LaTeX Author Guidelines for 8.5×11-Inch Proceedings Manuscripts

Miguel Belm 83531
Tiago Gonalves 83567
Vtor Nunes 83576

Abstract

In this project we will apply two different aproaches to solve the problem of having a distributed tupplespace. These aproaches have different tradeoffs considerations regarding time spent executing a request, network congestion and time spent recovering from a fault. For this work we consider a perfect failure detector, in which if a server stop responding to requests, then is because it crashed.

1. Introduction

A tuple space consists in distributed collection of tuples. A tuple contains fields that are some how related. A basic tuple space needs to be writable, readable and deletable.

Challenges. At a first glace seems easy to implement a tuple space in a distributed way however problems like consistency, data loss and performance rise.

To overcome this challenges we propose two diferent implementations of a distributed tuple space which solve the challenges listed above.

The State Machine Replication (SMR) which uses a coordination mechanism envolving one leader to be responsible for coordination.

And a Xu and Liskov implementation that discards the need of a leader to coordinate client's requests.

2. State Machine Replcation

The State Machine Replication is based on primary backup approach. Given one primary and several secondary nodes, clients will perform requests to the primary node only.

When the first server is switched on, we start by searching for alive servers requesting AREYOUTHEMASTER. If no reply is given then the server becomes the leader. When others servers join the process repeats but they will receive the reply containing the URL path of the master.

2.1. Clients request

In our solution when the client starts we begin with a list of all URL's servers, sends the request to all servers and only the leader will start processing, others simply discard the request.

2.2. Fault tolerance

This implementation has an obvious **week point**: if the master crashes the system will stop repling to requests. Our solution to this problem consists of using an heartbeat that is sent from the secondary nodes to the primary node every random seconds between 3 to 10 seconds. This ensures that in the worst case the system will took ten seconds to recovers (assuming instant propagation of messages) . The randomization process is used to optimize network bandwith and prevent heartbeats flooding on the master.

Every node in the SMR contains an identifier. When a master crashes the remaining node with the smallest ID will be elect as the new master. It starts by informing all secondary nodes about its leadership change. By that time all secondary nodes start sending heartbeets to the new master.

Another problem is when a **secondary node crashes**. In this case the availability of the system is not affected since the primary node continues to receive client's request. Nevertheless if that same node resurrects then it will be unconsistent with others. To fix this problem, every node has a log.

A log is used to keep track of every request made by the master **that was already executed.** When a secondary node crashes and recovers it will have no such tuple in the tuple space, then request to the master a copy of his own log. The master suspend the client's requests processing and replies to the new secondary node. As soon the new node is ready the master continues to processing client's request.

3. Xu and Liskov

De acordo com esta implementao j no se recorre a um master que garante a consistłncia do sistema coordenando todos os pedidos. Em vez disso, cada Front end de cada

cliente vai propagar o pedido para todos os servidores vivos, e cada um desses servidores vai tratar de o executar.

3.1. Clients request

Na nossa soluo, quando um cliente quer realizar uma lista de um ou mais pedidos, vai inicialmente criar um Front End, que vai procurar quais os servidores vivos, e com isso criada a View do cliente, que tem um ID. Quando um pedido enviado para os servidores, anexado ao mesmo o ID da view do cliente, o que permite ao servidor saber se a view do cliente est ou no sincronizada com as views dos servidores, caso no esteja, porque o sistema sofreu uma modificao e quando recuperar da mesma, a nova enviada ao cliente.

3.2. Fault tolerance

Esta abordagem, ao no apresentar um master, tem a vantagem de que quando um n crasha, seja ele qual for, o sistema no precisa de parar. Isto possvel pois no existe uma unidade central da qual todo o sistema dependa, e que caso a mesma crashe, todo o sistema comprometido. Apresenta uma desvantagem em relao ao SMR que a de precisar de parar o sistema por breves instantes quando um novo n se liga ao sistema. Isto feito pois as Views precisam de ser atualizadas para passar a conter o novo n, e no se podem perder pedidos que estejam a ser feitos enquanto as Views so atualizadas.

Quando um n crasha o cliente vai atualizar a sua view, e informar todos os seus membros para atualizarem a deles. Quando um n rescuscita, vai comear por informar todas as mquinas vivas de que precisa do seu espao de tuplos e vai pedir para que atualizem as suas views. Cada mquina ao receber este pedido vai recusar pedidos dos clientes que apresentem uma view antiga e vai dar o seu tuplespace ao novo n. O novo n vai fazer uma interseo de todos os tuplespaces (a interseo garante que ele apenas vai guardar os pedidos que j foram executados em todas as mquinas). Em seguida esse tuplespace propagado para todo o sistema, com isto garantimos que nunca se perdem pedidos, pois mesmo que um pedido j tenha sido enviado para uma mquina mas no para outra, vai acabar ser repetido. Aps todos os tuplespaces terem sido atualizados, o novo n sinaliza as outras mquinas para que mandem a sua nova view ao cliente, repetindo o mesmo todos os pedidos da mesma operao que foram enviados numa view antiga e que consequentemente foram recusados pelos servidores.

4. Evaluation

Nesta seco vamos analisar e avaliar como cada algortmo se comporta no que toca a alguns pontos chaves. Apresentamos ainda a justificao para algumas escolhas que tiveram de ser feitas.

4.1. Requests execution

4.1.1 SMR

Para a operao de Take e Write necessrio que os pedidos cheguem ao master e depois este propaga para o resto dos servidores/rplicas. Isto alm de sobrecarregar um servidor pois ele que tem de receber a transmitir os pedidos e posteriormente esperar pelas respostas dadas pelas rplicas tambm atrasa a respostas aos pedidos pois preciso que os mesmo passem por um intermedirio. Na operao de Read os pedidos no so propagados pera as rplicas, devolvendo o master instantneamente a resposta ao cliente, caso seja possvel (Take e Read so operaes bloquentas, para ser dada uma resposta preciso que a mesma conste nos espaos de tuplos).

4.1.2 XL

Para a operao Write o cliente propaga o pedido para todos os servidores. Para a operao de Read o cliente propaga o pedido para todos os servidores e devolve ao cliente quando obtiver a primeira resposta. Para a operao de Take foram tidos em conta alguns cuidados, nomeadamente: necessrio bloquear certas entradas do espao de tuplos e convem que esta parte seja feita de uma maneira que no obrigue a que o cliente fique muito tempo espera de uma resposta assim sendo o algortmo usado foi o de tentar bloquear tantas entradas que dłm match com o pedido quanto possvel, e quando no for possvel adquirir o trinco de uma dada entrada, o conjunto de entradas que cujo trinco foi adquirido sao devolvidas ao front end, em seguida o front end vai intersetar os conjuntos que recebeu dos vrios servidores e vai escolher um elemento comum a todos esses conjuntos e remove-lo, devolvendo ao cliente. Esta foi a melhor soluo encontrada, pois usamos uma heurstica que tenta devolver o mximo de entradas possvel no entanto no fica muito tempo espera de modo a conseguir todas a entradas possveis (se existirem muitos pedidos simultneos e todos fizerem isto, o que vai aconter o servidor no conseguir dar resposta a ningum e ficar muito lento), esta heurstica tambm no se contenta apenas com uma resposta (caso s fosse devolvida uma entrada, na parte da interseo, a probabilidade de a mesma ser nula seria muito elevada). Com esta heurstica conseguimos ter algum equilbrio.

4.2. Comparison

Para a operao Write expectvel que o XL a execute mais rapidamente. Para a operao de Read expectvel que ambas as implementaes tenham um tempo de execuo semelhante.

Para a operao de Take se a mquina master tiver uma capacidade de processamente muito elevada (no entupir com um elevado nmero de pedidos), acabamos por conseguir ter tempos de resposta mais baixos pois no ficamos dependentes da operao de interseo, que pode falhar, tendo de ser repetido todo o processo.

4.3. Network congestion

4.3.1 SMR

4.3.2 XL

4.4. Fault recovering

4.4.1 SMR

4.4.2 XL

5. Summary and conclusions

No algortmo SMR no so usadas Views, provocando uma latłncia enorme em todas as comunicaes servidor servidor e cliente servidor, pois necessrio estar sempre a verificar que mquinas esto vivas. Uma modificao futura a este trabalho seria implementar um sistema de Views semelhante ao existente no XL. No possvel dizer qual dos algortmos melhor, ambos apresentam vantagens e desvantagens, ser necessrio ver com ateno os tpicos referentes Evaluation, e conforme o intuto do sistema a ser desenvolvido, aplicar o que melhor se adeque.

References

[1] I. M. Author. Some related article I wrote. *Some Fine Journal*, 99(7):1–100, January 1999.

[2] A. N. Expert. *A Book He Wrote*. His Publisher, Erewhon, NC, 1999.