Exercício 1 - SimpleClientServer

Autor: Felipe Fazio da Costa.

RA: 23.00055-4.

Análise do Projeto SimpleClientServer

a. Criação do Projeto

Status: Projeto criado com as classes:

- `SimpleServerTest.java`
- `SimpleClientTest.java`

Descrição das Classes

SimpleServerTest.java

Classe que implementa um servidor TCP que:

- Escuta conexões na porta 12345
- Aceita conexão de um cliente por vez
- Recebe mensagens do cliente
- Envia resposta confirmando o recebimento
- Encerra quando recebe a mensagem "sair"

SimpleClientTest.java

Classe que implementa um cliente TCP que:

- Conecta ao servidor em localhost:12345
- Solicita mensagens ao usuário via `JOptionPane`
- Envia mensagens ao servidor
- Recebe e exibe respostas do servidor
- Encerra quando o usuário digita "sair"

---

b. Executar somente SimpleClientTest.java

O que ocorre:

ERRO DE CONEXÃO

Registro do erro:

```
java.net.ConnectException: Connection refused: connect
```

ou

```
Servidor não encontrado: Connection refused: connect
```

Explicação detalhada:

1. **Tentativa de Conexão:** O cliente tenta estabelecer uma conexão socket com o servidor no endereço `localhost` (127.0.0.1) na porta 12345.
2. **Falha na Conexão:** Como o servidor não está em execução, não há nenhum processo escutando na porta 12345.
3. **Exceção Lançada:** O sistema operacional recusa a conexão, lançando uma `ConnectException`.
4. **Tratamento do Erro:** O bloco `catch` captura a exceção e:
   - Imprime mensagem de erro no console
   - Exibe um `JOptionPane` com mensagem "Não foi possível conectar ao servidor"
   - Encerra a aplicação após fechar os recursos no bloco `finally`
5. **Resultado:** O cliente não consegue se comunicar com o servidor e a aplicação é encerrada.

Motivo técnico: Para estabelecer uma conexão TCP, é necessário que haja um processo servidor escutando na porta especificada. Sem o servidor ativo, a tentativa de conexão é rejeitada pelo sistema operacional.

---

c. Executar somente SimpleServerTest.java

O que ocorre:

SERVIDOR AGUARDANDO CONEXÃO

Registro da saída:

```
Servidor aguardando conexão na porta 12345...
```

Explicação detalhada:

1. **Criação do ServerSocket:** O servidor cria um `ServerSocket` vinculado à porta 12345.
2. **Bind na Porta:** O sistema operacional reserva a porta 12345 para este processo.
3. **Modo de Escuta:** O servidor entra em modo de escuta (listen), aguardando conexões TCP.
4. **Método accept() Bloqueante:** A chamada `serverSocket.accept()` é **bloqueante** (blocking), ou seja:
   - A thread principal para neste ponto
   - Aguarda indefinidamente até que um cliente se conecte

- O servidor fica "pendurado" nesta linha de código

5. **Estado do Servidor:** O servidor está ativo e pronto para aceitar conexões, mas não faz nada até que um cliente se conecte.

6. **Comportamento Visual:** A aplicação parece "travada", mas na verdade está apenas aguardando. O programa não encerra, permanecendo em execução.

Nota técnica: O método `accept()` é uma chamada de sistema (system call) que coloca o processo em estado de espera. O scheduler do sistema operacional não aloca CPU para este processo até que uma conexão seja estabelecida.

---

d. Executar SimpleServerTest.java, depois SimpleClientTest.java com múltiplas mensagens

O que ocorre:

COMUNICAÇÃO ESTABELECIDA COM SUCESSO

Registro da execução:

Console do Servidor:

```
Servidor aguardando conexão na porta 12345...
Cliente conectado: 127.0.0.1
Mensagem recebida do cliente: Olá servidor!
Resposta enviada ao cliente: Servidor recebeu: Olá servidor!
Mensagem recebida do cliente: Como você está?
Resposta enviada ao cliente: Servidor recebeu: Como você está?
Mensagem recebida do cliente: Esta é minha última mensagem
Resposta enviada ao cliente: Servidor recebeu: Esta é minha última mensagem
Mensagem recebida do cliente: sair
Resposta enviada ao cliente: Servidor recebeu: sair
Cliente solicitou encerramento.
Servidor encerrado.
```

Console do Cliente:

```
Conectado ao servidor.
Enviando mensagem: Olá servidor!
Resposta do servidor: Servidor recebeu: Olá servidor!
Enviando mensagem: Como você está?
Resposta do servidor: Servidor recebeu: Como você está?
Enviando mensagem: Esta é minha última mensagem
Resposta do servidor: Servidor recebeu: Esta é minha última mensagem
Enviando mensagem: sair
Resposta do servidor: Servidor recebeu: sair
```

```
Encerrando cliente.
Cliente encerrado.
```

Explicação detalhada:

1. **Estabelecimento da Conexão (Three-Way Handshake TCP):**
    - Cliente envia SYN ao servidor
    - Servidor responde com SYN-ACK
    - Cliente confirma com ACK
    - Conexão TCP estabelecida
2. **Aceitação pelo Servidor:**
    - O método `accept()` retorna um objeto `Socket` representando a conexão
    - Servidor exibe o IP do cliente conectado (127.0.0.1)
3. **Criação dos Streams de I/O:**
    - **Servidor:** cria `BufferedReader` (entrada) e `PrintWriter` (saída)
    - **Cliente:** cria os mesmos streams na direção oposta
4. **Loop de Comunicação:**
    - **Cliente:** exibe `JOptionPane` solicitando mensagem
    - **Cliente:** envia mensagem via `out.println()`
    - **Servidor:** recebe via `in.readLine()` (bloqueante até receber linha completa)
    - **Servidor:** imprime mensagem recebida no console
    - **Servidor:** envia resposta de confirmação via `out.println()`
    - **Cliente:** recebe resposta via `in.readLine()` (bloqueante)
    - **Cliente:** exibe resposta em `JOptionPane`
5. **Repetição do Ciclo:**
    - Para cada mensagem digitada, o ciclo se repete
    - A comunicação é **síncrona** e **sequencial**
6. **Encerramento da Conexão:**
    - Quando usuário digita "sair" ou cancela o `JOptionPane`
    - Cliente envia "sair" ao servidor
    - Servidor detecta "sair", sai do loop e fecha recursos
    - Cliente recebe última resposta e encerra
    - Ambos executam o bloco `finally` para limpar recursos
7. **Limpeza de Recursos:**
    - Fechamento dos streams (`in`, `out`)
    - Fechamento dos sockets
    - Servidor fecha também o `ServerSocket`

Protocolo de Comunicação:

- **Formato:** Texto puro (String) com delimitador de linha (`\n`)
- **Padrão:** Request-Response (cliente solicita, servidor responde)
- **Transporte:** TCP (confiável, orientado a conexão, ordem garantida)

---

e. Duas instâncias de SimpleClientTest.java conectadas simultaneamente

O que ocorre:

PRIMEIRA INSTÂNCIA CONECTA, SEGUNDA INSTÂNCIA FICA BLOQUEADA

Registro da execução:

Servidor (console):

```
Servidor aguardando conexão na porta 12345...
Cliente conectado: 127.0.0.1
Mensagem recebida do cliente: Mensagem da primeira instância
Resposta enviada ao cliente: Servidor recebeu: Mensagem da primeira instância
```

Cliente 1 (primeiro a conectar):

```
Conectado ao servidor.
Enviando mensagem: Mensagem da primeira instância
Resposta do servidor: Servidor recebeu: Mensagem da primeira instância
```

(Aguardando nova entrada do usuário)

Cliente 2 (segundo a conectar):

```
(Aparentemente travado, sem mensagem no console)
```

(A janela JOptionPane NÃO aparece)

Explicação detalhada:

1. **Limitação do Servidor:**
    - O servidor chama `accept()` apenas **UMA VEZ**
    - Após aceitar a primeira conexão, entra no loop de comunicação
    - Não há código para aceitar múltiplas conexões simultâneas
2. **Comportamento da Segunda Instância do Cliente:**

- Tenta estabelecer conexão via `new Socket("localhost", 12345)`
- A solicitação TCP SYN é enviada ao servidor
- O sistema operacional **enfileira** a conexão pendente (backlog queue)
- O método `Socket()` fica **bloqueado** aguardando o servidor aceitar
- Como o servidor nunca chama `accept()` novamente, a conexão fica pendente

3. **Estado da Primeira Instância:**

   - Continua funcionando normalmente
   - Pode enviar e receber mensagens
   - Servidor processa suas mensagens sem problemas

4. **Simultaneidade de Digitação:**

   - **Cliente 1:** pode digitar e enviar mensagens livremente
   - **Cliente 2:** completamente bloqueado na tentativa de conexão
   - As mensagens do Cliente 1 são processadas normalmente pelo servidor

5. **Queue de Conexões Pendentes:**

   - O TCP mantém uma fila de conexões pendentes (backlog)
   - O tamanho padrão desta fila depende do sistema operacional
   - Conexões nesta fila aguardam um `accept()` do servidor
   - Se a fila encher, novas conexões serão rejeitadas

6. **Problema Arquitetural:**

   - O servidor é **single-threaded** e **single-client**
   - Arquitetura inadequada para múltiplos clientes simultâneos
   - Soluções possíveis:
     - **Multi-threading:** criar uma thread para cada cliente
     - **Thread Pool:** pool de threads para gerenciar clientes
     - **NIO (Non-blocking I/O):** multiplexação de I/O com Selector
     - **Frameworks:** usar frameworks como Netty ou Java EE

Comportamento técnico do bloqueio:

```
// Cliente 2 fica bloqueado aqui:
socket = new Socket("localhost", 12345); // <- BLOQUEADO
```

---

f. Encerrar segunda instância e continuar com a primeira

O que ocorre:

PRIMEIRA INSTÂNCIA CONTINUA FUNCIONANDO NORMALMENTE

Registro da execução:

Cliente 2 (ao ser encerrado):

```
(Processo encerrado forçadamente ou timeout)
```

Cliente 1 (continua funcionando):

```
Enviando mensagem: Nova mensagem após encerrar Cliente 2
Resposta do servidor: Servidor recebeu: Nova mensagem após encerrar Cliente 2
Enviando mensagem: Tudo funcionando perfeitamente
Resposta do servidor: Servidor recebeu: Tudo funcionando perfeitamente
```

Servidor (console):

```
Mensagem recebida do cliente: Nova mensagem após encerrar Cliente 2
Resposta enviada ao cliente: Servidor recebeu: Nova mensagem após encerrar Cliente
2
Mensagem recebida do cliente: Tudo funcionando perfeitamente
Resposta enviada ao cliente: Servidor recebeu: Tudo funcionando perfeitamente
```

Explicação detalhada:

1. **Encerramento do Cliente 2:**
    - Se encerrado forçadamente (Ctrl+C, kill, fechar IDE):
        - O sistema operacional fecha o socket pendente
        - Remove a conexão da fila de backlog
        - Libera recursos do processo
    - Se houve timeout (dependendo do SO):
        - Lança `ConnectException` ou `SocketTimeoutException`
        - Cliente 2 encerra com erro
2. **Independência das Conexões:**
    - A conexão do Cliente 1 é **independente** do Cliente 2
    - O socket do Cliente 1 permanece válido e ativo
    - Streams de I/O (BufferedReader, PrintWriter) continuam operacionais
3. **Servidor Não Afetado:**
    - Servidor está em loop processando mensagens do Cliente 1
    - Não tem conhecimento da existência ou encerramento do Cliente 2
    - O Cliente 2 nunca foi aceito pelo servidor, então não há conexão a ser quebrada
4. **Funcionamento Contínuo:**

- Cliente 1 pode continuar enviando mensagens indefinidamente
- Cada mensagem é processada normalmente
- O ciclo request-response continua funcionando

5. **Sem Impacto na Fila de Backlog:**

- Com o encerramento do Cliente 2, a fila de conexões pendentes fica vazia
- Se um novo cliente tentar conectar, será enfileirado da mesma forma
- Mesma situação se repetirá (bloqueio esperando `accept()`)

Importante: O Cliente 1 não tem conhecimento da existência de outros clientes (conectados ou tentando conectar). A comunicação é exclusiva entre Cliente 1 e Servidor.

---

g. Executar segunda instância de SimpleServerTest.java

O que ocorre:

ERRO: PORTA JÁ EM USO

Registro da execução:

Servidor 1 (já em execução):

```
Servidor aguardando conexão na porta 12345...
Cliente conectado: 127.0.0.1
(Continuando normalmente...)
```

Servidor 2 (nova instância):

```
Erro no servidor: Address already in use: JVM_Bind
java.net.BindException: Address already in use: JVM_Bind
    at java.net.DualStackPlainSocketImpl.bind0(Native Method)
    at
java.net.DualStackPlainSocketImpl.socketBind(DualStackPlainSocketImpl.java:106)
    at java.net.AbstractPlainSocketImpl.bind(AbstractPlainSocketImpl.java:387)
    at java.net.PlainSocketImpl.bind(PlainSocketImpl.java:190)
    at java.net.ServerSocket.bind(ServerSocket.java:375)
    at java.net.ServerSocket.<init>(ServerSocket.java:237)
    at SimpleServerTest.main(SimpleServerTest.java:11)
Servidor encerrado.
```

Explicação detalhada:

1. **Conceito de Porta TCP:**

- Portas TCP são identificadores numéricos (0-65535)
- Usadas para multiplexar conexões em um único endereço IP

- Apenas **UM processo** pode escutar (bind) em uma porta específica por vez

2. **Tentativa de Bind do Servidor 2:**

```
serverSocket = new ServerSocket(12345); // <- ERRO AQUI
```

3. **Verificação do Sistema Operacional:**

- SO mantém uma tabela de portas em uso
- Ao tentar criar ServerSocket na porta 12345:
  - SO verifica se a porta já está ocupada
  - Detecta que Servidor 1 está escutando nesta porta
  - **REJEITA** a operação

4. **Exceção Lançada:**

- `BindException: Address already in use`
- Indica que a combinação IP:Porta já está em uso
- Mensagem específica: `JVM_Bind` (binding da JVM)

5. **Motivo da Restrição:**

- **Ambiguidade de Roteamento:** Se dois processos escutassem na mesma porta, o SO não saberia para qual entregar as conexões
- **Segurança:** Previne que processos maliciosos "roubem" conexões de serviços legítimos
- **Consistência:** Garante que cada serviço tenha um identificador único

6. **Tratamento pelo Código:**

- Bloco `catch (IOException e)` captura a exceção
- Imprime stack trace completo
- Bloco `finally` executa limpeza (embora serverSocket seja null)
- Processo encerra

7. **Servidor 1 Não Afetado:**

- Continua operando normalmente
- Não recebe notificação da tentativa falhada
- Mantém controle exclusivo da porta 12345

Soluções possíveis:

Para permitir múltiplos servidores:

1. **Portas Diferentes:**

```
serverSocket = new ServerSocket(12346); // Servidor 2
serverSocket = new ServerSocket(12347); // Servidor 3
```

2. **SO_REUSEADDR (com cuidado):**

```
serverSocket = new ServerSocket();
serverSocket.setReuseAddress(true);
serverSocket.bind(new InetSocketAddress(12345));
```

   - Permite reutilização após TIME_WAIT
   - NÃO permite múltiplos servidores simultâneos na mesma porta

3. **Load Balancer:**

   - Usar um balanceador de carga externo
   - Nginx, HAProxy, ou cloud load balancers
   - Distribuir conexões entre múltiplas instâncias do servidor

Estado do Sistema Operacional:

```
netstat -ano | findstr :12345   (Windows)
```

Mostraria:

```
TCP    0.0.0.0:12345    0.0.0.0:0    LISTENING    <PID_Servidor1>
```
