STRONGLY INTRUMENTAL GENERIC MOBILITY ARCHITECTURE

by

NC Misbah Mubarak

PC Sara Sultana

NC Zarrar khan

Submitted to the faculty of Computer Science Department Military College of Signals, National University of Sciences and Technology, Rawalpindi in partial fulfillment of the requirements of BE degree in Computer Software Engineering

April 2007

List of figures

1 Introduction

With the growth of Internet as a primary environment for communication and development of distributed applications, there is a strong need to change the old design paradigms and to switch onto the new ones. A global network such as Internet must exploit different forms of mobility in order to increase its usability and scalability requirements. Mobile agents can be used to satisfy the scalability needs of the highly dynamic global network. Mobile agents are autonomous software entities with the capability of dynamically changing their execution environments in a network aware manner. Mobile agent technology is being promoted as an emerging paradigm that helps in the design and implementation of more robust and flexible agents. Mobile agents are preferable over other design paradigms due to certain features that other paradigms do not provide. These features include disconnected operation, reduced bandwidth, reduced latency, increased stability and increased sever flexibility.

In case of code mobility, distributed applications move mobile agents while they are executing. When these mobile agents acquire the capability of resuming their execution instead of restarting at the destination, they are called strongly mobile agents. Therefore, Strong mobility is the movement of code, data and execution state of an agent whereas weak mobility only moves the code and data of the agent. Code of a program consists of programs whose methods are being executed by the thread. Data consists of the values of local variables and registers. Thus in case of weak mobility, execution of the agent or the object restarts on the destination.

Java has accelerated the use of transportable code over the internet. It provides many impressive features for distributed computing like serialization, dynamic class loading

Appendix A

An approach to ontological interoperability

Misbah Mubarak[1], Sara Sultana[1], Zarrar Khan[1]

Hajra Batool Asghar[1], H. Farooq Ahmad[2], Fakhra Jabeen[1]

[1]Computer Science Department, MCS, National University of Sciences and Technology (NUST)

[2]Communication Technologies, 2-15-28 Omachi Aoba-ku, Sendai, Japan

hajra-mcs@nust.edu.pk, fakhra@niit.edu.pk

Abstract

Multiagent systems (MAS) have gained importance in recent years due to the numerous advantages they provide for the field of distributed computing. Agents in MAS have specific collective goals to accomplish. In order to achieve these goals agents need to have a meaningful communication. This can be achieved through the use of common shared vocabulary. A more formal term used for this vocabulary is 'Ontology'. While using ontologies, there are different problems that can be encountered. For example, if an agent from one domain having an ontology wants to communicate with an agent in another domain having a different ontology the problem of interoperability arises. Our paper focuses on the problem of interoperability among different ontologies.

Key words: *ontology, Knowledge Base, Multi Agent system, First Order Logic, ontology merging, ontology mapping, ontology integration.*

1- INTRODUCTION

Ontologies define the vocabulary of a specific domain in terms of its concepts, terms and their relationships. For the entities of a domain to have a shared understanding of certain concepts, a common ontology is required. Ontology has the added benefit of capturing the semantics of a domain independent of its representation. [1, 12]

Today, heterogeneous and independent information sources are widely distributed. With the advent of semantic web and other services, there is a stringent need for interoperability among various data sources. Ontologies, being an explicit and formal specification of data, help in finding correspondence between different data sources [2]. Ontological interoperability problem arises when dealing with ontologies from different domains. When there are two domains having different ontologies following two conflicts can usually occur

- Homonym terms; i.e. terms having same names but different semantics/meaning.

  AND

- Synonym terms; i.e. different names for semantically same terms

Three possible solutions have been presented to the problem [2]

1. Ontologies can be standardized.
2. Mediation between ontologies can be done.
3. Ontologies can be merged.

Initially, the solution presented to the problem was the standardization of ontologies. However, it is very hard to standardize ontologies. When talking about a single organization, standardization requires a lot of changes within that organization. It requires consensus among the group of people within the organization. Across organizations, the group of people involved in the decision expands and they all need to reach consensus, thus aggregating the problem still further.

A better solution to the problem is to include a mediator or any third party agent between the two ontologies. The role of mediator is to bridge the differences between the ontologies. This can be done through the 'mapping' process. Mapping is used to identify and eliminate the differences between two source ontologies. In the mapping process any conflicting term in one ontology, is expressed using the corresponding conflicting term in the other ontology. The mediator must be able to identify synonym and homonym terms. It must then perform a mapping so that the conflicts are resolved.

One form of ontological mapping is merging. In merging a new ontology is formed by combining the existing ontologies. The old ontologies are then replaced by the new ones. However this produces the overhead of replacements. Moreover it may create dangling references to the old ontologies. In our paper we have focused on the approach of mediation between ontologies.

There are various ways in which ontology can be represented. In our paper we have chosen First Order Logic (FOL) as it is more expressive and simple [5]. It is built on the top of propositional logic which is a declarative, compositional semantics that is context independent and unambiguous. FOL also borrows representational ideas from natural languages while avoiding their drawbacks of ambiguity. It is a powerful tool for knowledge representation and reasoning. Being mathematical and formal, FOL notation presents the most suitable means to represent ontologies unambiguously. Moreover, others ontologies have an inherent self-learning characteristic which enable them to grow dynamically. They use axioms and reasoning mechanism to infer new concepts from the existing ones [3, 4].

The rest of the paper is organized as follows. Section 2 describes the related work. Section 3 presents our approach and Section 4 explains an example to demonstrate the application of our approach. Conclusion and directions for future work are given in Section 5.

2- RELATED WORK

There are various ways in which agents in a certain MultiAgent System (MAS) communicate with agents having different ontologies outside the MAS.

In [6, 13], a similar approach is proposed which consists of an agent that can learn to improve its ontology in two ways and thus communicate with agents having different ontologies. First, users can teach them by supplying a list of words and the classifying concepts for that list. Second, an agent can learn through interactions with its neighbors by applying K nearest neighbor algorithm. As a result, each agent learns its own concepts based on its experiences and specialties. Whenever a new concept arrives, the agent needs to incorporate it into its dictionary. Learning can be incremental in which concepts are refined on a new submission, or it can be collaborative i.e. refining translation table whenever there is a query that prompts the agent seeks for help from its neighbors. When an agent receives a query, it checks it against its ontology knowledge base. For relaying the messages to the agents, a central controller may be used. However, when all the agents want to communicate with their neighbor, there is a chance of bottle neck. Our approach eliminates any chances of such performance bottlenecks. Moreover, if an agent encounters a synonym term it may not learn it under the assumption that it already has that concept. We provide a mechanism in our proposal that ensures to provide correct interpretations under naming conflicts.

In [2], for the integration of ontologies three different approaches have been analyzed. One of them is ontology mapping. In case of ontology mapping,

- Similarities are found between the two source ontologies that are to be mapped. The Match operator performs this functionality in many systems.
- Once the similarities have been found, the mappings between the ontologies need to be specified.

For mapping ontologies there are two options. First is of 'one to one ontology mapping'. In this case mappings are created between any two ontologies. This approach lacks efficiency since the complexity of this approach is O (n^2) where n is the number of ontologies. The second option is to use a global ontology. It's similar to the standardization of ontologies as mentioned in previous section and has got the same drawbacks. Our approach eliminates the use of a central repository for storing ontologies.

Another approach described in [7] proposes wrappers and mediators based architecture. The mediator mediates between the differences among the individual data sources. A wrapper is attached with every single data source to resolve the representation differences. There can be several mediators in case of one to one mapping of ontologies. However, this approach can be inefficient as the large number of mappings that may be stored in the mediator. Our approach eliminates the problem of such a large number of mappings as discussed in Section 3.

An approach for ontology merging is presented in [8]. In this approach the source ontologies are mapped onto single target ontology. There can be two options

- The source ontologies disappear and the target (merged) ontology exists.

OR

- Source ontologies exist along with mappings to the target or merged ontology.

The approach of merging ontology creates the overhead of replacement of the old ontologies with the new one.

In [9] an architecture for mapping distributed ontologies in semantic web has been proposed. The architecture is called MAFRA (MApping FRAmework for distributed ontologies). It is divided into five modules. The first is 'lift and normalization' module which makes a single uniform representation for ontologies. Second is 'similarities' module which calculates the similarities between the ontologies. The third module is 'Semantic Bridging'. It semantically relates source entities to target entities. Then is the 'Execution' module which actually carries out transformation from source ontology into target ontology using the information from the previous modules. Last is the 'post processing' module which analyzes the results in order to improve its quality for subsequent versions. In the MAFRA architecture, the intervention of an expert during the 'execution' module is necessary.

In [11] the approach of enriched conceptual mode is used. In order to use the enriched conceptual model knowledge engineers have to provide more details about the concepts which requires a lot of human intervention. By adding information on the concepts it is able to measure better the similarity between them (Synonym terms), to disambiguate between concepts that *seem* similar while they are not (Homonym terms). The aim is not to build a new knowledge model for ontologies but to support a semantically enriched description of attributes when defining concepts in ontologies. A disadvantage of this approach is that it cannot handle structural problems. Our approach will solve the problems related to structural conflicts as well as the large number of facts that would need to be resolved.

3- OUR APPROACH

Our approach focuses on distributed ontology deployment in MAS where each agent may have a separate ontology of its own (totally partitioned ontology deployment).

3.1 Architecture

The major components of our architecture are as follows.

- There is a separate ontology of every agent that is stored in a repository along with the agent.
- There is a shared and central ontological repository, but unlike other approaches its task is not to store the complete ontologies of all agents in it. Rather, it only stores the mapping and merging axioms between all the Ontologies of different agents.
- A rational agent is attached with this central repository [10]. The properties of this rational agent are given Table 1.The reason for choosing a rational agent is its ability to learn reason and acquire concepts. These characteristics are required in our architecture.

3.2 Working of the architecture

Our approach is divided into two phases.

- Handshaking phase
- Learning/integration phase

The handshaking phase starts with an agent from another world (MAS) say the invoker agent that starts the communication by invoking the rational agent in our MAS to inquire about the concept it wants. Once the communication is initiated, our rational agent performs two tasks.

1. It consults the global repository and selects the agent whose ontology has the concept required by the external invoker agent.

2. It then directs the invoker agent to the selected agent present in our MAS.

This completes the handshaking phase.

In the second phase of learning/ integration both agents are allowed to communicate with one another. The agents learn the concepts that are different. This can be done using the concept learning algorithm of machine learning [6, 13]. In this concept learning task, there is no intermediate rational agent required. Ontologies are then shared; the mapping is acquired from the repository whenever required in the concept learning process. The agent who wants to learn may present some examples that it thinks will fit in the concept [6]. If it wants to confirm that the concept is right, it may take help from the other agents by voting. Since mapping of all concepts is present, the semantic conflicts will be avoided. Based on the mutual classification of positive and negative examples the learning agent will decide about the classification of the concept.

Solution of homonym and synonym terms

The rational agent solves the conflicting concepts by taking homonym concepts and applies automated reasoning [4] to these concepts. It applies certain examples to the concepts and determines if their implications are the same. If it is not the same, then the rational agent constructs the bridging axiom and saves it in the global repository for the pair of conflicting concepts.

Similarly, when the rational agent encounters a synonym concept, it applies the same inference procedure to the concepts and determines if the implication is the same despite of different names. If yes, it maps the two concepts, makes the bridging axiom, stores it in the global repository and thus resolves the conflicts.

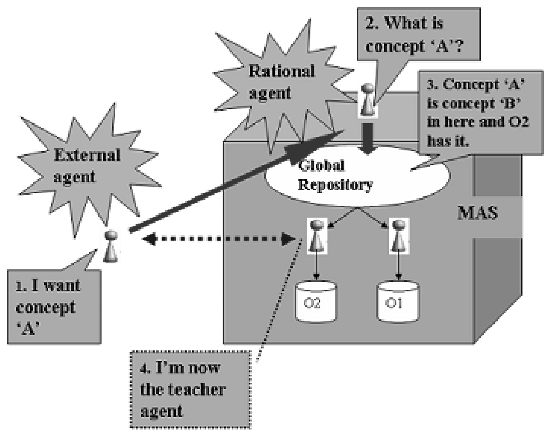


Figure 1: the design model

Performance of the system

If the number of concepts is very large, then many agents may be deployed. The social goals of a rational agent are competition, cooperation and negotiation. Therefore a small team of rational agents can manage this task of mapping concepts. The mapping of ontologies and the bridging axioms are kept in the central repository. For reliability purposes, and to avoid the bottle neck, replicated copies of the repository can be created. Since the repository only holds mappings, therefore the replication will not involve large amount of memory. The existing ontologies are kept up to date which resolves the issue of inconsistency in the ontologies.

This approach solves many problems highlighted in the previous approaches. There is an automated reasoning system which requires minimum human intervention. Bottle neck is not created as there is no central heavy repository which is being accessed by all agents. Replication is also done which will not occupy much of the memory space. The semantic conflicts are also avoided. An intermediate rational agent is required only in the initial phases. Later on communication is carried out without this intermediate agent.

The semantic conflicts are also avoided. An intermediate rational agent is required only in the initial phases. Later on communication is carried out without this intermediate agent.

3.5 Evaluation against software quality parameters

Reasons are

- The ontological mapping is performed by a rational agent who is faster than a mediator. A mediator is only an object, whereas the rational agent not only performs the role of an object but it has also got many additional properties associated with it [10]
- The rational agent performs its tasks in the initial phases of communication whereas a mediator is required through out the communication according to the architecture proposed in [7]. The overhead of indirect communication which introduces delay in the system is thus eliminated.

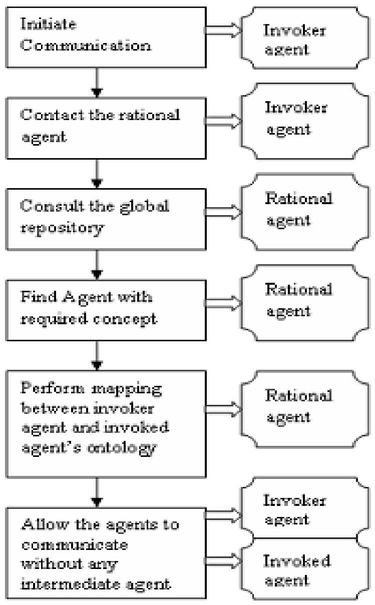
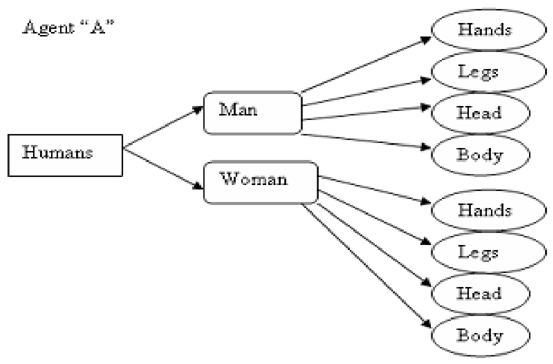

Initiate Communication
Invoker agent
Contact the rational agent
Invoker agent
Consult the global repository
Rational agent
Find Agent with required concept
Rational agent
Perform mapping between invoker agent and invoked agent's ontology
Rational agent
Allow the agents to communicate without any intermediate agent
Invoker agent
Invoked agent

Figure 2: Working of the architecture

The system is reliable as there can be many copies of the same rational agent without the need for heavy replication. A single point of failure will not affect the system.

4- EXAMPLE ILLUSTRATING OUR APPROACH

To illustrate our approach we present an example demonstrating the working of our ontology. Initially the design model is as given in the Figure2.



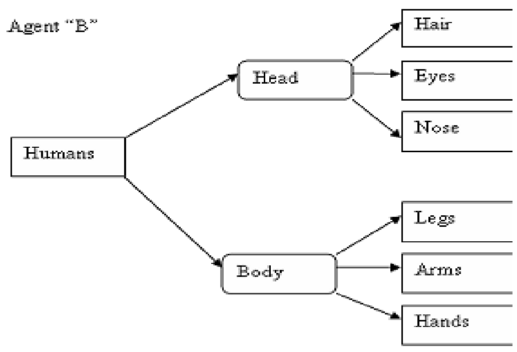


Figure 3: ontology of two different Agents 'A' and 'B'

First of all, we need to make a Knowledge Base (KB) of agents.

The FOL expression for the above mentioned agents having different ontologies will be as follows:

For Agent 'A', the knowledge base comprising of FOL expressions will become:

For all (x, y, z)

1- Man(x) ^ hands(y) ^ belongsTo(y,x)→ Human(x)

2- Woman(x) ^ hands(y) ^ belongsTo(y,x)→ Human(x)

3- Man(x) ^ Legs(y) ^ belongTo(y,x)→ Human(x)

4- Woman(x) ^ Legs(y) ^ belongTo(y,x)→ Human(x)

5- Man(x) ^ Head(y) ^ belongTo(y,x)→ Human(x)

6- Woman(x) ^ Head(y) ^ belongTo(y,x)→ Human(x)

7- Man(x) ^ Body(y) ^ belongTo(y,x)→ Human(x)

8- Woman(x) ^ Body(y) ^ belongTo(y,x)→ Human(x)

9- Man(x) or Woman(x) → Human(x)

10-Fingers (x) ^ total (x,5) ^_ palm(y) ^ attached(x,y) ^ belongsTo(y,z) → Hand(z)

11-Total (z,2)→ Hands (z)

12-Fingers(x) ^ total (x, 5) ^ palm(y) ^ attached(x, y) ^ belongsTo(y,z) → Leg(z)

13-Hands(x) ^ Legs(y) ^ attached(x,z) ^ attached(y,z) → Body(z)

14-Body(x) ^ attached(x, z) ^ round (z) → Head (z)

The KB of Agent 'B' is

For all x, y, z

1- Hair(x)^belongsTo(x,z)^ round(z) → head(z)

2- Eyes(x) ^ Total (x,2) ^ belongsTo (x,z) ^ round(z) → head(z)

3- Nose(x) ^ Total (x,1) ^ belongsTo (x,z) ^ round(z) → head(z)

4- (Straight(x) or curly(x)) ^_(color(x, brown) or color(x, golden) or color(x, black)) ^ belongsTo (x,z) ^ round(z) → hair(x)

5- Spherical(x) ^ Total(x,2) →_Eyes(x)

6- ColorOf(x, skin) ^ Long (x) ^ belongsTo(x, face) → Nose(x)

7- Fingers(x) ^ total (x, 5) ^ palm(y) ^ attached(x, y) ^ belongsTo(x,z) ^ belongsTo(y,z) → Leg(z)

```
10- Fingers (x) ^ total (x,5)
^ palm(y) ^  attached(x,y) ^
belongsTo(x,z) ^
belongsTo(y,z) → Hand(z)
```

```
11- Total (z,2)→ Hands (z)
```

```
12 – Long(x) ^ attached(x,
hands) → arms(x)
```

```
13- Has(x, head) → Human(x)
```

```
14- Has(x, body) → Human(x)
```

The ontology has been defined. Now it needs to be designed and implemented.

The deployment of ontologies is done using a hybrid approach which is shown in Figure 5.

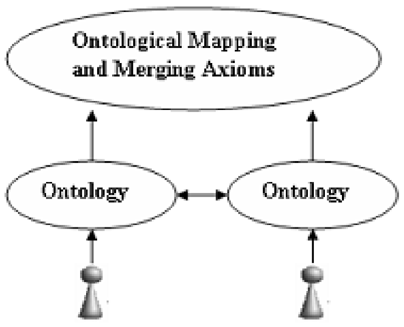


Figure 4: a hybrid ontology approach [8]

This approach is distributed; however there is a central, replicated ontological mapping.

Now comes the next phase, where the conflicts are resolved. Here, the ontologies are merged using bridging axioms. Agent 'A' wants to learn about the concept 'Arms' which is not present in its ontology. It does the following steps

- It contacts the rational agent asking for the concept 'arms'.
- The rational agent checks the global repository if any of the mappings of this concept is present. It finds that Agent 'B' has the required concept.
- Now the rational agent will check the concepts and will create certain bridging axioms. In this example, there are three conflicting concepts that need to be considered. These concepts are 'human', 'legs' and 'hands'.

Given below are the bridging axioms that will be used to demonstrate how the agent distinguishes between the hyponym concepts.

Let the domain of Agent 'A' be domA and let Agent B's domain be domB.

```
For all w, x, y, z
```

```
1-( ( ( Man(x) or woman(x) )
^  domain ( domA, x) ) or (
Head (z) ^ Body (y) ^ Belongs
to(z,x) ^ BelongsTo ( y, x)
)→ Human(x)
```

```
2-((( Man(x) ^ BelongsTo(y,x)
^ Long(y) ^ Total(y,2) ^
attached(y,z) ^ feet(z) ^
domainOf (domA, y)) or ((
Woman(x) ^ BelongsTo(y,x) ^
Long(y) ^ Total(y,2) ^
attached(y,z) ^ feet(z) ^
domain (domA, y ) or (
(Long(y) ^ Total(y,2) ^
attached(y,z) ^
attached(y,w) ^ Body(w) ^
feet(z) ^ domainOf(domB, y))
→ Legs(y) )
```

```
3- - (( Man(w) ^ Fingers (x)
^ total (x,5) ^ palm(y) ^
attached(x,y) ^
belongsTo(x,w) ^
belongsTo(y,w)^ domain (domA,
w) or (Fingers (x) ^ total
(x,5) ^ palm(y) ^
attached(x,y) ^
belongsTo(x,w) ^
belongsTo(y,w)  ^ domain
(domA, w)  ) or (Fingers (x)
^ total (x,5) ^ palm(y) ^
attached(x,y) ^
belongsTo(x,w) ^
belongsTo(y,w) )→ Hand(w)
```

The term 'human' is interpreted as a 'man' or a 'woman' in domA whereas in domB, it refers to 'head' and 'body'. So there is an axiom that tells that in domA, if there's a 'man' or a 'woman', then it refers to 'human' and in domB, if there is a 'head' and 'body', it is also 'human'. Same is the case for 'hand' and 'legs'. In this way, the rational agent resolves the conflicts.

Once the rational agent is done with the mapping and merging, it stores the above three axioms in the global ontological repository. Now, the two agents (Agent 'A' and Agent 'B') can communicate directly with each other without the help of rational agent. Agent 'B' becomes the teaching agent because its ontology contains the desired concept and Agent 'A' is the learning agent. Agent 'B' will present it concept of the axiom 'arm' to Agent 'A'. Agent 'A' will store that axiom in its local repository of ontology so that it can use it in the future.

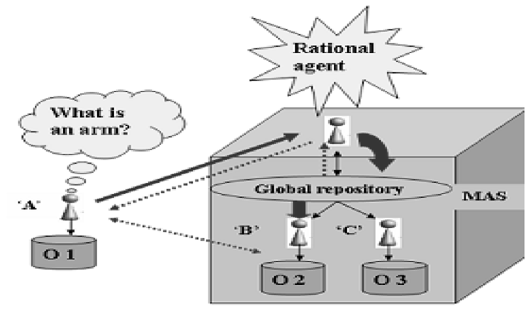


Figure 5: An example to illustrate our approach.

Agent 'A' will receive the concept of 'arm' as:

$Long(x) \wedge attached(x, hands) \rightarrow arms(x)$

5- CONCLUSION AND FUTURE WORK

In this paper, we described our approach for the development of a hybrid approach in order to solve ontological problems in Multi-Agent systems. This approach includes all the processes involved in ontological development, deployment and learning. It solves ontological problems like naming conflicts, and bottle neck. It makes the system more efficient. It employs a distributed as well as central approach for ontologies storage and mapping. Each agent has its own separate local ontology as well as a mapping of its concepts in a central ontology.

As a further enhancement of this approach, we intend to include the location awareness in ontologies. This can be done by making all the agents location aware where as their core ontologies can be stored in the repository. Moreover, as a continuation of our work, we want to implement this proposed approach.

REFERENCES

[1] M Andrea Rodriguez : Similarity based ontology integration M Andrea Rodriguez. In Proceedings of the 1st International Conference on Geographic Information Science, October 2000

[2] Jos de Bruijn (DERI), Francisco Mart´ın Recuerda (DERI), Dimitar Manov (SIRMA),

Marc Ehrig (UKARL) : State of the art survey on ontology merging and aligning. Semantically Enabled Knowledge Technologies (SEKT) deliverables, July 2004

[3] Dejing Dou, Drew McDermott, and Peishen Qi: Ontology Translation by Ontology Merging and Automated Reasoning, Computer Science University at Yale, July 2003

[4] Peter Baumgartner, Fabian M. Suchanek

: Automated Reasoning Support for First-Order Ontologies. National ICT Australia (NICTA), Institute for Computer Science, Germany, 2004

[5] A Modern Approach to Artificial Intelligence, Stuart Russel and Peter Norving. Prentice Hall Series in Artificial Intelligence, 2nd edition.

[6] Leen-Kiat Soh : Collaborative Understanding of Distributed Ontologies in a Multi-Agent Framework: Design and Experiments. In Proceedings of the 2nd International Joint Conference on Autonomous Agents and Multi-Agent Systems, Mellbourne Australia, July 2003.

[7] Sonia Bergamaschi, Silvana Castano, Maurizio Vincini, and Domenico Beneventano.

Semantic integration of heterogeneous information sources. Special

Issue on Intelligent Information Integration, Data & Knowledge Engineering,36(1):215–249, 2001.

[8] Natalya F. Noy and Mark A. Musen. Prompt: Algorithm and tool for automated

ontology merging and alignment. In Proc. 17th Natl. Conf. On Artificial Intelligence (AAAI2000), Austin, Texas, USA, July/August 2000.

[9] Alexander Maedche, Boris Motik, Nuno Silva, and Raphael Volz. Mafra a

mapping framework for distributed ontologies. In Proceedings of the 13th

European Conference on Knowledge Engineering and Knowledge Management

EKAW-2002, Madrid, Spain, 2002.

[10] Multiagent Systems, A distributed approach

Edited by: Gerhard Weiss. The MIT Press, Cambridge, Massachusetts. London, England.

[11V. Tamma, and T.J.M. Bench-Capon: An ontology model to facilitate knowledge sharing in multi-agent systems. In Knowledge Engineering Review, 17(1), 41-60, 2002

[12] B. Chandrasekaran and John R. Josephson: What are ontologies and why do we need them? University of Buffalow, Workshop on Ontologies 2

[13] Mohsen Afsharchi: Ontology Guided Collaborative Concept Learning In Multi-Agent Systems. Aamas 2006 Doctoral Mentoring Program, May 2006.

A review of Mobility Techniques

Misbah Mubarak[1], Sara Sultana[1], Zarrar Khan[1]

Hiroki Suguri[2], Hajra Batool Asghar[1], H. Farooq Ahmad[2], Fakhra Jabeen[1]

[1]Computer Science Department, MCS, National University of Sciences and Technology (NUST)

[2]Communication Technologies, 2-15-28 Omachi Aoba-ku, Sendai, Japan

hajra-mcs@nust.edu.pk, fakhra@niit.edu.pk

Abstract

Strong mobility deals with the movement of execution state of a program from one computational unit to another where as weak mobility only allows code and data to be moved. Many languages due to security reasons, do not allow accessing the execution state of a program. Therefore, implementing strong mobility in Multi-Agent Systems (MAS) is one of the biggest challenges these days. In this paper we analyze weak mobility, provide a classification for certain techniques that implement strong mobility. Based on this we categorize different MASs implementing these techniques used for implementing mobility.

Keywords: Strong Mobility, weak mobility, Multi-Agent Systems, Source code instrumentation, Byte code instrumentation, Java Platform Debugger Architecture (JPDA).

1- Introduction:

Mobile agents are the upcoming wave in the agent community. 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 accomplishes the operations by the use of mobility. Mobility is movement of code, data or execution state from one point to another. It is further classified into two types 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.

Many of the Multi-Agent Systems (MAS) are programmed in Java. Java is very popular in network programming because of the following reasons

Serialization: This property enables objects and data to be sent from source machine in marshaled form into an output stream. Later, upon reaching the destination machine the code and data are deserialized into their original form [2].

Dynamic Class loading: It allows class files to be loaded at run time.

Machine Independence: Java is a portable language as it supports virtual machine technology. It can be executed on heterogeneous platforms independent of the underlying architecture.

Code and data can be easily transferred using the above mechanisms of java language. However, execution state can only be transferred if one has got access to the run time execution stack of threads. However, Java does not allow access to run time execution stack of threads due to security reasons. Therefore, all systems that have been built at the top of java, support only weak mobility with the help of serialization and dynamic class loading mechanisms.

Multiagent Systems (MAS) in java are mostly multi threaded. Achieving mobility for these multi-threaded applications is an uphill task. The execution state of a thread consists of three main components [3]

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.

The object heap and method area are entities that are shared by all the threads.

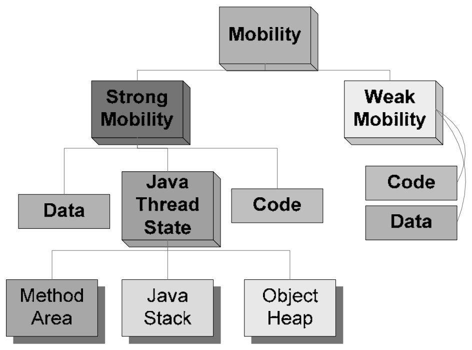


Figure 1: Different forms of Mobility

Since java does not allow dynamically inspecting the execution states of threads, therefore various techniques have been introduced in achieving strong mobility. These techniques are discussed in Section 2.

2- Mobile Code Techniques

The objective of this review is to analyze the different techniques that are used to achieve mobility. Based on these techniques, we can classify different Multi-Agent Systems and comment on their pros and cons.

The evaluation criteria are based on the form of mobility supported by different MultiAgent Systems.

2.1 Weak Mobility in MultiAgent Systems

MultiAgent Systems supporting weak mobility mandate the mobile agent to resume its execution at a pre-specified check point. This loses all the execution of the agent carried out on the source machine [4]. In most of the cases, only the data initialization constructs are carried along. The programmer or the developer has to deal with re-starting the agent at that specified point upon reaching the destination. Usually, this is done through event based programming. This consists of transferring the code and data in case of some specified events. This results in unnecessary overhead on part of the programmer [5]. It leads to an entirely different programming style that violates software design rules. It is also difficult to adapt by the programmers.

This approach of implementing weak mobility has the advantage of efficiency and portability. Weak mobility is efficient as there is no over head of state capturing and restoring. It is portable as it can execute on any system since it does not require any specific system requirements for its execution. Designers of weak mobility claim that it gives flexibility to the programmer in case of synchronization problems like dead locks. Moreover in case of multithreaded applications, weak mobility is claimed to be much more useful than strong mobility. However, there are many different forms of strong mobility that do not create these problems and negate these claims. This is further discussed in Section 3.

Figure 2 demonstrates the basic architecture of a weakly mobile system. It consists of code and data that is fetched from the application layer. This code and data is then serialized into an output stream and sent through the network layer to the destination machine where it is deserialized to its original form.

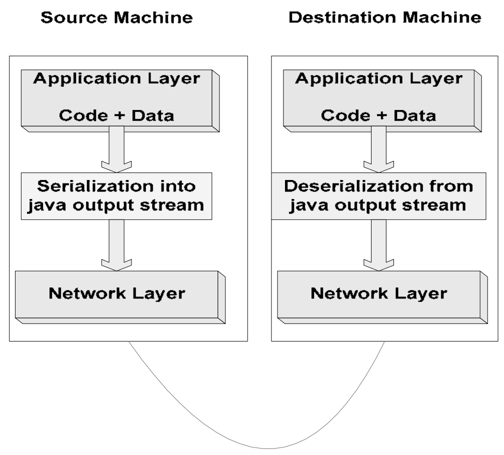


Figure 2: Basic Architecture of Weak mobility

Consider the scenario where an agent is executing on a source machine where it modifies certain global variables after which, it migrates to the destination machine. Since it cannot keep a track of its execution, it will restart at a specified point and re-modify the variables. Therefore, weak mobility is highly unsuitable in case of non-idempotent operations.

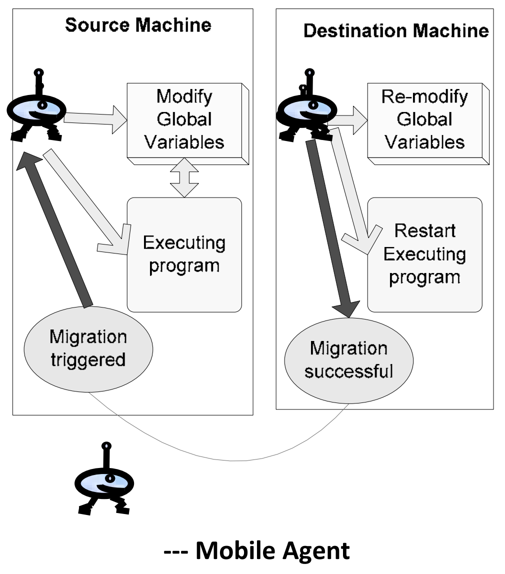


Figure 3: Problems with weak mobility

2.2 MultiAgent Systems Supporting Weak Mobility

Following are the systems that provide weak mobility support in Multi-Agent systems

2.2.1- IBM Aglets: Aglets is a system that is built on the top of Java and therefore supports weak mobility. It uses an event driven programming model. Agents are represented by threads. Operations such as dispatch are used to send code to the destination. The system also supports an operation namely 'retract' in which code is brought from any specified aglet. As mentioned previously, aglets in case of migration have to continue their execution from the beginning due to inaccessibility of execution state in java. However, due to serialization and class loading mechanisms, the values of certain objects are preserved [10].

2.2.2- Mole: mole was developed at the University of Stuttgart. It also supports weak mobility. Agents in mole are objects which act as threads. There is a differentiation between user agents and service agents.

2.2.3- Obliq: Obliq is another system providing a support for weak mobility. It uses synchronous agent execution. Agents are threads in Obliq. Whenever any agent is transferred to the remote site, it is suspended at the source site until the entire code is executed at the destination. In this way it supports synchronous weak mobility.

2.2.4- TACOMA

It's an architecture that provides support for weak mobility. The agents are implemented as UNIX processes that can move from one machine to another without preserving the execution states. The initialization data is preserved in a brief case.

2.3 Multiagent Systems supporting Strong Mobility

In order to incorporate strong mobility in MultiAgent systems, there is a need to change the compilation model or to modify the Java Virtual Machine (JVM) [5].

2.3.1 Strong Mobility using JVM modification approach In MultiAgent Systems

Since the JVM contains all the execution information needed to achieve strong mobility, one of the approaches to implement strong mobility is the modification of JVM. This change is made so that all the execution information (Thread state) is accessible to the programmer via the JVM. It thus requires changing the security constraints implemented by SUN. However, modifying and redistributing the source code of the JVM is against the licensing constraints of Java [6].

In this case, the thread is required to be captured at the source machine. At the destination machine, a new thread is created whose state is initialized as that of the thread that was captured at the source machine [3]

The benefit of this approach is increased efficiency as compared to the other approaches. There is no additional overhead of code generation. Therefore the system is highly efficient.

One of the major drawbacks of this approach is lack of portability. Since the JVM is modified, therefore strong mobility is applicable for only those systems which support that particular modified JVM. Moreover, redistributing modified JVM is against the licensing constraints of Java.

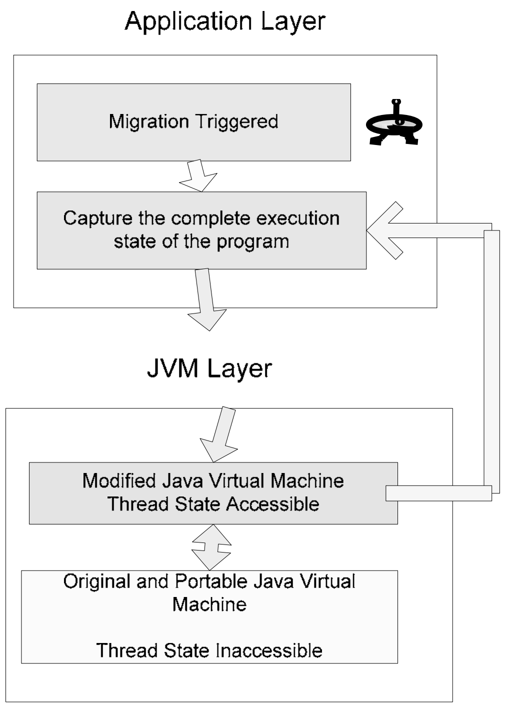

Application Layer
Migration Triggered
Capture the complete execution state of the program
JVM Layer
Modified Java Virtual Machine Thread State Accessible
Original and Portable Java Virtual Machine
Thread State Inaccessible

Figure 4: Basic Architecture of Strong mobility using modified JVM

2.3.2 MultiAgent Systems Supporting JVM modification approach

The Multiagent system supporting strong mobility using modified JVM is

2.3.2.1- NOMADS

NOMADS is a java based system with support of strong mobility. NOMADS environment is composed of two parts. One is an agent execution environment called Oasis and the other is a Java Compatible virtual machine called Aroma Virtual Machine (VM). Aroma VM is another system that provides a support for strong mobility. It provides this by using a modified Virtual Machine. Since this modification is not made in the Java virtual machine, therefore the licensing constraints do not matter. Whenever a host goes offline, 'forced migration' is implemented. It captures all the threads, objects and classes in the VM [13].

2.3.3 Strong Mobility using source code instrumentation approach In MultiAgent Systems

Another approach that can be used for implementing strong mobility is that of Source code instrumentation. In this approach, code statements comprising of state saving or state resuming operations are added to the source code of the program. This preprocessed code is then compiled to generate byte code. The byte code then supports strong mobility. It transmits the state of the program in the form of java objects [8]. For this purpose java RMI or any other mechanism (any directory or discovery service according to the requirements) can be used.

The major advantage of this approach is portability. Since there is no modification at the JVM level (underlying architecture), all the changes are made at the source code level which makes this approach portable. Moreover at the source code level, variable types are known which helps saving and restoring them much more easily [7].

The disadvantages associated with the approach are its lack of efficiency. Since additional code is generated and compiled, much more time is required as compared to other approaches. Moreover, this approach modifies the signatures of the methods. Therefore it cannot be used with reflection API and call back methods (like actionPerformed) in which method signatures are critical and should not be changed [4].

2.3.4 Systems supporting strong mobility using source code level instrumentation

Following are the systems that support strong mobility using source code instrumentation technique in Multi-Agent systems.

2.3.4.1- ARA (Agents for Remote Access): ARA is a multi-agent system supporting strong mobility. In ARA, the agent is a program which can move from one point to another during its execution. In case of ARA C/C++ is also compiled into byte codes for RISC virtual machine. In this way, strong mobility is easily accomplished.

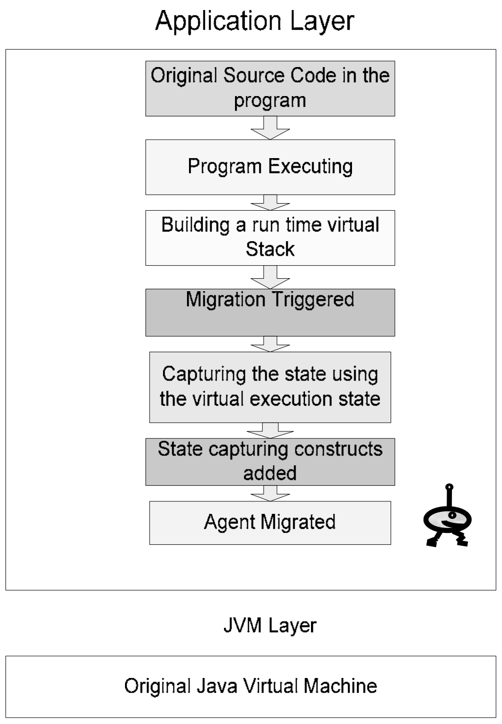

Application Layer
Original Source Code in the program
Program Executing
Building a run time virtual Stack
Migration Triggered
Capturing the state using the virtual execution state
State capturing constructs added
Agent Migrated
JVM Layer
Original Java Virtual Machine

Figure 5: Basic Architecture of Strong mobility using source code level instrumentation

2.3.4.2- D'Agents

D'Agents is an architecture that can support multiple languages including Tcl, Java and scheme. It provides a support for strong mobility. It also supports a system of policies under which mobility support is provided.

2.3.4.3- Telescript

Telescript is an object oriented language very much similar to Java and C++. There is a major focus on strong mobility. Proactive strong agent migration is provided using a 'go' operation. Remote cloning is also supported in telescript.

2.3.4.4- Sumatra

Sumatra is a system based on Java language but it includes certain extensions to it. These extensions help the Sumatra system to support strong mobility. The extensions are made in the Java Virtual Machine, the approach mentioned in the previous section. Agents in Sumatra are java threads. It supports proactive migration and proactive remote cloning.

<u>2.3.4 Strong Mobility using byte code instrumentation approach In MultiAgent Systems</u>

Byte code instrumentation is one of the other approaches used for implementing strong_mobility. This approach is much similar to that of source code instrumentation technique. The major difference is that the modifications required in order to support state saving and state resuming are made at the byte code level instead of the source code layer. In this case the stack frames are saved at the source and then restored at the destination. Goto statements can be used for the implementation of compound statements in case of byte code. This statement is very important in restoring the instruction pointer state. It reduces the additional code size due to its simple implementation.

At the byte code level, in case of stack frame, neither variable types nor their scopes are known. Moreover, due to security reasons, the instrumented byte code has to pass through a byte code verifier. Copying the byte code at the destination can cause various security problems [9].

Java programs are normally available in their byte code format (class files). This makes the byte code level instrumentation technique very useful [4]. Moreover, the additional code overhead is very small as compared to that of the source code level instrumentation.

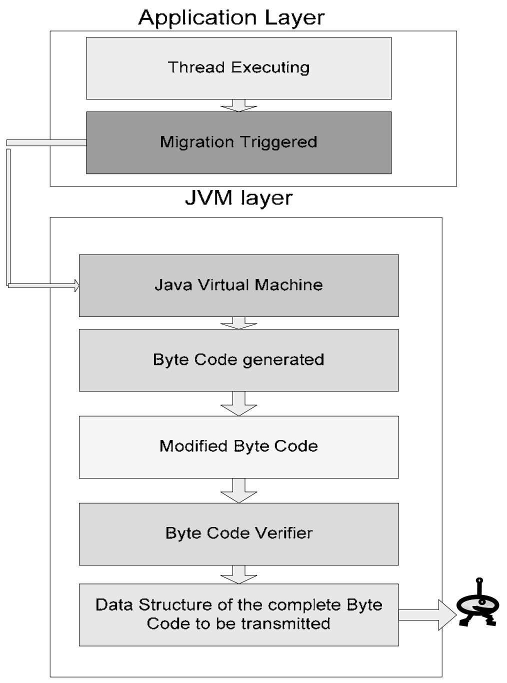


Figure 6: Basic Architecture of Strong mobility using byte code level instrumentation

One of its major disadvantages is that the byte code has to pass through a byte code verifier. Moreover, as mentioned before, at the byte code level, frame variables and operand stack is available which does not give any clue about the type of variables being used. Therefore a separate type system is required that recognizes the types of variables statically [9].

2.3.6 Strong Mobility using Java Platform Debugger Architecture approach In MultiAgent Systems

The last approach for implementing strong mobility is of accessing execution state information from Java Platform Debugger Architecture (JPDA) [4]. JPDA is a part of JVM that's used for debugging of java programs. Since it is used for debugging, it has got a large amount of execution state information in it. The problem is that the direct modifications of the execution state variables are not allowed. Therefore, separate technique should be devised in which the execution state information is allowed to be accessed. One can access Program Counter (PC), Thread information and stack from the JPDA.

The major advantages of JPDA approach is its portability and efficiency. The approach is portable as JPDA is an integral part of every JVM and therefore can be used on every platform supporting Java. Efficiency is achieved as there is almost no code overhead in extracting the execution information from JPDA (excluding the modification overheads).

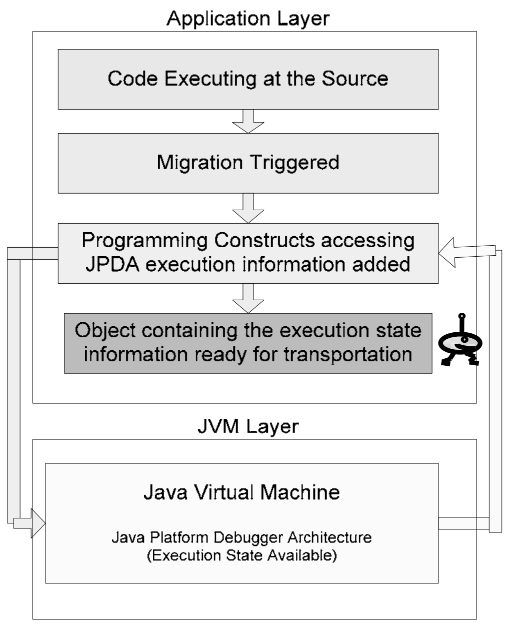

Application Layer
Code Executing at the Source
Migration Triggered
Programming Constructs accessing JPDA execution information added
Object containing the execution state information ready for transportation
JVM Layer
Java Virtual Machine
Java Platform Debugger Architecture (Execution State Available)

Figure 7: Basic Architecture of Strong mobility using Java Platform Debugger Architecture (JPDA)

The major problem with this approach is that the code has to be compiled with debugging option turned on. Moreover, complete strong mobility cannot be achieved. This is because complete state information is not available. Status of the operand stacks is one of the major unavailable components. [4]

2.4 Resource Management in Mobility

Another variation in the case of strong mobility is the management of resources. If an object or an agent is executing by using certain computer resources like files, memory etc and it migrates to a remote site, it will still need those resources in order to continue its execution at the destination. Therefore an important factor in the implementation of strong mobility is resource management. The options for managing these resource bindings are

- The bindings are voided i.e. removed or nullified before migration.
- The bindings are re-established at the destination.
- Some of the resources are moved to the remote site for easy access.
- Resources can also be accessed remotely by staying at the destination site [10].

There are mainly three types of resource bindings that need to be handled.

The first one is the most difficult to cater with. It's the binding by id or identifier [10, 4]. In this case the resource is bound to a uniquely identified resource for which there is no other substitute available. If one has to handle with the migration of this type of binding, then the exact resource must be copied at the destination. There is no other alternative available.

The second type of binding is by value. In this case, the value of the resource is important. One may not need to copy the entire resource at the remote site. Instead, the only thing required is to have the same value of the required resource at the remote site. This type of binding is a bit easier to cater with.

The third type of binding is by type. In this case, resource's identity and value don't matter. The only thing that concerns is the resource type. It should be the same at the remote site. For example, in case the agent is using printer at the source then at the destination in may get bound to another printer as a substitute since the resource type is the same. [10]

<u>2.5 Resource management in different Multi-Agent Systems</u>

The following systems provide the implementation of different types of resource management

2.5.1 IBM Aglets

Aglets, supporting weak mobility model, does not provide the support for resource management. [10]

2.5.2 Mole

The system provides resource management such that all references to the resource are made void before migration.

2.5.3 Obliq

The data space management support is by id. In this case the resources are bound to a single agent. In case the agents move, the resources also move along with the agents.

2.5.4 TACOMA

For data space management, there is a binding for resource movement by 'copy'. In this case the resources are moved in data structures. These data structures are called 'cabinets'.

2.5.5 ARA

Data space management is preserved in ARA. Agents only share system resources which removes the binding problems.

2.5.6 D'Agents

In order to support data space management, a file manager is provided.

2.5.7 Telescript

Data space management is preserved by connecting the resources only to their owners. The associated resources are moved along with the agent to the destination. Other bindings to the resources at the source machine are removed.

2.5.8 Sumatra

The data space management support is by value or by move. Network references are maintained to the resources at the source machine

Chart showing the different characteristics of Multi-Agent Systems with respect to mobility

Name Of the Multi-Agent System	**Weak Mobility**	**Strong Mobility**	**Data Space Management Support**	**Languages Support**	**Format of Agents**	**Resource binding supported**
IBM Aglets	Event driven weak Mobility	No	No	Java	Threads	None
Mole	Supports	No	Supports	Java	Objects acting as Threads	Void references to resources
Obliq	Supports	No	Supports	Obliq	Thread	Binding by identifier
TACOMA	Supports	No	Supports	TCL	UNIX Processes	Binding by copy
ARA	Supports	Source code level instrumentation Support	Supports	C/C++	Any Program	Void references to resources
D'Agents	Supports	Source code level instrumentation Support	Supports	TCL Java Scheme	Object	References maintained by file manager
NOMADS	Supports	JVM Modification approach	Supports	Java C++	Threads	References by identifier
Telescript	Supports	Source code level instrumentation Support	Supports	Telescript	Object	Management by move
Sumatra	Supports	Source code level instrumentation Support	Supports	Java	Thread	Management by move

Conclusion

In this paper, we have classified mobility into two basic types i.e. strong and weak mobility. Techniques for implementing strong mobility have been discussed in detail. We have provided a comparison of all these techniques based on software quality issues like performance and portability. Certain multi-Agent systems implementing mobility have been discussed. They have been further classified based on the technique of mobility that they support. Their properties have been discussed and compared.

References:

[1] Sarmad Sadik, H Farooq Ahmad, Arshad Ali, and Hiroki Suguri: Enhanced inter platform mobility in SAGE Multiagent system

[2] *Tim Walsh, Paddy Nixon, Simon Dobson:* As strong as possible mobility: An Architecture for stateful object migration on the Internet

Department of Computer Science, Trinty college Dublin

[3] *Sara Bouchenak, Daniel Hagimont:*

Picking thread states in the java system

[4] ChowkYuk Thesis

[5] Xiaojin Wang Jason Hallstrom Gerald : Reliability Through Strong Mobility

Baumgartner, Dept. of Computer and Information Science, the Ohio State University

[6] 1- Niranjan Suri1, Jeffrey M. Bradshaw1, 2, Maggie R. Breedy1, Paul T. Groth1, Gregory A. Hill1, and Renia Jeffers2: Strong Mobility and Fine-Grained Resource Control in NOMADS

[7] Sara Bouchenak: Making Java applications mobile or persistent

SIRAC Laboratory (INPG-INRIA-UJF)

[8] Tatsurou Sekiguchi, Hidehiko Masuhara, and Akinori Yonezawa : A simple extension of java language for controllable transparent migration and it's portable implementation

[9] Takahiro Sakamoto, Tatsurou Sekiguchi, and Akinori Yonezawa :Bytecode Transformation for Portable Thread Migration in Java

[10] Understanding Code Mobility

[11] Alfonso Fuggetta, Member, IEEE, Gian Pietro Picco, Member, IEEE,

and Giovanni Vigna, Member, IEEE : Mobile Agents in Distributed Computing

George Cybenko and Bob Gray

Thayer School of Engineering

Dartmouth College

{george.cybenko,robert.gray}@dartmouth.edu

[12] Mobile Agents

Edited by: William T Cockayne, Micheal Zyada

[13] Niranjan Suri, Jeffrey M. Bradshaw, Maggie R. Breedy, Kenneth M. Ford,

Paul T. Groth, Gregory A. Hill, and Raul Saavedra : State Capture and Resource Control for Java: The Design and Implementation of the Aroma Virtual Machine1

[14] Niranjan Suri1, Jeffrey M. Bradshaw1, 2, Maggie R. Breedy1, Paul T. Groth1,

Gregory A. Hill1, and Renia Jeffers2 : Strong Mobility and Fine-Grained Resource Control in NOMADS1

Introducing Strong Mobility in Open Source SAGE MultiAgent System

Misbah Mubarak[1], Sara Sultana[1], Zarrar Khan[1], Hajra Batool Asghar[1], H Farooq Ahmad[2], Fakhra Jabeen[1]

[1]National University of Sciences and Technology (NUST)

[2]Communication 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].

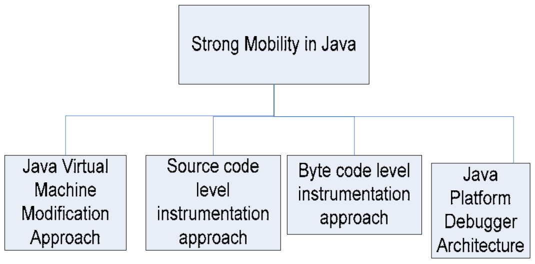
A MultiAgent System (MAS) is a platform in which agents communicate with each other in order to accomplish certain combined goals [6]. Software Fault Tolerant Agent Grooming Environment (SAGE) is an open source MultiAgent system. Foundation for Intelligent Physical Agents (FIPA) is an IEEE standard for 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
Java Virtual Machine Modification Approach
Source code level instrumentation approach
Byte code level instrumentation approach
Java Platform Debugger Architecture

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.

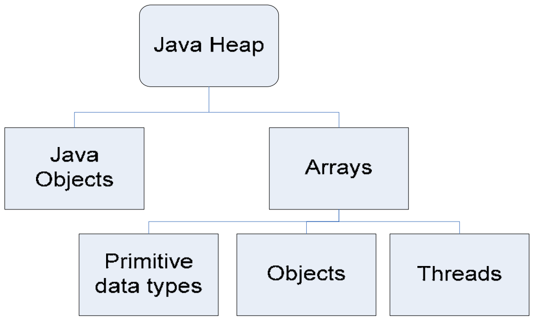


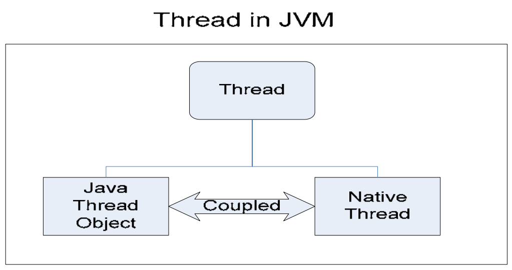
Figure 2: Components of Java Heap

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.



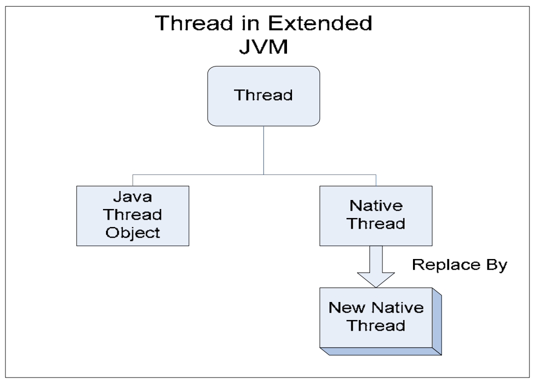


Figure 4: Extent of coupling in JVM and extended JVM

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 retains 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 it's 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.

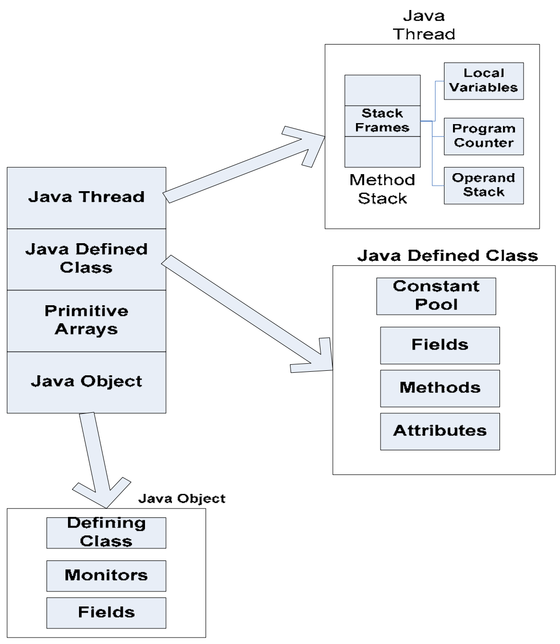


Figure 3: Architecture of the JVM

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.

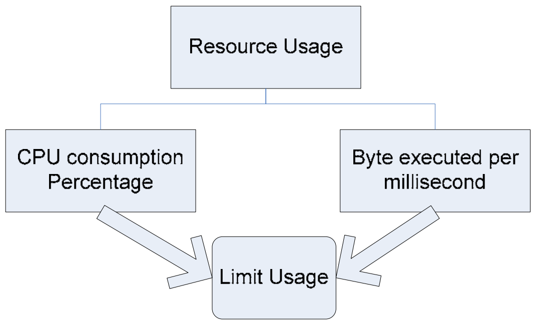


Figure 5: Resource Consumption

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 over view 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 of the local variables and the value of the program counter.

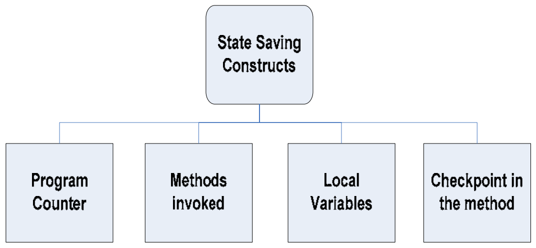


Figure 6: State saving constructs used in preprocessing the code

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. At the source code level, java forbids dealing with the compound statements. Therefore, if one has to deal with compound statements at the source code level, an enormous amount of programming capability is required. At the byte code level, this programming effort is reduced by the fact that we can use a 'goto' instruction. The 'goto' instruction allows transfer to 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.

<u>A comparison of Strong Mobility's approaches</u>

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

Name of the Approach	Efficiency Level	Reliability	Portability	Difficulty of implementation	Licensing constraints	Quality of Mobility Achieved	Source code requirement
Source code modification	Low	Moderate	Available	Moderate	None	Moderate	Yes
Byte code modification	Moderate	Moderate	Available	Difficult to implement	None	Moderately High	No
JVM modification approach	High	High	Not available	Difficult to implement	Sun licensing constraints	High	No

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

[2] S. Bouchenak1, D. Hagimont2: Approaches to capturing Java Thread States. Middleware'2000, poster session, New York, USA, April 2000.

[3] Niranjan Suri, Jeffrey M. Bradshaw, Maggie R. Breedy, Kenneth M. Ford, Paul T. Groth,

Gregory A. Hill, and Raul Saavedra: State Capture and Resource Control for Java:The Design and Implementation of the Aroma Virtual Machine. In proceedings to Java Virtual Machine and research technology Symposium, Monterey, California

[4] Misbah Mubarak, Sara Sultana, Zarrar Khan, Hajra Batool Asghar, H. Farooq Ahmad, Fakhra Jabeen: A review of mobility techniques*. In Proceedings of 19th Assurance Symposium, Tokyo Institute of Technology Japan*

[5] Niranjan 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*

[6] Misbah Mubarak, Sara Sultana, Zarrar Khan, Hajra Batool Asghar, H. Farooq Ahmad, Fakhra Jabeen: An approach to ontological interoperability*. In Proceedings of the 2nd International Conference for Emerging technologies, 2006.*

[7] Takahiro Sakamoto, Tatsurou Sekiguchi, and Akinori Yonezawa: Bytecode Transformation for Portable Thread Migration,*In Java. Proceedings of the Second International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents*, *Department of Information Science, Faculty of Science, University of Tokyo 7-3-1 Hongo, Bunkyo-ku, Tokyo, Japan 113-0033.*

[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

[10] FIPA Abstract Architecture Specification, Standard 2002. www.fipa.org

A Dynamic Policy based Security Architecture for Mobile Agents

Misbah Mubarak[1], Zarrar Khan[1], Sara Sultana[1], Hajra Batool Asghar[1], H Farooq Ahmad[2],

Hiroki Suguri[2], Fakhra Jabeen[1]

[1]National University of Sciences and Technology (NUST)

[2]Communication Technologies, 2-15-28 Omachi Aoba-ku, Sendai, Japan

hajra-mcs@nust.edu.pk, fakhra@niit.edu.pk

***Abstract*— Mobile agents are undoubtedly the upcoming trend in the agent community. As their use is increasing, security problems are coming forth. A mobile agent must address several security issues. An agent may be malicious, the platform on which it executes may carry on sinister activities on it or the agent may be harmed by another malicious agent. Most of the security approaches do not provide any flexibility or dynamic decision making. Therefore, security threats increase further. In our paper, we suggest an approach that uses dynamic ontology based policies to enforce security in platforms as well as in mobile agents. The approach is a hybrid one which utilizes other security measures as well. We have also evaluated the approach on software quality parameters other than security like portability, flexibility and efficiency. The results have been satisfactory.**

Index Terms: **Mobile agents, encryption, digital signatures, ontologies, reconfigurable policies.**

I. INTRODUCTION

Distributed computing has undergone several fundamental changes in the past few years. Initially, the resources of a system were bound to that particular system only. Mobile agents [1] have changed the trend. They offer a lot of flexibility to the distributed computing applications. In mobile agent system, the agent moves from one host to another in order to utilize the services and resources that are available locally at the systems. An agent is an autonomous active entity that acts on behalf of its owner in order to accomplish a certain goal. It performs tasks that are favorable for the users but on the other hand it may also perform malicious tasks depending on its goals.

The advent of mobile agents was hailed as a solution to the ever increasing demand for greater bandwidth requirement. However, technology of mobile agents did not take off as it was foreseen in the beginning solely due to security issues. Although mobile agents greatly increase the efficiency in a distributed system but the risk of security threats becomes far too great. The mobile agent may over utilize system resources, steal important information and use the system as a point of attack to other systems. Security threats due to mobile agents are classified in three basic types

- Confidential information retrieval
- Denial of service attacks
- Corruption of information

The problem specially becomes serious once mobile agents are used for mission critical and real world applications [1]. Another issue that may possibly occur is that the systems the mobile agent accesses may be malicious. It may be possible that the system tries to take important information from the mobile agent or modify the agent by adding or removing its code [2]. Therefore in general four categories of threats for the mobile agents are possible

- Threats from a malicious agent that may harm the system
- A malicious system may attack the agent
- A malicious agent may attack a useful agent
- Any other entity may attack the system

There may be several solutions to the security issues in mobile agents.

- Cryptographic authentication of the owner on behalf of which the agent is acting.
- Security in the execution environment of the languages.
- Policy decisions that are based on the owner's identity by the software that runs the agents.

A policy is a course of action, guiding principle, or procedure considered expedient, prudent, or advantageous. In terms of MultiAgent systems they are declarative rules governing choices in a system's behavior. They are able to restrict system's behavior. They can be used for flexibility, adaptability and security of a system. Policies may be expressed using formal policy languages, rule based policy notations and attribute table representations [3]. They loosely couple the code so that the runtime behavior of applications can be easily adapted according to the requirements. A misconception about policies is that they impose certain constraints on the execution of an application. Policies actually are rules that aid an agent in achieving its social goals resolving security issues. Policies require strong support of middleware. One such middleware is Java Virtual Machine (JVM) in which the high level language policies are compiled into byte codes. In the paper we use Java as the language to express policies and the underlying MultiAgent system.

An advanced variation of policies is dynamically reconfigurable policies. By the use of such policies, an application can adapt according to the requirements at runtime. However, in order to achieve this, the policies need to be semantically rich which is only possible with the help of ontologies. These policies are able to perform the operation without changing the source code of the application thus decreasing security issues.

Mobility can be classified into two basic types i.e. weak mobility and strong mobility. Weak mobility is the movement of code and data from one computational unit to another. Strong mobility is the movement of code, data and execution state from one point to another. Java due to its strong network programming capabilities, offers strong support for achieving mobility. It provides the features of dynamic class loading, serialization and machine independence which are very helpful in achieving mobility [4]. However, due to security reasons, Java does not allow any access to its run time execution state. The reason for this is that a malicious platform may harm the agent severely by modifying the execution state. Therefore, security becomes a big issue once strong mobility is implemented in the case of mobile agents.

With the help of mobility, dynamic customization and configuration of internet applications is also possible. Mobile agents also have the ability to adapt functionality according to user needs with the help of dynamically reconfigurable policies. Therefore, it is vital to deal with the security aspect of mobile agents.

The paper first discusses different possible approaches for implementing security. It then analyzes the draw backs of these approaches and argues how our proposed approach is better than those discussed in Section 2. After that, it gives the basic idea and concept of the proposed approach with the help of detailed modeling in Section 3. It then proposes the implementation of the approach in a strongly mobile FIPA compliant MultiAgent system. Conclusion and future work is given in Section 5

II. RELATED WORK

In this section we discuss different approaches for security and prove that these approaches, when used along with policies provide a better and highly flexible solution to the problem.

We first discuss some approaches that provide security to the platforms against malicious mobile agents.

A. Sand boxing

This approach is used for the protection of the platform from malicious mobile agents. In this case, local code has got access to the critical system resources. However, the remote code or the mobile agent has got restricted access rights. The remote code executes inside a restricted area called ‘Sand Box’. It may affect operations like

- Interacting with local file systems
- Creating a network connection
- Accessing system properties on a local system
- Invoking programs in a local system

The approach is commonly implemented in Java since it has got a class loader, verifier and security manager that enforce security. The problem is that if any of the components fails, it leads to security violation. For instance a remote class may be wrongly classified as a local class. Our system removes this deficiency since it has got a dual authentication mechanism which leads to two pass verification procedure [10].

B. Code Signing

This is another approach that is used to protect platforms against malicious mobile agents. It uses digital signature which verifies that the code of the mobile agent has not been modified since it has been signed by its creator. This method verifies the identity of the owner by checking it against a list that is maintained on the system. However, it only verifies the owner and does not guarantee the safety of the code. Once the mobile agent is verified, it is given full privileges to system resources. The drawbacks of the approach are

- The approach assumes that all the entities mentioned in the list are clean
- The mobile agent is granted complete access to the resources. It may then change the trusted list and invite other malicious agents [11, 12].

Our architecture solves the problem by dynamically deciding about the nature of the mobile agent based on certain parameters. After a close inspection, the architecture allows a certain percentage of access to the system resources depending upon the authenticity. Complete access to system resources is granted in highly exceptional cases.

C. State Appraisal function

The state appraisal function is another approach that protects the platform. It keeps a check on the execution state of the mobile agent. In this case, the author who creates the mobile agent writes a state appraisal function in it. The function is composed of the access rights that the agent has on the platform, depending upon the current state of the agent. Therefore, the sender, instead of the platform, decides the set of permissions that are to be requested. The approach solves the problem in case the platform is malicious. However, if the mobile agent itself is malicious, it may create serious security threats. Moreover, it's very difficult to write an appraisal function every time one sends the mobile agent. Our architecture solves the problem as the dynamic policies are already encoded which may be used by the agent at runtime. Moreover, our architecture provides protection against malicious agents as well [13].

We now discuss approaches that provide security to the mobile agent against malicious platforms.

A. Execution tracing

This is an approach that provides protection to the mobile agent. In this case, the mobile agent keeps a log of the action that it performs at each platform. The log consists of an identifier that identifies all the statements that the agent has executed on any platform. In case, any information is required from an external execution environment, the platform's digital signatures are mandatory. The messages that are attached to the mobile agents consist of unique identifier of the message, identity of the sender platform, time stamp, a finger print and the final state of the mobile agent. If the owner of the agent processes the execution trace of the mobile agent and suspects that a certain platform has been malicious with the mobile agent, he may ask the platform to reproduce the trace. A comparison is made between the finger prints of previous and reproduced trace. This helps in identifying any sinister activity. Disadvantage of the approach is the heavy logs that the agent has to maintain while traveling. Our proposed architecture solves the problem as the mobile agent carries only policies along with it [14].

Since the use of mobile agents poses security threats as mentioned in Section 1, **encryption of mobile agents** is an easy way to get protection against these threats. The mobile agent along with its messages is transformed into cipher text by a function that is parameterized using a public key. The cipher text is then transmitted over the network via Java RMI in the form

$C = E_K(M)$

C refers to the cipher text of the mobile agent

K represents the public key that parameterizes the function

M is the mobile agent to be encrypted

E refers to encryption of the mobile agent

The enemy may copy the entire cipher text. He is, however, unaware of the key. In some of the cases, the intruder may inject fake messages and later send them back or they may break the ciphers by cryptanalysis [8]. Therefore, making an intelligent choice for the key is mandatory. Nevertheless, if the encrypted key is found by the enemy, he may inject malicious code in the mobile agent which can then cause a serious damage to the platform. Therefore, a strong security mechanism is not provided in this case. Our dual layer security architecture solves this problem by enforcing policies on the use of system resources as mentioned in Section 3. Moreover without introducing policies, the entire agent has to be encrypted which requires a lot of computation thus lacking efficiency. Our approach solves the problem by encrypting only the confidential parts of the agent [8].

B. Cooperating agents

It is an approach in which a critical task is carried out by two mobile agents. These mobile agents carry out their operations in entirely separate platforms. There is a secret and confidential communication channel between the two agents. With the help of these channels, the agents transfer their confidential information to each other like

- Agent itinerary
- List of messages sent,
- List of all the platforms that have been recently visited current one and the ones that are still pending to be visited.

In this way if malicious platform changes the route of the agent, the cooperating agent will immediately get to know. It will then notify its sender.

The approach assumes that there are very few platforms that are actually malicious. Moreover, setting up a reliable communication channel between agents is not cost effective. Our proposed architecture poses no such problem as it does not require a separate communication channel. The security architecture software, once installed, protects the platform and all the outgoing mobile agents from the platform.

III. PROPOSED ARCHITECTURE

We propose dual layer security architecture in the use of mobile agents. Policies provide a most promising approach to control access to system resources. However, one may question what guarantee do we have that the policies are correct? Simple policies are enforced automatically without any reasoning. Therefore, if any incorrect policy is enforced, it may lead to security violation. That is the reason due to which we have used ‘Ontology based semantically rich policies’. Ontologies [15, 17], formally specify a concept and the inter relationship between those concepts. They can provide automated reasoning using certain inference procedures. This helps to enforce a meaningful runtime policy after automated reasoning with the help of ontologies. Ontologies also aid in resolving the heterogeneity of platforms. The condition however is, that there should be a single shared ontological representation. For an explanation see [15].

A. Policy Enforcement Architecture

The architecture for policy enforcement consists of the following main components

Policy Storage area: This is the storage area where all the policies and ontologies are stored.

Policy Distributor: This is the module that distributes all the policies to the desired agents.

Policy Implementer: This module maps the policies to its correct action. It uses automated reasoning and inference procedure in order to size up the situation.

Monitor: It monitors the agent execution state and its environment.

Policy specifier: It helps to edit policies and ontologies. The module can only be used by a system administrator.

Figure 1 demonstrates the lay out of security architecture.

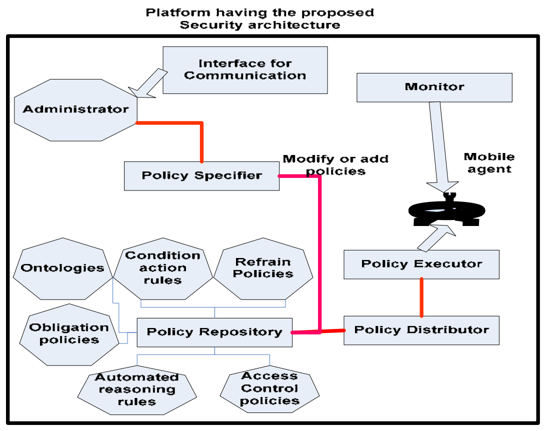


Figure 1: Proposed Security architecture

B. Policies enforced in the architecture

The categories of policies that have been dealt in our architecture are

- Authorization policies
- Obligation policies
- Refrain policies

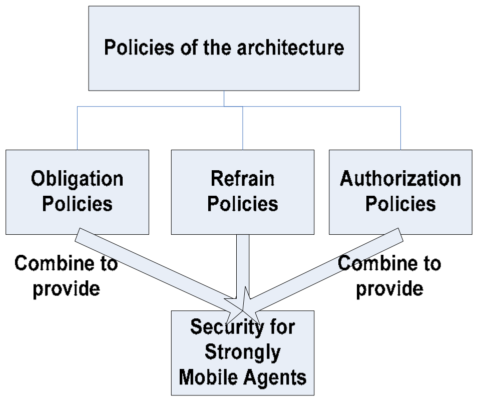


Figure 2: Policies of the architecture

In the end we discuss how all these policies give security to strongly mobile agents as shown in Figure 2.

C. Authorization policies in the architecture

These are access control policies that control the access of the mobile agents to the platform resources. Policies are added in the form of event condition rules in this case. The policies of this category are further divided into

- Encryption policies
- Percentage of access to system resources
- Monitoring the agent in case of complete access
- Monitoring the behavior of the platform on which the agent is executing.

We discuss all the enforcement of the above mentioned policies in detail.

Encryption policies*:* In order to enforce encryption policies, the creator of the agent identifies the part that he/she feels is sensitive. That sensitive part is encrypted using a public key encryption algorithm. This saves a lot of computation effort that is wasted in case the entire agent is encrypted. The creator specifies a certain confidentiality level in integer format which if above a certain threshold (the threshold is also in integer format) will encrypt the agent automatically as shown in Figure 3.

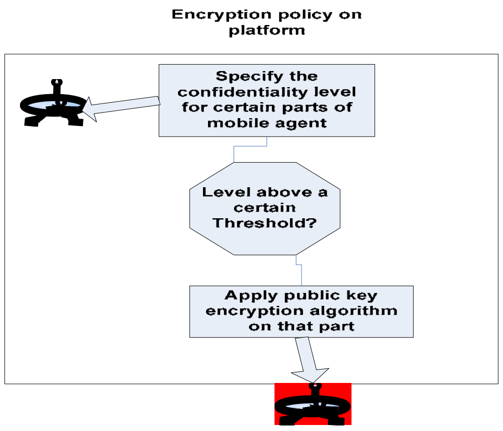
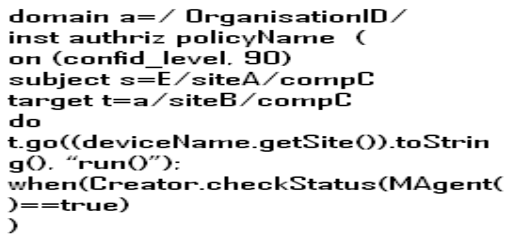

Encryption policy on platform
Specify the confidentiality level for certain parts of mobile agent
Level above a certain Threshold?
Apply public key encryption algorithm on that part

Figure 3: Encryption policy on specific part of the mobile agent

An example of encryption policy for a mobile agent is



```
domain a=/ OrganisationID/
inst authriz policyName  (
on (confid_level, 90)
subject s=E/siteA/compC
target t=a/siteB/compC
do
t.go((deviceName.getSite()).toStrin
g(), "run()");
when(Creator.checkStatus(MAgent(
)==true)
)
```

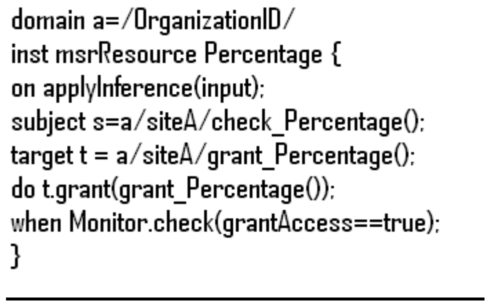
Example 3: Example of an encryption authorization policy

The above example represents an authorization policy which checks the confidentiality level assigned by the creator of the mobile agent. If it is above 90 for example, the subject that is the mobile agent is encrypted. After encryption, it is sent to the destination by the ***'t.go()'*** function.

Policies granting access to system resources*:* There are other policies in the systems that check the characteristics of incoming mobile agents by using ontology based automated reasoning techniques. In other words, the credibility of the mobile agent is deduced based on certain factors like

- Sender identity
- Operation to be performed by the agent
- Check against the list of harmful threats in the code

After making all these checks using automated reasoning and inference procedures, the system decides which percentage of access is to be granted to the mobile agent.

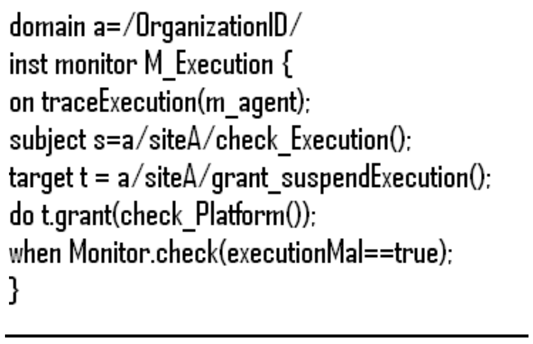


```
domain a=/OrganizationID/
inst msrResource Percentage {
on applyInference(input);
subject s=a/siteA/check_Percentage();
target t = a/siteA/grant_Percentage();
do t.grant(grant_Percentage());
when Monitor.check(grantAccess==true);
}
```

Example 4: Policy specifying percentage of resource usage for the mobile agent

The above policy is an example of one that grants access to system resources based on the dynamic decision about the mobile agent. applyInference(input) is the name of the function that applies the inference procedure using ontologies and automated reasoning on the mobile agent. Check_Percentage( ) is a function that is based on the result of applyInference(input) function. It checks the level of resources that should be granted to the mobile agent. GrantAccess (Boolean flag) is the function that grants access to the system resources.

Policies monitoring agent execution in a platform*:* Monitoring policies are implemented by the 'Monitor' module in the architecture. The purpose of these policies is to dynamically monitor the actions of the agent. It keeps a track of whatever the agent executes. If the agent performs any malicious activity on the resources, it has the right to block the agent and suspend its execution at that very instant.

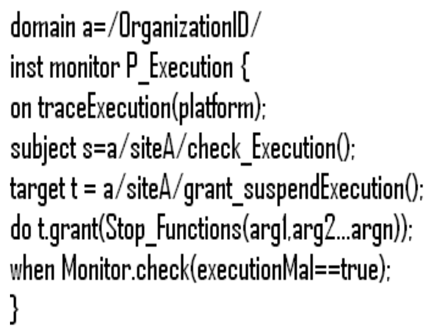


```
domain a=/OrganizationID/
inst monitor M_Execution {
on traceExecution(m_agent);
subject s=a/siteA/check_Execution();
target t = a/siteA/grant_suspendExecution();
do t.grant(check_Platform());
when Monitor.check(executionMal==true);
}
```

Example 5: monitor policy

This policy keeps a track on the execution of the agent by a function traceExecution(m_agent). The argument m_agent specifies the mobile agent. The function check_Execution() checks for malicious activity by comparing the agent activities against a list of malicious activities. Whenever the condition is found to be true, the execution of the agent is suspended using grant_suspendExecution() function.

Policies protecting the agent from malicious activities in the platform: There is another policy that monitors the platform in case if there is some virus program hidden in the system which may harm the execution of the mobile agent. Whenever there is some malicious activity going on the platform, the execution of all the programs is suspended until the malicious program is repaired or quarantined.



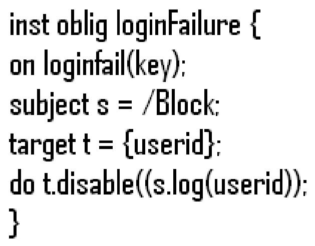
```
domain a=/OrganizationID/
inst monitor P_Execution {
on traceExecution(platform);
subject s=a/siteA/check_Execution();
target t = a/siteA/grant_suspendExecution();
do t.grant(Stop_Functions(arg1,arg2...argn));
when Monitor.check(executionMal==true);
}
```

Example 6: Policy catering with harmful platform

The function 'traceExecution (platform)' keeps a check on the executions that are performed on it. If Monitor.check(Boolean flag) returns true which means there is malicious activity going on the platform, the execution of all programs is suspended.

D. Obligation policies in the architecture

Obligation policies are event condition rules. They specify what actions are to be performed as a result of an event. In the proposed architecture, these policies are stored in the policy repository and whenever a condition meets, the policy is enforced by the policy executor. For example, in case when the creator of the mobile agent tries to access the agent results and execution state in case of strong mobility by decrypting it, if he enters the wrong key, results are blocked.



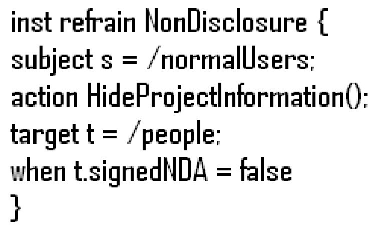
```
inst oblig loginFailure {
on loginfail(key);
subject s = /Block;
target t = {userid};
do t.disable((s.log(userid));
}
```

Example 7: Login policy

The obligation policy 'login failure' blocks access to the mobile agent in case if wrong decryption key is entered. Loginfail(key) is the function which checks if there has been a login failure condition. The user id is blocked and disabled using the functions t.disable((s.log(userid)).

E. Refrain policies in the architecture

Refrain policies allow specifying actions that the user must not perform under certain conditions. These policies control the subject (a mobile agent in our case) to act even if it has got full access to the target [16]. For example, a normal user of a platform may not access the sensitive code, data and execution state information from the mobile agents when refrain policies are enforced. However, the creator of the mobile agent may be able to access it. The verification may be made using digital signatures.



```
inst refrain NonDisclosure {
subject s = /normalUsers;
action HideProjectInformation();
target t = /people;
when t.signedNDA = false
}
```

Example 8: Disclosure of information policy

With the help of this policy, the information is not disclosed to local users as they cannot verify the digital signature process. The function HideProjectInformation() hides the project or the application information whenever the user does not confirm the digital signatures. Test for digital signature verification is made by t.signedNDA function [3].

F. Policies for Strongly mobile agents

In case of strong mobility, the agents carry their execution state along with them as mentioned in Section 1. Though carrying the execution state of an agent increases the efficiency as the overhead of restarting the mobile agent at the destination is eliminated. However, it poses severe security threats since if the execution state of the agent is modified maliciously, it may lead to severe damage to all the platforms that the agent visits. Therefore, the Monitor component of the security architecture monitors the agent execution all the time. Whenever the agent's execution state is tried to be modified by some external source, this module performs a check policy and enforces blockage of that external source. Policies in Example 5 and 6 are instances of strong mobility's security policies. Therefore, the architecture solves the security problems that are posed by strong mobility. See Figure 3. It shows whenever a malicious program tries to access the execution state of an agent, it gets terminated.

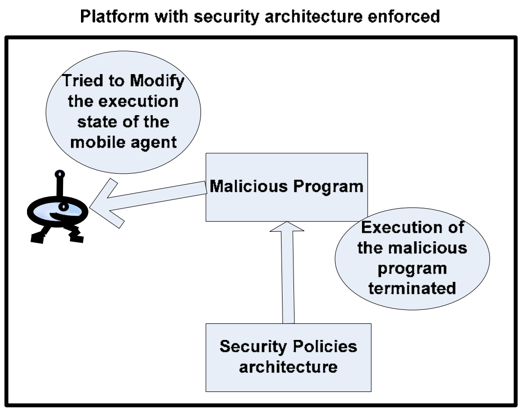

Platform with security architecture enforced
Tried to Modify the execution state of the mobile agent
Malicious Program
Execution of the malicious program terminated
Security Policies architecture

Figure 3: Malicious program trying to execute in security architecture

G. Evaluation of the architecture against quality parameters

We evaluate the software against the following quality parameters

Reliability: The security architecture enforced is hybrid architecture. It employs a manifold technique of security including policies, encryption and digital signature. It deals with all issues which may harm the platform or the mobile agent. At present, there is no security loop hole. However, we are working further on the architecture for the identification of any other security problems.

Efficiency: The architecture imposes such policies which encrypt only the sensitive portion of the agent. This requires less computation and makes the encryption process efficient. The mobile agent does not carry logs along with it which require more bandwidth and flood the network. The architecture is therefore efficient.

Portability: the policy enforcer is a pluggable component which may get plugged into any system independent of the underlying architecture or MultiAgent system.

Security: As proved in the previous section, the architecture implements manifold security mechanism.

IV. CONCLUSION AND FUTURE WORK

In the paper we have proposed a security framework for strongly mobile agents. The security plug in once installed on any platform has the ability to protect all the incoming and outgoing mobile agents. It also makes sure that no malicious activity is carried out on the agent platform. Moreover, the architecture is a hybrid approach which implements policies on the top of other security mechanisms like encryption, digital signature etc. This adds flexibility in the system as the security mechanisms can be utilized using dynamic and reconfigurable system policies. Without these policies, the security mechanisms included a single pass security check. Moreover, the system lacked flexibility as the mechanisms were hard coded. The dynamic policies introduce the element of intelligence and runtime decision making in the system by inference procedures. Currently, we are working on Strong mobility in any FIPA compliant MultiAgent system. In the future we intend to plug in this security architecture in the Strong mobility system.

REFERENCES

[1] "Mobile Agents White Paper", General Magic, 1998.

<URL:http://www.genmagic.com/technology/techwhitepaper.html>

[2] Robert S. Gray and David Kotz and George Cybenko and Daniela Rus: "D'Agents: Security in a multiple-language,mobile-agent system"

[3] Rebecca Montanari, Emil Lupu, Cesare

Stefanelli: "Policy based dynamic reconfiguration of mobile code"

[4] Misbah Mubarak, Sara Sultana, Zarrar Khan, Hajra Batool Asghar, H Farooq Ahmad, Fakhra Jabeen: "A review of mobility techniques". *In proceedings of 19th Assurance Systems Symposium, Tokyo Institute of Technology, Japan*

[5] Wolfgang Nejdl, Daniel Olmedilla, Marianne Winslett, and Charles C. Zhang: "Ontology-Based Policy Specification and Management", Research Center and University of Hannover, Germany

[6] Todd Papaioannou : "On the structuring of distributed systems, the argument for mobility", doctoral thesis

[7] Wayne Jansen, Tom Karygiannis

National Institute of Standards and Technology: NIST Special Publication 800-19 – "Mobile Agent Security"

[8] Andrew S Tenanbaum: "Computer networks", 4th edition, vrije universiteit Amsterdam, Netherlands

[9] "Fast software encryption": 4th International workshop, FSE, Haifa, Israel

[10] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based

fault isolation," In Proceedings of the 14th ACM Symposium on Operating Systems

Principles, pages 203--216, Dec. 1993.

[11] "Signed Code," (n.d.). Retrieved December 15, 2003, from James Madison

University, IT Technical Services Web site: http://www.jmu.edu/computing/infosecurity/

engineering/issues/signedcode.shtml

[12] “Introduction to Code Signing,” (n.d.). Retrieved December 15, 2003, from Microsoft

Corporation, Microsoft Developer Network (MSDN) Web site: http://msdn.microsoft

.com/library/default.asp?url=/workshop/security/authcode/intro_authenticode.asp

[13] W. M. Farmer, J. D. Guttman, and V. Swarup, “Security for mobile agents:Authentication and state appraisal,” In Proceedings of the European Symposium on Research in Computer Security (ESORICS), pages 118--130, Sep. 1996.

[14] H. K. Tan and L. Moreau, “Extending Execution Tracing for Mobile Code Security,” In K. Fischer and D. Hutter (Eds.), Proceedings of Second International Workshop on Security of Mobile MultiAgent Systems (SEMAS'2002), pages 51-59, Bologna, Italy.2002.

[15] Misbah Mubarak, Sara Sultana, Zarrar Khan, Hajra Batool Asghar, H Farooq Ahmad, Fakhra Jabeen: “An approach to ontological interoperability”, In proceedings of 2nd IEEE international conference on emerging technologies. (ICET 2006)

[16] John A. Knottenbelt: A report on Policies for Agent Systems, Imperial College of Science, Technology and Medicine

[17] M Andrea Rodriguez : “Similarity based ontology integration”. In Proceedings of the 1st International Conference on Geographic Information Science, October 2000

GeBTA: Architecture for Strongly Mobile Agents

Misbah Mubarak[1], Zarrar Khan, Sara Sultana, Hajra Batool Asghar[1], H Farooq Ahmad[2],

Hiroki Suguri[2], Fakhra Jabeen[1]

[1]National University of Sciences and Technology (NUST)

[2]Communication Technologies, 2-15-28 Omachi Aoba-ku, Sendai, Japan

hajra-mcs@nust.edu.pk, fakhra@niit.edu.pk

Abstract **— Strong mobility is the movement of code, data and execution state of distributed entities from one computational node to another. General issues related to strong mobility are inefficiency and increased code overhead. Our main concern is the optimization of strong mobility so that it can lead to improved performance and reliability. In the paper we argue that separation of concerns when applied to strong mobility leads to better results. Contribution of this paper is an architecture named GeBTA (Generic Byte Code Transformation Architecture) that provides efficiency and thereby reduces code overheads. This helps in achieving reliability and fault tolerance. We have implemented the architecture using Java. Although java is a strong network programming language, it does not allow access to the execution state of a thread. Therefore, achieving strong mobility in a way that software quality parameters like portability, efficiency and reliability are also preserved becomes a challenging task. The paper proposes a generic plug-in system GeBTA for strong mobility that can be**

integrated with any FIPA compliant MultiAgent system. The system can also run independent of any MultiAgent system. We have evaluated it by checking its performance in different scenarios. The results have proved that it consumes less time and memory as compared to many other available systems.

Index terms: **Byte code transformation, Java Virtual Machine (JVM), MultiAgent System (MAS), Strong Mobility.**

V. INTRODUCTION

With the growth of Internet as a primary environment for communication and development of distributed applications, there is a strong need to change the old design paradigms and to switch onto the new ones. A global network such as Internet must exploit different forms of mobility in order to increase its usability and scalability requirements. Mobile agents [1] can be used to satisfy the scalability needs of the highly dynamic global network. Mobile agents are autonomous software entities with the capability of dynamically changing their execution environments in a network aware manner [2, 3]. Mobile agent technology is being promoted as an emerging paradigm that helps in the design and implementation of more robust and flexible agents. Mobile agents are preferable over other design paradigms due to certain features that other paradigms do not provide. These features include disconnected operation, reduced bandwidth, reduced latency, increased stability and increased sever flexibility [4, 5].

In case of code mobility, distributed applications move mobile agents while they are executing. When these mobile agents acquire the capability of resuming their execution instead of restarting at the destination, they are called strongly mobile agents [2]. Therefore, Strong mobility is the movement of code, data and execution state of an agent whereas weak mobility only moves the code and data of the agent. Code of a program consists of programs whose methods are being executed by the thread. Data consists of the values of local variables and registers. Thus in case of weak mobility, execution of the agent or the object restarts on the destination.

Java has accelerated the use of transportable code over the internet [6]. It provides many impressive features for distributed computing like serialization, dynamic class loading and machine independence which help in the movement of code and data. Therefore, most of the mobile agent systems are built using Java. Mobile agents that are built on the top of java only provide weak mobility because java does not allow access to the execution stack of a program. Execution state of a program comprises of a java stack which is organized in frames. Whenever a method is invoked, a frame is pushed onto the stack and is popped when the method returns. A frame may consist of

- Local variables of the method
- Operand stack consisting partial results of the method
- Program Counter (PC)

Although java is currently gaining popularity in distributed application, it lacks many features which are a major requirement in today's distributed computing. There are a large number of MultiAgent systems that provide agent services of different quality and maturity. Majority mobile agent systems are java based which do not provide the support for strong mobility. Such systems include IBM Aglets [7], Mole, Voyager [8], Software Fault tolerant Agent Grooming Environment (SAGE) [9], Java Agent Development Framework (JADE) [10] and many other systems that are further discussed in Section 2.

In most of the above mentioned MultiAgent systems (MASs), the mobility service that is provided to the agents is specific to that particular MAS only. In some cases, the mobility service is not available for heterogeneous architectures. Separation of concerns therefore becomes important in mobility services. In most of the cases, the code for mobility is strongly coupled and entangled to an extent that it is very difficult to utilize that code in some other MAS [11]. The motivation for our byte code transformation project was to design and implement a system that is not dependent on any particular MultiAgent system. We have designed generic byte code transformation architecture for achieving Strong Mobility. The architecture acts as a generic plug-in which can be used with any MAS or independent of any MAS. Guidance and help in the implementation of the architecture has been taken from [12].

Many of the systems that support strong mobility cannot be used in time critical applications due to performance issues. Due to the excessive delay in state saving, their use is also difficult in load balancing. On account of these issues we have designed an architecture that can function in general and time critical applications. At the same time, the architecture should provide efficient load balancing. For that, it is necessary to reduce the code overhead imposed by the transformation processes. Efficiency is also an important parameter in this regard. We have performed several experiments to determine the code overhead, efficiency and memory consumption of the architecture given in Section 9. We have then evaluated our architecture under specific scenarios in order to check its usability in different situations as mentioned in Section 10.

Rest of the paper is organized as follows: Section 2 gives the related work. Section 3 explains the major features in our architecture. Section 4 discusses general strong mobility procedure and behaviors. Section 5 gives a general structure of our architecture. Section 6 explains the rewriting process in byte code transformation. Section 7 provides a description of the state saving and resuming constructs. Section 8 demonstrates the architecture using an example. Section 11 concludes the work.

VI. RELATED WORK

In this section we discuss various approaches for implementing strong mobility. We then analyze the MultiAgent Systems (MASs) that implement mobility in accordance to these approaches.

A. Weak Mobility

In weak mobility, all the execution state of the agent is lost once it migrates to the destination node. The programmer has to restart the agent at the destination. Weak mobility is often achieved using event based programming i.e. the code and data gets transferred when some specific event or condition is satisfied [7,8]. Weakly mobile agents are efficient as compared to strongly mobile ones because there is no code overhead introduced due to state saving and resumption. Moreover, the MASs that support weak mobility are built on the top of java without modifying any of its internal architecture. This preserves portability. Therefore weak mobility has the added advantage of efficiency and portability [2,7].

Systems that implement weak mobility consist of IBM Aglets, Mole, Obliq and Tacoma to name a few. Agents are represented as either threads or processes.

There are basically two approaches for implementing strong mobility. First is to use a modified Virtual Machine (VM) and the other is to change the compilation model.

B. Strong Mobility through VM modification

The JVM consists of all the execution state information that is required to migrate a thread or a process. Therefore strong mobility can be achieved if the JVM is modified in a way that the execution state is accessible to the user. In order to achieve this, security constraints imposed by the JVM must be broken. Sun imposes licensing constraints that forbid the distribution of modified JVM [13]. The advantage of this approach is its efficiency as no additional code is generated in achieving mobility. However, the disadvantage is that in modifying JVM, portability is compromised as modification violates the standardization of JVM. Ara [14, 15] and Sumatra [16] are MASs that use the modified JVM approach for achieving strong mobility. Another problem associated with these systems is that they use JDK 1.0.2 VM that does not make use of native threads.

NOMADS [13] is a system that uses the modified VM approach and resolves many of the problems that exist with systems like Sumatra and Ara. It is composed of two basic parts, an agent execution environment called Oasis and a custom VM called Aroma. Using these two components, NOMADS introduces features like strong mobility and safe agent execution. Safe agent execution is the ability of the agents to control resource usage and to protect themselves against malicious attacks like denial of service. However, the Aroma VM adopts the strategy of mapping Java threads to native threads for execution which has the draw back of complexity and inefficiency. Since the feature for state capturing of native threads is not available, the Aroma VM waits until the java threads finish executing the native code and starts state capturing after that. Another problem associated with the approach is increased memory usage. The Oasis runs each agent in a separate instance of the Aroma VM which causes increased resource usage and memory utilization.

C. Strong Mobility through changing the compilation model

The compilation model approach can be further divided into two parts i.e. Source code instrumentation technique and byte code instrumentation technique.

In the source code modification scheme [17], state saving and resuming constructs are added at the source code level. This is called 'preprocessing' the code. The preprocessed code is then compiled and as a result the generated byte code supports strong mobility. Major disadvantage of the source code instrumentation scheme is the code overhead introduced due to the state saving constructs. This also increases the bandwidth usage during migration. In this case, migration can not take place when the object is in its constructor since it is not possible to transfer an object that has not been constructed yet. Moreover, in some of the methods an additional state saving parameter is added to the source code which modifies the method signature. Due to this reason, methods like actionPerformed cannot use this migration technique. Systems like WASP and JavaGo support source code level instrumentation technique [18]. As we will show in Sections 3 and 4, our architecture does not introduce these migration constraints.

Another variation of the compilation model technique is byte code modification [17,19]. This scheme inserts the state saving and resuming constructs at the byte code of the program. This strategy does introduce some code overheads but they are little as compared to source code instrumentation scheme (as the compilation overhead is eliminated). Another advantage of this scheme is that instructions like 'Goto' that are quite useful in state saving and resuming are only available at the byte code level. A major issue that one faces while using the byte code instrumentation technique is of type recognition. At the byte code level it is quite difficult to know the variables types. However, the problem gets resolved if the design and implementation are thoroughly done.

VII. MAJOR FEATURES OF GeBTA

Some of the major features of our system are

- Our architecture performs state saving and resumption at the byte code level.
- The subject of mobility in our system is a thread. Thread migrates from one computational node to another. In our system, we have captured the execution state of a thread. Whenever 'yield' method is called with argument 'true', the computation of the current thread stops. The state of the current thread is then saved and other threads start executing. The saved thread is then transported to the destination where it continues execution from the same point where it left at the source.
- The object or thread can be transported during the call to the <init> method or the constructor which is not possible in many other approaches.
- Complete instructions at the byte code level are handled in a way that no exceptions are raised.
- The machine dependent types are converted into a 'Virtual Type'. Different machines and operating systems have different byte allocations for data types. This is done in for covering up difference in type representations.
- We have used a byte code transformation approach, therefore presence of source code at either side (source or destination is not necessary).
- Strong mobility is triggered dynamically during the execution of the thread whenever the argument of yield method becomes 'true'.

VIII. TRANSPARENT THREAD MIGRATION

Transparent thread migration occurs by adapting the following sequence

At the source machine: At the source machine, whenever the yield method is called, it in turn calls the transformer which in turn calculates and saves the execution state of the thread in the form of state saving byte code instructions in a machine independent data structure called 'Instructions_List'. This data structure is then written into the class file of the thread by the transformer.

On the network: The rewritten class file is then transferred over the network via RMI to the destination machine.

At the destination: At the destination, a new thread object is created. The execution state of the new thread is then retrieved from the 'Instructions_List' data structure.

IX. GENERAL STRUCTURE OF GeBTA

Our architecture provides transparent thread migration using byte code transformation approach. It is divided into two basic components. One is byte code analyzer and the other is translator. First basic task of the byte code analyzer is to extract byte code attributes like Line number table, fields, methods, classes, inner classes, exception table and other byte code attributes using reflection. The second task is to convert byte code instructions into portable java format in the 'Instructions' package. These portable instructions are then used by the transformer during state saving and resumption. In this way every executed instruction can be traced.

The problem of type recognition mentioned in Section 2 is resolved by the byte code instructions. The instructions at the byte code level are typed for example, ILOAD refers to loading an integer from a local variable onto stack. Similarly LLOAD refers to loading a long value from a local variable into the operand stack [20]. Figure 2 gives some of the byte code level instructions that we have handled.

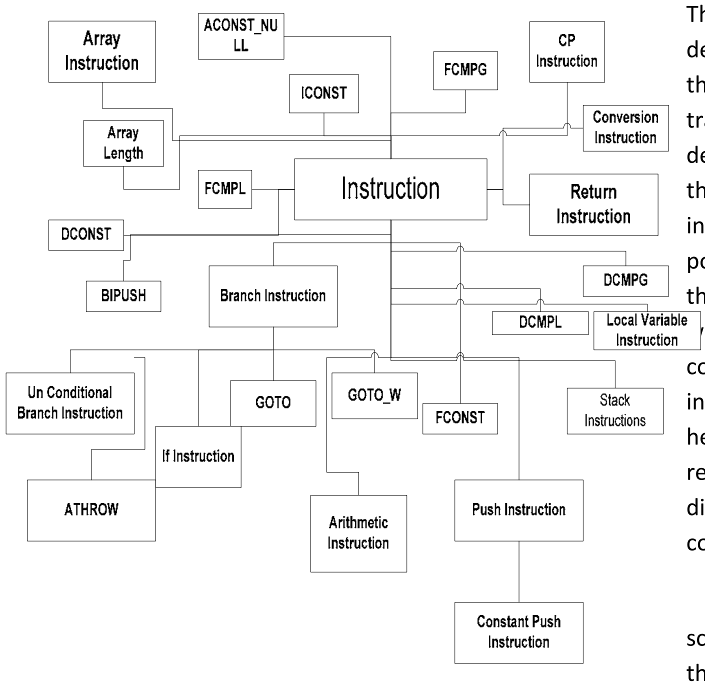


Figure 2: Byte code level instructions

Whenever the yield method is called for a thread, a call to the Transformer is made. Task of 'transformer' is to suspend the execution of the current thread and to call the rewriters that in turn instantiate objects that perform stack saving using portable byte code instructions (that have been mentioned in context of analyzer component). The state saving (at the source) and resuming (at destination) instructions for the stack are then written to the byte code. The transformed byte code is then transmitted via serialization to the destination. At the destination, running the transformed byte code restores the state of the program by executing the state restoring instructions. It starts execution from exactly the same point where it left execution at the source machine. In this way strong mobility can be easily accomplished without any additional compilation overheads. Thus the component modifies the byte code of the class file and inserts state saving and establishing constructs. With the help of these constructs, the application can easily resume its computation at the destination even at difficult points (In between a loop or during the construction of an object).

Another component in the architecture is the scheduler. Its task is to schedule the threads according to their priorities. It accepts and determines computations of the threads and embeds them in a runnable interface. It then starts the threads one by one. It can also restart the threads in case they get rescheduled. It also provides special handling for monitor entry and exit.

Figure 3 explains the complete procedure of byte code transformation.

Two basic methods isSwitching and isRestoring determine the state of the thread. Whenever the transformer is called, context of the thread is saved and isSwitching becomes true. At the destination, while restoring the state of the thread isRestoring becomes true.

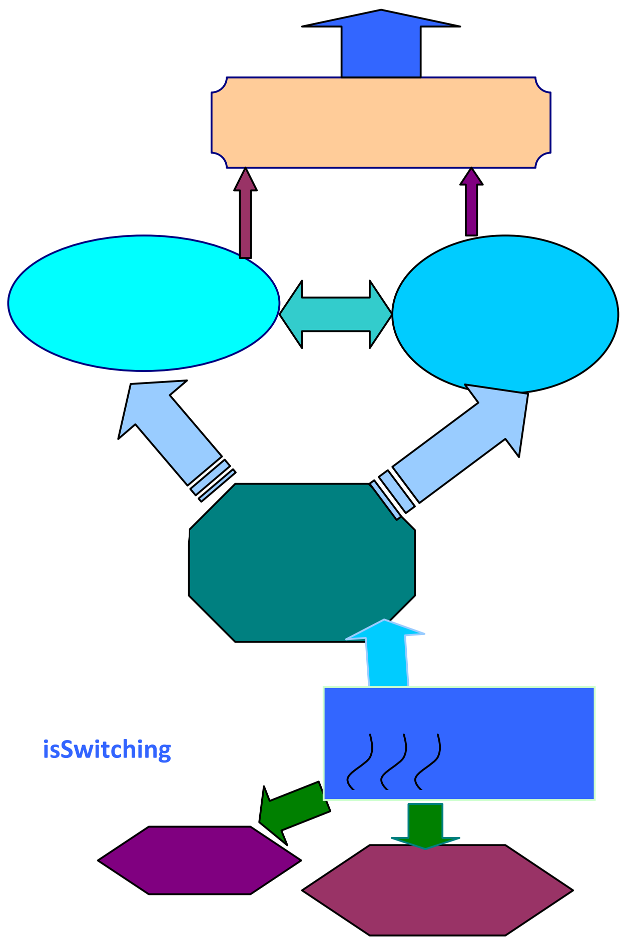


Figure 3: Transforming the byte code

Varying type representation problem occurs while dealing with different data types. Different systems allocate different bytes to represent the data types. This creates problems in migration between heterogeneous architecture. In order to resolve the issue, the architecture converts the data types (primitive data types and reference types) into virtual types. The stack therefore stores and transmits the data in the form of virtual types. This eliminates the problem that occurs due to varying size of data types in heterogeneous architectures.

In our system, we have defined all the primitive types and reference types. For the uninitialized values, our architecture gives an automatic warning that the values are not reliable. When state saving is triggered, these primitive and reference data types are converted into virtual types in a way that the type information is preserved. The advantage associated with having a well defined typed system is easy passage through the byte code verifier. If the typed system is specific, the transformed byte code passes easily through the byte code verifier.

The virtual java stack that is saved in our architecture consists of an array of virtual types. It accepts operations like push, pop, peek, merge, size, elements etc. With the help of these operations, one can easily analyze the stack elements.

Our architecture provides a chain of three rewriters that perform the rewriting process. Functions of these rewriters are:

Stripper Rewriter: This rewriter removes the local variables and line numbers of a particular method so that they can get updated by the subsequent rewriters.

Aux-rewriter: This rewriter checks for methods that have an 'InvokeSpecial' instruction. Invoke special instruction deals with the invocation of instance methods. It provides a special handling for super classes, private classes and instance initialization methods [15]. This rewriter traverses through all the instructions that are present in the 'Instruction_List' data structure associated with every method. It keeps on updating those instructions according to their execution. Whenever the aux-rewriter encounters this 'InvokeSpecial' instruction, it obtains the constant pool that consists of all the variables and their type information. After getting the constant pool, it updates all the variables of the method and inserts them in the 'Instructions_List' data structure. This rewriter also updates the targets of all Goto instructions.

Main rewriter: This rewriter basically deals with all types of invoke instructions like 'invoke static', 'invoke interface' and 'invoke virtual'. While dealing with these 'invoke instructions' in a method, the rewriter saves all the instructions that get executed in context of these invoke instructions. It saves the variables that are associated with execution of these instructions. It keeps on updating the values on the stack. In the end, it saves the stack, local variables and the program counter in the machine independent data structure 'Instructions_List'. For a structured working of the rewriters and their operation see Figure 4.

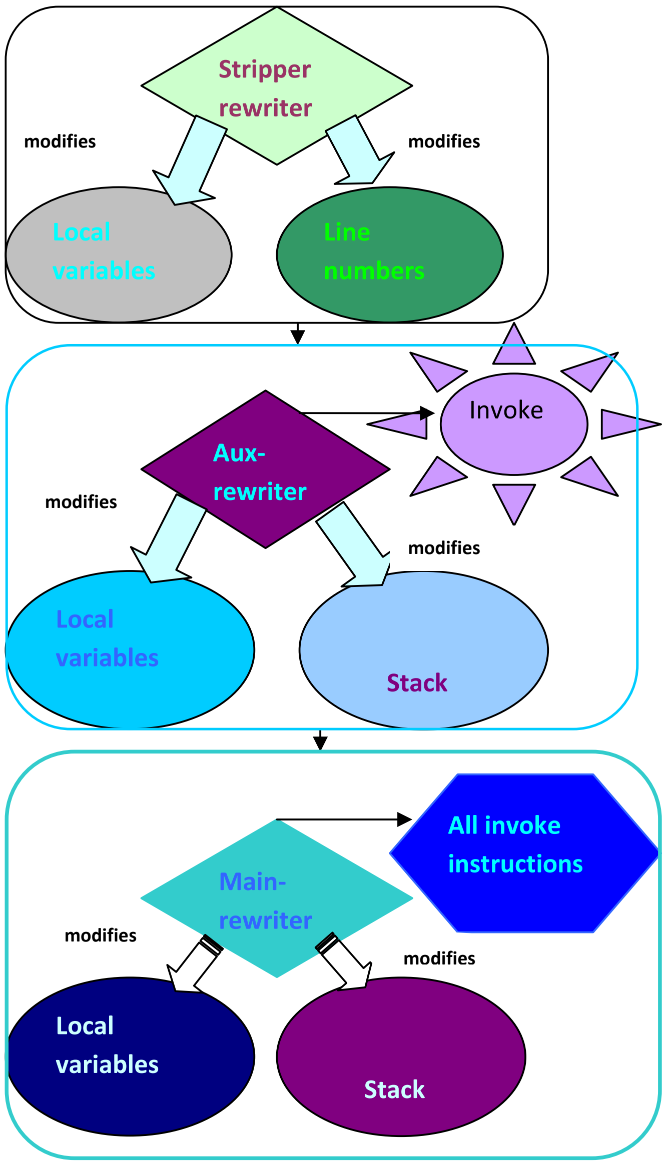
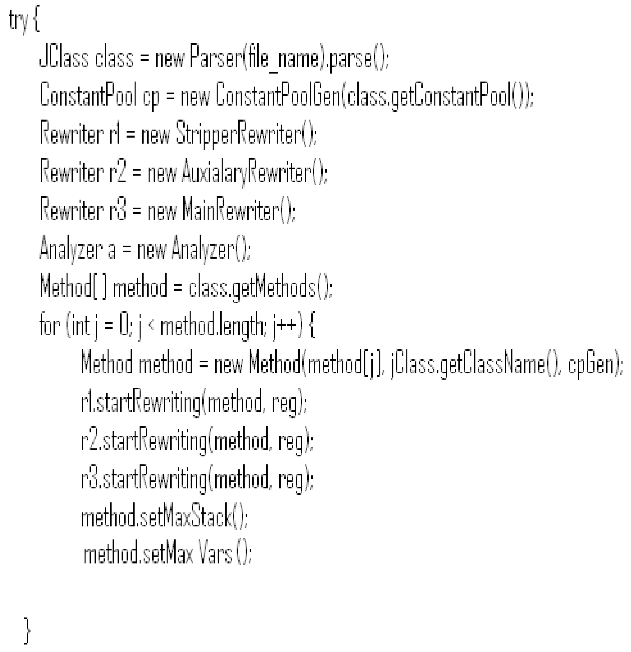


Figure 4: Working of rewriters

X. STATE STORING CLASSES

The transformer is the component that triggers mobility by invoking all the rewriters and saving context of the methods.

In Figure 5, the transformer class takes every method and rewrites the methods according to the procedure explained in Section 6. Concept of transformer class and its respective rewriters has been taken from [13].

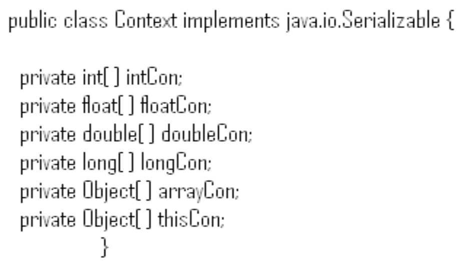


```
try {
    JClass class = new Parser(file_name).parse();
    ConstantPool cp = new ConstantPoolGen(class.getConstantPool());
    Rewriter r1 = new StripperRewriter();
    Rewriter r2 = new AuxialaryRewriter();
    Rewriter r3 = new MainRewriter();
    Analyzer a = new Analyzer();
    Method[ ] method = class.getMethods();
    for (int j = 0; j < method.length; j++) {
        Method method = new Method(method[j], jClass.getClassName(), cpGen);
        r1.startRewriting(method, reg);
        r2.startRewriting(method, reg);
        r3.startRewriting(method, reg);
        method.setMaxStack();
        method.setMax Vars ();

 }
```

Figure 5: Transformer function

There is a context class associated with every method. The task of this context class is to store the values of the variables that belong to different data types like long, double, float, array etc. In this way, the types will be easily converted into virtual types. See Figure 6 for the implementation of the class.



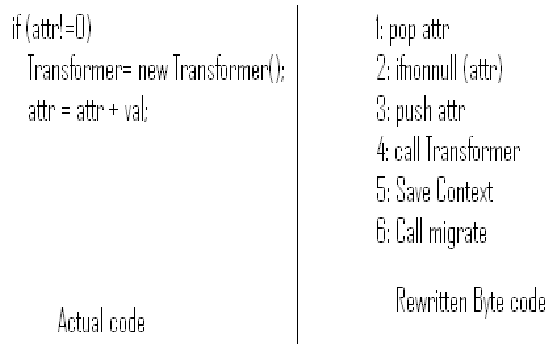
```
public class Context implements java.io.Serializable {

  private int[ ] intCon;
  private float[ ] floatCon;
  private double[ ] doubleCon;
  private long[ ] longCon;
  private Object[ ] arrayCon;
  private Object[ ] thisCon;
          }
```

Figure 6: Context class for threads

XI. EVALUATION OF GeBTA

We here discuss an example that demonstrates the working of the architecture explicitly. We have tested the transformation process on several algorithms. Here we give the original as well as the transformed code of a method in a computational algorithm FindS. In the middle of the computation of FindS, strong mobility is triggered. The state is then stored in a machine independent data structure 'Instructions_List'.



```
if (attr!=0)                              1: pop attr
   Transformer= new Transformer();        2: ifnonnull (attr)
   attr = attr + val;                     3: push attr
                                          4: call Transformer
                                          5: Save Context
                                          6: Call migrate

                                             Rewritten Byte code
      Actual code
```

Figure 7: Actual and rewritten byte code

In Figure 7, the actual code checks if the value of the variable 'attr' is not equal to zero. The byte code corresponding to it is then called which first pops the stack for the value 'attr' and then checks if it is equal to null. The next instruction in the actual code is the invocation of 'Transformer'. When transformer is invoked in the actual code, the rewritten byte code pushes the attribute on the stack and then calls the method 'transformer'. The context is then saved. After that migration gets takes place. The format of byte code in Figure 7 is modified for clarity.

This state saving and resumption often leads to a problem i.e. which method should be transformed? There are two approaches that are usually adopted by the programmers [19]. First approach uses programmer's choice in which the programmer specifies which methods to transform. The second approach deals with the construction of a complete call graph for method calls. In this approach, every class that is directly or indirectly related to migration is transformed. In our byte code transformation approach, we are using the second scheme in which every class related to migration get transformed.

XII. EXPERIMENTAL RESULTS

We have evaluated our architecture by performing certain experiments. The major parameters for these experiments are efficiency and memory consumption. These parameters in turn determine the code overhead imposed by the transformation process.

A. Total time consumed during transformation

Table 1 shows the time consumed by various computational algorithms when tested on the architecture. The calculated time is less as compared to [19]. The architecture given in [19] gives a similar byte code transformation technique.

TABLE 1:

TIME CONSUMED BY VARIOUS ALGORITHMS

Algorithm	No. of methods	Analyzer time (ms)	Transformer time (ms)	Total time consumed (ms)
FindS	4	800	550	1350
Quick Sort	1	287	112	399
Fibonacci series	1	227	99	326
Bubble sort	1	272	89	361

B. Increase in size of code

For checking the increase in size of code in our transformation process, we have compared our architecture with Java Go and the byte code transformation architecture given in [19]. The results of the comparison are given in Table 2. The mathematical figures for Java Go and Byte code transformation architecture are taken from [19].

TABLE 2

CODE OVERHEAD INTRODUCED BY GeBTA

	Byte code size (bytes)			
Algorithm	Original	Ours	Java Go	Byte code transformation [14]
Quick Sort	383	1100 (2.8 times)	1177 (3.2 times)	1253 (3.2 times)
Fibonacci Series	276	850 (3.1 times)	884 (3.2 times)	891 (3.2 times)
FindS	780	1300 (1.6 times)	1326 (1.7 times)	1430 (1.8 times)

C. Comparison in terms of recursive methods

We have evaluated our system on Fibonacci series to provide a comparison for time consumed during recursive calls to methods. We have compared our system with Java Go architecture. The results are shown in Figure 8.

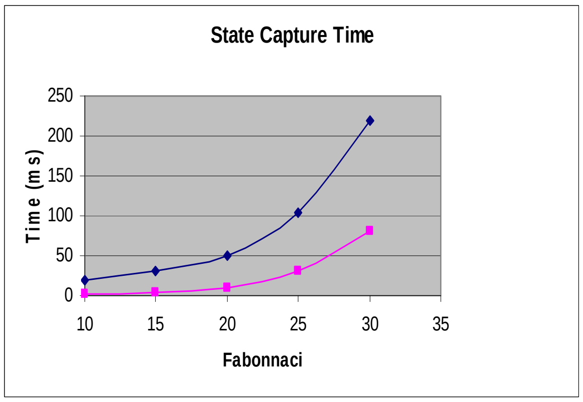


Figure 8: Comparison in terms of recursion calls

The above results help us achieve our aim i.e. to use strong mobility in efficient load balancing and time critical application. As shown in the results, the overhead introduced by our architecture is quite small as compared to other systems. We are currently in a process of optimizing the code so that the elapsed time for mobility also decreases.

D. Test for usability of GeBTA

The architecture can easily be integrated with any MAS. It can also operate independently without any MAS due to the separation of concerns in the code. It can be used as a general purpose system since it does not have any specific code dependencies (or any other dependencies). Therefore, it acts as a generic plug-in. It is also very useful in load balancing applications. A further enhancement of the architecture can also be incorporated in time critical and fault tolerant applications. Suppose a mission critical application is running on a system and accidentally the power of the system fails. The architecture can then transfer the complete application along with the execution state to another system. The application can then continue to run on another system from the same point where it left execution on the source machine. This increases fault tolerance. We have also evaluated this aspect of the architecture by making it work in such a scenario. The additional requirements are

a) To add a condition that detects power failure.

b) Give the destination IP address of the system to which the application should get transferred via RMI.

Therefore, the architecture can be instrumented easily according to user specific needs by adding certain conditions.

XIII. CONCLUSION AND FUTURE WORK

We have designed and implemented a generic architecture for strong mobility with the intention that the code overhead and efficiency issues are resolved. The architecture can be used with or without MAS. We have evaluated our architecture on several algorithms. We have also compared our architecture with other architectures. The results have been satisfactory. Apart from that, in order to demonstrate the general usability of the architecture, we have evaluated it in a scenario that involves power failure. We have used a byte code instrumentation approach for implementing the architecture. The architecture is based on pure java byte code and it instruments them accordingly. It does not take help from any existing software. In the future, we intend to further optimize the architecture's code in order to increase the utilization of the architecture in time critical mobile agents or applications. As far as security issues are concerned, our architecture currently does deal with security but it can be integrated with other security techniques. In this way it will guarantee safe execution of agents, a feature that is also present in NOMADS

References

[1] Takahiro Sakamoto, "Mobile Agents White Paper", NIST special Publication 800-19 Mobile Agents Security, General Magic, 1998.

Available: http://www.genmagic.com/technology/techwhitepaper.html

[2] Misbah Mubarak: "A review of mobility techniques", In proceedings of 19th Assurance Systems Symposium, Tokyo Institute of Technology, Japan, 2006

[3] Giacomo Cabri, Letizia Leonardi & Franco Zambonelli: "Weak and Strong Mobility in Mobile Agent Applications", Università di Modena e Reggio Emilia Via Campi, Italy

<URL: http://polaris.ing.unimo.it/>

[4] Lorenzo Bettini: "linguistic constructs for Object-oriented mobile code programming & their implementations", PhD thesis, Dip. di Matematica, Universit a di Siena, 2003

Available:http://citeseer.ist.psu.edu/bettini03linguistic.html

[5] Todd Papaioannou : "On the structuring of distributed systems, the argument for mobility", PhD thesis, Loughborough University, 2000

[6] Geoff A. Cohen, Jeffrey S. Chase, David L. Kaminsky: "Automatic Program Transformation with JOIE", USENIX annual technical conference, 1998.

[7] "Lange, D. B., & Oshima", M: Programming and Deploying Java Mobile Agents withAglets. Reading, MA: Addison-Wesley.

[8] ObjectSpace. ObjectSpace Voyager

<URL http://www.objectspace.com/products/voyager>

[9] Misbah Mubarak, Sara Sultana, Zarrar Khan, Hajra Batool Asghar, H Farooq Ahmad, Fakhra Jabeen: "Strong Mobility in Open Source SAGE MultiAgent System", In proceedings of 1st International Conference on Open Sources Technology, 2006

[10] "Fabio Bellifemine" (TILAB, formerly CSELT) : JADE PROGRAMMER'S GUIDE, Giovanni Rimassa (University of Parma), Last Updated: February 2003

[11]Noury M. N. Bouraqadi-Saˆadani, Thomas Ledoux and Mario S¨udholt: "A Reflective Infrastructure for Coarse-Grained Strong Mobility and its Tool-Based

Implementation", International Workshop on Experiences with reflective systems (held in conjunction with Reflection 2001, the ph``3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns''), 2001

[12] Geoff Cohen, Olaf Georlitz: An implementation of Strong Mobility using byte code transformation (Help has been taken in terms of architecture working and evaluation), Duke University, 2003

[13] Niranjan Suri: "Strong Mobility and Fine-Grained Resource Control in NOMADS", In Proceedings of the Second International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents, 2000

[14] "Peine, H., & Stolpmann, T".: The architecture of the Ara platform for mobile agents. In KRothernel & R. Popescu-Zeletin (Ed.), Proceedings of the First International Workshop

on Mobile Agents, Springer-Verlag.

[15] Maurer, J. Porting: "The Java runtime system to the Ara platform for mobile agents". Diploma Thesis, University of Kaiserslautern.

[16] Acharya, A., Ragnganathan, M., & Saltz, J. "Sumatra: A language for resource-aware mobile programs". In J. Vitek & C. Tschudin (Ed.), Mobile Object Systems. Springer-Verlag, 1997

[17] S. Bouchenak, D. Hagimont: "Approaches to Capturing Java Threads State", In proceedings of Middleware 2000.

[18] Stefan F¨unfrocken "Transparent Migration of Java-Based Mobile Agents." In *Mobile Agents*,pages 26–37, 1998.

[19] Takahiro Sakamoto, Tatsurou Sekiguchi, and Akinori Yonezawa: "Bytecode Transformation for Portable Thread Migration in Java", Proceedings of the Second International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents, 2000

[20] Tim Lindholm & Frank Yellin, "VM Specifications", 2nd edition, Sun Microsystems, California, US