Googol: motor de pesquisa de páginas Web

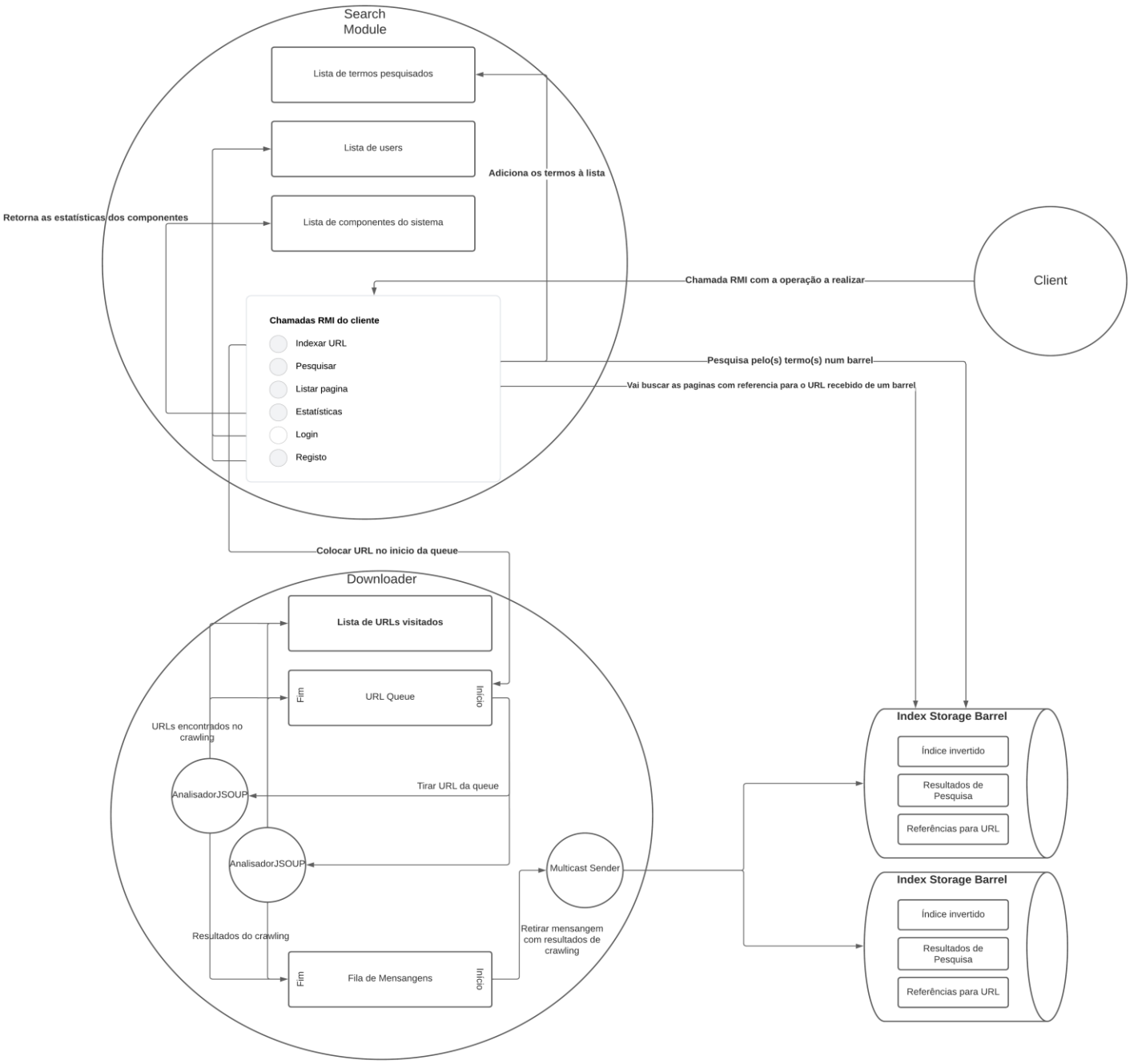
SISTEMAS DISTRIBUÍDS - META 1

David Silva – 2020217642
Eduardo Carneiro 2020240332

Introdução

Este projeto tem como objetivo o desenvolvimento de um motor de pesquisa, semelhante às funcionalidades primitivas do Google.com. Para isso, utilizámos Java como lingaguem de programação para obter esta finalidade.

Arquitetura de software



Arquitetura alto nível do sistema

O sistema está dividido em 4 programas centrais: Cliente, Search Module, Downloader e Index Storage Barrels(ISB).

Cada um destes componentes é fundamental para o funcionamento do sistema. Começando pelo Cliente, este apresenta uma estrutura simples. Realiza o lookup RMI para receber uma referência ao Search Module. Seguidamente, recebe input do utilizador sobre que operação realizar e, dependendo desta, faz a respetiva chamada, através de RMI, ao Search Module para a efetuar essa mesma operação. Em relação aos resultados recebidos, são processados de acordo com a sua especificação.

O Search Module é a porta de entrada do cliente ao sistema. Neste é recebida a operação a realizar pelo cliente e, dependendo dessa, acede às outras componentes do sistema (Downloader e ISB). É também onde se encontra a lista de utilizadores, com os respetivos usernames e passwords, a lista de componentes do sistema, utilizada para manter informações relativas a alguns componentes, e a lista de termos pesquisados, recorrida para obter os termos mais pesquisados.

O Downloader é a componente responsável pela realização do crawling dos URLs, tanto introduzidos pelos utilizadores, bem como os encontrados aquando do crawling de outros URLs. Existe nesta componente uma queue de URLs, na qual são introduzidos novos URLs para futuro crawl, bem como uma fila de mensagens e uma lista de URLs visitados (explicados mais adiante). Embora seja um só processo, são criadas várias threads do tipo 'AnalisadorJSOUP' (subcomponente onde é efetivamente efetuado o crawl do URL), de modo a realizar processamento em paralelo, aumentando a eficiência do sistema, e ainda uma thread 'MulticastSender' (subcomponente utilizada para envio de informação para os ISB). Estas threads 'AnalisadorJSOUP' retiram concorrentemente URLs da queue de URLs, verificam se estes já foram visitados (averiguando se estes se encontram na lista de URLs visitados) e, caso não, dão crawl utilizando o JSOUP, adicionando os URLs encontrados ao fim da queue de URLs. Quanto à informação sobre a página, é colocada numa estrutura própria e colocada na fila de mensangens. A thread 'MulticastSender' encarrega-se de retirar mensagem a mensagem da fila e enviar por multicast para todos os ISB.

Por fim, os Index Storage Barrels são a componente de armazenamento do sistema. É aqui que se encontra o índice invertido, com informação das páginas como título, uma citação curta, URL, bem como o conjunto de URLs acessíveis através dessa página. Existe também um HashMap com resultados de pesquisa, utilizado para fracionar a pesquisa enviada para o utilizador.

Protocolo de comunicação multicast

O Multicast é utilizado neste sistema distribuído para enviar a informação recolhida pelas threads AnalisadorJsoup (com recurso à biblioteca JSOUP) para os Barrels, para estes indexarem a informação e armazenarem-na.

Assim, quando a thread AnalisadorJSOUP acaba de analisar um URL adiciona para uma fila de mensagens (todas as threads AnalisadorJSOUP têm acesso direto a essa fila) a classe JSOUPData (contém a informação do url, o título da página associada ao url, uma citação da página, a lista de termos que essa página contém e a lista de URLs que podem ser chamados através desta página).

Posteriormente, a thread MulticastSender retira da lista uma classe JSOUPData a enviar para todos os barrels.

Para enviar a informação para os Barrels, primeiramente, serializamos a classe JSOUPData e enviamos o tamanho para o Barrel, de seguida, comprimimos o array de bytes obtido na serialização, e enviamos a classe comprimida para o Barrel.

Assim, primeiramente, o Barrel quando recebe a mensagem do tamanho guarda numa variável o tamanho da classe não comprimida que vai receber de seguida, e posteriormente recebe então a classe comprimida, descomprime e faz a desserialização.

Importa ainda referir que foram introduzidas medidas para tornar o Multicast o mais fiável possível, nomeadamente, o envio de uma mensagem quando o Barrel recebe a mensagem enviada por Multicast, como também um byte de controlo para distinguir se a mensagem se refere ao tamanho ou a uma classe.

Assim, quando enviamos uma mensagem Multicast e o Barrel recebe-a, o Barrel envia um ACK (neste caso, com o seu hashCode(), o seu identificador) para o MulticastSender (que o guarda num HashSet<Integers>, o MulticastSender conhece o número de ACKs que espera (quando se adiciona um Barrel é atualizada via RMI uma variável do tipo AtomicInteger presente no Downloader, que a thread MulticastSender tem acesso que permite saber quantos barrels estão ativos. Logo, no caso da primeira vez o tamanho do HashSet não coincidir com o número de Barrels disponíveis, tenta-se enviar novamente a mensagem e esperar as respostas do número do Barrels que faltaram enviar os ACKs, caso numa segunda tentativa não consigamos, assumimos que o Barrel está desligado, e, portanto, enviamos o HashSet de ACKs que recolhemos para o SearchModule eliminar os Barrels que não estão presentes nesse HashSet.

Quanto à parte dos bytes de controlo, escrevemos na primeira posição do array de bytes, um 0, caso o tipo da mensagem a enviar seja um tamanho, e escrevemos na priemira posição do array de bytes um 1, caso o tipo de mensagem a enviar seja uma classe, e de seguida escrevemos a informação que queremos enviar propriamente. Assim, quando o Barrel está à espera de uma mensagem do tipo 1 (classe) e receber uma mensagem do tipo 0 (tamanho), não tenta descomprimir o array de bytes (neste caso a string que indica tamanho) e desserializar a classe descomprimida.

Além disso, caso o número de Barrels seja o não retiramos nenhuma classe da fila de mensagens e enviamos por Multicast, pois, não temos nenhum Barrel para receber esta informação. Assim, de forma a evitar a espera ativa, fazemos um wait() no while que retira as mensagens da fila de mensagens e "acordamos esse wait" quando o número de barrels passar a ser 1, ou seja, quando o Barrel é adicionado, o SearchModule envia via RMI o número de Barrels conectados, e caso o número de Barrels conectados seja igual a 1 notifica o objeto que está em wait(), e assim a thread (re)começa a retirar classes da fila de mensagens e a enviar para os barrels.

Chamadas RMI e callbacks

São utilizadas chamadas RMI em todas as componentes do sistema. É possível efetuar 6 tipos diferentes de chamadas entre o cliente e o SearchModule: indexar um URL, efetuar uma pesquisa, listar páginas com referência para um URL recebido, mostrar as estatísticas do sistema, login e registo. Para cada uma destas operações, existe um método correspondente no SearchModule. Nesta mesma componente, de seguida e caso seja necessário, estas operações fazem também chamadas RMI tanto ao Downloader como aos ISB. No caso de indexar um URL, é efetuada uma chamada RMI ao Downloader, de modo a introduzir este URL no início da queue de URLs. Para realizar uma pesquisa ou obter a lista de URLs que fazem referência para um URL específico(informações contidas nos ISB), é também efetuada uma chamada RMI respetiva aos métodos nos ISB (o SearchModule escolhe 1 barrel aleatoriamente).

Em relação a callbacks, estes existem entre o Search Module e o Downloader e entre o Search Module e os ISB. Como já foi referido, existem chamadas do SearchModule para ambas estas componentes, no entanto também foram introduzidas chamadas RMI destas mesma componentes para o SearchModule, as quais são utilizadas para atualizar informações relativas a estatísticas do sistema, isto é, se certa Thread do Downloader está livre ou não, ou se certo ISB está livre ou não.

Componentes

Como descrito na arquitetatura do sistema, o sistema está dividido em 4 compoentes essenciais: Cliente, Search Module, Downloader e Index Storage Barrels(ISB).

SearchModule

O SearchModule tem como objetivo servir de "ponte" entre os diversos componentes.

Assim, o SearchModule utiliza algumas estruturas de dados que permitem armazenar a informação, descritas de seguida:

- Uma lista de Barrels conectados, quando iniciamos um Barrel, o barrel envia via RMI a sua referência da sua interface, de forma ao SearchModule puder fazer pedidos via RMI a qualquer barrel ativo. Esta lista é armazenada

utilizando a estrutura de dados, CopyOnWriteArrayList<BarrelRMI>, dado que é uma estrutura que é Thread-Safe e lida com concorrência.
- Uma lista de pesquisas com todas as pesquisas feitas por todos os utilizadores, assim, conseguimos saber os termos que foram mais pesquisados, a lista é armazenada utilizando estrutura de dados, CopyOnWriteArrayList<BarrelRMI>, dado que é uma estrutura que é Thread-Safe e lida com concorrência.
- Uma HashMap que tem como valor o username e como chave a password, permite armazenar os users, utiliza-se um ConcurrentHashMap, de forma a lidar com problemas de concorrência (mudar no codigo))
- Uma HashMap que tem como valor o nome do componente e a classe que representa o componente (contém o IP e um booleano que representa a disponibilidade do componente), utiliza-se um ConcurrentHashMap, de forma a lidar com problemas de concorrência.

Downloader

O Downloader tem como principal objetivo analisar os URLs e enviar a informação recolhida por Multicast para os Barrels.

Assim, o Downloader necessita de guardar alguma informação, descrita de seguida:

- Lista de URLs a analisar, podendo ser introduzidos pelo utilizador (neste caso, é introduzido à cabeça da lista, de forma a indexar de forma instantânea a página introduzida) ou então introduzidos pelo crawler quando encontra um url no ulr que está a analisar (neste caso, é introduzido na cauda da lista). A estrutura que usamos é uma LinkedBlockingDequeue, pois permite inserções à entrada ou à saída da lista, além disso, implementa mecanismos thread-safe, como também espera bloqueante (útil para quando a fila está vazia e queremos retirar algum URL, assim, só quando a fila voltar a ter algum elemento é que é devolvido)). Esta lista é usada entre as diferentes threads.
- Lista de URLs visitados, de forma a não repetir o crawl do mesmo URL, contudo, esta implementação não funciona bem com páginas dinâmicas, pois não voltaremos a analisar este URL. A estrutura dados que utilizamos é um Set<String> inicializado com um ConcurrentHashMap.newKeySet() de forma a ser thread-safe a assim lidar com problemas de concorrência.
- Fila de mensagens Multicast, as threads AnalisadorJsoup inserem a classe JSOUPData na fila (que contém as informações recolhidas pelo crawl) e a thread MulticastSender retira a classe JSOUPData da fila e envia por Multicast. Assim, os diferentes tipos de threads têm acesso a esta fila de mensagens. Além disso, a estrutura usada é um LinkedBlockingQueue pois é thread-safe e bloqueante (ou seja, quando a lista não tem elementos não fica em espera ativa a procurar um elemento).

Importa ainda referir que este componente se baseia essencialmente em 2 subcomponentes, uma thread MulticastSender que envia a info para os barrels por

Multicast e um número de threads definido que faz o crawl, propriamente dito, aos URLs que estão na fila de URLs.

Index Storage Barrels

Os Barrels servem para guardar o índice invertido dos urls, isto é, para cada termo guardar os urls em que esse termo aparece, o core para fazer pesquisas de forma rápida e eficiente.

Assim, o barrel armazena diferentes informações, descritas de seguida:

- Um HashMap que contém o termo como chave e um HashSet de classes infoURL (contém o url em si, o título, uma citação, e um HashSet de classes infoURL que conseguem aceder esse url) como valor. Esta informação é armazenada usando um ConcurrentHashMap de forma a lidar com problemas de concorrência.
- Um HashMap que tem como chave a string URL e como valor a classe infoURL. Assim, este HashMap serve como referência para que toda as alterações feitas numa classe infoURL reflitam-se nos HashSets que têm essas classes infoURL. Esta informação é armazenada usando um ConcurrentHashMap de forma a lidar com problemas de concorrência.
- Um HashMap que tem como chave o identificador de um cliente (neste caso o hashCode() enviado pelo cliente ao fazer uma pesquisa) e como valor os resultados da pesquisa que faltam enviar para o cliente. Esta informação é armazenada usando um ConcurrentHashMap de forma a lidar com problemas de concorrência.

Tratamento de exceções e failover

EXCEÇÕES

Para o tratamento de exceções, no geral, utilizamos o simples try/catch do Java, adequando depois a falha ao contexto, isto é, caso seja uma falha que necessite de ser recuperada, como uma falha numa pesquisa de um ISB, realizamos novamente a pesquisa, mas utilizando outro ISB. Mais especificamente:

- Caso o Search Module falhe a meio de uma operação, o cliente recebe uma exceção e volta a tentar realizar a mesma operação automaticamente.
- Caso um ISB falhe a meio de uma operação, o Search Module recebe uma exceção e tenta executar a mesma operação noutro ISB.
- Caso o Downloader falhe a meio da operação de introduzir um novo URL na queue através da chamada RMI, o Search Module volta a tentar fazer novamente esta operação.

FAILOVERS

Até agora foram referidos problemas intra-processo, isto é, caso ocorresse algum problema na execução normal do programa. Tendo em conta que é suposto simular um sistema distribuído, isto é, ter programas a correr em locais diferentes e computadores diferentes, sabemos que podem existir falhas externas que intercetam o funcionamento normal do programa, tais como falhas de energia ou quebras de hardware. Desse modo, efetuamos algumas proteções ao nosso código de modo a, caso exista uma destas falhas, seja possível recuperar informação já processada.

Deste modo, introduzimos a escrita de informação crucial para disco, utilizando objetos serializáveis. Caso, por exemplo, um ISB falhe, ao ser reiniciado toda a informação sobre as páginas será recuperada via estes ficheiros. O mesmo acontece para o Downloader e para o Search Module. Como estes componentes poderiam falhar a meio da escrita, utilizámos um mecanismo de duplicação de ficheiros aquando da escrita, de modo a existir sempre um ficheiro íntegro dos dados. A escrita para disco é efetuada apenas após algumas iterações do programa, minimizando o overhead de escrever a cada alteração, no entanto gerando alguma perda em caso de falha.

Distribuição de tarefas

Ao longo do desenvolvimento do projeto fomos comunicando as tarefas e os problemas que apareciam e fomos fazendo em conjunto em espírito de entreajuda.

Deste modo, o trabalho acabou por ser dividido justamente entre os elementos do grupo, tendo ambos trabalhado um pouco por todas as funcionalidades.

Testagem

Teste	Resultado
Indexar o URL 'https://www.uc.pt'	Pass
Pesquisar pelo termo 'uc'	Pass
Avançar para a próxima página dos resultados	Pass
Efetuar registo	Pass
Efetuar Login	Pass
Lista páginas com ligação para o URL 'https://www.uc.pt'	Pass
Pedir estatísticas do sistema	Pass
Fail do Downloader e recuperação dos dados	Pass
Fail do ISB e recuperação dos dados	Pass

Informações adicionais

São apresentadas algumas exceções aquando do crawling de alguns Downloaders. Devem-se maioritariamente a URLs inválidos ou cookies inválidos, pelo que não atrapalham a execução normal do programa.

Conclusão

Através do desenvolvimento desta meta foi-nos possível aprofundar os nosso conhecimentos acerca de RMI, bem como entender melhor o funcionamento de arquiteturas que apresentam redundância e processamento paralelo, de modo a aumentar a eficiência e o *uptime*.

Em suma, parece-nos que a solução desenvolvida pelo grupo seja adequada ao problema, apresentando todas as funcionalidades pedidas, bem como praticamente todos os requisitos não-funcionais e *failover*.