pare e espere




(a) operação do protocolo com paralelismo

# Paralelismo: aumento da utilização




Aumento da utilização por um fator de 3!

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

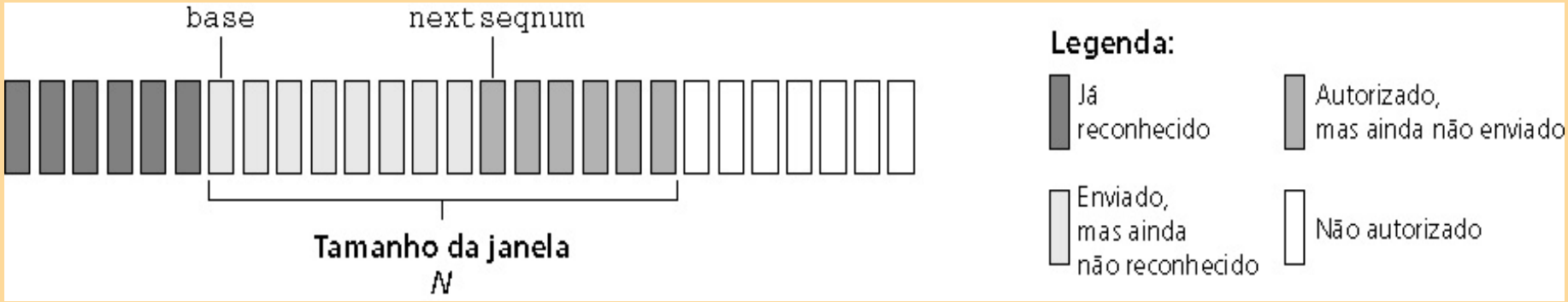
# Protocolos com paralelismo

- Alterações necessárias:
  - faixa de números de seqüência deve ser aumentada;
  - armazenamento no transmissor e/ou no receptor.
- Duas formas genéricas de protocolos com paralelismo:
  - go-Back-N;
  - retransmissão seletiva.

# Go-back-N (GBN)

**Remetente:**

- ∀ número de seqüência com k bits no cabeçalho do pacote;
- ∀ "janela" de até N pacotes não reconhecidos, consecutivos, são permitidos.



# GBN: Remetente

- ACK(n): reconhece todos os pacotes até o número de seqüência N (incluindo este limite) = “ACK cumulativo”.
  - pode receber ACKs duplicados (veja receptor).
- Temporizador para cada pacote enviado e não confirmado.
- **Tempo de confirmação** (**n**): retransmite pacote n e todos os pacotes com número de seqüência maior que estejam dentro da janela.

# GBN: FSM do remetente

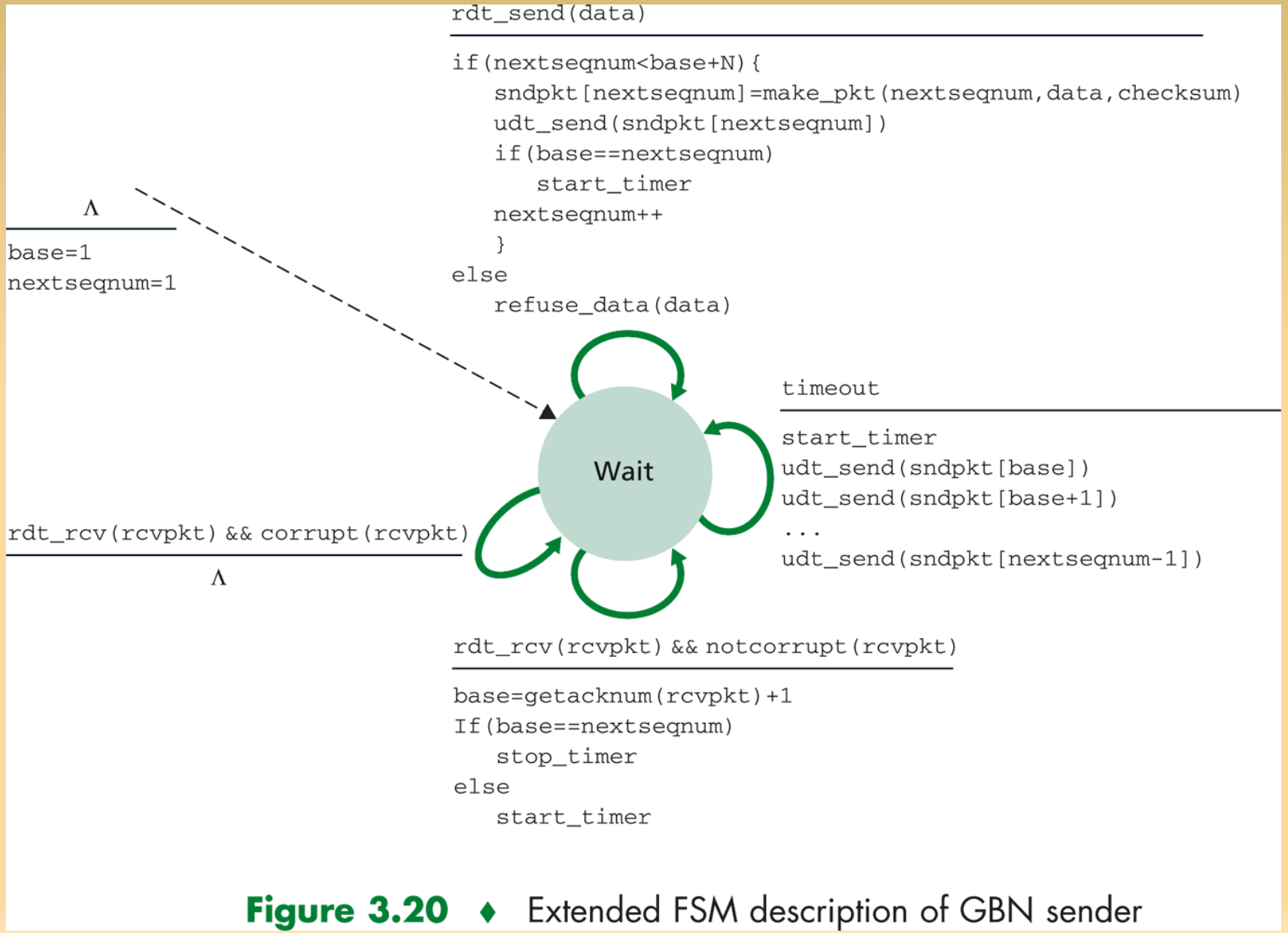



**Figure 3.20** ♦ Extended FSM description of GBN sender

# GBN: destinatário

- Somente ACK: sempre envia ACK para pacotes corretamente recebidos com o mais alto número de seqüência em ordem
  - pode gerar ACKs duplicados;
  - precisa lembrar apenas do **expectedseqnum;**
- Pacotes fora de ordem:
  - descarta (não armazena) -> não há buffer de recepção!
  - reconhece pacote com o mais alto número de seqüência em ordem.

# GBN: destinatário

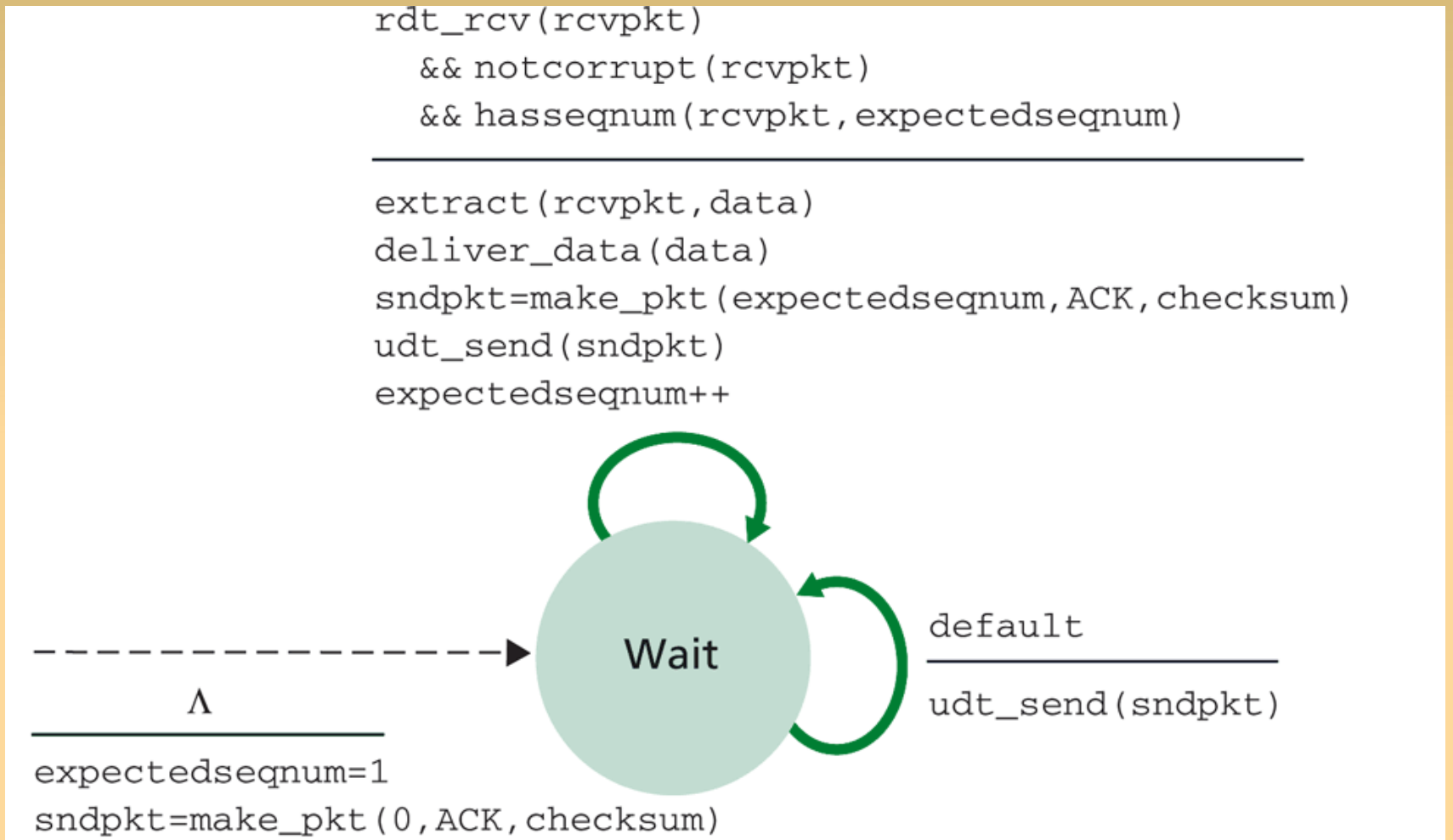



**Figure 3.21** ♦ Extended FSM description of GBN receiver

# Operação GBN

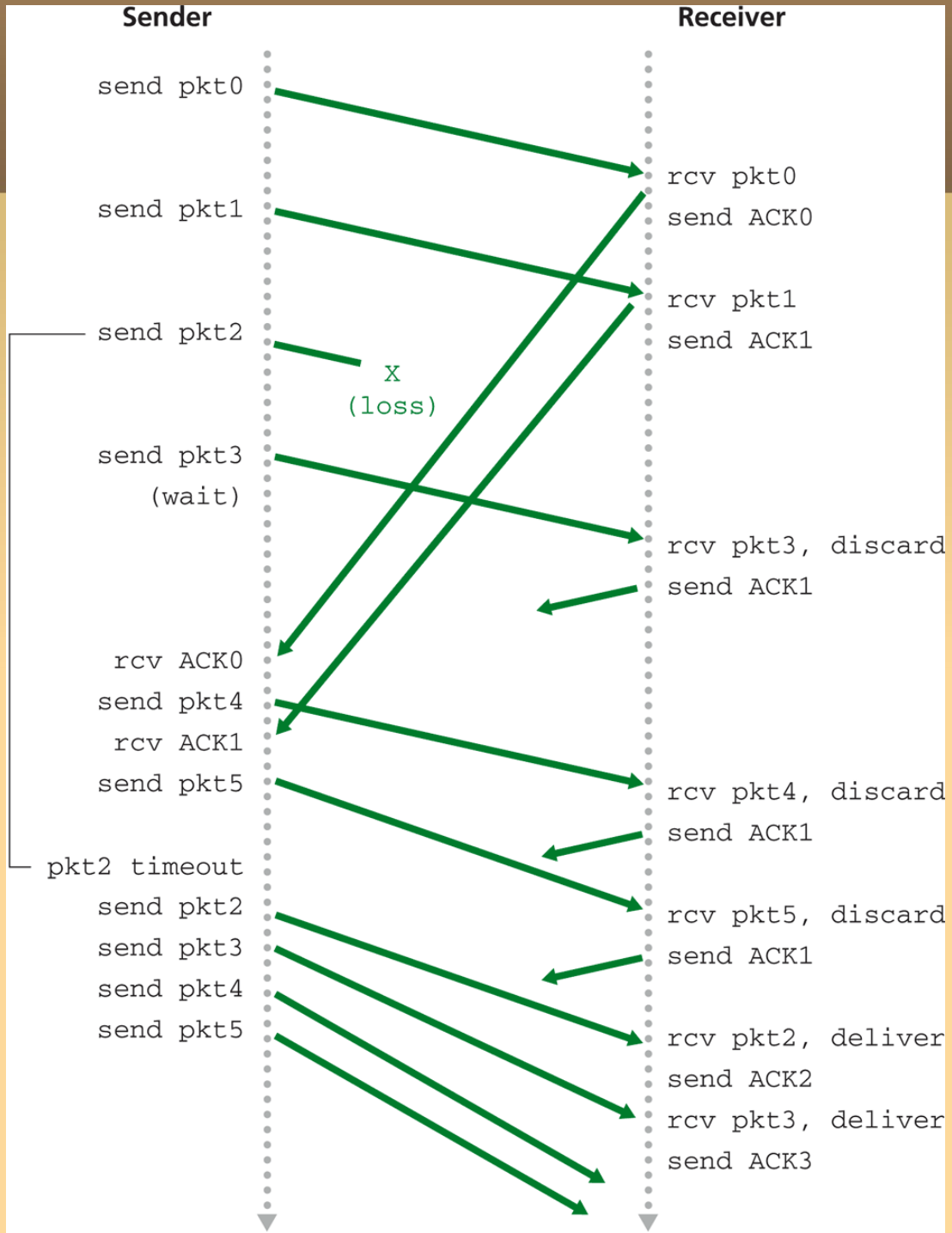



**Figure 3.22** ♦ Go-Back-N in operation

# GBN: applet

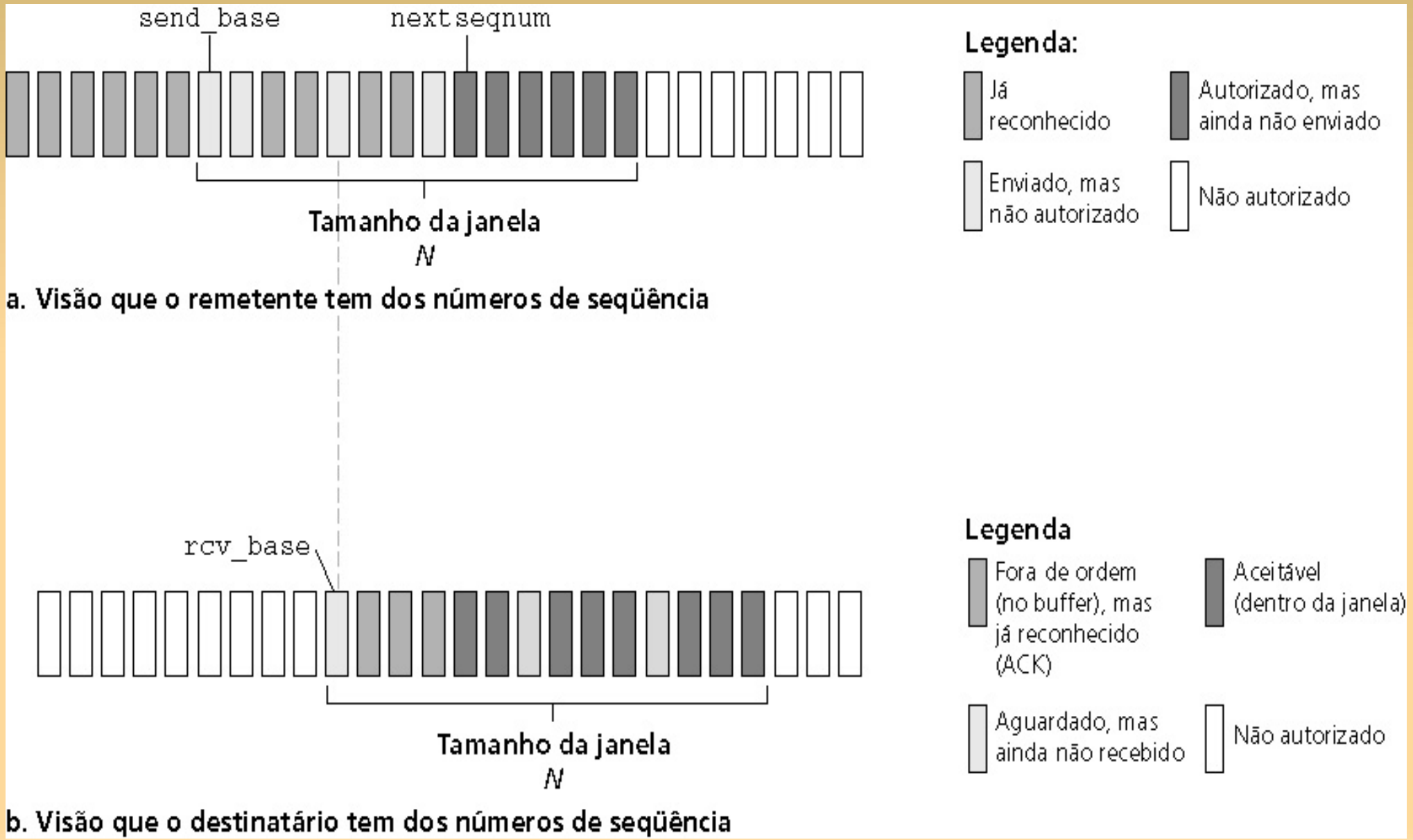
- Link:

http://media.pearsoncmg.com/aw/aw_kurose_network_2/applets/go-back-n/go-back-n.html

# Retransmissão Seletiva

- Receptor reconhece **individualmente** pacotes recebidos corretamente.
- Armazena pacotes, se necessário, para eventual entrega em ordem.
- Transmissor somente reenvia os pacotes para os quais um ACK não foi recebido.
  - temporiza cada pacote não reconhecido.
- Janela de transmissão:
  - N números de seqüência consecutivos;
  - Novamente limita a quantidade de pacotes enviados, mas não reconhecidos.

# RS: janelas


send_base
next seqnum
Tamanho da janela
N
a. Visão que o remetente tem dos números de seqüência
Legenda:
Já reconhecido
Autorizado, mas ainda não enviado
Enviado, mas não autorizado
Não autorizado
rcv_base
Tamanho da janela
N
b. Visão que o destinatário tem dos números de seqüência
Legenda
Fora de ordem (no buffer), mas já reconhecido (ACK)
Aceitável (dentro da janela)
Aguardado, mas ainda não recebido
Não autorizado

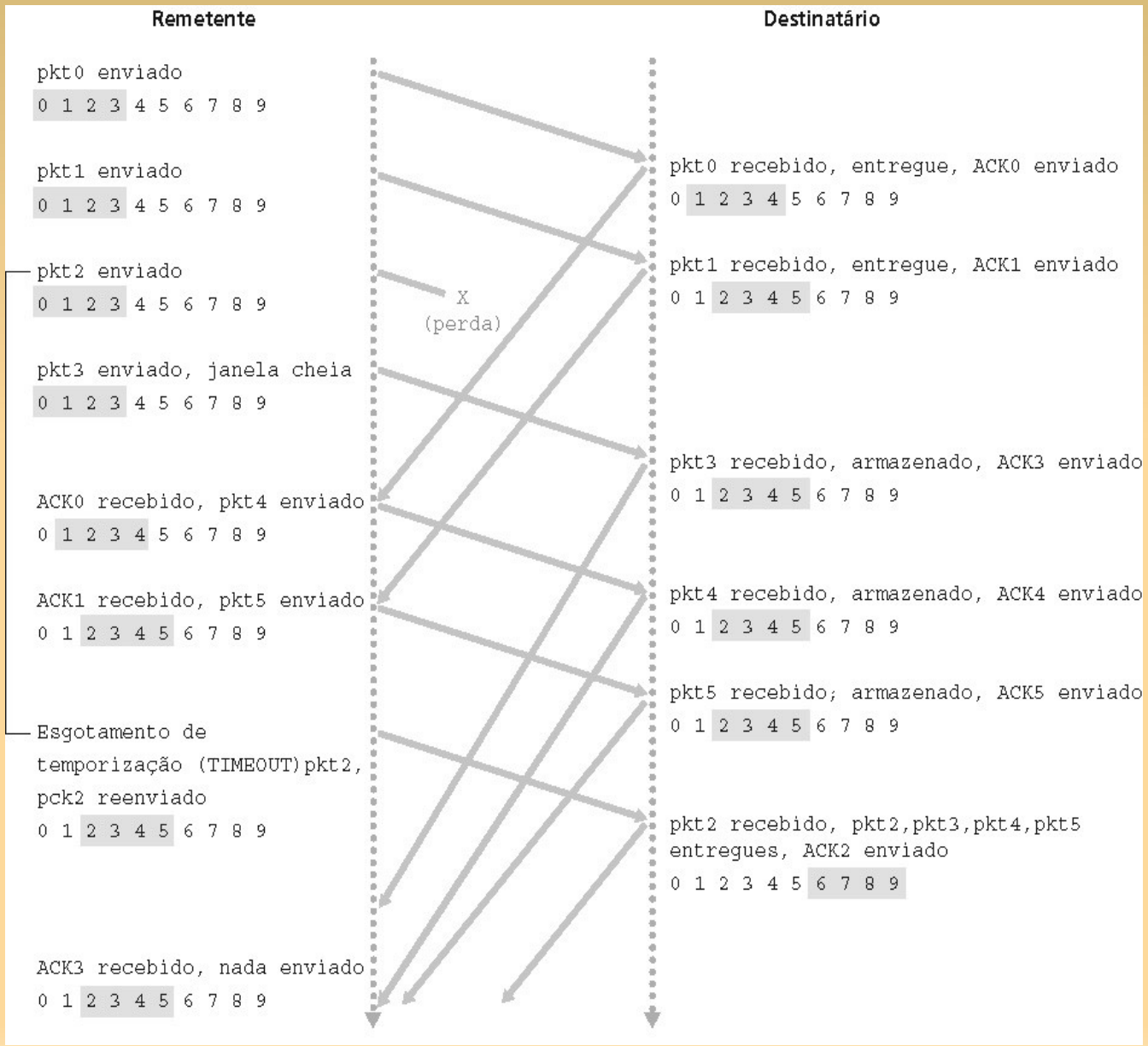
# RS: remetente

- Dados da camada superior:
  - Se o próximo número de seqüência disponível está na janela, envia o pacote.
- Tempo de confirmação(n):
  - reenvia pacote n, reinicia temporizador.
- ACK (n) em [sendbase,sendbase+N]:
  - marca pacote n como recebido
  - se n é o menor pacote não reconhecido, avança a base da janela para o próximo número de seqüência não reconhecido.

# RS: destinatário

- Pacote n em [rcvbase, rcvbase + N -1]:
  - envia ACK(n);
  - fora de ordem: armazena;
  - em ordem: entrega (também entrega pacotes armazenados em ordem), avança janela para o próximo pacote ainda não recebido.
- pkt n em [rcvbase-N,rcvbase-1]:
  - pacote encontra-se na janela anterior
  - ACK(n)
- Caso contrário:
  - Ignora.

# RS: operação


Remetente
Destinatário
pkt0 enviado
0 1 2 3 4 5 6 7 8 9
pkt1 enviado
0 1 2 3 4 5 6 7 8 9
pkt2 enviado
0 1 2 3 4 5 6 7 8 9
X
(perda)
pkt3 enviado, janela cheia
0 1 2 3 4 5 6 7 8 9
ACK0 recebido, pkt4 enviado
0 1 2 3 4 5 6 7 8 9
ACK1 recebido, pkt5 enviado
0 1 2 3 4 5 6 7 8 9
Esgotamento de temporização (TIMEOUT)pkt2, pck2 reenviado
0 1 2 3 4 5 6 7 8 9
ACK3 recebido, nada enviado
0 1 2 3 4 5 6 7 8 9
pkt0 recebido, entregue, ACK0 enviado
0 1 2 3 4 5 6 7 8 9
pkt1 recebido, entregue, ACK1 enviado
0 1 2 3 4 5 6 7 8 9
pkt3 recebido, armazenado, ACK3 enviado
0 1 2 3 4 5 6 7 8 9
pkt4 recebido, armazenado, ACK4 enviado
0 1 2 3 4 5 6 7 8 9
pkt5 recebido; armazenado, ACK5 enviado
0 1 2 3 4 5 6 7 8 9
pkt2 recebido, pkt2,pkt3,pkt4,pkt5 entregues, ACK2 enviado
0 1 2 3 4 5 6 7 8 9

# RS: dilema

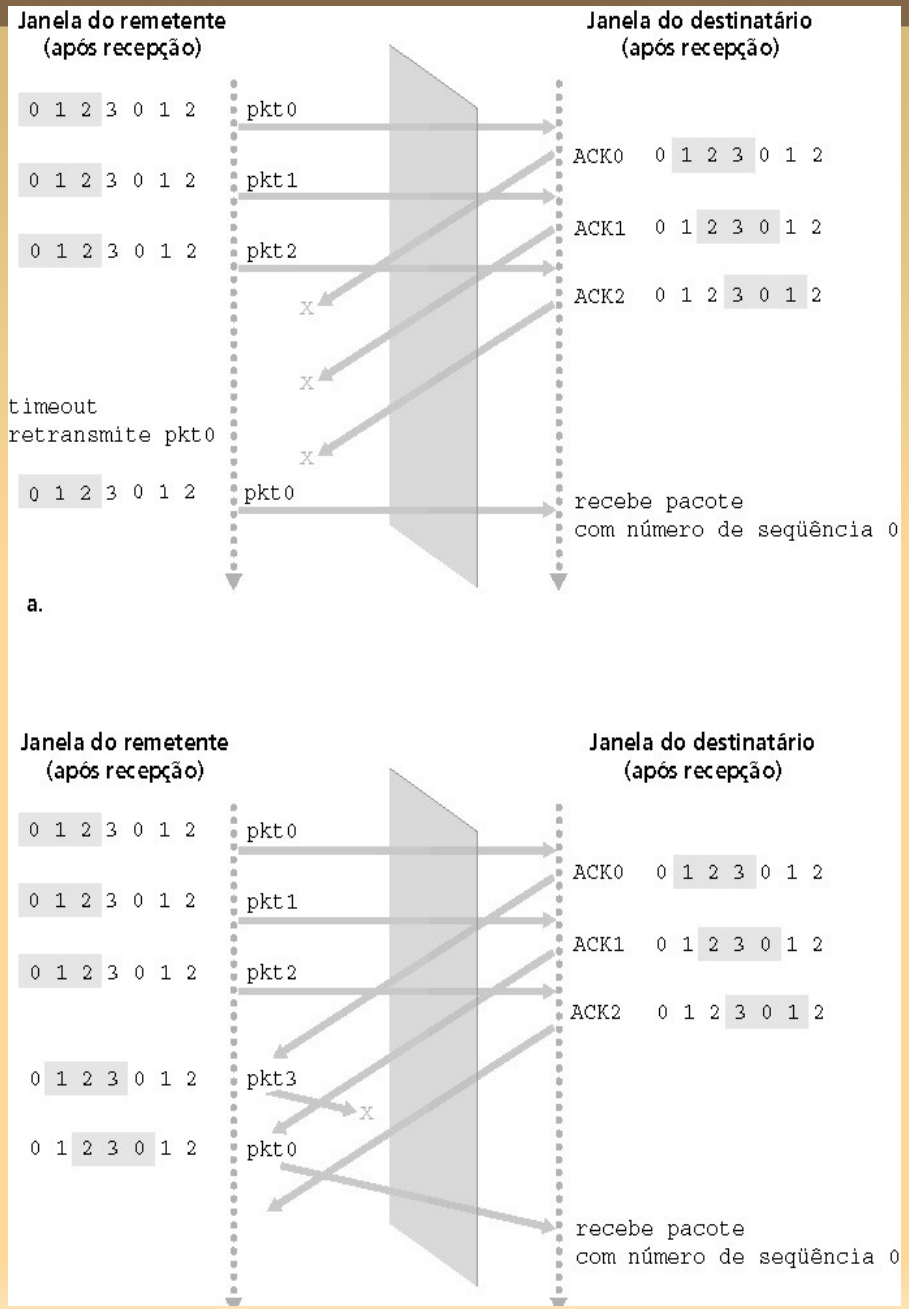
Exemplo:

- Seqüências: 0, 1, 2, 3
- Tamanho da janela = 3
- Receptor não vê diferença nos dois cenários!
- Incorretamente passa dados duplicados como novos (figura a)

Tamanho da janela = espaço de numeração sequencial – 1

(não vai funcionar)

P.: Qual a relação entre o espaço de numeração seqüencial e o tamanho da janela?



# Perguntas??