Message-Passing in Java

Nenad Stankovic
FSJ Inc., Daiwa Naka-Meguro Bldg. 5-8F, 4-6-1 Naka-Meguro
Meguro-ku, Tokyo 153-0061, Japan
nstankov@comp.mq.edu.au

Abstract

We present here our work aimed at developing a network based visual software engineering environment for parallel processing by the name of Visper. 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 key properties when using the system. We believe the approach taken simplifies the development and testing of parallel programs by enabling modular, object oriented technique based on our extensions to the Java API.

Keywords: MPI, PVM, message passing, performance, Java

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 metcomputing 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 MPI [11] and 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. Java model offers preemptive multithreading, where thread methods offer a set of synchronization primitives based on monitor and conditional paradigm by C.A.R. Hoare. Java threads, however, do not have built-in point-to-point communication primitives needed for message-passing parallel programming. Java binaries are shipped across the network and executed on client machines. Security is therefore a critical issue and strongly enforced in Java. Java interpreter that operates on virtual memory executes the binaries at the client side. Hence, the model is safe since the sent bytecodes are verified at the client that prevents corrupted or evil classes from causing problems [20]. Based on these 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 and objects can be sent over a network rather than the data or method invocation. Moreover, it is possible to preserve object's state, to initialize it, etc. We can think of classes (threads) that are distributed over networks, rather than just methods that execute on remote hosts.
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. Sections 5 and 6 describe the API, while Sections 7 and 8 provide a simple code example and performance data. Section 9 informs on the related Java based work in metacomputing. A description on some older systems can be found in [2]. The paper concludes with Section 10.

2. Visper

Visper is an interactive, object-oriented environment implemented in Java that provides a set of tools for the 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. Figure 1 shows the tool organization and main components. Visper consists of a front-end (i.e. console) and a backend (i.e. system services). 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.

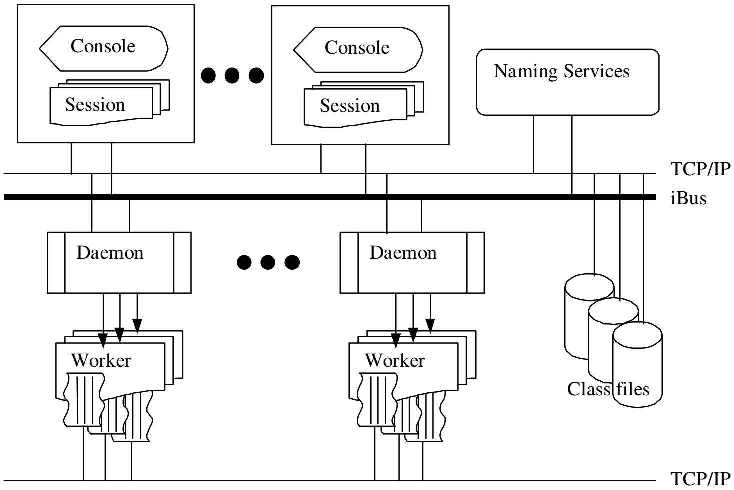

Figure 1 Visper Overview

Visper is built on top of two communication techniques: a reliable point-to-point (TCP/IP), and an intranet multicast (iBus [16]). Both techniques can be used simultaneously, with the iBus being optional. Each user starts a console that represents the front-end to the system. As described later, the console provides the means to compose, analyze and start parallel programs. Each user can create multiple sessions, where each session runs only one parallel program at a time. A session is an ordered collection of the available machines and represents a virtual parallel computer. It allows control and data collection of the running process. It can dynamically grow or shrink while a program is running, since machines can join or leave (i.e. crash). A machine that dynamically joins a session, remains part of it until manually removed by the user.
The backend consists of the services to run parallel programs and generate debugging data for communication primitives. The naming services represent a centralized database of the available resources, active sessions and acts as a port-mapper and detector of faulty machines for point-to-point mode. Each machine that is part of a Visper environment must run a Visper daemon. Daemons fork one worker process that runs parallel programs per session, and maintain workers' state (e.g. active, dead). To minimize the start-up cost, each session maintains its set of workers as long as it is active. As shown in Figure 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. To allow more flexible and efficient loading of the Java class files (i.e. remote threads and its components), the user can define multiple loading points, and different access modes.

<table>
<tr><td>VComms</td><td>RT</td><td colspan="2">Group/Proc</td></tr>
<tr><td colspan="4">Session</td></tr>
<tr><td colspan="2">Sockets</td><td colspan="2">iBus</td></tr>
</table>

Figure 2 API

From the programmers' perspective, Visper provides three classes of objects that enable communication and parallel processing (Figure 2). The VComms is an MPI like library that provides point-to-point, collective, synchronous and asynchronous communication and synchronization primitives. The VRunnable interface allows any Java class to be turned into a network-runnable object (i.e. remote thread). The group and process library provides classes to group remote threads, create and control them from within a program, and to provide the scope for communications. The session layer contains the information about the current session status and configuration. All these services are built on top of the Java network API and iBus middleware.

3. The Console

To the programmer, the console is the only tangible component of the tool. The purpose of the console 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 a console for the system in which the user constructs a parallel program, then exercises it through an interface.
Figure 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) [17,18]. 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.

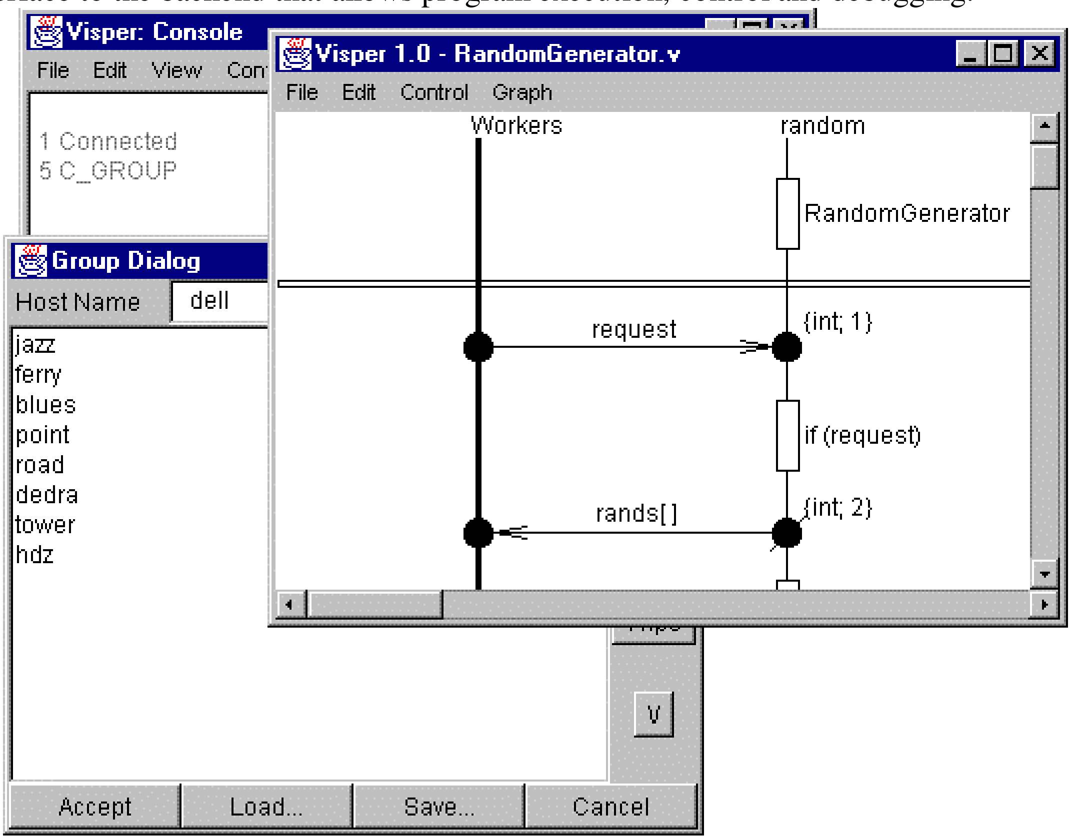

Figure 3 Front-end Components

To run a program, the user first connects to a naming service, 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 yet provide a system wide interface to jdb.

4. The Backend

The backend consists of a naming services and 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 the front-end, the backend will report information regarding the program activity per session, generate space-time diagram data, and host allocation.
Visper is built on top of the socket API interface that comes with Java and the iBus intranet middleware. 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, in Java 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 a simple 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.

4.1 Daemon

A daemon is a standalone Java process that enables its host machine for processing in Visper. Upon startup, each daemon registers with the naming services, and after that it is ready to accept requests from a console. The naming services will ping from time to time all the registered daemons to check their status. At the moment, the daemon also serves as a resource manager by implementing a Round-Robin algorithm. When a user creates a session, the console broadcasts the group of machines into the environment, and the affected daemons create workers. When a resource manager adds a new machine to a session, it informs the parent console about the change. Daemons use the Process class in Java to control and communicate to the workers. To simplify the design, all the communication from the console to the workers is directed through the parent daemon.

4.2 Worker

A worker is a standalone Java process that executes remote threads. From the programer's perspective, a worker represents a virtual processor with memory. They are persistent, meaning that as long as the session they belong to is active, they remain active, too. Therefore, they are capable of storing the bytecode locally across multiple executions. Workers load bytecodes by a customized class loader. There is only one loader per worker. When testing a program, the user can refresh the stored classes without restarting the worker. A worker can execute multiple remote threads at a time, if those threads belong to different groups. If not, they are executed as a FIFO. Workers also provide a default communication channel for the parallel programs, which eliminates the need for the initial synchronization among a group of running remote threads.

5. Message-Passing Primitives

At the application level, Visper provides a TCP based communication class called VComms, 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, the VComms 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 the IP names and ports. However, VComms is an object based communication library taking advantage of the Java Serialization mechanism [19], 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. The VComms supports blocking and non-blocking point-to-point and collective message-passing primitives in a raw and trace mode. Messages can be sent within a group or among different groups. The synchronization capabilities are represented by the barrier method. There are two variants, one that waits for all processes in a group and the other that releases the block after a specified number of processes have issued the call.

6. Remote Processes and Groups

A parallel program consists of many processes running on remote computers. Visper, provides two modes of execution: the MPI mode and the dynamic RT. In the MPI mode the system will execute a requested remote thread by sending it to all the workers in the session. In the RT mode the system will send the request only to the first worker in the session (the one with ID 0). That remote thread acts like a bootstrap. To populate other machines in a session, the API provides two classes: VRemoteProcess and VRemoteGroup. The first one creates a new remote thread within a given group on a remote or local worker. The latter creates a group of threads. Instances of these classes serve as peers to the remote threads. We can use them to check a thread or group status, start, stop or synchronize (i.e. group join), invoke methods via the Java Reflection API, etc.

The following example shows how a remote group of process is created:

```
VGroup vgrp = new VGroup(GroupName,ParentGroup,4);
VRemoteGroup rgrp = new VRemoteGroup(RemoteThreadClass,vgrp);
```

First, the parent thread creates a new group with 4 processes. Then, a remote group of threads is forked off within that group.

7. Example Program

Here, we present a simple example program. It was written for the MPI mode of execution, since it does not use the VRemoteProcess or VRemoteGroup primitives. It is important to notice that when writing a program, these modes are not relevant. They are simply primitives provided by the console to simplify the development.

```
import visper.*
public class test implements VRunnable {
        public void Run(String[] args, ThreadGroup tg,VSession vs)
        {
                VComms comms = new VComms(vs);
                VWorld world = new VWorld(vs);
                if (vs.HostID(world) == 0) {
                        VDataSend ds = new VDataSend(10,1,world);
                        comms.Send(ds,new Integer(5));
                }
                else if (vs.HostID(world) == 1) {
                        VDataRecv dr = new VDataRecv(10,0,world);
                        Integer msg = (Integer)comms.Recv(dr);
                        comms.Console(msg.toString());
                }
        }
}
```

The program consists of two processes, one that sends and the other receives a message. It requires only the default group called World. The receiving process sends a message to the console that displays the number being sent.

8. Performance

We have measured the basic performance of Visper using: 7 UltraSparc/Solaris2.5 with 256MB RAM, two Pentium200/NT 4.00.1381 with 64 MB RAM and two HP A9000-780/HP-UX B.10.20 with 512MB RAM connected with a 10 Mbps Ethernet. On the HP boxes we have used HP-UX Java C.01.15.05, on the Sun machines Sun JDK1.1.6, and on the PCs Sun JDK1.2. Since the Solaris version does not support just-in-time compilation (JIT), the just-in-time compiler was disabled for the test, except were noted. We have also used MPICH 1.1.1 [14] and PVM 3.4 [15], built with Sun's C/C++ 4.2 compilers.

8.1 Speedup

The speedup test represents a simple Jacobi iteration with 2000 columns and 300 rows with a maximum of 100 iterations. The graph in Figure 4 displays an average over 8 runs for each different number of nodes. For a 500 by 500 matrix, the best speedup value was only 2.7 and it peeked at 7 nodes.

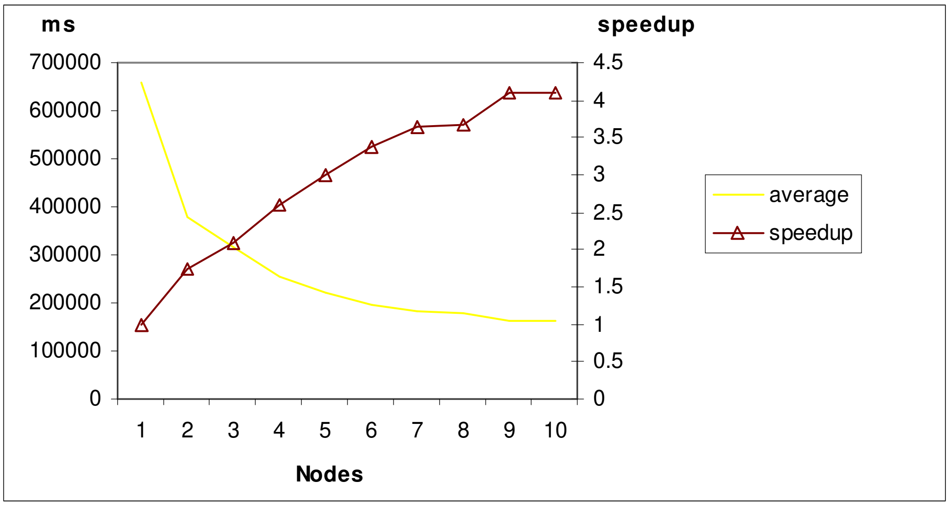

Figure 4 Speedup Results

It is interesting to compare these speedup values to one obtained by the MPICH. For the first example, the speedup was 2.11 on the 7 UltraSparc machines, and for the second example 2.27. However, the computation times were 15.162 seconds and 5.094 seconds respectively for 1 node, and 7.183 and 2.242 seconds for 7 nodes.

8.2 Point-to-Point Communication

Generally speaking, to send a message, the following steps are required:

- Address resolution,
- Data marshaling, and
- Data Transfer.

With message-passing systems, the address resolution involves conversion of a logical process identifier with respect to a communication group into the actual IP address and port. The data marshaling involves conversion of the data from the local host format into the network (i.e. architecture independent) format, and back. The data transfer takes place over a network, the characteristics of which usually have large impact on the overall performance of parallel programs. These three steps were included in the time cost benchmark, below.

Here we present the test results for the COMMS1 [7] test performed on the mentioned libraries. The purpose of this benchmark is to measure the basic communication properties of a message-passing environment: latency and unidirectional bandwidth. A master processor sends a message to a slave processor, and the slave immediately returns it to the master. Messages are of variable length, and it is assumed that the amount of time to send a message from the master to the slave is equal to the time required for the return. The tests were performed for messages of 1, 100, 1000, 10000, 100000, and 1000000 bytes in length. The latency and bandwidth values were calculated by a linear function via a least squares linear regression.

Table 1 Native Systems (ms)

Bytes Sent	PVM	jPVM	MPICH
1	0.563	0.95	0.603
100	0.768	1.1	0.709
1000	2.363	2.65	2.203
10000	10.41	10.8	10.21
100000	95.84	97.6	103.1
1000000	966.3	985.05	1026

As a reference, in Table 1 we first present the COMMS1 results taken for the 3 native systems: PVM, MPICH and jPVM [22], collected on the Sun workstations. As expected, the PVM and MPICH times are very similar. The jPVM times are marginally inferior to the PVM times, due to two causes of overhead: an additional call to switch between Java and C, and cost of the Java to native data types conversion.

Table 2 VComms (ms)

Bytes Sent	HP+JIT	PC+JIT	Sun
1	12	8	63
100	13	8	60
1000	13	8	60
10000	106	18	121
100000	237	118	236
1000000	1089	1192	1230

Table 2 shows the VComms times for the three platforms. The tests were performed between 2 computers of the same type. The HP and PC results were obtained by using JIT compiler. It is interesting to notice that although VComms was built directly on top of the Java Object Serialization mechanism [19], so the impact of the bytecode interpreter is minimal, it is two orders of magnitude behind the jPVM for short messages. As messages were getting larger, the impact of Java on the overall performance has decreased. The cross-architecture results were dominated by the slower node. For example, a Sun-to-PC test yielded results similar to the Sun column in Table 2, while a PC-to-HP test produced results slightly better than the HP column.

Table 3 Communication Cost (ms)

	jPVM	MPICH	Sun	HP+JIT	PC+JIT
Latency	0.784	0.905	81.554	51.249	5.782
Time/Byte	0.00098	0.00097	0.0016	0.0013	0.0012

Table 3 shows the communication costs for jPVM, MPICH on Sun and VComms on the three target architectures. The VComms values reflect the high cost of networking in Java for small messages. jPVM and MPICH have outperformed Java on Sun by two orders of magnitude, but it is interesting to notice that JIT has significantly reduced PC latency, while on HP the impact was only marginal. Good time-per-byte values can be explained by the fact that as messages were getting larger, the differences in times were getting smaller, for all the three Java Virtual Machines (JVM).

Table 4 Bandwidth (MB/s)

Bytes Sent	jPVM	MPICH	Sun	HP+JIT	PC+JIT
1	0.001	0.002	1.6e-5	8.3e-5	1.3e-4
100	0.091	0.145	0.002	0.007	0.013
1000	0.377	0.465	0.017	0.077	0.125
10000	0.926	0.986	0.083	0.094	0.556
100000	1.024	1.018	0.424	0.422	0.847
1000000	1.015	1.036	0.813	0.918	0.839

As a result of rather poor latencies, the effective bandwidth is reduced for small messages in VComms compared to jPVM and MPICH by one to two orders of magnitude (Table 4). Again, the HP results are rather surprising regarding the fact that all tests were performed with JIT compilation. On the other hand, our implementation and tests have proved that a pure Java message-passing library, though inferior in performance to the systems based on native code, is a viable option.

9. 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. Netsolve [3] provides a Java based system for exporting computational resources and libraries of preconfigured modules to solve computationally intensive problems. The user communicates with the system via a GUI that provides a set of available modules, an input window to specify the input data and an output window to view the results. Upon receiving the input data and the problem name, the Netsolve system automatically allocates the resources to carry out the computation and sends back the result.

Other 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 project has also done some work at providing a PVM like environment for parallel processing. The Parallel Java Extension [13] uses the Converse [12] interoperability framework, which makes it possible to integrate parallel libraries written in Java with modules in other parallel languages in a single application. 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 VComms, 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 popular message-passing libraries like the MPI and PVM to run under the JVM. jPVM [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, but misses some important PVM features, e.g.: dynamic process group, group broadcast and barrier synchronization. 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.

10. Conclusion

Visper is a novel, pure Java based tool for MIMD parallel programming. It combines best features provided by Java with the standard practices and techniques pioneered by the systems like MPI and PVM. It allows a remote execution of Java bytecodes, by transforming a network of machines 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 two-layer system, where system services are cleanly decoupled from the message-passing API. 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 communication primitives represented by the VComms class follow the primitives of the MPI standard. They provide synchronous and asynchronous modes of communication between processes. The programmer can choose between point-to-point and collective communications. Visper supports synchronous and asynchronous modes of execution and primitives to control processes and groups of processes as if they were real parallel computers. The performed tests have showed that Java is a viable option for parallel processing, especially when enhanced at runtime by a JIT compiler.

As future enhancements to the system we can mention better strategy for dynamic resource allocation, object migration and fault tolerance. 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). The fault tolerance is currently limited to the system level. Faulty nodes are detected and removed from the system. However, the object migration, and evacuation in the event of failure are not implemented yet. Due to the JVM specification, a migrated object cannot be restarted from the last execution point transparently. For example, Ninflet [21] requires explicit checkpointing by the programmer. It is the programmer's responsibility to skip over the executed operations when a Ninflet execution resumes from a snapshot state. Due to the partial implementation of the above features, the current implementation of Visper is more suitable for LAN environments.

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., et al. 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] Casanova, H., and Dongarra, J. Netsolve: A Network Solver for Solving Computational Science Problems. Technical Report CS-95-313, University of Tennessee, November 1995.

[4] Chandy, K. M, et al. 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., Meuer, H-W., and Strohmaier, E. The 1994 TOP500 Report. http://www.top500.org/.

[8] Ferrari, A. JPVM. http://www.cs.virginia.edu/~ajf2j/jpvm.html.

[9] Geist, G. A., et al. 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] 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.

[13] 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.

[14] MPICH-A Portable Implementation of MPI. http://www-unix.mcs.anl.gov/mpi/mpich.

[15] PVM: Parallel Virtual Machine. http://www.epm.ornl.gov/pvm/pvm_home.html.

[16] SoftWired AG. Programmer's Manual. Version 0.5. August, 20, 1998. http://www.softwired.ch/ibus.htm.

[17] Stankovic, N., and Zhang, K. Graphical Composition and Visualization of Message-Passing Programs. SoftVis'97, December, 1997, Flinders University, Adelaide, South Australia, pp.35-40.

[18] Stankovic, N., and Zhang, K. A Parallel Programming Environment for Networks. ISHPC'99, Kyoto, Japan, May 26-29, 1999.

[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.

NENAD STANKOVIC received a B.S. in electrical engineering from the University of Zagreb, Croatia in 1983 and a M.S. degree in computer science from Macquarie University, Australia in 1997. Currently he is a member of research staff at Macquarie University and also working towards his Ph.D. in computer science. Prior to coming to Macquarie University he spent 13 years working on different projects in computing. His research interest is in the areas of parallel programming environment, parallel and distributed system and architecture design and artificial intelligence.