Efficient Support for Mobile Computations on the Internet

Sarah Monisha Pulimood and Boumediene Belkhouche
Tulane University, New Orleans, LA, USA

Abstract: This paper discusses the necessary framework for efficient and secure support for mobile computations that can commence execution at one site, move to a different site, and continue execution from the point where they left off. We also discuss the design and implementation of a Mobile Computation System (MCS) that includes an object oriented Mobile Computation Language (MCL). Portability of computations is ensured through a Mobile Computation Manager (MCM) and a RISC-based virtual machine (RVM). Since the MCL, the MCM and the RVM are all specifically designed and implemented for mobile computations, the system can efficiently deal with the requirements and effects of mobility on a heterogeneous network, like the Internet.

Keywords: Mobile Computational Model, Internet Programming Language

1. Introduction
Internet programming is a new paradigm that envisages being able to exploit mobile computations to share resources on a large heterogeneous network. Most programming languages in use today were designed originally for computation on individual machines or distributed systems. Merely extending these languages to support mobile computation is insufficient and results in inefficient languages. Any model that can fully support the Internet Programming paradigm has to be specially designed for mobile computation and should be able to deal efficiently with its effects. We discuss the design and implementation of our computational model that fits these requirements.

2. Background
Languages, like Java [1] and Limbo [2], [3], provide code mobility or ‘weak mobility’ [4], where the source code is pre-processed to obtain intermediate code. This code is sent to (or called by) a host in its entirety for execution. The closure or state of the execution is not captured and moved, and is therefore lost as a result of the transfer. Efforts have been made to modify or extend the Java development environment in a variety of ways [5], [6], [7] to provide support for capturing and restoring the closure or state of execution of mobile agents. Aglets [8], [9], Concordia [10], Odyssey [9], Voyager [9], etc. are examples of systems resulting from such efforts. Autonomous agents, like those generated by Agent Tcl [11], carry the entire code and closure of the execution when they move. Cardelli [12] aims to describe all aspects of mobility within a single framework that encompasses mobile agents, the ambients where agents interact, and the mobility of the ambients themselves. (An ambient is a bounded place where computations take place.) Obliq [13], [14] and Emerald [15] permit closure mobility and distribution of smaller units of code. However these were not intended to run on large long-haul networks.

Most approaches that provide for closure mobility build layers on top of, or adapt, existing languages or systems that were not originally designed for this purpose. Such approaches inherit security and other weaknesses of the base systems. Additional layers also degrade the performance of the system. A model that supports mobile computations should be able to guarantee that no application in that model will be malicious in nature and that no access violations will be made [16]. Language safety helps guarantee integrity of run-time structures and enables garbage-collection.

Our approach has been to keep the issues of closure mobility, heterogeneity, and security as the focal points of our design and implementation. We implement safety and security measures within the context of the programming language, mobile computation manager, and virtual machine to prevent malicious computations that may tie up resources, like memory, or form security breaches.

3. Modeling the Mobile Computation Environment

Figure 1 shows the architecture of the Mobile Computation System (MCS). A program written in the high level mobile computation language (MCL), is converted into a mobile computation that is sent to a host with a resident Mobile Computation Manager (MCM). The MCM manages all mobile computations that visit the site, while the virtual machine executes the computation. If a resource that is unavailable at the current site is required, or an explicit ‘move’ is encountered, the current state of execution is captured and a new mobile computation is generated. This new computation is then sent to the specified (or next) host.

3.1 The Language

We have designed the object oriented programming language, MCL, specifically for writing mobile computations to be executed within our system. A prototype for this language has been implemented and mobility of the computations among the Sun SPARC workstations (running the Solaris operating system) in our department has been tested. The unique requirements of mobility (including transfer of current state of execution), security, and communication and information sharing among computations were primary concerns in the design process. While the computational power of the language is comparable to other popular object oriented programming languages, we provide, additionally, the mobility construct ‘move’.

The syntax of this construct is:

<stmt> → move <location> [ <stmt_list> ]
move [ <stmt_list> ]

Safety within the language is ensured by implementing strong typing. Attributes of classes are inherently ‘private’ while methods may be ‘private’ or ‘public’. These restrictions help reduce the risk of unauthorized accesses to class information. Global variables are not allowed so that problems with tracking references to them and maintaining data integrity can be avoided. Instead all data must be passed to methods through the parameters. Pointers are disallowed to prevent unauthorized accesses to memory locations.

The initial compilation process converts a program in this language into a mobile computation which is encrypted and sent out over the network for execution. Information about the route the computation will follow, the tasks it needs to perform, identification of the initiator of the computation, and access privileges at each site or ‘credentials’ are included in the computation.

Below, we present a sample program written in this language to illustrate the features supporting mobility. The piece of code enclosed in the curly braces of the ‘move’ command is converted to an encrypted mobile computation and transferred to the next site in the ‘route’ list unless a site has been specified as shown in the example.

```java
header
route
"pegasus.eecs.tulane.edu" rw ("task1" "resource1" done; ...) pending; ...
endroute
id "pegasus.eecs.tulane.edu"; "Moni"; endid
endheader

class Money {
   new int stoQrt; ...
   public void initialize () {... }
   public int makeChange (int depQ, int depD, int depN) {
      ...
      if (change < 0) retval = 0;
      else {
         move ("juno.eecs.tulane.edu");
         ...
      }
   }
}
class Vend {
   new Money Change; ...
   public int main () {
      new int depQrt; new int depDime;
      ...
      rets = Change.makeChange (depQrt, depDime, depNick);
   }
}
```

3.2 The Computation Host

One of the primary objectives of our research is to be able to deal efficiently with the inherent heterogeneity of the Internet. Since there are several different types of operating systems and architectures available on the Internet, each accepting host must have some mechanism to recognize a computation and execute the code. The MCM and the RVM resident on every host in the subnet are responsible for ensuring portability of the mobile computations in our model.

Conceptually the RVM sits on top of the operating system with the MCM acting as the interface between it and the mobile computation. In reality, however, the RVM is very lightweight and inter-
acts directly with the architecture, while the MCM is integrated with the operating system.

3.3 The Mobile Computation Manager
Figure 2 shows the main components of the Mobile Computation Manager (MCM) and the interactions between them. MCM may be viewed as a daemon process that lies dormant until it is informed by the Event Monitor that a new computation has arrived or that there is a change in the current state of the mobile computation environment. The Activity Manager initiates and oversees the various tasks performed by the MCM. The Security Administrator receives the mobile computation, decrypts it, and verifies its credentials. Insecure or invalid computations are rejected and queued for transfer to another host (preferably the initiating one). Valid computations are queued for further action by the Activity Manager. Before a mobile computation is transferred to another host, it is encrypted by the Security Administrator. The Queue Administrator manages the queues and informs the Activity Manager when a computation arrives in one of its queues, e.g. the Execution Queue, Load Queue, Unload Queue, Move Queue etc. The structures required for execution of the mobile computation are generated and loaded by the Loader component in preparation for execution. When a ‘move’ operation is encountered, the RVM halts execution. The current state / closure of the execution is captured by the Unloader in preparation for the transfer of the computation to another host. A mobile computation is generated for the code specified by the ‘move’ instruction. In the preliminary implementation, the closure for the entire application is included in the computation. We are now evaluating what context will be required for the computation to be able to execute completely. The management of references to information not included in the computation is also being studied. The Communication Administrator monitors the network to ensure that the route specified in the mobile computation is still valid. Any changes to the network, e.g. the next host on the route is currently unavailable, may require modifications to the route.

3.4 The RISC Virtual Machine (RVM)
The RVM is a very thin layer residing as close to the machine architecture while retaining generality.

3.4.1 Computation Execution
The mobile computation is loaded into data structures by the Loader component of the MCM. An object is instantiated only when it is used for the first time. The Object List keeps track of each object instantiated during execution. The Permanent Activation Record (PermAR) maintains information about the attributes of each object instantiated. The Temporary Activation Record (TempAR) maintains information about the local variables and parameters for a method. Figure 3 represents the execution structures of the RVM.

The Program Counter Stack (PCS) keeps track of the current instruction to be executed. When a method is called a new program counter item is created and pushed onto the program counter stack. The resumption point (RP) i.e. the next instruction to be executed when the control returns to the calling method is also stored. The method completes execution when it encounters the ‘stop method’ instruction. All relevant items are popped from the respective stacks and the control returns to the new ‘top of stack’ of the PCS, which is the calling function.

If a ‘move’ operation is encountered before execution is completed, the state of each of these structures is captured and included as part of the mobile computation. On receipt by the new host, these structures are once again loaded into the execution area by the Loader component of that host’s MCM.
3.4.2 The Instruction Set

We compare the instruction set of our RVM to that of a Reduced Instruction Set Computer (RISC) machine. This is in contrast to the instruction sets of other virtual machines, like the Java Virtual Machine (JVM), which can be compared to those of Complex Instruction Set Computer (CISC) machines.

The JVM has distinct bytecodes for operations on different data types. Each instruction has a variable size with a varying number of operands. Some information required for the execution of the instruction is implicitly specified in the opcode itself. Other information is gained by popping the current stack. JVM instructions were designed for maximum functionality and minimal storage access, however, being more complex, they each can take several processor cycles to execute.

Internet-based applications can include various types of computations, possibly even large scientific ones, that can be broken down into smaller computations to be performed at various sites depending on the resources available. In such an environment, especially a mobile one, where any type of operation may be performed, we cannot make assumptions about which operations may or may not be performed on specific data types without adversely affecting the generality of the system being offered. Our aim is to provide full functionality while maintaining generality. We incorporate features of instruction sets of RISC machines to build our RVM. Hence the instruction set of the RVM has a small number of simple instructions, each of which has a fixed size and format.

RVM Instruction

Each RVM instruction is of the form

\(<index> <opcode> <operand>\)

The \(<index>\) is the index of the opcode of the instruction in the array that contains the bytes of code for this method.

The \(<opcode>\) is the instruction’s opcode.

The instruction has exactly one \(<operand>\).

Mapping JVM instructions to RVM instructions

Many instructions in the JVM Instruction Set implicitly specify the data type of the operands for the operation. The JVM also uses a pool of constants and local variables in an effort to speed up certain operations. Thus the JVM not only has specialized instructions to implicitly specify the type of operands, but it also has instructions that implicitly specify the address locations to be accessed within the constant pool. In order to keep the size of the Instruction Set small, the JVM does not provide corresponding operations for every type. The assumption is that operations on certain data types are more likely to occur than others. For example the ‘add’ instruction indicates that the operation to be performed is ‘add’ and that the operands will be of type integer. However other first class data types do not all have the same level of support. This specialization approach results in a loss of generality.

The aim of the RVM is to provide simplicity, generality, and orthogonality. To this end we carefully evaluated each instruction in the JVM Instruction Set and eliminated several specialized instructions, replacing them with simpler more general RVM instructions. The RVM does not use a pool of constants. Data types are explicitly specified in the operand of the instruction when necessary. As a result the size of the RVM Instruction Set is about one-fifth the size of the JVM Instruction Set.

For example, JVM instructions [17] of the form

\(<t>load_{<n>}\) and \(<t>load index\), (where \(<t>\) indicates the data type, e.g ‘i’ for integer) are replaced uniformly by an RVM instruction of the form

\(load <operand>\) (where \(<operand>\) is a constant value or a reference to a variable).

As a general rule we can replace JVM instructions of the form

\(<t>instruction_{<n>}\) and \(<t>instruction\) by

\(instruction\) operand (where \(<operand>\) may be the type, the address for a variable, or a constant value).

We thus obviate the necessity of constant pool resolution in addition to reducing the number of instructions, without loss of functionality [18].

3.4.3 Evaluating the advantages

Straightforward and small Instruction Set

The small size of the RVM Instruction Set results in a more lightweight virtual machine. A lighter virtual machine can be expected to execute computations faster and more efficiently, especially since the instructions can be loaded, decoded and executed faster. Such a machine will also be easier to maintain, less of a burden on the host’s resources, and more efficient than a heavier machine. The simplicity and size of the instruction set does not detract from the features and capabilities of the language.

Code Execution

Since a JVM instruction has variable length, the time taken to load it into the processor can vary. We denote this time by ‘x’ where x > 1. The time taken to decode and locate a JVM instruction in the Instruction Set is denoted by ‘y’. An RVM instruction can be loaded into the processor in one cycle since it has a fixed length. The time taken to decode and locate an instruction in the RVM Instruction Set is denoted by ‘z’, and

\[ z \equiv y \gg 1 \]

The time taken for constant pool resolution is denoted by ‘w’ where

\[ w \gg 1 \]

For data types for which the JVM has specialized support, the RVM execution may take more processor cycles and memory accesses, than the
corresponding JVM instructions. However, the RVM execution is more efficient for operations on data types for which the JVM does not have special or direct support. The performance of the RVM for an instruction is the same regardless of the data type. In contrast, the JVM performs efficiently for data types that are directly supported for the instruction, while its performance degrades for the data types that are supported indirectly or not at all. This is illustrated by the examples in Table 1 which provides a comparison of the execution times for some corresponding instructions.

**Closure mobility**
The JVM was not originally intended for mobile applications and therefore does not support capture of the closure of a computation. An ‘applet’ must be executed in its entirety at any host. On the other hand, the RVM has been designed specifically for mobile computations. Capturing the closure of a computation at any point is inherent in the system.

**Translation**
It would be most efficient to directly compile a computation into RVM code. However, the use of Java is fairly widespread and there is a considerable amount of code already compiled into Java bytecodes. By translating JVM code to RVM code, the benefits of the RVM can be harnessed for existing code as well. The translation from JVM code to RVM code is trivial and needs to be performed just once possibly at the initiating host before the mobile computation is sent out over the Internet. The translation may also be performed at the first computing host the computation visits. Since the correspondence between the JVM and RVM instructions is straightforward, we can show that any JVM code that is reliable and guaranteed to work remains so even when converted to RVM code.

**Size of Code**
In the process of translating code from JVM to RVM, it’s size will change due to the different formats of instructions. For applications that the JVM specifically supports, the size of each JVM instruction will be compact. The JVM→RVM translator may need to replace a single JVM instruction by a sequence of 2 or more RVM instructions. Hence, the size of the RVM code may be larger than that of the corresponding JVM code for certain applications. However, for applications that have several operations not directly supported by the JVM, the number of instructions required to perform these operations will be more than that required by the RVM. Additionally, each of the JVM instructions could have one or more operands, thereby increasing its size. The simplicity of the RVM instruction coupled with the reduced time to execute each instruction offset increases due to size of code and memory accesses.

**Orthogonality**
We make no assumptions about the type of applications that our mobile computations will be used for. Hence the RVM has an orthogonal instruction set providing equal support for all first class data types. The addition of a new first class data type in the system can be handled easily by the RVM without adding new instructions to the Instruction Set. For example the ‘add’ operation can still use the form `nadd <type>` where `<type>` is the new data type. The RVM will have to be extended only with the implementation of the various operations for the new data type. However the JVM would have to add new instructions to the Instruction Set, e.g. `nadd`, for the new data type, in addition to extending the implementation of the various operations for the new data type. The alternative would be to perform the addition operation in a very roundabout and inefficient manner. Considering that the JVM set already has 255 instructions (requiring one byte to represent the opcode), any additions would mean that an additional byte would be required to represent the opcode. This would further inflate the code.

**Optimization and compilers**
Having a reduced fixed length Instruction Set makes it easier to optimize the RVM code. One way of optimizing code is to find patterns within the code.

---

**Table 1: Comparing Execution Times for the JVM and RVM**

<table>
<thead>
<tr>
<th>Operation</th>
<th>JVM OC</th>
<th>RVM OC</th>
<th>JVM cycles</th>
<th>RVM cycles</th>
<th>JVM vs. RVM</th>
</tr>
</thead>
<tbody>
<tr>
<td>load data onto stack</td>
<td>aload</td>
<td>load data</td>
<td>4+x+y≥5+x</td>
<td>4+z≥5</td>
<td>JVM &gt; RVM</td>
</tr>
<tr>
<td>load data onto stack</td>
<td>aload_&lt;n&gt;</td>
<td>load data</td>
<td>4+w+x+y≥5+w+x</td>
<td>4+z≥5</td>
<td>JVM &gt; RVM</td>
</tr>
<tr>
<td>add integers</td>
<td>iadd</td>
<td>add int</td>
<td>4+x+y≥5+x</td>
<td>4 + z≥5</td>
<td>JVM &gt; RVM</td>
</tr>
<tr>
<td>add floats</td>
<td>fadd</td>
<td>add float</td>
<td>4+x+y≥5+x</td>
<td>4 + z≥5</td>
<td>JVM &gt; RVM</td>
</tr>
<tr>
<td>integer comparison</td>
<td>if_icmplt</td>
<td>cmpint</td>
<td>5+x+y≥6+x</td>
<td>8 + 2z≥10</td>
<td>JVM &lt; RVM</td>
</tr>
<tr>
<td>double comparison</td>
<td>dcmpl</td>
<td>cmplt</td>
<td>7+2x+2y ≥9+2x</td>
<td>8 + 2z≥10</td>
<td>JVM &gt; RVM</td>
</tr>
<tr>
<td>char comparison</td>
<td>internal_&lt;type&gt;</td>
<td>cmpint</td>
<td>9+2w+x+y ≥10+2w+x</td>
<td>8 + 2z≥10</td>
<td>JVM &gt; RVM</td>
</tr>
</tbody>
</table>

---
and generate macros for them [19]. These macros can be called every time that piece of code is to be executed. This reduces the size of the code to be transported and is advantageous for the mobile computation. Finding patterns in code that has the same opcode for all variations of an instruction is easier than finding patterns in code that has a different opcode for each variation of the instruction.

Printing compilers for a reduced Instruction Set is also more straightforward.

4. Conclusions
In this paper we discussed our MCS that provides the necessary framework for mobile computations over a heterogeneous network like the Internet. The MCL, MCM and RVM are all specifically designed and implemented for mobile computations and can therefore efficiently deal with the requirements and effects of mobility. The comparative simplicity of the Instruction Set for the virtual machine does not detract from the generality of the language, its features, or functionality. On the other hand, several benefits are accrued, including simplification of assessment of security risks and program verifiability.

5. References