DIDA-TUPLE PROJECT REPORT
GROUP 7 - PADI - IST

Dorin Gujuman, Hugo Guerreiro, Mihail Brinza
{dorin.gujuman, hugo.guerreiro, mihail.brinza}@tecnico.ulisboa.pt
Instituto Superior Técnico
Avenida Rovisco Pais 1, 1049-001 Lisboa

Abstract

In this report we describe our implementation of two fault-tolerant tuple space systems. The first one implementing the Xu and B. Liskov [2] algorithm, and the second one implementing Raft. [1] We compare the relative advantages of both solutions and state the results and limitations of our system.

1. Introduction

Real world applications are often provided by several servers and these servers can go down at any moment, however the application itself should continue to be up and running. This is the reason why fault-tolerance is very important in highly available systems.

A tuple space is an example of the associative memory paradigm for distributed computing. It provides a repository of tuples that can be accessed concurrently. Our goal is to mix these two and make tuple spaces fault tolerant.

2. Developed solution

We developed two solutions based on two different fault tolerant tuple space systems. The first is based on the system developed by Xu and B. Liskov [2] and the second one based on a state machine replication.

At first, we describe the overall structure of our solution and then deep dive into the details of each implementation.

2.1. Solution structure

The various algorithms are divided in multiple projects in a solution:

- **DIDA-API** - Responsible for implementing the XL algorithm logic and providing and API for the client to make requests.
- **DIDA-API-SMR** - Responsible for implementing the SMR algorithm logic and providing an API for the client to make requests.
- **DIDA-FAIL-DETECT** - Responsible to detect failures on the nodes. Each node runs an instance of this project.
- **DIDA-CLIENT** - Implementation of a client that reads and executes a given script.
- **DIDA-SERVER** - A modular basic server that supports changing between different algorithms
- **DIDA-PCS** - Responsible for creating new processes in a computer.
- **DIDA-PUPPETMASTER** - Responsible for managing the existing servers and clients.
- **DIDA-LIB** - Shared library for communication between the servers and clients.
- **DIDA-RESOURCES** - Shared library for common functions, files and different system configuration variables.

2.2. DIDA-TUPLE-XL

The Xu and B. Liskov variant requires coordination between both the client and the replicas, called. This is achieved by slightly changing the way operations are executed (compared to the original Linda) to guarantee that the overall state of the underlying tuple space in each replica is the same.

2.3. Operations

In this version updates are carried out in the context of a view of servers, which is mutable and defines the agreed server replicas, and does the following operations:

Add The Client sends a multicast message to every replica where this new tuple is added and return an ack to the client. The Client repeats until acks from all replicas are received. The replicas detect duplicate requests and therefore acknowledge if the same Add operation is received but do not add the tuple.

Read Sends a multicast message to every replica where each replica seeks a match to the received schema and delivers a result to the client. The Client repeats until it receives one match from any replica.

Take The Take operation is divided into two parts and three operations: **TakeSelect**, **TakeRemove** and **Unlock**.

TakeSelect First, select the tuple to be removed. This operation is repeated until a client is able to obtain a non-null intersection of the tuple set responses of all the replicas. The servers seek all of the matching tuples and and try to acquire a lock on them, returning null otherwise. Locked tuples cannot be accessed until unlocked either by **TakeRemove** or **Unlock**.

TakeRemove Second, the client picks one tuple from the intersection of the sets and commands the replicas to remove that one and unlock the locked set they sent to the client.

Unlock In the case the intersection of sets is null the client has to unlock the tuples and try again.

2.4. Views

Servers using our failure detection module are aware of which other servers are up. In the case of XL, we need a way to define our view since some operations must receive acks from all of the servers on the view. We elect a leader and consider his view as the system's view.

Clients communicate with XL Servers by requesting a view from the leader. If a leader failed or it's the first time a client is doing an operation, it will then multicast this request. A Server who is not a leader will respond with null and the leader will return his view. After obtaining a new view the client can execute a command.

2.5. Joining the View

When a server wishes to join the View it sends a special request to one of the servers asking it to finish processing the current client requests, to halt incoming requests and share its state with the server attempting to join as a replica. When the copy is finished, the new server now has a valid state of the tuple space. It then notifies the copied Server to resume processing requests.

2.6. Failures

Failed nodes are detected by the failure detection module and removed from the view accordingly. The clients eventually will be notified of this failure whenever they execute or repeat a operation.

2.7. DIDA-TUPLE-SMR

In the state machine replication (SMR) version, we coordinate the client's interactions with the server by electing a leader that guarantees a total order of the operations of all clients and assures a coherent state at all times.

To achieve a total order of all operations, the leader is the one that is responsible to execute all operations and to replicate them to the other replicas in the cluster. When a majority of replication is achieved, the answer is then sent to the client.

Apart from guaranteeing the total order of operations, we also need to guarantee the following:

- Consistency between replicas.
- Tolerate 1 failure at each time before the system stabilized.
- Support only crash/timeout failures, not Byzantine.
- The number of replicas doesn't change dynamically when the system is running.

To achieve this requirements, we implemented the Raft algorithm [1].

2.7.1 Raft

" Raft is a consensus algorithm that is designed to be easy to understand. It's equivalent to Paxos in fault-tolerance and performance. The difference is that it's decomposed into relatively independent subproblems, and it cleanly addresses all major pieces needed for practical systems. We hope Raft will make consensus available to a wider audience, and that this wider audience will be able to develop a variety of higher quality consensus-based systems than are available today."

2.7.2 Leader election

We consider each tuple space server in our system to be a replica in a Raft cluster. Each node in our system start in a follower state and, after passing a certain time without having received an heartbeat from the leader, they become candidates with a new term.
As a candidate, he requests other members of the cluster for a vote. After acquiring a majority of votes, the candidate

becomes a leader and starts sending heart beats to the other members of the cluster.
Upon receiving an heart beat, the replica becomes a follower to the new leader.

Having a leader allows for clients to send requests to the same server which then coordinates the operations between the other members of the cluster.

2.7.3 Log replication

As mentioned above, the leader is responsible for executing the clients commands and giving them the responses. To guarantee consistency between each replica, while still being able to achieve fault tolerance, we maintain a log of executed operations in each server. The leader is the one that propagates the operations to the other nodes making use of the heart beats. Upon receiving the new operations, the node updates his log.

2.7.4 Failure handling

When a node fails or a new one joins the cluster, we guarantee that the system continues working as intended and that it continues consistent.

These are the possible failures:

Follower or Candidate When one of these failures happen, the failure detection module will perceive this change in the topology of the network and will update the view. As long as a majority can be achieved in the system, it will continue working as intended.

Leader If a leader dies, the failure detector will again perceive this change and act accordingly. The followers will have a timeout and will become candidates and a new election will happen. The overall state of the system will continue consistent because Raft guarantees it.

New node When a new node joins the server, the current (or future) leader will begin updating this new node and he will become a follower for that leader and also update his internal log.

This ”new node” can also be a node that was perceived as dead by the failure detector. Raft also assures that the system will continue consistent and working as intended due to the logging mechanism.

2.7.5 Client discovery

The client can send the messages to any server he wishes. If the receiving server is a follower it will respond with a failure and a direction to the location of the leader. If it is a candidate, the client will wait for some time and then retry the request. When he finally reaches the leader, the operation will be executed.

3. Failure Detection

Failure detection is handled by a separate module (DIDA-FAIL-DETECT).
We decoupled the detection of failure from the implementation of the actual algorithms to avoid repeated code, ease of the development and to separate responsibilities.

3.1. Failure detection Basic

The failure detection in the basic solution works by, first making the assumption that the system is asynchronous and then working with this assumption in our favor. The failure detection works by sending constant ping messages to the other (supposedly) alive nodes and, if a node doesn’t respond in a certain time window, the failure detector will assume that he is in fact dead. If, for any reason the failed node starts responding again, the failure detector will treat him as a new node and add him to the view (the node itself will also erase all of his previous state).

3.2. Failure detection Advanced

The advanced solution of the failure detection works in the same manner as the basic, however, when a timeout happens, the node is not treated as failed but instead as a suspect of having failed. If the node starts responding again, he will be added to the view with the same previous state. underlying algorithms are the ones that will guarantee that the detected failed node continues consistent with the rest of the system.

4. Results

In this section we compare the two developed solutions with some empirical results that we obtained. Although not extensive, we consider the gathered data sufficient to be able to draw some conclusions.
The major thing we observed was that our implementation of XL was not as worked as the SMR one, therefore the results might have suffered from it. fortunately these discrepancies (from what was actually expected) are not that significant.

Another observation we made was that .Net remoting technology acted as a bottleneck to some common operations like hearth beats. This bottleneck had some impact in the time that the system took to detect and recover from some failure

4.1. XL vs SMR- General performance

(**Note:**We performed these measurements with both systems stabilized and without any failure happening.)

When we evaluated the two systems in terms of a normal behaviour in production (i.e more reads than adds and takes) the SMR implementation thrived. As we can see in figures 1 and 2, XL took, on average, more time to execute this normal workload than SMR.

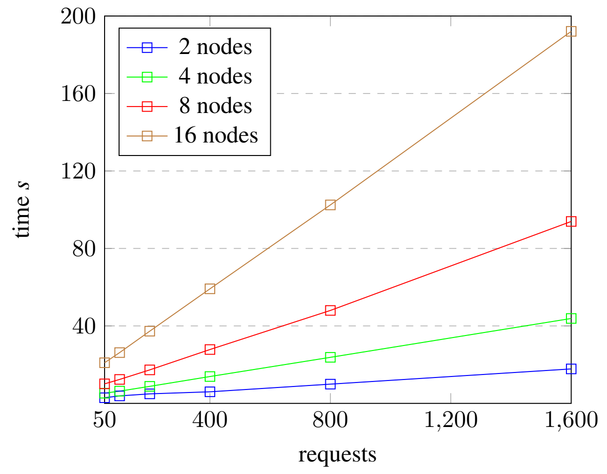

Figure 1: XL - Execution Time (80% read, 10% add and 10% take)

This happens mainly because it takes two remoting calls for each operation in the XL variant. Reads are optimized in both SMR and XL, the client waits for any response in the case of XL and SMR doesn't have to notify its replicas of the read operation. These optimizations allow faster reads but in the end on SMR has the upper hand on XL who needs to query every active node in the system and ask for the view.

As the nodes increase, the more time it takes for XL to query every node when compared to SMR that only needs to query one node.

Although this results might not make that much sense, we will see that with tests made purposely for stress testing, the overall picture changes.

4.2. XL vs SMR- Add and Take operations

After the weird results from the previous section, we decided to test specific operations of each algorithm.

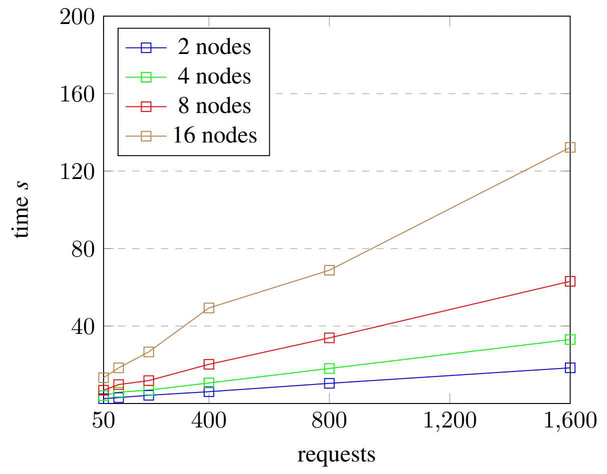

Figure 2: SMR - Execution Time (80% read, 10% add and 10% take)

4.2.1 Add

As we can see from figure 3, XL has a better performance than SMR when the requests are all Add operations. This happens because of the centralized nature of the SMR algorithm, he takes a lot of time replication all the operations made by the client to every replica in the node, while XL broadcasts the operations.

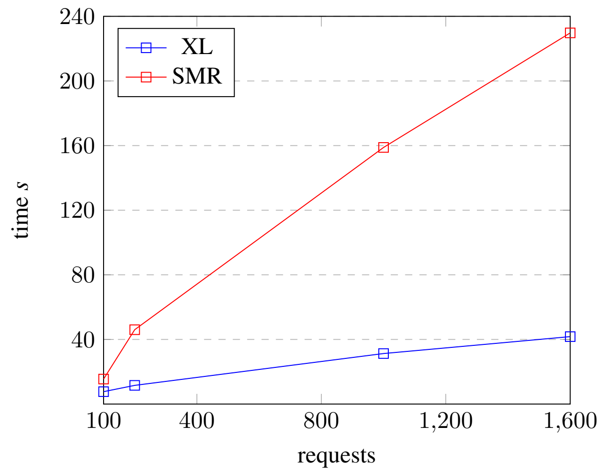

Figure 3: Execution Time (100% add requests)

4.2.2 Take

The stress testing dedicated to the take operation had two phases, an add phase and an take phase. We only measured the take phase for analysis purposes. What we observe is that the behaviour is almost identical in the two different implementations. XL however still has the tendency to take less time that SMR. One possible explanation to why XL still takes a lot of time is that every node sends all matches of the query to the client to then decide wich one he will remove. What can happen (in the worst case) is the client receiving the entire tuple space opposed to the SMR version that just sends the response back to the client.

Even though the SMR version doesn't have this particular issue from XL, it still takes a lot of time to execute the take instruction due to the overall synchronization of the operations throughout the entire node space.

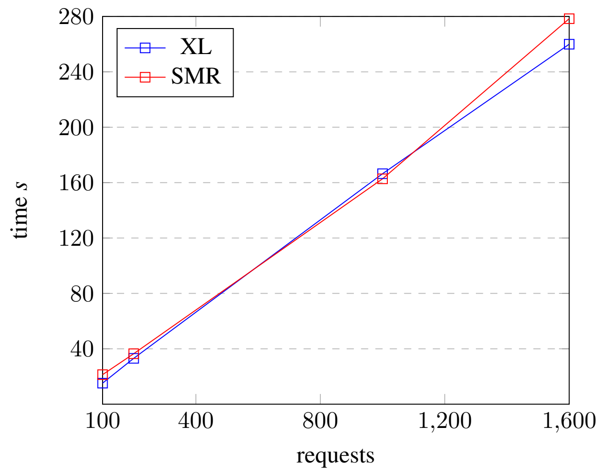

Figure 4: Execution Time (100% take requests)

4.3. XL vs SMR - Failure recovery

Failure detection in both algorithms is highly tied to the underlying Failure detection module, therefore the times that both implementations take to detect a failure are really similar. In average, the nodes take a predefined T amount of time (based on RTT) to detect a failure. For testing purposes, we set this T to 1 second.

The actual time difference in both implementations comes from the synchronization and the way that they uniquely handle it. Both tests were made on a relatively busy system and with a low density of tuples being added and taken from it.

4.3.1 XL

For the XL implementation we consider the system stable when a client can resume making requests. With the experimental parameters, the XL implementation takes, on average **3.8** seconds to recover from a failure.

This extra time is due to the fact that the client might be starting or still in the middle of a request and, when this failure happens he will also wait about N seconds/milliseconds (also based on RTT) before asking for a new view and then resume the requests.

4.3.2 SMR

On the other hand, a stable system on the SMR implementation is considered to be achieved when a new leader was found and this new leader has started accepting requests.

Apart from the parameter T, the SMR variant has another detail that we need to take into account. After detecting a failure, the Nodes currently in a Follower state don't take immediate action, they wait for their internal election timeout to trigger a new Election round in which they become candidates. Candidates themselves also have an internal timeout to know when to start a new election round as well. These internal timeouts are randomized between a predefined range.

For experimental purposes we set these two timeouts to two different ranges ([700ms-900ms] and [1.5s-3s]).

On average, the system took 4.34 seconds to recover from a failure with a random timeout range of 700ms-900ms and 5.47 seconds with the other range. A reason for this difference is due to the bottleneck imposed by .NET remoting that leads to election timeouts being triggered more often because of the latency in the communication. This would sometimes lead to a cascade of failed elections and re-elections.

4.3.3 XL vs SMR

Overall, the SMR implementation took more time to react and recover from failures due to the process of electing a new leader.

5. Conclusion

In this project we explored the challenges of developing services in an asynchronous distributed environment.

We described two techniques for constructing a highly-available tuple space system. One based on State Machine Replication implemented using the Raft Consensus Algorithm and another, described by Xu and Liskov, that leverages the semantics of the tuple space operations to try and optimize their delay while keeping the fault tolerant.

References

[1] Ongaro and Ousterhout. In search of an understandable consensus algorithm (extended version). 2013.

[2] Xu and B. Liskov. A design for a fault-tolerant, distributed implementation of linda. *The Nineteenth International Symposium on Fault-Tolerant Computing*, 00:199–206, 1989.