Distributed Threads in Java

Danny Weyns, Eddy Truyen, Pierre Verbaeten

Computer Science Department, Katholieke Universiteit Leuven
Celestijnenlaan 200A, B-3001 Leuven Belgium

{danny,eddy,pv}@cs.kuleuven.ac.be

Abstract. In this paper, we study the problems of *thread identity* that arise with adapting a local Java program for execution in a distributed environment. When using a *distributed control flow* programming model like Java RMI or OMG CORBA, the programmer should take into account an inherent *shift of semantics*. We experienced a particular problem with shift of *thread* semantics when extending a serialization mechanism for JVM threads to a distributed setting. More specific, we encountered the problem of losing *logical thread identity* when the control crosses system boundaries. We solved this problem by introduced the generic notion of *distributed thread identity* in Java programming. Propagation of a globally unique, distributed thread identity provides a uniform mechanism by which all the program's constituent objects involved in a distributed control flow can uniquely refer to that distributed thread as one and the same computational entity. We have implemented distributed thread identity by means of byte code transformation of application programs. In the paper, we will use the serialization of a distributed execution state as a case to illustrate the value of our Java extension.

Keywords
Threads, Distributed systems, Java

1. INTRODUCTION

Different distributed programming models exist, such as asynchronous messaging, publish/subscribe systems, blackboard/tuple spaces (e.g. Linda, JavaSpaces). However the mainstream of intra-domain distributed systems are developed using an *object-based control flow* programming model such as Java RMI or OMG CORBA (method invocation between objects). This model is popular because it inherits some of the benefits of object-oriented programming languages such as Java and C++ and is similar to the 'good old' RPC inter-process communication style. Another advantage is that control flow programming models provide a good level of location transparency.
However, using an object-based control flow programming model has a number of consequences too. Remote method invocations involve blocking calls on remote objects and that is only feasible on a reliable, high-bandwidth and secure network.

Besides this dependability, writing distributed applications or adapting a local program[1] for execution in a distributed environment remains difficult because of the inherent shift of paradigms and semantics [4]. In Java RMI, well-known differences with 'normal' Java programming are the separation between class and interface of a remote object, the pass-by-copy semantics of non-remote arguments to a remote method invocation and the inherently more complicated failure modes of remote method invocation. The shift makes it necessary to re-engineer large parts of existing centralized programs. Shift of semantics potentially leads to unexpected execution results or run-time errors, if the programmer did not take these differences into account. Some of these problems are well studied and practical solutions have been worked out that make the implementation of distribution-related aspects more transparent to the programmer [4][1].

In this paper, we study a very particular shift of semantics, namely the *shift of thread semantics* that arises when adapting a local Java program for execution in a distributed environment. A thread is the unit of computation. It is a sequential flow of control within a single address space (i.e. JVM). More specifically we focus on distributed applications that are developed by means of an *object-based control flow programming model* like Java RMI. As we stated above, the nature of this programming model require a high-performance network. It is important to note that our work is situated in this kind of environments. Computational entities of this kind of applications execute as flows of control that may cross physical node boundaries, contrary to how conventional Java threads are confined to a single address space. In the remainder of this paper we refer to such a distributed computational entity as a *distributed thread of control*, in short distributed thread [2]. A distributed thread is a logical sequential flow of control that may span several address spaces (i.e. JVMs). As shown in Figure 1, a distributed thread τ is physically implemented as a concatenation of local (per JVM) threads [t1,...,t4] sequentially performing remote method invocations when they transit JVM boundaries.

The semantics of distributed threads differs from local JVM threads. The programmer has to take these differences into account.

In this paper we extend Java programs with the notion of distributed thread identity in order to reduce the shift of threading semantics between local and distributed programming. Propagation of a globally unique distributed thread identity provides a uniform mechanism by which all the program's constituent objects involved in a distributed thread can *uniquely refer* to that distributed thread as one and the same computational entity.

We have implemented this Java extension by means of byte code transformation of application programs. As such the mechanism of propagation can be added to existing applications and is reusable on top of different middleware platforms.

This paper is structured as follows: first, in section 2 we introduce a problem of shift of thread semantics we encountered in a project we are working on. In section 3 we give a definition of distributed thread and distributed thread identity. In section 4 we describe the implementation of distributed thread identity in Java. Section 5 illustrates the benefits of distributed task identity and explains how we used it as the key

[1] A local program executes completely within the boundaries of one logical address space, e.g. Java Virtual Machine (JVM)

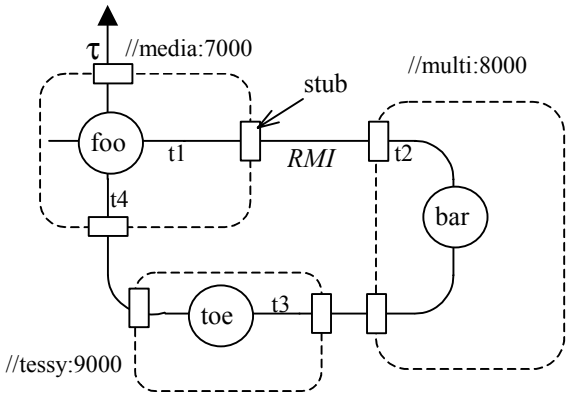

Figure 1. A Distributed Thread.

mechanism to solve the problems mentioned in section 2. Finally we conclude in section 6.

2. THE PROBLEM OF A "LOGICAL" THREAD IDENTITY IN A DISTRIBUTED CONTEXT

In this section we briefly describe a new project we are working on and situate the problem of shift of thread semantic we encountered in it. We will use this problem as a case to illustrate the value of distributed thread identity.

2.1. Introduction to the project

The goal of the project is the development a mechanism for serializing the execution state of a distributed Java application that is programmed by means of an Object Request Broker like Java RMI. Such a mechanism can serve many purposes like for example migrating execution state over the network or storing it on disk. To validate our research we build a prototype for repartitioning distributed Java applications at runtime. This mechanism enables the applying of existing partitioning methods at any point in an on-going distributed computation. Runtime repartitioning aims to improve the global load balance or network communication overhead of a running application by repartitioning the object configuration of the application dynamically over the available physical nodes at run-time. Existing work offers this support in the form of new middleware platforms with a dedicated execution model and object migration support that aligns well with run-time repartitioning [8][11]. A disadvantage of this approach is that existing ordinary RMI-based applications, which have obviously not been developed with support for repartitioning in mind, must partially be rewritten such that they become compatible with the programming model of the new middleware platform.

The approach of our project is to develop a byte code transformer that transparently adds new functionality to an existing Java application such that this application becomes automatically repartition-able by the external monitoring and management architecture.
The run-time repartitioning process consists of 4 successive phases. In the first phase, the management architecture allows an administrator to monitor the application's execution and specify a specific request for repartitioning the application over the available physical nodes. In the second phase, the application takes a snapshot of its own *global execution state*, *capturing* the state of all threads that are executing in the application. After this, the execution of all application objects is temporarily suspended and the corresponding thread states are stored as s*erialized data* in a global thread context repository. In the third phase, the management architecture carries out the initial request for repartitioning by migrating the necessary objects over the network. In the final and fourth phase, the execution of the application is resumed. Before the execution can be resumed, the execution state of all threads is first *reestablished* from the stored data in the global thread repository.
The advantage of having the phases two and four is that execution and object migration are kept completely orthogonal to each other, avoiding race conditions and tricky problems with migration. For example a typical problem, solved in this way, is that an object waiting on a pending reply of a (remote) method invocation can anyway be migrated.
In this paper we will focus only on phases two and four, i.e. on the serialization of the execution state. For more information about runtime repartitioning we refer to [12].
Current Java technology completely don't support the serialization of execution state. However, we already had implemented a portable mechanism for serialization of local JVM threads, called Brakes [10]. This thread serialization mechanism is implemented by instrumenting the original application code at the byte code level, without modifying the Java Virtual Machine.

2.2. Brakes for capturing JVM threads

In Brakes the execution state of a thread is extracted from the application code that is executed in that thread. For this, a byte code transformer (i.e. the Brakes transformer) inserts capture and reestablishing code blocks at specific code positions in the application code.
With each thread three *flags* (called isSwitching, isRestoring, isRunning) are associated that represent the execution mode of that specific thread. When the isSwitching flag is on, the thread is in the process of capturing its state. Likewise, a thread is in the process of reestablishing its state when its isRestoring flag is on. When the isRunning flag is on, the thread is in normal execution.
Each thread is associated with a separate Context object into which its state is switched during capturing, and from which its execution state is restored during reestablishing.
The process of capturing a thread's state (indicated by the empty-headed arrows in Figure 2) is then implemented by tracking back the *control flow*, i.e. the sequence of nested method invocations that are on the stack of that thread. For this the byte code

transformer inserts after every method invocation instruction a code block that switches the stack frame of the current method into the context and returns control to the previous method on the stack, etc. This code block is only executed when the isSwitching flag is set.
The process of reestablishing a thread's state (indicated by the full-headed arrows in Figure 2) is similar but restores the stack frames in reverse order on the stack. For this, the byte code transformer inserts in the beginning of each method definition a code block that restores stack frame data of the current method and subsequently creates a new stack frame for the next method that was on the stack, etc. This code block is only executed when the isRestoring flag is set.

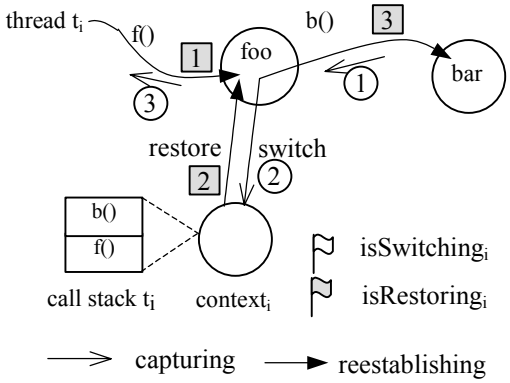

Figure 2. Thread Capturing/Reestablishing in Brakes.

In Brakes context objects are managed on *a per thread basis* by the context manager. So every thread has its own Context object, exclusively used for switching the state of that thread. The right context object is looked up by the context manager using the *thread identity* as hashing key.
Capturing (and reestablishing) the execution state of a distributed application, developed with a distributed control flow programming model such as Java RMI, requires however a mechanism for serializing (and de-serializing) *a distributed execution state.* Reusing Brakes for this required a complete redesign of the Brakes byte code transformer. In section 5 we will show how we integrated distributed thread identity into the byte code transformation scheme and solved this problem in an elegant way.

3. DISTRIBUTED THREADS AND DISTRIBUTED THREAD IDENTITY

The notion of distributed thread has already been introduced in the Alpha distributed real-time OS kernel at CMU [2]. D. Jensen has identified the notion of distributed thread as a powerful basis for solving distributed resource management problems. We borrow the definition of distributed threads from this work:

"A distributed thread is the locus of control point movement among objects via operation invocation. It is a distributed computation, which transparently and reliably spans physical nodes, contrary to how conventional threads are confined to a single address space."

As such a distributed thread is a sequential flow of control that can cross physical node boundaries. Opposite to this, a conventional thread is a sequential flow of control that remains within the boundaries of its local address space.

With respect to the subject of this paper, a distributed thread can be implemented by the concatenation of local (per node) threads sequentially performing remote method invocations when they transit nodes. Distributed control flow programming models, such as Java RMI or OMG CORBA, naturally support this. By remote method invocations and returns, a distributed thread can extend and retract its locus of control point movement among a program's constituent methods in object instances that may be dispersed across a multiplicity of physical computing nodes.

"A distributed thread carries attributes related to the nature, state and service requirements of the computation it represents. These attributes may be inspected by the kernel and its clients."

Distributed thread attributes are thus *propagated* with the locus of execution point movement and can be *inspected* by a kernel object and the application objects executing in the environment provided by that kernel. A kernel object is here referred to as the whole of protocols and run-time data structures that support the execution of applications. Examples of kernel objects in the context of this paper are the JVM and middleware platforms with a distributed control flow programming model (e.g. Java RMI). Kernel objects can also be external control instances that provide non-functional services to applications such as load balancing or fault tolerance.

With respect to the subject of this paper, we claim that *distributed thread identity*, implemented as an attribute, provides the basis for "logical" thread identity in a distributed context. A distributed thread identity provides a uniform mechanism by which all the program's constituent objects involved in a distributed thread can *uniquely refer* to that distributed thread as one and the same computational entity. Propagation and inspection of distributed thread identity enables kernel object as well as application objects to access the currently executing distributed thread.

"A distributed thread's attributes may be modified and accumulated (created) in a nested fashion as it executes operations within objects."

A programming model for distributed threads sets out the rules for a valid creation and modification of attributes during the distributed thread's life.

With respect to the subject of this paper, two simple rules should be enforced for distributed thread identities:

1. A distributed thread identity must be created once, more specifically when the distributed thread is scheduled for the first time.
2. A distributed thread identity must not be modified after its creation. This is because it must provide physically dispersed objects with a unique and *immutable* reference to a distributed thread.

With support for distributed thread identification, we will solve the problem described in section 2 in an elegant way. The key is to save executed state on the base of *distributed thread identities*.

4. DISTRIBUTED THREAD IDENTITY FOR JAVA

We implemented a generic Java extension that implements a simple programming model for distributed threads, in particular distributed thread identity. However before extending distributed Java programming with this notion, the question arises how to do this best. We propose three requirements that must be fulfilled by the chosen implementation strategy.

- It must be possible to integrate the functionality transparently into existing Java applications.
- The implementation of distributed threads must be portable across different heterogeneous platforms.
- The mechanism of distributed threads should be dynamically integrate-able with the existing middleware platforms such as Java RMI, OMG CORBA, and Microsoft DCOM.

4.1. Byte code transformation

Byte code transformation [3] is an excellent candidate that satisfies all three requirements. First, it extends Java language features in a user transparent way. For example, a custom class loader can automatically perform the byte code transformation at load-time. Second, the functionality of distributed threads can be added without modifying the JVM, as such it is portable on any system, as long as a standard JVM is installed on that system. Dynamic integration with a distributed control flow programming mechanism is possible if done carefully. In the remainder of this section, we will occasionally point out in more detail how dynamic integration can be achieved.

We used M. Dahm's tool for byte code transformation, BCEL [3] to develop a byte code transformer that extends Java programs with the notion of distributed threads, more specifically distributed thread identity. In the rest of this paper we refer to this transformer as the *DTI transformer*.

4.1.1. Propagation of distributed thread identity

To achieve propagation of distributed thread identities, the DTI transformer extends the signature of each method with an additional argument of class D_Thread_ID. D_Thread_ID is a serializable class that implements an immutable, globally unique identifier. The signature of every method invoked in the body of the methods must be extended with the same D_Thread_ID argument type too. For example, a method f() of a class C is rewritten as:

```
//original method code          //transformed method code
f(int i, Bar bar) {             f(int i, Bar bar, D_Thread_ID id) {
   …                              …
   bar.b(i);                      bar.b(i, id);
   …                              …
}                               }
```

When f() is called the D_Thread_ID is passed as an actual parameter to f(). Inside the body of f(), b() is invoked on bar. The body of f() passes on its turn the D_Thread_ID it received as an extra argument to b(). This way the distributed thread identity is automatically propagated with the control flow from method to method. Dynamic integration with a distributed object-based middleware such as Java RMI is simply achieved if the stub classes for the different remote interfaces are generated after the DTI transformation has been performed.

4.1.2. Creation and modification of a distributed thread

The Java thread programming model offers the application programmer the possibility to start up a new thread from within the run() method of a class that implements the java.lang.Runnable interface.
The DTI transformer will instrument this kind of classes such that they will implement the D_Runnable interface instead. D_Runnable is defined as follows:

```
interface D_Runnable {
    void run(D_Thread_ID id);
}
```

The example class Bar that originally implements the Runnable interface illustrates the transformation:

```
//original class definition
class Bar implements
    java.lang.Runnable
{   …
    void run() {…}
}
```

```
//transformed class definition
class Bar implements
   D_Runnable
{  …
   void run(D_Thread_ID id) {…}
}
```

As stated in section 3, the identity of a distributed thread must be *created* at the moment the distributed thread is created. This behavior is encapsulated in the D_Thread class, which serves as an abstraction for creating a new distributed thread. A new distributed thread can simply be started with:

```
Bar b = new Bar();
D_Thread dt = new D_Thread(b);
dt.start();
```

The D_Thread class itself is defined as:

```
class D_Thread implements java.lang.Runnable {
   public static D_Thread_ID getCurrentTheadID() {
   }
   public D_Thead(D_Runnable o) {
     object = o;
     id = new D_Thread_ID();
   }
   public void run() {
     object.run(id);
```

```
    }
    public void start() {
      new Thread(this).start();
    }
    private D_Runnable object;
    private D_Thread_ID id;
}
```

As stated in section 3, distributed thread identities may never be *modified*. As such, D_Thread does not provide any method operations for this. Furthermore, D_Thread_ID objects are stored either as a private instance member of class D_Thread or as a local variable on a JVM stack. Nonetheless it remains possible for a malicious person to modify distributed thread identities. For example it is possible to corrupt D_Thread_ID's when they are sent over the physical network. Or a byte code transformer could be written that inserts malicious code for modifying the value of the local variable pointing to a D_Thread_ID object. It's clear that additional security measures are necessary. This is subject to further research.

4.1.3. Inspection of distributed thread identity

Since distributed thread identities are propagated with method invocation as an additional, last argument, it is possible to compute for every method definition which local variable on the JVM stack points to the corresponding D_Thread_ID object. Based on this information specific byte codes can inspect the value of the local D_Thread_ID variable at any point in the method code.

As stated in section 3, as well application objects as kernel object must be able to inspect the distributed thread identity. Different approaches for implementing inspection must be followed for the two kinds of objects.

D_Thread_ID inspection by application object. The application's constituent methods are the locus of control point movement of a distributed thread. Therefore, every application object is able to inspect the identity of the distributed thread that is currently executing one of its methods. To support this in the distributed thread programming model, the D_Thread class defines a static operation getCurrentThreadID(). The implementation of this method is encapsulated in the DTI transformer, which transparently replaces all invocations of this operation with a code block that returns the value of the local D_Thread_ID variable.

D_Thread_ID inspection by a kernel object can be supported by a kernel-defined (static method, transformer) pair. First, a kernel object's class implements a kernel-defined interface for accepting notification of the event when a distributed thread passes by a specific marked point in the program. This event listener-style interface defines in particular one or more static 'bottleneck' methods [9] for passing the identity of that distributed thread to the kernel object. Second, a kernel-defined byte code transformer inserts on the marked points in the program, a code block for invoking this static method with the current D_Thread_ID. As such the kernel is notified when a distributed thread's locus of movement passes by that specific point. The strong combination of a kernel-defined bottleneck interface plus a conforming transformer makes it easy to *replace* JVM mechanisms (e.g. monitors) or *augment* existing middleware implementations (e.g. mobile object platforms) with new kernel-

defined functionality acting upon whole distributed threads as individual computational entities.

4.2. Implementation of the DTI transformer

At byte code level each method has an indexed list of local variables. For each non-static method the local variable with index 0 is the this-reference. Thereafter follows the arguments as defined in the signature. For static methods the first argument of the argument list gets the index 0. Furthermore the list of non-abstract methods is extended with the 'classic' local variables as defined in the body of the method. Thus extending the signature of a method with the D_Thread_ID argument constructs an additional entry in the list of the local variables of that method. This must be done in a fully consistent way, as well for all references to local variables in the code of the method, as for the Constant Pool references of the class file. The signature of all invoked methods in the body of each method is extended with the D_Thread_ID variable, so we have to put this local variable as an extra argument on the JVM stack before the method is invoked.
Replacing the Runnable interface with the D_Runnable interface is quite easy. We only have to screen the implemented interfaces of each application class and change the corresponding entry in the Constant Pool.
Note that the byte code transformer rewrites the byte code in a selective way. The current implementation does not rewrite calls on methods of the JDK classes. This choice is basically made for practical reasons. This is however subject to further research, because for certain applications or requirements of kernel objects it may be necessary to rewrite the JDK libraries.

5. DISTRIBUTED THREAD IDENTITY AT WORK

In this section we illustrate the benefits of distributed threads and distributed thread identity. We illustrate how we extend Brakes, a mechanism for the serialization of a local execution state, to a tool for the serialization of a distributed execution state by means of distributed thread identity.

5.1. The problem for Brakes with capturing Distributed Threads

This section describes the problem that we encountered when trying to reuse Brakes for capturing distributed threads without underlying support for distributed thread identity. In order to implement phases two and four of the repartitioning process (see section 2.1) robustly, such distributed threads should be captured/reestablished as whole entities in one atomic step.
However, we developed Brakes without having anticipated the need for capturing distributed threads. In Brakes execution state is after all saved on per local JVM thread basis. This works well for capturing local control flow but not for capturing a distributed thread of control. Figure 3 illustrates the problem for an example

distributed thread τ, implemented by a concatenation of multiple local (per JVM) threads, sequentially performing remote method invocations when they transit JVMs. For example, once thread t_i reaches method b() on object bar, the call p() on object poe is performed as a remote method invocation. This remote call implicitly starts a new thread t_j at host multi.

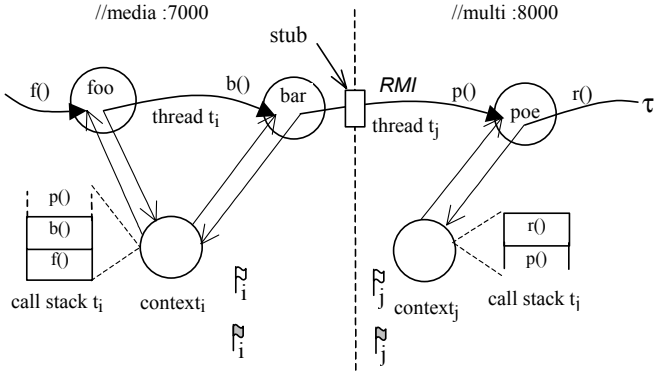

Figure 3. Context per JVM.

Physically, the threads t_i and t_j hold their own local subset of stack frames, but logically the total set of frames belongs to the same distributed thread. The context manager is however not aware of this logical connection between threads t_i and t_j. As a consequence contexts and flags of these JVM threads will be managed as separate computational entities, although they should be logically connected. Without this logical connection, it becomes difficult to robustly capture and reestablish a distributed thread as a whole entity in one atomic step. For example, it becomes quasi impossible for the context manager to determine the correct sequence of contexts that must be restored for reestablishment of a specific distributed thread. Of course we could modify the Brakes transformer with support for capturing distributed thread of controls, but this involves huge changes in the design of the Brakes transformer.

5.2. Distributed thread identity to the rescue

Distributed thread identity provides an elegant solution to this problem. More specifically if we augment the Brakes transformer with functionality for D_Thread_ID inspection by a kernel object (i.e. the context manager) via a bottleneck interface, it is possible to implement a context manager that manages contexts on a per distributed thread basis (see Figure 4). This update to the Brakes transformer entails only slight modification.

The local implementation of the context manager must be adapted in two ways:

New static bottleneck interface for inspecting distributed thread identity. As stated in the description of Brakes, JVM thread stack frames are switched into the associated Context object via a static bottleneck interface defined by the context manager. For example, switching an integer from the stack into the context is done

by inserting in the method's byte code at the appropriate code position an invocation of the following static method:

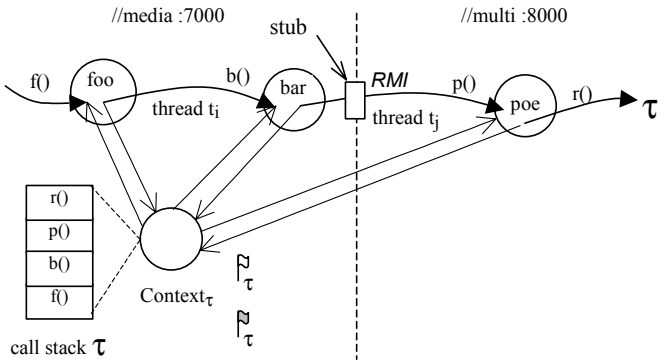

Figure 4. Context per Distributed Thread.

```
public static curPushInt(int i) {
   Context c = getContext(Thread.currentThread());
   c.pushInt(i);
}
```

The appropriate Context object into which to switch is looked up with the current thread identity as hashing key. Such push-methods are defined for all basic types as well as for object references. Complementary, there are pop-methods for restoring execution state from the context. Crucial in the push and pop methods is the identification of the Context object for the actual thread.
However, in order to allow the context manager to manage Context objects on a per distributed thread basis, this static bottleneck interface must be changed such that the context manager can inspect the identity of the current distributed thread.

```
public static pushInt(int i, D_Thread_ID id) {
   Context c = getContext(id);
   c.pushInt(i);
}
```

As such the Brakes byte code transformer must only be modified to support "D_Thread_ID inspection by a kernel object" (see section 4.1.3). Propagation of D_Thread_ID along the call graph is guaranteed by DTI transformation of the application classes (see section 4.1.1).
Distributed Architecture. Figure 4 motivates clearly that the implementation of the context manager must become distributed now. Figure 5 sketches the architecture of such a distributed implementation. Capturing and restoring code blocks still communicate with the bottleneck interface of the local context manager, but the captured execution state is now managed per distributed thread by a central *distributed thread manager*.

The distributed thread manager manages global flags that represent the execution state of the distributed application as a whole. These global flags are kept synchronized with the flags of the local context managers.

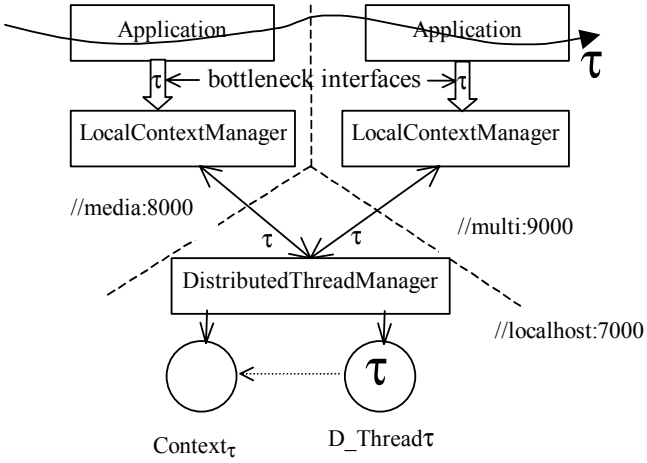

Figure 5. Distributed Architecture of the Context Manager.

DistributedThreadManager offers a public interface to initiate the capturing and reestablishing of the distributed execution state. The capturing of the execution state of a distributed application is started by calling the operation `captureState()` on the distributed thread manager. This method sets the isSwitching flag on all local context managers by broadcast. As soon as a distributed thread detects that the isSwitching flag is set, the distributed thread will start switching itself into the context.

Reestablishment of the distributed application is initiated by calling the operation `resumeApplication()` on the distributed thread manager. This method sets the isRestoring flag on each local context manager and restarts the execution of all distributed threads by calling the start() method defined on D_Thread. Each distributed thread detects immediately that the isRestoring flag is set, and thus restores itself from the context.

5.3. Evaluation of the serialization mechanism

In this section we evaluate the serialization mechanism for a distributed execution state. Since inserting byte code introduces time and space overhead we look to the blowup of the class files and give results of performance measurements. In our overviews we distinguished between the influence from byte code transformation to add distributed thread identity and the influence to add full support for serialization of distributed execution state.

The blowup of the byte code for a particular class highly depends on the number and kind of defined methods. Since the Brakes transformer inserts code for each invoke-

instruction that occurs in the program, the space overhead is directly proportional to the total number of invoke-instructions that occur in the agent's application code. Per invoke-instruction, the number of additional byte code instructions is a function of the number of local variables in the scope of that instruction, the number of values that are on the operand stack before executing the instruction and the number of arguments expected by the method to be invoked. The DTI transformer rewrites method and class signatures. This adds a space overhead proportional to the number of signature transformations.

To get a representative picture, we measured the blowup for three kinds of applications:

- Low degree of method invocation (i.e. the program has a structure *main{m1;}* thus the code is compacted in one method body)
- Nested method invocations (i.e. the program has a structure *main{m1;}; m1{m2; m3;}; m3{m4;}* thus the code is scattered over a number of nested methods)
- Sequential method invocations (i.e. the program has a structure *main{m1; m2; m3; m4;}* thus the code is scattered over a number of sequential non-nested methods)

Tabel 1 shows the results of our measurements. Functionality for distributed thread identity produces an average blowup of 27 % while the average blowup for full serialization functionality is 83 %. The expansion for Sequential is rather high, but its code is a severe test for blowup.

Tabel 1. Byte code Blowup for Three kind of Applications.

Code size (Bytes)	Low degree	Nested	Sequential
Original	377	431	431
DTI	399	616	524
DTI + Brakes	573	718	991

5.4. Performance measurements

For performance measurements, we used a 500 MHz Pentium III machine with 128 MB RAM with Linux 2.2 and the SUN 2SDK. We limited our tests to the overhead during normal execution (i.e. overhead as a consequence of the execution of inserted byte code). Tabel 2 shows the results of our tests. For distributed thread identity we measured an average overhead of only 3 %. For full serialization functionality we get an average overhead of 17 %.

Tabel 2. Overhead for Three kind of Applications.

Overhead (ms)	Low degree	Nested	Sequential
Original	190	811	1011
DTI	192	852	1054
DTI + Brakes	199	949	1314

From this results we may conclude that blowup and performance overhead for integrating distributed thread identity as well as for adding full serialization functionality are quit acceptable.

6. CONCLUSION

In this paper we integrate the notion of distributed threads into Java programs as a mechanism to reduce the shift of threading semantics between local and distributed programming. Propagation of a globally unique distributed thread identity provides a uniform mechanism by which all the program's constituent objects involved in a distributed thread can uniquely refer to that distributed thread as one and the same computational entity. We have employed byte code transformation to implement this mechanism at the application-level. The strong combination of a middleware-defined bottleneck interface and a conforming byte code transformer enables also middleware platforms to act upon a distributed thread as one and these same computational entity. We verified our approach by extending Brakes, an existing tool for the serialization of JVM threads, to serialize a distributed executing state.
Besides the points of attention we already mentioned in the paper, in the future we intend to investigate how other problems with thread identity in a distributed Java setting can be solved with the notion of distributed threads. Distributed threads may for example flattens the way for implementing a kernel object that implements a distributed monitor conforming the monitor semantics for local Java programs.

7. ACKNOWLEDGEMENTS

This research was supported by a grant from the Flemish Institute for the advancement of scientific-technological research in the industry (IWT). We would like to thank Tim Coninx and Bart Vanhaute for their valuable contribution to this work and the many fruitful discussions with them. Finally, a word of appreciation goes to Tom Holvoet and Frank Piessens for their useful comments to improve this paper.

8. REFERENCES

[1] G. Agha. Actors: "A Model of Concurrent Computation in Distributed Systems", MIT Press, 1986.

[2] R Clark, D.E. Jensen, and F.D Reynolds, "An Architectural Overview of the Alpha Real-time Distributed Kernel", in Proceedings of the USENIX Workshop on Microkernel and Other Kernel Architectures, April 1992.

[3] Markus Dahm, Freie Universität Berlin. Byte Code Engineering. In Clemens Cap, editor, Proceedings JIT'99, 1999.

[4] Markus Dahm, Freie Universität Berlin. The Doorastha system, Technical Report B-I-2000.

[5] D. E. Jensen, Distributed Real-Time Specification Java Specification Request 000050, http://java.sun.com/aboutJava/communityprocess/jsr.html

[6] ObjectSpace Inc., Object space VOYAGER, Core Technology 2.0, 1998.

[7] OMG, Joint Initial Proposal for Dynamic Real-Time CORBA, October 1999, Object Management Group

[8] B. Robben. Language Technology and Metalevel Architectures for Distributed Objects, Phd KULeuven, Belgium, 1999, ISBN 90-5682-194-6.

[9] C. Szyperski, Component Software: Beyond Object-Oriented Programming, Addison-Wesley, 1996.

[10] Eddy Truyen, Bert Robben, Bart Vanhaute, Tim Coninx, Wouter Joosen and Pierre Verbaeten, KULeuven, Department of Computer Science, B-3000 Leuven, Belgium. Portable Support for Transparent Thread Migration in Java, 2000.

[11] Eddy Truyen, Bart VanHaute, Bert Robben, Frank Matthijs, Erik Van Hoeymissen, Wouter Joosen, Pierre Verbaeten, "Supporting Object Mobility - from thread migration to dynamic load balancing", OOPSLA'99 Demonstration, November '99, Denver, USA.

[12] Runtime repartitioning with DistributedBrakes, background information and prototype available for download at: http://www.cs.kuleuven.ac.be/~danny/DistributedBrakes.html