ABSTRACT

NEMLEKAR, MILIND NILKANTH. Scalable Distributed Tuplespaces. (Under the direction of Dr. Gregory T. Byrd.)

The purpose of the research has been to develop a multiple tuplespace model that would scale as much as the Internet. A tuplespace is like a shared cache, in which tuples are accessed associatively. One issue in designing a multiple tuplespaces model is keeping track of tuples over multiple space servers. Since, replication is used to reduce access latencies to tuples, another issue is of establishing coherency of replicas and consistency of tuplespace operations over multiple replicas.

The thesis looks at design of a hierarchical directory structure over a flat organization of tuplespaces, which addresses the above issues. With this model scalable protocols are proposed that keep track of tuples/templates among multiple nodes, and establish coherency of tuple replicas.

A prototype of this model has been implemented within the Jini™/JavaSpaces™ framework. The operation semantics of the directory-based multiple tuplespaces model are verified for their correctness by running representative tuplespace applications.

SCALABLE DISTRIBUTED TUPLESPACES

By

Milind Nilkanth Nemlekar

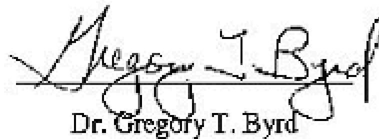
A thesis submitted to the Graduate Faculty of
North Carolina State University in partial fulfillment of the
Requirements for the Degree of
Master of Science

Computer Networking

Raleigh

2001

APPROVED BY:



(Chair of Advisory Committee)

Dr. Edward Gehringer

Dr. Mark White

To

Aai and Baba

BIOGRAPHY

Milind Nemlekar was born in Mumbai (Bombay), India on January 22, 1977. He received his Bachelors degree in Electronics Engineering from University of Mumbai (Thadomal Shahani Engineering College), Mumbai, India in 1999. Milind has been a graduate student with the Masters program in Computer Networking at North Carolina State University, Raleigh, NC, since August 1999. During this time he worked as a Research Assistant for Dr. Gregory Byrd in the field of tuplespace systems.

ACKNOWLEDGEMENTS

There are many people who have made the pursuit of my Masters degree possible. I would first like to thank my parents and my sister for giving me the opportunity to achieve my goals and having stood by me in every decision I have made.

I would like to thank and express my deepest appreciation to my advisor Dr. Gregory T. Byrd for his invaluable guidance, and sincere help throughout my graduate study at North Carolina State University. I would also like to thank him for his trust, continuous support and for the time and effort he invested on me. Working with him has been a highly rewarding experience.

I would like to thank Dr. Edward Gehringer and Dr. Mark White for their constructive comments and for serving on my thesis committee.

Special thanks to all my roommates who have encouraged me and helped me sort things out.

Table of Contents

List of Figures viii

List of Tables x

List of Abbreviations xi

1 Introduction **1**

1.1 Study 2

1.2 Implementation 3

1.3 Outline 4

2 Tuplespace Paradigm **5**

2.1 Generative Communication 5

2.2 Tuplespace Primitives 5

2.2.1 Predicate primitives and EVAL primitive 7

2.2.2 Properties 8

2.3 Consistency Models 8

2.4 Semantics for Centralized Tuplespace primitives 9

2.4.1 Synchronous Operations 10

2.4.2 Asynchronous Operations 11

2.5 Consistency model for a Distributed Tuplespace 11

2.5.1 Synchronous Operations 12

2.5.2 Asynchronous Operation 13

2.6 Replica update protocol 13

2.6.1 Protocols for a partition of Spaces 13

2.6.2 Protocols for spaces arranged in a network topology 16

2.6.3 Discussion 17

3 Tuple Distribution/Replication Issues **18**

3.1 Replication policies over multiple spaces 18

3.2 Full Replication 19

3.3 Partial Replication 20

3.3.1 Deterministic Replication 20

3.3.1.1 Optimization of Distributed Tuplespace messages 21

3.3.1.2 Compile time analysis for tuple distribution 21

3.3.1.3 Runtime Optimizations 22

3.3.2 Non-Deterministic Replication 23

3.3.2.1 Grid based replication 23

3.3.2.2 Tree based replication 24

3.4 Cache only Memory Architectures 25

3.4.1 Localization and Replacement 25

3.4.2 DDM Architecture 26

3.5 Summary 26

4 Directory based structure 28

4.1 Directory based model 28

4.2 Hierarchical Directory 30

4.2.1 Directory Tags 31

4.3 Primitive Distribution 31

4.3.1 OUT 32

4.3.2 IN 32

4.3.3 INP 34

4.3.4 READ 34

4.3.5 READP 35

4.4 Coherence Protocols 36

4.5 Summary 36

5 Implementation 38

5.1 Implementation Infrastructure 38

5.1.1 RMI 38

5.1.2 Jini 39

5.1.2.1 Jini Lookup Service 39
5.1.2.2 Discovery and Join Protocols 41
5.1.2.3 Reggie and RMID 41
5.1.3 JavaSpaces 42
5.1.3.1 Overview 43
5.1.3.2 Other features 43
5.2 The JavaSpaces Implementation 44
5.2.1 UML Class Diagram 44
5.2.2 SpaceProxy: Client proxy for Outrigger 44
5.2.2.1 Consistency 46
5.2.3 Entries, Directory and Directory Tags 47
5.2.4 BasicSpace: Space server for Outrigger 48
5.3 UML Flow Diagrams 49
5.3.1 OUT 49
5.3.2 READ 49
5.3.3 IN 50
5.4 Summary 51

6 Performance Evaluation 52
6.1 Experimental Setup 52
6.2 Analysis of operation propagation time 52
6.3 Analysis based on application performance 54
6.3.1 Matrix multiplication 55
6.3.2 Mandelbrot fractal 58
6.4 Summary 59

7 Conclusion and Future work 61
7.1 Conclusion 61
7.2 Future Work 62

List of References 63

List of Figures

1.1	Symmetric Multi Processors	2
1.2	Distributed Shared Memory	3
2.1	A tuple space representation	5
2.2	Two processes concurrently performing IN on multiple tuplespaces	12
2.3	Protocols for maintaining global atomicity of IN operation	14
2.4	Active replication	14
2.5	Primary copy replication	15
2.6	Voting protocol	16
3.1(a)	State replication	19
3.1(b)	Operation replication	19
3.2(a)	Positive broadcast	20
3.2(b)	Negative broadcast	20
3.3	Hash based partial replication	21
3.4	Rendezvous assignment	22
3.5	Grid based replication	23
3.6	Tree based replication	24
3.7	Hierarchical DDM	26
4.1	Clustering of space servers	29
4.2	The cluster tree, connected by a hierarchical directory structure	29
4.3	Tuple tags	31
4.4	Tuple propagation due to OUT primitive	32
4.5	Template propagation due to IN/READ	33
4.6(a)	Tuple-template match with the directory	34
4.6(b)	Remote IN operation	34
4.7(a)	Tuple-template match with the directory	35
4.7(b)	Remote READ operation	35
5.1	Remote method invocation	39

5.2	Locating services through the look-up server	40
5.3	Discover and join protocol	41
5.4	JavaSpaces	42
5.5	Outrigger UML diagram	44
5.6	Modified Outrigger UML diagram	46
5.7	BasicSpace data structure	48
5.8	UML diagram for OUT	49
5.9	UML diagram for IN	50
5.10	UML diagram for READ	51
6.1	write operation latency	53
6.2	read operation latency	53
6.3	take operation latency	54
6.4	Configuration for Matrix multiplication	56
6.5	Matrix multiplication results	57
6.6	Screenshot of the mandelbrot fractal	58
6.7	Mandelbrot fractal results	59

List of Tables

2.1 Dependency between semantic results of tuplespace primitives due to internal ordering of concurrent operations 10

List of Abbreviations

COMA	Cache Only Memory Architecture
DDM	Data Diffusion Machine
DSM	Distributed Shared Memory
HTTP	Hyper Text Transfer Protocol
JAR	Java Archive
JVM	Java Virtual Machine
NUMA	Non Uniform Memory Architecture
RMI	Remote Method Invocation
rmid	RMI daemon
SMP	Symmetric Multi Processors
TCP	Transmission Control Protocol
TS	Tuple Space
UML	Unified Modeling Language

Chapter 1

Introduction

The Internet-computer vision, broadly, is to transform the Web from a medium mainly for viewing and downloading into a genuine computing platform. That is, a "programmable" medium, in which more data that can be manipulated and programmed is transported across the Internet to do all kinds of useful things.

The Internet-computer is in a very primitive stage. Most of the traffic on the network connections is HTTP traffic that allows data transfer and one-to-one interaction. Distributed programming systems that are in wide spread use, like CORBA (Common Object Request Broker Architecture) and RMI (Remote Method Invocation) [Sun99a], are based on the client-server protocol. These systems do allow distributed computation, but they don't have the necessary elegance for dynamic scheduling or allocation of resources, which are essential in building an Internet-computer. In addition, the Internet-computer is required to provide a platform for parallel program execution. Parallelism involves dividing a problem into pieces, and arranging for all the pieces to be solved simultaneously.

The tuple space paradigm provides such a platform for data coordination among different parallel entities. In this model, different processes communicate though the use of elementary memory units – called tuples, which are records of typed fields. These tuples are associatively accessed via a few primitives based on a pattern-matching mechanism. Tuple space is a global memory that allows storage of tuples and provides for primitives to operate on the stored tuples. The simplicity and the generality of this model have motivated its adoption at a higher level of abstraction, which is evident from the implementation of systems like JavaSpaces [Sun99b] and T-spaces [WL98]. These implementations provide a general coordination model for parallel programming.

1.1Study

A centralized tuplespace can be compared to a shared memory multiprocessor system. These systems, referred to as Symmetric Multi Processors (SMP), have a global memory space that is equally accessible to all the processors that use its service. However it typically suffers from increased contention in accessing the centralized space server, which limits the scalability. In addition to the scalability issue, there is also a concern about reliability of such centralized system.

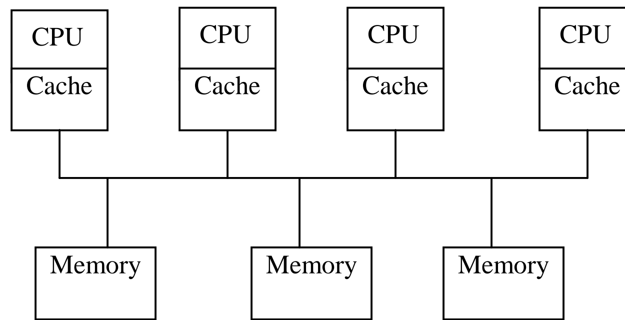


Figure 1.1: Symmetric Multi Processors

For a scalable solution, distributed local memories of multiple computers are mapped by using memory-management units or hardware directories, to provide a global virtual memory abstraction. These distributed shared memory (DSM) [NL91] systems consist of autonomous processing nodes, having independent flow of control and local memory modules. These systems are referred to as Non Uniform Memory Architecture (NUMA) systems. This is because each node has a very low latency access for local data and a high latency for accessing remote data from other nodes. To reduce access latencies, NUMA systems use replication and migration protocols to make data available closer to the nodes that access them. This brings in additional requirements of maintaining data consistency and coherency.

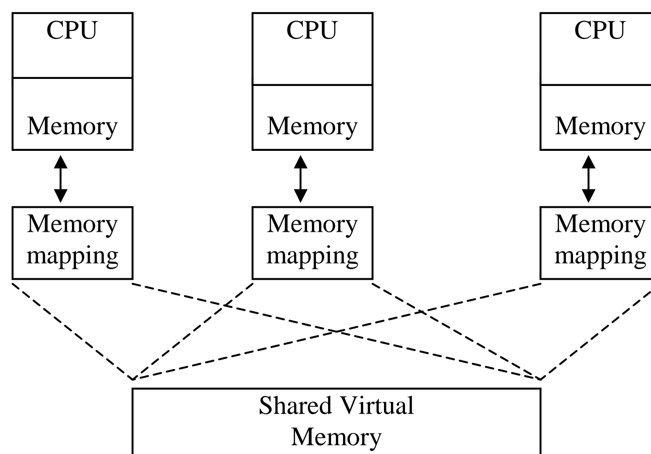


Figure1.2: Distributed Shared Memory

Thus, in view of their logical similarity we study different DSM algorithms and their relevance in implementing a scalable multiple tuplespace solution. We seek a solution where multiple tuplespaces coordinate to provide an illusion of a centralized space server, and thus provide scalability by taking advantage of locality of tuple accesses and replication of tuples. This approach hides the mechanism of communication between multiple tuple servers from the application, so the ease of programming and portability of a centralized tuplespace coordination server, and cost-effectiveness of distributed tuplespace implementations can be achieved.

This thesis studies different tuple distribution and replication strategies toward an efficient and scalable implementation. The replication strategy exploits locality of tuples and follows the access patterns of the application. The purpose of tuple replication is to achieve low latency accesses for tuples distributed over multiple tuple spaces. A major challenge imposed by replication is that of maintaining replica consistency. The challenge here relates to the trade-off between scalability and performance, as improving one property affects the other negatively.

1.2 Implementation

The implementation model proposed in this thesis defines a flat organization of tuple spaces, connected by a hierarchical structure of directories. Tuple signatures are

propagated as tags with a depth that depends on the scope of the tuplespace. This solution avoids bottlenecks in the hierarchy and achieves tuple access times proportional to degree of locality of access and logarithmically bounded with respect to the system size. There is no deterministic assignment of tuple replicas and the degree of replication is independent of system size and dictated mainly by application patterns. We study this implementation within the JavaSpaces framework, which is a space server that runs on Sun's Jini infrastructure [Sun99c].

The prototype implementation involves construction of a single level hierarchy, i.e. a directory interfacing with multiple spaces. We have modified *Outrigger*, which is Sun's contributed implementation of JavaSpaces, in building this prototype.

1.3 Outline

The outline of the thesis is as follows.

Chapter 2 introduces the tuplespace paradigm. This introduction includes discussion of space-based primitives and their properties based on the Linda coordination language. We provide an overview of the consistency semantics for executing tuplespace primitives over a central space server as well as over multiple space servers.

Chapter 3 discusses tuple replication protocols. Through replication, a tuple/template match is assured over multiple spaces. We explore full replication as well as partial replication protocols. We also talk about other interesting protocols that are used to locate data in DSM systems, and can be put to use to keep track of tuples among multiple spaces.

Chapter 4 explains the design of the Directory based model. We explain the reason for our choice of a hierarchical directory based structure. Further we describe the protocols for executing different tuple space primitives.

Chapter 5 looks at the implementation of the model within the Jini/JavaSpaces Framework. We begin with a brief overview of RMI, Jini and JavaSpaces and conclude with a description of the Outrigger implementation and its modifications.

Chapter 6 presents some performance results on a cluster of computers for matrix multiplication and mandelbrot applications.

Chapter 7 concludes the thesis and suggests directions for future work.

Chapter 2

The Tuplespace Paradigm

This chapter explains the tuplespace paradigm and forms a background for the terms frequently used in further chapters. We also discuss the synchronous nature of the IN primitive and the protocols required for maintaining global atomicity of such an operation.

2.1 Generative communication

The *generative communication* model proposed by Gelernter [Gel85] combines the advantages of the shared memory model as well as the message passing. Here we have a shared data space as shown in figure 2.1, referred to as tuplespace, or TS, which acts like a mailbox for several processes. Different processes drop messages in the mailbox and other processes pick them up (or copy them instead of picking). The messages exist as ordered set of values called *tuples*. They are accessed associatively by their logical name, where a tuple's name is any selection of its values.

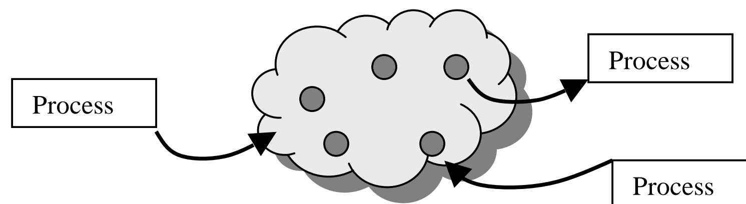


Figure 2.1: A tuple space representation

2.2 Tuplespace primitives

There are three main primitives defined over TS: OUT(), IN(), and READ(), all with a tuple as a parameter.

OUT("A", int i, int 5) will cause the tuple ("A", int i, int 5) to be inserted into the TS. For this tuple *'int i'* is a *formal* field while *'int 5'* is an *actual* field. *Formal* fields are placeholders and receive the values of the corresponding field of the matching tuple. A field that is not formal is termed as *actual* field. Tuples match when their corresponding actual fields are identical and the data-type of the formal field matches with the data-type of the actual field.

IN("B", int i, float x, int 3) will search the TS for the matching tuple in the TS. The matching tuples are merged with the formal fields replaced by actual fields and the resulting tuple is extracted from the TS.

READ("A", int 6, int j) operation is similar to in except that the value of the merged tuple is copied and not extracted from TS. IN and READ operations are termed as coordinating operations as the process blocks till the match is found.

It is a consequence of the IN operation that the tuples in a space cannot be modified just as a write operation. To be changed, they must be physically removed, updated, and then reinserted. This makes it possible for many processes to share access to TS simultaneously. Thus, we can build distributed data structure over TS [ACG86] that, unlike conventional ones, may be manipulated by many processes in parallel. In addition, the processes operating on TS are independent and do not coordinate the ordering of their primitives. Hence, TS primitives remain partially ordered, that is, they are performed sequentially with respect to individual processes, but when multiple processes are taken into consideration their order of access to the space is not in sequence.

Both the IN and READ operations can block the calling process: the execution does not continue until a match occurs. When operations produce multiple matches, i.e. when one template matches multiple tuples, only one of the tuples is committed in a non-deterministic way, so as to grant fairness.

There is a strict duality of the IN (and READ) operations, whose parameter is called the template, and the OUT, whose parameter is referred to as a tuple. When an IN or READ template does not result in a match, it is stored into the space, so as to test for future match possibilities. Conversely, for an OUT operation, it is necessary to test the presence of "request" templates to match with it. For a template to match a tuple in

tuplespace, the template and tuple must agree on the number, types and values of corresponding actual fields, and should have no corresponding formal fields.

The tuple exists as a serialized stream within the space server and it cannot undergo changes while it is stored with the space.

2.2.1 Predicate primitives and EVAL primitive

Linda includes predicate variations of the IN and the READ primitives, named INP and READP. These are non-blocking versions of the primitives, which inspect the present state of tuplespace. In other words, if no tuple matches these operations (a matching tuple does not exist a priori), these primitives return *null*. The semantics of these primitives are inappropriate for a multiple tuplespace implementation, where the most recent operation is not defined. If the tuples matching the templates for these primitives are not present in a local space that does not rule out their existence in other spaces. Some TS implementations like the FT-Linda [BS95] provide absolute guarantee as to whether there is a matching tuple or not. They refer to this property as *strong INP/READP semantics*.

In our implementation we do not guarantee that there are no matching tuples when these predicate operations are invoked, and return a *null* if a tuple is not found in the local space.

The final TS primitive is the EVAL, which inserts an active tuple into the space. This active tuple allows Linda implementations to spawn work in the tuplespace. This tuple can be read or withdrawn only when it returns to a passive state after completing its active process. The EVAL primitive is not feasible for multiple space implementations because it brings up requirements in security and fairness. In addition, there is no clear definition for implementing an EVAL operation, and the machine's or the overall system's available process creation primitives decide the semantics of EVAL operation [FP97].

2.2.2 Properties

Generative communication has the following fundamental characteristics: *Communication orthogonality,* due to which neither the sender nor the receiver have prior

knowledge of each other, which allows a type of asynchronous communication between them. This has two important consequences: *space-uncoupling* and *time-uncoupling*.

Space uncoupling allows spaces to exchange tuples without knowing the identity of the process that inputs or outputs the tuple. *Time uncoupling* takes away the requirement for two processes to be present at the same time. Any tuple inserted into a persistent space waits for its matching dual, even beyond the lifetime of the originating process.

A third property, *distributed sharing* is a consequence of the first two. Any shared variable deposited with the tuple-space is assured of atomic maintenance through the tuple-space operators.

2.3 Consistency Models

The memory consistency model of a tuplespace specifies how this shared memory will appear to the programmer. We provide an overview of generic consistency models and try to fit in those for a centralized as well as multiple tuplespace implementations.

For a uni-processor program execution, the order of memory accesses is strictly according to the sequential order specified by the program, called the *program order*. For a multiprocessor system accessing a common global memory we need the existence of a global time, which can strictly determine the sequence of operations executed by multiple processors. If such a clock were to exist we would have a very restrictive model characterized by the condition that a read to memory location x returns the value stored by the most recent write operation to x. This model is referred to as a *strict consistency*, and the coordination protocol required for implementing such a model limits the scalability of a multiprocessor system [GGH91].

The uni-processor model can however be extended to apply to multi-processors in a natural way. The resulting model is called *sequential consistency.* Sequential consistency requires that all memory operations appear to execute one at a time and that all operations of a single processor appear to execute in the order described by the processor's program order [AG96]. Thus unlike strict consistency, processes do not have to agree on the exact time, but have to agree on an exact operation order. Thus for a system with multiple replicas of data objects, sequential consistency requires a memory

operation to execute atomically with respect to other memory operations. Now if this global memory is distributed across multiple machines, the requirement for sequential consistency extends such that any operation on a data object has to be visible in program order to all the replicas of that data object.

Sequential consistency provides a simple, intuitive programming model. However it disallows uni-processor optimizations that would help in minimizing read or write access latencies. Most of the programming languages present simple sequential consistency semantics. This requires that memory operations should be executed in the program order. However, we can relax this restriction and re-order operations to achieve optimizations. At the same time by ensuring that two operations are executed in program order only if they are to the same data object or if one controls the execution of other, we can support the illusion of sequential execution.

Relaxed memory consistency models have the following characteristics: First, they relax program order requirement. In distributed shared memory systems this is done typically by inserting synchronizing operations before accessing critical sections of the code. Second, they relax the atomicity requirement. For example, in an SMP environment this is mostly done by using some hardware optimization that allows a subsequent read to access the value of another processor's *write*, even if that write was not made available to all the nodes.

2.4 Semantics for Centralized Tuplespace Primitives

In this section we look at the behavior of tuplespace primitives over a central tuplespace server and the consequences of ordering concurrent tuplespace operations. As discussed in section 2.2 the semantics of tuplespace primitives suit a distributed environment where it is not known which event is the most recent.

The following discussion about tuplespace semantic analysis is based on the discussion in the GLOBE [LS99] thesis. We study two or more tuplespace primitives as they operate concurrently on a centralized space. Two arbitrary operations received by a tuplespace server can be classified as being either sequential or concurrent. For sequential tuplespace operations, we are assured of a well-defined result. This is also true for concurrent tuplespace operations that do not *interfere*. Operations are said to *interfere*

when there is a race condition for access to a single tuple or if one operation controls the execution of other concurrent operation. Example: When two concurrent IN operations match on a single tuple.

For interfering concurrent operations, we need to inspect the outcome of ordering such operations. Table 2.1, shows the dependencies among combinations of all tuplespace primitives. The dependency between semantic result and internal ordering of concurrent operations is classified into three categories: Strongly dependent (S), Dependent (D) and Independent (I).

	OUT(t)	READ(t)	IN(t)	READP(t)	INP(t)	S: Strongly Dependent D: Dependent I: Independent
OUT(t)	I	I	I	D	D	
READ(t)		I	D	I	D	
IN(t)			S	D	S	
READP(t)				I	D	
INP(t)					S	

Table 2.1 [LS99]: Dependency between semantic results of tuplespace primitives due to internal ordering of concurrent operations.

Tuplespace operations are assumed to execute with sequential consistency and the programmer is required to introduce enough synchronization to suit this model. But even this sequential consistency seems to be restrictive and we could optimize the behavior of individual space primitives to reduce the number of messages in the system.

Tuplespace primitives can be classified into synchronous and asynchronous operations. The former have to be executed with sequential consistency, while for the later consistency can be relaxed.

2.4.1 Synchronous Operations

The IN primitive is a blocking operation, which synchronously extracts a tuple from the tuplespace. In other words, if any process requires exclusive access to a tuple, it executes the IN operation. The semantic result of the execution of two concurrent, interfering IN/INP primitives is strongly dependent on the order of execution. In other

words, when two IN/INP operations match on a single tuple, either of them is equally likely to remove the tuple from space and the other would block. Since the two operations are concurrent either of the results are valid from the process point of view.

2.4.2 Asynchronous Operations

The OUT primitive is classified under asynchronous operation. Execution of the OUT primitive results in insertion of a tuple into the tuple space; the executing process continues immediately. The semantic result due to internal ordering of a concurrent OUT primitive with any other tuplespace primitive is mostly independent of the sequence of execution. In other words, the OUT operation can be bypassed by any other tuplespace operation (READ, IN) without affecting the outcome of either of these operations.

The READ operation is an asynchronous receive, which acts like IN, only it does not remove the tuple from tuplespace. Thus, the semantic result of executing two concurrent READ operations is independent of the order. In other words, two read operations can be overlapped or their order of execution can be changed without affecting the result. On the other hand if an IN and a READ operations are concurrent and match on the same tuple, the IN always gets the tuple regardless of ordering. The READ operation will get the tuple if it is ordered before the IN, otherwise the READ will block. Thus for the IN-READ (INP-READP, IN-READP, INP-READ) combination, the semantic result is dependent of internal ordering of concurrent tuplespace operations. But again, the tuplespace operations are globally unordered and either of the results is semantically valid.

2.5 Consistency model for a Distributed TupleSpace

Large parallel systems require distributed tuple-space implementations: tuples should be distributed among multiple space servers and made transparently available to every node of the system. The consistency semantics for a distributed environment should not differ from that of a centralized space. As mentioned in the previous section the relaxed consistency of certain primitives can be used to our advantage in building distributed systems with multiple tuplespaces. We illustrate the consistency issues over two spaces and with two interfering and concurrent operations, one executing on each of the spaces. In the distributed environment, we define operations as concurrent if the result

of one operation is not visible in the entire distributed tuplespace when another tuplespace operation is performed. Compared with distributed shared memory systems, tuplespace matches relaxed consistency. It provides separate operations to modify synchronously or otherwise.

According to S. Chiba [CKM92], a tuplespace is consistent if:

1. An OUT precedes a READ and an IN that manipulates a tuple generated by that OUT,
2. An IN is never followed by a READ that reads a tuple removed by it, and
3. Two different INs do not remove a tuple generated by the same OUT.

2.5.1 Synchronous Operations

As we explained in section 2.4.1, the semantics of executing two concurrent interfering IN operations is strongly dependent on the order of execution. Because IN is synchronous and modifies the state of the tuplespace, global atomicity is required in executing this operation. The race condition when two IN operations attempt to extract a tuple from a centralized tuplespace can be easily resolved by serializing the operation over the central space. But for a multiple tuple space implementation, two IN operations may be performed on different tuplespace replicas. In such a case if the operation were not performed with global atomicity over all the replicas, both operations would yield the same tuple from the distributed tuplespace, which would be semantically incorrect. Because IN is synchronous, its effect in the form of an “invalidate” request has to immediately propagate to all the replicas upon which its template matches.

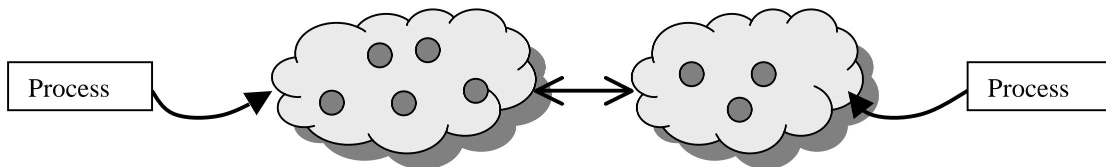


Figure 2.2: Two processes concurrently performing IN on multiple tuplespaces.

2.5.2 Asynchronous Operation

Combinations of the orderings of an OUT operation with any other operation do not interfere when performed. Thus, they can be performed in any order regardless of a centralized tuplespace or a distributed tuplespace and it is not necessary to enforce global atomicity. Relaxed consistency can be used in executing this primitive, hence effects of OUT need not be immediately applied to replicas in other spaces.

The READ operation being non-destructive, it need not be propagated to all the replicas. In section 2.4.1, we discussed that the order of execution of two reads can be reordered without affecting the result. But, there was dependence when an interfering IN and READ were executed concurrently. Those outcomes are also possible in a distributed tuplespace, but since we do not insist on global atomicity for READ primitives, it is possible that a replica is read from a space after an IN primitive has removed the tuple from some other space. This may be due to the delay in the propagation of an 'invalidate' (associated with an IN operation) to the concerned replica. However this situation is not semantically incorrect, as there is no global time in this distributed scenario. However, the atomicity of the IN operation should assure that if a READ is executed in program order after an IN, it should not read the tuple removed by IN.

2.6 Replica update protocols

Based on the consistency model as discussed in section 2.3, we discuss various replica update protocols and consider their tradeoffs in performance and the objective of maintaining sequential consistency for IN primitives.

2.6.1 Protocols for a flat partitioning of Spaces

S. Chiba [CKM92] proposes three protocols for IN: the strict protocol, the non-exclusive protocol and the weak protocol, based on decreasing order of consistency. These are shown in the figure 2.3.

The *strict protocol* guarantees that replications are consistent in any case. It is based on the 2-phase commit protocol and makes sure that all the replicas are removed before a tuple is yielded to an IN operation. When any node attempts to remove a tuple, it sends 'lock' message to all the replicas. The nodes receiving this message check the

status of their local replica and send an ‘accept’ message if the tuple has not been locked; otherwise a node replies with a ‘refuse’ message. When a node receives all ‘accept’ messages, it removes the tuple and sends ‘delete’ messages to other replicas. Otherwise, the removal fails, the lock is released and operation tries for another tuple. This protocol is usually used with *Active Replication* mechanism of establishing global atomicity. In *active replication* as shown in figure 2.4, all the replicas have the same responsibility of establishing global atomicity. This scheme is used to implement a distributed tuplespace in FT-Linda. Overall, active-replication with the two-phase commit protocol for tuple replication requires a lot of messages to be transmitted to ensure global atomicity, and is costly in terms of performance and scalability.

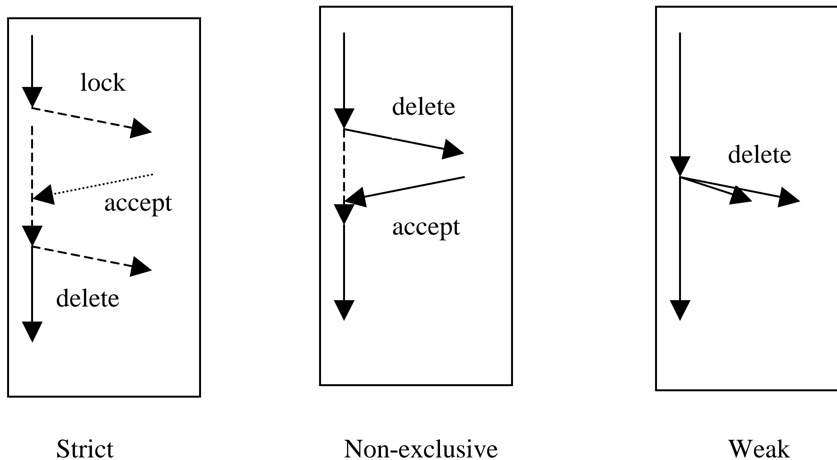


Figure 2.3: [CKM92] Protocols for maintaining global atomicity of IN operation

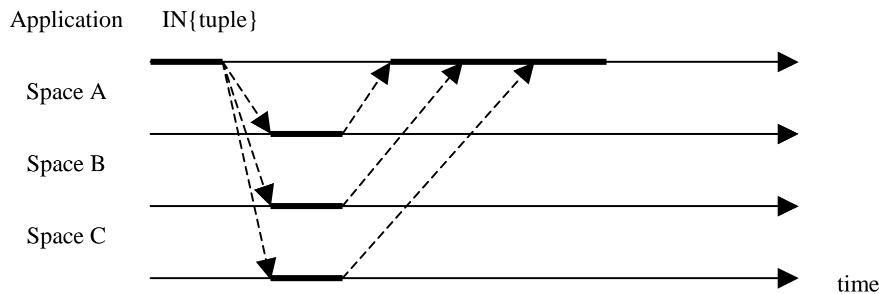


Figure 2.4: Active Replication

The *nonexclusive* IN protocol provides consistency only when the IN operations are non-interfering. Other READs can try to read it simultaneously. This protocol uses fewer messages than the strict protocol but guarantees a weaker consistency. The ‘delete’ messages are sent only once, and replies to them are returned. To make this protocol sequentially consistent, it can be combined with the *Primary Copy Replication* protocol. In Primary copy replication, one of the tuple copies is considered as the *primary.* Before an IN operation is successfully returned over any of the replicas, ownership over the *primary* is required. Thus, the *primary* acts as a serialization point for concurrent, interfering IN operations. Figure 2.5 illustrates a scenario of the Primary copy replication. In this protocol changes to the primary are immediately propagated to the other replicas. We can relax this tight implementation by propagating changes in batches and performing multiple changes at the *primary* tuplespace.

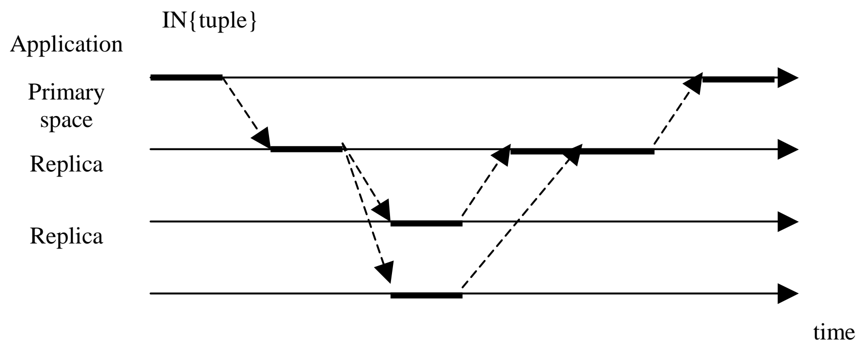


Figure 2.5: Primary copy Replication

Weak Protocol guarantees the weakest consistency and requires fewest messages. In this protocol, the node at which the IN operation executes, sends ‘delete’ message to other replicas. This protocol does not wait for acknowledgements from other replicas and goes on to the next step before replicas are invalidated in other spaces. Therefore, with this protocol it is likely that a READ operation executed sequentially after an IN operation would return the same tuple that the IN removed. Tuples that no other tuplespace operation manipulates can be removed by IN according to this protocol. This

protocol performs very well due to its minimal cost, but cannot be used in a distributed environment that promises sequential consistency.

Another technique for spaces not arranged in a topology, is the *weighted voting* protocol as used in the design by S. Kambhatla [Kam91]. This protocol is based on distributed control and uses the voting mechanism to decide on replica update based on sequential consistency.

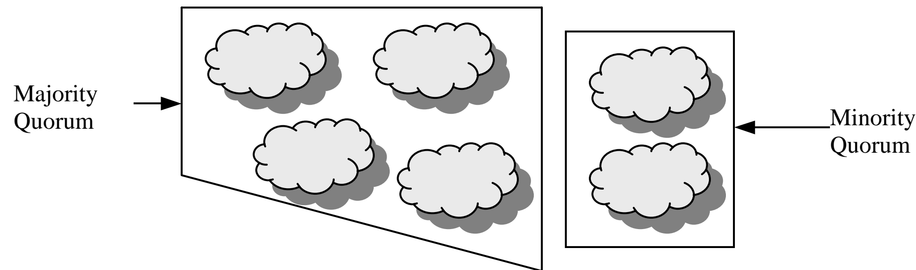


Figure 2.6: Voting protocol

In this protocol each tuplespace replica in the distributed tuple space is assigned one vote, and any of the spaces can initiate tuple withdrawal on receiving an IN operation. That replica now acts as a primary and communicates with the other replicas of the partition to invalidate their tuples. This approach potentially leads to a race condition as multiple replicas can initiate withdrawal of the same tuple. However, voting ensures that the replica that gets the maximum number of votes in its partition succeeds in removing the tuple. The approach potentially leads to a deadlock, if multiple replicas manage to get equal number of votes. This deadlock can be solved by introducing a random delay to competing, deadlocked processes. This protocol requires a prior knowledge of the number of replicas in the partition before the quorum could be established through voting.

2.6.2 Protocols for spaces arranged in a network topology

For spaces arranged according to a network topology, tuples and templates are replicated such that there is a non-null intersection node, to allow a match. Therefore,

after the match the system has to ensure that all the replicated tuples are invalidated to prevent some other template from matching with the tuple replica. In addition all the templates have to be removed to ensure that a single IN operation does not remove two tuples. This is a distributed protocol as any node can initiate the update. The complexity and cost of this protocol is in proportion to the system size.

2.6.3 Discussion

In our implementation, we do not replicate tuples or templates along a network topology partition. So the only protocols suitable for our implementation would be the strict protocol along with active replication, the non-exclusive protocol with primary copy replication and the voting based replica update protocol.

We use the non-exclusive protocol along with primary copy replication to establish global atomicity in executing the IN operation. This protocol uses fewer messages than the *strict protocol* and scales well as the number of replicas increase. Also, we need not know the number of replicas a priori for the execution of this protocol, as required in weighted voting schemes.

Chapter 3

Tuple Replication Issues

In the previous chapter we explained the tuplespace paradigm using the Linda system model. We discussed different tuplespace primitives and studied the semantics of their execution. We also talked about consistency models and protocols used for maintaining global atomicity of IN primitives in a distributed environment.

In this section we look at different tuple replication policies. Replication of data has been used in DSM systems to increase availability and to decrease access time to shared data. Distributed tuplespace implementations base their replication policy on the requirement that it should grant any potential tuple-template match to occur. In other words, for a template P, if there exists a matching tuple T anywhere in the system, the replication policy should guarantee their match somewhere in the system. Thus we have tuples replicated over multiple spaces and scalable solutions must be found to deal with the consistency problem. The degree of coordination among multiple spaces as required by the coherence protocols characterizes the scalability of a replication policy.

3.1 Replication Policies over Multiple spaces

Replication policies over tuplespaces have been classified as (1) State Replication, or (2) Operation Replication. With the *state replication* approach, as used in FT-Linda [BS95], the state of the space data structure is serialized and replicated on multiple spaces. This replication has to be done on every operation, and it is a costly operation if the data structure is large and changes frequently. An optimization would be to replicate only the changes in the space data structure and is referred to as partial state replication. As the data structures are implementation specific, the technique may not be reusable among multiple tuplespace implementations.

The *operation replication* approach is most common among tuplespace implementations [XL89][LS99][CG85]. Individual tuplespace primitives are replicated across all or a part of the multiple space system, depending on the degree of replication and the consistency of the primitives. In other words, rather than the primitives changing the state of one space and then propagating the state-change to all the spaces, the primitive itself is sent to all the replicas. We use the *operation replication* policy for our implementation, as this approach is more scalable.

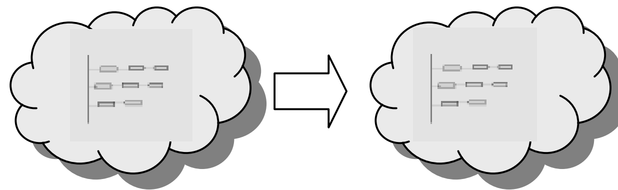


Figure 3.1 (a): State Replication

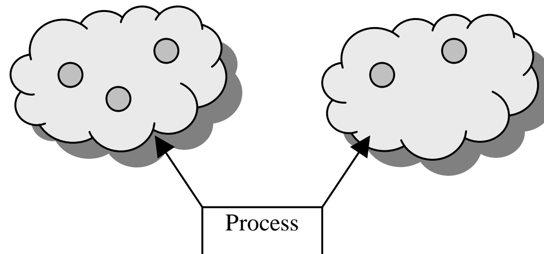


Figure 3.1 (b): Operation Replication

3.2 Full replication

Full replication techniques [Lei89] [AB89] [CG85], can be used in two ways. One option, referred to as *negative broadcast*, is to store the tuple generated by the OUT primitive at the local node. Templates generated by the IN and READ primitives are sent to all nodes. The dual technique is the inverse of negative broadcast or *positive broadcast*. Tuples are broadcast to all nodes with the OUT primitive, and the node executing the IN and READ primitive tries do a local match. Both these solutions are simple and make no assumption about the topology of the system. These schemes also

provide the best fault tolerance. However, since they are based on broadcast and global replication, the scalability of such systems is limited. The cost of needed coherence protocols increases with the system size and becomes a bottleneck for the implementation.

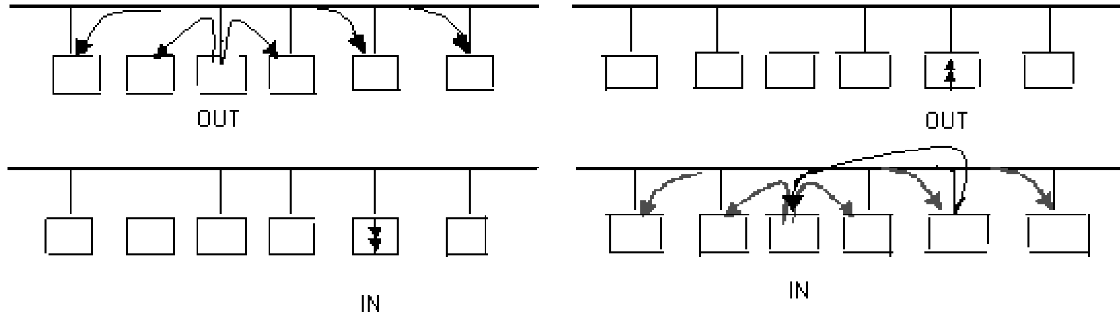


Figure 3.2 (a): Positive broadcast Figure 3.2 (b): Negative broadcast

3.3 Partial replication

Partial replication techniques partition the tuplespace based on the following techniques: (1) Deterministic replication, or (2) Non deterministic replication.

3.3.1 Deterministic Replication

With deterministic replication, tuples are assigned to nodes of the system on the basis of hash keys generated from the type signature of the tuples. Hence, tuples/templates belonging to the same type are allocated to matching subset of nodes, so as to allow a possible match. Deterministic allocation allows tuplespace clients to access spaces directly without sending requests to several space replicas [Bjo93].

However, the hash solution does not guarantee a homogenous distribution of tuples among spaces. This is because of the observation [FP97] that the number of type signatures that identify tuples for many applications is typically small. Also, it does not exploit the locality principle, as tuples are not stored close to the node where they are produced.

3.3.1.1 Optimization of Distributed Tuplespace messages

Static or dynamic optimizations at compile-time or run-time help reduce the average access time of tuples by changing the tuple/template distribution patterns. In the following sections, the optimizations proposed are for a deterministic hash-based protocol. In this protocol tuples/templates are replicated to partitions based on their hash value and type signatures.

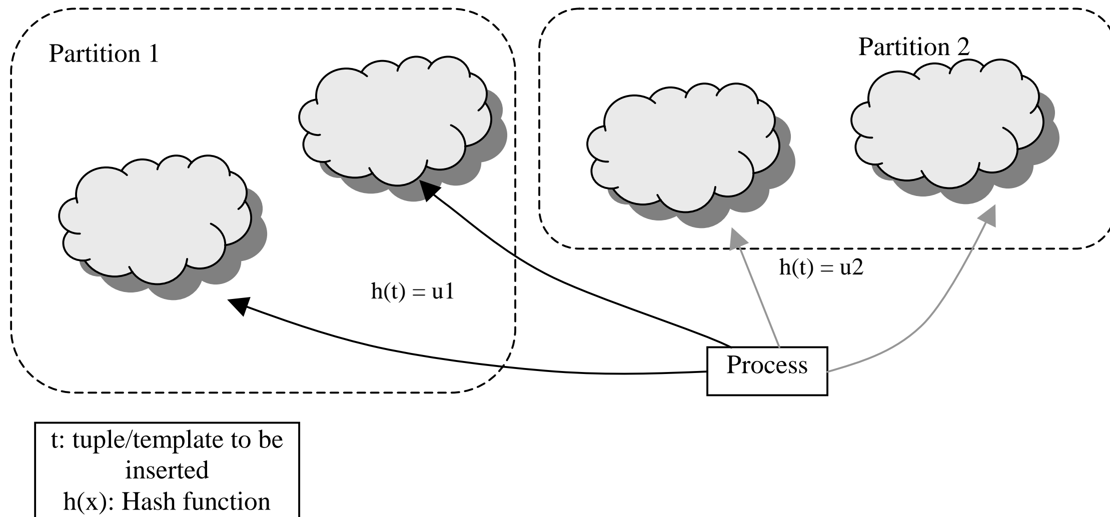


Figure 3.3: Hash based partial replication

3.3.1.2 Compile time analysis for tuple distribution

Compile time analysis is used to detect patterns of tuple access. The analysis done by Bjornson [Bjo93] is as follows: A pattern recognizer looks for IN-OUT operations and updates the tuple in the space. In other words, the IN-OUT operation is replaced by a single operation that updates the tuple within the space itself, instead of extracting it for modification and putting it back to space. Another optimization for multiple-tuplespace implementations is to detect existence of a READ operation and replicate tuples for that partition.

Fenwick's compile time optimizations [FP96] decide replication patterns based on 'shared variable' analysis. Examples of shared variable tuples are 'Head' and 'Tail' tuples of a distributed stream data structure. In such a data structure the head and the tail tuples are accessed and updated frequently by multiple processes during node insertion and deletion. Optimization of a shared variable tuple minimizes the number of messages,

which results in a factor of 2 speedup in access to the tuple. For example, if in a Linda program a tuple is shared by a smaller number of processes, using direct messages rather than an expensive broadcast may improve performance. If a shared tuple is read more often than it is written, replicating the tuple may improve performance by making reads local and inexpensive. The replication would be at the cost of update protocols for an infrequent IN operation.

3.3.1.3 Runtime Optimizations

Bjornson [Bjo93] identifies two runtime techniques for improving the performance of tuple distribution strategy. In the first strategy, called *Rendezvous reassignment*, the runtime system keeps careful statistics about communication patterns of tuples and assigns rendezvous nodes for tuples and templates based on the on these patterns. For example, as shown in figure 3.4, consider a producer consumer type application such that tuples sent to partition X are matched with templates sent to space Y. In such a case the optimization would lead to tuples being directed to space Y instead of all the spaces in partition X.

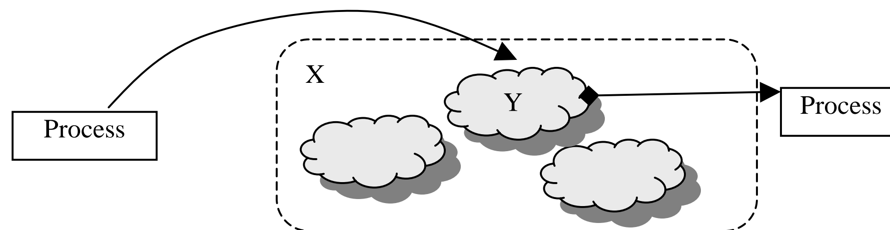


Figure 3.4: Rendezvous reassignment

Bjornson's other dynamic optimization is called *randomized rendezvous node*. Again, the runtime maintains detailed statistics to determine if the *rendezvous node* is being overloaded. The runtime then distributes the workload among other spaces in the partition.

3.3.2 Non Deterministic replication

Non-Deterministic replication does not make a priori decision about the spaces where tuples would be replicated. Tuples/templates are replicated along a network topology such that there is always a non-null intersection node for matches. These techniques try to provide matches close to the nodes where tuples/templates are generated. The fault tolerance of these systems is expressed in terms of number of nodes on which the tuple is replicated and the scalability is inversely dependent on the number of replicas required to be contacted for maintaining consistency. For performing the IN operation with global atomicity, the position of the tuples in the network topology or in the hierarchy is put to use.

3.3.2.1 Grid based replication

Here we have a 2-D grid topology, as shown in figure 3.5, with tuple spaces at the intersection of the horizontal and vertical connections. This design was proposed by Krishnaswamy [Kri91] as an approach for building a Linda coprocessor. When a process performs an OUT operation, the tuple is replicated along the column of the grid of its source node. Another process performing an IN would send the template along its row of the grid. This protocol therefore guarantees a tuple/template match at one of the spaces in the grid. With this technique it is not possible to deterministically say which of the spaces would have a potentially matching tuple. Therefore, the tuple/template has to be replicated over the entire column/row. When a tuple is found, all of its replicas along the column have to be invalidated. The replication degree of tuples and the cost of associated coherence protocol grow proportional to square root of system size.

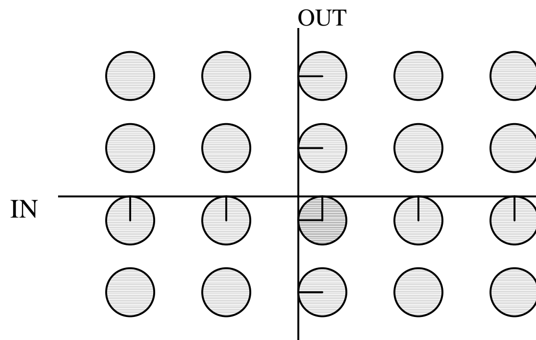


Figure 3.5: Grid based replication

3.3.2.2 Tree based replication

The tree based replication model consists of the space nodes arranged in a tree hierarchy, and with the clients (execution nodes) as the leaves of the tree structure. The design was proposed by Corradi [CLZ97] and implemented on a transputer-based architecture. When the processes execute the space primitives, tuples and templates are replicated vertically along the subtree till they reach the root. The tree-based topology assures that there is at least one common node between any two tuple/template propagation paths. As an optimization, the replication process is stopped at the node where a match is found. The main point of this implementation is to make the replication degree of tuples grow logarithmically with the system size and allow multiple space references to take advantage of the physical locality in a sub-tree.

In the tree implementation, both tuples and templates are replicated. Therefore the coherence policy assures that the conditions stated in section 2.4 are followed. When any IN-template extracts a matching tuple from the node, it retracts its steps down the branch on which the tuple has replicated. If it finds that the tuple has been removed from any node below in the hierarchy, it replaces all the removed tuples. Thus for concurrent, interfering IN operations, the IN primitive that matches closer to the execution nodes has a higher preference than the one up in the hierarchy. Tuple extraction succeeds when the IN-template reaches the leaf and an OK message is sent to the matching node. Similarly, the tuple which matches with a template closer to the leaves is extracted and has a preference over the tuple that matches higher up in the hierarchy with the same IN-template.

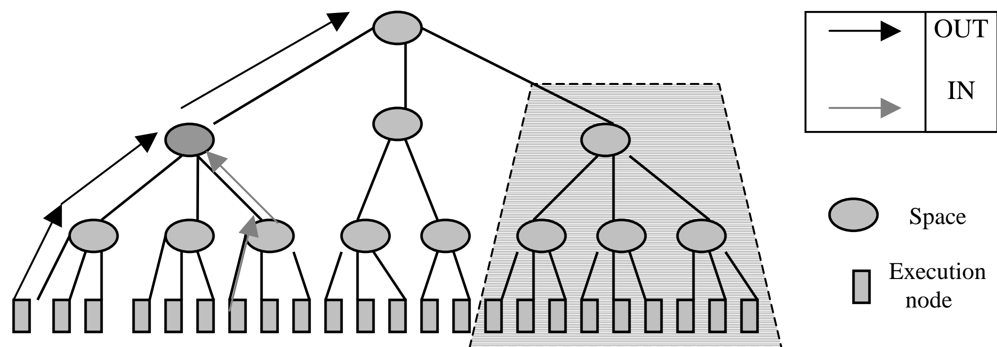


Figure 3. 6: Tree based replication

3.4 Cache Only Memory Architectures

The Cache-Only Memory Architecture (COMA) [DT99] is a Distributed Shared Memory (DSM) model, which is analogous in some ways to the tuplespace systems. These systems support a logical shared address space in a loosely-coupled environment by replicating and migrating virtual memory pages, in response to memory accesses by application processes.

In COMA each processor holds a portion of the address space. However the partitioning of data among memories is not static, since all distributed memories are organized like large caches. Each memory module acts as a huge cache memory in which each block has a tag with the address and the state. COMA increases the chances of data being available locally because the hardware transparently replicates the data and migrates it to the memory module of the node that is currently accessing it. Thereby, COMA promises a better average access time to data than traditional Non-Uniform Memory Architecture (NUMA) multiprocessors. This is because the large local memory module is more capable of containing the node's current working set than a cache is. Therefore, more of the cache misses are satisfied locally within the node.

3.4.1 Localization and Replacement

In COMA systems, a block's address is a global identifier, not an indicator of its physical memory location. The local memory keeps information about the address and the state information of the data block (tag). On an access, the memory controller has to look up for this tag to determine whether or not the access can be served locally. There is no physical memory home location for a particular data item; therefore the distribution of data is dynamically adaptable to the application behavior.

The first COMA designs, the KSR-1 from Kendall Square Research [FBR93] and Data Diffusion Machine (DDM) from the Swedish Institute of Computer Science [HLH92], are characterized by hierarchical network topologies. These hierarchical topologies simplify two main problems in these types of systems: location of data block and replacement. Replacement is required when the local memory overflows and a chunk of data, which may be the master copy, has to be relocated to another local memory.

3.4.2 DDM Architecture

The DDM [HLH92] as shown in figure 3.7, has a hierarchical organization of directories above the processor and attraction memory nodes. The directory keeps information for all items in the attraction memory below it. The directory structure passes through only those transactions from below that cannot be completed in its subsystem or those transactions from above that can be processed in its subsystem. If the subsystems connected to the lowest bus cannot satisfy any memory request, the next higher directory retransmits the request on the next higher bus. The state of the directories at each level is modified for the request, till the request reaches a level in the hierarchy where a directory containing a copy of the item is selected to answer the request.

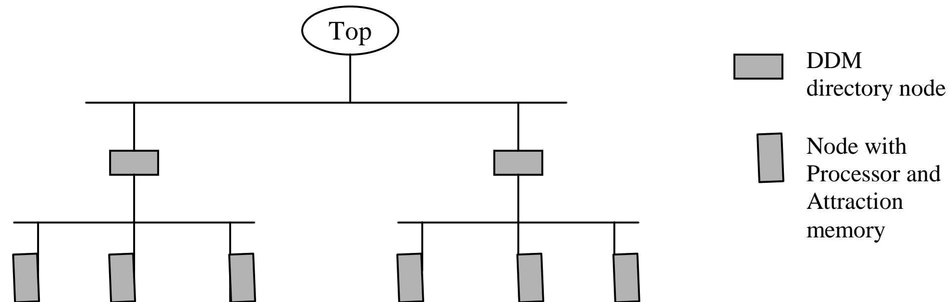


Figure 3.7: Hierarchical DDM

3.5 Summary

Full replication protocols based on broadcasting either of the tuples or templates, are good in terms of fault tolerance, but are expensive in usage of network bandwidth and are not scalable.

Partial replication protocols are an improvement over full replication protocols. This is because the cost of replicating tuples, and further the cost of maintaining global updates for sequential consistency is reduced. The *deterministic* partial replication protocol manages the distribution policy by allocating on the same nodes all tuples of the same type and grants any possible match to occur. But the protocol lacks scalability, as it is not able to achieve a homogeneous tuple distribution, especially when the number of nodes is high. Consequently, in large sized systems, a hash solution is likely to produce a

high overhead of communication and would increase access times to remote tuples. Non-deterministic partitioning allows tuple distribution along a network topology and tuples are partitioned independent of their type and hash function. This technique assumes good distribution of applications along the partitions. This approach is expensive as regards to the cost of establishing global atomicity for IN operations, as the degree of replication is in proportion to system size.

The COMA architecture is analogous in some ways to the tuplespace systems in which we could attract multiple remote tuples that match a particular template to be replicated or migrated to a space server. This would reduce the access times to these remote tuples for subsequent accesses with identical templates as they can be found locally. We are interested in the data block location technique with the hierarchical COMA designs like DDM. This is because their memory access operations are similar to tuplespace primitives. Moreover, the search policy for locating a dynamic data block could be applied to tuplespaces.

Chapter 4

Directory based design

In the previous chapters we discussed different protocols for tuple distribution and the associated coherence protocol required for maintaining global atomicity for IN primitive. In this chapter we present arguments for building a scalable tuplespace and suggest our directory-based model to support a scalable and efficient design.

4.1 Directory based model

The idea for the directory-based model has been derived from many sources. Prominent among them is the hierarchical architecture of the DDM COMA [HLH92] machine. Also important is the tree based multiple-tuplespace implementation [CLZ97]. Based on these we present our model for implementing distributed tuplespaces. However, this is the initial work in this direction and hence we make several simplifying assumptions in order to focus on the scalability problem. One assumption is that all the space servers in the system along with the networks over which they communicate are reliable. Second, we assume that all the space servers involved with this model are evenly loaded. An interesting area for future work is to relax these assumptions.

In order to capture the locality between the space servers, we use a clustering based network model. The space servers are clustered according to their network topologies as shown in figure 4.1. For example, space servers that belong to different departments of a single university may be clustered together. This cluster structure can be captured using a tree, based on the network-locality of the nodes. The space servers may form the leaves of the tree, such that servers that belong to a cluster are located in a subtree. A directory forms the root of such a subtree, and further more directories are arranged in a tree-based hierarchy as shown in figure 4.2. The spaces themselves are

arranged in a planar topology with a hierarchical directory structure above them. Thus we achieve partitioning of spaces based on their network topology.

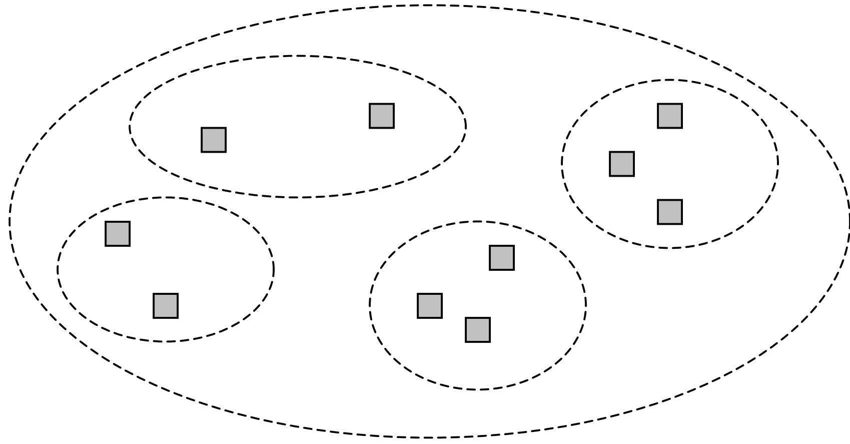


Figure 4.1: Clustering of Space servers

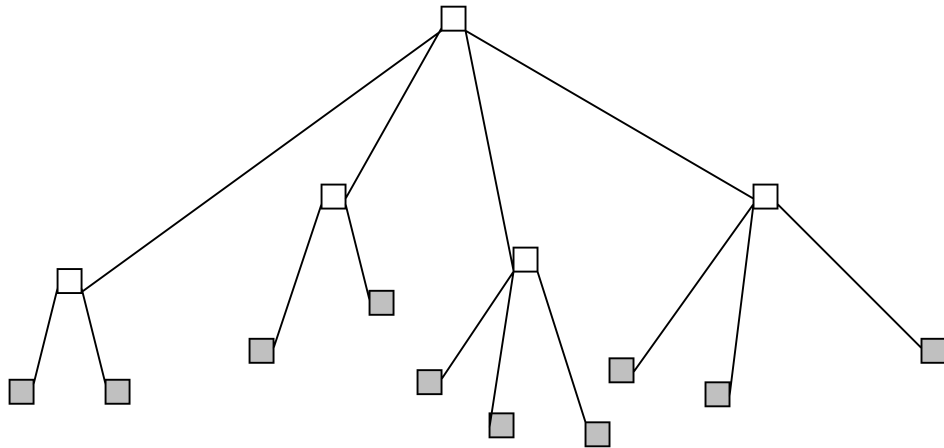


Figure 4.2: The cluster tree, connected by a hierarchical directory structure

Each space server is accessed by distributed client processes, which we refer to as execution nodes. Thus a set of execution nodes has to know only about a single 'local' space server. For example, we may have a single space server in a subnet interfacing with processes running on machines in that subnet. This space server is further registered with a directory OR we can say that the space server is using the directory service to

explore tuples/templates in other spaces. Thus, each space server provides the abstraction of a global space, which is formed by a coalition of multiple spaces.

4.2 Hierarchical Directory

In a tuplespace model, data is accessed associatively and we do not have a fixed address to look for (as in a DSM system). Thus unless there is a common space server where a tuple and a template can physically meet, a template due to an IN or READ will not know where it has to search for its matching tuple.

In space-based systems, the main purpose of tuple/template replication has been to assure a match over multiple spaces. As we have seen in the previous chapter, the hash based replication provides the least latency for tuple/template accesses, but, this method is not scalable due to non-homogenous distribution of tuples. Alternatively, with the tree-based implementation the tuples/templates are propagated along a branch of the subtree. The degree of replication can be adjusted up to a certain depth, thus setting limits on the scope of the tuplespace. The advantage of this approach is that the degree of replication grows logarithmically with the system size. We are interested in the tree-based model, because we want to exploit locality, by virtue of which we get tuple/template matches closer to where the tuples are generated. At the same time we want to avoid the bottleneck associated with links and nodes, up in the tree-based hierarchy.

Therefore, in our design we allow tuple signatures to be propagated up a tree-based directory hierarchy. The directory then provides information about the spaces where the tuple might be located. There are a couple of reasons for using directories and not actual spaces in the hierarchy. First among these is the fact that, on an average, tuple signatures are assumed to carry a much lesser percent of the actual tuple. This reduces the demands on bandwidth requirement as well as serialization time (for the tuples to be serialized and passed through the network). Second, the directory structure allows templates and tuples registered with different space servers to locate each other. Thus we need not replicate tuples to allow tuple/template matches over multiple spaces. This takes away the additional requirement of implementing costly protocols to invalidate replicas that have been distributed only for the sake of assuring a match. The tree based tag propagation puts a definite bound for tuple/template searches. Thus, any tuple/ template,

generated anywhere in the system of associated space servers is made visible to its dual as soon as it gets registered with the directory hierarchy.

The protocols required for executing remote space operations run among the space servers and do not make use of the directory hierarchy. Tuples could be selectively replicated for the purpose of reducing access latencies or to increase availability.

4.2.1 Directory Tags

Each directory stores entries in the form of signature tags. These tags represent a hash-based description of the tuples as shown in figure 4.3. Each field of the tuple is hashed to form a unique tuple signature. This tag also carries a vector list of the names of the spaces (space handles), where the tuple was originally stored and then replicated. The tuple that is inserted in a local space due to an OUT operation is referred to as the '*primary*' tuple. All the subsequent replicas and tags of this tuple carry the name of the local space as the first element in their vector list of addresses. With every replication, the name (handle) of the remote server is appended to the vector list of addresses. The *primary* tuple thus maintains a list of space servers where the tuple has been replicated.

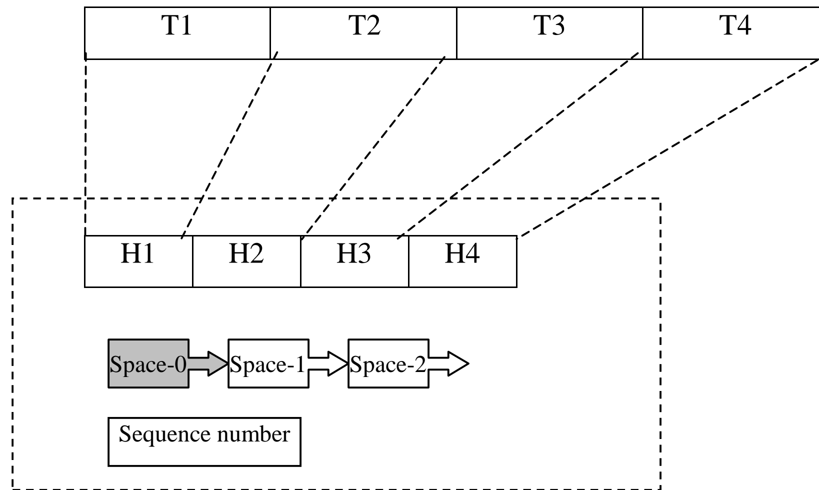


Figure 4.3: Tuple tags

4.3 Primitive Distribution

After discussing the basic physical layout of our model, we further discuss the propagation of different primitives through the directory structure. We also talk about the

protocol by which a local space server distributes tuples/templates to remote spaces, transparent to the execution nodes.

4.3.1 OUT

This is an asynchronous primitive and the semantic result is independent of the order of execution of other concurrent blocking primitives. We send the tuple associated with the OUT primitive to the local space. After this, the application has no further responsibility for its replication. This tuple is further replicated and registered as a signature tag with the hierarchical directory structure. As such, the application does not have to wait till the tuple tag reaches all the directories.

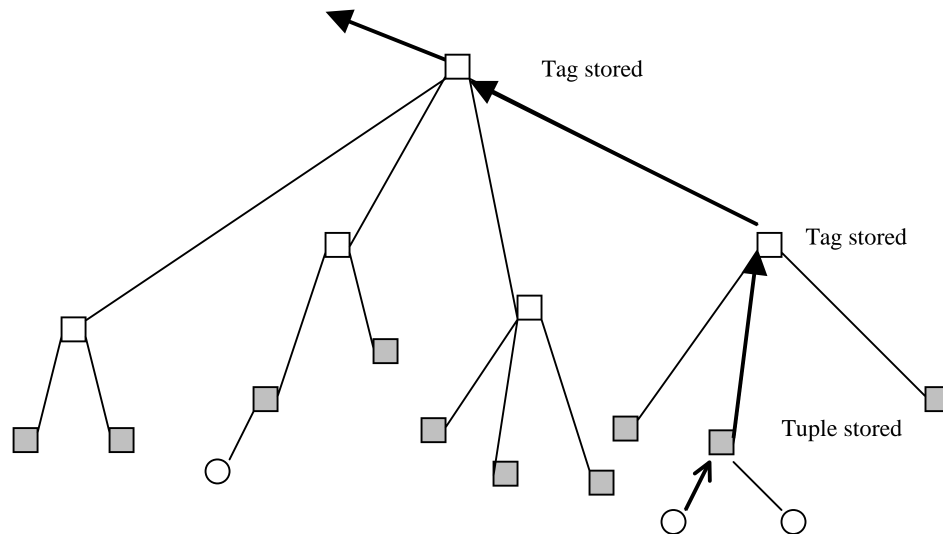


Figure 4.4: Tuple propagation due to OUT primitive

4.3.2 IN

This is a synchronous primitive and global atomicity is required so that a subsequent IN or READ operation does not find a replica of the tuple that has been already removed by a previous IN operation. Therefore this operation is executed with the stronger form of consistency, which is *sequential consistency*. The IN template is stored with the local space. If a match is not found, the template registers its interest and the signature template is sent to the directories. The signature is a per-field hashcode of the template and the formal fields (that act as placeholders) are masked out.

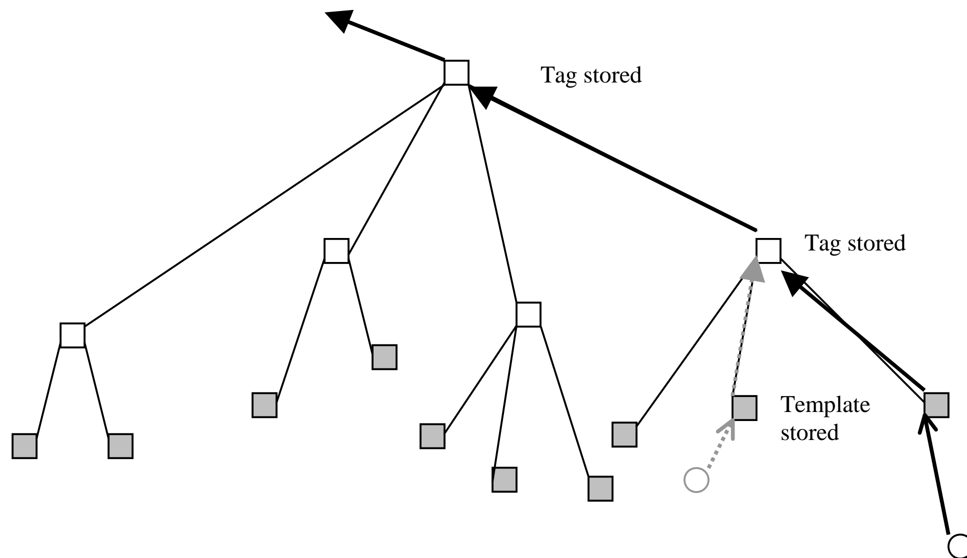


Figure 4.5: Template propagation due to IN/READ

The above tuple/template distribution scheme assures that the tuple and template signatures would meet at one of the directories in the hierarchy. As an immediate optimization, the distribution of the tuple/template signature stops as soon a matching dual signature is found. If a match is found with the local space, the IN operation should assure that the tuple for which there was a match is yielded to the process that initiated the IN operation; and if any replicas or corresponding directory entries exist they have to be invalidated. If there is a match between a tuple and a template signature in the directory, the directory sends back the tuple signature along with its space-name vector to the space that initiated the template signature replication. The space then sends an INP operation to the space that contains the *primary* tuple. If there is a match within the remote space, the *primary* tuple sends invalidates to replicas in other spaces. We use the non-exclusive protocol for maintaining consistency as it uses fewer messages. To provide sequential consistency, we combine this protocol with primary replication. In other words, multiple requests for invalidating a tuple are serialized over the *primary* tuple.

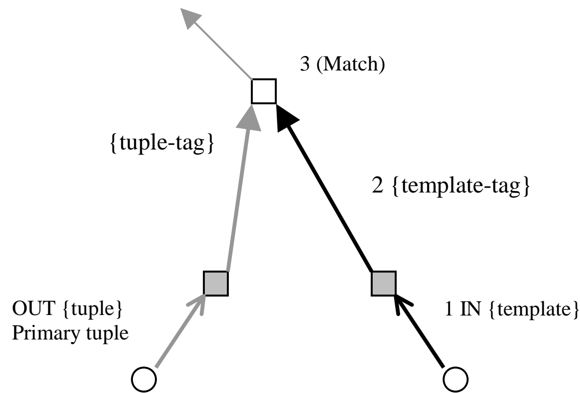


Figure 4.6 (a): tuple/template match with the directory

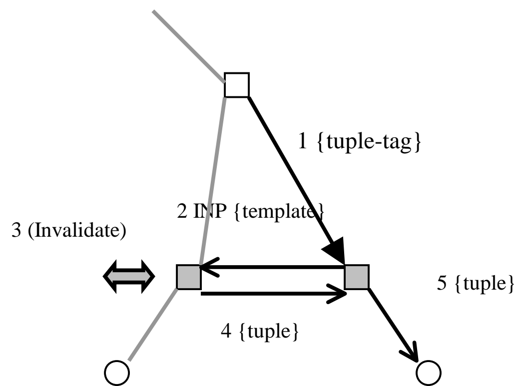


Figure 4.6 (b): Remote IN operation

4.3.3 INP

The INP operation is similar to the IN operation. The only difference is that instead of blocking if no tuple can be found in the associated tuplespace, INP returns a value to the application indicating that no matching tuple was found in accordance with loose READP/INP semantics.

4.3.4 READ

The READ operation is very similar to the IN operation in that the corresponding template is sent to the space. If there is a match within the local space, the tuple is read and yielded to the requesting process. If no match is found with the local space, the template signature climbs up the directory hierarchy. If there is a tuple-template signature

match within the directory, the tuple-tag is sent to the local space. The space then sends a template with the READP operation to the remote space. If a match is found, the tuple is replicated at the local space and the *space-name* vector associated with the primary tuple is appended with the ‘local-space’ name.

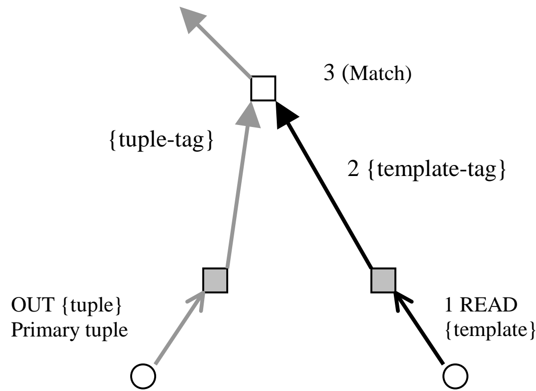


Figure 4.7 (a): tuple/template match with the directory

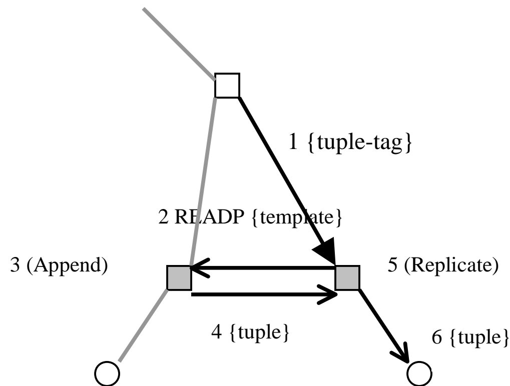


Figure 4.7 (b): Remote READ operation

4.3.5 READP

Similarly, a READP operation will be handled by the associated tuplespace. If a matching tuple is found it is returned to the application, otherwise READP returns a value to the application indicating that no matching tuple was found in accordance with loose READP/INP semantics.

4.4 Coherence Protocols

The coherence protocols associated with the IN operation is required to be checked for the following coherence conditions:

1) Any given tuple is extracted by only one template
2) Any given template extracts only one tuple

We put to test our non-exclusive protocol combined with primary copy replication to satisfy the above conditions. To test the first coherence condition, let us assume that a certain tuple has been replicated on multiple space servers. Now, any number of IN-templates that match on any of the replicas try to gain ownership of the primary tuple. After the primary tuple has been locked by a template, invalidates are sent to all the replicas of the primary. If the invalidations are successful, the tuple is yielded to the space server that initiated the corresponding IN operation. Thus multiple IN operations serialize on the primary tuple and ensure that it is extracted by only one template. When a tuple is extracted from a space server, the directory tags associated with that tuple must be also removed. Even if this is not an immediate requirement, it is necessary because their presence in the upper levels of the directory sub-tree wastes memory resources. For that, the space server could send an invalidate message through the directory hierarchy. This invalidate can be sent any time and is transparent to any of the processes.

In the directory-based model templates are not replicated, which satisfies the second coherence condition. But, multiple tuples may match with template signatures in different directories. All these directories then send the tuple tags to the local space. The space serializes all these requests, and allows the template to extract only one tuple. After the template finds a matching tuple, the directory tags associated with the template have to be removed. Again, this is not an immediate requirement; these invalidations can be delayed and performed transparent to the client.

4.6 Summary

The purpose of the study was to find a scalable implementation for the tuplespace based platform. A platform that would provide low access latencies for parallel applications executing on different nodes, spread over the Internet. This chapter described the design and functionality of the directory based tuple space model that

allows various space servers to coordinate exchange of tuples and templates. The most important issue described is the propagation of tuple signatures in a directory hierarchy. Tuplespace primitives are executed with varying degree of consistency, depending on the nature of the primitive. Directory updates can be delayed without changing the consistency semantics of tuple accesses.

The OUT operation does not spread the tuple replicas and thus matches generally occur at the node where the tuple is generated. The chapter also described protocols that allow replication of tuples and at the same time allow synchronous operations to execute with global atomicity. The replication protocol is independent of tuple template location protocol. Replication is due to remote read accesses and degree of replication is independent of system size.

Finally, a space can be dynamically registered with any of the directories in the hierarchy. Each space is independent of other spaces in the sub-tree partition.

Chapter 5

Implementation

The previous chapters analyzed the issues related to distributed multiple tuplespaces. We discussed issues related to location, replication and consistency protocols for tuple replicas distributed over multiple spaces. We discussed our design model and presented arguments for its scalability and how it is an improvement over the centralized implementation or the current distributed tuplespace implementations. In this chapter we will describe the prototype implementation within the JavaSpaces framework.

5.1 Implementation Infrastructure

We have selected JavaSpaces for implementation as it offers a simple interface for performing basic tuplespace operations. In this chapter we begin by giving a brief introduction to the Jini infrastructure and the JavaSpaces service, which runs on the Jini platform.

5.1.1 RMI

JavaSpaces uses Remote Method Invocation (RMI) as a base protocol for client-server communication. RMI provides a higher-level abstraction to the socket based network communication. RMI allows a client to access methods of a remote object. In other words, with RMI, method calls and arguments are serialized to be transferred over a network. This call is deserialized and operated upon in a different Java Virtual Machine (JVM). The result of this call is serialized and passed back to the caller. The method definitions for the remote object are defined within an interface and the remote object implements this interface. This protocol requires some mechanism to marshal the remote calls and parameters, and get back with a returned value. The Java environment provides an RMI compiler that creates stubs and skeletons of the remote object, which perform

these tasks transparent to the application. For a more dynamic environment, these stubs and class files could be served by an HTTP server and could be downloaded on demand during remote execution. Figure 5.1 illustrates a client using RMI to access methods in a remote object.

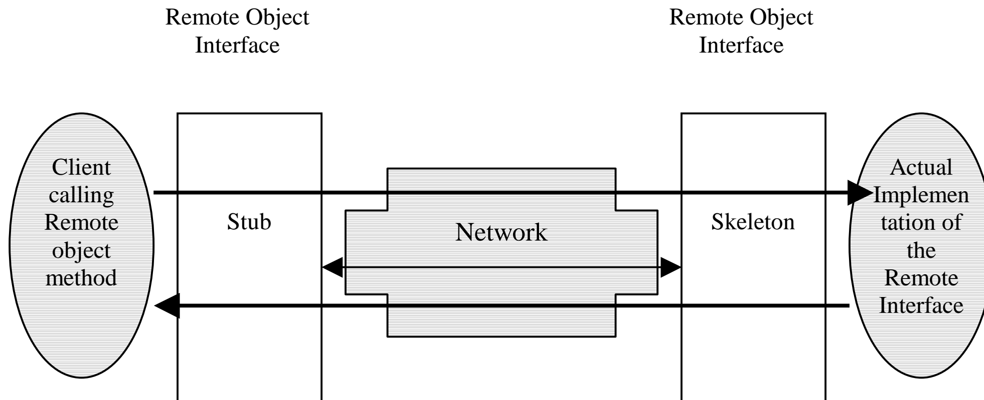


Figure 5.1: Remote method Invocation

5.1.2 Jini

Jini provides the necessary distributed infrastructure to allow a plug and play facility among various clients and services distributed across the network. The Jini solution is an answer to some basic questions in building a network computer with distributed services. The Jini layer conceptually spans the network and provides location transparency to clients and services. Being an extension to the Java application environment, Jini exploits the existing advantages of Java and is suitable for building distributed, platform-independent applications.

5.1.2.1 Jini Lookup Service

The first question is: how to locate services in a network? Central to the Jini concept is the lookup server [Sun99d]. This lookup server maintains registrations for different services that can be accessed over the network. This lookup server acts as a directory and provides the client with the client proxy of the service (called the service proxy) to which access is desired. The service proxy object executes in the client's JVM and gets the requested work done either by itself or by delegating it to the server over

RMI or other network protocol. The idea of downloadable service proxies is the key idea that gives Jini the ability to use services and devices without doing any explicit driver or software installation.

Other than acting as a registrar, the Jini lookup service is a mechanism through which the Jini network gets up and running. In addition, it manages a persistent database for storing service items, leases, and registrations, and assists in maintaining a self-healing network.

To deal with machine failures, Jini uses leases during service registration or lookup. The requesting service or client is responsible for renewing the lease in order to maintain registration. Using leases ensures automated cleanup of allocated resources following their expiration or a system failure. Jini also uses redundancy and transaction managers to deal with partial network failures.

The Jini lookup service handles synchronization issues between clients and the network services. It allows clients to register for a service with the lookup service, if a matching service is not currently available. This registration is for a specified duration and the lookup service sends an event to the client should the desired service register with it later on, during the lease period.

Figure 5.2 shows the timing diagram for a client searching for a service with the lookup server and eventually talking to the service.

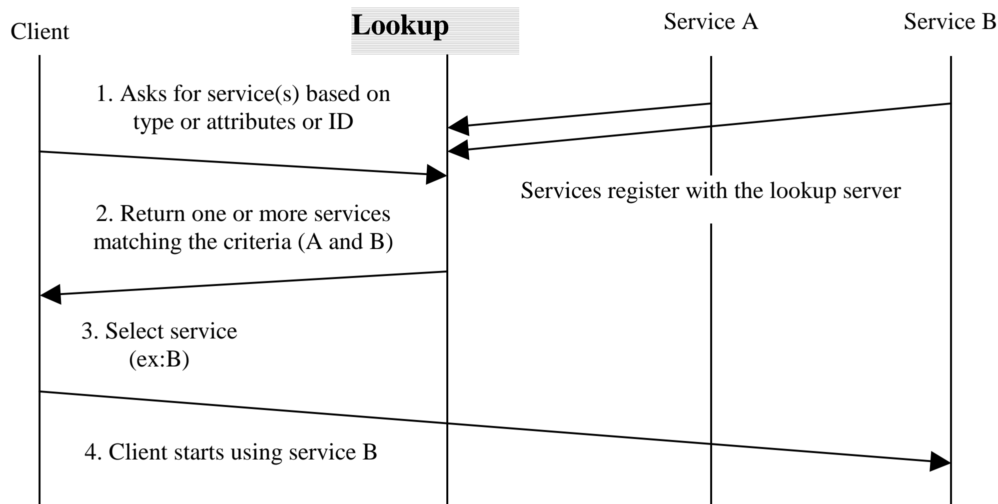


Figure 5.2: Locating services through a Jini lookup server

5.1.2.2 Discovery and Join Protocols

For new services joining the network, Jini follows a '*discover and join*' protocol [Sun99e] as shown in figure 5.3. The '*discover*' phase of the protocol comprises of the service sending UDP multicast request on the network. On receiving such a multicast request, the lookup server responds by sending its proxy through a unicast TCP protocol. The multicast request sent during discovery may result in multiple lookup servers being discovered for that network. Using the lookup service proxy, the service initiates '*join*', which is used to register the service with its attributes and ID and provides information about its client proxy. The join protocol is based on RMI.

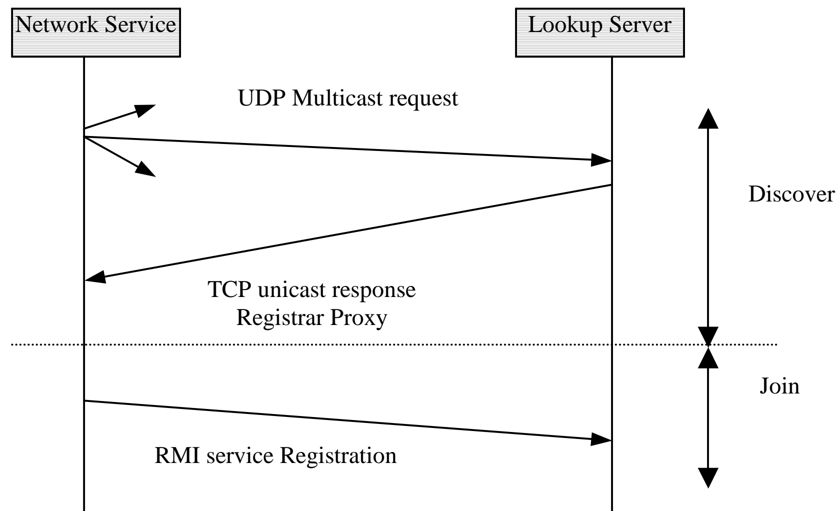


Figure 5.3: Discover and Join protocol

5.1.2.3 Reggie and RMID

Reggie is Sun's reference implementation of a lookup service and is built upon RMI infrastructure. It is an *activatable* process and therefore requires an RMI daemon. Activatable processes are remote server objects that start "on demand" as they are needed. The RMI deamon or *rmid* is the activation deamon that records the information required to restart activatable objects.

In order for RMI to function correctly, it needs some mechanism to provide the client with class files that may not be in the classpath of the client. In this context, HTTP server comes into picture. It provides the class files that are required to be dynamically loaded by the client, and the client can be informed of it by setting the

"java.rmi.codebase" property. This information is passed to the client through the client-proxy of the service (in the case of reggie, it is the Registrar) exported to the client.

Therefore, to set up the Jini environment, we start the HTTP server in the directory containing the jar files for different services. Then we start the RMI activation deamon (rmid). The lookup service (reggie) is started and the server object gets registered with the rmid.

Now when we start any other service, the reggie lookup service gets activated when it receives the multicast message from the 'discover and join' protocol. Now reggie takes over and sends its client proxy to the service which allows it (the service) to register with reggie. After service registration, reggie is again deactivated and remains that way till it is accessed again by another service registration request or a lookup request from a client.

5.1.3 JavaSpaces

JavaSpaces is a space-based service that runs over the Jini infrastructure. Jini allows a rendezvous between network clients and services, while JavaSpaces service allows different clients to coordinate and share entries. JavaSpaces architecture is designed to solve two main problems related to distributed systems: (1) *distributed persistence* and the (2) *design of distributed algorithms*. JavaSpaces implementations use RMI and the serialization feature of the Java programming language to accomplish these goals [Sun99b].

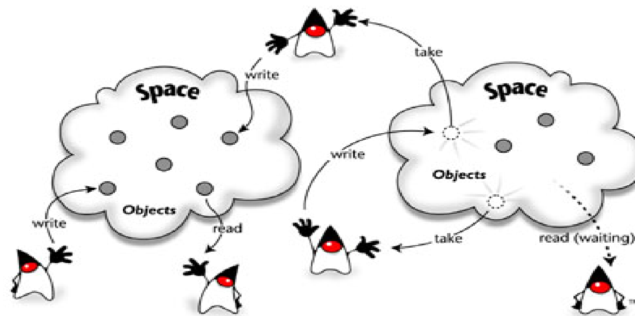


Figure 5.4: JavaSpaces

5.1.3.1 Overview

Every JavaSpace consists of entries, which are objects that implement the *Entry* interface. An entry is a group of serializable object references, which represent the fields in the Entry. Entries in JavaSpaces are equivalent to the tuples in Linda, and are accessed associatively. Each entry has one or more fields that can be used to match incoming requests from clients. In order for an entry in the JavaSpace to match, the entry must be of the same type as the template object. Each field in the template can have either a non-null value, which must match the fields in a matching entry in the JavaSpace, or a null value, which matches any value in that field.

Access to the entries in the JavaSpaces is through a small set of basic operations:

read (READ): Read an entry from the space that matches a template.

readIfExists (READP): Read an entry from the space that matches a template. Return *null* if entry not found.

write (OUT): Add an entry to the space.

take (IN): Read and remove an entry from the space.

takeIfExists (INP): Read and remove an entry from the space. Return *null* if entry not found.

notify: Register a template along with a reference to an event handler, with the space. When a matching entry is added to the space, the client is notified about it.

5.1.3.2 Other Features

Each of the entries or templates that are sent into the space has a time-out period associated with it. If a match is not found within the time-out period, the entry is dropped from the JavaSpace.

All operations on JavaSpaces are 'transactionaly secure,' which means that each operation or transaction is either committed or entirely non-committed to a JavaSpace. An operation on JavaSpaces can be either in the form of a simple operation, or a group of operations within a single Transaction. Transactions are maintained by a separate 'Transaction manager' service, which can be located over the Jini network.

5.2 The JavaSpaces Implementation

The presented tuple space model has been implemented using the JavaSpaces interface and by building on Outrigger, which is a contributed implementation from Sun Microsystems.

Outrigger implements the JavaSpaces specification in two parts. It has a client proxy class called *SpaceProxy* that is downloaded by the client from the look-up server. This proxy serializes the method calls and their parameters and sends it to the Outrigger server, which is implemented by the class *BasicSpace*.

5.2.1 UML Class Diagram

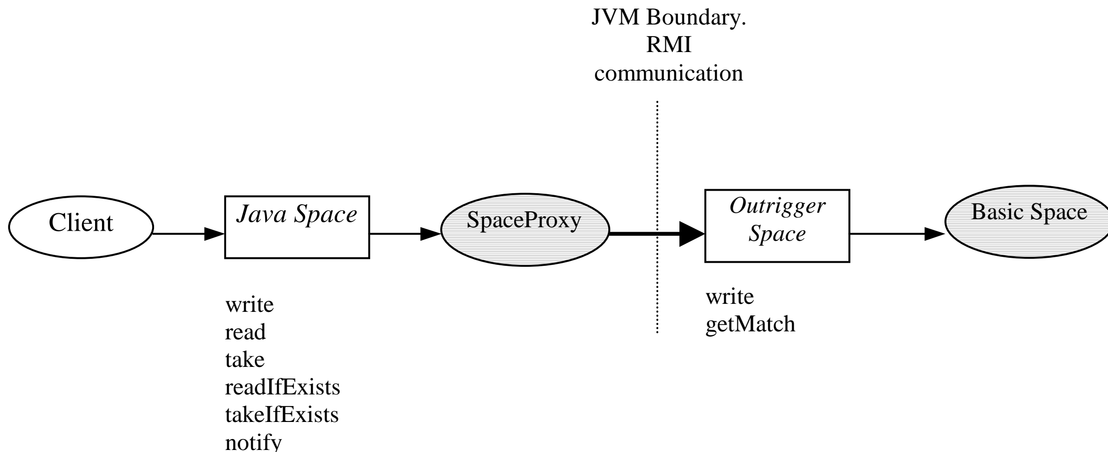


Figure 5.5: Outrigger UML diagram

5.2.2 Space Proxy: Client proxy for Outrigger

The *JavaSpace* interface defines the write, read, take, readIfExists, takeIfExists and notify primitives. The client proxy class *SpaceProxy* intercepts the primitives sent by the client process and provides an intermediate implementation for these calls. The entries in these calls are serialized and converted to objects of type *EntryRep*. The proxy then accesses remote methods of the Outrigger server through RMI.

Writes to the space server are sent through the *write* method, while the *getMatch* method is a common code in the *SpaceProxy* for all reads and takes. Blocking reads and

takes are implemented via events and a local listener object is instantiated for every such operation. For a blocking IN or READ operation, if a matching tuple is not found with the space server, the *getMatch* request registers the template and returns an *EventRegistration* object. The proxy then waits, renewing the registration if necessary, until either the requested lease time expires or it gets notified that a potential match was written. If the proxy is notified, it reattempts the getMatch. This is repeated until time expires. The proxy maintains the state of each blocking operation as *Answer* objects in a hashtable. The *Answer* object stores the ID of the operation as well as the *EventRegistration ID.*

We modify the *SpaceProxy* as follows: As shown in figure 5.6, we define an additional interface called *DirSpace* to access other spaces from the SpaceProxy class or the BasicSpace class.

After the *getMatch* code checks the local space for a matching tuple, if a match is not found the proxy probes the directory for a matching tag. If a match is found, it returns with the tag, which has a vector list of '*spaceNames*', that is, spaces that could possibly carry the tuple. The proxy then executes a read(take)IfExists operation on the other spaces.

If no match is found with the directory, the *getMatch* method with the proxy sends a *notify()* and registers interest with the directory for a matching tag. The *EventRegistration* object received from the directory is stored with the *Answer* object. The proxy then waits for the duration of the blocking call.

When a match is found either in the local space or the directory, a remote *notify* call is made on the listener object of the proxy. The proxy pops out the *Answer* object corresponding to the call from the hashtable, and matches the stored registration IDs with the event ID of the received remote object. The ID match decides whether the call is from local space or the directory. If the call is from the local space, the *getMatch* method probes the local space for the tuple. Else, the directory is probed for a matching tag. Again, the tag carries the '*spaceNames*', which are used to find the location of the tuple as well as its replicas.

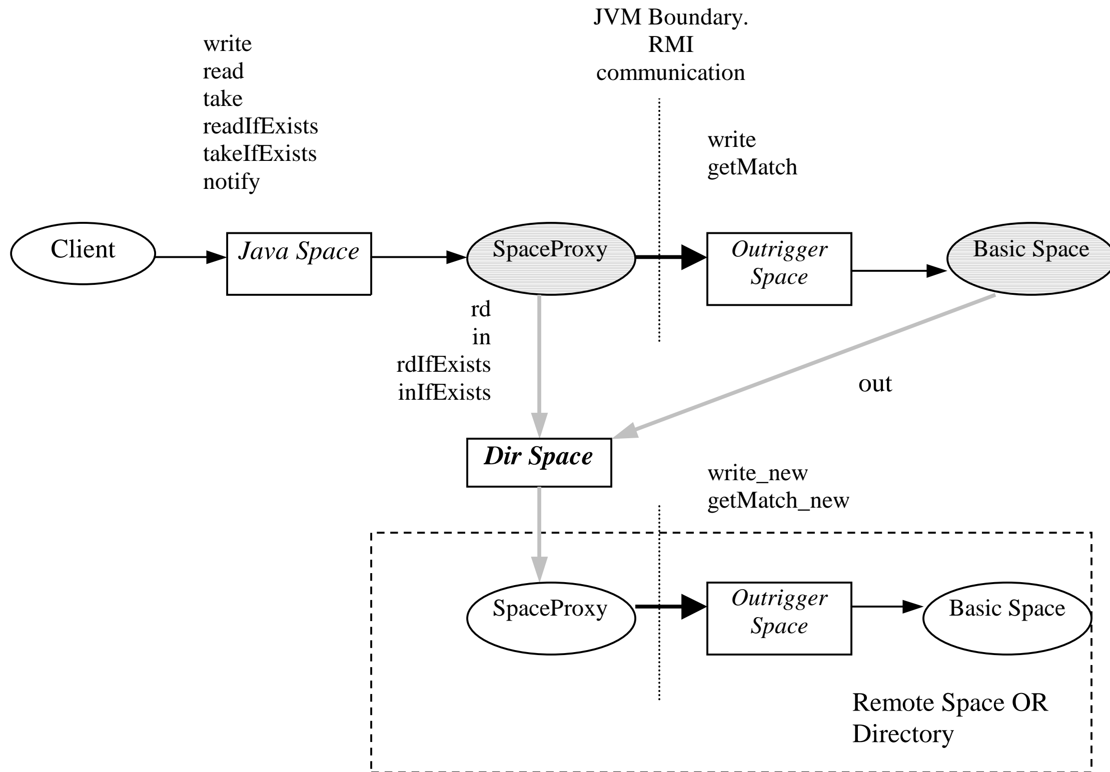


Figure 5.6: Modified Outrigger UML diagram

5.2.2.1 Consistency

An additional responsibility with the client proxy in our implementation is to maintain consistency across tuples and their replicas. Tuples stored with spaces are classified among *primary* and replica tuples. The client inserts a primary tuple with the local space server. This tuple then replicates on remote read accesses. Each tuple carries a vector list of space names with the first position reserved for the *primary* space for that tuple.

When a tuple is found in the local space, the client proxy of that space checks to see if the tuple is a *primary* tuple. If it is, then invalidates are sent to all the space servers whose names are stored in the vector list. Replicas are identified by their ID, which is same as the *primary* tuple's ID. If all the 'invalidates' are successful, the tuple is then sent to the client, otherwise the IN operation blocks.

If the tuple is a replica, then an IN operation is sent to the primary space for that tuple. Again, it is the responsibility of the primary to invalidate all other replicas. Upon successful completion, the local proxy passes on the tuple to the client.

5.2.3 Entries, Directory, Directory Tags

As defined in section 5.1.3, the JavaSpaces interface uses objects that implement the *Entry* interface to carry data from the client to the proxy of the space server. The client proxy converts Entry object fields to *MarshalledObjects,* and packages them as objects of type *EntryRep*, which are then sent to the space server in a serialized form. The *MarshalledObject* class is used as an envelope to ship objects around. Marshalling refers to the action of serializing one or more objects into a single byte stream. The stream may optionally contain a URL codebase annotation; this allows the remote server who will un-marshall the object to load the corresponding class file if it is not available in the local classpath.

EntryRep objects carry information about the class and the subclasses of the object. In our implementation, we have a vector that maintains a list of all the spaces where the tuple was originally inserted and then replicated. The EntryRep object also carries variables that maintain the state of the Object during various space operations.

As described in the theoretical model, we would like to propagate a tuple signature rather than an entire tuple to the directory hierarchy. The Outrigger implementation [Arn00] uses a 64-bit hash to represent each stored entry object. We suggest use of this 64-bit hash, to be propagated as a signature representation of the tuple. This hash is currently being used as a tuple handle within the space server. The first N fields of the entry are represented with the hash, where the maximum value of N is 16. For example, a 3-field entry would get a hash with 21 bits for each field, where a 16-field entry would have 4 bits per field in the hash. This 64-bit hash value can effectively and uniquely represent a tuple in the directory. The signature is used to indicate a potential match between the tuples and templates within the spaces. Therefore, template signatures carry along with them masks for formal fields, that is, fields that are used as placeholders to match with tuples. The signature also carries information about the class and the super-classes of the tuple.

5.2.4 Basic Space: Space Server of Outrigger

The space server for Outrigger is like an Object-oriented database, and it stores objects as serialized marshalled objects in the memory.

The Outrigger server can be configured for *transient* or *persistent* storage. Persistence provides the state of the server to be immune to data loss due to network or other failure. The *transient* space server stores the entries in the form of Data-structures and does not commit them to the object database. The *BasicSpace* class allows transient object storage. This class is extended and made persistent by logging all relevant operations in a log file. The back end ensures that the transient operations are logged and written through to the object database.

A type tree holds the known types of entries in the space (class TypeTree). In the type tree each type has a list of its known subclasses. This allows a template to search through the type-tree and match on its class as well as all its subclasses. For example: Consider an entry class called ‘Book’ with two subclasses ‘TextBook’ and ‘NoteBook’. Then, a template of type ‘Book’ can match on a tuple of all of the above three class types.

The contents of the space (marshalled entries) are stored as objects of the *SimpleEntryHolder* class, which holds all the entries of a given type as a doubly- linked list.

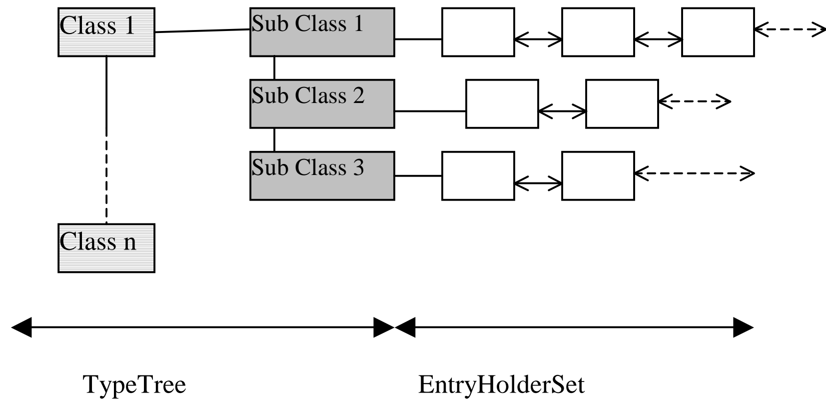


Figure 5.7: BasicSpace data structures

Each EntryRep object stored with the list is associated with an *EntryHandle*, which calculates a 64-bit hash value of the entry's contents. The hash is used as a quick-reject when scanning the list for a matching entry. The hash is computed from the hashCode of the entry's MarshalledObject fields, as explained in 5.2.3.

The Basic Space server is modified such that on a write method call, the EntryRep object is further propagated to the directory space. An optimization over this approach would be to buffer the out tuple for some time and propagate it only when it has not already been removed by a local IN operation. An approach would be to send such directory updates in batches rather than sending individual tuples. These optimizations are possible because the 'write' operation can be executed with relaxed consistency.

Also we identify remote READ operations on the Space and replicate the corresponding tuple to the requesting space. Another proposed optimization to exploit locality would be to replicate all entries of the same type that match with the requesting template based on a modified version of Black's [BAW89] competitive algorithm.

5.3 UML Design diagrams

When the block says "directory", it abstracts the entire directory hierarchy.

5.3.1 OUT

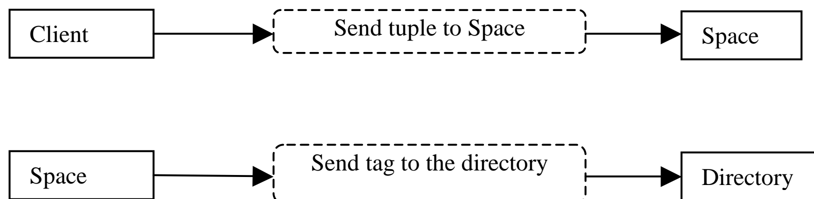
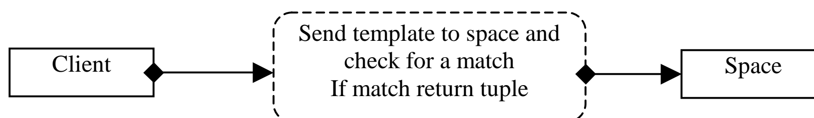


Figure 5.8: UML diagram for OUT

5.3.2 READ



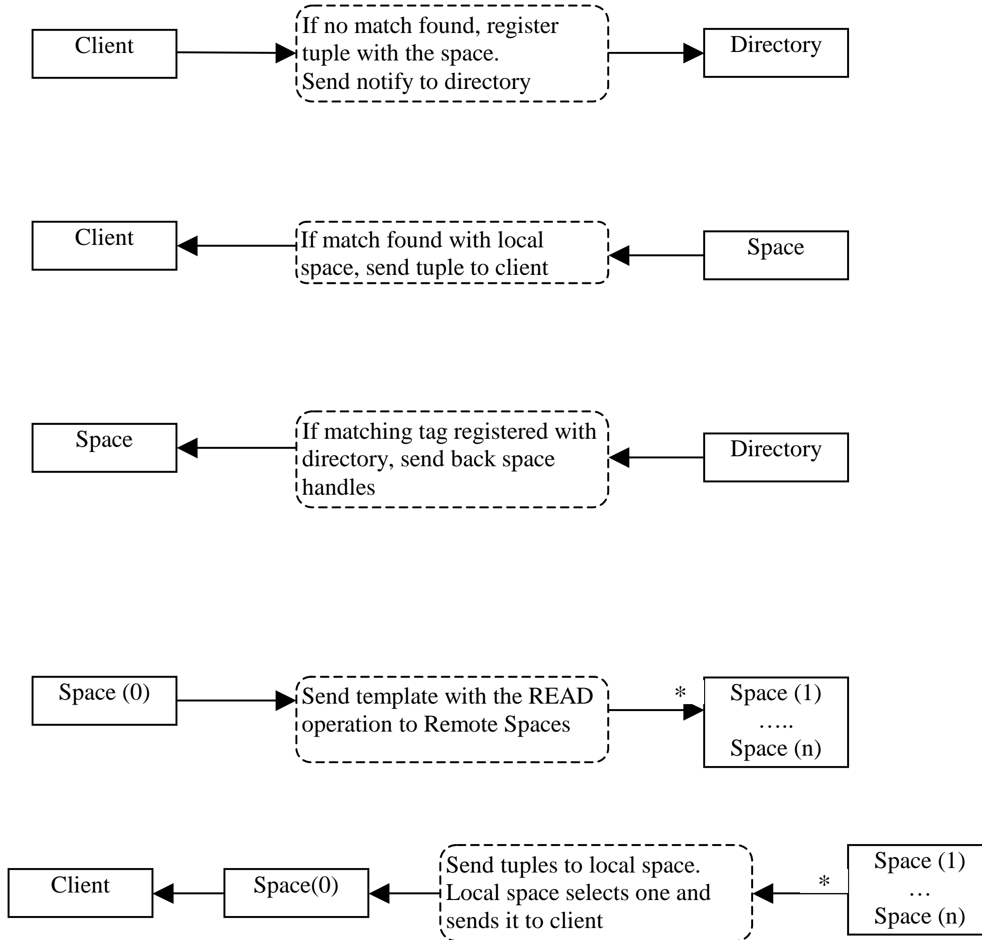
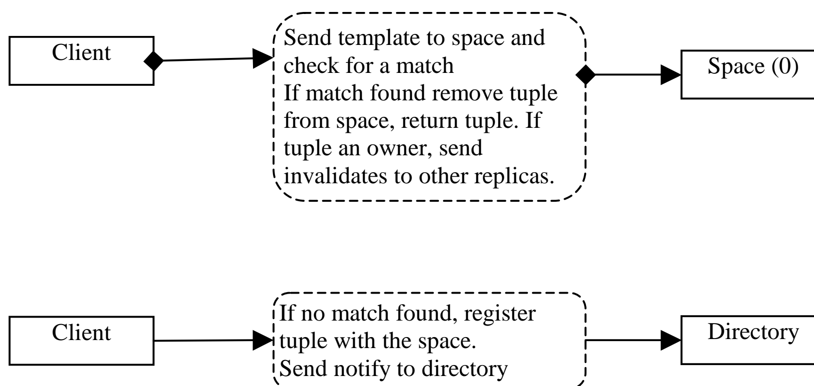


Figure 5.9: UML diagram for READ

5.3.3 IN



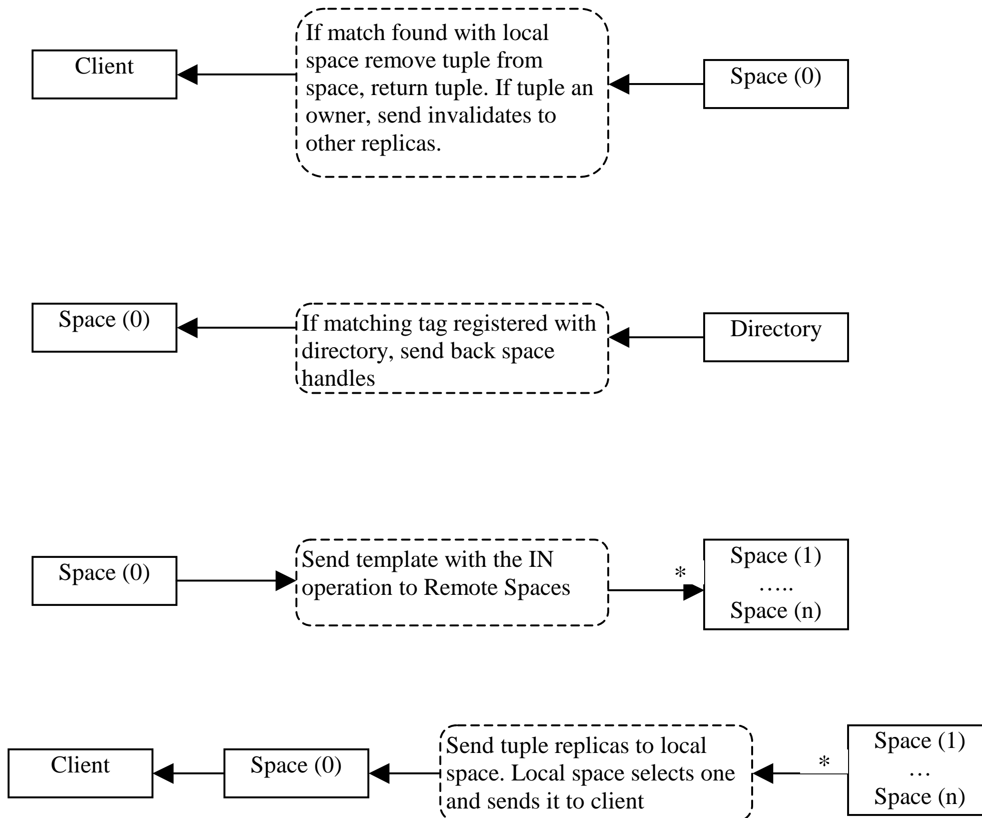


Figure 5.10: UML diagram for IN

5.4 Summary

This chapter described the implementation of a prototype of the directory based multiple tuple space model. We started with a brief overview of the Jini/JavaSpaces infrastructure over which the system is implemented. The chapter then describes the Outrigger implementation from Sun Microsystems and our code modification to build the functionality for communication among multiple space servers; and between the space server and the directory.

Chapter 6

Performance Evaluation

This chapter presents the results of the directory based prototype implementation. In order to measure the effectiveness of the implementation we examine latencies associated with each tuplespace operation and observe execution time for a few 'representative' tuplespace applications. The performance over the prototype is compared with results obtained over the JavaSpaces server.

6.1 Experimental setup

The experiments were performed on a cluster of eight Linux boxes (N0-N7). One of the Linux boxes (N1) was used specifically to run the servers essential to set up the Jini environment. These servers include the HTTP server that serves JAR files, the lookup server 'reggie' and the transaction manager 'mahalo'. The rest of the nodes of the cluster were used to run the space or directory servers, or the client processes.

6.2 Analysis of operation propagation time

write: Figure 6.1 shows the time required for the tuple to be propagated to its associated space. The average time for executing a 'write' operation is 11 ms. This time is comparable to the time required for a 'write' operation on the central JavaSpaces server.

read: Figure 6.2 shows the time required for the 'read' operation. This operation involves the template to be inserted, matched upon in the local server and the tuple to return. This is a local read operation and its latencies are comparable to the time required for a 'read' operation on the central JavaSpaces server.

If the tuple is not found, the request is sent to the directory. The directory informs of a possible match within a remote space. The template is then propagated to the remote

space and the tuple is read and replicated. This is termed as a remote read operation and it involves higher latencies. For a single directory hierarchy, the access latencies for a remote read are independent of the number of tuple spaces.

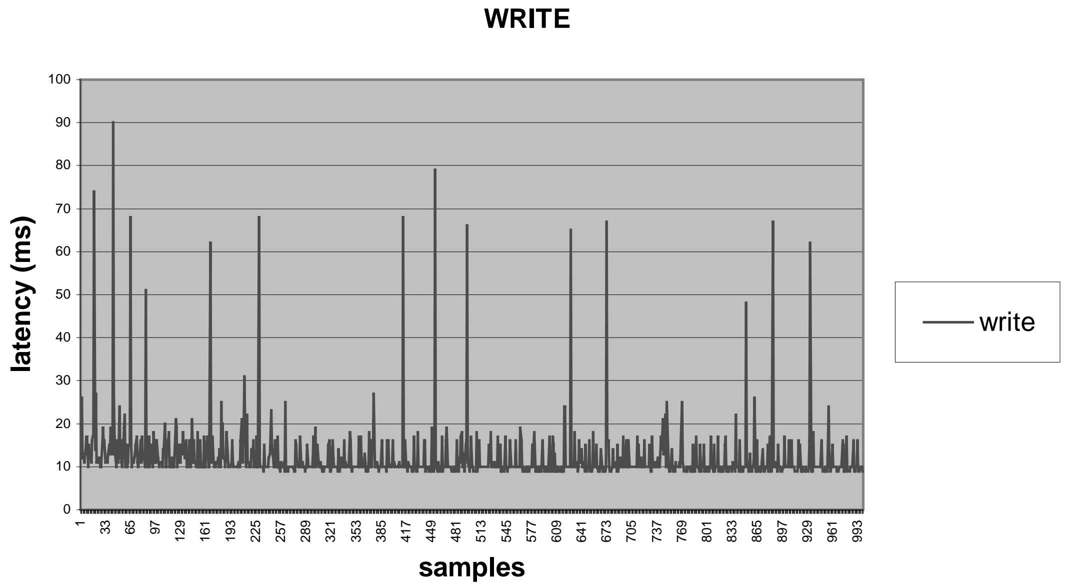


Figure 6.1: write operation latency

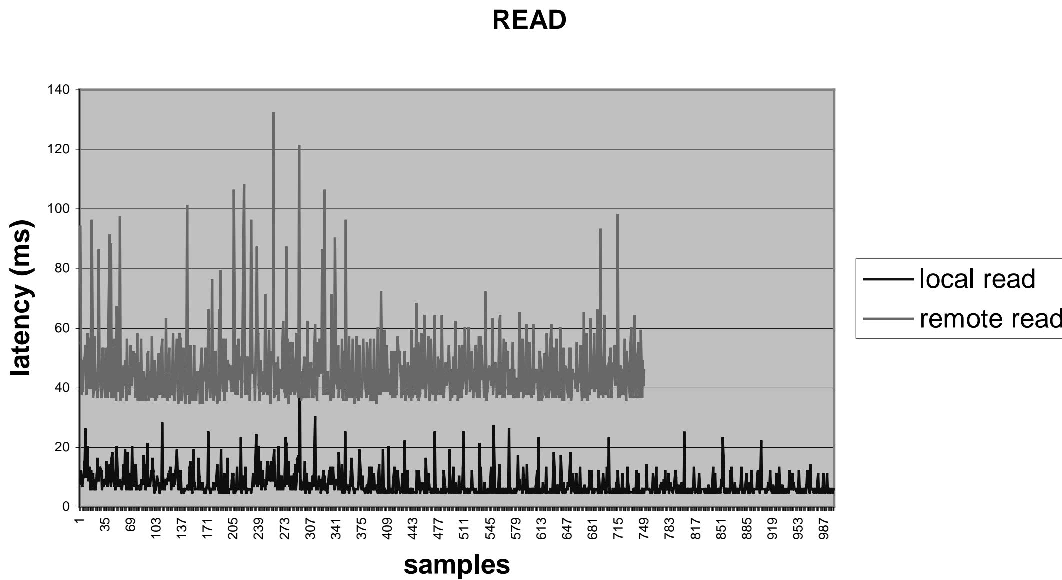


Figure 6.2: read operation latency

take: Figure 6.3 shows the time required for the 'take' operation. This operation is similar to the 'read,' except that the tuple has to be removed from space. In addition, if the tuple has replicas in other spaces, they have to be invalidated. Thus we have four scenarios, the first one of them being local take. The execution time for this operation is comparable to the time required for executing a take on a central JavaSpaces server.

If a matching tuple is not available in the local space, the directory has to be searched and the tuple is extracted from a remote space. This operation is referred to as remote take.

The other two scenarios involve invalidation of replicas along with a local or a remote take. Since we use the non-exclusive protocol along with primary replication, for a network with fixed latencies, the cost of remote 'take' operation does not increase for more than two replicas.

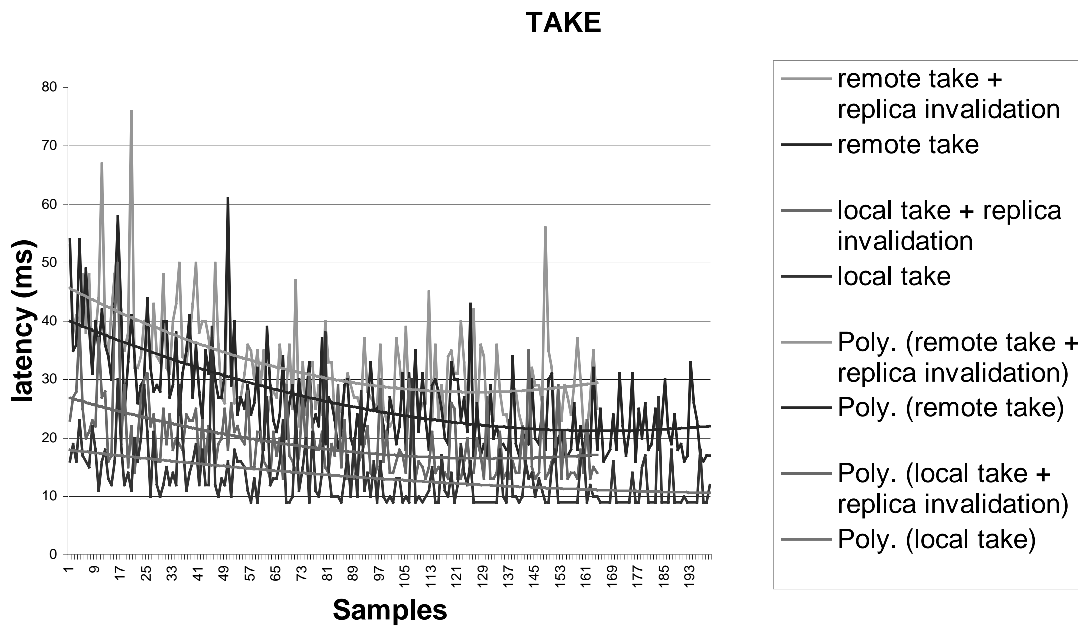


Figure 6.3: take operation latency

6.3 Analysis based on application performance

Successful execution of 'representative' applications over the directory-based model demonstrates that the semantics of the tuplespace operations is suitable from the application point of view.

6.3.1 Matrix multiplication

The matrix multiplication program works on a master worker pattern. Here we have one process that is responsible for distributing two matrices in the form of tuples of rows and columns. One or more worker processes are involved in computing the matrix product tuple, which is again dropped into the space for the master to pick up.

If A and B are the two matrices then the initialization process is as follows [ACG86]:

a) for(i=1; i<=n; i++)
    a. OUT("A", i, A's ith row)
b) for(j=1; i<=n; j++)
    a. OUT("B", j, B's jth row)
c) OUT ("Next", 1)

The cleanup process removes the individual product tuples as they are generated and forms the resultant Matrix.

d) for (count=1; count<= (NumRows *NumCols); count++)
    a. IN("Result", row?, col?, product ?)
    b. prod[row][col] = product

Each worker repeatedly decides on the element it has to process next.

Code1:

a) IN("Next", count?)
b) Count++
c) If(count <= (NumRows*NumCols)) OUT("Next", count)
d) r = (count –2)/ NumRow +1
e) c = (count –2)% NumRow +1
f) READ("A", r, Row?)
g) READ("B", c, Col?)
h) OUT("Result", r, c, product[Row, Col])

For the matrix multiplication application, we have a shared variable 'Next', which is key to its performance. The 'Next' tuple sets the order for processing distributed tasks. We distribute the matrix Rows and columns within the spaces and different workers probe the 'Next' tuple and decide the tasks, which they have to execute.

Tuplespace systems are not optimized for applications that require strict sequencing of the tasks. Instead of ordering the access to tuples by using a shared

variable, tuples should identify themselves for their resultant order. In applications where the use of a shared variable is necessary, an attempt should be made to transform their strict ordering requirements into coarse ordering. This would result in lesser access to the shared variable and thereby lesser delays in communication. We have modified the above algorithm, to read and modify the 'Next' variable per row, rather than per each element of the resultant matrix. The algorithm for the worker threads is modified as follows:

Code2:

```
a)  IN("Next", count?)
b)  count++
c)  If(count <= NumRows ) OUT("Next", count)
d)  r  = count
e)  READ("A", r, Row?)
f)  for (c=1; c<= NumCols; c++)
        a.  READ("B", c, Col?)
        b.  OUT("Result", r, c, product[Row, Col])
```

For executing the application over the distributed environment, the program was executed on three nodes of the cluster. Each node was connected to one space server. A single level directory, in turn coordinates the space servers. The configuration is shown in figure 6.4.

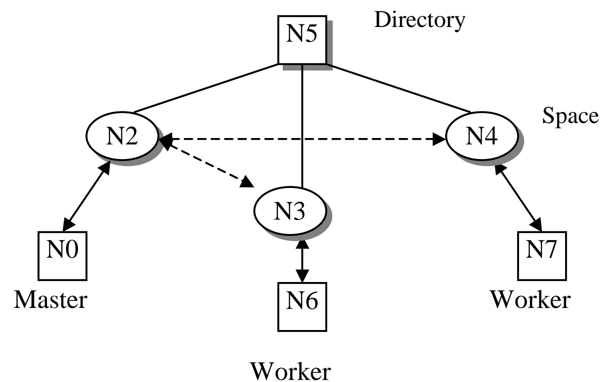


Figure 6.4: Configuration for Matrix Multiplication

The row and column tuples dumped by the master process are stored with N2 and registered with the directory N5. The workers probe spaces N3 and N4 respectively for the tuples. The directory informs N3/N4 about their location and the tuples are replicated to the associated nodes. This system would provide much better performance over a WAN, where the latencies in accessing tuples directly from N2 are much higher than the latencies in accessing tuples from the spaces associated with the Worker processes.

The results are as shown in figure 6.5

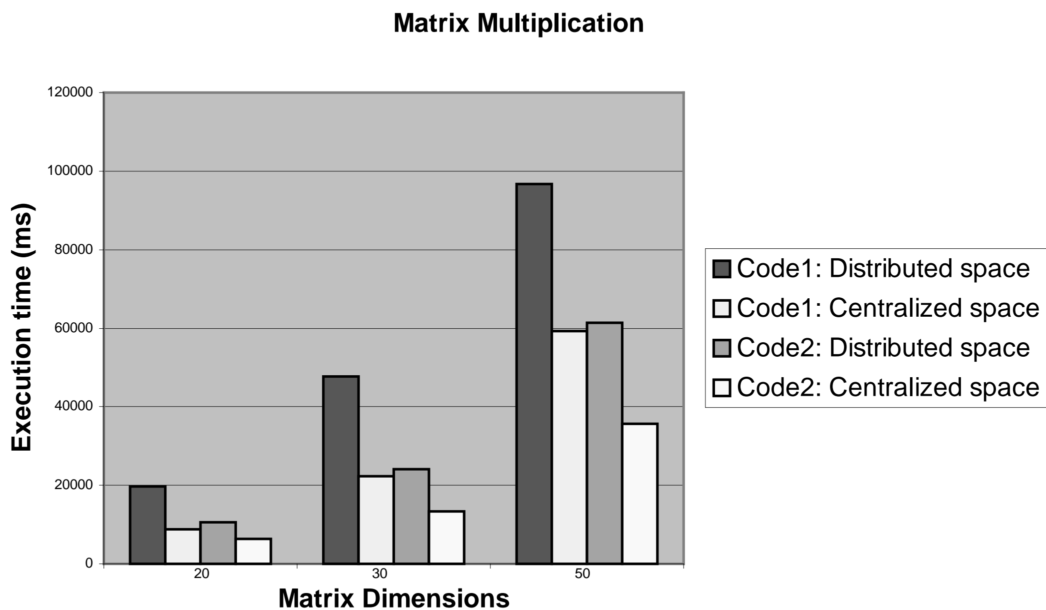


Figure 6.5: Matrix multiplication results

As we can see from the graph there is, on an average, 40% drop in execution time associated with the distributed system as compared to a 30% drop associated with the centralized system, when we compare the performance of code1 to code2. This is because of the reduction in the number of IN operations for the "Next" tuple. For the distributed environment most of the IN operations are remote INs, which adds to the latencies.

The execution time for the matrix computation is proportional to the number of tuples processed for a specific dimension of the matrix. The execution time for the distributed space implementation is 40% more than required on the central space server.

This is mainly because the execution times for remote READ and IN operation is much higher than a local READ or IN.

6.3.2 Mandelbrot fractal

The Mandelbrot fractal application calculates and displays the Mandelbrot Set fractal as shown in figure 6.6. The application is based on single master- replicated worker model. The master divides the display area into chunks and generates a task tuple for each chunk and inserts the task into the tuplespace. The workers remove these tuples, compute the mandelbrot function for that chunk and dump the result into the space. The master in turn picks up the result tuples when they are present and displays them on the screen.

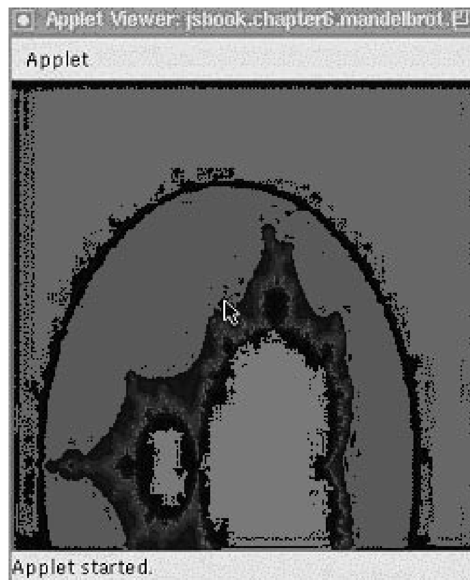


Figure 6.6: Screenshot of the Mandelbrot fractal

Figure 6.7 shows execution results based on computation of the Mandelbrot fractal.

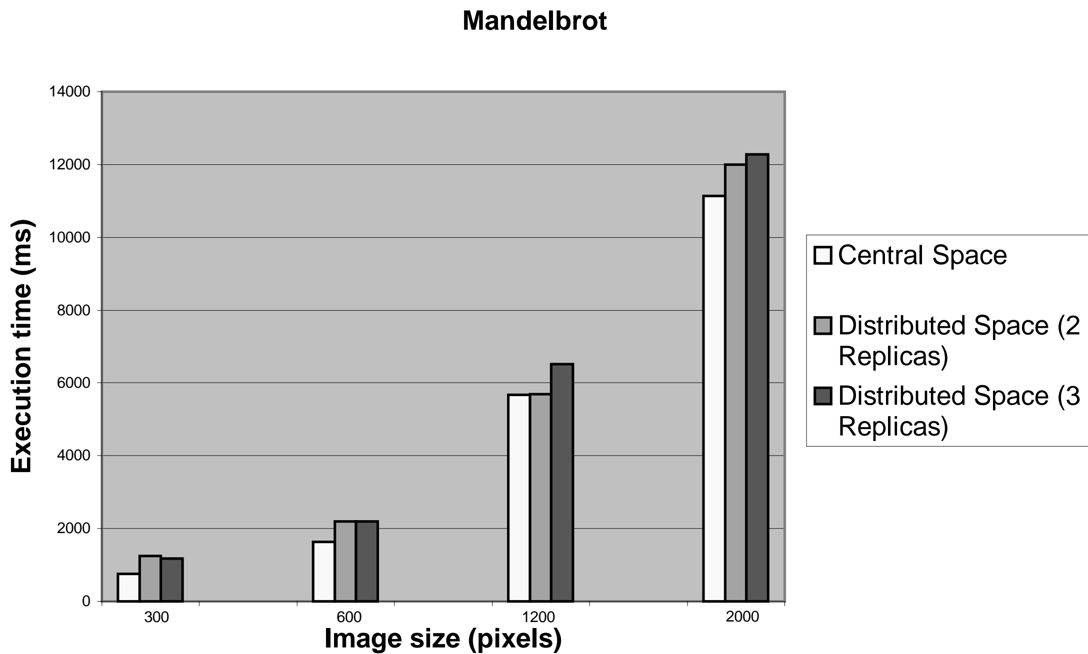


Figure 6.7: Mandelbrot fractal results

We can see from the figure 6.7 that as the image size increases, the execution period for the application increases. But we keep a constant number of tuples (20) even as we increase the image size. Therefore, as we see from the figure, the execution time difference between the distributed implementation and the centralized implementation is constant. This difference is mainly due to the higher average cost of operations of the distributed implementation.

In addition, there is not much execution time difference between the configuration with two tuplespace replicas and with three tuplespace replicas (we can generalize this to more than three replicas). Thus the execution time is independent of the number of replicas over which the tuples are distributed.

6.4 Summary

This chapter described the evaluation of the prototype of the directory based multiple tuplespace model. We showed that the latencies associated with the 'take' and the 'read' operations are higher as compared to the centralized implementation. But this is an expected tradeoff between scalability and performance.

Successful execution of ‘representative’ applications over the directory-based model demonstrates that the semantics of the tuplespace operations is suitable from the application point of view. As shown in this chapter, the system is designed to connect space servers located over a WAN and thereby provide low access latencies to parallel applications.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

This thesis studied tuple location and replication issues in a multiple tuplespace environment. The purpose of tuplespace distribution is to provide low access latencies for parallel applications executing on different nodes, spread over the Internet.

This thesis proposed a directory based tuple space model that allows various space servers to coordinate exchange of tuples and templates. This thesis also described protocols that allow replication of tuples and at the same time allow synchronous operations to execute with global atomicity. Tuplespace primitives are executed with varying degree of consistency, depending on the nature of the primitive. Using 'representative' applications the proposed semantics were evaluated. The evaluation found that the semantics of the directory-based model is suitable from the applications' point of view.

The most important contribution of this thesis is that the directory based model separates protocols that allow tuple and template matches over multiple spaces from the protocols that are used for tuple replication. We believe that tuples should not be replicated unless they are forced by the application requirements for reducing latencies of future accesses. The semantics of tuplespace operators allow us to make that distinction. Shared tuples, which have a high probability of change on every access are accessed using IN operations and are not replicated. Whereas, constants and other data tuples, which are not prone to frequent changes are accessed using the READ operation and hence get replicated over multiple spaces. Replica invalidations are carried out in parallel, therefore the cost of the IN operation would not increase linearly with an increasing degree of replication. In addition, the tree based tag propagation puts a definite bound for tuple/template searches. Thus, any tuple/ template, generated anywhere in the system of

associated space servers is made visible to its dual as soon as it gets registered with the directory hierarchy. The above contributions lead to a scalable solution as the tuples are partitioned based on their locality.

For our implementation, we have retained the JavaSpaces interface for the applications to access the space. So the current JavaSpaces applications can be easily ported, to be used on this multiple tuplespace platform. The application interacts with its associated space through the client proxy of the space server. The proxy has been modified to interact with the directory and remote spaces, and the distributed space accesses are transparent to the application process.

7.2 Future Work

The prototype implementation is a single level directory interfacing with multiple spaces. Future work would include extension of this prototype to implement the multi-level hierarchical directory structure as proposed in the design. Following this would be an optimization to dynamically register spaces and space servers with a lookup server, which would control the scope of the global tuplespace.

For the current prototype implementation, the primary space is statically allocated to the client. That is, the client does not make decisions about the cost of access or the load associated with a specific tuplespace. In future implementations, the client could be allowed to have dynamic access to spaces based on the feedback it gets from the lookup server.

Generally, the directory-based model could be optimized. For instance, the tag propagation due to an OUT could be buffered and sent after some finite delay. This is possible because, the OUT operation need not be visible immediately to all the spaces. Another optimization derived from the caching theory would be to replicate multiple tuples that match on a particular template, during a remote READ operation.

The design and implementation experience with the tuplespace paradigm within the Java/Jini technology framework shows that the Internet-computer has begun to evolve to the next level. This platform can be scaled across the Internet to facilitate development of distributed applications in heterogeneous environments.

List of References

[AB89] M. Arango and D. Berndt, 'Tsnet: A Linda implementation for networks of Unix-based computers,' *Technical Report YALE/DCS/TR-739,* Yale University, Department of Computer Science, September 1989.

[ACG86] S. Ahuja, N. Carriero, and D. Gelernter, 'Linda and friends,' *Computer*, Vol. 19, No. 8, pages 26-34, August 1986.

[AG96] S. Adve and K. Gharachorloo, 'Shared memory consistency models: a tutorial,' *Computer*, Pages 66 – 76, December 1996.

[Arn00] K. Arnold, '*About Outrigger Implementation.*' http://www.cdegroot.com/cgi-bin/jini?AboutOutriggerImplementation, 2000.

[BS95] D. Baken and R. Schlichting, 'Supporting fault-tolerant parallel programming in Linda,' *IEEE Transactions on Parallel and Distributed Systems,* 6(3), 1995.

[Bjo93] R. D. Bjornson, '*Linda on distributed memory multiprocessors*,' PhD thesis, Yale University, TR 931, 1993.

[Bal91] H. Bal, '*Programming Distributed Systems*,' Prentice Hall, 1991

[CG85] N. Carriero, and D. Gelernter, 'The S/Net's Linda Kernel,' *Proc. Symp. Operating System Principles,* Dec. 1985

[BAW89] D. Black, A. Gupta, and Wolf-Dietrich Weber, 'Competitive management of distributed shared memory,' *Proceeding COMPCON*, pages 184-190, 1989.

[CG90] N. Carriero, and D. Gelernter, '*How to write parallel programs*,' MIT press, 1990.

[CKM92] S. Chiba, K. Kato, and T. Masuda, 'Exploiting a weak consistency to implement distributed tuple space,' *Proceedings of the 12th International Conference on Distributed Computing Systems,* pages 416-423, IEEE Computer Society Press, June 1992

[CLZ97] A. Corradi, L. Leonardi and F. Zambonelli, 'Multiple tuple spaces onto massively parallel architectures: a hierarchical approach,' *5th EUROMICRO*

workshop on Parallel and Distributed Processing, London (UK), IEEE CS press, January 1997.

[DT99] F. Dahlgren and J. Torrellas, 'Cache-only memory architectures,' *IEEE Computer Magazine*, June 1999.

[FP96] J. Fenwick Jr., L. Pollock, 'Implementing an optimizing Linda compiler using SUIF,' *Proceeding of SUIF Compiler Workshop*, Stanford, California, January 1996.

[FP97] J. Fenwick Jr., L. Pollock, 'Issues and Experiences in Implementing a Distributed Tuplespace,' *Software – Practice and Experience*, Vol. 27(10), 1199-1232, October 1997.

[FBR93] S. Frank, H. Burkhardt III, and J. Rothnie, 'The KSR1:Bridging the gap between shared memory and MPPs,' *Proceedings COMPCON'93*, 1993

[GGH91] K. Gharachorloo, A. Gupta, and J. Hennessy, 'Performance evaluation of memory consistency models for shared-memory multiprocessors,' *Proceeding 4th International Conference Architectural Support for Programming Languages and Operating Systems,'* ACM press, New York, Pages 245-257, 1991.

[Gel85] D. Gelernter, 'Generative communication in Linda,' *ACM Transactions on Programming Languages and Systems*, 7(1), 80-112, January 1985.

[HLH92] E. Hagersten, A. Landin, and S. Haridi, 'DDM- A Cache only memory architecture,' *Computer,* September 1992.

[Kam91] S. Kambhatla, '*Replication issues for distributed and highly available Linda tuple space*,' Master's thesis, Oregon Graduate Institute of Science and Technology, February 1991.

[Kri91] V. Krishnaswamy, '*A language based architecture for parallel computing*,' PhD thesis, Yale University, 1991.

[LS99] J. Larsen, and J. H. Spring, '*GLOBE – Global Object Exchange – a distributed tuplespace enabling fault-tolerance and scalability in a heterogeneous, loosely coupled network.*' PhD thesis, University of Copenhagen, October 1999.

[Lei89] J. S. Leichter, '*Shared tuple memories, shared memories, busses and Lan's-Linda implementations across spectrum of connectivity*,' PhD thesis, Yale University, July 1989.

[Li88] Kai Li, 'IVY: A Shared Virtual Memory System for Parallel Computing,' *Proc. 1988 Int'l Conf. Parallel Processing,* Vol. II, 94-101, 1988.

[LH89] Kai Li, and P. Hudak, 'Memory coherence in shared virtual memory systems,' *ACM Transactions on Computer Systems*, Vol. 7, No. 4, Pages 321-359, November 1989.

[MSW96] H. Muller, P. Stallard, and D. Warren, 'Implementing the data diffusion machine using crossbar routers,' *Proceedings of the 10th International Parallel Processing Symposium, IPPS'96,* Honolulu, Hawaii, pages 152-158, April 1996.

[NL91] B. Nitzberg and V. Lo, 'Distributed shared memory: A survey of issues and algorithms,' *Computer*, 24(8), 52-60, August 1991

[Sun99a] Sun Microsystems. *Java Remote Method Invocation Specification*, 1999.

[Sun99b] Sun Microsystems. *JavaSpacs Specification Revision 1.0*, 1999.

[Sun99c] Sun Microsystems. *Jini Architecture Specification*, 1999

[Sun99d] Sun Microsystems. *Jini lookup Service Specification*, 1999

[Sun99e] Sun Microsystems. *Jini Discovery and Join Specification,* 1999

[WL98] P. Wyckoff, T.J. Lehman, et al., 'T Spaces,' *IBM Systems Journal*, 37(3): 454 - 474, 1998.

[XL89] A. Xu and B. Liskov, 'A design for a fault-tolerant, distributed implementation of linda,' *Proceedings of the ninth International Symposium on Fault Tolerant Computing,* IEEE, pages 199-206, 1989.