Title	A grid middleware for distributed Java computing with MPI binding and process migration supports
Author(s)	Chen, L; Wang, CL; Lau, FCM
Citation	Journal Of Computer Science And Technology, 2003, v. 18 n. 4, p. 505-514
Issued Date	2003
URL	http://hdl.handle.net/10722/88960
Rights	Creative Commons: Attribution 3.0 Hong Kong License

A Grid Middleware for Distributed Java Computing with MPI Binding and Process Migration Supports

Lin Chen, Cho-Li Wang, Francis C.M. Lau, and Ricky K. K. Ma
Department of Computer Science & Information Systems
The University of Hong Kong

Abstract

"Grid" computing has emerged as an important new research field. With years of efforts, Grid researchers have successfully developed Grid technologies including security solutions, resource management protocols, information query protocols, and data management services. However, as the ultimate goal of Grid Computing is to design an infrastructure which supports dynamic, cross-organizational resource sharing, there is a need of solutions for efficient and transparent task re-scheduling in the Grid.

In this research, we propose a new Grid middleware, named G-JavaMPI. This middleware adds the parallel computing capability of Java on the Grid with the support of a Grid-enabled message passing interface (MPI) for inter-process communication between Java processes executed at different Grid points. A special feature of the proposed G-JavaMPI is the support of Java process migration with post-migration message redirection. With these supports, we can migrate executing Java process from site to site for continuous computation, if some site is scheduled to turn down for system reconfiguration. The proposed middleware is useful for Grid computing as most applications in Grid require long processing time. Restart the execution of those applications will certainly lessen the productivity. Moreover, the proposed G-JavaMPI middleware is very portable as it requires no modification of underlying OS, Java virtual machine, and MPI package. Preliminary performance tests have been conducted. The proposed mechanisms have shown good migration efficiency in a simulated Grid environment.

1. Introduction

"Grid" computing [2], distinguished from conventional distributed computing, is featured by its focus on large-scale resource sharing, innovative applications, and, in some cases, high-performance orientation. Previous Grid research mainly address the issues related to authentication, authorization, resource access, and resource discovery for enabling resource sharing among dynamic, multi-institutional organizations connected across a WAN. As these organizations vary tremendously in their purpose, scope, size, duration, and structure; particularly, the resource configurations of organizations have the potential to change dramatically. To achieve continuous computation and make better utilization of the available computing resources among these organizations, runtime process re-scheduling is required. Process migration is an attractive feature for re-scheduling tasks in Grid environments. With the support of process migration, various runtime load balancing schemes can be employed for improving the execution efficiency of coarse-grained Grid applications. Process migration can also help those long-running applications by relocating them at suitable times to prevent interruption due to system activities or the execution of other applications. It also can help relocate processes closer to the Grid point with data that they need to access.

This research describes a new Grid middleware called G-JavaMPI for distributed Java computing using message passing interface (MPI) on Grid. This middleware supports MPI-style inter-process communication between multiple Java processes that are running on different Grid sites, through a special binding between Java applications and the Grid-ready, MPICH-G2 libraries [3]. This middleware also supports security-enhanced Java process

migration functions, which allows transparent migration of Java programs in the Grid environment.

G-JavaMPI uses the Java built-in debugging interface (JVMDI) to capture execution states, and uses object serialization mechanism [16] to store data in portable form. As JVMDI [15] is a standard interface, this approach is potentially more portable and suitable for the heterogeneous Grid environments, where different JVMs are used. In a Grid environment, any attempt of remote process creation must first pass the authentication process with their appropriate credentials to the destination site before the process can be actually restarted. In G-JavaMPI, the authentication processes are based on the security services provided by GSI [8, 19] (Globus Security Infrastructure) implemented in Globus toolkit [12].

The implementation of the native MPI library that we use in this research is MPICH-G2 [3]. MPICH-G2 is a Grid-enabled implementation of the MPI v1.1 standard. MPICH-G2 uses services provided by the Globus Toolkit to coordinate work and enable authenticated inter-site communication in the Grid environment. MPICH-G2 can also automatically convert data in messages sent between machines of different architectures and supports multi-protocol communication by automatically selecting TCP for inter-machine messaging and vendor-supplied MPI (where available) for intra-machine messaging. To support transparent Java process migration, the communication channels between distributed Java processes must be re-constructed before the migrated process can be restarted. In G-JavaMPI we implement a restorable MPI communication layer to reconstruct the communication channels. It provides a unified communication abstraction for post-migration interprocess communication. Therefore, other processes don't need to care about the physical location of the migrated Java processes.

All the above stated mechanisms require no modifications of underlying OS, Java Virtual Machine, and MPI, which make the proposed system portable to run on various platforms. With this support, the parallel Java processes can be migrated freely and transparently between sites in the Grid and continue their execution and inter-process communication using MPI to achieve dynamic load balancing.

The rest of the paper is organized as follows. In section 2, we show an overview of the proposed middleware. Section 3 discusses our mechanism on saving and restoring execution state of programs. Section 4 describes the authenticated restorable MPI communication layer. Performance results and evaluation are given in section 5. The related works are presented in section 6. Conclusion is given in section 7.

2. System Architecture

Fig. 1. shows the layered design of the G-JavaMPI middleware. The middleware consists of several layers, including the *Java-MPI layer*, the *Migration layer*, the *Authenticated & Restorable MPI Communication layer*, *Load Balancing and Controlling Modules*.

Java-MPI API layer: The Java-MPI API layer implements MPI calling interfaces to link Java programs to the native MPI library. Thus, efficient message passing among distributed Java processes can be achieved. We opted for a modular, client-server design of a message redirection mechanism for migrated Java processes. The Java-MPI API layer acts as a client which sends MPI-related messaging requests to the MPI daemon (a server) in the same node in the Restorable MPI communication layer (will be discussed later). The MPI daemon is responsible for delivering messages on behalf of the Java process. This client-server message redirection model makes it possible to avoid conflicts on the use of system resources (e.g. conflicts on the system signals) between the native MPI library and the JVM.

Authenticated & Restorable MPI Communication layer: The authenticated restorable MPI communication layer (we call it MPI communication layer for short in the following content) is implemented as a daemon. For short, we call it *MPI daemon*. The MPI daemons themselves use the authenticated communication services from the underlying MPICH-G2. And then the

MPI daemons provide MPI communication services only to those Java-MPI applications or processes which have a set of authenticated credentials. Therefore, this chain of authenticated communication channels assures the communication security. The set of credentials includes the credential (usually proxy or delegated credential) for the user that the process is on behalf of, and the Java-MPI job credential which identifies this unique application in the global range. The representation of credentials and authentication method adhere to those in the GSI (Grid Security Infrastructure) which the Globus toolkit [12] is also based on. Therefore, it enables the processes of a Java-MPI application to do the authenticated inter-process communication with each other. In addition, this communication services are restorable. Therefore, the communication channels can be re-constructed automatically after migration. This allows Java processes to communicate with each other after migration as if no migration has occurred.

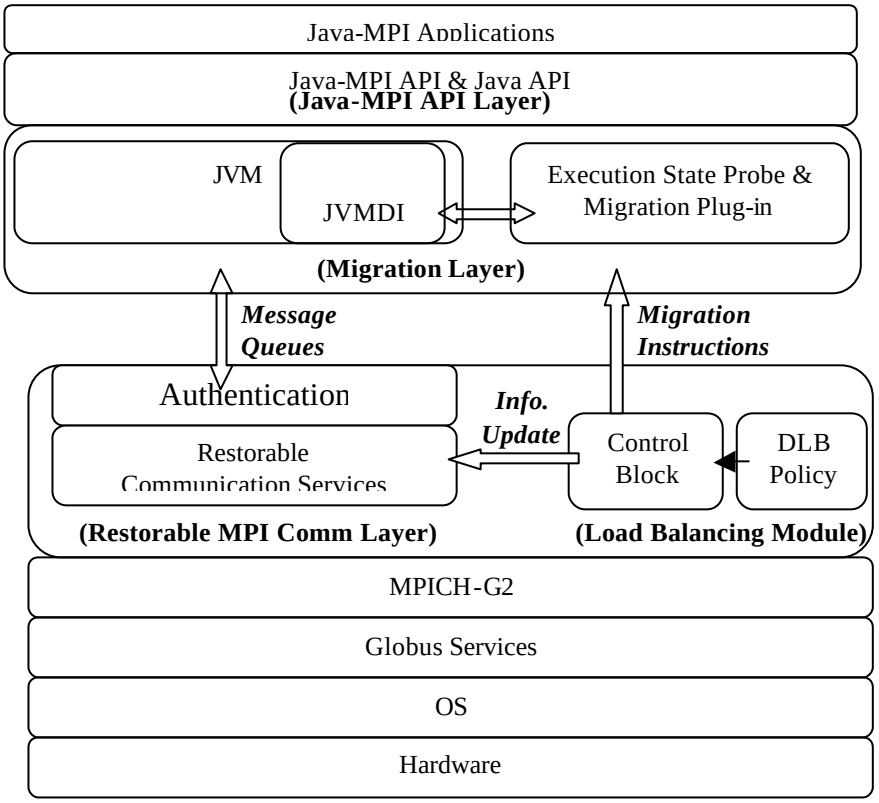

Fig. 1. The layered design of G-JavaMPI

Migration layer: The Migration layer performs two main tasks: (1) to capture and save the execution state of the migrating process in the source node, and to restore the execution state of the migrated process in the destination node; (2) to cooperate with the **Authenticated Restorable MPI Communication layer** to reconstruct the communication channels of the parallel application. In G-JavaMPI, we use the Java built-in interface, JVMDI, to capture Java process execution states. To enable this feature, we need only to compile Java programs with the debugging option switched on. In addition, we set **the migration safe point** at the first bytecode instruction of each source code line. It means that migration can only happen after the complete execution of all Java bytecode corresponding to a single Java source code line, and before the execution of the next Java source code line. If a migration request is received in the middle of executing a Java source code line, the migration will be delayed until the end of execution of the current source code line. This source-code-level granularity eliminates the need to save operand stacks which are usually non-empty in the middle of the execution of a source line, and also avoids the need to save machine-dependent process state information which is present during the execution of a native method. Therefore, without modifying the JVM, the resulting system can then be as portable as any ordinary Java program.

Load Balancing and Controlling Module: A monitor agent is used to detect the available computing resources and network bandwidth between Grid points connected across the WAN. On the other hand, prediction on the application's future communication pattern is performed through on-the-fly bytecode analysis. The load balancing policy will integrate and analyze all those information to give some effective process migration instructions. Therefore, with the load balancing and controlling module, the whole middleware can control the applications' execution and make the execution much more efficient. This part of work is still under development. We hope to present this in the full version of the paper.

3. Authenticated and Restorable MPI Communication layer

3.1. Client-Server Message Redirection Model

The restorable MPI communication service is based on a client-server model. This layer is implemented as a group of MPI daemons in all participating processing nodes in all Grid sites. It is a bridge between Java-MPI communication API and underlying native MPI libraries. The Java-MPI communication API is the interface for parallel Java processes to send requests to MPI daemons. The MPI daemon running on each node of Grid sites is responsible for sending messages and receiving messages on behalf of the calling Java programs in the same node. The Java programs and the MPI daemon in the same node communicate through one or several message queues reserved by the middleware.

In order to provide efficient MPI communication, inter-node or inter-site communication done by MPI daemons are through the native MPI library. Instead of linking the Java program directly with the native MPI library, the native MPI library is linked by the MPI daemon such that MPI communication is used exclusively by MPI daemons in different nodes for their communication. This approach requires no modification of the existing MPI library. The implementation of MPI library we use is MPICH-G2. MPICH-G2 uses services provided by the Globus Toolkit to coordinate work and enable authenticated inter-site communication in the Grid environment. And it can automatically convert data in messages sent between machines of different architectures and supports multi-protocol communication by automatically selecting TCP for inter-machine messaging and vendor-supplied MPI (where available) for intra-machine messaging.

3.2. Authenticated Communication services

In the Grid environment, security becomes an important issue. This characteristic is one of most significant differences between traditional clusters with Grid. Thus, any sharing mechanism must have the ability to authenticate the user's or job's identity and determine whether the user is authorized to request the resource. The restorable MPI communication services can also be taken as a kind of shared resource. Like normal Globus jobs, the processes of a Java-MPI job can access normal resources on a user's half with their delegated credentials associated with the user's credential. However, our MPI communication layer, as a server, aims to provide communication services to multiple Java-MPI jobs which may belong to the same user or different users. Therefore, to identify different Java-MPI jobs, we especially introduce another kind of credential (called Java-MPI job credential) associated with a particular Java-MPI job. Through this credential, the processes belonging to the same Java-MPI job can identify and authenticate each other, therefore do authenticated communication.

This credential has two versions of implementations, among which one is for the server (MPI communication layer), another for each Java-MPI process. (1) The sever-side credential is implemented as a table which is shared among all servers in the global range. It consists of a globally unique identity key (identifying this Java-MPI job) signed by the MPI communication layer, the process distribution information, and the communication record

information for those processes residing in this node. The communication record information may include the sending and receiving message sequence numbers of those processes. This information is important for the global communication integrity. (2) The client-site credential is used for the processes to interface with the server. Most time it is used per-message, so that is implemented as a message envelop. Especially this kind of credential encloses the Java-MPI credential together with the corresponding *process rank* which exclusively identifies that single process. This credential also encloses that process's message sequence information for the message integrity. Every process will get a credential signed by the underlying MPI communication layer during the job start-up.

The authentication process consists of two stages. Fig. 2 shows the basic idea. In first stage, the Globus security mechanism checks the user program's identity using its authentication algorithm which is defined by Secure Socket Layer Version 3 (SSLv3) protocol. Usually this stage is needed for only one time. With proxy credentials and delegation, processes running on the user's behalf can access all resources other than MPI communication resource. In the second stage, the MPI communication layer checks the particular Java-MPI job's identity when the processes of this job attempt to request MPI communication service. Then the processes use their own process rank to identify with each other during inter-process communication. The most common requests on MPI communication service are sending and receiving an MPI message. The second stage of authentication is associated with the communication, therefore, this stage happens in per-message level. Basically, the authentication process involves the following five events:

(1) user credential authentication
(2) allocate resource (new process creation)
(3) request for Java-MPI communication services
(4) signing a Java-MPI job credential for process(C_{DJ})
(5) request of sending or receiving a MPI message with C_{DJ}

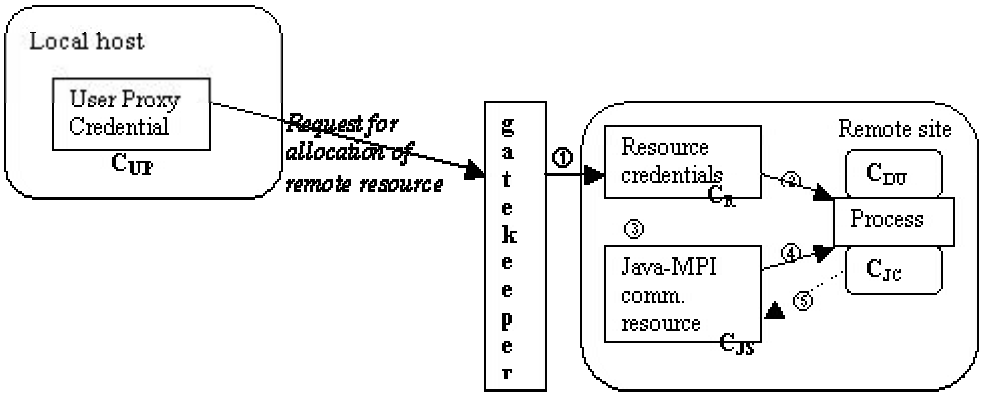

Fig. 2. Authentication Process: C_{UP}: user proxy credential. C_R: resource credential. C_{JS}: Java-MPI job credential in server-side. C_{DU}: delegated user credential. C_{JC}: Java-MPI job credential for process (in client-side).

Therefore, in G-JavaMPI, each Java-MPI process belonging to a user's application has a set of two credentials including the delegated credential associated with the user and the job credential associated with the particular Java-MPI job and process.

3.3 Restorable Communication Channel

In authenticated communication services, the Java-MPI job credential is mainly used by the processes to authenticate their identities to the underlying MPI communication layer, therefore get authorized interprocess communication with their peers. Except for that function, the Java-MPI job credential also provides important support for re-constructing

communication channel for the migrated process. In the global Grid system, a Java-MPI job credential identifies one unique G-JavaMPI job. A G-JavaMPI job may consist of many parallel processes which are distributed on different local nodes of all Grid points. When a process is migrated from one location to another location, its physical location changes. But its peer processes will not know this change as this migration mechanism is transparent to high-level applications. The migrated process will re-construct the communication channels with its peer processes through the abstract interface provided by its job credential. Because the job credential is constant and in the global range, the migrated process can always get contact with its peers in any physical location.

4. Transparent Migration Layer

4.1. State Capturing using JVMDI

The Java Virtual Machine Debugger Interface (JVMDI) [15] is a native interface available for the JVM since Java 2, and is used typically by debuggers. It defines the standard services that a JVM must provide for debugging. There are ways to inspect the state and to control the execution of applications. On the one hand, we can obtain the runtime information of threads, stack frames, local variables, classes, objects, and methods. On the other hand, JVMDI can be used to control threads, to set local variables, and to receive notifications of events such as method exit/entry and frame pop-up. JVMDI is called by the JVMDI client running in the same virtual machine as the application program being debugged. The application runs continuously if no debugging requests have been issued.

In G-JavaMPI, we make use of JVMDI to capture process states, as this can be done much more easily than other existing approaches. The migration layer is implemented as a JVMDI client. All the actions performed by the JVMDI client are transparent to the applications. In addition, the capturing mechanism is all on top of an ordinary JVM so that no modifications of the JVM are required.

Although JVMDI functions can inspect information on threads, stack frames, local variables, classes, objects, and methods, no functions are provided to extract and rebuild operand stacks. And data in the operand stack are JVM-dependent. In addition, when the execution point is inside the native method (frame), the local data in the frame are machine-dependent. All these factors make it very hard to capture and restore operand stacks and also destroy the portability of the middleware. Therefore we introduce an approach to makes sure that all operand stacks are empty and the execution point is outside of a native method at the time of migration. This is achieved through setting **Migration Safe Point** and **bytecode rearrangement**.

Migration Safe Point is defined as any execution point on the first bytecode instruction of each source code line only in a Java method. If the execution point which a Java process reaches to when migration instruction is received is in the middle of executing a Java source code line or inside a native method, the point is not safe for migration. In these two conditions, the migration will be delayed until reaching the next migration safe point. However, this **source-code-level migration granularity** only assures the empty operand stack in the current frame. The bytecode rearrangement introduces some new local variables to store those intermediate values in the program's bytecode file before execution. Therefore, the combination of the source-code-level granularity and bytecode rearrangement can make the operand stacks of all frames always empty during migration.

4.2. State Restoring using Exception Handler

For the state restoration, we basically follow the same mechanism used in M-JavaMPI [18]. For the sake of completeness, the working mechanisms are briefly discussed here.

This mechanism is to add the capacity of restoration exception handling to the application's bytecode in advance, so that the new process can perform restoration of execution state with the help of migration layer. Restoration functions are inserted as exception handlers to perform restoration under the control of migration layer. Exception handlers are inserted in each of the methods. The exception handlers catch and react to *restoration exceptions*. Inside these exception handlers, local variables of the called methods are pre-set with the saved information, and a "jump" command is issued to branch to the position saved during capturing.

During restoration, a breakpoint is set at the start of each method associated with stack frmes. When any breakpoint is caught, the migration layer will throw a restoration exception. Then, in the corresponding method, the exception is caught by the **Restoration Exception Handler** where local variables of the method are restored to the saved values. A "branch" command is then performed to jump to the last executed location of the current frame. This action is repeated for each frame of the program until the last frame is re-established. Then the program will execute again from the last executed position.

4.3. Delegation of Credentials During Migration

Apart from the execution state capture and restoration, the migration process also involves stopping the old process and creating a new process at the remote destination site. Unlike those in a cluster environment, the remote process creation cannot be done simply through remote shell or other mechanism without specialized protection of security. In G-JavaMPI, the security mechanism for the remote process creation is an extension from the mechanism provided by GSI (Globus Security Infrastructure) in the Globus toolkit. The extension is mainly for the delegation of Java-MPI job credentials. Fig. 3. shows the basic operations.

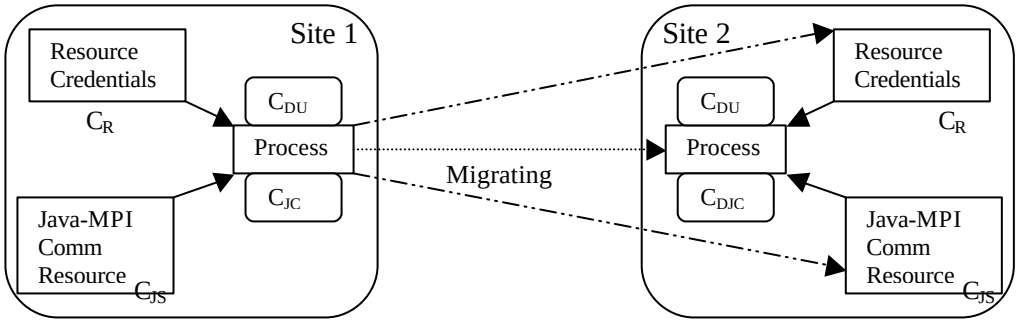

Fig. 3. Delegation of Credentials During Migration: C_{DU} : delegated user credential C_{JC} : Java-MPI job credential in client-side. C_R : resource credential. C_{JS}: Java-MPI job credential in server-side. C_{DJC}: Delegated Java-MPI job credential

In fig.3, the process in site 1 will be migrated to site 2. Before the process in site 1 is stopped completely, it sends requests to the resources in the remote site 2 with its set of credentials. The requests include the request for the new process creation to normal resource and the request for the MPI communication services to the Java-MPI communication resource (Java-MPI Authenticated & Restorable Communication Layer). On one hand, the new process creation is authorized after the authentication check of the delegated user credential. On the other hand, the new process created gets a delegated Java-MPI job credential (C_{DJC}) from the old process, and then is authorized to use the MPI communication resource with this delegated credential. After that, the old process in site 1 can be stopped safely, and the new process continues the execution in remote site 2, on behalf of the original process. The underlying restorable communication layer will help to re-construct the communication channel between the new process with all other processes.

4.4. Migration Process

The migration process is like the following. When starting JVM, the JVMDI client is started as well to wait any migration instruction from the middleware control block. When receiving a migration instruction, the initialization for migration is firstly done. The initialization process mainly checks whether the execution point is on the **Migration Safe Point.** When migration is ready to occur, the client suspends the execution of that process. And at the same time it sends a message to the local MPI daemon to notify it of the migration. After that, it inspects and saves all the Java stack frames created by the migrating Java process. For each frame, local variables, referenced objects, the name of the classes and the class methods, and the program counter need to be saved into a package using *object serialization*. After the saving, the captured data package is sent to the destination node. And before the JVM where the migrated process resides in is stopped, the process sends a request of new process creation to the destination node resource with its own set of two credentials. After passing the authentication, a new Java-MPI process is created with its own set of two credentials that are both delegated from the original process.

In the destination node, when the data package is received by the MPI daemon, the MPI daemon sends a notification message to the migration layer. Then the migration layer throws a **Restoration Exception** to the newly created instance of the process. The Restoration Exception handler inserted in each method reads values from the data package using mechanisms provide by *object serialization* and restores those values to local variables. After all frames finish restoration, the thread can be resumed to continue its execution.

5. Performance Evaluation

The implementation of G-JavaMPI involves three system softwares: (1) Java Development Kit : Sun JDK 1.4.1 [21] (2) MPI library : MPICH-G2 [3] (3) Grid toolkit: Globus Toolkit 2.0 [12]. The introduction of our middleware may have an impact on the performance of both the computation and communication parts. The computation part could be affected by the state-capturing and state-restoring actions and the use of JVMDI, while the communication part could be affected by the authenticated & restorable MPI communication mechanism. We divide the evaluations into two important modules: evaluation of the performance of the MPI communication layer among intra-sites or inter-sites, evaluation of the performance of the process state-capturing and restoring mechanism.

The experiments were conducted on two 16-node clusters. Each cluster is taken as a Grid point. All PCs in a duster are connected by a 24-port Fast Ethernet switch. The two clusters are connected by a single 100Mb/s fast Ethernet link. Each PC has a 733MHz Pentium III processor and 392MB of memory, running Linux 2.2.14 with Sun JDK 1.4.1 and MPICH 1.2.4. All Java programs were executed in full-speed debugging mode which is new feature provided by Sun Java Hotspot JVM in Sun JDK1.4.1 [20].

We have tested the communication bandwidths attained respectively for inter-site communication and intra-site communication. The fig.4 and fig.5 show the performance difference caused by different communication mechanisms. Inter-site communication has caused extra latency delay based on intra-site (about 48-52 ms for short messages with size of 1-32 bytes) and lower bandwidth (lost 7.54% of the peak bandwidth of the intra-site communication at 2KB). As the physical network bandwidth inside a grid point and between two grid points are basically the same in our experiment, the communication performance gap between intra-site and inter-site is mainly caused by the different communication mechanism inside a Grid point and between two Grid points. Intra-site communication is done by the MPI implementation, while inter-site communication is done through TCP communication. In addition, for the security purpose in the underlying Globus service, the inter-site TCP communication must go through the local Grid point job-manager to the remote Grid point. We think this kind of mechanism cause part of the performance lost. And we analyze that the amount of lost bandwidth should be somewhat bigger than 7.54% in the real Grid

environment, as this inter-site communication performance will be slowed down by the real WAN communication environment.

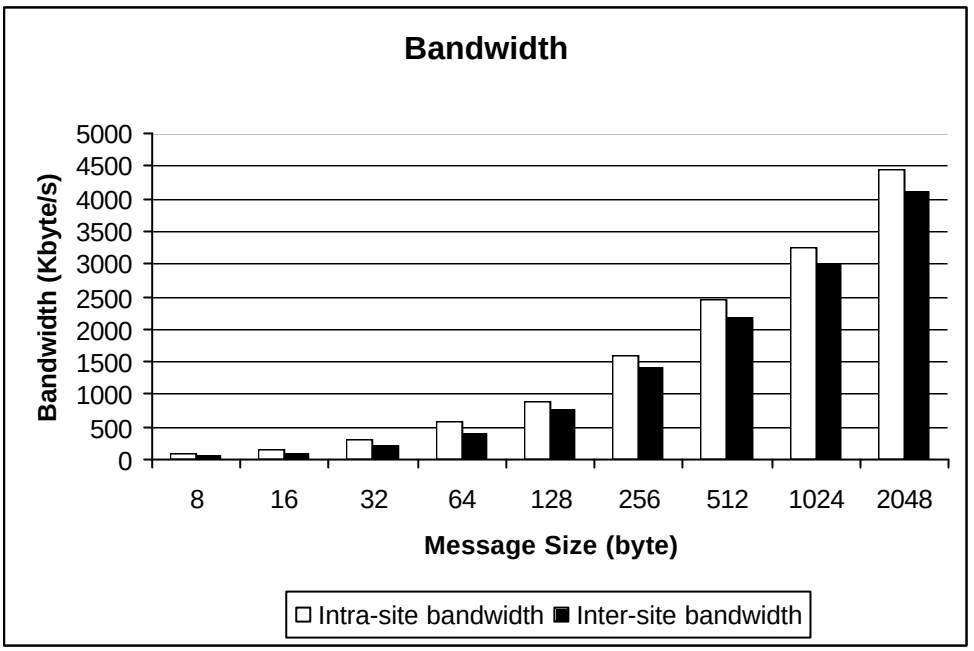

Fig.4. Bandwidth comparison between inter-site and intra-site communication with the installation of the MPI communication layer.

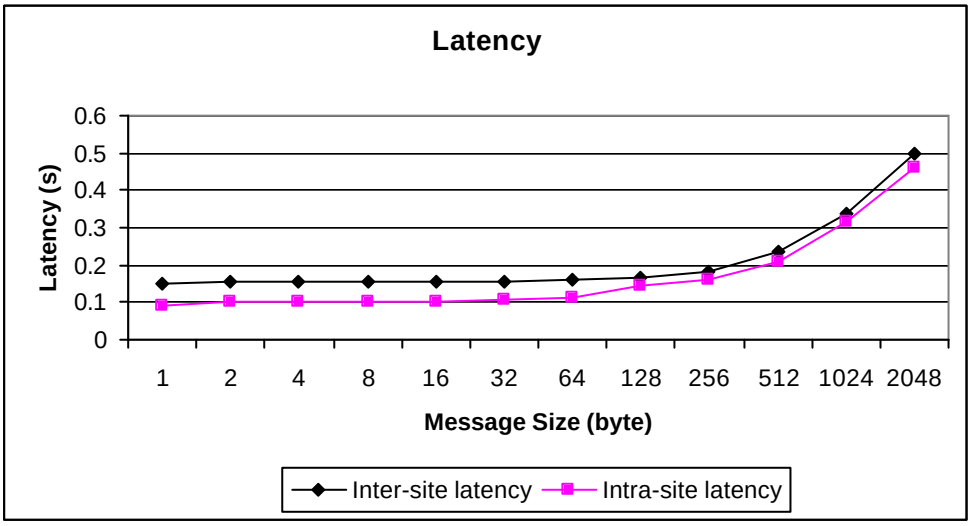

Fig.5. Latency comparison for small messages between intra-site and inter-site communication with the installation of the MPI communication layer.

The migration cost equals to the sum of time spent in capturing state in the source node, the time spent in restoring state in the destination node, and the time spent in starting the JVM and loading the program in the destination node. We evaluated the time spent in capturing state separately on the objects and the frames. Similarly, the time spent in restoring state includes two parts: time spent in restoring the objects and time spent in restoring the frames. The times spent in restoring both objects and frames are also evaluated.

Fig.6.shows the time needed in capturing and restoring objects of different sizes. In this test, objects that were used are arrays with variable type "byte". The data size of a "byte" is 1 byte. The minimum overheads in capturing and restoring objects are 1.01 and 0.175 ms respectively. The capturing time is about 0.158 μs/bytes and the restoring time is about 0.038 μs/bytes.

Fig.7 shows the time needed in capturing and restoring frames. In this test, no local variables were defined in each frame. Hence, the measured time is the minimum overhead in capturing and restoring different number of frames. The capturing time is about 4.5ms/frame and the restoring time is about 2.68ms/frame. The overhead is mainly caused by the process of retrieving the frame information and storing the data structure describing that frame.

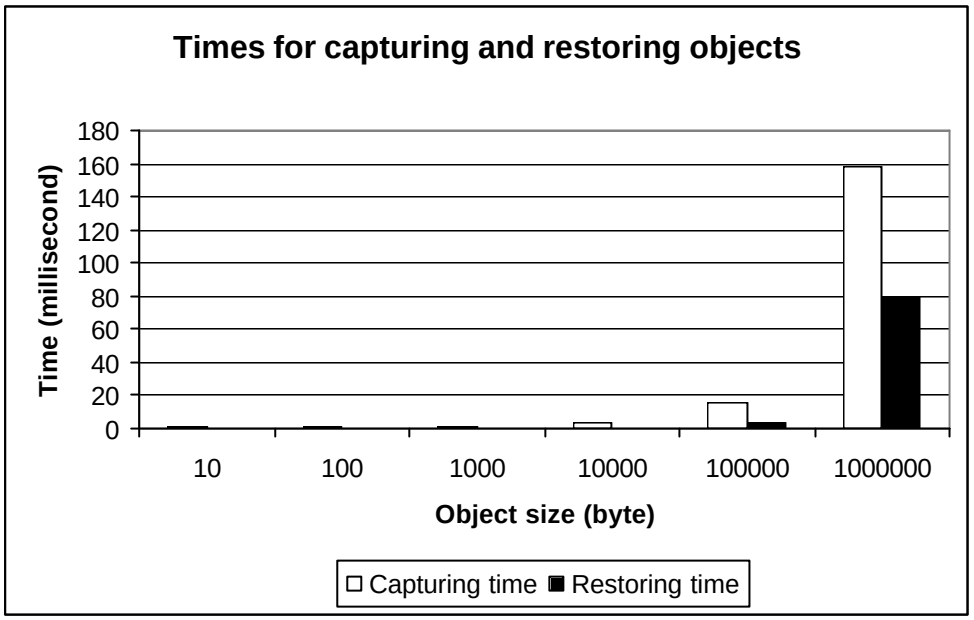

Fig.6. Time spent in capturing and restoring objects

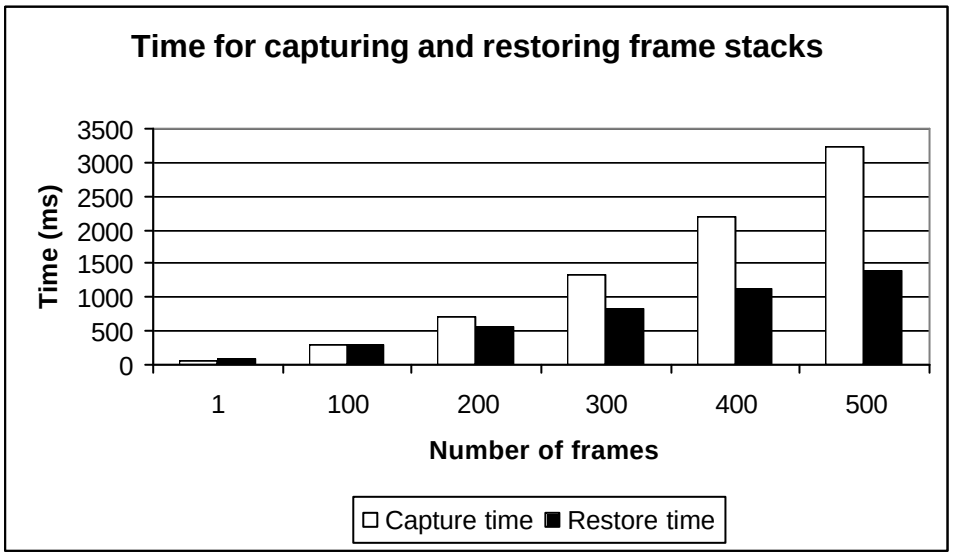

Fig.7. Time spent in capturing and restoring frames

Related Work

The Message Passing Interface (MPI) is a widely adopted communication library for parallel and distributed computing. There have been efforts to provide MPI for Java language [9,10,11,13,14]. Existing approaches to MPI for Java can be grouped into two main types: (1) native MPI bindings where the some native MPI library is called by Java programs through Java wrappers [9,10,11], and (2) pure Java implementations [13,14]. The native MPI binding

approach provides efficient MPI communication through calling native MPI methods. Conflicts could arise on the use of system resources such as signals between the MPI library and the JVM. The pure Java implementation approach on the other hand can provide a portable MPI implementation since the whole MPI library is rewritten in Java, but the MPI communication would be relatively less efficient since Java operates at a higher level. All of them however did not include any restorable message-passing communication feature. Moreover, our Java-MPI layer is "MPI-implementation-independent," which makes our system more portable. All those researches only support the execution in the cluster environment, and our middleware is Grid-enabled so that it provides necessary mechanisms for the security protection (authentication and authorization) and message integrity.

There are systems, such as JESSICA [1], Ara [6], and among others [5,7], that provide state-capturing and restoring of Java programs, but these systems need to modify the JVM. Some work [4,17] has been done to allow state-capturing and restoring via pre-processing of bytecode. Our approach is different from this in that extra bytecode is added only for state-restoring but not for capturing execution state. Besides, in their approaches, code is added to change the original program flow in order to do state-capturing and restoring. This could translate into considerable amount of overhead during runtime. In our approach, code is inserted as exception handlers which will only be executed during restoring.

Our Grid security mechanism is based on GSI (Grid Security Infrastructure)[8, 19]. GSI, a key component of the Globus Toolkit [12], greatly simplifies the usage of multiple machines in Grid environment through user single sign-on mechanism. Different from this, our mechanism is specially providing the security protection and message integrity for the communication-toward applications running across Grid points. Its special support for the global-range communication enables those communicating processes to move anywhere freely and transparently, therefore enabling better resource sharing.

7. Conclusion

In this paper, we have presented a new middleware for the Grid with Java-MPI communication and transparent process migration features. This middleware enable people to write MPI-style programs in Java language easily in the Grid environment. The transparent Java process migration mechanism will help the user applications to exploit and use the most appropriate resources. This will also support the development of any dynamic load balancing policy or fault tolerance mechanism. All these features will help Grid applications to utilize the Grid resource more efficiently and survive from the poor WAN bandwidth in the Grid environment. In future, we will make more evaluation on the Grid applications' behavior to help developing efficient dynamic load balancing policy.

References

[1] M.J.M. Ma, C.L. Wang, F.C.M. Lau, ``JESSICA: Java-Enabled Single-System-Image Computing Architecture," *Journal of Parallel and Distributed Computing*, Vol. 60, No. 10, October 2000, pp. 1194-1222

[2] Ian Foster, "The Grid: A New Infrastructure for 21st Century Science", *Physics Today*, Vol. 55, Februrary 2002 (URL: http://www.aip.org/pt/vol-55/iss-2/p42.html)

[3] MPICH-G2: http://www.hpclab.niu.edu/mpi/

[4] M. Dahm. "Byte Code Engineering". *Proceedings JIT'99*, 1999.

[5] M.Ranganthan, A. Acharya, S. D. Sharma and J Saltz. "Network-aware Mobile programs". *Proceedings of the USENIX Annual Technical Conference*, Anaheim, California, 1997.

[6] H. Peine and T. Stolpmann. "The architecture of the Ara platform for mobile agents". *Proceedings of the Second International Workshop of Mobile Agents* (MA'97), 1997.

[7] S. Bouchenak. "Pickling threads state in the Java system". *Proceedings of the third European Research Seminar on Advances in Distributed Systems* (ERSADS'99), 1999.

[8] R. Butler, D. Engert, I. Foster, C. Kesselman, S. Tuecke, J. Volmer, V. Welch. A National-Scale Authentication Infrastructure**,** *IEEE Computer*, 33(12): 60-66, 2000.
[9] S. Mintchev. "Writing Programs in JavaMPI". TR MAN-CSPE-02, Univ. of Westminster, London, UK, 1997
[10] Sava Mintchev and Vladimir Getov. "Towards portable message passing in Java: Binding MPI" Technical Report TR-CSPE-07". University of Westminster, School of Computer Science, Harrow Campus, July 1997.
[11] M. Bake. "mpiJava: A Java interface to MPI". *1st UK Workshop on Java for HKCN*, 1998.
[12] Globus Toolkit 2.0: http://www.globus.org
[13] Tong WeiQin, Ye Hua, Yao WenSheng. "PJMPI: pure Java implementation of MPI". *Proceedings of the 4th International Conference on High Performance Computing in the Asia-Pacific Region*, 2000.
[14] K. Dincer. "A Ubiquitous Message Passing Interface Implementation in Java: jmpi". Proceedings of 13th International and 10th Symposium on Parallel and Distributed Processing, 1999.
[15] Javasoft. "Java Virtual Machine Debugger Interface". http://java.sun.com/j2se/1.4/docs/guide/jpda/jvmdi-spec.html
[16] Javasoft. "Java Object Serialization". http://java.sun.com/j2se/1.4/docs/guide/serialization/index.html
[17] Eddy Truyen, Bert Robben, Bart Vanhaute, Tim Coninx, Wouter Joosen and Pierre Verbaeten. "Portable Support for Transparent Thread Migration in Java". In Proceedings of International Symposium on Agent Systems and Applications/Mobile Agents (ASA/MA'2000), September 2000, Zurich, Suisse.
[18] Ricky K. K. Ma, Cho-Li Wang, and Francis C. M. Lau, ``M-JavaMPI: A Java-MPI Binding with Process Migration Support," *The Second IEEE/ACM International Symposium on Cluster Computing and the Grid* (CCGrid 2002), Berlin, Germany.
[19] I. Foster, C. Kesselman, G. Tsudik, S. Tuecke. A Security Architecture for Computational Grids**.** *Proc. 5th ACM Conference on Computer and Communications Security Conference*, pp. 83-92, 1998.
[20] Java HotSpot[TM] virtual machine "full-speed debugging": http://java.sun.com/j2se/1.4.1/docs/guide/jpda/enhancements.html
[21] Sun Java2 SDK1.4.1: http://java.sun.com/j2se/1.4.1/docs/