Proceedings of the 5th WSEAS Int. Conf. on SIMULATION, MODELING AND OPTIMIZATION, Corfu, Greece, August 17-19, 2005 (pp31-36)

Migration Transparency in a Mobile Agent Based Computational Grid

RAFAEL FERNANDES LOPES and FRANCISCO JOSÉ DA SILVA E SILVA
Departamento de Informática
Universidade Federal do Maranhão, UFMA
Av dos Portugueses, s/n, Campus do Bacanga, CCET
CEP 65085-580, São Luís - MA
BRAZIL
{rafaelf,fssilva}@deinf.ufma.br

Abstract: - In recent years, Grid computing has emerged as a promising alternative to increase the capacity of processing and storage, through integration and sharing of multi-institutional resources.
MAG (Mobile Agents for Grid Computing Environments)is a research project that explores the mobile agent paradigm as a way to overcome the design and implementation challenges of developing a Grid middleware. MAG executes Grid applications by dynamically loading the application code into a mobile agent. The MAG agent can be dynamically reallocated to Grid nodes though a transparent migration mechanism, as a way to provide load balancing and support for non-dedicated nodes. This paper describes MAG transparent migration mechanism implementation and performance evaluation.

Key-Words: - distributed computing, grid computing, mobile agents, agent systems, migration, transparent mobility

1 Introduction

In recent years, Grid computing has emerged as a promising alternative to increase the capacity of processing and storage, through integration and sharing of multi-institutional resources, such as software, data and peripherals, stimulating the cooperation among users and organizations.

Through a Grid infrastructure it is possible to execute a large number of applications, such as: distributed super-computing applications (e.g. simulation of complex physical processes like climatic modeling), high throughput (e.g. cryptographic problems resolution), on demand applications (e.g. medical instrumentation applications and requests for software use), data intensive applications (e.g. weather forecast systems) and collaborative applications (e.g. collaborative and educational projects).

MAG (*Mobile Agents Technology for Grid Computing Environments*) is a Grid middleware currently being developed at the Computer Science Department of the Federal University of Maranhão (DEINF/UFMA). The project main goal is the development of a free software infrastructure based on mobile agents technology that allows the resolution of computationally intensive problems in computer Grids.

Constructing a Grid middleware is a complex task. Developers must address several design and implementation challenges, such as: efficient management of distributed resources, dynamic task scheduling, high scalability, fault-tolerance, heterogeneity, configurability and the presence of efficient mechanisms for collaborative communication among Grid nodes.

In order to overcome some of the above challenges, the Grid infrastructure must be able to move computations among Grid nodes. Process migration offers several advantages for distributed systems that can be explored in the context of a Grid middleware, such as: [12]

- **Load sharing:** by moving objects around the system, one can take advantage of lightly used processors. Thus, the system can move computations from overloaded machines to idle machines, making a better use of existing resources;
- **Communication performance:** active objects that interact intensively can be moved to the same node to reduce the communications cost for the duration of their interaction. This also happens if a computation needs to manipulate a great data volume: moving computations to the data may be less costly than moving data to the computations;
- **Availability:** the migration mechanism can be used for capturing and re-establishment of the computation execution state. Thus, in case of a failure, the computation can be restarted in a new place;
- **Reconfiguration:** if a machine becomes unavailable (e.g. its user does not want to share its machine any more) the computations executing in that node can be moved to other locations;
- **Utilizing special capabilities:** an object can move to take advantage of unique hardware or software capabilities on a particular machine.

Despite of its benefits, moving a process to a different machine is a costly task. Thus, the use of process migration must be adopted only when there are good reasons for doing so, e.g., improve overall system performance.

This paper presents the process migration mechanism provided by the MAG/Brakes framework. It is organized in the following sections: section 2 presents an overview of the MAG project. Section 3 shows the problems and solutions related to process migration in the context of a Grid middleware, while Section 4 describes MAG/Brakes, the implementation of the MAG process migration mechanism. In section 5 we evaluate the performance of our mechanism in respect to the imposed execution time and space overhead, comparing the results with other approaches described in the literature. Section 6 discuss related works and Section 7 shows the conclusions obtained from the work performed, and describes its next steps.

2 MAG Overview

MAG is a Grid middleware that explores the use of mobile agents as a way to overcome several of the Grid design and implementation challenges, cited in the Section 1. MAG architecture is organized in layers as shown on Figure 1.

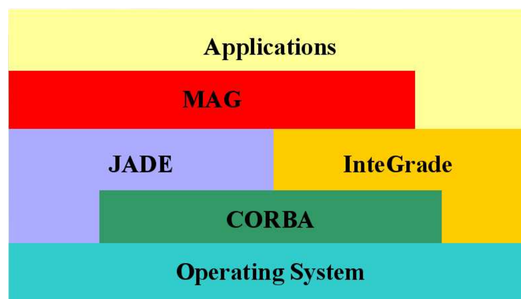

Fig. 1: Layers architecture of the MAG / InteGrade Grid middleware

MAG uses the Integrade Grid middleware [8] as the foundation of its implementation, reusing several Grid services instated of implementing the Grid infrastructure from scratch. Integrade offers an application repository, components for resource management, a task scheduler and tools for submitting applications and collecting the execution results.

The JADE layer (Java Agent Development Framework) is a framework used for building multi agent systems. It was totally coded in Java and provides functions such as communication facilities, life span, and monitoring of the mobile agents execution, following the FIPA[1] specification.

MAG architecture was projected to be a natural extension of the InteGrade middleware, allowing the execution of native applications as well as applications written in Java. Native applications are executed directly above Integrade. Among them, there are parallel applications that follow the BSP model. The MAG layer supplies an application execution mechanism through mobile agents and adds new services to the Integrade platform, such as the use of mobile computing devices as Grid clients, fault tolerance, and migration of application executions among Grid nodes.

The upper layers use the CORBA distributed objects technology and many services provided by this middleware such as the trading service. At last, the operating system layer may be variable, since MAG is platform independent.

3 Process Migration

Migration is not a new concept. Many systems have been developed providing this ability like Charlotte[2], Sprite[5] and Emerald[12]. However, with the advent of the Java technology, the research work on this area has been focused on the creation of an efficient mechanism for migration of Java threads. This was stimulated by the fact that the majority of the mobile agent systems have been developed using the Java language.

To migrate a process, some state information must be saved and shipped to the new locations. At the target destination the process is restarted based on shipped state. However, to migrate a process, is necessary to know what exactly comprises the process and its state.

Fuggetta, in 1998, has defined that a process consists of three basic segments: the code segment, the resource segment and the execution segment[6]. *Code segment* is the part that contains the set of instructions that make up the program that is being executed. The *resource segment* contains references to external resources needed by the process, such as files, printers, devices, other processes, and so on. Finally, an *execution segment* is used to store the current execution state of a process, consisting of private data, the stack, and the program counter.

A migration mechanism that migrates these three segments, and restart a process in exactly the same state and at the same code position as it was before the migration, is called *transparent* or characterized as *strong migration*[7]. The strong migration provides to process the abstraction that the execution was not interrupted. Strong mobility is very powerful, but much harder to implement.

In contrast, the *non-transparent* migration (known as *weak migration*) can be defined as every migration that is not strong[4]. In the weak migration only the code segment is transferred, and perhaps some initialization data. A characteristic feature of weak mobility is that a transferred program is always started from its initial state.

Strong mobility simplifies the task of the programmer, therefore it does not have to implement the migration process explicitly - the process state is saved automatically. This approach distinguishes from the weak, where the programmer has to create the code responsible for save and reestablish the complete state of the process.

[1] *Foundation for Intelligent Physical Agents* - non-profit organization aimed at producing standards for the interoperation of heterogeneous software agents. Available in: *http://www.fipa.org/*

Proceedings of the 5th WSEAS Int. Conf. on SIMULATION, MODELING AND OPTIMIZATION, Corfu, Greece, August 17-19, 2005 (pp31-36)

3.1 Migration Transparency in Java

Java has been put forward as the preferred language for the developing of mobile agents platforms. This choice is explained by the fact that agents must be able to run on heterogeneous platforms. Java's machine independent bytecode is responsible for this. Moreover, the ease of transport over the net of Java bytecode, the Sun's serialization mechanism (that allows the migration of objects' data) and the security concepts provided by the Java platform, are some of other features that makes Java a good choice for the developing of mobile applications.

However, the use of Java introduces some problems for process migration. Java does not provide sufficient mechanisms for capturing the computation state. The Java serialization mechanism only allows the saving of code and values of object members. Java classes cannot access intern and native information of the Java Virtual Machine (e.g. program counter, call stack, etc.), necessary to save the full execution state[10] of a Java thread.

During the last years, several techniques were developed that artificially gathers the information required to fully capture the state of Java threads. They can be classified in four basic approaches [10, 4, 11, 7]:

- **Modification of the Virtual Machine:** the virtual machine is modified to export execution information. The major disadvantage of this approach is the lack of portability in respect to the standard VM. It is very difficult to maintain complete compatibility with the Sun Java specification;

- **Instrumentation of the application source code:** this technique consists in the use of a preprocessor (a source code compiler). This preprocessor inserts source code responsible for capturing and restoring the execution state. The main problem with this technique is that the source code must be available, which is not always possible, such as the case when libraries are used. Another disadvantage with this technique is the time and space overhead caused by the insertion of code;

- **Instrumentation of the application bytecode:** in this approach, the code for capturing and restoring the execution state is inserted directly in the application bytecode. As well as the source code instrumentation approach, some time and space overhead is generated by the insertion of code in the original code. However, the overhead is generally lower than the source code approach. Another advantage is that, in the bytecode level, we have access to an extended set of instructions, such as the goto instruction;

- **Modification of the Java Platform Debugger Architecture:** the Java Platform Debugger Architecture (JPDA) is part of the virtual machine specification. Using JPDA, runtime information about applications can be accessed in debug mode. This can be used to perform transparent migration. Since the JPDA does not provide all information necessary for transparent migration, it is necessary some modifications in the JPDA core. Moreover, this approach does not allow the use of JIT compilation, raising the application execution time, generating the much higher overhead of all approaches.

4 Migration in MAG

The design of the MAG process migration mechanism took into consideration the benefits and disadvantages of each approach for Java thread migration described on Section3.1 in respect to Grid middleware requirements. For example, Grid systems must transpose several administrative domains. Imposing a standard JVM for several Institutions as required by the JVM modification approach is not well suited for Grid systems. Grid systems should be able to execute scientific applications that are developed using shared libraries, whose source code are not always available. This limits the applicability of the source code instrumentation approach for Grid systems. A good application execution performance is essential in Grid systems, the major disadvantage of the JPDA modification approach.

Given the above considerations, we decided to implement the MAG migration mechanism based on the instrumentation of application bytecode. The mechanism is based on a modified version of the Brakes[16] framework, developed at the Katholieke Universiteit Leuven, Belgium, by the Distributed systems and computer Networks (DistriNet) research group.

Brakes is a framework that provides a thread serialization mechanism for Java. It was developed at the Katholieke Universiteit Leuven, Belgium, by the Distributed systems and computer Networks (DistriNet) research group. It allows the capture and re-establishment of Java threads execution state through the bytecode instrumentation approach. This feature is basic for the development of many mechanisms in distributed systems like transparent migration, fault tolerance and object persistence.

Currently, Brakes consists of two parts:

- A ByteCode transformer (based on version 1.4 of the ByteCode Engineering Library[2]) which instruments Java class-files, so they are able to capture their current internal state at any given time;

- A small framework which uses the ability of the "patched" classes to allow Java threads to pause and resume whenever desirable.

The standard version of the Brakes framework is called "Brakes-serial". This version does not allow the concurrent execution of threads. This feature becomes this version inadequate for use in the MAG, given that several applications can be running in each Grid node.

[2] Available in *http://bcel.sourceforge.net/*

Proceedings of the 5th WSEAS Int. Conf. on SIMULATION, MODELING AND OPTIMIZATION, Corfu, Greece, August 17-19, 2005 (pp31-36)

The Brakes has another version called "Brakes-parallel", that allows the execution of concurrent threads. This version was developed as a proof-of-concept system, without any optimizations for real-world use. It is an unoptimized add-on of the first prototype, and has a much higher overhead. This higher overhead is caused by a set of new components that was added to the serial version.

4.1 MAG/Brakes

The MAG/Brakes is a subset of the original Brakes. Given the limitations in Brakes, we decided to use, in MAG, only its bytecode transformer. Thus, the Brakes components (e.g. a thread scheduler) were discarded in our implementation.

As was said in the previous section, the "Brakes-serial" cannot be used in the MAG, because of the limitation of execute only one thread per time. Thus, we had to use the "Brakes-parallel" as the basis of our implementation. Thus, when we say Brakes, we are talking about the "Brakes-parallel" version.

We have changed the structure of Brakes to obtain a better performance and to provide new functionalities:

- The Brakes framework is composed by several components used to store execution information and execution states, and to manage the application execution, e.g. start, suspend, resume, stop, etc. The use of these components was causing a large overhead, because of the great number of accesses to these objects in the heap. We modified the original transformer in order to store the state information inside each object, as attribute members, reducing the access cost to data necessary for the migration process;
- Our framework makes possible to the migration process be initiated by an external source (e.g. another thread). It is very interesting in the MAG context, given that MAG agents must be able to migrate applications over the Grid;
- As a new feature, the user can now explicitly inform to the bytecode transformer, a code position where he wants that the application state have to be saved. This is done by the user through a `doCheckpoint` method invocation;
- The default implementation of Brakes inserts code to save the application state after each method invocation. Now, the user can inform to the transformer that ONLY the invocations to the `doCheckpoint` method have to save the application's state, ignoring the other invocations.

4.1.1 MAG/Brakes Implementation

In the MAG/Brakes implementation, each task being executed in a Grid node has some boolean flags associated with it. This flags represents three different modes of execution:

- *Running:* the task is executing normally;
- *Switching:* the task is in the process of capturing its current execution state into its context;
- *Restoring:* the task is in the process of reestablishing its previous execution state from its context;

These flags and the context of the thread are members of a class called `MagApplication`, which must be inherited by all MAG applications.

After the compilation, the MAG/Brakes transformer inserts code into the application bytecode. The bytecode transformer inserts a state capturing block after every method invocation. The code tests if the flag `switching` is set. In the positive case, the capture of context is done. This code can be seen in the figure 2.

For each method invocation, we save the corresponding stack frame (the method local variables and the operand stack). An artificial program counter is also stored in the tasks context. This is a cardinal index that refers to LPI-index (each invocation has an unique LPI-index associated).

```
//-------------------------------------
int m;
(...)
fibonacci (m);   // <- method invocation
if(this.isSwitching())  {
 <store stack frame into myContext>

 <store artificial program counter>
 return;
} (...)
//-------------------------------------
```

Fig. 2: State capturing

When an agent wants to capture the state of a thread, it invokes (in the thread object) the method `doYield()`. The method `doYield()` sets the `switching` flag to true. The modification of this flag causes the capturing of the thread state, done by the code inserted by the MAG/Brakes transformer.

The MAG/Brakes also provides the complementary operation: the reestablishment of execution state. The transformer inserts additional code in the beginning of each invoked method body. This inserted code consists of several state reestablishing code blocks, one for each code position where the previous execution of the method may have been suspended. This code block is shown in the figure 3.

In order to resume the execution of an application, the agent invokes the method `doResume()` that sets the `restoring` flag to true. The computation is then restored and continued transparently.

Proceedings of the 5th WSEAS Int. Conf. on SIMULATION, MODELING AND OPTIMIZATION, Corfu, Greece, August 17-19, 2005 (pp31-36)

```
//-------------------------------------
// start of the method body
public int fibonacci (int i) {
 if(this.isRestoring())  {
   // if this is the last stack
   // frame in the context
   if (this.lastStackInContext()) {
     switching = false;
     restoring = false;
   }
   <get the program counter from myContext>
   <restore the stack frame>
   <go to the correct instruction>
   }
} (...) } (...)
//-------------------------------------
```

Fig. 3: State reestablishment

In the actual MAG/Brakes implementation, the resource segment is not migrated. The external environment of application is not taken into consideration and the connections to devices, like printers, files and others, are not migrated in a transparent way. Only the migration of single-threaded applications can be done.

5 Performance Evaluation

Instrumenting and inserting code introduces time and space overhead. Since code is inserted for each method invocation, the space overhead is directly proportional to the number of invocations that occur in the code[16]. We performed several tests to measure the overhead imposed by our implementation in comparison to normal execution of a recursive `fibonacci` application. We have chosen this algorithm because of the great number of invocations that it performs. Thus, we can consider the recursive `fibonacci` as our "worst case" analysis.

All tests were performed on a Sempron 2200+ overclocked to 1.8 GHz, 256 MB of RAM, running Linux (kernel 2.6.10). We measured two aspects:

1. The time execution overhead incurred by the checkpoint mechanism in comparison with the "normal" (non-instrumented) execution, the Brakes-serial, and Brakes-parallel;

2. The class file space overhead, in comparison to the non-instrumented one.

Figure 1 shows a graph that compares the execution times for the same application (Fibonacci) given an input of 20, 25, 30, and 35. We can notice that the time overhead caused by MAG/Brakes is only marginally higher than Brakes-serial. The difference is due the fact that Brakes-serial uses static invocations, which are faster than conventional invocations to objects. However, this limits Brakes-serial to manipulate only a single (static) reference to the thread being executed.

Mechanism	Fib(20)	Fib(25)	Fib(30)	Fib(35)
Normal	25.87	31.80	158.60	1493.80
Brakes-serial	31.90	41.07	207.40	1984.40
MAG/Brakes	33.03	42.37	226.23	2245.13
Brakes-parallel	50.00	274.13	2695.13	29523.87

Table 1: Execution time for a recursive Fibonacci (in ms)

We also compared the file size penalty caused by each mechanism. Table 2 shows the values (in KB) of class files generated by each approach.

Mechanism	File size	Overhead
Normal	1.3	-
Brakes-serial	1.7	30,77 %
MAG/Brakes	1.9	46,15 %
Brakes-parallel	1.7	30,77 %

Table 2: Comparison of space overhead caused by each mechanism

Based on the tests performed, our conclusion is that the time and space overhead caused by the MAG/Brakes, despite of being a little higher than the one caused by Brakes-serial, is justified by the advantage of being able to execute threads concurrently. This feature is extremely important in the context of the MAG Grid infrastructure.

6 Related Work

There are many systems that provide mechanisms for capture and reestablish the execution state of Java threads. In the following, we cite some projects, associating them with the implementation approach adopted.

WASP[7] and JavaGo[14] support the capture of execution state through source code instrumentation. In the Grid, many times it will be necessary to execute applications where the source code are not available, making this alternative inappropriate. JavaThread[3], D'Agents[9], Sumatra[1], Merpati[15] and ARA[13] depend on extensions of the standard VM from Sun. This approach is also not attractive for Grid systems considering the great heterogeneity of Grid environments. The CIA[11] project uses the JPDA API (Java Platform Debugger Architecture) leading to great overhead in execution time, caused by the absence of JIT (Just-in-Time) compilation.

7 Conclusions and Future Work

In this paper, we described a strong migration mechanism in the context of MAG, a mobile agent based Grid middleware. We have detailed the requirements involved with the construction of this mechanism and the problems associated with the capture and reestablishment of Java threads

state. We have also presented the MAG/Brakes, a framework based on Brakes that provides the strong migration feature in MAG.

In the nearby future, we intend to overcome the limitations of the actual MAG/Brakes version, providing the migration of multi-threaded applications and the migration of the resources (printers, files, etc.) associated with the threads.

References

[1] A. Acharya, M. Ranganathan, and J. Saltz. Sumatra: A Language for Resource-aware Mobile Programs. In *Mobile Object Systems: Towards the Programmable Internet*, volume 1222, pages 111–130. Springer-Verlag: Heidelberg, Germany, 1997.

[2] Y. Artsy and R. A. Finkel. Designing a process migration facility: The charlotte experience. *IEEE Computer*, 22(9):47–56, 1989.

[3] S. Bouchenak, D. Hagimont, S. Krakowiak, and N. D. Palma. Experiences implementing efficient java thread serialization, mobility and persistence. In *Software - Practice and Experience*, volume 34, pages 355–394, 2004.

[4] A. J. Chakravarti, X. Wang, J. O. Hallstrom, and G. Baumgartner. Implementation of strong mobility for multi-threaded agents in java. In *2003 International Conference on Parallel Processing (ICPP '03). IEEE Computer Society Press.*, pages 321–330, Koahsiung, Taiwan, 6-9 October 2003.

[5] F. Douglis and J. K. Ousterhout. Transparent process migration: Design alternatives and the sprite implementation. *Software - Practice and Experience*, 21(8):757–785, 1991.

[6] A. Fuggetta, G. P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, 1998.

[7] S. Funfrocken. Transparent Migration of Java-Based Mobile Agents: Capturing and Reestablishing the State of Java Programs. In *Proc. of the Second International Workshop on Mobile Agents*, pages 26–37, Stuttgart, Germany, September 1998.

[8] A. Goldchleger, F. Kon, A. Goldman, M. Finger, and G. C. Bezerra. Integrade: Object-oriented grid middleware leveraging idle computing power of desktop machines. *Concurrency and Computation: Practice & Experience. Vol. 16, pp. 449-459*, 2004.

[9] R. S. Gray, G. Cybenko, D. Kotz, R. A. Peterson, and D. Rus. D'agents: Applications and performance of a mobile-agent system. In *Software - Practice and Experience*, number 32(6), May 2002.

[10] T. Illmann, F. Kargl, T. Krueger, and M. Weber. Migration in Java: problems, classifications and solutions. In *MAMA'2000*, Wollongong, Australia, 2000.

[11] T. Illmann, T. Krueger, F. Kargl, and M. Weber. Transparent migration of mobile agents using the Java Platform Debugger Architecture. In *Lecture Notes in Computer Science*, volume 2240, page 198, Jan 2001.

[12] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[13] H. Peine and T. Stolpmann. The architecture of the Ara platform for mobile agents. In R. Popescu-Zeletin and K. Rothermel, editors, *First International Workshop on Mobile Agents MA'97*, volume 1219 of *Lecture Notes in Computer Science*, pages 50–61, Berlin, Germany, Apr. 1997. Springer Verlag.

[14] T. Sekiguchi, H. Masuhara, and A. Yonezawa. A Simple Extension of Java Language for Controllable Transparent Migration and Its Portable Implementation. In *Coordination Models and Languages*, pages 211–226, 1999.

[15] T. Suezawa. Persistent execution state of a java virtual machine. In *Proc. of the ACM 2000 conference on Java Grande*, 2000.

[16] E. Truyen, B. Robben, B. Vanhaute, T. Coninx, W. Joosen, and P. Verbaeten. Portable support for transparent thread migration in java. In *ASA/MA*, 2000.