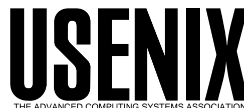
USENIX Association

Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS '01)

San Antonio, Texas, USA
January 29 - February 2, 2001



© 2001 by The USENIX Association All Rights Reserved For more information about the USENIX Association:
Phone: 1 510 528 8649 FAX: 1 510 548 5738 Email: office@usenix.org WWW: http://www.usenix.org
Rights to individual papers remain with the author or the author's employer.
Permission is granted for noncommercial reproduction of the work for educational or research purposes.
This copyright notice must be included in the reproduced paper. USENIX acknowledges all trademarks herein.

Making Java Applications Mobile or Persistent

Sara Bouchenak*

SIRAC Laboratory (INPG-INRIA-UJF)
INRIA Rhône-Alpes, 655 av. de l'Europe, 38330 Montbonnot St Martin, France.
Sara.Bouchenak@inria.fr
** INPG (Institut National Polytechnique de Grenoble)*

Abstract

Today, mobility and persistence are important aspects of distributed applications. They have many fields of use such as load balancing, fault tolerance and dynamic reconfiguration of applications. In this context, the Java virtual machine provides many useful services such as dynamic class loading and object serialization which allow Java code and objects to be mobile or persistent. However, Java does not provide any service for the mobility or the persistence of control flows (threads), the execution state of a Java program remains inaccessible.

We designed and implemented new services that make Java threads mobile or persistent. With these services, a running Java thread can, at an arbitrary state of its execution, migrate to a remote machine or be checkpointed on disk for a possible subsequent recovery.

Therefore migrating a Java thread is simply performed by the call of our *go* primitive, by the thread itself or by an external thread. In other words, the migration or the checkpointing of a thread can be initiated by the thread itself or by another thread.

We integrated these services into the JVM, so they provide reasonable and competitive performance figures without inducing an overhead on JVM performance. Finally, we experimented a dynamic reconfiguration tool based on our mobility service and applied to a running distributed application.

Keywords: mobility, persistence, migration, checkpointing, recovery, Java, thread, JVM

1. Introduction

Today, mobility and persistence are important aspects of distributed applications and have several fields of use [Milojicic99] [Ambler99]. Application mobility can be used to dynamically balance the load between several machines in a distributed system [Nichols87], to reduce network traffic by moving clients closer to servers [Douglis92], to dynamically reconfigure distributed applications [Hofmeister93], to implement mobile agent platforms [Chess95] or as a machine administration tool [Oueichek96]. Application persistence can be used for fault tolerance [Wojcik95] or for application debugging.

Distributed applications development is an important research direction in computing systems. In this context, the object paradigm has proven to be well suited to distributed applications development and the Java Virtual Machine (JVM) is now considered as a reference platform [Gosling96]. The Java compiler produces *bytecode*, an intermediate code that is interpreted by the JVM. Today, the JVM is ported on almost every platform and can therefore be viewed as a universal machine.

In order to facilitate the development of distributed applications, the JVM provides several services [Sun00a] among which:

- Object serialization. The serialization service allows the transfer of Java objects between several nodes or the storage of objects on disk.
- Dynamic class loading. The dynamic class loading service enables the transfer of Java code between several nodes.

Therefore, Java provides useful services for the mobility and the persistence of code and data. However, Java does not provide any service enabling the mobility and the persistence of applications during their execution. Thus, if a running Java application migrates to a new location, only using object serialization and dynamic class loading, the execution state of the application is lost. In other words, when arriving on its new location, the migratory application can access to its code and its re-actualized data but it has to restart the execution from the beginning. Consequently, the provided Java services are not sufficient for either enabling dynamic load balancing of distributed Java executions or allowing the state of running applications to be checkpointed and then recovered.

We designed and implemented new services that make Java threads, i.e. executions, mobile or persistent. With these services, a running Java thread can, at an

arbitrary state of its execution, migrate to a remote machine or be checkpointed on disk for a possible subsequent recovery.

Our *java.lang.threadpack* Java package provides several primitives, among which *go* performs thread migration, *store* is used for thread checkpointing and *load* for thread recovery. Therefore migrating a Java thread is simply performed by the call of the *go* primitive, by the thread itself or by an external thread. In other words, the migration or the checkpointing of a thread can be initiated by the thread itself or by another thread.

We integrated these services into the JVM, so they provide acceptable performance figures without inducing an overhead on JVM performance. Finally, we experimented with a prototype implementation a dynamic reconfiguration tool based on our mobility service and applied to a running distributed application.

The rest of this paper consists of three main parts. We first describe our service for capturing/restoring Java thread state in section 2 and then present the services of thread mobility and thread persistence in section 3. In sections 4 and 5, we respectively present performance figures and describe some experiments that we performed with our services. Finally, we discuss related work and present our conclusions and future directions in section 6 and 7.

2. Thread state capture/restoration service

Both services allowing the mobility and the persistence of Java threads are based on a common service: a thread state capture/restoration service. We first describe the representation of a thread state in the JVM and then present the design principles of our capture/restoration service and its implementation details.

2.1. Java thread state

The JVM can support the concurrent execution of several threads [Lindholm96]. The state of a Java thread is illustrated by figure 1, it consists of three main data structures:

- *The Java stack*. A Java stack is associated with each thread in the JVM. The Java stack consists of a succession of *frames* (see figure 2). A new frame is pushed onto the stack each time a Java method is invoked and popped from the stack when the method returns. A frame includes the local variables of the associated method and the partial results of this method. The values of local variables and partial results may be of several types: integer, float, Java reference, etc. A frame also contains registers such as the program counter and the top of the stack.
- *The object heap*. The heap of the JVM includes all the Java objects created during the lifetime of the JVM. The heap associated with a thread consists of all the objects used by the thread (objects referenced in the thread's Java stack).
- *The method area*. The method area of the JVM includes all classes (and their methods) that have been loaded by the JVM. The method area associated with a thread contains the classes used by the thread (classes whose some methods are references by the thread's Java stack).

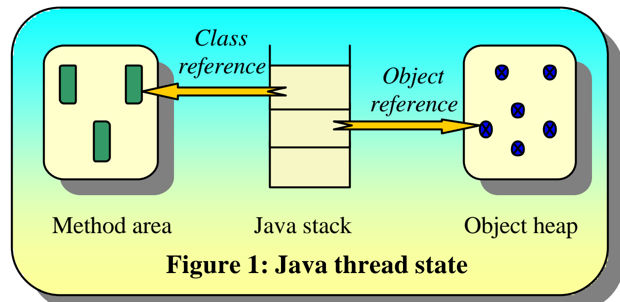


Figure 1: Java thread state

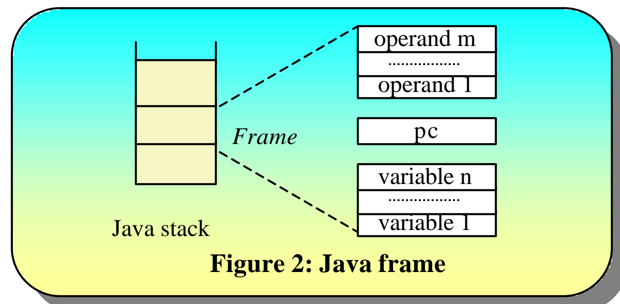


Figure 2: Java frame

2.2. Design of the capture/restoration service

Here are the design principles and the design decisions of our Java thread state capture/restoration service.

2.2.1. Design principles

The thread state capture/restoration service enables, on the one hand, the *capture* of the current state of a running thread, and on the other hand, the *restoration* of a previously captured state in a new thread: the new thread starts running at the point at which the execution of the previous thread was interrupted.

Thread state capture consists in interrupting the thread during its execution and extracting its current state. The extraction amounts to build a data structure (a Java object) containing all information necessary for restoring the Java stack, the heap and the method area associated with the thread. To build such a data

structure, the Java stack associated with the thread is scanned in order to identify the objects and the classes that are referenced from the stack. After state capture, the resulting data structure can be serialized and sent to another virtual machine in order to implement mobility or it can be stored on disk for persistence purpose.

One of our motivations was to provide a generic service which allows the implementation of various capture policies. Consequently we rely on Java object serialization and dynamic class loading features in order to capture respectively the heap and the method area.

The restoration of a thread consists first in creating a new thread and initializing its state with a previously captured state. After that, the Java stack, the heap and the method area associated with the new thread are identical to those associated with the thread whose state was previously captured. Finally, the new thread is started, it resumes the execution of the previous thread.

2.2.2. Design decisions

There were mainly two problems for designing a Java thread state capture/restoration service. The first issue is to have access to the state of Java threads, a state that is internal to the JVM. The second issue is that the state of Java threads is not portable on heterogeneous architectures.

2.2.2.1. Non accessible state

The state of Java threads is internal to the JVM. This state is not accessible by Java programs and can therefore not be directly captured. In order to allow the capture of threads state, we extended the JVM and externalized the state of Java threads.

2.2.2.2. Non portable state

Unlike the heap and the method area that consist of information portable on heterogeneous architectures (Java objects and Java classes), the Java stack is a native data structure (C structure). The representation of the information contained in the Java stack depends on the underlying architecture. The thread state capture service must translate this non portable data structure (C structure) to a portable data structure (Java object).

Translating the Java stack into a portable data structure consists more precisely in translating the native values of local variables and partial results (figure 2) into Java values. This translation requires the knowledge of the types of the values. But the Java stack does not provide any information about the types of the values it contains: a four bytes word may represent a Java reference as well as an *int* value or a *float* value. The thread state capture service must recognize the types of the values contained in the Java stack. We propose two approaches for type recognition:

- *Type recognition at runtime*. The first approach consists in recognizing the types during runtime. The type information is built in parallel with thread execution. The type information is actualized each time a bytecode instruction is interpreted by the thread because the bytecode instructions are typed and are applied to particular types [Lindholm96]. Therefore, at state capture time, the type information is available. The drawback of this approach is that it induces an overhead on application performance.
- *Type recognition at capture time*. The second approach consists in recognizing the types at capture time. The type information is built by analyzing the bytecode to determine the execution path of the thread. This analyze is similar to the bytecode verifier algorithm [Lindholm96]. This approach avoids any overhead on application performance but causes a latency due to type information building.

We first implemented the approach based on type recognition at runtime [Bouchenak00] and then implemented the approach based on type recognition at capture time*. In this paper, we focus our attention on the design principles of our services and do not tackle the implementation details.

2.3. Implementation of the capture/restoration service

Our Java thread state capture/restoration service was integrated to the Java 2 SDK (formerly called JDK 1.2) [Sun00a]. Our new Java package, called *java.lang.threadpack*, provides many classes such as the *ThreadState* class whose instances represent the state of Java threads and the *ThreadStateManagement* class that provides the necessary features for capturing and restoring Java threads state.

Figure 3 illustrates a part of the application programming interface (API) of the *ThreadStateManagement* class. The *capture* method allows the capture of the current state of a Java thread, the captured state is returned as a result of this method, as a *ThreadState* object. Symmetrically, the *restore* method creates a new Java thread, initializes its state with the *ThreadState* argument, starts the new thread and returns it as a result of the method. The new thread resumes the execution of the thread whose state was

* The evaluation presented in section 4 concerns the first approach

previously captured and passed as an argument of the *restore* method.

java.lang.threadpack
Class ThreadStateManagement
public final class **ThreadStateManagement** extends Object
The ThreadStateManagement class provides several useful services for the capture and restoration of Java thread states.

Method Summary	
static ThreadState	capture(Thread thread) Captures the state of the thread argument and returns it as a ThreadState object.
static Thread	restore(ThreadState threadState) Creates a new Java thread, initializes it with a previously captured state and starts its execution.
static void	captureAndSend(Thread thread, SendInterface sndItf, boolean toStop) Captures the state of the thread argument and sends it (to a remote node or to the disk) by calling the sendState method of the SendInterface interface.
static Thread	receiveAndRestore(ReceiveInterface rcvItf) Receives the state of a Java thread by calling the receiveState method of the ReceiveInterface interface, creates a new Java thread, initializes it with the received state and starts its execution.

Figure 3: Interface of the capture/restoration service

The *captureAndSend* and *receiveAndRestore* methods are generic and can specialize the capture and restoration operations to application needs. Besides capturing the state of a Java thread, the *captureAndSend* method allows the programmer to specify the way the captured state is handled: the captured state can for example be sent to a remote machine for a mobility purpose, it can be stored on disk in the case of application persistence, etc. The specialization of the handling of the captured state is specified by the second argument of the *captureAndSend* method. In fact, this argument implements our *SendInterface* interface and so provides a *sendState* method that is called by our *captureAndSend* method, just after the capture of the thread state. The third argument of the *captureAndSend* method is a boolean that specifies if the thread whose state is captured is stopped or resumed. This argument is for example set to *true* in the case of thread migration and is set to *false* for remote thread cloning.

Symmetrically, the *receiveAndRestore* method specifies the way a thread state is received before it is restored: the state can for example be received from a remote machine, it can be read from disk, etc. The specialization of the way the thread state is received is possible thanks to the argument of the *receiveAndRestore* method: this argument implements our *ReceiveInterface* interface and so provides a *receiveState* method that is called by our *receiveAndRestore* method, just before the restoration operation.

3. Thread mobility and thread persistence services

Besides our system service for capturing/restoring the state of Java threads, we provide higher-level services for the mobility and the persistence of Java applications.

Making an application mobile is the action of moving an application, during its execution, from one node to another: the application starts running, on the new node, at the point at which the execution was interrupted on the first node. Therefore, making a Java application mobile consists in making the underlying Java thread mobile. In the case of a multi-threaded application, the whole group of threads has to be moved.

Making a thread mobile is the action of capturing the current state of the thread, sending this state to a target machine and restoring the state in a new thread on the target machine: the new thread resumes the execution in the state left by the original thread.

In the same way, application persistence consists first in saving the current state of a running application on stable storage (disk). The saved state can subsequently be restored in order to resume the execution of the application. Therefore, making a Java application persistent consists in making the underlying Java thread(s) persistent.

Making a thread persistent is, first, the action of capturing the current state of the thread and saving it on disk and then, the ability of restoring the saved state in a new thread: the new thread resumes the execution of the previous thread.

Our *MobileThreadManagement* class belongs to the *java.lang.threadpack* package and provides services necessary for the mobility of Java threads. Figure 4.a illustrates a part of the application programming

interface of this class. The *go* method allows the transfer of a running Java thread to a Java virtual machine identified by an IP address and a port number. And the *arrive* method enables the reception of a Java thread coming from a machine specified by an IP address and a port number.

java.lang.threadpack
Class MobileThreadManagement
public final class **MobileThreadManagement** extends Object
The MobileThreadManagement class provides several useful services for making Java threads mobile.

Method Summary	
static void	go(Thread thread, String tagetHost, int targetPort) Moves the execution of the thread argument to the machine specified by the host name and the port number arguments.
static Thread	arrive(String sourceHost, int sourcePort) Receives a thread from the machine specified by the host name and the port number arguments.

Figure 4.a: Service for the mobility of Java threads

```
public static void go(Thread thread, String targetHost,
                      int targetPort) {

MySender sndItf = new MySender(targetHost,
targetPort) ;
ThreadStateManagement.captureAndSend(thread,
sndItf) ;

}
```

```
public static Thread arrive(String sourceHost,
                            int sourcePort) {

MyReceiver rcvItf = new MyReceiver(sourceHost,
                                   sourcePort) ;
return
ThreadStateManagement.receiveAndRestore(rcvItf) ;

}
```

Figure 4.b: Implementation of mobility service

The *go* and *arrive* methods are implemented using respectively the *captureAndSend* and *receiveAndRestore* generic methods (see section 2.3). The *go* method is implemented as follows:

- The *go* method calls the *captureAndSend* method (figure 4.b).
- The *captureAndSend* method is adapted using an instance of the *MySender* class.
- The *MySender* class implements the *SendStateInterface* interface and therefore provides a *sendState* method, this method aims at establishing a connection to a machine and sending the *ThreadState* object using serialization (figure 4.c).

```
class MySender
  implements SendInterface {

String host ;
int port ;

MySender(String host, int port) {
  this.host = host ;
  this.port = port ;
}

public void sendState(ThreadState state) {
// Send state to <host, port>.
…
}
}
```

```
class MyReceiver
  implements ReceiveInterface {

String host ;
int port ;

MyReceiver(String host, int port) {
  this.host = host ;
  this.port = port ;
}

public ThreadState receiveState () {
// Receive a state from <host, port> and return
it.
…
}
}
```

Figure 4.c: Implementation of mobility service

The *arrive* method is implemented as follows:

- The *arrive* method calls the *receiveAndRestore* method (figure 4.b).
- The *receiveAndRestore* method is adapted using an instance of the *MyReceiver* class.
- The *MyReceiver* class implements the *ReceiveStateInterface* interface and therefore provides a *receiveState* method, this method aims at establishing a connection to a machine and receiving a *ThreadState* object using de-serialization (figure 4.c). The classes associated with this *ThreadState* object are received relying on the Java dynamic class loading service.

We can also imagine *go* and *arrive* methods that rely on the Wireless Application Protocol instead of IP in order to perform thread migration between JVM installed on wireless hosts [WAPFactory00].

java.lang.threadpack
Class PersistentThreadManagement
public final class **PersistentThreadManagement**
extends Object
The PersistentThreadManagement class provides several useful services for making Java threads persistent.

Method Summary	
static void	store(Thread thread, String fileName) Saves the state of the thread argument in the file specified by the name argument.
static Thread	load(String fileName) Restores the execution of a Java thread from the state stored in the file specified by the name argument.

Figure 5: Service for the persistence of Java threads

In the same way, the *PersistentThreadManagement* provides several services for the persistence of Java threads. A part of its application programming interface is illustrated by figure 5. The *store* method saves the current state of a Java thread in a file specified by a name and the *load* method restores a Java thread from a state saved in a file identified by a name. These two methods are also implemented using our *captureAndSend* and *receiveAndRestore* generic methods.

Finally, the *MobileThreadManagement* and *PersistentThreadManagement* classes are two possible adaptations of our generic service of Java thread state capture/restoration. In the same way and for a particular application, our generic service can be adapted to build tools that meet application's needs.

4. Evaluation

This section first presents the performance figures of our thread state capture/restoration service. The cost of migrating a Java thread between two machines and the cost of checkpointing/recovering a thread are then presented. Finally, a comparison between the results of benchmarking our extended JVM and the standard JVM is described. Our evaluation environment is as follows:

- JDK 1.2.2,
- Solaris 2.6, Sun Ultra-1 (Sparc Ultra-1 167 MHz),
- Ethernet 100Mb/s.

4.1. Basic costs

The time spent in capturing/restoring a Java thread state depends on the size of the state at capture time. The size of a Java thread depends on the number and the size of the frames pushed onto the Java stack associated with the thread. In the following, we focus our attention on the influence of the number of frames on the cost of our services. In order to vary the number of frames pushed onto thread's Java stack, we used a recursive program (the factorial function).

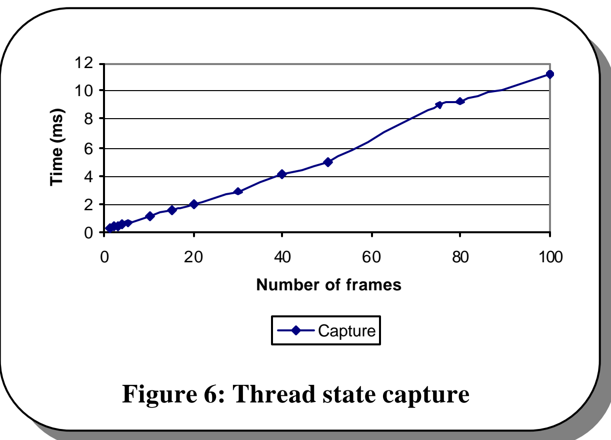


Figure 6: Thread state capture

Figure 6 describes the variation of the cost of a thread state capture operation according to the number of frames on thread's Java stack at capture time. The cost of a capture operation is less than 1 ms when the number of frames is lower than 10. This cost reaches 2 ms when the number of frames is 20 and 9 ms when the number of frames is 80.

Figure 7 presents the cost of a thread state restoration operation when varying the number of frames on thread's Java stack at capture time. The curve shows that the cost of a restoration operation is less than 1 ms when the number of frames is lower than 80.

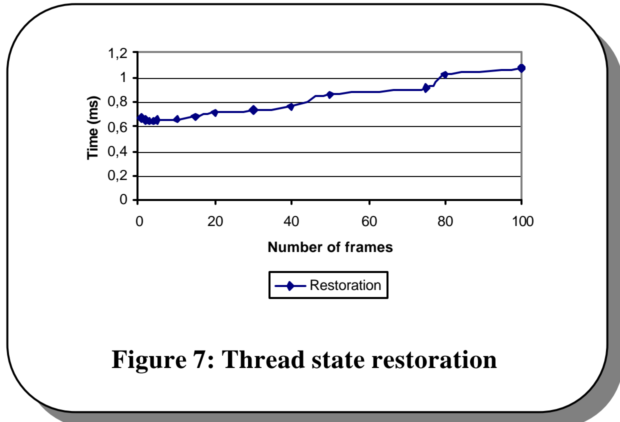


Figure 7: Thread state restoration

Finally, the costs of the capture and the restoration of Java thread's state are acceptable, especially in the case of threads with few frames on the Java stack.

4.2. Evaluation of thread migration

We measured the cost of a Java thread migration based on our thread mobility service. In figure 8, the solid curve represents the variation of the cost of a Java thread migration operation according to the number of frames on thread's Java stack at migration time. The dotted curve represents the cost of a thread state transfer between two machines when varying the number of frames on thread's Java stack.

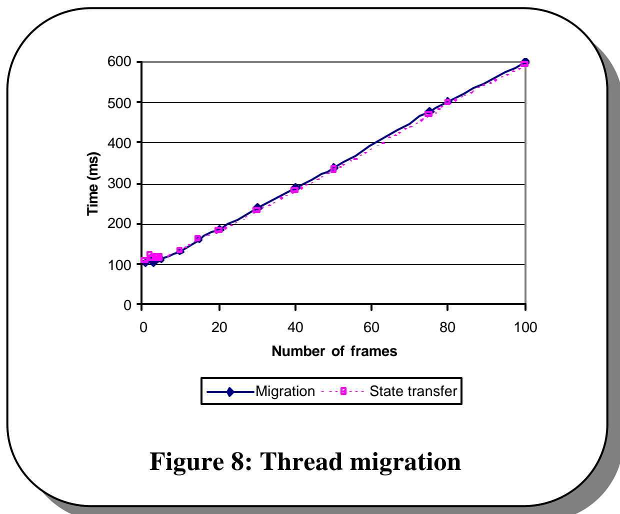


Figure 8: Thread migration

The cost of a thread migration linearly varies from 100 ms to 600 ms when the number of frames on the thread's stack is between 1 and 100. This cost may seem significant but it is mainly due to the cost of thread state transfer, as shown by the two almost superimposed curves. In fact, thread migration consists in capturing thread's state, sending this state to a destination machine and restoring the state in a new thread on the destination machine. So state transfer represents 98% of the total cost of thread migration.

On the other hand, the transfer of a thread state to a destination machine consists in first serializing the state object in order to translate the object graph to a byte array, then transferring the resulting array of bytes over the network to the destination machine and finally de-serializing the byte array on the destination machine in order to rebuild the object graph. The state transfer time can partly be reduced using Java externalization rather than serialization. Externalization allows the application programmer to write its own object transfer policy by only saving information necessary for rebuilding object graphs. Externalization may be until 40% faster than serialization [Sun00c].

4.3. Evaluation of thread checkpointing and recovery

Besides thread migration, we also measured the cost of checkpointing a running Java thread and saving its state on disk and the cost of recovering an execution from a state previously stored on disk. In figure 9, the solid curve represents the variation of the cost of a Java thread checkpointing operation according to the number of frames on the thread's Java stack at checkpointing time. The dotted curve represents the cost of writing a thread state on disk according to the number of frames on the thread's Java stack. In figure 10, the solid curve represents the cost of a Java thread recovery and the dotted curve illustrates the cost of reading a thread state from disk.

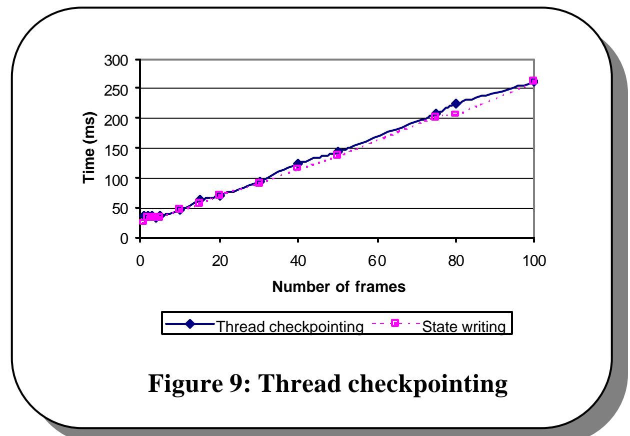


Figure 9: Thread checkpointing

We notice, on the one hand, that the cost of thread checkpointing and the cost of thread recovery increase linearly when the number of frame on the thread's Java stack increases. On the other hand, 97% of the time of thread checkpointing is spent in writing thread's state on disk and 99% of the time of thread recovery is spent in reading thread's state from disk. As explained in section 4.2, the costs of serialization/de-serialization can be decreased by using externalization. The

performance of thread checkpointing can also be improved by performing asynchronous disk writing.

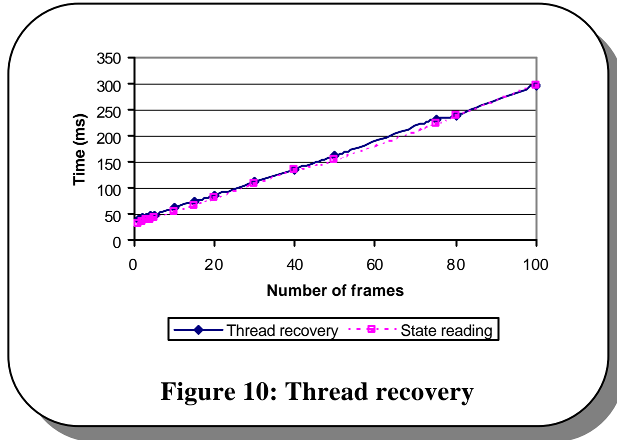


Figure 10: Thread recovery

4.4. Benchmarking the JVM

Our thread mobility and thread persistence services were integrated into the JVM. In order to evaluate the performance of our extension of the JVM, we compared them to the performance figures of the standard JVM. We used two benchmarks: the Embedded CaffeineMark 3.0 general Java benchmark [Pendragon99] and the SciMark 2.0 numeric Java benchmark [Pozo00]. In order to measure JVM performance, the benchmarks were run by disabling JIT compilation.

Tests	Standard JDK 1.2.2	Extended JDK 1.2.2
Overall Score	**1913**	**1913**
Sieve	1447	1457
Loop	3314	3311
Logic	5403	5265
String	1922	1985
Float	1130	1134
Method	871	858

Figure 11: Benchmarking the JVM with Embedded CaffeineMark 3.0

Embedded CaffeineMark consists of 6 tests: finding prime numbers, loops, logic tests, String and Float tests and method calls. The overall score is the geometric mean of the individual scores, i.e., it is the 6th root of the product of all the scores. The score for each test is proportional to the number of times the test was executed divided by the time taken to execute the test, i.e. a higher number represents a better score. Figure 11 presents the results of benchmarking the standard JDK 1.2.2 and our extended JDK 1.2.2. It shows that our extension does not induce any loss of performance on the JVM.

Tests	Standard JDK 1.2.2	Extended JDK 1.2.2
Composite Score	**8.9891**	**9.0977**
FFT (1024)	7.4484	7.6727
SOR (100x100)	18.6662	18.7242
Monte Carlo	0.9157	1.1166
LU (100x100)	7.4484	7.6727
Sparse matmult (N=1000, nz=5000)	7.7224	7.5683

Figure 12: Benchmarking the JVM with SciMark 2.0

SciMark 2.0 is a Java benchmark for scientific and numerical computing. It consists of five computational kernels: Fast Fourier Transform (FFT), Jacobi Successive Over-relaxation (SOR), Monte Carlo integration, dense LU matrix factorization and Sparse matrix-multiply. The kernels are chosen to provide an indication of how well the underlying JVM performs on applications utilizing these types of algorithms. The problems sizes are purposely chosen to be small in order to isolate the effects of memory hierarchy and focus on internal JVM and CPU issues. This benchmark reports a composite score in approximate Mflops (Millions of floating point operations per second). Figure 12 shows the performance figures resulting from benchmarking the standard JDK 1.2.2 and our extended JDK 1.2.2. It shows that our extension does not induce any loss of performance on the JVM.

5. Experimentation

In this section, we describe two experiments that use our mobility service. The first experiment shows the usefulness of strong mobility and the second experiment shows how to build a dynamic reconfiguration tool on top of our mobility service. Finally, we discuss some implementation issues and solutions.

5.1. Strong mobility: Mobile recursive *Fractal*

Two degrees of application mobility can be distinguished: *weak mobility* and *strong mobility* [Fuggetta98]. With weak mobility, only data state information and application's code are transferred. Therefore, on the new location, the mobile application has its actualized data but restarts execution from the beginning. With strong mobility, the code of the application and the state of data and execution are transferred: the application on the destination location

resumes the execution at the point where it was interrupted on the source location.

The usage of weak or strong mobility depends on application's needs. Let's consider a recursive Java application. The recursive calls are translated by a succession of frames on the Java stack associated with the underlying thread. How is this application made mobile?

- Weak mobility does not consider the state of execution (thread's state), so frames previously pushed onto the Java stack are lost after the transfer and the execution restarts from the beginning.
- Strong mobility captures the state of execution and allows the execution to be resumed after the transfer.

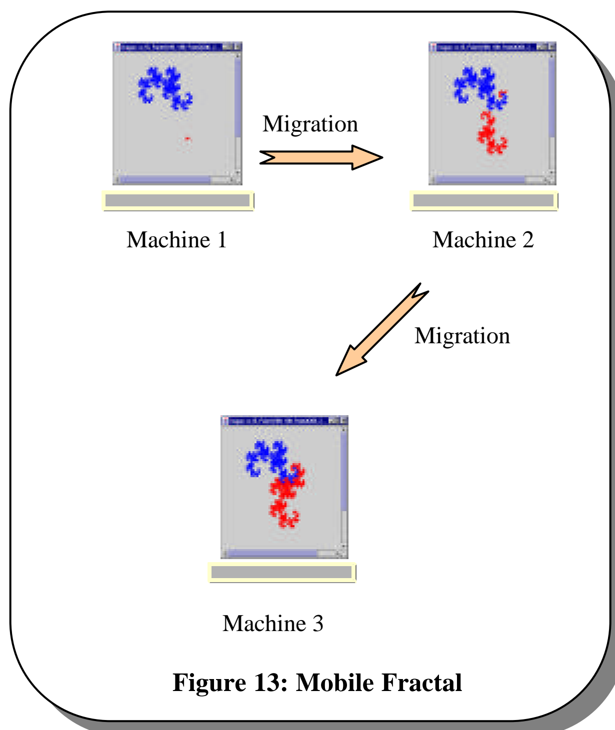


Figure 13: Mobile Fractal

We considered a recursive graphical Java application: the *Dragon* fractal curve where a small dragon appears at a certain depth of recursion [Mandelbrot75]. We implemented a Java *Dragon* application and used our thread mobility service in order to move the application, when it is running, between several machines. Figure 13 illustrates this experiment. The *Dragon* application is first started on a first machine, then moved to a second machine where it resumes its execution and finally moved to a third machine where it finishes its execution. The transfer of the thread calculating the fractal is performed by an external thread that calls the *go* method of our *MobileThreadManagement* class.

5.2. Dynamic reconfiguration: Mobile *Talk*

In this section, we describe how our mobility service can be combined with other Java services (object serialization, dynamic class loading) in order to build a dynamic reconfiguration tool.

We consider a *Talk* application where two remote users exchange messages. Initially, each user starts an instance of the *Talk* application on its personal computer with a graphical user interface. Each user has two communication channels: an input channel to receive messages from the remote user and an output channel to send messages to the remote user. During the talk, one of the users decides to transfer its application to a minimal host with limited physical characteristics (a mobile phone for example) and to resume its execution. This dynamic reconfiguration of the *Talk* application is illustrated by figure 14 and has the following requirements:

- Moving a running application from one host to another.
- Handling communication channels during transfer.
- Replacing the graphical user interface by a textual user interface when arriving on the destination host because of the limited physical characteristics.

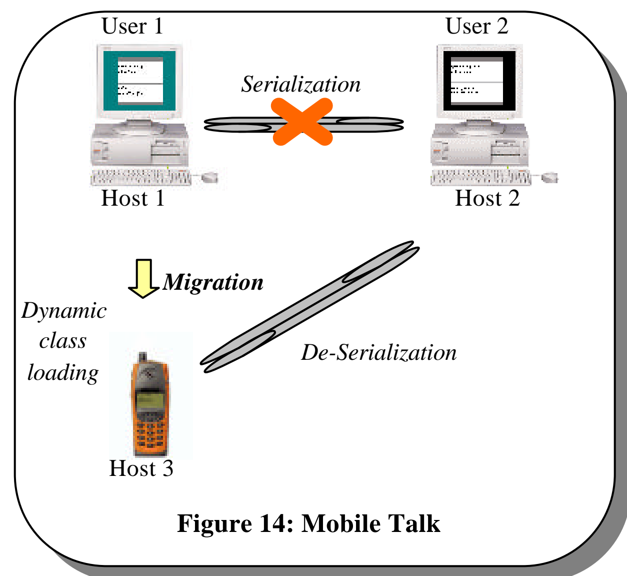


Figure 14: Mobile Talk

To transfer the running mobile *Talk* application to a new host, our mobility service can be used: it takes the current state of the application into account. To transfer the application to a mobile phone, the mobility service must use the Wireless Application Protocol (WAP) [WAPfactory 00].

To tackle the problems of communication channels and user interface, the Java serialization and dynamic class loading services are adapted. In fact, our mobility service relies on both serialization and

dynamic class loading to respectively transfer the objects and the classes used by the application at migration time. These two features can be specialized as follows:

- The serialization of the communication channels can be adapted in order to send a particular message to the remote user informing him about the next migration and then to close the connections. Symmetrically, the de-serialization of the communication channels can be adapted in order to recreate new channels and reestablish the connection with the remote user.
- The dynamic class loading can be adapted in order to use a textual user interface rather than the graphical one on the mobile phone.

Finally, the combination of our mobility service, the serialization and the dynamic class loading enables the building of a complete dynamic reconfiguration tool. This application has been experimented with a prototype implementation on our extended JDK 1.2.2. A port of our services to the K Virtual Machine, a lightweight JVM, is planned [Sun00b].

5.3. Discussion

In this section, we discuss some issues encountered when implementing thread mobility and thread persistence. Let's focus our attention on the mobility of a thread:

- What happens if a thread moves from a source host to a destination host while it is using objects shared with other threads on the source host?
- How are the communication channels connecting several threads handled when one of these threads moves to a new host?
- What happens if a thread that belongs to a multi-threaded application move to a new host?

We now tackle each of these issues and propose possible solutions.

What happens if a thread moves from a source host to a destination host while it is using objects shared with other threads on the source host? A first solution consists in replicating the shared object and transferring it with the mobile thread [Garcia-Molina86]. In this case, the consistency of the replicas must be managed. Another solution to the problem of shared objects is to use proxies on the destination host in order to allow remote access to shared objects. A problem of object availability occurs if the source host crashes [Chou83].

How are the communication channels connecting several threads handled when one of these threads moves to a new host? A first approach consists in using proxies on the destination host in order to access the communication channels remotely. A problem of channel availability occurs if the source host crashes. Another approach consists in closing the channels on the source host and recreating them on the destination host. In this case, messages in transit must be redirected to the new location and the naming of the new channels must be actualized on other hosts.

What happens if a thread that belongs to a multi-threaded application move to a new host? The thread can move alone to the destination host and communicate with the other threads remotely, or it can move with all the other threads or with a sub-set of them.

Finally, for each of the discussed issues, the solution strongly depends on application's needs. That is why we deliberately chose not to impose a particular solution at the level of our thread mobility and thread persistence services. The programmer of the application is thus free to choose the more appropriate approach.

6. Related work

Many systems have been developed providing process mobility and persistence, considering either homogeneous or heterogeneous processor architectures. There are a number of surveys discussing these features [Milojicic97] [Deconinck93]. Both mobility and persistence of control flows are based on a mechanism that enables the capture and the restoration of executions' state. Let's focus our attention on such mechanisms in the Java environment.

Three main approaches to address the problem of capturing/restoring the state of Java threads are distinguished: an *explicit* approach, an *implicit* approach based on a pre-processor of the application code and an *implicit* approach based on an extension of the JVM.

In the first approach, which we call *explicit management*, the programmer of an application has to entirely manage the capture and the restoration of the state of his application. For this purpose, the programmer has to explicitly add supplementary code in fixed points of his program and usually has to manage his own program counter. The added code manages a backup object in which information relative to the state of the application is stored. The backup object is then used in order to restore the application execution. When restoring the state of the application, the first statement of the program is a branch to the point where the program must continue. This approach is not flexible and implies a modification of the application itself if new backup points are added. This approach is used in most of applications based on mobile agent platforms [Chess95] that provide weak

mobility, such as Aglets [IBM96] and Mole [Baumann98].

The two other approaches, which we call *implicit*, provide a transparent service for capturing/restoring thread state. The service is independent from the application code and is provided as a function that may be called by the application itself or by an external application. These two approaches differ by their implementation:

- The first implicit approach consists in pre-processing the source (or byte) code of the application in order to insert statements. The inserted code attaches a backup object to the application. While the application is running, the backup object is re-actualized with the state of the application. When an application requires a snapshot of its state, it just has to use the associated backup object. In order to restore the execution state, data stored in the backup object are used to re-initialize the application 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) in order to rebuild the stack and re-initialize the local variables with the values stored in the backup object. The main motivation of this approach is that it does not modify the JVM. But its drawback is that it induces a significant overhead on application performance due to the inserted code, and on execution restoration which requires a partial re-execution of the application. The Wasp project provides a Java mobile agent platform based on a pre-processor which instruments the source code of Java applications [Fünfrocken98]. Several implementations of Java thread mobility based on a pre-processor of the bytecode are proposed [Truyen00] [Sakamoto00].
- 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. The extension must also provide a facility for building a new thread initialized with a previously captured state. These facilities can only be used on extended virtual machines. We followed this last approach for two reasons:
  - It reduces the overhead on application performance (no inserted code) and reduces also the cost of the capture/restoration service (its implementation is mainly native).
  - Since the thread state capture/restoration service has many applications, we believe that it is a basic functionality which must be integrated within the JVM.

  This solution has been used in the implementation of the Sumatra mobile agent platform [Ranganathan97]. Unlike Sumatra which supplies a mobility service, our implementation provides a generic service intended for other uses than mobility, like persistence [Bouchena99]. The recently proposed Merpati system also follows this approach [Suezawa00]. It makes the whole JVM mobile or persistent, with all its threads, while our services are fine-grained and can be applied to one thread.

To summarize, our services provide a transparent and fine-grained Java thread state capture/restoration facility. They can be used for several purposes among which thread mobility and thread persistence. They are integrated into the JVM and thus present competitive performance figures. A comparison between the performance of the first implicit approach (Wasp's mobility service) and the second implicit approach (our mobility service) can be found in [Bouchenak00].

7. Conclusion and future work

Since the Java virtual machine does not provide any access to the state of Java threads, we designed and implemented a new service for the capture and restoration of thread state. Our capture/restoration service is generic: we used it as a basis for the implementation of thread mobility and thread persistence services. With these services, a running Java thread can, at an arbitrary state of its execution, migrate to a remote machine or be checkpointed on disk and then recovered. In addition, the migration or the checkpointing of a thread can be initiated by the thread itself or by another thread.

Our services were integrated into the JVM, so they provide acceptable performance figures without inducing overhead on JVM performance. Finally, we experimented with a prototype implementation a dynamic reconfiguration tool based on our mobility service and applied to a running distributed application.

The lessons learned from this experiment are that:

- It is possible to extend the Java virtual machine with thread mobility and persistence services without re-designing the whole JVM.
- This implementation provides reasonable and competitive performance costs.

At the present time, we are considering the usage of our services in real world applications such as dynamic load balancing in distributed systems and the integration of our services into distributed Java virtual machines. We also plan to port our services to the K Virtual machine, the lightweight JVM, in order to make

them available on small devices such as phones and PDA [Sun00b].

Acknowledgments

I would like to thank Sacha Krakowiak and Jacques Mossière for providing many useful suggestions that significantly improved this paper.

References

[Ambler99] S. W. Ambler. The Design of a Robust Persistence Layer For Relational Databases. *AmbySoft Inc. White Paper*, October 1999. http://www.ambysoft.com/persistenceLayer.html

[Baumann98] J. Baumann, F. Hohl, M. Straber and K. Rothermel. Mole - Concepts of Mobile Agent System. *WWW Journal, Special issue on Applications and Techniques of Web Agents*, volume 1, no 3, 1998. http://mole.informatik.uni-stuttgart.de/

[Bouchenak00] S. Bouchenak and D. Hagimont. Pickling threads state in the Java system. *Proceedings of 33rd International Conference on Technology of Object-Oriented Languages (TOOLS Europe'2000)*, Mont-Saint-Michel, France, June 2000. http://sirac.inrialpes.fr/~bouchena

[Chess95] D. Chess, C. Harrison and A. Kershenbaum. Mobile Agents: Are They a Good Idea? *T.J. Watson Research Center White Paper*, IBM Research Division, March 1995. http://www.research.ibm.com/iagents/ publications.html

[Chou83] T. C. K. Chou and J. A. Abraham. Load Redistribution under Failure in Distributed Systems. *IEEE Transactions on Computers*, volume 32, no 9, September 1983.

[Deconinck93] Geert Deconinck, Johan Vounckx, Rudi Cuyvers and Rudy Lauwereins. Survey of Checkpointing and Rollback Techniques. *Technical Report, Katholieke Universiteit Leuven*, Belgium, June 1993.

[Douglis92] F. Douglis and B. Marsh. The Workstation as a Waystation: Integrating Mobility into Computing Environment. *The Third Workshop on Workstation Operating System (IEEE)*, Key Biscayne, Florida, USA, April 1992. http://www.douglis.org/fred

[Fuggetta98] A. Fuggetta, G. P. Picco and G. Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, volume 24, no 5, 1998. http://www.cs.ucsb.edu/~vigna/listpub.html

[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, Germany, September 1998. http://www.informatik.tu-darmstadt.de/~fuenf

[Garcia-Molina86] H. Garcia-Molina. The Future of Data Replication. *Symposium on Reliability in Distributed Software and Database Systems*, Los Angeles, California, USA, January 1986.

[Gosling96] J. Gosling and H. McGilton. The Java Language Environment. *Sun Microsystems White Paper*, May 1996. http://java.sun.com/docs/white

[Hofmeister93] C. Hofmeister and J. M. Purtilo. Dynamic Reconfiguration in Distributed Systems : Adapting Software Modules for replacement. *Proceedings of the 13th International Conference on Distributed Computing Systems*, Pittsburgh, USA, May 1993.

[IBM96] IBM Tokyo Research Labs. *Aglets Software Development Kit*, 1996. http://www.trl.ibm.co.jp/aglets

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

[Mandelbrot75] B. Mandelbrot . *Les Objets fractals : forme, hasard et dimension*. Flammarion, 1975.

[Milojicic97] D. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler and S. Zhou. Process Migration. *TOG Research Institute Technical Report*. March 1999. http://www.camb.opengroup.org/RI/java/moa

[Milojicic99] D. Milojicic, F. Douglis and R. Wheeler. *Mobility: Processes, Computers and Agents*. Addison Wesley, February 1999.

[Nichols87] D. A. Nichols. Using Idle Workstations in a Shared Computing Environment. *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, Austin, Texas, USA, November 1987.

[Oueichek96] I. Oueichek. *Conception et réalisation d'un noyau d'administration pour un système réparti à objets persistants*. Thèse de Doctorat, Institut National Polytechnique de Grenoble, France, October 1996.

[Pendragon99] Pendragon Software Corporation. CaffeineMark™ 3.0. *CaffeineMark 3.0 documentation*, 1999. http://www.pendragon-software.com/pendragon/cm3

[Pozo00] R. Pozo and B. Miller. SciMark 2.0. *SciMark 2.0 documentation*, 2000. http://math.nist.gov/scimark2

[Ranganathan97] M. Ranganathan, A. Acharya, S. D. Sharma and J. Saltz. Network-aware Mobile Programs. *Proceedings of the USENIX Annual Technical Conference*, Anaheim, California, USA, January 1997. http://searchpdf.adobe.com/proxies/0/38/54/9.html

[Sakamoto00] T. Sakamoto, T. Sekiguchi, A. Yonezawa. *Bytecode* Transformation for Portable Thread Migration in Java. *Proceedings of Second International Workshop Mobile Agents 2000 (MA'2000)*, Zurich, Switzerland, September 2000. http://web.yl.is.s.u-tokyo.ac.jp/~takas/

[Suezawa00] T. Suezawa. Persistent Execution State of a Java Virtual Machine. *Proceedings of the ACM 2000 Java Grande Conference*, San Francisco, California, USA, June 2000. http://www.ifi.unizh.ch/staff/suezawa/

[Sun00a] Sun Microsystems. Java 2 SDK, Standard Edition. *Sun Microsystems documentation*, 2000. http://java.sun.com/products/jdk/1.2

[Sun00b] Sun Microsystems. Java 2 Platform Micro Edition (J2ME) Technology for Creating Mobile Devices. *Sun Microsystems White Paper*, 2000. http://java.sun.com/products/cldc

[Sun00c] Sun Microsystems. Improving Serialization Performance with Externalizable. *Technical Tips*, Sun Microsystems, April 2000. http://developer.java.sun.com/developer/TechTips/2000/tt0425.html

[Truyen00] E. Truyen, B. Robben, B. Vanhaute, T. Coninx, W. Joosen and P. Verbaeten. Portable Support for Transparent Thread Migration in Java. *Proceedings of Second International Workshop Mobile Agents 2000 (MA'2000)*, Zurich, Switzerland, September 2000. http://www.cs.kuleuven.ac.be/~eddy/research.html

[WAPfactory00] WAPfactory. *Wap.com*. 2000. http://www.wap.com/

[Wojcik95] Z. M. Wojcik and B. E. Wojcik. Optimal Algorithm for Real-Time Fault Tolerant Distributed Processing Using Checkpoints. *Informatica*, volume 19, no 1, February 1995. http://ai.ijs.si/informatica/