PTC 3450 - Aula 11

2. A camada de aplicação e tráfego multimídia
2.8 Voz sobre IP
2.9 Programação de socket: criando aplicativos de rede

(Kurose, 7ª edição, Seções 2.7 e 9.3)

30/04/2024

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 Aplicativos de rede multimídia

2.7 *Streaming* de video e redes de distribuição de conteúdo

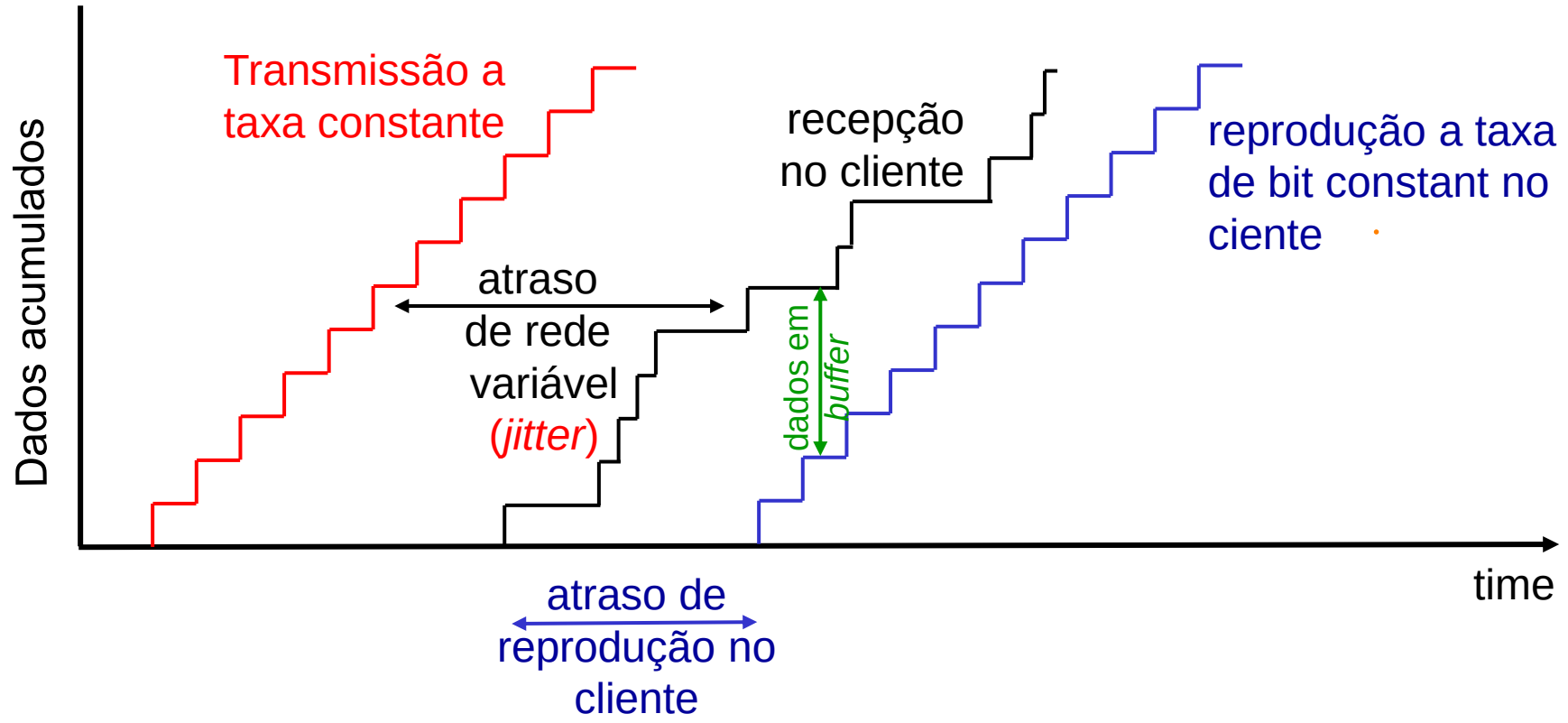
2.8 Voz sobre IP

2.9 Programando *socket* com UDP e TCP

VoIP: características

- Voz: alterna rajadas de fala e períodos de silêncio.
- Situação típica:
  - Pacotes gerados apenas durante rajadas de fala
  - 64 kbits/s durante rajadas de fala
  - Blocos de 20 ms a taxa de 64 kbits/s: 160 bytes de dados
- Cabeçalho da camada de aplicação adicionado a cada bloco – bloco + cabeçalho = pacote
- Aplicação entrega pacote para *socket* a cada 20 ms durante rajada de fala
- Pacote de dados encapsulado em segmento UDP ou TCP
- Numa rede ideal, pacotes chegam com mesmo atraso e não se perdem: basta receptor reproduzir cada pacote que chega

Problemas para o VoIP: 1) *Jitter* do atraso



- *Jitter*: *atraso variáveis entre momentos de transmissão e recepção*
- Atraso fim-a-fim de dois pacotes consecutivos: diferença pode ser maior ou menor do que 20 ms (diferença no tempo de transmissão)

Problemas para o VoIP: **2) perda de pacotes**

- *Perdas de pacotes na rede:* datagrama IP perdido devido a congestionamento da rede (estouro de buffer de roteador) – retransmissão não resolve…
- *Perdas por atraso:* datagramas IP chegam muito tarde para ser reproduzidas no receptor
  - Atrasos: processamento e filas na rede; atrasos nos sistemas finais (transmissor, receptor)
    - < 150 ms não perceptível
    - Entre 150ms e 400 ms – aceitável
    - Acima de 400 ms prejudica muito interatividade da conversa
- *Boa notícia: Tolerância a perdas:* dependendo da codificação da voz e cancelamento da perda, taxas de perda de até 20% podem ser toleradas

- Vamos ver como ***aplicativos*** resolvem os problemas na sequência... ☺

Resolvendo o problema 1

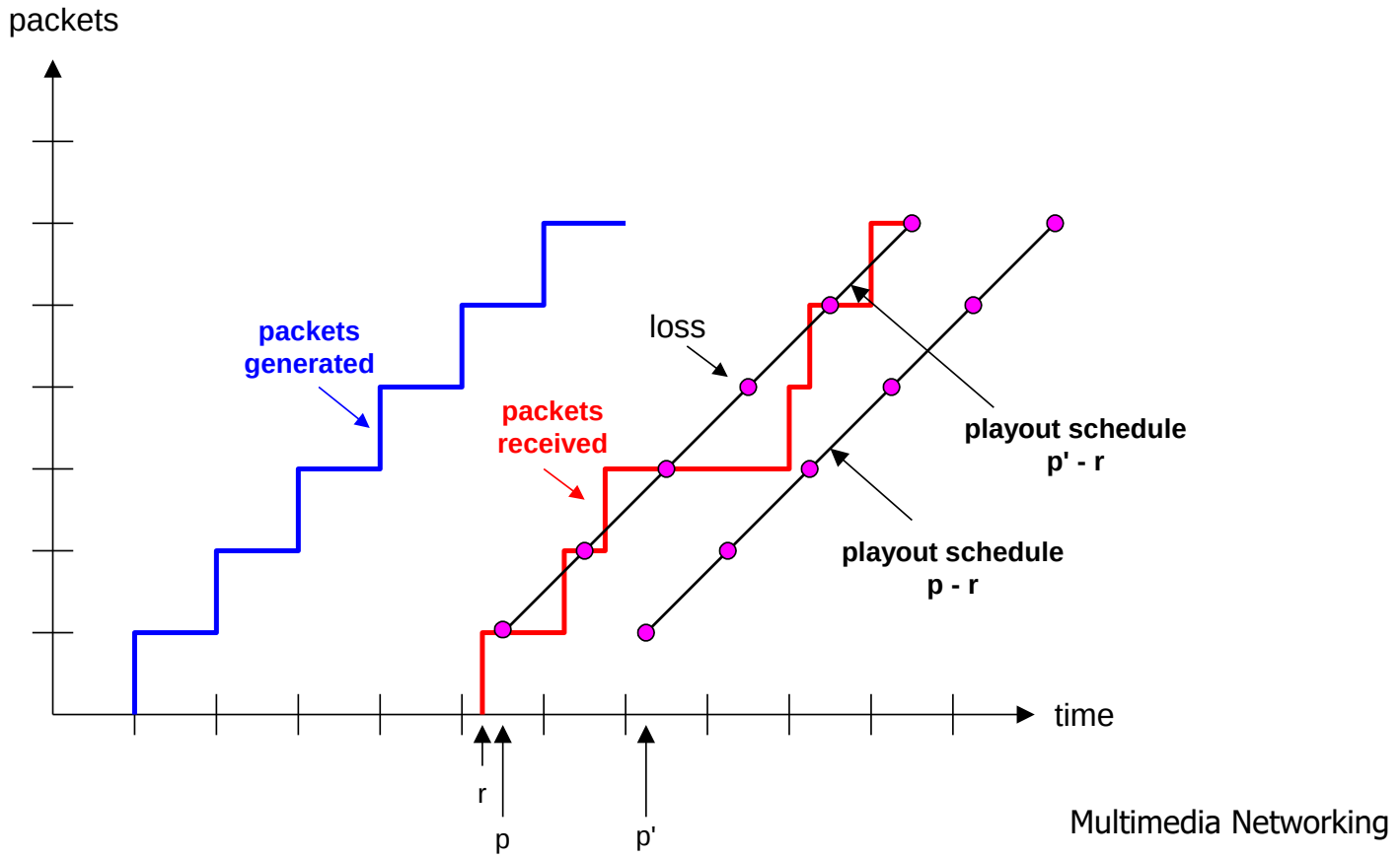
- Como encontrar o atraso de reprodução ótimo para evitar os problemas de *jitter*?

VoIP: Atraso de reprodução fixo

- Receptor tenta reproduzir cada bloco exatamente *q* *ms* após o bloco ter sido gerado.
  - Cabeçalho do bloco tem marca de tempo *t*: receptor reproduz bloco no instante *t*+*q*
  - Se bloco chega depois de *t*+*q*: dados chegaram tarde demais para reprodução – dados "perdidos"
- Compromisso na escolha de *q*:
  - *q grande:* menos perdas de pacotes
  - *q pequeno:* melhor experiência interativa
- Ideal abaixo de 150 ms

VoIP: Atraso de reprodução fixo

- Transmissor gera pacotes a cada 20 ms durante rajada de fala
- Primeiro pacote recebido no instante *r*
- Exemplo 1) Agendamento de reprodução: começa em *p*
- Exemplo 2) Agendamento de reprodução: começa em *p'*



Atraso de reprodução adaptativo [Ramjee 1994]

- *Objetivo:* baixo atraso de reprodução, baixa taxa de perdas por atraso
- *Abordagem:* ajuste adaptativo do atraso de reprodução:
  - Estima atraso de rede, ajustando atraso de reprodução no começo de cada rajada de voz
  - Períodos de silêncio podem ser comprimidos ou alongados devido ao ajuste – não é perceptível
  - Blocos ainda reproduzidos a cada 20 ms durante a rajada de fala
- Atraso de pacote estimado adaptativamente. (EWMA - *Exponentially Weighted Moving Average*, mesma ideia da estimação do RTT no TCP):

$$d_i = (1-\alpha)d_{i-1} + \alpha\,(r_i - t_i)$$

- d_i: *estimativa do atraso após o i-ésimo pacote*
- α: *constante pequena, por exemplo 0,01*
- r_i: *instante de recepção -*
- t_i: *instante de envio* (marca de tempo no pacote)
- $r_i - t_i$: *atraso medido do i-ésimo pacote*

Atraso de reprodução adaptativo

- Estima-se também o desvio-padrão do atraso, v_i :

$$v_i = (1-\beta)v_{i-1} + \beta\,|r_i - t_i - d_i|$$

- Estimativas de d_i e v_i calculadas para todo pacote recebido, mas usado apenas no início de uma rajada de voz
- Para o primeiro pacote na rajada de voz, o instante para reprodução é:

$$\textit{tempo-reprodução}_i = t_i + d_i + Kv_i$$

- Pacotes remanescentes na rajada de fala são então reproduzidos periodicamente (a cada 20 ms por exemplo)

Atraso de reprodução adaptativo

Q: Como o receptor determina se um pacote é o primeiro de uma rajada de voz?

- Se não há perdas, receptor olha em marcas de tempo sucessivas
  - Se diferença entre marcas sucessivas > 20 ms --> rajada de fala começando.
- Com perdas possíveis, receptor precisa olhar tanto marcas de tempo como números sequenciais no cabeçalho do pacote *(não confundir com TCP)*
  - Diferença entre marcas sucessivas > 20 ms e números sequenciais sem lacunas --> --> rajada de fala começando.
- Exemplo de pesquisa: aqui

Resolvendo o problema 2

- Como lidar com perdas de pacotes?

VoiP: recuperando-se da perda de pacotes [IEEE Network, RFC 5109]

Desafio: recuperar-se de perda de pacote dado o pequeno atraso tolerável entre a transmissão original e a reprodução

- Cada ACK/NAK leva ~ 1 RTT – não adianta para VoIP ☹
- Alternativa: *Forward Error Correction (FEC)*
  - Ideia: enviar bits suficientes pra permitir recuperação sem retransmissão

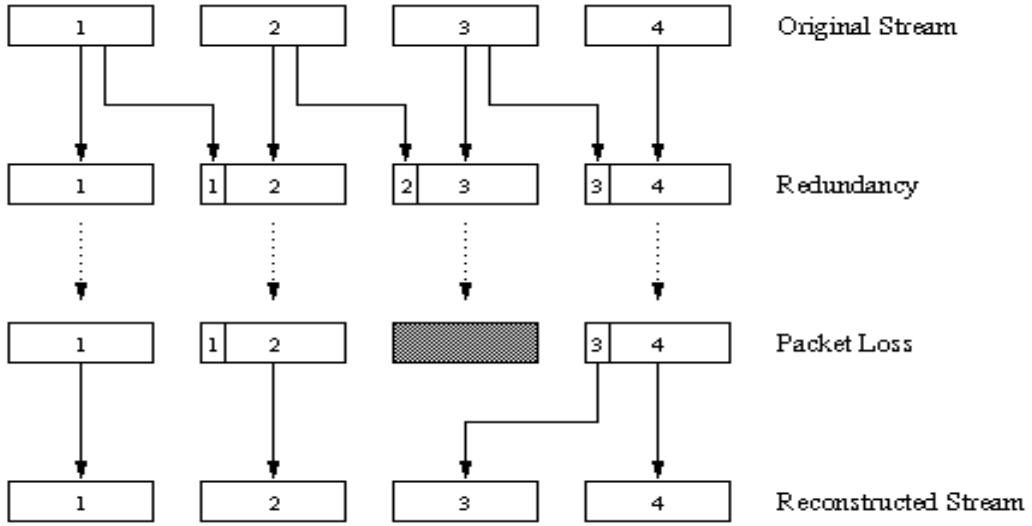
FEC simples

- Para cada grupo de *n* blocos, cria um bloco redundante fazendo XOR com os *n* blocos originais
- Envia *n+1* blocos, aumentando a largura de banda por um fator *1/n*
- Pode-se reconstruir os *n* blocos originais se no máximo 1 bloco se perder entre os *n+1* blocos, com atraso para reprodução

VoiP: recuperando-se da perda de pacotes

Um FEC mais sofisticado:

- "Embutir um fluxo de baixa qualidade na sequência de dados"
- Enviar fluxo de áudio de baixa resolução como informação redundante
- Por exemplo, fluxo nominal é PCM em 64 kbits/s e fluxo redundante é GSM em 13 kbits/s



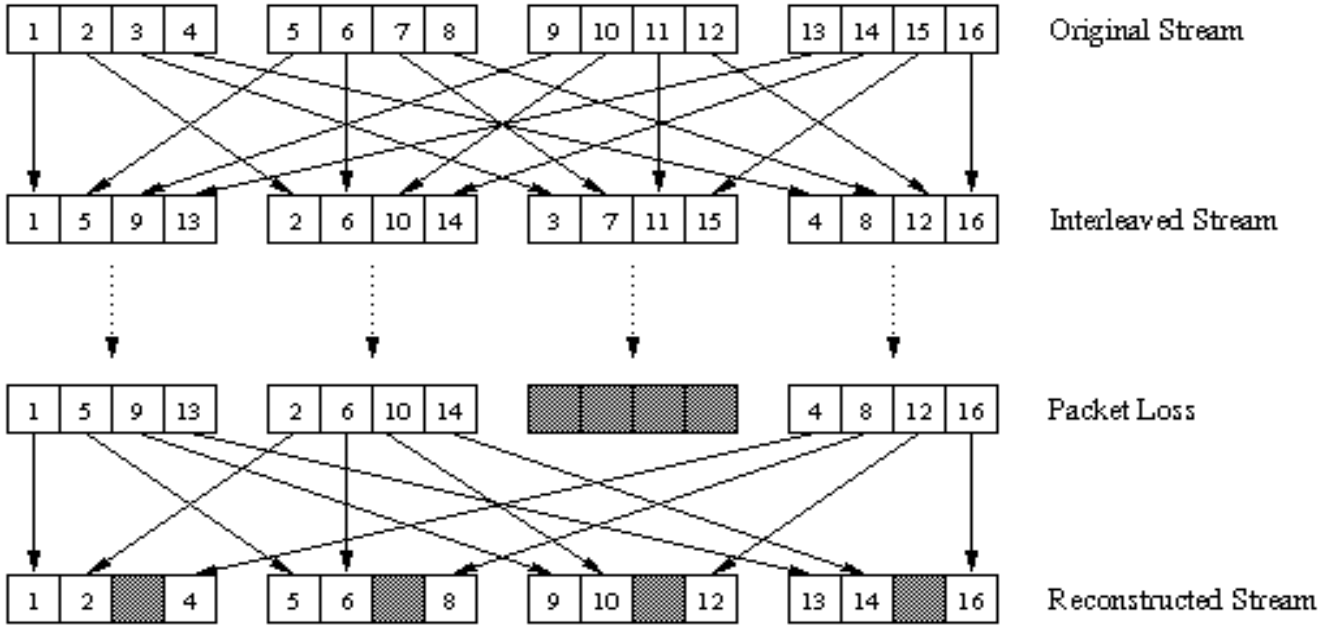
- Perdas não consecutivas: receptor pode cancelar perda
- Generalização: também pode-se anexar bloco (n-1) e (n-2) em baixa taxa no bloco n

VoiP: recuperando-se da perda de pacotes

Entrelaçamento para cancelar perda:

- Blocos de áudio divididos em unidade menores, por exemplo, 4 unidades de 5 ms por bloco de áudio de 20 ms
- Pacote contém pequenas unidades de diferentes blocos

- Se pacote se perder, ainda tem-se *a maior parte* de cada bloco original
- Não há aumento de taxa por redundância, mas há aumento no atraso para reprodução – aplicação limitada em VoIP mas muito usada em áudio gravado

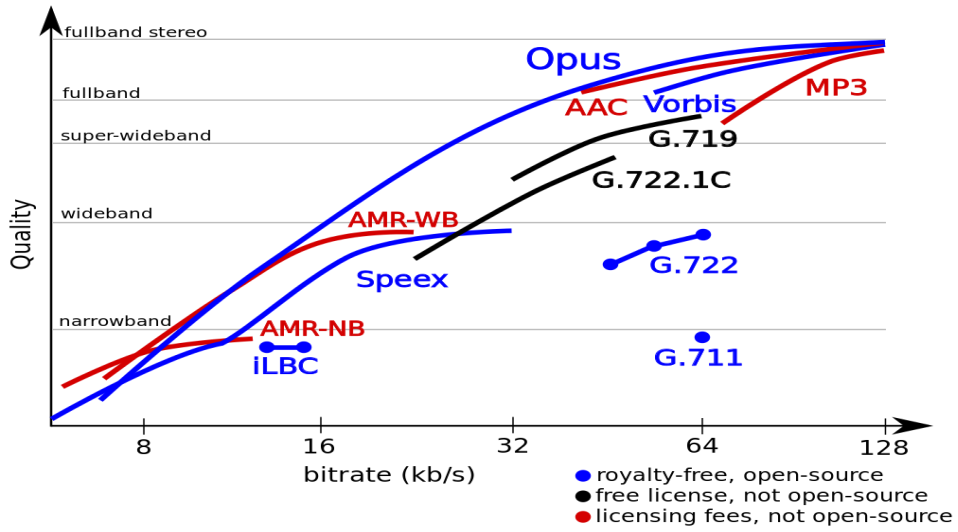


VoiP: recuperando-se da perda de pacotes

- Outras ideias:
  - Repetir bloco anterior no lugar do bloco perdido
  - Fazer interpolação no trecho de áudio
  - ...
- Tema de pesquisa importante!
- Ver por exemplo artigo da IEEE Network

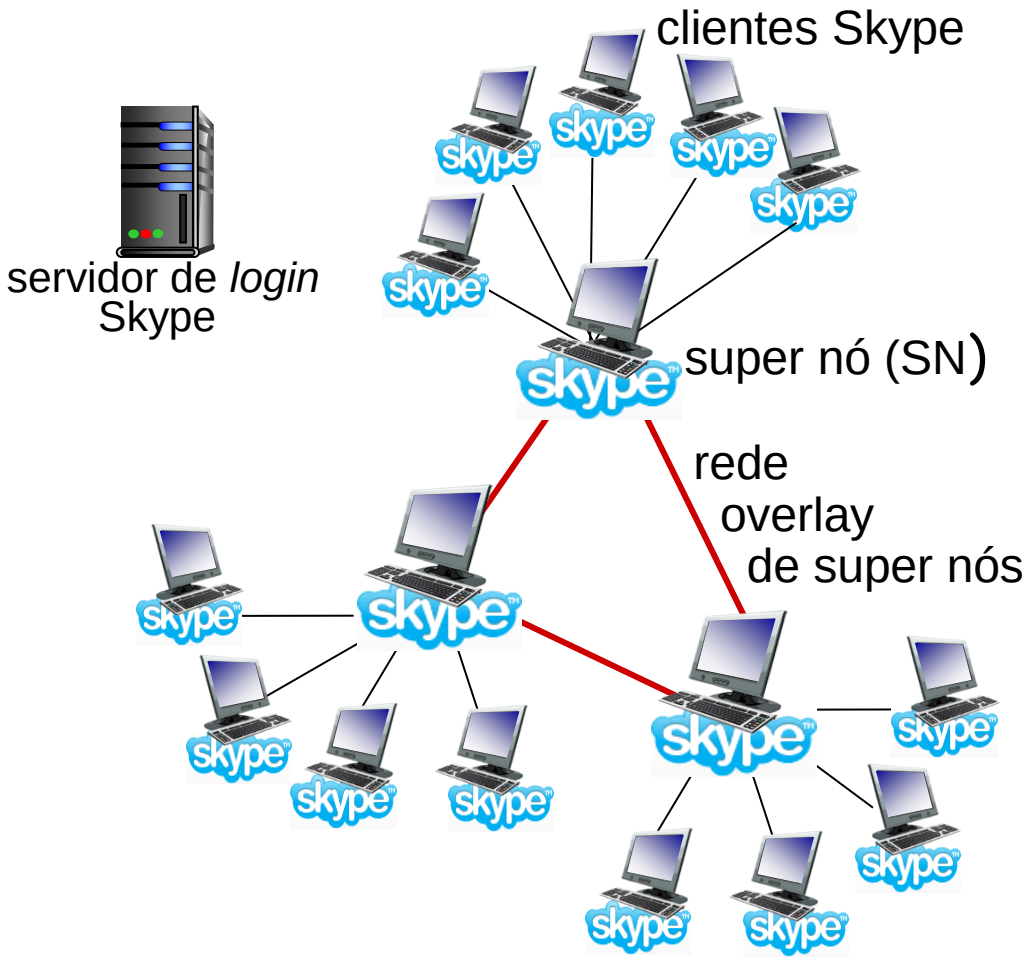
Voice-over-IP: Skype

- Aplicativo VoIP popular– Chegou a ter 50 milhões de usuários concomitantemente
- Fundado em 2003 e comprado pela Microsoft em 2011
- 40% do tráfego de telefonia internacional em 2014
- Protocolo da camada de aplicação proprietário (funcionamento inferido via engenharia reversa, veja por exemplo [Baset 2006])
  - Mensagens criptogradas, mas segurança tem problemas: áudio descriptografado; IP rastreado
- CODEC de áudio Opus; CODEC de vídeo VP7 – taxas variáveis de acordo com qualidade da conexão de 30kbps a 1 Mbps (vídeo)
- Áudio amostrado tipicamente a 16 000 amostras/s – qualidade melhor do que telefonia
- Tenta usar UDP, se não consegue passar vai para TCP



Voice-over-IP: Skype

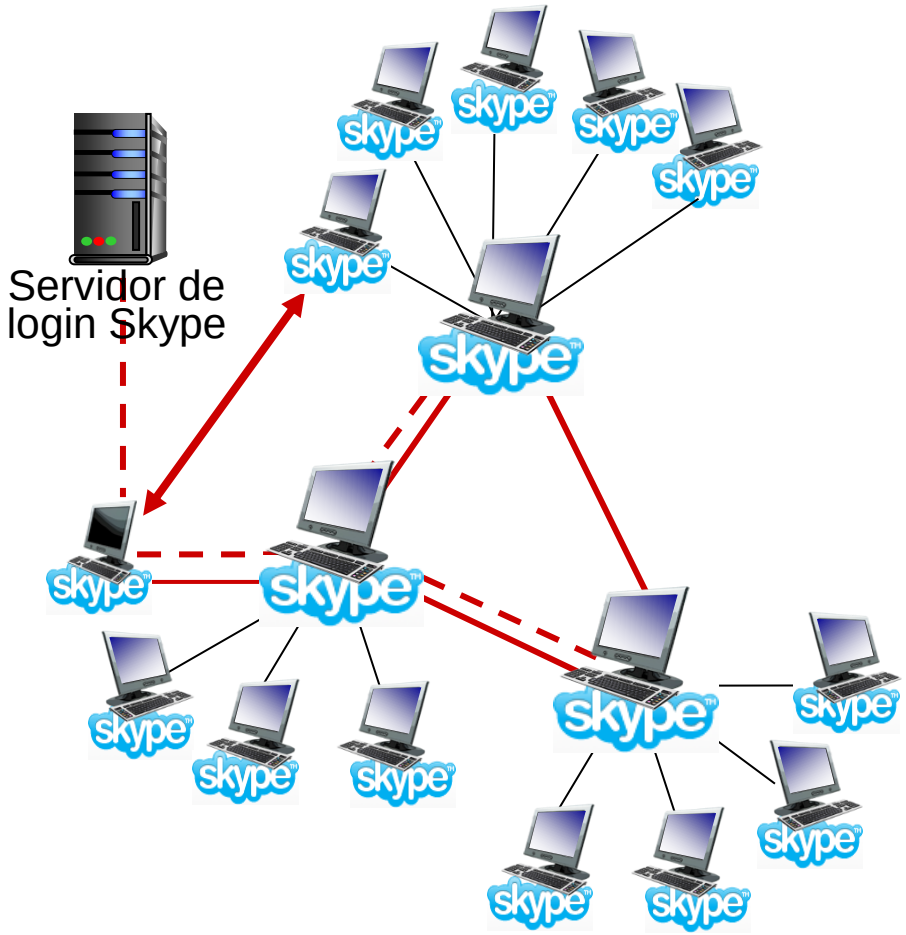
- Componentes P2P:
  - Clientes: *peers* Skype conectam-se diretamente entre si para uma chamada VoIP
  - Super nós (SN): *peers* Skype com funções especiais
  - *Rede overlay*: entre SNs para localizar clientes Skype
  - Servidor de *login*



P2P VoIP: Skype

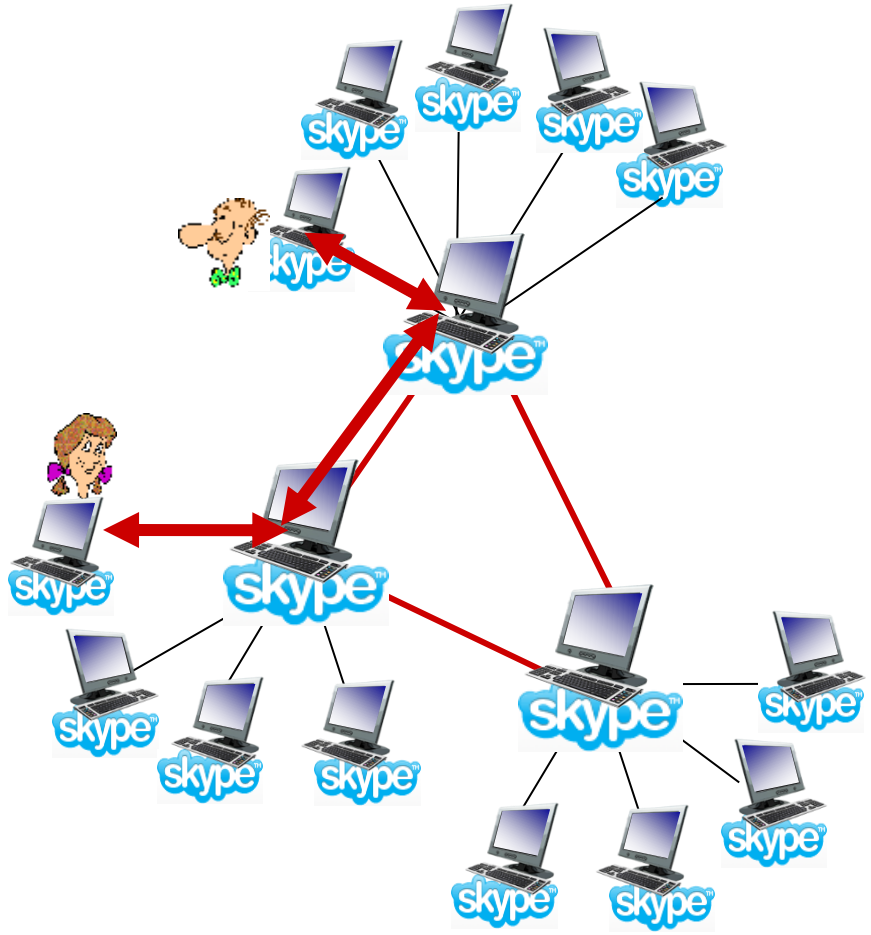
Operação cliente Skype:

1. Junta-se à rede Skype contatando um SN (endereço IP em *cache*) usandoTCP
2. Loga-se (nome de usuário, senha) a um servidor de *login Skype* centralizado
3. Obtém endereço IP do destino do SN, *overlay* SN
   - ou da lista de contatos do cliente
4. Inicia chamada diretamente com destinatário



Skype: *peers* como retransmissores

- *Problema:* tanto Alice quanto Bob estão atrás de NATs
  - NAT impede que *peer* externo inicie conexão com *peer interno*
  - *Peer interno ao NAT consegue iniciar conexão a* peer externo



- Solução com retransmissor (*relay*) : Alice e Bob mantém conexão aberta com seus SNs
  - Alice avisa seu SN que quer se conectar a Bob
  - O SN da Alice se connecta ao SN do Bob
  - O SN de Bob se conecta a Bob por meio da conexão aberta que Bob inicialmente começou com seu SN

Voice-over-IP: Skype

- Outras aplicações e ferramentas interessantes do Skype:
  - Chamada entre *N* usuários
  - Videoconferência com múltiplos usuários
- Precisa de técnicas para não sobrecarregar a rede e os *peers*!
- *Mais sobre as ferramentas utilizadas na página 727 do Kurose e referências indicadas lá, por exemplo [Zhang 2012]*

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 Aplicativos de rede multimídia

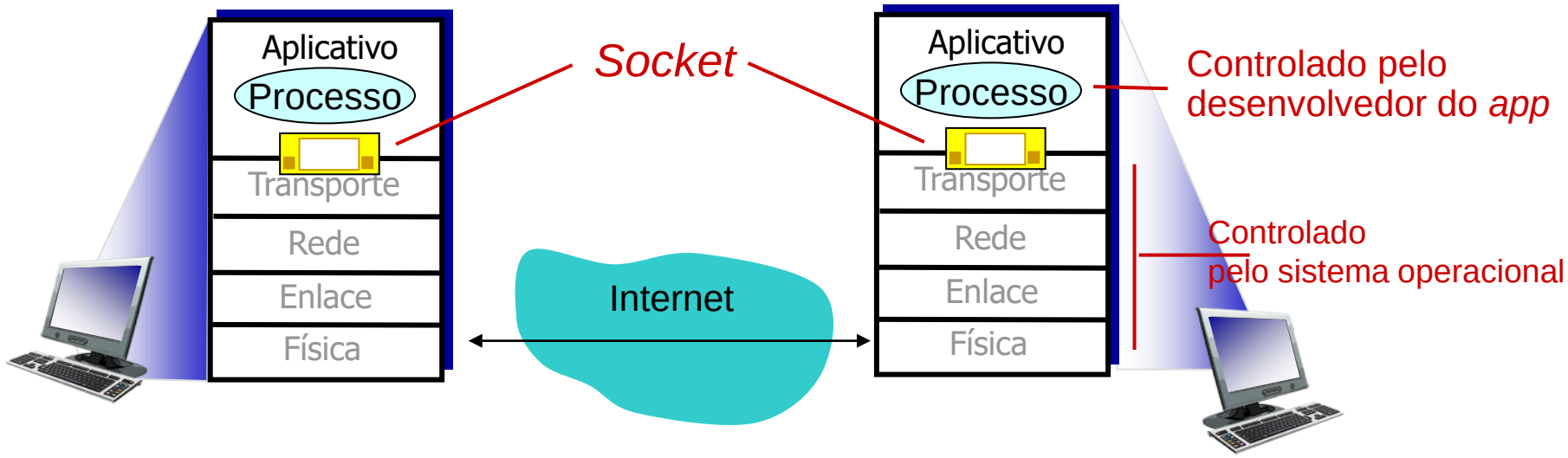
2.7 *Streaming* de video e redes de distribuição de conteúdo

2.8 Voz sobre IP

2.9 Programando *socket* com UDP e TCP

Programando *Sockets*

Socket: porta entre aplicativo e protocolo da camada de transporte



Programando *Sockets*

2 tipos de socket para 2 serviços de transporte:

- *UDP:* datagrama não confiável
- *TCP:* confiável, orientada a fluxo de bytes

Exemplo simples de aplicativo:

1. Cliente lê uma linha de caracteres (dados) do seu teclado e envia os dados ao servidor.
2. Servidor recebe os dados e converte os caracteres em maiúsculas.
3. Servidor envia os dados modificados ao cliente.
4. Cliente recebe os dados modificados e mostra a linha na tela.

Programando *Sockets com UDP*

UDP: sem "conexão" entre cliente & servidor

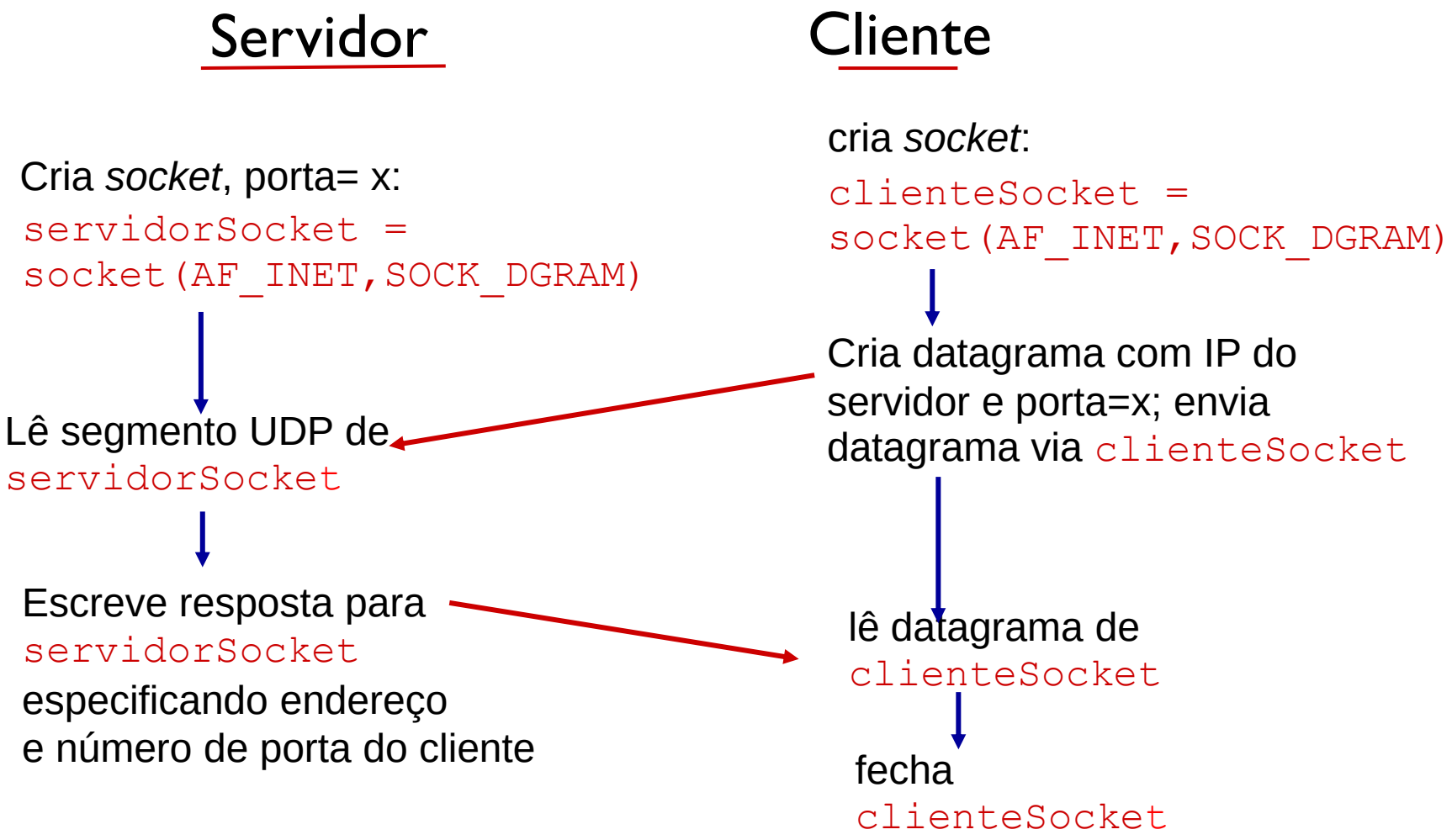
- Sem "*handshaking*" antes de enviar dados
- Remetente explicitamente anexa endereço IP e # porta do destino em cada pacote
- Destinatário extrai endereço IP e # porta do emissor de cada pacote recebido

UDP: dados transmitidos podem ser perdidos ou recebidos fora de ordem

Ponto de vista do aplicativo:

- UDP provê transferência de grupos de bytes ("datagramas") de forma não confiável entre cliente e servidor

Interação dos *socket* cliente/servidor: UDP



Exemplo de *app*: cliente UDP

UDPCliente.py em Python

Inclui biblioteca de *socket* do Python

Cria socket UDP no cliente

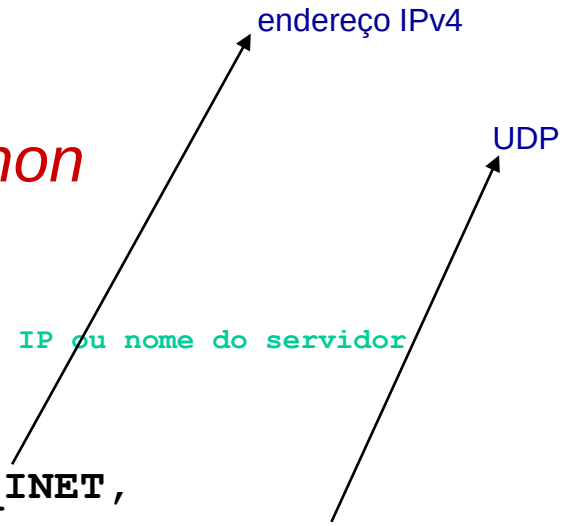
Obtém entrada do teclado do usúário

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 = input('Entre uma frase em minusc.:')
clienteSocket.sendto(mensagem,(Nomeservidor, Portaservidor))
Mensagemmodificada, Enderecoservidor= clienteSocket.recvfrom(2048)
print(Mensagemmodificada.decode())
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.decode().upper()
    servidorSocket.sendto(mensagemModificada.encode(), 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

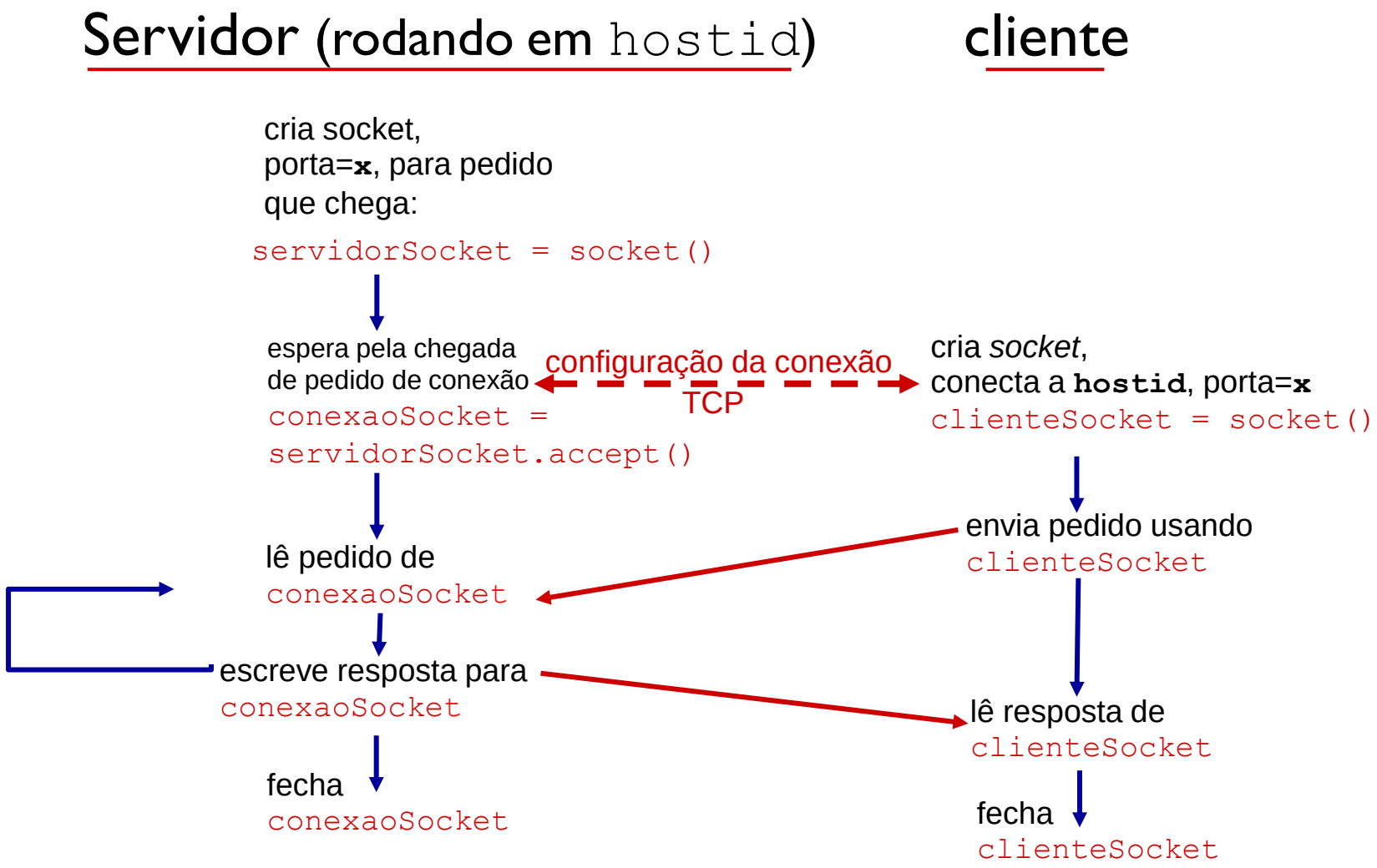
- Quando contatado por cliente, *servidor TCP cria novo socket* para processo servidor se comunicar com aquele cliente em particular

  - 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 = input('Entre com frase em minusc.:')
clienteSocket.send(frase.encode())
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).decode()
    frasemaiusc = frase.upper()
    conexaoSocket.send(frasemaiusc.encode())
    conexaoSocket.close()
```

Cria socket TCP de acolhimento

Servidor começa a escutar pedidos TCP que chegam

Laço infinito

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

- Arquiteturas de aplicativos
  - Cliente-servidor
  - P2P
- Requisitos de serviços dos aplicativos:
  - Confiabilidade, apacidade, latência
- Modelo de serviço de transporte Internet
  - Orientada a conexão, confiável: TCP
  - Não confiável, datagramas: UDP

- Exemplos de protocolos da camada de aplicação
  - HTTP
  - SMTP, POP, IMAP
  - DNS
  - Streaming, VoIP
  - P2P: BitTorrent
- Programando *sockets* : *sockets* TCP, UDP