FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D COIMBRA

Googol

Projeto de Sistemas Distribuídos

Projeto desenvolvido por:

André Moreira, nº2020239416

João Pinto, nº2020220907

Arquitetura do sistema

1- Ligações:

Ligação Interface -> server:

Esta conexão RMI permite que o cliente especifique as ações que deseja realizar e permite que o servidor saiba o que fazer com base nas escolhas do cliente.

Ligação server -> serverb:

Esta conexão RMI permite que o servidor realize pesquisas. Quando o *serverb* recebe uma chamada do *server*, ele inicia a conexão com os *Storage Barrels,* o que permite que a pesquisa solicitada pelo cliente seja realizada.

Ligação Storage_Barrels_RMI -> serverb:

Ligação RMI que irá permitir os *Storage Barrels* fornecer o resultado da sua pesquisa ao servidor. Após os *Storage Barrels* receberem o tipo de pesquisa que devem realizar e os parâmetros de pesquisa do *serverb,* os *Storage Barrels* realizam a pesquisa e informam o server dos seus resultados.

Ligação server -> Interface:

Esta ligação RMI é o que permite que, após realizada a tarefa pedida pelo *cliente* ao *server*, o *server* consiga devolver os seus resultados. É utilizada no final das operações realizadas pelo *server*, fornecendo apenas os resultados ao *cliente*. O *cliente* irá realizar diferentes operações tendo em conta os resultados fornecidos pelo *server*.

Ligação Downloader_RMI -> server:

Ligação RMI que serve para a página de administração saber quais os downloaders que se encontram ativos em tempo real.

Ligação URLQueue <-> Downloader_Multicast:

Esta ligação RMI serve para os Downloaders poderem ir buscar os URLs que têm de processar e também para armazenar todos os URLs que encontram durante o processamento.

Ligação Downloader_Multicast <-> Storage_Barrels_Multicast:

Durante o processamento dos URLs por parte dos Downloaders, é enviado, através de Multicast, pacotes com informações para serem processadas e armazenadas posteriormente nas bases de dados dos Barrels. Por sua vez, os Barrels ao receber esses pacotes fazem uma verificação do número de sequência do pacote enviando um NACK com a informação que tem em falta, tornando isto um protocolo Multicast fiável (explicado mais ao detalhe posteriormente).

2 - Threads:

Search_module:

O *search module* é responsável pela criação de 2 threads, *server* e *serverb*. Como o *search module* irá realizar a comunicação entre cliente->servidor e entre Storage Barrel -> servidor decidimos separar em 2, permitindo ao *server* realizar a comunicação cliente -> servidor e, ao *serverb*, a comunicação Storage Barrel -> servidor.

Server:

O *server* ficou responsável pela criação de uma *thread*, a *pagina_administracao.* A página de administração precisa de estar sempre ativa, uma vez que sempre que houver uma alteração no sistema, ela precisa de realizar atualizações. Como a *pagina*_administracao utiliza várias variáveis e funções presentes no *server*, decidimos optar pela criação da thread no *server*, facilitando o compartilhamento de funções e variáveis necessárias.

Downloader:

Em cada Downloader, temos 3 Threads:

A primeira é a principal onde vai processar todos os URLs que encontrar na lista. Durante este processamento também vai enviar a informação processada para os Barrels via um protocolo Multicast fiável. Estas mensagens vão ter um formato inspirado no que estava no enunciado do projeto: quando se trata de um url, ***type|url;url|<url>;title|<title>;citation|<citation>;*** , quando se trata de uma lista de palavras, ***type|word_list;url|<url>;word_<n>|<word>;*** , quando se trata de uma lista de URLs, ***type|url_list;url_0|<url>;url_<n>|<new_url>; .***

Para poder ter um protocolo Multicast fiável, foi necessário a implementação de uma outra Thread. Esta tem o objetivo de ficar à escuta de qualquer NACK proveniente dos Barrels. Sempre que receber um NACK, conforme as informações presentes nesse pacote, envia todas as mensagens que esse Barrel perdeu até ao momento. Quando envia esses pacotes, espera até receber um ACK para confirmar a receção por parte do Barrel.

Por último, optámos pela realização de uma thread, *downloader_RMI*. Embora não existisse uma conexão RMI entre o search module e o downloader na arquitetura inicial apresentada no projeto, decidimos criar uma thread que realizava essa conexão, uma vez que, facilitava a verificação dos downloaders ativos no sistema. Através desta thread, a página de administração pode facilmente saber quantos downloaders estão ativos no sistema, tornando esse processo mais eficiente.

Storage Barrels:

Optamos por realizar uma separação dos storage barrels em 2 threads, uma para multicast e uma para RMI, uma vez que ambas irão realizar operações bastante diferentes achamos que seria o mais correto.

Na Thread de multicast, o Barrel fica à escuta de pacotes provenientes dos Downloaders. Sempre que recebe um pacote, verifica se era o esperado, se for adiciona a informação necessária na base de dados. Caso contrário, envia um NACK com a informação que foi perdida, via multicast (explicado mais ao detalhe posteriormente).

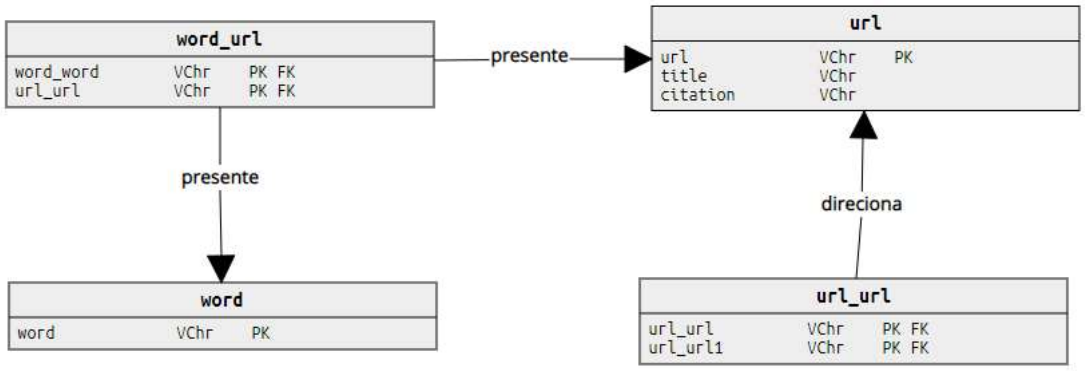
3 – URLQueue:

Optámos por desenvolver um programa próprio para fila de URLs, devido à conveniência das ligações RMI entre os *Downloaders*. Cada objeto URLObject contem uma *String* que corresponde ao url em si, o titulo da página web e uma citação presente na página. Estes objetos são adicionados, a uma lista ligada que se expande recursivamente. Para os Downloaders conseguirem aceder aos métodos remotos que possibilita a adição de elementos à fila e remoção, foi necessário criar um registo RMI no *main* da URLQueue e fazer *bind* da própria fila.

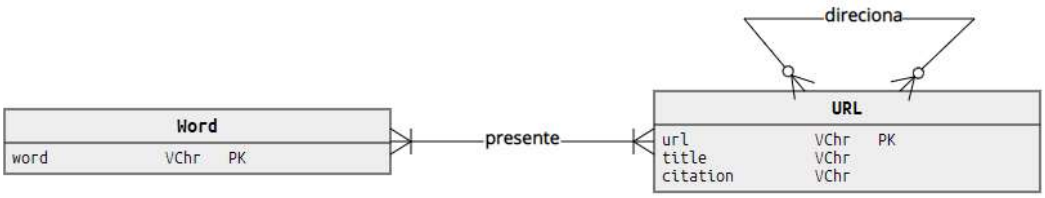
Estes métodos remotos, possibilitam uma adição e remoção de URLs sincronizada à fila.

4 – Base de Dados:

Toda a informação está armazenada nos Storage Barrels. Decidimos, para isto, criar uma base de dados por Barrel. Foi utilizado o PostgresSQL para esta implementação. O diagrama conceptual da base de dados é o seguinte:



Como podemos observar pelo diagrama, cada URL tem zero ou mais ligações para outros URLs, também cada URL tem pelo menos uma palavra e essa palavra tem de estar presente em pelo menos um URL. De modo que, por fim, tenhamos este diagrama físico da base de dados:



A informação adicionada na base de dados tem de ir por ordem, de modo que não ocorra problemas como ForeignKey Constraint, por exemplo. Portanto, começa por adicionar URLs à tabela URL, depois adiciona as palavras na tabela Word, com isto já conseguimos adicionar o par (word, url) à tabela Word_url. Por fim, são adicionados à tabela url_url os pares (url_url1), onde url é o url que foi processado e url1 é um dos urls que foram encontrados no processamento da página web.

Protocolo Multicast Fiável:

O protocolo *Multicast* fiável foi implementado na conexão entre os *Downloaders* e os *Index Barrels*.

No lado dos *Downloaders*, cada um tem 2 *Threads* correspondentes a este protocolo. A principal, é onde se encontra também o processamento da informação da página *web*, que ao traduzir para *Strings* de mensagens, são enviadas por *multicast*, chamando para isso o método *send* da Classe *ReliableMulticastServer*. Nesta classe, é preparada uma *socket multicast* com capacidade para receber pacotes e também enviar. Sempre que uma mensagem é enviada utilizando o método *send,* essa mensagem é codificada com o seu número de sequência e o identificador do Downloader como prefixo. É, também, adicionada a uma *ArrayList,* de modo a possibilitar o seu reencaminhamento caso algum dos Barrels o assim pedir.

A segunda *Thread* referente ao *multicast* dos *Downloaders*, é responsável por ficar à escuta de pacotes, provenientes dos *Barrels*. Todos os pacotes são ignorados, exceto quando for um *NACK*. Um *NACK* é um pacote especial, enviado pelos Barrels quando ocorre alguma falha na receção dos pacotes. O NACK tem a seguinte estrutura: ***-1--expectedPacket--seqNum--nDownloader.*** O ***-1*** identifica o packet como sendo um NACK, ***expectedPacket*** é o número de sequência do packet que o Barrels estava à espera de receber, ***seqNum*** é o número de sequência do packet que acabou por receber, ***nDownloader*** é o número de identificação do Downloader em que ocorreu esse problema.

Com a receção deste *NACK*, é possível ao Downloader reenviar todos os packets desde o *expectedPacket* até *seqNUM,* já que estas mensagens são guardadas numa *ArrayList* no *Downloader.* Sempre que envia uma destas mensagens, fica à espera da receção por parte dos Barrels e o posterior envio do ACK a confirmar a sua receção. Quando recebe o ACK, a thread continua para o próximo packet perdido a ser enviado, se este existir.

Da parte dos *Storage Barrels,* a inicialização é semelhante, pois o construtor da classe *ReliableMulticastClient* cria um *socket* também capaz de enviar e receber mensagens *multicast.* Neste construtor, também conectamos o *Barrel* à sua base de dados consoante o seu *id*. Na Thread *Storage_Barrels_Multicast,* o *Barrel* está à espera da receção de pacotes apenas provenientes dos Downloaders, ignorando tudo o resto. Quando recebe um, chama o método *checkPacket* da classe *ReliableMulticastClient*, com o objetivo de verificar o número de sequência. Neste método, a mensagem é descodificada de modo a obter o seu número de sequência e o número identificador do *Downloader* que enviou o *packet*. É depois comparado ao número esperado, se for igual ao esperado adiciona as informações na base de dados. Caso contrário, é enviado um *NACK* para o *Downloader* em questão, de acordo com a estrutura apresentada anteriormente. Quando recebe o *packet* que pediu, envia um *ACK* para notificar o *Downloader* da sua receção. Repete este processo até ter todos os *packets* em falta.

Com este protocolo de Multicast fiável, é possível ter segurança sobre os *packets* que são enviados e recebidos, mantendo assim uniformidade entre os *Storage Barrels*.

Organização do código:

Interface:

A *interface* ficou responsável por realizar a comunicação com o usuário, fornecendo um menu de escolhas e realizando diferentes operações tendo em conta o input inserido.

Para além da comunicação com o usuário a *interface* é responsável pela ligação RMI cliente->server, fornecendo os inputs realizados pelo usuário ao *server*, permitindo assim o server saber o que o usuário deseja realizar.

Para além disso, após o server realizar as operações necessárias a *interface* recebe os resultados da operação realizada e, imprime os resultados para o usuário.

Server:

O *server* será a parte do código que realiza as diferentes operações fornecidas aos usuários, realizando diferentes coisas tendo em conta o input enviado pela *interface*.

As várias operações podem ou não necessitar de comunicação RMI com o *serverb*, sendo essa ligação realizada pelo *server* quando necessário.

Após realizar o necessário, o *server* envia os resultados para a Interface.

Serverb:

O *serverb* será responsável por realizar as pesquisas, recebendo uma chamada pelo *server* quando for necessário realizar qualquer pesquisa.

Após receber a chamada do *server* o *serverb* irá fornecer ao *storage_barrel_RMI* os dados que são necessário pesquisar, sendo que este irá

realizar a pesquisa e fornecer os resultados ao *serverb.*

Depois da realização da pesquisa o *serverb* insere os resultados numa lista partilhada com o *server* e notifica o server para ele ir buscar os resultados.

Storage Barrel RMI:

O *Storage_barrel_RMI* é encarregue de realizar todas as pesquisas pedidas pelo usuário.

Após receber uma mensagem do *serverb* o *Storage_barrel_RMI* realiza a pesquisa pretendida, devolvendo os resultados obtidos por RMI ao *serverb*.

A realização das pesquisas foi realizada através de querys, pesquisando os valores necessários na base de dados.

Página administração:

A página de administração consiste numa thread que se encontra sempre a correr, a thread começa por esperar por um notify, recebendo-o sempre que uma alteração á sua informação acontece, após receber a notificação a thread

verifica todos os seus valores, verificando qual dos storage barrels e downloaders se encontram válidos.

Em seguida, a thread acessa um hashmap que contém todas as pesquisas realizadas, e guarda as 10 mais frequentes em uma lista. Depois de concluir essas verificações e armazenar as 10 pesquisas mais frequentes, a thread avisa o servidor, caso ele esteja esperando pelos valores atualizados.

Componente RMI:

Métodos Remotos:

Para os métodos remotos separamos em duas classes, uma para os clientes *Hello_C_I* e uma para os servidores *Hello_S_I*. Cada classe contém métodos específicos que foram desenvolvidos para atender às suas respectivas necessidades. Para a URLQueue também foi necessário a criação de métodos remotos para os Downloaders poderem manipular a fila. O funcionamento de todos estes métodos está presente no JavaDoc.

Failovers:

Foram realizados testes para detecção de possíveis erros nas conexões RMIs. Sempre que uma conexão era estabelecida, um bloco try/catch era utilizado. Se o catch capturasse uma exceção java.rmi.ConnectException, seria feita uma nova tentativa de conexão (retry). Se o número de tentativas falhasse, a conexão seria encerrada.

No caso da conexão do cliente para o servidor, se ocorresse uma falha no envio da mensagem, seria feita uma nova tentativa (retry) mantendo a mensagem. Se o servidor estivesse novamente disponível, a mensagem seria enviada. No entanto, se após seis tentativas de retry, o servidor permanecesse indisponível, a conexão seria encerrada e o cliente seria informado da indisponibilidade do servidor.

Para realizar os failovers na conexão dos Storage Barrels realizou-se algo similar. Realizando retrys caso não houvesse nenhum Storage Barrel ativo e, enviando uma pesquisa em branco caso fosse alcançado as 20 tentativas. Caso ainda houvesse Storage Barrels na lista, são todos percorridos, verificando a sua ligação, até ser encontrado uma ligação que funcione ou, até não haver mais Storage Barrels na lista.

CallBacks:

No programa, são realizados vários callbacks, sendo o método remoto "print_on_server" o local onde eles são mais frequentes.

Esse método é chamado por um cliente e, em seguida, é usado para imprimir no cliente por meio do método "print_on_client".

A seguir, apresenta-se um exemplo de callback que pode ser encontrado nesse método:

```
9 usages  Andre Moreira +1
public void print_on_server(String s, Hello_C_I c) throws RemoteException {

c.print_on_client(a);
```

Distribuição de tarefas pelos elementos do grupo:

As tarefas foram distribuídas como referido no enunciado do projeto. João Pinto ficou responsável pelo protocolo *multicast* fiável entre a comunicação entre os *Downloaders* e *Storage Barrels*, pela criação do programa da fila de URLs e a sua componente RMI, e pelo planeamento e implementação da base de dados. André Moreira ficou responsável pela realização do search module, interface do cliente e da componente RMI dos Barrels.

Testes realizados:

1. **Login com user não registado:** passed, mostra mensagem de erro a dizer que username ou password estão errados.

2. **Register com carateres inválidos**: passed, diz que não pode conter carateres inválidos.

3. **Register com username já utilizado:** passed, diz que username já se encontra utilizado.

4. **Register com username e password válidos:** passed, diz que registo foi realizado com sucesso.

5. **Login a user válido:** passed, diz que login foi realizado com sucesso.

6. **Logout sem login realizado:** passed, diz que necessita ter login realizado.

7. **Logout com login realizado:** passed, diz que logout foi realizado com sucesso.

8. **Indexar url invalido:** passed, downloader diz que houve erro na procura.

9. **Indexar url válido:** passed, url é indexado com sucesso.

10. **Fornecer parâmetros errados na pesquisa de termos:** passed, diz que houve erro nos parâmetros.

11. **Fornecer carateres inválidos na pesquisa de termos:** passed, diz que não pode conter carateres inválidos.

12. **Realizar pesquisa por URL sem login realizado:** passed, diz que necessita ter login realizado.

13. **Fornecer carateres inválidos para pesquisa de URL:** passed, diz que não pode conter carateres inválidos.

14. **Tentativa de comunicação cliente->server com server desligado:** passed, cliente tenta fazer retry na conexão até conseguir ou realizar 6 retries, após seis é lhe dito que o server não se encontra disponível.

15. **Tentativa de comunicação server->cliente com cliente desligado:** passed, server desiste da conexão e refere que houve erro a enviar para o cliente.

16. **Tentativa de comunicação serverb->Storage barrel com Storage Barrel desligado:** passed, server desiste do Storage barrel e tenta realizar conexão com outro.

17. **Storage Barrel volta a ter conexão enquanto server está à procura de um storage barrel:** passed, server consegue conectar ao storage barrel.

18. **Servidor volta a conectar enquanto cliente está a tentar voltar a ter conexão com ele:** passed, cliente realiza a conexão com o servidor mantendo o seu pedido.

19. **Realizar pesquisa de termos:** passed, devolve todos os sites com os termos pretendidos.

20. **Realizar pesquisa de URL:** passed, devolve todas as ligações ao URL fornecido.

21. **Consultar a informação geral do sistema:** passed, devolve todas as informações em tempo real.

22. **Quando um Barrel se desliga e volta a ligar volta a receber todas as packets que perdeu:** passed, recebe todos os packets perdidos até ao envio do NACK.

23. **Server é desligado e volta a ser ligado a meio do programa:** failed, embora o server continue a correr e receba os pedidos dos clientes normalmente, como nós não realizados a recuperação da informação que ele continha ele não irá detetar os Storage Barrels e Downloaders ativos, sendo que só poderá continuar a realizar operações caso seja realizada uma nova conexão entre eles.