A Framework for Object-Oriented Metacomputing

Nenad Stankovic

FSJ Inc., Daiwa Naka-Meguro Bldg. 7F, 4-6-1 Naka-Meguro, Meguro-ku, Tokyo 153-0061, Japan
nstankov@comp.mq.edu.au

Abstract. Visper is a network-based, visual software-engineering environment for parallel processing. It is completely implemented in Java and supports the message-passing model. Java offers the basic platform independent services needed to integrate heterogeneous hardware into a seamless computational resource. Easy installation, participation and flexibility are seen as the key properties when using the system. Visper is designed to make use of Java features to implement important services like efficient creation of processes on remote hosts, inter-process communication, security, checkpointing and object migration. We believe the approach taken simplifies the development and testing of parallel programs by enabling a modular, object-oriented technique based on our extensions to the Java API, without modifying the language. Our experimental results show that good speedup is achievable for course grained parallel applications.

1 Introduction

Wide and local area networks represent an important computing resource for parallel processing community. The processing power of such environments is deemed huge. The problem is that they are made up of heterogeneous hardware and software. In this paper we focus on describing a novel metacomputing environment named Visper. The aim of the project is to allow a programmer to make an efficient use of the networked computing resources. Visper is implemented in and for the Java language. Java, due to its platform independence and uniform interface to system services simplifies the implementation of MIMD parallel applications and the system software needed to support them.

A number of tools and libraries have been designed to foster the use of networks to run parallel programs. In the domain of the message-passing model products like the MPI [11] and the PVM [9] transform heterogeneous networks of workstations and supercomputers into a scalable virtual parallel computer. While MPI was focused on message passing aspects, PVM acts as a runtime system. The aim of both is to provide a standard and portable programming interface for several high level languages. They have been adapted to a variety of architectures and operating systems. The problem with both systems is that programmers have to build a different executable for each target architecture or operating system and programs must be started manually. The

insufficient security and dependence on a shared file system limits their use for large or widely distributed applications. For example, while it is possible for a PVM process to run code that is loaded from other network nodes, the virtual machine provides no protection for its hosts against malicious or errant behavior from the downloaded code.

With the advent of Java and the Internet the world of computing has become richer for a new programming technique that is inherently driven by platform independence. This explains why Java should not be recognized merely as yet another programming language but rather as a concept or an environment. Based on that premise we can extend the network OO programming from the procedural one based on RPC/RMI to the class one based on the fact that classes (code) and objects can be sent over a network rather than data or method invocation. Moreover, it is possible to preserve object's state, to initialize it, etc. We can think of objects (threads) that are distributed over networks, rather than just methods that execute on remote hosts. Java provides secure, yet flexible environment to allow efficient parallel computing between nodes in a network. The Java Security model [20] allows control of local file and threads access, network connection, etc., while enabling the implementation to set up its own security model.

Visper is conceived as an integrated metacomputing environment that provides the services to design, develop, test and run parallel programs. It provides an MPI like communication library, and features to spawn and control (lightweight) processes from within the environment. In this paper we will expand upon the mentioned points. Section 2 describes the organization and the main services supported by the environment. Sections 3 and 4 describe Visper in terms of the front-end and backend. Section 5 describes the communications API, while Section 6 provides performance data. Section 7 informs on the related Java based work in metacomputing. A comprehensive description of some older systems can be found in [2]. The paper concludes with Section 8.

2 Visper

Visper is an interactive, object-oriented environment implemented in Java that provides a set of tools for construction, testing and execution of SPMD applications. It is conceived as a tool for research into parallel and distributed Java programming, and as an efficient, reliable and platform independent environment that produces useful results. It can be used by multiple users and run multiple programs concurrently.

Fig. 1 shows the tool organization and main components. Visper consists of a front-end and a backend. The purpose of the front-end is to implement the graphical user interface and to encapsulate as much of the syntax of the model as possible. The backend implements the semantics of the model independent of the front design. The overall system operation can be described in terms of the two components.

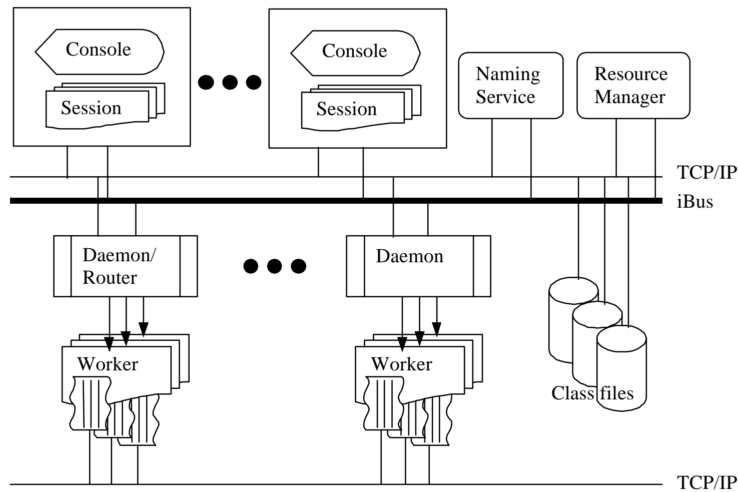

Fig. 1. Overview

Visper is built on top of two communication techniques: a point-to-point (TCP/IP), and an intranet multicast iBus [17]. Each user starts a console that represents the front-end to the system. As described later, the console provides the means to compose, start and analyze parallel programs. Each user can create multiple sessions, where each session can run one parallel program at a time. A session is an ordered collection of processes and represents a virtual parallel computer. It allows control and data collection of the running process. It can dynamically grow or shrink while program is running, since machines can join or leave (i.e. crash). To minimize the start-up cost, a machine that dynamically joins a session, remains part of it until manually removed by the user. Consequently, the loaded (and JIT compiled) bytecodes are preserved across multiple runs of a program, which also compensates for the code generation overhead that occurs during program execution. (The current generations of JIT compilers do not, however, save the native code for future invocations of the same program [12].) For a HotSpot compiler that would also mean more time to perform optimizations, as multiple runs are concatenated.

The backend consists of the services to run parallel programs and generate debugging data. The naming service represents a distributed database of the available resources, active sessions and acts as a port-mapper and detector of faulty machines for the point-to-point mode. Each machine that is part of a Visper environment must run a Visper daemon. The daemons fork one worker-process (i.e. one Java Virtual Machine or JVM) that runs parallel programs per session, and maintain workers state (e.g. active, dead). As shown in **Fig. 1**, workers can run multiple remote threads. Even though they do not share the same address space, we call them threads since they execute as Java threads inside a worker process. By default, the dynamic class loading requires that all class files are located either locally (the CLASSPATH environment variable) or at a predefined URL. As this was deemed inappropriate for a distributed processing environment, to allow more flexible and efficient loading of Java class

files (i.e. remote threads and its components), the user can define multiple loading points, and different access modes (e.g. file://, http://).

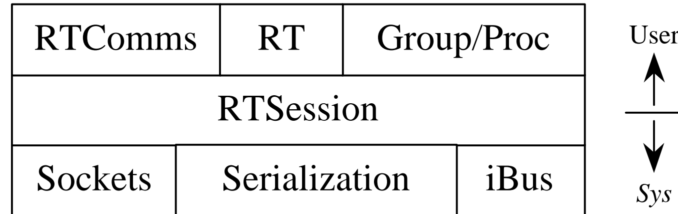

Fig. 2. API

From the programmer's perspective, Visper provides three classes of objects that enable communication and parallel processing (**Fig. 2**). The RTComms is an MPI like library that provides point-to-point, collective, synchronous and asynchronous communication and synchronization primitives. The RTRunnable interface allows any Java class to be turned into a network-runnable and serializable object (i.e. remote thread). The group and process library provides the classes to group remote threads, create and control them from within a program, and to provide communication scope. Groups are the smallest unit of organization in Visper since, at programer's level, a group creation represents allocation of computational resources. Groups can be dynamic and static. A resource manager defines a dynamic group at runtime, while a static group is defined at compile-time and therefore does not scale, but is faster. The session layer contains the information about the current session status (e.g. migrated, restarted) and configuration. It provides methods to migrate, restart and checkpoint remote threads. All these services are built on top of the Java Networking API, the iBus middleware and the Java Serialization mechanism [19].

3 The Front-end

To the programmer, the front-end is the only tangible component of the tool. The purpose of the front-end is to enable user-to-system interaction and to implement the syntax of the programming model or program specification. Its main strength is in the visualization of parallel program composition and execution. Thus, it is the console for the system in which the user constructs a parallel program, then exercises it through an interface.

Fig. 3 shows various components of the console. For example, the front-end provides a visual programming environment with a pallet of model primitives that can be placed on a canvas and which can be interconnected using arcs to create a process communication graph (PCG) [18]. The front-end also generates a structured internal representation of the model, performs syntactic analysis of the graph as the programmer is constructing it, and translates the graph into a Java program. Finally, the front-end must be the user interface to the backend that allows program execution, control and debugging.

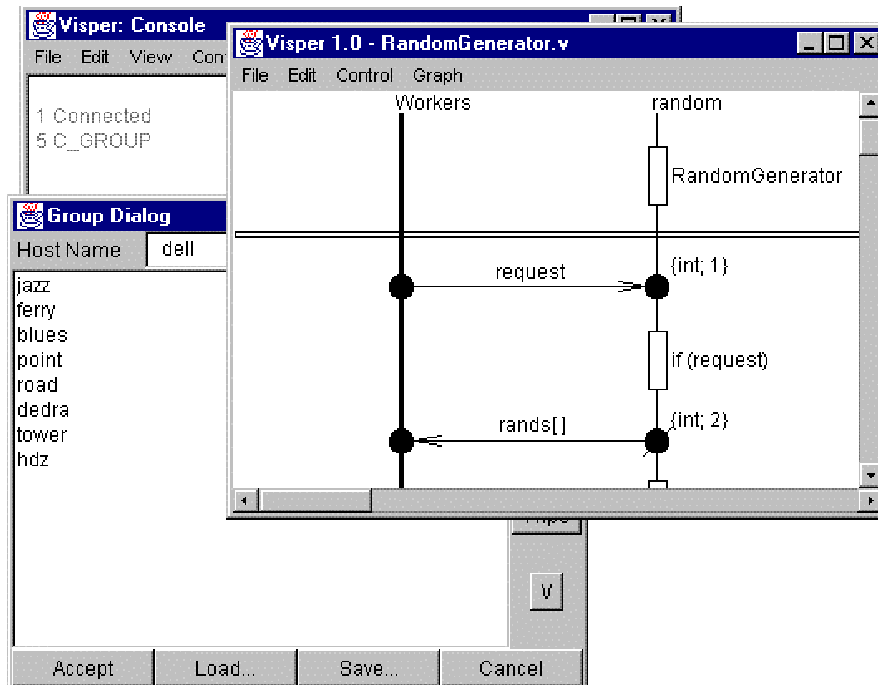

Fig. 3. Front-end Components

To run a program, the user first connects to any daemon, from where the currently available machines are obtained. In the group dialog, the user then creates one or more sessions. Machines can be shared between sessions, as each session has its own set of workers. Then, a system wide path is defined that instructs the workers where to look for the application class files. In the execution dialog, the user can then start a program and pass input arguments within a defined session. These operations do not require special scripts or valid user accounts on the involved machines. When running a program, the user can switch on and off the collection of run-time data and later analyze it in a space-time diagram. To help with debugging, the system also intercepts the exceptions generated by a parallel program and displays it in the console. However, it does not provide a system-wide interface to jdb, at the moment.

4 The Backend

The backend consists of a naming service, resource manager and a set of daemons and workers running on a network or networks of computers. The backend reacts to the directives from the console, then notifies the front-end of the changing program status as the execution process takes place. At a direction of a front-end, the backend will report information regarding the program activity per session, generate space-time diagram data, and perform host allocation.

Visper is built on top of the socket API interface that comes with Java and the iBus intranet middleware supported by our router to enable the across (sub)domain multicast and membership monitoring. Java sockets provide basic reliable point-to-point communications by the TCP/IP protocol, and their scope is not limited by the local intranet or subnet boundaries. The problem with this approach is its limited scalability. Ethernet and Token ring based networks allow efficient hardware multicast facilities. However, they are not reliable and therefore we use iBus that enables a model of spontaneous networking, where applications can join and leave reliable multicast channels dynamically. Along with the multicast capabilities, iBus also provides an interface to receive membership changes and failure detection. When a view change occurs, either due to a new host joining or leaving the channel, the iBus membership protocol detects it and delivers a view change notification to the registered listeners. Based on that, we have implemented a fault tolerant system similar to the Dome [3]. It has been implemented at an application level, due to the JVM specification that prohibits a migrated object to be restarted from the last execution point transparently. The programmer inserts checkpoint calls at an appropriate place in the code, and must reinitialize and skip over already computed operations when a remote thread resumes execution from a snapshot state in a new worker. Either a complete remote thread can be checkpointed, or only a specified list of variables. Both, the checkpointing and evacuation are based on the Java Serialization mechanism and stream compression filters, and require that all objects registered with the fault tolerance mechanism are serializable. Before a thread is restarted, all system wide references to it are automatically updated. That requires that the process ID gets translated to a new host address, on all processes that are members of that process group.

5 Message-Passing Primitives

At the application level, Visper provides a TCP/IP based communication class called RTComms, the style of which follows the MPI standard but in its implementation, is driven by the features of and services provided by Java, and is fully implemented in Java. Further, RTComms hides the addressing and communication mechanisms from the programmer. Issues, such as the establishing of communication channels and handling of network exceptions are performed internally, therefore making the programmer's code smaller than the corresponding socket version. Similar to MPI, the programmer is reasoning in terms of transparent process identifiers, rather than IP names and ports. However, RTComms is an object based communication library taking advantage of the Java Serialization mechanism, and it does not support directly native data types. The benefit of this approach is that the programmer does not have to define the type of the data being sent to the system. The drawback is that the programmer must provide a wrapper class to send native types. However, based on the Reflection API, the RTComms can recognize the content and perform arithmetic, logical or user-defined operations on it (e.g. as required by MPI_Reduce). The library supports blocking and non-blocking point-to-point and collective message-passing primitives, together with a barrier synchronization.

6 Performance

Table 1 summarizes the hardware we have used for our tests. It consisted of 7 Sun boxes running Solaris 2.5 and Sun JDK1.1.6 (no JIT), 2 HP A9000/780/HP-UX B.10.20 with HP-UX Java C.01.15.05, 5 PCs running NT 4.00.1381 and Sun JDK/JRE 1.2. The computers were connected with a 10 Mbps Ethernet.

Table 1. Hardware

Vendor	Architecture	RAM(MB)	CPU(MHz)
Sun	Ultra 2	256	168 * 2
HP	A 9000/780	512	180
Compaq (PC1)	Pentium II	64	200
Micron (PC2,3)	Pentium II	128	200
Micron (PC4)	Pentium II	256	400
Micron(PC5)	Pentium II	384	400 * 2

To test the basic performance of our system we have used Jama [13], a linear algebra package that is intended to serve as the standard matrix class for Java. The test was performed to compare the performance of a standard JVM to that of Visper. To run the test on Visper, we have converted the program into a remote thread. The results are presented in **Table 2**. They do not include the time to load (and compile) the bytecodes. The numbers in brackets represent the values for a second run in the same persistent worker.

Table 2. Magic Square (sec)

	Visper+JIT	Java+JIT	Visper-JIT	Java-JIT
PC5	1.016 (0.328)	1.063	4.968	5.203
PC1	3.174 (1.152)	2.994	9.063	9.444
Sun			12.396	12.152
HP	4.267 (2.679)	4.170	30.676	27.902

The results of the COMMS1 test [7] for messages from 1 to 1000000 bytes are presented in **Table 3**. The JavaMPI results were obtained on the LAM system [16] built by gcc2.7.2 on Sun. The test program for iBus is based on the push model.

Table 3. Communication Cost (ms)

Vendor	Library	Latency	Time/Byte
Sun	LAM	0.518	9.2e-4
Sun	JavaMPI	1.023	9.6e-4
Sun	RTComms	81.55	0.0016
HP	RTComms	51.25	0.0013
PC	RTComms	5.857	0.0012
Sun	iBus	61.84	0.0016
HP	iBus	106.47	0.0015
PC	iBus	57.28	0.0015

The test in **Fig. 4** represents a matrix multiply with 1000 columns and 1000 rows. As our hardware differs greatly in speed, this algorithm was chosen because slow machines do not stall the fast. The *Pc* lines show the effect of JIT compilation. The *NowSpeedup* and *NowTime* were collected on the Sun (no JIT) and HP boxes, while for the PC lines the other was PC1, PC5, PC2, PC4, and PC3.

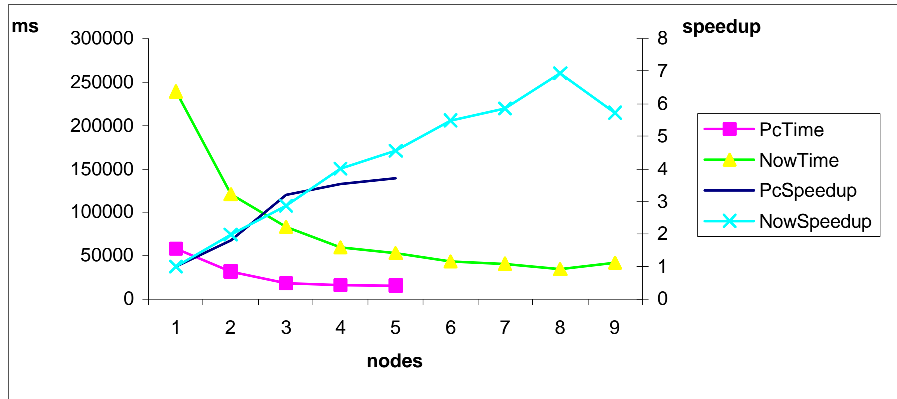

Fig. 4. Performance Results

The speedup lines show that Visper is suitable for course grained applications, while for finer grained problems, the networking overhead becomes dominant, as demonstrated by the *PcSpeedup* line that flattens out very quickly, due to the effect of JIT. A similar LAM program would take 10234 ms on 2 nodes and 8937 ms on 4 nodes.

7 Metacomputing in Java

So far, a typical distributed or parallel application would consist of one user, processes and data that are bound to a particular architecture or file system and resources to which the user has access. The WEB has the potential to be the infrastructure in integrating remote and heterogeneous computer systems into a global computing resource. There is a growing body of work on how to utilize best this new technology. Systems like IceT [10], KnittingFactory [2] and InfoSphere [4] see the Internet as a cooperative or group environment in which multiple users can work together. The IceT programming model is basically an evolution of PVM and does not account for fault tolerance, checkpointing, etc. The Parallel Java Extension [15] uses the Converse [14] interoperability framework, which makes it possible to integrate parallel libraries written in Java with modules in other parallel languages in a single application. Ninflet [21] is a Java system that evolved from Ninf, an RPC-based environment for distributed processing on a wide-area network. The main drive behind these projects was the recognition of the *write once run many* capability of the Java environment. DOGMA [6] is a metacomputing system designed for running parallel programs on networks and supercomputers (IBM SP/2). The goals of

DOGMA are similar to those of Visper (e.g. platform independence, dynamic configuration, and fault tolerance). It provides a communication library, the MPIJ that implements MPI completely in Java. Unlike RTComms, the MPIJ implementation of MPI is based on the MPI C++ bindings as much as possible, and the programming style in DOGMA resembles closely the one found in MPI.

Parallel to this type of research, there have been efforts to enable MPI and PVM to run under the JVM. JavaPVM [22] and MPI Java [5] provide wrappers for the C-calls by using the Java native methods capability. JPVM [8] is a PVM library written in Java. These systems utilize Java mainly for heterogeneity, since they lack the ability to download the application code from the network, fault tolerance, or use any available machine on the network irrespective of valid user accounts. Consequently, they are limited to local networks.

8 Conclusion

We have presented Visper, a novel pure Java based tool for MIMD parallel programming in a LAN/WAN environment. It combines best features provided by Java with the standard practices and techniques pioneered by systems like the MPI and PVM. It allows a remote execution of Java bytecodes, by transforming a network into a virtual parallel computer. The program executes as a group or groups of asynchronous threads that communicate with each other by sending messages. In the shared variable paradigm, processes communicate by writing to and reading from shareable memory locations. This paradigm although simple violates, however, the principles of abstraction and encapsulation, making it difficult to implement large systems reliably [1]. The problems with such implementations include mutual exclusion, condition synchronization and lack of control over communication modes for better performance. On the other hand, remote threads are autonomous, interacting computing elements that encapsulate data and procedure. They can be dynamically created and configured, thus providing flexibility in organizing their activity.

Visper is designed as a multi-layer system, where system services are cleanly decoupled from the message-passing API. It has been implemented as a collection of Java classes without any modifications to the language and therefore can be executed on any standard Java virtual machine. It provides a secure, object-oriented, peer-to-peer message-passing environment in which programmers can compose, test and run parallel programs within a persistent session. The RTComms communication class follows the primitives of the MPI standard. Visper supports synchronous and asynchronous modes of execution and primitives to control processes and groups of processes as if they were real parallel computers.

As a future enhancement to the system we can mention better strategy for dynamic resource allocation. The current resource management is rather naive, since it does not take any relevant parameters into account (e.g. computational power, network bandwidth, load, or task size). On the other hand, Java does not provide an interface to the operating system that would allow access to system's statistics, so a solution like the one found in DOME might be worth looking at.

References

1. Agha, G. A., and Jamali, N. Concurrent programming for Distributed Artificial Intelligence. To appear in: ed. Weiss, G. Distributed Artificial Intelligence, MIT Press, 1998.
2. Baratloo, A., Karaul, M., and Kedem, Z. M. KnittingFactory: An Infrastructure for Distributed Web Applications. Technical Report TR 1997-748, Department of Computer Science, Courant Institute of Mathematical Sciences, New York University. November 1997.
3. Beguelin, A., Arabe, J. N. C., Lowekamp, B., Seligman, E., Starkey, M., and Stephan, P. *Dome User's Guide Version 1.0*. School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May 20, 1996.
4. Chandy, K. M, Rifkin, A, Sivilotti, P. A., Mandelson, and J., Richardson, M. A World Wide Distributed System Using Java and the Internet, http://www.infospheres.caltech.edu.
5. Chang, Y-J., and Carpenter, B. MPI Java Wrapper Download Page, March 27, 1997. http://www.npac.syr.edu/users/yjchang/javaMPI.
6. DOGMA: Distributed Object Group Metacomputing Architecure. September, 1998. http://zodiac.cs.byu.edu/DOGMA.
7. Dongarra, J. J. et al. The 1994 TOP500 Report. http://www.top500.org/.
8. Ferrari, A. JPVM. http://www.cs.virginia.edu/~ajf2j/jpvm.html.
9. Geist, G. A., Beguelin, A., Dongarra, J. J, Jiang, W., Manchek, R., and Sunderam, V. PVM 3 User's Guide and Reference Manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, 1993.
10. Gray, P., and Sunderam, V. The IceT Project: An Environment for Cooperative Distributed Computing. http://www.mathcs.emory.edu/~gray/IceT.ps.
11. Gropp, W., Lusk, E., and Skjellum, A. Using MPI, Portable Parallel Programming with the Message-Passing Interface. The MIT Press, 1994.
12. Hsieh, C-H. A., Conte, M. T., Johnson, T. L., Gyllenhaal, J. C., and Hwu, W-M. W. A Study of the Cache and Branche Performance Issues with Running Java and Current Hardware Platforms. *Proceedings of IEEE CompCon'97*, San Jose, CA, 1997, p.211-216.
13. Jama: A Java Matrix Package. http://math.nist.gov/javanumerics/jama.
14. Kalé, L. V., Bhandarkar, M., Jagathesan, N., Krishnan, S. and Yelon, J. Converse: An Interoperable Framework for Parallel Programming. Proceedings of the 10th International Parallel Processing Symposium, Honolulu, Hawaii, April 1996, pp.212-217.
15. Kalé, L. V., Cox, A. L. and Wilmarth, T. Design and Implementation of Parallel Java with Global Object Space. Proceedings of Conference on Distributed Processing Technology and Applications, Las Vegas, Nevada, 1997.
16. Ohio LAM 6.1. MPI Primer / Developing with LAM, 1996. http://www.mpi.nd.edu/lam.
17. SoftWired AG. Programmer's Manual. Version 0.5. August, 20, 1998. http://www.softwired.ch/ibus.htm.
18. Stankovic, N., Zhang, K. Graphical Composition and Visualization of Message-Passing Programs. SoftVis'97, December, 1997, Flinders University, Adelaide, South Australia, pp.35-40.
19. Sun Microsystems, Inc. Java Object Serialization Specification. Revision 1.4, July 3, 1997, http://java.sun.com.
20. Sun Microsystems, Inc. Java Security Architecture (JDK 1.2). Revision 0.7, October 1, 1997, http://java.sun.com.
21. Takagi, H. et al. Ninflet: a Migratable Parallel Objects Framework using Java. ACM Workshop on Java for High-Performance Network Computing, 1998, pp.151-159
22. Thurman, D. JavaPVM. http://homer.isye.gatech.edu/chmsr/JavaPVM.