Projeto Napster

Nome: Tiago Henrique Simionato Machado
RA: 11201810899
Email: tiago.simionato@aluno.ufabc.edu.br

27/07/2022

Indice

Link do vídeo de Funcionamento

link para o vídeo:

Funcionalidades Servidor

Threads

O Servidor possui 3 *threads* no total, uma principal, a *ServerAnswerThread* e a *AliveSender*, usadas para poder atender diversos *Peers* de uma vez e verificar se estão vivos. Mais informações sobre a função de cada *thread* nesta seção

Join_Ok

Quando um peer manda um **JOIN** para o Servidor, ele armazena em um ArrayList um registro para cada arquivo que o *Peer* possui. Os registros ficam salvos como 'Ip:PortaUDP:PortaTCP,nomeArquivo' e são salvos apenas caso o peer ainda não tenha pedido por join e não esteja vivo.

Em seguida cria a mensagem de **JOIN_OK** e manda para o *Peer*. A função `joinOk` está implementada entre as linhas `115` e `158` do arquivo `Servidor.java` e é chamada na linha `86`.

Leave_Ok

Quando o Servidor recebe uma requisição **LEAVE**, ele usa o Ip e porta udp do *peer* que mandou a requisição para iterar sobre o ArrayList com os registro e apaga todos os que contem aquele Ip:PortaUDP.

Em seguida envia uma mensagem de **LEAVE_OK**. A função `leaveOk` está implementada entre as linhas `165` e `178` do arquivo `Servidor.java` e é chamada na linha `89`.

Search

Quando o Servidor recebe uma requisição **SEARCH**, ele usa a string guardada na classe `Mensagem`, que representa o nome do arquivo a ser procurado, para comparar os nomes dos arquivos guardados no ArrayList do Servidor e faz uma mensagem com uma lista contendo o Ip:PortaTCP daqueles *Peer* que tem exatamente o mesmo nome de arquivo procurado.

Então ele cria a mensagem **SEARCH** de resposta e envia ao peer. A função `search` está implementada entre as linhas `187` e `214` do arquivo `Servidor.java`e é chamada na linha `93`.

Update_Ok

Quando o *Peer* envia uma requisição **UPDATE** para o Servidor, este adiciona um resgistro no ArrayList informando que o *Peer* agora possui o arquivo. Para o resgistro poder ser adicionado o servidor precisa procurar por um arquivo daquele peer para encontrar sua porta TCP.

Por fim uma resposta **UPDATE_OK** é enviada de volta ao *Peer*. A função `update` está implementada entre as linhas `222` e `244` do arquivo `Servidor.java`e é chamada na linha `97`.

Alive

Para enviar requisições **ALIVE** aos *Peers*, o Servidor usa uma thread separada que é executada a cada 30 segundos. Ela olha os registros do Servidor para descobrir os *Peers* que estão conectados, envia uma requisição **ALIVE** para cada um deles. Então a thread espera por 3 segundos para dar tempo dos *Peers* responderem. Cada mensagem **ALIVE_OK** recebida é tratada pela *thread ServerAnswerThread*, que informa ao Servidor que o *Peer* está vivo.

Por fim, a *thread* compara sua lista de *Peers* conectados com a lista do Servidor de *Peers* vivos e elimina dos registros do Servidor os *Peers* que não estão na lista de *Peers* vivos. Para continuar o loop a thread cria uma nova instância dela mesma.

A *thread* que cuida das requisições **ALIVE** está implmentada estre as linhas `247` e `329` do arquivo `Servidor.java`.

Inicialização do Servidor

Quando a classe `napster.Servidor` é iniciada, ela começa aguardando para receber o Ip do Servidor. É assumido que o endereço a ser digitado é `127.0.0.1`. Então o servidor entra em seu *loop* aguardando requisições e nenhuma entrada mais é recebida do usuário.

A inicialização do Servidor ocorre entre as linhas `27` e `34` de `Servidor.java`

Funcionalidades Peer

Threads

O *Peer* possui ao todo `7` *threads*: a principal, *PeerListenerThread*, *PeerAnswerThread*, *DownloadListener*, *FileSenderThread*, *DownloadThread* e *TimeoutTimer*. Mais informações sobre a função de cada *thread* nesta seção

Join

Quando o usuário informar que quer realizar um **JOIN**, o *Peer* criará uma lista com os arquivos que estão na sua pasta (informada no momento da inicialização do *Peer*). A função `join` recebe essa lista, cria uma mensagem com seu tipo de requisição e envia a mensagem através de UDP para o Servidor.

No caso do servidor não responder a mensagem, ela cria uma *thread* de timer que reenviará a mensagem.

A função `join` está nas linhas `160` a `171` e é chamada nas linhas `116` e `601`. Tudo no arquivo `Peer`.

Leave

Quando o usuário informar que quer realizar um **LEAVE**, o *Peer* simplesmente enviará uma mensagem **LEAVE** para o Servidor. Se o *Peer* não tiver pedido por **JOIN** antes nada acontece.

No caso do servidor não responder a mensagem, ela cria uma *thread* de timer que reenviará a mensagem.

A função `leave` está nas linhas `178` a `189` e é chamada nas linhas `140` e `604`. Tudo no arquivo `Peer.java`.

Update

Quando o *Peer* conseguir baixar um arquivo com sucesso ele automaticamente envia uma requisição **UPDATE** ao Servidor com o nome do arquivo salvo na classe `Mensagem`.

No caso do servidor não responder a mensagem, ela cria uma *thread* de timer que reenviará a mensagem.

A função `update` está nas linhas `196` a `211` e é chamada nas linhas `293` e `607`. Tudo no arquivo `Peer.java`.

Alive_Ok

Quando a *thread PeerAnswerThread* receber uma Mensagem do tipo **ALIVE**, ela chama o método do peer `alive` que apenas manda uma mensagem **ALIVE_OK** para o servidor.

Essa função está nas linhas `302` a `305` e é chamada na linha `436`. Tudo no arquivo `Peer.java`.

Search

Quando o usuário informar que quer realizar um **SEARCH**, o *Peer* perguntará pelo nome do arquivo a ser procurado, colocará a string do nome em uma `Mensagem` com requisição do tipo **SEARCH** e enviará a `Mensagem` para o Servidor. Se o *Peer* não tiver pedido por **JOIN** antes nada acontece.

No caso do servidor não responder a mensagem, ela cria uma *thread* de timer que reenviará a mensagem.

A função `update` está nas linhas `217` a `232` e é chamada nas linhas `123` e `610`. Tudo no arquivo `Peer.java`.

Caso não receber resposta

A classe `Peer` tem um atributo de *timer* (que é uma *thread*) para cada requisição que deve receber uma resposta. Quando uma função envia uma requisição, o *timer* da respectiva requisição é acionado. Se o *Peer* recebeu o ok, um *booleano* do *timer* é marcado como *true* e ele desativa. Senão o *timer* chama o método do *Peer* que envia a mensagem outra vez e o método por sua vez iniciará o *timer* novamente.

A *Thread* de timer está nas linhas `57` a `617` do arquivo `Peer.java`.

Download

EXPLICAR O DOWNLOAD (e ta faltando informar algumas linhas ai)

A função `download` está nas linhas `242` a `299` e é chamada na linha `57`. Tudo no arquivo `Peer.java`.

Download_Negado ou aprovado

Para o *Peer* negar ou aprovar o *download*, ele olhar em sua pasta de armazenamento (informada na inicialização do *Peer*) se o arquivo está la. Se não estiver nega o *download*. Se estiver, aprova.

Recebimento do arquivo

O *Peer* recebe o arquivo lendo bytes através de um *DataInputStream* que recebe os dados que vem do *Socket* TCP e escreve no arquivo criado usando um *FileOutputtStream* do Java. O recebimento acontece entre as linhas `267` e `294` do `Peer.java`.

Download_Negado

Quando o *download* é negado, é mostrado no console através de um print e nenhum arquivo é recebido, ou seja, nada é feito. Isso acontece no `else` da linha `294`. O outro *Peer* nega o envio se o arquivo não existir em sua pasta de armazenamento e isso pode ser visto na linha `501` de `Peer.java`.

Inicialização do Peer

Quando a classe `napster.Peer` é iniciada, ela aguarda o usuário digitar um enderaço de IP e depois uma porta. Se algum dos dois for inválido (porta já em uso, por exemplo), o programa irá pedir para informar os dois novamente. Quando o socket puder ser criado o programa então pedirá pela pasta de armazenamento do *Peer*.

Com esta inicialização terminada o programa fica em *loop* no menu interativo até que o programa seja encerrado pelo terminal.

Threads do projeto

Threads do servidor

Thread principal

A *thread* principal do Servidor começa perguntando pelo ip que o Servidor deverá ter e depois fica em loop esperando por requisições chegarem. Cada vez que uma requisição chega, a *thread ServerAnswerThread* é instanciada e trata a requisição devidamente.

Esta *thread* foi implementada nas linhas `17` a `66` do arquivo`Servidor.java`.

ServerAnswerThread

Usada para reponder devidamente cada requisição que o Servidor receber. Ela le o campo da Mensagem que diz o tipo de requisição recebida e chama sua função correspondente para responder a mensagem.

Ela está implementada nas linhas `69` a `244` do arquivo `Servidor.java`.

AliveSender

Por fim, a *thread AliveSender* fica repetidamente mandando requisições *Alive* para os peers e elimina os que não responderem.

Ela está implementada nas linhas `247` a `329` do arquivo `Servidor.java`.

Threads do peer

Thread principal

Esta é a *thread* que le o Ip, porta e pasta de arquivos do peer e fica em *loop* no menu interativo até que o programa seja encerrado.

Ela está implementada entre as linhas `27` e `374` do arquivo `Peer.java`.

PeerListenerThread

Esta *thread* é usada para o *Peer* ouvir requisições UDP. Ela é iniciada uma vez que o *Peer* pedir por um **JOIN** e é pausada caso o *Peer* peça por **LEAVE**. Ela fica em *loop* esperando por requisições chegarem e cria uma *thread* separada para tratar cada requisição que chegar.

Essa *thread* está implementada entre as linhas `374` e `398` do arquivo `Peer.java`.

PeerAnswerThread

Essa *thread* é usada para tratar cada requisição que o *Peer* receber. Ela le o tipo de requisição da mensagem e altera os atributos necessários na classe principal `Peer`.

A *thread* está implementada entre as linhas `401` e `443` do arquivo `Peer.java`.

DownloadListener

Essa *thread* é usada para esperar por pedidos de *download* vindos de outros *Peers*. Para cada pedido que chegar ela cria uma nova *thread FileSender* para enviar o arquivo para o respectivo *Peer* (caso o *download* não seja negado).

Esta *thread* está implementada entre as linhas `447` e `464` do arquivo `Peer.java`.

FileSenderThread

Thread usada para o envio de arquivos. Primeiro verifica se o *Peer* tem o arquivo. Se não tiver o *download* é negado e **DOWNLOAD_NEGADO** é enviado de volta ao *Peer*. Se o *Peer* aprovar o *download*, o arquivo é enviado através da conexão tcp estabelecida e então a conexão é fechada.

A *thread* está implementada entre as linhas `467` e `537` do arquivo `Peer.java`.

DownloadThread

Esta *Thread* é usada para o *Peer* poder fazer o *download* de um arquivo separado da *thread* principal, assim o usuário pode continuar usando o menu interativo enquanto o arquivo é baixado. A thread apenas chama a função de download implementada nas linhas `242` a `299` da classe `Peer`.

A *thread foi implementada entre as linhas `540` e `565` do arquivo `Peer.java`.

TimeoutTimer

Esta *thread* serve para reenviar requisições que o servidor eventualmente não responder. Ela espera por 4 segundo e então chama o método correspondente no *Peer* que reenvia a mensagem que não teve resposta. Há um booleano nela que sinaliza se a mensagem ainda deve ser reenviada depois dos 4 segundos de espera. Se não for a *thread* apena é interrompida.

A *thread* foi implementada entre as linhas `571` e `617` do arquivo `Peer.java`.

Implementação de arquivos gigantes

Para que seja possível baixar arquivos muito grandes, primeiramente o *Peer* fornecedor informa o tamanho do arquivo e em seguinda entra em um *loop while* que divide o arquivo em vários pedaços de 4 KB e é escrito pelo `DataOutPutStream` do *socket* aos poucos. O *Peer* receptor sabe quantas vezes precisa ler informações do *socket* pois antes é enviado através de um `long` e tamanho do arquivo. Quando todo o arquivo tiver sido lido a função `read` do `FileInputStream` retornará -1, saindo do *loop* e fechando a conexão.