Implementing Distributed Linda in Standard ML

Ellen H. Siegel and Eric C. Cooper

October 1991
CMU-CS-91-151

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

We have implemented the Linda model of shared distributed tuple space in a functional programming language, Standard ML. We use ML's flexible type system and pattern matching facilities to provide ML programmers with the basic Linda operations on tuples. No preprocessor is used, and no compiler changes are required.

We use separate ML modules to implement the Linda interface, operations on tuple space, communication of tuples over the network, and replication of tuple spaces. Our approach allows different compositions of these modules to be used to configure a system with either local or remote access to tuple space, and with either a centralized or distributed implementation of tuple space.

The resulting implementation of Linda in Standard ML offers an attractive way to separate the functional and the imperative portions of a distributed system. Individual processes can be written in ML in a pure functional style and the Linda shared tuple space can be used to interconnect the processes and maintain the state of the system.

This research was sponsored by the Defense Advanced Research Projects Agency and monitored by the Air Force Systems Command under Contract F19628-91-C-0128.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Keywords: programming languages, parallel and distributed systems, Linda, Standard ML

1 Introduction

We have constructed a flexible, expressive environment for implementing distributed systems by combining Standard ML's support for functional programming and flexible type system with the Linda model of parallel programming. We have implemented Linda's shared distributed tuple space in ML without using a preprocessor or making any compiler modifications. We use separate ML modules to implement the Linda interface, operations on tuple space, communication of tuples over the network, and replication of tuple spaces. Our approach allows different compositions of these modules to be used to configure a system with either local or remote access to tuple space, and with either a centralized or distributed implementation of tuple space. Linda shared distributed tuple space complements the functional style of ML by providing a natural mechanism for maintaining shared global state; location transparency and all necessary synchronization are provided transparently by the Linda system.

Section 2 provides some background on the various systems used in the implementation of ML-Linda. Section 3 discusses some of the relevant design issues. Section 4 examines the major interfaces and some of the implementation choices, and the major protocols are sketched in Section 5. The status of the system and some directions for future work are discussed in Section 6.

2 Background

Linda is a set of high-level operations that can be added to a base language to yield a parallel dialect of that language [7, 10]. The Linda programming model consists of an associative memory called *tuple space* and a set of operators: `out`, `eval`, `in`, and `rd`. The unit of communication is the *tuple,* a list of typed fields each of which is either an *actual* or a *formal.* The `out` operation adds a tuple to tuple space. The `in` operation removes from tuple space a tuple that matches a specified *template*, blocking if necessary until a match is found, and binding any formal parameters in the template to the corresponding actuals in the matching tuple. The `rd` operation is similar to `in`, but does not remove the matching tuple from tuple space. The predicates `inp` and `rdp` are nonblocking versions of `in` and `rd`. Finally, `eval` adds an "active" tuple to tuple space: the tuple's fields are computed in an independent process, after which it resolves into a conventional "passive" tuple.

A Linda program selects a tuple by specifying a more general tuple as a *template* to be matched against the contents of tuple space. If tuple space contains more than one matching tuple, a nondeterministic selection is made. The matching algorithm is described in detail in Section 5.1.

Standard ML is a strongly typed functional programming language that supports abstract and polymorphic types, exception handling, garbage collection, and a powerful module system [13]. Its expressivity and type safety combined with its incremental approach to constructing large programs make it an attractive candidate for building complex distributed or parallel programs. The modular unit in ML is the *structure;* ML uses *signatures* to describe the functions and types exported by structures. ML structures can be combined hierarchically using parameterized modules called *functors.*

The *ML Threads* package [9] provides facilities for creating and synchronizing multiple threads of control in a single ML address space.

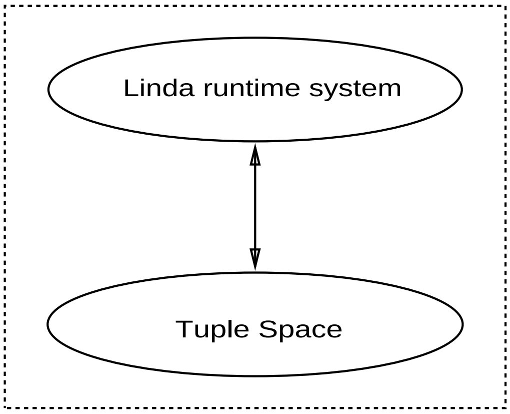

Figure 1: Single local tuple space

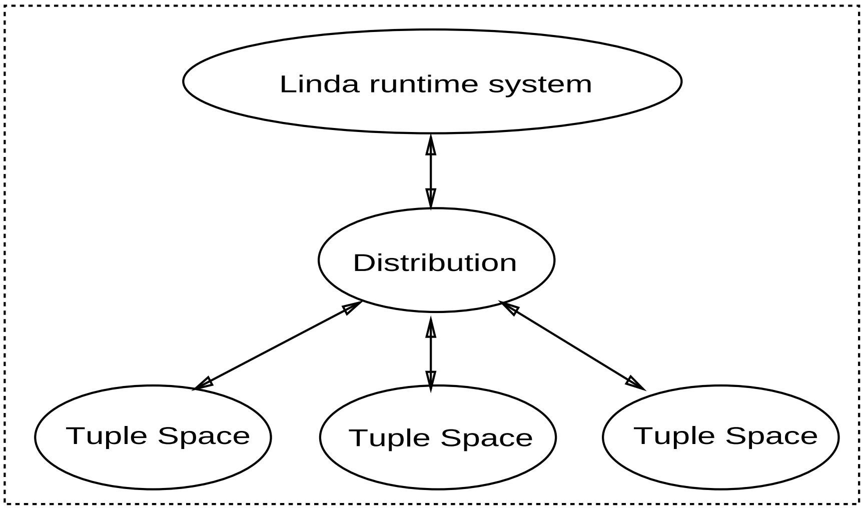

Figure 2: Multiple local tuple spaces

3 Design Issues

Our high-level design goal was to build an implementation of Linda in Standard ML that could be transparently operated in any of several modes: locally (Figures 1 and 2), remotely with a single tuple space node (Figure 3), or remotely with multiple nodes functioning as a distributed tuple space (Figure 4). The result is a distributed Linda system in which the client and server processes may reside on separate nodes and communicate via remote procedure call. Our design avoids any compiler modifications, and minimizes changes to Linda syntax and semantics.

The design takes a layered approach, achieving transparency by taking advantage of ML's strong typing and modular structure. The visible layers are the Linda runtime layer and the tuple space storage layer. The communication layer insulates the system from the network and provides support for remote procedure call, and the distribution layer manages state associated with the multiple nodes of distributed tuple space.

Transparency and flexibility are achieved by having the communication and distribution structures export the same ML signature as the tuple space structure. This transparent

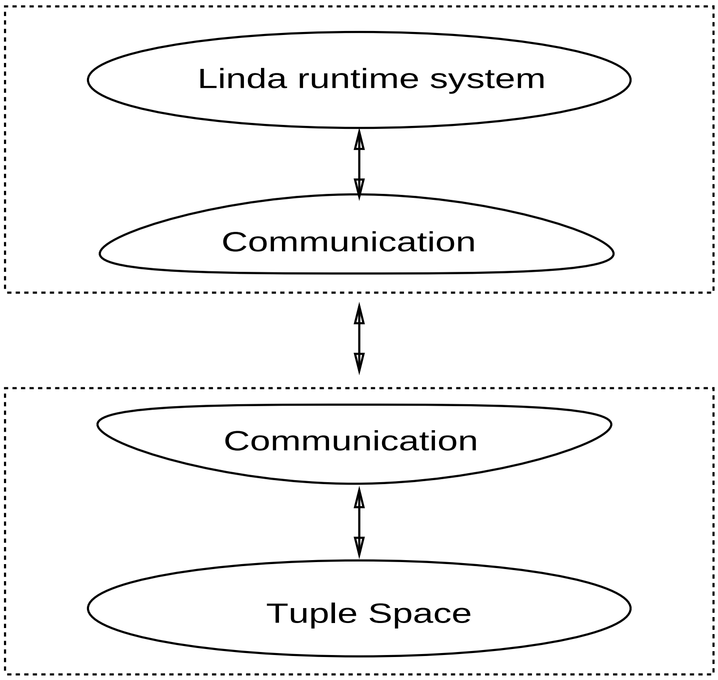

Figure 3: Single remote tuple space

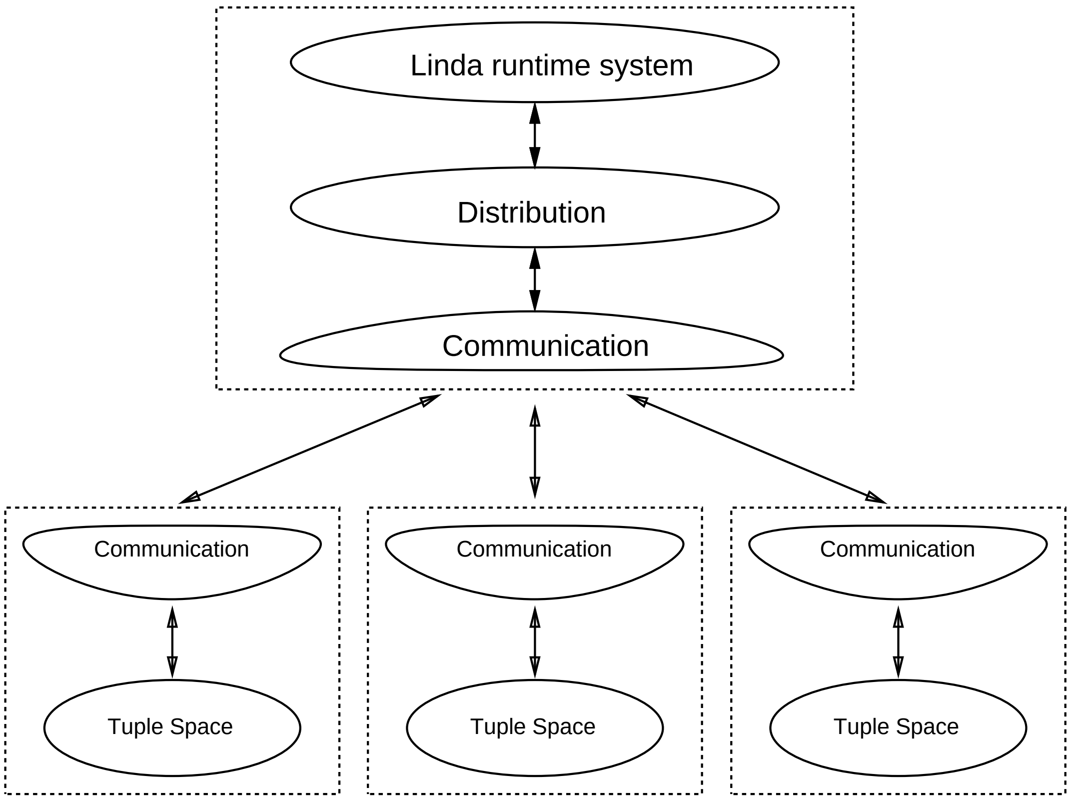

Figure 4: Multiple remote tuple spaces

layered approach allows the ML-Linda system to be configured at link-time with either local or remote access and either centralized or distributed tuple space, simply by deciding which modules to include in the functor application used to build the system. The specification of distributed tuple space nodes, should the configuration require it, is performed at run-time. The system can also be trivially expanded to use multiple tuple space environments [11] by instantiating the desired number of Linda runtime modules.

The basic building block in a Linda system is the *tuple*. Tuples are represented in the ML-Linda prototype as a list of elements from the discriminated union of `INT`, `STRING`, `BOOL`, `PAIR`, `INT_FORMAL`, `STRING_FORMAL`, `BOOL_FORMAL`, or `WILDCARD` (collectively called `ltype`). The constructors `INT`, `STRING`, and `BOOL` take the appropriate arguments, the recursive `PAIR` constructor takes a pair of `ltypes`, and those representing Linda formals take none. Using an ML list type allows an arbitrary number of tuple fields. The generic `WILDCARD` formal is an extension of the Linda typed formals. The runtime type information provided by the explicit `ltype` constructors increases the expressive power of the ML-Linda operators over the power of those in a purely statically typed scheme, albeit at some cost in ease of use.

The set of primitive types provided in the initial implementation is somewhat limited, but the `PAIR` constructor provides the equivalent of a *cons* cell and can be used to build arbitrary data structures. In fact, tuples could be represented directly by nested applications of the `PAIR` constructor; we chose the list representation instead because it provides a simpler syntax for the application programmer. Arbitrarily complex distributed or *live* data structures can also be constructed in tuple space [7]. Such distributed data structures have the advantage that the distributed components are themselves tuples; these components are automatically available for concurrent access because the Linda runtime system provides the necessary synchronization.

A Linda implementation without compiler modifications requires access to runtime type information for the match procedure. Standard ML is statically typechecked and does not require the availability of runtime type information.[1] Our implementation gets its runtime type information by requiring a slightly modified syntax for Linda calls that involves explicitly specifying the field types of any tuple or template arguments, using the `ltype` constructors.

4 Interfaces and Implementations

There are two interface specifications in the system: the Linda runtime system interface and the tuple space server interface.

4.1 Linda Runtime

The Linda runtime structure provides the application level interface to the ML-Linda system. Its signature describes the types, exceptions, and operations that can be used by applications. All of the tuple space operators are supported except `eval`. The main function of `eval` is to introduce concurrent processes, and the ability to fork concurrent threads in the ML environment provides essentially the same functionality. The Linda runtime signature is found in Figure 5; in addition to the basic tuple space operators, it exports the exception

[1] A forthcoming version of Standard ML of New Jersey will support dynamic (runtime) types; this may allow us to simplify our Linda interface.

```
signature LINDA =
    sig
        structure TupleSpace: TUPLESPACE
        exception BadType

        val Init: string list -> unit
        val Out: TupleSpace.ltype list -> unit
        val Rd: TupleSpace.ltype list -> TupleSpace.ltype list
        val In: TupleSpace.ltype list -> TupleSpace.ltype list
        val RdP: TupleSpace.ltype list -> TupleSpace.ltype list
        val InP: TupleSpace.ltype list -> TupleSpace.ltype list
    end
```

Figure 5: Linda Runtime Signature

`BadType` and indirectly exports the discriminated union type `ltype` from within the `TupleSpace` substructure (referenced as `TupleSpace.ltype`). We use capitalized operation names such as `In` and `Out` because `in` is a reserved word in ML.

The syntax seen by the programmer differs slightly from other Linda dialects. A Linda tuple is represented in ML by a list of `ltypes`. Invocation of Linda operations in ML requires on-the-fly construction of the `ltype` list from the desired tuple parameters and deconstruction of the result tuple; the result is returned as an `ltype` list rather than via direct binding of the formal parameters to values as in C-Linda [8]. To illustrate the differences, Figure 6 compares the syntax of ML-Linda with that of C-Linda. Each of the two program fragments invokes three Linda operations, `out`, `in`, and `rd`. Since C-Linda uses a preprocessor to extract type information, the arguments to the `out` call need not specify types as long as they have been previously declared in the program. In contrast, the same call in ML-Linda is explicitly represented as a list (denoted by the square bracket delimiters), and each list element (tuple field) is tagged with the appropriate constructor.

The Linda runtime structure exports the application level interface, maintains state by generating hash strings and unique IDs, and translates the runtime operations into the appropriate calls to tuple space.

An example of a small application program using Linda is shown in Figure 7. The program exports two functions: one repeatedly writes a tuple with the string “ping” and reads one with the string “pong”; the other reads a tuple with the string “ping” and writes one with the string “pong”. Using the ML Threads package, these functions can be forked as concurrently executing threads; this portion of the code is not shown.

An ML-Linda solution of the classic Dining Philosophers problem is shown in Figure 8. The code to create the five philosopher threads is not shown.

4.2 Tuple Space

The tuple space interface (Figure 9) is exported by the tuple space server and also by any additional layers between it and the Linda runtime system. These additional layers thus “virtualize” the tuple space interface. The operations correspond roughly to those in the runtime interface, although some of the high-level Linda operations have been broken down into parts in the tuple space interface. The `in` and `inp` operations have been broken down into two phases, the `rmv` and `purge/restore` phases, and the blocking `rd` operation likewise has a second phase (`rd_done`). The second phases of these operations are required to clean up state and undo any side effects of the initial operations. The two phase

```
C-Linda:

    {
        int i = 3;
        char s[32] = "myprog";
        int j;
        struct pair { char str[32]; int flag; } p;

        out (s, i, 7);
        /* C-Linda uses the ? operator to label formals */
        in ("myprog", ? j);
        rd (? p);
        /* j and p can be used here */
         ...
    }

ML-Linda:

      let val i = 3
          val s = "myprog"
      in
          Out [STRING s, INT i, INT 7];
          let val [_, INT j] = In [STRING "myprog", INT_FORMAL]
              val template = [PAIR (STRING_FORMAL, INT_FORMAL)]
              val [PAIR (STRING str, INT flag)] = Rd template
          in
              (* j, str, and flag can be used here *)
              ...
          end
      end
```

Figure 6: Syntax of ML-Linda and C-Linda

```
local open Linda TupleSpace
in
    fun ping () =
        (Out [STRING "ping"]; print "ping ";
         In  [STRING "pong"]; ping ())

    fun pong () =
        (In  [STRING "ping"]; print "pong ";
         Out [STRING "pong"]; pong ())
end
```

Figure 7: A Simple Example

```
local open Linda TupleSpace
in
    val num = 5

    val room_ticket = [STRING "room ticket"]
    fun left_chopstick i = [STRING "chopstick", INT i]
    fun right_chopstick i = [STRING "chopstick", INT ((i+1) mod num)]

    fun philosopher i =
        ((* THINK *)
         In room_ticket;
         In (left_chopstick i); In (right_chopstick i);
         (* EAT *)
         Out (left_chopstick i); Out (right_chopstick i);
         Out room_ticket;
         philosopher i)
end
```

Figure 8: Dining Philosophers in ML-Linda

approach is not necessary for a nondistributed tuple space, but the implementation does not distinguish between the distributed and nondistributed cases in order to keep distribution transparent to the Linda runtime system.

Tuple space is represented by a data structure that maps hash strings to tuples. The data structure is implemented as an array of linked bucket structures. Since a tuple space module can have multiple clients and may be multithreaded, the implementation must provide appropriate synchronization and locking on shared data structures. Each array slot has a corresponding lock. Array slots are locked for `add` and `rmv` operations, but not for `lookup`. This serializes all `out` and `in` operations, but each modification involves updating the value in the array slot because the bucket structures are immutable.

Each node of distributed tuple space maintains its own version of the tuple space data structure. Tuple field types are restricted to those that can be expressed with constructors of the `ltype` discriminated union exported by the tuple space structure. Within each table, tuple matching is based on the types, values, and order of tuple fields. The initial implementation takes a naive approach and hashes on a string constructed from substrings representing the type of each field in the order of its appearance in the tuple, although this will tend to result in an unbalanced table with all tuples of the same type grouped together. We implement a slightly modified version of the Linda matching algorithm in which tuples with formal fields are allowed only as template parameters, and do not reside in tuple space. The details of the matching protocol are described in Section 5.1.

4.3 Distributed Tuple Space

Distributed tuple space is a single logical associative memory that is implemented as a set of distinct tuple space servers distributed over a collection of physically separate nodes. The tuple storage and matching systems are replicated on each node of distributed tuple space, although the stored contents are not replicated. Each node of a distributed tuple space manages its own resources and exports all of the Linda functionality. All of the logic involved in combining the individual nodes into a single logical tuple space is located in the distribution module, which is layered transparently between the Linda runtime and its communication layer.

```
signature TUPLESPACE =
    sig
        datatype ltype =
            INT of int | STRING of string | BOOL of bool
          | INT_FORMAL | STRING_FORMAL | BOOL_FORMAL
          | PAIR of ltype * ltype
          | WILDCARD

        exception NotFound

        val init : string list -> unit

        (* add is used by out *)
        val add : string * ltype list -> unit

        (* lookup is used in phase 1 of rd/rdp *)
        val lookup : string * string * ltype list * bool -> ltype list

        (* rd_done is used in phase 2 of rd/rdp *)
        val rd_done : string -> unit

        (* rmv is used in phase 1 of in/inp *)
        val rmv : string * string * ltype list * bool -> unit

        (* purge and restore are used in phase 2 of in/inp *)
        val purge : string * bool -> ltype list
        val restore : string -> unit
    end
```

Figure 9: Tuple Space Signature

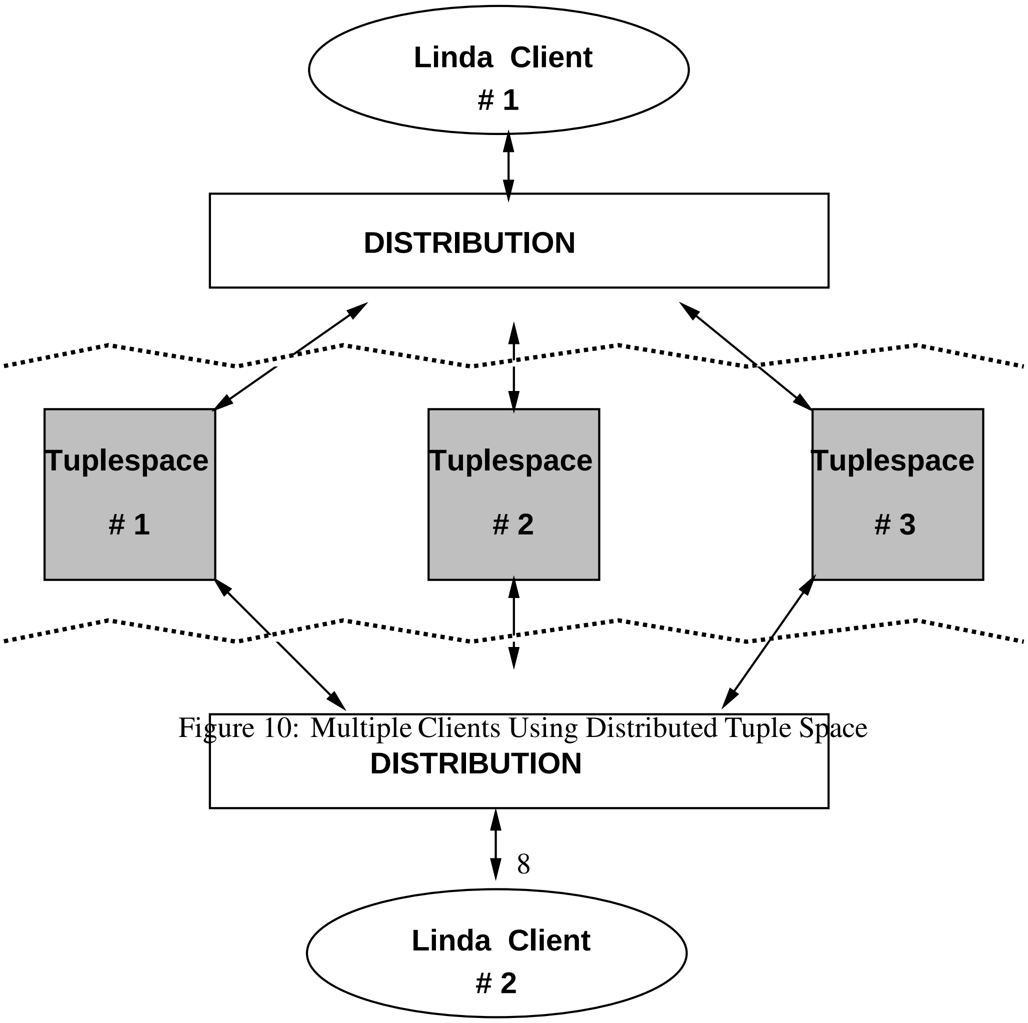

Figure 10: Multiple Clients Using Distributed Tuple Space

An important goal of the distributed Linda implementation is transparency. The client application should depend only on the interface to tuple space, not on any implementation-specific aspects having to do with communication, distribution, or physical location. This means that the distribution of tuples in distributed tuple space must be independent of implementation choices such as the hashing algorithm. Tuples may reside on any valid node of distributed tuple space (see Figure 10), but the client application sees only a single unified view of tuple space. Ideally, the tuples end up distributed over the available nodes in such a way as to optimize both the necessary network traffic and balance the loads on the relevant nodes. However, the initial implementation takes a simple approach and selects destination nodes by cycling through the tuple space nodes.

A general approach to managing consistency and availability of data in a distributed system involves using replication with appropriate read and write quorums [12]. To ensure that valid data is always available, accesses are made from *read quorums* and updates to *write quorums.* The sum of the read and write quorums must be greater than the total number of nodes in the system, ensuring that the two quorums must overlap by at least one node.

Although a replication scheme can normally use any legal combination of read and write quorums, a Linda implementation can only safely use read-one-write-all or read-all-write-one quorums without compromising the location transparency of the system.[2] This peculiarity is due to the ambiguity of the `rd` operation. Rather than reading a single logical object, the `rd` operation returns any member of an arbitrarily large set of possible matching tuples. Since a set of n replies is not guaranteed to include n copies of the same object, an arbitrary read quorum would not necessarily provide the required overlap with the corresponding write quorum.

Our ML-Linda distribution layer uses a read-all-write-one approach to communicate with distributed tuple space. This means that `out` sends a tuple to a single tuple space node, while `in` and `rd` request matches from all nodes. We simulate multicast communication by using multiple threads to make the remote procedure calls in parallel. We chose read-all-write-one semantics over read-one-write-all in order to minimize the amount of data transmitted: only one tuple needs to be transmitted for each operation. [3] Although `rd` and `in` are sent to all tuple space nodes, only one match is chosen; furthermore, it is likely that only one version of tuple space will be modified since requests are likely to target a specific tuple. If we used a read-one-write-all scheme, the `in/inp` operation would be too expensive: although the `lookup` operation would be performed on a single node, all of the remaining nodes would have to be involved in order to delete the tuple from tuple space.

In our distributed tuple space, with read-all-write-one semantics, an application can receive several different tuples in response to an `in/inp` or a `rd` operation. The `rmv` phase of the `in` operation must tentatively remove the matched tuple from tuple space, to make it unavailable to any subsequent match requests. Since `in/inp` is a destructive operation, only one match can be accepted; the tentative removals associated with any other responses must be undone. In addition to restoring unwanted matches, it is also necessary to terminate any remote threads that are still blocked at the end of an operation. The details

[2]The $n \times n$ processor grid discussed by Bjornson *et al.* [5] allows a valid intermediate quorum assignment that depends on the physical configuration of the system. For this configuration, the write quorum for node i consists of the n nodes in i's row, and the read quorum consists of the n nodes in i's column.

[3]We actually have `rd` return the matching tuple as an optimization, since we generally expect only a single match and `rd` does not require a second phase for any nodes returning `NotFound`. This is discussed in more detail in Section 5.4.

of managing distributed tuple space are discussed in greater detail in Section 5.

4.4 Communication

Our implementation of distributed Linda requires typesafe, transparent communication of complex objects from one node to another. A `rd` or `in` operation binds the fields of the appropriate matching tuple to a set of local variables that can then be modified or referenced at will. In order to transmit an opaque type it would be necessary to provide the communication system with information about the implementation that is not available in the signature; this effectively makes the type explicit, and requires a new mechanism such as a preprocessor to extract the relevant type information and make it available to the communication system. Complex types whose implementations are included in the signature are already explicit, and can be readily transmitted. Given the constraints of the ML type system, it is impractical to use abstract (opaque) types in tuple fields. In order to use Linda's shared tuple space, the concrete representations must be visible.

The ML-Linda implementation follows a client-server model, with one or more individual tuple space nodes acting as remote servers to the local Linda client application. The communication support provides client and server stub structures that hide the communication details from the application code. The stub structures provide routines for marshaling and unmarshaling each of the relevant argument and result types, as well as control structures for making remote procedure calls. The server communication layer also provides a listener loop that waits for incoming requests from the network and forks threads to service them. The communication structures appear transparent to the application and server code because they export the same interface as the tuple space structure.

For a distributed version of tuple space, an additional stub layer is added to hide the distribution (see Figure 2). A client application will therefore see the same tuple space abstraction regardless of whether the tuple space structure is local, remote, or distributed. Ideally the distribution module would be able to use multicast for some of its operations, but support for multicast is not yet available. The current implementation therefore simulates multicast with multiple threads performing calls in parallel.

5 Protocols

This section describes the Linda tuple matching algorithm and presents the protocols we use to implement distributed tuple space, using the read-all-write-one scheme described in Section 4.3.

5.1 Tuple Matching

All communication and synchronization in the Linda model is accomplished by tuples moving into and out of tuple space. The Linda tuple matching algorithm is defined as follows [14]. Call the tuple defined by the fields in an `in` or `rd` operation a *template*. A template μ matches a tuple τ in tuple space if all of the following conditions hold:

- μ and τ have the same number of fields.
- Corresponding fields have the same types.
- Each pair of corresponding fields F_μ and F_τ match as follows:

- If both F_μ and F_τ are actuals, they match if and only if their respective values are equal, where equality is defined by the base language for objects of this type.
- If F_μ is a formal and F_τ is an actual, they match; the value of F_τ may eventually be assigned to some variable. No assignment takes place unless *all* the fields match, however.
- If F_μ is an actual and F_τ is a formal, they match unconditionally. The value of F_μ is discarded.
- If both F_μ and F_τ are formals, they never match.

5.2 State Information

The protocol for `in/inp` is discussed in detail from both the runtime system and tuple space points of view. Protocols are sketched for the `rd/rdp` and `out` operations from the runtime system's point of view. All operations begin with the construction of a hash string based on the template or tuple argument at the Linda runtime system level (see Figure 11).

```
[STRING "hello", INT 5, BOOL true]
        => "string int bool "

[STRING "hello", PAIR (INT_FORMAL, BOOL_FORMAL), WILDCARD]
        => "string pair int_formal bool_formal wildcard "
```

Figure 11: Hash String Construction

The distribution structure maintains a set of data structures that are used to record the state of the match for each tuple space node participating in phase one of `in`, `inp`, and `rd`. Each data structure includes:

- A use indicator, either empty or a string that uniquely identifies the operation using the structure.
- A pointer to the connection supplying the chosen match.
- A tuple space status variable for each tuple space node.

The tuple space status is indicated via a discriminated union of `NULL`, `MATCH`, and `NOTFOUND`; the state is reset to `NULL` at the end of each operation; at any other time `NULL` indicates a thread that hasn't yet returned (in our no-failure model, this is equivalent to a blocked thread). Synchronization is achieved by locking the use indicators. A process must acquire a lock in order to modify a use field, and a field must be set to a null value before it can be used. Once a process sets a use field to its unique ID value, the corresponding data structure is reserved until it resets the use field.

Each tuple space implements a data structure that maintains the state required for the two-phase operations `in`, `inp`, and `rd`. The data structure maps a unique ID supplied by the Linda runtime system into either a *match* record consisting of a unique ID, hash string, and tuple; or a *kill* record consisting of the unique ID. The data structure is currently implemented as an array of linked structures; there is a lock on each array slot to ensure that access by threads with the same unique ID is always serialized.

The Linda runtime module is responsible for generating hash strings for each of the tuple space operations as well as the unique ID for the `in`, `inp`, and `rd` operations. The operation hash strings are constructed by simply concatenating together the strings representing the type of each tuple or template field. The unique ID is a concatenation of the string representations of a sequence number (incremented after each use), the host name, the process ID, and the hash string.

5.3 The `in` and `inp` operations

The `in` operation is a two-phase operation implemented by two sequential remote calls (see Figure 12). The first phase is the `rmv` operation, which involves all *n* tuple space nodes. The second phase, which can be either a `purge` or `restore` operation, must be invoked on each node participating in the `rmv` phase. To connect the two phases of the operation, the Linda runtime assigns a unique ID that is sent as an argument along with each remote call. To emulate a multicast operation, the calls in each phase are made concurrently by forking one thread for each tuple space node.

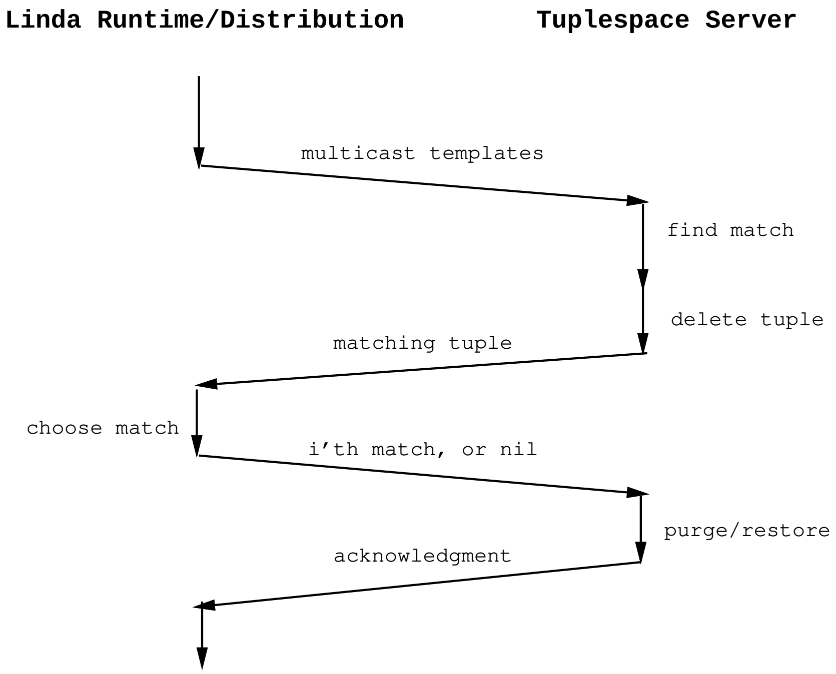

Figure 12: Protocol for `in` operation

On the application side, the distribution layer reserves a status structure by writing its unique ID into the use field. It then emulates a multicast operation, concurrently sending the unique ID, request template, and hash string from the Linda runtime module to all tuple space nodes. As each thread returns, it records the appropriate status in the status structure; the first thread with a match may modify the match connection variable. Once a match is found, the call may be terminated. This is done by making a `purge` request to the tuple space node specified by the match connection variable and making `restore` requests to each of the remaining tuple space nodes; this will undo any tuple space modifications and kill any blocked threads remaining from the `rmv` request. The `purge` and `restore` requests use the unique ID to access the tuple and hash string information stored in the state array. The `purge` operation returns the matched tuple, which is then returned to the application process.

The `rmv` phase of the blocking `in` operation may leave blocked threads on some of the tuple space nodes after the application has already chosen a tuple. These threads must be located and terminated as part of the cleanup for the operation. Since the remote threads are running concurrently it is also possible that a remote match can be found on a particular tuple space node at the same time that the restore request is running. These conditions create the need for communication and synchronization, which is filled by the tuple space state structure. The application provides both phases with the same unique ID, one that is unique per operation per application. This hash table holds either a match record from the lookup phase, or a kill record from the restore phase; the latter indicates that the lookup thread should terminate itself. Each thread acquires the appropriate lock before executing to ensure that the blocked thread can never return a match once a kill command has been issued.

The remote server thread executes the `rmv` operation as a loop. At the top of the loop, it immediately acquires a lock on the part of the state structure corresponding to its unique ID. It then checks for a kill command from a restore operation; if one exists, the thread clears the command from the state array, releases the lock, and exits. If no record of a restore operation is found, the thread searches tuple space for a match. If a match is found, the matching tuple and hash string are recorded in the state array along with the operation unique ID. The thread then releases its lock and returns. If no match is found, the lock is released and the thread blocks. All blocked threads are awakened each time a tuple is added or restored to tuple space. When a blocked thread is awakened it continues execution at the top of the loop.

The second phase of the `in` operation consists of a single `purge` invocation on the node that provided the chosen match, and invocations of `restore` on the remaining tuple space nodes. Each remote thread executing `restore` acquires the lock on the part of the state array corresponding to its unique ID and checks the state array to see whether the thread from the `rmv` phase has found a matching tuple. If it has, the `restore` thread clears the entry from the state array, uses the recorded information to restore the appropriate tuple to tuple space, releases the lock and returns. If no matching tuple has been found, the remote thread records a kill command for the blocked thread to find when it next awakens, releases the lock, and returns.

A `purge` operation is similar to the `restore` operation, but simpler. The remote thread acquires the lock on the appropriate part of the state structure, clears the existing entry from the state array, releases the lock and returns. A `purge` request is only issued for a node that has already returned a match, so the thread from phase one will always have left a record in the status array and exited.

The differences between the blocking `in` and nonblocking `inp` operations are relatively minor. Tuple space threads executing `inp` that fail to find a match return `NotFound` exceptions to the distribution module rather than blocking until a matching tuple appears. If all threads return `NotFound`, the `inp` call will relay the exception to the application. For the second phase, the nonblocking version need not issue `restore` requests to remote nodes that raised the `NotFound` exception, since they will not have left any state on the remote tuple space. Otherwise, the protocol is the same as for `in`.

5.4 The `rd` and `out` operations

The `rd` operation is similar to the `rmv` phase of the `in` operation except that it has no side effects on tuple space. The Linda runtime system multicasts the template argument and the hash string to all nodes of distributed tuple space. The blocking `rd` call also includes a unique ID so that a status record may be written if a match is found. The status record is unnecessary for the nonblocking `rdp` call since all remote threads exit and there is no need to clean up any remote state. For the blocking `rd`, the threads involved in the call (both local and remote) remain blocked until a match is found. Once the first response tuple is received, any outstanding requests to other nodes are canceled with a `rd_done` operation, and the tuple is returned to the application. For `rdp`, which is nonblocking, each tuple space node searches its database only once and either returns the tuple match or raises a `NotFound` exception. The Linda runtime simply returns the first matching tuple, or reraises the `NotFound` exception if no node finds a match.

Since `out` is a write operation, the runtime system forwards the tuple argument with the constructed hash string to one node of the distributed tuple space. Assuming a reasonable distribution of processes, it might be optimal to write the tuple to the local node if it is a tuple space node. The current implementation simply rotates through the list of tuple space nodes to provide a wider distribution of tuples. The addition of the new tuple to each tuple space causes any blocked threads to be awakened so that they can try to match against the new tuple.

6 Discussion

Distributed Linda can be implemented using either *replication* [2, 6, 14] or *hashing* [4]. Some designs also include the notion of dynamic migration: if a set of tuples is being consumed on a particular node, it is more efficient if they are stored directly on that node.

The hash-based approach implements tuple space as a distributed hash table in which one or more buckets map to each node. Tuples are categorized into disjoint sets, and a hash function is provided which ensures that all tuples with common characteristics hash to the same bucket (node). All tuple stores and match requests are then directed to the node specified by the hash function, localizing the search area.

The replicated approach makes it easier to add new types of tuples to the system, or migrate existing tuples from one node to another. Depending on the relative frequency of the different operations, the replicated approach may also require less network traffic (assuming that multicast or broadcast is available). The overall performance of the two approaches is likely to depend significantly on the pattern of communication among the active processes.

6.1 Fault Tolerance

Although it is an unrealistic assumption for a production system, the ML-Linda prototype does not guarantee consistency or correctness in the presence of failures. The correctness of the Linda communication model is threatened by failures such as crashes and partitions. Unlike more conventional systems, communication in Linda is not from process to process or node to node; since the Linda model constrains neither the locations of processes nor the lifetime of data in tuple space, there is no external way to tell if data are absent because the producer process died, because one or more of the relevant nodes have crashed or are partitioned, or because the data have not yet been produced. Furthermore, some operations require participation by all nodes in the distributed tuple space, making the system highly vulnerable to node or communication failure.

Replicated multi-phase operations are particularly difficult to implement correctly in the presence of failures. A fault tolerant implementation would require the support of persistent storage and a transaction system to ensure correctness, and even so leaves open the possibility of indefinitely long blocking at multiple stages in the protocol.

6.2 Status

The prototype ML-Linda system is implemented with the Standard ML of New Jersey compiler [3] running on the Mach operating system [1]. It consists of roughly 700 lines of ML, implementing three functors: Linda runtime, distribution, and tuple space. The communication subsystem, which is generated from the tuple space signature by an RPC stub generator, is not included in this tally. The system runs on VAX, MIPS, and Sun machines. An earlier version implemented the blocking `in` and `rd` operations by repeatedly polling tuple space from the Linda runtime layer; the current version uses blocking threads as described in Sections 5.3 and 5.4, but the ML thread scheduler has not yet been integrated with the Mach communication facilities.

6.3 Performance

The current system would require some tuning and enhancements before it could be used as a production system. Some of these are straightforward, but some require an understanding of typical application behavior in order to identify bottlenecks and other stresses on the system. One important direction for future work is to implement different types of Linda applications, and monitor the distribution and frequency of each of the Linda operations and the reference patterns and average lifetimes of tuples. It would also be useful to compare the performance of several different heuristics for choosing tuple space nodes for the `out` operation, and possibly for dynamic migration. Dynamic migration might be useful for systems with many more `rd` than `in` operations. Both types of heuristics might be integrated with dynamic tuple space resizing as part of an attempt to keep the distribution of tuples properly balanced. Designing a better hashing algorithm would also help to keep tuples evenly distributed within the tuple space of a single node.

7 Conclusion

The prototype implementation of ML-Linda demonstrates the feasibility of combining ML's functional approach with the Linda model of parallel programming in distributed tuple space. The ease of decomposing the Linda system into ML modules reinforces the suitability of ML for constructing complex distributed programs. The addition of Linda shared tuple space complements ML's functional style and provides a flexible environment with the benefits of both programming models for the development of distributed systems.

References

[1] Michael J. Accetta, Robert V. Baron, William Bolosky, David B. Golub, Richard F. Rashid, Avadis Tevanian, Jr., and Michael W. Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the Summer 1986 USENIX Conference*, pages 93–113, July 1986.

[2] Sudhir Ahuja, Nicholas J. Carriero, David H. Gelernter, and Venkatesh Krishnaswamy. Matching language and hardware for parallel computation in the Linda machine. *IEEE Transactions on Computers*, C–37(8):921–929, August 1988.

[3] Andrew W. Appel and David B. MacQueen. A Standard ML compiler. In *Functional Programming Languages and Computer Architecture*, pages 301–324. Springer-Verlag, 1987. Volume 274 of *Lecture Notes in Computer Science*.

[4] Robert Bjornson. Experience with Linda on the iPSC/2. Research Report 698, Department of Computer Science, Yale University, March 1989.

[5] Robert Bjornson, Nicholas Carriero, David Gelernter, and Jerrold Leichter. Linda, the portable parallel. Research Report 520, Department of Computer Science, Yale University, February 1987.

[6] Nicholas Carriero and David Gelernter. The S/Net's Linda kernel. *ACM Transactions on Computer Systems*, 4(2):110–129, May 1986.

[7] Nicholas Carriero and David Gelernter. How to write parallel programs: A guide to the perplexed. *ACM Computing Surveys*, 21(3):323–357, September 1989.

[8] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[9] Eric C. Cooper and J. Gregory Morrisett. Adding threads to Standard ML. Technical Report CMU-CS-90-186, School of Computer Science, Carnegie Mellon University, December 1990.

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

[11] David Gelernter. Multiple tuple spaces in Linda. In *PARLE 89*, volume 366 of *Lecture Notes in Computer Science*, pages 20–27. Springer-Verlag, June 1989.

[12] David K. Gifford. Weighted voting for replicated data. In *Proceedings of the 7th Symposium on Operating Systems Principles*, pages 150–162, December 1979. Published as *Operating Systems Review,* 13(5).

[13] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[14] Robert A. Whiteside and Jerrold S. Leichter. Using Linda for supercomputing on a local area network. In *Proceedings of Supercomputing*, pages 192–199, November 1988.