Implementation and Analysis of a JavaSpace Supported by a Relational Database

Geoffrey C. Arnold, Gregory M. Kapfhammer, and Robert Roos

***Abstract*—The combination of the Jini network technology and the JavaSpaces object repository provides an exceptional environment for the design and implementation of loosely coupled distributed systems. We explore the strengths and weaknesses of the most popular implementation of a persistent JavaSpace. We report on the design, implementation, and analysis of RDBSpace, a JavaSpace that is supported by a relational database. Our experimental results indicate that it is possible to build an efficient and scalable JavaSpace that relies on a relational database back-end.**

***Index Terms*—Distributed System, Tuple-space, JavaSpace, Relational Database**

I. Introduction

DISTRIBUTED computation normally relies upon a collection of cooperating processes that are distributed across a network of machines [1]. For certain sets of complex problems, distributed computations offer numerous advantages over their sequential counterparts, including performance improvements, enhanced scalability, resource sharing, fault tolerance, and system design elegance [1]. Unfortunately, the implementation of distributed applications often introduces a variety of problems. Designers of distributed systems must concern themselves with the heterogeneity of the software and hardware platforms on which the application is to be executed, while also addressing the problems that are imposed by network environments such as latency, concurrency, and partial failure [1], [2].

A. The Tuple-Space Concept

In the early 1980s, Gelernter et al. broke away from the traditional models of distributed computation and introduced the concept of a *tuple-space* [3],[4],[5]. Figure 1 depicts a tuple-space that facilitates process interaction through the exploitation of the concept of a shared memory [6]. By combining a persistent data store with a small set of operations, Gelernter pioneered the concept of loosely coupled process communication that could be independent of both space and time [1], [3], [4]. Initial implementations of the tuple-space concept in the Linda coordination language produced benchmarks similar to traditional message passing models, while maintaining the ability to create complex parallel and distributed applications [1].

B. Tuple-Space Concerns

The tuple-space has since become the inspiration of numerous commercial implementations of the concept, including the JavaSpaces service from Sun Microsystems. By combining the power of the Java programming language with the flexibility of the Jini network technology, JavaSpaces has become one of the most commercially viable tuple-space implementations available today. Unfortunately, there are shortcomings in Sun's implementation of a persistent tuple-space. The implementation uses an object-oriented database back-end for persistent tuple storage. Our initial experiments indicate that the current JavaSpaces back-end has introduced scalability and performance issues into the implementation [7]. The chosen back-end database also has a limited availability of advanced database features such as data replication. Finally, all of the upgrade paths for the back-end database used in the current JavaSpaces implementation require the purchase of commercial object-oriented databases.

Department of Computer Science
Allegheny College, Meadville PA, 16335, USA
Email: geoffrey@geoffreyarnold.com
Email: {gkapfham,rroos}@allegheny.edu

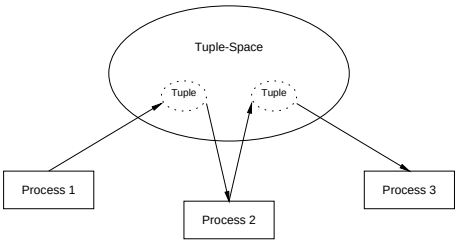

Fig. 1. Tuple-space based process communication.

C. Related Work

We have created an implementation of a persistent JavaSpaces service that utilizes an open-source relational database back-end to persistently store tuples. Through the usage of native database functionality, our implementation supports advanced fault protection features such as data replication. After evaluating different architectures for a JavaSpaces service, we implemented *RDBSpace*, a JavaSpace that is supported by a relational database. Our JavaSpace employs both a novel architecture to ensure the best possible performance gains.

To the best of our knowledge, our implementation and analysis of a free, open-source JavaSpaces service supported by a relational database back-end is the first of its nature. However, Larsen et al. have focused on improving the fault-tolerance and scalability of the existing JavaSpaces implementation by directly adding mechanisms for replication management [8]. Furthermore, Noble et al. have conducted several empirical analyses to evaluate the applicability of JavaSpaces for different types of scientific computation [6]. Finally, there are alternative commercial implementations of a JavaSpace, such as GigaSpaces [9].

In Section II, we review the Jini network technology and the JavaSpaces object repository. Next, Section III describes our alternative implementation of a JavaSpace. In this section we discuss a service design spectrum for JavaSpaces and explore

the architecture of our implementation of a JavaSpace. Also, we explain our approach for using a relational database to support a persistent JavaSpace. Finally, Section IV offers an empirical analysis of our JavaSpace and we conclude with some future avenues for research in Section V.

II. Jini and JavaSpaces

A. Jini Network Technology

As explained by Oaks et al., "Jini allows cooperating devices, services, and applications to access each other seamlessly, to adapt to a continually changing environment, and to share code and configurations transparently" [10]. Jini is not a network service itself, but rather a set of specifications that enables service interaction [10]. The Jini specification describes a collection of protocols that allow clients and services to dynamically interact in an attempt to effectively solve problems in a distributed fashion. Jini not only allows for the creation of remote services, but also provides a self-healing framework for the maintenance of services [10]. A more detailed discussion of the Jini network technology can be found in [10] and [11].

B. JavaSpaces

The JavaSpaces service is a shared object warehouse that can be utilized by a group of Jini clients and services. Unlike other tuple repositories, JavaSpaces requires that the objects residing within a space act as passive data. Thus, an object that is stored in a JavaSpace can only be modified if it is explicitly removed, updated, and returned to the space [1]. The JavaSpaces service can persistently stores tuples that can later be accessed by clients that submit queries for specific types of Java object. For a detailed review of the JavaSpaces service, refer to [1].

B.1 JavaSpaces Operations

Figure 2 depicts the three primary operations that facilitate JavaSpaces interaction. Objects are written to, and read or taken from a space. Since these operations support the creation of solutions for a large domain of problems, JavaSpaces are said to be *simple* and *expressive* [1]. Additional functionalities of JavaSpaces include a tuple notification mechanism and a snapshotting technique designed to reduce the performance impact of repeated queries.

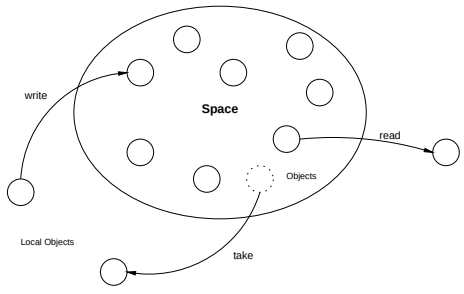

Fig. 2. An overview of JavaSpaces operations: `read`, `write`, and `take`.

B.2 Associative Lookup

Associative lookup is the mechanism by which objects are retrieved from a JavaSpace. Before a client can retrieve an entry from a space, it must first create a template to describe the desired object(s). When creating a template, the client explicitly specifies values for object fields [1]. The JavaSpaces object repository also facilitates wildcard matching by exploiting the usage of `null` field values in templates [1].

Associative lookup is performed by comparing each field of the serialized form of entries within a space and the provided template [1]. Instead of examining the contents of an object's fields, the JavaSpaces service compares the bytes of the serialized form of each field [1]. It is important to note that any field annotations added by Java object serialization are ignored when matching objects [1].

III. An Alternative JavaSpaces Implementation

We have implemented RDBSpace so that it utilizes a relational database back-end for the storage of tuples. Our design goals are briefly explained below:

Speed To be a viable implementation, the service should meet or exceed performance benchmarks set by the default persistent JavaSpaces implementation.

Scalability As hardware performance continues to improve, so should the performance of the service.

Portability It should be easy to enable our implementation to exploit the capabilities of different operating systems and relational database servers.

Advanced Functionality It should be possible to utilize features like data replication and extended backup capabilities in the chosen implementation.

A. JavaSpaces Service Architectures

By combining the design requirements for our JavaSpace with the capabilities of the Jini network technology, we can form a service design spectrum for JavaSpaces. At one end of the spectrum is a *client-centric* service model in which the service is downloaded and run completely within the the client's own Java Virtual Machine (JVM) [12]. At the other end of the spectrum is a *server-centric* model in which most of the processing takes place within the remote service's JVM [12]. The *hybrid* model incorporates facets of both the client-centric and the server-centric models. This JavaSpaces architecture still performs much of its processing within the client's JVM, while using an external data-source to store objects. Although each of the models require very different implementations, all architectures imply that clients must be able to access the well-known interface to the JavaSpace service. Our implementation of RDBSpace uses the server-centric model during a client's usage of the `read` and `take` operations. Also, RDBSpace employs the hybrid model when a client performs a `write` operation.

A.1 The Client-Centric JavaSpaces Service

In a *client-centric* JavaSpace service architecture, all of the processing occurs within the client's JVM. Jini provides the

mechanism for the lookup and discovery of the service, pointing clients to the necessary code-base needed to download the JavaSpaces service. As noted by Newmarch, client-centric services are well-suited to applications that are independent of time and location, such as is the case of a JavaSpaces service [12]. Furthermore, the client-centric architecture essentially eliminates the JavaSpace as single point of failure in a distributed system. However, the distributed synchronization of the numerous JavaSpace clients makes it difficult for the client-centric JavaSpaces service to facilitate concurrent access by multiple clients [13]. A diagram of the client-centric service architecture is presented in Figure 3.

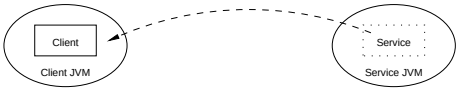

Fig. 3. Diagram of a client-centric service.

A.2 The Server-Centric JavaSpaces Service

In a *server-centric* JavaSpaces service architecture, a proxy to the service is downloaded to the client upon request in much the same fashion as the client-centric architecture. However, the proxy is not a full implementation of the JavaSpaces service. Instead, it is simply a link to a remote service that provides access to the desired methods via the use of remote method invocation (RMI) stubs and skeletons. Providing that the server-centric JavaSpaces service is implemented in a fashion that correctly handles synchronization issues, it can easily be shared among multiple clients. A potential shortcoming of the server-centric design is the introduction of a single point of failure in the distributed system. Also, a server-centric service can suffer performance bottlenecks as the number of clients concurrently accessing the service increases. The experimental analyses conducted by Zorman indicate that this performance bottleneck can limit the architecture's scalability [14]. Figure 4 outlines the details of a server-centric service.

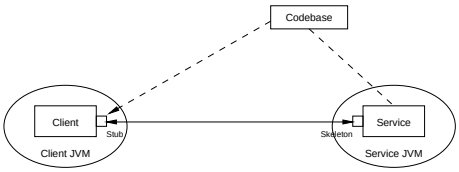

Fig. 4. Diagram of a server-centric service.

A.3 The Hybrid JavaSpaces Service

The *hybrid* JavaSpaces service architecture represents an attempt to capitalize on the strengths of the client-centric and server-centric architectures. Although a hybrid service performs much of its processing within the client's JVM, it also utilizes an external data-source for the storage of objects. Communication between the client and the external service is commonly accomplished through the usage of a proprietary protocol. For example, Java natively supports database access through drivers that implement the Java Database Connectivity (JDBC) application programmer interface (API). Hence, these drivers could be used to facilitate the communication between a hybrid service and an external data-source. By transferring responsibility to existing technologies, the hybrid service can dramatically decrease the complexity of a service, while utilizing proven technologies to provide performance enhancements and additional features. A diagram of this hybrid service design is presented in Figure 5.

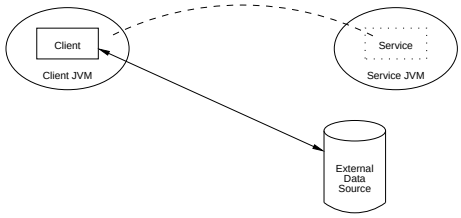

Fig. 5. Diagram of a hybrid service.

A.4 Architecture Analysis

Initial experiments with the different architectures of a JavaSpace confirmed our suspicions about the strengths and weaknesses of the three architectures for a JavaSpace [7]. Since the client-centric architecture was plagued with implementation concerns related to synchronization, we compared the service-centric and the hybrid models. In our initial analyses, the server-centric JavaSpaces architecture exhibited far less of a slowdown during `take` operations, while the hybrid service was crippled by network latencies [7]. The experiments with the hybrid service clearly indicate the costs that are accrued when a distributed system ignores the heuristic of "keep the computation close to the data" [13]. To this end, we chose to implement RDBSpace in a manner that combined the strengths of both the server-centric and hybrid architectures.

B. Object Storage in a Relational Database

The term *object serialization* describes the ability to encode objects as raw byte streams in such a way as to preserve the state of the object for later usage [15]. During the serialization process, all data that comprises an object, including references to other objects, is stored in the byte stream in such a way that upon reconstitution the object is returned to its exact state before serialization [15]. It is serialization that facilitates the storage of objects in a relational database.

B.1 Object Storage Schema

The storage approach for our persistent JavaSpace uses the database as a metaphorical filing cabinet for objects. Instead of building complex database schemas where tables express relations between objects, we simply file objects of different types into separate tables. However, our filing cabinet approach cannot support the associative lookup techniques provided by JavaSpaces without additional information about the type hierarchies associated with the objects stored in the relational database. Since objects represented in the space are only stored

as byte streams in the database, the standard structured query language (SQL) does not support JavaSpaces template matching semantics.

In order to effectively support JavaSpace queries based upon templates, we maintain an external object hierarchy in our conceptual filing cabinet. Figure 6 shows the relation between the object hierarchy and its corresponding object tables. The object hierarchy is persistently stored in the database as a form of meta-data. Therefore, it does not have to be recreated across multiple invocations of the JavaSpace. Also, our JavaSpaces implementation stores additional meta-data in the database to describe certain properties (such as lease duration, etc.) of objects that were recently placed in the space.

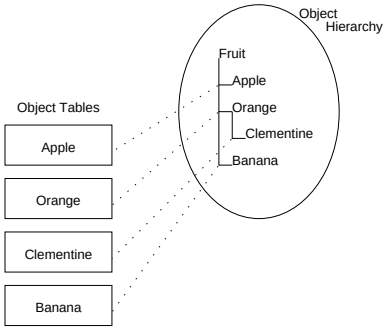

Fig. 6. The object filing cabinet.

B.2 Implementation Details

Once we had determined our approach for object storage in a relational schema, developing our JavaSpaces service became as simple as implementing the `net.jini.space.JavaSpace` interface. Refer to [1] and [10] for a overview of the JavaSpaces API.

Algorithm *writeEntry*(E)
($*$ writes an entry to a JavaSpace $*$)
Input: A `net.jini.core.entry.Entry` E
Output: A `net.jini.core.lease.Lease` L
1. **if** $E =$ **nil**
2. **then stop** ($*$ do not write blank entries $*$)
3. **else** write E to the space and assign lease L
4. **return** L

Fig. 7. Algorithm `writeEntry`.

At an abstract level, we can view our JavaSpace as implementing the three primary JavaSpaces operations, `write`, `readIfExists`, and `takeIfExists`, with the two algorithms `writeEntry` and `findEntry`. The algorithm for `writeEntry` is presented in Figure 7. The high-level algorithm for `findEntry` is provided in Figure 8.

Since the `findEntry` algorithm only returns an object if it exists in the space at the time of the request, it accurately describes the non-blocking `readIfExists` and `takeIfExists` operations. The `waitForEntry` algorithm can easily use `findEntry` to implement the blocking `read` and `take` operations provided by a JavaSpace. Figure 9 depicts the algorithm for these functions. For a detailed discussion of our implementation of the JavaSpace interface, please refer to [7].

Algorithm *findEntry*(T)
($*$ finds an entry in a JavaSpace matching a template T $*$)
Input: A `net.jini.core.entry.Entry` template T
Output: A matching `net.jini.core.entry.Entry` E
1. **if** $T =$ **nil**
2. **then return** any entry E ($*$ **nil** template matches any entry $*$)
3. **else if** found an entry E that matches template T
4. **then** remove E from space if taking
5. **return** E
6. **else return nil** ($*$ return **nil** if no match found $*$)

Fig. 8. Algorithm `findEntry`.

Algorithm *waitForEntry*(T, P)
($*$ waits for period P for an entry matching a template T $*$)
Input: A `net.jini.core.entry.Entry` template T
Input: A time period P
Output: A matching `net.jini.core.entry.Entry` E
1. **repeat**
2. let E be the value returned from *findEntry*(T)
3. **until** $E \neq$ **nil or** P has expired
4. **return** E

Fig. 9. Algorithm `waitForEntry`.

B.3 Database Configuration

The current implementation of RDBSpace uses the MM mySQL JDBC driver and the mySQL relational database management system (RDBMS) [16], [17]. Our JavaSpaces service implementation uses standard SQL database queries to manipulate the objects in the database. RDBSpace uses column index optimizations to improve the performance of the JavaSpace operations. Moreover, we have used a mySQL-specific optimization that allows for the usage of asynchronous `insert` statements during the processing of `write` operations. We believe that there is the potential to incorporate a significant number of other relational database performance enhancements to further improve the efficiency of our space. RDBSpace dynamically creates object tables as needed, and updates the object hierarchy accordingly. Each client refreshes its object hierarchy before every space operation in order to ensure that it contains the latest information about the current type hierarchy.

IV. Analysis

Sun Microsystems has developed an implementation of a persistent JavaSpace calleded *Outrigger*. To effectively formulate a comparison between our persistent JavaSpaces service implementation and Sun's own persistent JavaSpace, we calculated low-level input/output benchmarks with the help of the Tonic benchmarking system [6]. Service scalability was measured using a number of systems with notably different hardware con-

figurations. Each machine was connected via 100 MB/s full duplex on board Ethernet to a semi-loaded local area network. In an attempt to abstract the JavaSpaces services from all other Jini components, an independent machine was configured to act as both the lookup host and the service code-base.

From this configuration we began our first experiment to formulate an overall comparison of the performance differences between the two implementations. The first test, called the Null IO benchmark, attempts to show the raw speed of an implementation by performing space operations using a small object of that was only 343 bytes after serialization. The test recorded the average amount of time needed to sequentially write and take 100 small objects from a space over 10 iterations. An RDBSpace and a JavaSpace were both started on an experiment machine, and Tonic was run from a remote control machine. The test was repeated for each implementation on all experiment machines, and the benchmarks were recorded. The results are displayed in Figure 10.

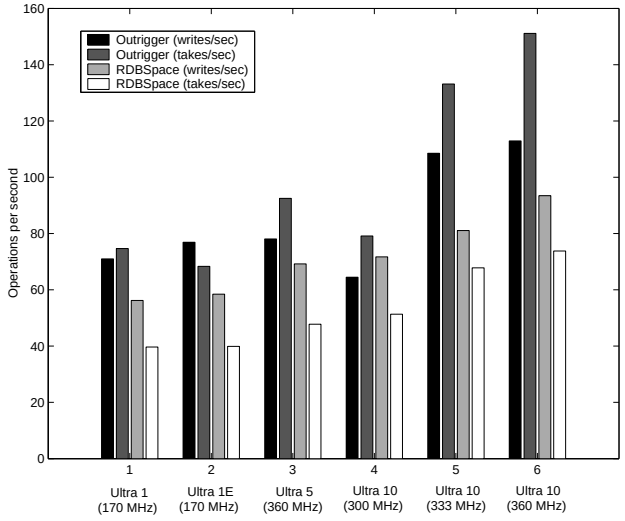

Fig. 10. Results of the Null IO benchmark.

Our initial experiments show that Outrigger performs better than RDBSpace on a wide variety of different workstations. However, it is important to note that both Outrigger and RDBSpace scale when they are executed on machines with increasingly better hardware configurations. It is interesting to note that RDBSpace always performs more `write` operations per second than `take` operations. This characteristic is due to the fact that RDBSpace adheres to a hybrid architecture during `write`s and a server-centric architecture during `take`s. When a small object is placed inside of our JavaSpace through the usage of the `write` operation, the direct access to the relational database back-end appears to improve performance.

In order to further explore the strengths and weaknesses of RDBSpace, our second performance test was conducted using large object during space operations. This test, called the Array IO benchmark, utilizes an array of 1000 double values that are approximately 8485 bytes after serialization. Our test environment was configured in exactly the same manner as the one that was used during the Null IO benchmark. The results of the experiment are displayed in Figure 11.

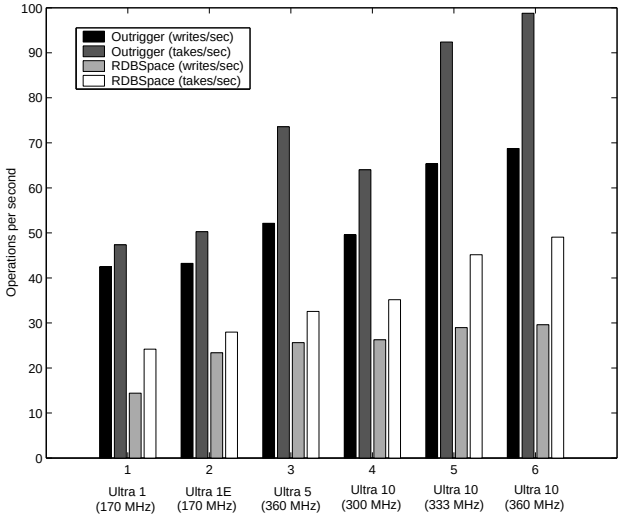

Fig. 11. Results of the Array IO benchmark.

Our second experiment also showed that Outrigger performs better than RDBspace on many different types of machines. Once again, we see that both Outrigger and RDBSpace scale when the space is moved to a higher performance workstation. In this experiment, it is important to observe that RDBSpace always performs more `take` operations per second than `write` operations. Interestingly, this is exactly the opposite of the behavior that was observed during the usage of the Null IO benchmark. As noted in Section III-B.1, our implementation of RDBSpace must write meta-data to the relational database to accurately capture the complete semantic meaning of a Java object. Apparently, the size of the the meta-data and the actual object detract from any perforance gains that could be realized through the usage of the hybrid architecture. Clearly, RDBSpace is able to `take` more objects per second because it is not responsible for actually returning all of the meta-data to the requesting client.

In our next experiment, we varied the number of objects that were sequentially written to Outrigger and RDBSpace. In this experiment, we used both `null` objects and double arrays of size 1000. Furthermore, we chose to perform each benchmark on a Sun Ultra 10 360 MHz workstation. The results of this experiment are shown in Figure 12. Since the current implementation of RDBSpace does not use the `snapshot` operation to streamline the frequent reading or taking of the same Java object, it does not perform as well as Outrigger. Indeed, Outrigger is able to perform more `read` and `take` operations per second as the number of sequentially written objects increases. Clearly, this is because Outrigger can avoid the frequent serialiazation of Java objects that must be incurred by RDBSpace.

V. Conclusion

The current implementation of RDBSpace does not attain the performance benchmarks that have been set by Sun's imple-

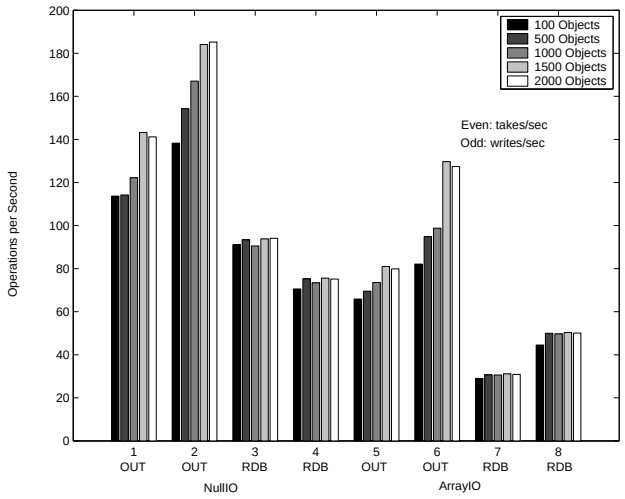

Fig. 12. Results from the Number of Objects Benchmark.

mentation of a persistent JavaSpace. However, RDBSpace does exhibit scalability when the service is moved to higher performance host machines. We believe that RDBSpace, in its current form, shows potential for being competitive with other persistent spaces. Also, the native data preservation features provided by our service, such as simple backup facilities and data replication for increased availability, add to the significance of RDBSpace. Moreover, the replication facilities provided by standard relational database backends provide an avenue for easily introducing fault tolerance into JavaSpaces. In future research, we plan to implement the `snapshot` operation in RDBSpace and then empirically evaluate the benefits of using this operation. Also, we would like to experiment with the usage of RDBSpace is several real-world applications.

REFERENCES

[1] Eric Freeman, Susanne Hupfer, and Ken Arnold, *JavaSpaces: Principles, Patterns, and Practice*, Addison-Wesley, Reading, Massachusetts, 1999.

[2] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall, "A note on distributed computing," Tech. Rep. SMLI TR-94-29, Sun Microsystems Laboratories, Inc., November 1994.

[3] Nicholas Carriero and David Gelernter, "A computational model of everything," *Communications of the ACM*, vol. 44, no. 11, pp. 77–81, November 2001.

[4] David Gelernter and Nicholas Carriero, "Coordination languages and their significance," *Communications of the ACM*, vol. 35, no. 2, pp. 97–107, February 1992.

[5] David Gelernter, *An Integrated Microcomputer Network for Experiments in Distributed Programming*, Ph.D. thesis, SUNY Stony Brook, Department of Computer Science, 1983.

[6] Michael S. Noble and Stoyanka Zlateva, "Scientific computation with javaspaces," in *Proceedings of the 9th International Conference on High Performance Computing and Networking*, June 2001.

[7] Geoffrey Arnold, "Trading space: Implementation and analysis of a relational database javaspaces service," Tech. Rep. CS02-01, Allegheny College, Department of Computer Science, March 2002.

[8] Jakob Eg Larsen and Jesper Honig Spring, *GLOBE: A Dynamically Fault-tolerant and dynamically scalable distributed tuplespace for heterogeneous, loosely couple networks*, Ph.D. thesis, University of Copenhagen, Department of Computer Science, October 1999.

[9] GigaSpaces Technologies Ltd., "Gigaspaces platform," 2002, http://www.gigaspaces.com/index.htm.

[10] Scott Oaks and Henry Wong, *JINI in a Nutshell*, O'Reilly and Associates Inc., Sebastopol, California, 2000.

[11] Ken Arnold, Bryan O'Sullivan, Robert W. Scheifler, Jim Waldo, and Ann Wollrath, *The Jini Specification*, Addison-Wesley, Inc., Reading, MA, 1999.

[12] Jan Newmarch, *Jan Newmarch's Guide to JINI Technologies*, Jun 2001, http://jan.netcomp.monash.edu.au/java/jini/tutorial/Jini.xml.

[13] Andrew S. Tanenbaum and Maarten van Steen, *Distributed Systems: Principles and Paradigms*, Prentice Hall, Inc., Upper Saddle River, NJ, 2002.

[14] Brian Zorman, "Creation and analysis of a javaspaces-based distributed genetic algorithm," Tech. Rep. CS02-18, Allegheny College, Department of Computer Science, March 2002.

[15] John Zukowski, "Advanced object serialization," Aug 2001, http://developer.java.sun.com/developer/technicalArticles/ALT/serialization/.

[16] "MM mySQL JDBC drivers," 2001, http://mmmysql.sourceforge.net/.

[17] Randy Jay Yarger, George Reese, and Tim King, *MySQL and mSQL*, O'Reilly and Associates, 1999.