Exercício 7 - Projeto MultiClientServer

Objetivo

Criar e testar um sistema cliente-servidor que suporta múltiplos clientes simultâneos usando sockets TCP e threads.

---

a) Criar o Projeto ProjetoMultiClientServer

Projeto Criado

Foram criadas três classes no pacote `Ex5e6`:

1. **`ClientSocket.java`** - Classe auxiliar que encapsula a comunicação via Socket
2. **`Server.java`** - Servidor que aceita múltiplas conexões simultâneas
3. **`Client.java`** - Cliente que se conecta ao servidor e envia mensagens

Estrutura do projeto:

```
Ex5e6/
├── ClientSocket.java
├── Server.java
└── Client.java
```

Características principais:

- Servidor escuta na porta **4000**
- Endereço local: **127.0.0.1** (localhost)
- Suporte a múltiplos clientes através de threads

---

b) Executar somente a classe `Client.java`

Resultado Observado

```
====== Console do Cliente ======
Erro ao iniciar o cliente: Connection refused: connect
Cliente finalizado!
```

Explicação Detalhada

O que aconteceu:

O cliente tentou se conectar ao servidor no endereço `127.0.0.1:4000`, porém **o servidor não estava em execução**. Isso resulta em uma exceção do tipo `ConnectException` com a mensagem "Connection refused:

connect".

Análise do código:

```
public void start() throws IOException{
    clientSocket = new Socket(Server.ADDRESS, Server.PORT); //  Falha aqui
    saida = new PrintWriter(clientSocket.getOutputStream(), true);
    // ...
}
```

Por que ocorre:

1. O construtor `new Socket(Server.ADDRESS, Server.PORT)` tenta estabelecer uma conexão TCP com o servidor
2. O sistema operacional verifica se existe algum processo escutando na porta 4000
3. Como **não há servidor ativo**, o SO responde com "Connection refused"
4. A exceção é capturada no bloco `catch` do `main()`
5. A mensagem de erro é impressa e o programa termina

Conceito de Rede:

- **TCP** requer um handshake (SYN → SYN-ACK → ACK) para estabelecer conexão
- Sem um servidor escutando, o handshake falha imediatamente
- O erro "Connection refused" é uma resposta ativa do SO (diferente de timeout)

---

c) Executar a classe `Server.java`

Resultado Observado

```
====== CONSOLE DO SERVIDOR ======
Servidor iniciado na porta 4000
Aguardando conexão de um cliente!
```

Explicação Detalhada

O que aconteceu:

O servidor foi iniciado com sucesso e entrou em modo de **escuta ativa** (listening), aguardando que clientes se conectem na porta 4000.

Análise do código:

```
public void start() throws IOException{
    serverSocket = new ServerSocket(PORT); // Abre a porta 4000
    System.out.println("Servidor iniciado na porta " + PORT);
    clientConnectionLoop(); // Entra em loop infinito
```

```
    do {
        ClientSocket clientSocket = new ClientSocket(serverSocket.accept()); //
⏸ Bloqueia aqui
        new Thread(() -> clientMessageLoop(clientSocket)).start();
    } while (true);
}
```

Funcionamento:

1. `new ServerSocket(PORT)` cria um socket que escuta na porta 4000
2. O método `serverSocket.accept()` é **bloqueante** (blocking)
3. O programa fica "parado" nessa linha aguardando conexões
4. Quando um cliente se conecta, `accept()` retorna um novo `Socket`
5. Uma thread é criada para gerenciar esse cliente específico
6. O servidor volta a aguardar novas conexões (loop infinito)

Estado do servidor:

- **Socket state:** LISTENING
- **Port:** 4000 (TCP)
- **Address:** 0.0.0.0 (aceita conexões de qualquer interface)
- **Connections:** 0 (aguardando primeira conexão)

** Observação importante:**

O servidor continuará rodando indefinidamente até ser interrompido manualmente (Ctrl+C ou encerrar o processo).

---

d) Executar a primeira instância do `Client.java` (com servidor ativo)

Resultado Observado

Console do Cliente:

```
====== Console do Cliente ======
Cliente 127.0.0.1:4000 conectado ao servidor!
Aguardando a mensagem de uma mensagem!
Digite uma mensagem (ou <sair> para finalizar): oi
Digite uma mensagem (ou <sair> para finalizar): tchau
Digite uma mensagem (ou <sair> para finalizar): <sair>
Cliente finalizado!

Process finished with exit code 0
```

Console do Servidor:

```
Cliente /127.0.0.1:xxxxx se conectou!
Mensagem recebida do cliente /127.0.0.1:xxxxx: oi
Mensagem recebida do cliente /127.0.0.1:xxxxx: tchau
Aguardando conexão de um cliente!
```

Explicação Detalhada

Sequência de eventos:

1. Conexão estabelecida

```
Cliente → SYN → Servidor
Cliente ← SYN-ACK ← Servidor
Cliente → ACK → Servidor
Conexão TCP estabelecida
```

O cliente se conecta com sucesso ao servidor na porta 4000.

2. Criação da Thread no Servidor

Quando `accept()` retorna, o servidor:

- Cria um objeto `ClientSocket` encapsulando a conexão
- Lança uma nova thread: `new Thread(() -> clientMessageLoop(clientSocket)).start()`
- Esta thread fica dedicada a esse cliente específico
- O servidor volta a escutar novas conexões (paralelamente)

3. Troca de mensagens

Mensagem "oi":

- Cliente: envia via `saida.println("oi")`
- Servidor: recebe via `entrada.readLine()` na thread do cliente
- Servidor: imprime "Mensagem recebida do cliente /127.0.0.1:xxxxx: oi"

Mensagem "tchau":

- Mesmo processo se repete

4. Encerramento da conexão

Quando o cliente digita `<sair>`:

- O loop `while(!msg.equals("<sair>"))` do cliente termina
- A conexão é fechada
- No servidor, `entrada.readLine()` retorna `null` ou a string ""
- A thread do cliente executa `clientSocket.close()` no bloco `finally`

- O servidor continua aguardando novas conexões

Estado final:

- Cliente: encerrado (exit code 0)
- Servidor: continua rodando, aguardando novas conexões
- Thread do cliente no servidor: finalizada

---

e) Executar novamente o `Client.java` SEM sair

Resultado Observado

Console do Cliente:

```
====== Console do Cliente ======
Cliente 127.0.0.1:4000 conectado ao servidor!
Aguardando a mensagem de uma mensagem!
Digite uma mensagem (ou <sair> para finalizar): olá servidor
Digite uma mensagem (ou <sair> para finalizar): ainda estou aqui
Digite uma mensagem (ou <sair> para finalizar): [aguardando mais mensagens...]
```

Console do Servidor:

```
Cliente /127.0.0.1:yyyyy se conectou!
Mensagem recebida do cliente /127.0.0.1:yyyyy: olá servidor
Mensagem recebida do cliente /127.0.0.1:yyyyy: ainda estou aqui
Aguardando conexão de um cliente!
```

Explicação Detalhada

O que aconteceu:

O servidor continua aguardando a conexão de novos clientes **enquanto mantém a conexão ativa** com o cliente atual.

Conceitos importantes:

1. Multithreading no Servidor

```
Thread Principal (main)          Thread do Cliente 1
      |                                  |
      |------ accept() [BLOQUEANDO] -----+------ readLine() [BLOQUEANDO]
      |                                  |
      |                                  |------ processa "olá servidor"
      |                                  |
      |                                  |------ readLine() [BLOQUEANDO]
```

```
        |                                   |
        |                                   |------ processa "ainda estou aqui"
        |                                   |
        |                                   |------ readLine() [BLOQUEANDO]
        v                                   v
```

- A **thread principal** aguarda novas conexões
- A **thread do cliente** aguarda novas mensagens desse cliente
- Ambas executam **simultaneamente** (paralelismo)

2. Conexão Persistente

- A conexão TCP permanece **aberta** enquanto o cliente não digitar `<sair>`
- O socket mantém um buffer de entrada/saída
- Não há timeout configurado (conexão infinita até sinal de encerramento)

3. Estado do Sistema

```
Servidor (porta 4000)
│
├── Thread Principal: accept() aguardando na porta 4000
│
└── Thread Cliente 1: readLine() aguardando mensagens
    └── Socket: 127.0.0.1:yyyyy ↔ 127.0.0.1:4000
```

Benefício dessa arquitetura:

O servidor pode atender **múltiplos clientes simultaneamente** sem bloquear uns aos outros, pois cada cliente tem sua própria thread.

---

f) Executar a segunda instância do `Client.java` (múltiplos clientes)

Resultado Observado

Console do Cliente 2:

```
====== Console do Cliente ======
Cliente 127.0.0.1:4000 conectado ao servidor!
Aguardando a mensagem de uma mensagem!
Digite uma mensagem (ou <sair> para finalizar): sou o cliente 2
Digite uma mensagem (ou <sair> para finalizar): conexão simultânea
Digite uma mensagem (ou <sair> para finalizar): [aguardando...]
```

Console do Servidor:

```
Cliente /127.0.0.1:zzzzz se conectou!
Mensagem recebida do cliente /127.0.0.1:yyyyy: ainda estou aqui
Mensagem recebida do cliente /127.0.0.1:zzzzz: sou o cliente 2
Mensagem recebida do cliente /127.0.0.1:zzzzz: conexão simultânea
Aguardando conexão de um cliente!
```

Explicação Detalhada

O que aconteceu:

O segundo cliente se conecta ao servidor **enquanto o primeiro ainda está conectado**. O servidor cria uma **thread separada** para cada cliente, permitindo comunicação simultânea e independente.

Arquitetura Multithreading:

```
Servidor (porta 4000)
│
├── Thread Principal: accept() aguardando novas conexões
│
├── Thread Cliente 1 (porta efêmera yyyyy)
│   ├── Socket: 127.0.0.1:yyyyy ↔ 127.0.0.1:4000
│   └── readLine() aguardando mensagens do Cliente 1
│
└── Thread Cliente 2 (porta efêmera zzzzz)
    ├── Socket: 127.0.0.1:zzzzz ↔ 127.0.0.1:4000
    └── readLine() aguardando mensagens do Cliente 2
```

Funcionamento simultâneo:

1. **Cliente 1** envia "ainda estou aqui" → Thread 1 processa
2. **Cliente 2** envia "sou o cliente 2" → Thread 2 processa
3. **Cliente 2** envia "conexão simultânea" → Thread 2 processa
4. Ambos clientes podem enviar mensagens **ao mesmo tempo** sem interferência

Identificação dos clientes:

O servidor identifica cada cliente pelo endereço do socket:

- Cliente 1: `/127.0.0.1:yyyyy`
- Cliente 2: `/127.0.0.1:zzzzz`

Onde `yyyyy` e `zzzzz` são portas **efêmeras** (aleatórias entre 49152-65535) atribuídas pelo SO.

Limite teórico:

Como mencionado no código:

```
// Existe um número máximo por conta da porta, 16^4 = 65536 clientes!
```

- Portas disponíveis: 0-65535
- Portas reservadas (0-1023): sistema
- Portas registradas (1024-49151): aplicações conhecidas
- Portas efêmeras (49152-65535): ~16.384 portas para clientes
- Limite real: menor entre portas disponíveis, threads do SO, e memória

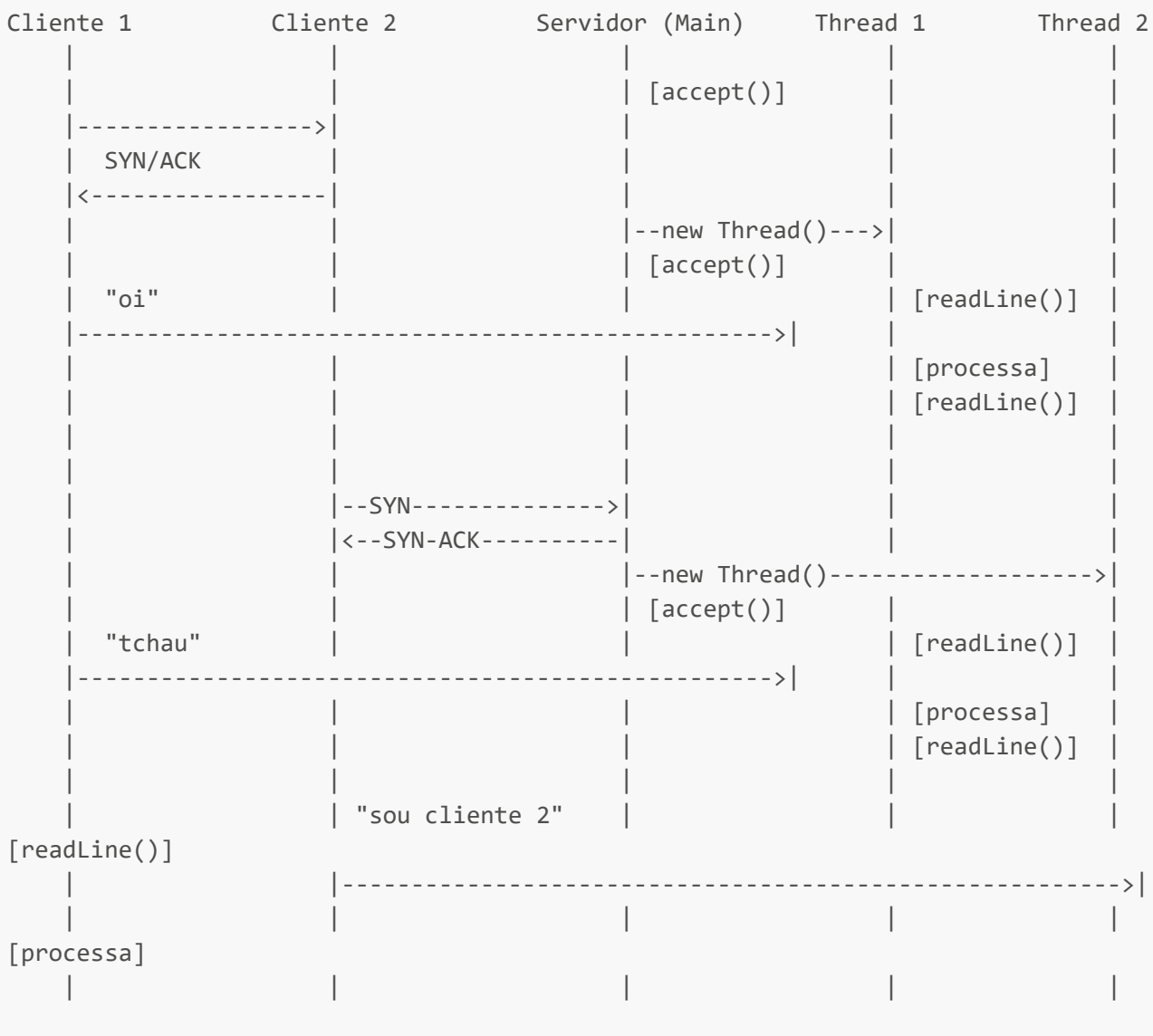
Conceito importante:

> **"O servidor continua aguardando a conexão de um cliente, porque cria uma thread separada para cada cliente que se conecta ao servidor e recebe as suas mensagens"**

Essa arquitetura permite:

- **Escalabilidade**: múltiplos clientes simultâneos
- **Isolamento**: falha em um cliente não afeta outros
- **Responsividade**: servidor não bloqueia em nenhum cliente
- **Concorrência**: processamento paralelo de mensagens

Diagrama de Sequência Completo



```
[readLine()]
    | <sair>          |                  |                |              |
    |-------------------------------------------------->|   |              |
    |                 |                  |                | [close()]    |
    |                 |                  |                | [finaliza]   |
    | [encerra]       |                  | [accept()]     |              |
[readLine()]
    |                 |                  |                |              |
```

Conclusões

Principais Aprendizados

1. **Arquitetura Cliente-Servidor**
   - Cliente inicia conexão; servidor aguarda passivamente
   - Protocolo TCP garante entrega confiável de dados
2. **Programação Concorrente**
   - Threads permitem atendimento simultâneo de múltiplos clientes
   - Cada cliente é isolado em sua própria thread
3. **Gestão de Conexões**
   - `ServerSocket.accept()` é bloqueante mas não impede novas conexões
   - Cada conexão resulta em um novo `Socket` independente
4. **Tratamento de Erros**
   - "Connection refused": servidor não está rodando
   - Importante tratar exceções de rede adequadamente
5. **Escalabilidade**
   - Modelo de thread-per-client é simples mas tem limites
   - Para milhares de conexões, modelos assíncronos são preferíveis (NIO, Netty)

Possíveis Melhorias

- Implementar um pool de threads para limitar o número de threads criadas
- Adicionar timeout para conexões inativas
- Implementar broadcast (servidor envia para todos os clientes)
- Adicionar autenticação de clientes
- Usar protocolo bidirecional (servidor também envia mensagens)

Documento elaborado para ECM251 - Aula 22 - Exercício 7