Pickling threads state in the Java system

S. Bouchenak[1], D. Hagimont[2]

SIRAC Project
INRIA, 655 av. de l'Europe, 38330 Montbonnot Saint-Martin, France

Internet: {Sara.Bouchenak, Daniel.Hagimont}@inria.fr

Abstract

Today, distributed object-based computing is closely linked with Java. The Java virtual machine is ported to most current operating systems and provides many services which help developing distributed object-based applications (e.g. RMI). In Java, code and data mobility is a very important aspect. Java provides a serialisation mechanism which allows the capture and restoration of objects' states and therefore the migration of objects between machines. It also allows classes to be dynamically loaded and therefore to be moved between nodes.

However, Java does not provide a mechanism for capturing and restoring a thread state. The stack of a Java thread is not accessible. Such a mechanism would allow a thread to be checkpointed or migrated between different nodes.

In this paper, we report on our experience which consisted in extending the Java virtual machine in order to allow the capture and restoration of a thread state. We describe the principles of the implementation of this extension and provide a performance evaluation.

1 Introduction

Distributed applications development is an important research direction in computer systems. In this context, the object paradigm has proven to be well suited to distributed applications development and the Java Virtual Machine [Gosling95] (JVM) is now considered as a reference platform. This is explained by the fact that today, the JVM runs on almost every operating system and can therefore be viewed as a universal virtual machine. The Java compiler produces byte code which is interpreted by the JVM and can therefore be run on every node.

The JVM provides many services for distributed applications development, the best known being Remote Method Invocation (RMI). But one of the most important aspects of Java (regarding distribution) is mobility. Java allows instances and code to be moved between

[1] INPG (Institut National Polytechnique de Grenoble)

[2] INRIA (Institut National de Recherche en Informatique et Automatique)

machines. Java provides a serialisation mechanism [Riggs96] which allows the capture and restoration of objects' states and therefore the migration of objects between machines. It also allows classes to be dynamically loaded and therefore to be moved between nodes. All these features led to the development of mobile agent systems, whose main advantage is to provide agent migration between machines [Chess95].

However, a mechanism is missing. Java does not provide any mechanism for capturing and restoring a thread state. The stack of a Java thread is not accessible. If one wants to capture the state of an application, Java only grants access to the application's objects and classes, the stack of the thread remaining inaccessible.

The consequence is that most of the mobile agent systems implemented on top of Java only provide *weak migration*. Whenever an agent moves, the agent's program restarts from the beginning on the arrival site. Thus, the programmer has to take care of that and must manage a state machine within its algorithm, which led to overly complex programs.

Such a capture/restoration mechanism has many applications, the main application being naturally thread migration between machines. Thread migration can be used to balance the load between nodes [Nichols87], to reduce network traffic by moving clients closer to the accessed servers [Douglis92] or to implement mobile agents [Chess95]. This mechanism also allows threads to be made persistent in order to implement a checkpointing service [Osman97].

In this paper, we report on our experience which consisted in extending the JVM with a service for thread state capture/restoration. A prototype has been implemented as an extension of the JDK. This service allows a thread state to be captured and later to be restored as the initial state of a new thread. We evaluated this prototype using different applications, including a mobile agent platform and a checkpointing mechanism.

The lessons learned from this experiment are that:

- it is possible to extends the Java Virtual Machine (JVM) with such a mechanism without re-designing the whole JVM.
- this feature can be provided as a generic mechanism which can be specialised by client applications (as the serialisation mechanism is).
- this implementation compares much favourably with the solution based on application code post-processing (code injection).

The rest of the paper is structured as follows. Section 2 presents the overall design choices. Section 3 describes the implementation principles. We present performance results in section 4 and we conclude the paper in section 5.

2 Overall design choices

We first present our motivations for implementing this mechanism within the JVM (compared to other approaches). Then, we describe the characteristics of the JVM which led us to our design.

2.1 Motivations and related work

There are mainly three ways to address the problem of capturing/restoring the state of Java threads.

In the first approach, which we call *explicit management*, the programmer has to explicitly manage backups in his programs. Managing a backup consists in storing in a memory area the data managed on the stack on which the application depends. In Java, this memory area is a Java object which belongs to the state of the application. When the application state is restored, this backup object is explicitly used by the application code in order to restart the application at the point it was interrupted. For instance, in applications using mobile agents platforms which implement weak migration (e.g. Aglets [IBM96], Mole [Baumann98] or Odyssey [GenMagic98]), the programmer usually has to manage his own program counter; the first statement of the program is a "switch" which branches to the point where the program must continue.

In the two other approaches, which we call *implicit*, a transparent mechanism is provided. The mechanism is independent from the application code and is able to capture the application state, including its thread state. The application can invoke a primitive which captures the thread state, including the contexts of all method invocations on the stack. These two other approaches differ by their implementations:

- The first implicit approach consists in pre-processing the source (or byte) code of the application in order to insert statements which back up the thread state (essentially local variables) in a backup object. The main motivation of this approach is not to modify the JVM. When an application 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 application code. In order to restore the thread state, data stored in the backup object are used to re-initialise the thread in the same state as at snapshot time. This restoration is achieved by re-executing a different version of the application code (produced by the pre-processor) which rebuilds the stack and initialises the local variables with the values stored in the back up object. The drawback of this solution is that it induces a significant overhead on application performance (due to inserted code) and on thread state restoration which requires a partial re-execution of the application. This solution has been implemented in the Wasp project [Fünfrocken98].
- The second implicit approach consists in extending the JVM in order to make threads' state accessible from Java programs. This extension must provide a facility for extracting the thread state and storing it in a Java object (which can be later stored in a file or sent to another machine). This extension must also provide a facility for building a new thread initialised with a previously captured state. This solution has been used in the implementation of the Sumatra mobile agent platform [Ranganathan97][3]. This is the approach we followed for two reasons:
  - It reduces the overhead on applications performance and reduces also the cost of the capture/restoration mechanism.
  - Since this mechanism has many applications, we believe that it is a basic functionality which must be integrated within the JVM.

Unlike the Sumatra mobile agent platform which supplies a mobility mechanism, our implementation provides a generic service intended for other uses than mobility, like checkpointing.

[3] However, we are not aware of any publication describing the implementation nor providing performance results.

When implementing this thread capture/restoration service, we wanted to provide an interface as close as possible to the one provided by Java to capture/restore objects state. Therefore, we did not aim at directly capturing (within the JVM) the set of objects which are accessible from the thread. Instead, the capture of a thread takes the form of a Java structure which includes a table of the Java references to the objects accessible from the stack. Therefore, the serialisation of the captured thread state will serialise these objects. The programmer is thus free to manage the snapshot of the application by specialising the serialisation methods of the objects managed in the application.

2.2 The Java Virtual Machine

We recall the structure of the Java virtual machine (JDK 1.1.3 [Sun99]) within which our extension has been implemented.

The JVM can support the concurrent execution of several threads [Lindholm96]. The JVM manages three main types of data structures:

- *The Java stack.* A Java stack is associated with each thread in the JVM. A new *frame* is pushed on the stack each time a method is invoked and popped from the stack each time a method returns. A method frame notably includes the method local variables and *registers* such as the top of the stack or the program counter.
- *The object heap.* There is a unique heap per JVM, shared between all the threads. The heap includes all the Java objects created during the lifetime of the JVM.
- *The method area.* The method area includes all the classes (and their methods) which have been loaded by the JVM. The method area is shared between all the threads.

To summarise, the context (or state) of a Java thread, illustrated in Figure 1, is composed of the three following data structures: the *Java stack* associated with the thread, a portion of the *object heap* including all the objects used by the thread and a portion of the *method area* including the classes used by the thread.

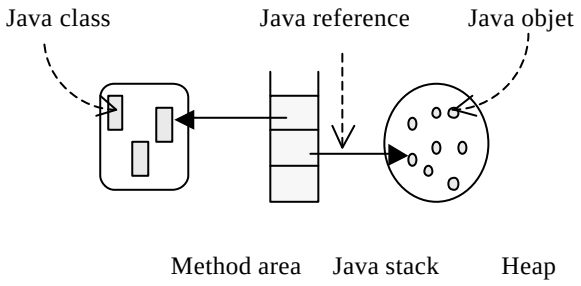

Figure 1 : Java thread context

3 Implementation principles of our mechanism

This section describes the implementation principles of our thread state capture/restoration mechanism. We first describe the interface of the mechanism and then describe the main steps of the implementation.

3.1 Interface of the mechanism

We provide a Java class called *MobileThread* [Bouchenak98], integrated into the *java.lang.* package. This class is a sub-class of the *Thread* class; it implements Java threads whose state can be captured and restored.

In addition, we provide another class added to the *java.lang.* package, called *ExecutionEnvironment.* This class defines the data structure which hosts a thread state. For security reasons, all the methods and variables of the *ExecutionEnvironment* class are private; only our capture/restoration mechanism can manipulate *ExecutionEnvironment* objects (except the array of class names which must be accessible to the class loader). Thus, captured threads states are always consistent.

The interface of the *MobileThread* class is given in Figure 2. A thread, of class *MobileThread*, has a variable called *ExecEnv* which is an *ExecutionEnvironment* object. This variable is initialised when the state of the associated thread is captured.

```
abstract class MobileThread
 extends Thread implements java.io.Serializable {

        public ExecutionEnvironment ExecEnv;
        public void extractExecEnv(boolean toStop, String[] args);
        public abstract void transferExecEnv(ExecutionEnvironment execEnv, String[] args);
        public static MobileThread integrateExecEnv(ExecutionEnvironment execEnv);

 }
```

Figure 2 : Interface of the *MobileThread* class

The *extractExecEnv* method allows the capture of the current state of a *MobileThread.* First, the thread execution is interrupted and the current thread state is captured and stored in the *ExecEnv* variable of the thread. The thread execution can be either resumed if the *toStop* parameter of the *extractExecEnv* method is false or definitively stopped if the *toStop* parameter is true. Finally, the *transferExecEnv* method is called (this is an upcall as explained below); the *args* parameter of the *extractExecEnv* is passed to the *transferExecEnv* method.

The *transferExecEnv* method describes how and where an extracted thread state is transferred. This method is abstract (its interface is defined but not its implementation) because its implementation depends on applications needs. If an application uses our mechanism for thread migration, the *transferExecEnv* method has to send the extracted thread state to a remote node. If the application uses our mechanism to build persistent threads, the *transferExecEnv* method has to store the extracted thread state in a persistent (non volatile) storage. Therefore, the *transferExecEnv* method must be implemented by the application programmer (or the designer of a mobile agents platform). The first parameter of this method, *execEnv*, is the

ExecutionEnvironment object in which the state is saved and the second parameter, *args*, allows receiving any information necessary to the transfer operation.

A last method is provided by our mechanism, the *integrateExecEnv* method. This method restores a thread state by creating a new thread and initialising its context with the *execEnv* parameter. The classes that are named in the *ExecutionEnvironment* object are supposed to be loaded prior to the invocation of this method. Finally, the execution of this newly created thread is resumed: it restarts at the point where it was interrupted.

3.2 Implementation of the mechanism

Two main services are provided by our mechanism: the capture of a thread state and the restoration of a thread state. We detail both of them.

Thread state capture consists in interrupting the thread during its execution and extracting its current state. The extraction amounts to build a data structure containing the current state of the thread. This data structure must contain all information necessary to restore a thread state (its Java stack, heap and method area). To build such a data structure, the Java stack associated with the thread must be captured and scanned to identify the Java objects and the Java classes which are referenced from the stack. This information can then be used to capture the heap and the method area associated with the thread. One of our motivations was to provide a generic service (such as object serialisation [Sun99]) which allows the implementation of various capture policies. Therefore, we rely on Java object serialisation and class loading features in order to capture respectively the object heap and the method area.

The data structure returned by the capture operation is composed of three parts:

- A byte array which includes the state of the stack, including relocation information which allows internal pointer (within the stack) to be relocated.
- An array which includes a description of all the object references stored in the stack. Each entry includes the object reference and its offset in the stack, which is used at restoration time in order to re-assign the reference in the stack (with the reference to the de-serialised object). Serialising this array will serialise the object heap associated with the thread. It is up to the layer which uses our mechanism to implement a consistent object serialisation policy and to adapt it to specific application needs, especially in the case of objects shared between several threads or open communication channels.
- A structure which includes a description of all the references to the method area. This structure includes an array of the class names for which a method frame is pushed on the stack. It is up to the layer which uses our mechanism to use this array in order to load (with a Java class loader) the required classes on the destination node. On the destination node, the references to these classes must be passed to the restoration operation, allowing it to update the references to the method area.

When this data structure has been extracted, it can be serialised and sent to another virtual machine on which it can be restored.

The most difficult issue for building this data structure is to have access to the types of the values that are pushed on the stack. Unfortunately, thread stacks in the JVM only include non-typed values. Therefore, it is not possible to determine whether a value on the stack is a Java reference, an integer or a boolean. However, the byte code instructions in Java are typed. Each instruction which pushes a value on the stack pushes a value of a given type. We extended the

JVM interpreter in order to manage a separate stack of types. For each value pushed on the stack, the byte code instruction is modified in order to push the type of the value on the stack of types. This provides us with the knowledge of the type of each value on each stack, and practically to find the object references that are stored in the stack.

Restoration consists in integrating the previously extracted state into the context of a new thread; the execution of this thread is resumed at the point it was interrupted. The new thread is initialised with a Java stack, a heap and a method area identical to those associated with the thread whose state was captured.

When the data structure is de-serialised, the object heap is reconstructed. The array of class names stored in the data structure allows the layer which uses the mechanism to load the classes (with a class loader) and to pass their references to the restore operation.

Notice that threads which are programmed using the *Thread* class execute on the standard Java interpreter and don't incur any performance overhead.

4 Experimentation and evaluation

This section describes the experiments performed with our thread state capture/restoration mechanism. It also provides performance results.

4.1 Experiments

These experiments use our thread state capture/restoration mechanism to implement thread migration, remote thread cloning and thread checkpointing. The source code of these experiments can be found in [Bouchenak98].

4.1.1 Thread migration

Java thread migration is the action of transferring a thread execution from a source JVM to a destination JVM, potentially located on different nodes. The thread execution in the destination JVM must restart where it was interrupted in the source JVM.

When migrating, a thread is interrupted in the source JVM and its current state is extracted. Then, this state is transferred to the destination JVM where it is integrated to a new thread. Finally, the thread in the source JVM is definitively stopped and the execution of the new thread in the destination JVM is restarted.

To experiment with thread migration, we first start two Java virtual machines. A *MobileThread* thread executes a program in the source JVM. In order to move, this thread calls the *extractExecEnv* method with the *toStop* parameter set to true and the *args* parameter containing the IP address and port number of the destination JVM. The *extractExecEnv* method captures the current thread state, calls the *transferExecEnv* method and stops the thread execution in the source JVM.

For this experiment, a *transferExecEnv* method is implemented, whose role is to send an *ExecutionEnvironment* object to a destination JVM. This method uses Java object serialisation to transfer the *ExecutionEnvironment* object.

Finally, an *ExecutionEnvironment* object is received by the destination JVM (using Java object de-serialisation) then the *integrateExecEnv* method is called. This causes the restoration of

the received thread state in a newly created thread. This new thread resumes the interrupted execution.

4.1.2 Remote thread cloning

Cloning a thread causes the creation of a new thread which has the same execution context as the original one. Remote thread cloning can be used to replicate an application execution on different nodes for fault tolerance reasons [Kim97].

Remote thread cloning is similar to thread migration with only one difference: in remote thread cloning, the thread execution on the source node is not stopped but resumed. To experiment remote thread cloning with our mechanism, we used the same program as thread migration. The only difference lies in calling the *extractExecEnv* method with an *args* parameter set to false.

4.1.3 Thread checkpointing

Thread checkpointing consists in taking periodical snapshots of the thread during its execution; these snapshots are stored in a non volatile storage. Thus, when a crash occurs, the last snapshot can be restored and the thread execution can be resumed at the point of this snapshot.

A thread state snapshot is implemented as follows. The *extractExecEnv* method captures the current thread state, the *transferExecEnv* method stores this state in a non volatile storage and the *integrateExecEnv* method restores a previously saved state. For this experiment, we have implemented a *transferExecEnv* method that stores an *ExecutionEnvironment* object in a file.

4.2 Evaluation

This section presents a performance comparison between our thread state capture/restoration mechanism and the mechanism implemented in the Wasp project [Fünfrocken98].

4.2.1 Capture/restoration cost

In order to measure and compare both implementations, we implemented a ping-pong thread between two machines (two Sun Ultra 1 workstations), using Wasp's and our mechanism. This allowed us to measure the average cost of a thread migration in both cases. Since we are mainly interested in the cost of thread state capture/restoration, we also measured the cost of thread state transfer: the difference between these two costs represents the cost of thread state capture/restoration.

The cost of a capture/restoration mechanism highly depends on the number, the size and the content of frames pushed on the thread stack. Thus, we measured the variation of this cost according to the number of frames on the stack. The results are given in Table 1.

Number of frames	1	5	20	50
Our mechanism (ms)	6	8	8	9
Wasp's mechanism (ms)	7	20	23	41

Table 1: Thread state capture/restoration costs

The results in Table 1 show that Wasp's mechanism is much more sensitive to the number of frames pushed on the captured stack. The difference between the performance of the two mechanisms is first due to the fact that our mechanism is integrated into the JVM while the Wasp's mechanism is implemented on top of the JVM. Also, Wasp's restoration mechanism requires a partial re-execution of the application, thus ensuring a performance overhead.

4.2.2 Overhead on applications performances

Both implementations (Wasp's and ours) of the capture/restoration mechanism may have a significant performance overhead on local thread execution for the following reasons:

- Wasp's mechanism injects code in the application code.
- For our mechanism, we have extended the Java interpreter for mobile threads (in order to have access to the types of values pushed on the stack).

In order to evaluate this overhead, we measured the execution time of a simple program in different cases:

- The program is executed by a thread of the *Thread* class, on a standard JVM (JDK 1.1.3).
- The program is executed by a thread of the *MobileThread* class, on our extended JVM.
- The program uses Wasp's mechanism and is executed on a standard JVM (JDK 1.1.3).

Type of JVM	Class of the thread	Execution time (ms)
standard JVM	*Thread* class	0,18
extended JVM	*MobileThread* class	1,85
standard JVM	Wasp class	25

Table 2: Cost of an execution of the factorial(100) function

Table 2 presents the execution costs of a program computing factorial of 100, in the different considered scenarios. Compared to the execution on a standard JVM (line 1), the execution using *MobileThread* performs 10 times slower (line 2). This overhead is due to the treatments we added to the Java interpreter in order to manage the stack of types. This overhead is significant but still much less than the one induced by the Wasp's mechanism (line 3) which performs 130 times slower than the standard execution. This last overhead is due to the code inserted into the application code.

5 Conclusion

While the Java virtual machine provides most of the basic services for objects mobility, it does not allow (in its current version) thread mobility. A Java program cannot access the Java stack associated with its thread.

Our goal was to tackle this deficiency by adding to the JVM a mechanism which allows thread state capture and restoration. There are mainly two ways to address this problem. The first approach, used by the Wasp project [Fünfrocken98], consists in pre-processing the application source code to insert statements which back up the thread state. The second consists in extending the Java virtual machine in order to make threads states accessible to Java programs. This is the approach we followed for two reasons. First, we believe it is a basic mechanism that should be integrated within the JVM as data and code mobility are. Second, it allows a more efficient implementation.

Our prototype was implemented by extending the JDK 1.1.3. We evaluated its functionality by applying it to thread migration and thread persistency. On the other hand, we measured the performance of our mechanism and compared it to Wasp's implementation. The measurements show that the cost of our mechanism is reasonable and that it outperforms the implementation based on code injection. The performance would be ameliorated if the mechanism was really integrated within the Java virtual machine by its constructors.

At the present time, this work is going on. We recently ported our mechanisms to Java 2 SDK (formerly called JDK 1.2). We are also considering a different implementation, based on a dynamic analyses of Java byte code in order to determine the types of the values on the stacks; this solution would avoid the modification of the Java interpreter and would lead to a JVM which allows thread capture without any overhead on code interpretation.

Acknowledgements. We would like to thank Xavier Rousset de Pina for his contribution to the work presented in this paper, and Stefan Fünfrocken for his help with the Wasp implementation. We are grateful to Sun Inc. for providing us the source code of the JDK.

Bibliography

[Baumann98] J. Baumann, F. Hohl, M. Straβer et K. Rothermel. *Mole – Concepts of a Mobile Agent System*. WWW Journal, Special issue on Applications and Techniques of Web Agents, 1998. http://mole.informatik.uni-stuttgart.de/

[Bouchenak98] S. Bouchenak. *MobileThread API.* http:// sirac.inrialpes.fr/~bouchena/JavaThread/

[Chess95] D. Chess, C. Harrison et A. Kershenbaum. *Mobile Agents: Are They a Good Idea ?*. IBM Research Report. IBM Research Division, T.J. Watson Research Center, Yorktown Heights, New York, march 1995. http://www.research.ibm.com/iagents/publications.html

[Douglis92] F. Douglis et B. Marsh. *The Workstation as a Waystation : Integrating Mobility into Computing Environments*. The 3rd Workshop on Workstation Operating System (IEEE), april 1992.

[Fünfrocken98] S. Fünfrocken. *Transparent Migration of Java-based Mobile Agents (Capturing and Reestablishing the State of Java Programs)*. Proceedings of Second International Workshop Mobile Agents 98

(MA'98), Stuttgart, Allemagne, september 1998.
http://www.informatik.tu-darmstadt.de/~fuenf/

[GenMagic98] General Magic. *Odyssey mobile agents*.
http://www.genmagic.com/

[Gosling95] J. Gosling and H. McGilton, The Java Language Environment: a White Paper, Sun Microsystems Inc., 1995.
http://java.sun.com/docs/white/

[IBM96] IBM Tokyo Research Labs. *Aglets Workbench : Programming Mobile Agents in Java*. 1996.
http://www.trl.ibm.co.jp/aglets/

[Kim97] J. Kim, H. Lee et S. Lee. *Replicated Process Allocation for Load Distributed in Fault-Tolerant Multicomputers*. IEEE Transactions on Computers, pages 499-505, 1997.

[Lindholm96] T. Lindholm et F. Yellin. *Java Virtual Machine Specification*. Addison Wesley, 1996.

[Nichols87] D.A. Nichols. Using Idle Workstations in a Shared Computing Environment. Proceedings of the 11th ACM Symposium on Operating Systems Principles, pages 5-12, ACM 8-11, November 1987.

[Osman97] T. Osman et A. Bargiela. Process Checkpointing in an Open Distributed Environment. Proceeding of European Simulation Multiconference (ESM'97), June 1997.

[Ranganathan97] M. Ranganathan, A. Acharya, S. D. Sharma et J. Saltz. *Network-aware Mobile Programs*. Proceedings of the USENIX Annual Technical Conference, Anaheim, California, 1997.
http://www.cs.umd.edu/~acha/

[Riggs96] R. Riggs, J. Waldo, A. Wollrath, K. Bharat. *Pickling State in the Java System*. USENIX Conference on Object-Oriented Technologies (COOTS), Ontario, Canada, 1996.

[Sun99] Sun Microsystems. *Java 2 Platform*, Sun Microsystems.
http://java.sun.com/products/jdk/