

# Camada de Transporte e TCP - UDP

Material baseado nas apresentações (*slides*) disponibilizados junto com o livro referência a seguir.

*Bibliografia:*  
*Computer Networking:  
A Top Down Approach*

7<sup>th</sup> Edition, Global Edition  
Jim Kurose, Keith Ross  
Pearson  
April 2016

A note on the use of these Powerpoint slides:  
We're making these slides freely available to all (faculty, students, readers). They're in PowerPoint form so you see the animations; and can add, modify, and delete slides (including this one) and slide content to suit your needs. They obviously represent a *lot* of work on our part. In return for use, we only ask the following:



All material copyright 1996-2016  
J.F Kurose and K.W. Ross, All Rights Reserved

# Objetivos

- Entender os princípios dos serviços de camada de transporte: **multiplexação, demultiplexação, transferência confiável, controle de fluxo, controle de congestionamento**
- Protocolos de transporte da Internet:  
UDP: transporte sem conexão  
TCP: transporte confiável orientado à conexão; controle de congestionamento

# sumário

## Serviços de transporte

Multiplexação e demultiplexação

Transporte sem conexão: UDP

Princípios de transporte com confiança

Transporte orientado a conexão: TCP

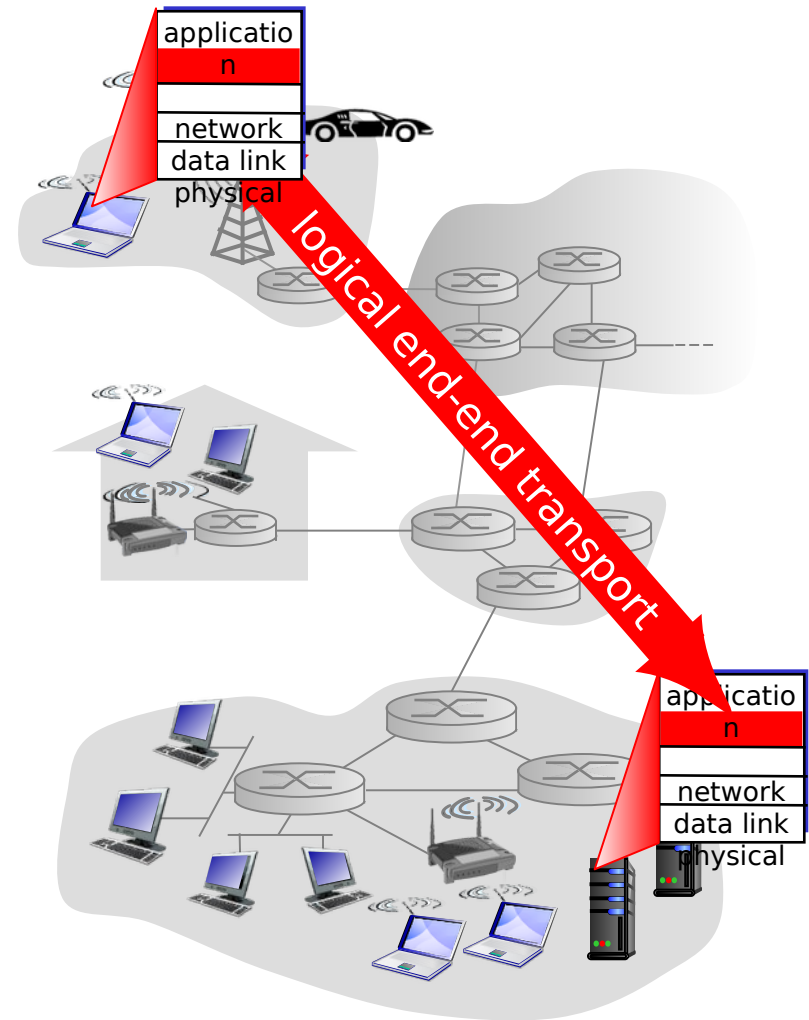
- Estrutura do segmento
- Transferência de dados confiável
- Controle de fluxo
- Gerenciamento da conexão

Princípios de controle de congestionamento

Controle de congestionamento no TCP

# Serviços de transporte e protocolos

- Provê *comunicação lógica* entre **aplicações** executando em diferentes *hosts*
- Protocolos de transporte são executados nas entidades finais
  - Fonte: quebra as mensagens da aplicação em *segmentos*, e os passa para a camada de rede
  - Destino: remonta os segmentos em mensagens e passa para a camada da aplicação
- O número de protocolos de transporte está ligado ao tipo de serviço oferecido às aplicações
  - Internet: TCP e UDP



# Camada de transporte

Camada de **rede**:  
comunicação lógica  
entre hosts

Camada de **transporte**:  
comunicação lógica  
entre processos

- usa e acrescenta  
melhorias nos serviços  
prestados pela camada  
de rede

*Analogia de residências:*

*12 crianças na casa da  
Ann enviam cartas a 12  
crianças na casa de Bill:*

- hosts = casas
- processos = crianças
- mensagens de aplicação = cartas nos envelopes
- protocolo de transporte = Ann e Bill que demultiplexam os envelopes
- protocolo de camada de rede = serviço postal

# Protocolos de transporte da Internet

**TCP:** entrega confiável e em ordem

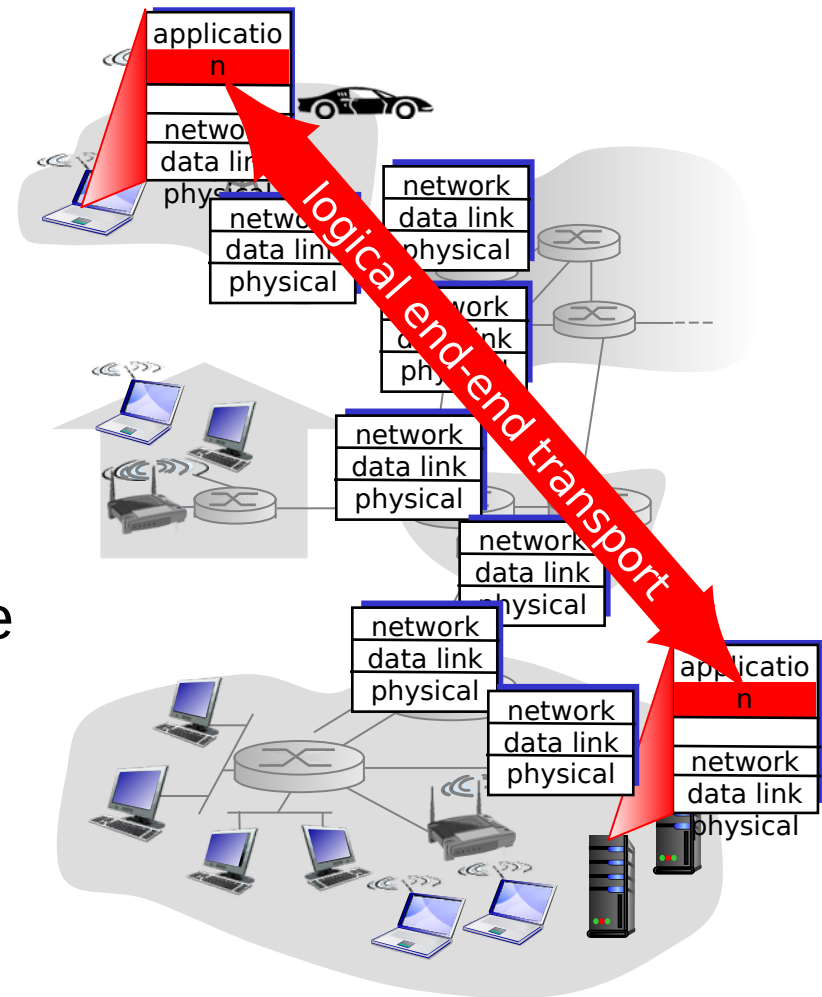
- controle de congestionamento
- controle de fluxo
- configuração da conexão

**UDP:** entrega não confiável e sem garantia de ordem

- Extensão simples ao modelo de entrega melhor esforço do IP

**Serviços não disponibilizados:**

- garantia de atrasos determinísticos
- garantia de largura de banda



# sumário

Serviços de transporte

Multiplexação e demultiplexação

Transporte sem conexão: UDP

Princípios de transporte com confiança

Transporte orientado a conexão: TCP

- Estrutura do segmento
- Transferência de dados confiável
- Controle de fluxo
- Gerenciamento da conexão

Princípios de controle de congestionamento

Controle de congestionamento no TCP

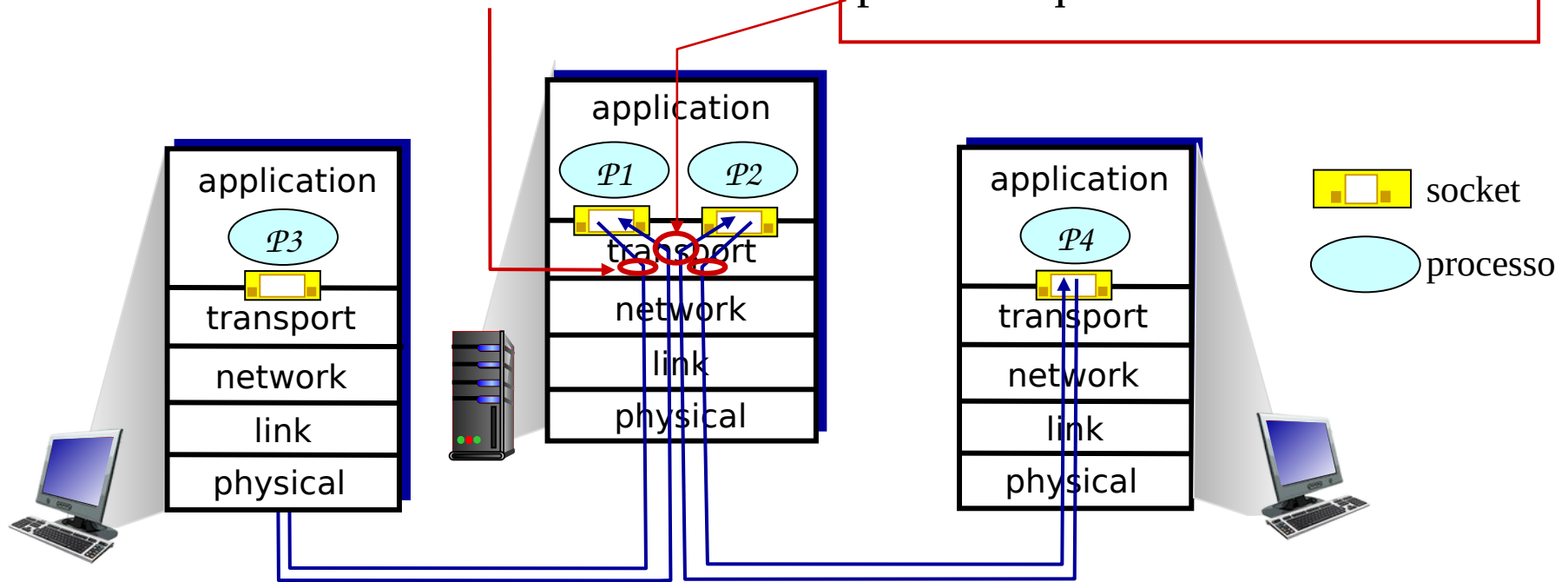
# Multiplexação e demultiplexação

## Multiplexação na fonte

Trata os dados de múltiplos soquetes; adiciona cabeçalho transporte (será usado mais tarde durante demult)

## Demultiplexação no destino

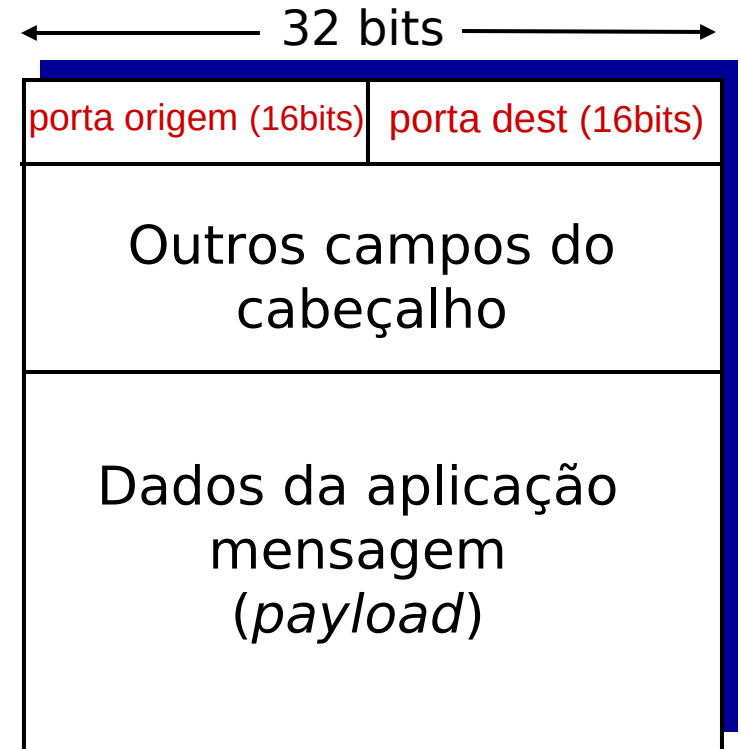
Usa as informações do cabeçalho para destinar os segmentos para o soquete correto.





# Como a multiplexação funciona

- Host recebe datagramas IP
  - cada datagrama possui um endereço fonte e um endereço destino
  - cada datagrama carrega um segmento de dados da camada de transporte
  - cada segmento possui uma porta de origem e de destino
- Host usa *endereço IP e números de porta* para direcionar o segmento para o soquete apropriado



Formato básico **segmento UDP**

# Demultiplexação sem conexão

- Criação de um *socket* no lado do cliente:  
`clientSocket = socket (AF_INET, SOCK_DGRAM)`

- Criação de um *socket* no lado do servidor:  
`serverSocket = socket (AF_INET, SOCK_DGRAM)`  
`ServerSocket.bind ("", serverPort)`

- 
- Quando um host recebe um segmento UDP:
    - checa a porta de destino no segmento
    - direciona o segmento UDP para o respectivo soquete que ouve na porta destino



Se dois datagramas IP tiverem o mesmo *end destino e mesma porta destino*, mas diferentes endereços de origem e/ou diferente porta de origem, então serão direcionados ao *mesmo soquete* no destino

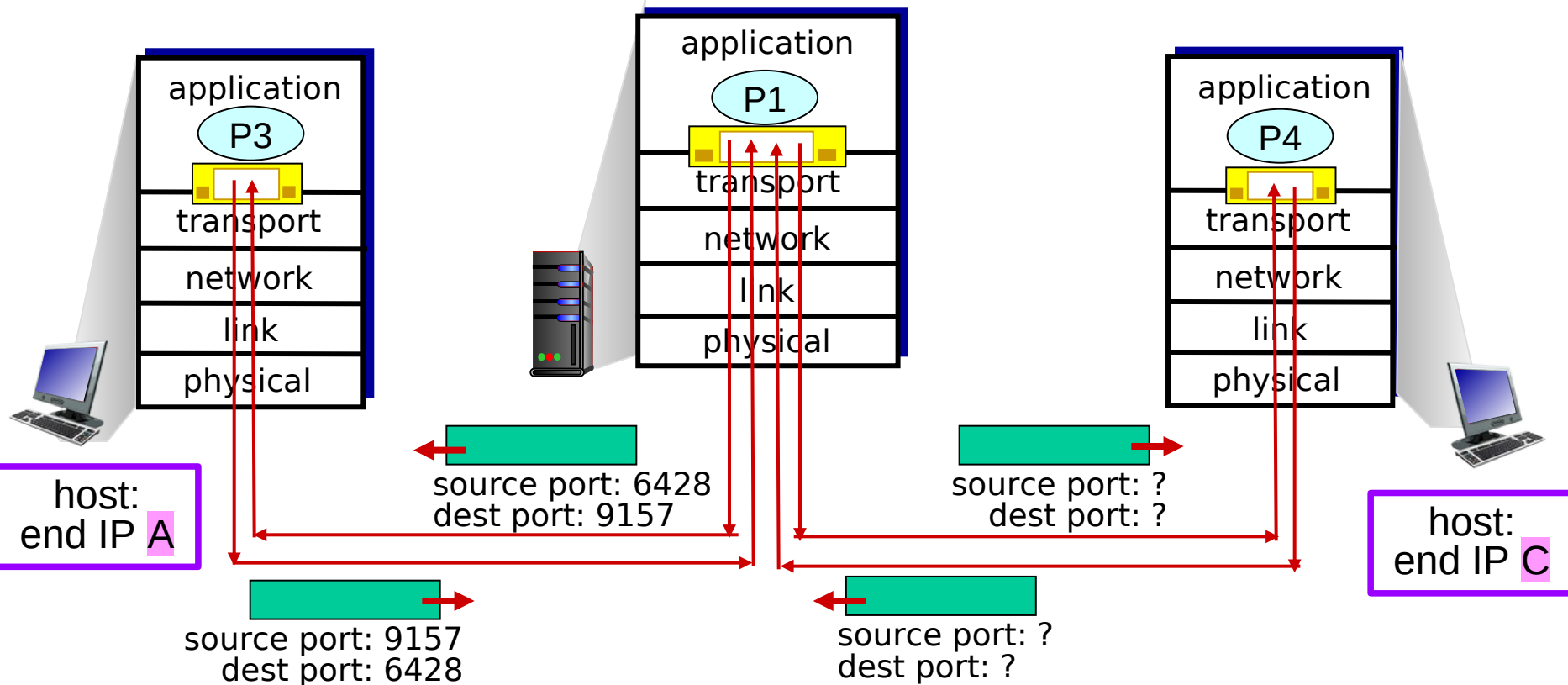
# Demultiplexação sem conexão: exemplo

```
clientSocket = socket  
(AF_INET, SOCK_DGRAM)  
ClientSocket.sendto(  
message, (B, 6428))
```

```
serverSocket = socket (AF_INET,  
SOCK_DGRAM)  
serverSocket.bind (("", 6428))
```

```
clientSocket = socket  
(AF_INET, SOCK_DGRAM)  
ClientSocket.sendto(  
message, (B, 6428))
```

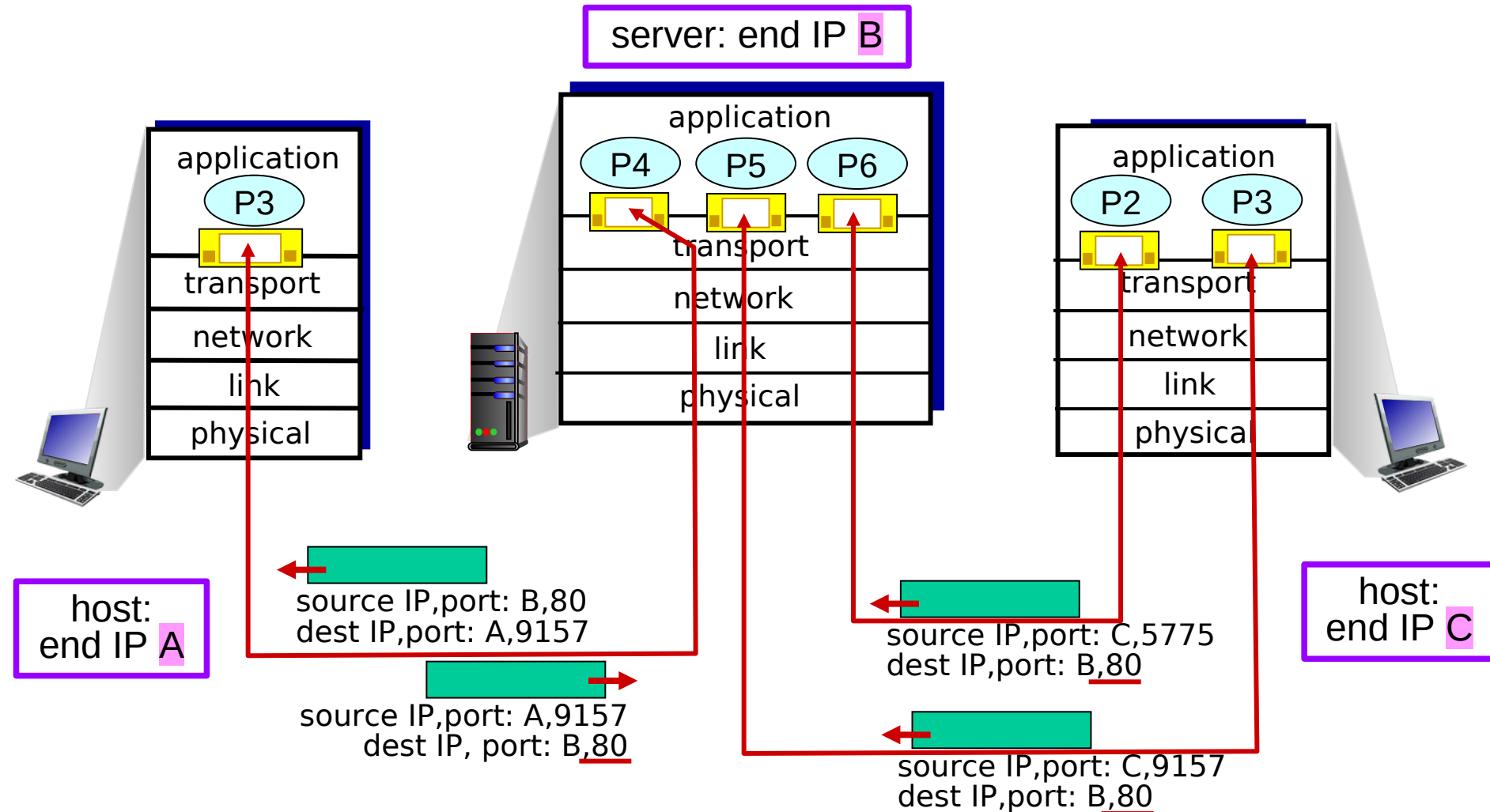
server: end IP B



# Demultiplexação orientada a conexão

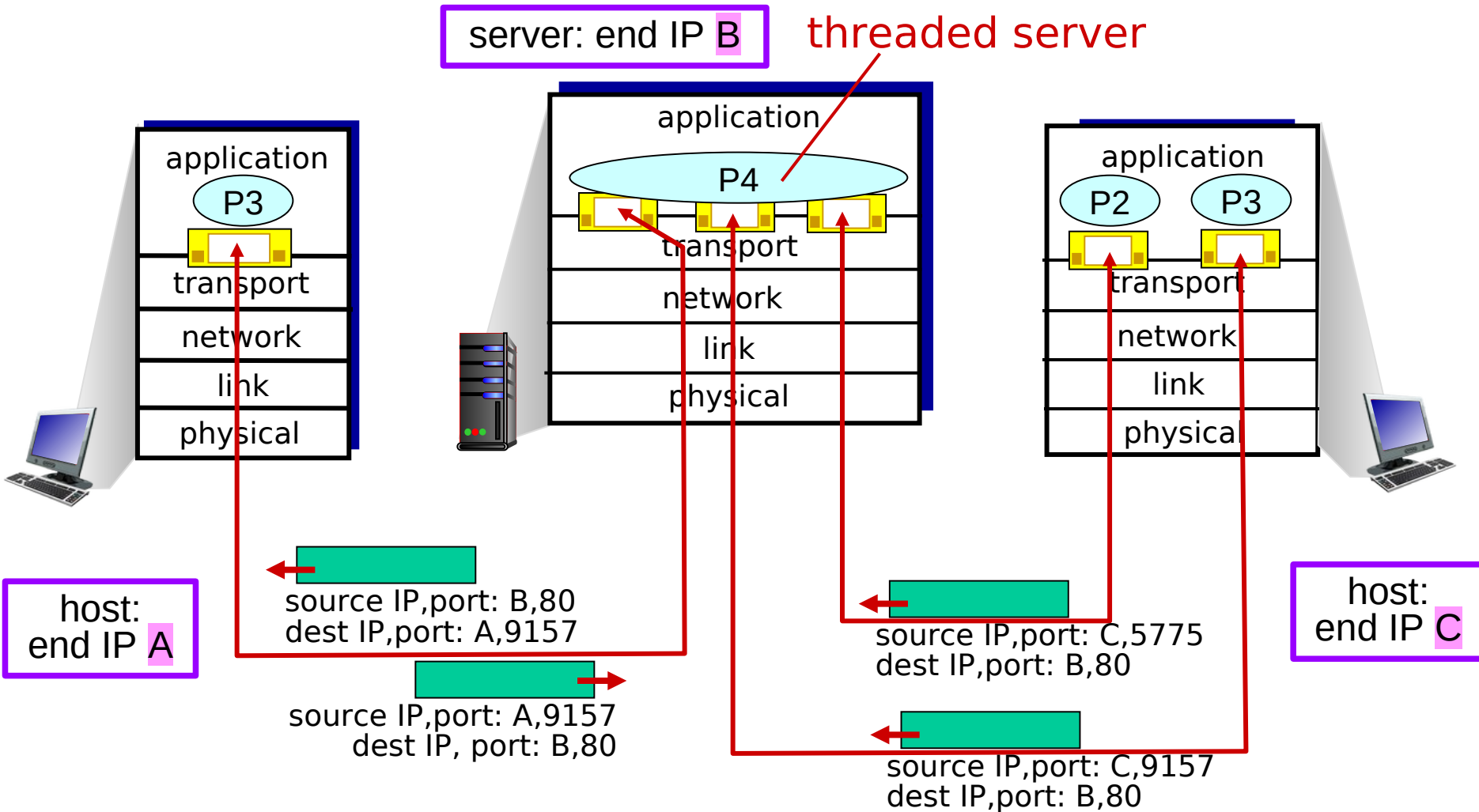
- TCP *socket*: identificado por uma 4-tupla:
  - Endereço IP origem
  - Número de porta de origem
  - Endereço IP destino
  - Número de porta destino
- demux: destinatário usa os quatro valores para direcionar o segmento para o *socket* apropriado
- Um servidor pode suportar muitas conexões simultâneas (*sockets*):
  - cada socket identificado por uma 4-tupla
- Servidores Web tem diferentes *sockets* para cliente que faz uma conexão
  - HTTP não persistente terá diferentes *sockets* para cada requisição

# Demultiplexação orientada a conexão



Três segmentos: todos destinados ao endereço IP B e porta destino 80:  
Demultiplexados para diferentes sockets

# Demultiplexação orientada a conexão



# Portas e serviços

As portas que especificam soquetes estão relacionadas à:

- serviços no lado servidor (*socket* no servidor)
- identificação de soquetes no lado cliente

- Servidores:
  - números de porta atribuídos pela IANA ([www.iana.org](http://www.iana.org)): *well-known ports*
  - No Linux:  
`/etc/services`
  - No Windows  
`C:\Windows\System32\drivers\etc`
- No hosts clientes: os números de porta são definidos pelo SO e devem ser aleatórios (16 bits= 65536 valores – portas serviços (abaixo de 1024))

# Portas e serviços

Comando no Linux e Windows para visualizar as conexões com os respectivos endereços IP e portas:

\$ `netstat -teaun`

\$ `netstat -tunlp` (somente portas TCP (**t**) e UDP (**u**) em estado de escuta (**l**), ou seja, processos servidores, e os respectivos processos associados (**p**))

No linux o aplicativo **lsof** permite visualizar os processos associados a uma determinada porta.

\$ `lsof -i :53` (mostra qual processo associado à porta 53)

No Windows é o mesmo comando, mas algumas opções são diferentes:

> `netstat -a -n -o -b` (-o mostra os processos associados)

Também há o aplicativo (*standalone*) TCPView do pacote Sysinternals ([www.sysinternals.com](http://www.sysinternals.com) Site original, antes da aquisição pela Microsoft: atual <https://live.sysinternals.com/>). Também, contém muitas ferramentas para conhecer aspectos internos do SO Windows.



# sumário

Serviços de transporte

Multiplexação e demultiplexação

Transporte sem conexão: UDP

Princípios de transporte com confiança

Transporte orientado a conexão: TCP

- Estrutura do segmento
- Transferência de dados confiável
- Controle de fluxo
- Gerenciamento da conexão

Princípios de controle de congestionamento

Controle de congestionamento no TCP

# UDP: User Datagram Protocol [RFC 768]

- Protocolo de transporte de Internet básico
- Serviço “*best effort*”.
- Segmentos UDP podem ser:
  - perdidos
  - entregues fora-de-ordem às aplicações
- *Sem conexão:*
  - Não há estabelecimento de conexão (*handshaking*) entre origem e destino
  - Cada segmento UDP tratado independentemente dos outros

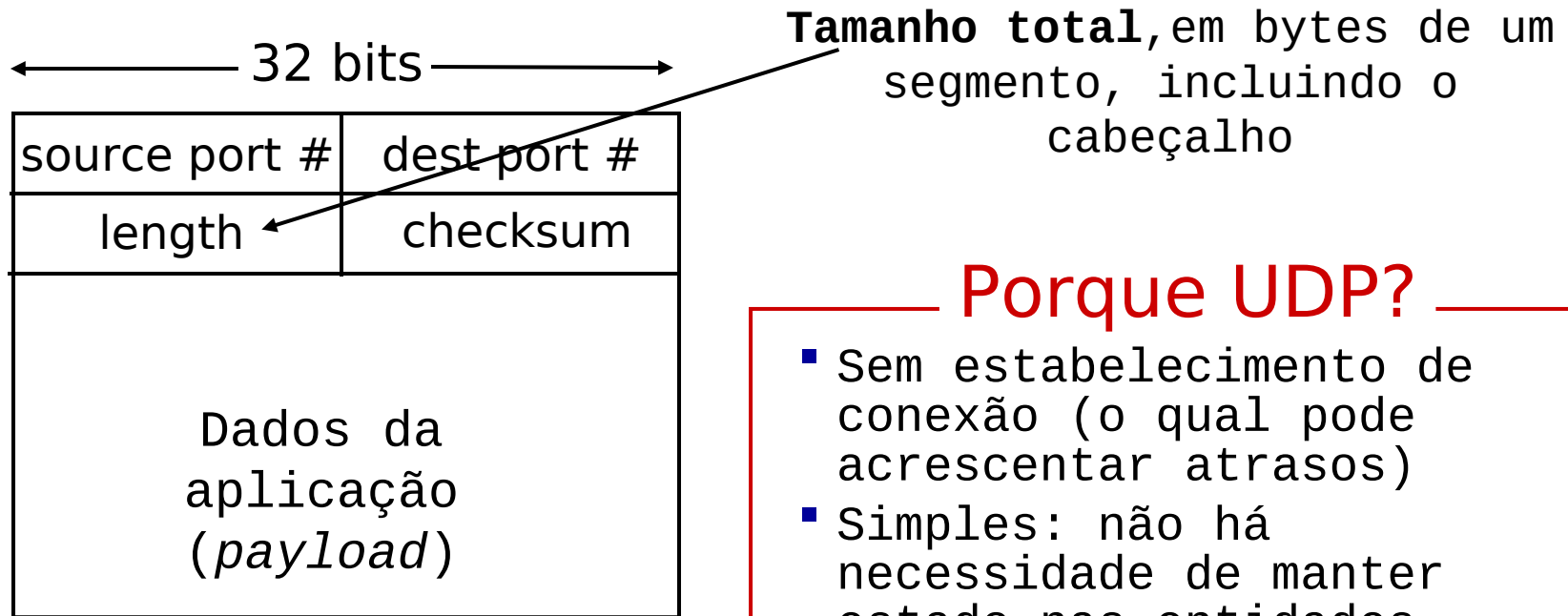
## Usos do UDP:

- Aplicações de streaming (tolerantes a perdas, mas sensíveis à taxa de transmissão)
- DNS
- SNMP

## Transferência confiável sobre o UDP:

- Adicionar confiança ao nível da aplicação
- Recuperação de erros na transmissão realizada pela aplicação

# UDP: formato do segmento (cabeçalho)



## Porque UDP?

- Sem estabelecimento de conexão (o qual pode acrescentar atrasos)
- Simples: não há necessidade de manter estado nas entidades emissoras e receptoras
- Tamanho de cabeçalho pequeno
- Sem controle de congestionamento: UDP pode ser transmitido na velocidade máxima da rede

**Checksum**, permite ao receptor detectar erros no segmento. O checksum inclui cabeçalho e dados. Campo opcional. O UDP não prevê medidas de correção para erros detectados. Fica a cargo da aplicação.

# UDP checksum

*objetivo:* detectar erros no segmento transmitido

## Origem:

- Trata os segmentos, incluindo os campos do cabeçalho como sequência de inteiros de 16 bits
- *checksum*: adição (soma complemento de um) dos conteúdos dos segmentos
- Emissor coloca o valor do *checksum* dentro do respectivo campo no UDP

## Destinatário:

- Calcula o *checksum* do segmento recebido
- Checa se o *checksum* calculado é igual ao valor contido no campo *checksum*

# Internet: *checksum* – um exemplo

Exemplo: adicionar dois inteiros de 16 bits

$$\begin{array}{r} 1\ 1\ 1\ 0\ 0\ 1\ 1\ 0\ 0\ 1\ 1\ 0\ 0\ 1\ 1\ 0 \\ 1\ 1\ 0\ 1\ 0\ 1\ 0\ 1\ 0\ 1\ 0\ 1\ 0\ 1\ 0\ 1 \\ \hline \text{wraparound } 1\ 1\ 0\ 1\ 1\ 1\ 0\ 1\ 1\ 1\ 0\ 1\ 1\ 1\ 0\ 1\ 1 \\ \hline \text{sum } 1\ 0\ 1\ 1\ 1\ 0\ 1\ 1\ 1\ 0\ 1\ 1\ 1\ 1\ 0\ 0 \\ \text{checksum } 0\ 1\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 1\ 1 \end{array}$$

*Nota:* quando adicionando números, um *carry* do bit mais significativo precisa ser adicionado ao resultado

# UDP: interação cliente/servidor por meio de sockets

Servidor  
(executando em um serverIP)

Cria um socket e o vincula à porta=serverPort  
`serverSocket = socket(AF_INET, SOCK_DGRAM)`  
`ServerSocket.bind('', serverPort)`

Lê o datagrama do  
`serverSocket`

Escreve resposta no  
`serverSocket`  
especificando o endereço cliente e a respectiva porta da aplicação cliente

cliente

Cria socket  
`clientSocket = socket(AF_INET, SOCK_DGRAM)`

Envia um datagrama contendo a mensagem Para o endereço do servidor UDP (end IP) e especificando a respectiva porta  
`clientSocket.sendto(message, (serverName, serverPort))`

Lê o a mensagem contida no datagrama  
`clientSocket.recvfrom(2048)`

Fecha socket  
`clientSocket.close()`



# Cliente UDP: um exemplo de aplicação

## *Python UDPClient*

Inclusão de biblioteca  
Python

Cria um socket

Usa o socket criado para  
enviar a mensagem contendo  
o nome do destinatário e a  
respectiva porta do  
servidor

Lê a resposta do  
mesmo socket

Imprime a saída

```
from socket import *
serverName='hostname'
serverPort=12000
clientSocket=socket(AF_INET, SOCK_DGRAM)
message=raw_input('Input lowercase sentence:')
clientSocket.sendto(message.encode(),
                    (serverName, serverPort))
modifiedMessage,serverAddress=
    clientSocket.recvfrom(2048)
print modifiedMessage.decode()
clientSocket.close()
```

# Servidor UDP: um exemplo de aplicação

## *Python UDPServer*

```
from socket import *
```

```
serverPort = 12000
```

Cria um socket → `serverSocket = socket(AF_INET, SOCK_DGRAM)`

Vincula (bind) o socket com uma porta → `serverSocket.bind(('', serverPort))`

```
print ("The server is ready to receive")
```

Loop → `while True:`

Read from UDP socket into message, getting client's address (client IP) → `message, clientAddress=serverSocket.recvfrom(2048)`

send uppercase string back to this client → `modifiedMessage=message.decode().upper()`

→ `serverSocket.sendto(modifiedMessage.encode(), clientAddress)`



# Código exemplo de um cliente **UDP**

```
# UDPClient.py
from socket import *
serverName = 'localhost'
serverPort = 33333

# Cria um soquete UDP: SOCK_DGRAM
clientSocket = socket(AF_INET, SOCK_DGRAM)
message = input('Input lowercase sentence: ')

clientSocket.sendto (message.encode(), (serverName, serverPort))
modifiedMessage, serverAddress = clientSocket.recvfrom(2048)
print (modifiedMessage.decode())
clientSocket.close()
```

# Código exemplo de um servidor UDP

```
# UDPServer.py
from socket import *
serverPort = 33333

# Cria um soquete UDP: SOCK_DGRAM
serverSocket = socket(AF_INET, SOCK_DGRAM)

# Associa uma porta (bem conhecida) ao processo servidor
serverSocket.bind(("", serverPort))
print ("The server is ready to receive. ")

while True:
    message, clientAddress = serverSocket.recvfrom (2048)
    modifiedMessage = message.decode().upper()
    serverSocket.sendto (modifiedMessage.encode(), clientAddress)
```

# sumário

Serviços de transporte

Multiplexação e demultiplexação

Transporte sem conexão: UDP

Princípios de transporte com confiança

Transporte orientado a conexão: TCP

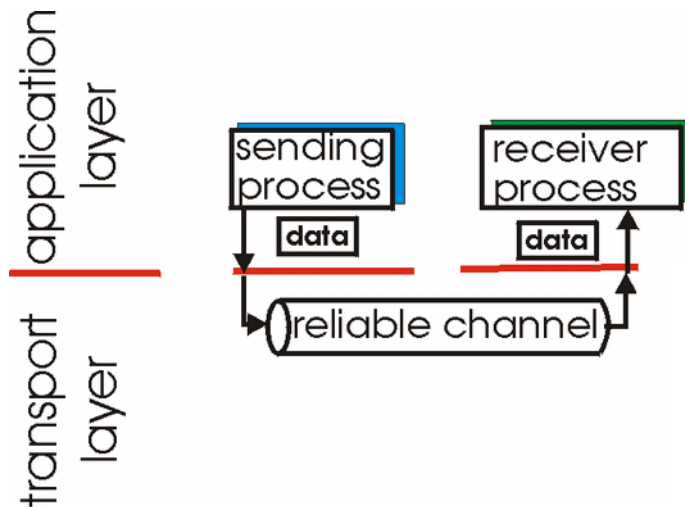
- Estrutura do segmento
- Transferência de dados confiável
- Controle de fluxo
- Gerenciamento da conexão

Princípios de controle de congestionamento

Controle de congestionamento no TCP

# Princípios de transferência confiável de dados

- Aspecto importante nos contextos das camadas de aplicação, transporte e de enlace

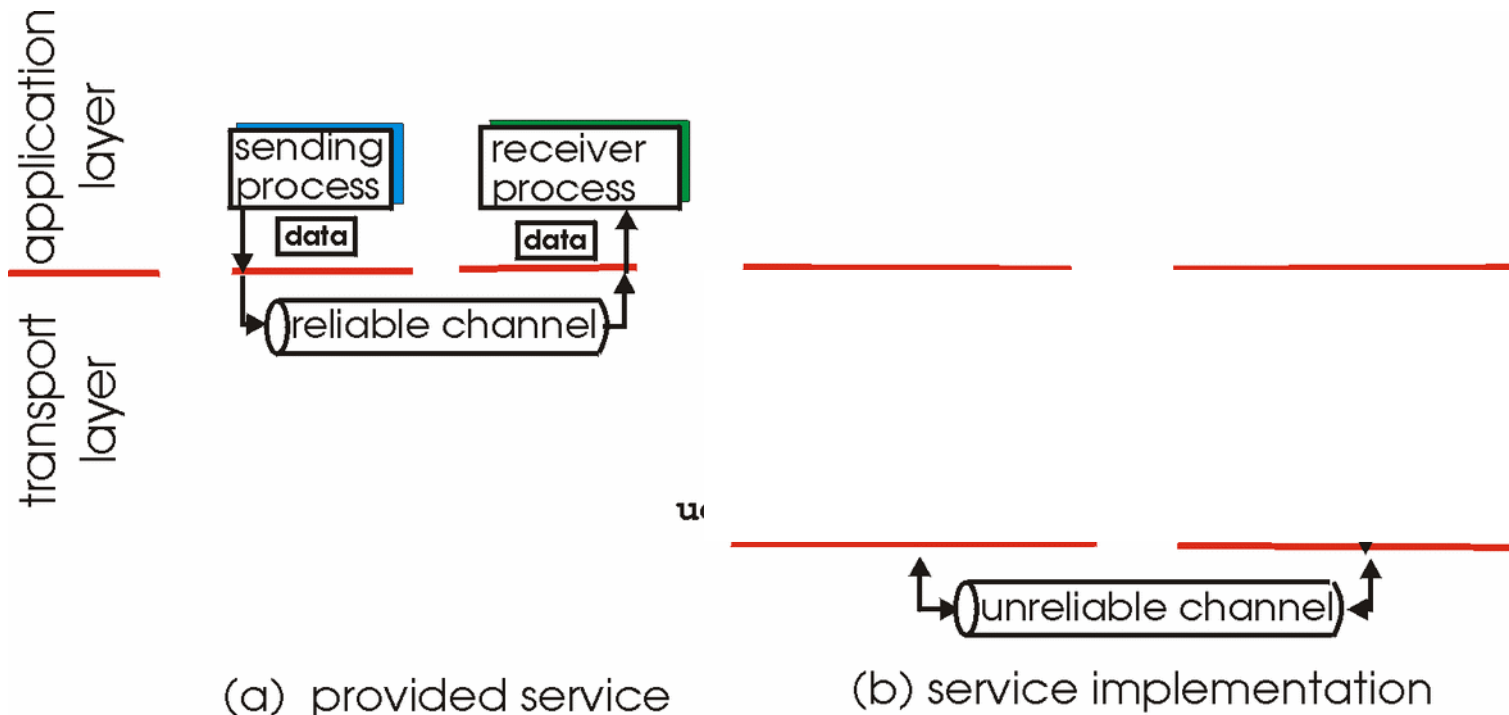


(a) provided service

- A característica subjacente do canal não confiável vai determinar a complexidade do protocolo de transferência confiável de dados (rdt)

# Princípios de transferência confiável de dados

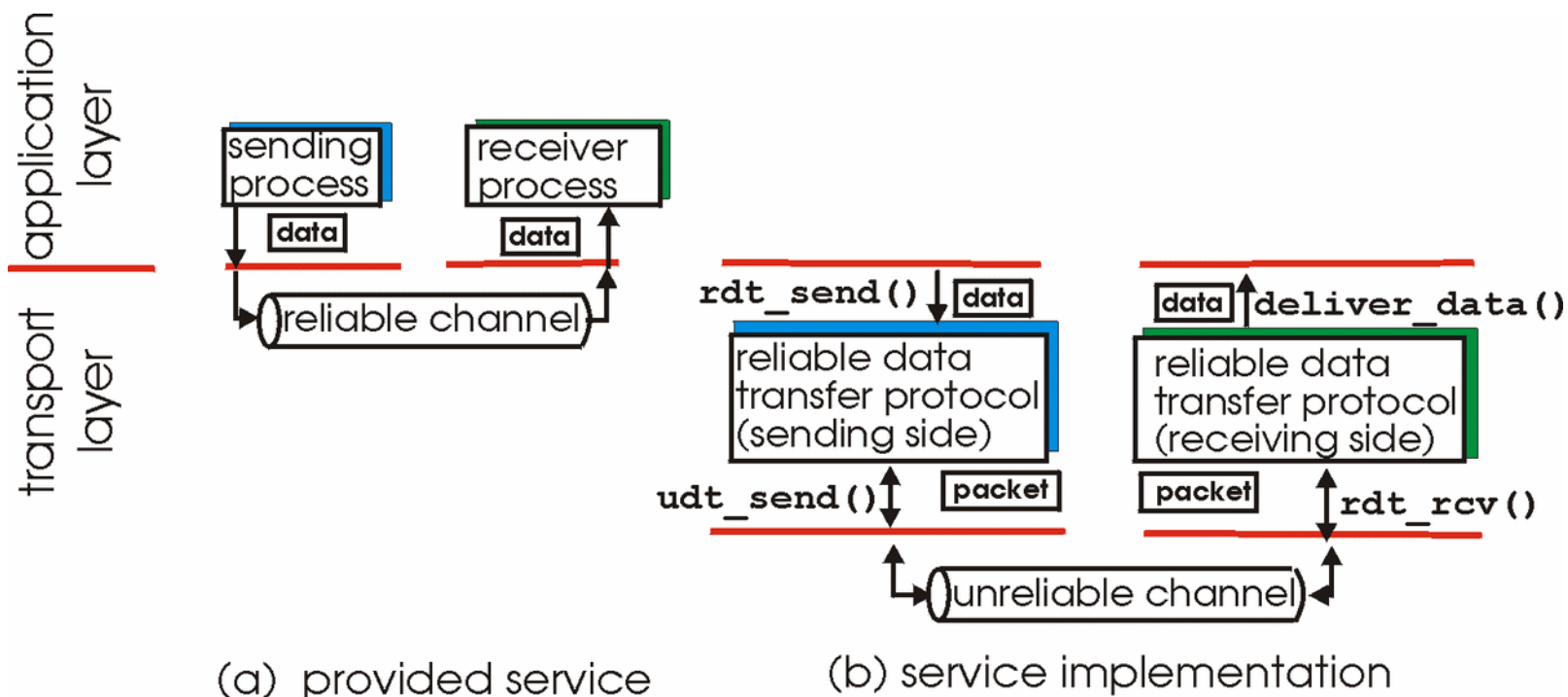
- Aspecto importante nos contextos das camadas de aplicação, transporte e de enlace



A característica subjacente do canal não confiável vai determinar a complexidade do **protocolo de transferência confiável de dados (rdt)**

# Princípios de transferência confiável de dados

Aspecto importante nos contextos das camadas de aplicação, transporte e de enlace. A figura da esquerda representa o serviço confiável provido pela camada de transporte, enquanto a da direita, aspectos relacionados à implementação **rdt**.

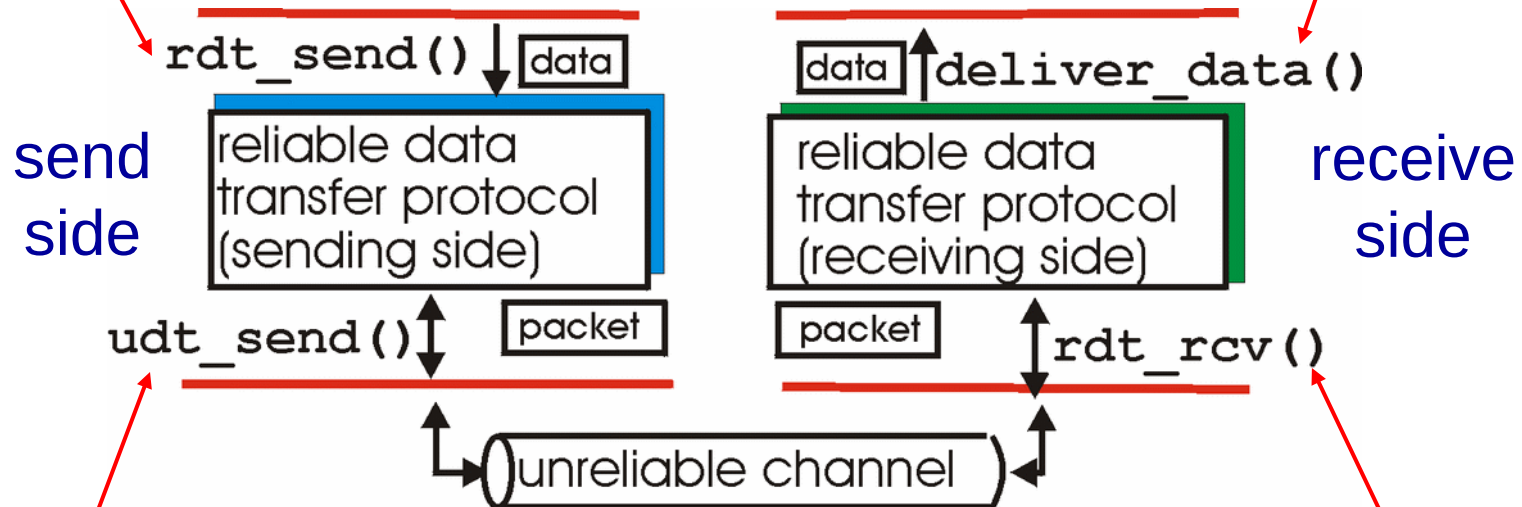


A característica subjacente do canal não confiável vai determinar a complexidade do **protocolo de transferência confiável de dados (rdt)**

# Transferência confiável de dados

**rdt\_send():** chamada da camada superior (aplicação) passando dados a serem transmitidos.

**deliver\_data():** chamada por rdt para entregar dados à camada superior



**udt\_send():** chamada pelo rdt, para transferir pacotes de dados sobre o canal não confiável

**rdt\_rcv():** chamada quando um pacote chega ao lado do receptor

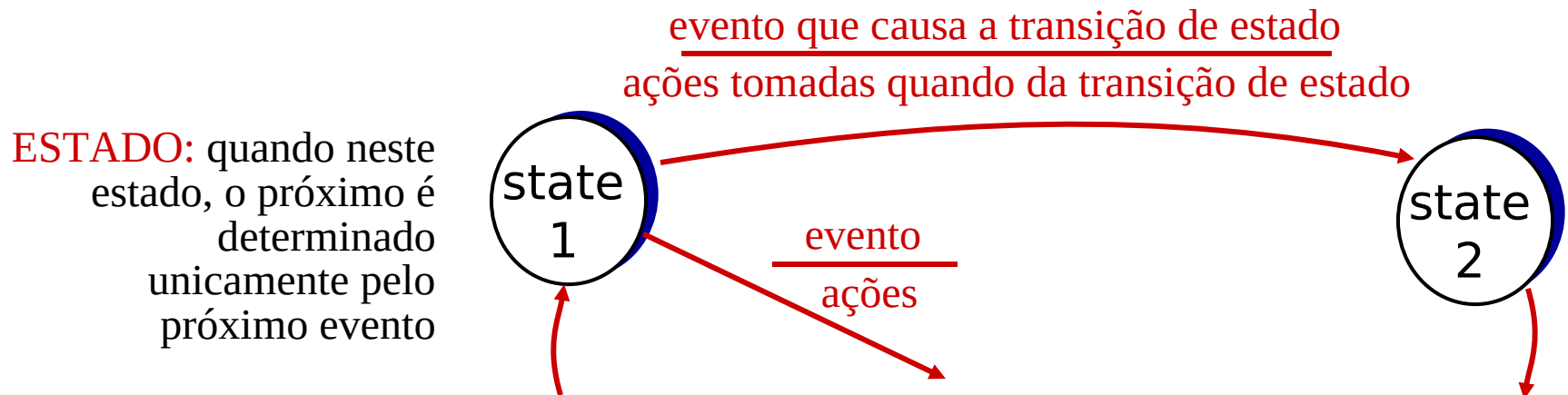
# Transferência confiável de dados

***reliable data transfer protocol*** (rdt): será desenvolvido incrementalmente (lados do emissor e receptor)

No momento, somente transferência unidirecional

- Mas dados de controle fluem em ambas as direções.

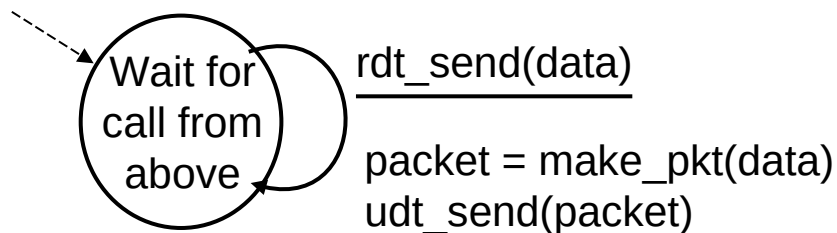
Usa máquinas de estado finito (FSM) para especificar os estados do emissor e do receptor



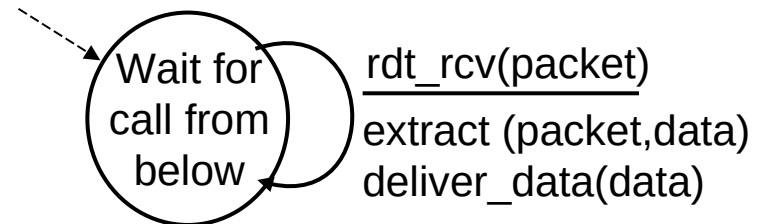


# Transferência confiável de dados sobre canal confiável: **rdt1.0**

- Canal perfeitamente confiável
  - Sem erros nos bits
  - Sem perda de pacotes
- FSMs para o emissor e para o receptor:
  - Emissor envia dados pelo canal
  - Receptor lê os dados do canal



sender



receiver

# Transferência confiável de dados sobre canal com erros: **rdt1.0**

Canal pode provocar erros (mudanças) em bits do pacote

- Campo *checksum* pode detectar erros em bits trocados

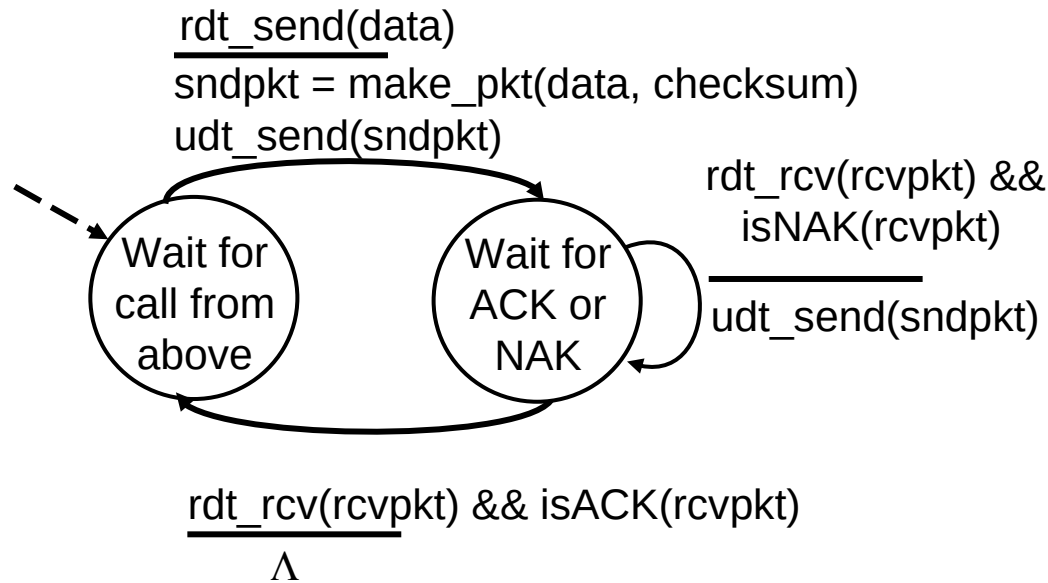
Questão: como se recuperar desses erros?

*Como seres humanos recuperam  
“erros” que ocorrem durante  
uma conversa?*

# Transferência confiável de dados sobre canal com erros: **rdt2.0**

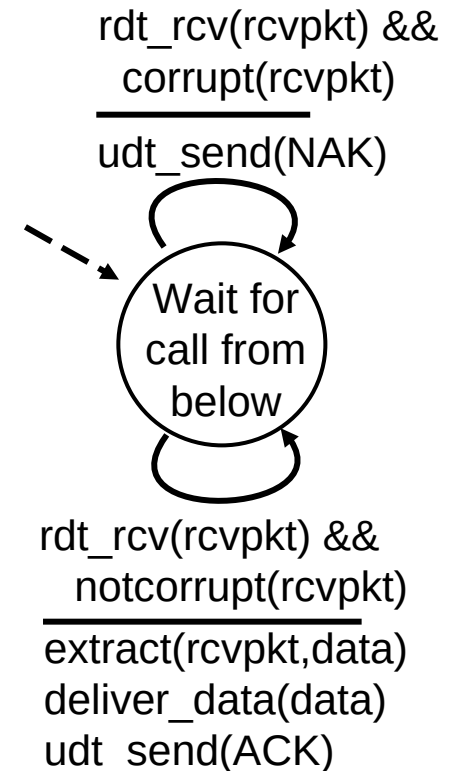
- Canal pode provocar erros (mudanças) em bits do pacote
  - Campo *checksum* pode detectar erros em bits trocados
- Questão: como se recuperar desses erros:
  - *acknowledgements (ACKs)*: receptor explicitamente informa ao emissor que pacote recebido estava OK
  - *negative acknowledgements (NAKs)*: receptor explicitamente informa ao emissor que pacote com erros
  - Emissor reenvia pacotes ao receber pacotes contendo **NAK**
- Novos mecanismos no **rdt2.0**:
  - detecção de erro
  - *feedback* do receptor: mensagens de controle (**ACK, NAK**) rcvr->sender

# rdt2.0: especificação usando FSM

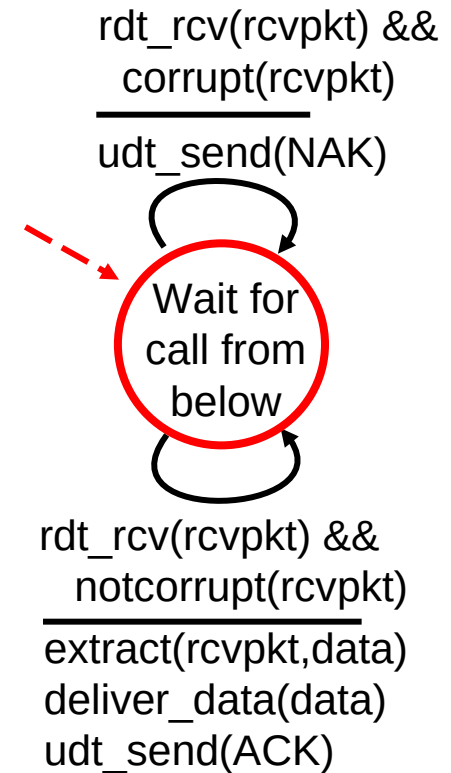
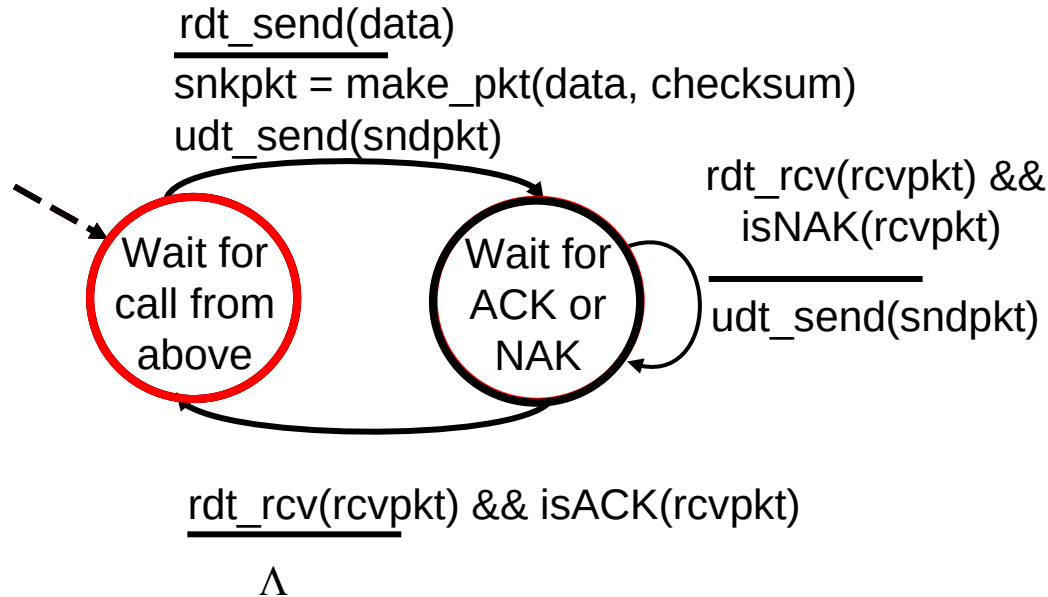


sender

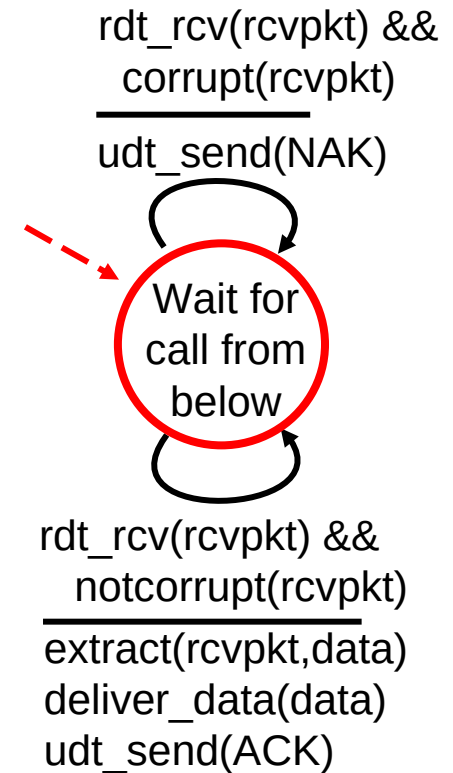
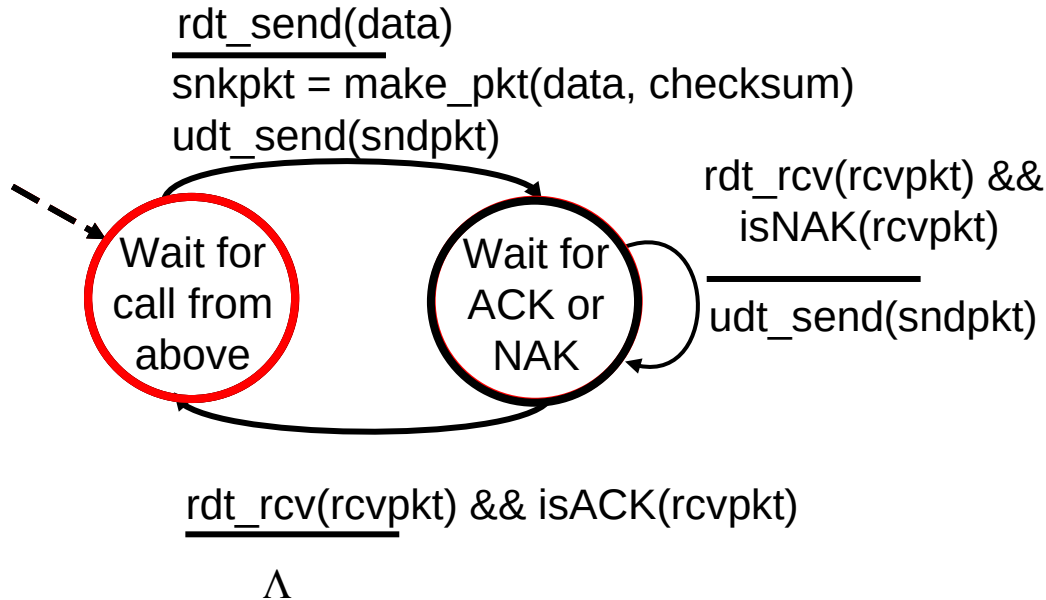
receiver



# rdt2.0: operação sem erros



# rdt2.0: cenário com erros



# rdt2.0 possui uma falha

O que acontece se mensagens ACK/NAK chegarem corrompidas?

- O emissor não sabe o que aconteceu no lado do destinatário.
- Não pode retransmitir, pois pode gerar pacotes duplicados.

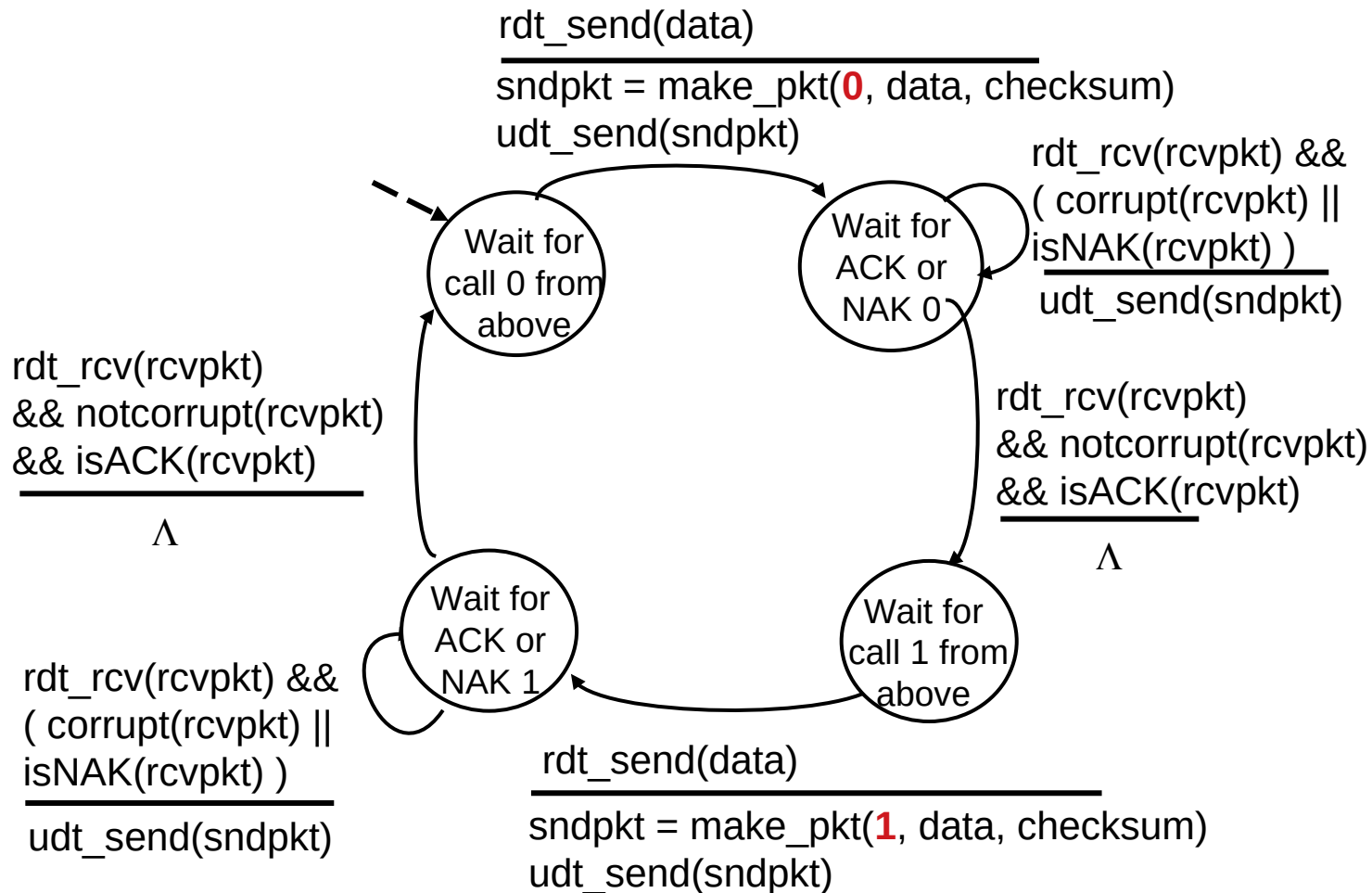
Como tratar duplicatas:

- O remetente retransmite o pacote corrente se um ACK ou NAK for corrompido
- Emissor adiciona um *número de sequência* em cada pacote
- O receptor descarta pacotes recebidos duplicados

## *stop-and-wait*

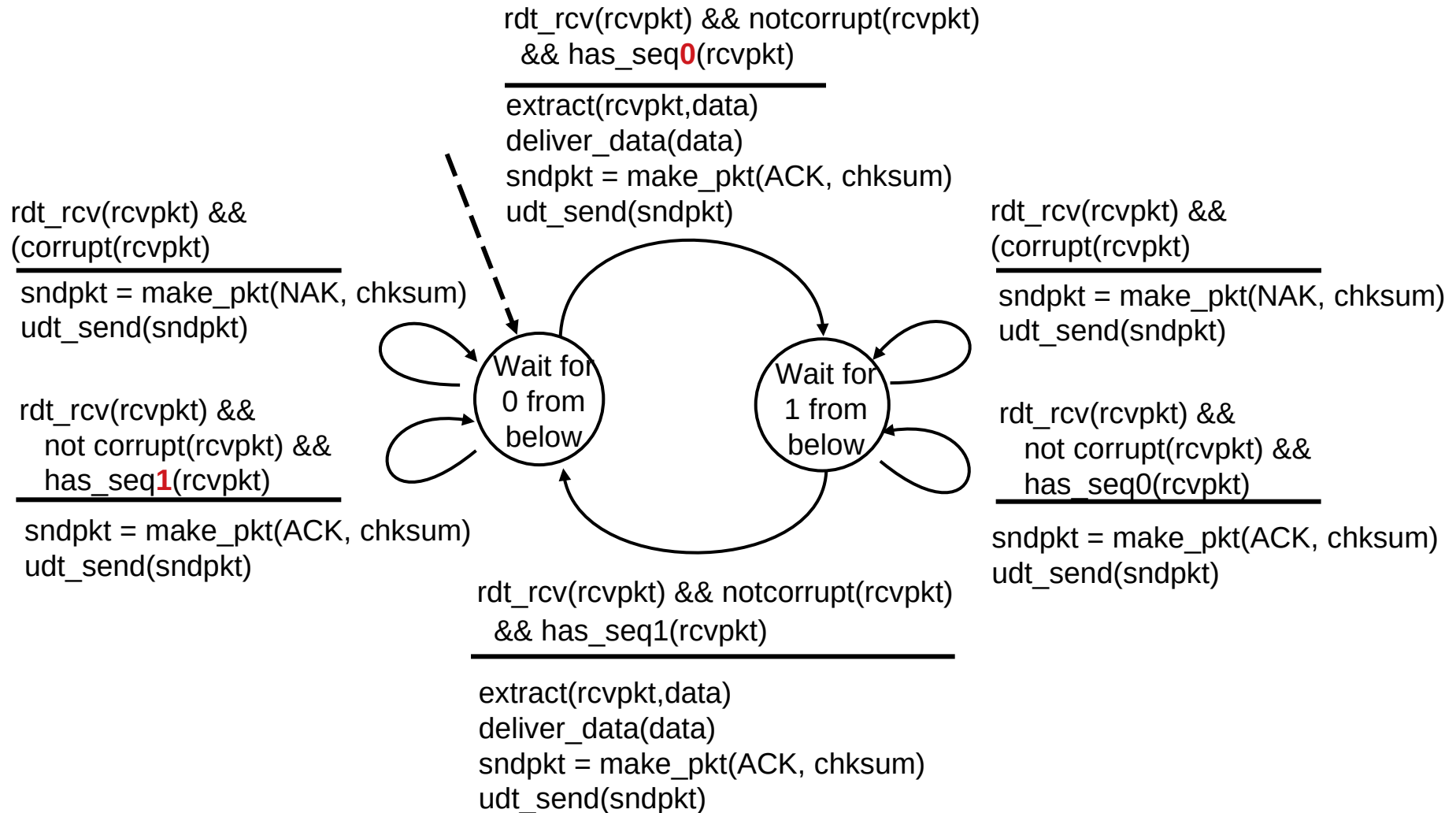
Emissor envia um pacote e então entra no estado de espera, aguardando mensagem de resposta do receptor (mensagem de controle).

## rdt2.1: remetente (trata pacotes ACK/NAKs corrompidos)





## rdt2.1: destinatário (trata pacotes ACK/NAKs corrompidos)



## rdt2.1: remetente (trata pacotes ACK/NAKs corrompidos)

### remetente:

- Adiciona número seq ao **pkt**
- Dois números (0,1) serão suficientes?
- Precisa checar se recebeu pacotes ACK/NAK corrompidos
- 0 dobro de estados
  - Estados precisam ser lembrados para permitir que o protocolo espere o próximo pacote correto (ordem)

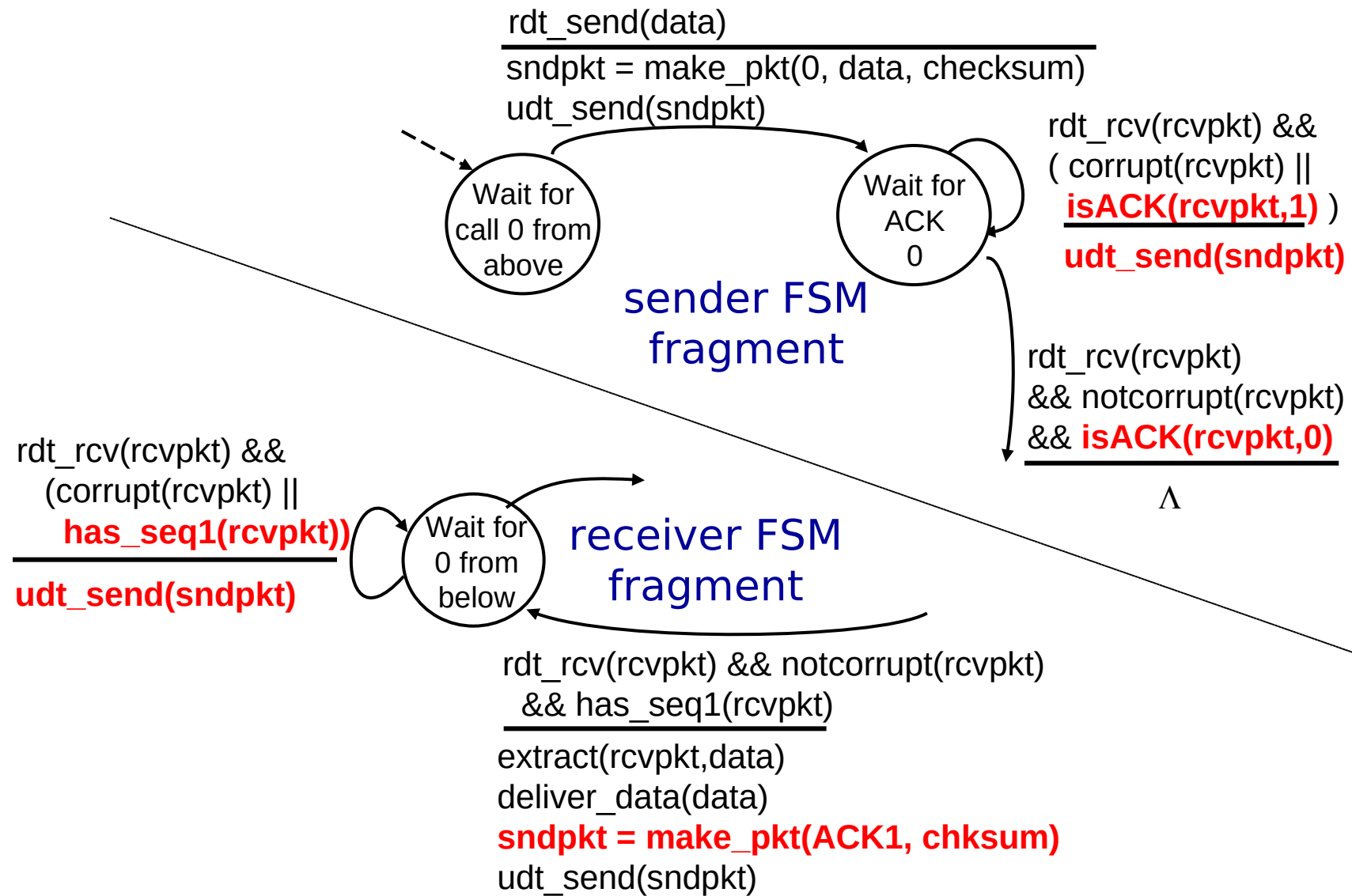
### destinatário:

- Precisa checar se o pacote recebido não foi duplicado
  - 0 estado indica se 0 ou 1 é esperado como núm de pacote
- Nota: não há formas do destinatário "saber" se o receptor recebeu os últimos ACK/NAK

# rdt2.2: protocolo sem NAKs

- Mesma funcionalidade do protocolo rdt2.1, mas somente com ACKs
- Em vez de NAK, o destinatário envia um ACK para o último pacote recebido sem erros
  - A resposta precisa explicitamente incluir o número de sequência relativo ao ACK
- **ACKs duplicados** no remetente resultam na mesma ação que NAK: **retransmitir o pacote corrente**

# rdt2.2: fragmentos do *sender* e *receiver*



# rdt3.0: canais com erros e perdas

## Nova questão

O canal pode **perder pacotes** (dados, ACKs)

Os mecanismos anteriores já não são suficientes:

checksum

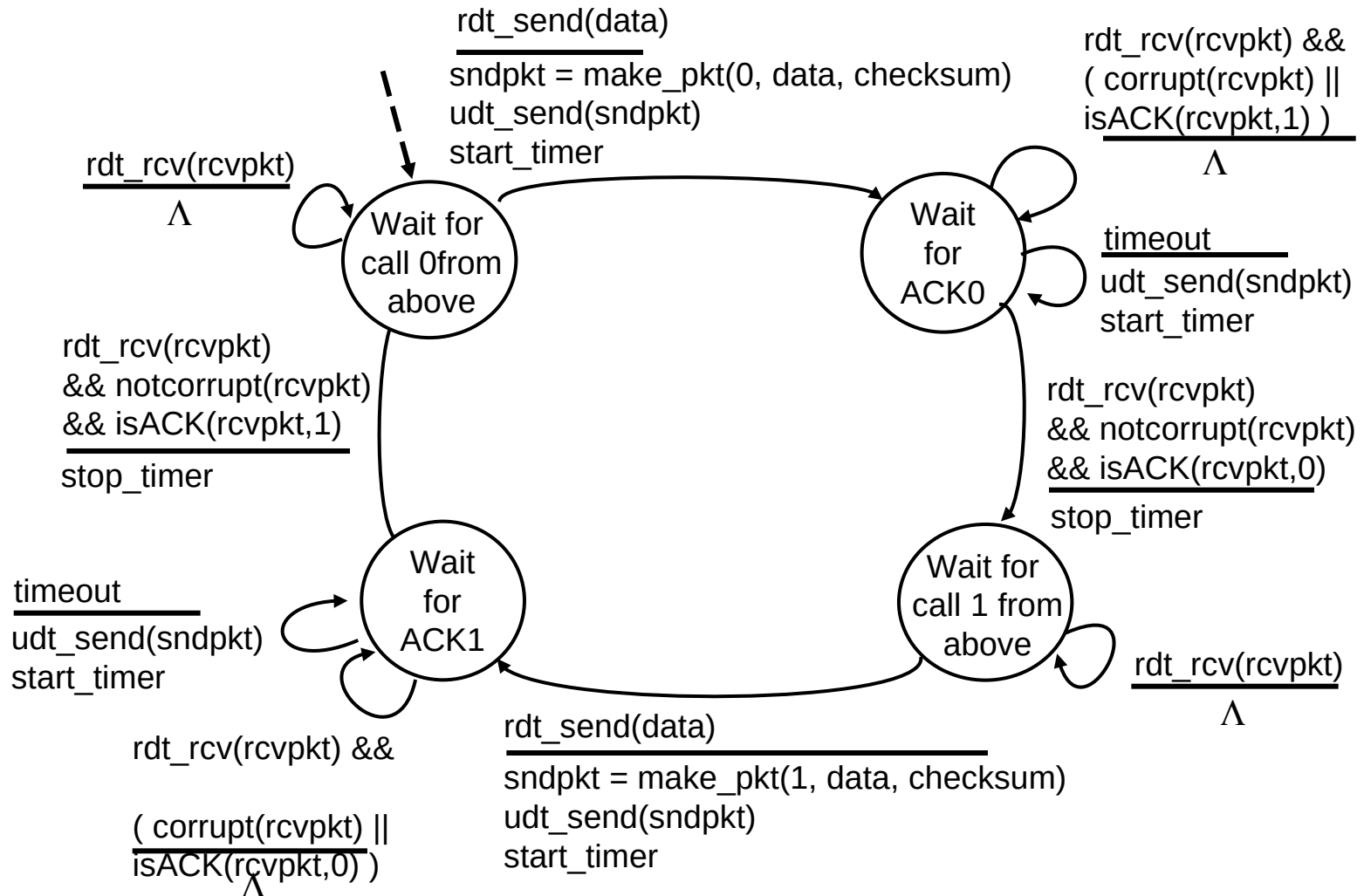
seq. #

ACKs

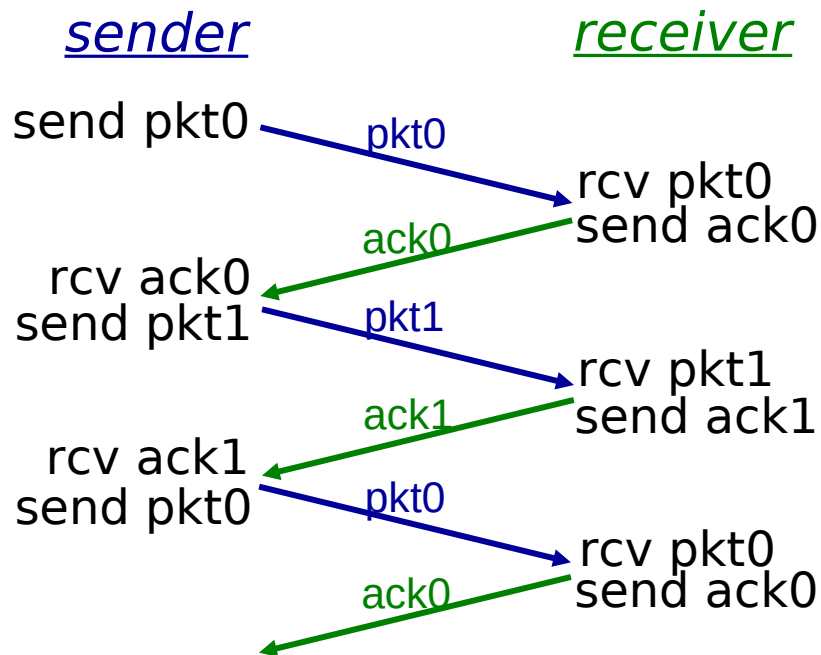
retransmissões

- Abordagem: o remetente espera uma quantidade de tempo razoável para o recebimento do ACK
- Retransmite se nenhum ACK chegou neste tempo
  - Se o pacote (ou ACK) acabou de ser enviado (não perdido)
    - Retransmissão será duplicada: números de sequência tratarão disso
    - **Receptor precisa especificar o número de sequência do pacote sendo reconhecido (ACKed)**
  - Requer temporizador de contagem regressiva (*countdown timer*)

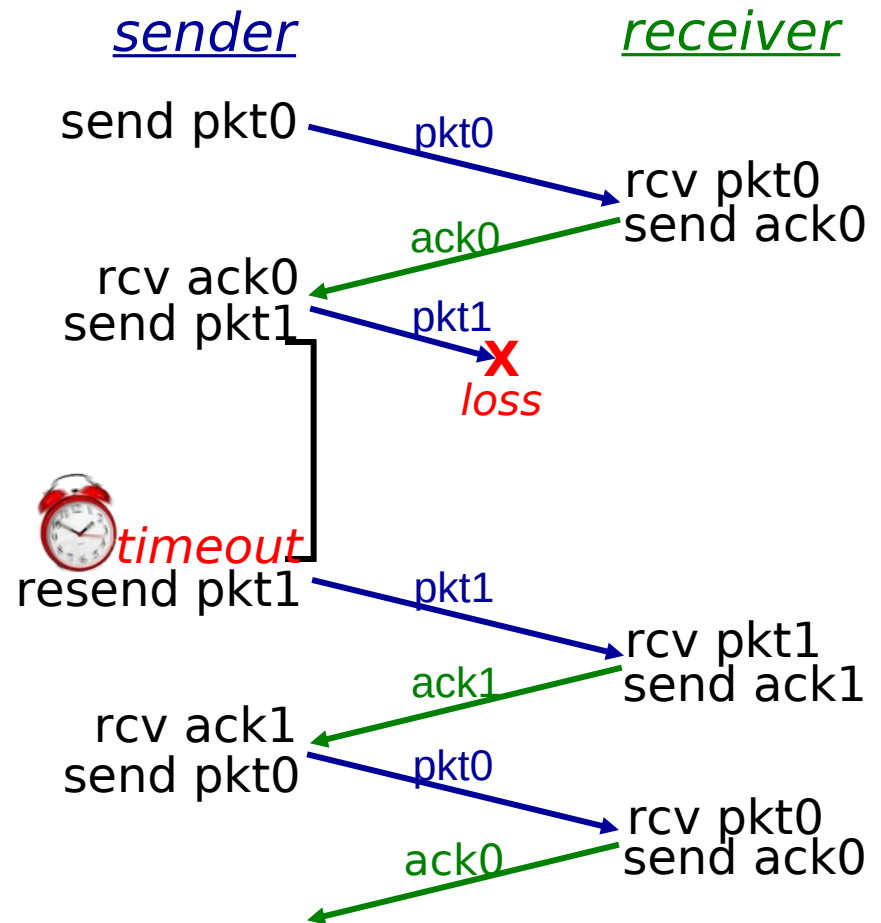
# rdt3.0: remetente



# rdt3.0: funcionamento

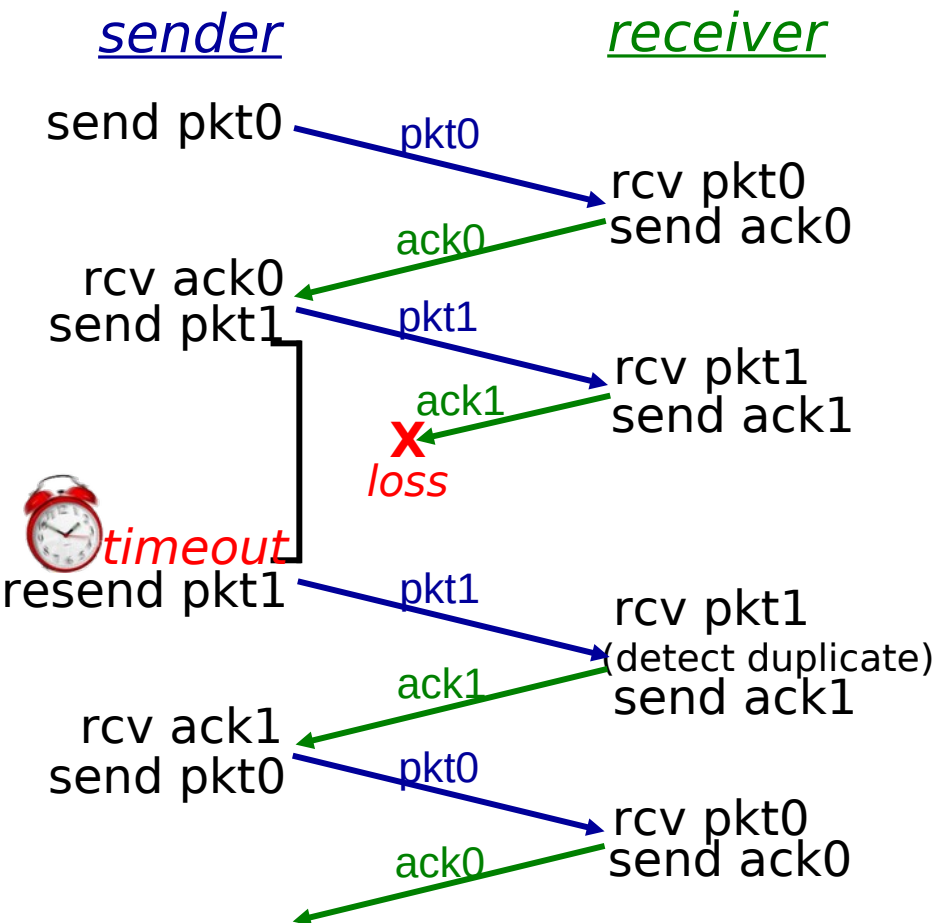


(a) sem perda  
de pacotes

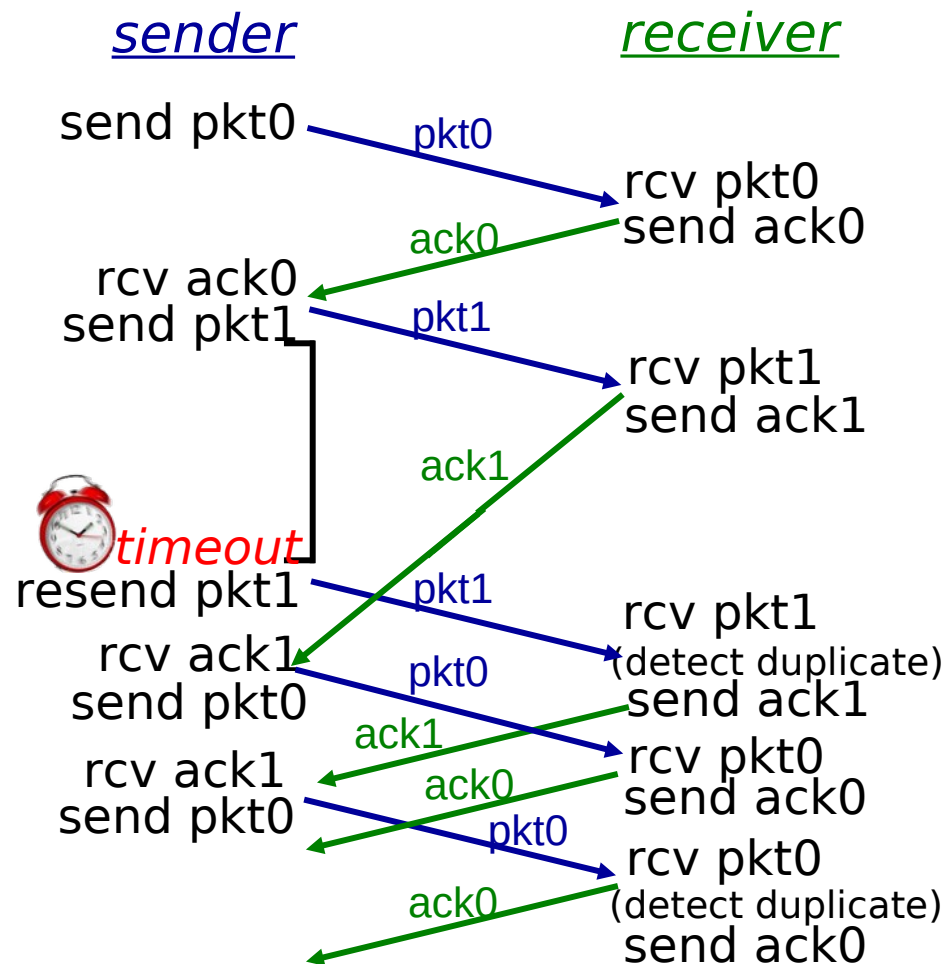


(b) com perda de pacotes

# rdt3.0: funcionamento



(c) ACK perdido. Reenvio do pacote.



(d) Timeout prematuro, pois não houve perda de pacote, apenas atraso do ACK.



# rdt3.0: performance

- rdt3.0 está correto, mas a performance é ruim
- P. ex.: enlace de 1 Gbps, 15 ms *propagation delay*, pacote com tamanho de 8000 bits:

$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \mu s$$

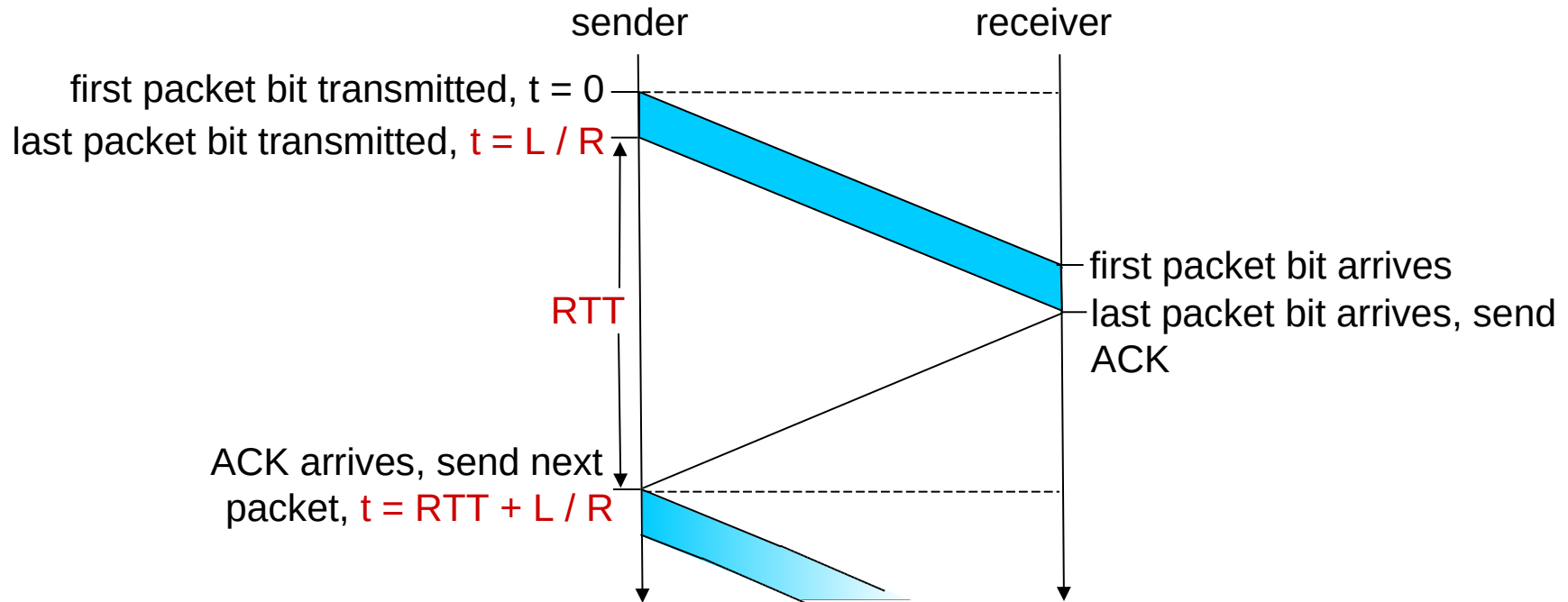
- $U_{sender}$ : **uso** - fração do tempo do remetente (*busy sending*)

$$U_{sender} = \frac{L/R}{RTT + L/R} = \frac{.008}{30.008} = 0.00027$$

- if RTT=30ms, 1KB pkt a cada 30ms: 33kB/s vazão (*throughput*) sobre um enlace de 1 Gbps

Obs.: um protocolo de rede limitando o uso dos recursos físicos da rede

# rdt3.0: operação *stop-and-wait*

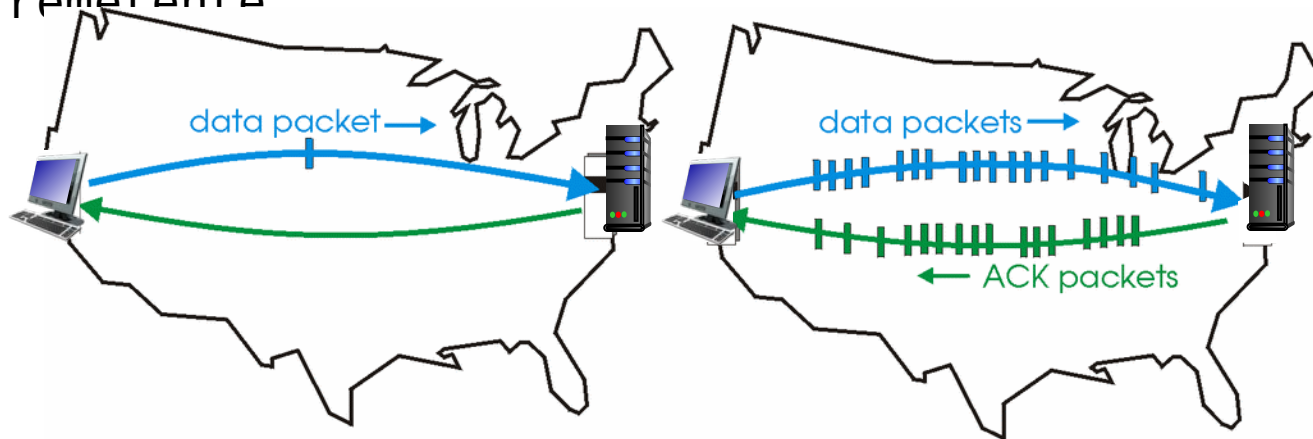


$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

# Protocolos *pipelined* (paralelizados)

**Pipelining:** remetente sempre permite envios múltiplos, mesmo que ainda não reconhecidos

- O tamanho dos números de sequência precisam ser aumentados
- Necessidade de buffers tanto no destinatário quanto no remetente



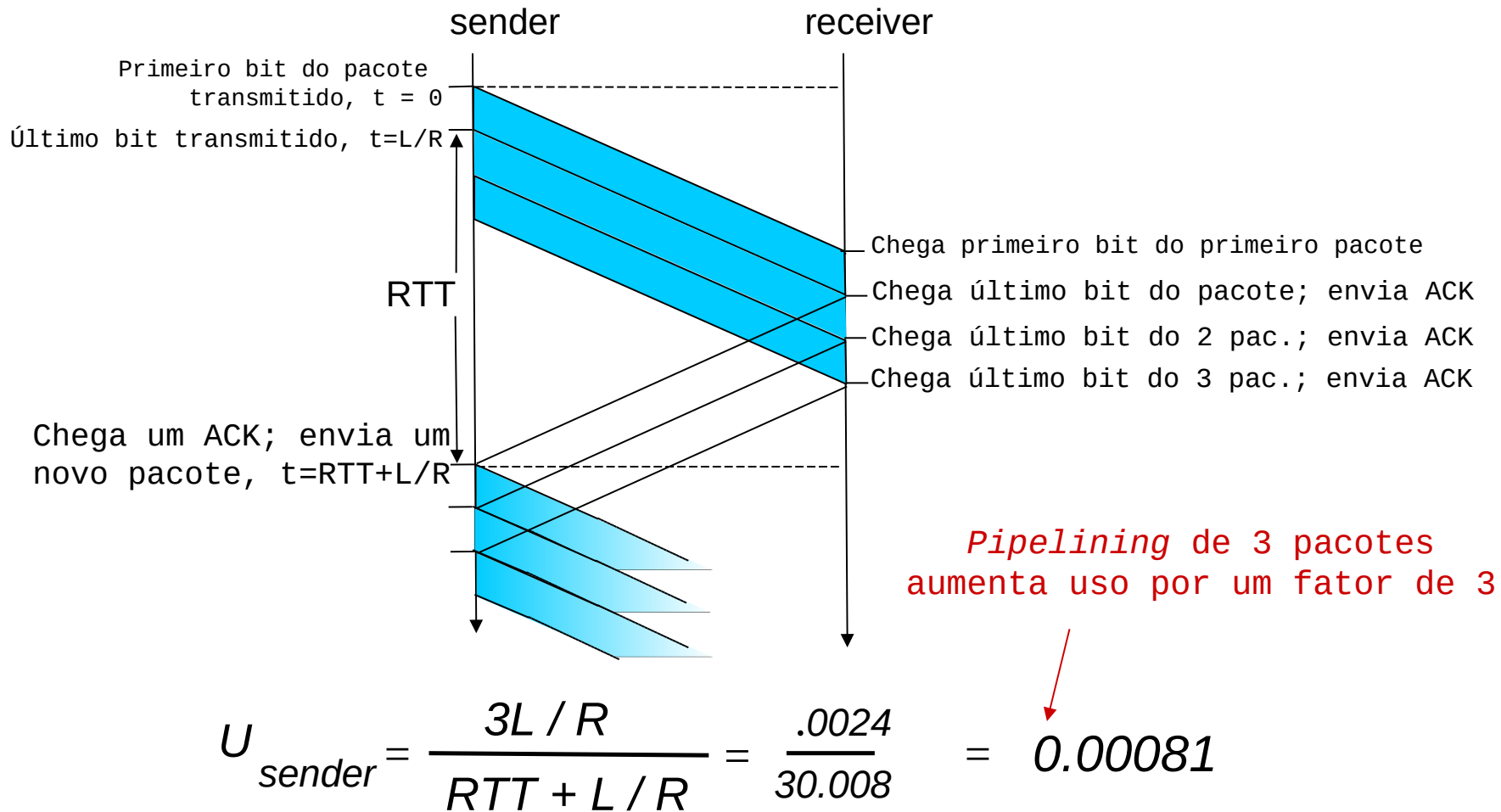
(a) a stop-and-wait protocol in operation

(b) a pipelined protocol in operation

Dois protocolos genéricos:

- *Go-Back-N*
- Repetição seletiva (*selective repeat- SR*)

# Pipelining: aumento da utilização do canal



Observação: cada pacote ainda registra o mesmo tempo de transmissão e de propagação. A vazão aumentada é devido à quantidade de pacotes transmitidos por unidade tempo.

# Resumo dos mecanismos para confiança

**Checksum**: detecção de erros (UDP e TCP não corrigem: o UDP descarta e o TCP usa mecanismos de retransmissão)

**Timer**: usado para retransmissão de pacotes após um *timeout*, pois um pacote ou seu ACK podem ter sido perdidos no canal. Podem ocorrer duplicatas quando não ocorre perda, mas atraso maior que o timeout.

**Número de sequência**: usado para numerar sequencialmente um conjunto de pacotes que fluem do transmissor para o receptor. Permite detectar pacotes perdidos e duplicados.

# Resumo dos mecanismos para confiança

**Reconhecimento** (*Acknowledgment*): permite o receptor informar ao transmissor o recebimento correto de um conjunto de pacotes. No reconhecimento é informado o número de sequência para indicar quais dados estão sendo reconhecidos. O reconhecimento pode ser individual ou coletivo.

**Reconhecimento NEGATIVO** (*Negative Acknowledgment*): informa ao receptor que um conjunto de dados referenciado por um número de sequência não foi recebido corretamente. Resolve questões de erros nos dados recebidos.

# Pipelining e mecanismo de janelas

**Pipelining**: o remetente envia pacotes de dados com números de sequência dentro de determinados valores. Pode ser permitido enviar múltiplos pacotes de dados sem confirmação de recebimentos dos anteriores.

O mecanismo de **janelas (*window*)** (controlado por números de sequência) permitem resolver dois problemas:

- controle de **fluxo**: capacidade do receptor em receber uma determinada qtd de dados
- controle de **congestionamento** da rede

# sumário

Serviços de transporte

Multiplexação e demultiplexação

Transporte sem conexão: UDP

Princípios de transporte com confiança

**Transporte orientado a conexão: TCP**

- **Estrutura do segmento**

- **Transferência de dados confiável**

- **Controle de fluxo**

- **Gerenciamento da conexão**

Princípios de controle de congestionamento

Controle de congestionamento no TCP



# TCP: *visão geral*

RFCs: 793,1122,1323, 2018, 2581

- **Protocolo ponto a ponto:** um remetente e um destinatário
- ***Stream* de bytes confiável e em ordem:** sem fronteiras
- **Paralelismo (*pipelined*):** controle de congestionamento e controle de fluxo através de tamanho de **janela**
- **Serviço de dados *full-duplex*:** fluxo de dados nas duas direções, usando a mesma conexão. Limitado pelo MSS (*maximum segment size*)
- **Orientado a conexão:** apresentação (*handshaking*), na qual há troca de mensagens para iniciar os estados do remetente e do destinatário, antes de iniciar a troca de dados
- **Controle de fluxo:** o remetente não sobrecarregará o buffer do destinatário

# TCP: *visão geral*

RFCs: 793,1122,1323, 2018, 2581

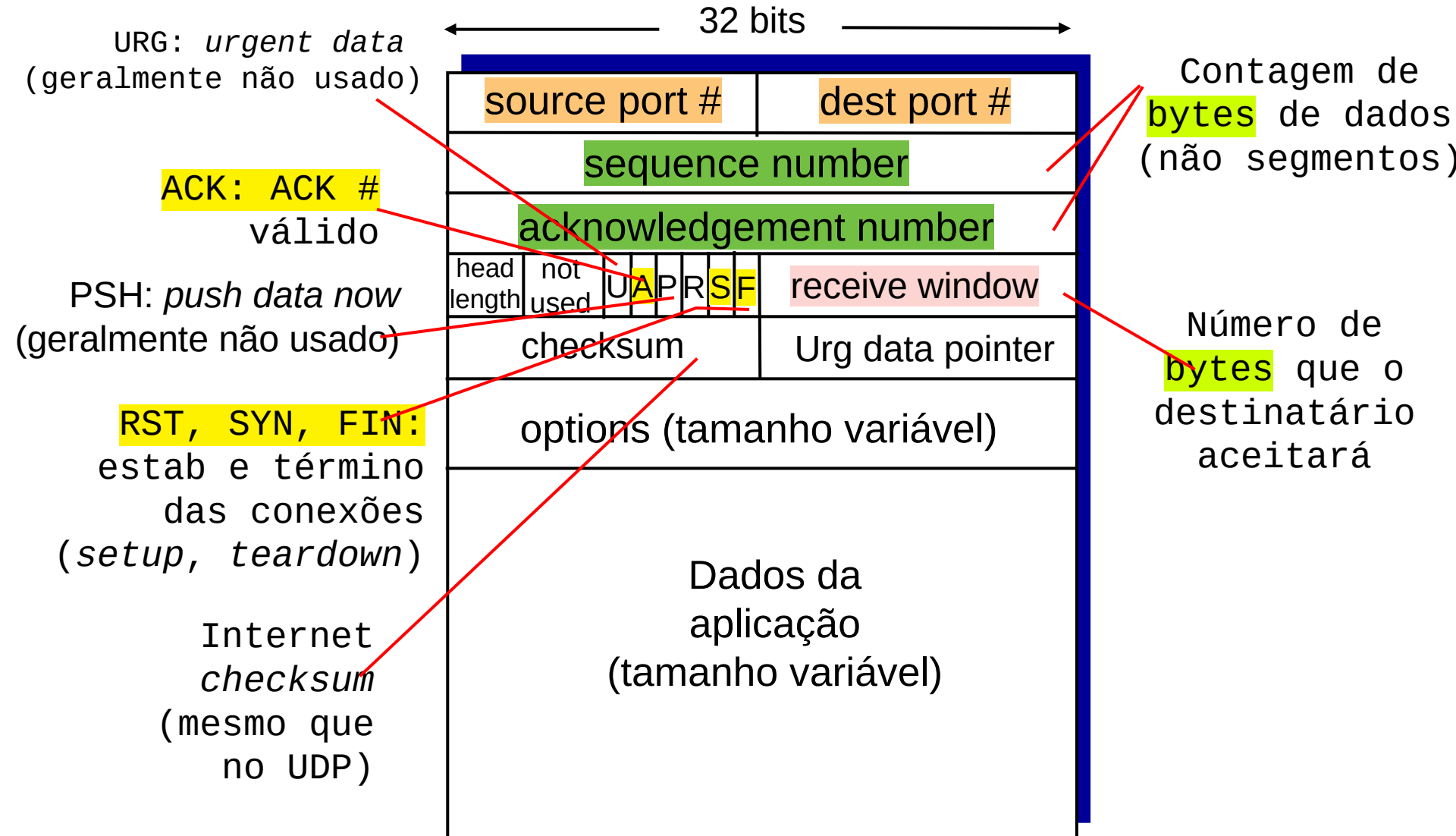
## Controle de fluxo

O receptor controla o remetente, de forma que o remetente não enviará dados que “estourem” o *buffer* do receptor. O processo é fazer o remetente diminuir a taxa de envio.

## Controle de congestionamento

A rede (protocolo IP) não fornece controle de congestionamento, de forma que o TCP implementa os mecanismos para que a rede não sature (uma rede saturada gera perda de pacotes e degradação).

# TCP: estrutura do segmento



# TCP: números sequência e ACKs

## Núm Sequência (32bits)

Número do octeto no *stream* remetente (número do primeiro byte no segmento de dados)

## Núm Reconhecimento (32bits)

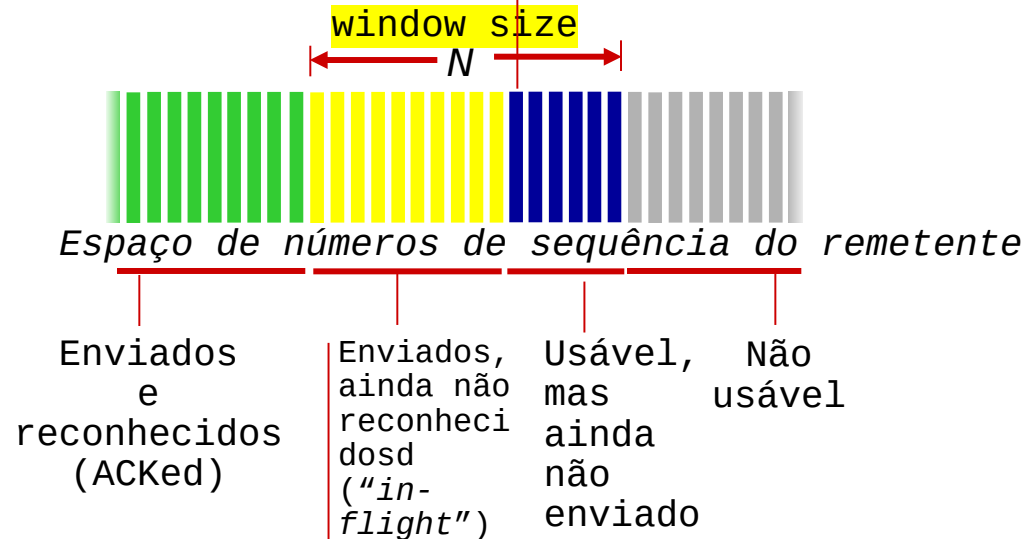
Número do próximo octeto que o receptor espera receber do remetente

**Q:** como o receptor trata segmentos fora de ordem

- **A:** a especificação TCP não indica – a cargo do implementador

Segmento saindo do remetente

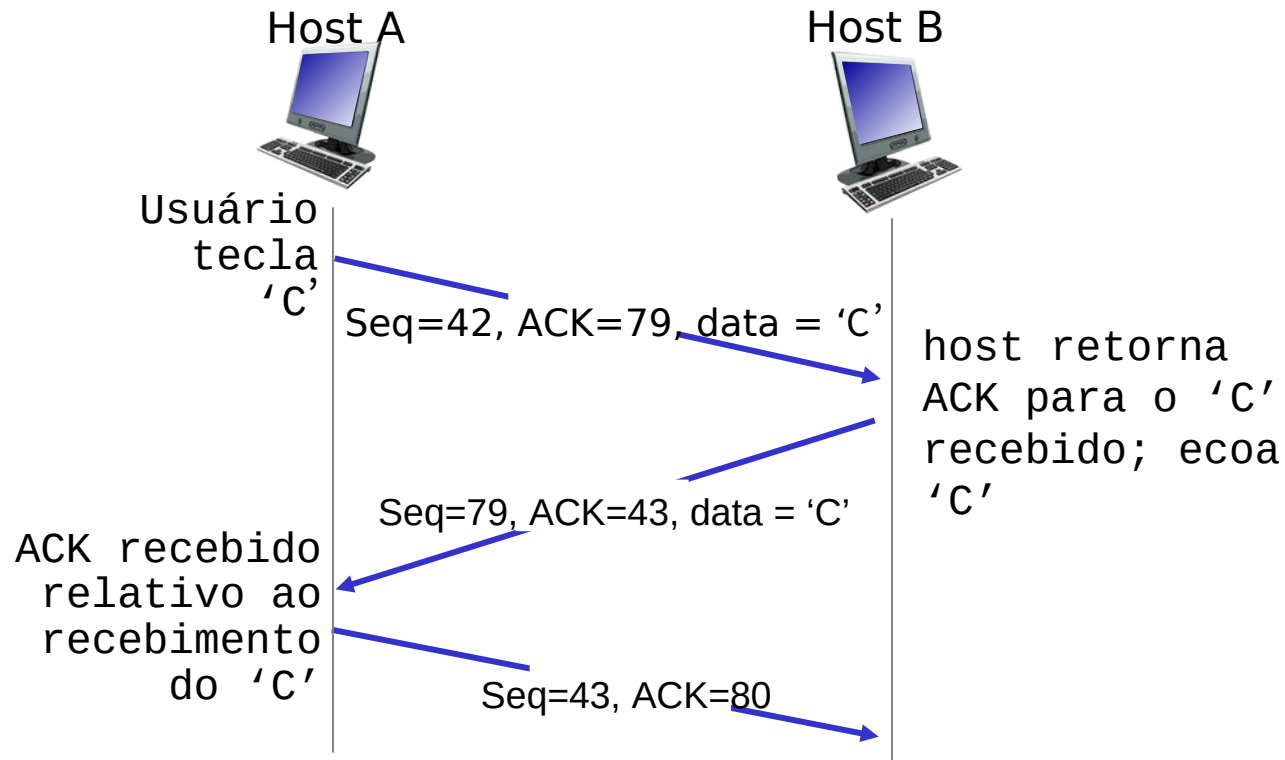
source port #	dest port #
sequence number	
acknowledgement number	
	rwnd
checksum	urg pointer



Segmento chegando ao remetente

source port #	dest port #
sequence number	
acknowledgement number	
	rwnd
checksum	urg pointer

# TCP: números sequência e ACKs



Cenário do protocolo Telnet

# TCP: RTT (*round trip time*) e *timeout*

Q: como e qual valor deve ser o *timeout*?

- Maior que o RTT, mas o **RTT varia**
- *Muito curto:* *timeout* prematuro, resultando em retransmissões desnecessárias
- *Muito longo:* o TCP reage lentamente à perda de segmentos

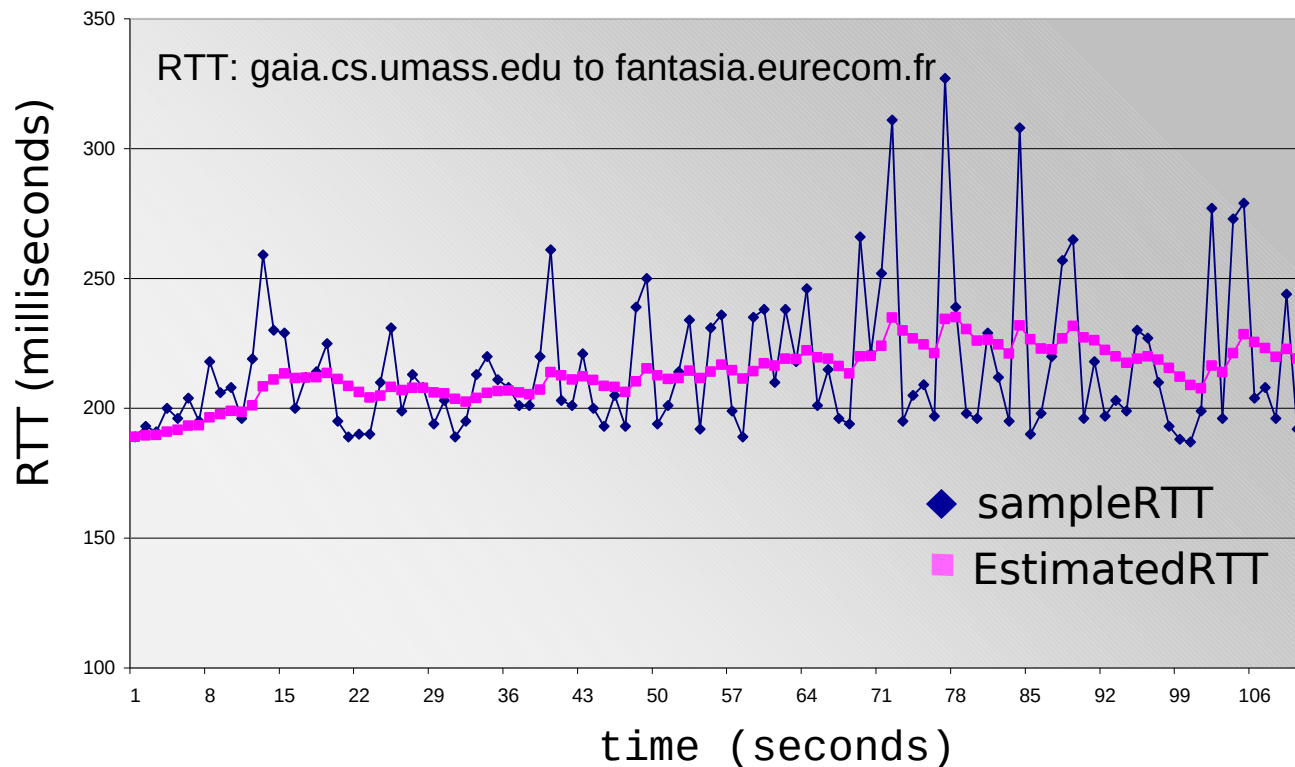
Q: Como determinar o RTT?

- **SampleRTT:** medida do tempo em que um segmento sai até receber um ACK; ignora retransmissões
- **SampleRTT** vai variar, de forma que se deve estimar uma média
  - Média de várias medidas recentes, não somente a última medida**SampleRTT**

# TCP: RTT (*round trip time*) e *timeout*

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

- Valor típico:  $\alpha = 0.125$




# TCP: RTT (*round trip time*) e *timeout*

- **Intervalo *timeout*:** **EstimatedRTT** mais uma margem de confiança "*safety margin*"
  - Variações grandes em **EstimatedRTT** -> margem de confiança maior
- Estimar desvio SampleRTT a partir de EstimatedRTT:

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

(tipicamente,  $\beta = 0.25$ )

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

 ↑  
estimated RTT ↑  
“safety margin”



# sumário

Serviços de transporte

Multiplexação e demultiplexação

Transporte sem conexão: UDP

Princípios de transporte com confiança

**Transporte orientado a conexão: TCP**

- Estrutura do segmento

- **Transferência de dados confiável**

- Controle de fluxo

- Gerenciamento da conexão

Princípios de controle de congestionamento

Controle de congestionamento no TCP

# TCP: transferência confiável de dados (rdt)

TCP cria um serviço rdt no topo do serviço IP não confiável

- Paralelização (*pipelined*) de segmentos
- ACKs cumulativos
- Um único timer para retransmissão

As **retransmissões** disparadas por:

- Eventos de *timeout*
- ACKs duplicados

Para o exemplo a seguir, considerar um remetente TCP mais simples que:

- Ignora ACKs duplicados, e
- Ignora controle de fluxo e controle de congestionamento

# TCP: eventos no remetente:

## *Dados recebidos da aplicação:*

- Criação do segmento com **seq #**
- seq # é o número do primeiro byte do segmento referente ao *stream* de dados
- Inicia o relógio se não estiver rodando
  - O temporizador é o mais antigo segmento não reconhecido
  - Intervalo de expiração:  
`TimeoutInterval`

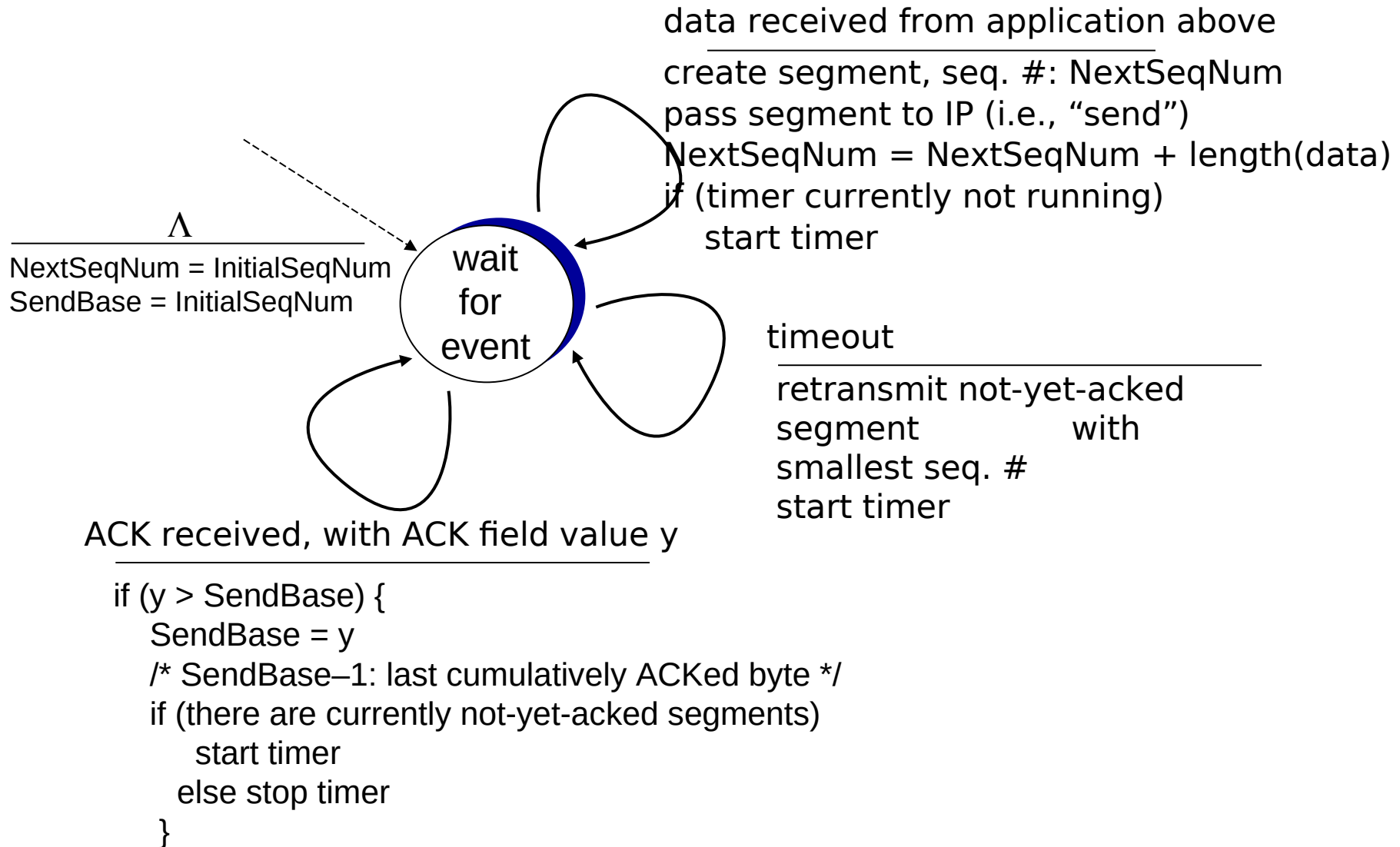
## *Evento timeout:*

- Retransmite segmento que causou *timeout*
- Reinicia o temporizador

## *Evento de recebimento de ACK:*

- Se o ACK reconhece um segmento anteriormente não reconhecido
  - Atualiza o segmento que foi reconhecido
  - Inicia o temporizador se ainda há segmentos sem reconhecimento

# TCP: remeteente (simplificado)



# sumário

Serviços de transporte

Multiplexação e demultiplexação

Transporte sem conexão: UDP

Princípios de transporte com confiança

**Transporte orientado a conexão: TCP**

- Estrutura do segmento
- Transferência de dados confiável
- **Controle de fluxo**
- Gerenciamento da conexão

Princípios de controle de congestionamento

Controle de congestionamento no TCP

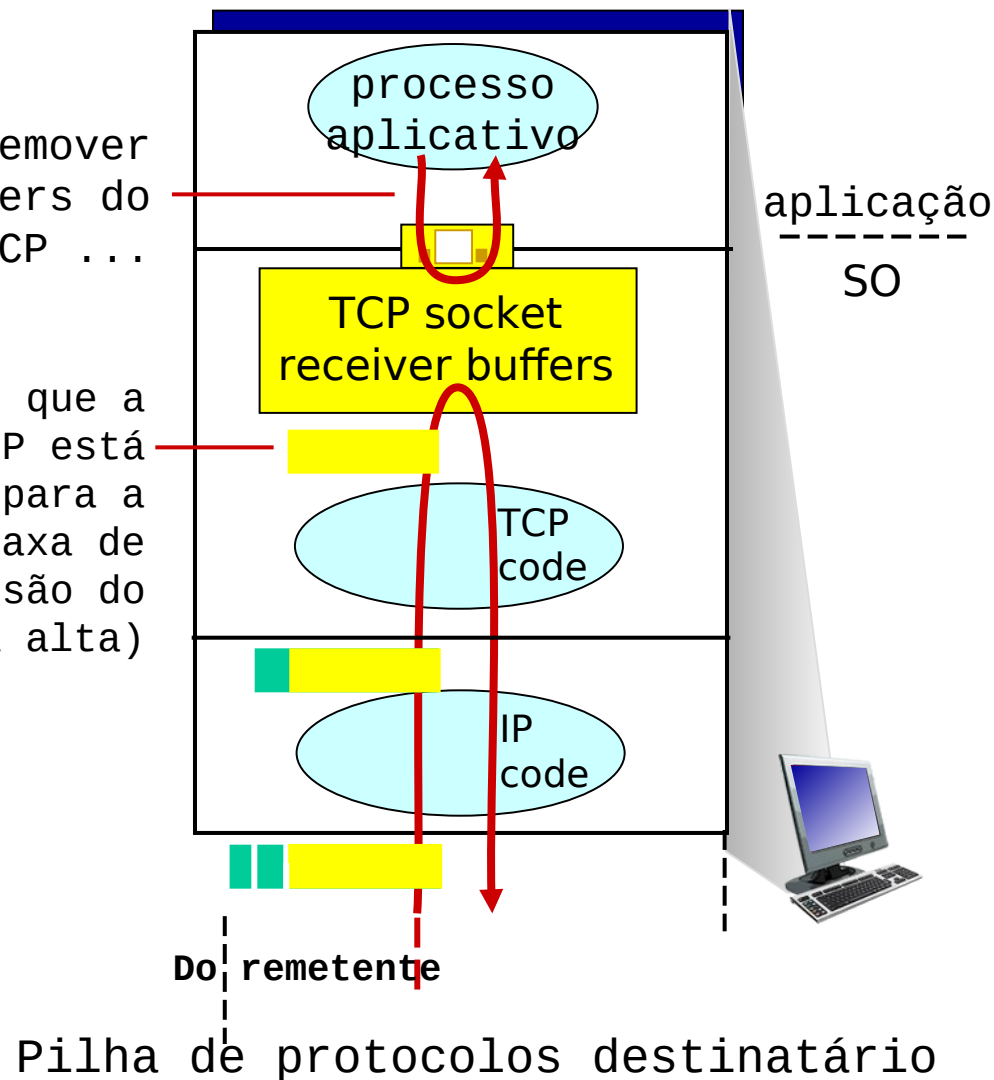
# TCP: controle de fluxo

Aplicação pode remover dados dos buffers do soquete TCP ...

... mais lento do que a pilha TCP está despachando para a aplicação (a taxa de transmissão do remetente está alta)

## Controle Fluxo

O destinatário controle o remetente, de forma que o emissor não sobrecarregará (*overflow*) o buffer do receptor com taxa de transmissão muito alta.



# sumário

Serviços de transporte

Multiplexação e demultiplexação

Transporte sem conexão: UDP

Princípios de transporte com confiança

**Transporte orientado a conexão: TCP**

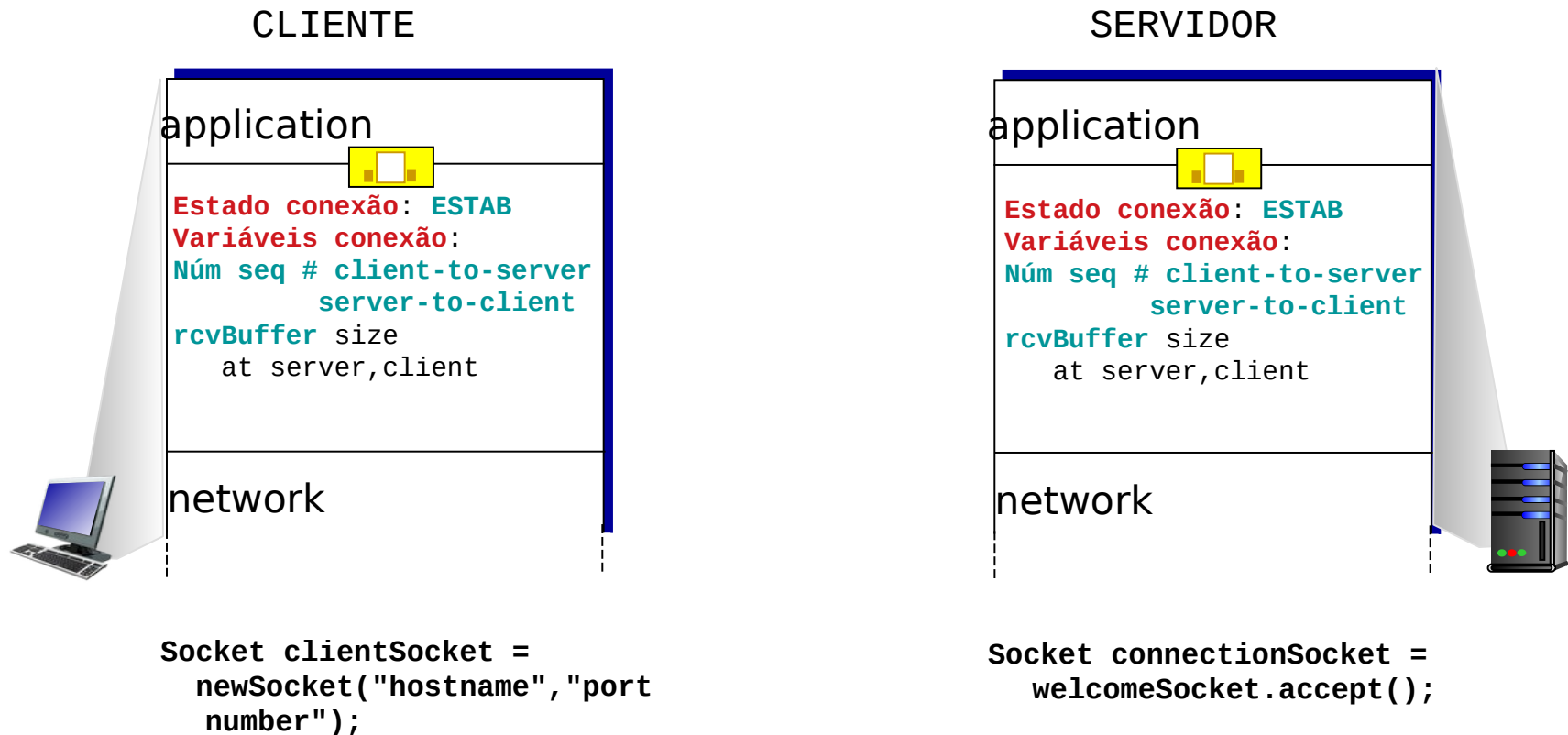
- Estrutura do segmento
- Transferência de dados confiável
- Controle de fluxo
- **Gerenciamento da conexão**

Princípios de controle de congestionamento

Controle de congestionamento no TCP

# Gerenciamento da conexão

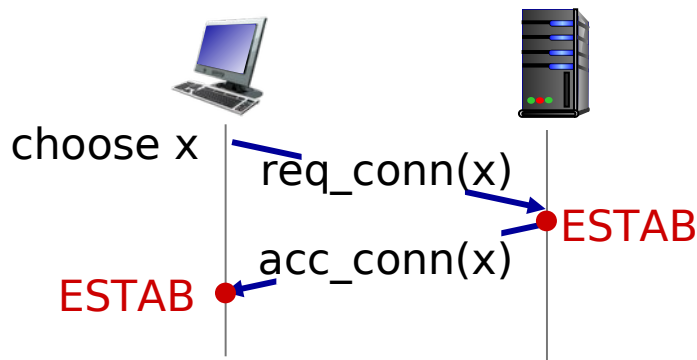
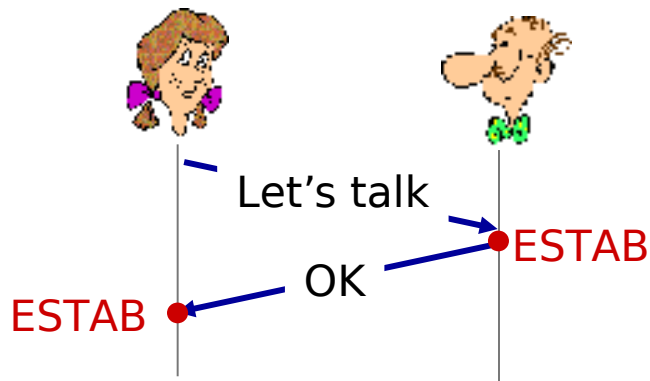
Antes de enviar dados, remetente e receptor devem realizar o “*handshake*” (**protocolo de apresentação**): acordo de ambas as partes em criar a conexão e acordo nos parâmetros





# Acordo para estabelecer uma conexão

*Handshake de 2-vias:*



Q: Como o handshake 2-way trabalha numa rede?

- Delays variados
- Mensagens de retransmissão devido perda de pacotes
- Mensagens de reordenamento
- O outro lado não pode "ver"

# TCP: *handshake* de 3-vias (3-way)

Estado  
do  
cliente

Estado  
do  
Servidor



Inicia Conexão

LISTEN

SYNSENT

Escolhe **init seq num=x**,  
e o envia com TCP SYN

SYNbit=1, Seq=x

Escolhe um **init seq num=y**  
e o envia junto com  
TCP SYNACK;  
Também reconhece  
x recebido

SYN RCVD

SYNbit=1, Seq=y  
ACKbit=1; ACKnum=x+1

ESTAB

Recebido SYNACK(x)  
indica servidor ativo;  
envio de ACK para SYNACK;  
este segmento pode  
conter dados  
do cliente para o servidor

ACKbit=1, ACKnum=y+1

O recebimento ACK(y)  
indica cliente ativo

ESTAB

# TCP: *handshake* de 3-vias (3-way)

Estado  
do  
cliente

Estado  
do  
Servidor



Inicia Conexão

LISTEN

SYNSENT

Escolhe **init seq num=x**,  
e o envia com TCP SYN

SYNbit=1, Seq=x

Escolhe um **init seq num=y**  
e o envia junto com  
TCP SYNACK;  
Também reconhece  
x recebido

SYN RCVD

SYNbit=1, Seq=y  
ACKbit=1; ACKnum=x+1

ESTAB

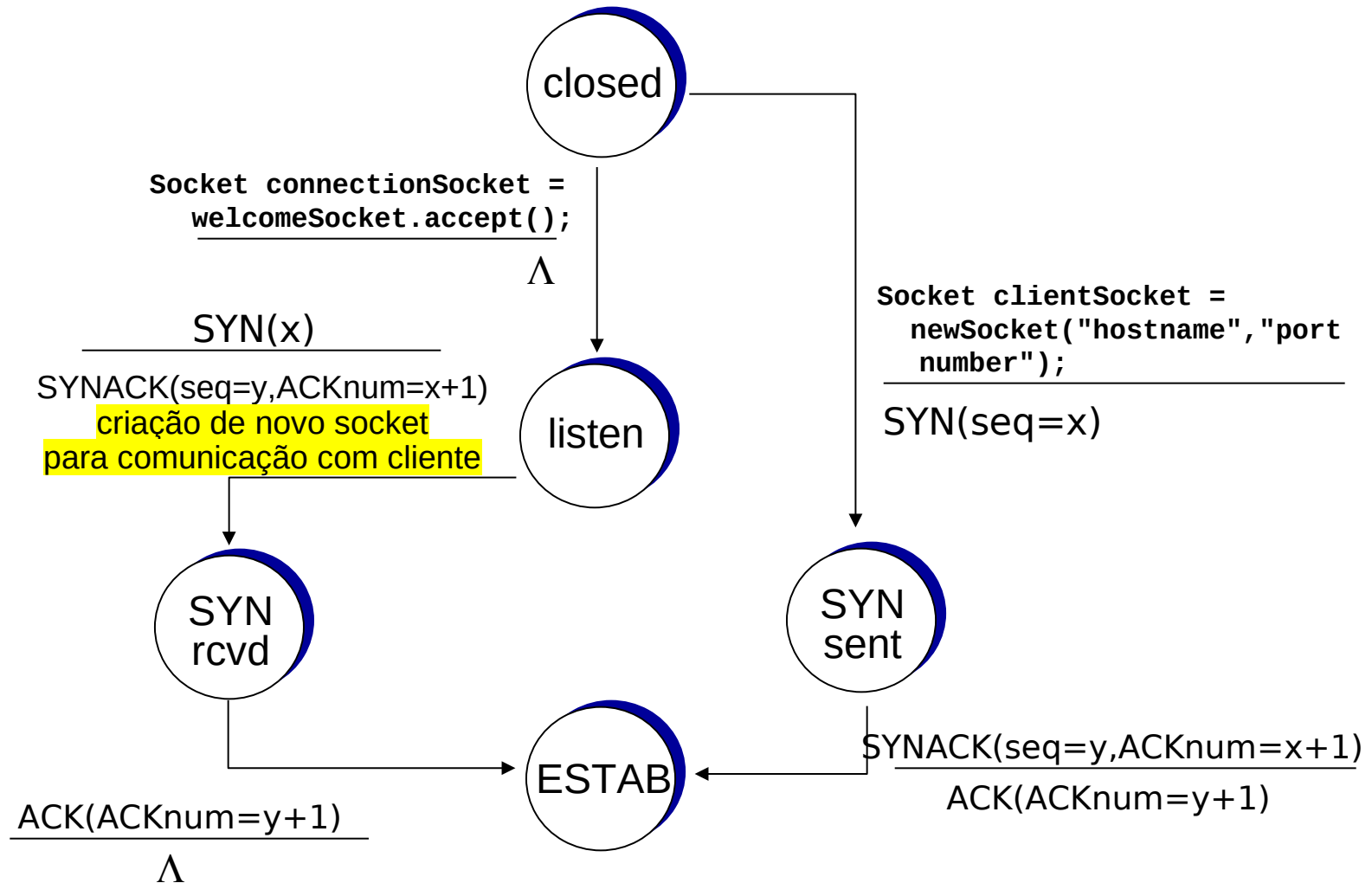
Recebido SYNACK(x)  
indica servidor ativo;  
envio de ACK para SYNACK;  
este segmento pode  
conter dados  
do cliente para o servidor

ACKbit=1, ACKnum=y+1

O recebimento ACK(y)  
indica cliente ativo

ESTAB

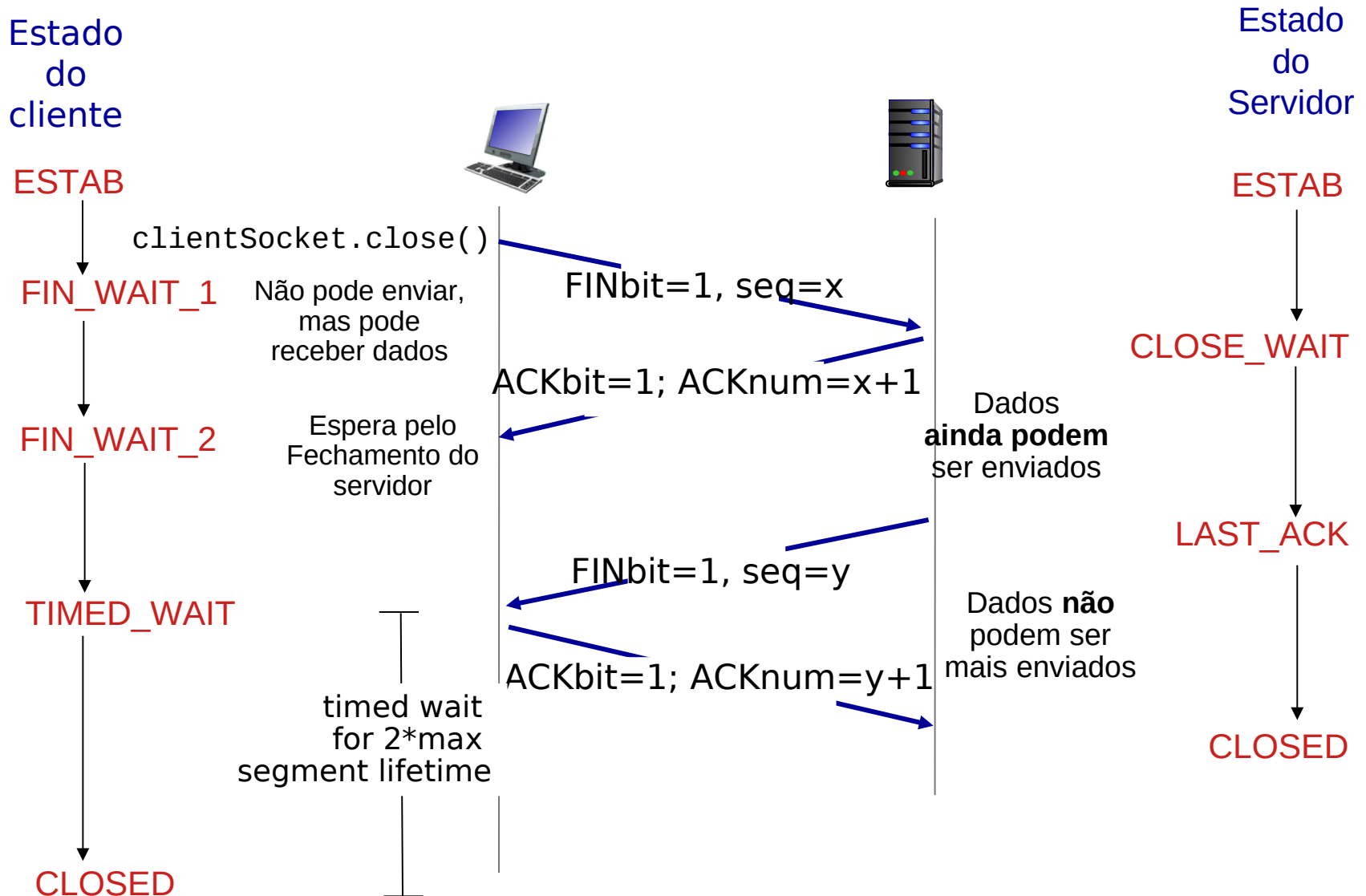
# TCP: *handshake* de 3-vias - FSM



# TCP: fechando a conexão

- Cada qual (cliente e servidor) fecham o seu lado da conexão
  - Envio de segmentos com o bit FIN=1 ativado
- Cada par responde com um ACK ao FIN recebido
  - Ao receber um FIN, o ACK pode ser combinado com seu próprio FIN
- Pode haver troca de FIN simultâneos

# TCP: fechando a conexão



# sumário

Serviços de transporte

Multiplexação e demultiplexação

Transporte sem conexão: UDP

Princípios de transporte com confiança

**Transporte orientado a conexão: TCP**

- Estrutura do segmento
- Transferência de dados confiável
- Controle de fluxo
- Gerenciamento da conexão

**Princípios de controle de congestionamento**

Controle de congestionamento no TCP

# Princípios de controle de congestionamento

## *Congestionamento:*

Informalmente: “muitas origens enviando muitos dados de forma tão rápida que a **rede** não consegue manipular.”

Diferente do controle de fluxo.

Manifesta-se por:

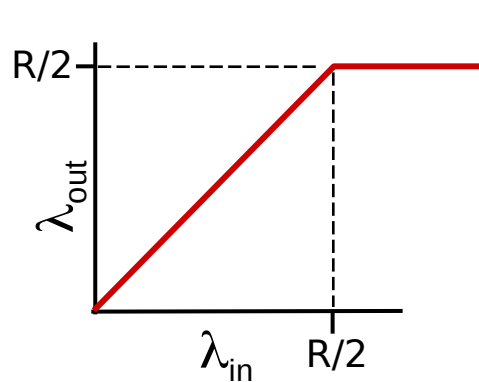
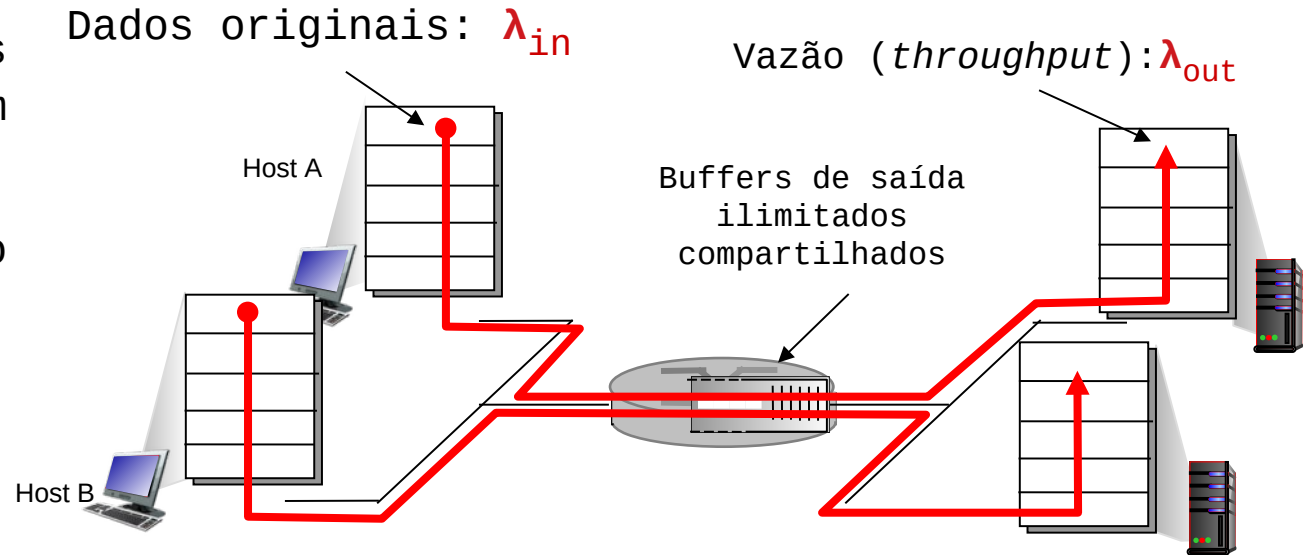
**Perda de pacotes** (os *buffers* estouram nos roteadores: *overflow*)

**Longos atrasos** (enfileiramento nos *buffers* dos roteadores)

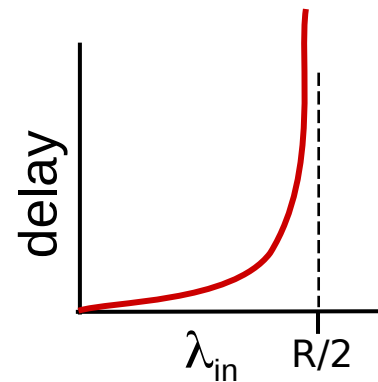


# Congestionamento: causas e custos

- Dois remetentes e dois destinatários
- Um roteador com um *buffer* infinito
- Capacidade do enlace de saída do roteador: **R**
- Sem retransmissão



Desempenho máximo por conexão:  $R/2$



Grandes atrasos quando a taxa de chegada ( $\lambda_{in}$ ) se aproxima da capacidade

# Congestionamento: causas e custos

Um roteador com *buffers finitos*

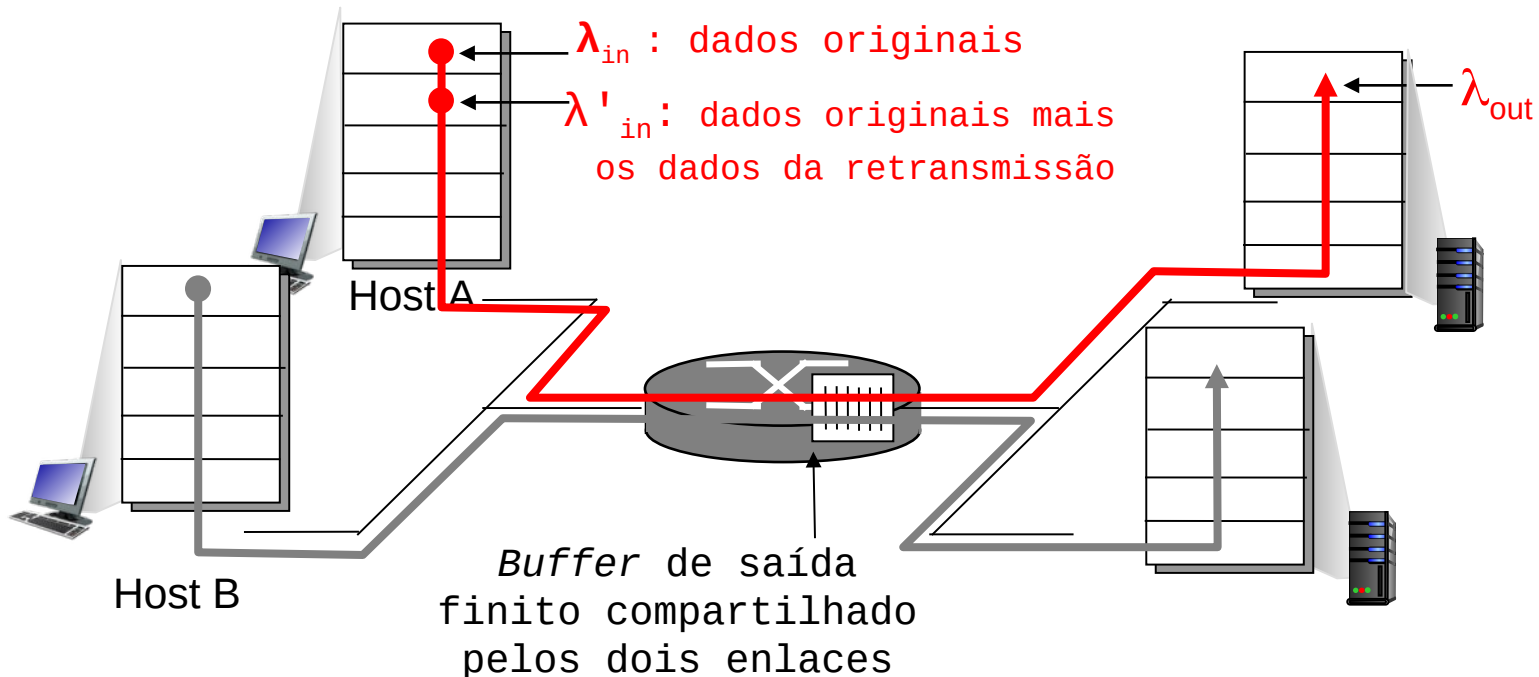
Remetente retransmite pacotes em caso de *timeout*

- Entrada camada-aplicação = saída camada-aplicação:

$$\lambda_{in} = \lambda_{out}$$

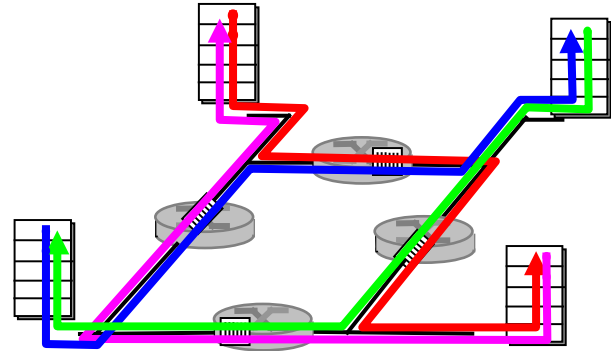
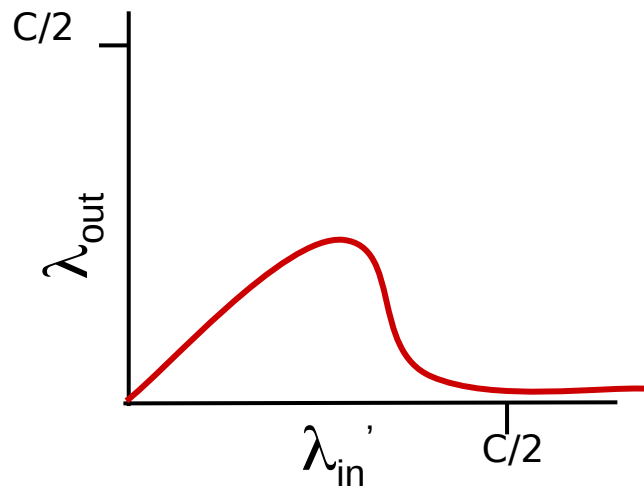
- Entrada camada-transporte inclui retransmissões:

$$\lambda'_{in} \geq \lambda_{in}$$



# Congestionamento: causas e custos

Outro custo do congestionamento: quando um pacote é descartado pelo roteador, capacidade de transmissão deve ser usada para contornar essa falha (retransmissões)



# sumário

Serviços de transporte

Multiplexação e demultiplexação

Transporte sem conexão: UDP

Princípios de transporte com confiança

**Transporte orientado a conexão: TCP**

- Estrutura do segmento
- Transferência de dados confiável
- Controle de fluxo
- Gerenciamento da conexão

Princípios de controle de congestionamento

**Controle de congestionamento no TCP**

# TCP: controle congestionamento

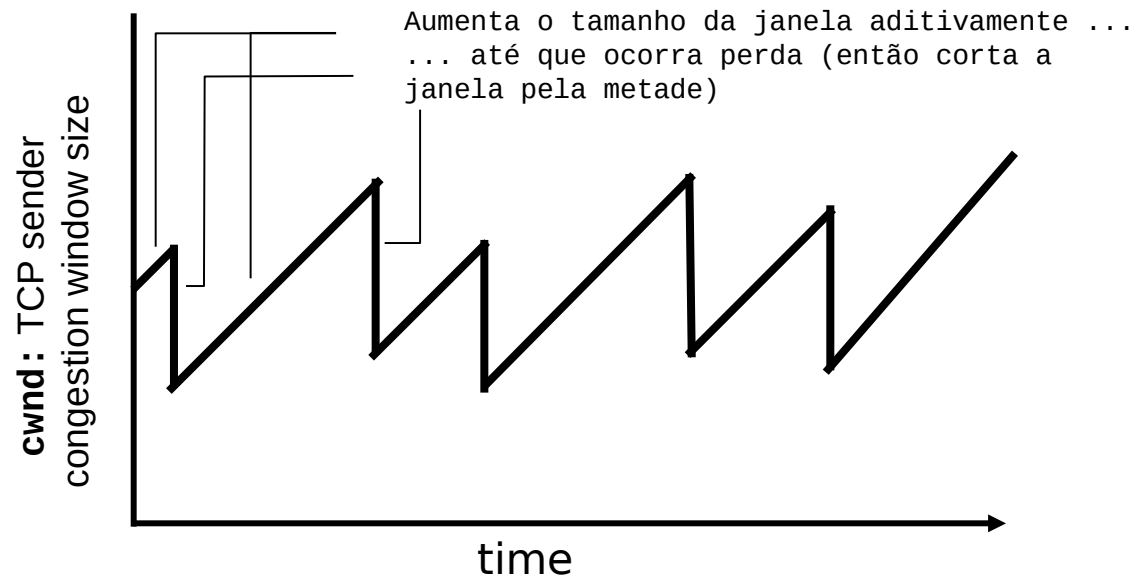
## AIMD: *Additive Increase Multiplicative Decrease*

**Abordagem:** remetente aumenta a taxa de transmissão (tamanho janela), tentando descobrir (*probing*) a banda de rede disponível, até que um evento de perda ocorra.

**Aumento aditivo:** aumenta o tamanho de **cwnd** por 1 MSS a cada RTT, até que uma perda de segmento ocorra.

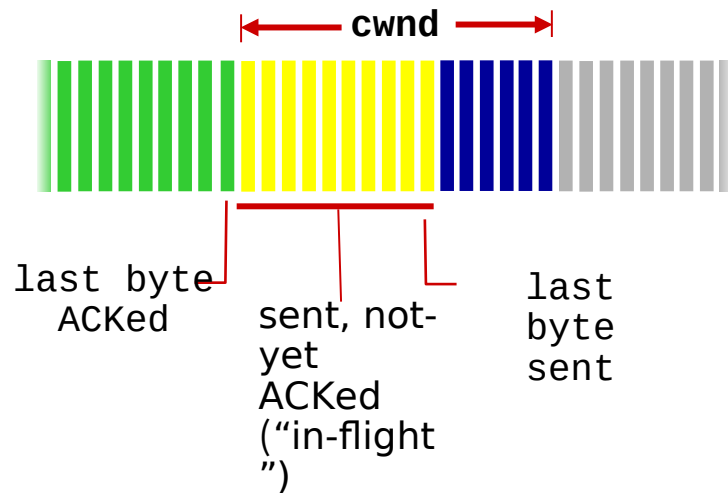
**Diminuição multiplicativa:** corta a janela **cwnd** pela metade a cada evento de perda de segmento.

AIMD comportamento  
**dente de serra:**  
Sondando por  
banda de rede  
disponível.



# TCP: detalhes do Controle de Congestionamento

*Espaço de números de sequência remetente*



Remetente limita a transmissão:

$$\text{LastByteSent} - \text{LastByteAcked} \leq \text{cwnd}$$

**cwnd** é dinâmico e é função do congestionamento da rede percebido pelo remetente.

*TCP – razão de envio:*

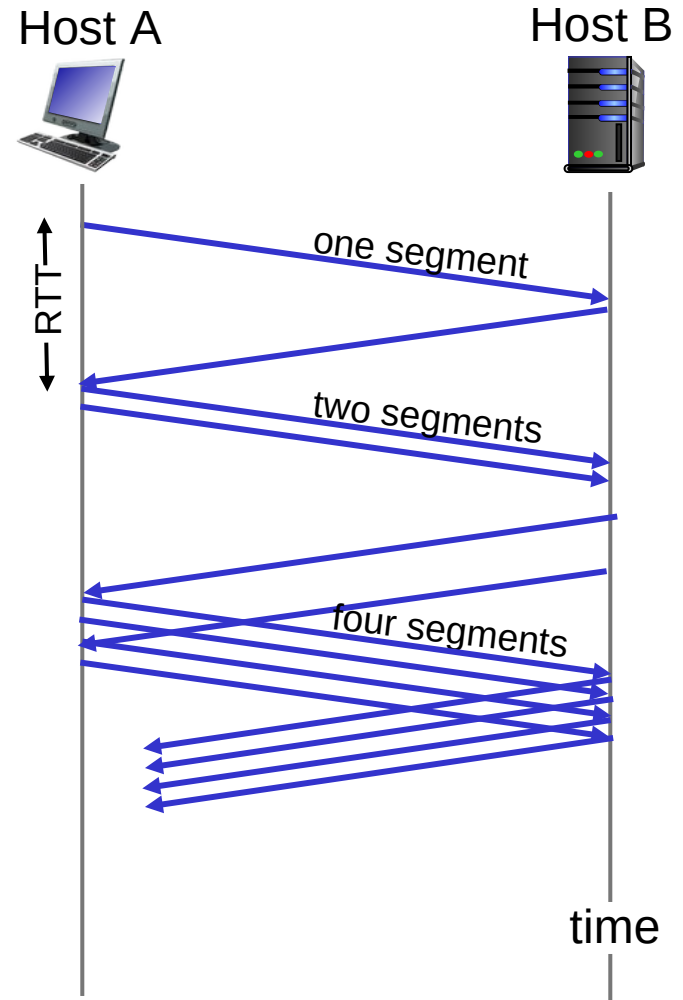
- *Basicamente:* envia **cwnd** bytes, espera tempo RTT pelos ACKs, então envia mais bytes.

$$\text{razão} \approx \frac{\text{cwnd}}{\text{RTT}} \text{ bytes/s}$$

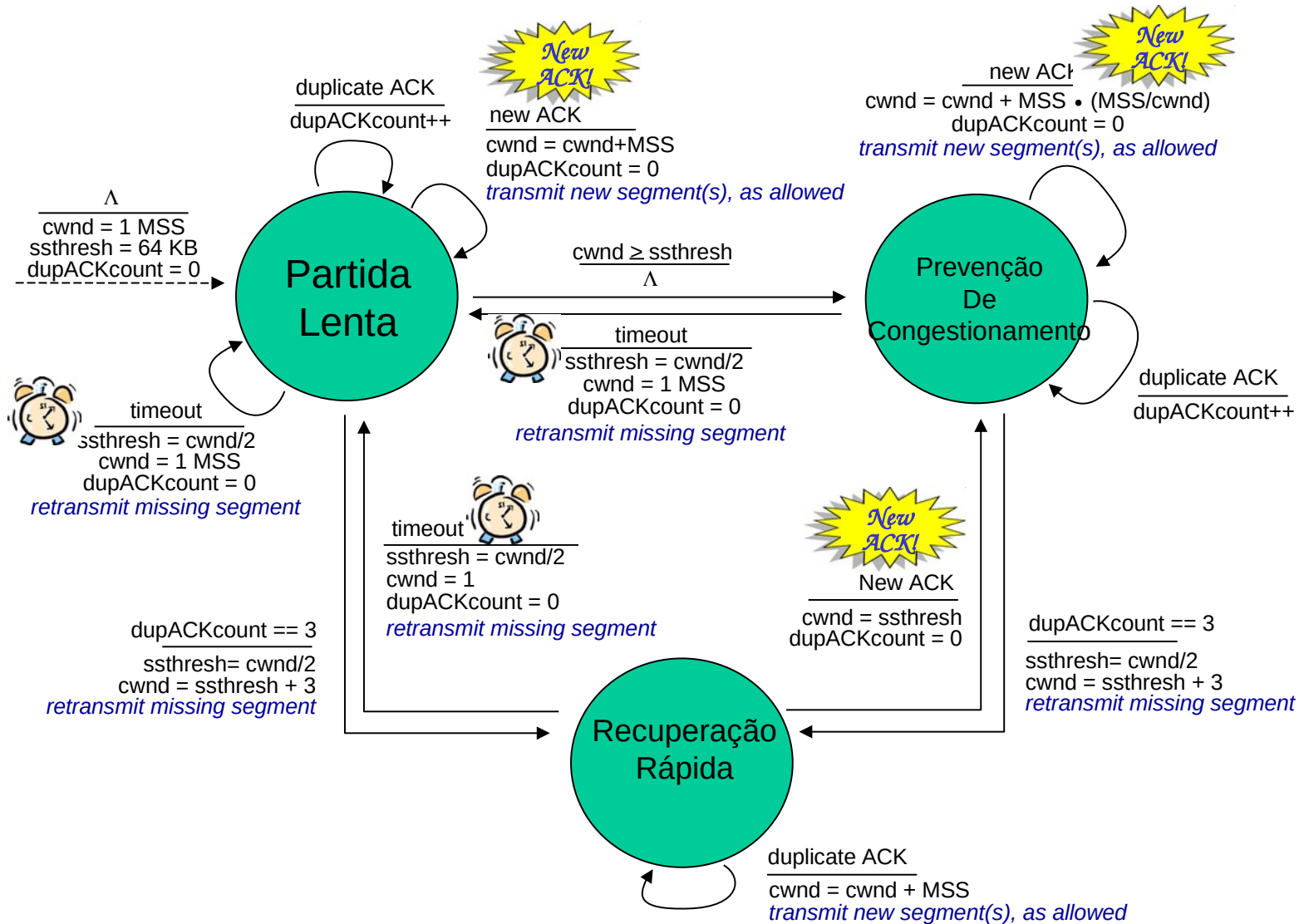
# TCP: estado de partida lenta

- Quando a conexão inicia, há um aumento exponencial na razão de transmissão até que o primeiro evento de perda de segmento ocorra:
  - No início **cwnd** = 1 MSS
  - Depois: dobra **cwnd** a cada RTT
  - Então realiza os incrementos **cwnd** para cada ACK

Sumário: razão inicial é lenta mas sobe exponencialmente rápido (diz-se que o TCP testa a quantidade de banda de rede disponível)



# TCP Controle Congestionamento - sumário





# sumário

Serviços de transporte

Multiplexação e demultiplexação

Transporte sem conexão: UDP

Princípios de transporte com confiança

Transporte orientado a conexão: TCP

- Estrutura do segmento
- Transferência de dados confiável
- Controle de fluxo
- Gerenciamento da conexão

Princípios de controle de congestionamento

Controle de congestionamento no TCP

**Exemplos de código para cliente e servidor TCP em Python**

# Example app: TCP client

## *Python TCPClient*

```
from socket import *
serverName = 'servername'
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName, serverPort))
sentence = raw_input('Input lowercase sentence:')
clientSocket.send(sentence.encode())
modifiedSentence = clientSocket.recv(1024)
print ('From Server:', modifiedSentence.decode())
clientSocket.close()
```

Cria um socket TCP  
(SOCK\_STREAM)



Cliente inicia a conexão com um  
socket servidor (serverPort) que  
está esperando conexões na máquina  
servidora (especificado pelo  
serverName).



Recebe dados do  
servidor



# Example app: TCP server

## Python TCPServer

Cria um socket  
servidor TCP  
(*welcoming socket*)

Servidor inicia "escutando"  
por requisições de clientes  
TCP

Servidor aguarda conexão através  
da chamada **accept()**. Em chegando  
uma requisição, novo socket é  
criado e a função retorna.

Lê bytes do socket (mas não o  
endereço do remetente, como no  
UDP, pois, neste caso, o  
endereço já está associado à  
conexão)

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_STREAM)
serverSocket.bind(('', serverPort))
serverSocket.listen(1)
print 'The server is ready to receive'
while True:
    connectionSocket, addr = serverSocket.accept()
    sentence = connectionSocket.recv(1024).decode()
    capitalizedSentence = sentence.upper()
    connectionSocket.send(capitalizedSentence.encode())
    connectionSocket.close()
```

Atribuição (*bind*) de uma  
porta para o socket servidor  
(*welcoming socket*).

Fecha a conexão com o  
cliente associado à  
conexão (mas não o  
socket *welcoming*)