Two variants of a distributed fault-tolerant Tuple Space system

Mafalda Ferreira, Leonardo Epifânio and Pedro Lopes
Instituto Superior Técnico
Av. Rovisco Pais 1, 1049-001 Lisbon
DIDA 18/19 (MEIC-A / MEIC-T / METI)
Group 4

Abstract

We describe the developed solution for the implementation of two variants of a distributed fault-tolerant Tuple Space system: State Machine Replication [1] and Xu-Liskov [5]. This paper outlines the key design philosophies and explanation of the algorithms used in each variant. We discuss the relative advantages of each solution, and present experimental results.

1. Introduction

A Tuple Space (referred to as TS from this point on) is an approach to shared memory systems, where the processes involved communicate indirectly by placing tuples in a TS. A tuple can be accessed and removed by pattern matching. Most implementations use a centralized solution where the TS is managed by a single server. This is a simple solution but has several restrictions. It doesn't scale, being more prone to availability issues, and it is also not fault tolerant, since there's a single point of failure.

Distributed systems are interesting for four main reasons: scalability, high performance, availability and reliability. However, implementing a distributed system rises many issues. The computing nodes can crash, messages can be lost, duplicated or delivered out of order and the network can fail. If we don't take these issues into consideration, the system as a whole can fail.

This paper presents two design variants of a fault-tolerant, distributed system for a TS where high availability and fault tolerance are achieved with replication in several nodes. The main challenge with replication is maintaining data consistency among replicas. To surpass this issue, we consider the two following variants:

- The first variant is called State Machine Replication [1] (referred to as SMR from this point on) and it's based on the approach in which the TS behaves like a state machine, maintaining state and changing it deterministically, in response to events received. In order to ensure consistency between replicas, it is required that the replicas must start in the same state and must execute events in the same order. To satisfy the second requirement, we will adopt a total order algorithm.
- The second variant is based in the Xu-Liskov implementation [5] (referred to as XL from now on) which is designed to minimize delay of responses, given the semantics of the three TS operations: *add*, *read* and *take*. In this approach, the consistency in the *take* operation is assured by the agreement of the replicas on the tuple to be selected, and removing this tuple from all replicas.

We begin in section 2 by briefly describing the implementation tools and the main design and architecture of the solution. Section 3 gives an overview of the common parts in the two variants, including leader election, how a replica joins the view, and the perfect and imperfect failure detector. Both variants will be described in more detail in sections 4, 5 and 6. In Section 7 we'll discuss the relative advantages of each variant and present experimental results. The conclusion is then presented in section 8.

2. Design and Architecture

The solution was implemented in C# and .Net Remoting using Microsoft Visual Studio and the OOP paradigm.

2.1. Architecture

For the client and server we chose a layered architecture and a modular design for simplicity in development as we can see in the figure 1.

The module *Message Service Client* has two methods: *Request* and *Multicast*. Both call the remote method *Request* of the module *Message Service Server*. The Message

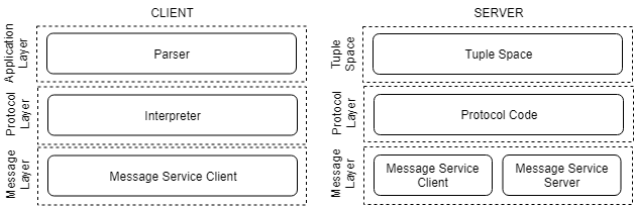

Figure 1: Layered architecture of the client and server.

Layer receives the messages and passes them to the Protocol Layer where the system protocol runs. In the case of the server, the Protocol Layer calls the Tuple Space module to conclude the operations over the TS in the replica.

Each layer has a clean API that can be called by the under layer. The requests and responses correspond to serialized objects that are processed in the Protocol Layer using the Visitor design pattern.

3. Overview

Both variants have the following different status: INITIALIZATION, VIEWCHANGE and NORMAL. The SMR has one extra status, which is RECOVERY. The server only processes requests when in the NORMAL status. The behaviours in each state are detailed in the succeeding sections.

Despite the differences between the two, they share some of the problems that are inherited from the development of distributed systems. All replicas must agree on the current view, they must detect failures and, consequentially, must all act the same way. Messages can be duplicated, therefore, they must be filtered. Delays can also happen and messages can arrive out of order.

3.1. Filter Duplicate Messages

Each replica has a Client Table which saves, for each client, the client-id of the last request, the request number and the respective response. The algorithm to filter duplicates works as follows:

1. The client sends the message $\langle$REQUEST *op*, *c*, *s*$\rangle$ where *op* is the *operation* with its arguments, *c* is the *client-id* and *s* is the *request-number* assigned to the request (which is an incremented counter starting at 0).

2. When the server receives the message, it compares the *request-number* with the information in the Client Table (on the row that corresponds to the client that sent it). If the *request-number* is smaller than the *request-number* in the Client Table, it drops it. If it's equal, it re-sends the response that is stored in the Client Table, otherwise it accepts and processes the request.

3.2. View Changes

Although the information passed in the messages about the replica's state is different for each variant, the semantic of the messages remains. A configuration can be seen as group of $2f + 1$ replicas. The algorithm is as follows:

1. When a replica i notices the need for a view change, it sets its status to *ViewChange* and sends the message $\langle$STARTVIEWCHANGE i, v, $c\rangle$ to all the other replicas, where v is the new *view-number* and c is the new *configuration*.

2. When a replica i receives f STARTVIEWCHANGE messages for its *view-number* and *configuration*, it sends $\langle$DOVIEWCHANGE i, v, v', c, s, $k\rangle$ to the node that will be the leader in the new view, where v' is the previous view-number, s is the state of the replica and k is the commit number.

   In SMR: $s = (l,\ n)$, where l is the logger and n is the op-number.

   In XL: $s = (t,\ d)$, where t is the TS and d is the Client Table.

3. When the new leader receives $f+1$ DOVIEWCHANGE messages, it selects the best state and updates its state, then informs all the other replicas by sending the message $\langle$STARTCHANGE i, v, c, s, $k\rangle$ and sets its status back to *Normal*. The best state is the one with the biggest view-number. In case of a draw, it selects the biggest op-number and the biggest commit-number in SMR and XL respectively. The replicas that receive the STARTCHANGE message, update their state.

In the case of SMR, there are two extra steps:

4. The new leader starts accepting client requests. It also executes (in order) any committed operations that it hadnt executed previously and updates the Client Table.

5. When the other replicas receive STARTCHANGE, they update their state and set the status back to *Normal*. If there are non-committed operations in the log, they send a PREPAREOK to the leader. Then execute, in order, all operations known to be commited.

3.3. Leader Election

The leader of a configuration is the first server-id in lexicographical order. Thus, when a configuration is agreed upon in the view-change, the leader is implicitly elected. The leader in SMR also becomes the primary.

3.4. Initialization Operation

When a replica starts up or thinks that it's out of the current view, the status is changed to *Initialization*. The protocol runs as following:

1. A replica i sends, to a all possible servers, the message $\langle$SERVERHANDSHAKE $i\rangle$. If it doesn't receive a response in a determinate timeout, it thinks that it's the first server and sets its status to *Normal*.

2. If one or more replicas are alive, they will respond with the current view configuration, and replica i sends, to all servers in the configuration, the message $\langle$JOINVIEW i, $u\rangle$, where u represents the URL of the replica i. Then it waits for a STARTCHANGE message and updates its status as explained in section 3.2. If the message doesn't arrive in a determinate time span, it repeats step 1.

3.5. Failure Detector

Each replica i is monitored by all the other replicas in the view. Replica i periodically sends a lease renewal request $\langle$HEARTBEAT $i\rangle$. When a monitor acknowledges with $\langle$HEARTBEATRESPONSE $v\rangle$, where v is the view number[1], replica i is said to obtain a lease.

3.5.1 Perfect Failure Detector

When a replica i fails to renew a lease in a replica j, the perfect failure detector says promptly that i is faulty and j sets its status to *ViewChange* and sends the message STARTVIEWCHANGE to everyone. The other replicas, as they receive the STARTVIEWCHANGE message, change their status to *ViewChange* and also send the message STARTVIEWCHANGE to everyone. From here on, the protocol runs as explained in section 3.2. Also, the next leader may not be aware of the detection of STARTVIEWCHANGE messages, but when it receives DOVIEWCHANGE, it changes the status to *ViewChange* and the protocol proceeds.

3.5.2 Imperfect Failure Detector

When a replica i fails to renew a lease in a replica j, the imperfect failure detector is conservative. It changes its status to *ViewChange* and proposes a view change by sending STARTVIEWCHANGE messages to the other replicas. However, it waits a determinate timeout for f STARTVIEWCHANGE messages, confirming its view change. If it doesn't receive f STARTVIEWCHANGE messages, the change is cancelled and the replica changes the status back to *Normal*. There needs to be quorum, to decide that i is faulty.

[1]The view-number is a health check to see if replica i is in the current view.

4. State Machine Replication

The core of SMR is the implementation of a total order protocol. For this, we decided to implement Viewstamped Replication [3] since it is simple, correct, easy to understand and easy to implement in comparison with Paxos [2] or Raft [4].

4.1. Normal Operation

- ***op-number*:** assigned to the most recently received request, initially 0.
- ***log*:** This is an array containing *op-number* entries. The entries contain the requests that have been received so far in their assigned order.
- ***commit-number*:** is the op-number of the most recently committed operation.

Figure 2: State of the SMR replica.

This section describes the process of requests that are accepted when the primary isn't faulty. The client only communicates with the primary when performing a request. It runs as follows:

1. When the primary accepts a request, it increments its *op-number*, adds the request to the end of the log and updates the information in the client table. Then it sends a $\langle$PREPARE v, m, n, $k\rangle$ to the other replicas, where m is the message received from the client.

2. The backup replicas process the PREPARE messages in order. They wont accept a prepare message with *op-number* n until they have entries for all earlier requests in their log (it may require *Recovery*), putting the message on hold until the condition is met. Then they increment their *op-number*, add the request to the end of the log, update the client's information in the client table, and send a $\langle$PREPAREOK v, $n\,i\rangle$ message to the primary, to indicate that this operation and all earlier ones have been prepared locally.

3. The primary waits for f PREPAREOK messages from different backups. After it receives these messages, it considers the operation (and all earlier ones) to be *committed*. Then, after it has executed all earlier operations

(those assigned smaller *op-numbers*), the primary executes the operation by making an upper call to the TS, and increments its *commit-number*. Then, it sends a $\langle$REPLY v, s, $x\rangle$ message to the client, where v is the *view number*, s the *request-number* and x the result of the the call to the TS. The primary saves this reply in the client table.

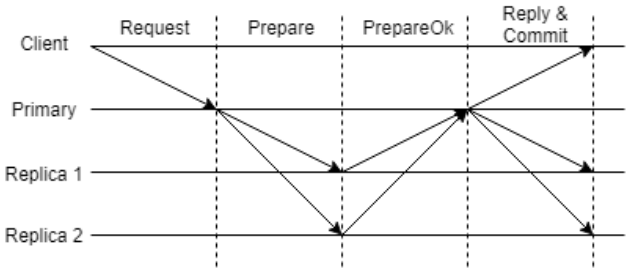

Figure 3: Normal case processing for a configuration f=1.

4. The primary informs the other replicas about the commit, by sending a $\langle$COMMIT i, v, $k\rangle$.

5. When a backup learns about the commit it waits until it has the request in its log (which may require Recovery) and until it has executed all earlier operations. Then it executes the operation by doing an upper call to the TS, increments its *commit-number* and updates the client table.

Figure 3 shows the phases of the normal processing protocol.

4.2. Recovery Operation

Recovery is used when a replica lags behind (but hasn't crashed), and needs to be brought up-to-date. After a certain time without executing anything, the replica changes its status to *Recovery* and starts the following protocol:

1. The replica i sends a $\langle$RECOVERY i, v, n', $k\rangle$ to all replicas in its configuration, where n' is its *op-number*.

2. A replica only responds to a RECOVERY message if, and only if, its status is *Normal* and if it's in the same view. In this case it sends a $\langle$RECOVERYRESPONSE v, n, k, $l'\rangle$, where l' corresponds to its log after n'.

3. When replica i receives the RECOVERYRESPONSE message, it appends it to its log and updates its state using the other information in the message.

5. Xu-Liskov

The XL variant is simpler than SMR as it doesn't require total order. However, to maintain consistency and simplify the implementation, we decided that the requests of the client should be executed in order and, thus, it respects casual order. This means that the add operation also blocks, and that it will not return until it gets all the acknowledge messages from the servers in the current configuration.

5.1. Normal Operation

- ***commit-number*:** The number of executed operations.

Figure 4: State of the XL replica.

Besides the TS and the client table, the XL also has a *commit-number* attribute as shown in figure 4.

The XL variant provides 2 more requests in addition to the three (*add*, *read* and *take*):

- ***getAndLock***: This request has an argument, which is the tuple pattern that the client wants to take. The response will be the set of tuples that match that pattern. The server locks those tuples in the process, atomically. If the lock process fails, it returns a refusal response.

- ***unlock***: The request *unlock* unlocks any tuple set that the client has locked in the moment. Also atomically.

In the *Normal* status, when a replica accepts a request from the client (if the client sends a request with a smaller view number, it is dropped), it executes the operation by doing an upper call to the TS, advances its commit-number, updates its client table, and sends to the client the message $\langle$REPLY v, s, $x\rangle$.

6. Hiding failure from the client

The biggest advantage in a fault-tolerant system, is masking the failure to the client. Here, it is accomplished by also saving the current view status in the client. When the client doesn't receive a timely response to a request, it performs a CLIENTHANDSHAKE to retrieve the new view and hence, in the case of SMR, retrieving the new leader. The CLIENTHANDSHAKE works as follows:

1. When the client initializes or doesn't receive a timely response to a request, it sends $\langle$CLIENTHANDSHAKE $c\rangle$ to a set of possible servers, where c is the client-id.

2. The client waits (with a timeout) until receives a response ⟨CLIENTHANDSHAKERESPONSE p, v, r, l⟩, where p is the protocol used by the server, v is the view-number, r is the configuration and l is leader (in case of SMR). If it doesn't receive the response within the timeout period, step 1 is repeated.

7. Evaluation

We evaluate the performance for the different variants of the TS. We identify the relative advantages of each implementation and show the practical results of the evaluation.

7.1. Detection: Perfect vs Imperfect

In Perfect Failure Detector (PFD), when a replica suspects that a node is down, the other replicas trust this judgement and act accordingly. In Imperfect Failure Detector (IFD), there needs to be a quorum of replicas suspecting that a node is faulty, in order to agree with a view change.

Although IFD is better at not excluding replicas that are not faulty (but that some of the other replicas think they are), it brings a lot more overhead into the system and the network. In our implementation of IFD, explained in section 3.5.2, a replica can switch to status *ViewChange* several times, halting the process of requests each time, before a view-change is accepted by a quorum, leading to a decrease in performance in comparison with PFD. And as more replicas join the view, there's a higher probability that this number of view-change requests by a replica increases until it is accepted.

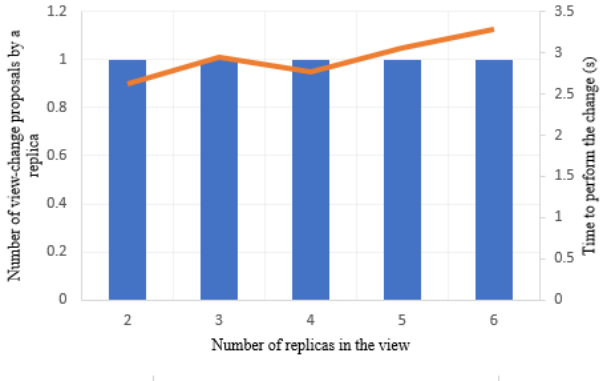

Figure 5: Number of view-change proposals and time until the view change is done in PFD.

Figures 5 and 6 show the number of view-change proposals made by a specific replica, when it starts to detect a faulty replica, until the view-change is completed successfully. As we can see, the PFD is pretty much constant, and it scales well with the number of replicas. In the IFD, many

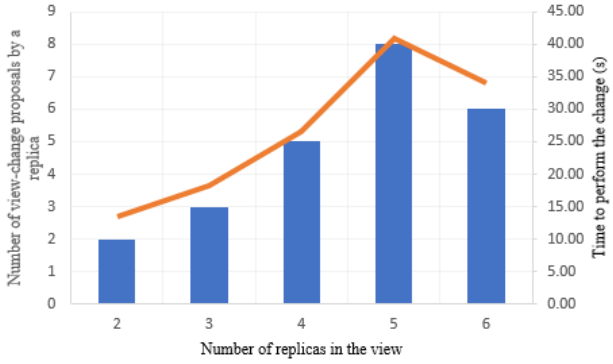

Figure 6: Number of view-change proposals and time until the view change is done in IFD.

proposals are made until all the replicas agree at relatively same time. This, however, can take a long time or, on the contrary, be very quick. It depends on the circumstances of the asynchronous system. There is a tight correlation between the number of view-change proposals and the time it takes for a view change to be processed successfully: the more proposals there are, the more time will pass until the view change is accepted by the quorum.

7.2. SMR vs XL

In this section we'll discuss the main advantages of a variant in comparison with the other, having in account the idiosyncrasies of each one.

The XL variant takes advantage of the semantics of the operations and improves the performance as the *add* and *read* operations can be executed simultaneously. This doesn't happen in SMR, where each request is executed at its time in order to guarantee that all replicas execute them in the same order. This sometimes results in the halt of the request execution because it didn't execute the previous one, which is not necessary in some cases (e.g. when there are two read requests). To verify the validity of this premise, we did a performance test with several clients performing read requests at the same time. We added a random delay between 10ms and 300ms.

```
begin-repeat 200
add <"a",DADTestA(1,"b")>
read <"*",null>
end-repeat
```

Figure 7: Client script for multiple reads and adds.

Each client performs 400 requests (add and read alternated), and the number of clients are incremented on each

iteration (and so are the total of requests). We used $f = 1$.

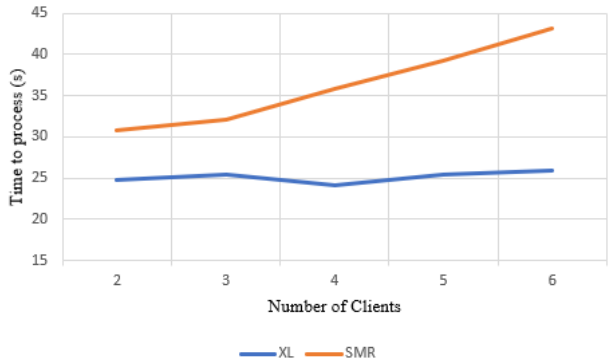

Figure 8: Multiple reads in parallel.

In figure 8, although the number of requests is increasing, the XL variant keeps processing the read requests almost at the same time. In contrast, in SMR, it takes longer to execute as the number of requests increases.

SMR has an advantage in comparison with XL in the take operation. As explained in Xu-Liskov's paper [5], the take operation is like a two-phase commit. A client asks the replicas to lock all the tuples that match the given pattern and the replicas return them. If successful, the client does the intersection of all replicas' responses and, if not empty, selects the tuple to take and broadcasts to every other replica. The bottleneck in this approach is the lock operation. The lock fails when it tries to lock an already locked tuple. So, if many clients attempt to take the same tuple pattern, the system will struggle so hard, that it would be better to have total order.

```
begin-repeat 50
add <"a",DADTestA(1,"b")>
end-repeat
begin-repeat 50
take<"*",null>
end-repeat
```

Figure 9: Client script for multiple take operation.

In figure 10 we can see how poorly XL performs when there are multiple take requests working with the same tuple set. In this case we used the same tuple, however that was just a simple example of the problem. A tuple pattern can generate a huge set of tuples that match. So, even if two clients are working with different tuple patterns, there might be a conflict if they sets intersect. In this case, both can fail to acquire a lock to their set.

In SMR, on the other hand, this isn't a problem. The requests are ordered and executed strictly in that agreed sequence, so, there aren't these type of conflicts.

Figure 10: Multiple takes in parallel.

8. Conclusion

This paper presented two approaches for the implementation of a distributed fault-tolerant Tuple Space system. We described the used algorithms and their implementation meticulously. In section 7 we verified that each variant's performance depends on the provided workload. For systems that require a higher availability, XL might, in general, be a better option, since it doesn't require total order, unlike SMR. However, if there is a high amount of removals (*take* requests), SMR may outperform XL, especially if this type of operations are performed on tuple sets that intersect with each other.

Regarding failure detection, PFD will perform better if the network is stable. In the opposite case, where the network's health is impaired by the existence of partitions, big delays, or partial shutdowns, PFD might generate a lot of false positives, and non-faulty servers may wrongly leave the view, compromising the availability. In this case, IFD would be a safer choice.

References

[1] D. E. Bakken and R. D. Schlichting. Supporting fault-tolerant parallel programming in linda. *IEEE Trans. Parallel Distrib. Syst.*, 6(3):287–302, Mar. 1995.

[2] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[3] B. M. Oki and B. H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. PODC '88, pages 8–17, New York, NY, USA, 1988. ACM.

[4] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. USENIX ATC'14, pages 305–320, Berkeley, CA, USA, 2014. USENIX Association.

[5] A. Xu and B. Liskov. A design for a fault-tolerant, distributed implementation of linda. In *FTCS*, 1989.