# Improving Aglets with Strong Agent Mobility through the IBM JikesRVM

Giacomo Cabri, Luca Ferrari, Letizia Leonardi, Raffaele Quitadamo, *Member, IEEE*

***Abstract* — Agents are problem-solving entities that, thanks to characteristics such as autonomy, reactivity, proactivity and sociality, together with mobility, can be used to develop complex and distributed systems. In particular, 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 mobility of threads along with their execution state; thus developers cannot develop agents as real mobile entities. This paper reports our approach for Java thread strong migration, based on the IBM Jikes Research Virtual Machine, presenting our results and proposing an enrichment of the Aglets mobile agent platform in order to exploit strong agent mobility.**

***Index Terms* — Mobile Agents, JikesRVM, Aglets, strong mobility.**

## I. Introduction

AGENTS are autonomous, proactive, active and social entities able to perform their task without requiring a continue user interaction [23]; thanks to the above features, the agent-oriented paradigm is emerging as a feasible approach to the development of today's complex software systems [18]. Moreover agents can be *mobile*, which means they can migrate among different sites/hosts during their execution.

Mobility is an interesting feature for agents, since they are able to move among networks to find out data and information, to perform load balancing activities, and so on. The exploiting of mobile agents can simplify different issues in the design and implementation of applications and enables developers to quickly build distributed and parallel systems.

Mobile agent execution is hosted by a special software layer, called Mobile Agent Platform (MAP) that enables also the agent migration and allows security checks over agents. In order to enable agents to migrate among different platforms and, thus, architectures, portable technologies and languages must be adopted to develop MAPs and the corresponding agents. Thanks to its portability and network facilities, Java is today the most exploited language to develop mobile agents, and in fact several Java-based MAPs exist [17, 3, 28].

With regard to mobility, we distinguish [34] *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 [14]. Some distributed operating systems [32] go even further and make it possible for entire processes to be migrated with also their kernel mediated state, comprising I/O descriptor, alarm timers and others. This extremely transparent migration is known as *full migration* [16]. Unfortunately, current standard Java Virtual Machines (JVMs) do not support thread migration natively, and thus a Mobile Agent Platform running on top of them cannot provide strong mobility of agents. Moreover, the Java language itself [15] does not support constructs or mechanisms for thread serialization and migration: that is, there is no way, using the standard Java language and JVMs, to enable agents to exploit strong mobility. Even if this does not represent a problem for many applications based on mobile agents, such as those of automated booking [13], it does not allow using the agent paradigm to develop more complex and distributed systems, such as those for load balancing [25].

To overcome this limitation, this paper proposes an approach to support strong thread migration for Java MAPs, based on the IBM JikesRVM [5], which is a Java virtual machine with very interesting features. Our approach significantly differs from other proposals, since it requires neither any modification to the JVM, nor it exploits any pre-processing, but it simply defines an appropriate Java library.

The paper is organized as follows: section II presents the state of the art, explaining the existing approaches and pointing out their limitations. Section III introduces the features of JikesRVM and explains how they can be exploited to build a library for supporting strong mobility. Section IV presents the Aglets platform and shows how strong mobility can be designed and implemented in such a platform by using our mobility library. Finally, Section V reports our first performance measures and Section VI concludes the paper.

## II. State of the Art

The approach presented in this paper aims at implementing strong thread migration in Java, which is not a new idea. Several approaches have been proposed so far and they can be, typically, split into two categories, depending on the fact that they require to modify the JVM (*JVM-level approach*) to support an advanced thread management or exploit some kind of bytecode instrumentation (*Application-level approach*) to track the state of each thread.

Approaches that modify the JVM (such as Sumatra [2], ITS [10], Merpati [29], Jessica2 [33] and JavaThread [9]) often introduce the problem of the management of the virtual machine itself. First of all, it is worth noting that they are applied to JVM that are at least one (or even more) version older than the SUN production one. Second, the adoption of a modified JVM can introduce problems of trust and security bugs. Third, virtual machines are usually written in a language different from Java (e.g., C++), thus suffering from portability problems.

Instead, approaches that exploit bytecode manipulation (e.g. JavaGoX [26] or Brakes [30]) or Java source code manipulation [16], even if based on a pure Java technique (and thus really portable), do not provide a full thread management and suffer from problems related to performances. In fact, the idea of these approaches is to transparently place a few control instructions, similar to recovery-points, which 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. Un23ily, 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. Moreover, the use of bytecode manipulation produces low performances, thus these techniques are not appropriate for those applications where speed represents a strong requirement.

In general, all existing strongly mobile systems have to deal with the problem of locating object references when they want to migrate a thread with all its set of stack-referenced objects: they force the use of some "type inference" mechanism [9, 33], either at execution or at compilation time, thus introducing a significant performance overhead in threads execution. In order to tackle the drawbacks of strong mobility while saving its clarity and power, some interesting algorithms have been proposed [8] that translate transparently the apparently "strongly mobile code" into a "weakly mobile" form, with the above mentioned benefits of weak mobility.

Starting from the above considerations, we have decided to design and implement a thread migration system able to overcome all the problems of the above-explained approaches. In particular, it is written entirely in Java, thus portable as much as possible and it grants high performances even without modifying the JVM. 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, changing or patching the virtual machine, in specific, the IBM JikesRVM. Programmers and users do not have to download a modified, untrustworthy, version of JikesRVM, but can import the implemented mobility package into their code and execute it on their own copy of JikesRVM. Therefore, our JikesRVM-based approach can be classified as a midway approach between the above-mentioned *JVM-level* and *Application-level* approaches.

## III. From Weak to Strong mobility: a JikesRVM-based Approach

A Mobile Agent Platform realizes an environment for the execution of agents, featured with a bent for mobility. The support for mobility is often one of the first design choices when implementing such a platform, since it has a great impact on the remainder of the design.

### A. *Weak vs. Strong Mobility*

Current execution environments for programming languages (e.g., the Java Virtual Machine [22] and the Common Language Runtime, embedded into Microsoft .NET Framework [1]) are usually not suited or not capable of providing the required level of mobility to the execution units (i.e. the threads) that they host. They all lack an explicit support for the mobility of their execution units and, in order to overcome this lack, MAP designers must choice between two directions [14]:

- Adopting one of the techniques explained in the previous section so that threads can suspend their execution locally and resume elsewhere transparently (*strong mobility*).
- Introducing a further abstraction level above the thread concept: the *weak mobile* agent, which is explicitly thought as a serializable representation of an execution unit.

From the complexity point of view, weak mobility is quite simple to implement using well-established techniques like network class loading or object serialization [27]. However, weak mobility 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 is required in order to manually save the execution state. The transparency of the migration offered by strong mobility systems has instead a twofold advantage: it reduces the migration programming effort to the invocation of a single operation (e.g. a `migrate()` method), and requires a size of the migrated code smaller because it does not add artificial code.

Despite these advantages, most of the 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. It is a common idea that strong mobility should be convenient only in load balancing contexts or when thread persistence is needed to build fault-tolerant applications [9].

Recently, an innovative project is drawing researcher's attention to the benefits that a virtual machine written in the Java language can offer. The main features of this open-source project, called *JikesRVM* [5], are outlined in the following subsection: 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.

## B. *The Jikes Research Virtual Machine*

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" [4]. 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 architecture's peculiarities. The all-in-Java philosophy of this VM makes very easy for researchers to manipulate or extend its functionalities. Further, JikesRVM source code can be built, with a prior custom compilation, both on IA32 and on PPC platforms [19], but the bulk of the runtime is made up of Java objects portable across different architectures.

The first step toward the development of our MAP based on JikesRVM has been the implementation of the strong Java thread mobility. Threads embody concurrent flows of execution within an instance of the JVM and are represented by the `java.lang.Thread` object [22], used by the Java programmers disregarding any knowledge of their underlying physical implementation. In JikesRVM, threads are full-fledged Java objects and are designed explicitly to be as lightweight as possible [4]. Many server applications need to create new threads for each incoming request and 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* [24]: Java threads are hosted by the same operating-system thread, implemented by a so-called *virtual processor*, through an object of class `VM_Processor` [6]. Each virtual processor manages the scheduling of its virtual threads (i.e., Java threads), represented by objects of the class `VM_Thread`. The scheduling of virtual threads was defined *quasi-preemptive*, since it is driven by the JikesRVM compiler. What happens is that the compiler introduces, within each compiled method body, special code (*yield points*) 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 yield point is reached, otherwise it suspends itself so that the virtual processor can execute another virtual thread.

The choice of using virtual processors not only allows JikesRVM to reduce the number of threads the operating system is in charge of, but also allows it to perform an efficient and well-controlled thread-switch. As a consequence, this allows elegantly addressing the problem of precisely locating object references when a garbage collection occurs. JikesRVM uses *type-accurate* collectors [31] that build the so-called *reference maps* automatically at compile-time, unlike conservative collectors, which attempt somehow to infer whether a stack word is a reference or not. These reference maps are periodical snapshots of the situation of references in each method frame.

The tracks of object references used to speed up the JikesRVM *type-accurate* garbage collectors can be exploited by MAP designers to collect stack-referenced objects for strong thread migration. This eliminates the need for "type inference" mechanisms required by existing strongly mobile systems.

In general, many JVMs do not permit the programmer to access the execution state (i.e. the stack and the context registers), in order to enforce the security model of the Java language. As a consequence, they do not allow strong mobility. Instead, JikesRVM provides, once again, a built-in facility to extract correctly the execution state of a suspended thread. This facility is an efficient implementation of the *On-Stack Replacement* (OSR) technique, originally developed for the Self language [35]. It enables a method to be automatically replaced by the system while it is executing. In particular, the system replaces the runtime stack activation frame of the method with that of the new version, and continues execution at the same point within the new version. JikesRVM exploits the OSR mechanism [12] in order to enable the dynamic optimization of methods. The Adaptive Optimization System (AOS) [7] samples the execution of programs to identify frequently executed (i.e. "hot") methods and, when their optimization is predicted to be beneficial, the system compiles the method with JikesRVM *optimizing compiler* [11]. The old less-optimal frame is discarded and a new optimized frame is placed, initialized with the current state of the method (i.e. the value of the local variables and stack operands, together with the current bytecode index).

This mechanism has been successfully exploited to quickly get a complete and portable representation of the serialized call stack. The structure of the *OSR scope descriptor* [12] inspired the idea of the `MobileFrame`: an object representing the current state of the method execution in a format that should be understandable by any JVM since it refers purely to bytecode-level entities (bytecode program counter, locals and stack operands). Our mechanism applies the capturing to all user frames in the stack of the serialized thread and, on the one hand, offers the advantage of the portability of the frames and, on the other hand, exploits a fully integrated component of the JVM. The latter aspect is crucial from both the reliability and the performance point of view, since no unsafe manipulations are carried out on the JVM code to force the externalization of the execution state of the thread.

The presented features of JikesRVM allow the addition of strong thread migration, without modifying the virtual machine, but simply extending it. The entire system is available as a library comprised in a Java package that can be imported as usual into the application code. This means that the implemented JikesRVM extension does not affect the performance of other applications, since no permanent modifications have been made to the VM itself.

## IV. Strong Mobility in Aglets

### A. *Overview Of The Aglets Workbench*

The Aglets Workbench [3] 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 [20], 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 [21]. Each Aglet can exploit the special method `dispatch(..)` to move to another host; such method is the equivalent of the generic `migrate(..)` previously mentioned.

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 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 destination machine.

```
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
      // ….
    }
  }
}
```

Figure 1. An example of Aglet with a single migration.

The code of Figure 1 is just a simple example, but similar agents can be written for other agent platforms. The point here is that with weak mobility, which is the one provided by the Java language and the most existing MAPs, it is as the code routinely performs rollbacks. In fact, looking at the code in Figure 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.

### B. *Designing Strong Mobility*

In Section III.B we have presented the innovative features of JikesRVM that can be exploited to strongly migrate threads. Now we apply these features to the Aglets to realize the idea of an Aglet as a strong migrable thread. Instead of using one of the pre-created threads to execute 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. Further, 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 aforementioned 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.

Message handling or events are implemented using the quasi pre-emptive JikesRVM scheduler, described earlier. Yield points are used to let the running aglet/thread extract messages from its message queue and handle them. Thus, for example, the aglet can process a *dispatch* message even in the middle of its execution (i.e. while the `run()` method is still in the stack) and strongly migrate to the destination site, where it will resume transparently restarting from the last execution point. The programmer gets rid of the burden of saving intermediate results into serializable fields and of structuring its code with entry points (such as methods) from which the agent execution is restarted each time it arrives at a new host, as mentioned above.

The conceptual model of our prototype was thought as intuitive and understandable as possible in this development stage: we took inspiration from the fantastic world of space travels through *black holes*. According to this model, a mobile agent (i.e. "the traveller") invokes the services offered by (i.e. "gets himself absorbed by") a black hole on one host (i.e. "planet") to move through the network (i.e. "the space") and arrives at the destination host (i.e. "another planet on a distant universe"), being extracted from the other side of the black hole.

## C. *Implementing Strong Mobility*

After having tested the mobility library building a simple prototypal framework whose classes manage the departure and arrival of the mobile threads, we now in our research are integrating the mobility support in the Aglets framework to have a full-fledged MAP endowed with strong mobility.

The implementation of the *black hole* model is based on JikesRVM and embedded into the Aglets runtime to make available the migration services to agents. In Figure 2, our system is described using the classical notation of *queuing networks*. The software components added by our approach are highlighted with boxes and it can be clearly seen how these parts are dynamically integrated into JikesRVM scheduler, when the programmer opens a black hole to enable migration services: no JVM manipulations are performed, therefore a non invasive extension is carried out.

Agents are classified into three main categories:

1. *Incoming agents*, coming from the outside world and requesting execution on the current host. They are read from a network socket and re-established in the local execution context, to be scheduled there.
2. *Outgoing agents*, which are leaving the scheduler queues to be transferred on another machine. They invoked a dispatch() method and got queued into the hole’s migration queue.
3. *Stationary agents*, not interested or affected by the migration facilities of our mechanism.

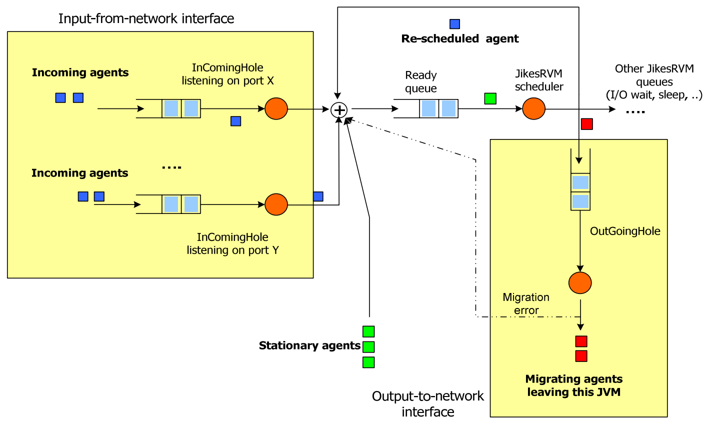
Input-from-network interface
Incoming agents
InComingHole listening on port X
Incoming agents
....
InComingHole listening on port Y
Re-scheduled agent
Ready queue
JikesRVM scheduler
Other JikesRVM queues (I/O wait, sleep, ..)
....
OutGoingHole
Migration error
Stationary agents
Migrating agents leaving this JVM
Output-to-network interface

Figure 2. The queuing network model of the mobility framework

The black hole provides two kinds of services: a sending service, for agents exiting the local JVM, and a receiving service, for incoming execution requests. The former service is implemented by a server thread created with `BlackHole` instantiation, started when the `BlackHole` gets opened. This thread, instance of the `OutGoingHole` class, tests a migration queue in an endless loop, until the application closes its parent `BlackHole`, and analyzes every extracted mobile agent: its execution state is retrieved using OSR built-in state capturing and the thread object, together with the chain of all the stack frames, are written into the socket established with another peer host. In more details, the JikesRVM thread/agent is suspended before the state capturing can occur and the stack is walked back from the last pushed frame to the first one (i.e. the `run()` method). At every step, the corresponding physical frame is analyzed invoking the OSR extraction service and the OSR descriptor is produced; but this intermediate form is not yet fully portable, mainly because it has been conceived only to refer to structures that are supposed to stay in the local memory: in particular, we are talking about the compiled methods in method area and the corresponding program counters (the so called *return address* of each frame) in the machine code body. So, the next essential stage performed by our mechanism is to retrieve a return address as much portable as possible: the bytecode index corresponding to the machine code index of the method. The mapping between the two indexes is, once again, granted by JikesRVM compilers and can be calculated in very little times. Local variable and stack operands are converted also into portable objects and stored into the `MobileFrame` for each method in the stack, as shown in Figure 3.

It must be pointed out that this representation of the serialized thread is a very general one, as it uses only bytecode level entities (e.g. bytecode indexes as program counters, local variables and stack operands and so on) and this grants high portability of the state. Dataspace objects are packed into the mobile frames (or in the thread object) and serialized as well. When all the necessary frames are successfully captured, the system can send them all to the destination host.

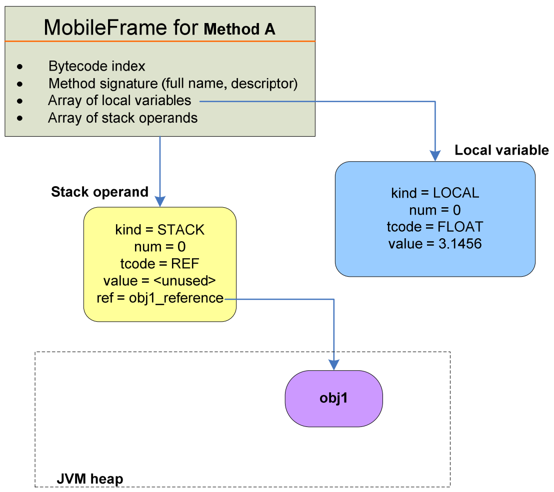

Figure 3. A MobileFrame object

To let external mobile agents enter a local environment, a group of `InComingHole` threads are created and started at `BlackHole` opening time. Each `InComingHole` opens a server socket bound to a specified TCP port and waits, in an endless loop, for incoming connection requests. When a connection is accepted and established, the `InComingHole` reads all the information about the state of the agent and, when finished reading, resumes the agent in the local instance of JikesRVM. At the arrival the aglet rebuilding is performed following some essential steps:

1. the aglet object is read from the network stream into the memory;
2. a new thread is created for this aglet or an existing one acquired from the pool, if available;
3. this agent is notified the arrival event and its execution is temporarily frozen;
4. the physical frames, produced by the `MobileFrame` objects, are injected on the fly into its stack;
5. the execution of the thread/aglet is transparently resumed.

The injection task is performed by a *frame installer* component, which adds each frame to a newly allocated stack, adjusting thread context registers and frame pointers. Frames are constructed in compliance with the baseline layout of the target platform. We have currently implemented a working frame installer for the IA32 architecture, but we are planning to complete the system with the PPC frame installer.

The migrated aglet will be, by default, destroyed in the source JVM and its associated thread added back to the thread pool, for a possible future reuse. 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".

Referring to the code example of Figure 1, the adoption of strong thread mobility overtakes the mentioned drawback, since the code restarts at the destination machine from the same point it stopped at the source one. Thus the code shown in Figure 1 becomes the one of Figure 4.

```
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
  }
}
```

Figure 4. An example of Aglet code using our approach.

As readers can see, the code is simpler (no flags and branches are required) and shorter than the previous one.

## V. Performance and Open Issues

At the current stage of our research, the thread serialization mechanism, integrated into the Aglets framework, has been successfully tested, focusing mainly on the state capturing and restoring of the threads executing the aglet.

First of all, we made some first performance tests 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 2.4.1. We tested the 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 1 refers to the thread serialization process, while Table 2 refers to the symmetrical de-serialization process at the arrival host.

| | **5 frames** | **15 frames** | **25 frames** |
|---|---|---|---|
| Frame extraction | 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 |

Table 1. Evaluated times for thread serialization (sec.)

| | **5 frames** | **15 frames** | **25 frames** |
|---|---|---|---|
| Pure deserialization | 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 |

Table 2. Evaluated times for thread rebuilding (sec.)

Considering how these times are partitioned among the different phases of each process, we can see that the bulk of the time is wasted in the pure Java serialization of the captured state, while the extraction mechanism (i.e. the core of the entire facility) 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 the methods for the injected frames.

## VI. Conclusions and Future Work

This paper has introduced our approach to support Java thread strong mobility based on the IBM JikesRVM virtual machine, and has outlined how this mechanism is being integrated in the Aglets Mobile Agent Platform in order to exploit such approach. 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 be read since a single execution flow will be followed from the beginning to the end.

Our approach represents an extension of JikesRVM but does not change any part of this JVM. Rather, it exploits some interesting facilities provided by that JVM to avoid many of the drawbacks of the presented solutions. OSR facility also allowed us to capture the state 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 will keep the thread management efficient, and allows having one thread for each agent, overcoming the limitation of the current implementation of the Aglets system.

With regard to future work, we will perform a comparison test between the current Aglets release (with weak mobility) and our JikesRVM-based version (with strong mobility). This comparison will be performed also under critical conditions (such as a large number of agents). ). From the first results reported in section V, we can draw the conclusion that the prototype can be further optimized with respect to the Java serialization bottleneck, in particular trying to reduce the size of the thread state data to be serialized. This perhaps will allow us to reduce strongly the unavoidable gap between a weak agent serialization and a strong one.

## Acknowledgment

Work supported by the Italian MIUR and CNR within the project "IS-MANET, Infrastructures for Mobile ad-hoc Networks" and by the European Community within the project "CASCADAS".

## References

[1] ECMA TC39/TG3. The CLI Architecture. Technical Report, ECMA, October 2001.

[2] A. Acharya, M. Ranganathan, J. Saltz, "Sumatra: A Language for Resource-aware Mobile Programs". 2nd International Workshop on Mobile Object Systems (MOS'96), Linz, Austria, 1996

[3] The Aglets Mobile Agent Platform website http://aglets.sourceforge.net

[4] B.Alpern, C.R. Attanasio, D. Grove and others, "The Jalapeno virtual machine", IBM System Journal, Vol. 39, N°1, 2000

[5] 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

[6] 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

[7] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter 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

[8] L. Bettini and R. De Nicola, "Translating Strong Mobility into Weak Mobility", MA2001, pages 182-197, number 2240, Springer, 2001.

[9] 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

[10] 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

[11] 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

[12] Stephen Fink, and Feng Qian, "Design, Implementation and Evaluation of Adaptive Recompilation with On-Stack Replacement", International Symposium on Code Generation and Optimization San Francisco, California, March 2003

[13] M. 13chetti, "Tireless travel agent Special Report: The Rise Of E-Business/Wheeling And Dealing", available at http://domino.research.ibm.com/comm/wwwr_thinkresearch.nsf/pages/travel199.html

[14] A. Fuggetta, G. P. Picco, G. Vigna, "Understanding Code Mobility", IEEE Transactions on Software Engineering, Vol 24, 1998

[15] J. Gosling, B. Joy, G. Steele, G. Bracha, "The Java Language Specification, second edition", SUN Microsystem

[16] M. Hohlfeld and B.S. Yee, "How to Migrate Agents", Unpublished, available at http://www.cse.ucsd.edu/~bsy/, 1998.

[17] F. Bellifemine, G. Caire, A. Poggi, G. Rimassa, "JADE - A White Paper", EXP in Search of Innovation, TILAB, vol. 3, 2003

[18] N. R. Jennings, "An agent-based approach for building complex software systems", Communications of the ACM, Vol. 44, No. 4, pp. 35-41 (2001)

[19] The JikesRVM project site: http://jikesrvm.sourceforge.net

[20] 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

[21] D. B. Lange, Y. Aridor, "Agent Transfer Protocol (ATP)", IBM=TRL, draft number 4, 19 March 1997

[22] T. Lindholm, F. Yellin, "The Java Virtual Machine Specification, second edition", SUN Microsystem

[23] M. 23, P. McBurney, C. Preist, "Agent Technology: Enabling Next Generation Computing – A Roadmap for Agent Based Computing", AgentLink, http://www.agentlink.org/roadmap

[24] Scott Oaks and Henry Wong, "Java Threads, 2nd edition", Oreilly, 1999

[25] The 25 Project web site: http://25.sourceforge.net/

[26] 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.

[27] “The Java Object Serialization Specification”, Sun Microsystems, 1997

[28] 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

[29] T. Suezawa, "Persistent Execution State of a Java Virtual Machine", ACM Java Grande 2000 Conference, San Francisco, CA, USA, Jun. 2000

[30] E. Truyen, B. Robben, B. Vanhaute, T. Coninx, W. Joosen, P. Verbaeten, "Portable support for Transparent Thread Migration in Java" 4th International Symposium on Mobile Agents 2000 (MA'2000), Zurich, Switzerland, Sep. 2000

[31] Paul R. Wilson, “Uniprocessor Garbage Collector Techniques”, in the Proceedings of the International Workshop on Memory Management (IWMM92), St. Malo, France, September 1992

[32] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel, “The LOCUS distributed operating system”, In Proceeding of the Ninth Symposium on Operating Systems Principles, pages 49-70, ACM 1983.

[33] W. Zhu, C. Wang, F. C. M. Lau, "JESSICA2: A Distributed Java Virtual Machine with Transparent Thread Migration Support". IEEE Fourth International Conference on Cluster Computing, Chicago, USA, September 2002

[34] G. Vigna, G. Cugola, C. Grezzi and G.P. Picco, “Analyzing Mobile Code Languages”, Mobile Object Systems n. 1222, Springer, 1997.

[35] 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.