LaTeX Author Guidelines for 8.5×11-Inch Proceedings Manuscripts

Miguel Belém
83531

Tiago Gonçalves
83567

Vítor 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.

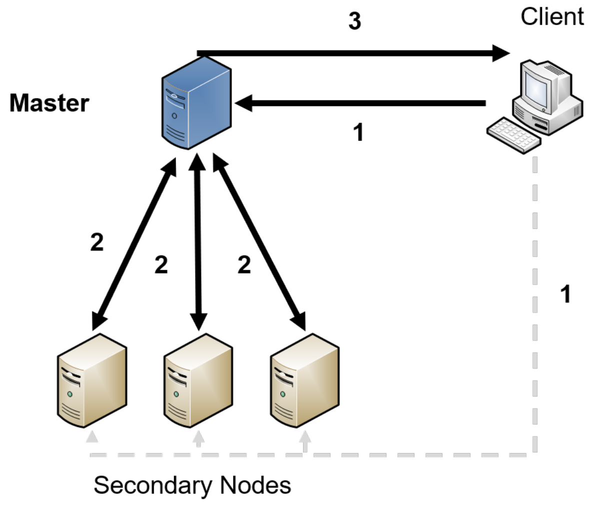

Figure 1. SMR basic communication schema

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.

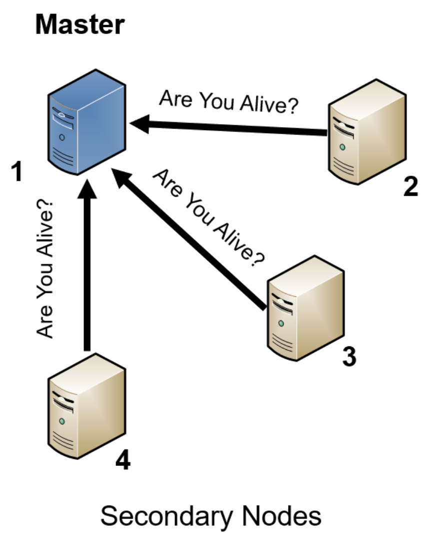

Figure 2. SMR master failure detector

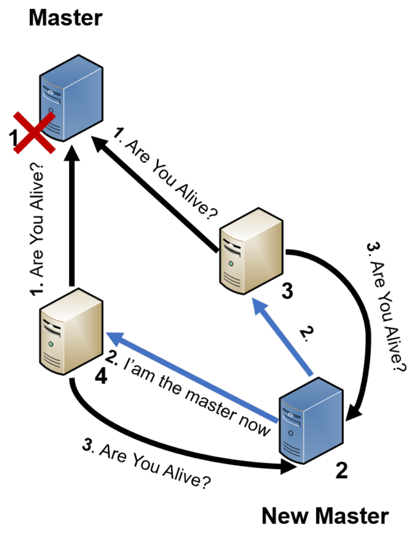

Figure 3. SMR master failure and system's recovering process

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 implementação já não se recorre a um master que garante a consistência 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. A Client's Front End is an interface to coordinate request to all nodes and made them transparent to the user. For example, user invoke Front End's take operation and FE is responsible for getting the current view and propagate the request to multiple nodes.

3.1. Clients request

In this approach, when a client wants to perform some operation, stats by creating a Front End. The client reads the server's URL path from the file and asks for the view.

When a request READ, WRITE or TAKE is sent to the servers, it is checked server-side if the view is up-to-date. The server will only execute the request if the view is the most recent. Otherwise simply returns null and forces the client to update its view first and repeat the request.

Every view contains a **version** which allows compare two views and determine the most recent view. When a client starts up we gets one view from the server before making requests to the client.

The **biggest advantage** over our SMR implementation is that the client will only send requests to servers that are supose to be alive.

3.2. Fault tolerance

This approach has an advantage of time in recovering, this happens because when a node crashes the system don't need to stop. There are no coordinator node (a central point of failure) that could compromise the system availability.

However has the following **pitfall**: when a new node joins it is necessary to pause the entire system to performe a

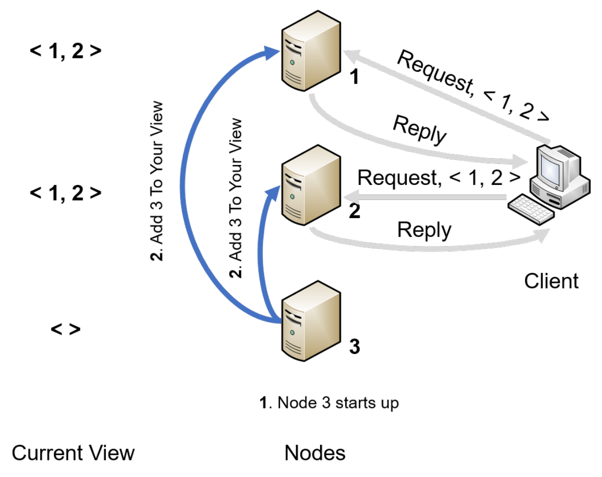

Figure 4. XL view update process

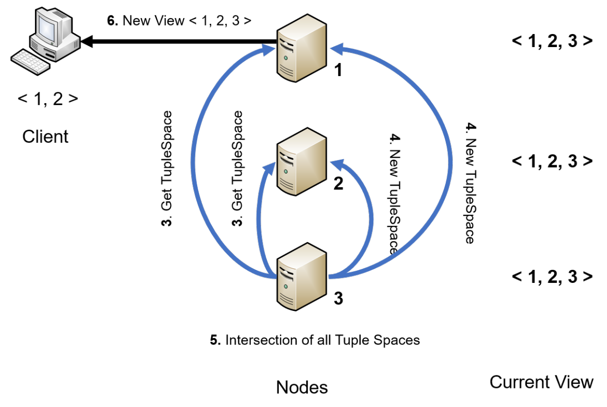

Figure 5. XL tuple space recover

view update in order to ensure consistency and garantee that the clients request during the view update will not disapear.

The following steps are executed when a node crashes:

- Client suspend all requests
- Client detects the crash and inform all other view members
- Client resume the requests

Quando um nó crasha o cliente vai atualizar a sua view, e informar todos os seus membros para atualizarem a deles.

The following steps are executed when a new node joins the system:

- New node starts up and for each path in file tries to update the view to inform his arrival
- As soon the already alive nodes receive a request to change the view they perform the request immediately and **do not send the new view to the client.** (Figure 4)
- All client's request are reject because the client view is out-of-date.
- New node request a copy of the tuple space to all already alive nodes.
- New node performs a tuple space intersection (see more in detail above) and propagates the changes to all already alive nodes. (Omitted on Figure 4 due to preserve draw simplicity)
- Only when the new tuple space is sent to all nodes then the client is informed of the new view. (Figure 5)
- The client can resume the requests to all nodes.

Tuple Space Intersection: We defined a intersection of tuple space as the set of tuples that belongs to both tuple spaces.

Let's supose, for example that we have **2 nodes already alive** and a new node joins the system. The following schema shows the tuple space content of each node:

$TupleSpace_1 \leftarrow < dog, brown >, < cat, white >, < cat, gray >$

$TupleSpace_2 \leftarrow < dog, brown >, < cat, white >, < cat, gray >, < horse, white >$

$TupleSpace_3 \leftarrow \emptyset$

A closer look to $TupleSpace_2$ shows a partial write operation, this is the client's front end wrote on node 2 but lost his view and therefore couldn't wrote on other nodes.

Remember that node 3 will request all tuples spaces (from node 1 and 2) and perform a intersection, this is:

$TupleSpace_3 \leftarrow TupleSpace_1 \cup TupleSpace_2$

$TupleSpace_3 \leftarrow < dog, brown >, < cat, white >, < cat, gray >$

At a first glance seams like the new tuple $< horse, white >$ has been lost due to the expiration of the view. Nevertheless the client needs to perform the request again, and because of this the tuple will be added again.

This **garantees consistency** as the partial writes/takes will be discarded and repeated as complete write/takes operations!

4. Evaluation

In this section we will study how these approachs (SMR and XL) performs in some metrics. We still present some justifications about concrete implementation aspects.

4.1. Requests execution

4.1.1 SMR

For TAKE and WRITE operations it is necessary that the master receive the requests and then propagate it tho secondary nodes. The biggest disadvantage of this approach is the extra load that a master needs to handle, this is the client's and nodes requests/replies and even heartbeets.

READ requests are the only type of client request that the master won't propagate to other nodes. As soon the master receives the READ it respondes immediately to the client.

4.1.2 XL

For the WRITE operation the client's front end propagate the request to all alive servers.

The READ operation is performed by the front end and returns to the client the first response it gets.

The TAKE operation is a little bit more tricky, in the sense that has two phases:

TAKE Phase 1

- Node receives a specific tuple object to be taken from the tuple space.
- Returns a list of all matching tuples and locks them on a lock list (see more details above)
- The client's front end performes a intersection operation on all responses of all nodes and sends the result back.

TAKE Phase 2

- Node receives a specific tuple object to be taken from the tuple space.
- Removes that tuple.
- Unlocks all other tuples from the lock list.

Lock List: It is a log that creates a mapping between the client Id and the tuples that are locked to that client during the TAKE Phase 1.

Every tuple can only be locked to a user at the same time.

Esta foi a melhor solução encontrada, pois usamos uma heurística que tenta devolver o máximo de entradas possível no entanto não fica muito tempo à espera de modo a conseguir todas a entradas possíveis (se existirem muitos pedidos simultâneos e todos fizerem isto, o que vai aconter é o servidor não conseguir dar resposta a ninguém e ficar muito lento), esta heurística também não se contenta apenas com uma resposta (caso só fosse devolvida uma entrada, na parte da interseção, a probabilidade de a mesma ser nula seria muito elevada). Com esta heurística conseguimos ter algum equilíbrio.

4.2. Comparison

Para a operação Write é expectável que o XL a execute mais rapidamente. Para a operação de Read é expectável que ambas as implementações tenham um tempo de execução semelhante. Para a operação de Take se a máquina master tiver uma capacidade de processamente muito elevada (não entupir com um elevado número de pedidos), acabamos por conseguir ter tempos de resposta mais baixos pois não ficamos dependentes da operação de interseção, que pode falhar, tendo de ser repetido todo o processo.

4.3. Network congestion

4.3.1 SMR

Quando é feito um pedido de execução de uma operação, este é propagado para todos os servidores. Aqueles que são réplicas ignoram o pedido (só executam pedidos do master), o master quando recebe vai propagá-lo para as réplicas. A juntar a isto temos ainda mensagens a serem trocadas a cada 3 a 10 segundos de cada uma das réplicas para o master, para verificar se o mesmo não crashou.

4.3.2 XL

No normal desenrolar do sistema, o pedido é propagado para todos os servidores na View do Front End do cliente, por norma os pedidos são feitos uma única vez. Exceptuando os casos em que por algum motivo a interseção na operação Take falha, ou quando um novo servidor entra no sistema e o mesmo tem de parar para as Views serem atualizadas, pedidos enviados durante essa altura teem de ser repetidos.

4.3.3 Comparison

A implementação SMR congestiona muito mais a rede que a implementação XL. XL poderá congestionar mais a rede que o SMR no caso em que a View tenha de ser alterada, existam muitos pedidos a ser feitos e todos vão ter de ser repetidos.

4.4. Fault recovering

4.4.1 SMR

No caso de existir um crash numa das réplicas, o sistema não fica comprometido e pode fluir normalmente. No caso de ser o master a crashar os pedidos dos clientes vão para um buffer e não são executados até um novo Master ser eleito. Para eleição do novo master é usado um algoritmo de leader election que escolhe a réplica que tenha um ID mais baixo.

4.4.2 XL

Quando alguns dos servidores crasha o sistema não para. Simplesmente quando um cliente deteta isso propaga a nova View para os servidores e deixa de fazer pedidos a quem crashou.

4.4.3 Comparison

Para o SMR quando o Master crasha o sistema preciso de algum tempo para recuperar. Para o XL isso nunca acontece. Por outro lado no caso do XL, quando uma nova máquina se liga, o sistema tem de parar por causa da operação de interseção. Dependendo do sistema que se pretende implementar pode ser mais vantajosa uma ou outra abordagem. No caso em que novas máquinas se estejam sempre a ligar o SMR reage melhor, no caso em que existam crashs constantes o XL reage melhor porque não possui nenhuma unidade central de coordenação.

5. Summary and conclusions

In the SMR implementation we didn't use Views, leading to a latency growth between all server¡-¿server and client¡-¿server communications. This occurs because we need to check the availability of all servers contained in the servers file, which contains both the alive and death servers. One further modification to ours SMR implementation is to apply Views like the ones used in XL. Another problem that should be fixed in SMR is the one regarding the constant messages exchange from all Replicas to the Master. It's not possible to say which implementation is better, it is necessary do read carefully the evaluation topics written in the previous section, and based on which are the the requirements for the new system, make a decision.

Alguns cenários exemplificativos: -Sistema 1 -¿ Para o caso dos minutos finais de um leilão queremos ter um elevado throghput, nessa altura não deverão existir máquinas novas a quererem ligar-se ao sistema e caso alguma máquina crashe, o sistema não pode ficar bloqueado. Para este cenário será muito mais vantajoso a utilização do XL, pois a rede não é tão congestionada e os pedidos são executados mais rapidamente. -Sistema 2 -¿ Para uma arquiterura Peer to Peer, na qual novas máquinas estão sempre a existir vai ser mais lucrativo utilizar o SMR pois o sistema não bloqueia com a entrada de novos nós.

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.