On the use of Threads in Mobile Object Systems

Tim Coninx, Eddy Truyen, Bart Vanhaute, Yolande Berbers,
Wouter Joosen and Pierre Verbaeten

KULeuven Department of Computer Science
Celestijnenlaan 200A
B-3001 Leuven, Belgium
{tim,eddy,bartvh,yolande,wouter,pv}@cs.kuleuven.ac.be

Abstract

We have developed a portable mechanism for transparent thread migration in Java. This thread migration mechanism is implemented by instrumenting the original application code through a bytecode transformer without modifying the Java Virtual Machine.

In this paper we examine how this thread state capturing mechanism can be extended such that JVM thread semantics can be maintained in mobile object systems.

1 Introduction and overview

The Java language today is one of the most popular development languages, and also a topic of many research projects. Java is also often used as a language to develop mobile agent applications. There are various features of Java that triggered this evolution. First, in a large number of application domains, Javas machine-independent byte code has solved a long-lasting problem known to agent-based systems, namely the fact that agents must be able to run on *heterogeneous* platforms. A Java program is compiled into portable byte code that can execute on any system, as long as a Java Virtual Machine (JVM) is installed on that system. Nowadays, JVMs are running on systems with different hardware and system software characteristics (ranging from off-the-shelf PCs to Smart Cards). Second, byte code can be downloaded whenever necessary by means of the customizable Java class loading mechanism [LB98]. This flattens the way for supporting *code mobility*. Third, The Java security architecture allows construction of safe agent execution environments [SBH97]. Fourth, Sun's powerful serialization mechanism allows transparent migration of Java objects (i.e. the contents of instance variables), making *object state mobility* possible.

However, as a disadvantage, when an object is to be serialized, the execution state of threads that are active in this object is lost, leading to inconsistencies.

We implemented a mechanism that supports *transparent thread migration* [TRV$^+$00]. Using this mechanism, we allow the execution state of a thread to be captured before migration, and to be reestablished at the target location after migration. Furthermore, when the trace crosses host boundaries, the thread will be split up into different threads that are connected using an object request broker like RMI [Sun97]. These splitup threads form one *distributed tasks.*

First we will explain transparent thread migration, the technology that makes distributed tasks possible. Then we will show an example conversion of a normal thread to a distributed task. Finally, we point out two possible fields of direct use, namely *distributed locks* and *distributed application partitioning.*

2 Transparent Thread Migration

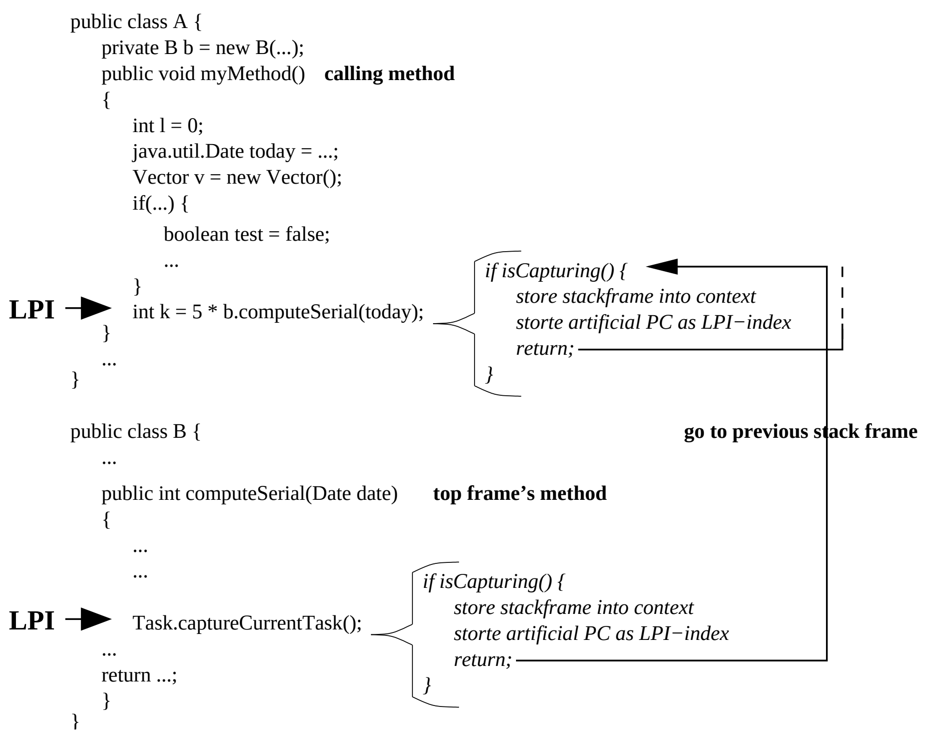
The main technique which is going to be used is transparent thread migration, as presented in [TRV$^+$00]. In order to migrate a thread, its execution must be suspended and its Java stack and program counter must be captured in a serializable format that is then sent to the target location. At the target location, the stack must be reestablished and the program counter must be set to the old code position. Finally the thread must be rescheduled for execution.

This is called transparent thread migration. We implemented this thread serialization mechanism by extracting the state of a running thread from the application code that is running in that thread. To achieve this, a *byte code transformer* instruments the application code by inserting code blocks that do the actual capturing en reestablishing of the current thread's state. This makes that our thread migration mechanism is portable across standard JVM platforms.

In order to deploy this thread serialization mechanism practically for migration, we offer *tasks*, a complement to JVM threads. A task encapsulates the JVM thread that is used for executing that task. As such, a task's execution state is the execution state of a JVM thread in which the task is running.

A task is serializable at any execution point, making transparent thread migration possible. Each task is associated with a separate *Context* object into which its execution state is captured, and from which its execution state is later reestablished. Finally, each task is associated with a number of boolean flags that represent a specific execution mode of the task. A task can be in three different modes of execution: running (normal execution), capturing (before serialization) and restoring (after deserialization).

Whenever a task enters capturing state (its capturing flag is set), the current method returns after saving its stack frame, like on figure 1. The inserted bytecode causes every method to check this capturing flag. When set, the method will first save its stack frame and its last performed invoke-instruction (LPI), and immediatly return. This will continue until the thread is back where it started. At that time, the task's context contains the whole trace the thread followed.



```
public class A {
    private B b = new B(...);
    public void myMethod()
    {
        int l = 0;
        java.util.Date today = ...;
        Vector v = new Vector();
        if(...) {
            boolean test = false;
            ...
        }
        int k = 5 * b.computeSerial(today);
    }
    ...
}
```

```
if isCapturing() {
    store stackframe into context
    storte artificial PC as LPI−index
    return;
}
```

```
public class B {
    ...
    public int computeSerial(Date date)
    {
        ...
        ...
        Task.captureCurrentTask();
    ...
    return ...;
    }
}
```

```
if isCapturing() {
    store stackframe into context
    storte artificial PC as LPI−index
    return;
}
```

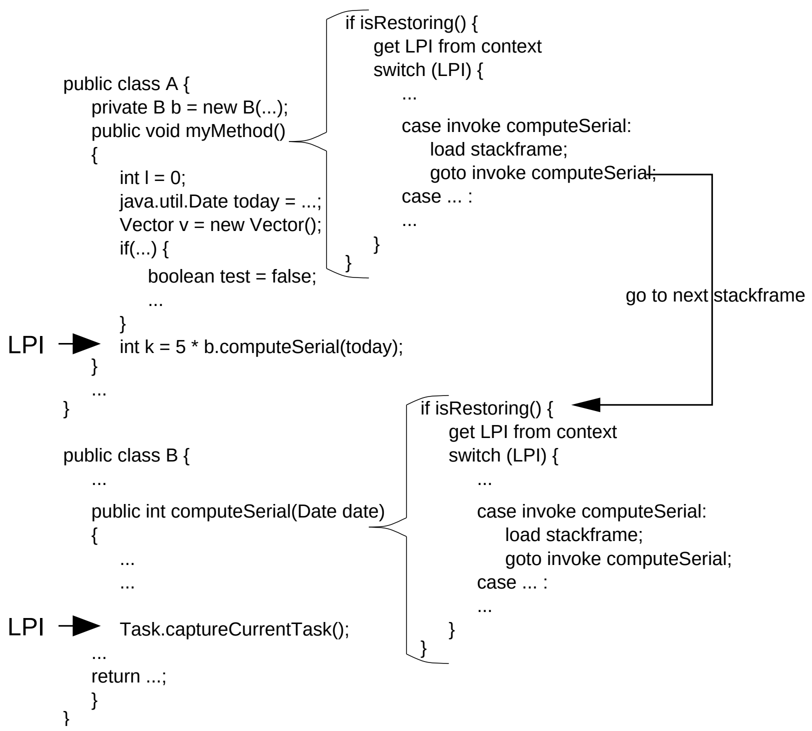
Figure 1: State Capturing

When a task has to be restored (figure 2), a new thread is started from the task (its restoring flag is set). The first stackframe is restored, and the LPI is checked to know which method was the next the task was in. Due to the inserted bytecode, each method restores the next stackframe, and checks the LPI for which method next to invoke. When the thread ultimatly reaches the method that started capturing, the restoring flag is unset, and the thread continues like nothing happened.

3 Distributed Tasks

3.1 The Problem

Figure 3 shows an example of a thread executing in four objects. Suppose now we would like to migrate object C. Normally, in order to have safe migration, we should wait until the object has become passive. However, using



```
public class A {
    private B b = new B(...);
    public void myMethod()
    {
        int I = 0;
        java.util.Date today = ...;
        Vector v = new Vector();
        if(...) {
            boolean test = false;
            ...
        }
        int k = 5 * b.computeSerial(today);
    }
    ...
}
```

```
if isRestoring() {
    get LPI from context
    switch (LPI) {
        ...
        case invoke computeSerial:
            load stackframe;
            goto invoke computeSerial;
        case ... :
        ...
    }
}
```

```
public class B {
    ...
    public int computeSerial(Date date)
    {
        ...
        ...
        Task.captureCurrentTask();
        ...
        return ...;
    }
}
```

```
if isRestoring() {
    get LPI from context
    switch (LPI) {
        ...
        case invoke computeSerial:
            load stackframe;
            goto invoke computeSerial;
        case ... :
        ...
    }
}
```

Figure 2: State Reestablishment

the techniques shown in section 2, we can serialize the executing thread, together with the objects, to begin migration much sooner.

When object C wants to migrate, the current task must be stopped, and its capturing flag is set. The running thread is being serialized before actual serialization of object C. In order to realize this, we need to keep record of the threads that are active in this object. Additional bytecode inserted at the start of each method invocation can ensure we have this information at hand.

During reestablishment, the task is split into three subtasks that are each assigned a separate JVM thread. B's call to C, and C's call to D are each converted into a remote method call. Although the three subtasks are executing on two different JVM's, like in figure 3, from a logical point of view they still belong to the same task. This introduces the notion of a *distributed task* that defines a global thread identifier, logically uniting the three subtasks. Looking at the call stack of the thread, we see it gets broken up into three callstacks, like in figure 4. These three callstacks form one logical whole.

When *during reestablishment of the thread*, an object reference is to be restored on the operand stack frame, there are two possibilities:

1. The object referenced to is present on the current host, so the reference

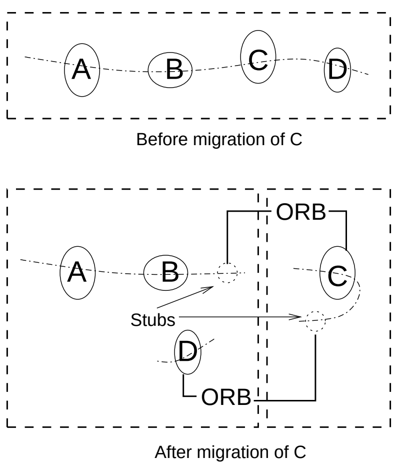


Figure 3: a thread in exectution, when C is migrated

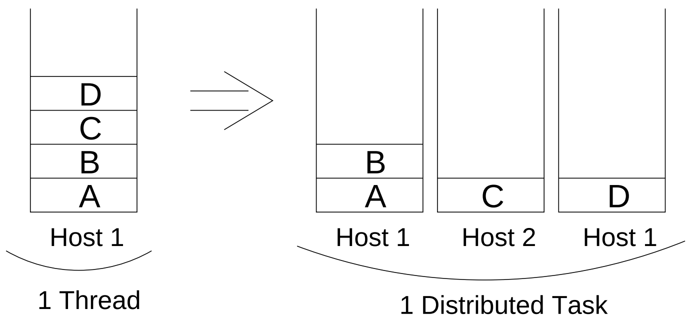


Figure 4: splitting up the callstack

is restored without further consideration.

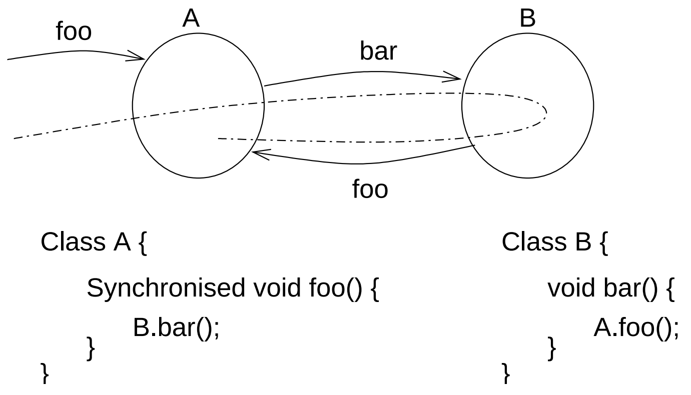
2. The object referenced to is not present on the current host. The object's stub is looked up, and the reference is replaced by a reference to the stub.

4 Discussion

4.1 Distributed locks

Suppose we have two classes, like shown in figure 5. Class A has its method synchronised, which means there can be at most one thread executing the method. However, the thread is granted access for a second time, because it is easily verified it is the same thread executing.

We have a problem when we would like to migrate object B. Because we have a distributed task, in reality there is a second thread trying to access the synchronised method of A. The object doesn't recognise the second thread



```
Class A {
    Synchronised void foo() {
        B.bar();
    }
}

Class B {
    void bar() {
        A.foo();
    }
}
```

Figure 5: locked method

as being of the same distributed task.

A solution for this problem consists of giving each distributed task a *global thread identifier*. Before access to the method is being denied, the thread's global thread identifier is checked, and possible deadlock is avoided.

4.2 Partitioning of distributed object applications

Proper placement of software components onto different hosts is critical to the performance of distributed applications. This is particularly true for object-oriented distributed applications, as the starting point is a large population of fine-grained objects. This pool of objects must first be partitioned, before assignment can take place. The primary consideration in object placement is that it must minimize the amount of communication over the network [KRR$^+$98].

Methods for automated partitioning exist, and employ a graph-based model of the application being paritioned. However, an open question is if these methods can be adapted to be applied at any point in an on-going distributed computation. Our mechanism of distributed tasks now provides a way to do this partitioning at run-time.

We illustrate this by an example system were two partitions are identified. At some point, the second partition is migrated while a thread is still active in both partitions. After migration (figure 6), the shared thread is converted to a distributed task, which still connects the two partitions over host boundaries.

5 Related Work

The Sumatra Project [ARS97] aims to support resource-aware mobile programs in Java. It does so by adding programming abstractions to Java.

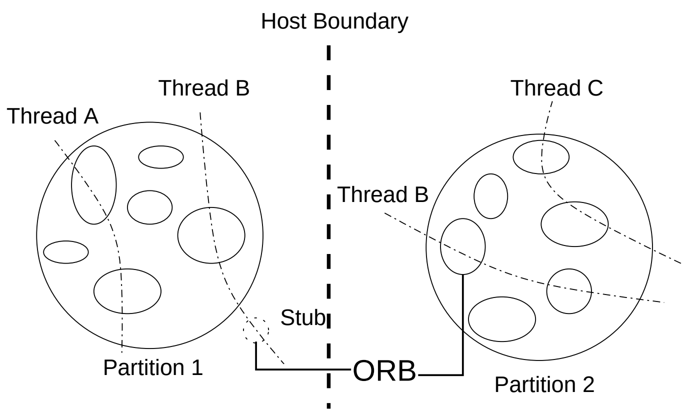


Figure 6: A partitioned application after migration

The application programmer can use these abstractions to write mobile programs. Sumatra does a very good job at managing resources that might be in use by objects that are about to move, something our proposal does not (yet) address. However, The programming abstractions are implemented as native methods on a unix system, as resource management is done by a system profiling service. Herefore, Sumatra is not as portable as one wants it to be.

Work from Jon Howell [How99] shows how to reach Java persistence through checkpointing. This is done by an extension of the JVM that first analyses the classfiles to set up checkpoints. At these checkpoints, the program state may be saved and later restored. This technique also deals very well with resources in use by the program, for example files and communication links. On the other hand, the checkpointing information is located in a flat, platform dependent file. So an object must always stay within the same platform.

The FarGo Project [HBSG99] introduces a model for programming the layout of distributed applications separately from their basic logic, by attaching relocation semantics to inter-component references, and by using built-in monitoring support for making relocation decisions. Like our proposal, FarGo offers complete location transparency and in addition gives portable monitoring support. The only disadvantage of FarGo is it only supports weak mobility, which means that the stack and program counter do not move, only the object states.

Besides these, many more projects relating to the use of threads in mobile object systems exist. Some older solutions are the Diamonds system [BCS+93] and the Emerald system [JLH+88]. Both systems are however not intended for Java.

6 Conclusions and Future Work

In this paper we presented a way how the technique of transparent thread migration can be used to introduce distributed tasks. Hereby, migrating objects is made possible to occur while threads are executing in those objects. Two possible fields of use, distributed locks and distributed object partitioning were also showed.

Looking at the related work, we have to admit our work is still very experimental, and hasn't yet dealt with issues like resource management. Also, while the technique of applying bytecode is a very useful one, we have to take care it doesn't go out of control. First by not adding too much, so it causes size blowups. Second by not applying it wrongly so normal program flow is corrupted.

Future work will consist mainly of creating a mobility framework that uses distributed tasks.

References

[ARS97] Anurag Acharya, M. Ranganathan, and Joel Saltz. Resource-aware meta-computing, March 1997.

[BCS+93] Umesh Bellur, Gary Craig, Kevin Shank, et al. Diamonds: Principles and philosophy. Technical report, Dept. of Electrical and Computer Engineering, Syracuse University, June 1993.

[HBSG99] Ophir Holder, Israel Ben-Shaul, and Hovav Gazit. System support for dynamic layout of distributed applications. Technical report, Dept. of Electrical Engineering, Technion – Israel Institute of Technology, 1999.

[How99] Jon Howell. Straightforward java persistence through checkpointing. In Ron Morrison, Mick Jordan, and Malcom Atkinson, editors, *Advances in Persistent Object Systems*, pages 322–334. 1999. URL: http://www.cs.dartmouth.edu/ jonh/research/.

[JLH+88] Eric Jul, Henry Levy, Norman Hutchinson, et al. Fine-grained mobility in the emerald system, 1988.

[KRR+98] Doug Kimelman, V.T. Rajan, Tova Roth, et al. Partitioning and assignment of distributed object applications incorporating object replication and caching. In *Proceedings of ECOOP '98*, June 1998.

[LB98] S. Liang and G. Bracha. Dynamic class loading in the java virtual machine. In *Proceedings of the Conference on Object-Oriented Technologies and Systems*, pages 36–44, April 1998.

[SBH97] M. Strasser, J. Baumann, and F. Hohl. Mole – a java based mobile agent system. In M. Mühlhäuser, editor, *Special Issues in Object Oriented Programming*, pages 301–308. 1997.

[Sun97] Sun Microsystems, Inc. *Java Remote Method Invokation Specification*, 1997. Java 1.2.

[TRV$^+$00] Eddy Truyen, Bert Robben, Bart Vanhaute, et al. Portable support for transparent thread migration in java, 2000. Submitted to AM2000.