MS IN COMPUTER SCIENCE PROGRAM
INDEPENDENT STUDY ABSTRACT

Student Name: Andrew B. Sudell
Project Title: Design and Implementation of a
Tuple Space Server for Java
Advisor: Edward J. Segall
Date: 10 December 1998

Abstract

Linda is a language for coordinating parallel processes via a Tuple Space shared memory. The Linda model is examined as well as issues involved in implementing a Java based Tuple Space Server. Finally a prototype of a Java based Tuple Server is developed addressing the issues of on-line optimization, segmentation of the Tuple Space and the adaptation of Linda semantics to Java.

MS IN COMPUTER SCIENCE PROGRAM
INDEPENDENT STUDY EVALUATION FORM

Part 1.
Student Name: Andrew B. Sudell
Semester of Independent Study Registration: Spring 1998
Date of Completion: 10 December 1998
Project Title: Design and Implementation of a
Tuple Space Server for Java
Advisor Name: Edward J. Segall

Part 2.
Grade:

Advisor Evaluation:

Advisor's Signature Date

Contents

Chapter 1

Introduction

Linda is a parallel processing coordination language developed largely by the Linda Group at Yale University [Lin], which is a popular system for parallel and distributed computing. However, traditional Linda implementations have been embedded in host languages via dedicated compilers and preprocessors. Optimization has been, by and large, via static code analysis. This use of pre-compilation has been viewed by Carriero and Gelernter as a distinguishing feature of Linda.

> The "language" in "coordination language" distinguishes a system like Linda from a coordination *library* — for example, a message passing library like PVM. [CG93]

However, Linda implementations have been built [Seg93] avoiding the need for pre-compilation and providing for online optimization.

Java [AG96] is a new and rapidly evolving programming language with built in support for networking and concurrent programming. So, it is interesting to adapt Linda to the Java Language, while avoiding the need for pre-compiling so as to take advantage of the concurrent efforts of others to enhance and improve Java systems. In fact, a number of commercial systems [Mic98] [Sys] are being developed to facilitate building distributed systems in Java via a Linda-like Tuple Space shared memory. While these systems are strictly speaking not Linda implementations, their use of Linda-like tuple spaces as a model for anonymous interprocess communication among distributed Java applications lends some credibility to our goals.

In this paper, we examine the Linda language in some detail, as well as some of the parallel computing models that can be (and often are) implemented in Linda. We then examine how these characteristics effect the design and optimization of the Tuple Space. Finally we develop a prototype of a Linda implementation in Java which takes advantage of online optimization of the Tuple Space.

Chapter 2

The Linda Coordination Language

2.1 Tuple Spaces

There are a number of ways one could build a parallel processing system. The most apparent way, at least to software practitioners, is to build shared memory machines with many processors. In this way, the semantics of data access are unchanged from those of conventional sequential programming, and the programmer need only deal with the issue synchronizing access to that data However, this approach is limited to the availability of such solutions from hardware vendors and may only be applied to such systems. In general, such systems tend not to be inter-operable and scaling is limited to the availability of specific hardware from specific vendors. Moreover, software implemented on such specialized systems is often not easily ported to newly available but incompatible systems.

Assuming that one opts for a software solution dealing with individual processors which do not directly share memory, perhaps the next most apparent solution is to pass data back and forth between processes on separate processors via a message passing system. In fact, such messages passing systems exist [PVM] [MPI] and are quite popular in the parallel processing world

Linda provides a third alternative called a Tuple Space shared memory. A Tuple Space memory consists of a number of tuples, that is ordered sets of typed values much like tuples in relational database theory, which live outside each process, but are accessible to each process. Unlike shared memory systems and like message passing systems, data must be copied between the individual processes and the tuple space. Like hardware shared memory systems, and unlike message passing systems, shared data is accessed directly and anonymously by each process, and processes do not communicate directly with one another.

Linda's tuple space model is interesting to parallel and distributed processing because over time experience has shown [CG93] that Linda is at least competitive with message passing in speed, and in some ways more expressive than message passing. More over, since all communication is between the process and a single virtual entity (the Tuple Space), the anonymous nature of interprocess communication in Linda is more amenable to adaptive parallelism than message passing.

2.2 Linda Operations

The Linda language consists of only six simple operations by which the process interacts with the Tuple Space. These operations are `out`, `in`, `rd`, `inp`, `rdp`, and `eval`.

The out operator places a tuple into the tuple space. For example, `out(5)` places a tuple consisting of a single integer, 5, into the tuple space. Likewise `out("foo", 12.7)` places a tuple consisting of the string "foo" and the floating point number 12.7 into the tuple space. In addition to constants, variables may also be used, so if the variable `s` is the string "x" and `i` is the integer 10, then `out(s,i)` is equivalent to `out("x", 10)`.

The in operator is the dual of out and removes a tuple from the tuple space. Unlike out, in is a blocking command and if no matching tuple is found, will block until another process outs a matching tuple So, `in("x", 10)` would remove the tuple which had been "outed" before. In addition to constants and variables, Linda support the notion of a formal variable. Formals may match any value of the same type and have the side effect of setting the value of the formal. So, `in("x", ?i)` could match either the tuple `("x", 5)` or the tuple `("x", 6)`, and have the side effect of setting `i` to either 5 or 6.

The rd (or read) operator is similar to the in operator. rd will block until it finds a matching tuple, however it will not remove the matching tuple from the tuple space.

In addition to the blocking queries, in and rd, Linda provides non-blocking queries inp and rdp (in predicate and read predicate). These operators act similar to in and rd, but return immediately with a true or false status indicating if a matching tuple was found.

Finally, Linda provides eval (evaluate) which is similar to out, but capable of creating processes. In addition to supporting actual and formal variables, eval supports the use of functions which return values. For each tuple element which is a function, eval will create a new process to evaluate the function. Once all functions have been evaluated, eval will place the resulting tuple into the tuple space. For example, if the function `f(x)` computes the factorial of `x`, then `eval(5, f(5))` spawns a new process to evaluate `f(5)` and then places the tuple `(5,120)` into the tuple space.

There are several fine points of Linda worth considering. First of all, since out is asynchronous, it is not guaranteed that the an outed tuple will be in the tuple space and available for matching either when the out operation returns or any finite time later. So in the following code

```
out("mytuple", 5)
```

```
x = rdp("mytuple", ?i)
in("mytuple", ?i)
```

While the in will block until the tuple (or some other matching tuple) arrives, there is no guarantee that the rdp will match the tuple even if no other processes exist. Thus the value of `x` is indeterminate. Second, while functions within evals ultimately return values, which result in tuples being placed into the tuple space, they may also create side effects. In fact the use of eval to create processes, which in turn perform other Linda operations, is not only acceptable but commonplace. Without such side effects, the tuplespace could not be used to provide for parallel processing unless some other external method of process creation were used. Finally, while a formal value can match any similarly typed value, it must match an actual value, since one side effect is to update the formal with the matched value.

2.3 Building Parallel Programs with Linda

Before considering how to implement Linda, it is probably prudent to consider if and how Linda allows us to write parallel programs. One reasonable taxonomy of parallel programs [CG89] divides parallel programs into three broad classes: *result parallelism*, *agenda parallelism*, and *specialist parallelism*. In a result parallel program, the program is structured around a set of required results and individual processes are responsible for each of results. In an agenda parallel program, the problem is broken down into an agenda of tasks, often called a "bag of tasks". A number of non-specialized worker processes are spawned, who each continue grabbing tasks, working them and returning the result, so long as outstanding tasks exist. In a specialist parallel program, a number of specialized processes are started. Each can perform some part of the problem and must pass the intermediate results onto another specialist until the problem is solved.

While any particular actual program is likely to demonstrate aspects of each form of parallelism, it is sufficient to show that we can accomplish each using Linda.

Perhaps the easiest form of parallelism to implement in Linda is agenda parallelism. In this case, one need only implement a bag of tasks as a multi-set of task tuples. Processes can add tasks to the bag by executing `out("task list", task-description)` and workers can fetch tasks from the bag by executing `in("task list", ?task)`.

Without much more difficulty, one can implement parallelization for specialization. It's most natural to think of using message passing to do so. However, one can easily implement a message queue in Linda, achieving the same effect. A simple queue of messages for a single process might consist of a series of tuples `("queue", 0, message-0)`, `("queue", 1, message-1)`, `("queue", 2, message-2)`, etc. Another tuple is used to keep track of the tail of the queue, `("queue-counter", 2)`, where the queue counter is the index of the last message added to the queue. Processes may add messages to the queue by incrementing the queue counter and adding new tuple to the queue.

```
in("queue-counter", ?n)
n = n + 1
out("queue", n, message)
out("queue-counter", n)
```

Note that if only one queue counter tuple exists, the fact that in both blocks and removes the tuple effectively serializes access to the counter. The receiving process can then simply in successive messages, assuming that a blocking read is acceptable. Alternately, the reader could perform a non-blocking poll via a series of inp operations until a message arrives.

Finally we can consider result parallelism. In this case, the result takes the form of some arbitrary data structure or structures. Without too much imagination, we can see that if we can implement the data structure in a conventional programming language we are likely to be able to implement it with some combination of arbitrarily named variables and arrays. Simple variables can be implemented as tuples with the variable name as the first element in the tuple. Distributed arrays can be implemented in Linda by a series of tuples containing the array name, the array indexes and the value as elements. For example, one can implement a n by m matrix as a series of $n \times m$ tuples of the form `("matrix-name", i, j, value)`, where `i` and `j` are indexes into the matrix. In fact, just about any data structure that can be implemented in a conventional language can be implemented in Linda, as well as a few that have no conventional equivalent [CGL85].

Chapter 3

Designing a Tuple Space Server

3.1 Matching Tuples

The most basic operation of a Tuple Space server is matching tuples. The Linda operators can be divided into two basic sets, those that place tuples into the tuple space (`out` and `eval`) and those who remove or copy tuples from the tuple space (`in`, `inp`, `rd`, and `rdp`). In considering the interaction between a Linda process and the Tuple space, it is useful to think of `out` and `eval` as placing tuples into the tuple space, but `in`, `inp`, `rd`, and `rdp` placing *anti-tuples*, that is tuple patterns or *templates* which exists in the tuple space until a matching tuple arrives. Thus we can speak of "tuple matching" as either a tuple rendezvousing with a existing anti-tuple or as a anti-tuple rendezvousing with an waiting tuple, depending on the order of operation. In either case, the semantics of matching are the same.

In on order to match, then, a tuple/anti-tuple pair must first of all must in fact be a tuple and an anti-tuple, i.e. they must differ by *polarity*. Secondly, they must have the same number of elements. This measure is referred to as the *arity* of the tuple. Third tuple members are strongly typed by definition, so the type of each element in each tuple of the pair must match. So, for an example, a tuple who's *type signature* is `(int, int, string)` can only match an anti-tuple who's signature is also `(int, int, string)` and not an anti-tuple who's signature is `(int, string, int)` or `(string, float)`. Finally, the individual parameters of the tuples must match.

Matching the individual parameters, however, is complicated by the existence of formal parameters, since a formal can match any actual value of the same type, but can never match another formal. Either tuples or anti-tuples may contain formal parameters as well as actual values. However, in practice, formals rarely if ever appear in tuples, since they are to some intent and purposes the equivalent to wild cards and thus serve well in query like operations but poorly in outs and evals. There are potentially a number of ways to match parameter values, but a framework exists [Seg93] which deals with the issue in a straight forward and expeditious manner.

Consider the subset of all tuples and anti-tuples with the same arity and signature. Assign each a *actual/formal pattern*, consisting of a bit field of arity bits such that the bit is set for a actual value in the corresponding position or unset if the corresponding parameter is a formal. For example the pattern for the tuple `("foo", 1, ?a)` is 110 and the pattern for `(?a, ?b, 12, ?d)` is 0010. A tuple/anti-tuple pair can be said to be *pattern compatible* if the inclusive bitwise or of their patterns yields a pattern of all ones. So, for example consider three tuples `t1`, `t2` and `t3`. `t1` is `(1, 2, ?i)` and has a pattern of 110. `t2` is `(1, ?i, 3)` and has a pattern of 101. Finally `t3` is `(1, ?j, ?i)` and has a pattern of 100. `t1` and `t2` are pattern compatible since the bitwise or of their patterns is 111. However, neither `t1` nor `t2` is pattern compatible with `t3` since the resulting patterns are 110 and 101, respectively.

Once we know that the pair is pattern compatible, we must still compare the individual parameters for equity. However, there is no desire to attempt to compare any parameters in which a formal is matching an actual, since this is an unnecessary step and in some implementations my involve attempting to compare potentially undefined values. Rather we need only compare pairs of parameters if both are actual values. So, we can build a *must match pattern* by taking the bitwise and of the two actual/formal patterns. Only those parameters for which the corresponding must match pattern bit is set need be compared. Continuing the example, the must match pattern for `t1` and `t2` is 100, so only the first parameters of each need actually be examined. The list of parameters for which the must match pattern is set, is therefor referred to as the *must match values*. In conclusion, a tuple and a template match if

- they differ in polarity
- they have the same arity
- they have the same type signature
- their patterns are compatible
- and their must match values are equal.

3.2 Partitioning the Tuple Space

Even with a simple and efficient way to match tuples, for many programs, there will be many tuples to match. Sequentially matching all tuples or anti-tuples in the tuple space would be a huge waste of time and largely defeat the purpose of parallelizing the program in the first place. Thus it is desirable to examine patterns of tuple matching and attempt to minimize the number of tuples examined in any match.

As we have seen above, No tuple/template pair can match if they differ in arity or signature. Nor can they match if they do not differ in polarity. It has been noted [Car87] that this allows us to partition the tuple space into a number of orthogonal subspaces. For any given operation, only the single subspace whose arity/signature match need be examined for any given operation. However, it is possible to limit our search further. Consider a program which exports the following tuples

```
out("foo", 5)
out("bar", 12)
```

and then later retrieves them as

```
in("foo", ?x)
in("bar", ?y)
```

Assuming the rest of the program acts in a similar fashion, it is clear that these operations also consist of separate sets of tuples (or *out sets*) for which there is no overlap between their potentially matching templates. In fact, many common patterns of Linda usage [CGL85] [CG89] create many such patterns of tuple usage. This presents an opportunity to consider further subdividing the tuple space.

Partitioning the tuple space by constants presents us with both a challenge and new opportunity. In classical Linda implementations, host languages were extended via pre-processors to embed Linda operations into the language. This allowed for static analysis of the code to detect the use of constants [Car87]. However, out goal is to use only runtime optimizations in our Java implementation. In most cases, the use of constants by the programmer is quite intentional. Thus some Linda implementations, including that on which we are most heavily based [Seg93], allow the programmer to add a *constant* qualifier to the type of a parameter. Thus allowing us to further partition the tuple space if desired, following the programmer's hints. This additionally affords two additional potential optimizations. By giving control of this partitioning to the programmer, some level of user control of the optimization is allowed for fine tuning of particular applications. Secondly, if we partition the tuple space by constants as well as by arity/signature, then all tuples or templates in a subspace will naturally have the same value in that parameter. Thus we can drop that parameter from the must match pattern.

3.3 Searching Subspaces

Given that we can reduce tuple matching to just comparing tuples within a limited subspace, one would still not want to simply perform a linear search of the subspace. Therefore, we should consider how one might organize the subspace, in order to make tuple matching quick and efficient.

There are a number of cases in which the out set can be quite simply organized and easily managed. As the simplest case consider a out set in which all parameters are constants. One might, for example implement a mutex using `in("mutex")` to acquire a lock and `out("mutex")` to release it. In this case one could ignore the tuples altogether and simply track the number of outstanding tuples in the space in a single integer, incrementing the count for `out`s and decrementing for `in`s and blocking `in`s when the count reached zero. That is, the out set could be reduced to a semaphore.

Now consider an out set in which one or more parameters are non-constant, but where all `in`s match formals for all non-constant parameters and all `out`s have only actuals for non-constant parameters. There would still be no need to match any actual values, and the managing the out set presents little more difficulty. Consider the out set defined by the operations `out("foo", i)` and `in("foo", ?j)`. This could be implemented without too much trouble as a queue. `out`s would append the non-constant value or values to the tail of the queue. `in`s would delete the head of the queue or block if the queue is empty.

Once the out set contains patterns in which we must match one of more actuals, then one must finally index the out set in some way. If only one parameter must ever be matched, then this parameter is the logical choice for a key in a index or hash table of tuples. If a larger number of parameters must always be matched, then a composite key can be used. However, a composite key works only if the must match pattern is a constant. Consider the case in which one process creates a tuples for the form `out("foo",i,j)`. Other processes create two templates of the forms `in("foo", 5, ?i)` and `in("foo", ?j, 7)`. In this case, called a *hybrid set*, no single key can be used to index both searches.

3.4 Multi-Key Search

Since we can not take advantage of simple solutions such as semaphores and queues, which require static analysis of the Linda program, we need to consider other schemes for managing the out sets which are amenable to online optimization. In addition we prefer to use a solution which addresses the issue of hybrid sets. Fortunately, such a solution exists in Multi-Key Search [Seg93] and has been implemented previously in Java [Des98].

Multi-key Search (MKS) is essentially an adaptation of secondary indexes. When matching a template against the tuples in an out set (the same logic will also work for matching tuples against waiting anti-tuples), one must consider all tuples whose actual/formal patterns are compatible with the template's pattern. For each of those patterns, one must consider the value of all parameters in the must match pattern for the combination of the template's pattern and the tuple pattern under consideration. Therefore if we build a dictionary, in which each tuple is indexed once for each compatible pattern, and for which the key for that index is the composite of the compatible pattern, the corresponding must match pattern and the values of the must match variables of the tuple, we can find search the indexes of the dictionary for matching tuples in at worst n searches where n is the number of compatible patterns for the template. Since the tuple space is a multi-set, i.e. multiple identical tuples could exist, each index points not to a physical tuple but to a list of tuples with identical indexes. Upon removal the tuple is removed from the list, and the index node deleted only if the list is then null.

In many Linda programs only a few fixed patterns are actually used. At first blush, one might assume that MKS must bear a much higher overhead maintaining indexes than a scheme that benefited from static code analysis. However, this is not necessarily the case. Since the index for any given pattern is unneeded until a template arrives which has that pattern, once can defer creating the index until the first template arrives. Even for a out set with a small arity, this can produce a significant savings.

Consider an out set of arity 2. For all compatible template and tuple patterns, the must match patterns are given below.

	Tuple Pattern			
Template Pattern	00	01	10	11
00	-	-	-	00
01	-	-	00	01
10	-	00	-	10
11	00	01	10	11

So in the worst case, a tuple with two actuals would be indexed four ways and once using both parameters in the key. But if only the patterns 01 and 10 were ever used, no tuple would need more than two indexes. It is worthwhile to note that many of the indexes in use are in fact indexes on no fields at all. In that case the list of tuples with that index degenerates into the queue that we would have opted to use has we been using static analysis.

3.5 Distributing the Tuple Space

While our initial prototype runs on a single processor, or at least a single Java Virtual Machine which may its self be run on a multiprocessor, the whole point of Linda is parallel processing. Thus ultimately we will want the tuple space available from multiple processors.

There have been a number of schemes for distributing the tuple space used in various Linda implementations. Perhaps the simplest is to replicate the entire tuple space on each processor as was done in the S/Net implementation [Car87]. This means that any operations which change the state of the tuple space (`out`, `eval`, `in`, and `inp`, but not `rd` nor `inp`) must be broadcast to all nodes and synchronized. This can serve as a bottleneck in performance.

Another scheme used by some Linda and Linda-like implementations, including Laura [Tol92] is to build a rectangular grid of tuple servers. Each server is connected to an “out bus” which runs in one direction and an “in bus” which runs in the other. `out` (and `eval`) operations are broadcast on the the out bus to which a process' local tuple server is connected and the tuple replicated on each server on that bus. Likewise `in` (and `rd`) operations are broadcast to the local in bus and performed by all servers on that bus. Since the buses run orthogonally, every tuple and every template meet on exactly one node. At best a n processor system can perform $\sqrt{n}$ parallel searches.

Better yet is Hash to Rendezvous. If we divide the tuple space into subspaces based on out sets as a function of arity, signature, and potentially constant values, we know that any search need only involve a single subspace. We can assign each subspace to a processor based upon a hash calculated on the out set signature. In this way, we know *a priori* which processor is the only processor on which we need search. Tuples and Templates can then "rendezvous" on their assigned processor, via independent efforts. For any given match at worst three messages must be sent. First an original `in` that blocked, second an `out` that matches, and finally the return to the `in`. Only the processors where the operations are initiated and the processor holding the subspace need be involved. Thus in the best case, all processors may be involved in parallel searches at the same time, assuming that the number of out sets used by the application is large with respect to the number of processors and the hash function used results in a uniform distribution.

Chapter 4

Implementation Issues

4.1 Language Binding

The first issue that appears when adapting the Linda language to a Java class library is that the classic pre-processor implementations can hide a lot of detail behind the pre-processor and therefor their syntax is not bound as tightly by the host language. In our case, our language binding must not only feel like Java, it must be Java. This presented a minor issue in that while Java is capable of polymorphism, its primitive types are not. Therefore one could not support the primitive types directly without a plethora of constructors for tuples. In addition the need to support formal variables and to allow programmers to designate actuals as constants, required type modifiers beyond those provided for in Java. In the implementations on which we based this project [Seg93] [Des98], this was dealt with by passing all tuple parameters as strings which would be parsed to set the variables. As an alternative we opted to build a wrapper class, `LindaParam`, which dealt with type modifiers but required the use of Java's wrapper classes for integer and floating types. In addition to float and int, the `LindaParam` class also supports Java `String`s as well as `LindaFunctions`.

Tuples themselves are implemented in the classes `Tuple` and `Template`, both of which are implemented as subclasses of `AbstractTuple`, and contain an array of `LindaParam`s as shown in figure 4.1. Tuples are instantiated with a variable number of parameters, which is presently limited to four. One could, for example build the tuple `out("matrix", i, j, value)` as

```
out(new Tuple(new LindaParam(LindaParam.LINDA_CONSTANT,
                             "matrix"),
              new LindaParam(LindaParam.LINDA_ACTUAL,
                             new Integer(i)),
              new LindaParam(LindaParam.LINDA_ACTUAL,
                             new Integer(j)),
              new LindaParam(LindaParam.LINDA_ACTUAL,
                             new Float(value))));
```

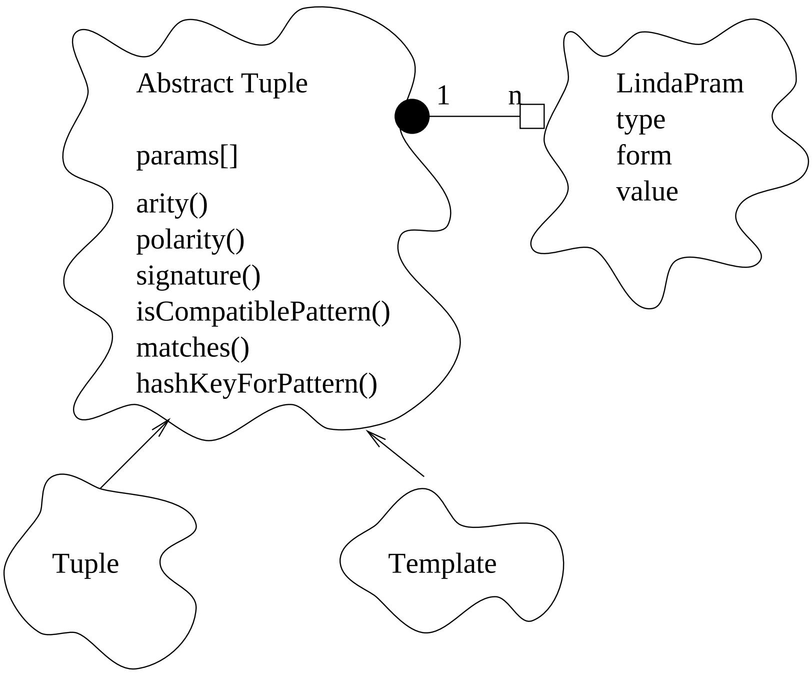

Figure 4.1: Tuple class hierarchy

Admitedly, this is a more awkward syntax than in an embedded Linda implementation.

The `AbstractTuple` class also implements the common methods to support the `TupleServer` and it's underlying `Subspace`, such as `signature()`, which returns the type signature of a tuple; `pattern()`, which returns the tuple's actual/formal pattern, `isCompatiblePattern()`. which checks if a pattern is compatible with this tuple's pattern; `matches()`, which checks a tuple for a match; and `hashKeyForPattern()`, which builds a hash key for the must match variables for a given compatible pattern. For the most part these are straight forward implementations of the algorithms discussed in the previous chapter.

The one method of `AbstractTuple`, which requires some mention is `hashKeyForPattern()`. In previous implementations of MKS, `AbstractTuple` directly calculated an integer used to determine the hash bucket, as a function of its parameters. In this implementation, native Java Hashtables are used, on the theory that unless they prove too inefficient, it is preferable to use native language feature. Thus, `hashKeyForPatterns` merely returns a "stringification" if its parameters to be used to form the hash key. Should the native hash algorithm prove insufficient, a subclass of `String` could be built to override the `toHash()` method.

Another issue was the assignment of values to formals upon the completion of `in`, and `rd` operations. Unlike a pre-processed implementation, in our online

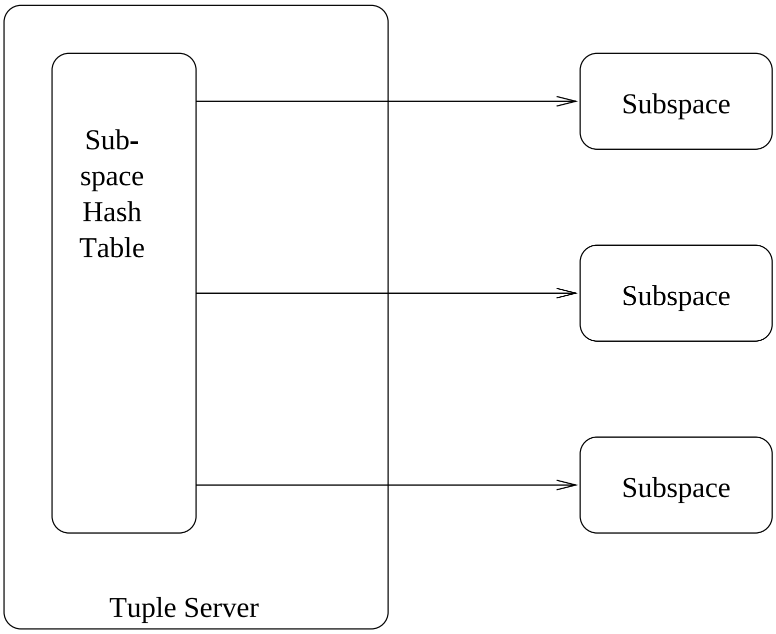

Figure 4.2: Tuple Server Data Structures

system one can not easily hide the implicit assignment. While it may have been possible to perform this via Java Reflection, we opted to simply return the matched tuple and allow the programmer to assign its value to the formal. Upon reflection, this does not inhibit the use of the system, though it does create some additional work for programmers.

4.2 Tuple Server Design

For the design of the `TupleServer`, its self, it was decided to partition into a series of SubSpaces based upon the signature and arity of the tuples. A single dictionary is kept with references to the individual `SubSpace`s as shown in figure 4.2. Most tuple operations were simply passed through to the `SubSpace`s, with the exception of `eval`, as discussed below. `SubSpace`s are created on demand and the dictionary is keyed by the type signature of the tuples in the subspace. `SubSpace`s are never destroyed.

The `SubSpace` is an implementation of the sequential MKS of [Seg93], using a monitor built with synchronized methods. Linked lists of all `Tuple`s and all `Templates` are kept, to enable indexing new patterns as they are found to be used. Two separate `Hashtable`s are maintained for `Tuple`s and `Template`s. The major structure of the `SubSpace` is shown in figure 4.3. Each hash is indexed by a string built of the pattern of a compatible anti-tuple or anti-template, the pattern of the `Tuple` or `Template` being indexed and the must patch values

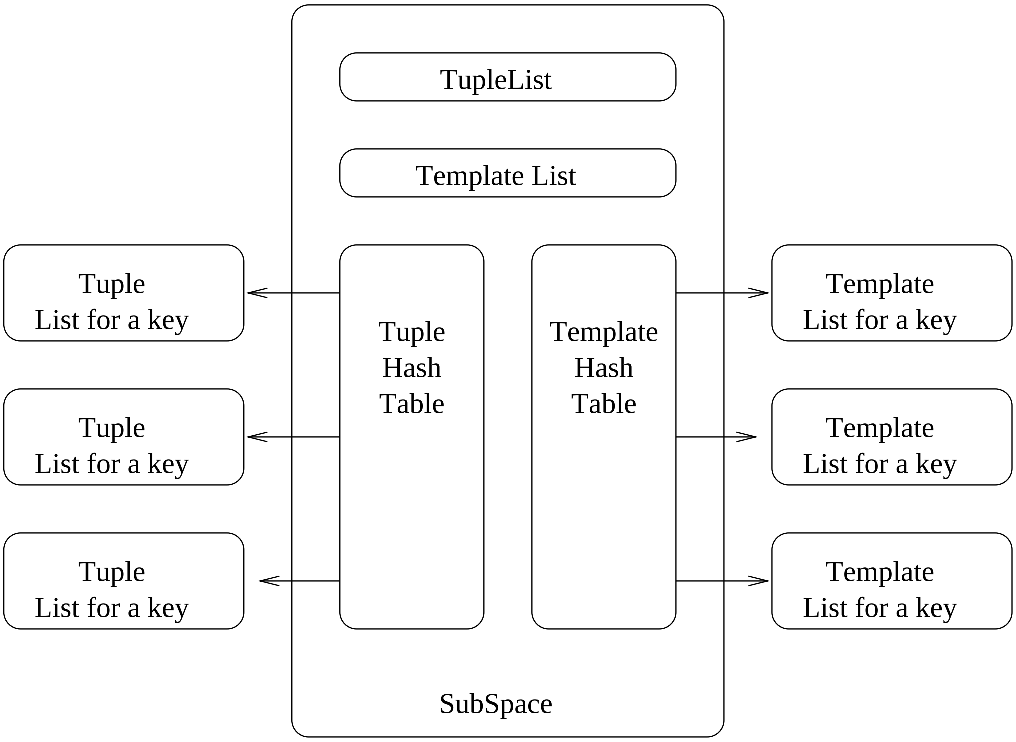

Figure 4.3: Tuple Subspace Data Structures

of the pair. For example, the tuple `("mytuple", 5)` would be indexed for the template pattern 10 as the string "10:11:mytuple". For the pattern 11, the same tuple would be indexed as "11:11:mytuple,5". For pattern combinations with must match patterns which are not full, many tuples may share the same keys. In addition it is perfectly acceptable for multiple tuples (or anti-tuples) to exist with identical values. Therefore the entries in the hash tables are linked lists of tuples and not the tuples themselves. Upon insert lists are created for new hash keys. Upon delete the tuples are removed from the lists, and the lists and keys deleted if the list is then empty.

One point in the implementation of `SubSpace` which is worthy of examination is that of synchronization. For operations that consume tuples such as `in` and `rd` (or their predicates), it is possible for the operation to block. In this case the process must await the arrival of a matching tuple. Perhaps the simplest way to deal with this in Java is to have the thread performing the operation `wait` after adding it's `Template` to the `SubSpace`. This raises the question of how to reawaken the process. Unfortunately, Java provide only two forms of `notify` in it's monitor implementation. The `notify` method awakens one, potentially randomly selected, thread waiting on a particular object. If synchronization is done at the level of an entire `SubSpace`, that may potentially be a `Template` awaiting a different `Tuple`. The only other option is to `notifyAll`, which awakens *all* processes waiting on the object. In our implementation, we chose to use `notifyAll`. This potentially creates a "mad rush" of processes rechecking for tuples each time an arriving tuple does a notification. For this reason, the `out` only performs the notification if at least one matching `Template` exists. However, for subspaces with large numbers of outstanding anti-tuples, this could become a performance problem.

There are two interesting ways to expand the functionality of the server significantly. First of all, a RMI tuple server could return remote references to subspaces on foreign processors with little extra complexity. Coupled with hash to rendezvous, this could be the basis for a distributed implementation. Secondly, the subspace searches could be reworked to use parallel MKS [Seg93] and more fine grained locking could be used. The optimization and distribution of the Tuple Space Server are the most obvious areas for future enhancement.

4.3 Processes

In the classic Linda model, the preprocessor inserts code that runs at startup to create the tuple space and then invokes a fixed user function. Additional processes are then created via `eval`. In a similar bent, we have provided two interfaces for creating processes.

Initial setup is done by the `main` method in `LindaMain`, which initializes the `TupleServer` and starts the program's main thread. The abstract class `LindaProgram` provides a frame work for Linda Programs which `LindaMain` can start. A class extending `LindaProgram` inherits a `TupleSpace` from it along with methods to invoke Linda operations on the space. In addition, an extending class must implement the method `lmain` which is called after setup with the program's arguments to begin execution of the user's program proper.

However, unlike the main process, processes spawned by `eval` must return useful parameters to the resulting tuple. Since in is not practical to pass the reference to a function as part of the tuple in Java, the interface `LindaFunction` provides the necessary syntactic sugar. When executing `eval`s, the `TupleServer` creates a thread for each `LindaFunction`, and creates a tuple for `out` once the threads have finished. So, by creating tuples which contain `LindaFunction`s and evaluating them, concurrent sub-threads can be created. Since it is executed as a Java thread, `LindaFunction` is its self an extension of Java's `Runable` interface.

Chapter 5

Conclusion and Observations

The Linda coordination language provides a useful framework for building parallel programs which can be implemented without specialized hardware and is competitive with message passing systems. Using Mult-Key Searching, and partitioning the tuple space we can build a usable Linda implementation without relying on pre-compiling for embedding nor the use of static code analysis for optimization. The potential exists to build a truly scalable system by implementing Hash to Rendezvous. By adapting the Linda language slightly, we are able to adapt our implementation to Java, Therefore once a true distributed implementation exists, it can take advantage of the availability of Virtual Machines on most new high performance platforms to create a easily expandable and highly scalable system.

Along the way of doing this independent study, we had the opportunity to learn a great deal about Linda as well as a fair bit about parallel programs in general. By examining the issues of tuple matching and distribution, we delved deeper into issues closely related to the world of physical database design and layout, which has been a personal interest for some time. In addition, we had to opportunity to explore a number of interesting aspects of Java technology, though many, such as Remote Method Invocation and Reflection, ended up not being used in this project and would have needed a must more fully developed project to have been of use. The effort was interesting, none the less. Finally we had the opportunity to come to understand the notion of tuple space distributed shared memories, a concept we were completely unfamiliar with at the start of the project, and which in light of efforts such a JavaSpaces and T-Spaces, may be experiencing a resurgence.

Source code for this project, along with electronic copies of this paper are available at `http://renoir.vill.edu/~asudell/csc9020`.

Bibliography

[AG96] Ken Arnold and James Gosling. *The Java Programming Language*. Addison-Wesley, 1996.

[Car87] Nicholas J. Carriero. *Implementation of Tuple Space Machines*. PhD thesis, Yale University, Department of Computer Science, 1987.

[CG89] Nicholas Carriero and David Gelernter. How to Write Parallel Programs: A Guide to the Perplexed. *ACM Computing Surveys*, 21(3):323–357, September 1989.

[CG93] Nicholas Carriero and David Gelernter. Linda and Message Passing: What have we learned? Technical Report 984, Yale University, Department of Computer Science, 1993.

[CGL85] Nicholas Carriero, David Gelernter, and Jerry Linchter. Distributed Data Structures in Linda. Technical Report 438, Yale University, Department of Computer Science, November 1985.

[Des98] Atul A. Deshmukh. Implementation of Multiple Key Search (for Linda) in Java. Independent study, Villanova University, Department of Computer Science, 1998.

[Lin] The Linda Group. http://www.cs.yale.edu/HTML/YALE/CS/Linda/linda.html.

[Mic98] Sun MicroSystems. Javaspaces specification, 1998. http://chatsubo.javasoft.com/products/javaspaces/specs/index.html.

[MPI] Message Passing Interface. http://www.mcs.anl.gov/mpi/.

[PVM] Parallel Virtual Machine. http://www.epm.ornl.gov/pvm/pvm_home.html.

[Seg93] Edward J. Segall. *Tuple Space Operations: Multiple-Key Search, Online Matching and Wait-Free Synchronization*. PhD thesis, Rutgers University, Department of Computer Science, 1993.

[Sys] International Buisness Systems. T-Spaces. http://www.almaden.ibm.com/cs/TSpaces/.

[Tol92] Robert Tolksdorf. Laura: A Coordination Language for Open Distributed Systems. Technical Report 1992/35, Technical University of Berlin, Department of Computer Science, 1992.