Portable Support for Transparent Thread Migration in Java

Eddy Truyen*, Bert Robben, Bart Vanhaute, Tim Coninx, Wouter Joosen and Pierre Verbaeten

Departement Computerwetenschappen, K.U.Leuven
Celestijnenlaan 200A, B-3001 Heverlee, Belgium
{eddy, bartvh}@cs.kuleuven.ac.be
http://www.cs.kuleuven.ac.be/~xenoops/CORRELATE/

Abstract. In this paper, we present a mechanism to capture and reestablish the state of Java threads. We achieve this by extracting a thread's execution state from the application code that is executing in this thread. This thread serialization mechanism is implemented by instrumenting the original application code at the byte code level, without modifying the Java Virtual Machine. We describe this thread serialization technique in the context of middleware support for mobile agent technology. We present a simple execution model for agents that guarantees correct thread migration semantics when moving an agent to another location. Our thread serialization mechanism is however generally applicable in other domains as well, such as load balancing and checkpointing.

1 Introduction

Mobile agent technology is promoted as an emerging technology that makes it much easier to design, implement and maintain distributed systems. Adding mobility to the object-oriented paradigm creates new opportunities to reduce network traffic, overcome network latency and eventually construct more robust programs. Mobile agents are active, autonomous objects or object clusters, which are able to move between distributed locations (e.g. hosts, web servers, etc…) during their lifetime.

Java has been put forward as the platform for developing mobile applications. There are various features of Java that triggered this evolution. First, in a large number of application domains, Java's machine-independent byte code has solved a long-lasting problem known to agent-based systems, namely the fact that agents must be able to run on *heterogeneous* platforms. A Java program is compiled into portable byte code that can execute on any system, as long as a Java Virtual Machine (JVM) is installed on that system. Nowadays, JVM's are running on systems with different

* This research was supported by a grant from the Flemish Institute for the advancement of scientific-technological research in industry (IWT).

D. Kotz and F. Mattern (Eds.): ASA/MA 2000, LNCS 1882, pp. 29-43, 2000.

hardware and system software characteristics (ranging from off-the-shelf PC's to Smart Cards). Second, byte code is easily transportable over the net and can be downloaded whenever necessary by means of the customizable Java class loading mechanism [1]. This flattens the way for supporting *code mobility*. Third, Sun's powerful serialization mechanism allows migrating transparently data state of Java objects (i.e. the contents of instance variables), making *object state mobility* possible. Fourth, Java offers security concepts, allowing construction of secure agent execution environments [2].

Unfortunately, Java is not designed as an agent-based system programming language and therefore most agent-related functionality has to be added. Current research tackles this problem by offering this functionality in the form of *middleware support* for mobile agents. Conventional middleware technologies offer communication mechanisms and a programming model to application programmers for developing distributed applications. In order to support agent-based applications, such conventional middleware functionality is extended with mobile object semantics (e.g. semantics concerning the location of an object, etc.), mechanisms for migration of agents, infrastructure for receiving arriving agents, resource management, execution support, security etc. [3, 4]. In the past, research groups and companies have built various mobile agent systems, fully implemented in Java [2,5].

1.1 Problem Statement: Transparent Thread Migration in Java

Migration is a mechanism to continue the current execution of an agent at another location in the distributed system. To migrate an agent, some state information of the agent has to be saved and shipped to the new destination. At the target destination, the state of the agent is reestablished and finally the execution of the agent is rescheduled.

From the technical view, an agent consists of an object or a cluster of objects. In Java, each object consists of the following states:

- Program state: this is the byte code of the object's class.
- Data state: the contents of the instance variables of the object.
- Execution state: a Java object executes in one or more *JVM threads*. Each JVM thread has its own *program counter register* (pc register) and has a private *Java stack*. The Java stack is equivalent to the stack of conventional languages. A Java stack stores *frames*. A new frame is created each time a Java method is invoked. A frame is destroyed when its method completes. Each frame holds local variables and an operand stack for storing partial results and passing arguments to methods and receiving return values [6].

In order to migrate a thread, its execution must be suspended and its Java stack and program counter must be captured in a serializable format that is then send to the target location. At the target location, the stack must be reestablished and the program counter must be set to the old code position. Finally the thread must be rescheduled for execution. If migration exhibits this property it is called *transparent* or *strong* migration. If the programmer has to provide explicit code to read and reestablish the state of an agent, migration is called *non-transparent* or characterized as weak migration [7].

Although code migration and data migration is strongly supported in Java, thread migration is completely not supported by current Java technology. JVM threads are not implemented as serializable. Furthermore, the Java language does not define any abstractions for capturing and reestablishing the thread state information inside the JVM. Due to these technical obstructions, recent state-of-the art middleware support for Java-based mobile agents such as Mole is not able to provide agent applications with transparent thread migration. Weak thread migration is supported, but burdens the programmer with manually encoding the 'logical' execution state of an agent into the agent's data state [2].

1.2 Byte Code Rewriting of Application Code

We implemented a thread serialization mechanism by extracting the state of a running thread from the application code that is running in that thread. To achieve this, we developed a byte code transformer that instruments the application code by inserting code blocks that do the actual capturing and reestablishing of the current thread's state. We implemented this transformer using the byte code rewriting tool JavaClass[8], that offers a programming interface for byte code reengineering.

Our system makes it possible that running threads can be saved at any execution point, even at difficult points such as in the middle of evaluating an expression, or during the construction of an object. The implemented mechanism is fully implemented at the code level. As such, it does not require changes to the JVM.

We now give an overview of the remainder of this paper. In the next section we give an overview of our approach in the context of mobile agents. In section 3, we describe the implementation of how a thread's execution state can be captured and later reestablished. In the following two sections we give a quantitative analysis of our thread serialization mechanism and we discuss related work with regard to this. In section 6, we describe how our thread serialization mechanism can be used in other domains such as load balancing. In section 7, we describe the implementation status of the prototype and raise some final issues related to the subject of this paper. Finally, we conclude.

2 Overview of Our Approach

In this section we give a high-level overview of how our thread serialization mechanism is used for migrating mobile agents. We present a simple execution model for agents that guarantees correct thread migration semantics when moving an agent to another location.

2.1 Execution Model for Agents

In order to allow easy migration, a suitable model for executing agents must be deployed in the middleware layer of the agent system. Figure 1 gives an overview of

this execution model [9]. We offer a complement to JVM threads at a higher abstraction level, namely *tasks*. A task is a higher-level construct for executing a computation (i.e. a sequence of instructions) concurrently with other computations. A task encapsulates a JVM thread that is used for executing that task. As such, a task's execution state is the execution state of a JVM thread in which the task is running.

An agent is implemented as a cluster of Java objects that cooperate together to implement its expected functionality. For executing its program, an agent owns a task that is exclusively used for that agent. This gives an important property to agents, namely that they are self-contained. They do not share any execution state. When migrating the agent its task is migrated with it, without impacting the execution of other agents. In principle, an agent may encapsulate multiple tasks in more intelligent execution schemes (e.g. a mobile agent that has one main task and several helper tasks).

A so-called `TaskScheduler` schedules the execution of tasks. The task scheduler controls what task is to execute next. As such, we can experiment with different scheduling policies, tailored to a specific application domain. When the scheduler starts a task, the task is assigned a JVM thread for executing itself. In principle, the execution model allows creating a new thread for each task, let the different threads run concurrently and rely on the efficient JVM implementation for context switching support.

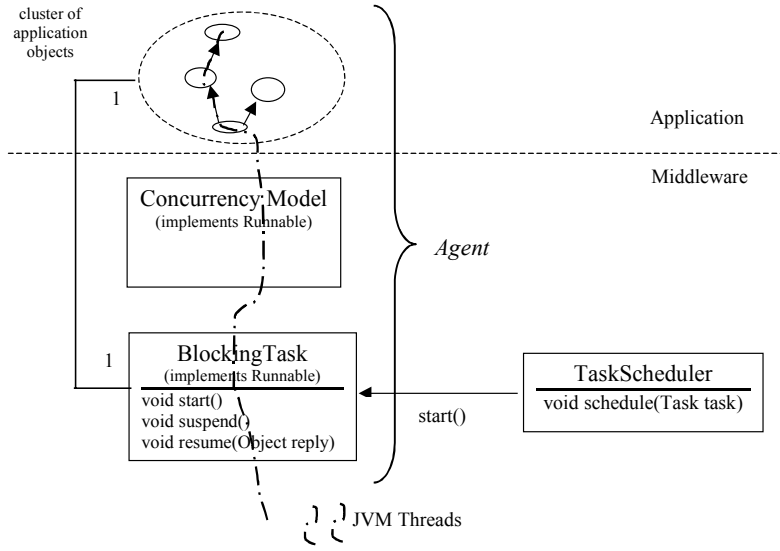

Fig. 1. Execution Model for Agents

It is important to realize that this execution model is transparently enforced upon the mobile agent application by the middleware layer. For example, the enforcement of synchronization constraints inherent to concurrent execution should ideally be realized within the middleware layer of the agent system. Tasks support this, since they are programmable entities: system developers (who build middleware)

implement specialized tasks that offer primitives useful for constructing various powerful concurrency models with built-in support for specific synchronization mechanisms. For example in Figure 1, a `BlockingTask` is a specialized task derived from the base class `Task` that is used for implementing a concurrency model supporting synchronous invocation semantics between agents.

2.2 Serializable Tasks

A task is serializable at any execution point, making transparent thread migration possible. In opposition to the Java object serialization mechanism, task (de)serialization is not automatic but must be initiated by calling two special primitives, which are each defined as a method on `Task`. These primitives are meant for requesting capturing and reestablishment of a task's execution state. We illustrate the use of these primitives in the context of a migration scenario of an agent:

- `public void capture()`. This method must be called whenever the current executing task must be suspended and its execution state must be captured in a serializable format. For example, an agent that wants to migrate invokes this method (indirectly) on its task. After state capturing is finished, a migrator middleware component migrates the agent together with its serialized task to the target location. Notice that our task serialization mechanism requires that state capturing must be initiated within the execution of the task that is to be migrated.
- `public void resume()`. Invoking this method on a `Task` object, will reschedule the task with the task scheduler. When the task scheduler restarts the task (by calling `start()` on it), the execution state of the task is first reestablished before resuming execution. In the migration example, this method will be called by the peer migrator middleware component at the target location after it receives the migrating agent.

As a consequence, tasks are not serializable at each moment, but only after the execution state capturing process is finished and before the execution state reestablishment process is started.

Each task is associated with a separate `Context` object into which its thread execution state is captured, and from which its execution state is later reestablished. This context object can be serialized by means of the Java object serialization mechanism. To capture and reestablish a task's thread state, byte code instructions are inserted into the code of the encapsulating agent. These instructions capture the current Java stack and the last executed instruction (i.e. program counter index) into the task's context. Obviously, byte code instructions are also inserted that reestablish the original stack from the context and jump to the instruction where execution was suspended. The next section discusses this in detail.

3 Implementation of Thread Serialization

Each task is associated with a number of boolean flags that represent a specific execution mode of the task. A task can be in three different modes of execution:

- Running: the task is normally executing.
- Capturing: the task is in the process of capturing its current execution state into its context. When the task is in this mode, its flag `isCapturing` is set.
- Restoring: the task is in the process of reestablishing its previous execution state from its context. When the task is in this mode, its flag `isRestoring` is set.

In the rest of this section, we describe respectively the mechanisms behind the state capturing and reestablishing process. Finally we shortly describe the implementation of the byte code transformer itself.

3.1 Capturing Execution State

Whenever an agent wants to migrate, it calls indirectly on its task the operation `capture()` that suspends the execution of the agent and initiates the state capturing process by setting the flag `isCapturing`.

```
public void capture() {
    Context currentContext = getContext(Thread.currentThread());
     if (currentContext.isRestoring) {
       currentContext.isRestoring = false;
    } else  {
       currentContext.isCapturing = true;
       currentContext.pushThis(this);
    }
}
```

Fig. 2. Starting Capturing and Finishing Reestablishment

Since the execution state of an ongoing task is a sequence of stack frames located in the method call stack of the task's JVM thread, we traverse that stack and do state capturing for each stack frame. This is realized by subsequently suspending the execution of each method on the stack after the *last performed invoke-instruction (LPI)* executed by that method, and starting with the top frame's method.[1]

Figure 3 illustrates how this works. In this example, the method `computeSerial()` is the top frame's method. When control returns from `capture()`, a state capturing code block for the top frame is first executed. The frame is then discarded by suspending the execution of its corresponding method (i.e. `computeSerial()`) through an inserted `return` instruction, initiating the state

[1] The top frame is also called current frame [6] and corresponds with the method that invokes `capture()` - the current method.

capturing process for the previous frame on the stack (i.e. `myMethod()`). The same process is then recursively repeated for each frame on the stack.

For each method on the stack, we save the corresponding stack frame in the state it was before executing the method's LPI. An *artificial program counter* is also stored in the task's context. This is a cardinal index that refers to the LPI.

Our byte code transformer inserts a state capturing code block after every invoke-instruction occurring in the application code. These are all the code positions in a method `myMethod()` where control may be transferred back to, after the capturing of the called method `computeSerial()` is finished.

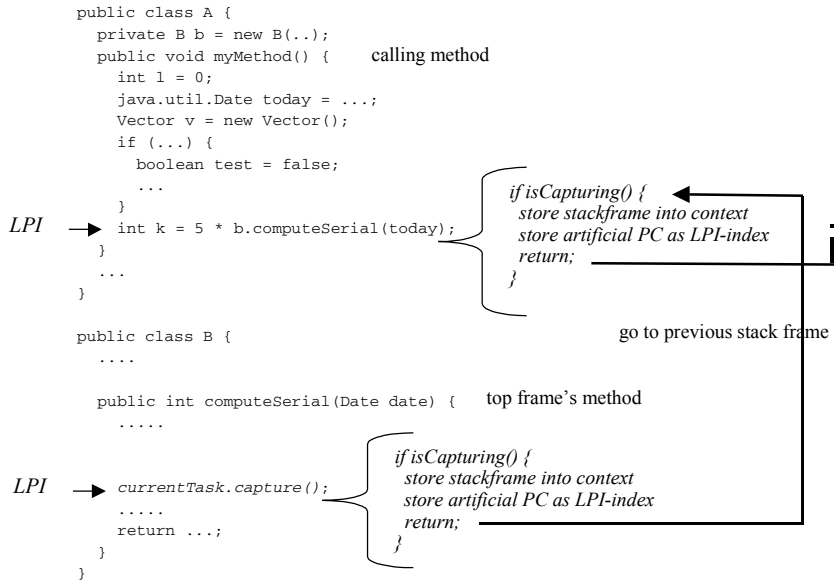

Fig. 3. State Capturing

3.2 Reestablishing Execution State

Calling the operation `resume()` upon a suspended `Task` object reschedules this task with the task scheduler. Actual reestablishment of the task's execution state is however initiated when the task scheduler restarts the task. To reestablish the execution state, we just call again all the relevant methods in the order they have been on the stack when the state capturing took place. Figure 4 illustrates this for the same example as in section 3.1.

Our transformer inserts additional code in the beginning of each 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; these code positions are the invoke-instructions that occur in the method body. In a switch

instruction the appropriate state reestablishing code block is selected based on the stored LPI-index of the method. The chosen code block then restores the method's frame to its original state and restores the method's pc register by jumping to the method's LPI, skipping the already executed instructions. Executing its LPI initiates the reestablishing process of the next stack frame. Finally, during reestablishment of the top stack frame, the operation `capture()` is invoked by our inserted code for a second time. Now, this operation sets the `isRestoring` flag back to false, finishing the reestablishing process (see Figure 2).

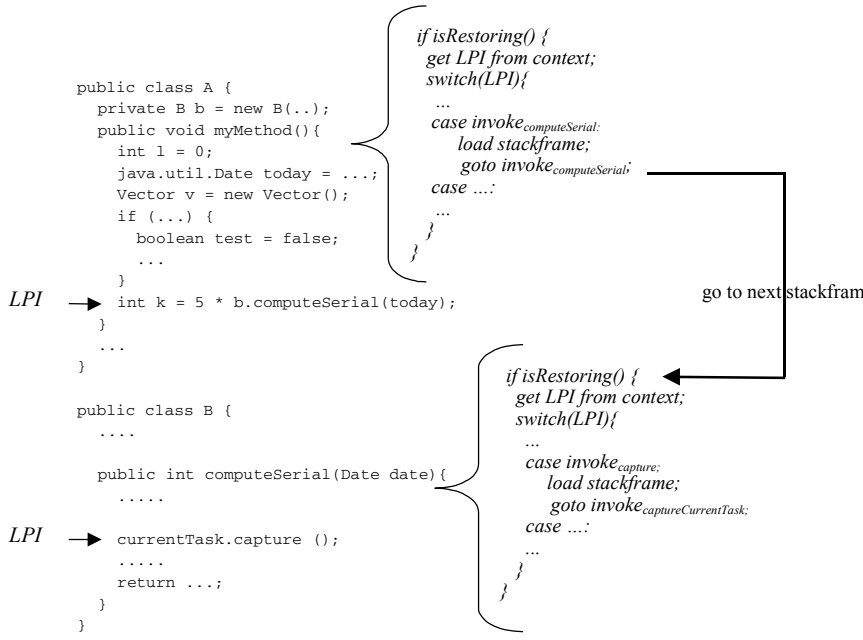

Fig. 4. State Reestablishment

3.3 Implementation of Transformer

In order to generate the correct state capturing and reestablishing code block for a invoke-instruction occurring in a method, we first need to know what's on the method's stack frame before that invoke-instruction is executed. That is, we need to analyze the type of the local variables visible in the scope of the instruction and the type of the values that are on the operand stack before the instruction is executed. This analysis is rather complex, since type information is not anymore explicitly represented in the byte codes of the method. The implementation of this analysis is based on a type inference process that is similar to the one used in the Java byte code verifier, as described in the Java Virtual Machine specification [6].

After the type inference analysis completes, our transformer starts rewriting the original application code method per method. Each method is rewritten invoke-

instruction per invoke-instruction. Our transformer distinguishes between instance method invocations, static method invocations, super-calls and constructor invocation. For example, Figure 5 shows the information captured and reestablished when the method's LPI is an instance method invocation (`invokevirtual`).

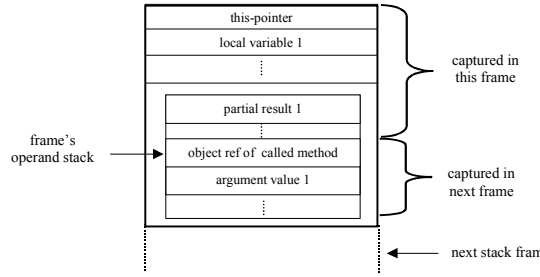

Fig. 5. Stack Frame before Invoking an Instance Method

4 Quantitative Analysis

Instrumenting and inserting code introduces time and space overhead. Since code is inserted for each invoke-instruction that occurs in the program, the space overhead is directly proportional to the total number of invoke-instructions that occur in the agent's application code. Note that JDK method calls are not instrumented (see section 7.2).

Per invoke-instruction, the number of additional byte code instructions is a function of the number of local variables in the scope of that instruction (L), the number of values that are on the operand stack before executing the instruction (V) and the number of arguments expected by the method to be invoked (A). Table 1 shows per invoke-instruction the file space penalty in terms of the number of additional byte code instructions.

Table 1. Space and Time Overheads

L	*V*	*A*	*File space penalty* **+ 2c (c<1)** *(max/avg #instr)*	*Overhead normal execution (# instr)*	*Duration capturing frame (# instr / avg. time)*	*Duration reestablishing frame (# instr/ avg. time)*
0	0	0	14 / 14	4	71 / ≤0.001 ms	45 / ≤0.001 ms
1	0	0	19 / 18	4	105 / ≤0.001 ms	68 / ≤0.001 ms
0	1	0	17 / 16	4	104 / ≤0.001 ms	67 / ≤0.001 ms
0	0	1	15 / 15	4	71 / ≤0.001 ms	46 / ≤0.001 ms
5	1	2	44 / 41	4	274 / 0.002 ms	184 / ≤0.002 ms
3	3	3	41 / 38	4	272 / 0.002 ms	183 / ≤0.002 ms
5	5	5	59 / 54	4	406 / 0.007 ms	275 / 0.004 ms

The last three columns of the table show respectively the run-time overhead per invoke-instruction during normal execution and the overheads of the capturing and

reestablishment of one stack frame. These run-time overheads are expressed as the number of additional byte code instructions executed, and as an average time measurement (expressed in milliseconds). Notice that run-time overhead during normal execution is constant per invoke-instruction.

To summarize, we give a more formal analysis of the different overheads (in terms of additional inserted byte code instructions):

- Maximum file space penalty = 14 +5L + 3V + A + 2c (c <1)
- Runtime overhead during normal execution = 4
- Maximum duration capturing frame = 71 + 34L + 33V
- Maximum duration reestablishing frame = 45 + 23L + 22V + A

In practice, file space penalty and run-time overhead is proportional to the number of nested method invocations. This is of course completely dependent on the complexity of the application under consideration. Table 2 shows experimental time and file space measurements for two sample programs. File space penalty is relatively high for these examples. This is because these sample programs are characterized by a high ratio between the number of invoke-instructions and the total number of byte code instructions. However in more 'normal' software development, we experienced for two in-house developed applications an average byte code blow-up factor of 30% and 37%.

Table 2. Experimental Data

	Overhead normal execution	*File space penalty*
Factorial(100)	28.8 ms – 28.4 ms: 1.5 %	1031KB – 498 KB: 107%
Fibonnaci(30)	430 ms – 340 ms: 27 %	1018KB – 494KB: 106%

All the above run-time tests were performed on a Pentium II 350 MHz, 64 Mbytes, Linux, Blackdown JDK1.2.2, JIT enabled.

Data overhead per captured stack frame is small since this consists of an integer index pointing to the last performed invoke-instruction (LPI), and the associated object reference in case of an instance method invocation.

5 Related Work

There are systems [11,12,13] that provide the required state capturing of Java programs. However they modified the Java VM. The big disadvantage of these systems is that the implemented support is not portable across existing Java platforms. We found that it is possible to capture execution state efficiently at the application code level, without requiring a modification of the JVM. This makes our thread serialization mechanism portable across standard Java platforms

Researchers at TU Darmstadt [7] have implemented a thread serialization mechanism that takes the same approach as we do. However, this is done by pre-

processing the *source code* of the application, which is rather limited compared with byte code transformation. First, with source code transformation, it is not possible to extract the complete execution state of a running thread. It is for example not possible to inspect the values that are on the operand stack of the current executing method. Secondly, our thread serialization mechanism is more efficient in terms of space and time overhead, due to the higher precise control offered at the byte code level. For example, in [7] one reports a worst-case byte code blow-up factor of 470%, while we experienced a worst-case blow-up factor of 107% for a similar sample program. This difference in efficiency is because the low-level byte code instructions make it much easier to manipulate the control flow in a program. For example, to prevent re-execution of already executed method code during reestablishment we skip the already executed code with a simple `goto` instruction. This instruction is however not available at the source code level: in [7] one introduces instead a not small number of if-statements to organize the skipping of already executed code. Although this can be optimized using an unfolding technique [14], the general claim of improved efficiency with byte code transformations remains. Third, byte code transformation provides us with more flexibility. For example, with byte code transformations it is allowed to insert reestablishing code before the execution of the default super-call within a constructor, while this is not allowed at the source code level. Finally, several practical benefits arise from the use of byte code transformations: load-time modification and instrumentation of third party libraries (of which the source code is not a priori available) is possible at the byte code level, while it is not at the source code level.

The existing approaches that perform instrumentation at the source code level [7, 14] have used the exception throwing facility to capture execution state. We have chosen not to use the exception mechanism, since entries on the operand stack are discarded when an exception is thrown, which means that their values cannot be captured from an exception handler.

6 Thread Migration Initiated by an External Control Instance

Until now we have only illustrated a migration scenario, where migration is initiated by an agent itself. Another scenario is where migration of an agent is initiated by an external control instance such as a load balancer component.

Our current thread serialization implementation is more difficult to use for such systems. Remember that state capturing of a task can only be initiated within the execution of that task itself. This makes it difficult for the external control instance to initiate the state capturing process of another agent's task. This deficiency may be solved by associating with each task an additional fourth boolean flag, that signals an external thread serialization request when set to true. An additional byte code instruction that checks the value of this flag must then also be inserted after each candidate LPI.

Currently, we follow another working approach in stead that consists of using a more restricted variant of the agent execution model (see section 2.1). This rigid execution model requires that on each point of time only one task is running while all

other active tasks in the application are temporarily suspended and captured in a serializable format.

As stated a general requirement for the agent execution model in section 2.1, also this variant execution model must transparently be enforced upon the application by a dedicated middleware implementation, relieving the agent programmer from any responsibility in complying with this variant execution model. This middleware implementation is based on an in-house developed meta-level architecture called Correlate with a generic Meta-Object Protocol (MOP) for run-time reflection [9]. Here, the middleware implementation is deployed at the meta-level, while the application logic of the agents resides at the base-level.

Not only task scheduling but also context switching is now controlled at the meta-level, without relying on the preemptive scheduling support of the JVM at all. There is only one JVM thread running in the system. All tasks are scheduled within this JVM thread by the task scheduler. To achieve context switching between tasks, we use our thread state capturing mechanism. That is, at each meta-level interception point (i.e. when meta-level logic takes control over base-level logic), we suspend and capture the currently executing agent and let the task scheduler decide which task is to be executed next, always using the one existing JVM thread. The Correlate MOP guarantees that all this happens transparently to the application logic and thus puts no responsibility on the agent programmer to call `capture()` for each separate interception point.

As such, the Correlate MOP allows us to implement at the meta-level an *automated* and *coordinated multitasking* scheme that satisfies the variant execution model. The external control instance is now implemented as a meta-level component. Whenever the external control instance is executing, it can safely assume that all other active tasks are a priori suspended in a serializable form. Although time-inefficient, this approach avoids polling of the external control instance to discover when an agent's task is ready for migration. We demonstrated a load balancing application using this variant execution model at OOPSLA'99 [10].

7 Discussion

In this section we discuss relevant issues that relate to the subject of this paper and we describe the implementation status of our current prototype.

7.1 A Classloader for Mobile Code

Since we perform a byte code level transformation, our thread serialization mechanism requires that all methods that might initiate state capturing must be transformed. Since mobile agent applications are in general of very dynamic nature, it is often not possible to predict on advance which classes need to be transformed and which classes not. This problem can be handled by deferring transformation until run-time. In Java, this can easily be realized by implementing a custom classloader for mobile code that automatically performs the transformation. In this regard, the

overhead induced by the transformation process – which is not small for the current implementation - becomes a relevant performance factor.

7.2 Transforming the Java API Libraries

Since a mobile agent – like any Java program – may use the JDK libraries, the question arises whether it is necessary to transform these standard provided libraries too. From a technical point of view, this is only a problem when a library call causes a native method to be placed on the thread stack. We cannot handle this case, since we extract thread execution state at the byte code level.

In our current prototype we chose however not to transform the JDK libraries at all (nor the JDK method calls that happen from within the application code). In most cases this is indeed not necessary, since library calls do not initiate state saving by themselves. The exceptions to this are library calls that result in a callback to the application code [7]. For example when the agent programmer uses the Observer pattern with graphical packages such as Swing, callbacks occur. We believe however that using callbacks is a dangerous programming style for agents, since it may violate the thread encapsulation principle (see section 2.1).

7.3 Implementation Status of Current Prototype

An interesting problem with state capturing arises when so-called *non-initialized* values are on the stack. These values cannot be saved. This problem occurs when suspending the execution during the evaluation of the arguments for a constructor operation. In this case, a non-initialized object reference was earlier pushed on the operand stack by the byte code instruction `new`. Our transformer deals with this problem by taking the code block that computes the argument values, moving it before the `new` instruction and storing these values in temporal local variables. These temporal variables are then used for retrieving the argument values when invoking the constructor.

Although possible, we have not yet implemented state capturing during the execution of an exception handler. The major difficulty here is dealing with the `finally` statement of a `try` clause.

A third, more pragmatic issue is that our byte code transformer currently throws away all debugging information associated with a Java class. This affects of course the ability to debug a transformed class with the source-code debugger.

8 Conclusion

In this paper, we presented a portable mechanism for thread serialization in Java, enabling transparent migration of mobile agents. This mechanism is realized by capturing a thread's execution state at the byte code level. The implemented prototype is more efficient then similar approaches that extract thread state at the source code level. This prototype has been used at an OOPSLA'99 demonstration [10].

Acknowledgements

We would like to thank Erik Van Hoeymissen for his work on applying the thread serialization mechanism in the domain of load balancing. We also wish to thank Bo Nørregaard Jørgensen for the many fruitful discussions. Finally, we would like to express our appreciation to Danny Lange, Mitsuru Oshima and the anonymous reviewers for their useful suggestions to improve this paper.

References

1. S. Liang and G. Bracha. Dynamic Class Loading in the Java Virtual Machine. In Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'98), pp. 36-44, 1998.
2. M. Straßer, J. Baumann and F. Hohl. Mole - A Java based Mobile Agent System. In *M. Mühlhäuser: (ed.), Special Issues in Object Oriented Programming*, pp. 301-308, 1997.
3. Y. Berbers, B. De Decker, W. Joosen. Infrastructure for mobile agents. In Proceedings of the Seventh ACM SIGOPS European Workshop: System Support for Worldwide Applications, pp. 173-180, 1996.
4. S. Fünfrocken. Integrating Java-based Mobile Agents into Web Servers under Security Concerns. In *Proceeding of the IEEE Hawai'i International Conference on System Sciences,* 1998.
5. Mitsubishi Electric, Concordia online information, http://www.meitca.com/ HSL/Projects/ Concordia.
6. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.
7. S. Fünfrocken. Transparent Migration of Java-based Mobile Agents (Capturing and Reestablishing the State of Java Programs). In *Kurt Rothermel, Fritz Hohl (Eds.), Proceedings of the Second International Workshop on Mobile Agents (MA'98)*, pp. 26-37, 1998.
8. M. Dahm. Byte Code Engineering. In *Clemens Cap, editor, Proceedings JIT'99*, 1999.
9. B. Robben. Language Technology and Metalevel Architectures for Distributed Objects. Phd KULeuven, 1999. ISBN 90-5682-194-6.
10. E. Truyen, F. Matthijs, W. Joosen, B. Vanhaute, B. Robben, R. Slootmaekers, P. Verbaeten. Supporting Object Mobility - from Thread Migration to Dynamic Load Balancing. Demonstration at OOPSLA'99. (Correlate v3.3, Java prototype), www.cs.kuleuven.ac.be/~eddy/PUBLICATIONS/OOPSLADemoProceed.ps
11. M. Ranganathan, A. Acharya, S. D. Sharma and J Saltz. Network-aware Mobile Programs. Proceedings of the USENIX Annual Technical Conference, Anaheim, California, 1997.
12. H. Peine and T. Stolpmann. The architecture of the ara platform for mobile agents. In *Proceedings of the Second International Workshop on Mobile Agents (MA' 97)*, 1997.

13. S. Bouchenak. Pickling threads state in the Java system. In Proceedings of the third European Research Seminar on Advances in Distributed Systems (ERSADS'99), 1999.
14. T. Sekiguchi, H. Masuhara, and A. Yonezawa. A Simple Extension of Java Language for Controllable Transparent Migration and its Portable Implementation. In *Coordination Languages and Models,* volume 1594 of LNCS, pages 211-226, Springer-Verlag, April 1999.