A Performance Evaluation of Distributed Algorithms on Shared Memory and Message Passing Middleware Platforms

Sanjay P. Ahuja, Roger Eggen and Anjani K. Jha
Department of Computer and Information Sciences
University of North Florida
Jacksonville, FL – 32224.
E-mail: {sahuja, ree, jhaa0001}@unf.edu

Keywords: JavaSpaces, CORBA, shared memory, message passing, middleware.

Received: Januar 19, 2005

The fundamental characteristics of a distributed computing environment are heterogeneity, partial failure, latency and difficulty of "gluing together" multiple, independent processes into a robust, scalable application. JavaSpaces, which is a shared memory paradigm, provides high-level coordination mechanism for Java easing the burden of creating distributed applications. A large class of distributed problems can be approached using JavaSpaces simple framework. JavaSpaces allows processes to communicate even if each was wholly ignorant of the others.

Common Object Request Broker Architecture (CORBA), on the other hand, is a standard developed by the Object Management Group (OMG), which allows communication between objects that are written in different programming languages. It provides common message passing mechanism for interchanging data and discovering services. In this project, we compare these two platforms for distributed computing both quantitatively and qualitatively.

To do so, we analyze the performance of distributed algorithms that divide a task into small sub-tasks which are distributed over a network of computers to perform computations in parallel. Specifically, we measure the performance of an insertion sort algorithm of O (n2) complexity on both the JavaSpaces and CORBA platforms. We measure latency, speed-up, and efficiency and analyze the implications on overall performance and scalability.

Povzetek: Članek opisuje ovrednotenje porazdeljenih algoritmov na platformah.

1 Introduction

Client/server and multi-tier models operating within a single business enterprise have given way to an Internet/Web environment where services are provided by nodes scattered over a far-flung network. Next generation of network interaction is emerging that place unprecedented demands upon existing network technologies and architectures. For example, participants in one network will need to directly access and use the services provided by participants in another network. It is in this distributed environment - one of mind-numbing complexity driven by geometric increases in scale, rate of change, and multiplicity of participant interactions that technologies such as JavaSpaces and CORBA present competing options. Software architects, engineers, and distributed systems designers have multiple competing options and opportunities, each providing advantages and disadvantages.

Distributed systems are hard to build. They require careful thinking about problems that do not occur in single process computation. The early solutions to the challenges facing distributed computing involved sockets which pass messages between client and server. This kind of communication required the application programmer to know the Berkeley Socket API. Applications developed were onerous leading to the next generation of message passing protocols such as RPC (Remote Procedure Call), MPI (Message Passing Interface) and PVM (Parallel Virtual Machine) which hid the low-level socket communication, but the applications tended to be tightly coupled as with socket programming. In other words, the client-side application invoking procedures on the server-side needed to know exactly what services the server was prepared to offer the client. Such distributed systems were less robust and could not withstand partial failures. The advent of object-oriented languages such as C++ and Java led to the development of the distributed object computation platforms such as DCOM, CORBA and RMI. While these are excellent in that they provide an object-oriented framework for developing distributed systems, these are essentially RPC-oriented, tightly coupled, message passing systems with the ability to marshal objects when the objects are used as parameters in the method calls. These protocols

left the task of object persistence and recovery from partial failure to the developers and the application designers [8]. This has led to the development of the JavaSpaces model by the Java Development community, which is essentially a loosely coupled virtual shared memory model for distributed system development.

JavaSpaces technology is a simple, expressive, and powerful tool that eases the burden of creating distributed applications. Processes are loosely coupled; communicating and synchronizing their activities using a persistent object store called a space, rather than through direct communication [1]. CORBA on the other hand allows communication between objects that are written in different programming languages. CORBA is an open, vendor-independent architecture and infrastructure for distributed object technology. CORBA standards define a common message passing mechanism for interchanging data and discovering services. It is widely used today as the basis for many mission-critical software applications. Objects do not talk directly to each other; they always use an object request broker (ORB) to find out information and activating any requested services. CORBA technology uses an Interface Definition Language (IDL) to specify the signatures of the messages and the types of the data objects can send and understand [2]. These technologies introduce a new paradigm for developing distributed applications that are loosely coupled, dynamically and naturally scalable, and fault tolerant.

For evaluating JavaSpaces and CORBA technologies both quantitatively and non-quantitatively, we have chosen a distributed, parallel application to provide data to determine the performance of the two technologies under various load conditions. We have implemented an application that sorts a large array of positive integers by partitioning the sort space into smaller components (smaller arrays) and dropping each such smaller “job” into the shared memory space and then each worker application, which was free, would pick up the job, do the sorting, drop off the result back into the shared memory space. Then the main thread would merge the individually sorted jobs into the proper overall order. On another dimension, we also increase the number of workers, or processors, to measure the performance of the applications developed in JavaSpaces and CORBA under these varying and increasing load conditions. The hardware platforms for both implementations are identical.

The remainder of this paper is organized as follows. Section 2 discusses JavaSpaces technology and GigaSpaces platform, while section 3 discusses CORBA and the ORBacus platform by Iona Technologies. Section 4 discusses the result of our experiments to evaluate the performance of GigaSpaces and ORBacus. In section 5 we discuss our conclusions and future scope of this research.

2 JavaSpaces

2.1 JavaSpaces and the Shared Memory Model – A Historical Perspective

The distributed shared memory model is described by Tam et al in [11]. Hosts in a distributed system visualize the disjoint memory spaces as a common memory space through which they can communicate. The Linda parallel programming environment, described by Gelernter et al in [13, 14], began as a Yale University research project. Communication between processors is handled through a tuple-space where processors post and read messages. The tuple-space concept is basically an abstraction of distributed shared memory, with one important difference: tuple-spaces are associative. Since everyone shares the tuple space, the "look and feel" a developer gets is somewhat similar to that of the shared-memory worldview. On the other hand, the posting and reading of tuples is similar to the sending and receiving of messages in a message-passing system. Unlike shared memory systems and like RPC systems, data must be copied between the individual processes and the tuple space. An advantage of this approach is that processing elements can enter and leave the computation pool at will, without announcing their arrival or departure. Processing elements do not send to or receive from specific nodes. Like hardware shared memory systems, and unlike message passing systems, shared data is accessed directly and anonymously by each process, and processes do not communicate directly with one another.
Tuples are written into the tuple space with an *out* operation, are removed with an *in*, and are read without being removed with an *rd*. For an *in* or *rd*, the tuple accessed in tuple space must match the tuple provided with the command. The number and types of fields must be identical. A value must match an identical value. A variable in either must match a value in the other. A variable will not match a variable. The *in* or *rd* will block until there is a matching tuple in tuple space. Jini/Javaspaces developed by Sun Microsystems was modeled after the Linda concept and is essentially a loosely coupled virtual shared memory model for distributed system development in Java.
JavaSpaces technology is a simple, expressive, and powerful tool that eases the burden of creating distributed applications. Processes are loosely coupled; communicating and synchronizing their activities using a persistent object store called a space, rather than through direct communication [12]. In essence, JavaSpaces is a Java-optimized version of the original C-based tuple-spaces. The major advantage of JavaSpaces over Linda is the Java Virtual Machine (JVM). Linda had many cross-platform obstacles but JavaSpaces runs in a JVM and hence is platform independent [10].

2.2 JavaSpaces - A New Distributed Computing Model

Building distributed applications with conventional network tools usually entails passing messages between processes or invoking methods on remote objects. In JavaSpaces applications, in contrast, processes don't communicate directly, but instead coordinate their activities by exchanging objects through a *space,* or shared memory [3, 9]. JavaSpaces is a specification developed by SUN Microsystems that presents a model of interaction between (mostly) Java applications. Applications seek to exchange information in an asynchronous but transactional-secure manner and can use a space to coordinate the exchange.

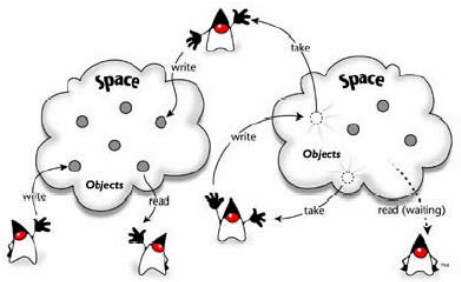

Figure 1: Flow of Objects between JavaSpaces [10]

Figure 1 depicts several applications (the Duke images) interacting with two spaces [10]. Each application can write objects (called Entries) to a space, read objects from a space, and take objects from a space (take means read+delete). In addition, applications may express interest in special entries arriving at a space by registering for notifications. The JavaSpaces API is very simple and elegant, and it provides software developers with a simple and effective tool to solve coordination problems in distributed systems, especially areas like parallel processing and distributed persistence. The developer can design the solution as a flow of objects rather than a traditional request/reply message based scenario. Combined with the fact that JavaSpaces is a Jini service, thus inheriting the dynamic nature of Jini, JavaSpaces is a good model for programming highly dynamic distributed applications.

The JavaSpaces API consists of four main method types:
· Write() - writes an entry to a space.
· Read() - reads an entry from a space.
· Take() - reads an entry and deletes it from a space.
· Notify()- registers interest in entries arriving at a space.
In addition, the API enables JavaSpaces clients (applications) to provide optimization hints to the space implementation (the method snapshot()).

This minimal set of APIs reduces the learning curve of developers and encourages them to adopt the technology quickly. JavaSpaces enable full use of transactions, leveraging the default semantic of Jini Distributed Transactions model. This enables developers to build transactional-secure distributed applications using JavaSpaces as a coordination mechanism. The APIs themselves provide non-blocking versions, where a read() or take() operation may take a maximum timeout to wait before returning to the caller. This is very important for applications that cannot permit themselves to block for a long time or in the case that the space itself is in some kind of a deadlock. JavaSpaces also make extensive use of Jini leases, as it mandates that entries in the space be leased and thus, expire at a certain time unless renewed by a client. This prevents out-of-date entries, and saves the need for manual cleanup administration work [1].

2.3 GigaSpaces

GigaSpaces Technologies has built an industrial-strength JavaSpaces implementation. This implementation is called “the GigaSpaces platform”, or “GigaSpaces” in short. We selected GigaSpaces because it is freely available for evaluation. GigaSpaces is a 100% conforming and a 100% pure Java implementation of the JavaSpaces specification. Moreover, GigaSpaces blends naturally with SUNs' implementation of the Jini API.

The application accesses the space API through a space proxy, which is embedded in the application JVM. This proxy is usually obtained by a lookup in a directory service, like a Jini Lookup service or a JNDI name space. The space proxy communicates with the server-side part of the space, which holds most of the logic and data of the space. The space itself may be an in-memory space or a persistent space. An in-memory space holds all its data in virtual memory. This results in fast access. However, memory spaces are bounded by the amount of virtual memory in the system, and are vulnerable to server crashes. A persistent space uses a DBMS backend to persist its data, while still caching some of the data in memory. Persistent spaces do not lose data as a result of server reboots/crashes and can hold a large amount of data. The server-side part of the space is shared among all applications that refer to the same logical space. This is how different applications can share and exchange information through the space. A GigaSpaces Container is a service that can contain and manage several spaces in one JVM. Spaces in the same container share resources in order to reduce resource consumption. The container is also responsible of registering spaces to directory services in the environment. A GigaSpaces Server can launch several services such as the HTTP Service, Transaction Service, Lookup Service and GigaSpaces Container. This is a single point of configuration for launching several services in a single physical process [4].

3 CORBA

3.1 Background

The early solutions to the challenges facing distributed computing involved message passing using sockets that

pass messages between client and server. This kind of communication required the application programmer to know the Berkeley Socket API. Applications developed were onerous leading to the next generation of message passing protocols such as RPC which hid the low-level socket communication, but the applications tended to be tightly coupled as with socket programming. In other words, the client-side application invoking procedures on the server-side needed to know exactly what services the server was prepared to offer the client. Such distributed systems were less robust and could not withstand partial failures. The literature contains a good description of remote procedure calls. Birrel and Nelson in [15] describe the implementation of RPC and Tay et al in [16] provide a good survey of remote procedure calls. The advent of object-oriented languages such as C++ and Java led to the development of the distributed object computation platforms such as DCOM [17], CORBA [18], and RMI [19]. While these were excellent in that they provided an object-oriented framework for developing distributed systems, they were essentially RPC-oriented, tightly coupled, message passing systems with the ability to marshal objects when the objects are used as parameters in the method calls.

3.2 The CORBA standard

The Common Object Request Broker Architecture (CORBA) is a standard for transparent communication between applications objects [5]. The CORBA specification is developed by Object Management Group (OMG), which is a non-profit industry consortium. It allows a distributed, heterogeneous collection of objects to inter-operate. Part of CORBA standard is the Interface Definition Language (IDL), which is an implementation-independent language for describing the interface of remote objects. CORBA offers greater portability in that it isn't tied to one language, and as such, can integrate with legacy systems, as well as future languages that include support for CORBA.

CORBA applications are composed of objects, individual units of running software that combine functionality and data. There could be many instances of an object of a single type or only one instance. For each object type, we define an interface in OMG IDL. The interface is the syntax part of the contract that the server object offers to the clients that invoke it. Any client that wants to invoke an operation on the object must use this IDL interface to specify the operation it wants to perform and to marshal the arguments that it sends. When the invocation reaches the target object, the same interface definition is used there to unmarshal the arguments so that the object can perform the requested operation with them. The interface definition is then used to marshal the results for their trip back and to unmarshal them when they reach their destination. The IDL interface definition is independent of programming language, but maps to all of the popular programming languages via OMG standards. The separation of interface from implementation, enabled by OMG IDL, is the essence of CORBA - how it enables interoperability, with all of the transparencies we have mentioned. In contrast, the implementation of an object - its running code, and its data - is hidden from the rest of the system (that is, encapsulated) behind a boundary that the client may not cross. Clients access objects only through their advertised interface, invoking only those operations that the object exposes through its IDL interface, with only those parameters (input and output) that are included in the invocation. Figure 2 shows how everything fits together, at least within a single process: Compile the IDL into client stubs and object skeletons.

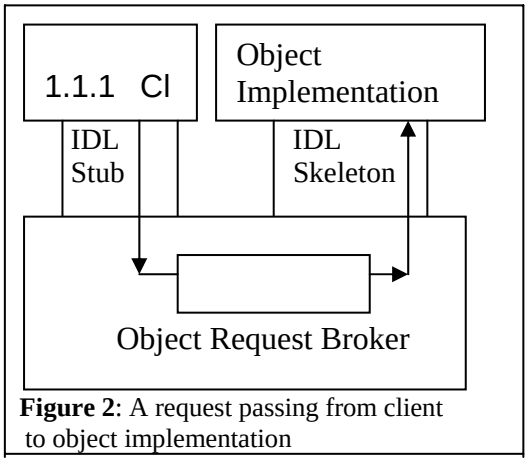

Figure 2: A request passing from client to object implementation

Next, write the object and a client for it. Stubs and skeletons serve as proxies for clients and servers, respectively. Because IDL defines interfaces so strictly, the stub on the client side has no trouble meshing perfectly with the skeleton on the server side, even if the two are compiled into different programming languages, or even running on different ORBs from different vendors. In order to invoke the remote object instance, the client first obtains its object reference using Trader service or naming service. The client knows the type of object it is invoking and the client stub and object skeleton are generated from the same IDL. Although the ORB can tell from the object reference that the target object is remote, the client cannot.

3.3 ORBacus

ORBacus is a mature CORBA product that has been deployed around the world in mission critical systems. ORBacus is 'CORBA 2.5 compliant' and is designed for rapid development, deployment and support in the language of our choice C++ or Java; its small footprint allows it to be easily embedded into memory-constrained applications [6]. We chose ORBacus for evaluation, as it is freely available for evaluation and is an industry grade CORBA product.

4 Results

4.1 Overview

We implemented a distributed, parallel insertion sort application because such an algorithm significantly exercises the CPU computationally. The insertion sort

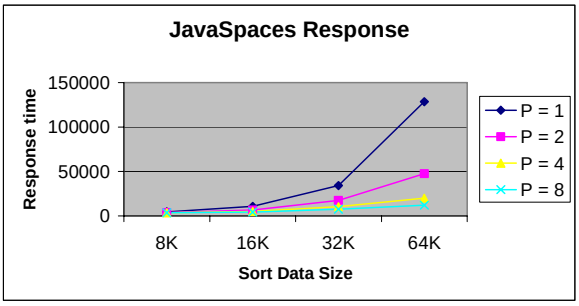

Figure 3: JavaSpaces response with varying processors and varying data size

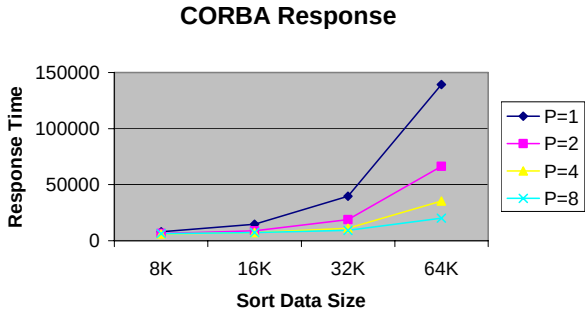

Figure 4: CORBA response with varying processors and varying data size

algorithm has a complexity of $O(n^2)$. This application sorts a very large array of positive integers by partitioning the sort space into smaller components (smaller arrays) and dropping each such smaller "job" into the shared memory space and then each worker application, which was free, picked up the job, do the sorting, drop off the result back into the shared memory space, and then the main thread puts back the individually sorted jobs into the proper overall order. The performance was measured by increasing the number of processors or servers as well as increasing the problem size by increasing the size of the array that needed sorting. Implementing the same application using JavaSpaces and CORBA allowed comparison of performance, ease of development and maintenance, and portability across platforms between two technologies.

JavaSpaces No. of workers				
Input Size	P=1	P=2	P=4	P=8
8K	4636	3726	3451	3573
16K	10744	6701	4898	4465
32K	34223	17529	10459	7488
64K	128508	47488	20003	12056

Table 1: JavaSpaces Response time

CORBA (No. of workers)				
Input Size	P=1	P=2	P=4	P=8
8K	7947	6438	5941	6399
16K	14747	8839	7395	7263
32K	39599	18816	11097	9282
64K	139199	66365	35280	20119

Table 2: CORBA Response time

4.2 Hardware

The hardware for this project consists of a cluster of homogeneous workstations all running RedHat Linux v7.2. The machine are all Intel based PCs consisting of single 500 MHz processors connected by 100 megabit fast Ethernet.

4.3 Software

The software for the project consists of Java™ 2 Runtime environment, Standard Edition version 1.3.1. We used Java language for coding for the entire application to keep variables in performance evaluation to a minimum. We used GigaSpaces3.0 an implementation of JavaSpaces, and ORBACUS 4.1.2, an implementation of CORBA.

4.4 Testing

We ran a series of executions for both the architectures by changing parameters for each run. We used 8K, 16K, 32K and 64K integers, which were randomly generated and used 1, 2, 4 and 8 workers/servers. The data was distributed so as each server has access to same amount of data. The servers do all the work while the client only distributes and collects data. All the executions were run under similar conditions for both the technologies. We ran our measurements when the load on network and servers was at a minimum. Table 1 summarizes the data obtained from the experiments for JavaSpaces.

Figure 3 is a graph of the response time with increasing sort work and number of workers for JavaSpaces implementation. Figure 4 is a graph of the response time with increasing sort work and number of workers for the CORBA implementation. Table 2 summarizes this data in table format.
Speed-up is defined as ratio of time taken to sort the same work using one worker to time taken by using more than one worker.

Figures 5 and 6 are graphs of the speed-up for JavaSpaces and CORBA respectively. Comparing figures 5 and 6, we derive that we have improved speed-up when processing large amount of sort data. We also observe that we have better speed-up in JavaSpaces.

The mean response time graph is shown in Figure 7 where each pair of the mean response time is compared at the 0.05 level, i.e., differences are due to chance only 5% of the time. From Figure 7 we observe that for each data size, CORBA takes significantly longer than JavaSpaces. The difference is the same for all data sizes.

We also observed that when we employed two workers CORBA is significantly higher in response time than JavaSpaces for all but input data size of 32K, where there is no significant difference. The difference is higher in data size 64K. We have similar observations as above when we have four workers. CORBA is significantly higher in response time than JavaSpaces in all data sizes except 32K, where there is no difference. The difference is higher in data size of 64K. For eight workers CORBA is significantly higher in response time for all data sizes. The difference is higher in data sets of 64K.

5 Conclusions

GigaSpaces, the JavaSpaces implementation, consistently outperformed ORBacus, the CORBA implementation, in terms of response time on both the parameters - size of the problem and number of processors deployed to work as workers/servers. Hence we conclude from the observed data that distributed parallel algorithms of master-worker pattern may be able to perform more efficiently when developed using the JavaSpaces platform. CORBA is language neutral and thousands of sites rely on CORBA for enterprise, Internet-based, and other computing. Both CORBA and JavaSpaces architectures provide tremendous benefits in terms of fault-tolerance and scalability. In terms of ease of use and implementation of the two technologies, implementation of JavaSpaces was easier than CORBA.

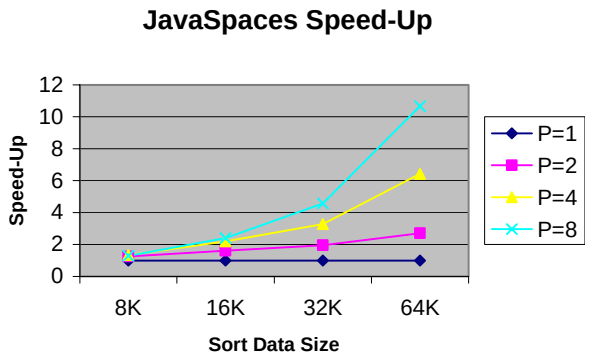

Figure 5: JavaSpaces speed-up

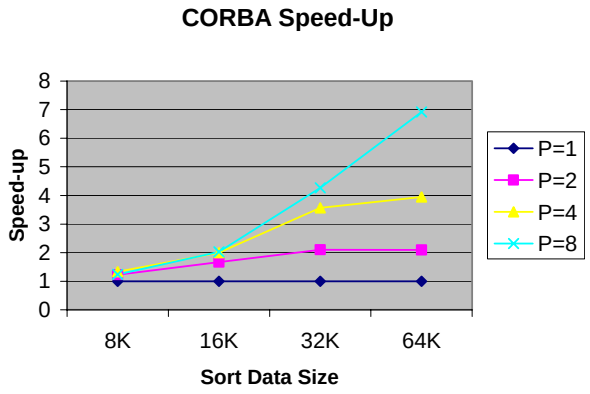

Figure 6: CORBA speed-up

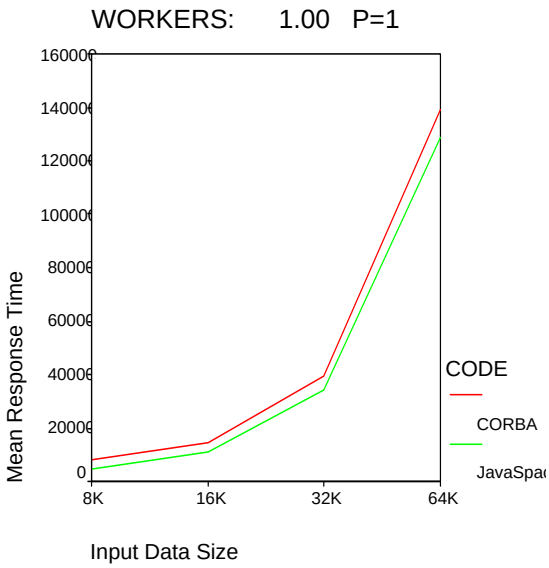

Figure 7: Mean response time for P=1 for JavaSpaces and CORBA

GigaSpaces platform already provides most of the implementation details and from an application programmer's perspective; there are only five commands to learn. We did face some challenges in implementing JavaSpaces due to its increased security considerations that is in-built within the JavaSpaces and its underlying Jini technologies and GigaSpaces platform. JavaSpaces does have the limitation that it can be only implemented on the Java platform supporting Jini architecture. In comparison, implementation of CORBA platform is harder due to much-detailed standards that developers must adhere.

References

[1] Freeman, E., Hupfer, S., Ken Arnold, "JavaSpaces Principles, Patterns, and Practice", Addison Wesley, 1999, pp. 4-16.

[2] http://www.capescience.com/resources/

[3] http://www.artima.com/jini/

[4] http://www.gigaspaces.com/download/GigaSpaces WhitePaper.pdf

[5] http://www.omg.org/technology/documents/formal/

[6] http://www.orbacus.com/support/new_site/pdf/OrbacusWP.pdf

[7] Triola, Mario F., "Essentials of Statistics", Addison Wesley, 1999, pp. 4-16.

[8] JavaSpaces Service Specification http://www.sun.com/software/jini/specs/js1_1.pdf

[9] Teo, Y.M., Ng, Y. K, Onggo, B.S.S., "Conservative Simulation Using Distributed-Shared Memory", Proceedings of the 16th Workshop of Parallel and Distributed Simulation. May 2002.

[10] http://java.sun.com/developer/Books/JavaSpaces/introduction.html

[11] Tam, M., Smith, J., Farber, D., "A Taxonomy-based Comparison of Several Distributed Shared

Memory Systems", *ACM Operat. Syst. Review 24,* July 1990, pp. 40-67.

[12] Eugster, P. T., Felber, P.A., Guerrauoi, R., Kermarrec, A., "The Many Faces of Publish-Subscribe", *ACM Computing Surveys, vol. 35, no. 2,* June 2003, pp. 114-131.

[13] Gelernter, David, "Generative Communication in Linda," *ACM TOPLAS*, 7:1, January 1985.

[14] Carriero, Nicholas, and David Gelernter, "Linda in Context," *CACM*, 32:4, April 1989.

[15] Birrell, A. D., and Nelson, B. J., "Implementing Remote Procedure Calls", *Proceedings of the ACM Symposium on Operating System Principles,* ACM Press, New York, NY, 1983.

[16] Tay, B. H., Ananda, A. L., "A Survey of Remote Procedure Calls", *ACM Operat. Syst. Review 24,* July 1990, pp. 68-79.

[17] Sessions, R., "COM and DCOM: Microsoft's Vision for Distributed Objects", *John Wiley and Sons,* New York, NY, 1997.

[18] OMG, "The Common Object Request Broker: Core Specification", *The Object Management Group,* Needham, MA, 2002.

[19] Sun Microsystems, "Java Remote Method Invocation Specification", *Sun Microsystems*, Santa Clara, CA, 2000.