Strong Mobility in Open Source SAGE MultiAgent System
Misbah Mubarak¹, Sara Sultana¹, Zarrar Khan¹, Hajra Batool Asghar¹, H Farooq Ahmad², Fakhra Jabeen¹
¹National University of Sciences and Technology (NUST)
²Communication Technologies, 2-15-28 Omachi Aoba-ku, Sendai, Japan
hajra-mcs@nust.edu.pk, fakhra@niit.edu.pk

Abstract
Implementing mobility in MultiAgent systems is one of the major challenges these days. True mobility consists of movement of code, data and execution state. There are various techniques through which true mobility can be achieved in MultiAgent systems. Certain mechanisms are present that are used to capture the state of a thread and re-establish it at the destination. All these mechanisms have different aspects of implementation. In this paper we analyze these approaches for the implementation of mobility and evaluate them based on software quality parameters. We determine their pros and cons. The ultimate goal is to suggest the implementation of mobility for SAGE MultiAgent system.

1- Introduction
Mobility is currently riding the rising wave in the computing industry as it offers a large amount of flexibility to users. It is the movement of code, data or execution state from one point to another. There are two types of mobility i.e. weak mobility and strong mobility. Weak mobility is the movement of code and data to a remote host whereas strong mobility is the movement of code, data and execution state from one computational unit to another.

An agent is a piece of software that can take an autonomous and independent action on behalf of its owner or user [1]. A mobile agent is an agent that can move among various nodes of a network and carry out its operations depending upon its goals and available resources. It therefore uses mobility in its operations [4].

A MultiAgent System (MAS) is a platform in which agents communicate with each other in order to accomplish certain combined goals [6]. Software Fault Tolerant Agent Grooming Environment (SAGE) is an open source MultiAgent system. Foundation for Intelligent Physical Agents (FIPA) is an IEEE standard for MultiAgent systems. SAGE is a MAS that is built in accordance to FIPA specifications. Currently, it lacks strong mobility. In the paper we discuss several different approaches which can be used for the implementation of Strong Mobility in SAGE.

Mobile agents are mostly implemented in java. Java is a popular language in network programming due to the properties of object serialization, dynamic class loading and machine independence. Object serialization helps capturing the object state whereas dynamic class loading loads Java code [2]. These mechanisms of java language help the user to achieve weak mobility (code and data movement only). However, Java does not provide any service for achieving strong mobility. In java, access to execution state (Thread stack and Program Counter) is denied due to security reasons.

Strong mobility in java can be achieved by either:
- Extending the java virtual machine
- Modifying the source code
- Modifying the byte code
- Using java platform debugger architecture

![Strong Mobility in Java](image)

Figure 1: Strong mobility in java
Among these approaches, the first three approaches are most commonly used. In this paper we will discuss these approaches in detail. We will then provide a comparison for all these approaches based on software quality parameters. Execution state in java consists of the following basic parameters

**Java Stack:** A separate java stack is associated with every thread. This stack consists of a frame for every method that is called. When a method returns, that frame is popped from the stack.

**Object heap:** The object heap consists of all those objects that are used during the life cycle of a thread.
Method Area: The method area consists of all the classes that are used during the life cycle of a thread. Object heap and method area are shared by all objects [8].

2- Approaches for implementing Strong Mobility

Following are the approaches for implementing Strong mobility.

2.1 Extending the Java Virtual Machine

Java language compiler translates the java source code into intermediate byte code. This byte code consists of specific instruction set for the Java Virtual Machine (JVM). The byte code makes Java language portable, since it can execute on heterogeneous platforms. At the byte code level, it is easier to insert malicious code. For this reason, JVM does not provide any access to the run time execution stack of the program.

In order to accomplish true mobility, the Virtual Machine must be able to do the following

Complete Execution State Capture: The VM must be able to capture the complete execution state of threads, objects and classes that are present in the heap.

Resource Management in Java: Whenever a process starts, it has got specific permission rights. These rights stay with the process through out its execution till termination. Therefore, the amount of resources doesn’t vary. The VM must grant change in allocation of resources during its execution so as to accomplish strong mobility. [11]

In order to modify the JVM, the C++ data structures of the Virtual Machine are modified. We now discuss how this can be accomplished at the VM level.

The Java heap, data structure consists of Java objects, threads and classes.

Java Threads are instances of C++ class ‘JavaThread’. The Thread objects consist of method stacks. The method stack contains the frames of methods invoked by the thread. These frames are called ‘Stack frames’. The stack frame is usually composed of

- Operand stack which consists of the operands used in the life time of the thread
- Local Variables invoked in the method
- Program counter that keeps track of each instruction that is executed
- Pointer to the method

The java thread objects are based on native threads. Native threads are Operating System threads on which the Java threads are based. Native thread consists of a go () function which when called, executes the java thread. However, using native threads causes a lot of problems in capturing the state of the thread.

One of the key to extending JVM is to decouple the execution of native threads and Java threads. For this, the compilation model needs to be changed. The native thread has a stack that remains constant through out the execution of the byte code whereas a java thread consists of a stack which changes during the execution of the byte code. The coupling of native thread with the Java thread makes it very difficult to capture the execution state of a thread. Therefore, its important to decouple the native thread from the Java thread.

![Figure 2: Components of Java Heap](image)

![Figure 4: Extent of coupling in JVM and extended JVM](image)
When the old native thread is replaced by a new one, its only functionality is to call the go() function. However, it's not easy to decouple the original native thread from the Java thread. Whenever a class is loaded or initialized, there is always a call to the old native thread function instead of the Java Thread. A solution to the problem is to replace the native function calls with Finite State Machines (FSMs) which can be placed in the VM stack frame of the java thread. These FSMs retain their states and thereby call java functions based on the new native thread instead of calling old native thread functions. In this way, the native threads can be completely decoupled from the Java Thread. The state of the Java Thread can then easily be accessed by modifying its C++ data structure [3].

In order to enforce security in terms of resource usage, a limit should be placed on it. Resource usage like CPU usage, disk reads and writes should be kept in proper check and control. These parameters are usually expressed in terms of bytes executed per milliseconds. Sometimes they are also expressed in terms of percentage of CPU time. Therefore, a limit should be placed on the bytes read or written in the first case. In the other case a limit can be placed on the CPU usage in the parameters. This satisfies the security demands of Java. In case any virus attempts to infect the system through excess CPU usage, it cannot succeed due to the limit placed on its usage.

Another issue that needs to be resolved during the extension of JVM is that of type checking. Java stack is only a C++ data structure which doesn’t have the capability of type recognition. Since the variables types are represented in a different manner on heterogeneous architectures, this creates a strong need for type recognition. The problem can be resolved by using ‘byte code instructions’. The instructions at the byte code level are typed. There are different instructions for integers, double and arrays. With the help of these byte code instructions, a type stack can be built in parallel to the Java stack. This type stack consists of the type recognition of the operands and variables being used in the program.

**Figure 5: Resource Consumption**

One of the major advantages of this approach is its added efficiency. Since the JVM is directly modified and there are no code over heads, the approach is highly efficient. The disadvantages of this approach are lack of portability. Modification of JVM causes lack of standardization which violates portability [4].

### 2.2 Changing the compilation model

We give a brief overview of the approaches that can be utilized by changing the compilation model.

#### 2.2.1 Adding state saving constructs in the Source code

In this case modification is done at the source code level instead of the JVM. Programming effort is to design a separate virtual stack made of Java programs. This virtual stack is kept up to date during the entire execution of the program. Whenever migration is triggered in a program, the source code of the program is modified by adding the state saving constructs of this virtual stack. This makes a back up object which now consists of the complete program along with the code, data and execution state. The state saving information consists of the methods that the program was executing when the migration got triggered, state

---

**Figure 3: Architecture of the JVM**

**Figure 5: Resource Consumption**

---
of the local variables and the value of the program counter.

Figure 6: State saving constructs used in preprocessing the code

In this approach, the parameters related to Java Defined Class are important. These parameters can be used for state saving and resuming.

At the destination site, whenever the state of the program is to be restored, the state saving constructs consisting of the program’s execution state need to be recompiled.

The advantage of the approach is its portability. Since it runs on the same standardized JVM, it can execute on any machine that supports Java. Its disadvantage is its increased code overhead that is generated whenever the additional state saving constructs are added. Moreover, source code in java programs is often unavailable.

2.2.2 Modifying the byte code

In this approach, state saving constructs are added in the byte code. The thread then migrates to the remote site along with its heap image and stack. A new thread is then created at the destination site whose state is initialized with that of the migrated thread. However, in transferring the heap image of a thread, severe security threats are present. The execution state of the program consists of the similar constructs that are added in the source code modification technique. These constructs include program counter, local frame variables and operand stack. These constructs are now added at the byte code level. The thread or any agent migrates between the Java Virtual Machines (JVMs). Therefore the underlying platform supported by the approach is the JVM.

The approach has got several advantages. The point at which migration is triggered can be a compound statement. This instruction is unavailable at the byte code level.

However, when modifications are done at the byte code level, they must pass through a byte code verifier. The byte code verifier does not allow modified byte code to pass through it. Therefore, it has to be redesigned to allow such modifications. Moreover another disadvantage of the approach is that the variables types are unknown at the byte code level.

A comparison of Strong Mobility’s approaches

In terms of efficiency and performance, the JVM level approach is better since it does not consist of additional coding constructs that have to be added in the case of source and byte code level approaches. Whenever mobility is triggered, the execution state always moves along the mobile agent.

In reliability of achieved mobility in terms of failure, the JVM level approach is again better. It does not require addition of any coding constructs which reduces chances of failure. Once the JVM is modified and works correctly, there are few chances of failure.

The JVM modification approach is not portable. Since changes are made at the JVM level, therefore the modified JVM lacks standardization. The source code and byte code instrumentation techniques are portable as there is no change in the VM. Changes are only limited to the compilation model.

The JVM level approach is difficult to implement since it requires working at the middle ware level. The byte code approach is also difficult to implement as it requires modification at byte code level which is closely related to machine language.

JVM level approach has got restrictions in modification of the VM as Sun puts licensing constraints on it. Remaining techniques are free from such restrictions as they don’t require modifications at the JVM level.

JVM approach achieves true mobility as it does not has any problems as are mentioned in the compilation techniques in the previous section.

In case of source code instrumentation technique, source code is required on both source and the destination sites. JVM modification and byte code instrumentation do not require source code.
Table 1: A comparison of Strong Mobility approaches

<table>
<thead>
<tr>
<th>Name of the Approach</th>
<th>Efficiency Level</th>
<th>Reliability</th>
<th>Portability</th>
<th>Difficulty of implementation</th>
<th>Licensing constraints</th>
<th>Quality of Mobility Achieved</th>
<th>Source code requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source code modification</td>
<td>Low</td>
<td>Moderate</td>
<td>Available</td>
<td>Moderate</td>
<td>None</td>
<td>Moderate</td>
<td>Yes</td>
</tr>
<tr>
<td>Byte code modification</td>
<td>Moderate</td>
<td>Moderate</td>
<td>Available</td>
<td>Difficult to implement</td>
<td>None</td>
<td>Moderate</td>
<td>No</td>
</tr>
<tr>
<td>JVM modification approach</td>
<td>High</td>
<td>High</td>
<td>Not available</td>
<td>Difficult to implement</td>
<td>Sun licensing constraints</td>
<td>High</td>
<td>No</td>
</tr>
</tbody>
</table>

Conclusion

We have discussed several approaches for implementing strong mobility in SAGE. Every approach has got its positive and negative aspects. Any of the approaches can be used for implementing mobility in SAGE Multiagent system. The pre-requisite is to include all the possible cases for example if the thread is in the monitor, the case should be handled. In this way strong mobility under any circumstances will be achieved.

References

[1] Sarmad Sadik, H Farooq Ahmad, Arshad Ali, and Hiroki Suguri: Enhanced inter platform mobility in SAGE Multiagent system
[5] Niranj Suri, Jeffrey M. Bradshaw, Maggie R. Breedy, Paul T. Groth, Gregory A. Hill, and Renia Jeffers: Strong Mobility and Fine-Grained Resource Control in NOMADS, Lecture notes in computer science (Lect. notes comput. sci.) ISSN 0302-9743, Institute for Human and Machine Cognition, University of West Florida, USA
[8] Sara Bouchenak, Daniel Hagimont: Picking thread states in the java system, In proceedings to the SIRAC project, Montbonnot Saint Martin, France
[9] Chow Yuk thesis: Journal and author unknown