Reliability Through Strong Mobility

Xiaojin Wang Jason Hallstrom Gerald Baumgartner

Dept. of Computer and Information Science
The Ohio State University
395 Dreese Lab., 2015 Neil Ave.
Columbus, OH 43210–1277
{xw, hallstro, gb}@cis.ohio-state.edu

June 17, 2001

Abstract

We argue that weak mobility is unreliable because it leads to an unnatural, non-modular programming style that is difficult to reason about and hard to debug. We present an extension of Java with strong mobility and show how it can be translated to weak mobility by a preprocessor. The language is designed to give programmers the same flexibility in resolving synchronization issues as with weak mobility.

1 Motivation

As with any software system, when building distributed systems it is important that the program is designed in a modular fashion. A modular structure of data and control flow is essential for developing a proper mental model of the software and for reasoning about it. Without a programming language supporting such modularity, and a consequently poor mental model, developers cannot be expected to manage the complexity of a large-scale system.

Weak mobility [1], such as in IBM's Aglets framework [6] or in other Java-based agent frameworks [5, 8, 4], does not allow migration of the execution state of methods (i.e., local variables and the program counter). The `dispatch` operation simply does not return. Instead, the framework allows the developer to tie code to certain mobility-related events. In Aglets, the developer can provide callback code that will execute when an object is first created, just before an object is dispatched, just after an object arrives at a site, etc. This results in a quite unnatural 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. A solution in an Aglets-like framework might look as follows:

```
private String hosts[];
private String 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++;    // must be incremented before dispatch()
```

```
        try {
            dispatch(hosts[i-1]);
        }
        catch (Exception ex) {
            // skip hosts that are not reachable
        }
    }
```

Since `onCreation()` can only take a single `Object` as argument, the user must manually package the parameters `message` and `hosts` in an array.

A language with support for strong mobility provides an much better mental model for writing mobile agents since it allows the programmer to express the control flow across migration within the language. Using a Java-like language supporting strong mobility the solution to the above problem is straightforward:

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

It should be clear that the first implementation violates a number of software design principles. While the code is intended to execute a single logical action, namely the network broadcast, the program logic is scattered across multiple methods. Perhaps even more obtrusive is the introduction of numerous fields that are more naturally suited as local variables. This is a common blunder in disguise: the classic beginning programmer's faux pas of introducing global variables in place of locals. In addition, call-back methods that take arguments, such as `onCreation()`, cannot be statically type-checked.

Even for a beginning programmer the second implementation of the broadcast algorithm is immediately clear. Conversely, even for an experienced developer it takes some time to understand the first implementation, because part of the control flow of the agent is hidden in the agent environment and scattered across the callback methods. While the above example is very simplistic and could be implemented more easily with remote method invocation, it is representative of any mobile agent application with a complicated pattern of movement from host to host. With any weak mobility library, the program logic for migrating the agent, i.e., the for-loop and the call of the method `broadcast()` in the strongly mobile version, must be distributed over multiple methods. Reasoning about this program logic in weakly mobile code is similar in complexity to reasoning about programs with goto.

It is often argued that weak mobility is preferable over strong mobility since it gives the programmer more flexibility in resolving synchronization problems. For example, if the operation to migrate an agent is a primitive operation and if two agents try to dispatch each other, a deadlock could result. Similarly, migrating a multithreaded agent can cause problems. However, by using a `go()` method that might throw an exception in case of synchronization problems it is possible to give programmers the same flexibility in resolving problems as with weak mobility. On the contrary, the simpler programming model and the more modular style of a strongly mobile agent also makes the code for resolving synchronization problems more modular and easier to reason about.

We claim that the improved modularity of strongly mobile code results in more reliability because it is easier to reason about the code and to debug it.

The problem with providing strong mobility in Java [3] is that the Java Virtual Machine [7] does not provide any access to the runtime stack or to the program counter for security reasons. Any mobility library for Java can, therefore, only provide weak mobility. For providing strong mobility, it is necessary to change the compilation model (by using a preprocessor, by modifying the compiler, or by modifying the generated bytecode) or to modify the virtual machine. E.g., Fünfrocken uses a preprocessor [2], NOMADS uses a custom virtual machine [10, 9].

In the following section, we present the design of our language extension and of our library support for providing mobility. In Section 3, we explain the design of our translation from strong mobility to weak mobility. Following an example in Section 4, we discuss synchronization issues in Section 5, and then provide conclusions.

2 Language and API Design

Our support for strong mobility consists currently of the interface `Mobile` and the two classes `MobileObject` and `ContextInfo`. While the design looks like a library API, it is really a language extension since our proposed translation mechanism compiles away the interface `Mobile` and the class `MobileObject`.

2.1 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;
}
```

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.

As explained in Section 3 below, we use the IBM Aglets library for implementing our support for strong mobility. This is currently reflected in the list of exceptions that can be thrown by `go()`. In a future version, we will add our own exception class(es) so that the surface language is independent of the implementation.

2.2 Class **`MobileObject`**

Class `MobileObject` implements interface `Mobile` and provides the 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.

```
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;
    protected ContextInfo getContextInfo() {
        return realGetContextInfo();
    }
    public void go(java.net.URL dest)
        throws java.io.IOException,
               com.ibm.aglet.RequestRefusedException {
        realGo(dest);
    }
}
```

A mobile agent class is defined by extending class `MobileObject`.

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.

The method `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. If `go()` is called from within an agent method `foo()`, 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.

2.3 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 hostURL;
    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. In a future version, we will extend class `ContextInfo`.

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.

2.4 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 an exception of type `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. In our current design, we use the exception class `AgletException`. In a future version, we will provide our own exception class.

3 Translation from Strong to Weak Mobility

For efficiency reasons it would be desirable to provide virtual machine support for strong mobility. However, a preprocessor or compiler implementation has the advantage that the generated code can run on any Java VM, and that it is easier to implement and to experiment with the language design. Once the semantics of the language and API, and the translation are properly designed, virtual machine support could be added for efficiency.

We chose to design the translation mechanism for a preprocessor that translates strongly mobile code into weakly mobile code that uses the Aglets library. This allows leveraging off of Aglets for the implementation of the transport mechanism.

For implementing strong mobility in a preprocessor, it is necessary to save the state of a computation before moving an agent so it can be recovered afterwards. Fünfrocken describes a translation mechanism that inserts code for saving local variables just before moving the agent [2]. This has the disadvantage that the `go()` method cannot be called from arbitrary points outside the agent.

Our translation approach is to maintain a serializable version of the computation state at all times by letting the agent implement its own run-time stack. This increases the cost of regular computation as compared to Fünfrocken's approach, but it simplifies restarting the agent at the remote site.

3.1 Translation of Methods

For making the local state of a method serializable, we implement activation records of agent methods as objects. For each agent method, the preprocessor generates a class whose instances represent the activation records for this method.

An activation record class for a method is a subclass of the abstract class `Frame`:

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

Activation records must be cloneable for implementing recursion as explained below. The translated method code will be generated in method `run()`.

For example, given an agent class `C` with a method `foo` of the form

```
void foo(int x) throws AgletsException {
    int y = x + 1;
    go(new URL(dest));
    System.out.println(y);
}
```

(and ignoring exception handling and synchronization for simplicity) we might generate a class `Foo` of activation records for `foo` of the form

```
class Foo extends Frame {
    C This;
    int x;
    int y;
    int pc = 0;         // program counter

    Foo(C t) { this.This = t; }
    void setArgs(int x) { this.x = x; }
    void run() {
        if (pc == 0) { pc++;  y = x + 1; }
        if (pc == 1) { pc++;  go(new URL(This.dest));  This.run1();}
        if (pc == 2) { pc++;  System.out.println(y); }
    }
}
```

The parameter and the local variable of method `foo()` became fields of class `Foo`. In addition, we introduced a program counter field `pc` and a variable `This` for accessing fields in the agent object.

The method `run()` contains the original code of `foo()` together with code for incrementing the program counter and for allowing `run()` to resume computation after moving. Calls of agent methods are broken up into a call of the generated method followed by `This.run1()`, as explained below. For allowing the agent to be dispatched by code outside the agent class, the program counter increment and the following instruction must be performed atomically, which requires additional synchronization code.

For efficiency, the preprocessor could group multiple statements into a single statement and only allow the agent to be moved at certain strategic locations.

3.2 Translation of Agent Classes

An agent now must carry along its own run-time stack and method table. The generated agent class contains a `Frame` array as a method table and a `Stack` of `Frame`s as the run-time stack. When calling a method, the appropriate entry from the method table is cloned and put on the stack. After passing the arguments, the `run` method executes the body of the original method `foo` while updating the program counter.

Suppose we have an agent class `AgentImpl` of the form

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

    public AgentImpl() { /* initialization code */ }

    public void foo(int x) throws AgletsException { /* ... */ }
}
```

Since this class indirectly implements interface `Mobile`, the preprocessor translates it into the following code:

```
public class AgentImpl extends Aglet {
    int a;
```

```
        Frame[] vtable = { new Foo(this) };
        final int _foo = 0;
        Stack stack = new Stack();

        public void onCreation (Object init) {
            /* initialization code */
        }

        public void foo(int x) {
            Foo frame = (Foo) (vtable[_foo].clone());
            stack.push(frame);
            frame.setArgs(x);
        }

        public void run1() {
            Frame frame = (Frame) stack.peek();
            frame.run();
            stack.pop();
        }

        class Foo extends Frame {
            // as described above
        }
    }
```

The preprocessor eliminates interface `Mobile` and class `MobileObject` and lets the agent class extend class `Aglets`.

For implementing method dispatch, the agent includes a method table `vtable` of type `Frame[]`. The constant `_foo` is the index into the method table for method `foo`. The field `stack` implements the run-time stack.

The constructor of class `AgentImpl` is translated into the method `onCreation()`. Since Aglets only allows a single `Object` as argument of `onCreation()`, any original constructor arguments must be packaged in an array or vector by the preprocessor.

As described above, the original agent method `foo()` gets translated into a local class `Foo` of activation records. The method `foo()` in the generated code implements the calling sequence: it allocates an activation record on the stack, passes the arguments. The code for executing the method on the top of the stack and for popping the activation record in method `run1()` is shared between all methods. A client must first call `foo()` followed by a call to `run1()`.

For resuming execution after arriving at the destination, we must also generate a method `run()` inside class `AgentImpl`:

```
    public void run() {
        while (! stack.empty())
            run1();
    }
```

The Aglets environment will call this method after an agent is created and after it arrives at a remote host. If the user provided any code for `run()` in the strongly mobile version, it is translated like other methods into a class `Run`. In this case the generated method `run()` contains the code to set up the activation record for `Run` before the while loop.

3.3 Method Call

A constructor call in a client is translated to a request to the Aglet context to create the agent and to initialize it via `onCreation()`. Since Aglets does not allow methods to be called by a client, a method call has to be translated into a message that is sent to the agent.

For example, the constructor and method calls

```
    Agent a = new AgentImpl();
    a.foo(1);
```

are translated to

```
AgletProxy a = TheAgletContext().createAglet(getCodeBase(), "Agent", null);
a.sendMessage(new Message("foo", fooargs);
```

with appropriate code for packaging up the argument to `foo` as an array `fooargs`.

The generated agent class then must also contain a dispatch method `handleMessage()` that accepts the message send and calls `foo`. For details see the complete example below.

3.4 Return Values and Exceptions

Like parameter passing, return statements must be translated to pass the return value on the agent stack instead of on the run-time stack of the Java virtual machine. For a single-threaded system, for simplicity, the return value can be kept in a variable `rv` in the agent itself. A return statement

```
return 42;
```

is then translated to an assignment to this variable:

```
This.rv = new Integer(42);
return;
```

Similarly, exceptions must be implemented by passing them on the agent stack. The easiest implementation 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 `rv` and by setting a boolean variable `exn` to true to indicate that an exception was thrown. After a method call, the variable `exn` must be tested to check whether a method was thrown or whether a normal value was returned. A try statement is then translated into an if-then-else chain that skips any code after an exception occurred and tests the exception type with `instanceof` tests to find the appropriate handler code to execute.

4 Example: A Broadcast Agent

In this section we present the translation of a broadcast agent that takes a message and relays it to several hosts.

4.1 Strongly Mobile Interface

The user must define an interface `Agent` as follows:

```
public interface Agent extends Mobile {
    public String bcast(String message) throws AgletException;
}
```

4.2 Strongly Mobile Class

The mobile agent takes an array of hosts to visit as arguments to the constructor. Calling `bcast()` with a message as argument cause the agent to deliver the message to all the hosts in the array that are reachable, while keeping track of the traveled route in a string.

```
public class AgentImpl extends MobileObject implements Agent {
    String[] hosts;
    String home;
    String traveled_route;

    public AgentImpl(String[] h) throws AgletException {
        hosts = h;
        home = getContextInfo().getHostURL().toString();
    }
    public void bcast(String message) throws AgletException {
        ContextInfo c;
        String hostname;
        String route = "Traveled from " + home;
        for (int i = 0; i < hosts.length; i++) {
            try {
                go(new URL(hosts[i]));
                c = getContextInfo();
                hostname = c.getHostURL().toString();
                route = route + " to " + hostname;
                System.out.println("Message from " + home +
                                   " to " + hostname + ": " + message);
            }
            catch(Exception ex) {
                System.out.println("Exception: " + ex);
            }
        }
        traveled_route = route;
    }
}
```

4.3 Generated Weakly Mobile Class

The weakly mobile code generated by our translation mechanism would look as follows:

```
public class AgentImpl extends Aglet {
    // fields copied from the strongly mobile code
    String[] hosts;
    String home;
    String traveled_route;

    // fields for implementing the agent stack
    Frame[] vtable = { new Bcast(this) };
    final int _bcast = 0;
    java.util.Stack stack = new java.util.Stack();
    Object rv;
    boolean exn;

    // the callback method generated from the constructor
    public void onCreation (Object init) {
        String[] h = (String[]) ((Object[]) init)[0];
        hosts = h;
        home = getContextInfo().getHostURL().toString();
    }

    // the Aglets mechanism for allowing clients to call agent methods
    public boolean handleMessage(Message msg) {
        if (msg.sameKind("bcast")) {
            bcast(msg.getArg());
            run1();
            return true;
        }
        else
            return false;
    }

    // run the method on top of the agent stack
    public void run1() {
```

```
    Frame frame = (Frame) stack.peek();
    frame.run();
    stack.pop();
}

// continue execution of the methods on the stack after arrival
public void run() {
    while (! stack.empty())
        run1();
}

// the method for setting up the frame and passing the parameters for bcast
void bcast(Object init) {
    Bcast frame = (Bcast) (vtable[_bcast].clone());
    stack.push(frame);
    frame.setArgs(init);
}

// the frame class for method bcast
class Bcast extends Frame {
    AgentImpl This;
    String message;
    ContextInfo c;
    String hostname;
    String route;
    int i;
    int pc = 0;

    Bcast(AgentImpl t) { this.This = t; }

    void setArgs (Object init) {
        message = (String) ((Object[]) init)[0];
    }

    // the run method generated from the body of bcast
    void run() {
        if (pc == 0) { pc++; route = "Traveled from " + This.home; }
        if (pc == 1) { pc++; i = 0; }
        while (((pc==2) && (i < This.hosts.length)) || ((pc >= 3) && (pc <= 9))) {
            // try block
                if (pc == 2) { pc++; dispatch(new URL(This.hosts[i])); }
                if ((pc == 3) && This.exn) { pc = 7; } // handle exception
                if (pc == 3) { pc++; c = getContextInfo(); }
                if (pc == 4) { pc++; hostname = c.getHostURL().toString(); }
                if (pc == 5) { pc++; route = route + " to " + hostname;}
                if (pc == 6) {
                    pc == 9;
                    System.out.println("Message from " + This.home + " to "
                        + c.getHostURL().toString() + ": " + message);
                }
            // catch(Exception ex)
                if (((pc == 7) && (This.rv instanceof Exception)) || (pc == 8)) {
                    if (pc == 7) { pc++; ex = This.rv; This.exn = false; }
                    if (pc == 8) { pc++; System.out.println("Exception: " + ex); }
                }
            if (pc == 9) { pc = 2;  i++; }
        }  // end of while
        if (pc == 2) {  pc = 10;  This.traveled_route = route; }
        }
    }  // end of class Bcast
}  // end of class Agent
```

5 Synchronization Issues

So far, we have only described the translation mechanism for single-threaded control of the agent, either by a single thread within the agent or by a single thread outside of the agent. For a multithreaded system, we need some modifications: we must ensure that the program counter is incremented atomically with the following instruction; we must prevent two agents from dispatching one another at the same time; and we must translate user-specified synchronized blocks.

Critics of strongly mobile languages cite the difficulty of synchronization control as a motivating factor when arguing against the development of such languages. We concede that synchronization control is difficult to program, but offer an approach that we believe is no more taxing than programming for a weakly mobile system, or even for a traditional (non-mobile) system.

5.1 Protection of Agent Stacks

It is imperative that an agent cannot be dispatched by another thread between incrementing the program counter and executing the following statement. If the program counter increment and the following statement were not executed atomically, a thread could be dispatched after the program counter increment and incorrectly miss execution of the statement upon arrival. Since by definition this type of synchronization need not be maintained across VM boundaries, standard Java synchronization techniques are used. For a single-threaded agent, we simply synchronize on the agent object itself. For method calls, we only need to protect the call to set up the activation record. The actual execution of `run1()` does not need to be synchronized since by then a new activation record with its own `pc` will be on top of the stack:

```
    synchronized(This) { pc++; go(new URL(This.dest)); }  This.run1();
```

For preventing the agent from being dispatched between the program counter increment and the next instruction, the call of `realGo()` in `MobileObject.go()` must also be synchronized on the agent object.

If two agents try to dispatch one another, this synchronization code 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 stack. If similarly `b` would execute `a.go(dest)`, a deadlock would result. To prevent this, the call of `realGo()` is synchronized on the agent context instead of on the caller.

```
    public class MobileObject implements Mobile {
        // ...
        public void go(java.net.URL dest)
            throws java.io.IOException,
                   com.ibm.aglet.RequestRefusedException {
            synchronized(TheAgentContext) { synchronized(this) { realGo(dest); }}
        }
    }
```

The only time any thread synchronizes on two objects is in the call of `realGo()`, in which case the first synchronization is on the agent context. Deadlocks are, therefore, prevented.

This synchronization mechanism ensures that only one agent can migrate at a time. If two agents `a` and `b` try to dispatch one another, the first one, say `a`, will succeed. By the time `b` tries to dispatch `a`, `a` is already on a different host. The call to `a.go()` will, therefore, throw an exception that must be handled by `b`.

This synchronization mechanism could be extended to multithreaded agents. In this case, each thread would have its own execution stack. Instead of on the agent, we would have to synchronize on the thread. In effect, the threads would become agents. When migrating an agent, we would have to ensure that all thread are in a safe state. This can be done by maintaining a counting semaphore whose count indicates how many thread are currently running.

5.2 Synchronization Blocks

The Java semantics for synchronized blocks or synchronized methods is that the lock is released when the agent is migrated. When users use synchronization in an agent to protect the agents internal data structure, this protection, however, 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.

Though it may appear under different guises, there is really only one synchronization primitive offered through Java: the synchronized code block. Before entering a synchronized portion of code, which may span an entire method, a thread must acquire a lock on a particular object or class. Once the lock is acquired, the thread enters the protected region, executes the code block, and finally releasing the lock. In this manner mutual exclusion is maintained, and in some cases atomicity. Whether a thread exits a synchronized section normally or after a call to `go()`, the object lock is released. For weakly mobile languages this is a non-issue since code never executes beyond the call to `go()`. When a call to `go()` appears at the end of a synchronized code block, there is no distinction between exiting a thread normally or after dispatch.

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. Consider the following example:

```
public class AgentImpl() extends MobileObject implements Agent{
    int i = 0;

    public AgentImpl() { }

    public void synchronized foo(URL dest) {
        i++;
        go(dest);
        if (i == 1) System.out.println("Synchronized!");
    }
}
```

As `foo()` is a synchronized method, when it is invoked the calling thread must first acquire a lock on an instance of `AgentImpl`. Using traditional Java synchronization semantics all calls to `foo()` should result in the message `"Synchronized!"` being displayed to `stdout` once the thread reaches the new host. This need not be the case, however. If thread A calls `foo()` and reaches the call to `go()`, it immediately loses the lock upon dispatch. If a different thread on the destination site should call `foo()` before the original thread continues execution, the original thread will have the variable `i` equal to 2 when the thread reaches the `println()` statement.

The difficulties stem from the fact that object locks are not stored within the object during serialization, but rather hidden within the virtual machine. To tackle this problem we introduce serializable locks. Consider the following class `MobileMutex` designed to maintain synchronization locks across VM boundaries:

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

    public MobileMutex() {}

    public synchronized void lock() {
        if(!locked)
            locked = true;
        else {
            while(true) {
                try {
                    wait();
                    locked = true;
```

```
                        break;
                    }
                    catch(InterruptedException ex) { }
                }
            }
        }

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

Client programmers using this extension continue to use the standard `synchronized` keyword to enforce synchronization constraints. To these developers it appears that object locks are in fact preserved across dispatch.

The transparency is achieved by introducing serializable locks in place of the standard Java object locks. During the translation phase an object of type `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 via a call to `lock()` 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 preserves synchronization semantics while the agent is on the move.

6 Performance

We have performed some preliminary measurements to estimate the cost of the described translation mechanism. We compared hand-translated strongly mobile agents to equivalent weakly mobile agents in Aglets.

For an agent that goes to a remote host, performs some simple computation on large arrays in a tight loop, and returns to the originating host, the tests and increments of the program counter result in on overhead of 45%. With synchronization, the overhead was 80%.

For an agent that uses a recursive method to count down from 100 to 0, the strongly mobile code was slower by a factor of 6.1. Repeatedly calling an empty method resulted in a factor of 173 (!), but this appears to be due to optimizations in the virtual machine that we do not perform in our translation mechanism or due to additional garbage collections.

For these measurements, we did not perform any optimizations of the weakly mobile code resulting from our translation. The cost for incrementing the program counter and for synchronization can be reduced by combining multiple statements into one synchronized block with synchronized blocks ending at method calls:

```
    synchronized(This) {
        pc++;
        x = 31;
        y = 11;
        foo (x + y);
    }
    This.run1();
```

As long as there is no non-terminating computation within a synchronized block, this does not change the semantics of strong mobility, it only results in the dispatch being delayed.

Method calls can be optimized through tail-call optimization. Since Java code typically is not highly recursive, we do not expect the average performance penalty to be worse than a factor of two or three, possibly less than that.

7 Conclusions

We have argued that strong mobility is important for developing mobile code in a modular fashion and, therefore, for making this code reliable. We have outlined a translation scheme that translates strongly mobile code into weakly mobile code. As the target language for the preprocessor we use Java with IBM's Aglets library. The API for the strongly mobile code and the translation mechanism are designed to give programmers full flexibility in dealing with any synchronization problems through exception handling.

References

[1] 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.

[2] 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.

[3] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, Massachusetts, 2nd edition, 2000.

[4] 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. To be published in Springer Lecture Notes in Computer Science.

[5] Joseph Kiniry and Daniel Zimmerman. A hands-on look at Java mobile agents. *IEEE Internet Computing*, 1(4):68–77, August 1997.

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

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

[8] 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.

[9] 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.

[10] 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. To be published in Springer Lecture Notes in Computer Science.