Virtual Virtual Machines

Bertil Folliot,[1] Ian Piumarta, Fabio Riccardi

Projet SOR, INRIA Rocquencourt,
B.P. 105, 78153 Le Chesnay Cedex, France.

bertil.folliot@ibp.fr
{ian.piumarta, fabio.riccardi}@inria.fr

Keywords: distribution, interoperability, programming languages, virtual machines, operating systems, research directions.

1 Introduction

This paper originated from a series of discussions about what will be interesting in systems research during the next five years.

Systems researchers are seeing less and less scope for OS kernel hacking, network architectures and communication paradigms, which have now entered the industrial domain. In trying to answer the question "What's left over for us?" we found that there is a lot to do, as long as we accept a change of focus. Systems can no longer be a collection of "mechanisms" but rather have to address problems of engineering, and actively propose methodologies that enable application programmers to produce better products and to solve the emerging problems of distributed application domains.

We realised first that complexity and cost are driving quality out of software, and that present solutions will only introduce more problems later on. At the same time new applications coming from the wider acceptance of distributed computing, and the ubiquitous use of "intelligent" devices, are presenting new challenges to programmers—making the job even more difficult.

The result of this reflection is a proposal for a novel virtual execution environment, capable of efficiently and safely executing many different programming languages and paradigms.

It seems to be irrefutable that virtual execution environments and more abstract programming languages effectively improve the quality of software while reducing its cost. This technology has so far been considered too expensive (in terms of performance), although the new generation of high-speed processors have in many cases removed this objection.

[1] Also with the LIP6, Universités Paris 6 & 7, France.

What remains is the problem of rigidity in VM environments, which eventually leads to lack of interoperability. Our proposal renders the VM environment flexible, removing this final objection.

2 The problem

Many new distributed applications—such as cooperative work and multimedia—do not try to exploit distribution to increase performance, but rather to deal effectively with the geographical distribution of users and resources, and with security concerns such as data sensitivity, ownership, access rights, copyright, and so on. Such applications are usually characterised by complex data models, complex transactions, and dynamic (and randomly variable) configurations of heterogeneous interacting parties.

Dealing with heterogeneity poses severe obstacles to interoperability. Java is a step in the right direction, but it is still too rigid. It uses bytecodes as its distribution language, which implies a stable virtual machine that cannot be modified to fit particular applications. However, a single language (and execution model) cannot address every problem domain: creating Web applets, scripting a text editor, and programming a smart card reader or mobile host support station are domains in which the effective solutions are very different. These solutions require different programming languages, compiled code representations, data formats, security checks, and so on.

These different applications should nevertheless be executed in the *same* environment. This reduces resource consumption, promotes integration and component reuse (internal interoperability) and allows adaptation to new languages, code and data formats (external interoperability).

3 Virtual virtual machines

We propose a single environment—the virtual virtual machine (VVM)—supporting target applications built in almost any "bytecoded" programming language. We make no assumptions about the origin of these applications: they might come from a local disk, be "telecharged" over the Internet, downloaded from a smart card, and so on.

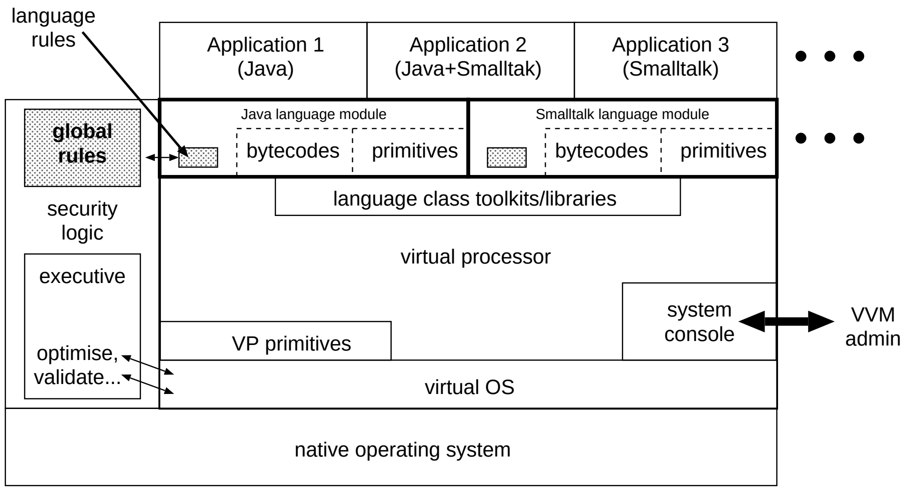

Figure 1: VVM architecture.

Applications are "typed" with an appropriate execution model (in accordance with current Web stan dards). Each application "type" corresponds to a virtual machine description (a *VMlet*) which includes a mapping for the application's bytecodes and data onto the VVM's executable mechanism and object model, and definitions of the primitives needed by the language. Internal interoperability is provided by making the data mapping reversible, allowing object to be shared safely between languages. VMlets are loaded on demand, whenever a new application "type" is encountered.

The VVM's execution mechanism is efficient in both space and time, translating application bytecodes into a "direct threaded" representation. This offers good performance, a significantly smaller footprint than dynamically-translated native code, is inherently highly portable, and provides a large "opcode space" for the executable representation.

The VVM must be able to verify that it is safe to add a given VMlet (which might have been downloaded along with the application it supports). This verification is similar to the static checks performed by a Java VM, although the VVM must apply these in a dynamic context defined by the VMlet itself. These rules operate at two levels: verifying the safety of application code, and (in conjunction with the data mapping defined in the VMlet) verifying the safety of object sharing between languages. Access control lists, and "code verification" prior to execution, are not sufficient to ensure safety in the virtual machine—since these presuppose that all VMlets are known in advance.

3.1 Architecture

A simplified diagram of the VVM architecture is shown in Figure 1.

3.1.1 Virtual Processor

The core machine is the VP, which provides a low-level execution engine. It has a simple intrinsic objet model (and corresponding object memory), instruction set and execution model. These are designed to be very general, supporting the many different object and execution models found in various classes of programming language—without corresponding precisely to any of them. The bytecodes supported by any given virtual machine are mapped onto the underlying VP object and execution models.

3.1.2 Virtual operating system

The VP runs the virtual operating system (VOS), which provides an abstraction over the native operating system's facilities. VOS services are obtained by VP programs by invoking virtual processor primitives. For example, the virtual operating system (VOS) is in charge of handling the VP scheduling and of administering protected resources. It is also responsible for dealing with reconfiguration when a new VMlet or toolkit is loaded.

The VP provides "introspective" instructions that can be used by trusted software (such as the VOS) to monitor and manage the operation of the VVM.

3.1.3 Virtual console

The VP also includes a tiny, but complete, programming and command language which we call the "virtual console". This language is used by a "VVM administrator" to perform manual interrogation and configuration of the VVM, and is used to implement the lowest layer of the VVM software. Because of the high degree of internal interoperability in the VVM, the higher layers of software (extended VP instructions in the VP libraries, bytecodes in the VMlets, and the security module) can be written in whatever is the most appropriate language.

3.1.4 Language toolkits

The core VP functionality can be extended with libraries (or "toolkits") that implement features common to an entire family of programming languages. Typical VP libraries provide cloning, class objects with instantiation, delegation or inheritance mechanisms, and special data representations such as tagged "immediate" types. Once defined, a library instruction can be used as if it were a core VP instruction; in effect, the VP instructions are used to define new VP instructions.

3.1.5 VMlets

Support for each specific language within a family is provided by a VMlet that defines the bytecodes and primitives required in order to execute compiled applications written in that language. Each language bytecode maps onto a single VP instruction. Where intrinsic (or toolkit) instructions are not appropriate, the VP instruction set is extended with new instructions (defined by the VMlet) in terms of existing VP instructions.

3.1.6 Application programs

The application bytecodes of each language are dynamically translated into the VP's executable instruction representation. Since all applications are translated into a single executable format (regardless of which VM defines their semantics), a single execution mechanism (defined by the VVM) can be used to run the code.

3.1.7 Security module

The VOS cooperates with the security module to perform the "management" tasks that are required to ensure (above all) safety and security. The global (inter-language) aspects of these tasks are described by a set of rules in the security module itself.

Rules relating to individual languages are provided by each VMlet. These rules describe the security constraints for applications running on top of that VM, and how the VP is allowed to manipulate the objects belonging to this VM. Applying these rules transitively allows safe interoperability and exchange of data between applications running on different VMs.

3.2 VVM operation

When an application arrives at the VVM, its "type" is checked. If the VVM has not already loaded the appropriate VMlet, then it is located and loaded.

The security module verifies the VMlet according to the global rules, to ensure that the safety of the VVM is not compromised.

The VMlet defines translations from the VM bytecodes to the execution mechanisms of the virtual processor (VP). The VP's instruction set is extended, if necessary, with operations defined either by the VMlet itself or imported from one or more VP libraries containing common operations for the class of languages to which the VM belongs. (The VMlet also defines how its VM's object model maps onto the VP's object model; making these mappings "reversible" allows the VVM to map from the VP's model back to each VM's model, permitting safe internal interoperability.)

The application is then loaded. Language-specific security rules, defined by the VMlet, are applied to the application's bytecodes (and data if appropriate).

Application execution can now begin. Methods (procedures, member functions, or whatever else the VM might call its units of executable code) are dynamically translated, on demand, into the corresponding VP instructions for execution.

The VMlet provides language-specific primitives for its applications. These are defined in terms of the VOS services, and the intrinsic primitives provided by the VP.

The VOS abstracts the underlying operating system. It provides each VM's primitives with controlled access to the machine's resources: communications, filestore, and so on. It is also responsible for allocating and managing the resources needed by the VVM itself.

4 Related work

IBM and Taligent have announced plans to build a Universal Virtual Machine capable of executing programs written in Java, Smalltalk and Visual Basic. Internal interoperability permits inter-language reuse of components, and external interoperability allows ”applets” built in any of these languages to be downloaded and executed.

5 Conclusions

This document proposes a “universal”, or “virtual” virtual machine that provides a maximum of support for many essential aspects of the emerging distributed application domains:

External Interoperability: for smart cards, mobile computing, cooperative applications, and other kinds of problem related to the exchange of functionality as well as the data that it operates on.

Internal interoperability: for applications that need to exchange or manipulate data between components compiled into different bytecoded representations, and to promote reuse of components between languages.

Adaptability: to extend or adapt existing services in (for example) embedded systems.

Reusability: in the VM implementations, application components, and operating system services. This in turn leads to…

Reliability: since the increased costs of developing each high-quality component is amortised over the many uses of that component within a single integrated environment;

Extensibility: not only in application components, but in the mechanisms used to execute them. Embedded systems in particular can benefit from an execution environment that can be upgraded easily, specialised for each application, or dynamically reconfigured depending on the needs of the host system at any particular moment.

6 Future work

A comparative analysis of current VM and dynamic object-oriented language implementations would be the first step towards a VVM.

A study of the implementation techniques for representative languages belonging to several different “families” (Java, Smalltalk, ML, and so on) will lead to a design of the intrinsic VP instructions and object model. From this we can develop the “virtual console” and its “language”.

The design of the VOS requires an analysis of languages’ runtime requirements, and a study of the appropriate abstractions of native OS services and their interaction with the VVM.

The appropriate introspection and security features of the VVM can build on current reflective system techniques. The “morphology” of the global and VMlet rules should be designed so that these are functional while remaining simple, readable, and *provable*.

7 References

[Atkinson 96] Atkinson, M.P. et al, *Design issues for persistent Java: A type-safe, object-oriented, orthogonally persistent system*, in Proc. 7th International Workshop on Persistent Object Systems (POS 7).

[Feldt 97] Robert Feldt, *Dependable Software Systems by means of AI techniques*, 3rd Cabernet Plenary Workshop, Rennes, April 1997

[IBM 97] *IBM plans cross-platform competitor to Java*, InfoWorld Electronic, April 1997. `http://www.computerworld.com/search/AT-html/online/9706/970613ibmplan.html`

[JAVA 96] Arnold, K. and Gosling, J. *The Java Programming Language*, Addison Wesley, 1996. ISBN 0-201-63455-4

[Maziere 97] David Mazieres, Frans Kaashoek, *Secure Applications Need Flexible Operating Systems*, 6th Workshop on Hot Topics in Operating Systems (HotOS-VI), May 1997, Cape Cod, Massachusetts.

[Seltzer 97] Margo Seltzer, Christopher Small, *Self-Modifying and Self-Adapting Operating Systems*, 6th Workshop on Hot Topics in Operating Systems (HotOS-VI), May 1997, Cape Cod, Massachusetts.