

[favorito \(7\)](#)[imprimir](#)[anotar](#)[marcar como lido](#)[tirar dúvidas](#)

# Como criar um Chat Multithread com Socket em Java

Veja nesse artigo como criar um chat Multithread com Java. Para isso será necessário usar e aplicar conceitos de Thread e Socket, além da programação Orientado a Objetos.

[Gostei \(5\)](#)[\(0\)](#)

Publicidade

Uma [Thread](#) pode ser considerada um fluxo de controle sequencial dentro de um programa, onde damos algum job e ela o realiza, provendo maior performance. Em programação é muito importante saber aplicar formas de [processamento assíncrono](#), pois atualmente temos processadores altamente velozes e não sabemos explorá-los devidamente. No nosso chat que criaremos nesse artigo, a Thread será usada para controlar o fluxo de mensagens enviadas e recebidas por um cliente, pois imagina se todos elas fossem armazenadas numa fila e processadas unicamente por uma thread: o serviço seria precário e provavelmente ninguém usaria.

## O que é Socket

[Socket](#) é um meio de comunicação usado para viabilizar a conexão cliente/servidor, onde um cliente informa o endereço de IP e a respectiva porta do servidor. Se este aceitar a conexão, ele irá criar um meio de comunicação com esse cliente. Logo, a combinação de Threads e Socket é perfeita para implementação de um chat.

## OOP

Frente a quantidade de linguagens orientadas a objetos (OO) existentes, como C#, C++, Java, entre outras, fica evidente que para dominá-las é necessário entender bem

os [pilares OO](#), como Herança (por interface e por classe), Encapsulamento, Polimorfismo e Abstração. Se dominar bem esses assuntos, com certeza terá mais facilidade em construir códigos simples e com qualidade, tendo baixo acoplamento e alta coesão. Neste artigo nos depararemos com Herança por interface e classe base, abstração e encapsulamento, mas não será o foco do mesmo abordar seus conceitos.

A seguir são descritas as responsabilidades e comportamentos das classes `Server.java` e `Cliente.java` usadas para a construção do Chat:

- **Responsabilidade e comportamentos do `Server.java`:** o servidor servirá como unidade centralizadora de todas as conexões recebidas via socket e terá como responsabilidade o envio de uma mensagem (recebida de um cliente) para todos os demais conectados no servidor. Quando um cliente se conecta a ele o mesmo cria uma Thread para aquele cliente, ou seja, cada conexão terá sua respectiva Thread e o servidor fará a gestão disso;
- **Responsabilidade e comportamentos do `Client.java`:** Cada usuário criará uma instância do cliente e fará uma conexão com o servidor socket. O cliente deverá informar o endereço do server socket e a respectiva porta, por isso é necessário executar o `Server.java` antes.

**Lembre-se:** escolha uma porta que não esteja sendo usada para a execução do server socket e certifique-se que o firewall ou algum antivírus não esteja bloqueando a porta escolhida. Para esse artigo definimos como 12345.

Na **Listagem 1** temos a declaração dos pacotes usados na classe `servidor.java`. Veja que usamos “*streams*”, “*collections*” e classes para a construção de formulários.

#### Listagem 1. Declaração dos imports

```
import java.io.BufferedReader;
```

```
import java.io.BufferedWriter;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.Writer;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.ArrayList;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import javax.swing.JTextField;
```

A seguir temos a declaração da classe `servidor.java`. Veja que ela `extends Thread`, logo é um tipo de `Thread`, adotando todos os comportamentos e propriedades desta classe

```
public class Servidor extends Thread {
```

Na **Listagem 2** temos a declaração dos atributos estáticos e de instâncias da classe `servidor.java`. O atributo “**clientes**” é usado para armazenar o `BufferedWriter` de cada cliente conectado e o `server socket` é usado para a criação do servidor, que teoricamente deve ser feita apenas uma vez.

## Listagem 2. Atributos estáticos

```
private static ArrayList<BufferedWriter>clientes;
private static ServerSocket server;
private String nome;
private Socket con;
private InputStream in;
private InputStreamReader inr;
private BufferedReader bfr;
```

Na **Listagem 3** temos a declaração do método construtor, que recebe um objeto socket como parâmetro e cria um objeto do tipo `BufferedReader`, que aponta para o

[Baixe o APP](#)[Login](#)

### Listagem 3. Declaração do método construtor

```
/**
 * Método construtor
 * @param con do tipo Socket
 */
public Servidor(Socket con){
    this.con = con;
    try {
        in  = con.getInputStream();
        inr = new InputStreamReader(in);
        bfr = new BufferedReader(inr);
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

A **Listagem 4** mostra a declaração do método “*run*”: toda vez que um cliente novo chega ao servidor, esse método é acionado e alocado numa `Thread` e também fica verificando se existe alguma mensagem nova. Caso exista, esta será lida e o evento “*sentToAll*” será acionado para enviar a mensagem para os demais usuários conectados no chat.

### Listagem 4. Declaração do método run

```
/**
 * Método run
 */
public void run(){

    try{

        String msg;
```

```
OutputStream ou = this.con.getOutputStream();
Writer ouw = new OutputStreamWriter(ou);
BufferedWriter bfw = new BufferedWriter(ouw);
clientes.add(bfw);
nome = msg = bfr.readLine();

while(!"Sair".equalsIgnoreCase(msg) && msg != null)
{
    msg = bfr.readLine();
    sendToAll(bfw, msg);
    System.out.println(msg);
}

}catch (Exception e) {
    e.printStackTrace();
}
}
```

Na **Listagem 5** temos a declaração do método “*sendToAll*”. Quando um cliente envia uma mensagem, o servidor recebe e manda esta para todos os outros clientes conectados. Veja que para isso é necessário percorrer a lista de clientes e mandar uma cópia da mensagem para cada um.

### Listagem 5. Declaração do método sendToAll

```
/**
 * Método usado para enviar mensagem para todos os clients
 * @param bwSaida do tipo BufferedWriter
 * @param msg do tipo String
 * @throws IOException
 */
public void sendToAll(BufferedWriter bwSaida, String msg) throws IOException
{
    BufferedWriter bwS;

    for(BufferedWriter bw : clientes){
        bwS = (BufferedWriter)bw;
        if(!(bwSaida == bwS)){
            bw.write(nome + " -> " + msg+"\r\n");
            bw.flush();
        }
    }
}
```

```
}  
}  
}
```

A **Listagem 6** mostra a declaração do método main, que ao iniciar o servidor, fará a configuração do servidor socket e sua respectiva porta. Veja que ele começa criando uma janela para informar a porta e depois entra no “*while(true)*”. Na linha “*server.accept()*” o sistema fica bloqueado até que um cliente socket se conecte: se ele fizer isso é criada uma nova Thread do tipo servidor.

Lembre-se que a classe servidor é um tipo de Thread e é iniciada na instrução “*t.start()*”. Então o controle do fluxo retorna para a linha “*server.accept()*” e aguarda outro cliente se conectar.

#### Listagem 6. Declaração do método main

```
/**  
 * Método main  
 * @param args  
 */  
public static void main(String []args) {  
  
    try{  
        //Cria os objetos necessário para instanciar o servidor  
        JLabel lblMessage = new JLabel("Porta do Servidor:");  
        JTextField txtPorta = new JTextField("12345");  
        Object[] texts = {lblMessage, txtPorta };  
        JOptionPane.showMessageDialog(null, texts);  
        server = new ServerSocket(Integer.parseInt(txtPorta.getText()));  
        clientes = new ArrayList<BufferedWriter>();  
        JOptionPane.showMessageDialog(null,"Servidor ativo na porta: "+  
            txtPorta.getText());  
  
        while(true){  
            System.out.println("Aguardando conexão...");  
            Socket con = server.accept();  
            System.out.println("Cliente conectado...");  
            Thread t = new Servidor(con);
```

```
        t.start();
    }

    }catch (Exception e) {

        e.printStackTrace();
    }
} // Fim do método main
} // Fim da classe
```

A **Listagem 7** mostra a declaração dos pacotes usados na classe cliente.java.

### Listagem 7. Declaração dos Imports

```
import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.Writer;
import java.net.Socket;
import javax.swing.*;
```

A seguir temos a declaração da classe Cliente.java:

```
public class Cliente extends JFrame implements ActionListener, KeyListener {
```

Para a construção do formulário foram usados objetos do pacote javax.swing. A

**Listagem 8** mostra a declaração dos atributos estáticos e de instâncias da classe



Cliente.java.

### Listagem 8. Declaração dos atributos

```
private static final long serialVersionUID = 1L;
private JTextArea texto;
private JTextField txtMsg;
private JButton btnSend;
private JButton btnSair;
private JLabel lblHistorico;
private JLabel lblMsg;
private JPanel pnlContent;
private Socket socket;
private OutputStream ou ;
private Writer ouw;
private BufferedWriter bfw;
private JTextField txtIP;
private JTextField txtPorta;
private JTextField txtNome;
```

Ao executar a classe cliente aparecerá uma tela para o usuário informar alguns parâmetros como o IP do servidor, a porta e o nome que será visto para os demais usuários no chat. No código está definido como padrão o IP 127.0.0.1, porta 12345 e nome cliente.

Observe também que a classe herda de JFrame, possibilitando a criação de formulários e implementação das interfaces ActionListener e KeyListener para prover ações nos botões e ações das teclas, respectivamente.

A **Listagem 9** mostra a declaração do método construtor, que verifica os objetos sendo instanciados para a construção da tela do chat. Lembre-se que cada cliente deverá ser uma instância independente.

### Listagem 9. Declaração do método construtor

```
public Cliente() throws IOException{
    JLabel lblMessage = new JLabel("Verificar!");
    txtIP = new JTextField("127.0.0.1");
    txtPorta = new JTextField("12345");
    txtNome = new JTextField("Cliente");
    Object[] texts = {lblMessage, txtIP, txtPorta, txtNome };
    JOptionPane.showMessageDialog(null, texts);
    pnlContent = new JPanel();
    texto
        = new JTextArea(10,20);
    texto.setEditable(false);
    texto.setBackground(new Color(240,240,240));
    txtMsg
        = new JTextField(20);
    lblHistorico
        = new JLabel("Histórico");
    lblMsg
        = new JLabel("Mensagem");
    btnSend
        = new JButton("Enviar");
    btnSend.setToolTipText("Enviar Mensagem");
    btnSair
        = new JButton("Sair");
    btnSair.setToolTipText("Sair do Chat");
    btnSend.addActionListener(this);
    btnSair.addActionListener(this);
    btnSend.addKeyListener(this);
    txtMsg.addKeyListener(this);
    JScrollPane scroll = new JScrollPane(texto);
    texto.setLineWrap(true);
    pnlContent.add(lblHistorico);
    pnlContent.add(scroll);
    pnlContent.add(lblMsg);
    pnlContent.add(txtMsg);
    pnlContent.add(btnSair);
    pnlContent.add(btnSend);
    pnlContent.setBackground(Color.LIGHT_GRAY);
    texto.setBorder(BorderFactory.createEtchedBorder(Color.BLUE,Color.BLUE));
    txtMsg.setBorder(BorderFactory.createEtchedBorder(Color.BLUE, Color.BLUE));
    setTitle(txtNome.getText());
    setContentPane(pnlContent);
    setLocationRelativeTo(null);
    setResizable(false);
    setSize(250,300);
    setVisible(true);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
}
```

O método da **Listagem 10** é usado para conectar o cliente com o servidor socket.

Nesse método é possível visualizar a criação do socket cliente e dos streams de comunicação.

### Listagem 10. Declaração do método conectar

```
/**
 * Método usado para conectar no server socket, retorna IO Exception caso dê algum
 * @throws IOException
 */
public void conectar() throws IOException{

    socket = new Socket(txtIP.getText(),Integer.parseInt(txtPorta.getText()));
    ou = socket.getOutputStream();
    ouw = new OutputStreamWriter(ou);
    bfw = new BufferedWriter(ouw);
    bfw.write(txtNome.getText()+"\r\n");
    bfw.flush();
}
```

A **Listagem 11** tem o método usado para enviar mensagens do cliente para o servidor socket. Assim, toda vez que ele escrever uma mensagem e apertar o botão “Enter”, esta será enviada para o servidor.

### Listagem 11. Declaração do método enviar mensagem

```
/**
 * Método usado para enviar mensagem para o server socket
 * @param msg do tipo String
 * @throws IOException retorna IO Exception caso dê algum erro.
 */
public void enviarMensagem(String msg) throws IOException{

    if(msg.equals("Sair")){
        bfw.write("Desconectado \r\n");
        texto.append("Desconectado \r\n");
    }else{
        bfw.write(msg+"\r\n");
        texto.append( txtNome.getText() + " diz -> " + txtMsg.getText()+"\r\n"
```

```
}  
    bfw.flush();  
    txtMsg.setText("");  
}
```

Na **Listagem 12** temos o método usado para escutar (receber) mensagens do servidor. Toda vez que alguém enviar uma, o método será processado pelo servidor e envia para todos os clientes conectados, por isso a necessidade do código.

### Listagem 12. Declaração do método escutar

```
/**  
 * Método usado para receber mensagem do servidor  
 * @throws IOException retorna IO Exception caso dê algum erro.  
 */  
public void escutar() throws IOException{  
  
    InputStream in = socket.getInputStream();  
    InputStreamReader inr = new InputStreamReader(in);  
    BufferedReader bfr = new BufferedReader(inr);  
    String msg = "";  
  
    while(!"Sair".equalsIgnoreCase(msg))  
  
        if(bfr.ready()){  
            msg = bfr.readLine();  
            if(msg.equals("Sair"))  
                texto.append("Servidor caiu! \r\n");  
            else  
                texto.append(msg+"\r\n");  
        }  
}
```

O método da **Listagem 13** é usado para desconectar do server socket. Nele o sistema apenas fecha os streams de comunicação.

### Listagem 13. Declaração do método sair

```
/**
 * Método usado quando o usuário clica em sair
 * @throws IOException retorna IO Exception caso dê algum erro.
 */
public void sair() throws IOException{

    enviarMensagem("Sair");
    bfw.close();
    ouw.close();
    ou.close();
    socket.close();
}
```

O método usado para receber as ações dos botões dos usuários é visto na **Listagem 14**. Nele foi feito um chaveamento: se o usuário pressionar o botão “send” então será enviada uma mensagem, senão será encerrado o chat.

#### Listagem 14. Declaração do método actionPerformed

```
@Override
public void actionPerformed(ActionEvent e) {

    try {
        if(e.getActionCommand().equals(btnSend.getActionCommand()))
            enviarMensagem(txtMsg.getText());
        else
            if(e.getActionCommand().equals(btnSair.getActionCommand()))
                sair();
    } catch (IOException e1) {
        // TODO Auto-generated catch block
        e1.printStackTrace();
    }
}
```

O método da **Listagem 15** é acionado quando o usuário pressiona “Enter”, verificando se o key code é o Enter. Caso seja, a mensagem é enviada para o servidor.

## Listagem 15. Declaração do método keyPressed

```
@Override
public void keyPressed(KeyEvent e) {

    if(e.getKeyCode() == KeyEvent.VK_ENTER){
        try {
            enviarMensagem(txtMsg.getText());
        } catch (IOException e1) {
            // TODO Auto-generated catch block
            e1.printStackTrace();
        }
    }
}

@Override
public void keyReleased(KeyEvent arg0) {
    // TODO Auto-generated method stub
}

@Override
public void keyTyped(KeyEvent arg0) {
    // TODO Auto-generated method stub
}
```

A **Listagem 16** mostra o método main, onde é criado apenas um cliente e são configurados os métodos conectar e escutar.

## Listagem 16. Declaração do método main

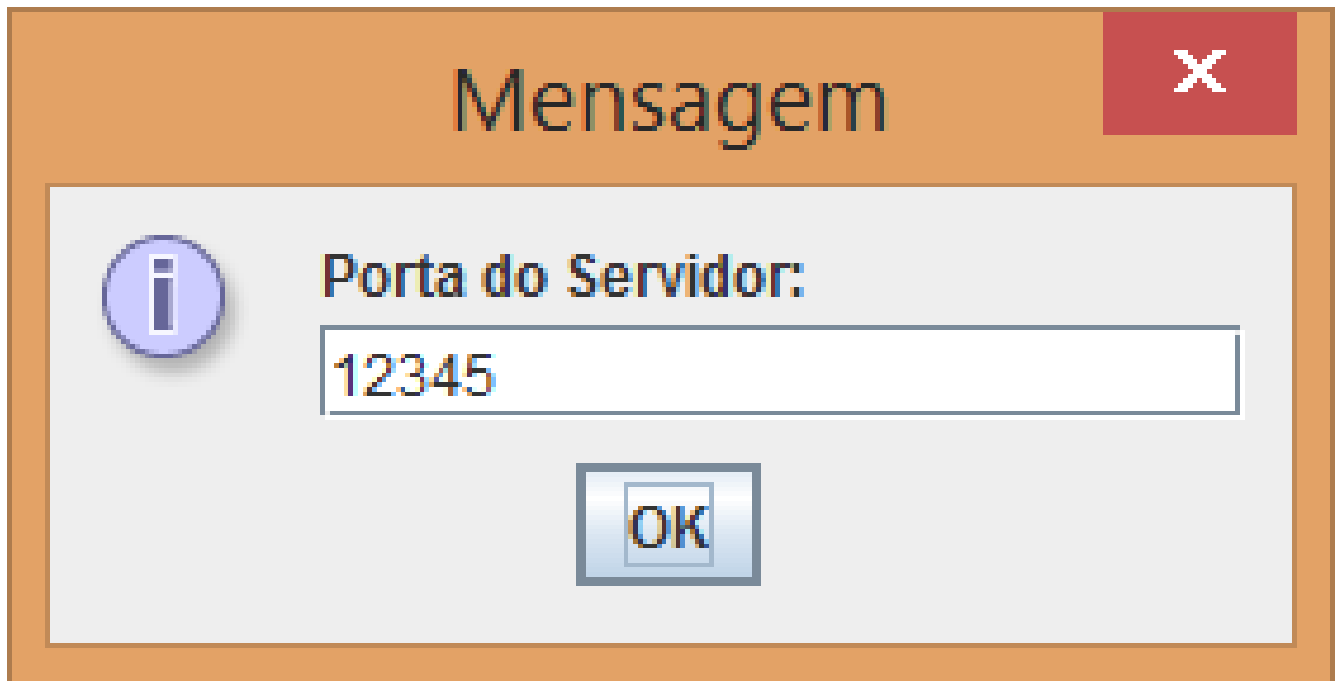
```
public static void main(String []args) throws IOException{

    Cliente app = new Cliente();
    app.conectar();
    app.escutar();
}
```

Depois do código implementado na IDE *Eclipse* ou *Netbeans*, temos duas classes: a

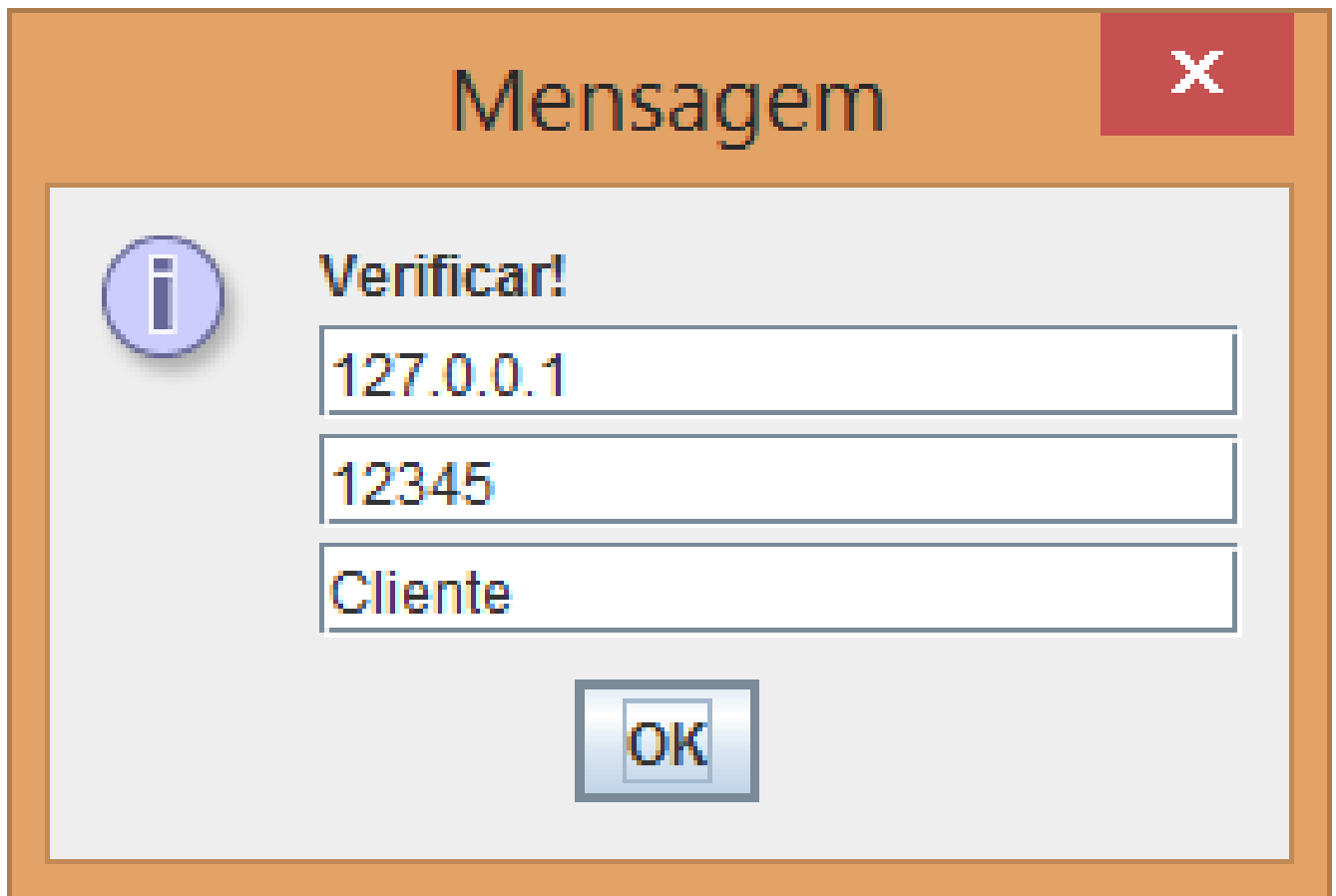
Servidor.java e a Cliente.java. Execute uma vez a classe Servidor.java e a classe Cliente.java quantas vezes achar necessário, porém, se executar apenas uma vez, você não verá sua mensagem sendo enviada para ninguém.

Quando executar o servidor pela primeira vez aparecerá uma caixa para informar os parâmetros necessários. Informe o número da porta, conforme mostra a **Figura 1**.

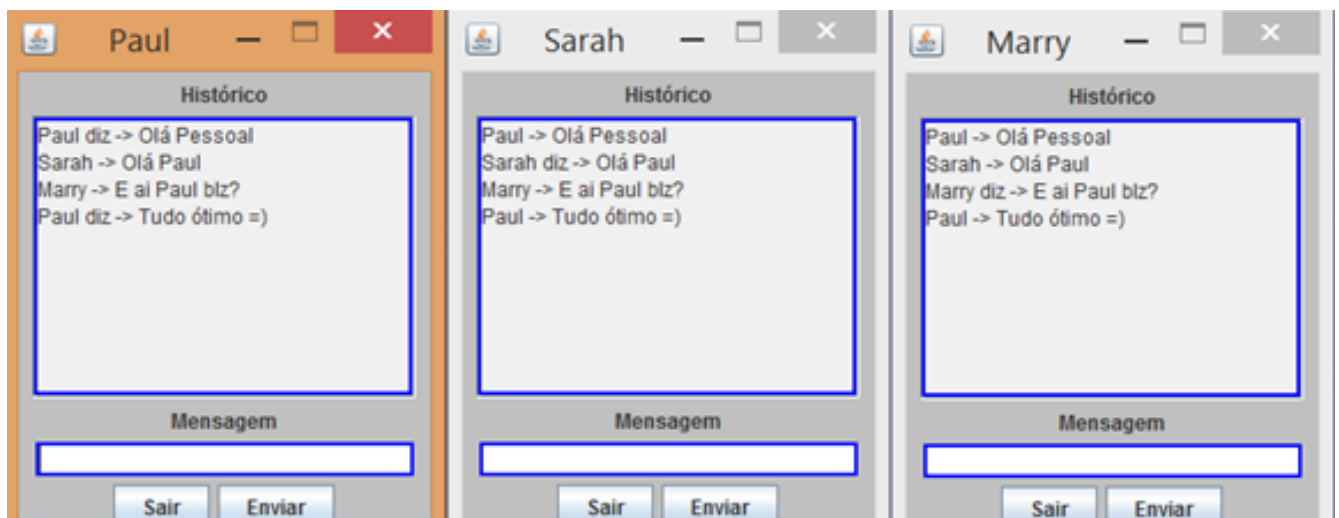


**Figura 1.** Input para informar o número da porta onde o servidor socket receberá as conexões.

Ao executar um cliente também será necessário informar alguns parâmetros como a porta e o endereço de IP do servidor socket, como vemos na **Figuras 2**.



**Figura 2.** Inputs do IP do servidor socket, a porta e o nome que será visto por você e pelos outros usuários.




**Figura 3.** Chat em execução.

Após executar o servidor e dois ou mais clientes será possível conversar com todos eles. Repare na primeira tela da **Figura 3**, em que o Paul começa a conversa.



Vimos como é simples a construção básica de um chat, desde de que a ideia de Thread e Socket seja bem compreendida.

Espero que tenham gostado!

 Publicado no [Canal Java](#)



por Leonardo Rocha

Expert em Java e programação Web

Gostou?



Sim (5)



(0)

Ficou com alguma dúvida?

Não há comentários

[Postar dúvida / Comentário](#)

[Meus comentarios](#)



[Anuncie](#) | [Loja](#) | [Publique](#) | [Assine](#) | [Fale conosco](#)



Hospedagem web por **Porta 80 Web Hosting**