Migration of Mobile Agents in Java: Problems, Classification and Solutions

Torsten Illmann
Department of Multimedia Computing
University of Ulm, Germany
torsten.illmann@informatik.uni-ulm.de

Frank Kargl
Department of Multimedia Computing
University of Ulm, Germany
frank.kargl@informatik.uni-ulm.de

Michael Weber
Department of Multimedia Computing
University of Ulm, Germany
weber@informatik.uni-ulm.de

Tilmann Krüger
Department of Multimedia Computing
University of Ulm, Germany
tilmann.krueger@informatik.uni-ulm.de

Abstract

In this paper, we examine migration techniques of mobile agents in Java. We identify the problems in Java technology, classify different migration styles and present possible solutions and related work. The proposed classification distinguishes between code migration, execution migration and data migration. The classification defines a partial order to compare different migration approaches. For realizing strong migration in Java, two solutions are proposed. On the one hand, a pre-processor adds all the necessary information for migration to the source code before compilation time. On the other hand, A JNI-based plugin for any virtual machine provides mechanisms to captures the agent's execution state. The restoration of the execution state is done by the plugin in combination with a byte code modifier which slightly changes the byte code of the agent.

1 Introduction

Agent technology is increasing more and more. Since society is moving steadily towards an information society [1], the need for personal assistants for searching, doing e-commerce and communicating 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 satisfyingly:

- Agent communication standards have to be specified, widely accepted and integrated into existing agent systems to provide interoperability and flexibility. For example FIPA (Foundation for Intelligent Physical Agents) [2] or MAF (Mobile Agent Facility) [3] are approaches which might fulfill that.
- Agent security raises questions about how to protect agents against hostile agents or hosts and hosts against hostile agents.
- Sophisticated, transparent, robust, easy-to-use, easy-to-install and easy-to-program agent systems have to be developed and accepted by others than the developers.
- Agent systems should meet requirements of the information society such as internet compatibility, mobile computing and e-commerce integration.

The realization of mobile agents is a suitable paradigm especially for mobile and distributed computing. Considering this in combination with the third aspect mentioned above, the need for the development of transparent mobile agents and, in consequence, transparent migration techniques is there. By migration we mean the 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 such as data and execution information of the agent must be transferred to and restored at the other location. Since transferring all these aspects is difficult and very expensive, one distinguishes between strong and weak migration. Whereas strong migration means the transfer of the agent's code and its complete state, weak migration can be defined as every migration that is not strong. A migration technique for code and object members is often called weak.

Modern agent systems are mainly implemented in Java because of its features platform independency, dynamic class loading, security issues and object-orientation. In this paper, we examine the problems of migrating mobile agents in Java. We identify and classify different kinds of migration aspects and build a more detailed differentiation between weak and strong migration. Finally, different solutions are proposed for these aspects and related work realizing these aspects is mentioned. A conclusion summarizes the proposed migration aspects and compares advantages and disadvantages of different solutions.

2 Problems

Strong migration means that the movement of the agent is totally transparent to the agent programmer. The agent may migrate at any point during execution and continues transparently at the remote location. To achieve this, code, object data, execution data, all references to objects or resources (locally or remote) and all threads created by the agent have to be successfully restored at the new location. The execution data encloses the call stack and the program counter (the position in the last method on the call stack) before migration. The call stack carries all methods that are still in execution and their cascading calling order. The object data includes object members and local variables of all methods being on the call stack.

The main problem for realizing a transparent migration in Java is that Java classes are being interpreted and executed securely by the virtual machine. Therefore only limited access to intern and native information like program counter, local stack frames and open resources of the current running threads is available. A classification of the problems differentiates and structures these problems in more detail. Later, solutions to these problems will be presented.

3 Classification

Figure 1 shows a classification of different migration aspects built according to the problems mentioned above.

At the top level, the classification consists of two orthogonal aspects: *code* and *state migration*. These refer to code transfer and state transfer of the agent. That is the usual way of describing strong migration. The state migration is composed of many 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 *thread migration*. *Stack*, *member* and *resource migration* form the data migration. Figure 1 additionally contains two weak migration alternatives to the program counter migration: *initialization* and *method migration*. In the following, the leafs of the classification hierarchy are described in detail:

Code migration means that all the code of the agent has to be transmitted from source to destination location. Since agents in general does not consist of only one class but hold references to other objects, their code has to be transmitted, too. Nevertheless, code migration should merely transmit code of referenced objects as needed. That means, Java core classes and classes of the agent framework are available at the destination site as well.

Program counter migration means that the execution at the destination location continues at the same point where it was interrupted before. Additionally, we assign the calling order of cascading executed methods (call stack) to this kind of migration. Since the agent autonomously decides to migrate in contrast to migration in load-balancing systems, there are pre-defined points in the code of the agent where a migration may occur. These points are method call usually called *move*, *jump*, *fork go* or *migrate*. Wasp [6], D'Agents [7] and Ara [8] are implementing the program counter migration.

Initialization migration. This migration technique is a weak alternative to the program counter migration. It means that the execution of an mobile agent always starts at the same initial point, e.g. an *init* method as used for Java applets. The agent programmer may overwrite this method in order to branch to different locations in the agent's code dependent on location and members. For that reason, the agent programmer has to store and restore the agent' s state on his own. Some agent systems extend this migration technique with event handlers being called before and after a migration occurs, e.g. in the Aglets system [4]. In any case, the migration process is not transparent any more. The realization of this kind of migration is quite common.

Method migration. As the initialization migration, this type of migration is also a weak alternative to the program counter migration. Here, the agent programmer specifies in which method the execution

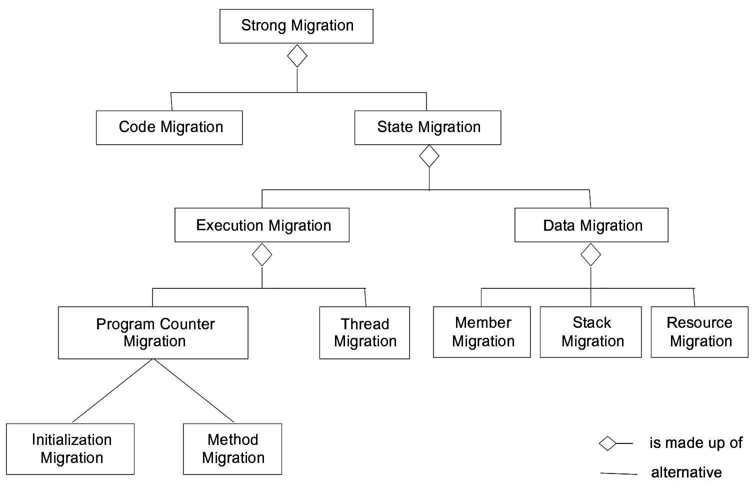

Figure 1. Classification of Different Migration Aspects

should continue after migration. Therefore, different entry points of the agent are indicated as different method definitions. This involves that the code is structured due to their migration entry points. An extension to this approach is that migration data may be passed as parameters to the next method being called, e.g. in the Voyager system [5].

Thread migration. To migrate an agent all child threads being created by the agent have to be considered. The correct restoration of these threads and their thread states (running, suspended, blocked, etc.) requires the restoration of all synchronization variables values (semaphores, condition variables) or monitor states as well as execution points within each thread. A realization of thread migration has to make use of all other aspects like program counter, stack, member and resource migration. The main difference between a single-threaded agent and a multi-threaded one is that the migration in a multi-thread environment is initiated by one thread only. Therefore, for all other threads the migration could happen at every point in their code and not only at pre-defined points. For that reasons, similar problems as in load-balancing systems apply.

Member Migration. The migration of the member variables of an agent is a very important aspect when migrating the state of an agent, because this is where an agent usually stores its query and result data. The member migration must be applied to all referenced objects (see code migration) and all concurrent threads (see thread 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. The systems [6], D'Agents [7] and Ara [8] are implementing this feature.

Resource migration. External resources that an agent may hold in the moment of migration are references to external objects such as remote objects (CORBA, EJB, RMI), open databases, messaging middleware components, open sockets or local files. Except local file access, all problems can be reduced to the migration of single and multicast socket streams.

We can now define *strong migration* as a migration technique which realizes code migration and strong state migration. Strong state migration demands strong execution migration and strong data migration. Strong execution migration is achieved by realizing strong program counter migration and strong thread migration. Strong data migration requires stack, resource and member migration. *Weak migration* is any migration technique that is not strong, i.e. if any aspect is missing or not strong. To compare two agent systems with regard to their migration technique a partial order ”stronger than” may be derived from this classification.

4 Solutions

This chapter suggests solutions of the previously defined migration aspects. We do not take into account solutions realizing *thread* and *resource migration* since we have not finished our research on these topics yet. Instead, we first mention migration aspects on which solutions are already provided by the Java core API. Next, we propose two ideas that realize strong migration in Java without regarding thread and resource migration. In general, strong migration can be implemented at three different levels of abstraction: at source code, at byte code and at interpreter level (virtual machine). Here, we introduce a solution for capturing the agent's state at source code level and another solution at both, the byte code and interpreter level. There are already implementations integrating migration in the source code [6] and the interpreter [7] and [8], but as far as we know there is none for byte code.

4.1 Solutions Provided by Java Core API

The Java core API already provides solutions for code and member migration.

Code migration is achieved by the concept of dynamic class loading and the possibility of implementing and integrating customized class loaders [9]. Therefore, a network classloader which transfers the Java byte code of all classes belonging to the context of an mobile agent can be realized easily. The classloader framework automatically takes over the tasks of transferring classes referenced by the agent. All Java agent systems have such a network classloader.

The Java serialization mechanism [10] already enables *member migration*. It allows to serialize and transmit objects via an object stream. An object stream may be used for all kinds of transmission (e.g. socket stream or file stream). Object instances being referenced by the serializing object and their members are serialized recursively, too. The only requirement to explicitly use the serialization API is that the agent and all its object members have to be serializable. Therefore, they have to implement the *java.io.Serializable* interface which actually has no methods to implement and the Java core uses a default implementation to serialize the object. A second possibility is to implement the *java.io.Externalizable* interface which has methods that specify how to explicitly serialize and de-serialize a specific instance of the class. Most Java agent systems use this mechanism to migrate the member variables of an agent.

The remaining migration aspects *program counter* and *stack migration* cannot be solved just using the Java core API since it does not allow access to the necessary intern information. The following two paragraphs show different solutions to these problems.

4.2 Instrumenting the Java Source Code

Our first approach modifies the source code of an agent and integrates all the functionality needed to do the migration. We developed a pre-processor *migratec* based on the Java compiler compiler ANTLR [11] which instruments the necessary migration statements. To realize stack migration two additional classes to store and restore local stack frames are required:

```
public class StackFrame ... {
  public void save(Object o);
  public Object restore();
}

public class State ... {
  public void push(StackFrame frame);
  public StackFrame pop();
  public void reset();
  public StackFrame next();
}
```

Local variables are stored and restored using the *StackFrame* class. The *State* class allows to *push* and *pop* stack frames of different methods and compound statements. The *reset* and *next* methods allow the iteration of the stack frame in the order they have been pushed. This iteration is needed to restore all stack frames after migration.

The following members and method either have to be implemented in the agent's base class (if existent) or instrumented in the agent's class. In any case, it does not have to be done by the agent programmer himself.

```
class MobileAgent {
  final static int NONE = -1;
  int migration = NONE;
  State state = new State();

  void migrate(Location l)
       throws MigrationException {...}
}
```

Each migration call within the code is assigned a unique index by the pre-processor. The migration counter *migration* holds the current migration index during a certain migration. After migration, this counter is reset to *NONE* again. The migration counter replaces the usual program counter. To actually cause a migration, the *migrate* method is provided. It throws a *MigrationException* which is a run time exception and thus does not have to be caught by calling methods. Only the initial calling method of the agent system (first one on the call stack) catches this exception and performs the migration procedure. The instrumentation of a simple method causing a migration. Line 5 illustrates the usage of these classes, members and the *migrate* method:

```
01  public int test() {
02    int a = 10;
03    String result;
04    result = otherMethod();
05    migrate(...);
06    return a;
07  }
```

The pre-processor inserts the following code:

```
01  public int test() {
02    int a;
03    String result;
04    if (migration != 1) {
05      a = 10;
06      result = otherMethod();
07      StackFrame frame = new StackFrame();
08      frame.save(a);
09      frame.save(result);
10      state.push(frame);
11      migration = 1;
12      migrate(...);
13    } else {
14      StackFrame frame = state.pop();
15      result = (String) frame.restore();
16      a      = (int)    frame.restore();
17      migration = NONE;
18    }
19    return a;
20  }
```

The pre-processor inserts an *if-else* statement (line 4, 13, 18) around the code from method entry to migrate call. It enables to execute the code in front of *migrate* and skips the code after migration. The migration counter is only set when causing migration (line 11) and unset after total restoration (line 17). The local variables are saved within a stack frame (line 7-10) before and restored after migration (line 14-16). Additionally, all variable declarations that will be executed before *migrate* must be moved to the top of the method (line 2,3), since access to them is needed in the *if* as well as the *else* branch of the condition.

If *migrate* occurs within a compound statement, especially loops, the code instrumentation is more complex. The local stack at the point before executing the compound statement also has to be stored and restored to enter the compound statement always in the same way. In the below example, the migration occurs within a *while* loop:

```
01 int i = 3;
02 int a = 1;
03 while (i > 0) {
04   i--;
05   a = a * 2;
06   int b = a;
07   migrate(...);
08   a = a + 1;
09 }
```

In the following example, code saving and restoring stack frames is only indicated by *state.push* and *state.pop* statements instead of mentioning every stack frame statement. The example is converted to:

```
01  int i;
02  int a;
03  if (migration != 1) {
04    i = 3;
05    a = 1;
06    ... // save i,a
07    state.push(frame);
08  } else {
09      StackFrame frame = state.next();
10      ... // restore a,i
11  }
12
13  while (i > 0) {
14    int b;
15    if (migration != 1) {
16      i--;
17      a = a * 2;
18      b = a;
19      ... // store i,a,b
20      state.push(frame);
21      migration = 1;
22      migrate(...);
23    } else {
24        StackFrame frame = state.pop();
25        ... // restore b,a,i
26        migration = NONE;
27    }
28    a = a + 1;
30  }
31
32  state.pop();
```

To process the loop three times, the *if-else* statement to execute or skip code according to the current migration state must be split into two parts: The first part (line 3,8,11) above the compound statement skips the initial statements after migration (line 4,5) and the inner block (line 15,23,27) is generated similar to the algorithm of the first example. To restore the correct execution point and local context within the loop, it is important that the while condition has initially always the same value. For that reason, the local stack of the outer statements must be saved (line 6,7) and restored (line 9,10) before entering the loop. Then, the loop is always entered with the *i*'s initial value of 3. And only the inner *else* branch (line 24-25) restores the real last value of *i*. Note that the stack frame of the outer statement in case of restoration is not popped but only read using *next* (line 9). The stack frame has to reside on the stack up to the termination of the loop (line 32).

Methods calls that possibly cause migrations within a method are treated the same way as compound statements doing it. Both must be associated with all migration indices possibly being caused somewhere in sub methods or sub statements. For every migration index that a method call or compound statement is associated with, the outer *if* condition (line 3) has to be extended with this index.

A recursive call of methods with intermediate migrations occurs if a method containing an explicit *migrate* statement calls another method being associated with exactly this migration index. The pre-processor inserts code around the recursive method to convert the migration index into a new one.

In order to proof the feasibility of this approach, we already implemented this approach in our agent system CIA [14] [15].

4.3 Extending the Virtual Machine and Instrumenting the Java Byte Code

This solution does not change the agent's code but slightly modifies the compiled Java byte code and extends the virtual machine (VM) [12].

We propose to extend the virtual machine with a JNI-based shared library. Based on the Java Virtual Machine Debugger Interface (JVMDI) [13] this plugin has access to intern execution data such as call stack and local data stack. Even the program counter of the current thread within a method can be read using JVMDI. Event handlers may be set e.g. for entering and leaving methods and the JVMDI allowing to execute single byte code instructions using the single step mode. Unfortunately it prohibits to set a new location for a thread. Therefore we instrument this in the byte code with a *tableswitch* statement using a byte code modifier. The tableswitch statement is a conditional branch to a pre-defined set of locations depending on the current value on the stack (similar to a *switch-case* statement in Java). We modify the byte code as shown in this example:

```
Method void myMethod (java.lang.String[])
  // >>>> BEGIN OF inserted code
  1 iconst_0  ; push 0 on stack
  2 istore 99 ; store it in slot 99
                (will be set by JVMDI plugin)
  3 iload 99  ; push slot 99 on stack
  4 tableswitch 0 to 2: default=29
        0: 29
        1: 54
        2: 68
  // <<<< END OF inserted code
  29 ...      ; original method code
  ...
  54 invokevirtual #5 <Method void migrate()>
  ...
  68 invokevirtual #5 <Method void migrate()>
  ...

  75 return   ; return to calling method
```

The instrumentation is performed by this algorithm being applied on all methods causing a migration:

1. Increment all locations within statements of the method being jumped to (e.g. in goto statements) by a constant offset o (the length of the total inserted statement).
2. Insert the defininition of a new local variable, e.g. *slot 99* at the beginning of the method. It will be used to control the succeeding tableswitch (in case of stack reconstruction this variable is set by the JVMDI). It's the replacement of the program counter.
3. Insert a tableswitch statement with the first element being the first original location of the method.
4. For each migrate call in the method at location l:
   - Add a new possible branch to the tableswitch statement at l + o.

The JVMDI plugin now uses these byte code instructions to re-establish the local call and data stack. The storage of the agent's state follows this algorithm:

1. Lookup current thread information.
2. Call *GetCurrentFrame* to get the id of current stack frame.
3. Call *GetFrameLocation* to get the frame's class, a method id and the current execution location
   - Class and method id are required to get the local variable table.
   - Execution location is used to find the current instruction number to calculate the value of the "pc" (*slot 99*).

4. Call *GetLocalVariableTable* to get the local variables of the current frame, each containing variable name, variable type and slot number.
5. For each variable: get value and push on agent's stack.
6. Call *GetCallerFrame* to get next frame and continue with 2. if not null.

Then the agent's state has been saved and may be serialized to the destination location. The rebuild of the agent's state works as follows:

1. Set up event handler for method entry events.
2. Create thread with name.
3. Wait for events from this thread.
4. On each event, the event handler receives thread, class, method and frame information. For this method in thread:
5. Activate single step mode.
6. Step two byte code instructions.
7. Call *GetLocalVariableTable* to get all variables.
8. For each variable: Pop from stack and set local variable to correct value including "pc" *slot 99*
9. Deactivate single step mode.
10. Thread continues to run with restored local variables and artificial program counter. The inserted tableswitch instruction jumps to the right instruction.

We are currently implementing this approach in the project CIA [14] [15].

5 Conclusion

In this paper, we identified the need and problems of sophisticated transparent migration techniques for mobile agents in Java. We further classified different aspects of migration techniques according to the traditional types of strong and weak migration. For each aspect, if suitable, a well-known agent system implementing this feature was mentioned. We proposed two possible implementations for strong migration in Java, one at the source code level and one at the byte code and interpreter level. Both solutions do not have to modify the user's virtual machine. We also mentioned other agent systems realizing similar approaches.

Our future work will further concentrate on strong migrations techniques. Next we next will examine the realization of the migration totally at the byte code level. Moreover, thread and resource migration will be considered as well.

References

[1] C. Erbas: On the Socio-Economic Impact of Information Revolution. IDPT00, Texas, USA (2000)

[2] FIPA: FIPA Homepage. http://www.fipa.org, visited 31.08.2000

[3] Crystaliz Inc, General Magic Inc, GMD FOKUS, IBM, TOG: OMG Joint Submission: Mobile Agent Facility Specification, ftp://ftp.omg.org/pub/docs/orbos/97-10-05.pdf, 1997

[4] IBM: Aglets Homepage. http://www.trl.ibm.co.jp/aglets/index.html, visited 31.08.2000

[5] Objectspace: Voyager Homepage. http://www.objectspace.com/voyager/prodVoyager.asp, visited 14.07.2000

[6] S. Fuenfrocken: Transparent Migration of Java-based Mobile Agents. MA98, Stuttgart, Germany (1998)

[7] R. Gray: Agent Tcl: A transportable agent system. CIKM95, Baltimore, Maryland (1995)

[8] H. Peine, T. Solpmann: The Architecture of the Ara Platform for Mobile Agents. MA97, Berlin, Germany (1997)

[9] Sun Microsystems Inc: Java2 Platform Standard Edition V1.3 Homepage. http://www.javasoft.com/j2se/1.3/, visited 31.08.00

[10] Sun Microsystems Inc: Java Object Serialization Specification. ftp://ftp.java.sun.com/docs/j2se1.3/serial-spec.pdf, visited 31.08.00

[11] T. Parr: ANTLR Homepage. http://www.antlr.org, visited 31.08.00

[12] Sun Microsystems Inc: The Java Virtual Machine. http://java.sun.com/docs/books/vmspec/index.html, visited 31.08.00

[13] Sun Microsystems Inc: Java Virtual Machine Debugger Interface Specification. http://jsp2.java.sun.com/j2se/1.3/docs/guide/jpda/jvmdi-spec.html, visited 31.08.00

[14] F. Kargl, T.Illmann, M. Weber: CIA - Collaboration and Coordination Infrastructure for Personal Agents. DAIS99, Helsinki, Finnland (1999)

[15] T.Illmann, F. Kargl, M. Weber: An Agent Cluster as Intergrative Environment for Personal Agents. ICIIS99, Washington, USA (1999)