Projeto Napster

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

30/07/2022

Indice

Link do vídeo de Funcionamento

link para o vídeo: https://youtu.be/C1WkW5vw724

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 se o peer já está na rede, nada é salvo. Se o peer não tem arquivos apenas é salvo um registro 'Ip:PortaUDP:PortaTCP,' para saber que o peer esta conectado.

Em seguida cria a mensagem de **JOIN_OK** e manda para o *Peer*. A função `joinOk`, que realiza as operações mencionadas acima, está implementada entre as linhas `116` e `165` do arquivo `Servidor.java` e é chamada na linha `87`, que é quando a thread de tratamento deve lidar com uma requisição **JOIN**.

Leave_Ok

Quando o Servidor recebe uma requisição **LEAVE**, ele usa o Ip e porta udp do *peer* (como uma string 'ip:PortaUdp') 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` realiza essa funcionalidade e está implementada entre as linhas `172` e `191` do arquivo `Servidor.java` e é chamada na linha `90`, que é quando a *thread* de tratamento recebe uma requisição **LEAVE**.

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 *Peers* que tem exatamente o mesmo nome de arquivo procurado.

Então ele cria a mensagem **SEARCH** de resposta e envia ao *Peer*. Essa funcionalidade essa implementada na função `search` que está entre as linhas `200` e `237` do arquivo `Servidor.javae` é chamada na linha `94`, que é quando a *thread* de tratamento recebe uma requisição **SEARCH**.

Update_Ok

Quando o *Peer* envia uma requisição **UPDATE** para o Servidor, este adiciona um resgistro ao final do *ArrayList* (seguindo o mesmo padrão de registro) 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` realiza essa funcionalidade e está implementada entre as linhas `245` e `272` do arquivo `Servidor.javae` é chamada na linha `98`, quando a *thread* de tratamento recebe uma requisição **UPDATE**.

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 (linhas `291` a `313` de `Servidor.java`), envia uma requisição **ALIVE** para cada um deles (linhas `319` a `328` de `Servidor.java`). Então a thread espera por 3 segundos para dar tempo dos *Peers* responderem. Cada mensagem **ALIVE_OK** recebida é tratada pela *thread ServerAnswerThread*, que adiciona um registro do ip:PortaUdp do *Peer* em um *array* do Servidor que contem os *Peers* vivos (linha `103`).

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 (linhas `337` a `358`). Para continuar o loop a *thread* limpa o *array* de *Peers* vivos do Servidor e cria uma nova instância dela mesma.

A *thread* que cuida das requisições **ALIVE** está implmentada estre as linhas `276` e `373` do arquivo `Servidor.java`.

Inicialização do Servidor

Quando a classe `napster.Servidor` é iniciada, ela começa aguardando para receber o Ip do Servidor (liha `31`). É assumido que o endereço a ser digitado é `127.0.0.1`. Então o servidor entra em seu *loop* aguardando requisições (linhas `38` a `49`) e nenhuma entrada mais é recebida do usuário.

A inicialização do Servidor ocorre entre as linhas `27` e `35` 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** (linha `79`), o *Peer* criará uma lista com os arquivos que estão na sua pasta (linhas `97` a `112`) (informada no momento da inicialização do *Peer* (linha `65`)). 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` que realiza a funcionaliade descrita está nas linhas `158` a `169` e é chamada nas linhas `114` (quando usuario pede join) e `601` (quando ocorre um timeout). Tudo no arquivo `Peer.java`.

Leave

Quando o usuário informar que quer realizar um **LEAVE** (linha `136`), 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 `176` a `187` e é chamada nas linhas `138` (usuario pediu) e `604` (timeout ocorreu). Tudo no arquivo `Peer.java`.

Update

Quando o *Peer* conseguir baixar um arquivo com sucesso ele automaticamente envia uma requisição **UPDATE** (linha `291`) 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 `194` a `209` e é chamada nas linhas `293` (download com sucesso) e `607` (timeout ocorreu). Tudo no arquivo `Peer.java`.

Alive_Ok

Quando a *thread PeerAnswerThread* receber uma Mensagem do tipo **ALIVE** (linha `431`), ela chama o método do peer `alive` que apenas manda uma mensagem **ALIVE_OK** para o servidor.

Essa função está nas linhas `300` a `303` e é chamada na linha `433`. Tudo no arquivo `Peer.java`.

Search

Quando o usuário informar que quer realizar um **SEARCH** (linha `117`), o *Peer* perguntará pelo nome do arquivo a ser procurado. A partir de então a função `search` é chamada e ela coloca o nome arquivo em uma `Mensagem` com requisição do tipo **SEARCH** e envia 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 `search` está nas linhas `215` a `230` e é chamada nas linhas `121` e `610`. Tudo no arquivo `Peer.java`.

Caso não receber resposta

A classe `Peer` tem alguns atributos de *timer* (que é uma *thread*), um para cada tipo de requisição que deve receber uma resposta (linhas `33` a `39`). 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 `576` a `613` do arquivo `Peer.java`.

Download

Primeiro é criado o socket tcp, juntamente com os canais para escrita e leitura de dados (linhas `242` a `248`). Atraves do socket é enviada uma mensagem contendo o nome do arquivo desejado (linhas `252` a `260`). Então é lido um inteiro que diz se o *download* foi negado ou aprovado. Se o download é negado, a função mostra na tela essa informação e não faz mais nada. Se o download é aprovado ele é recebido conforme descrito nesta seção

A função `download` está nas linhas `240` a `297` e é chamada na linha `554` na *thread* para *download*. 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*) e verifica se o arquivo está la. Se não estiver nega o *download*. Se estiver, ele aleatoriamente aprova ou nega a requisição (linha `498`).

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 novo arquivo criado usando um *FileOutputtStream* do Java. O arquivo é enviado/recebido usando um *loop while* (linhas `281` a `285`), pois não se sabe exatamente o tamanho do arquivo, ou seja, quantas iteralçoes são necessárias para enviar o arquivo inteiro. As leituras e escritas são funcionam pois sempre é lido/enviado a mesma quantidade de *bytes*. O recebimento acontece entre as linhas `265` e `291` 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 `292`. O outro *Peer* nega o envio se o arquivo não existir em sua pasta de armazenamento (ou com uma chance pequena aleatoria se existir) e isso pode ser visto na linha `498` 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. Isso pode ser visto entre `53` e `62`.Quando o socket puder ser criado o programa então pedirá pela pasta de armazenamento do *Peer* (`65`).

Com esta inicialização terminada o programa fica em *loop* no menu interativo até que o programa seja encerrado pelo terminal (`77` a `149`).

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 (`46`) e trata a requisição devidamente.

Esta *thread* foi implementada nas linhas `16` a `67` 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 `70` a `273` 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 `276` a `373` 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 `25` e `368` 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** (`87`) e é pausada caso o *Peer* peça por **LEAVE** (`142`). Ela fica em *loop* esperando por requisições chegarem e cria uma *thread* separada para tratar cada requisição que chegar (`378` a `390`).

Essa *thread* está implementada entre as linhas `371` e `395` 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 (`412`) e altera os atributos necessários na classe principal `Peer`, isto é, ela marca como `True` os respectivos timers do *Peer* (`413` a `426`) para informar que o ok foi recebido. No caso de receber **ALIVE** ela chama `aliveOk` para enviar a mensagem.

A *thread* está implementada entre as linhas `398` e `440` 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 `444` e `461` do arquivo `Peer.java`.

FileSenderThread

Thread usada para o envio de arquivos. Primeiro verifica se o *Peer* tem o arquivo (`493` a `497`). 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 (`507` a `526`) e então a conexão é fechada.

A *thread* está implementada entre as linhas `464` e `534` 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 `240` a `297` da classe `Peer`.

A *thread foi implementada entre as linhas `537` e `561` 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 `567` e `613` 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` o tamanho do arquivo. Quando todo o arquivo tiver sido lido a função `read` do `FileInputStream` retornará -1, saindo do *loop* e fechando a conexão.