DAD Final Report

André Fonseca
n^{o} 84698

Leonor Loureiro
n^{o} 84736

Sebatião Amaro
n^{o} 84767

Abstract

This report explains the idea and implementation of DIDA-TUPLE, a project of Distributed Application Development's subject. The objective is to implement and compare 2 variants of a distributed fault tolerant tuple space system by their performance. The report details the various aspects of each implementation, their performance evaluation and the test cases for which each version performs better.

1. Introduction

A tuple space is an implementation of Content-addressable memory for parallel/distributed computing, provides a storage of tuples that can be accessed concurrently.

DIDA-TUPLE is a project that aims to implement and compare 2 variants of a distributed fault tolerant tuple space system, providing strong guarantees on the reliability of the distributed computation and on bounded performance in the presence of failures.

DIDA-TUPLE-SMR following SMR (State Machine Replication) - a general method for implementing a fault-tolerant service by replicating servers and coordinating client interactions with server replicas. State Machines start in the same state (an empty tuple space), and receive the same inputs in the same order, acting deterministically to to each event.

DIDA-TUPLE-XL following XL (Xu and Liskov) - replication technique takes advantage of the semantics of a Tuple Space so that processes encounters little delay in accessing the tuple space.

For each of the 2 implementation we implemented and analyzed 2 variants: **Basic** version which assumes a perfect failure detector and an **Advanced** version that does not assume a perfect failure detector.

We begin in section 2. by detailing our proposed solution and implementation. Section 3. and 4. explain the 2 different implementations and their respective variances. Section 5 briefly explains Failure Detection and View Synchronization. Section 6 is the evaluation of the performance of the proposed solution implementation.

2. Proposed Solution

We implemented the systems using **C#**, in conjunction with the **.Net** framework.

2.1. Structure

The project is composed of four modules: Client, Server, CommonTypes, PuppetMaster and ProcccessCreationService.

In the **CommonTypes** are the interfaces implemented in the Client and Server modules, that are available for any entity so that it can be fetched using .Net Remoting. It also contains the wrapper classes that represent a tuple, a view and the messages exchanges between the server and the client.

The **Client** module contains the implementations of each version of the DIDA-TUPLES API and the classes responsible for parsing the client scripts. The AbstractClient implements the methods that are common between the four different versions, mainly the methods that deal with the view synchronization.

The **Server** module contains: the TSpaceStorage that represents a tuple space, containing a list will all the tuples, and implementing the operations that can be done over a tuple space; the TSLockHandler manages the tuples' locking for the take operation, in the DIDA-TUPLE-XL; the TSpaceManager and the TSpaceAdvManager implement the common methods for the DIDA-TUPLE service for the basic and advanced versions, respectively; the TSLog stores a log of all the requests processed by the server and the corresponding answers.

The **ProcessCreationService** is responsible for creating client and server processes as described in the project's assignment.

The **PuppetMaster** implements the PuppetMaster has described in the project's assignment.

3. DIDA-TUPLE-SMR

In this variation, the replication of operations is based on the State Machine Replication model. To implement the message deliver mechanism that ensures that all servers execute operations in the same order we followed the ISIS algorithm. The client periodically re-sends each request until it has received answers from all servers in the current view.

3.1. ISIS Algorithm

The client sends a request for a proposed sequence number to all servers, identified by a message id, to which they reply with the highest sequence number they have seen so far and add request message to a queue sorted by sequence number, setting it as *undeliverable*. Upon receiving all suggestions, the client selects the highest sequence number proposed, referred to as agreed sequence number and re-send the message with it. We chose to keep this logic in the client side, because they never fail, meaning there won't be pending messages in the queue that will never be processed, which would happen were the client that sent them fail. The servers updates the sequence number for the corresponding message, setting it as deliverable and reorders the queue. Once a deliverable message is at the head of the queue then the corresponding operation is executed.

3.2. Advanced

In the advanced version, a perfect failure detection is no longer assumed. All servers and clients requests now need only a majority of replies to progress.

4. DIDA-TUPLE-XL

This variant follows the implementation proposed by Xu and Liskov.

4.1. Xu-Liskov Algorithm

All operations are implemented by sending a request to all servers in the current view. When a server receives the **add** request, it adds the tuple into its' space and acknowledges received. The clients re-send the requests until all servers have acknowledged. When a server receives the **read** request, searches for a matching tuple and returns the first match. While no match is found, the client repeats the request. The client waits only for the first answer that is not empty. The **take** operation is split into two phases, take1 and take2. When a server receives a **take1** request, it finds all the tuples matching tuples, locks those tuples and returns a list with locked tuples to client. When the client receives answer from all servers, calculates the intersection between the received tuples, and if the intersection is not empty, chooses one tuple and sends **take2** with the chosen tuple to all the servers in view. If the intersection is empty, the client sends a request to all the servers to release the tuples locked by the client. When the server receives a **take2** operation, the server deletes the received tuple and releases all other tuples locked by the client. Until the take operation is successful the client will repeat this process.

4.2. Advanced Version

In the advanced version, a perfect failure detection is no longer assumed. All servers and clients requests now need only a majority of replies to progress.

5. View Synchronization and Failure Detection

The view synchronization and failure detection varies between implementations

5.1. SMR and XL Basic

The servers periodically poll all other servers in the current view to ensure they're still alive. If a server does not respond, its' failure is assumed.
The view synchronization is handled by the servers. When a server failure is detected, the view is updated by removing it. When a new server wishes to join the distributed tuple space, it connects to another server, which then temporarily holds all incoming requests, informs all other servers in the view of the newcomer, updating the current view and copies its state onto the new server.
When the server receives a request in an old view, replies with bad view message informing the client of the updated view. The client then updates its current view and re-sends the request in the new view. If the server receives a request that has already processed in a previous view, then returns the same answer, which was stored the log.

5.2. SMR Advanced

As the basic version each server periodically checks the connection to all servers in the view. When no answer is received in the defined timeout, then it suspects that the server has failed. Then, it will query all other servers in the view to determine if a majority suspects the server's failure. If the majority is achieved, the server is assumed to have failed.
When a server is assumed to have failed, it is removed from the view and all the other servers are informed of the update.
As the basic version each server periodically checks the

connection to all servers in the view. When no answer is received in the defined timeout, then it suspects that the server has failed.

Then, it will query all other servers in the view to determine if a majority suspects the server's failure. If the majority is achieved, the server is assumed to have failed.

5.3. XL Advanced

The failure detection in XL advanced version was implemented in the same as in the SMR advanced version. The synchronization in XL advanced differs from SMR advanced only in that view synchronization operations are not done in total order.

6. Evaluation

Our solution was tested in a single machine, but it could be deployed and should work in multi-machine environments, however we were unable to test this.

We tried to find the workloads for each of the implementations performs best and designed clients that test those situations.

The XL performs better in a scenario were there aren't many concurrent takes. The basic version will perform better if there are many changes to the view, because the failure recovery time is shorter in the basic version, as the advanced version will have to contact all other replicas and query them about the suspected failure and await the responses from the majority. However, if the view is stable, the advanced version will perform better because the clients only need the majority of servers to answer in order to progress. This will be more accentuated if a minority of servers are significantly slower when responding.

The SMR performs better when there are many concurrent takes for the same tuples because the XL will lock all matching tuples, preventing one (or more) of the clients from progressing. The SMR advanced will perform worse than the basic version when the view changes more frequently, since view update operations are done in total order and require a majority of votes, and no requests can be processed while the view is updating.

6.1. Client Designs

This section will describe the tests we designed that showcase the situations described above. Each test was executed three times.

6.1.1 XL Basic

In this test we launch 3 servers and 3 clients. These clients perform a sequence of add, read and take operations over the same object (each client of a different object). Then we alternate between crashing a server and adding another one, with an interval of 1 min between each operation, twice.

SMR Basic

	Average Time (mm:s,mm)
Client1	05:36.5
Client2	05:36.4
Client3	05:36.1

XL Basic

	Average Time (mm:s,mm)
Client1	03:53.7
Client2	03:53.6
Client3	03:53.3

SMR Advanced

	Average Time (mm:s,mm)
Client1	05:59.3
Client2	05:59.4
Client3	05:59.3

XL Advanced

	Average Time (mm:s,mm)
Client1	04:11,5
Client2	04:11,2
Client3	04:11,2

6.1.2 XL Advanced

In this test, we alternately launch 7 servers and 3 clients. These clients perform a sequence of add, read and take operations over the same object (each client of a different object). No server is removed from the view once it's launched.

SMR Basic

	Average Time (mm:s,mm)
Client1	04:30.2
Client2	04:29.9
Client3	04:30.1

XL Basic

	Average Time (mm:s,mm)
Client1	03:06.6
Client2	03:07.1
Client3	03:07.1

SMR Advanced

	Average Time (mm:s,mm)
Client1	04:34.3
Client2	04:33.2
Client3	04:33.6

XL Advanced

	Average Time (mm:s,mm)
Client1	03:04.6
Client2	03:05.7
Client3	03:05.1

6.1.3 SMR Basic

In this test, we launch 6 servers, followed by a client that will perform 300 adds, in order to fill the tuple space. We wait 3min to ensure the first client finishes execution before launching 3 clients that will perform 100 takes each, which will match any tuple in the tuple space, to simulate the concurrency similar takes. We wait 15 seconds to allow the system to stabilize, before crashing a server.

SMR Basic

	Average Time (mm:s,mm)
ClientAdds	02:16.5
Client1	00:55.5
Client2	00:55.5
Client3	00:55.6

XL Basic

	Average Time (mm:s,mm)
ClientAdds	01:15.2
Client1	01:46.1
Client2	01:48.5
Client3	01:42.8

SMR Advanced

	Average Time (mm:s,mm)
ClientAdds	02:16.1
Client1	01:13.8
Client2	01:13.5
Client3	01:13.6

XL Advanced

	Average Time (mm:s,mm)
ClientAdds	01:17.8
Client1	01:16.4
Client2	01:59.2
Client3	02:43.0

6.1.4 SMR Advanced

This test is similar to the test described above, with the exception that we don't crash any server, meaning the view is more stable throughout the whole execution.

SMR Basic

	Average Time (mm:s,mm)
ClientAdds	02:19.5
Client1	00:55.0
Client2	00:54.2
Client3	00:54.1

XL Basic

	Average Time (mm:s,mm)
ClientAdds	01:10.8
Client1	00:55.0
Client2	00:54.2
Client3	00:54.1

SMR Advanced

	Average Time (mm:s,mm)
ClientAdds	02:21.7
Client1	00:50.3
Client2	00:50.8
Client3	00:50.2

XL Advanced

	Average Time (mm:s,mm)
ClientAdds	01:14.8
Client1	00:49.4
Client2	01:32.1
Client3	02:14.9

7. Conclusions

DIDA-TUPLE was an interesting and challenging project. We were able to implement all four requested versions. Fault tolerance was successfully implemented and tested. No matter which replica may crash, the system is able to recover and the clients can finish their execution. We found that the XL algorithm is a better implementation for a distributed tuple space, as was expected. However, for an application were there are many concurrent takes, the SMR will perform better. We also found that the basic versions, which assume perfect failure detection, perform better when the view is frequently changing. However, the penalty when not using a perfect failure detector is not very significant. Future improvements might including reducing the failure recovery time and code refactoring.

References

[1] Xu and B. Liskov *A* design for a fault-tolerant, distributed implementation of linda. In 1989 The Nineteenth International Symposium on Fault-Tolerant Computing. Digest of Papers(FTCS), volume 00, pages 199–206.

[2] Kenneth P. Birman and Robbert van Rennesse *R*eliable Distributed Computing Using the Isis Toolkit. IEEE Press, 1994. .