

Transparent Migration of Mobile Agents Using the Java Platform Debugger Architecture

Torsten Illmann, Tilman Krueger Frank Kargl, and Michael Weber

University of Ulm, Dep. of Media Computer Science, 89069 Ulm, Germany,
torsten.illmann@informatik.uni-ulm.de,
WWW home page: <http://cia.informatik.uni-ulm.de/>

Abstract. In this paper we describe a transparent migration of mobile agents in Java using the Java Platform Debugger Architecture (JPDA). The JPDA allows debuggers to access and modify runtime information of running Java applications. In the context of mobile agents, the JPDA can be used to capture and restore the state of a running program. Since JPDA does not support to set the program counter, we introduce two different solutions to solve this. We either slightly modify the virtual machine or instrument some byte code instructions. Finally we measure the produced overhead in code and time compared to normal execution and other approaches addressing this problem. Altogether, we show that developing Java-based mobile agents with a transparent migration can be performed nearly without changing the source code, the byte code or the interpreter.

1 Introduction

Agent technology is increasing more and more. Since society is moving steadily towards an information society, the need of personal assistants for searching, supporting in e-commerce transactions and communicating with others is increasing in the same way. Personal software agents are a paradigm that promises to support these needs. Nevertheless, establishing and spreading agent technology in the real world still requires some important aspects to be solved.

Mobile agents are a suitable paradigm especially for mobile and distributed computing. Considering this in combination with the desire of sophisticated and easy-to-use agent systems, the need for development of transparent mobile agents and, in consequence, transparent migration techniques is there.

By migration we mean the movement of an agent to another location in the network (e.g. computer) and transparent continuation at the point before the migration occurred. That means, code and state of the agent must be captured, transferred to and restored at the destination location.

Modern agent systems are mainly implemented in Java because of its features platform independency, dynamic class loading, security issues and object-orientation. Unfortunately, standard Java does not allow to access all the internal runtime information structures of an agent. There has already been done some research to establish transparent migration by modifying source code, byte

code and the interpreter. Since these changes are expensive and/or highly complicated, we introduce another solution where complicated transformations or modifications are not necessary. Using the Java Platform Debugger Architecture (JDPA), which is part of the virtual machine specification, runtime information may be accessed in debug mode. This can be used to perform a transparent migration.

In the following paper, we first introduce the project where this work was developed in. Next, we describe problems of transparent migration and classify our approach. In chapter 4 and 5, we describe in detail how transparent migration is realized using the JPDA. Chapter 6 describes other approaches of transparent migration in Java. In chapter 7 we measure the produced growth in code and execution time of our approach in comparison to others. Finally we discuss our approach in a conclusion and name future work in this area.

2 The CIA Project

The CIA project deals with the development of an infrastructure for personal software agents. The system is called Collaboration and Coordination Infrastructure for Personal Agents [16] and is totally implemented in Java. It combines technologies like Java Messaging (JMS), Jini, Java Enterprise Beans, RMI and applets.

Basically the CIA System consists of three layers: The *agent layer*, the *directory, broker and trading layer (DBT)* and the *service layer*.

The agent layer defines programming models for static and mobile agents as well as topic-based inter-agent communication primitives. As communication models, one may choose among synchronous or asynchronous and uni- or multi-cast communication. Agents belonging to one user (or organization) are clustered in a so called agent cluster [18]. Clusters may be easily spread on workstations, portable computers or PDAs. Permanent connections and temporary connections may be handled in one cluster.

The DBT uses the Jini technology [15] to automatically and spontaneously combine different clusters and services without configuration. It assists agent clusters to find and trade with other clusters or extern services.

The service layer allows to integrate extern services of different kinds of technologies and from different locations into one service platform. For example it is possible to integrate RMI, EJB or CORBA services. With introducing the service layer, services are easily and transparently accessible for agents.

Whereas the design is modular and open, the current development mainly focuses the following topics:

- Communication transparency by using a software bus
- Network transparency with ad-hoc networking mechanisms
- Device- and location transparency of agents' user interfaces
- Integration of extern services in the agents' infrastructure
- Migration transparency of mobile agents

In this paper, we concentrate on transparent migration of Java-based mobile agents.

3 Problems of Migrating Java Programs

To perform a migration of a mobile agent, one has to consider many aspects. Transparent migration in the view of an agent programmer means that he does not have to care about how to migrate. He specifies a migration statement (often called *go* or *move*) anywhere in his agent's code and expects that the agent system manages the whole migration. After migration, code, private data and execution state are the same as before. The only difference should be that the agent resides on a different host.

Figure 1 shows a classification of problems that have to be taken into account when realizing a transparent (or strong) migration [17].

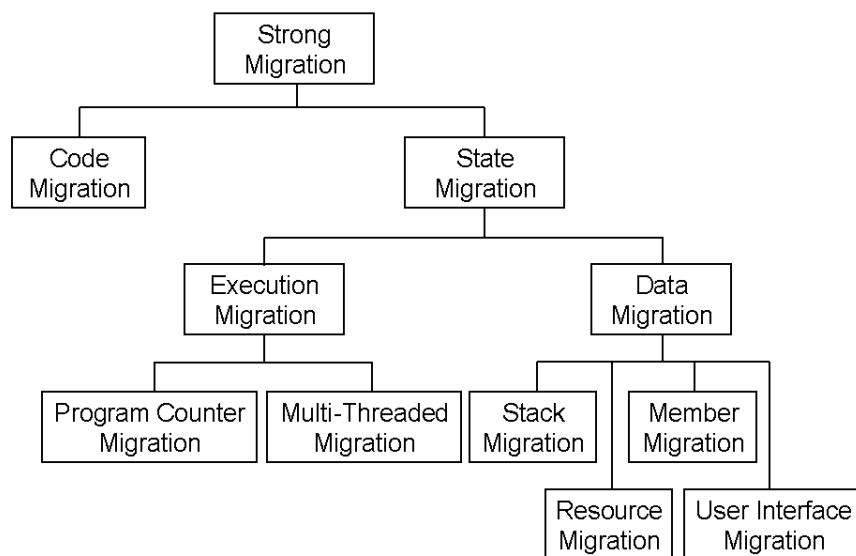


Fig. 1. Problems of transparent migration

At the top level, the classification consists of two aspects: *code* and *state migration*. These refer to code transfer and state transfer of the agent. The state migration is composed of more aspects. On the second level, it is *execution* and *data migration* which means that the state of an agent is generally made up of the current execution point and the current data of the agent. Execution and data migrations are again made up of further parts. The execution migration is composed of the *program counter* and *multi-threaded migration*. *Stack*, *member*, *resource* and *user interface migration* define the data migration.

Whereas code migration and member migration can be implemented using the standard Java mechanisms dynamic classloading [14] and serialization [10], the others (program counter, multi-threaded, stack, resource and user interface migration) are not supported in the standard.

Program counter migration means that the execution at the destination location continues at the same point where it was interrupted before. We assign the re-establishment of the correct calling order of all nested executed methods and their inner code locations to this kind of migration. Generally, there are two peculiarities of this kind of migration: self-migration and forced migration. To develop autonomous and self-responsible mobile agents, self-migration is required. Forced migration is rather needed in load balancing systems than in mobile agent ones.

Multi-threaded migration means the migration of all threads of a multi-threaded agent. That means, threads that the agent has created during this lifetime have to migrate as well. The currently running thread which requests the migration is in a well-known execution state (namely at the point of calling the migrate statement). When interrupting the other threads belonging to the agent, they may either be blocked or at any unknown point of their execution. We call the problem of capturing and restoring the state of all dedicated threads of an agent in the right order, the multi-threaded migration.

Stack migration implies the migration of all local data in every method on the call stack. That data consists of all values of local variables (variable stack) and operands of computations being on the stack (operand stack) up to the point of interruption. Stack migration depends on the program counter migration. It is not be achievable without.

Resource migration addresses the problem of migrating open connections to system resources the agent is connected to. System resources are for example network connections, files or databases. In most reasonable mobile agent scenarios, an agent does not have open connections any more. To achieve total transparency, this problem has to be considered as well.

User interface migration means the problem of migrating the state of the user interface of agents. Because agents are mainly working invisible or the user interface is not bound to the agents location like in our system, this problem does not arise often. Nevertheless, this task also belongs to a transparent migration.

In this paper we focus on the problem to migrate the program counter and the stack. So we assume that the agent is single-threaded, is not connected to open system resources and has no open user interface.

4 Stack Migration Using Java Platform Debugger Architecture

Whereas other projects [1] [2] [3] [4] [5] solve the problem of transparently migrating a thread by modifying source code, byte code or virtual machine, we use the Java Platform Debugger Architecture (JPDA) [13] to perform this task.

JPDA is part of the virtual machine specification [11] (that means implemented by every standard virtual machine) and normally used to develop (remote) debuggers for Java applications. JPDA gives access to runtime information

like running threads, their call stack and their program counter. It is possible to suspend and resume execution, execute single byte code instructions and set/unset breakpoints at arbitrary points. Furthermore, it enables programs to define event handlers that are called when methods are being entered or exited.

The JPDA is made up of two parts: The Java Virtual Machine Debugger Interface (JVMDI) and the Java Debugger Interface (JDI). JVMDI is a native implementation and JDI a Java API built on top of it in order to access the debugger functionality. Since JDI is implemented in Java, all code to access runtime information can be implemented in pure Java. For example, reading stack frames (their local variables) of all currently invoked methods – starting from the deepest one – shows up in JDI as follows:

Code example 1:

```
1 import com.sun.jdi.*;
2
3 // The program can connect to the virtual machine of the
4 // running thread via JDI
5 VirtualMachine vm = ...
6 List threads = vm.allThreads();
7
8 // index (or name) of current thread must be known
9 int current = ...
10 ThreadReference thread = (ThreadReference) threads.get(current);
11
12 // read all stack frames (starting from last one)
13 List frames = thread.frames();
14 for(Iterator i=frames.iterator(); i.hasNext(); ) {
15     StackFrame frame = (StackFrame) i.next();
16
17     // read all local variables and output their values
18     List locals = frame.visibleVariables();
19     for(Iterator j=locals.iterator(); j.hasNext(); ) {
20         LocalVariable var = (LocalVariable) j.next();
21         System.out.println(var.typeName() + " " +
22                             var.name() + " = " +
23                             frame.getValue(var));
24     }
25 }
```

Using the above functionality, it is simple to store all stack frames in the case an agent requests to migrate. The agent activates the debugger who suspends the agent thread, stores all values of local variables of all stack frames in a serializable variable called *stack*. This variable is for example implemented as member field in the mobile agent's base class. Consequently, stack frames are automatically transmitted with the serialized agent. To achieve this behavior, the above algorithm has to be changed from line 22 to 24.

Code example 2:

```
22     Value value = frame.getValue(var);
23     stack.push(value);
```

The restoration of stack frames is nearly as simple. The *StackFrame* class allows to set local variables using the *setValue* method. Since the values of all local variables are popped from the serialized stack in opposite order, the local variables have to be iterated in reverse order as shown below (change line 20 to 25 of code example 1):

Code example 3:

```
20  for (int i=locals.size(); i <= 0; i--; ) {
21      LocalVariable var = (LocalVariable) locals.get(i);
22      Value value = (Value) stack.pop();
23      frame.setValue(var, value);
24  }
```

Unfortunately, JDI does not provide interfaces to access the operand stack of a Java thread. We requested this feature to be integrated within future JDI specifications. Nevertheless, one may achieve a migration without transmitting the operand stack by either defining a simple programming convention for agent programmers. To avoid this problem, one may not specify nested calls of methods that request to migrate without storing intermediate results in local variables. The following example illustrates this convention.

Code example 4:

```
01  int result = method1() + method2();

01  int temp   = method1();
02  int result = temp + method2();
```

In line one of the upper sample, the method *method1* is called first and its result is being pushed on the stack as operand for the later addition. If method *method2* requests a migration, this operand still has to be on the stack after restoration to perform a correct addition. Using the lower modification (line 01 and 02) where the intermediate result are stored in a temporary local variable, this problem does not arise any more. Certainly, this convention can be automated by slightly instrumenting the agent's byte code.

The restoration of local variable values must be performed for all methods being called in the moment the migration is being initiated. This method invocation list is restored using the program counter migration described in the following chapter.

5 Program Counter Migration

The previous chapter describes the possibility to migrate the stack of a mobile agent's thread. To perform a transparent migration, we further need to migrate

the program counter. In more detail, that means that the sequence of nested invoked methods that are in execution right before migration and the locations within each method have to be captured, transferred and re-established.

Standard Java does not allow to access the program counter of the currently executing methods. Again in debug mode, the JPDA allows to access the location within a stack frame of a method. The value of the location identifies the byte code index relative to the start instruction of the method. Storing locations of all methods of the agent's thread, makes it possible to capture the program counter of this thread. Code example 5 illustrates this:

Code example 5:

```
15 for(Iterator i=frames.iterator(); i.hasNext(); ) {
16     StackFrame frame = (StackFrame) i.next();
17     ...
..    // store local variables' values
24    // store program counter
25    long pc = frame.location().codeIndex();
26    stack.add(pc);
27 }
```

The methods' program counters are also pushed on the stack to be transmitted. Since the program counter of a method is required earlier than values of local variables during the restoration process (see below), we push it on the stack after pushing local variables.

Re-establishing the program counter could have been in analogy to capturing them. An event handler is registered at the JPDA framework to be called whenever a method in the agent's thread is entered. The agent's thread is suspended if an entry event occurs. In this handler, we execute the code to restore the program counter and the previous mentioned local variables of this method. At last, we continue the execution. The following code fragment shows the simplicity of this event handler:

Code example 6:

```
1 public void methodEntered(MethodEntryEvent e) {
2     e.disable();
3     StackFrame frame = e.thread().frame(0);
4     long pc = stack.pop();
5     Method method = frame.location().method();
6     Location newLocation = method.locationOfCodeIndex(pc);
7     frame.setLocation(newLocation);          <--- does not exist!
8     // set local variables ...
9     e.enable();
10    e.thread().resume();
11 }
```

The algorithm terminates when the *migrate* method of the agent framework is being entered. Unfortunately, the *StackFrame* class does not support a method

to set the location (line 7). If there was such a method, the total migration functionality could be done in JPDA. No modification or transformation of source code, byte code or virtual machine would be needed.

Because of the lack of this method, we realize two small alternative extensions to support this functionality: we slightly change the virtual machine or the byte code of the agent.

5.1 Changing the Hotspot Virtual Machine

One alternative to set the program counter is to modify the virtual machine. Since the standard virtual machine of Sun Microsystems [12] is open-source, we examined the source code for adding this functionality. Within the implementation of the Java Virtual Machine Debugger Interface (JVMDI) which is the native part of JPDA, it is possible to access runtime information of current executing threads on C++ basis. The JDI described earlier is built on top this native implementation. In there, the following method headers are declared:

```
jvmdiError GetCurrentFrame(jthread *thread, jFrameID *frame);
jvmdiError SetCurrentFrame(jthread *thread, jFrameID *frame,
                           jlocation *location);
```

The implementation of *GetCurrentFrame* enables to access the frame location within a running thread. The function *SetCurrentFrame* should allow to set the frame location. Surprisingly, this function is declared but has never been implemented. It seems to us that Sun planned to implement it and either forgot to do it or, since debuggers normally do not need it, forgot to remove this declaration.

In addition, there is an intern method *frame::interpreter_frame_set_bci* which sets the location of a stack frame by specifying the byte code instruction index (BCI). We add the implementation of *SetFrameLocation* using this method (about 20 lines of code) and finally make it accessible using a JNI-interface. Whenever a program counter restoration is requested, this method is called with the byte code index being de-serialized from the stack.

We propose to integrate this patch in the virtual machine specification and reference implementation. The current release version of Sun's reference implementation included in the jdk1.3.1 is Hotspot 2.0 (client and server). We further suggest to integrate the function to set frame locations in the Java Debugger API (JDI). If this patch gets accepted, a transparent migration is feasible in pure Java without modification of any line of code.

The added code is located in the platform-independent *share* part of the Hotspot's source code. That means that all changes are portable for other operating systems. We already re-compiled the Hotspot 2.0 for Linux and Win32.

5.2 Instrumenting Byte Code

Another alternative to re-establish the program counter is the instrumentation of byte code instructions. Only few instructions enable the functionality of setting

the program counter. First, the minimal set of byte code instructions to be instrumented have to be identified. Studying the virtual machine specification [11] in detail, we found out three small modifications that achieve this goal:

1. A program counter must be specified as local variable in each method.
2. A branch statement must be instrumented at the beginning of each method. It branches in dependency on the program counter to all locations where the execution may continue after a migration has been occurred.
3. Before invoking a method possibly performing a migration, the program counter has to be incremented.

In order to determine which methods are to be instrumented and which locations needs to be branched to, the following code example is examined:

Code example 7:

```
1 public void calculate() {
2     System.out.println("Do migration ...");
3     migrate();
4     System.out.println("Do migration recursively ....");
5     calculate();
6 }
```

The migration is initiated directly in line 3. In line 5 the method is called recursively and thus initiates further migrations (for simplicity, this method never ends). To determine the methods being instrumented we first have to find out all methods that call *migrate* directly. Then, all parent methods calling them have to be retrieved iteratively using a bottom up search algorithm. We call these methods *migratory methods*. All migratory methods have to be instrumented by a program counter and a branch instruction that branches to all invocations of other migratory methods within this method. Looking at the byte code of method *calculate* of the previous code example, its transformation is explained:

Code example 8:

```
0: getstatic    java.lang.System.out Ljava/io/PrintStream;
3: ldc         "Do migration ..."
5: invokevirtual java.io.PrintStream.println (Ljava/lang/String;)V
8: aload_0
9: invokevirtual Agent.migrate ()V
12: getstatic   java.lang.System.out Ljava/io/PrintStream;
15: ldc         "Do migration recursively ..."
17: invokevirtual java.io.PrintStream.println (Ljava/lang/String;)V
20: aload_0
21: invokevirtual Agent.calculate ()V
24: return
```

The migratory methods are *migrate* and *calculate*. They are invoked using the *invokevirtual* instruction at locations 9 and 21. The instrumentation of byte code as described above produces the following result:

Code example 9:

```
0:   iconst_0
1:   istore_1
2:   iload_1
3:   tableswitch default = 24, low = 1, high = 2(34, 47)
24:  getstatic   java.lang.System.out Ljava/io/PrintStream;
27:  ldc         "Do migration ..."
29:  invokevirtual java.io.PrintStream.println (Ljava/lang/String;)V
32:  iconst_1
33:  istore_1
34:  invokevirtual Agent.migrate ()V
37:  getstatic   java.lang.System.out Ljava/io/PrintStream;
40:  ldc         "Migration done."
42:  invokevirtual java.io.PrintStream.println (Ljava/lang/String;)V
45:  iconst_2
46:  istore_1
47:  invokevirtual Agent.calculate ()V
50:  return
```

We inserted four byte code instructions at the beginning of the method. The original start instruction moved to location 24. From location 0 to 1, we defined the program counter as local variable and initialized it to 0 for normal execution. From location 2 to 3, we branch according to its value to either the original start location (24) or all invocations of migratory methods (34 and 47). Whenever a migration method is invoked, an incrementation of the program counter is inserted before (32-33 and 45-46).

During normal or initial execution the same byte code than before transformation is executed, since the *table switch* statement (3) defaults to the original start instruction (24). In case of re-establishing the program counter after migration, the JPDA event handler of code example 6 is used. Instead of invoking the method *setLocation* in line 5, the following steps have to be performed:

1. Single step two byte code instructions until program counter is initialized. The execution is before location 2 now.
2. Set the program counter to the value popped from the stack via JDI.
3. Continue execution.

To implement these byte code modifications we use the Byte Code Engineering Library (BCEL) [8] developed at the University of Berlin by M. Damm. BCEL is open source [9] and provides a comfortable Java API in order to load, inspect, modify and save back byte code. Every byte code instruction and entity like classes, methods, fields and the constant pool is represented by a separate class. Creating instances of these instruction classes and adding them to the appropriate method instance easily modifies existing byte code. To instrument the functionality of initializing the program counter and branching to different locations, the following calls of BCEL have been used:

Code example 10:

```
1 instructions.insert(new TABLESWITCH(pcValues, destinations,  
2         instructions.getStart()));  
3 instructions.insert(new ILOAD(pc));  
4 instructions.insert(new ISTORE(pc));  
5 instructions.insert(new ICONST(0));
```

Instructions is the list of all instructions of a certain method. Instructions are inserted in opposite order (line 5 to 1) using the *insert* method. The branch instruction (line 1) needs all possible values of the program counter *pcValues*, the default branch location *instructions.getStart()* and all destination locations *destinations* where calls to migratory methods occur.

This byte code transformation is executed in a self-developed classloader [14] which transforms mobile agents' code once after loading their original byte code. Therefore, the modification never is persistently visible.

6 Related Work

There are several projects that implemented transparent strong migration as well. In the following, we describe these projects and state differences compared to our approach.

Wasp [1] The WASP project being developed at the University of Darmstadt aims at agents being integrated in web servers. The system implements transparent migration using a source code transformation (pre-compiler). The transformation inserts code to store and restore the state of the agents. The program counter is implemented by adding conditional branches to the migratory methods. The capturing of the agent's stack is done within exception handlers. When the agent requests to migrate, an exception is thrown and in every called method an exception handler is instrumented with stores all local variables within a stack variable. Re-establishing the stack is done by setting the local variables after each invocation of methods. Unfortunately, the source code is modified in many more ways to cover all possible cases. After the transformation the source code is unreadable and fully blown-up. The agent class has to be recompiled before execution.

Sirac [5] In Sirac project, the standard Java virtual machine was modified for thread mobility and persistence. It also extends the standard Java API by introducing lower level methods to capture and restore threads. A higher level API provides primitives to perform thread mobility or thread persistence. In addition to self-initiated thread mobility, they also allow to force the migration by other threads. This functionality is for example needed to develop load balancing or automatic activation/deactivation systems. Its application context is not focused on mobile agent systems. So far, the modified virtual machine is only available for JDK1.2.2.

Nomads [2] The Nomad system implements their own virtual machine for performing transparent migration. They further realize special functionalities within this machine called fine-grained resource control. Connections to resources (e.g. CPU, network connections, file connections) can be restricted by passing parameters to the virtual machine.

Brakes [4]

The Brakes approach realizes migration transparency by instrumenting all necessary functionality, i.e. stack and program counter migration, in the byte code via a post-compiler. For multi-threaded environments they define their own threading framework: *Tasks* have to be used instead of threads and a separate scheduler has been implemented. The framework supports cooperative multitasking only. After every task change (which is initiated by the task itself), the task is being serialized and the next activated. If a migration is requested, all tasks instead of the running one are already in a serialized form and may be transferred to the other location.

JavaGoX [3] The approach of the JavaGoX system is quite similar to the one of Brakes. They instrument the byte code also by using a post-compiler. As far as we know, it does not implement support for multi-threading environments so far.

Many concepts and systems concerning thread mobility and persistence of native (non Java-based) applications for homogenous and heterogeneous platforms have already been developed and implemented. Information about these may be taken from [6] or [7].

7 Measurements

To underline our results and rank them in contrast to other approaches, we made two kinds of measurements:

1. The growth of byte code after instrumentation.
2. The execution time overhead when executing highly recursive methods.

7.1 Growth of byte code after instrumentation

We measured the growth in byte code of three Java programs:

- A simple agent with only one migration call and one normal statement
- the Fibonacci algorithm
- a complex agent performing 10 migration statements and 50 normal statements

Table 1. Comparison of relative growth of byte code size

Approach	Simple Agent	Fibonacci Program	Complex Agent
Sirac	+0%	+0%	+0%
JPDA & Hotspot-Modification	+0%	+0%	+0%
JPDA & Byte Code Transformation	+10%	+5%	+3%
Brakes	+49%	+42%	+25%
JavaGoX	+114%	+61%	+102%

Whereas other projects that modify the byte code produce a high overhead (between 25% and 100%), our byte code instrumentation approach produces only between 3% and 10%. The more *normal* code (not specifying migrations) the agent contains the less overhead the approach produces. The modification of the Hotspot machine (Sirac and our approach) certainly produces no overhead in byte code size because no instructions are inserted.

7.2 Execution time overhead

We use a highly recursive function (the Fibonacci algorithm) to measure execution time and compare results to other approaches. We executed the tests on a PIII-660 workstation with WinME operating system and JDK 1.3.1. Here are the results:

Table 2. Comparison of execution efficiency

Approach	fib(35)
Brakes parallel	+3700%
JPDA & Hotspot-Modification	+810%
JPDA & Byte Code Transformation	+814%
JPDA only	+758%
JavaGoX	+56%
Brakes serial	+28%
Java only	+0%

As we can see from the results, we get one total outlier. The brakes implementation with multi-threaded migration slows immensely down, probably because thread serialization is invoked very often. Since the other tested implementations only consider single-threaded agents, we do not further rate this outlier.

The other results show that the execution in debug mode already produces an overhead of more than 750% even if no migration functionality is used. Our approaches including migration functionality are about 6% slower than the one in debug mode without. In consequence, the main overhead is the requirement that the agent has to be executed in debug mode. In debug mode, the JIT compiler is always disabled. The other results of Brakes serial and JavaGoX use the JIT compiler. The Sirac approach is not included in this evaluation since it is only accessible for JDK 1.2.2.

Looking at these results, a useful feature of the JPDA would be to partly switch on/off the debug mode. Then, the normal execution of an agent where no debug access is needed, could be executed using the JIT as well and produce better results. Furthermore, the discussion to integrate debugger functionality in the JIT has already started in the mailing lists. This clearly would improve our results immensely as our measurements show.

8 Conclusion

In this paper we presented two mechanisms for transparent stack and program counter migration of a Java thread using the Java Debugger Interface (JDI). Since JDI provides access to runtime information like stack frames, local variables and the program counter, the state of a Java program can be captured without modifying source, byte or virtual machine code. Nevertheless, there are two drawbacks to perform a transparent strong migration. On the one hand, the operand stack is not accessible from JDPA yet and, on the other hand, the functionality of setting the program counter is unexplainably missing. For avoiding the problem of the operand stack we propose either a simple programming convention for migratory methods or the use of a pre- or post processor which eliminates these problems [1]. To re-establish the program counter, we propose two alternatives to solve this problem. On the one hand, we slightly modify the standard Hotspot virtual machine to perform this task. On the other hand, we slightly instrument the byte code of the agent to set the program counter.

Algorithms and code examples showing the use of JDI and our suggestion to set the program counter are presented.

Since JDPA is part of the Virtual Machine Specification and our changes to set the program counter are either at byte code level or at the portable parts of the virtual machine's source, this approach is totally portable to other operating systems and different versions of JDK.

Measurements of the growth in byte code show that our approaches add – depending on the considered alternative – no or only little instructions to the agent's code. But they also show that the execution in debug mode is much slower than without.

9 Future Work

In the nearby future, we will try to convince Sun to integrate our small extension to set the program counter in the virtual machine and the JPDA specification. A transparent migration of stack and program counter would then completely be possible using the JPDA.

We will think about an extension of the virtual machine to temporarily suspend the debug mode to activate the JIT compiler and resume it later. Only the process of capturing and restoration the agent's thread would then be slowed down. As long as no migration is requested, the agent could run with usual performance.

Furthermore, we will concentrate our research in other, often disregarded migration aspects, like multi-threaded migration, resource migration and user interface migration.

Our long-term goal is to develop an open source Migration API. Agent system developers or other developers requiring thread mobility or persistence in Java should be able use it in their own systems in order to enable migration techniques. Like a construction kit, it could be possible to choose the desired level of migration transparency and the desired implementation.

References

1. Fuenfrocken, S.: Transparent Migration of Java-Based Mobile Agents. In Proceedings of MA'98, Springer, Stuttgart, Germany (1998) 26–37.
2. Suri, N., Bradshaw, J., Breedy, M., Groth, P., Hill, A. G., Jeffers, R.: Strong Mobility and Fine-Grained Resource Control in NOMADS, In Proceedings of ASAMA'2000, Springer, Zuerich, Germany (2000) 2–15.
3. Sakamoto, T., Sekiguchi, T., Yonezawa, A.: Byte Code Transformation for Portable Thread Migration in Java, In Proceedings of ASAMA'2000, Springer, Zuerich, Germany (2000) 16–28.
4. Truyen, E., Robben, B., Vanhaute, B., Coninx, T., Joosen, W., Verbaeten, P.: Portable Support for Transparent Thread Migration in Java, In Proceedings of ASAMA'2000, Springer, Zuerich, Germany (2000) 29–43.
5. Bouchenak, S.: Making Java Applications Mobile or Persistent, In Proceedings of COOTS'01, San Antonio, Texas, USA (2001).
6. Milojicic, D., Douglis F., Wheeler, R.: Mobility: Processes, Computers and Agents, Addison Wesley, Massachusetts, USA (1999).
7. Douglis F., Marsh, B.: The Workstation as a Waystation: Integrating Mobility into Computing Environment, In Proceedings of Third Workshop on Workstation Operating System (IEEE), Key Biscane, Florida, USA (1992).
8. Damm, M.: Byte Code Engineering, In Proceedings of JIT'99, Duesseldorf, Germany (1999).
9. Damm, M.: The Byte Code Engineering Library, <http://bcel.sourceforge.net/>, visited 12.05.01.
10. Sun Microsystems Inc: Java Object Serialization Specification. <ftp://ftp.java.sun.com/docs/j2se1.3/serial-spec.pdf>, visited 31.08.00
11. Sun Microsystems Inc: The Java Virtual Machine. <http://java.sun.com/docs/books/vmspec/index.html>, visited 12.05.01.
12. Sun Microsystems Inc: Java HotSpot Technology. <http://java.sun.com/products/hotspot/index.html>, visited 12.05.01.
13. Sun Microsystems Inc: Java Platform Debugger Architecture. <http://java.sun.com/j2se/1.3/docs/guide/jpda/>, visited 12.05.01.
14. Sun Microsystems Inc: Java2 Platform Standard Edition V1.3 Homepage. <http://www.javasoft.com/j2se/1.3/>, visited 12.05.01
15. Sun Microsystems Inc: Jini Technology. <http://java.sun.com/jini/>, visited 12.05.01.
16. Kargl F., Illmann, T., Weber, M.: CIA - Collaboration and Coordination Infrastructure for Personal Agents, In Proceedings of DAIIS'99, Helsinki, Finland (1999).

17. Illmann, T., Kargl, F., Krueger, T., Weber, M.: Migration in Java: problems, classifications and solutions, In Proceedings of MAMA'2000, Wollongong, Australia (2000).
18. Illmann, T., Kargl, F., Weber, M.: An Agent Cluster as integrative environment for personal agents, In Proceedings of ICIIS'1999, Washington, USA (1999).