UNIVERSIDADE FEDERAL DO ABC

SISTEMAS DISTRIBUÍDOS

Prof. Dr. Vladimir Emiliano Moreira Rocha

Projeto EP1

Gabriel Sponda Freitas Bettarello

11201932580

Santo André, 2023

Introdução

O sistema foi desenvolvido de acordo com os requisitos do projeto. Foram criadas as seguintes classes: Server.java e Peer.java, juntamente com a classe RequestImpl.java e a interface Request.java, que são utilizadas na comunicação por RMI.

Neste documento, você encontrará instruções completas sobre a operação adequada do programa, além de informações detalhadas sobre sua implementação. Os detalhes de implementação foram organizados em duas seções principais: Servidor e Peer. Dentro de cada seção, são abordados aspectos específicos, como a implementação do objeto remoto, o uso de threads e o funcionamento das transferências de arquivos grandes.

Link do vídeo:

https://youtu.be/xY7AC3ZhOtE

1. Instruções

Para operar o programa corretamente, siga estas etapas:

- Compile os arquivos da pasta “src”. Basta usar o terminal para acessar o diretório onde a pasta foi salva e então usar os comandos “javac model/Request.java”, “javac model/RequestImpl.java”, “javac server/Server.java” e “javac peer/Peer.java”.

- Inicialize o servidor a partir do terminal, acessando o diretório onde a pasta “src” foi salva e use o comando “java server.Server”. Um identificador será exibido no topo do console.

- Em seguida, inicialize um peer, também a partir do terminal já no diretório onde a pasta “src” foi salva e utilize o comando “java peer.Peer”. Outro identificador aparecerá no topo do console. Digite as informações solicitadas, como endereço IP, porta de escuta e diretório.

- Após inserir as informações requeridas, um menu será exibido na tela, e uma linha começando com "E: ". Digite o número correspondente à opção desejada.

- Se você acabou de iniciar o peer, é necessário selecionar a opção 1 para fazer a operação "JOIN". O peer será notificado de que ingressou na rede e o servidor imprimirá uma mensagem confirmando o cadastro do peer.

- Outros peers podem ser inicializados seguindo o mesmo procedimento.

- Após o "JOIN" ser bem-sucedido, a linha "E: " será exibida novamente, indicando que o programa está pronto para receber uma nova opção de menu. O menu completo não será exibido novamente para evitar poluição do console.

- Agora, se você escolher a opção 2, "SEARCH". Você será solicitado a informar o nome do arquivo desejado. O programa retornará a lista de peers que possuem o arquivo e voltará à seleção de opções do menu.

- A última opção disponível é a 3, chamada "DOWNLOAD". Essa opção deve ser selecionada após uma busca bem-sucedida, ou seja, quando uma lista de peers for retornada. Caso contrário, você será avisado e retornará ao menu para realizar uma nova busca.

- Se houver uma lista válida de peers, você poderá enviar uma solicitação de download, mas a escolha é feita aleatoriamente entre a lista de peers válidos. Nesse ponto, duas situações podem ocorrer: o download é aceito e você apenas precisa aguardar a confirmação de conclusão, ou o envio é recusado e uma mensagem de aviso será exibida. Nesse caso, você poderá retornar à seleção do menu e tentar o download novamente selecionando a opção 3 ou fazer uma nova busca usando a opção 2.

- Após concluir com êxito um download, o usuário retornará à seleção do menu, onde poderá realizar uma nova busca se desejar.

2. Servidor

A estrutura escolhida para o armazenamento de dados foi um ConcurrentHashMap, para evitar alguns problemas de concorrência. Nesse mapa, a chave é uma *String* que contém o IP e a porta do peer, enquanto o valor armazenado é um objeto da classe *Peer*. Essa classe será detalhada mais adiante e contém um atributo do tipo *List<String>* com os nomes dos arquivos.

Ao executar a classe *Server*, o método *main* é chamado, e em seguida, o método construtor é invocado, criando uma instância do servidor juntamente com uma instância do mapa de hash.

Posteriormente, é criada uma instância da classe *RequestImpl*, que será explicada ao longo do texto. Em seguida, é criado um registro RMI na porta 1099 do servidor, onde o registro é armazenado e associado a um nome específico (neste caso, *rmi://127.0.0.1/request*) ao objeto remoto.

Dessa forma, o servidor fica pronto para receber requisições dos peers através do objeto remoto por RMI.

Além do método construtor, existem dois outros métodos: o *getPeerMap()*, que retorna o mapa de hash completo, e os métodos *addPeer(Peer peer)*, que recebe um objeto da classe Peer como argumento e armazena os valores em uma nova chave, e o *updatePeer(Peer peer)*, que atualiza os valores em uma chave existente. A chave gerada é uma concatenação do IP, um traço e a porta de escuta do servidor, conforme mencionado anteriormente.

2.1. Objeto remoto

A comunicação entre os peers e o servidor central ocorre por meio de RMI. O objeto remoto é responsável por receber as requisições dos peers e processá-las. A interface *Request* define os métodos que podem ser chamados pelos objetos da classe Peer, enquanto a classe RequestImpl implementa a lógica desses métodos.

Durante a inicialização do servidor, como mencionado anteriormente, é criada uma instância do objeto remoto da classe *RequestImpl*. Essa instância recebe como parâmetro a instância do servidor, permitindo que os dados sejam armazenados nele. Os métodos disponíveis para serem chamados pelos peers são os seguintes:

- *join(Peer peer)*: Recebe uma instância de um peer e chama o método correspondente na classe *Server* para armazenar seus dados no mapa de hash. Esse método exibe uma mensagem no console do servidor e retorna a string "JOIN_OK" caso nenhum erro ocorra.

- *search(Peer p, String fileName)*: Realiza uma busca pelo nome do arquivo passado como parâmetro (*fileName*) entre os peers conectados à rede. Para isso, utiliza o método *getPeerMap()* e percorre os elementos do mapa de hash. Quando o nome do arquivo é encontrado, a chave do peer que possui esse arquivo é adicionada a uma lista de strings. Ao final da execução do método, exibe uma mensagem no console do servidor e retorna a lista de peers que possuem o arquivo.

- *update(Peer peer)*: Similar ao método join, este método chama o método do servidor para atualizar os valores de uma chave do mapa de hash. Também exibe uma mensagem no console do servidor com as informações atualizadas do peer. Embora chame o mesmo método do servidor, é usado em um contexto diferente do método join. Retorna a string "UPDATE_OK".

3. Peer

Vamos abordar a classe *Peer* de forma sequencial, acompanhando as declarações e chamadas de métodos relevantes que ocorrem durante a sua execução. Essa abordagem facilita o

entendimento dessas informações, bem como a organização do texto. Além disso, trataremos mais adiante da classe aninhada PortListenerThread.

Ao executar a classe *Peer*, o método *main* *[linha 256]*, o usuário é solicitado a inserir seu endereço IP, a porta que será usada para escutar requisições e o diretório onde estão armazenados os arquivos, que também será o local de salvamento dos arquivos baixados. Esses dados são lidos por um *Scanner* e passados como parâmetros para o método construtor da classe. É importante observar que o último parâmetro recebe o método *listFiles(String peerDirectory)* *[linha 175]*, que lê os nomes dos arquivos contidos no diretório informado e retorna uma lista de *File* contendo esses nomes.

Em seguida, o método *startListening* é chamado. Esse método utiliza uma thread e, portanto, não bloqueia a execução do programa. Ele será abordado mais detalhadamente na seção dedicada a threads.

Em seguida, o programa obtém o registro RMI do servidor e cria um objeto para referenciar o objeto remoto do servidor, buscando-o pelo nome associado a ele no registro. Com isso, o peer pode chamar métodos do objeto remoto para executar ações no servidor.

O menu interativo é apresentado e aguarda a seleção do usuário. Pressupõe-se que o usuário esteja familiarizado com o programa e saiba que a opção "JOIN" deve ser escolhida caso suas informações ainda não estejam cadastradas no servidor. Todas as opções do menu chamam um método correspondente que recebe como parâmetro a instância do peer e o objeto remoto. Abaixo estão os métodos correspondentes a cada opção:

- *menuJoin* *[linha 188]*: É um método simples que chama o método *join* do objeto remoto e armazena a resposta em uma string. Se a resposta for "JOIN_OK", conforme esperado, uma mensagem é exibida no console do peer.

- *menuSearch* *[linha 196]*: Recebe o nome do arquivo digitado pelo peer e chama o método *search* do objeto remoto. Como mencionado na seção *2.1*, esse método retorna uma lista de strings contendo informações dos peers que possuem o arquivo procurado. Essa lista é armazenada em uma variável de instância do peer que solicitou a busca e também é exibida na tela desse peer.

Observe que o nome do arquivo procurado e a lista de peers retornada pela busca são armazenados como atributos do peer. Essa implementação facilita algumas chamadas.

- *menuDownload [linha 215]*: Verifica se a variável de instância que deve conter a lista de peers com o arquivo buscado está vazia. Se estiver vazia, o usuário retorna à seleção do menu, o que indica que nenhum arquivo foi buscado ou nenhum peer possui o arquivo. Caso a lista de peers não esteja vazia, um peer é selecionado aleatoriamente (existe um trecho no código que permitia a seleção de qual peer baixar, mas foi colocado em comentário para facilitar o uso [linha 222]) e uma solicitação de download é enviada a ele por meio do método *sendDownloadRequest*, que será explicado em mais detalhes na seção de transferência de arquivos. Após a conclusão da transferência, o peer chama o método *update* do objeto remoto, uma mensagem é apresentada no console do servidor e, se o peer receber "UPDATE_OK", uma mensagem de confirmação é exibida no console do peer.

  Observe que nesse método há a criação de um objeto randômico. Ele é usado para simular algum tipo de falha na rede ou até mesmo a recusa do envio do arquivo por parte do peer solicitado. Na prática, essa falha ou recusa não ocorreria nesse momento de execução do programa, mas é apenas uma simulação. Nos casos em que ocorre uma falha ou recusa, o peer solicitante é informado de que a solicitação foi recusada e retorna à seleção do menu.

Além dos métodos do menu e dos métodos startListening e *sendDownloadRequest*, há também os métodos “getters”, que não requerem explicação adicional. Além disso, temos o método *receiveDownloadRequest*, que também será discutido mais detalhadamente na seção de transferência de arquivos.

3.1. Threads

Vamos começar tratando do método *startListening [linha 89]*. Primeiramente, é criado um socket do servidor na porta de escuta informada pelo peer. Em seguida, é criado um objeto da classe *Executors*, que utiliza a interface *ExecutorService*. A classe *Executors*, pertencente ao pacote *java.util.concurrent*, fornece métodos para gerenciar

threads. No caso, o método *newSingleThreadExecutor* é usado para criar uma instância que utilizará uma única thread para executar as tarefas sequencialmente.

Essa sequência de tarefas é enviada para a thread usando o método *submit* do objeto *ExecutorService*. Uma expressão lambda é passada como parâmetro para o método *submit* e contém um loop infinito.

Dentro do loop, a primeira instrução faz com que o socket do servidor criado anteriormente fique aguardando novas requisições de conexão. Quando uma nova conexão é reconhecida, um novo socket é criado em uma porta aleatória. É nesse momento que uma conexão TCP é estabelecida entre os peers. Na segunda instrução, esse novo socket é passado como parâmetro para criar uma instância da classe *PortListenerThread*. Assim que esse objeto é criado e executado pelo método *start*, a responsabilidade da requisição é transferida para ele, o loop reinicia e a porta de escuta fica livre para novas requisições. Isso garante que o peer esteja sempre disponível para receber requisições, e para cada requisição, uma nova instância da *PortListenerThread* é criada e dedicada a processar o pedido.

Agora vamos nos concentrar na classe *PortListenerThread* *[linha 63]* e como ela lida com as requisições. Quando uma instância dessa classe é criada e o método *start* é chamado, ela executa seu método *run* e chama o método *receiveDownloadRequest*. Após a execução desse método, não há mais referências para essa instância, e ela é removida da memória. Isso significa que, embora uma nova instância seja criada para cada requisição, o desempenho do programa não é impactado.

É importante observar que o parâmetro do diretório passado para a instância de *PortListenerThread* é obtido a partir da chamada do seu construtor pelo método *startListening*, que por sua vez recebe esse parâmetro na inicialização da classe *Peer*. No entanto, esse parâmetro não é compartilhado com o peer requisitante. Ele é utilizado apenas para que o programa possa buscar o nome do arquivo requisitado nesse diretório específico.

3.2. Transferências de arquivos

Para facilitar o entendimento, chamaremos de "cliente" o peer que faz a requisição e de "servidor" o peer que é requisitado. Existem dois métodos responsáveis pela transferência de arquivos: *sendDownloadRequest* *[linha 145]* (executado no lado do cliente) e *receiveDownloadRequest* *[linha 116]* (executado no lado do servidor). Ambos são executados simultaneamente após o estabelecimento da conexão.

Quando o cliente seleciona a opção de download no menu e chama o método *sendDownloadRequest*, é feita uma solicitação de conexão ao servidor, criando um novo socket em uma porta aleatória do cliente. O servidor recebe a requisição e atende por uma thread, como visto anteriormente. A partir deste ponto, trataremos do que acontece após a conexão já ter sido estabelecida.

Durante a execução desses métodos, instâncias de classes do pacote *java.io* são criadas para manipular a leitura, escrita e envio de dados durante a conexão. Vamos omitir os detalhes específicos desses objetos, pois são semelhantes aos apresentados no vídeo tutorial disponibilizado nas instruções do projeto. Além disso, buffers (arrays de bytes) com tamanho de 1024 também são utilizados.

No lado do servidor, o nome do arquivo desejado é lido e, em seguida, o arquivo é lido em bytes e armazenado. Em seguida, um loop é usado para enviar partes do arquivo em pacotes de 1024 bytes para o cliente. Isso é feito escrevendo esses dados no buffer e enviando-os.

No lado do cliente, ocorre um processo similar. O nome do arquivo é enviado para o servidor e, em seguida, um arquivo em bytes é escrito no diretório informado pelo cliente. Da mesma forma, é usado um loop para receber os bytes do servidor, armazená-los no buffer e escrevê-los no arquivo. O recebimento é feito em pacotes de 1024 bytes.

Dessa forma, quando o servidor envia o primeiro pacote, o cliente o recebe e começa a escrevê-lo no arquivo. Enquanto o cliente escreve os dados recebidos, o servidor já está preparando o próximo pacote para envio. Isso permite que o envio pelo servidor ocorra quase simultaneamente à escrita pelo cliente.

Essa implementação permite o envio de grandes arquivos, pois apenas pequenas partes do arquivo ocupam a memória de cada vez, ao invés de carregar um arquivo gigante por completo, o que seria inviável para alguns sistemas.

4. Considerações

O objetivo do projeto era implementar e compreender as comunicações realizadas por RMI e TCP, bem como entender a diferença entre elas. No entanto, é importante ressaltar que muitas possíveis falhas não foram abordadas neste projeto, como entradas incorretas dos usuários, falhas de comunicação e questões de segurança. Além disso, é importante destacar que esse modelo não seria adequado para uma rede de grande escala, já que não trata problemas de concorrência na comunicação entre peers e servidor, bem como problemas de sincronização. O objetivo futuro é desenvolver soluções que permitam a execução efetiva e remota desse modelo em uma escala maior, explorando novas tecnologias e aprendendo com elas.

Referências

Nenhuma parte deste código foi copiada indiscriminadamente (ou alterada) de outras fontes. Durante o desenvolvimento, foram consultados sites populares, como Stack Overflow, Geeks for Geeks e GitHub, bem como a documentação oficial do Java. O Chat GPT foi utilizado como uma ferramenta adicional para obter *insights* e sugestões ao longo do processo de desenvolvimento. O texto foi escrito originalmente pelo autor, e o Chat GPT foi utilizado para auxiliar na melhoria da clareza e da compreensão do texto.