AALBORG UNIVERSITY

# Distributed Programming with Multiple Tuple Space Linda

by

Brian Nielsen
Tom Słrensen

June 24, 1994

INSTITUTE FOR ELECTRONIC SYSTEMS
DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE
Fredrik Bajers Vej 7E — DK 9220 Aalborg Ø — Denmark
Tel.: +45 98 15 85 22 — TELEX 69 790 aub dk

AALBORG UNIVERSITY
INSTITUTE FOR ELECTRONIC SYSTEMS

**DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE**

**TITLE:**
Distributed Programming with Multiple Tuple Space Linda

**PROJECT PERIOD:**
February 1, 1993 - June 31, 1993

**PROJECT GROUP:**
Brian Nielsen
Tom Słrensen

**SUPERVISOR:**
Keld Kondrup Jensen

Copies: 9
Pages: 121

ABSTRACT:
This thesis presents a novel model for distributed programming based on the coordination language Linda augmented with multiple tuple spaces (MTS-Linda). MTS-Linda offers a shared memory abstraction of the distributed system, and provides a powerful coordination model, integrating manipulation of distributed computations, data storage, modularity, and communication into the same handful of primitives.
We propose MTS-Linda as a model supporting sharing systems, i.e., distributed systems used by multiple interacting users sharing resources and data, and for communication. We give a comprehensive analysis of sharing systems to extract their requirements. Among the important requirements are strong means for coordination and data sharing, but notably full transparency is often undesirable.
We conclude that MTS-Linda fulfils the basic functionality of sharing systems, but the analysis also showed that MTS-Linda has shortcomings and that it, among other things, should be extended with a failure model, selective transparency, and persistent storage. Our recommended solution is through special tuple spaces which allows the programmer to adjust the properties of tuple spaces to his special needs. For example a tuple space may specified to reside on a given node sub set, to be persistent or replicated.
We have a prototype implementation of parts of the MTS-Linda model running on a collection of workstations. It works satisfactory, but its performance needs a boost.
We conclude that Multiple Tuple Space Linda is a good model for distributed programming.

# Preface

This Master's Thesis presents work done by the two authors during the 10th semester of the "Civilingeniłr i datateknik med speciale i datamatiske systemer" program at Aalborg University in Denmark. This project was carried out during the period February to June 1991, and was supervised by Keld Kondrup Jensen.

Since our 8th semester, our specialization within Computer Science/Computer Engineering has been directed towards distributed and parallel programming. At the 8th semester we wrote a so called synopsis describing our work at the 9th and 10th semester, and participated in further development of the AUC C++Linda system for a network of transputers. At the 9th semester the authors developed a prototype runtime system for a distributed programming model termed MTS-Linda. The work presented here is a continuation of this work, giving an evaluation of MTS-Linda in context of its application domain—sharing systems.

The report consists of 6 chapters and an appendix, and is structured as follows:

**Chapter 1.** *Introduction:* An introduction to distributed systems, and an presentation our motivation for dealing with MTS-Linda and sharing systems.

**Chapter 2.** *The MTS-Linda Model:* Presentation of an abstract multiple tuple space model. We introduce traditional Linda, and augment it with first class tuple spaces deriving the model that was subject for prototype implementation. Further, we augment the prototype MTS-Linda model with 1st class tuple spaces allowing for manipulation of behavior. Further, the developed prototype is briefly introduced.

**Chapter 3.** *Sharing Systems:* An analysis of sharing systems. This chapter describes sharing systems in detail by analyzing 3 examples of sharing systems: multi-programming environments, computer supported cooperative work, and distributed operating systems.

**Chapter 4.** *Supporting Sharing Systems:* A prescription of the requirements that should be fulfilled to support sharing. We compare these requirements to the functional abilities of MTS-Linda, to find the areas where MTS-Linda should be extended to fulfill the requirements.

**Chapter 5.** *Qualifying MTS-Linda:* The abstract MTS-Linda model presented in Chapter 2 needs to be twisted to accommodate the requirements of the real-word. In this Chapter we extend MTS-Linda with special tuple spaces which allow visibility of the distributed hardware, tuple spaces reflecting failures, and tuple spaces which are persistent. Furthermore, special tuple spaces allow sharing applications to pay only for the generality of MTS-Linda when it is needed.

**Chapter 6.** *Conclusions:* Conclusions and overall evaluation of MTS-Linda for sharing. It also includes suggestions for future work.

**Appendix A.** *Prototype Performance:* A benchmark of the developed prototype.

For additional background material we refer to the bibliography, which can be found at page 94. A Citation of the bibliography is written as a number enclosed by square brackets as e.g.,[60].

We are grateful to Keld Kondrup Jensen, and wish to thank him for supervising this project, for the many inspiring discussions about MTS-Linda and distributed systems in general, and for the constructive criticism he gave throughout the project period.

*Aalborg, June 9, 1993*

**Brian Nielsen** & **Tom Słrensen**

# Contents

# 1 Introduction

An economical way to build still larger and more powerful computing systems is to combine a number of smaller computers into a network of cooperating computers. Such distributed systems give a better price/performance ratio than mainframe based systems; mass produced VLSI electronics make high performance electronic devices relatively cheaper than higher speed devices produced in less numbers. Moreover, distributed systems have the potential of scalability. If an organization using a mainframe runs out of computing power, it requires a large investment to buy a larger one. However, it is a piece of cake to add a few more workstations to a distributed system! As Tanenbaum in [85] concludes: "a distributed system gives more bang for the buck".

There is little consensus about what the precise definition of a distributed system is, so we use the following general, but loose definition of a distributed system:

> *A distributed system consists of multiple autonomous computers that do not share primary memory.*

A consequence of this definition is that shared-memory multi-processor machines are not considered a distributed system in their own rights, though they may be part of one. The term autonomous means here that the computers have their own power supply, memory, a set of private I/O devices, and the possibility of partial failure. Dependent on the context we will throughout the report alternately use the terms node, computer, and machine to denote an autonomous processing unit.

## 1.1 Distributed Systems

We have identified 4 general classes of applications of distributed systems. The following list, which is inspired by [85] and [90], briefly summarize these:

**Parallelism:** The "speed of light" argument tells us that it is impossible to continue to increase the speed of the single CPUs. Because it is impossible for anything (including electrons) to move faster than the speed of light, physics puts limits to the speed of micro-electronics.

A promising work-around of this limitation is to make several computers cooperate simultaneously in solving a problem, i.e., let several processors compute in parallel.

Examples of applications which use distribution for speedup are large scale scientific numerical computations.

**Reliability:** Distributed systems have the potential of partial failure, whereas centralized systems are either fully operational or not operational at all. When a processor in a fault tolerant distributed system fails, the system should still be operational, possibly with a loss of performance, and a loss of the resources located at the failing processor. Increased availability may be obtained by replicating hardware and software components to several processors. A resource may in this way be available even when one of the replicates have failed.

Thus, distributed systems may be used for fault critical applications such as control of airplanes and nuclear power plants.

**Sharing:** Distributed systems can allow users to share expensive hardware devices such as color-laser printers and dedicated file- and compute-servers. Further, users with their own computers located at geographically separated places may be integrated into a distributed system. This allows users to share information, e.g. files and software, and to communicate over a network. Without a computer network sharing and communication would be inconvenient, and would be done manually by “snail-mail”, by telephone, or by exchanging disks.

Examples of systems which allows users to transparently access physically distributed resources and data are network operating systems and distributed operating systems.

**Inherently distribution:** The activities of an inherently distributed application takes place in distinct physical locations. When connected in a network the activities may coordinate jointly.

An example is a factory assembly line. Each place (perhaps a robot) in the assembly line is controlled by a computer. The computer coordinates the place’s activities with the neighboring computers, and perhaps with another computer monitoring the total state of the assembly line.

These application domains are not isolated islands, but rather, span a large continuum. Applications are not disjunctly defined to be in one of these categories, but may contain characteristics from more of the application domains.

Our application domain of MTS-Linda emphasizes sharing. We will use the term *sharing system* to denote a distributed system applied by multiple interacting persons to share resources and data, and to communicate[1].

If we take a look at the work carried out at our department’s network, we encounter students and researchers which are working together in groups, writing projects, articles, and software. Thus,

[1] A resource is a continuously existing immutable entity, like devices, which can only be used by one process at a time. Though the individual shared data items can be thought of as resources, we differentiate because data is mutable.

besides basic resource sharing in form of files and hardware, cooperation and information sharing, through, e.g., talk, email and news, is a daily life activity in a distributed system; assuming that our department is not the exception which confirms the rule.

We claim that supporting sharing and cooperation among multiple users are becoming increasingly important. To take an example from software development: Modern programs are large and complex, and their development requires participation of many programmers. The "lonely wolf" picture of a programmer who sits up at night coding his program is history!

## 1.2 Programming Distributed Systems is Difficult

However glorifying and promising distributed systems in principle may seem, their potential for better performance through parallelism, potential high reliability through redundancy, and abilities for resource sharing are hard-earned. Their added possibility for, e.g., partial failure, concurrent access, and non-determinism significantly increases the development effort. Furthermore, to speed up a sequential application by making it parallel often need rewriting and restructuring of existing code, and careful placement of program components onto processors is needed to optimize processor utilization, and to minimize communication. A parallel program needs to be designed very carefully, or it may easily end up actually running slower on a distributed system than its sequential version. Similarly, fault-tolerance is earned by using complex distributed algorithms and redundancy techniques to ensure consistency and consensus.

Further, the users of a distributed system may engage in a struggle for the resources, they were supposed to share cooperatively, and they may try to read information they are not supposed to. In addition, much research takes place within theoretical computer science to find formal specification and verification techniques to aid the correctness of distributed programs. These problems make distributed systems difficult to develop. The lack of global state information, the lack of global time reference, and nondeterminism due to internal arbitration are major contributors to this difficulty.

The above have led researchers to investigate new programming languages as well as new operating systems and other support facilities for distributed systems to ease the development of distributed systems. The client-server based Lynx [79], which gives the ability to dynamically setup client-server relationships, and for thorough error checking, the transaction based Argus [60], which goal is to make handling of partial error (node crashes) easy, and the object based Emerald [50], which seeks to simplify distributed programming through language support for distribution, are just a few languages of an army of hundreds dedicated to distributed programming. Distributed operating systems like Amoeba [85] and Chorus [78] managing the resources of an entire computer network, are also beginning to emerge. Also, existing operating systems have been augmented with high-level programming facilities such as remote procedure calls [11] and distributed process groups [8] to make the distributed system available to the average programmer.

Although, these systems has provided great advances, there is still plenty of room for improvement, and more needs to be done to mature the field. Distributed programming is not yet a daily life activity of the average programmer, and is thus often considered the business of hackers and gurus.

The work presented in this thesis should be seen as an attempt to add a new brick to the distributed-systems-puzzle. Hopefully, this and future research will, brick by brick, make distributed systems better understood and more widely used.

## 1.3 Programming with Distributed Data Structures is Easy

Programming languages for distributed systems has by[5] been classified as having either logically distributed address space, or logically shared address space. In distributed address space systems communication mostly takes the form of message passing or remote procedure calls. In shared address space systems on the other hand, communication takes place through a shared data store. Note that the distinction between logical distributed and shared does not imply that their implementation requires physical shared or distributed memory respectively. Logically shared memory implemented on distributed architectures are often referred to as distributed shared memory.

The advantages of shared memory over message passing is that programming is made easier. According to Stumm and Zhou[84]: “The primary advantage of distributed shared memory over data passing is the simpler abstraction provided to the application programmer, an abstraction the programmer already understands well.”.

The Linda[2] approach to distributed programming proposed by David Gelernter in 1985[31] provides a distributed data structure abstraction of a distributed system, and can thus be regarded as a shared memory system, though Linda differentiates between access to local and shared memory. Linda has successfully been implemented on both shared memory multiprocessors and on computer systems with disjunct physical memory, in which case it effectively gives distributed shared memory. The shared memory of Linda is called tuple space, and is accessible in pieces called tuples, where a tuple is a sequence of typed values. A tuple can be thought as a Pascal like record guarded as a whole by mutual exclusion. Linda is a coordination language. This means that it only deals with communication, synchronization, process creation, not with computation of expressions. These are calculated by the host language in which Linda is embedded. Linda will be introduced more thoroughly in Chapter 2.

Our programming experiences, gained through development of a distributed MTS-Linda run-time system, and a number of smaller distributed applications, tell us that development of parallel and distributed applications constitutes an iteration of three activities: First identify logical sequential pieces and their shared data. Then, develop each of the sequential pieces taking on a “single process view”—a sequential view. Finally, use a global view to ensure that processes interact so shared data is kept consistent, and that the program is live- and dead-lock free. Depending on the complexity of the program, may it require several iterations before correct behavior is achieved.

We have experienced that programming with distributed data structures is like designing protocols for accessing and updating abstract data types with the addition of concurrent operations.

[2]Linda is a registred trademark of Scientific Computing Associates, Inc.

The time and space decoupling properties, and the anonymous communication from Linda, and the conceptually shared memory, are major contributors to make this “top-down, one problem at a time” programming style possible. Jensen reports in[47] a similar experience:

> “This decoupling of the computational part of Linda processes from their coordination effectively simplifies the programming task: the programmer need not comprehend both at the same time, but may concentrate on each in turn in the iterative process of constructing the application.”

Linda has proved to be both a simple and an efficient way to develop parallel programs. It is our thesis that the Linda concepts of coordination and distributed shared memory can be used to simplify programming of **sharing systems**, and we shall deal with these in great detail in Chapter 3 and Chapter 4.

In summary, we seek a general, high-level abstraction of a distributed system which relieves the programmer from being concerned with the processors and wires of underlying distributed hardware. The user should partition his program into tasks, and the operating system (or runtime system) should map and schedule these, thus restoring the traditional distinctions between the programmer and the operating system,[28]. Furthermore, a distributed shared memory model gives simpler abstraction of communication between processes, than models based on data passing.

## 1.4 MTS-Linda

Attractive as it may seem, the globally shared memory of Linda is inadequate for sharing systems. Intuitively, it is clear that it is necessary to partition memory into segments belonging to different users, both to ensure privacy and comprehensibility; a user may not be willing to let other users see or modify his information. Even within large programs such modularity is a concern; it is common practice to partition a program into modules providing a set of operations to access a sub-set of the total memory used by the program. Moreover, Linda’s tuple contents based addressing system causes problems, as tuples intended to be used in one context can be mixed up with alike tuples from another context. Hence, we lack a way of partitioning or “sub-spacing” Linda’s lone tuple space into multiple tuple spaces. We will term Linda augmented with multiple tuple spaces for MTS-Linda.

The software of a distributed system is regarded as a dynamically evolving collection of agents. Each agent defines a region of concurrent behavior and shared state. An agent may span several nodes, be recursively defined, like agents may use the functionality of each other in the makeup of distributed ensembles. In MTS-Linda terms such an agent is a tuple space. A tuple space enclose shared state (data) and concurrent behavior (processes) which logically is considered a whole. A tuple space may thus encompass a program, a collection of related services, a module, or may merely be a group of (shared) data. Tuple spaces are the universal grouping mechanism in MTS-Linda programs.

A difference between MTS-Linda and client-server based models is that it is possible to expose a state, and to share this state among a set of cooperating processes. This is opposed to encapsulating the state in server-processes. This turns out to be a flexible programming model, however, in certain ways this flexibility may also be non-restrictive. Furthermore, a tuple space defines a virtual centralized entity; though the entity may be physically distributed, it can still be manipulated atomically a whole.

MTS-Linda provides for time and space decoupled coordination. Modules located in physical or logical disperse address spaces are able to communicate and synchronize (coordination in space). Because tuple space is effectively a storage, modules separated in time can also communicate and synchronize (coordination in time). Tuple spaces can be used for storage of both persistent and temporary shared data, using the same set of primitives, thus providing for a unified memory model.

Furthermore, MTS-Linda allows the programmer to compose, and to manipulate modules of concurrent activity and related shared state as first class objects, thus obliterating much of the traditional distinction between process and data, between the active and the passive; the same primitives manipulates both.

In summary, MTS-Linda offers a set of features important for the implementation of distributed applications: a unified memory model, communication, modularity, and manipulation of concurrent modules. All of that, integrated into one simple coordination model with only a handful of primitives seems utterly attractive. It is simple, powerful, flexible, and fits with our conception of programming a distributed system!

We have through our literature studies noticed a shift in application domains of Linda: From being a parallel programming language, [20], Linda through its extensions with multiple tuple spaces has become envisioned as an operating system model[44], [32], or as a programming environment [21]. We regard MTS-Linda as a coordination language. Both operating systems, programming languages, and application level interactive languages provide a coordination language, and the MTS-Linda model may be manifest as either of these. We will deal with MTS-Linda as an abstract model focusing on the operating system or distributed programming language level. In addition, Linda has actually been used by Cogent Inc. as foundation for an operating system for a multiprocessor computer [24]. In contrast, our work reported in this thesis deals with MTS-Linda in the context of general multi-user distributed sharing systems.

## 1.5 Our Thesis

Before starting this project we gained some experience with MTS-Linda. At the s9d semester[3] we implemented a prototype runtime system for a part of the full MTS-Linda model proposed by Jensen [48]. The purpose of the prototype was to find implementation techniques and to analyse the synchronization and communication patterns needed to manipulate tuple spaces. Thus, the work took place purely from a implementation point of view without much concern about the application domain.

We had a faint idea that MTS-Linda would be used by multiple users who could deposit tuples

[3]The second to last semester in the MS program in computer science at Aalborg University.

in each others' home tuple spaces to communicate, but that was about all. We needed an attitude towards the application domain—an analysis—and a set of overall requirements for a MTS-Linda programming system. We lacked knowledge about how applications was to use MTS-Linda i.e., had only little understanding for what purpose, and how tuple spaces would be used.

This report addresses the applicability of MTS-Linda; just because it is implementable it may not necessarily a good idea. The purpose of this report has been to analyse and provide answers to some of the above raised questions. It gives an analysis of the intended application domain, and describes how MTS-Linda can accommodate the requirements of the applications. The result of this work has lead us to the following thesis:

> MTS-Linda is a good model for distributed programming.

Here, distributed programming means development of applications for distributed systems i.e., the development of a set of processes execution on multiple computers providing multiple users the possibilities for sharing resources and information, and communication. We here focus on testing whether MTS-Linda provides the necessary primitives for distributed programming.

To be a good model for distributed programming MTS-Linda, or any other model, must fulfill the following general requirements:

**Simplicity.** It must be easy to learn and use, and make it easy for programmers to develop a great variety of distributed programs—a flexible, high-level abstraction of a distributed system is desired.

**Suitability.** It must be suitable for the problems at hand. It must fulfil the requirements of sharing systems, and provide the programmers with the means to solve the problems in programming sharing systems.

**Efficiency.** Efficiency is always a concern in computer systems. People are interested in getting fast and predictable response times, to maximize hardware utilization, and in getting their job done as quickly as possible. It must be possible to for programmers to design efficient programs, as the system in general cannot guarantee this.

Given our thesis and the 3 means for measuring it, we start by outlining the MTS-Linda model to give needed insight into the MTS-Linda and its components. Next, we analyze the application domain of distributed sharing systems, through 3 examples, to find requirements which MTS-Linda must fulfill to be a good model for distributed programming. We confront MTS-Linda with these requirements, and show how some of them are nicely expressed using MTS-Linda while others are not. For requirements which are not satisfied by the MTS-Linda model we present number of solutions, mainly based on introducing tuple spaces with special characteristics depending on what requirement they seek to fulfill. Last, we confront the solutions with the 3 initial requirement: simplicity, suitability, and efficiency, an draw our final conclusions.

# 2 The MTS-Linda Model

This chapter introduces the Linda paradigm, and its argumentation with first class tuple spaces and first class processes. Multiple tuple spaces have been envisioned since the first descriptions of Linda [33]. The development of the MTS-Linda model is still under way. The last five years have seen several multiple tuple space proposals, these include Ciancarini[21] and Cogent Research [24] with a flat tuple space structure and no manipulation of tuple space as a whole, Gelernter[33] and Hupfer [44] nested tuple spaces with path names for access and the ability to manipulates tuple spaces as a whole, and latest Jensen[48] which allows for both flat and nested tuple spaces and introduction of theses as first class objects.

The ultimate MTS-Linda model is yet to emerge. There is little agreement upon the specification of tuple space and the operations allowed on and with it. The diversities are for example found in the ability to manipulate tuple space, topology of tuple spaces; flat versus hierarchical, and tuple space naming. The only agreed upon part is that the operations found in Linda must be preserved.

We are working with a model giving first class tuple spaces and processes. We have designed and partly implemented a prototype MTS-Linda system at our 9th semester. This prototype is based on the MTS-Linda model under development by Jensen[48]. To allow the prototype design and implementation to be realized in one semester it was restricted in a number of areas, most notably are first class process objects not part of the design. The remainder of this chapter gives insight into Linda and our crippled prototype MTS-Linda model. Further, we present the essential performance measurements for our implementation, allowing us to give performance estimates of a non-prototype implementation. Last, the MTS-Linda model by Jensen, which we use in the rest of the report is outlined, to adding the features omitted in the prototype.

## 2.1 Linda

Linda is the predecessor of MTS-Linda. Linda is what have been termed a *coordination language* [34]. They made the observation that programming languages in general consists of two orthogonal components: a traditional sequential programming language component—a computation language—to express how active pieces *compute*, and a coordination language to describes how active pieces *coordinate*. Loosely speaking, coordination deals with process creation, and synchronization and communication among processes. When Linda is embedded into a computation language the outcome is a language for parallel programming. Linda have been embedded in several computation languages: in imperative languages as C[24], C++ [74], and Modula2 [16], in a object oriented language SmallTalk [67], and in functional languages, e.g., Standard ML [80]. A computation language used to host Linda is termed a *host-language*. The Linda and MTS-Linda examples in this report uses C++[83]as host-language.

Linda provides the notion of a *tuple space* as the medium for inter process coordination. Tuple space is a conceptual shared memory realized as a multi-set of *tuples*. A tuple is the unit of communication, and constitutes an ordered sequence of fields—each made out of a type, and possibly a value of the given type.

We use the term process to denote a single thread of control. Each Linda process have access to tuple space and can attempt to coordinate by inserting tuples into, and retrieving tuples from tuple space. Linda provides 4 simple primitives for manipulating tuples; **out**, **in**, **rd**, and **eval**. **out** and **eval** inserts a tuple into tuple space, while **in** and **rd**, respectively removes and reads (copies) a tuple from tuple space.

```
1.   int main() {
2.        out("Test",42);                    // Insert a tuple into tuple space
3.        int i;                             // Declare a local integer variable
4.        in("Test",?i);                     // Remove the tuple from tuple space

5.        process int f(int x) { return g(x); }// Declare f() to be a process
6.        eval(42,f(137));                   // Insert an active tuple
7.        rd(42,?i);                         // Read the resulting passive tuple
8.   }
```

Figure 2.1: A sample Linda program using all 4 Linda primitives. The host language is C++, and the ’?’ indicates a formal field.

The Linda **out** operation inserts a tuple into tuple space, based on a template given as argument. A template describes the contents of a tuple in host-language terms. In Figure 2.1 line 2, a tuple with two fields is inserted into tuple space. The template for the **out** operation is $(\text{"}\texttt{Test}\text{"}, 42)$, which, when inserted into tuple space becomes a tuple where the first field is of type string with value `Test` and the second field of type integer with value 42. An **out** operation is non-blocking leaving the tuple in tuple space independent of the creator process. Figure 2.2 illustrates the contents of tuple space after each of the Linda operations in Figure 2.1 is performed.

The Linda **in** operation retrieves a tuple from tuple space. Tuples are addressed associatively, i.e., by contents rather than by explicit sender and receiver names. The address system is based on the notion of tuple *match*. Two tuples match if they have the same number of fields, and if corresponding fields match pairwise. Fields are divided into two categories based on their contents. An *actual field* contains a type and a value, e.g., $(integer : 7)$. In a template this field can be either a literal, $7$, or a host-language variable of type integer and with value 7. A *formal field* contains only a type, e.g., $(integer :?)$, the $?$ indicates that there is no value. In a template a formal field is denoted by a $?$ followed by a variable of type integer, e.g, $(?i)$. Fields can match actual-to-actual and actual-to-formal, but not formal-to-formal. Match actual-to-actual requires that both types and values are equal, e.g., $(integer : 7)$ and $(integer : 7)$ match while $(integer : 7)$ and $(double : 7.0)$ does not match. Match actual-to-formal requires that the fields are of the same type, thus ignoring the value of the actual field, e.g., $(integer : 7)$ and $(integer :?)$ match, while $(integer : 7)$ and $(double :?)$ does not match.

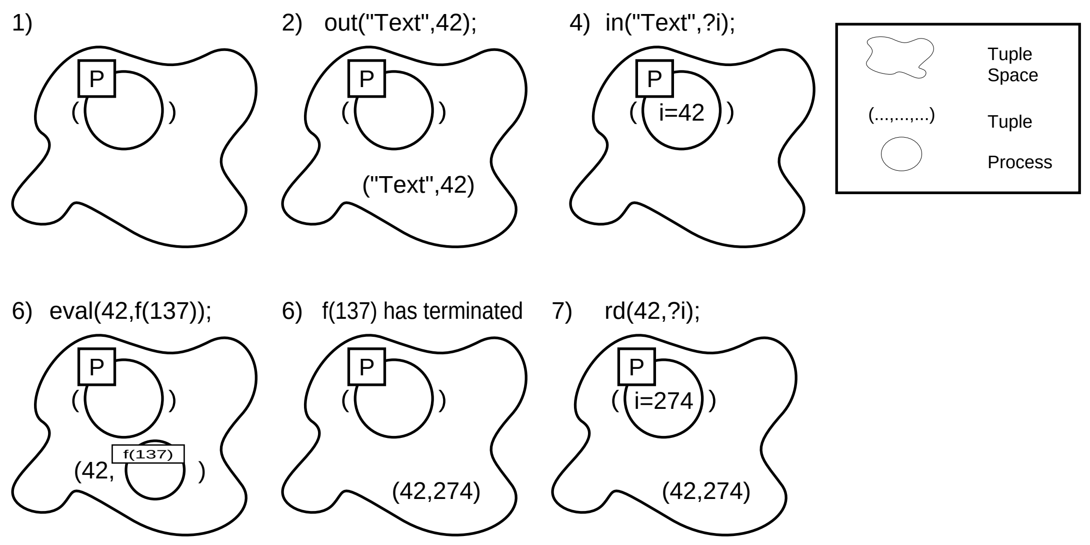

Figure 2.2: The effects on tuple space when performing the Linda operations of Figure 2.1. Each drawing show the contents of tuple space just after the associated operation. The x) numbers refers to a line in the program.

The **in** operation takes a template as argument, turns it into a tuple, and uses it to search tuple space for a match. The process performing the operation is blocked until a matching tuple is found. If a match cannot be found the process will block until a matching tuple is inserted into tuple space. A match results in removing a tuple from tuple space, and returning it to the process performing the **in**. A formal field in the template is associated with a host-language variable, which is assigned the value of the actual field from the matched tuple. Figure 2.1 line 4, shows an **in** operation, trying to remove a tuple corresponding to the one put into tuple space by the previous **out** operation. The in template has as first field an actual string matching the tuple from the **out** operation, the second field of the template is a formal integer field, indicated by the '?' and the integer host-language variable $i$, which matches any integer. Upon match, the host-language variable $i$ will be assigned the value of the matching tuples field. In the example $i$ will be assigned the value $42$. The Linda **rd** operation is equivalent to **in**, except that the matched tuple is left in tuple space while returning a copy to the process performing the **rd** operation.

The last Linda operation, **eval**, inserts a tuple into tuple space similarly to **out**. The difference is that the fields of an eval tuple are evaluated concurrently and asynchronously with the creator process, i.e., each field represent a new process. A tuple is termed an active tuple if there are fields containing a process, and a passive tuple if all fields are passive data values. A process is a value yielding computation which eventually returns a value. When all processes of an active tuple have yielded their value, the tuple is converted into a passive tuple which can be matched through an **in** or **rd** operation. The AUC-Linda implementation[74] uses a qualifier, `process`, to denote C++ functions which are evaluated concurrently with the creator when used in an **eval** operation. Figure 2.1 line 6, shows an eval operation where two fields in principle are evaluated concurrently. But, only the C++ function $f()$ is qualified as a process and will thus be evaluated concurrently with the main process.

The Linda coordination model allows processes located in disparate space and time to coordinate. The conceptual shared tuple space memory allows processes to deposit tuples and then terminate. At some later time at new process may be started and retrieve those tuples disregarding that the two processes never existed at the same time. Linda thus provides space and time decoupled communication.

## 2.2 The Prototype MTS-Linda Model

Multiple tuple spaces is a logical extension of the Linda coordination language. It allows conceptual entities to be grouped, and allow for manipulation of these as wholes. Tuple space allows for grouping of both data and activity in form of passive and active tuples.

The *philosophy* behind MTS-Linda is that, from a process's point of view, a tuple space is a data structure that implements a multi-set of tuples. The Linda operations are abstract operators which add, copy, and remove tuples form the multi-set. This observation leads to the logical extension of Linda with multiple tuple spaces, and allow processes to instantiate tuple space as a multi-set datatype. This in effect introduces tuple space as a first class data type in host-language and MTS-Linda.

The prototype design and implementation is based on a MTS-Linda model featuring two kinds of tuple spaces: Local tuple spaces to contain local data and processes, allowing for coordination in a hierarchal manner and shared tuple spaces to contain data and processes, allowing for coordination across hierarchies. Further, the model introduces tuple space as first class objects in both the host language and in MTS-Linda, allowing them to be bound to host-language variables and communicated as part of tuples.

### 2.2.1 Local tuple space

A local tuple space is conceptually a local data structure within a process. Local computations are placed in local tuple spaces. A Local tuple space is introduced in a host-language through the $TupleSpace$ type. Figure 2.3 line 2, shows the creation of a local tuple space bound to the host-language variable $myts$. A local tuple space obeys the scoping rule of the host language, i.e., is the local tuple space and its contents of tuples is deleted when the $myts$ variable in the example leaves scope, this happens in line 12 of Figure 2.3 (any line references in the following are directed to Figure 2.3).

```
1.   main() {
2.     TupleSpace myts;             // Create a local tuple space
3.     myts.out("Test",42);         // Insert a tuple into local tuple space
4.     myts.rd("Test",?i);          // Read the tuple from local tuple space

5.     eval(42,f(137));             // Insert an active tuple in context tuple space
6.     rd(42,?i);                   // Read the resulting passive tuple
7.     out("Ts",myts);              // Put a copy of myts into the context tuple space
8.     in("Ts",?myts);              // Get a tuple space and bind it to myts
9.     TupleSpace A,B;              // Create two new local tuple spaces
10.    A=B;                         // Assign a copy of tuple space B to A
11.    if(A == B){...}              // Compare tuple space A and B.
12.  }                              // myts leaves scope and is thus deleted
```

Figure 2.3: A sample MTS-Linda program showing allocation and operations on and with local tuple spaces.

The Linda operations; **in**, **rd**, **out**, and **eval** are used in the traditional way on a local tuple space, only they must be prefixed with a local tuple space variable, denoting the destination of the operation. Line 3 and 4, shows an **out** and **rd** operation performed on the local tuple space $myts$.

Tuple space operations which are not explicitly target at a local tuple space, are sent to the *context tuple space*, defined as the tuple space where the process performing the operation resides as a field of an active tuple. Line 5 and 6, show the usage on an **eval** and **rd** operation on the context tuple space. Figure 2.4 illustrates the processes and tuple spaces contents just after the execution of the **eval** operation in line 5.

A local tuple space is a first class object in the host language and in MTS-Linda. This implies that it is possible to compare tuple spaces, to copy tuple spaces through assignment, and to pass tuple space as argument to a MTS-Linda operation.

The MTS-Linda operations gives pass-by-value semantics. A tuple space used in a template to a MTS-Linda operation is copied, and the copy becomes part of the tuple in tuple space. Line

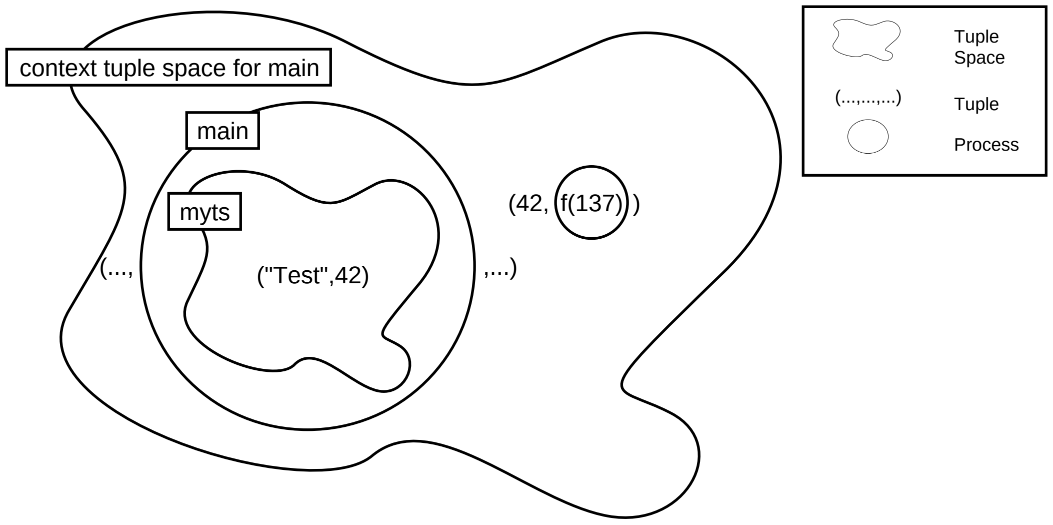

Figure 2.4: The $main$ process and its context and local tuple space just after execution of the **eval** operation in line 5, of Figure 2.3.

7 shows the local tuple space $myts$ being put into the context tuple space, resulting in a tuple containing a copy on the $myts$ tuple space, called $myts'$. Figure 2.5 shows the context tuple space just after execution of line 7. A tuple space stored as field of an tuple is for convenience termed a field-tuple space. The **in** operation in line 8 attempts to retrieve the tuple, containing a tuple space, from the context tuple space, to do so it must determine if the $myts$ and $myts'$ tuple spaces match.

Match of two tuple spaces is based on the equality of two multi-sets of tuples, intuitively defined as: Removing pairwise equal tuples from the two tuple spaces until no more tuples can be removed. If all tuples are removed from both tuple spaces they are equal, otherwise they are not. The **in** operation in line 8, is satisfied as removing pairwise equal tuples form tuple space $myts$ and $myts'$ results in emptying both.

Assignment of one local tuple space to another, shown in line 10, implies making a copy of the source tuple space tuple ($B$) and bind this copy to the destination tuple space variable ($A$), overwriting the previous contents of the destination tuple space. The effect of an assignment is two tuple spaces with the identical contents.

Comparing 2 tuple spaces is the same as determining if they match. Line 11 shows comparison of 2 tuple spaces $A$ and $B$. Comparison evaluates to true if the tuple spaces match and to false if they do not. The comparison in line 11 evaluates to true as tuple space $A$ is a copy of $B$.

When used as argument to a MTS-Linda operation, or in a tuple space comparison and assignment operation. A local tuple space can contain active processes. Consequently it must be possible to

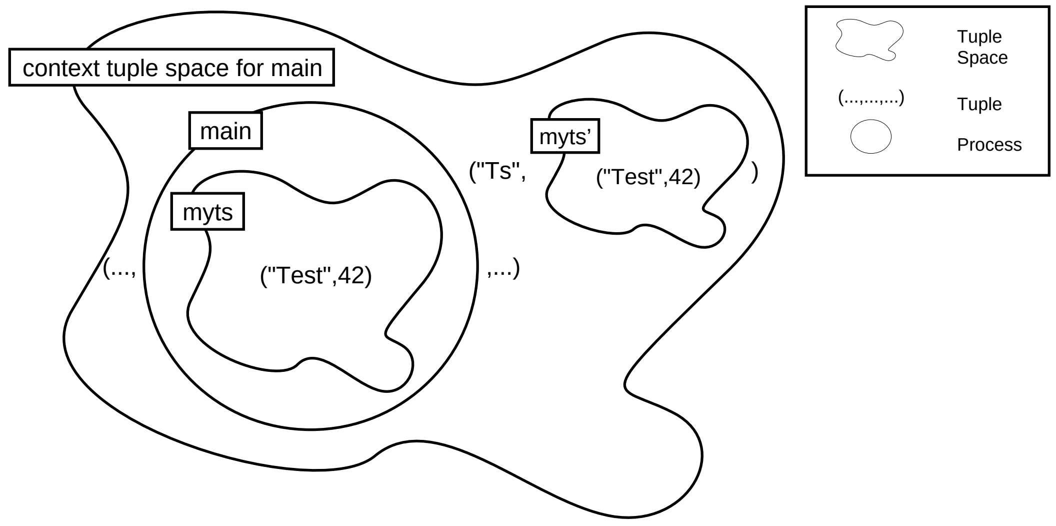

Figure 2.5: The **out** operation in line 7, of Figure 2.3, results in a copy of the tuple space $myts$ being inserted into the context tuple space.

compare and copy active tuples, and thus to compare and copy processes. The prototype design and implementation have been restricted not to include the introduction of processes as first class objects thus not allowing processes to be manipulated. Section 2.4 outlines the implications of having processes as first class objects, allowing for suspension, copying, moving, and removing of processes.

The prototype MTS-Linda model does not allow for manipulation of active processes. A process is defined as a *future*, i.e., a value that is to be[40]. An MTS-Linda process will eventually[1] return a value and thus show future behavior. However, being unable to manipulate an active process we can only manipulate the value it eventually will return, i.e. its future. The future semantics propagates to tuples and tuple spaces, thus only allowing us to manipulate passive tuples and passive tuple spaces. The future of an active tuple is defined as a passive tuple. Similar an active tuple space, a tuple space which contain one or more active tuples, becomes passive when it contains only passive tuples.

A Tuple space may switch between passive and active state depending on the operation performed on it. The important observation is that the model only allows manipulation of a tuple space if it is passive. Any operation performed with one or more active tuple spaces as arguments will be delayed until all these tuple spaces have become passive. This in effect implies that MTS-Linda operations, with a tuple space as parameter, are delayed until all tuple spaces passed in the template yields their values. As a consequence, field-tuple spaces are always passive.

[1]Well infinite loops are possible, but they must be programming errors ;-)

### 2.2.2 Shared tuple space

Local tuple spaces define a strict bureaucratic hierarchy of computations and coordination. A process can only coordinate directly with the processes on the level “below” and the level “above”, i.e., through the context tuple space or through local tuple spaces. Reconfigurations can be made by copying and deleting sub-hierarchies. Copying, or deleting, a local tuple space implies recursively copying, respectively deleting, any nested tuple spaces.

This strict hierarchal coordination patterns defined by local tuple spaces is not always appropriate to describe the coordination taking place between individual processes. For processes located in different part of the tuple space hierarchy to coordinate, tuples must be propagated through the hierarchy, involving participation of other processes. It is desirable that processes can coordinate across hierarchal boundaries. What we need is a shared tuple space, not bound within the local tuple space hierarchy.

A shared tuple space is a freely evolving tuple space where the creator does not have any spatial or temporal relation to the tuple space (they can be compared to a file) except that the process of cause must exist before it can be created. Shared tuple spaces represents an organic coordination model. This model is flat, there are no relations among tuple space, except they may contain references for each other. This is opposite the local tuple space enforcing a bureaucratic coordination model.

For a process to access a shared tuple space, the process must possess a reference to the tuple space. A reference to a shared tuple space is realized through a tuple space capability ($TSCap$), which intuitively can be seen as a traditional pointer. The tuple space capability is introduced as a type in the both host-language and MTS-Linda. A variable of $TSCap$ type allows a process to perform MTS-Linda operations on the shared tuple space denoted by the capability.

```
1.  TSCap shared_ts;                  // Declare a tuple space capability
2.  shared_ts = newts();              // Create a new shared tuple space

3.  shared_ts.out("Test",42);         // Place a tuple in the shared ts
4.  int i;
5.  shared_ts.in("Test",?i);          // Remove a tuple from the shared ts

6.  out("A_TSCap", shared_ts);        // Put a capability into the context ts
```

Figure 2.6: Creation and usage of a shared tuple space (ts is a shorthand for tuple space).

A new shared tuple space is created by the $newts()$ operation. Figure 2.6 line 1 shows a declaration of a host-language variable capable of holding a capability to a shared tuple space. When declared the $shared\_ts$ variable does not name any tuple space. The $newts()$ operation in line 2 create a new shared tuple space and binds its capability to the $shared\_ts$ variable. The

$shared\_ts$ variable follows the scope-rules of the host-language.

Deleting a variable of $TSCap$ type will, however, not delete the shared tuple space, but only the capability. A shared tuple space is conceptually ever lasting, but will in practice be garbage collected when no capability naming it exists. Figure 2.7 illustrates two processes $P_1$ and $P_2$ which each possess a capability naming the same shared tuple space. This allows the two processes to coordinate directly via tuple space $B$ despite they are themselves located in different tuple spaces.

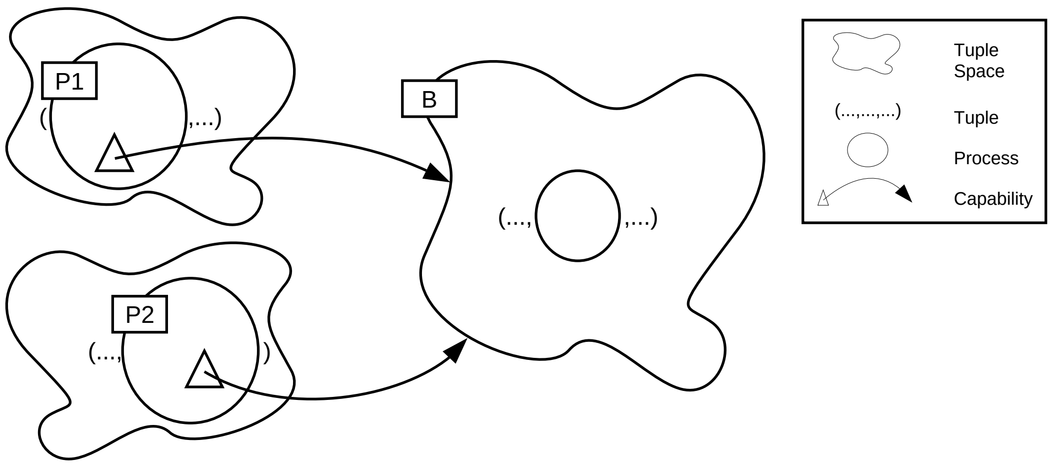

Figure 2.7: Two processes $P_1$ and $P_2$ each have a capability for the shared tuple space $B$.

By prefixing the Linda operations, **in**, **rd**, **out**, and **eval** with a $TSCap$ variables are they performed in the referenced shared tuple space. Line 3 and 5 show an **out** and **in** operation performed on the shared tuple space referenced by the $shared\_ts$ variable.

Shared tuple spaces are, as already mentioned, named by a tuple space capability which is a first class object in the host language and in MTS-Linda. This implies, as for the local tuple space, that capabilities can be compared, assign, and communicated as part of tuples. The tuple space capability has roughly the same properties as a C++ pointer. Copying a capability, results in two capabilities naming the same object. There is not future semantics on a capability, so a copy operation is not delayed until the shared tuple space becomes passive. Comparison is also based entirely on the capability and evaluates to true if the two capabilities name the same shared tuple space. Passing a tuple space capability as part of a tuple allows processes outside the shared tuple space to pick-up the capability and gain access to the shared tuple space. The **out** operation in line 6 places a copy of the $shared\_ts$ capability in its context tuple space, thus allowing other processes to retrieve the capability and access the shared tuple space.

However, the analog between capabilities and pointers does not go all the way. It is not possible to dereference a shared tuple space to a local tuple space, or use an address-of operation on a local tuple space to yield a tuple space capability.

### 2.2.3 Copy and Move of Tuple Space Contents

A local tuple space can be manipulated as a whole, a shared tuple space cannot. To remove this irregularity we must be able to manipulate a share tuple space as a whole. MTS-Linda provides two *multi-set* operations, **contents_copy** and **contents_move**, which respectively copies or moves the contents, in form of all tuples, from one tuple space (source) into another (destination).

The multi-set operations are operations on a shared or a local tuple space, just as the Linda operations. They do, however, take a shared or a local tuple space as argument not a template. The multi-set operations are overloaded. It is possible to, e.g., move the contents of both shared and local tuple spaces into a local tuple space. Making it possible to mix the types of the source and destination tuple spaces.

Due to the inability to manipulated active tuples, a multi-set operation is delayed until the source tuple space becomes passive. This is the case even when the source tuple space is represented by a capability for a shared tuple space!!! The destination tuple space can be passive or active, as the tuples from the source are appended to the destination.

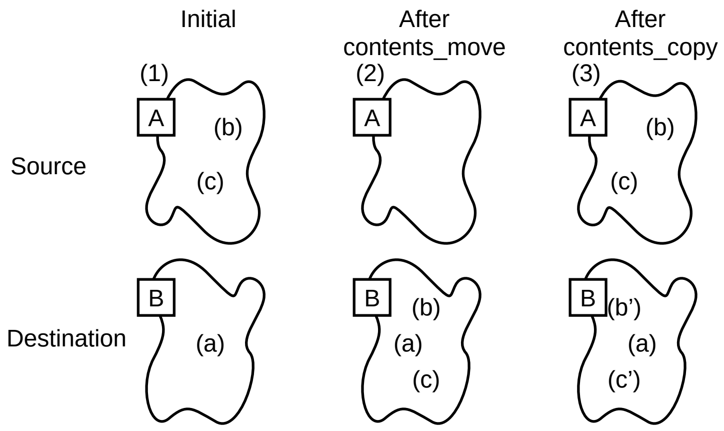

Figure 2.8: The effects of the multi-set operations. (1) the initial state of source tuple space $A$ and destination tuple space $B$. (2) contents of $A$ and $B$ after a $B.contents\_move(A)$ on the initial contents. (3) contents $A$ and $B$ after a $B.contents\_copy(A)$ on the initial contents.

Figure 2.8 illustrate how the contents of the source and destination tuple space is changed when the multi-set operations are used. Part (1) shows the initial state of source tuple space $A$ and destination tuple space $B$. Part (2) shows the contents of tuple space $A$ and $B$ after a $B.contents\_move(A)$, where the contents of tuple space $A$ is moved to tuple space $B$. Similarly part(3) shows the result of a $B.contents\_copy(A)$, the $b'$ and $c'$ tuples are indistinguishable copies of the $b$ and $c$ tuples from the initial $A$ tuple space.

Usage of a multi-set operation may result in a process trying to manipulate itself. If a process resides in a shared tuple space and it is in possession of a capability $C$ for the tuple space, then performing a multi-set operation with tuple space as source would block infinitely waiting for the process itself to yield its future, which will not happen. To avoid such programming errors, it is a general rule that processes may not manipulate themselves trying to do so is an error, i.e., the operation will fail.

## 2.3 Prototype Implementation

At our 9th semester we designed and partly implemented a run-time system for the MTS-Linda model just described in Section 2.2. We started our 10th semester by "finishing" the implementation giving a usable system. This included writing a new socket based back-end, allowing us to run the system on a distributed network, instead of the single node system used to realize the first running system. Approximately 1 month of the project period was used for finishing and distribution of our solution, some error-fixing, and on carrying out a set of performance tests.

### 2.3.1 Implementation Status

We have implemented the prototype MTS-Linda model and running with a few restrictions: Garbage collection of shared tuple spaces is not implemented, and tuple spaces cannot be given as process arguments, and cannot be used among the fields of **eval**-tuples. Most notably, the tuple space comparison operator is not implemented.

The prototype amounts to approximately 21.000 lines of C++ source code. The use of C++ as implementation language allow us to map the object-oriented design directly to a object-oriented programming language. Thus, the classes in the design become C++ classes in the implementation. For a detail description of the design and implementation, refer to an earlier report, [71].

The prototype is running on a network of SUN[2] sparc workstations. It consists of a set of nuclei; a nucleus is running on each machine participating in the system. A limitation of the prototype is that the set of machines is statically configured, i.e., described by a configuration file read by the system at boot-time.

For convenience, the prototype is implemented on top of an existing UNIX[3] operating system. The system configuration is illustrated in Figure 2.9. The nucleus runs as a UNIX-process, and forks MTS-Linda client processes by use of `vfork()` and `exec()` where exec requires that the executable file exists in the UNIX file system. The runtime system allows several process declarations (C++ functions) in a file, and uses a dispatch function to select the right function to be called. A small MTS-Linda runtime-environment is linked with each client process. This

[2]Sun, Sun Workstation are registred trademarks of Sun Microsystems, Inc.

[3]UNIX is a registred trademark of AT&T.

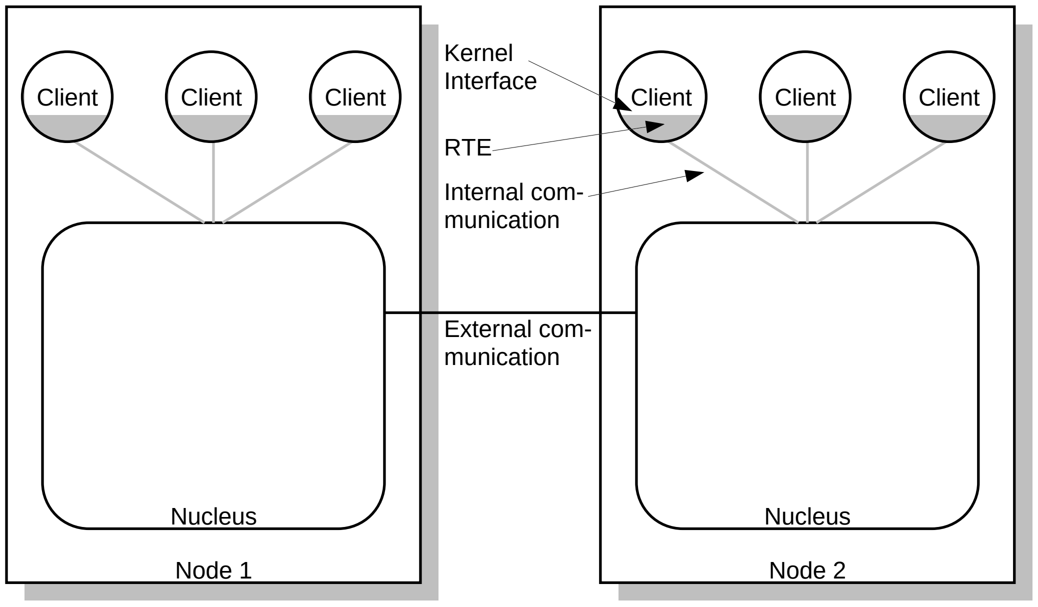

Figure 2.9: System configuration of the prototype. *RTE* is a short for runtime-environment.

converts MTS-Linda operations initiated by calls to the kernel interface into messages which runtime environment sends to the local nucleus.

The prototype assumes a reliable message passing protocol for inter-nucleus communication. BSD UNIX's tcp (transmission control protocol) sockets providing a reliable byte-stream protocol, fulfils this requirement. For simplicity Sockets are used both for client-nucleus communication locally at a single machine, and among nuclei implying sending messages across the network. The design is prepared for other interprocess communication mechanisms, such as shared memory. First, we outline the status of the prototype implementation.

### 2.3.2 Prototype Performance

MTS-Linda gives a high-level abstraction of a distributed system. As such, there is the danger that the abstraction is too costly in terms of performance. We are concerned with performance, and we regard it as being equally important as a good abstraction. The combination of good abstractions and good performance makes a practical usable system. This have motivated a closer inspection of the performance of the Linda concept, and of the prototype in particular.

We have made a sample of performance measurements, to determine whether or not the system show or can be made to show satisfactory performance. Following we present an extract of our measurements and draw a few conclusions. A throughout presentation of performance results, and how they were measured is given in Appendix A.

### 2.3.3 Tuple Exchange

We expect that the simple tuple operations will dominate the communication in MTS-Linda programs, because of their large number compared to tuple space manipulations. Measurements shows that a simple tuple operation costs 3 ms – 3.5 ms when destined internally (the tuple resides at the same node as the process issuing the operation), and about 6 ms – 6.5 ms when destined externally (the tuple resides on a node different from the process issuing the operation).

A synchronization via tuple space involves both an **out** and an **in**-operation. This motivates our requirement that the cost of tuple exchange as opposite to each single tuple operation, should be comparable with a remote procedure call. By a tuple exchange we mean a process doing a an **out** followed by an **in**, together constituting an **out-in**-pair.

The test shows that tuple exchange in the prototype are 2-3 times as expensive as remote procedure calls, and are thus far from meeting our requirement. A speedup of simple tuple-operations is needed.

There is a tendency that tuple exchange will always be slightly more expensive than remote procedure calls. Tuple exchange involves 3 parties and 2 synchronizations: one synchronization between tuple-producer and tuple-memory (kernel) for inserting the tuple into tuple space, and one between tuple-memory and tuple-consumer for removing it. In contrast, the sender and receiver of remote procedure calls knows each other without an intermediate station, and is thus a 2 party synchronization.

To put in a line of defense about the large overhead in the prototype, it was not designed and implemented to be optimal. Effort was on modularity and flexibility, thus compromising performance. Moreover, the interprocess communication facility (tcp-sockets) used between client-process and the MTS-Linda kernel is a major contributor of the prototype's inefficiency. 70% to 80 % of the cost of a tuple-operation is due to interprocess communication.

### 2.3.4 Tuple Space Manipulation

The cost of tuple space operations is proportional with the number of nodes spanning the tuple space ($2 * number\_of\_nodes$ messages). For a 4 node system the cost manipulating a tuple spaces starts at about 10 ms, and then increases linearly, dependent on the number of tuples in tuple space. Manipulation of tuple spaces as wholes is thus costly, but considering that they are likely to be rare, they have a cost of acceptable level[4].

#### Improvements

We have experienced that there is plenty of room for traditional sequential optimizations in the prototype, but sources for significant improvements is in the communication system. Improvements may include:

[4]If tuple space manipulations are used to debug running applications, as is a possible application of MTS-Linda first class active tuple spaces, the implied disturbance may be a problem.

1. Internal inter-process communication through sockets should be replaced by "kernel-traps", or tuple space should be placed in shared memory. This would bring tuple-operations down to about 1 ms internally and 4 ms externally.

2. Effort should be put into devising a communication system which removes the need for acknowledgment messages of **out**-tuples and tuple-space operations. The overall requirement for the communication system is that messages are causally ordered. ISIS[] reports a number of 350 causally ordered multicasts per second (2.9 ms each) for a process group of 4 members. We expect that use of these protocols could reduce the (initial) tuple space manipulation cost by a factor of 3.

3. Find an improved tuple-mapping strategy which reduces the number of cases where two processes residing on different nodes exchange tuples via a third machine; only one of the processes should do external access. Runtime monitoring in the kernel is a possibility, but may be difficult. Explicit programmer supplied hints or compiler support are more realistic, but also compromises transparency.

4. Programs which does not require "open" tuple spaces should benefit from the compiler based optimization techniques known from YALE implementations of Linda, see Section 2.3.5.

Finally, if a tailored low-level communication protocol for MTS-Linda could exhibit the same performance as the optimized remote procedure call communication of the Amoeba distributed operating system ([85] reports 1 ms for a remote procedure call) this would enhance performance considerably, allowing an external tuple exchange to take place within a couple of milliseconds. Hardware configurations should be exploited, e.g., broadcast hardware or sub-linear message diffusion in a transputer grid, to speed up manipulation of distributed tuple spaces.

### 2.3.5 Compiler Support

Yale implementations of Linda intended for parallel computing relies on extensive compile time analysis of Linda programs to reduce the overhead caused by match[14]. The compiler partitions tuples which cannot match into disjoint sets. It then classifies tuples further, dependent on how they can be represented at runtime: counting semaphores, queues, hash tables, private hash tables, and tables requiring linear search.

In the extreme case where tuple exchange (tuples with only constant fields) can be implemented as operations on a semaphore, [14] reports a reduction caused by the compiler optimization from $700\mu s$ to $200\mu s$ (on a shared memory machine). It it thus likely that compiler support has favorable impact on the runtime match overhead (this is at least evident in this one case).

However, these techniques is not always applicable in sharing systems. All potential tuple signatures for a tuple space must be known and analyzed at compile-time. This implies that processes, potentially residing in another tuple space, cannot insert or remove tuples of arbitrary kind in a tuple space whose runtime representation has been layed out by the compiler. Likewise, contents from a non-compiled tuple space cannot be inserted into the compiled tuple space (via `contents_move` or `contents_copy`). Compiled tuple spaces are closed entities, whereas open (standard) tuple spaces accept foreign tuples. The compiler could probably be implemented to automatically detect when it needs an open representation, and when it can optimize.

Nevertheless, programs not needing the openness of standard tuple spaces should still be able to benefit from these existing implementation techniques. In particular we expect processes with local parallelism encompassed in a local tuple space to benefit from compiler based optimization. We would like to support compiled tuple space with a $CompiledTupleSpace$. The kernel should provide the necessary handles and data structures needed by the compiler.

Because most of the cost of MTS-Linda implemented on distributed memory architectures is due to communication, it could be interesting to examine whether compiler support also could reduce the amount of communication. By analyzing the tuple templates and their producers and consumers it might be possible to find a mapping reducing the inter-node-communication.

### 2.3.6 Parallelism

Parallelism is a possible way to achieve speedup of program execution. Parallelism has often been connected with speeding up large scientific computations, but parallelism is also essential in conjunction with sharing systems, e.g., in a programming environment an obvious utilization of parallelism is to speedup compilation. We have therefore developed a small tool which can be used to execute UNIX shell commands (including compiler invocations) in parallel. The application demonstrates what kind of performance can be expected for parallel MTS-Linda applications.

The parallel make is a hybrid UNIX and MTS-Linda application. A traditional UNIX make tool calculates the set of files which needs recompilation, but instead of invoking the compiler for each of these files, it runs a MTS-Linda program (parmake), giving parmake the file-set as argument. Parmake takes 3 additional arguments: The number of processes which should work on the compilation, the UNIX compiler command including compiler options, and the path to the current working directory in the UNIX file system.

Parmake sets up a master/worker style parallel program (agenda-parallelism), as is customary for parallel Linda programs. First it creates a local tuple space, in which an unordered job-pool of tuples is set up, each containing the name of a file for recompilation. Parmake then creates the specified number of worker processes. The worker processes takes two arguments: a string describing the command to be executed, and a string denoting the path-name to the current working directory. Each worker iteratively grabs a job and invokes the compiler. Errors from the compiler is redirected to "error" files, which can be inspected (while or) after compilation. The worker terminates when it receives a null-job. Parmake does not make any assumptions about which UNIX command is to be executed, and don't know about source-code files, so the tool is useful for parallel execution of other UNIX tools than compilers.

Parmake also illustrates a possible application of MTS-Linda's module concept. Parmake is an example of a module which contains internal parallelism, but appears to its surroundings as a sequential thread. The local tuple space holds a parallel (distributed) computation spanning several nodes of the system. Multiple users may invoke the tool simultaneously without interfering each other (perhaps with the exception of a performance degradation). Tuples from one invocation are not mixed up with the other's. This would be the case with Linda's single global tuple space (or the applications could run on separate instances of a Linda kernel, but the applications would then not be able to share data or resources via tuple space). This is modularity in action.

Figure 2.10 outlines the source code for the worker processes, and Figure 2.11 outlines the master.

```
process int worker(char * command,char * cwd)
{
    //initialization
    char filenamebuf[FNSZ];                         //string to hold file name

    while(1)
    {
        in("Work", ?filenambuf);                    //grab job from tuple space

        if(strlen(filenamebuf)== 0)                 //if null job then terminate
            return(error);

        //Build command for execution
        sprintf(command_buf,"%s %s/%s 2>
%s/%s.err",command,cwd,filenamebuf,cwd,filenamebuf);

        error=system(command_buf);                  //execute unix tool
        out("Result",filanembuf,error);             //insert result tuple in tuple space
    }
}
```

Figure 2.10: Outline of source code for the worker process of the MTS-Linda parallel make tool.

We have tested parmake on a 15 node MTS-Linda system by letting it compile the MTS-Linda prototype kernel. The job size is approximately 21000 lines of C++ source code partitioned among 47 files. Compiling the job sequentially takes 885 seconds (14 min. 45 sec.) The same job, using parmake and a single worker takes 909 seconds (15 min. 9 sec.) yielding an overhead of 24 seconds compared to traditional make. These figures and those given below are measured as the duration of the entire compilation process, i.e., from the user enters the $make$ command until it exits with a linked and executable file. The figures thus includes the sequential overhead of generating file dependencies, and linking (15 seconds).

Figure 2.12 shows execution time as function of the number of workers. Parmake exhibits good speedup until 4 workers, but beyond 5 workers speedup declines and thereafter vanishes. However, the largest job takes 90 seconds to compile, and by adding the constant overhead of 15

```
//Master usage:   Parmake No_workers Command working_dir Files ...
int real_main(int argc, char* argv[])
{
    //initialization
    char filenamebuf[FNSZ];                         //string to hold filename
    LocalTupleSpace WorkSpace;                      //Tuple space for job-
                                                     pool and workers

    for(i=0;i<no_files;i++)                         //setup job-pool of filenames
    {
        strcpy(filenamebuf,argv[NO_ARGS+i]);
        WorkSpace.out("Work",filenamebuf);          //insert work tuple
    }

    for(i = 0; i < no_workers; i++)                 //create worker processes
        WorkSpace.eval(worker(command,cwd));

    for(i=0;i<no_files;i++)                         //display execution status
    {
        WorkSpace.in("Result", ?filenamebuf, ?worker_status);
        printf("Status %d was returned for file
%s\n",worker_status,filenamebuf);
    }

    printf("Compilation Finished --- Commencing Termination!!\n");

    for(i=0;i<no_workers;i++)                       //Terminate workers with a null-
                                                     job
        WorkSpace.out("Work" ,"");

    return 0;
}                                                   // due to future semantics of
                                                    //tuple spaces we here nicely
                                                    //waits until the workers termi-
                                                     nates
```

Figure 2.11: Outline of source code for the master process (real_main()) of the MTS-Linda parallel make tool.

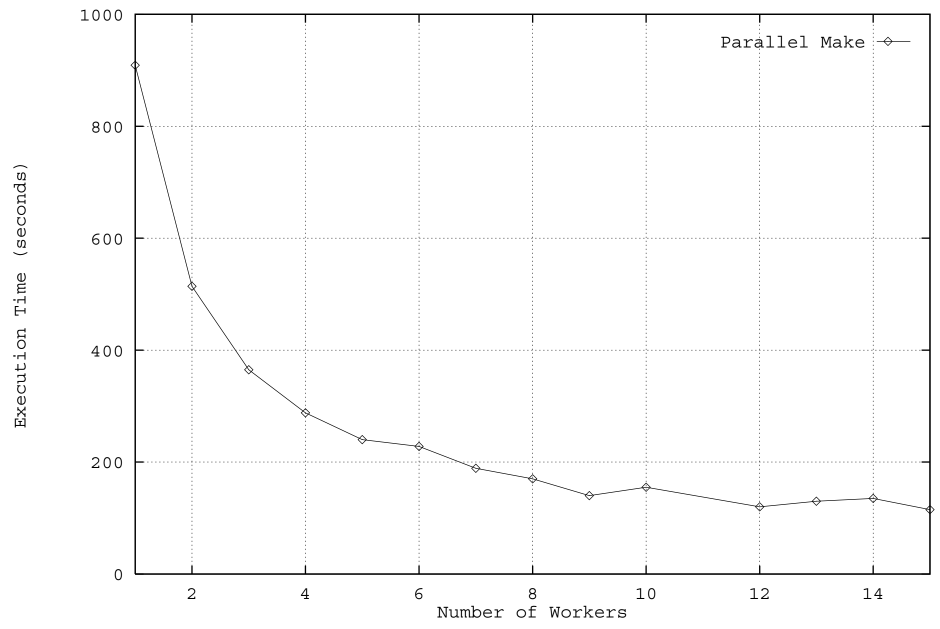

Figure 2.12: Test run of MTS-Linda parallel make. The figure shows the execution time in seconds as function of number of worker processes. The graph is averaged over 2 test runs.

seconds, making the best obtainable 105 seconds. In comparison, the best test run was measured to 115 seconds, which is close to the best expectable.

There are two reasons that the graph is slightly dented. First, the nodes have unequal processing capacity. Depending on how workers map to nodes the total compile time may be faster or slower. Second, it can be explained by taking a closer look at the job-pool. It does not prioritize jobs according to their size. In the best case, where the largest job is grabbed first, the other workers will cooperatively finish up the small jobs before the unlucky worker finishes, giving an overall compilation time equal to the compilation time for the largest job. But if the largest job is grabbed after a smaller one, the overall computation will be much delayed by the overworking process. So, the minimum time of 115 seconds also required a bit of luck.

In conclusion, parmake's performance is satisfactory. A cut down of $(885sec. - 115sec. = 760sec.)$ approximately 13 minutes was obtained. Thus, using the parallel make is clearly feasible. However, the parmake program is quite simple. It cannot be used for execution of commands with internal dependencies among jobs. A more advanced program must be implemented to take this into account.

Note that parallel make is an easy parallelizable problem, with a large computation/communication ratio, and is therefore not representative for all parallel problems. Other problems are likely to show better[5] or worse performance curves.

In addition, Kaminsky has in [52] and [53] experimented with parallelism and Linda on local area networks. The purpose of their scheme is to exploit the loads of spare CPU cycles in a local area network to execute parallel programs. Experience shows that this is both possible and desirable for a wide range of parallel problems. Unlike our example with a parallel make, they have experimented with schemes for the use of idle workstations. The essential problem is to figure out when a workstation is idle, or should be kept idle. One policy allows the owners of the workstations to specify when their workstation may participate in parallel execution. In our prototype, processors are organized in a processor pool where the kernel ofloads the process to an appropriate node where the process is started. Thus, the prototype does thus recognize the concept of owners of workstations sidestepping this issue. The approaches are similar in the sense that they make it possible for users to share the CPU-resources or a sharing system for parallelism.

**Performance Conclusions**

The prototype implementation of MTS-Linda is inefficient, but given the magnitude of the performance figures of the prototype, and the possible optimizations, we believe that the performance of a non-prototype MTS-Linda system will be acceptable, and performance is not a reason to discharge the idea of Linda with multiple tuple spaces. MTS-Linda gives a high-level abstraction, and can be implemented so its costs are kept at an acceptable level. We find it unlikely that tuple-exchange-times with current technology will be significally faster than a couple of milliseconds.

Our tests also show that parallelism is a possible means to reduce execution time of programs in sharing systems. However, as a consequence of the high communication and process manipulation cost dictated by current technology, parallelism is likely to be coarse grained; fine-grained parallelism with MTS-Linda on a network architecture is (currently) out of the question.

However, we conclude that MTS-Linda in an optimized production version only will cause modest overhead compared to the native run-time system. Largely it is the technology which puts limits to the performance of MTS-Linda, not the concepts[6] itself.

## 2.4 The Full MTS-Linda Model

We have have now presented the crippled MTS-Linda model used in the prototype design and implementation. The prototype model is only a subset of the model presented by Jensen[48]. The full model allows, as already mentioned, for processes as first class objects, and as a consequence for manipulation of active tuples and tuple spaces. Allowing for first class processes allows us to manipulate these both in the host-language and in MTS-Linda. We can suspend, copy, remove, and reactivate processes. And as a consequence can we manipulate active tuples and tuple spaces.

[5]Invoking the compiler is quite expensive, thus creating a large per-job overhead.

[6]This is valid under the assuming that MTS-Linda not inherently communicates more than other support programs, which we believe it does not.

The MTS-Linda model referenced and used in the remainder of this report is the full model developed by Jensen.

### 2.4.1 First Class Processes in The Host-language

The **process** qualifier, shortly introduced as part of the AUC-Linda model to qualify a C++ function as a process, is used to qualify a variable to hold a suspended process image returning a given type. Figure 2.13 line 2, show the declaration of an integer variable $g$ qualified as a process. This allows $g$ to hold a suspended process image which future evaluates to an integer.

```
1.  process int f(int x)
        { in(?y); return x+y};      // A function which when evaled becomes a pro-
                                      cess
2.  process int g;                  // A declaration of process variable g.
3.  eval("Test",f(7));              // Evaluate function f() in tuple space
4.  in("Test",?g);                  // Match the active tuple just put into tuple space
5.  out(9);                         // Place an integer in tuple space
6.  eval(g);                        // Make a copy of g and evaluate it
7.  touch(g);                       // Force g to evaluate its future
```

Figure 2.13: The usage of processes in **out**, **eval**, and **rd** operations.

A process bound to a variable is suspended and can only be reactivated if it is touched. Touching a process forces it to yield its future. Touching a process can done explicitly through the **touch** keyword, as shown in line 7 where $g$ is forced to evaluate its future (23), or implicitly when used when a used in, e.g., a MTS-Linda operation. A process variable obeys the scope rules of the host-language, thus leaving scope will delete (kill) the process bound to it.

### 2.4.2 Manipulating Active Tuples

Using a process variable as field of an **in** or **rd** operation allows us to match an active tuple in tuple space. The **in** operation in line 4 will match the active tuple put into tuple space by the **eval** operation in line 3, and the active process evaluating $f(7)$ will be suspended and bound to the variable $g$. Active fields are only allowed to match actual-to-formal. To avoid actual-to-actual process match attempts, are processes passed as actual-fields touched when used in an **in** and **rd** operation, forcing them to become a passive data value before they are used in the search for a matching tuple.

In line 4, the active tuple created in line 3 is matched, and the $f(7)$ process is suspended and bound to the $g$ process variable. Line 6 shows how a copy of $g$ is evaluated in tuple space, eventually resulting in a tuple $(integer : 16)$ as the $\mathbf{in}(?y)$ operation in line 1 will match the tuple put into tuple space by the **out** operation in line 5. In line 7, $g$ is touched resulting in the remainder of $f(7)$ being evaluated locally. $g$ will endup with the value $23$.

Besides active processes, a formal process field can also match a passive value corresponding to the future of a process. The $?g$ formal process field can match both a process which future yields an integer, and an integer field.

A process can potentially match the tuple it itself is part of, e.g., if the process $f$ declared in line 1 performs an $in(\text{"}\texttt{Test}\text{"}, ?g)$, it could potentially match the tuple it is part of. The result of the operation, is something like the process being suspended and bound to a local variable within itself which is meaningless. A process cannot as a general rule manipulate it self.

### 2.4.3 Manipulating Active Tuple Spaces

Being able to manipulate an active tuple, it is also possible to manipulate an active tuple space, by manipulating all the individual active tuples in tuple space. This gives us the ability to suspend, copy, remove, and reactivate whole computations.

A local tuple spaces in the prototype, has to become passive before if could be manipulated as a whole, as we could not manipulate active tuples. Now, being able to suspend, copy, remove, and reactivate processes makes it possible to manipulate active local and field tuple spaces in the same way.

Local tuple spaces are similar to processes: they are suspended when bound to a tuple space variable. And the touch operation forces a local tuple space to yield its passive state. Further, using a local tuple space as part of an **in**, **rd**, **out** operation will touch each tuple space passed as argument, allowing only passive tuple space in these operations. The **eval** operation reactivated a copy of each field tuple space and evaluates these concurrently.

#### MTS-Linda Operations in a Local Tuple Spaces

A local tuple space is conceptually a local data structure within the address space of a process and is, as already mentioned, suspended when bound to a variable.

The **in** and **rd** operations may reactivate a local tuple space, if they cannot find a match in tuple space. Further, these operations are defined to fail when it is determined for certain that the operation never can be satisfied. Intuitively a search for a match in a local tuple space goes through at most 3 steps:

1. The suspended tuple space is searched for a matching tuple.
2. If a matching tuple was not found in step 1, the processes in the tuple space are reactivated and running until the match is satisfied.
3. If all active components in the local tuple space terminates or endup blocked in an **in** or **rd** operation, the search for a match fails, and the **in** or **rd** operation unblocks the process returning an error.

The detection of a dead-lock situation in a local tuple space, and reporting this to the process holding the tuple space in form of a failure, allows for simple searching in passive tuple spaces.

## 2.5 Summary

We have described Linda, the foundation of MTS-Linda, as a simple model of coordination between parallel processes. It provides a shared memory abstraction $tuplespace$, where processes through 4 simple primitives, **in**, **rd**, **out**, and **eval**, can insert and remove data.

We have outlined the crippled multiple tuple space model used in our prototype design. It is an extension of the Linda model with first class tuple spaces. The MTS-Linda model have two different types of tuple spaces. Local tuple spaces which conceptually resides as part of a processes. And shared tuple space, living independently of processes. The local tuple space defines bureaucratic (hierarchal) coordination patterns, while the shared tuple space allows for organic (flat) coordination patterns.

The prototype implementation, constituting of approximately 21.000 lines of C++ code, have be distributed and runs on a network of SUN Workstations. Performance measurements show that the prototype is inefficient, but given possible optimizations a non-prototype MTS-Linda system will show acceptable performance.

Last we outlined the full MTS-Linda model which the prototype is a subset of. The full model introduces both tuple space and processes as first class objects, Allowing for manipulation of active processes and tuple spaces and hence distributed computations. Evaluation of MTS-Linda in the remainder of the report is based on this full model.

# 3 Sharing Systems

Applications of distributed systems are diverse and large in number. They span from high speed scientific parallel computing to multi-media systems. We therefore attempt to nail down the specific application domain of MTS-Linda: sharing systems. We here characterize 3 samples of sharing systems to analyze their requirements.

## 3.1 The Characteristics of Sharing Systems

A distributed system consists of two components, hardware and software. Software is the most important because it determines most of the system's functionality and how the system appears to its users [85]. The following characterize each in turn.

### 3.1.1 Hardware Related Issues in Sharing Systems

The hardware configuration of a typical sharing system is illustrated in Figure 3.1. The backbone of the system is a set of workstations interconnected in a local area network. Thus, the system is typically located within a single building. The consequence is that we deviate from very tightly and very loosely coupled systems such as respectively the transputer and internet systems.

Each workstation is autonomous in the sense that it has its own processor, memory, and some private i/o devices e.g., keyboard, mouse, screen. Some workstations also have a local disk. Additional input/output devices such as X-terminals, vt100 terminals, printers and scanners are usually connected to a hosting workstation.

Furthermore, hardware dedicated to storage and computing exist. A file server can be a computer especially manufactured for filing, and is thus good at it (fast, and good backup facilities), or it can be a less dedicated workstation only doing filing. Compute-servers are beginning to emerge. They provide fast execution of compute intensive applications. A compute server can be either a very fast workstation, or a shared memory multiprocessor, or a tightly coupled network of processors, such as a transputer network. The system may have connections to external systems. They are handled by a gateway, which is a machine or device which handles the communication with the external systems.

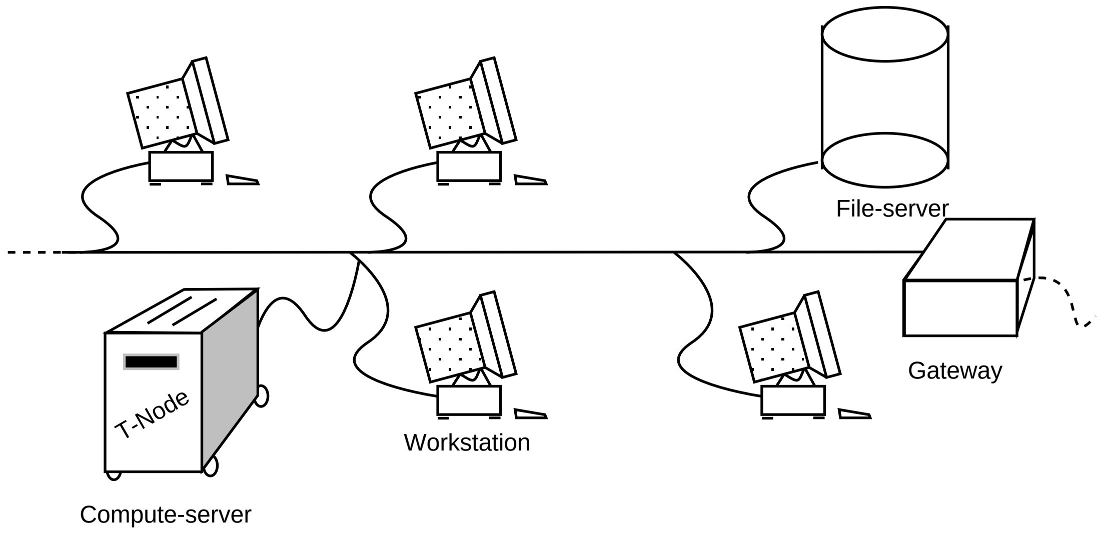

Figure 3.1: A model of a typically hardware configuration for sharing applications.

Current electrical networks are in the future likely to be replaced by high speed, giga-bit capacity, optical networks making communication less prohibitive than today. Workstations will become still faster, and have more main-memory. Multi-processor workstations are already beginning to appear. Future systems have more CPUs than users. Optical storage media provides for storage of large amounts of data.

With better communication technology the borderline between system components in today's computer networks disappears. [57] foretells that "... much of the distinction between shared-memory multiprocessors, tightly coupled multiprocessors (e.g., connection machine) and loosely coupled systems (e.g., network of workstations) disappears". If this prediction is correct, and we believe it is, we have the opportunity to describe any coordination using only one shared memory based model.

Wide-area-networks (the internet) of today is little used, but for file-transfer, remote login and electronic mail. With future world-wide optical networks, sharing will become possible across greater distances. "The result will be a set of general purpose and personalized computer systems with islands of very fast specialized computers and massive databases. Mankind will start to paint its dreams on this global computing canvas" [90]. Communication technologies with greater bandwidth will make it possible to give a closer integration of computers across longer distances.

### 3.1.2 Software Related Issues in Sharing Systems

A programming model convenient for sharing systems should encompass support for most of the problems encountered in sharing systems. What are the requirements and conditions for such a model? –We propose the following listing of issues which we use to analyze 3 examples of sharing systems: Multi-programming environments, computer supported cooperative work, and distributed operating systems. Afterwards we discuss MTS-Linda's abilities to support sharing systems.

- How is **shared information** to be maintained and kept consistent?
- Which requisites for **coordination** are required?
- What is the view on **protection**, and how should protection be handled?
- What is the view on **flexibility**, and how much is required?
- How should the system appear to its users? –Which aspects of the distributed system should be **transparent**, and which should be visible?
- How should **failures** be handled, and which degree of fault tolerance is wanted?
- Should it be possible to integrate **heterogeneous** computers?
- What are the **performance** requirements?

We dot not claim that the above is a complete list; but it contains most of the central aspects. We use these issues as a set of evaluation criteria, which can be used to characterize sharing systems and their support. In the following we describe in more detail the issues encountered in the enlisted questions.

#### Maintaining a Shared Data Space

An essential issue of a sharing systems is how a shared data space is created and accessed. We define *data* as the representation of information. A characteristic of shared data is its *life-time*. Data must often be retained beyond the life-time of the creator program, beyond the time the user has logged on or beyond the up-time for a machine, i.e., shared data is usually connected with the notion of persistent storage. Other kinds of data have a more temporary nature.

An equally important issue is the available means for querying and browsing the shared data space. The ability to extract exactly the desired information from shared data is one of the primary motivations for sharing in the first place.

Another aspect is how consistency constraints are defended. Incorrectly interleaved updates and queries resulting from failures and concurrency can possibly, if no countermeasures are taken, leave data in an inconsistent state. Sharing data is thus intimately related to coordination.

### Coordination

Concurrent activities is an integral characteristic of sharing systems. Multiple users are using the system at the same time, and different parts of a program may execute at distinct nodes simultaneously. This may result in non-determinism and a need for serialization of concurrent events. The means available for coordination is therefore an issue.

Coordination is the activity of creating new activities, and providing for communication among these. We define the elements of*coordination* as:

**Communication:** Exchange of information among two or more processes.

**Synchronization:** Time-wise ordering of events.

**Process manipulation:** Creation, termination, suspension and duplication of processes.

**Modularity:** Composition or structuring of ensembles of processes working jointly. The formation of a landscape of activity.

The ability for components to communicate and synchronize with each other is an unavoidable topic in sharing systems. Sharing is about interaction, and coordination is about enabling and controlling interaction. It may seem unusual that we count modularity as a coordination phenomena, but we put process groups, objects, and tuple spaces on this shelf, and these relations have much impact on with whom and how we can communicate and synchronize. A further issue which quickly may become relevant in a sharing system is about restricting interaction: protection.

### Protection

An important issue in sharing system is the ability to share data and resources. But an equally important issue is the ability to prevent users from accessing data they have no business looking at. The users may not be able to trust each other; they may try to read or manipulate information to their own advantage, and may participate in a struggle for resources.

The goal of *protection* is to prevent unauthorized access to data and resources. The main issues of protection includes according to[70]:

**Secrecy.** The ability to keep data a secret.

**Privacy.** A guarantee that data can be kept secret.

**Authentication.** It must be possible to ensure that data are authentic.

**Integrity.** The system and unauthorized access must not corrupt data.

Whom may do what with which objects is largely a matter of policy, and policies should be separated from the mechanisms which enforce policies.

### Flexibility

The use of and requirements for a sharing systems may drift with time, and hardware and software quickly becomes old fashioned. It is an issue whether, and to what extend, these components can be replaced by new and modern ones which conform with the new required functionality.

We term the ability to change programs or modules designed in isolation, and compiled and loaded at disparate times, in a running system as *reprogramming*. Newly implemented programs or modules may be added, or overtake the role of existing one to provide the new wanted functionality. Or old modules may simply be removed.

The software of a sharing system can be regarded as a dynamically evolving set of autonomously and concurrently executing active entities that operates without complete knowledge of each other. They are prepared to communicate with any other components using an agreed upon protocol. The entities may be written in different programming languages, and may both cooperate and conflict to achieve their goals. The entities must be able to share resources and negotiate to achieve a common goal. Such systems have in[42], [51], been termed *open systems*[1]. Hence, an issue is to what degree the system should be open for association with foreign subsystems.

### Transparency

A distributed system's appearance to its users is characterized by its transparency properties. An object, data or resource, in a distributed system is located at some node though the exact location can be either visible or transparent to the user. If it is transparent, the user has no means of determining an object's location. If the location is visible, he can determine where the object is located, and might have primitives enabling him to place objects on specific nodes.

A distributed system may give transparent views on several distribution aspects besides location Figure 3.2 list the more important ones, and shortly describes how they are manifest[37, 85].

A system may try to give a full transparent view of the distributed system, i.e., transparency in all aspects, or only in some of these, or may allow selection between a transparent or a visible view.

### Failures

Failures in a distributed system is a delicate issue. Many more possible points of failures exist in a distributed system, than in a centralized one. The individual computers may fail, and the communication network may be errornous or broken; especially it may fail in such a way that the network becomes partitioned.

Handling failures in a distributed system has many facets. One is to keep data consistent and available in spite of failures, or recoverable after failure. The classic example here is in a banking

[1] One of the authors personal definitions: In open systems you do not know all the source code at compile-time `:-)`

| Transparency | Description |
|---|---|
| Location | Physical location of an object is not visible. |
| Access | Access to an object is the same regardless of its location. |
| Migration | An object can move without the user knowing. |
| Concurrency | Concurrent access to objects is handled automatically. |
| Replication | Multiple instances of an objects may exist, and consistent update to these is performed automatically. |
| Failure | The failure of an object is handled automatically by the system. |
| Parallelism | An application may execute in parallel without users knowing. |

Figure 3.2: The main types of distribution transparency.

system, where a failure of a transaction may cause accounts to imbalance. Money must not disappear. *Consistency* is when all constraints are met.

A distributed system can be designed to have the partial failure property. This allows all workstations but the failing still to be operational. In[86] Tanenbaum defines a *fault tolerant system* to be one that can continue functioning (perhaps in degraded form) even if something goes wrong. *Availability* is a term used to denote the fraction of the time the system is usable.

The presence of failures implies re-consideration of the semantics of the communication primitives. Another hardware related problem which have severe impact on the software is heterogeneity.

### Heterogeneity

The hardware of a distributed system is, in general, heterogeneous. An important issue is whether heterogeneity is buried in the lowest levels of the system, and thus hidden from the users, or if they are able to exploit the computers different computing abilities.

According to [37] there are 3 types of heterogeneity. The first case is when the computer hardware uses incompatible data-representations or instruction-sets. Second, the network may consist of parts with different transmission techniques (e.g., token ring vs. CSMA/CD ethernet) and interfaces (e.g., electrical vs. optical). Finally, the computers may run different operating systems.

**Performance**

Performance is always an issue. Users are interested in making the most of their hardware, in getting fast response times, and in having their programs executed as fast as necessary.

In a distributed system the communication network may be a problem. With current technology access to the network is expensive, and a overloaded or saturated network will slow down the entire system. The network thus influences how and how much the applications running at top of it may communicate.

Because a distributed system consists of several processors, the idle ones may be employed in parallel exection of a program to achieve faster execution, and thus improve response times.

### 3.1.3 Summary

The above was a classification of a number of important issues in sharing systems. A support system should address and decide on these. It should be clear, just by looking at the number of issues, that supporting sharing systems may be difficult.

In order to better understand the needs of sharing systems, and to put these in perspective, we in the following 3 sections analyze 3 cases of sharing systems. Hopefully, we will also be able to answer some of the questions raised in the start of this section (Section 3.1.2).

The three kinds of systems were obtained by talking to three research groups at our department: the programming systems group, the information systems group, and the distributed systems group. The replies were respectively multi-programming environments, computer supported cooperative work, and distributed operating systems.

## 3.2 Multi-programming Environments

As software developers we are quite familiar with at least one multi-user application of a distributed system: program development. Programming takes place in a programming environment:

> A *programming environment* is an integrated collection of programming tools. A programming tool is a piece of software or hardware that can be used to support a well-defined task during the program development process. [72]

Thus, the programming environment constitutes the programmer's work place and tool box. Ideally, a programming environment should support all functions of program development: Program construction, program examination, and program administration.

The development of large programs often requires participation of many programmers, perhaps each with their own graphical workstation. A programming environment that supports multiple-users, located at different machines, in cooperating on the same project has in[38] been termed a *multiprogramming environment*. Multiprogramming environments impose new questions not found in single-user, single machine environments. Consider for instance the following:

- What should happen when two programmers simultaneously work on the same document? Should the environment prevent this, should it notify the programmers about it and leave it to them to sort out the conflict, or should it try to merge the changes made by the programmers?
- If a programmer alters a common module, should this change be visible to other programmers immediately, or on command by the programmer?
- How are multiple versions a program managed when its components cooperatively developed?
- Should the environment supply tools for inter-programmer communication, or is this best taken care of face-to-face?
- Where are information stored? –Is it stored on the creator's machine, in a centralized database, or on some other machine with low load?
- When a programmer invokes a tool, is it then running at his own machine, at a machine of choice, or at one the environment picks for him? –Is all of this transparent to the programmer?

### What is Distribution used for?

Distributing information in a multi-programming environment has several advantages compared to a centralized "programming environment server". According to[89] efficiency and reliability are the main reasons for distributing the hypertext-database of the EHTS hypertext based multiprogramming environment. Efficiency and reliability was also the first two reasons for distributing the FENRIS hypertext back-end in [38], but they also include scalability and heterogeneity.

**Efficiency:** By distributing the hypertext-database to several machines the maintenance load is shared, and a bottleneck has been avoided. Further, documents may be located close to the programmers which use them mostly; this provides fast access.

**Reliability:** When a machine crashes in a properly designed distributed system, only the information stored at that machine becomes unavailable. Other programmers, which are independent of information from the crashed machine, can continue to program. Increased availability is obtainable by replicating information to several machines. The disadvantage of replication is increased difficulty in keeping the replicates consistent, and a potential performance degradation as result of the management of replications. Replication could also increase performance, though, since the information is more likely to be close to its point of use.

Furthermore, since each programmer is assumed to execute tools at their own workstation, the total load caused by tools is also distributed.

### Coordination and Protection

A predominant issue in programming environments on distributed systems employed by multiple programmers is how the coordination among programmers take place. Dependent on the size of the programming environment this coordination takes very different forms.

In [75] the authors have proposed a taxonomy for programming environments from the viewpoint of scale (number of programmers). The authors uses a sociology metaphor and distinguishes between 4 classes of programming environments: The individual, the family, the city, and the state.

The individual class is used by a single programmer, and typically consists of a closely integrated set of tools supporting a single language. The highly recognized SmallTalk 80 programming environment [36] is an example of an environment which only supports a single user at a single machine.

The family class model are used by small groups of programmers, relying on the assumption that programmers are trustworthy—they are not intentionally hostile to each other. A characteristic of the family model is that of enforced coordination; programmers may be clumsy[2], and may alter the same document simultaneous or remove a document which definitely should not have been removed! Family class environments provide the necessary facilities for concurrency control, e.g., version control, locking mechanisms, and atomic updates, which aid programmers to avoid such unfortunacies.

The primary characteristic of the city model is enforced cooperation. A requirement for environments with many programmers is a system for regulating and restricting the freedom of the individual programmers. Security is a problem. The environment provides the means for specifying cooperation policies, assigning different roles to the users, e.g., managers and programmers may have different rights for different documents. Moreover, the environment needs to provide a forum for negotiating and resolving conflicts, either initiated on demand by the environment or voluntarily by the programmers.

Very little research has been done within the state class model. Focus is on commonality policies, e.g., the use of a common software development process model. To reduce cost and improve productivity, standardization of interfaces, methologies and languages are issues.

Coordination is mostly initiated by the programmer; he invokes tools, and composes tools to the desired functionality. In a rule-based environment much of the manual coordination is seized by tools to coordinate tools, [13]—generalized make-tools. As noted, the environment may also enforce coordination and cooperation among programmers to ensure consistency of shared data.

### What is Shared?

The main reason for sharing in a multiprogramming environment is that programmers have easy access to documents developed by others. The documents may be specifications, interface

[2]We know at least 2 clumsy programmers!

definitions, documentation, source-code, object-code/executables (even the state of a program during execution etc.)

There are 4 problems in relation to maintaining shared documents in a programming environment: persistency, searching and browsing, consistency, and visualizing state changes.

**Persistency.** Means for persistent storage must be available. Documents such as source code or reports have long lifetime and it takes a long time to develop these documents. When a tool terminates the contents of its address space is lost, and because it may take several sessions to develop a document, there is a need to store documents between sessions.

Further, the documents must not be lost. Loss of primary memory (RAM) can for example take place due to software or hardware errors or if the power-supply drops out. Even a rare event such as a disc-crash may be disastrous for critical documents. Programmers grow hot from anger if they loose more than a few minutes of work, and may become very unhappy with the programming environment if this happens too often.

In addition, current computers have insufficient primary memory to contain all the necessary documents of a modern computer system. The storage capacity of a process is limited to the size of the virtual address space, which according to[85] is insufficient for some applications like airline-reservation or record keeping in large companies. We believe that this also applies for programming environments.

**Searching and Browsing.** The ability to extract exactly the desired information is in the heart of programming environments. For example, the use of a module requires knowledge of its interface. Looking the interface specification up quickly and making other related documentation easy available greatly helps the programmer. Thus, means for querying and browsing shared documents must be available to the users.

**Consistency.** The next problem is consistency. Updating a data-structure usually implies using a sequence of simple update-actions. The sequence of update actions may temporarily create inconsistent intermediate states, i.e., states where one or more consistency constraints are violated. A concurrently running update-sequence reading an inconsistent intermediate state may, if no countermeasures are taken, produce inconsistent output. The correctness relation of concurrent updates is usually that of serializability; the produced output is same as if the two updates were executed in some sequential order[55].

Further, a hardware failure may abrupt the update-sequence, with the inconsistent intermediate states as the result. To take care of failures the correctness relation is atomicity; the update has the effect as if it was executed entirely or not at all. Thus, means for defending consistency in spite of concurrency and failure must be present in the environment.

**Visibility of State-changes.** The final problem we will discuss is concerned with the visibility of changes made by multiple cooperating programmers. There are 2 related issues when shared documents are updated: when the actual state-change takes place, and when other programmers view of the shared document as consequence is changed. Suppose one programmer is reading the shared document and another is editing it. When is the changes made by the editor actually reflected in the data-space and when is the viewer's display updated? –This can be done at several granularities.

In one extreme each programmer gets their own version of the document at login-time. At logout-time their version is re-stored in the data-space. Another granularity is per session-basis. The state-change takes place at the end of an editing session—when the document is "saved". Similarly, the other programmer's view change when they re-read the document. In the other extreme as well state as view changes takes place in real-time. As soon as the editing programmer hits a key, all others' view also change. To support a intimate cooperative situation changes of state and view should take place in real-time, and thus a support system should provide the means for these, fine-grained state and view changes.

Examples of multi-programming environments include CASE-tools (Computer Aided Software Engineering) and hyper-text based environments. Both are based on database management systems (DBMS), either general-purpose or specially tailored. The CASE-tool StP (Software through Pictures) uses a general purpose DBMS to store diagrams and documents. The atomic transactions of DBMSs have in [77] and [89] been criticized for their lack of support for cooperation—atomic transactions supply the user with a single user view. Atomic transaction is also inadequate for long-term transaction, such as editing a file. This will block the transaction of other users in a intolerable amount of time. Locking and notification are also used strategies, which suffers from the same problems.

### Extendability

Programmers like to tailor their environment, and extend the environment incrementally with tools of own production to obtain the wanted functionality. An important question in the design of environments is the underlying data-structures shared by tools—how objects are represented. A byte stream representation like UNIX-files makes sharing of data among tools easy, but requires parsing in each tool. In contrast, representing data as abstract syntax trees etc. does not require interpretation of type-less streams, but according to [75] this makes it difficult to integrate additional tools.

### Transparency is the Issue

The fact that the programming environment is using multiple computers should be transparent to the programmers. Programmers should be able to put their brains at their tasks, not in location of information, in how data is stored, and at which machines their tools run. The programming environment should be ready-at-hand, not present-at-hand. No matter at which machine the user logs in, the user should be presented with the same view on the location of resources. It is convenient for a user or programmer to be able to use the same name, for a device or some shared data, independent of his current location or where his program is running at the moment.

Programmers are interested in a fast and smooth retrieval of information and of fast execution of their tools, and the programming environment should provide for this. If this can be done transparently, the programmers are happy—under the assumption that other programmers are only causing a tolerable load on his machine.

An exception to the trend of desiring transparency is concurrency transparency. In multiprogramming environments users should be aware of the cooperative situation, of other users and the updates made by these.

#### Summary

Multiprogramming environments is an example of sharing systems used by multiple programmers, located at different machines, sharing and cooperating on development of programming documents such as source code and related documentation.

We have encountered a number of indications of requirements for sharing systems. Among others that sharing systems have different characteristics based on their scale. In small environments, the users can trust each other, while this is an invalid assumption in large systems. Further, data sharing and the implied maintenance is seemingly a requirement.

The next case of sharing systems we analyze is a generalization of a problem also found in multiprogramming environments: How can we apply computers to support cooperation among people?

## 3.3 Computer Supported Cooperative Work

Until recently, computers have mainly been used for applications supporting a single user. The ability to share resources and data—introduced with multi-user operating systems—has introduced computers as a medium for cooperation among users. This broad field is known as *Computer Supported Cooperative Work* (CSCW), and includes the Multi-programming Environments in the previous section as an instance. CSCW applications puts emphasizes on *cooperation*, i.e., support for several humans cooperation to achieve some goal(s). CSCW applications attempts to support the rich patterns of inter human cooperation.

Applications supporting cooperative work includes argumenting collaboration and problem solving within a group co-located in real time, real time tools for collaboration among people, and applications for asynchronous collaboration among teams distributed geographically—multimedia. Another example of a CSCW application is a multi-person-calendar where multiple users share a computer-based calendar. The calendar allows users to examine entries made by them self and by others. Further, users have roles which define what entries they are allowed to see.

CSCW research includes topics from both computer and social science. The goal is to “define” how groups work, and to discover how technology (here computers) can help them do so[26]. Our interest in CSCW are those of computer science, and especially those related to distributed systems and sharing. Groupware is often seen as a synonym for CSCW, and in our context we will use it in this sense. Ellis [26] defines groupware as follows:

> Groupware: computer-based systems that support groups of people engaged in a common task (or goal) and provide an interface to a shared environment.

Our main interest—as outlined in the previous section(s)—are distribution and sharing aspects. CSCW applications are usually inherently distributed, and is thus as a rule implemented in distributed environments. Cooperation is based on a shared environment through which users cooperate. The remainder of this section outlines the characteristics of distribution in CSCW and its shared environment.

### Coordination in CSCW

CSCW applications may be characterized by the coordination required by the applications, i.e., to what extend the participants of a collaborating group are separated in real-time and geographically. Figure 3.3 captures this in the groupware time space matrix[26]. CSCW applications are found in all four quadrants of the matrix, e.g., meeting room technology would be in the top left, real-time document editors in the bottom left, bulletin boards in the top right, and electronic mail system at the bottom right. CSCW applications therefore need the support for both synchronous and asynchronous communication, and orthogonal to this the ability for users to interact even if they are in disjointed space.

| | **Same Time** | **Different Time** |
|---|---|---|
| **Same Place** | Face-to-face Interaction | Asynchronous Interaction |
| **Different Place** | Synchronous Distributed Interaction | Asynchronous Distributed Interaction |

Figure 3.3: The groupware time space matrix, classifying coordination among users according to their relative position in both time and space.

In general, coordination among members of a group cannot be seen as one-to-one, but rather as many-to-many. Group communication must enable a user to communicate both one-to-one and one-to-many; it must be possible to have private "conversation" and to participate in conferences counting several participants. Typical examples from the UNIX internet are: synchronous `talk` and asynchronous mail representing one-to-one communication tools, and `IRC`[3] and news representing one-to-many communication. Both one-to-one and one-to-many communications is found in all quadrants of the groupware time space matrix.

[3]interface to the Internet Relay Chat system.

### Shared context

Sharing of objects—that is information and resources—plays the central role in CSCW applications. *Sharing is obtained through a shared context, that is a set of objects where the objects and the actions performed on the objects are visible to a set of users (the group)*[26]. There is a need to specify the types of objects in a shared context, need for ways to describe what actions are allowed on what objects, and descriptions of whom may access what objects and how. Traditional applications handling these specifications are databases, allowing multiple users to execute transactions on a shared data structure. This may also take place through operating systems where multiple users share resources such as printers, scanners, and files.

The shared context of a CSCW application is characterized by its potential dynamic structure, and ability to contain a multitude of different objects from text in a mail message, to spoken words and live-motion video in multi-media applications. The static structures of traditional databases are not sufficient to capture this shared and dynamic context. More flexible means of handling the dynamic structure is needed.

An important issue in CSCW applications is the awareness of other users, and the actions they perform. Several users may concurrently access objects in a shared context, e.g. a shared design document. Users have interest in the state of some objects, i.e., in their shared context. Actions of other users may result in objects changing state, forcing a user to “monitor” objects to continuously be informed of changes, e.g., when the interface specification for the module he works on is changed. Inherently, cooperation implies knowing what other users are doing.

However, issues of access control and user rights, i.e., protection mechanics are not yet clear within the CSCW application domain. Furthermore, development of group interaction theories and user roles are needed before adequate models for CSCW protection mechanisms can be developed [77]. It is clear that protection is necessary in a system with multiple users to stop malicious or inexperienced users in doing harm. It must be possible to protect information, on both single user and group basis. However, the dominating strategy having a single person, the super-user (database or system administrator) as the only person to assign users roles (e.g. place users as member of a group) is insufficient. It implies centralized control, and further he may be unavailable, or may become a bottleneck.

Besides protection, the roles assigned to a user influences how he views the objects of the shared context. A manager is concerned with the overall state of a project, where the programmer only needs be concerned with details of some part of the project. The interface to the shared context must be customizable by the user, allowing him to extract exactly the information he needs in his daily work.

### Selective Transparency

CSCW applications benefit from distribution transparency. Users are interested in with whom they interact, but do not care where they are located. The UNIX talk utility requires the user both to specify the username and location to start a talk session. In an ideal CSCW application the user should not need to be concerned with the location of the peer with which he wants to establish a talk session, as this is not essential.

In another situation, the user could be interested in the location of the peer, because he intend to talk face-to-face. CSCW encourages a user-centered approach to control of distribution, where the needs of the user decides which aspects should be transparent and which should not. The view taken is that the user knows best what transparency he needs.

CSCW application's requirements regarding distribution is currently best provided by distributed operating systems [77]. There are problems though, CSCW, encourages a user-centered approach to distribution control in the development of cooperative applications, whereas distributed operating systems gives a system-centered approach. Instead of the more-or-less fixed transparency characteristics of a distributed operating systems, there is a need for some sort of tailoring of the transparency presented to the user (selective transparency), and thereby giving developers the flexibility demanded by cooperation applications. According to[77] CSCW applications are unhappy with full transparency because:

> "The problem with this approach is that presumed control decisions are embedded in the system and hence cannot be avoided or tailored for specific classes of application. This is the root of the problem in supporting CSCW. Because of the dynamic requirements of CSCW applications, it is unlikely that such prescribed solutions will be suitable."

### Performance is a Key Issue

CSCW applications strives for computer based cooperation which is just as natural as face-to-face interaction, found mainly in the multi-media, hyper-media areas of CSCW. For this reason, speech and pictures need be sent between interacting persons, giving a real-time feeling. Transferring telephone quality speech across a net-work in real-time is possible and does not require too much bandwidth, but transferring motion picture in real-time is much more expensive requiring large bandwidths.

### Summary

In conclusion, CSCW is about using computers for supporting the rich patterns of inter-personal interaction found in cooperative working situations. CSCW applications requires means for coordination among users, both co-located and co-existing as well as among users separated in time and in space. Further, CSCW requires selective transparency, and in particular it wants awareness of the presence of multiple users.

Our next example of a sharing system is the contemporary version of traditional resource sharing operating systems, which is often seen as foundation for supporting sharing.

## 3.4 Distributed Operating Systems

Distributed operating systems may be seen as the latest continuation in the evolution of operating systems. An *operating system* is a program that controls the resources of a computer and provides its users with an interface or virtual machine that is more convenient than the bare machine[86]. We enter in the history of operating systems systems with *time-sharing operating systems*. Time-sharing operating systems were justified by the observation that users, seen from the CPU's point of view, spend oceans of time thinking and drinking coffee instead of using the computer, resulting in much idle CPU cycles. Thus, greater computer utilization was achieved by having several users share the same machine. A time-sharing operating system provides users with their own virtual single user machine, though the physical machine is shared among several users. The problem with these centralized mainframe based systems is that they are expensive compared to their computing power, and they scales poorly.

With the advent of cheap and yet powerful personal micro-computers (PC's and workstations) came *Network operating systems*. Network operating systems are like traditional (time-sharing) operating systems, only with networking capabilities. Networks consist of autonomous machines, which are able to exchange messages, and which provide services to allow users to use remote facilities. Characteristically for network operating systems is that users are aware of multiple-computers, and that explicit notions are required for remote access, e.g., the rsh (remote shell), rlogin (remote login), and rcp (remote copy) commands of UNIX (SUN OS 4.1). Newer network operating systems often allow a workstation to perform transparent file access across the network. This gives its users the same view on his files no matter where he logs in. Further, networking operating systems provide automatic forms of communication and information sharing, which is more convenient than doing remote copies, e.g., SUN NFS (network file system).

Currently, much research in the field of operating systems is concerned with distributed operating systems, e.g., [85], [78], [3]. A *distributed operating system* is one that looks to its users like an ordinary centralized operating system but runs on multiple, independent central processing units (CPUs) [86]. An important goal of distributed operating systems is to provide a user with the illusion that he runs on a uniprocessor system, relieving the users from being concerned about the distribution, resulting in a simpler interface to the distributed system. At the same time distributed operating systems try to earn the advantages of scalability, reliability, and economics of distributed systems.

To achieve the view of a virtual uniprocessor, transparency is the most important issue in the design of distributed operating systems. Characteristically for distributing operating systems is that users are not aware of on which CPUs their processes are running, and of where their files are stored.

### What is Shared?

When moving from a network to a distributed operating system the machines become closely integrated and give up their autonomity in the interest for optimality for the system as a whole. With a large number of machines running a distributed operating system, it is likely that many

CPUs are idle. Other users which need the extra resources can benefit from these, giving a better overall utilization. However, the close integration may also decrease performance, due to increased inter-processor communication.

Further, with the decreasing cost of memory and micro-processors there may even be more CPUs in the system than users. Moreover, special compute-server devices are beginning to emerge. Thus, a distributed operating system provides each user with a larger amount of resources (e.g., disk space, memory and CPU power), than do uniprocessor systems. A distributed operating system provides for sharing of computational as well as perhipherical resources, and gives transparent access to these.

Sharing of data usually takes place through a file-system. A file is often in UNIX-like operating systems am unstructured byte-string—no interpretation of the contents is given. The files have names which are organized in a hierarchical global name space[4]. The file-system is location transparent i.e., the files have the same name disregarding from which machine it is being accessed. The files may, if appropriate permissions are set-up, be accessed by several users. If the user needs concurrency control he must often lock and unlock the files explicitly.

## Flexibility

The current trend in distributed operating system is to use a design based on micro-kernels. According to [85] a micro-kernel at each machine handles inter-process communication, memory management, basic process management and low-level input/output. The file-service, the name-service, the print-service and the other operating system services are supplied as user-level server processes, each managing a specific function. Interaction is often based on the client/server model with Remote Procedure Call as the primary means of communication.

The micro-kernel scheme is more flexible than a monolithic-approach where all functionality is assembled in one large kernel at each machine. Services can be added, modified or removed without halting and rebooting the system. If the user is dissatisfied with a standard service he is free to rewrite it. Since the difference between standard operating system services and user supplied servers is weak, the new functionality of user-supplied services is well-integrated with the existing system.

This flexibility is especially desired due to the immaturity of the field; leaving all initially chosen policies to user-level processes makes it possible to change these if it turns out that they are bad. Additional advantages of micro-kernels are modularity and simplicity—a small kernel is more likely to be implemented correctly than a large. However, an advantage of monolithic operating system is that of performance; trapping to the kernel is faster than sending messages to a remote server.

[4] A least one exception is the Saguaro distributed operating system,[3], where all data-items in the system are typed according to its universal type system.

**Summary**

Distributed operating system manages the resources of the distributed system, and gives its users a virtual uniprocessor abstraction of the distributed hardware. Operating systems are concerned with giving its users a fair share of its resources, and in contrast with CSCW and multiprogramming environments, an operating system sets up protective walls to provide secure services and to hinder undesired interaction. Due to the immaturity of the field flexibility even at kernel level is highly attractive.

## 3.5 Summary

This chapter has given an analysis of the application domain of MTS-Linda: sharing systems i.e., distributed system applied by multiple users to share resources and data, and to communicate. On the basis of a list of relevant issues in sharing systems we have analyzed multi-programming environments, computer supported cooperative work, and distributed operating systems. These are not totally different: A multi-programming environment can be regarded as a special case of CSCW where the work being done is programming. Distributed operating systems may support CSCW applications, and a distributed operating system is nearly useless if it does not provide a programming environment.

However, there are also differences. A distributed operating system strives for unawareness of multiple users, whereas CSCW applications strives for setting up a cooperative working situation. In distributed operating systems the focus is on resource sharing; in CSCW the focus is on human-interaction through a computer media, and important issues in multi-programming environments are sharing of program-documents and to find a common data-representation which can integrate the environment's tools.

On basis on the analysis given here, the subject of the next chapter is to prescribe how to support sharing systems.

# 4 Supporting Sharing Systems

The previous chapter analyzed 3 examples of sharing systems with different perspectives on sharing: data sharing, communication, and resource management. Based on this analysis we new move to the tough question of how to support these. We present a list of requirements, and compare MTS-Linda's abilities with these.

## 4.1 The Functionality of Sharing Systems

Using the terms of CSCW, cooperation among users takes place through a shared context, which may show up as shared data or resources. It is essential for a support system that it is possible to establish and maintain such a shared context. Users cooperating through a sharing system wish to be aware of the activities of other users and encourage the propagation of these activities between users.

Further, sharing systems makes use of distribution. As consequence of presence multiple users and that they often are located in different rooms makes a network of computers working together well suited to support groups of users. As captured by the groupware time space matrix of Figure 3.3 the interaction patterns among users may be classified by the users placement in space and in time.

Thus, the overall functional requirement for sharing systems is that it should provide for interaction among multiple users possibly separated by time and/or by space, and enable them to establish and maintain their shared context.

## 4.2 Requirements of Sharing Systems

Based on the analyses of the 3 examples of sharing systems, we in the following sum up the requirements for a sharing support system, presented in terms of the classification of issues:

**Maintaining shared data:** It is essential for a sharing support system that it is good at arranging sharing, cooperative development and modification of data. It should specifically allow for:

- Persistency for storage of documents.
- Searching and browsing for desired information.
- Consistency despite concurrency and failures.
- Multi-user awareness to give a cooperative working situation by making state changes incrementally visible.

**Coordination:** Equally essential is coordination. A support system must be good at synchronizing access to shared data and resources, and provide for communication and interaction among the tools and users.

- Powerful means for synchronization and communication among a dynamic collection of asynchronously processes must be available to allow the programmer to express the nontrivial interaction patterns of sharing applications.
- Modularity: It must be possible to group related processes and data, and to defend the integrity of these by strong encapsulation.

**Transparency:** Transparency significantly simplifies the programming of distributed applications as programmers and designers need not worry about irrelevant aspects of the underlying distributed system, such as location and failures. But full transparency is often too inflexible.

- The 3 examples suggest transparency as the default, but visibility as an option, i.e., selective transparency.

**Flexibility:** Sharing systems are continually evolving. In addition, flexibility of the support system is important because sharing systems are very diverse. Thus:

- It should be possible to add, remove, and to reprogram components of the system without terminating the system (reprogramming).
- Dynamic reconfiguration of hardware components should be possible.
- Possibilities for tailoring the support system to the applications' needs is considered highly useful.

**Protection:** Protection is a requirement in all 3 examples. The 3 examples focus on secrecy, privacy, and authentication, i.e., the rights of a user or a group to read and alter the shared context.

Hence, protection is in general a necessity, especially in sharing systems beyond the family model, but the protection scheme should enable cooperation.

**Failures:** We have the following requirements regarding failures in sharing system:

- A sharing support system should possess the partial failure property.
- It must be capable of detecting failures, and provide for handling of these.
- It must make the programmer potent for defending consistency of data.
- High availability of shared data and services is important.

**Heterogeneity:** The hardware of a sharing system is in general heterogeneous.

- A support system should make it possible to integrate heterogeneous components to form a coherent, coordinating system.
- It should allow exploration of the difference in computing capabilities possessed by the heterogeneous components.

**Performance:** The overall requirement for interaction with a sharing system is what people experience as “real time”. Predictable response times, preferably “immediately”, are desired. A sharing support system should enable the use of parallelism as a means of speed up.

In general, the requirements can be characterized as being motivated by functional, organisatorical or technological concerns. Functionally motivated requirements determines whether the system can serve its purpose. For a sharing support system this equals, as earlier stated, that it should provide for interaction among multiple users possibly separated by time and/or by space, and enable them to establish and maintain their shared context. Technologically motivated requirements are induced by current technology. We demand persistency and intensive handling of failures to avoid loss or inconsistency of data, and we are very concerned with the efficiency of the software. Organisatorical requirements deal with maintaining and controlling the distributed system; we must be able to prevent unauthorized access or illegal interaction among users. We should be able to dynamically reconfigure the hardware and to update the software.

There is not entirely a one-to-one correspondence between requirements and motivations. Selective transparency is functionally motivated by CSCW applications need to control the underlying system, or the need for exploiting the underlying hardware. Selective transparency is technologically motivated because current technology cannot optimally manage its resources, as humans (may) know how to do this better. Parallel applications are very concerned about the mapping of processes to minimize communication. Further, selective transparency may be motivated by management. Partitioning of responsibilities may require clear-cut responsibility domains, a need to know what information is located where and when, and accessed by whom.

Characteristically for sharing systems is the diversity; the applications are diverse, and so are their requirements. A support system suitable for sharing should address all of these. This makes it challenging to support sharing, but not we believe not impossible.

Until now we have not discussed MTS-Linda in the context of sharing. We need to determine if, and in that case, how MTS-Linda can be used for sharing, and potentially what MTS-Linda lacks.

## 4.3 MTS-Linda and Sharing

In MTS-Linda, we see a tuple space serving the role as metaphor for the shared context promoted by sharing systems. A tuple space groups related data and resources into sharable units.

Data sharing in MTS-Linda takes the form of tuples in tuple spaces. MTS-Linda can be thought as an in-core data-storage, where the tuples of a tuple space constitutes a data structure accessible and shared by a set of asynchronously evolving processes. Further, the MTS-Linda primitives guarantees mutual exclusion of access to tuples, giving the necessary means for coordinating sumoultanious access to shared data.

A shared context—a shared tuple space—is named and accessed via a tuple space capability. The prototype model provides shared tuple spaces which are created using a $newTS()$ operator returning a capability to the created tuple space. Capabilities may be used among the fields of tuples, and may hence be communicated to the processes (on behalf of the users) needing access to the shared data.

Moreover, the space and time decoupled communication of MTS-Linda gives natural support for the asynchronous distributed user interaction pattern found in the groupware time space matrix of Figure 3.3, i.e., communication taking place at different times and different places. Inserting a tuple in tuple space is like leaving a note for the receiver(s), an removing a tuple is like picking up the note. The direct support for this interaction pattern does not, however, mean that MTS-Linda does not support the 3 other interaction patterns, but that these must be programmed. Dependent on where the producers and consumers of the tuples are placed the tuple space may provide for communication at the same or different location, although MTS-Linda as presented in Chapter 2 gives no explicit means for expressing locality. Similarly, with communication at the same time, the tuples are only stored very shortly in tuple space, which then can be thought of as a message-buffer.

Based on these observations, we claim that MTS-Linda provides the sufficient primitives for supporting the basic functionality of a sharing system. It is possible via tuples and tuple space to establish and maintain a shared data space, and the MTS-Linda primitives allows expression of the communication patterns defined by the groupware time space matrix.

To illustrate MTS-Linda's abilities, and to justify the above claim further, we give in the following a few programming examples of solutions to some of the problems of sharing data through tuple spaces. One example builds a flexible lock for concurrency control, another illustrates how a hypertext data structure can be build using tuples, and the final example is concerned with creating a multi-user awareness view of the shared data.

### 4.3.1 Concurrency Control Example

MTS-Linda provides concurrency control by the atomic manipulation of tuples. Thus, if concurrency control is needed on the tuple level this is supported directly by MTS-Linda. Concurrency control at larger granularities are easily build using one or more simple tuples. An alternative is to manipulate a tuple space as a whole, which is also an atomic operation in MTS-Linda.

```
voin InitLock() {                               // create and initialize lock tuple
    out("lock", SHARED, 0);
}
void GetSharedLock() {
    in("lock", SHARED, ?cnt);                   // wait until lock is in shared state
    out("lock", SHARED, ++cnt);                 // one more shared lock holder
}
void GetExclusiveLock() {
    in("lock", SHARED, 0);                      // wait until lock is shared and 0 lock hold-
                                                  ers
    out("lock", EXCLUSIVE, 0);                  // change lock state to exclusive
}
void FreeLock() {
    in("lock", ?state, ?cnt);                   // fetch the lock tuple
    if(state==SHARED)
        out("lock", SHARED, --cnt);             // 1 less shared lock holder
    else                                        //it was an exclusive lock
        out("lock", SHARED, 0);                 // change state to shared and 0 lock hold-
                                                  ers
}
int TestLock() {
    rd("lock", ?state, ?cnt);                   // read the lock tuple
    if((state==SHARED)&&(cnt==0))               // the lock is free if state is shared
        return TRUE;                            // and no one holds the lock

    else return FALSE;
}
```

Figure 4.1: An implementation of a lock allowing both shared and exclusive access.

Figure 4.1 shows an implementation of a lock allowing both shared and exclusive access, and for lock-testing. The implementation uses a single $(”lock”,\ state,\ count)$ tuple where state is either $shared$ or $exclusive$, and $count$ is the number of shared-lock holders.

Nevertheless, much of the locking business can often be avoided in MTS-Linda by exploiting the flexibility of match, and by careful encoding of information into tuples representing data-portions at granularities manipulable in units. An advantage of the atomicity guaranteed by the MTS-Linda primitives is that they then automatically take care of mutual exclusion—holding a tuple implies that it may be updated without disturbance. Other accessors will automatically wait for the tuple to be returned. The necessary concurrency control is designed into the distributed data structure, and thus building and testing explicit lock tuples is uncommon.

In conclusion, MTS-Linda allows flexible forms of concurrency control, but this must be programmed explicitly.

### 4.3.2 Hypertext Example

In the following we give an example of how data can be represented and organized using MTS-Linda. We describe the data structures (tuple signatures) needed to express a simple hypertext structure in MTS-Linda. A hypertext system is a way to organize text, and is in[38] defined as

> “The concept of HyperText is quite simple: Windows on the screen are associated with objects in a database, and links are provided between these objects, both graphically (as labeled tokens) and in the database as pointers”

Thus, the documents of a hypertext system are organized in a graph structure where hyper-nodes contains the documents, and links express relations among the documents[1]. Further, hyper-nodes and links contain attributes, which are used to describe their name, kind, and purpose etc. A typical link attribute is the relation type describing how the link's source hyper-node and destination hyper-node are related, e.g., dependent-on, documented-by, specified-by. For example, a piece of source code can be represented in a hyper-node, and its documentation in another. A link between these nodes can express a documented-by relation.

Figure 4.2 shows an MTS-Linda implementation of a simple hypertext structure[2]. The hyper-nodes and the links attributes are represented as a shared tuple space. This enables more processes concurrently to access the contents of a hyper-node and the attributes of a link. The unidirectional links are represented as tuples with the following signature:

$$(”link”,\ TSCap\ to\_node,\ TSCap\ from\_node,\ TSCap\ link\_id)$$

[1]Hyper-systems where hyper-nodes contains data of any kind e.g., audio, graphics and text are called hypermedia.

[2]The concept of anchor points have been omitted. An anchor point is a link which points from a position within the data field of a hyper-node to another hyper-node. An anchor point is typically manifest at the screen-level as a highlighted word one can click. The destination hyper-node is then shown on the screen.

```
// Create a new node entity initialized with typical
// attrributes in a programming environment
TSCap CreateNode() {
    aNode = newTS();                                    //allocate the new entity
    aNode.out("title", node_title);                     //create and initialize at-
                                                          tributes
    aNode.out("language",programming_language);
    aNode.out("creator", creator);
    aNode.out("the_data", init_data);
    return aNode;                                       //return capability for the new node
}

TSCap CreateLink(TSCap to_node){                        //create a new initialized link en-
                                                          tity
      aLink = newTS();                                  //allocate the new entity
      aLink.out("Title", title);                        //create and initialize at-
                                                          tributes
      aLink.out("Relation Type",relation);              //The link's relation type
      hyper.out("link",to_node, no_ts, aLink);          //Insert new link with empty source node
      return aNode;                                     //return capability for the new link
}

void UseLink(TSCap from_node, link_id){                 //Create a reference for from_node
      hyper.in("link", ? to_node, no_ts, link_id);
      hyper.out("link", to_node, from_node, link_id);   //attatch link to from_node
}

void MoveLink(TSCap link_id, new_to_node){              //move link_id to new_to_node
      hyper.in("link", ? to_node, ? from_node, link_id);
      hyper.out("link", new_to_node, from_node, link_id); //point to the new node

}

void RemoveLink (TSCap from_node, to_link){             //remove a link
      hyper.in("link", ?to_node,from_node,to_link);
}
```

Figure 4.2: A simple hypertext implementation in MTS-Linda. For brevity a number of variable declarations has been omitted.

where $to\_node$ is the hyper-node the link points to, $from\_node$ is the hyper-node from which the link points, and $link\_id$ is a capability to the tuple space where the link's attributes are stored. The link tuples are stored in a hypertext tuple space named $hyper$.

There is one inconvenience with the current MTS-Linda primitives which often becomes evident when several tuples constitutes a dynamically growing and shrinking table, for example in order to follow all links from a hyper-node. We miss a way to iterate over a set of tuples which matches a given template. A solution is to maintain an extra tuple describing the count of tuples in the table. However, this is a bit cumbersome and degrades performance, because the counter tuple must be updated at each insert and remove operation. Programming of searching would be eased if an iterator primitive was available to the programmer.

This implementation assumes that tuple spaces are light-weight objects which can be used plentiful, and are inexpensive to manipulate. Tuple space capabilities are used as a simple way to group related tuples. The same functionality could be obtained by explicitly maintaining an integer which are to be used as a unique tag-field on all hyper-node attributes. However, we believe that it will be a general trend to use MTS-Linda's built-in modularization potentials in form of tuple spaces. An implementation of MTS-Linda should acknowledge this by providing light-weight tuple spaces.

In conclusion, by using MTS-Linda's tuples it is possible to set up even quite complex ways of organizing documents such as hypertext, but we miss an iterator primitive.

### 4.3.3 Event Management

The following describes a problem in relation to creating a multi-user awareness situation of updates of shared data. Users should be notified about interesting events generated by others. A related problem is to update users' view of a shared data space. For example, the problem is found in a hypertext system, where a browsing tool shows a part of the hypertext structure on a display. The browser should update its display incrementally as the underlying hypertext structure is changed.

A possible solution is event-management. When a tool alters the hypertext structure it generates a set of events which are propagated to the browsers. An example of interesting events is when a hyper-node is being locked or unlocked, and when hyper-nodes are added or removed. The browser can have a set of symbols on its display illustrating the locking state of a hyper-node. The browser alters the display in the order of which events are generated, i.e. causality must be preserved.

In general, a browser is interested in a set of events, and several browsers may be interested in the same events. Furthermore, a browser dynamically changes its view on which events is of interest. For instance if the browser's display is scrolled, and now shows a new part of the hyper structure, events of the old display is no longer of interest while the new become so. Furthermore, there may be several event types.

In the following we will call a process which generates an event an "event-producer", and a process which receives event notifications an "event-subscriber" (consumer). Each subscriber holds a queue of events in a tuple space denotable to by a tuple space capability. This could denote a user's workspace tuple space where e.g., a browser and an editor tools are running. Further, assume that each event is distinguishable by a kind (integer) and an object identifier (tuple space capability) pair. Assume further, that an event-subscriber-tuple exist for each (object, event) pair. The subscriber field is a set of capabilities to the tuple spaces where the subscribers event queue is stored:

$$(\text{"}event\text{"},\ TSCap\ object,\ int\ kind,\ TupleSpace\ subscribers)$$

When an event-subscriber gets or looses interest in an event it respectively executes the $subscribe()$ and $unsubscribe()$ procedures of Figure 4.3. When an event-producer has performed an action interesting to others it executes the $generate\_event()$ procedure of Figure 4.4. Delivering an event consists of reading the subscriber table and of inserting an event-tuple in each subscriber's event-queue.[3]

```
//This procedure is executed by subscriber's (with id new_subscriber) to subscribe to event.
void subscribe(TSCap new_subscriber, TSCap object, int kind) {
   TupleSpace subscribers;
   //fetch the corresponding event tuple, and store the subscriber set in a local tuple space
   in("event", object, kind, ? subscribers);
   subscribers.out(new_subscriber);              //add new subscriber to the "mailing list"
   out("event", object,kind,subscribers);        //insert the updated event tuple again.
}

//This procedure is executed by subscribers (with id old_subscriber) to unsubscribe event.
void unsubscribe(TSCap old_subscriber, TSCap object, int kind){
    TupleSpace subscribers;
    in("event", object, kind, ? subscribers);    //fetch subscriber table
    subscribers.in(old_subscriber);              //remove subscriber from "mailing" list.
    out("event", object, kind,subscribers);      //insert the updated event tuple again
}
```

Figure 4.3: Procedures to subscribe and unsubscribe events.

Passive field and local tuple space are here used as a convenient set-type, but an array of tuple space capabilities could have been used as well. For field-tuple-space to be competitive with traditional data structures, they should be manipulable with a speed comparable with manipulation of arrays

[3]There is another solution which avoids the reading of the subscriber table for each event generation. The solution caches the subscribers in the event generator. Cache modifications are initiated on basis of subscribe and unsubscribe events.

```
void generate_event(TSCap object, int kind)
{
    TupleSpace subscribers;
    TupleSpace empty={};                        // allocate new empty tuple space
    TSCap subscriber;                           // address of subscriber
    // read out a copy of the event tuple, and store subscriber "mailing list"
    // in a local tuple space.
    rd("event", object, kind, ?subscribers);
    // send the event to all subscribers for this event.
    // Assumes that subscriber tuple space is a homogeneous collection of TSCaps
    while(subscribers ≠empty)                   // propagate event to all subscribers
    {
        subscribers.in( ? subscriber);          // get new subscriber,
        QueueInsert(subscriber,kind);           // and store the event in his event queue
    }
}
```

Figure 4.4: A procedure which generates a new event and delivers it to all subscribers. The $QueueInsert()$ procedure is for brevity unspecified.

or lists. Further, passive local tuple spaces should be accessible with a performance comparable to access to a traditional linked list.

MTS-Linda does not provide any built-in facilities designed for event-management, but MTS-Linda's generality allows it to be programmed in an inexcessive way. A challenge when programming MTS-Linda is to find a clever coding of tuples which is flexible and uses few tuple space accesses.

By combining hypertext, locks, and event-management we have a good starting point for implementing a CSCW style shared data space. The given examples may seem very focused on implementing a hyper-text application. However, they are merely instances of a set of general problems encountered in sharing systems: Event management is a way to propagate the activities made by one user to the others in real-time to encourage a multi-user awareness situation, as is often desired in sharing systems. The hyper-text data structure is one possible way of establishing a shared context, and the lock is a typical problem encountered in maintaining a shared data space.

### 4.3.4 Deficiencies of MTS-Linda

Having argued that MTS-Linda supports the basic functionality of a sharing system, there is still a number of issues MTS-Linda, as presented hitherto, does not address:

- Protection beyond what is buried in holding capabilities should be added.
- MTS-Linda's means for encapsulation in a untrustworthy multi-user environment should be further examined.
- We have not justified that MTS-Linda can accommodate the flexibility requirements of sharing systems.
- It should be extended with selective transparency.
- We have no way of being assured that the tuples of a tuple space is kept on persistent storage.
- A model for handling of failures should be added (or this is done transparently by the system, which may not be entirely realistic).
- MTS-Linda possibly lacks support for heterogeneity.
- It may turn up that its performance is insufficient for sharing systems.

In the next chapter we discuss these deficiencies in context of MTS-Linda, and presents a set of solutions to some of the complaints in order to make MTS-Linda better qualified for supporting sharing systems.

## 4.4 Summary

This chapter has prescribed a list of requirements for a sharing support system. The overall functional requirement is that processes (on behalf of users) possibly separated by time and/or by space are able to create shared resources and data-entities, the so-called shared context, and to synchronize and communicate about the access to these.

High flexibility, reliability, and performance plays an important role, and protection mechanisms acknowledging sharing and cooperation are also necessary. Notably, the requirement of selective transparency was surprising. Visibility of multiple users and the distributed system is equally important as transparency. Sharing systems often require awareness of the presence of multiple users and of the cooperative working situation, and has an inherent need to control the underlying system.

Basically, MTS-Linda gives the desired functionality. However, there is a number of requirements MTS-Linda should be extended to handle. Most prevalently it lacks persistent storage, selective transparency, and encapsulation. We look at these in the next chapter.

# 5 Qualifying MTS-Linda

As argued in Chapter 3 MTS-Linda provides the basic functionality necessary to support sharing systems. However, Chapter 3 also found a number of issues MTS-Linda needs to address. This chapter is concerned with qualifying MTS-Linda to accommodate the ascertained requirements i.e., it instantiates the abstract model presented in Chapter 2 to a practical model suitable for the programming of sharing systems.

Our general solution is to modify the behavior of MTS-Linda on the tuple space level, since we regard this as the unit of modularization in MTS-Linda. We propose that MTS-Linda is extending with *special tuple spaces*. Special tuple spaces mostly have the same interface as the abstract tuple space presented in Chapter 2, but may at instantiation time be specified to have special functionality and performance characteristics e.g., be persistent or replicated. Such special tuple spaces have great resemblance with device independent I/O facilities found in other systems. One of our proposals for encapsulation even goes as far as allowing the programmers to specify interfaces for tuple space, effectively regarding these special tuple spaces as abstract data-types.

We attack the problems found regarding encapsulation, reprogramming, and protection of tuple space. These problems can be characterized as being organizatorically motivated, i.e., deals with management, control, ownership, and responsibility. Following this we direct our attention to some of the technologically motivated problems with failures, durability, and performance of tuple space. Furthermore, we attent to selective-transparency, and visibility of location and failures.

We end the chapter by outlining a layered system level model for realizing special tuple spaces, and hint how tuple spaces abstract interfaces may be interpreted in a MTS-Linda kernel.

## 5.1 Special Tuple Spaces

The tuple spaces as introduced in Chapter 2 are ideal: The operations on such never fail, they never loose their contents, they have infinite memory, and tuple space access and manipulation is infinitely fast. Unfortunately, the real world is far from being ideal. Both hardware and software may fail. Real life computers have finite memory, and communication takes a considerable amount of time.

An ideal tuple space is extremely difficult to implement. We are faced with a dilemma between performance and generality. The full generality may require use of persistent storage as well as redundancy techniques. These techniques are expensive in terms of processing, storage, and communication, and is thus compromising for performance. And the other way around: If a performance optimal solution is desired for a problem, this often requires exploration of the special cases and circumstances applicable to that specific problem. These special cases may not hold in the general case, making the solution generally inapplicable.

The solution is not a single tuple space constituting a compromise between the two, but rather a set of compromises where the actual choice depends on the situation. In some cases we are willing to pay with performance, simply because the generality is required or because performance is not critical. However, in most situations both performance and generality matters. We want to be able to select from a range of solutions spanning from high-performance to the full generality. The actual choice depends on the problem at hand.

It is our claim that the MTS-Linda model can be twisted to accommodate many of the practical needs of the real world by supplying special tuple spaces. In some cases we want a tuple space to be persistent, to have its tuples replicated, to behave like a queue or a byte-stream instead of as a multi-set, or to be accessed using an index instead of match.

Tuple spaces are the modularization mechanism in MTS-Linda programs, whereas active and passive tuples are seen as inhabitants of tuple spaces. Specialization at tuple space level, and not at tuple or process level, conforms with this idea.

The design room for specialized tuple spaces is illustrated in Figure 5.1. In the performance/generality plane we have plotted the positions of a set of different specialized tuple spaces. They spans a wide range: In one extreme we have a $LocalPassiveTupleSpace$ which is located within the address space of a single process. Access to this tuple space should be as fast as a access to an instance of a normal abstract data type. In the other extreme is a $RecoverableTupleSpace$. Almost everything is attempted to keep such a tuple space available. It may be replicated, and if the system crashes it recovers automatically. It may span many nodes and disks to exploit all the memory in the system to provide “infinite” memory. As noted, an ideal tuple space with infinite performance and generality may be unrealizable. The $StandardTupleSpace$ is the “usual” implementation of tuple space; its contents are kept in-core, little automatic fault handling is performed, and it is reasonable fast.

We emphasize that the points are rough estimates. The figure just states that, for example, a $CompiledTupleSpace$ is likely to be faster than a standard tuple space, because its runtime representation is determined at compile time, and therefore can be tailored to a given use at prehand. As another example, a $LocalPassiveTupleSpace$ is likely to be the fastest implementable tuple space, because it is implementable as a traditional data-type within the address space of a process. It should also be intuitively clear that persistent tuple spaces are considerable slower than the standard in-core tuple spaces. Note also that the replicated tuple space is plotted 2 times.

This is not a mistake, but the performance of a $ReplicatedTupleSpace$ depends of its use. If more **rd**-operations takes place than updates, the **rd**-operations can use the local replica, making a replicated tuple space faster on the average than a standard tuple space.

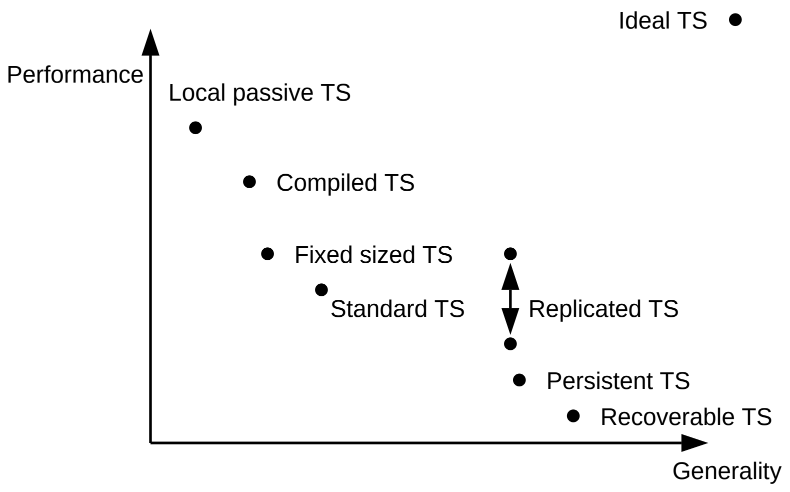

Figure 5.1: The design space for specialized tuple spaces. $TS$ is a short for tuple space.

A tuple space can be regarded as an abstract data-type with a private internal state (tuples) and a set of public operations (MTS-Linda primitives) to manipulate the state. It may be a bit peculiar data type though, one could argue, as it may exhibit active behavior[1]. A tuple space is a concurrent abstract data type, with concurrent internal behavior, and where several methods can be invoked concurrently. A declaration of the tuple space interface using C++ syntax is given in Figure 5.2. Following this observation the envisioned special tuple spaces are just versions of the abstract tuple space data type.

Special tuple spaces appears to the programmers as data types with different names e.g., $ReplicatedTupleSpace$ or $QueuedTupleSpace$. The properties of the tuple space is thus specified at creation time.

As long as the interface to the tuple space remains to be the MTS-Linda primitives, and only the implementation is changed, the application will not know if the tuple space is replicated or persistent etc. We find this property attractive because the application can be written in a transparent style, and the programmer needs only to be concerned with special tuple spaces at configuration (instantiation) time.

Such special tuple spaces bear great resemblance with the idea of device independent input/output known from operating systems. For example, the UNIX file descriptors are abstractions of both disk-files, pipes, and sockets. Once the file is created to be either of these, it can be manipulated

[1]Unlike many other object based languages as Emerald, or Actors, a MTS-Linda object may have many threads per object. A MTS-Linda tuple space encompasses entire, potentially recursively defined computations, and further MTS-Linda objects are manipulable as wholes. MTS-Linda objects are distributed objects.

```
class TupleSpace
{
      void out(tuple);
      void eval(tuple);
      void in(tuple *t);
      void rd(tuple *t);
      void = (TupleSpace);
      int == (TupleSpace);
      void contents_copy(TupleSpace);
      void contents_move(TupleSpace);
};
```

Figure 5.2: C++ like interface specification of a tuple space.

using the same `read` and `write` calls. This implies that a program can be written in largely the same way independent of the actual device it uses.

Special tuple space decouples the functional specification of a problem from the technical requirements and conditions under which the program must execute, allowing the programmer to attack these almost as an afterthought! This makes programming of distributed programs easier.

We want a tuple space to be persistent to allow for permanent storage of tuples. This implies that the file-system can replaced by a persistent tuple spaces, giving one less coordination concept, and getting closer to the idea of unified memory. Further, we could imagine replicated or recoverable tuple spaces which are able to survive failures, or a tuple space whose operations may fail or timeout in case of failure, thus allowing the programmer to handle errors. We discuss tuple spaces with different durability characteristics in Section 5.3.

We might also find a compiled tuple space useful. A compiled tuple space is one which potential contents (tuple signatures) is known at compile time. This allows a compiler to use the partial evaluation optimization techniques know from Yale implementations of Linda, giving a more efficient runtime representation of tuple space. The cost of compilation is less flexibility—the tuple space is "closed" for run-time addition of tuples with signatures different from those found at compile time. Performance of tuple space is discussed in Section 2.3.2.

We have, as result of a brain-storm, envisioned a bunch of interesting special tuple spaces. We envision tuple spaces with different priorities, or tuple spaces which can meet real-time constraints; the list seems endless. This motivates our demand for a flexible system where users can add their own special tuple spaces. A design open for such extensions will accommodate the user's needs, allow diverse applications, and will allow the system to evolve, resulting in greater flexibility. Section 5.5.1 discuss a layered model for how this form of flexibility can be accomplished.

Pursuing abstract data types, it is attractive to define a new interface to a tuple space, i.e., replace the MTS-Linda primitives with a new set of user-specified operations. The purpose is to give the programmer the ability to define other concurrent objects as abstractions of tuple space. He can define a distributed queue as a “tuple space” with $enqueue$ and $dequeue$, operations. And as we shall see in Section 5.5.2 giving the runtime system knowledge about interfaces allows the runtime system to enforce them.

Introduction of special tuple space also raises new semantic problems. One is that the should be a way of reporting errors if the programmer tries to use an operation not applicable to a special tuple space. For example trying to perform an **eval** on a $LocalPassiveTupleSpace$ should result in an error.

Another problem is the semantics of tuple space hierarchies. Should a local tuple space inherit the properties of the lower level tuple space to obey the bureaucracy of the tuple space hierarchy. For example, should the node subset of a local tuple space be a subset of the node subset of the owner process’ context tuple space? –And if a tuple space should be recoverable, should tuple spaces at lower levels in the hierarchy also be required to be recoverable? –Local tuple spaces are part of a lower level’s state, and if the local tuple space is lost so is that state, hence the lower level’s tuple space cannot be recovered. The situation is analogous to a hierarchal file system where sub-directories are located at the same disk as the parent directory.

Another interpretation is that the special properties only applies at a single tuple space. This gives a more flexible use of special tuple spaces. Local tuple spaces may for example have a node sub-set independent of that of the lower level tuple space. This may be useful to hide that a process owns a local tuple space holding a parallel computation. If the process is evaluated in a tuple space with a located at a single node the computation contained in the local tuple space will still be performed in parallel. We leave it as an open question, what the semantics should be in this case.

### Summary

We have found that the ideal tuple space, cannot be implemented. We have to compromise between the performance of and the generality of a tuple space. Different situations requires different compromises between the two, ranging from high-performance low-generality, to low-performance high-generality.

By introducing special tuple spaces and having several tuple spaces all with the same interface, we have decoupled the functional specification of a problem from the technical requirements. This we allows a programmer to put his mind on the problem and first afterwards choose what special tuple space he needs. Thus making programming easier.

Special tuple spaces can be taken a step further, allowing special interfaces to be declared, we look how this and the traditional MTS-Linda constructs can be used for encapsulation.

## 5.2 Module Separation

This section is concerned with restricting access to shared modules. In general, the modules of a sharing systems cannot trust each other, so the offering of many shared modules are dependent on facilities which guarantees against misuse. We discuss the problems with, and solutions for encapsulation and protection in MTS-Linda. We also discuss a problem related to encapsulation: dynamically extension and replacement of program modules.

### 5.2.1 Encapsulation in MTS-Linda

It is a requirement for a sharing support system that the programmer or owner is able to restrict the access to the internals of a module, and the ability to modify it through a specified interface. The predominant reason for this is the lack of trust found in even small sharing systems (beyond the family model, c.f. [75]). For example, without these restrictions the color-laser-printer is likely to be misused, and users may insert themselves in front of the printer queue etc. The owner of the printer-module want to make certain that the users of the printer is accounted for their usage. Another example is a multi-person-calendar. Some persons believing that they are funny will undoubtly try to modify the entries of others, if possible. In these systems the person making the entry should be registered along with it.

Hence, there is a for need encapsulation. Encapsulated modules, as known from sequential programming languages, consist of an interface part and an implementation part[63]. The interface part defines the information about a module that must be known to the user. The implementation part contains the actual variable declarations and code to realize the module. This allows the implementation of a module to be *encapsulated* and hidden from the user of the module. This is captured in Parna's information hiding principles[63]:

1. One must provide the intended user with all the information needed to use the module correctly and nothing more.
2. One must provide the implementor with all the information needed to complete the module and nothing more.

Reprogramming and fault resiliency are also reasons for using encapsulated modules. Encapsulation makes it possible to alter (*reprogram*) the implementation of a module without affecting the use of the module. A single interface may have several implementations, e.g., a stack can be implemented as an array, or a chained list. As long as the new interface contains the old one as a "sub-set" the module can be replaced[2]. Fault resiliency is increased by encapsulation, because it clearly localizes errors to the errornous module.

MTS-Linda gives tuple spaces as the means for grouping related data and activity in the form of respectively passive and active tuples. However, MTS-Linda gives no means of specifying and enforcing special interfaces on tuple spaces, e.g., an interface ensuring that only the specified

[2]This corresponds closely to type conformity on which the Emarald abstract type system is build[15]

queue and dequeue operations can insert and remove elements from the stack. Encapsulation requires that we are able to specify and implement the methods of an interface.

We see 2 ways to encapsulate shared data structures and behavior in MTS-Linda:

- Use an appropriate host-language constructs to build a library to encapsulate the tuple space implementation. For instance, C++ classes can be used.
- Use a process (server) which implements the set of allowed operations, and use the **in** and **out** primitives to set up a message-passing like communication with the server.

Further, we have the idea that special tuple spaces, as presented below can be used for encapsulation:

- Use a special tuple space to define an interface and implementation for access to a tuple space.

We use an example distributed queue allowing for concurrent access to illustrate the different techniques of encapsulation. Implementing the enqueue method of a distributed queue data-structure in MTS-Linda requires 3 operations, retrieval of the tail pointer which then acts as mutex granting entity, insertion of the data element at the tail position, and storage of the new tail allowing another enqueue to get mutex on the tail of the queue.

### Host Language Based Encapsulation

The queue operations can be encapsulated within a host-language construct, e.g., as queue and enqueue methods on a queue-class in C++. However, this is an unsatisfactory solution, because in only gives weak encapsulation of a tuple space—the interface cannot be enforced. Users operating on the (printer) queue using the C++ class would manipulate the queue as intended. But any users holding a capability for the queue tuple space can by-pass the interface by using the MTS-Linda primitives to havoc the tuple space contents. A capability is like a C++-pointer to a piece of memory (tuple space). This permits the programmer to manipulate the pointer address without respect of module boundaries.

### Server Based Encapsulation

The traditional solution to encapsulation, as used in distributed operating systems, is to use a server processes which is programmed to accept only those requests defined by its interface. This approach gives strong encapsulation, because it cannot be by-passed by mischievous users (assuming that there is protection on the memory used by the server).

A similar approach can be used in MTS-Linda, using an MTS-Linda process to implement the interface of the tuple space. The process acts like a server, which accepts request-tuples, and either processes them itself or forwards them to internal processes. For example, the queue is held in a local tuple space, to which only the server process have access. The context tuple space of the server process acts like a mail-box of requests. These processes may be characterized as being functionally specialized, and communication with a strongly encapsulated module is similar to client/server programming.

The storage capability of tuple space allows for automatic buffering of concurrent requests, not giving the traditional protocols problems which may fail when concurrent access gets to hectic.

However, there are more problems in making this client/server relationship work. A strongly encapsulated module usually provides more than one function. A printer server may have functions for spooling a printer job, for showing the queue of printer jobs, and for removing a printer job. It must be possible for a client to specify which function it wants, and it must be possible for a server to arbitrate between its functions. Further, each of the functions take different parameters. However, the MTS-Linda primitives does allow this to be modeled conveniently. A process can only **in** (search with) a single tuple template at a time. We seek something with the same functionality as the guarded commands found in CSP[43], which allows for arbitration across a set of input channels. Indeed, one proposal adding arbitration to MTS-Linda exists[41]. It is possible to specify several templates of an **in**-operation, but only one is selected, and the selection is non-deterministic.

However, there are also programming techniques to work-around the problem of arbitration:

**Union field type.** One solution is to use a tuple with 2 fields; one of an enumerate type to indicate which service is requested, and another of union type to contain the servers parameters. The disadvantages of this is that the union causes unnecessary overhead if the union-members has different size. Further, it implies weak typing—a piece of memory is interpreted according to the selected tag field in the union. This is bad programming practice.

**Two tuples.** Another solution is to use two tuples per request, one with an enumerate type to indicate the requested service, and another with the parameters. Because the server knows the parameter tuples signature for each kind of service it does not need the union. However, there are two problem with this solution. Using two tuples—one to identify the service, and one with its parameters—suffices from the overhead of creating and consuming two tuples. Second, if the client deliberately or unintentionally forgets to deposit the parameter tuple, the server will block (infinitely) for that tuple (assuming inp and time-outs of tuple operations is disallowed). In general, the server cannot trust its clients to deliver both tuples.

**Task cluster structure.** The final solution we discuss here uses a server with a task cluster structure. One process exists for each service. Each process blocks waiting for a tuple with an appropriate parameter-list. Data shared among the services are placed in a tuple space local to the server process. In a printer example, the shared data space contains the queue, and the lpr, lprm and lpq services-processes. This constellation is illustrated in Figure 5.3.

The disadvantage is that if processes are heavyweight, or if the manipulation of the state in the

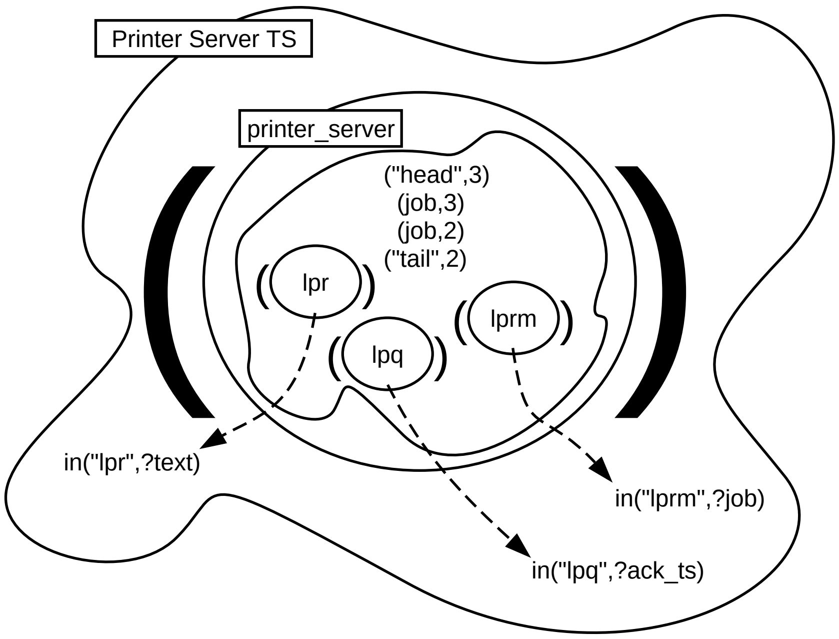

Figure 5.3: Illustration of an MTS-Linda server (strong encapsulated module) with a cluster task structure. The dashed lines indicates that the process associated with the line is searching for a tuple of with template indicated at the arrow head.

services tuple space is intensive a process per service can give an expensive server implementation. The potentially degree of parallelism is higher in the cluster solution than a single server solution.

The conclusion of the above discussion is that it definitely is possible to encapsulate data-structures within server processes in MTS-Linda. However, we have 2 complaints about the solutions. Arbitration is a bit cumbersome to program in MTS-Linda, and encapsulation in processes do not follow our general idea that a tuple space should be the unit of modularization. Moreover, client/server programming seems intuitively “un-Linda”-like. This has motivated us for the proposal for tuple space level encapsulation presented below.

In conclusion, MTS-Linda supports both strong and weak encapsulation. It provides for flexible, yet easy to use, synchronization patterns among clients and servers. However, Linda has problems arbitrating between services, though work-arounds exists.

### Special Tuple Space for Encapsulation

This proposal builds on the view that tuple space is an abstraction over data and activity with the MTS-Linda primitives as accessor methods. We extend this idea by allowing user supplied interfaces to tuple spaces. The purpose is to give the programmer the ability to define other concurrent objects as abstractions of tuple space. The proposal requires addition of an interface specification language for tuple spaces, and modification of the runtime system to enforce these interfaces upon usage resulting in strong encapsulation.

Figure 5.4 shows the specification and implementation of a special tuple space implementing a distributed queue. The tuple space is specified to have 2 operations, one for enqueuing and one for dequeuing. The specification in C++ notion but this is only for illustration of the idea. We do not have a finished and formally defined proposal for the interface specification language. For brevity both the interface and its implementation is specified in the same figure.

```
class Queue:TupleSpace              // spec. of a dist. queue based on a Tu-
                                     pleSpace
{
    enqueue(integer item) {         // the enqueue method
        in("tail",?cnt);            // insert at tail
        out("item",++cnt,item);
        out("tail",cnt);
    }
    integer dequeue(){              // the enqueue method
        in("head",?cnt);            // remove at head
        in("item",cnt,?item);
        out("count",--cnt);
        return item;
    }
}
```

Figure 5.4: Interface specification of distributed queue encapsulated in a tuple. The Queue is based on a standard distributed $TupleSpace$. We stress that this illustrates the intuition behind the specification of a special tuple space using C++ syntax.

The philosophy behind special tuple spaces with user supplied interfaces is to use the memory structuring mechanisms, known from sequential imperative or object based languages to structure the distributed shared memory of MTS-Linda. A possible source for inspiration for a design of the interface specification language is Agora[12], which also provides a shared memory abstraction, and which allows the programmers to encapsulate memory into abstract data-types.

Note that even though the invocation of an operation may syntactically look like a remote procedure call there is substantial difference. Special tuple spaces is a shared memory abstraction, and flow of control logically stays with the process performing the operation, as a local activation. In a remote procedure call, on the other hand, flow of control is transferred to the server when an operation is invoked. An advantage of special tuple spaces is that, if it encapsulates data, these may be accessed in a decentralized manner, and thus avoiding the bottle-neck of encapsulation in a server process.

Nevertheless, we have a set of worries which makes us unkeen on recommending this abstraction-of-tuple-space approach:

- Unlike most of the other special tuple space we will discuss, these are *non-trivial extensions* to the multiple tuple space model given in Chapter 2. An interface specification language must be added.
- Handling of the *name space* of special tuple space may in a untrustworthy multi-user environment can be difficult.
- Should the interface specification language used by MTS-Linda and the host language be the same? –Or should the MTS-Linda specification language be orthogonal to that of the any computation language? –Further, the problems of embedding MTS-Linda will be greater, as MTS-Linda now may supply a new type system based on tuple space types.
- What about perception of equality between tuple spaces? –Should this as now be on tuple basis (structural equivalence) or should the name of the special tuple space be taken into account?
- How are tuple spaces to be related through inheritance? –And how does special tuple spaces match?

We leave these abstract tuple spaces with enforceable interfaces as an interesting idea for further thoughts, and settle on server-based encapsulation in current MTS-Linda.

**Encapsulation Summary**

Although client/server-like interaction has a number of good points in distributed systems generally, we deviate from the idea that this should be the only supported model of interaction. As Peter Wegner in [88] notes:

> "Detailed analysis of a program finding primes suggest that, at least for this example, agenda parallelism with distributed (shared) data structures is preferred over result-based or message-passing approaches. If widely true, this suggests that the object-based message paradigm for concurrency is too restrictive in that it binds data too closely to its operations."

Ideally, the problem at hand should determine how the shared data should be accessed, and the programming model should support both strongly encapsulated entities, and entities which are "freely" accessible. MTS-Linda has both possibilities and allows the programmer to choose.

To give a rule of thumb of when to use strong encapsulation, it seems as the encapsulation through host language constructions is sufficient within a single application (intra-application communication) or between closely related processes. Strong encapsulation through servers or special tuple spaces is appropriate for interaction among applications, and to encapsulate system services. In terms of the taxonomy for programming environments, presented in Section 3.2, weak encapsulation is only sufficient in the individual and the family model, whereas and strong encapsulation becomes necessary along with the city and state model.

### 5.2.2 Reprogramming in MTS-Linda

A requirement for a sharing support system is the ability to reprogram components. It must be possible to change the implementation of an module to account for new and better implementation strategies, e.g., going from a linear search algorithm to a binary search algorithm. A sharing system cannot be terminated and then restarted each time a slight modification is made to a component.

Changing the implementation of a shared module can be quite difficult. To change the implementation it is necessary to make sure that any process accessing the data will use the new improved implementation, as using the old may result in havoc, or in best case result in errors.

Using a library of accessor routines statically linked with the processes using a module results in poor reprogrammability. Every such process must be tracked down, stopped, and recompiled. At least, they should receive an error indicating that they are using an obsolete library.

The are 2 conditions that may ease reprogramming: dynamic binding of names (or dynamic linking) and encapsulation. Encapsulation assures that it is easy to localize the entity—its code and its data—that needs reprogramming. Further, encapsulation separates the implementation from the interface, and thus separates those entities using the modules from those implementing it. Dynamic binding re-binds the old names dynamically to the new implementation of the module. Replacement of a module can thus take place without terminating the employers of the module. They only experiences a slight delay while the reprogramming takes place.

Using servers as means of encapsulation significantly increases the possibility for reprogrammability. A state-less server can easily be removed, and a new one can replace it without the client-program ever noticing the server change, as long as the new server implements the same interface as the old. If a server is state-full, that is the server have information about “connections” with clients, some communication will need to take place between the old and the new before the change can take place. Any state information held by the old server must be transferred to the new server for it to operate correctly, i.e., state-full servers must be programmed with reprogramming in mind, they must include a method which allows them to pass on the state information.

The prototype implements “open” tuple spaces, i.e., tuple spaces which are not born with given contents in mind. The internal runtime representation of these tuple spaces are prepared for receiving tuples of any possible signature. This flexibility aids reprogramming as the contents of a tuple space can by dynamically be modified. It is possible to insert (**eval**) new processes not programmed when the tuple space was created, thus allowing for, e.g., new serves to be added in a tuple space created before the servers.

### 5.2.3 Protection in MTS-Linda

Encapsulating is not always sufficient, in an untrusty system we also needs to ensure certain processes are restricted in the manipulation of a tuple space's contents, while others are not. What we need is protection of tuple space access. MTS-Linda is not much different from other distributed systems when it comes to the solution of protection issues: the well known solutions apply easily in the MTS-Linda context, we therefor only takes a brief look at protection.

Protection mechanisms are required at all levels where a user can access a system. Our main focus is on protection on the user level, i.e., protection of the tuples and tuple spaces in MTS-Linda. Protection against network hackers and the like is assumed to be handled in an implementation. The degree of low-level protection is of cause depends on the needed security, i.e., the context in which the MTS-Linda system is placed.

We see protection as the combination of encapsulation and access rights. Encapsulation define what methods are allowed on an object, and access rights define which domains[3] may access the methods. MTS-Linda already have protection on tuple spaces, as access requires a correct capability. This coarse grained access restriction is often insufficient and does not define access rights on the individual tuple space methods. It must be possible to declare a tuple space as e.g., **rd**-only, when accessed with a given capability.

Further, it is advantageous to have protection on tuple signatures, i.e., only allowing tuples with a signature to be manipulated. This allows, e.g., a server process to reside in the same tuple space as it uses to communicate with the clients. Furthermore, it makes it possible to ensure that only the server process can remove tuples requesting some operation, dissalowing a malicious process to steal tuples from the server. Special tuple spaces completely removes this problem as the interface defines what is allowed on tuple space. We look no further into this protection possibility.

There are two main strategies for realizing protection rights. These are capability or access control based [70]. Both strategies can be used in MTS-Linda, but considering the already capability based access of tuple space, it is most appropriate to include rights as part of the tuple space capability.

The rights of a tuple space capability can be changed by the holder, allowing a degraded capability to be passed on. To implement protection in MTS-Linda requires, addition of a rights field to the tuple space capability and methods to change the rights of the capability. To enforce protection a capability must validate before the operation in question is performed. The validation process must, beside checking the rights, also assure that forged capabilities are rejected; [70] poses strategies for achieving this. Figure 5.5 shows the rights of a MTS-Linda tuple space, allowing access to be restricted in each MTS-Linda operation which apply to a tuple space.

The envisioned use of capabilities to implement protection, is inspired by that used in the server based Amoeba operating system, where capabilities is used to implement protection of servers and the data they manage.

[3] A domain is a group of, e.g., users, processes, or objects, which may access an object in certain ways.

| | | | | | | |
|---|---|---|---|---|---|---|
| In | Rd | Out | Eval | CCS | CMS | CXD |

Figure 5.5: The rights filed of a MTS-Linda tuple space capability. Each right denotes one of the operations possible on a tuple space. CCS: Contents Copy Source, CMS: Contents Move Source, and CXD: Contents copy or move Destination

## 5.3 Durability

The use of tuple spaces in sharing systems requires that their contents can be preserved in spite of failures or power-downs, and it is significant that they are available, running, and consistent. The users of a sharing system expect things to work; otherwise they will become impatient and unhappy, and it is totally unacceptable if they loose hours or even minutes of work due to unsaved work or inconsistencies. The potential of higher reliability was also stated as one of the most important reasons for using a distributed programming environment. These requirements motivates us to propose tuple spaces with different durability characteristics: Tuple spaces which are persistent, recoverable, and/or replicated.

### 5.3.1 Persistent Tuple Space

The simplest way of introducing persistency in MTS-Linda is by a $PersistentTupleSpace$. This is one whose passive tuples are stored on a disk instead of in-core, as is the case for the standard tuple space[4]. Active tuples are either disallowed or transient in this simple proposal. A persistent tuple space is very similar to a traditional file, except that its contents are structured into tuples, and that these tuples are accessed associatively instead of in stream-order. The atomicity of the MTS-Linda is still guaranteed on a $PersistentTupleSpace$ so sharing can be safe.

Another simple solution would be to supply a checkpoint method on tuple spaces. When a process has performed a set of changes it wishes to be permanent it calls the checkpoint method. Alternatively, this could be done regularly by the runtime-system. The latest checkpoint is used to re-establish the situation as before the crash.

### 5.3.2 Transaction Tuple Space

The problem of maintaining data online and consistent in spite of failures is usually solved using atomic transactions. This is the approach used in database management systems, and e.g. in the distributed programming language Argus[60].

We know of one proposal for a persistent Linda system: [2] suggests a database like atomic

[4]Assuming disks have names (capability to a system tuple space) like nodes, another possible way of specifying a tuple space to be persistent would to use the location visibility primitives given in Section 5.4.1. A persistent tuple space is simply a tuple space located on a disk!

transactions as an extension to the existing Linda primitives. An atomic transaction is a set of tuple operations which looks like they were executed as an indivisible action. That is, either all of the actions associated with the transaction are executed to completion or none at all are performed.

For recovery purposes the system maintains a log on stable storage, i.e., writing is atomic and its contents outlives failures. The entries in the log describes the tuple-operations which are performed or about to be performed. The log is used in a recovery situation to redo the effect of committed transactions and undo the effect of aborted transactions. If the tuple spaces are distributed it is necessary to set up a voting scheme, such as the two phase commit protocol, to decide whether the transaction should be committed or aborted. The transaction is committed if all participants (nodes) are willing to guarantee the execution of their part of transaction) or aborted if one participant cannot make this promise. Abortion may take place either due to failure or due to concurrency control.

Atomic transactions have serializability semantics, and can in some situations alleviate the programming task of maintaining e.g., explicit locks.

In [2] the authors provides 2 ways of performing new transaction, one based on a process constituting a transaction and the other based on the beginning and end of a transaction being explicit stated. The `xeval` operation adds an active tuple to the tuple space. All tuple-operations performed by active processes within the tuple is part of the transaction. The transaction is completed (committed) when all processes in the tuple have become passive. The second way of performing a transaction is by explicitly start and stop. `xstart` denotes the start of a new transaction, tuple operations performed hereafter is part of the transaction. The transaction is committed or aborted using `xcommit` and `xabort` respectively.

We have 2 complaints about this solution. It adds new primitives to the language, and thus compromises our idea about "device independence". Second, due to uniformity between active and passive tuples it should be possible to store a computation in a persistent tuple space an later resume it. This is an additional requirement for persistence in MTS-Linda is that the active tuples (processes) belonging to a tuple space also should be subject for persistency[5].

### 5.3.3 Recoverable Tuple Spaces

An alternative to the atomic transaction approach to persistent tuple space is a recoverable tuple space. A recoverable tuple space is one whose contents after a crash automatically is brought up to the same state as just before the crash. Again, we do not wish to distinguish between active and passive tuples. The philosophy of MTS-Linda is to give active tuples first class rights. A traditional database recovery scheme can be used to recover the passive tuples; each MTS-Linda operation is like a small atomic transaction.

To recover processes, we propose a technique known from replaying of processes in distributed debugging, [27]. All communications in form of tuple operations (message contents) done by

[5]A possible third complaint is that their solution implies nested transactions and recovery across **eval**-operations, which intuitively sounds like hairy business.

the process are written to a log kept on stable storage. If the process performs other system calls, they too are logged. Further, processes are regularly check-pointed to stable storage, to inhibit large logs, and to enable fast recovery. Note that a log can be implemented using a $PersistentTupleSpace$, and that the MTS-Linda primitives manipulating processes can, with slight modification, be used to implement process checkpointing. The runtime system is thus likely to contain many of the necessary functions needed anyway.

The recovery (replay) procedure goes as follows: The latest check-point is replayed with the contents of the log as input, instead of performing the prescribed communication. Output from the process is discharged under the recovery procedure. When the log is empty the process is in the same state as before the crash, and it is ready to continue where it left off[6].

There are 2 basic assumptions for this scheme to work: A partial ordering of tuple space operations (or deterministic behavior of tuple space) must exist to recover the tuple space. Replaying processes requires that the process responds to the same input with the same output. That is processes which behavior is dependent on, e.g., hardware clocks, or some means of random behavior cannot in general be replayed and thus not recovered. Hence, recoverable tuple spaces is an alternative to atomic transactions to obtain reliable persistency[7], without sacrificing processes as first class objects, and without needing to introduce new linguistic means to express recovery procedures.

### 5.3.4 Replicated Tuple Spaces

Replication (to achieve redundancy) is a general technique for increasing availablility of a system function. Xu [91] has proposed a replication scheme for making a tuple space highly available in spite of failures. His scheme accomplishes consistent tuple space updates, transparent replication, toleration of simultaneous failures, progress as long as a majority of the replica are able to communicate, and, he claims, the scheme imposes little delay on the user program (much processing can be done by background processes).

However, [91] only replicates passive tuples, whereas we see processes as fields of active tuples, and thus these should be replicated equally to passive tuples. Replicating of processes is also necessary to write highly available services. An existing system which supports this is ISIS[10]. ISIS allows for formulation of fault-tolerant resilient process groups. These consists of a collection of processes that are cooperating to perform a distributed computation, and interacting using the ISIS communication protocols providing message passing and causally ordered multicasting.

Replicating passive tuples solely is only half the solution. The other half, replication of processes, can be done by using inactive backup-processes, [5]. A replicated process is realized by having a single primary process, and a number of backup processes shadowing the primary process's actions. That is, messages sent to the primary process are also sent to the backups. And the backup processes also closely watches the messages produced by the primary process. In MTS-Linda terms a message corresponds to a tuple or a tuple space-operation.

[6]This may not be as easy in practice as it is stated here!

[7]This is under the assumption the only the processes residing in that tuple space is manipulating it—the state kept in tuple space is then always preserved, either as a tuple in tuple space or as state associated with a process.

Furthermore, the entire state of the primary process is periodically check-pointed to the backup processes to discharge message logs, and to ensure that the backups and the primary agree on what is the actual state. This also shortens recovery time. In this way the backups are able to mimic the behavior of the primary process, either continuous or in a recovery procedure. When they discover crash of the primary process one is elected to overtake the primary process' place.

By combining replication of passive tuples and backing up processes we believe that it is possible to provide replicated tuple spaces which in a convenient manner allows for writing highly available programs in MTS-Linda.

### 5.3.5 Summary

It is our claim that in principle failing primitives, persistency, and location visibility is sufficient for handling persistency and failures. However, making data and services available and consistent is not a simple task. It requires advanced algorithms and protocols unfamiliar to the average programmer. Therefore, due to the importance of reliability in sharing systems, we are convinced that a sharing support system should offer a transparent, high-level abstraction which hides the technical and difficult problems of handling failure, and allows the programmers to develop their programs peacefully! The special tuple spaces transaction, recoverable, and replicated described above gives exactly this high level abstraction.

The problems of reliability is in MTS-Linda similar to reliability in other systems. Known techniques are applicable, and, as far we as can see, MTS-Linda does not supply new solutions, but allows us to hide away ugly details in special tuple spaces. As indicated, the design space offers several possibilities for how this can be done in MTS-Linda.

Both atomic transactions, highly replicated, or recoverable tuple spaces are costly in terms of performance and implementation. To the common sharing application they seem like an over kill. The simple $PersistentTupleSpace$ will in our judgment often suffice. Thus we conclude that MTS-Linda can be tuned to meet the persistency and reliability requirements enlisted in Section 4.2.

The construction of these durable tuple space relies on the ability to map tuple spaces to the underlying architecture, and the ability to detect failure of MTS-Linda operations. In the next section we look at these issues in terms of non-transparency.

## 5.4 Non-transparency in MTS-Linda

We found that MTS-Linda gives transparency in all issues of distribution except parallelism. Parallelism must be created and managed explicitly by the programmer. MTS-Linda gives concurrency transparency on manipulation of tuples and tuple spaces as wholes as the MTS-Linda operations are atomic. The remaining distribution issues, location, failure, migration, replication, concurrency, and heterogeneity, are not specified as part of the MTS-Linda semantics; MTS-Linda does not demand that, e.g., tuples must be replicated to enhance fault tolerance, or e.g. tuples having an integer as first field must be located at node 0. Omitting these distribution issues, implies not giving any handles to control their possible behavior.

In the previous chapter we found that distribution transparency must be the default in sharing systems. But we also noted that it must be possible to select visibility when required. We see that MTS-Linda gives transparency as default, but does not have any means of expressing visibility. We need visibility to satisfy the needs for awareness of concurrent updates of shared data, performance requirements allowing a user to map tuples optimally, and residual requirements, allowing usage of special hardware components. Allowing for visibility in the MTS-Linda model we need means of manipulating (handles) these visibility concepts in the host-language.

To find out what distribution issues we need handles for, we take a look at their relations. We claim that location and failure visibility are the essential distribution issues, which when introduced in a host-language also gives us handles for migration and replication visibility using the MTS-Linda operations. Migration visibility is based on location visibility: to migrate a tuple space explicitly we need only move an active tuple or a tuple space from one node to another, e.g., by using a multi-set operation. In effect replication is concerned with having a tuple space present at different nodes. Location visibility allows us to see what nodes a tuple space in replicated to, and a list of tuple space capabilities represents each tuple space which is part of the replicated tuple space.

Our system model, Section 3.1.1, showed that a distributed systems in general is heterogeneous. MTS-Linda does not specify how a possible heterogeneous hardware used as implementation platform is visible to a programmer. We do not intent to introduce heterogeneity visibility in MTS-Linda. First of all are there several complications in exchanging information between different hardware components. There exists solutions to the problems of exchanging data and code [82], they are however, costly in terms run-time of packing and unpacking of data. We are not going to go into any further details with heterogeneity in MTS-Linda. As we need to have design and implementations for homogeneous systems before it is feasible to consider MTS-Linda in a heterogeneous system.

The last distribution issue is concurrency control. It must be possible to explicitly perform concurrency control on shared data. The atomic MTS-Linda operations ensure mutual-exclusion on data stored as part of an tuple, thus giving transparent concurrency control. Concurrency visibility is also possible be having tuples representing e.g. a lock, as already shown in Section 4.3.1, thus also allowing for concurrency visibility. The difference of having transparent or visible concurrency control lie in different programming styles[8].

We have found that location and failure are the two essential distribution issues, which are not introduced as visible components in MTS-Linda. They allow us, together with the MTS-Linda operations, to make the other distribution issues visible in the host-language. Next we are concerned with the introduction the essential tuple space visibility features, location and failure, as a component of a host-language.

[8]Placing the data as part of the tuple gives transparent concurrency control, while placing data and a mutex tuple apart gives visible concurrency control.

### 5.4.1 Location Visibility in MTS-Linda

We have found that location is one of the essential distribution issues, and that it must be possible to specify a tuple space to span a given node set. Here we look at the introduction of location visibility in MTS-Linda, and how location is specified in a host-language.

We see two main strategies for introducing location in a host-language: First, it can be introduced explicitly as a new type, $NodeName$, and then use variables of this type to specify which nodes a tuple space must span, e.g., nodes could be given symbolic names as see in the inter-net domain, where every connected machine has a name in the form `node.local.group.site` (e.g. sleipner.iesd.auc.dk). Second, we can introduce location implicitly by specifying that a new tuple space must span the same nodes as another tuple space (or, a list of other tuple spaces), similar to the positioning of objects in Emerald[15], where an object can be located explicitly at the same node as some other object. We favor the second solution, as it allows us to handle location of one tuple space in terms of the location of others.

To allow a programmer to specify where a new tuple space must be located. We must extend the operations for creation of a tuple space, such they take an argument specifying nodes the new tuple space must span. The argument can be either a single local or shared tuple space variable, or a list of variables. If a list is given, the new tuple space will span all the of the nodes represented by the list of tuple spaces. Figure 5.6, shows the creation of a new local tuple space $my\_ts$, and a shared tuple space $my\_tscap$. The local tuple space will span the same nodes as those spanned by the shared tuple space $a\_tscap$ and the local tuple space $a\_localts$ together. The shared tuple space $my\_tscap$ contains a special tuple space variable, $context$, which denotes the nodes spanned by the context tuple space for the process performing the create operation. The $context$ variable must be introduced to allow a process to create tuple spaces spanning the nodes context tuple space.

**TupleSpace** my_ts({a_tscap, a_localts});
**TSCap** my_tscap = **newts**({context, a_tscap, my_ts});

Figure 5.6: Creation of two tuple spaces, spanning the nodes specified as argument when they are created.

One of the reasons for having location visibility is the ability to place computations on a physical node with certain characteristics. To make specific physical nodes explicitly accessible, the run-time environment in a MTS-linda system, must supply tuple spaces representing only a single physical node. We introduce a special tuple signature, *NamedNode* tuple, to denote a single node by a name: $(\text{"}\texttt{NodeName}\text{"}, TSCap)$. The tuple space capability $TSCap$ is a capability to a tuple space only spanning the node(s) named $NodeName$. For a process to use this named node, it only have to read the tuple, and create a new tuple space using the $TSCap$ as argument. The run-time environment must include a tuple space containing a NamedNode tuple for each tuple in the system, allowing a process to retrieve a capability for a tuple space spanning a single node, based on a symbolic name for that node.

To give full location visibility, it must be possible to get the location of a tuple space. We envision a new MTS-Linda tuple space operation:

```
tscap_list = my_ts.NodeSet();
```

Which returns a list of NamedNode tuple space capabilities, one for each node which is part of the tuple space. Here we are confronted with the problems in conjunction with special tuple spaces attributes and their scope in a tuple space hierarchy. We cannot give the exact semantics of $NodeSet$ operation before the hierarchical problems are solved. We get the strongest $NodeSet$ semantics if location does not follow the hierarchy, but of cause all the conceptual problems.

A consequence of having a strict hierarchy is that the node set for nested local tuple spaces must obey the hierarchy. Thus, demanding a local tuple space to span a given node set can be rejected if the request includes nodes outside the node set spanned by the processes context tuple space. Further, after using the $NodeSet$ operation, a process can be copied to some other tuple space resulting in a remapping of the local tuple space and invalidating a tscap list previously returned by the $NodeSet$ operation. The semantics of the command is thus “Tuple space $my\_ts$ **has been** located on the node set $tscap\_list$”. Having the possibility to fix a tuple space at a specific node set, requires that it cannot be copied to some other node set. And if the operation is attempted then it must fail.

Next we take a look at the introduction of the second essential distribution issue into a host-language: Failures.

### 5.4.2 Failure Visible Special Tuple Space

Failure visibility is the ability to see if an operation in and with a tuple space fails. Here we introduce failure visibility of a distributed special tuple space in a host-language. First we consider which failures are possible, i.e., in which ways access to a distributed tuple space can fail.

In a distributed system a tuple space may become inaccessible in two ways either due to crash or communication failure. In case of a volatile tuple space, a node crash implies that a tuple spaces or parts of a tuple space stored at the failing node are lost, while a persistent tuple space becomes temporarily inaccessible. Link failures may partition a distributed system, thus prohibiting processes in one partition from accessing tuple spaces located in other partitions. Partitioning is usually a temporary phenomena, that is the failing links will eventually be repaired or replaced.

We are not going to look any closer at the exact failure situations, but do require that they are detectable. We need to know if a node is inaccessible, in which case an MTS-Linda operation should fail. The exact semantics of a failing MTS-Linda operation may vary from (special) tuple space to (special) tuple space. Some may give reliable operation while others gives unreliable. The problem is similar to that know from remote procedure call, where the communication primitives guarantee at-most once execution of a procedure call in case of an failure[69].

The important requirement in MTS-Linda is that an operation is atomic thus not leaving tuple space in some intermediate state which potentially may result in failure of subsequent operations.

We need to present these failures to the programmer, this can be done either by exception handling, or by redefining the MTS-Linda operation to return an integer representing the failure status, instead of being void. In example, performing a MTS-Linda operation with a operation return a failure status requires that we check the status of the operation if we need to ensure that it is successful:

```
if( status = in("Test",?i)) {
    // An error has occurred take preventive actions
}
// The operation succeeded!
```

Further, a number of descriptive error messages should be given. A users ability to correct or continue after an error is increased if better error descriptions are given, e.g. if the user knows that an operation failed because a node has crashed he go reboot it!

### 5.4.3 Summary

MTS-Linda supply distribution transparency as default, but no means of selecting visibility. To introduce visibility we found that the visibility builds on visibility of two essential distribution issue; location and failure. The other issues of distribution can be build on using these and the language constructs of MTS-Linda.

We introduced location and failure in MTS-Linda, allowing tuple space to be span a given set of nodes, and allowing the MTS-Linda operation to return an integer value specifying the success of the operation.

Next, we look at the realization of special tuple spaces.

## 5.5 Implementing Special Tuple Spaces

We now turn the question of how to realize specialized tuple spaces. First, we propose a layered system model which provides a very flexible runtime system, allowing users to implement their own tuple spaces. Section 5.5.2 then gives a description of how to simulate tuple spaces with user supplied interfaces almost by using existing MTS-Linda facilities.

### 5.5.1 A Layered System Model

In the previous sections we have hinted a number of tuple spaces with special implementations, but given little information on how they could be implemented.

Our idea is that the runtime system comes with a set of specialized tuple spaces which the system designers have considered the most common and useful. The system does not support obscure combinations special tuple spaces, like a $PropabalisticRealtimeRecoverableTupleSpace$. However, taking the idea a step further we would like that the users themselves were able to implement their own special tuple spaces. Just because the system cannot directly support user's special needs, it does not imply that it should disable them from implementing their specialty themselves. A design open minded for such changes and extensions will accommodate the user's needs, allow diverse applications, and will allow the system to evolve thus providing greater flexibility.

Experiences with the ISIS project, [8], shows similar user demands:

> "Thus the key to satisfying user demands for performance consisted not only of speeding up the basic ISIS protocols, but for providing an interface by which users could plug in their own multicast protocols. Redesigning ISIS so that this interface was simple enough for practical use, while still maintaining the reliability and consistency semantics for ISIS has been challenging."

Similarly, we would like to allow the users to plug in their own tuple spaces. The system supplies a set of basic tuple spaces, and a set of primitive handles which the user can use along with other user supplied tuple spaces to compose his own special tuple spaces.

We propose a layered system model to provide the demanded flexibility. A layered model allows the programmer to exploit hardware, while at the same time maintaining a high-level, transparent view when low-level features are not explicitly needed. The system supplied special tuple spaces should be designed to suffice for the programmer's normal needs. Hackers and system programmers need these low-level features.

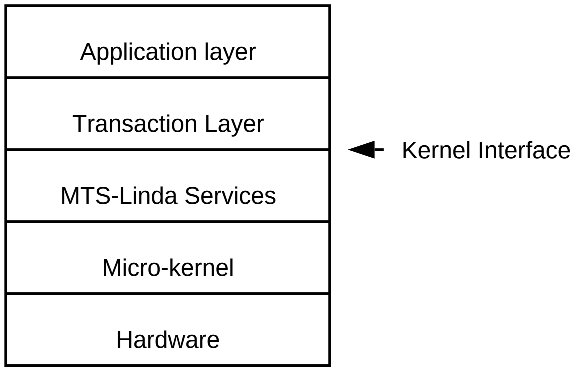

Figure 5.7: A layered system model to realize specialized tuple spaces.

Figure 5.7 illustrates our ideas by means of a model of five layers. At the bottom is the bare hardware, while moving upwards in layers increases transparency.

The functionality of the layers are:

**Hardware Layer:** The physical hardware of the computer system.

**Micro Kernel Layer:** This layer manages the bare hardware, and hides the architecture dependent facilities. It provides basic communication primitives, process and memory management, and some device handling. The layer supplies reliable point to point communication, and a causally ordered multicast mechanism.

**MTS-Linda Service Layer:** This layer is responsible for the basic MTS-Linda functionality. It provides a native, “single node tuple space” i.e., a tuple space data structures and a set of operations to create and delete these, to insert, remove, and match tuples. An iterator is also given for tuple spaces.

Further, this layer also implements a set of services needed to implement special tuple spaces. A persistent single node tuple space is given, and it provides handles and data structures for a compiled tuple space, etc. The interface to this layer gives fully location visibility, and gives failing primitives.

This interface should be fixed and standardized to support portability of the programs using it.

**Transaction Layer:** This layers is concerned with the composition of calls to the basic services to obtain the functionality demanded by MTS-Linda. The standard tuple space, and special tuple spaces as they appears to the users are implemented using this layer. Also new tuple space interfaces supplied by the users are converted in to operations manipulating the underlying data structures.

**Application Layer:** Normal user applications using the standard or special tuple spaces supplied by the transaction layer.

We realize that this is a functional description, and that there is a long way to a design which is simple and consistent enough to be of practical use. A group of our fellow students have designed a system which allows programmers do experiments with distributed systems, See[]. Further, they have proposed a highly flexible Linda system build on the experimental system. We expect that such a system in particular could shed further light on the functionality of the two lower levels. Inspection of the MTS-Linda prototype may also give some strong hints to realization of these layers.

The prototype does not recognize the idea of a layered system[9]. There is a one to one correspondence between the kernel interface and the prototype MTS-Linda model i.e., a system call for each MTS-Linda primitive. This is too inflexible for sharing systems. The transaction layer is buried in the kernel as multi-programmed operation-objects of which each implements a transaction of operations manipulating the underlying distributed single node tuple spaces, called tuple-buckets. Separation of the basic kernel services from the transaction also raises hope for a slightly simpler implementation. A layered system provides only the basic services in the kernel, and interprets

[9] This does not mean that it is inmodular!

simple commands constituting the overall functionality. This philosophy is similar to that of micro-kernels in distributed operating systems: By only implementing the basic services in the kernel, and leaving the unstable functionality in user space, makes it easier to change.

To illustrate the functionality of the transaction layer we in Figure 5.8 sketch how parts of a tuple space spanning several nodes may be implemented, based on single node tuple spaces. The code is in principle very like the prototype implementation of a distributed tuple space. A distributed tuple space is represented as an array of single-node tuple spaces. A hash function is used to map tuples to nodes. Moreover, a coordinator tuple space is used to hold a single tuple used to lock access to tuple space. This lock tuple has a count field describing the number of active processes in the tuple space allowing us to keep determine whether a tuple space is passive or active. To copy the contents of the tuple space it is necessary to hold locks for both source and destination tuple space and the source tuple space must be passive, before copying the individual tuples can begin.

In a similar manner we imagine a replicated tuple space as an array of tuple spaces. When a tuple is inserted in the replicated tuple space the tuple is infact inserted in each of the single node tuple spaces (in practice this may not be as easy as it sounds, as if one of the primitives fail, it should invoke a view change algorithm to make sure that all client processes agree on the replication set)

With a open and flexible system we are, however, faced with a new dilemma: performance versus flexibility. It may take many system calls and communications to build the desired functionality, and further, the runtime environment can make few assumptions about the above behavior for optimization. We expect that the system will be non-strict layered, i.e., upper layers may skip the pie of lower levels to gain performance. Standard or system supplied tuple spaces may also be buried within the kernel for performance reasons.

Other new problems are semantics, and ease of use. Since it is possible to implement many semantic model, this may lead to semantic problems: The system may not be able to ensure, or to rectify proper or "saine" MTS-Linda semantics. Moreover, where the user has a choice of primitives with different semantics, he may choose one that is inadequate for his problem. The flexibility thus has its costs.

### 5.5.2 Enforcing Interfaces

In this chapter we have used the concepts of special tuple spaces, and hinted that a MTS-Linda kernel supporting these can be realized. Here we present a conceptual[10] design of a system supporting special tuple spaces, allowing for enforcement of their interfaces. The design is presented in terms of MTS-Linda, i.e., we use the MTS-Linda coordination abilities to describe communication and process manipulation.

To enforce the interface of a special tuple space, we must makes sure that a process cannot use other methods than those defined. Further, for the system to distinguish between different special tuple spaces they must be named, have type names. These types must be maintained by by a trusted component ensuring that a named special tuple spaces actually represents the type each

[10] That is, it would never be feasible to implement a kernel with special tuple spaces exactly the way outlined below.

```
class TupleSpace
{
    TupleSpace ts[N];                       //A dist ts, an array of N single node ts's
    TupleSpace coordinator;                 //A ts to implement the active/passive and lock
    // state of a tuple space
    void init() {                           //initialize global lock
        coordinator.out(0,FREE);            //global state: 0 active locks, and free glob lock
    }

    tuple in(tuple t) {
        return tuple t2 = ts[hash(t)].in(t);    //find a matching tuple
    }

    void eval(tuple t){
        coodinator.in(? no_active,FREE);    //wait for appropriate global state
        ts[hash(t)].eval(t);                //create new active tuple
        coodinator.out(no_active+no_processes(t),FREE);
//alter and release glo
        bal lock
    }
    void terminate() {                      //called by a process upon termination
        coodinator.in(? no_active,FREE);    //wait for appropriate global state
        coodinator.out(--no_active,FREE);   //alter and release global lock
    }
    void contents_copy(DistTupleSpace ots) {
//lock both tuple spaces and do copy
        c1=min(ots.coordinator.gettsd(),coordinator.gettsd());
        c2=max(ots.coordinator.gettsd(),coordinator.gettsd());
        //locking order matters, and is defined by order
        //of tuple space descriptors: lock c1 before c2
        repeat {                            //Get a hold on both locks (Get double lock)
            c1.in(0);                       //wait for passive 1. ts and lock it
            ots.coodinator.in(? no_active); //Test 2. lock
            if(no_active>0)                 //it was active - we cannot copy!
            {                               //backtrack and wait for passive 2. ts
                c1.out(0);                  //release 1. lock -
                                              allow further activity in 1. ts
                c2.out(no_active);          // release 2. lock tuple
                c2.in(0);                   //wait for a2. ts to become passive
            }
            else
                GotBothLocks=TRUE;
        }
        until(GotBothLocks==TRUE);          //start all over again

        for(i=0;i<N;i++){                   // do the copy operation
            ots[i].rd(?t) all               //retrieve all tuples in other ts
            {
                ts[i].out(t);               //insert in destination ts
            }
        }
```

time it is used.

Intuitively interfaces are enforced by representing the interface for a single method as a tuple, and implementation of the method as a suspended process within the tuple. A process performs an operation on a special tuple space, by either sending a request to the kernel having it to perform the operation or it does it itself. To perform the operation the tuple containing the methods implementation must be dynamically loaded and evaluated. The design presented here only allows for the kernel to load and execute special tuple space methods. But, can easily be extended/changed to allow clients processes to do the same.

To maintain the special tuple space types, we maintain a global special tuple space type-name space. We need for a trusted component to ensure that there is not specified different special tuple spaces types with the same name. Here we introduce a trusted kernel process to handle special tuple space types defined in the system.

Next, we in greater detail presents the kernel enforcing special tuple space interfaces, and maintaining a name space for special tuple space types.

### A Kernel with Special Tuple Spaces

Communication between clients and kernel components is conducted through a system tuple space. Clients **out** tuples with request for the kernel to perform some operation, and **in**'s acknowledgment and data. The kernel, represented by the I-fetch process, **in**'s requests and **out**'s acknowledgment and return data to the client. Access to system tuple space is protected to ensure that client processes only **out** and **in** tuples with certain signatures, defined by the communications protocol.

The kernel must know of each special tuple space and know what code to run when a method is called. Figure 5.9 shows a kernel allowing for special tuple spaces. The 3 interesting components are: the Types-tuple-space, the TS-name server, and the TS-Instances-tuple-space. The types-tuple-space stores “templetes” for instantiating the special tuple spaces known by the kernel.

The TS-Instances tuple space contains the code which must be executed when a method is invoked on an instance of a special tuple space. For each tuple space instance there exists a tuple for each method defined in the instance. To perform a method $f$ on a tuple space $A$, the tuple representing tuple space $A$ and method $f$ must be read, and the process stored in that tuple must be evaluated. Each method is stored as a tuple with tuple space identifier, method name, and a process implementing the method blocked in a `in(?tuple)` operation ready to receive any arguments.

The trusted TS-name server maintains the name space for special tuple spaces. Any special tuple space must be created using this server. The TS-name server allows users to register new special tuple spaces, and remove or reprogram old special tuple spaces if desired. The TS-name server stores definitions of special tuple spaces in the Types-tuple-space. Each special tuple space is stored in the Type tuple space under a name and with a process blocked in an `in(?tuple)` operation. The process knows how to create an instance of the special tuple space, when evaluated in the TS-Instances tuple space.

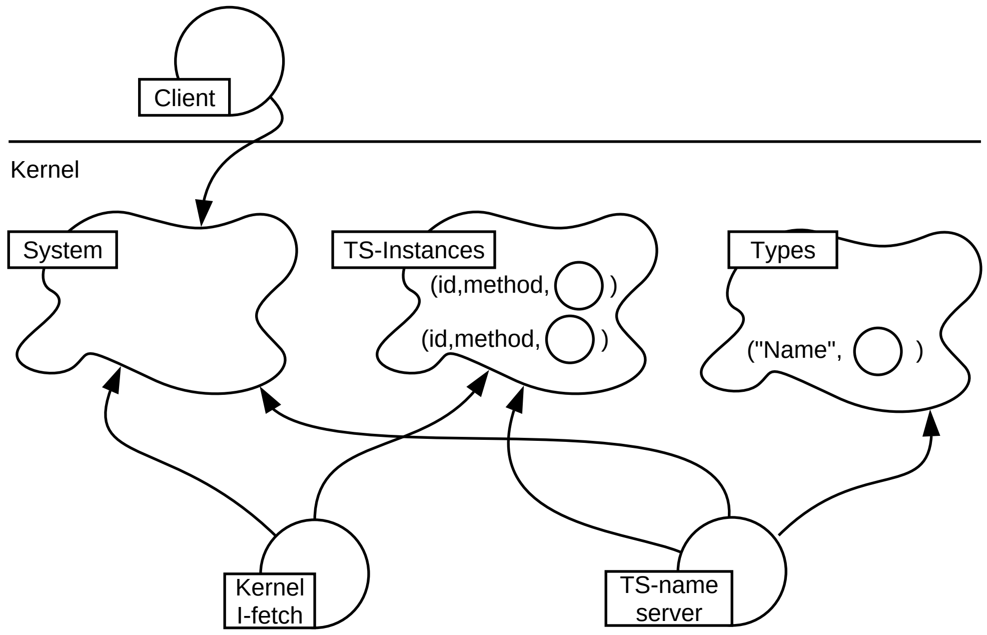

Figure 5.9: The implementation of a single node special tuple space kernel.

Both active kernel components, the Kernel I-fetch and the TS-name server, are quite simple. Figure 5.10 show the Kernel I-fetch routine. Intuitively it fetches a tuple space operation request from the system tuple space and performs it. If the requested method is targeted at a kernel tuple space the method is evaluate directly by the kernel: if it is targeted at a special tuple space, the requested method is retrieved from TS-Instances tuple space and evaluated in the system tuple space.

The TS-name server, Figure 5.11, is a bit more complex as it must execute different methods depending on the request. The example only shows the actions performed when a create-tuple space request is received.

### Implementation of a Queue Tuple Space

In Section 5.2.1 we showed an abstract description of a special queue tuple space. To illustrate how it actually is implemented using the just presented kernel, have we include the implementation of the enqueue and dequeue methods and the create method encapsulating the queue as a special tuple space. A programming environment for programming special tuple spaces would supply a precompiler capable of transforming an abstract special tuple space description into the format required by the kernel. Such a transformation is quite simple.

Figure 5.12 shows the implementation of the enqueue method of the queue tuple space. It is called with a tuple space capability denoting the tuple space on which it is build. Next it performs and **in**

```
while(TRUE){
    System.in("Operation",?tscap,?method-name,?tuple);
    if(tscap.type == kernel_ts){
        // Call the appropriate kernel routine
    }
    else{
        TS_Instance.rd(tscap.id,method-name,?f("Method",? tpl));
        System.eval("Response", f(tuple));
    }
}
```

Figure 5.10: The I-Fetch routine in a MTS-Linda kernel allowing for special tuple spaces.

```
while(TRUE){
    System(?operation,?tuple);
    switch(operation)
    {
    case create_ts:                    // Create a new tuple space instance
        Types.rd(tuple.name,?create("TS-Type",? tpl));
        TS-Instances.eval(create(tuple));
        System.out("Response",NewTSCap);
        break;
    case . . .
    }
}
```

Figure 5.11: The TS-name server code. Only the create tuple space operation is showed.

operation, which try to receive the arguments for the operation. The remainder of the operation is the same as in the abstract description.

```
process tuple enqueue(TSCap ts){
    Tuple tpl;
    in("Method",? tpl);

    ts.in("end",?cnt);
    ts.out("item",cnt++,tpl);
    ts.out("end",cnt);
    return EMPTY_TPL;
}

process tuple dequeue(TSCap ts){
    Tuple tpl;
    in("Method",? tpl);

    ts.in("start",?cnt);
    ts.in("item",cnt++,?tpl);
    ts.out("start",cnt);
    return tpl;
}
```

Figure 5.12: Implementation of the enqueue and dequeue methods, in a queue special tuple space.

When creating a queue special tuple space instance, both methods are eval'ed in the TS-Instance tuple space with a common tuple space capability as argument. This is done by the create process encapsulating the queue tuple space. Figure 5.13 shows the implementation of the create process. The create method starts by being blocked on the **in** operation, waiting for a tuple indicating a new queue tuple space must be instantiated. When the **in** is satisfied, a tuple space representing self is allocated, and each method define on the tuple space is evaluated in the TS-Instance tuple space, each with the self tuple space as argument.

## 5.6 Summary

In this chapter we have discussed a number of the problems in making MTS-Linda a model suitable for distributed programming.

We have given some programming techniques for implementing strongly encapsulated tuple space modules in MTS-Linda. However, these were not entirely satisfactory which led us to a

```
process create(){
    Tuple tpl;
    in.("TS-Type",? tpl);                // Block here until a instance is created

    TSCap tscap = newts();

    eval(tpl.name,"enqueue",enqueue(tscap));
    eval(tpl.name,"dequeue",dequeue(tscap));
    tscap.out("start",1);                // Initial start and end of queue
    tscap.out("end",1);
}
```

Figure 5.13: A create method implementing a queue tuple space with enqueue and dequeue methods.

proposal for tuple spaces as an abstract type with both data and activity. Users may specialize such by giving a tuple space a new interface, which is thus enforced by the kernel. Last in the chapter we described in MTS-Linda terms how the kernel logically can implement these interfaces. MTS-Linda was also extended with a capability based protection scheme.

We claimed that an ideal tuple space is impossible to implement, and that it therefore is necessary to supply special tuple spaces with different degrees of performance and generality. We propose special tuple spaces with location, failure visibility, persistency, replication etc. By concerns of flexibility we furthermore propose a layered system model, which can be used to implement special tuple spaces. The users may also implement their own.

In conclusion, MTS-Linda can be twisted to accommodate most of the requirements found in supporting sharing systems. However, the programmer have to do this twisting, since the system cannot come equipped with predefined solutions to the special needs of the user community.

# 6 Conclusions

In their early days computers were proposed, and used as a an apparatus for numerical solution of differential equations. Today, computers are still being used for numerical computations, but their widely acceptance and application is due the computer's information processing capabilities. Computers are more used for text-editing, storage, and communication of information, and for tools liberating us from laborious tasks, than they are for salvation of differential equations. Distributed systems have, due to their parallel processing capabilities, been conceived as the apparatus for numerical computation of our time. However, taking a glow into the glass-bowl, we see a quite different usage of distributed systems in the future: We envision sharing to be the pre-dominant use of distributed systems. Instead of numerical computations, distributed systems will be used for resource and information sharing, as an interaction media for cooperation, and for communication!

The aim of this project is to propose and to analyze a model for programming of such sharing systems. We attack the problem of supporting the programming of sharing systems; first we present an abstract programming model based on Linda with multiple tuple spaces, which intuitively seems attractive for sharing systems. Thereafter, we give an comprehensive analysis of the requirements imposed on a support system based on a proposal for a taxonomy for a number of important issues in sharing systems. By holding MTS-Linda up against these requirements we find that MTS-Linda fulfils the basic functionality needed to express sharing systems.

However, we also find that MTS-Linda has a number of problems facing the technological demands of current distributed systems and the lack of trust in sharing systems. We therefore propose a number of both minor and major modifications of the abstract tuple space model to accommodate these requirements, mainly by perceiving MTS-Linda's tuple space as an abstract data type which may be instantiated in various forms through what we have termed special tuple spaces. These may come with different generality and performance characteristics, e.g., persistent or replicated tuple spaces, or tuple spaces tailored for queue behavior.

## MTS-Linda as abstraction for Distributed Doing

We propose the MTS-Linda coordination language, Linda augmented with multiple tuple spaces, as a new, high-level model for the programming of sharing systems. Linda gives a distributed shared memory abstraction, called tuple space, of the distributed hardware, and supplies a handful of primitives to manipulate the memory. It provides flexible, and easy-to-use communication and synchronization mechanisms through the anonymous, associative addressing scheme implemented through matching of tuples.

MTS-Linda extends Linda with a module concept by introducing multiple tuple spaces, thus allowing groups of related data and processes to exist in a tuple space distinct from the tuple spaces of other, unrelated groups of data and processes. Furthermore, such modules can be manipulated as a whole thus promoting distributed computations as first class entities. The presented MTS-Linda model provides 2 kinds of tuple spaces: local tuple spaces which conceptually exists in the local environment of a process, and shared tuple spaces accessible through tuple space capabilities. A local tuple space resembles a stack allocated variable, whereas a shared tuple space resembles a heap allocated one.

We conclude that MTS-Linda provides a very powerful coordination model, integrating process manipulation, data storage, and communication into the same handful of primitives. However, the current model has a tendency to be a “take 2, pay for 1” model. The multi-set operations provided to manipulated shared tuple spaces as wholes seem more like system calls, than a language construct. It is not a part of this thesis to give the actual language design for MTS-Linda, but we are looking forward to see a more regular model with better integration of the idea of tuple space hierarchies with communication across these.

Currently we have a prototype implementation for a part of the MTS-Linda model. This prototype is restricted from the manipulation of processes as first class entities, and consequently also from the manipulation of active tuple spaces. Instead the prototype model defines future semantics of tuple spaces i.e., they must become passive before they can be manipulated.

The prototype runs on a local area network consisting of a set of workstations connected by an ethernet. The implementation experiences have shown how multiple tuple spaces and their operators may be implemented. From this experience we conclude that multiple tuple spaces indeed are implementable, but benchmarking the prototype shows that it does not satisfy the performance requirements. However, it is our belief that the performance can be enhanced considerably by implementing a new communication system: Currently, a write-read pair of tuple space memory is 2 to 3 times as slow as a remote procedure call.

## MTS-Linda and Sharing Systems

Chapter 3 gives a detailed analysis of sharing systems and Chapter 4 argues for MTS-Linda’s abilities to express such. The analysis propose a list of relevant issues of sharing systems, and uses these in an examination of 3 example systems: programming environments used by many programmers developing large systems and sharing a number of the developed documents, computer supported cooperative work (CSCW) where the computers aid a cooperative working situation providing for interaction among people, and distributed operating systems managing the resources of the distributed system. The result is an extensive set of qualitative requirements a sharing support system must address.

The overall functional requirement is that processes (on behalf of users) separated in time, or by logically or physically disjoint address spaces must be able to create shared resources and data-entities, and to synchronize and communicate about the access to these. Sharing systems thus require strong means for coordination and data-sharing. High flexibility, reliability, and performance plays an important role. Moreover, protection mechanisms acknowledging sharing and cooperation are also mandatory.

A surprising result, to us at least, is multi-programming environment's and CSCW application's assault on transparency. Visibility of multiple users and the distributed hardware is equally important as transparency. The 2 types of sharing systems require awareness of the presence of multiple users, and of the cooperative working situation. Moreover, supporting this cooperation implies an inherent need to be able to control the underlying system: The traditional database and file-sharing technology is often unsatisfactory for sharing system, and thus needs a reconsideration. Perhaps MTS-Linda's associative data store with explicit synchronization on access has something new to offer.

From the analysis it is our thesis that MTS-Linda fulfils the basic functionality of sharing systems, however, this still needs to be proved by actual using the model in constructing practical sharing systems. MTS-Linda gives a set of flexible primitives that can be used to program the desired interaction patterns in a simple way. However, the abstract MTS-Linda model also has a number of shortcomings: MTS-Linda should be extended with a failure model, selective transparency, persistent storage, protection, and its encapsulation mechanisms should also be examined further.

## Qualifying MTS-Linda

To make MTS-Linda qualified for the programming of "real" sharing systems we attack the above mentioned problems. We are faced with the problem of implementing an ideal tuple space that never fails and is infinitely fast. This is of course impossible, so a number of compromises need be made in an implementation.

Our recommended solution is the notion of special tuple spaces which allows the programmer to adjust the properties of tuple spaces to his special needs: The programmer may dictate a tuple space to reside at a subset of the nodes of the distributed system, or to be recoverable or replicated to enhance fault tolerance etc. By making tuple spaces persistent MTS-Linda provides the necessary storage media for sharing systems, resulting in a unified memory model where the same means are used for accessing shared in-core memory and shared persistent memory. As an aside it should also be acknowledged that MTS-Linda needs an iterator mechanism to enhance MTS-Linda's querying facilities.

As a natural way to accommodate the flexibility and transparency demands of sharing systems we propose a layered system model, which allows the users to take advantage of the hardware when this is desired. The lower levels of this model provides full visibility of the underlying distributed hardware, while the higher layers provides more and more transparency such that the top-most level is the fully transparent abstract MTS-Linda model.

However, a particular concern of MTS-Linda is whether it will be robust enough; communication in MTS-Linda is by side-effecting common data-stores, and is thus inherently state-full.

As another demand introduced by sharing systems, protection can be integrated in MTS-Linda by associating tuple space capabilities with rights and user domains. On the issue of separating modules we deal with the problems of strong encapsulation, protection and reprogramming in MTS-Linda. Multiple tuple spaces provides for grouping related data and active behaviors into modules. However, we find that the encapsulation given by tuple space may be too weak for sharing systems; sharing systems need strongly encapsulated modules where the only method of interaction with the module is via a specified set of operations. Weak encapsulation is only applicable within a single application, or in a closely coupled and trusted collection of applications. Many of the other distributed shared memory models we have seen in the literature suffers from the same problem, and must either extend their models with strong encapsulation, or accept another coordination model among loosely coupled applications.

It is our conviction that it is possible to integrate protection and reprogramming with MTS-Linda. Strong encapsulation is provided by encapsulation in processes, while more research on strongly encapsulated tuple space modules is required.

Finally, as a disadvantage we hitherto have not dealt with explicitly, scaling of MTS-Linda systems may become problematic. The model seems most appropriate for networks located within a single building: The distributed shared memory abstraction given by MTS-Linda has a tendency to give a close coupling among the machines, implying low interdependency. Further, while nothing technological hinders that MTS-Linda is used over wide geographically distances, and across organizatorical boundaries, it does not support the management problems of enforcing cooperation and specification of common policies found in these systems. In terms of the taxonomy of programming environments described in Chapter 3.2, MTS-Linda seems appropriate for the individual, families, and small cities, while it is inappropriate for large city and state scale systems. In general, the large scale models are not well understood and demands for more research, so this deficiency may not be specific for MTS-Linda.

## Overall Conclusions

In the introduction we gave 3 overall requirements for a programming system supporting sharing. It should be simple and easy to learn and use, it should be suitable for programming of sharing systems, and it should be efficient. Through MTS-Linda we have presented a programming model for distributed system which encompasses communication, synchronization, storage, and process and module manipulation in a handful of primitives. This is an indicator of simplicity, but it is of course no guarantee. Furthermore, it is our claim that programming with distributed data structure as suggested by MTS-Linda is “distributed programming without tears”.

It is our firm belief that MTS-Linda along with special tuple spaces and a flexible implementation fulfills a large number of the requirements raised by particularly small scale sharing systems. MTS-Linda is therefore suitable for sharing systems.

However, being concerned about efficiency, our prototype does not exhibit the required performance. Still, throughout the report we have proposed a number of improvements and estimates of the obtainable performance. Given the high abstraction provided by MTS-Linda, we are prepared to accept the performance of MTS-Linda when these improvements are incorporated into an implementation.

*It is our conclusion that MTS-Linda is a good model for distributed programming, which deserves further exploration in the context of sharing systems.*

## Future Work

In this thesis we have concentrated on the analysis of the intended application domain of MTS-Linda, and on an overall examination of MTS-Linda's abilities in this context. We have concluded that MTS-Linda has a number of strong points which deserves further exploration. Some of the relevant aspects which deserves to be touched upon by future work are the following:

**Reconsideration of linguistics:** We adopted the model for multiple tuple spaces as an "as is" product, before we analyzed its application domain! The linguistics should be reconsidered to reflect our choice of application domain—sharing systems—and the analysis of it provided in this report. If special tuple spaces providing user specified interfaces is conceded, this will have major impact on the linguistics, as it requires a interface specification language, and an attitude to a type system for MTS-Linda.

**A new MTS-Linda implementation:** Given our new knowledge about the application domain of MTS-Linda it is feasible with a new implementation to try out the ideas with special tuple spaces and a flexible, layered system model in practice. However, we believe that there is a large design work ahead before these ideas can be turned into a working design ready for implementation. One specific issue is the semantics of hierarchies of special tuple spaces.

**Improvements of performance:** The performance of the prototype is insufficient, and thus we have hinted a number of possible improvements. One was a new communication system designed for MTS-Linda, which avoids the large overhead found in the prototype. Another is the use of special tuple spaces as means of performance improvements—the implementation of a tuple space can be optimized according to the specific use of the special tuple space. Generally, implementation techniques improving the performance of MTS-Linda and sharing systems deserves much further work.

**Development of large programs:** MTS-Linda may seem quite attractive from the theoretical framework presented here, but MTS-Linda still needs to pass the most demanding test: to prove suitable in practice. MTS-Linda should be used in realistic sharing-situations, and we lack programming experience with developing large programs in MTS-Linda!

In closing, it is our belief that it will be interesting to take the idea of a coordination languages a number of steps further than it is today. Like it in sequential programming languages is possible to specify abstractions over the computation of expressions in form of functions, we imagine that a coordination language will make it possible to specify, name, instantiate, and compose abstractions of common synchronization and communication patterns—a kind of coordination schemas. It is thought provoking that many contemporary distributed programming languages treat coordination as an aside to the language facilities found in traditional programming languages, while the essential problems in sharing systems is about coordination over time and space. We see Linda and MTS-Linda as some of the first languages dealing with coordination as a language phenomena in its own right. Perhaps MTS-Linda will turn up as “the Fortran of coordination languages”? –Nevertheless, we believe that coordination abstractions, as an off-the-shelf product which can be used when a particular interaction pattern is needed, will make distributed programming easier!

# Bibliography

[1] Shakil Ahmed. YALE — The Yale Automated Linda Editor. Technical report, Yale University. Department of Computer science.

[2] Brian G. Anderson and Dennis Shasha. Persistent Linda: Linda + Transactions + Query Processing. In editor D. LeMetayer, editor, *Proceedings of the Workshop on Research Directions in High-Level Parallel Programming Languages*, pages 129–141. IRISA-INRIA, June 17-19 1991. Mont Saint-Michel, France.

[3] Gregory R. Andrews, Richard D. Schlichting, Roger Hayes, and Titus D. M. Purdin. The Design of the Saguaro Distributed Operating System. *IEEE Transactions on Software Engineering*, se-13(1):104–118, January 1987.

[4] Keld Ramstedt P. Bach, Peter Bisgaard, Thomas Jesper Hansen, Morten Irskov, Peter Krogh Jensen, Brian Nielsen, Jørgen Lundbeck Nielsen, and Tom Sørensen. AUC-Linda: System Manual. 8th semester project report (supervised by Keld K. Jensen, Michael J. Manthey and Arne Skou), University of Aalborg, Institute for Electronic Systems, Department of Mathematics and Computer Science, June 1992.

[5] Henri E. Bal, Jenniger G. Steiner, and Andrew S. Tanenbaum. Programming Languages for Distributed Computing Systems. *ACM Communicating Surveys*, 21(3):261–322, September 1989. Special issue on Programming Language Paradigms.

[6] M. Ben-Ari. *Principles of Concurrent and Distributed Programming*, chapter 8, pages 88–102. Computer Sciece. Prentice Hall International, Hartfordshire, London, 1990. ISBN 0-13-711821-X.

[7] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison–Wesley Series in Computer Science. Addison Wesley, Reading, Massachusetts, 1987. ISBN 0-201-10715-5.

[8] Kenneth Birman and Robert Cooper. The ISIS Project: Real Experience woth a Fault Tolerant Programming System. Technical Report TR 90-1138, Department of Computer science, Cornell University, Ithaca, New York, July 1990.

[9] Kenneth P. Birman. The Process Group Approach to Reliable Distributed Computation. Technical Report TR 91-1216, Department of Computer science, Cornell University, Ithaca, New York, July 1991.

[10] Kenneth P. Birman and Thomas A Joseph. Reliable Communication in the Presence of Failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[11] Andrew D. Birrell and Brouce Jay Nelson. Implementing Remote Procedure Calls. In *Proceedings of the ninth ACM Symposium in Operating System Principles*, pages 3, 17–37, Mount Washington Hotel, Bretton Woods, New Hampshire, October 10–13 1983.

[12] Roberto Bisiani and Alessandro Forin. Multilanguage Parallel Programming of Heterogeneous Machines. *IEEE Transactions on Computers*, 37(8):930–945, August 1988.

[13] Roberto Bisiani, François Lecouat, and Vincenzo Ambriola. A Tool to Coordinate Tools. *IEEE Software*, pages 17–25, November 1988.

[14] Robert Bjornson, Nicholas Carriero, David Gelernter, and Jerrold Leichter. Linda, the Portable Parallel. Research Report YALEU/DCS/RR-520, Yale University. Department of Computer science., January 1988.

[15] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy, and Larry Carter. Distribution and Abstract Types in Emerald. *IEEE Transactions on software engineering*, SE-13(1):65–76, January 1987.

[16] Lothar Borrmann, Martin Herdieckerhoff, and Axel Klein. Tuple Space Integrated into Modula-2, Implementation of the Linda Concept on a Hierarchical Multiprocessor. In Reinartz Jesshope, editor, *Proceedings CONPAR '88*. Cambridge University Press, 1988.

[17] Paul Butcher and Hussein Zedan. Lucinda — A Polymorphic Linda. In editor D. LeMetayer, editor, *Proceedings of the Workshop on Research Directions in High-Level Parallel Programming Languages*, pages 65–81. IRISA-INRIA, June 17-19 1991. Mont Saint-Michel, France.

[18] Christian J. Callsen. Environments for Developing and Using Distributed Systems. Aalborg University. Department of Mathematics and Computer Science, November 27 1991.

[19] Christian J. Callsen, Ivan Cheng, and Per L. Hagen. The AUC C++Linda System. In *EPCC Workshop in Linda-Like Systems.*, Edinburgh Parallel Computing Center. University of Edinburgh. UK, June 24 1991.

[20] Nicholas Carriero and David Gelernter. How to Write Parallel Programs: A guide to the Perplexed. *ACM Communicating Surveys*, 21(3):323–358, September 1989. Special issue on Programming Language Paradigms.

[21] Paolo Ciancarini. Cases of Coordination: a Multiple Tuple Space Approach. Internal note, Yale University, Department of Computer Science, December 1990.

[22] Paolo Ciancarini. Parallel Logic Programming Using the Linda Model of Computation: Semantics and Implementation. In *Proceedings of the Workshop on Research Directions in High-Level Parallel Programming Languages*, pages 101–115. IRISA-INRIA, June 17-19 1991. Mont Saint-Michel, France.

[23] Paolo Ciancarini, Keld K. Jensen, and Dani Yankelevich. The Semantics of a Parallel Lanuguage based on a Shared Data Space. Technical Report TR - 26/92, Universiàt di pisa. Dipartimento di informatica, Corso Italia 40 — 56100 Pisa — Italy, August 1992.

[24] Inc Cogent Research. Kernel-Linda Specification – Revision 3.9, March, 10 1989.

[25] Partha Dasgupta, R. Anantharayanan, Sathis Menon, Ajay Mohindra, Mark Pearson, Raymond Chen, and Christopher Wilkenlog. Language and operating system support for distributed programming in clouds. In USENIX *Workshop on Experiences with Distributed and Multiprocessor Systems (SEDMS II)*, pages 321–340, Fort Lauderdale, FL, March 1991. USENIX Association.

[26] C.A. Ellis, S.J. Gibbs, and G.L. Rein. Groupware — Some Issues and Experiences. *Communications of the ACM*, 34(1):38–58, January 1991.

[27] I. J. P. Elshoff. A distributed debugger for Amoeba. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN NOTICES*, 24(1):1–10, January 1989.

[28] Dror G. Fietelson and Larry Rudolph. Distributed Hierarchical Processing. *IEEE Computer*, 23(5):65 – 77, May 1990.

[29] Charles J. Fleckenstein and David Hemmendunger. Using a Global Name Space for Parallel Execution of UNIX Tools. *Communications of the ACM*, 32(9):1085–1090, September 1989.

[30] Brett D. Fleisch and Gerald J. Popek. Mirage: A Coherent Distributed Memory Design. In *Proceedings of the Twelfth ACM Symposium on Operating System Principles*, pages 211–223, December 1989.

[31] David Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[32] David Gelernter. Elastic Computing Envelopes and their Operators. Internal note, Yale University, Department of Computer Science, September 1988.

[33] David Gelernter. Multiple Tuple Spaces in Linda. *Yale University, Department of Computer Science*, March 1989. Invited paper PARLE 89.

[34] David Gelernter and Nicholas Carriero. Coordination Languages and their Significance. *Communications of the ACM*, 35(2):96–107, February 1992.

[35] David Gelernter, Suresh Jagganathan, and Thomas London. Environtments as First Class Objects. In *Proceedings of the ACM symposium on Principles of Programming Languages*, pages 98–110, 1987.

[36] Adele Goldberg and David Robson. *Smalltalk-80 – The Language and its Implementation*. Addison-Wesley, 1983.

[37] Andrzej Goscinski. *Distributed Operating Systems, the logical design*, chapter 4, pages 73–129. Addison Wesley, Reading Massachusetts, 1991. ISBN 0-201-41704-9.

[38] Asbjørn Gregersen and Frank Hejselbæk Møller. Distribution af HyperText backend. Master's thesis, Aalborg University, Institute for Electronic Systems, Department of Mathematics and Computer Science, Denmark, June 1991. In Danish.

[39] Andreas Gustafsson, Hannu Aronsson, and Heikki Suonsivu. GDM — Global Distributed Memory. Technical Report TKO-C49, Helsinki University. Faculty of Information Technology. Department of Computer science, May 1991.

[40] Robert H. Halstead, Jr. Parallel Symbolic Computing. *IEEE Computer*, 19(8):35–43, August 1986.

[41] W. Hasselbring. On Integrating Generative Communication into the Prototyping Language ProSet. Technical Report Bericht 05-91, University of Hessen, Germany. Department of Computer science / Software Engineering., December 1991.

[42] Carl Hewitt. The Challenge of Open Systems. *Byte*, 10(4):223–242, April 1985.

[43] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[44] Susanne C. Hupfer. Melinda: Linda with Multiple Tuple Spaces. Research Report YALEU/DCS/RR-766, Yale University. Department of Computer science., February 1990.

[45] Suresh Jagannathan. Semanatics and Analysis of First-Class Tuple-Spaces. Research Report YALEU DCS/RR-783, Yale University, Department of Computer Science, April 1990.

[46] Keld K. Jensen and Susanne Hupfer. Tuple Spaces as First-Class Linda Objects. Unpublished article. Received from Keld K. Jensen (kondrup@iesd.auc.dk) Aalborg University, Denmark, Institute for Electronic Systems, Department for Mathematics and Computer Science., April 21 1991.

[47] Keld Kondrup. Jensen. Decoupling of Computation and Coordination in Linda. In D. Heidrich and J. C. Grossetie, editors, *Computing with T.Node Parallel Architectures*, pages 43–62, 1991.

[48] Keld Kondrup Jensen. *Towards a Multiple Tuple Space Model*. PhD thesis, Aalborg University. Department of Mathematics and Computer Science, 1993. Upcomming.

[49] Eric Jul. Object Mobility in a Distributed Object–Oriented System, December 13 1988. Also Available as DIKU Technical Report no. 89/1. University of Copenhagen.

[50] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[51] Kenneth M. Kahn and Mark S. Miller. Language Design and Open Systems. In B. A Huberman, editor, *The Ecology of COMPUTATION*, pages 291–314. North–Holland, 1988. ISBN 0444 70379 9.

[52] David Kaminsky and Nicholas Carriero. Experiments with Piranha Parallelism. In *Research Directions in High-Level Parallel Programming Languages*, pages 45–64, June 17-19 1991. Mont Saint-Michel, France.

[53] David L. Kaminsky. The Hypercomputer: A Network Process management System. Research Report YALEU/DCS/RR-826, Yale University. Department of Computer science., September 1990.

[54] Brian W. Kernighan and Rob Pike. *The UNIX Pprogramming Environment*. Prentice Hall Software Series. Prentice Hall, Inc, Englewood Cliffs, New Jersey 07632, 1984. ISBN 0-13-937699-2.

[55] Henry F. Korth and Abrahan Silberschatz. *Database System Concepts*. McGraw-Hill Series in Systems. McGraw-HILL, Inc., New York, second edition edition, 1991. ISBN 0-007-100804-7.

[56] Wm Leler. Linda Meets Unix. *IEEE computer*, 23(2):43–54, February 1990.

[57] Ian M. Leslie, Derek McAuley, and Sape J. Mullender. Pegasus — Operating System Support for Distributed Multimedia Systems. *ACM Operating Systems Review*, 27(1):69–78, January 1993.

[58] Ted G. Lewis and Hesham El-Rewini. *Introduction to Parallel Computing*. Prentice–Hall International, Englewood Cliffs, New Jersey, U.S.A., 1992. ISBN 0-13-498916-3.

[59] Luping Liang, Chanson, Samuel T. Neufeld, and Gerald W. Process Groups and Group Communications. *IEEE Computer*, pages 56–66, February 1990.

[60] Barbara Liskov. Distributed Programming in Argus. *Communications of the ACM*, 31(3):300–312, March 1988.

[61] Ami Litman. The DUNIX Distributed Operating System. *ACM Operating Systems Review*, 22(1):42–51, January 1988.

[62] Inmos Ltd. Occam model. Transputer manual, September 5 1986.

[63] Bruce J. MacLennan. *Principles of Programming Languages: Design, Evaluation and Implementation*. Hold, Reinhart and Winston, Inc., New York, second edition edition, 1987. ISBN 0-03-005163-0.

[64] Michael Manthey. Transparent Transputing. To be Published, August 1991. Aalborg University, Institute for Electronic Systems, Department of Matemathics and Computer Science.

[65] John Markoff. David Gelernter's Romance With Linda. *New York Times*, Business Section (Section 3):1, January 19 1992.

[66] W. Anthony Mason. Distributed Processing: The State of the Art. *BYTE*, pages 291–298, November 1987.

[67] Satoshi Matsuoka and Satoru Kawai. Using Tuple Space Communication in Distributed Object-Oriented Languages. In *ACM Conference Procedings, Object Oriented Programming Systems, Languages and Applications, San Diego California*, pages 276–284, Septemeber 25–30 1988.

[68] Robert M. Metcalfe and David R. Boggs. Ethernet: Distributed Packet Switching for Local Computer Networks. *Communications of the ACM*, 19(7):395–404, July 1976.

[69] Sun Microsystems. *Network Programming Guide*. Sun Microsystems, Inc., 1990. Part Number: 800-3850-10.

[70] S.J. Mullender. Protection. In S.J. Mullender, editor, *Distributed Systems*, pages 117–132 (ch. 7). ACM Press (Addison-Westley Publishing Company), Reading, Massachusetts, 1989.

[71] Brian Nielsen and Tom Sørensen. Implementing Linda with Multiple Tuple Spaces. Internal s9d semester project report, University of Aalborg, Institute for Electronic Systems, Department of Mathematics and Computer Science, January 1993.

[72] Kurt Nørmark. Programming Environments—Concepts, Architectures, and Tools. Technical Report R-89-5, Department of Mathematics and Computer Science,Institute of Electronic Systems, Aalborg University, March 1989.

[73] Cherri M. Pancake and Donna Bergmark. Do Parallel Languages Respond to the Needs of Scientific Programmers. *IEEE Computer*, 23(12):13–23, December 1990.

[74] Henrik Pedersen, Dieter Gehrke, Martin Guido Jensen, Jens Normann Larsen, and Lars Ruben Skyum. C++Linda. Internal s9d semester project report, University of Aalborg, Institute for Electronic Systems, Department of Mathematics and Computer Science, January 1989.

[75] D. E. Perry and G. E. Kaiser. Models for Software Development Environments. *IEEE Transactions on Software Engineering*, 17(3):283–295, March 1991.

[76] Michel Raynal. *Distributed Algorithms and Protocols*. Wiley series in computing. John Wiley & Sons Ltd., New York, 1 edition, 1988. ISBN 0-471-91754-0.

[77] T. Rodden, J.A. Mariani, and G. Blair. Supporting Cooperative Applications. In *Computer Supported Cooperative Work (CSCW)*, pages 41–67. Kluwer Academic Publishers, 1992.

[78] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, S. Langlois C. Kaiser, P. Léonard, and W. Neuhauser. Overview of the CHORUS Distributed Operating systems. Research Report CS/TR-90-25, Chorus systèmes., April 15 1990. A revised and updated version of an article published in "Computing Systems", The Journal of the Usenix Association, Volume 1, Number 4.

[79] Michael L. Scott. Language Support for Loosly Coupled Distributed Programs. *IEEE Transactions on Software Engineering*, SE-13(1):88–103, January 1987.

[80] Ellen H. Siegel and Eric C. Cooper. Implementing Distributed Linda in Standart ML. Technical Report CMU-CS-91-151, School of Computer science, Carnigie Mellon University, Pittsburgh, PA 15231, October 1991.

[81] Morten O. N. Simoni and Poul Sloth. Hypertekstbaseret konfigurationsstyring og versionskontrol. Master's thesis, Aalborg University, Institute for Electronic Systems, Department of Mathematics and Computer Science, Denmark, June 1991. In Danish with english abstract.

[82] W. Richard Stevens. *UNIX Network Programming*. Prentice Hall, Inc., Englewood Cliffs, New Jersey, 1990. ISBN 0-13-940876-1.

[83] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, Workingham, England, second edition edition, 1991. ISBN 0-201-53992-6.

[84] Michael Stumm and Songnian Zhou. Algorithms Implementing Distributed Shared Memory. *IEEE Computer*, 23(5):54 – 64, May 1990.

[85] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice/Hall International, Englewood Cliffs, New Jersey, U.S.A., 1992. ISBN 0-13-595752-4.

[86] Andrew S. Tanenbaum and Robbert Van Renesse. Distributed Operating Systems. *Computing Surveys*, 17(4), December 1985.

[87] B. H. Tay and A. L. Ananda. A Survey of Remote Procedure Calls. *Operating Systems Review*, 24(3):68–79, July 1990.

[88] Peter Wegner. Guest Editor's Introduction to Special Issue of Computing Surveys. *ACM Communicating Surveys*, 21(3):253–258, September 1989. Special issue on Programming Language Paradigms.

[89] Uffe Kock Wiil. Issues in the Design of EHTS: A Multiuser Hypertext System for Collaboration. Technical Report IR 91-24, Aalborg University. Department of Mathematics and Computer science., 1991.

[90] Larry D. Wittie. Computer Networks and Distributed Systems. *IEEE Computer*, 24(9):67–75, September 1991.

[91] Andrew S. Xu. A Fault-Tolerant Network Kernel for Linda. Technical Report MIT/LCS/TR-424, Massachusetts Institute of Technology. Laboratory for Computer science, 545 TEchnology Square, Cambridge,Massachusetts 02139, August 1988.

# A Performance of the Prototype

This appendix gives a set of performance measurements performed on the MTS-Linda prototype.

## A.1 Performance Measurements

An important issue in distributed programming is performance. To give an indication of the performance of MTS-Linda we here present a set of basic timing measurements of the prototype.

Whether a system exhibits "good performance" is a subjective matter, which dependent on the expressiveness of the model. To impose a more objective evaluation criteria we require that the performance of MTS-Linda is comparable to the interprocess communication and process management facilities of UNIX. However, since the prototype uses UNIX as the implementation platform, we actually expect the performance of the prototype to be slower than UNIX.

Since MTS-Linda conceptually is shared memory, a more appropriate measure of comparison would arguable be shared memory access. However, by default we are working on a distributed architecture so comparison with memory access is neither really fair nor fertile. We find comparison with competing message passing systems more appropriate. Comparing with UNIX also allows us to determine how much additional overhead MTS-Linda causes compared to basic operating system facilities.

To test the performance of the prototype we have carried out the following tests:

- **1000 out/in-pairs.**
  This test shows how fast tuple-exchanges are performed in the prototype.
- **1000 out's followed by 1000 rd's.**
  The test examines how much overhead tuple matching imposes.
- **100 eval-operations.**
  This test is used to determine the cost of managing concurrency in MTS-Linda programs.

- **100 creations and deletions of tuple spaces.**
  The test determines the cost of creating and deleting tuple spaces.
- **20 copy and move operations on tuple spaces as function of the amount of tuples in tuple space.**
  This test determines the cost of manipulating a tuple spaces as a whole.

All test programs run as client-programs, and timing is performed in client code. Hence, the tests show the performance as client processes experiences it (the real thing): The test programs uses a "Timer" class which implements a handy stopwatch. It uses UNIX' `gettimeofday()` call to read the system clock; the time-figures are thus in real time, not cpu-time. The timer is started using `start()`, and stopped using `stop()`. At `stop()` points the timer automatically calculates average, remembers maximum and minimum elapsed times, over several measurements.

The tests were performed on the departments 10 Mbs ethernet network. The network was relatively unloaded, but some background activities existed, e.g., the network file system and network information service. The nodes were a set SUN sparc ELC workstations.

In the following sections we present each test and its results in turn.

## A.2 Measurement of out/in pairs

As communication among MTS-Linda processes primarily takes place by exchanging simple tuples (**out**, **in**, **rd**, of tuples without tuple space fields), it is important examine their cost. We therefore measure the tuple-exchange-time and the match-overhead.

Tuple exchange time is measured as the time it takes a process to do an **out** followed by a matching **in**. The test program is illustrated in Figure A.1. Timing results are provided both in the case where the exchanged tuple resides on the same node as the process (internal), and in the case where it resides on another node (external). Compared to UNIX, an **out**/**in**-pair should have same (or less) cost as a remote procedure call[1].

### Results

The results of the tests is shown in Figure A.2. The cost of the single MTS-Linda operations is in the magnitude of milli-seconds, and is within the ranges: 3 ms – 3.5 ms for access to tuples located internally, and 6 ms – 6.5 ms for tuples located externally. The cost of tuple exchange is calculated as the sum of the cost of an **out** and **in**, giving 6.41 ms internally and 12.46 ms externally. These figures are included in Figure A.2.

[1]One can argue wheter this comparison with remote procedure calls is good. One could also argue that it takes 4 Linda operations to implement a remote procedure call (though they run 2 and 2 in parallel). However, it is not the purpose to set up an exact meassure, but to give an idea aboput the cost of MTS-Linda compared to another highlevel communication model. Exact comparison is made very dificult, because MTS-Linda and remote procedure calls are different programming paragdigms (client/server versus distributed data structures).

```
Timer t_in, t_rd, t_out;      // Timer to meassure time of in, rd and out -
                              //   operations
for(int i=0;i<1000;i++)       // perform 1000 probes
{
    t_out.start();            // start out-timer
    WorkSpace.out(i);         // insert test tuple
    t_out.stop();             // stop out-timer

    t_rd.start();             // start rd-timer
    WorkSpace.rd(i);          // read the test tuple
    t_rd.stop();              // stop rd-timer

    t_in.start();             // start in-timer
    WorkSpace.in(i);          // retrieve the test tuple
    t_in.stop();              // stop in-timer
}
```

Figure A.1: Test program for time-measurements of tuple-exchange time(**out**-**in**-pairs)

| | internal (ms) | external (ms) |
|---|---:|---:|
| Out | 3.09 | 6.13 |
| In | 3.32 | 6.33 |
| Rd | 3.44 | 6.46 |
| In-Out-pair | 6.41 | 12.46 |
| tcp-RPC | 2.72 | 3.57 |
| tcp-roundtrip | 2.13 | 2.91 |
| shared-mem-roundtrip | 0.57 | – |

Figure A.2: Performance results of **out**/**in**-pairs in MTS-Linda and comparison with UNIX interprocess communication. Time units are in milli-seconds.

The **in**-operations are slightly slower than **out**-operations (0.2 ms). This difference stems from the test program, where the **in**-operation always finds a matching **out**-tuple, whereas the **out**-operation finds the **in**-operation table empty and no match attempt takes place. However, the **in**-operations are slightly faster than **rd**-operations (0.13 ms). The difference is due to the duplication of the **out**-tuple. This is a substantial overhead, which is due to the copy operation in the kernel—tuples and fields are represented in a parse-tree-like structure, and duplication of this structure is too expensive!

The test program exchanges a tuple with a single field only. We have measured the additional overhead per field, dependent on its size, to be approximately 0.1 ms, which are used for packing, unpacking, and match.

**Communication Overhead.**

The 5 steps involved in a simple **out**-operation are: The client packs its tuple into a message which is the sent to the local nucleus. The client then awaits an acknowledgment message. When the nucleus receives a new message from a client it unpacks the message and calculates the tuple-to-node hash key. If the receiver node is itself it looks up the tuple's destination tuple space, performs tuple-to-bucket hashing, obtains the bucket-lock, and searches for matching **in**/or **rd** operations. Finally, it creates an acknowledge message and routes it back to the client. If the receiver of the tuple is another nucleus the tuple is routed to the receiver where it is handled similarly. When the client receives the acknowledgment it continues computing. An **in** or an **rd** proceeds similarly, but a matched tuple is returned instead of an acknowledgment. Thus, a local simple operation requires two internal tcp-messages, whereas an external operation requires two internal tcp-messages and two external tcp-messages. The messages involved in an external tuple access is illustrated in Figure A.3.

The cost of tcp communication is included in Figure A.2 as the roundtrip time[2]. In percentages the client-nucleus tcp communication amounts to (the $T$ with index refers to the figures of Figure A.2)

$$T_{internal\_tcp\_roundtrip} * 100/T_{internal\_out}\% = 69\%$$

of the cost for internal tuple-operations, and

$$(T_{internal\_tcp\_roundtrip} + T_{external\_tcp\_roundtrip}) * 100/T_{external\_out}\% = 82\%$$

for external tuples-operations. Thus, inter process communication is the predominate cost of tuple-operations in the prototype.

[2]The roundtrip time involves two tcp-messages: A client process sends a message to a server process, and then awaits a reply message from the server. The server awaits a message from the client and immediately sends it back. The roundtrip-time is the time evolving from the point where the client sends a message to the client until it receives the reply. The client and server was 2 independent heavy-weight processes, and the size of the exchanged messages was 128 bytes.

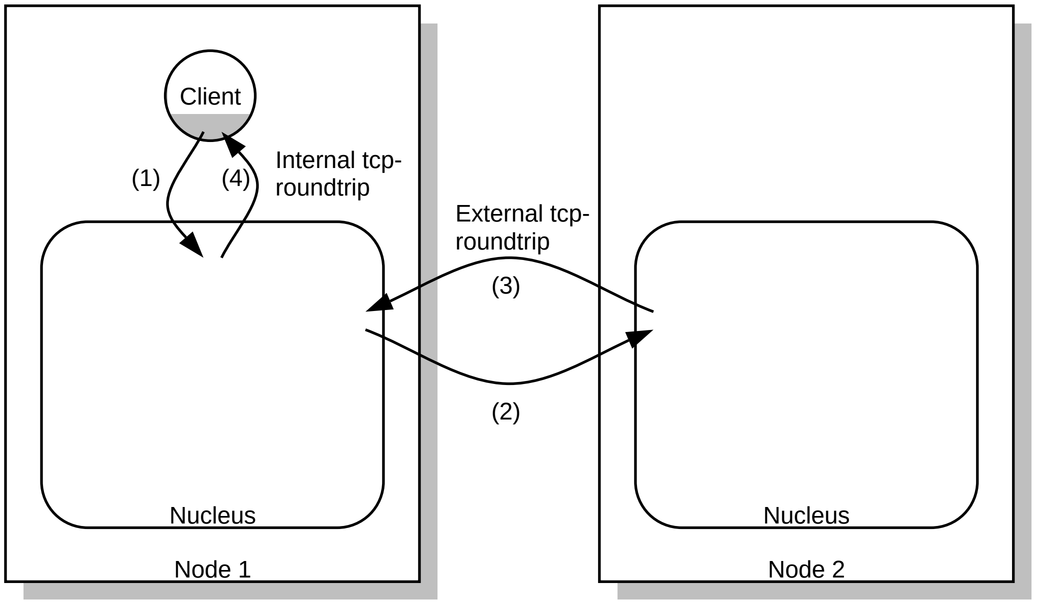

Figure A.3: The messages involved in an external tuple access. The numbers given as (x) describes the message sending sequence.

The tcp roundtrip time subtracted from the total time for a tuple operation is overhead caused by the runtime-environment in the clients and by the kernel. The figures shows that this adds up to an unacceptable amount of time $(T_{internal\_out} - T_{internal\_tcp\_roundtrip} = 0.96ms)$ spent on packing/unpacking buffers, looking up destination tuple space, hashing, lock management and matching.

We know from profilations of the nucleus (using gprof++) that there is plenty of room for optimizations. Especially dynamically memory allocation and deallocation is very time consuming: the `malloc()` and `free()` calls are well within the top 10 score of the routines where the kernel spends most of its time. Thus, a traditional sequential optimization task is required. However, as our timing measurements shows, communication is the dominant cost of tuple operations, and before optimizing the nuclei internally, the communication system should be reconsidered.

Tuple operations destined on external nodes are about twice as expensive as local tuple access. Since many match attempts are possible within this difference (3 ms), see Section A.3, distribution of tuples to reduce match overhead is unlikely to pay off, especially in a system with multiple tuple spaces where we expect fewer tuples per tuple space than a single-tuple-space system. Distributing tuples in a way such that tuples are placed close to either the producer or consumer, seems better. Consider for example a master/worker program, e.g., the parallel make tool of Section 2.3.6, where the master creates a local tuple space for its workers. The passive tuples could be stored at the master's node[3]. The master is producer of work-tuples and consumer of result-tuples. It would

[3]The passive tuples of a tuple space may be represented on a single “server” node for that tuple space. Logically

improve performance if, e.g., only workers would need to do external access for tuples. This example assumes that communication among workers is moderate compared to communication with the master.

### Comparison

In comparison with competing communication mechanisms like remote procedure calls, the performance of MTS-Linda is unfavorable: **out**/**in**-pairs are for both internal and external operations roughly 3 times as expensive as remote procedure calls. The client and server UNIX processes was placed on the same node for the internal test, and on 2 different nodes for the external test[4]. The underlying protocol for RPC in this test is tcp.

### What is Obtainable?

An interesting question is how much of the above unsatisfactory performance is an artifact of the prototype implementation, and how much is due to inherent properties of the MTS-Linda concepts? –A simple replacement of the tcp-communication between MTS-Linda clients and local nucleus with shared memory[5] would reduce the tuple access with

$$T_{internal\_tcp\_roundtrip} - T_{shared\_memory\_roundtrip} = (2.13 - 0.57)ms = 1.56ms$$

giving a access times for an internal **out** at $(T_{internal\_out} - 1.56)$ ms = 1.53 ms. If we further optimize the 0.96 ms spent by the nucleus and runtime environment with 50%, internal tuple space access would be in the magnitude of one milli-second.

The shared memory test shows another interesting result. Of the 0.57 ms shared memory roundtrip time only the 70 $\mu$ s is due to the copy of the $2 * 128$ bytes[6]. The remaining is due to process (context) switching. This is thus quite costly.

Obtaining significantly better access times would imply a larger re-organization of the prototype implementation. The communication between MTS-Linda clients and local nucleus could be replaced be kernel traps, or the entire tuple space could be placed in shared memory. In this case access would need to be performed by trusted code in the runtime environment. The internal communication would then be neglectable. However, the current configuration has an attractive advantage: the kernel is very robust to failures and crashes of its client processes. Such failures are detected on the socket for that client, and the kernel simply discharges future messages for

however, the tuple space still spans several nodes to hold a distributed program.

[4]Before a remote procedure is accessible the server must be localized, and afterwards be detached. This adds a cost to remote procedure calls not included in the figures: localization internally takes 7.74 ms, and detachment 2.51 ms. For external procedures localization amounts to 10.00 ms and detachment to 0.50 ms.

[5]The shared memory roundtrip has essentially the same functionality as an internal tcp roundtrip, but the client and server processes communicates through a shared memory region, and synchronizes via signals.

[6]The shared memory roundtrip test uses signals to synchronize access to shared memory. Using shared memory semaphores instead of signals actually produces a 200 $\mu$ s worse result.

that process. Other clients are unaffected. However, when it comes to a nucleus chrash, the whole system will come to a stop, and eventually be shut down.

To reduce the network cost of accessing external tuples it would be necessary to implement a low-level tailored communication protocol for MTS-Linda. As candidate for the "cheapest possible" network communication is the remote procedure call mechanism found in the Amoeba distributed operation system, [85]. A remote procedure call in Amoeba costs about 1 ms. Using shared memory communication internally, and the Amoeba figure instead of the $T_{external\_roundtrip}$ externally would give an external **out** time of:

$$T_{external\_out} - T_{internal\_tcp\_roundtrip} - T_{external\_tcp\_roundtrip} + T_{shared\_memory\_roundtrip} + T_{amoeba_rpc} = 2.66ms$$

Again, including the 50 % optimization of the nucleus gives an external **out** time of about 2 ms.

In conclusion, significant improvements the prototype performance is possible, but requires considerably optimization and re-organization of the prototype implementation.

## A.3 Match-overhead

In the tuple-exchange test only a single tuple existed in the tuple space, implying a modest match overhead. However, testing many potentially matching tuples may be costly. The next test, illustrated in Figure A.4, inserts 1000 tuples with identical tuple-signature in the tuple space and then **rd**'s them again.

Figure A.4 shows the **rd**-time as a function of the number of tuples needing to be tested. The gradient of the graph shows the time per match attempt: 14.25 $\mu s$. As noted in the foregoing section, additional fields cause additional match overhead. Note that the gradient was plotted by eye-fit, not by statistical linear regression, so it may not be entirely accurate.

The graph's intersection with the y-axis at 3400 $\mu$s is the constant communication time imposed on all tuples, disregarding the number of potentially matching tuples.

Unless many potential matching tuples (tuples located in the same bucket) exist in a tuple space match overhead is unimportant.

## A.4 Process Creation

Management of processes (creation, termination and scheduelling) adds to the execution time of distributed programs. The cost of process management affects how processes are used at the programming level: If processes are cheap, they can be used plentiful as a logical structuring tool, but if processes are expensive they are likely to be used sparsely. Further, when processes are used to speedup execution, the parallel execution must counterbalance the cost of creating the processes; the parallel program will otherwise run slower than the sequential version.

```
Timer match[1000];                  // an array of 1000 timers

for(int i=0;i<1000;i++)             // insert 1000 tuples into tuple space
    WorkSpace.out(i);

for(int j=0;j<1000;j++)             // perform the test 1000 times
{
    for(i=0;i<1000;i+=10)           // read each 10th tuple in the interval 0 to 1000
    {
        match[i].start();           // start timer for meassure point i –
                                       means testing i tuples
        WorkSpace.rd(i);            // read the tuple
        match[i].stop();            // stop the timer for this meassure poins
    }
}
```

Figure A.4: The program used to test match overhead.

This motivates timing measurements of process creation and termination in MTS-Linda. The test program of Figure A.6 is used to determine the process management cost in MTS-Linda.

The UNIX internal process creation time is measured as the time taking a `vfork(); exec(); waitpid();` command sequence to complete. Remote processes are created by doing a `rexec()` library call, which performs an RPC to the remote-execution-daemon—this is the most direct way we know of to start remote processes in UNIX. rexec avoids the overhead of the rsh (remote shell) command which reads a set of record files, e.g., `.cshrc` for C-shell, before it invokes the actual command. The direct start of processes makes the UNIX figures comparable with those of the MTS-Linda prototype. All tests starts an immediately terminating process, as illustrated in the MTS-Linda test program of Figure A.7

Process manipulation in MTS-Linda is about 45 % slower compared to UNIX when the process is created at the same node. However, when the process is started at another node MTS-Linda is more than twice as fast. As a matter of sincerness, UNIX external process creation involves authentication: the `rexec()` parameters includes a user-name and a password-string[7]. Further, it should be possible to implement a remote execution facility that would be faster than the `rexec()` call; a call to a remote procedure in a server to fork off the new process would be faster. In this case UNIX would be faster than MTS-Linda.

The MTS-Linda processes runs as UNIX heavy-weight processes and are therefore costly to create

[7]On the other hand, to be certain that the external process have terminated it is necessary to do `read()` calls on a socket (returned by the `rexec()` call) until it indicates that the connection is closed. This wait is not included in Figure A.7. Waiting results in a external process creation/termination time of 3323 ms.

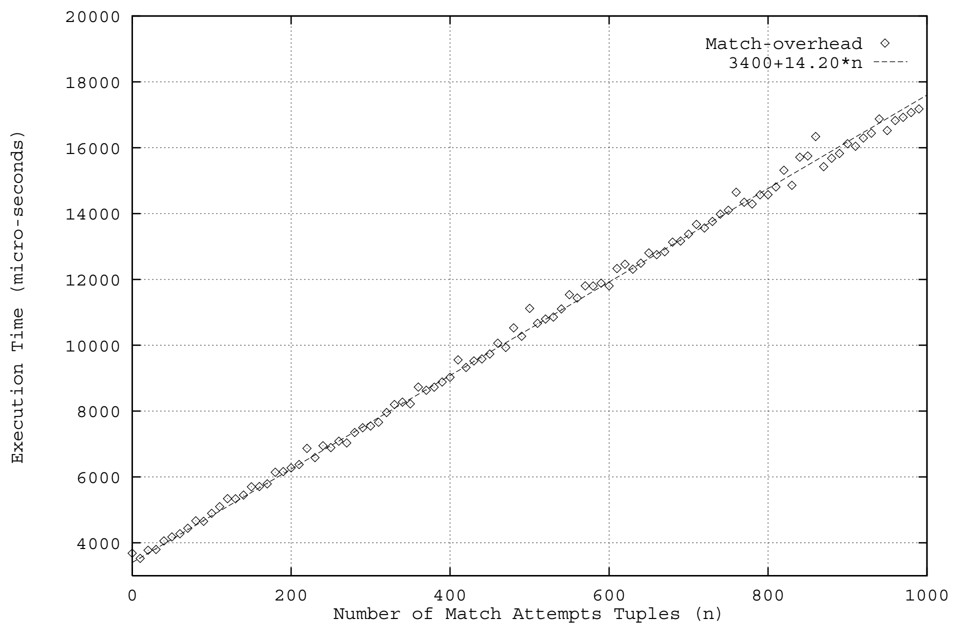

Figure A.5: Illustration of match overhead. Execution time as function of the number of tuples in tuple space. The program used to meassure this is in Figure A.4.

```
process int test(int param)          // A null-process which immediately terminates
{ return param;}
...

Timer p_time;

for(int i=0;i<100;i++)               // repeat test 100 times
{
    p_time.start();                  // start timer
    eval(test(i));                   // Fork off process
    in(i);                           // wait and retrieve termination result
    p_time.stop();                   // stop timer
}
```

Figure A.6: Test program for process creation/termination in MTS-Linda.

| | Eval (ms) | UNIX (ms) |
|---|---|---|
| internal | 238 | 193 |
| external | 402 | 855 |

Figure A.7: Performance results of process creation MTS-Linda and comparison with UNIX. Time units are in milli-seconds.

and schedule, but compared to UNIX we are satisfied with the performance of MTS-Linda.

## A.5 Creation and Deletion of Tuple Spaces

Manipulation of a tuple space as a whole is a novel feature, and we therefore interrogate the performance of tuple space manipulation. In this section we focus on their creation and destruction as a measure of basic tuple space manipulation times, and in Section A.6 take a look at the cost of copying and moving contents of entire tuple spaces.

In the prototype tuple spaces are distributed among all nodes; this applies for both shared and local tuple spaces and tuple spaces residing as fields of tuples. We found that this was the most general and most challenging case to design and implement.

Creation of a tuple space involves the following steps in the prototype: The client sends a create-tuple-space-coordinator message to its local nucleus. The receiver nucleus creates a tuple-space-coordinator object[8], which handles the global information about a tuple space—whether it is locked (the tuple space is already being manipulated) and whether it is active or passive (the tuple space contains active tuples). The nucleus then broadcasts the create-operation to all participating nuclei by sending each a create-tuple-space-slave operation. They update their tuple space hierarchy tree with the new information and acknowledges the initiator nucleus that the operation has been carried out. When the initiator has received all acknowledgments it acknowledges the awaiting client process. The delete operation works similarly, but before destruction may take place the destructing nucleus consults the coordinator to wait for the tuple space to become passive to ensure future semantics.

Thus, the create operation uses

$$2 * (internal\_tcp\_roundtrip + (N-1) * external\_tcp\_roundtrip)$$

number of tcp-messages, where $N$ is the number of nodes participating in the system.

The test program is outlined in Figure A.8, and the results of its execution varying the number of nodes of the system is shown in Figure A.9.

[8]Currently, the prototype creates the coordinators on the same node as the create-operation was received, but the design and implementation allows this to take place at any node. If the coordinator is being located on a external node a cost 2 of additional messages exists.

```
Timer ts_create, ts_delete;          // test timers
for(int i=0;i<100;i++)               // repeat test 100 times
{
    {                                // A new scope starts here
        ts_create.start();           // start create timer
        LocalTupleSpace A;           // create local tuple space
        ts_create.stop();            // stop create timer
        ts_delete.start();           // delete timing begins
    }                                // scope ends here, so tuple space A is destructed
    ts_delete.stop();                // End of delete timing
}
```

Figure A.8: Test program for creation and deletion of tuple spaces.

| | Create (ms) | Delete (ms) | Create file (ms) | Delete file (ms) |
|---|---|---|---|---|
| 1 node | 3.78 | 3.66 | 33.28 | 27.02 |
| 2 nodes | 7.25 | 7.03 | | |
| 3 nodes | 8.13 | 8.06 | | |
| 4 nodes | 9.41 | 9.31 | | |
| 5 nodes | 10.94 | 10.67 | | |
| 15 nodes | 28.26 | 28.29 | | |

Figure A.9: Performance results of tuple space creations and deletions. A tuple space is distributed to all nodes in the system. The table shows creation and deletion time as function of the number of nodes participating in the system.

Creation of a single-node tuple space is $(3.78 - 3.09 = 0.79)$ ms slower that an **out**-operation, and compared to the involved communication (an internal tcp-roundtrip), the communication amounts to 56%. The computational burden overhead of creating a tuple space is thus larger than manipulation of simple tuples.

Although, creation is about 0.2 ms slower than deletion, the operation's cost is, as expected, approximately same since both involve the same number of messages. The difference is probably due to creation of the coordinator object.

The times for creation and deletion increases, but less than linearly. This is because the initiator nucleus broadcasts (simulated by message passing) the operation-message to the receiver nuclei, and first afterwards awaits their acknowledgments. The kernel thus employs the possible parallelism when manipulating tuple spaces[9].

No obvious features of UNIX is comparable with tuple spaces, but comparison with manipulation of files is a possibility, and figures for creating and deleting files (via NFS) is included in figure A.9. The comparison is not entirely fair since the manipulation of directories implicates disk access, whereas tuple spaces does not. If the comparison is bearable, a MTS-Linda kernel with few nodes challenges UNIX.

## A.6 Copy and Moving tuple space contents

This test shows the cost of copying and moving tuple space contents as a function of the amount of tuples in the source tuple space. The test program, shown in Figure A.10, performs `contents_copy` and `contents_move` operations on a tuple space with a varying number of tuples.

The contents are moved from tuple space $A$ to tuple space $B$, and is then copied back to tuple space $A$. Tuple space $B$ is emptied by deleting and recreating it for each round. The operations are then repeated with an increasing amount of tuples in source tuple space. To average results the test is executed 20 times. When the test is performed on a 4 machine system, the tuples will be distributed equally among the nodes in the system[10].

The test-results of Figure A.11 shows that the cost increases linearly with the number of tuples in the tuple space. The cost increases because a tuple is inserted by a `contents_move`, or by a `contents_copy`, the nucleus is required to determine if any **in** or **rd** operations are waiting for a matching tuple.[11]

For both kinds of operations a basic initiation cost exists, which for 4 nodes is about 13 ms. This figure is very close to the time for creating and deleting a tuple space (9.41 ms and 9.31 ms). Hence, all tuple space operations have a constant initialization cost in the magnitude of 10 ms, and a constant cost per tuple in the tuple space. This is not surprising since all tuple space operations have the same communication pattern.

Thanks to the tuple space hierarchy tree the manipulation of nested tuple space does not require extra messages, and the performance of manipulating nested tuple spaces is thus limited to the cost of moving or copying tuple and tuple space parts within a nucleus.

The graph in Figure A.11 reveals a substantial overhead of copying tuples. As noted for the **rd**-operation, the implementation of tuple-copy is bad. This explains some the large copy-overhead, but not all. This could indicate a bug in the nucleus!!

A fertile optimization of tuple space manipulation would be to replace the tcp-simulated multicast

[9]The linear increase may be invalid for a large number of nodes, since operation-requests and acknowledgments then may compete for ethernet-access.

[10]We know this because we know how the kernel distributes the tuples among node.

[11]A potential optimization would be to check the special case of an empty destination tuple space. In this case the nucleus does not need to test for match when inserting each new tuple. This optimization, however, is employed in the prototype.

```
Timer ts_copy[250];             // Array of timers to hold copy meassurement-
                                  table
Timer ts_move[250];             // Array of timers to hold move meassurement-
                                  table
TSCap dummy;                    // we just need a capability for a field

for(int i=0;i<20;i++)           // perform test 20 times
{
    LocalTupleSpace A;          // creat one of the 2 tuple spaces
    for(int j=0;j<250;j+=25)    // sweep x-
                                  axis: 0 to 250 tuples per node step 25
    {
        LocalTupleSpace B;      // create another tuple space

        //Move test
        ts_move[j].start();     // start timer for y-axis points
        B.contents_move(&A);    // move contents from A and insert into B
        ts_move[j].stop();      // stop timer

        //Copy test
        ts_copy[j].start();     // start timer (y-axis points)
        A.contents_copy(&B);    // copy contents of B back to A
        ts_copy[j].stop();      // stop timer

        for(int k=0;k<25;k++)   // insert 4*25 tuples more in A -
                                   25 at each node!
        {
            A.out(i);
            A.out('c');
            A.out(42.42);
            A.out(dummy);
        }
    }                           // Tuple space B is deleted here -
                                   to be ready for a fresh round
}                               // Tuple space A is deleted here -
                                   to be ready for a fresh round
```

Figure A.10: The test program for examining the cost of copying and moving contents of a tuple space.

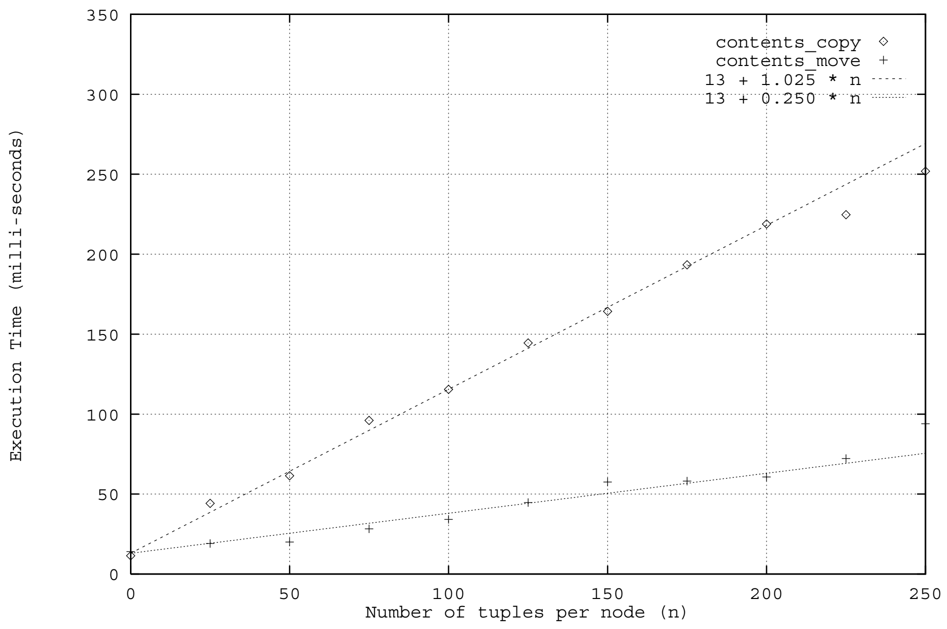

Figure A.11: Cost of `contents_move` and `contents_copy` as a function of the number of tuples in tuple space. The Test was performed on a 4 machine system. All machines are involved in an operation because the tuple spaces are distributed among all machines.

protocol with protocols designed for this. The ISIS project [9] reports a performance of 350 causally ordered multicastings per second for a 4 node process group, i.e., 2.86 ms per multicast. Employment of these protocols could improve the performance of tuple space manipulation significantly.