EEL878 - Redes de Computadores I

Prof. Luís Henrique Maciel Kosmalski Costa

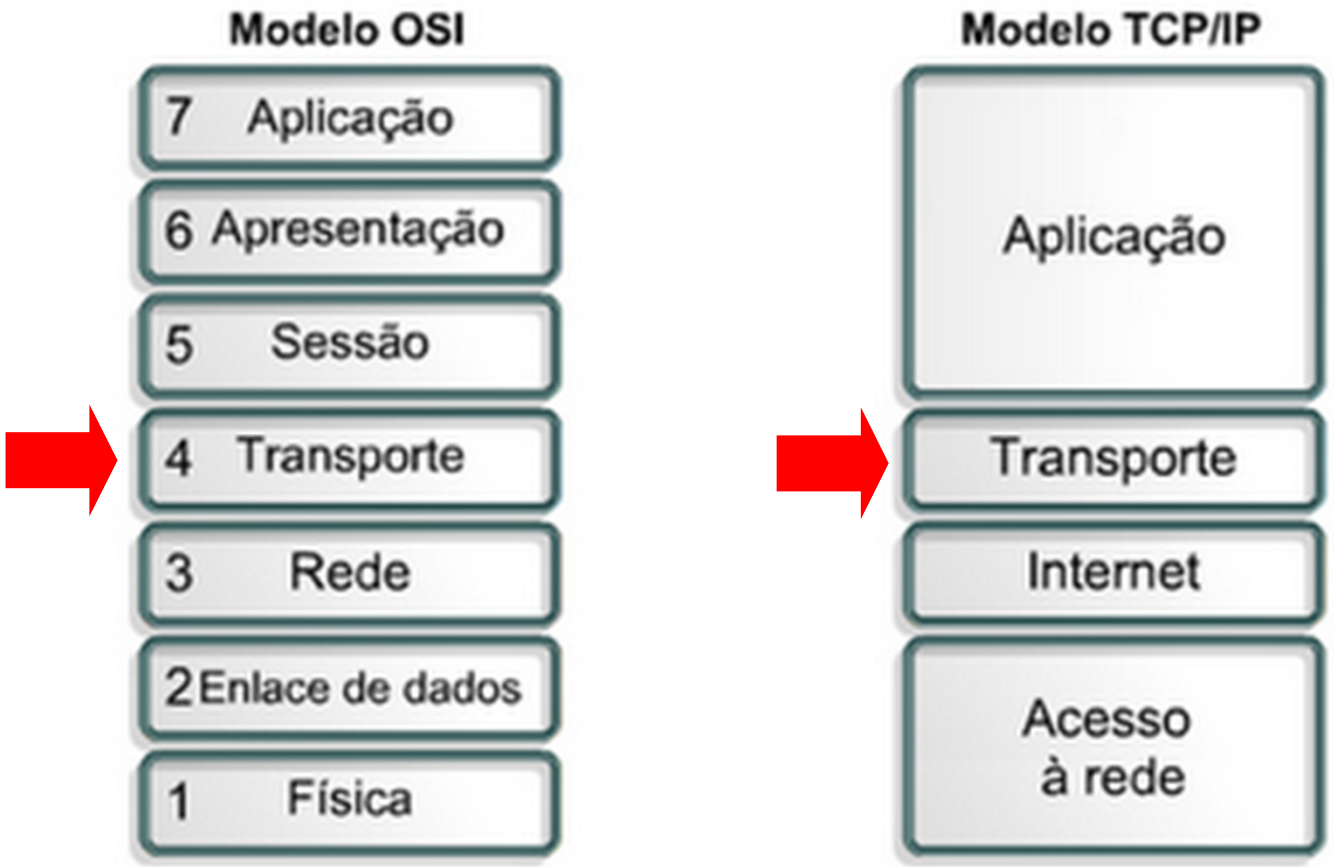
http://www.gta.ufrj.br/ensino/eel878

luish@gta.ufrj.br

Parte III

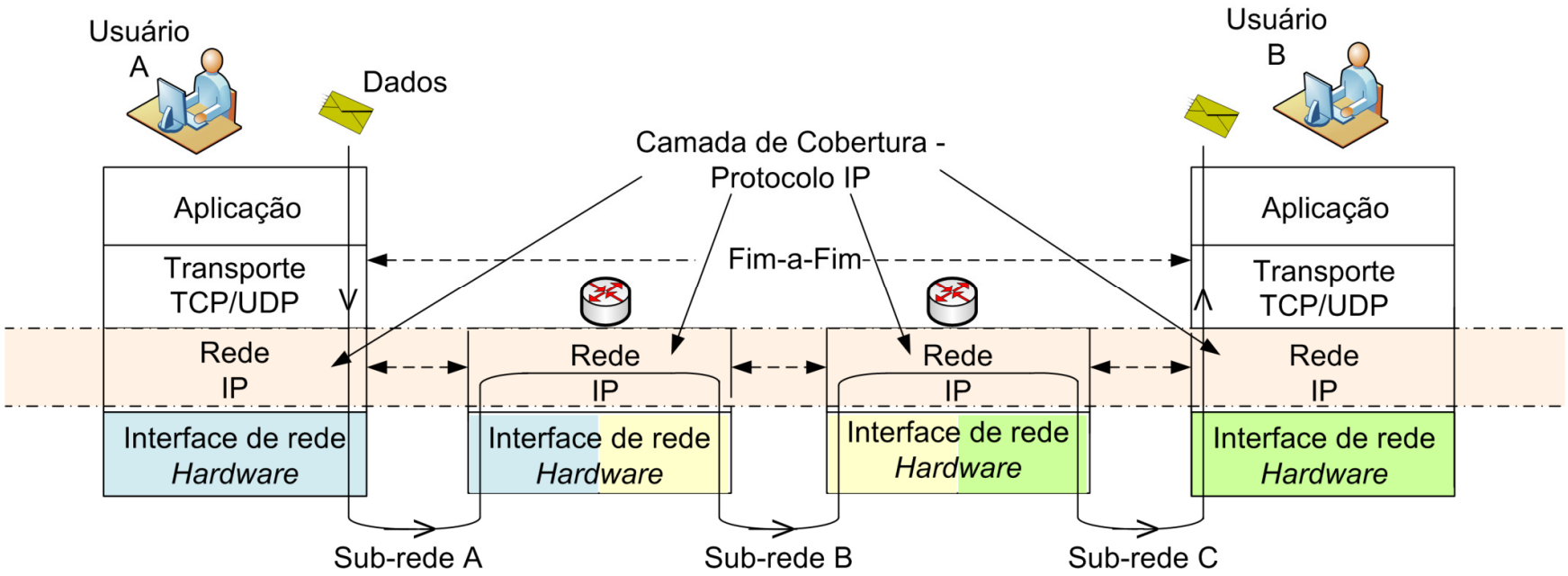
Camada de Transporte e seus Protocolos

Camada de Transporte



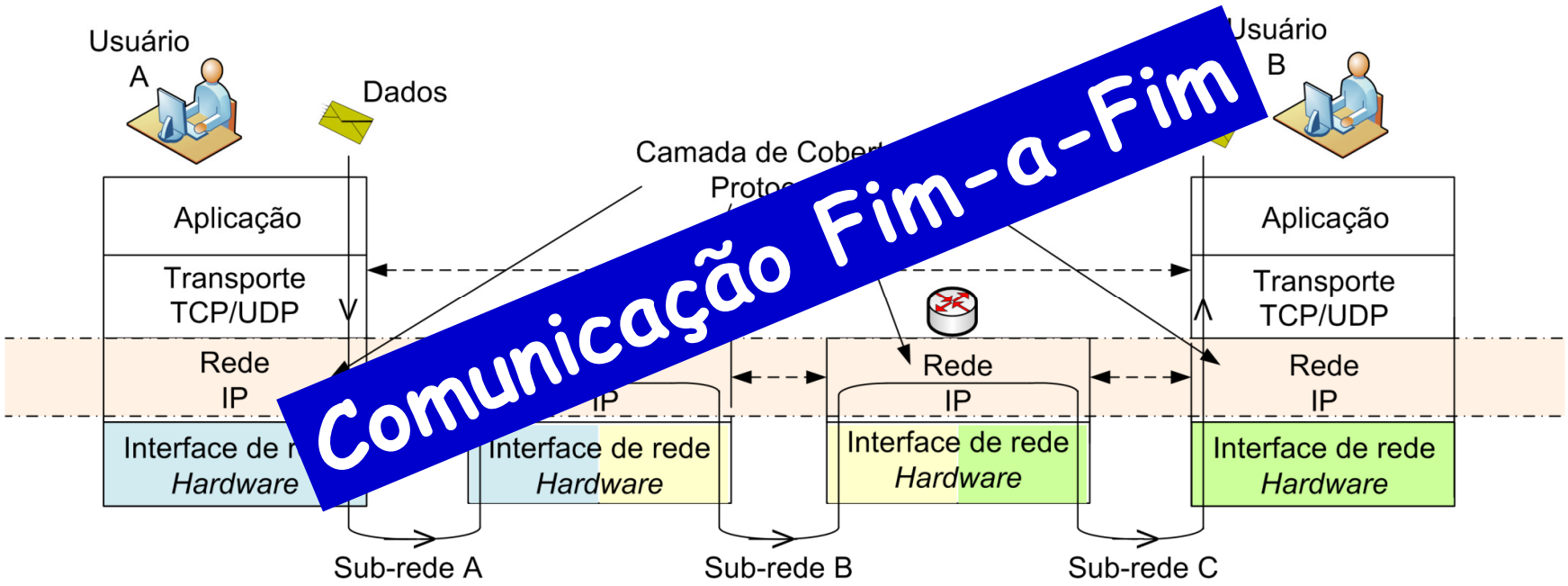
Camada de Transporte

- Provê um **canal lógico** de comunicação entre **processos** em **diferentes sistemas finais**
  - Para a aplicação, os sistemas finais estão diretamente conectados

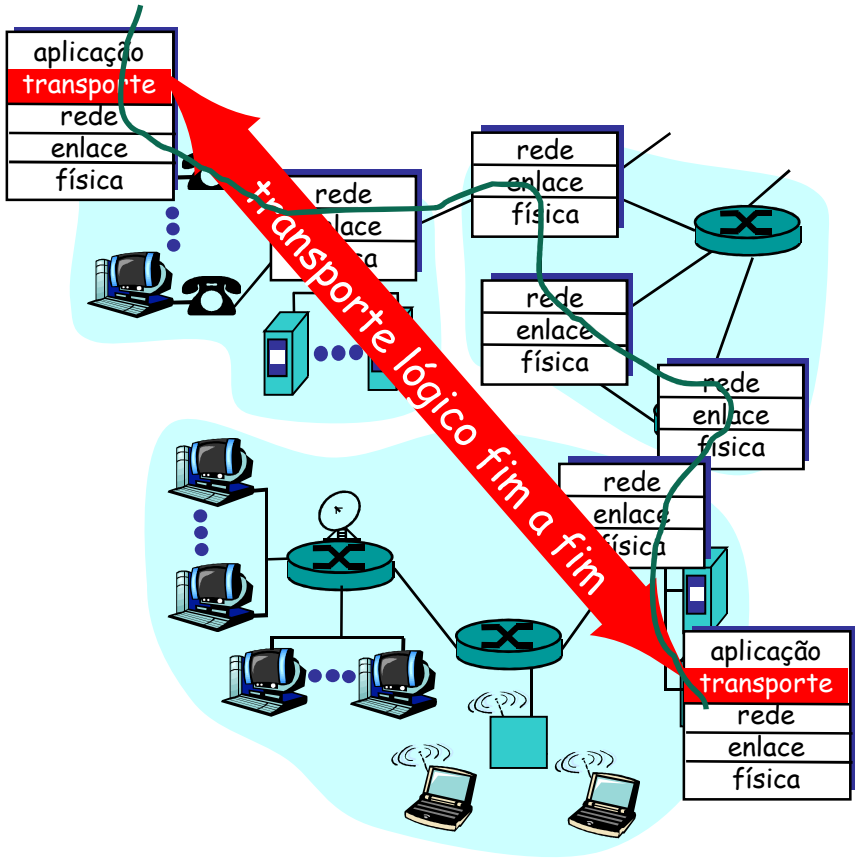


Camada de Transporte

- Provê um **canal lógico** de comunicação entre **processos** em **diferentes sistemas finais**
  - Para a aplicação, os sistemas finais estão diretamente conectados

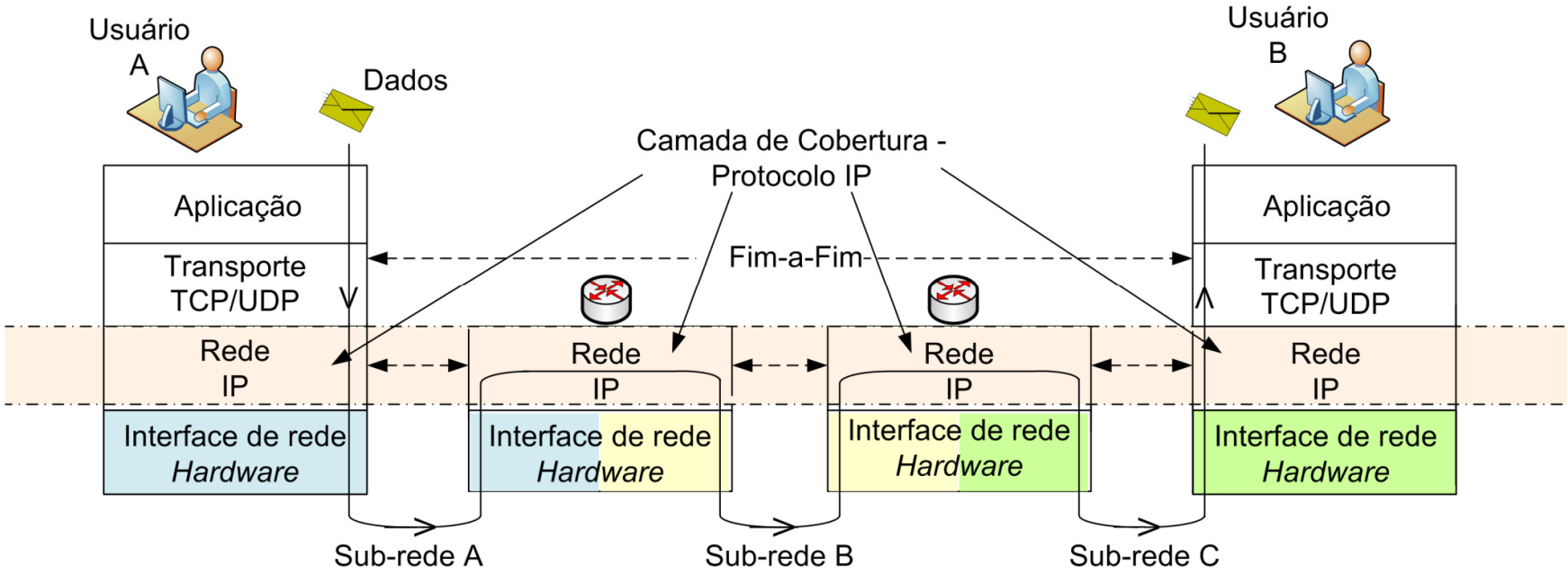


Comunicação Fim-a-Fim



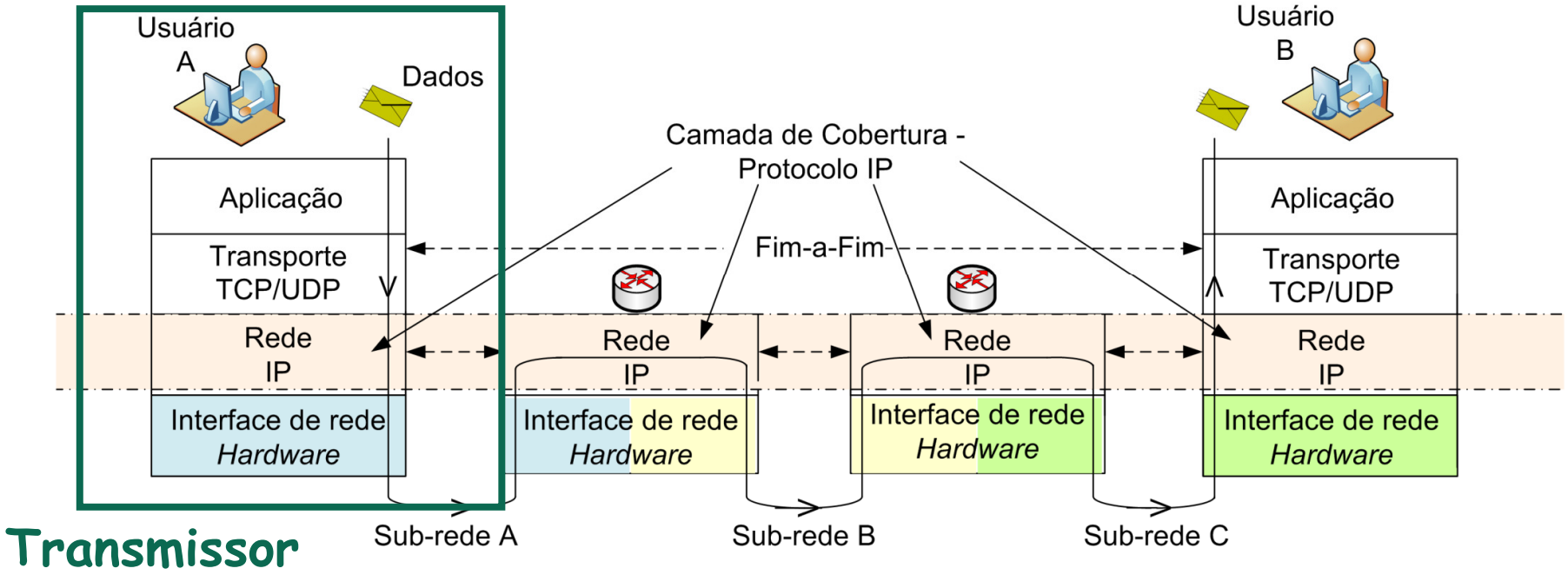
Camada de Transporte

- Protocolos de transporte
  - Executados nos **sistemas finais**



Camada de Transporte

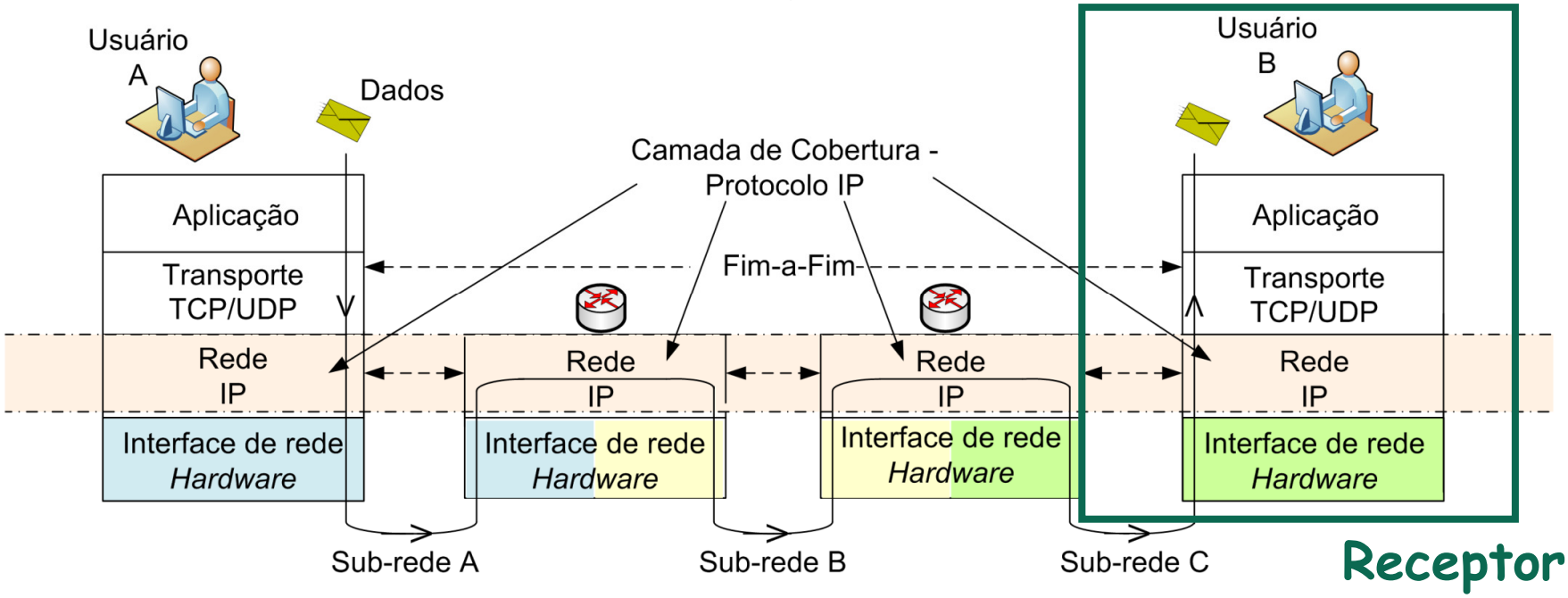
- Protocolos de transporte
  - Executados nos **sistemas finais**



- Converte as mensagens da aplicação em **segmentos**
- Encaminha os segmentos para a camada de rede

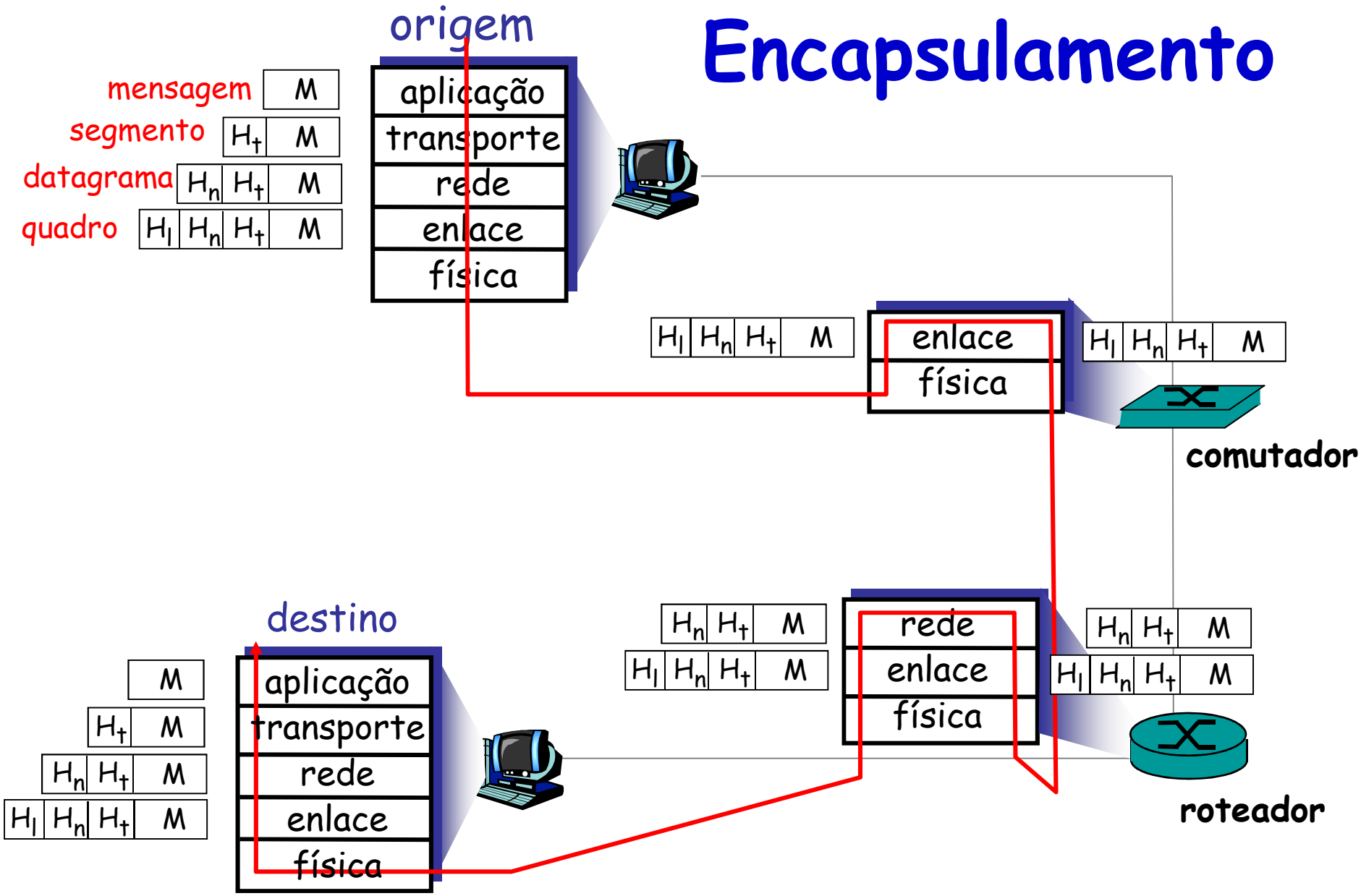
Camada de Transporte

- Protocolos de transporte
  - Executados nos **sistemas finais**



- Recebe os segmentos da camada de rede
- Remonta as mensagens e encaminha para a aplicação

Encapsulamento



Transporte X Rede

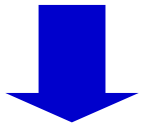
- Camada de transporte
  - Canal lógico de comunicação entre **processos**

depende dos serviços e pode estender os serviços

- Camada de rede
  - Canal lógico de comunicação entre **estações**

Transporte X Rede

- Serviço da camada de rede
  - Entrega de **melhor esforço**
  - Não garante:
    - Entrega dos segmentos
    - Ordenação dos segmentos
    - Integridade dos dados contidos nos segmentos



Serviço não-confiável

Transporte X Rede

- Serviços da camada de transporte
  - Estender o serviço de entrega da camada de rede
    - **Rede**: entre **sistemas finais**
    - **Transporte**: entre **processos** em execução nos sistemas finais
      - Multiplexação e demultiplexação
  - Verificação de integridade
    - Campos de detecção de erros no cabeçalho

Serviços mínimos

Protocolos

- Existem diferentes protocolos de transporte
  - Fornecem diferentes tipos de serviços
  - Aplicações usam o mais adequado ao seu propósito

- Na Internet
  - User Datagram Protocol (UDP)
  - Transmission Control Protocol (TCP)

Protocolos

- UDP
  - Somente os serviços mínimos
    - Entrega não-confiável e não-ordenada
- TCP
  - Mais do que os serviços mínimos
    - Entrega confiável e ordenada
      - Estabelecimento de conexão
      - Controle de congestionamento
      - Controle de fluxo

Protocolos

- UDP
  - Somente os serviços mínimos
    - Entrega não-confiável e não-ordenada
- TCP
  - Mais do que os serviços mínimos
    - Entrega confiável e ordenada
      - Estabelecimento de conexão
      - Controle de congestionamento
      - Controle de fluxo

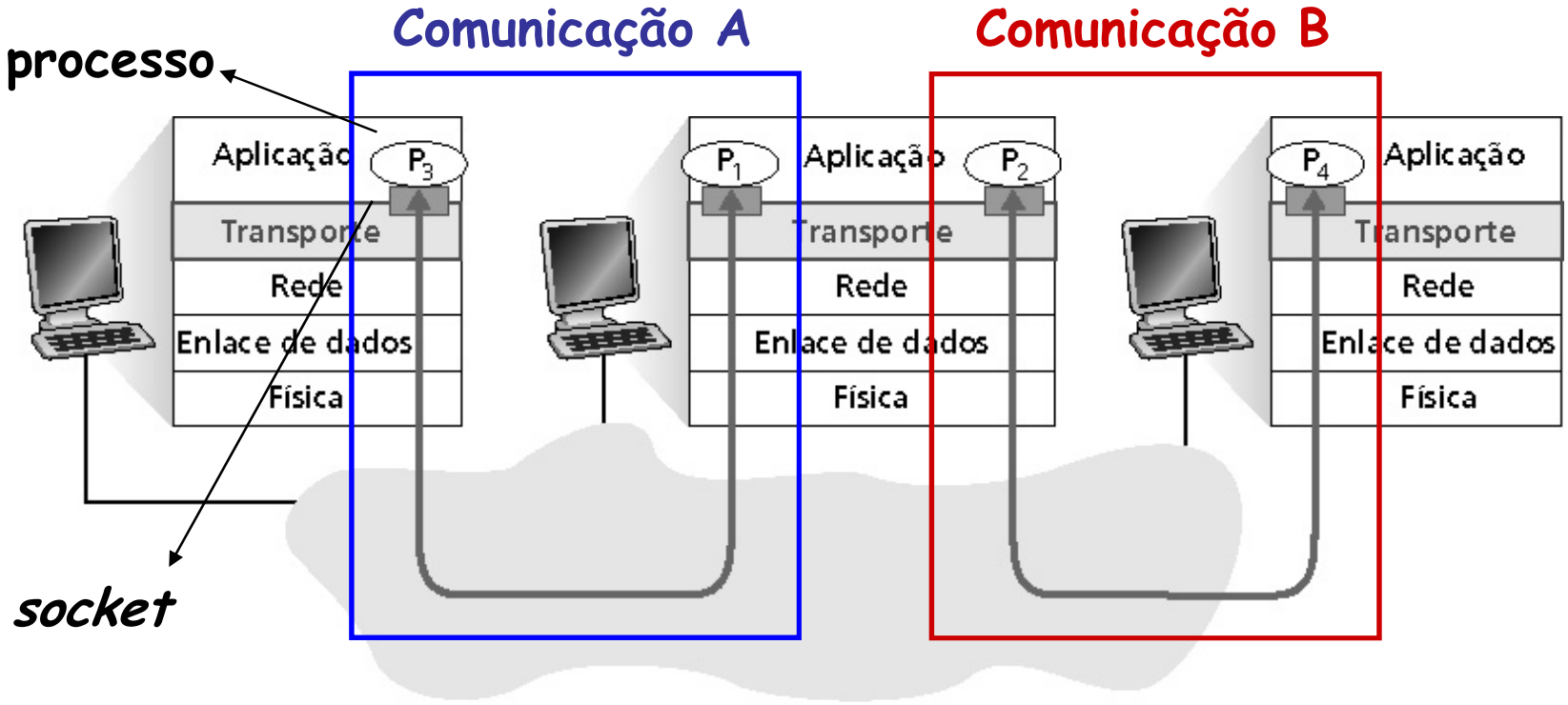
Não garantem requisitos de atraso e de banda passante

(De)Multiplexação

- É um dos serviços mínimos
  - Identificar a qual processo pertence um segmento
  - Encaminhar para o processo correto

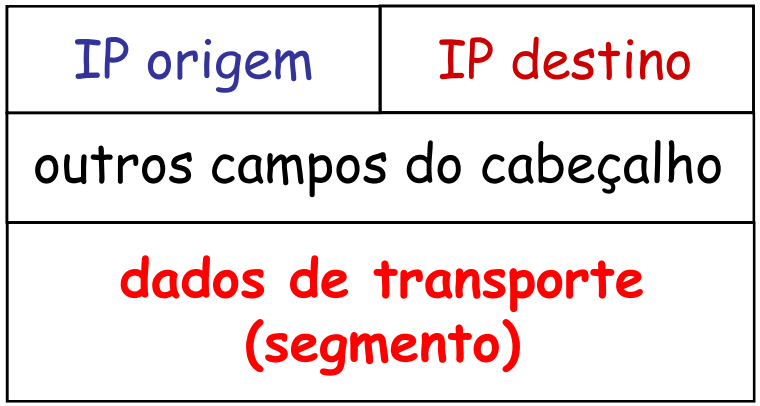
- Socket
  - Interface entre a camada de aplicação e a de transporte dentro de uma máquina

(De)Multiplexação



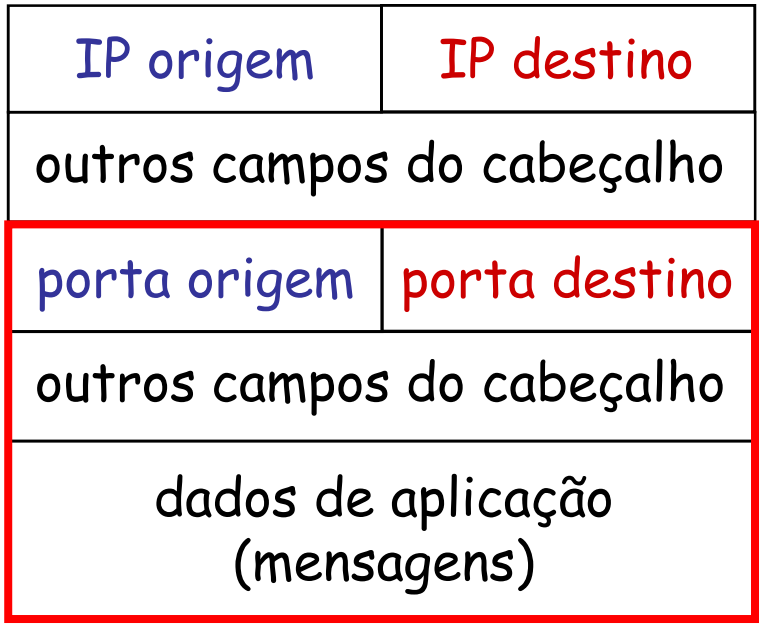
Demultiplexação

- Feita com base nos campos do cabeçalho dos segmentos e datagramas



Demultiplexação

- Feita com base nos campos do cabeçalho dos segmentos e datagramas



Demultiplexação

- Depende do tipo de serviço oferecido pela camada de transporte
  - Orientado à conexão
  - Não-orientado à conexão

Demultiplexação com UDP

- Não-orientada à conexão
- Identificação feita por
  - Endereço IP de **destino**
    - Chegar ao sistema final correspondente
  - Número da porta de **destino**

- Quando o sistema final recebe um segmento UDP:
  1. Verifica o número da porta de destino no segmento
  2. Encaminha o segmento UDP para o socket com aquele número de porta

Demultiplexação com UDP

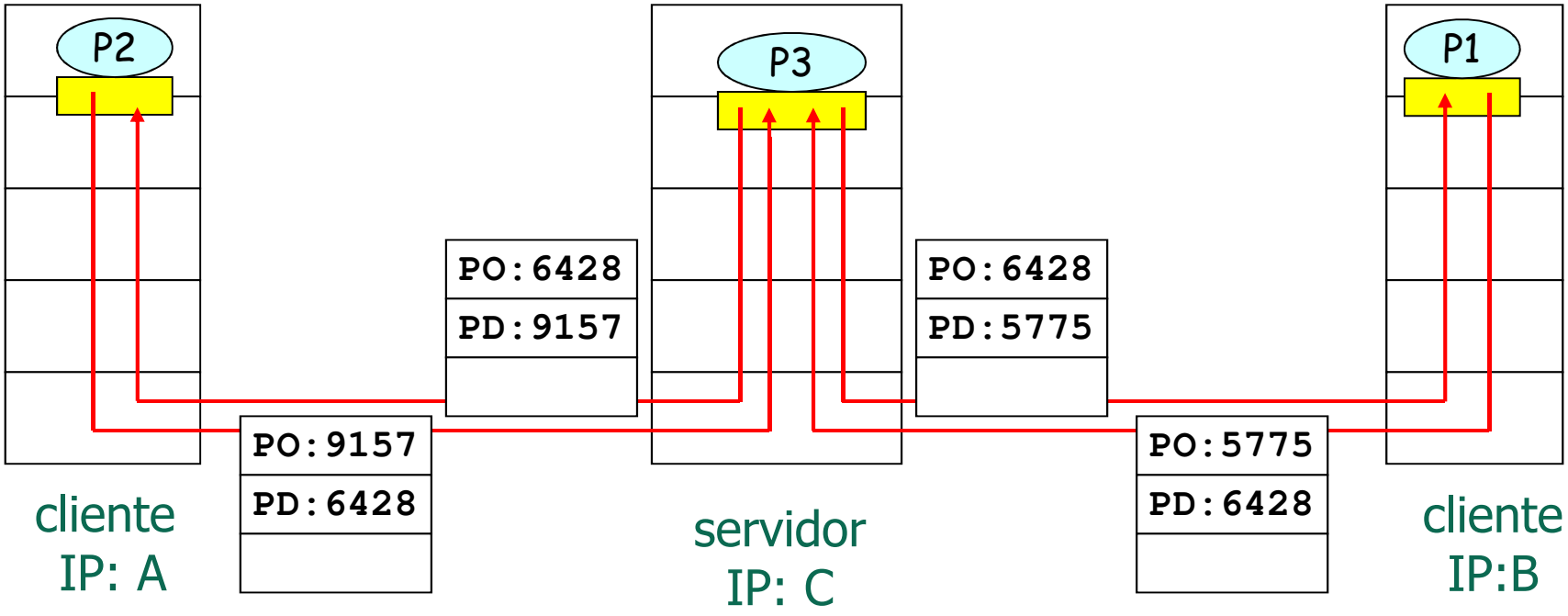
- Um socket pode receber datagramas com **diferentes** endereços **IP origem** e/ou # de **porta de origem**?

Sim!

Somente as informações do destino são usadas e, caso o IP de destino e a porta de destino sejam iguais, o datagrama é encaminhado para o mesmo serviço

Demultiplexação com UDP

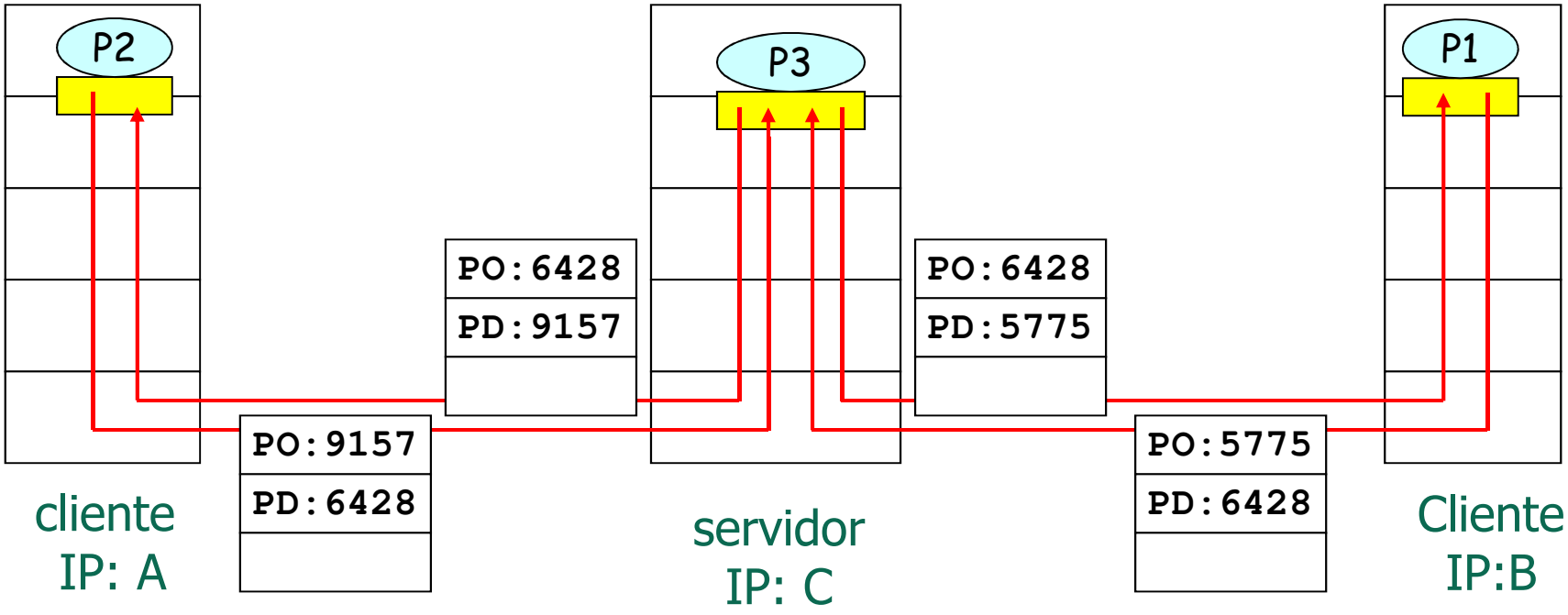
- Como são usadas as informações de origem?



Porta de origem é usada apenas como "endereço de retorno", caso seja necessário

Demultiplexação com UDP

- Como são usadas as informações de origem?



O mesmo processo P3 atende os processos P1 e P2 em estações finais distintas

Demultiplexação com TCP

- Orientada à conexão
- Identificação feita por
  - Endereço IP de **origem**
  - Número da porta de **origem**
  - Endereço IP de **destino**
  - Número da porta de **destino**
- Quando o hospedeiro recebe um segmento TCP
  - Verifica o número das portas de origem e destino no segmento
  - Encaminha o segmento TCP para o socket correspondente à tupla (Ip_{orig}, $Porta_{orig}$, Ip_{dest}, $Porta_{dest}$)

Demultiplexação com TCP

- Um socket (de conexão TCP) pode receber segmentos com **diferentes** endereços **IP origem** e/ou # de **porta de origem**?

Não!

Cada segmento será direcionado para um *socket* específico

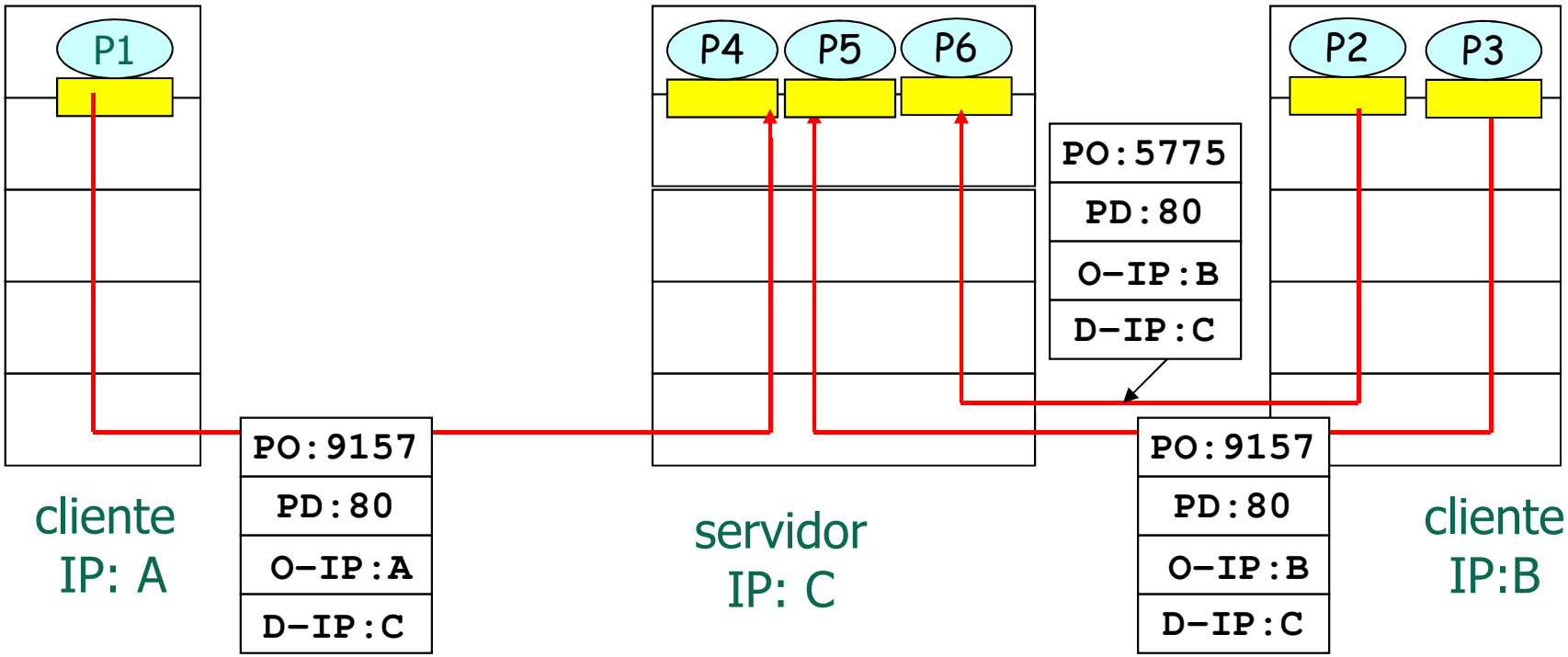
(Exceção: socket receptivo)

Demultiplexação com TCP

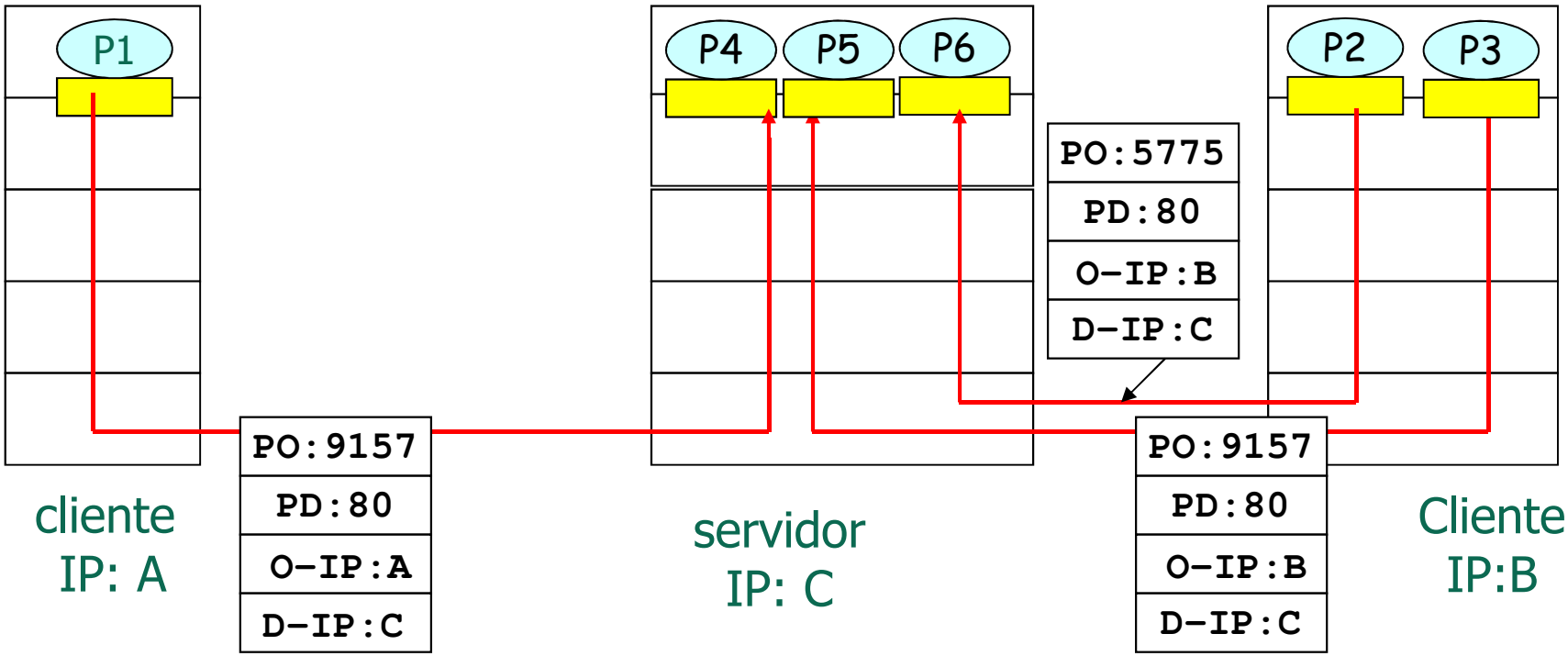
- Um servidor pode dar suporte a muitos sockets TCP simultâneos
  - Cada socket é identificado pela sua própria quádrupla

- Servidores Web têm sockets diferentes para cada conexão cliente
  - HTTP não persistente terá sockets diferentes para cada pedido

Demultiplexação com TCP



Demultiplexação com TCP



Os processos se comunicam aos pares, sendo que todos se comunicam a processos distintos

User Datagram Protocol (UDP)

UDP

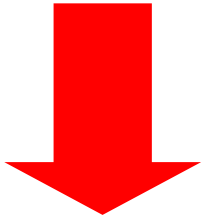
- Definido pela RFC 768
- Protocolo de transporte mínimo
  - Oferece multiplexação e detecção de erros
- Segmentos UDP podem ser:
  - Perdidos
  - Entregues à aplicação fora de ordem
- Não orientado à conexão
  - Não há conexão entre remetente e o receptor
  - Tratamento independente de cada segmento UDP

UDP

- Quais as vantagens?
  - Elimina o estabelecimento de conexão
    - Menor latência
  - É simples
    - Não mantém "estado" da conexão nem no remetente, nem no receptor
  - Cabeçalho de segmento reduzido
  - Não há controle de congestionamento
    - UDP pode transmitir tão rápido quanto desejado (e possível)

UDP

- Se ao retirar o controle de congestionamento o UDP pode transmitir o mais rápido possível...
  - Por que não criar apenas aplicações sobre o UDP?



1. A rede pode se tornar totalmente congestionada
2. Protocolos com controle de congestionamento podem ter suas taxas reduzidas drasticamente

UDP

- Utilizado para aplicações multimídias
  - Tolerantes a perdas
  - Sensíveis à taxa de transmissão
- Outros usos
  - DNS → Reduzir a latência na requisição de páginas Web
  - SNMP → Reduzir o tempo de reação a um problema na rede
- Transferência confiável com UDP?
  - É necessário acrescentar confiabilidade na camada de aplicação
    - Recuperação de erro específica para cada aplicação

Requisitos das Aplicações

Aplicação	Perda	Banda passante	Atraso
Transferência de arquivos	sem perdas	elástica	tolerante
Email	sem perdas	elástica	tolerante
Web	sem perdas	elástica	tolerante
Áudio/vídeo em tempo real	Tolerante	áudio: 5kb-1Mb vídeo:10kb-5Mb	centenas de miliseg.
Áudio/vídeo gravado	tolerante	Idem	poucos seg.
Jogos interativos	Tolerante	até 10 kbps	centenas de miliseg.
Mensagens instantâneas	sem perdas	elástica	sim/não (?)

Protocolos por Aplicação

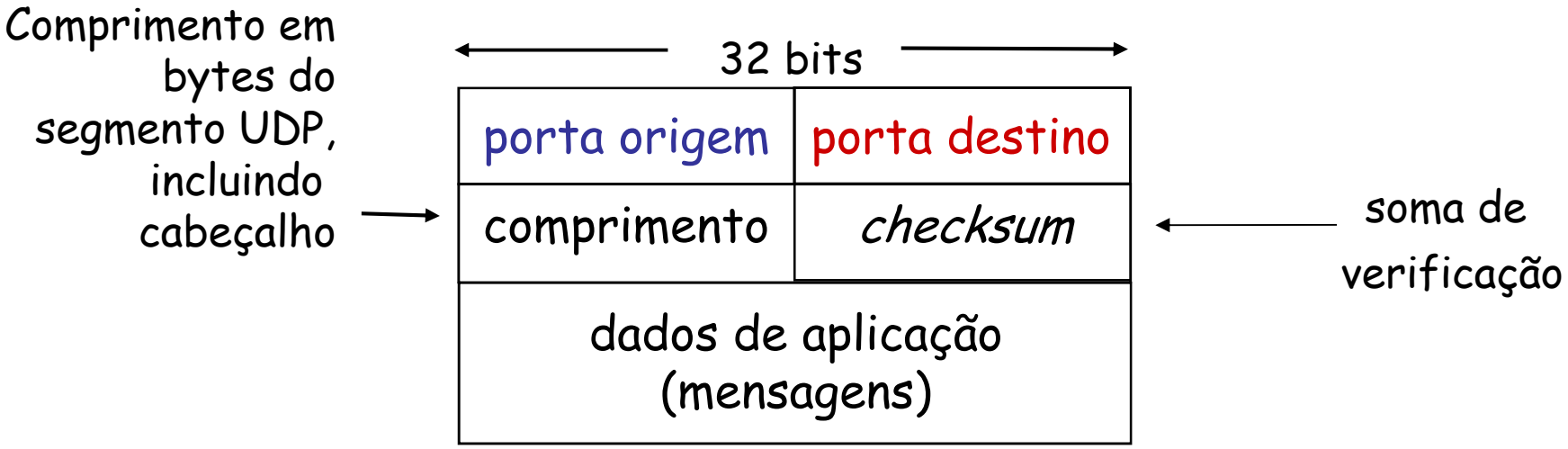
Aplicação	Protocolo de aplicação	Protocolo de transporte
Servidor de arquivos remoto	NFS	Tipicamente UDP
Gerenciamento de rede	SNMP	Tipicamente UDP
Protocolo de roteamento	RIP	Tipicamente UDP
Tradução de nomes	DNS	Tipicamente UDP

Protocolos por Aplicação

Aplicação	Protocolo de aplicação	Protocolo de transporte
Email	SMTP	TCP
Acesso remoto	Telnet, SSH	TCP
Web	HTTP	TCP
Transferência de arquivos	FTP	TCP
Distribuição multimídia	HTTP, RTP	TCP ou UDP
Telefonia na Internet	SIP, RTP, proprietário (Skype)	TCP ou UDP

Segmento UDP

- Formato do segmento
  - Cabeçalho de 8 bytes



Checksum (Soma de Verificação)

- Usada para detectar "erros" no segmento transmitido
  - Ex.: bits trocados

Transmissor:

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

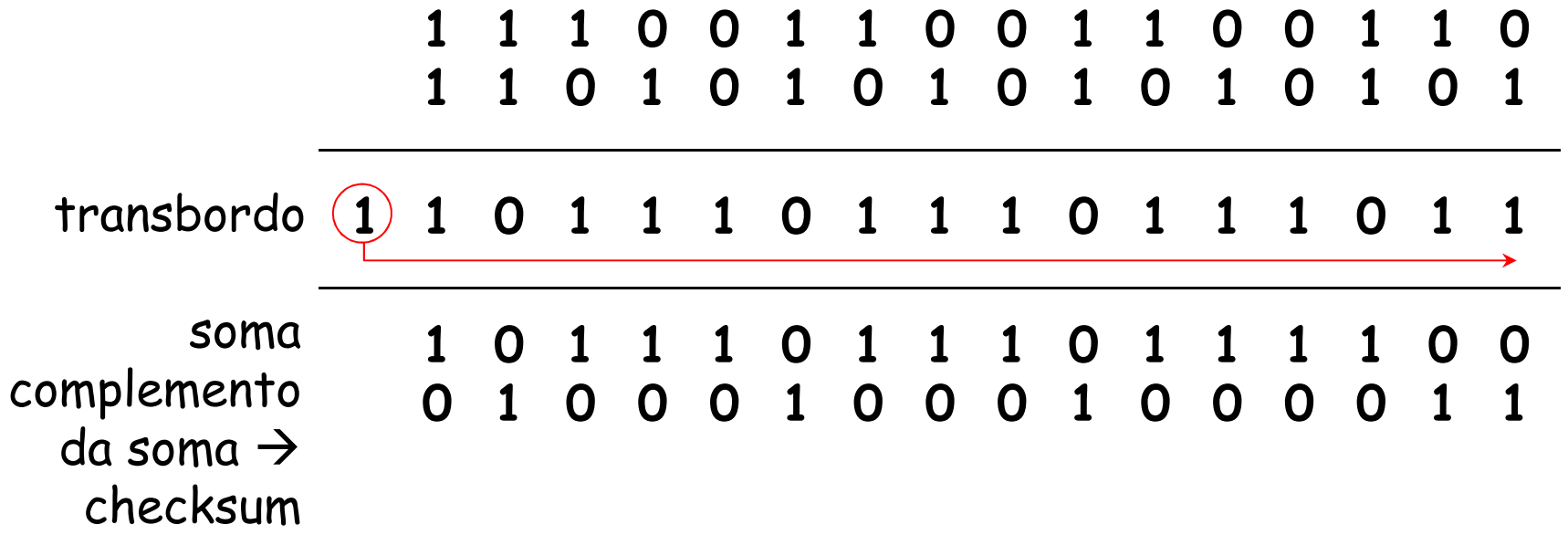
Receptor:

- calcula checksum do segmento recebido
- verifica se checksum computado é tudo um 'FFFF':
  - **NÃO - erro detectado**
  - **SIM - nenhum erro detectado**

Exemplo do Cálculo do Checksum

- Ao adicionar números
  - O transbordo (vai um) do bit mais significativo deve ser adicionado ao resultado

- Exemplo: adição de dois inteiros de 16-bits



Papel da Detecção de Erro

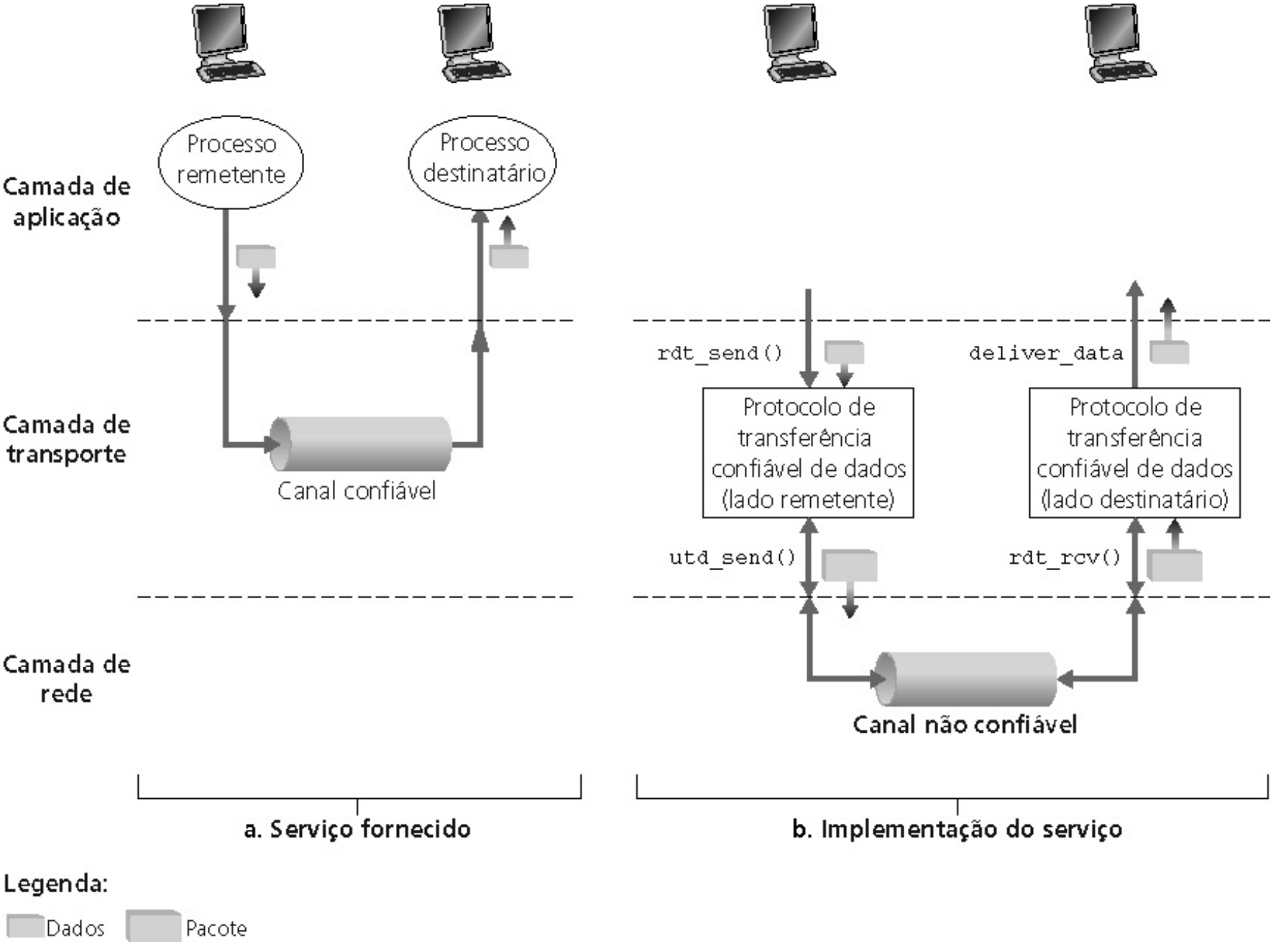
- Não corrige o erro
  - Uso do checksum ainda não é infalível...
    - Mas já é uma iniciativa na direção da confiabilidade

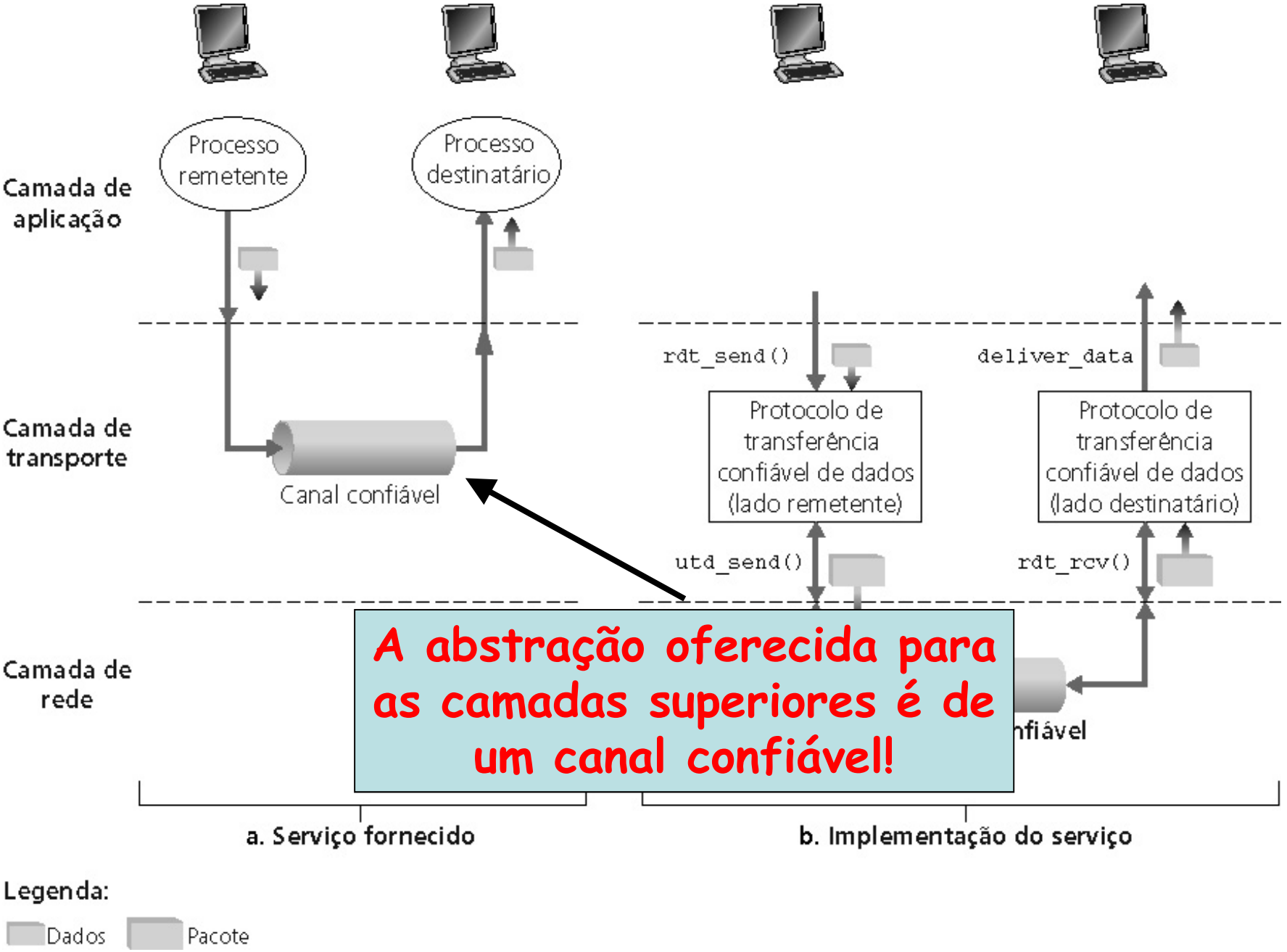
O que mais pode ser feito?

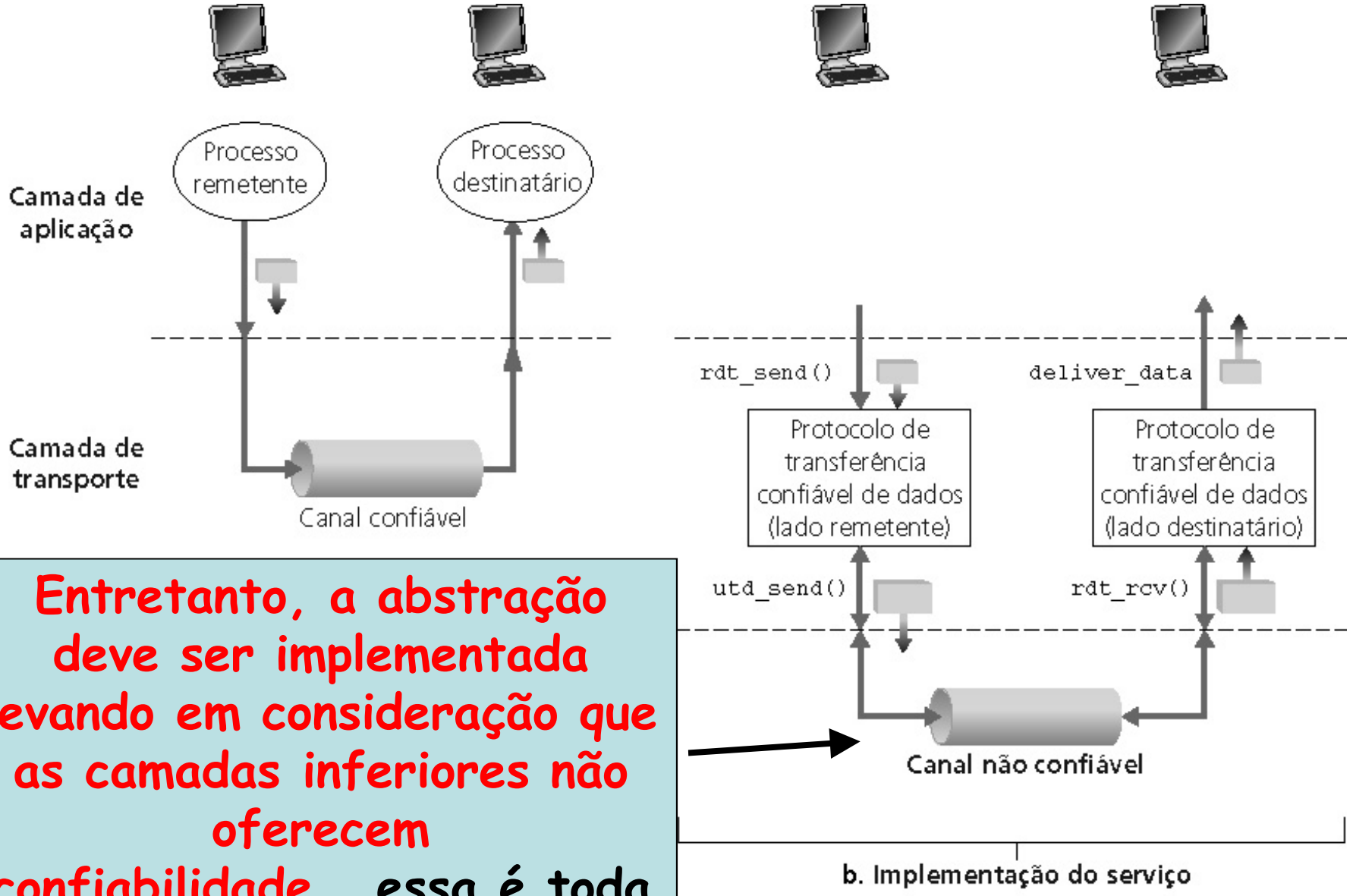
Transferência Confiável: Princípios

Transferência Confiável

- Importante nas camadas de transporte, enlace, etc.
  - 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 (reliable data transfer - rdt)


Camada de aplicação
Camada de transporte
Camada de rede
Processo remetente
Processo destinatário
Canal confiável
rdt_send()
deliver_data
Protocolo de transferência confiável de dados (lado remetente)
Protocolo de transferência confiável de dados (lado destinatário)
utd_send()
rdt_rcv()
Canal não confiável
a. Serviço fornecido
b. Implementação do serviço
Legenda:
Dados
Pacote


Camada de aplicação
Camada de transporte
Camada de rede
Processo remetente
Processo destinatário
Canal confiável
rdt_send()
deliver_data
Protocolo de transferência confiável de dados (lado remetente)
Protocolo de transferência confiável de dados (lado destinatário)
utd_send()
rdt_rcv()
A abstração oferecida para as camadas superiores é de um canal confiável!
a. Serviço fornecido
b. Implementação do serviço
Legenda:
Dados
Pacote

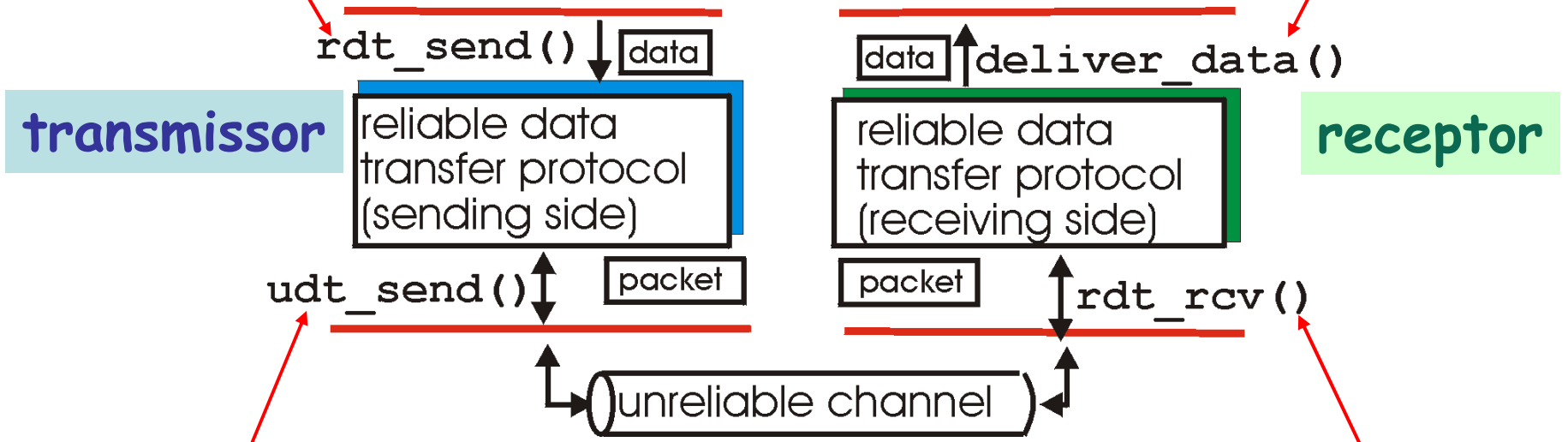


Entretanto, a abstração deve ser implementada levando em consideração que as camadas inferiores não oferecem confiabilidade...essa é toda a complexidade!

Transferência Confiável

rdt_send(): chamada de cima, (ex.: pela apl.). Passa dados p/ serem entregues à camada sup. do receptor

deliver_data(): chamada pela entidade de transporte p/ entregar dados p/ camada superior



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

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

Transferência Confiável

- O que é um **canal confiável**?
  - Nenhum dado transmitido é **corrompido**
  - Nenhum dado transmitido é **perdido**
  - Todos os dados são entregues **ordenadamente**

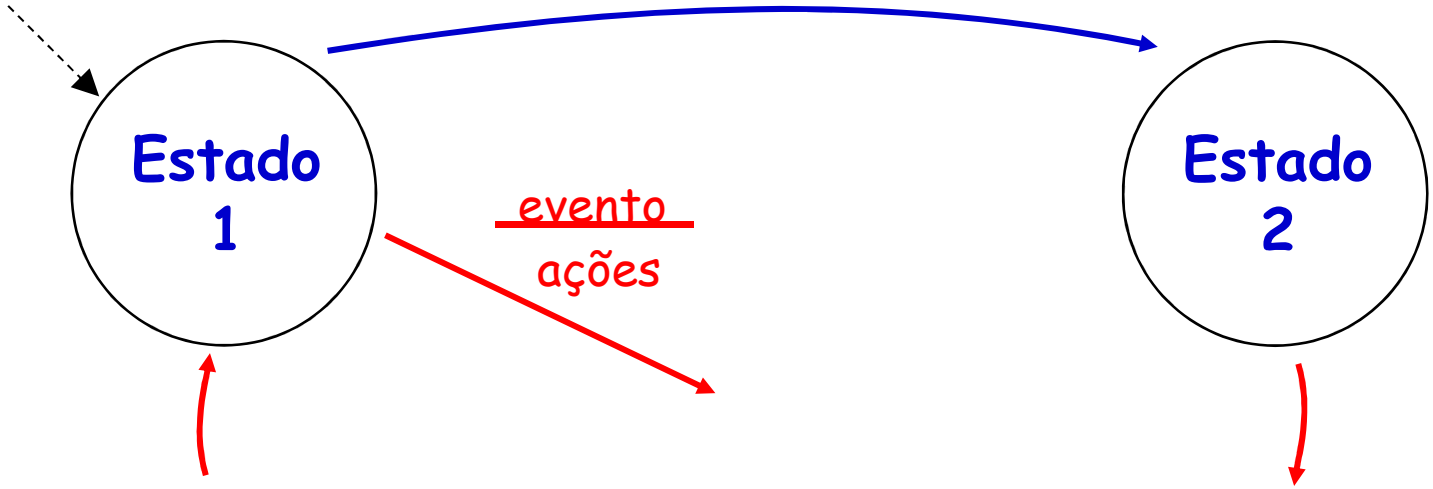
- Protocolo de transferência confiável de dados
  - Responsável por implementar um canal confiável

Transferência Confiável

- Quais os mecanismos usados para prover um canal confiável?
  - Serão vistos "passo-a-passo"
- O desenvolvimento de um protocolo confiável para transferência de dados é incremental
  - Tanto do lado transmissor quanto receptor
- Os fluxos podem ser considerados unidirecionais
  - Apesar das informações de controle fluírem em ambos os sentidos
- As máquinas de estados finitos (FSM) devem ser usadas na especificação dos protocolos

Transferência Confiável

Seta tracejada indica o estado inicial da FSM



estado:

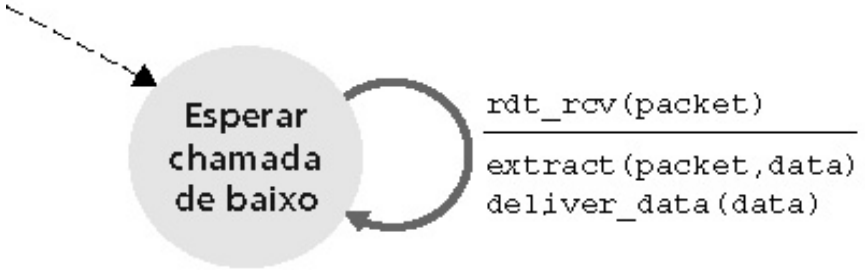
o próximo estado é determinado unicamente pelo próximo evento

Símbolo Λ: Indica ausência de evento ou ação

Canal Totalmente Confiável

- Protocolo **rdt 1.0**
- Canal de transmissão perfeitamente confiável
  - não há erros de bits
  - não há perda de pacotes
- FSMs separadas para transmissor e receptor
  - transmissor envia dados pelo canal subjacente
  - receptor lê os dados do canal subjacente

a. rdt1.0: lado remetente

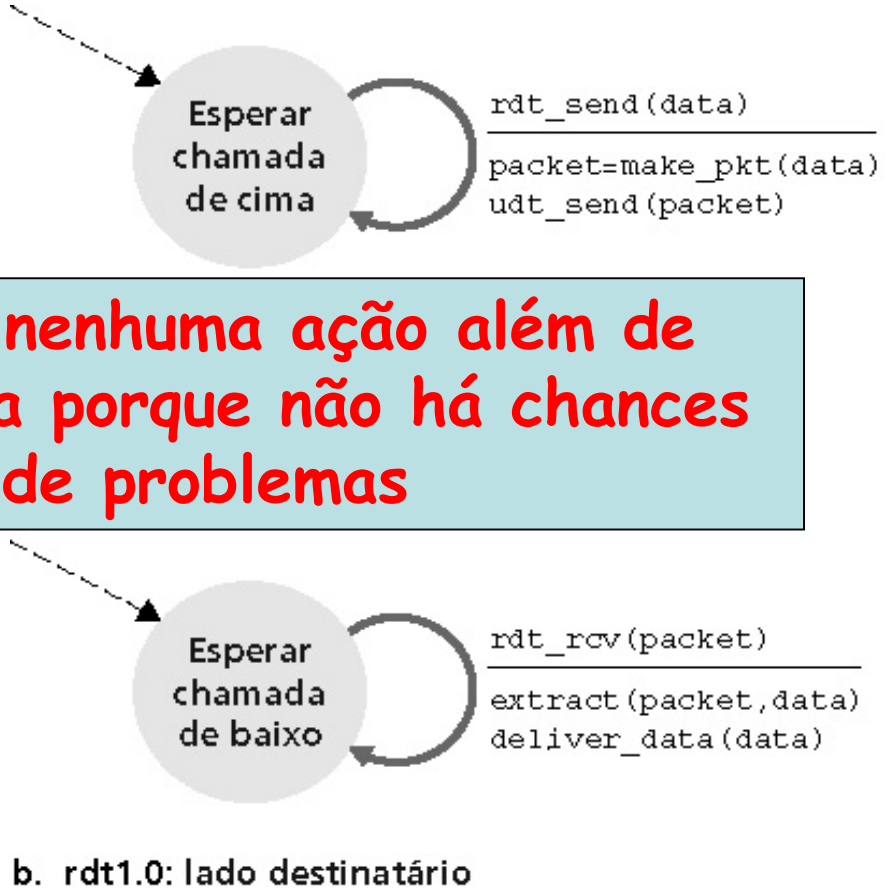


b. rdt1.0: lado destinatário

Canal Totalmente Confiável

- Protocolo **rdt 1.0**
- Canal de transmissão perfeitamente confiável
  - não há erros de bits
-
  - transmissor envia dados pelo canal subjacente
  - receptor lê os dados do canal subjacente

O receptor não realiza nenhuma ação além de enviar os dados para cima porque não há chances de ocorrência de problemas



b. rdt1.0: lado destinatário

Canal com Erros

- Protocolo rdt 2.0
- Canal pode trocar valores dos bits num pacote
  - É necessário detectar os erros: checksum

- Como recuperar esses erros?
  - Enviando retransmissões (*Automatic Repeat reQuest* – ARQ)
    - **Reconhecimentos positivos (ACKs)**
      - Receptor avisa o transmissor a recepção correta de um pacote
    - **Reconhecimentos negativos (NAKs)**
      - Receptor avisa o transmissor que o pacote tinha erros
        - » Transmissor reenvia o pacote ao receber um NAK

Canal com Erros

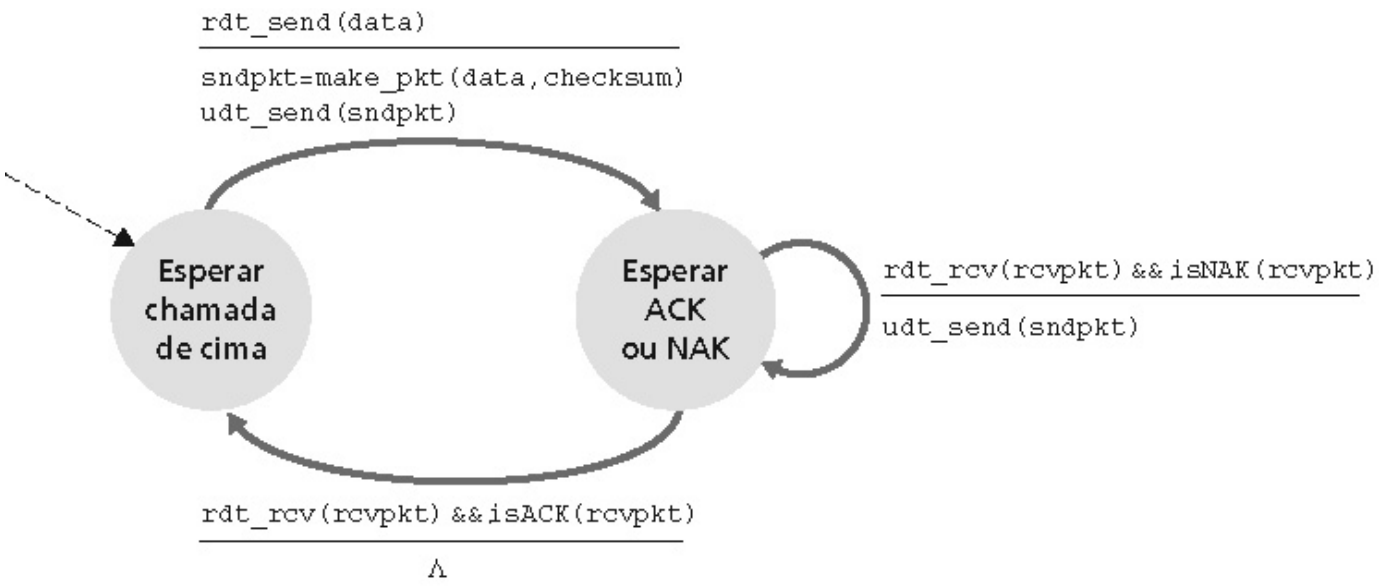
- Se o canal tem erros...

Detecção de erros

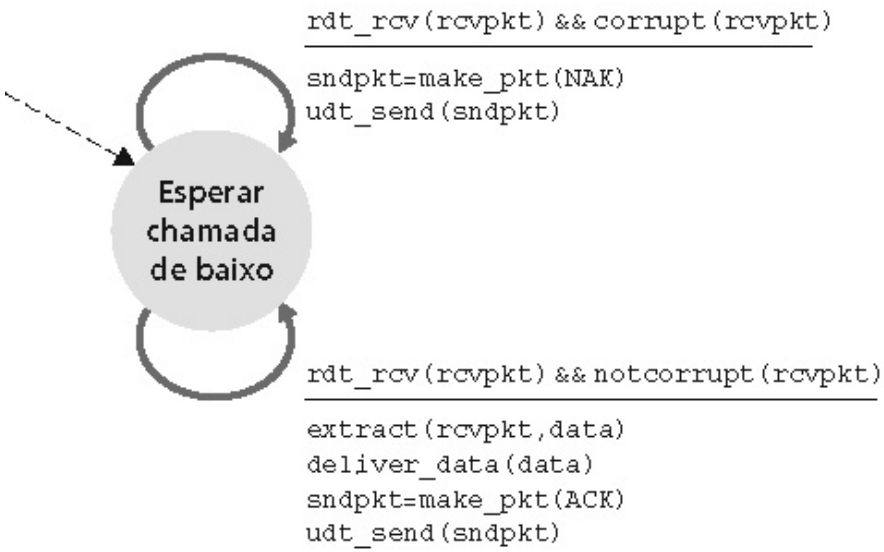
+

Mecanismo de pedido de repetição automática (ARQ - *Automatic Repeat Request*)

Canal com Erros

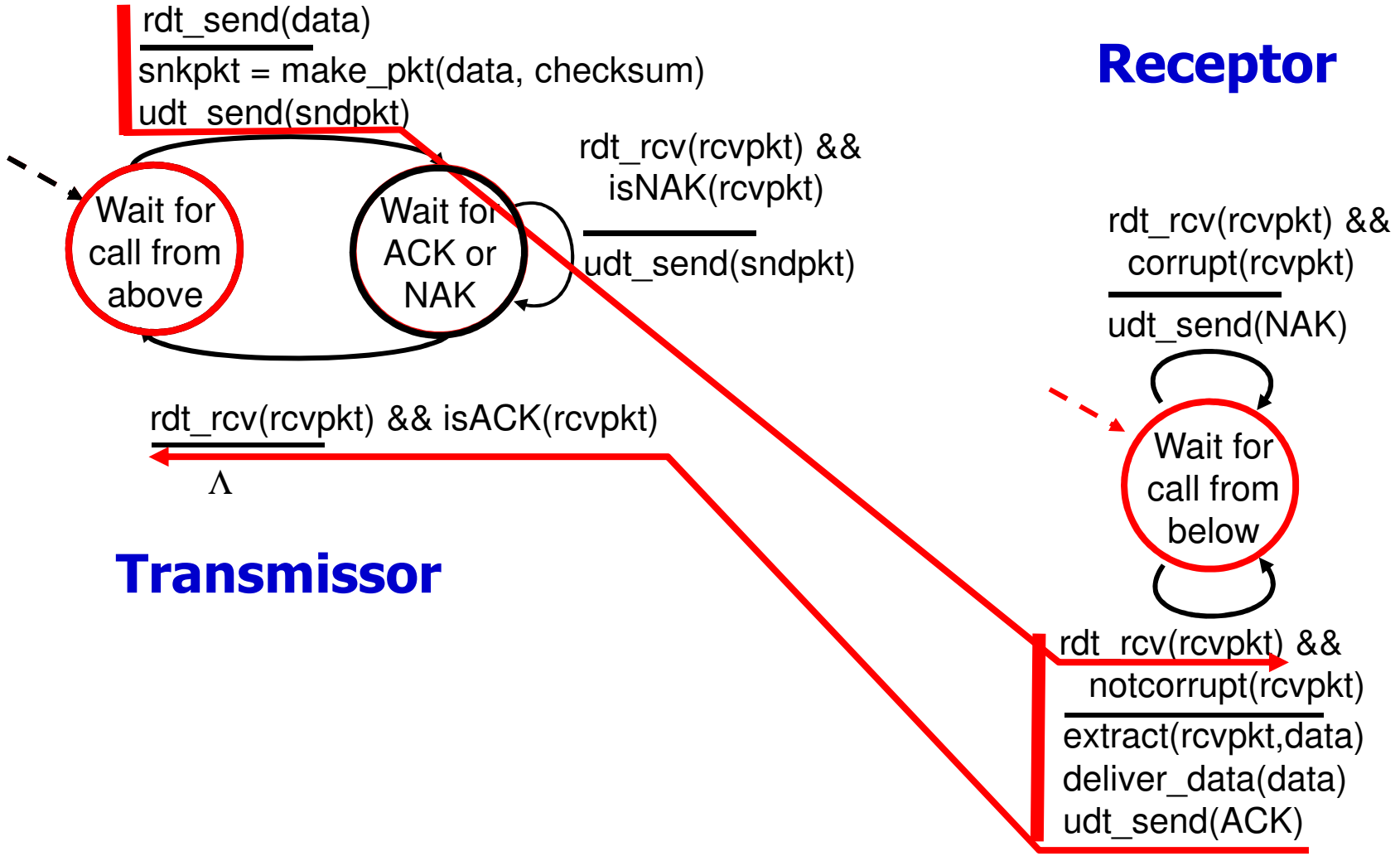


a. rdt2.0: lado remetente

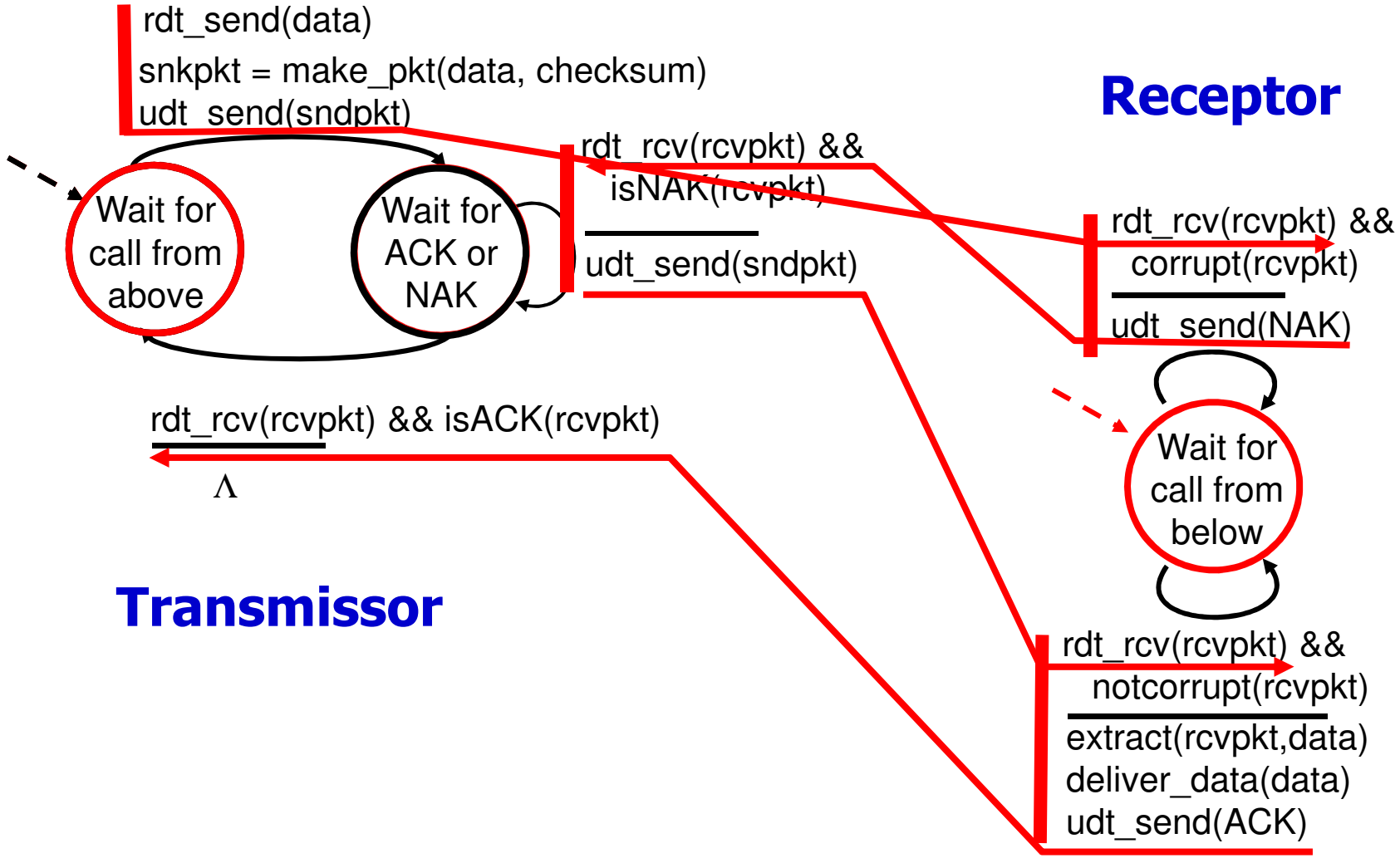


b. rdt2.0: lado destinatário

Canal com Erros: Operação Normal



Canal com Erros: Operação com Erros



Problema das Retransmissões em um Canal com Erros

- E se o ACK/NAK for corrompido?
  - Transmissor não sabe o que se passou no receptor
- O que fazer?
  - Retransmitir
    - Pode causar retransmissão de pacote corretamente recebido
      - **Possibilidade de recepção de pacotes duplicados**
  - Usar ACKs/NAKs para cada ACK/NAK do receptor
    - E se perder ACK/NAK do remetente?
      - **O processo não teria fim!**

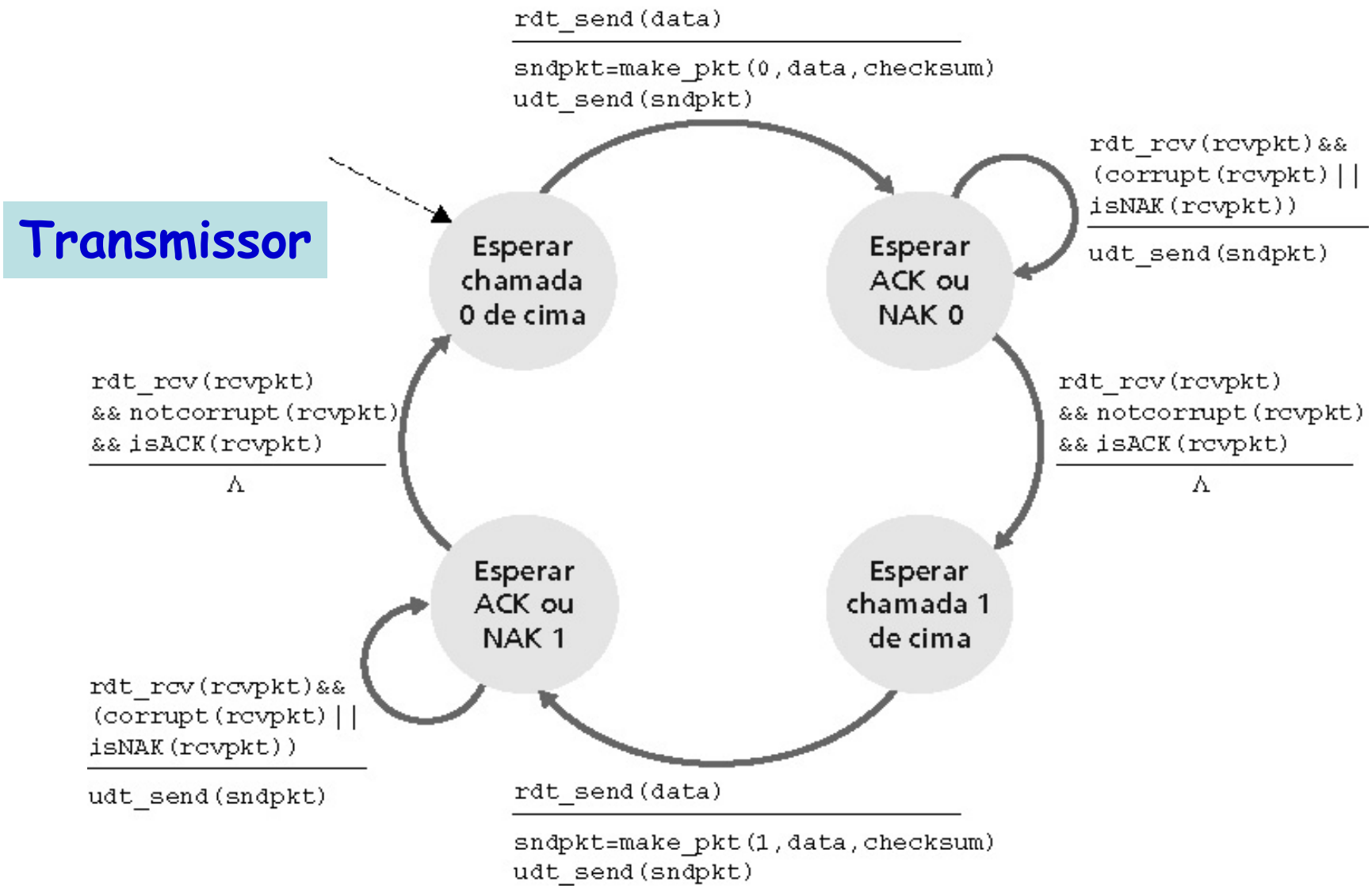
Problema das Retransmissões em um Canal com Erros

- **Problema dos pacotes duplicados...**
  - Transmissor inclui um número de sequência por pacote
    - Receptor pode detectar pacotes duplicados e descartá-los sem entregar para a aplicação
- **Problema do envio de ACK/NAK para ACK/NAK recebido com erro...**
  - Transmissor sempre retransmite o último pacote se ACK/NAK chegar com erro

Protocolos onde o transmissor envia um pacote, e então aguarda resposta do receptor

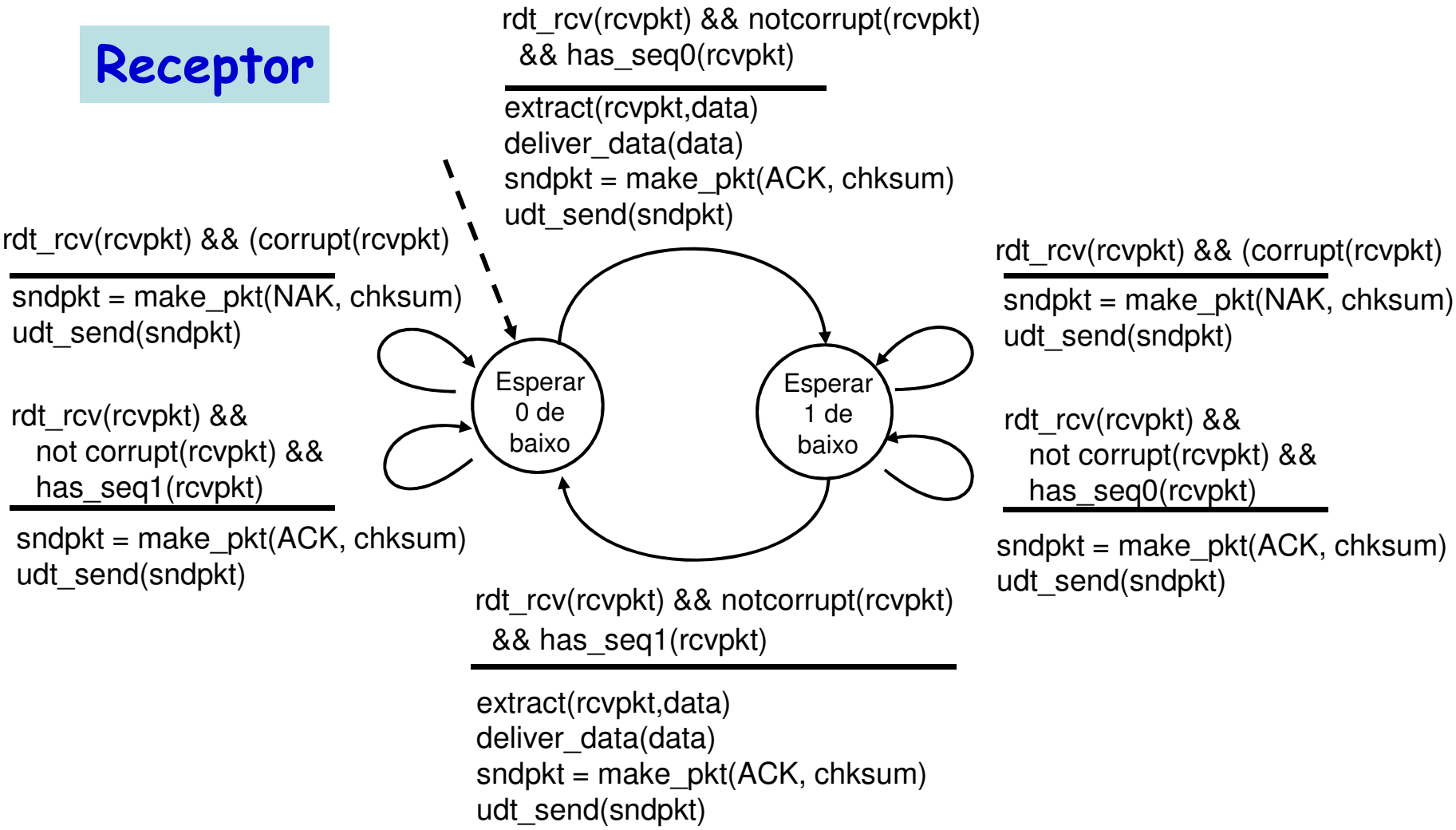
Protocolos do tipo para-e-espera (*stop-and-wait*)

Canal com Erros: ACK/NAK corrompidos



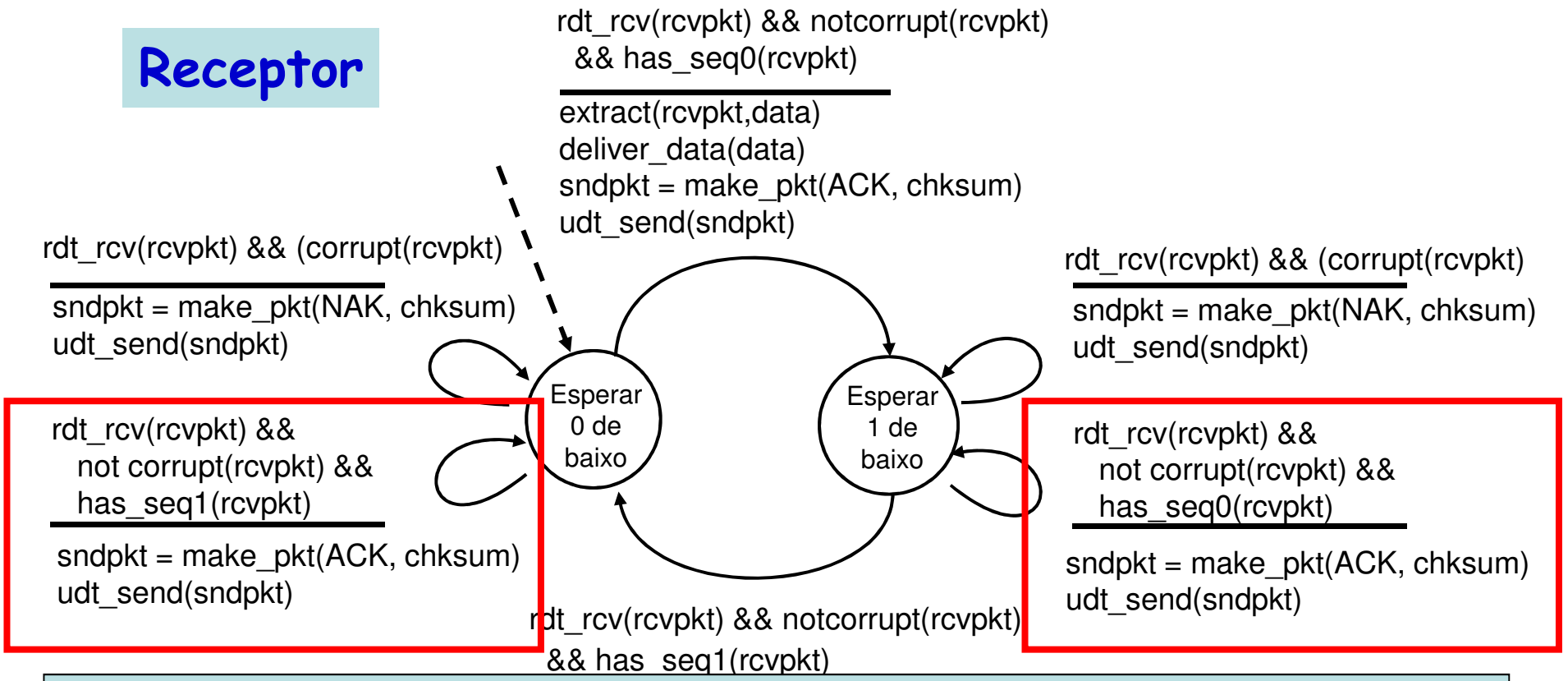
Canal com Erros: ACK/NAK corrompidos

Receptor



Canal com Erros: ACK/NAK corrompidos

Receptor



Recepção de segmentos duplicados, ACK anterior é repetido já que foi provavelmente perdido pelo transmissor

Canal com Erros: ACK/NAK corrompidos

Transmissor:

- Insere número de sequência no pacote
  - Um bit de número de sequência é suficiente
    - Bit comparado com o da transmissão anterior pode identificar se o pacote é duplicado
      - **Funcionamento stop-and-wait**
- Após envio do pacote...
  - Espera por ACK/NAK e verifica se ACK/NAK estão corrompidos

Duplicou o número de estados: transmissor deve "lembrar" se o número de sequência do pacote atual é 0 ou 1...

Canal com Erros: ACK/NAK corrompidos

Receptor:

- Verifica se o pacote recebido é duplicado
  - Estado indica se número de sequência esperado é 0 ou 1

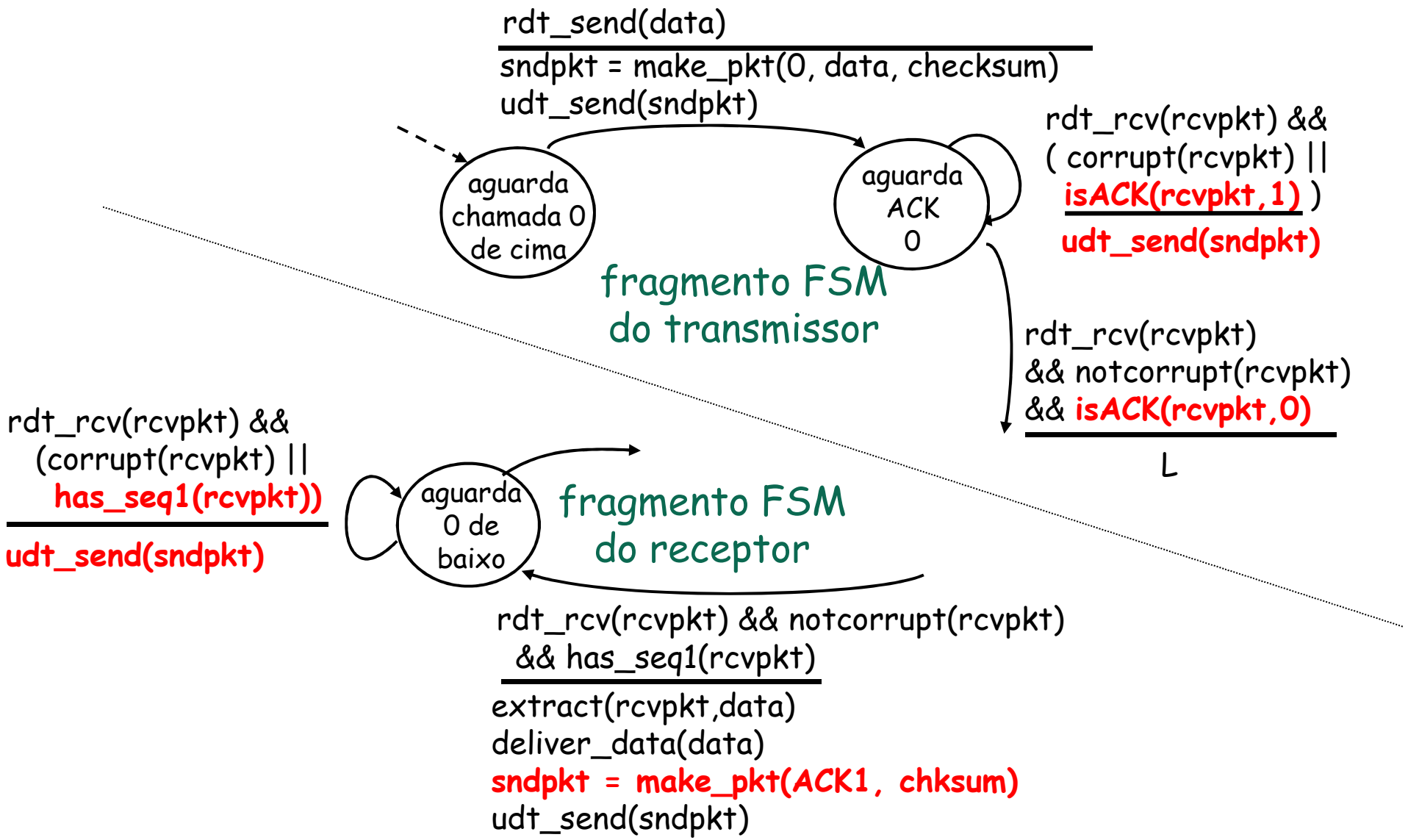
- Obs.:
  - Receptor não tem como saber se último ACK/NAK foi recebido bem pelo transmissor
    - (ACK/NAK não são identificados com no. de seq.)

Canal com Erros: Sem NAK

- Mesma funcionalidade usando apenas ACKs
  - Ao invés de NAK, receptor envia ACK para último pacote recebido sem erro

- Receptor deve incluir explicitamente o número de sequência do pacote reconhecido
  - Assim, **ACKs duplicados** no Transmissor resultam na **mesma ação do NAK**
    - **Retransmissão do pacote corrente!**

Canal com Erros: Sem NAK



Canal com Erros e Perdas

- Canal de transmissão também pode perder pacotes
  - Dados ou ACKs
- Checksum, número de sequência, ACKs e retransmissões podem ajudar...
  - Mas ainda não são suficientes

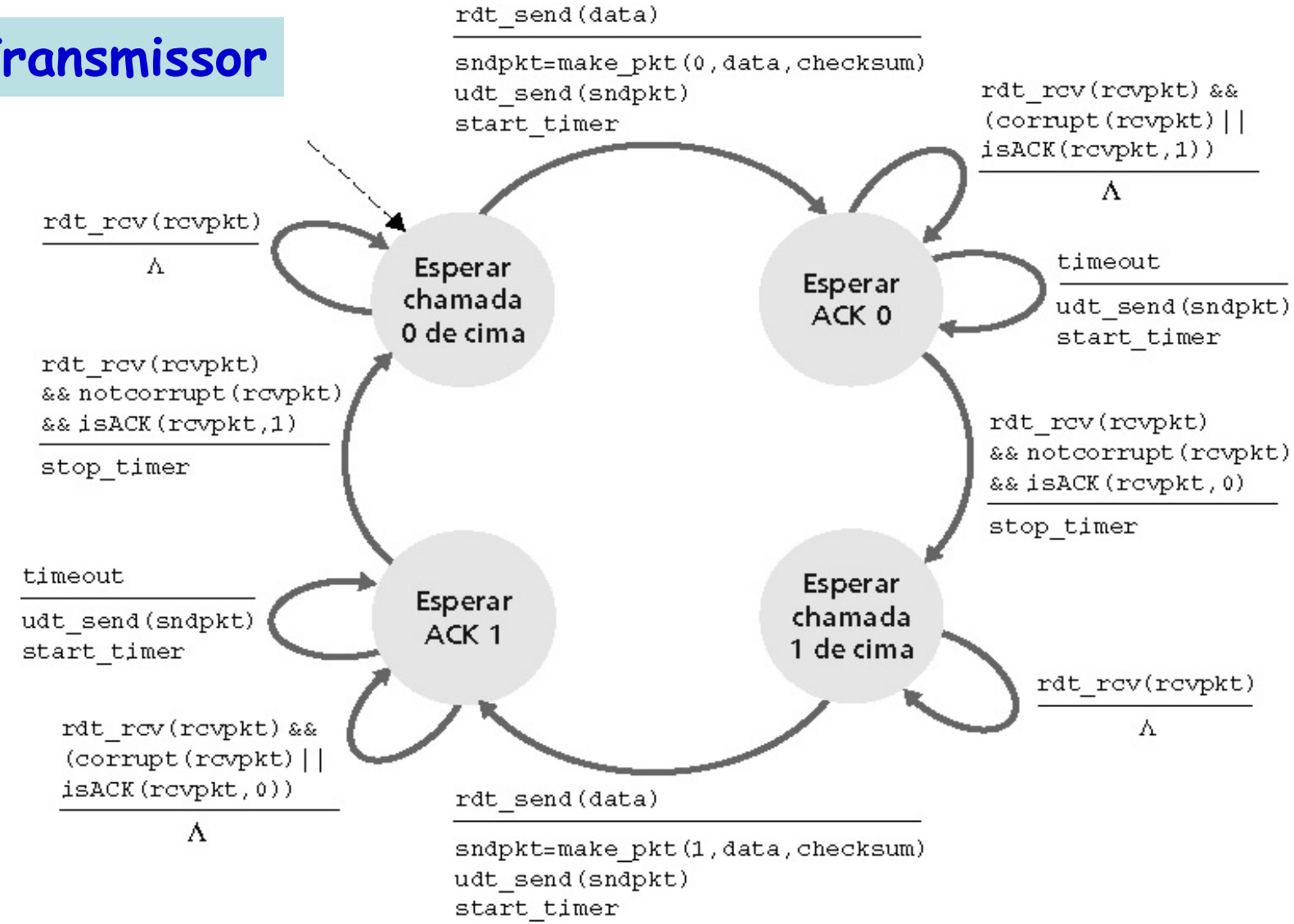
- Como lidar com as perdas?
  - Transmissor **espera** até ter certeza que um pacote ou um ACK foi perdido
    - **Então retransmite**

Canal com Erros e Perdas

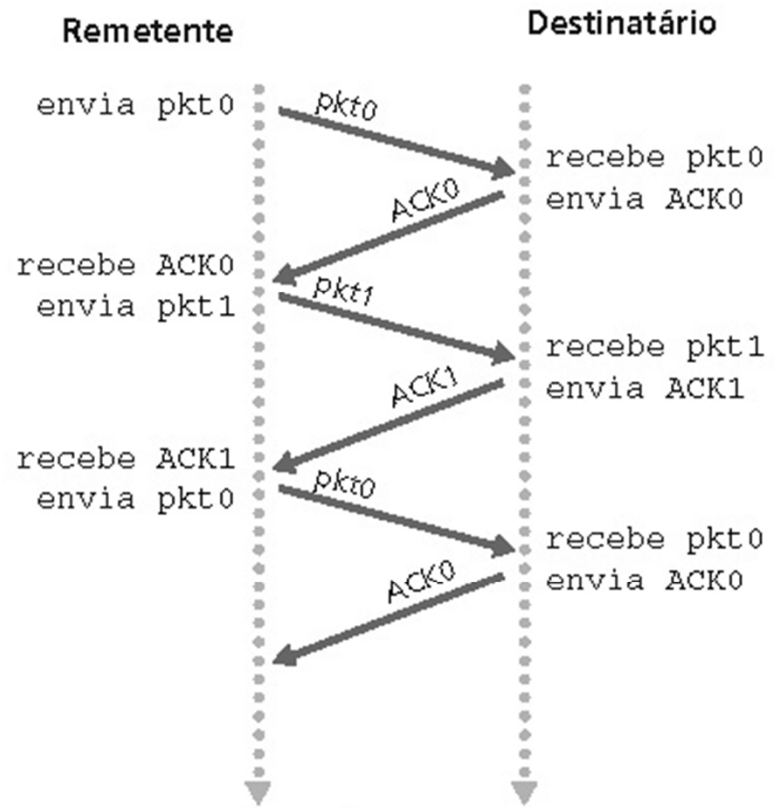
- Transmissor aguarda um tempo "razoável" pelo ACK
  - Retransmite se nenhum ACK for recebido neste intervalo
  - Se pacote (ou ACK) estiver apenas atrasado (e não perdido)
    - Retransmissão será duplicada, mas uso de número de sequência já identifica esse caso
    - Receptor deve especificar o número de sequência do pacote sendo reconhecido
  - Requer o uso de **temporizador**

Canal com Erros e Perdas

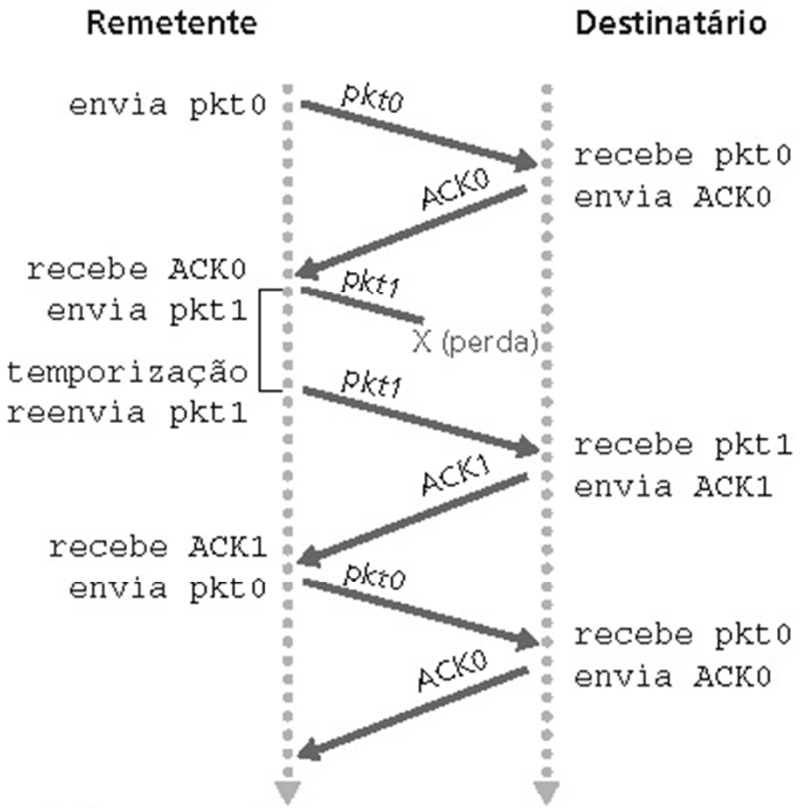
Transmissor



Canal com Erros e Perdas

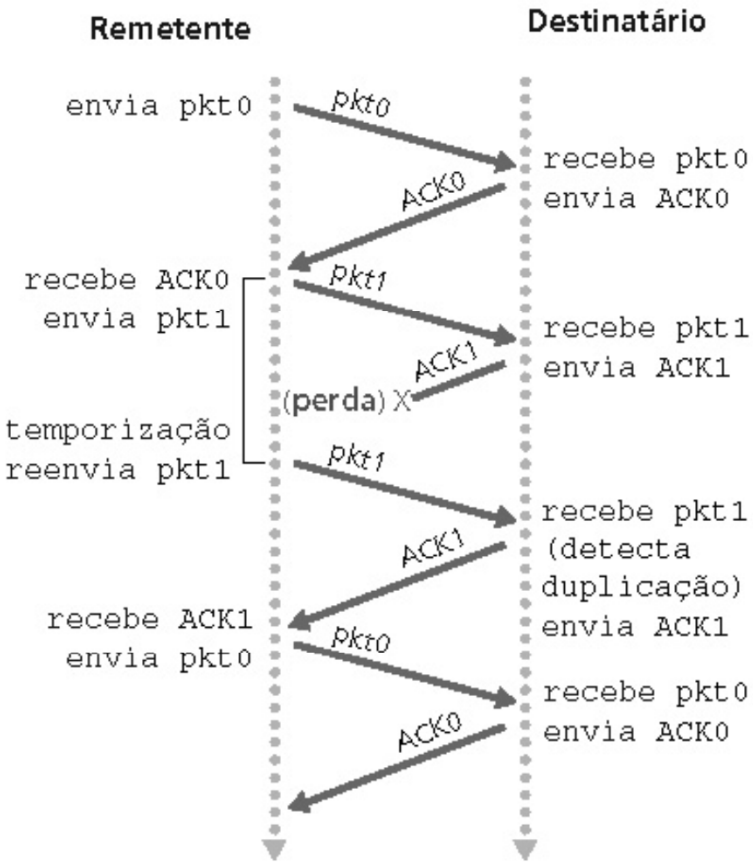


a. Operação sem perda

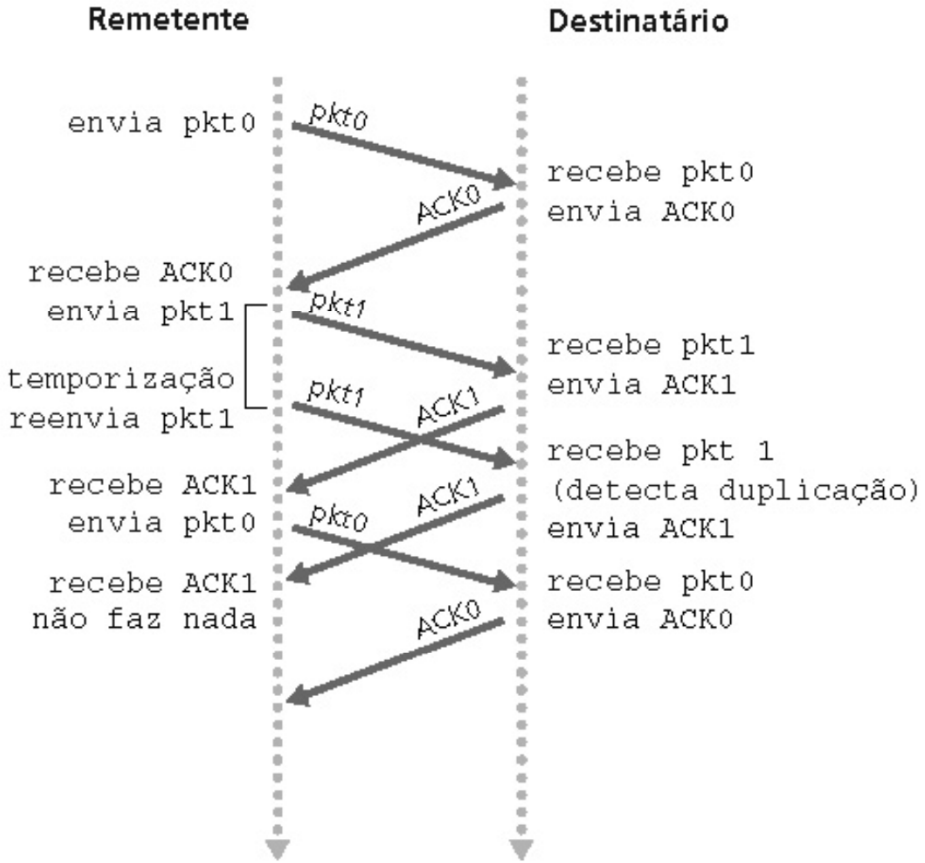


b. Pacote perdido

Canal com Erros e Perdas



c. ACK perdido



d. temporização prematura

Canal com Erros e Perdas

- Uso de...
  - Checksum
  - Número de sequência
  - Temporizadores
  - Pacotes de reconhecimento

Já é suficiente para a operação de um protocolo de transferência confiável de dados!

É possível melhorar o desempenho da transferência de dados?

Canal com Erros e Perdas

- Canal é confiável, mas **o desempenho é um problema**
  - Ex.: Enlace de 1Gb/s, retardo de 15ms e pacotes de 1kB

$$d_{trans} = \frac{L}{R} = \frac{8000\text{bits}}{10^9\,\text{bps}} = 8\,\text{microsegundos}$$

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

Pacotes de 1kB são enviados a cada 30ms

→ Vazão de 1kB/30ms=33kB/s num enlace de 1Gb/s

Canal com Erros e Perdas

- Canal é confiável, mas **o desempenho é um problema**
  - Ex.: Enlace de 1Gb/s, retardo de 15ms e pacotes de 1kB

$$d_{trans} = \frac{L}{R} = \frac{8000\text{bits}}{10^9\,\text{bps}} = 8\,\text{m}$$

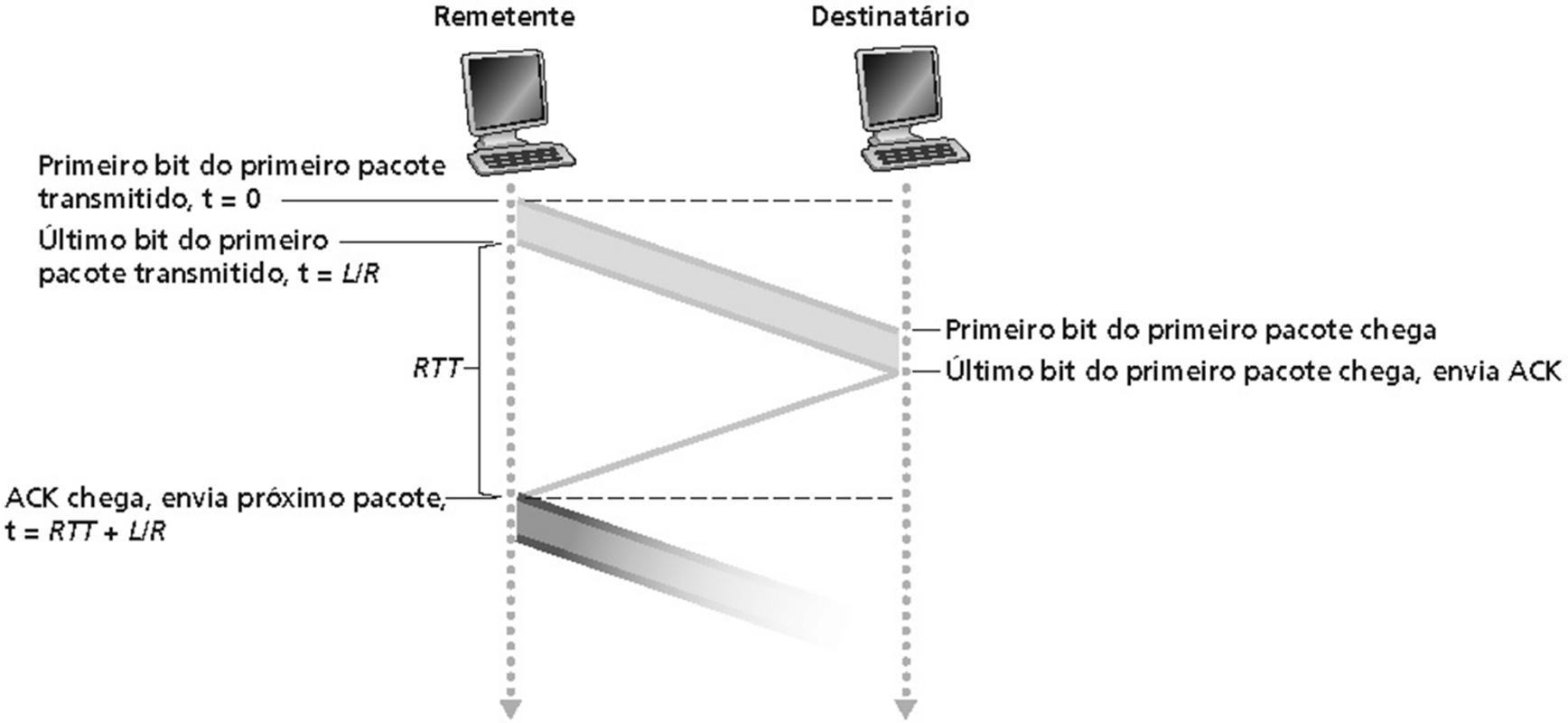
Utilização $= \frac{\quad}{30{,}008} = 0{,}00027$

Protocolo limita o uso dos recursos físicos

1kB são enviados a cada 30ms

→ Vaz de 1kB/30ms=33kB/s num enlace de 1Gb/s

Operação Pare-e-Espere

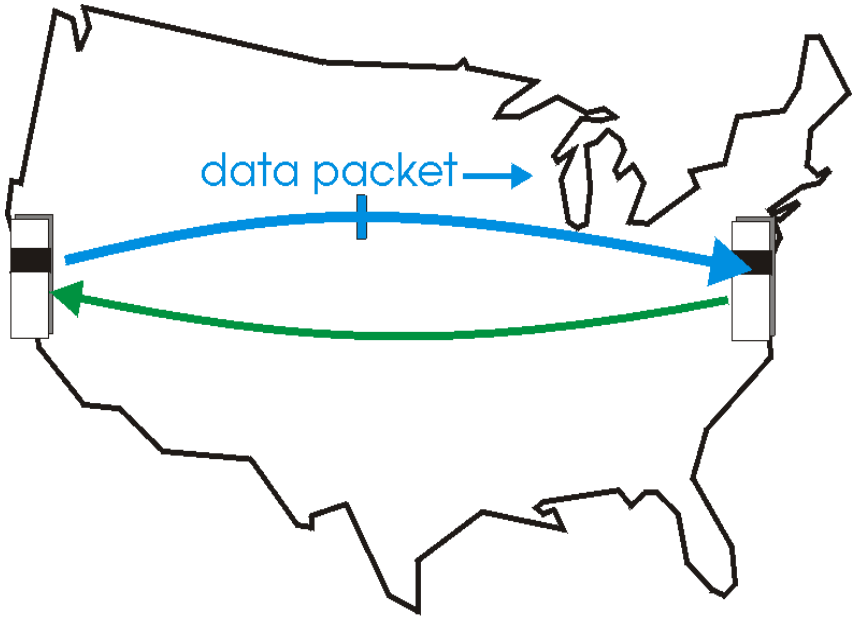


a. Operação pare e espere

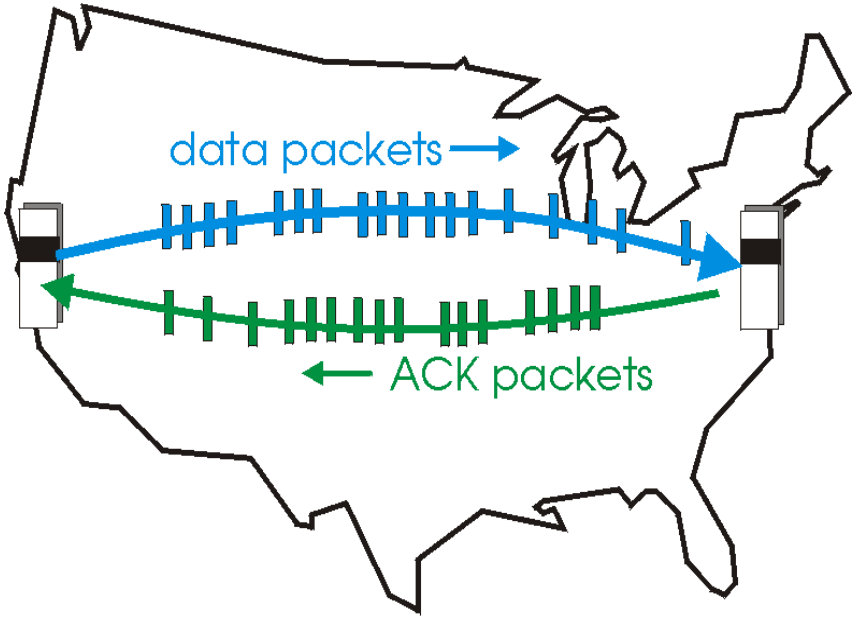
$$\text{Utilização}_{\text{canal}} = \frac{L / R}{RTT + L / R} = \frac{0{,}008}{30{,}008} = 0{,}00027$$

Paralelismo (pipelining)

- Transmissor envia vários pacotes em sequência
  - **Todos** esperando para serem **reconhecidos**
- Faixa de números de sequência deve ser aumentada
- Armazenamento no Transmissor e/ou no receptor

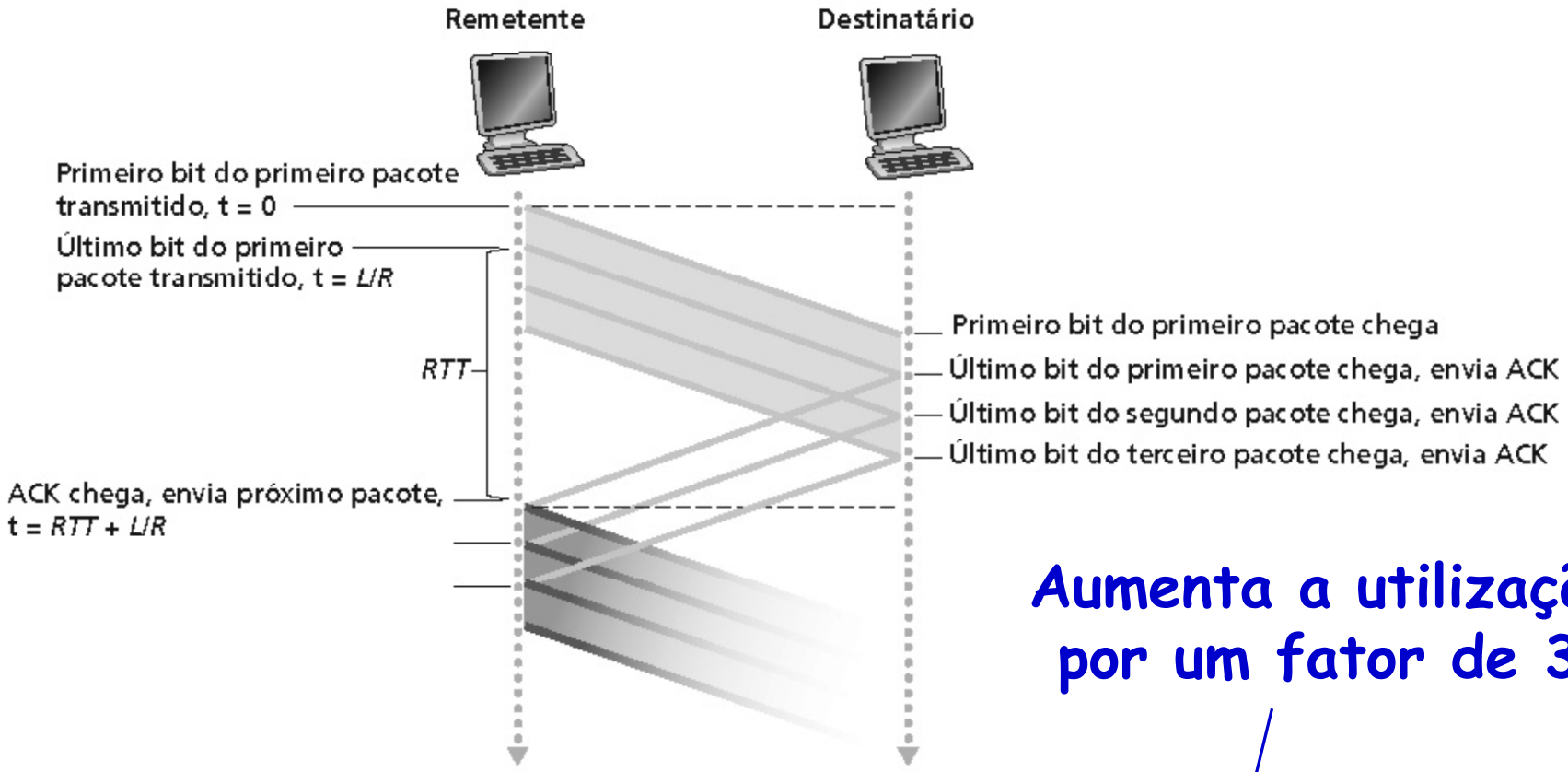


(a) operação do protocolo pare e espere



(a) operação do protocolo com paralelismo

Paralelismo (pipelining)



b. Operação com paralelismo

Aumenta a utilização por um fator de 3!

$$\text{Utilização}_{\text{canal}} \equiv \frac{3xL/R}{RTT + L/R} \equiv \frac{0{,}024}{30{,}008} \equiv 0{,}00081$$

Paralelismo (pipelining)

- Recuperação de segmentos perdidos
  - Duas formas genéricas
    - *Go-back-N*
    - Retransmissão seletiva

Protocolos com Paralelismo

Go-back-N:

- O transmissor pode ter até N pacotes não reconhecidos "em trânsito"
- Receptor envia apenas ACKs cumulativos
  - Não reconhece pacote se houver falha de sequência
- Transmissor possui um temporizador para o pacote mais antigo ainda não reconhecido
  - Se o temporizador estourar, retransmite todos os pacotes ainda não reconhecidos

Protocolos com Paralelismo

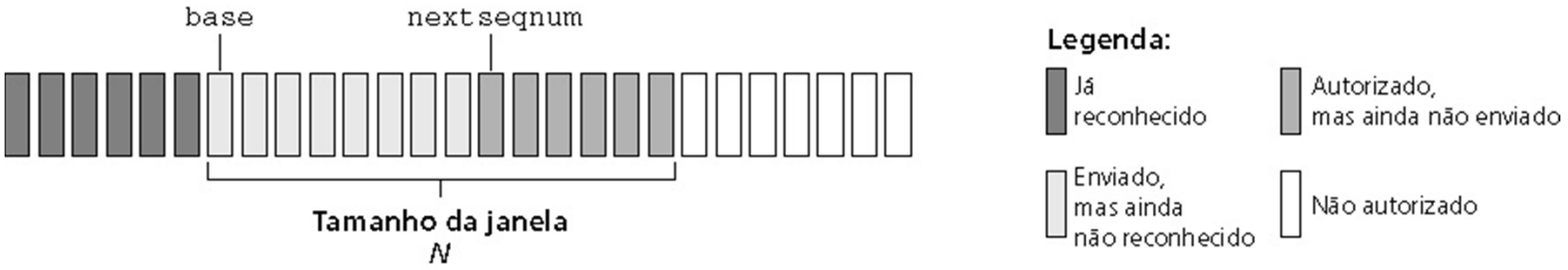
Retransmissão seletiva:

- O transmissor pode ter até N pacotes não reconhecidos "em trânsito"
- Receptor reconhece pacotes individuais
- Transmissor possui um temporizador para cada pacote ainda não reconhecido
  - Se o temporizador estourar, retransmite apenas o pacote correspondente

Go-back-N (GBN)

Transmissor:

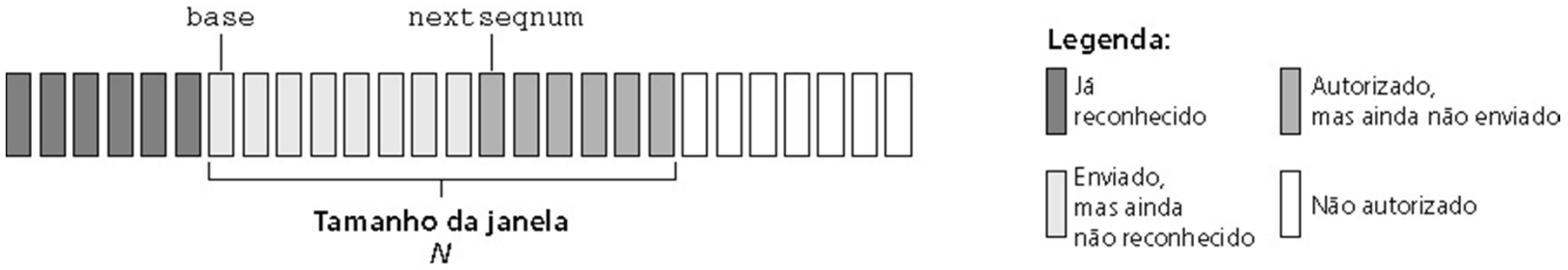
- Número de sequência de k-bits no cabeçalho do pacote
- Admite "janela" de até N pacotes consecutivos não reconhecidos



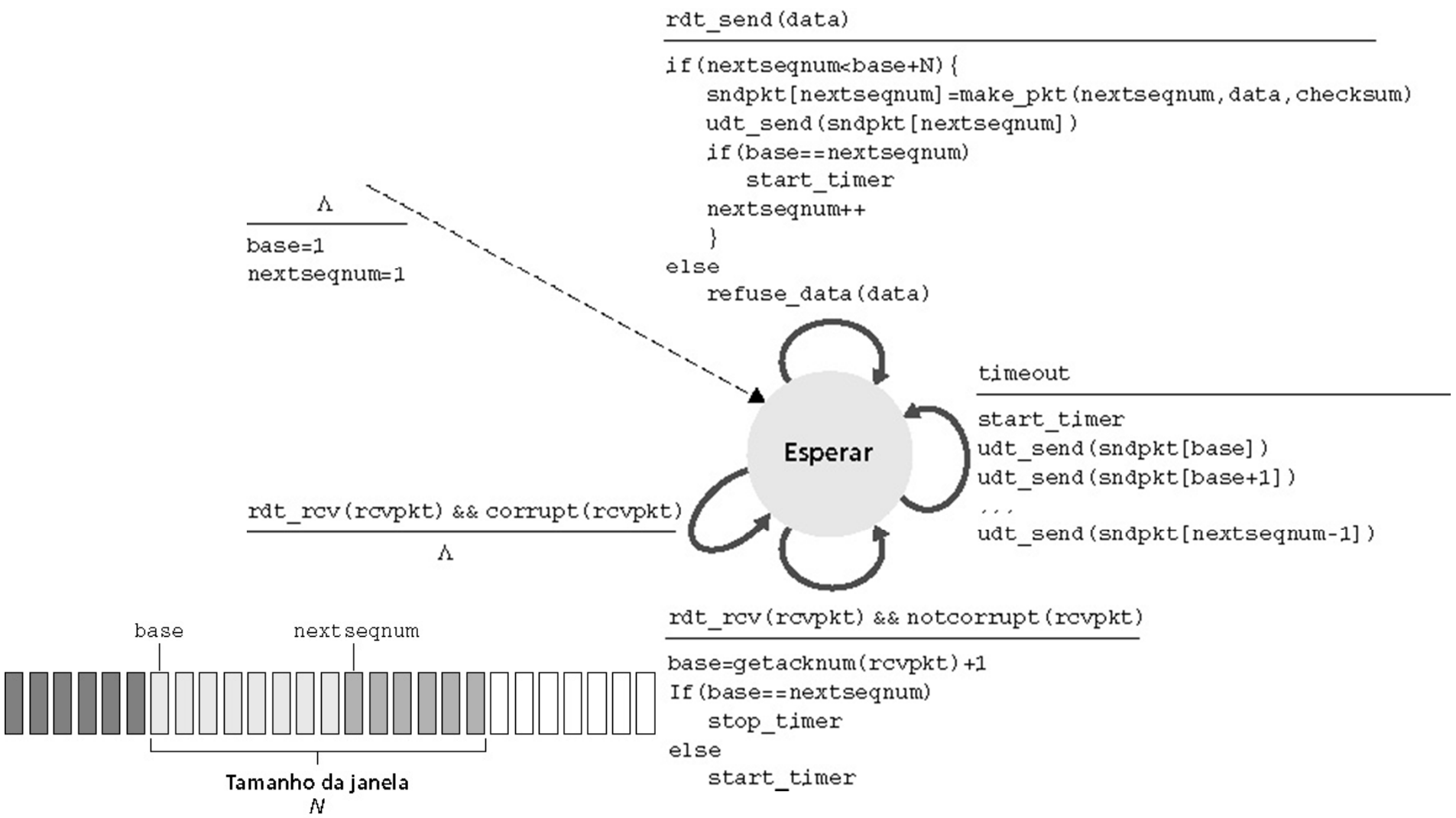
Go-back-N (GBN)

Transmissor:

- ACK(n): reconhece todos pacotes, até e inclusive número de sequência n - "ACK cumulativo"
  - Pode receber ACKs duplicados
- Temporizador para o pacote mais antigo ainda não reconhecido
- *timeout(b):* retransmite o pacote b e todos os outros com número de sequência maiores dentro da janela

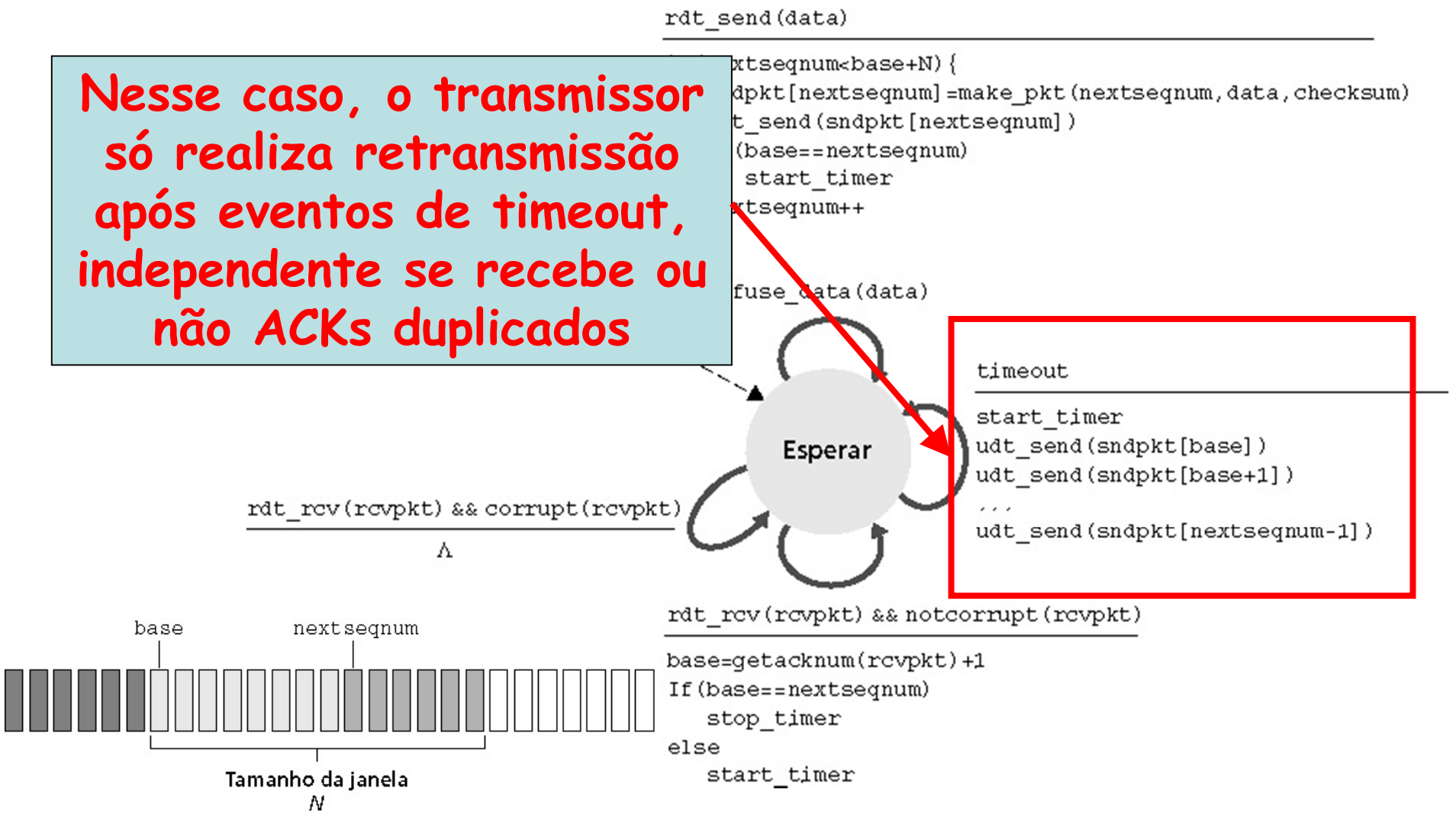


GBN: FSM estendida para o transmissor

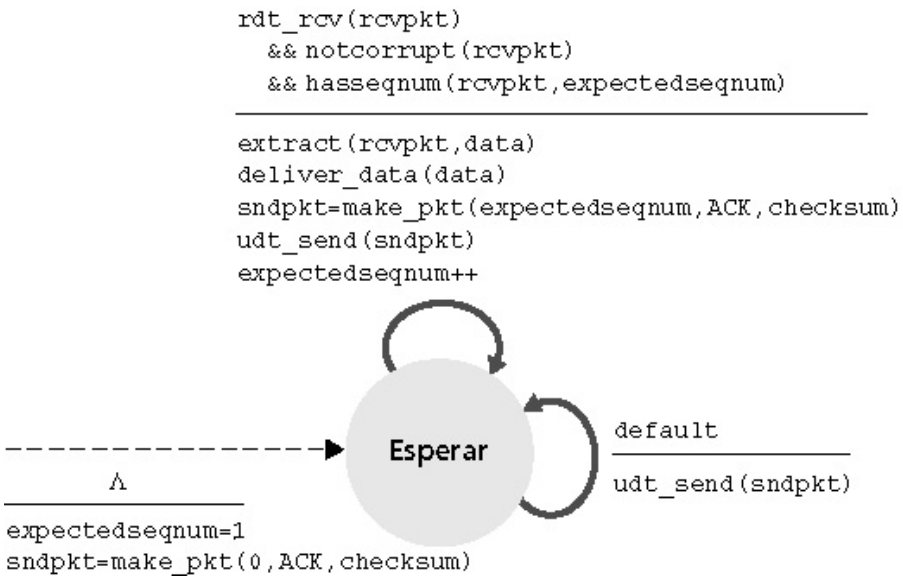


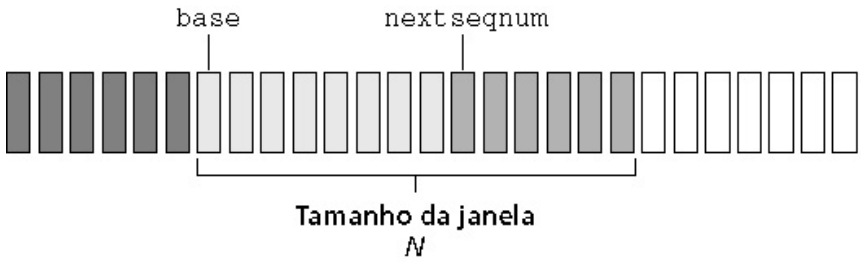
GBN: FSM estendida para o transmissor

Nesse caso, o transmissor só realiza retransmissão após eventos de timeout, independente se recebe ou não ACKs duplicados



GBN: FSM estendida para o receptor

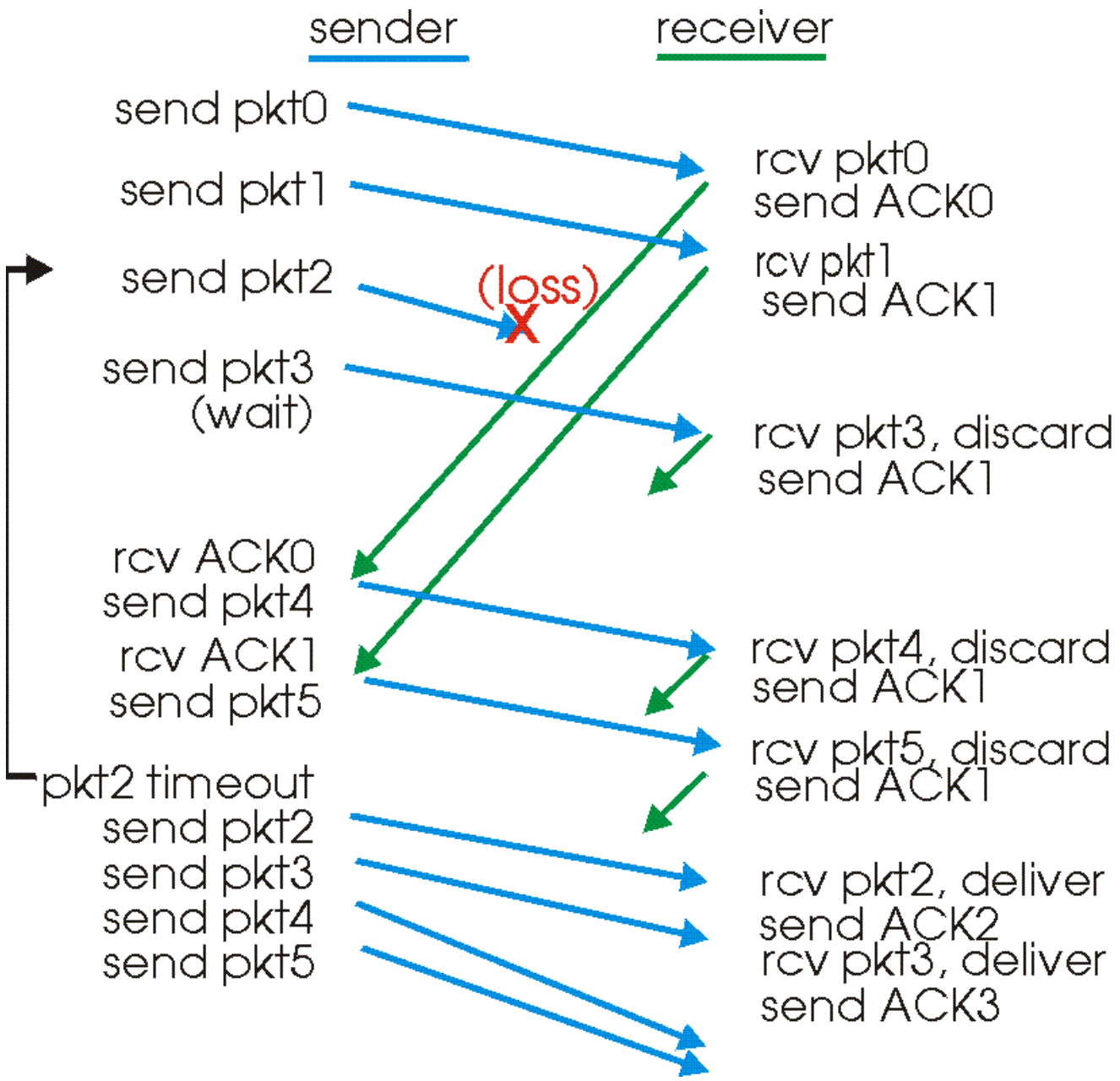




Receptor simples

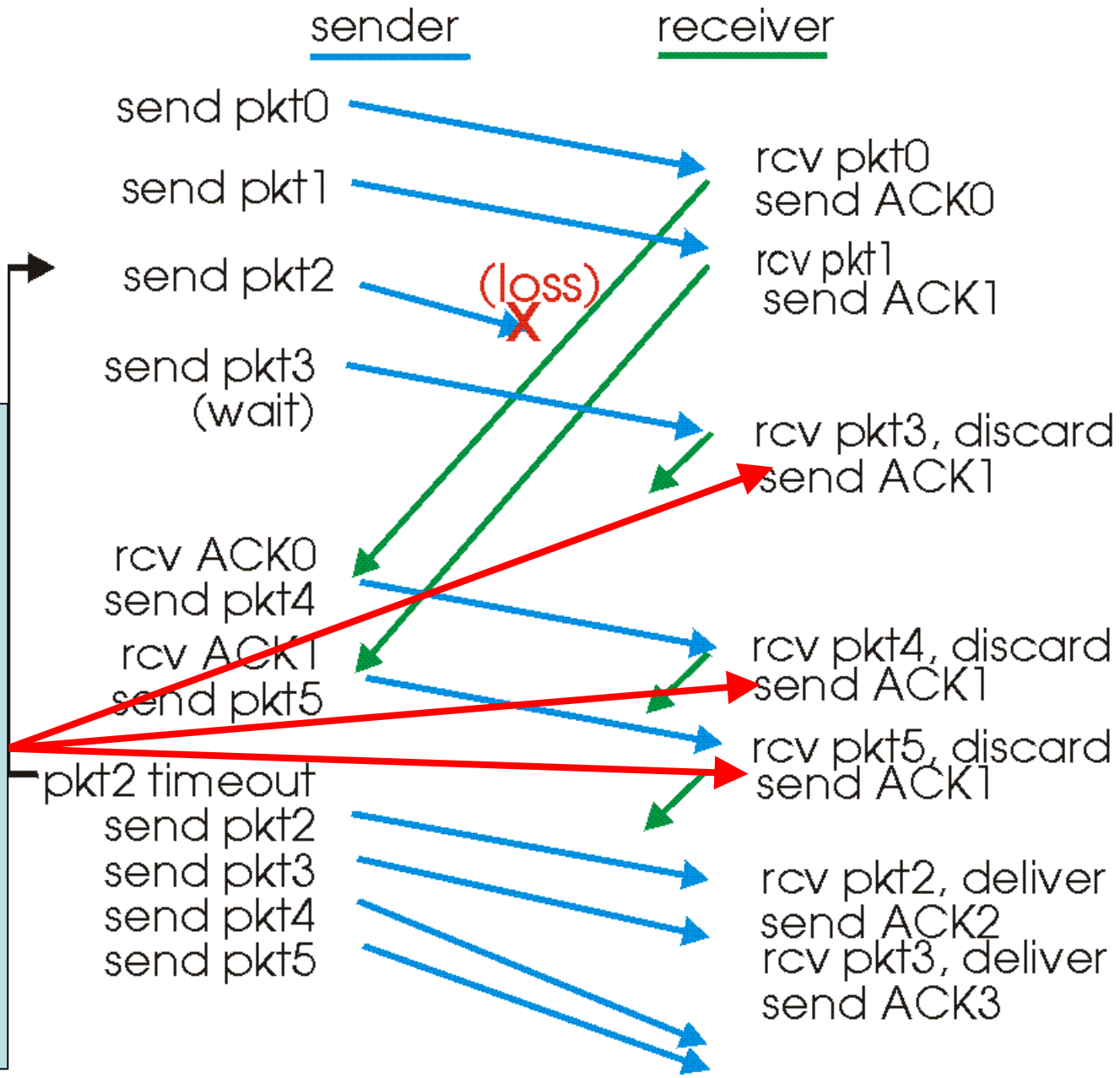
- Usa apenas *ACK*
  - Sempre envia *ACK* para pacote recebido corretamente com o maior número de sequência **em ordem**
  - Pode gerar *ACKs* duplicados
    - Evento "default"
- Pacotes fora de ordem
  - Descarta (não armazena)
    - **Receptor não usa buffers**
    - Evento "default"

GBN em Ação


sender
receiver
send pkt0
send pkt1
send pkt2
(loss)
send pkt3
(wait)
rcv pkt0
send ACK0
rcv pkt1
send ACK1
rcv pkt3, discard
send ACK1
rcv ACK0
send pkt4
rcv ACK1
send pkt5
rcv pkt4, discard
send ACK1
rcv pkt5, discard
send ACK1
pkt2 timeout
send pkt2
send pkt3
send pkt4
send pkt5
rcv pkt2, deliver
send ACK2
rcv pkt3, deliver
send ACK3

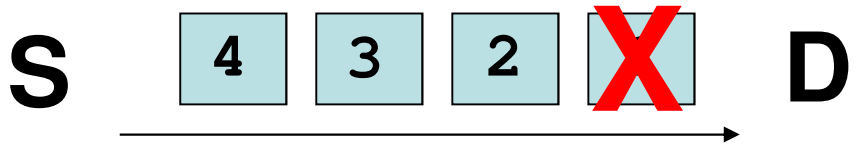
GBN em Ação

Como o pkt2 foi perdido, todos os pacotes em sequência são considerados fora de ordem e, portanto, são descartados

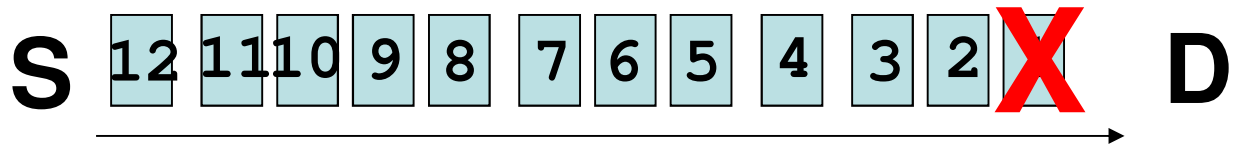


Retransmissão Seletiva

- Problema de desempenho do GBN
  - Se o tamanho da janela N for grande e o produto do atraso com a largura de banda também for grande
    - Muitos pacotes podem ser retransmitidos



Menor produto atraso x largura de banda:
4 retransmissões

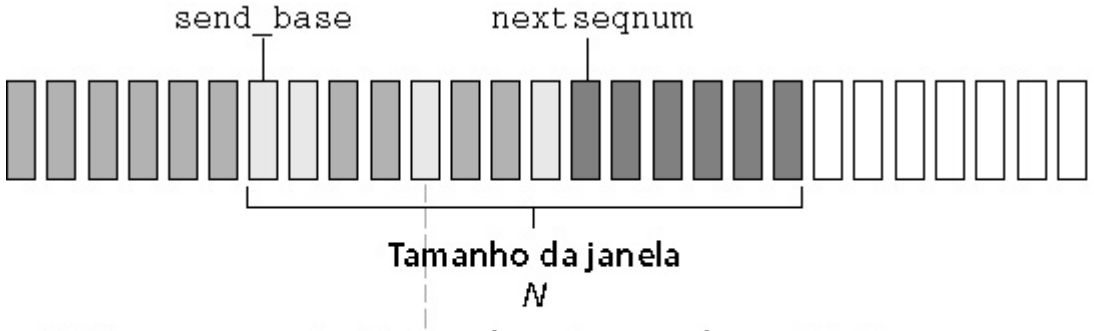


Maior produto atraso x largura de banda:
12 retransmissões

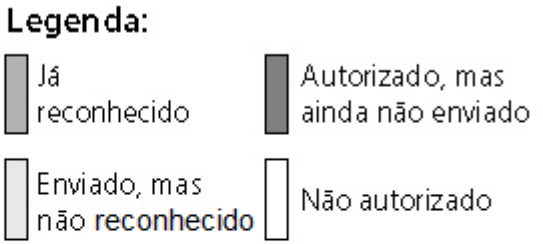
Retransmissão Seletiva

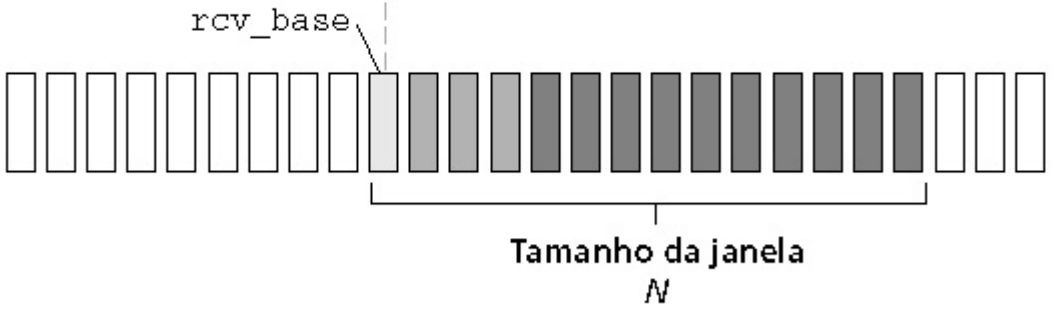
- Receptor reconhece individualmente todos os pacotes recebidos corretamente
  - Armazena pacotes no buffer, conforme necessário, para posterior entrega ordenada à camada superior
- Transmissor apenas reenvia pacotes para os quais um ACK não foi recebido
  - Temporizador no remetente para cada pacote sem ACK
- Janela de transmissão
  - N números de sequência consecutivos
  - Outra vez limita números de sequência de pacotes enviados, mas ainda não reconhecidos

Retransmissão Seletiva

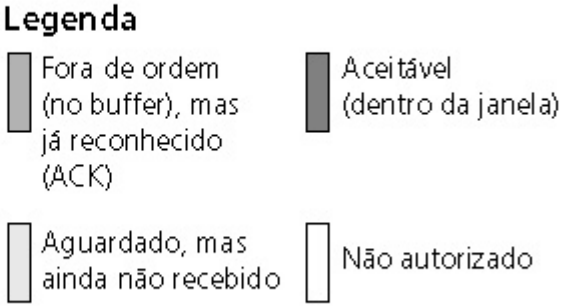


a. Visão que o remetente tem dos números de seqüência





b. Visão que o destinatário tem dos números de seqüência

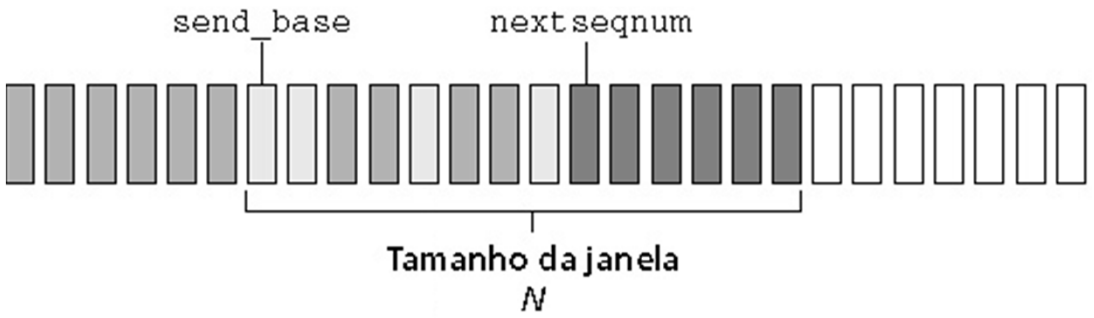


Retransmissão Seletiva

Retransmissão Seletiva

Transmissor:

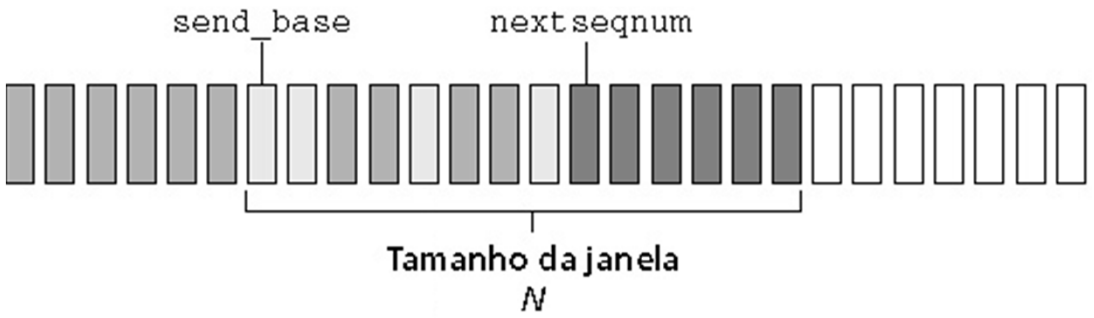
- Se próximo número de sequência n está disponível na janela
  - Envia o pacote e inicializa o temporizador(n)
- Estouro do temporizador(n):
  - Reenvia o pacote n e reinicia o temporizador(n)



Retransmissão Seletiva

Transmissor:

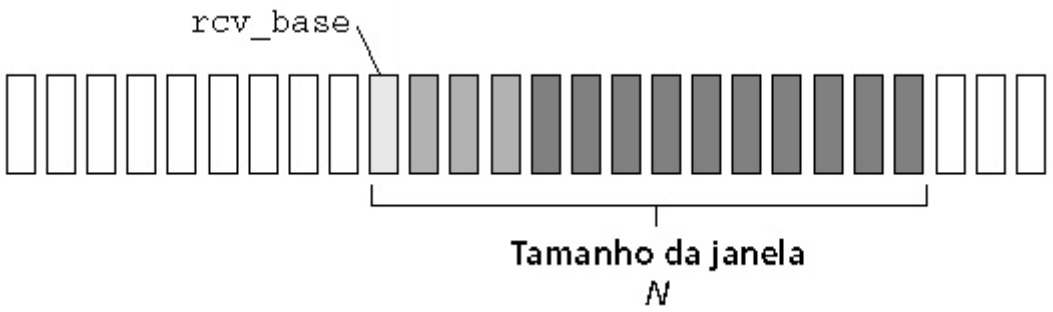
- ACK(n) na janela ([send_base,nextseqnum-1])
  - Marca pacote n como "recebido"
- Se n for o menor pacote não reconhecido
  - Janela avança ao próximo número de sequência não reconhecido



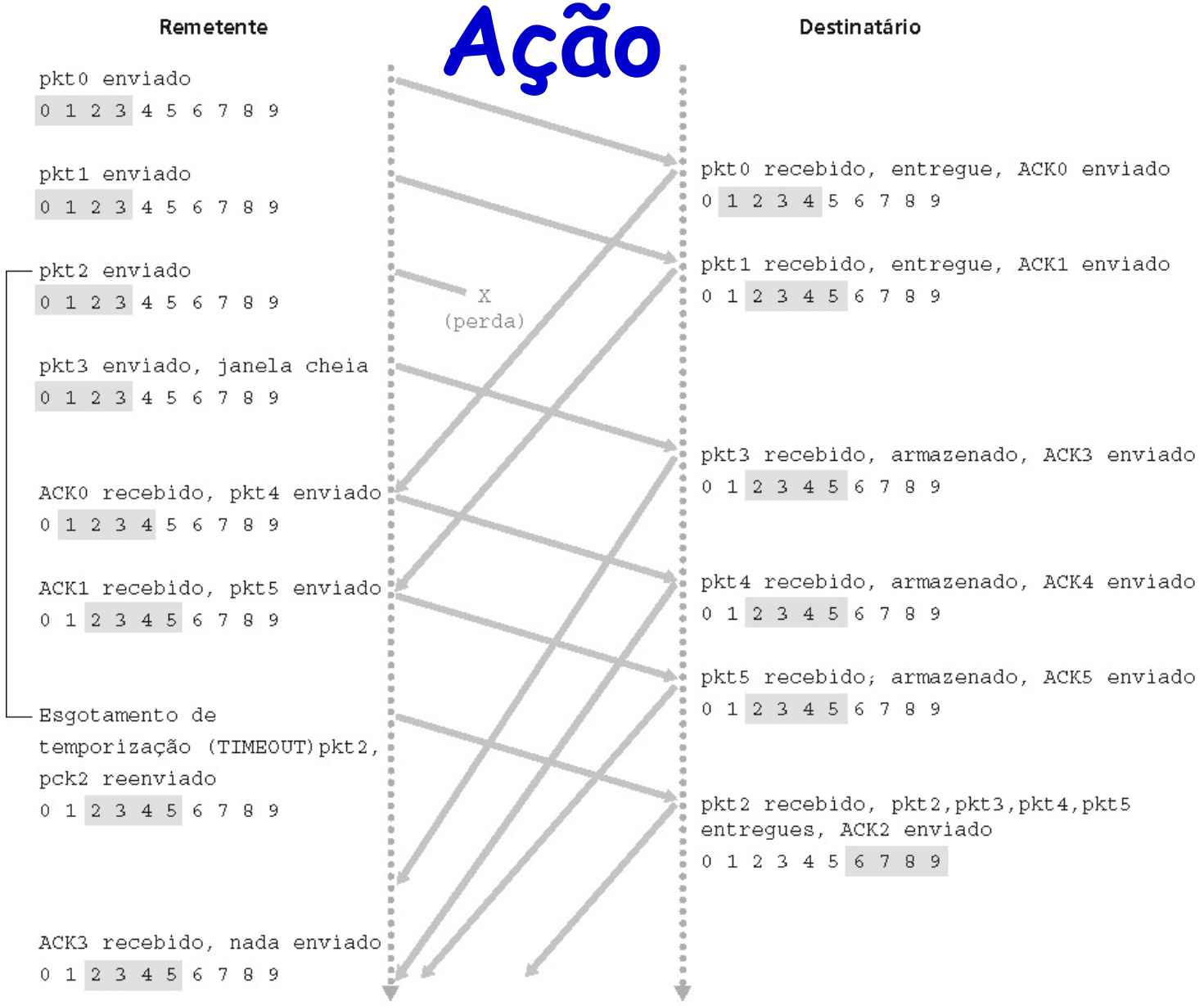
Retransmissão Seletiva

Receptor:

- Pacote n em [rcv_base, rcv_base+N-1]
  - Envia ACK(n)
    - Fora de ordem
      - Armazena
    - Em ordem
      - Entrega (tb. entrega pacotes armazenados em ordem), Avança janela p/ próxima pacote ainda não recebido
- Pacote n em [rcv_base-N,rcv_base-1]
  - Envia ACK(n) duplicado
- Senão
  - Ignora

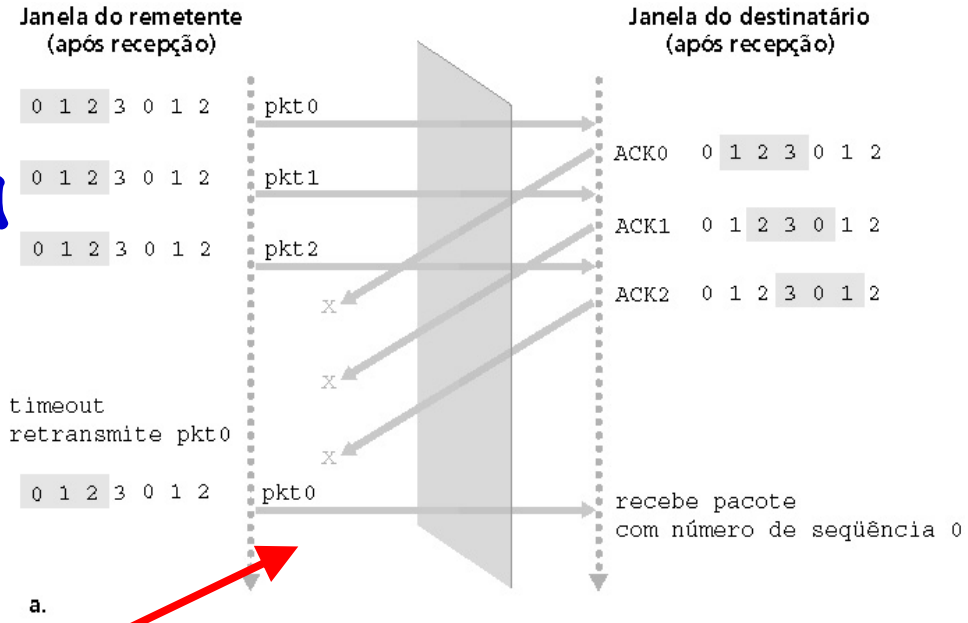


Retransmissão Seletiva em Ação

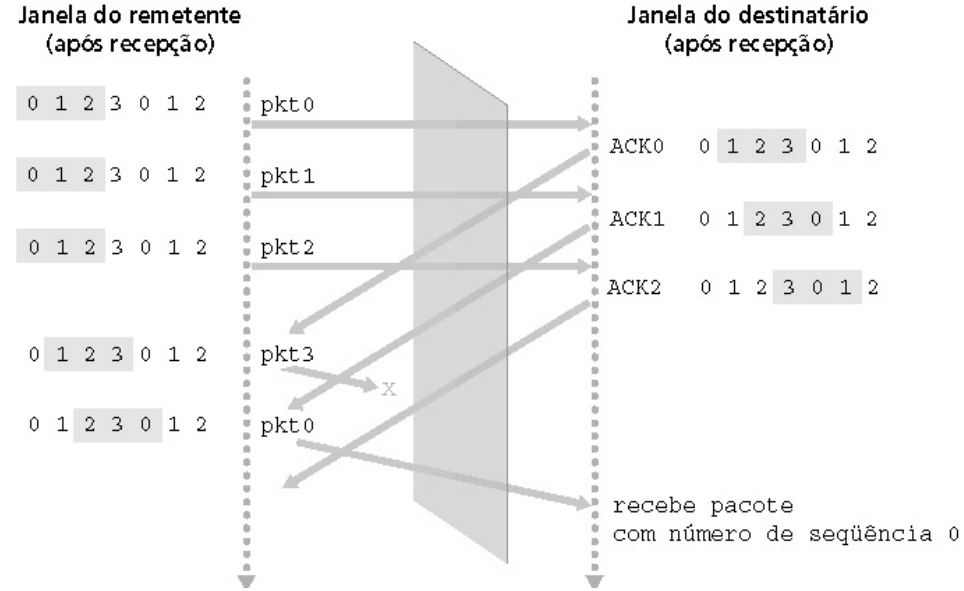


Retransmissão Seletiva: Dilema

- Considere:
  - Número de Sequência
    - 0, 1, 2, 3
  - Tamanho da Janela
    - 3

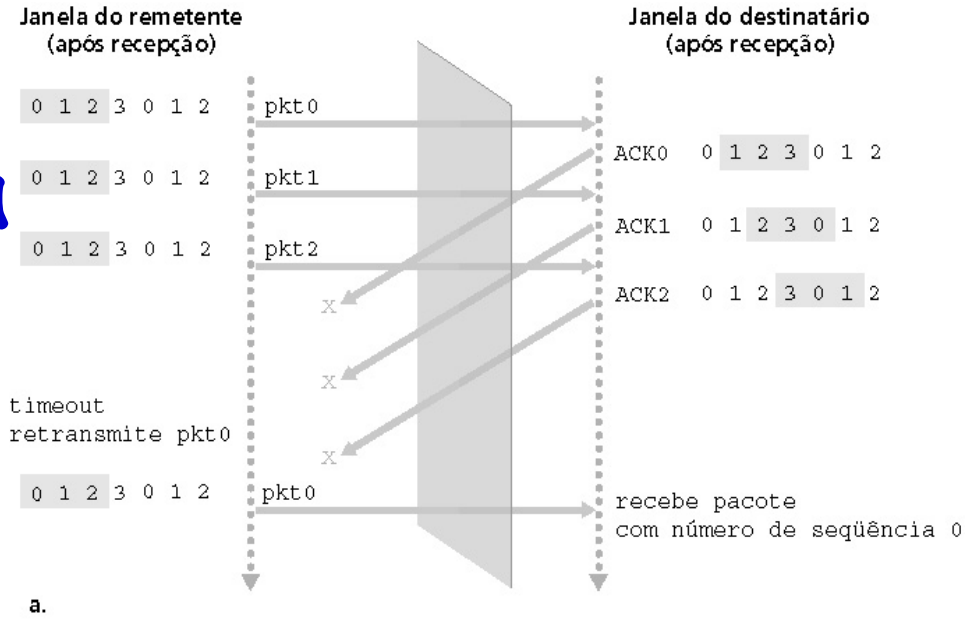


Como o receptor pode notar a diferença entre um novo pacote 0 ou uma retransmissão do primeiro pacote 0?

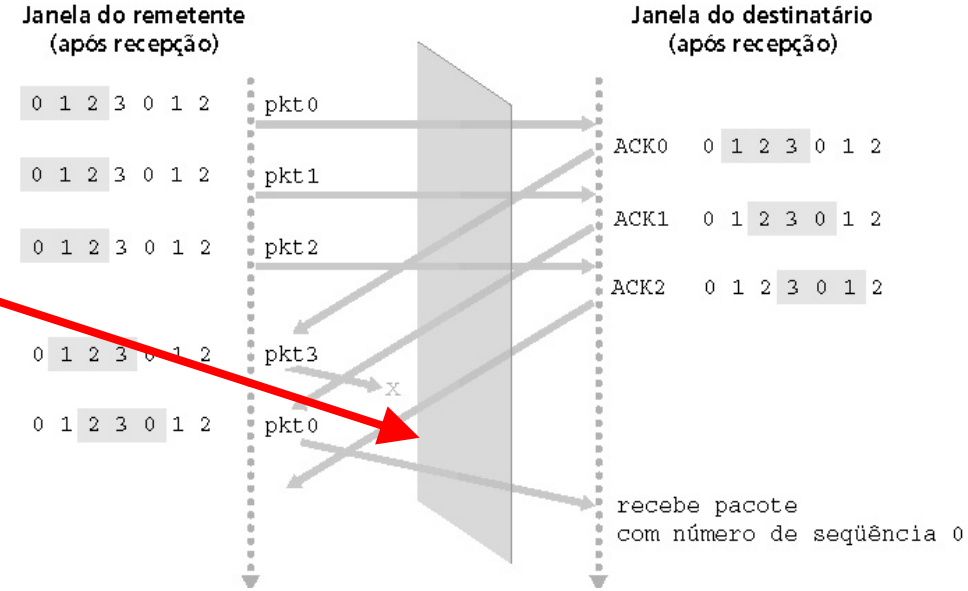


Retransmissão Seletiva: Dilema

- Considere:
  - Número de Sequência
    - 0, 1, 2, 3
  - Tamanho da Janela
    - 3

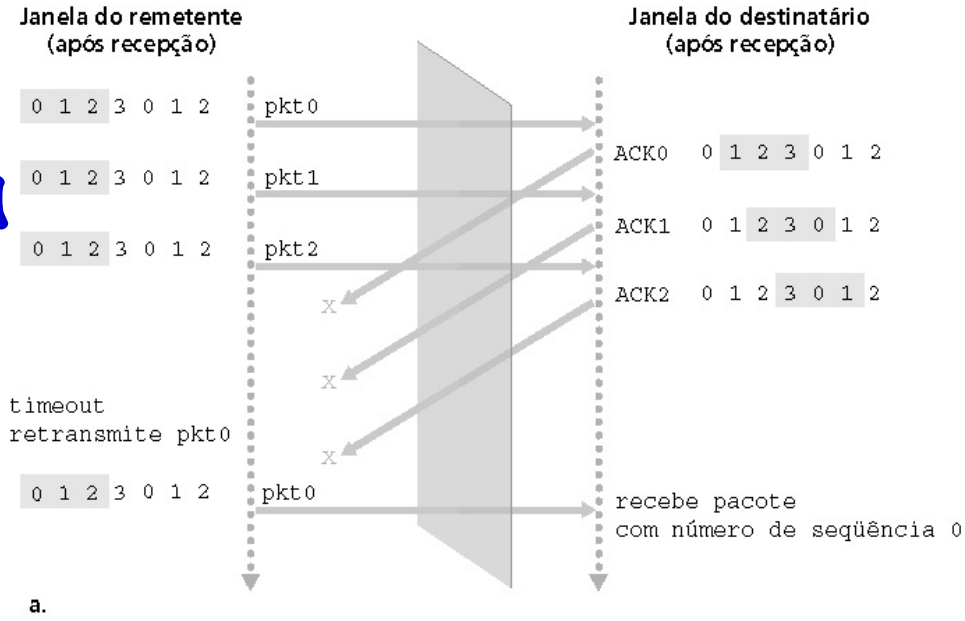


E agora? Será que os três primeiros ACKs foram perdidos?



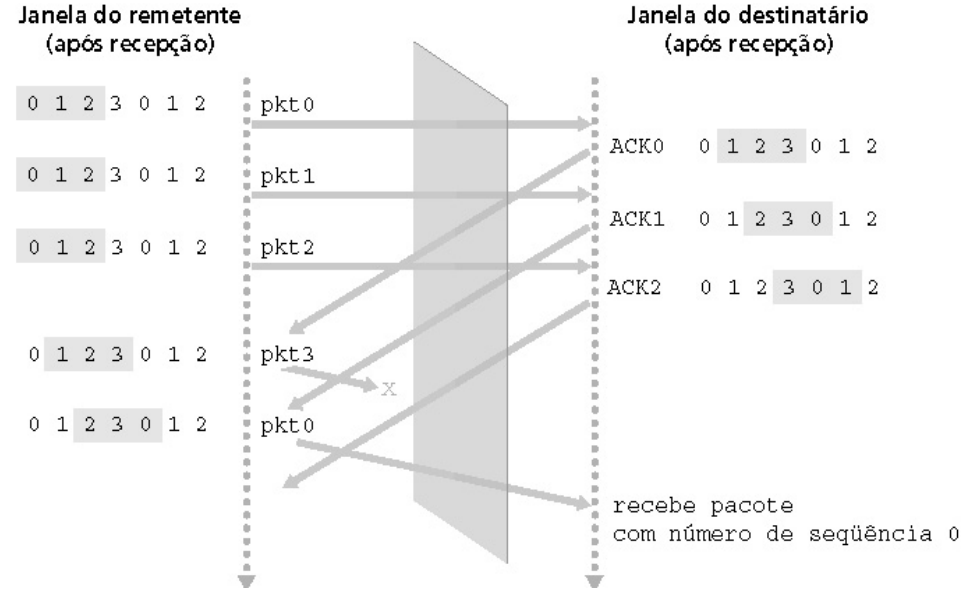
Retransmissão Seletiva: Dilema

- Considere:
  - Número de Sequência
    - 0, 1, 2, 3
  - Tamanho da Janela
    - 3



Qual a relação entre o número de elementos no intervalo dos números de sequência (I) e o tamanho da janela (N)?

Um bom valor é N ≤ I/2



Transmission Control Protocol (TCP)

TCP

- Muito mais complexo que o UDP
  - UDP: RFC 768
  - TCP: RFCs 793, 1122, 1323, 2018 e 2581

- Orientado à conexão
  - Antes do início da transmissão há um *three-way handshake* (apresentação em três vias) entre as estações finais
    - Dois processos trocam segmentos para definir parâmetros

TCP

- Muito mais complexo que o UDP
  - UDP: RFC 768
  - TCP: RFCs 793, 1122, 1323, 2018 e 2581

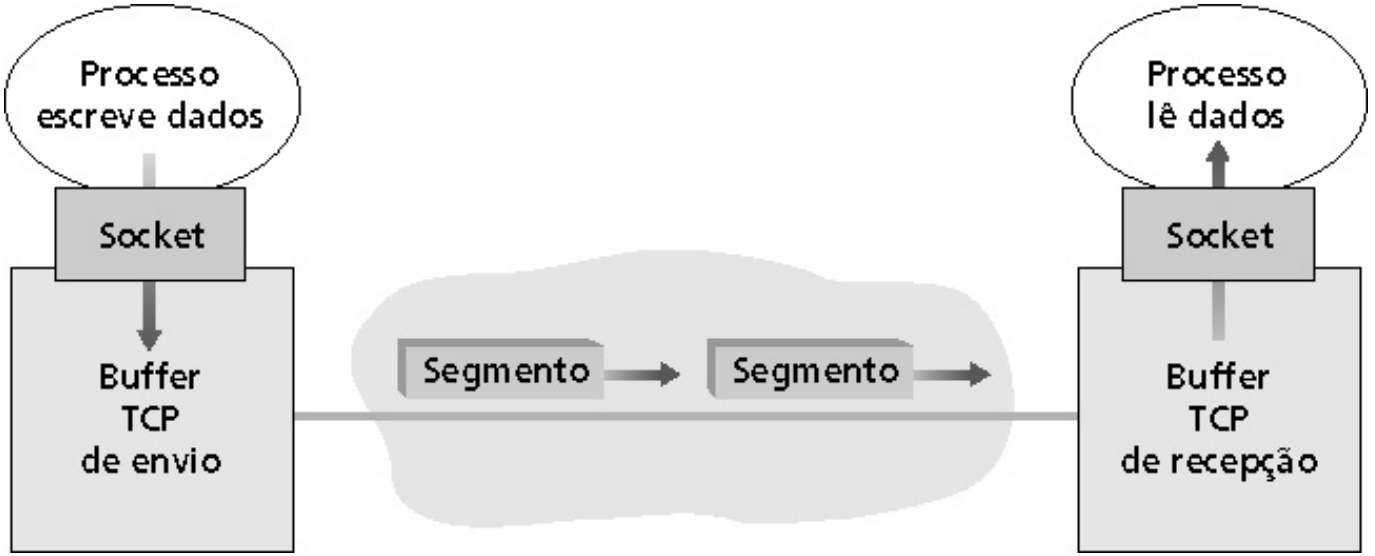
- Orientado à conexão
  - É uma conexão lógica
    - Diferente da comutação de **circuitos**
      - Não há um caminho definido e nem reserva de recursos nos elementos intermediários
    - Reserva de recursos "só existe" nos **sistemas finais**
      - Variáveis de estado são mantidas nesses sistemas

TCP

- É ponto-a-ponto
  - Um transmissor e um receptor
- Transmissão *full duplex*
  - Fluxo de dados bidirecional na mesma conexão
- Quantidade máxima de bits por segmento
  - Definição do MSS (tamanho máximo de segmento)
- Controle de fluxo
  - Receptor não será afogado pelo transmissor
- Controle de congestionamento
  - Evita a saturação dos enlaces da rede

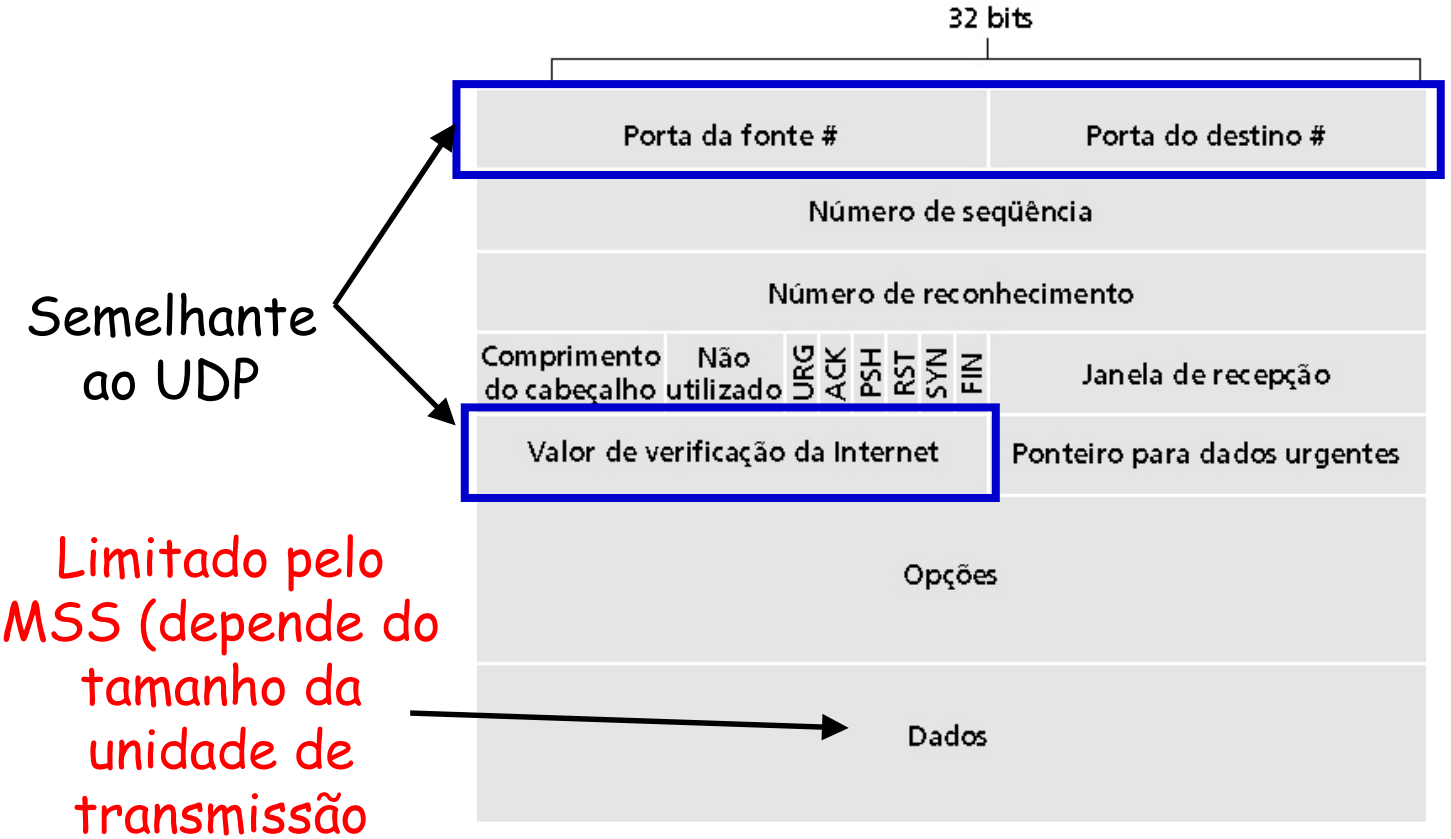
TCP

- *Buffers*
  - Transmissão e recepção
  - Tamanho definido durante a conexão



Segmento TCP

- Cabeçalho: 20 bytes (se opções não forem usadas)



Segmento TCP

- Cabeçalho: 20 bytes (se opções não forem usadas)

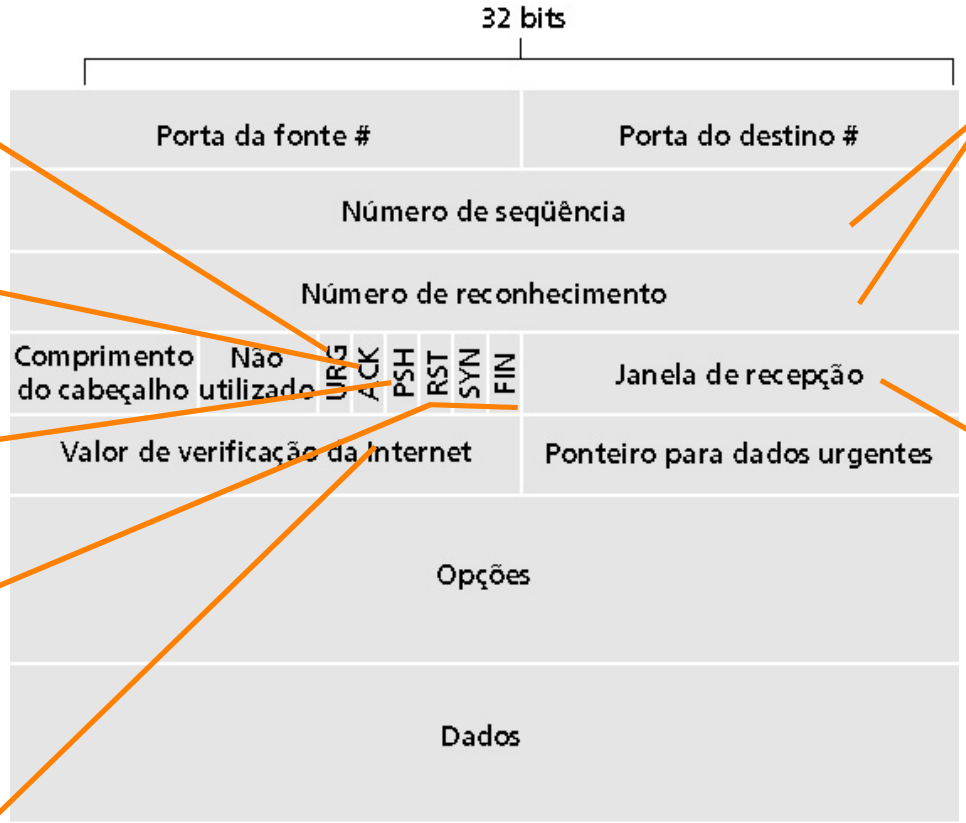
URG: dados urgentes indicados pelo ponteiro (pouco usado)

ACK: campo de ACK é válido

PSH: envio imediato para a aplicação

RST, SYN, FIN: estabelec. de conexão (comandos de criação e término)

Internet checksum (como no UDP)



contagem por bytes de dados (não segmentos!)

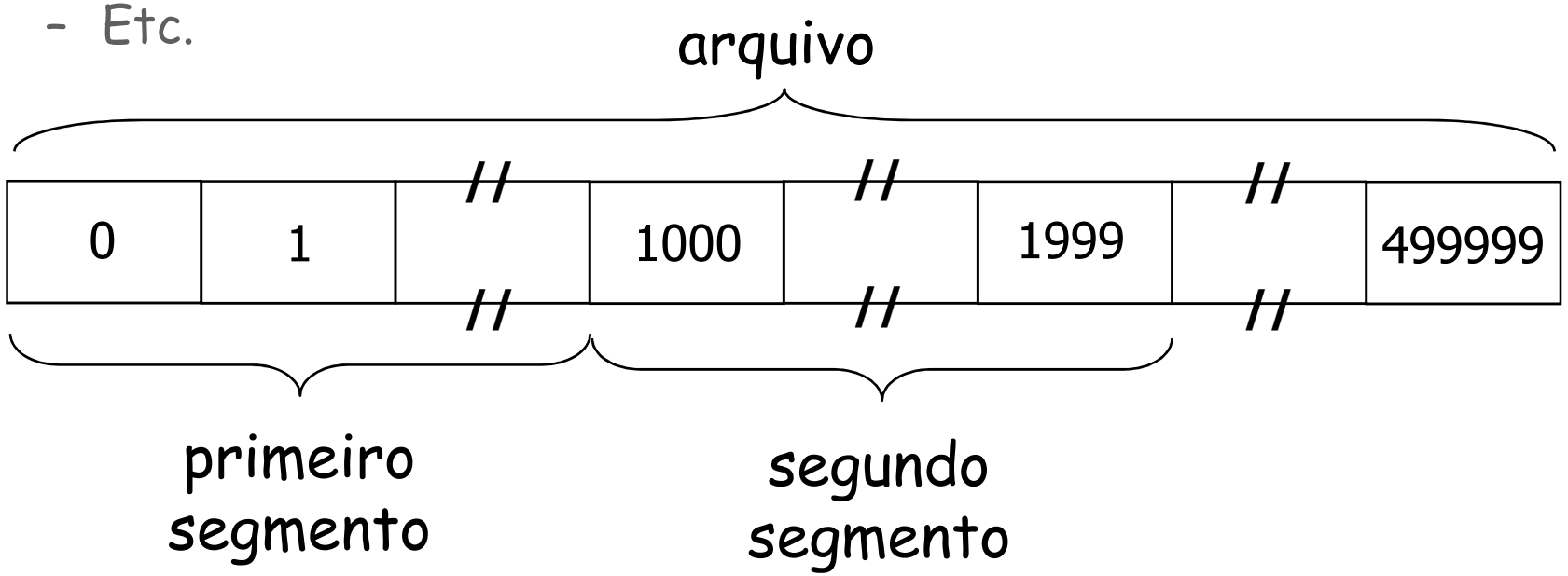
número de bytes que o receptor está pronto para aceitar

Número de Sequência e ACKs

- Fundamentais para a transferência confiável
- Para o TCP, dados são um **fluxo de bytes ordenados**
  - Organizados a partir do **número de sequência**
    - Baseado no número de bytes e não no de segmentos
    - Igual ao "número" do primeiro byte de dados do segmento, estabelecido conforme a sua posição no fluxo de bytes

Número de Sequência e ACKs

- Ex.: fluxo de dados com 500 kB e MSS 1 kB
  - 500 segmentos de 1000 bytes
  - Primeiro segmento: # seq → 0
  - Segundo segmento: # seq → 1000
  - Terceiro segmento: # seq → 2000
  - Etc.

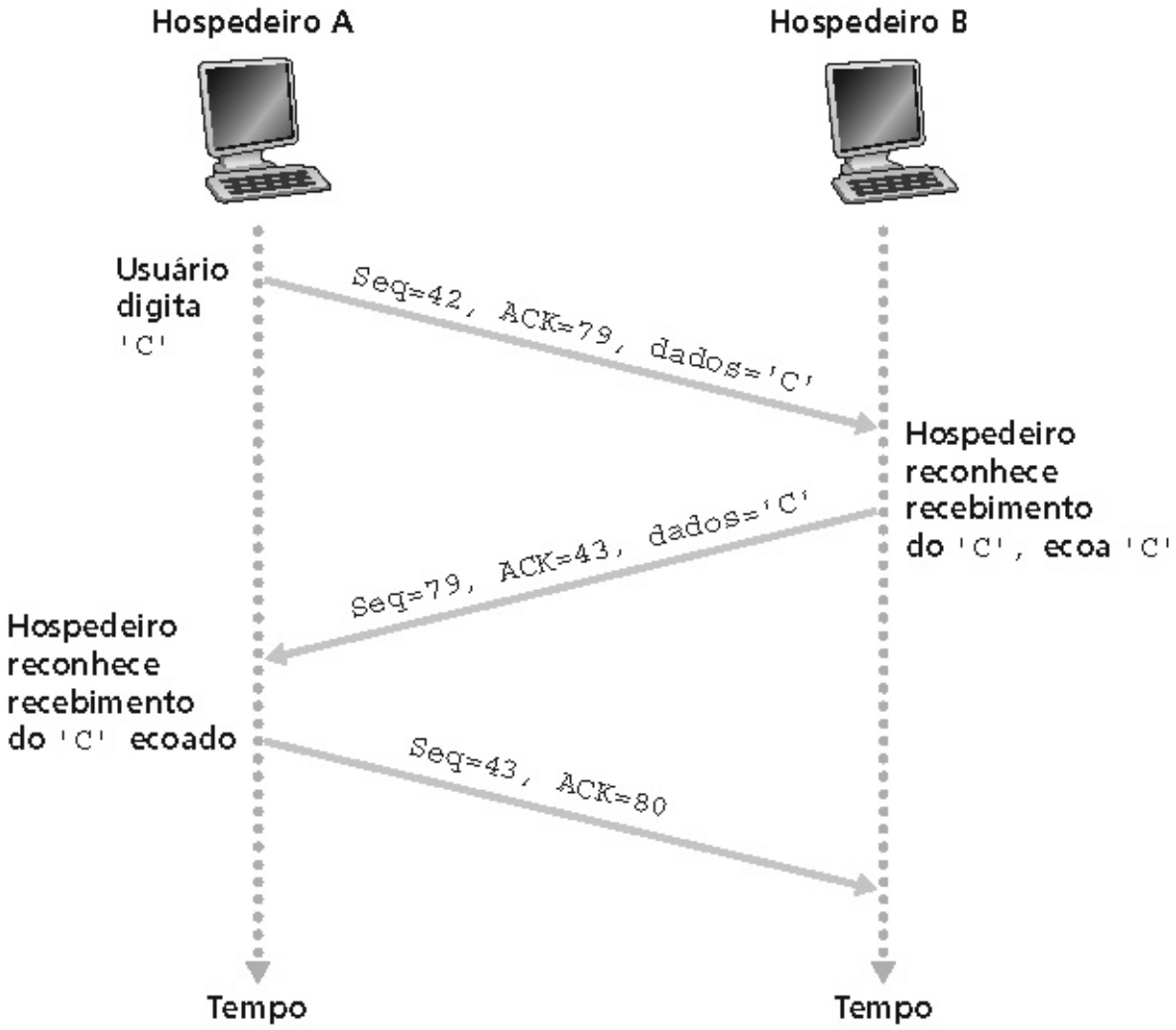


Número de Sequência e ACKs

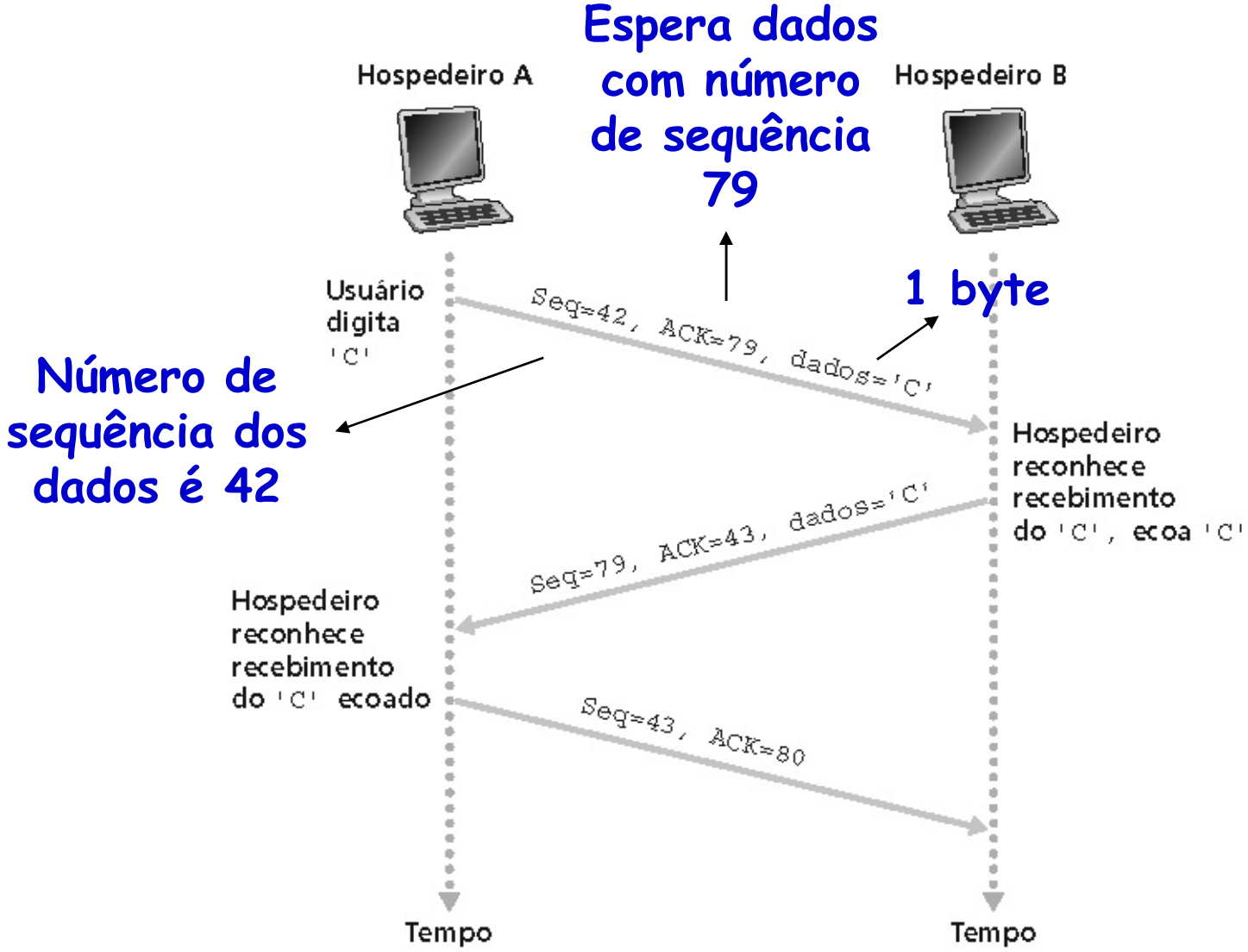
- Número de reconhecimento
  - Número de sequência do próximo byte esperado do "outro lado"
  - ACK cumulativo
    - TCP só reconhece os bytes até o primeiro que estiver faltando, mesmo se outros segmentos fora de ordem já tiverem sido recebidos

- Como o receptor trata os segmentos fora da ordem?
  - Nada é especificado pela RFC
  - É definido por quem implementa o protocolo
    - Ex.: GBN, repetição seletiva (SR) ou uma opção diferente

Número de Sequência e ACKs

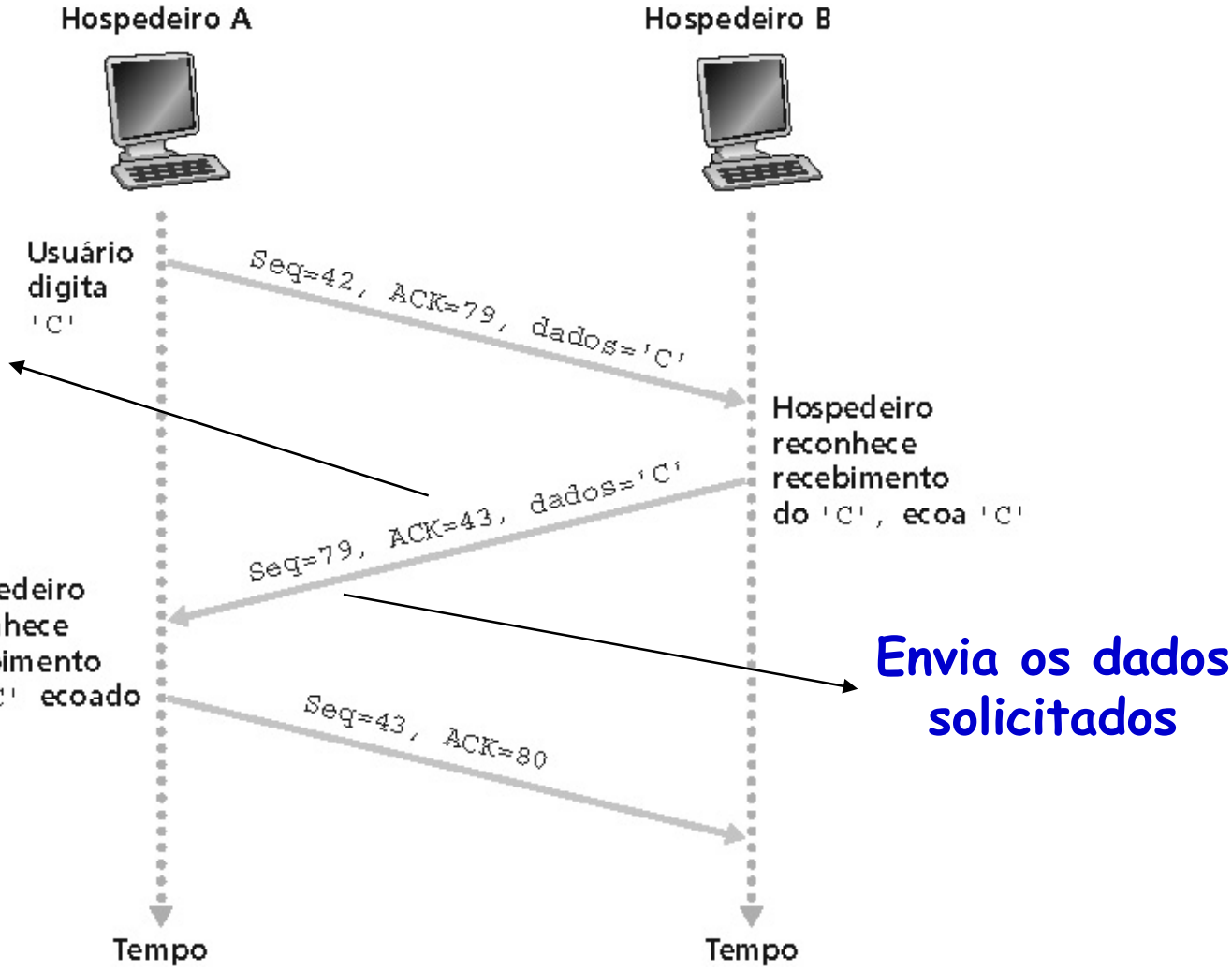


Número de Sequência e ACKs



Número de Sequência e ACKs

Reconhece o recebimento do anterior e indica que está esperando os próximos bytes com número de sequência 43



Envia os dados solicitados

Temporização

- Como escolher valor do temporizador TCP?
  - Deve ser **maior que o RTT**

RTT é variável!

- Muito curto
  - Estouro prematuro do temporizador
    - Retransmissões desnecessárias
- Muito longo
  - Reação demorada à perda de segmentos

Temporização

- Como estimar o RTT?
  - Medir o tempo entre a transmissão de um segmento e o recebimento do ACK correspondente
    - Ignorar retransmissões
- RTT de cada amostra pode ter grande variação
  - Solução: usar várias amostras recentes (`SampleRTT`) e calcular uma média ponderada (`EstimatedRTT`)

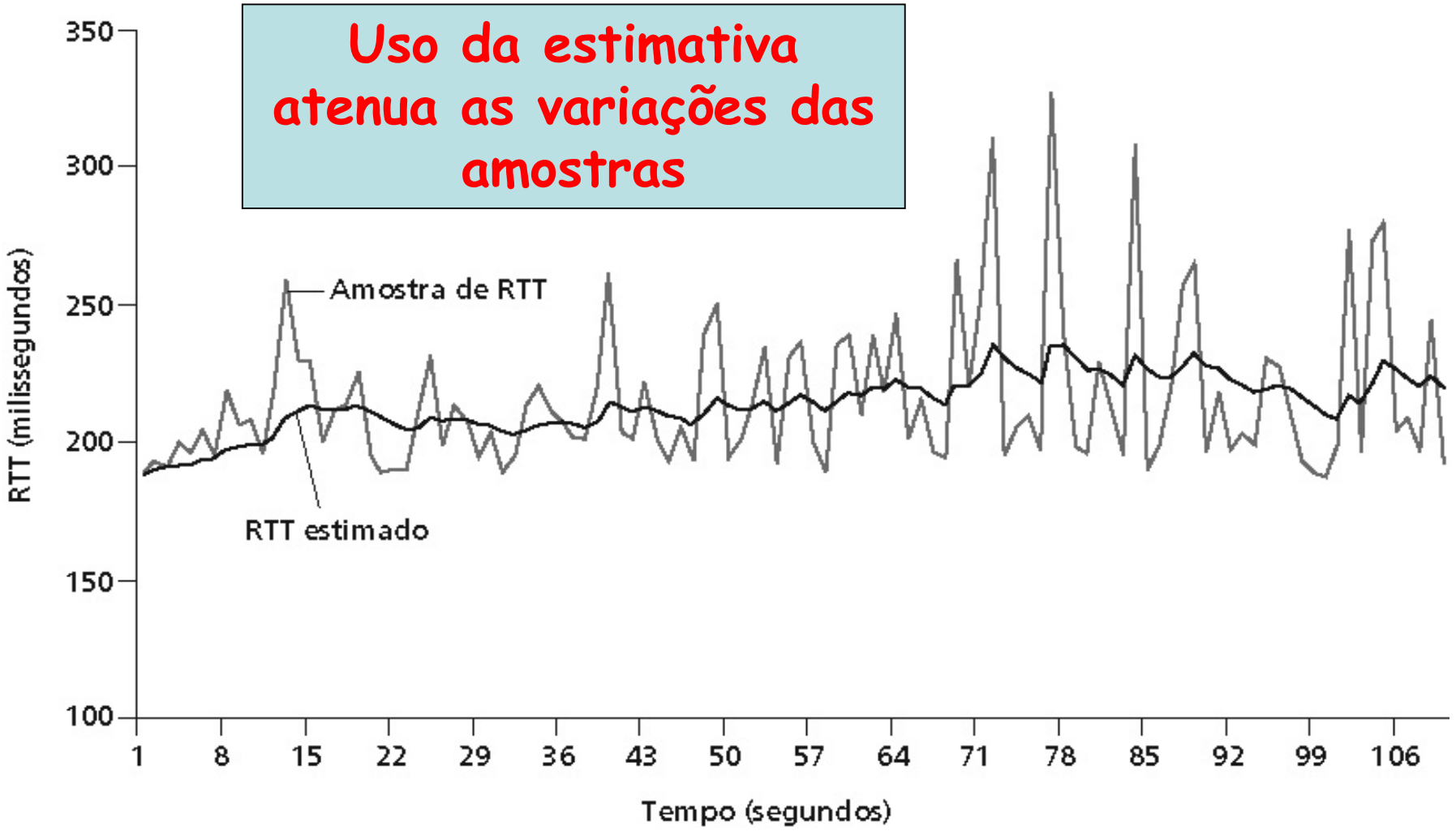
Temporização

- Cálculo do `EstimatedRTT`:

$$\mathbf{EstimatedRTT = (1-\alpha)* EstimatedRTT + \alpha*SampleRTT}$$

  - Média móvel exponencialmente ponderada
  - Influência de uma amostra **diminui** exponencialmente no tempo
  - Valor típico de α = 0,125

Temporização



Temporização

- Intervalo de temporização é somado a uma "margem de segurança"
  - Definida pela **desvio das amostras** em relação à `EstimatedRTT`

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

(valor típico de $\beta = 0{,}25$)

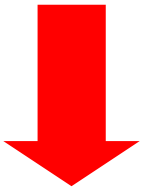
- Temporizador é definido por:

$$\mathbf{TimeoutInterval = EstimatedRTT + 4*DevRTT}$$

- Ideia básica: ***timeout*** maior quando o RTT está mais variável, menor quando RTT mais estável

Transferência Confiável do TCP

- Provê um **serviço confiável sobre o serviço não confiável** do protocolo IP
  - O IP não garante a entrega dos datagramas em ordem
    - Transbordo dos roteadores e problemas de ordenamento
  - O IP não garante a integridade dos dados
    - Os bits podem ser corrompidos



O TCP então cria um serviço confiável sobre o serviço de melhor esforço do IP

Transferência Confiável do TCP

- Para prover esse serviço confiável...
  - TCP garante que a cadeia de dados lida em um buffer de recepção é **exatamente a mesma** enviada
    - Segmentos transmitidos em "paralelo"
      - Princípio do *Go-Back-N*
    - ACKs cumulativos
    - Único temporizador para retransmissões

Transferência Confiável do TCP

- As retransmissões são disparadas por:
  - Estouros de temporização
  - ACKs duplicados

Transmissor TCP

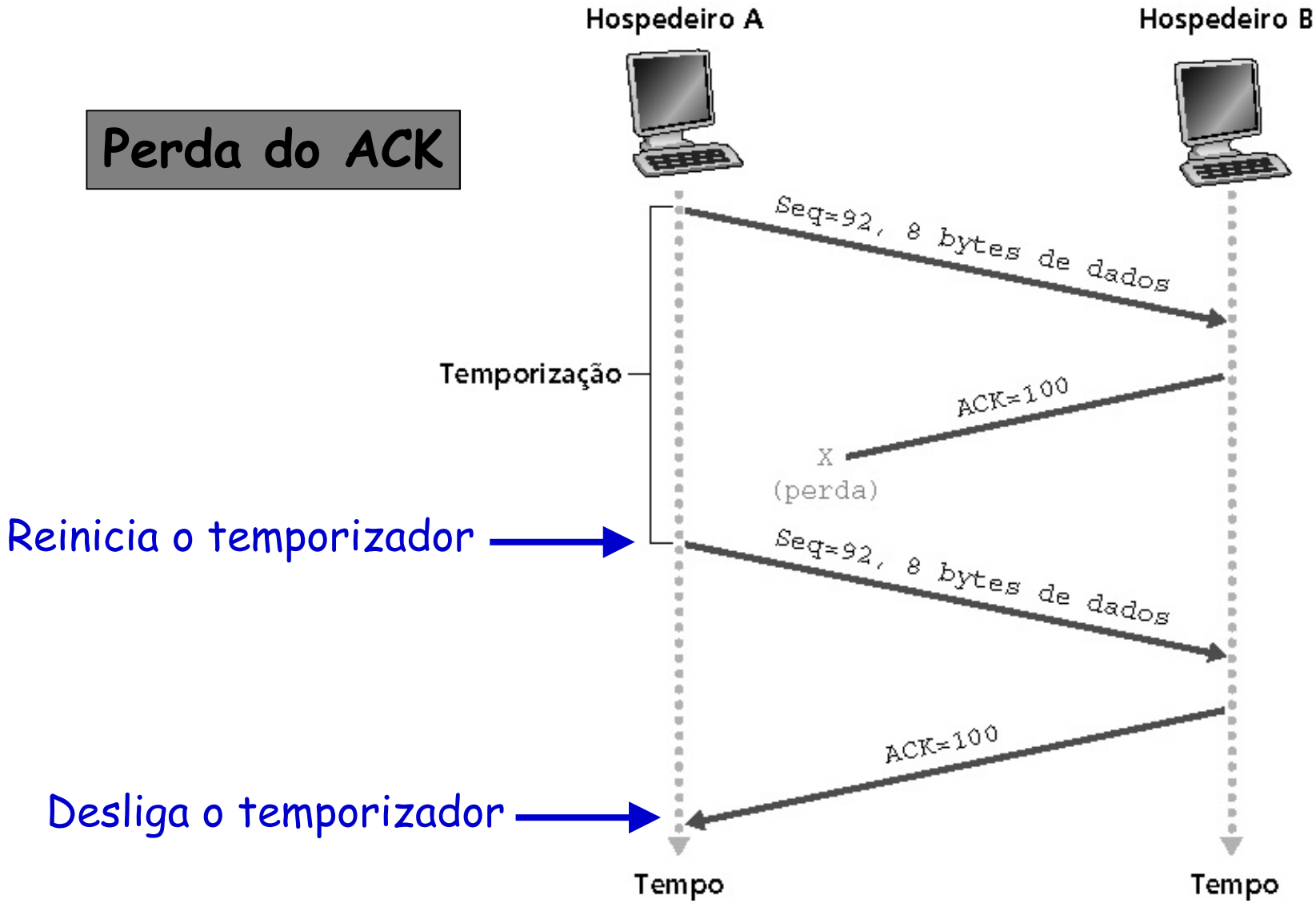
- Transmissor TCP **simplificado**
  - Ignora *ACKs* duplicados
  - Ignora controles de fluxo e de congestionamento
- Ao receber os dados da aplicação
  - Cria segmento com número de sequência (nseq)
    - nseq é o número de sequência do primeiro byte de dados do segmento
  - Dispara o temporizador
    - Se já não estiver disparado
    - Relativo ao segmento mais antigo ainda não reconhecido
    - Valor calculado previamente

Transmissor TCP

- Quando ocorre um estouro do temporizador
  - Retransmitir o segmento que causou o estouro do temporizador
  - Reiniciar o temporizador

- Quando um ACK é recebido
  - Se reconhecer segmentos ainda não reconhecidos
    - Atualizar informação sobre o que foi reconhecido
    - Disparar novamente o temporizador se ainda houver segmentos não reconhecidos

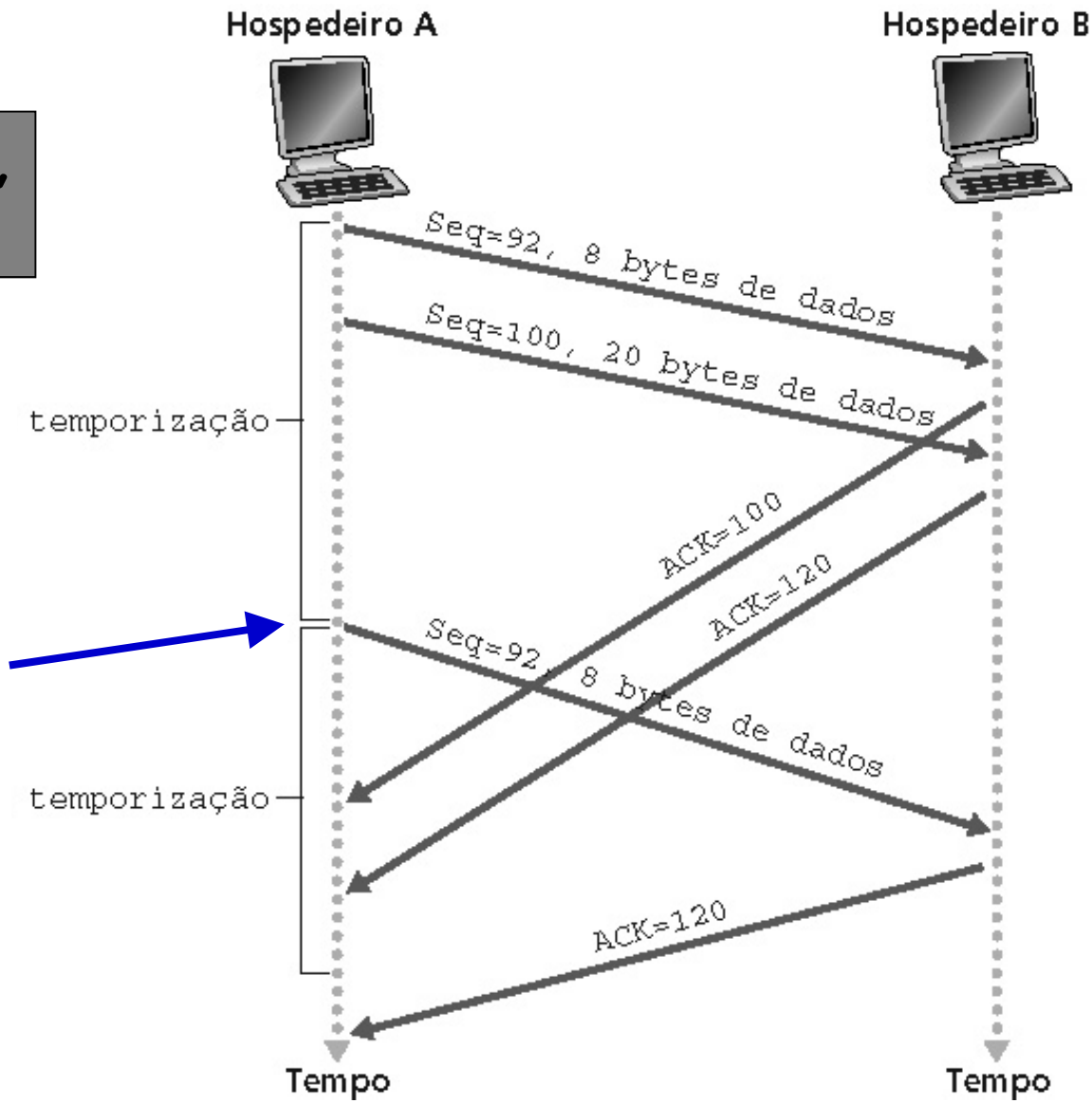
Cenário de Retransmissão



Cenário de Retransmissão

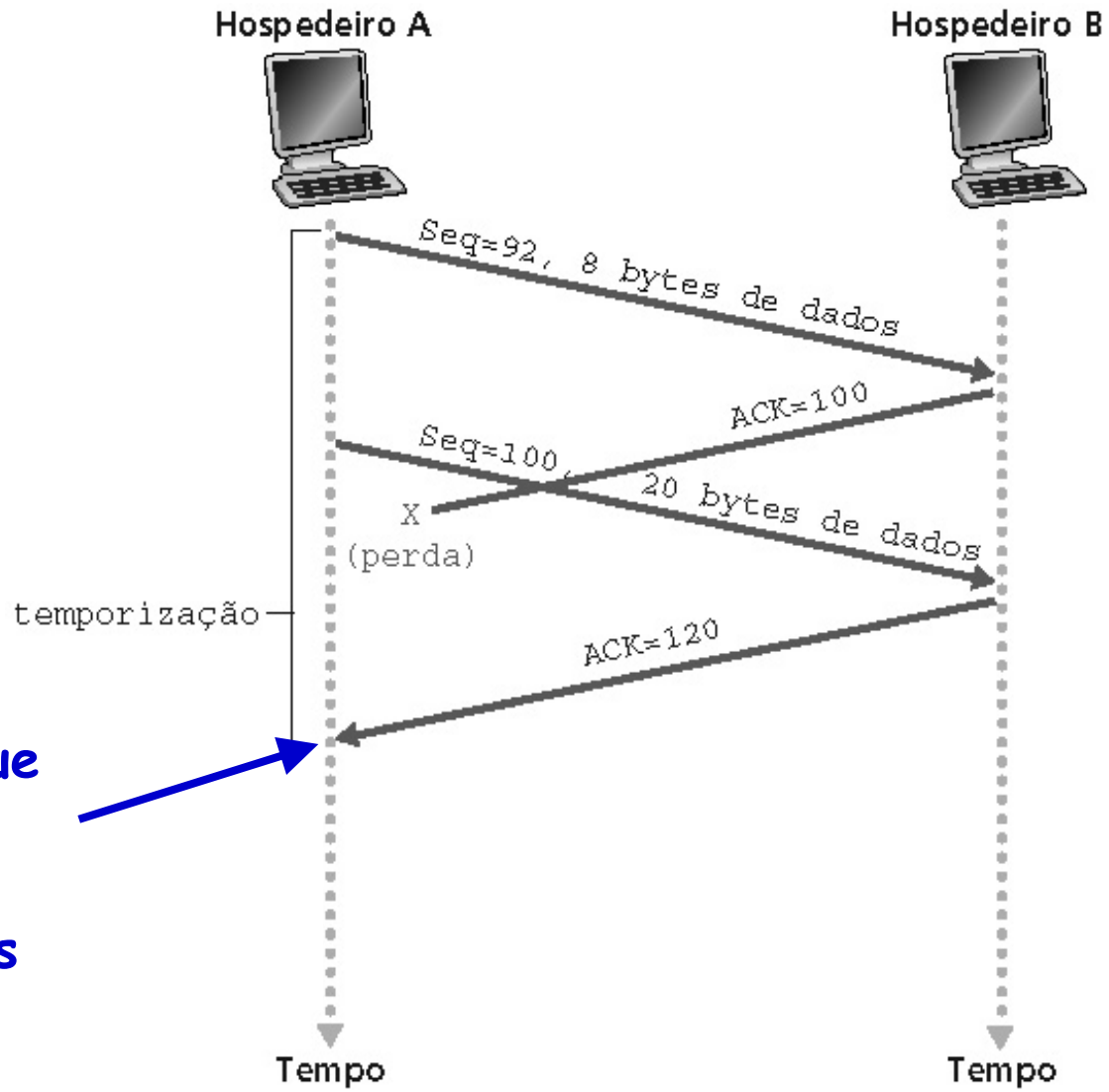
Estouro prematuro, ACKs cumulativos

Reinicia o temporizador, só há retransmissão do primeiro segmento. O segundo segmento só será transmitido se um ACK cumulativo não for recebido



Cenário de Retransmissão

Perda de um ACK cumulativo



Desliga o temporizador, já que o ACK cumulativo reconhece os dois segmentos enviados

Geração de ACKs

Evento no receptor	Ação do receptor
Chegada de segmento em ordem sem lacunas. Anteriores já reconhecidos	ACK retardado. Espera até 500 ms pelo próximo 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 número de sequência do próximo byte esperado
Chegada de segmento que preenche a lacuna parcial ou completamente	ACK imediato se segmento começa no início da lacuna

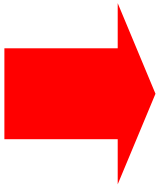
Retransmissão Rápida

- Se o intervalo do temporizador for grande...
  - A espera para retransmissão de um pacote perdido pode levar à queda de desempenho
- Forma alternativa para detectar segmentos perdidos: **Através de ACKs duplicados**
  - O transmissor normalmente envia diversos segmentos
    - Se um segmento se perder, provavelmente haverá muitos ACKs duplicados

Retransmissão Rápida

- Se o intervalo do temporizador for grande...
  - A espera para retransmissão de um pacote perdido pode levar a queda de desempenho

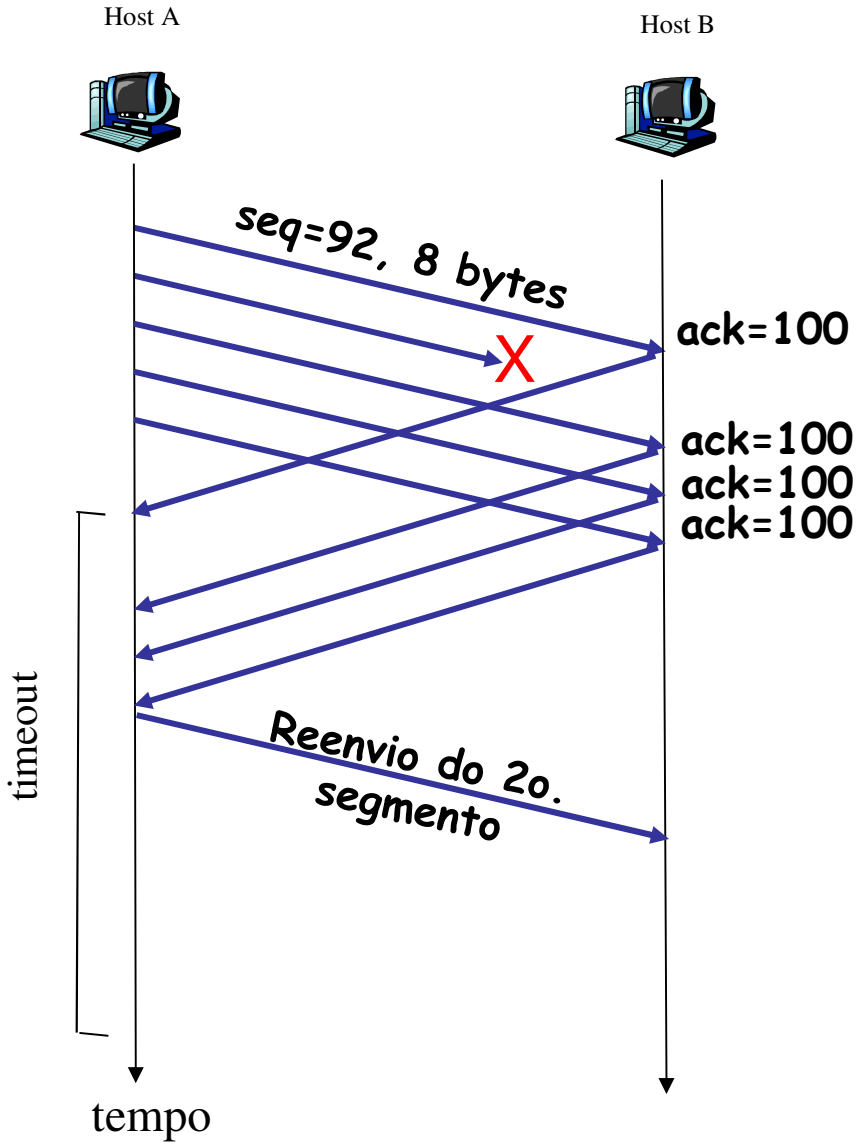
- Como é feita a detecção de segmentos perdidos através de ACKs duplicados?
  - Se o transmissor receber **três ACKs duplicados** para o mesmo segmento...
    - Assume-se que o segmento após o último reconhecido se perdeu



Retransmite o segmento antes que o temporizador estoure

Retransmissão Rápida

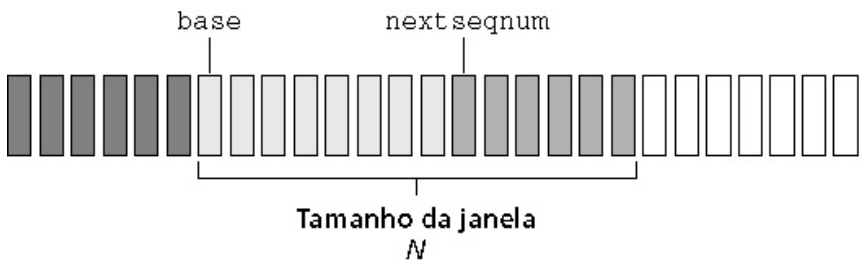
Retransmissão de um segmento após três ACKs duplicados



Retransmissão Rápida

event: recebido ACK, com valor do campo ACK de y

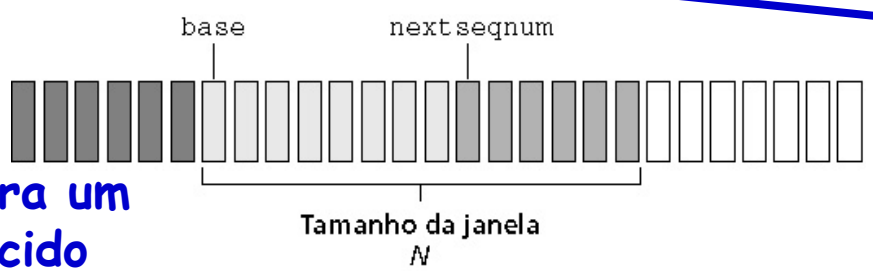
```
if (y > SendBase) {
   SendBase = y
   if (houver segmentos ainda não reconhecidos)
     liga temporizador
   else desliga temporizador
} else {
  incrementa contador de ACKs duplicados recebidos para y
  if (contador de ACKs duplicados recebidos para y = 3) {
    retransmita segmento com número de sequência y
  }
}
```



Retransmissão Rápida

```
event: recebido ACK, com valor do campo ACK de y
if (y > SendBase) {
   SendBase = y
   if (houver segmentos ainda não reconhecidos)
     liga temporizador
   else desliga temporizador
} else {
   incrementa contador de ACKs duplicados recebidos para y
   if (contador de ACKs duplicados recebidos para y = 3) {
     retransmita segmento com número de seqüência y
   }
}
```

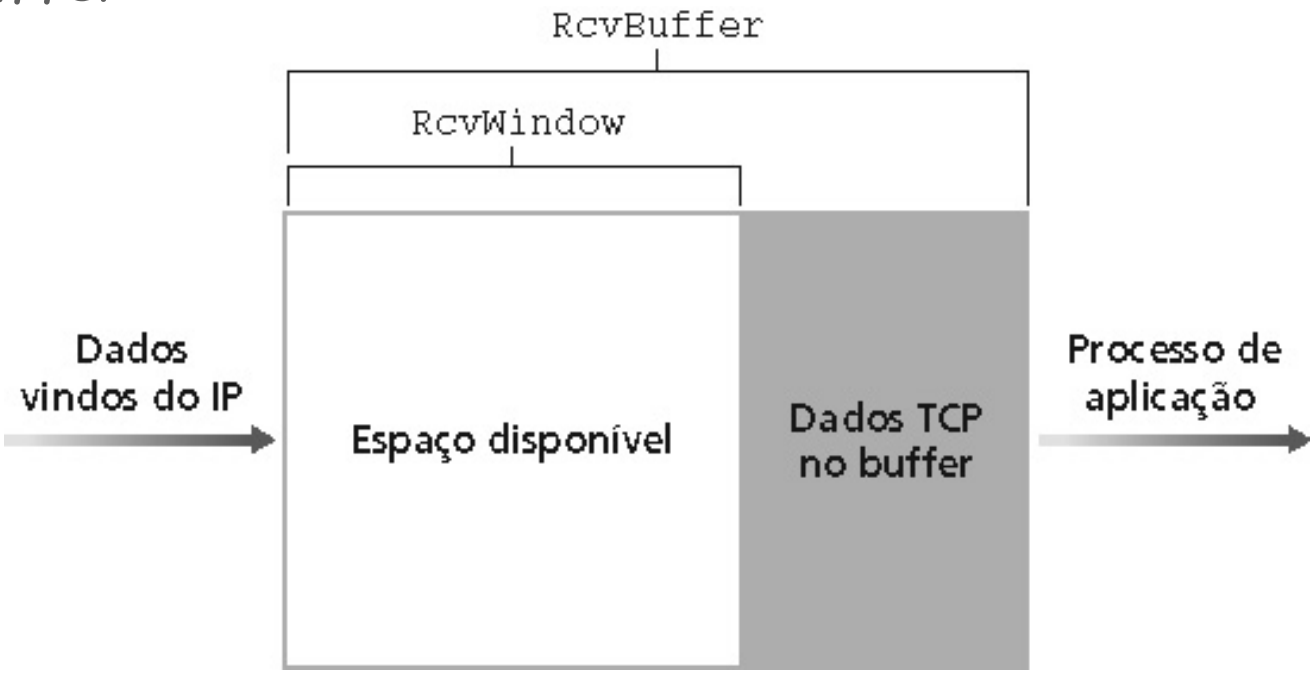
um ACK duplicado para um segmento já reconhecido



Retransmissão rápida

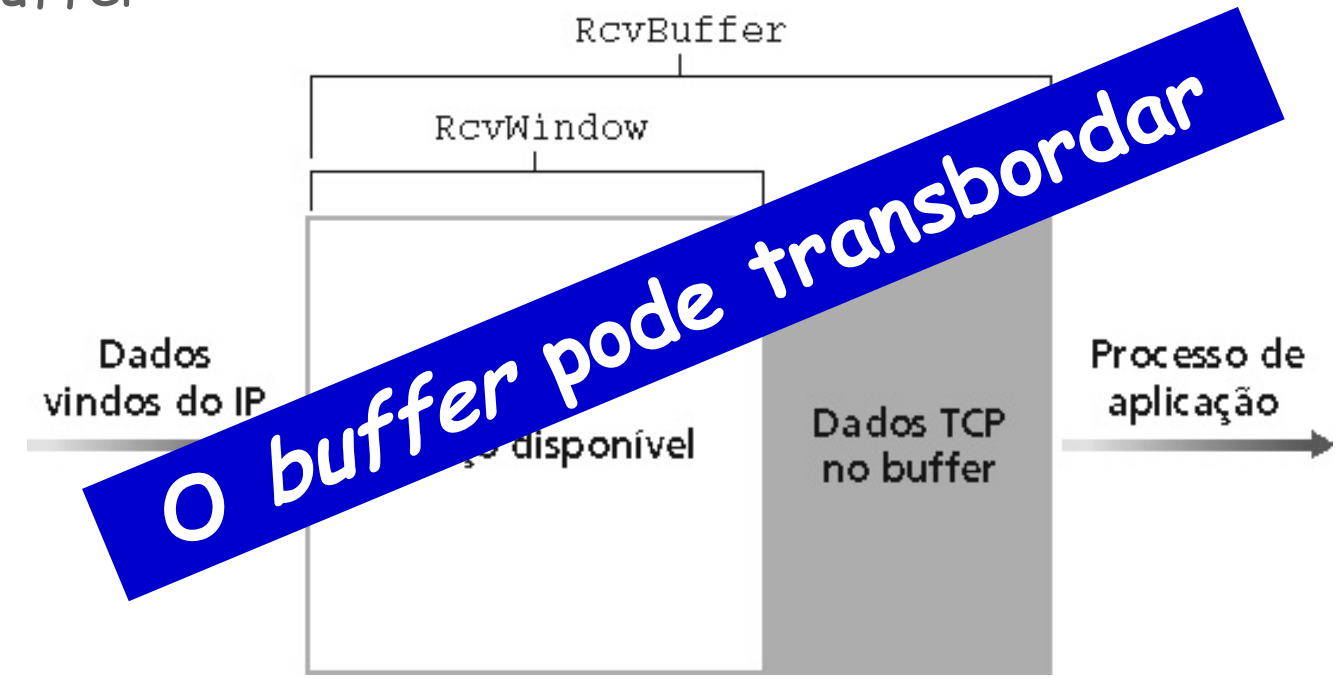
Controle de Fluxo do TCP

- Receptor possui um buffer de recepção
  - Processos das aplicações podem demorar a ler do buffer



Controle de Fluxo do TCP

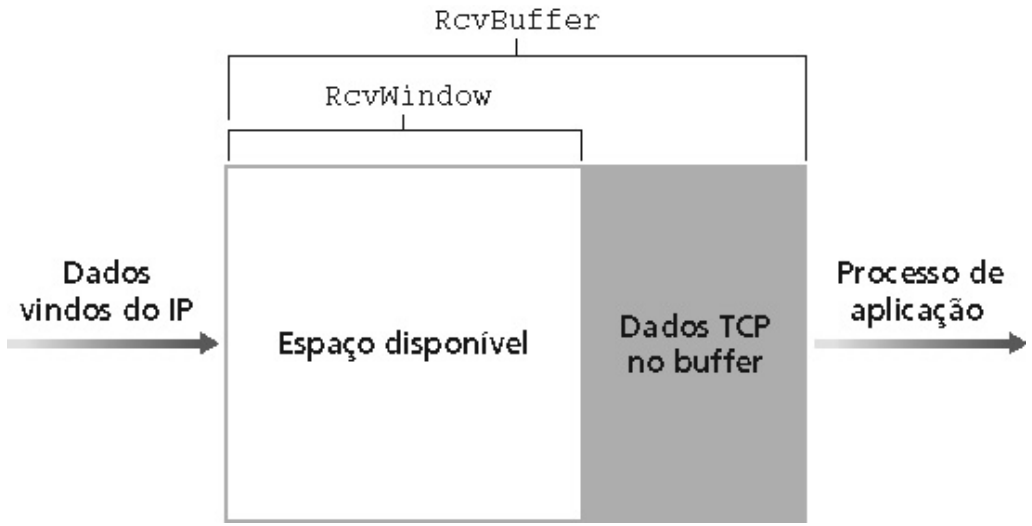
- Receptor possui um buffer de recepção
  - Processos das aplicações podem demorar a ler do buffer



Controle de Fluxo do TCP

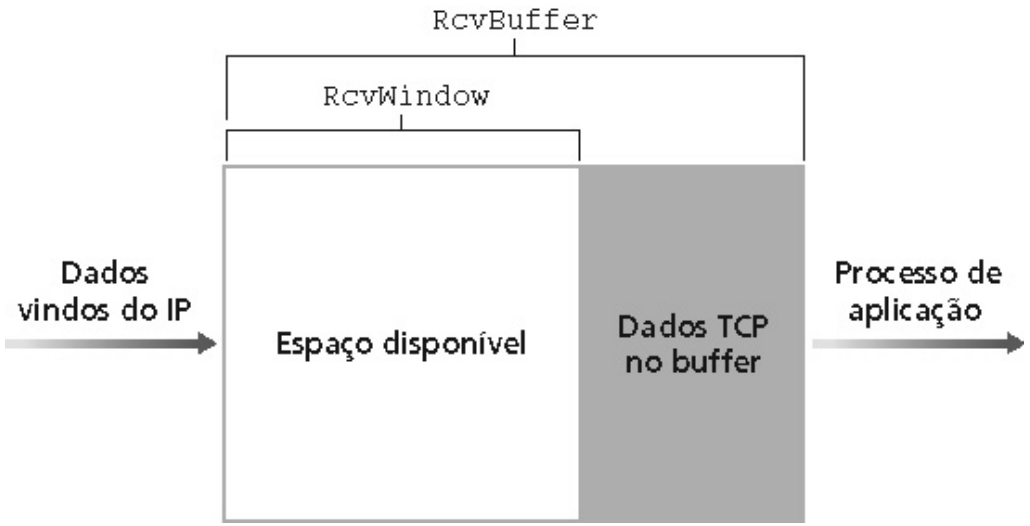
- Funcionamento
  - Suposição:
    - Receptor descarta segmentos recebidos fora de ordem
  - Espaço livre no buffer = Janela de recepção

RcvWindow = RcvBuffer – [LastByteRcvd – LastByteRead]



Controle de Fluxo do TCP

- Funcionamento
  - O receptor anuncia o espaço livre no buffer
    - O valor da janela (`RcvWindow`) é informado nos segmentos
  - O transmissor limita os dados não reconhecidos ao tamanho da janela de recepção
    - Garante que o **buffer do receptor não transbordará**



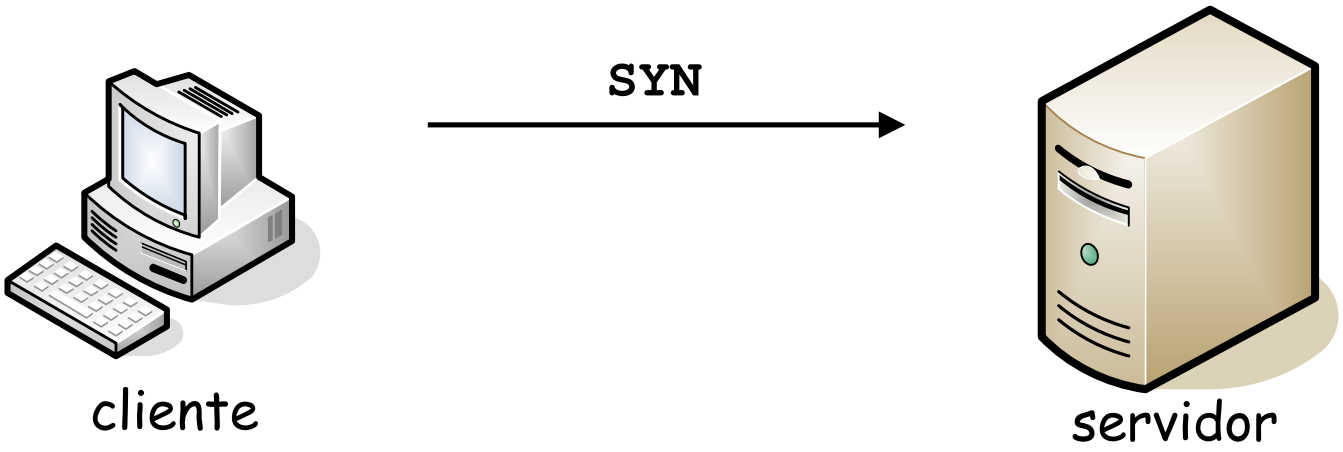
Estabelecimento de Conexão

- É feita **antes da troca de dados**

- Inicialização de variáveis
  - Números de sequência
  - Tamanho dos buffers,
  - Variáveis do mecanismo de controle de fluxo
    - Janela de recepção (`RcvWindow`)
  - etc.

Estabelecimento de Conexão

" Three-way handshake"

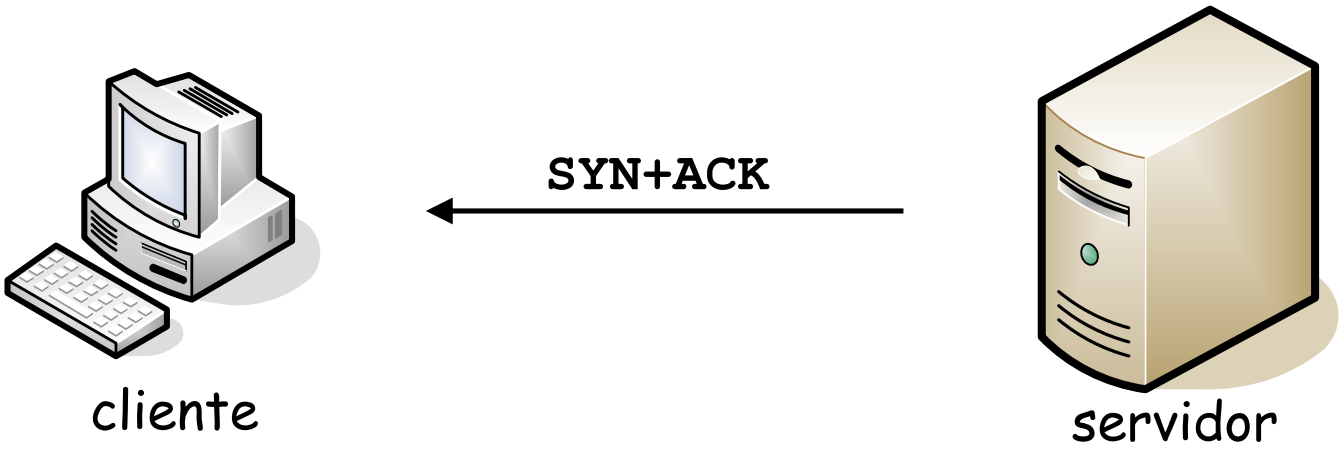


1. Cliente envia segmento de controle SYN para o servidor

   Especifica o número de sequência inicial e não envia dados

Estabelecimento de Conexão

" Three-way handshake"

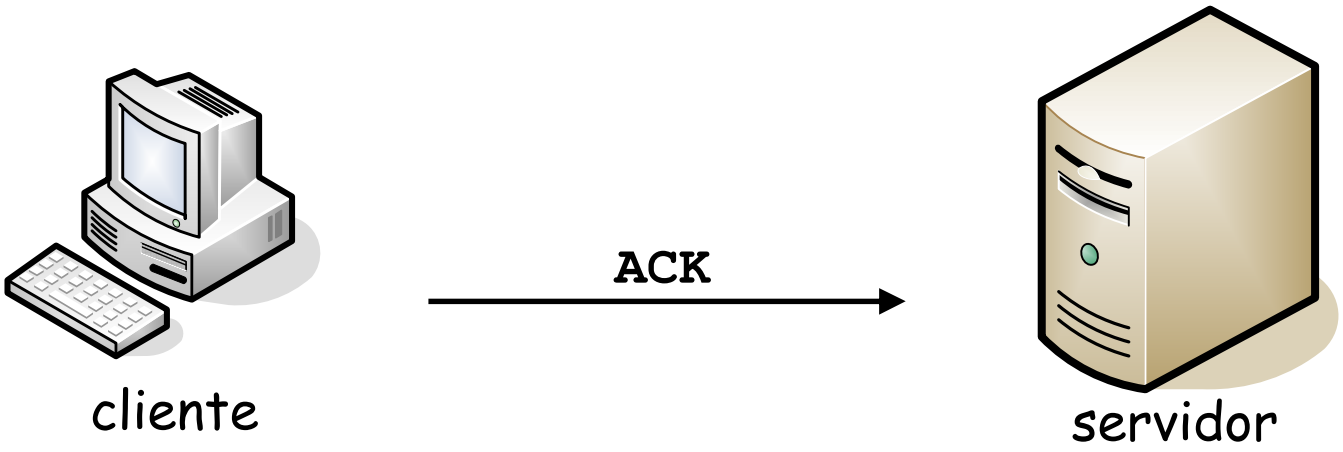


2. Ao receber o SYN, o servidor responde com segmento de controle SYN+ACK

Define o tamanho dos *buffers* e
especifica o número inicial de sequência do servidor para o receptor

Estabelecimento de Conexão

" Three-way handshake"



3. Ao receber SYN+ACK, o cliente responde com segmento ACK

Pode conter dados (*piggyback*)

Estabelecimento de Conexão

- Como uma estação poderia escolher o número de sequência inicial?
  - Pacote enviado com número de sequência N deve desaparecer da rede após o intervalo T (=120s)
    - Caso contrário, pacotes diferentes com o mesmo número de sequência podem coexistir

Basta calcular quantos pacotes uma fonte pode enviar no intervalo T e garantir que o números de sequência suficientemente grandes!

Estabelecimento de Conexão

- Como uma estação poderia escolher o número de sequência inicial?
  - Pacote enviado com número de sequência N deve desaparecer da rede após o intervalo T (=120s)
    - Senão, pacotes diferentes com o mesmo número de sequência podem coexistir

E se a máquina parar inesperadamente, o que aconteceria com a contagem do número de sequência?

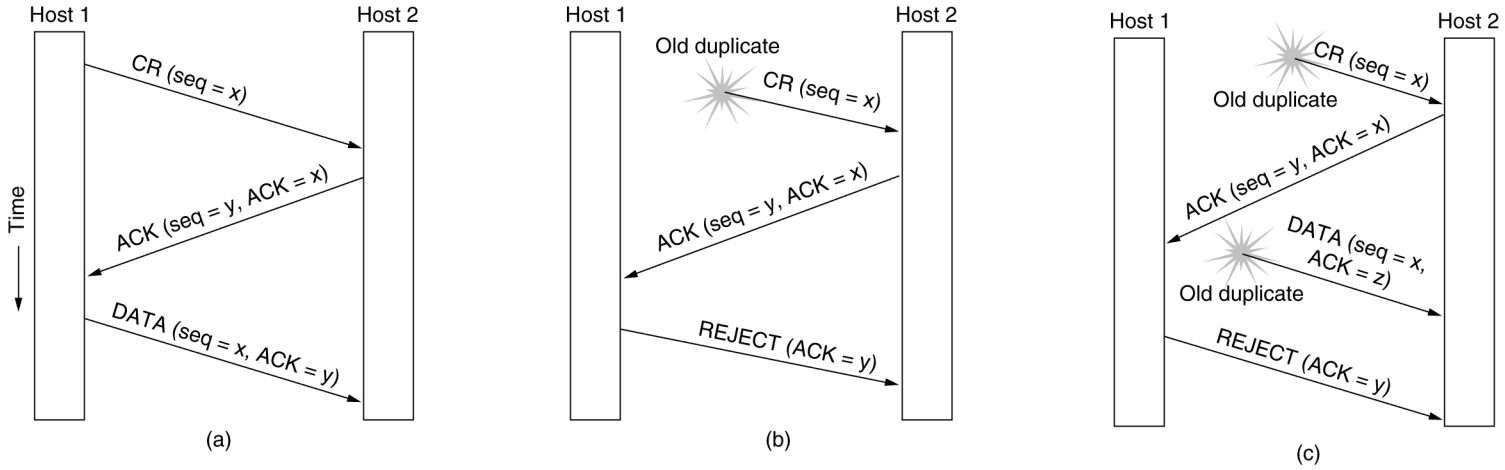
A contagem recomeçaria e não haveria como garantir a ausência de números de sequência repetidos... Nesse caso, como resolver?

Estabelecimento de Conexão

- Solução:
  - Escolha de números de sequência aleatórios
  - *Three-way handshake* ("apresentação de três vias")

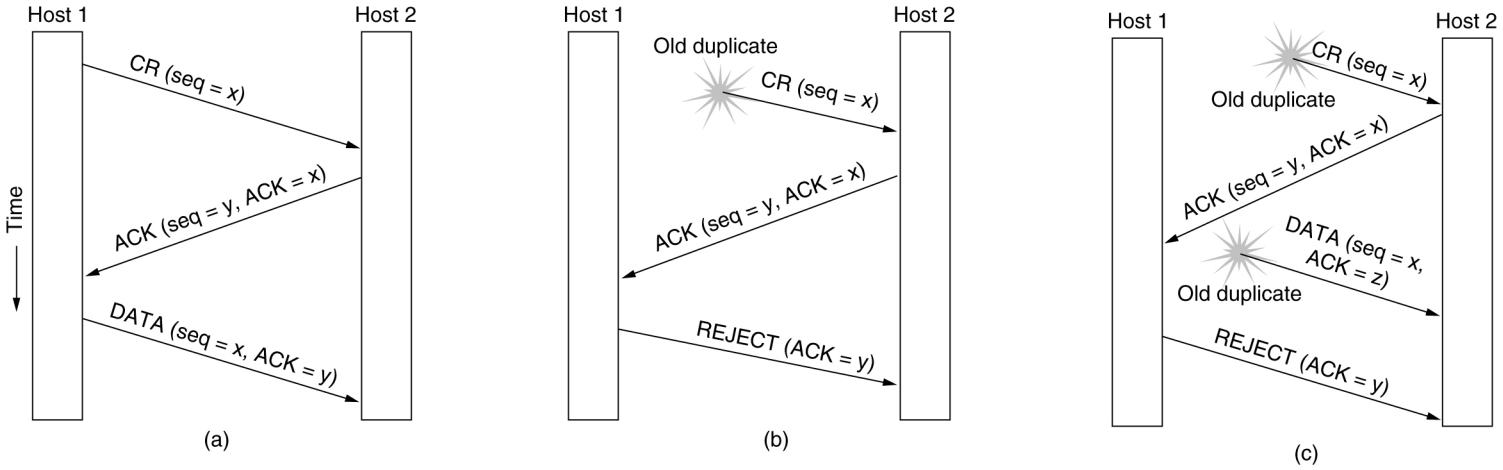
Estabelecimento de Conexão

- Por que o *three-way handshake* resolve?
  a) Operação normal: números de sequência x e y usados
  b) Abertura de conexão (CR) com número de sequência duplicado chega no Host 2 sem que o Host 1 saiba
     - Host 2 reconhece caso seja uma nova conexão, mas Host 1 rejeita ACK pois x é um número de sequência repetido

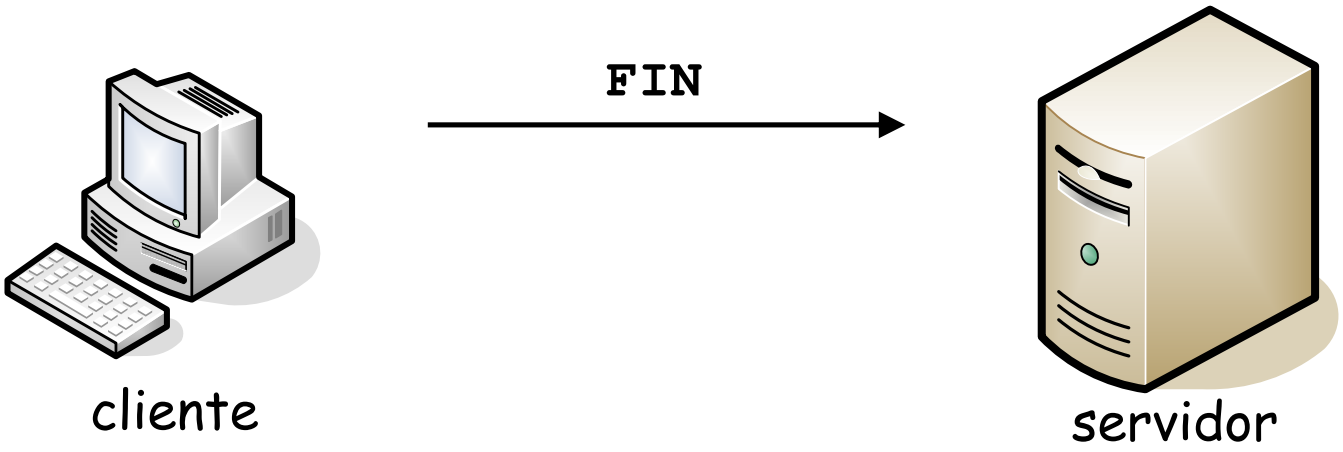


Estabelecimento de Conexão

- Por que o *three-way handshake* resolve?
  - c) Abertura de conexão e DATA duplicados
    - Host 2, assim como em (b), responde CR pois pode se tratar de uma nova conexão
    - Host 1 rejeita ACK pois x é repetido e Host 2 rejeitaria o DATA pois desconhece o número de sequência z
      - Host 2 estaria esperando ACK de y



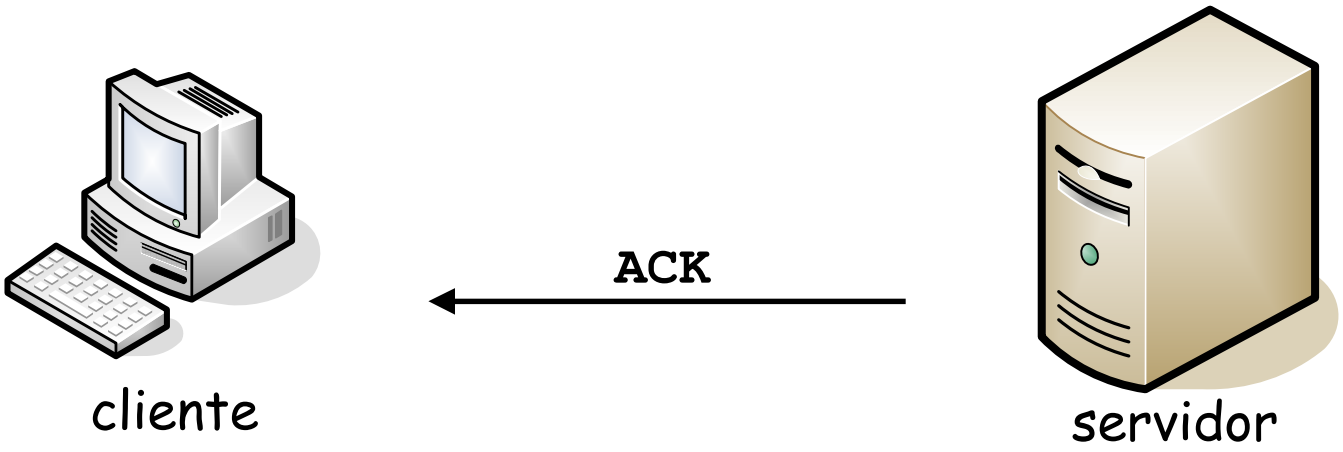
Encerramento de Conexão



1. Cliente envia segmento de controle FIN ao servidor

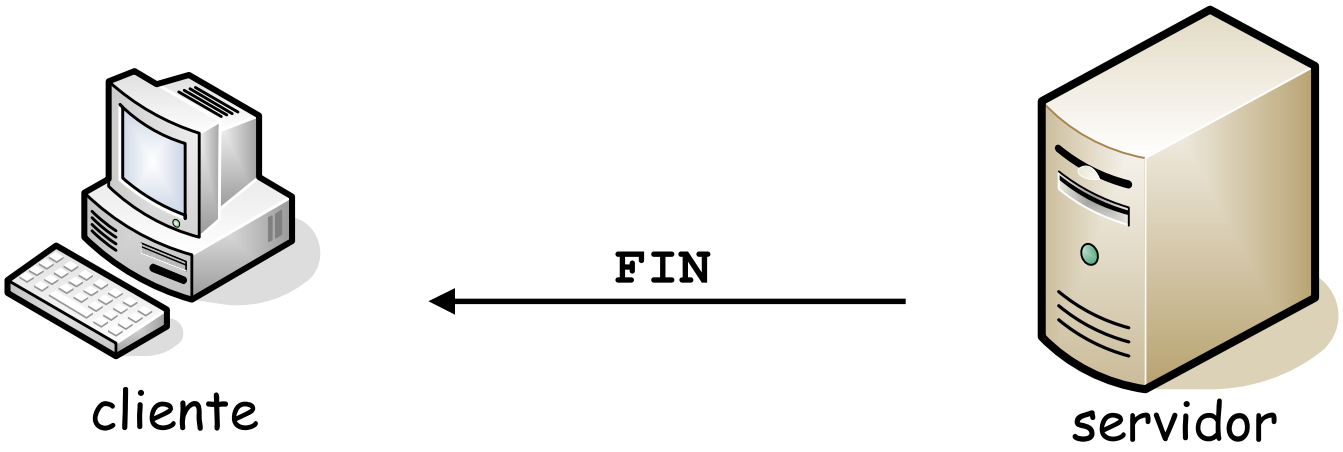
Qualquer um dos lados pode iniciar o encerramento da conexão

Encerramento de Conexão



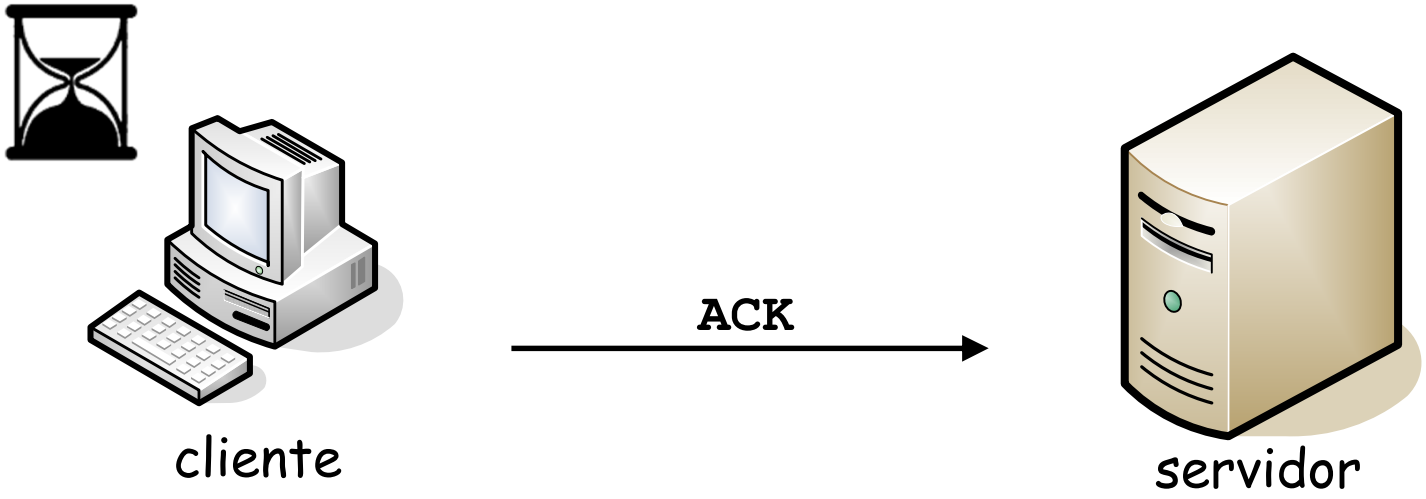
2. Ao receber FIN, o servidor responde com ACK

Encerramento de Conexão



3. Em seguida, o servidor envia FIN e encerra a conexão

Encerramento de Conexão

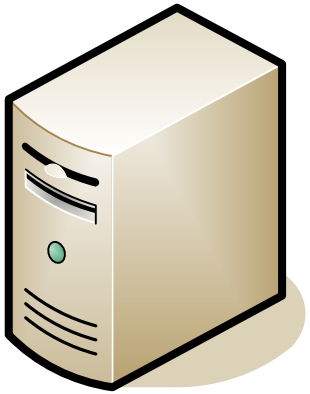


4. Ao receber FIN, o cliente responde com ACK

Cliente entra em "espera temporizada" → reenvio de ACK caso o anterior seja perdido

Encerramento de Conexão

cliente



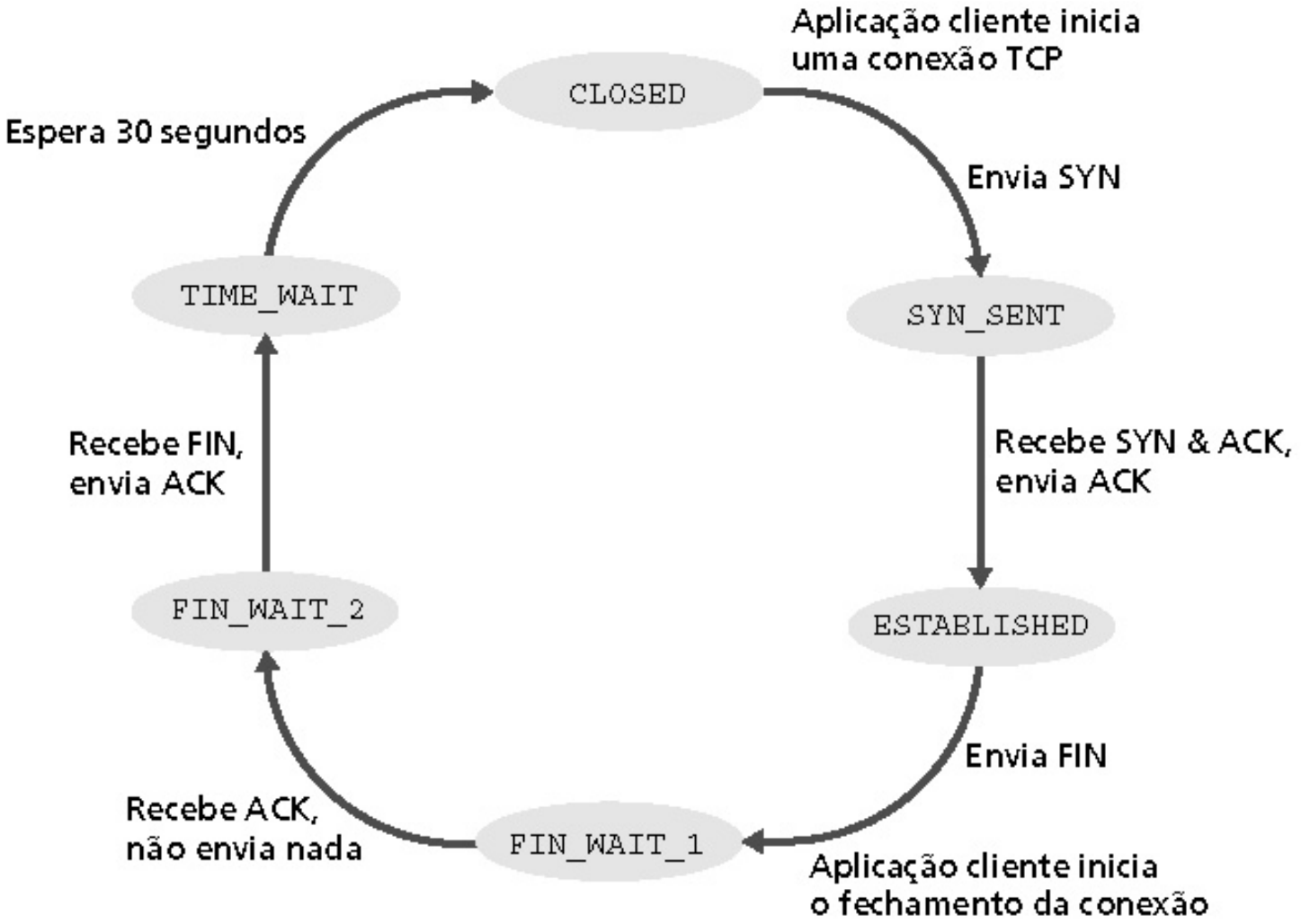
servidor

5. Quando o temporizador estoura a conexão é encerrada

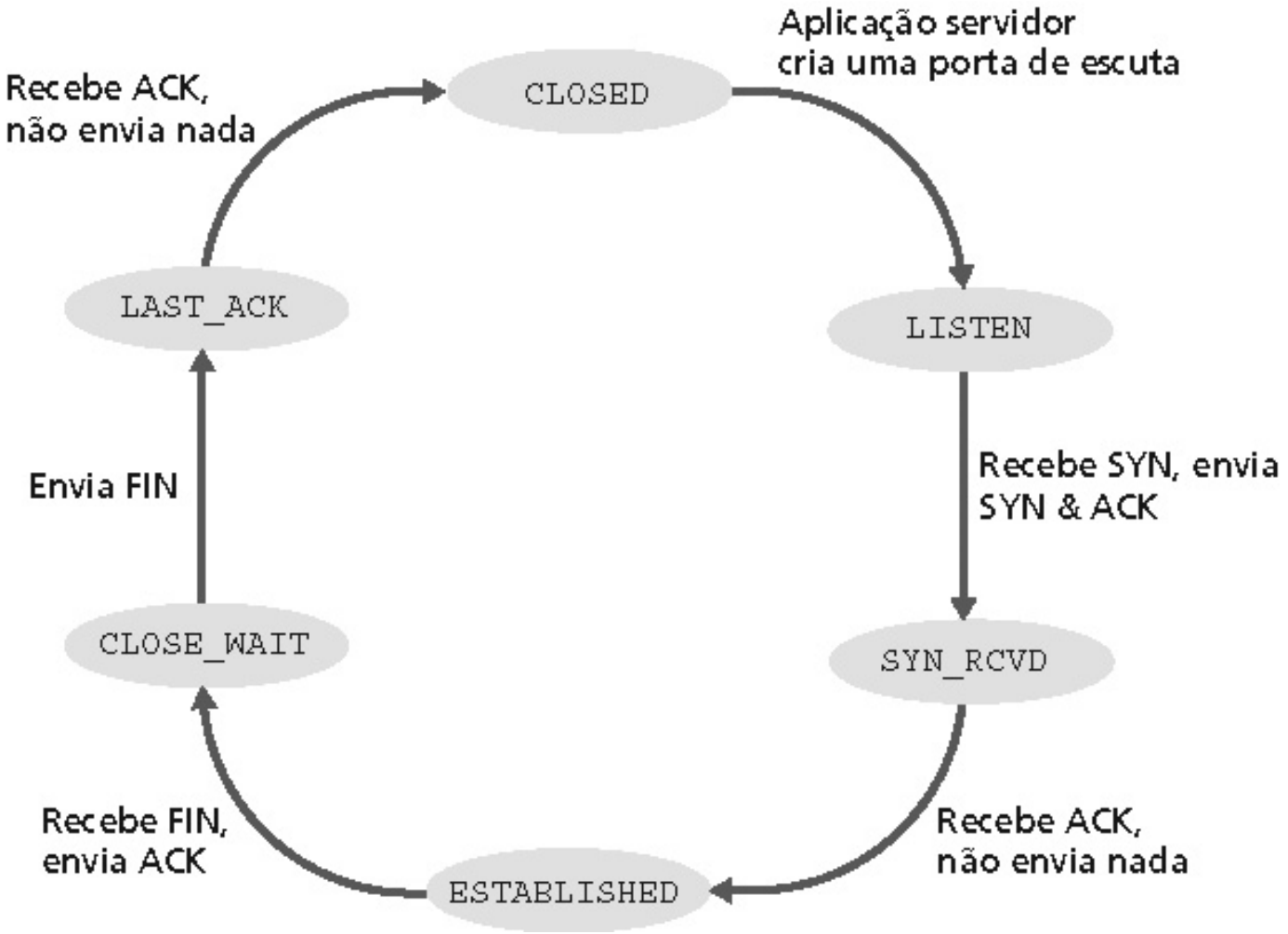
Encerramento de Conexão

- Conexão assimétrica
  - Basta que um nó desligue para que a conexão seja desfeita
    - Pode resultar em perda de dados caso o nó que desfez a conexão a faça enquanto o outro par ainda envia dados
- Conexão simétrica
  - Trata a conexão como duas conexões unidirecionais isoladas → Realizada pelo TCP
    - Um nó pode continuar a receber dados mesmo se já tiver solicitado o encerramento da conexão
      - Nós precisam solicitar desconexão de maneira independente

Estados no Cliente TCP

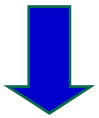


Estados no Servidor TCP



Controle de Congestionamento

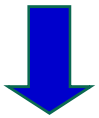
- Fontes enviam dados acima da capacidade da rede de tratá-los
  - Perda de pacotes
    - Saturação de buffers nos roteadores
  - Atrasos maiores
    - Espera nos buffers dos roteadores



A rede está congestionada!

Controle de Congestionamento

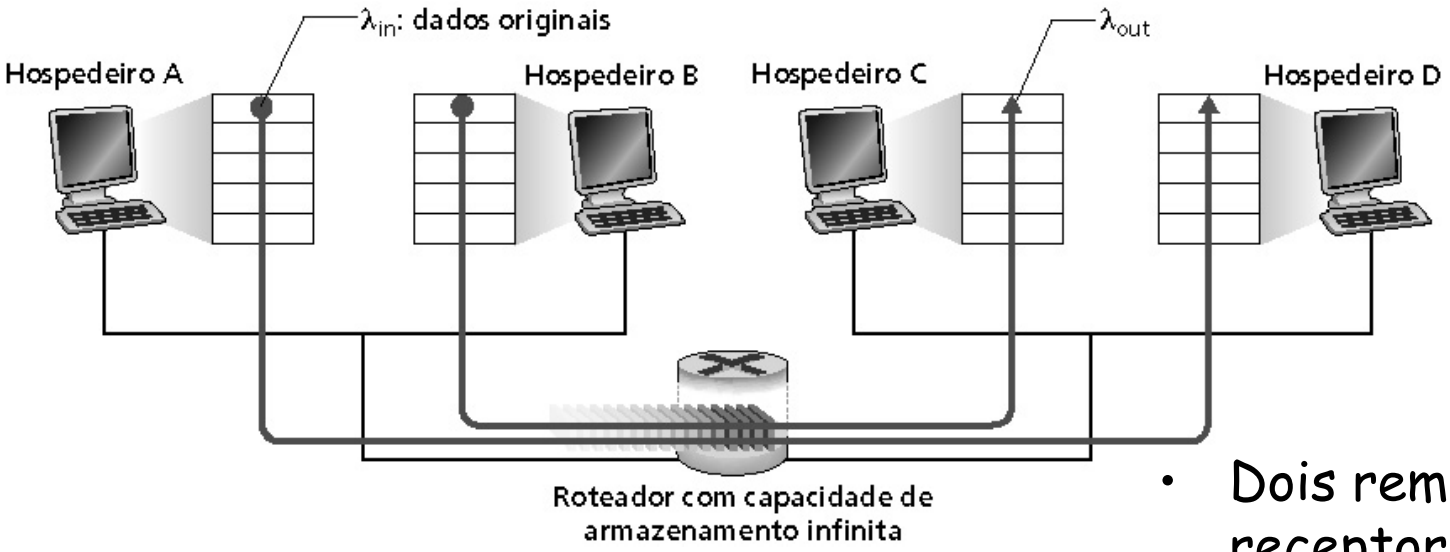
- Fontes enviam dados acima da capacidade da rede de tratá-los
  - Perda de pacotes
    - Saturação de buffers nos roteadores
  - Atrasos maiores
    - Espera nos buffers dos roteadores

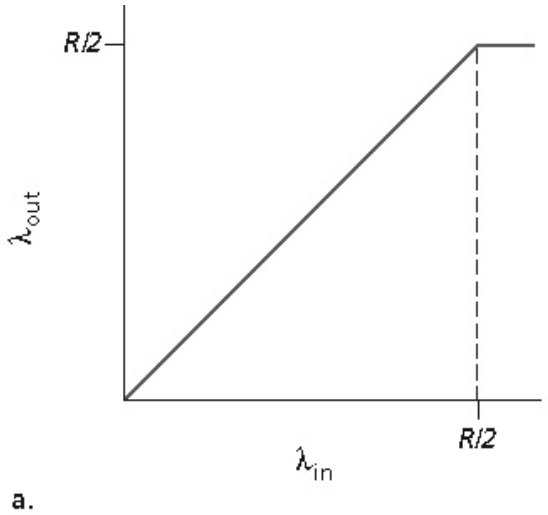


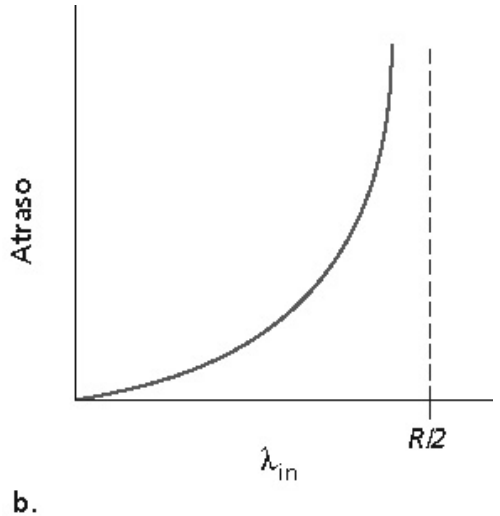
A rede está congestionada!

- **É diferente do controle de fluxo**
  - **É um estado da rede e não dos sistemas finais**

Congestionamento: Buffers Infinitos



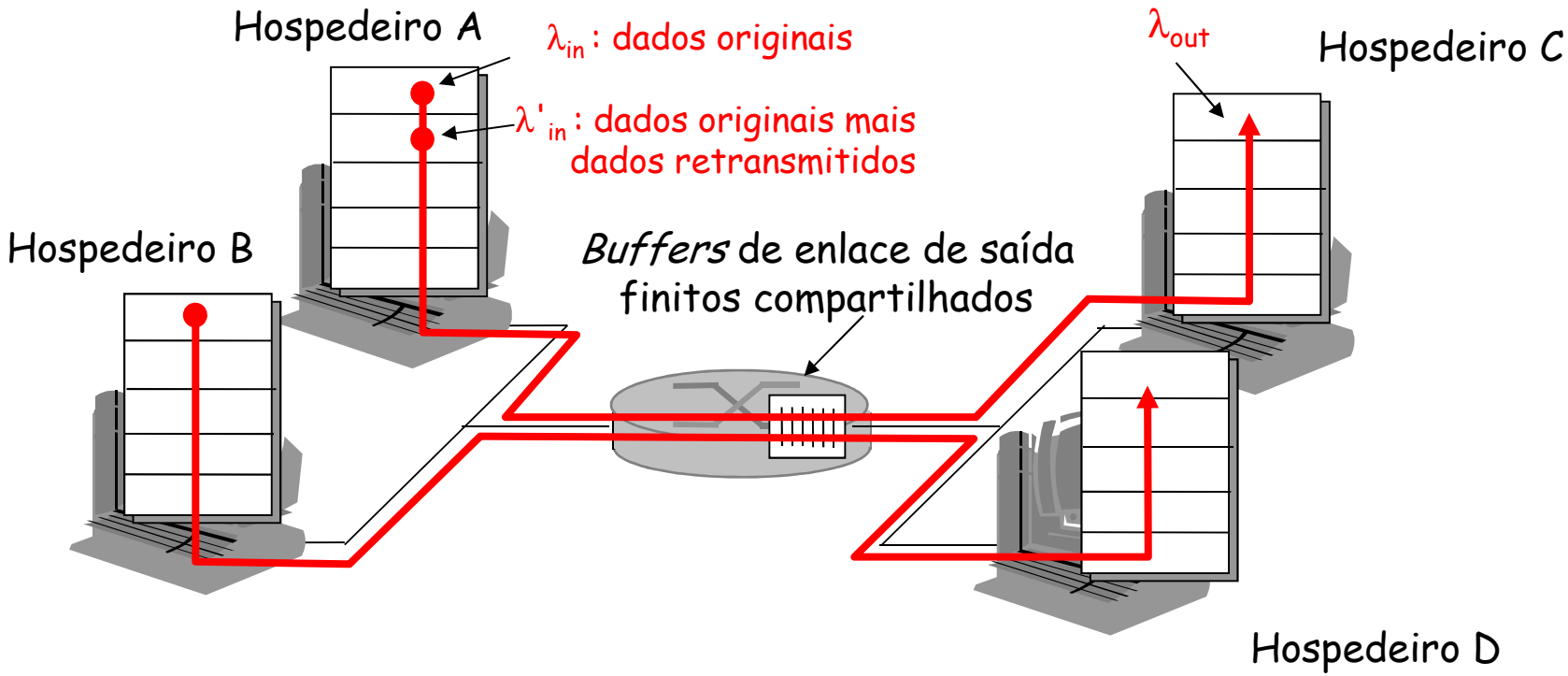




- Dois remetentes, dois receptores
- Um roteador com buffers infinitos
- Sem retransmissão
- **Grandes retardos quando congestionada**
- **Máxima vazão alcançável**

Congestionamento: Buffers Finitos

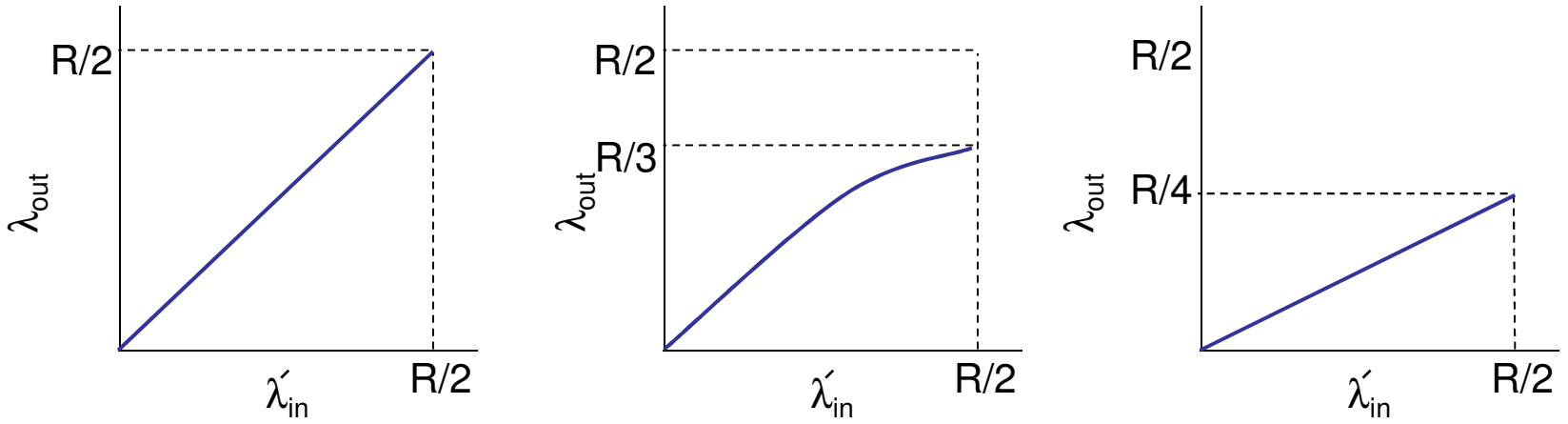
- Um roteador, buffers **finitos**
- Retransmissão pelo remetente de pacote perdido



Congestionamento: Buffers Finitos

a. Sempre: $\lambda_{in} = \lambda_{out}$ (goodput)

Transmissores conseguem descobrir quando o buffer do roteador está livre para evitar perdas

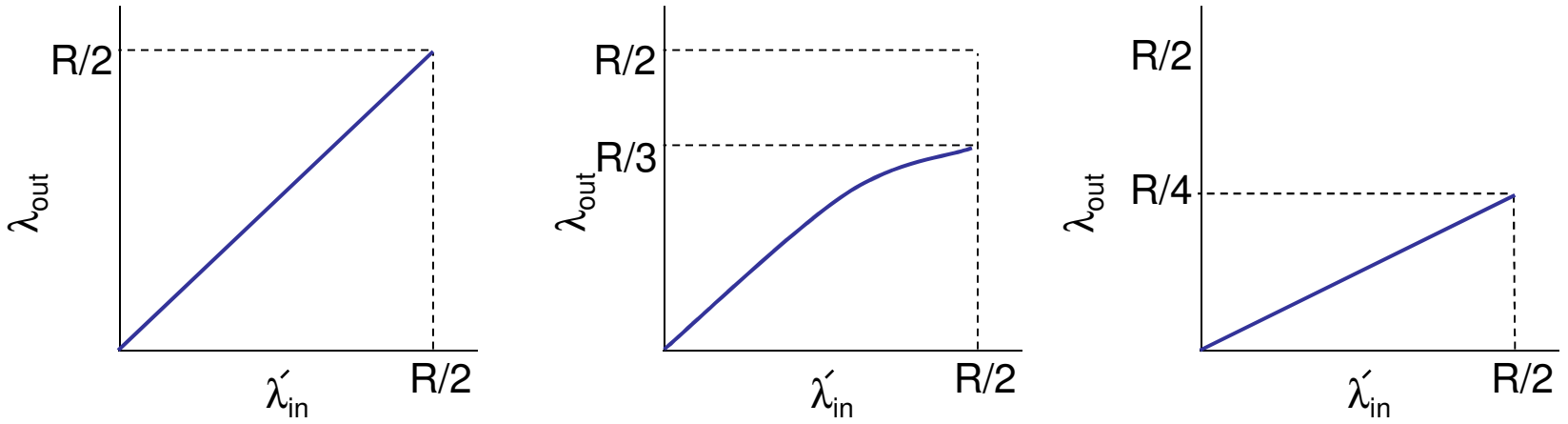


Congestionamento: Buffers Finitos

a. Sempre: $\lambda_{in} = \lambda_{out}$ (goodput)

b. Retransmissão "perfeita" apenas com perdas: $\lambda'_{in} > \lambda_{out}$

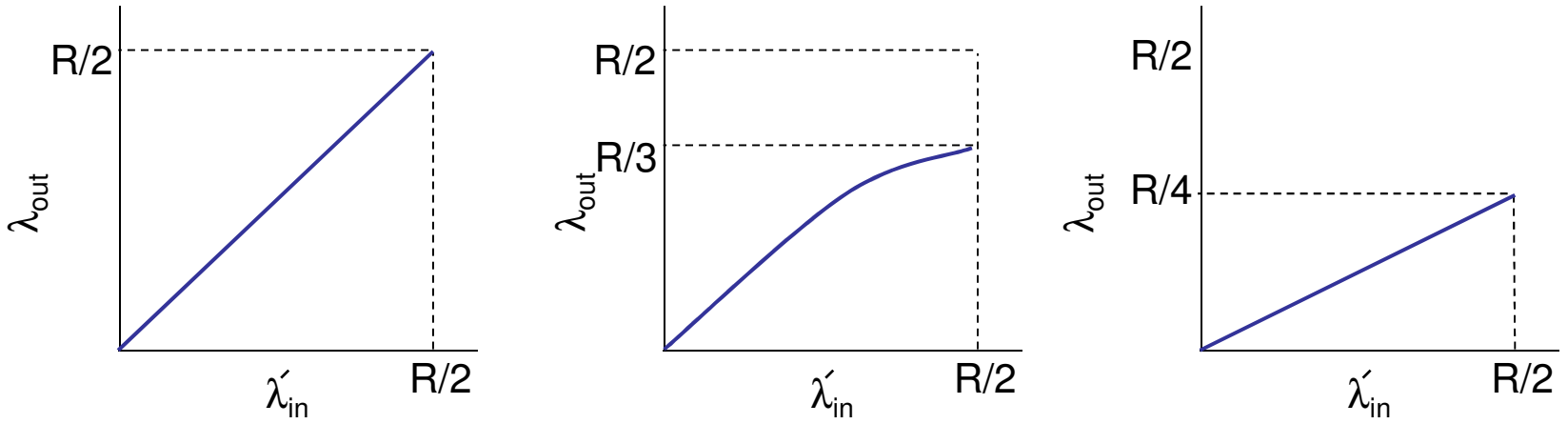
Transmissor sabe quando uma perda ocorre e ajusta o temporizador para retransmissão. A carga oferecida λ'_{in} é igual a taxa de transmissão + a de retransmissão



Congestionamento: Buffers Finitos

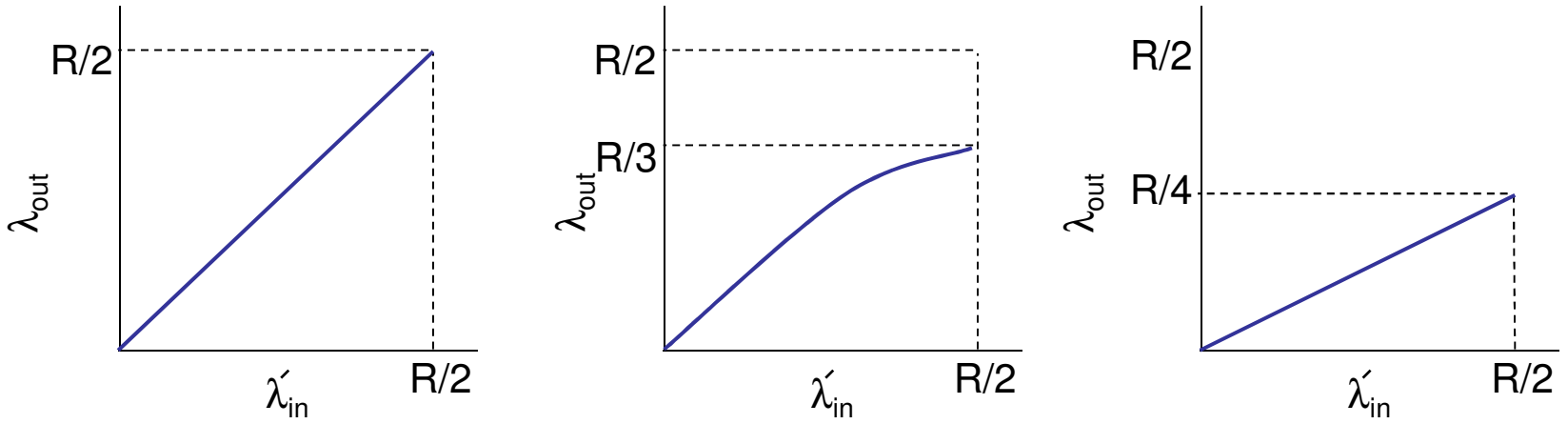
a. Sempre: $\lambda_{in} = \lambda_{out}$ (goodput)
b. Retransmissão "perfeita" apenas com perdas: $\lambda'_{in} > \lambda_{out}$
c. Retransmissão de pacotes atrasados (não perdidos) faz com que λ'_{in} seja maior (do que o caso perfeito) para o mesmo λ_{out}

Presença de pacotes duplicados



Congestionamento: Buffers Finitos

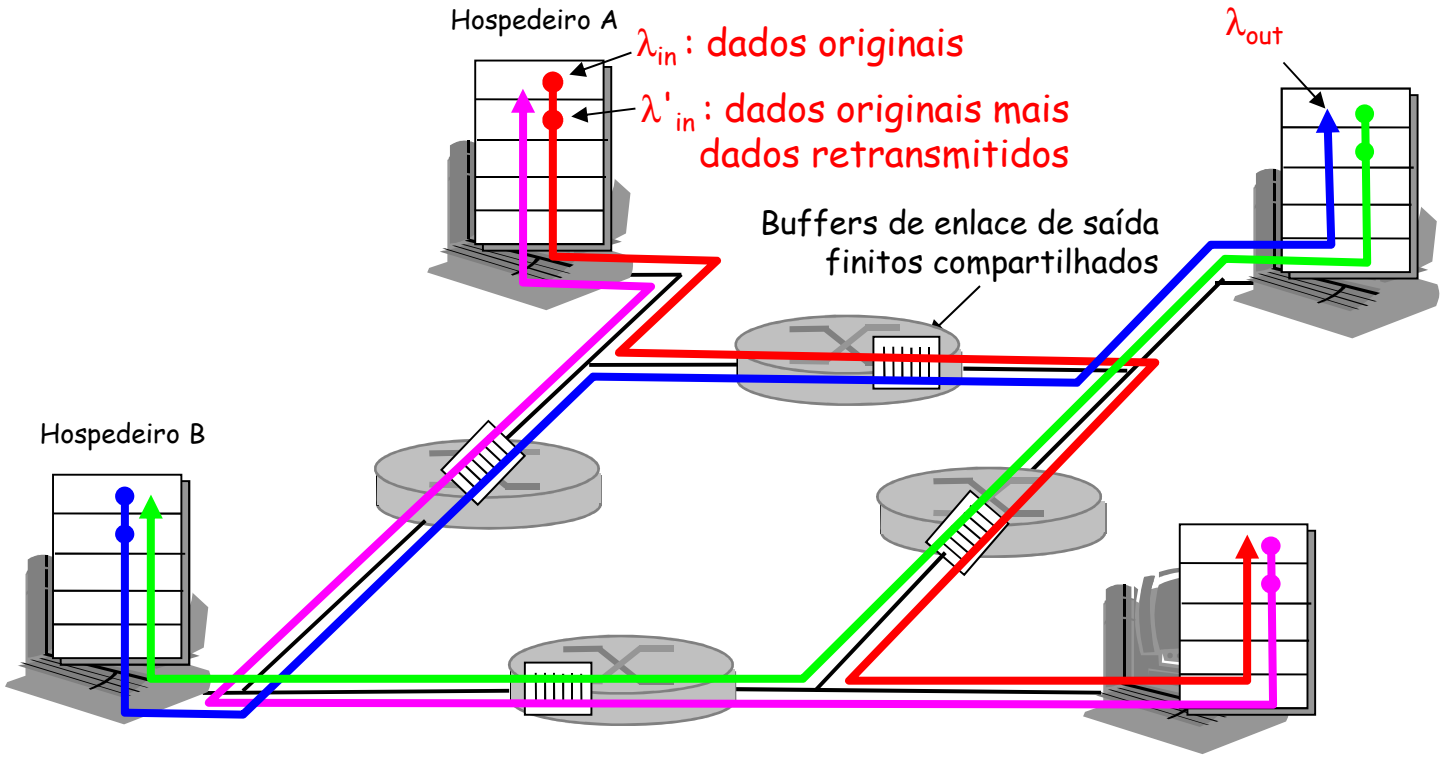
- "Custos" de congestionamento:
  - Mais trabalho (retransmissão) para um dado "goodput"
  - Retransmissões desnecessárias: são enviadas em média duas cópias do mesmo pacote (Caso da letra c.)



Congestionamento: Quatro Remetentes

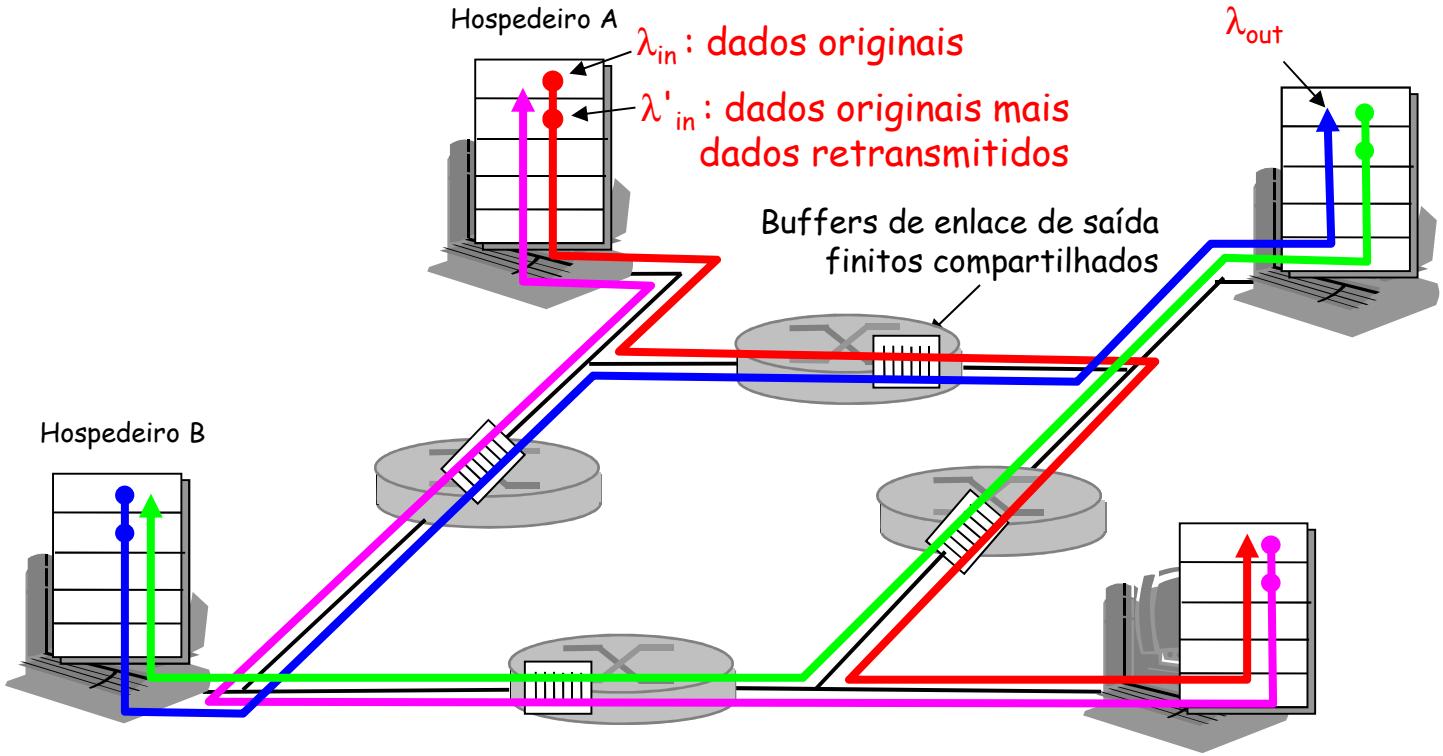
- Quatro remetentes
- Caminhos com múltiplos enlaces
- Temporização/retransmissão

O que acontece à medida que λ_{in} e λ'_{in} crescem?

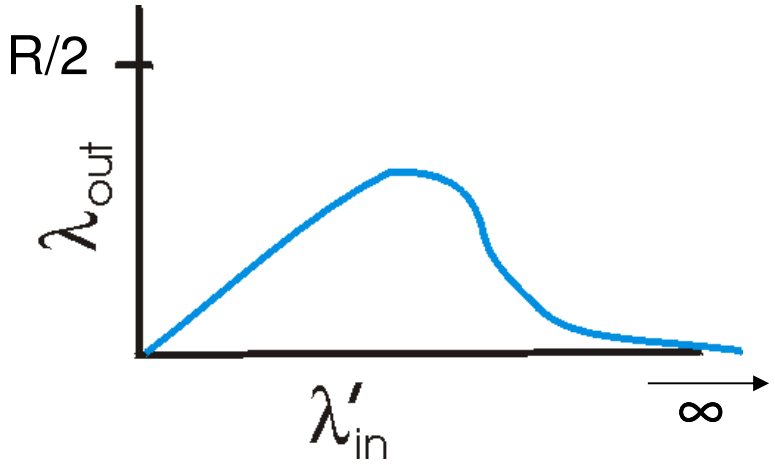


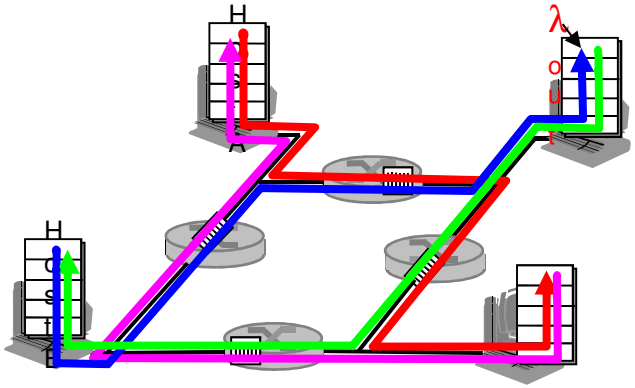
Congestionamento: Quatro Remetentes

Caso o primeiro roteador esteja ocupado com os pacotes de um dado hospedeiro, caso o próximo roteador não esteja disponível, todo o trabalho é desperdiçado...



Congestionamento: Quatro Remetentes

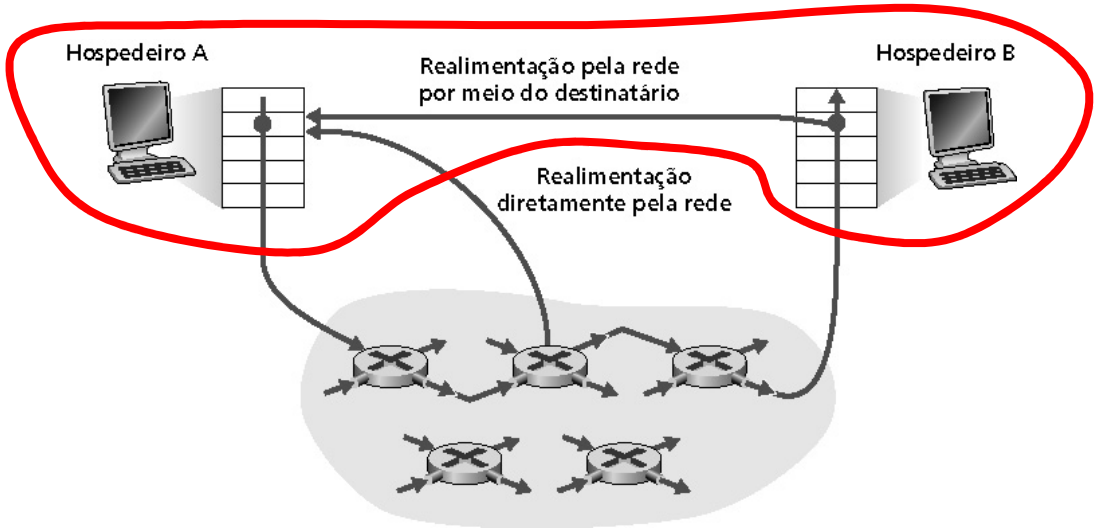




- Outro "custo" do congestionamento
  - Quando o pacote é descartado, qq. capacidade de transmissão já usada (antes do descarte) para esse pacote foi desperdiçada

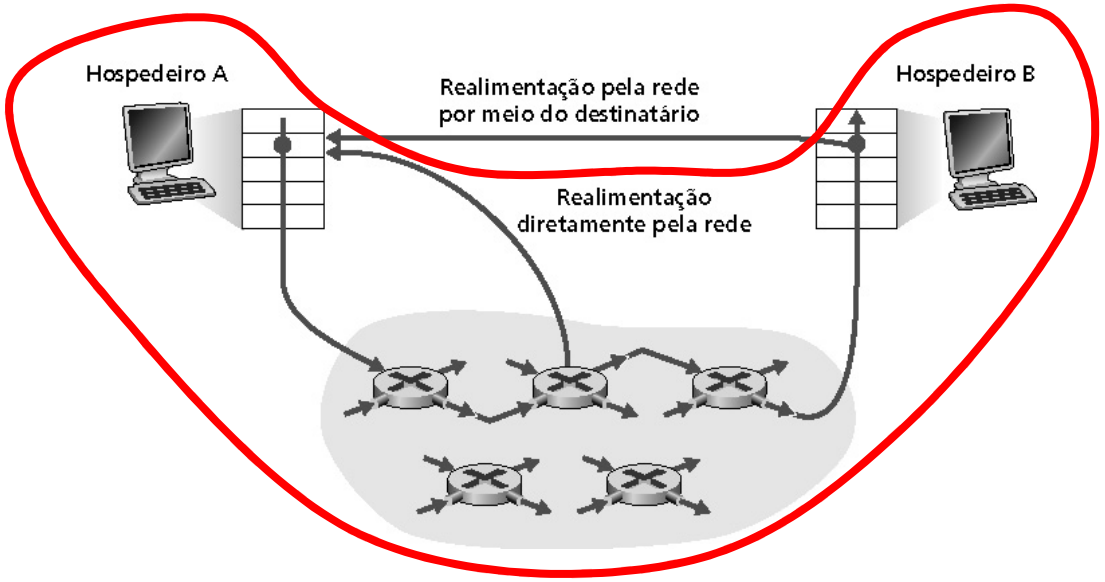
Controle de Congestionamento

- Pode ser:
  - Fim-a-fim:
    - Não usa realimentação explícita da rede
    - Congestionamento é inferido a partir das perdas e dos atrasos observados nos sistemas finais
    - **Abordagem usada pelo TCP**



Controle de Congestionamento

- Pode ser:
  - Assistido pela rede
    - Roteadores enviam informações para os sistemas finais
    - Bit indicando congestionamento (SNA, DECbit, TCP/IP ECN, ATM)
    - Taxa explícita para envio pelo transmissor



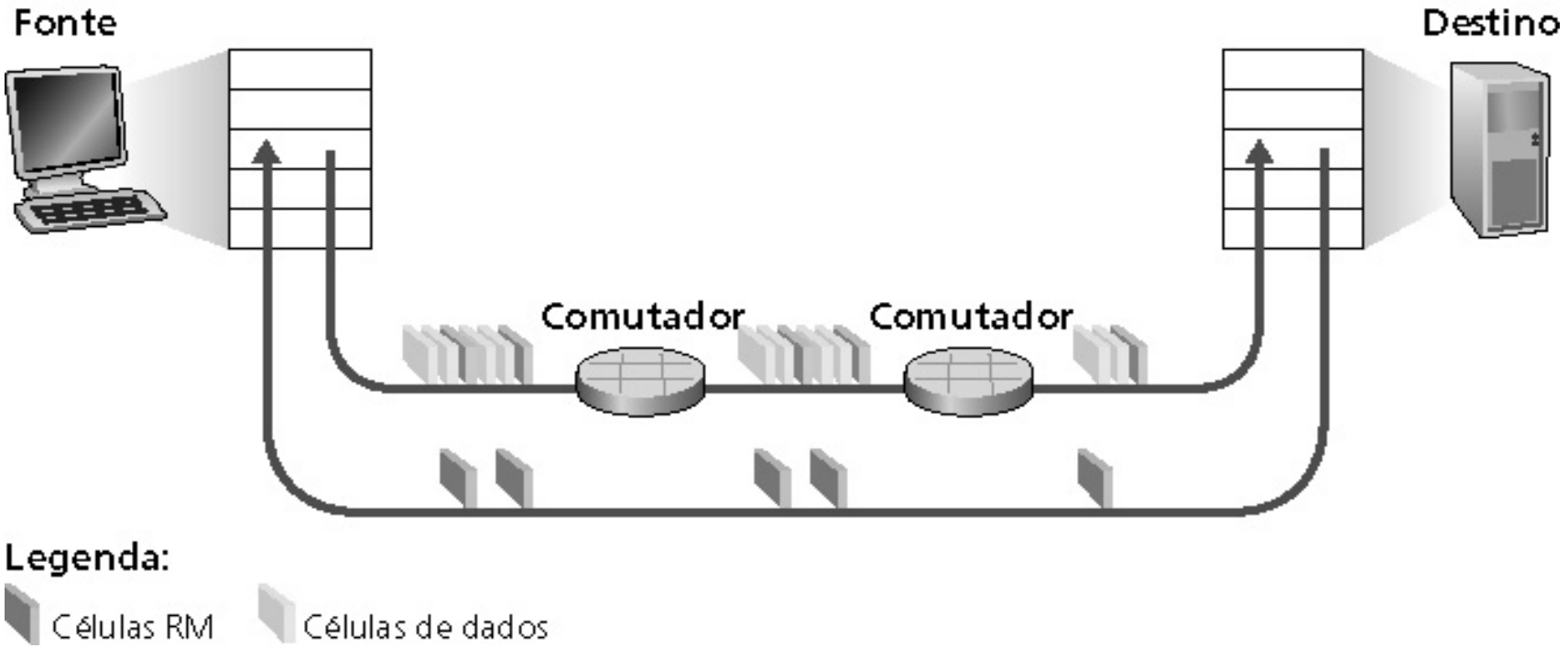
Controle de Congestionamento

- Serviço ATM ABR (Asynchronous Transfer Mode - Available Bit Rate)
  - "Serviço elástico"
  - Se caminho do transmissor está pouco usado
    - Transmissor pode usar banda disponível
  - Se caminho do transmissor estiver congestionado
    - Transmissor limitado à taxa mínima garantida

Controle de Congestionamento

- Serviço ATM ABR (Available Bit Rate)
  - Células RM (Resource Management)
    - Enviadas pelo transmissor, entremeadas com células de dados
    - Bits na célula RM iniciados por comutadores ("assistido pela rede")
      - Bit NI: não aumente a taxa (congestionamento moderado)
      - Bit CI: indicação de congestionamento
      - Nesse caso, as células RM que chegam no receptor são devolvidas ao transmissor sem alteração dos bits

Controle de Congestionamento



Controle de Congestionamento

- Campo ER (*explicit rate*) de 2 bytes nas **células RM**
  - Comutador congestionado pode reduzir valor de ER nas células
  - Taxa do transmissor ajustada para o menor valor possível entre os comutadores do caminho
- Bit EFCI em **células de dados** ligado pelos comutadores congestionados
  - Se EFCI ligado em células de dados que precedem a célula RM
    - Receptor liga bit CI na célula RM devolvida ao transmissor

Controle de Congestionamento do TCP

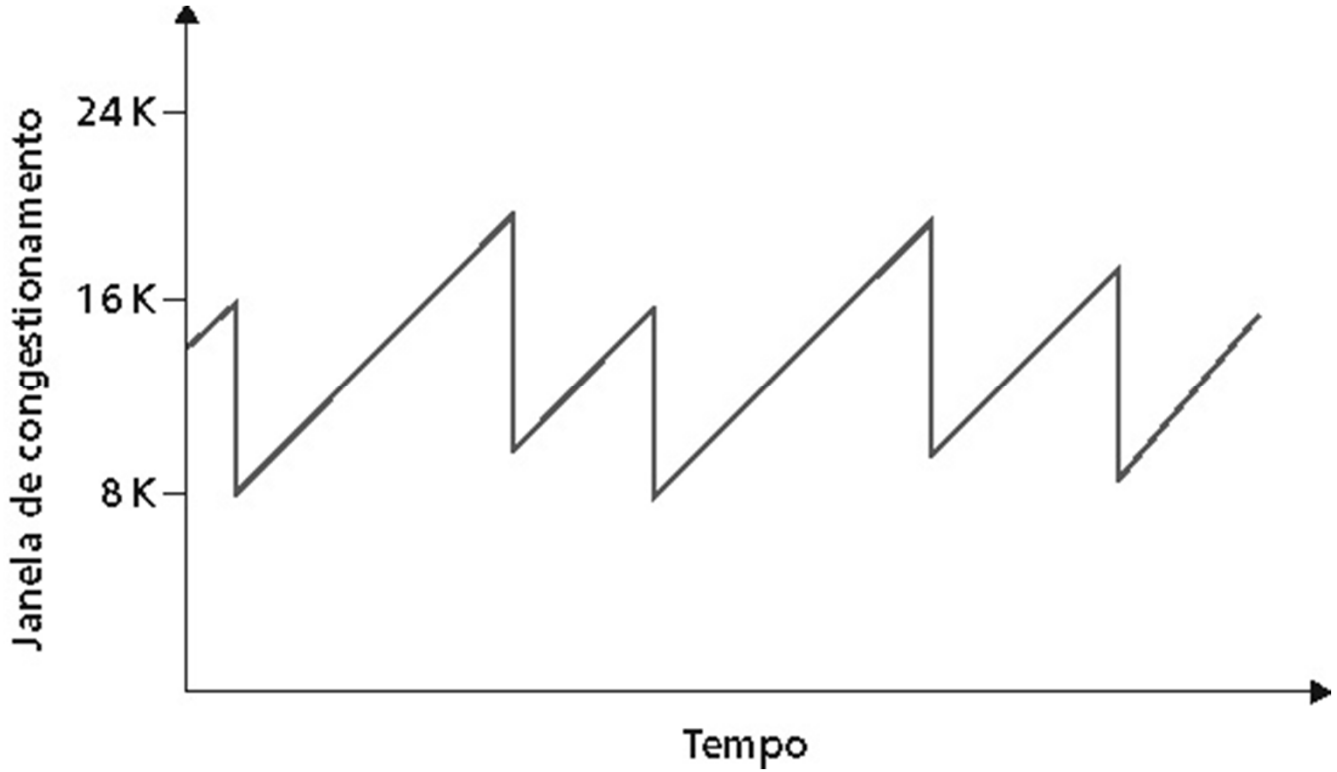
- Ideia
  - Aumentar a taxa de transmissão (tamanho da janela) até que ocorra uma perda
    - Largura de banda utilizável é testada

- Aumento aditivo
  - Incrementa a janela de congestionamento (`CongWin`) de 1 MSS a cada RTT até detectar uma perda
- Diminuição multiplicativa
  - Reduz a janela de congestionamento (`CongWin`) pela metade após evento de perda

Controle de Congestionamento do TCP

Comportamento de dente de serra

"Testando" a largura de banda



Controle de Congestionamento do TCP

- Transmissor limita a transmissão

$$\texttt{LastByteSent-LastByteAcked} \leq \texttt{CongWin}$$

$$\texttt{taxa} = \frac{\texttt{CongWin}}{\texttt{RTT}} \text{ bytes/s}$$

- `CongWin` é dinâmica, em função do congestionamento detectado da rede

Controle de Congestionamento do TCP

- Como o transmissor detecta o congestionamento?
  - Evento de perda
    - **Estouro do temporizador ou 3 ACKs duplicados**
  - Transmissor reduz a taxa (`CongWin`) após evento de perda

- Algoritmo composto de três etapas
  - Partida lenta
  - Prevenção de congestionamento
  - Recuperação rápida

Partida Lenta do TCP

- No início da conexão: `CongWin` = 1 MSS
  - Exemplo: MSS = 500 bytes = 4000 bits e RTT = 200 ms
  - Taxa inicial = `CongWin/RTT` = 20 kb/s

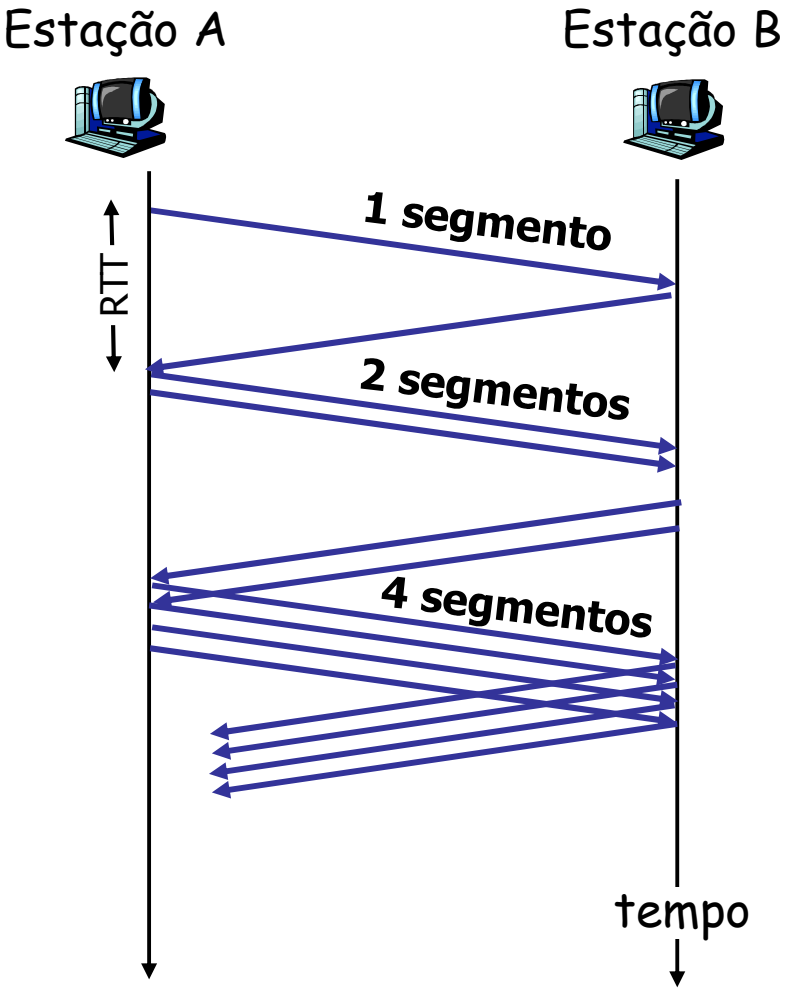
- Largura de banda disponível pode ser muito maior do que MSS/RTT
  - É desejável um crescimento rápido até uma taxa considerável

No início da conexão, a taxa aumenta exponencialmente até o primeiro evento de perda

Partida Lenta do TCP

- Duplica `CongWin` a cada RTT
- Através do incremento da `CongWin` para cada ACK recebido

taxa inicial é baixa, mas cresce rapidamente de forma exponencial



Término da Partida Lenta

- Após estouro de temporizador
  - `CongWin` é reduzida a 1 MSS
  - Reinicia processo de partida lenta até o limiar (`ssthresh=CongWin/2`) e depois cresce linearmente
  - Retransmite os segmentos perdidos

- Ou ainda, ao chegar no limiar `ssthresh=CongWin/2`
  - Caso já tenha havido um estouro de temporizador
  - A janela cresce conforme o modo de **prevenção de congestionamento**

Término da Partida Lenta

- Após 3 ACKs duplicados
  - Realiza uma retransmissão rápida
    - Antes do estouro do temporizador
  - Ajusta `ssthresh=CongWin/2` e `CongWin = ssthresh + 3`, relativo aos 3 ACKs duplicados
  - A janela cresce conforme modo de **recuperação rápida**

Término da Partida Lenta

- Após 3 ACKs duplicados
  - Realiza uma retransmissão rápida
    - Antes do estouro do temporizador
  - Janela cresce conforme modo de **recuperação rápida**

Estouro de temporizador antes de 3 ACKs duplicados é mais "alarmante" e por isso é que tem a consequência mais drástica

Prevenção de Congestionamento

- Ao chegar no limiar `ssthresh=CongWin/2`
  - TCP deixa de duplicar a sua janela a cada RTT e adota uma abordagem mais conservadora
    - Janela é incrementada de um MSS a cada RTT
- Quando o modo de prevenção de congestionamento deve parar?
  - Caso haja estouro de temporizador
    - Volta ao estado de **partida lenta**
      - `ssthresh=CongWin/2` **e** `CongWin = 1`
    - Retransmite os segmentos perdidos

Prevenção de Congestionamento

- Ao chegar no limiar `ssthresh=CongWin/2`
  - TCP deixa de duplicar a sua janela a cada RTT e adota uma abordagem mais conservadora
    - Janela é incrementada de um MSS a cada RTT

- Quando o modo de prevenção de congestionamento deve parar?
  - Caso haja 3 ACKs duplicados
    - Retransmite os 3 segmentos perdidos
    - Ajusta o limiar novamente para `ssthresh=CongWin/2` e `CongWin=ssthresh+3`
      - Redução da `CongWin` depende da versão do TCP
    - Assim como na partida lenta, a transmissão entra no modo de **recuperação rápida**

Recuperação Rápida

- Não é utilizada por todas as versões do TCP

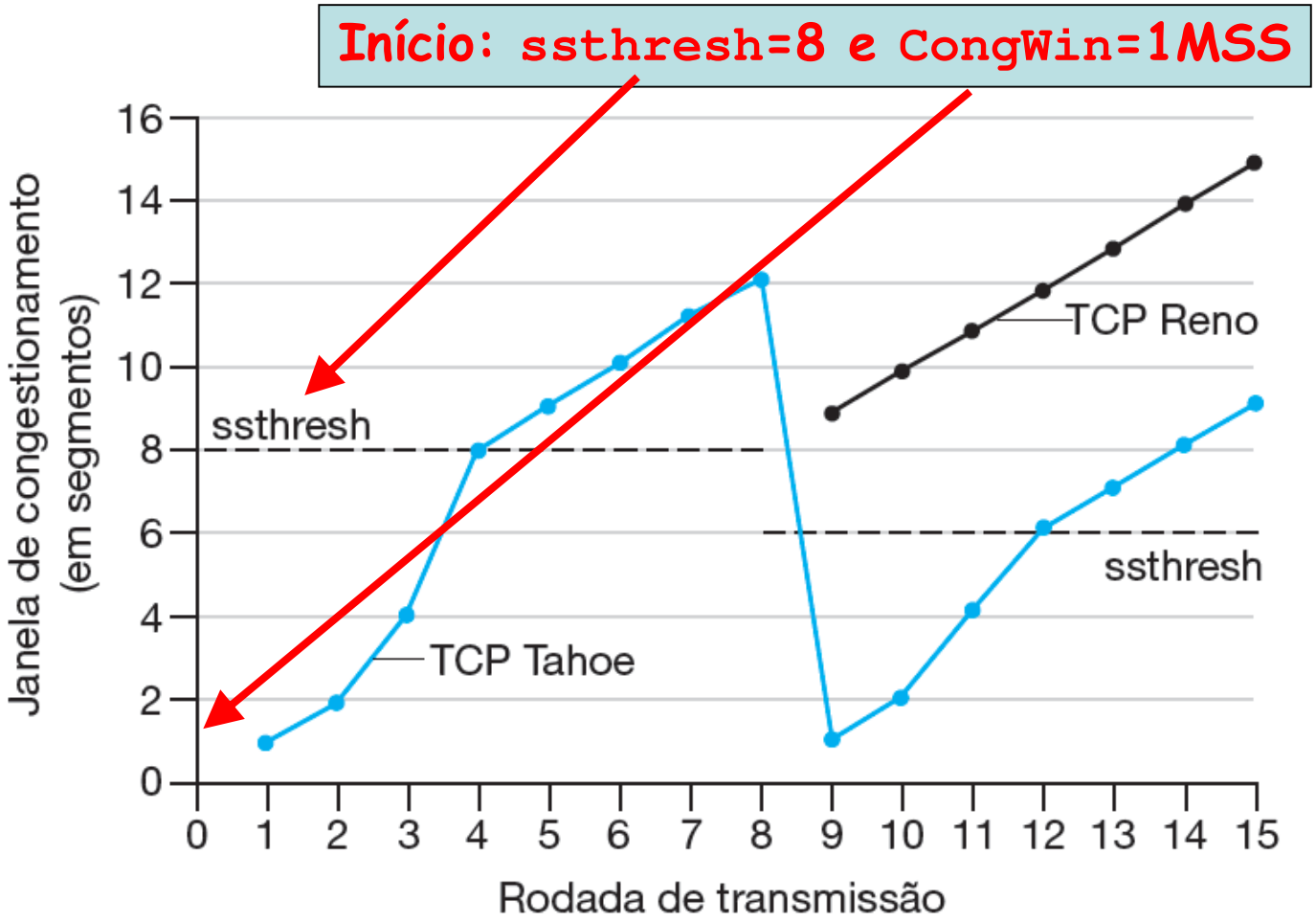
- Redução da CongWin (de acordo com a versão do TCP)
- Janela é aumentada de 1 MSS para cada ACK duplicado recebido
  - Mesmos ACKs que provocaram a entrada no modo de recuperação rápida

Recuperação Rápida

- Quando um ACK chega para o segmento perdido (possivelmente um ACK cumulativo)
  - Transmissão entra em modo de **prevenção de congestionamento**
    - `CongWin = ssthresh`

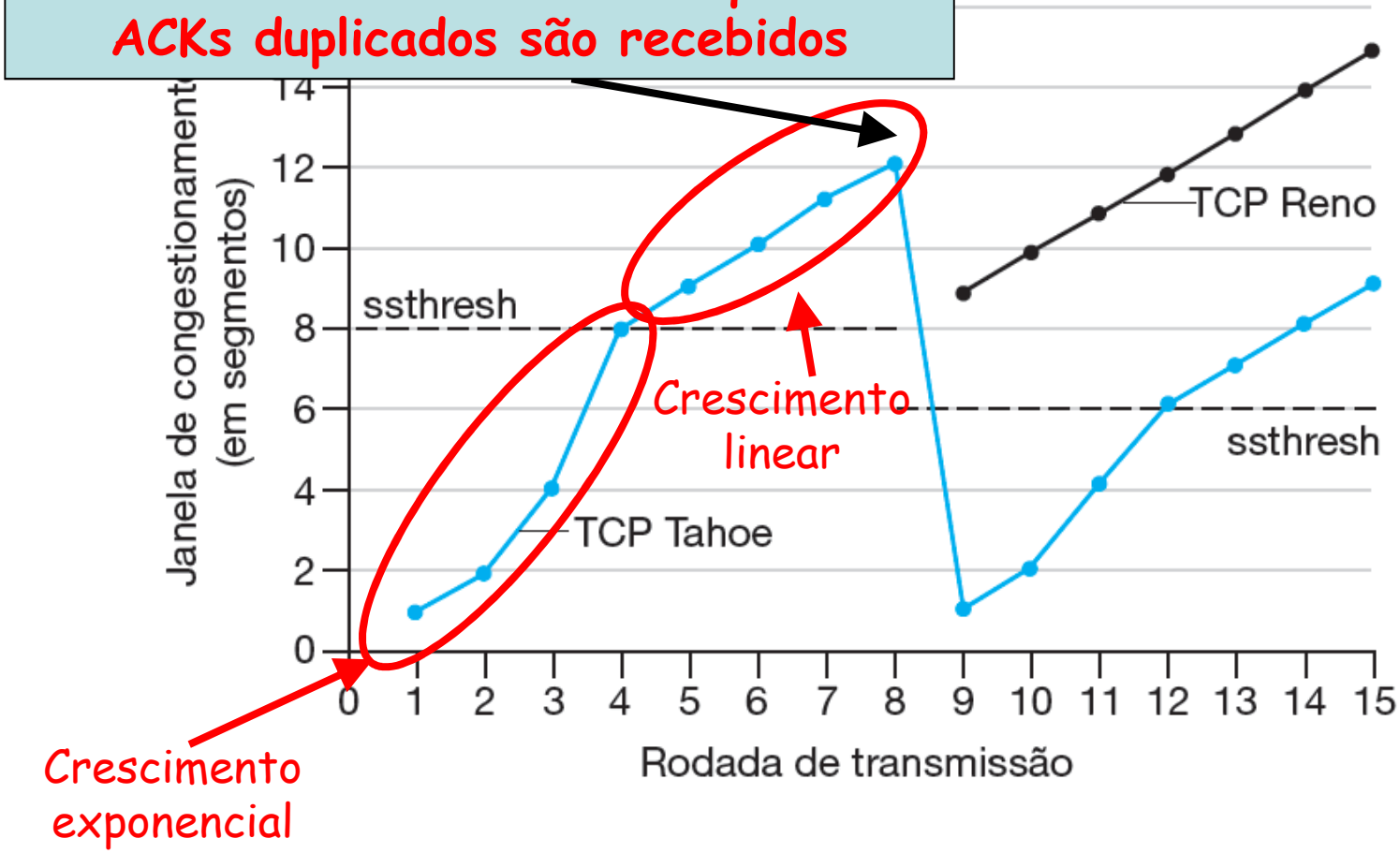
- Se um estouro de temporizador ocorrer
  - Volta ao modo de partida lenta...

Controle de Congestionamento Do TCP: Tahoe e Reno

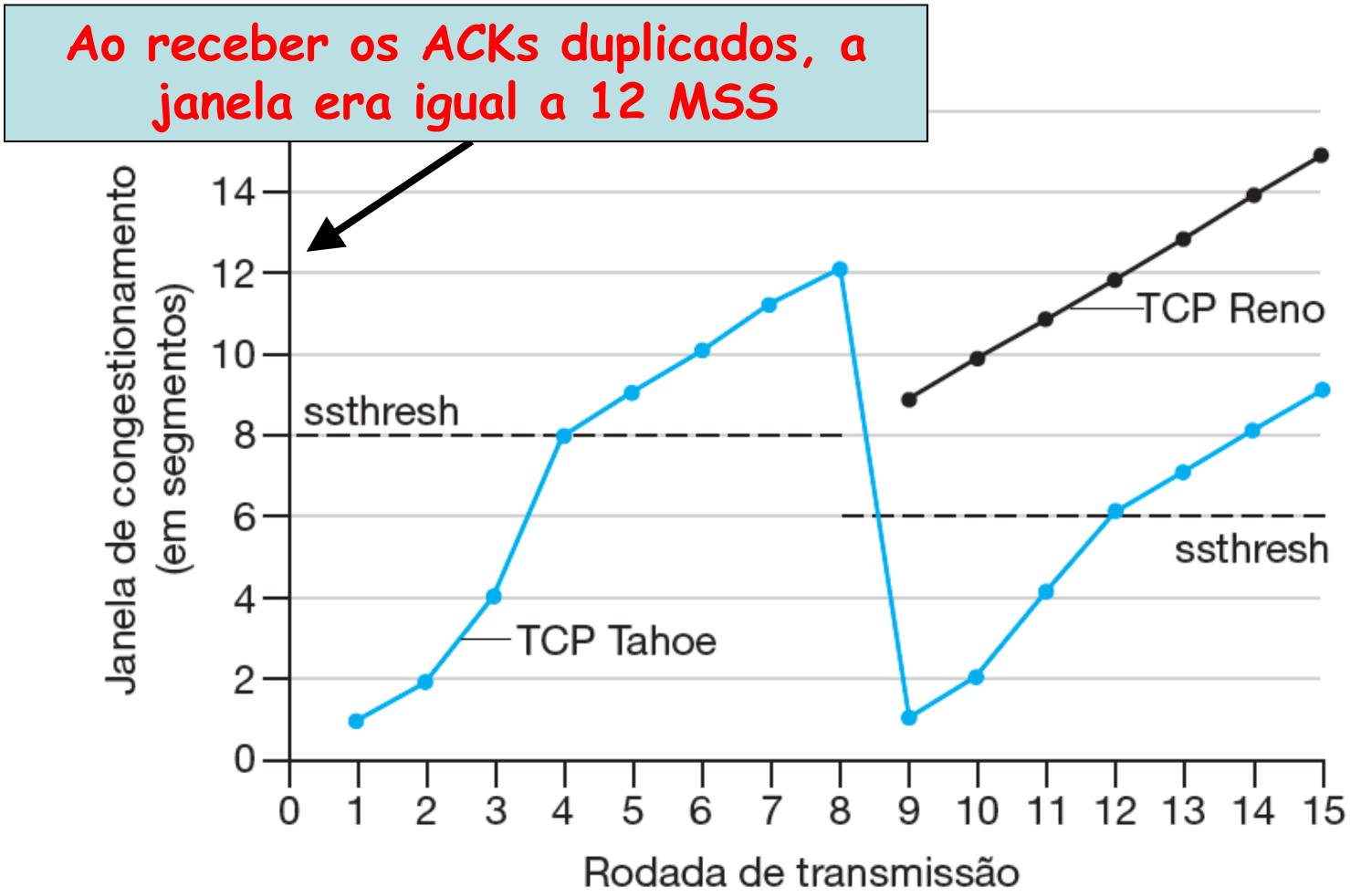


Controle de Congestionamento Do TCP: Tahoe e Reno

Desempenho idêntico do Tahoe e do Reno até a oitava rodada quando ACKs duplicados são recebidos

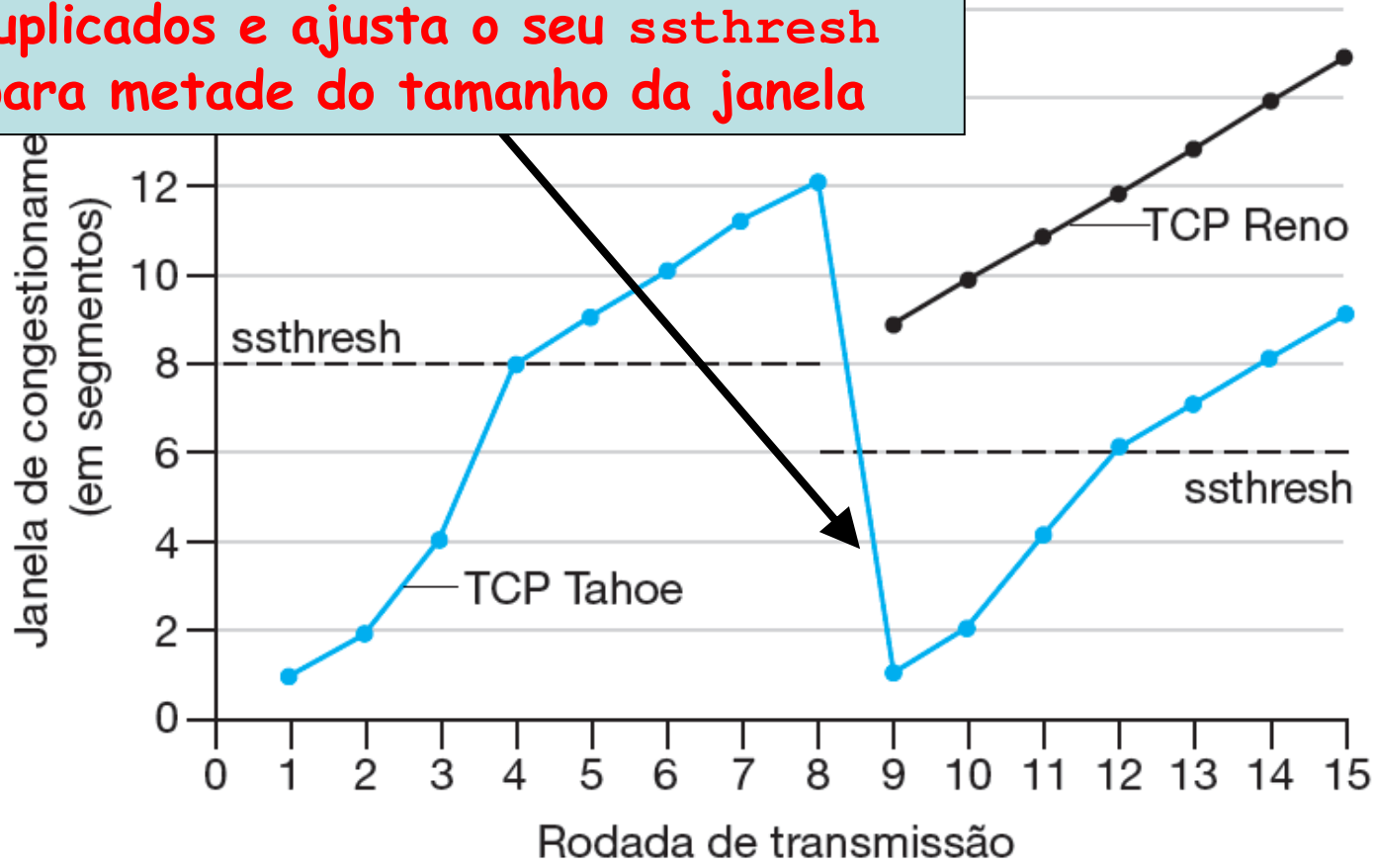


Controle de Congestionamento Do TCP: Tahoe e Reno



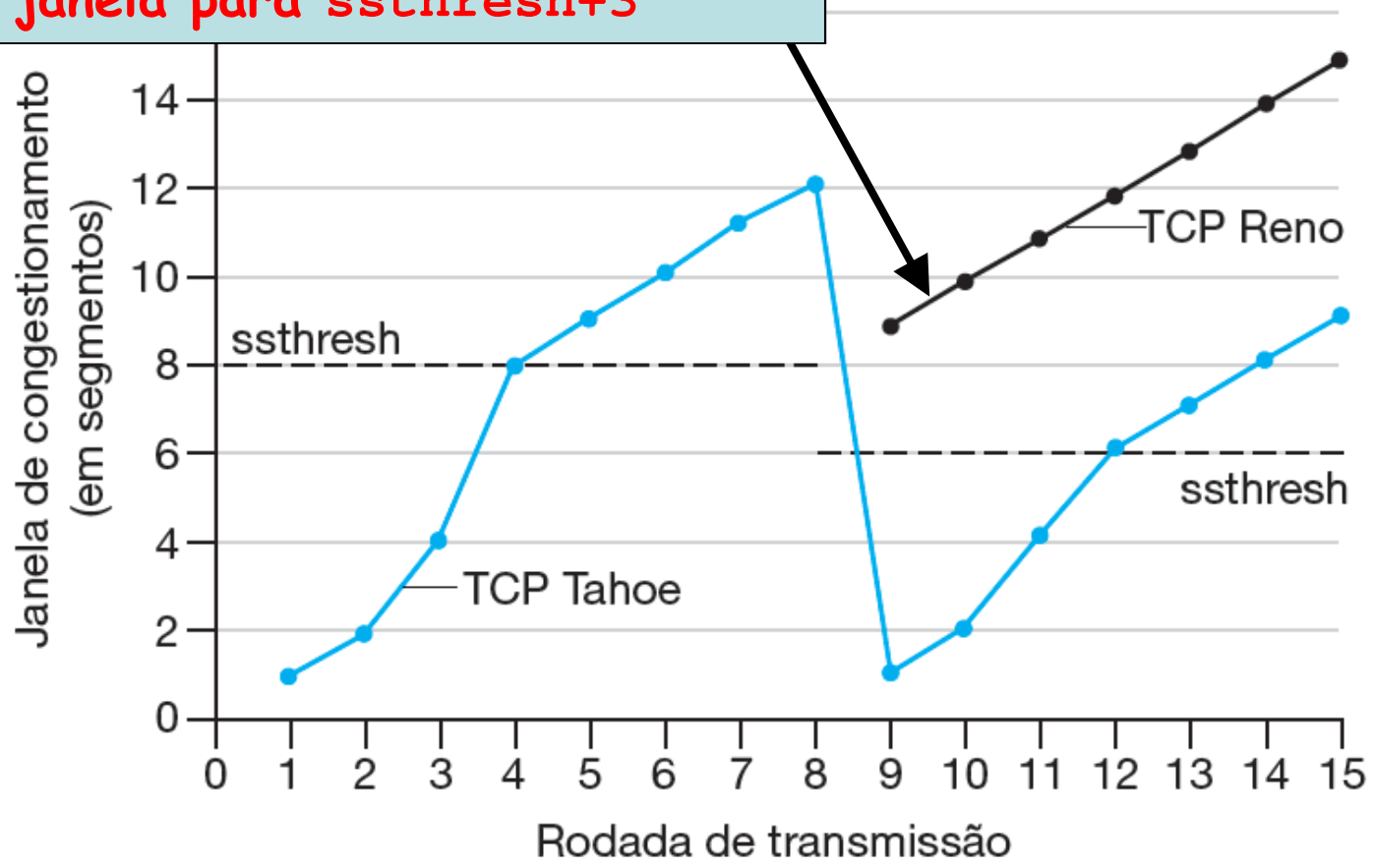
Controle de Congestionamento Do TCP: Tahoe e Reno

O TCP Tahoe reduz a janela para 1 MSS mesmo como consequência de ACKs duplicados e ajusta o seu `ssthresh` para metade do tamanho da janela



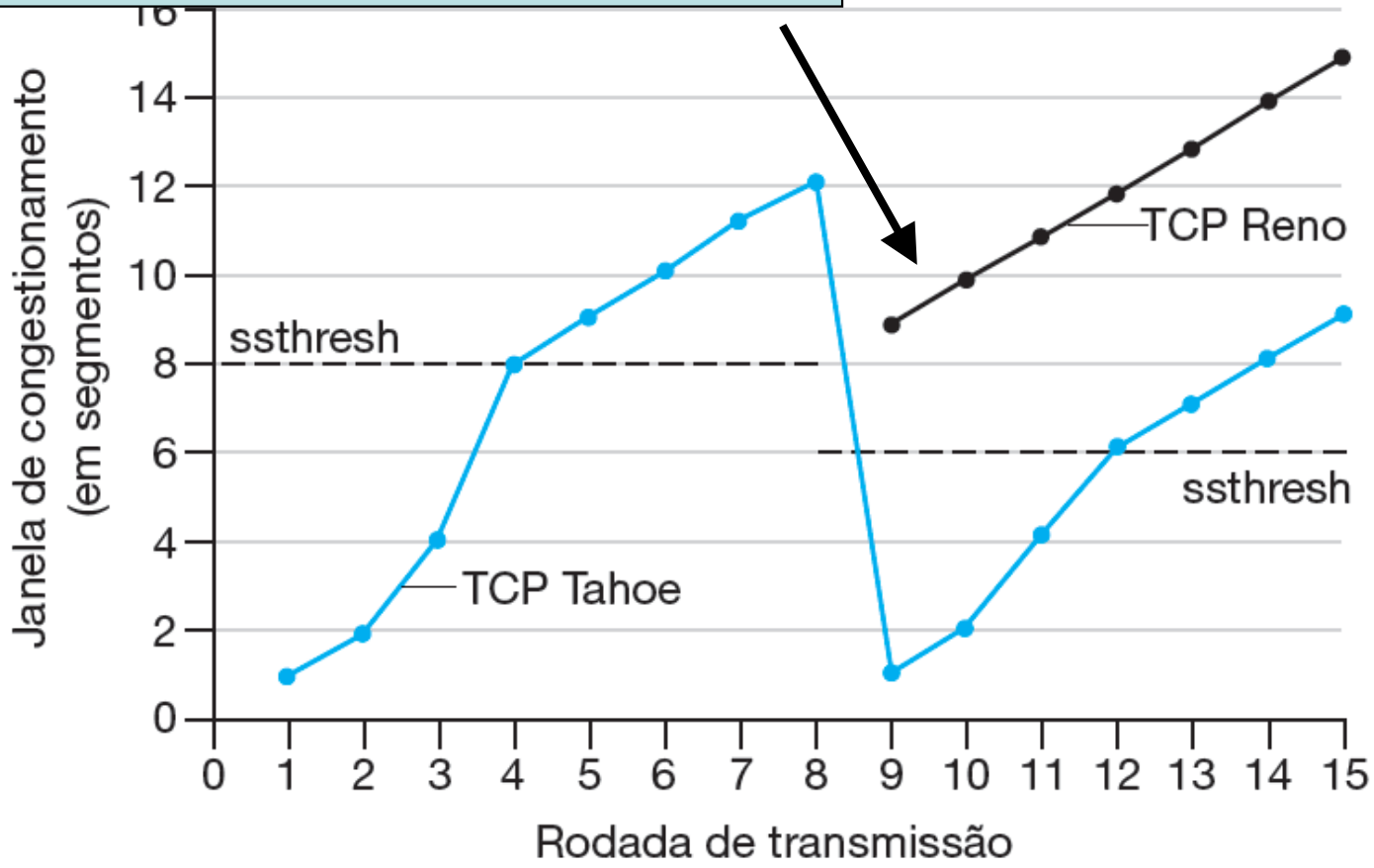
Controle de Congestionamento Do TCP: Tahoe e Reno

O TCP Reno ajusta o `ssthresh` para metade do tamanho da janela e a janela para `ssthresh+3`



Controle de Congestionamento Do TCP: Tahoe e Reno

O TCP Reno inclui recuperação rápida enquanto o Tahoe não usa



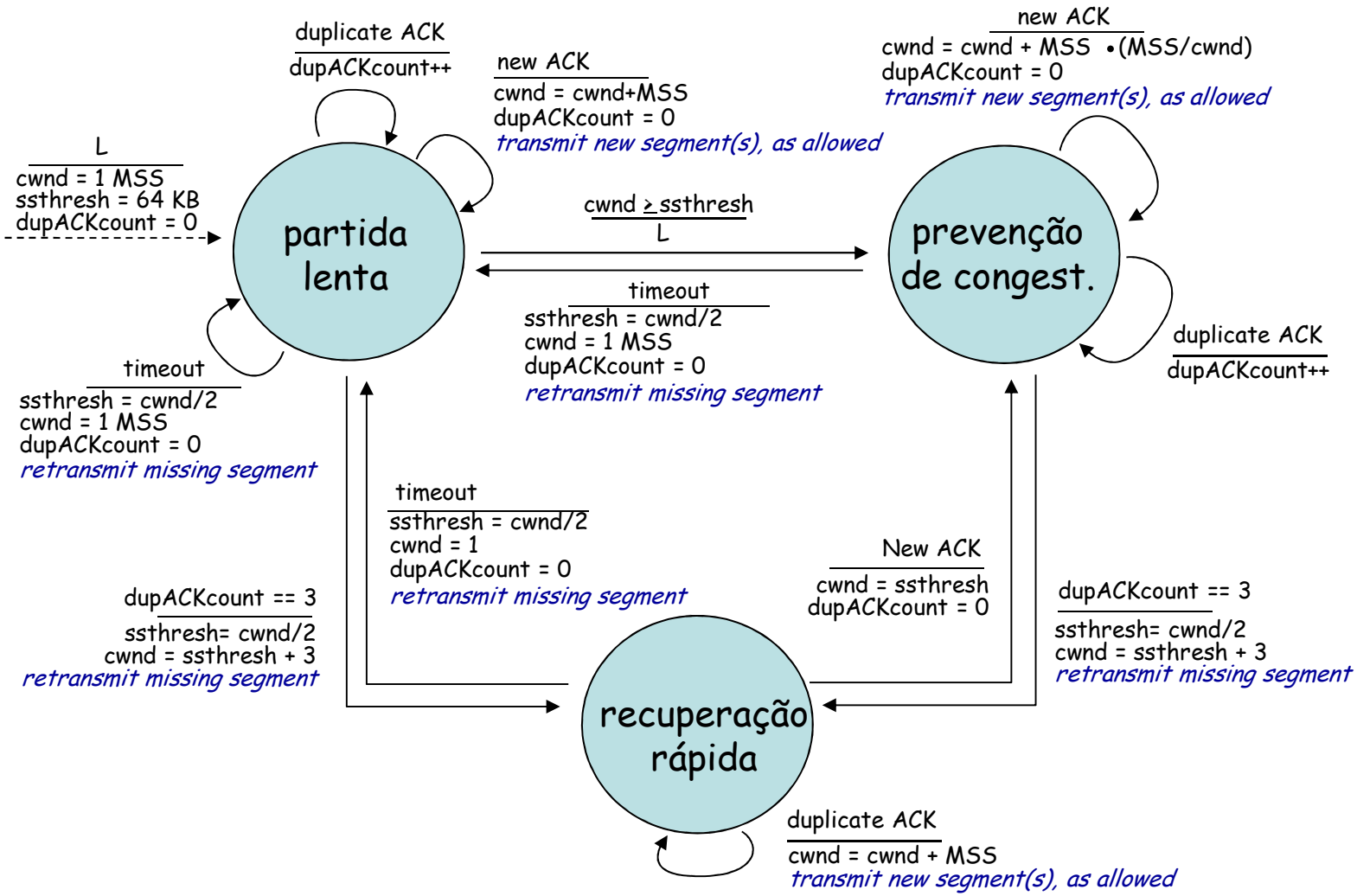
Controle de Congestionamento do TCP Reno

- Quando a `CongWin` está abaixo do limiar (`ssthresh`)
  - Transmissor está na fase de partida lenta
  - Janela cresce exponencialmente
- Quando a `CongWin` está acima do limiar (`ssthresh`)
  - Transmissor está na fase de prevenção de congestionamento
  - Janela cresce linearmente
- Quando chegam três *ACKs* duplicados
  - `ssthresh` passa a ser `CongWin`/2 e `CongWin=ssthresh + 3`
- Quando estoura o temporizador
  - `ssthresh` é ajustado para `CongWin`/2 e `CongWin`=1 MSS

Controle de Congestionamento do TCP

Evento	Estado	Ação do Transmissor TCP	Comentário
ACK recebido para dados ainda não reconhecidos	Partida lenta	CongWin = CongWin + MSS, Se (CongWin > ssthresh) ajustar estado para "Prevenção de congestionamento"	Resulta na duplicação da CongWin a cada RTT
ACK recebido para dados ainda não reconhecidos	Prevenção de congestio-namento	CongWin = CongWin + 1	Aumento aditivo, resultando no incremento da CongWin de 1 MSS a cada RTT
Perda detectada por três ACKs duplicados	qualquer	ssthresh = CongWin/2, CongWin = ssthresh + 3, Ajusta estado para "Prevenção de Congestionamento"	Recuperação rápida, implementa diminuição multiplicativa. CongWin não cai abaixo de 1 MSS.
Estouro de temporizador	qualquer	ssthresh = CongWin/2, CongWin = 1 MSS, Ajusta estado para "Partida lenta"	Entra estado de "partida lenta"
ACK duplicado	qualquer	Incrementa contador de ACKs duplicados para o segmento que está sendo reconhecido	Depende da implementação do TCP (CongWin = CongWin + 3 e ssthresh = CongWin/2)

Resumo: Controle de Congestionamento do TCP

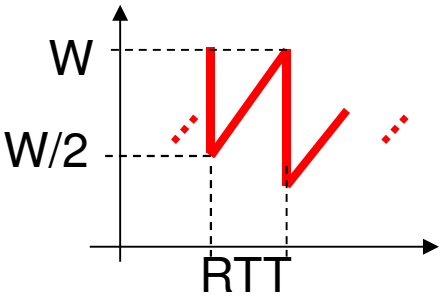


Fonte: Livro do Kurose e Ross

Vazão do TCP

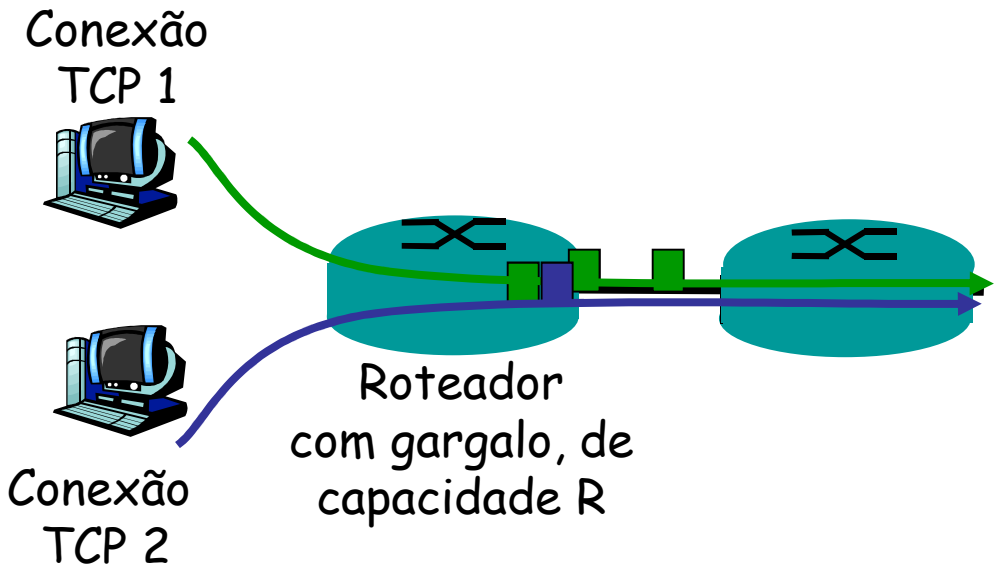
- Qual é a vazão média do TCP em função do tamanho da janela e do RTT?
  - Ignore a partida lenta
    - Aumento aditivo, diminuição multiplicativa (AIMD)

- Seja W o tamanho da janela quando ocorre a perda:
  - Quando a janela é W a vazão é: W/RTT
  - Logo após a perda, a janela cai para W/2
    - Nesse caso, a vazão cai para W/(2*RTT)
  - Vazão média = 0,75*W/RTT
    - Área do trapézio de altura RTT e largura W/2 e W



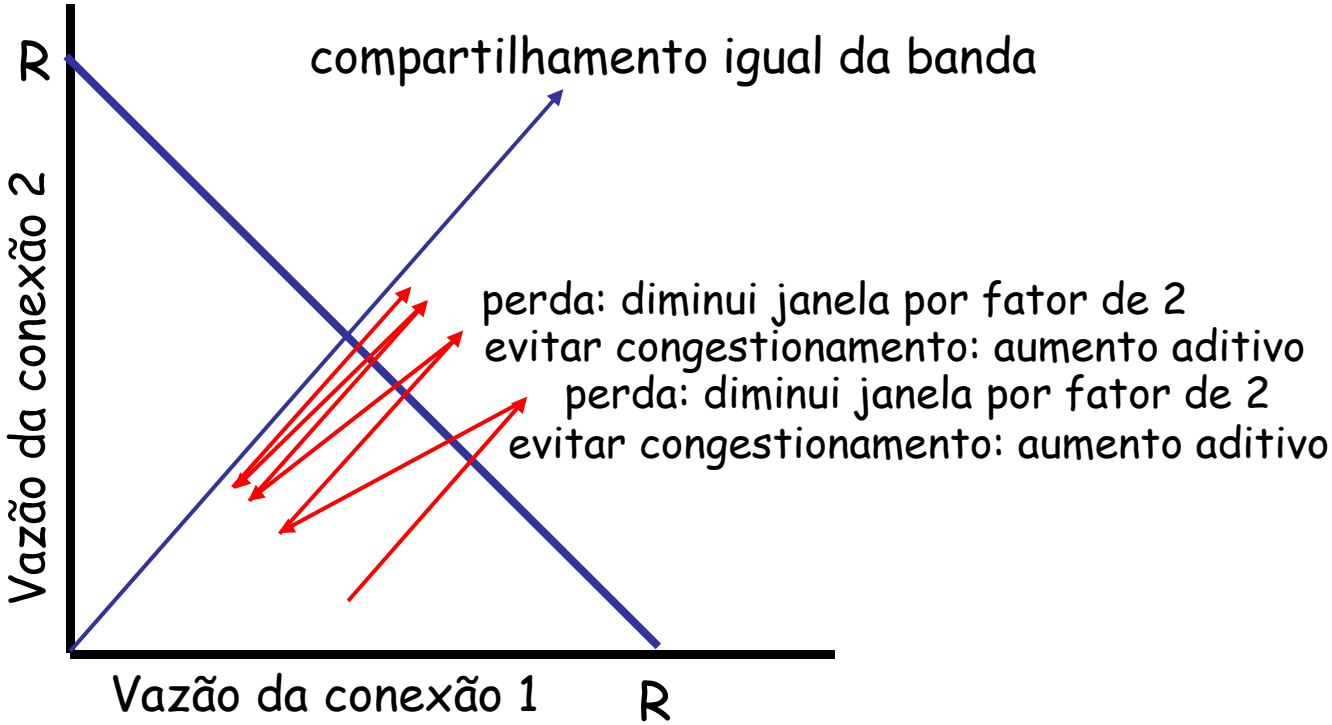
Equidade (Fairness) do TCP

- Se K sessões TCP compartilham o mesmo enlace de gargalo com largura de banda R
  - Cada uma deve obter uma taxa média de R/K



Justiça do TCP

- Duas sessões competindo pela banda:
  - Aumento aditivo dá gradiente de 1, enquanto vazão aumenta
  - Redução multiplicativa diminui vazão proporcionalmente



Equidade do TCP X UDP

- Aplicações multimídia frequentemente não usam TCP
  - Não querem a taxa estrangulada pelo controle de congestionamento
  - Preferem usar o UDP
    - Injeta áudio/vídeo a taxas constantes, toleram perdas de pacotes

- Aplicações multimídia devem-se tornar amigáveis ao TCP (TCP friendly)

Justiça X Conexões em Paralelo

- Aplicações podem abrir conexões paralelas entre dois sistemas finais
  - Os navegadores Web fazem isso

- Exemplo:
  - Dado um canal com taxa R compartilhado por 9 conexões:
    - Caso **uma nova aplicação** surja estabelecendo **uma** conexão TCP → **ela obterá uma taxa de R/10**
    - Caso **uma nova aplicação** surja estabelecendo **11** conexões TCP em paralelo → **ela obterá uma taxa de 11*R/20**

Implementações

- Tahoe
  - Original
- Reno
- Vegas
- SACK
- NewReno (RFC 2582)
  - Usado no Windows Vista
- Cubic
  - Usado pelo Debian (`/proc/sys/net/ipv4/tcp_congestion_control`)
- Etc.

Ambientes Desafiadores para o TCP

Redes de Alta Velocidade

- Exemplo
  - Segmentos de 1500 bytes e RTT=100 ms,
  - Vazão desejada de 10 Gb/s

$$\underbrace{\text{vazão*RTT}}_{\text{produto banda x latência}} = W \times MSS \text{ bytes}$$

  - Requer janela de W = (10Gb/s*100ms)/(1500*8) = **83.333 segmentos em trânsito!**

O que aconteceria se um deles fosse perdido?

Redes de Alta Velocidade

- Vazão em função da taxa de perdas:

$$\frac{1{,}22 \cdot MSS}{RTT\sqrt{L}}$$

- Para o exemplo:
  - $L = 2*10^{-10}$ → **taxa de perdas tem que ser muito baixa para "encher" o meio!**

Novas versões do TCP para alta velocidade

Redes de Alta Velocidade

- Novos protocolos de transporte para redes gigabit propostos
  - HSTCP (HighSpeed TCP)
  - XCP (eXplicit Control Protocol)
  - STCP (Scalable TCP)
  - FAST TCP (Fast Active-queue-management Scalable TCP)

Material Utilizado

- Notas de aula do Prof. Igor Monteiro Moraes, disponíveis em http://www2.ic.uff.br/~igor/cursos/redespg

Leitura Recomendada

- Capítulo 3 do Livro "*Computer Networking: A Top Down Approach*", 5a. Ed., Jim Kurose and Keith Ross, Pearson, 2010

- Capítulo 6 do Livro "*Computer Networks*", Andrew S. Tanenbaum e David J. Wetherall, 5a. Ed., Pearson, 2011

Leitura Recomendada

- S. Floyd, S. Ratnasamy e S. Shenker, "Modifying TCP's Congestion Control for High Speeds", draft, maio de 2002
- Dina Katabi, Mark Handley e Charlie Rohrs, "Congestion control for high bandwidth-delay product networks", em ACM Sigcomm, pp. 89-102, agosto de 2002
- H. Balakrishnan, S. Seshan, E. Amir e R. H. Katz, "Improving TCP/IP Performance over Wireless Networks", em ACM MobiCom, pp. 2-11, novembro de 1995