Exercício 01 - Respostas Detalhadas

Cliente-Servidor com Java Sockets Bidirecional

Autor: Enzo Oliveira D'Onofrio

RA: 23.01561-6

---

a) Criar projeto SimpleClientServer

Realizado: Projeto criado com as classes:

- `SimpleServerTest.java` - Servidor socket na porta 3334
- `SimpleClientTest.java` - Cliente que conecta ao servidor

---

b) Executar somente SimpleClientTest.java

O que ocorre:

```
*v*v*v* CONSOLE DO CLIENTE *v*v*v*
Erro no Cliente: Connection refused
```

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", 3334);
   ```

2. **Servidor não está disponível**: Como não iniciamos o servidor (`SimpleServerTest.java`), não há nenhum processo escutando na porta 3334.

3. **Connection Refused**: O sistema operacional verifica que ninguém está "ouvindo" na porta 3334 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:

```
*v*v*v* CONSOLE DO SERVIDOR *v*v*v*
Servidor iniciado e escutando a porta 3334
```

Explicação detalhada:

Comportamento do servidor:

1. **ServerSocket é criado**:

   ```
   ServerSocket serverSocket = new ServerSocket(3334);
   ```

   - O servidor "reserva" a porta 3334 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 3334 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 3334.

---

d) Executar SimpleClientTest.java com o servidor ativo

O que ocorre:

Terminal do Cliente:

```
*v*v*v* CONSOLE DO CLIENTE *v*v*v*
Cliente IP 127.0.0.1 conectado ao Servidor pela porta 3334
Digite na Entrada a mensagem para o Servidor!
Enviou ao Servidor: olá
Ecoou do Servidor: olá
Comunicacao OK!
Digite na Entrada a mensagem para o Servidor!
Enviou ao Servidor: como você está?
Ecoou do Servidor: como você está?
Comunicacao OK!
Digite na Entrada a mensagem para o Servidor!
Enviou ao Servidor: teste
Ecoou do Servidor: teste
Comunicacao OK!
Digite na Entrada a mensagem para o Servidor!
Cliente se desconectou do Servidor!
Cliente finalizado!
```

Terminal do Servidor:

```
*v*v*v* CONSOLE DO SERVIDOR *v*v*v*
Servidor iniciado e escutando a porta 3334
Cliente IP 127.0.0.1 conectado ao Servidor pela porta 3334
Chegou do Cliente: olá
Ecoou ao Cliente: olá
Chegou do Cliente: como você está?
Ecoou ao Cliente: como você está?
Chegou do Cliente: teste
Ecoou ao Cliente: teste
Cliente se desconectou do Servidor!
Servidor finalizado!
```

Explicação detalhada:

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

1. **Cliente inicia handshake**:
    - Cliente envia SYN para porta 3334
    - 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 "sair"**:
   - Comando especial detectado pelo servidor no `if(!msg.equalsIgnoreCase("sair))"`
   - Servidor envia mensagem de confirmação: "Conexão encerrada."
   - Loops em ambos os lados param de executar
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:

O documento afirma explicitamente: "Essas aplicações não permitem conexões ao servidor por mais de um cliente por vez;". O código do servidor é single-threaded. Após aceitar o Cliente 1, o servidor entra no método conversaComCliente() e permanece preso no loop while(entrada.hasNextLine()), dedicando-se exclusivamente a esse cliente. A execução do programa nunca retorna à chamada servidor.accept() para aceitar novas conexões. Portanto, o Cliente 2 não tem como ser aceito pelo servidor.

f) Encerrar segunda instância e voltar à primeira

O que ocorre:

Como explicado no item (e), o Cliente 2 nunca estabeleceu uma conexão com o servidor. Sua execução (e falha) foi um evento totalmente independente que não afetou a comunicação já estabelecida entre o Servidor e o Cliente 1.

---

g) Executar segunda instância do servidor

O que ocorre:

A primeira instância do servidor, ao executar servidor = new ServerSocket(PORTA), "vincula" (bind) o programa à porta 3334. Um sistema operacional não permite que dois aplicativos diferentes escutem na mesma porta ao mesmo tempo. Quando o segundo servidor tenta fazer o bind na mesma porta 3334, o sistema operacional nega a solicitação, lançando uma java.net.BindException, que é capturada e impressa no console.
