Aula 8 - Programação com Sockets

Diego Passos

Universidade Federal Fluminense

Redes de Computadores I

Material adaptado a partir dos slides originais de J.F Kurose and K.W. Ross.

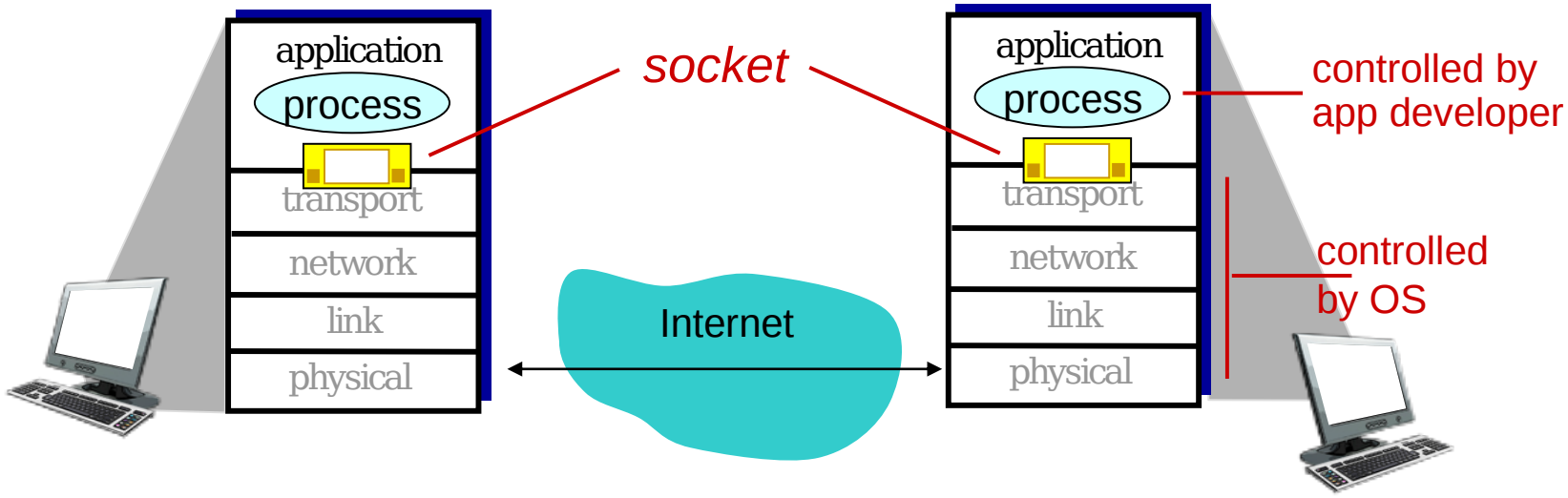
Revisão da Última Aula...

- **DNS: objetivo.**
  - Sistema que mapeia IPs a nomes.
  - Simplifica identificação de *hosts*.
- **DNS: características.**
  - Base de dados distribuída.
  - Nomeação hierárquica.
    - Domínios, subdomínios, ...
  - **Evita ponto único de falha**.
  - Evita concentração do tráfego.
  - Evita distância excessiva de certos clientes.
- **DNS: tipos de servidores.**
  - Raiz, TLD, autoritativo, local.
- **DNS: métodos de resolução.**
  - **Iterativo**: servidor responde com próximo servidor a ser consultado.
  - **Recursivo**: servidor assume responsabilidade de achar o mapeamento.
- **DNS: resgistros.**
  - Tipo=A: definição de **nome canônico**.
  - Tipo=NS: definição de servidor autoritativo para o domínio.
  - Tipo=CNAME: definição de apelidos para *hosts*.
  - Tipo=MX: definição de servidor de e-mail para o domínio.
- **P2P: escalabilidade.**
  - Mais demanda, mais oferta.
  - Desde que pares contribuam.
    - *i.e.*, evitar ***free-riders***.
    - Bit-torrent: *tit-for-tat*.

Conceitos Básicos

Programação com Sockets (I)

- **Objetivo:** aprender a construir aplicações Cliente–Servidor que se comuniquem utilizando sockets.
- **Socket:** janela entre processo da aplicação e protocolo de transporte.



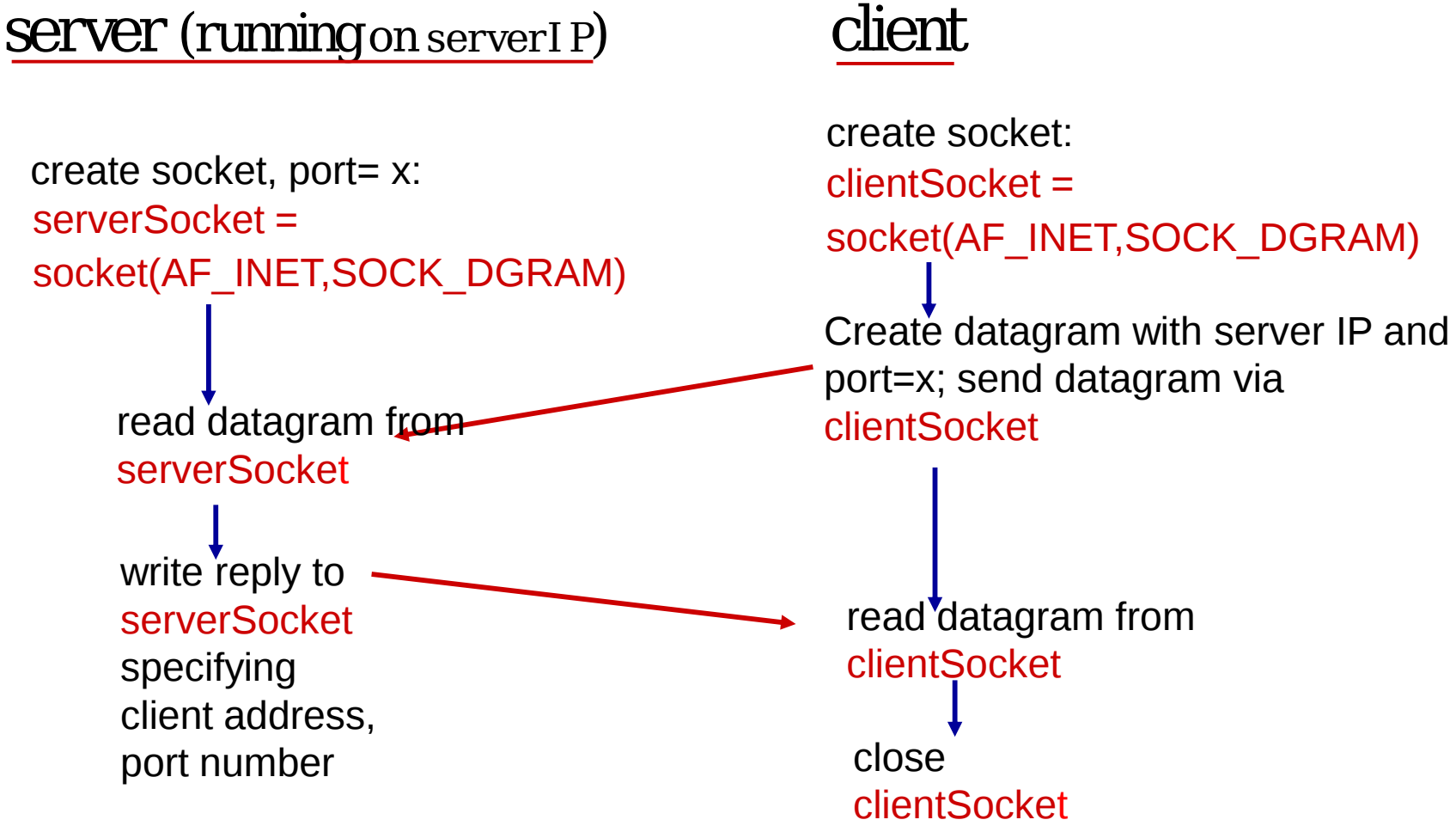
Programação com Sockets (II)

- **Dois tipos de socket para dois modelos de serviço de transporte:**
  - **UDP:** serviço de datagramas não-confiável.
  - **TCP:** serviço de entrega confiável, orientado a fluxo de bytes.
- **Aplicação de exemplo**:
  1. Cliente lê *string* do teclado e envia o dado para o servidor.
  2. O servidor recebe o dado e converte a *string* para caixa alta.
  3. Servidor envia dados modificados para o cliente.
  4. Cliente recebe dado modificado e imprime na tela.

Programação com Sockets UDP

- **UDP: não há "conexão" entre cliente e servidor.**
  - Não existe handshaking antes do envio de dados.
  - Transmissor explicitamente informa o endereço IP e o número de porta de destino a cada pacote.
  - Receptor extrai endereço IP do transmissor e número de porta do pacote recebido.
- **UDP: dados transmitidos podem ser perdidos ou recebidos fora de ordem!**
- **Ponto de vista da aplicação:**
  - UDP provê serviço **não-confiável** de transmissão de grupos de bytes ("datagramas") entre cliente e servidor.

Interação entre Cliente/Servidor e o Socket: UDP



Aplicação de Exemplo: Cliente UDP

```
from socket import *

serverName = 'localhost'
serverPort = 12000

# Criacao do socket
clientSocket = socket(AF_INET, SOCK_DGRAM)

sentence = raw_input('Input lowercase sentence:')

# Envio de bytes. Repare que o endereco do destinatario eh necessario
clientSocket.sendto(sentence, (serverName, serverPort))

# Recepcao
modifiedSentence, addr = clientSocket.recvfrom(1024)
print 'From Server {}: {}'.format(addr, modifiedSentence)

# Fechamento
clientSocket.close()
```

Aplicação de Exemplo: Servidor UDP (I)

```
from socket import *

# Numero de porta na qual o servidor estara esperando conexoes.
serverPort = 12000

# Criar o socket. AF_INET e SOCK_DGRAM indicam UDP.
serverSocket = socket(AF_INET, SOCK_DGRAM)

# Associar o socket a porta escolhida. Primeiro argumento vazio indica
# que desejamos aceitar conexoes em qualquer interface de rede desse host
serverSocket.bind(('', serverPort))

print 'O servidor esta pronto para receber pacotes.'

# Loop infinito: servidor eh capaz de tratar multiplas conexoes
while 1:
```

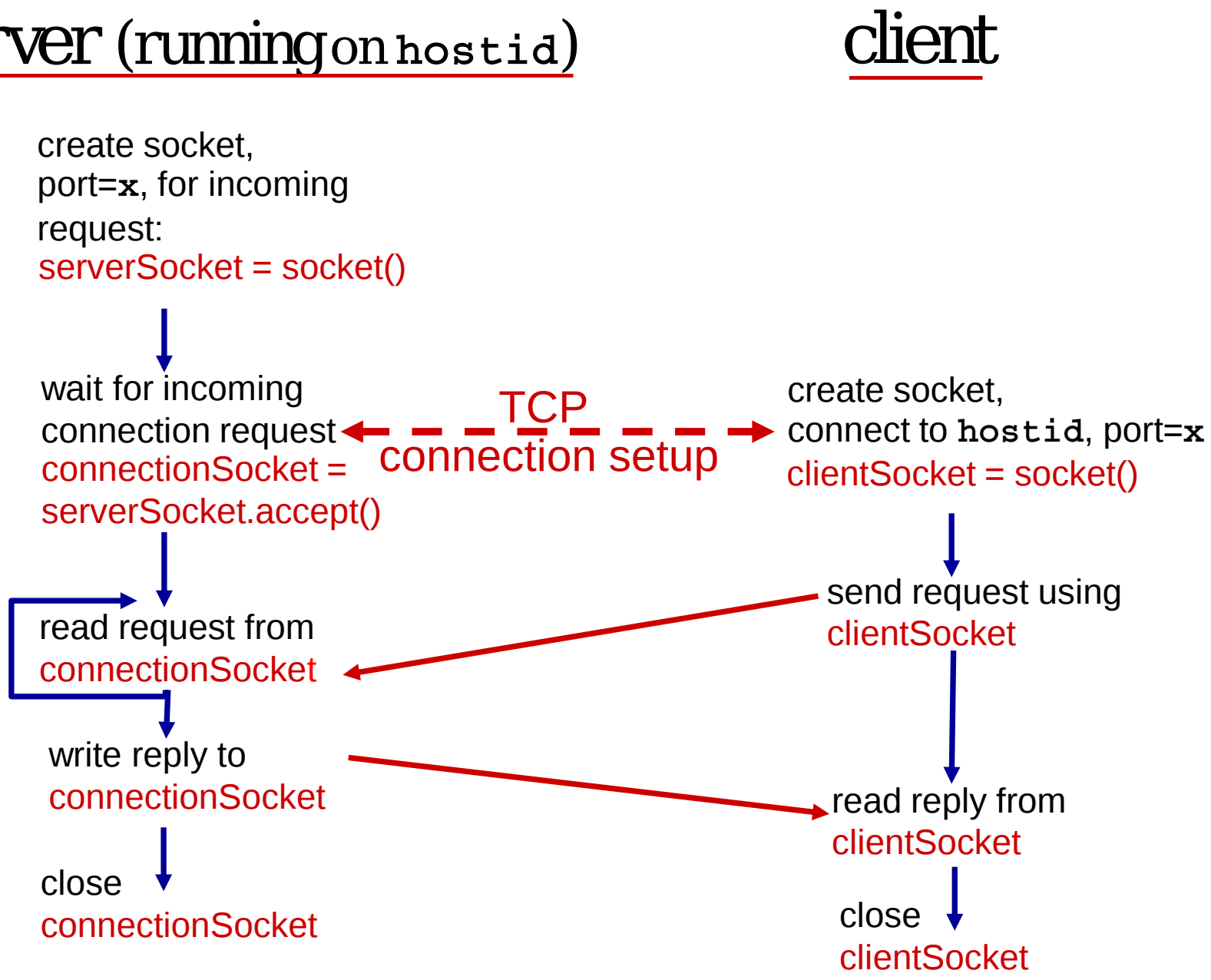
Aplicação de Exemplo: Servidor UDP (II)

```
    # Aguardar novo pacote
    print 'Aguardando pacote...'
    sentence, addr = serverSocket.recvfrom(1024)
    print 'Nova pacote recebido!'

    # Processamento
    capitalizedSentence = sentence.upper()

    # Envio. Repare que o endereco do destinatario eh necessario.
    print 'Realizando envio...'
    serverSocket.sendto(capitalizedSentence, addr)

# Fechamento
print 'Fechando socket...'
serverSocket.close()
```

Programação com Sockets TCP

- **Cliente deve contactar servidor**.
  - Processo do servidor precisa estar previamente em execução.
  - Servidor precisa ter criado socket que aceitará contato do cliente.
- **Cliente contacta servidor:**
  - Criando socket TCP, especificando IP e número de porta do processo servidor.
  - **Quando cliente cria o socket**: TCP do cliente estabelece conexão para o TCP do servidor.
- Quando contactado pelo cliente, **TCP do servidor cria um novo socket**.
  - Novo socket utilizado para a comunicação do processo servidor com o processo cliente.
  - Este esquema de dois sockets permite ao servidor falar com múltiplos clientes.
  - Número de porta **de origem** são usados para distinguir clientes.
    - Mais detalhes no próximo capítulo
- **Ponto de vista da aplicação:**
  - TCP provê transferência confiável e ordenada de fluxo de bytes entre cliente e servidor.

Interação entre Cliente/Servidor e Socket TCP



Aplicação de Exemplo: Cliente TCP

```
from socket import *

serverName = 'localhost'
serverPort = 12000

# Criacao do socket
clientSocket = socket(AF_INET, SOCK_STREAM)
# Conexao com o servidor
clientSocket.connect((serverName,serverPort))

sentence = raw_input('Input lowercase sentence:')
# Envio de bytes
clientSocket.send(sentence)

# Recepcao
modifiedSentence = clientSocket.recv(1024)
print 'From Server:', modifiedSentence

# Fechamento
clientSocket.close()
```

Aplicação de Exemplo: Servidor TCP (I)

```
from socket import *

# Numero de porta na qual o servidor estara esperando conexoes.
serverPort = 12000

# Criar o socket. AF_INET e SOCK_STREAM indicam TCP.
serverSocket = socket(AF_INET, SOCK_STREAM)

# Associar o socket a porta escolhida. Primeiro argumento vazio indica
# que desejamos aceitar conexoes em qualquer interface de rede desse host
serverSocket.bind(('', serverPort))

# Habilitar socket para aceitar conexoes. O argumento 1 indica que ate
# uma conexao sera deixada em espera, caso recebamos multiplas conexoes
# simultaneas
serverSocket.listen(1)

print 'O servidor esta pronto para receber conexoes'

# Loop infinito: servidor eh capaz de tratar multiplas conexoes
while 1:
```

Aplicação de Exemplo: Servidor TCP (II)

```
# Aguardar nova conexao
print 'Aguardando conexao...'
connectionSocket, addr = serverSocket.accept()
print 'Nova conexao recebida!'

# Recepcao de dados
print 'Aguardando dados...'
sentence = connectionSocket.recv(1024)

print 'Dado recebido do cliente'
# Processamento
capitalizedSentence = sentence.upper()
# Envio
print 'Realizando envio...'
connectionSocket.send(capitalizedSentence)
# Fechamento
print 'Fechando socket...'
connectionSocket.close()
```

Exemplos Mais Complexos

Um (Protótipo de) Servidor Web (I)

```
# Numero de porta na qual o servidor estara esperando conexoes.
serverPort = 8001

# Criar o socket. AF_INET e SOCK_STREAM indicam TCP.
serverSocket = socket(AF_INET, SOCK_STREAM)

# Associar o socket a porta escolhida. Primeiro argumento vazio indica
# que desejamos aceitar conexoes em qualquer interface de rede desse host
serverSocket.bind(('', serverPort))

# Habilitar socket para aceitar conexoes. O argumento 1 indica que ate
# uma conexao sera deixada em espera, caso recebamos multiplas conexoes
# simultaneas
serverSocket.listen(1)

# Loop infinito: servidor eh capaz de tratar multiplas conexoes
while 1:

  # Aguardar nova conexao
  connectionSocket, addr = serverSocket.accept()

  # Tratamento da conexao encapsulado em uma funcao
  trataConexao(connectionSocket)
```

Um (Protótipo de) Servidor Web (II)

```
from socket import *
import os
import re

def trataConexao(s):

    # Podemos tratar um socket como um arquivo.
    sf = s.makefile()
    # Primeira linha deve informar a requisicao. Ignoraremos as demais (e.g., cabecalhos).
    requestMessageLine = sf.readline()
    # Campos sao divididos por espacos em branco. Primeiro campo deve ser tipo
    # do metodo. Neste prototipo, tratamos apenas requisicoes do tipo GET.
    tokens = requestMessageLine.split()
    if tokens[0] == "GET":

        # Proximo campo eh o caminho do objeto requisitado.
        filename = tokens[1]

        # Arquivos servidos pelo servidor web sao confinados ao diretorio do qual ele eh
        # executado (e subdiretorios). Logo, se a requisicao referencia um caminho
        # absoluto (i.e., iniciado por '/'), precisamos transformar isso em relativo
        if filename[0] == '/':
            filename = filename[1:]
```

Um (Protótipo de) Servidor Web (III)

```
# Requisitamos a leitura do arquivo a partir do sistema de arquivos local.
if os.path.isfile(filename):

    # Levantar informacoes sobre o objeto
    statinfo = os.stat(filename)
    f = open(filename)
    # Montar a saida
    s.send('HTTP/1.0 200 OK\r\n')
    # Precisamos de algumas linhas de cabecalho. A primeira eh o tipo
    # do objeto.
    if re.search('.jpg$', filename):
        s.send('Content-Type: image/jpeg\r\n')
    elif re.search('.gif$', filename):
        s.send('Content-Type: image/gif\r\n')
    elif re.search('.html$', filename):
        s.send('Content-Type: text/html\r\n')
    elif re.search('.js$', filename):
        s.send('Content-Type: text/js\r\n')
    elif re.search('.css$', filename):
        s.send('Content-Type: text/css\r\n')
    else:
        s.send('Content-Type: application/octet-stream\r\n')
```

Um (Protótipo de) Servidor Web (IV)

```
            # Precisamos tambem do tamanho do objeto
            s.send('Content-Length: {}\r\n'.format(statinfo.st_size))

            # Cabecalho eh separado do corpo da mensagem por uma linha em branco
            s.send('\r\n')

            # Agora enviamos o conteudo do arquivo no corpo da mensagem
            s.send(f.read())
        else:

            # Se o arquivo nao foi encontrado.
            s.send('HTTP/1.0 404 Not Found\r\n')
    else:

        # Tratamento (muito basico) de erros
        print 'Requisicao nao compreendida!'

    s.close()
```

Ferramenta de Medição de Vazão TCP: Cliente (I)

```
from socket import *
import sys
import time

# Leitura do endereco do servidor
serverName = sys.argv[1]
serverPort = 6789

# Criacao do socket
clientSocket = socket(AF_INET, SOCK_STREAM)
# Conexao com o servidor
clientSocket.connect((serverName,serverPort))

# Criacao de um buffer para envio pelo socket. O conteudo nao eh importante.
buffer = bytearray(8192)

# Pegar o horario atual para sabermos quando terminar o teste.
now = int(round(time.time() * 1000))

# Teste sempre dura 10 segundos = 10000 ms.
end = now + 10000
```

Ferramenta de Medição de Vazão TCP: Cliente (II)

```
# Variavel i guarda o numero de vezes que o buffer foi escrito no socket.
# Auxiliara no calculo da vazao.
i = 0
while True:
    # Envio de bytes
    clientSocket.send(buffer)
    i = i + 1
    now = int(round(time.time() * 1000))
    if now >= end:
        break

# Calcular e imprimir a vazao. A cada iteracao do loop anterior, transmitimos
# 64 kb. Para calcular vazao, basta multiplicar i por 64 e dividir por 10.
print "Vazao media durante o teste foi de {} kb/s".format(i * 64 / 10.0)

# Fechamento
clientSocket.close()
```

Ferramenta de Medição de Vazão TCP: Servidor (I)

```python
from socket import *

serverPort = 6789

# Criacao do socket.
serverSocket = socket(AF_INET,SOCK_STREAM)

# Associacao do socket a porta correta.
serverSocket.bind(('',serverPort))

# Habilitar escuta por conexoes.
serverSocket.listen(1)
```

Ferramenta de Medição de Vazão TCP: Servidor (II)

```
# Loop infinito para tratar multiplas conexoes
while 1:

    print 'Aguardando conexao...'
    # Aguardar nova conexao
    connectionSocket, addr = serverSocket.accept()

    print 'Conexao recebida.'

    # Recepcao de dados
    while True:
        try:
            # Leitura dos dados
            buffer = connectionSocket.recv(8192)
        except socket.error, exc:
            # Houve algum erro. Assumimos que a conexao foi fechada
            break

    connectionSocket.close()
    print 'Conexao encerrada'
```

Outras Linguagens: Funções/Métodos Típicos

Funções/Métodos Tipicamente Utilizados

Cliente

- **socket():** criar novo socket de um determinado tipo.
- **write():** "passa" dados/mensagens pelo socket p/ transporte.
- **sendto():** envia mensagem por socket sem conexão (UDP).
- **read():** "recebe" dados/mensagens pelo socket do transporte.
- **recvfrom():** recebe mensagem por socket sem conexão (UDP).
- **connect():** abre uma conexão (TCP) para servidor/porta especificados.
- **getByName() ou getHostByName():** resolve nome para endereço IP.
- **close():** fecha o socket (e conexão, se aplicável).

Servidor

- **socket():** criar novo socket de um determinado tipo.
- **write():** "passa" dados/mensagens pelo socket p/ transporte.
- **sendto():** envia mensagem por socket sem conexão (UDP).
- **read():** "recebe" dados/mensagens pelo socket do transporte.
- **recvfrom():** recebe mensagem por socket sem conexão (UDP).
- **bind():** associa socket à porta especificada.
- **listen():** habilita socket (TCP) a receber conexões.
- **close():** fecha o socket (e conexão, se aplicável).

Sockets em Outras Linguagens: Java (Cliente TCP, I)

```
import java.io.*;
import java.net.*;

class TCPClient {

  public static void main(String argv[]) throws Exception {

      String sentence;
    String modifiedSentence;

    BufferedReader inFromUser = new BufferedReader(new InputStreamReader(System.in));

    // Criação do socket TCP. Note que aqui, diferentemente da versão UDP, já especificamos
    // o endereço do servidor (nome do host/ip e porta).
    Socket clientSocket = new Socket("hostname", 6789);

    // Do ponto de vista do programador, um socket TCP pode ser manipulado de forma similar
    // a um arquivo, com escrita e leitura de um fluxo de bytes.
    DataOutputStream outToServer = new DataOutputStream(clientSocket.getOutputStream());
    BufferedReader inFromServer =  new BufferedReader(new InputStreamReader(
                                                    clientSocket.getInputStream()));
    // Leitura da entrada do usuário.
    sentence = inFromUser.readLine();
```

Sockets em Outras Linguagens: Java (Cliente TCP, II)

```
    // String é simplesmente "escrita" no socket. Notem que adicionamos uma quebra de linha
    // ao final da string (caractere '\n'). Isso demarcará ao servidor onde termina a mensagem
    // a ser processada.
    outToServer.writeBytes(sentence + '\n');

    // Aguardamos uma resposta do servidor. Note mais uma vez a manipulação do socket como
    // se fosse um arquivo. Aqui também uma quebra de linha denota fim da mensagem. Por fim,
    // assim como no cliente UDP, leituras são (geralmente) bloqueantes.
    modifiedSentence = inFromServer.readLine();

    // Impressão do resultado da tela.
    System.out.println("FROM SERVER: " + modifiedSentence);

    // Fechamento do socket.
    clientSocket.close();
  }
}
```

Sockets em Outras Linguagens: Java (Servidor TCP, I)

```java
import java.io.*;
import java.net.*;

class TCPServer {

  public static void main(String argv[]) throws Exception {
    String clientSentence;
    String capitalizedSentence;

    // Criação do socket do servidor. Este socket será usado para esperar por novas conexões.
    // Repare que especificamos um # de porta na qual desejamos esperar pelas conexões.
    ServerSocket welcomeSocket = new ServerSocket(6789);

    // Assim como o servidor UDP, servidor TCP também executa um loop infinito permitindo
    // o atendimento de múltiplos clientes.
    while(true) {

      // Função/método accept(): executada sobre socket, diz ao SO para aguardar (e aceitar)
      // novas conexões. Só faz sentindo para sockets orientados a conexão (TCP). Note que
      // o resultado da função/método é um novo socket.
      Socket connectionSocket = welcomeSocket.accept();
```

Sockets em Outras Linguagens: Java (Servidor TCP, II)

```
        // O socket original é serve apenas para aguardar por novas conexões. Já o socket
        // retornado pela função/método accept representa uma conexão, realmente. É dele que
        // "leremos" os dados enviados pelo cliente e escreveremos os dados de resposta. Mais
        //  uma vez, note a abstração de arquivo.
        BufferedReader inFromClient = new BufferedReader(new
                                          InputStreamReader(connectionSocket.getInputStream()));
        DataOutputStream  outToClient = new DataOutputStream(connectionSocket.getOutputStream());

        // Aguardamos dados do cliente. Por convenção, dados terminam em uma quebra de linha.
        clientSentence = inFromClient.readLine();

        // Implementação da lógica da aplicação.
        capitalizedSentence = clientSentence.toUpperCase() + '\n';

        // Escrita do resultado no socket.
        outToClient.writeBytes(capitalizedSentence);
    }
  }
}
```

Resumo da Aula...

- **Sockets**: API para aplicações de rede.
  - Presente na maioria das linguagens.
  - Abstrações similares.
    - Criação do socket, conexão, envio de dados, recepção, fechamento.
  - Fornece modelos de serviço diferentes: UDP *vs.* TCP.
    - Sem conexão *vs.* orientado a conexão.
    - Sem confiabilidade *vs.* entrega confiável de dados.
    - ...
  - **Bind()**: associa socket a uma porta.
    - **Não pode haver dois ou mais sockets associados à mesma porta de um mesmo protocolo de transporte!**

Resumo do Capítulo 2 (I)

- **Nosso estudo sobre as aplicações de rede está completo!**

- Arquiteturas de aplicação.
  - Cliente—Servidor.
  - P2P.
- Requisitos das aplicações.
  - Confiabilidade, vazão mínima, atraso máximo.
- Modelos de serviço da camada de transporte da Internet.
  - Serviço confiável, orientado a conexão: TCP.
  - Serviço não confiável, orientado a datagramas: UDP.
- Protocolos específicos.
  - HTTP.
  - FTP.
  - SMTP, POP, IMAP.
  - DNS.
  - P2P: BitTorrent.
- Programação com sockets.
  - TCP.
  - UDP.

Resumo do Capítulo 2 (II)

- Também discutimos sobre o funcionamento geral de protocolos.

- Modelo de funcionamento baseado em requisição e resposta.
  - Cliente requisita informação ou serviço.
  - Servidor responde com dados, código de *status*.
- Formatos de mensagem.
  - Cabeçalhos: campos contendo informação sobre dados.
  - Dados: informação útil sendo comunicada.

- **Conceitos importantes:**
  - Mensagens de controle *vs.* mensagens de dados.
    - Comunicação em-banda e fora-de-banda.
  - Soluções centralizadas *vs.* descentralizadas.
  - *Stateless vs. Stateful*.
  - Transferência confiável *vs.* não confiável de dados.
  - "Complexidade nas bordas".

Próxima Aula...

- Capítulo sobre camada de aplicação está encerrado.
  - Assim como o conteúdo para a primeira prova: capítulos 1 e 2.
- Na próxima aula, iniciaremos um novo tópico: camada de transporte.
- Na primeira aula:
  - Conceitos básicos.
  - Modelos de serviço.
  - O protocolo UDP.