Faculdade de Engenharia da Universidade do Porto

Relatório do 2º Trabalho Laboratorial

«Distributed and Partitioned Key-Value Store»

Computação Paralela e Distribuída, Maio de 2022 - L.EIC:

Professor Jorge Manuel Gomes Barbosa

Professor Pedro Alexandre Guimarães Lobo Ferreira Souto

Professor Pedro Miranda de Andrade de Albuquerque d'Orey

Professor António Jesus Monteiro de Castro

Grupo 3LEIC03G04 (Estudantes & Autores**):**

Henrique Ribeiro Nunes up201906852@up.pt

Patrícia do Carmo Nunes Oliveira up201905427@up.pt

Rafael Fernando Ribeiro Camelo up201907729@up.pt

1. Introdução

O trabalho foi realizado no âmbito da unidade curricular de Computação Paralela e Distribuída, do ano letivo 2021/22, com o objetivo de compreender as vantagens e desafios adjacentes à implementação de um sistema distribuído, desenvolvendo uma *Distributed and Partitioned Key-Value Store*.

O relatório tem como objetivo documentar, focar e explorar as estratégias utilizadas no desenvolvimento dos requisitos do projeto que contribuem para o aumento da nota.

2. Membership Service

2.1. Formato das Mensagens

As mensagens na secção do *membership service* seguem um formato genérico composto por duas partes, ambas não codificadas e legíveis por humanos: cabeçalho e corpo. O cabeçalho de cada mensagem contém toda a informação de controle: o **tipo de mensagem** (*join*, *leave, membership* ou *msUpdate*), o **tamanho do corpo**, caso esta tenha conteúdo, e outras informações que variam com o tipo de mensagem específica. Cada linha do cabeçalho apresenta o formato "***key: value"***, terminando no caráter <*LF*>. A separação entre o cabeçalho e o corpo é realizada por uma linha vazia (apenas contendo o caráter <*LF*>). O corpo da mensagem não tem restrições de tamanho e estrutura.

No caso de operações do tipo ***join*** ou ***leave*** a mensagem irá conter o identificador, o endereço IP, o *membership counter* e a porta atribuída para a troca de mensagens *key-value* do nó. Nas operações de ***join*** é ainda agrupada a porta por onde este espera receber as mensagens *membership* iniciais.

O último tipo de operação, ***membership***, é utilizado no contexto da transmissão de dados relativos à visão pessoal, de um nó para um nó individual ou para o *cluster*. Assim, o cabeçalho desta mensagem é constituído, para além das entradas obrigatórias, apenas pelo identificador do nó que criou a mensagem, mas, em contrapartida, o corpo desta é mais extenso, contendo toda a informação guardada no nó sobre o *cluster*, isto é, a lista de nós presentes no *cluster* e os últimos 32 *membership logs*. Primeiro são adicionados ao corpo os nós do cluster no formato ***"<ip>:<port>|<id>"***, de seguida é utilizado um separador ***"---sep---"*** e finalmente são adicionados os *logs* no formato ***"<membershipCounter>|<id>"***. Cada entrada do corpo é finalizada com o caráter <*LF*>.

2.2. Protocolo de Membership

Os nós do cluster são identificados pelo conjunto do seu endereço IP e *porta*. O seu identificador (*id*) é calculado aplicando a função de *hash* SHA-256 a este conjunto no formato ***"<ip>:<port>"***.

Os nós guardam internamente a sua visão do *cluster* (***MembershipView***) que é definida pelo

conjunto de nós atualmente no *cluster* (***MembershipTable***) e o histórico de registos de entrada e saída dos nós do *cluster* (***MembershipLog***).

Para um nó se juntar ao *cluster* é necessário que este receba 3 e apenas 3 mensagens de *membership* por outros nós do *cluster*. Os nós do *cluster* ao receberem uma mensagem de *join* esperam um tempo aleatório (entre 0 a 500 milisegundos) antes de mandarem a sua mensagem *membership*. No entanto, no caso dos 3 primeiros nós isso nunca acontecerá. Nesta situação, a estratégia implementada passa por, independentemente do número de mensagens recebidas, o nó se juntar com sucesso ao *cluster*. Na eventualidade de tal junção acontecer enquanto os outros nós do *cluster* estejam em baixo, a implementação de robustos mecanismos de sincronização e a capacidade de um nó fundir a sua *MembershipView* com as *MembershipViews* que recebe por mensagens periódicas garante que a regularização do *cluster* é realizada rapidamente.

Para fundir uma *MembershipView* são usados algoritmos definidos pelos protocolos específicos para esta tarefa. Na fusão de um *MembershipLog* noutro, o primeiro é percorrido do registo mais antigo para o mais recente e adiciona este registo ao final da lista de registos do segundo caso este não possua um registo com o mesmo identificador ou possua um registo com o mesmo identificador, mas com um *membershipCounter* associado inferior, sendo que neste caso o registo desatualizado é eliminado da lista. Na fusão de duas *MembershipTable*, o nó recetor da mensagem procede à comparação da sua tabela com a recebida e guarda as entradas que não possui atualmente. Posteriormente, filtra desta todas as entradas que não estão de acordo com os logs já atualizados, isto é, elimina as entradas cujo identificador está associado a um *log* com *MembershipCounter* ímpar.

2.3. Envio das Memberships mais Atualizadas

Foi introduzido o conceito de **probabilidade de um nó estar atualizado** para poder assim decidir se este deveria ser o escolhido para enviar as mensagens *membership*. A técnica utilizada consiste em assumir a completa atualização de um nó quando este termina uma operação de *join* (*P=1*). De seguida, sempre que recebe uma mensagem de *membership* periódica, caso os seus *logs* estejam desatualizados em relação aos recebidos a sua probabilidade é reduzida por um fator de 20% (*P=P-0.2*), caso contrário é aumentada em 10% (*P=P+0.1*). Esta probabilidade, limitada entre 0 e 1.0, caso seja inferior a 0.5, implica que o nó não deva enviar a sua visão *membership* caso esta lhe seja pedida através de *multicast*. Esta técnica foi utilizada com base no pressuposto de que um nó que esteja frequentemente desatualizado tem uma maior probabilidade de estar novamente desatualizado e que, por isso, não deveria ter a responsabilidade de enviar mensagens *membership* até ter uma taxa de sucesso de atualização maior.

2.4. Remote Method Invocation (RMI)

O **RMI** foi implementado de forma a substituir as mensagens de *join* e *leave* por parte do cliente que assim invoca diretamente o método remoto. Neste sentido, foi necessário definir a *Interface Remota*, associar uma *Store* a um registo, e invocar o método dessa *Interface Remota* no *Test Client*.

2.5. Tolerância a Falhas

Nesta secção foram **introduzidas duas tolerâncias a falhas principais**, a primeira consiste na eleição do nó que deverá enviar as mensagens *membership* periódicas em caso de falha e a segunda envolve a reação de um nó a um *crash* quando este se encontrava no *cluster* antes da falha.

No primeiro caso, inicialmente o nó escolhido para enviar as mensagens periódicas é o nó com id mais baixo. No entanto, este pode falhar sem ter forma de notificar os restantes nós do *cluster*. Perante esta possível falha é importante que outro nó consiga assumir a responsabilidade desta tarefa. Para isso, se um nó não receber uma mensagem *membership* no dobro do tempo esperado, percebe que algo de errado se passa e atualiza internamente a sua visão do nó que reconhece como o responsável pelo envio das mensagens periódicas (***smallestOnline***) para o nó com o id seguinte. Quando este nó for ele próprio, então deverá assumir a responsabilidade de enviar as mensagens. Caso, durante este processo, receba uma mensagem *membership* de um nó com id inferior ao ***smallestOnline*** atualiza o seu ***smallestOnline***, caso contrário continua a incrementar para o nó seguinte a cada intervalo de tempo.

No segundo caso, quando um nó estava ***join*** e foi abaixo, então quando volta a ficar ativo, tira proveito de guardar o seu *membership counter* e *membership logs* em memória não volátil para repor o seu estado anterior, voltando a introduzir-se no *cluster* e enviando uma mensagem ***msUpdate*** para o *multicast*, de forma a pedir que os nós ativos no *cluster* o atualizem com as suas *membership views*. Este método de ***rejoin*** é semelhante ao ***join*** embora tenha menos implicações no recetor e seja mais leve, uma vez que a mensagem apenas é enviada uma vez, mesmo que não tenha recebido três mensagens *membership*.

3. Storage Service

Nesta fase, a *thread* principal da *Store* inicia um servidor TCP e sempre que é aceite uma nova conexão, é criada uma nova *thread* para lidar com o pedido recebido (***put***, ***get*** ou ***delete***). Operação de *put* guarda um par chave-valor no nó do *cluster* responsável por este; operação de *get* obtém o valor do par de uma dada chave; operação de *delete* elimina o par chave-valor de uma determinada chave.

3.1. Formato das Mensagens

As mensagens seguem o formato genérico descrito na seção 2.1. O cabeçalho de cada mensagem contém informação relevante e necessária para compreender a operação a realizar (*put*, *get* ou *delete*), tal como a **chave** do par chave-valor relacionado e o **tamanho do corpo** da mensagem apenas no caso das operações ***put***, pois apenas estas transmitem conteúdo no corpo da mensagem, nomeadamente o **valor** do par relacionado.

3.2. Key-value stores e o cluster

Cada nó do cluster, *store*, mantém registo em memória volátil e não volátil de todas as chaves pelas quais é responsável. De forma a encontrar mais rápida e eficientemente o nó responsável por uma determinada chave, a ***MembershipTable***, que contém todos os nós do *cluster*, é constituída por entradas que associam o *ip* e *porta* de cada nó ao *hash* da sequência *ip:port*. O id do nó corresponde, exatamente, a este *hash*. A tabela é implementada por um ***TreeMap*** do *package java.utils* de forma a manter estas entradas ordenadas pelo valor do *hash* e permitir utilizar a **técnica *consistent hashing***.

Todas as operações do *storage service* podem ser direcionadas para qualquer nó do *cluster*, uma vez que existe internamente o **redirecionamento** para os responsáveis que deverão responder ao pedido. Com esta estratégia é possível abstrair o cliente dos detalhes de organização e distribuição dos pares pelo *cluster* pois independentemente do nó ao qual faça o pedido, irá receber por ele a resposta ao mesmo, uma vez que esta é propagada de volta à origem.

3.3. Nós do cluster e pares chave-valor

Os pares chave-valor são guardados no nó do *cluster* cujo *id* sucede a chave do par. Tomando partido do ***TreeMap***, que já mantém todos os nós do cluster ordenados pelo *id*, para encontrar o nó responsável por uma chave, basta apenas encontrar a entrada nesta estrutura igual ou superior à chave do par, com o detalhe de que, caso não existisse nenhuma, o sucessor da chave do par seria a primeira entrada do *Map*.

3.4. Alteração do membership e transferência de pares

Após a receção de um evento *membership*, junção ou saída de um nó do *cluster* (evento *join* e *leave*, respetivamente), existe um reajuste e realocação dos pares chave-valor afetados, de forma a que cada par fique sempre associado e guardado no nó sucessor à sua chave.

Na presença de um **evento *join*** são transferidos para o nó que se juntou todos os pares pelos quais este nó passa a ser o responsável, ou seja, quando é o novo sucessor da chave desses pares na ***MembershipTable***. Este pedido é enviado pelos nós que deixaram de ser responsáveis por esses pares chave-valor que possuem.

Na presença de um **evento *leave*** antes de um nó sair do *cluster* este é que é responsável por enviar para o seu sucessor todos os seus pares chave-valor, uma vez que este é que passa a ser reconhecido como responsável desse par chave-valor transferido.

Para a transferência desta informação é usado o protocolo TCP.

Na seção 4.2.2 é descrita a variante utilizada deste protocolo devido à noção de **replicação**.

3.5. Tolerância a falhas

A coordenação entre alteração da *membership* e transferências de pares chave-valor aquando a receção de eventos *membership* é uma preocupação que tem de ser considerada de forma a garantir

que os nós do *cluster* conseguem sempre encontrar o responsável por um determinado par chave-valor e os pedidos que envolvam estes sejam realizados com sucesso. Este problema foi resolvido como consequência da implementação de replicação, uma vez que um par encontra-se disponível em vários nós do *cluster*.

Por outro lado, lidar com possíveis nós que possam estar em baixo é outro aspeto relevante no desenvolvimento deste serviço. Neste sentido são adotadas as seguintes estratégias:

1. Quando um nó é iniciado após um *crash* recupera a informação de todas as chaves dos pares pelos quais era responsável que se encontravam em disco. Assim, ao fazer *rejoin*, após atualizar a sua informação *membership*, volta ao estado *pré-crash* e, se necessário devido às movimentações do *cluster* perdidas, a transferência dos seus pares será realizada com base nesta informação atualizada.
2. Conceito de fila de pedidos pendentes que reúne todos os pedidos falhados após 2 tentativas de um nó comunicar com outro do *cluster*. Indicado pela falta de resposta por parte desse nó requisitado. (***PendingRequests***)
3. Da fila de pedidos pendentes são libertados pedidos quando cada nó associado aos mesmos fizer *rejoin*, ou assim que perceber que o nó está novamente ativo pela receção da sua mensagem de *membership*, com o objetivo de regularizar o estado do *cluster*.
4. Um evento *leave* envolve, para além da transferência de pares chave-valor, a posterior transferência dos seus pedidos pendentes para o próximo nó imediatamente disponível (nó que não tenha pedidos pendentes a ser recebidos por parte do nó que faz *leave*) com o objetivo de que estes pedidos pendentes não sejam perdidos.
5. Se na receção de um evento *join* o nó responsável por transferir para este alguns dos seus pares estiver em baixo, ao recuperar, após atualizar a sua visão do *cluster,* apercebe-se que tem pares pelos quais já não é responsável e trata de transferir estes para os nós apropriados.
6. Se na ocorrência de um evento *leave* o nó recetor dos pares deste estiver em baixo, esses pedidos de transferência são guardados na fila ***PendingRequests*** que é transferida ao próximo nó disponível. Quando o nó recuperar, irá obter por intermédio do atual detentor dos ***PendingRequests*** os pares que lhe pertencem que não conseguiu receber com o *leave* inicial.

4. Replicação

O conceito de réplica tem como objetivo aumentar a disponibilidade e integridade dos pares chave-valor, uma vez que cada par chave-valor passa a estar disponível em 3 diferentes nós do *cluster* (o nó sucessor da chave do par e os 2 nós seguintes), bem como visa a evitar a centralização de informação num único nó e assim diminuir consequências prejudiciais para o sistema em caso de falha de um nó.

Para encontrar as réplicas para uma determinada chave toma-se partido da estrutura usada para

representar a ***MembershipTable*****,** um ***TreeMap***, explicitada na secção 3.2, da seguinte forma:

1. Conhecer o fator de replicação (*n*);
2. Obter o sucessor da chave do par;
3. Obter as 2 entradas na tabela (*n*-1) após o sucessor encontrado (descartando repetidas).

Para a implementação de replicação foram criadas mais 3 mensagens (***replica_put***, ***replica_get*** e ***replica_del***), estruturalmente semelhantes às mensagens *put, get* e *delete*, mas distintas apenas no tipo de operação, de forma a distinguir uma operação da *store* de uma operação de replicação. Tal estratégia foi adotada de forma a que, no processamento das mensagens o nó seja limitado a realizar a ação, sem a possibilidade de retransmitir a mensagem para outro nó.

4.1. Implicações no membership

A replicação exige do protocolo de *membership* uma melhor performance para assim ser mais facilmente garantida a consistência dos dados espalhados pelas diferentes réplicas. Para isso, uma possível solução passaria por diminuir o tempo entre atualizações de cada nó do *cluster*.

4.2. Implicações no storage service

4.2.1 Operações put, get e delete

Nas operações de ***put*** e ***delete*** passa a existir um conceito de ***coordenador***, que corresponde ao nó sucessor da chave do par associado à operação a realizar, sendo da responsabilidade deste garantir a replicação para todos os responsáveis do par em questão. Neste sentido, os pedidos de *put* e *delete* quando não são enviados diretamente para o coordenador são redirecionados para este. Importante realçar que o coordenador utiliza as mensagens de replicação relativas à operação para guardar ou eliminar os pares nas suas réplicas. Caso o suposto nó coordenador esteja indisponível, o próprio nó que recebeu o pedido do cliente assume a coordenação e redireciona o pedido, na forma de réplica, para as 3 réplicas. Neste contexto, aplica-se, igualmente, o conceito de fila de espera de pedidos falhados (***PendingRequests***) descrito anteriormente na seção 3.5. Esta estratégia tem como objetivo regularizar o estado do *cluster* e o número de réplicas de cada par chave-valor no *cluster*.

Nas operações de ***get*** o nó alvo desta operação, caso não seja detentor de uma das réplicas, **sequencialmente** redireciona o pedido, no formato de réplica, a cada um dos nós do conjunto de réplicas da chave recebida. Assim, o nó inicial espera uma resposta, com o valor do par correspondente à chave ou com a indicação de que aquele nó não possui o valor pretendido, dentro de um intervalo de tempo definido.

4.2.2 Transferência de ficheiros com join e leave

Na presença de um **evento *join*** a transferência dos pares chave-valor é da responsabilidade de todos os nós que guardam a antiga terceira réplica de um par chave-valor de quem agora este novo nó também é responsável, mantendo 3 réplicas de cada par em todo o cluster e existindo um reajuste das réplicas de cada par a cada join. No caso especial de um *cluster* ter tamanho inferior ao fator de

replicação, todos os ficheiros são copiados para este novo membro do *cluster*.

Na presença de um **evento *leave*** o nó que pretende sair do *cluster* transfere primeiramente todos os seus pares chave-valor para a nova terceira réplica de cada chave dos seus pares.

A noção de fila de espera de pedidos falhados (***PendingRequests***) é a mesma que a referida na seção 3.5.

5. Tolerância a Falhas

Uma operação de ***put***, ***get*** ou ***delete*** que seja requisitada a um nó que não esteja ativo no *cluster* é considerada indisponível, e respondida com a indicação de que esse nó não se encontra online.

Caso um nó que não esteja atualizado envie o pedido, que lhe foi requerido pelo cliente, para o nó errado, este nó é capaz de avaliar se é ele que tem de processar o pedido e caso não seja consegue simplesmente redirecionar o pedido para o nó que ele pensa ser o correto. Esta solução simples pode levar a que, caso estejam os dois desatualizados, o pedido não seja entregue de forma imediata, mas, sendo esta situação pouco provável e uma vez que estes nós são atualizados frequentemente, rapidamente processará o pedido o que permite concluir que é uma boa solução para o problema de falha.

Outra situação que se pretendeu resolver foi a de, caso um nó estivesse muito tempo ausente, não ter uma completa visão do *cluster*. Para resolver esta situação, entendeu-se que, para completar a sua visão do *cluster*, o nó pretendia obter a lista completa de *logs*, uma vez que a sua visão do *cluster* estaria sempre completa devido à fusão de mensagens *membership*. Assim, a estratégia utilizada implica observar o número de alterações provocadas nos próprios *logs* de um nó aquando da receção de uma mensagem *membership*. Caso o número de alterações fosse máximo (32), então o nó envia por *multicast* uma mensagem a pedir todos os *logs* e faz uma coleta e posterior fusão dos resultados (semelhante ao método ***rejoin*** referido anteriormente mas desta feita com um campo no cabeçalho indicando a pretensão da plenitude dos *logs*).

6. Concorrência

6.1. Thread-pools

Na implementação de *thread-pools* foram utilizados ***ExecutorServices*** de forma a tomar partido da gestão de *threads* que estes já têm adjacente. Como escolha arquitetural, foram usados 2 *executors* com objetivos específicos e distintos: um ***ExecutorService*** para gerir tudo o que era pretendido ser executado de forma paralela e assíncrona e um ***ScheduleExecutorService*** para permitir executar uma tarefa periodicamente, no caso específico do envio periódico da mensagem *membership* caso fosse o nó responsável. À primeira e principal *pool* foi-lhe associado um número de

threads corresponde ao número de processadores físicos e lógicos disponíveis no computador em *runtime* para tirar o máximo partido do paralelismo e ao mesmo tempo não introduzir um *overhead* significativo.

Com esta utilização de *thread-pools* foi possível aumentar a disponibilidade de cada *Store*, uma vez que permitiu aceitar várias conexões ao mesmo tempo, não ficando presa a uma estrutura sequencial que poderia em alguns casos diminuir a velocidade de utilização da própria *Store* para valores não desejados. Assim, foi útil ter a possibilidade de introduzir *Tasks/Jobs* numa fila que seria atendida por uma *thread* do *executor* assim que possível.

6.2. Race Conditions

Para lidar com possíveis *race conditions* foram utilizadas algumas estruturas *thread-safe*, nomeadamente a ***ConcurrentLinkedQueue***, mas também blocos ***synchronized*** em zonas críticas onde era importante manter a consistência dos dados.

7. Conclusões

Um sistema distribuído é um sistema que intrinsecamente está associado a problemas e inconsistências. Qualquer estratégia implementada para a resolução de um desses problemas parece sempre insuficiente e por vezes redundante, podendo no limite levar a um número ainda maior de inconsistências. A construção de protocolos robustos é uma tarefa complicada e fortemente apoiada por um conjunto de mecanismos já estudados e testados que permitem que funcione corretamente. Idealmente, pretende-se assumir desde de início que falhas nunca acontecerão, algo que é irrealista.

Apesar de um sistema distribuído ser característico de falhas, sentimos que pensamos nos problemas e chegamos a soluções que ainda que não perfeitas salvaguardam alguns casos.