Capítulo 3: Camada de Transporte

Objetivos do Capítulo:

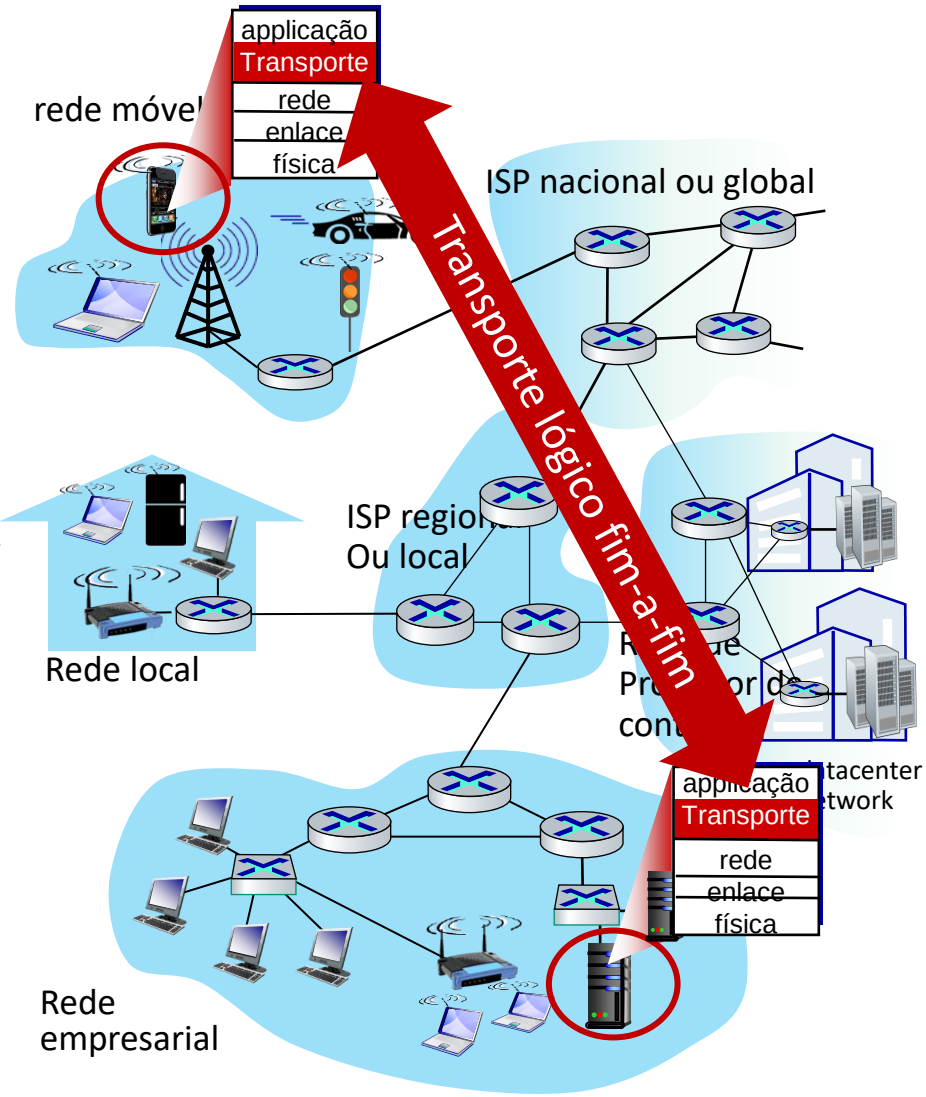
- entender os serviços da camada de transporte:
  - multiplexação/demultiplexação
  - transferência confiável de dados
  - controle de fluxo
  - controle de congestionamento
- instanciação e implementação destes princípios na Internet
- Aprender sobre os protocolos da camada de transporte da Internet:
  - UDP: transporte sem conexão
  - TCP: transporte orientado a conexão e confiável
  - TCP: controle de congestionamento

Resumo do Capítulo:

- ✓ serviços da camada de transporte
- ✓ multiplexação/demultiplexação
- ✓ transporte sem conexão: UDP
- ✓ princípios de transferência confiável de dados
- ✓ transporte orientado a conexões: **TCP**
  - ✓ transferência confiável
  - ✓ controle de fluxo
  - ✓ gerenciamento de conexão
- ✓ princípios de controle de congestionamento
- ✓ controle de congestionamento do TCP
- ✓ Evolução das funcionalidades da camada de transporte

Serviços e protocolos de transporte

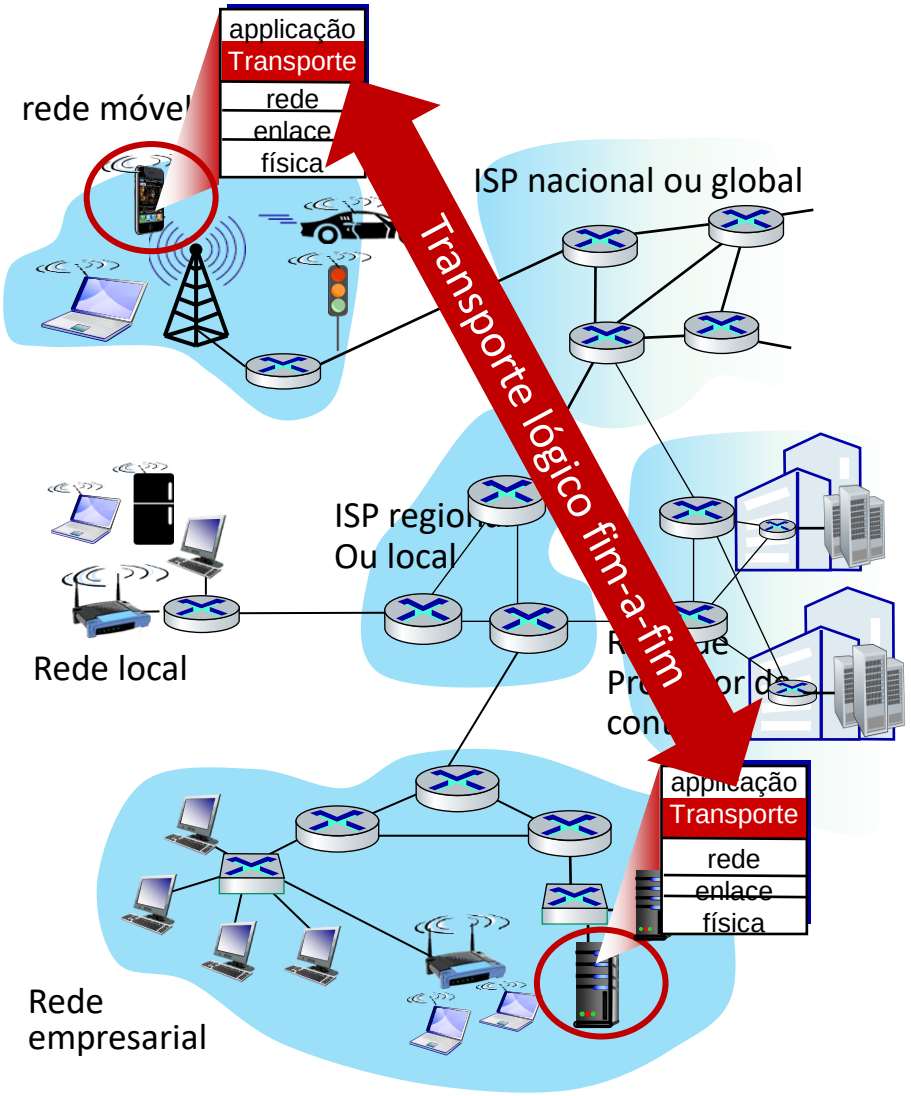
- provê *comunicação lógica* entre processos de aplicação executando em *hosts* diferentes
- protocolos de transporte são executados em sistemas finais (*end systems*)
  - Lado TX: quebra msgs da camada de aplic. em ***segmentos***, passando-os para a camada de rede
  - Lado Rx: remonta os segmentos em msgs, passando-as para a camada de aplicação
- dois protocolos de transporte disponíveis para as aplicações da Internet: UDP e TCP



Protocolos e serviços: camada de transporte vs camada de rede

Serviços da camada de transporte versus serviços da camada de rede:

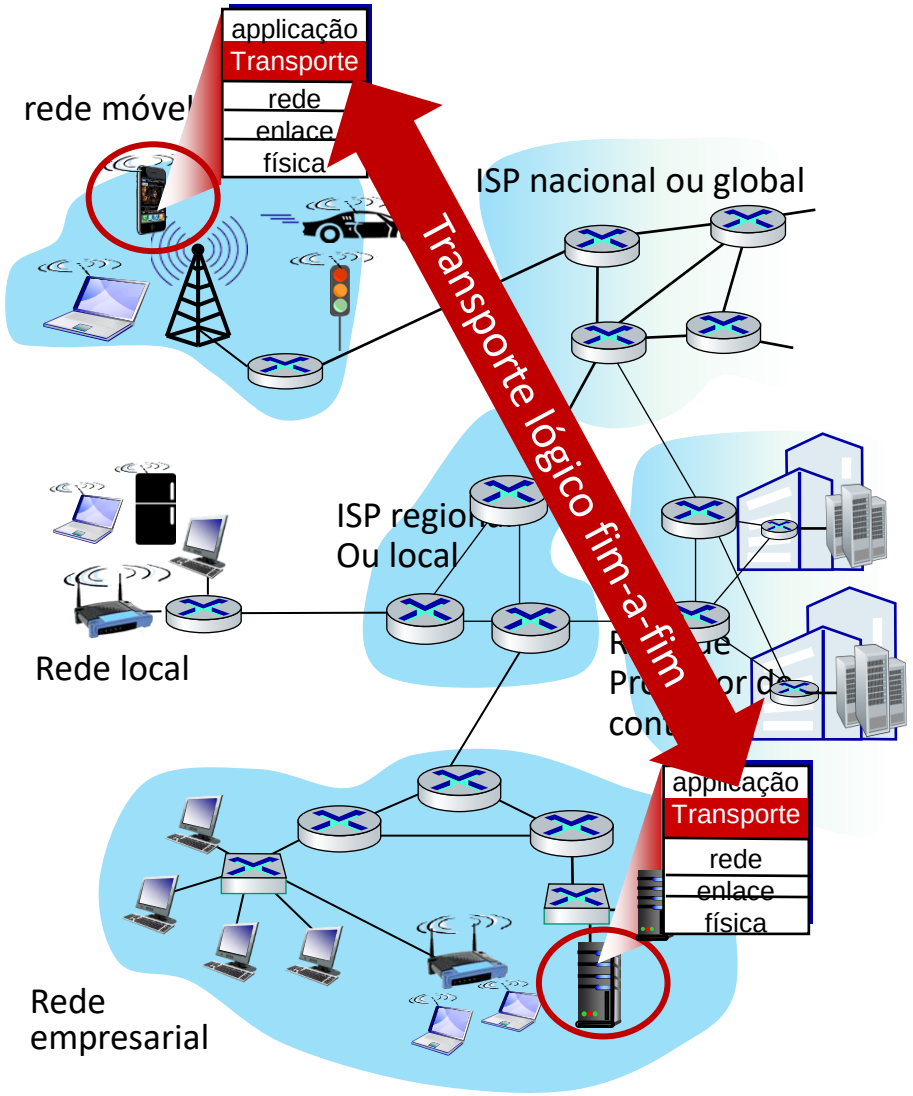
- *camada de rede:* comunicação lógica entre ***hosts*** (dados são transferidos entre hosts)
- *camada de transporte:* comunicação lógica entre processos (dados são transferidos entre processos)
  - Utiliza e aprimora os serviços oferecidos pela camada de rede



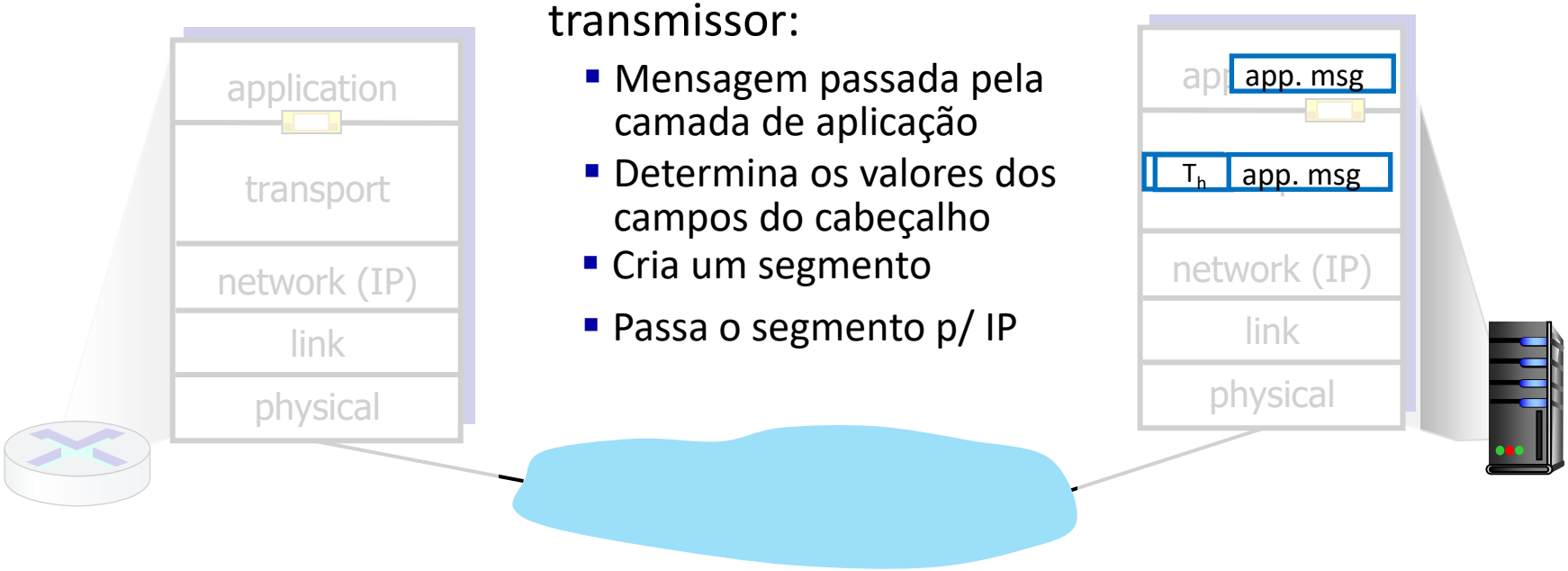
Camada de transporte vs Camada de rede

Analogia: 12 pessoas numa casa enviando cartas para 12 pessoas em outra casa

- Hosts – casas
- Processos – pessoas
- Msgs da aplicação - cartas nos envelopes
- Segmentos da camada de transporte – envelopes com cartas
- Protocolo da Camada de rede: serviço postal
- Protocolo da Camada de transporte: donos das casas que distribuem as cartas (sem envelope) às pessoas



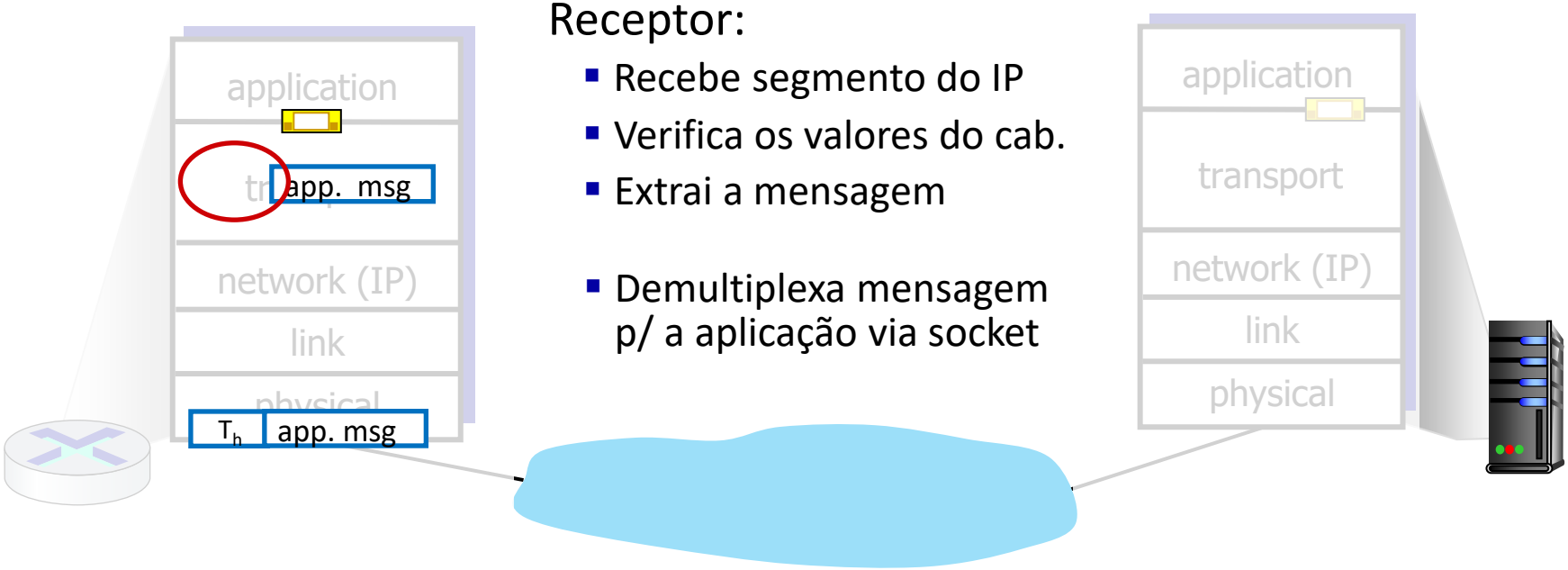
Camada de transporte: Ações



transmissor:

- Mensagem passada pela camada de aplicação
- Determina os valores dos campos do cabeçalho
- Cria um segmento
- Passa o segmento p/ IP

Camada de transporte: Ações

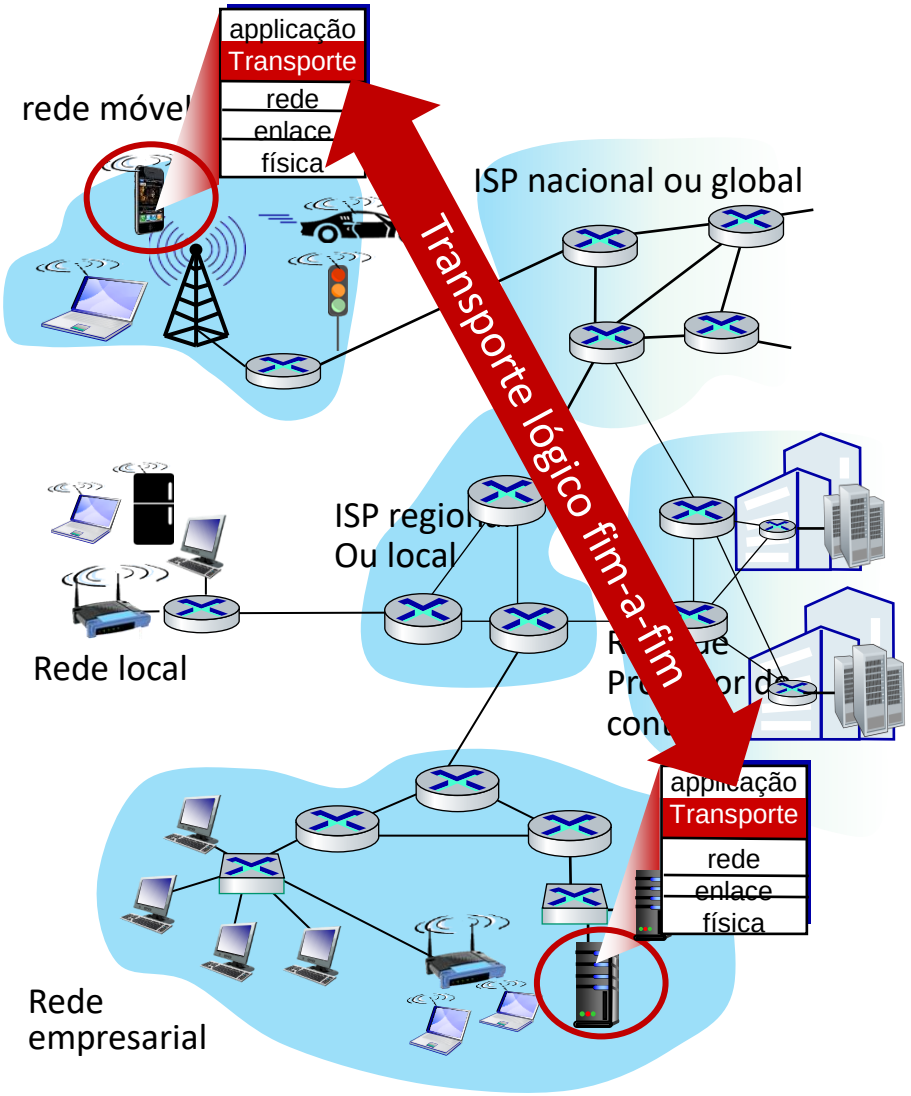


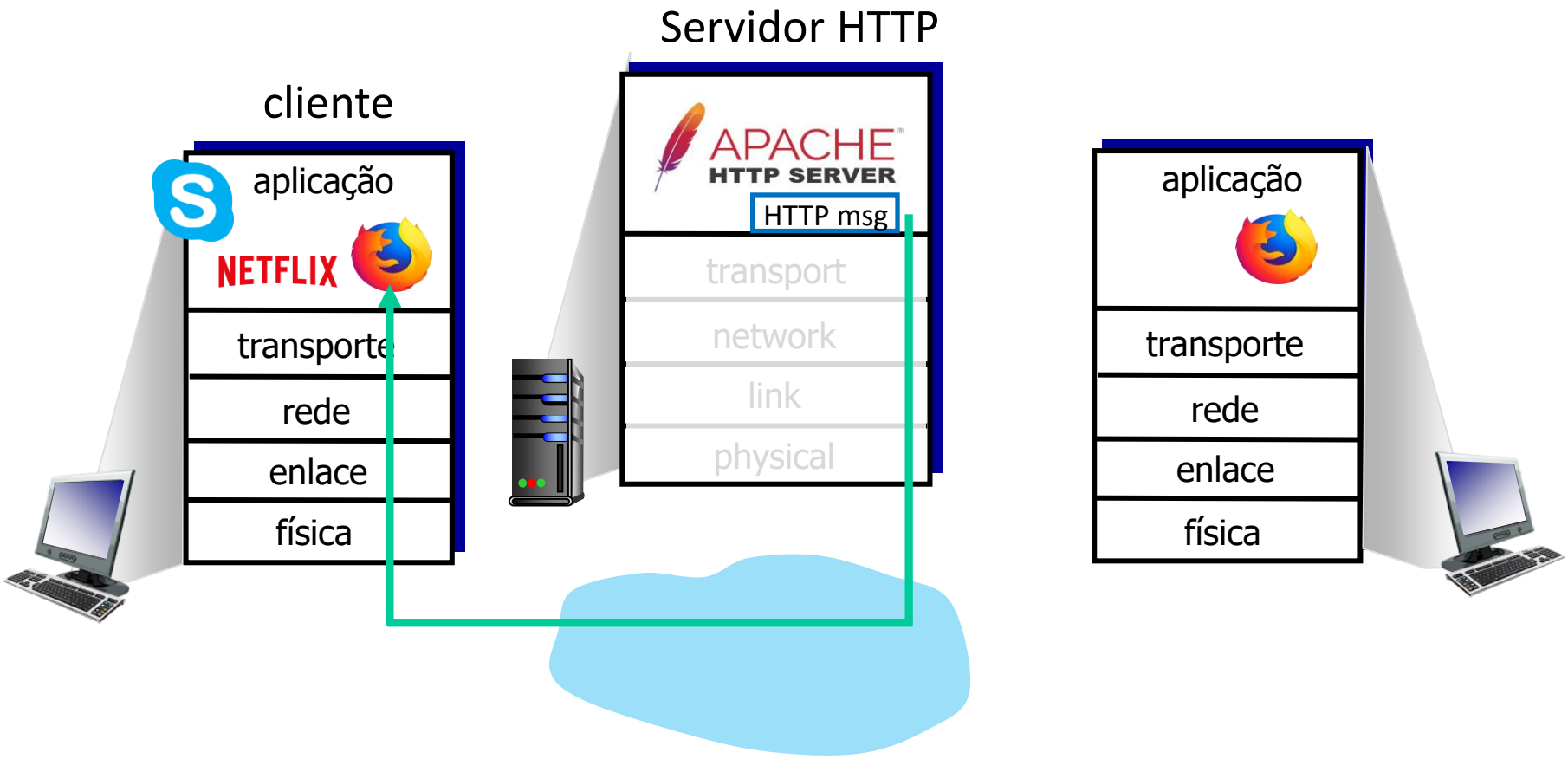
Receptor:

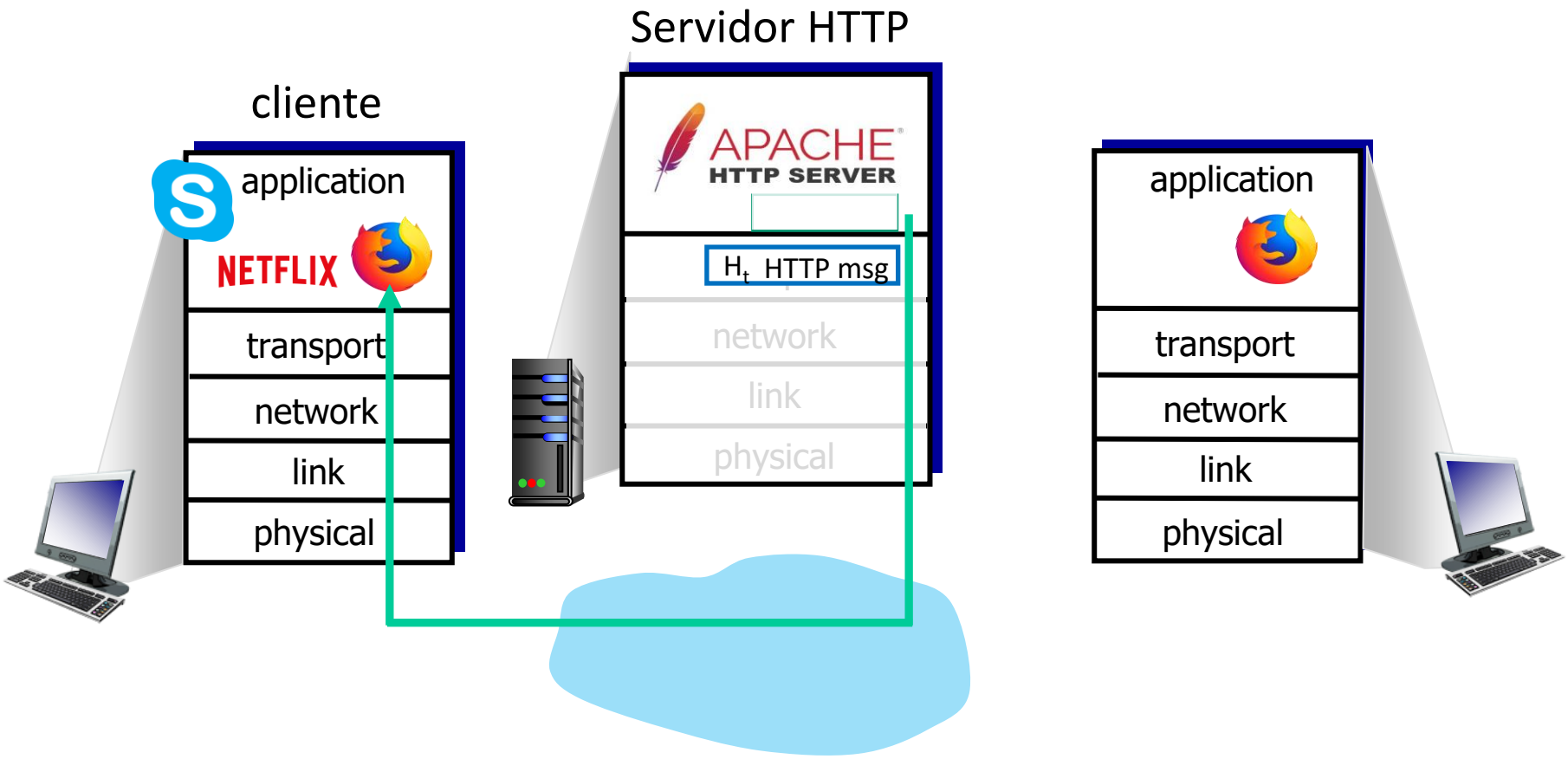
- Recebe segmento do IP
- Verifica os valores do cab.
- Extrai a mensagem
- Demultiplexa mensagem p/ a aplicação via socket

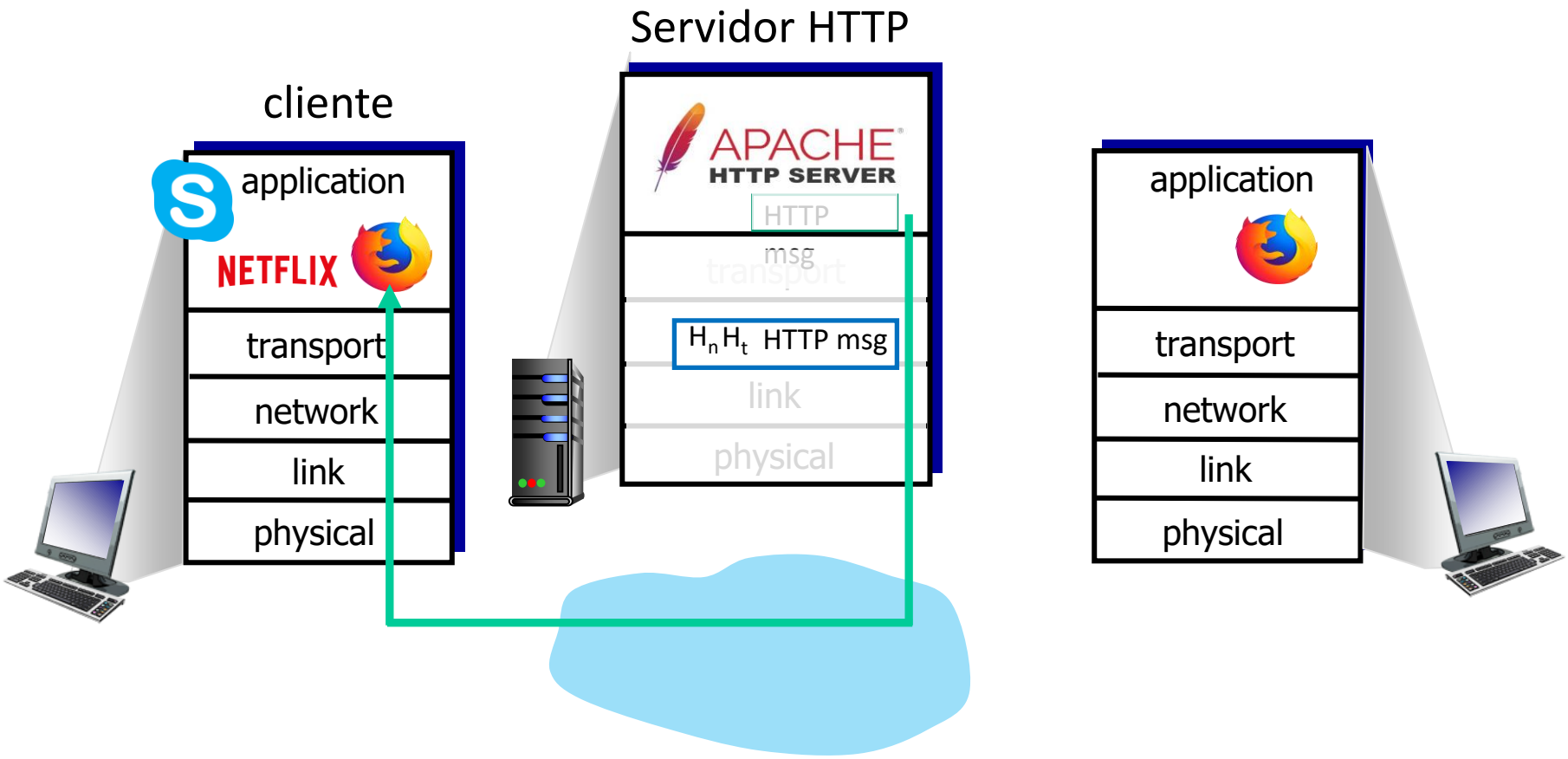
Protocolos da camada de transporte da Internet

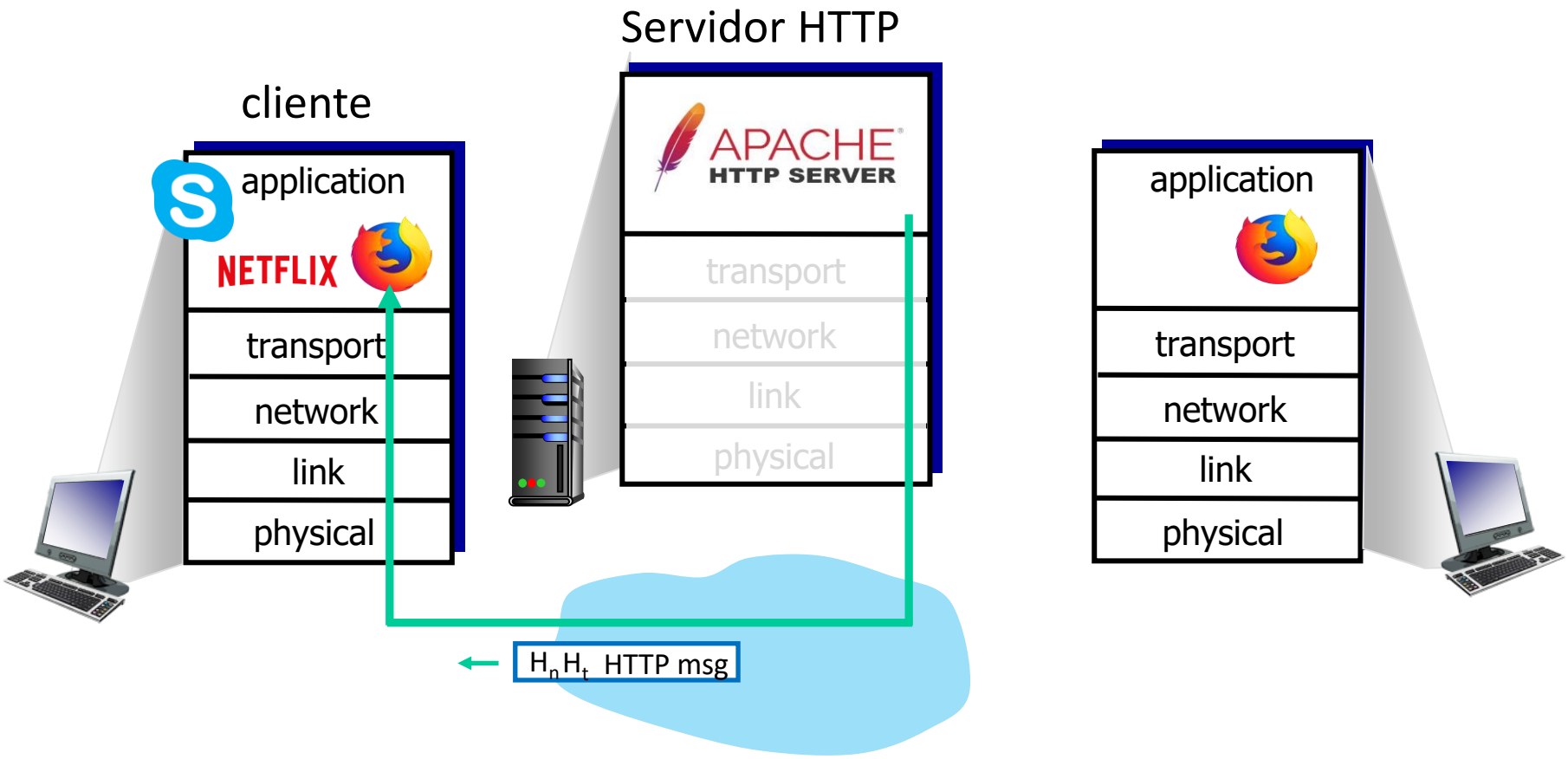
- TCP (Transmission Control Protocol):
  - entrega confiável, ordenada, ponto a ponto
  - controle de congestionamento
  - controle de fluxo
  - orientado a conexões
- UDP (User Datagram Protocol):
  - entrega não confiável, não seqüencial
  - Extensão do serviço de "melhor esforço" do IP
- serviços não disponíveis (para ambos):
  - garantias de atraso
  - garantias de banda

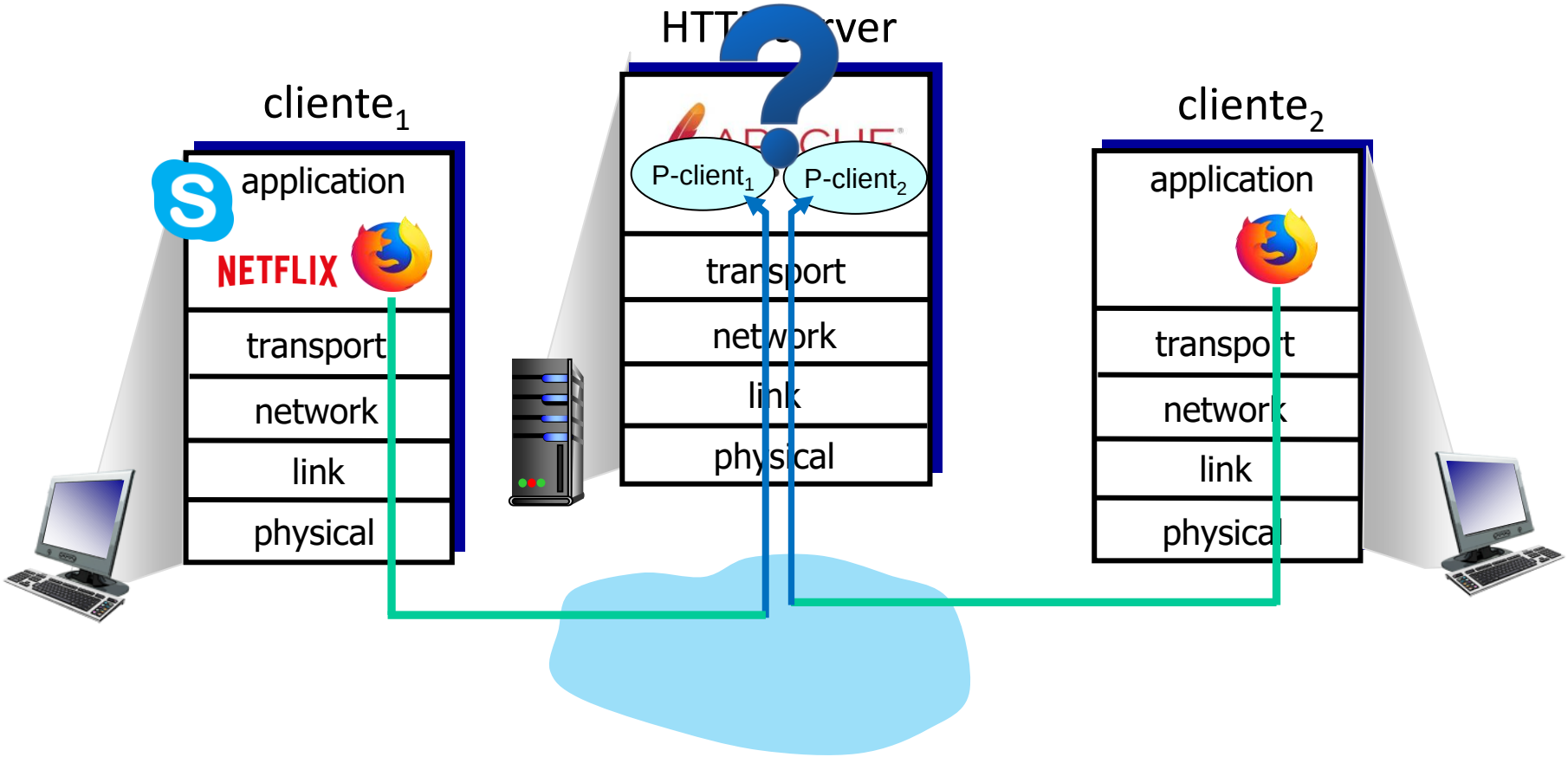



Servidor HTTP
cliente
aplicação
NETFLIX
transporte
rede
enlace
física
APACHE
HTTP SERVER
HTTP msg
transport
network
link
physical
aplicação
transporte
rede
enlace
física


Servidor HTTP
cliente
application
NETFLIX
transport
network
link
physical
APACHE
HTTP SERVER
H_t HTTP msg
network
link
physical
application
transport
network
link
physical


Servidor HTTP
cliente
application
NETFLIX
transport
network
link
physical
APACHE
HTTP SERVER
HTTP
msg
transport
$H_n H_t$ HTTP msg
link
physical
application
transport
network
link
physical


Servidor HTTP
cliente
application
NETFLIX
transport
network
link
physical
APACHE
HTTP SERVER
transport
network
link
physical
application
transport
network
link
physical
$H_n H_t$ HTTP msg


cliente1
application
NETFLIX
transport
network
link
physical
HTTP server
P-client1
P-client2
transport
network
link
physical
cliente2
application
transport
network
link
physical

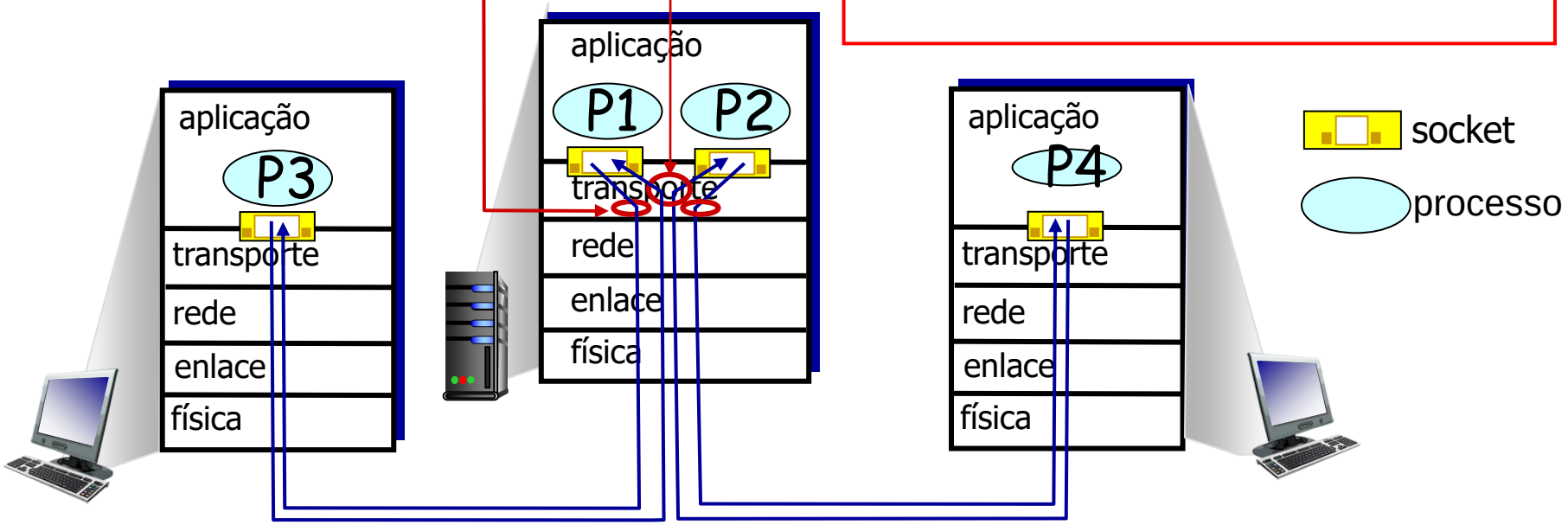
Multiplexação/demultiplexação

Multiplexação no transmissor:

reúne dados de muitos sockets, adiciona o cabeçalho de transporte (usado posteriormente para a demultiplexação)

Demultiplexação no receptor:

Usa info do cabeçalho para entregar os segmentos recebidos aos sockets corretos

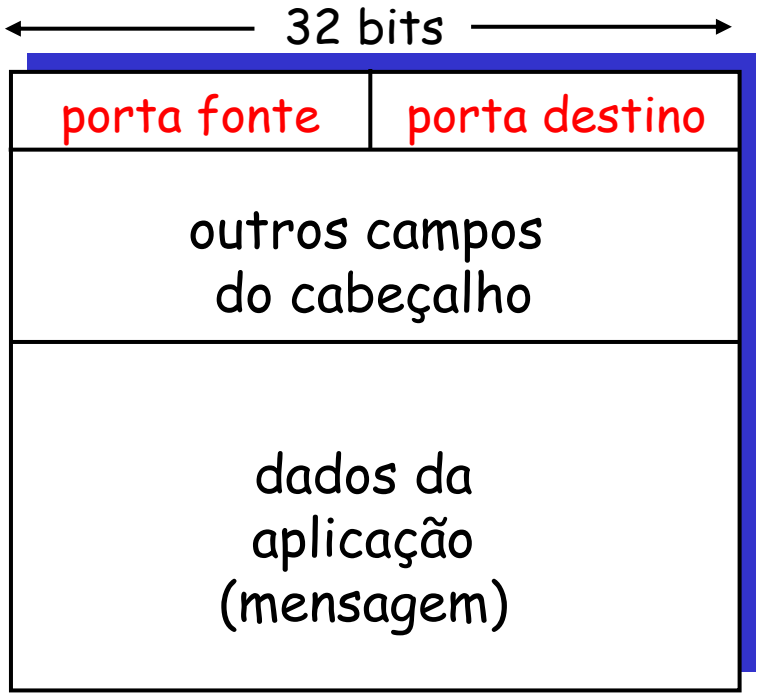


Como a demultiplexação funciona

Demultiplexação: entrega de segmentos recebidos para os processos corretos da camada de aplicação

host recebe datagramas IP

- cada datagrama tem o endereço IP da fonte e do destino
- cada datagrama carrega um segmento da camada de transporte
- cada segmento possui o nº da porta da fonte e o nº da porta destino



Formato do segmento TCP/UDP

- host usa o endereço *IP e os números das portas* para direcionar o segmento para o socket apropriado

Demultiplexação s/ conexão

Socket criado tem um número de porta local ao host:

DatagramSocket mySocket1 = new DatagramSocket(12534);

Qdo um datagrama é criado para ser enviado p/ socket UDP, deve especificado:

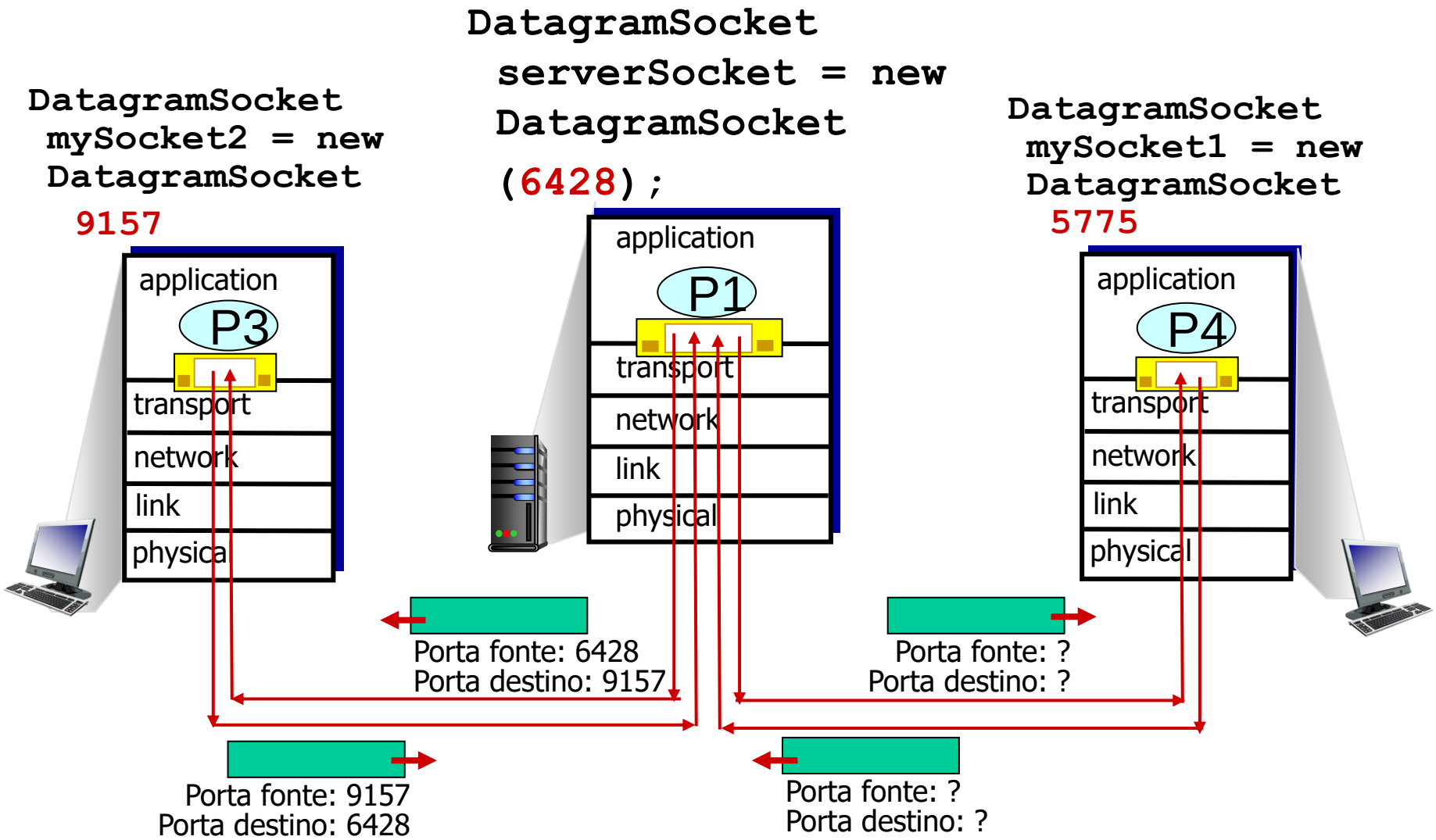
- *destination IP address*
- *destination port #*

Qdo host recebe segmento UDP:

- checa nº da porta de destino no segment
- direciona o segmento UDP para o socket com aquele número de porta

Datagramas IP/UDP com o *mesmo nº de porta de destino,* mas diferentes endereços IP (de origem) e/ou nº de porta fontes serão direcionados para o ***mesmo socket no destino***

Demultiplexação não orientada a conexão: exemplo



Demultiplexação orientada a conexão

Socket TCP identificado pela quádrupla:

- endereço IP origem
- número da porta origem
- endereço IP destino
- número da porta destino

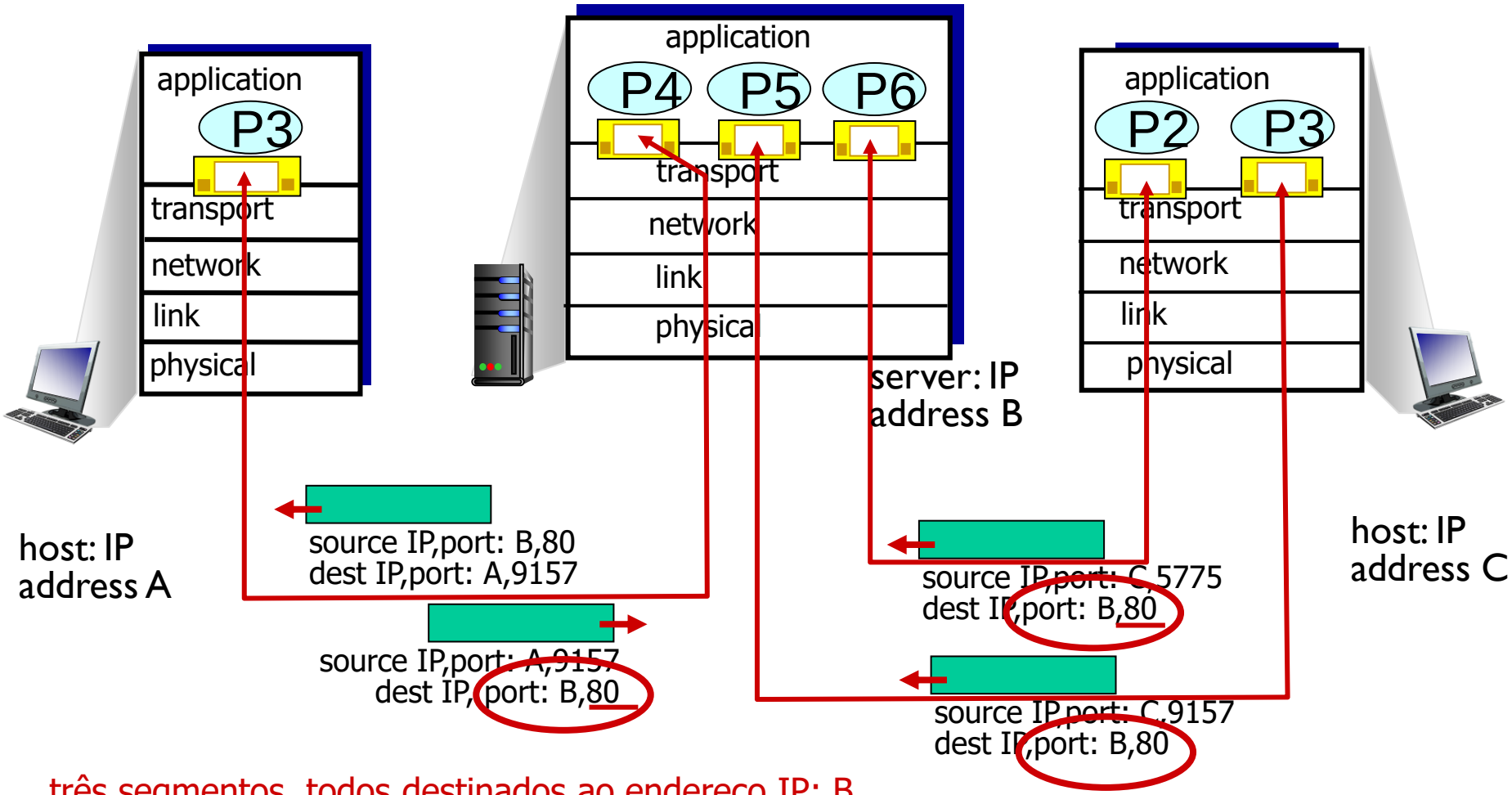
Demultiplexação: receptor usa todos os quatro valores para direcionar o segmento para o socket apropriado

Servidor pode dar suporte a muitos sockets TCP simultâneos:

- cada socket é identificado pela sua própria quádrupla
- cada socket está associado com um cliente (conexão diferente)

Ex: HTTP não persistente terá sockets diferentes para cada pedido

Demultiplexação Orientada a Conexões: exemplo



três segmentos, todos destinados ao endereço IP: B, dest port: 80 são demultiplexados para *sockets* distintos

Resumo

- Multiplexação, demultiplexação: baseada nos valores dos cabeçalhos do segmento e do datagrama
- **UDP:** demultiplexação usando somente nº da porta de destino
- **TCP:** demultiplexação usando endereços IP da origem e do destino e nº das portas
- Multiplexção/demultiplexação acontecem em todas as camadas

Protocolo de Transporte não orientado para conexão: UDP

- O UDP, definido no [RFC 768], faz apenas quase tão pouco quanto um protocolo de transporte pode fazer.
- A não ser sua função de multiplexação/demultiplexação e de alguma verificação de erros simples, ele nada adiciona ao IP.
- Se o desenvolvedor de aplicação escolher o UDP, em vez do TCP, a aplicação estará "falando" quase diretamente com o IP.
- O UDP é *não orientado para conexão*.

UDP: User Datagram Protocol [RFC 768]

- Protocolo de transporte da Internet "sem gorduras", "sem frescuras"
- Serviço "*melhor esforço*"; segmentos UDP podem ser:
  - perdidos
  - entregues fora de ordem para a aplicação
- *Sem conexão:*
  - não há apresentação entre o UDP transmissor e o receptor
  - cada segmento UDP é tratado de forma independente dos outros

UDP: User Datagram Protocol [RFC 768]

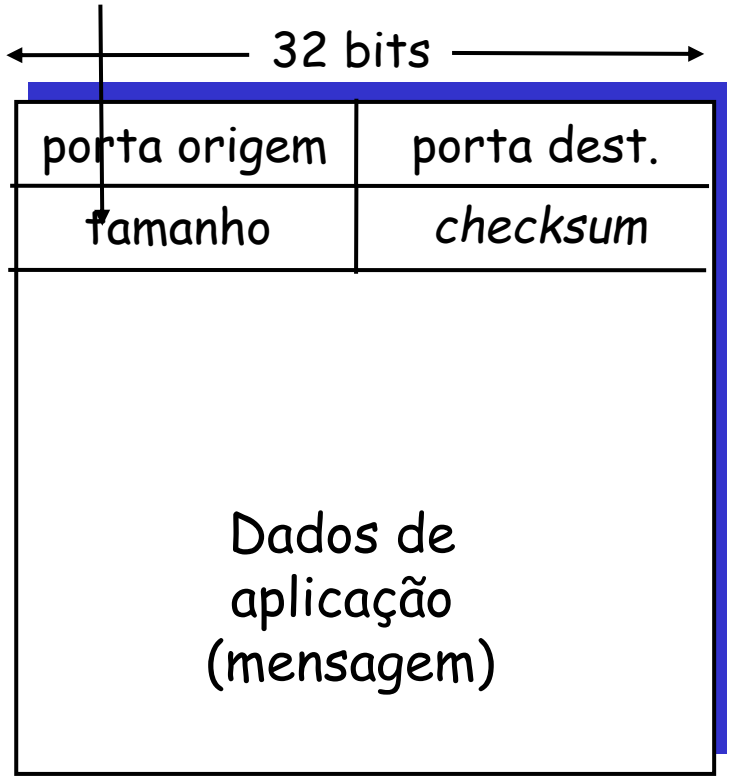
Por que UDP?

- não existe a fase de estabelecimento de conexão (o que pode causar retardo)
- simples: não se mantém o **estado** da conexão, nem no transmissor, nem no receptor
- cabeçalho de ***segmento*** reduzido → **baixo *overhead***
- não existe controle de congestionamento: **UDP pode enviar segmentos tão rápido quanto desejado (e possível)**

UDP: User Datagram Protocol [RFC 768]

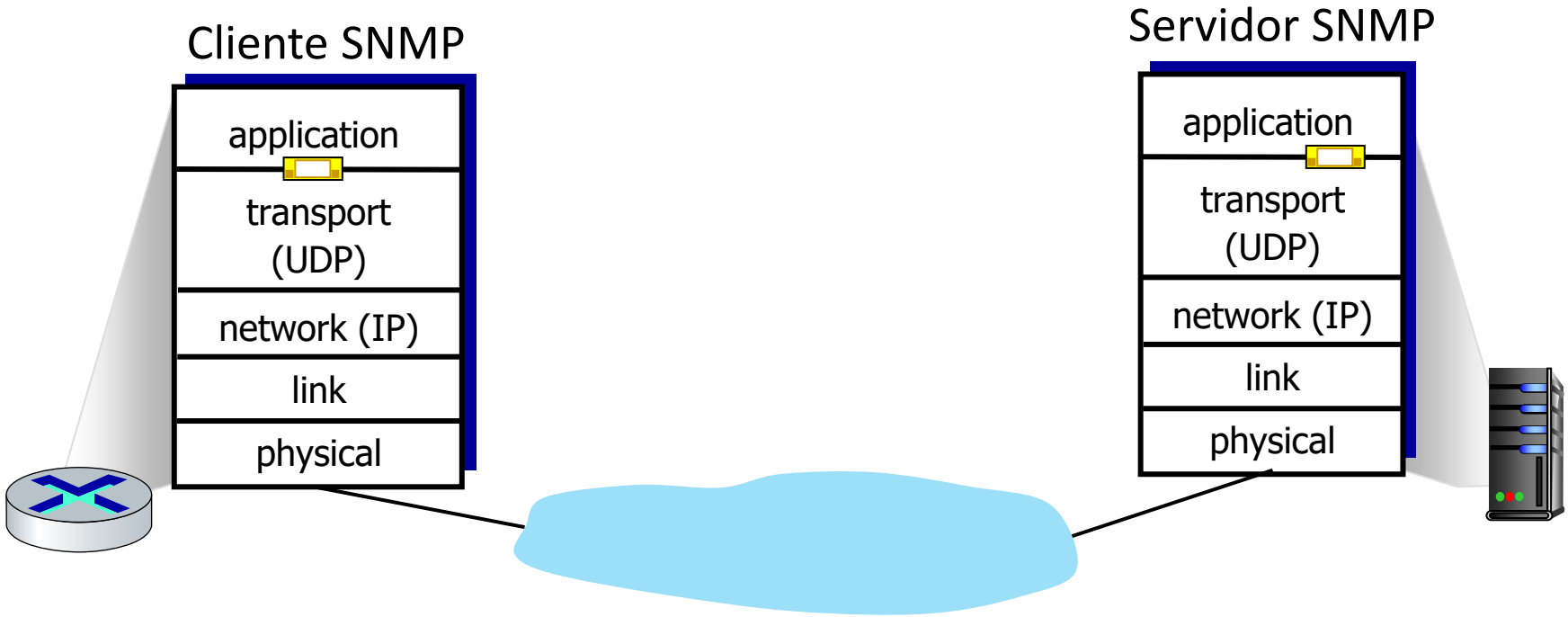
- muito utilizado para apls. de mídias contínuas (voz, vídeo)
  - mais tolerantes a perdas
  - sensíveis à taxa de transmissão
- outros usos de UDP:
  - DNS (resolução de nomes)
  - SNMP (gerenciamento)
  - HTTP/3
- Se transferência confiável sobre UDP for necessária (HTTP/3: acrescentar confiabilidade (recup. de erros) e controle de congestionamento na camada de aplicação)

tamanho em bytes do segmento UDP, incluindo cabeçalho

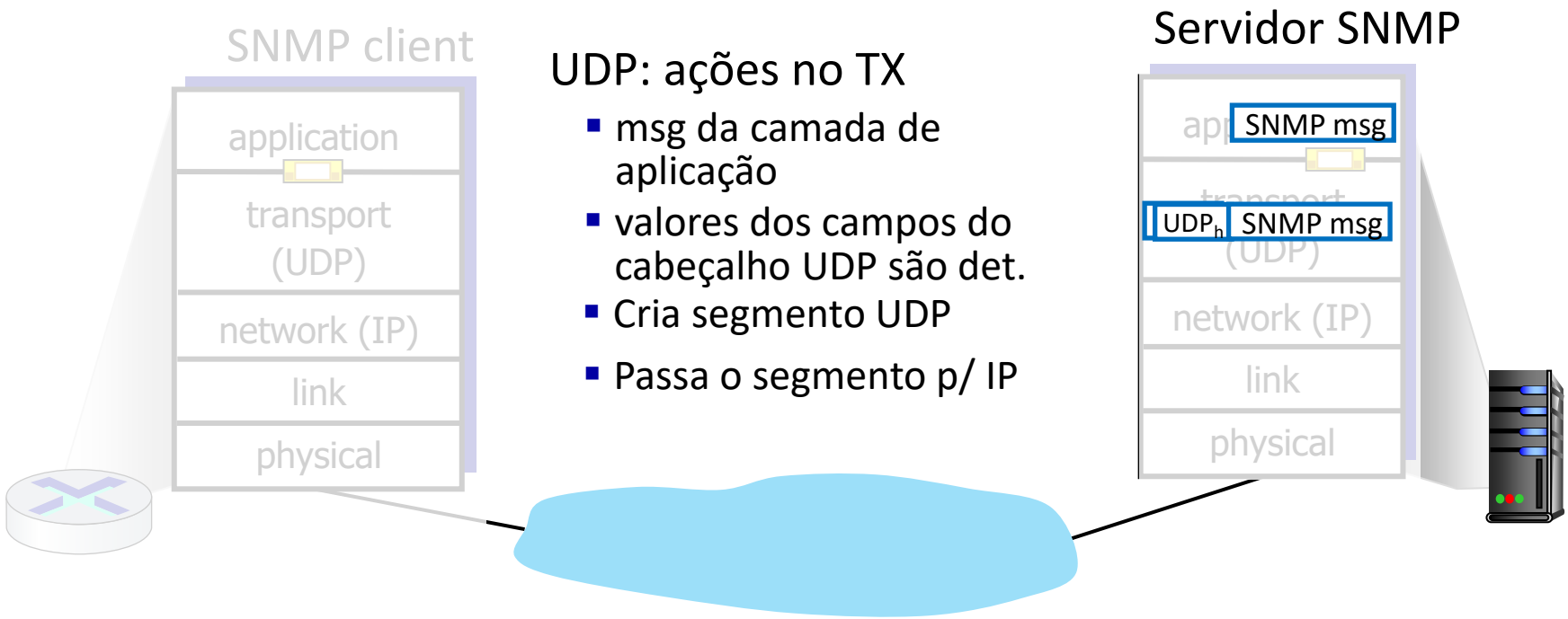


Formato do segmento UDP

UDP: Ações


Cliente SNMP
application
transport (UDP)
network (IP)
link
physical
Servidor SNMP
application
transport (UDP)
network (IP)
link
physical

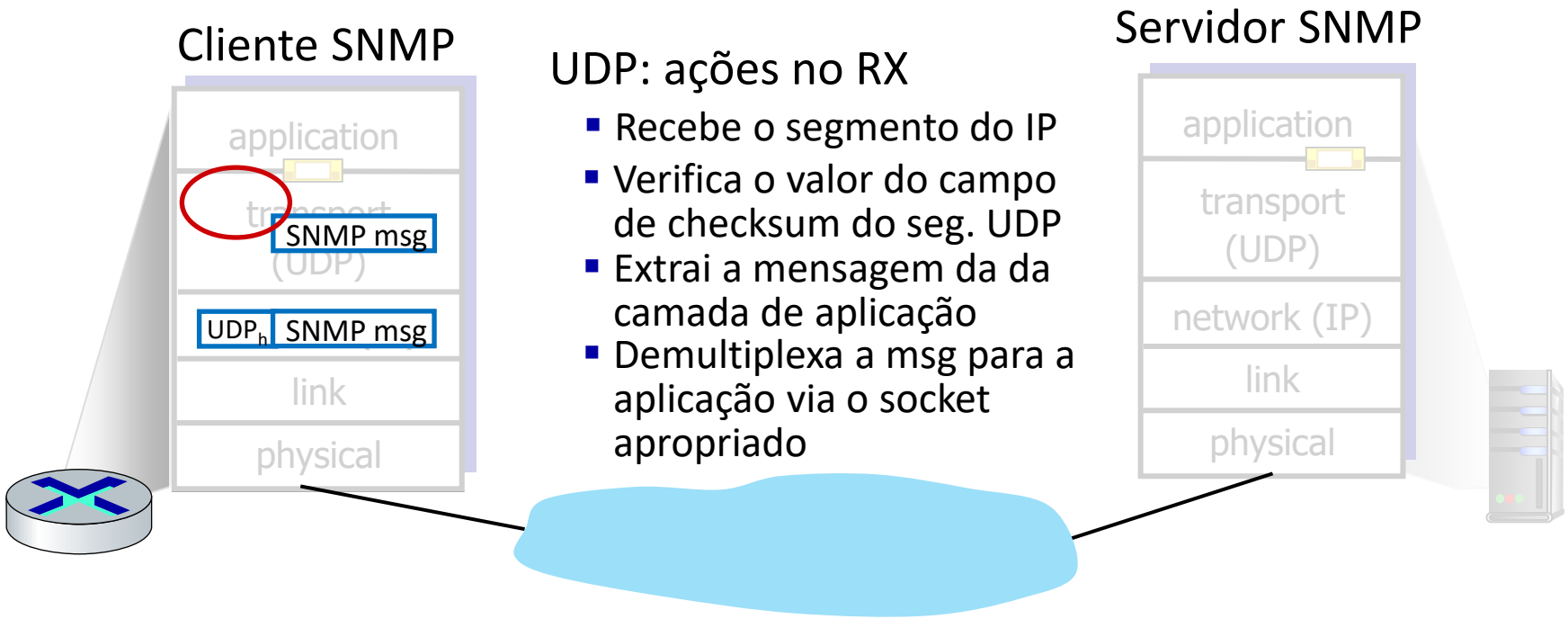
UDP: Ações



UDP: ações no TX

- msg da camada de aplicação
- valores dos campos do cabeçalho UDP são det.
- Cria segmento UDP
- Passa o segmento p/ IP

UDP: Ações



UDP: ações no RX

- Recebe o segmento do IP
- Verifica o valor do campo de checksum do seg. UDP
- Extrai a mensagem da da camada de aplicação
- Demultiplexa a msg para a aplicação via o socket apropriado

Soma de verificação (checksum) UDP

Objetivo: detectar erros (*p. ex,* bits invertidos) na transmissão de um segmento

	1º número	2º número	soma
Transmitido:	5	6	11
Recebido:	4	6	11

Checksum calculado pelo receptor ≠ Checksum calculado pelo remetente (e recebido)

Soma de verificação (checksum) UDP

Objetivo: detectar erros (isto é, bits invertidos) no segmento transmitido

Transmissor:

- Trata o conteúdo do segmento como uma seqüência de inteiros de 16-bits
- checksum: complemento de 1 da soma do conteúdo do segmento
- transmissor coloca o valor calculado no campo **checksum** do cabeçalho UDP

Receptor:

- calcula o checksum do segmento recebido
- verifica se checksum calculado é igual ao valor do campo **checksum** :

NÃO - erro detectado

⇒ descarta pacote

SIM - nenhum erro detectado.

Internet checksum: exemplo

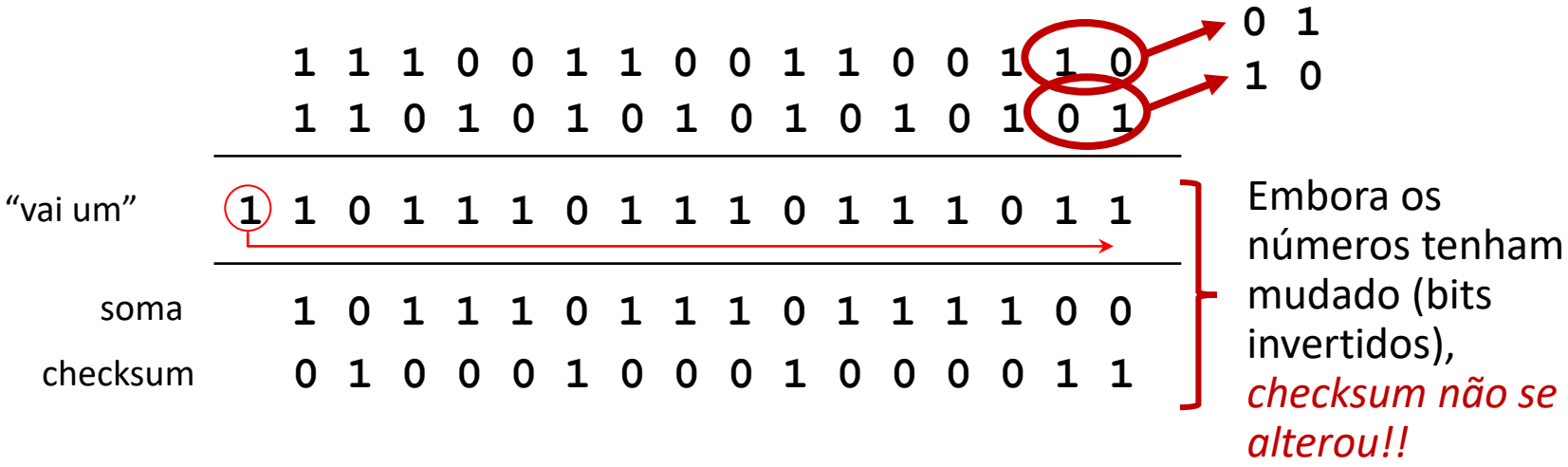
exemplo: soma de dois inteiros de 16 bits

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

Nota: qdo se soma números, um "vai-um" no bit mais significativo necessita ser adicionado ao resultado

Internet checksum: proteção é fraca!

exemplo: soma de dois inteiros de 16 bits



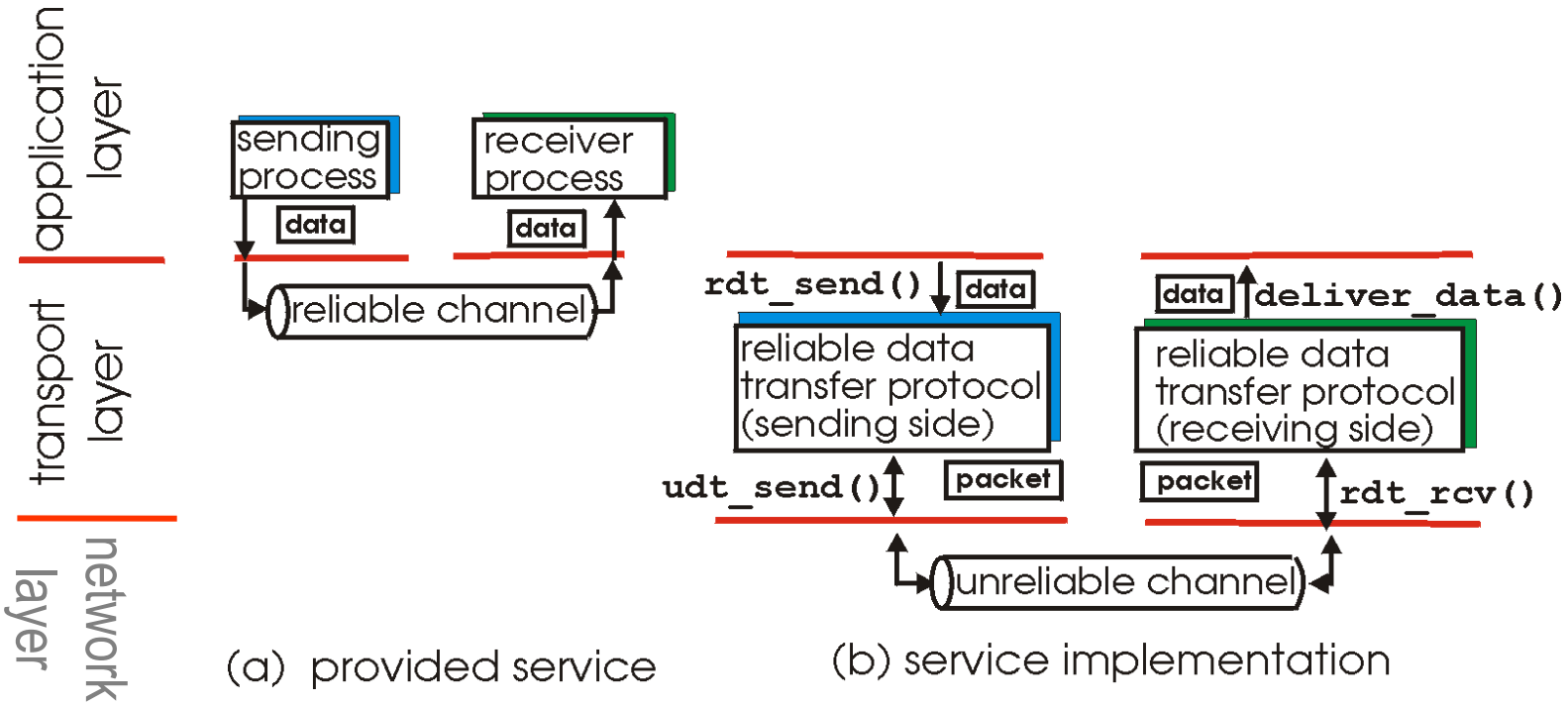
Embora os números tenham mudado (bits invertidos), *checksum não se alterou!!*

Resumo do UDP

- Protocolo “sem frescuras”:
  - Segmentos podem ser perdidos ou entregues for a de ordem
  - Serviço de “melhor esforço”: “envia e espera pelo melhor”
- UDP tem pontos positivos:
  - Não é necessário o estabelecimento de conexão (sem RTT)
  - Pode funcionar mesmo quando o serviço de rede está comprometido
  - Ajuda na confiabilidade (***checksum***)
- Fornece funcionalidade suficiente para o uso do HTTP/3

Princípios de Transferência Confiável de Dados

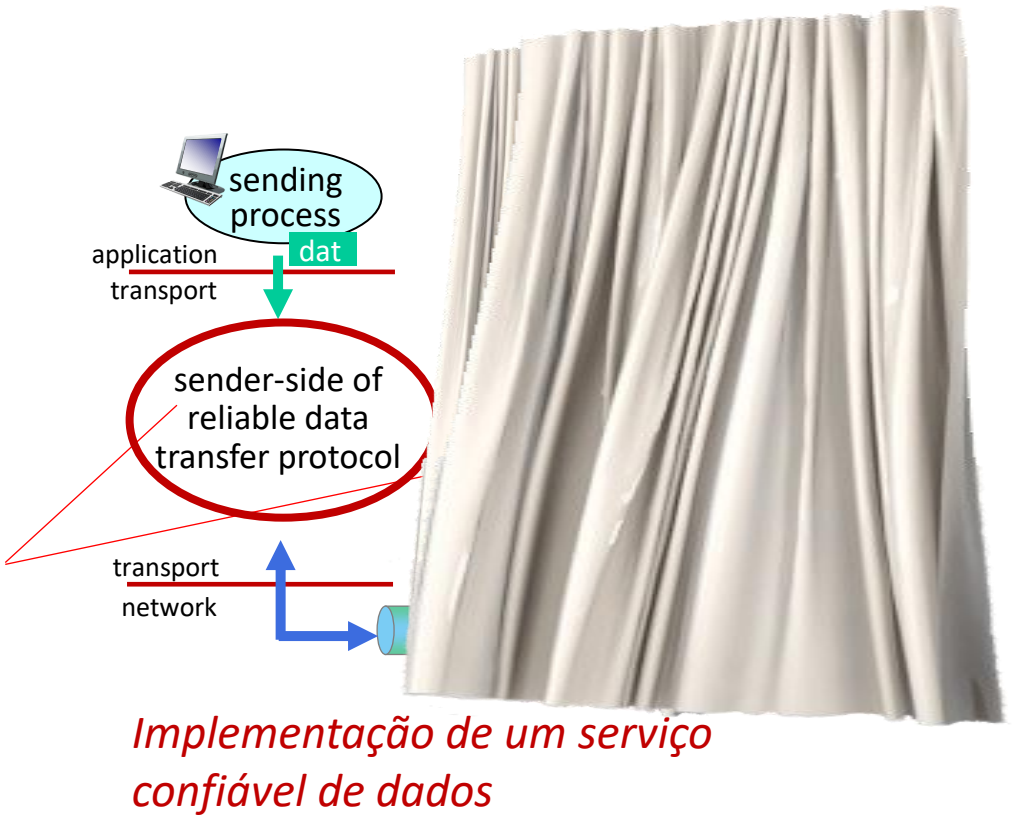
- Importante nas camadas de aplicação, transporte e enlace
- Top 10 na lista dos tópicos mais importantes de redes!



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

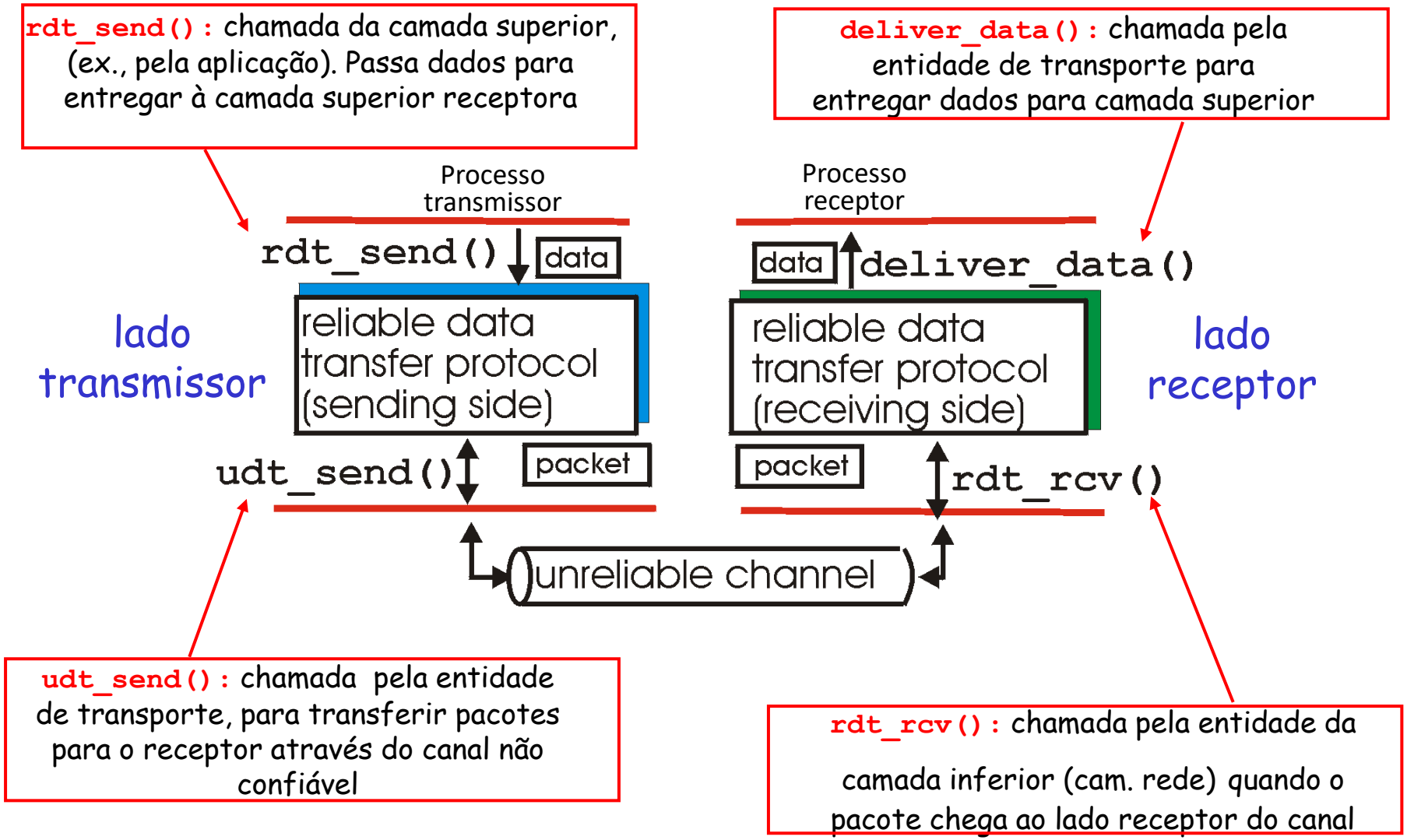
Princípios de Transferência Confiável de Dados

Transmissor e receptor não conhecem o "estado" um do outro, ou seja, se a mensagem foi recebida corretamente



Implementação de um serviço confiável de dados

Transferência Confiável: Modelo Básico

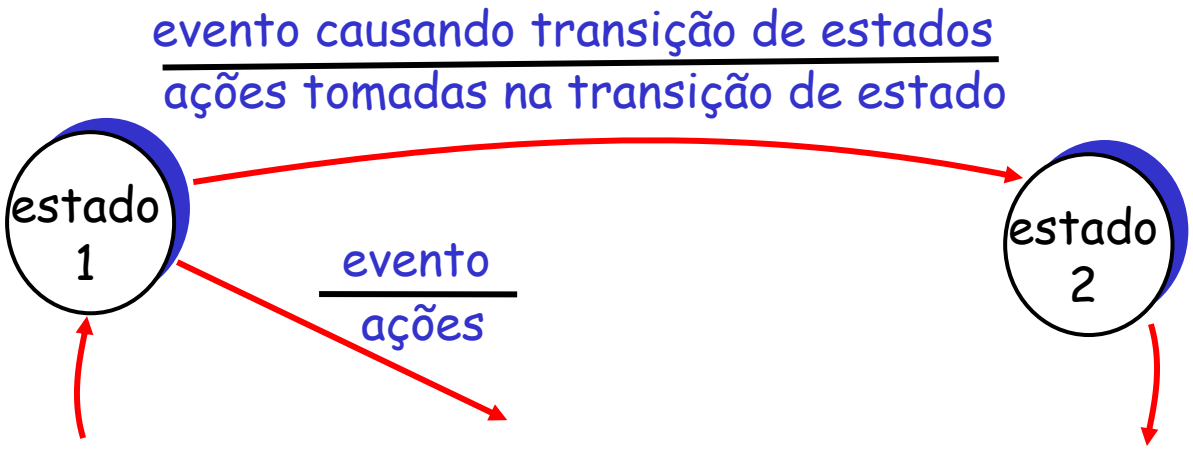


Transferência confiável: O ponto de partida

Etapas:

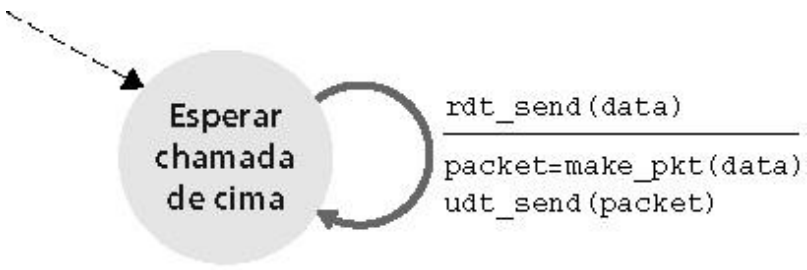
- desenvolver incrementalmente o transmissor e o receptor de um protocolo confiável de transferência de dados (rdt)
- considerar apenas transferências de dados unidirecionais
  - mas informação de controle deve fluir em ambas as direções!
- usar máquinas de estado finitas (FSM) para especificar o protocolo transmissor e o receptor

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

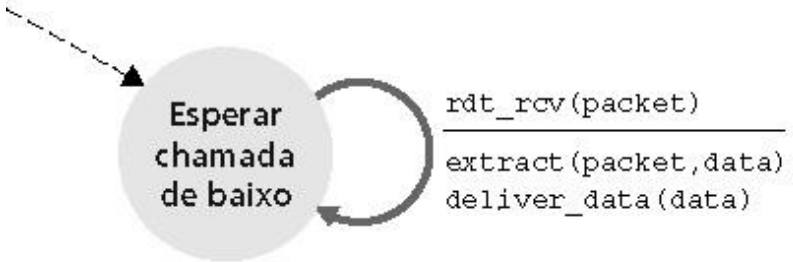


rdt1.0: Transferência confiável sobre um canal perfeitamente confiável

- canal de transmissão perfeitamente confiável
  - não há erros de bits
  - não há perdas de pacotes
- FSMs separadas para transmissor e receptor:
  - transmissor envia dados para o canal subjacente
  - receptor lê os dados do canal subjacente
  - Não há ack´s e velocidades do receptor e do transmissor são consideradas iguais



a. rdt1.0: lado remetente



b. rdt1.0: lado destinatário

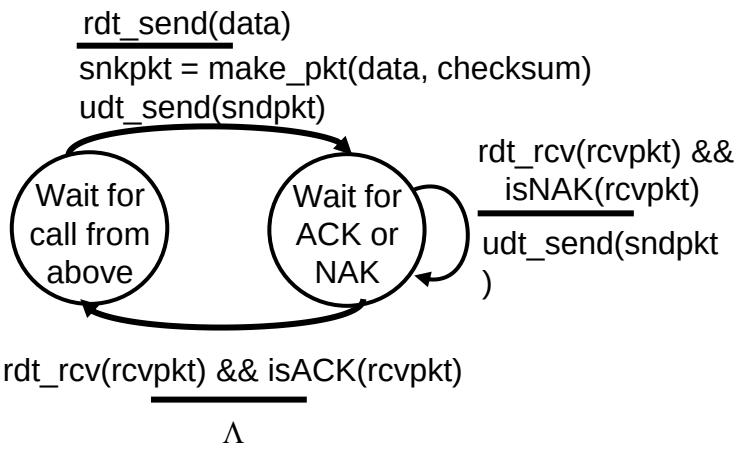
rdt2.0: Canal com erros de bit

- Canal subjacente pode trocar valores dos bits num pacote, porém continuam a ser recebidos em ordem
  - lembrete: checksum do UDP pode detectar erros de bits
- Questão: Como recuperar-se desses erros?
  - *reconhecimentos (ACKs):* receptor avisa explicitamente ao transmissor que o pacote foi recebido corretamente
  - *reconhecimentos negativos (NAKs):* receptor avisa explicitamente ao transmissor que o pacote tem erros

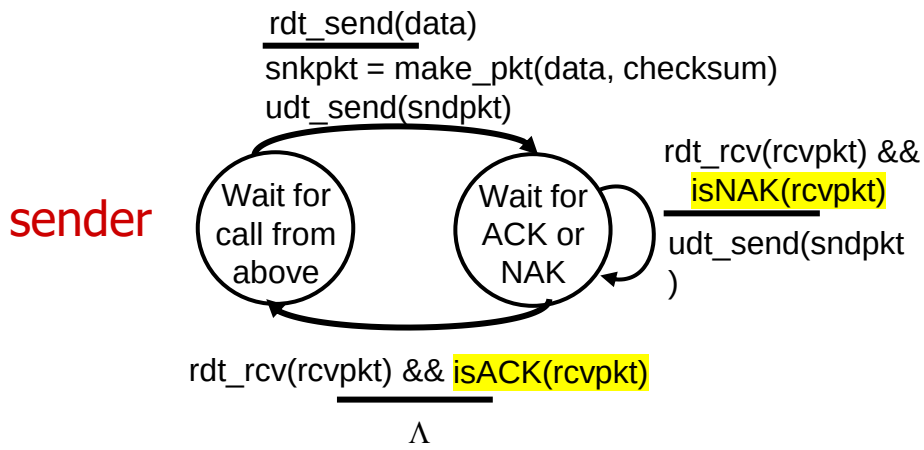
  ⇒ transmissor reenvia o pacote quando da recepção de um NAK
- Novos mecanismos no `rdt2.0` (além do `rdt1.0`):
  - deteção de erros
  - retorno do receptor: mensagens de controle (ACK,NAK) do receptor para o transmissor

rdt2.0: especificação da FSM

sender



rdt2.0: especificação da FSM

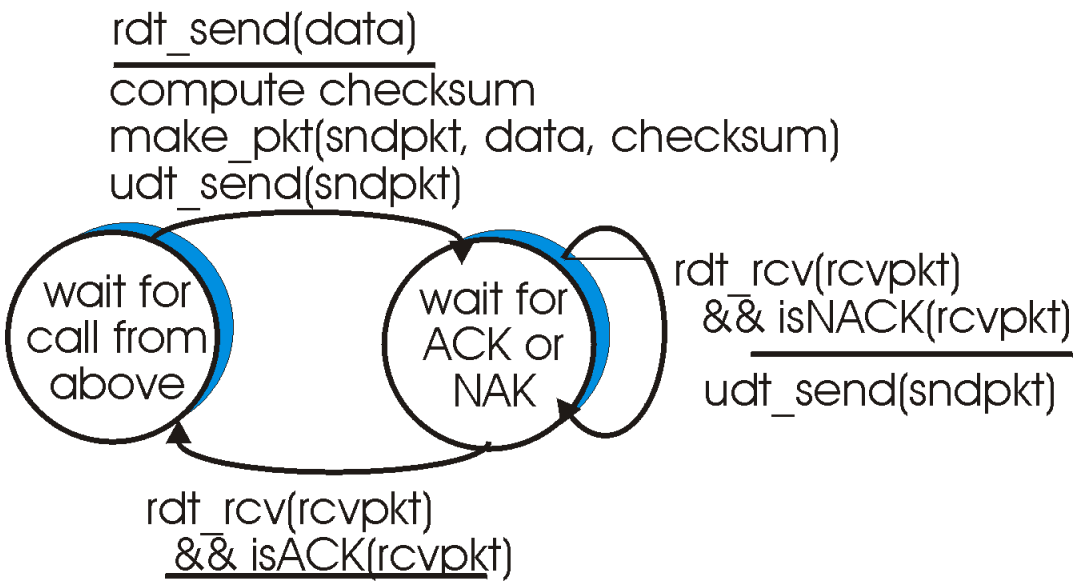


Nota:

O "estado" do destinatário (o destinatário recebeu minha mensagem corretamente?) não é conhecido pelo remetente, a menos que seja comunicado, de alguma forma, pelo destinatário.

É por isso que precisamos de um protocolo!

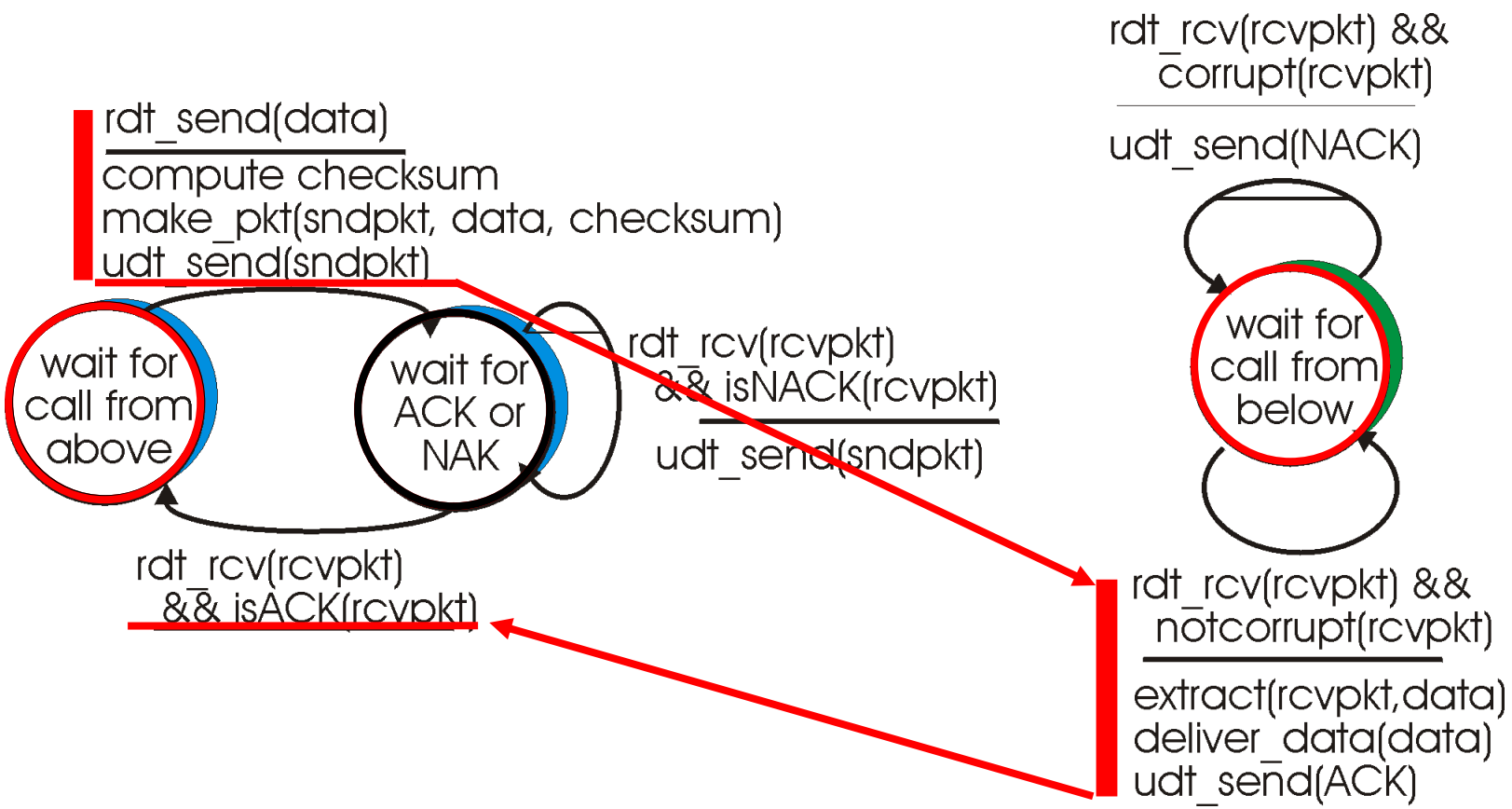
rdt2.0: especificação da FSM



FSM do transmissor

FSM do receptor

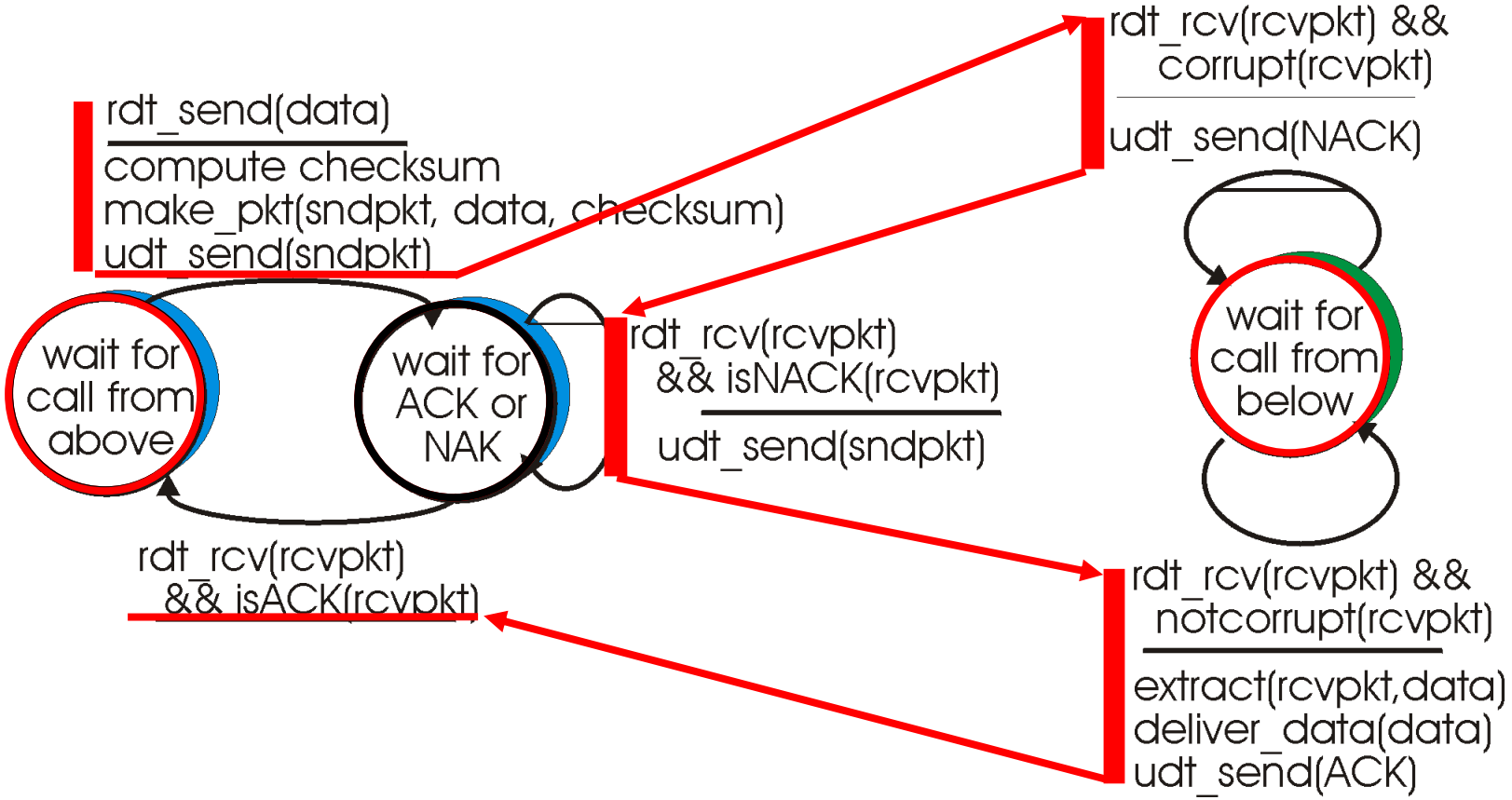
rdt2.0: em ação (ausência de erros)



FSM do transmissor

FSM do receptor

rdt2.0: em ação (cenário com erros)



FSM do transmissor

FSM do receptor

rdt2.0 tem um problema fatal!

O que acontece se o ACK/NAK é corrompido?

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

O que fazer?

- transmissor envia um ACK/NAK para reconhecer o ACK/NAK do receptor. Mas o que acontece se este ACK/NAK se perdem?
- receptor retransmite o ACK/NAK. Mas isto poderia causar confusão
  - p. ex.: a retransmissão de um pacote recebido corretamente, isto é, duplicatas

Claramente, esta não parece ser uma solução satisfatória!

rdt2.0 tem um problema fatal!

Uma solução mais simples:

Baseada na deteção de duplicatas pelo receptor

Tratando duplicatas:

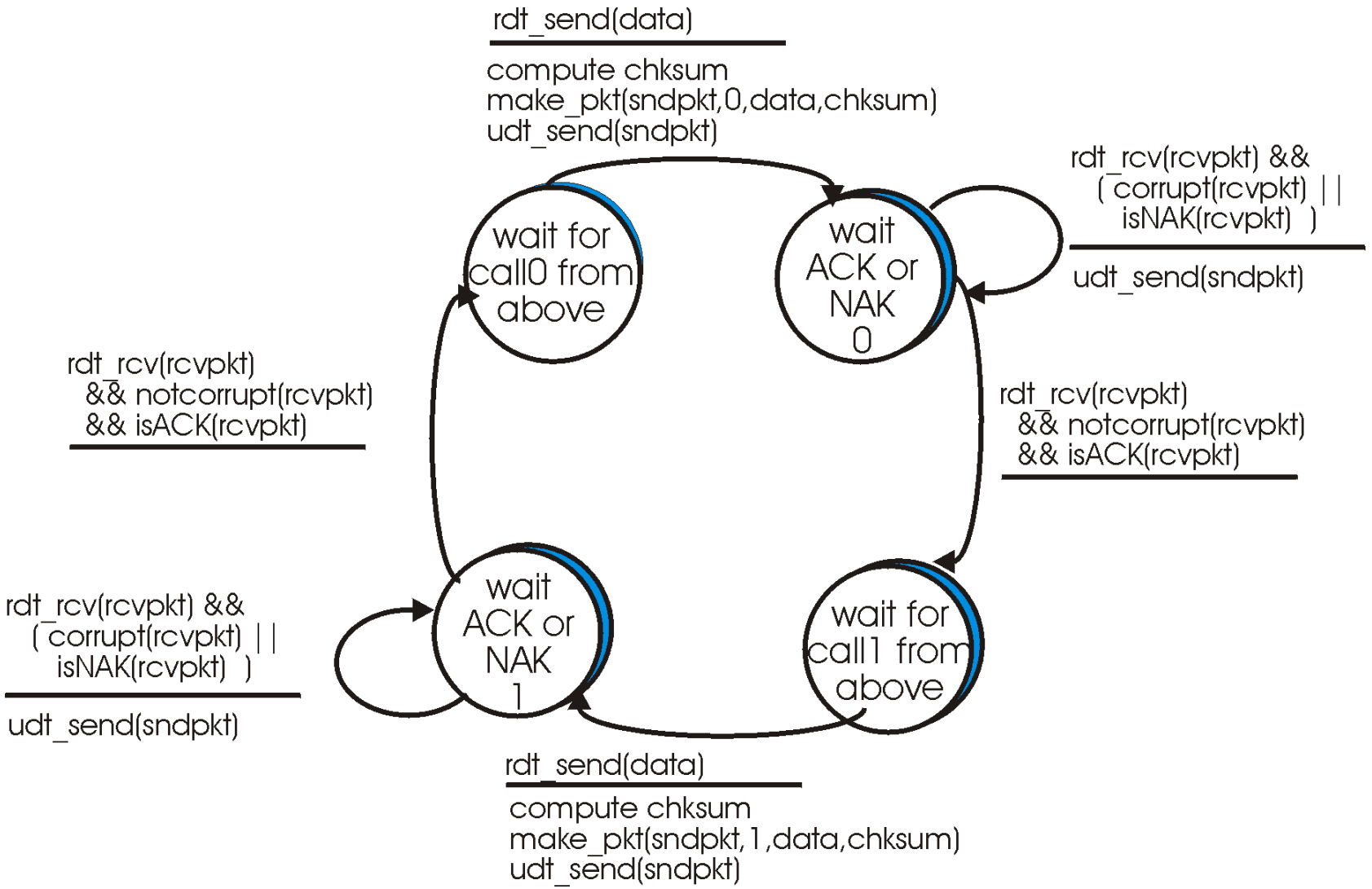
- transmissor acrescenta *número de seqüência* em cada pacote
- transmissor reenvia o último pacote se ACK/NAK for corrompido
- receptor descarta (não passa para a aplicação) pacotes duplicados

Obs: supõe que não há perda de pcts

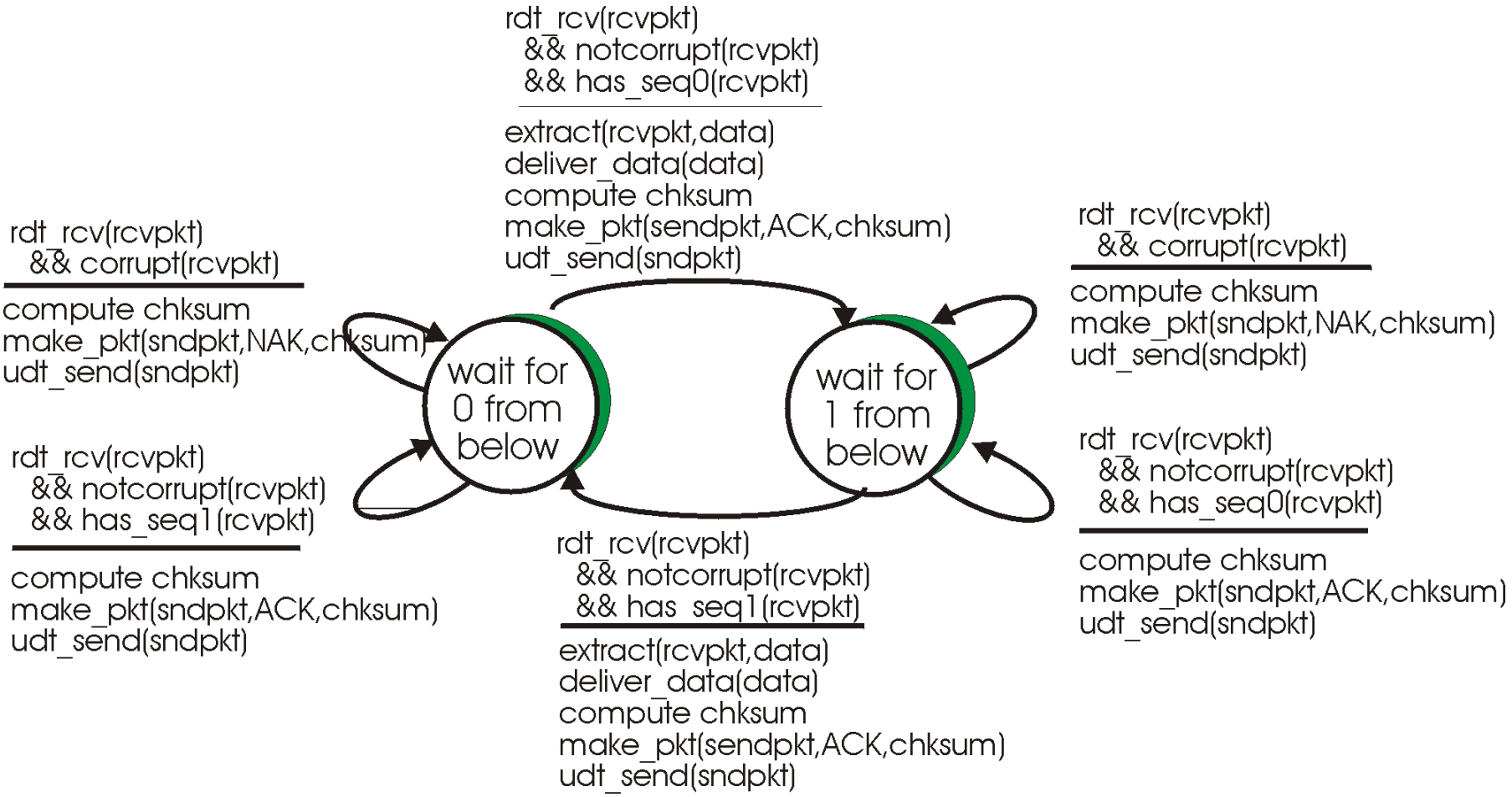
stop and wait

Transmissor envia um pacote e então espera pela resposta do receptor

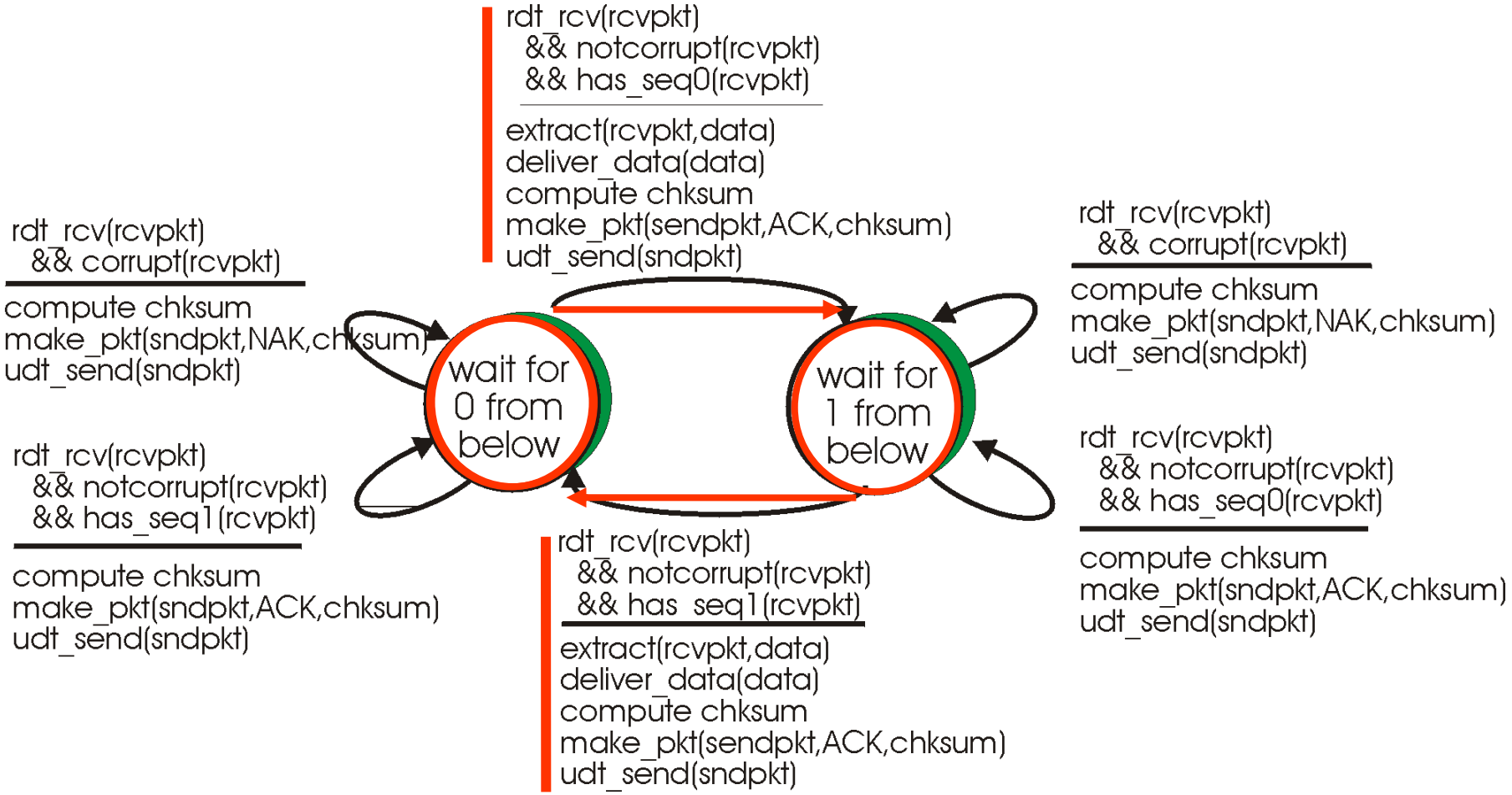
rdt2.1: Transmissor, trata ACK/NAKs corrompidos



rdt2.1: Receptor, trata ACK/NAKs perdidos



rdt2.1: Receptor, trata ACK/NAKs perdidos: operação sem erros



rdt2.1: Discussão

Transmissor:

- adiciona número de seqüência ao pacote
- Dois números (0 e 1) bastam. Porque?
  => stop and wait
- 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 seqüência 0 ou 1

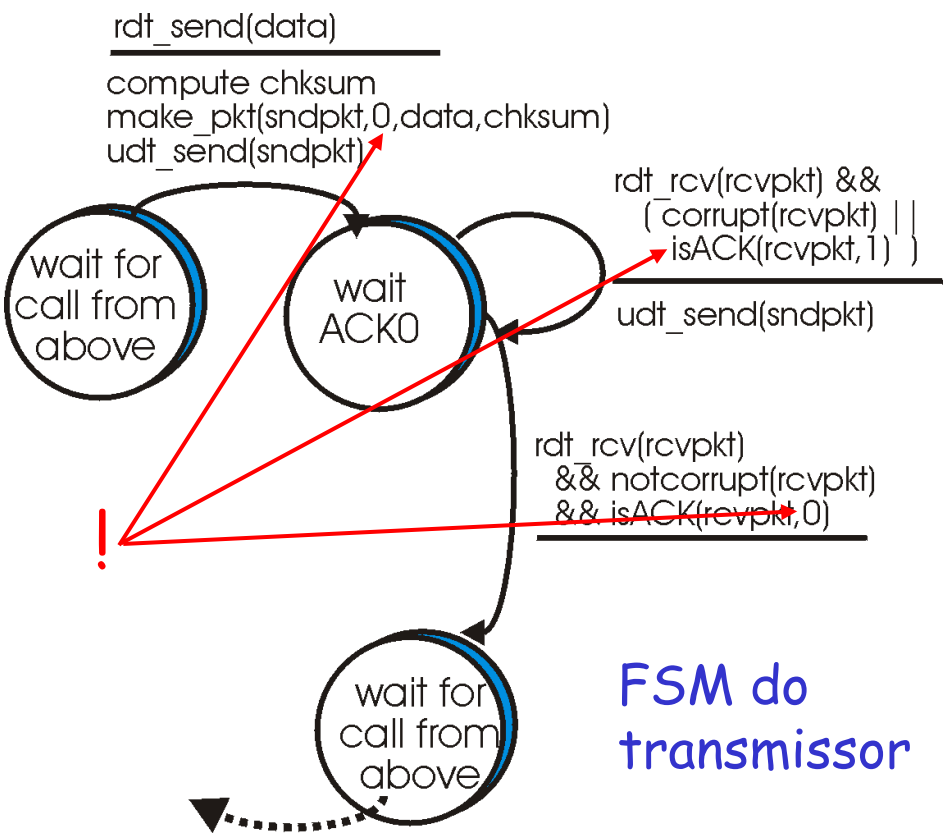
Receptor:

- deve verificar se o pacote recebido é duplicado
- estado indica se o pacote 0 ou 1 é esperado

nota: receptor pode não saber se o seu último ACK/NAK foi recebido pelo transmissor

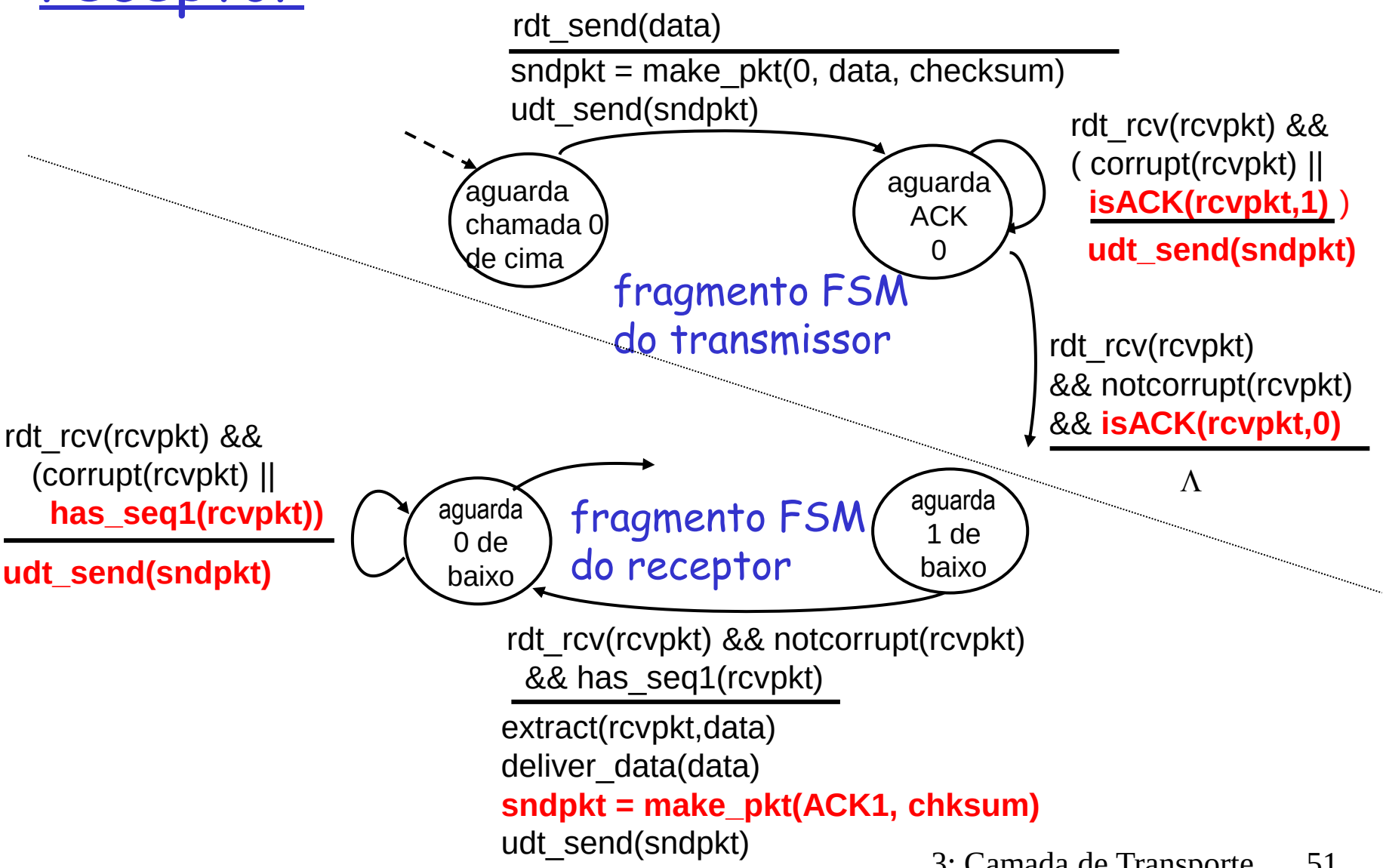
rdt2.2: um protocolo sem NAK

- mesma funcionalidade do rdt2.1, usando somente *ACKs*
- ao invés de enviar NAK, o receptor envia *ACK* para o último pacote recebido sem erro
  - receptor deve incluir explicitamente o número de seqüência do pacote sendo reconhecido
- *ACKs* duplicados no transmissor resultam na mesma ação do NAK: *retransmição do pacote corrente*



FSM do transmissor

rdt2.2: fragmentos do transmissor e receptor



rdt3.0: canal c/ erro de bit e perda de pct

Nova Hipótese:

- canal de transmissão pode também perder pacotes (de dados ou ACKs)
- checksum, números de seqüência, ACKs, retransmissões serão de ajuda, mas não o bastante

Q: como tratar com perdas?

- transmissor espera um tempo até que certos dados ou ACKs sejam perdidos, então retransmite

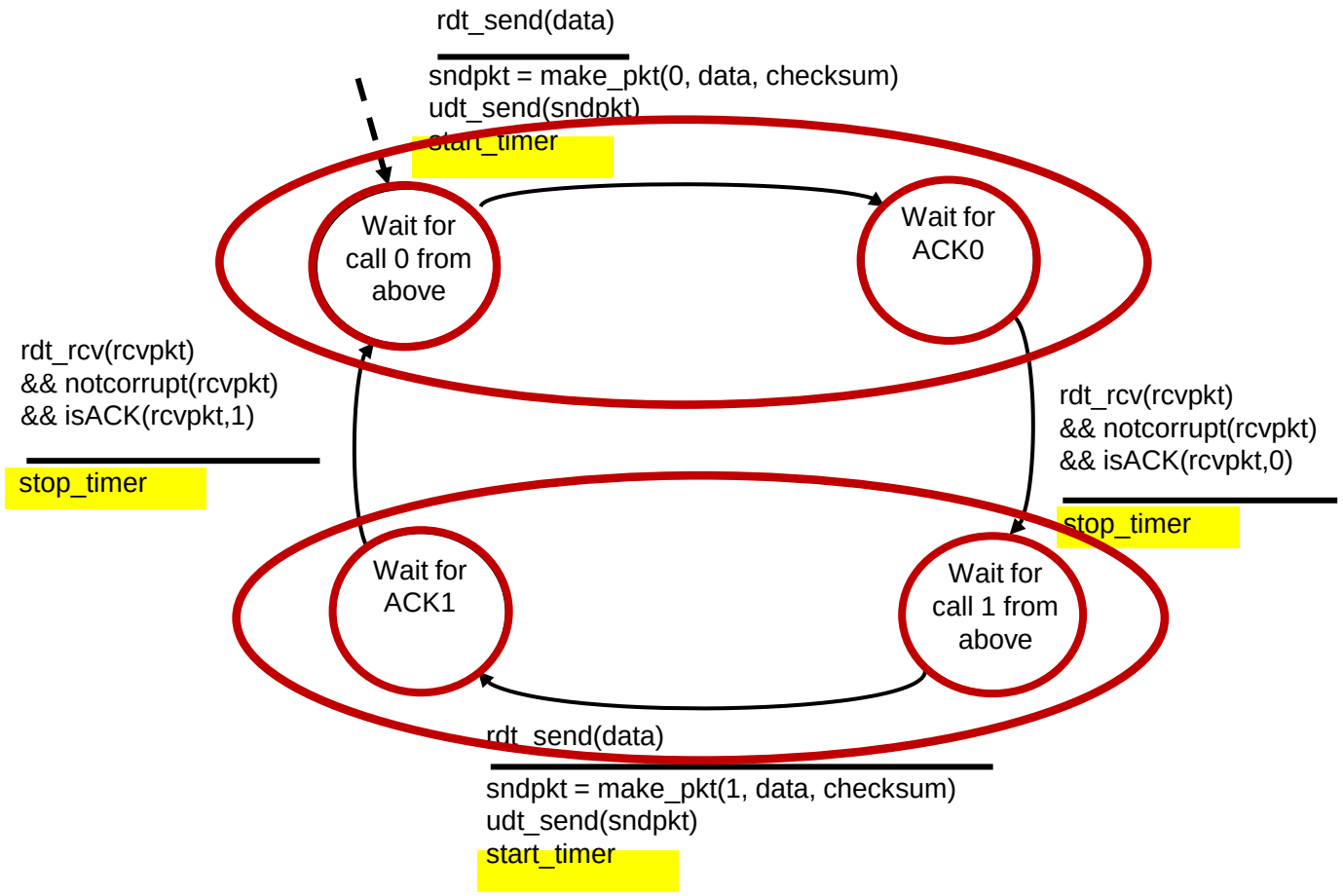
rdt3.0: canal c/ erro de bit e perda de pct

Abordagem:

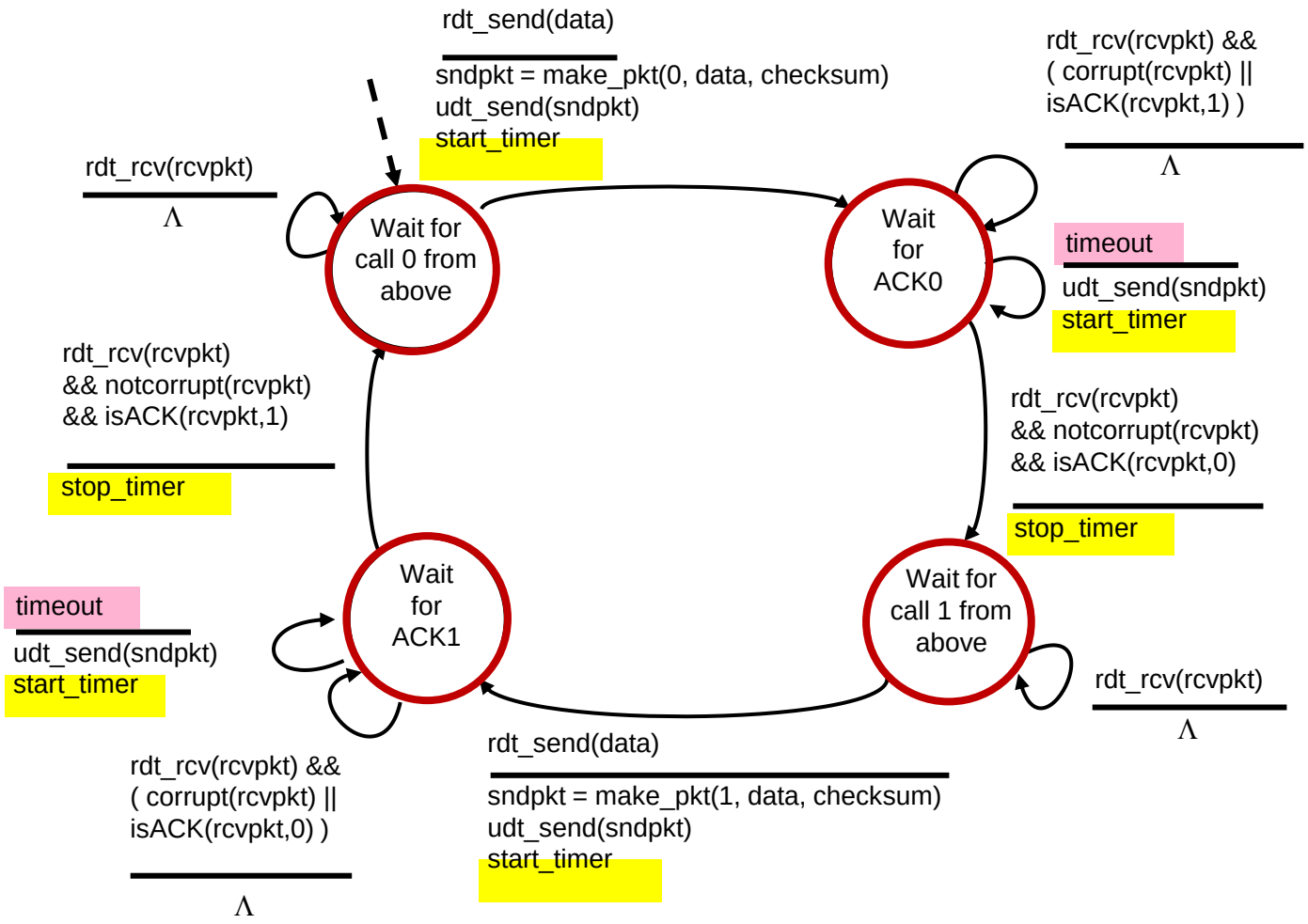
- transmissor espera um tempo "razoável" pelo ACK
- retransmite se nenhum ACK for recebido neste tempo
  - timeout
- se o pacote (ou ACK) estiver apenas atrasado (não perdido):
  - retransmissão será duplicata, mas os números de seqüência já tratam com isso
  - receptor deve especificar o número de seqüência do pacote sendo reconhecido
- usa um temporizador para retransmitir o pacote após um intervalo de tempo "razoável"

timeout

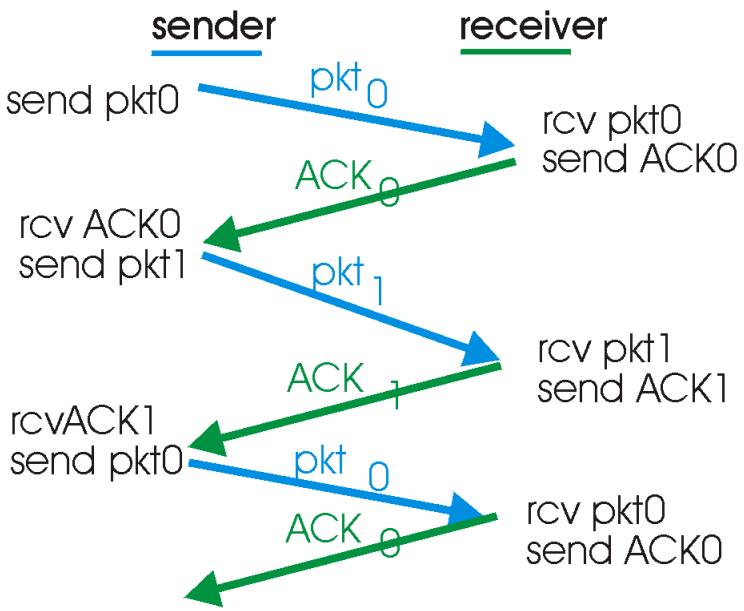
rdt3.0 transmissor



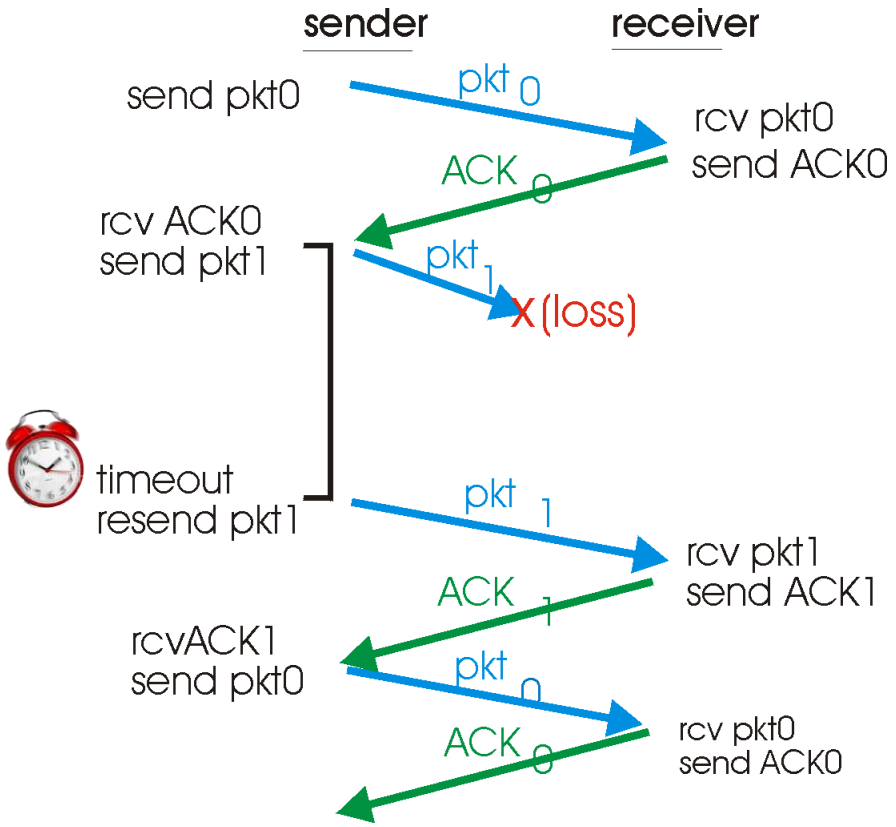
rdt3.0 transmissor



rdt3.0 em ação

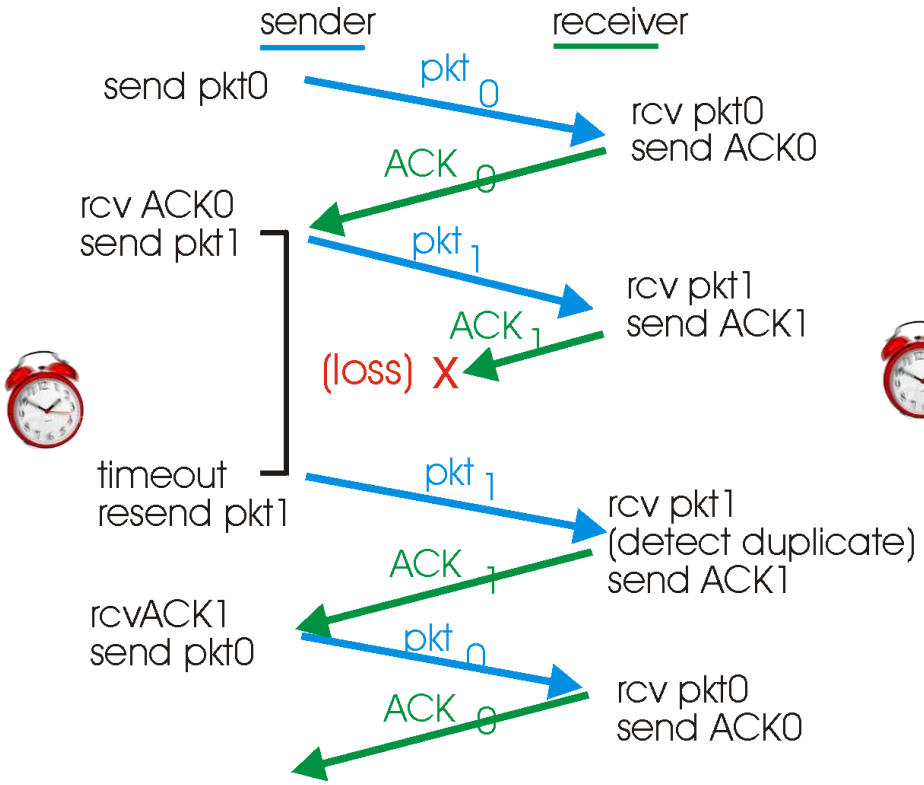


(a) operação sem perda

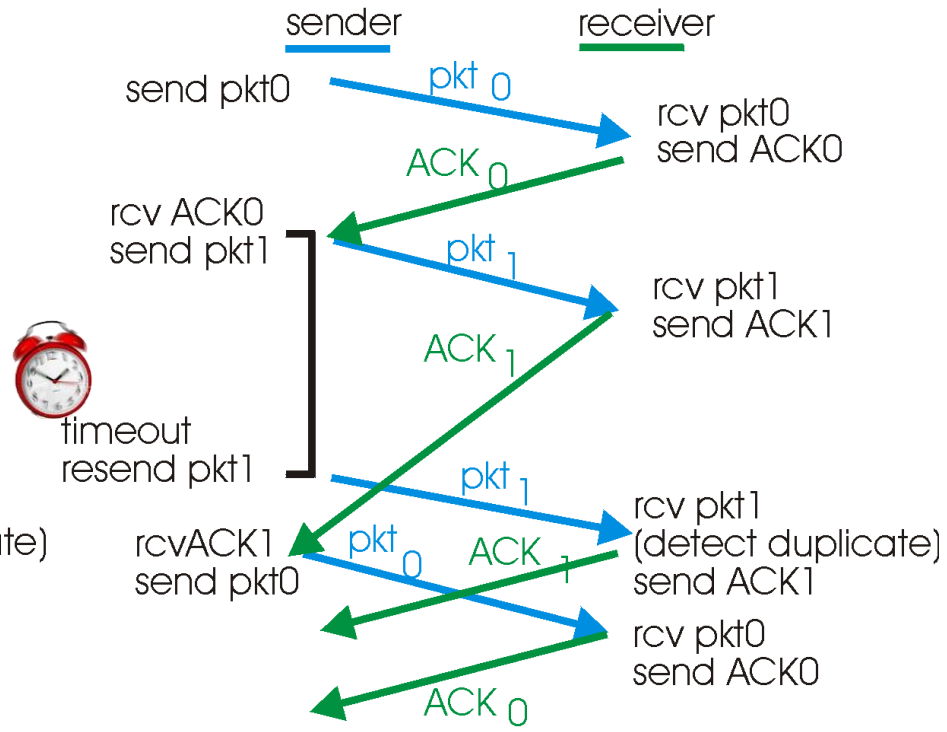


(b) pacote perdido

rdt3.0 em ação



(c) ACK perdido



(d) timeout prematuro

Desempenho do rdt3.0

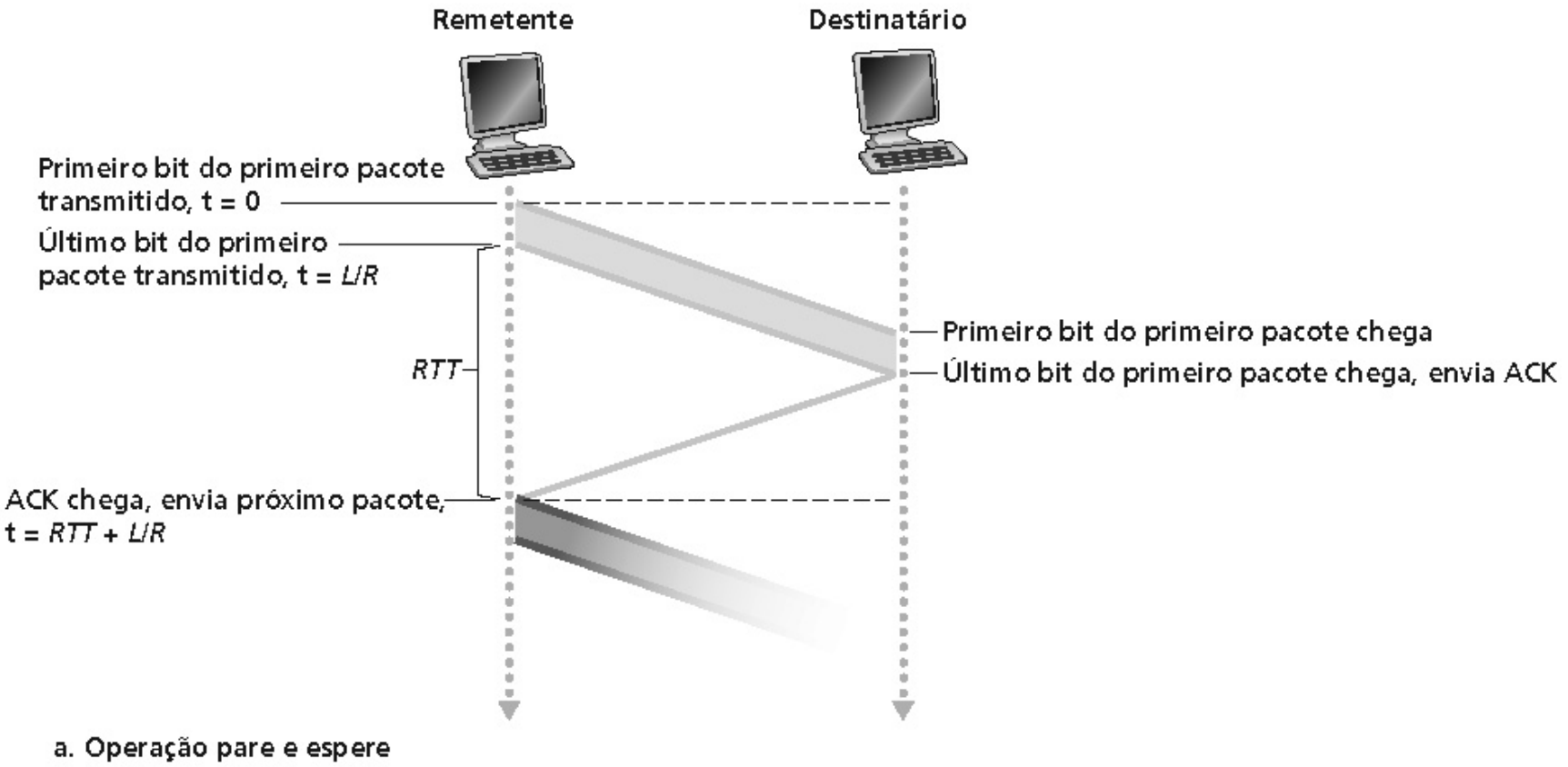
- rdt3.0 funciona, mas o desempenho é sofrível
- exemplo: enlace de 1 Gbps, 15 ms de atraso fim a fim, pacotes de 8000 bits:

$$\text{transmissão} = \frac{8000 \text{ bits}}{10^{**}9 \text{ b/seg}} = 8\ \mu s$$

$$\text{Utilização} = U = \frac{\text{fração do tempo}}{\text{transmissor ocupado}} = \frac{8\ \mu s}{30{,}016 \text{ ms}} = 0.00027$$

- Um pacote de 1KB cada 30 ms -> 33kB/seg de vazão sobre um canal de 1 Gbps
- o protocolo de rede limita o uso dos recursos físicos!
- Se considerarmos os tempos de processamento das camadas inferiores bem como os tempos de espera nas filas dos buffers, o desempenho pioraria

rdt3.0: operação pare e espere



a. Operação pare e espere

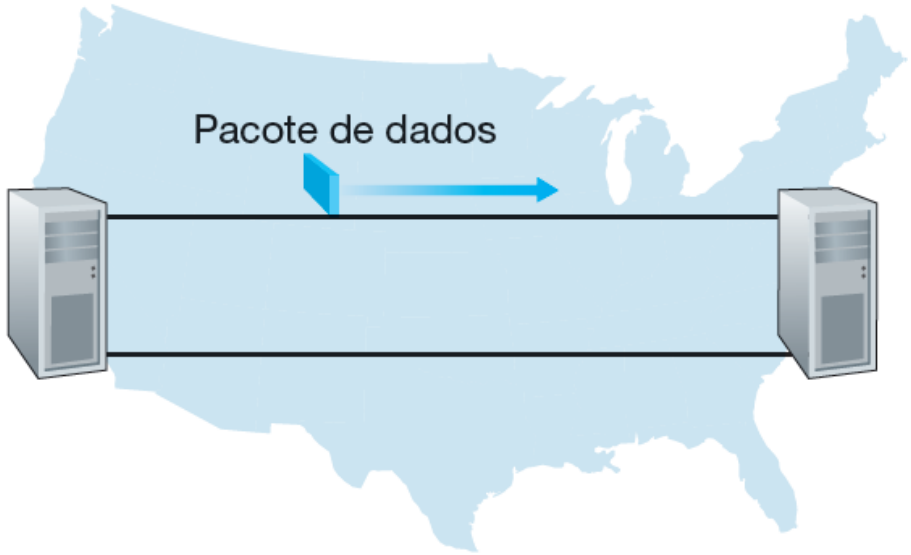
$$U_{\text{tx}} = \frac{L/R}{\text{RTT} + L/R} = \frac{0{,}008}{30{,}008} = 0{,}00027$$

Protocolos de transferência confiável de dados com paralelismo

KUROSE | ROSS
Redes de computadores e a internet
uma abordagem top-down
6ª edição

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

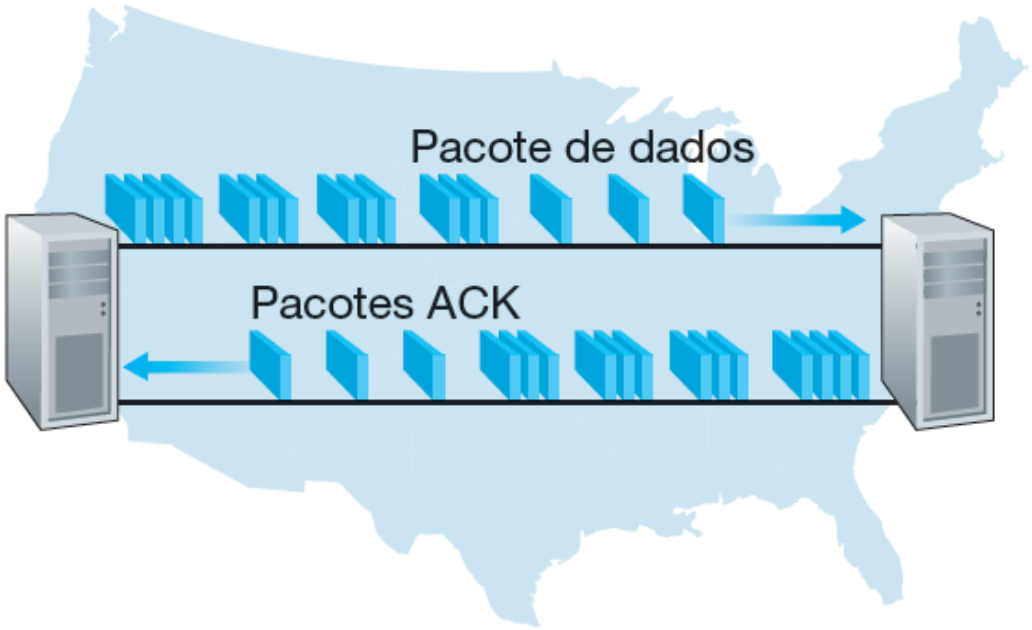
- Um protocolo pare e espere em operação



Protocolos de transferência confiável de dados com paralelismo

KUROSE | ROSS
Redes de computadores e a internet
uma abordagem top-down
6ª edição

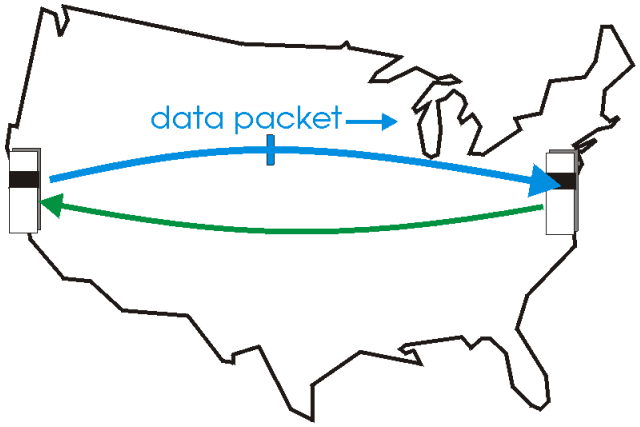
- Um protocolo com paralelismo em operação



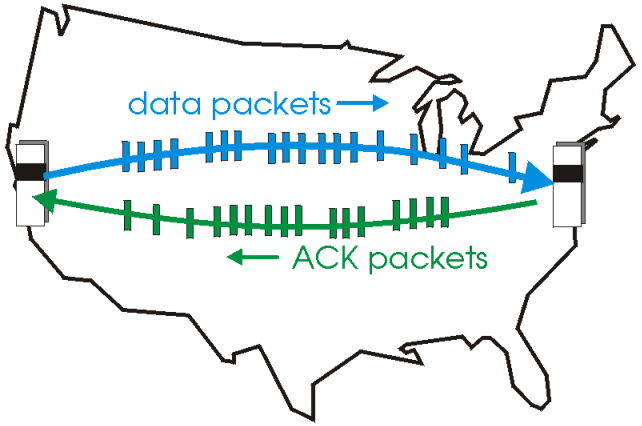
Protocolos com Paralelismo (pipelining)

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

- faixa de números de seqüência deve ser aumentada
- armazenamento dos pacotes no transmissor e/ou no receptor



(a) operação do protocolo stop-and-wait



(a) operação do protocolo com paralelismo

Duas formas genéricas de protocolos com paralelismo:

Go-Back-N e selective repeat