SCALABLE COMPUTING: PRACTICE AND EXPERIENCE
Volume 7, Number 4, pp. 37–51. http://www.scpe.org
ISSN 1895-1767

LEVERAGING STRONG AGENT MOBILITY FOR AGLETS WITH THE *MOBILE JIKESRVM* FRAMEWORK

RAFFAELE QUITADAMO*, LETIZIA LEONARDI* , AND GIACOMO CABRI*

Abstract. Mobility enables agents to migrate among several hosts, becoming active entities of networks. Java is today one of the most exploited languages to build mobile agent systems, thanks to its object-oriented support, portability and network facilities. Nevertheless, Java does not support *strong mobility*, i. e. the possibility of relocating running threads along with their execution state; challenges arising from implementing strong mobility upon the JVM has led to the choice of a weaker form of agent mobility (i. e. *weak mobility*): although in many agent scenarios (e.g. in simple reactive agents) weak mobility could be enough, it usually complicates programming parallel and distributed applications, as it forces developers to structure their agent-based programs as sort of FSMs (Finite State Machine). In this paper, we present our *Mobile JikesRVM* framework to enable strong Java thread migration, based on the IBM Jikes Research Virtual Machine. Moreover, we show how it is possible (and often desirable) to exploit such a framework to enrich a Mobile Agent Platform, like the IBM Aglets, with strong agent mobility and to leverage software agents potential in parallel and distributed computing.

Key words. Java Threads, distributed system, thread migration, strong mobility, IBM JikesRVM

1. Introduction. Agents are autonomous, proactive, active and social entities able to perform their task without requiring a continue user interaction [22]; thanks to the above features, the agent-oriented paradigm is emerging as a feasible approach to the development of today's complex software systems [16].

Moreover, agents can be *mobile*. The concept is simple and elegant: an agent that resides in one node migrates to another node where execution is continued. Code mobility [13] is reshaping the logical structure of modern distributed systems as it enriches software components (in particular, agents) with the capability to dynamically reconfigure their bindings with the underlying execution environment. The main advantages of mobile computations, be they agent-based or not, are as follows:

1. *Load balancing*: distributing agent-based computations among many processors as opposed to one processor gives faster performance for those tasks that can be fragmented.
2. *Communication performance*:agents which interact intensively can be moved to the same node to reduce the communication cost for the duration of their interaction.
3. *Availability*: agents can be moved to different nodes to improve the service and provide better failure coverage or to mitigate against lost or broken connections.
4. *Reconfiguration*: migrating agents permits continued service during upgrade or node failure.
5. *Location independence*: an agent visiting a node can rebind to generic services without needing to specifically locate them. Agents can also move to take advantage of services or capabilities of particular nodes.

With regard to mobility, we have to distinguish between *strong mobility*, which enables the migration of code, data and execution state of execution units (for instance, *threads*), from *weak mobility*, which migrates only code and data [13]. From the complexity point of view, *weak mobility* is quite simple to implement using well-established techniques like network class loading or object serialization [29]. However, weakly mobile systems, by definition, discard the execution state across migration and hence, if the application requires the ability to retain the thread of control, extra programming effort is required in order to manually save the execution state. The migration transparency offered by *strong mobility* systems has instead a twofold advantage: it allows a more natural sequential programming style, without the need to awkwardly structure the code with recovery points or flags; moreover, it is more suited to the requirements of many distributed and parallel applications, in which complex computations (e.g. scientific calculations) make manual state capturing (and recovering) somehow unfeasible or, at least, tedious.

Thanks to its portability and network facilities, Java is today the most exploited language to develop mobile agents, and in fact several Java-based Mobile Agent Platforms (MAP) exist [15, 2, 30]. Unfortunately, current standard Java Virtual Machines (JVMs) do not support strong thread migration natively. Thus, despite the advantages above, most mobile agent systems support only weak mobility and the reason lies mainly in the complexity issues of strong mobility and in the insufficient support of existing JVMs to deal with the execution state. Therefore, in order to concretely touch the advantages of strong mobility in mobile agent applications, a suitable framework or "JVM enhancement" is advisable. In this paper, after introducing the motivations for this research work and surveying the related work on this topic (Section 2), we introduce our Java framework (called *Mobile JikesRVM* [23]) to enable thread migration (Section 3), based on the

*Dipartimento di Ingegneria dell'Informazione, University of Modena and Reggio Emilia, Via Vignolese, 905, 41100 Modena, Italy {quitadamo.raffale, leonardi.letizia, cabri.giacomo}@unimore.it

IBM JikesRVM [4]. We found that this JVM offers great support for hosting a strongly mobile agent platform, and we prove this by showing (in Section 4) how the IBM Aglets platform has been successfully adapted to run on top of our mobility framework. Finally, (in Section 5) some first performance evaluation tests are reported. Section 6 concludes the paper and illustrates future research to be done on this framework.

2. Motivations and related work. This section introduces some motivations for our research on strong thread migration and its adoption in some mobile agent scenarios. It sketches some real applications that would benefit from the work explained later and provides a brief overview of proposed approaches in literature.

2.1. Motivations. The choice of strong thread mobility, when designing distributed Java applications, has to be carefully motivated, since it is not always the best one in most simple cases. Distributed and parallel computations can be considered perhaps the "killer application" of such technique.

For instance, complex elaborations, possibly with a high degree of parallelism, carried out on a cluster of servers would certainly benefit from a strong thread migration facility in the JVM. Well-know cases of such applications are mathematical computations, which are often recursive by their own nature (e.g. fractal calculations) and can be parallelized to achieve better elaboration times.

Another field of application for strong mobile threads is load balancing in distributed systems (e.g. in the Grid Computing field), where a number of worker nodes have several tasks appointed to them. In order to avoid overloading some nodes while leaving some others idle (for a better exploitation of the available resources and an increased throughput), these systems need to constantly monitor the execution of their tasks and possibly re-assign them, according to an established load-balancing algorithm. As we will see later, a particular kind of strong thread migration (called *reactive migration*), that we provide in our framework, fits very well the requirements of these systems.

2.2. Related work. Several approaches have been proposed so far to overcome the limitations of the JVM as concerns the execution state management. The main decision that each approach has to take into account is how to capture the internal state of threads, providing a fair trade-off between performances and portability. In literature, we can typically find two categories of approaches:

- modifying or extending the source code of existing JVMs to introduce APIs for enabling migration (*JVM-level approach*);
- translating somehow the application's source code in order to trace constantly the state of each thread and using the gathered information to rebuild the state remotely (*application-level approach*).

JVM-level approach. The former approach is, with no doubt, more intuitive because it provides the user with an advanced version of the JVM, which can completely externalize the state of Java threads (for thread serialization) and can, furthermore, initialize a thread with a particular state (for thread de-serialization). The kind of manipulations made upon the JVM can be several. The first proposed projects following the JVM-level approach like Sumatra [1], Merpati [32], JavaThread [7] and NOMADS [33], extend the Java interpreter to precisely monitor the execution state evolution. They, usually, face the problem of stack references collection modifying the interpreter in such a way that each time a bytecode instruction pushes a value on the stack, the type of this value is determined and stored "somewhere" (e.g., in a parallel stack). The drawback of this solution is that it introduces a significant performance overhead on thread execution, since additional computation has to be performed in parallel with bytecode interpretation. Other projects tried to reduce this penalization avoiding interpreter extension, but rather using JIT (Just In Time) re-compilation (such as Jessica2 [38]) or performing type inference only at serialization time (and not during thread normal execution). In ITS [8], the bytecode of each method in the call stack is analyzed with one pass at serialization time: the type of stacked data is retrieved and used to build a portable data structure representing the state. The main drawback of every JVM-level solution is that they implement special modified JVM versions that users have often to download; therefore they are forced to run their applications on a prototypal and possibly unreliable JVM.

Application-level approach. In order to address the issue of non-portability on multiple Java environments, some projects propose a solution at the application level. In these approaches, the application code is filtered by a pre-processor, prior to execution, and new statements are inserted, with the purpose of managing state capturing and restoration. In fact, the idea of these approaches is to transparently place a few control instructions, similar to recovery-points, that allow a thread to deactivate itself once it has reached one of them. Recovery-points are quite similar to entry points used in most Java MAPs (i. e., methods that are executed when an agent is reactivated at the destination host), even if the former ones enable a finer grain control than entry points. Unluckily, a thread cannot deactivate (or reactivate) itself outside of these recovery-points, which are also not customizable, thus a thread cannot really suspend itself in an arbitrary point of the computation. Some of these solutions rely on a bytecode pre-processor (e.g. JavaGoX [27] or Brakes [36]), while others provide source code translation (e.g. Wasp [12], JavaGo [28], Wang's proposal [37]). Two of them [28, 37] hide a weak

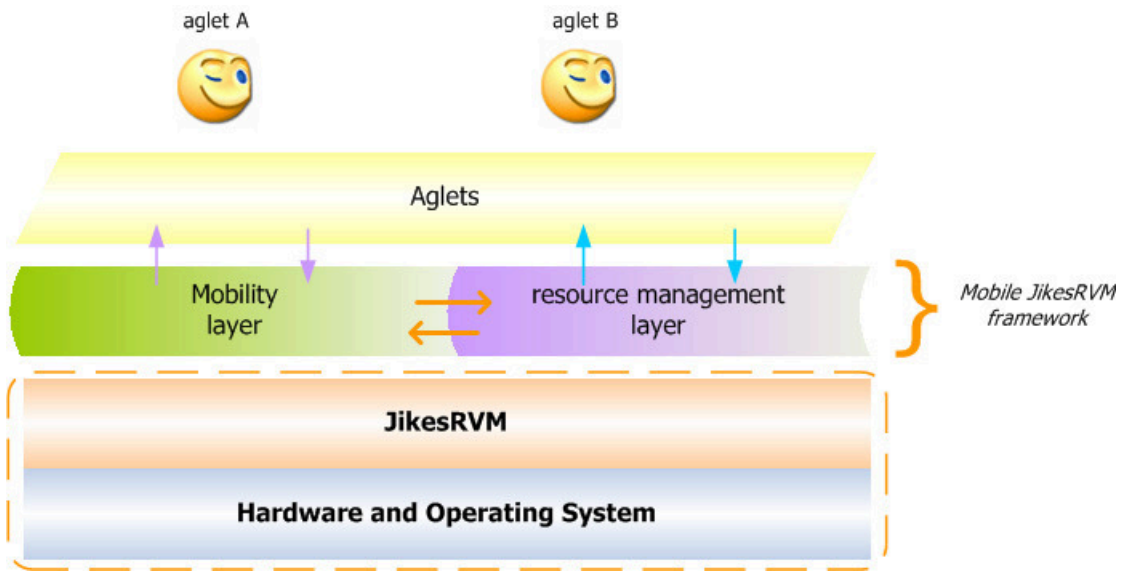

FIG. 3.1. *A layered view of Mobile JikesRVM*

mobility system behind the appearance of a strong mobility one: they, in fact, re-organize "strongly-mobile" written code into a "weakly-mobile" style, so that weak mobility can be used instead. Portability is achieved at the price of a slowdown, due to the many added statements.

Discussion. Starting from the above considerations, we have decided to design and implement a strong thread migration system able to overcome many of the problems of the above-explained approaches. In particular, our framework is written entirely in Java and it does neither suffer performance overheads, due to bytecode instrumentations, nor reliability problems, because the user does not have to download a new, possibly untrustworthy, version of JikesRVM. The framework is capable of dynamically installing itself on several recent versions of JikesRVM (we carried out successful tests starting from release 2.3.2). In fact, every single component of the migration system has been designed and developed to be used as a normal Java library, without requiring rebuilding or changing the VM source code. Therefore, our JikesRVM-based approach can be classified as a midway approach between the above-mentioned JVM-level and Application-level approaches. Other midway approaches [14] exploit the JPDA (Java Platform Debugger Architecture) that allows debuggers to access and modify runtime information of running Java applications. The JPDA can be used to capture and restore the state of a running program, obtaining a transparent migration of mobile agents in Java, although it suffers from some performance degradation due to the debugger intrusion.

3. A Layered View of our Framework. As already stated, in order to successfully exploit the benefits of mobile agents, an efficient and well-designed software support is needed on top of the bare JVM. Such a middleware should provide a precise, though flexible and customizable, answer to the questions of mobile applications developers. Following the seminal work of Fuggetta et al. [13], we can identify three main parts conceptually comprising a mobile code application:

- the *code segment* (i. e. the set of compiled methods of the application);
- the *data space*, a collection of all the resources accessed by the execution unit. In an object-oriented system, these resources are represented by objects in the heap;
- an *execution state*, containing private data as well as control information, such as the call stack and the instruction pointer.

From a mere technological standpoint, the capability to move code and regular objects is already a consolidated matter: the Java language provides very powerful tools to this purpose, like *object serialization* (used to migrate data in the heap) and *bytecode and dynamic class-loaders* (which facilitate the task of moving the code across distant JVMs, hosted by heterogeneous hardware platforms and operating systems). The main problem to tackle here is how to detach the execution state of a Java thread from its native environment and then re-install it at some other site. This requires diving into the internals of the JVM core and externalizing a complete representation of the running thread. Such functionality is provided in our framework by the *mobility layer* in Figure 3.1, which is built just upon JikesRVM. As we stressed earlier, this layer is installed dynamically into the runtime simply importing the *mobility* package, without requiring a

dedicated version of JikesRVM. Further details on this layer and its interactions with JikesRVM are the subject of the next subsection. Shifting to a more application-level point of view, every mobility system (both weak and strong) will sooner or later run across the non-negligible issue of *data space management* [13]: every thread has a set of referenced objects into the heap (i. e. the data space) and, when it migrates to the destination site, the set of bindings to passive (i. e. *resources*) and active objects (i. e. other threads) has to be rearranged. The way this set is rearranged depends on the nature of the resources (whether they can be migrated or not over the network), the type of the binding to such resources, as well as requirements posed by the application. The very fact that it eventually depends on application specific requirements makes it impossible to fully automate the choice of the adequate strategy, entailing the need for its programmatic specification. The *resource management* layer in Figure 3.1 is responsible for handling references to resources and relocating them according to such programmatic specifications. Some ongoing research ideas on this topic are outlined later in subsection 3.2. On top of the *Mobile JikesRVM* framework, it is possible to develop different distributed applications, which can benefit from the provided strong mobility and data space management support. IBM Aglets [2] is a well-known MAP, completely written in Java and open-source project. In Section 4 of this paper, we report on our effort to port this platform on Mobile JikesRVM, so that agents (i. e. aglets) are able to strongly migrate among network nodes: in our opinion, the possibility for agents to exploit the benefits of strong mobility, without many of its well-known drawbacks, can open new applicative scenarios for this paradigm.

3.1. The Mobility Layer. This layer contains a package of classes required to extend the runtime of JikesRVM and enable thread migration on top of it. JikesRVM [4] is now an open-source project, whose innovative and ambitious features are drawing researchers interest from all over the world. JikesRVM began life in 1997 at IBM T. J. Watson Research Center as a project with two main design goals: supporting high performance Java servers and providing a flexible research platform where novel VM ideas can be explored, tested and evaluated [3]. JikesRVM is almost totally written in the Java language, but with great care to achieving maximum performance and scalability exploiting as much as possible the target architectures peculiarities. The all-in-Java philosophy of this VM makes it very easy for researchers to manipulate or extend its functionalities. Furthermore, JikesRVM source code can be built, with a prior custom compilation, both on IA32 and on PPC platforms [17], but the bulk of the runtime is made up of Java objects portable across different architectures. For the sake of brevity, we will focus on those aspects that make JikesRVM an ideal execution environment for strongly mobile agents, overcoming the drawbacks and the limitations of many existing solutions. Further details can be obtained from its users guide [17].

As depicted in the UML excerpt diagram of Figure 3.2, the `MobileThread` class is the basic abstract class, through which thread migration services could be accessed. Users threads have just to subclass `MobileThread` and use some of the inherited methods to extract the execution state (i. e. `collectFrames()`) or to re-install it (i. e. `installFrames()`) at the destination site/host. It must be pointed out that in JikesRVM threads are full-fledged Java objects and are designed explicitly to be as lightweight as possible [3]. As well as many server applications need to create new threads for each incoming request, a Mobile Agent Platform has similar requirements since thousands of agents may request to execute within it.

While some JVMs adopted the so-called *native-thread model* (i. e. the threads are scheduled by the operating system that is hosting the virtual machine), JikesRVM designers chose the *green-thread model* [25]: Java threads are hosted by the same operating-system POSIX thread, implemented by a so-called *virtual processor*, through an object of class `VM_Processor` [5]. Each virtual processor manages the scheduling of its virtual threads (i. e., Java threads), represented by internal objects of the class `VM_Thread`. Moreover, each `java.lang.Thread` has a protected `vmdata` field, pointing to the corresponding instance of `VM_Thread`. When a `MobileThread` is instantiated by the application, it initially points to a standard internal `VM_Thread` object (see the dashed UML composition link between `Thread` and `VM_Thread` in Figure 3.2). This thread becomes truly mobile only when its `enableMobileThread()` method is invoked, since this method changes the reference to the original `VM_Thread` object to an instance of our special `VM_MobileThread` (see the UML composition link between `MobileThread` and `VM_MobileThread` in Figure 3.2).

Before going deeper into the details of the *mobility layer*, we report in Figure 3.3 a possible `migrate()` method, implemented by the user to perform migration of her threads. This method simply opens a socket towards a destination host, captures the execution state of that thread (in a chain of frames, as explained later) and serializes it through the socket stream. Please note that a migration/serialization unit in the example is composed of the thread instance and the chain of frames of its execution state, packed into a dedicated `Transport` object.

Let us suppose we have at destination another service thread, listening on a certain TCP port, whose task is to read incoming threads from the network and resume them locally. A possible method to perform this has been depicted in the excerpt of Figure 3.4. Deserializing the `Transport` object into memory implicitly creates a local instance of the

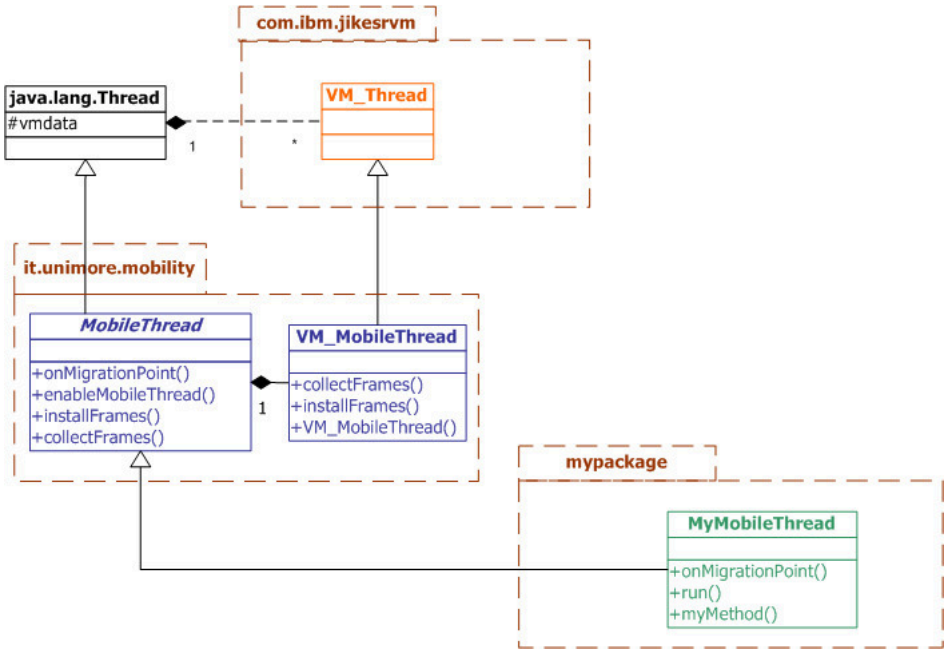

FIG. 3.2. *A simplified view of the mobility package and its dependencies (excerpt)*

```
private void migrate(String hostname, int port)
{
try {
      Socket s = new Socket(hostname, port);

      TransportObject t = new TransportObject();
      t.thread=this;
      t.framesChain=collectFrames(...);

      ObjectOutputStream oos =
              new ObjectOutputStream(s.getOutputStream());
      oos.writeObject(t);
      oos.flush();

      s.close();
      } catch (Exception e) {
      ...
      }
}
```

FIG. 3.3. *Building a mobile thread application with mobility (source machine)*

`MobileThread` object, which has to be manipulated in order to accept the received execution state. The task simply boils down to

- starting the deserialized thread and waiting for its auto suspension,
- installing the received frames in the chain into the suspended thread,
- resuming the thread locally.

If such phases are successfully carried out, the outcome will be that the thread will continue its execution from the next instruction following the `migrate()` method call of Figure 3.3.

```
void handleTransportObject(ObjectInputStream ois, ObjectOutputStream oos)
{
    /* Read the object from the socket */
    Object o = ois.readObject();

    /* Cast the deserialized object to a Transport object */
    TransportObject t = (TransportObject) o;

    /* Create a new autosuspending MobileThread */
    MobileThread newThread = t.thread;

    /* Make this thread autosuspended...  */
    newThread.enableMobileThread(true);
    newThread.start();

    /*... and wait for its suspension */
    while(!newThread.isAutoSuspended())
       Thread.yield();

    /* Install frames into the new thread */
    newThread.installFrames(t.framesChain);

    /* Resume the thread locally */
    newThread.resume();

}
```

FIG. 3.4. *Building a mobile thread application with mobility (destination machine)*

Capturing the execution state of a thread. When the above `collectFrames()` method is called on a `MobileThread` object, the framework starts a walk back through its call stack, from the last frame to the `run()` method of the thread. This jumping is shown schematically in Figure 3.5, where the stack is logically partitioned into three areas: (i) *internal preamble frames*, which are always present and do not need to be migrated; (ii) *user-pushed frames*, to be fully captured as explained later; (iii) *thread-switch internal frames*, which can be safely replaced at the destination and, thus, not captured at all. A special utility class, called `FrameExtractor`, has been implemented in the *mobility* framework, with the precise goal of capturing all the frames in the user area in a portable bytecode-level form. This class uses an *OSR extractor* to capture the frame state representation and returns it to the caller, ready to be serialized and sent to destination or to be check-pointed on disk.

The JikesRVM OSR Extractor. The OSR (On-Stack Replacement) extractor is another fundamental component of the framework: it draws inspiration from the OSR extractors provided by JikesRVM [11], though it has been re-written for the purposes of our project. The OSR technique was introduced in JikesRVM, with a completely different objective: enabling adaptive re-compilation of hot methods. In fact, JikesRVM can rely not only on a baseline compiler but also on an optimized one [9]. Every bytecode method is initially compiled with the baseline compiler, but when the Adaptive Optimization System (AOS) [6] decides that the current executing method is worth being optimized, the thread is drawn from the ready queue and the previous less-optimized frame is replaced by a new more-optimized frame. The thread is then rescheduled and continues its execution in that method. This technique was first pioneered by the Self programming language [10]. An innovative implementation of the OSR was integrated into the JikesRVM [11], which uses source code specialization to set up the new stack frame and continue execution at the desired program counter. The transition between different kinds of frames required the definition of the so-called *JVM scope descriptor* that is the compiler-independent state of a running activation of a method based on the stack model of the JVM [21]. When an OSR is triggered by JikesRVM, the scope descriptor for the current method is retrieved and is used to construct a method, in bytecode, that sets up the new stack frame and continues execution, preserving semantics.

Our modified OSR Extractor. The JikesRVM OSR frame extractor has been rewritten for the purpose of our mobility framework (i. e. the `OSR_MobilityExtractor`) to produce a frame representation, suitable for a thread migration context.

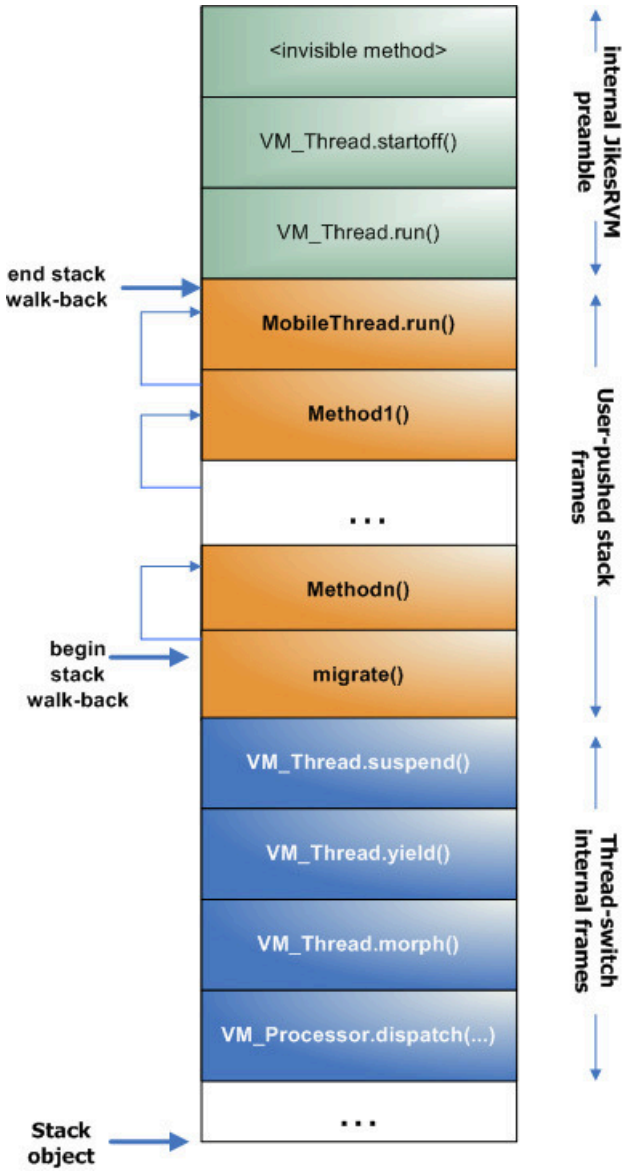

FIG. 3.5. *The stack walk-back of a suspended* `MobileThread`

The scenario we are talking about is a wide-opened one, where different machines running JikesRVM mutually exchange their `MobileThread` objects without sharing the main memory. We introduced, therefore, a portable version of the scope descriptor, called `MobileFrame`, whose structure is reported in Figure 3.6. While the OSR implementation in JikesRVM uses an internal object of class `VM_NormalMethod` to identify the method of the frame, we cannot make such an assumption; the only way to identify that method is through the triplet [method name, method descriptor, full class name] that is universally valid. This triplet (represented by the three fields `methodName`, `methodDescriptor` and `methodClass` in Figure 3.6) is used to refer the method at the destination (e.g. its bytecode must be downloaded if not locally available yet), maybe after a local compilation. The bytecode index (i. e. the `bcIndex` field) is the most portable form to represent the return address of each method body and it is already provided in JikesRVM by default OSR. Finally, we have two arrays (i. e. the `locals` and `stack_operands` fields) that, respectively, contain the values of local variables (including parameters) and stack operands in that frame. These values are extracted from the physical frame at the specified bytecode index and converted into their corresponding Java types (`int`, `float`, `Object` references and so on). In addition, it must be pointed out that the `OSR_MobilityExtractor` class fixes up some problems that we run across during our implementation: here, we think it is worthwhile mentioning the problem of uninitialized local

```
class MobileFrame {
    /** Name of the method which adds this frame*/
    public String methodName;

    /** Method descriptor
    e.g. (I)V for a method
    getting an integer and returning void */
    public String methodDescriptor;

    /** Fully qualified method class
    (e.g.mypackage.myClass)*/
    public String methodClass;

    /** The bytecode index (i.˜e. return address)
    within this method*/
    public int bcIndex;

    /** The local bytecode-level local variable
    including parameters */
    public MobileFrameElement[] locals;

    /** The value of the stack operands at the
    specified bytecode index */
    public MobileFrameElement[] stack_operands;

    // methods and static fields omitted
}
```

FIG. 3.6. *The main fields of the* `MobileFrame` *class*

variables. Default OSR extractor does not consider, in the JVM scope descriptor, those variables that are not active at the specified bytecode index. Nevertheless, these local variables have their space allocated in the stack and this fact should be taken into account when that frame is re-established at the destination.

To summarize, in our mobility framework, threads are serialized in a strong fashion: the `MobileThread` object is serialized as a regular object, while the execution state is transferred as a chain of fully serializable `MobileFrame` objects.

Resuming a migrated thread. The symmetrical part of the migration process is the creation, at the destination host, of a local instance of the migrated thread. This task should be appointed to some user listener thread like the one mentioned above, while in this section we are going to see how the thread is rebuilt in the *mobility layer*. This phase assumes that the target thread is suspended: this allows the infrastructure to safely reshape the current stack object of this thread, injecting one by one all the frames, belonging to the arrived thread. In more details, a new stack is allocated and it is filled in with the thread-switch internal frames, taken from the auto-suspended thread. Then, every `MobileFrame` object is installed, in the same order as they were read from the socket stream (i. e. from the `Methodn()` to `run()`, looking at Figure 3.5). The brand-new stack is closed with the remaining preamble frames, again borrowed from the auto-suspended thread. Now, the new stack has been prepared and the context registers are properly adjusted (pointers are updated to refer to the new stack memory). This stack takes the place of the old stack belonging to the auto-suspended thread (the old one is discarded and becomes garbage). The new `MobileThread` object, with its execution state completely re-established, can be transparently resumed and continues from the next instruction.

Proactive migration vs. reactive migration. In the previous code example, we have shown a kind of migration that has been defined [13] as *proactive migration*: i. e. the mobile thread autonomously determines the time and destination for its migration, explicitly calling a `migrate(URL destination)` method; another interesting, though quite tricky, kind of thread migration is *reactive migration*, where the threads movement is triggered by a different thread that can have some kind of relationship with the thread to be migrated, e.g. acting as a manager of roaming threads. Exploiting JikesRVM features, we successfully implemented both migration types, in particular the *reactive migration*. As anticipated in Subsection 2, an application, in which reactive migration can be essential, is a load-balancing facility in a distributed system.

If the virtual machine provides such functionality to authorized threads, a load monitor thread may want to suspend the execution of a worker thread A, assign it to the least overloaded machine and resume its execution from the next instruction in As code. This form of transparent externally-requested migration is harder to implement with respect to the proactive case, mainly because of its asynchronous nature. Proactive migration raises, in fact, less semantic issues than the reactive one, though identical to the latter from the technological/implementation point of view: in both cases we have to walk back the call stack of the thread, extract the meaningful frames and send the entire thread data to destination (as explained earlier). The fundamental difference is that proactive migration is synchronized by its own very nature (the thread invokes `migrate()` when it means to migrate), while for reactive migration the time when the thread has to be interrupted could be unpredictable (the requester thread notifies the migration request to the destination thread, but the operation is not supposed to be instantaneous). Therefore, in the latter case, the critical design-level decision is about the degree of asynchronism to provide. In a few words, the question is: should the designated thread be interruptible anywhere in its code or just in specific safe migration points? We chose to provide a coarse-grained migration in the reactive case. Our choice has a twofold motivation: (i) designing the migration facility is simpler; (ii) decreasing migration granularity reduces inconsistency risks. Although these motivations can be considered general rules-of-thumb, they are indeed related to the VM we adopted. In fact, the scheduling of the threads in JikesRVM has been defined as *quasi-preemptive* [5], since it is driven by JikesRVM compilers. As mentioned, JikesRVM threads are objects that can be executed and scheduled by several kernel-level threads, called *virtual processors*, each one running on a physical processor. What happens is that the compiler introduces, within each compiled method body, special code (*yieldpoints*) that causes the thread to request its virtual processor if it can continue the execution or not. If the virtual processor grants the execution, the virtual thread continues until a new yieldpoint is reached, otherwise it suspends itself so that the virtual processor can execute another virtual thread. In particular, when the thread reaches a certain yieldpoint (e.g. because its time slice is expired), it prepares itself to dismiss the scheduler and let a context switch occur. If we allow a reactive migration with a too fine granularity (i. e. potentially at any yieldpoint in threads life), inconsistency problems will almost surely occur. The thread can potentially lose control in any methods, from its own user-implemented methods to internal Java library methods (e.g. `System.out.println()`, `Object.wait()` and so forth). It may occur that a critical I/O operation is being carried out and a blind thread migration would result in possible inconsistency errors. We are currently tackling the reactive migration issues thanks to JikesRVM yieldpoints and the JIT compiler. In order to make mobile threads interruptible with the mentioned coarse granularity, we introduced the *migration point* concept: migration points are always a subset of yieldpoints, because they are reached only if a yieldpoint is taken. The only difference is that migration points are inserted only:

1. in the methods of the `MobileThread` class (*by default*);
2. in all user-defined class implementing the special `Dispatchable` interface (*class-level granularity*);
3. in those user-methods throwing `DispatchablePragmaException` (*method-level granularity*).

The introduction of a migration point forces the thread to check also for a possibly pending migration request. If the mobile thread takes the migration point, it invokes a special abstract handler method (i. e. the `onMigrationPoint()` of Figure 3.2) of the `MobileThread` class and this method is responsible for carrying out user-specific migration, as we exemplified in Figure 3.3 and Figure 3.4. This approach has several advantages: firstly, it rids us of the problem of unpredictable interruptions in internal Java library methods (not interested by migration points at all); then, it also gives the programmer more control over the migration, by letting her select those safely interruptible methods; last but not least, it leaves the stack of the suspended thread in a well-defined state, making the state capturing phase simpler. We achieved the insertion of migration points, simply substituting at runtime the method of the JIT compiler object, responsible for inserting yield points, with our migration points insertion method (the source code of the VM is left untouched and one can use every OSR-enabled version of the JikesRVM). We must point out that JikesRVMs compiler does not allow unauthorized users code to access and patch internal runtime structures. Users code, compiled with a standard JDK implementation, will not have any visibility of such low-level JikesRVM-specific details.

3.2. The resource management layer. The set of all referenced objects of a thread has been previously defined as its *data space* [13] and, at any point during the execution, is composed of all the objects that can be reached by the thread through the call stack or its fields. As concerns the stack, the space that the thread is supposed to bring with itself comprises all the objects pointed by the parameters and local variables of methods, together with those objects pushed on the operand stack of each frame in the stack. In the previous section, it was explained how the problem of collecting object references in stack frames has been easily tackled by means of the JikesRVM OSR extractor. The next step, as concerns the data space, will be dealing with objects relocation and reference rebinding. Although such issue pertains more to the application than to the thread migration middleware, we claim that its importance demands some kind of tool

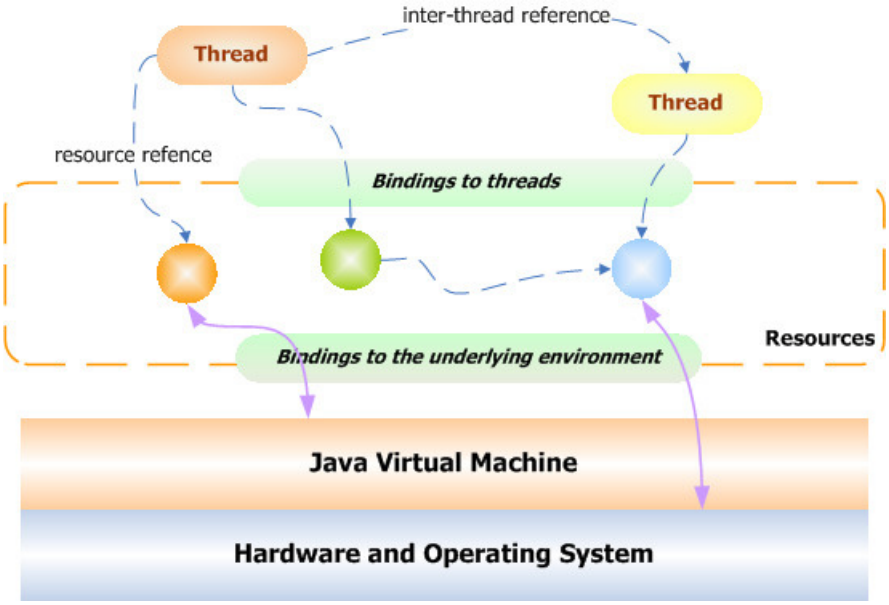

FIG. 3.7. *A conceptual view of resources and threads*

or support from a framework layer, in order to present a coherent set of mobile computing abstractions. In this section, we briefly sketch our vision of the problem and propose some ideas that are nonetheless part of our future research work. Our conceptual view of resources is depicted in Figure 3.7: Java threads can have references to either active (i. e. other threads) or passive resources (i. e. regular Java objects in the heap). The bindings to needed resources must be properly rearranged to maintain accessibility and consistency when the computation migrates to new locations. This poses two kinds of problems:

1. *handling the bindings of resources to their underlying execution environment*. This is not a problem if we consider only resources, like pure Java objects, which are not bound to any OS physical entity; on the contrary, resources, such as files, sockets or database objects, cannot be barely serialized without carefully managing their binding to the underlying environment.
2. *handling the binding of resources to migratory threads*. Fuggetta et al. [13] identified three typologies for this bindings (by identifier, by value or by type) and proposed some relocation strategies for each of them (by move, by copy, network reference, rebinding).

As for the first point, it must be pointed out that moving some resources (e.g. a centralized database) may not be technically (e.g. the bandwidth is not enough for its size) or semantically (e.g. it is already in use for queries by other threads) possible. We think that such issues should be coped with by explicitly introducing the *Resource* concept in our programming model and letting the programmer specify the right policy for her resources. Introducing the `Resource` entity as an interface, the programmer will be asked to make its resource objects implement such interface, together with a set of useful methods for:

- extracting the resource from its environment in a portable/serializable format (if the resource is fixed an exception will be raised and caught by the framework);
- attaching the resource to the destination environment;
- performing a correct cleanup of the resource, if it is detached from the source JVM (see the proposal by Park and Rice [26]).

A simple example of a resource can be a `java.io.File` object. A mere serialization of such an object will not produce the actual movement of the underlying file system object. To accomplish this task, the programmer has to introduce its `MovableFile` object, inheriting from `File` and implementing the `Resource` interface, with some of the methods listed above: in particular, calling the extraction method will likely return a `byte[]` filled in with the file content; calling the attach method will recreate that file in the file system at destination, with its previous content. Moreover, this part of the resource management layer is responsible for properly handling other non negligible issues, pertaining to *dependencies among resources*, *protection* and *concurrency* (e.g. the resource is shared among threads, it is synchronized with a lock and so forth), *inter-thread references*. Focusing on the second point above, the problem of the *bindings between resources*

```
public class MyAgent extends Aglet{
    protected boolean migrated = false; //indicates if the agent has moved yet
    public void run(){
        if( ! migrated ){
            // things to do before the migration
            migrated = true;
            try{ dispatch(new URL(atp://nexthost.unimore.it);}
            catch(Exception e){ migrated = false; }
        }
        else{ // things to do on the destination host }
    }
}
```

FIG. 4.1. *An example of* `Aglet` *with a single migration*

and migratory threads should be addressed [34]. The choice of the right re-binding strategy depends on several factors, from runtime conditions and access-device properties to management requirements and user properties. For instance, a fixed server with no strict constraints on network bandwidth or memory could copy or move the needed resources and work on them locally, whereas a wireless-enabled laptop might want to access that resource remotely without moving it. However, the programming language adopted usually determines the binding strategy (apart from heavily restricted cases, like in Java RMI). Moreover, the strategy is typically embedded within the mobile application code, thus limiting *binding-management flexibility*. We envision that the resource management layer ought to give the programmers the means to specify which reference management policy [24] to use, on a per-instance basis. Furthermore, since the semantics of a given strategy is the same whatever the resource is (e.g. network reference, rebinding, etc.), strategies should be implemented as *basic blocks* that can be reused and programmatically attached to any object, thus achieving a clear and beneficial *separation of concerns* [18] (i. e. between application/functional and non functional/rebinding concerns). Providing an effective and clear support for such abstractions on top of the JikesRVM is part of our future work on this topic.

4. Strong Mobility in Aglets. This section offers an example of a testbed application that we have implemented on top of the *Mobile JikesRVM* framework. It consists of the well-known IBM Aglets MAP, which provides only weak mobility support to mobile agent applications. Simply modifying some parts of its Java source code, we succeeded in implementing a porting of this MAP endowed with strong agent mobility.

4.1. Overview of the Aglets Workbench. The Aglets Workbench [2] is a project originally developed by the IBM Tokyo Research Laboratory with the aim of producing a platform for the development of mobile agent based applications by means of a 100% Java library. The Aglets Workbench provides developer with applet-like APIs [19], thus creating a mobile agent (called *aglet*) is a quite straightforward task. It suffices to inherit from the base class Aglet and to override some methods transparently invoked by the platform during the agent life. Weak mobility is provided through the Java serialization mechanism, and a specific agent transfer protocol (ATP) has been built on top of such mechanism [20]. Each `Aglet` can exploit the special method `dispatch()` to move to another host.

As many other Java MAPs, Aglets exploits weak mobility, that means, from a programming point of view, that each time an agent is resumed at a destination machine, its execution restarts from a defined entry point, that is the `run()` method call. Due to this, dealing with migrations is not always trivial, and developers have to adopt different techniques to handle the fact an agent will execute several times the same code but on different machines. Even if the Aglets library provides a set of classes that helps dealing with migrations, the code will appear like the one shown in the simple example of Figure 4.1. There, in case of a single migration, the `migrated` flag is used to select a code branch for the execution either on the source or on the destination machine. The code of Figure 4.1 is just a simple example, but more complex agents follow the same programming style. In all such cases the point is that with weak mobility it is as the code routinely performs rollbacks. In fact, looking at the code in Figure 4.1, it is clear how, after a successful `dispatch(..)` method call that causes the agent migration, the code does not continue its execution in the `run()` method from that point. Instead, the code restarts from the beginning of the run method (on the destination machine, of course), and thus there is a code rollback. The fact that an agent restarts its execution always from a defined entry point, could produce awkward solutions, forcing the developer to use flags and other indicators to take care of the host the agent is currently running on.

```
public class MyAgent extends Aglet{
    public void run(){
        // things to do before the migration
        try{
            migrate(new URL(atp://nexthost.unimore.it);
        }catch(Exception e){  }
        // things to do after migration
    }
}
```

FIG. 4.2. *An example of* `Aglet` *code using Mobile JikesRVM*

4.2. Implementing Strong Mobility. Our major aim here has been to realize the idea of an Aglet that is to be executed by a strong migratory thread and this required some modifications to the underlying Aglets execution model. In particular, instead of using one of the pre-created threads (i. e., thread pool) to execute the methods of the aglets, JikesRVM makes feasible to have a single independent thread for each aglet. As already mentioned, this is possible because of the lightweight implementation of Java threads in that JVM, being targeted to server architectures, where scalability and performance are key requirements. Furthermore, having a separate thread for each aglet ensures a high level of *isolation* between agents: consider, for example, the case where an agent wants to sleep for some time, without being deactivated (i. e. serialized on the hard disk). Using the classical `sleep()` method on the `java.lang.Thread` object will produce strange effects on the current Aglets implementation platform (such as locking the message passing mechanism). These shortcomings are due to the thread sharing among multiple agents through the pool of threads. Instead, potentially dangerous actions by malicious (or bugged) aglets do not affect the stability of our platform, allowing possibly a clean removal of the dangerous agent without the need of a MAP reboot. In our prototypal implementation, there is only one thread responsible for handling the messages posted to the aglet and this thread will invoke the appropriate handler function to perform the necessary actions in response to the delivered message. In the official Aglets framework, the thread running into the handler function cannot be interrupted asynchronously by a migration request, notified by another thread by means of the *dispatch* message. In our prototype, the dispatch message has the effect of interrupting/preempting the execution of the function (in particular, the handler of the run message, i. e. the `run()` method) and migrating the aglet to the designated host. The `OnDispatching()` handler method is executed to allow preparatory actions to be done, but the current execution stack is preserved, together with local variables, stack operands and method parameters. There is no more need for saving intermediate results into serializable fields or structuring the code with entry points from which the agent execution is restarted each time it arrives at a new host. Referring to the code example of Figure 4.1, the adoption of strong thread mobility overtakes the mentioned drawbacks, since the code restarts at the destination machine from the same point it has stopped at the source one. Thus the code shown in Figure 4.1 becomes the one of Figure 4.2. As readers can see, the code is simpler (no flags and branches are required) and shorter than the previous one.

This kind of message-driven strong mobility is achieved serializing the aglet object and its fields but also appending the sequence of stack frames (as we have explained in Section 3.1) representing the state of the execution at the time of the suspension. Reactive migration has been achieved exploiting the migration point concept provided by the mobility layer (see subsection 3.1) underneath. On the other hand, the de-serialization process involves

1. reading the aglet object from the network stream into the memory;
2. creating a new thread for this aglet or acquiring an existing one, if available;
3. notifying this thread of the arrival event and suspending its execution;
4. injecting on the fly all the migrated frames into its stack;
5. resuming the execution of the thread/aglet transparently.

The migrated aglet will be, by default, destroyed in the source JVM and its associated thread added back to the thread pool, if available. Nevertheless, the *dispose* message can be explicitly intercepted by the programmer so that the aglet can continue executing, thus realizing a form of agent cloning. In summary, the new Aglets implementation tries to overcome the drawbacks of weak agent mobility, using the thread migration facilities of the underlying *mobility layer* and hopefully the resource abstractions of the *resource management layer*.

5. Performance and discussions. At the current stage of our research, the mobility layer of our framework has been successfully tested, focusing mainly on the state capturing and restoring of the threads executing the aglets. First of all, we made some first performance tests (running some simple agent applications) to discover possible bottlenecks and evaluate

the cost of each migration phase. The times measured are expressed in seconds and are average values computed across multiple runs, on a Pentium IV 3.4Ghz with 1GB RAM on JikesRVM release 2.4.5. Thanks to a Fibonacci recursive algorithm we were able to test thread serialization with increasing stack sizes (5, 15 and 25 frames) and found a very graceful time degradation. These times are conceptually divided into two tables, where Table 5.1 refers to the thread serialization process, while Table 5.2 refers to the symmetrical de-serialization process at the arrival host.

TABLE 5.1
Evaluated times for thread serialization (sec.)

	5 frames	**10 frames**	**25 frames**
OSR capturing	1.78E-5	1.89E-5	1.96E-5
State building	3.44E-5	3.75E-5	3.43E-5
Pure serialization	2.49E-3	7.32E-3	1.50E-2
Overall times	2.54E-3	7.38E-3	1.51E-2

Considering how these times are partitioned among the different phases of the thread serialization, we can see that the bulk of the time is wasted in the pure Java serialization of the captured state, while the frame extraction mechanism (i. e. the core of our entire facility, comprising OSR extraction and state building) has very short times instead. The same bottleneck due the Java serialization may be observed in the de-serialization of the thread. In the latter case, however, we have an additional overhead in the stack installation phase, since the system has often to create a new thread and compile (if not yet compiled) the methods for the injected frames. These performance bottlenecks can be further minimized, perhaps using externalization to speed up the serialization of the thread state [35]. Moreover, we had to modify the size of JikesRVM LOS (Large Object Space) to allow the instantiation of a bigger number of thread objects into the runtime image. Nevertheless, the developed prototype has some limitations that will be dealt with in the future: the first one is about the kind of supported compilers. JikesRVM basically provides two compilers, designed to achieve different levels of code optimization: a *baseline* and an *optimizing compiler* [5] (a third *quick compiler* is, at the time of writing, still in a prototypal phase). Our prototype can actually migrate baseline compiled methods JikesRVM, mainly because of an OSR mechanism limitation: it can actually capture method scope descriptors for those methods compiled by optimized compilers, but this requires maintaining additional structures to cope with parameters allocated into registers, inlined methods and other challenging optimization techniques [11]. Currently, JikesRVM designers allows OSR to occur only at yield points (i. e. thread pre-emption points in the code) and this implies that not all the optimized frames in the stack have their maps updated. Nonetheless, we are aware of a project by Krintz et al. [31] trying to present a more general-purpose version of OSR that is more amenable to optimizations than the current one. The improvement descending from this work will be exploited to perform a more complete thread state capturing, even in presence of code optimizations.

TABLE 5.2
Evaluated times for thread rebuilding (sec.)

	5 frames	**10 frames**	**25 frames**
Pure de-serialization	4.46E-3	5.33E-3	7.06E-3
State rebuilding	5.45E-4	5.27E-4	5.06E-4
Stack installation	1.53E-3	1.60E-3	1.71E-3
Overall times	6.54E-3	7.46E-3	9.28E-3

6. Conclusions and Future Work. This paper has introduced our *Mobile JikesRVM* framework to support Java thread strong mobility based on the IBM JikesRVM virtual machine, and has shown how its migration services can be effectively exploited to build mobile computing applications, such as the presented Aglets Mobile Agent Platform. Thanks to the support to thread serialization, agents will be simpler in terms of code, and, at the same time, the code will be easier to read. Our approach represents an extension of JikesRVM, which is pluggable at runtime in any OSR-enabled version of that JVM. It exploits, in fact, some interesting JikesRVM built-in facilities to avoid many of the drawbacks of past solutions. In particular, OSR facility allowed us to capture the execution state (i. e. method frames) in a very portable (i. e. bytecode-level) format. Thanks to the scheduling policy of the JikesRVM, which enables the support of thousands of Java threads, our approach keeps the thread management more lightweight, experimenting the possibility of having one thread for each agent, which is not possible in the current implementation of Aglets. Our JikesRVM-based migration library enriches the Aglets framework with strong mobility benefits. Additional features will be, of course, implemented

to extend our thread mobility framework in the future. Future work includes a comparison with other proposed thread migration systems, to improve our performance evaluation understanding and identify possible undetected bottlenecks. Finally we are currently working to port the implemented code also on PPC architectures (JikesRVM is available also for this architecture), allowing the migration of a thread among heterogeneous platforms as well.

Acknowledgments. Work supported by the European Community within the IST FET project "CASCADAS".

REFERENCES

[1] A. ACHARYA, M. RANGANATHAN, J. SALTZ, *Sumatra: A Language for Resource-aware Mobile Programs*, in 2nd International Workshop on Mobile Object Systems (MOS'96), Linz, Austria, 1996.
[2] THE AGLETS MOBILE AGENT PLATFORM WEBSITE, `http://aglets.sourceforge.net`
[3] B.ALPERN, C.R. ATTANASIO, D. GROVE AND OTHERS, *The Jalapeno virtual machine*, IBM System Journal, Vol. 39, N1, 2000.
[4] B. ALPERN, S. AUGART, S.M. BLACKBURN, M. BUTRICO, A. COCCHI, P. CHENG, J. DOLBY, S. FINK, D. GROVE, M. HIND AND OTHERS, *The Jikes Research Virtual Machine project: Building an open-source research community*, IBM Systems Journal, Vol. 44, No. 2, 2005.
[5] B. ALPERN, D. ATTANASIO, J. J. BARTON, A. COCCHI, S. F. HUMMEL, D. LIEBER, M. MERGEN, T. NGO, J. SHEPHERD, S. SMITH, *Implementing Jalapeño in Java*, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '99), Denver, Colorado, November 1, 1999.
[6] M. ARNOLD, S. FINK, D. GROVE, M. HIND, P. F. SWEENEY, *Adaptive Optimization in the Jalapeño JVM*, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2000), Minneapolis, Minnesota, October 15-19, 2000.
[7] S. BOUCHENAK, D. HAGIMONT, S. KRAKOWIAK, N. DE PALMA AND F. BOYER, *Experiences Implementing Efficient Java Thread Serialization, Mobility and Persistence*, I.N.R.I.A., Research report n° 4662, December 2002.
[8] S. BOUCHENAK, D. HAGIMOT, *Pickling Threads State in the Java System*, Technology of Object-Oriented Languages and Systems Europe (TOOLS Europe'2000) Mont-Saint-Michel/Saint-Malo, France, Jun. 2000.
[9] G. BURKE, J.CHOI, S. FINK, D.GROVE, M. HIND, V. SARKAR, M.J. SERRANO, V.C. SREEDHAR, H. SRINIVASAN, The Jalapeno Dynamic Optimizing Compiler for Java, ACM Java Grande Conference, June 1999.
[10] C. CHAMBERS, *The Design and Implementation of the Self Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*, PhD thesis, Stanford University, Mar. 1992. Published as technical report STAN-CS-92-1420.
[11] S. FINK, AND F. QIAN, *Design, Implementation and Evaluation of Adaptive Recompilation with On-Stack Replacement*, International Symposium on Code Generation and Optimization San Francisco, California, March 2003.
[12] S. FNFROCKEN, *Transparent migration of Java-based mobile agents: Capturing and reestablishing the state of Java programs*, In K. Rothermel and F. Hohl (Ed.), Mobile Agents: Proceedings of the Second International Workshop (MA 98), Stuttgart, Germany. (pp. 26-37). Berlin, Germany: Springer-Verlag.
[13] A. FUGGETTA, G. P. PICCO, G. VIGNA, *Understanding Code Mobility*, IEEE Transactions on Software Engineering, Vol 24, 1998.
[14] T. ILLMANN, T. KRUEGER, F. KARGL, M. WEBER, *Transparent Migration of Mobile Agents Using the Java Platform Debugger Architecture*, Proceedings of the 5th International Conference on Mobile Agents, December 02 - 04, 2001, Atlanta, Georgia, USA.
[15] F. BELLIFEMINE, G. CAIRE, A. POGGI, G. RIMASSA, *JADE—A White Paper*, EXP in Search of Innovation, TILAB, vol. 3, 2003.
[16] N. R. JENNINGS, *An agent-based approach for building complex software systems*, Communications of the ACM, Vol. 44, No. 4, pp. 35-41 (2001).
[17] THE JIKESRVM PROJECT SITE, `http://jikesrvm.sourceforge.net`
[18] G. KICZALES, J. IRWIN, J. LAMPING, J. LOINGTIER, C. LOPES, C. MAEDA, AND A. MENDHEKAR, *Aspect Oriented Programming*, in Special Issues in Object-Oriented Programming, Max Muehlhaeuser (general editor) et al., 1996.
[19] D. B. LANGE, M. OSHIMA, G. KARJOTH, K. KOSAKA, *Aglets: Programming Mobile Agents in Java*, in the Proceedings of the International Conference on Worldwide Computing and Its Applications (WWCA), 1997.
[20] D. B. LANGE, Y. ARIDOR, *Agent Transfer Protocol (ATP)*, IBM=TRL, draft number 4, 19 March 1997.
[21] T. LINDHOLM, F. YELLIN, *The Java Virtual Machine Specification, second edition*, SUN Microsystem.
[22] M. LUCK, P. MCBURNEY, C. PREIST, *Agent Technology: Enabling Next Generation Computing A Roadmap for Agent Based Computing*, AgentLink, `http://www.agentlink.org/roadmap`
[23] Mobile JikesRVM is available at the projects website, `http://www.agentgroup.unimore.it/didattica/curriculum/raffaele`
[24] R. MONTANARI, E. LUPU AND C. STEFANELLI, *Policy-Based Dynamic Reconfiguration of Mobile-Code Applications*, in IEEE Computer, July 2004
[25] SCOTT OAKS AND HENRY WONG, *Java Threads, 2nd edition*, Oreilly, 1999
[26] D. PARKA AND S. RICE, *A Framework for Unified Resource Management in Java*, in the Proceedings of the International Conference on Principles and Practices of Programming In Java (PPPJ 2006), Mannheim, Germany, August 30 – September 1, 2006
[27] T. SAKAMOTO, T. SEKIGUCHI, A. YONEZAWA, *A bytecode transformation for Portable Thread Migration in Java*, 4th International Symposium on Mobile Agents 2000 (MA'2000), Zurich, Sep. 2000.
[28] T. SEKIGUCHI, A. YONEZAWA, H. MASUHARA, *A Simple Extension of Java Language for Controllable Transparent Migration and its Portable Implementation*, 3rd International Conference on Coordination Models and Languages, Amsterdam, The Netherlands, Apr. 1999.
[29] *The Java Object Serialization Specification*, Sun Microsystems, 1997.
[30] D. SISLAK, M. ROLLO, M. PECHOUCEK, *A-globe: Agent Platform with Inaccessibility and Mobility Support*, in Cooperative Information Agents VIII , n. 3191, Springer-Verlag Heidelberg, 2004.
[31] S. SOMAN, C. KRINTZ, Efficient, General-Purpose, On-Stack Replacement for Aggressive Program Specialization, University of California, Santa Barbara Technical Report 2004-24.
[32] T. SUEZAWA, *Persistent Execution State of a Java Virtual Machine*, ACM Java Grande 2000 Conference, San Francisco, CA, USA, Jun. 2000.
[33] N. SURI ET AL., *An Overview of the NOMADS Mobile Agent System*, Workshop On Mobile Object Systems in association with the 14th European Conference on Object-Oriented Programming (ECOOP 2000), Cannes, France.

[34] É. TANTER, M. VERNAILLEN AND J. PIQUER, *Towards Transparent Adaptation of Migration Policies*, in the 8th ECOOP Workshop on Mobile Object Systems (EWMOS 2002), in conjunction with the 16th European Conference on Object-Oriented Programming (ECOOP 2002), June 2001, Malaga, Spain.

[35] SUN MICROSYSTEMS, *Improving Serialization Performance with Externalizable*, `http://java.sun.com/developer/TechTips/txtarchive/2000/Apr00_StuH.txt`

[36] E. TRUYEN, B. ROBBEN, B. VANHAUTE, T. CONINX, W. JOOSEN, P. VERBAETEN, *Portable support for Transparent Thread Migration in Java*, in 4th International Symposium on Mobile Agents 2000 (MA'2000), Zurich, Switzerland, Sep. 2000.

[37] X. WANG, *Translation from Strong Mobility to Weak Mobility for Java*, Master's thesis, The Ohio State University, 2001.

[38] W. ZHU, C. WANG, F. C. M. LAU, *JESSICA2: A Distributed Java Virtual Machine with Transparent Thread Migration Support*, in IEEE Fourth International Conference on Cluster Computing, Chicago, USA, September 2002.

Edited by: Henry Hexmoor, Marcin Paprzycki, Niranjan Suri
Received: October 1, 2006
Accepted: December 14, 2006