Serialization of Distributed Execution-state in Java

Danny Weyns Eddy Truyen Pierre Verbaeten

Department of Computer Science, DistriNet, K.U.Leuven,
Celestijnenlaan 200A, Leuven 3001, Belgium
++32 (0) 16 32 76 02
http://www.cs.kuleuven.ac.be/~danny/DistributedBRAKES.html
{danny,eddy,pv}@cs.kuleuven.ac.be

Abstract. In this paper we present a mechanism for serializing the execution-state of a distributed Java application that is implemented on a conventional Object Request Broker (ORB) architecture such as Java Remote Method Invocation (RMI). To support capturing and reestablishment of distributed execution-state, we developed a byte code transformer that adds this functionality to a Java application by extracting execution-state from the application code. An important benefit of the serialization mechanism is its portability. It can transparently be integrated into any legacy Java application. Furthermore, it does require no modifications to the Java Virtual Machine (JVM) nor to the underlying ORB. The serialization mechanism can serve many purposes such as migrating execution-state over the network or storing it on disk. In particular, we describe the implementation of a prototype for repartitioning distributed Java applications at runtime. Proper partitioning of distributed objects over the different machines is critical to the global performance of the distributed application. Methods for partitioning exist, and employ a graph-based model of the application being partitioned. Our serialization mechanism enables then applying these methods at any point in an ongoing distributed computation.

1 Introduction

In this paper we present a mechanism for serializing the execution-state of a distributed Java application. We describe this mechanism in the context of a system for run-time repartitioning of distributed Java applications. For distributed object-oriented applications, an important management aspect is the partitioning of objects such that workload is equally spread over the available machines and network communication is minimized. Traditional techniques for automatic partitioning of distributed object applications use graph-based algorithms, e.g. [7].
In a static approach an external monitor automatically determines the best possible partitioning of the application, based on observation of behavior of the application (i.e., the dispersal of costs) during a number of representative runs. This partitioning is fixed for the entire execution of the application. However in a dynamic environment the optimal object distribution may change during execution of the

application. To cope with this, the external monitor may periodically check the workload at run-time on each separate machine. Whenever the workload on one or more machines crosses a certain threshold, e.g., following the low-water high-water workload model as described in [10], the monitor immediately triggers the repartitioning algorithm and relocates one or more objects to another machine.
The relocation of a running object involves the migration of its object code, data state and execution-state. Conventional Java-based Object Request Brokers (ORB), such as the Voyager ORB [8], support passive object migration, i.e., migration of object code and data state, but no migration of execution-state. However, run-time repartitioning doesn't want to wait with object relocation until that object and eventually all objects involved in the execution of that object are passive. Instead it aims to handle the triggers for object repartitioning *immediately*. As a consequence existing methods for repartitioning must be adapted to be applied at any point in an ongoing distributed computation. As such, it is necessary to support object relocation with migration of execution-state. Migration of execution-state is in the literature often referred to as strong thread migration [6]. The fact that the Voyager ORB does not support strong thread migration is not just a missing feature, but the real problem is that migration of the execution-state is simply not supported by current Java technology.
To solve this we developed a byte code transformer and an associated management subsystem that enables an external control instance (such as the above load balancing monitor) to capture and reestablish the execution-state of a running distributed application. We call this *(de)serialization of distributed execution-state*. The byte code transformer instruments the application code by inserting code blocks that extract the execution-state from the application code. The management subsystem, which is invoked by the inserted codes, is responsible for managing the captured execution-state efficiently. The management subsystem provides operations by which an external control instance can initiate serialization of the distributed execution-state at its own will.
It is important to know that we solely focus on distributed applications that are developed using conventional ORBs such as Java Remote Method Invocation (RMI) or Voyager. Programmers often use these middleware platforms because of their object-based Remote Procedure Call (RPC) like programming model, which is very similar to the well-known object-oriented programming style.

1.1 Important aspects of our work

Serialization of a distributed execution-state. In this paper we first describe how we realized serialization of a distributed execution-state. Note that in the past, several algorithms have been proposed to capture the execution state of Java Virtual Machine (JVM) threads in serialized form. Some require the modification of the JVM [2]. Others are based on the modification of source code [6]. Some models rely on byte code rewrite schemes, e.g., [10][15]. We too had already implemented such a byte code rewrite algorithm called *Brakes* [13].
However, most of these schemes are presented in the domain of mobile agents systems. Whenever a mobile agent wants to migrate, it initiates the capturing of its own execution-state. As soon as the execution-state is serialized the agent migrates with its serialized execution-state to the target host where execution is resumed.

However, serialization of distributed execution-state of Java RMI-like applications introduces two aspects that are not covered in the migration scenario of mobile agents. First, the computational entities in Java RMI applications execute as distributed flows of control that may cross physical JVM boundaries, contrary to how conventional Java threads are confined to a single address space. As such, serializing the execution-state of such a distributed control flow introduces a lot of unexplored problems that are not an issue for the migration of mobile agents. The second aspect is that mobile agents initiate the capturing/reestablishment of their execution-state themselves, whereas capturing/reestablishment of distributed execution-state must often be initiated by an external control instance.

The Brakes thread serialization scheme is not designed for being initiated by such an external control instance. In this paper we describe how we have extended Brakes with a mechanism for *serialization of the execution-state of distributed control flows* that can be *initiated by an external control instance*.

Runtime repartitioning of distributed Java applications. Subsequently, we show how we used this serialization mechanism to implement a prototype for runtime repartitioning. The idea is that the load balancing monitor, that plays the role of external control instance here, captures the execution-state of an application whenever it wants to repartition that application and reestablishes the execution-state after the repartitioning is finished.

The advantage of having separate phases for migration of execution state and object migration is that objects can migrate independently of their (suspended) activity. Requests for object migration can immediately be performed, without having to wait for the objects to become passive. This is possible because, by using the serialization mechanism, application objects can be turned passive on demand by the monitor, while their actual execution-state is safely stored in the management subsystem of the serialization mechanism. So when the actual repartitioning takes place, all application objects are a priori passive. As a result, we can still use a conventional passive object migration to implement the run-time repartitioning prototype. In this paper we assume that the application's underlying ORB supports passive object migration, e.g. the Voyager ORB, but existing work [5] has shown that support for passive object migration can also be added to the application by means of byte code transformation.

Previous work [9][14] already offers support for run-time repartitioning, but this is implemented in the form of a new middleware platform with a dedicated execution model and programming model. A disadvantage of this approach is that Java RMI legacy applications, which have obviously not been developed with support for runtime repartitioning in mind, must partially be rewritten such that they become compatible with the programming model of the new middleware platform. Instead, our new approach is to develop a byte code transformer that transparently injects new functionality to an existing distributed Java application such that this application becomes automatically run-time repartition-able by the monitor. The motivation behind this approach taken is that programmers do not want to distort their applications to match the programming model of whatever new middleware platform.

Portability of the serialization mechanism. An important benefit of the serialization mechanism for capturing distributed execution-state is its portability: (1) byte code transformations integrate the required functionality transparently into existing Java applications. A custom class loader can automatically perform the byte code

transformation at load-time. (2) The serialization mechanism does require no modifications of the JVM. This makes the implementation portable on any system, as long as a standard JVM is installed on that system. (3) The serialization mechanism does require no modifications of the underlying ORB. It works seamlessly on top of any ORB with an RPC-like programming model, provided that our byte code transformation is performed before stub code generation.
However, a limitation is that our serialization mechanism is only applicable on top of a dedicated cluster of machines where network latencies are low and faults are rare. This is not directly a dependability of our approach, but rather a dependability of the RPC-like programming model: performing blocking calls on remote objects is after all only feasible on a reliable, high-bandwidth and secure network. As such our serialization mechanism is not well suited for run-time repartitioning of Internet applications or wireless applications.

1.2 Structure of the paper

This paper is structured as follows. In section 2 we present our mechanism for serializing a distributed execution-state for a Java RMI-based application. In section 3 we introduce our prototype for runtime repartitioning and demonstrate how it works by means of a concrete example application. In section 4 we evaluate the performance overhead and byte code blowup that is generated by our approach. Section 5 discusses related work. Finally we conclude and look to future work in section 6.

2 Distributed thread serialization

In this section we describe our mechanism for serializing a distributed execution-state for a Java RMI-based application. First we shortly describe the implementation of Brakes. We discuss the problem we encountered when trying to reuse Brakes for capturing distributed execution-state. Then we introduce the notion of distributed thread identity and show how we used it to extend Brakes for serialization of distributed execution-state of Java RMI-based applications. In section 2.3 we give an overview of the associated management subsystem that is responsible for managing the captured execution-state efficiently.

2.1 Brakes for JVM thread serialization

In Brakes the execution-state of a thread is extracted from the application code that is executing in that thread. For this, a byte code transformer inserts capturing and reestablishing code blocks at specific positions in the application code. We will refer to this transformer as the *Brakes transformer*.
With each thread two *flags* (called isSwitching and isRestoring) 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 both flags are off, 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 1, 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 1, 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.

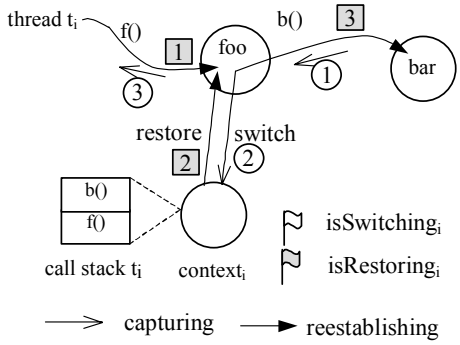


Fig. 1. Thread Capturing/Reestablishing in Brakes

A context manager per JVM manages both Context objects and flags. The inserted byte codes switch/restore the state of the current thread into/from its context via a context-manager-defined static interface. The context manager manages context objects on a per thread basis. So every thread has its own Context object, exclusively used for switching the state of that thread. The context manager looks up the right context object with the *thread identity* as hashing key. For more information about Brakes, we refer the reader to [13].

2.1.1 Problem with Brakes to capture distributed execution-state

This section describes the problem we encountered when trying to reuse Brakes for capturing distributed execution-state. In Brakes, execution-state is saved per local JVM thread. This works well for capturing local control flow but not for capturing a control flow that crosses system boundaries. Figure 2 illustrates the problem.
Once thread t_i in the example 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. 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

control flow. The context manager is however not aware of this logical connection between threads t_i and t_j. As a consequence Brakes will manage contexts and flags of these JVM threads as separate computational entities, although they should be logically connected. Without this logical connection, it becomes difficult to robustly capture and reestablish a distributed control flow as a whole entity. 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 control flow.

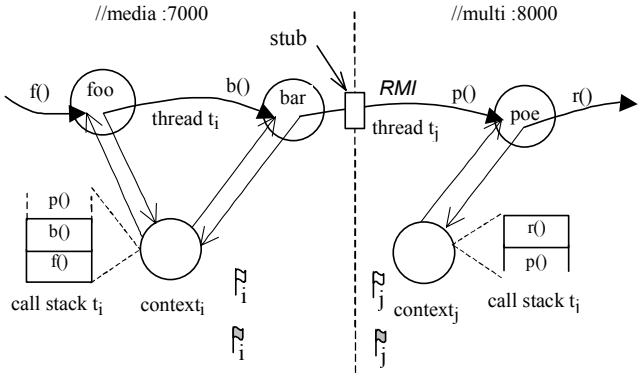


Fig. 2. Context per JVM Thread

2.2 Distributed thread identity to the rescue

A Java program is executed by means of a JVM thread. Such a thread is the unit of computation. It is a sequential flow of control within a single address space, i.e. JVM. We introduce the notion of *distributed thread* [3] as a logical sequential flow of control that may span several address spaces, i.e. JVMs. A distributed thread is physically implemented as a concatenation of local JVM threads, sequentially performing remote method invocations when they transit JVM boundaries. For example, in Figure 2 the distributed thread that spans the JVMs on media and multi is the concatenation of the JVM threads t_i and t_j.

In a local execution environment, i.e., for centralized programs that run on one JVM, the JVM thread identifier offers a unique reference for a single computation entity. In a distributed environment however, a new JVM thread is created whenever the control flow crosses system boundaries. Thereby *logical thread identity* gets lost.

To solve this problem, we extend Java programs with the notion of *distributed thread identity*. Propagation of a globally unique distributed thread identity provides a uniform mechanism to refer to that distributed thread as one and the same computational entity.

We implemented distributed thread identity by means of byte code transformation based on M. Dahm' BCEL [4]. Hereafter we will refer to this transformer as the *DTI transformer*. 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);
 …                               …
}                               }
```

This way the distributed thread identity is automatically propagated with the control flow along the method call graph.
Applying the byte code transformation to applications developed with an off-the-shelf Object Request Broker demands some attention. The programmer must be aware of generating the stub classes for the different remote interfaces only after our byte code transformation has been applied. This to make sure that stubs and skeletons (that are automatically generated) would propagate distributed thread identity appropriately.
The identity of a distributed thread is assigned at creation time. This behavior is encapsulated in the D_Thread class. The DTI-transformer wraps each Java Thread object in a D_Thread object which serves as abstraction for creating a new distributed thread. For more details about distributed threads and distributed thread identity we refer to [16].
Since distributed thread identity is now available in every method frame as the D_Thread_ID argument, it can be inspected by the context manager of Brakes. The only adjustment required to make it work, is that the inserted capturing and reestablishing code blocks of Brakes must pass the D_Thread_ID argument to the context manager along its static interface.

2.3 Efficient management of serialized distributed execution-state

The above adjustments allow us to build an associated distributed management system, illustrated in Figure 3 that manages captured execution-state on a per distributed thread basis. The management subsystem consists of a context manager for each JVM where the distributed application executes, to which we will refer as *local context managers*. Capturing and restoring code blocks still communicate with the static interface of the local context manager, but the captured execution-state is now managed per distributed thread by one central manager, the *distributed thread manager*.
To further deal with the problem that Brakes is not designed for capturing distributed execution-state, we also had to rearrange the management of the isSwitching and isRestoring flags. First, while in Brakes there was a separate isSwitching and isRestoring flag for each JVM thread, we now manage only one isSwitching and one isRestoring flag for the entire distributed execution-state of the application. Both flags are stored as static global variables on the distributed thread manager and are replicated with strong consistency on each local context manager. Furthermore we introduce a new flag, isRunning, associated with each individual distributed thread

that marks the start of capturing and the end of reestablishing the execution-state of that distributed thread.

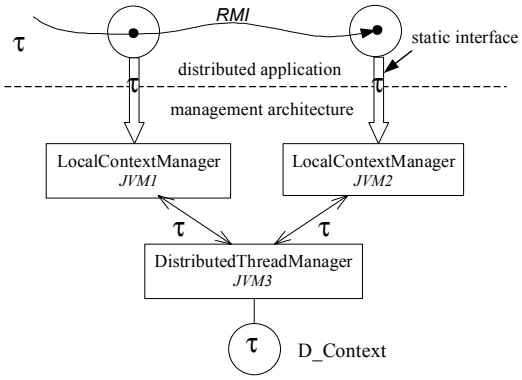


Fig. 3. Distributed Architecture of the Context Manager

A positive side effect of the rearrangement of flags is that we drastically reduce the overhead during normal execution. When inspecting the global isSwitching and isRestoring flags during normal execution, no costly hashtable lookup on distributed thread identity is performed anymore. Only during capturing and reestablishment the isRunning flag is looked up with D_Thread_ID as hashing key; however these lookups do not occur during normal execution. This choice may seem to be a trade-off between efficiency and flexibility. The rearrangement of flags results in a less flexible mechanism that can only capture execution-state at the level of the whole distributed execution state of the application. It is not possible to capture one distributed thread, without stopping other distributed threads. However, this coarse-grained scale is exactly what we want: it does not make sense to capture one thread, without stopping another thread when they are executing in the same application objects. In section 4.1 we discuss performance overhead more in detail.

External initiation of capturing and reestablishment. The distributed thread manager offers a public interface that enables an external control instance to initiate the capturing and reestablishment of distributed execution-state. However, the Brakes thread serialization mechanism is designed in the context of mobile agents and as such it is not designed for being initiated by an external control instance.

To deal with this problem, we extended the Brakes transformer to insert extra byte codes at the beginning of each method body to verify whether there has been an external request for capturing the execution-state. We will refer to this code as `{external capturing request check}`.

Capturing of execution-state is started by calling the operation `captureState()` on the distributed thread manager. This method sets the isSwitching flag on all local context managers through broadcast. As soon as a distributed thread detects the isSwitching flag is set, (inside the first executed `{external capturing request check}` code) the distributed thread sets off its isRunning flag and starts switching itself into its context. Reestablishment of execution 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. Each distributed thread detects immediately that the isRestoring flag is set, and thus restores itself from the context. Once the execution-state is reestablished the distributed thread sets on its isRunning flag (inside the `{external capturing request check} code`) and resumes execution. When all distributed threads execute again, the distributed thread manager sets the isRestoring flag off on all local context managers through broadcast.

3 Run-time repartitioning at work

In this section we present our prototype for run-time repartitioning and demonstrate it for a simple text translator application. First we describe the process of run-time repartitioning. Then we give a sketch of the prototype. Next we illustrate the byte code transformations. Finally we explain the process of run-time repartitioning by means of an example.

3.1 Runtime repartitioning

Run-time repartitioning aims to improve the global load balance or network communication overhead by repartitioning the object configuration of the application over the available physical nodes at run-time. We distinguish between 4 successive phases in the run-time repartitioning process. In the first phase, a load balancing monitor allows an administrator to monitor the application's execution and let him decide when to repartition the application objects over the available physical nodes. In the second phase, the monitor invokes the distributed thread manager to capture the distributed execution-state of the application. After this, the execution of all application objects is temporarily suspended and the corresponding distributed thread states are stored as serialized data in the context repository of the distributed thread manager. In the third phase, the monitor carries out the initial request for repartitioning by migrating the necessary objects over the network. In the final and fourth phase, the monitor invokes the distributed thread manager to reestablish the distributed execution-state of the application. As soon as the execution-state is reestablished the application continues where it left off.

3.2 Prototype

In the run-time repartitioning prototype, the serialization mechanism is integrated with a simple load balancing monitor. We demonstrate the repartitioning prototype for a simple text translator system, see Figure 4. The text translator is composed with a number of objects that can be distributed over some hosts. For each translation a new distributed thread is started. A client sends the text with a source and target language to a Server object. The Server forwards the job to a ParserBuilder, who sends each sentence for translation to a Translator. The Translator uses a Dictionary object for the translation of individual words. As soon as a sentence is translated the

Translator returns it to the ParserBuilder. The ParserBuilder assembles the translated text. Finally the translated text is returned to the Server who sends it back to the client.

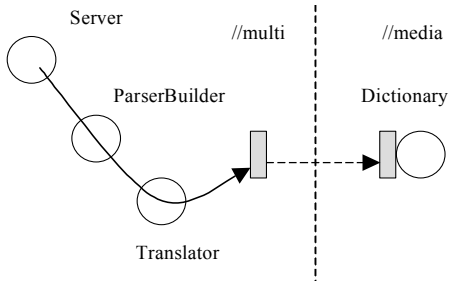


Fig. 4. Prototype for Run-time Repartitioning

Figure 5 gives a snapshot of the load balancing monitor. The monitor offers a GUI to an administrator that enables to do run-time repartitioning. The left and middle panels show the actual object distribution of the running application. The panels on the right show the captured context objects per distributed thread after a repartitioning request.
Since only passive object migration, i.e., code and data migration, but no migration of runtime information like the program counter and the call stack, is necessary during the third phase of the run-time repartitioning process we used the mobile object system Voyager 2.0 [8] as distributed programming model in our prototype.

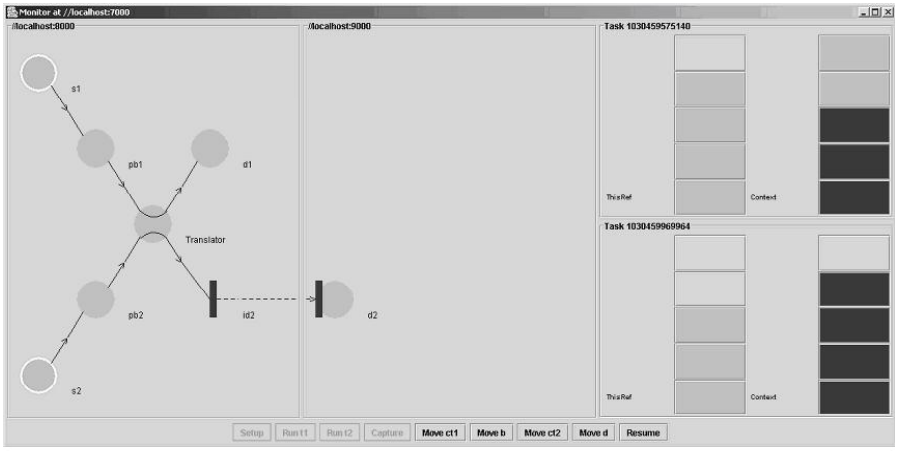


Fig. 5. Snapshot of the Runtime-Repartitioning Monitor

To enable capturing and reestablishment of distributed execution-state during phases two and four, the implementation code of the text translator system must be hauled through our byte code transformer.
Finally, the distributed architecture of the associated management subsystem as shown in Figure 3 defines an abstract framework that must be instantiated by a concrete distributed implementation. We used Voyager for this too, but another

distribution platform like Java RMI was also possible. Thus in our prototype LocalContextManager and DistributedThreadManager are implemented as Voyager objects.

3.3 Byte code transformations

Before illustrating the process of run-time repartitioning we first give an overview of the transformation of the application code. We limit the extract to the principal code as in Figure 6. The bold marked code is inserted byte code. Each method signature as well as each method invocation is extended with an extra D_Thread_ID argument by the DTI transformer. The Brakes transformer has inserted code blocks for the capture and restore of the execution-state of a distributed thread.

```
class Translator {
  Sentence analyze
     (Sentence sentence, D_Thread_ID threadID){
     {isRestoring code block}
     {external capturing request check}
     {switching code block}
     Sentence tSentence = new Sentence();
     Word word, tWord;
     sentence.resetCursor(threadID);
     {switching code block}
     while(!sentence.endOfSentence(threadID)){
         {switching code block}
         word = sentence.next(threadID);
         {switching code block}
**       tWord =
           dictionary.translate(word, threadID);
         {switching code block}
         tSentence.add(tWord, threadID);
         {switching code block}
     }
     return tSentence;
  }
  ...
  private Dictionary dictionary;
}

class Dictionary {
* public Word
    translate(Word word, D_Thread_ID threadID){
      {isRestoring code block}
      {external capturing request check}
      {switching code block}
  ...
  }
...
}
```

Fig. 6. Application code after Byte code Transformation

3.4 An example of run-time repartitioning

We will now explain the process of run-time repartitioning starting from Figure 4. Suppose the administrator decides to migrate the Dictionary object during the

translation of a text. At a certain moment, lets say when the control flow enters the `translate()` method in Dictionary, the administrator pushes the capture button on the monitor. This point is marked in Figure 6 with *. At that moment the execution-state of the distributed thread is scattered over two hosts as illustrated in Figure 7.
Pushing the capture button invokes `captureState()` on the distributed thread manager that sets the isSwitching flag. The distributed thread detects this in the `{external capturing request check}` code. Immediately it set off its isRunning flag and saves the execution-state of the `translate()` method. This includes: (1) the stack frame for `translate()` (i.e. the only frame for thread t2 on the call stack, see Figure 7); (2) the index[1] of the last invoked method in the body of translate (i.e. zero for `{external capturing request check}`); (3) the object reference to the Dictionary object (i.e. the this reference). For reasons of efficiency the execution-state is buffered per stack frame by the local context manager until completion of switching. Then the local manager forwards the serialized data to the distributed manager who stores it into the context repository of the distributed thread with the given D_Thread_ID.

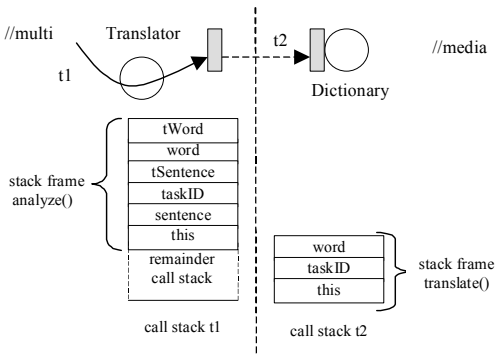


Fig. 7. Distributed Execution-state before Capturing

The last instruction of the `{switching code block}` is a return. This redirects the control to the previous frame on the call stack of the distributed thread, in our case the `analyze()` method. The control flow returns from host tot host (i.e. from media to multi, see Figure 7) which means that the execution-state of the JVM thread t2 now is completely saved. In the code of Figure 6, we then reach the point marked as **. Next the execution-state for `analyze()` is saved, i.e.: (1) the stack frame for `analyze()` (i.e. the top frame of JVM thread t1 as in Figure 7): (2) the index of the last invoked method in the body of analyze (i.e. 4, see Figure 6); (3) the object reference to the Translator object. As soon as the buffered data is written to the context repository of the distributed thread another return at the end of the `{switching code block}` redirects the control flow to the previous frame on the call stack. Subsequently, the execution-state for that method is saved. This process recursively continues until the

[1] this index refers to the number the Brakes transformer associates with the subsequent invoke-instructions in the body of each method, starting with 0 for {external capturing request check} code, 1 for the first invoke-instruction and so on; the index of the last performed invoke-instruction is saved in the context to remember which methods where on the stack

JVM thread t1 returns to the `run()` method of D_Thread. At that time the DistributedContext contains the complete distributed execution-state of the distributed thread.
Once the complete distributed execution-state is saved the Dictionary object can be migrated from media to multi. To this purpose the administrator pushes the corresponding migrate-button on the monitor. As explained in section 3.2, we used Voyager as distribution platform for our prototype. Voyager dynamically transfers the object references from remote to local and vise versa.
Once the repartitioning is ready the administrator can push the resume button which invokes `resumeApplication()` on the distributed thread manager. That turns off the isSwitching flag and sets the isRestoring flag. Next a new JVM thread is created at multi to resume the translation. During the reestablishing process the relevant methods are called again in the order they have been on the stack when state capturing took place. The new thread takes the original D_Thread_ID with it. This is the key mechanism for reconstructing the original call stack. Each time inserted byte code reestablish the next stack frame the managers use the distributed thread identity to select the right context object for that particular distributed thread.
Figure 8 illustrates the reestablishment of the last two frames in our example.

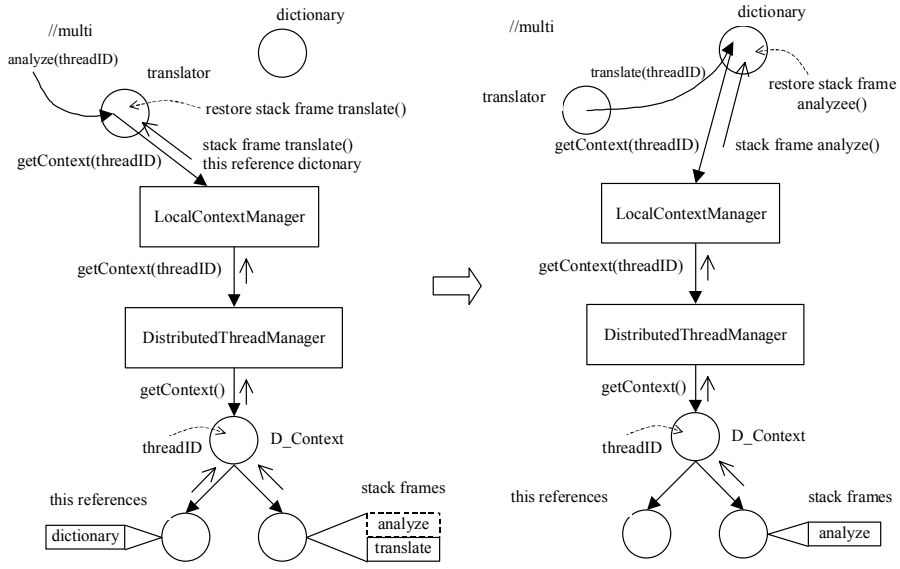


Fig. 8. Reestablishing a Distributed Execution-state

When `analyze()` on Translator is invoked, the isRestoring code block will be executed, based on the actual state of the flags (isRestoring = on, isRunning = off). The inserted byte codes restore the values of the local variables of the `analyze()` frame one by one via the local context manager (see the left part of Figure 8). At the end the index of the next method to invoke is picked up. For the `analyze()` frame an index 4 was earlier saved, so the fourth method, i.e. `translate()`, must be invoked

on the Dictionary object. The managers pickup the reference to this object and `translate()` will be invoked. Again the restoring code restores the local execution-state of the `translate()` method (the right part of Figure 8). This time the index for the next method is zero. This is the signal for the context manager to reset the isRunning flag of the current distributed thread so that it can resume its normal execution. At this point the expensive remote interaction between the translator and the dictionary objects is transferred into a cheap local invocation.

Note that in the example the execution-state is captured with the translator object waiting on a pending reply of a remote method invocation (i.e. `dictionary.translate()`, see Figure 6). This illustrates the advantage of keeping execution and object migration completely orthogonal to each other.

4 Evaluation

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. To get a representative picture, we did tests on different types of applications. At the end we look to limitations of our current implementation and restrictions of the model.

4.1 Measurements

Blowup of the byte code. 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.
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 sequence of non-nested methods

The middle band of Table 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.

Table 1. Measurement results for three kinds of applications

	Byte code blowup (Bytes)			Performance overhead (ms)		
	Low dg	Nested	Sequent	Low dg	Nested	Sequent
Original	377	431	431	190	811	1011
DTI	399	616	524	192	852	1054
DTI + Brakes	573	718	991	199	949	1314

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. This overhead is a consequence of the execution of inserted byte code. The right band of Table 1 shows the test results. For distributed thread identity we measured an average overhead of only 3 %. For full serialization functionality we get an average overhead of 17 %, a quite acceptable result. Note that "normal" applications typically are programmed in a nested invocation style. As such, the results for the nested application are a good indication for blowup and performance overhead in practice.
It is difficult to compare our measurement results with other systems, since to our knowledge, no related system truly covers functionality for serialization of *distributed* execution-state as our system does. In section 5 we discuss related work.

4.2 Limitations of the current implementation

Our byte code transformers have some limitations we intend to eliminate in the future. Although possible, we have not yet implemented state capturing during the execution of an exception handler. The major difficulty here is dealing with the `finally` statement of a `try` clause. Currently our byte code transformer throws away all debugging information associated with a Java class. This affects the ability to debug a transformed class with the source-code debugger.
The DTI byte code transformer encapsulates each user defined JVM thread into a D_Thread, but currently ignores other JVM thread related code. Thus our current model doesn't support aspects as e.g., thread locking in synchronized code sections.

4.3 Restrictions of the model

Our model is intended and only applicable for applications running on top of a dedicated cluster of machines where network latencies are low and faults are rare. This is not directly a dependability of our approach, but rather a dependability of the RPC-like programming model: performing blocking calls on remote objects is after all only feasible on a reliable, high-bandwidth and secure network. As such our serialization mechanism is not well suited for run-time repartitioning of Internet applications or wireless applications.
Furthermore, in section 2.3 we already mention that the granularity of our repartitioning algorithm is at JVM level. When the isSwitching flag is set all running distributed threads are suspended together irrespective of whatever application they

belong. Thus applications, which its classes are transformed with our byte code transformer and that executes on the set of involved JVMs will be suspended together. Since we extract thread execution-state at byte code level we cannot handle a method call that causes a native method to be placed on the thread stack. Thus programs that use reflection do not work properly with our repartitioning model. Therefore we actually don't transform JDK libraries and JDK method calls in the application code.
In our prototype we transformed the application classes before deployment, but it is possible to defer this byte code transformation until run-time. In Java, this can easily be realized by implementing a custom classloader that automatically performs the transformation. However, the overhead induced by the transformation process then becomes a relevant performance factor.

5 Related work

We discuss related work according to the related fields that touches our work.
Distributed threads. D. Jensen at CMU already introduced the notion of distributed thread in the Alpha distributed real-time OS kernel [3]. The main goal of distributed threads in the Alpha kernel was integrated end-to-end resource management based on propagation of scheduling parameters such as priority and time constraints. In our project we adopted the notion of distributed thread at the application level. This allows the distributed management subsystem to refer to a distributed control flow as one and the same computational entity.
Strong thread migration. Several researchers developed mechanisms for strong thread migration in the context of mobile agents. S. Funfrocken at TU Darmstadt [6] has implemented a transparent serialization mechanism for local JVM threads by processing the source code of the application. He used the Java exception mechanism to capture the state of an ongoing computation. Source code transformation requires the original Java files of the application. Besides, it is much easier to manipulate the control flow at byte code level than at source code level. Therefore, byte code transformation is more efficient especially in terms of space overhead.
Sakamoto et al. [10] also developed a transparent migration algorithm for Java application with the Java exception mechanism, but these researchers use also byte code transformation. We have chosen not to use the exception mechanism, since entries on the operand stack are discarded when an exception is thrown, which means that their values cannot be captured from an exception handler. Sakamoto et.al. solved this problem by copying al those values in extra local variables before method invocation, but this causes much more space penalty and performance overhead.
In her dissertation [15], W. Tao proposes another portable mechanism to support thread persistence and migration based on byte code rewriting and the Java exception mechanism. Tao's mechanisms supports synchronized execution state saving.
Multi-threading for distributed mobile objects in FarGo. Abu and Ben-Shaul integrated a multi-threading model for distributed and mobile objects in the FarGo framework [1]. A FarGo application consists of a number of 'complets'. Complets are similar to components. They are the unit of relocation in the model. The distributed mobile thread model of FarGo is based on a thread-partitioning scheme. Programmers

must mark a migratable complet as thread-migratable (T-migratable) by implementing the empty T_Migratable interface. The FarGo compiler uses this interface to generate proper thread partitioning code. Thread partitioning is integrated in the complet reference architecture. When a T_Migratable complet is referenced the invoking thread waits, and a new thread is started in the referenced complet. The migration itself is based on the source code transformation of Funfrocken' migration scheme.
Fargo introduces a new distributed programming model to support object migration, while our mechanism can be used for any legacy Java RMI application.
Byte code transformations for distributed execution of Java applications. The Doorastha system [5] allows implementing fine-grained optimizations for distributed applications just by adding code annotations to pure Java programs. By means of these annotations it is possible to dynamically select the required semantics for distributed execution. This allows a programmer to develop a program in a centralized (multi-threading) setting first, and then prepare it for distributed execution by annotation. Byte code transformation will generate a distributed program whose execution conforms to the selected annotations.
Researchers at the University of Tsukuba, Japan [12] developed a system named Addistant, which likewise enables the distributed execution of a Java program that originally was developed to run on a single JVM. Addistant guarantees that local synchronized method calls of one distributed control flow are always executed by one and the same JVM thread. Therefore it establishes a one-to-one communication channel (as thread local variable) for the threads that take part in such an invocation pattern. In this approach it isn't necessary to pass distributed thread identity along the call graph of the distributed control flow, but for a run-time repartitioning system, thread identity must be propagated with every remote invocation anyway.

6 Conclusion and future work

In this paper we presented a mechanism for serialization of a distributed execution-state of a Java application that is developed by means of a distributed control-flow programming model such as Java RMI. This mechanism can serve many purposes such as migrating execution-state over the network or storing it on disk. An important benefit of the serialization mechanism is its portability. It can be integrated into existing applications and requires no modifications of the JVM or the underlying ORB. However, because of its dependability on the control-flow programming model, our serialization mechanism is only applicable for distributed applications that execute on low latency networks where faults are rare.
Our contribution consists of two parts. First we integrated Brakes, our existing serialization mechanism for JVM threads, in a broader byte code translation scheme to serialize the execution-state of a distributed control flow. Second we integrated a mechanism to initiate the (de)serialization of the distributed execution-state from outside the application.
We applied the serialization mechanism in a prototype for run-time repartitioning of distributed Java applications. Our repartitioning mechanism enables an administrator to relocate application objects at any point in an ongoing distributed computation.

Often byte code transformation is criticized for blowup of the code and performance overhead due to the execution of inserted byte code. Based on a number of quantitative analyses we may conclude that the costs associated with our byte code translation algorithm are acceptable.
The latest implementation of the run-time repartitioning tool is available at:
http://www.cs.kuleuven.ac.be/~danny/DistributedBrakes.html.
Some limitations of our serialization mechanism for a distributed execution-state have to be solved. Finally it ‘s our intention to build a complete tool for run-time repartitioning for distributed Java RMI applications. Therefore we have to extend our current monitoring and management subsystem with several other features such as functionality for dynamic adaptation of object references after migration, support for different load balancing algorithms and an application adaptable monitor.

7 References

[1] M. Abu, I. Ben-Shaul, “A Multi-Threading model for Distributed Mobile Objects and its Realization in FarGo”, Proceedings of ICDCS’2001.

[2] S. Bouchenak, “Pickling threads state in the Java system”, Proceedings of ERSADS'99.

[3] R Clark, D.E. Jensen, and F.D Reynolds, “An Architectural Overview of the Alpha Real-time Distributed Kernel”, Proceedings of the USENIX Workshop, 1992.

[4] M. Dahm, “Byte Code Engineering”, Clemens Cap, editor, Proceedings JIT'99.

[5] M. Dahm, “The Doorastha system”, Technical Report B-I-2000 Freie Universität Berlin.

[6] S. Funfrocken, “Transparent Migration of Java-based Mobile Agents”, Proceedings of the 2nd International Workshop on Mobile Agents, 1998.

[7] K. C. Nwosu, “On Complex Object Distribution Techniques for Distributed Systems”, ICCI'94.

[8] ObjectSpace Inc., “VOYAGER 2.0”, http://www.objectspace.com/products/voyager/, 1998.

[9] B. Robben, “Language Technology and Metalevel Architectures for Distributed Objects”, Ph.D K.U.Leuven - Belgium, ISBN 90-5682-194-6, 1999.

[10] T. Sakamoto, T. Sekiguchi, A. Yonezawa, “Bytecode Transformation for Portable Thread Migration in Java”, Proceedings of International Symposium on Agent Systems and Applications / Mobile Agents, ASA/MA'2000.

[11] W. Shu, M.Wu, "An Incremental Parallel Scheduling Approach to Solving Dynamic and Irregular Problems", Proceedings of the 24th International Conference on Parallel Processing, 1995.

[12] M. Tatsubori, T. Sasaki, S. Chiba, K. Itano, “A Bytecode Translator for Distributed Execution of ‘Legacy’ Java Software”, ECOOP’2001.

[13] E. Truyen, B. Robben, B.Vanhaute, T. Coninx, W. Joosen and P. Verbaeten, KULeuven, “Portable Support for Transparent Thread Migration in Java”, Proceedings of International Symposium on Agent Systems and Applications / Mobile Agents, ASA/MA'2000.

[14] E. Truyen, B.Vanhaute, B. Robben, F. Matthijs, E. Van Hoeymissen, W. Joosen, P. Verbaeten, "Supporting Object Mobility - from thread migration to dynamic load balancing", OOPSLA'99 Demonstration.

[15] Wei Tao, "A Portable Mechanism for Thread Persistence and Migration", PhD thesis, University of Utah, http://www.cs.utah.edu/~tao/research/index.html, 2001.

[16] D. Weyns, E. Truyen, P. Verbaeten, “Distributed Threads in Java”, Proceedings of the International Symposium on Distributed and Parallel Computing, ISDPC '2002.