representation — again a source of inefficiency.

We stress the efficiency and flexibility of the communications services provided by Lipto. Our design makes no assumptions about the size, homogeneity and the kinds of protocols used in the underlying network, nor do we assume that all the nodes in this network run Lipto.

References

[1] B. Bershad, T. Anderson, E. Lazowska, and H. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, Feb. 1990.

[2] A. Black, N. C. Hutchinson, E. Jul, H. M. Levy, and L. Carter. Distribution and abstract types in Emerald. *IEEE Transactions on Software Engineering*, SE-13(1):65–76, Jan. 1987.

[3] 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.

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

[5] 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.

[6] N. C. Hutchinson, L. L. Peterson, M. Abbott, and S. O'Malley. RPC in the *x*-Kernel: Evaluating new design techniques. In *Proceedings of the Twelfth ACM Symposium on Operating System Principles*, pages 91–101, Dec. 1989.

[7] 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.

a cross-domain call. Note that the "protection boundary" in this figure could correspond to a machine boundary as well as an address space boundary.

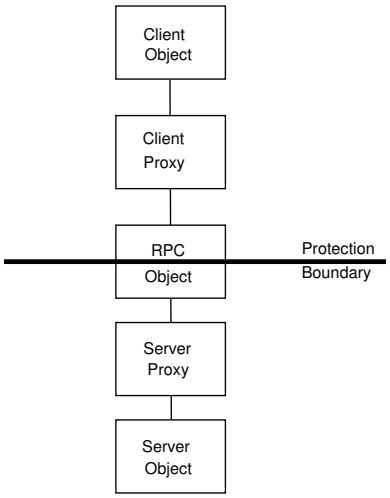

Figure 2: Proxy Objects and RPC

In other words, the fact that the server object is not local remains hidden from the client object. As a result of this location transparency, protection domain boundaries can be drawn arbitrarily, without affecting object implementations. Furthermore, an RPC object—which is itself a configurable service in Lipto—is used to cross these protection boundaries.

Concluding Remarks

Lipto is a dynamically configurable, object-oriented operating system kernel. Its 'unit of configurability' is a service. Our design does not impose an object model for application level objects. Moreover, our kernel does not provide an object management layer that supports the creation, migration and storage of such objects. Instead, it is our policy that distributed applications should be implemented in a language that supports distributed and persistent objects. One such language is Emerald. Lipto provides the communication services which permit the efficient implementation of such objects.

We think that we have good reasons for this design decision. Clearly, the services provided by the kernel should be language independent. However, every object-oriented language uses a different object model. A system service that supports object migration and storage must be able to identify all of an object's components and know which of them have to be moved. If the system imposes an object model, then the language either has to adapt this object model in its implementation — potentially at the price of losing efficiency — or it has to convert objects between the language representation and the system

interface defined in service class *A*. Then any object that implements this downcall interface—assuming that *O* supports the corresponding upcall interface of service class *A*—can act as *O*'s server in some syntactically valid configuration.

An object can implement abstract interfaces from one or more service classes. Typically, an object supports a single downcall interface from one class, representing the operations it offers to its clients. In the implementation of these operations, the object in general has to invoke lower-level server objects using downcall interfaces defined in different classes. The object therefore has to provide the appropriate upcall interfaces defined in its server object's service classes. Note that two objects do not have to support the same *upcall* interfaces in order to be interchangeable; they can rely on different lower-level services.

Location Transparency

Object implementations are *location transparent* in the sense that an object can be instantiated in any protection domain on any machine of the appropriate architecture throughout a distributed computer system. The object's implementation is not aware of any machine or protection domain boundaries. This flexibility can be restricted for various reasons. For example, if an object's implementation is not trusted, the object may be restricted to live in a non-privileged protection domain.

When a service becomes available, it registers its unique *service id* with the *system directory*. A client inquires about a service by presenting a service id and the type of the abstract interface that the client object expects to the system directory. If the requested server object can be located, it supports the expected interface, and the client supports any upcall interface that the server object expects, then a *reference* for the object is returned. The client can use this reference to invoke only those operations defined in the requested abstract interface.

If an instance of the service lives in the client's address space, the system directory returns a reference to that instance. If instead the system directory finds that the requested server object lives in a protection domain other than the client object's, then it instantiates a *client proxy object* in the client's domain [7]. This proxy object implements the required abstract interface. Note that a proxy object implementation is provided for every abstract interface of a class. Thus, the implementor of a service need not be concerned with the implementation of proxy objects.

The client proxy object forwards invocations to the remote object using a *remote procedure call* service. More specifically, it relies on operations of an abstract interface defined in the *RPC service class*. The RPC service actually used depends on the kind of protection boundary being crossed. If the remote object lives in a different protection domain on the same machine, an efficient *light-weight* remote procedure call service is used [1]. If the remote object resides on another machine, then a conventional remote procedure call protocol service is used. Figure 2 illustrates the basic components involved in

Services

A *service* provides a specific *resource*, for example, processors, memory, files, or communication channels. A service typically implements three types of objects: a *resource manager* object, a collection of *resource* objects, and an *address* object. Resource manager objects manage a class of similar resources. Only one instance of a particular resource manager object can exist within a protection domain. Resource objects are instances of the resource. Address objects specify a resource; when presented to the appropriate resource manager, they cause the creation of a resource object. Using the TCP network protocol as an example service, the resource manager object is the TCP protocol entity, an address object encapsulates a TCP address, and a resource object is an active TCP session (connection).

The code that implements a service—and thus provides implementations for the resource manager, resource, and address objects—is called a *service implementation module*. Service implementation modules are *dynamically loadable*, meaning that they can be added at runtime. In fact, the configuration of the operating system can change over time to accommodate changes in the environment and/or users' needs. Loading a new service implementation is triggered by events such as a user or another service requesting the new service, or the arrival of a message from the network whose interpretation requires a service (e.g, a particular protocol) that is not currently configured.

Abstract Interfaces and Service Classes

All interaction with an object is done by invoking one of the object's operations. An operation is uniquely identified by a name, a list of argument types, and a single result type. An *interface* is a group of related operations that an object supports. An *abstract interface* specifies a set of operations without providing an implementation for them. Thus, abstract interfaces are similar to *abstract types* in Emerald [2]: They specify a set of functions and their signatures. The set of all abstract interfaces and their operations that an object supports is called its *concrete type*.

A *service class*[1] specifies a set of abstract interfaces. First, it specifies a downcall interface for each of the resource manager, resource, and address objects. Second, it defines an optional upcall interface that a high-level object must support if it wants to use any of these objects. In other words, one can think of a service class as specifying how two layers of objects interact: the downcall operations that the low-level object exports and the upcall operations that the high-level object supports.

If an object supports the downcall interface specified in a particular service class then it is said to be *in* that class. Objects that are in the same class, and therefore support the same abstract downcall interface, are *interchangeable*. To illustrate this, assume that an object *O* requires a service that supports a downcall

[1] A service class is not to be confused with a C++ class; we will disambiguate when the meaning is not clear from context.

Nugget and Objects

Lipto consists of a fixed *nugget* and a variable collection of *objects*. The Lipto nugget provides two basic functions: (1) it implements the lowest-level process and memory management, and (2) it provides the machinery necessary to compose together a collection of objects. That is, the objects are the primary building blocks and the nugget provides the mortar. The low-level process and memory support offered by the nugget includes light-weight threads and multiple address spaces, including a privileged (kernel) address space. The Lipto nugget runs in the privileged address space. Objects can reside in any address space, including the privileged one. The basic Lipto architecture is illustrated in Figure 1.

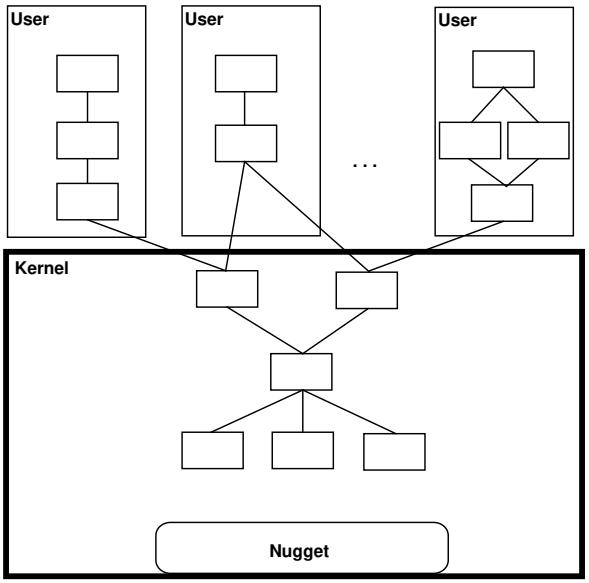

Figure 1: Lipto Architecture

A set of related operations supported by an object is called an *interface*. Motivated by our *x*-kernel experience with network protocol objects, Lipto objects are bi-directional, that is, they support two kinds of interfaces: *downcall interfaces* and *upcall interfaces*. Suppose object *A* uses services provided by object *B*. Then *A* is said to be a *client* of *B* and *B* is a *server* of *A*. *A* invokes operations in *B*'s downcall interface and *B* invokes operations in *A*'s upcall interface. Although one could argue that the up/down and client/server distinction is arbitrary, it is generally the case that the client object initiates the interaction by invoking an operation in the server's downcall interface, and only after such contact is made does the server object ever invoke an upcall operation on the client object. That is, one can think of the upcall interface as supporting a callback mechanism.

Lipto: A Dynamically Configurable Object-Oriented Kernel

Peter Druschel and Larry L. Peterson
University of Arizona

Norman C. Hutchinson
University of British Columbia

Lipto is a dynamically configurable, object-oriented operating system kernel [3]. It is based on the philosophy that an operating system is essentially a provider of communication services that are constructed out of a collection of 'building block' protocols. Consequently, our kernel focuses on providing an architecture and a set of tools that support the implementation and configuration of services.

Lipto grew out of our experiences with the *x*-kernel [6, 5]. The *x*-kernel is a configurable operating system kernel that provides explicit support for the implementation and composition of highly layered protocols. The *x*-kernel does not mandate a particular protocol suite; instead, it supports the implementation of a library of protocols. Different protocol suites can be composed from this library. This allows applications to uniformly access resources and services on both LANs and heterogeneous wide-area networks.

The most important way in which Lipto extends the *x*-kernel is that it allows configurable objects to live in any address space; in particular, object implementations are *location transparent*. In contrast, the *x*-kernel requires that all protocol objects reside in the kernel address space. Lipto also differs from the *x*-kernel in the following four ways:

- Lipto is implemented in C++ rather than C [4]. This change leads to a more structured and reliable implementation, with more natural and less error-prone interfaces to the kernel's services.
- Lipto provides architectural support for defining new service interfaces, whereas the *x*-kernel defines a single *uniform protocol interface*. This allows Lipto to accommodate a broader class of services than just network protocols.
- In Lipto, services are configured at run-time to provide a specific *service suite*. The *x*-kernel, in contrast, uses a statically configured protocol graph.
- Lipto allows object implementations to be dynamically loaded while the system is running. In contrast, the *x*-kernel is statically linked.

The following sections provide an overview of Lipto's architecture.