# PTC 3450 - Aula 10

2.7 Programação de socket: criando aplicações de rede

3.1 Introdução e serviços da camada de transporte

3.2 Multiplexação e Desmultiplexação

3.3 Transporte sem conexão: UDP

(Kurose, p. 139 - 149)

(Peterson, p. 239-242)

25/04/2017

# Capítulo 2: conteúdo

2.1 Princípios de aplicativos de rede

2.2 *Web* e HTTP

2.3 Correio eletrônico

- SMTP, POP3, IMAP

2.4 DNS

2.5 Aplicativos P2P

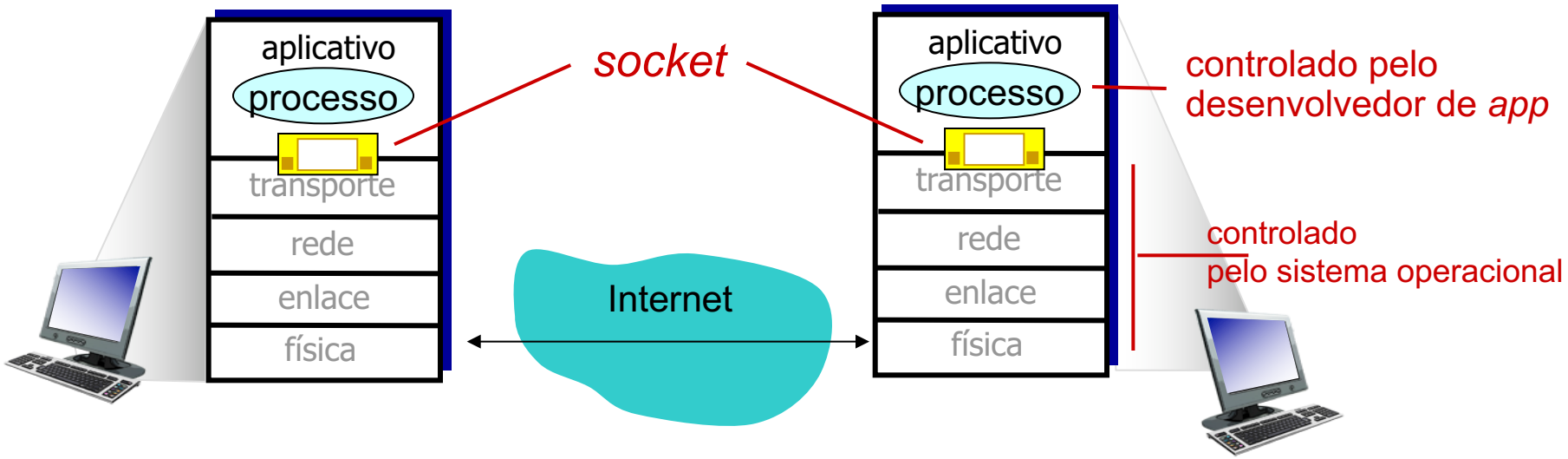
2.6 *Streaming* de vídeo e redes de distribuição de conteúdo

2.7 Programando *socket* com UDP e TCP

# Programando *Sockets*

*objetivo:* aprender como construir aplicativos cliente/servidor que se comunicam usando *sockets*

*socket:* porta entre processo aplicativo e protocolo de transporte fim-a-fim



# Exemplo de *app*: cliente UDP

## *UDPCliente.py em Python*

endereço IPv4

UDP

inclui biblioteca de *socket* do Python

cria socket UDP no cliente

obtém entrada do teclado do usúário

Anexa nome e número da porta do servidor à mensagem; enviar pelo *socket*

lê caracteres respondidos do *socket* para uma *string*

imprime mensagem recebido e fecha *socket*

```
from socket import *

Nomeservidor = "hostname" # endereço IP ou nome do servidor

Portaservidor = 12000

clienteSocket = socket(socket.AF_INET,
                                        socket.SOCK_DGRAM)

mensagem = raw_input('Entre uma frase em minusc.:')

clienteSocket.sendto(mensagem,(Nomeservidor, Portaservidor))

Mensagemmodificada, Enderecoservidor= clienteSocket.recvfrom(2048)

print Mensagemmodificada

clienteSocket.close()
```

# Exemplo de *app*: servidor UDP

## *ServidorUDP em Python*

```
from socket import *

Portaservidor = 12000

servidorSocket = socket(AF_INET, SOCK_DGRAM)

servidorSocket.bind(('', Portaservidor))

print "O servidor está pronto para receber"

while 1:

    mensagem, enderecoCliente = servidorSocket.recvfrom(2048)

    mensagemModificada = mensagem.upper()

    servidorSocket.sendto(mensagemModificada, enderecoCliente)
```

cria *socket* UDP

amarra *socket* à porta local número 12000

laço eterno

Lê de *socket* UDP para mensagem, obtendo endereço do cliente (IP e porta)

envia *string* em maiúsculas de volta para o cliente

# Programando *Sockets com TCP*

cliente precisa contatar servidor

- processo servidor precisa primeiro estar rodando
- servidor precisar ter criado *socket* (porta) que acolhe contato do cliente

cliente contata servidor:

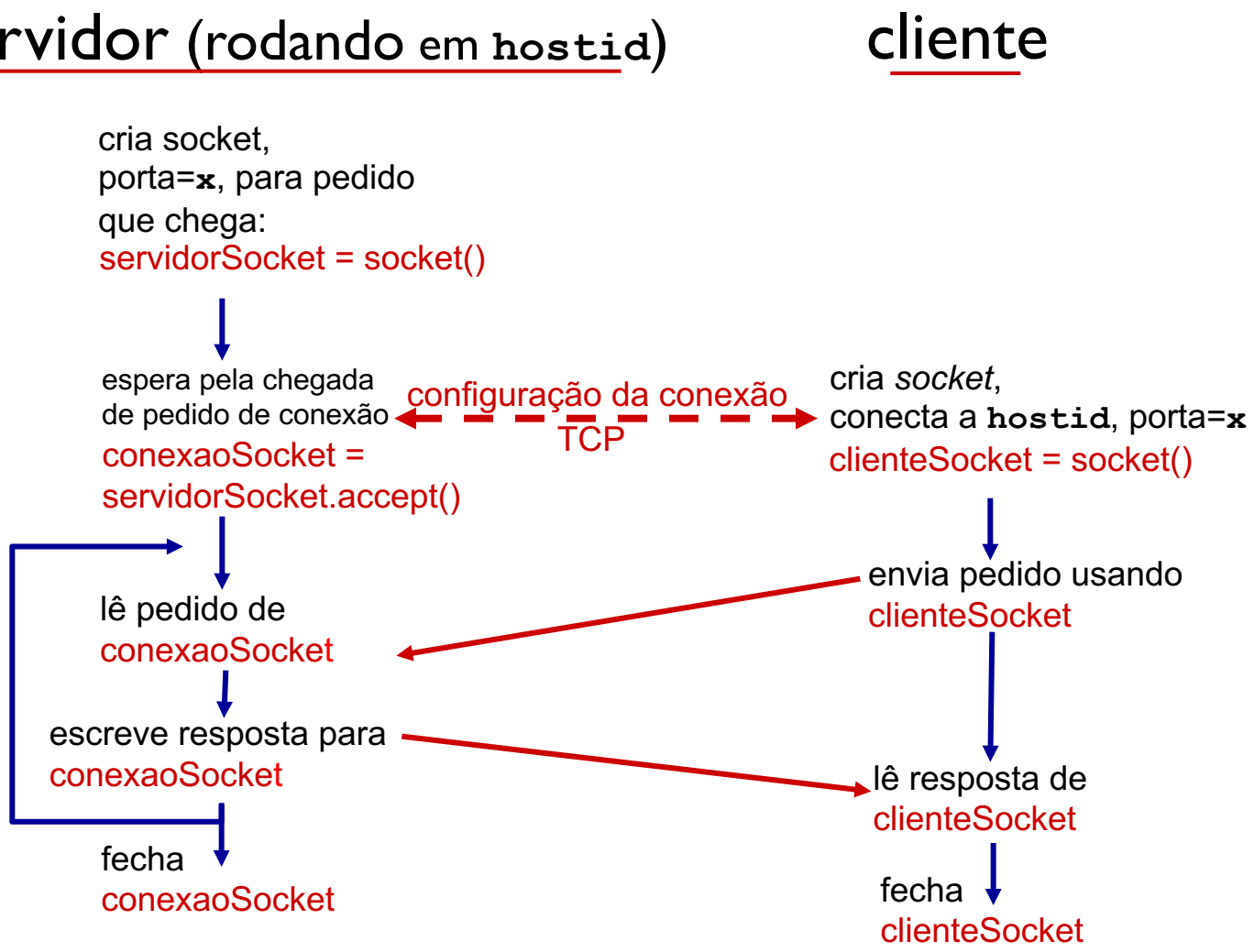
- Criando *socket* TCP, especificando endereço IP e número da porta do processo cliente
- *quando cliente cria socket:* cliente TCP estabelece conexão com servidor TCP

- quando contatado por cliente, *servidor TCP cria novo socket* para processo servidor se comunicar com aquele cliente em particular
  - permite servidor falar com múltiplos clientes
  - números de portas das fontes usados para distinguir clientes (mais no Cap. 3)

Ponto de vista do aplicativo:

TCP provê transferência confiável e em ordem de fluxo de bytes ("cano") entre cliente e servidor

# Interação dos *socket* cliente/servidor: TCP



# Exemplo de *app*: cliente TCP

## *TCPCliente em Python*

```
from socket import *
Nomeservidor = 'servername'
Portaservidor = 12000
clienteSocket = socket(AF_INET, SOCK_STREAM)
clienteSocket.connect((Nomeservidor, Portaservidor))
frase = raw_input('Entre com frase em minusc.:')
clienteSocket.send(frase)
fraseModificada = clienteSocket.recv(2048)
print 'Do servidor:', fraseModificada
clienteSocket.close()
```

cria conexão TCP para servidor, porta remota 12000

Não necessário anexar nome e porta do servidor

# Exemplo de *app*: servidor TCP

## *TCPServidor em Python*

```
from socket import *
servidorPorta = 12000
servidorSocket = socket(AF_INET,SOCK_STREAM)
servidorSocket.bind(('', servidorPorta))
servidorSocket.listen(1)
print 'O sevidor está pronto para receber!'
while 1:
    conexaoSocket, addr = servidorSocket.accept()

    frase = conexaoSocket.recv(1024)
    frasemaiusc = frase.upper()
    conexaoSocket.send(frasemaiusc)
    conexaoSocket.close()
```

cria socket TCP de acolhimento

servidor começa a escutar pedidos TCP que chegam

laço eterno

servidor espera no accept() por pedidos que chegam, novo *socket* criado em resposta

lê bytes do *socket* (mas não endereços como no UDP)

fecha conexão a esse cliente (mas não o *socket* de acolhimento)

# Capítulo 2:resumo

*nosso estudo dos apps* de rede agora completo!

- arquiteturas de aplicativos
  - cliente-servidor
  - P2P
- requisitos de serviços dos aplicativos:
  - confiabilidade, capacidade, latência
- modelo de serviço de transporte Internet
  - orientada a conexão, confiável: TCP
  - não confiável, datagramas: UDP
- protocolos específicos :
  - HTTP
  - SMTP, POP, IMAP
  - DNS
  - P2P: BitTorrent
- programando *sockets* : *sockets* TCP, UDP

# Capítulo 2: resumo

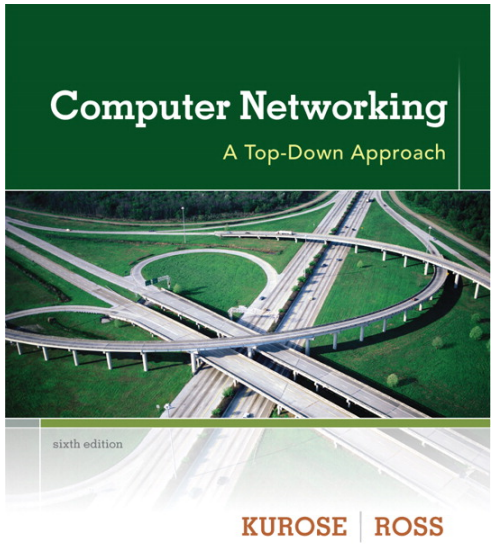
*mais importante: aprendemos sobre protocolos!*

- troca de mensagens pedido/resposta típica:
  - cliente pede info ou serviço
  - servidor responde com dados, código de estado
- formato das mensagens:
  - cabeçalhos: campos dão informações sobre dados
  - dados: informações sendo comunicadas

*temas relevantes:*

- centralizado vs. descentralizado
- com vs. sem memória (estado)
- transferência de msg confiável vs. não confiável
- "complexidade na borda da rede"

# Capítulo 3
# Camada de Transporte




*Computer Networking: A Top Down Approach*
6th edition
Jim Kurose, Keith Ross
Addison-Wesley
March 2012

# Capítulo 3: Camada de Transporte

nossos objetivos:

- entender *princípios* dos serviços da camada de transporte:
  - multiplexação, desmultiplexação
  - transferência de dados confiável
  - controle de fluxo
  - controle de congestionamento
- aprender sobre protocolos da camada de transporte Internet:
  - UDP: transporte sem conexão
  - TCP: transporte confiável orientado a conexão
  - controle de congestionamento no TCP

# Capítulo 3: conteúdo

3.1 serviços da camada de transporte

3.2 multiplexação e desmultiplexação

3.3 transporte sem conexão: UDP

3.4 princípios da transferência de dados confiável

3.5 transporte conectado a transporte: TCP

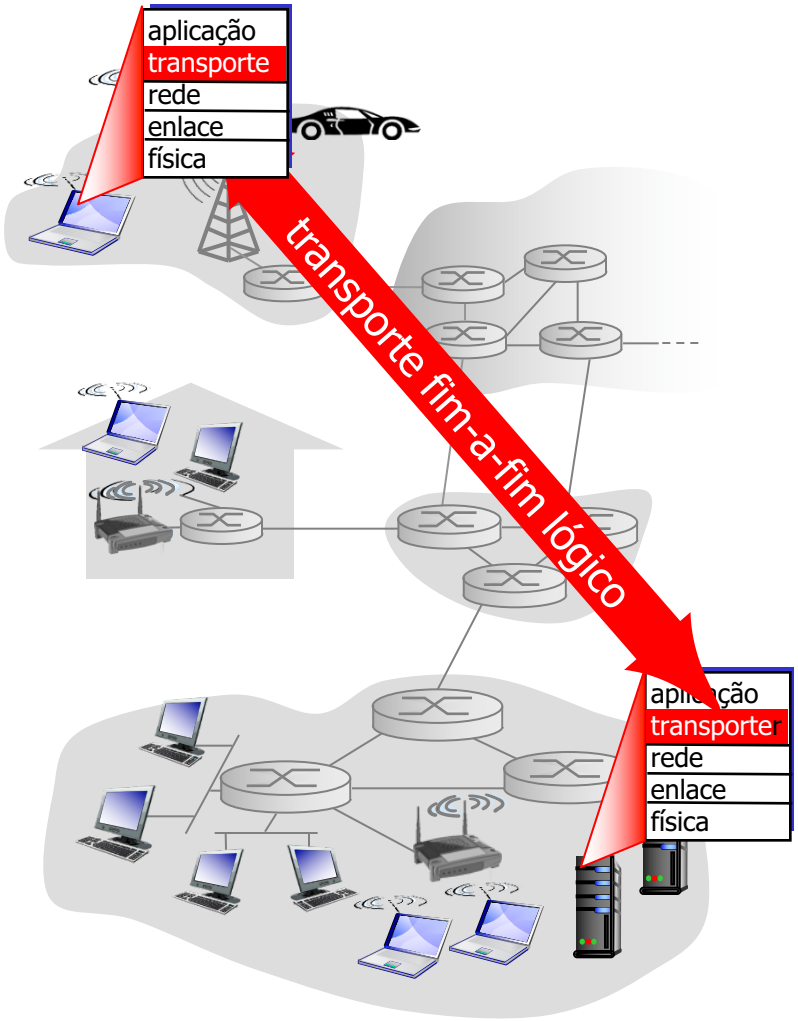
- estrutura dos segmentos
- transferência de dados confiável
- controle de fluxo
- gerenciamento de conexão

3.6 princípios do controle de congestionamento

3.7 controle de congestionamento no TCP

# Serviços e protocolos de transporte

- provê *comunicação lógica* entre *aplicativos* rodando em *hosts* diferentes
- protocolos de transporte rodam em sistemas finais
  - lado enviar: converte (quebrar + inserir cabeçalho) mensagem do *aplicativo* em *segmentos*, passa para a camada de rede
  - lado receber: reconverte segmentos em mensagens, passa para camada de aplicação
- mais de um protocolo de transporte pode estar disponível para aplicativo
  - Internet: TCP e UDP



# Camada de transporte vs. camada de rede

- *camada de rede :* comunicação lógica entre *hosts*
- *camada de transporte:* comunicação lógica entre processos
  - conta com e aprimora serviços da camada de rede

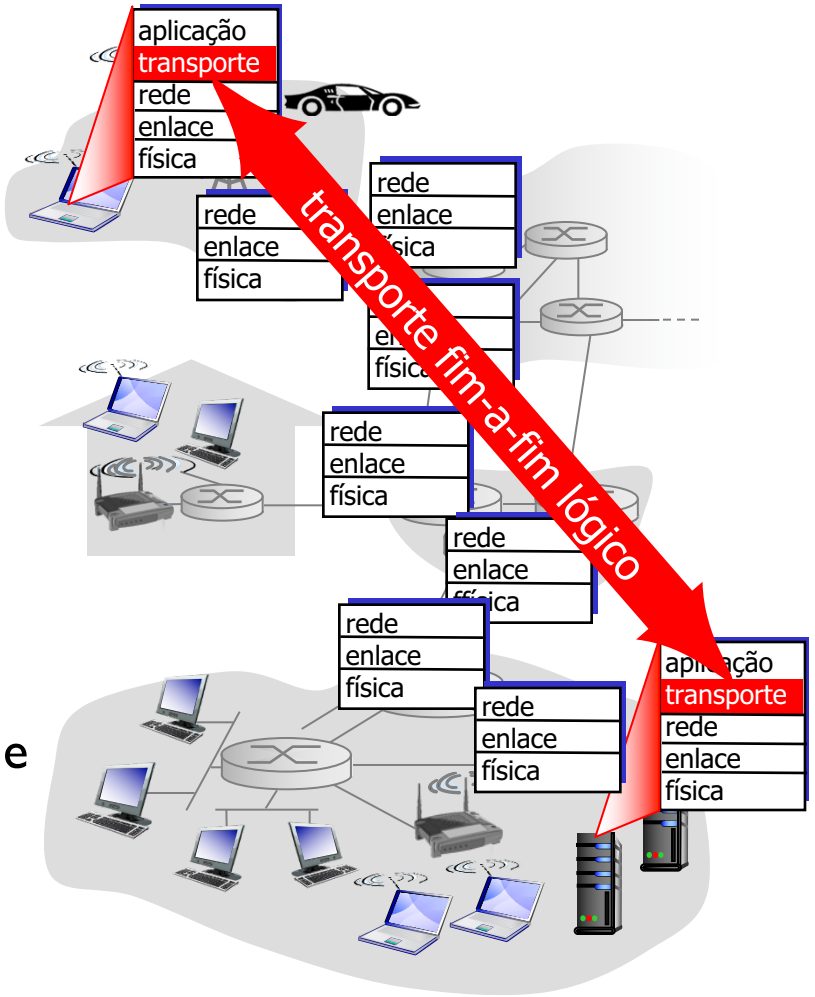
*analogia familiar:*

*12 crianças na casa de Ana enviando cartas para 12 crianças na casa de Bruno:*

- *hosts* = casas
- processos = crianças
- mensagens *app* = cartas em envelopes
- protocolo de transporte = Ana e Bruno que fazem demux interno entre os irmãos
- protocolo da camada de rede = serviço de correio

# Protocolos da camada de transporte na Internet

- entrega confiável e em ordem (TCP)
  - controle de congestionamento
  - controle de fluxo
  - *setup* de conexão
- entrega não confiável e desordenada: UDP
  - extensão "sem frescuras" do "melhor-esforço" do IP
  - Apenas multiplexação/desmultiplexação e detecção de erro
- serviços não disponíveis:
  - garantias de latência
  - garantias de capacidade



# Capítulo 3: conteúdo

3.1 serviços da camada de transporte

3.2 multiplexação e desmultiplexação

3.3 transporte sem conexão: UDP

3.4 princípios da transferência de dados confiável

3.5 transporte conectado a transporte: TCP

- estrutura dos segmentos
- transferência de dados confiável
- controle de fluxo
- gerenciamento de conexão

3.6 princípios do controle de congestionamento

3.7 controle de congestionamento no TCP

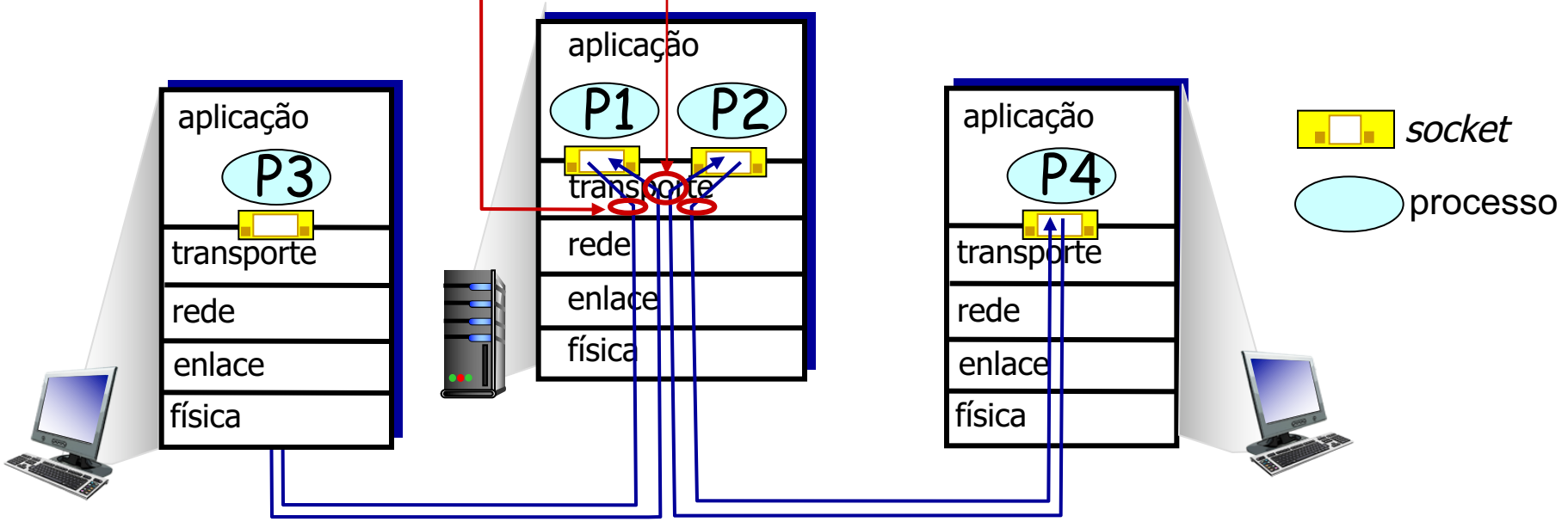
# Multiplexação/desmultiplexação

*multiplexando no remetente:*

manipula dados de múltiplos *sockets*, adiciona cabeçalho de transporte (usado depois para desmultiplexar)

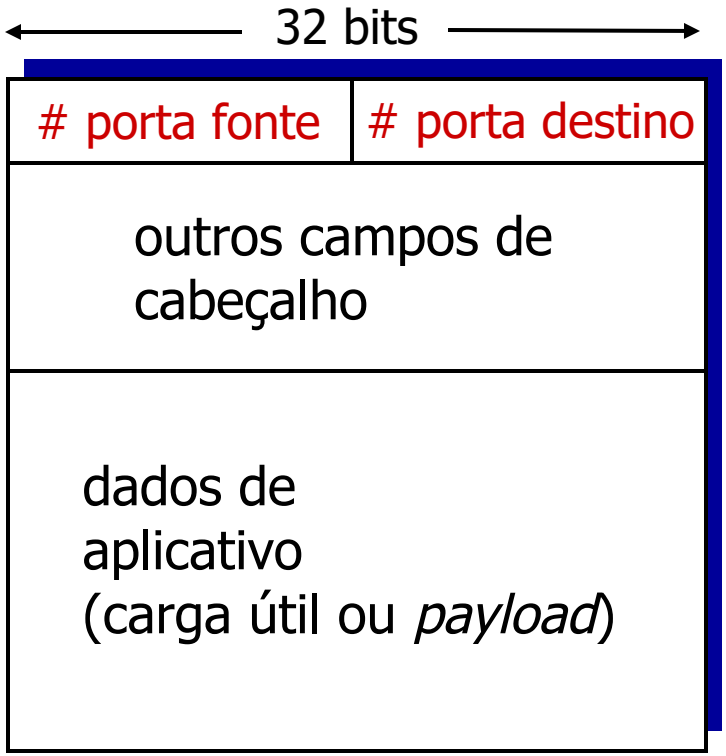
*desmultiplexando no destinatário:*

usa info do cabeçalho para entregar segmentos recebidos para *socket* correto



# Como a *desmultiplexação* funciona

- *host* recebe datagramas IP
  - cada datagrama tem endereço IP da fonte e endereço IP do destino
  - cada datagrama carrega um segmento da camada de transporte
  - cada segmento tem números de porta destino e fonte (16 bits cada – de 0 a 1023 – reservado – RFC 1700)
- *host* usa *endereços IP & números de portas* para direcionar segmento para o *socket* apropriado
- Teste nmap!




formato do segmento TCP/UDP

# Desmultiplexando sem conecção (UDP)

- *lembrando: socket* criado tem # porta do *host* local (automático ou não)

```
clienteSocket = socket(socket.AF_INET,socket.SOCK_DGRAM)

clienteSocket.bind(('', 19157))
```

- *lembrando:* quando se cria datagrama a ser enviado pelo *socket* UDP, especifica-se
  - endereço IP destino
  - # porta destino
  - `clienteSocket.sendto(mensagem,(Nomeservidor, Portaservidor))`

---

- quando *host* recebe segmento UDP :
  - verifica # da porta destino no segmento
  - direciona segmento UDP para *socket* com esse # de porta

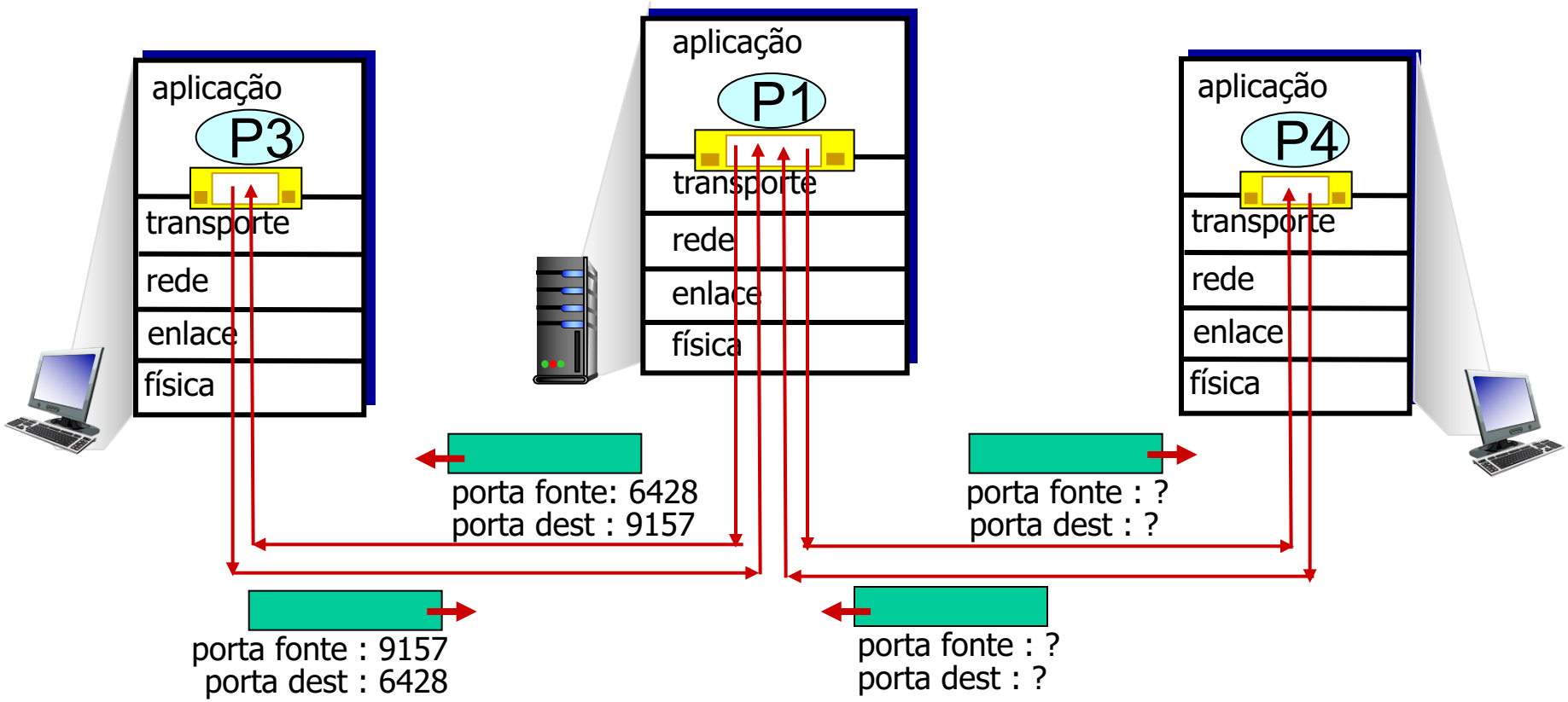
datagramas IP com *mesmo # porta dest.*, mas diferentes endereços IP ou números de portas fontes serão direcionados ao *mesmo socket* no dest

# Demux sem conexão : exemplo

```
mySocket2 =
  socket(socket.AF_INET,socket.
  SOCK_DGRAM)
clienteSocket.bind(('', 9157))
```

```
serversocket =
    socket(socket.AF_INET,socket.
    SOCK_DGRAM)
serversocket.bind(('',6428))
```

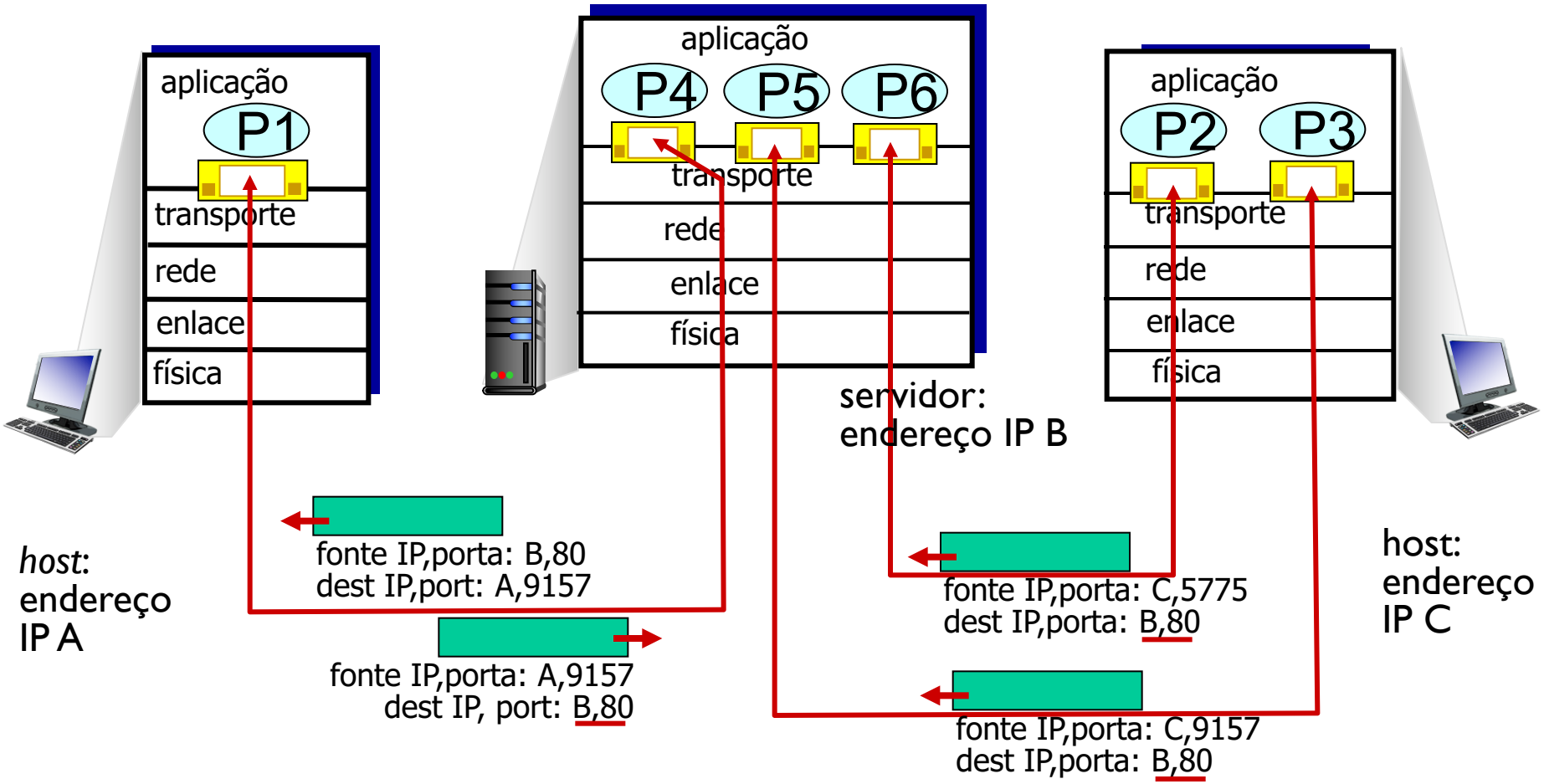
```
mySocket2 =
  socket(socket.AF_INET,socket.
  SOCK_DGRAM)
clienteSocket.bind(('', 5775)
```



# Demux orientada a conexão

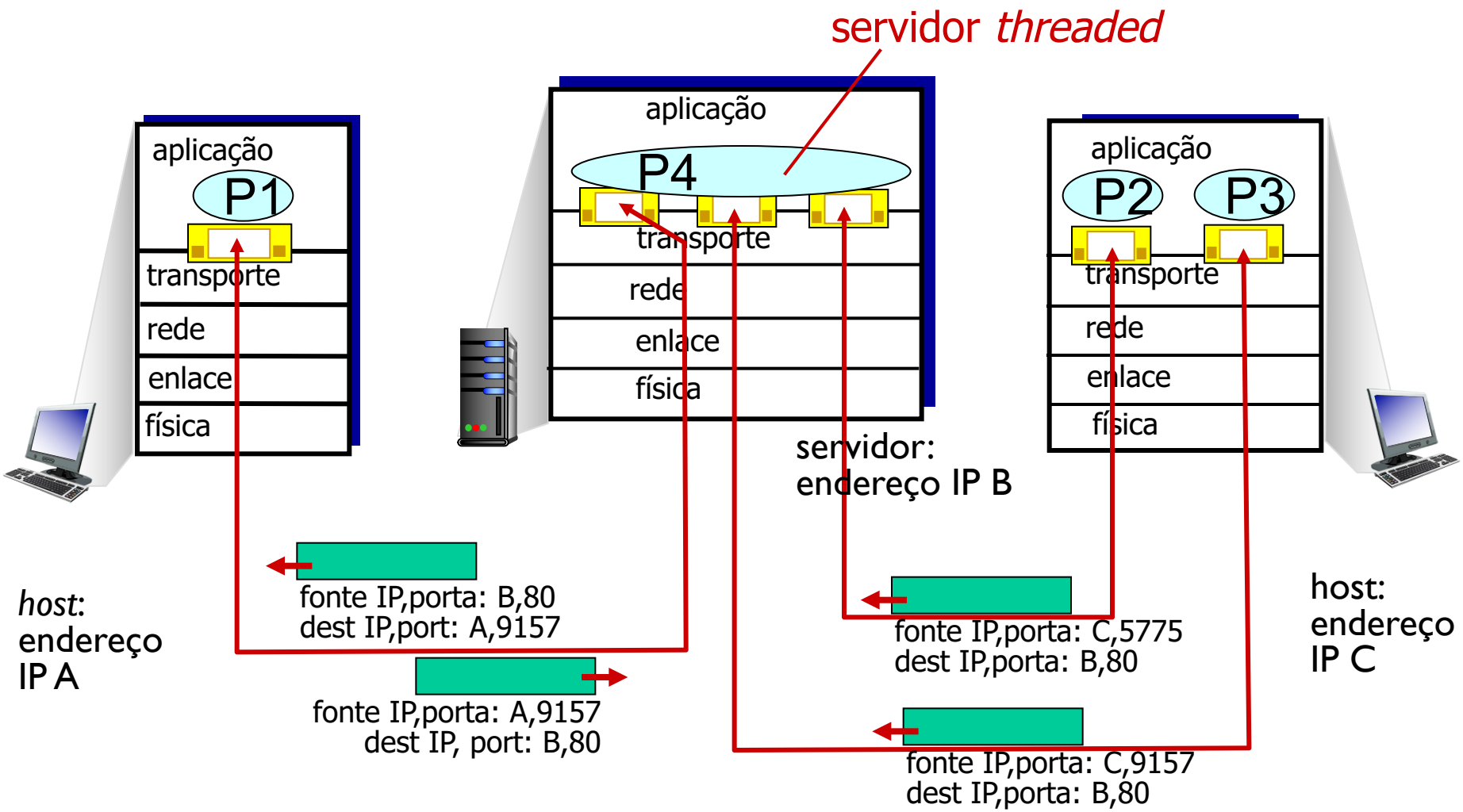
- *Socket* TCP identificado por 4-upla:
  - endereço IP fonte
  - número porta fonte
  - endereço IP dest
  - número porta dest
- demux: destinatário usa todos 4 valores para direcionar segmento para *socket* apropriado
- *host* servidor pode manter muitos *sockets* TCP simultâneos:
  - cada *socket* identificado por sua própria 4-upla
- Exemplo: servidores *web* têm *sockets* diferentes para cada cliente se conectando
  - HTTP não-persistente terá *socket* diferente para cada pedido

# Demux orientado a conexão: exemplo




3 segmentos, todos destinados ao endereço IP: B,
porta dest : 80 são desmultiplexados para sockets *diferentes*

# Demux orientado a conexão: exemplo



# Capítulo 3: conteúdo

3.1 serviços da camada de transporte

3.2 multiplexação e desmultiplexação

3.3 transporte sem conexão: UDP

3.4 princípios da transferência de dados confiável

3.5 transporte conectado a transporte: TCP

- estrutura dos segmentos
- transferência de dados confiável
- controle de fluxo
- gerenciamento de conexão

3.6 princípios do controle de congestionamento

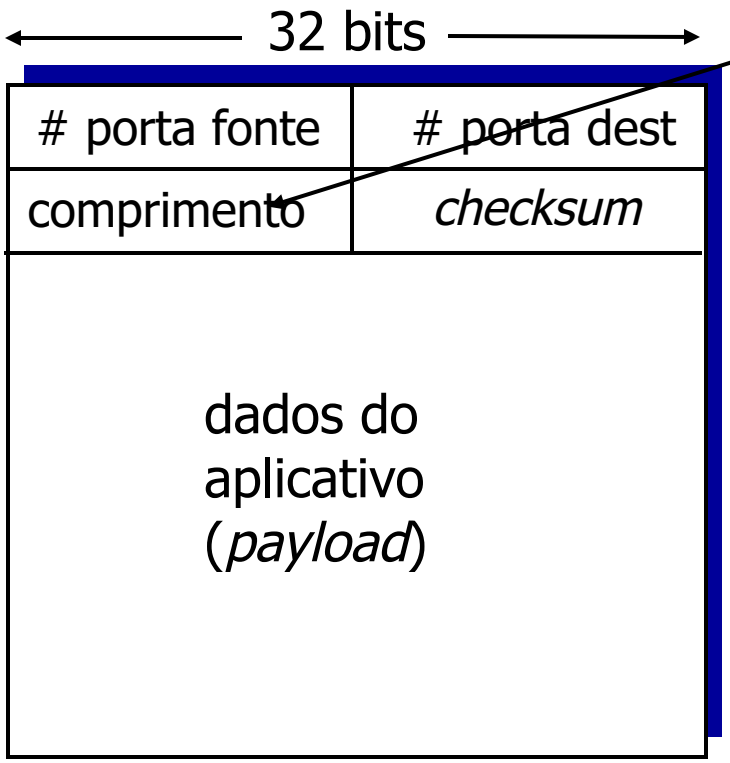
3.7 controle de congestionamento no TCP

# UDP: *User Datagram Protocol* [RFC 768]

- protocolo de transporte Internet "sem frescura", "minimalista", faz o mínimo possível
- serviço "melhor esforço", segmentos UDP podem ser:
  - perdidos
  - entregues fora de ordem para *app*
- *sem conexão:*
  - não há *handshaking* entre remetente e destinatário UDP
  - cada segmento UDP tratado independentemente de outros

- usam UDP:
  - alguns apps de *streaming* multimídia (tolerantes a perdas, sensíveis a taxa)
  - DNS
  - SNMP (*Simple Network Management Protocol*)
- transferência confiável sobre UDP:
  - adicionar confiabilidade na camada de aplicação
  - recuperação de erro específica da aplicação!

# Cabeçalho do segmento UDP (RFC 768)




formato do segmento UDP

comprimento, em bytes do segmento UDP, incluindo cabeçalho

## por que existe um UDP?

- sem estabecimento de conexão (que pode adicionar atraso)
- simples: sem estado da conexão no remetente e no destinatário
- pequeno cabeçalho(UDP = 8 bytes; TCP = 20 bytes)
- melhor controle no nível da aplicação sobre quais dados são enviados e quando

# *Checksum* Internet : exemplo

exemplo: soma de inteiros de 16-bit

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| wraparound | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 |
| soma | | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 |
| *checksum* | | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 |

*Notas:*

1) quando somamos números, o "vai um" do bit mais significativo precisa ser adicionado ao resultado
2) Se não houver erros, soma de todas as palavras de 16 bits no destino deve ser 1111111111111111.

*Veja mais exemplos iterativos aqui!*

# *Checksum* UDP

*Objetivo:* detectar "erros" (e.g., bits alterados) em segmento transmitido

## remetente:

- trata conteúdo do segmento, incluindo campos de cabeçalho, como sequência de inteiros de 16-bit
- *checksum*: complemento de 1 da soma do conteúdo do segmento
- remetente coloca valor *checksum* no campo UDP *checksum*

## destinatário:

- calcula *checksum* do segmento recebido
- verifica se *checksum* calculado é igual ao valor no campo *checksum*:
  - NÃO - erro detectado
  - SIM – nenhum erro detectado. Mas podem haver erros mesmo assim? Mais adiante…