Respostas dos Exercícios - Cliente-Servidor Java Sockets e Chat

Exercício 2: Limite de Clientes e Escalabilidade

Pergunta

O que acontece se na aplicação do Exercício 1, o número de Clientes instanciados for elevado? Há um limite para a quantidade de Clientes na aplicação do Servidor fornecida? Justifique todas as respostas!

Resposta

Sim, há um limite para a quantidade de clientes. Quando o número de clientes é elevado, podem ocorrer os seguintes problemas:

1. Limite de Threads do Sistema Operacional

- O sistema operacional possui um limite máximo de threads que podem ser criadas
- Este limite varia conforme o SO e suas configurações:
  - **Linux**: Tipicamente 32.768 threads (configurável via `/proc/sys/kernel/threads-max`)
  - **Windows**: Depende da memória disponível e configurações do sistema
  - **macOS**: Limite similar ao Linux

2. Recursos de Memória

- Cada thread consome memória para sua stack
- Stack size típico: **512KB a 1MB por thread**
- Com 1000 clientes = 500MB a 1GB apenas para stacks das threads
- Isso sem contar a memória dos objetos Socket, BufferedReader, PrintWriter, etc.

3. Sobrecarga de Context Switching

- Com muitas threads, o sistema operacional gasta tempo excessivo alternando entre elas
- Cada troca de contexto tem um custo de processamento
- **Resultado**: Redução significativa de performance com muitas threads ativas

4. Limite de Descritores de Arquivo (File Descriptors)

- Cada socket aberto consome um file descriptor
- Linux tem limite por processo (tipicamente 1024, configurável via `ulimit`)
- Windows tem limites similares

5. Limites de Portas e Conexões TCP

- Embora o servidor use apenas uma porta (12345), cada conexão cliente consome recursos
- Limite teórico de conexões TCP: ~65.535 por IP (limitado pelas portas efêmeras)

Justificativa Técnica

A implementação fornecida usa o modelo **thread-per-connection**, onde:

- Cada cliente conectado = 1 nova Thread criada
- Thread permanece ativa enquanto cliente está conectado
- Modelo simples, mas **não escalável** para milhares de clientes

Limites práticos estimados:

- **Aplicações pequenas**: 100-500 clientes simultâneos (viável)
- **Aplicações médias**: 1.000-2.000 clientes (possível, mas com otimizações)
- **Acima de 2.000 clientes**: Problemas sérios de performance e estabilidade

Alternativas Escaláveis

Para aplicações com muitos clientes simultâneos, seria necessário usar:

1. **Java NIO (Non-blocking I/O)**
    - Permite gerenciar múltiplas conexões em poucas threads
    - Usa seletores (Selectors) para multiplexação de I/O
2. **Thread Pools**
    - Limita o número de threads criadas
    - Reutiliza threads através de ExecutorService
3. **Frameworks Assíncronos**
    - Netty, Vert.x, ou Spring WebFlux
    - Arquitetura orientada a eventos
4. **Arquiteturas Reativas**
    - Project Reactor, RxJava
    - Melhor uso de recursos

Exercício 3: Executar apenas ClienteBatepapo

Pergunta

Com base no ProjetoBatepapo do Exercício 1, executar, somente, a classe ClienteBatePapo.java, verificar, registrar e explicar, em detalhes, o ocorrido.

Resposta

O que Ocorre

Ao executar apenas o `ClienteBatepapo.java` **sem o servidor rodando**, ocorre uma **falha de conexão**.

Sequência Detalhada de Eventos

1. **Inicialização do Cliente**
   - Método `main()` é executado
   - Programa tenta criar uma instância de `SocketCliente`
2. **Tentativa de Conexão**

```
new Socket(ENDERECO_SERVIDOR, PORTA)
```

   - Cliente tenta estabelecer conexão TCP com `localhost:12345`
   - Sistema operacional envia pacote SYN para o endereço/porta especificada
3. **Recusa de Conexão**
   - Como não há processo escutando na porta 12345, o SO responde com RST (reset)
   - Java lança uma exceção `java.net.ConnectException`
4. **Tratamento de Exceção**
   - Exceção é capturada pelo bloco `catch (IOException e)`
   - Mensagens de erro são exibidas:

```
Erro ao conectar ao servidor: Connection refused (connect failed)
Certifique-se de que o servidor está em execução.
```

5. **Encerramento**
   - Programa encerra imediatamente
   - Nenhuma thread é criada (RecebedorMensagens e EnviadorMensagens não iniciam)

Explicação Técnica

Por que a conexão falha?

- O método `Socket(String host, int port)` é **síncrono e bloqueante**
- Ele tenta estabelecer uma conexão TCP através do **Three-Way Handshake**:
  1. Cliente envia SYN
  2. Servidor deveria responder SYN-ACK
  3. Cliente enviaria ACK final
- Sem servidor, o handshake falha na etapa 2

Tipo de Exceção:

- `java.net.ConnectException: Connection refused`
- Subclasse de `IOException`
- Indica que a conexão foi ativamente recusada pelo host de destino

Mensagens no Console

```
Erro ao conectar ao servidor: Connection refused (connect failed)
Certifique-se de que o servidor está em execução.
```

Conclusão

Este exercício demonstra:

- **Dependência do cliente em relação ao servidor**: Cliente não pode operar sozinho
- **Importância do tratamento de exceções**: Código robusto prevê falhas de conexão
- **Ordem de inicialização**: Servidor DEVE estar rodando ANTES dos clientes
- **Feedback ao usuário**: Mensagens claras sobre o problema ocorrido

---

Exercício 4: Executar apenas ServidorBatepapo

Pergunta

Com base no ProjetoBatepapo do Exercício 1, executar a classe ServidorBatepapo.java, verificar, registrar e explicar, em detalhes, a operação do Servidor, sem sair dele.

Resposta

Operação do Servidor

Ao executar apenas o `ServidorBatepapo.java`, o servidor inicia com sucesso e fica aguardando conexões.

Sequência Detalhada de Eventos

1. **Inicialização**

   ```
   Servidor de Bate-Papo Iniciado...
   ```

   - Método `main()` é executado
   - Variável `servidorSocket` é inicializada como `null`

2. **Criação do ServerSocket**

   ```
   servidorSocket = new ServerSocket(PORTA);
   ```

   - Cria um socket servidor vinculado (bind) à porta 12345
   - Socket entra em modo de escuta (listening)
   - Porta 12345 fica reservada exclusivamente para este processo

3. **Confirmação de Inicialização**

```
Aguardando conexões na porta 12345
```

- Servidor confirma que está pronto para aceitar clientes
- Esta é a última mensagem exibida (por enquanto)

4. **Loop Infinito de Aceitação**

```
while (true) {
    Socket clienteSocket = servidorSocket.accept();
    ...
}
```

- Entra no loop infinito
- Executa o método `accept()`, que é **BLOQUEANTE**

5. **Estado de Espera Bloqueante**

- Thread principal fica "suspensa" no método `accept()`
- Não consome CPU (estado de espera do SO)
- Cursor do console fica "travado" (não retorna prompt)
- Servidor aguarda indefinidamente por conexões

Explicação Técnica

Método `accept()` Bloqueante:

- O método `ServerSocket.accept()` **bloqueia a execução** da thread
- Thread entra em estado WAITING/BLOCKED
- Sistema operacional acorda a thread quando:
  - Um cliente tenta se conectar, OU
  - Ocorre um erro/exceção, OU
  - Socket é fechado

Estado do Sistema:

- **Porta TCP 12345**: Em estado LISTEN
- **Processo Java**: Executando e aguardando
- **Thread Principal**: Bloqueada no accept()
- **Recursos**: Mínimos (apenas socket de escuta)

Verificação no Sistema Operacional:

Linux/Mac:

```
netstat -an | grep 12345
# Resultado: tcp 0 0 :::12345 :::* LISTEN
```

Windows:

```
netstat -an | findstr 12345
# Resultado: TCP 0.0.0.0:12345 0.0.0.0:0 LISTENING
```

Mensagens no Console

```
Servidor de Bate-Papo Iniciado...
Aguardando conexões na porta 12345
[cursor aguardando - sem prompt]
```

Comportamento Esperado

O servidor permanece neste estado até que:

1. **Um cliente se conecte**: Thread desbloqueia, processa conexão, cria thread filha
2. **Ctrl+C seja pressionado**: Interrupção manual do processo
3. **Ocorra erro fatal**: Exceção não tratada (improvável neste código)

Conclusão

Este exercício demonstra:

- **Modelo de servidor bloqueante**: Usa operações síncronas
- **Arquitetura preparada para multithreading**: Loop aceita múltiplas conexões
- **Servidor independente**: Não precisa de clientes para iniciar
- **Estado de espera eficiente**: Não consome CPU enquanto aguarda
- **Design robusto**: Servidor pode rodar indefinidamente aguardando clientes

---

Exercício 5: Servidor + 1 Cliente com Saída

Pergunta

Com base no ProjetoBatepapo do Exercício 1, com o Servidor em operação, executar uma primeira instância da classe ClienteBatepapo.java, digitando algumas mensagens no seu console e, por fim, sair do Cliente, sem sair do Servidor. Verificar, registrar e explicar, em detalhes, as operações do Cliente e do Servidor.

Resposta

Operações do Cliente

1. Conexão Inicial

```
Conectado ao servidor de bate-papo!
```

- Socket TCP estabelecido com sucesso
- Instância de `SocketCliente` criada
- Conexão full-duplex ativa

2. Criação de Threads

- **Thread RecebedorMensagens**: Criada e iniciada
- **Thread EnviadorMensagens**: Criada e iniciada
- Thread principal continua existindo

3. Recepção de Mensagens Iniciais

```
Bem-vindo ao Chat! Digite suas mensagens (digite 'sair' para desconectar)
*** Novo cliente entrou no chat ***
```

- Thread RecebedorMensagens recebe e exibe estas mensagens
- Primeira mensagem vem do servidor (boas-vindas)
- Segunda mensagem é a notificação broadcast

4. Envio de Mensagens do Usuário

Usuário digita: `Olá servidor!`

- Thread EnviadorMensagens captura via `scanner.nextLine()`
- Envia ao servidor via `cliente.enviarMensagem(mensagem)`
- Aguarda próxima entrada do usuário

5. Recepção das Próprias Mensagens (Echo)

```
127.0.0.1: Olá servidor!
```

- Thread RecebedorMensagens recebe mensagem de volta
- Servidor fez broadcast incluindo o remetente
- Cliente vê sua própria mensagem com prefixo de IP

6. Processo de Saída

Usuário digita: `sair`

- Thread EnviadorMensagens detecta comando especial
- Envia "sair" ao servidor
- Exibe: `Desconectando do chat...`
- Executa `cliente.fecharConexao()`
- Fecha BufferedReader, PrintWriter e Socket
- Executa `System.exit(0)` - **encerra JVM**
- Thread RecebedorMensagens é interrompida (daemon ou encerrada)

Operações do Servidor

1. Detecção de Nova Conexão

```
Novo cliente conectado: /127.0.0.1
```

- Método `accept()` retorna com novo Socket
- Thread principal desbloqueia
- Obtém endereço IP do cliente conectado

2. Criação de Thread Dedicada

```
Thread threadCliente = new Thread(new ManipuladorCliente(clienteSocket));
threadCliente.start();
```

- Nova instância de `ManipuladorCliente` criada
- Thread filha iniciada para gerenciar este cliente
- Thread principal **retorna ao accept()** para aguardar novos clientes

3. Inicialização do ManipuladorCliente (Thread Filha)

- Cria `BufferedReader` e `PrintWriter` para o socket do cliente
- Adiciona `PrintWriter` ao conjunto sincronizado `escritoresClientes`

```
synchronized (escritoresClientes) {
    escritoresClientes.add(saida);
}
```

- Envia mensagem de boas-vindas ao cliente
- Faz broadcast: `"*** Novo cliente entrou no chat ***"`

4. Recepção de Mensagens do Cliente

Cliente envia: `Olá servidor!`

```
Mensagem recebida: Olá servidor!
```

- Thread ManipuladorCliente recebe via `entrada.readLine()`
- Exibe no console do servidor
- Chama método `transmitirMensagem()`

5. Broadcast de Mensagens

```
transmitirMensagem(socket.getInetAddress().getHostAddress() + ": " + mensagem);
```

- Adiciona prefixo com IP do remetente
- Percorre conjunto `escritoresClientes` de forma sincronizada
- Envia para **TODOS** os clientes conectados (incluindo remetente)

6. Detecção de Desconexão

Cliente envia: `sair`

- Loop de leitura detecta comando "sair"
- Condição `if (mensagem.equalsIgnoreCase("sair"))` é verdadeira
- Executa `break` - sai do loop while

7. Limpeza de Recursos (bloco finally)

```
synchronized (escritoresClientes) {
    escritoresClientes.remove(saida);
}
transmitirMensagem("*** Um cliente saiu do chat ***");
socket.close();
```

- Remove `PrintWriter` do cliente do conjunto
- Notifica outros clientes (neste caso, nenhum outro)
- Fecha socket do cliente

```
Cliente desconectado: /127.0.0.1
```

8. Continuidade do Servidor

- Thread ManipuladorCliente encerra naturalmente
- Thread principal **continua no accept()** aguardando novos clientes
- Servidor permanece operacional e pronto para novas conexões
- Conjunto `escritoresClientes` agora está vazio

Diagrama de Comunicação

```
[Cliente]                    [Servidor]
    |                            |
    |-------- SYN -------------->|
    |<------- SYN-ACK -----------|
    |-------- ACK -------------->|
    |                            |
    |<--- Bem-vindo ao Chat ------|
    |<--- Novo cliente entrou ----|
```

```
    |                              |
    |--- Olá servidor! ---------->| (recebe)
    |<-- 127.0.0.1: Olá ---------|  (broadcast)
    |                              |
    |-------- sair ------------->| (detecta)
    |                              | (remove cliente)
    |<------ FIN ---------------- | (fecha socket)
    |                              |
  [ENCERRA]                [CONTINUA]
```

Conclusão

Este exercício demonstra:

1. **Comunicação Bidirecional**: Cliente e servidor trocam mensagens em ambas direções
2. **Multithreading no Cliente**: Duas threads permitem envio e recepção simultâneos
3. **Multithreading no Servidor**: Thread dedicada por cliente
4. **Broadcast funcional**: Mensagens são retransmitidas para todos
5. **Desconexão limpa**: Recursos liberados adequadamente
6. **Servidor persistente**: Continua rodando após cliente desconectar
7. **Sincronização**: Acesso ao conjunto de clientes é thread-safe

---

Exercício 6: Servidor + 1 Cliente sem Sair

Pergunta

Com base no ProjetoBatepapo do Exercício 1, com o Servidor em operação, executar, novamente, uma primeira instância da classe ClienteBatepapo.java, digitando algumas mensagens no seu console e, por fim, não sair do Cliente, nem do Servidor. Verificar, registrar e explicar, em detalhes, as operações do Cliente e do Servidor.

Resposta

Operações Iniciais (Idênticas ao Exercício 5)

Cliente:

1. Conecta ao servidor com sucesso
2. Cria threads RecebedorMensagens e EnviadorMensagens
3. Recebe mensagens de boas-vindas e notificação
4. Usuário digita algumas mensagens

Servidor:

1. Aceita conexão do cliente
2. Cria thread ManipuladorCliente
3. Adiciona PrintWriter ao conjunto
4. Recebe e faz broadcast das mensagens

Diferença Principal: Estado de Espera Contínua

Cliente Permanece Conectado:

Thread EnviadorMensagens:

```
while (cliente.estaConectado()) {
    String mensagem = scanner.nextLine(); // BLOQUEADO AQUI
    ...
}
```

- Fica bloqueada aguardando entrada do usuário
- Cursor ativo no console aguardando digitação
- Pronta para capturar próxima mensagem

Thread RecebedorMensagens:

```
while (cliente.estaConectado() && (mensagem = cliente.receberMensagem()) != null)
{
    // BLOQUEADA AQUI
    System.out.println(mensagem);
}
```

- Fica bloqueada aguardando mensagem do servidor
- `readLine()` aguarda dados no InputStream
- Pronta para exibir mensagens recebidas

Servidor Mantém Conexão Ativa:

Thread Principal:

```
while (true) {
    Socket clienteSocket = servidorSocket.accept(); // BLOQUEADO AQUI
    ...
}
```

- Retornou ao accept() após criar thread do cliente
- Bloqueada aguardando **NOVOS** clientes
- Cliente atual já está sendo gerenciado por thread filha

Thread ManipuladorCliente:

```
while ((mensagem = entrada.readLine()) != null) { // BLOQUEADA AQUI
    if (mensagem.equalsIgnoreCase("sair")) {
        break;
```

```
    }
    transmitirMensagem(...);
}
```

- Fica bloqueada aguardando mensagens do cliente
- `readLine()` espera dados no InputStream do socket
- Pronta para processar próxima mensagem

Estado dos Recursos

Cliente:

- **1 Socket TCP**: Aberto e conectado
- **3 Threads ativas**:
  - Thread main (pode ter encerrado dependendo da implementação)
  - Thread RecebedorMensagens (BLOCKED em read)
  - Thread EnviadorMensagens (BLOCKED em readLine)
- **Streams**: BufferedReader e PrintWriter abertos
- **Scanner**: Aguardando input do usuário

Servidor:

- **1 ServerSocket**: Escutando na porta 12345
- **1 Socket de cliente**: Conectado ao cliente
- **2+ Threads ativas**:
  - Thread main (BLOCKED em accept)
  - Thread ManipuladorCliente (BLOCKED em readLine)
- **Conjunto escritoresClientes**: Contém 1 PrintWriter
- **Recursos mínimos**: Threads em estado de espera não consomem CPU

Comportamento Durante Espera

Usuário pode:

1. **Continuar enviando mensagens**: Basta digitar e pressionar Enter
2. **Receber mensagens**: Se servidor enviar, RecebedorMensagens exibirá
3. **Deixar idle indefinidamente**: Conexão permanece ativa

Mensagens aparecem assim:

```
[Console Cliente]
Conectado ao servidor de bate-papo!
Bem-vindo ao Chat! Digite suas mensagens (digite 'sair' para desconectar)
*** Novo cliente entrou no chat ***
Primeira mensagem
127.0.0.1: Primeira mensagem
Segunda mensagem
127.0.0.1: Segunda mensagem
[cursor aguardando...]
```

```
[Console Servidor]
Servidor de Bate-Papo Iniciado...
Aguardando conexões na porta 12345
Novo cliente conectado: /127.0.0.1
Mensagem recebida: Primeira mensagem
Mensagem recebida: Segunda mensagem
[aguardando...]
```

Verificação no Sistema Operacional

Linux/Mac:

```
netstat -an | grep 12345
# Resultado:
tcp  0  0  :::12345       :::*         LISTEN      (servidor)
tcp  0  0  127.0.0.1:12345  127.0.0.1:xxxxx  ESTABLISHED (servidor-cliente)
tcp  0  0  127.0.0.1:xxxxx  127.0.0.1:12345  ESTABLISHED (cliente-servidor)
```

Windows:

```
netstat -an | findstr 12345
# Mostra conexões ESTABLISHED
```

Timeout e Keep-Alive

Importante:

- Conexão TCP não tem timeout por padrão
- Socket permanece aberto indefinidamente se:
  - Nenhum lado fechar explicitamente
  - Não houver falha de rede
  - Sistema operacional não forçar fechamento

Keep-Alive TCP:

- Por padrão, TCP pode enviar keep-alive packets
- Detecta conexões "mortas" (ambos os lados silenciosos)
- Configurável via socket options

Cenários Possíveis

1. Cliente continua enviando mensagens:

- Funciona normalmente

- Cada mensagem é recebida e retransmitida

2. Novo cliente se conecta:

- Servidor aceita na thread principal
- Ambos os clientes recebem notificação
- Mensagens passam a ser enviadas para AMBOS

3. Falha de rede:

- Socket detecta erro em próxima operação I/O
- IOException é lançada
- Recursos são liberados no bloco finally

4. Ctrl+C no cliente:

- JVM encerra abruptamente
- Socket é fechado pelo SO
- Servidor detecta `readLine() == null`
- Servidor executa limpeza

5. Ctrl+C no servidor:

- Todos os sockets são fechados
- Clientes recebem IOException
- Clientes detectam desconexão

Conclusão

Este exercício demonstra:

1. **Persistência de Conexão**: Conexões TCP não expiram automaticamente
2. **Operações Bloqueantes**: Threads ficam em estado de espera eficiente
3. **Prontidão Contínua**: Sistema está sempre pronto para processar eventos
4. **Independência das Threads**: Cada thread opera em seu próprio loop
5. **Baixo Consumo de Recursos**: Threads bloqueadas não consomem CPU
6. **Escalabilidade Preparada**: Servidor pode aceitar mais clientes a qualquer momento
7. **Design Assíncrono**: Envio e recepção são independentes

Exercício 7: Servidor + 2 Clientes

Pergunta

Com base no ProjetoBatepapo do Exercício 1, com o Servidor e a primeira instância do Cliente em operação, executar uma segunda instância da classe ClienteBatepapo.java, digitando algumas mensagens no seu console e no console da instância do outro Cliente, alternadamente e, por fim, não sair dos Clientes, nem do Servidor. Verificar, registrar e explicar, em detalhes, as operações dos Clientes e do Servidor.

Resposta

Estado Inicial

Servidor:

- Thread principal em accept() aguardando
- Thread ManipuladorCliente-1 gerenciando Cliente 1
- Conjunto escritoresClientes com 1 elemento

Cliente 1:

- Conectado e operacional
- Threads ativas aguardando entrada/mensagens

Conexão do Cliente 2

1. Cliente 2 Inicia Conexão

```
[Cliente 2]
Conectado ao servidor de bate-papo!
Bem-vindo ao Chat! Digite suas mensagens (digite 'sair' para desconectar)
*** Novo cliente entrou no chat ***
```

2. Servidor Aceita Cliente 2

```
[Servidor]
Novo cliente conectado: /127.0.0.1
```

- Thread principal desbloqueia do accept()
- Cria nova Thread ManipuladorCliente-2
- Retorna ao accept() aguardando Cliente 3

3. Cliente 1 Recebe Notificação

```
[Cliente 1]
*** Novo cliente entrou no chat ***
```

- Thread RecebedorMensagens do Cliente 1 recebe broadcast
- Usuário do Cliente 1 é notificado da chegada do Cliente 2

Estado Após Cliente 2 Conectar

Servidor:

```
escritoresClientes = [PrintWriter-Cliente1, PrintWriter-Cliente2]
```

- 3 Threads ativas:
  - Thread Main (BLOCKED em accept)
  - Thread ManipuladorCliente-1 (BLOCKED lendo Cliente 1)
  - Thread ManipuladorCliente-2 (BLOCKED lendo Cliente 2)

Cliente 1:

- 2 Threads: RecebedorMensagens e EnviadorMensagens
- Aguardando entrada do usuário

Cliente 2:

- 2 Threads: RecebedorMensagens e EnviadorMensagens
- Aguardando entrada do usuário

Troca de Mensagens

Cenário 1: Cliente 1 Envia Mensagem

Cliente 1 digita: `Olá Cliente 2!`

Fluxo:

1. Thread EnviadorMensagens (Cliente 1) envia ao servidor
2. Thread ManipuladorCliente-1 (Servidor) recebe
3. Servidor exibe: `Mensagem recebida: Olá Cliente 2!`
4. Método `transmitirMensagem()` percorre `escritoresClientes`
5. Envia para **ambos** os PrintWriters

Resultado:

```
[Cliente 1 - console]
Olá Cliente 2!
127.0.0.1: Olá Cliente 2!
```

```
[Cliente 2 - console]
127.0.0.1: Olá Cliente 2!
```

Observação: Cliente 1 vê sua própria mensagem com prefixo de IP!

Cenário 2: Cliente 2 Responde

Cliente 2 digita: `Olá Cliente 1! Tudo bem?`

Fluxo:

1. Thread EnviadorMensagens (Cliente 2) envia ao servidor
2. Thread ManipuladorCliente-2 (Servidor) recebe

3. Servidor exibe: `Mensagem recebida: Olá Cliente 1! Tudo bem?`
4. Método `transmitirMensagem()` percorre `escritoresClientes`
5. Envia para **ambos** os PrintWriters

Resultado:

```
[Cliente 1 - console]
127.0.0.1: Olá Cliente 2!
127.0.0.1: Olá Cliente 1! Tudo bem?
```

```
[Cliente 2 - console]
127.0.0.1: Olá Cliente 2!
Olá Cliente 1! Tudo bem?
127.0.0.1: Olá Cliente 1! Tudo bem?
```

Cenário 3: Mensagens Alternadas

Cliente 1: `Sim, e você?` Cliente 2: `Também!` Cliente 1: `Legal!`

Resultado em ambos os consoles:

```
127.0.0.1: Sim, e você?
127.0.0.1: Também!
127.0.0.1: Legal!
```

Análise do Método transmitirMensagem()

```
private void transmitirMensagem(String mensagem) {
    synchronized (escritoresClientes) {
        for (PrintWriter escritor : escritoresClientes) {
            escritor.println(mensagem);
        }
    }
}
```

Funcionamento:

1. **Sincronização**: Bloco `synchronized` garante thread-safety
2. **Iteração**: Percorre TODOS os clientes no conjunto
3. **Broadcast**: Cada PrintWriter recebe a mesma mensagem
4. **Inclusão do remetente**: Até quem enviou recebe de volta

Por que remetente recebe própria mensagem?

- Design simples: broadcast para todos sem exceções
- Vantagem: Confirmação visual de envio
- Desvantagem: Duplicação (usuário já viu o que digitou)

Comportamento de Sincronização

Thread-Safety:

```
synchronized (escritoresClientes) {
    escritoresClientes.add(saida);
}

synchronized (escritoresClientes) {
    escritoresClientes.remove(saida);
}

synchronized (escritoresClientes) {
    for (PrintWriter escritor : escritoresClientes) {
        escritor.println(mensagem);
    }
}
```

Por que necessário?

- Múltiplas threads acessam o mesmo conjunto
- Operações de adicionar/remover/iterar devem ser atômicas
- Previne `ConcurrentModificationException`
- Garante que broadcast use lista consistente

Ordenação de Mensagens

Questão: Mensagens sempre chegam na ordem correta?

Resposta: Depende!

- **Dentro de uma conexão**: TCP garante ordem (FIFO)
- **Entre conexões diferentes**: Não há garantia absoluta
    - Servidor processa threads independentemente
    - Broadcast percorre conjunto em ordem de iteração
    - Race conditions podem causar