Pickling threads state in the Java system

Sara Bouchenak

Third European Research Seminar on Advances in Distributed Systems (ERSADS'99), Madeira Island - Portugal. April 23rd-28th 1999

Pickling threads state in the Java system

S. Bouchenak

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

Internet: Sahra.Bouchenak-Khelladi@inria.fr

Abstract

Java provides a serialization mechanism which allows the capture and restoration of objects' state 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 preliminary results of its evaluation.

1 Introduction

One of the most important aspects of Java, regarding distribution, is mobility. Java provides a serialization mechanism which allows objects state to be captured in one JVM (Java Virtual Machine) and restored in any other JVM. Java also provides a dynamic class loading facility, allowing code to be dynamically moved between nodes. All these features led to the development of mobile agent systems, whose main advantage is to provide agent migration between machines.

However, one mechanism is missing. Java does not provide any facility for accessing the state of a thread (essentially its stack). 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. This explains why most of current mobile agent systems only implement weak migration: the state of an agent does not include the state of its thread, and on arrival on a node, the agent always restarts from the beginning.

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

2 Overall design choices

In this section, we present our motivation for implementing this mechanism within the JVM (compared to other approaches) and describe the Java thread state structure.

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 safe memory area the

data on the stack which will be lost when the application state is captured. In Java, this memory area is a Java object. 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] 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 generic mechanism is provided. The mechanism is independent from the application code and is able to capture the application state, including its thread state. These two other approaches differ by their implementations.

The first consists in pre-processing the source 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-initialize the thread in the same state as at snapshot time. This restoration operation updates variables on the stack according to the thread state in the back up object. The drawback of this solution is two-fold: it induces a significant overhead on application performance (due to inserted code) and thread state restoration requires a partial re-execution of the application. This solution has been implemented in the Wasp project [Fünfrocken98].

Another 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 initialized with a previously captured state. This solution has been used in the implementation of the Sumatra mobile agent platform [Ranganathan97]. This is the approach we followed for two reasons:

- It reduces the overhead on applications performance and reduces also the cost of the 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.

2.2 The Java thread state

The JVM can support the concurrent execution of several threads [Lindholm96]. The context (or state) of a Java thread is composed of the three following data structures: the *Java stack* associated to the thread, the *heap* and the *method area* respectively consisting of the objects and the classes used by the thread.

A new *frame* (method block) is pushed on the Java 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.

3 Implementation principles of our mechanism

This section describes the implementation principles of our thread state capture/ restoration mechanism. This mechanism was integrated into the Java virtual machine by extending the JDK 1.1.3 [Sun].

3.1 Principles of our capture/restoration mechanism

The capture/restoration mechanism is designed to interrupt a thread during its execution and **extract** its current state and to **integrate** 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 extraction operation 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 data structure, the Java stack associated with the thread

must be captured and scanned to identify the Java objects references and the Java classes references. These references are used to capture the heap and the method area associated with the thread.

The second service provided by our mechanism is the integration operation. This operation aims at creating a new thread which is initialized with the thread state previously saved in a data structure. The new thread is initialized with a Java stack, a heap and a method area identical to those associated with the thread whose state was captured.

3.2 Implementation design

Our extension provides a Java class called *MobileThread* [Bouchenak98], integrated into the *java.lang.* package. This class is a sub-class of the *Thread* class; it characterizes Java threads whose state can be captured and restored. In addition, we provide the *ExecutionEnvironment* class, added to the *java.lang.* package. This class defines the data structure which hosts a threads state.

The *MobileThread* class provides an instance variable called *ExecEnv* and three methods respectively called *extractExecEnv*, *transferExecEnv* and *integrateExecEnv*.

The *ExecEnv* variable is an *ExecutionEnvironment* object which is initialized when the state of the associated thread is captured.

The *extractExecEnv* method allows the capture of the current state of a *MobileThread* thread. Firstly, the thread execution is interrupted and the current thread state is captured and stored in the *ExecEnv* variable of the thread. Then, the thread execution can be either resumed or definitively stopped. Finally, the *transferExecEnv* method is called (this is an upcall as explained below).

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 non volatile storage. Therefore, the *transferExecEnv* method must be implemented by the application programmer.

The *integrateExecEnv* method restores a thread state by creating a new thread and initializing its context with an *ExecutionEnvironment* object. Finally, the execution of this newly created thread is resumed: it restarts at the point where it was interrupted.

4 Experimentation and evaluation

Some preliminary experiments were performed with our extended Java virtual machine. These experiments use our thread state capture/restoration mechanism to implement thread migration which can be used for load balancing and remote thread cloning, thread checkpointing which can be used for fault tolerance [Kim97]. The source code of these experiments can be found in [Bouchenak98].

Some measurements were done to evaluate our thread state capture/restoration mechanism and compare it to the mechanism implemented by the Wasp project [Fünfrocken98]. These measurements aimed at evaluating the capture/restoration cost and the overhead of this mechanism on applications performances.

The cost of a capture/restoration mechanism highly depends on the Java stack size of the thread. Thus, we measured the variation of this cost according to the stack size (by varying the number of *frames* on the stack). A comparison between our mechanism and the Wasp's one shows that Wasp's mechanism is much more sensitive to the size of the captured stack. The difference between the performances of the two mechanisms is due to the fact that our mechanism was integrated into the Java virtual machine (a native implementation) while the Wasp's mechanism was implemented on top of the virtual machine.

Our extension of the Java virtual machine does not involve any overhead on applications that do not use the capture/restoration mechanism. But using this mechanism induces performance overhead on application execution for both implementations (the Wasp's and the ours). Our overhead is significant but still much less than the one induced by the Wasp's mechanism because this last mechanism injects Java code in the application code while our mechanism was implemented by extending the Java interpreter for mobile threads (this implementation was principally native).

5 Conclusion

Our implementation aimed at adding to the JVM a mechanism that allows thread state capture and restoration. We have extended the Java virtual machine in order to make threads state accessible by Java programs; this approach provides a generic mechanism intended for other uses than mobility and allows a more efficient implementation.

We evaluated the functionality of our prototype by using it to experiment thread migration and thread persistency. On the other hand, the preliminary measurements show that the costs of our mechanism are reasonable compared to those of its pair (the Wasp's mechanism).

At the present time, this work is going on. Complementary measurements are in progress in order to compare our mechanism with other implementations and to experiment this mechanism with "real" applications.

Bibliography

[Bouchenak98] S.Bouchenak-Khelladi. *Mécanismes pour la Migration de Processus – Extension de la Machine Virtuelle Java.* Rapport de Magistère d'Informatique, Université Joseph Fourier, Grenoble, France, 1998.
URL:http:// sirac.inrialpes.fr/~bouchena

[Chess95] D. Chess, C. Harrison et A. Kershenbaum. *Mobile Agents: Are They a Good Idea ?.* IBM Research Division, T.J. Watson Research Center, Yorktown Heights, New York, march 1995.
URL:*http://www.cs.dartmouth.edu/?agent/papers/chapter.ps.Z.*

[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.

[GenMagic98] General Magic. *Odyssey Web Site*. URL : http://www.genmagic.com/agents/

[IBM96] IBM Tokyo Research Labs. *Aglets Workbench : Programming Mobile Agents in Java*. 1996.
URL : 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, april 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.
URL :http://www.javasoft.com/products/jdk/1.1/docs/

[Sun] Sun Microsystems. *JDK 1.1 Documentation*, Sun Microsystems.
URL : http://java.sun.com/products