A Strong Migration Method of Mobile Agents Based on Java

Hong Wang
Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100080, P.R. China
sunzfc@jn-public.sd.cninfo.net

Guangzhou Zeng
Dept. of Computer Science
Shandong University
Jinan, 250061, P.R. China
gzeng@sdu.edu.cn

Shouxun Lin
Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100080, P.R. China
sxlin@ict.ac.cn

Abstract

In many CSCW systems, mobile agents are used to do some important work. However, mobile agents add a notable amount of complexity to all common problems of distributed systems, and cause a couple of new ones, such as location management, frequent disconnection and migration, etc. In this article, we focus on the technique for strong migration of mobile agents. Capturing and reestablishing the agent's state is the most important problem in strong migration of mobile agents. In this paper we describe a way to save and restore the state of a running Java-based agent. We achieve this on the language level, without modifying the Java Virtual Machine, by instructing the Java source programs with a preprocessor which is used in a mobile agent scenario to offer strong migration for mobile agents based on Java. The automatically inserted code saves the runtime information when the program requests state saving and reestablishes the program's runtime state on restart.

1. Introduction

Recently research on mobile agents has attracted much interest. Mobile agents facilitate the support of heterogeneous distributed systems, the reduction of network traffic and a smaller overall delay by encapsulating protocol steps into an agent. Today applications for mobile agents have existed in many fields such as information retrieval[1], management of large heterogeneous networks[2] and mobile computing[3][4]. However, applications are still evolving. In many CSCW systems, mobile agents are used to do part work[5]. However mobile agents add a notable amount of complexity to all common problems of distributed systems and cause a couple of new ones such as location management, frequent disconnection and migration, etc. In this article we focus on the technique for strong migration of mobile agents.

Mobile agents are actually programs that can move from one host to another. They can initiate their own transfer by executing a special instruction in their code. The agent's state can be divided into two parts: the runtime state and the date state. The runtime state refers to the control information for the agent running, consisting of the program counter, the stack, etc. The data state refers to the data information for the agent running, consisting of the agent basic attribute, the local variables, etc.

The agent migration can be divided into two kinds according to whether it transfers the runtime state or not: Strong Migration and Weak Migration[6]. Strong Migration also called Transparent Migration, means: When the agent requests migration, both the runtime state and the data state have to be saved and shipped to the destination together with the agent program code. At the target destination the moved agent can be restarted in exactly the same state and at the same code position as it was before migration. Weak Migration also called non-Transparent Migration, refers to: When the agent requests migration, only the data state is saved and transferred to the destination together with the agent code. At the target destination the moved agent can not be restarted at the same code position as it was before migration but from the program entrance. Both mechanisms require the capturing of the 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[7]. For example, it is used in distributed operating systems to provide load balance functionality. In such a scenario, the state of a program in execution (i.e., the process state) is captured and sent to some other hosts 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[8] in a distributed system. The state of programs or processes is captured at regular intervals and is written to a stable secondary storage. When the system restarts after a crash or a regular system shutdown, the saved information is used to reestablish the processes and the processes can continue to work. Our application scenario is the migration of mobile agents from one host to another. This differs from the process migration for 'traditional' purposes (e.g., load balancing) in the sense that a migration is initiated by an agent itself but not by an external control instance.

One problem in capturing the state of the 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. The state capturing mechanism has to
collect all that information from different places.
Java is an object-oriented programming language. It is
very suitable for agent programming. Firstly, it can
directly support agent programming by its attribute of
platform independence, security, static type test, distinct
type transformation, automatic storage administration, etc.
Secondly, Java provides a lot of flexible extended
mechanisms such as Object Serialization, Threads,
Reflection, etc. Although Java does not support state
capturing, process migration now, we can realize these
functions by using the flexible extended mechanisms in
Java[10].

This paper is structured as follows: Section 2
introduces three approaches for realizing Java-based agent
migration. Section 3 describes a method of state capturing
and reestablishment of Java-based agents, including the
method of state capturing, the method of state
reestablishment and the method in the scenario of multi-
threads. Section 4 discusses the limitation and the cost of
this method and provides concluding remarks.

2. Migration methods of mobile agents based
on Java

There are mainly three ways to address the problem of
capturing/restoring the state of mobile agents based on
Java.

In the first approach, which is called explicit
management, the programmer has to explicitly manage
backups in his agents. Managing a backup consists in
storing in a memory area the data managed on the stack
on which the agent depends. In Java, this memory area is
a Java object that belongs to the state of the agent. When
the agent state is restored, this backup object is explicitly
used by the agent code in order to restart the agent at the
point it was interrupted. For instance, in applications
which implement weak migration of mobile agents, the
programmer usually has to manage his own program
counter[10].

The two other approaches in which a transparent
mechanism is provided are called implicit management.
The mechanism is independent from the agent code and is
able to capture the agent state (data state and runtime
state). These two approaches differ by their
implementations: The first approach is implemented by
extending the JVM in order to make threads' state
accessible from agents based on Java. This extension
provides a facility for extracting the thread state and
storing it in a Java object which can be later sent to
another machine. This extension also provides a facility
for building a new thread initialized with a previously
captured state. Currently, there are some systems provide
the required state capturing of agents based on Java by
modifying the Java Virtual Machine[10][11]. The second
approach is implemented by pre-processing the source
code of the agent in order to insert statements that back up
the thread's state in a backup object. The main motivation
of this approach is not to modify the JVM. When an agent
requires a snapshot of the thread state, it just has to use the
backup object produced by the code inserted by the pre-
processor in the agent code. The restoration is achieved by
re-executing a different version of the agent code
produced by the pre-processor) which rebuilds the stack
and initialized the local variables with the values stored in
the back up object. This is the basic idea of our
mechanism.

3. Capturing and reestablishing state in Java

As Java is an object oriented programming language,
the state of each Java program comprises the state of all
the objects that exist at the time when the capturing takes
place, the method call stack resulting from the method
invocations during the program execution, and the
program counter. Java is an interpreted language that
requires an interpreter (the Java Virtual Machine, or VM
for short) to execute Java programs. In fact, the method
call stack and program counter information is located in
the VM. It would be sufficient to have access to the
information inside the VM to capture the method call
stack and program counter information. And the state of
all the objects that exist at the time when the capturing
takes place can be obtained by using Java object
serialization. So we found that it is indeed possible to
capture the state of a Java program at the language level.

3.1. Capturing the state

Java object serialization offers an easy way to dump
the state of all Java objects that exist in the Java-based
agent. 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 agent's state
can be captured. What is missing, however, is information
located in the VM: the method call stack with the values
of each method's local variables, and the current value of
the program counter. So, we developed a preprocessor. By
using it, we can instrument 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[12] from a Java 1.1 grammar. The preprocessor uses
and modifies the parse tree from which the new code is generated.

Since additional code introduces time and space penalties, 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). Now, a special method that is responsible for saving all local variables of methods and the value of the program counter is described as follows:

```java
protected void saveState() throws Migration
{
    save every local variables, throw mig;
}
```

Consider the program in Figure 1 which defines local variables for the method mymethod of class Myprogram and uses saveState() to save the value of new variables.

Since all 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 which the preprocessor inserted in each method on the stack. 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 method currently on the stack. The approach is:

1. The method that initiates the process of saving state throws an error.
2. The preprocessor inserts an encapsulating try-catch statement for each method that might initiate the state saving in order to save the local variables of this method.
3. After executing the code related to saving the local variables in try-catch statement, the error is re-thrown. Thus each method in the stack in turn catches the error leading to the execution of the variable saving code in this method.

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 do not have to be declared in the method’s signature.

To save all local variable values we use a special save object that is inserted by the preprocessor in the top level class of the Java-based agent. In addition to that, all methods that might be part of the state saving process are

```java
public void mymethod(int I, real j, MyObject m)
{  int k; //k can be given any value anywhere in this program. //
   Hashtable h;
   ...
   saveState(); //save the value of I, j, k, m, h
   if (k=5)
   ( Vector x = new Vector();  ... saveState(); ...)
   // save the value of x //
   ...
   int v=10; //v is also a local variable in this method. //
   ...
   saveState(); //save the value of v //
   ...
}
```

**Figure 1. Program using state saving.**

```java
class Myprogram{ // variables are saved by normal serialization

class Myprogram
{  // variables are saved by normal serialization

    public void mymethod ( int I, real j, MyObject m )
    {  int k;  //variables are saved by normal serialization

        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; }
    }
}
```

**Figure 2. Transformed code of class Myprogram.**
provided to this special object. By this way, the method can be called as a local method. Because of this, all relevant method signatures have to be instrumented. Unfortunately, this leads to problems in the inheritance tree when such methods are parts of an interface that the class has to implement. Our solution to this problem is to generate a new interface incorporating the instrumented method signatures.

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. Depending on the purpose of the state saving mechanism, 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 agent applications).

3.2. Reestablishing the state

Capturing the state of a running Java-based agent is only half the way. It must also be possible to construct a program state from the saved state information that is equivalent to the state of the agent 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. 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 stack. Most of the program's state can be automatically reconstructed from the serialization information provided by the normal de-serialization 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 that is not automatically rebuilt.

Rebuilding the method call stack. Since the 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. So we introduce an artificial program counter. The artificial program counter indicates for each modified method which statements are 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, since successive statements that do not initiate state saving can be treated as a compound one. Therefore the artificial program counter is updated only before and after such a compound statement region. Each compound statement region is guarded by an if-statement that checks whether the artificial program counter indicates that this compound statement region has to be entered or skipped.

Setting the values of local variables. The method call stack has been 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 agent start, and the value stored in the save object in the case of agent restart. To satisfy the Java compiler, all variables are initialized to a default value. The actual value assignment is done in an if statement. Figure 3 show the transformation.

3.3. Threads

In Java it is not possible to transfer the state of running threads by the means of object serialization. Since each Java program is executed as a thread by the Java VM, the 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, which 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 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. But it is impossible for us to be prepared to save the thread's state after each instruction, since it is rather inefficient.

So, we provide the programmer with a pair of a new

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

Figure 3. Transformation for local variables.
methods called setflag() and allowgo(). Any thread should execute allowgo() before it requests state saving. The method allowgo() 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 setflag(). By this way, state saving occurs only if all running threads have called the method setflag(). Actually, this can be seen as barrier synchronization of all running threads. We describe these methods as follows:

4. Summary

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

It is the most important that strong migration greatly lessens the programmer’s burden. Then the question is the cost (in terms of run time penalties and additional code). Inserting code introduces time and space overheads, such as the time penalty at compile time and the time penalty at run time that is needed to collect the program state after the state saving process is initiated and reestablish the agent state before normal agent execution continues. 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. It is up to the programmer to decide, whether he or she is willing to pay the cost of the mechanism in order to program conveniently.

In order to realize the agent strong migration, we have to consider other two questions. First is environment. When moving an agent to another environment, this environment will usually differ from the former one. However, changing the environment between saving and restarting introduces additional difficulties when providing strong migration[1]. For example, it is only the local system layer that can realize opening a file, disconnecting temporarily from the file, and reconnecting to the open file some time later. It is required that the system should accord with some criterion such as CORBA or tag variables that carry local references. Secondly, the preprocessor requires that all methods that might initiate state saving are instrumented. This causes a problem when using standard program libraries. Normally, standard program libraries come without source code. In most cases this is not a serious 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 callback method could raise a flag that indicates to initiate state saving to another method.

The essential difference between our mechanism of capturing and reestablishing the state of a Java-based agent and other mechanism is that our mechanism realizes these functions at the language level and not at some lower hierarchy level (i.e., from outside the program). Our mechanism supports agent’s strong migration even in the presence of multiple program threads. We think the price of our method is reasonable. Our work in future is eliminating some limitations in the preprocessor and using the reflection classes Java offers to do some parts of the transformation at runtime.

5. Acknowledgement

Our research is supported by the Chinese natural science fund (69975010) and Shandong province natural science fund (Y98G07103).

6. References

[8] Mira da silva M., Mobility and Persistence, Chapter in


