A virtual-machine-based middleware

Alcides Calsavara , Agnaldo K. Noda , Juarez da Costa , Carlos Kolb , Leonardo Nunes

[1] Pontifícia Universidade Católica do Paraná
Programa de Pós-Graduação em Informática Aplicada
Rua Imaculada Conceição, 1155, Prado Velho,
80215-901 Curitiba, PR, Brazil

{alcides,anoda,juarez,kolb,leonardo}@ppgia.pucpr.br

Abstract. *Currently, a number of distributed software systems development tools exist, but typically they are designed either to satisfy industrial standards – industrial perspective – or to experiment new concepts – research perspective. There is a need for software development tools where programmers can both learn about distributed computing – pedagogical perspective – and build quality distributed software systems through prototyping – experimental perspective. This paper introduces the* VIRTUOSI *Project, which aims at building a toolkit to assist in developing and executing distributed software systems from both pedagogical and experimental perspectives. It combines virtual machine, object-oriented programming and computational reflection concepts to give those perspectives. The* VIRTUOSI *runtime environment can be seen as a reflective composed of virtual machines, which provides mechanisms and gives transparency to critical aspects of distributed systems, such as objects migration, remote method invocation and code placement.*

Resumo. *Atualmente, existem muitas ferramentas para o desenvolvimento de sistemas de software distribuído, mas, tipicamente, são desenhadas para satisfazer padrões industriais – perspectiva industrial – ou então para experimentar novos conceitos – perspectiva de pesquisa. Há necessidade de ferramentas para o desenvolvimento de software através das quais programadores possam tanto aprender sobre computação distribuída – perspectiva pedagógica – quanto construir sistemas de software distribuídos com qualidade através de prototipação – perspectiva experimental. Este artigo apresenta o Projeto* VIRTUOSI, *o qual tem por objetivo a construção de ferramentas para apoio no desenvolvimento e na execução de sistemas de software distribuído com as perspectivas pedagógica e experimental. O projeto combina conceitos de máquinas virtuais, programação orientada a objetos e reflexão computacional a fim de dar tais perspectivas. O ambiente de execução da* VIRTUOSI *pode ser visto como um* middleware *formado por máquinas virtuais, no qual há mecanismos que permitem e dão transparência ao principais aspectos de sistemas distribuídos, tais como migração de objetos, invocação remota de métodos e alocação de código executável.*

1. Introduction

The importance of distributed computing has grown significantly in the last years due to the incresing use of the Internet as a means of information systems deployment. Many new applications have emerged and new ones are expected in the near future, especially in the fields of embbeded systems and mobile devices – the so called ubiquitous computing. This scenario promises a high demand for distributed software system development in the next years.

However, distributed computing introduces great complexity in software systems development, deployment and maintenance. A number of requirements which are not normally present in centralized systems may need to be fulfilled in distributed systems, such as reliability of an interprocess message exchange protocol. Also, requirements which are already present in centralized systems may be more difficult to implement in distributed systems, such as security. As a consequence, developing quality distributed software systems is hard and relies fundamentally on programmers expertise and good tool assistance.

A programmer becomes an expert in developing distributed software systems firstly when she is properly taught distributed computing concepts and secondly when she is properly trained to use specific technological artifacts, such as a distributed programming language or a middleware for distributed execution. Often, concepts are learned by experimenting with technological artifacts, where knowledge on theory and practice come together. The learning process is complex and the learning curve depends on many factors, but surely the technological artifacts employed are decisive.

A tool for developing distributed software systems can provide a series of features to programmers, from conceptual modeling to physical system installation. Naturally, the quality of a distributed software system is strongly influenced by the features provided by such a tool and how programmers use them. One such feature that is decisive for the success of a system is the capability to easyly create prototypes, that is, create a preliminary version for the target system where its requirements – either established in the first place or introduced later – can be quickly implemented, debugged, tested and simulated.

Currently, a number of distributed software systems development tools exist, but they hardly favor learning about distributed computing and hardly favor prototyping because they are typically designed either to satisfy industrial standards – industrial perspective – or to experiment new concepts – research perspective. Industrial tools are concerned with productivity and software efficiency and robustness; they hardly permit a programmer to develop any task with simplicity and focused on a single problem, i.e., industrial tools invariably forces programmers to care about requirements that operational releases of real-world applications have and need to be considered despite the problem under study. That surely distracts programmers and may compromise both developing and learning curve. On the other hand, research tools normally have complex user interfaces and require the knowledge of particular concepts. Programmers often find it difficult to use research tools because they require a considerably large amount of work and time to build even small applications.

Therefore, there is a need for software development tools where programmers can both learn about distributed computing – pedagogical perspective – and build quality distributed software systems through prototyping – experimental perspective. A pedagogical tool should implement the main established principles of distributed computing in a clean way and should be open to be enhanced with trial concepts. An experimental tool should conform with the main established technologies, so that it would be possible to convert a prototype to a corresponding operational release.

The remaining of this paper is organized as follows. Section 2 presents the objectives of a new toolkit for building distributed applications named VIRTUOSI. Section 3 describes the main design principles of VIRTUOSI. Section 4 discusses how distributed objects are managed in VIRTUOSI, gives an overview on how they can migrate and how remote method invocation is implemented. Finally, Section 6 presents some conclusions and discusses future work.

2. Objectives

VIRTUOSI Project aims at building a toolkit to assist in developing and executing distributed software systems from both pedagogical and experimental perspectives. From pedagogical perspective, VIRTUOSI will permit programmers to be taught about distributed computing in a structured manner; distributed programming concepts and techniques would be introduced one by one and each studied separately from others. As a consequence, programmers should get a better understanding of distributed computing and the learning curve should get accelerated. From experimental perspective, VIRTUOSI will permit programmers to create prototypes which are mature and robust; they will be mature because all typical system requirements will be implementable, and they will be robust because it should be easy to carry tests on separate units, followed by integrated tests, where it would be possible to simulate all real-world operational configurations and circumstances, independtly of particular technological aspects. Because of their maturity and robustness, such prototypes will be the basis for easyly developing the corresponding operational releases by using specific technologies. As a net effect, VIRTUOSI will assist in developing distributed software systems of great quality in a short period of time, since programmers will be better trained and will be able to implement and test critical system requirements in a controlled manner.

3. Key Design Decisions

The VIRTUOSI toolkit encompasses many aspects of distributed computing and of software engineering. It should comprise artifacts to build software systems and a full-fledged distributed runtime system. The pedagogical perspective requires an environment where a programmer can write a program by using a simple yet powerful set of abstractions, and then test that program in a way that all abstractions employed can be easyly traced, i.e., translations from programming abstractions to runtime structures should be minimized. Another requirement from the pedagogical perspective is that the environment should be as neutral as possible with respect to the actual runtime platform in order to avoid unnecessary distractions. Finally, the pedagogical perspective requires an environment where the programmer can easyly select which system aspects should be either transparent or translucent in a given moment. The experimental perspective, on the other hand, requires an environment where real-world applications can be quickly developed and carefully tested. The subsequent sections present the key design decisions made for the VIRTUOSI toolkit in order to satisfy the requirements discussed so far, namely *virtual machine*, *object-oriented programming* and *computacional reflection*.

3.1. Virtual Machine

The VIRTUOSI runtime environment is composed of a collection of communicating virtual machines. In a simplified way, 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. 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 environment for distributed software systems. In fact, this runtime environment can be seen as a *middleware*, similarly to systems based on the CORBA Standard [Soley and Kent, 1995], since a distributed software system can run on a heterogeneous computer network.

Such an approach to distributed computing – based on virtual machines – is in accordance with the objectives of VIRTUOSI Project (Section 2) due to the following reasons:

neutral architecture A virtual machine is not tied to any particular computer architecture; it implements only core computer features which are common to standard computer technologies. From experimental perspective, this ensures that prototype software systems which run on VIRTUOSI machines can be easyly translated into operational releases that run on any typical real-world machines, while not precluding code optimization for better use of particular computer architeture features. On the other hand, from pedagogical perspective, the simplicity of a VIRTUOSI machine architecture makes it appropriate for training computer programmers since the number of concepts to work with is small; consequently, programmers are forced to know how to combine such concepts to build complex applications.

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 VIRTUOSI machine for each operating system. Another consequence is that a software system that runs on a specific VIRTUOSI machine implementation will run 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. From experimental perspective, this portability helps building prototypes when a group of programmers who use distinct operating systems work cooperatively; they can share code without getting down to runtime environment specifics, thus improving productivity. From pedagogical perspective, it helps programmers to write exercises in steps where several distinct computers can be employed without causing any distractions. Yet another consequence of the use of virtual machines is that VIRTUOSI applications are mobile:

they can move through heterogeneous computers, at runtime, as long as proper implementations of the VIRTUOSI machine are provided. This mobility can be very useful since it is a requirement that often appears in modern applications, especially in ubiquitous computing.

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. From experimental perspective, this is an essential feature to improve productivity. From pedagogical perspective, it is also important because programmers can use debugging to understand better software systems behaviour.

flexible network configuration Since a VIRTUOSI machine is a user-level process, there may exist any number of instances of the virtual machine running on a single computer. As a consequence, a collection of n virtual machines may run atop a network containing from 1 to n computers. In the extreme, there is no need to have a real computer newtork to run a VIRTUOSI distributed application. According to [Silberchatz and Galvin, 1998], this concept was first experimented by the IBM VM Operating System, where a set of virtual machines run on a single physical computer, giving the illusion that each user has its own computer; communication between the virtual machines happens through a virtual network. From experimental perspective, such feature may easy the development of prototypes, since any network configuration can be simulated. From pedagogical perspective, it may help programmers to experiment with distributed computing even when only a single machine is available.

3.2. Object-Oriented Programming

Probably, object-oriented programming is the most widely accepted paradigm for distributed computing, both in academy and industry. Object orientation was first introduced by Simula-67 [Dahl and Nygaard, 1970] as means to represent real-world entities for the purpose of simulation only, and got popularity after Smalltak-80 [Goldberg and Robson, 1983] and C++ [Stroustrup, 1986]. Currently, there is a number of programming languages that support object-oriented programming concepts and they are largely employed in computer programmer training for more than a decade. More recently, with the incresing demand for Internet-based applications, new languages and tools have appeared and, practically, all of them are object oriented. Perhaps, the most significant example is Java [Arnold and Gosling, 1996], which, despite its industrial flavor, is very much used in introductory programming courses and also motivates much of the current research in distributed computing. Another important example is Eiffel [Meyer, 1997], a languague that rigorously implements object-oriented concepts.

In fact, the object-oriented paradigm is present in almost every new architectural development in the distributed system community. For instance, both the Open Distributed Processing (ODP) and the Object Management Group (OMG), the main standardization initiatives for heterogeneous distributed computing, are based on object concepts. In the software industry, two important examples of the use of object-oriented concepts are the Sun Microsystems' Java-based J2EE and the Microsoft .NET plaftorm.

VIRTUOSI project adopts object orientation as the paradigm for both applications development and runtime system. Programmers should develop applications by employing solely object-oriented concepts, assisted by proper artifacts, such as a rigorously object-oriented programming language, and tools, such as a compiler built according to the pedadogical perspective, i.e., a compiler that helps in training rather than simply checking the source code. The runtime system – defined by a collection of virtual machines (Section 3.1) – should preserve all object-oriented abstractions in order to minimize translations that could make difficult debugging applications; that helps in fast building prototypes – the experimental perspective – and helps programmers to understand better programming concepts – the pedagogical perspective.

3.3. Computational Reflection

Three different yet complementary approaches to the use of the object paradigm in concurrent and distributed contexts are discussed in [Briot et al., 1998]:

library approach Object-oriented concepts, such as encapsulation, genericity, class and inheritance, are applied to structure concurrent and distributed software systems through class libraries. It is oriented towards *system builders* and aims at identifying basic concurrent and distributed abstractions – it can be viewd as a bottom-up approach where flexibility is priority. Its main limitation is that programming is represented by unrelated sets of concepts and objects, thus requiring great programmers expertise. Examples of the library approach are the ISIS System [Birman, 1985] and the Arjuna System [Parrington et al., 1995].

integrative approach Object-oriented concepts are unified with concurrent and distributed system concepts, such as object with activity. It is oriented towards *application builders* and aims at defining a high-level programming languague with few unified concepts – it makes mechanisms more transparent. Its disadvantage is the cost of possibly reducing the flexibility and efficiency of the mechanisms. Examples of the integrative approach are the distributed operating systems Amoeba [Mullender et al., 1990] and Mach [Boykin et al., 1993].

reflective approach Integrates protocol libraries within an object-based programming language; the application program is separated from various aspects of its implementation and computation contexts – *separation of concerns* – by describing them in terms of metaprograms, according to the concept of computational reflection, firstly disseminated by [Maes, 1987]. It is oriented towards both application builders and system builders and, in fact, bridges the two previous approaches by providing a framework for integrating protocol libraries within a programming language or system – combination of flexibility and transparency.

VIRTUOSI Project adopts the reflective approach, thus allowing programmers change systems behaviour in two levels: application level and runtime system level. Such approach conforms to the established project objectives (Section 2) because, from pedagogical perspective, programmers can selectively choose what system features should be or not transparent, so that it is possible to study each feature individually or combine just some of them. And, from experimental perspective, programmers have a great dynamism for developing software systems as components can be easyly replaced and tested. In fact, the VIRTUOSI runtime environment can be seen as a reflective middleware, like the CORBA-based implementations DynamicTAO [Kon et al., 2000] and Open ORB

[Blair et al., 2001]. The reflective middleware model is a principled and efficient way of dealing with highly dynamic environments yet supports development of flexible and adaptative systems and applications [Kon et al., 2002]. Naturally, such flexibility may be difficult to achieve and should require a consistent model for composing meta-level system resources, such as the framework proposed in [Venkatasubramanian, 2002].

In VIRTUOSI, the combination of virtual machine and object orientation brings a particularly interesting feature for the implementation of computational reflection: objects and their corresponding code can be explicitly stored and manipulated at runtime, thus permiting reflection on practically any computation aspect, easying dynamic modification of systems behaviour. The architecture of a VIRTUOSI machine is, therefore, oriented towards computational reflection, besides all aspects related to the pedagogical and experimental perspectives, discussed so far. In other words, a VIRTUOSI machine should have access to all the semantics of an application for providing good support to programmers for both pedagogical and experimental purposes and for permiting full reflection. This feature differs VIRTUOSI from other attempts such as the Guaraná Project [Oliva, 1998], where the Java Virtual Machine was modified to support computational reflection, but entirely preserving its standard code format – the so called bytecode – and programming language compatibility. It differs, as well, from the PJama Project [Atkinson, 1998], where the Java Virtual Machine is modified in order to support orthogonal persistence.

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. Program trees are successfully employed in the Juice Virtual Machine [Kistler and Franz, 1997, Franz and Kistler, 1997] for transfering code through the network; when a program tree reaches its destination is then translated to a specific machine code for execution. 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 *metamodel* which are expressed in the Unified Modeling Language (UML) [Rumbaugh et al., 1997]; the objects of a program tree are instances of the classes present in the metamodel.

4. Distributed Objects

Objects reside within virtual machines and can reference each other locally and remotely. When an object has a reference to another, it can invoke methods; the invocation is local when the reference is local, otherwise it is remote. An object *A* obtains a reference to an object *B* by one of the following means:

- Object *A* creates object *B*.
- Object *A* receives a method invocation where an object *B* is passed as a parameter.
- Object *A* invokes a methods of some object and receives object *B* as return.

Because in VIRTUOSI objects are always created locally, an object can only obtain a reference to a remote object either when it is passed as parameter or when it is returned. A third case may happen when an object migrates: a local reference may become a remote reference.

As discussed, the management of objects in VIRTUOSI can be very complex, thus requiring a proper implementation. The subsequent sections describes the handle table mechanism adopted and how migration and remote method invocation use it.

4.1. Handle Table

All objects are referenced through a structure called *handle table*, similarly to they way it is implemented in 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 VM named *Alpha* stores objects identified as *12* and *17*, while the VM named *Beta* stores objects identified as *45* and *67*. A handle table is an array of entries of two types: entry for local object and entry for remote object. Thus, for each object there is an entry in the handle 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 a handle 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 contains a virtual machine name and a handle table entry index.

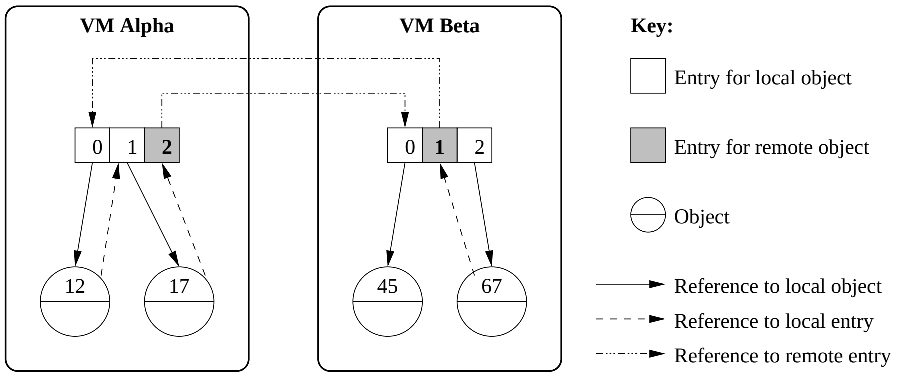


Figure 1: Example of handle table

4.2. Object Migration

Object can migrate from one virtual machine to another. Typically, object migrate for efficiency (load balance) and accessibility purposes. From pedagogical perspective, it may be interesting to migrate objects to observe differences between local and remote communication. From experimental perspective, it may be interesting to migrate objects to simulate 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 a local object to another machine.
fix Fix an object on the machine where it resides, so it cannot migrate anymore.

unfix Undoes a previous *fix* operation, so that the object can migrate again.
refix Atomically, moves an object to another machine and fixes it there.
locate Returns the name of the virtual machine where a given object resides.

When an object migrates, the handle table of the incoming machine and the handle table of the destination machine must be updated. In the destination machine, a new entry must be inserted: an entry for local object. In the incoming table, the existing entry for local object must be replaced for a entry for remote object that references the newly created entry in the destination machine.

The migration mechanism brings some constraints to object behaviour:

- An object cannot migrate while it performs any activity.
- An object cannot initiate a new activity while migrating.

Composed objects must migrate all together. As a consequence, the move operation has no effect for an object that belongs to another. Also, an object cannot migrate if it contains any object that is fixed.

4.3. 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 handle table helps identifying whether a method invocation is either local or remote, thus providing *access transparency* [Tanenbaum, Andrew S., 1995]: a programmer does not need to concern about distinguishing local and remote calls. Also, the handle 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 disconnects from the network, message loss and target object is under migration. All these faults require proper treatment.

5. Architecture Overview

The previous Section described how distributed objects are managed by employing handle tables. This very same concept is exploited as well to manage distributed code in the VIRTUOSI middleware. This Section presents an overview of the architecture of VIRTUOSI, where practically all elements can be distributed, including objects, code and activities.

The code interpreted by a VIRTUOSI machine has a very special format: it is graph of objects which represent the original source code, in such a way that all the semantics is preserved and made available at runtime. Such objects are instances of classes defined in the VIRTUOSI metamodel. (It is out of the scope of this paper to describe the metamodel.) For each user class, our compiler creates a graph of objects, which we simply call an *ADT* (Abstract Data Type). Thus, there will be an object that represents the whole ADT, and there will be an object that represents a method that belongs to the ADT, and there will be an object that represents a formal parameter of a method, and there will be an object that represents a method call, and so on. Such complex representation, when compared to Java bytecode, is justified in our project because we intend to give full support to computational reflection.

Another important characteristic of VIRTUOSI is that methods can be invoked in two different modes: synchronous or asyncronous. However, such mode is not fixed per method, as it occurs in traditional systems. In VIRTUOSI, the caller must choose the mode it desires. Thus, a certain method can be called synchronously at one moment (by one client), while asynchronously later (by another client, perhaps).

Figure 2 shows two VIRTUOSI machines and their main elements. Each machine contains a the following spaces:

ADT Space A collection of ADT's (one for each user class). Each ADT has a corresponding entry within the local *ADT Table*. Thus, if an ADT needs to reference another, it does it through the ADT Table. For example, a certain ADT, say *A*, may define an attribute whose type is a second ADT, say *B*. As the Figure shows, it does not matter whether *A* and *B* are located in the same machine because de ADT Table makes referencing remote ADT's transparent. There is also a *Method Table*: each method (defined within a certain ADT) must have an entry in that table. The consequence of this is that every time there is a method call, the object that represents such a call references an entry of 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 its target a local method or a remote method (with respect to code, which is independent of object placement).

Object Space A collection of objects which are instances of ADT's available on the local ADT Space. That implies that an object may exist within a certain machine only if its corresponding ADT is also there. Obviously, it may happen that the same ADT gets replicated amongst machines if its instances are distributed. Another consequence is that when an object migrates to a certain machine, the migration mechanism must check if the corresponding ADT is already there or not. If not, the ADT itself must migrate or copied to the target machine, thus requiring update on ADT and Method Tables. References between objects – either local or remote – are managed by the *Object Table*, which was described in the previous Section.

Activity Space An activity is the execution of a method. VIRTUOSI runtime system employes the traditional stack-based execution algorithm. There are two important detais, however. Firstly, activities can be either synchronous or asynchronous, thus requiring some special care. 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.

6. Conclusions and Future Work

We have introduced a new toolkit named VIRTUOSI for building distributed object systems with pedagogical and experimental perspectives. It combines virtual machine, object-oriented programming and computational reflection concepts to give those perspectives. 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 toolkit is under development.

References

Arnold, K. and Gosling, J. (1996). *The Java Programming Language*. Addison Wesley.

Atkinson, M. (1998). Providing orthogonal persistence for java. *Lecture Notes in Computer Science*, (1445):383–395. ECOOP'98.

Birman, K. P. (1985). Replication and fault-tolerance in the ISIS System. *ACM Operating System Review*, 19(5). Proceedings of the 10th ACM Symposium on Operating System Principles.

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

Blair, G., Coulson, G., Andersen, A., Blair, L., Clarke, M., Costa, F., Duran-Limon, H., Fitspatrick, T., Johnston, L., Moreira, R., Parlavantzas, N., and Saikoski, K. (2001). The design and implementation of Open ORB. In *IEEE Distributed Systems Online*.

Boykin, J., Kirschen, D., Langerman, A., and Loverso, S. (1993). *Programming under Mach*. Addison-Wesley, Reading, MA.

Briot, J.-P., Guerraoui, R., and Lohr, K.-P. (1998). Concurrency and distribution in object-oriented programming. *ACM Computing Surveys*, 30(3):291–329.

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.

Dahl, O.-J. and Nygaard, K. (1970). Simula-67 common base language. Technical Report S-22, Norwegian Computing Centre, Oslo.

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

Goldberg, A. and Robson, D. (1983). *Smalltalk-80: The Language*. Addison-Wesley, Reading, MA.

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, 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, T. and Franz, M. (1997). 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.

Kon, F., Costa, F., Blair, G., and Campbell, R. H. (2002). The case for reflective middleware. *Communications of the ACM*, 45(6):33–38.

Kon, F., Roman, M., Liu, P., Mao, J., T., Y., Magalhães, L., and Campbell, R. (2000). Monitoring, security, and dynamic configuration with the DynamicTAO reflective ORB. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware2000)*, pages 121–143.

Maes, P. (1987). Concepts and experiments in computational reflection. *ACM SIGPLAN Notices*, 22(12):147–155. OOPSLA'87.

Meyer, B. (1997). *Object-Oriented Software Construction*. Prentice Hall PTR, second edition.

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.

Oliva, A. (1998). Guaraná: Uma arquitetura de software para reflexão computacional implementada em java. Master's thesis, Universidade Estadual de Campinas, Instituto de Ciência da Computação.

Parrington, G. D., Shrivastava, S. K., Wheater, S. M., and Little, M. C. (1995). The design and implementation of Arjuna. *USENIX Computing Systems Journal*, 8(3).

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

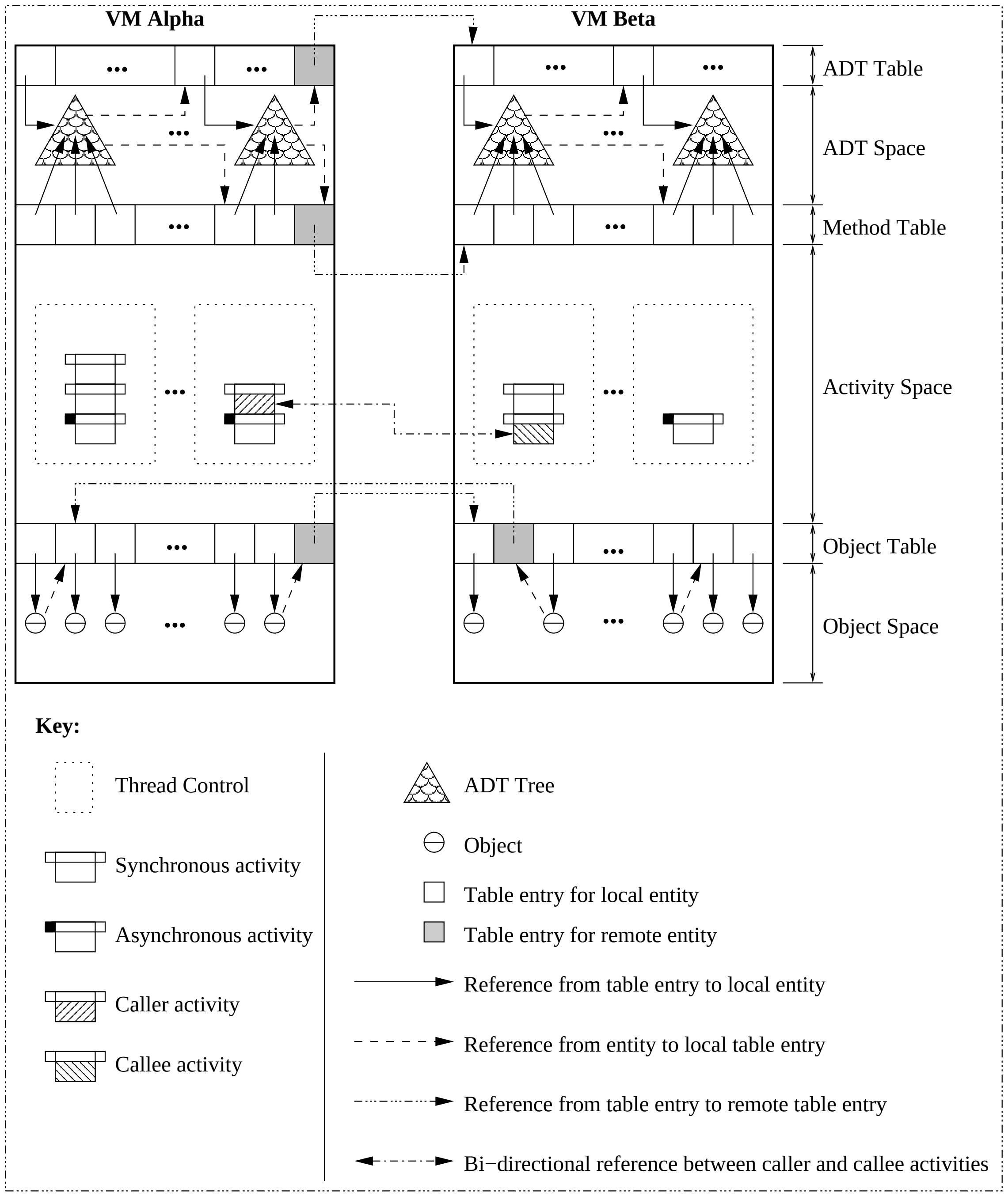
Silberchatz, A. and Galvin, P. B. (1998). *Operating System Concepts*. Addison-Wesley, fifth edition.

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.

Stroustrup, B. (1986). *The C++ Programming Language*. Addison Wesley, Reading, Massachusetts.

Tanenbaum, Andrew S. (1995). *Distributed Operating Systems*. Prentice Hall.

Venkatasubramanian, N. (2002). Safe composability of middleware services. *Communications of the ACM*, 45(6):49–52.



VM Alpha
VM Beta
ADT Table
ADT Space
Method Table
Activity Space
Object Table
Object Space
Key:
Thread Control
Synchronous activity
Asynchronous activity
Caller activity
Callee activity
ADT Tree
Object
Table entry for local entity
Table entry for remote entity
Reference from table entry to local entity
Reference from entity to local table entry
Reference from table entry to remote table entry
Bi−directional reference between caller and callee activities

Figure 2: Elements of the VIRTUOSI **middleware**