Journal of Universal Computer Science, vol. 12, no. 7 (2006), 885-902
submitted: 28/1/06, accepted: 6/4/06, appeared: 28/7/06 © *J.UCS*

An Object Model for Interoperable Systems

Alcides Calsavara
(Pontifícia Universidade Católica do Paraná, Brazil
alcides@ppgia.pucpr.br)

Aron Borges
(Pontifícia Universidade Católica do Paraná, Brazil
aron@ppgia.pucpr.br)

Leonardo Nunes
(Sumersoft Tecnologia, Brazil
leonardo@sumersoft.com)

Diogo Variani
(Global Village Telecom, Brazil
diogo.variani@gvt.com.br)

Carlos Kolb
(Companhia Paranaense de Energia, Brazil
kolb@copel.com)

Abstract: Most modern computer applications should run on heterogeneous platforms and, moreover, objects and respective code should be easily interchangeable between distinct platforms at runtime. This paper describes a runtime platform based on distributed and cooperating virtual machines named Virtuosi. A unified object model permits easy inter-operation between applications written on different languages. All applications must be compiled to a standard runtime code format so they all can run on any platform where an implementation of the virtual machine exists. A novel code format which is entirely based on instances of the classes that define the object model itself is employed. A proper programming language has been defined, a corresponding compiler implemented, a virtual machine that includes a class loader, a code interpreter, a single-threaded execution control and a distributed object store implemented and tested through example applications.
Key Words: object model, interoperability, virtual machine
Category: C.2.4, D.1.5

1 Introduction

Most modern computer applications should run on heterogeneous platforms and, moreover, their components, including objects and respective code, should be easily interchangeable at runtime. That is even more critical for peer-to-peer and ad hoc networks where any kind of hardware platforms and operating systems may inter-operate. Another important requirement is the absence of server-

centered services to provide for scalability, permit a flexible network configuration and avoid single points of failure. A number of current research projects are devoted to adapt standard runtime platforms, such as CORBA-based platforms [Soley and Kent, 1995], DCE-based platform [DCE, 1992], and the Java platform, to those new requirements. Also, several academic projects envisage a high degree of portability and interoperability, such as the Virtual Virtual Machine project [Folliot et al., 1997] [Baillarguet and Piumarta, 1999].

This paper proposes an object model and the architecture of a runtime platform based on distributed and cooperating virtual machines named Virtuosi. Differently from traditional virtual machines, such as the Java virtual machine, where distribution mechanisms are added by means of library classes, the proposed architecture intrinsicly gives distribution transparency to applications that conform to the proposed object model. In its essence, Virtuosi is similar to the Microsoft .NET platform, where all applications should be compiled to a standard runtime code format so they all can run on any platform where an implementation of the virtual machine exists. Yet, a unified object model permits easy inter-operation between applications written on different languages. Nevertheless, Virtuosi uses a novel code format which is entirely based on objects that are instances of the classes that define the object model itself. That permits to build and maintain references to classes and methods across machines, thus making it easy to implement a code mobility mechanism. Another important difference is the way Virtuosi stores objects and maintains references to them: each object is indexed within the virtual machine where it is stored, so any reference to an object is always indirect, that is, through its corresponding index; when an object happens to be remote, a proxy entry is used to refer to the corresponding remote machine and object index. Such feature permits a simple implementation of a mechanism for object mobility. Thus, only each virtual machine needs a unique identification in order to allow any object to be accessed anywhere, like in a distributed shared memory where there is a single address space.

A prototype implementation of Virtuosi has been developed in order to assess the proposed object model, its code format performance and its ability to deal with distribution and heterogeneity. For that, a proper programming language has been defined, a corresponding compiler implemented, a virtual machine that includes a class loader, a code interpreter, a multi-threaded execution control and a distributed object store implemented. Moreover, a mechanism for object and code mobility and a mechanism for transparent remote method invocation were defined, implemented and tested through example applications.

The remainder of this paper is structured as follows: [Section 2] gives an overview of the Virtuosi runtime platform; [Section 3] describes the proposed core object model; [Section 4] describes – through examples – a programming language fully compatible with the proposed object model; [Section 5] describes

a distributed virtual machine architecture that fully supports the proposed object model and gives transparency to distribution and platform heterogeneity; [Section 6] discusses the implementation status of Virtuosi; and, finally, [Section 7] gives some conclusions about the developed research work and discusses some future work.

2 Runtime Platform Overview

The Virtuosi runtime platform, initially defined in [Calsavara, 2000], is composed of a collection of communicating virtual machines. Each virtual machine is a user-level process that emulates a real-world computer, including its hardware components and corresponding operating system. Thus, each virtual machine is able to host any typical software systems that store and process data and, as well, communicate with peripherals. Virtual machines are grouped in collections where each virtual machine can be unambiguously addressed and can exchange messages with any other in the collection; dynamically, objects are created and they interact with each other through method invocation. That allows a software system running on a certain machine to communicate with a software system running on a different machine, i.e., a collection of communicating virtual machines is a runtime platform for distributed software systems. In fact, this runtime platform can be seen as a *middle-ware*, similar to systems based on the CORBA Standard [Soley and Kent, 1995], since a distributed software system can run atop a heterogeneous computer network. The difference, in this case, is that, in CORBA, distribution is accomplished by standard services which applications have to aware and make the necessary operation calls at the right moment, while, in Virtuosi, such services and calls are done automatically. Also, Virtuosi can be compared with object systems such as Oberon [Wirth and Gutknecht, 1992] and Amoeba [Mullender et al., 1990], where the object-oriented programming paradigm is fully supported through a programming languague, corresponding compiler and operating system. Now, the difference is that these are real operating systems, while Virtuosi is based on virtual machines which can run atop any operating systems.

2.1 Portability and Mobility

A virtual machine sits between applications and the actual operating system; applications interact with the virtual machine which, in turn, interacts with the operating system. As a consequence, there must be a specific implementation of the virtual machine for each operating system. Another consequence is that a software system that runs on a specific virtual machine implementation runs on any other. In other words, Virtuosi applications are portable: they run on

heterogeneous computers, as long as there is proper implementation of the virtual machine for each different platform.

2.2 Controlled Execution

Because a virtual machine is a software system that controls the execution of other software systems, it can fully assist in debugging applications; a virtual machine can keep very precise data about execution context, thus providing programmers with more accurate information when some bug happens. This is an essential feature to improve productivity because programmers can use debugging to better understand software systems behavior. The solution found in Virtuosi for the purpose of having full application semantics at runtime is to represent and store program code in the form of a *program tree*: a graph of objects that represents all elements of a given source code, including their relationships [Kistler and Franz, 1996, Franz and Kistler, 1997]. Thus, the virtual machine loads and interprets program trees which represent classes. Many examples of program trees and a detailed discussion on how they are loaded and interpreted by a virtual machine can be found in [Kolb, 2004]. Since there is a direct mapping between a program tree and a source code, the rules for building a program tree are the same for writing an object-oriented program. Such rules are established by an object model formalized by means of a class diagram, as discussed in [Section 3]; the objects of a program tree are instances of the those classes. For each user class, our compiler creates a graph of objects, a *program tree*. Thus, for a given application class, there will be an object to represent the whole class, so there will be an object to represent each method that belongs to the class, an object to represent each formal parameter of a method, an object to represent a method call, and so on.

3 Object Model

The Virtuosi object model formalizes basic concepts of the object-oriented programming paradigm which are supported by the virtual machine. It is a core object model because, except for *actions* and *data blocks*, described below, it just comprises concepts which are rigourously object oriented and normally present in most traditional programming languages, such as C++, Java, Smalltalk and Eiffel. The object model is formalized through a class diagram composed of about 50 classes, with many associations between them. A complete description of the object model, including a class diagram in UML [Rumbaugh et al., 1997] can be found in [Kolb, 2004]. Here, its main classes and associations are discussed.

3.1 Class

Every application class is represented by an instance of Class. Each application class has a unique name and defines a scope that may contain:

datablock references References to instances of Datablock (described below).

index references References to instances of Index (described below).

object references References to instances of application classes. Each object reference corresponds to an attribute which represents either object association or object composition.

constructors Operations which are invoked to construct new instances of the class.

methods Operations which are invoked for a certain instance of the class to either read or change object state.

actions Operations which are invoked for a certain instance of the class specifically to test object state, according to some criteria.

3.2 Literal

A literal is a constant string corresponding to the textual representation of some primitive value, such as an integer number, a real number, a character, a string, a boolean value, a bit value, and so on. Literals are necessary for programmers to express data directly on the source code and for the virtual machine to exchange data with the real world through computer peripherals.

3.3 Datablock and Index

A datablock is an array of bits. The semantics of a sequence of bits is given by the implementation of its container class, i.e., by the methods that access the bits. For example, a sequence of bits may represent an integer value, so its container class should have methods to set the bits to store a certain integer value, to change the bits to represent the result value of an arithmetic operation, and so on. Each bit of a datablock may be addressed in order to call write (either set or clear) and read operations. A bit of a datablock must be addressed through an index associated to the datablock. An index is a sequential iterator that defines operations to set its value to a certain address of the associated datablock and to change its value forwards as well as backwards. If an attempt to change the value of an index beyond the limits of the associated datablock is made, an exception is raised by the virtual machine. There are no other primitive data types in

Virtuosi. Standard primitive types, such as integer, real, char, string and boolean are available through a separate class library. All those types and any other a user may wish to define are built on top of datablock and index. Although such design brings some performance penalties, all data is independent of machine representation and, moreover, any data is uniformly handled as objects, so they can migrate (move) and can be remotely accessed in a transparent way.

3.4 Constructor

A constructor of an application class is an operation called to create a new instance of the class. Each class may have a set of constructors. Each constructor of a class is uniquely identified through its signature, which is composed of its name and an ordered list of formal parameter types (the name of a constructor does not need to be identical to the name of the container class).

3.5 Method

A method of an application class is an operation called to perform a sequence of statements on an existing instance of the class. Such statements have full access to object state, so they can either read or modify it. A method may return an object reference as a result. Each method of a class is uniquely identified through its signature, similar to a constructor.

3.6 Action

An action of an application class is an operation called to perform a test on an existing instance of the class. They correspond to methods that return a boolean value in traditional programming languages. Here, the real difference is that calling an action is restricted to the context of a conditional branch expression, i.e., an expression which must be evaluated to decide whether a statement should be executed or skipped. A typical use of actions is in `if` statements, where an expression must be evaluated to decide whether or not a certain statement should be executed. Thus, the result of an action must be either *execute* or *skip*, and such result is not directly handled by user programs; instead, it is handled by the virtual machine to select the next statement to execute. An action is also implemented by a sequence of statements which have full access to object state, but they are restricted to perform read operations. The execution of an action finishes when either a `execute` or a `skip` statement is performed.

3.7 Formal Parameter

A parameter of a class operation (constructor, method or action) must be either an object reference or a literal; datablock and index references are not allowed as parameters because they must be always encapsulated by an object.

3.8 Exportation List

Every class operation has an associated exportation list: a list of classes (besides the container class itself) which have permission to call that method. This solution, adapted from the Eiffel programming language, is more general than specific qualifiers, such as *private* and *public*, normally found in traditional languages.

3.9 Local Variable

Every class operation may define a set of local variables. A local variable can be an object reference, a datablock reference or an index reference. Object references can be passed as parameters in operation calls, while datablock and index references have their use restricted within the scope of their definition.

3.10 Statement

Every class operation has an associated sequence of statements. The kinds of statements are the following:

variable declaration A local variable is declared within a class operation.

datablock creation A datablock of a given length is created (which implies memory allocation by the virtual machine) and its address is assigned to an existing datablock reference.

index creation An index is created, its address assigned to an existing index reference and it is associated to an existing datablock.

constructor invocation A constructor of a given class is called and, as a consequence, a new object (instance of that class) is created (which implies memory allocation by the virtual machine) and its address is assigned to an existing object reference.

method invocation A method of a given class is called for an existing instance of that class. If the method returns an object reference, then such a reference is assigned to an existing object reference. Methods can be invoked in two different modes: synchronous or asynchronous, like in the SR language [Andrews and Olsson, 1993]. Differently from typical programming languages, such mode is not fixed per method: the caller must choose the mode it desires. Thus, a certain method can be called synchronously at one moment, while asynchronously later.

reference bind An object, datablock or index reference is assigned to another. (There is no copy of contents; only an address is copied.)

return It may only occur within methods which have a return class specified. It returns a copy of an object reference of the specified class to the caller of the method.

execute It may only occur within actions. When performed finishes the action and returns a signal to the caller to indicate that the test succeeded.

skip It may only occur within actions. When performed finishes the action and returns a signal to the caller to indicate that the test failed.

unconditional branch A given statement is immediately executed.

conditional branch An expression containing only testable elements (action invocation, comparison of two object references, comparison of two datablock references and comparison of two index references) structured as a logical expression (with *and* and *or* operators) is evaluated. If the final result is *execute* then a given statement is executed, otherwise it is skipped.

4 Programming Language

We have designed a new programming language named Aram for the purposes of validating the object model and the distributed virtual machine architecture. Although there were good candidates for this purpose, such as Smalltalk, Eiffel, C++, Java and Common Lisp, the proposed object model merges concepts from several programming languages and, as a consequence, none of them alone fully supports the object model and, conversely, they all have features which are not supported the object model. Since our initial purpose is to validate the object model and the distributed virtual machine architecture at the same time, for a while, we found it simpler to define a new language, rather than adapting an existing one, which we left as a future work. It should be noted, though, that Aram is very similar to Java and, in fact, it can be seen as a first step to adapt Java to Virtuosi. A detailed description of Aram is out of the scope of this paper; its use is exemplified in the sequel.

4.1 Example 1

The code below shows two application classes: *Agent* and *Boss*. The code at line 3 says that every instance of *Agent* references an instance of *String* (a library class) which represents its *name*; the keyword `composition` implies that the object of class *String* is conceptually *part of* the object of class *Agent*. The code at lines 4 and 5 defines a constructor named *make* for *Agent* which takes a `literal` *n* as argument and can be exported to *Boss*, i.e., invoked by methods of class *Boss*. The constructor's body (line 5) invokes the constructor *make* of class *String* and

assigns the resulting reference to the object reference *name*. The code at lines 6 and 7 defines a method named *work*, also exported to class *Boss*. The class *Boss* has a constructor named *start* (line 12), which is exported to `all` classes. Such a constructor is the entry point of this application. So, firstly, an instance of *Boss* is created. During its construction, an instance of *Agent* named "James" (a literal given as a constructor's argument) is created (line 14), next its method *work* is invoked (line 15), next the object and its inner object of class *String* are migrated to a virtual machine named "Japan" (line 16) and, finally, the method *work* is invoked again, but this time it is a remote invocation since the object migrated (line 17).

```
.01 class Agent
.02 {
.03   composition String name;
.04   constructor make( literal n ) export { Boss }
.05   { name = String.make( n ); }
.06   method void work( ) export { Boss }
.07   { // do something useful }
.08 }
.09
.10 class Boss
.11 {
.12   constructor start( ) export all
.13   {
.14     Agent james = Agent.make(''James'' );
.15     james.work( );
.16     james.migrate( ''Japan'' );
.17     james.work( );
.18   }
.19 }
```

4.2 Example 2

The code below shows four application classes: *Patient*, *Filter* and *Doctor*. There are also classes named *Hospital* and *Nurse* which are not shown. Each class is described separately, as follows.

An instance of *Patient* contains an attribute `weight` (line 3) and an attribute `height` (line 4), both of class *Integer*, a library class. Those attributes are initiated by the constructor *instantiate* (line 5) and may be updated through the methods *SetWeight* (line 8) and *SetHeight* (line 11); the constructor can be invoked by class *Hospital*, while the two update methods can be invoked by class *Nurse*. The action named *NonStandardWeight* (line 14) implements a criteria to test whether a patient's weight is out of a certain range; if that is the case, the action answers `execute` (line 16), otherwise it answers `skip` (line 17). Similarly, the action named *NonStandardHeight* (line 18) implements a criteria to test whether a patient's height is out of a certain range. Those two actions can be invoked by class *Filter*.

```
.01 class Patient
.02 {
.03   composition Integer weight;
.04   composition Integer height;
.05   constructor instantiate( Integer weight, Integer height )
.06               export {Hospital}
.07   { this.weight = weight; this.height = height; }
.08   method void SetWeight( Integer weight )
.09               export {Nurse}
.10   { this.weight = weight; }
.11   method void SetHeight( Integer height )
.12               export {Nurse}
.13   { this.height = height; }
.14   action NonStandardWeight( Integer min, Integer max )
.15          export {Filter}
.16   { if ( weight.lt( min ) || weight.gt( max ) ) execute;
.17     else skip; }
.18   action NonStandardHeight( Integer min, Integer max )
.19          export {Filter}
.20   { if ( weight.lt( min ) || weight.gt( max ) ) execute;
.21   else skip; }
.22 }
```

An instance of *Filter* contains attributes *LowerWeight* (line 2), *UpperWeight* (line 3), *LowerHeight* (line 4) and *UpperHeight* (line 5). They are initiated by the constructor *instantiate* (line 6), which can be invoked by class *Hospital* (line 8). The action *accept* (line 13), which can be invoked by class *Nurse*, takes a reference to an instance of *Patient* as an argument and invokes its actions *NonStandardWeight* (line 15) and *NonStandardHeight* (line 16); if one of them answers `execute`, then the action answers `execute` as well (line 17); otherwise, it answers `skip` (line 18).

```
.01 class Filter {
.02   composition Integer LowerWeight;
.03   composition Integer UpperWeight;
.04   composition Integer LowerHeight;
.05   composition Integer UpperHeight;
.06   constructor instantiate( Integer LW, Integer UW,
.07                             Integer LH, Integer UW )
.08               export {Hospital}
.09   {
.10     LowerWeight = LW; UpperWeight = UW;
.11     LowerHeight = LH; UpperHeight = UH;
.12   }
.13   action accept( Patient patient ) export {Nurse}
.14   {
.15     if ( patient.NonStandardWeight(LowerWeight, UpperWeight) ||
.16          patient.NonStandardHeight(LowerHeight, UpperHeight) )
.17          execute;
.18     else skip;
.19   }
.20 }
```

An instance of *Doctor* has an associated instance of *Patient* (line 3). The constructor *instantiate* (line 4), which can be invoked by *Hospital*, initializes such attribute as `null` (line 5). The method *treat* (line 6), which can be invoked

by *Nurse*, takes a reference to an instance of *Patient* as an argument and assigns it as the currently associated patient (line 7). The action *CurrentPatient* (line 8), which can be invoked by *Hospital*, checks whether a certain patient is the patient currently associated to the doctor; the test at line 10 compares two object references: the method's argument and the class' attribute, both named *patient*.

```
.01 class Doctor
.02 {
.03   association Patient patient;
.04   constructor instantiate( ) export {Hospital}
.05   { patient = null; }
.06   method void treat( Patient patient ) export {Nurse}
.07   { this.patient = patient; }
.08   action CurrentPatient( Patient patient ) export {Hospital}
.09   {
.10     if ( patient == this.patient ) execute;
.11     else skip;
.12   }
.13 }
```

4.3 Example 3

The code below shows an application class named *Image2D* whose purpose is to illustrate the use of data blocks and how they serve to create new classes. An instance of *Image2D* encapsulates a data block referenced by *bitmap* (line 3). The two dimensions of the image stored by the data block is defined by the attributes *width* (line 4) and *height* (line 5). The constructor *make* (line 6), which can be invoked by `all` classes, takes two references to instances of *Integer* and, if their values are greater than zero (line 8), use them to set the stored image's width (line 10) and height (line 11) and to allocate memory space for the datablock (line 12). The method *set* (line 15), which can be invoked by `all` classes, set a specific bit of the data block, according to the coordinates x and y given as arguments. If they are valid coordinates (lines 17 and 18), the position of the corresponding bit is calculated, so that the resulting instance of *Integer* is assigned to the local variable p (line 20). An index, referenced as i, is created by invoking the method *makeIndex* through reference p (line 21); that is, the index value is set as the same integer value stored by the *Integer* object. The newly created index referenced as i is then bound to the data block referenced as *bitmap*. Finally, the bit of the data block indexed by i is set (line 23) and the method returns an instance of *Boolean*, a library class, containing the value true.

```
.01 class Image2D
.02 {
.03   datablock bitmap;
.04   composition Integer width;
.05   composition Integer height;
.06   constructor make( Integer width, Integer height ) export all
.07   {
```

```
.08     if ( width.gt( 0 ) && height.gt( 0 ) )
.09       {
.10         this.width = width;
.11         this.height = height;
.12         bitmap = datablock.make( width.multiply( height ) );
.13       }
.14    }
.15    method Boolean set( Integer x, Integer y ) export all
.16    {
.17      if ( x.geq( 0 ) && x.lt( width ) &&
.18           y.geq( 0 ) && y.lt( height ) )
.19        {
.20          Integer p = x.multiply( height ).add( y);
.21          index i = p.makeIndex( );
.22          i.bind( bitmap );
.23          bitmap.set( i );
.24          return Boolean.make( true );
.25        }
.26      else
.27          return Boolean.make( false );
.28     }
.29 }
```

5 Distributed Virtual Machine Architecture

The proposed architecture described here makes it possible to run code which conforms to the described object model and gives support to code and object mobility and to remote method invocation. Every virtual machine is composed of a Class Space, an Object Space and an Activity Space, detailed in the sequel.

5.1 Object Space

A collection of objects which are instances of application classes available at the local Class Space. All objects are referenced through a structure called *object table*, similarly to the *handle table* implemented by DOSA (Distributed Object System Architecture) [Hu et al., 2003]. [Figure 1] illustrates how objects are referenced both within a virtual machine and between virtual machines. The virtual machine named *Alpha* stores objects identified as *12* and *17*, while the virtual machine named *Beta* stores objects identified as *45* and *67*. An object table is an array of entries of two types: entry for local object and entry for remote object; this allows to distinguish local operation calls from remote ones. Thus, for each object there is an entry in the object table of the machine where the object resides. For instance, the object *12* is referenced by entry *0* of *Alpha*. An object cannot directly reference another; an object can only reference an object table entry in the same machine. For example, object *12* references entry *1* of *Alpha*, which, in turn, references object *17*; conceptually, object *12* references object *17*. An object may also conceptually reference an object that resides remotely. For example, object *17* – that resides in *Alpha* – references object *45* – that resides in *Beta*. This is implemented through the entry *2* of *Alpha*, which references entry

0 of *Beta*. Therefore, an entry for local object must contain an object reference, while an entry for remote object must contain a virtual machine name and an object table entry index.

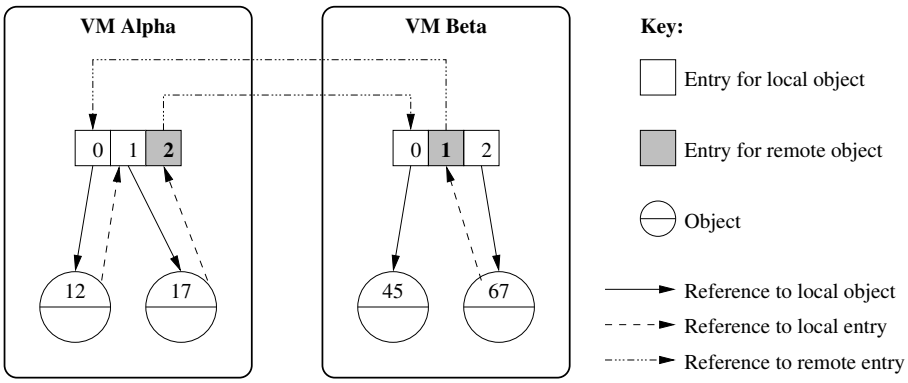

Figure 1: Example of object table

5.2 Class Space

A collection of application classes, a class table and a method table. Each class has a corresponding entry within the class table, and each method of each class has a corresponding entry within the method table. A class A references a class B when it contains an object reference of type B. According to the object model, that may happen in the following situations: *(i)* A has an attribute of type B; *(ii)* some method of A has either a formal parameter of type B or a return of type B; *(iii)* some method of A defines a local variable which is an object reference of type B. Anyway, a class references another through the class table. If class A references class B, it does not matter whether or not A and B are located in the same machine because de class table makes it transparent to refer to remote classes. Similarly, everywhere there is a method call, the object that represents such a call references an entry of the method table, instead of referencing the (object that represents the) method directly. Again, this gives transparency to code distribution because it does not matter whether a method call has as its target a local method or a remote method (with respect to code, which is independent of object placement). An object may exist within a certain machine as long as its corresponding class is also there. Thus, it may happen that a class gets replicated everywhere its instances exist.

5.3 Activity Space

An activity is the execution of a method. Virtuosi runtime system employees the traditional stack-based execution scheme to control activities: each element of the stack is an activity and each stack of activities correspond to a thread. Thus, the Activity Space is a collection of threads. There are two important details, though. Firstly, activities can be either synchronous or asynchronous, thus requiring specific management: every time a synchronous method invocation occurs, a new activity is pushed onto the caller activity stack, while every time an asynchronous activity is initiated, a new stack of activities (a new thread) is created. Secondly, Virtuosi aims at giving total transparency to remote method invocation, so an activity on one machine can start a new activity (actually, a new stack of activities) on a remote machine. Again, the runtime system must take care of the dependencies between such activities, including the necessary message exchange and fault tolerance measures.

5.4 Object and Code Mobility

Objects can migrate (move) from one virtual machine to another. Typically, an object migrates for efficiency and accessibility purposes, such as in applications where mobile devices carry some software. In Virtuosi, object migration can be programmed by using a set of operations defined according to [Jul et al., 1988], as follows.

move Moves an object to a certain machine.

fix Fixes an object on the machine where it resides, so it cannot migrate until it is unfixed.

unfix Undoes a previous *fix* operation, so that the object can migrate again.

refix Atomically, moves an object to a certain machine and fixes it there.

locate Returns the identity of the virtual machine where a given object resides.

When an object migrates, the object table of the source machine and the object table of the destination machine must be updated. In the destination machine, a new entry must be inserted: an entry for a local object. In the source object table, the existing entry for local object must be replaced for an entry for a remote object that references the newly created entry in the destination machine. As an object migrates to a different machine, it may be possible to perform some optimizations on the object table: if the object is referenced by an entry for remote object, then such entry is replaced by an entry for local object. For the moment, it is not possible to migrate an object while it is active. Thus,

the migration mechanism brings some constraints to object behavior: an object cannot migrate while it performs any activity and, conversely, an object cannot initiate a new activity while migrating. Composed objects must migrate all together, that is, the whole and its parts. As a consequence, the move operation may be not applied to an object that is part of another. Also, an object cannot migrate if it contains any object that is fixed.

Finally, when an object migrates to a certain machine, the migration mechanism must check whether or not the corresponding class is already there. If not, the class itself must be replicated (copied) to the target machine, thereby requiring update on class table and method table: a new entry for local class must be inserted in the class table of the new machine; a new entry for local method must be inserted in the method table of the new machine for every operation of the class; all references the class used to make to other classes (due to object references that represent attributes, local variables and formal parameters) and operations (constructor, method and action invocations) have to have new entries for remote class and remote operation in the tables of the new machine.

5.5 Remote Method Invocation

The remote method invocation mechanism is totally transparent in Virtuosi. Like any Remote Procedure Call (RPC) mechanism [Birrel and Nelson, 1984], there must be parameter marshalling, message exchange and some level of fault tolerance. The object table helps identifying whether a method invocation is either local or remote, thus providing *access transparency*: a programmer does not need to concern about distinguishing local and remote calls. Also, the object table helps finding an object when it happens to be remote, thus providing *location transparency*. The marshalling process is automatically done by using the information provided by program trees, which are available at runtime. In other words, there is no need to prepare stub code in advance. Some typical faults that may happen include: remote machine disconnection from the network and message loss. Such faults are properly handled by the mechanism.

6 Implementation Status

We have implemented a single-threaded version of the Virtuosi virtual machine, a compiler for the Aram language and a debugger. The compiler transforms classes coded in the Aram language into program trees which can be loaded and interpreted by the virtual machine. The debugger is a graphical tool that permits users to execute code step-by-step and watch the state of the virtual machine, i.e., it is possible to set breakpoints, step into methods, examine variables (object space), as well as the activity stack. All the implementation has been done in Java, just to easy the portability of the virtual machine implementation itself.

As a consequence, a simple strategy to represent program trees in intermediary format (between the compiler and the virtual machine) was to employ Java's object serialization scheme. Only a few concepts defined by the object model are not implemented by Aram yet, including class inheritance. The virtual machine implementation has been tested through a series of about 70 applications classes which fully verifies the proposed object model.

Object state and code migration is also implemented by the virtual machine, while object activity migration is under development [da Costa Cesar Filho, 2004]. The implementation has been done as a natural extension to the virtual machine kernel, since its structure (Object Space and Class Space) is intrinsically prepared for distribution. However, the migration primitives (move, fix, unfix, refix and locate), though supported by the virtual machine implementation, needs to be implemented by the Aram language and compiler yet.

Remote method invocation has been implemented separately from the virtual machine kernel, by employing the TCP/IP protocol to exchange messages between virtual machines [Noda, 2005]. Our experiments have shown that a remote method invocation takes about 340ms in average in the case where the invoked code is present at the same machine as the caller code; otherwise this time can reach about 2,300ms. The mechanism is currently being re-implemented by employing the Sun Microsystems JXTA peer-to-peer protocol and, then, integrated to virtual machine kernel. The purpose is to evolve Virtuosi as a platform for developing peer-to-peer applications.

7 Conclusion

We have introduced a new runtime platform named Virtuosi for building distributed object systems. It is based on virtual machine and object-oriented programming concepts. A previous work [Calsavara and Nunes, 2001] has shown that the main design principles of Virtuosi are feasible. Currently, a full-fledged version of the platform is under development. To date, we have developed a programming language and corresponding compiler, a prototype version of the virtual machine kernel, a mechanism for transparent remote method invocation and a mechanism for object and code mobility. That showed that our object model and our design of distributed virtual machine architecture work well. The main contribution of our research work, so far, is to show that is possible to build a virtual machines which are intrinsicly distributed and cooperate to provide for transparency to such distribution, including interoperability between applications which run atop heterogeneous systems.

In the near future, we expect to implement a mechanism for multi-threaded execution and respective concurrency control proposed in [Nunes, 2005]. That is going to allow building complex applications and further validate our object model and distributed virtual machine architecture. Also, a mechanism for

event-based distributed programming is under development, as an alternative to remote method invocation for communication between objects. That is, actually, a firm step towards evolving Virtuosi into a platform for developing peer-to-peer applications. Another critical work to do is to implement compilers for different programming languages in order to further assess the object model and also verify how applications developed by using distinct programming language interoperate.

References

[Andrews and Olsson, 1993] Andrews, G. R. and Olsson, R. A. (1993). *The SR Programming Language: Concurrency in Practice*. Benjamin/Cummings.

[Baillarguet and Piumarta, 1999] Baillarguet, C. and Piumarta, I. (1999). An highly-configurable, modular system for mobility, interoperability, specialization, and reuse. In *2nd ECOOP Workshop on Object-Orientation and Operating Systems (ECOOP-OOOSWS'99)*.

[Birrel and Nelson, 1984] Birrel, A. D. and Nelson, B. J. (1984). Implementing remote procedure calls. *ACM Transactions and Computer Systems*, 2(1):39–59.

[Calsavara, 2000] Calsavara, A. (2000). Virtuosi: Máquinas virtuais para objetos distribuídos. Technical report approved on internal examination for career ascencionx, Pontifícia Universidade Católica do Paraná, Curitiba, Brazil. 99 pages in Portuguese.

[Calsavara and Nunes, 2001] Calsavara, A. and Nunes, L. (2001). Estudos sobre a concepção de uma linguagem de programação reflexiva e correspondente ambiente de execução. In *V Simpósio Brasileiro de Linguagens de Programação*, pages 193–204. In Portuguese.

[da Costa Cesar Filho, 2004] da Costa Cesar Filho, J. (2004). Mecanismo de mobilidade de objetos para a virtuosi. Master's thesis, Pontifícia Universidade Católica do Paraná.

[DCE, 1992] DCE (1992). *Introduction to OSF DCE*. Prentice Hall, Englewood Cliffs, NJ.

[Folliot et al., 1997] Folliot, B., Piumarta, I., and Riccardi, F. (1997). Virtual virtual machines. In *Proceedings of the 4th Cabernet Radical Workshop*.

[Franz and Kistler, 1997] Franz, M. and Kistler, T. (1997). Does java have alternatives? In *Proceedings of the California Software Symposium CSS '97*, pages 5–10.

[Hu et al., 2003] Hu, Y. C., Yu, W., Cox, A., Wallach, D., and Zwaenepoel, W. (2003). Run-time support for distributed sharing in safe languages. *ACM Transactions on Computer Systems (TOCS)*, 21(1):1–35.

[Jul et al., 1988] Jul, E., Levy, H., Hutchinson, N., and Black, A. (1988). Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6:109–133.

[Kistler and Franz, 1996] Kistler, T. and Franz, M. (1996). A tree-based alternative to java byte-codes. In *Proceedings of the International Workshop on Security and Efficiency Aspects of Java '97*. Also published as Technical Report No. 96-58, Department of Information and Computer Science, University of California, Irvine, December 1996.

[Kolb, 2004] Kolb, C. J. J. (2004). Um sistema de execução para software orientado a objeto baseado em Árvores de programa. Master's thesis, Pontifícia Universidade Católica do Paraná.

[Mullender et al., 1990] Mullender, S. J., Rossum, G. v., Tanenbaum, A. S., Renesse, R. v., and Staveren, H. v. (1990). Amoeba: A distributed operating system for the 1990s. *IEEE Computer*, 23:44–53.

[Noda, 2005] Noda, A. K. (2005). Mecanismo de invocação remota de métodos em máquinas virtuais. Master's thesis, Pontifícia Universidade Católica do Paraná.

[Nunes, 2005] Nunes, L. R. (2005). Um mecanismo de controle de concorrência para jogos. Master's thesis, Pontifícia Universidade Católica do Paraná.

[Rumbaugh et al., 1997] Rumbaugh, J., Jacobson, I., and Booch, G. (1997). *Unified Modeling Language Reference Manual*. Addison-Wesley, Reading, MA.

[Soley and Kent, 1995] Soley, R. M. and Kent, W. (1995). The OMG object model. In Kim, W., editor, *Modern Database Systems*, chapter 2, pages 18–41. Addison-Wesley.

[Wirth and Gutknecht, 1992] Wirth, N. and Gutknecht, J. (1992). *Project Oberon - The Design of an Operating System and Compiler*. Addison-Wesley, Reading, MA.