Relatório EP Distribuídos

Aluno: Francisco Oliveira Gomes Junior

Número USP: 12683190

Como compilar e executar o código

Ferramentas necessárias

`maven 3.5.2` ou superior.

`java 1.8.0` ou superior.

Para compilar e executar o sistema peer-to-peer, siga os seguintes passos:

Baixar as dependências do maven, executar comando `mvn clean install`.

Em seguida, para complilar o projeto, executar comando `mvn package`.

Para executar o projeto, rode o comando a baixo na raiz do diretório.

```
java -jar target/peerToPeer-0.0.1-SNAPSHOT.jar <endereco_ip>:<porta> <arquivo_com_vizinhos>
<diretorio_compartilhado>
```

Onde:

- `<endereço:porta>` é o endereço IP e porta que este peer usará (ex: 127.0.0.1:8080)
- `<arquivo_vizinhos>` é o caminho para um arquivo contendo a lista de vizinhos iniciais (um por linha, formato IP:porta)
- `<diretório_compartilhado>` é o caminho para o diretório que este peer compartilhará com a rede

Dificuldades e desafios encontrados

Organização do código

Conforme o desenvolvimento avançava, tornou-se evidente a complexidade inerente a sistemas peer-to-peer. Manter uma organização clara do código exigiu esforço constante, especialmente na separação de responsabilidades entre as classes. A medida que novas funcionalidades eram adicionadas, a interdependência entre os componentes aumentava, dificultando a manutenção e a evolução do sistema.

A complexidade de soluções P2P em grande escala ficou aparente ao implementar funcionalidades como descoberta de peers, atualização do status e lógica do clock. O desafio de coordenar múltiplos nós sem um ponto central de controle revelou a necessidade de um design cuidadoso das mensagens e protocolos de comunicação.

Lidar com perspectivas de execução

Um dos maiores desafios foi lidar simultaneamente com as perspectivas do peer que envia mensagens e do que as recebe. Essa dualidade exigiu que cada nó atuasse tanto como cliente quanto como servidor, implementando lógicas complexas de tratamento de mensagens e mantendo estados consistentes entre as diferentes threads de execução.

Testes unitários

A complexidade da estrutura final do código e a natureza distribuída do sistema tornaram desafiador a implementação de testes unitários abrangentes. O tempo necessário para criar um conjunto significativo de testes que cobrisse todas as interações possíveis entre peers seria considerável, o que levou a uma priorização de testes manuais em redes reais.

Paradigma

O sistema foi desenvolvido seguindo o paradigma orientado a objetos, que se mostrou particularmente adequado para modelar as entidades de uma rede peer-to-peer. Essa abordagem permitiu encapsular comportamentos específicos em classes bem definidas, facilitando a manutenção e extensão do código.

Na classe `No`, por exemplo, vemos um exemplo dos princípios OO:

```
@Data
public class No {
    private Rede rede;

    public No(String endereco, Integer porta, List<No> vizinhos) {
        this.rede = new Rede(endereco, porta, vizinhos);
    }

    // Construtores adicionais...
}
```

Esta classe encapsula todo o estado e comportamento relacionado a um nó da rede, usando composição para delegar funcionalidades de rede à classe `Rede`. O uso de anotações como `@Data` do Lombok segue o princípio DRY (Don't Repeat Yourself), evitando código boilerplate.

Divisão do programa em threads

A arquitetura do sistema emprega threads para lidar com diferentes aspectos da comunicação:

1. **Thread principal**: Responsável pela interface do usuário (classe `InterfaceUsuario`)
2. **Thread de escuta**: Criada na classe `Rede` para aceitar conexões de entrada:

```
private void threadEscuta() {
    new Thread(() -> {
        while (running) {
            try {
                Socket novoSocket = serverSocket.accept();
                new ThreadComunicacao(novoSocket, new No(this.enderecoIP, this.porta),
                    this.vizinhos, this.caixaDeMensagens).run();
            } catch (IOException e) {
                if (!running) {
                    System.out.println(SOCKET_ENCERRADO);
                } else {
                    System.err.println(ERRO_ACEITAR_CONECAO + e.getMessage());
                }
            }
        }
    }).start();
}
```

3. **Threads de comunicação**: Instâncias de `ThreadComunicacao` criadas para cada conexão estabelecida, como mostrado no trecho acima.

Esta divisão permite que o sistema mantenha responsividade na interface enquanto processa múltiplas mensagens simultaneamente.

Operações bloqueantes vs não bloqueantes

O sistema utiliza predominantemente operações bloqueantes, como evidenciado no tratamento de sockets:

```
// Na classe ThreadComunicacao - operação bloqueante de leitura
public Mensagem receberMensagem() throws IOException {
    InputStream inputStream = this.socket.getInputStream();
    BufferedReader reader = new BufferedReader(new InputStreamReader(inputStream));
    String mensagemEmTexto = reader.readLine(); // Bloqueia até receber dados
    ThreadComunicacaoUtil.exibirMensagem(mensagemEmTexto);
    return serializarMensagem(mensagemEmTexto);
}

// Na classe RedeService - operação bloqueante de escrita
public boolean enviarMensagem(Mensagem mensagemEnviada, CaixaDeMensagens caixaMensagens) {
    try (Socket socket = new Socket(mensagemEnviada.getDestino().getRede().getEnderecoIP(),
                                    mensagemEnviada.getDestino().getRede().getPorta())) {
        OutputStream output = socket.getOutputStream();
        PrintWriter writer = new PrintWriter(output, true);
        writer.println(mensagemEnviada.toString()); // Bloqueia até concluir escrita
        return true;
    } catch (IOException e) {
        return false;
    }
}
```

Essa escolha simplifica o fluxo de controle, mas limita a escalabilidade em cenários com muitos peers simultâneos.

Estruturas de dados utilizadas

HashMap

A classe `CaixaDeMensagens` utiliza `HashMap` para armazenar mensagens de forma eficiente:

```
@Data
public class CaixaDeMensagens {
    private HashMap<Integer, Mensagem> recebidas;
    private HashMap<Integer, Mensagem> enviadas;

    public CaixaDeMensagens() {
        this.recebidas = new HashMap<>();
        this.enviadas = new HashMap<>();
    }

    public void adicionarMensagemRecebida(Mensagem mensagemRecebida) {
        recebidas.put(quantidadeRecebidas() + 1, mensagemRecebida);
    }
}
```

Listas

Utilizadas extensivamente para armazenar vizinhos e argumentos de mensagens. Foram escolhidas porque:

```java
// Na classe NoUtil
public List<No> decoderListaVizinhos(String arquivoVizinhos) {
    List<No> listaVizinhos = new ArrayList<>();
    // ... leitura do arquivo e preenchimento da lista
    return listaVizinhos;
}

// Na classe Mensagem
private List<String> argumentos = new ArrayList<>();
```

ServerSocket/Socket: Para comunicação de rede, padrão em Java para operações TCP

```java
// Na classe Rede
private ServerSocket serverSocket;
```

File/FileReader: Para leitura de arquivos de configuração e listagem de arquivos compartilhados

```java
// Na classe NoUtil
try (BufferedReader br = new BufferedReader(new FileReader(arquivoVizinhos))) {
    // leitura do arquivo
}
```

Classes e pacotes

Pacote	Classes Principais	Responsabilidade
dsid.peerToPeer	Main	Ponto de entrada do sistema
dsid.peerToPeer.model	No, Rede	Modelagem dos peers e sua rede
dsid.peerToPeer.model.rede	Mensagem, CaixaDeMensagens, ThreadComunicacao	Comunicação entre peers
dsid.peerToPeer.controller	InterfaceUsuario	Interação com o usuário
dsid.peerToPeer.service	RedeService, MensagemService, CaixaMensagensService	Lógica de negócio
dsid.peerToPeer.utils	Constantes, NoUtil, MensagemUtil, ThreadComunicacaoUtil	Utilidades e helpers

Testes realizados

Os principais testes realizados incluíram:

1. Comunicação entre peers na mesma máquina (localhost)
2. Comunicação entre peers em diferentes máquinas na mesma rede local