On the Use of Linda as a Framework for Distributed Database Systems

(Position Paper)

Madhan Thirukonda
Florida Tech
Computer Sciences
Melbourne, Florida
mthiruko@cs.fit.edu

Ronaldo Menezes
Florida Tech
Computer Sciences
Melbourne, Florida
rmenezes@cs.fit.edu

ABSTRACT

Linda *is a coordination language capable of solving issues in distributed computing environments that relate to process synchronization, communication and creation. The expressiveness of* Linda *in distributed systems is such that researchers are proposing novel applications using* Linda *as a primary means of coordination. The examples range from peer-to-peer to groupware computing, from simple chat applications to control systems. Surprisingly,* Linda *has not been used in the field of distributed databases, although* Linda *can be helpful in solving coordination issues in a distributed database system. In this paper, we look at a possibility of using* Linda *in the context of distributed databases.*

1. INTRODUCTION

Linda [10] is a coordination language proposed by Gelernter in 1985. A coordination language provides necessary support for computational activities to communicate with each other as they often need to interact in order to achieve a goal. Coordination languages provide support for such interactions as they can be thought of as extensions to traditional languages to handle synchronization, communication, and process creation tasks.

A significant characteristic of Linda is that computational activities are uncoupled in time and space. Time uncoupling offers the possibility that two processes can communicate with each other through-time. That is, their execution times do not have to intersect (overlap) in order for the communication to take place. Space uncoupling relates to the fact that processes do not need to know the location of other processes — communication takes place independently of the location of the processes involved. A consequence of space uncoupling is process mobility, which is inherent in the Linda model.

The expressiveness and simplicity of Linda is what makes it the most successful coordination model today. It can deal with complex problems in a very simple and clear way. The coordination perspective helps us to effectively tackle complex problems effectively. The solution to problems such as *Distributed Summation* [4] and *Stable Marriages* [20] are just a few examples where Linda has demonstrated its expressiveness.

Though Linda and databases were proposed for different purposes, the close relationship between them has long been recognized in the coordination community. IBM *TSpaces* [12] was developed based on Linda and provides database query capabilities [21].

Databases, on the other hand, have been extensively using the power of data distribution. Database vendors have realized the the distribution of data may lead to a better scalability of database systems. The urge for scalability and the advances in the database technology have led to the development of many distributed databases such as INGRES/STAR [1].

The intrinsic characteristics of Linda, such as tuple based communication and support for queries, make Linda an excellent candidate to serve as a framework for distributed database systems. The complex coordination mechanisms needed for the interaction between different databases can be handled more elegantly and efficiently in coordination systems such as Linda

In this paper, we argue that Linda can be used to decrease the complexity of developing distributed databases by acting as the framework where databases system can be developed. The paper is divided as follows: the paper start in Section 2 with a description of the Linda model. Section 3 talks about distributed databases and their main characteristics. In Section 4 ventures into using Linda as a framework for the implementation of distributed databases and presents our envisioned architecture. Section 5 concludes the paper with some discussion on future research directives.

2. LINDA

The Linda coordination model [10] is based on the concept of tuple space to provide a unified view of process creation, communication and synchronization. These concept are treated uniformly by being all implemented in terms of tuple-space operations.

The tuple space communication model consists of associative shared memories (tuple spaces) which are able to store *tuples*. Processes can store and retrieve tuples from tuple spaces, but are unable to communicate

with each other directly. The retrieving of tuples uses an associative matching mechanism based on *templates*. Templates differ from tuples because they may contains non-valued field (holes) represented by `?type` or `?typed-variable`. For instance, the template `[?int, "Hello"]` matches the tuple `[1,"Hello"]`.

Associative matching is extensively used in database applications. Most queries of database systems are based on some variation of associative matching. Take for instance, the `select` operation in SQL where the characteristics of the records are given in the query. Describing the characteristics (in abstract terms) is how associative matching works.

The multiple tuple space LINDA model [11] is an evolution of the single tuple space model in which processes have access to a unique tuple space representing the whole associative memory [10]. The multiple tuple space model provides a more realistic view of memory by allowing processes to organize information in local tuple spaces. A processes can create a local tuple space to store its information which will be unavailable to the rest of the system until the process decides to make the tuple-space name (its handle) available in the Universal Tuple Space (`UTS`) — a tuple space known to all the process.

The model provides primitives to store, read and withdraw tuples from tuple spaces, to create tuple spaces and to spawn processes. The primitives below includes the description of two bulk primitives [17] which although not considered standard have been widely used in several LINDA implementations.

`out(TSm,tuple)`: Stores the `tuple` in the tuple space `TSm`.

`in(TSm,template)`: Removes from `TSm` a tuple matching the `template`. If there is more than one candidate tuple, one is chosen non-deterministically. If there is no such a tuple the process is blocked until a matching tuple appears within `TSm`.

`rd(TSm,template)`: Same as `in` but non-destructive, that is, the tuple is copied as opposed to removed.

`n = collect(TSm,TSq,template)`: Bulk primitive which moves all tuples matching the `template` from `TSm` to `TSq` and assigns to `n` the number of tuples moved [4].

`n = copy-collect(TSm,TSq,template)`: Bulk primitive which copies all tuples matching the `template` from `TSm` to `TSq` and assigns to `n` the number of tuples copied [18].

`eval(TSm,tuple)`: The LINDA way of spawning processes. Processes are created to evaluate the elements of the `tuple` in parallel.

`handle = tsc()`: Creates a new local tuple space and assigns a unique identifier to `handle`.

The remainder of this paper look at the main issues encountered in the distributed database systems and how LINDA can cope with them. It should be clear that LINDA on itself does not deal with all the aspects covered. Therefore extensions to the basic model are described whenever necessary. The goal is to show how one could use LINDA as the basis of a distributed database system.

3. DISTRIBUTED DATABASE SYSTEMS

A distributed database system (DDBS) is a collection of several logically related databases which are physically distributed in different computers (otherwise called *sites*) over a computer network [15, 1]. All sites participating in the distributed database enjoy local autonomy in the sense that the database at each site has full control over itself in terms of managing the data. Also, the sites can inter-operate whenever required. The user of a distributed database has the impression that the whole database is local except for the possible communication delays between the sites. This is because a distributed database is a logical union of all the sites and the distribution is hidden from the user [7].

A distributed database management system (DDBMS) is a software system that is responsible for managing all the databases of a DDBS [15]. For transparent management of data distribution, each participant database has a component that provides a logical extension to interoperate with other databases. This component plays a major role of coordination in a DDBMS.

DDBS is preferred over a non-distributed or central database system (DBS) for various reasons. Distribution is quite common in an enterprise. For instance, various branches of the enterprise can be located at various geographic locations. Also, there can be logical divisions in a branch such as human resource department and administration department. Each logical division can be treated as a site. At each site, a DDBS provides all capabilities of a DBS local to that site in addition to providing other advantages of a distributed system such as replication and fault tolerance. Though the concept of data distribution seems to be attractive, the cost associated with distributing the data is very high. Also, data distribution makes concurrency control and recovery mechanisms complex. A few of these problems are described in detail in section 3.1.

3.1 How is a DDBS different from a DBS

In a DDBS, the presence of distribution complicates the coordination mechanism of sites. We are considering three important issues namely replication, concurrency control, and fault tolerance in this paper.

For better readability, we make the following distinction between the terms *record* and *tuple*. While discussing the records of a table in a database, we use the term *record* and we use the term *tuple* to refer to a tuple in a tuple space in LINDA.

3.1.1 Replication

Replication can be done by storing multiple copies of an entire table or partitions of a table (called *fragment*) at multiple sites [8]. A table can be partitioned horizontally or vertically. A horizontal partition consists of

a subset of records of a table (the attribute structure of the table being the same). A vertical partition consists of a subset of attributes of a table, each partition sharing at least one common attribute. The fragments can then be replicated across the sites.

In a DDBS, replication provides at least two benefits: high availability and improved performance [7]. High availability can be achieved by replicating and distributing the data to different sites that require frequent access to the data. The site requiring the data retrieves locally and this reduces the communication overhead with other sites. Replication improves performance by taking advantage of local processing, which eliminates the latency due to access to the communication network.

Replication may be expensive. When a replicated data item is updated, all the replicated data items should be updated to maintain data integrity. It is referred to as *update propagation* [7]. During update propagation, different replica would be in different states. If any other process accesses a replicated data item during update propagation, it would result in processes using different state of the same data item. The complexity lies in avoiding the processes using the data item during update propagation. In addition to update propagation, replication requires the DDBMS to provide replication transparency. An user should be able to use a data item as if it was not replicated at all.

3.1.2 Concurrency control

The concept of transactions is often used in databases (in DBS and in DDBS). A transaction can be defined as a series of actions carried out by a user/application. The actions defined in a transaction are either executed completely or not executed at all (this property is also termed as *atomicity*). A transaction transforms the database from one consistent state to another consistent state and it may start when a command like *begin transaction* is executed and it may end when a command like *commit* is executed [1].

In a DDBS, the concept of transaction is complicated because a transaction may access data stored at more than one site. Typically, a transaction is divided into sub transactions that can be executed at different sites. Now, the complexity is due to the fact that the sub transactions can inter-operate or interfere with other sub transactions leading to concurrency problems. The uncertainty in the order of execution of sub transactions may result in different output or different state of the database. In this context, concurrency control is a process of controlling the relative order of two or more sub transactions that interfere with each other.

The algorithms for solving such concurrency problems are known as synchronization techniques. There have been many synchronization techniques proposed in the literature. Unfortunately, all of them are hard to understand and complex to implement [2].

3.1.3 Fault Tolerance

The problems due to distribution becomes complex during site failure(s). Failures can occur at various levels of a DDBS. Bell *et Al.* [1] categorized failures in a DDBS under four headings:

- Transaction-local failures (failures in completing a transaction)
- Site failures (failures such as power supply that affect all transactions at that site)
- Media failures (a portion of database being corrupted)
- Network failures (communication failures in the network)

The above identified failures can occur in both DBS and in DDBS. In a DBS, these failures may be handled by restarting the transaction, the DBS could solve the issue. In a DDBS, the global dependence of sub transactions makes recovery more complex. This is because restarting a sub transaction at a site may require the DDBMS to restart all participant sub transactions of that global transaction to maintain global consistency and atomicity. Maintaining atomicity at both local level and global level during site failures adds a new dimension to recovery algorithms.

4. A LINDA-BASED FRAMEWORK

In a distributed environment, processes have to interact with each other in order to solve a problem. The communication mechanisms are complex due to dependencies between the components. Since coordination deals with managing dependencies between activities, a coordination language is a good candidate to deal with the complexities that naturally arise in distributed environments [13].

The practical use of LINDA can be seen in recent implementations: TSpaces [12] by IBM and JavaSpaces [9] by Sun Microsystems, which were developed under the strong influence of LINDA. TSpaces was developed with a notion of persistence and it can perform many duties of a simple database system as it has indexing and query capabilities similar to a (relational) database [21]. JavaSpaces offers transactions to the users to maintain data integrity [9]. PLINDA [3] is another implementation based on LINDA that has provided persistence for tuples. The features present in these implementations can be used to demonstrate that LINDA is capable of delivering the services of a simple distributed database.

4.1 Linda and the DDBS issues

Before one can positively argue in favor of using LINDA as a framework for implementing distributed databases, one must address how LINDA can cope with the issues that are common place in distributed databases (described in Section 3.1)

Fortunately, there has been extensive research in LINDA in the field of replication, concurrency control, and fault tolerance. These studies were performed with the intention of improving the scalability LINDA implementations and its suitability as a model for developing large scale open systems.

4.1.1 Replication

Replication was not a big concern when the original LINDA was proposed [10]. The nature of applications that were developed in LINDA did not require replication — LINDA was used in parallel computation (tightly coupled processing). However, as LINDA became more popular in the field of distributed system, scalability became a concern. Realizing the need for replication, Corradi *et Al.* [6] proposed a policy for replicating tuples in LINDA. Their proposal also include mechanisms for managing replicas in the system.

Corradi *et Al.* proposes that tuple spaces should be organized in a tree like structure where the leaf nodes of the tree represent the processes and the non leaf nodes of the tree represent the tuple spaces. Conceptually, a tuple is considered as an *out* request and an *in* request is created when a process requests a tuple. *in* and *out* requests of nodes are replicated along the path starting from the source node up to the root of the tree or to the node where a match (associative) between an *in* request and an *out* request occurs.

For an *in* operation, when a match occurs in some node, then the runtime system explores the sub tree down the path that goes to the source node of the tuple. During this traversal, it extracts the matching tuple from every node on its way to the source node. If there is no matching in any one of these nodes, the tuple has already been extracted. In this case, a failure message is sent to the node where the original match occurred.

The main issue in replication is maintaining data consistency. Here, the runtime system guarantees data consistency by imposing following the conditions that are met during the tuple matching process.

- a tuple (or *out* request) matches with only one *in* request. That means that if there is a tuple, then it should be retrieved (by *in*) by only one process and it should not be available for retrieval (by *in*) by other processes.
- a *in* request retrieves only one tuple (or matches with only one *out* request). Again, when a match occurs, the runtime system prefers matching occurring at source node even though matching could occur at any nodes other than the source node. The remaining matching tuples on the way are extracted and ignored and this is equivalent to deleting replicas.

The update propagation can also be achieved. It is important to note that a tuple cannot be updated. However, updating can be simulated to certain extent by doing an *in* on the tuple and doing an *out* of the new tuple having the updated value. During the execution of *in*, the runtime system extracts all the matching tuples, if any, in the subtree. A subsequent *out* operation, puts the new (updated) tuple in the tuple space based on the replication policy. This has the effect of update propagation in the sense that all the replicas are updated and thus data integrity is maintained.

The replication transparency is guaranteed by the runtime system. When requesting a tuple, processes are not aware of the replication and should not notice any difference in the system (except improved performance). From a process point of view, tuples are not necessary replicated.

In a distributed database using LINDA, tuples (representing records) could be replicated as above. However it may be interesting to consider partial replication techniques as described by Carriero and Gelernter [5].

4.1.2 Concurrency Control

Transactions typically maintain a private workspace that functions as a temporary buffer for values read from and written to the database. If a *commit* is executed, the buffer is synchronized with the database. On the other hand, if a *rollback* is executed then the buffer is destroyed.

Merrick and Wood [14] proposed the concept of *scopes* of tuples. A scope is a viewpoint through which certain tuples can be seen [14]. Scoping affects visibility of tuples, which can be used to simulate the private work space. By this method, imposing appropriate scopes, certain tuples can be made visible only to the DBMS and not to other processes. We can call these tuples as *private tuples*. These private tuples can be moved in and out of visibility using appropriate scope operations.

Scopes are created from atomic entities called names, which are created by processes at any time. A scope is represented as a set with name(s) as its element(s). The scopes models provide operations that create new scopes from existing ones. These operations are based on the idea of set operators. Before forming private tuples, appropriate scopes for the tuples need to be defined. The scope formed out of these private tuples may be used as a private workspace during transactions.

In addition to transactions, *locks* are essential to guarantee data consistency [1]. In a distributed environment, a transaction may involve multiple sites. As described in section 3.1, sub transactions may interfere with each other leading to concurrency problems. The basic idea behind locking is that a transaction must claim a lock, either read lock or write lock, before the executing the corresponding operation on a data item. The lock is released after executing the transaction.

Being a coordination language, LINDA provides us with synchronization mechanisms that can be used to implement locks. The primitives *in* and *out* can be thought of as implementing an implicit lock. *in* is a blocking call meaning that the request would wait until the tuple is available. Here, there is no need for an explicit lock request. Similarly, an *out* on a data item means an implicit lock release after which a process waiting for this data item will get it (if no other process is waiting for the same data item). Thus, it is guaranteed that only one process uses the tuple (if the tuple is retrieved by an *in*) at a time.

An explicit lock can be also be implemented by defining a tuple to be exclusively used as a *lock*. A process that needs to access a tuple x must obtain the lock by doing an *in* on the lock first. When a process is using tuple x, no other process can use that tuple because the lock used to grant exclusive access to that tuple was

already obtained by the former. Other processes that require tuple x will have to wait until the process using that tuple releases the lock by doing an *out* on the lock. Thus, the inherent blocking mechanism of LINDA is very helpful to implement a lock easily.

4.1.3 Fault Tolerance

The original LINDA did not have support for fault tolerance. There has been several proposals to add fault tolerance to LINDA. One such proposal is PLINDA, a variation of LINDA. PLINDA [3] was proposed with a notion of providing persistence to tuple space systems. Two of the key concepts added to the model are fault tolerance and resilient processes.

PLINDA uses the concept of *transactions* to guarantee consistent states for successful recovery after a failure. A transaction starts by executing an operation *xstart* and ends by executing an operation *xcommit*. The effects of executing primitives are delayed until a *xcommit* is executed. After a failure, PLINDA restores the previous consistent states which have been periodically saved to the disk by *checkpointing*. During checkpointing, the runtime system saves only the latest committed tuple space data to the disk and not the uncommitted ones. When a tuple space server fails, it restores the latest checkpointed state from the storage device by executing *rollback*. After *rollback*, the PLINDA system resumes execution from that state, by re-spawning the required processes to take over the failed processes, if any. By this way, fault tolerance is guaranteed by PLINDA.

In another implementation of LINDA, by Patterson *et Al.* [16], fault tolerance is achieved primarily by tuple replication. In this implementation, all tuples have a *fingerprint*, a sequence number generated by the source node, that uniquely identifies the tuple. A new tuple is copied to all the nodes of the *sub space*, which is a logical collection of all tuple spaces used by a particular application. During an *in* operation, all the replicated tuples are deleted by the system in addition to providing the matching tuple to the requesting process. When a node containing a tuple fails, the tuple is still available in other nodes of the subspace.

The nodes communicate with each other by passing messages. Messages are also used to reconciliate the replica after a failure. All the messages in the system are uniquely identifiable by a sequence number called *footprint*. When a node fails, a special daemon *repair-manager* detects the failure and takes necessary actions to synchronize the messages (thereby tuple replication messages) sent by the failed node to the remaining nodes of the subspace. The synchronization guarantees data consistency after a failure.

4.2 A Simple Architecture

We describe a conceptual view of a very simple potential architecture (Figure 1) that would work with SQL requests and responses. The characteristics of the architecture described in the section 4.1 can be used to extend this architecture to a complete framework for a distributed database system.

The central part of the architecture is what we call *DBEngine*, which generates the internal tables and mappings for the framework. The DBEngine and the LINDA server are the two main constituents of the architecture.

4.2.1 DBEngine

DBEngine consists of a set of APIs that handle all the SQL queries from the user. The LINDA server accepts requests from the DB Engine and retrieves the results (tuples) based on associative matching, which is functionally equivalent to accessing records in a database. DBEngine provides transparency to the user by converting all SQL statements into series of LINDA primitives. We use the term *LindaSQL* to refer to such LINDA primitives generated by the DBEngine.

The DBEngine may reside on the client side for the following advantages:

- DBEngine validates all SQL queries and sends only the correct LindaSQL to the server. Thus error handling is done on the client side, which reduces the server load and network traffic.
- DBEngine has a local tuple space (a tuple space that no one except the DBEngine is aware of) that acts as a cache. So, the DBEngine after obtaining the required information, can use its local tuple space to do further processing like data formatting without the need of contacting a server.

In a possible DBLinda[1] the implementation of the DBEngine is the most complex job. The complexity of this component lies on the fact that SQL statement will have to be translated into LINDA operations. Since it is not clear whether there is a relation between a SQL statement and a set of the Linda primitives, one would have to define (maybe formally) this equivalence before taking on the job of implementing the DBEngine.

For instance, how would a join be translated into LINDA operations? Standard LINDA primitives may not be expressive enough to deal with more complex SQL operations. However, there are variants on LINDA that may be able to express these. One of such variant is the LogOp model [19] which improves the expressiveness of LINDA.

The DBEngine is envisioned as having five components (as depicted in Figure 1):

Syntax checker: The syntax checker checks for the correctness of the SQL syntax. The output of the syntax checker is either 'OK' or 'Invalid Syntax'. If the output is 'OK' then the SQL request passes to the Validator else a suitable error message is thrown to the application and the DBEngine starts over.

Validator: The Validator validates the SQL obtained from the syntax checker for any invalid requests like referring a table and/or attribute that does not exist and so on. The output of the Validator is either 'OK' or 'Invalid SQL'. If the output is 'Invalid SQL', suitable error message is displayed

[1] A conceptual model for databases supported by LINDA.

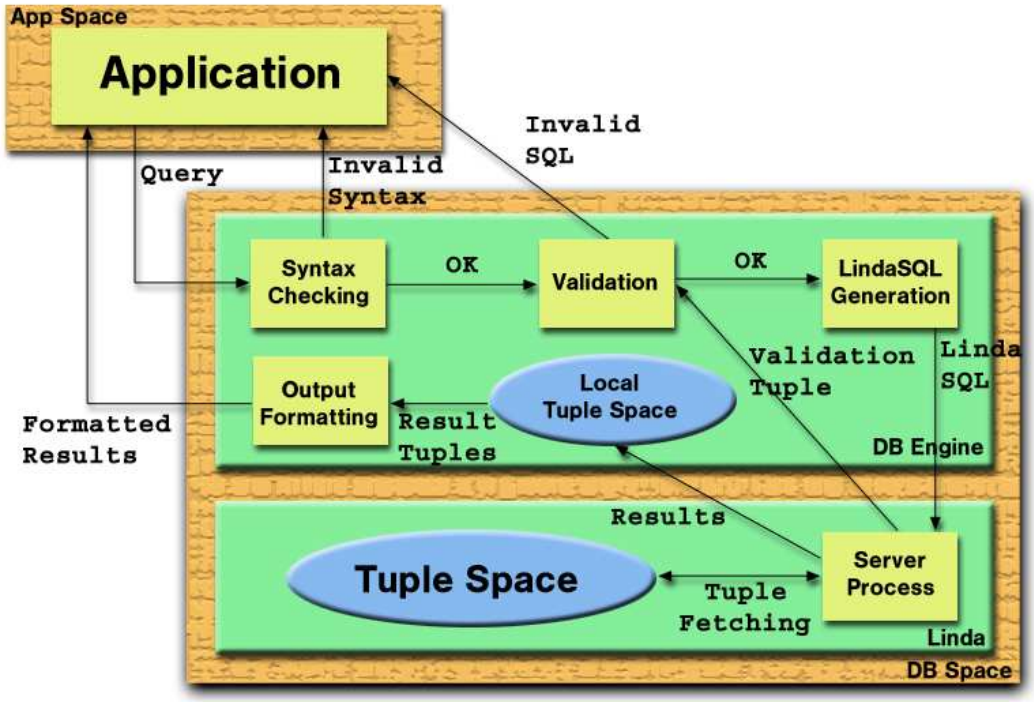

Figure 1: Conceptual architecture that shows LINDA supporting database queries

to the application and the DBEngine starts over. If the output is 'OK', the SQL request is sent to the LindaSQL Generator.

LindaSQL generator: This component gets the SQL request from the Validator and generates corresponding LindaSQL based on the representation of data in LINDA as described in the next section.

Local tuple space: While validating, the Validator uses the local copy of the tables obtained from the server. These tables may be temporarily stored in the local tuple space. Similarly, the results obtained from the server may be stored locally in the local tuple space and operations like data mapping can be done locally without having a necessity to operate on the server. This may improve the performance of the server by simulating a cache on the client side.

Data mapper: Data mapper converts the results obtained from the server and maps them to a form similar to the results of a SQL query, which is conceptually the same as the reverse of the operation of LindaSQL Generator.

4.2.2 Representation of Relational Database objects

Representation of Tables : Tables of a database can be represented in LINDA using a tuple space for each table. Thus, all the records of the table become the tuples of the tuple space. By doing so, all the tuples in a tuple space are 'logically' tied up similar to the logical relation of records with a table. The table name becomes the name of the tuple space. Each tuple space name is stored as a tuple in a special tuple space named 'NameSpace', which may act like a metadata.

Representation of Attributes : The information about each attribute of a table can be stored as a special tuple perhaps with the first field being 'TD' (for Table Definition) inside the tuple space that represents the table. Other information such as attribute name and data length can also be stored in this tuple as subsequent fields.

Representation of Records : Records may be represented as tuples with the first field being a value 'T'(for Tuple). The order of the fields in this tuple can be the same as that of the attributes of the table.

Representation of Other Metadata : Other metadata such as constraint information of an attribute can be stored as tuples with identification information at the front of the tuple.

So far, we have discussed about our conceptual architecture that makes LINDA act as a very simple database. The architecture seems to be very basic given the tremendous capabilities of modern databases. Clearly substantial improvements need to be done to make it LINDA

behave like a real database management system. However, the real benefit of such approach comes from the simplicity and power offered my the LINDA model in dealing with coordination issues such as transaction control. This architecture, along with the features of (extended) Linda model discussed in section 4.1 forms the basis of a possible framework for developing distributed databases. We believe that this framework could help reduce the complexities of coordination issues while developing a distributed database system.

5. FUTURE WORK AND CONCLUSION

Data distribution in distributed databases and the characteristics of coordination languages have similarities, which could lead to an elegant distributed database system with the advantages of data distribution and the expressiveness of coordination languages. A possible LINDA based framework for distributed database is presented in this paper.

Clearly, the framework explained this paper is primitive and requires further efforts to formalize the ideas put forth and to build a system upon these ideas. A new extended LINDA model implementing various research ideas discussed in this paper would be a good future work. It would also be useful to make such LINDA based distributed database system compatible with existing distributed database systems. That way, LINDA can slowly enter and make a stronger impact in the field of distributed databases.

6. REFERENCES

[1] D. Bell and J. Grimson. *Distributed Database Systems*. Addison-Wesley, Wokingham, 1992.

[2] P. A. Bernstein and N. Goldman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–221, June 1981.

[3] T. Brown, K. Jeong, B. Li, S. Talla, P. Wyckoff, and D. Shasha. PLinda User Manual. Technical Report TR1996-729, New York University, December 1996.

[4] P. Butcher, A. Wood, and M. Atkins. Global Synchronisation in Linda. *Concurrency: Practice and Experience*, 6(6):505–516, 1994.

[5] N. Carriero and D. Gelernter. The S/Net's Linda Kernel. *ACM Transactions on Computer Systems*, 4(2):110–129, 1986.

[6] A. Corradi, L. Leonardi, and F. Zambonelli. Strategies and protocols for highly parallel Linda servers. *Software Practice and Experience*, 28(14):1493–1517, Dec. 1998.

[7] C. J. Date. *An Introduction to Database Systems*. Addison-Wesley, Reading, Mass., fifth edition, 1991.

[8] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Benjamin/Cummings, Redwood City, 3 edition, 2000.

[9] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces Principles, Patterns and Practice*. Addison-Wesley, Reading, MA, USA, 1999. The Jini Technology Series.

[10] D. Gelernter. Generative Communication in LINDA. *ACM transactions in programming languages and systems*, 1:80–112, 1985.

[11] D. Gelernter. Multiple Tuple Spaces in LINDA. In *Proc. of PARLE 89*, pages 20–27. Springer-Verlag, 1989.

[12] IBM Corporation. *T Spaces Programmer's Guide*, 1998. Eletronic version only. http://www.almaden.ibm.com/cs/TSpaces/.

[13] T. W. Malone and K. Crowston. The interdisciplinary study of coordination. *ACM Computing Surveys*, 26(1), Mar. 1994.

[14] I. Merrick and A. Wood. Coordination with scopes. In *Proceedings of the 2000 ACM symposium on Applied computing 2000*, volume 1, pages 210–217, Como, Italy, March 2000. ACM.

[15] M. T. Özsu and P. Valduriez. Distributed and parallel database systems. *ACM Computing Surveys*, 28(1):125–128, Mar. 1996.

[16] L. L. Patterson, R. S. Turner, R. M. Hyatt, and K. D. Reilly. Construction of a fault-tolerant distributed tuple-space. In E. Deaton, K. M. George, H. Berghel, and G. Hedrick, editors, *Proceedings of the ACM/SIGAPP Symposium on Applied Computing*, pages 279–285, Indianapolis, IN, Feb. 1993. ACM Press.

[17] A. Rowstron. WCL: A co-ordination Language for geographically distributed agents. *World Wide Web*, 1:167–179, 1998.

[18] A. Rowstron and A. Wood. Solving the LINDA Multiple `rd` Problem. In P. Ciancarini and C. Hankin, editors, *Proc. of 1st. International Conference - COORDINATION 96*, Lecture Notes in Computer Science, pages 357–367. Springer-Verlag, 1996.

[19] J. Snyder and R. Menezes. Using Logical Operators as an Extended Coordination Mechanism in Linda. In F. Arbab and C. Talcott, editors, *Proceedings of the 5th International Conference, COORDINATION 2002*, number 2315 in Lecture Notes in Computer Science, pages 317–331, York, UK, April 2002. Springer-Verlag.

[20] A. Wood and A. Rowstron. Deadlock and Algorithm Design: Stable Marriages in LINDA. Technical report, The University of York, 1996.

[21] P. Wyckoff, S. W. McLaughry, T. J. Lehman, and D. A. Ford. T Spaces. *IBM Systems Journal*, 37(3), 1998. Special Issue on Java Technology.