# DIDA-TUPLE

Miguel Belém 83531
Tiago Gonçalves 83567
Vítor Nunes 83576

## Abstract

*In this project we will apply two different approaches to solve the problem of having a distributed tupple space. These approaches 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 it's because it crashed.*

## 1. Introduction

A tuple space consists in a distributed collection of tuples. A tuple contains fields that are somehow 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 clients requests.

## 2. State Machine Replcation

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

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.

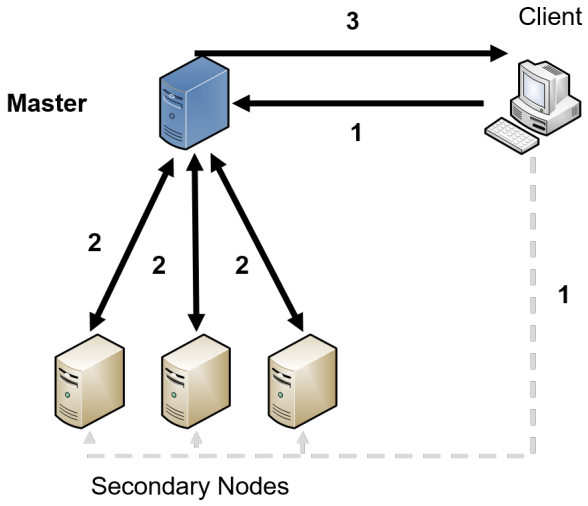


**Figure 1. SMR basic communication schema**

### 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, the others will simply discard the request.

### 2.2. Fault tolerance

This implementation has an obvious **weak point**: if the master crashes the system will stop replying to requests. Our solution to this problem consists on 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 take ten seconds to realise (assuming instant propagation of messages). The randomization process is used to optimize network bandwidth and prevent heartbeats flooding on the master.

Every node in the SMR contains an identifier. When a master crashes the remaining nodes with the smallest ID will be elected as the new master. It starts by informing all secondary nodes about it's leadership change. By that

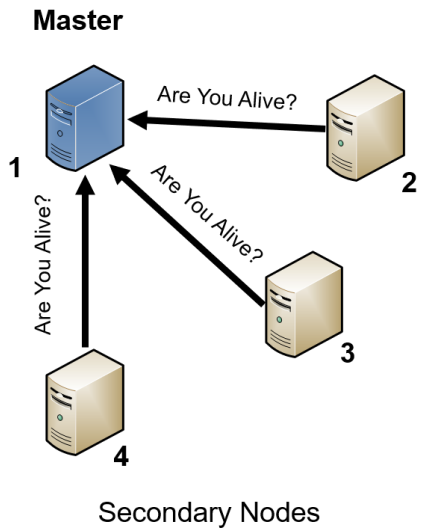


**Figure 2. SMR master failure detector**

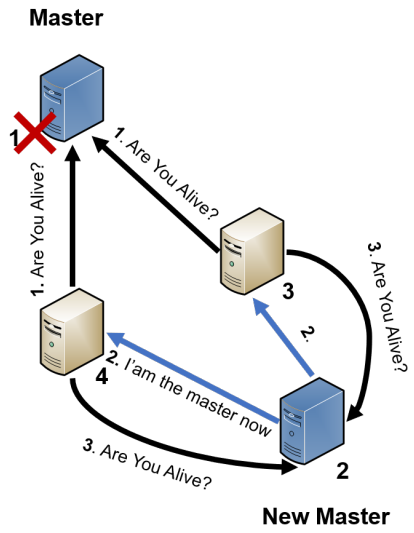


**Figure 3. SMR master failure and system's recovering process**

time all secondary nodes start sending heartbeats 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 requests. 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 tuples in the tuple space and makes a request to the master for a copy of his own log. The master suspends the clients requests processing and replies to the new secondary node. As soon the new node is ready the master continues processing client's request.

## 3. Xu and Liskov

In this implementation we no longer need one master to ensure the consistency of the system. Instead we introduce the concept of client's front end and the concept of view. Note that Front end concept already exists in SMR but in XL it will have extra functions, like coordinating the servers Views, since the servers don't communicate.

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 operations, stats by creating a Front End. The client reads the servers URLs 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 (synchronized with the one in the front end). 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 comparing two views and determine the most recent one. When a client starts up it gets one view from the server before making requests to it.

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

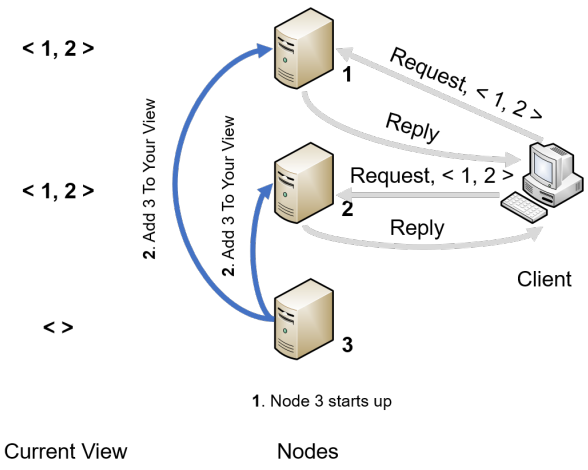


**Figure 4. XL view update process**

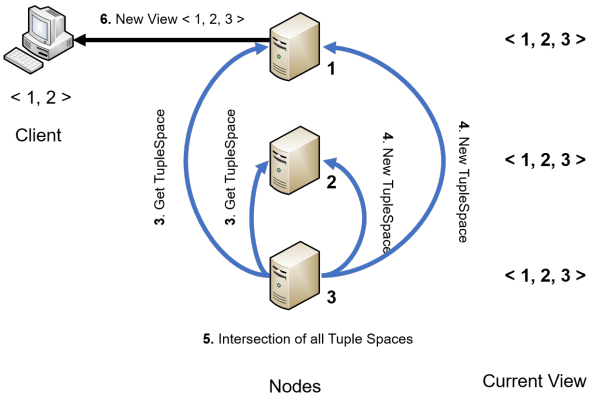


**Figure 5. XL tuple space recover**

### 3.2. Fault tolerance

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

However it has the following **pitfall**: when a new node joins it is necessary to pause the entire system to perform a view update in order to ensure consistency and guarantee that the clients requests during the view update will not disappear.

The following steps are executed when a node crashes:

- Client suspends all requests
- Client detects the crash and informs all other view members
- Client resumes the requests

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

- New node starts up and for each path in the file tries to update its view informing his arrival
- As soon as the 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 rejected because the client view is out-of-date.
- New node requests 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 alive nodes. (Omitted on Figure 4 to preserve draw simplicity)
- Only when the new tuple space is sent to all nodes the client is informed of the new view. (Figure 5)
- The client can repeat the requests to all nodes.

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

Let's suppose, for example that we have **2 nodes 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, the client's front end wrote on node 2 but lost his view and therefore couldn't write on other nodes.

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

$TupleSpace_3 \leftarrow TupleSpace_1 \cup TupleSpace_2$

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

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

This **guarantees 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 approaches (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 receives the requests and then propagates them to the secondary nodes. The biggest disadvantage of this approach is the extra load that a master needs to handle, like clients and nodes requests/replies and even heartbeats.

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 responds immediately to the client.

### 4.1.2 XL

For the WRITE operation the client's front end propagates 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 it 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.

This lock solution has a **trivial problem**: two clients can be dead locked with one single tuple item. To overcome this situation we use the following criteria to lock matched tuples:

- Try to lock as many tuple items as possible
- If it is not possible to acquire the lock on a specific tuple just ignore that one
- Return to the client's FE a list containing only lock-aquired tuple items

**Simple Deadlock problem:** Supose that two client $client1$,$client2$ request a $Take(< cat, * >)$ :

$TupleSpace_1 \leftarrow < dog, brown >, Locked_{client1}(< cat, white >), Locked_{client2}(< cat, gray >)$

$client1$ will wait to lock $< cat, gray >$ and vice-versa.

**Our approach:** $TupleSpace_1 \leftarrow < dog, brown >, Locked_{client1}(< cat, white >), Locked_{client2}(< cat, gray >)$

$client1$ will return the list $< cat, white >$ and $client2$ will return the list $< cat, gray >$.

If one of the clients returns a empty list then the **TAKE Phase 1** is repeated.

If there are too many matching tuples in the tuple space and too many clients then the possibility of having an empty list, i.e, no unlocked matching tuples are available is high.

## 4.2. Comparison

WRITE operation in Xu and Liskov implementation tends to be faster than SMR, because it doesn't exist a master that needs to retransmit all the requests to all the replicas.

READ operations are supposed to be similar in performance as the system returns the first result to the client.

TAKE operation on SMR tends to be more efficient than XL. This occurs because we do not depend on the result of intersection operation, in which a request has a not so low probability of failing and being repeated. We are considering that the master isn't the bottleneck of the system.

### 4.3. Network congestion

#### 4.3.1 SMR

When a client makes a request to a SMR system the client sends the request to all nodes because the master's identity is unknown.

All secondary nodes discard the request immediately. The master node sends the request to the secondary nodes. For each client request only one request is useful, nodes keep checking master's liveliness every random seconds (between 3 and 10). Compararing with XL, SMR is more network intensive.

#### 4.3.2 XL

Following the normal flow of the system, the request is broadcasted to all the machines contained in the client front end view, normally the requests are broadcasted once. The excepetion is when for some reason, the take intersection operation fails, or when a new machine connects to the system and the system needs to stop to update the Views. All the requests sent while the system is stopped need to be repeated.

#### 4.3.3 Comparison

The SMR implementation congests much more the network then the XL implementation. The XL might congests more when the View needs to be updated while a lot of requests are occurring.

### 4.4. Fault recovering

#### 4.4.1 SMR

If a (secondary) node crashes then the system can continue to receive client's requests. Nevertheless when the master crashes, client request will be suspended until the new master election takes place.

The node with the smallest id is elected as the the new master.

#### 4.4.2 XL

With this implementation when a server crashes, the system doesn't need to stop. The client just detects that a change occured in his view and broadcasts the new view to the servers.

#### 4.4.3 Comparison

In the SMR, the worst pitfall is recovering from a master failure. In the XL the worst pitfall is to add a new node to the system and compute an intersection.

According to the context it is more reliable to choose an implementation wisely.

## 5. Summary and conclusions

In the SMR implementation we didn't use Views, leading to a latency growth between all server $\Leftrightarrow$ server and client $\Leftrightarrow$ 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 to 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.

Some context examples:

- **Example 1:** Supose a limited sale system. In the final minutes is required that the system has a bigger throughput, and is very unlikely that new nodes will join the system. For this scenario a XL approach would be much more effective as the network bandwith is low in contrast with SMR.
- **Example 2:** In a Peer-2-Peer architecture, i.e, when there are always machines joining and leaving the network, the SMR approach tends to be better than SMR because the system doesn't need to stop when a new node appears.

## References


[1] E. A. Alysson Neves Bessani. *A Guided Tour on the Theory and Practice of State Machine Replication*.

[2] B. L. Andrew Xu. A design for a fault-tolerant, distributed implementation of linda. (7), January 1989.