Separation of Concerns and Operating Systems for Highly Heterogeneous Distributed Computing

Yasuhiko Yokote
Sony Computer Science Laboratory Inc.
Takanawa Muse Building
3-14-13 Higashi-gotanda, Shinagawa-ku,
Tokyo 141 JAPAN
ykt@csl.sony.co.jp

Gregor Kiczales
Xerox PARC
3333 Coyote Hill Rd. Palo Alto,
CA 94304 U.S.A
gregor@parc.xerox.com

John Lamping
Xerox PARC
3333 Coyote Hill Rd. Palo Alto,
CA 94304 U.S.A
lamping@parc.xerox.com

Abstract. Highly heterogeneous distributed computing applications present a tremendous challenge for operating system implementors. These applications present an extremely wide range of demands and must run in an extremely wide range of environments. In order to recognize the wide range of environments, without losing the benefits of traditional modularity, we propose a separation of concerns approach to the design of operating systems for this domain. In this paper, we first discuss importance of separation of concerns particularly in highly heterogeneous distributed computing. The discussion is based on the issues we encounter when programming of distributed applications such as for networked copiers, settop boxes for cablenets, networked analyzers/oscilloscopes, or palmtop computers. We then propose a way to solve them without sacrifice of performance. This paper contributes to this workshop in the sense that it addresses frustrations of programmers, particularly of embedded systems, and demonstrates our technology to solve their frustration from an operating systems' perspective.

1 Introduction

The goal of the Apertos project is to explore the operating system requirements of applications in a highly heterogeneous, highly distributed environment[Yokote *et al.* 89][Yokote *et al.* 91][Yokote 92]. The kinds of applications we envision are those running on embedded networked systems ranging among camcorders, VCR's, TV's, servers/settop boxes for cablenets, (networked) game machines, copiers, fax machines, phones, pagers, networked analyzers/oscilloscopes, palmtop computers and the like. These applications are characterized by the usual embedded concerns such as realtime constraints and limitation of available memory, as well as by being distributed. These systems provide sophisticated functions to users, and at the same time interact with an external computational environment. The amount of complexity of this code is growing continuously, for example, one common VCR is currently equipped with two or more processors each of which has 54K bytes of code in an assembly language.

Our goal is to provide an infrastructure of operating system support that simplifies these programs, make them more portable, and make them more maintainable. However, the problems of distribution and heterogeneity introduce significant complexity. Also rapidly changing substrate means we need portability and maintainability. Even though we have, over the years, amassed tremendous expertise at organizing medium scale programs running on a single processor, and even some expertise at large relatively heterogeneous distributed systems, it is not clear what programming style makes programs portable and what operating system interface is suitable for programming these class of applications. In particular, when we begin writing these applications, they are completely cluttered with the complexities of distribution and heterogeneity, which includes mapping application modules to system processes, allocating a process to a processor, persistence of data, etc.

The above applications have sometimes been constructed on top of a realtime operating system, which gives programmers the abstraction of time management as well as process synchronization and memory management, but not abstraction of distribution. This paper suggests an approach for supporting distribution heterogenous substrates.

2 Separation of Concerns

The introductory section should have motivated that distribution, heterogeneity, persistence, scheduling, message passing, etc. cannot be ignored. We made a fundamental strategic decision to think about two concerns separately: ideal abstraction and mapping to its efficient implementation. We call this separation of concerns.

Programmers want to concentrate programming on solving given problems such as user interface, communication protocols, information retrieval, and visualization of information. That is, they do not want to be overwhelmed by the complexities discussed in the introductory section. Also, to increase reusability, applications should not depend on the underlying hardware/environment.

For example, the process abstraction provided by the underlying realtime operating system is useful for application programming by dividing an application module into several processes. This makes programmers free from implementation of detailed process scheduling, process allocation to a processor, and reduces superfluous CPU cycles. However, each process consumes extra memory to keep its control information. This penalizes memory usage. Also, inter-process communication needs more instructions than intra-process communication to deliver a message to the target process. These create a gap between application modules and system processes. Thus, programmers sometimes have to write their own light-weight process facility to meet their needs.

This problem comes from a gap between ideal abstraction and its efficient implementation, particularly on distribution and heterogeneity. Assume we have ideal module granularity for a given problem. Here, we have at least three other dimensions of the interaction between modules i.e., *protection* to guard data from malicious access, *activity* to control instruction stream execution, and *storage* to keep data. In the implementation of the ideal module, we have to map that module onto these dimensions. In case of

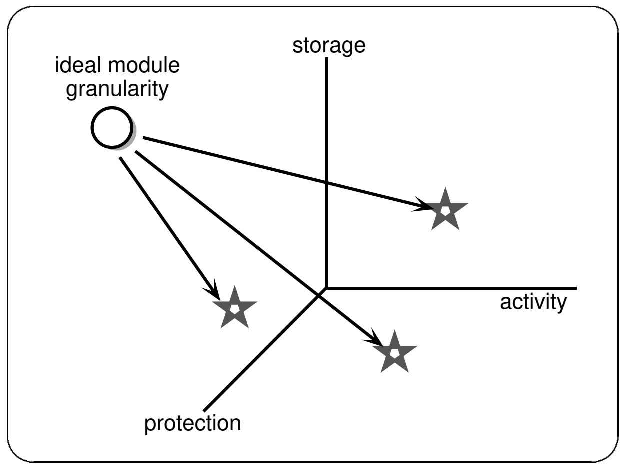

Figure 1: Mapping Ideal Module Granularity onto Actual Implementation

UNIX, an object can be mapped onto a process with multiple threads, where a thread is a unit of execution, object's storage is a process's virtual address space, and an object is protected by a process protection mechanism. Inter-process communication takes time, but we can reduce it by allocating two or more objects on the same process. In this case, the protection dimension is moved, i.e., object's protection is done by a compiler, or ignored.

Many systems provide only a single fixed pattern of mapping code onto these dimensions. In this situation, programmers are forced to try organize their program so that the mapping pattern will turn out well for their program. This is often impossible. Even if they succeed, the program is now dependent on the mapping used by the underlying operating system and hardware. This is a barrier to module reusability.

When, as in this case, no one mapping will be best for all programs, we call this a mapping dilemma[Kiczales and Lamping 93]. We can see mapping dilemmas in various fields such as realtime process scheduling, user-level thread management, page replacement in virtual memory systems, file placement, data placement on parallel processors, data dependent compression, network protocols, and a degree of causality. And, we can see solution of these mapping dilemma problems in each domain, including reflection systems such as [Stroud 92], [Banerji *et al.* 93], and [Okamura and Ishikawa 94].

What we want is separation of concerns: keep the notion of a basic function of a program that is independent of the other dimensions, and be able to write code in terms of that function, but also be able to control the distribution level concerns and heterogeneity level concerns. In the above example, if the mapping decisions of the implementation are properly exposed, they will allow us to create an optimal mapping to an efficient implementation. Programmers will just think about the mapping between ideal abstraction and efficient implementation after writing their applications. They will use another interface to the implementation that is called a meta-level interface. This gets simplicity, because the concerns are separated as much as possible, and reusability, because distribution and heterogeneity issues depending on the underlying execution environment can be altered without changing the base-level code.

3 The Apertos Architecture — How to Support Separation of Concerns

The Apertos operating system has a systematic way to support separation of concerns, particularly on distribution and heterogeneity. We have developed object/metaobject separation, which helps us to realize separation of concerns in Apertos and consists of the following three principles:

- An object has its own execution environment[1], which is given by a group of metaobjects.
- A metaobject is also an object.
- An object changes its execution environment to another.

With the first principle, an object has its optimal implementation to the underlying environment. Since we have the second principle, giving an object forms hierarchy of the object/metaobject relationship[2]. The last principle is interesting in the sense that it makes an object independent of the underlying environment and helps to support separation of concerns.

In Apertos, the base-level algorithms are written in a medium granularity object system of prosaic design. Distribution and heterogeneity issues are addressed by a meta-level. The meta-level reifies two kinds of things: execution environments and objects in those environments. Operations at the meta-level are *about* activity at the base-level, hence the name "meta." The execution environment determines the physical location of objects, heterogeneity issues including support for optimal object execution, and other issues, such as persistence and scheduling policies. The system is open-ended in that users can write new execution environments or specialize existing ones to support new machines or new policies.

Execution environments are united into cohesive whole via protocols, which are supported by all execution environments, for sending messages to objects in other environments and for the transfer of objects between environments. Thus objects in different environments can communicate and objects can migrate between execution environments to move them to a different machine or to get a change of policy.

In addition to these external protocols, execution environments also have internal protocols to access representations of the objects they contain and to intercept and introduce message sends. Execution environments can be customized by altering their implementation of these protocols. These internal protocols are analogous to the facilities provided by meta-object protocols for object-oriented programming languages [Kiczales *et al.* 91]. In Apertos, they serve the additional role of providing mechanisms needed by the external protocols.

[1] which we call a metaspace.

[2] which we call a metahierarchy.

4 Example

This section demonstrates how the Apertos architecture works well to support separation of concerns. Assume that we are developing an application for a settop box and servers for cablenets. A settop box is equipped with a high performance processor and a graphic engine. Servers are constructed by many distributed processors. An application is written by a collection of objects. Figure 2 is a possible structure of the base-level objects. The code fragments of a program is like followings. Expression for sending a message written in MC++, extension to metaobject programming, is:

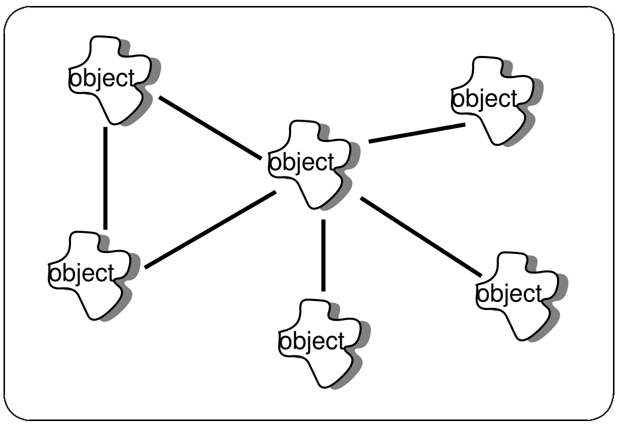

Figure 2: The Base-view of Some Objects

```
anEntry = myAccount -> Lookup ("keyword");
```

Expression for creating a new object is:

```
myAccount = new Account;
```

In these ways, there is no description of object location and heterogeneity when an application is written at the *base-level*.

At the *meta-level*, processors are assigned to these objects as shown in Figure 3. We can specify a processor for an operation at this meta-level. The processor assignment is done either when an object is created or when an object is explicitly moved to the specified processor. For example, the following expression creates a new object at the specified processor:

```
creator -> new (
        class of object A,
        processor A);
```

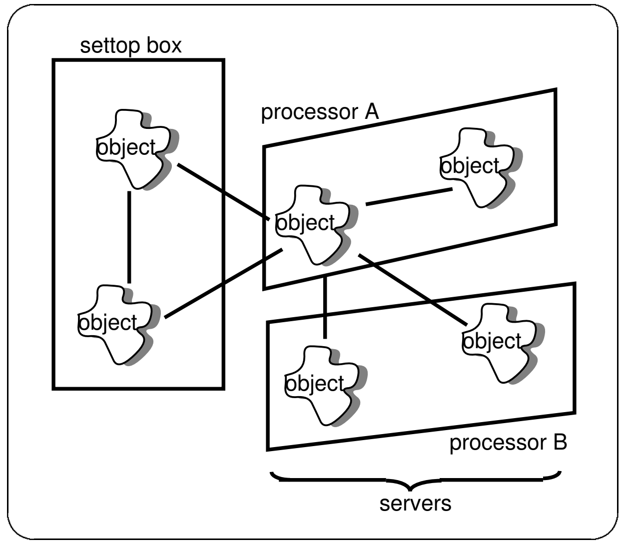

Figure 3: The Processor Allocation of the Objects

Here variable `creator` denotes a meta-object which is responsible for creating a new object of the class (the first argument) at the specified processor (the second argument). This meta-object forwards the request to `processor A` for object creation, i.e., on the specified processor, retrieving a template for the object, allocating a virtual address space, and pasting a program text and initial data on that space.

On a settop box, some objects should be persistent to record user's profile, operation history, etc., by evaluating the following expression at the *base-level*:

```
meta -> migrate (
        object A,
        metaspace supporting
                persistence);
```

Here, we introduce the object migration to make objects persistent. This is a concrete implementation of the last principle of object/metaobject separation (in Section 3), and an external protocol basically available to every object. Operation `migrate` accepts two arguments: one denotes an object to be moved and the other denotes execution environment (i.e. metaspace) making an object persistent. Object migration mechanism can be used for down loading an object to a settop box in order to install a new functionality and update software.

Figure 4 shows a *meta-level* structure of objects, where two objects are persistent. At the

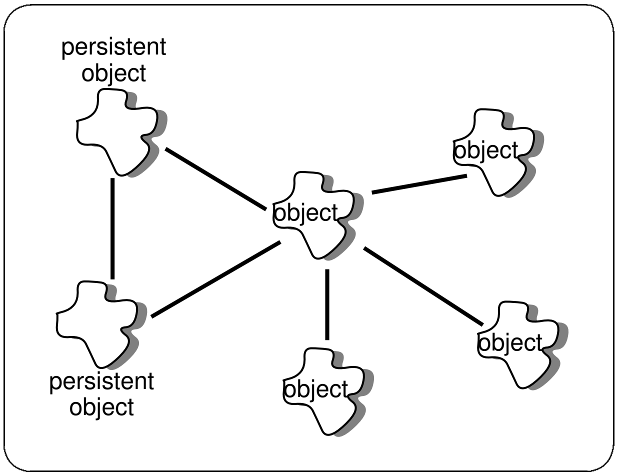

Figure 4: The Persistence of the Objects

meta-level, the way of transformation of the internals of an object is determined by migration protocols. We have introduced a descriptor at the meta-level, which maintains references to metaobjects representing some aspects (such as the number of activity inside of an object, the property of object's local storage, and processor capacity of object execution) of a base-level of object. Since an object is implemented by a collection of metaobjects at its meta-level, object migration protocols reconfigure the implementation of metaobjects. In Figure 5), when an object migrates to another execution environment (Figure (a)), a new descriptor is created in that environment and a group of metaobjects optimal for that object is constructed and referenced by the descriptor (Figure (b)). The following is a simplified code fragment of object migration protocol.

```
 1: Descriptor::Migrate (
 2:   Object*  pObj,
 3:   MCommon* pMeta)
 4: {
 5:   Descriptor* pOld;
 6:   Descriptor* pNew;
 7:   pOld = this -> GetDescriptor (pObj);
 8:   pNew = pMeta -> NewDescriptor ();
 9:   *pNew = *pOld;
10:   delete pOld;
11: }
```

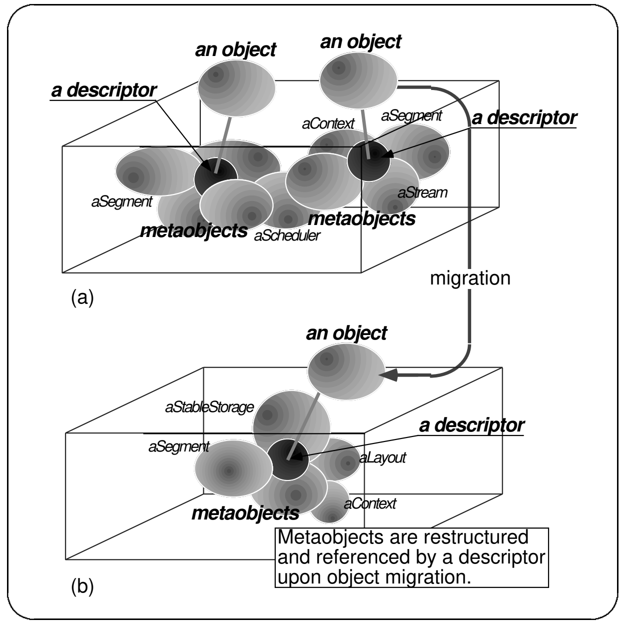

Figure 5: Restructuring Metaobjects upon Object Migration

Here, `Descriptor` is the abstract class, which defines the common meta-level representation of a base-level object, i.e., the minimum set of metaobjects to provide an object with execution environment. In the above example, `pOld` and `pNew` denotes concrete descriptors as subclasses of `Descriptor`. Line 9 is the body of a migration protocol, in which some metaobjects are created to make an object persistent, object's local storage is transferred to the new location, and the internal representation is converted if necessary. In this example, two objects are transferred to the servers, when a settop box has no facility of stable storage. Apertos object migration is powerful mechanism in heterogeneous environment, in the sense that it makes objects independent of their execution environment.

5 Implementation

The Apertos system is running on R3000 based Sony's NEWS workstations and i486 based PC-compatible machines. Object/metaobject separation is implemented by introducing the *Context* objects and their metaobject MetaCore. These provides objects with meta-level context management [Yokote 93] and define a virtual machine for implementing object/metaobject separation. Object's source programs are translated into the native codes and utilize object/metaobject separation with the instructions provided by this virtual machine.

We have successfully implemented the virtual machine to minimize the duration of operations inside the virtual machine. This helps us to implement a highly preemptable system for real-time computing. In the implementation, object *Context* is accessed by metaobjects and keeps CPU dependent information to execute object's methods, which includes the relationship between an object and its execution environment. *Context* is allocated at meta-level, i.e. outside of the kernel, so that its operation can be preemptable. The atomic operations supporting *Context* execution is implemented by MetaCore, which provides three time-predictable services and code size is about 2K bytes.

6 Summary

The discussion here can be applied to any fields such as programming for parallel computers, programming to support mobile computers, and programming network virtual reality. Separation of concerns is software technology/programming strategy. We conclude that operating systems and programming languages should support that strategy in order to match them to application needs.

Acknowledgments

A number of people have contributed to this work and our understanding of it. In particular we would like to thank: Mario Tokoro, Director of Sony Computer Science Laboratory Inc., Mark Weiser, Andreas Paepcke, Brian Smith, Dylan McNamee, Mike Dixon, and Andy Berlin of Xerox PARC, Chris Maeda of Carnegie Mellon University, Takao Tenma, Nobuhisa Fujinami, and Jun-ichiro Ito, members of the Apertos project, Sony Computer Science Laboratory Inc.

References

[Banerji *et al.* 93] Arindam Banerji, Dinesh Kulkarni, John Tracy, and David Cohn. The Substrate Object Model And Architecture. In *Proceedings of the 3rd International Workshop on Object Orientation in Operating Systems*, December 1993.

[Kiczales and Lamping 93] Gregor Kiczales and John Lamping. Operating Systems: Why Object-Oriented? In *Proceedings of 3rd International Workshop on Object-Orientation in Operating Systems*. The Computer Society of the IEEE, December 1993.

[Kiczales *et al.* 91] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. The MIT Press, 1991.

[Okamura and Ishikawa 94] Hideaki Okamura and Yutaka Ishikawa. Object Location Control Using Meta-level Programming. In Mario Tokoro and Remo Pareschi, editors, *LNCS 821, Object-Oriented Programming, Proceedings of ECOOP'94*, July 1994.

[Stroud 92] Rober Stroud. Transparency and Reflection in Distributed Systsms. In *Proceedings of the 5th ACM SIGOPS Workshop on Models and Paradigms for Distributed Systems Structuring*, September 1992.

[Yokote 92] Yasuhiko Yokote. The Apertos Reflective Operating System: The Concept and Its Implementation. In *Proceedings of Object-Oriented Programming Systems, Languages and Applications in 1992*. ACM Press, October 1992. Also appeared in SCSL-TR-92-014 of Sony Computer Science Laboratory Inc.

[Yokote 93] Yasuhiko Yokote. Kernel Structuring for Object-Oriented Operating Systems: The Apertos Approach. In *the Proceedings of the International Symposium on Object Technologies for Advanced Software (ISOTAS)*. Springer LNCS, 1993. Also appeared in SCSL-TR-93-014 of Sony Computer Science Laboratory Inc.

[Yokote *et al.* 89] Yasuhiko Yokote, Fumio Teraoka, and Mario Tokoro. A Reflective Architecture for an Object-Oriented Distributed Operating System. In *Proceedings of ECOOP'89 European Conference on Object-Oriented Programming*, July 1989. Also appeared in SCSL-TR-89-001 of Sony Computer Science Laboratory Inc.

[Yokote *et al.* 91] Yasuhiko Yokote, Fumio Teraoka, Atsushi Mitsuzawa, Nobuhisa Fujinami, and Mario Tokoro. The Muse Object Architecture: A New Operating System Structuring Concept. *ACM Operating Systems Review*, Vol. 25, No. 2, pp.22–46, April 1991. Also appeared in SCSL-TR-91-002 of Sony Computer Science Laboratory Inc.