Exercício 01 - Respostas Detalhadas

Cliente-Servidor com Java Sockets

Autor: Felipe Fazio da Costa

RA: 23.00055-4

---

a) Criar projeto SimpleClientServer

Realizado: Projeto criado com as classes:

- `SimpleServerTest.java` - Servidor socket na porta 6789
- `SimpleClientTest.java` - Cliente que conecta ao servidor

---

b) Executar somente SimpleClientTest.java

O que ocorre:

```
Exception in thread "main" java.net.ConnectException: Connection refused: connect
```

Explicação detalhada:

Por que o erro ocorre:

1. **Cliente tenta conectar**: Quando executamos apenas o `SimpleClientTest.java`, ele tenta criar um socket de conexão com o servidor:

   ```
   Socket clientSocket = new Socket("localhost", 6789);
   ```

2. **Servidor não está disponível**: Como não iniciamos o servidor (`SimpleServerTest.java`), não há nenhum processo escutando na porta 6789.

3. **Connection Refused**: O sistema operacional verifica que ninguém está "ouvindo" na porta 6789 e recusa a conexão imediatamente.

Detalhes técnicos:

- O cliente envia um pacote SYN (synchronize) para iniciar o handshake TCP/IP
- O sistema operacional do host local responde com RST (reset) porque a porta não está em estado LISTEN
- A JVM lança `ConnectException` indicando que a conexão foi ativamente recusada
- O programa termina abruptamente sem conseguir estabelecer comunicação

Conclusão: É impossível executar um cliente sem que o servidor esteja ativo e aguardando conexões.

---

c) Executar SimpleServerTest.java

O que ocorre:

```
Servidor aguardando conexão na porta 6789...
```

Explicação detalhada:

Comportamento do servidor:

1. **ServerSocket é criado**:

   ```
   ServerSocket serverSocket = new ServerSocket(6789);
   ```

   - O servidor "reserva" a porta 6789 do sistema operacional
   - O socket entra em estado LISTEN, aguardando conexões

2. **Método accept() bloqueia a execução**:

   ```
   Socket connectionSocket = serverSocket.accept();
   ```

   - O método `accept()` é **bloqueante** (blocking)
   - O thread fica suspenso aguardando uma conexão
   - O programa não continua até que um cliente se conecte

3. **Servidor fica aguardando indefinidamente**:

   - CPU em estado de espera (não consome processamento)
   - Porta 6789 fica reservada e monitorada pelo SO
   - Sistema operacional enfileira tentativas de conexão (backlog)

Detalhes técnicos:

- O SO mantém uma fila de conexões pendentes (default: 50 conexões)
- O socket está em modo passivo, aguardando handshake TCP/IP
- O servidor não faz polling ativo - o SO notifica quando há conexão

Conclusão: O servidor permanece em execução, aguardando pacientemente que algum cliente inicie uma conexão na porta 6789.

---

d) Executar SimpleClientTest.java com o servidor ativo

O que ocorre:

Terminal do Cliente:

```
Conectado ao servidor!
Digite uma mensagem (ou -1 para sair): Olá servidor
Resposta do servidor: OLÁ SERVIDOR
Digite uma mensagem (ou -1 para sair): Como você está?
Resposta do servidor: COMO VOCÊ ESTÁ?
Digite uma mensagem (ou -1 para sair): Teste de conexão
Resposta do servidor: TESTE DE CONEXÃO
Digite uma mensagem (ou -1 para sair): -1
Servidor: Conexão encerrada.
Encerrando conexão...
Cliente desconectado.
```

Terminal do Servidor:

```
Servidor aguardando conexão na porta 6789...
Cliente conectado!
Recebido do cliente: Olá servidor
Resposta enviada ao cliente: OLÁ SERVIDOR
Recebido do cliente: Como você está?
Resposta enviada ao cliente: COMO VOCÊ ESTÁ?
Recebido do cliente: Teste de conexão
Resposta enviada ao cliente: TESTE DE CONEXÃO
Recebido do cliente: -1
Comando de encerramento recebido. Fechando servidor...
Servidor encerrado.
```

Explicação detalhada:

Fase 1 - Estabelecimento da conexão (TCP Three-Way Handshake):

1. **Cliente inicia handshake**:
    - Cliente envia SYN para porta 6789
    - Servidor responde com SYN-ACK
    - Cliente confirma com ACK
    - Conexão TCP estabelecida!
2. **Accept() retorna**:
    - Método `accept()` que estava bloqueado retorna um novo `Socket`
    - Este socket representa a conexão específica com este cliente
    - ServerSocket original continua disponível para outras conexões

Fase 2 - Comunicação bidirecional:

3. **Streams são criados**:
    - **Cliente**: `DataOutputStream` (saída) e `BufferedReader` (entrada)
    - **Servidor**: `BufferedReader` (entrada) e `DataOutputStream` (saída)
    - Streams ficam vinculados ao socket TCP

4. **Loop de mensagens**:
    - Cliente lê entrada do usuário com `inFromUser.readLine()`
    - Cliente envia para servidor: `outToServer.writeBytes(sentence + "\n")`
    - Dados trafegam via TCP/IP pela interface loopback (localhost)
    - Servidor recebe: `inFromClient.readLine()` (bloqueante até receber `\n`)
    - Servidor processa: `toUpperCase()`
    - Servidor envia resposta: `outToClient.writeBytes()`
    - Cliente recebe resposta: `inFromServer.readLine()`
    - Ciclo se repete

Fase 3 - Encerramento gracioso:

5. **Cliente envia "-1"**:
    - Comando especial detectado pelo servidor no `if (clientSentence.equals("-1"))`
    - Servidor envia mensagem de confirmação: "Conexão encerrada."
    - Loops em ambos os lados executam `break`

6. **Fechamento da conexão (TCP Four-Way Handshake)**:
    - Cliente fecha socket: `clientSocket.close()`
    - Servidor fecha sockets: `connectionSocket.close()` e `serverSocket.close()`
    - FIN-ACK trocado entre cliente e servidor
    - Recursos são liberados pelo SO

Detalhes técnicos importantes:

- **Protocolo síncrono**: Cliente aguarda resposta antes de enviar nova mensagem
- **Linha como delimitador**: O `\n` é essencial - sem ele, `readLine()` fica bloqueado
- **Buffering**: Dados podem ser bufferizados antes de serem enviados (Nagle's algorithm)
- **Persistência da conexão**: Uma única conexão TCP é mantida para múltiplas mensagens
- **Porta efêmera do cliente**: O SO atribui automaticamente uma porta alta (> 1024) para o cliente

Conclusão: A comunicação cliente-servidor funciona perfeitamente através de sockets TCP, com o servidor processando mensagens (convertendo para maiúsculas) até receber o comando de término.

---

e) Múltiplas instâncias simultâneas de clientes

O que ocorre:

Terminal do Servidor:

```
Servidor aguardando conexão na porta 6789...
Cliente conectado!
Recebido do cliente: Primeira mensagem do cliente 1
Resposta enviada ao cliente: PRIMEIRA MENSAGEM DO CLIENTE 1
(servidor fica aguardando próxima mensagem do cliente 1)
```

Terminal do Cliente 1 (primeira instância):

```
Conectado ao servidor!
Digite uma mensagem (ou -1 para sair): Primeira mensagem do cliente 1
Resposta do servidor: PRIMEIRA MENSAGEM DO CLIENTE 1
Digite uma mensagem (ou -1 para sair): (aguardando...)
```

Terminal do Cliente 2 (segunda instância):

```
Servidor aguardando conexão na porta 6789...
Cliente conectado!
Recebido do cliente: Primeira mensagem do cliente 1
.
.
.
```

Explicação detalhada:

Por que a segunda instância do cliente não conecta:

1. **Servidor single-threaded**:

```
Socket connectionSocket = serverSocket.accept(); // Chamado UMA vez
while (true) {
    String clientSentence = inFromClient.readLine(); // BLOQUEIA aqui
    // ...
}
```

   - O servidor chama `accept()` apenas **uma vez**
   - Após aceitar o primeiro cliente, entra no loop de leitura
   - `readLine()` bloqueia a thread aguardando dados do Cliente 1

2. **Segunda conexão fica na fila (backlog)**:

   - Quando Cliente 2 tenta conectar, o TCP handshake ocorre normalmente
   - SO coloca a conexão na **fila de pending connections** do ServerSocket
   - Cliente 2 completa `new Socket()` com sucesso
   - MAS o servidor nunca chama `accept()` novamente para processar esta conexão

- Cliente 2 fica travado em `inFromServer.readLine()` esperando dados que nunca virão

3. **Comportamento ao digitar mensagens**:

   Cliente 1 (mensagens funcionam):

   - Consegue enviar e receber normalmente
   - Servidor está em loop atendendo esta conexão

   Cliente 2 (travado):

   - `outToServer.writeBytes()` pode até enviar dados
   - Dados ficam no buffer TCP, mas servidor não lê
   - `inFromServer.readLine()` bloqueia indefinidamente
   - Interface parece "congelada"

Diagrama do problema:

```
Servidor (Thread única)
    ↓
[accept()] → Cliente 1 conectado
    ↓
[loop while] → readLine() ← BLOQUEADO aguardando Cliente 1
    ↑
    └── NUNCA chega aqui para fazer novo accept()

ServerSocket (backlog queue)
    ← Cliente 2 (aguardando na fila, nunca aceito)
```

Detalhes técnicos:

- **Fila de backlog**: Parâmetro do `ServerSocket(int port, int backlog)`
- **Default backlog**: Geralmente 50 conexões pendentes
- **TCP estabelecido mas não aceito**: Conexão TCP existe, mas aplicação não a processa
- **Timeout eventual**: Cliente 2 pode ter timeout após minutos (SO dependent)

Limitação fundamental: Este servidor é **sequencial/iterativo**, não **concorrente/paralelo**. Só pode atender **um cliente por vez**.

Solução necessária: Implementar servidor **multi-threaded** onde cada cliente é atendido por uma thread separada.

Conclusão: Um servidor single-threaded só consegue atender um cliente por vez. Clientes adicionais ficam aguardando na fila de conexões pendentes, mas nunca são efetivamente atendidos pelo código do servidor.

---

f) Encerrar segunda instância e voltar à primeira

O que ocorre:

Após encerrar Cliente 2 (Ctrl+C ou fechar terminal):

Terminal do Cliente 1:

```
Digite uma mensagem (ou -1 para sair): Nova mensagem após encerrar cliente 2
Resposta do servidor: NOVA MENSAGEM APÓS ENCERRAR CLIENTE 2
Digite uma mensagem (ou -1 para sair): Tudo funcionando
Resposta do servidor: TUDO FUNCIONANDO
```

Terminal do Servidor:

```
Recebido do cliente: Nova mensagem após encerrar cliente 2
Resposta enviada ao cliente: NOVA MENSAGEM APÓS ENCERRAR CLIENTE 2
Recebido do cliente: Tudo funcionando
Resposta enviada ao cliente: TUDO FUNCIONANDO
```

Explicação detalhada:

O que acontece com Cliente 2:

1. **Encerramento abrupto**:
    - Ao fechar o terminal ou Ctrl+C, a JVM é encerrada sem `close()` gracioso
    - SO fecha o socket do Cliente 2 forçadamente
    - TCP envia FIN ou RST para o servidor
    - **MAS** o servidor nunca havia chamado `accept()` para esta conexão
    - Conexão é removida da fila de backlog do ServerSocket
2. **Servidor nem percebe**:
    - Servidor continua bloqueado em `readLine()` aguardando Cliente 1
    - Cliente 2 nunca foi "aceito" pelo código da aplicação
    - Não há recursos da aplicação alocados para Cliente 2
    - SO simplesmente remove a conexão pendente

O que acontece com Cliente 1:

3. **Comunicação normal continua**:
    - Cliente 1 tem conexão estabelecida e sendo processada
    - Thread do servidor está em loop dedicado a este cliente
    - Cada mensagem digitada é enviada, processada e respondida
    - Nenhum impacto do encerramento do Cliente 2
4. **Estado da conexão TCP**:
    - Conexão TCP entre Cliente 1 e Servidor permanece ativa (ESTABLISHED)
    - Buffers de envio/recebimento funcionando normalmente
    - Keep-alive packets podem ser enviados (se habilitado)

Análise importante:

```
ANTES do encerramento do Cliente 2:
- Servidor atendendo: Cliente 1
- Fila de espera: [Cliente 2]

DEPOIS do encerramento do Cliente 2:
- Servidor atendendo: Cliente 1
- Fila de espera: []
```

Por que Cliente 1 não é afetado:

- **Isolamento de conexões**: Cada socket é independente
- **File descriptors diferentes**: SO mantém FDs separados
- **Buffers independentes**: Cada conexão tem seus próprios buffers TCP
- **Sem compartilhamento de estado**: Servidor não compartilha dados entre conexões

Se fosse servidor multi-threaded:

- Cliente 2 teria sua própria thread
- Encerrar Cliente 2 terminaria apenas sua thread
- Cliente 1 continuaria normalmente em sua própria thread
- Resultado seria o mesmo, mas por motivos diferentes

Conclusão: O encerramento do Cliente 2 (que nunca foi efetivamente atendido pelo servidor) não afeta em nada o Cliente 1, que continua sua comunicação normalmente. Cada conexão TCP é independente e isolada.

---

g) Executar segunda instância do servidor

O que ocorre:

Terminal do Servidor 2 (nova instância):

```
Exception in thread "main" java.net.BindException: Address already in use:
JVM_Bind
    at java.net.DualStackPlainSocketImpl.bind0(Native Method)
    at java.net.DualStackPlainSocketImpl.socketBind(...)
    at java.net.AbstractPlainSocketImpl.bind(...)
    at java.net.PlainSocketImpl.bind(...)
    at java.net.ServerSocket.bind(...)
    at java.net.ServerSocket.<init>(...)
    at SimpleServerTest.main(SimpleServerTest.java:8)
```

Terminal do Servidor 1 (continua funcionando):

```
Cliente conectado!
Recebido do cliente: Mensagem de teste
Resposta enviada ao cliente: MENSAGEM DE TESTE
(continua operando normalmente)
```

Explicação detalhada:

Por que o erro ocorre:

1. **Porta já está em uso**:

```
ServerSocket serverSocket = new ServerSocket(6789); // ERRO aqui no Servidor 2
```

   - Porta 6789 já está vinculada (bound) ao Servidor 1
   - SO mantém tabela de portas alocadas
   - Cada porta pode ser vinculada a apenas **um processo por vez** (mesma interface)

2. **Bind operation falha**:

   - `ServerSocket` tenta fazer `bind()` na porta 6789
   - SO verifica que a porta está em uso
   - Sistema retorna erro EADDRINUSE (Unix/Linux) ou WSAEADDRINUSE (Windows)
   - JVM traduz para `BindException`

3. **Proteção do sistema operacional**:

   - Previne conflitos de portas
   - Garante que apenas um servidor escute em cada porta
   - Evita ambiguidade: "Para qual processo devo enviar dados na porta 6789?"

Detalhes técnicos:

Estrutura da vinculação de porta:

```
Interface de rede: 127.0.0.1 (localhost)
Porta: 6789
Protocolo: TCP
Estado: LISTEN
Processo: Java (PID 12345) - Servidor 1
```

Tentativa do Servidor 2:

```
Interface: 127.0.0.1
Porta: 6789 ← JÁ ESTÁ OCUPADA!
```

```
Protocolo: TCP
Resultado: ERRO - BindException
```

Estados possíveis de uma porta TCP:

- **CLOSED**: Porta não está em uso
- **LISTEN**: Servidor aguardando conexões (Servidor 1)
- **ESTABLISHED**: Conexão ativa entre cliente-servidor
- **TIME_WAIT**: Aguardando encerramento completo

Exceções à regra (casos avançados):

1. **SO_REUSEADDR**:

```
serverSocket.setReuseAddress(true); // Antes do bind
```

   - Permite reusar porta em TIME_WAIT
   - NÃO permite dois servidores simultâneos
   - Útil para restart rápido de servidores

2. **Portas em interfaces diferentes**:

```
ServerSocket s1 = new ServerSocket(6789, 50,
InetAddress.getByName("192.168.1.10"));
ServerSocket s2 = new ServerSocket(6789, 50,
InetAddress.getByName("192.168.1.11"));
```

   - Possível se o host tiver múltiplas interfaces de rede
   - Não se aplica ao localhost (interface única)

3. **Load balancing com SO_REUSEPORT** (Linux):

   - Recurso do kernel que permite múltiplos sockets na mesma porta
   - Requer suporte do SO e configuração especial
   - Não disponível em Java padrão (requer JNI ou bibliotecas nativas)

Como verificar portas em uso:

Windows (PowerShell):

```
netstat -ano | findstr :6789
# Mostra: TCP  127.0.0.1:6789  0.0.0.0:0  LISTENING  PID
```

Linux/Mac:

```
lsof -i :6789
netstat -tuln | grep 6789
```

Soluções possíveis:

1. **Usar porta diferente**:

```
ServerSocket serverSocket = new ServerSocket(6790); // Porta diferente
```

2. **Encerrar primeiro servidor**:
   - Fechar Servidor 1 libera a porta
   - Aguardar alguns segundos (TIME_WAIT)
   - Iniciar Servidor 2

3. **Servidor multi-threaded (arquitetura adequada)**:
   - Um único ServerSocket com múltiplas threads
   - Cada thread atende um cliente diferente
   - Todos compartilham a mesma porta

Analogia do mundo real: É como tentar abrir duas lojas com o mesmo endereço. O correio não saberia para qual loja entregar as cartas. O SO garante que cada "endereço" (porta) seja único.

Conclusão: É impossível ter dois servidores escutando na mesma porta simultaneamente. Esta é uma restrição fundamental do TCP/IP implementada pelo sistema operacional para evitar ambiguidade no roteamento de pacotes.
