Sistemas Distribuídos

Anhanguera

AVALIE SUA PROFISSÃO

QUANDO APARECER EM SEU PORTAL UMA AVALIAÇÃO SOBRE SEU CURSO, RESPONDA:

NOTAS **9 OU 10** SIGNIFICA QUE VOCÊ INDICA

NOTAS **7 OU 8** SIGNIFICA QUE VOCÊ NÃO INDICA

Anhanguera

Anhanguera

Utilizando sockets com Java

Anhanguera

A comunicação entre máquinas em rede é essencial para diversas aplicações que utilizamos, tanto dentro de empresas, quanto no nosso dia a dia pessoal. Para facilitar o entendimento dos diferentes elementos envolvidos em uma comunicação em rede de computadores, frequentemente utilizamos o modelo de comunicação de 7 camadas ISO/OSI como referência (MAIA, 2013). Nesse modelo, a comunicação dos dados de uma máquina para outra ocorre a partir da quarta camada (a chamada camada de transporte). Nessa camada, destacam-se dois protocolos muito utilizados para realizar tal comunicação: o protocolo TCP e o protocolo UDP. Apesar da maioria das comunicações utilizarem o protocolo TCP – razão pela qual utilizaremos esse protocolo de comunicação na nossa aplicação de bate-papo – não significa que o protocolo TCP seja melhor que o protocolo UDP. O que ocorre é que esses dois protocolos possuem finalidades diferentes, o que implica que não existe um protocolo "melhor".

Vamos analisar, primeiramente, o protocolo TCP, citar suas vantagens e desvantagens.

O Protocolo de Controle de Transmissão – TCP (do inglês, Transmission Control Protocol), é um protocolo utilizado como base para comunicação entre máquinas, onde, caso haja alguma perda de informação, durante a transmissão, aquela informação seja retransmitida (MAIA, 2013). Por essa razão, é dito que o TCP é um protocolo orientado à conexão. Assim, esse é o protocolo de escolha para comunicações que não necessitem de uma transmissão em tempo real, ou seja, que não são muito sensíveis a atrasos, já que essa retransmissão leva mais tempo para ser realizada.

Por outro lado, o protocolo UDP (do inglês, User Datagram Protocol), é um protocolo utilizado como base para comunicação entre máquinas nas quais é importante que o atraso entre o envio e o recebimento da mensagem seja minimizado. Por essa razão, é dito que o UDP é um protocolo que não é orientado à conexão (referido como connectionless). Assim, esse é o protocolo de escolha para comunicações que necessitem de uma transmissão em tempo real, ou seja, não podem ocorrer atrasos, já que esse protocolo não retransmite a informação, como é o caso para a maioria das aplicações de tempo real (real time).

Anhanguera

Acredito que agora que você sabe as principais características e diferenças entre os protocolos de comunicação TCP e UDP, consegue entender o porquê de utilizar o protocolo TCP em uma aplicação de bate-papo (chat), certo?

Agora podemos passar para o próximo assunto, a comunicação através de sockets.

Os sockets utilizam o TCP (ou UDP) para realizar a comunicação entre aplicações que estejam sendo executadas em um sistema operacional, razão pela qual essa comunicação é chamada de interprocessos. Além disso, os sockets abstraem, do programador, a necessidade de um aprofundamento nas camadas mais inferiores de comunicação (camadas 1 a 3 do modelo de referência ISO/OSI).

Para que essa abstração possa ocorrer, existem funcionalidades (por vezes chamadas de primitivas) que normalmente são fornecidas por qualquer implementação de socket; em qualquer linguagem de programação orientada a objetos, essas funcionalidades representam métodos, já implementados, em determinadas classes relativas à comunicação via rede. Um resumo dessas funcionalidades é apresentado na Quadro 4.1.

Significado	Significado
Socket	Cria um novo terminal de comunicação.
Bind	Atrela um endereço IP local a um *socket*.
Listen	Aviso de que o socket está aceitando conexões.
Accept	Aguarda o recebimento de uma solicitação de conexão.
Connect	Ativamente tenta estabelecer conexão com um *socket*.
Send	Envia dados através de uma conexão previamente estabelecida.
Receive	Recebe dados através de uma conexão previamente estabelecida.
Close	Libera a conexão.

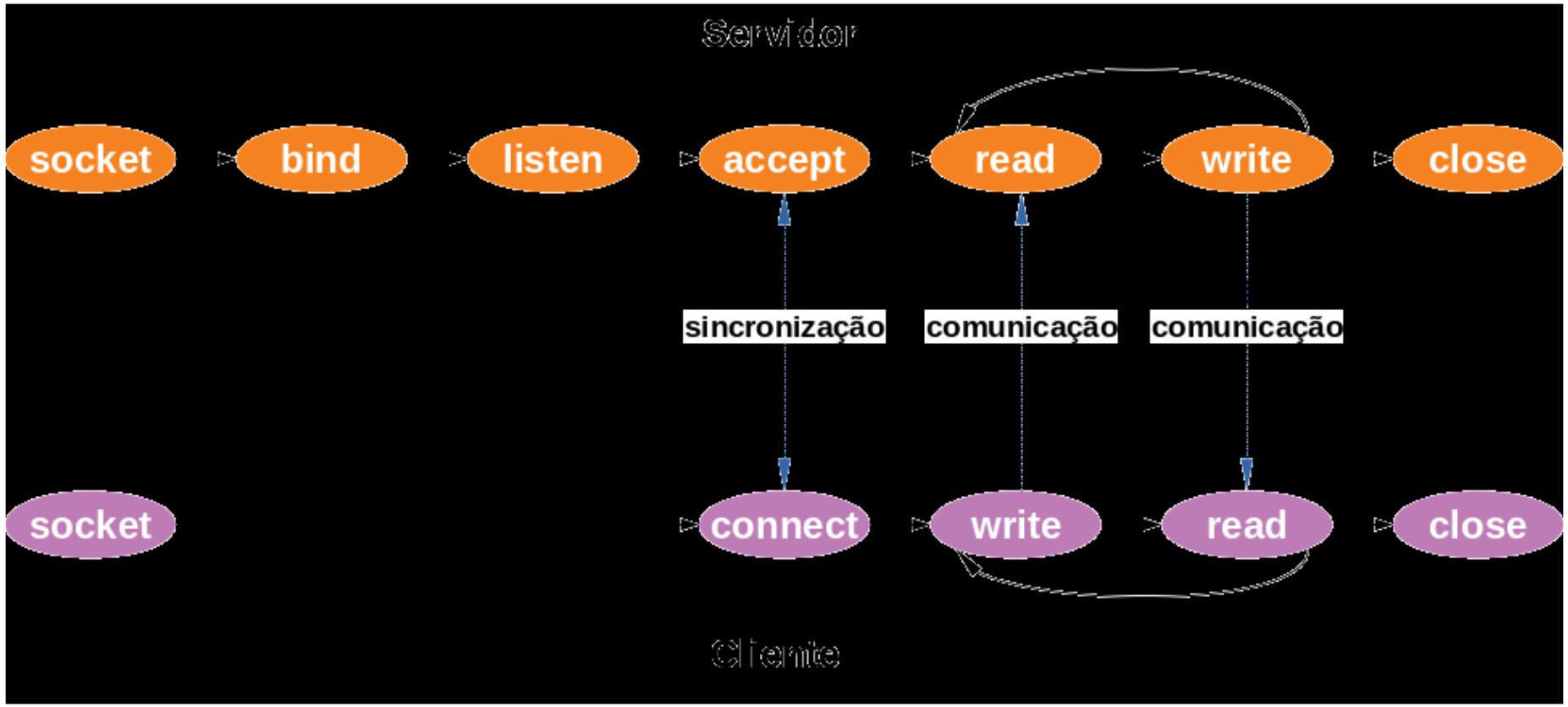
Em termos práticos, um socket é uma combinação de endereço IP e porta de comunicação. Conforme observa Coulouris et al. (2013), para um processo de enviar e receber mensagens, seu socket precisa estar atrelado a uma porta e a um endereço IP roteável na máquina onde esse processo está sendo executado.

Observe que existe uma ordem “natural” de chamada dessas primitivas, tanto por uma máquina atuando como cliente, quanto por uma máquina atuando como servidor, para que a comunicação entre essas máquinas possa ocorrer. Como veremos a seguir em nosso exemplo com a linguagem Java.

Do ponto de vista de um servidor, este deve executar a primitiva “socket”, momento no qual o servidor cria um novo terminal de comunicação, utilizando o protocolo de transporte apropriado à aplicação (TCP ou UDP). Internamente o sistema operacional aloca os recursos necessários para que essa máquina possa enviar e receber mensagens obedecendo as regras daquele – previamente definido – protocolo de transporte. Então deve ser executada a primitiva “bind”, responsável por associar um socket específico (conjunto de endereço IP e porta de comunicação, conforme mencionado anteriormente) ao servidor, de forma que o sistema operacional saiba dessa associação.

A seguir, deve ser executada a primitiva “listen”, que faz com que o sistema operacional reserve recursos suficientes para ser capaz de lidar com a quantidade de conexões que o servidor suporta. A chamada à primitiva “accept” fica aguardando o estabelecimento de uma conexão por uma máquina cliente; quando esta ocorre, o sistema operacional retorna um socket com as mesmas características daquele criado pelo servidor, para a máquina cliente. Como veremos a seguir em nosso exemplo com a linguagem Java.

Agora vamos analisar as primitivas a serem executadas pela máquina cliente. Essa máquina também precisa criar um socket, mas não há necessidade de especificar uma porta de comunicação, pois esta é alocada automaticamente pelo sistema operacional, uma vez que seu número não precisa ser explicitamente especificado na máquina remota (diferentemente do que acontece com relação à porta de comunicação do lado do servidor). A primitiva “connect” recebe a informação de qual protocolo de transporte foi adotado pelo servidor e, ao ser executada, fica aguardando o estabelecimento de uma conexão com a máquina servidora. Uma vez que esta foi estabelecida, é possível utilizar as primitivas de “send” e “receive” para enviar e receber dados (ou seja, trocar informações ou mensagens entre as máquinas). Por fim, podemos executar a primitiva “close” para explicitamente encerrar a conexão e liberar os recursos eventualmente utilizados por esta, como por exemplo, a própria porta de comunicação. A Figura 4.5 foi criada a fim de ilustrar esse processo de comunicação de maneira mais integrada.


Servidor
socket
bind
listen
accept
read
write
close
sincronização
comunicação
comunicação
socket
connect
write
read
close
Cliente

Agora que você entendeu as etapas envolvidas no processo de comunicação de máquinas utilizando sockets, vamos analisar um código Java para uma aplicação – didática – de um serviço de bate-papo. Na Figura 4.6 é apresentado o código que seria compilado e executado no servidor, enquanto que na Figura 4.7 é apresentado o código que seria compilado e executado
no cliente.

```java
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

public class Servidor {

	public static void main(String[] args) {
		Socket soc = null;
		ServerSocket socServidor = null;
		try {
			socServidor = new ServerSocket(5001);
			soc = socServidor.accept();
			DataInputStream recebido = new DataInputStream(soc.getInputStream());
			DataOutputStream enviado = new DataOutputStream(soc.getOutputStream());
			System.out.println("(cliente): " + recebido.readUTF());
			enviado.writeUTF("O servidor recebeu sua mensagem.");
		} catch (IOException ex) {
			System.err.println("Falha na conexão");
		} finally {
			try {
				soc.close();
				socServidor.close();
			} catch (IOException e) {
				System.err.println("Falha ao encerrar a conexão");
			}
		}
	}
}
```

```java
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.net.Socket;

public class Cliente {

	public static void main(String[] args) {
		Socket soc = null;
		try {
			soc = new Socket("127.0.0.1", 5001);
			DataInputStream recebido = new DataInputStream(soc.getInputStream());
			DataOutputStream enviado = new DataOutputStream(soc.getOutputStream());
			enviado.writeUTF("Aqui é um cliente falando...");
			System.out.println("(servidor): " + recebido.readUTF());
		} catch (IOException ex) {
			System.err.println("Falha na inicializar o servidor");
		} finally {
			try {
				soc.close();
			} catch (IOException e) {
				System.err.println("Falha ao encerrar a conexão");
			}
		}
	}
}
```

Em relação às Classes utilizadas e, analogamente à Figura 4.6, temos a primitiva “socket” implementada pelo objeto “soc” no código, tanto do servidor na linha 10, quanto do cliente na linha 9.

No código do servidor, a primitiva “bind” e “listen” é implementada pelo objeto “socServidor”, na linha 13. A primitiva “accept” é implementada pelométodo de mesmo nome, “accept()”, disponível na Classe “ServerSocket” do Java, na linha 14. No código do cliente, a primitiva “connect” é implementada através da instanciação de um objeto Socket, onde informamos o endereço IP e porta do servidor, na linha 11.

Observe que o endereço IP utilizado no exemplo de código “Cliente.java” (Figura 4.7), na linha 11, “127.0.0.1”, representa a comunicação com a própria máquina (comunicação esta chamada de localhost), o que significa que tanto a aplicação Servidor quanto a aplicação Cliente, neste caso em particular, estão sendo executadas em uma mesma máquina. A porta utilizada no exemplo, “5001” foi escolhida arbitrariamente, como uma das portas disponíveis (não utilizadas por outra aplicação quando da execução desse exemplo).

Tanto no cliente, nas linhas 12 e 13, quanto no servidor, nas linhas 15 e 16, as primitivas “read” e “write” são implementadas pelos objetos “recebido” e “enviado”, respectivamente. Por fim, após a troca de mensagens entre as máquinas, é utilizada a primitiva “close”, implementadas pelo método de mesmo nome, “close()”, disponível nas Classes “Socket” e “ServerSocket” do Java, uma vez que ambas implementam a interface Closeable, do Java, isso ocorre nas linhas 23 e 24 da classe Servidor e na linha 20 da classe Cliente.

A comunicação via sockets é muito comum no uso de sistemas distribuídos. Assim sendo, é importante entender seus conceitos, de forma que sua implementação possa ser realizada sem percalços, independentemente da linguagem de programação utilizada.

Quais são os dois componentes principais, que precisam ser conhecidos, em uma comunicação via sockets?

a) Endereço IP e protocolo.

b) Protocolo e porta de comunicação.

c) Porta de comunicação e endereço MAC.

d) Protocolo TCP ou protocolo UDP.

e) Endereço IP e porta de comunicação.

Muitas aplicações utilizam a comunicação via sockets para realizar troca de informações entre processos de diferentes máquinas, sejam essas máquinas sistemas embarcados (como as diversas placas com microcontroladores) ou entre cliente/servidor.

As Classes e métodos que devem ser utilizados na implementação de uma comunicação via sockets dependem:

I - Da linguagem de programação adotada.

II - Da versão da linguagem de programação.

III - Do tipo de sistema computacional.

IV - Do suporte a sockets estar disponível naquela linguagem de programação.

Assinale a alternativa CORRETA.

a) Apenas a afirmação IV está correta.

b) Apenas as afirmações I, II e IV estão corretas.

c) Apenas a afirmação III está correta.

d) Todas as afirmações estão corretas.

e) Apenas a afirmação I está correta.