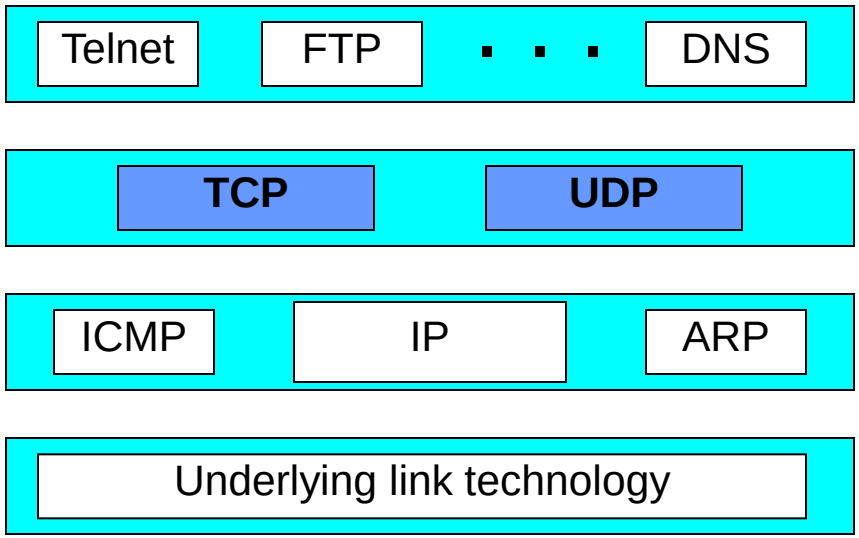
Universidade Federal de Santa Catarina
Departamento de Informática e de Estatística
Curso de Sistemas de Informação

Capítulo 5
Camada de Transporte

Prof. Roberto Willrich
INE - UFSC
willrich@inf.ufsc.br

Camada de TranposteTCP/IP

- 2 protocolos de transporte na pilha TCP/IP:
  - UDP – User Datagram Protocol
    - Serviço sem conexão não confiável
  - TCP – Transmission Control Protocol
    - Serviço confiável e orientado a conexão



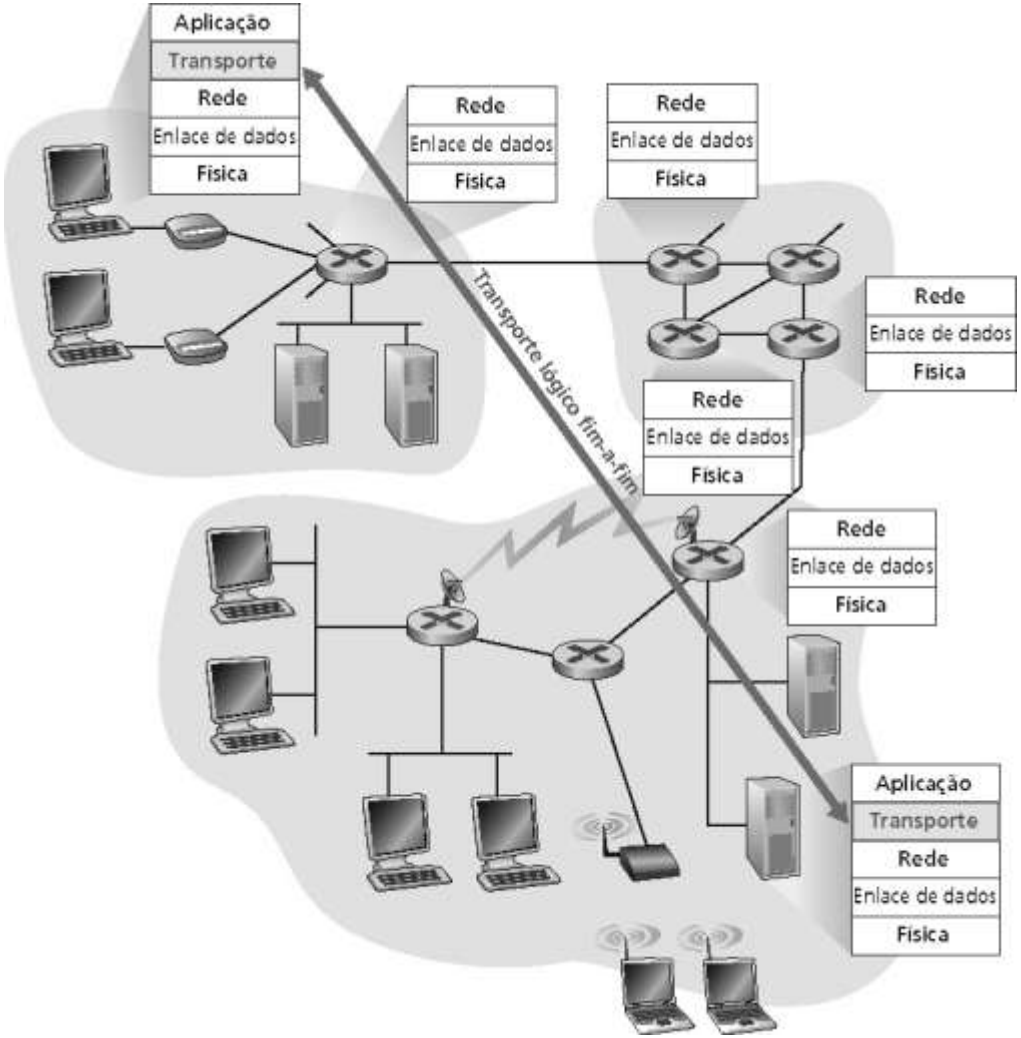
Capítulo 5: Camada de Transporte

Metas do capítulo:

- Entender os princípios atrás dos serviços da camada de transporte:
  - multiplexação/ demultiplexação
  - transferência confiável de dados
  - controle de fluxo
  - controle de congestionamento
- Aprender os protocolos de transporte da Internet:
  - transporte sem conexão: UDP
  - transporte orientado a conexão: TCP
    - transferência confiável
    - controle de fluxo e de congestionamento
    - gerenciamento de conexões

Serviços da camada de transporte

- Fornece comunicação lógica entre processos de aplicação rodando em hosts diferentes
- Protocolos de transporte rodam em sistemas finais
  - Lado emissor: quebra as mensagens da aplicação em segmentos e envia para a camada de rede
  - Lado receptor: remonta os segmentos em mensagens e passa para a camada de aplicação
- Há mais de um protocolo de transporte disponível para as aplicações
  - Internet: TCP e UDP



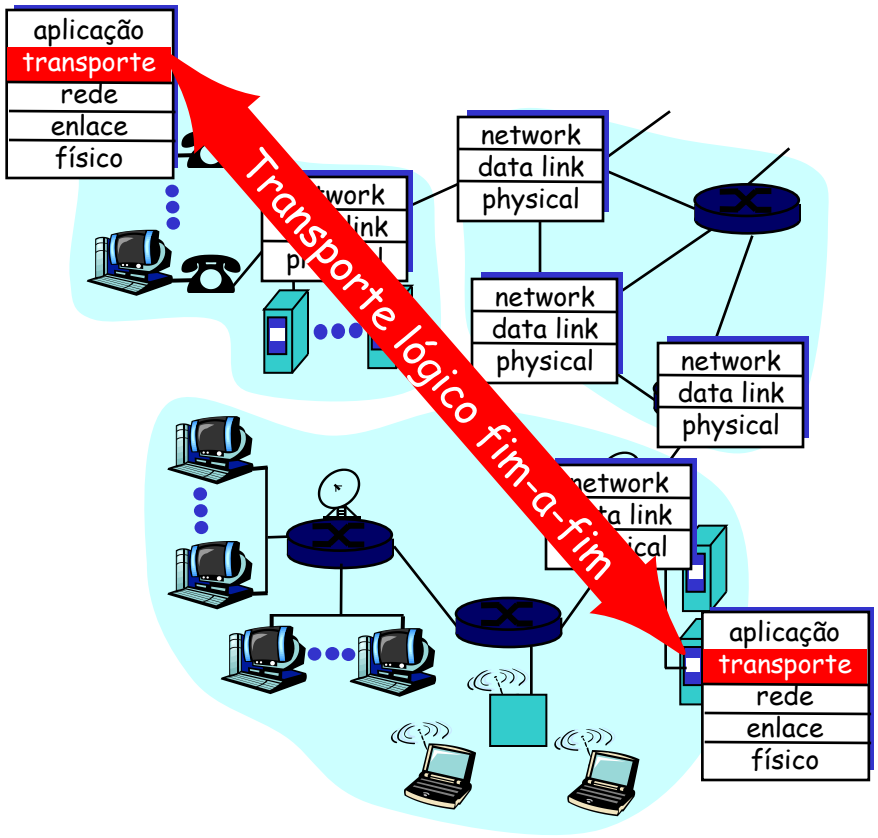
Camada de transporte vs. camada de rede

- Diferenças entre camada de transporte e de rede
  - Camada de rede: comunicação lógica entre os hospedeiros (computadores)
  - Camada de transporte: comunicação lógica entre os processos
    - Depende dos serviços da camada de rede

Protocolos da camada de transporte

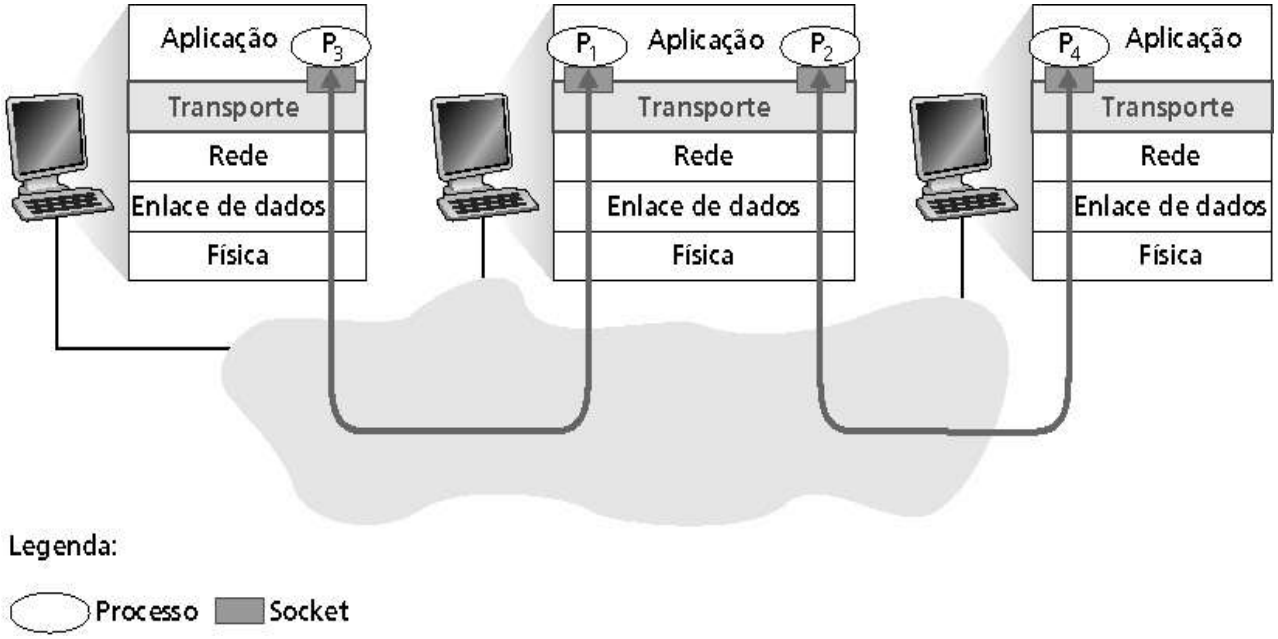
Serviços de transporte da Internet:

- TCP: Confiável, liberação em ordem unicast
  - Configuração de conexão
  - Sequenciamento
  - Controle de fluxo e de congestionamento
- UDP: Não confiável, liberação fora de ordem unicast ou multicast
- Serviços Não disponíveis:
  - Tempo-real
  - Garantias de largura de banda
  - Multicast confiável



Multiplexação/demultiplexação

- **Demultiplexação** no hospedeiro receptor:
  - entrega os segmentos recebidos ao socket correto
- **Multiplexação** no hospedeiro emissor:
  - coleta dados de múltiplos sockets, envelopa os dados com cabeçalho (usado depois para demultiplexação)



Portas de Protocolo

- Último fonte/destino de/para uma mensagem é uma porta de protocolo
- Um processo envia via/ouve uma porta de protocolo (identificado por um inteiro)
- Muitos sistemas operacionais fornecem acesso síncrono as portas
  - Um processo pode se bloquear aguardando chegada de mensagens na porta
- Em geral, portas são bufferizadas
  - Dados chegando antes da operação de leitura de um processo é colocada em uma fila (finita)
- Para se comunicar com uma porta, um emissor necessita conhecer o Endereço IP e a Porta do processo receptor
  - Combinação de endereço IP e porta é chamado de *socket*

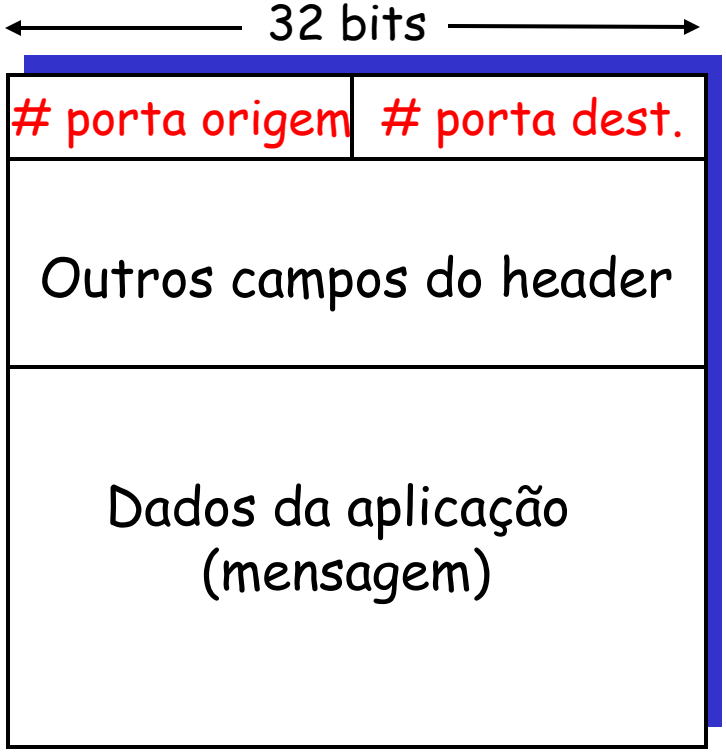
Número de portas em três grupos

Faixa	Propósito
1 .. 1023	Portas bem conhecidas são atribuídas pela Internet Assigned Numbers Authority (IANA)
1024 .. 49151	Portas registradas
49152 .. 65535	Portas dinâmicas

- Servidores são normalmente conhecidos por portas bem conhecidas (por exemplo, 80 para HTTP)
- Portas dinâmicas podem ser usados por qualquer processos (normalmente usados por processos clientes)

Multiplexação/Demultiplexação

- Baseado no número da porta do emissor e receptor, e endereços IP
  - Números de portas origem e destino em cada segmento (16 bits: 0..65535)
  - lembrete: número de portas bem conhecidas para aplicações específicas (0..1024)



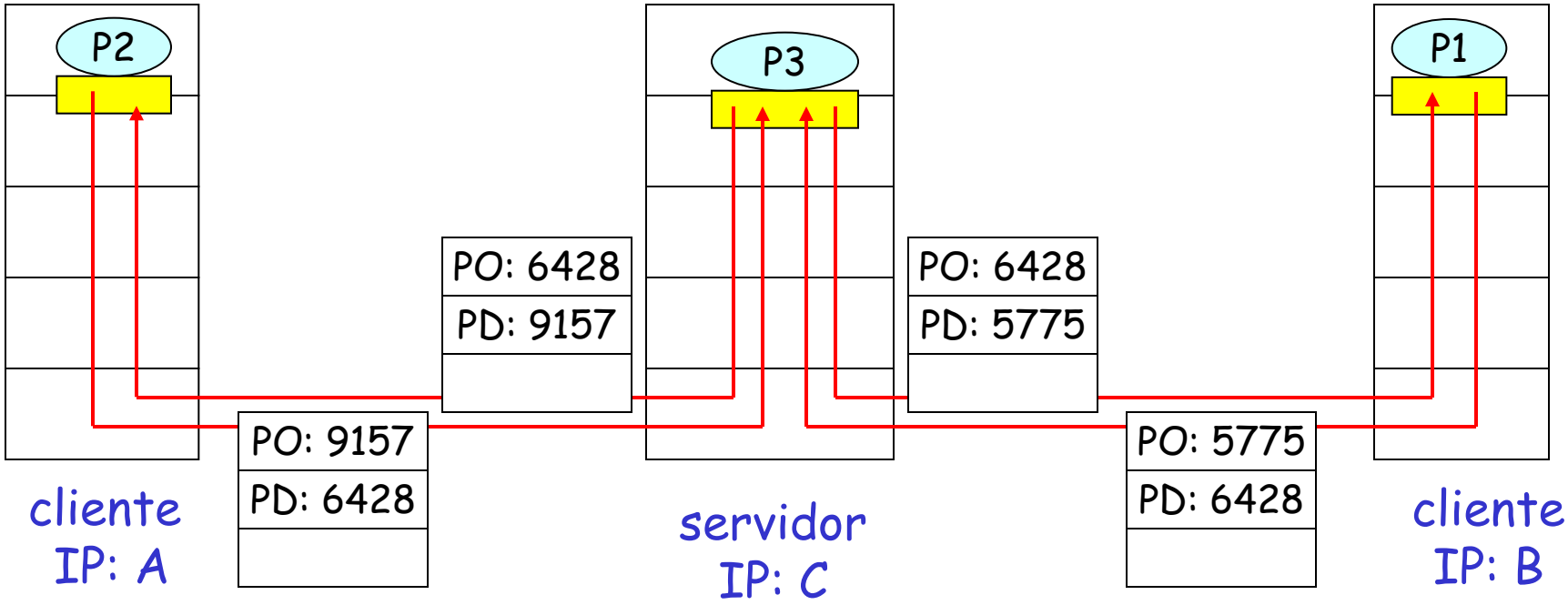
Formato do segmento TCP/UDP

Demultiplexação não orientada à conexão

- Cria sockets com números de porta:
  - `DatagramSocket mySocket1 = new DatagramSocket(99111);`
  - `DatagramSocket mySocket2 = new DatagramSocket(99222);`
- Socket UDP identificado pela porta que ele utiliza. No envio:
  - `DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length, IPAddress, 9876);`
  - `mySocket1.send(sendPacket);`
- O Pacote UDP recebido deve identificar:
  - (endereço IP de destino, número da porta de destino)
- Quando o hospedeiro recebe o segmento UDP:
  - Verifica o número da porta de destino no segmento
  - Direciona o segmento UDP para o socket com este número de porta

Demultiplexação não orientada à conexão

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

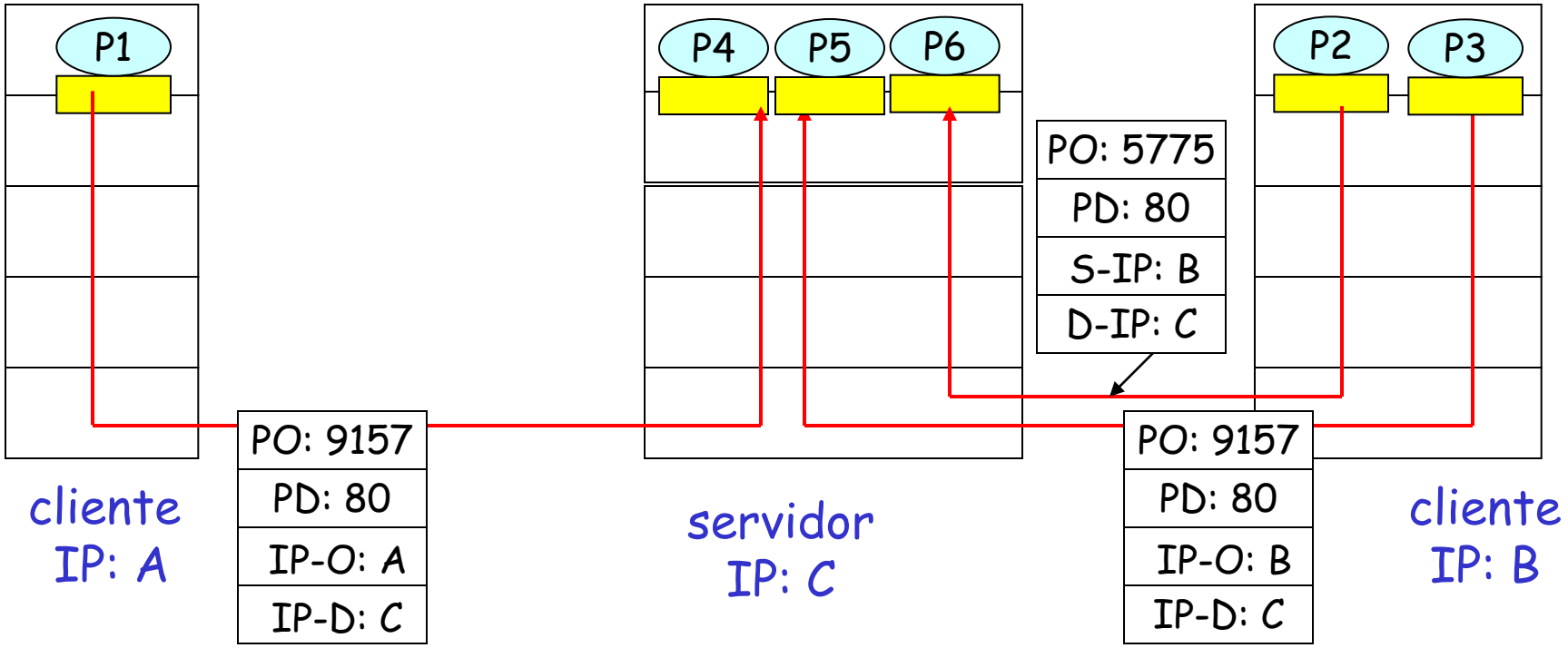


PO: Porta de Origem
PD: Porta de Destino

Demux orientada à conexão

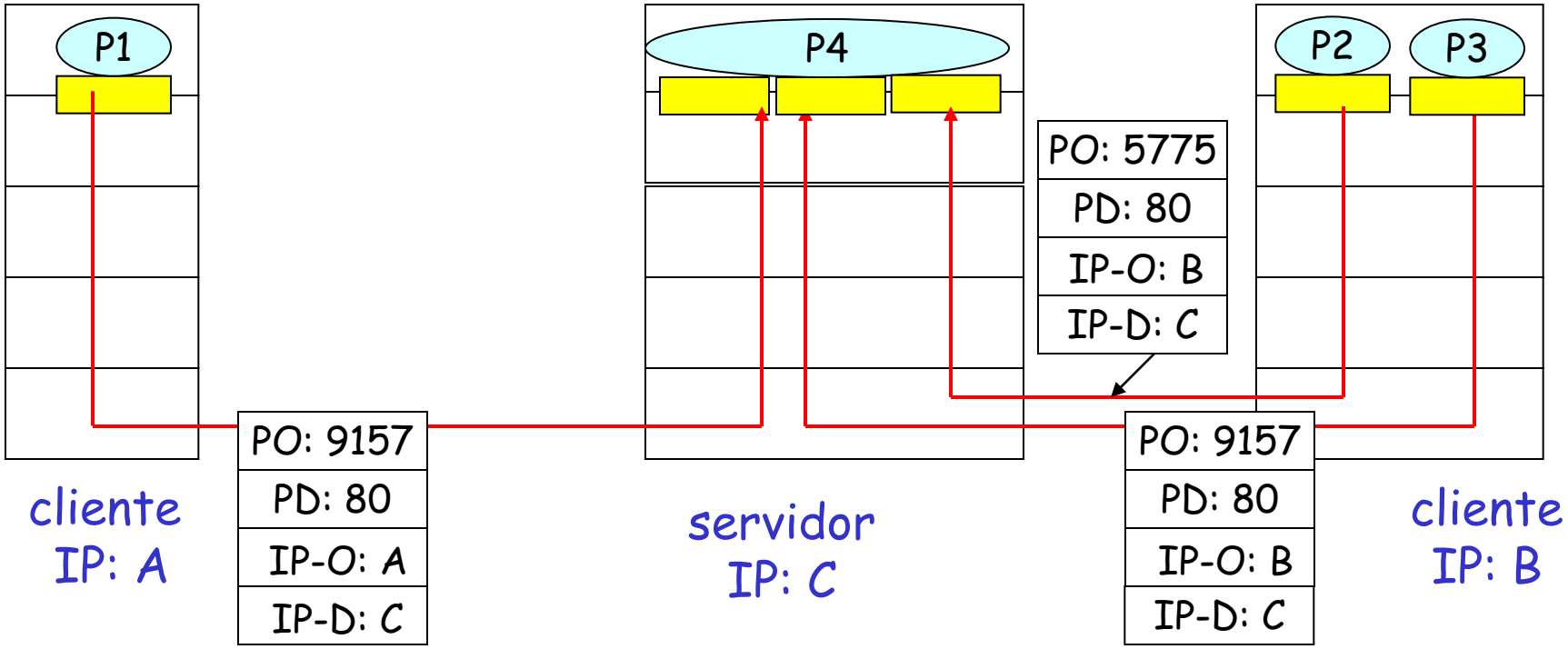
- Socket TCP identificado por 4 valores:
  - Endereço IP de origem
  - End. porta de origem
  - Endereço IP de destino
  - End. porta de destino
  - Na máquina venus:
    - `ServerSocket welcomeSocket = new ServerSocket(6789);`
  - No cliente se comunicando com venus:
    - `Socket clientSocket = new Socket("venus.inf.ufsc.br", 6789);`
- Hospedeiro receptor usa os quatro valores para direcionar o segmento ao socket apropriado
- Hospedeiro servidor pode suportar vários sockets TCP simultâneos:
  - Cada socket é identificado pelos seus próprios 4 valores
  - Servidores Web possuem sockets diferentes para cada cliente conectado
  - HTTP não persistente terá um socket diferente para cada requisição

Demux orientada à conexão



PO: Porta de Origem
PD: Porta de Destino
IP-O: IP de Origem
IP-D: IP de Destino

Demux orientada à conexão: Servidor Web "Threaded"



UDP: User Datagram Protocol [RFC 768]

- Protocolo de transporte da Internet mínimo, "sem frescura",
- Serviço "melhor esforço", segmentos UDP podem ser:
  - Não garante taxa, atraso e taxa de perdas de pacotes
  - entregues à aplicação fora de ordem do remesso
- *Sem conexão:*
  - não há "setup" UDP entre remetente, receptor
  - tratamento independente de cada segmento UDP pela rede

Por quê existe um UDP?

- Elimina estabelecimento de conexão (o que pode causar retardo)
- Simples: não se mantém "estado" da conexão no remetente/receptor
- Pequeno cabeçalho de segmento
- Sem controle de congestionamento e de fluxo: UDP pode transmitir o mais rápido possível (interessante para multimídia)

Protocolo UDP

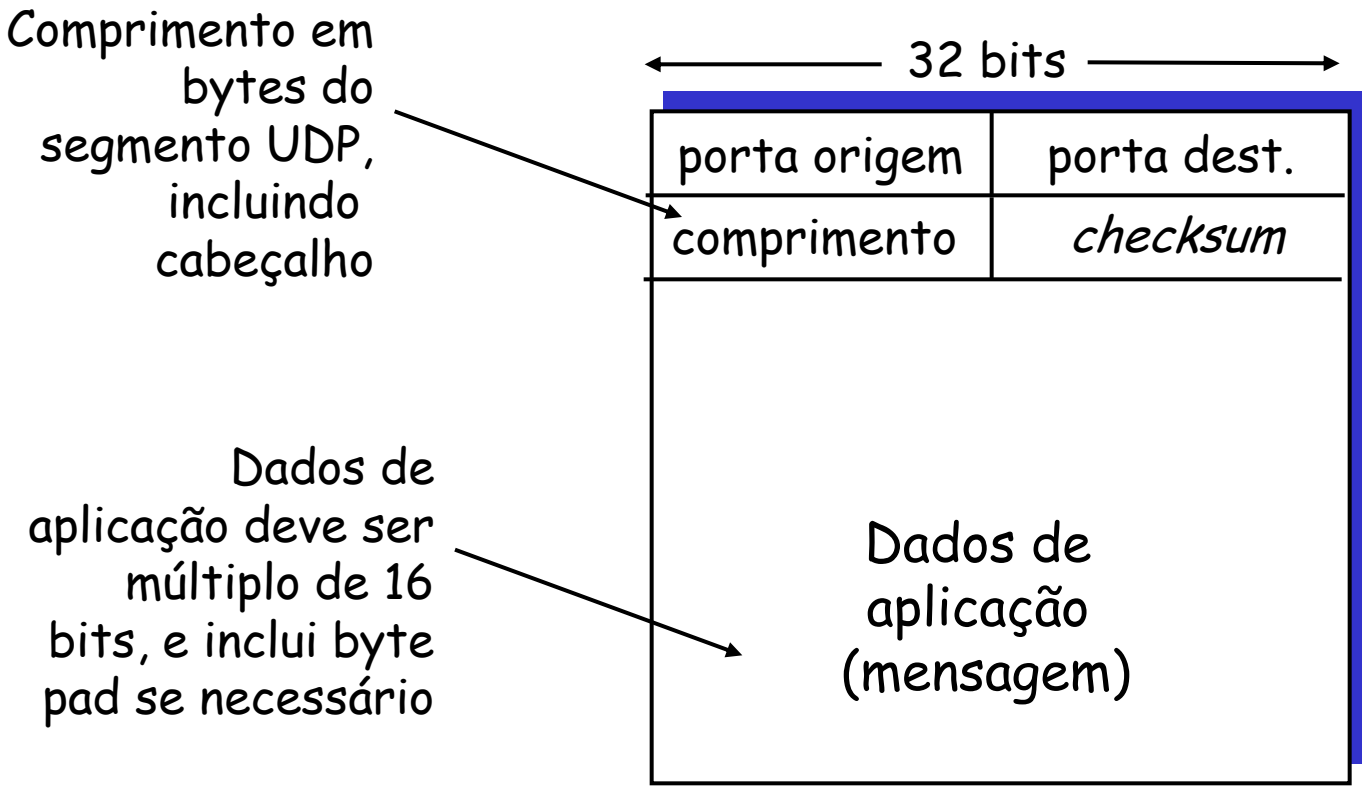
- Não oferece meios que permitam uma transferência confiável de dados
  - não podendo controlar a taxa com que as informações fluem entre as máquinas
  - não implementa mecanismos de reconhecimento, de sequenciação nem de controle de fluxo das mensagens de dados trocadas entre os dois sistemas
  - datagramas podem ser perdidos, duplicados, ou entregues fora de ordem ao sistema de destino
  - aplicação assume toda a responsabilidade pelo controle de erros
  - serve para transportar uma mensagem de uma estação para outra, utilizando o IP para enviar e receber estes datagramas

Protocolo UDP

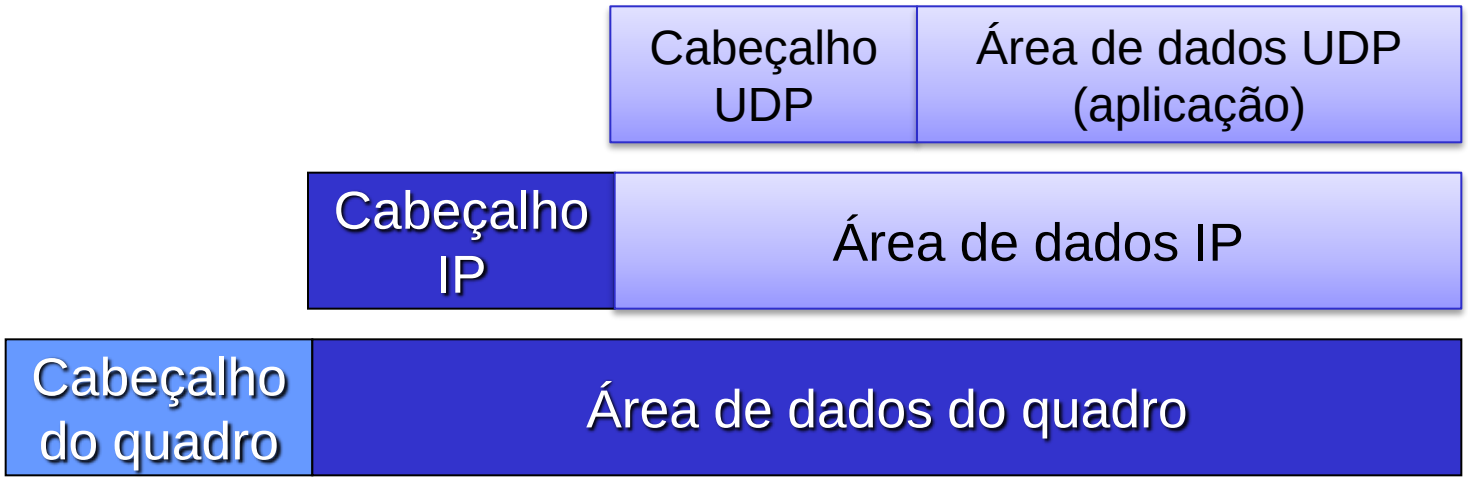
- É um protocolo simples
  - Latência menor
- Usa mais eficientemente a banda da rede
  - Cabeçalho por segmento é menor
  - Sem controle de congestionamento: permite usar a banda de maneira mais eficiente
  - Mas pode provocar taxa de perdas altas
- Muito usado para aplicações multimídia de streaming
  - Tolerantes a perda
  - Sensíveis a taxa
- Outras aplicações que usam UDP
  - DNS, NFS, SNMP e Protocolo de roteamento (RIP)
- Transferência confiável sobre UDP: adicionar confiabilidade na camada de aplicação
  - Recobrimento de erro específico de aplicação

Protocolo UDP

- Formato do Datagrama UDP



Datagrama UDP enviado dentro do datagrama IP



UDP checksum

Meta: detectar erros no segmento transmitido

Emissor:

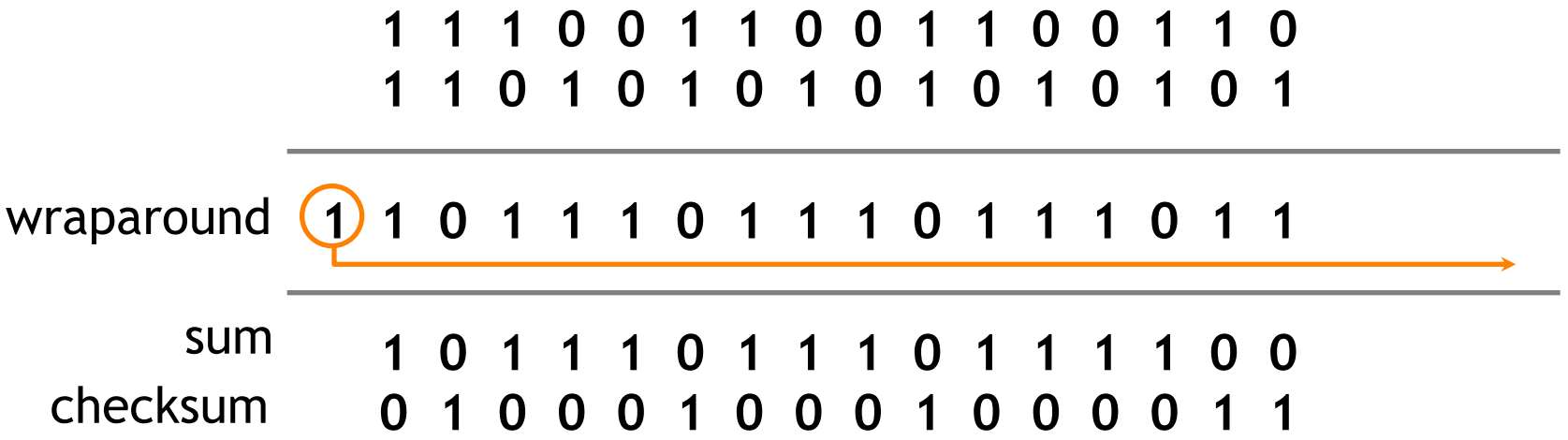
- Trata conteúdo do segmento como uma seqüência de inteiros de 16 bits
- checksum: adição do conteúdo do segmento
  - Resultado é complementado de 1
- Emissor coloca o valor checksum no campo UDP checksum

Receptor:

- Soma toda a seqüência (incluindo checksum)
- Checa se checksum calculada é igual ao valor $FFFFF_h$:
  - NÃO – erro detectado
  - SIM – nenhum erro detectado.

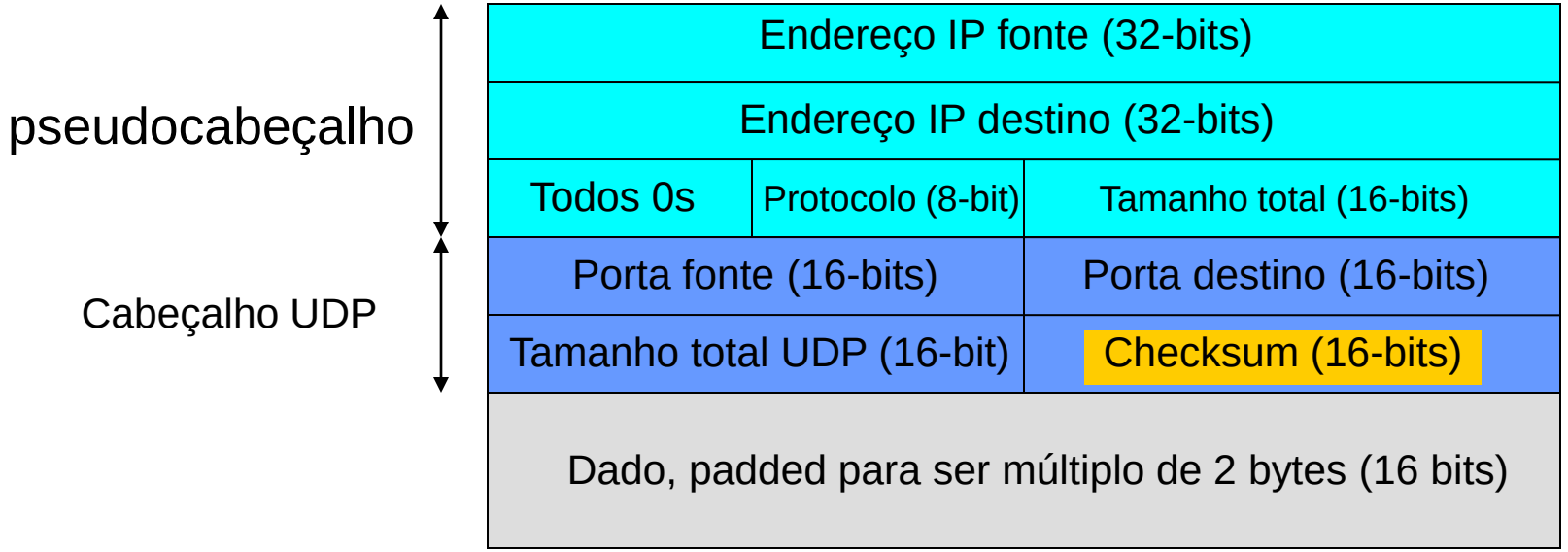
Exemplo: Internet checksum

- Note que:
  - ao se adicionar números, um vai um do bit mais significativo deve ser acrescentado ao resultado
- Exemplo: adicione dois inteiros de 16 bits



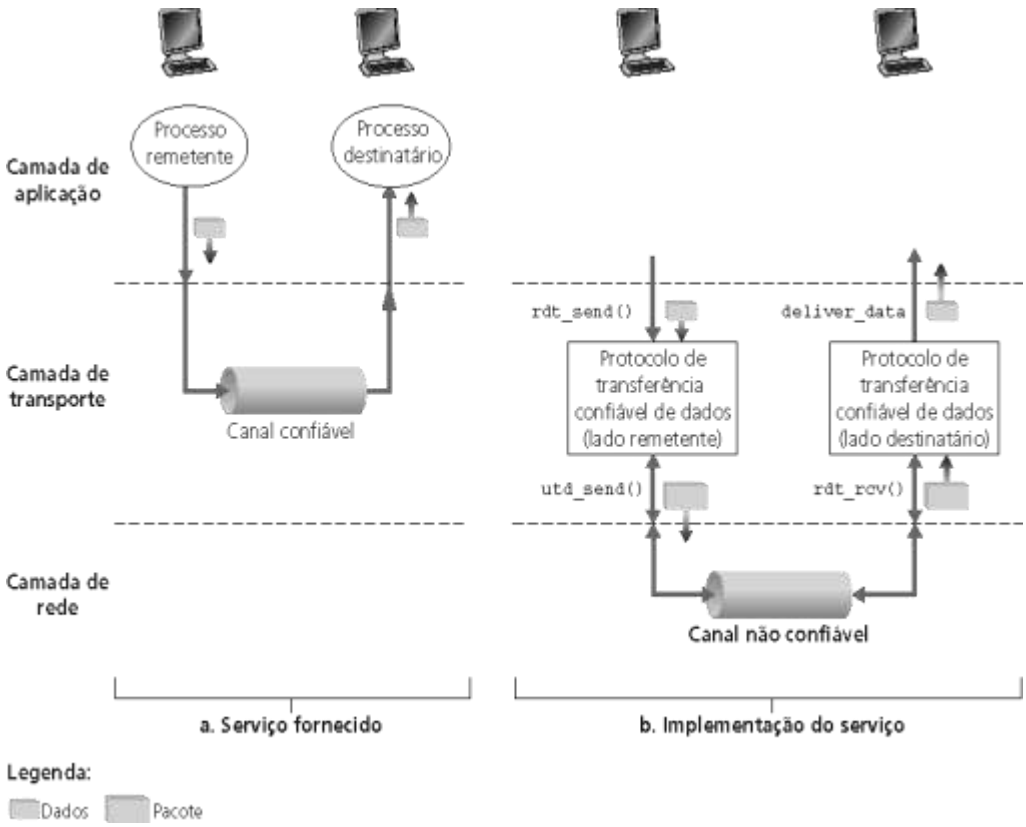
UDP Checksum e Pseudocabeçalho

- UDP checksum cobre
  - Dado da aplicação, cabeçalho UDP, um pseudocabeçalho, e o byte pad (se necessário)
- Propósido de incluir o pseudo-cabeçalho:
  - Checagem para ver se o pacote chegou no destino correto
  - Checa se o IP entregou para o protocolo correto (UDP/TCP)



Princípios de transferência confiável de dados

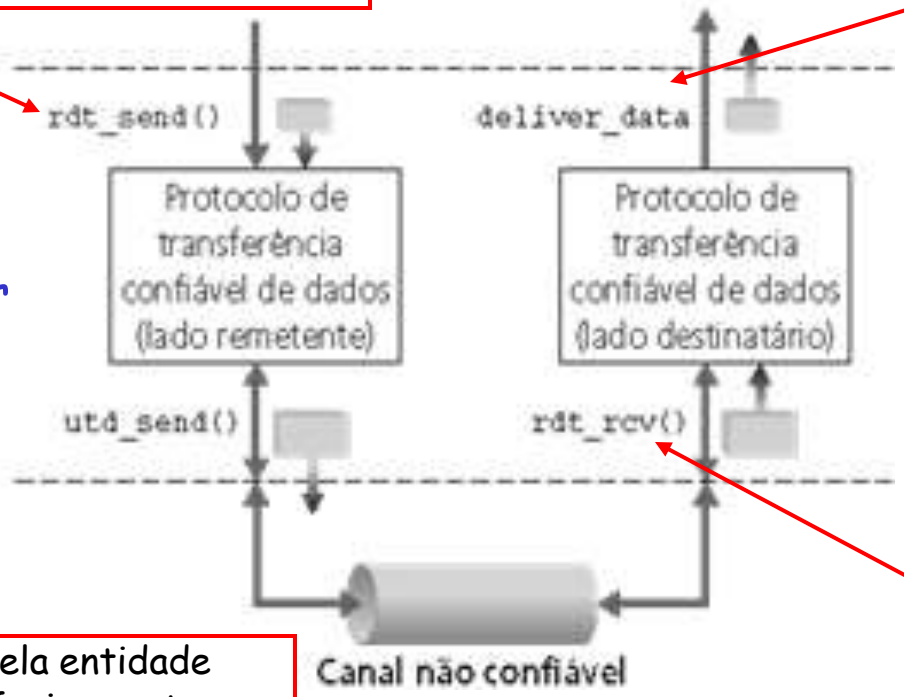
- Importante nas camadas de aplicação, transporte e enlace
- Top 10 na lista dos tópicos mais importantes de redes!
- Características dos canais não confiáveis determinarão a complexidade dos protocolos confiáveis de transferência de dados



Transferência confiável: o ponto de partida

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

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



lado transmissor

lado receptor

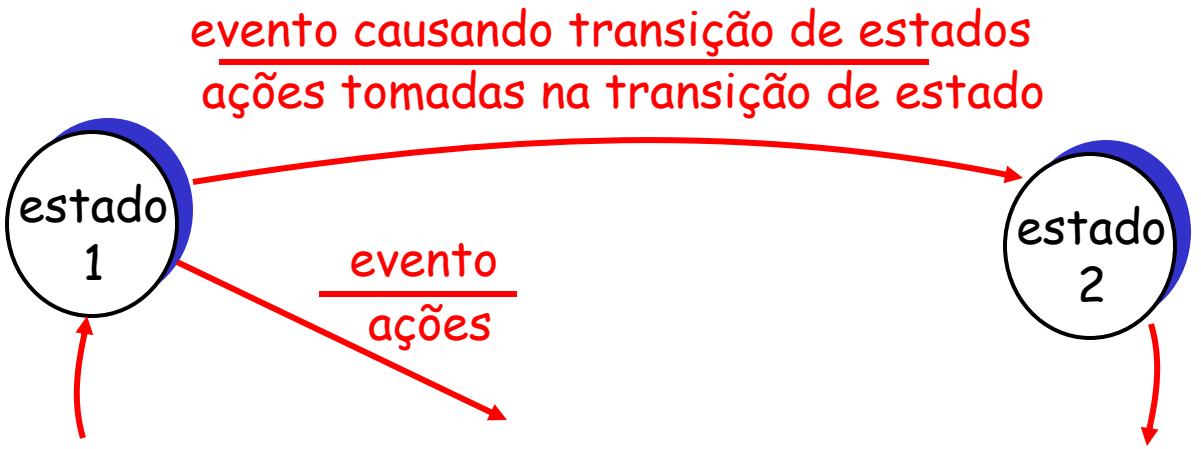
`udt_send()`: chamada pela entidade de transporte, para transferir pacotes para o receptor sobre o canal não confiável

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

Transferência confiável: o ponto de partida

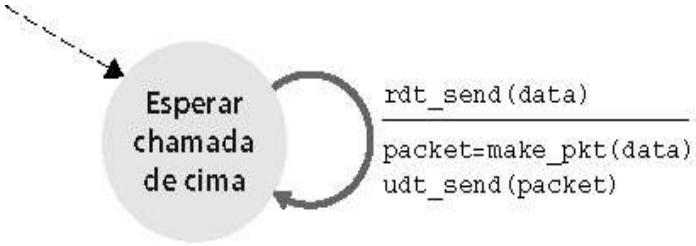
- Estudo incremental de mecanismos para prover confiabilidade na transmissão de dados:
  - Veremos algumas versões de um protocolo confiável (rdt)
  - Consideraremos apenas transferências de dados unidirecionais
    - Mas informação de controle deve fluir em ambas as direções!
  - Usaremos máquinas de estados finitos (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 canais confiáveis

- Canal de transmissão perfeitamente confiável
  - Não há erros de bits
  - Não há perdas de pacotes
  - Seria o serviço ofecido pelo TCP aos protocolos de aplicação (p.e., HTTP)
- FSMs separadas para transmissor e receptor:
  - Transmissor envia dados para o canal subjacente
  - Receptor lê os dados do canal subjacente

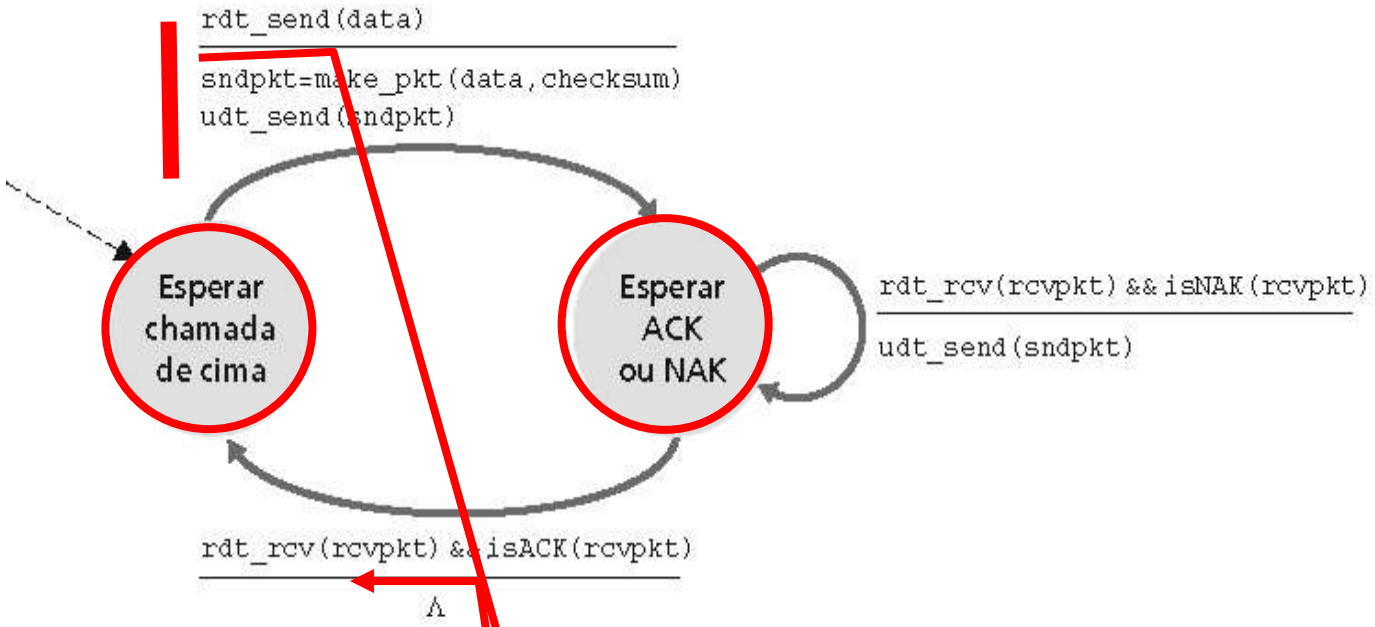


a. rdt1.0: lado remetente

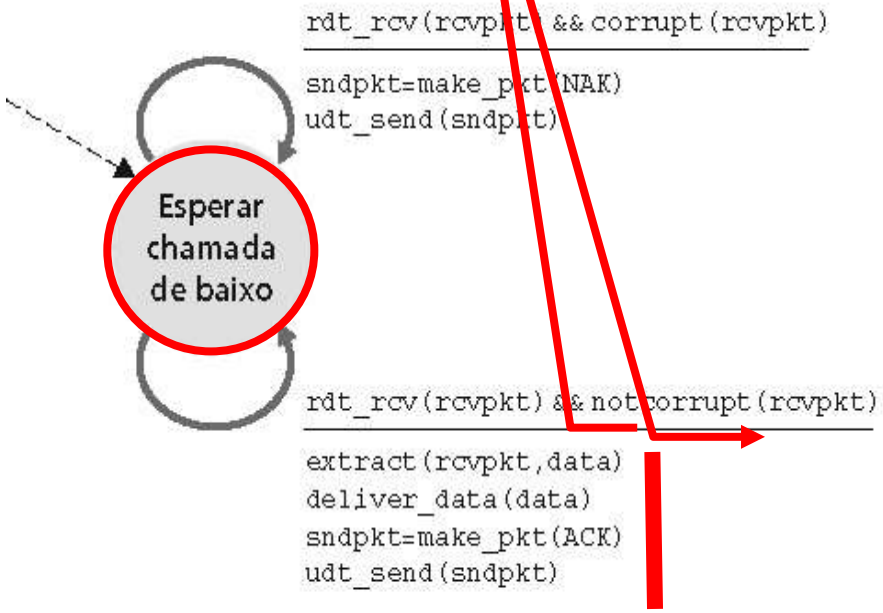
b. rdt1.0: lado destinatário

rdt2.0: canal com erros de bit

- Canal subjacente pode trocar valores dos bits num pacote
  - Mecanismos de segurança: **Checksum**
    - para detectar erros de bits
- A questão: como recuperar esses 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):
  - Detecção de erros
  - Retorno do receptor: mensagens de controle (ACK, NAK) rcvr->sender

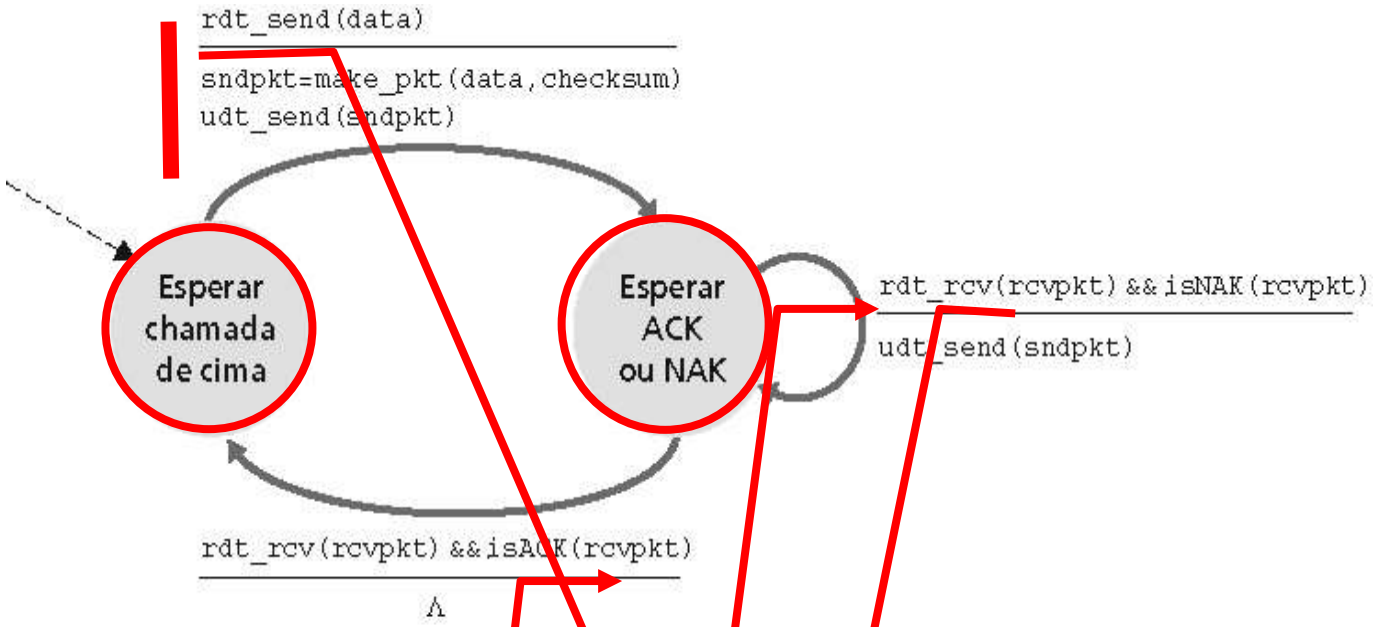


a. rdt2.0: lado remetente

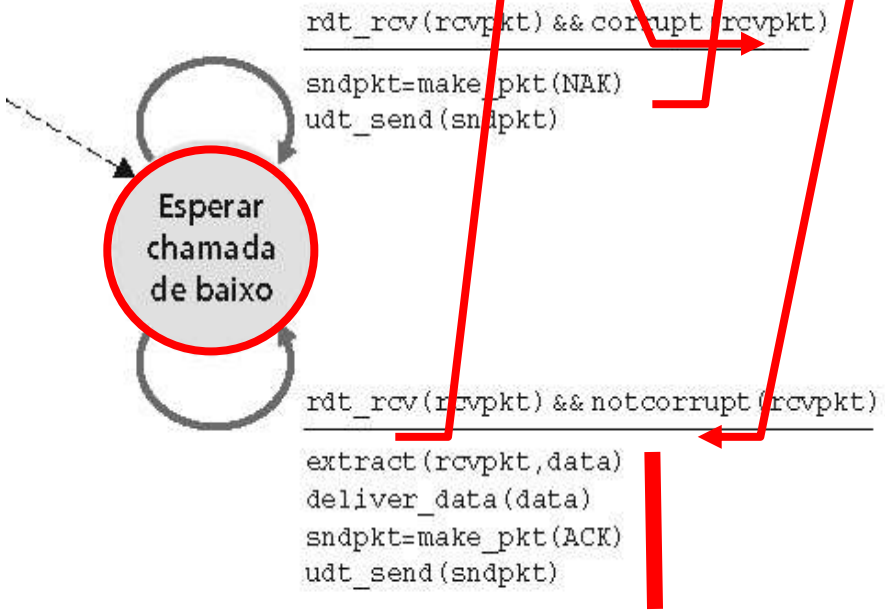


b. rdt2.0: lado destinatário

Cenário sem erros



a. rdt2.0: lado remetente

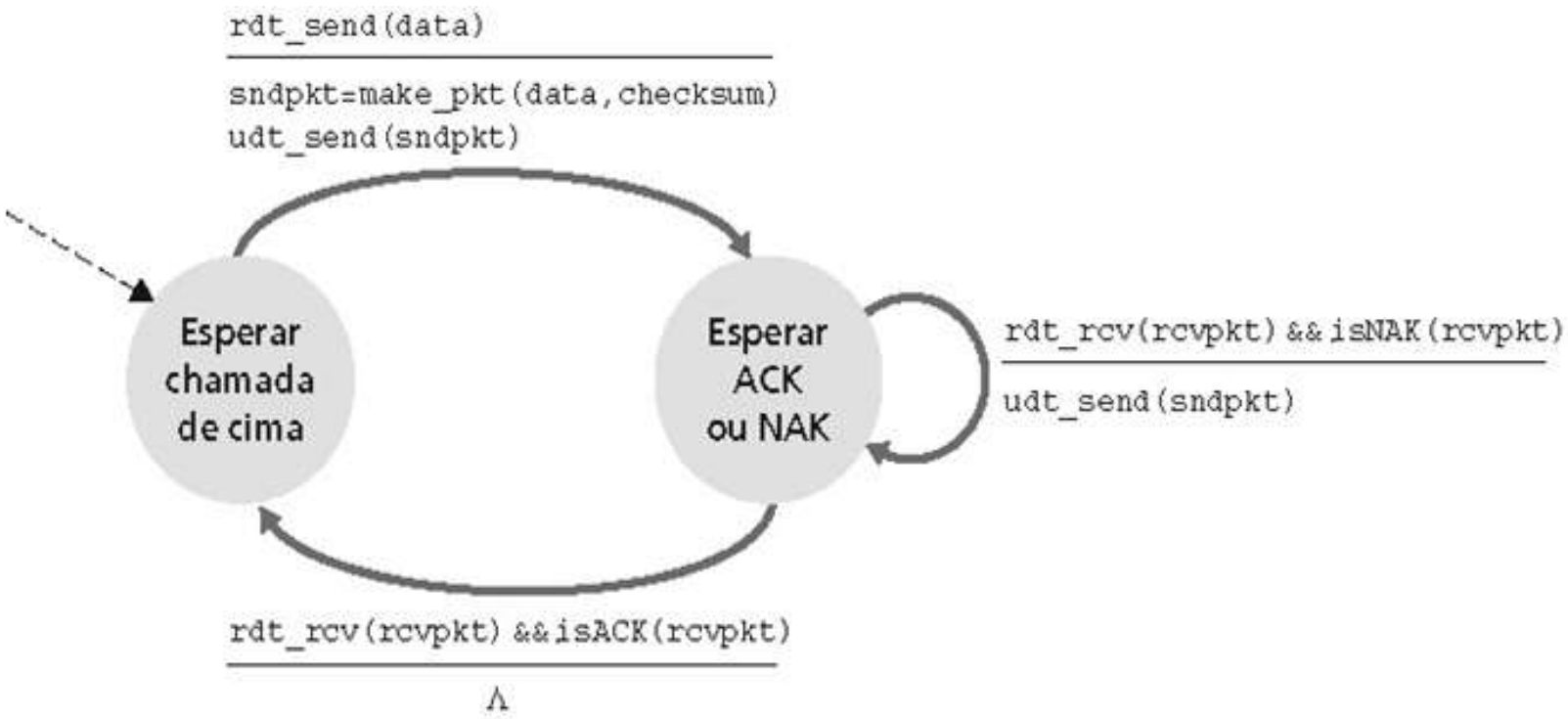


b. rdt2.0: lado destinatário

Cenário com erros

rdt2.0 tem um problema fatal!

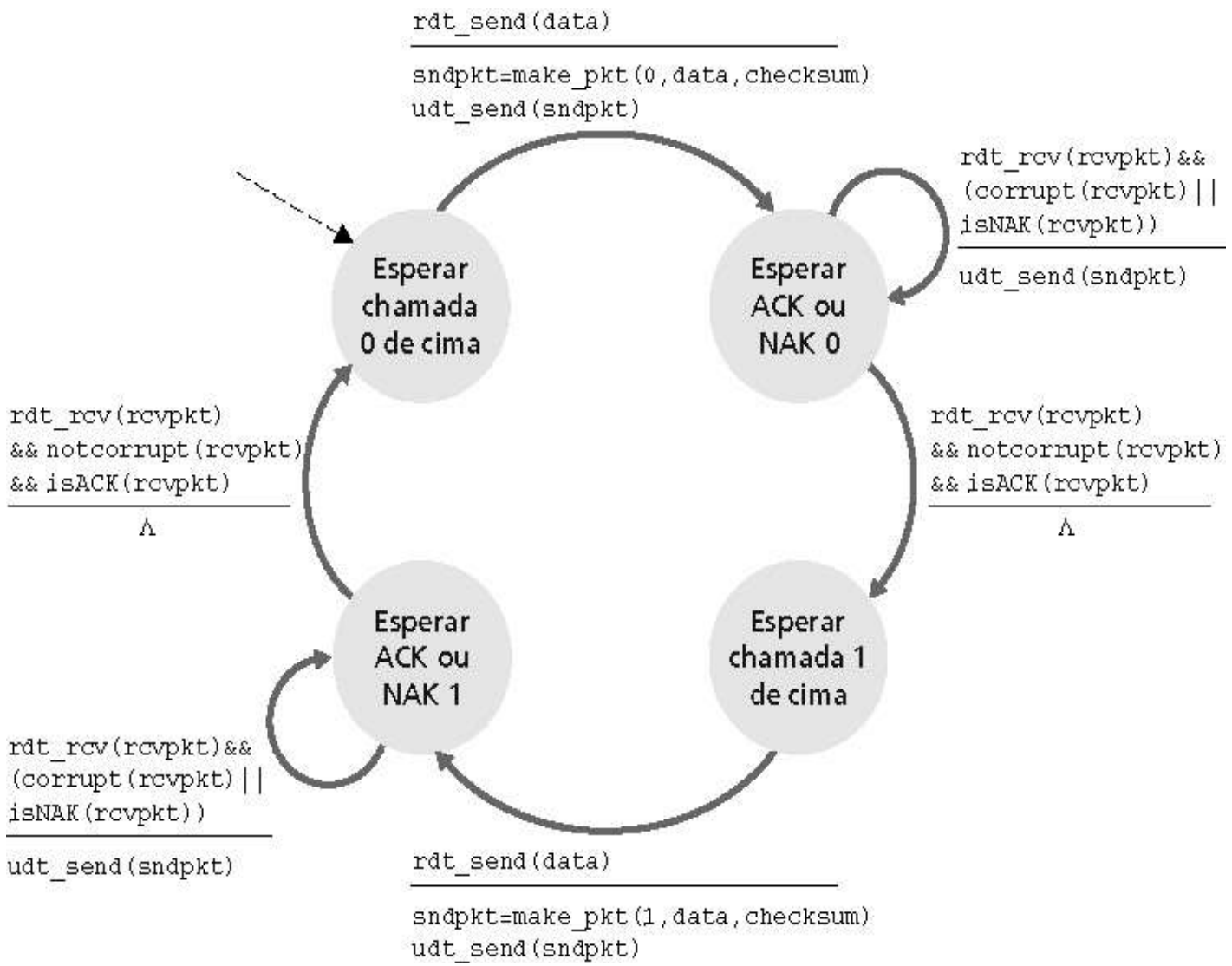
- O que acontece se o ACK/NAK é corrompido?
  - Assumimos que não há perda de pacote na rede
  - Transmissor não sabe o que aconteceu no receptor!
  - Não pode apenas retransmitir : possível duplicata



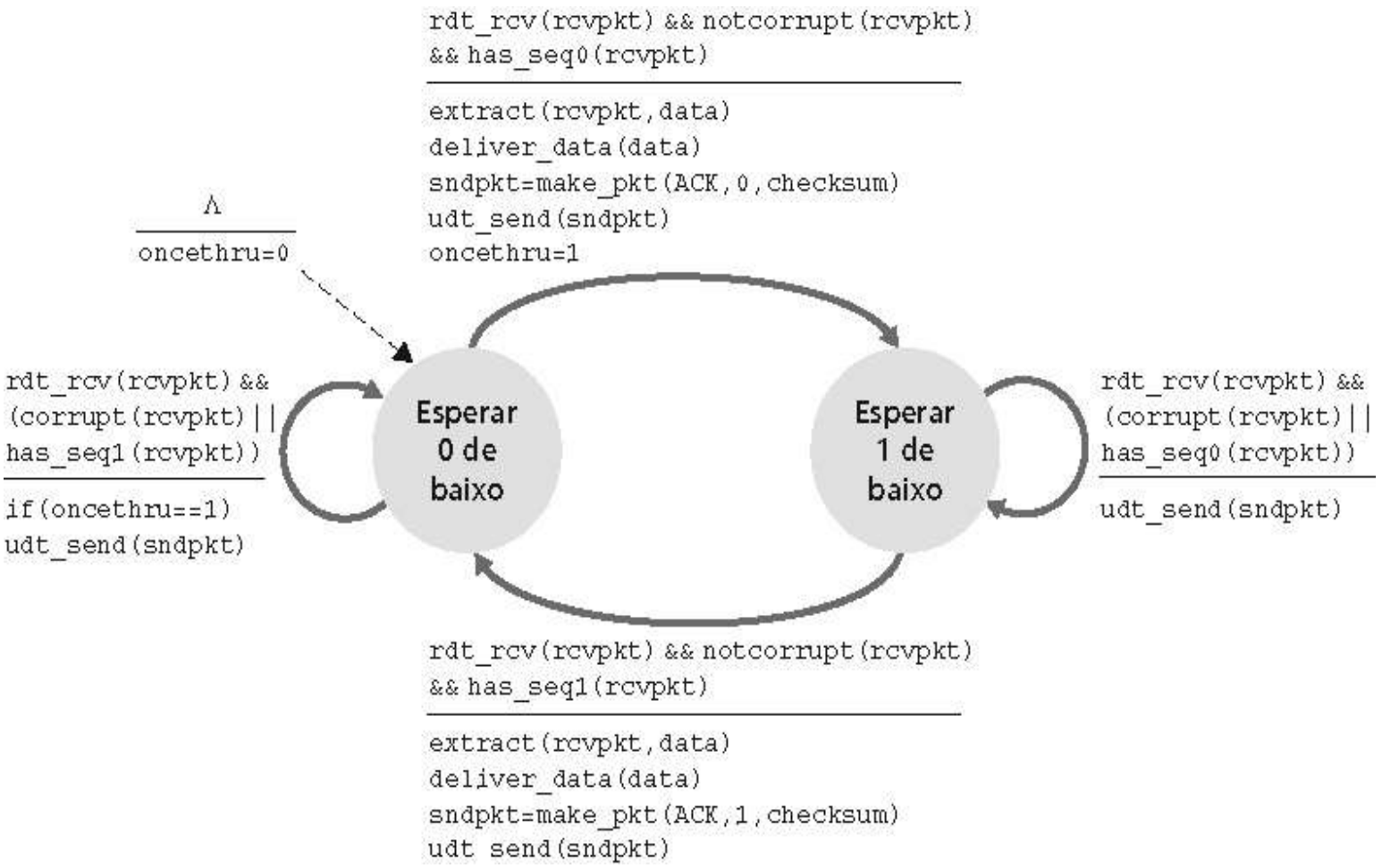
rdt2.0 tem um problema fatal!

- 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
- Consideramos o modo transmite e aguarde!!!
  - Transmissor envia um pacote e então espera pela resposta do receptor

rdt2.1: transmissor, trata ACK/NAKs perdidos



rdt2.1: receptor, trata ACK/NAKs perdidos



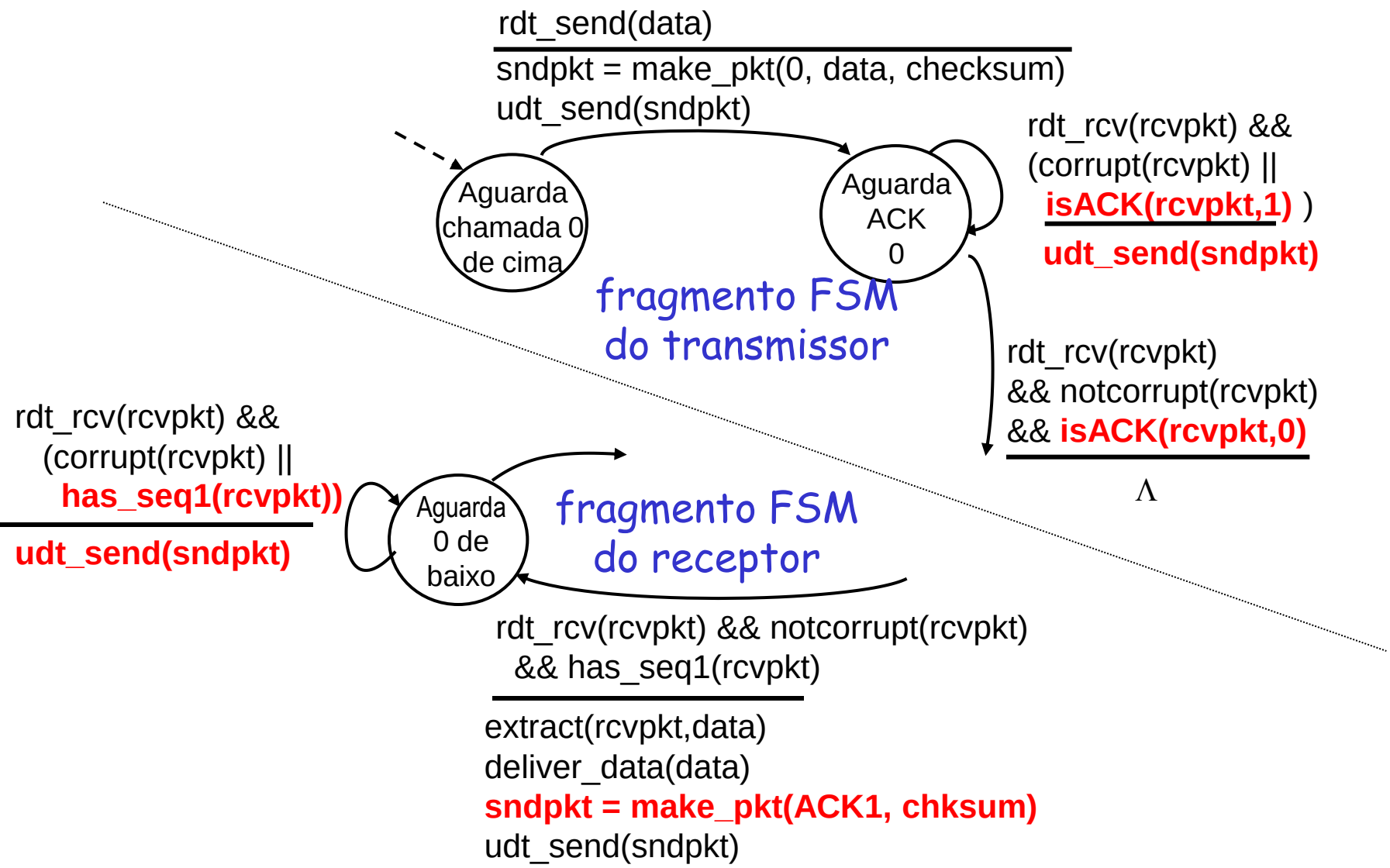
rdt2.1: discussão

- Transmissor:
  - Adiciona número de seqüência ao pacote
  - Dois números (0 e 1) bastam. Por quê?
  - 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 seu último ACK/NAK foi recebido pelo transmissor

rdt2.2: um protocolo sem NAK

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

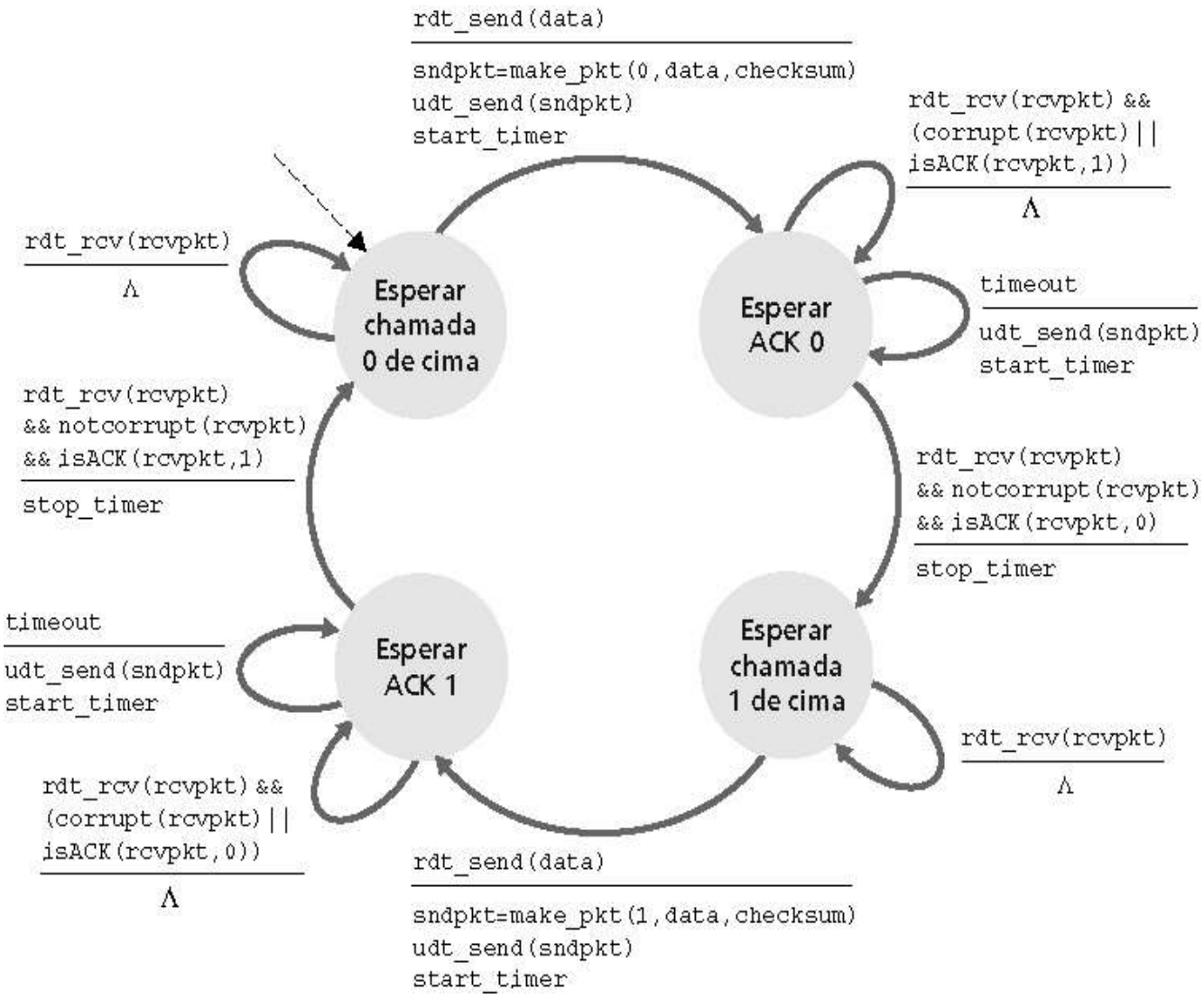
rdt2.2: fragmentos do transmissor e do receptor



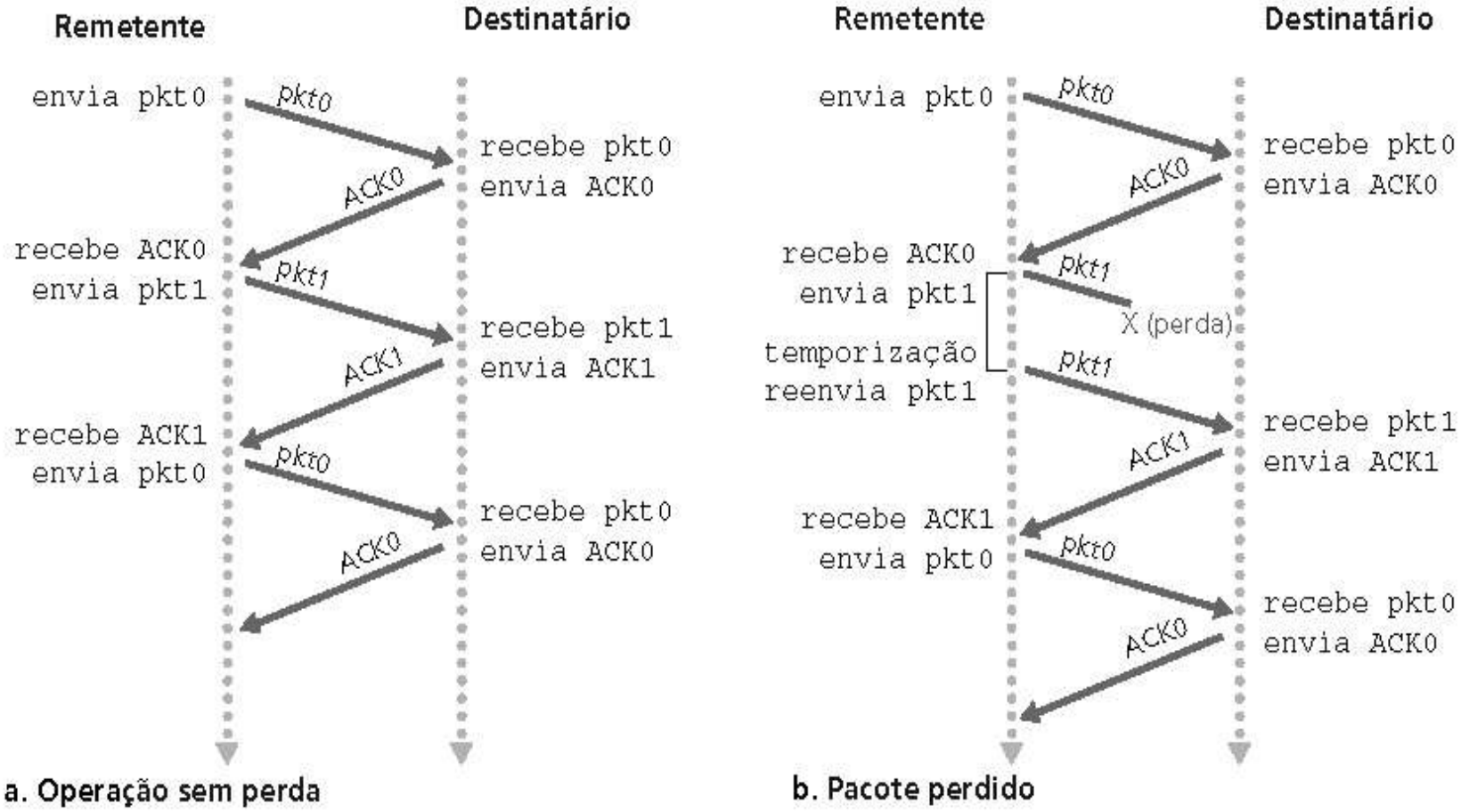
rdt3.0: canais com erros e perdas

- Nova hipótese: canal de transmissão pode também perder pacotes
  - Checksum, números de seqüência, ACKs, retransmissões serão de ajuda, mas não o bastante
- Abordagem: transmissor espera um tempo "razoável" pelo ACK
  - Retransmite se nenhum ACK for recebido nesse tempo
  - 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
  - Exige um temporizador decrescente

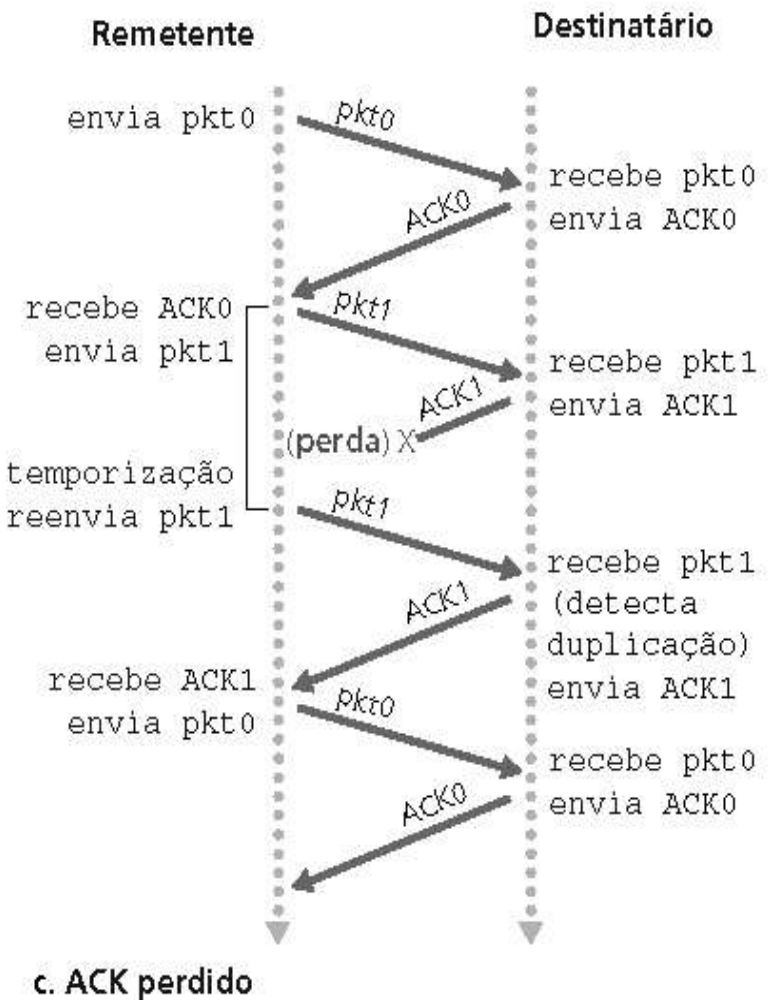
Transmissor rdt3.0



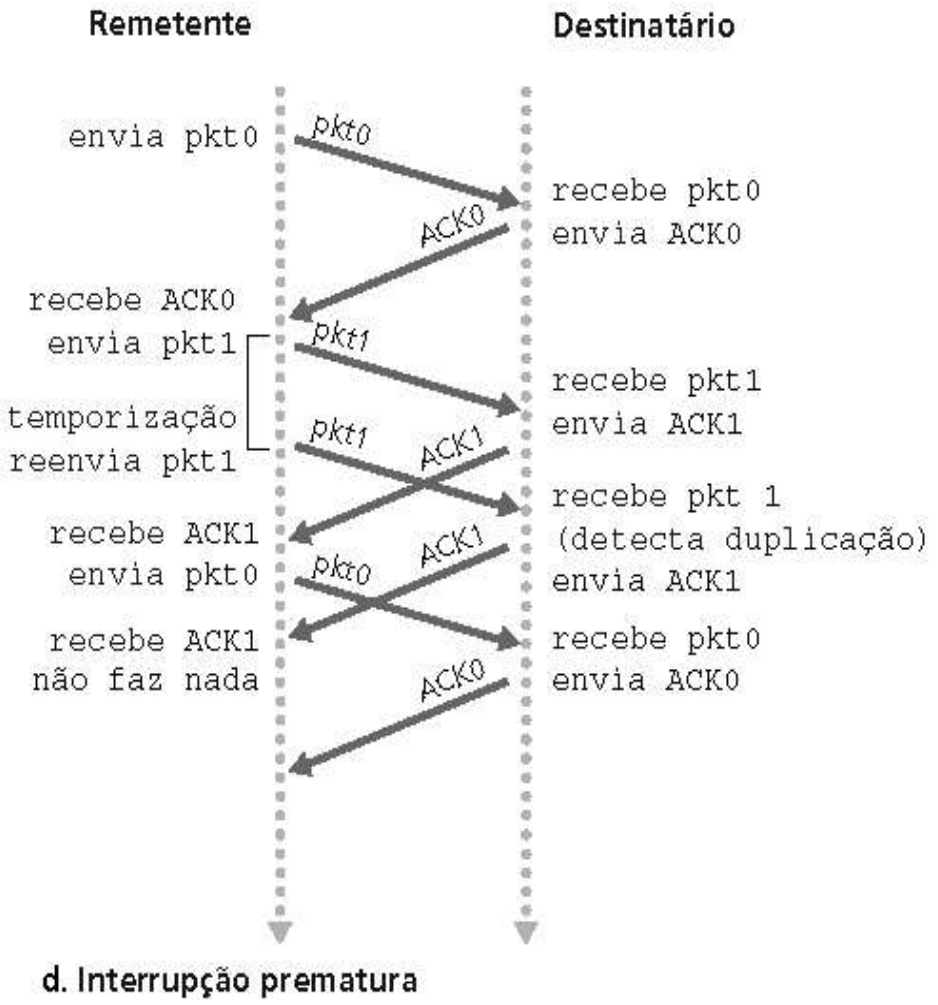
rdt3.0 em ação



rdt3.0 em ação



c. ACK perdido



d. Interrupção prematura

Desempenho do rdt3.0

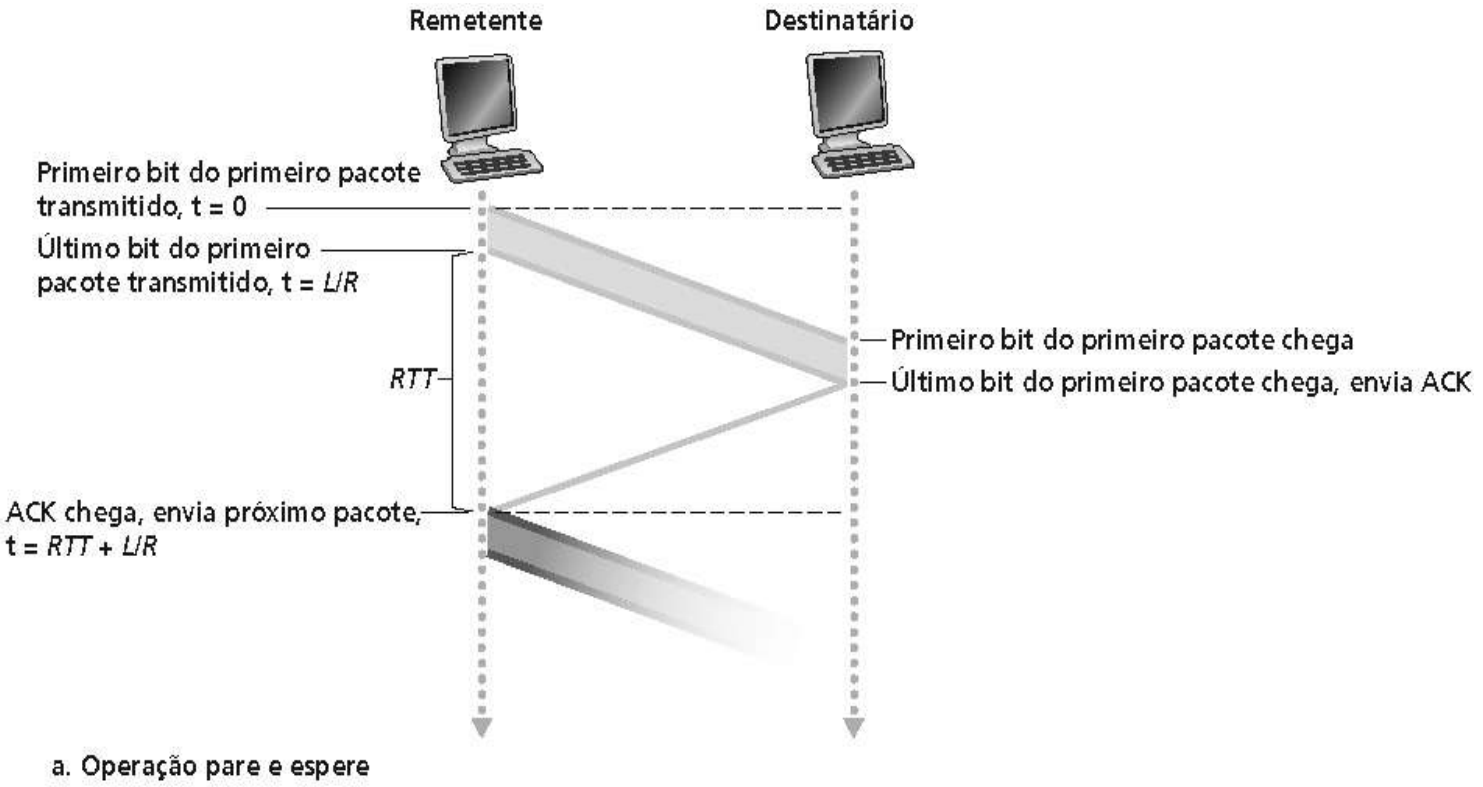
- rdt3.0 funciona, mas o desempenho é sofrível
- Exemplo: enlace de 1 Gbps, 15 ms de atraso de ida ao destino, pacotes de 1 KB:

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

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

- U_{remet}: utilização – fração de tempo do transmissor ocupado
- Um pacote de 1 KB cada 30 ms -> 33 kB/s de vazão sobre um canal de 1 Gbps
- O protocolo de rede limita o uso dos recursos físicos!

rdt3.0: operação pare e espere

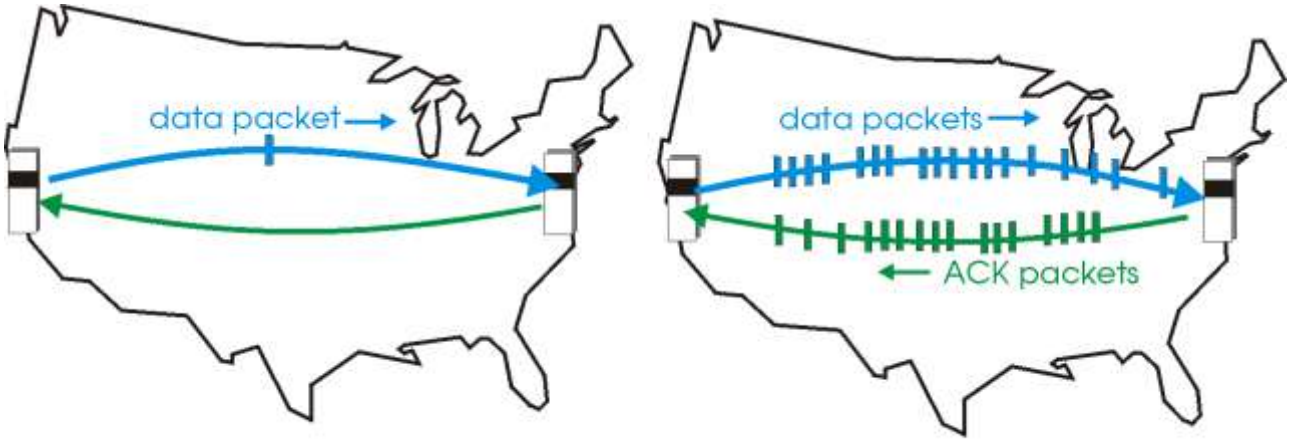


a. Operação pare e espere

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

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

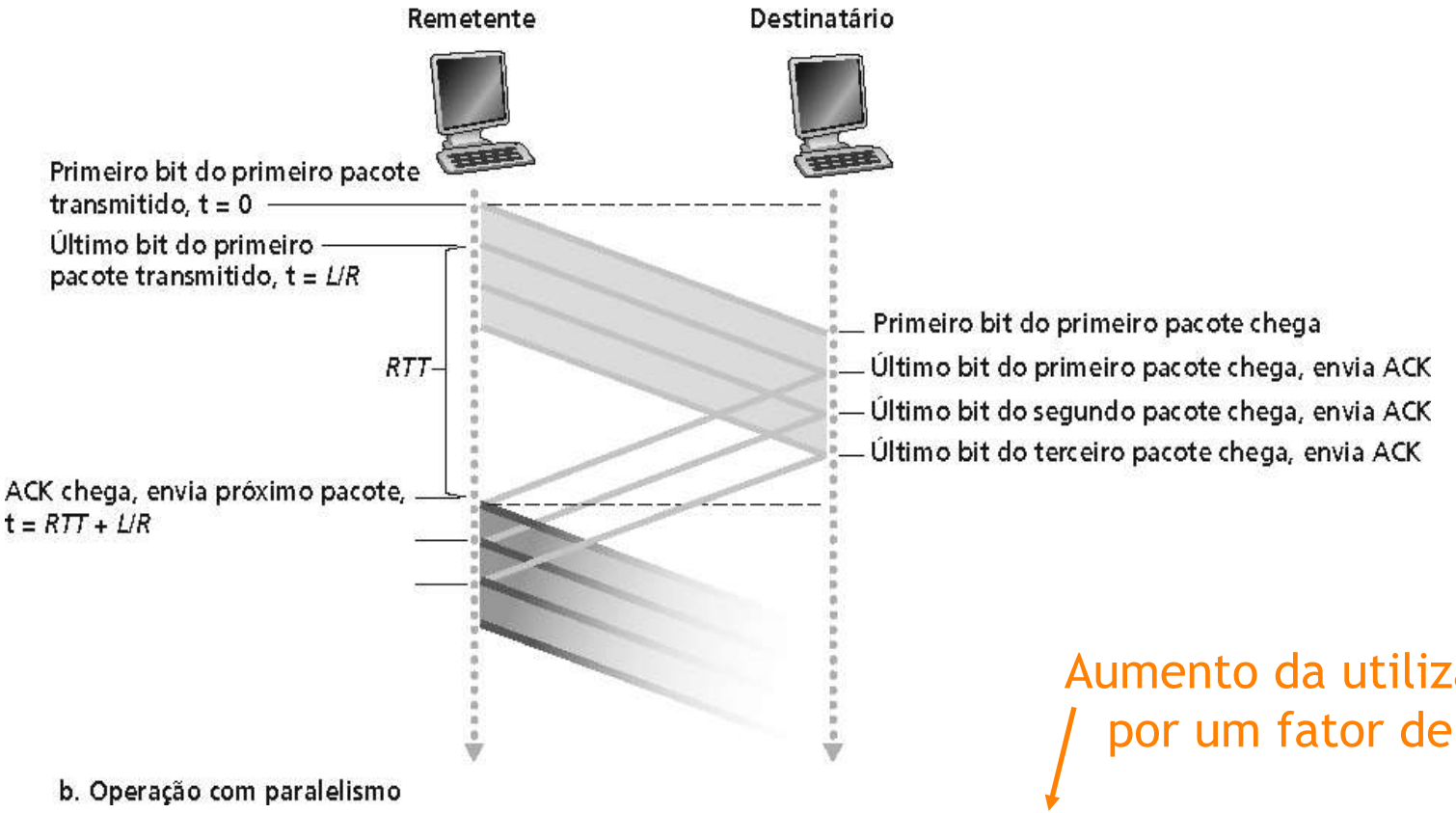


(a) operação do protocolo pare e espere

(a) operação do protocolo com paralelismo

- Duas formas genéricas de protocolos com paralelismo: **go-Back-N, retransmissão seletiva**

Pipelining: aumento da utilização



b. Operação com paralelismo

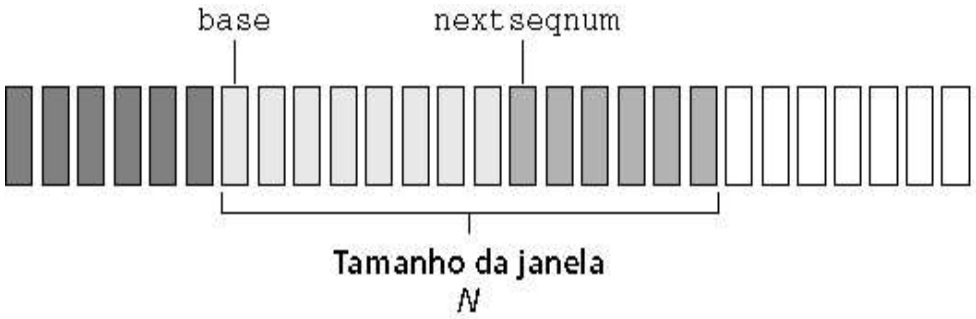
Aumento da utilização por um fator de 3!

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

Go-Back-N

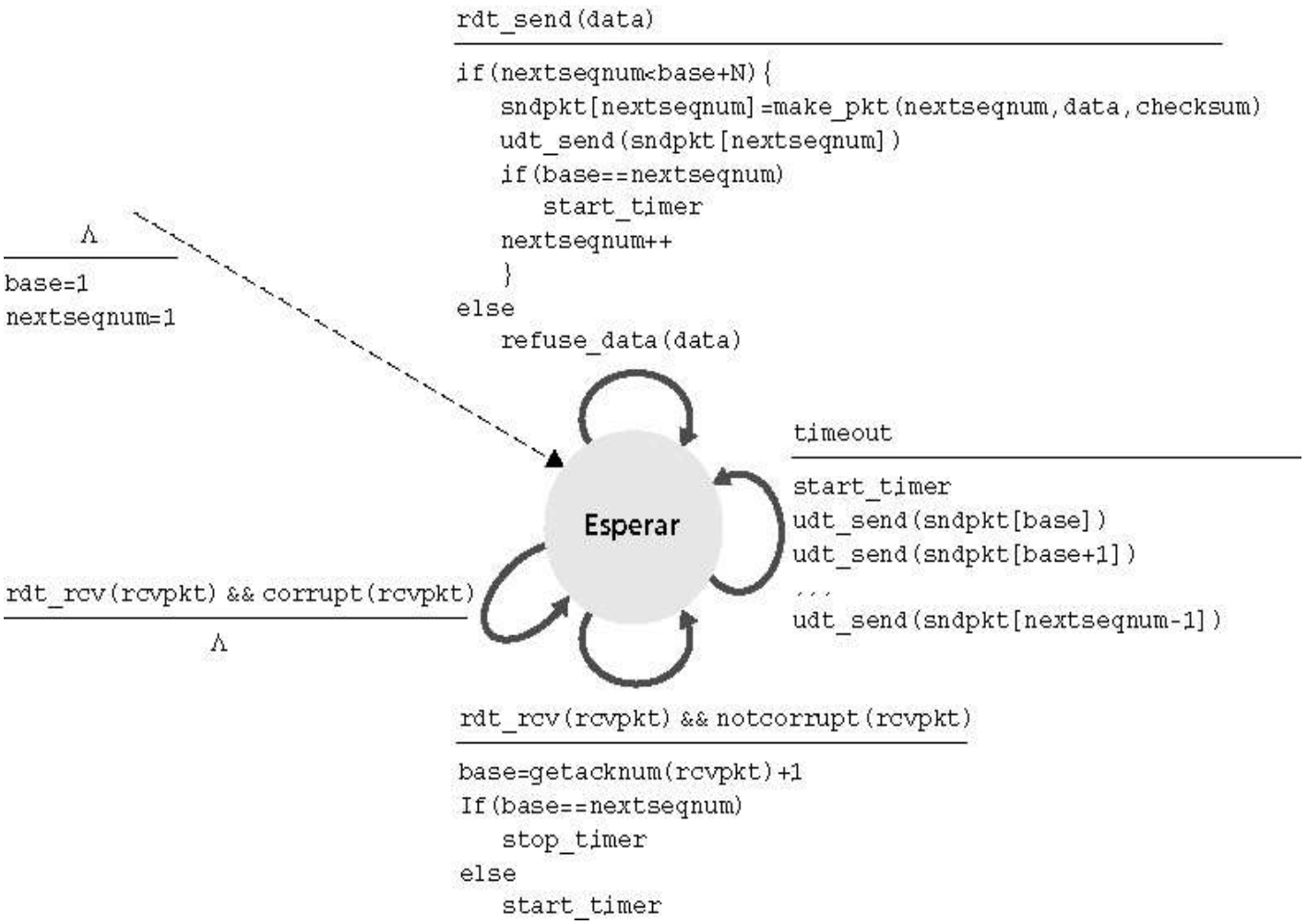
Transmissor:

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

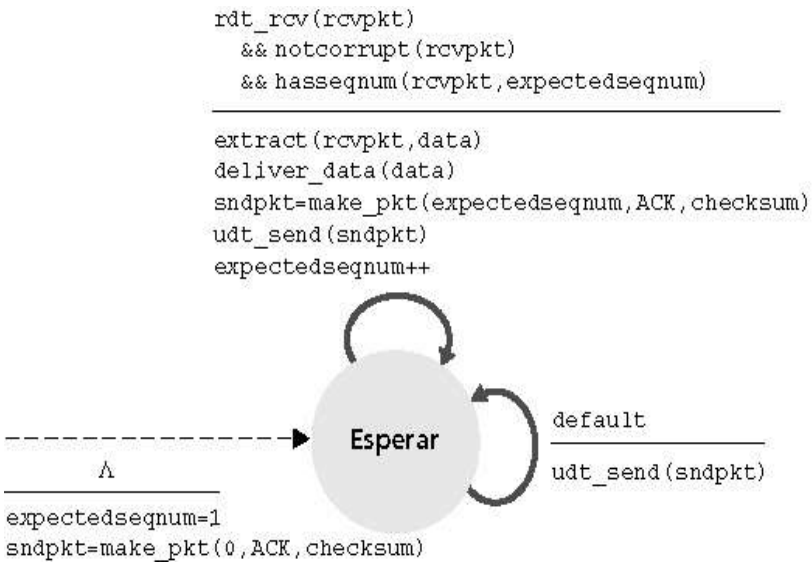


- ACK(n): reconhece todos os pacotes até o número de seqüência N (incluindo este limite). "ACK cumulativo"
  - Pode receber ACKs duplicados (veja receptor)
- Temporizador para cada pacote enviado e não confirmado
- **Tempo de confirmação** (**n**): retransmite pacote n e todos os pacotes com número de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

GBN: FSM estendida para o transmissor

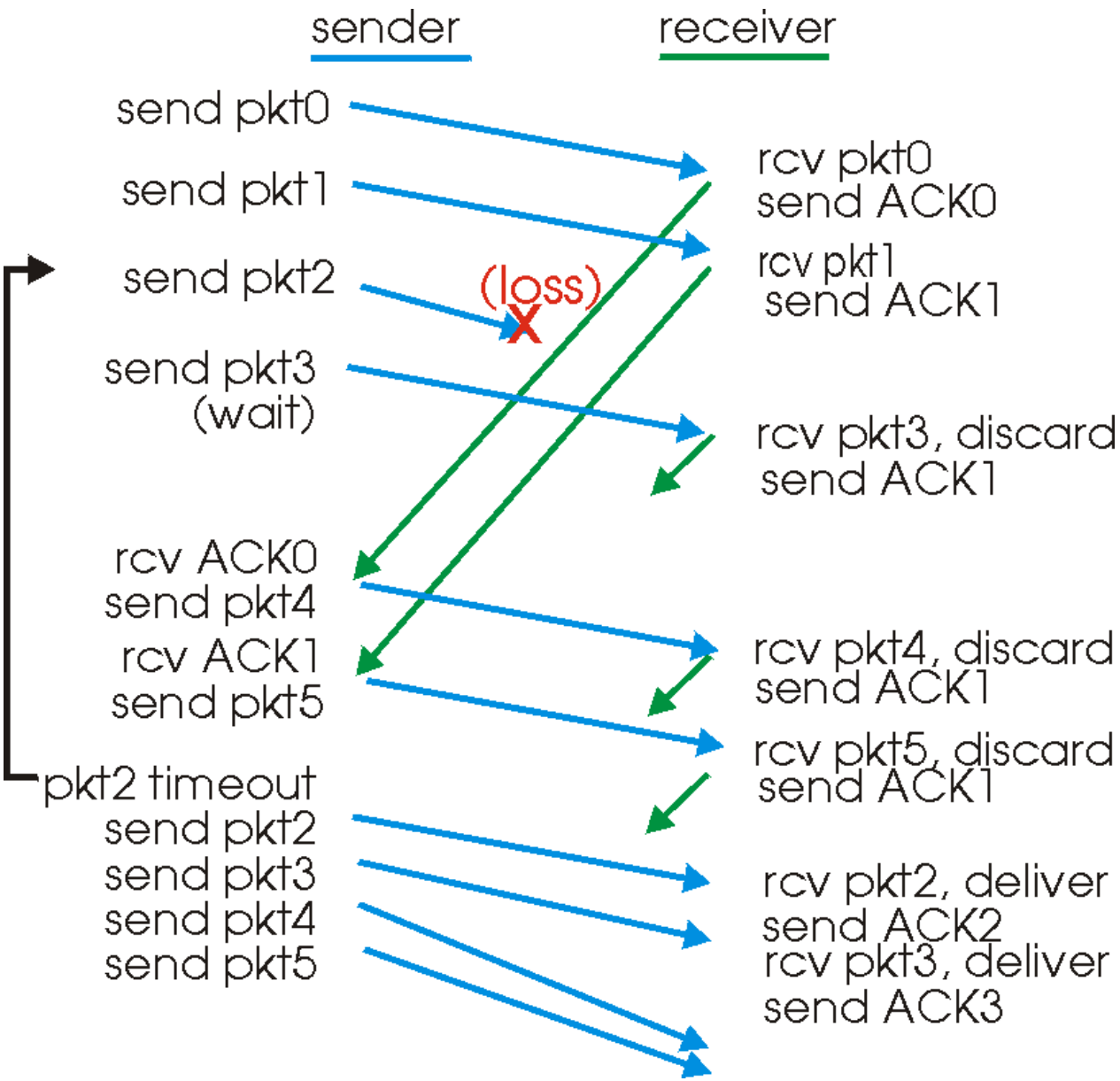


GBN: FSM estendida para o receptor



- Somente ACK: sempre envia ACK para pacotes corretamente recebidos com o mais alto número de seqüência em ordem
  - Pode gerar ACKs duplicados
  - Precisa lembrar apenas do **expectedseqnum**
- Pacotes fora de ordem:
  - Descarta (não armazena) -> não há buffer de recepção!
  - Reconhece pacote com o mais alto número de seqüência em ordem

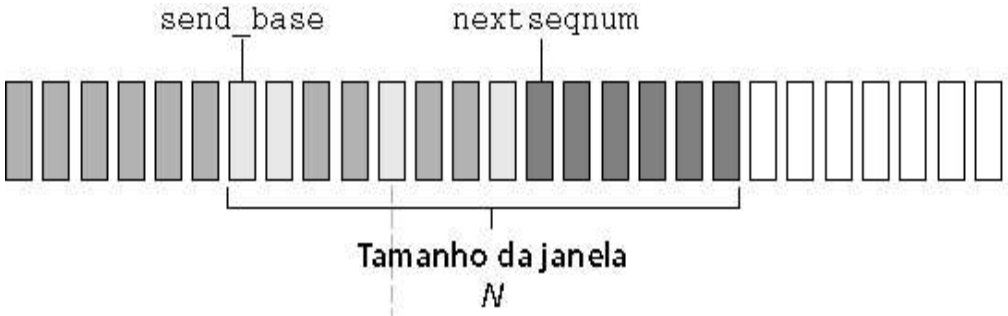
GBN em ação


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

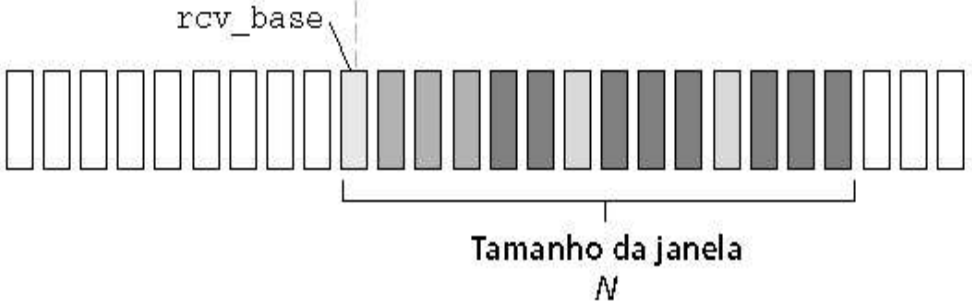
Retransmissão seletiva

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

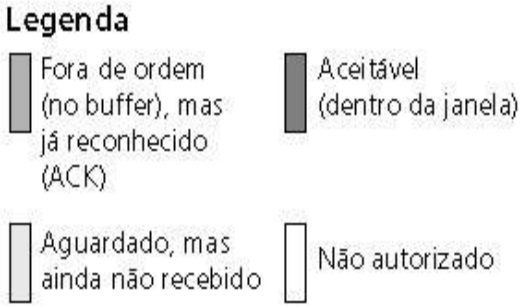
Retransmissão seletiva: janelas do transmissor e do receptor



a. Visão que o remetente tem dos números de seqüência



b. Visão que o destinatário tem dos números de seqüência



TRANSMISSOR

Dados da camada superior:

- Se o próximo número de seqüência disponível está na janela, envia o pacote

Tempo de confirmação(n):

- Reenvia pacote n, restart timer

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

- Marca pacote n como recebido
- Se n é o menor pacote não reconhecido, avança a base da janela para o próximo número de seqüência não reconhecido

RECEPTOR

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

- Envia ACK(n)
- Fora de ordem: armazena
- Em ordem: entrega (também entrega pacotes armazenados em ordem), avança janela para o próximo pacote ainda não recebido

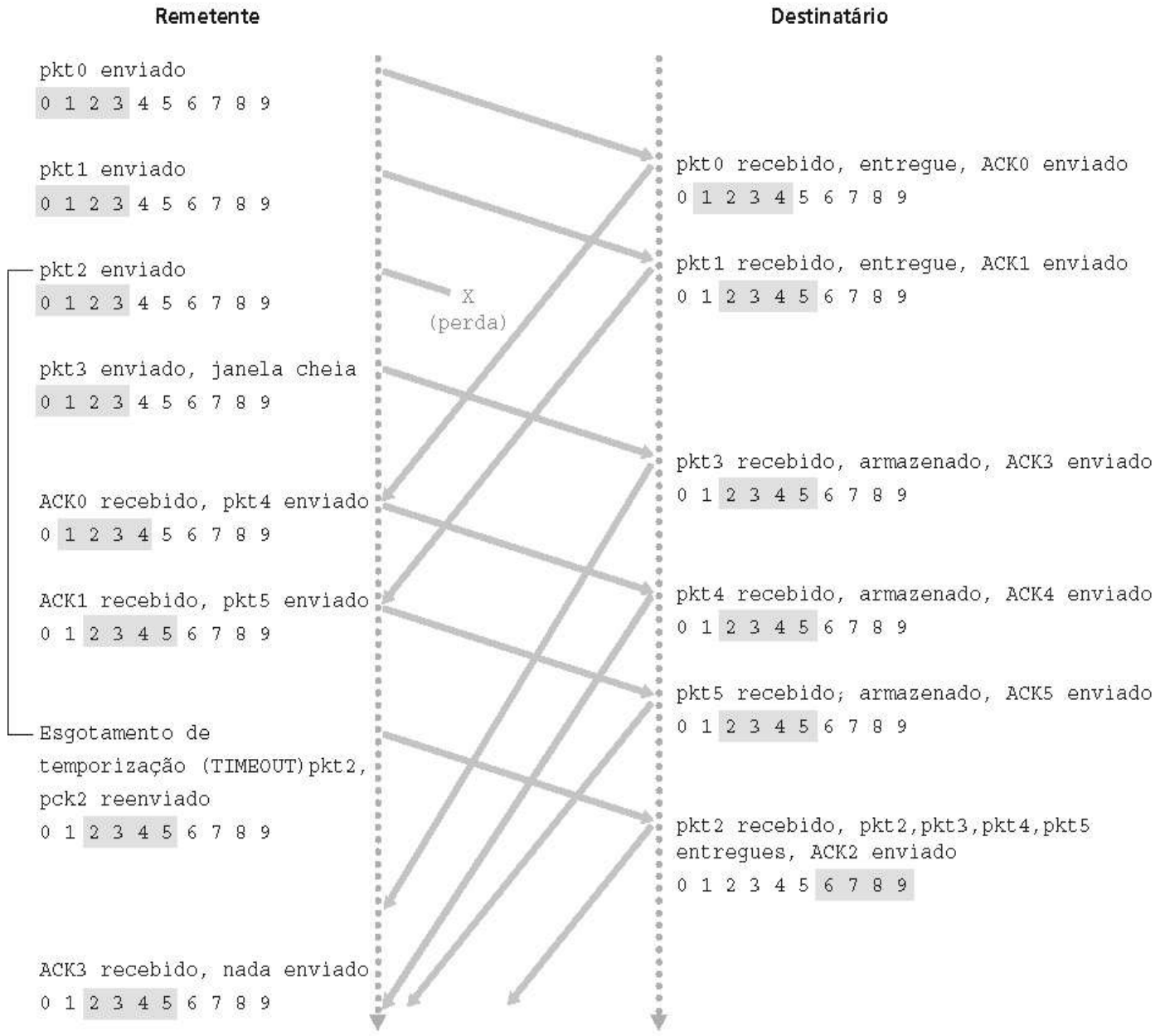
pkt n em [rcvbase-N,rcvbase-1]

- ACK(n)

Caso contrário:

- Ignora

Retransmissão seletiva em ação

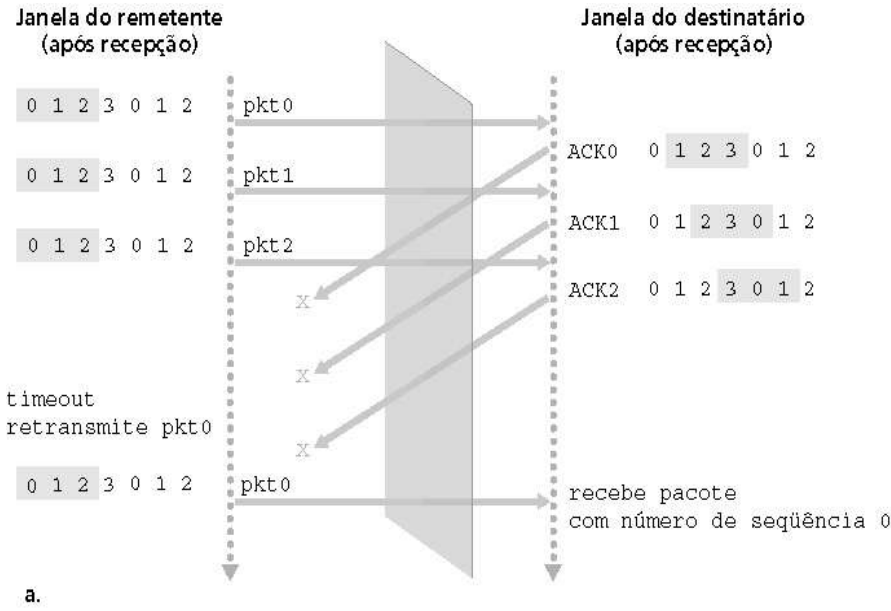


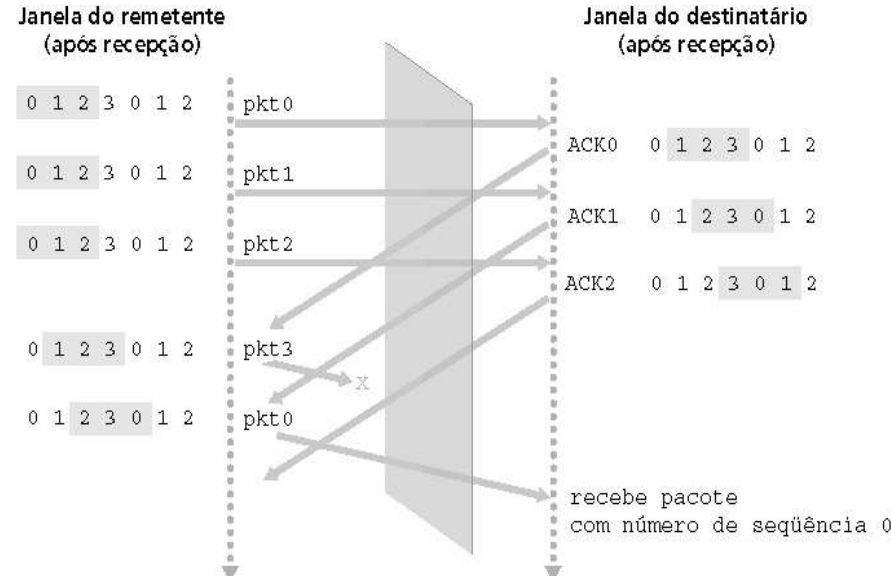
Retransmissão seletiva: dilema

Exemplo:

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

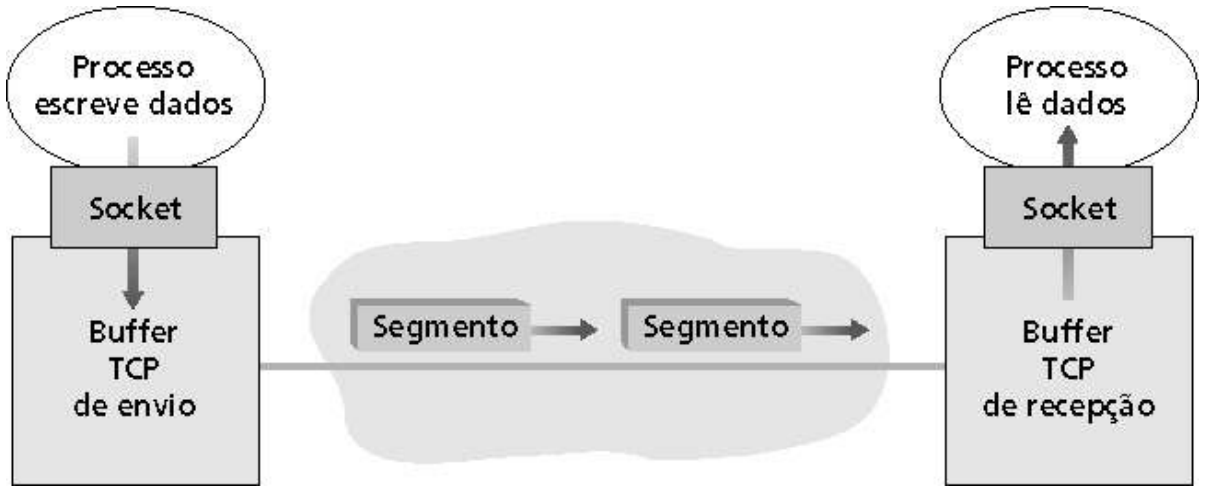
P.: Qual a relação entre o espaço de numeração seqüencial e o tamanho da janela?
-> Tamanho da janela deve ser menor ou igual à metade do tamanho do espaço de numeração sequencial.





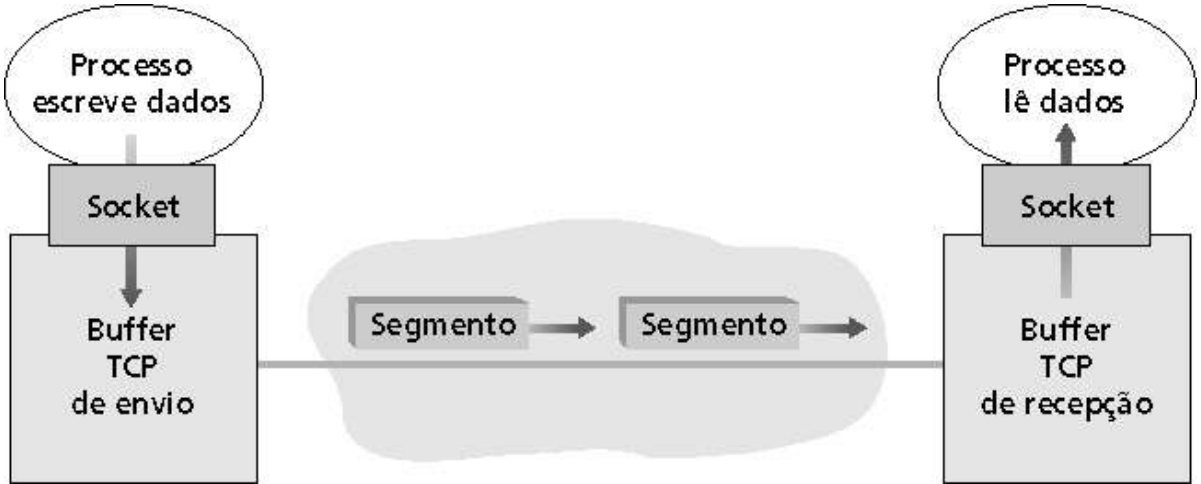
PROTOCOLO TCP

- Orientado à conexão
  - Apresentação (troca de mensagens de controle)
  - Inicia o estado do transmissor e do receptor antes da troca de dados
    - Buffers de transmissão e de recepção
- Ponto-a-ponto
  - Um transmissor, um receptor
- Dados full-duplex:
  - Transmissão bidirecional na mesma conexão



PROTOCOLO TCP

- Pipelined (Dutado)
  - Transporte confiável de pacotes via Acks
  - Controle de congestionamento e de fluxo definem tamanho da janela
- Fluxo de bytes sequencial
  - Não há inversão de ordem de mensagens
- Controle de fluxo:
  - Transmissor não esgota a capacidade do receptor
- Controle de congestionamento
  - Tenta evitar a sobrecarga da rede

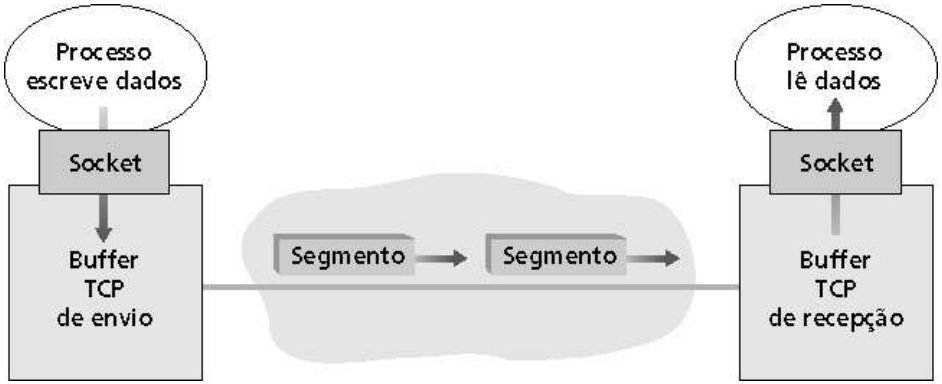


PROTOCOLO TCP

- Estabelecimento de conexão entre cliente e servidor
  - Cliente:
    - Socket clientSocket = new Socket ("nomeHost", numPorta)
    - Conexão é estabelecida
      - Cliente envia segmento TCP especial
      - Servidor responde com outro segmento especial
      - Cliente responde com outro segmento especial
        - » Carrega carga útil (dados da aplicação)
  - Conexão apresentação de três vias (3-way handshake)

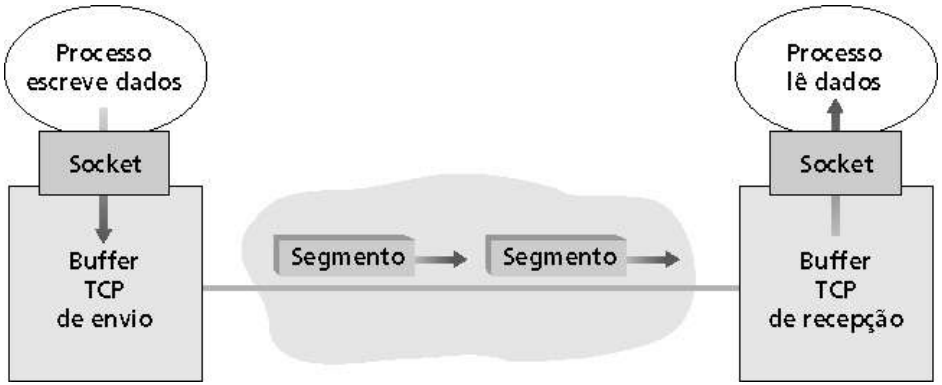
PROTOCOLO TCP

- Buffer de emissão e recepção
  - Processo Cliente envia uma string de dados através da porta do processo
  - String de dados é entregue ao processo TCP
    - Que envia a string para o buffer emissor TCP
    - De tempos-em-tempos pedaços de dados são extraídos do buffer
      - MSS: tamanho máximo do segmento - tamanho escolhido para evitar fragmentação no IP
    - TCP adiciona o cabeçalho formando o segmento TCP
    - Envia via rede para o buffer receptor TCP



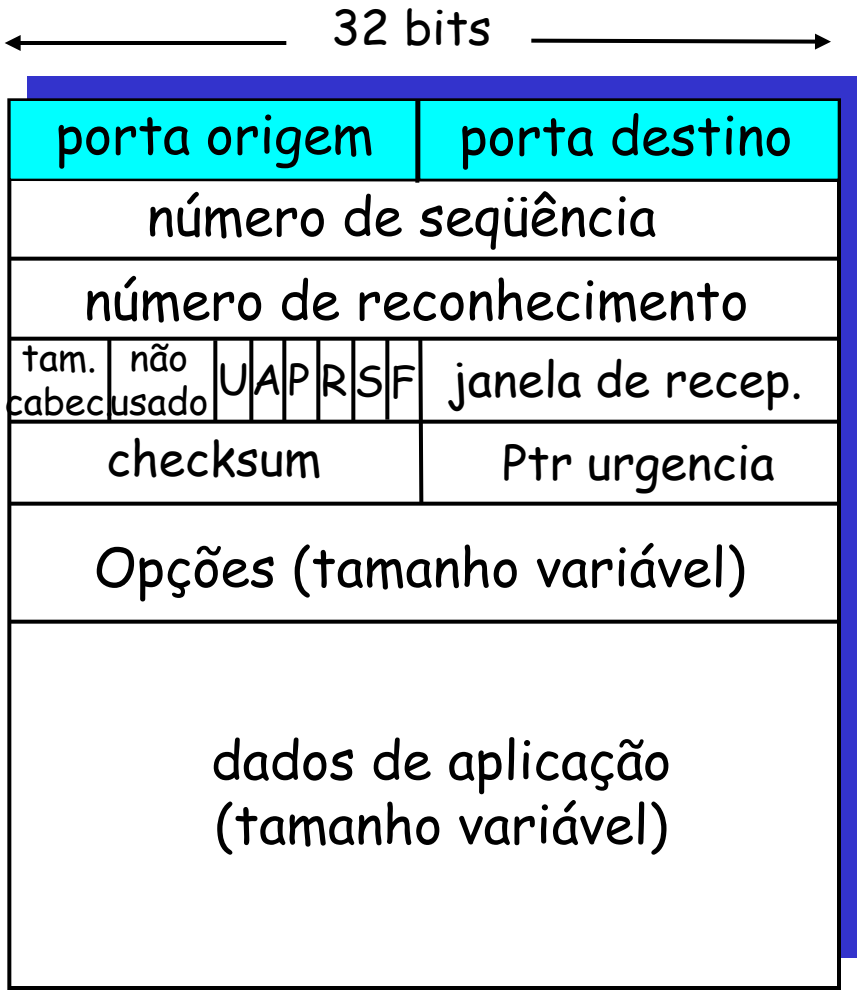
PROTOCOLO TCP

- Buffer de emissão e recepção
  - No host receptor
    - Segmento é colocado no buffer receptor TCP
    - Aplicação lê dados
  - Tanto no emissor como receptor existem buffers de emissão e recepção
    - Fluxo de dados bidirecional na mesma conexão



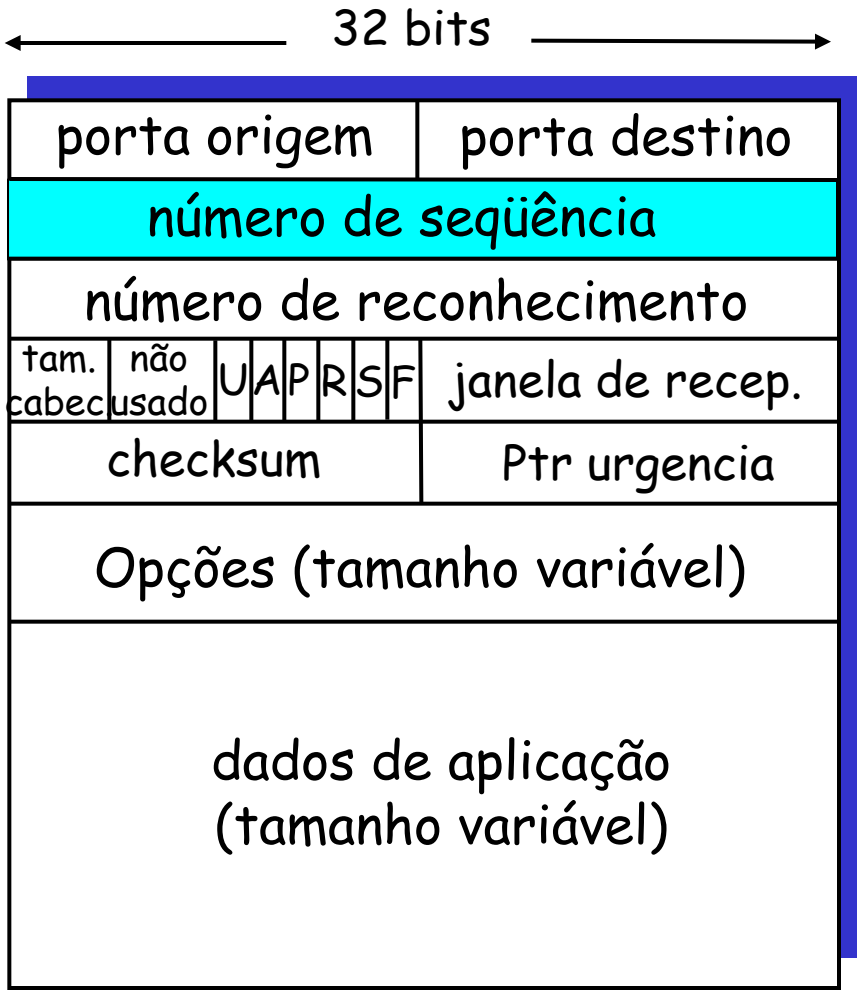
Estrutura do Segmento TCP

- Multiplexação/Demultiplexação



Estrutura do Segmento TCP

- Tratamento de segmentos



TCP: Número de seqüência

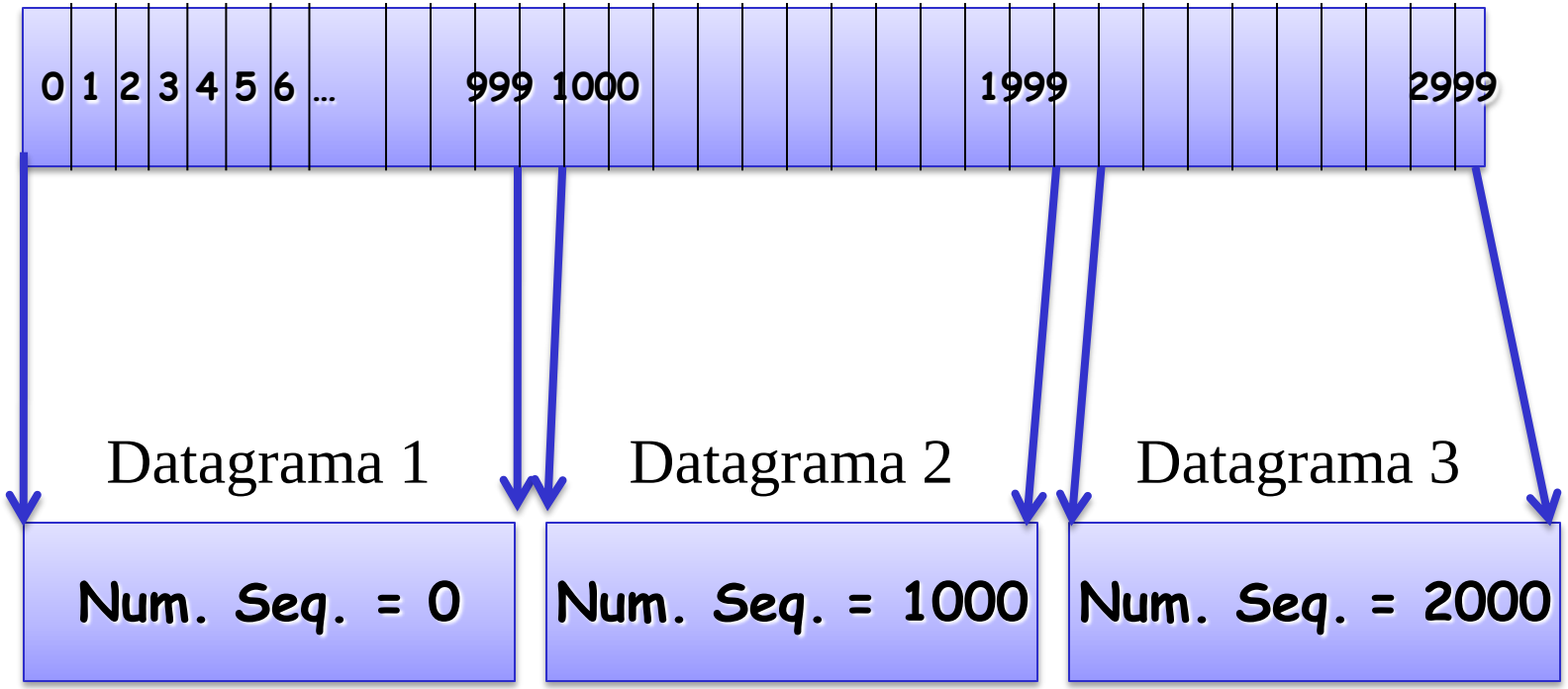
- Tratamento do Tamanho dos Segmentos
  - TCP deve manipular as diversidades de tamanhos das mensagens geradas pelas aplicações
  - Blocos de dados muito grandes
    - TCP deve fragmentá-los em unidades menores (segmentos) de modo a que os protocolos de nível inferior possam tratá-los
  - Blocos de dados pequenos
    - TCP "bufferiza" os segmentos para conduzi-los "juntos" num mesmo segmento TCP
    - Redução do "overhead" de transmissão

TCP: Número de seqüência

- Numeração dos segmentos
  - Conceitualmente cada byte da mensagem é associado a um número de seqüência
  - Número de seqüência do primeiro byte dos dados contidos em um segmento é transmitido junto com o segmento e é denominado *número de seqüência do segmento*
- Sequenciamento
  - Número de seqüência é usado para ordenar os segmentos que porventura tenham sido recebidos fora de ordem e para eliminar segmentos duplicados

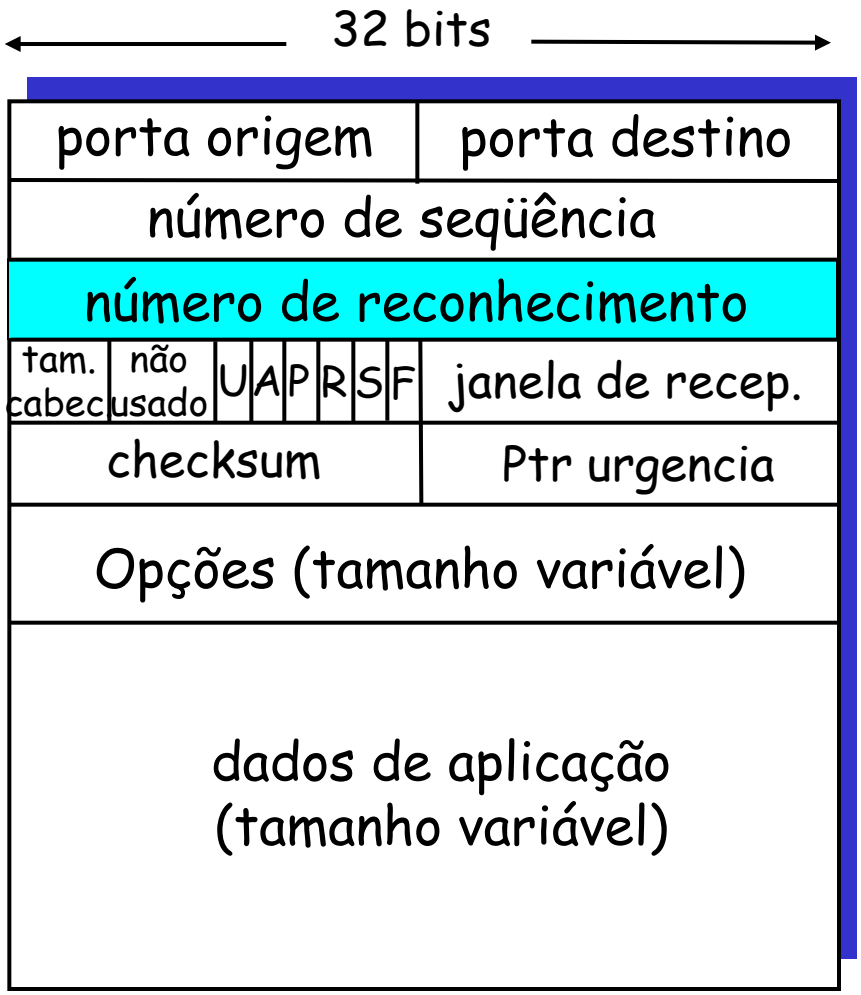
Protocolo TCP

- Sequenciamento
  - Exemplo:
    - Segmento de 3000 bytes fragmentado em 3 datagramas de 1000 bytes (MSS)



Estrutura do Segmento TCP

- Reconhecimento (controle de erro)
  - Número do próximo byte aguardado pelo receptor



Protocolo TCP

- Reconhecimento
  - Consiste do número de seqüência do próximo byte que a entidade TCP emissora espera receber do TCP receptor
  - Exemplo
    - Se número de reconhecimento X for transmitido no ACK
      - Indica que a entidade TCP receptora recebeu corretamente os bytes com número de seqüência menores que X
      - Ela espera receber o byte X na próxima mensagem
  - Segmentos carregam de carona (*piggybacking*) um reconhecimento

TCP: Reconhecimento

Número de seqüência:

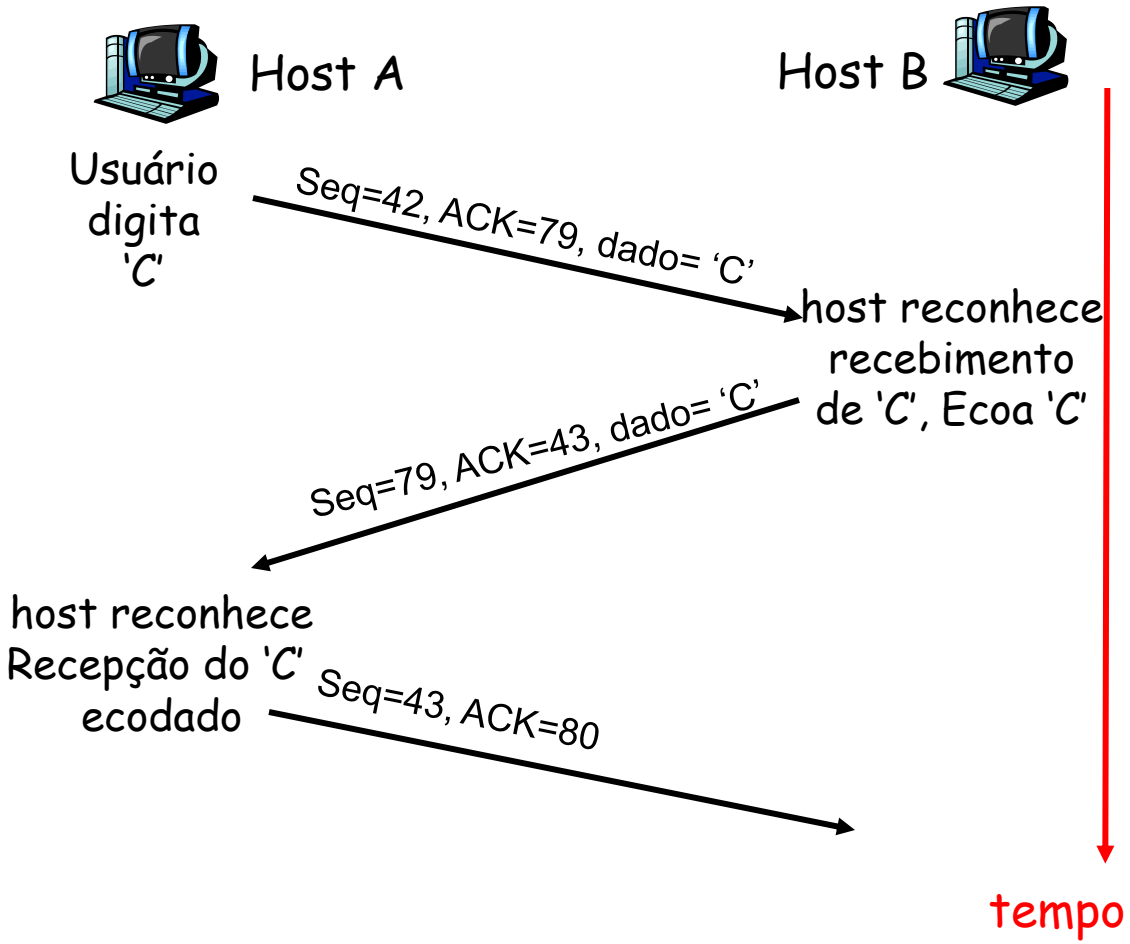
- Número do primeiro byte da string

Reconhecimento:

- Número de seqüência do próximo byte esperado

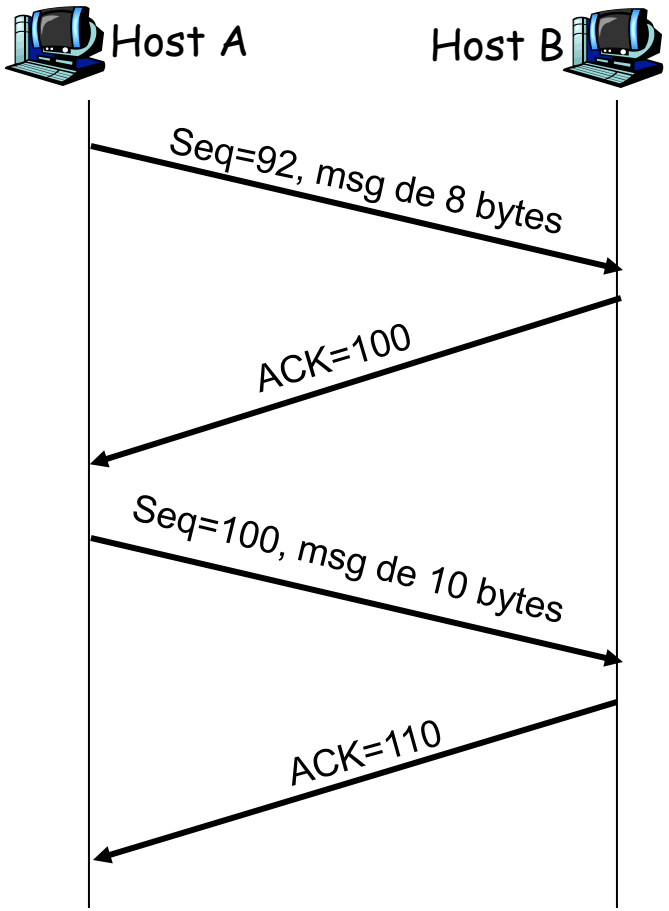
Sequenciamento:

- especificação do TCP não define
  - deixado ao implementador



Cenário telnet simples

TCP: Reconhecimento



Protocolo TCP

- Retransmissões
  - Quando uma entidade TCP transmite um segmento
    - Ela coloca uma cópia do segmento em uma fila de retransmissão e dispara um temporizador
    - Caso o reconhecimento do segmento é recebido
      - o segmento é retirado desta fila
    - Caso o reconhecimento não ocorra antes do temporizador expirar
      - Segmento é retransmitido

TCP: transferência de dados confiável

Evento: dados recebidos
da aplicação acima

Cria, envia segmento com seq=y
Lança temporização

transmissor simplificado, assumindo que não há controle de fluxo nem de congestionamento

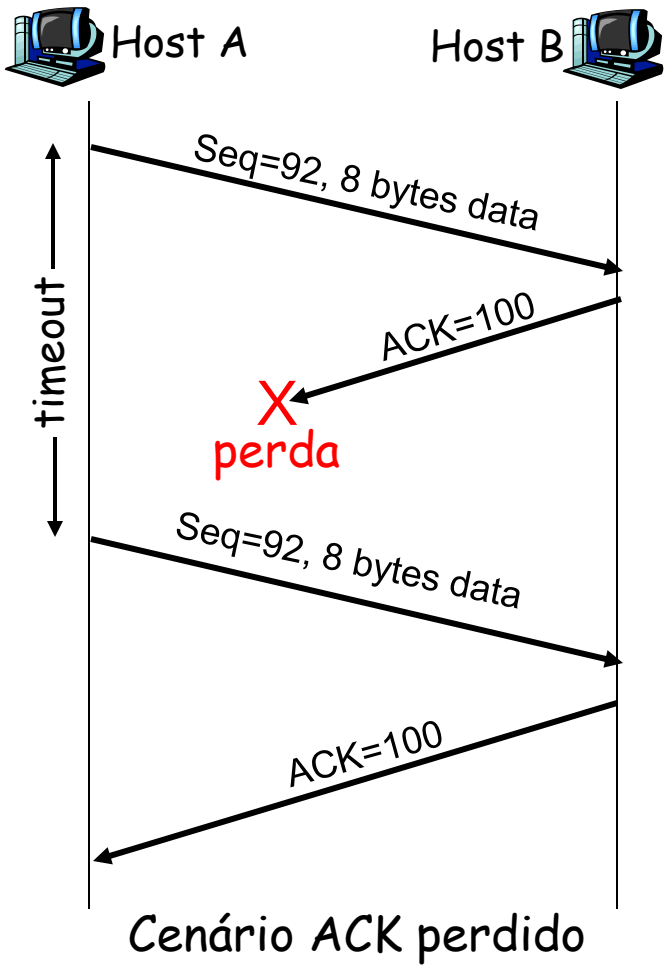
evento: temporização esgotada
para segmento com seq = y

retransmite segmento

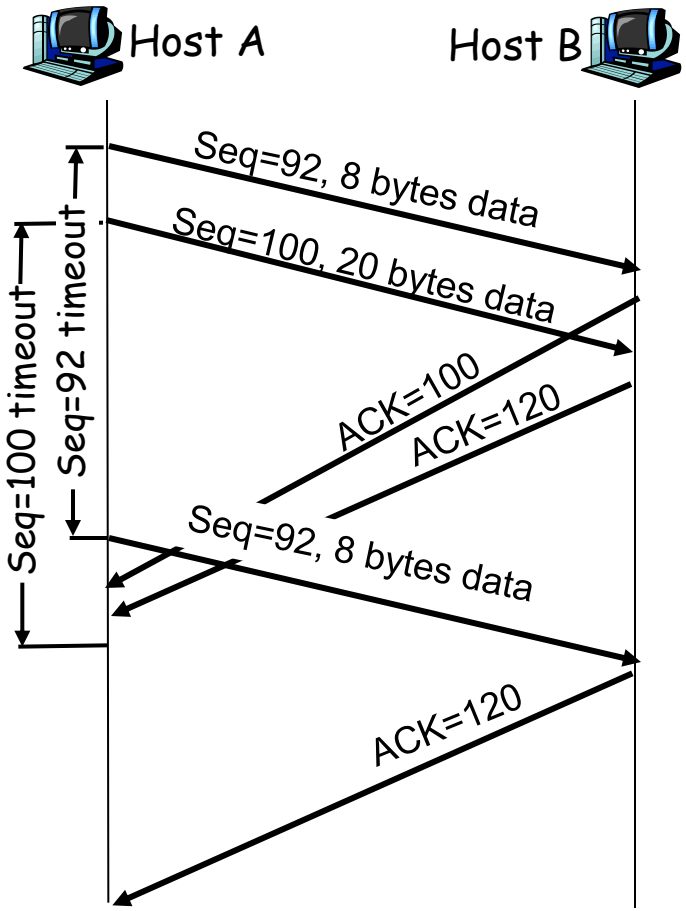
evento: ACK recebido,
com número de ACK > y

processamento do ACK

TCP: Cenários de retransmissão

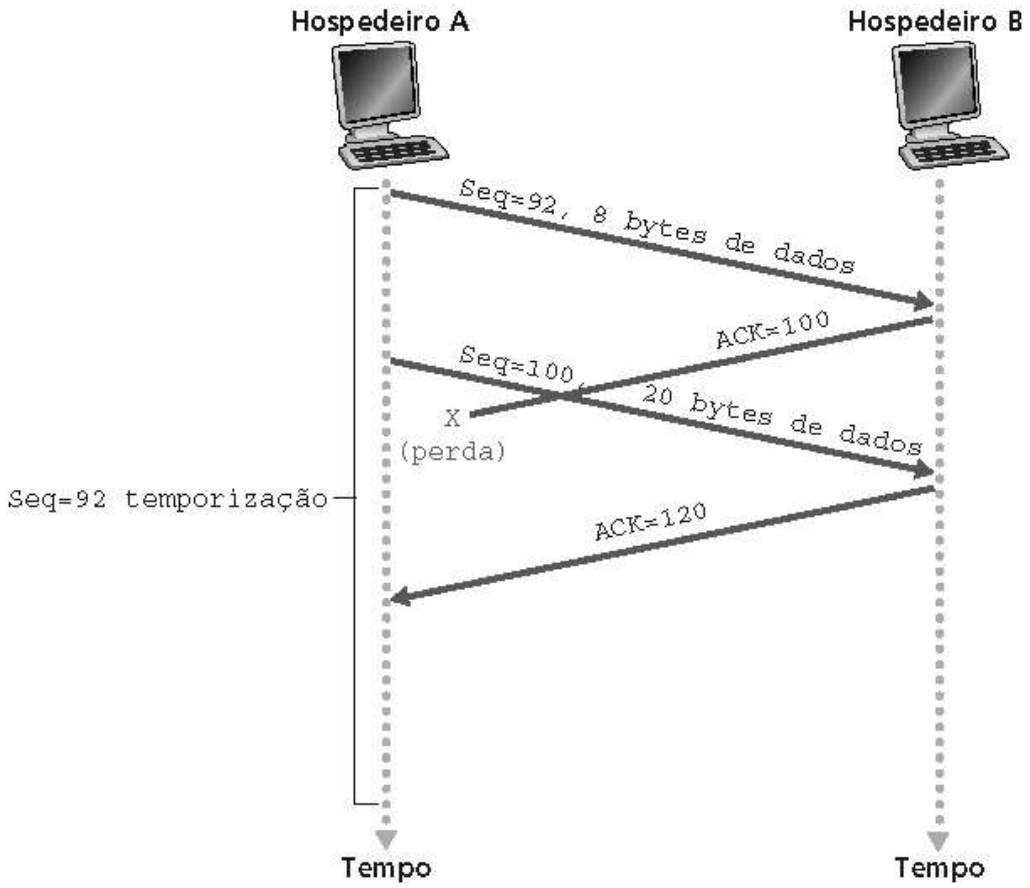


Cenário ACK perdido



Timeout prematuro,
ACKs acumulativo

TCP: Cenários de retransmissão



Cenário de ACK cumulativo

TCP Geração do Reconhecimento

[RFC 1122, RFC 2581]

Evento	Ação do receptor TCP
Chegada de segmento em ordem, sem gaps, segmentos anteriores já confirmados	ACK atrasado. Aguarda até 500ms pelo próximo segmento. Se o segmento não chegar, enviar ACK
Chegada de segmento em ordem, sem gaps, existe ACK atrasado	Enviar imediatamente um único ACK acumulativo
Chegada do segmento fora de ordem maior que o número de seq esperado (gap detectado)	Envia ACK duplicado, indicando o número de seq. do próximo byte esperado
Chegada de um segmento que preenche parcialmente ou complentamente um gap	ACK imediato se o segmento inicia na parte inicial mais baixa do gap

Número de Seqüência e de Reconhecimento

- Números iniciais são escolhidos aleatoriamente no estabelecimento da conexão
- Feito para minimizar a possibilidade que um segmento inválido seja confundido como válido
  - Que pertencente a uma conexão já terminada e chegue no receptor após estabelecida uma nova conexão entre os mesmos receptor e transmissor na mesma porta

TCP: Tempo de Resposta (RTT) e Temporização

P: como escolher valor do temporizador TCP?

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

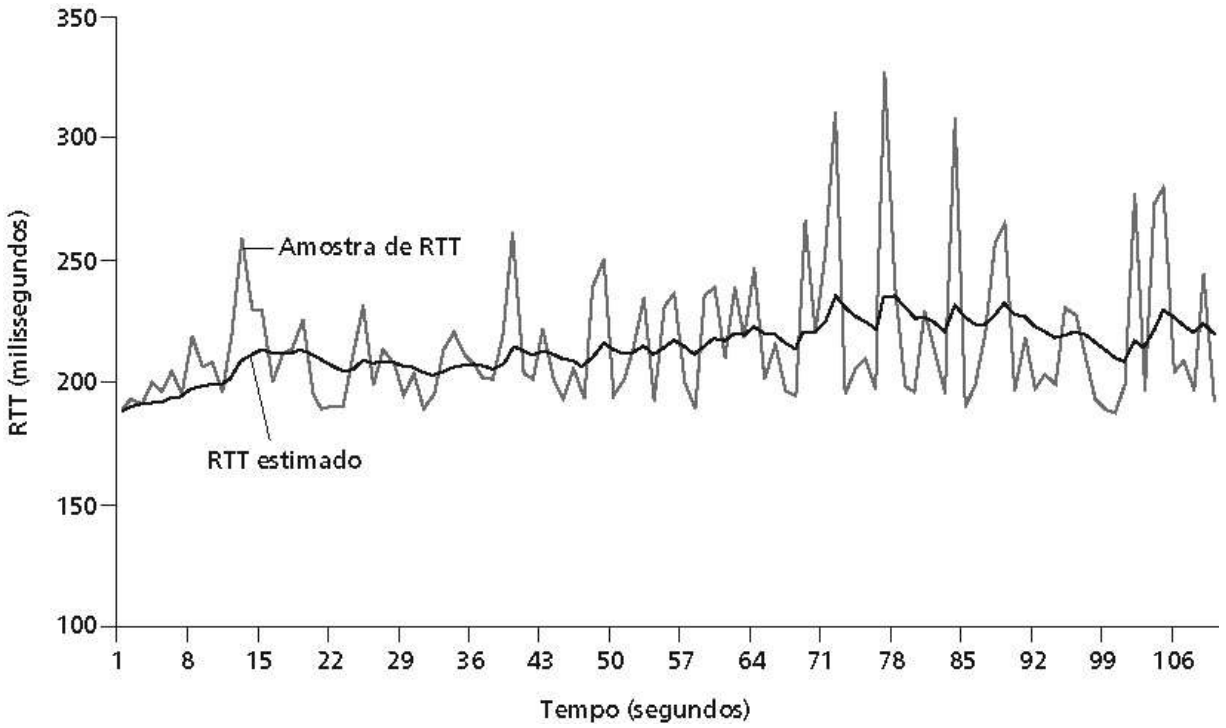
P: como estimar RTT?

- **SampleRTT**: tempo medido entre a transmissão do segmento e o recebimento do ACK correspondente
  - Cada segmento TCP terá seu próprio sampleRTT
  - Varia de segmento para segmento
  - Ignora retransmissões e segmentos reconhecidos de forma cumulativa
- **SampleRTT** vai variar devido a congestionamento e sistemas finais
  - TCP mantém média de SampleRTT (**EstimatedRTT**)

TCP: Tempo de Resposta (RTT) e Temporização

EstimatedRTT = (1-α)* EstimatedRTT + α*SampleRTT

- média corrente exponencialmente ponderada (MMEP)
- influência de cada amostra diminui exponencialmente com o tempo
- valor típico de α= 0,125 (isto é 1/8)



TCP: Tempo de Resposta (RTT) e Temporização

- Escolhendo o intervalo de temporização
  - Intervalo de temporização = EstimatedRTT mais uma "margem de segurança"
  - variação grande em EstimatedRTT
    -> margem de segurança maior
  - Desvio é uma estimativa de quando SampleRRT tipicamente se desvia de EstimatedRTT (MMPE)

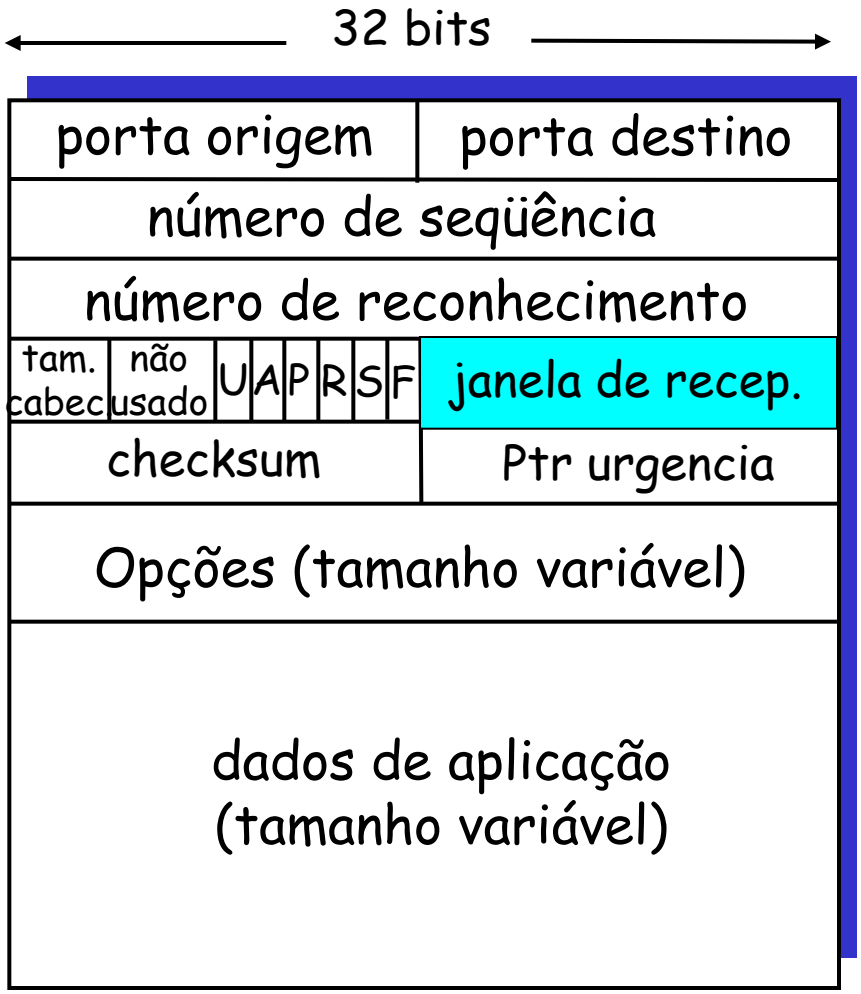
Temporização = RTT_estimado + 4*Desvio

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

β = 0.25 (valor típico)

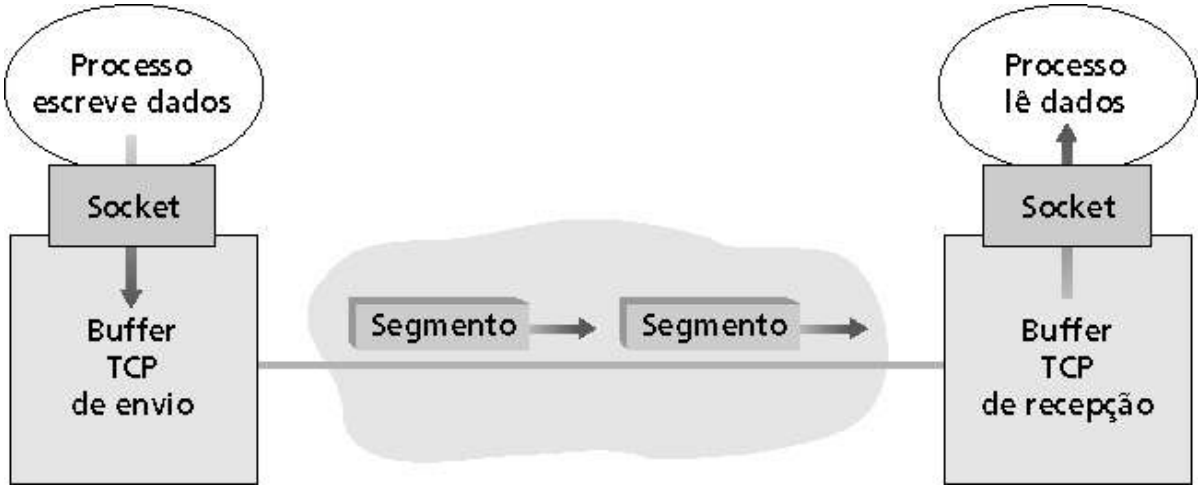
Estrutura do Segmento TCP

- Controle de fluxo



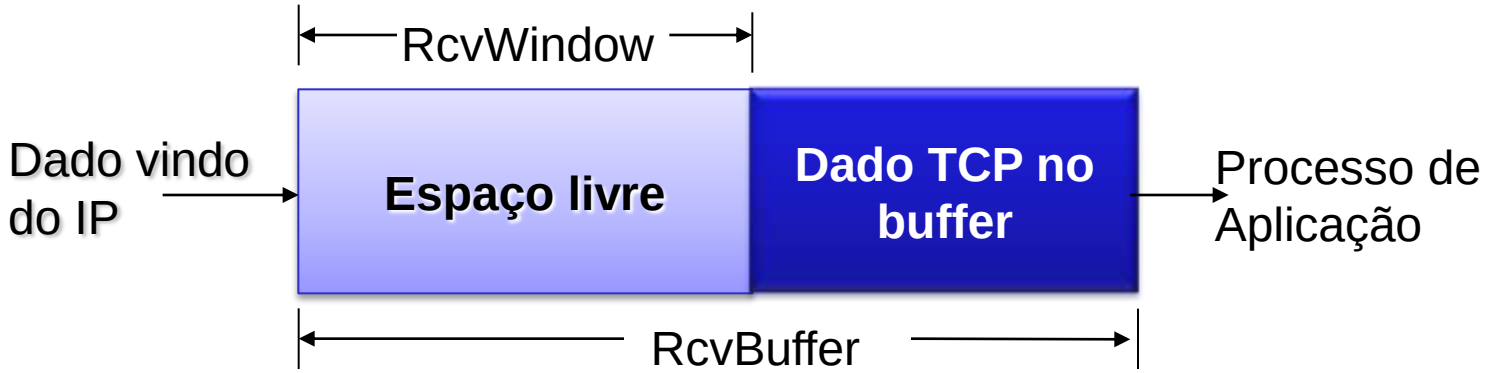
Protocolo TCP

- Controle de Fluxo
  - TCP provê mecanismo para que o transmissor possa determinar o volume de dados que o receptor pode acolher
    - Baseia-se no envio, junto com o reconhecimento, do número de octetos que o receptor tem condições de enviar contados a partir do último octeto da cadeia de dados recebido com sucesso
      - O tamanho da janela de recepção (RcvWindow)



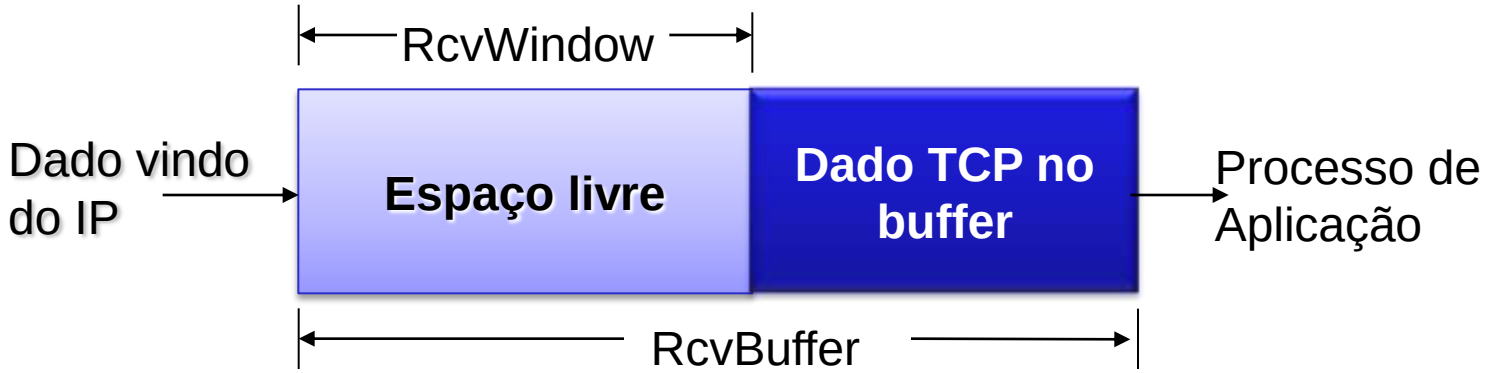
Protocolo TCP

- Controle de Fluxo
  - Transmissor atualiza sua janela de transmissão
    - Com base na janela de recepção
    - Calcula o número de octetos que pode enviar antes de receber outra liberação



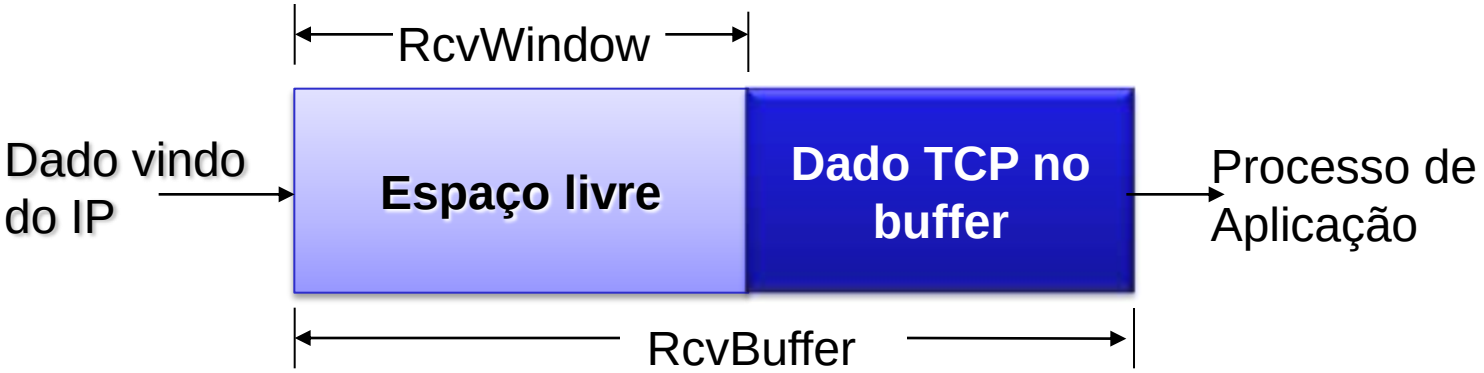
Protocolo TCP

- Processos de aplicação podem ser lentos para ler o buffer
- Controle de fluxo
  - Transmissor não deve esgotar os buffers de recepção enviando dados rápido demais
- Serviço de *speed-matching*: encontra a taxa de envio adequada à taxa de vazão da aplicação receptora



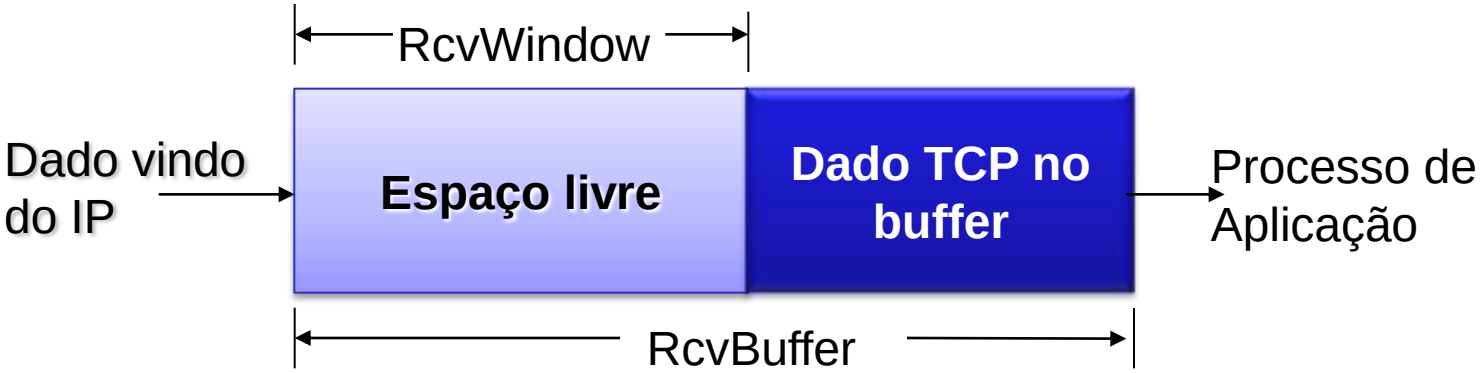
Protocolo TCP

- Controle de Fluxo
  - Receptor informa a área disponível incluindo valor RcvWindow nos segmentos
  - Transmissor limita os dados não confirmados ao RcvWindow
    - Garantia contra overflow no buffer do receptor



Protocolo TCP

- Controle de Fluxo
  - Receptor informa a área disponível incluindo valor RcvWindow nos segmentos
  - Transmissor limita os dados não confirmados ao RcvWindow
    - Garantia contra overflow no buffer do receptor

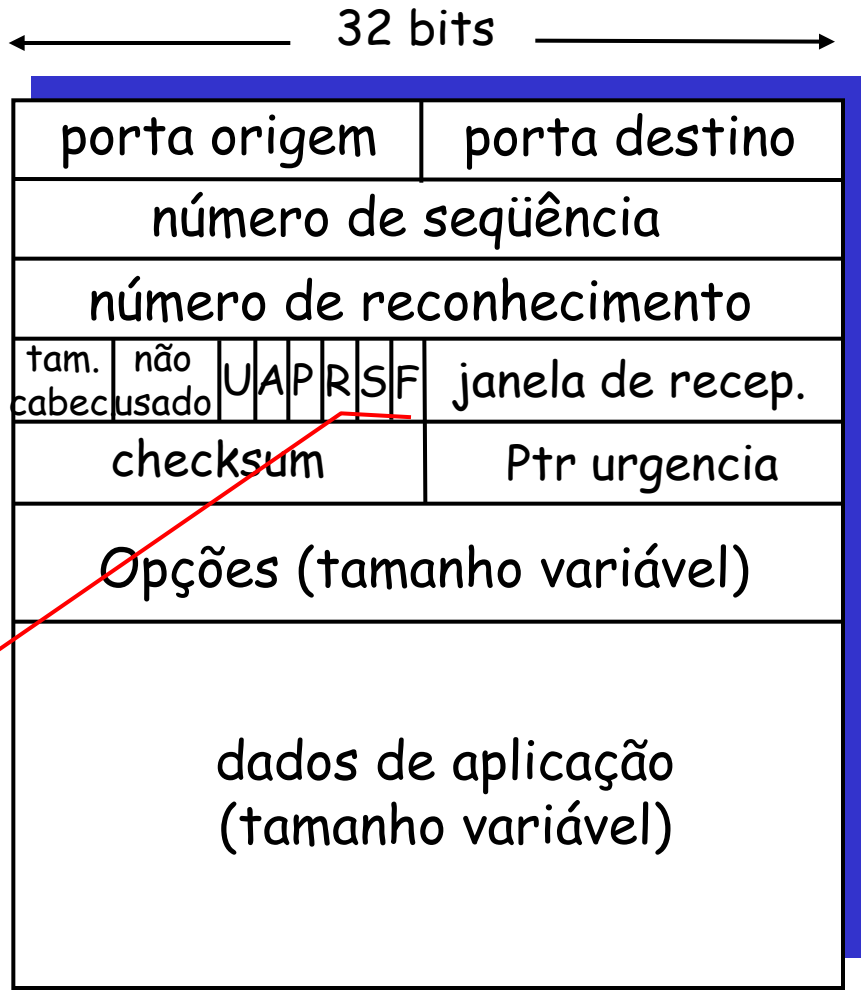


Gerenciamento da conexão TCP

- Emissor e Receptor TCP estabelecem uma conexão antes da troca de segmentos
- Inicializa variáveis TCP:
  - Números de seqüência
  - Buffers, informações de controle de fluxo (e.g. RcvWindow)
  - Etc.
- Cliente: Inicializador da conexão
  - Socket clientSocket = new Socket("nomeHost","numPorta");
- Servidor: contactado pelo cliente
  - Socket connectionSocket = welcomeSocket.accept();
- Usa o campo flags do TCP

Estrutura do Segmento TCP

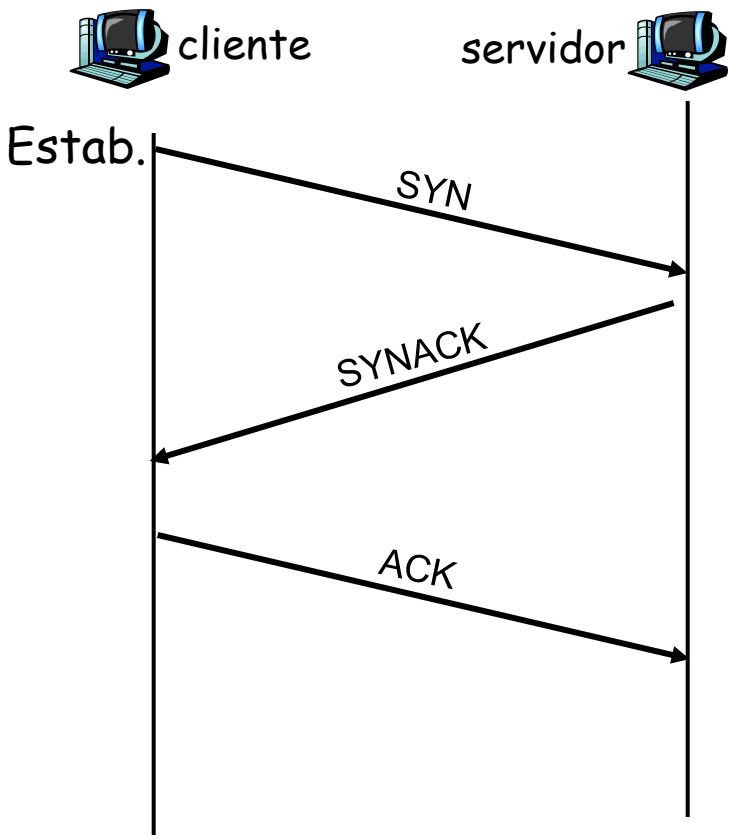
- Estabelecimento de conexão



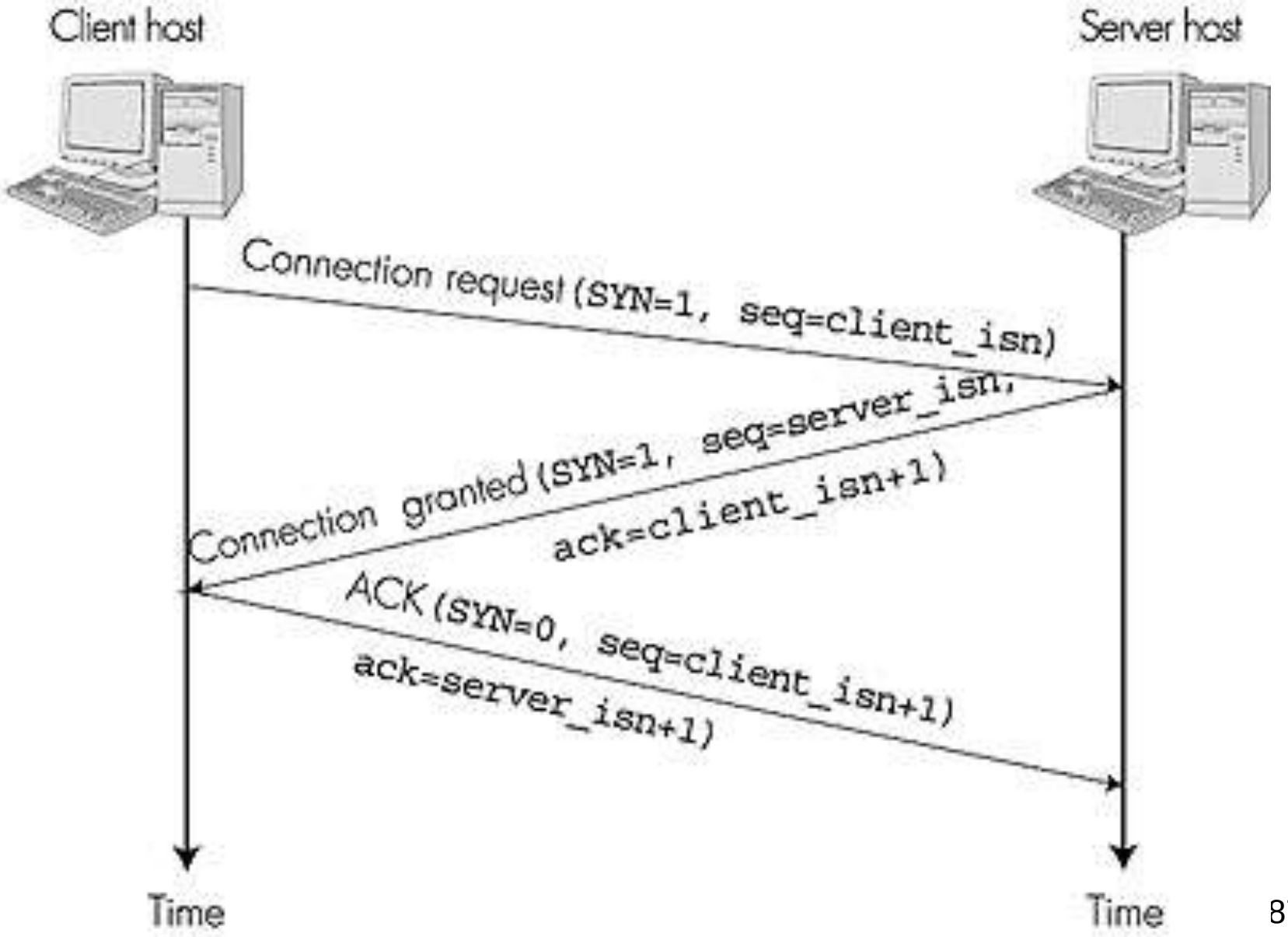
RST, SYN, FIN: Estabelecimento de conexão (comandos de estabelecimento e fechamento)

Gerenciamento da conexão TCP

- Algoritmo three-way handshake
  - **Passo 1**: cliente envia segmento de controle TCP SYN (setado a 1) para o servidor
    - Especifica o número inicial de seqüência (c_nmseq)
  - **Passo 2**: servidor recebe SYN, responde com o segmento de controle SYNACK
    - Aloca buffers e inicia variáveis da conexão
    - confirma SYN recebido
      - SYN bit é setado a 1
      - Campo ack do segmento TCP é setado com c_nmseq+1
      - Campo número de seqüência é setado com o número de seqüência inicial do servidor s_nmseq
  - **Passo 3**: Cliente recebe SYN ACK
    - Cliente aloca buffers e variáveis para a conexão
    - Cliente envia ao servidor outro segmento confirmando o SYNACK
      - O cliente coloca o valor s_nmseq+1 no campo ACK do segmento TCP e o bit SYN é setado a 0



Gerenciamento da conexão TCP



Gerenciamento da conexão (cont.)

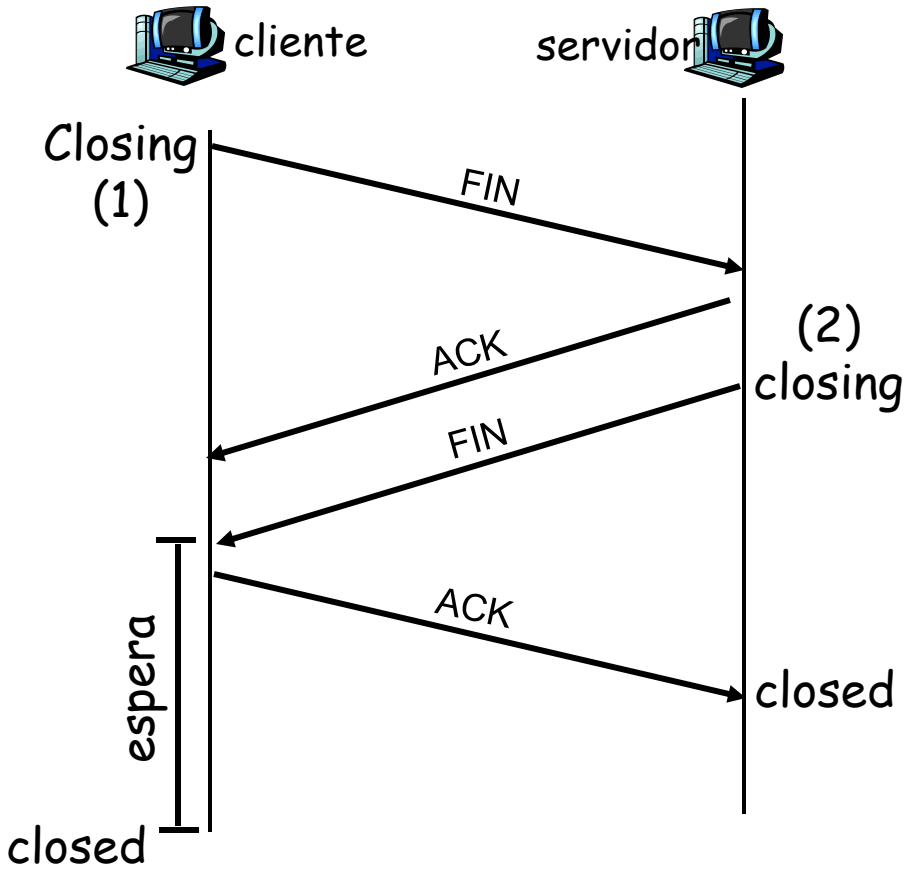
Fechando uma conexão:

Cliente fecha socket:

```
clientSocket.close();
```

Passo 1: cliente envia segmento de controle TCP FIN para o servidor

Passo 2: servidor recebe FIN, responde com ACK e envia FIN.

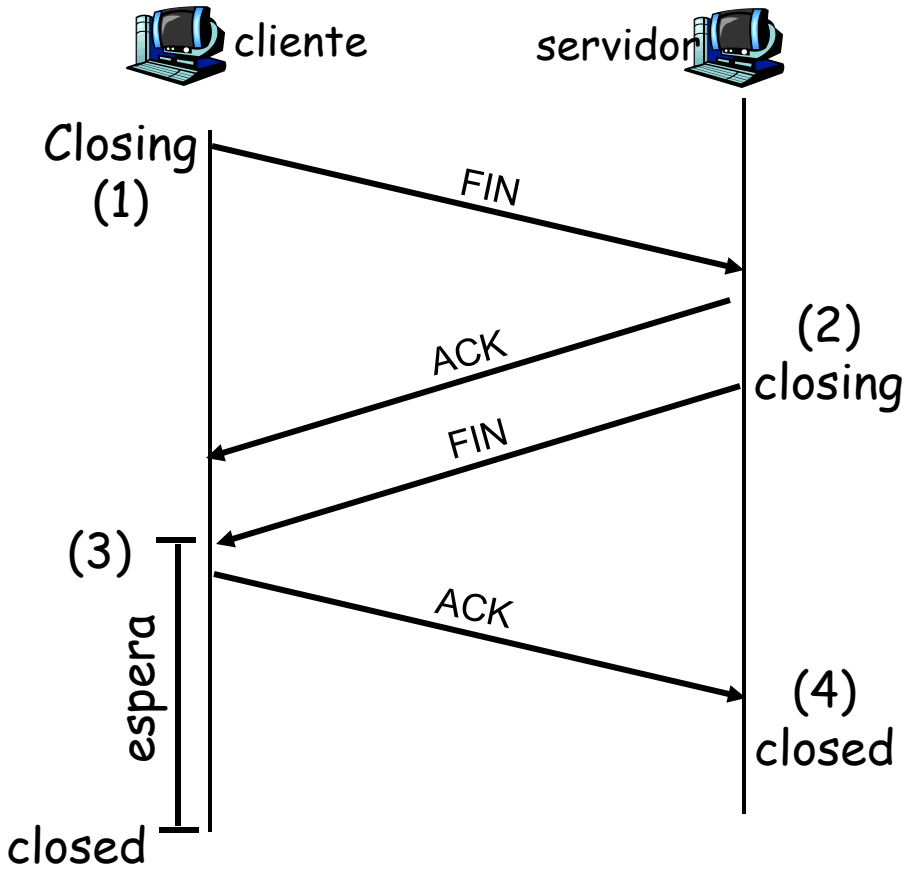


Gerenciamento da conexão (cont.)

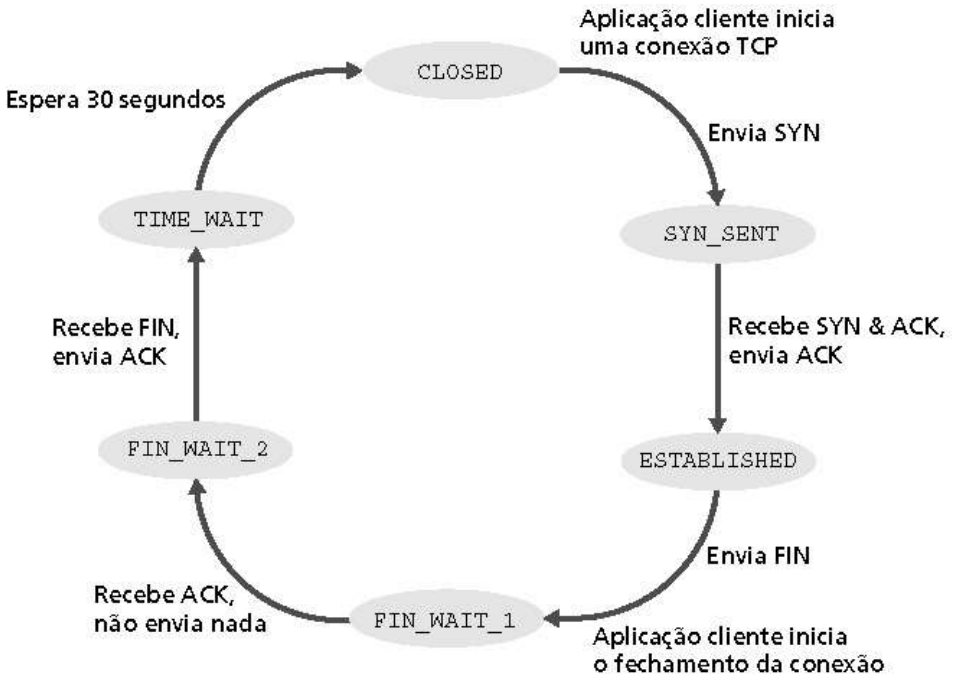
Passo 3: cliente recebe FIN, responde com ACK.

- Inicia "espera"

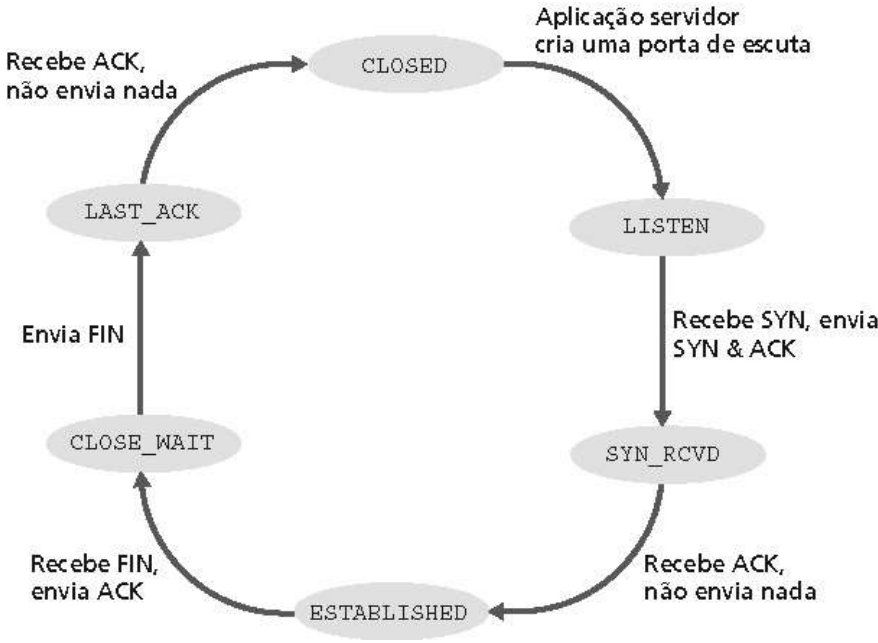
Passo 4: receptor recebe ACK. Conexão fechada.



Gerenciamento da conexão (cont.)



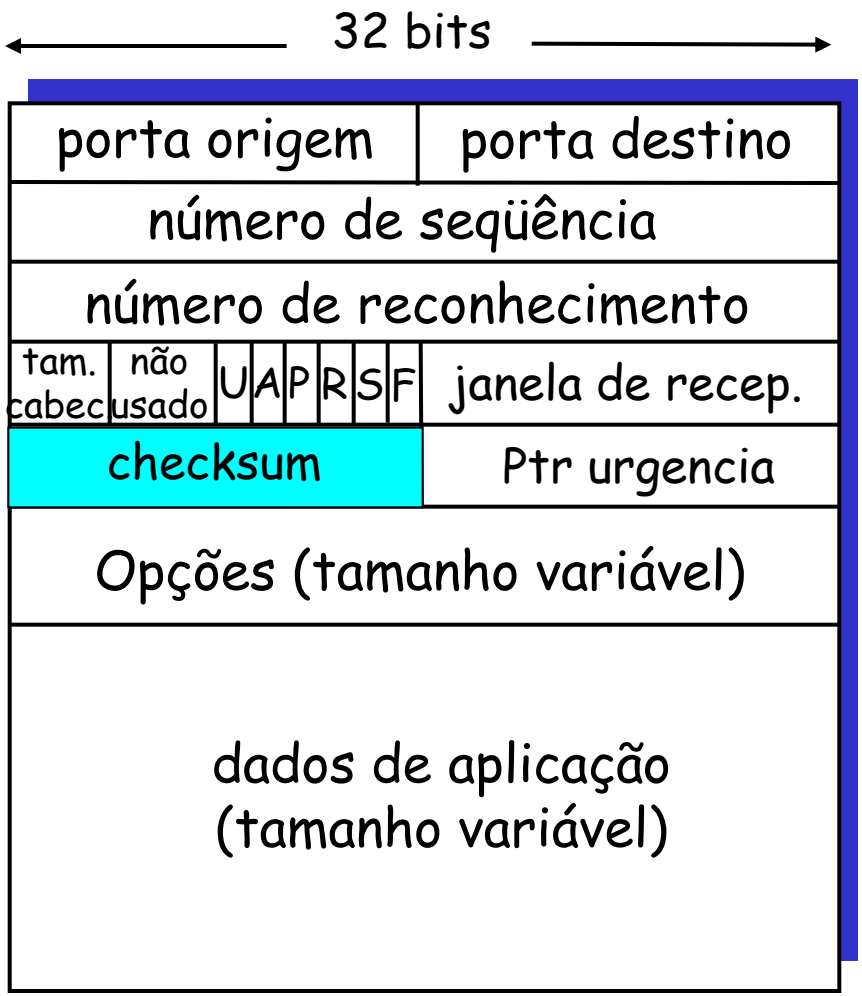
Estados do cliente



Estados do servidor

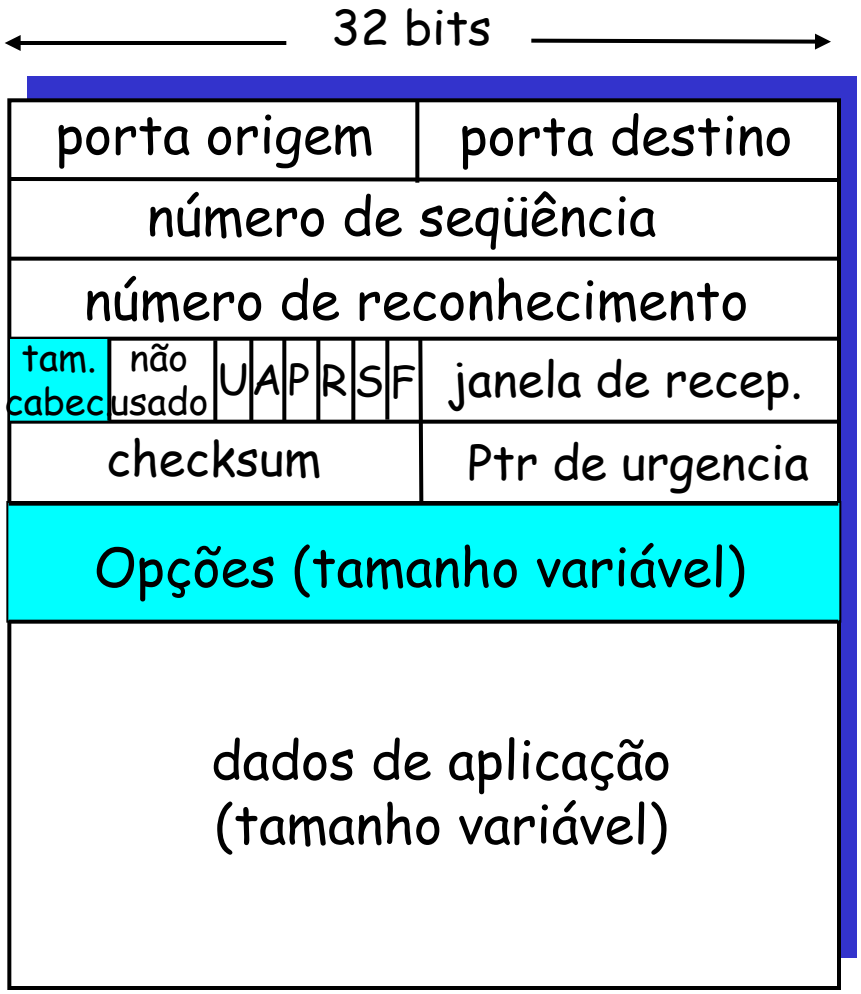
Estrutura do Segmento TCP

- Tratamento de erros
  - é adicionado um checksum a cada segmento transmitidos
    - Receptor faz uma verificação e os segmentos danificados são descartados



Estrutura do Segmento TCP

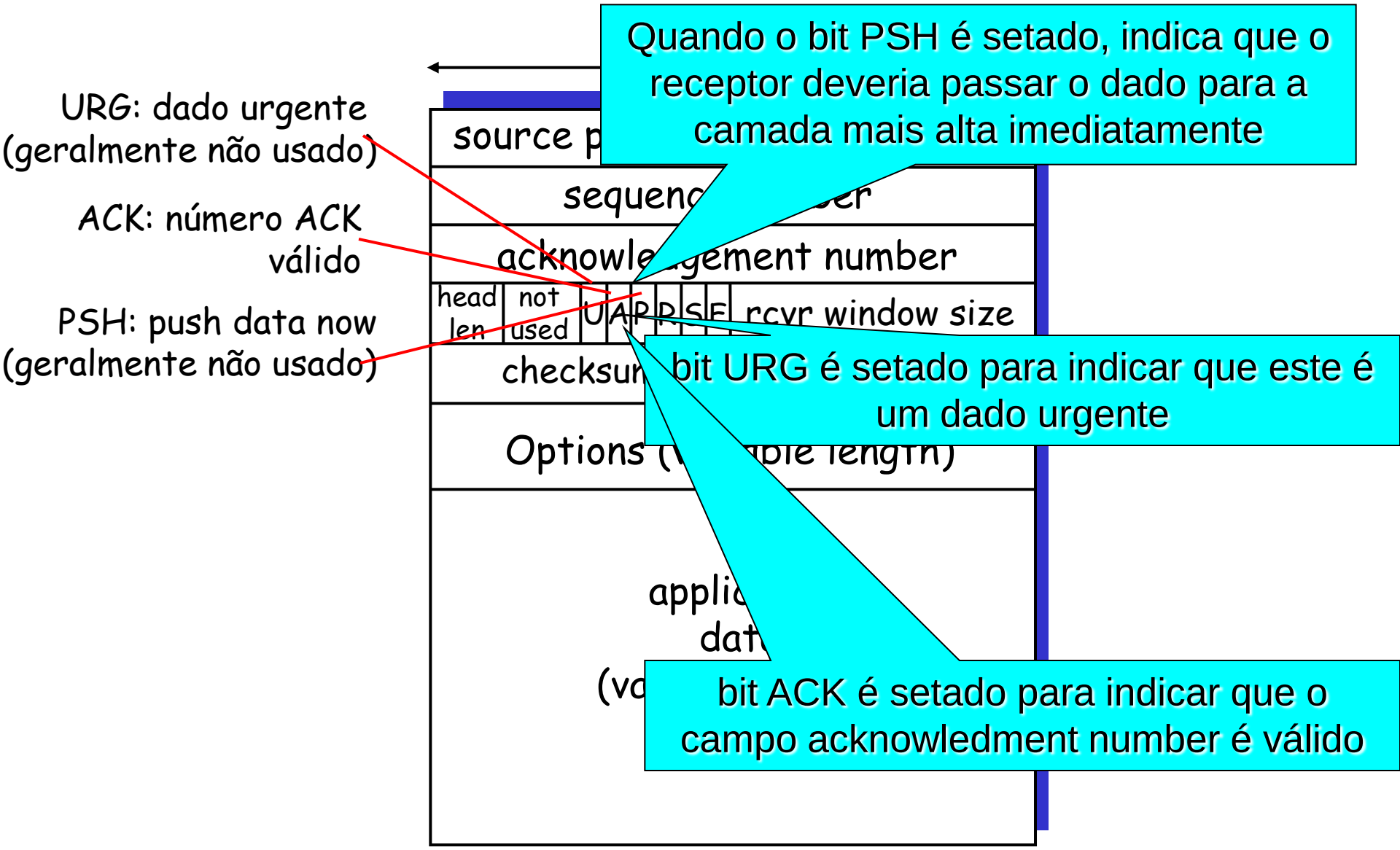
- Demais Campos



Demais Campos

- Campo length (4 bits)
  - Especifica o tamanho do cabeçalho TCP em palavras de 32 bits
  - Cabeçalho TCP pode ter um tamanho variável devido ao campo TCP options
    - geralmente vazio, assim o tamanho do cabeçalho TCP é de 20 bytes
- Campo options
  - Usado quando o emissor e receptor negociam um tamanho máximo de segmento (MSS) ou como um fator de escalamento de janela para uso em redes de alta velocidade
  - Uma opção de timestamping também é definida
- Ponteiro de urgência
  - Aponta para o último byte de dados urgente do pacote
- Campo pading
  - Usado quando necessário para completar tamanho de segmento múltiplo de 32 bits

Estrutura do segmento TCP



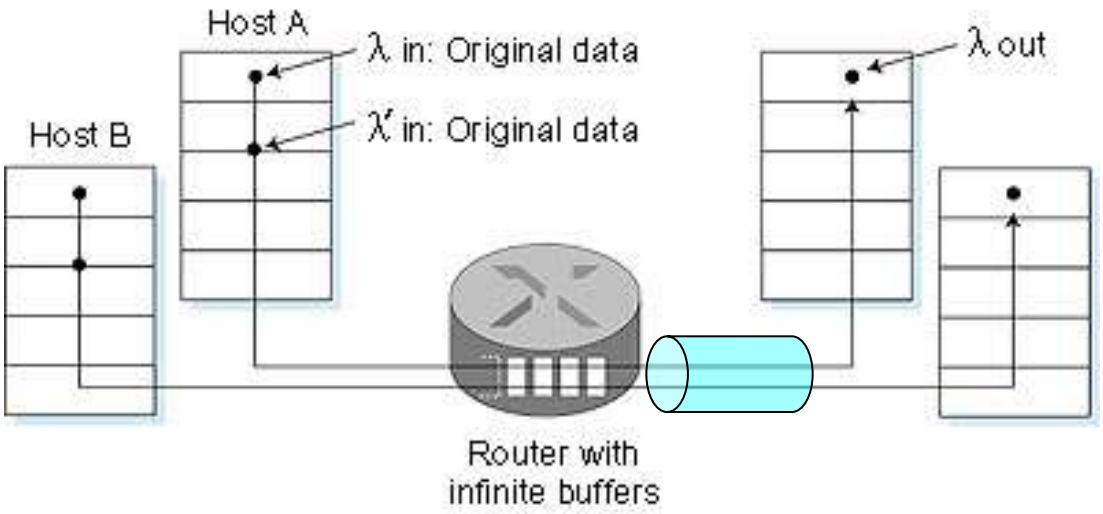
Controle de congestionamento

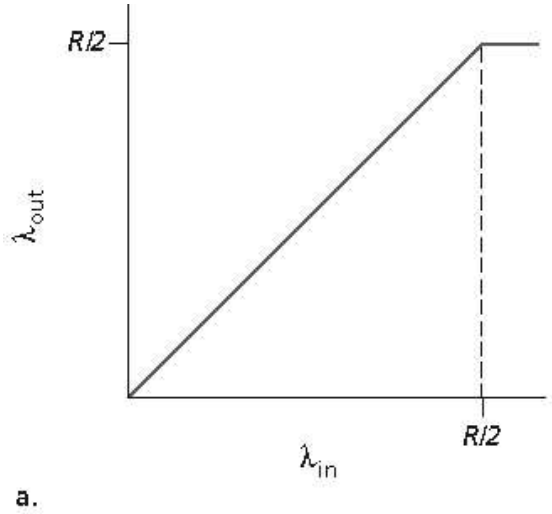
Congestionamento:

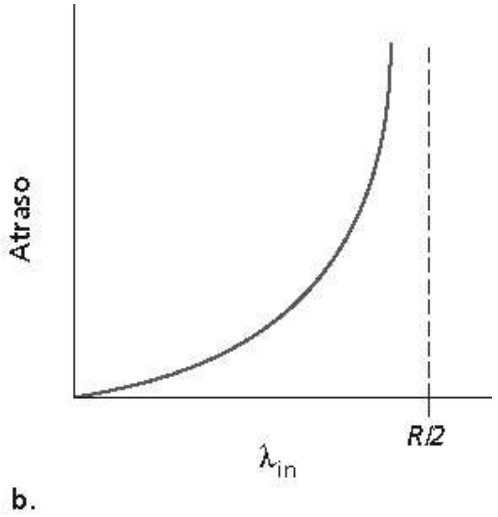
- Informalmente: "Excessivo número de fontes enviando grande quantidade de dados mais rápido que a rede possa manipular"
- Manifestações:
  - Pacotes perdidos (overflow dos buffers nos roteadores)
  - Grandes atrasos (enfileiramento nos buffers dos roteadores)
- Um grande problema de rede!

Causas/Custos do Congestionamento: Cenário 1

- Dois emissores, dois receptores
- Um roteador com buffers infinitos
- λ – Vazão em bps
- Dado original = dado enviado uma única vez
  - Sem retransmissão
- R é a capacidade dos enlaces do roteador





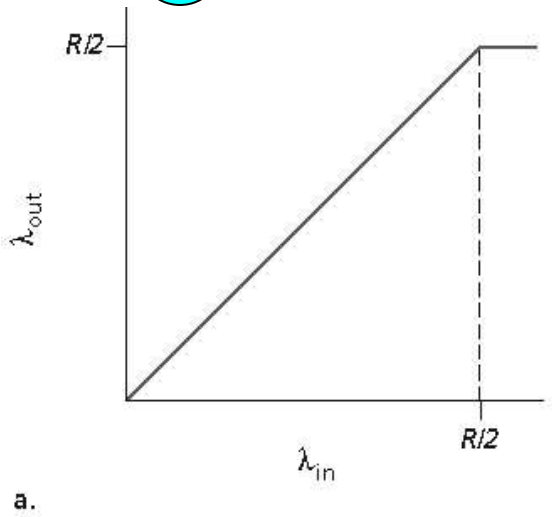


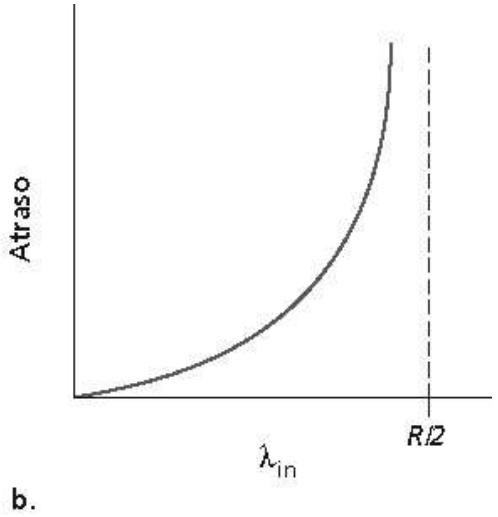
- Máxima vazão obtida
- Longo atraso quando congestionado

Causas/Custos do Congestionamento: Cenário 1

- D
- U
- λ
- C
- R

Mesmo neste cenário idealizado, constata-se um custo de uma rede congestionada – grandes atrasos de enfileiramento são verificados quando a taxa de chegada de pacotes é próxima da capacidade do enlace

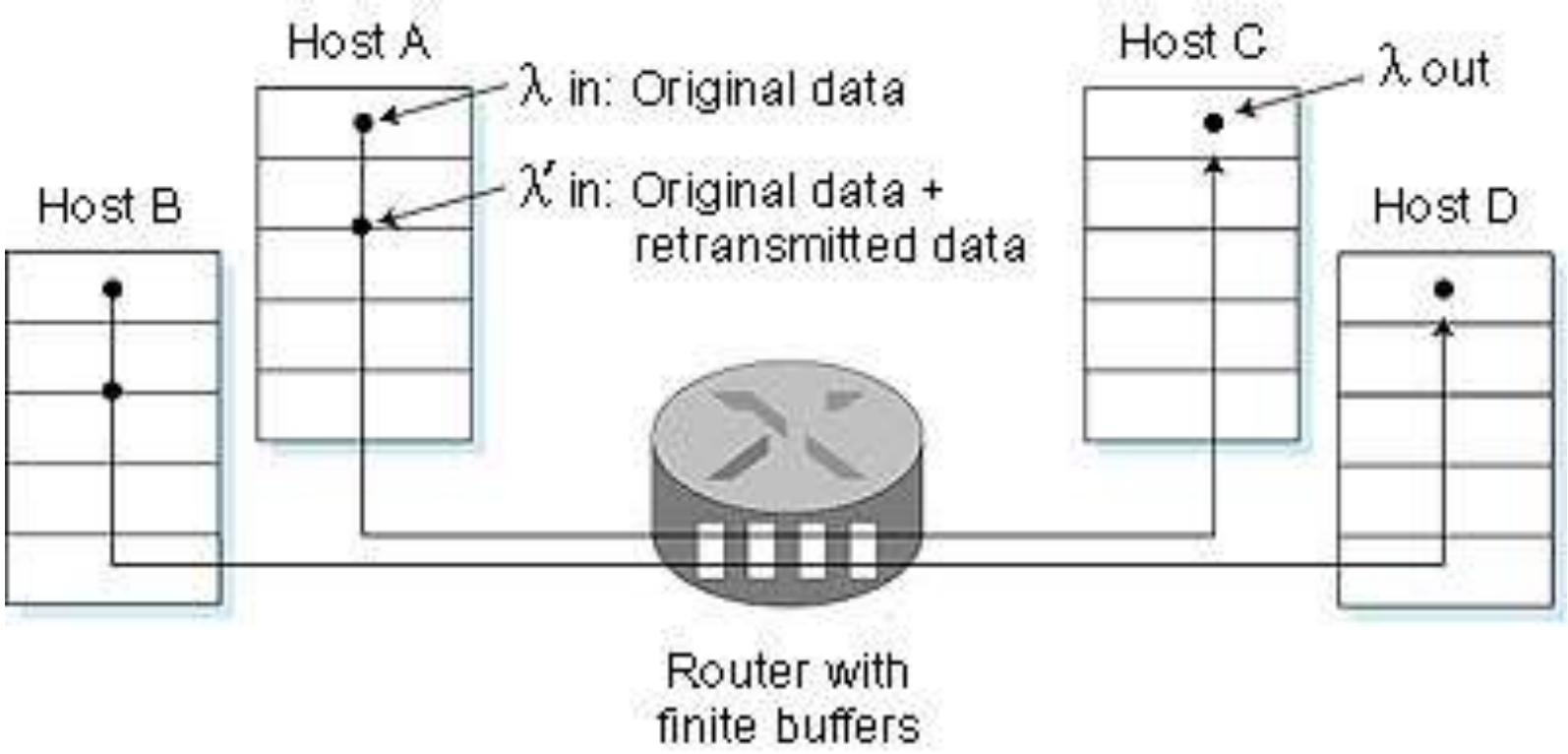




- Máxima vazão obtida
- Longo atraso quando congestionado

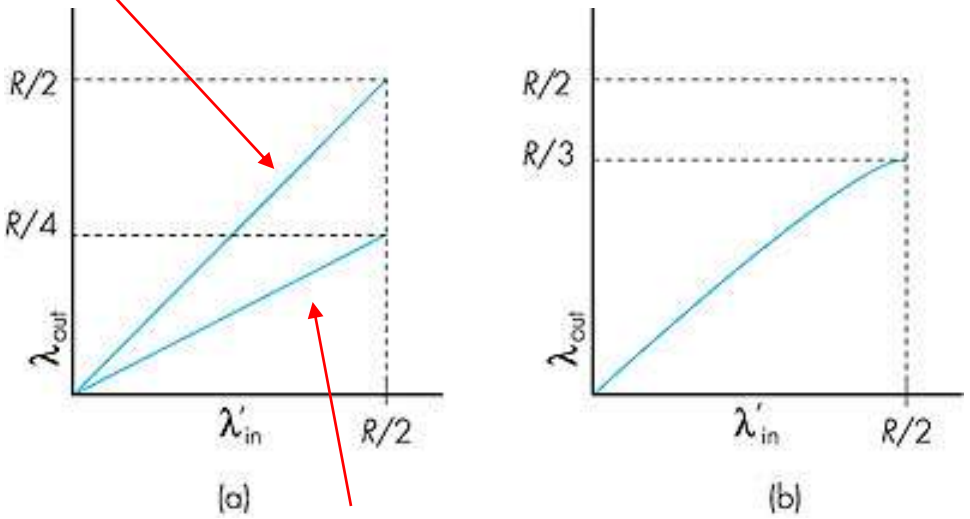
Causas/Custos do Congestionamento: Cenário 2

- Um roteador, buffers finitos
- Emissor retransmite pacotes perdidos



Causas/custos de congestionamento: cenário 2

Sem perdas ($\lambda_{in} = \lambda'_{in}$): Host só envia pacote quando houver um buffer livre no roteador.



Retransmissões apenas quando o pacote tiver realmente se perdido.

Cada pacote é transmitido duas vezes. Retransmissão de pacotes atrasados (mas não perdidos)

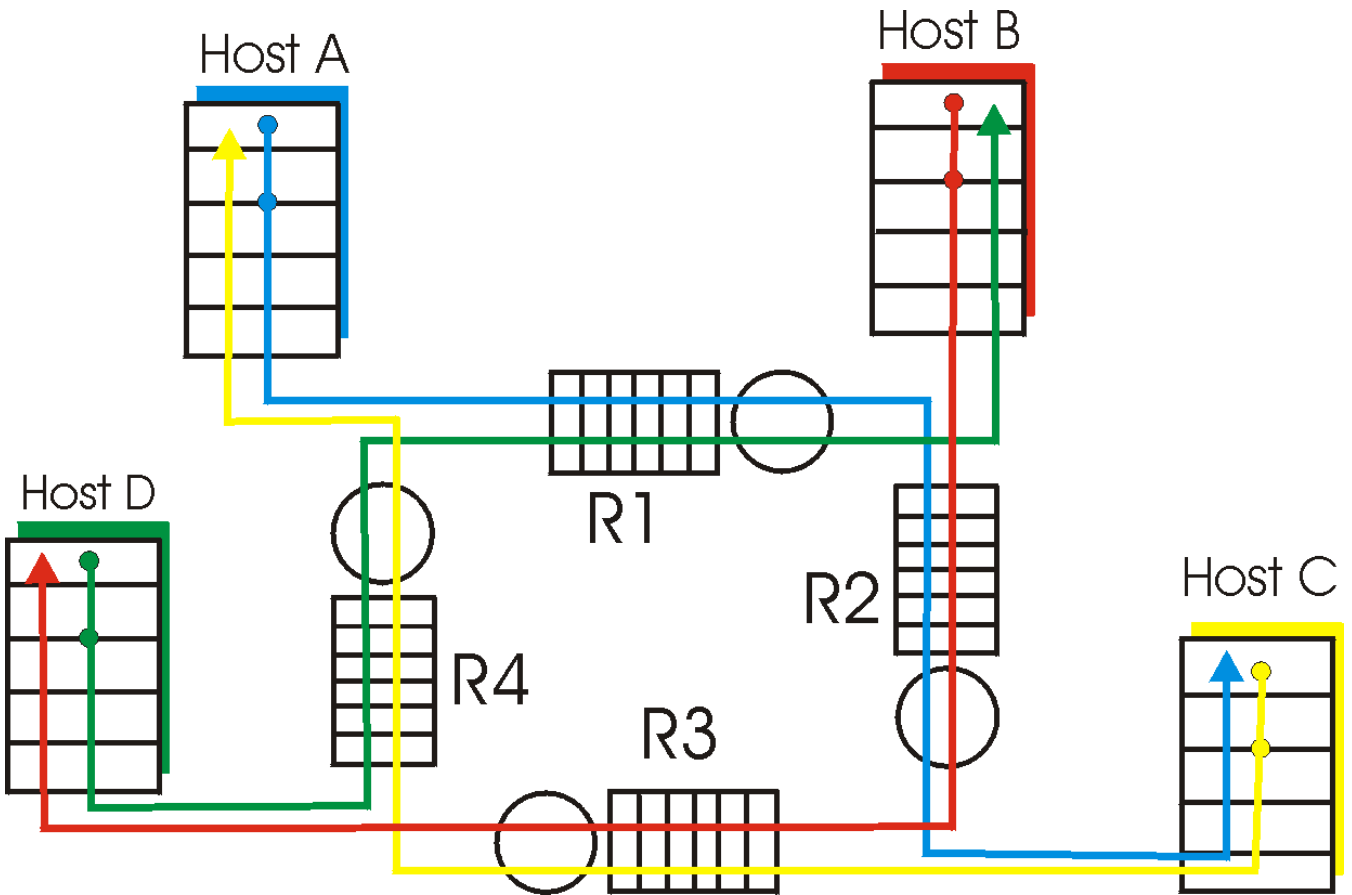
"custos" de congestionamento:

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

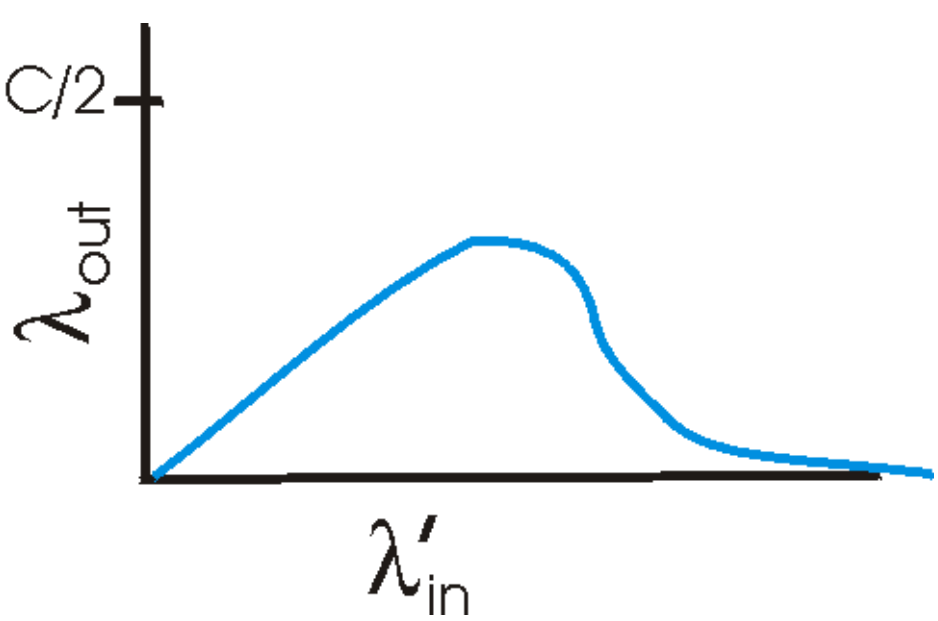
Causas/Custos do Congestionamento: Cenário 3

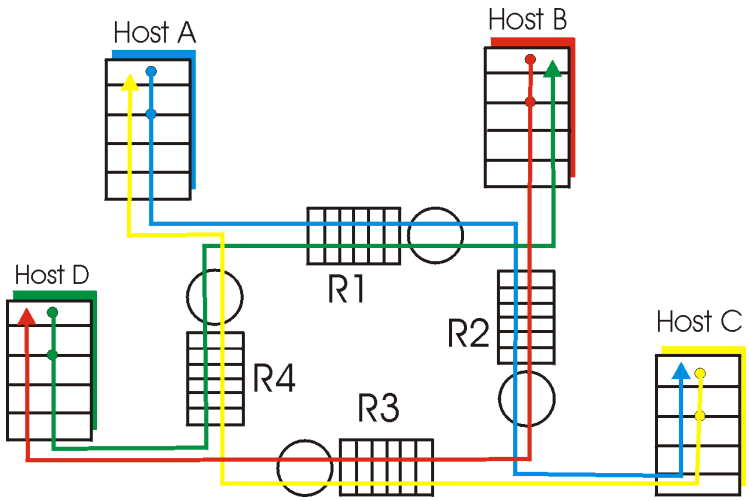
- Quatro emissores
- Vários caminhos
- timeout/retransmissão

Q: O que ocorre quando λ_{in} e λ'_{in} aumentam ?



Causas/Custos do Congestionamento: Cenário 3



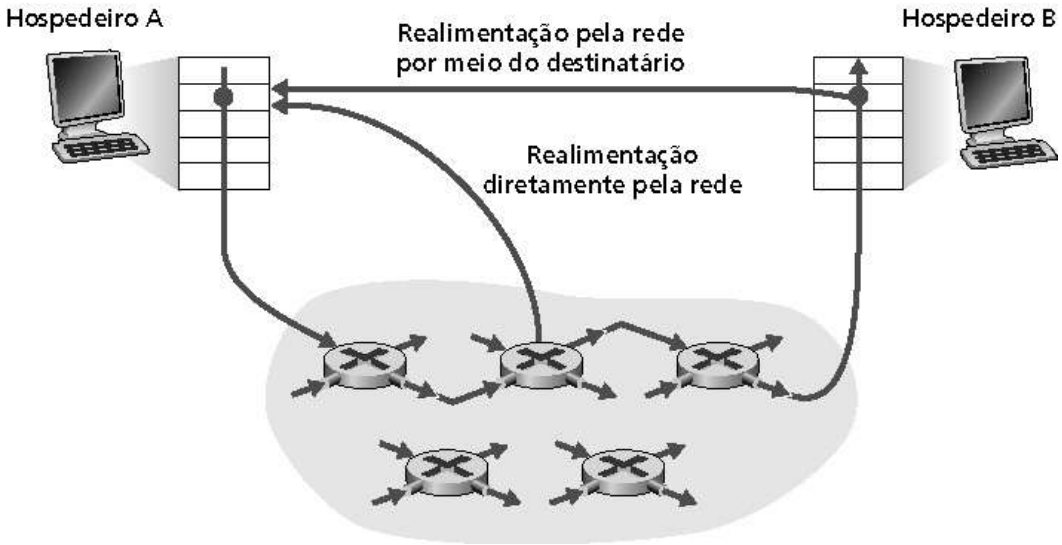


Outro custo do congestionamento

- Quando um pacote é cortado em um caminho, a capacidade de transmissão que foi usada em cada roteador de upstream para encaminhar o pacote para o ponto em que o pacote foi descartado é um desperdício a nível de rede

Abordagens para o controle de congestionamento

- Duas grandes abordagens para controle de congestionamento
  - **Controle de congestionamento fim-a-fim**
    - Não há realimentação explícita vinda da rede
    - Congestionamento é inferido pelos sistemas finais a partir das perdas e atrasos observados
      - Indicado por um timeout ou um reconhecimento triplicado
    - Abordagem adotada pelo TCP
  - **Controle de congestionamento assistida pela rede**
    - Roteadores fornecem realimentação aos sistemas finais
    - SNA, DECbit, TCP/IP ECN, ATM



TCP: controle de congestionamento

- Controle fim-a-fim (sem assistência da rede)
- Transmissor limita a transmissão:
  - LastByteSent-LastByteAcked ≤ CongWin
- Aproximadamente,

$$\text{Taxa} = \frac{\text{CongWin}}{\text{RTT}} \text{ Bytes/sec}$$

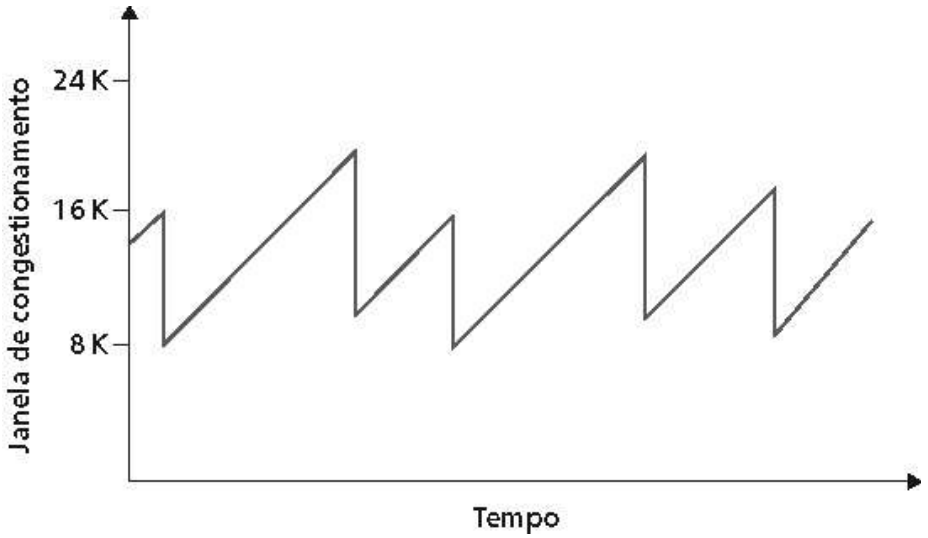
- CongWin é dinâmico, função de congestionamento das redes detectadas
- Como o transmissor detecta o congestionamento?
  - Evento de perda = timeout do temporizador ou 3 ACKs duplicados
- Transmissor TCP reduz a taxa (CongWin) após o evento de perda

TCP: controle de congestionamento

- Três mecanismos:
  - AIMD (additive-increase, multiplicative-decrease)
  - Partida lenta
  - Reação a eventos de esgotamento de temporização

TCP: controle de congestionamento

- AIMD (additive-increase, multiplicative-decrease)
  - Redução multiplicativa: diminui o CongWin pela metade após o evento de perda
  - Aumento aditivo: aumenta o CongWin com 1 MSS a cada RTT na ausência de eventos de perda: probing



Controle de Congestionamento TCP

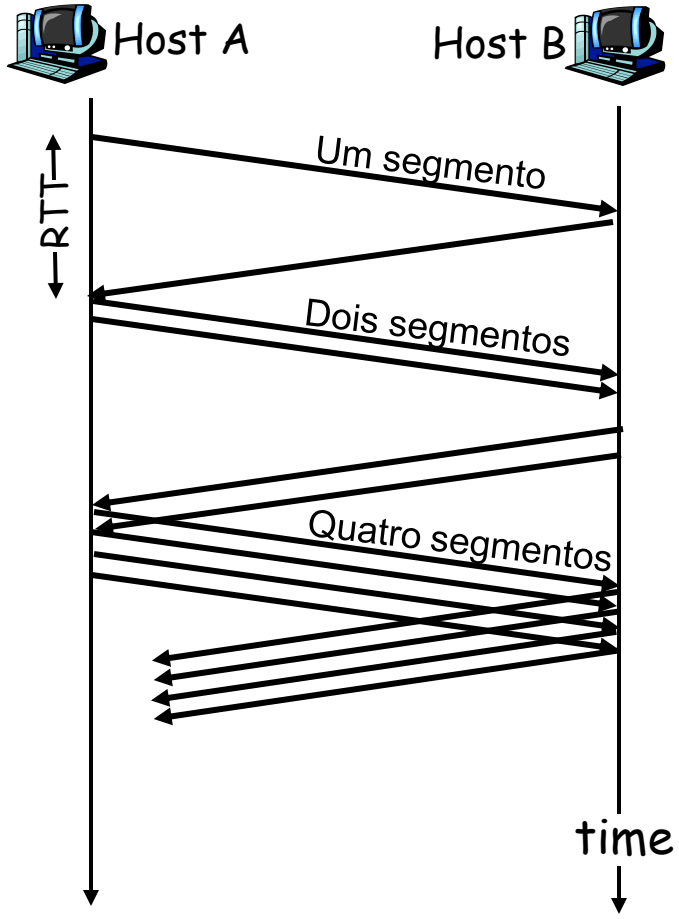
- **Partida Lenta**
  - Banda oferecida cresce exponencialmente até chegar a um limiar ou ocorrência de perda
    - Na ocorrência da perda recomeça a partida lenta
  - Quando a conexão começa, CongWin = 1 MSS
    - Exemplo: MSS = 500 bytes e RTT = 200 milissegundos
      - Taxa inicial = 20 kbps
  - Largura de banda disponível pode ser >> MSS/RTT
    - Desejável aumentar rapidamente até a taxa respeitável
    - Quando a conexão começa, a taxa aumenta rapidamente de modo exponencial até a ocorrência do primeiro evento de perda

Partida lenta TCP

Algoritmo Partida lenta

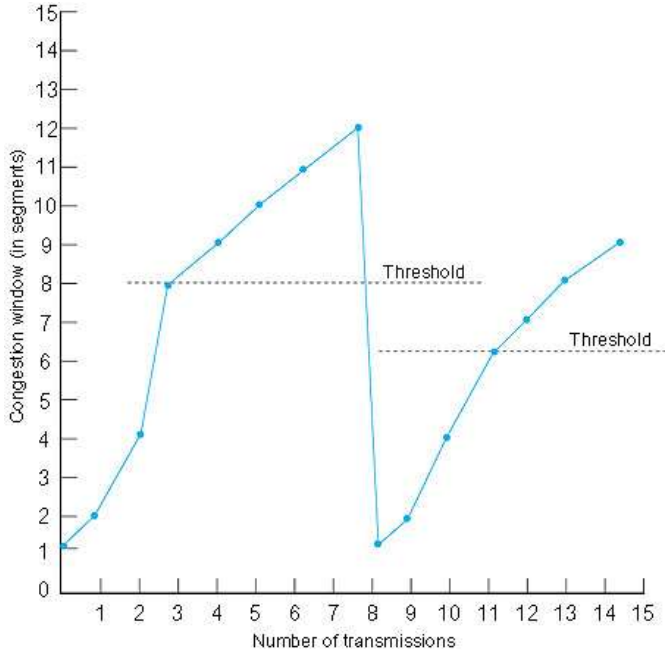
inicializa: Congwin = 1 MSS
Para (cada segm com ack)
    Congwin++
Até (evento de perda OU
    CongWin > threshold)

- Incremento exponencial no tamanho da janela (não muito lenta!)
- Evento de perda: timeout e/ou três ACKs duplicados



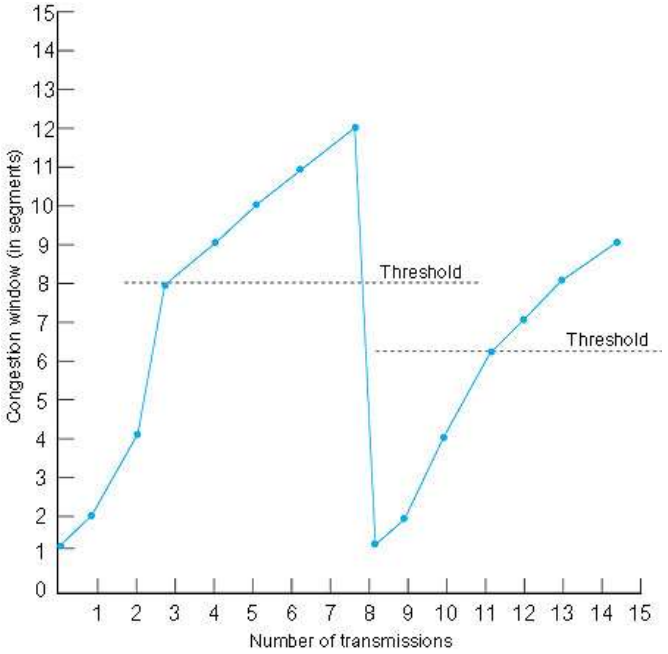
Controle de Congestionamento TCP

- Fase de partida lenta
  - Termina quando o tamanho da janela excede o valor do threshold
    - Uma vez que a janela de congestionamento é maior que o valor atual do threshold, a janela de congestionamento cresce linearmente (e não mais exponencialmente)
  - Esta fase é chamada de prevenção do congestionamento



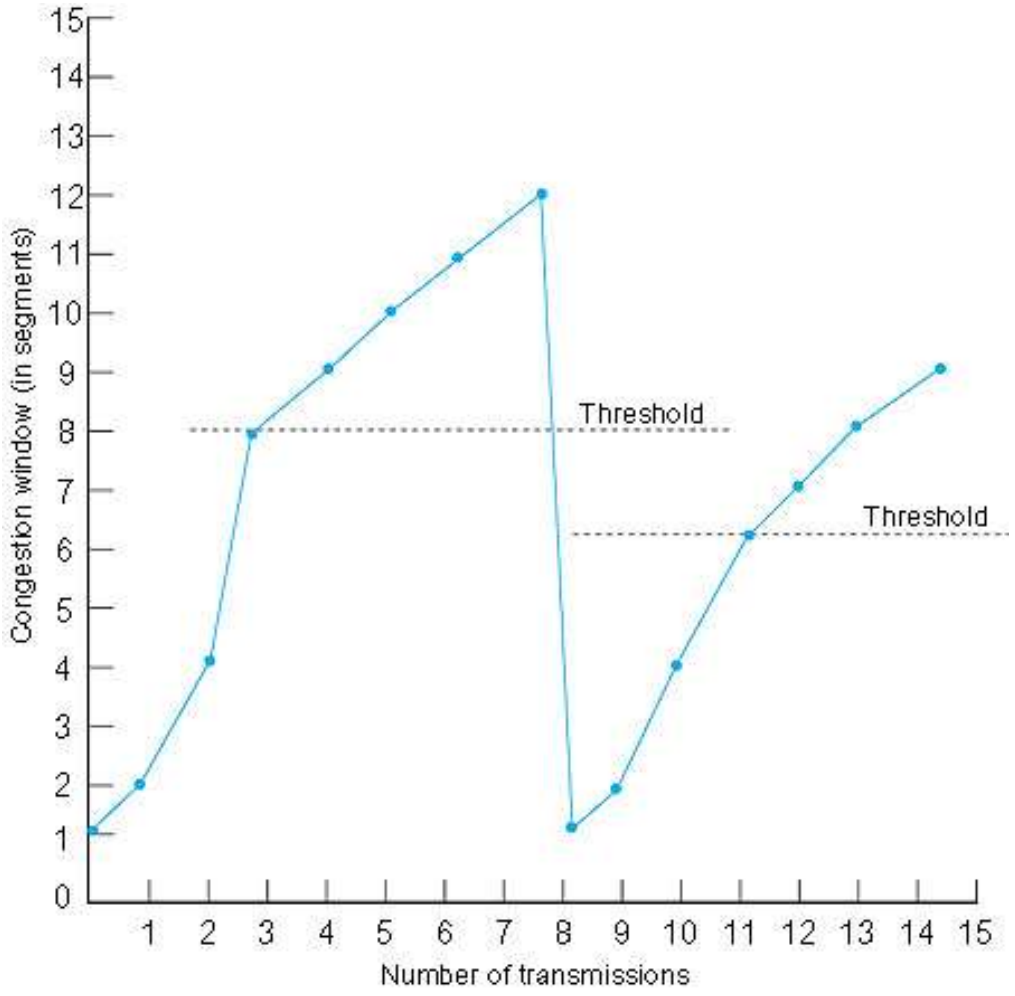
Controle de Congestionamento TCP

- **Fase de prevenção de congestionamento**
  - Mas o tamanho da janela não podem crescer para sempre
  - Eventualmente, a taxa TCP será tal que um dos enlaces seja saturado
    - Em que um ponto de perdas (e resultante timeout no emissor) ocorrerão
  - Quando um timeout ocorrer
    - O valor do threshold é setado como a metade do valor da janela de congestionamento atual, e a janela de congestionamento é resetada para um MSS
    - O emissor então procede o incremento exponencial da janela de congestionamento usando o procedimento de partida lenta até a janela de congestionamento alcançar o threshold



Controle de Congestionamento TCP

- Fase de prevenção de congestionamento



Controle de Congestionamento TCP

Evento	Estado	Ação do transmissor TCP	Comentário
ACK recebido para dado previamente não confirmado	partida lenta (SS)	CongWin = CongWin + MSS, If (CongWin > Threshold) ajusta estado para "prevenção de congestionamento"	Resulta em dobrar o CongWin a cada RTT
ACK recebido para dado previamente não confirmado	prevenção de congestiona-mento (CA)	CongWin = CongWin + MSS * (MSS/CongWin)	Aumento aditivo, resulta no aumento do CongWin em 1 MSS a cada RTT
Evento de perda detectado por três ACKs duplicados	SS or CA	Threshold = CongWin/2, CongWin = Threshold, ajusta estado para "prevenção de congestionamento"	Recuperacao rápida, implementando redução multiplicativa o CongWin não cairá abaixo de 1 MSS.
Tempo de confirmação	SS or CA	Threshold = CongWin/2, CongWin = 1 MSS, ajusta estado para "partida lenta"	Entra em partida lenta
ACK duplicado	SS or CA	Incrementa o contador de ACK duplicado para o segmento que está sendo confirmado	CongWin e Threshold não mudam

Controle de Congestionamento TCP

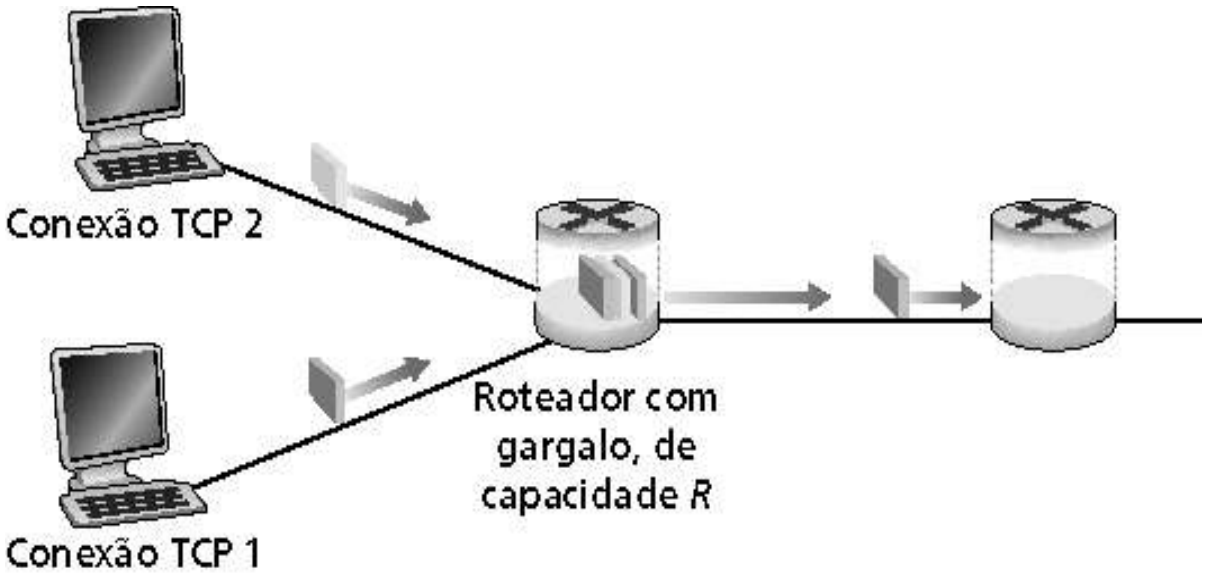
- Algoritmo Tahoe
  - Aquele descrito neste capítulo
  - Problema
    - Quando um segmento é perdido, o emissor tem que esperar um período longo até a ocorrência do timeout
- Algoritmo Reno
  - Implementado por muitos sistemas operacionais
  - Como Tahoe, Reno seta sua janela de congestionamento para um segmento na expiração de um timer
  - Reno também inclui uma retransmissão rápida
    - Retransmite um segmento perdido se três ACKs duplicados para um segmento são recebidos antes do timeout
  - Reno também emprega um mecanismo de recuperação rápida que cancela a fase de partida lenta após uma retransmissão rápida

Controle de Congestionamento TCP

- Algoritmo Vegas
  - Aumenta o desempenho do Reno
  - Tenta evitar o congestionamento mantendo uma boa vazão
  - Idéia básica
    - detectar o congestionamento nos roteadores entre a fonte e destino antes da ocorrência da perda
    - reduz a taxa linearmente quando a perda eminente de pacotes é detectada
    - Perda eminente é prevista pela observação do atraso de ida-e-volta
      - Maior o atraso de ida-e-volta dos pacotes, maior o congestionamento nos roteadores

Eqüidade do TCP

- Objetivo de eqüidade: se K sessões TCP compartilham o mesmo enlace do gargalo com largura de banda R, cada uma deve ter taxa média de R/K



Eqüidade do TCP

- Eqüidade e UDP
  - Aplicações multimídia normalmente não usam TCP
    - Não querem a taxa estrangulada pelo controle de congestionamento
  - Em vez disso, usam UDP:
    - Trafega áudio/vídeo a taxas constantes, toleram perda de pacotes
- Eqüidade e conexões TCP paralelas
  - Nada previne as aplicações de abrirem conexões paralelas entre 2 hospedeiros
  - Web browsers fazem isso
  - Exemplo: enlace de taxa R suportando 9 conexões;
    - Nova aplicação pede 1 TCP, obtém taxa de R/10
    - Nova aplicação pede 11 TCPs, cada uma obtém R/20!