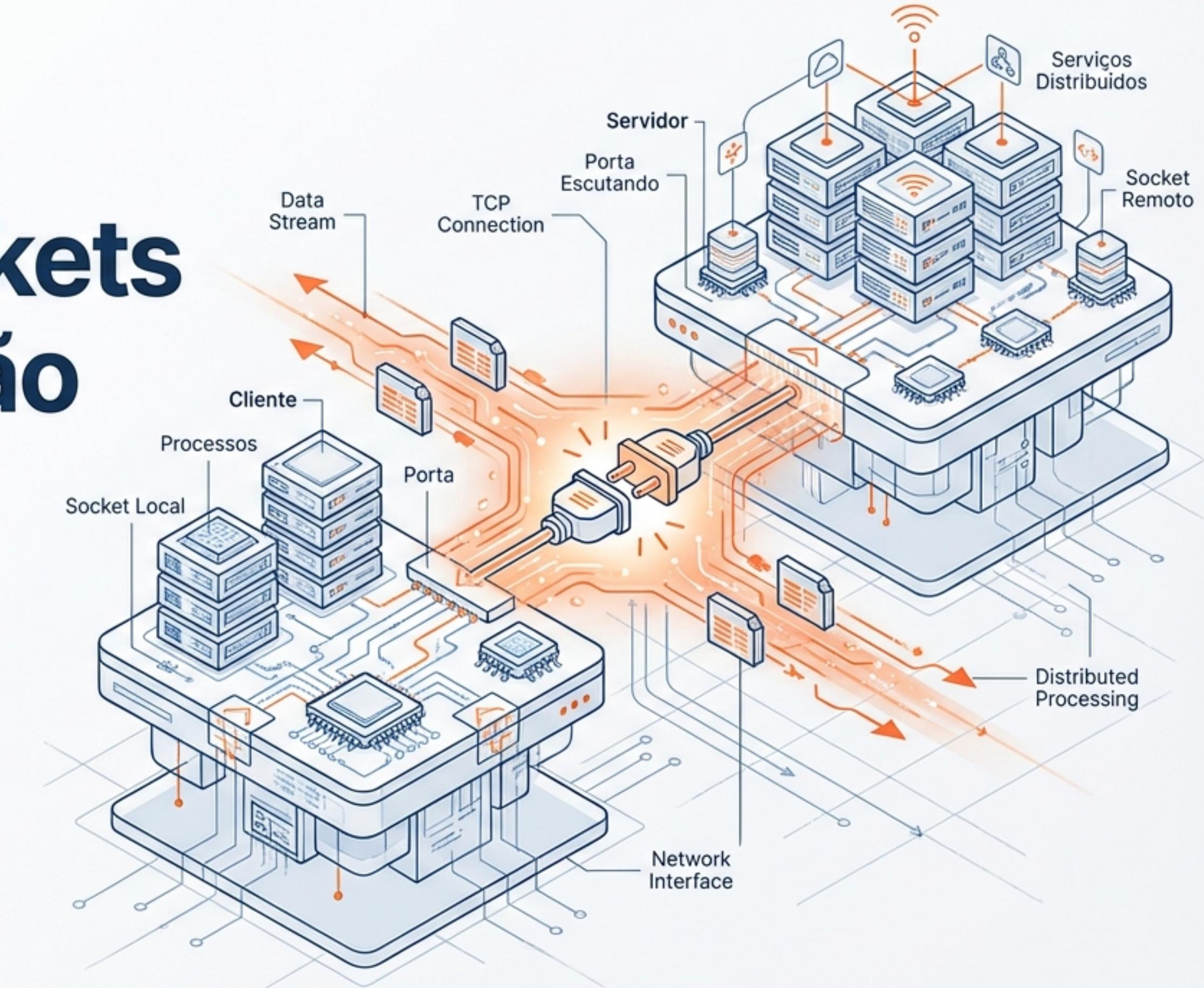


# Conectando Mundos: Sockets e Programação Distribuída em Java

Uma Jornada Arquitetural  
do TCP Básico ao  
Processamento Distribuído

Baseado no material do Prof. Rafael Vargas Mesquita | IFES



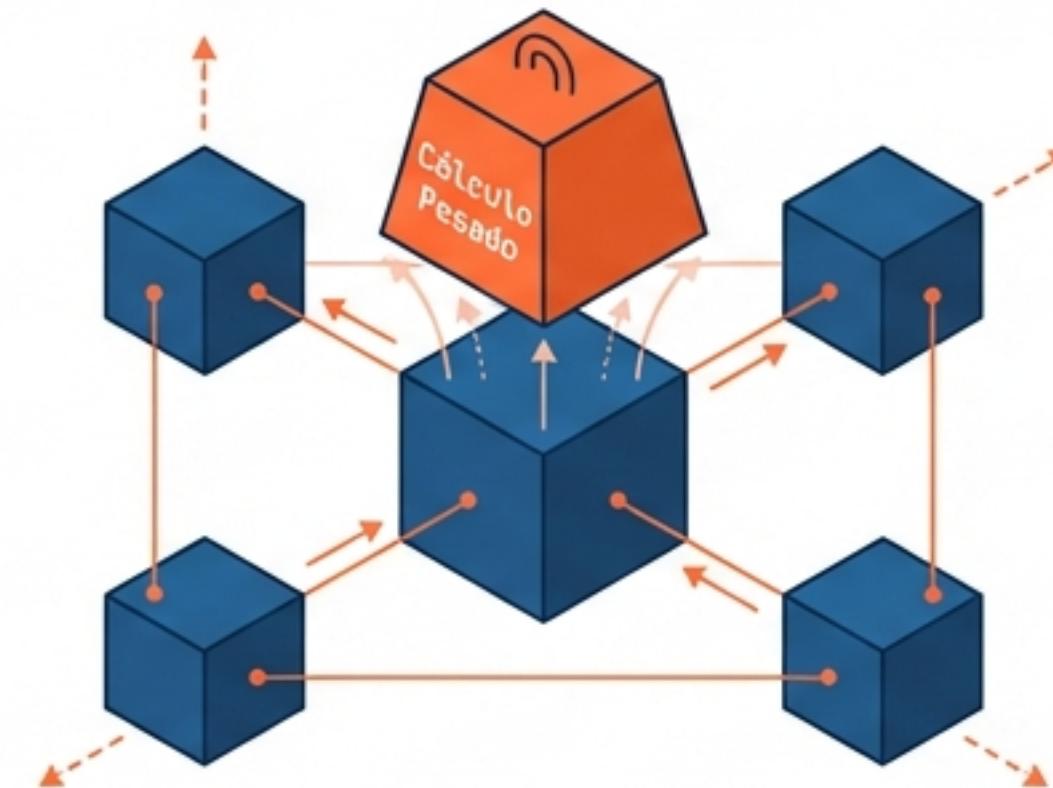
# O Desafio: Por que dividir para conquistar?

## O Problema (Monolítico)



- **Aplicações Isoladas:** Todo processamento em uma única máquina.
- **Gargalo:** Algoritmos pesados travam o sistema e limitam a escalabilidade.

## A Solução (Distribuída)

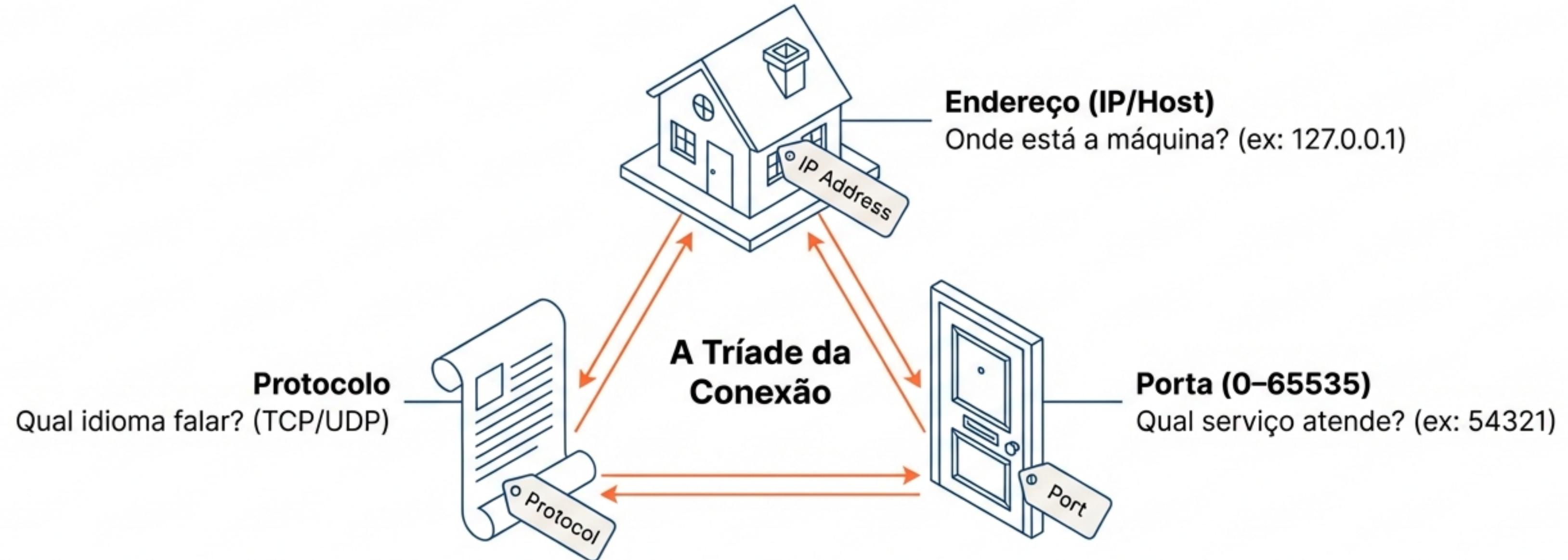


- **Processamento Compartilhado:** O software roda em processos ou máquinas diferentes.
- **Exemplo Prático:** Uma máquina 'Mestre' delega intervalos para várias 'Trabalhadoras'.

"Programação distribuída trata da construção de sistemas cujas partes executam em processos diferentes e precisam se comunicar." (Fonte: Deitel & Deitel)

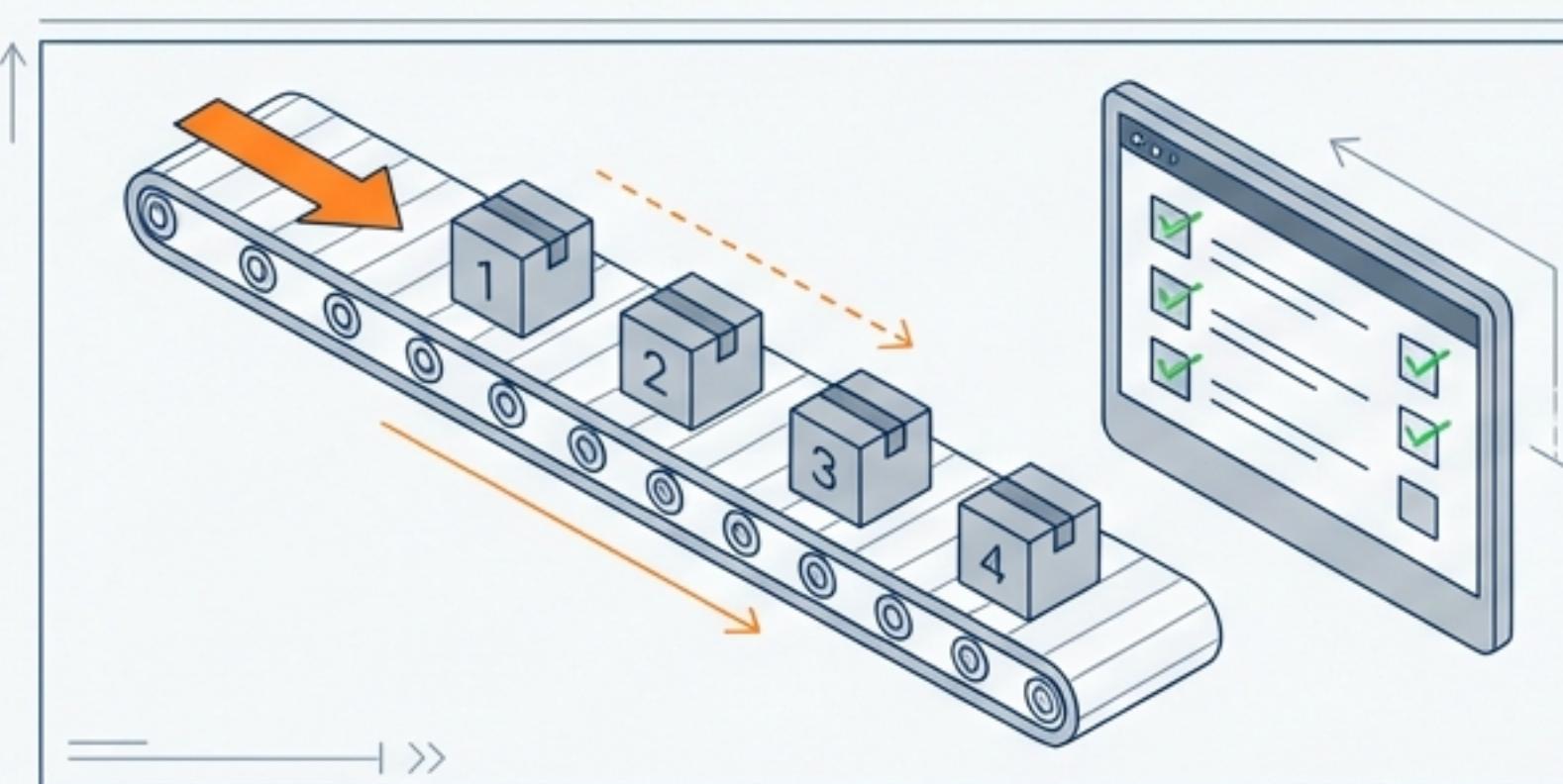
# O Socket: O Ponto de Encontro Digital

**Definição:** Um socket é um ponto final de comunicação entre dois processos, permitindo fluxo bidirecional de dados.



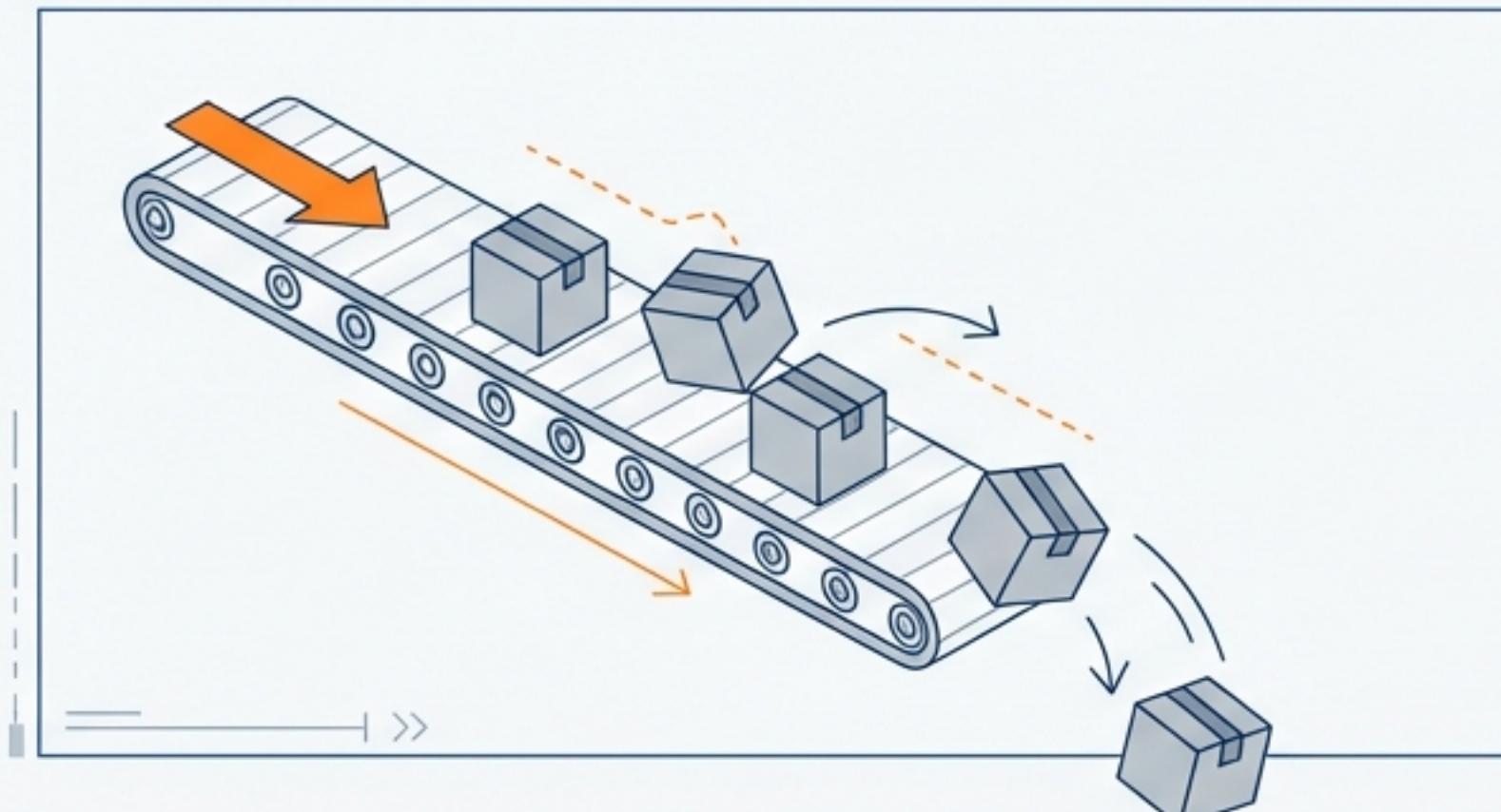
```
// Descobrindo quem sou eu na rede
InetAddress local = InetAddress.getLocalHost();
System.out.println("IP: " + local.getHostAddress());
```

# As Regras da Estrada: TCP vs. UDP



## TCP (Transmission Control Protocol)

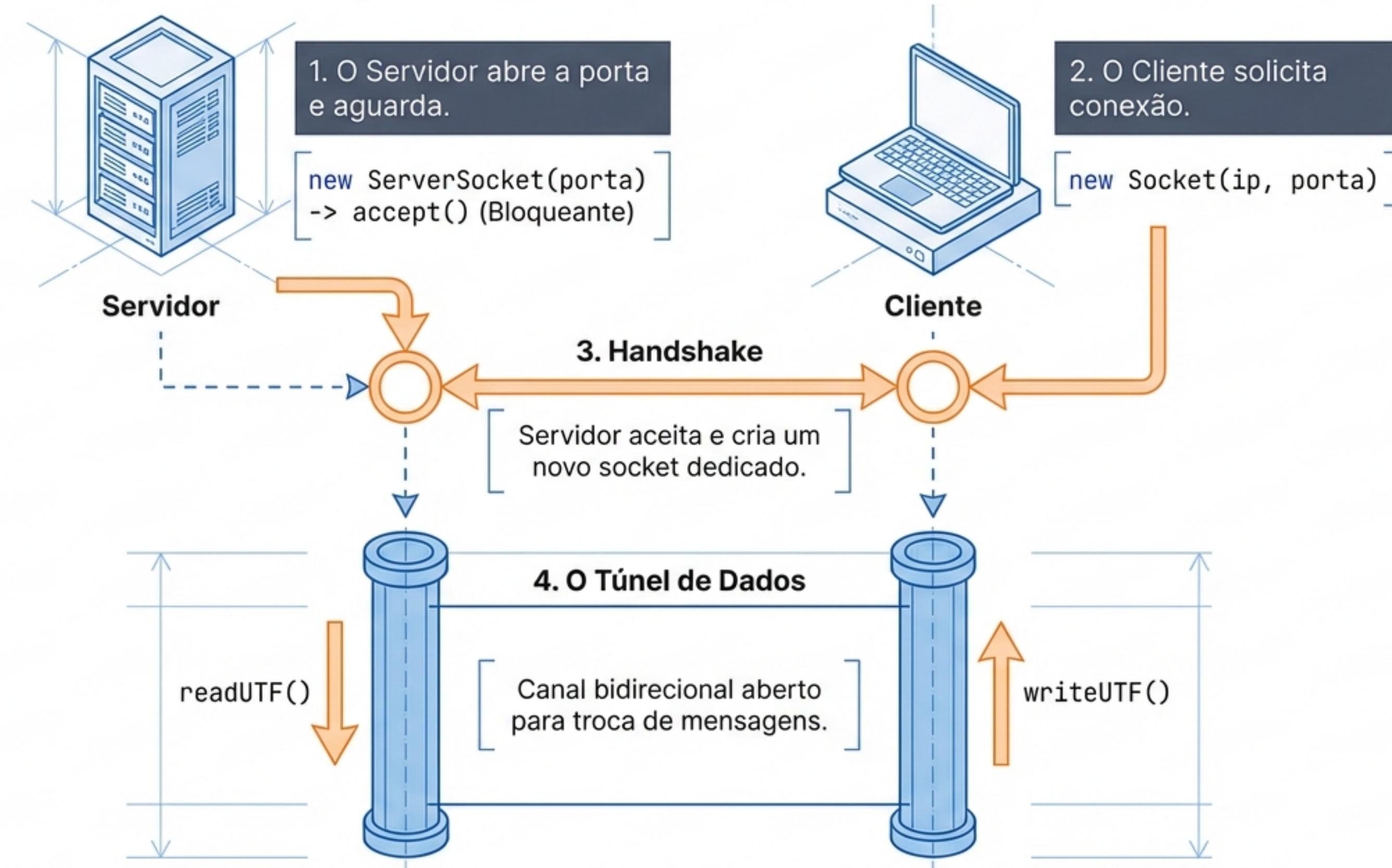
- Metáfora: Uma ligação telefônica.
- Características: Conexão garantida, entrega ordenada.
- Uso: Downloads, Chat, E-mail.
- STATUS: Foco desta apresentação.



## UDP (User Datagram Protocol)

- Metáfora: Uma carta no correio (ou jogar tijolos por cima de um muro).
- Características: Sem conexão, rápido, sem garantia de entrega.
- Uso: Streaming de vídeo, Jogos FPS, Telemetria.

# A Dança da Conexão (Modelo Cliente-Servidor TCP)

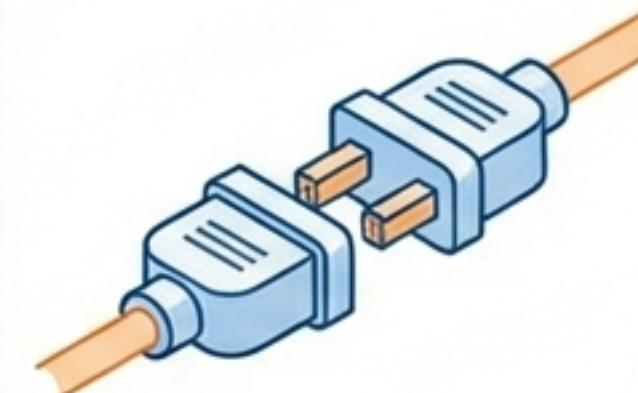


# A Caixa de Ferramentas Java (java.net & java.io)



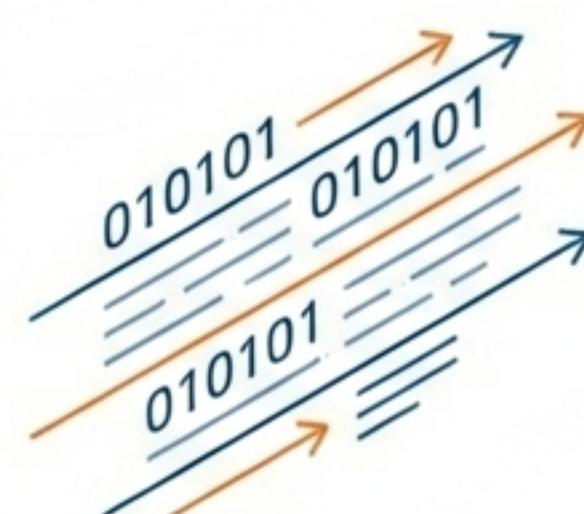
## ServerSocket

Exclusivo do Servidor.  
Aguarda e aceita  
conexões iniciais.



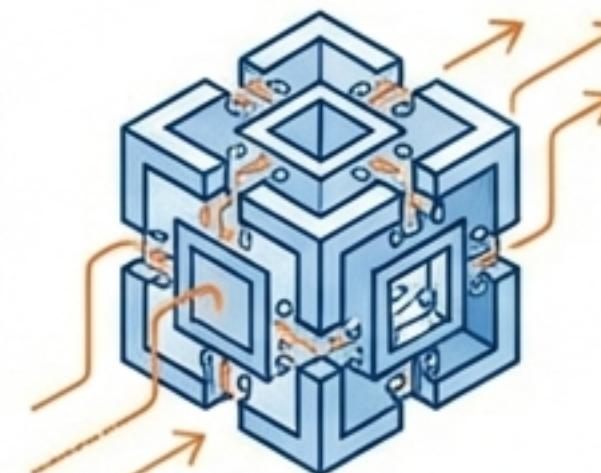
## Socket

Usado por ambos.  
Representa a conexão  
ativa e o canal de dados.



## DataInputStream / DataOutputStream

Para dados primitivos (int,  
boolean, UTF Strings).  
*\*Usado no Nível 1.\**



## ObjectInputStream / ObjectOutputStream

Para objetos Java  
complexos (Serialização).  
*\*Usado no Nível 3.\**



**Dica:** Os streams funcionam como canos cruzados. O OutputStream de um conecta-se ao InputStream do outro.

# Nível 1: O Echo Simples (Texto)

**Problema:** Enviar "rafael" e receber "RAFAEL".



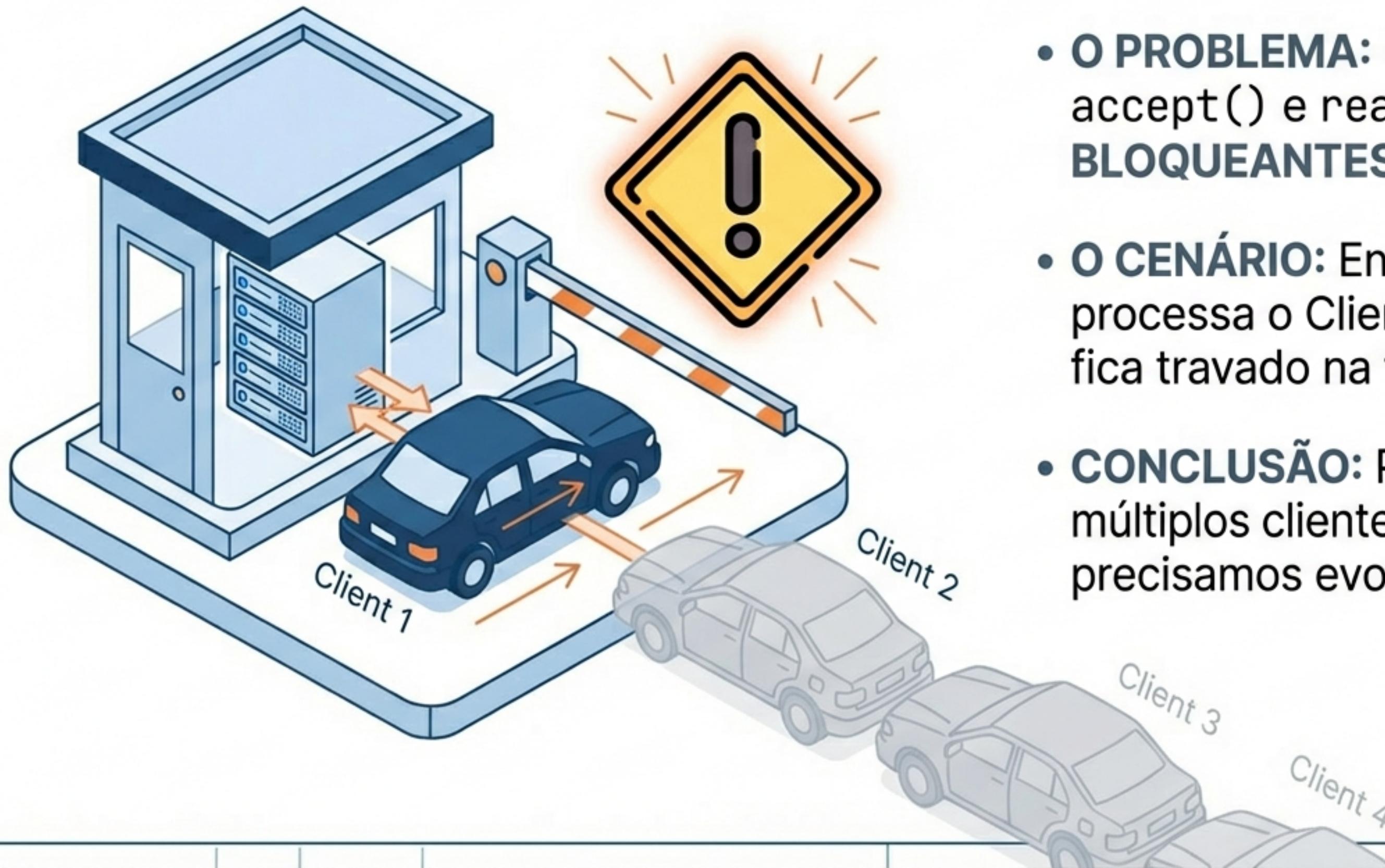
## Cliente

```
// Cliente  
Socket socket = new Socket("127.0.0.1", 54321);  
DataOutputStream saida = new  
    DataOutputStream(socket.getOutputStream());  
saida.writeUTF("rafael"); // Envia  
DataInputStream entrada = new  
    DataInputStream(socket.getInputStream());  
String resposta = entrada.readUTF(); // Aguarda
```

## Servidor

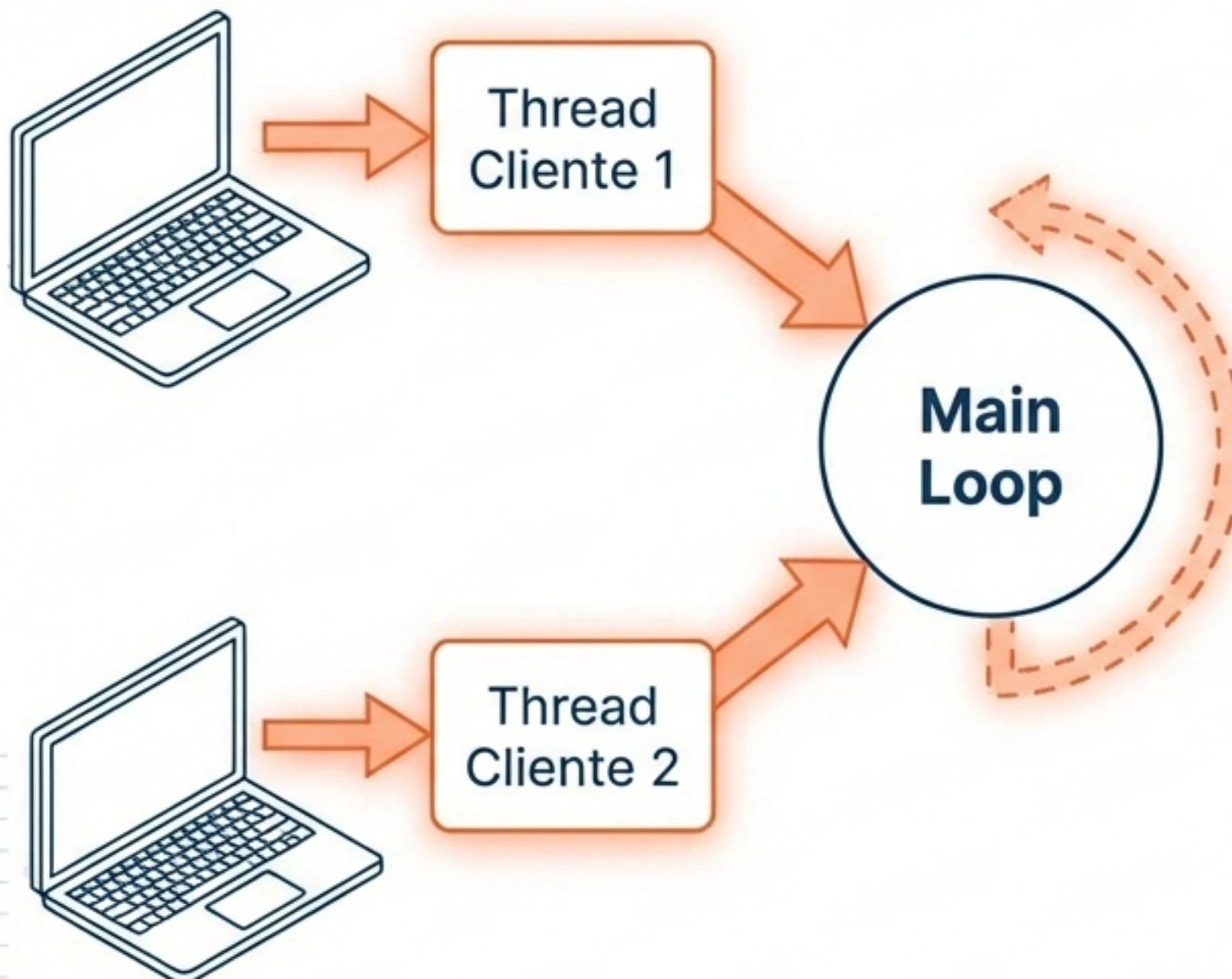
```
// Servidor  
ServerSocket server = new ServerSocket(54321);  
Socket socket = server.accept(); // Bloqueia  
DataInputStream entrada = new  
    DataInputStream(socket.getInputStream());  
String recebido = entrada.readUTF();  
DataOutputStream saida = new  
    DataOutputStream(socket.getOutputStream());  
saida.writeUTF(recebido.toUpperCase()); // Devolve
```

# O Gargalo: O Problema do filo Bloqueio



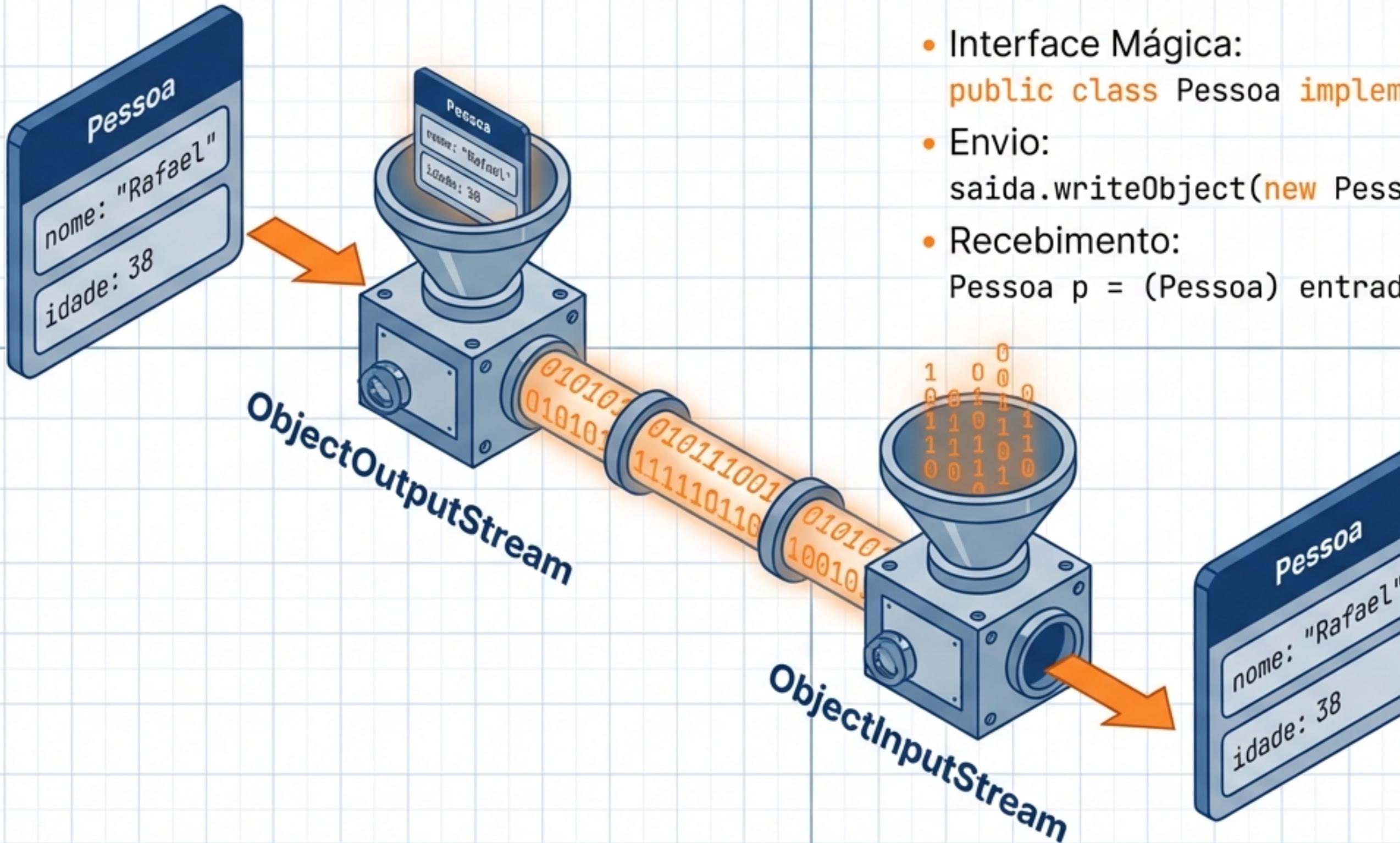
- **O PROBLEMA:** Métodos como `accept()` e `readUTF()` são **BLOQUEANTES**.
- **O CENÁRIO:** Enquanto o servidor processa o Cliente 1, o Cliente 2 fica travado na fila de conexão.
- **CONCLUSÃO:** Para atender múltiplos clientes simultaneamente, precisamos evoluir a arquitetura.

# Nível 2: Escalando com Multithreading



```
while(true) {  
    // 1. Aceita a conexão  
    Socket cliente = server.accept();  
  
    // 2. Cria uma "filial" (Thread) para este cliente  
    ThreadSockets thread = new ThreadSockets(cliente);  
    thread.start();  
  
    // 3. Loop volta imediatamente para o topo  
}
```

# Nível 3: Trafegando Objetos Complexos



- Interface Mágica:

```
public class Pessoa implements Serializable
```

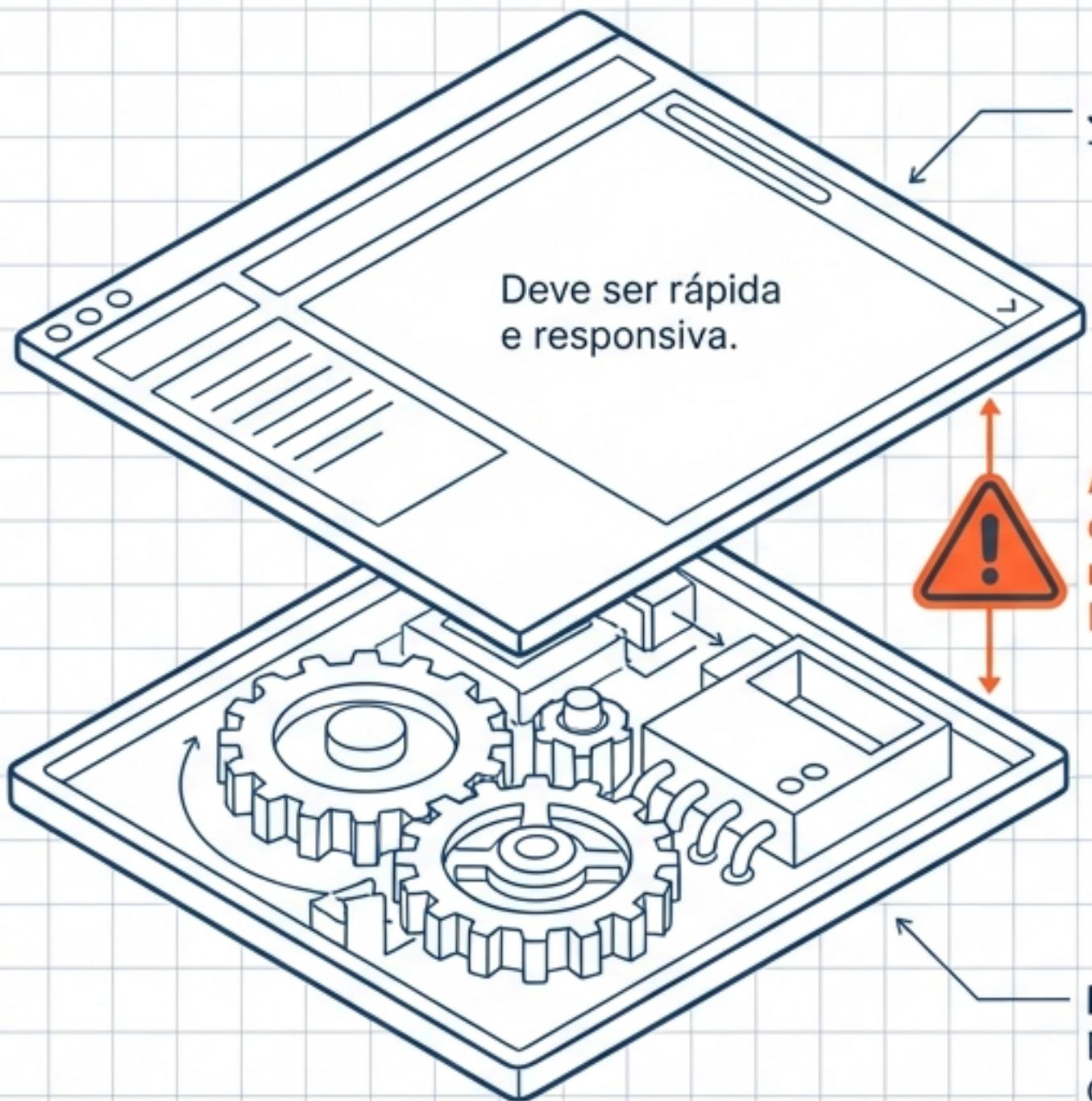
- Envio:

```
saida.writeObject(new Pessoa(...));
```

- Re却imento:

```
Pessoa p = (Pessoa) entrada.readObject();
```

# O Desafio da Interface Gráfica (JavaFX)



**JavaFX Application Thread (UI)**  
Technical Slate (#4A5568)

**A REGRA DE OURO:** Nunca execute operações de rede na **thread da interface**. Isso congela a tela!

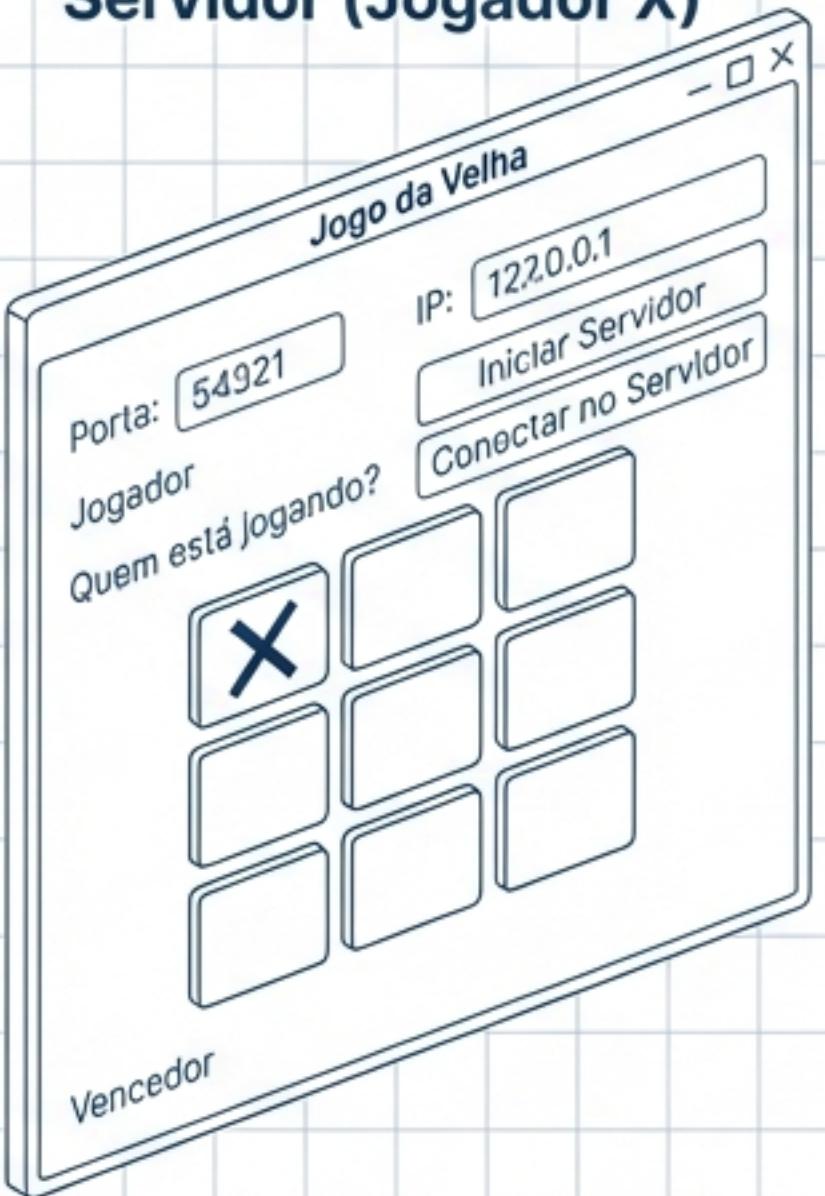
**Background Thread**  
Rede, Sockets,  
Operações lentas.

```
new Thread(() -> {
    // Background: Espera a mensagem (Bloqueante)
    String msg = entrada.readUTF();

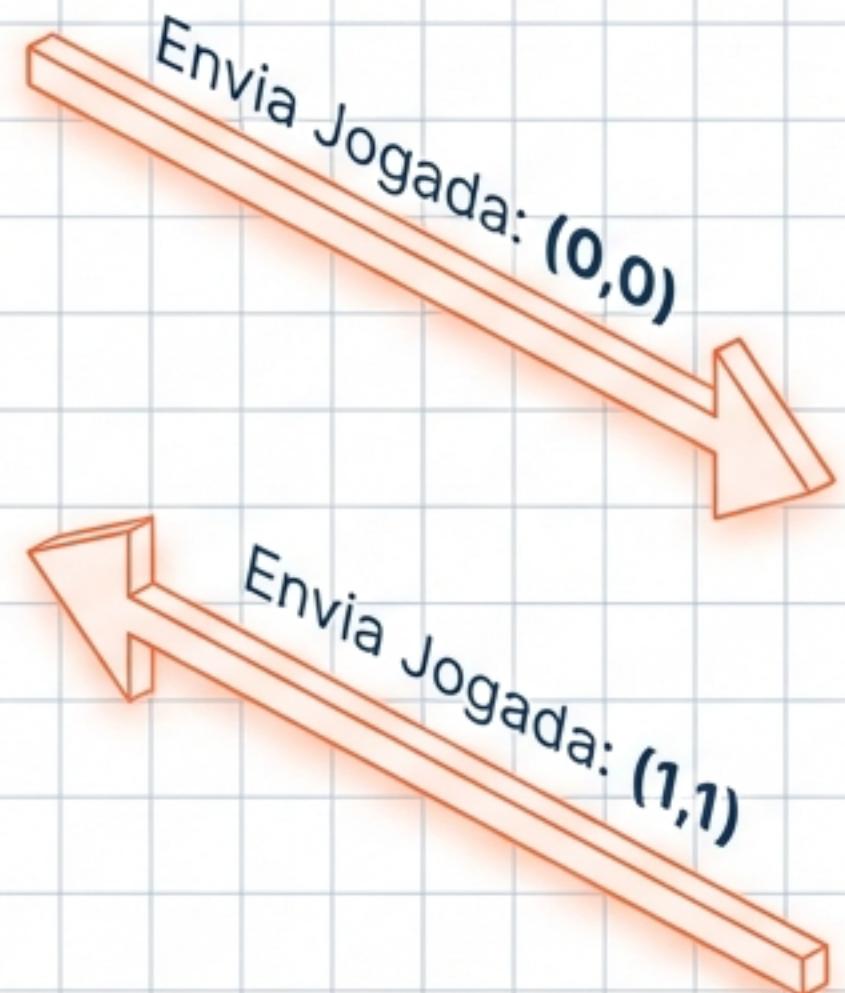
    // Foreground: Atualiza a tela
    Platform.runLater(()-> label.setText(msg));
}).start();
```

# Estudo de Caso 1: Jogo da Velha Distribuído

Servidor (Jogador X)



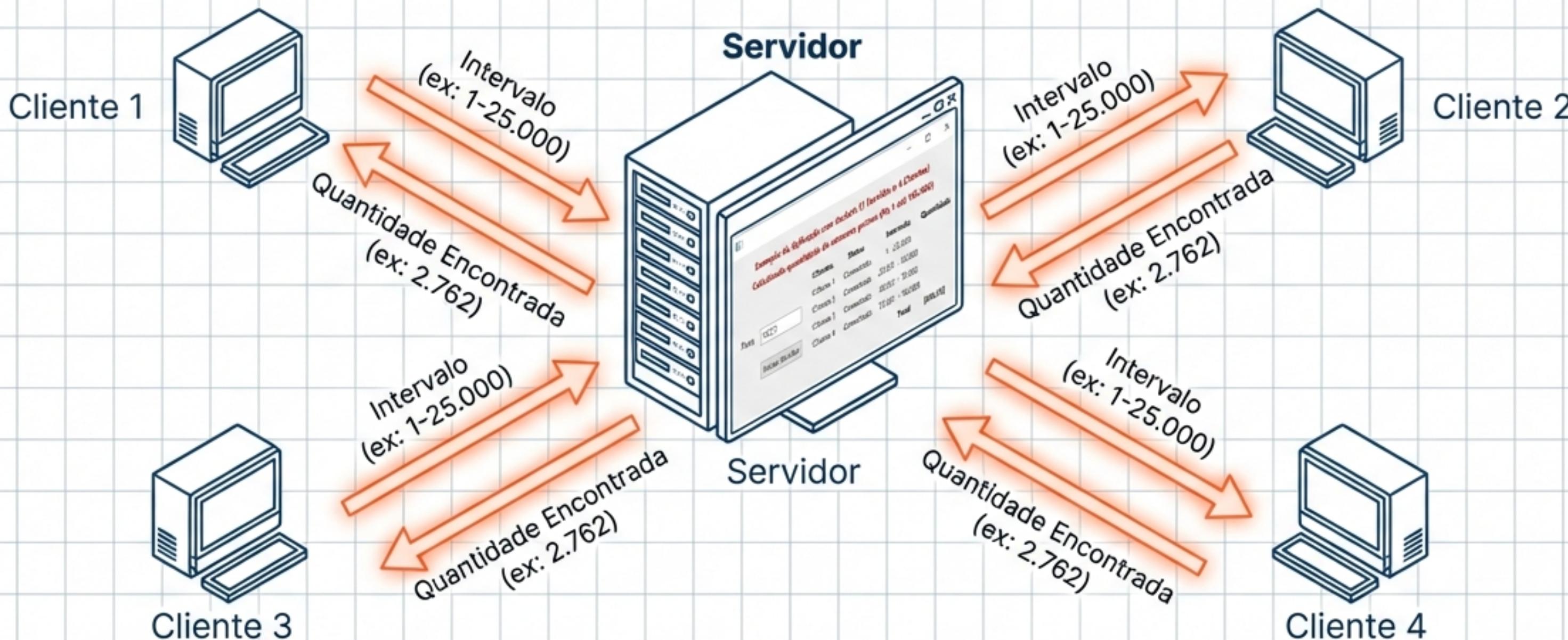
Cliente (Jogador O)



- **Sincronização de Estado:** O que trafega na rede não é o gráfico, mas as COORDENADAS da jogada.
- **Interface:** Bloqueia o turno enquanto espera a resposta do socket.

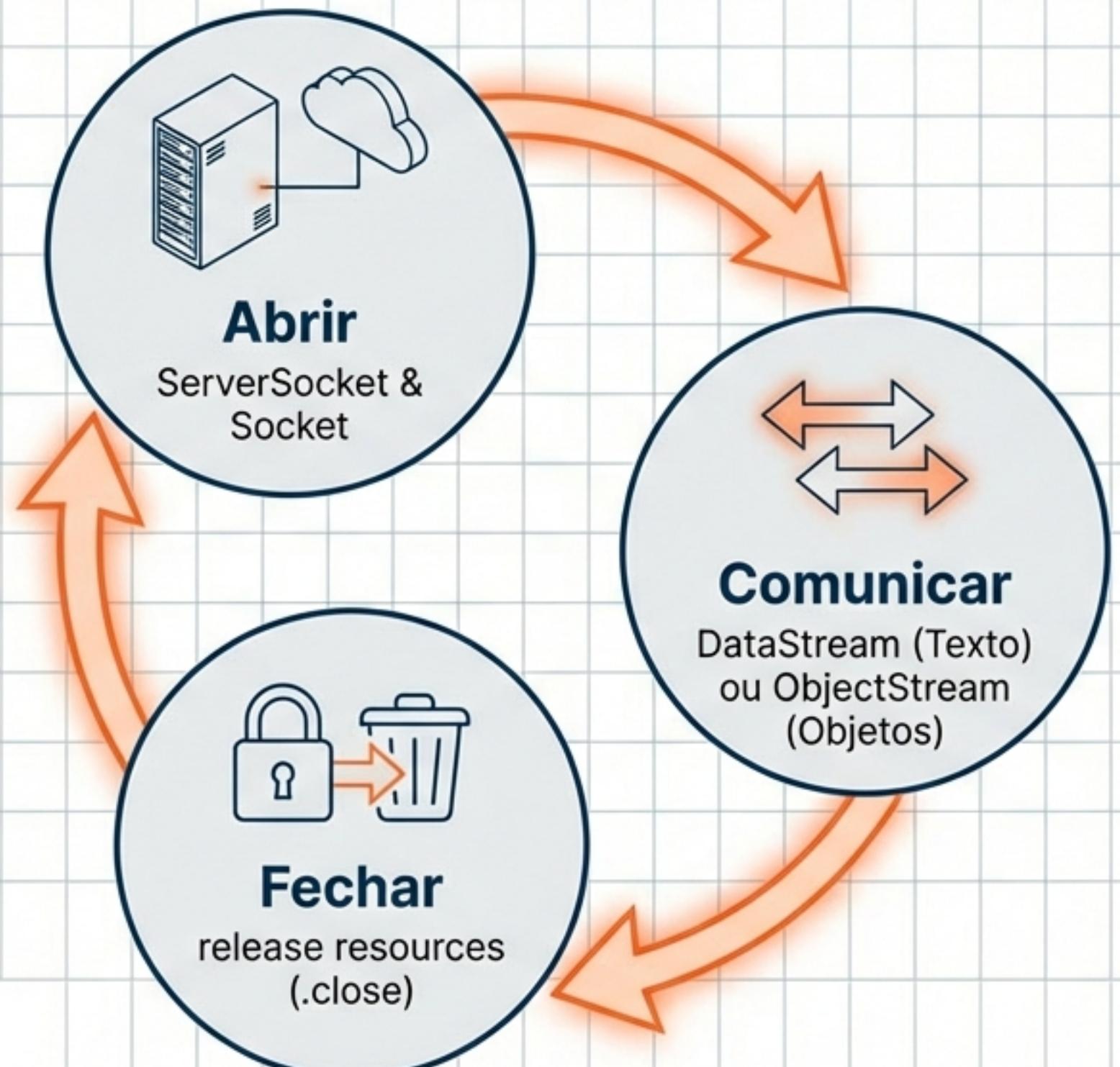
# Estudo de Caso 2: Computação Distribuída

Dividir para Conquistar: Números Primos



- **Arquitetura Mestre-Escravo**: O servidor orquestra, os clientes processam.
- **Resultado**: 4 máquinas trabalhando em paralelo reduzem o tempo total de execução em ~75%.

# Resumo da Jornada e Boas Práticas



## Checklist Final:

- Tratou exceções (try/catch)?
- Fechou os recursos (finally ou try-with-resources)?
- Usou Threads para múltiplos clientes?
- Usou `Serializable` para objetos?
- Usou `Platform.runLater` para atualizar a UI?

# Referências e Créditos

## Fontes:

- Prof. Rafael Vargas Mesquita (IFES) - Códigos e Exemplos Práticos.
- Deitel & Deitel - Java: Como Programar (8<sup>a</sup> Edição), Capítulo 27: Redes.

## Repositórios de Código:

- [github.com/ravarmes/sockets-clienteservidor-java ↗](https://github.com/ravarmes/sockets-clienteservidor-java)
- [github.com/ravarmes/sockets-primos-javafx ↗](https://github.com/ravarmes/sockets-primos-javafx)

