Relatório Sistemas Distribuídos

Grupo 43: Lara Faria (106059), Tiago Santos (106329) | Turno L04, Professor Martim Monis

Introdução

Este projeto foi realizado na dificuldade “I am Death Incarnate”, com todas as partes desenvolvidas na totalidade, à exceção da C2 onde não foram implementadas todas as otimizações idealizadas inicialmente.

Desenvolvimento do projeto

B2. De que forma é que o take suporta operações concorrentes e os acessos concorrentes aos tuplos são geridos?

Para estender a operação take a um sistema replicado, implementámos uma solução baseada no algoritmo de exclusão mútua de Maekawa. Alterámos o ficheiro TupleSpacesServer.proto, adicionando duas operações gRPC: enterCriticalSection e exitCriticalSection (que foi otimizada na entrega 3).
Quando o front-end recebe um pedido take, identifica o voter set e envia um pedido às réplicas respetivas para colocarem o tuplo pretendido na secção crítica. Esta secção representa um conjunto de tuplos temporariamente "bloqueados", ou seja, que não podem ser acedidos por outras operações take concorrentes. Utilizamos locks finos, ou seja, apenas o tuplo correspondente ao pedido é bloqueado, permitindo a execução paralela de takes sobre tuplos diferentes (para esta entrega, tuplos iguais eram bloqueados num certo pedido take mas este aspeto foi melhorado no ponto C1)
As operações read e put não são afetadas por esta secção. Caso uma réplica não consiga colocar o tuplo na secção crítica (por já lá estar ou não existir), fica em espera até conseguir fazê-lo.
Após todas as réplicas do voter set confirmarem que o tuplo foi adicionado à secção crítica, o front-end realiza a operação take como nas versões anteriores, enviando o pedido a todas as réplicas e aguardando as respostas. Depois, pede às réplicas do voter set que removam o tuplo da secção crítica e notifiquem as threads em espera. Quando todas confirmam, o front-end envia a resposta ao cliente, concluindo o processo.

C1.i. Como é que as expressões regulares alteram o funcionamento do take?

Para suportar expressões regulares, foi necessário adaptar o comportamento do front-end, das réplicas e a estrutura das operações gRPC.
O front-end envia a expressão regular às réplicas do voter set, que identificam os tuplos do espaço que ainda não estão na secção crítica. Desse conjunto, adicionam à secção crítica apenas os tuplos distintos que correspondem à expressão, evitando bloquear tuplos iguais desnecessariamente, já que apenas um será removido. Cada réplica responde com os tuplos que colocou na secção crítica. O front-end escolhe um tuplo comum a todas; se não existir, pede que removam os tuplos bloqueados e repete o processo até encontrar um tuplo comum. Este é então usado no pedido take enviado a todas as réplicas, como na entrega 2. No fim, os tuplos em secção crítica são libertados, e após todas as réplicas confirmarem o exitCriticalSection, o front-end responde ao cliente.

C2. Como implementaram as otimizações de desempenho propostas no artigo Xu-Liskov?

Com o objetivo de reduzir o número de mensagens trocadas e o tempo que o cliente espera por uma resposta, implementamos as seguintes otimizações:
1- Na operação put, o Front-end responde ao cliente assim que envia o pedido às réplicas, sem esperar pelas respostas. Para isso, alteramos o cliente para que passasse a usar stubs assíncronas no put, permitindo que o front-end retorne onCompleted ou onError após obter resposta de todas as réplicas, sem bloquear o cliente.
2- Juntámos as mensagens take e exitCriticalSection numa única operação gRPC, reduzindo assim o número de mensagens trocadas na operação take com as réplicas.
3- Na operação take, o Front-end responde ao cliente assim que identifica o tuplo a retirar e envia o pedido de take às réplicas. Assim, conseguimos reduzir o delay na resposta ao cliente. Neste caso não optámos por usar stubs assíncronas no cliente, o que fará com que o cliente não consiga receber erros que ocorreram na operação take. No entanto, não só estes erros são raros e de contexto grave que põe em causa todo o sistema e operações seguintes (que irão retornar o mesmo erro), como ter optado por stubs assíncronas iria comprometer o funcionamento da operação take, já que esta, ao contrário do put, retorna o tuplo retirado.
4- Adicionamos um pequeno delay no reenvio de pedidos para entrar na secção crítica para reduzir o número de mensagens trocadas com as réplicas e não sobrecarregar o sistema.

Neste projeto faltou-nos implementar 2 otimizações descritas no arquivo Xu-Liskov:
1- As operações de cada cliente devem ser executadas em cada réplica na mesma ordem que foram enviadas pelo cliente
2- Operações put não podem ser executadas em nenhuma réplica até que todas as operações de take do mesmo cliente tenham sido completadas por todas as réplicas.

Testes-Finais

Teste 1

Para testar a interseção de tuplos entre réplicas do voter set, realizámos um teste com dois clientes e três operações put de tuplos distintos com atrasos de forma a que, os servidores do voter set tenham apenas um tuplo em comum. Assim, ao fazer um take com uma expressão regular que corresponde a todos os tuplos, a interseção resulta num único tuplo (o tuplo <a>), validando a coordenação correta entre réplicas na sua remoção.

Teste 2

Para testar a interseção vazia de tuplos entre réplicas do voter set, realizámos um teste com dois clientes e duas operações put de tuplos distintos, com delays de forma a que, ao fazer um take com uma expressão regular que corresponde a todos os tuplos, a interseção irá retornar vazia. O Front-end ficará a enviar pedidos de entrada na secção crítica às réplicas até que as operações put sejam totalmente realizadas de forma a existirem tuplos na interseção.

Teste 3

Para testar os locks finos, realizámos um teste com dois clientes, duas operações put do mesmo tuplo e dois takes, tendo o primeiro um pequeno delay. Assim, o primeiro take só coloca um dos tuplos na secção crítica, permitindo que o segundo aceda ao outro, validando a execução paralela de takes sobre tuplos iguais mas distintos.