Transparent Migration of Java-based Mobile Agents

Capturing and Reestablishing the State of Java Programs

Stefan Fünfrocken

Department of Computer Science, Darmstadt University of Technology,
Alexanderstr. 6, D 64283 Darmstadt, Germany
Email: fuenf@informatik.tu-darmstadt.de

Abstract. In this paper we describe a way to save and restore the state of a running Java program. We achieve this on the language level, without modifying the Java virtual machine, by instrumenting the programmer's original code with a preprocessor. The automatically inserted code saves the runtime information when the program requests state saving and reestablishes the program's runtime state on restart. The current preprocessor prototype is used in a mobile agent scenario to offer transparent agent migration for Java based mobile agents, but could generally be used to save and reestablish the execution state of any Java program.

1 Introduction

Mobile agents are programs that can move from one host to another. These programs can initiate their own transfer by executing a special instruction in their code. To migrate an agent, some state information of the agent program has to be saved and shipped to the new destination. At the target destination the agent program is restarted. Ideally, the moved agent (or program) can be restarted in exactly the same state and at the same code position as it was before migration. If migration exhibits this property, it is called *transparent* or characterized as *strong migration* [1]. If the program has to prepare its migration by explicitly storing its state in some variables and is started again at the new location, and if the programmer has to provide explicit code to read and reestablish the stored state, migration is called *non-transparent* or characterized as *weak migration*. Both mechanisms require the capturing of state information and the reestablishment of the saved state during restart.

Capturing and reestablishing the state of a running program is a well-known issue in different areas of computer science [9]. For example, it is used in distributed operating systems to provide load balancing functionality. In such a scenario, the state of a program in execution (i.e., the process state) is captured and sent to some other host with low load. The receiving host creates a local process that has exactly the same state as the process whose state was captured. State capturing can also be used to provide fault tolerance or persistence [6] in a distributed system. The state of programs or processes is captured at

regular intervals and is written to stable secondary storage. When the system restarts after a crash or regular system shutdown, the saved information is used to reestablish the processes and continue operation.

Our application scenario is the migration of mobile agents from one host to another. This differs from process migration for 'traditional' purposes (e.g., load balancing) in the sense that migration is initiated by the program itself and not by an external control instance. Although the developed mechanism is designed for the mobile agent scenario, it can be used by every Java program to save and load the runtime state.

To capture the state of a program one has to know what exactly comprises that state. The state can be divided into the following different parts: the code of the program, the data of the program (located in its variables), and the runtime information (consisting of the program counter, the call stack, and a few more items). One problem in capturing the state of a program is that the required information is located in different places: the program variables are accessible from within the program itself (i.e., on the language level), but in contrast to this, all the runtime information is located in lower hierarchy levels (e.g., the program counter, which is located in the process executing the program). The state capturing mechanism has to collect all that information, and consequently there has to be a way to extract the information from the system.

2 Capturing and Reestablishing State in Java

Java is an object oriented programming language. Thus the state of each Java program comprises the state of all the objects that exist at the time the capturing takes place, the method call stack resulting from the method invocations during the program execution, and the program counter. Since Java is an interpreted language that requires an interpreter (the Java Virtual Machine, or VM for short) to execute Java programs, the method call stack and program counter information is located in the interpreter and not in the process executing the interpreter. It would be sufficient to have access to the information inside the VM to capture the state of a Java program.

Currently the Java VM only supports the capturing of all object states, known as serialization [10], but it does *not* support the capturing of the method call stack which includes all local variable values of methods or the capturing of the program counter. Because of this, transparent migration of processes or mobile agents is not possible in Java so far.

There are systems [7, 8, 4] that provide the required state capturing of Java programs. However, they modified the Java VM. In contrast to this, we aim at a solution that does not require the modification of the VM or any underlying component. We found that it is indeed possible to capture the state of a Java program at the language level.

Our mobile agent scenario requires that the state capturing process is initiated by the program itself at the language level and not at some lower hierarchy level (i.e., from outside the program). To capture the state of a Java program, we

developed a preprocessor that instruments the user's Java code by adding code that does the actual state capturing, and reestablishes the state on restart at the target machine. We do this instrumentation by parsing the original program code using a Java based parser generated with the JavaCC-tool [5] from a Java 1.1 grammar. Our preprocessor uses and modifies the parse tree from which the new code is generated.

Since additional code introduces time and space penalties (see section 3.3), we only instrument the code where it is necessary and make sure that the additional code is executed only when necessary (i.e., when state capturing occurs).

2.1 Capturing the State

Java object serialization offers an easy, although rather inefficiently implemented way to dump the state of all Java objects that exist in the program. This state consists of the values of all variables (i.e., class and instance variables) of each object, which represent each object's internal state, and the information about the type of each object. By using object serialization, a large part of the information (all language level information) required to reestablish the program's state can be captured. What is missing, however, is information located in the virtual machine: the method call stack with the values of each method's local variables, and the current value of the program counter.

To capture the missing state information of the program, the preprocessor inserts code that saves the values of all local variables of methods and program counter related information. Consider the program in Figure 1 which defines local variables for the method `mymethod` of class `Myprogram` and uses state saving to save the value of new variables.

```
class Myprogram {
    // definition of variables
    ...
    public void mymethod(int i, real j, MyObject m){
        int k;
        Hashtable h;
        ...
        saveState();
        ...
        if ( k == 5 ) {
            Vector x = new Vector();
            ...
            saveState();
            ...
        }
        ...
        int v = 10;
        ...
        saveState();
        ...
    }
}
```

Fig. 1. Program using state saving

There are two things to keep in mind: First, local variables may be defined anywhere in a method's code. When defined in a block, the variable is visible in that block only (e.g., Vector x in the example). When the program requests state saving, only those variables can be saved that are visible at the current program position (only those variables are on the stack). Second, the inserted code that saves the values of the local variables should be executed only in the

case when the program initiates the state capturing process, which is done by calling a special method (`saveState`[1] in the example) a programmer can use. Our system provides this special method as an extension to the Java language.

Because of efficiency reasons we instrument only those methods that might initiate the state capturing process when called. Since this can happen at the end of a method call chain, we have to detect which methods initiate the state capturing process indirectly by calling other methods that initiate that process. We do this by fixed-point-iteration starting with the method that initiates state capturing.

Since the current state is located in the method call stack of the VM, we have to be able to traverse that stack and execute the state saving code our preprocessor inserted in each method that possibly might be on the stack during state saving. In addition, no further code of the program must be executed after the state saving process is initiated. To conform to both requirements, we use the Java error mechanism. Similar to Java exceptions, errors can be thrown and successively caught. When thrown, the normal flow of execution stops immediately. An error can be caught by a `catch` clause of a `try` statement. If not caught, the error is propagated up the method call stack. This is automatically done by the exception/error handling mechanism of the Java VM. We make use of this behavior to traverse the method call stack and save all local variables of each of the methods currently on the stack: The method that initiates the state saving process throws an error. Our preprocessor inserts an encapsulating try-catch statement for each method that might initiate the state saving. The code that saves the local variables of such a method is located in the catch clause of the inserted try-catch statement. After executing this code, the error is re-thrown thus propagating it up the stack to the calling method, which in turn catches the error leading to the execution of the variable saving code in this method. This is done until the stack is completely deconstructed. In this way, the state saving code is only executed when state saving is requested. Thus the code of class `Myprogram` shown above is transformed to the code depicted in Figure 2.

Using an error instead of an exception to realize state saving has the advantage that errors don't have to be declared in the method's signature.

To save all local variable values we use a special save object which is inserted by our preprocessor in the top level class of the Java program. In addition to that, all methods that might be part of the state saving process are passed the reference to this special object. Because of this, all relevant method signatures have to be instrumented. Furthermore, we provide the class with the code for the special method that initiates the state saving process. By this way, the method can be called as a local method. Unfortunately, this leads to problems in the inheritance tree when such methods are part of an interface which the class has to implement. So far, our solution to this problem is to generate a new interface incorporating the instrumented method signatures. We are currently

[1] In our mobile agent scenario this method is called `go`. For presentation of the general case we will use the more generic name `saveState` in this paper.

```
class Myprogram {
// variables are saved by normal serialization
...
public void mymethod(int i, real j, MyObject m){
    int k;
    Hashtable h;
    ...
    try{
        saveState();
    }
    catch ( Migration mig} {
        save(h); save(k);
        save(m); save(j); save(i);
        throw mig;
    }

    ...
    if ( k == 5 ) {
        Vector x = new Vector();
        ...
        try{
            saveState();
        }
        catch ( Migration mig} {
            save(x);
            save(h); save(k);
            save(m); save(j); save(i);
            throw mig;
        }
        ...
    }
    ...
    int v = 10;
    ...
    try{
        saveState();
    }
    catch ( Migration mig} {
        save(v);
        save(h); save(k);
        save(m); save(j); save(i);
        throw mig;
    }
    ...
}
```

Fig. 2. Transformed code of class Myprogram

investigating a solution that does not need to instrument the method signature in order to avoid this overhead.

After saving and deconstructing the stack, all state information is held in the special save object. Since this object is part of the top-level program class, its value can be saved by the normal object serialization mechanism. The final step in the state saving procedure is to initiate the serialization of the program's current object graph. This is done in the catch clause of the top-level object, inserted by the preprocessor, which also carries the save object.

Depending on the purpose of the state saving mechanism, the captured state (i.e., the serialization information) can be written to a file – in the case of checkpointing – or to a network socket – in the case of state transfer, as in mobile agents applications.

2.2 Reestablishing the State

Capturing the state of a running Java program is only half the way: it must also be possible to construct a process and program state from the saved state information that is equivalent to the state of the process and program from which the state was saved.

From the program's point of view the flow of control should be continued directly after the statement that initiated the state saving process. Since the Java VM provides no means to load a saved state, we have to do the re-establishment on our own. This task requires rebuilding the program's object graph and its objects states, rebuilding the method call stack, and reestablishing the values of the local variables of each method on the rebuilt method stack.

Rebuilding the object graph and the object states. Most of the program's state can be automatically reconstructed from the serialization information provided by the normal deserialization process Java offers. This process results in an object graph which exhibits the same connectivity and object state properties as the object graph that represented the program at serialization time. What is missing is the method call stack which is not automatically rebuild.

Rebuilding the method call stack. Since our save object (which keeps the relevant information) is part of the program's object graph, we can make use of that information to fill all the local method variables with the correct values once we recreated the method call stack. To do so, we just call again all relevant methods in the order they have been on the stack when the state capturing took place. To prevent re-execution of already executed method code, we have to skip all the code parts of each method which have been executed before the state capturing took place. But we do have to call the method that was next on the call stack during state saving.

To ensure this, we introduce code regions and an artificial program counter. The artificial program counter indicates for each modified method which code statements were already executed and therefore have to be skipped when the method call stack is rebuilt. It is not necessary to modify the artificial program counter after every instruction: successive statements that do not initiate state saving can be treated as a compound region and therefore the artificial program counter is updated only before and after such a compound region.

Region boundaries are introduced by the methods that may lead to state saving. Each such method forms a region by itself, which also encapsulates the state saving code. To skip the already executed code regions, each region is guarded by an `if`-statement that checks whether the artificial program counter indicates that the specific region has to be entered or skipped. Since the methods that might initiate a state saving process may be located in control flow statements that may possibly be nested, we have to introduce code regions for each control flow statement. The code regions of nested control flow statements are formed by applying the code region modifications of the outer statements before the inner statements.

Code regions for control flow statements introduce the problem that the control flow decision (which was already decided before state saving by evaluating the condition of the statement) cannot simply be decided again on restart by just re-executing the whole statement or re-evaluating the condition. Consider for example the while loop of Figure 3, where a method `checkresource` is called and might initiate state saving for the purpose of checkpointing.

Assume that `checkresource` has initiated state saving for `i = 3`. On restart, this loop should continue – after returning from `checkresource` – with `statement5` and `i=3`. This means to skip the initialization part, the evaluation of the condition, the first two iterations and to skip `i++; statement2` and `statement3` of the third iteration. Now assume that the state saving will be initiated in the last iteration (i.e., `i=4` in the condition). After evaluating the condition to `true`, the

```
// init i somewhere to 0
while (i<5) {
    i++;
    statement2;
    statement3;

    //method might initiate state saving
    checkresource(res[i]);

    statement5;
    statement6;
}
```

Fig. 3. Transforming loops

loop body is executed and `i` is set to `5` immediately. Because of this, it is not possible to restore the value of `i` immediately before the while loop to the value saved in the captured state (which yields `5`). This would result in skipping the pending execution of `statement5` and `statement6` on restart. Because of this, the loop condition has to be saved in a generated variable which is restored on restart for re-evaluation. This modification applies also to `do-while`, and `for` loops accordingly. `For` loops are transformed into `while` loops before the actual modification takes place.

Setting the values of local variables. When the program initiates state saving, all local variables of each method on the method call stack are saved in our save object. On restart, the method stack is rebuilt as described above. Now we have to set all local method variables to the correct values (i.e., the values that we saved in the save object). To achieve this we insert declaration code for each variable that sets the correct value. For a variable there are two possibilities to get its value: the original initial value as provided by the programmer in the case of a normal program start, and the value stored in the save object in the case of program restart. Since the actual value assignment is done in an `if` statement, all variables are initialized to an irrelevant default value, to satisfy the Java compiler. Since variable declarations are possible anywhere in the Java code, extra code is inserted at the position of each variable declaration, which also leads to correct variable visibility. Figure 4 code shows the transformation.

2.3 Threads

In Java it is not possible to transfer the state of running threads by the means of object serialization. Since every Java program is executed as a thread by the Java VM, our converter is able to save the state of a single thread. To save the state of all program threads, we simply use a new save object for each thread, that stores the method stack information of the associated thread. On restart, all threads that existed at the time of state saving are newly created and read their runtime information from their save object. Saving the runtime information of each thread is simple, but there are other problems that require attention: since

```
real i;
int j = 7;
Integer x = new Integer(5);
```

Is transformed into:

```
// init i
real i = 0.0;
if ( restart ) i = so.restore(i);

// init j
int j = 0;
if ( restart ) j = so.restore(j);
else j = 7;

// init x
Integer x = null;
if ( restart ) x = (Integer) so.restore(x);
else x = new Integer(5);
```

Fig. 4. Transformation for local variables.

threads run concurrently, one cannot predict at what time a thread that requests state saving will initiate state saving and in which state all other threads will be at this moment. From the point of view of all other threads, state saving could occur at every instruction. Because of this, we would have to be prepared to save the thread state after each instruction. That, however, would lead to the insertion of state saving code after each instruction which clearly is rather inefficient.

State saving occurs rarely, and usually only after certain amount of work has been done by the program. Because of this, we need the help of the programmer: he or she has the knowledge which program statements should be executed prior to state saving. We provide the programmer with a new method called `allowGo()`, that when called, indicates that the calling thread is ready to save its state. By this way, state saving occurs only if all running threads have called this new method or initiated state saving. This can be seen as barrier synchronization of all running threads. The new method checks if another thread requested state saving. If not, it returns immediately, otherwise it blocks the current thread until all running threads called the synchronization method. Accordingly, the method that requests state saving blocks the calling thread until all running threads allow state saving by a call to the new method.

3 Discussion

We provide a mechanism by which it is possible to collect and reestablish the state of a running program. In the application area of mobile agents this allows strong migration. We do this by a mechanical transformation of code written for transparent migration into code written for non-transparent migration. By this way we allow a programmer to program code that assumes transparent migration for a system that only provides non-transparent migration.

Of course, transparent migration is not a necessity: it is always possible to provide the same program functionality by explicitly coding a program specific migration mechanism on top of a non-transparent system. However, it is more convenient to use an automatic (i.e., transparent) mechanism. The question then is the cost (in terms of run time penalties and additional code) of this mechanism.

It is up to the programmer to decide, whether he or she is willing to pay the cost of our mechanism in order not to have to design and code the restart of the program explicitly. In our opinion our preprocessor offers the comfort of writing code for transparent migration at a reasonable cost.

3.1 Limitations of Full Transparency

Since the use of our converter is targeted towards the mobile agent scenario, some more general aspects concerning full transparency have to be considered: when moving a running program to another environment, this environment will usually differ from the former one. In contrast to scenarios where the saved program is restarted at the same machine (e.g., checkpointing), moving the saved state to a different machine before restarting it is inherent in the mobile agent scenario. However, changing the environment between saving and restarting introduces additional difficulties when providing fully transparent migration.

One aspect is the problem of references into the local environment such as file handles. In general, hiding the differences of environments from the program is difficult and hard (if ever) to achieve, and this is why transparent migration is often considered expensive [1]. For files this would require a distributed system layer that allows to open a file, disconnect temporarily from the file, and reconnect to the open file some time later from some other place. This might be possible for appropriate filesystems (cf. NFS, CORBA), but basically the same has to be done for all local system resources. This clearly is beyond the scope of our prototype system. Because of that, we offer - at a reasonable cost - 'almost' transparent migration: the flow of control starts right behind the statement that initiated migration, and the state of the migrated program is the same as the old program, except for the references into the environment. This means that we require the programmer to be aware of these 'environment problems'. Furthermore, we require that he or she tags variables that carry local references as 'transient', and that code for local resource accesses can react to errors that result from 'old' resource handles or from new handles that from the program's point of view represent a resource in an unexpected state.

3.2 Limitations of Language Level Instrumentation

Our preprocessor instruments code and our mechanism requires that all methods that might initiate state saving are instrumented. This causes a problem when using program libraries. Normally, libraries come without source code. Because of that, we cannot instrument the library code. In most cases this is not a problem, since these calls do not initiate state saving by themselves. But if the library call results in a callback to a program's method that can initiate state saving, the uninstrumented library code will prevent the correct state saving and restoring. Because of this, we require that callback methods do not initiate state saving directly or indirectly by a method call. This is not a real limitation because the callback method could raise a flag that indicates to initiate state saving to another method. This of course requires a second thread of execution.

The same problem arises when using dynamic loading of code during runtime (e.g., `Class.forName`). Since it is not possible to know at convert-time which code is loaded at runtime, the current prototype cannot handle this case.

3.3 Overhead

Instrumenting and inserting code introduces time and space overheads. Since we add code to the program, there is always a file size space penalty: the code is blown up. At run time there is also a memory space penalty: we have to store all local variables of methods. But at the same time, the method stack is deconstructed, so that the maximum required memory size does not grow.

The time penalty at compile time consist of the runtime of the preprocessor that has to instrument the original code and the time the compiler needs to compile the additional code. The time penalty at runtime can be divided into the additional runtime during normal program execution, the time that is needed to collect the program state after the state saving process is initiated, and the time that is needed to reestablish the program state before normal program execution continues.

The additional code that is always executed at runtime consists mainly of the code parts that *organize* the re-establishment of the control flow (in contrast to the code that actually *does* the re-establishment). For each variable initialization an `if`-statement has to be evaluated that checks whether the program is running after a restart or migration, or whether it is the first program start. Each code region is guarded by an `if`-statement that checks whether the guarded code segment has to be skipped while reestablishing the program state. In addition, the artificial program counter has to be modified at the end of each code region. The code that is responsible for saving the program state is not executed when the program runs in normal mode.

All overheads depend on how often in the code the state saving method is called, how many other methods call a method that initiates state saving, and how many local variables have to be saved. Since we instrument only those methods that could be on the stack while saving the state, the instrumentation overhead is as small as possible. Preliminary measurements for the overhead of the instrumentation during normal program execution shows the following results:

No	Tested code	Orig	Instr	Overhead
1	100 loops: saveState + Factorial	260ms	300ms	15%
2	100 loops: saveState + Factorial + IO	2553ms	2653ms	4%
3	100 loops: encapsulated saveState + Factorial	262ms	311ms	19%
4	100 loops: encapsulated saveState + Factorial + IO	2520ms	2695ms	7%

No program did initiate state saving, to avoid measuring the overhead resulting from actually saving the program state. As the tests show, the overhead resulting

from a saveState call is approximately 5 to 20 percent. We also measured the Bytecode blow up factor of the instrumentation:

No	Original	Instrumented	Blow-up factor
1	1047 Byte	4916 Byte	4.7
2	1333 Byte	4866 Byte	3.65
3	1175 Byte	5058 Byte	4.3
4	1322 Byte	5198 Byte	3.93

Note that in our preliminary tests almost all instructions of the original code have been instrumented, this is not the case in 'normal' agent code. Because of this we expect the blow up factor of realistic agent code to be smaller. Also, one should compare the instrumented code with code written for non-transparent migration providing the same functionality.

3.4 Related Work

To our knowledge, providing transparent migration or save and restart possibility for Java is done in a few other projects only [4, 7, 8], and providing it on the language level (i.e., without modifying the Java VM), is done in our project only.

Concerning transparent agent migration, one should mention Telescript [11], an interpreted, object oriented programming language that was designed for mobile agents by General Magic. Because the design of Telescript was tailored especially for mobile agents, the language had a lot of agent specific features (e.g., object ownership, read only object references) including transparent migration of agents. Transparent migration was implemented inside the Telescript interpreter (called engine), which did all the state saving, and did not provide migration of multiple (agent) threads (called processes). Unfortunately, General Magic stopped the development of Telescript.

Concerning state saving of programs or processes in general, there are a several systems [2, 6] that use state saving mechanisms to provide for example transparent process migration or persistence. Since these systems are especially designed for this task, the state saving mechanisms are coded into the operating system, interpreter, or runtime system itself in order to provide fast and efficient access to the state information of every process.

All approaches differ from our approach by the fact that they have full control over the implementation of the underlying system (i.e., the language interpreter or operating system). In contrast to this we aim at not modifying the Java interpreter at all.

3.5 Future Work

Currently the preprocessor has some limitations which we will eliminate in the near future. An interesting extension would be the possibility to transform the

byte code of a program or using the reflection classes Java offers to do some parts of the transformation at runtime. We plan to study the feasibility of both ideas.

4 Summary

We presented a way to allow Java programs to save their state in such a way that the program can be restarted with exactly the same state and at exactly the same code position. This is achieved by using code instrumentation and Java's object serialization mechanism. The code instrumentation of Java programs is done by a preprocessor that analyzes the original program and adds code that saves the current runtime state and makes it possible to reestablish that state on restart.

The instrumentation also supports state saving in the presence of multiple program threads, but in this case the cooperation of the programmer is required by using a special method to indicate that a thread is ready to save its state.

The current preprocessor prototype is used in the WASP project [3] to allow transparent migration for mobile agents written in Java.

References

1. Baumann J., Hohl F., Rothermel K., Straßer M., *Mole - Concepts of a Mobile Agent System*, to appear in: WWW Journal, Special issue on Applications and Techniques of Web Agents, 1998
2. Douglis F., Ousterhout J., *Transparent Process Migration: Design Alternatives and the Sprite Implementation*, Software - Practice and Experience (SPE), Volume 21, Number 8, August 1991, pp 757-785
3. Fünfrocken S., *How to Integrate Mobile Agents into Web Servers*, Proceedings of the WETICE'97 Workshop on Collaborative Agents in Distributed Web Applications, Boston, MA, June 18-20, 1997, pp 94-99
4. Gray R., *AgentTcl: A Transportable Agent System.*, Proc. CIKM'95 Workshop of Intelligent Information Agents, 1995
5. Java Compiler Compiler, `http://www.suntest.com/JavaCC/`
6. Mira da Silva M., *Mobility and Persistence*, Chapter in Mobile Object Systems. LNCS 1222, Springer-Verlag, 1997, pp 157-175
7. Peine H., Stolpmann T., *The Architecture of the Ara Platform for Mobile Agents*, In: Rothermel K., Popescu-Zeletin R. (Eds.), *Mobile Agents*, Proc. of MA'97, Springer Verlag, Berlin, April 7-8, LNCS 1219, pp 50-61
8. Ranganathan M., Acharya A., Sharma S., Saltz J., *Network-aware Mobile Programs*, Proceedings of Usenix'97, Anaheim, CA, 1997
9. Smith J.M., *A Survey of Process Migration Mechanisms*, Operating System Review, Volume 22, Number 3, July 1988, pp 28-40
10. Sun Microsystems, *Object Serialization Specification*, JDK Online Documentation 'docs/guide/serialization/spec', 1996, 1997
11. White J.E., *Telescript Technology: The Foundation for the Electronic Marketplace*, Whitepaper by General Magic, Inc, Sunnyvale, CA, USA