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.

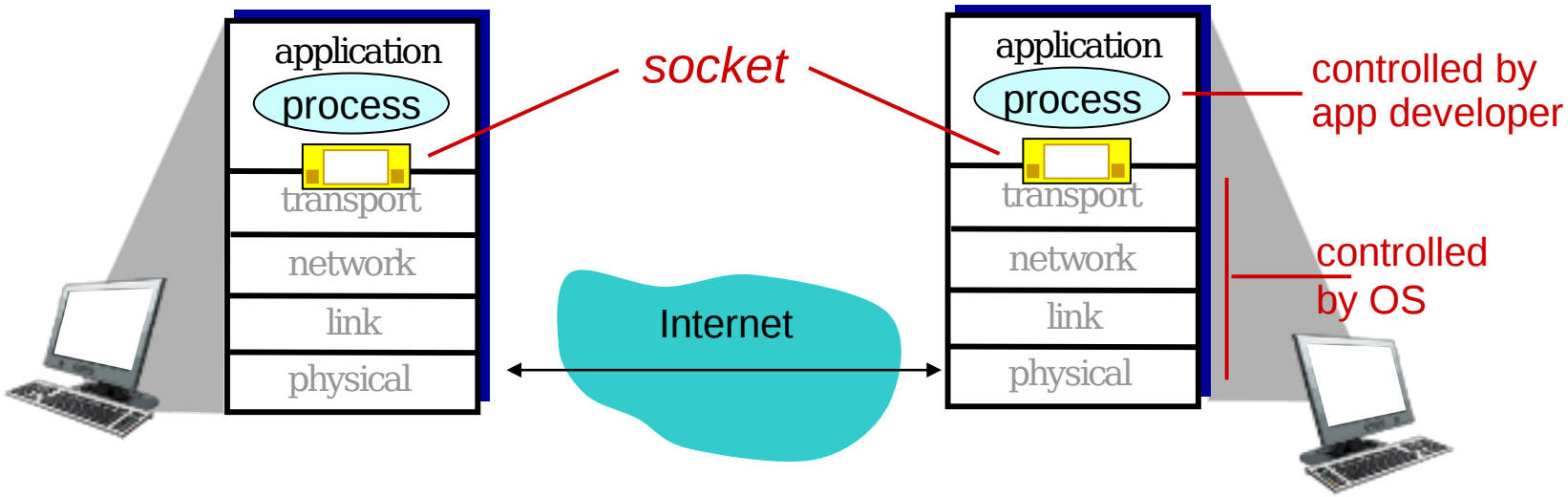
Revisão da Última Aula...

- **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.



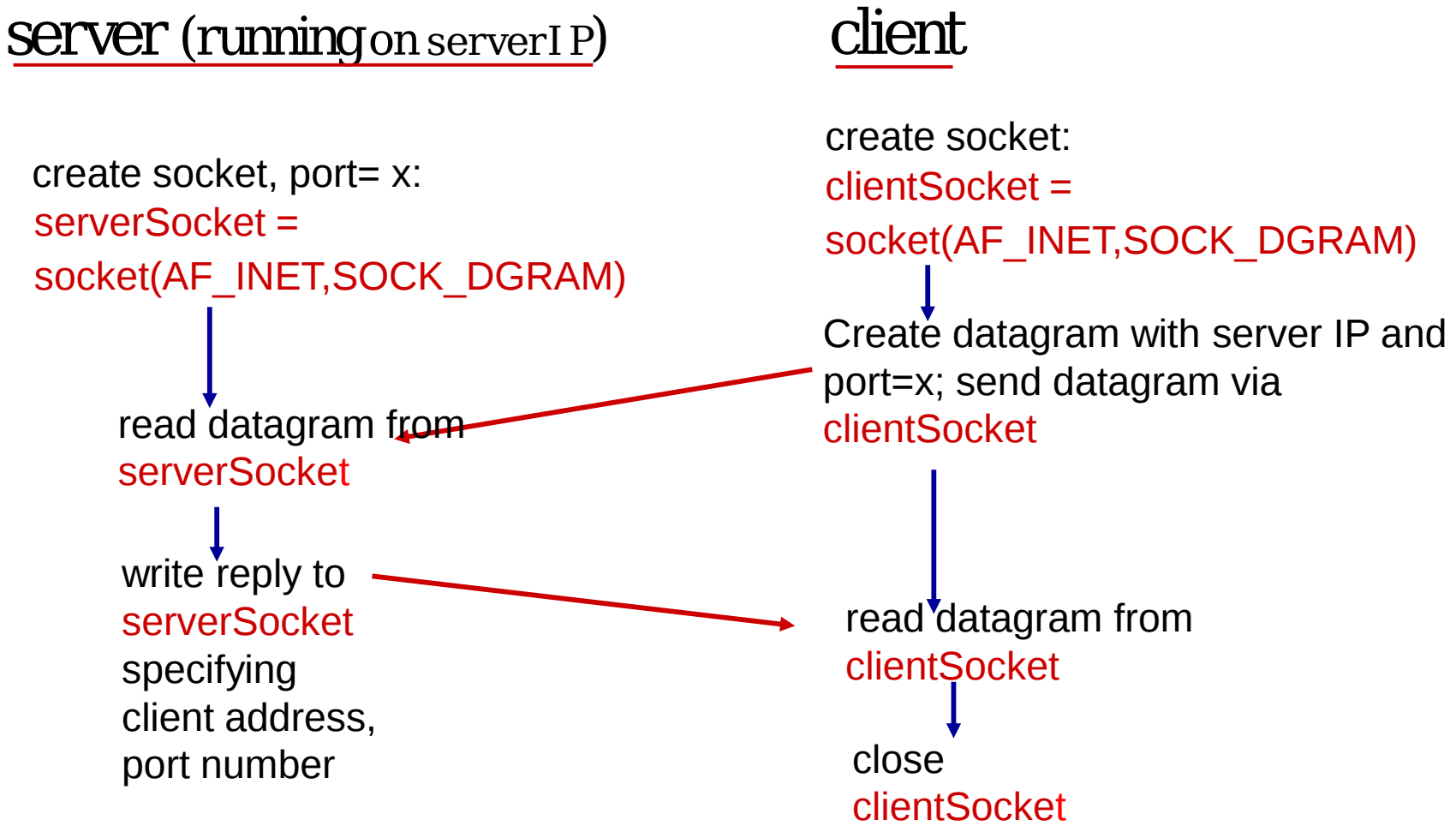
Programação com Sockets (II)

- **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 (I)

```
import java.io.*;
import java.net.*;      // API de sockets.

class UDPClient {

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

    BufferedReader inFromUser = new BufferedReader(new InputStreamReader(System.in));
    // Criação de Socket UDP (datagramas)
    DatagramSocket clientSocket = new DatagramSocket();
    // Resolução de nome de host.
    InetAddress IPAddress = InetAddress.getByName("hostname");

    // Alocação de buffers para mensagens transmitida e recebida
    byte[] sendData = new byte[1024];
    byte[] receiveData = new byte[1024];

    // Leitura de dados do usuário
    String sentence = inFromUser.readLine();
    // Formatação da mensagem da aplicação
    sendData = sentence.getBytes();
```

Aplicação de Exemplo: Cliente UDP (II)

```
    // Criação do datagrama e envio. Note a especificação do endereço
    // de destino (IP e porta).
    DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length, IPAddress, 9876);
    clientSocket.send(sendPacket);

    // Espera pela resposta. Funções/métodos de recepção são (normalmente) bloqueantes.
    DatagramPacket receivePacket = new DatagramPacket(receiveData, receiveData.length);
    clientSocket.receive(receivePacket);

    // Apresentação do resultado.
    String modifiedSentence = new String(receivePacket.getData());
    System.out.println("FROM SERVER:" + modifiedSentence);

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

Aplicação de Exemplo: Servidor UDP (I)

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

class UDPServer {

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

    // Criação do socket. Note que especificamos o # de porta na qual esperamos por datagramas.
    DatagramSocket serverSocket = new DatagramSocket(9876);
    byte[] receiveData = new byte[1024]; // Buffer de recepção de dados.
    byte[] sendData = new byte[1024]; // Buffer para envio de dados.

    // Servidores normalmente executam um loop infinito. Cada iteração representa o atendimento
    // a um cliente diferente.
    while(true) {

      // Criação de um datagrama para recepção de mensagem.
      DatagramPacket receivePacket = new DatagramPacket(receiveData, receiveData.length);

      // Aguardar recepção de um novo datagrama. Novamente, métodos/funções de recepção são,
      // em geral, bloqueantes.
      serverSocket.receive(receivePacket);
```

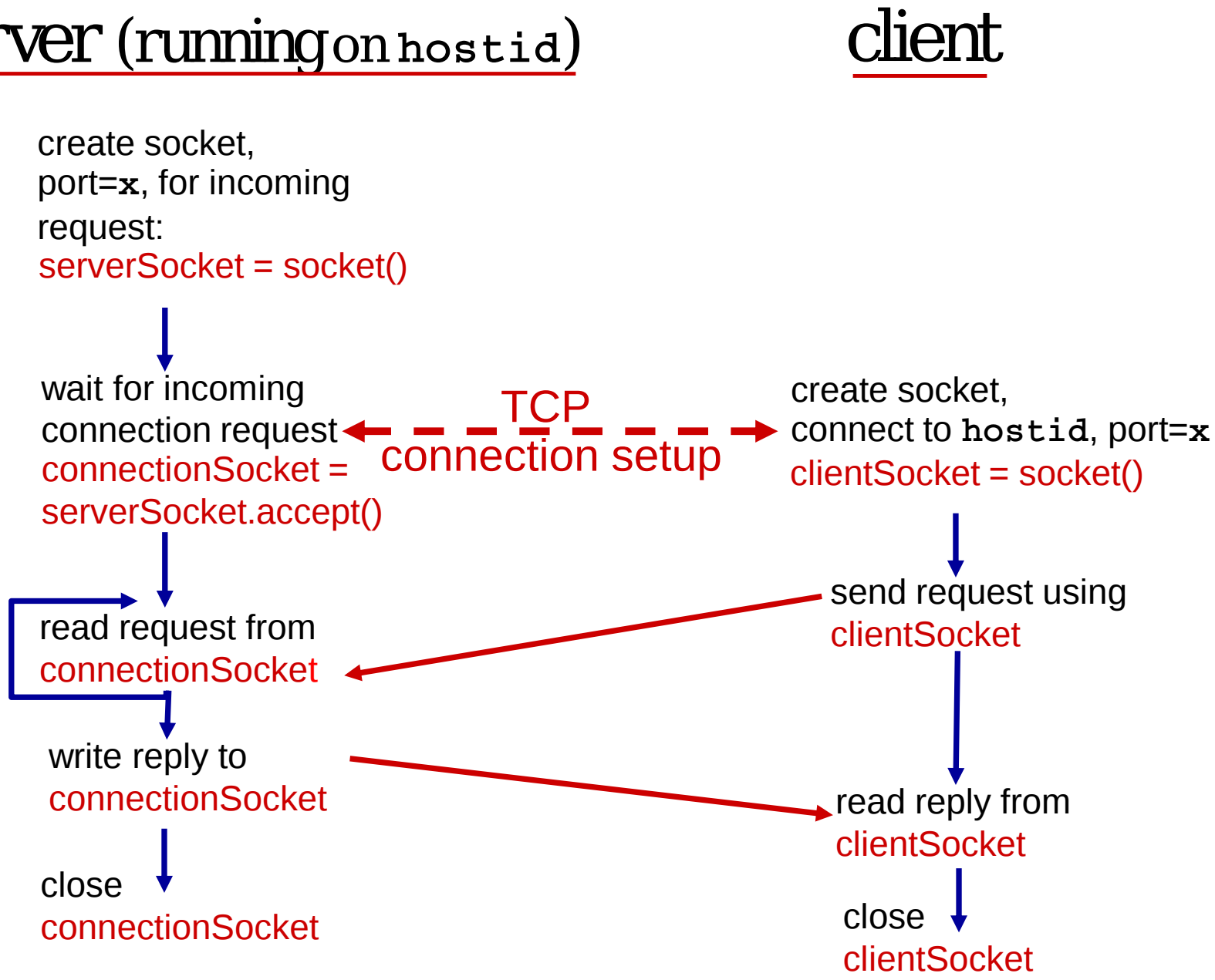
Aplicação de Exemplo: Servidor UDP (II)

```
        // Tratamento da mensagem. Aqui, é aplicada a lógica específica da aplicação.
        // No caso, apenas interpretamos os bytes da mensagem como uma string e calculamos
        // uma versão alternativa em caixa alta.
        String sentence = new String(receivePacket.getData());
        String capitalizedSentence = sentence.toUpperCase();

        // Preparação da resposta: é preciso descobrir o endereço do cliente (IP e porta).
        // Ambas as informações constam no datagrama recebido.
        InetAddress IPAddress = receivePacket.getAddress();
        int port = receivePacket.getPort();

        // Criação do datagrama de resposta. Transferimos a string para o buffer de envio e
        // construímos um datagrama a partir dele. Note, novamente, a especificação do
        // endereço de destino.
        sendData = capitalizedSentence.getBytes();
        DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length,
                                                       IPAddress, port);

        // Envio em si do datagrama.
        serverSocket.send(sendPacket);
      }
    }
}
```

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 (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();
```

Aplicação de Exemplo: 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();
  }
}
```

Aplicação de Exemplo: Servidor TCP (I)

```
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();
```

Aplicação de Exemplo: 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);
    }
  }
}
```

Exemplos Mais Complexos

Um (Protótipo de) Servidor Web (I)

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

class WebServer {

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

    // Servidor web ouve na porta 8001, ao invés da tradicional 80.
    ServerSocket listenSocket = new ServerSocket(8001);

    while(true) {
      Socket connectionSocket = listenSocket.accept(); // Aguarda conexão.

      // Tratamento da requisição está encapsulado em outro método.
      trataRequisicao(connectionSocket);
    }
  }

  private static void trataRequisicao(Socket connectionSocket) throws Exception {

    String requestMessageLine;
    String fileName;
```

Um (Protótipo de) Servidor Web (II)

```
// Criamos streams a partir do socket de conexão.
BufferedReader inFromClient =
          new BufferedReader(new InputStreamReader(connectionSocket.getInputStream()));
DataOutputStream outToClient = new DataOutputStream(connectionSocket.getOutputStream());

// Primeira linha deve informar a requisição. Ignoraremos as demais (e.g., cabeçalhos).
requestMessageLine = inFromClient.readLine();

// Campos são divididos por espaços em branco. Primeiro campo deve ser tipo
// do método. Neste protótipo, tratamos apenas requisições do tipo GET.
StringTokenizer tokenizedLine = new StringTokenizer(requestMessageLine);
if (tokenizedLine.nextToken().equals("GET")){

    // Próximo campo é o caminho do objeto requisitado.
    fileName = tokenizedLine.nextToken();

    // Arquivos servidos pelo servidor web são confinados ao diretório do qual ele é
    // executado (e subdiretórios). Logo, se a requisição referencia um caminho
    // absoluto (i.e., iniciado por '/'), precisamos transformar isso em um caminho
    // relativo ao diretório corrente.
    if (fileName.startsWith("/") == true)
      fileName  = fileName.substring(1);
```

Um (Protótipo de) Servidor Web (III)

```
File file = new File(fileName);
if (file.exists()) {
  int numOfBytes = (int) file.length();
  FileInputStream inFile  = new FileInputStream (fileName);
  byte[] fileInBytes = new byte[numOfBytes];
  inFile.read(fileInBytes);

  // Composição da mensagem de resposta: começamos com a linha de status.
  outToClient.writeBytes("HTTP/1.0 200 OK\r\n");
  // Precisamos de algumas linhas de cabeçalho na resposta. A primeira para
  // informar o tipo do arquivo.
  if (fileName.endsWith(".jpg")) outToClient.writeBytes("Content-Type: image/jpeg\r\n");
  else if (fileName.endsWith(".gif"))   outToClient.writeBytes("Content-Type: image/gif\r\n");
  else if (fileName.endsWith(".html")) outToClient.writeBytes("Content-Type: text/html\r\n");
  // ...
  // Outra linha de cabeçalho: tamanho do conteúdo anexado ao corpo da resposta.
  outToClient.writeBytes("Content-Length: " + numOfBytes + "\r\n");
  // Cabeçalhos são separados do corpo por uma linha em branco no HTTP.
  outToClient.writeBytes("\r\n");
  // Colocamos os bytes do objeto no corpo da mensagem.
  outToClient.write(fileInBytes, 0, numOfBytes);
}
```

Um (Protótipo de) Servidor Web (IV)

```
        else {
          // Objeto não encontrado.
          outToClient.writeBytes("HTTP/1.0 404 Not Found\r\n");
        }
      }
      // Tratamento (muito básico) de erros.
      else System.out.println("Bad Request Message");

      // Fechamos o socket da conexão.
      connectionSocket.close();
  }
}
```

Ferramenta de Medição de Vazão TCP: Cliente (I)

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

class BWTestClient {

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

    byte buffer[] = new byte[8192];
    int i = 0;
    long endTime, now;

    // 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("localhost", 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()));
```

Ferramenta de Medição de Vazão TCP: Cliente (II)

```
    // Armazenar hora do final do teste (testes sempre têm 10 segundos).
    endTime = System.currentTimeMillis() + 10000;

    // Simplesmente escrevemos continuamente no socket. Escritas *normalmente* não são
    // bloqueantes, mas o TCP limitará a taxa de envio de acordo com a capacidade da
    // rede. Quando excedermos esta capacidade, a chamada bloqueará.
    while(true) {
      outToServer.write(buffer);
      i = i + 1;
      now = System.currentTimeMillis();
      if (now >= endTime) break ;
    }

    // Impressão do resultado da tela. A cada iteração do loop anterior, transmitimos
    // 64 kb. Para calcular vazão, basta multiplicar i por 64 e dividir por 10.
    System.out.println("Vazão (kb/s): " + (i * 64 / 10.0));

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

Ferramenta de Medição de Vazão TCP: Servidor (I)

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

class BWTestServer {

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

    char buffer[] = new char[8192];

    // 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();
```

Ferramenta de Medição de Vazão TCP: Servidor (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()));

        // Simplesmente, aguardamos dados do cliente, indefinidamente.
        while(true) {

          try {

            if (inFromClient.read(buffer) < 0) break ;
          }
          catch(IOException e) {

            // Cliente fechou a conexão.
            break ;
          }
        }
      }
    }
}
```

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: Python (Cliente TCP)

```
from socket import *

serverName = 'servername'
serverPort = 12000

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

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

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

# Fechamento
clientSocket.close()
```

Sockets em Outras Linguagens: Python (Servidor TCP)

```
from socket import *

serverPort = 12000

# Criação do socket, associação à porta 12000 e habilitar escuta por conexões
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind((‘’,serverPort))
serverSocket.listen(1)

print ‘The server is ready to receive’

while 1:

  # Aguardar nova conexão
  connectionSocket, addr = serverSocket.accept()
  # Recepção de dados
  sentence = connectionSocket.recv(1024)
  capitalizedSentence = sentence.upper()
  # Envio
  connectionSocket.send(capitalizedSentence)
  # Fechamento
  connectionSocket.close()
```

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.