jMAD- A small Java Mobile Agent Development Platform

Abhishek Yenpure
Dept. of Information Technology,
Sinhgad Institute of technology and Science,
Pune, India

M Nitin Umesh
Dept. of Information Technology,
Sinhgad Institute of technology and Science,
Pune, India

Dhiraj Patil
Dept. of Information Technology,
Sinhgad Institute of technology and Science,
Pune, India

ABSTRACT
Grid Computing has been around for as long as the modern computers themselves yet have failed in obtaining a clear standard which would define its implementation across the computing domain. This project focuses on the Process Migration aspects of Grid Computing/Clusters for Java Applications. The available solutions for Java Process Migration and Mobile Agent Development require high level of Java Expertise for the user itself let alone the developer. These solutions also require changes to be made in the JVM (Java Virtual Machine) which is a hassle for SMEs who cannot afford to put such high level solutions into work. The project solves the problem by establishing a method of Process Migration which will not need any changes in the JVM and will not require the user to know the internals of the implemented method. The user will only have to worry about developing his code remaining totally aloof of what will be done to it later.

General Terms
Grid computing, Process Migration, Mobile Agents

Keywords
jMAD, Object Serialization, Code Instrumentation, Migration, Broadcast, JVM (Java Virtual Machine), JADE (java Agent Development Framework).

1. INTRODUCTION

1.1 Problem
Process Migration is the prime need of fault tolerant systems running in Grid Environments and Clusters. The standardization of such concepts and methods used to achieve the goals of such concepts are very difficult because of the vividness of the implementations and different perspectives of the ideas used to implement the same thing in different environments in terms of technologies (Languages, Architectures, etc.). In environment running/developed on languages like C/C++/Fortran the process Migration is much easier than the new higher level languages like Java etc.

In Java the main problem is that it does not let developers play around with the process its properties like address spaces etc. for security purposes and hence said to be running in a sandbox [3]. Unlike C/C++/Fortran Java does not allow access to method stacks and Program counters to track the program. This leads to a problem of Process Capture which will help in saving the executing process and resurrect it according to the need of the user.

Hence because of the above problems we needed to address the Process migration problem in Java. It's not that such a thing was never tried before; the difference is that all the existing implementations require the implementers to significantly change the JVM (Java Virtual Machine) its implementation [1]. This leads to another headache of how the novice users/developers of the system will deal with a thing highly technical like specifications of JVM/Java Language with bare minimum resources as in case of a lot of SMEs.

The Problem is how to migrate the Java Processes without modifying the JVM in any way and without the user needing to deal with the technicalities of the implementation so that he will have to worry only about his own Program its implementation and not how the system will handle his program.

This project provides a way to handle the exact need of the day and implements a platform for development of Mobile Agents to achieve Process Migration in Java using Method Level Granularity by the development of Mobile Agents.

1.2 Solution
Java does not allow Users/Developers to access the address spaces of the current objects in execution discouraging the system help for capturing the objects in execution. But however java provides various inbuilt interfaces which allow us to store the state of the current executing objects in Files which may be transferred across systems and can be used to resurrect the objects and form a part of some other process, this technique is called as 'object serialization' and the interface provided by Java for this is 'java.io.externalizable'.

Having thus solved the problem of saving executing the state, tracking of the process can be done using checkpoints. We can allow the whole process to run between checkpoint to checkpoint forming executable granular units in the same method. This will help in resuming the process on other machine by starting its execution form the checkpoint it had ended on previously. Checkpoints can be implemented in the .java file or the .class file of the user the parts of which are discussed further in section 2 .

The whole approach provides seamless migration of Java process running on a cluster only introducing the overhead at the time of the class load event or the compilation of the class depending on the approach of the introducing code in the user files for process capture and tracking (introducing checkpoints).

Various performance tests have been done on the already similar system [1] which our project is based on and the results were obtained positive thus supporting our take on the system which would yield in a high performance implementation of the Grids and Clusters.

1.3 Scope and Visibility
The project can take a really big scope if taken into consideration all the parameters of the Grid Process Implementation and Cluster Application deployment (as done

in MOSIX etc.) into account and hence the project is limited to a few select and important aspects of the implementation due to lack much insight, the major part of which will be dealing with the Java processes only.

The project will be as transparent, robust as possible so for it to become a part of the new ecosystem of middleware and system software which will be completely purpose built.

As a part of the commercial feasibility of the project will be subjected to various performance tests and will be compared to the approach of implementation with "M-JavaMPI" [1] on which our project is based on and JADE library [7].

The project is to make use new technologies used in industries for application development like bytecode instrumentation, reflection and introspection, object serialization apart from all the trivial technical aspects of the project and will give an insight of industrial application development and understanding computing in a whole new way to the users.

2. IMPLEMENTATION

The Project will incorporate java code instrumentation before compilation(not to be confused with Compile time Bytecode instrumentation) to insert checkpoints to track the progress of the program and object serialization [2,4,6] for the storage of objects and other important runtime information, by appointing Java only features. The developers can gain a great deal in debugging the system as well as the system will provide the user with the instrumented code too, unlike the JVMDI approach used in M-JavaMPI [1].

2.1 Pre Processor

The pre-processor will use the support of Java Reflection [2,5,6] to get all the Fields and Methods declared inside the Program, thus it will help cache all the variables, objects inside a class, while the user has no need to write his class to be externalizable, the pre-processor will be responsible to do that by instrumenting the java code, the java code will now necessarily need to implement the methods declared in the interface, hence the preprocessor will need to write the two methods readExternal() and writeExternal() [4,6] explicitly. These methods will help in the storage of all the objects in the current java Process.

To write all these objects to the storage, here the preprocessor will need to know all those objects by the method Class.getDeclaredFields(); [2,5,6] which returns all the Fields declared within the class, hence after it knows the names and types of all the objects in the class, it would write them to the storage by virtue of writeExternal().

These files after storing to the file system will then be sent to the node which is willing to accept the process form the current Node and then the accepting node will resurrect all the objects by using the method readExternal().The Pre Processor may make the code look really verbose, but it might not at all put a lot of effort on the system then it did earlier.

Apart from just declaring these two methods, the Pre-processor will also be responsible for the insertion of checkpoints in the program.

The schematic of the preprocessing layer is as shown below in figure 1.

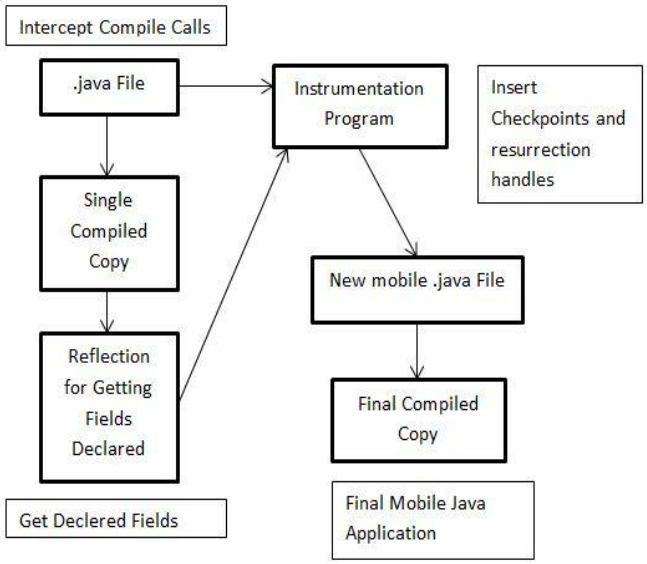

Fig 1: Schematic of the Pre Processor

2.2 Object Serialization

Object Serialization [2,4,6] is the Method provided by java for the storage of objects or synchronization of objects in the current execution context of the Java Virtual Machine the technique uses two of java's interfaces

- **java.io.serializable**
- **java.io.externalizable**

In the above two interfaces externalizable implements serializable interface too so that whenever the externalization of object is to be performed, the objects are serialized and would not require any other mechanism for locking the objects under externalization. The externalizable interface provides two methods for the storing and resurrection of the objects of the classes that implement the interface. This is necessary that whatever objects are to be saved need their classes to implement the externalizable interface of java. This technique will help us to store the state of the objects currently in the execution.

2.3 Checkpointing

The approach for keeping the track of the process of the during the execution appointed by M-java MPI [1] are the services provided by the JVMDI (now JVMTI) which gives access to all the regular program execution parameters (Stack, Program Counter), but this project implements this by using static variables as checkpoint storage and then using named blocks/conditional loops for one logical block of execution which could be encapsulated in cases of the switch block of Java. This will enable the original author of the program to gain transparency into the codes instrumentation and hence will put him in control again.

3. ARCHITECTURE

3.1 Architecture of the System

Based on the system's requirement and referring to the paper "M-JavaMPI" [1] the architecture we see of our system is somewhat like the figure 2 The Systems has the Following basic and predominantly important blocks

- **Pre-Processing Layer**
- **Migration Layer**
- **Communicating Layer**

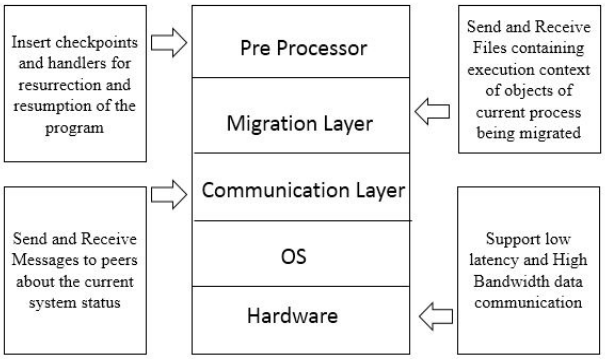

Fig 2: Architecture of the System

The functions of which will be briefed eventually in the further briefing of the architecture and mechanisms to be adapted in the system implementation.

3.2 Communication between Nodes

Fig 3 shows the communication will take place in the system and what messages will be sent by the nodes. Initially, each machine communicates all its processor and memory information/status to all other machines using a UDP message. Consider machine A from figure 3, it broadcasts a UDP message containing the information about its processor and memory information/status. In response, the other machines send a UDP message to machine A and all other Nodes containing the status of their processor and memory usage. Machine A stores the statuses of these machines in an Index file. From this Index file (in the actual implementation, we will be using IP-Value pair mapped in a HashMap in Java Collection interface), it selects the machine for process migration having the least processor and memory usage. The information about the Processor and memory utilization is updated periodically (after every 10 sec say) so that the system can be updated regularly without over populating the channel and letting resources for communication by waiting too short but also not letting the systems change their resource utilization by waiting too long.

This trade-off between communication times is very essential because the system cannot be risked for just communicating and then letting the process to a higher powerful machine sacrificing its own resources in communication itself.

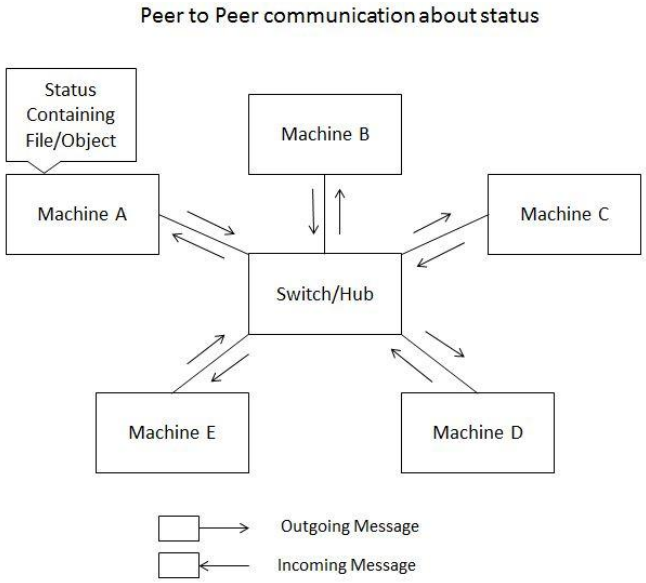

Fig 3: Communication between Peers/Nodes

3.3 Migration Initiation

Figure 4 describes the initiation of process migration. Now depending on the entries made in the Index file of machine A, the machine having the lowest value in terms of its status (processor and memory usage) is chosen. After getting the machine to migrate, the object saving process begins with object serialization. As described earlier the object Serialization is responsible for the saving of the objects of all the classes on the current running process. It's necessary to track all the possible classes the process is going to refer to before actually running the process so that proper provisions could be made so that the objects of those classes are saved and sent with the process to the other end and will not throw any run time exceptions on the other machine.

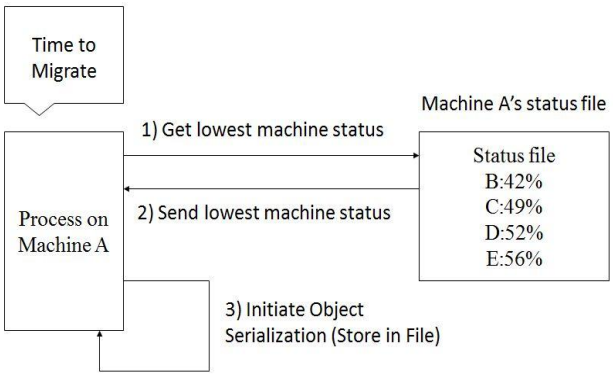

Fig 4: Initiating Process Migration

This is achieved by the concepts of Reflection and Introspection in java. The addition of the pre-processor layer will ease the things by changing the program of the user sufficiently so that the problems of referencing are solved by completely. The figure 5 shows the schematic of what was said previously about the referencing problem of the objects of other classes in the process. This approach is convenient in one way by establishing the channel between the main() methods of the two machines for the resurrection process as the objects in the main can be stored and forwarded and by using special

communication protocols the main() methods can communicate with each other and then form the objects explicitly without resurrecting them but by querying the objects of the other main() method however this will create a burden on the system and then deviate from our statement of not incurring extra cost on the system. (but not dealing with statement level granularity)

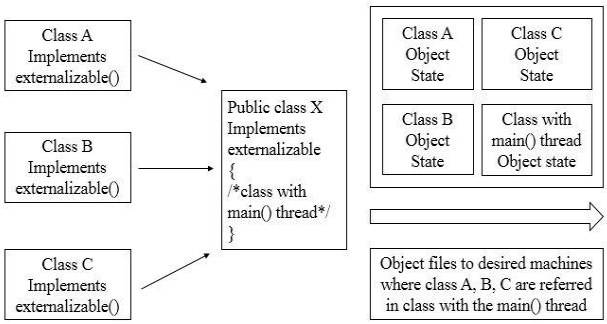

Fig 5: Classes referenced in the Class with main() thread

This is also needed to be known that the project is dealing with granularity as a certain block of statements and not at a statement level unlike M-JavaMPI [1] because without the support of JVMDI (now JVMTI) it is not possible to achieve that, and to a certain extent, it keeps the development of our proposed system a bit low in terms of complexity.

This also helps in omitting the need of capturing the intermediate results of the statement execution and storing them too while we try to migrate, hence instead letting the group of statements form that set run on the system and do all the changes to all the objects that occur in that block and then store them and migrate which reduces the system overhead for doing unnecessary thing than executing what is important in our perspective.

This ultimately benefits the node over which the execution is being shifted by reducing the no of statements that it requires to execute. Also bytecode instrumentation if not done properly give rise to a whole new class of exotic errors and exceptions, making our approach a bit more secure too.

3.4 Migration Exchange

Figure 6 represents objects that are to be saved by serialization are saved in the files by using the writeExternal() [2,4,6] method specified in the java.io.externalizable interface. The argument to the method is the ObjectOutput which is an interface implemented by the ObjectOutputSteram class so that the Objects can be stored in the files. After all those prerequisites of saving the object files, the files are sent to the other machine for resurrection, which is done by the method of readExternal() [2,4,6] method specified in the java.io.externalizable interface. After the resurrection of all the objects, the process could finally start the execution once again. Figure 6 can be described as; machine A sends the object state file to the (chosen) machine B. Machine B resurrects the objects and then begins the execution of the process from the last saved checkpoint. After completion of the process, the updated object state file is sent back to the parent machine A. To deal with the need of knowing whether the process being invoked is new or migrated, the system incorporated a custom environment cache which stores the information of all the processes and its parent in an IP-Process pair in a java HashMap similar to that of status storage.

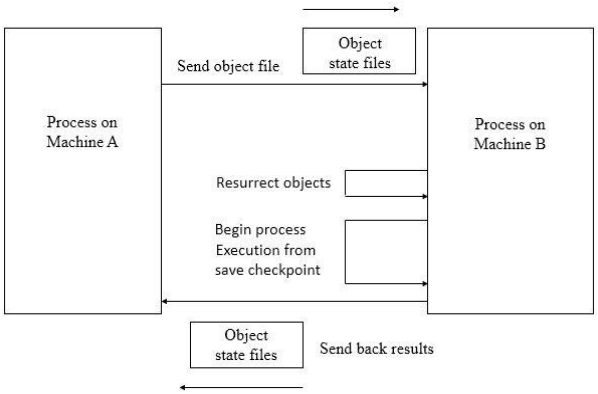

Fig 6: Steps in Migration of the Process

The overall System Work is not yet seen clearly even after understanding the entire Architecture of the System, more elaboration is needed in the light of how the migration might take place.

The Steps that every process must follow while migrating are:

A. Check is the system is overloaded, if so throw a SystemOverloadedException and continue to saving all the objects.

B. Get the IP of the host which has the minimum utilized resources, and ask it to register the process the migrated process in its cache by calling the Remote procedure register() (Using RMI).

C. Transfer all the Files (with objects and variables stored) to the remote Machine.

D. Resurrect the Process with all the objects by referring the local cache to determine that the process being resurrected is the new or a migrated process, accordingly restart the process by using the checkpoint transferred using the call(int checkpoint) method in the Process.

E. Establish a channel to transfer intermediate results of the process.(Channel is be established using sockets, and the results are captured by Process.getRuntime() defined in java.lang package)

4. PERFORMANCE

The Project is still in development in the significant parts, and therefore evaluating the practical performance of the system is difficult, however the theoretical evaluation is as follows. A more detailed and practical case study of performance evaluations will be provided subsequently after the system has been completed in its entirety. Table 1. provides an insight into the performance aspects of the system.

The project is primarily compared to the existing technologies of JADE (Java Agent development Framework) and M-JavaMPI (Migration-Java Message Passing Interface) in the aspects of Development, Compilation, Runtime of process.

Table 1. Predicted comparison of performance

Aspect	jMAD	Existing Platforms
Development	No expertise in the fields of Mobile agent development is needed, user need not be aware of the system	Expertise of Mobile agent development is needed (JADE) [7]
Compile Time	Longer than normal due to additional task of instrumentation and using Reflection on already compiled Time(2 Compilations are needed)	Short, the Agent code is totally developed prior to Compilation
Runtime	Transparent, as all the processes are cached in the Custom registry for their migration and execution	No such task is achieved, the user has no control and surveillance over the process activities like JADE
	No Runtime Code Rearrangement is needed, i.e. Byte code Instrumentation as done in M-javaMPI	M-JavaMPI needs byte code instrumentation which increases run time efforts on the system [1]

As discussed above jMAD requires two compilations, one for the original non migratable application on which we carry out reflection and introspection and the second time the instrumented program which is now a mobile agent. The time that sits between the two compilations is the time required to instrument the program. For convenience let us consider a base line as a non migratable application which needs to be developed into a Mobile Agent, and let this application have $(LoC)_0$ lines of Code. The compile time can be given by,

$$(C.T.) = \lambda \{(LoC)_0$$

Where, λ=Mean time required to compile each line

In terms of jMAD the Compile time can be calculated as

$$(C.T.)_{jMAD} = \lambda(LoC)_0 + \lambda\{(LoC)_0 + (LoC)_m\} + X$$

Where,
C.T =compile Time.
$(LoC)_0$ = Lines of Code in non migratable application
$(LoC)_m$=Lines of Code instrumented to make the code migratable
λ=Mean time for compiling each line
X=Time required to instrument the code

Here X is considered separately as the code is instrumented explicitly and not by the user. But in the case of JADE such instrumentation is done during the development of the code itself which requires the user to carry out the task of telling the program how the migration has to take place. Also only one compilation is required to get the migratable program unlike jMAD.

In terms of JADE the Compile time can be calculated as

$$(C.T.)_{JADE} = \lambda \{(LoC)_0 + (LoC)_m\}$$

Where,
C.T =compile Time.
$(LoC)_0$ = Lines of Code in non migratable application
$(LoC)_m$=Lines of Code instrumented to make the code migratable
λ=Mean time to for compiling each line.

Hence there is significant change in compile times of the applications developed by jMAD and JADE given by

$$(C.T.)_{jMAD} = (C.T.)_{JADE} + \lambda (LoC)_0 + X$$

This proves,

$$(C.T.)_{jMAD} > (C.T.)_{JADE}$$

Graphically the performance is shown as in figure 7. jMAD requires more time to be compiled but this can be preferred over JADE for convenience of transparency and ease in the development of Mobile Agents.

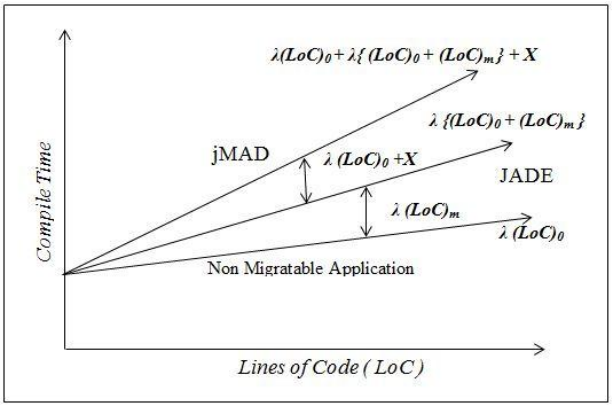

Figure 7: Performance graph compared to JADE

5. CONCLUSION

The Development of this technology will be a great availability to the small and medium enterprises without significant resources for planning a total full scale grid solution, and have a limited expertise in the development of the applications on the same. It might be developed into a full API for Mobile Agent Development once totally implemented.

A Hypervisor to analyze performance of all the Nodes in the system can be developed and integrated with the proposed library to give a complete solution for development and deployment of Mobile Agents.

6. FUITURE SCOPE

A full-fledged library/API could be developed for the development of the Mobile Agents in a small Grid Environment which would be of the likes of JADE.A

Distributed Hypervisor could be developed so that all the processes running in the environment could be made aware of the Nodes available for migration and the system will be aware of all the processes running on all the other nodes as a new processes or migrated processes.

7. ACKNOWLEDGEMENTS

We would like to thank **Prof Sushant Gote** of **Sinhgad Institute of Technology and Science, Pune-41, India** for guiding this research and enabling us get a vision to achieve the goals mentioned in the above project.

8. REFERENCES

[1] Ricky K. K. Ma, Cho-Li Wang, and Francis C.M. Lau. "M-JavaMPI: A Java-MPI Binding with Process Migration Support" presented at Cluster Computing and the Grid, 2002. 2nd IEEE/ACM International Symposium

[2] Herbert Schildt "Java 2:The Complete Reference" , Tata McGraw Hill,2002

[3] Oracle Technology Network," Java Language and Virtual Machine Specifications", Internet: "http://docs.oracle.com/javase/specs/".

[4] http://www.jusfortechies.com, Object Serialization, Internet:http://www.jusfortechies.com/java/core-java/externalization.php, [Sep 18,2012]

[5] Reflection in Java, Internet:"radio.javaranch.com/val/2004/05/18/1084891793000.html"

[6] JAVA 2 SE 7 Documentation(Oracle Java Documentation), Internet : "http://docs.oracle.com/javase/7/docs/api//"

[7] JADE Programmer's Guide, Internet:"http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.80.9915&rep=rep1&type=pdf"