[20] M. Shapiro, Y. Gourhant, S. Habert, L. Mosseri, M. Ruffin, and C. Valot. Sos: An object-oriented operating system—assessment and perspectives. *Computer Systems*, 2(4):287–338, Dec. 1989.

[21] R. Snodgrass and K. Shannon. Supporting flexible and efficient tool integration. In *Proceedings of the International Workshop on Advanced Programming Environments*, pages 290–313, Trondheim, Norway, jun 1986. IFIP WG 2.4, Springer-Verlag.

[22] C. P. Thacker, L. C. Stewart, and E. H. Satterthwaite. Firefly: A multiprocessor workstation. *IEEE Transactions on Computers*, 37(8):909–920, Aug. 1988.

[23] M. V. Wilkes and R. M. Needham. The Cambridge model distributed system. *OSR*, 14(1):21–29, Jan. 1980.

[24] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. Hydra: The kernel of a multiprocessor operating system. *Communications of the ACM*, 17(6):337–345, June 1974.

[25] W. A. Wulf, R. Levin, and S. P. Harbison. Hydra/c.mmp: An experimental computer system, 1981.

[26] M. Yudkin. Resource management in a distributed system. *Proceedings of the Eighth Data Communication Symposium*, pages 221–226, Oct. 1983.

[4] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):472–522, Dec. 1985.

[5] E. W. Dijkstra. The structure of "THE"-multiprogramming system. *Communications of the ACM*, 11(5):341–346, May 1968.

[6] P. Druschel, L. L. Peterson, and N. C. Hutchinson. Decoupling modularity and protection in Lipto. Technical Report 91-6, Department of Computer Science, University of Arizona, Feb. 1991.

[7] A. E. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison Wesley, Reading, MA, 1990.

[8] R. G. Guy, J. S. Heidemann, W. Mak, T. W. Page, G. J. Popek, and D. Rothmeier. Implementation of the ficus replicated file system. In *Proceedings USENIX Summer '90 Conference*, pages 63–71, June 1990.

[9] A. Habermann, L. Flon, and L. Cooprider. Modularization and hierarchy in a family of operating systems. *Communications of the ACM*, 19(5):266–272, May 1976.

[10] N. C. Hutchinson and L. L. Peterson. The *x*-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, Jan. 1991.

[11] Intel Corporation, Santa Clara, California. *System 432/600 System Reference Manual*, 1981.

[12] B. W. Lampson. Protection. In *Proc. Fifth Princeton Symposium on Information Sciences and Systems*, pages 437–443. Princeton University, Mar. 1971.

[13] M. A. Linton, J. M. Vlissides, and P. R. Calder. Composing user interfaces with InterViews. *Computer*, pages 8–22, Feb. 1989.

[14] J. Nestor, J. Newcomer, P. Giannini, and D. Stone. *IDL: The Language and Its Implementation*. Prentice-Hall, Englewood Cliffs, NJ, 1990.

[15] D. L. Parnas. On the critera to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, Dec. 1972.

[16] R. K. Raj and H. M. Levy. A compositional model for software reuse. *The Computer Journal*, 32(4):312–322, Aug. 1989.

[17] D. M. Ritchie. A stream input-output system. *AT&T Bell Laboratories Technical Journal*, 63(8):311–324, Oct. 1984.

[18] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. Chorus distributed operating systems. *Computing Systems Journal*, 1(4):305–370, Dec. 1988.

[19] M. Shapiro. Structure and encapsulation in distributed systems: The proxy principle. In *Proceedings of the Sixth International Conference on Distributed Computing Systems*, pages 198–204, May 1986.

is similar to the approach taken by Raj and Levy [16] at the language level in the Emerald programming language.

The mechanisms provided for this purpose are separation of interface and implementation and inheritance of interfaces. Together with the system directory and the module/object infrastructure, which implements location transparence, these mechanisms provide the ability to compose a dependency graph of modules that implement a service.

This compositional approach to software reuse has important advantages over implementation inheritance when applied to operating systems. First, it extends naturally into a distributed environment, because objects interact through invocations only. Second, it simplifies language heterogeneity, because the only externally visible attribute of an object is its procedural interface. Third, service composition can be done at run-time and without access to a module's source code; inheritance, on the other hand, is a compile-time mechanism and generally requires access to source code.

5.4 Service Composition

Composition of building blocks has been used in toolkit-based programming environments [14 , 21], for the construction of user interfaces [13], and other similarly specific application domains. In System V Unix, device drivers and protocols can be composed using the Streams I/O system [17]. In all these cases, composition is restricted to a specific application domain. The Unix shell allows the composition of naive utility programs to form a chain of filters connected by pipes. However, filters can only interact in a trivial, unidirectional fashion.

Lipto's architecture provides a unifying architectural framework for building composable subsystems like the ones mentioned above, allowing the definition of many interfaces. However, Lipto goes beyond this level of flexibility. First, it allows the composition of basic operating system services. Second, applications can compose services using a mixture of their own modules, system-provided modules, and modules provided by third parties. Moreover, system and third-party modules can be composed to depend on application-provided modules. This facilitates the separation of policy and mechanism; in particular, it allows applications to provide policy modules for system-provided mechanisms. Third, the modules that constitute a composed service can be distributed across machines and protection domains. As far as I can determine from available literature, this aspect of Lipto's architecture is new.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for Unix development. In *Proceedings of Summer Usenix*, July 1986.

[2] T. Anderson, H. Levy, B. Bershad, and E. Lazowska. The interaction of architecture and operating system design. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Apr. 1991.

[3] R. H. Campbell, V. Russo, and G. Johnston. Choices: The design of a multiprocessor operating system. In *Proceedings of the USENIC C++ Workshop*, pages 109–123, Nov. 1987.

such as the ioctl and fcntl system calls.

The BSD Unix socket interface, on the other hand, was designed as a general interface for the access of network services at different levels. Its generality, however, not only makes it difficult to use, but also leads to paradoxical cases where the use of a lower level network service is more expensive than the use of a higher level service. For example, in SunOS 4.0 the protocol suite UDP/IP/ETH performs better than the suite IP/ETH [10].

Lipto's architecture encourages the modular decomposition of system services, allowing and requiring the definition of many more interfaces. Each kind of service (File service, Network communication service), and each level of service of the same kind (Datagram, Stream, Remote procedure call) define their own set of interfaces in the form of a service class. An application simply accesses a service at the desired level, using the appropriate service class. Thus, a naive application with a very general requirement for the type of services it needs uses a general, high-level service class. A sophisticated application, on the other hand, accesses its services using a lower-level, more specific service class.

5.3 Object-Orientation

While Lipto's architecture is object-oriented, application programmers and, more generally, module implementors can use any implementation language, i.e., Lipto is language independent from the module implementor's perspective. In the case where an object-oriented application language is used, Lipto does not impose an object model for application level objects. Moreover, Lipto does not provide an object management layer that supports the creation, migration and storage of such objects. It is my view that these services should be provided at the level of a language run-time system or by an object support layer on top of the basic system services. This and other features distinguish Lipto from other object-oriented systems, such as Choices and SOS [3, 20].

Choices is a truly object-oriented operating system in that its architecture is object-oriented, it exposes an object interface to applications and it is implemented using an object-oriented language. Since the system is structured using the inheritance mechanism and the object model of its implementation language, C++ [7], its architecture is tightly coupled with this language. The system's services can be extended through the inheritance mechanisms of C++, however, only applications written in C++ can take advantage of this feature. Moreover, Choices is not a distributed system, and since its architecture uses inheritance of implementations, it does not readily generalize into a distributed environment.

SOS is a distributed object-oriented operating system that emphasizes the support of object-oriented application environments. It provides support for object migration and persistence based on a system-defined object model. Application level objects that want to use the system's services must conform to this model. The necessary translation between the representation of language-level objects and the system object representation can cause inefficiencies. Since the system object model is too heavy-weight for fine-grained objects, applications have to deal with two kinds of objects, those that are system-supported and those that are not.

The traditional approach to software reuse in object-oriented software systems is to support inheritance of implementation. Both Choices and SOS take this approach. Lipto's architecture abandons implementation inheritance and supports instead a model for software reuse through composition. This

Topaz, and Version 3 of Mach [1, 18, 22]. Server-based systems attempt to find a balance between modularity and performance by supporting coarse-grained protection and modularity, with the granularity at the level of the protection domain. Each service is allowed to define its own interface. The interfaces are implemented on top of a small, fixed set of interfaces to the kernel's primitive communication services. However, crossing an interface always requires crossing a protection boundary.

Lipto's architecture decouples modularity and protection by providing explicit support for modules independent of protection domains. Consequently, large software systems can be decomposed into modules without concern for inter-domain communication costs. The right tradeoff between protection and performance can be determined along a broad scale according to the hardware characteristics and application requirements by grouping modules into protection domains at configuration time. At one end of the scale, the system can be configured to behave like a capability-system with fine-grained protection; at the other extreme, it can be configured as a highly efficient system without protection at all. Moreover, the configuration can change during the software life cycle of the system. During debugging and validation, the system could be configured with fine-grained protection, i.e., with each object in a separate protection domain. Later, the system can be reconfigured according to the needs for protection and efficiency in a particular installation. The same ideas may be used separately for a new subsystem that is added to an existing system. Location transparence also makes certain performance improvements possible. For example, frequently called system services can be hoisted into the user-level domains of certain trusted applications. Also, certain objects or subsystems normally implemented in user-level domains can be placed into the kernel address space for performance if that is acceptable from the standpoint of protection.

5.2 Service Access

In traditional operating system architectures, all operating system services are accessed through a single, general, static system call interface. I believe that the traditional system call interface designs have three major flaws. First, they are only available at protection boundaries. This implies that it is difficult to layer new services atop existing ones. Second, it is difficult to modify the interface as the system evolves. Additions to the interface can be handled in a backwards compatible way, but modifications to the interface, or the introduction of alternate implementations of a service are problematic. Furthermore, the addition of new hardware to the system, which would most naturally be expressed by the addition of a new service interface, must be handled by less structured means such as the Unix ioctl call. Third, it is difficult to provide direct access to lower level services. Naive applications require insulation from the details of the hardware on which they are executing, but sophisticated applications (such as databases and real-time audio and video) require low level access to the hardware in order to achieve acceptable performance.

One of the innovations of the Unix operating system is its uniform treatment of files and devices; a general, uniform interface (open, close, read, write, seek) is provided for such access. Together with the convention for the usage of the file descriptors stdin, stdout and stderr, it is the key to the composability of naive applications such as filters. This Unix file interface is in fact an example of what would be called a service class in Lipto. The problem is that in Unix access to lower level file system features and features that are specific to certain devices are only accessible through awkward, unstructured interfaces

can be efficiently passed as arguments. In the case of a remote invocation, the proxy object converts the reference to one that is valid in the callee's domain. This is entirely transparent to both caller and callee and requires additional overhead only in the remote case.

Note that Lipto's object invocation mechanism is optimized for the local, i.e., intra-domain case. In this respect, it differs from existing location-transparent object invocation mechanisms. The resulting efficiency of local object invocation encourages a fine-grained decomposition of software subsystems, both at the system and the application level. b

5 Related Work

In this section I will discuss my proposal in the light of related work in the literature.

5.1 Modularity and Protection

Modularization as a technique to decompose and structure operating systems has been known for a long time [15, 5, 9]. Protection provides for isolation so that a failure or malice in one component of a system cannot adversely affect the operation and integrity of another component [12]. Such failure isolation is desirable whenever two components of a system enjoy different levels of trust, such as between two user components or between user and system components of a general purpose operating system. Protection requires implementing domains that are separate from each other, as well as safe mechanisms for cross-domain communication.

A review of previous operating system architectures indicates that modularity and protection have been implemented by a single mechanism, that is, they have been tightly coupled. At one end of the spectrum, several systems have employed capabilities as their modularity/protection mechanism. Capabilities may either be implemented in hardware as in the Cambridge Cap [23, 26] and the Intel iAPX432 [11], or in software on top of standard paged virtual memory hardware as in Hydra on C.mmp [24, 25]. Whether implemented by hardware or software, such systems offer fine grained modularity/protection, but limited flexibility since a single mechanism is used for both functions. Interfaces may be defined by the programmer, but every invocation through an interface requires crossing a protection boundary as well, resulting in poor performance.

At the other end of the spectrum, Unix is a classic example of a monolithic system. Both modularity and protection are supported only at a very coarse grain by the distinction of kernel and user contexts, where a single fixed invocation interface provides access to all system services. All services provided by the operating system are implemented in a single protection domain, and cannot be protected from each other. Moreover, the code contained in this single address space uses shared/global data, and thus provides almost no encapsulation. In such systems, both modularity and protection are sacrificed for speed.

In reaction to the maintenance problems caused by monolithic systems, and in order to provide more flexibility for implementing new kinds of services, several server-based systems are being implemented. These systems are characterized by a small kernel or *micro-kernel* that provides only basic communication services, and multiple servers providing all other services. Examples of such systems include Chorus,

The set of modules that participate in the implementation of a service form a dependency graph. The module at the root presents the service to its clients; the modules at the leaves are special in that they do not depend on any lower-level services. A service is composed by building the graph of modules bottom-up. In the case of a module that provides multiple *instances* of a service, a different graph can be composed for every instance of the service. Consider for example a module that implements a network file system and assume that the module depends on a suite of network protocols. The module can be composed to use a different suite of protocols for each file or directory, depending on the location and type of the file server that backs the file/ directory.

The architecture explicitly separates interface and implementation. A service class defines abstract interfaces for objects types; modules provide various implementations for these object types. Service classes form a hierarchy according to a conformity relation. The architecture supports inheritance of interfaces but not inheritance of implementations. This approach has several important advantages. In the absence of implementation inheritance, the only interaction between objects defined in different modules is through the invocation of each other's operations. Note that any two objects that are implemented in different modules may at run-time reside in different protection domains, and possibly on different machines. Lipto can support location independence merely through location-transparent invocation between objects. It implements this mechanism using the technique of *proxy objects* [19]. The lack of implementation inheritance also makes it easier to provide language heterogeneity for module implementations, because the external knowledge about modules is restricted to its set of interfaces. Also, Lipto's architecture allows the reuse of a module without access to its source code.

Polymorphism, or more specifically, *inclusion polymorphism* [4] is supported through the separation of interface and implementation, and additionally through inheritance of interfaces. A client that expects a service in a service class S can use any module that implements either S itself or any service class that is a descendant of S in the hierarchy of service classes. The formal parameters defined in an interface's operations can specify object types in terms of their interfaces. Any object type with the appropriate interface can serve as an actual argument.

Available services may be located by clients using the *system directory*. Services register with the system directory using a unique *service id*. A client inquires about a service by presenting a service id and the expected service class to the system directory. If the requested server object can be located and it is in the appropriate service class, then a *reference* for the server object is returned. Depending on the location of the server object, this reference refers either to the server object itself, or to a proxy object that represents the server object in the client's address space. The tasks of locating the server object, authentication and binding happen at the time when the object reference is obtained from the system directory.

The use of proxy objects and the implied implementation of object references as pointers is one of the keys to Lipto's performance. Object invocations are implemented by simply fetching a function pointer from a table indexed by the operation number, and calling the function. If the client and the server object are in the same domain, then this is the cost of an object invocation. In the cross-domain case, the proxy object forwards the invocation to the server object using the underlying communication mechanism. Since authentication and binding are performed when the reference is created, the invocation can be as fast as the communication facility permits. Because object references are simple pointers, they

adjusted at configuration time.

First, if modules can be implemented regardless of the protection domain in which they will eventually reside, i.e., their implementation is location-transparent, then the assignment of modules to protection domains becomes a matter of configuration. That is, protection, failure isolation, access control and performance can be traded off according to the needs of the application at module configuration time. What is necessary to achieve this degree of flexibility is architectural support for modularity that is independent of protection domains. The architecture must provide interfaces and communication mechanisms that allow the location-independent interaction of objects. Thus, interfaces and communication endpoints must be provided independently of protection domains. In terms of the fundamental abstractions, this means that communication endpoints and interfaces must not be coupled with protection domains; instead, these abstractions must be associated with modules.

Second, if modules are implemented as passive, procedural code that is executed by independent threads of control, then the level of concurrency can be controlled by the number of threads allowed to execute the code concurrently, which can also be established at module configuration time. What this means in terms of the fundamental abstractions is that threads and protection domains must be orthogonal abstractions, and that communication endpoints must not be tied to threads.

I conclude that execution, protection, and modularity are orthogonal and therefore an operating system should provide support for modules independent of threads and protection domains. Lipto supports modularity through the abstraction of a *module*, which is the unit of composability and configurability. A module/object infrastructure provides services to dynamically load and locate modules and implements location-transparent invocation based on communication endpoints provided by the nugget.

4 Architecture

This section presents an overview of Lipto's architecture. A more detailed description is given in [6].

Lipto's architecture is object-oriented. Its objects encapsulate state, and export a set of operations. Although the architecture is object-oriented, the implementor of individual modules can use the programming methodology and language of her choice, as long as the interfaces defined in the module's service class can be supported. All of the infrastructure necessary for the communication between objects and the composition of services is defined by the architecture.

A module provides the implementation or *behavior* for one or more types of objects. The object types defined by a module collectively provide a *service*. All objects are passive, i.e., they export procedural interfaces and their operations are executed by independent threads of control.

The composition of services from modules is governed by abstract interface definitions called *service classes*. When a module's object types implement the interfaces defined in a certain service class it is said to implement that service class. Each module implements exactly one service class, but many modules can implement the same service class, i.e., the same interface can be supported by many modules. In fact, when many modules implement the same service class, a high degree of composability results. The implementation of a module will generally depend on a set of lower-level services. A module specifies the lower-level services it needs in terms of a set of service classes. The module can be composed on top of any set of modules that are in the appropriate service classes.

argue that execution, protection and modularity are also orthogonal, and therefore modularity should be supported through a mechanism that is independent of protection and execution.

A *thread* is the unit of execution. The level of concurrency in the execution of a program can be controlled by the number of threads allowed to execute that program concurrently. The appropriate level of concurrency depends on the nature of the program as well as the characteristics of the hardware platform on which the program executes, for example the number of processors. A *protection domain* is the unit of resource allocation, protection and accounting. Protection domains are used to isolate modules of a program from each other. Such isolation may be necessary if the program modules do not trust each other, if the failure of one module must not affect other modules, or if the modules need to be treated differently with respect to resource access rights or accounting. The unit of communication is a *communication endpoint*. Communication endpoints allow run-time entities or objects that reside in different protection domains to communicate with each other.

Assuming that these basic abstractions are provided by a *nugget*, and given the goal of a modular architecture built on top of this nugget, let us consider how one would map modules onto these abstractions. One approach is to map modules onto protection domains. This seems like a natural mapping, because modules are protected from each other and access rights can be given to individual modules. Furthermore, it seems natural to define communication endpoints and interfaces at protection boundaries. With this approach, however, all module interaction involves inter-process communication, and thus imposes communication overhead proportional to the number of modules that constitute a service. Communication between different protection domains is inherently more costly than communication within a protection domain [2]. Consequently, there is a tradeoff between protection, failure isolation and fine-grained access control on one hand and performance on the other hand. The right tradeoff depends on the requirements of the program's users with respect to protection and failure isolation, the stability of the program code (debugging vs. post-release phase) and the characteristics of the hardware platform (distribution, IPC performance). If modules are mapped onto protection domains, then software designers are faced with a choice: They can either decompose at a fine grain, thereby gaining modularity at the cost of performance; or they can decompose at a coarse grain for good performance at the cost of modularity. This tradeoff is part of an early design decision, and cannot be adapted according to the needs of the users and the characteristics of the hardware.

Another possibility is to map modules onto threads. Here, multiple modules can share a protection domain, and the tradeoff between protection and performance can be delayed until module configuration time. The implementation of modules is simplified, because their code is executed exclusively by a single thread. However, each invocation of a module involves at least a context-switch among threads, which is a relatively expensive operation compared to a procedure call. Even worse, the level of concurrency in the execution of a module's code cannot be adapted to the characteristics of the hardware.

The right approach is to map a module onto a communication endpoint. In terms of the basic abstractions, a communication endpoint is all that is needed to support modularity. The architectural infrastructure implements the necessary conversion between the passive, procedural interface supported by the module and the communication facility, which is provided by the nugget. If modules are mapped onto communication endpoints in a system where threads, protection domains and communication endpoints are provided as orthogonal abstractions by the nugget, then both protection and concurrency can be

designed service classes (module interfaces) must be defined that allows a high degree of composability of modules. Third, large operating system services must be decomposed into modules in such a way that their decomposed implementation compares well with a monolithic implementation in terms of performance.

The dynamic composition of services from modules raises two other issues. In any composition where a module depends on another module that it does not trust, care must be taken that the failure of the untrusted module does not cause the failure of the client module. Furthermore, since dependency relations among modules change dynamically due to composition, the deadlock problem must be addressed accordingly. I will not cover these issues in this proposal.

As stated earlier, the primary goals of my architecture are dynamic composition of services and portability. A vital requirement for the feasibility of my approach is overall efficiency. Of particular importance is the efficiency of module interaction in the case where modules are in the same protection domain. If communication costs between modules in this case were significant, system and application designers would be discouraged from decomposing their systems, thereby defeating the purpose of the architecture. In other words, it must be possible to implement a decomposed system that performs as well as a monolithic implementation when its modules are configured to reside in a single protection domain.

The most important assumption underlying Lipto's architecture is that complex software systems can be decomposed into small reusable modules. Currently, no methods exist for the mechanical decomposition of large software systems, and the process remains somewhat of an art. However, previous work exists that strongly suggests that decomposition into relatively small, composable modules is feasible and can be efficient, given a suitable architecture and communications infrastructure. The goal of Lipto is to provide just that. Examples of work in the area of decomposition are the *x*-kernel [10] in the case of the communication subsystem and Ficus [8] in the case of the file system. Note that Lipto's architecture does not impose a particular granularity of decomposition. The granularity used is up to the designers of a particular set of modules. However, it is clear that composability will increase with the number of modules and the generality of the abstractions that the modules implement.

The architecture is designed to be independent of the implementation language used; in fact, individual modules can be implemented in different languages. The only constraint on the implementation language used is that it allows the expression of external interfaces as defined in a module's service class.

3 Decoupling Modularity, Protection, and Execution

In this section, I will argue that an operating system should provide explicit support for modularity that is independent of protection and execution. This idea, which I call the *decoupling principle*, underlies the design of Lipto's architecture and is the key to its flexibility. In its most general form, the decoupling principle states that orthogonal concepts arising in an operating system should be provided through orthogonal abstractions. Stated differently, orthogonal concepts should not be coupled in a single abstraction. The principle applies to many levels of an operating system. At the level of the fundamental services of a general-purpose operating system, it states that execution, protection and communication (which are clearly orthogonal) should be provided through orthogonal abstractions. This aspect of the principle has been observed by others. Consequently, several recent operating systems provide separate abstractions for these services, e.g. threads, tasks/actors, and ports [1, 18]. In the following, I will

management, file system and communication facilities, the goal of this research is to design and experiment with a new operating system *architecture*. In other words, this work focuses on the structure of an operating system and the interaction of its components rather than the design and implementation of its components.

The rest of this paper is organized as follows. Section 2 discusses motivation, goals and issues that need to be addressed. Section 3 justifies the decoupling of modularity, protection, and execution, an idea that underlies Lipto's architecture. Section 4 gives a brief overview of Lipto's architecture, and Section 5 discusses related work.

2 Motivation, Goals and Issues

The ever-increasing demand of applications for more specialized services calls for a new approach to the way an operating system provides services to applications.

The traditional approach is for an operating system to provide a fixed set of services. Demands for specialized services are satisfied by adding options to the existing services. This approach has several significant drawbacks. First, it obscures the interfaces to the system's services, making their use awkward and error-prone. Second, the never-ending addition of options leads to excessive complexity in the implementation of the operating system. Consequently, it becomes increasingly difficult to modify, debug and maintain the system's code. Moreover, the addition of options and thus complexity often leads to a degradation of overall performance, which also affects applications that do not make use of the new functionality. Third, the degree of flexibility in the use of the system's services is restricted to the set of options that the designer decided to provide.

Another problem with the traditional approach is that it provides services at a single, high level of abstraction. While a high-level view of the system's services is appropriate for many applications, there are applications that want to access services and devices at a lower, more specific level. For example, it does not make sense for a full-screen editor to open its input terminal as if it were a file, because the editor wants to make use of the special characteristics of a terminal, such as the ability to position the cursor. In Unix, an editor is forced to open the terminal as if it were a file. To access terminal-specific functionality, the editor has to use the ioctl system call with its highly unstructured, poorly defined interface, which is a major source of portability problems.

The approach I am proposing is to provide a collection of building block modules that can be used to compose a variety of services. New functionality can be provided by adding new modules. Such additions do not increase the complexity of existing modules, neither do they have adverse effects on the performance of existing services. Applications can access services at the most appropriate level of abstraction, thus eliminating the need for unstructured interfaces. Finally, the building block approach offers a potentially much higher degree of flexibility, because existing modules can be composed with new modules to provide arbitrary services, services that the designers of the existing building blocks have not intended or even imagined.

On the downside, there are challenges in realizing this approach. First, an infrastructure is needed that provides the necessary flexibility for the dynamic composition of services and still allows for an efficient implementation. This is the main subject of the work I am proposing. Second, a set of well-

1 Overview

In this paper I propose the design of an object-oriented architecture for a family of portable, distributed operating systems, and the implementation of an experimental prototype called Lipto. Lipto's architecture facilitates the *dynamic* composition of distributed services and applications from a set of building blocks or *modules*. My approach is based on two fundamental premises. The first premise is that a modern operating system should allow applications to dynamically compose higher-level services from a set of primitive building blocks. Some of these modules can be system-provided and others may be supplied by applications and third parties. This idea is motivated by the fact that many sophisticated applications have their own specific requirements with regard to the operating system services they need. Because of the diversity of requirements it has become impractical to define a fixed set of operating systems services that can satisfy all needs.

For example, applications may need to use different protocol suites depending on the remote entity with which they want to communicate; some applications require various levels of fault-tolerance support and others do not; a naive application is satisfied with the illusion of infinite virtual memory, while a more sophisticated application may need control over page replacement policies for its virtual memory; some applications are satisfied with simple, Unix-like sequential files, while others need database support and/or control over file caching policies. It is my view that applications should be offered exactly the services they need and should not have to pay a performance penalty for additional functionality for which they have not asked. Likewise, they should not be exposed to complexity that is not relevant to the service they need.

The second premise reads that a modern operating system must be readily portable across a variety of hardware architectures. In particular, the system must be able to exploit the potential of a multiprocessor as well as a uniprocessor, and that of a distributed system as well as a centralized system.

The solution I am proposing is to implement the system as a distributed collection of composable modules. Applications may add to, delete from, and replace modules in the pool of available modules and can compose arbitrary services from this pool, subject only to protection and security constraints. This enables sophisticated applications to compose services that exactly fit their needs from their own and/or system provided modules. Lipto employs a model of computation where execution, protection and modularity are orthogonal. Consequently, modules can be implemented without regard to their location and the level of concurrency in their execution. In this paper, I use the term *configuration* to refer to the assignment of modules to machines and protection domains. The term *composition* stands for the process of associating a set of modules so that they collectively provide a service.

Users can configure their applications, i.e., load their modules into protection domains on different machines, according to the application's needs for protection, concurrency and fault-tolerance. Likewise, system and application modules can be distributed across machines and protection domains to match the characteristics of the underlying computer system with respect to its parallelism, distribution and communication costs. The architecture defines an infrastructure that provides location transparence at the granularity of modules. That way, services can be composed dynamically regardless of the configuration of the system, that is, the assignment of modules to machines and protection domains.

Instead of proposing a new operating system with a specific design of its virtual memory, process

A Compositional Architecture for Portable, Scalable Distributed Operating Systems

Dissertation Proposal
Peter Druschel

TR 91-19

Abstract

The design of an object-oriented architecture for distributed operating systems is proposed that allows applications to dynamically compose distributed services from a mixture of system, application, and third-party provided software components. By decoupling execution, protection and modularity, the architecture supports the configuration of scalable, distributed operating systems, and ensures portability over a wide range of hardware platforms.

December 17, 1991

Department of Computer Science
The University of Arizona
Tucson, AZ 85721