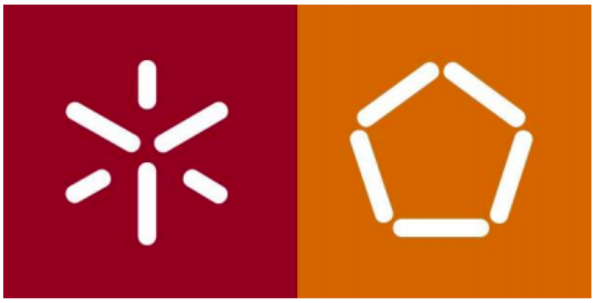

Universidade do Minho

Mestrado Integrado em Engenharia Informática

Fundamentos de Sistemas Distribuídos

Armazenamento de Pares Chave-Valor Distribuído

Eduardo Lourenço da Conceição (A83870)
Ricardo Filipe Dantas Costa (A85851)
Rui Nuno Borges Cruz Oliveira (A83610)
Cândido Filipe Lima do Vale (PG42816)

15/01/2021
Braga

Índice

1 Introdução

Para o trabalho de Fundamentos de Sistemas Distribuídos, foi-nos proposta a construção de um sistema de armazenamento de pares chave-valor distribuído, utilizando os conhecimentos que adquirimos sobre programação orientada a eventos e sobre relógios lógicos. Neste relatório iremos expor a arquitetura do sistema que implementamos, bem como a estratégia adotada para implementar as operações pedidas, seguindo os devidos requisitos, bem como as valorizações.

2 Implementação do Sistema

2.1 Arquitetura

2.1.1 Cliente

O cliente vai ser a entidade com que um utilizador poderá fazer a comunicação com os servidores. A classe que permite isto denomina-se *ClientStub*, e deverá ser inicializada com o seu *port number* e com o do servidor do tipo *Forwarder* a que se liga (que será explicado mais tarde). O *stub* só fornece dois métodos: *put* e *get*, como definidos nos requisitos. Ambos os métodos devolvem *CompletableFuture*, cujo conteúdo só será preenchido uma vez que o pedido é realmente terminado.

Para além deste *ClientStub*, temos também um cliente que invoca as operações deste *stub* deu ma forma bloqueante (invocando o método *get()* da classe *CompletableFuture* sobre os resultados do *get* e do *put* do *ClientStub*), ao qual chamamos *BlockingClient*.

2.1.2 Servidor

O servidor é bem mais complexo que o cliente, e pode ser de um de dois tipos: *Forwarder* ou *Storage*. Os servidores do tipo *Forwarder* têm o papel de receber os pedidos do cliente, dividí-los e enviá-los para as *Storages* que puderão responder (*scatter*), quando adquirirem o devido *lock*, e depois, receber a resposta das *Storages*, agrupá-la (*gather*) e enviar ao cliente. Por sua vez, os servidores do tipo *Storage* são os que guardam os pares chave-valor, e o seu papel é apenas responder aos pedidos dos servidores *Forwarder*. A divisão dos elementos pelos vários servidores é elementar: cada *Forwarder* recebe como parâmetro de construtor uma lista com o *port number* de todas as *Storages*, que será ordenada. Os elementos serão guardados, depois, consoante uma função de *hashing* muito simples, baseada no resto da divisão inteira pelo número de servidores *Storage* (se este resto for 0, por exemplo, o par vai ser guardado no servidor cujo *port* esteja na cabeça da lista, caso seja 1 será no segundo da lista, etc.).

Todos os servidores conhecem a rede, sendo que os clientes só conhecem o *Forwarder* a que estão ligados. Os *Forwarders* comunicam com as *Storages*, mas também com os outros *Forwarders*, de modo a podermos implementar um algoritmo de Exclusão Mútua Distribuída (a ver mais à frente), partilhando entre si os seus valores de *clock*.

A escolha deste tipo de arquitetura tem as seguintes vantagens:

- Diminui a memória ocupada em cada *Storage*, separando os pares por várias *storages*;
- Oferece privacidade ao cliente, uma vez que apenas o *Forwarder* conhece o seu endereço;
- Coloca o trabalho de comunicação de relógios e do processo de obtenção do *lock* nos *Forwarders*, livrando as *Storages* e os clientes do mesmo.

Uma visualização de uma possível topologia utilizando estas classes seria a seguinte:

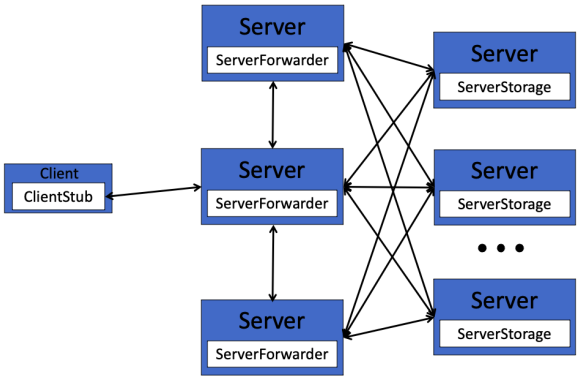

Figure 1: Topologia Exemplificativa

2.2 Estratégias Adotadas

2.2.1 Requisitos

1. O sistema deve usar vários servidores para armazenar subconjuntos disjuntos de chaves a que são associados valores

Para completar qualquer uma das duas operações que nos foram pedidas (*put* e *get*), o cliente enviará um *request* ao seu *Forwarder*. Esta operação eventualmente vai devolver um *CompletableFuture*.

Uma vez que o *Forwarder* recebe um pedido, vai separá-lo em vários, associados ao servidor *Storage* que os vai poder receber. Depois, o *Forwarder* vai enviar o pedido repartido a todos os servidores que o poderão responder, numa operação de *scatter*, no caso de obter o *lock* (explicado mais tarde). Depois, guarda um objeto num *map*, objeto tal que tem a informação sobre um pedido, nomeadamente em quantos pedaços foi repartido. Depois os devidos *Storages* irão processar os pedidos, e enviam de volta o resultado. O *Forwarder* agrega esses resultados, de modo a construir a resposta que

irá enviar ao cliente, e quando vir que recebeu todos os pedaços de um pedido, remove esse pedido do *map* mencionado anteriormente e envia a resposta ao cliente.

Quando o cliente recebe a resposta ao seu pedido, marca-o como completo e termina a operação.

2.1. Um cliente observa todas as escritas feitas por si próprio anteriormente

Quando um pedido é feito inicialmente, mesmo antes de finalizar a operação, os métodos *get* e *put* devolvem imediatamente um *CompletableFuture*, para já vazio, e guardam-no num *map* de futuros vazios no *ClientStub*, associados a um identificador de pedido. Quando o servidor responde que um pedido foi acabado, aí o *stub* vai procurar o futuro associado ao identificador do pedido cuja resposta acabou de obter e vai completá-lo com o corpo da resposta. Só aqui é que um pedido é verdadeiramente finalizado. No entanto, o *get* e o *put* não são operações bloqueantes por si, pelo que só fazendo *get()* ao *CompletableFuture* que devolvem, de forma a acedermos ao corpo do mesmo, é que garantimos que estamos à espera até que sejam completadas.

De notar que a classe *BlockingClient*, que é basicamente um *wrapper* à volta de um *ClientStub*, cujos métodos *put* e *get* que fornece são bloqueantes, implementa esta ideia que referimos.

2.2 Se dois clientes tentarem escrever concorrentemente os mesmos itens, os valores que persistem para serem lidos depois das operações terminarem são todos provenientes do mesmo cliente.

Para conseguirmos garantir este requisito, decidimos implementar uma versão do Algoritmo de Exclusão Mútua que exploramos nas aulas teóricas.

Desta forma, implementamos uma classe *WaitingRequest* que irá caracterizar um pedido em espera e que irá incluir a *timestamp* o tipo de pedido (*Put*/*Get*) e o *Forwarder* a que este está associado que será identificado pela porta em que esse determinado *Forwarder* está a correr.

Esta classe permitir-nos-á ter em cada *Forwarder* uma lista dos pedidos que aguardam execução. Assim sendo, cada *Forwarder*, cada vez que recebe um novo pedido, irá adicionar à sua lista e, independentemente do seu tipo, irá enviar para todos os todos os outros *Forwarders* uma mensagem de novo evento com o seu *timestamp* associado a esse evento e com a sua porta.

Quando um *Forwarder* recebe este "novo evento" este irá verificar se é um novo evento de processamento (eventos para processar pedidos que já foram inseridos na lista) e caso não tal não seja o caso (seja um novo evento originado por um novo pedido), irá adicionar à sua lista de pedidos uma entrada com os tipos de pedidos a *null* e com a *timestamp* e a porta recebida, desta forma todos os *Forwarders* serão capazes de construir filas de pedidos iguais permitindo-lhes identificar qual será o próximo que deve ser executado.

Uma vez adicionado o pedido na lista, cada *Forwarder* irá fazer *broadcast* de mensagens de *clock-update* com o seu valor de relógio para que todos os outros possam atualizar as suas tabelas de mínimos e não fiquem bloqueados.

Esta etapa é importante e decidimos implementar desta forma, apesar de ser menos

eficiente, porque permite que a aplicação continue a correr mesmo quando a carga não é distribuída de forma equivalente entre todos os servidores. A opção mais eficiente, seria apenas enviar a atualização de relógio para o *Forwarder* que originou o novo evento mas nesse caso o processamento de pedidos não iria progredir caso tivéssemos um *Forwarder* sem qualquer carga, pelo que decidimos optar pelo envio de mensagens de actualização dos relógios explícitas.

Desta forma, todos os *Forwarders* terão um *Event Listener* que será executado quando receberem as mensagens de *clock-update*, onde irão atualizar a tabela de relógios e percorrer a lista de pedidos seleccionado aquele que tem a *timestamp* e a porta mais pequena. Este processo será efectuado em todos os *Forwarders* e como todos eles têm listas de pedidos iguais, o pedido seleccionado irá ser o mesmo em todos os *Forwarders*.

Aqui entra outro ponto fundamental do nosso *Mutex*, cada *Forwarder* irá ter uma variavél, objecto *WaitingRequest current*, que irá representar uma abstracção do *Lock* dos *Storages*. Isto é, só pode haver um pedido a ocupar o *current* e esse pedido é que será processado. Uma vez identificado o pedido a ser executado cada *Forwarder* definirá o *current* como sendo esse pedido e irá tentar executá-lo.

É importante ressalvar que, apesar dos eventos que permitem definir o *current* serem disparados várias vezes assim que um *Forwarder* conseguir definir o pedido para processamento, *current*, para diferente de *null*, da próxima vez que o evento seja executado este processamento já não será efetuado, a não ser que *current* já tenha sido "libertado".

Se o valor nos campos dos pedidos (*Get* e *Put*) desse objecto for *null*, esse *Forwarder* sabe que não é ele que tem que tratar da execução desse pedido e irá aguardar. O *Forwarder* que verificar que o campo *Get* ou *Put* do objecto *current* é diferente de *null*, irá proceder à sua execução, enviando para os *Storage Servers* e irá aguardar a resposta destes.

Uma vez obtida esta resposta, irá "libertar o *lock*", retirando o pedido da lista de pedidos em espera e redefinindo o seu *current* a *null*. De seguida irá notificar todos os outros *Forwarders* para que façam o mesmo.

Após libertar o *current*, os *Forwarders* verificam se existem mais pedidos em fila para ser processados e, se tal for o caso, enviam para si próprios uma mensagem de *new-event* com o *payload* "*processing*" de modo a tratar os itens restantes que ainda existam na lista. Não havendo mais pedidos na lista, aguardam.

2.2.2 Valorização

1. A resolução proposta for modular, maximizando o código que é independente desta aplicação em concreto e adequado à utilização em grande escala, sem gargalos ou limitações artificiais

O código desenvolvido não foi implementado para uma aplicação específica, ou para uma topologia específica. Apenas foi para uma qualquer topologia, desde que sigam as regras que expusemos antes. É um sistema que serve para um qualquer número de servidores e clientes, onde a topologia é estabelecida desde início.

Se nós quiséssemos permitir que a topologia fosse dinâmica, teríamos de fazer três modificações:

- A função de *hashing* dos servidores teria de mudar, para que funcionasse mesmo quando o número de servidores altera ao longo do tempo, algo que não temos em consideração;

- Para que a rede de *Forwarders* possa mudar, teríamos que, em vez de passar uma lista dos *ports* de cada *Forwarder* ao construtor de classe, teríamos de construir a rede dinamicamente, fazendo com que um *Forwader* novo enviasse uma mensagem aos já na rede de modo a assinalar a sua presença.

- Implementar mecanismos mais fortes de controlo de concorrência ao nível dos servidores *Forwarders*, de modo a garantirmos que não temos *deadlocks* devido à escrita errónea em cima de valores que ainda estão a ser utilizados (que suspeitamos resultar nos *deadlocks* que mencionaremos mais à frente)

Tendo estes dois pontos em mente, poderíamos construir um sistema mais dinâmico. Contudo, devido a restrições de tempo, não pudemos implementar estas ideias (mas também não era o pedido no enunciado).

2. Garantir que leituras concorrentes de itens que estão a ser modificados por outros clientes não observam uma escrita parcial

Este ponto foi assegurado no nosso sistema pelo facto que o *Lock* que é feito sobre as *Storages* é também feito aquando de uma leitura para responder a um *get*. Assim, existe indubitavelmente uma garantia de que não estamos a observar valores a meio de uma escrita enquanto fazemos uma leitura.

No entanto, é de sublinhar que esta forma de implementar este ponto é, sem dúvida, pouco eficiente, uma vez que estamos a bloquear o sistema inteiro para uma leitura. Em sistemas de pequena a média escala, os efeitos deste *bottleneck* poderão não ser sentidos, mas em sistemas de grande escala, quase certamente serão.

3. Avaliação de desempenho

Quanto a testes que fizemos ao sistema, estes foram sempre feitos com as seguintes condições:

- Nos servidores utilizamos 7 *Forwarders* e 3 *Storages*. O número de *Storages* é relativamente indiferente, uma vez que os problemas cujo desempenho da solução queremos avaliar estão presentes nos *Forwarders*, uma vez que são estes que tratam da Exclusão Mútua. Apenas afetam em quantos pedaços um *scatter* é divido.

- Para os clientes, experimentamos com várias configurações e permutações de 5 clientes a tentar fazer *get* e *put*, concorrentemente. Nos diferentes testes que realizamos, experimentamos com configurações em que alguns clientes partilhavam o mesmo *Forwarder* e outras em que todos os clientes estavam associados a *Forwarders* diferentes.

Com estas condições, pudemos ver a reação do grupo de servidores a vários cenários.

Um dos aspetos mais notórios e mais negativos do sistema é o facto que, quando temos vários *Forwarders* a tentar aceder às *Storages* em rápida sequência (ou seja, a

tentar obter o *Lock*), chega a um ponto em que o sistema fica em estado de *deadlock*. Isto é um problema que nós reconhecemos no nosso sistema, mas que não tivemos tempo para o conseguir resolver, apesar de termos tentado imensas vezes. Reconhecemos também que isto se deverá dar ao facto que não implementamos controlo de concorrência em certas variáveis de instância nos servidores *Forwarder*. No entanto, pela altura que nós concluímos isto, já não tivemos tempo para resolver este problema.

Não obstante, com a configuração que utiliza **5 clientes em *Forwarders* distintos**, pudemos ver a seguinte atividade com a ferramenta *JConsole* para responder aos pedidos (sendo que foram 45 *put* e 30 *get*, um número relativamente pequeno, mas que não causa *deadlocks*):

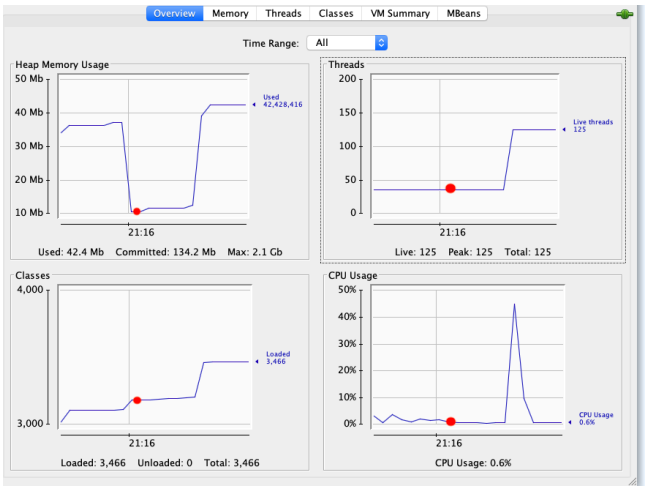

Figure 2: Com um cliente por *Forwarder*

O ponto vermelho representa aproximadamente o ponto no tempo em que o pedido é feito. Como seria de imaginar, a atividade do CPU tem um pico no momento seguinte. Isto deve-se à quantidade de mensagens que circulam no sistema no momento, para obter o *lock*, informar os outros *Forwarders* de um novo evento, etc. É interessante também observar como o número de *threads* quase triplica, e a memória da *heap* utilizada também. Tudo isto se deve, a nosso ver, também à quantidade de mensagens em circulação, e da forma como o compilador de Java trata da representação das mesmas em memória.

Para podermos ver outro cenário de execução com a mesma topologia, corremos o mesmo teste, mas com **4 clientes, ligados dois a um Forwarder e dois a outro na mesma topologia**. Os resultados capturados pelo *JConsole* foram os seguintes:

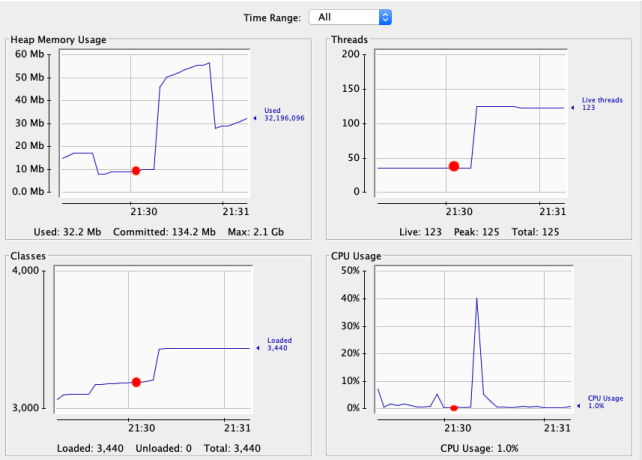

Figure 3: Com dois clientes por *Forwarder*

Como podemos ver, os resultados são muito semelhantes ao primeiro caso, e para o mesmo número de *Forwarders* e de mensagens, será sempre semelhante, pois um pedido tem de ser comunicado a todos os *Forwarders* independentemente de onde veio. Assim, o *overhead* de comunicação do sistema não será alterado de formas relevantes consoante a forma como os clientes estão ligados aos servidores.

Para referência, os testes foram realizados numa máquina *MacBook Air, early-2015*, com um processador *Intel Core i5* de 1.6GHz, com 8GBs de RAM DDR3, com 3MBs de cache L3. A versão do *JConsole* utilizada foi a versão *11.0.9+7-LTS*.

3 Conclusão

Em geral, acreditamos ter cumprido os requisitos do trabalho, tendo também alcançado alguns aspetos valorizados. No entanto, reconhecemos que o nosso trabalho tem falhas no que toca a controlo de concorrência, que leva a que um problema como *deadlocks* surjam periodicamente, o que não é de todo desejável.

Em suma, apesar de aceitarmos que o trabalho tem alguns problemas, acreditamos que cumprimos a ideia daquilo que nos foi pedido.