Serviço distribuído de escalonamento de tarefas

Hugo Abreu | A76203 João Padrão | A76438
João Reis | A75372

Tolerância a faltas
Universidade do Minho

4 de Julho de 2018

1 Descrição

Com a execução deste trabalho prático, o grupo pretende aplicar os conhecimentos adquiridos no âmbito da unidade curricular de Tolerância a Faltas para, desta forma, fomentar o conhecimento em sistemas distribuídos confiáveis de forma generalista e, mais especificamente em modelos de tolerância a faltas e incrementar o conhecimento no que toca a protocolos de comunicação em grupo, mais especificamente, o *Spread Toolkit*.

Este relatório descreve de forma sucinta e simples a abordagem tomada pelo grupo e respetivas decisões de forma a implementar um exemplo de replicação passiva de um serviço de escalonamento de tarefas tolerante a faltas.

2 Abordagem

Para a realização do trabalho prático o grupo optou por recorrer ao modelo de replicação **Primary-Backup**, a justificação da aplicação deste modelo é baseada no tipo de processamento necessário para a implementação deste escalonador, pois é possível compreender que caso não exista determinismo entre o processamento de cada réplica rapidamente as filas de espera de tarefas podem ficar completamente inconsistentes.

Por esta razão e visto que não existe a necessidade de transparência de falhas para o cliente pois este serve apenas de processador de tarefas e pode efetuar novas ligações quando necessário para pedir mais tarefas ou informar acerca de tarefas completas. Assim sendo o grupo implementou o modelo indicado enviando para as réplicas os pedidos de atualização determinísticos para evitar inconsistências entre filas de réplicas e garantindo que sempre que quando o servidor primário falha, um Backup torna-se primário de forma imediata.

Após a seleção do modelo de replicação o grupo deparou-se com duas opções abordadas na bibliografia, a abordagem *blocking* e *non-blocking* onde, respetivamente, esperávamos por Ack's das réplicas antes de enviarmos a resposta ou então enviávamos a resposta imediatamente ao cliente sem esperar pelos Ack's das réplicas.

Tendo estas duas opções o grupo optou por implementar o método *blocking*, mantendo também em mente a existência de réplicas que podem falhar antes de enviar o Ack. Com esta opção o grupo tenta garantir o máximo de consistência possível, de forma a saber que réplicas efetivamente conseguiram atualizar o seu estado e, como foi o método mais abordado nas aulas da unidade curricular o grupo sentiu-se mais à vontade para o implementar.

3 Implementações

Sendo a abordagem tomada baseada no modelo de replicação passiva, o protocolo de comunicação em grupo corre apenas entre os servidores e, os clientes conectam-se via sockets com o Catalyst.

Desta forma não existe a necessidade de integrar o cliente no protocolo de comunicação em grupo e portanto mantemos os clientes como processos que executam trabalho útil independentes entre si.

3.1 Cliente

A implementação do cliente, para além da Interface de comunicação com o servidor primário, teve como foco a questão do processamento de tarefas.

Foi criado um stub que implementa a interface Task, esta interface contém todos os métodos relevantes para a implementação do serviço do lado do cliente, os métodos implementados são os seguintes:

- getTask()
- addTask(String url)
- completeTask(String url, ArrayList<String> tasks)

Quando é criado o objeto Task, inicialmente o objeto tenta efetuar a conexão ao servidor primário. Todos os métodos implementados efetuam os pedidos ao servidor primário via sockets *Catalyst*, todos os objetos utilizados para efetuar os pedidos são abordados na secção 3.3. Sempre que não é possível utilizar o canal aberto na criação do objeto, isto é, caso o servidor primário tenha falhado, são efetuadas conexões contínuas até uma das réplicas efetuar o *bind* e começar a aceitar pedidos, sempre que a conexão não é possível e, para evitar o excesso tentativas, o cliente efetua um *sleep* de 5 segundos e depois tenta novamente.

Para efetuar o processamento de cada URL de forma a obter mais tarefas para processar, foi utilizado o *jsoup* que permite abrir a identificada pelo URL de forma a processa-la para obter outras ligações que esta possa conter. Desta forma quando um cliente processa um dado URL deve confirmar a sua conclusão, enviando ao servidor primário uma lista com mais tarefas para colocar na queue.

Este processamento está implementado em loop infinito e, quando o servidor não tem tarefas para fornecer, o cliente espera 5 segundos antes de voltar a pedir. Por esta razão existem duas formas de terminar o cliente. De forma normal, isto é, esperando que a Thread termine a execução e quebre o ciclo ou, então, de forma abrupta, onde termina o processo sem qualquer tipo de verificação, simulando uma falha do cliente.

3.2 Servidor

A implementação do servidor foi igual para o servidor primário como para as réplicas, tal como seria expectável, existindo apenas pequenas diferenças entre estes.

O servidor primário recebe pedidos dos clientes e, portanto, é o único que efetivamente efetua *bind* a uma porta. Todas as outras réplicas recebem pedidos do primário para atualizações de estado via *Spread*. Visto que o próprio primário recebe também esses pedidos pois se encontra no grupo, essa verificação é tida em conta e o primário não volta a aplicar o estado recebido que, neste caso, foi enviado por ele próprio.

3.2.1 Primário

O servidor primário é responsável por processar os pedidos e enviar as respetivas atualizações de estado para todas as réplicas, tendo sempre em conta que, as atualizações de estado das réplicas devem ser determinísticas e idempotentes. Para garantir o determinismo, os pedido enviados via *Spread* para as réplicas não são os

mesmos que os recebidos pelos clientes, os pedidos enviados pelo servidor primário contêm mais informação para ser possível efetuar a aplicação de estado de forma determinística tal como é abordado na secção 3.2.3.

No que toca ao processamento de tarefas, isto é, obter, guardar e completar tarefas, o servidor primário é o único que efetua processamento útil, ou seja, é o servidor primário que tem como função processar os itens nas *queues*, colocado-os ou removendo-os. Após o processamento correto de cada operação requerida pelo cliente, o servidor primário efetua o *multicast* da mensagem de atualização do respetivo estado das réplicas, esta mensagem de atualização de estado é preenchida pelo servidor primário com todas as informações necessárias, nomeadamente, os índices das listas onde foram colocadas as tarefas.

Cada cliente, sempre que se conecta a um servidor primário, deverá registar um ID único para ser possível manter registo das tarefas que este se encontra a executar. Este registo serve para manter todas as réplicas com o conhecimento de que cliente executa que tarefa, desta forma, quando o servidor primário falha, os clientes ao ligarem-se à réplica seguinte, registam novamente o seu ID único e dessa forma é possível garantir que a tarefa continua a ser executada. Este ID único é também associado à conexão para se, caso um cliente falhe, o servidor executa o *handler* que apanha o fecho da conexão, com a respetiva associação, consegue verificar que a tarefa atribuída ao cliente não foi executada com sucesso.

Quando o servidor primário falhar e uma das réplicas iniciar o novo primário, existe um *timer* de 10 segundos para registo de ID's únicos. Este mecanismo é abordado e justificado na secção 3.2.2.

3.2.2 Backup

O código utilizado nos servidores de Backup é exatamente o mesmo código que o servidor primário contém, a única diferença é o tipo de pedidos que este recebe.

Sempre que uma réplica inicia é criado um Set de réplicas que já se encontravam na *membership* antes desta, com este Set é possível uma dada réplica saber quando é a sua vez de se tornar servidor primário. Sempre que uma réplica falhar esta é removida do Set e, quando o Set apenas tiver a própria, significa que todas as que seriam primárias antes desta falharam, assim facilmente é possível garantir que existe sempre uma réplica pronta a efetuar o *bind* para tratar dos pedidos.

Cada réplica mantém também um registo das respostas completas e incompletas associadas a pedidos únicos, isto é, cada pedido contém um ID único como já abordado e, todas as réplicas mantém respostas para esses pedidos, quer estejam completos ou não, para quando o servidor primário falhar e, vierem pedidos repetidos, estes não sejam processados novamente e, portanto, é enviada a resposta para aquele ID único que já estava guardada. Este mecanismo serve para complementar a falha do servidor primário antes da resposta ao cliente, deixando o cliente sem saber se todas as réplicas processaram ou não e, por isso, reenvia o pedido.

Quando o servidor primário falhar e uma das réplicas tomar conta dos pedidos se, passado 10 segundos da réplica ter efetuado o respetivo *bind* houver algum cliente que não se tenha registado com o seu ID único, a tarefa que a ele estava atribuída é colocada como incompleta e permuta novamente para as tarefas a aguardar processamento. Este *timer* é necessário pela simples razão de que um ou n clientes podem vir a falhar na mesma altura que o servidor primário, nunca mais se anunciando à réplica e ficando sempre tarefas por completar.

3.2.3 Comunicação em grupo

Para implementar um protocolo de comunicação em grupo entre as diferentes réplicas, o grupo utilizou a *toolkit Spread* abordada nas aulas da unidade curricular.

Na seleção para as primitivas a utilizar nas mensagens enviadas via *Spread*, o grupo chegou a conclusão que necessitaria de garantias de entrega, ou seja, se um dado processo recebe a mensagem m, todos os processos corretos (que não falhem) garantidamente receberam e vão entregar a mensagem m. Isto é importante pois, caso o servidor primário falhe, é necessário que qualquer réplica contenha o estado consistente e, todas elas tenham o mesmo estado e, portanto, terem recebido as mesmas mensagens.

"Reliable delivery of a message m initiated by participant p guarantees that p will deliver m unless it crashes. If m is initiated in configuration C and no configuration change ever occurs, m will be delivered by every member of configuration C. Reliable delivery does not provide any additional guarantees regarding the order in which messages are delivered. Therefore, a participant can deliver a reliable message as soon as it is received." [1].

"The primary-backup replication scheme does not require a TOCAST primitive." [2]

Tendo isto em conta, a primitiva utilizada para efetuar o *multicast* da mensagem via *Spread* foi *reliable*, pois não existe necessidade de total order. Esta primitiva tal como a citação acima indica, garante as necessidade definidas pelo grupo, se um dado processo p entrega a mensagem m então, todos os processos da vista que não falharam inevitavelmente entregam a mensagem. Não existe a necessidade de ordenar totalmente as mensagens pelo que as atualizações são determinísticas e todos os pedidos são apenas processados pelo servidor primário.

Para garantir o determinismo entre réplicas, as mensagens trocadas via *multicast* contêm informações suficientes para garantir a atualização determinística do estado. Estas informações e o tipo de operações são garantidamente determinísticas tal como é mencionado na secção 3.3, nos objetos enviados via *Spread*.

Todos os objeto que inserem nas listas (queues), levam necessariamente os índices processados pelo servidor primário garantindo assim a atualização determinística, já todos os outros objetos de remoção ou obtenção de tarefas, levam o URL específico a remover ou a permutar para tarefas em curso, garantindo assim, mais uma vez uma atualização igual em todas as réplicas.

Quando um servidor inicia (ou está em recuperação duma falha), envia uma mensagem a requisitar o estado a um dos restantes servidores corretos. Esse servidor fica guardado na eventualidade de falhar, e desta maneira recuperar o estado por outra via. Até obter o estado, todos as outras mensagens são postas em *queue* para mais tarde atualizar o estado, ou seja, assim que o estado for recebido, as mensagens em *queue* serão tratadas, tal que, após a execução destas mensagens o estado fica correto e, depois a nova réplica encontrar-se-à disponível.

3.3 Objetos Serializáveis

Para implementar a comunicação entres os diferentes módulos descritos e garantir comunicação entre servidores para o correto funcionamento do respetivo protocolo de comunicação em grupo, foram criados objetos serializáveis para serem enviados via sockets ou via *Spread*.

Estes objetos estão divididos em duas secções. Objetos trocados entre servidores para garantir que as atualizações das réplicas são efetivamente determinísticas e objetos trocados entre o servidor primário e os diferentes clientes, onde os pedidos são determinísticos e o servidor que efetua o processamento, é responsável por colocar informação suficiente nos objetos trocados entre servidores para garantir a atualização determinística das réplicas.

Objetos trocados entre clientes e servidor primários para realizar o processamento:

- ClientUIDReq / ClientUIDRep

  Este objeto serve para cada cliente registar a sua conexão junto do servidor primário, para que, quando este se encontrar a executar uma dada tarefa, seja possível efetuar a respetiva associação. Sempre que o servidor primário morrer, o cliente deve registar novamente o seu ID único à réplica que se encontra a aceitar pedidos, para que a associação com a tarefa seja novamente efetuada. Uma vez que o cliente não se anuncie após 10 segundos, a réplica recoloca a tarefa para ser novamente executada. A associação efetuada é a seguinte:

  ```
  conexão -> UID
  UID -> Task
  ```

- AddTaskReq / AddTaskRep

  Sempre que um cliente deseje adicionar uma nova tarefa para ser escalonada no servidor, deve utilizar este objeto. Este objeto contém, além do ID único do pedido, o URL da tarefa a escalonar.

- GetTaskReq / GetTaskRep

  Para obter tarefas escalonadas no servidor, deve ser utilizado este objeto. O objeto utilizado para efetuar o pedido apenas contém o ID único deste (para garantir que não é processado múltiplas vezes), enquanto a resposta contém, para além do ID único o respetivo URL da tarefa atribuída para ser processada.

- CompleteTaskReq / CompleteTaskRep

  Este objeto tem como objetivo informar o servidor do término de uma tarefa. Para além do URL da tarefa completa, este objeto leva também a lista de novas tarefas que foram encontradas no processamento, para que sejam adicionadas a queue.

Objetos trocados pelo protocolo de comunicação em grupo para atualização de réplicas, confirmações e recuperações:

- Ack

  Este objeto é utilizado para confirmar uma dada operação de replicação. Cada réplica deve enviar este objeto de volta ao servidor primário após a correta aplicação do estado. Este objeto foi criado pois a abordagem tomada baseou-se no modelo *blocking*.

- AddTaskSpreadReq

  Para garantir a adição de tarefas de forma determinística foi necessario criar este objeto que contém a tarefa a adicionar e o respetivo índice onde colocar a tarefa. Este índice é o local que o servidor primário processou após a adição na queue, tipicamente será sempre no fim desta, mas por uma questão de determinismo perante as réplicas é necessário garantir o índice de colocação.

- GetTaskSpreadReq

  Este objeto trocado via *Spread* entre réplicas contém o URL específico que foi retirado pelo servidor primário. Todas as réplicas retiram, não o primeiro da queue, mas sim este URL específico, pois o mais importante é garantir o determinismo, evitanto ao máximo inconsistências.

- CompleteTaskSpreadReq

  Quando um cliente completa uma tarefa o servidor primário assinala como completa e coloca em queue todas as outras tarefas que advieram desta. Ao informar as réplicas desta conclusão, deverá, além dos campos do *CompleteTaskReq*, conter também uma lista com os índices onde cada nova tarefa deve ser adicionada na queue. Tendo a lista de tarefas e a lista de índices onde estas devem ser adicionadas, é garantido o processamento determinístico nas réplicas, isto porque todas elas adicionam as tarefas nos índices processados no servidor primário.

- IncompleteTaskSpreadReq

  A informação colocada nesta mensagem pelo servidor primário contém o URL da tarefa que não foi concluída com sucesso por um dado cliente. Cada réplica deve, de forma determinista, retirar a tarefa da lista de tarefas em curso colocando-a novamente na queue. Além disso, a associação da tarefa com o id único de utilizador é eliminada, visto que este já não se encontra a processar a tarefa em questão.

- RecoverReq / RecoverRep

  Quando um servidor está iniciar, é enviada uma mensagem *RecoverReq* a um servidor correto aleatório. Assim que esse servidor recebe essa mensagem, preenche um *RecoverRep* com as tarefas em execução, em espera, e as respostas em espera e envia ao de volta ao remetente. Este atualiza o seu estado e fica então disponível.

4 Limitações

As limitações atuais do trabalho prático prendem-se apenas pelo limite de tamanho de mensagens do *Spread*, sendo que não foi efetuada uma solução que complemente esta limitação por falta de tempo.

Sempre que alguma réplica num dado momento tenta recuperar estado e, esse mesmo estado exceda 100KB, é lançada uma exceção. Esta limitação poderia ser solucionada ao enviar pedaços limitados de cada *array* por mensagens e, enviar N mensagens, por uma questão de cumprimento de prazos o grupo não teve a oportunidade de implementar tal solução.

5 Conclusão

Com o término da presente fase, é admissível afirmar que a solução desenvolvida está de acordo com os objetivos de aprendizagem da unidade curricular, tendo sido implementados conhecimentos e mecanismos de tolerância a faltas.

Conhecimentos relativos à *toolkit Spread*, protocolos de comunicação em grupo e modelos de replicação foram incrementados, pois todos os membros do grupo tiveram a oportunidade de implementar as suas tarefas no projeto.

As limitações poderiam ter sido solucionadas com um pouco mais de tempo e, após o término deste trabalho prático, foi possível verificar que a abordagem tomada foi até certo ponto correta, dadas as justificações e assunções efetuadas antes da implementação.

A qualidade do trabalho realizado agrada ao grupo, tendo em conta a implementação das funcionalidades previstas e respetiva realização de testes de forma a validar a utilidade do sistema.

Referências

[1] Amy Babay - *The Accelerated Ring Protocol: Ordered Multicast for Modern Data Centers*
`http://www.dsn.jhu.edu/ yairamir/Amy_MSE_thesis.pdf`
Baltimore, Maryland on May, 2014

[2] Rachid Guerraoui & André Schiper - *Fault-Tolerance by Replication in Distributed Systems*
Département d'Informatique, Ecole Polytechnique Fédérale de Lausanne, 1015 Lausanne, Switzerland