Capítulo 3: Camada de Transporte

Metas do capítulo:

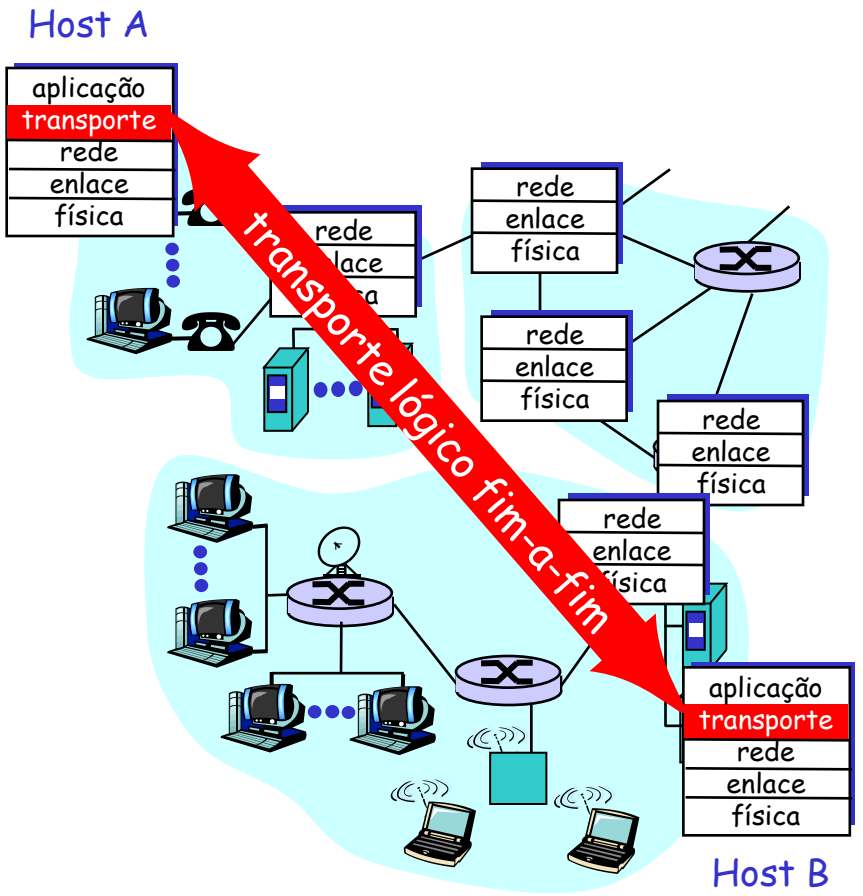
- Apresentar os serviços da camada de transporte:
  - multiplexação/ demultiplexação
  - transferência confiável de dados
  - controle de fluxo
  - controle de congestionamento
- Descrever a implementação dos serviços na Internet

Sumário do capítulo:

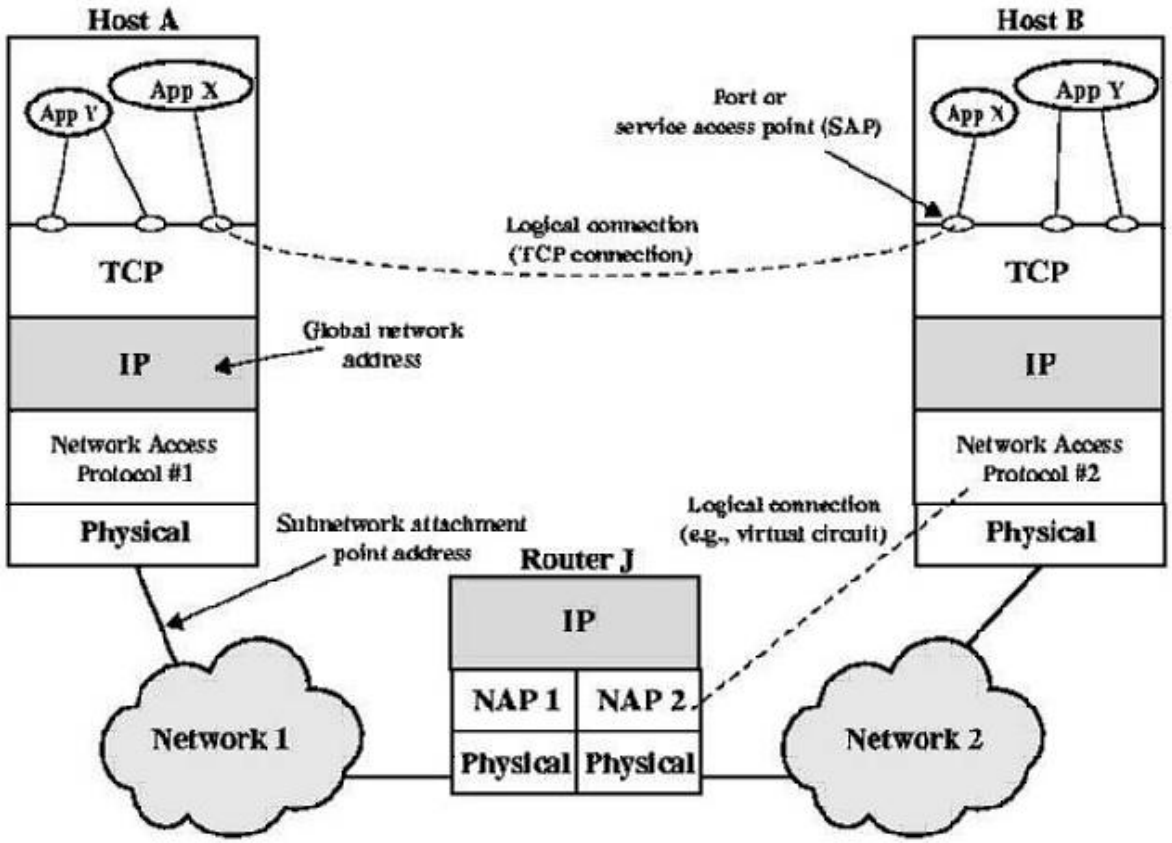
- Serviços da camada de transporte
- Multiplexação/demultiplexação
- Transporte sem conexão: UDP
- Princípios de transferência confiável de dados
- Transporte orientado à conexão: TCP
  - transferência confiável
  - controle de fluxo
  - gerenciamento de conexões
- Princípios de controle de congestionamento
- Controle de congestionamento em TCP

Serviços de Camada de Transporte

- Protocolos de transporte executam em sistemas terminais (*hosts*)
- Fornecem **comunicação lógica** entre processos de aplicação executando em *hosts* diferentes
- **Camada de transporte vs. camada de rede:**
  - **Camada de rede**: dados são transferidos entre *hosts*
  - **Camada de transporte**: dados são transferidos entre processos de aplicação
    - depende de serviços da camada de rede

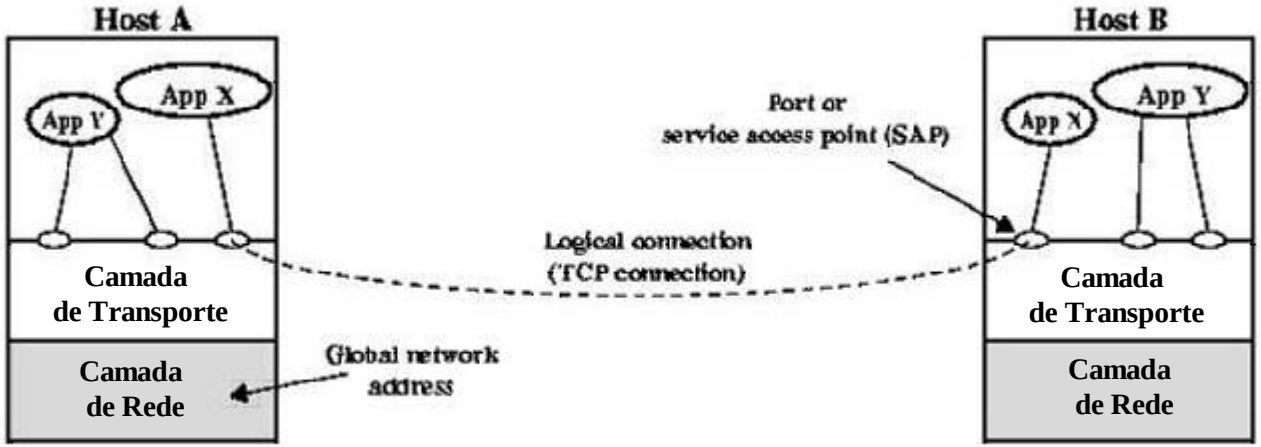


Serviços de Camada de Transporte - exemplo

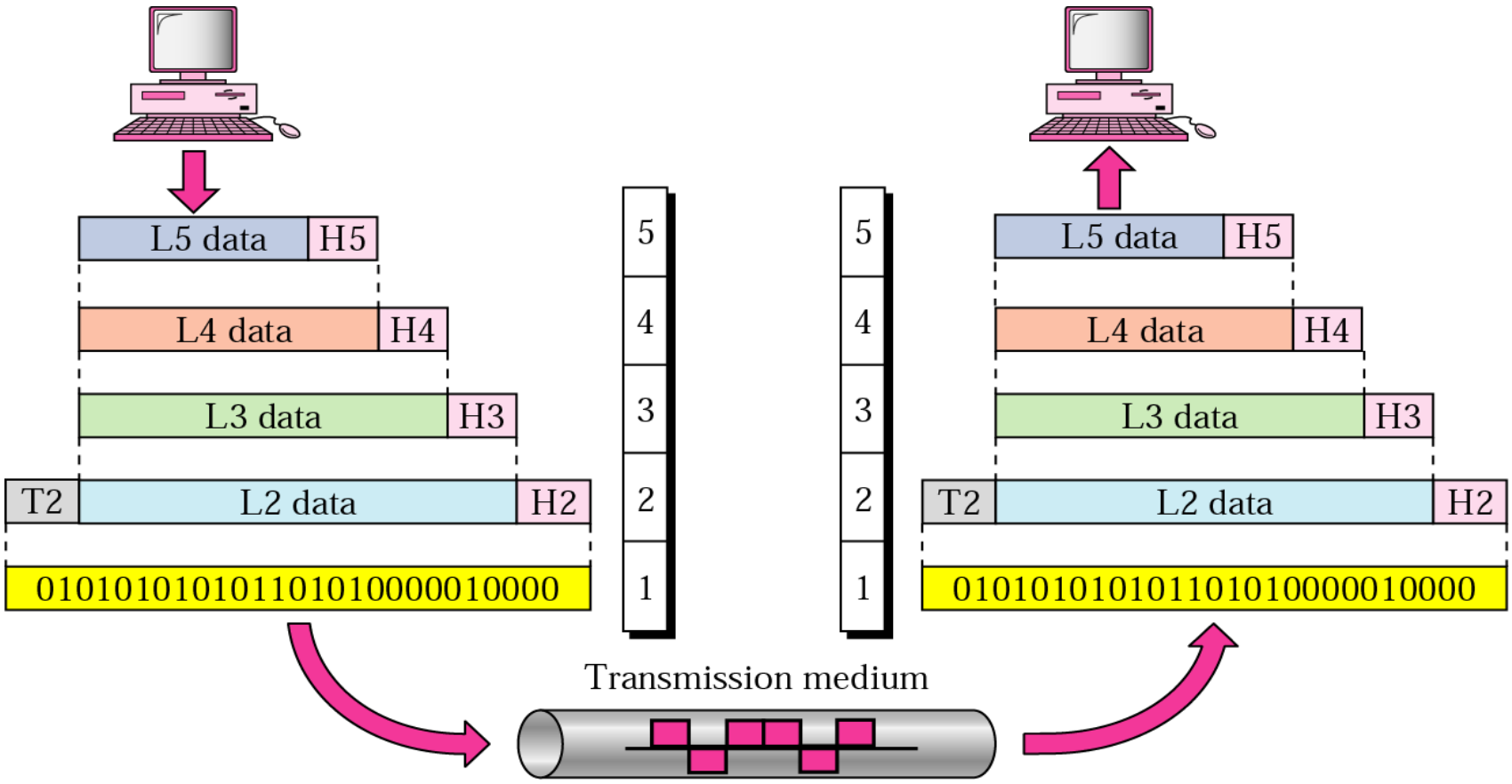


Camada de Transporte vs. Camada de Rede

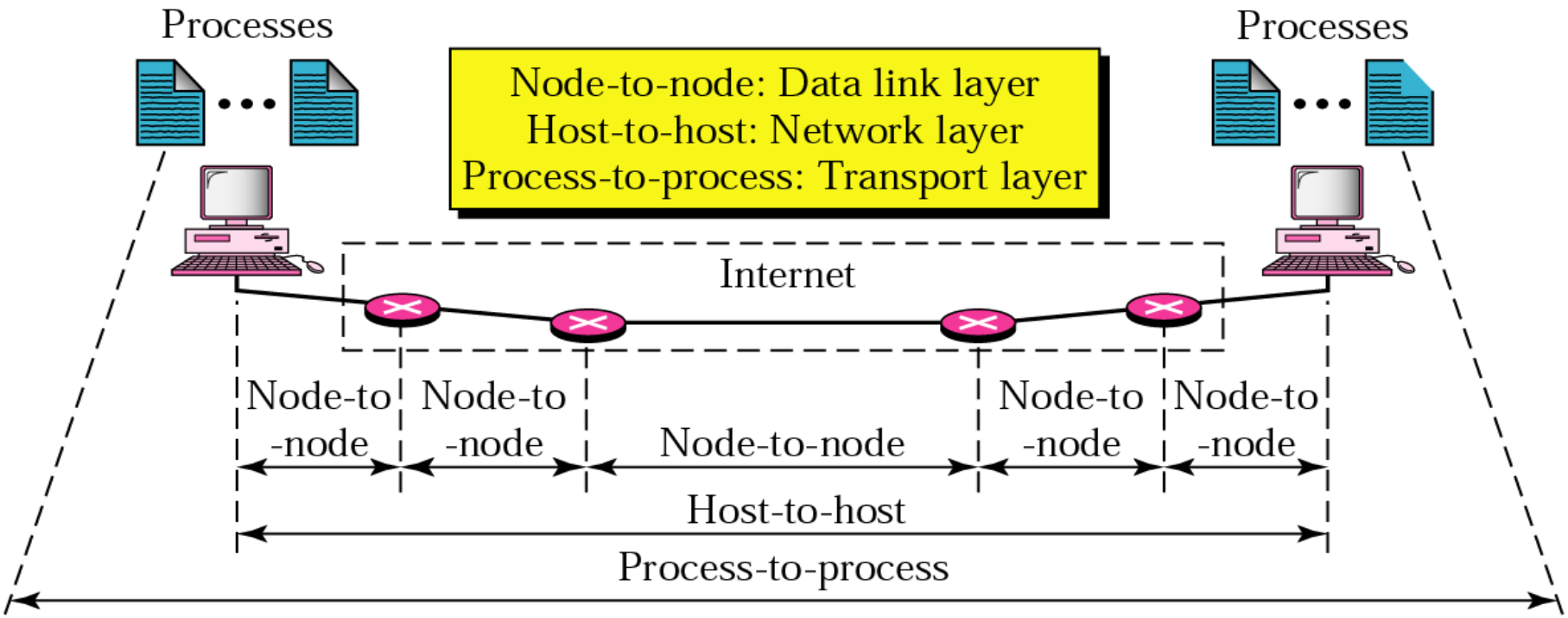
- Camada de Transporte: fornece comunicação lógica entre **processos de aplicação**
  - Depende do serviço de entrega entre sistemas finais (*hosts*) prestado pela camada de rede
- Camada de Rede: fornece comunicação lógica entre ***hosts***
  - Realiza o serviços de entrega entre *hosts* segundo o modelo de serviço "melhor esforço"



Transmissão usando o modelo da Internet – Comunicação física entre camadas



Tipos de comunicação



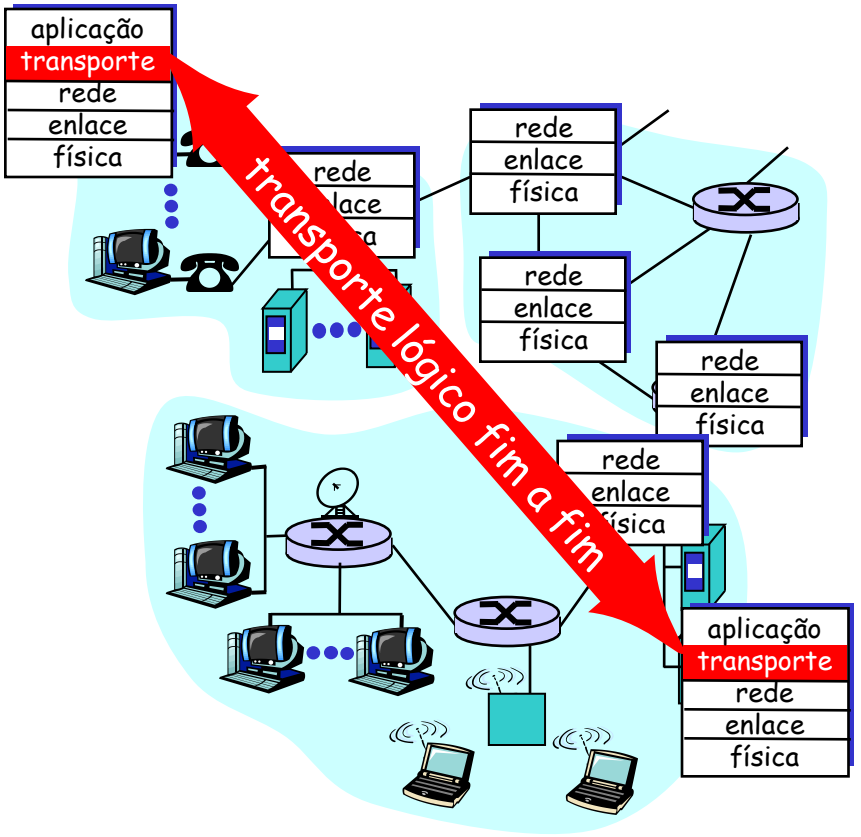
Camada de Transporte vs. Camada de Rede

Objeto do mundo real	Correspondente com a rede de computadores
Primos	Processos
Cartas em envelopes	Mensagens de aplicação
Casas	Hospedeiros
Primos mais velhos	Protocolo da camada de transporte
USPS	Protocolo da camada de rede

Visão geral da camada de transporte na Internet

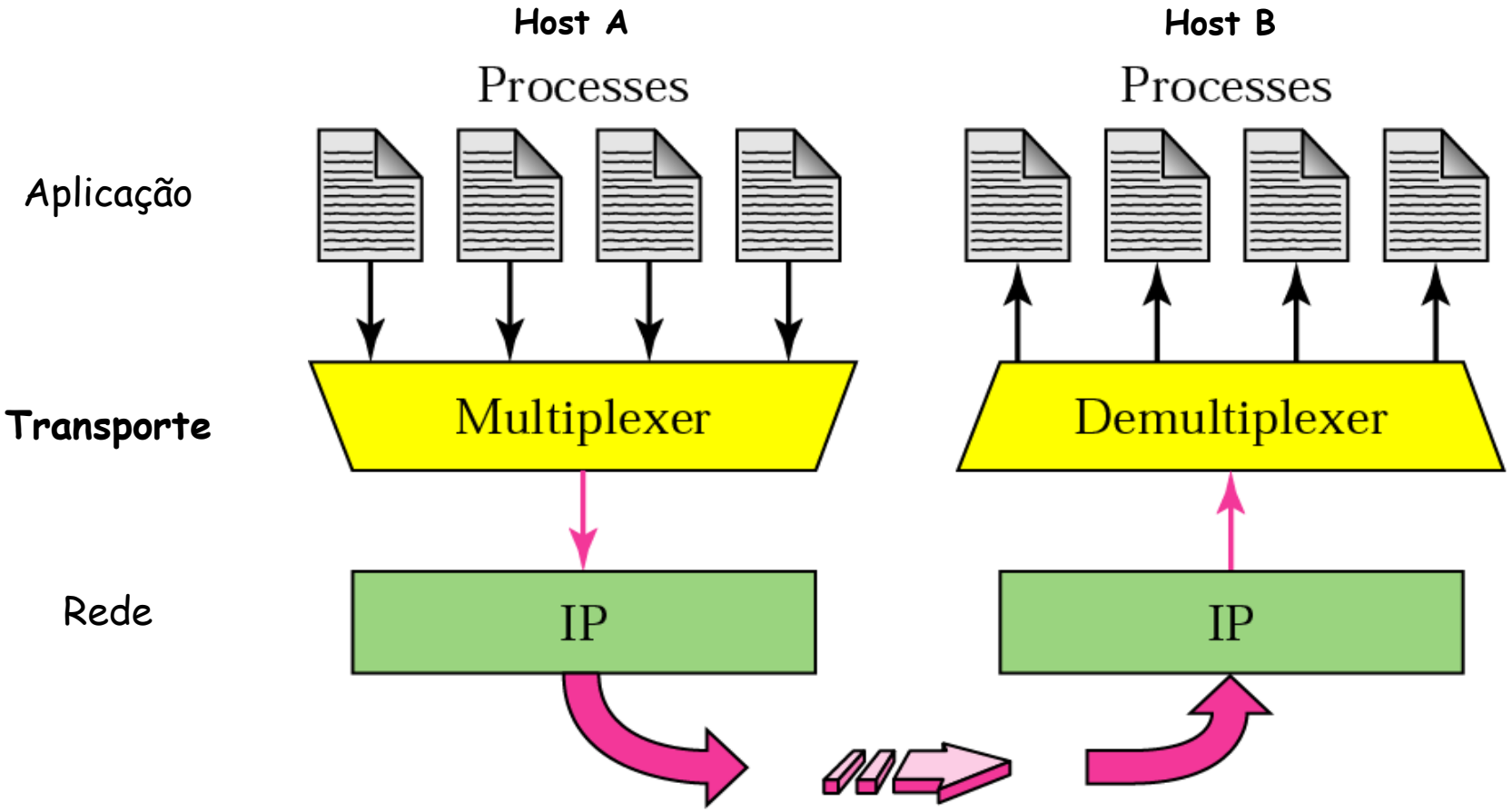
Alguns serviços da camada de transporte na Internet:

- entrega confiável, ordenada, ponto a ponto (TCP):
  - orientado à conexão
  - transferência confiável de dados
  - controle de fluxo
  - controle de congestionamento
- entrega não confiável, ("melhor esforço"), não ordenada, ponto a ponto ou multiponto (UDP)
- serviços não disponíveis:
  - atraso fim a fim limitado
  - garantia de vazão



Multiplexação/Demultiplexação

Serviço fundamental da camada de transporte: é uma ampliação do serviço de entrega host-a-host da camada de rede para um serviço de entrega processo-a-processo



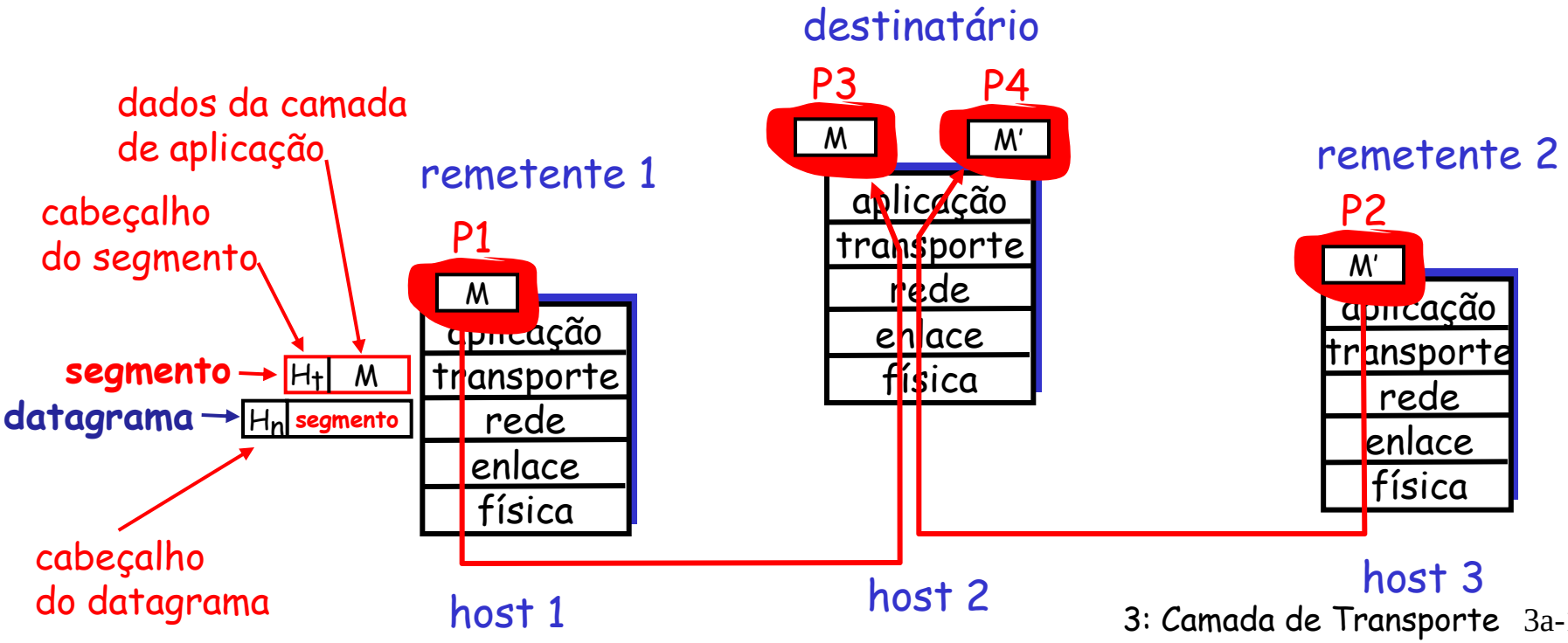
Multiplexação/Demultiplexação

Multiplexação (remetente):

reune os dados provenientes de um processo de aplicação e os envelopa com cabeçalho (útil na demultiplexação), formando um segmento

Demultiplexação (receptor):

entrega de dados contidos em um segmento da camada de transporte ao processo de aplicação destinatário

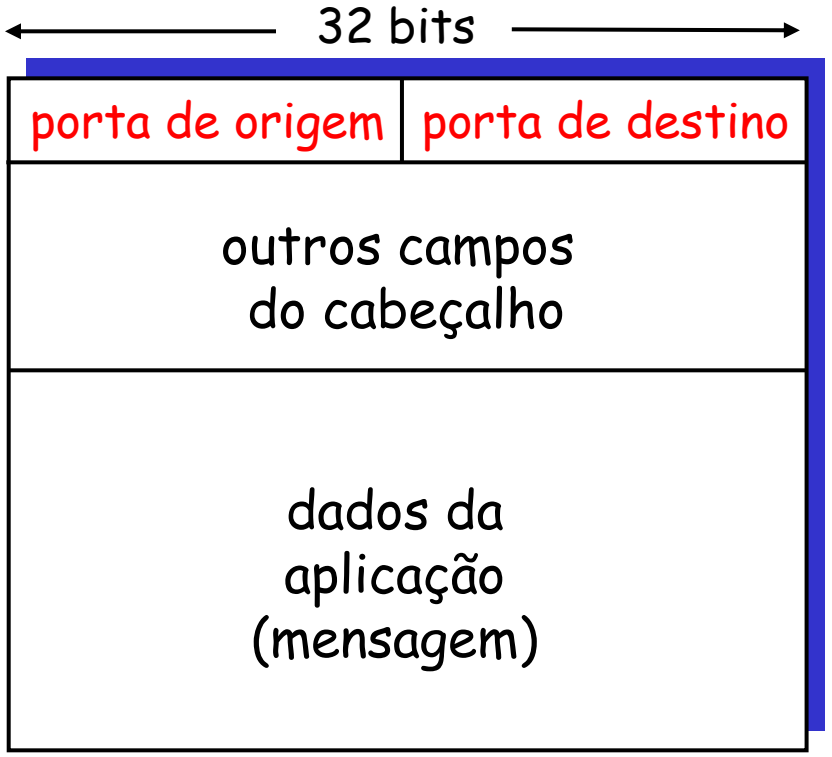


Multiplexação/Demultiplexação

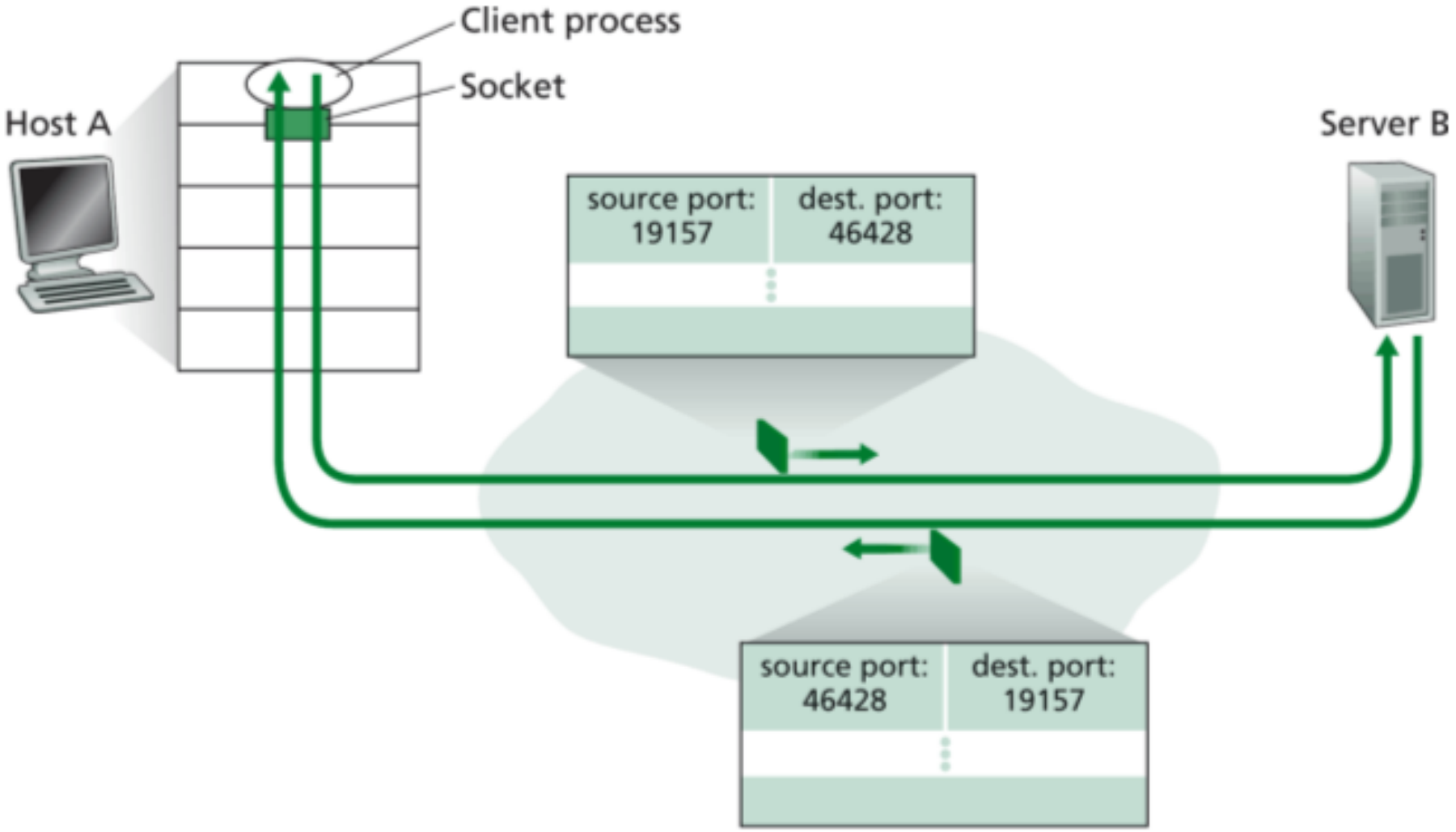
Como ocorre a demultiplexação:

- cada datagrama contém campos com o endereço IP de origem e o endereço IP de destino
- cada datagrama transporta um segmento da camada de transporte
- cada segmento contém campos com o número da porta de origem e o número da porta de destino (RFC 1700)
- os dados da aplicação são passados ao processo de aplicação identificado pelo número de porta de destino

Formato básico de um segmento:



Exemplo: inversão dos número de portas da fonte e do destino



Multiplexação e Demultiplexação Não Orientadas p/ Conexão

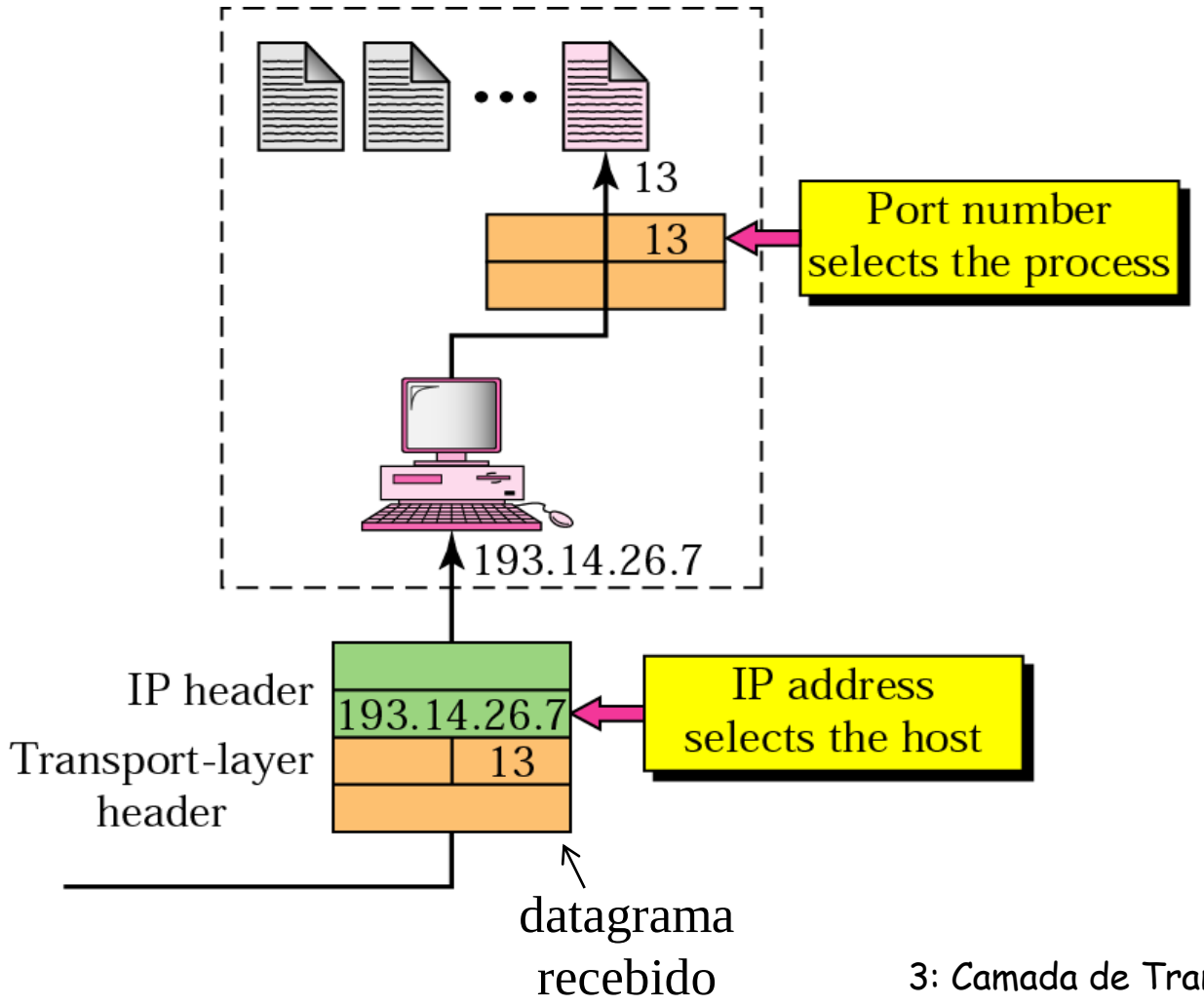
- Um socket UDP é totalmente identificado por dois valores:
  - **endereço IP de destino**
  - **número de porta de destino**

- 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

- Segmentos UDP oriundos de hosts com endereços IP diferentes e/ou portas de origem diferentes, porém com o mesmo endereço IP de destino e o mesmo número de porta de destino:
  - serão direcionados ao mesmo processo de aplicação de destino

Multiplexação/Demultiplexação

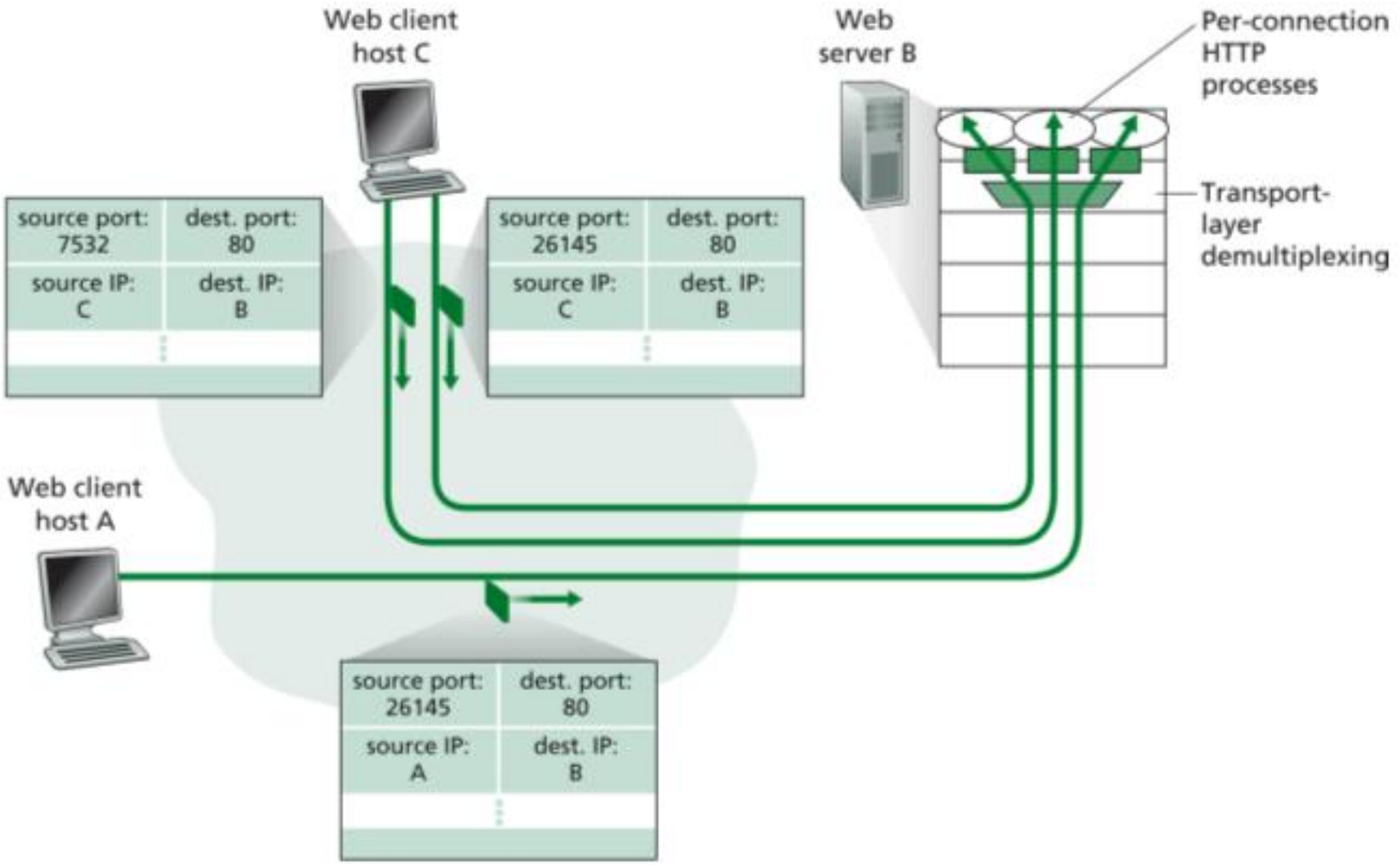
Exemplo



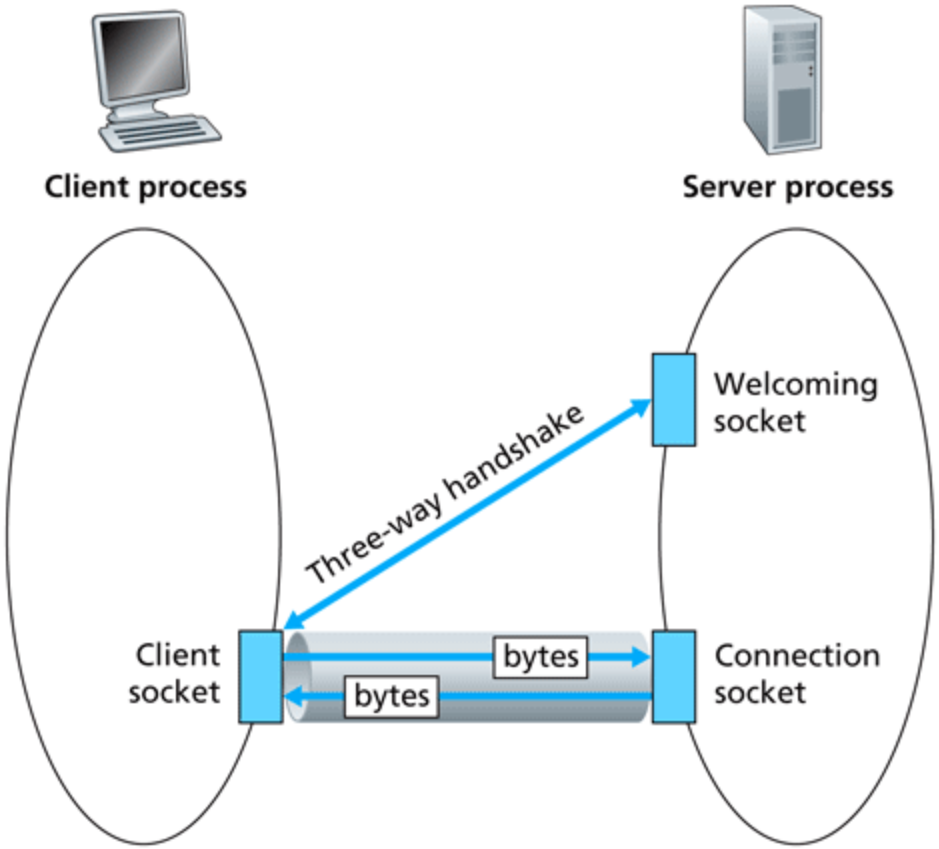
Multiplexação e Demultiplexação Orientadas para Conexão

- Um *socket* TCP é identificado por quatro elementos:
  - **endereço IP da fonte**
  - **número de porta da fonte**
  - **endereço IP de destino**
  - **número de porta de destino**
- A camada de transporte no host receptor usa todos os quatro elementos para direcionar (demultiplexar) o segmento TCP para o *socket* apropriado
- Dois segmentos TCP oriundos de hosts diferentes ou com números de portas diferentes serão encaminhados para *sockets* diferentes. Exceção: estabelecimento de conexão: usa o *Welcoming socket.*
- Um host servidor pode suportar vários *sockets* TCP simultâneos, sendo cada qual:
  - identificado pelos seus próprios quatro elementos
  - ligado a um processo

Multiplexação e Demultiplexação Orientados para Conexão: Exemplo



Multiplexação e Demultiplexação Orientadas para Conexão



UDP: User Datagram Protocol [RFC 768]

- Fornece comunicação processo a processo
- Protocolo mínimo de transporte na Internet
  - multiplexação / demultiplexação
- Serviço "melhor esforço", segmentos UDP podem ser:
  - perdidos
  - entregues à aplicação fora de ordem
- Não-orientado à conexão:
  - não há fase de "setup"
  - tratamento independente de cada segmento UDP

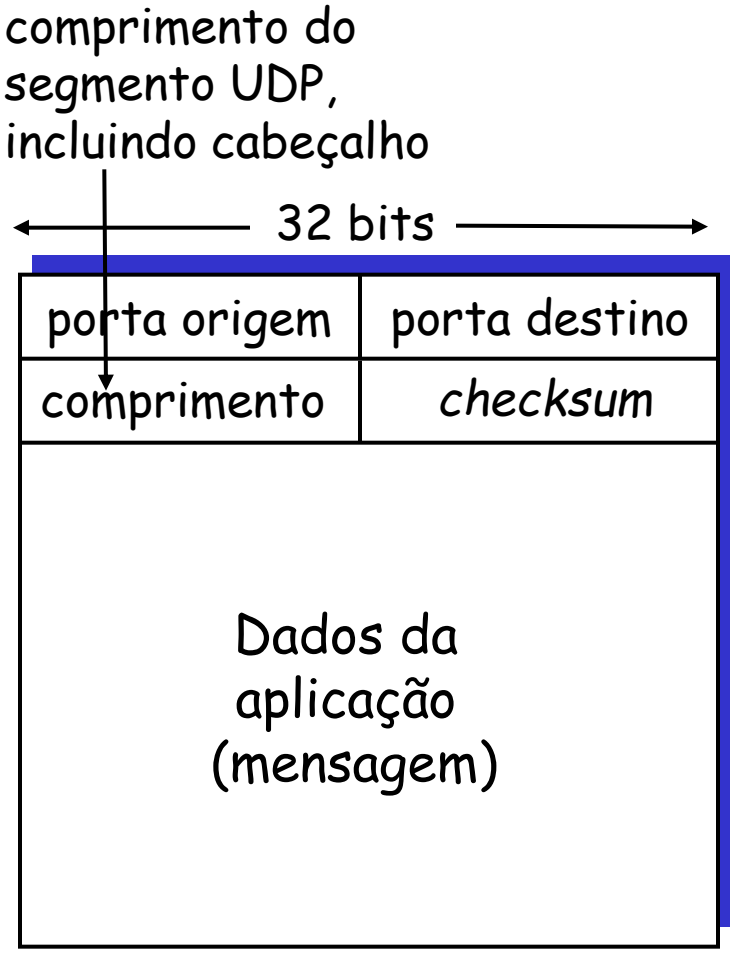
Por quê existe UDP?

- Elimina estabelecimento de conexão, que causa atraso
- Simples: não mantém estado de conexão no remetente/receptor
- Cabeçalho do segmento UDP de tamanho reduzido (*overhead* pequeno)
- Sem controle de congestionamento: UDP pode transmitir mais rápidamente, pois a taxa de envio não é regulada

Mais sobre UDP

- Muito utilizado pelas aplicações de fluxo contínuo (voz, vídeo)
  - tolerantes a perdas
  - sensíveis à vazão da rede
- Outros usos do UDP (por quê?):
  - DNS (nomes)
  - SNMP (gerenciamento)
- Transferência confiável com UDP:
  - incluir a confiabilidade na própria aplicação

Estrutura do segmento UDP:



UDP Checksum

Meta: detectar "erros" no segmento UDP recebido

Remetente:

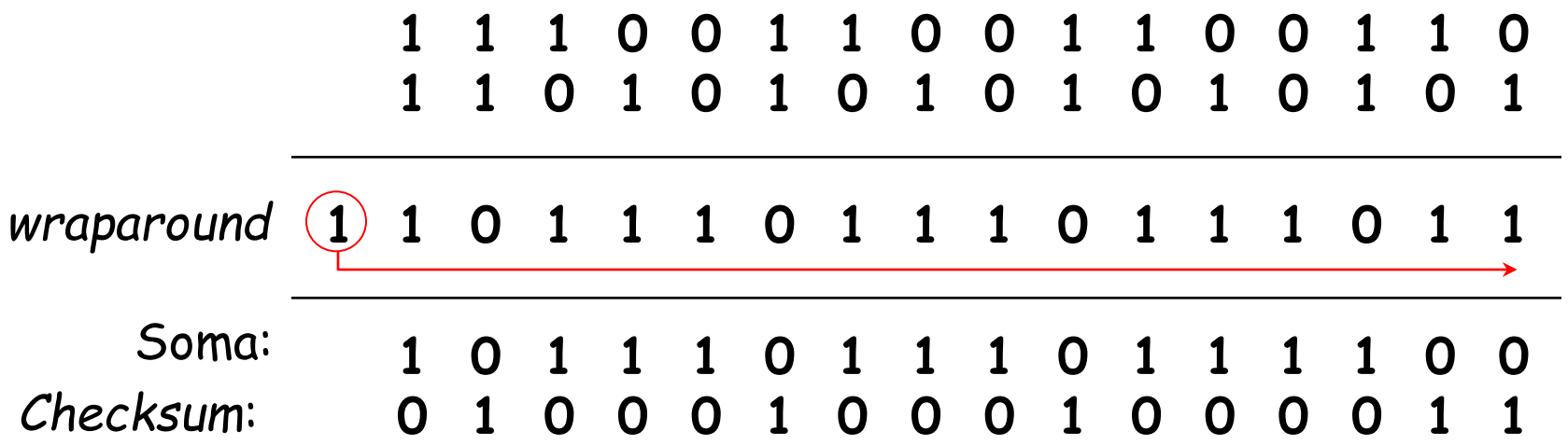
- Trata o conteúdo do segmento UDP como sequência de palavras de 16 bits
- Soma todas as palavras de 16 bits no segmento
- Obtém o complemento de 1 do resultado da soma e o coloca no campo *checksum* do cabeçalho do segmento UDP

Destinatário:

- Todas as palavras de 16 bits do segmento UDP recebido são somadas, inclusive o próprio *checksum*
- Todos os bits do resultado da soma anterior são iguais a "1"?
  - NÃO -> erro detectado
  - SIM -> nenhum erro detectado

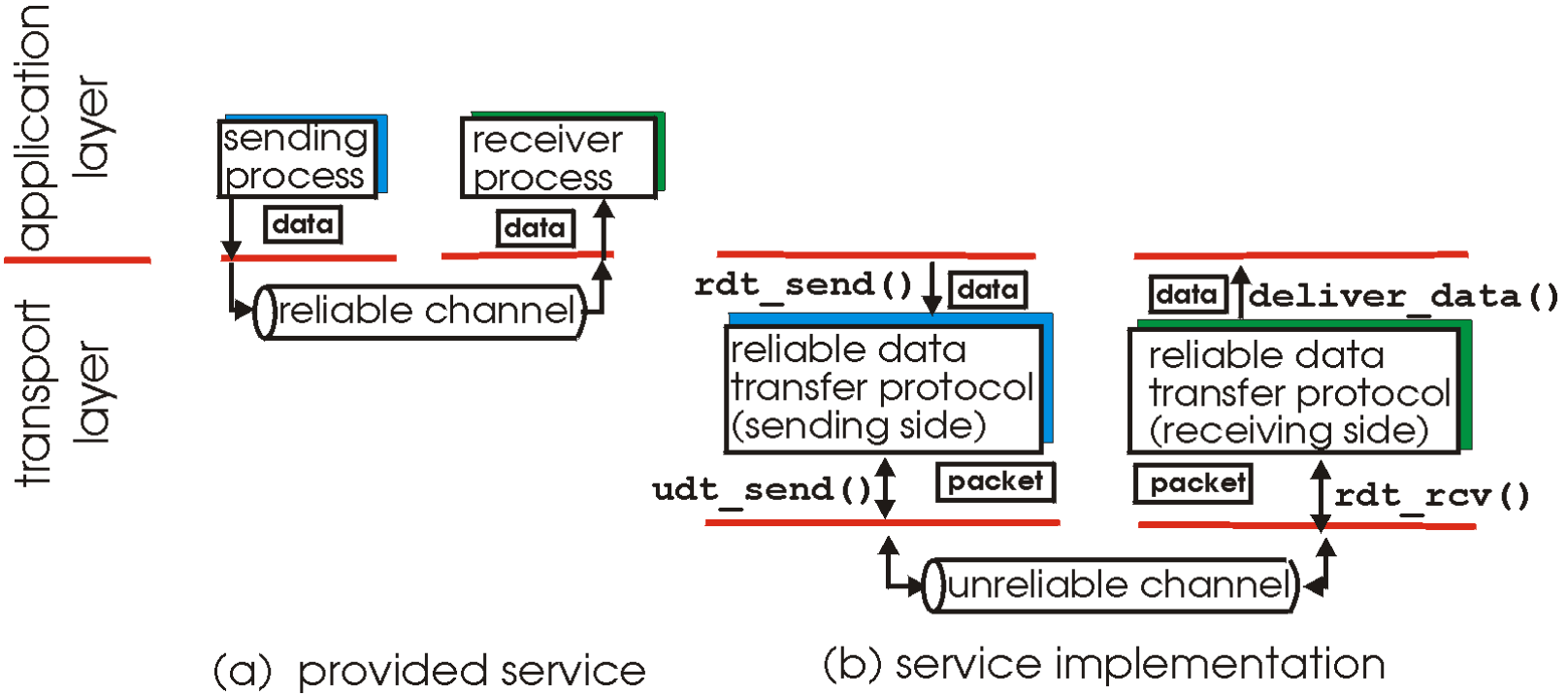
UDP Checksum

- Note que o "vai um" no bit mais significativo precisa ser adicionado ao resultado.
- Exemplo: adição de duas palavras de 16 bits



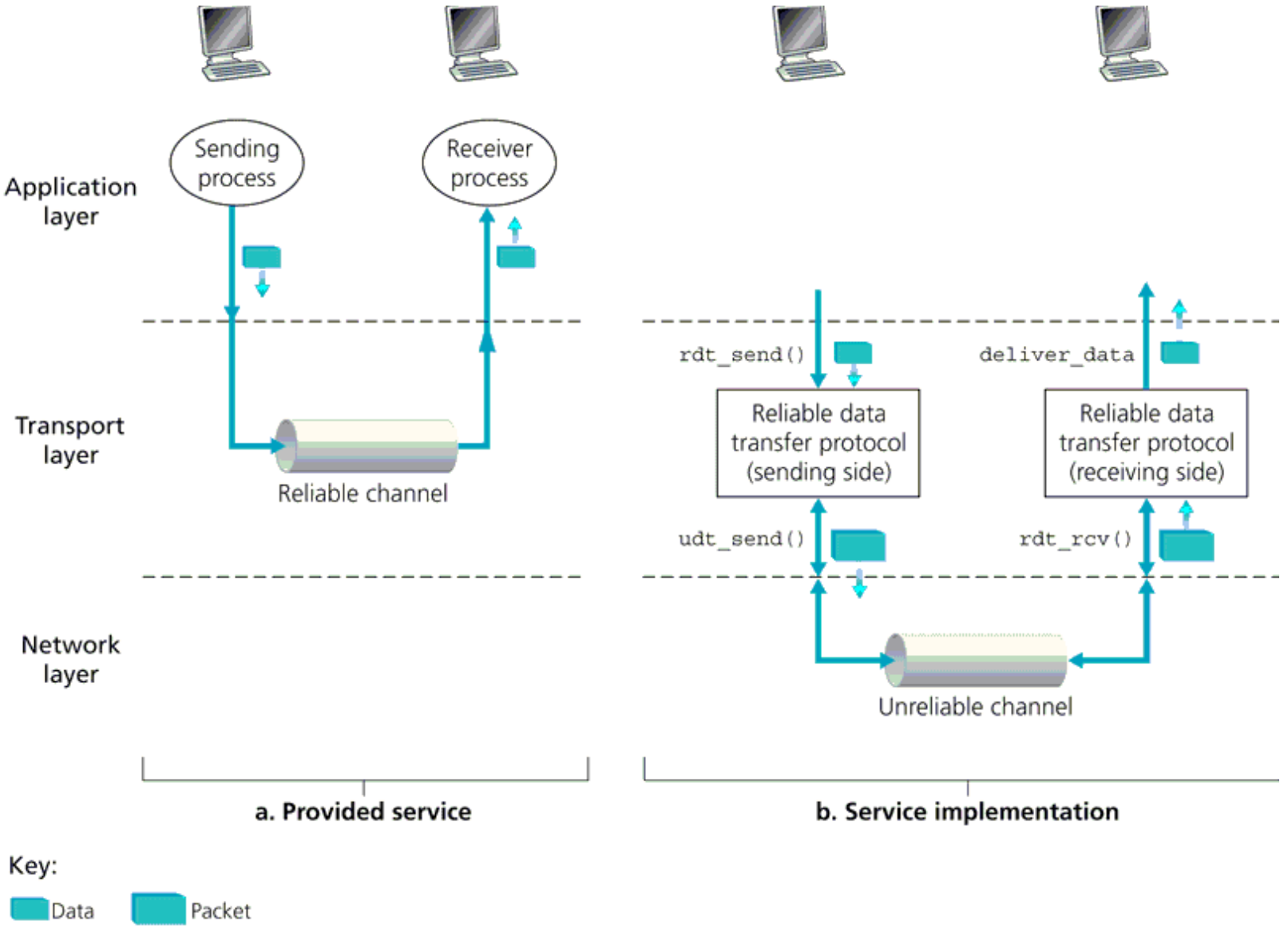
Princípios da transferência confiável de dados

- Importante nas camadas de transporte e de enlace
- Está na lista dos 10 tópicos mais importantes em redes!
- Modelo do serviço e implementação do serviço:



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

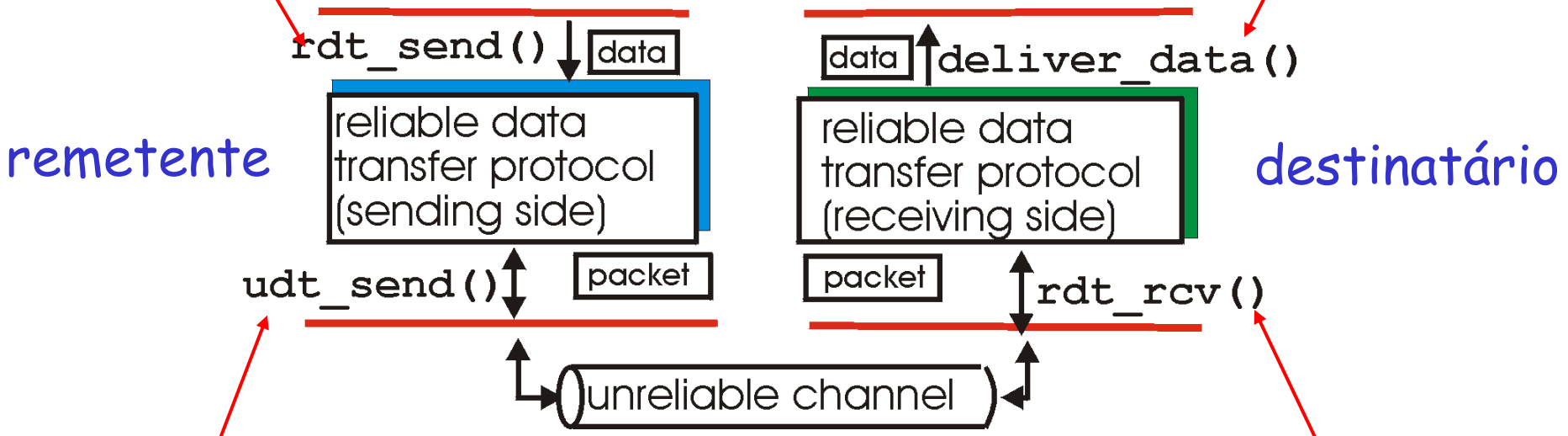
Princípios da transferência confiável de dados



Princípios da transferência confiável de dados (cont.)

rdt_send(): chamada de cima, (ex.: pela apl.). Passa dados a serem entregues à camada sup. receptora

deliver_data(): chamada para entregar dados para a camada superior



udt_send(): chamada para transferir pacotes para o receptor sobre o canal não confiável

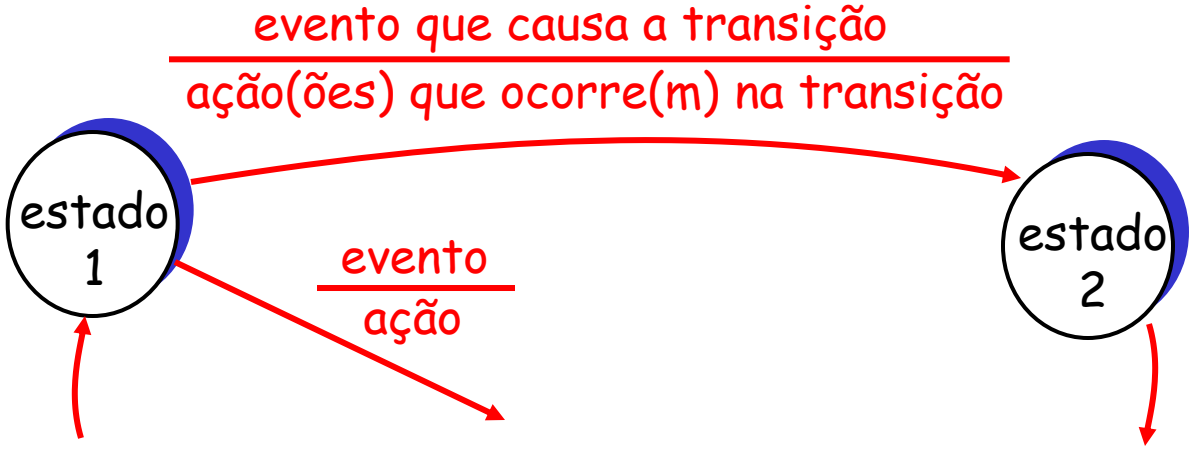
rdt_rcv(): chamada quando o pacote chega no lado receptor do canal

Transferência confiável de dados: como começar

Iremos:

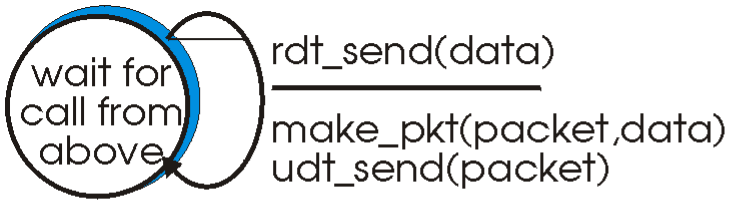
- Desenvolver incrementalmente os lados remetente e receptor do protocolo rdt (*reliable data transfer*)
- Considerar apenas fluxo unidirecional de dados
  - mas a informação de controle flui em ambos sentidos!
- Usar máquinas de estados finitos (FSM) p/ especificar remetente e receptor

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

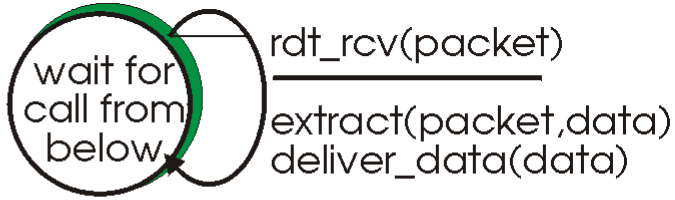


rdt1.0: Transferência confiável usando canal confiável

- Canal subjacente é perfeitamente confiável
  - não apresenta erros de bits
  - não apresenta perda de pacotes
- FSMs separadas para remetente e receptor:
  - remetente envia dados pelo canal confiável
  - receptor recebe dados do canal confiável



(a) rdt1.0: sending side



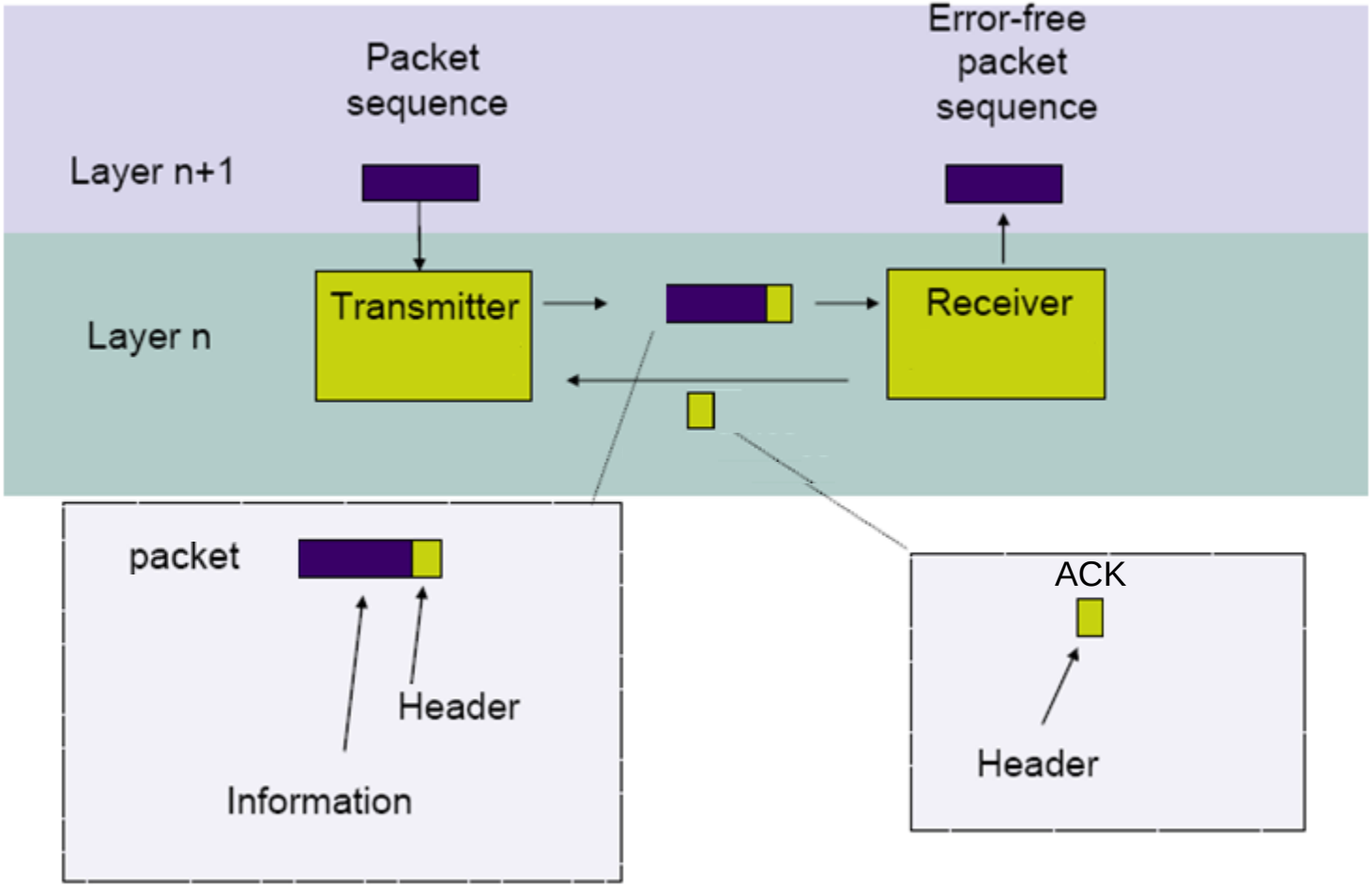
(b) rdt1.0: receiving side

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

- Canal subjacente pode inverter bits no pacote
  - lembrete: *checksum* UDP pode detectar erros de bits
- A questão: como recuperar dos erros detectados?
  - reconhecimento positivo (ACK): receptor avisa explicitamente ao remetente que pacote chegou OK
  - reconhecimento negativo (NAK): receptor avisa explicitamente ao remetente que pacote recebido continha erros
    - remetente retransmite pacote ao receber um NAK
  - protocolos ARQ (*Automatic Repeat reQuest*)
- Novos mecanismos em **`rdt2.0`** (além do **`rdt1.0`**):
  - deteção de erros
  - realimentação pelo receptor: mensagens de controle (ACK e NAK) receptor -> remetente
  - retransmissão de pacotes

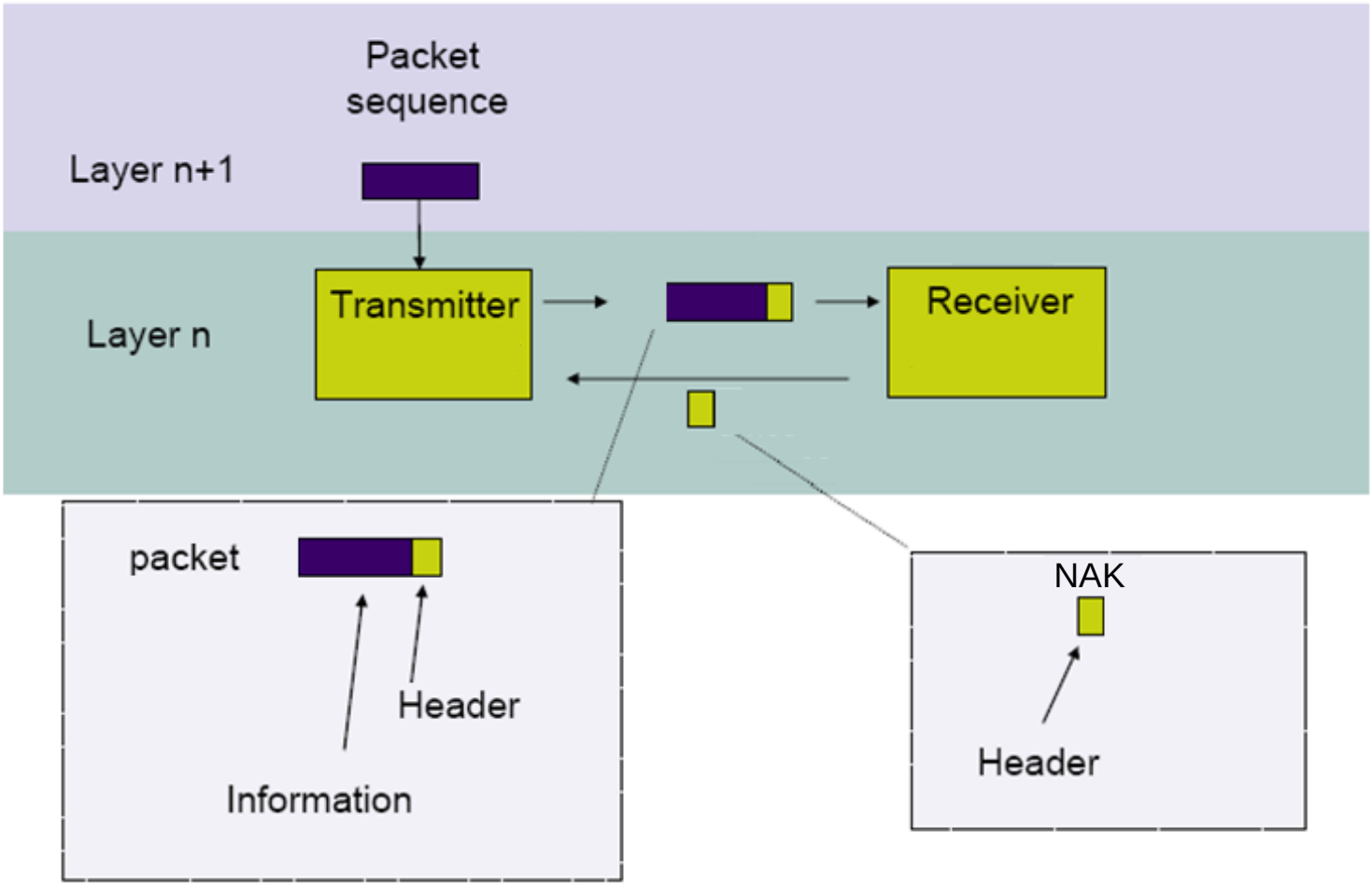
rdt2.0: Elementos Básicos do ARQ

Reconhecimento Positivo (ACK)

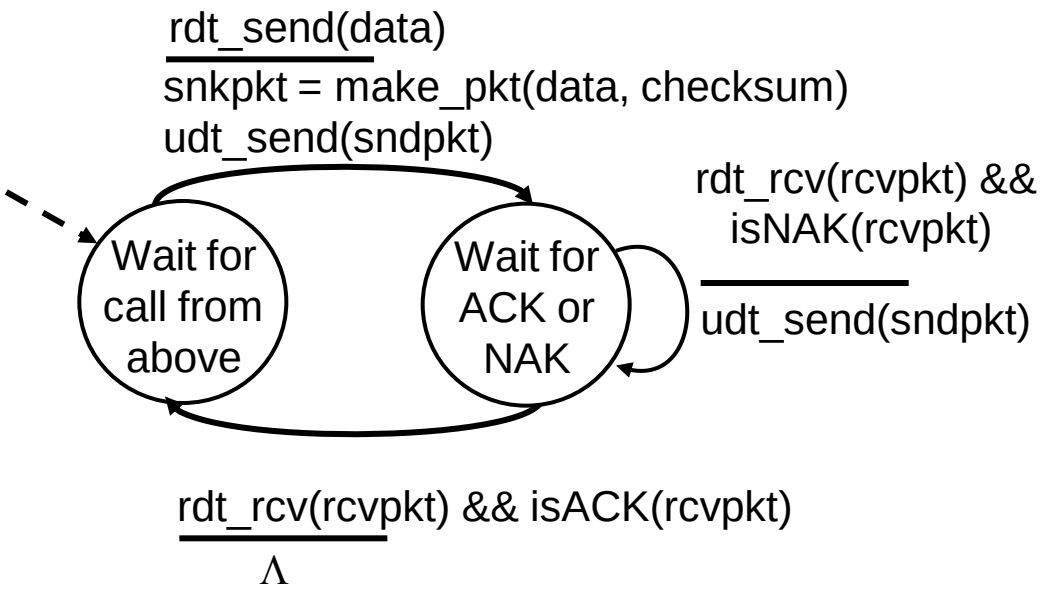


rdt2.0: Elementos Básicos do ARQ

Reconhecimento Negativo (NAK)

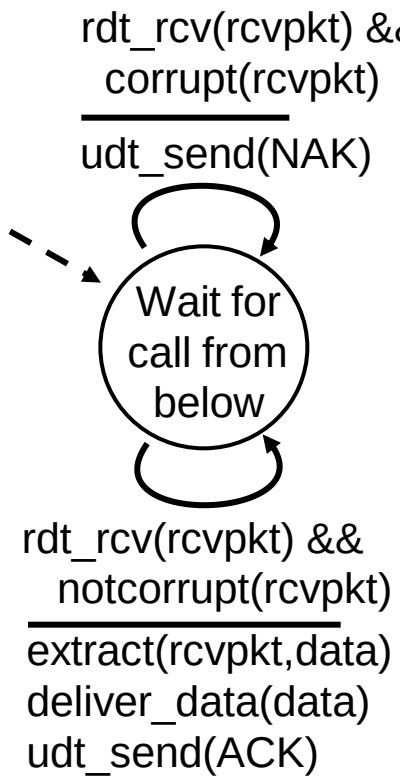


rdt2.0: especificação da FSM

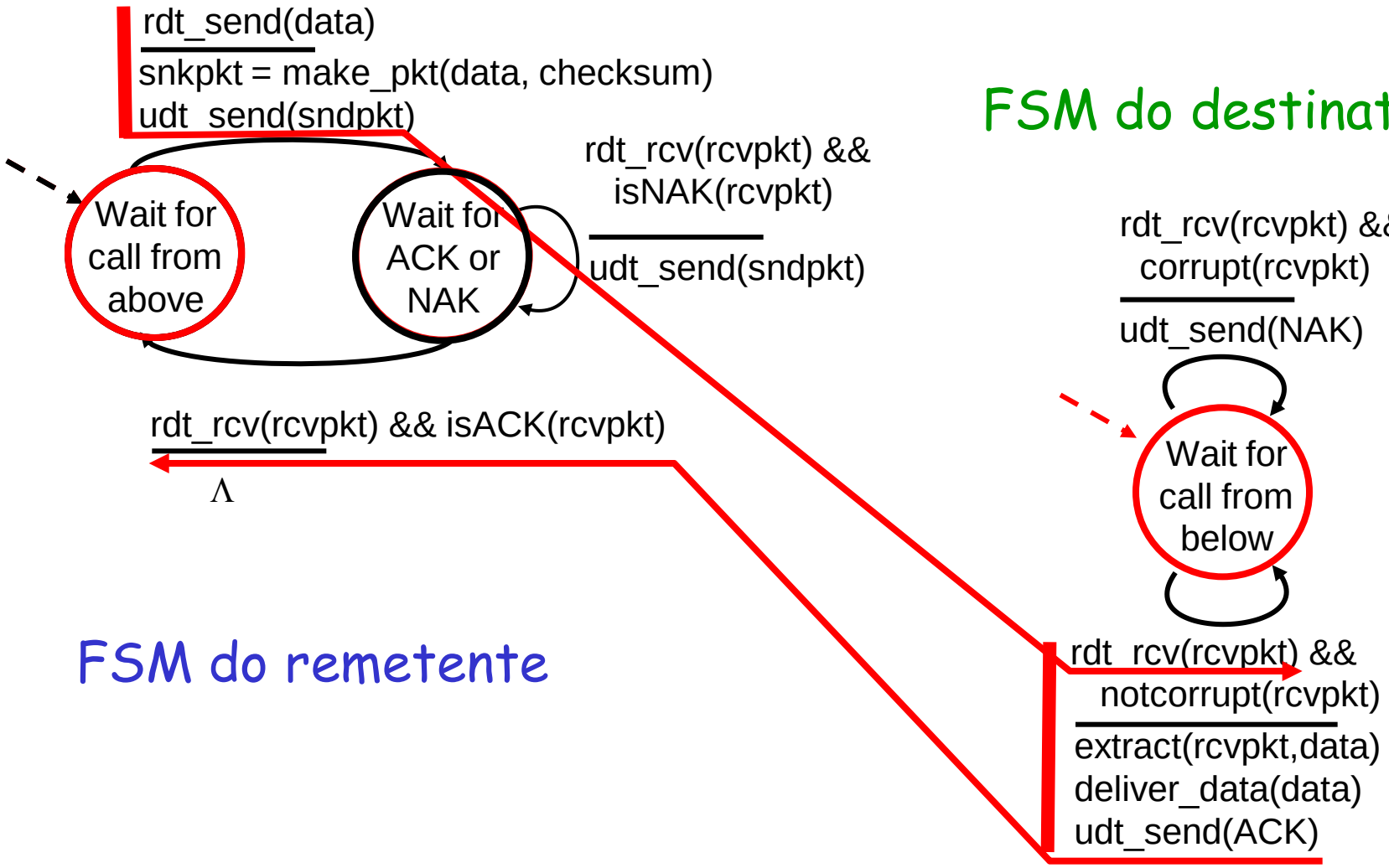


FSM do remetente

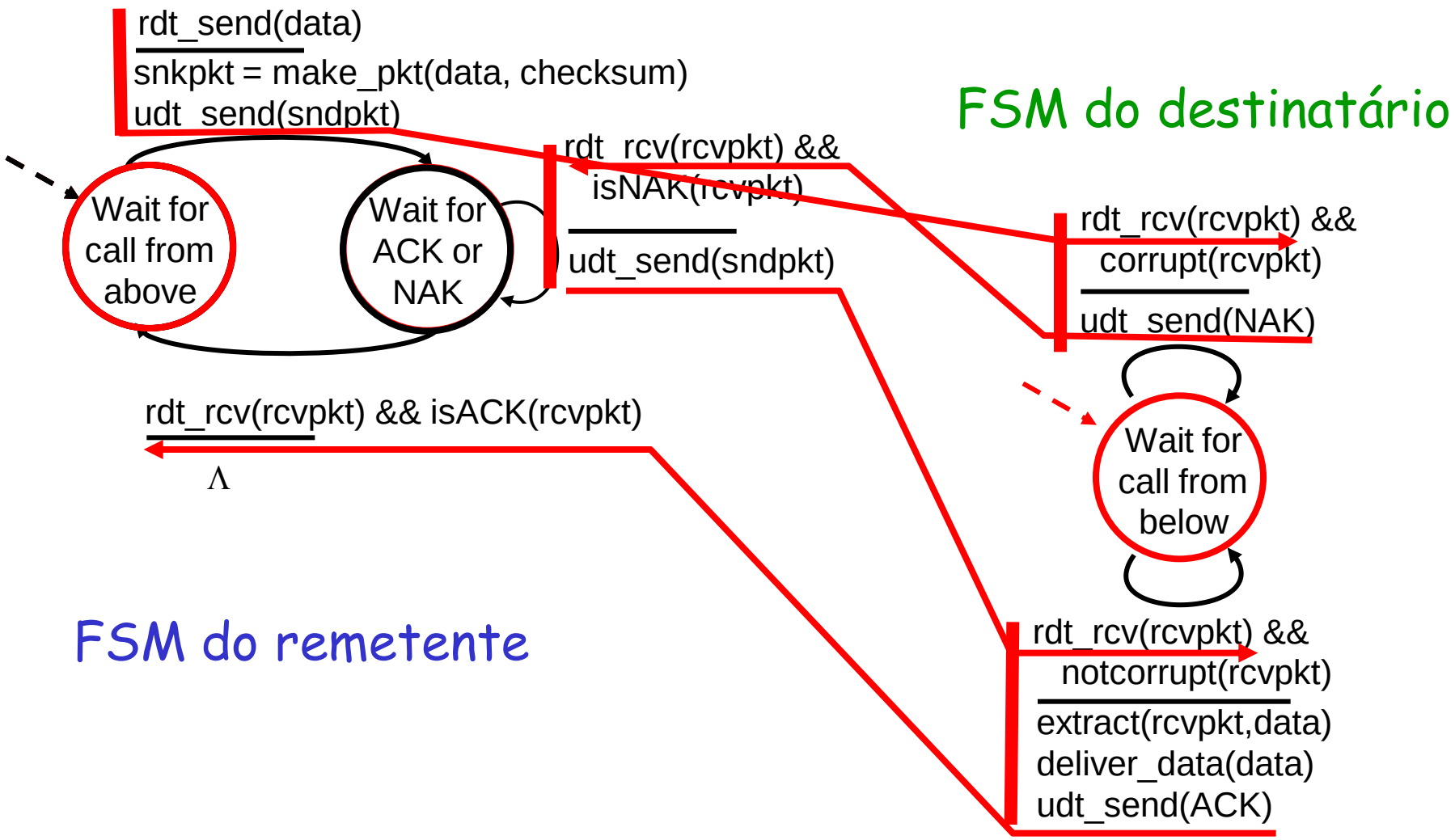
FSM do destinatário



rdt2.0: em ação (sem erros)



rdt2.0: em ação (Cenário de Erro)



rdt2.0 tem uma falha fatal!

O que pode acontecer se um ACK ou NAK estiver corrompido?

- Remetente não tem como saber se o destinatário recebeu OK os dados transmitidos!

O que fazer?

- O remetente simplesmente reenvia o pacote de dados se ACK ou NAK chegar corrompido.
- Todavia, este procedimento poderá causar a retransmissão de pacote recebido OK!

Enfrentando as duplicações:

- O remetente deve adicionar o campo *número de sequência* em cada pacote
- O remetente retransmite o pacote atual se ACK ou NAK for recebido com erro
- Destinatário descarta (não entrega) pacote duplicado

Stop-and-Wait

Remetente envia um pacote e, então aguarda a resposta do destinatário

rdt2.1: discussão

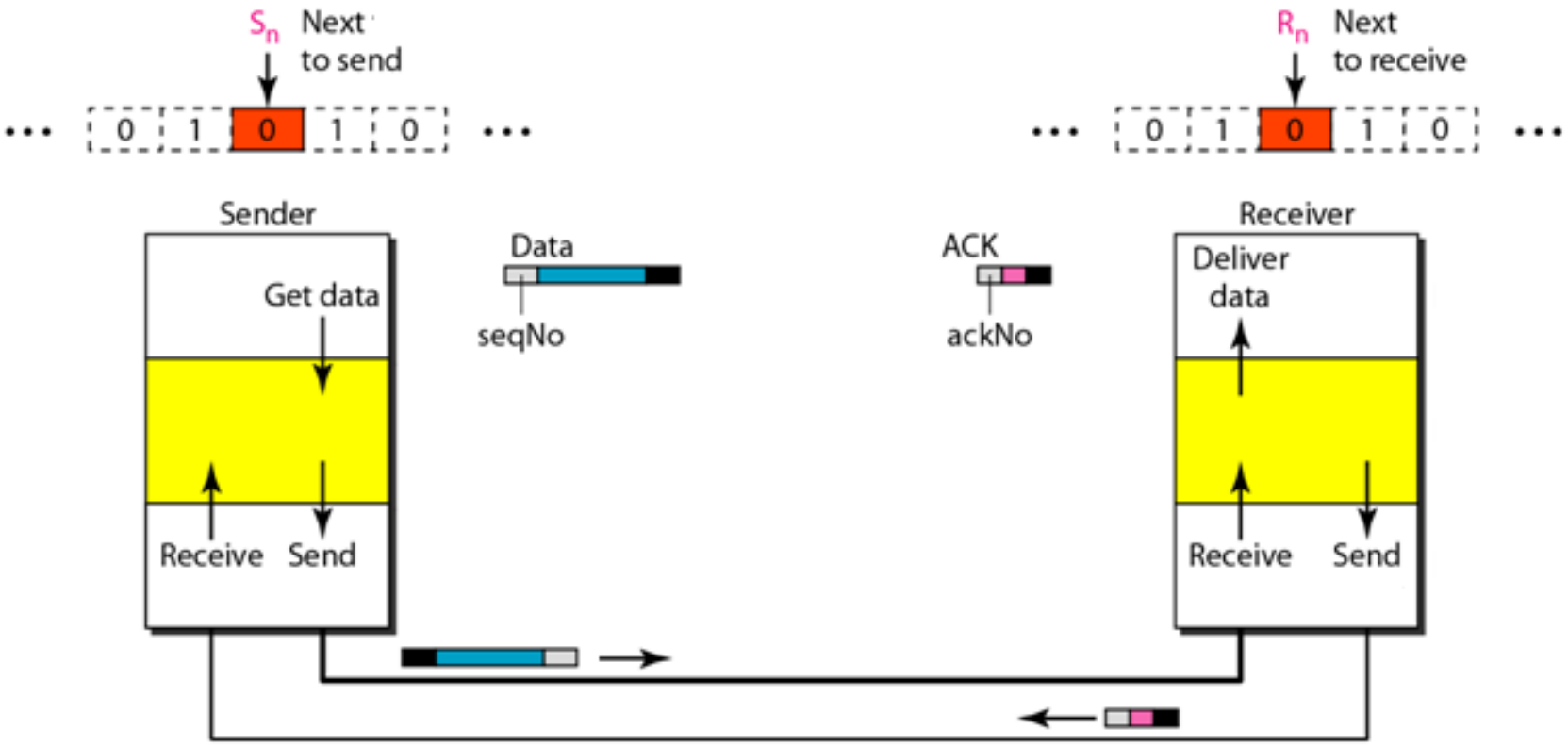
Remetente:

- Usa nº de sequência no pacote
- Bastam dois nºs de sequência (0,1). Por quê?
- O nº de estados é duplicado
  - Estado permite "lembrar" se pacote corrente tem nº de sequência 0 ou 1
- Deve "checar" se ACK/NAK recebido tem erro

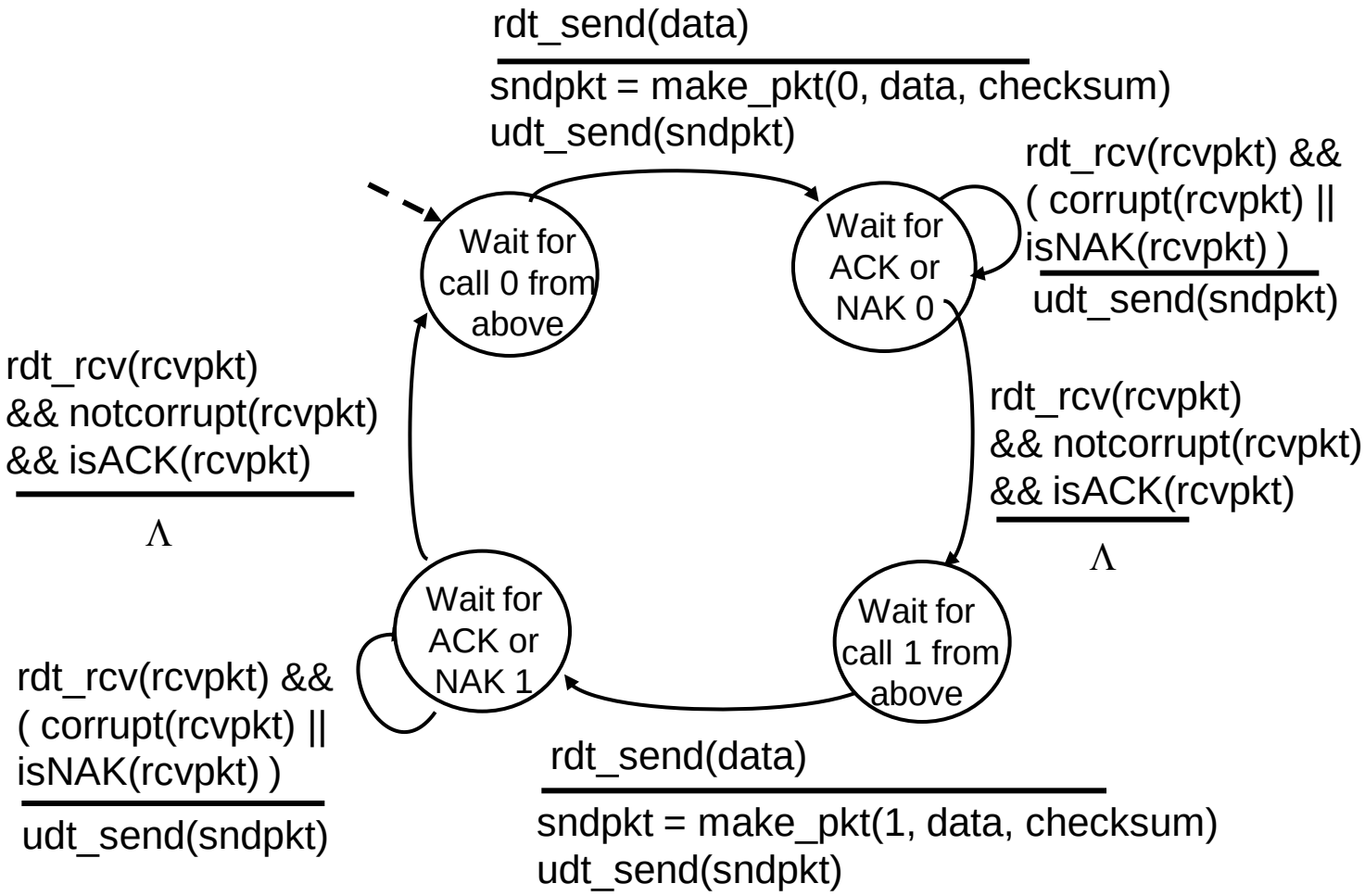
Receptor:

- Deve "checar" se pacote recebido foi duplicado
  - estado indica se nº de sequência esperado é 0 ou 1
- Não tem como saber se último ACK/NAK foi recebido OK pelo remetente

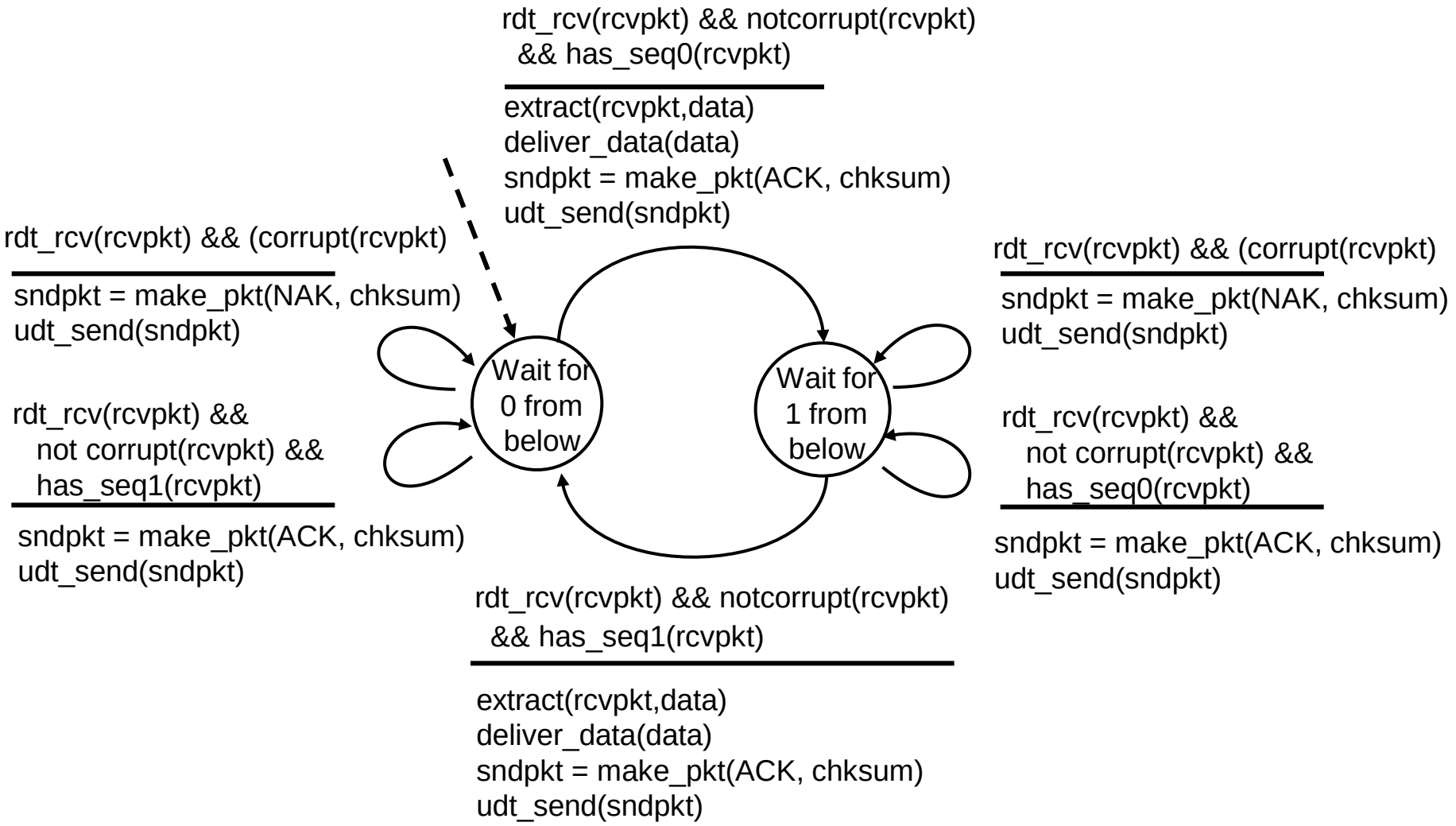
rdt2.1: discussão



rdt2.1 remetente (trata ACK ou NAK com erro)



rdt2.1 receptor (trata ACK ou NAK com erro)

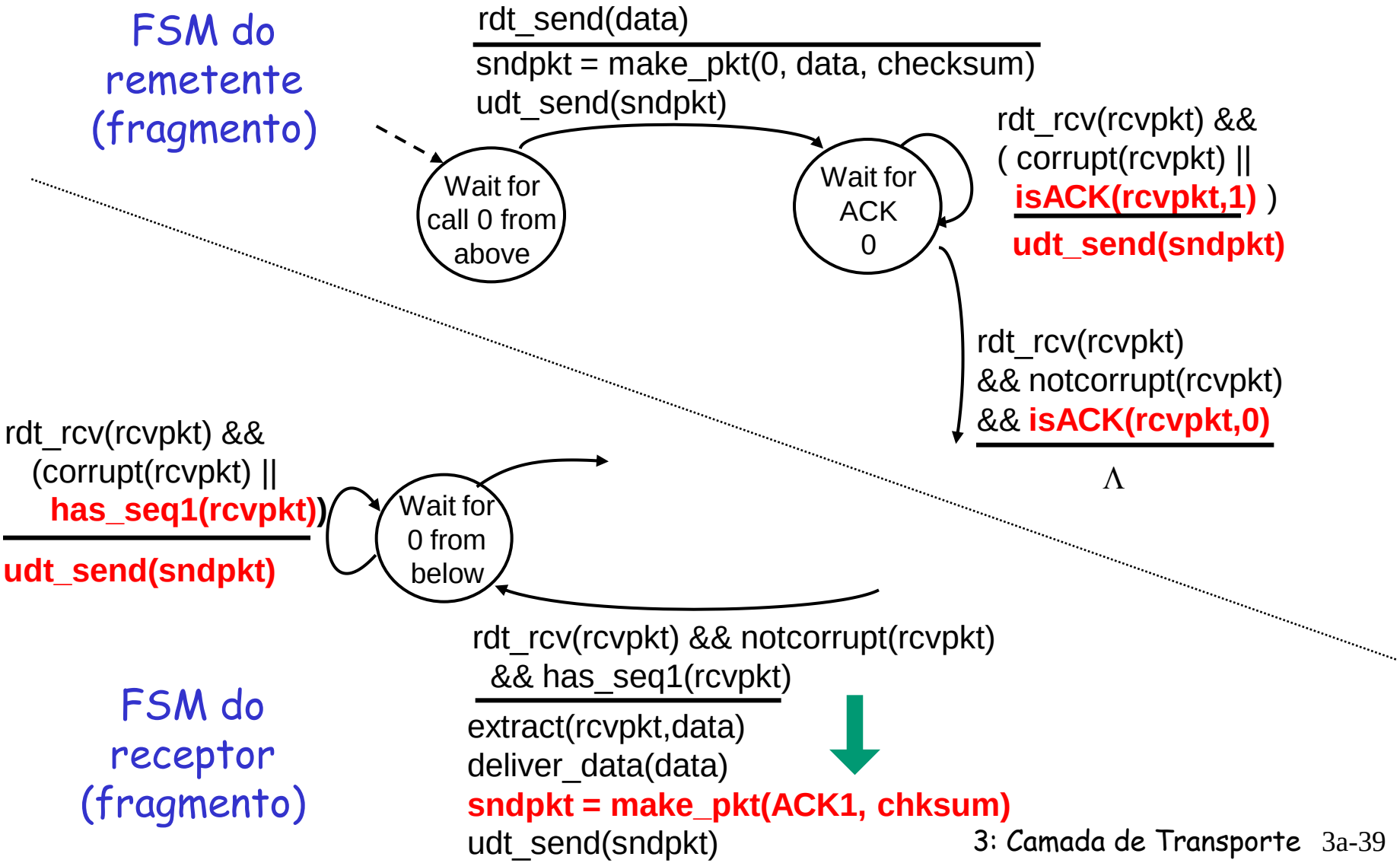


rdt2.2: um protocolo sem NAKs

- Só com ACKs, mas com a mesma funcionalidade de rdt2.1
- Ao invés de enviar um NAK, o receptor envia um ACK em seu lugar para o último pacote recebido OK
- **ACK duplicado:** no remetente resulta na mesma ação que o NAK: "retransmissão do pacote atual"

rdt2.2: um protocolo sem NAKs (cont.)

FSM do remetente (fragmento)



FSM do receptor (fragmento)

rdt3.0: canal com erros e perdas

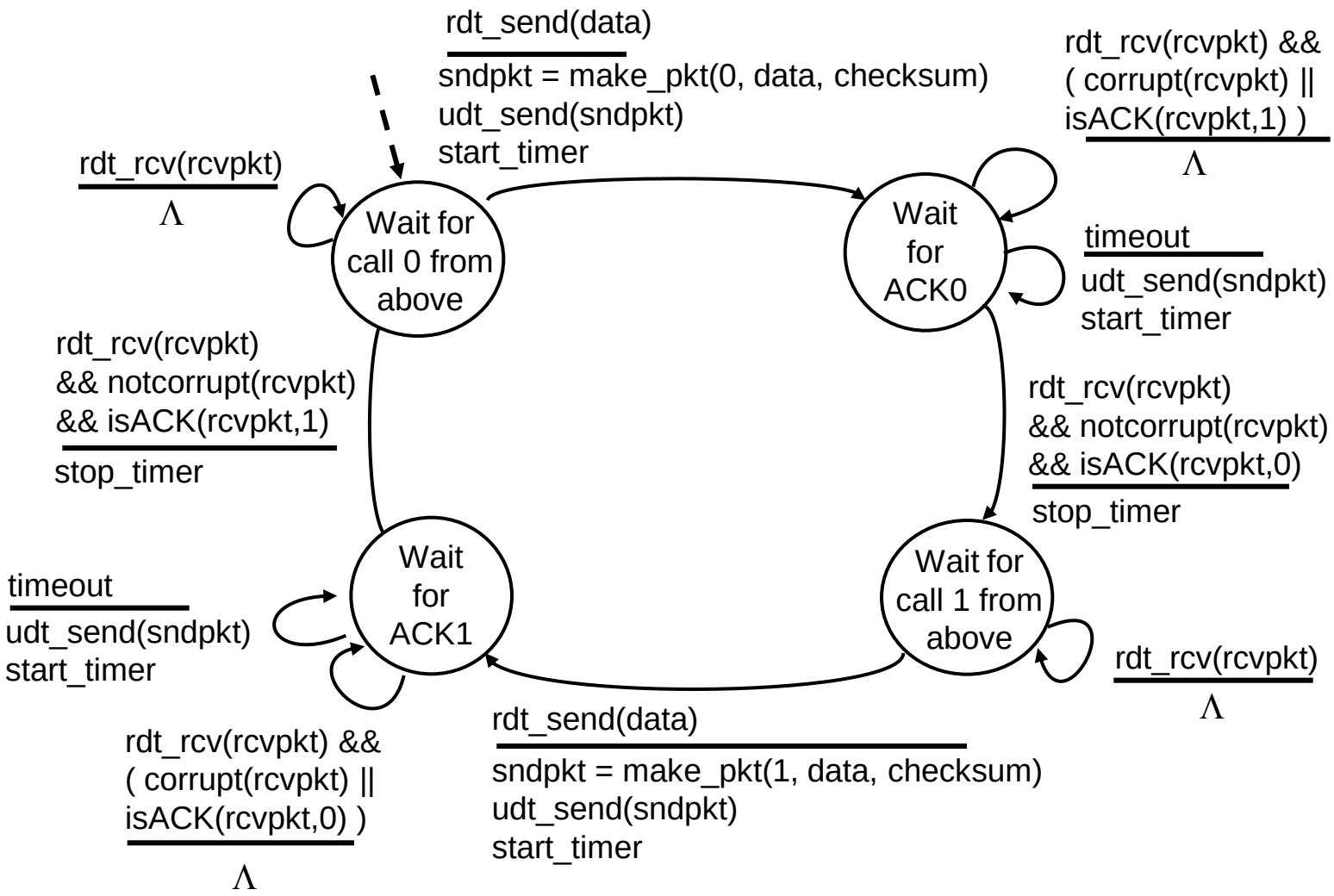
Nova suposição: canal subjacente também pode, adicionalmente, **perder pacotes** (dados ou ACKs)

- *Checksum*, nº de sequência, ACKs, retransmissões podem ajudar, mas não serão suficientes
- P: Como lidar com perdas?
  - Remetente espera até ter "certeza" que pacote ou ACK se perdeu e, então, retransmite o pacote
  - Quanto tempo esperar?

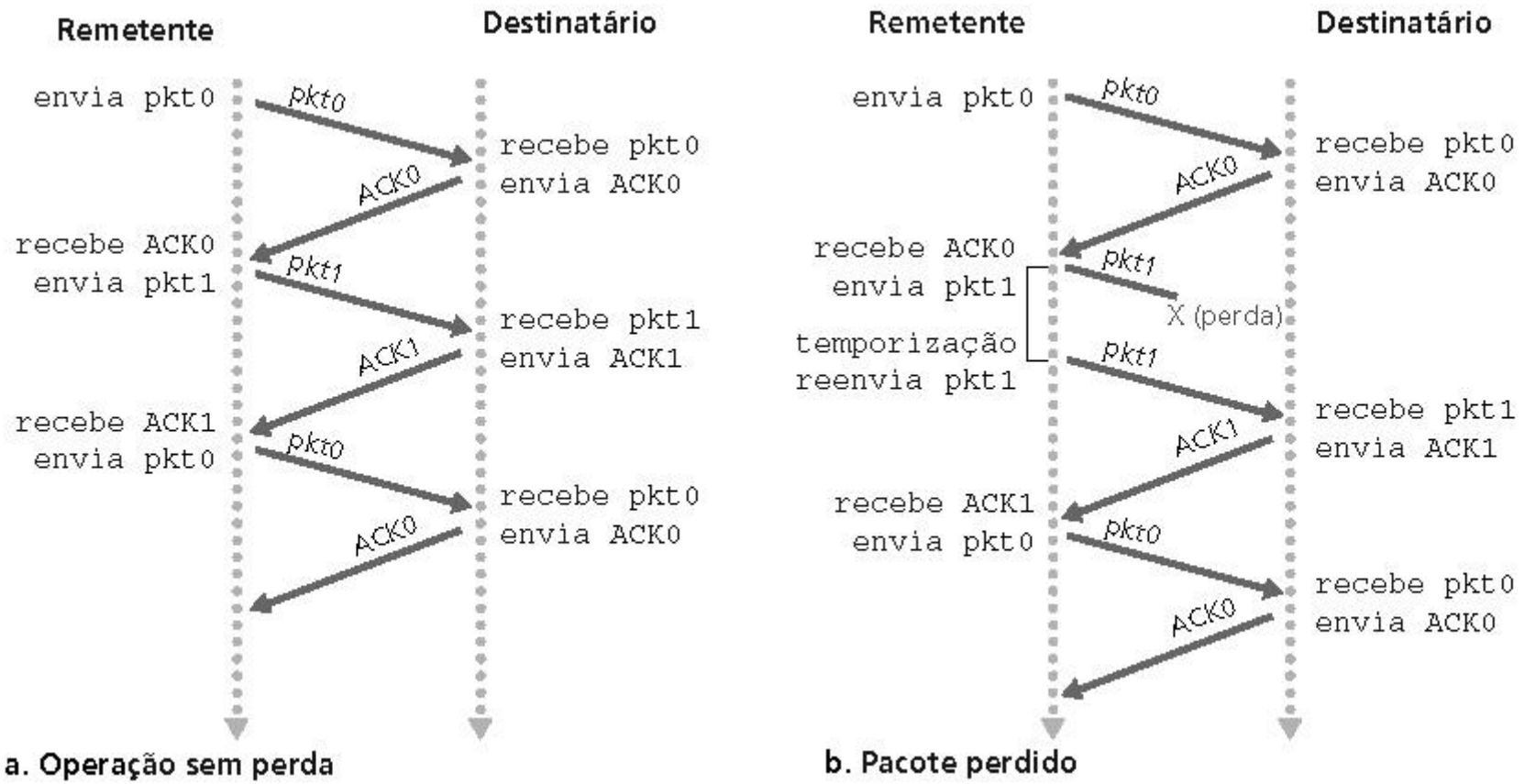
Abordagem: remetente aguarda um tempo "razoável" pelo ACK

- Retransmite se nenhum ACK é recebido neste intervalo
- Se pacote (ou ACK) está apenas atrasado (e não perdido):
  - retransmissão causa duplicação, mas uso de nº de sequência já cuida disso
  - receptor deve especificar nº de sequência do pacote que está sendo reconhecido
- Requer **temporizador de contagem regressiva** ("timeout")

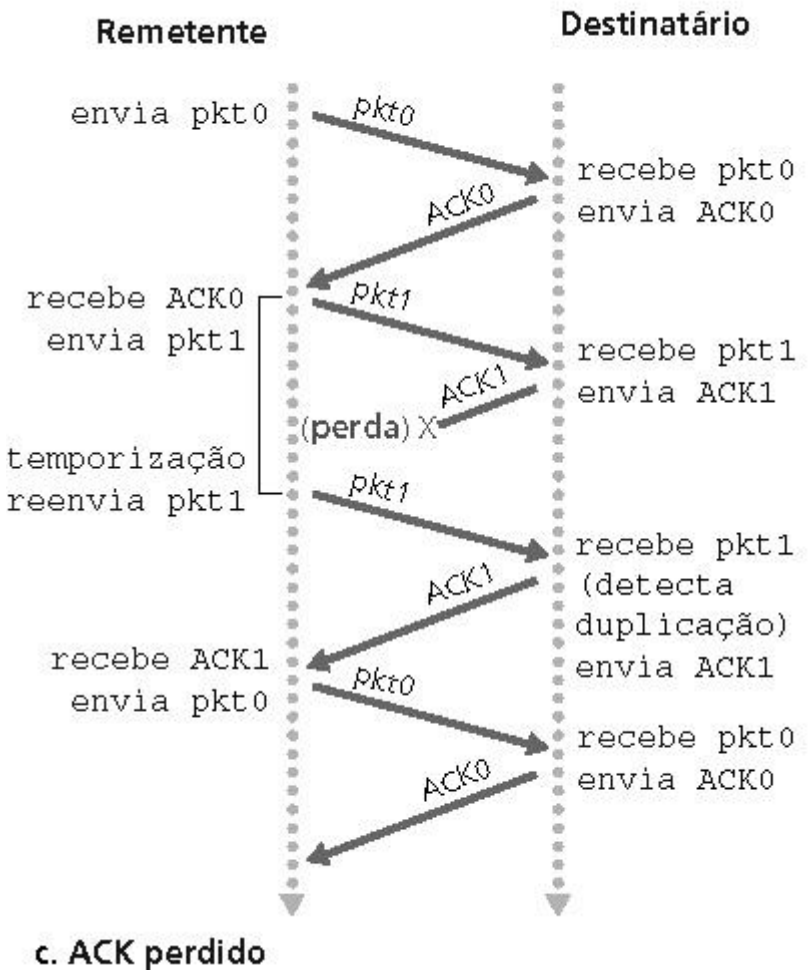
rdt3.0 remetente



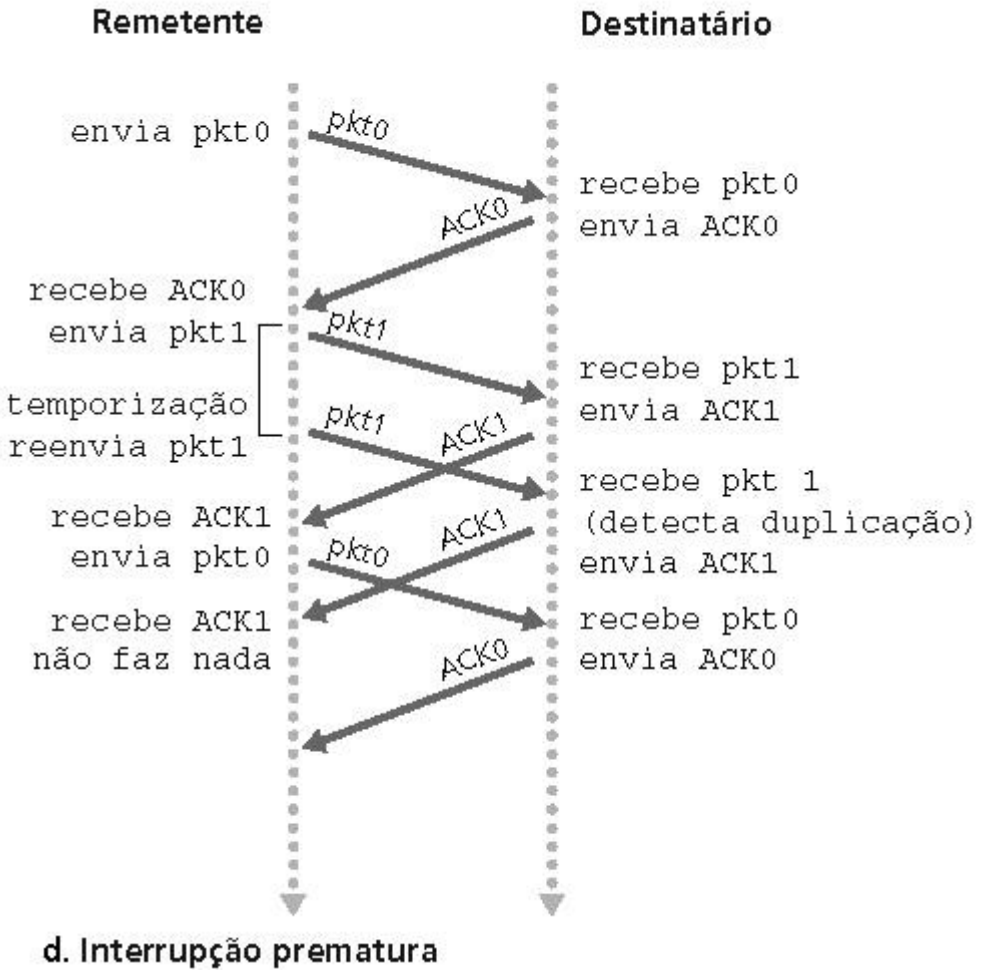
rdt3.0 em ação



rdt3.0 em ação



c. ACK perdido

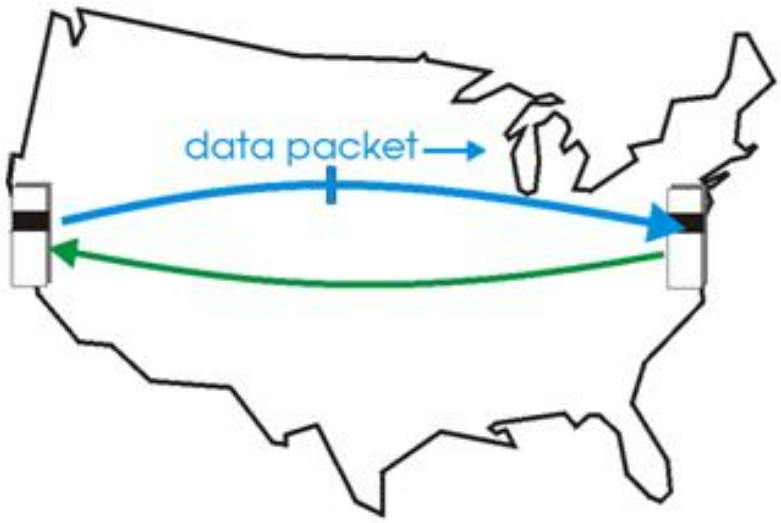


d. Interrupção prematura

Desempenho do rdt3.0

- O rdt3.0 funciona, porém seu desempenho pode ser muito ruim!
- **Exemplo:**

  Enlace costa-a-costa de 1 Gbps, t_{prop} = 15 ms, pacote de 1 kB



a stop-and-wait protocol in operation

Desempenho do rdt3.0

- **Exemplo:** (continuação do exemplo anterior)

U_{remet} = **Utilização do Remetente (ou do Canal)** = fração do tempo que o remetente fica ocupado transmitindo no canal

$U_{remet} = t_{trans}/t_T$ t_T = intervalo de tempo entre pacotes sucessivos

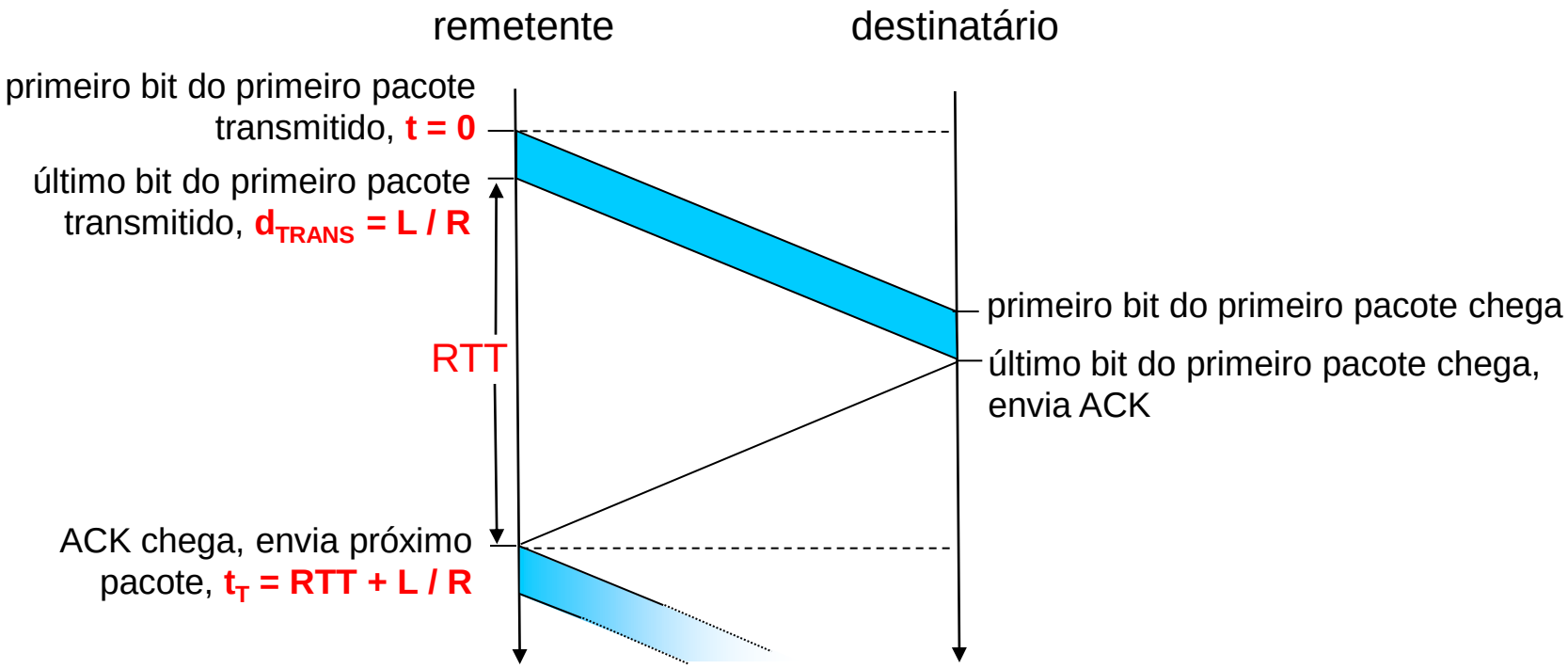
$$t_{trans} = \frac{L}{R} = \frac{8 \text{ kb}}{10^9 \text{ b/s}} = 0{,}008 \text{ ms} \qquad t_T = \frac{L}{R} + RTT$$

$$U_{remet} = t_{trans}/t_T = \frac{L/R}{L/R + RTT} = \frac{0{,}008 \text{ ms}}{30{,}008 \text{ ms}} = 0{,}00027$$

  - 1 pacote de 1 kB a cada 30,008 ms produz vazão de 267 kbps
  - protocolo "stop-and-wait" limita o uso dos recursos físicos!

Desempenho do rdt3.0

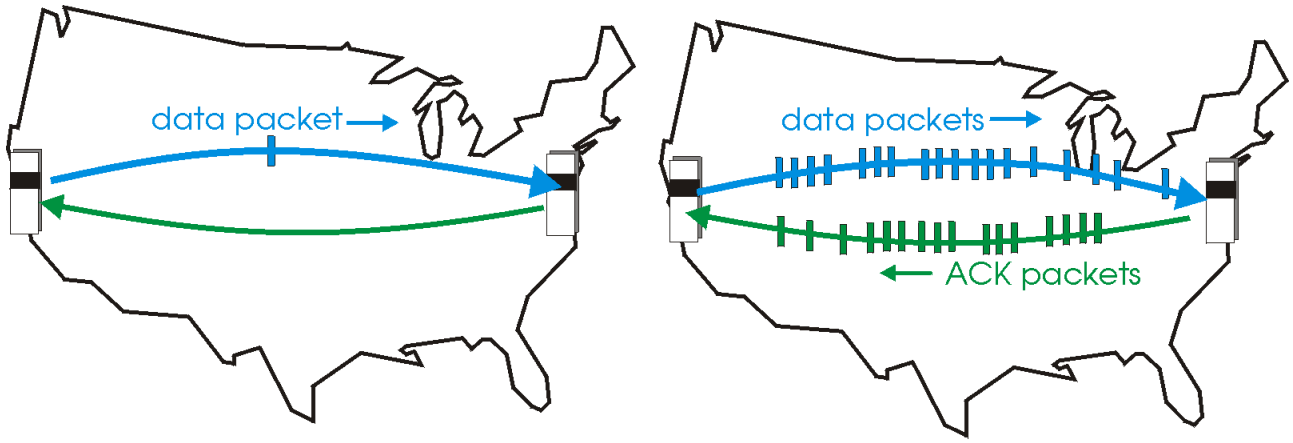
- **Exemplo:** (continuação do exemplo anterior):



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

Protocolos com *Pipelining*

- ***Pipelining*** : ao remetente é permitido múltiplos pacotes "em trânsito", ainda não reconhecidos
  - a faixa de números de sequência deve ser ampliada
  - requer aumento de capacidade dos buffers (remetente e receptor)



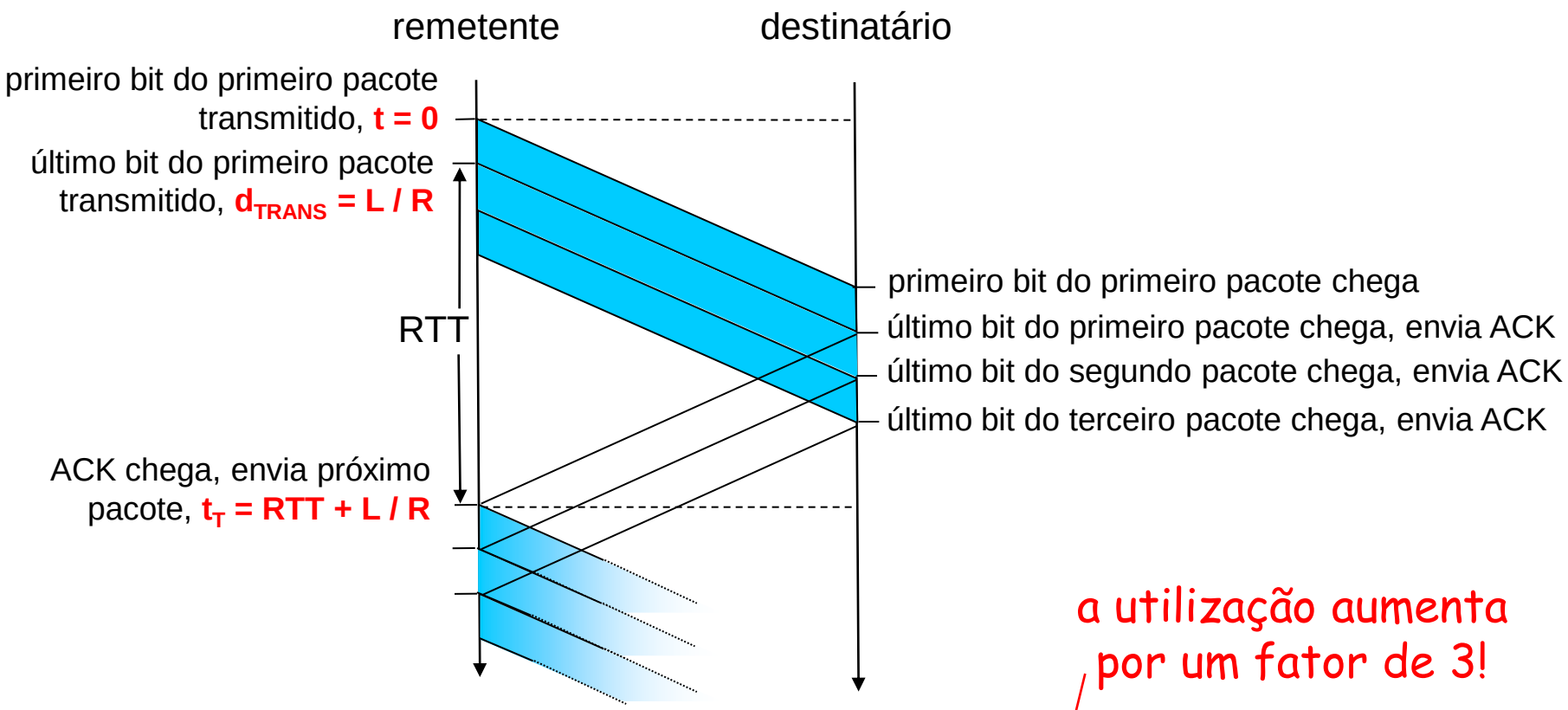
(a) a stop-and-wait protocol in operation

(b) a pipelined protocol in operation

- Duas formas genéricas de protocolos com *pipelining* :
  - ***Go-Back-N*** (retransmissão integral)
  - ***Selective Repeate*** (retransmissão seletiva)

Pipelining: aumento da utilização

- **Exemplo:** envio de três pacotes consecutivos



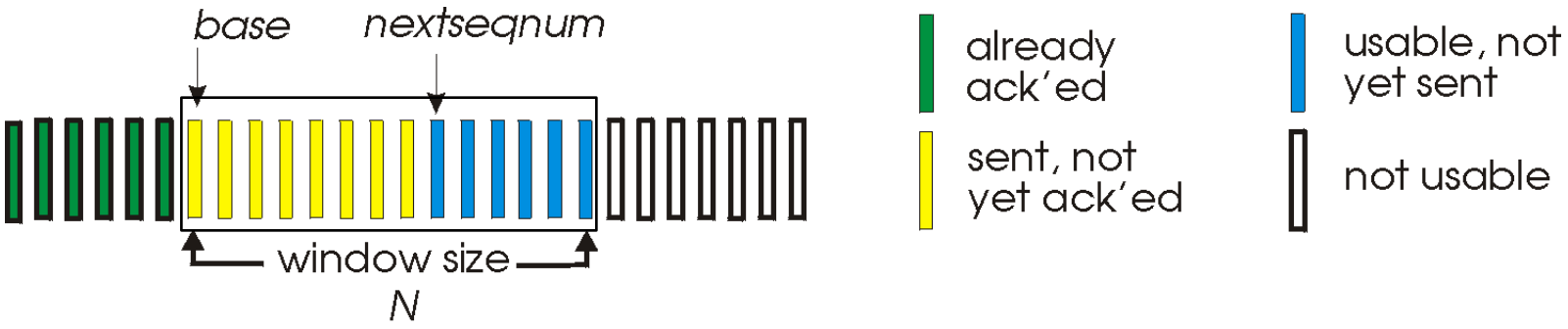
a utilização aumenta por um fator de 3!

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

Go-Back-N

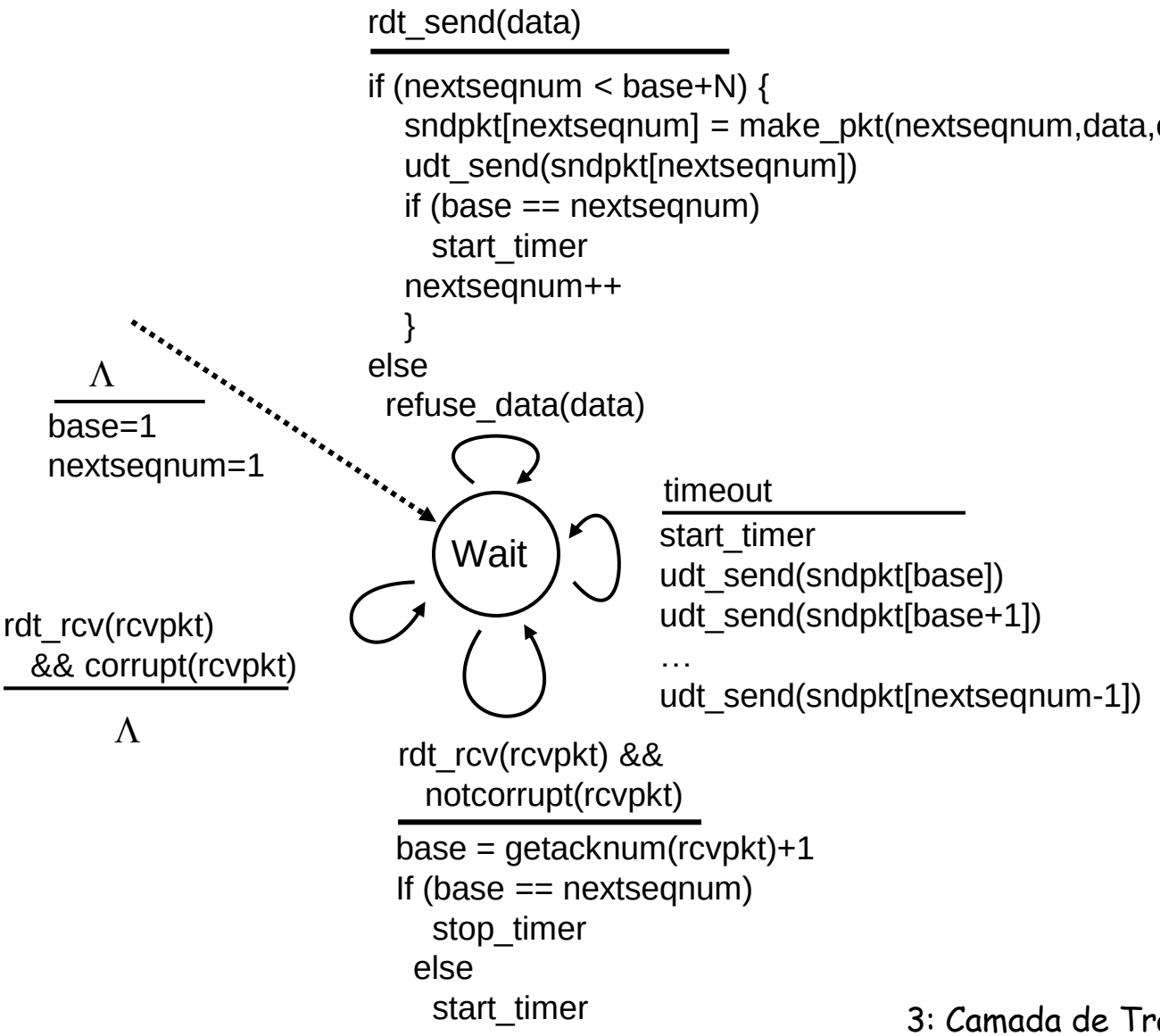
Remetente:

- pode transmitir múltiplos pacotes sem esperar por ACK, mas fica limitado em até N pacotes ("janela") consecutivos não reconhecidos

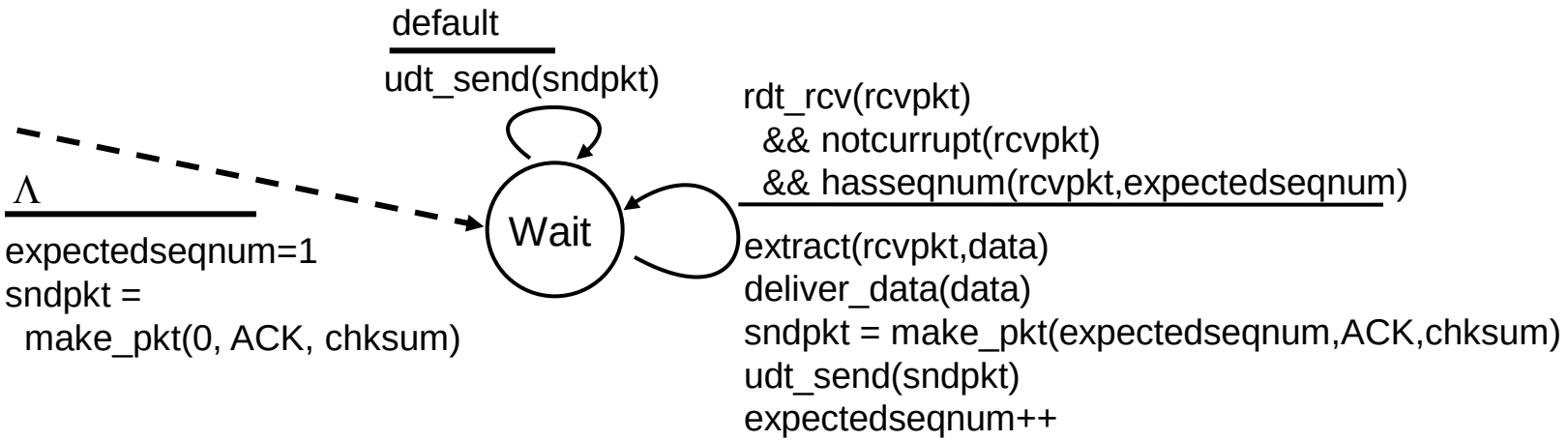


- protocolo *sliding-window* : "deslisa janela" após uma transmissão OK
- campo "número de sequência" no cabeçalho tem *m* bits -> faixa de números de sequência: [0, 2^m - 1]
- ACK cumulativo: um ACK de pacote com número de sequência *n* indica que todos os pacotes com número de sequência até e inclusive *n* foram corretamente recebidos
- "timeout" é usado para recuperar pacote ou ACK perdido

Go-Back-N: FSM estendida do remetente



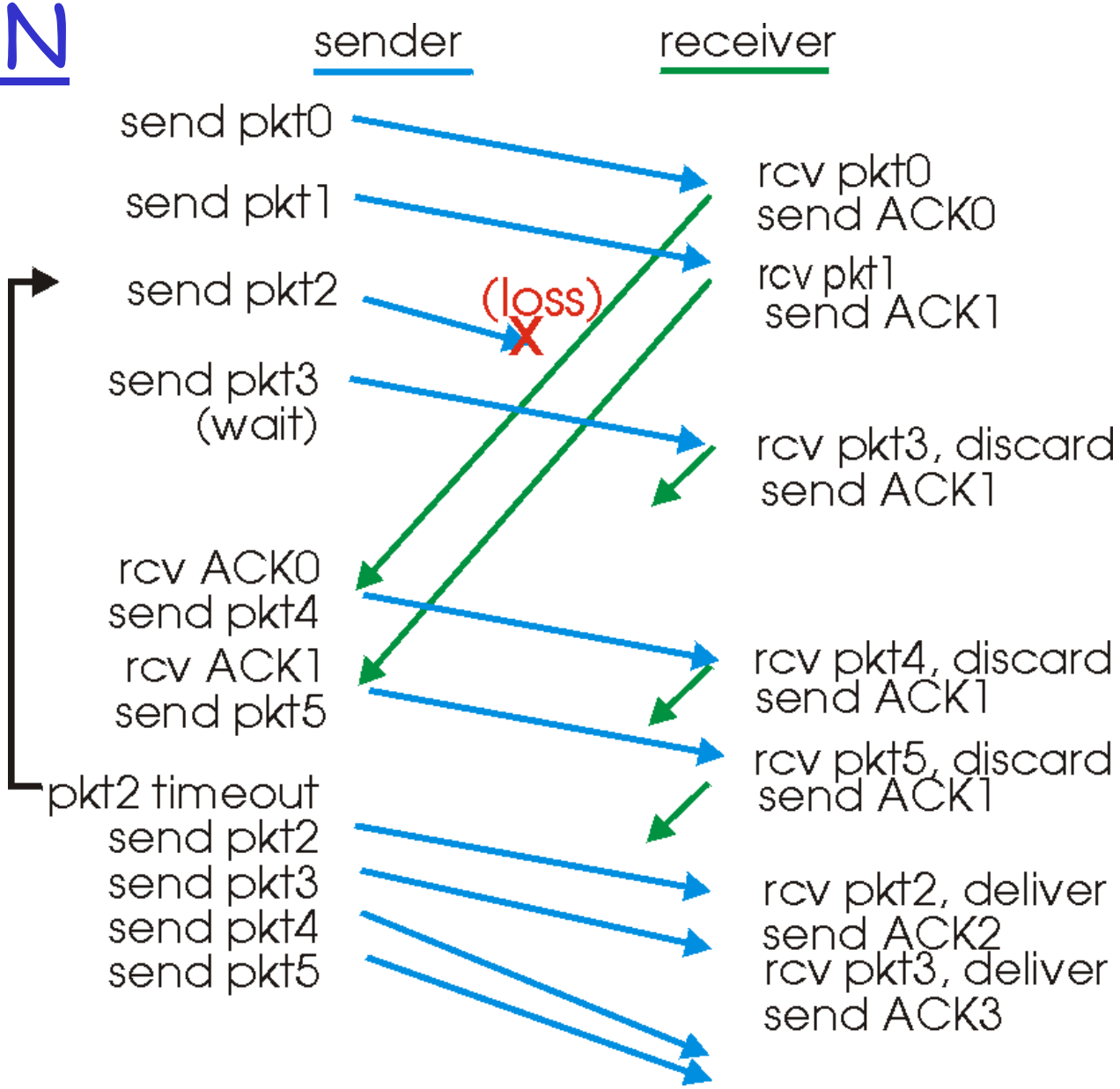
Go-Back-N: FSM estendida do destinatário



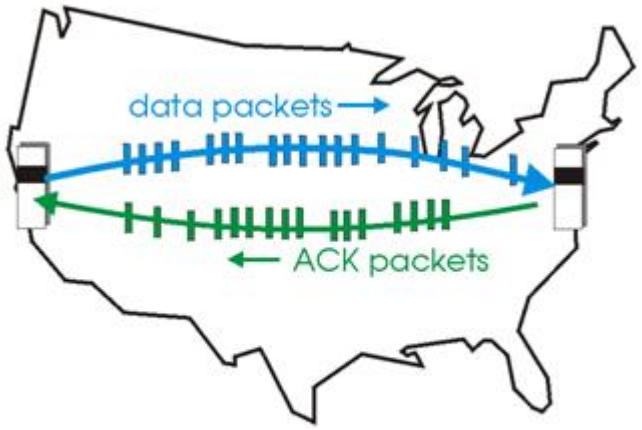
Destinatário:

- Envia ACK(n) para pacote(n) recebido livre de erros e com número de sequência de acordo com ordenamento
- Caso contrário (erro de bit e/ou fora de ordem): descarta pacote recebido (não armazena) e retransmite ACK(n-1)

Go-Back-N em ação



BDP (*Bandwidth-Delay Product*)



a pipelined protocol in operation

- **BDP**: **Produto largura de banda-atraso** é a medida do número de bits (ou bytes) que um remetente pode transmitir enquanto espera por um ACK do destinatário
- $BDP = R_{bps} \cdot (RTT)_s$ [bits]

Exemplo:

- Considere que precisamos projetar um protocolo *Go-Back-N* com janela deslizante para uma rede na qual a largura de banda é de 100 Mbps e a distância entre o emissor e o receptor é de 10.000 km. Considere que o tamanho médio dos pacotes é de 100.000 bits e a velocidade de propagação no meio físico de $2x10^8$ m/s. Determine: (a) O tamanho máximo das janelas de transmissão e de recepção. (b) O número de bits no campo de número de sequência (*m*). (c) Um valor de tempo limite inferior para o timeout.

- Solução:

a) Tamanho máximo das janelas de transmissão e de recepção:

RTT médio = 2x(10.000 km) / ($2x10^8$ m/s) = 100 ms

BDP em bits = 100 Mbps x 100 ms = 10.000.000 bits

BDP em pacotes = 10.000.000 / 100.000 = 100 pacotes

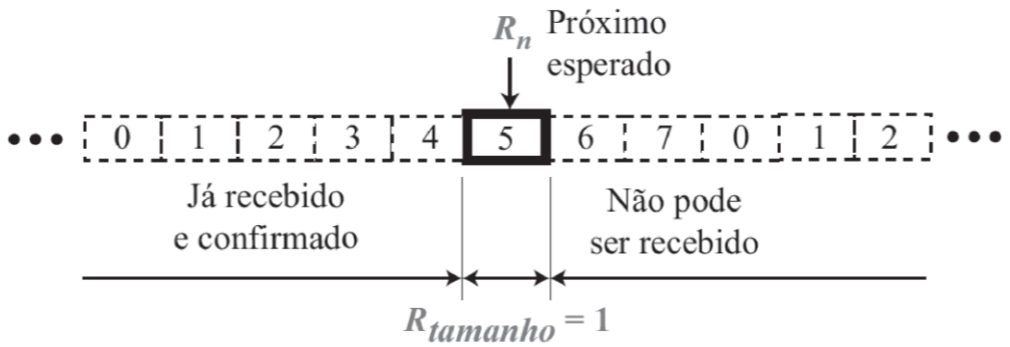
Tamanho máximo da janela de transmissão = 100 pacotes

No Go-Back-N, o tamanho da janela de recepção é sempre igual a 1 pacote.

Exemplo:

- Solução (Continuação):

  Janela de recepção:



b) Número de bits no campo de número de sequência (*m*):

(tamanho da janela de transmissão) < 2^m

$100 < 2^m$

Assim, tem-se que *m* = 7 (números de sequência de 0 a 127)

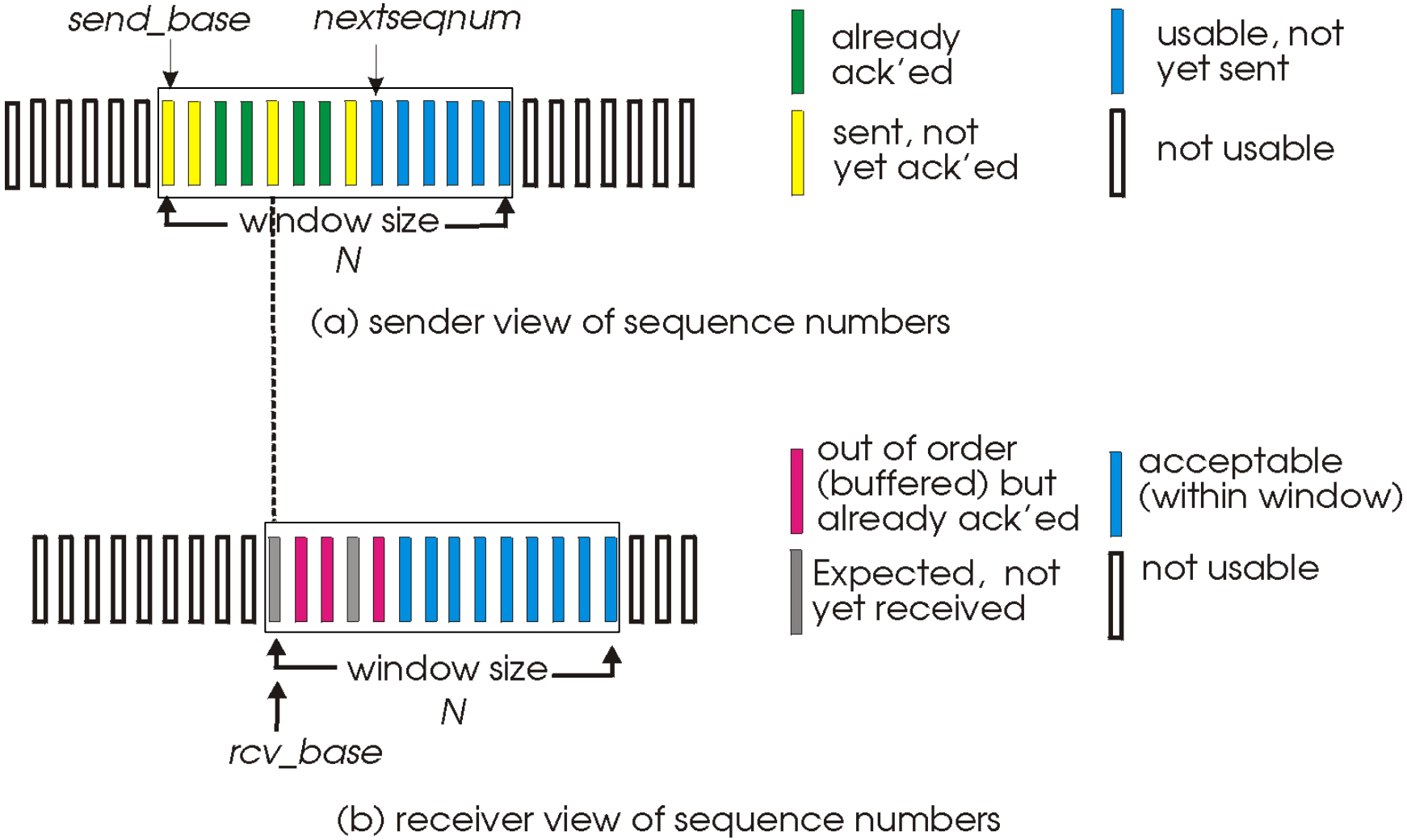
c) Valor de tempo limite inferior para o timeout:

O timeout deverá ser, no mínimo, igual ao RTT = 100 ms

Retransmissão seletiva (*Selective Repeate*)

- Se o tamanho da janela (N) e o BDP (*Bandwidth-Delay Product*) são grandes -> Go-Back-N apresenta baixo desempenho
- Solução: receptor reconhece individualmente todos os pacotes recebidos corretamente
  - armazena os pacotes recebidos fora de ordem no buffer de recepção para posterior entrega, em ordem, à camada superior
- O remetente retransmite somente os pacotes para os quais o ACK não foi recebido
  - dispara temporizador ("timeout") para cada pacote transmitido
- Usa janelas de tamanho N para transmissão e recepção
  - limita o número de pacotes pendentes não reconhecidos dentro da rede

Retransmissão seletiva: janelas do remetente e do receptor



Retransmissão seletiva: eventos e ações

Remetente

dados recebidos de cima:

- se o próximo nº de seq. está dentro da janela, cria e envia o pacote.

ocorre *timeout* (n):

- retransmite o pacote n e reinicia o temporizador

ACK(n) recebido no [send_base, send_base+(N-1)]:

- marca o pacote n como "recebido"
- se n for igual a send_base, avança base da janela até o pacote não reconhecido que tiver o menor nº de seq.

Receptor

pacote n recebido OK com nº de seq. no [rcv_base, rcv_base+(N-1)]

- envia ACK(n)
- se fora de ordem: buffer
- em ordem: entrega (inclui pacotes no buffer) e avança janela p/ próximo pacote ainda não recebido"

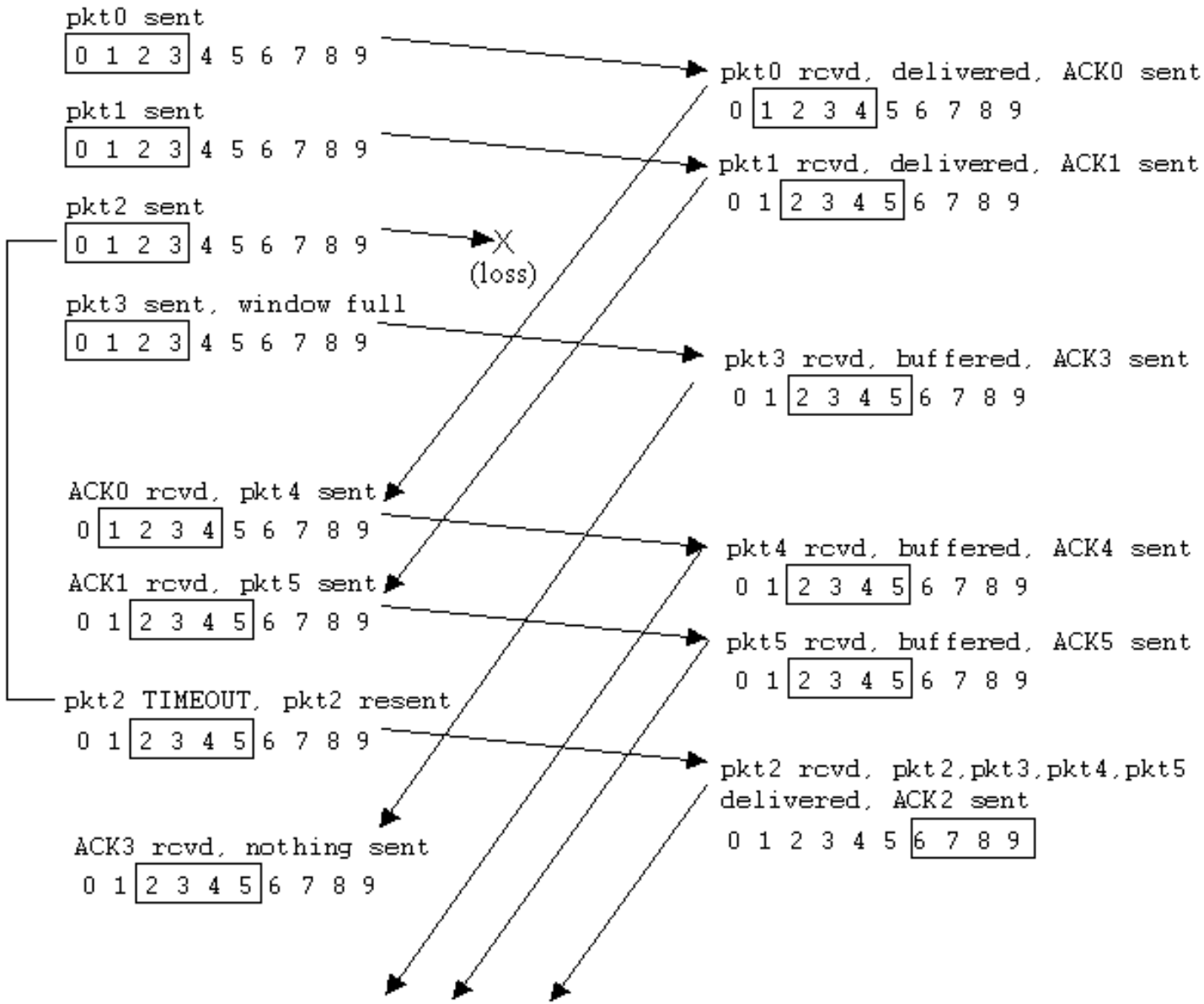
pacote n recebido OK no [rcv_base-N, rcv_base-1]

- um ACK(n) deve ser gerado

qualquer outro :

- ignora o pacote

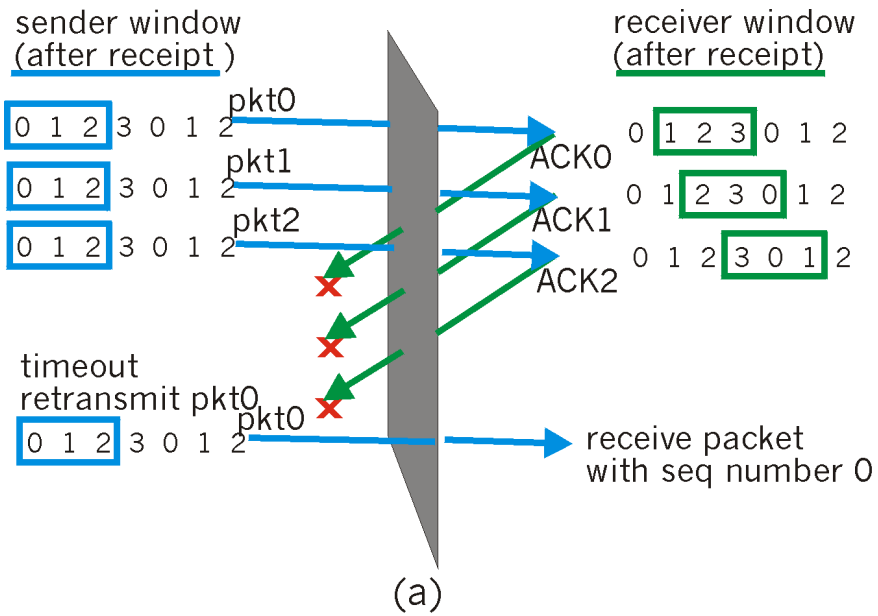
Retransmissão seletiva em ação



Retransmissão seletiva: dilema

Exemplo:

- Números de sequência: 0, 1, 2, 3
- Tamanho da janela = 3



- Receptor não "vê" diferença entre os dois cenários!
- Em (a), incorretamente passa-se dados duplicados como novos

P: Qual a relação entre os números de sequência e o tamanho da janela?