Implementation of Strong Mobility for Multi-Threaded Agents in Java

Arjav J. Chakravarti Xiaojin Wang
Jason O. Hallstrom Gerald Baumgartner
Dept. of Computer and Information Science
The Ohio State University
395 Dreese Lab., 2015 Neil Ave.
Columbus, OH 43210–1277, USA
{arjav,hallstro,gb}@cis.ohio-state.edu frozen_wang@hotmail.com

Technical Report: OSU-CISRC-2/03-TR06

October 15, 2003

Abstract

Strong mobility, which allows an external thread to transparently migrate an agent at any time, is difficult to implement in Java since the Java Virtual Machine does not allow serializing the runtime stack. We give an overview of our implementation strategy for strong mobility in which each agent thread maintains its own serializable execution state at all times. We explain how to solve the synchronization problems involved in migrating a multi-threaded agent and how to terminate the underlying Java threads in the originating virtual machine. We present experimental results that indicate that our implementation approach will be feasible in practice.

1 Introduction

The advent of Grid Computing [14, 13], and a coordinated approach to sharing computational resources, has improved the reliable utilization of these resources for the solution of complex problems, such as for [12, 32].

Peer-to-Peer computing [33, 40] is a paradigm that adopts a highly decentralized approach to achieve resource sharing. Peer-to-Peer systems are less reliable, and are mainly used for embarassingly parallel applications [39, 11], and simple applications like file-sharing [18, 25].

A confluence of these two technologies will facilitate the building of flexible systems to support dynamic communities of users [22, 15]. Applications that leverage the capabilities of such systems will make use of large amounts of data that are distributed across a network. Program code may be stored at multiple locations, and sub-programs will communicate with one another during execution, i.e., the applications need not necessarily be embarassingly parallel. Decentralized scheduling will be important to avoid bottlenecks, to keep applications scalable, and to dynamically utilize resources.

The vast majority of distributed applications are currently built with distributed object technologies, such as Java RMI, CORBA, COM, or SOAP. These use remote method invocation, where objects that encompass code and data are moved from one location to another on the Grid. These RPC-based approaches do not, however, consider the execution state of the arguments. If a thread is active within one of the arguments

passed to the remote procedure, it does not travel along with the argument. We believe that this is not the appropriate abstraction mechanism for applications that function in a dynamic environment because migration autonomy and transparent thread migration are not supported.

The Mobile Agent abstraction is the movement of code, data and threads from one location to another. Agents have relocation autonomy, i.e., they can relocate or be relocated at will. If the intermediate results of a computation are very large, shipping them over a Grid in a distributed object system can be less efficient than using mobile agents. Mobile agents are immune to brief network outages or congestion during their computation [48, 8, 29, 27]. Also, when using dynamic load-balancing algorithms for the Grid, an agent may need to be told to move at any instant, and to continue execution at a new location. In the peer-to-peer applications envisaged for Grid systems, mobile agents offer a flexible and reliable means of distributing the data and code of a computation around a network, of dynamically moving from host to host as resources become available, and of carrying multiple threads of execution to simultaneously perform computation, the scheduling of other agents, and communication with other agents on the network. Approaches to using mobile agents for Grid Computing have been discussed in [6, 36, 34, 17].

Java is the language of choice for an overwhelming majority of the mobile agent systems that have been developed till date. The popularity of Java is due to its good design, platform independence, security architecture, serialization mechanism, and dynamic class loading. However, the execution model of the Java Virtual Machine [31] does not permit an agent to transfer its execution state, because it is not possile to access the run-time stack and program counter.

A ramification of this constraint is that Java based mobility libraries can only provide *weak mobility* [9]. Weakly mobile agent systems, such as IBM's Aglets framework [28] or other Java-based agent frameworks [26, 41, 20], do not migrate the execution state of methods. The `go()` method, used to move an agent from one virtual machine to another, simply does not return. The agent environment kills the threads currently executing in the agent, without saving their state. The lifeless agent is then shipped to its destination, and is resurrected there. Weak mobility forces programmers to use a difficult programmming style, i.e., the use of callback methods, to account for the absence of migration trasparency.

By contrast, agent systems with *strong mobility* provide the abstraction that the execution of the agent is uninterrupted, only its location changes. Applications that require agents to migrate from host to host while communicating with one another to solve a problem, are affected by the absence of strong mobility. In a weakly mobile system, if two agents are communicating, and one of them relocates, the other agent cannot continue its operation normally. This problem does not exist in strongly mobile systems, because both agents will simply continue execution, with one of them at a new location. The ability of a system to support the migration of an agent at any time by an external thread, is termed Forced mobility. This is particularly useful for load-balancing, and for fault-tolerant applications, and is difficult to implement without strong mobility. Strong mobility allows programmers to use a far more natural programming style.

As an example, consider a network broadcast agent that takes a message and an array of host names as input and relays the message to the hosts. With strong mobility, the code for the broadcast operation could be implemented with a simple loop iterating over the array of host names:

```
public void broadcast(String msg, String[] hosts) {
  for(int i = 0; i < hosts.length; i++)
    try { go(hosts[i]); System.out.println(msg); }
    catch (Exception ex) { } dispose();
}
```

In a weakly mobile system, such as Aglets, the developer needs to provide callback methods for certain mobility-related events (such as arrival at a new host) for manually reconstructing the execution state after a move. For example, a typical implementation of the broadcast agent in an Aglets-like framework would appear as follows:

```
private String hosts[], message;
private int i = 0;

public void onCreation(Object init) {
  hosts = (String[]) ((Object[]) init)[0];
  message = (String) (init[1]);
}

public void onArrival() {
  System.out.println(message);
}

public void run() {
  if(i == hosts.length) dispose();
  i++;    // increment before dispatch()
  try { dispatch(hosts[i-1]); }
  catch (Exception ex) { ... }
}
```

While the intent of the strongly mobile code is easy to understand even for a beginning programmer, it might take an experienced developer some time to understand the weakly mobile code because part of the agent's control flow is hidden within the hosting environment. A single logical operation, such as this network broadcast, must be implemented as a combination of multiple callback methods. While the above example is simplistic, it is representative of any mobile agent application with a complicated pattern of migration.

A number of different approaches have been folllowed to add strong mobility to Java. These can be separated into two broad categories — those that use modified or custom VMs, and those that change the compilation model.

JavaThread[5, 3, 4], D'Agents[19], Sumatra[1], Merpati[42] and Ara[35], all depend on extensions to the standard VM from Sun, whereas the CIA[45] project uses a modification of the Java Platform Debugger Architecture. Forced mobility is not supported by JavaThread, CIA and Sumatra. In addition, JavaThread depends on the deprecated `stop()` method in `java.lang.Thread` to migrate an agent. The D'Agents, Sumatra, Ara and CIA systems do not migrate multi-threaded agents. Merpati does not migrate agent threads that are blocked in monitors. The NOMADS[43, 44] project uses a custom virtual machine known as Aroma, to provide support for forced mobility and the migration of multi-threaded agents. Support for thread migration within a cluster is provided by JESSICA2[49]. The solution does not scale to the Internet or a grid, however, because a distributed VM is used.

Modifying the Java VM, or using a custom VM, has the major disadvantage of a lack of portability. Existing virtual machines cannot be used. It is very difficult to maintain complete compatibility with the Sun Java specification. For example, JavaThread and NOMADS are JDK 1.2.2 compatible, D'Agents relies on a modified Java 1.0 VM, and Merpati and Sumatra are no longer supported. It is also difficult to achieve the efficient performance of the JVM from Sun. NOMADS, Sumatra and Merpati do not support JIT compilation. In addition, some users may prefer to use other VMs of their choice. These problems greatly impact the acceptability and widespread use of mobile agent systems that rely on VM modifications.

Another approach to adding strong mobility to Java is to change the compilation model (by using a preprocessor, by modifying the compiler, or by modifying the generated bytecode), such that the execution state of an agent can be captured before migration.

Projects that follow this approach are WASP [16] and JavaGo[38]. These use a source code preprocessor. However, neither approach supports forced mobility. In addition, JavaGo does not migrate multiple threads

of execution, or preserve locks on migration. Correlate[46] and JavaGoX[37] modify bytecode. Migration may only be initiated by the agent itself, i.e., forced mobility is not supported.

We have chosen to provide strong mobility for Java, by using a preprocessor to translate strongly mobile source code to weakly mobile source code [47]. We present an overview of our implementation approach, in which an agent maintains a movable execution state for each thread at all times. The generated weakly mobile code saves the state of a computation before moving an agent so that the state can be recovered once the agent arrives at the destination. The code translation could be done at the level of bytecode as well. The translation of method calls requires type information, however, and this would involve decompiling bytecode. To avoid this, we use the more convenient source translation mechanism.

Jiang and Chaudhary [24, 23] use a similar approach for C and C++. The scalability of their system is limited by its dependence on a global scheduler to migrate threads. It is also unclear whether they can handle multiple concurrent migrations, which could impact performance. Bettini and De Nicola [2] also use the same idea for agent migration, but they do this for a toy language. Our implementation is designed for the full Java programming language, without any assumptions about the nature of the systems in which mobile agents will be deployed.

2 Overview

Our implementation approach for strong mobility in Java is to translate strongly mobile code into weakly mobile code. We currently target the IBM Aglets weak mobility system.

To achieve strong mobility, every method in the original agent class is translated to a `Serializable` inner class of the agent; this represents the activation record for that method. The local variables, parameters and program counter are converted to fields of this class, and so can be saved. This inner class contains a `run()` method that represents the body of the original method. The generated weakly mobile agent class contains an array of activation record objects that acts as a virtual method table.

Threads in Java are not `Serializable` because they use native code. However, the state of every thread of execution also needs to be maintained, so that the thread can be restarted at the destination. This is achieved by using a `Serializable` wrapper around each Java `Thread`. This wrapper contains its own stack of activation record objects that mirrors the run-time stack of the underlying thread of execution. When a method is called, the appropriate entry from the method table is cloned and put on the stack. After passing the arguments, the `run()` method executes the original method body while updating the program counter.

To allow an agent to be moved at arbitrary points of time, statement execution and the program counter update should be executed atomically. To accomplish this, the original source code is translated to a form that allows the state of the agent to be saved for each executed statement; this while maintaining the semantics of these statements. This requires different translation rules for each type of statement in the Java language.

A `go()` method is called on a multi-threaded agent to send it to a new location. The `Serializable` wrappers then bring the agent threads to a standstill, and save the state of these threads. The agent then relocates. Only the `Serializable` wrappers of the threads are moved. These wrappers create new `Thread` objects at the destination, and set their state. Thus, the original execution state of the agent is recreated, and the execution continues. If potentially long-running operations like `Object.wait(long)` are executing at the source, they are interrupted, and the time left for them to finish execution is saved so that they can continue after a move.

The use of multi-threaded agents makes synchronization issues very important. For a multi-threaded system, the program counter must be incremented atomically with the following instruction; two agents must not dispatch one another at the same time, and two threads within the same agent must not dispatch the

agent simultaneously. User-specified `synchronized` blocks in the original Java source code also need to be translated so that they can be carried along by an agent.

Synchronization control in mobile agents is non-trivial, but we offer an approach that we believe is no more taxing than programming for a traditional non-mobile system.

Each statement and its corresponding program counter update are wrapped inside a logical synchronized block to preserve their atomicity, and prevent agent relocation before they complete. Synchronizing on the agent instance is unacceptable because it would prevent threads from executing translated statements in parallel. The problem is a basic *readers/writers conflict*, where the threads that execute the translated statements are readers, and the thread that executes the *go()* method acts as a writer. A *writers priority* solution is used which is similar to the reader/writer locks solution in [21].

Each agent maintains *locks* that represents the boolean predicate, 'OK to execute statements?'. The number of locks is the same as the number of reader threads, and are acquired and released by readers before and after statement execution. When a call is made to `go()`, the writer thread acquires all the locks. The state of the agent is then saved, and the agent moves to its destination.

To prevent deadlock when two agents call one another, the call to `go()` is synchronized on the agent context instead of on the caller. Similarly, if multiple threads within the same agent attempt to move the agent, deadlock is prevented by having each thread test a synchronized condition variable in the agent context. The first writer thread will set this variable, and the subsequent writers will read the variable and then give up their locks.

The Java semantics for `synchronized` blocks is that a thread must acquire and release a lock on a particular object or class before entering and leaving a synchronized portion of code. If the agent moves when execution is inside that protected region, the lock is released. Users may use `synchronized` to protect an agent's internal data structure, and protection must be extended across machine boundaries.

This transparency is achieved by introducing serializable locks in place of the standard Java object locks. During the translation phase, a serializable lock is introduced for each object that requires synchronization. Every `synchronized` block is replaced by a call to acquire and release a lock at the beginning and at the end of a block. `synchronized` blocks are often used in conjunction with the `wait()` and `notify()` operations. These too, are appropriately translated such that their semantics are preserved across a relocation.

We have run a number of benchmarks to test our translator for strong mobility. We then compared the performance of these translated agents to that of the corresponding IBM Aglets. Some simple optimizations to the generated code were performed by hand, and the performance enhancement was observed. The measurements confirm the feasibility of our approach.

3 Language and API Design

Our support for strong mobility consists currently of the interface `Mobile` and the classes `MobileObject` and `ContextInfo`.

3.1 Interface `Mobile`

Every mobile agent must (directly or indirectly) implement the interface `Mobile`. Similar to Java RMI, a client of an agent must access the agent through an interface variable of type `Mobile` or a subtype of `Mobile`. Interface `Mobile` is defined as follows:

```
public interface Mobile extends java.io.Serializable {
  public void go(java.net.URL dest)
    throws java.io.IOException, com.ibm.aglet.RequestRefusedException,
```

```
    edu.ohio_state.cis.brew.MoveRefusedException;

  // methods for synchronization of multi-threaded agents
  ...
}
```

Like `Serializable`, interface `Mobile` is a *marker interface*. It indicates to a compiler or preprocessor that special code might have to be generated for any class implementing this interface. `go()` moves the agent to the destination with the URL `dest`. This method can be called either from a client of the agent or from within the agent itself. The second parameter indicates whether the call was made from within the agent or from outside.

The `go()` method currently throws an Aglet exception. In a future version of the translator, we will add some more of our own exception classes so that the surface language is independent of the implementation.

3.2 Class `MobileObject`

Class `MobileObject` implements interface `Mobile` and provides two methods: `getContextInfo()` and `go()`. To allow programmers to override these methods, they are implemented as wrappers around `native` implementations that are translated into weakly mobile versions. A mobile agent class is defined by extending class `MobileObject`.

```
public class MobileObject implements Mobile {
  private native ContextInfo realGetContextInfo();

  private native void realGo(java.net.URL dest)
    throws java.io.IOException, com.ibm.aglet.RequestRefusedException,
    edu.ohio_state.cis.brew.MoveRefusedException;

  protected ContextInfo getContextInfo() {
    return realGetContextInfo();
  }

  public void go(java.net.URL dest)
    throws java.io.IOException, com.ibm.aglet.RequestRefusedException,
    edu.ohio_state.cis.brew.MoveRefusedException {
      realGo(dest, outsideCall);
  }

  // methods required for  synchronization of a multi-threaded agent.
  ...
}
```

The method `getContextInfo()` provides any information about the context in which the agent is currently running, including the host URL and any system objects that the host wants to make accessible to a mobile agent. If `go()` is called from within an agent method, the instruction following the call to `go()` is executed on the destination host. Typically, an agent would call `getContextInfo()` after a call to `go()` to get access to any system resources at the destination.

3.3 Class `ContextInfo`

Class `ContextInfo` is used for an agent to access any resources on the machine it is currently running on:

```
public class ContextInfo implements java.io.Serializable {
  private java.net.URL hnostURL;
  public ContextInfo (java.net.URL h) {
    hostURL = h;
  }
  public java.net.URL getHostURL() {
    return hostURL;
  }
  ...
}
```

Currently, we only provide a method `getHostURL()` that returns the URL of the agent environment in which the agent is running. We will extend the functionality of class `ContextInfo` in future translator versions.

For providing access to special-purpose resources such as databases, an agent environment can implement the method `getContextInfo()` to return an object of a subclass of class `ContextInfo`. By publishing the interface to this object, agents can be written to access those resources.

3.4 Strongly Mobile User Code

For writing a mobile agent, the programmer must first define an interface, say `Agent`, for it. This interface should extend interface `Mobile` and declare any additional methods. All additional methods must be declared to throw `AgletException`. An implementation of the mobile agent then extends class `MobileObject` and implements interface `Agent`. A client of the agent must access the agent through a variable of the interface type `Agent` and through a proxy object similar as in Java RMI or in Aglets.

When calling a method on an agent, an exception will be thrown if the agent is not reachable. As in Java RMI, this is expressed by declaring that the method might throw an exception. Our current implementation uses the exception class `AgletException`.

4 Translation from Strong to Weak Mobility

In this section, we present the translation mechanism for methods, classes, statements, and exceptions.

4.1 Translation of Methods

To make the execution state of a method serializable, we implement activation record objects for agent methods. For each agent method, the preprocessor generates a class whose instances represent the activation records for that method. As multiple invocations may be active simultaneously (e.g., recursive methods), these objects are cloneable. An activation record class for a method is a subclass of the abstract class `Frame`:

```
public abstract class Frame implements Cloneable, Serializable {
  public Object clone() { ... }
  public abstract void run();
  public abstract void setPCForMove();
}
```

The original method code will be inserted within the `run()` method. For example, consider a method `foo`:

```
public void foo(int x) throws AgletsException {
  int y;
  // blocks of statements
  BC1
  BC2
}
```

The parameter `x`, local variable `y` and the program counter become fields of class `Foo`. Method `setArgs()` passes the values of the method parameters.

```
public void setArgs(Object initial) {
  Object[] init = (Object[])initial;
  Integer r0 = (Integer)init[0];
  x = r0.intValue();
}
```

A `setPCForMove()` method is necessary to allow the arbitrary suspension and movement of a thread of execution. This method saves the current `programCounter`, before setting it to -1 to ensure that no further instructions get executed before the agent moves. The `run()` method contains the translated version of `foo()`, which includes code for incrementing the program counter, as well as code which allows `run()` to resume computation after a move. Every thread needs to poll whether it is time to move or not. It does this by acquiring and releasing a lock before and after every logical statement in the code. This is done by the `AgentImpl.this.request_read()` and `AgentImpl.this.read_accomplished()` calls. The generated activation record class for `foo` :

```
protected class Foo extends Frame {
  int x, y, progCounter = 0; Object trgt;
  void setPCForMove() { ... }
  void run() {
    try { ...
      AgentImpl.this.request_read();
      if ((progCounter == 0)) {
        progCounter+=1; BC1
      }
      AgentImpl.this.read_accomplished();
      AgentImpl.this.request_read();
      if ((progCounter == 1)) {
        progCounter+=1; BC2
      }
      AgentImpl.this.read_accomplished();
    }
    catch(AgletsException e) { ... }
  }
  ...
}
```

4.2 Translation of Agent Classes

The generated agent class contains an array of `Frame` objects that is used as a virtual method table. The appropriate entry from the method table is cloned and put on the thread wrapper stack, when a method is called.

Suppose, for example, we have an agent class `AgentImpl` of the form :

```
public class AgentImpl extends MobileObject implements Agent {
  int a;
  public AgentImpl() { /* init code */ }
  public void foo(int x) throws AgletsException {
    BC;
  }
}
```

Because this class (indirectly) implements the `Mobile` interface, the preprocessor translates it into the code described as follows :

The original agent method `foo()` gets translated into an inner class `Foo`. There are two `foo()` methods in the generated code, of which `foo(Object, Object)` is a preparatory method. Its first parameter is a reference to the wrapper of the thread on which the method is to be executed. An activation record is created and pushed onto the wrapper stack. The second parameter is an `Object` array that contains the arguments to the original `foo()` method. These are given to the activation record.

The second `foo()` method has the same order, type and number of parameters as the original untranslated method. All the calls to the original `foo` method from within the agent now go to this method. The method obtains a reference to the wrapper of the currently executing thread and packages its parameters in an `Object` array, before calling the `foo(java.lang.Object, java.lang.Object)` method described above. The activation record on the top of the stack is then executed.

```
public void foo(Object target, Object init) { ... }

public void foo(int x) { ...
  // fooThread is the wrapper of
  // currently executing thread.
  foo(fooThread, new Object[] {new Integer(x), ... });

  // method call to execute body of
  // original foo method
  fooThread.run1(); return;
}
```

The `handleMessage()` method, is an Aglets method that handle messages sent to the agent. The Aglets system does not allow method invocations from outside the agent; only message sends. All the non-private agent methods are specified inside `handleMessage()`. For example, if the `foo()` method in the untranslated agent could be invoked by an external thread, and a message is received for `foo()`, a new thread is *created*. `foo(java.lang.Object, java.lang.Object)` is called, and the activation record on top of the stack is executed.

```
public boolean handleMessage(Message msg) {
  // an option corresponding to every
  // non-private method in the agent
  if (msg.sameKind("foo")) { ...
    // fooThread is the wrapper of the new thread
    foo(fooThread, msg.getArg());
    fooThread.start();
    ...
    return true;
  }
  ...
}
```

4.2.1 Control Statements

Control statements such as conditionals and loops must be translated to make the control flow explicit by manipulating the program counter. The statement `stmt` is translated into `stmt'`, and `inpc(stmt)` and `outpc(stmt)` are the values of the program counter before and after executing the statement. In the following example, the translation of control statements have been shown without the synchronization code needed for a multi-threaded agent.

```
if (cond) stmt1; else stmt2;
```

is now translated as shown below. `thisIf` refers to the `if` statement being translated.

```
if (((progCounter > inpc(thisIf))
    && (progCounter < outpc(stmt1')))
    || (progCounter == inpc(thisIf) && cond)) {
  stmt1';
}
else {
  if (progCounter == inpc(thisIf))
    progCounter = outpc(stmt1') + 1;
}
if ((progCounter >= inpc(stmt2'))
   && (progCounter < outpc(stmt2'))) {
  stmt2';
}
if ((progCounter == outpc(stmt1'))
   || (progCounter == outpc(stmt2'))) {
  progCounter = outpc(stmt2') + 1;
}
```

Similarly, the translation of

```
while (cond) { stmt; }
```

generates the following code, where `thisWhile` refers to the `while` statement being translated.

```
while ((progCounter >= inpc(thisWhile))
      && (progCounter <= outpc(stmt'))) {
  if (progCounter == inpc(thisWhile) && !cond) {
    progCounter = outpc(stmt')+1;
    break;
  }
  stmt';
  if (progCounter == outpc(stmt')) {
    progCounter = inpc(thisWhile);
  }
}
```

4.2.2 Method Call

If an instance of an agent is to be created, a constructor is translated into a request to the Aglet context to create the appropriate agent, and to initialize it via `onCreation()`.

```
Agent a = new Agent1();
```

becomes

```
AgletProxy a=getAgletContext().createAglet(getCodeBase(),"Agent1",null);
```

Method calls to methods outside the agent must be translated into invocation request messages because methods cannot be invoked directly from outside an Aglet.

```
a1.m1(argObject);
```

is translated as follows if the target method is outside the agent

An Object array, `m1Args` in the example below, contains all the arguments to a method, as well as a proxy to the caller agent and an identifier for the caller thread. Once the message send has taken place, the calling thread must block, and wait for a return from the method. To accomplish this, a `wait()` operation is executed. When a reply is received from the callee, and the method is value-returning, the return value is stored in the wrapper of the calling thread, and a `notify()` is called to wake the thread.

```
Object[] m1Args = new Object[3];
m1Args[0] = argObject;
m1Args[1] = callerThread;
m1Args[2] = callerAglet;
a1.sendMessage(new Message("m1", m1Args));
```

4.2.3 Return Values and Exceptions

If the caller is outside the agent, the translation of the return statement is as follows. The `callerAglet` proxy, and the identification number of the calling thread are provided by the calling Aglet. The former is used to send a reply message to the correct Aglet. The `replyObject` array in the example below contains the return value, as well as a number to correctly identify the calling thread.

```
return obj;
```

is translated into

```
Object[] replyObject = new Object[2];
replyObject[0] = obj;
replyObject[1] = callerThread;
callerAglet.sendMessage("reply", replyObject);
```

The easiest strategy is to treat exceptions as a special kind of return value. A throw statement is implemented by assigning the exception to the return value variable. After a method call, the value returned by the method is be tested to check whether an exception was thrown, or whether a normal value was returned.

Our translator translates almost the entire Java language. Some portions of the translator have not yet been implemented completely, due to time constraints. We believe that these issues are simple enough, and that they can be satisfactorily resolved. The mobility translator has been implemented as a preprocessor to the Brew compiler. The compiler is still under development, and as as yet does not do type-checking. For this reason, it needs to be hard-coded into the translator as to whether method calls and returns are to targets outside the agent, or within the agent. The translation of inner classes, `try` blocks, label—s, and the `assert`, `break` and `continue` statements has not yet been implemented. Name-mangling to support nested blocks and overloaded methods, and the translation of method calls inside expressions also need to be completed.

5 Resource Access

When accessing global resources it is desirable to distinguish between global names on the current virtual machine and global names on the home platform of the agent. To allow agent developers to access platform-bound resources remotely, we introduce the `global` field declaration prefix. Use of the prefix indicates that a particular field should be created (and accessed) on the home platform. For example:

```
private global InputStream is;
```

For each field prefixed with the `global` keyword, the preprocessor generates code to register an RMI server with the home platform. Each RMI server is a simple wrapper, delegating calls to the original field instance, to which it maintains a reference. Special accessor methods are also provided by these servers to handle field assignment, scalar field access, and access to field members within a global field. These field servers are created and registered immediately after the agent is constructed. Any agent code that accesses the global field is translated to access the resource through the corresponding RMI proxy.

A similar problem arises when examining access to fields and methods which are both public and static. Consider, for example, an agent that wishes to roughly approximate the time it takes for it to move between two platforms. To get an accurate measurement without being affected by time-zone changes, the agent needs to access the method `System.currentTimeMillis()` from the home platform.

To provide agent developers flexibility in specifying whether access to a static method or field refers to the home VM, we introduce syntax for retroactively making a static method or field global. By default, access to a static method or field will refer to the VM on which the agent currently resides. To indicate that the home VM should be used to perform the access, we use a retroactive `global` declaration as follows:

```
global long System.currentTimeMillis();
global PrintStream System.out;
```

The first declaration indicates that all calls to `System.currentTimeMillis()` should refer to the agent's home VM, regardless of the agent's location. Similarly, the second declaration indicates that whenever the `out` field of the `System` class is referenced, the reference refers to the `out` instance on the home VM. The implementation of remote resource access has not yet been completed due to time constraints.

6 Multi-Threaded Agents

The multi-threading support provided by Java consists of the classes `Thread` and `ThreadGroup` and the interface `Runnable`, which allow us to create multiple threads of execution within the agent, and to manage groups of threads as a unit.

Java Threads are not serializable because they involve native code. In order to accomplish the migration of threads, the state of each thread needs to be saved in a serializable format that can then be relocated.

6.1 MobileThread and MobileThreadGroup

The serializable wrapper classes `MobileThread` and `MobileThreadGroup` are used around the Java library classes `Thread` and `ThreadGroup`, respectively. On the creation of new `MobileThread` and `MobileThreadGroup` objects, new `Thread` and `ThreadGroup` objects respectively, are created to perform the actual execution. `MobileThread` thus contains all the information about its underlying thread, that is needed to reconstruction the state of that thread after a move. `MobileThreadGroup`

acts similarly with respect to `ThreadGroup`. When an agent moves to a new location, only the wrappers are moved. At the destination these wrappers create new `Thread` and `ThreadGroup` objects, and set their state, so that execution can continue at the destination. Each `MobileThread` also belongs to a particular `MobileThreadGroup`, such that when a `MobileThread` object recreates a thread of execution, that `Thread` is also assigned to the same `ThreadGroup` as at the source location.

The class `MobileThread` contains a `start()` method which is called to begin execution of a `MobileThread`. This can happen after it has been created for the first time, or when the agent starts up all the threads after moving to a new location. This method calls the `start()` method of the underlying `Thread`, which then calls the `run()` method of its target, the `MobileThread` wrapper.

The `run()` method checks the `MobileThread` stack. If the stack is empty, the `MobileThread` is a newly created one, and has to call the `run()` method of its `Runnable` target. If the `MobileThread's` stack is not empty, this means that the activation records on the stack need to be executed instead. The `run2()` method is called to execute the contents of the stack. This method repeatedly checks if the stack is empty. If it is not, it calls `run1()`. `run1()` takes whatever activation record is on the top of the stack, and executes its body.

```
public class MobileThread implements Runnable, Serializable {
  ...
  // run-time stack
  public java.util.Stack stack;

  // reference to underlying Thread
  private transient Thread t = null;

  // called by the onArrival() method to create new Thread at destination
  public void reInit() { ... }

  // saves state of Thread before a move
  public void packUp() { ... }

  // called by the agent before a move to end wait(), sleep() and join()
  // operations, as well as save time remaining to complete operation
  // at destination
  public void interruptForMove() { ... }

  // called by the run() method of underlying Thread to execute either
  // current stack contents, or run() method of the Runnable target
  public void run() { ... }

  // executes activation record on top of the stack
  public void run1() { ... }

  // repeatedly calls run1() to execute activation records on stack
  public void run2() { ... }

  // For all the methods in the Thread API
  public void start() { ... }
  public final String getName() { ... }
  public static void sleep(long millis) throws InterruptedException
    { ... }
}
```

```
public class MobileThreadGroup implements Serializable {
  // reference to underlying ThreadGroup
  private transient ThreadGroup t;

  // called by onArrival() method to create new ThreadGroup
  // at destination
  public void reinit() { ... }

  // saves state of ThreadGroup before move
  public void packUp() { ... }

  // called by the agent before a move to end wait(), sleep() and join()
  // operations of every thread in threadgroup, as well as save time
  // remaining to complete operation at destination
  public void interruptForMove() { ... }

  // starts up all the MobileThreads within the MobileThreadGroup after
  // arrival at new location
  public void start() { ... }

  // For all the methods corresponding to the ThreadGroup API methods
  public int activeCount() { ... }
  public int enumerate(MobileThread list[]) { ... }
}
```

The pre-processor translates the strongly mobile agent code to weakly mobile code, as explained in Section 4. Furthermore, the pre-processor replaces every occurence of `Thread` and `ThreadGroup` in the original code with `MobileThread` and `MobileThreadGroup`, respectively. In this manner, every reference to a `Thread` or `ThreadGroup` object in the original code is now translated to a reference to a `MobileThread` or `MobileThreadGroup` object. We thus ensure that every original operation on a `Thread` or `ThreadGroup` in user code is now made to go through our new wrapper classes.

The mobility translator translates every occurrence of the word `Thread` in the source code with the word `MobileThread`. This ensures that the calls to the methods of `Thread` go through the serializable wrapper, and that the `run()` method of a multi-threaded Agent now executes as activation records on the stack of the thread wrapper.

```
Thread t = Thread.currentThread();
```

is translated to:

```
MobileThread t=MobileThread.currentThread();
```

6.2 Static Methods of java.lang.Thread

Translating each occurrence of `Thread` to `MobileThread` in source code raises the issue of handling the static methods of `java.lang.Thread` correctly. When `MobileThread.currentThread()` is called, it calls `Thread.currentThread()` in turn. This returns a reference to the currently executing `Thread`. Now however, it is necessary to return a reference to the `MobileThread` wrapper over the currently executing `Thread`. How is that to be known?

The solution is to maintain a `static` hashtable that contains a mapping of `Thread`s to their corresponding `MobileThread`s. In this way, `MobileThread.currentThread()` returns the correct `MobileThread` object.

Similarly, the other static methods of `MobileThread` (`sleep()`, `enumerate()`, etc.) use the hashtable, where necessary, to return the appropriate `MobileThread` references. A similar table is not required for the mobile threadgroups, because the `ThreadGroup` class contains no static methods.

6.3 Relocating a Multi-threaded Agent

The `go()` method is called on a multi-threaded agent. This calls the `realGo()` method, which first checks whether this agent is already being moved or not. If a move is in progress, a `MoveRefusedException` is thrown. Otherwise, the situation is that of a readers/writers problem, where the thread that wishes to move the agent, acquires locks such that every `Thread` within the agent blocks and comes to a standstill. Each `MobileThread` makes an —interrupt()— call to its `Thread`, thus terminating any `wait()`, `join()` or `sleep()` operations. If any of these operations are timed, the number of milliseconds that remain are saved such that the operations can be completed at the destination.

The `packUp()` method of the `main` agent threadgroup wrapper is called. From here, the `packUp()` method of each threadgroup and thread wrapper under `main` is called, and the state of its underlying threadgroup and thread saved. The system threads are then forced to terminate and the agent is relocated by using the Aglets `dispatch()` method. Thread termination is an important issue because the `java.lang.Thread API` does not permit direct termination of a thread. Section 7 explains how this is accomplished.

At the new location, the Aglets `onArrival()` method calls the `reinit()` method of the main threadgroup wrapper. This method then creates a new `ThreadGroup`, sets its state, and then calls the `reinit()` method of each threadgroup and thread wrapper under `main`. Each wrapper's `reinit()` method creates a new `Thread` or `ThreadGroup`, and sets its state.

To begin execution, the `onArrival()` method now calls the `start()` method of the `main` threadgroup wrapper, which results in calls to the `start()` methods of all `MobileThread`s to begin execution of their underlying threads.

7 Synchronization Issues

There are three major issues that need to be handled correctly for the synchronization control of a multi-threaded agent - preserving the atomicity of a logical instruction; preventing deadlock when agents dispatch one another, or when multiple threads attempt to dispatch the agent; preserving the semantics of Java `synchronized` blocks across a migration.

7.1 Protection of Agent Stacks

An agent should not be moved while it is executing a statement. To avoid this happening, it is necessary to protect every program counter increment and its following statement. Synchronizing on the agent will reduce parallelism dramatically. The problem can be reduced to a basic *readers/writers conflict*, where the increment of the program counter, and the execution of the following statement by each thread, acts as a *reader*; the *writer* being the thread that calls *go()*. This problem is solved by using a variant of the solution in [21]. *lock*s are maintained by each agent. The boolean predicate they represent is, 'OK to execute statements?'. The number of locks equals the number of executing threads within the agent. Reader threads acquire and release locks before and after executing logical statements, by `request_read()` and `read_accomplished()` calls.

```
AgentImpl.this.request_read();
if(pc==15) {
```

```
    pc++; stmt;
  }
  AgentImpl.this.read_accomplished();
```

Before executing a `wait()`, `join()` or `sleep()` operation in user code, a thread calls the overloaded `request_read(boolean)` method to acquire the lock and also to inform the thread wrapper that the operation is potentially long running. When a thread makes a call to `go()`, it is designated as a writer. The writer thread attempts to acquire all the agent locks. Once it makes this attempt, no reader can acquire a lock. The writer then calls `interruptForMove()` on all curently executing `MobileThread`s. `interruptForMove()` is meant to ensures that an agent is not made to wait for an undue amount of time before moving. The wrapper repeatedly calls `interrupt()` on its thread until it stops the operation. The method whose execution was interrupted, checks whether the wrapper interrupted the thread. If so, the program counter is decremented so that the interrupted operation resumes at the destination. If the interrupted operation was timed, like `wait(long)`, the time remaining for the operation to complete is saved by the `MobileThread`. An `InterruptedException` is thrown if the wrapper did not cause the interrupt. A count of the number of currently active readers is maintained. This count is incremented when a reader requests the lock by calling `request_read()`, and decremented when the lock is released by a `read_accomplished()` call. As the readers only relinquish their locks at this stage, depending on whether the writer is an internal or an external thread, the count must go down to one or zero. The writer then calls the `packUp()` method of the `main` threadgroup wrapper. This results in each `MobileThread` saving the state of its `Thread`, setting the program counter of the activation records on its stack to -1, and disallowing the popping of activation records. At this point, the writer releases its locks. All the waiting readers are released and are free to continue execution. But, the program counters have all been set to a negative value, and so no further statements can be executed. The reader threads run through to completion and are no longer active. None of the activation records are popped during this step.

It is necessary to place a `synchronized` block around a `wait()` and `join()` operation, in order to ensure that all the threads that were waiting at the source are restored to their original condition at the destination, before the other agent threads are restarted. For example, if `o.wait()` was the statement being executed, `o.wait()` must be called before any other thread can acquire the lock on `o` and call `o.notify()`. Avoiding this situation necessitates restarting the threads that were waiting at the source first. The thread wrappers create and start these threads. When the threads enter the `synchronized` block around `o.wait()`, they inform their wrappers. A thread is not allowed to execute beyond a `o.wait()`, because it is not allowed to acquire an agent lock again. Once all the threads have informed their wrappers that they have begun waiting, all the agent threads are allowed to proceed normally.

```
o.wait();
```

is translated to :

```
// acquire lock and inform thread wrapper that
// a long operation is to be executed.
request_read(longOperation);

if (progCounter == i) {
  progCounter += 1;
  synchronized (o) {
    // inform MobileThread that a synchronized block has been entered
    ...

    try { o.wait(); }
```

```
      catch (InterruptedException e) {
        // if execution was interrupted by the thread wrapper, inform
        // wrapper that a wait has been interrupted
          progCounter -= 1;

        // if thread interrupted by another thread, inform thread wrapper
        // that long operation terminated, and throw InterruptedException
          ...
          throw e;
      }
    }
    // inform thread wrapper that long operation has terminated,
    // and release lock
    ...
  }

  // release lock
  read_accomplished();
```

The translation of a timed `wait()` operation is similar to that of the untimed `wait()`, just discussed. The time that remains for the operation to complete is saved before a move.

```
o.wait(someLong);
```

becomes :

```
// acquire lock and inform thread wrapper that
// a long operation is to be executed.
request_read(longOperation);

if (progCounter == i) {
  progCounter += 1;
  waitTime = someLong;
}
if (progCounter == i+1) {
  progCounter += 1;
  synchronized (o) {
    // inform MobileThread that a synchronized block has been entered
    ...

    baseTime = System.currentTimeMillis();
    try { o.wait(waitTime); }
    catch (InterruptedException e) {
      // if execution was interrupted by the thread wrapper, inform
      // wrapper that a wait has been interrupted
        progCounter -= 1;
        waitTime = waitTime -
        (System.currentTimeMillis() - baseTime);

      // if thread interrupted by another thread, inform thread wrapper
      // that long operation terminated, and throw InterruptedException
        ...
        throw e;
```

```
      }
    }
    // inform thread wrapper that long operation has terminated,
    // and release lock
    ...
  }

  // release lock
  read_accomplished();
```

Extending the guarantee of transparent interruption and restoration of long-running operations across a relocation to library code, is non-trivial. Libraries may implement guarded `wait()`s, `sleep()`s or `join()`s by using loops with condition checks around these operations; an approach similar to that described in [30]. When a `MobileThread` interrupts its `Thread` with an `interrupt()` call, its held locks would not be released immediately in this case. Agent relocation would be delayed, perhaps for an unacceptable amount of time. The correct handling of long operations inside library code requires one of the following approaches :

- Library methods could be modified such that they do not implement guarded operations. This would need access to the library source code.
- The library could be passed through the mobility translator as well. This would also require access to the library source.
- A native code wrapper method could be placed around `wait()` in `java.lang.Object`, and around `sleep()` and `join()` in `java.lang.Thread`. The wrapper would be similar to the translation of `wait()` that was described earlier in this section. The addition of some accessor methods and fields to `java.lang.Thread` would permit a `MobileThread` to determine if its `Thread` was executing a long operation. Although implementing this approach is not possible with the current version of the translator, this portion of the translation could be done at the byte-code level.
- Library calls that potentially lead to `wait()`s, `sleep()`s or `join()`s could be flagged at compile-time to indicate that the call might be long-running, and that no guarantee about the immediate migration of an agent can be made. A message could be printed out to the programmer and the decision would have to be taken by him/her as to whether the call could take a long time to finish or not, or whether the delay in migration would be acceptable.

We believe that most calls to the standard Java library will terminate within an acceptable amount of time. In the absence of a mechanism that can save the state of a `Thread` executing a library call, we believe that the best approach is for the compiler to flag those calls that might not terminate quickly. The programmer then needs to decide whether a library call that might delay the dispatch of an agent is acceptable or not. Should the programmer desire a finer granularity of control over the potentially long-running operations, he/she should pass the library through the mobility translator.

If two agents try to dispatch one another, the synchronization technique we have adopted could lead to a deadlock. Agent `a` would synchronize on itself for executing the statement `b.go(dest)`, which would require synchronization on `b` to protect the integrity of `b`'s stacks. If similarly, `b` would execute `a.go(dest)`, a deadlock would result. To prevent this, the call of `dispatch()` within `realGo()` is synchronized on the agent context instead of on the caller.

There are two ways in which the `go()` method call in the strongly mobile agent can be translated and handled by a weakly mobile Aglet. The first is if the call is translated to the `sendMessage()` method call

of the Aglets library. The `handleMessage()` method of each Aglet simply calls `go()` directly. This causes a problem when the two agents try to move one another. If `a` executes `b.go(dest)` and `b` executes `a.go(dest)`, both calls get translated to the corresponding `sendMessage()` calls. Each Aglet sends a `go` message to the other. If `a`'s `go()` method synchronizes on the agent context first, every thread inside `a` is interrupted before a move. This includes the thread that is executing the `sendMessage()` call to move `b`. The expected behaviour is that all of `a`'s threads get interrupted, `a`'s state is saved, and `a` is moved to its destination. Since `a`'s message send to move `b` is interrupted, `b` does not move. However, the `sendMessage()` within `a`, in turn ends up calling the Aglets `waitForReply()` method. The `waitForReply()` method contains a guarded `wait()`; this is implemented using a loop around a `wait()`. Thus, even if the calling thread is interrupted before `waitForReply()` returns, the interrupt will have no effect until the loop condition becomes false. The loop condition, in turn depends on a reply being received from `b`'s `go()` method. `b`'s `go()` method cannot reply because it is blocked - waiting to synchronize on the agent context. Deadlock exists.

The other translation of the `go()` method call would also use `sendMessage()`. Considering the example above, when `a` executes `b.go(dest)` and `b` executes `a.go(dest)`, both calls again get translated to `sendMessage()` calls, and each Aglet sends a `go` message to the other. However, to avoid the deadlock condition previously described, a new `MobileThread` is created within the `handleMessage()` method of each Aglet. This then calls the `go()` method to actually dispatch the Aglet i.e. the `go()` methods of both `a` and `b` begin to execute. Considering the case where `a`'s `go()` method synchronizes on the agent context first, `a` gets dispatched to its destination. Then, `b` synchronizes on the agent context and is dispatched to its own destination. This approach might not be acceptable in some applications, where only the agent that first synchronizes on the context should be dispatched, and not both the agents.

Since it is the implementation of the `waitForReply()` method inside the Aglets library that is responsible for the deadlock condition, we chose to implement the second approach to agent dispatch, even though we believe the first to be better. We plan to target other weak mobility systems in the next version of our translator, and then switch to the first approach.

If multiple threads within the same agent attempt to move the agent, deadlock could still result. This is because more than one thread could call `go()`. Only one of them will actually synchronize on the agent context. Now, when this writer thread attempts to acquire all the locks, it will not be able to. This is because the other threads that are attempting to move the agent will have acquired their locks by a call to `request_read()`, and then called `go()`. These threads will be blocked, waiting to acquire a lock on the agent context, and will not release their locks with `read_accomplished()` calls. An additional level of synchronization is introduced in order to avoid this. Every agent maintains a condition variable in the agent context. This indicates whether the agent is currently being moved or not. The first writer thread will acquire a lock on this variable, test and set it, and then release the lock. Subsequent threads will acquire the lock, test the property, and release the lock by throwing a `MoveRefusedException`.

7.2 Synchronization Blocks

The Java semantics for `synchronized` blocks or methods is that their lock is released when the agent is migrated. When users use synchronization in an agent to protect the agents' internal data structure, this protection must extend across machine boundaries - to prevent the data structure from being corrupted after arrival.

The idea is to take the notion of a synchronized code block, and allow a thread to migrate within the code block, retaining the synchronization lock throughout the migration. This then extends the concept of a synchronized block across machine boundaries, enabling a programming style familiar to Java programmers.

For weakly mobile languages, synchronized blocks are a non-issue since code never executes beyond

the call to `go()`. In strongly mobile systems, however, a call to `go()` may appear at any point within a synchronized block. Since the lock is released upon dispatch, this enables other threads to enter the block before the original thread continues execution at the new site. The original thread then waits for the new thread to finish execution before re-acquiring the lock and proceeding. These semantics do not preserve the notion of a synchronized code block.

The difficulties stem from the fact that object locks are not stored within the object during serialization, but are hidden within the virtual machine. To tackle this problem we introduce serializable locks in place of the standard Java object locks. Client programmers use the standard `synchronized` keyword to enforce synchronization constraints, just as before. During the translation phase, an object of class `MobileMutex` is introduced for each object that requires synchronization. Whenever a programmer requests object locking through the use of the Java `synchronized` keyword, the lock is actually taken out and released via calls to `lock()` and `unlock()` on the associated `MobileMutex` object. In this way, synchronized blocks and methods, are eliminated from the original source, and re-implemented using the new serializable locking mechanism. The overhead is minimal, and synchronization semantics are preserved while the agent is on the move.

Consider the following class `MobileMutex`, which is designed to maintain synchronization locks across VM boundaries:

```
public class MobileMutex implements java.io.Serializable {
  boolean locked = false;
  public MobileMutex() {}

  public synchronized void lock() {
    // inform MobileThread that a synchronized block has been entered
    ...
    if (!locked) locked = true;
    else
      while(true)
        try { wait();  locked=true;  break; }
        catch(InterruptedException ex) {
          // if execution interrupted by thread wrapper, inform wrapper
          // of wait being interrupted, and decrement program counter
          ...
          break;
        }
  }

  public synchronized void unlock() {
    locked = false;  notify();
  }
}
```

`synchronized` blocks are often used in conjunction with `wait()` and `notify()` operations. These need to be translated such that their semantics are preserved, even after the translation of `synchronized` blocks. The semantics of wait are that a call to `o.wait()` must be inside a Java `synchronized` block that synchronizes on `o`, and that once a thread begins to wait, it must release its locks on the object `o`.

A Java `synchronized` block for the `Object o`, is translated to a logical synchronized block that is bounded by a `o.lock()` and an `o.unlock()`. Only one thread should be inside the logical synchronized block at a time. Before a thread calls `o.wait()`, it calls `o.unlock()` to release its lock on `Object o`. Both `o.unlock()` and `o.wait()` are inside the same `synchronized` block. This makes sure that no

other thread can call `o.wait()` or `o.notify()` before the current thread has actually begun executing the `wait()` operation in `java.lang.Object`.

If an interrupt is called on the waiting thread, an `InterruptedException` is thrown. A check is made as to whether the interrupt call was made because a move is in progress. If that is the case, the program counter is decremented by one. `wait()` will now be called at the destination, but `unlock()` will not be called again. If the interrupt was not caused by a move, an exception is thrown.

Now, when a `notify()` is called on the `Object o`, one thread is removed from `o`'s wait set, and must now compete to reacquire its lock on `Object o`. The thread does this by calling `lock()`. Only when it acquires the lock on `o`, can it proceed with the rest of the synchronized block.

```
synchronized (o) { o.wait(); }
```

is translated to

```
if (progCounter == i) {
  // acquire lock and inform thread wrapper that long operation
  // is to be executed
  ...
  progCounter += 1;
  o.lock();
  // inform thread wrapper that long operation has terminated
  ...
}

// acquire lock and inform thread wrapper that a long operation
// is to be executed
...
synchronized (o) {
  if (progCounter == i+1) {
    progCounter += 1; o.unlock(); }
  if (progCounter == i+2) {
    progCounter += 1;
    // inform MobileThread that a synchronized block has been entered
    try { o.wait(); }
    catch (InterruptedException e) {
      // if execution was interrupted by the thread wrapper, inform
      // wrapper that a wait has been interrupted
        ...
        progCounter -= 1; break;

      // if thread interrupted by another thread, inform wrapper that
      // long operation was ended, and throw InterruptedException
        ...
        throw e;
    }
  }
}

// inform thread wrapper that long operation has terminated
...
if (progCounter == i+3) {
  // acquire lock and inform thread wrapper that a long operation
  // is to be executed
```

```
    ...
    progCounter += 1; o.lock();
    // inform thread wrapper that long operation has terminated
    ...
  }

  if (progCounter == i+4) {
    // acquire lock
    ...
    progCounter += 1; o.unlock();
    // release lock
    ...
  }
```

Similarly, the `notify()` call inside the translated agent also needs to be within a logical synchronized block, and also within a Java `synchronized` block. `notify()` is translated as shown below :

```
synchronized (o) { o.notify(); }
```

becomes

```
// acquire lock and inform thread wrapper that a long operation
// is to be executed.
...
if (progCounter == i) {
  progCounter += 1; o.lock();
}

// inform thread wrapper that long operation has terminated,
// and release lock
...
if (progCounter == i+1) {
  // acquire lock
  ...
  progCounter += 1;
  synchronized (o) { o.notify(); }
  // release lock
  ...
}

if (progCounter == i+2) {
  // acquire lock
  ...
  progCounter += 1; o.unlock();
  // release lock
  ...
}
```

If synchronized blocks are to be made transparent across moves, a `MobileMutex` object needs to be added to the object on which synchronization is desired. At the current state of implementation, this is only possible if the programmer has access to the source code of that object, if the object is itself an agent, or if the programmer has source access to every synchronization on the object. In the next version of the translator, we will address this issue by associating a `MobileMutex` object with every `java.lang.Object`.

8 An Example Agent

We now use our preprocessor for strong mobility to translate a simple computing agent.

8.1 Strongly Mobile Interface

A user defines an interface as follows :

```
public interface Agent extends Mobile {
  public int calculate(int count)
         throws AgletException;
}
```

8.2 Strongly Mobile Class

The untranslated class in a strongly mobile system, contains a method with a simple double loop. The `calculate()` method is called, and passed an argument `count`. The sum of the natural numbers up until `count` is calculated 10 times and returned.

```
public class AgentImpl extends MobileObject implements Agent {
  int numberOfRuns;

  public AgentImpl() {
    numberOfRuns = 10;
  }

  public int calculate(int count) throws AgletException {
    int sum = 0;
    int outerIndex = 0;
    int innerIndex = 0;

    while (outerIndex < numberOfRuns) {
      innerIndex = 0;
      while (innerIndex <= count) {
        sum = sum + innerIndex;
        innerIndex++;
      }
      outerIndex++;
    }
    return sum;
  }
}
```

8.3 Generated Weakly Mobile Class

The untranslated class in a strongly mobile system, contains a method with a simple double loop. When `calculate()` is called, the sum of the natural numbers until its argument are calculated,

```
public class AgentImpl extends Aglet implements Runnable {

  int numberOfRuns;
```

```
protected class Calculate extends Frame {
  int count, sum, outerIndex, innerIndex;
  private int returnValue,progCounter=0,
    tempProgCounter=0;
  private Object trgt,callerThread,callerAglet;
  boolean messageSent = false;

  public void setArgs(Object initial) {
    Object[] init = ((Object[])initial);
    Object t0 = init[0];
    Integer r0 = ((Integer)t0);
    count = r0.intValue();
    callerThread = init[1];
    callerAglet = init[2];
  }

  protected void setTarget(Object target) {
    trgt = target;
  }

  protected void setPCForMove() {
    if ((progCounter > -1)) {
      tempProgCounter = progCounter;
      progCounter = -1;
    }
  }

  protected void run() {
    try {
      AgentImpl.this.read_accomplished();
      AgentImpl.this.request_read();
      if ((progCounter < 0))
        progCounter = tempProgCounter;
      AgentImpl.this.read_accomplished();
      AgentImpl.this.request_read();
      if ((progCounter == 0)) {
        progCounter += 1;
      }
      AgentImpl.this.read_accomplished();
      AgentImpl.this.request_read();
      if ((progCounter == 1)) {
        progCounter += 1;
        sum = 0;
      }
      AgentImpl.this.read_accomplished();
      AgentImpl.this.request_read();
      if ((progCounter == 2)) {
        progCounter += 1;
        outerIndex = 0;
      }
      AgentImpl.this.read_accomplished();
      AgentImpl.this.request_read();
      if ((progCounter == 3)) {
```

```
  progCounter += 1;
  innerIndex = 0;
}
AgentImpl.this.read_accomplished();
AgentImpl.this.request_read();
while (((progCounter >= 4) &&
    (progCounter <= 9))) {
  if (((progCounter == 4) &&
    !((outerIndex < numberOfRuns)))) {
    progCounter = 10;
    break;
  }
  AgentImpl.this.request_read();
  if ((progCounter == 4)) {
    progCounter += 1;
    innerIndex = 0;
  }
  AgentImpl.this.read_accomplished();
  while (((progCounter >= 5) &&
        (progCounter <= 7))) {
    if (((progCounter == 5) &&
       !((innerIndex < count)))) {
      progCounter = 8;
      break;
    }
    if ((progCounter == 5)) {
      progCounter += 1;
      sum = (sum + innerIndex);
    }
    AgentImpl.this.read_accomplished();
    AgentImpl.this.request_read();
    if ((progCounter == 6)) {
      progCounter += 1;
      (innerIndex)++;
    }
    AgentImpl.this.read_accomplished();
    AgentImpl.this.request_read();
    if ((progCounter == 7))
      progCounter = 5;
  }
  AgentImpl.this.read_accomplished();
  AgentImpl.this.request_read();
  if ((progCounter == 8)) {
    progCounter += 1;
    (outerIndex)++;
  }
  AgentImpl.this.read_accomplished();
  AgentImpl.this.request_read();
  if ((progCounter == 9))
    progCounter = 4;
}
AgentImpl.this.read_accomplished();
AgentImpl.this.request_read();
```

```
      if ((progCounter == 10)) {
        progCounter += 1;
        {
          Object[] replyObject = new Object[2];
          replyObject[0] = new Integer(sum);
          replyObject[1] = callerThread;
          AgletProxy caller = ((AgletProxy)callerAglet);
          caller.sendMessage(new Message("reply", replyObject));
        }
      }
      AgentImpl.this.read_accomplished();
      AgentImpl.this.request_read();
    }
    catch(AgletException e) {
      Object replyObject = e;
      MobileThread.currentThread().setResult(replyObject);
    }
  }
}

protected class OnCreation extends Frame {
  ...
  protected void run() {
    ...
    AgentImpl.this.request_read();
    if ((progCounter == 1)) {
      progCounter += 1;
      numberOfRuns = 10; }
    AgentImpl.this.read_accomplished();
    ...
  }
}

Frame[] vtable = new Frame[] {new Calculate(),new OnCreation()};
final int _calculate = 0;
final int _onCreation = 1;
Hashtable mTNosTable = new Hashtable();
boolean receiveMessage = true;
Object replyValue;
int alock = 0, mobileThreadNos = 0;
Readers_Writer rw = new Readers_Writer();
Readers_Writer msgRW = new Readers_Writer();
MobileMutex mM = new MobileMutex();
MobileThreadGroup agentGroup;

synchronized int getMobThNo() {
  return ++(mobileThreadNos);
}

public void lock() {
  mM.lock();
}
```

```
public void unlock() {
  mM.unlock();
}

public void incCount() {
  rw.incCount();
}

public void decCount() {
  rw.decCount();
}

public void request_read() {
  rw.request_read();
}

public void read_accomplished() {
  rw.read_accomplished();
}

public void request_write(MobileThreadGroup mTG) {
  rw.request_write(mTG);
}

public void write_accomplished() {
  rw.write_accomplished();
}

public void calculate(Object target, Object init) {
  MobileThread trgt = ((MobileThread)target);
  Calculate frame = ((Calculate)vtable[_calculate].clone());
  trgt.stack.push(frame);
  trgt.setSource(this);
  frame.setArgs(init);
  frame.setTarget(trgt);
}

public void calculate(int count) {
  MobileThread calculateThread = MobileThread.currentThread();
  calculate(calculateThread, new Object[] {new Integer(count),
    new Integer(calculateThread.getSerialNumber()), getProxy()});
  calculateThread.run1();
  return;
}

public void onCreation(Object init) {
  agentGroup = new MobileThreadGroup("agentGroup");
  MobilityListener myListener = new Listener();
  addMobilityListener(myListener);
  MobileThread trgt = new
  MobileThread(agentGroup, "onCreationThread");
  int mobThNo = getMobThNo();
  trgt.setSerialNumber(mobThNo);
```

```
  mTNosTable.put(new Integer(mobThNo), trgt);
  OnCreation frame = ((OnCreation)vtable[_onCreation].clone());
  trgt.stack.push(frame);
  trgt.setSource(this);
  frame.setArgs(init);
  frame.setTarget(trgt);
  synchronized (getAgletContext()) {
    Hashtable hT = new Hashtable();
    hT.put("moveInProgressKey", new Boolean(false));
    getAgletContext().
      setProperty(("moveInProgress" + getAgletID().toString()), hT);
  }
  trgt.start();
}

public boolean handleMessage(Message msg) {
  msgRW.request_read();
  if (!(receiveMessage)) {
    msgRW.read_accomplished();
    return false;
  }
  if (msg.sameKind("calculate")) {
    MobileThread calculateThread =
      new MobileThread(agentGroup, "calculateThread");
    int mobThNo = getMobThNo();
    calculateThread.setSerialNumber(mobThNo);
    mTNosTable.put(new Integer(mobThNo), calculateThread);
    calculate(calculateThread, msg.getArg());
    calculateThread.start();
    msgRW.read_accomplished();
    return true; }
  if (msg.sameKind("go")) {
    try {
      go(((URL)msg.getArg()), true);
    }
    catch(Exception e) {
      msgRW.read_accomplished();
      return false;
    }
    msgRW.read_accomplished();
    return true;
  }
  if (msg.sameKind("reply")) {
    Object[] replyObject = ((Object[])msg.getArg());
    replyValue = replyObject[0];
    MobileThread srcThread;
    try {
      Integer srcThreadNo = ((Integer)replyObject[1]);
      srcThread = ((MobileThread)mTNosTable.get(srcThreadNo));
    }
    catch(Exception e) {
      msgRW.read_accomplished();
      return false;
```

```
        }
        alock = 1;
        synchronized (srcThread) {
          srcThread.notify();
        }
        while ((alock != 0))
          ;
        msgRW.read_accomplished();
        return true;
      }
      else {
        msgRW.read_accomplished();
        return false;
      }
    }

    class Listener implements MobilityListener {
      public void onArrival(MobilityEvent ev) {
        rw = new Readers_Writer();
        msgRW = new Readers_Writer();
        receiveMessage = true;
        synchronized (getAgletContext()) {
          Hashtable hT = new Hashtable();
          hT.put("moveInProgressKey", new Boolean(false));
          getAgletContext().
            setProperty(("moveInProgress" + getAgletID().toString()), hT);
        }
        agentGroup.reinit();
        agentGroup.start();
      }
      ...
    }

    public void go(URL dest, boolean outsideCall) {
           throws MoveRefusedException, java.io.IOException,
           com.ibm.aglet.RequestRefusedException {
        realGo(dest, outsideCall);
    }

    public void realGo(URL dest, boolean outsideCall)
           throws MoveRefusedException, java.io.IOException,
           com.ibm.aglet.RequestRefusedException {
      ...
    }
  }
```

9 Performance

9.1 Optimizations

The translation mechanism discussed do far is overly conservative and thus inefficient. We have identified some optimizations for the above translation algorithms, that are simple enough to be done automatically by

a compiler:

- If a method is not recursive, or if it is tail-recursive and the compiler can determine that the execution time is bounded, it should not be translated into a class.
- To reduce the overhead of synchronization and program counter update, statements should be grouped to form logical, atomic statements.
- If the number of statements executed inside a loop is sufficiently small, and the statements are simple i.e. no method calls or loops, there is no need to take a checkpoint on every loop iteration. A lock acquire and release could be made every *N* (say 10,000) simple statements. This would mean that in the case of a `go()` call, upto *N* statements would need to be executed before the move actually takes place. We believe that with the processor speeds available today, this is acceptable, and that any further delay before a move will be insignificant.
- Loop unrolling could reduce loop overhead.
- Method call overhead would decrease if method inlining were to be used.
- If a local variable is limited in scope to only one logical statement, it should remain a local variable, and should not be translated into a field of the generated class.
- A further optimization would be to generate code that checkpoints every *N* simple statements, or every *N* milliseconds.

9.2 Measurements

Measurements were taken to estimate the cost of the described translation mechanism for agents. Standard Java benchmarks were rewritten in the form of both strongly mobile agents, and Aglets. This did not involve changing the timed code significantly. The only changes that needed to be made to the original benchmarks' code were made to avoid method calls inside expressions. This is because the preprocessor does not as yet handle these.

The strongly mobile agents were passed through the translator. We then used simple manual optimization techniques to improve the performance of the translated agents. These are - the grouping of simple statements to form logical, atomic statements; the obtaining and releasing of locks every 10,000 simple statements for a loop; the inlining of method calls to simple methods that in turn, do not contain method calls.

The running times and memory footprints of the translated agents and the manually optimized agents were compared with the equivalent weakly mobile Aglets. The results have been presented in table 1, and in table 2. A major contributor to the poor running times of the recursive benchmark programs, is the Garbage Collector that runs several times a second during their execution.

We performed some further optimzations on the Linpack benchmark. The time taken by Linpack depends to a great extent on a particular method call inside a double-nested loop. This method contains another loop. We manually inlined this method, and measured execution time with the inner-most loop untranslated, and with the translated loop unrolled. The running time comparisons are presented in table 3, and the memory footprint results are in table 4. A user could obtain a performance improvement by including annotations in the code; to inform the preprocessor how to optimally translate certain code portions.

A comparison of the code sizes of the agent code output by the preprocessor, and that of the corresponding simple Aglets, was made for the benchmarks discussed above. This was done by comparing their

Benchmark	Translated Code	Optimized Code
Crypt(array size - 3000000)	5.61X	1.23X
Crypt(array size - 3000000) multi-threaded version - 1 thread	5.96X	1.30X
Crypt(array size - 3000000) multi-threaded version - 2 threads	6.00X	1.41X
Crypt(array size - 3000000) multi-threaded version - 5 threads	5.60X	1.31X
Linpack(500 X 500)	10.00X	1.75X
Linpack(1000 X 1000)	9.48X	1.65X
Tak(100 passes)	245.30X	220.83X
Tak(10 passes)	247.00X	213.60X
Simple Recursion (sum of first 100 natural nos. - 10000 passes)	68.27X	60.75X

Table 1: Execution time of Strongly Mobile Agents compared to corresponding Aglets

Benchmark	Translated Code	Optimized Code	Aglet
Crypt	32.10	30.69	30.44
Crypt - multi-threaded 1 thread	32.54	30.82	30.35
Crypt - multi-threaded 2 threads	32.56	30.82	30.35
Crypt - multi-threaded 5 threads	32.54	30.83	30.38
Linpack(500 X 500)	31.02	30.02	28.34
Linpack(1000 X 1000)	58.27	52.94	51.24
Tak(100 passes)	22.04	21.99	20.98
Tak(10 passes)	22.05	22.02	20.98
Simple Recursion	22.03	21.82	21.02

Table 2: Memory utilization of Strongly Mobile Agents and the Aglets (MB)

`.class` files. For the benchmarks discussed previously, the translated agents are between 6 and 14 times the sizes of the simple Aglets.

The overhead of migrating agents depends on the amount of state that the agent requires to carry along with itself. This is dependent on the number of threads within the agent, and on the number of frames on the runtime stack of the threads. The migration costs of moving a single threaded agent with different numbers of frames on the stack have two components - the time required to pack up the agent state, and the time to move the agent. The latter is the time required for the translated agent to execute the Aglets dispatch method. We compare this against the time required for the transfer of the simple benchmark Aglet. Agents and Aglets are transferred between ports on the same machine, in order to obtain a meaningful comparison that is unaffected by network delay. The results for different stack sizes are shown in table 5.

Similarly, the dependence of the migration cost of a multi-threaded agent, on the number of threads is shown in table 6.

The measurements were taken on a Sun Enterprise 450 (4 X UltraSPARC-II 296MHz), with 1GB of main memory, running Solaris. We used the Sun JDK 1.4.0_01 HotSpot VM in mixed mode execution, with

Linpack Optimizations	Inner Loop Untranslated	Inner Loop Unrolled 2 times	Inner Loop Unrolled 10 times
Linpack (500 X 500)	1.02X	1.21X	0.75X
Linpack (1000 X 1000)	1.02X	1.15X	0.76X

Table 3: Execution time of Optimized Strongly Mobile Agents compared to Aglets for Linpack

Linpack Optimizations	Inner Loop Untranslated	Inner Loop Unrolled 2 times	Inner Loop Unrolled 10 times
Linpack (500 X 500)	29.9	30.19	30.48
Linpack (1000 X 1000)	52.8	53.12	53.40

Table 4: Memory utilization of Optimized Strongly Mobile Agents for Linpack (MB)

the heap size limited to 120MB.

10 Conclusions

We have argued that strong mobility is an important abstraction for developing Grid Computing applications, and have outlined a source translation scheme that translates strongly mobile code into weakly mobile code, by using a preprocessor. The API for the strongly mobile code and the translation mechanism are designed to give programmers full flexibility in using multi-threaded agents, and in dealing with any synchronization problems.

We are able to handle almost the entire Java programming language. Time constraints mean that the translation of some simple constructs like inner classes, and `try` blocks have not yet been implemented. If an agent uses library code that contains guarded `wait`, `sleep` or `join` calls, rapid termination before a move cannot be guaranteed. `synchronized` blocks that synchronize on an untranslated Object in user code cannot be transparently migrated. In both these situations, the translator is designed to display a warning for the programmer. Some resources need to be accessed on the machine where the agent originated, and should be declared `global` by the programmer. An RMI server to do this needs to be implemented. Timed operations, like open network connections, are not preserved across a relocation. Mobile agents need to be prevented from sharing objects with one another, or non-mobile objects. We will investigate using Isolates [10] for this purpose.

Source code, rather than bytecode translation, does not involve decompilation, and is more convenient. The performance measurements indicate that our approach to achieving strong mobility for Java is practical. In future, we will use analysis techniques to automate the generation of optimized source code. Measurements also indicate that performance can be improved further by allowing programmers to make annotations to source code.

Our preprocessor currently generates Java code that uses IBM's Aglets library. In future versions of our translator, we will instead target the *ProActive* [7] weak mobility system, or RMI directly.

Number of stack frames	Agent pack time	Agent dispatch time	Aglets dispatch time
1	6	2436	1750
2	5	5421	1875
3	5	5410	1830

Table 5: Migration Time for Single-threaded Strongly Mobile Agents and Aglets (ms) - Linpack

Number of threads	Agent pack time	Agent dispatch time	Aglets dispatch time
1	9	5266	1782
2	10	5133	1860
5	16	5126	1803

Table 6: Migration Time for Multi-threaded Strongly Mobile Agents and Aglets (ms) - 5 frames on main thread stack, 2 frames on other threads' stacks - Multi-threaded Crypt

References

[1] Anurag Acharya, Mudumbai Ranganathan, and Joel Saltz. Sumatra: A Language for Resource-Aware Mobile Programs. In Jan Vitek, editor, *Mobile Object Systems: Towards the Programmable Internet*, number 1222 in Lecture Notes in Computer Science, pages 111–130. Springer-Verlag, 1996.

[2] Lorenzo Bettini and Rocco De Nicola. Translating Strong Mobility into Weak Mobility. In *Mobile Agents*, pages 182–197, 2001.

[3] Sara Bouchenak. Making Java Applications Mobile or Persistent. In *6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'2001)*, San Antonio, Texas, USA, January 2001.

[4] Sara Bouchenak and Daniel Hagimont. Pickling threads state in the Java system. In *Technology of Object-Oriented Languages and Systems Europe (TOOLS Europe'2000)*, Mont Saint Michel/Saint Malo, France, June 2000.

[5] Sara Bouchenak, Daniel Hagimont, Sacha Krakowiak, Nol De Palma, and Fabienne Boyer. Experiences Implementing Efficient Java Thread Serialization, Mobility and Persistence. Technical Report RR-4662, INRIA, December 2002.

[6] Jeffrey Bradshaw, Niranjan Suri, Alberto J. Caas, Robert Davis, Kenneth M. Ford, Robert R. Hoffman, Renia Jeffers, and Thomas Reichherzer. Terraforming Cyberspace. In *Computer*, volume 34(7), pages 48–56. IEEE, July 2001.

[7] Denis Caromel, Wilfried Klauser, and Julien Vayssière. Towards seamless computing and metacomputing in Java. *Concurrency: Practice and Experience*, 10(11-13):1043–1061, 1998.

[8] Daniel T. Chang and Danny B. Lange. Mobile Agents: A New Paradigm for Distributed Object Computing on the WWW. In *OOPSLA96 Workshop : Toward the Integration of WWW and Distributed Object Technology*, pages 25–32, San Jose, CA, October 1996. ACM Press, NY.

[9] Gianpaolo Cugola, Carlo Ghezzi, Gian Pietro Picco, and Giovanni Vigna. Analyzing mobile code languages. In Jan Vitek, editor, *Mobile Object Systems: Towards the Programmable Internet*, number 1222 in Lecture Notes in Computer Science, pages 93–110. Springer-Verlag, 1996.

[10] Grzegorz Czajkowski and Laurent Daynès. Multitasking without Compromise: A Virtual Machine Evolution. In *Proceedings of the 2001 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2001*, Tampa, Florida, USA, October 2001.

[11] distributed.net. http://www.distributed.net.

[12] Network for Earthquake Engineering Simulation. http://www.neesgrid.org.

[13] I. Foster, C. Kesselman, J.Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration, 2002. http://www.globus.org/research/papers.html.

[14] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of High Performance Computing Applications*, 15(3):200–222, 2001.

[15] Ian Foster and Adriana Iamnitchi. On Death, Taxes, and the Convergence of Peer-to-Peer and Grid Computing. In *2nd International Workshop on Peer-to-Peer Systems*, Berkeley, CA, February 2003.

[16] Stefan Fünfrocken. Transparent Migration of Java-based Mobile Agents: Capturing and Reestablishing the State of Java Programs. In Kurt Rothermel and Fritz Hohl, editors, *Proceedings of the Second International Workshop on Mobile Agents (MA '98)*, volume 1477 of *Lecture Notes in Computer Science*, pages 26–37, Stuttgart, Germany, 9–11 September 1998. Springer-Verlag.

[17] R. Ghanea-Hercock, J.C. Collis, and D.T. Ndumu. Co-operating mobile agents for distributed parallel processing. In *Third International Conference on Autonomous Agents AA99*, Mineapolis, MN, May 1999. ACM Press.

[18] Gnutella. http://www.gnutella.com.

[19] Robert S. Gray, George Cybenko, David Kotz, Ronald A. Peterson, and Daniela Rus. D'Agents: Applications and Performance of a Mobile-Agent System. *Software— Practice and Experience*, 32(6):543–573, May 2002.

[20] Thomas Gschwind. Comparing Object Oriented Mobile Agent Systems. In Ciarán Bryce, editor, *6th ECOOP Workshop on Mobile Object Systems*, Sophia Antipolis, France, 13 June 2000.

[21] Allen Holub. Reader/writer locks. *Java World*, April 1999. http://www.javaworld.com/javaworld/jw-04-1999/jw-04-toolbox-p3.html.

[22] A. Iamnitchi, I. Foster, and D. Nurmi. A Peer-to-peer Approach to Resource Discovery in Grid Environments. In *11th Symposium on High Performance Distributed Computing*, Edinburgh, UK, August 2002.

[23] Hai Jiang and Vipin Chaudhary. Compile/Run-time Support for Thread Migration. In *16th International Parallel and Distributed Processing Symposium*, Fort Lauderdale, FL, April 2002.

[24] Hai Jiang and Vipin Chaudhary. On Improving Thread Migration: Safety and Performance. In Sartaj Sahni, Viktor K. Prasanna, and Uday Shukla, editors, *9th International Conference on High Performance Computing — HiPC2002*, volume 2552 of *Lecture Notes in Computer Science*, pages 474–484, Berlin, Germany, December 2002. Springer-Verlag.

[25] Kazaa. http://www.kazaa.com.

[26] Joseph Kiniry and Daniel Zimmerman. A Hands-on Look at Java Mobile Agents. *IEEE Internet Computing*, 1(4):68–77, August 1997.

[27] David Kotz, Robert Gray, and Daniela Rus. Future Directions for Mobile-Agent Research. *IEEE Distributed Systems Online*, 3(8), August 2002. http://dsonline.computer.org/0208/f/kot.htm.

[28] Danny B. Lange and Mitsuru Oshima. *Programming & Deploying Mobile Agents with Java Aglets*. Addison-Wesley, 1998.

[29] Danny B. Lange and Mitsuru Oshima. Seven Good Reasons for Mobile Agents. *Communications of the ACM*, 42(3), March 1999.

[30] Doug Lea. *Concurrent Programming in Java[tm]: Design Principles and Patterns*. The Java Series. Addison Wesley, 2nd edition, 1999.

[31] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, Massachusetts, 1996.

[32] Grid Physics Network. http://www.griphyn.org.

[33] A. Oram. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*. O'Reilly, 2001.

[34] B.J. Overeinder, N.J.E. Wijngaards, M. van Steen, and F.M.T. Brazier. Multi-Agent Support for Internet-Scale Grid Management. In O. Rana and M. Schroeder, editors, *AISB'02 Symposium on AI and Grid Computing*, pages 18–22, April 2002.

[35] Holger Peine and Torsten Stolpmann. The Architecture of the Ara Platform for Mobile Agents. In Radu Popescu-Zeletin and Kurt Rothermel, editors, *First International Workshop on Mobile Agents MA'97*, volume 1219 of *Lecture Notes in Computer Science*, pages 50–61, Berlin, Germany, April 1997. Springer Verlag.

[36] O.F. Rana and D.W. Walker. The Agent Grid: Agent-Based Resource Integration in PSEs. In *16th IMACS World Congress on Scientific Computation, Applied Mathematics and Simulation*, Lausanne, Switzerland, August 2000.

[37] Takahiro Sakamoto, Tatsurou Sekiguchi, and Akinori Yonezawa. Bytecode Transformation for Portable Thread Migration in Java. In *Proceedings of Agent Systems, Mobile Agents, and Applications*, volume 1882 of *Springer Verlag Lecture Notes in Comuter Science*, 2000.

[38] Tatsurou Sekiguchi, Hidehiko Masuhara, and Akinori Yonezawa. A Simple Extension of Java Language for Controllable Transparent Migration and its Portable Implementation. In *Coordination Models and Languages*, pages 211–226, 1999.

[39] SETI@home. http://setiathome.ssl.berkeley.edu.

[40] Clay Shirky. What is P2P ... And What Isn't? *O'Reilly Network*, November 2000. http://www.openp2p.com/pub/a/p2p/2000/11/24/ shirky1-whatisp2p.html.

[41] Luis Moura Silva, Guiherme Soares, Paulo Martins, Victor Batista, and Luis Santos. The Performance of Mobile Agent Platforms. In *Proceedings of the First International Symposium on Agent Systems and Applications and Third International Symposium on Mobile Agents*, pages 270–271. IEEE, 1999.

[42] Takashi Suezawa. Persistent execution state of a Java virtual machine. In *Proceedings of the ACM 2000 conference on Java Grande*, pages 160–167. ACM Press, 2000.

[43] Niranjan Suri, Jeffrey M. Bradshaw, Maggie R Breedy, Paul T. Groth, Gregory A. Hill, Renia Jeffers, and Timothy S. Mitrovich. An Overview of the NOMADS Mobile Agent System. In Ciarán Bryce, editor, *6th ECOOP Workshop on Mobile Object Systems*, Sophia Antipolis, France, 13 June 2000.

[44] Niranjan Suri, Jeffrey M. Bradshaw, Maggie R. Breedy, Paul T. Groth, Gregory A. Hill, and Renia Jeffers. Strong Mobility and Fine-Grained Resource Control in NOMADS. In *Proceedings of the Second International Symposium on Agent Systems and Applications / Fourth International Symposium on Mobile Agents*, pages 79–92, Zurich, September 2000.

[45] Illmann T., Krüger T., Kargl F., and Weber M. Transparent Migration of Mobile Agents using the Java Platform Debugger Architecture. In *Proceedings of the 5th International Conference on Mobile Agents*, Atlanta, Georgia, USA, December 2001.

[46] Eddy Truyen, Bert Robben, Bart Vanhaute, Tim Coninx, Wouter Joosen, and Pierre Verbaeten. Portable Support for Transparent Thread Migration in Java. In *Proceedings of the Joint Symposium on Agent Systems and Applications / Mobile Agents*, Zurich, Switzerland, 13 September 2000.

[47] Xiaojin Wang. Translation from Strong Mobility to Weak Mobility for Java. Master's thesis, The Ohio State University, 2001.

[48] J. White. Mobile Agents. In J. Bradshaw, editor, *Software Agents*. MIT Press, 1997.

[49] Wenzhang Zhu, Cho-Li Wang, and Francis 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.