INSTITUTO FEDERAL
Paraíba

Campus
Campina Grande

Alunos: José Werlandiê aureliano lacerda

Lucas rocha.

Turma: 2017.1

Curso: Engenharia de Computação.

Professor: Katyusco Santos.

Disciplina: Programação Orientada a Objeto.

Campina Grande PB (10/07/2018)

Índice

Objetivos.

- Entender a classe Socket e a ServerSocket .
- Mostrar o funcionamento de cada método e como utilizá-lo.
- Mostrar exemplos de modo que todos possam fazer implementações simples após a aula.

Introdução.

Com a constante necessidade de trabalhar em ambientes distribuídos surgiram alternativas, em **Java**, para que isso fosse possível. Uma dela e que estudaremos neste resumo é a **Socket**. Há diversos artigos e tutoriais na internet explicando como funciona o Socket e como realizar a conexão Cliente-Servidor, mas nosso objetivo vai mais além, pois queremos mostrar o funcionamento de cada método nessa poderosa classe e como utilizá-lo.

ServerSocket class

ServerSocket é a primeira classe importante, ela é rsponsavel por esperar a conexão do cliente.Esta classe possui um construtor onde passamos a porta que desejamos usar para escutar as conexções.

Listagem 1. Construdor

```
ServerSocket server = new ServerSOcket(3322);
```

Aqui, estamos crinado um objeto ServerSocket passando como parâmetro o argumento 3322 que corresponde ao número da porta que será aguardada uma conexão do cliente.

accept(): O método accept() escuta uma conexão e aceita se alguma for encontrada.

O accept() bloqueia todo o restante até que uma conexão seja feita, ele fica em espera aguardando que alguém conecte. Quando alguma conexão é aceita ele retorna um objeto Socket, que veremos mais à frente.

Listagem 2. accept()

```
ServerSocket server = new ServerSocket(3322);

System.out.println("Servidor iniciado na porta 3322");


Socket cliente = server.accept();


//OUTRAS LÓGICAS AQUI
```

Perceba na Listagem 2 que o trecho comentado “OUTRAS LÓGICAS AQUI” só será executado quando alguma conexão for aceita e o “accept()” for liberado, até então ele não passará desta linha

bind(): Vincula um endereço ao socket.

```
ServerSocket server = new ServerSocket(3322);

server.bind(new InetSocketAddress("192.168.0.1", 0));
```

Porém se você executar o código acima provavelmente terá o seguinte erro:

“java.net.SocketException: Already bound

at java.net.ServerSocket.bind(ServerSocket.java:314)

at java.net.ServerSocket.bind(ServerSocket.java:286)

at br.com.loginremoto.util.Server.main(Server.java:17)”

Isso ocorre porque ao criar o ServerSocket ele automaticamente já é vinculado (bind) ao IP atual, e por isso você não pode refazer o bind. Então fique atento para realizar o bind() apenas quando for necessário.

close(): Um método de extrema importância que fecha a conexão atual. Qualquer cliente que esteja conectado será automaticamente desconectado.

getChannel(): retorna um objeto ServerSocketChannel que corresponde a um canal criado através do método open() da classe ServerSocketChannel. Não entraremos em detalhes do seu uso neste artigo.

getInetAddress(): Retorna o endereço de IP local do servidor.

Listagem 4. Recuperando informação do IP

```
ServerSocket server = new ServerSocket(3322);

InetAddress inet = server.getInetAddress();

System.out.println("HostAddress="+inet.getHostAddress());

System.out.println("HostName="+inet.getHostName());
```

Recuperamos o objeto do tipo InetAddress e acessamos seus diversos métodos, um deles são: getHostAddress() e getHostName().

getLocalPort(): como o próprio nome já sugere ele retorna o número da porta no qual o servidor está escutando uma conexão.

Listagem 5. Recuperando a porta

```
ServerSocket server = new ServerSocket(3322);

System.out.println("Porta = "+server.getLocalPort());
```

Bem simples, veja que na Listagem 5 apenas usamos o getLocalPort() e retornamos no console através do System.out.println o valor da porta.

isClosed(): Verifica se o ServerSocket está fechado, assim podemos evitar erros chamando métodos que não poderíamos se ele estivesse fechado.

isBound(): Lembre que anteriormente nós tentamos realizar um bind, mas obtivemos um erro porque o ServerSocket já tinha uma endereço vinculado, com o isBound() nós evitamos este erro.

Vejamos na Listagem 6.

Listagem 6. isBound()

```
ServerSocket server = new ServerSocket(3322);

if (!server.isBound()){

server.bind(new InetSocketAddress("192.168.0.1", 0));

}
```

toString(): Este método nos retorna um valor formatado representando as informações de endereço IP e porta do ServerSocket.

Agora já sabemos como funcionam os principais e mais utilizados métodos da classe ServerSocket, porém ainda precisamos entender o uso da classe Socket.

Socket class

Ao contrário da classe ServerSocket que funciona como um Servidor escutando o cliente, a classeSocket é o cliente propriamente dito. Muito dos métodos estudados na classe ServerSocket estão presentes na Socket e não repetiremos os mesmos conceitos, tais como: bind, close, getChannel e etc, mas vamos ver alguns um pouco diferente que não abordamos anteriormente.

getInputStream(): É através deste método que o Servidor (ServerSocket) consegue capturar o que o cliente está enviado. Vejamos na Listagem 7.

Listagem 7. Lendo o que o cliente enviou

```
Scanner entrada = new Scanner(cliente.getInputStream());
```

Perceba que capturamos a mensagem do cliente e passamos para um objeto Scanner, depois disso podemos manipular da forma que acharmos necessário as mensagens enviadas.

getOutputStream(): Ao contrário do getInputStream() o getOutputStream() envia dados para o outro lado da comunicação, neste caso o ServerSocket.

Listagem 8. Enviando dados ao servidor

```
PrintStream saida = new PrintStream(cliente.getOutputStream());

Scanner teclado = new Scanner(System.in);

while (teclado.hasNextLine()) {
```

```
    saida.println(teclado.nextLine());
}
```

Na Listagem 8 colocamos o que o usuário está digitando no console na variável “saida” que é do tipo PrintStream, que consequentemente será o que o nosso servidor receberá do outro lado, ou seja, nosso ServerSocket.

Construção da Aplicação

Dado os conceitos que foram apresentados, com o detalhamento e uso dos métodos mais importantes e utilizados com maior frequência, podemos agora começar a desenvolver nossa aplicação para comunicação através do Socket.

Primeiramente vamos construir nossa classe Server com base no que foi ensinado Observe a Listagem 9.

Listagem 9. Server.class

```
package br.com.loginremoto.util;

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.Scanner;
import java.util.logging.Level;
import java.util.logging.Logger;

public class Server {
```

```
    public static void main(String args[]){
        try {
            ServerSocket server = new ServerSocket(3322);
            System.out.println("Servidor iniciado na porta 3322");

            Socket cliente = server.accept();
            System.out.println("Cliente conectado do IP "+cliente.getInetAddress().
                    getHostAddress());
            Scanner entrada = new Scanner(cliente.getInputStream());
            while(entrada.hasNextLine()){
                System.out.println(entrada.nextLine());
            }

            entrada.close();
            server.close();

        } catch (IOException ex) {
            Logger.getLogger(Server.class.getName()).log(Level.SEVERE, null, ex);
        }

    }

}
```

Vamos entender o código acima:

Primeiramente criar um objeto do tipo SocketServer na porta 3322, assim podemos escutar conexões do cliente nesta porta. É muito importante que você escolha uma porta que já não esteja em uso por algum serviço do seu sistema operacional. Por exemplo: Se você usar o tomcat na porta 8080, não poderá utilizar esta porta para testar a sua conexão Socket.

Depois nós chamamos o método “server.accept()” que irá “bloquear” a execução do restante da lógica até que uma conexão seja estabelecida. Se você quiser que vários clientes possam se conectar simultaneamente você poderia envolver o “server.accept()” em um laço “while” para que ele possa aceitar diversas conexões até que determinada condição seja atingida.

Ao usarmos o getInputStream() estamos capturando o que o cliente digitou, e neste ponto uma conexão já foi estabelecida, só precisamos trafegar os dados neste “canal de comunicação”. Nosso objetivo aqui é apenas usar o “while” e mostrar no console do servidor o que está sendo digitado no cliente, mas você poderia fazer vários tipos de tratamentos no lado do servidor com os dados recebidos.

Por fim nós fechamos a conexão do servidor e o Scanner, assim o fluxo de transmissão é interrompido.

Repare que nossa classe acima possui um método “main()” e devemos executá-la para que o Servidor comece a escutar uma conexão do cliente, ou seja, executamos a classe acima e depois executamos o cliente que fará a conexão automática neste servidor.

Nosso cliente será composto por um formulário simples com uma caixa de texto (JtextArea) e um Jbutton, assim poderemos enviar tudo que for digitado na caixa de texto para o console do servidor.

Formulário do Cliente

Figura 1. Formulário do Cliente

Na Figura 1 temos a demonstração de como criamos a interface do nosso usuário. Veja que ela é bem simples e leve, ao digitar qualquer texto basta clicar em Enviar Mensagem que esta chegará no Servidor. Vejamos agora na Listagem 10 como foi feita a construção do nosso formulário e a explicação dos métodos.

Listagem 10. Formulário client

```
package seminario;

import java.io.IOException;
import java.io.PrintStream;
import java.net.Socket;
import java.util.logging.Level;
import java.util.logging.Logger;

public class cliente_1 extends javax.swing.JFrame {

    private Socket cliente;

    public cliente_1() {
```

```
        initComponents();
        initCliente();
    }

    private void initCliente(){
        try {
            cliente = new Socket("192.168.3.100",3322);
        } catch (IOException ex) {
            Logger.getLogger(cliente_1.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
////////////////////////////////////////////////
    @SuppressWarnings("unchecked")
    // <editor-fold defaultstate="collapsed" desc="Generated Code">
    private void initComponents() {

        jScrollPane1 = new javax.swing.JScrollPane();
        jTextArea1 = new javax.swing.JTextArea();
        jButton1 = new javax.swing.JButton();

        setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);

        jTextArea1.setColumns(20);
        jTextArea1.setRows(5);
        jScrollPane1.setViewportView(jTextArea1);
```

```
        jButton1.setText("Enviar Mensagem");
        jButton1.addActionListener(new java.awt.event.ActionListener() {
            @Override
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                jButton1ActionPerformed(evt);
            }
        });

        javax.swing.GroupLayout layout = new javax.swing.GroupLayout(getContentPane());
        getContentPane().setLayout(layout);
        layout.setHorizontalGroup(
            layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
            .addGroup(layout.createSequentialGroup()
                .addContainerGap()

.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
                    .addComponent(jScrollPane1, javax.swing.GroupLayout.DEFAULT_SIZE,
376, Short.MAX_VALUE)
                    .addGroup(layout.createSequentialGroup()
                        .addComponent(jButton1)
                        .addGap(0, 0, Short.MAX_VALUE)))
                .addContainerGap())
        );
        layout.setVerticalGroup(
            layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
            .addGroup(layout.createSequentialGroup()
```

```
                        .addContainerGap()
                        .addComponent(jScrollPane1, javax.swing.GroupLayout.PREFERRED_SIZE,
228, javax.swing.GroupLayout.PREFERRED_SIZE)
                        .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)
                        .addComponent(jButton1)
                        .addContainerGap(25, Short.MAX_VALUE))
            );

            pack();
        }
    ////////////////////////////////////////////////////////////////////////////////
        private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
            try {
                PrintStream saida = new PrintStream(cliente.getOutputStream());
                saida.println(jTextArea1.getText());
                jTextArea1.setText("");
            } catch (IOException ex) {
                Logger.getLogger(cliente_1.class.getName()).log(Level.SEVERE, null, ex);
            }
        }

        public static void main(String args[]) {

            try {
                for (javax.swing.UIManager.LookAndFeelInfo info :
javax.swing.UIManager.getInstalledLookAndFeels()) {
```

```
                if ("Nimbus".equals(info.getName())) {
                    javax.swing.UIManager.setLookAndFeel(info.getClassName());
                    break;
                }
            }
        } catch (ClassNotFoundException ex) {
java.util.logging.Logger.getLogger(cliente_1.class.getName()).log(java.util.logging.Level.SEVERE,
null, ex);
        } catch (InstantiationException ex) {
java.util.logging.Logger.getLogger(cliente_1.class.getName()).log(java.util.logging.Level.SEVERE,
null, ex);
        } catch (IllegalAccessException ex) {
java.util.logging.Logger.getLogger(cliente_1.class.getName()).log(java.util.logging.Level.SEVERE,
null, ex);
        } catch (javax.swing.UnsupportedLookAndFeelException ex) {
java.util.logging.Logger.getLogger(cliente_1.class.getName()).log(java.util.logging.Level.SEVERE,
null, ex);
        }

        /* Create and display the form */
        java.awt.EventQueue.invokeLater(new Runnable() {
            @Override
            public void run() {
                new cliente_1().setVisible(true);
```

```
                }
            });
        }

        // Variables declaration - do not modify
        private javax.swing.JButton jButton1;
        private javax.swing.JScrollPane jScrollPane1;
        private javax.swing.JTextArea jTextArea1;
        // End of variables declaration
}
```

1.Perceba que temos logo no início um atributo “private Socket cliente” que usaremos para conexão cliente-servidor. Este atributo será útil para armazenamos a instância de conexão que foi estabelecida com o servidor e não precisar ficar reconectando a todo instante.

2.No método construtor Fcliente() fazemos chamada a outro método: initCliente().

3.O initCliente() é responsável por inicializar a conexão cliente com o servidor 3322 que criamos anteriormente, o IP 127.0.0.1 indica que o servidor está na mesma máquina que o cliente, ou seja, local. Nada impede que você faça o teste usando um outro computador como servidor ou até mesmo em uma rede externa.

4.O método initComponents() é responsável por inicializar e dispor os componentes no nosso formulário, que é um Jframe.

5.Temos a ação de clique do botão disparando o método jButton1ActionPerformed(), ou seja, quando o usuário digitar seu texto e clicar no botão Enviar Mensagem, esse método será chamado. Neste método temos o uso do getOutputStream() que é responsável por enviar os dados ao servidor, para isso nós capturamos o que o usuário escreveu através do getText() do componente JtextArea e colocando dentro do método println() do objeto PrintStream, assim cada vez que for clicado no botão Enviar Mensagem a mesma aparecerá no console do servidor.

6.Por fim temos o método main() que nos possibilita executar o Formulário em questão e realizar

as operações necessárias.

A nossa limitação é que apenas um cliente por vez pode se conectar ao servidor, isso ocorre porque quando um cliente se conecta ele ocupa a única Thread que o Servidor possui e se outro tentar consequentemente ele não conseguirá. A solução para isso é trabalhar com Multi-threading para aceitar várias conexões simultâneas, mas você deve ficar atento a quantidade máxima de conexões que o servidor pode suportar por isso é importante parametrizar tal recurso. Deixaremos como desafio para que você implemente a solução com vários clientes conectados, a dica é você usar o método accept() dentro de um laço para que várias conexões possam ser aceitas, assim o servidor estará “sempre” esperando uma nova conexão.

Neste testo vimos o uso da classe Socket e a ServerSocket para construção de uma aplicação básica que faz comunicação entre cliente e servidor. A aplicação de tais conceitos é muito útil em diversas situações, por exemplo:

Criar um mecanismo de envio de comandos do cliente para o servidor, onde o servidor irá executar este comando localmente. Ex: Envio um shutdown para o servidor, e este desliga a máquina servidora.

Sistema de chat entre Cliente-Cliente e Cliente-Servidor.A

Sincronismo de informações para sistemas distribuídos, ou seja, sistema que possuem diversos banco de dados podem ser centralizaos através de uma comunicação Socket.

E ainda há muitas outras aplicações no qual o Socket é utilizado.

BIBLIOGRAFICA

Artigo Java Socket: Entendendo a classe socket e ServerSocket.Autor-rolnadoRonaldo Lanhellas(2015). Bacharel em Ciência da Computação (UNAMA). Pós-Graduação em Tecnologias WEB (UFPA).

Material para consulta: Curso Java e Orientação a Objetos >Apostila >Capítulo 19 - Apêndice – Sockets.

Conteúdo extra;

Links para melhor entendimento do assunto.

https://docs.oracle.com/javase/7/docs/api/java/net/ServerSocket.html

https://docs.oracle.com/javase/7/docs/api/java/net/Socket.html

https://www.devmedia.com.br/

https://www.caelum.com.br/apostila-java-orientacao-objetos/apendice-sockets/

Códigos e exemplos:

https://github.com/jose-werlandie/POO-20181/tree/master/Seminario