mechanism is at most twice as expensive as a statically bound invocation mechanism. Moreover, a "macro" experiment comparing Lipto to the *x*-kernel suggests that this increased cost on the invocation mechanism has little effect on the performance of the system as a whole.

5 Related work

Capability systems offer both modularity and protection at a fine grain. Because every invocation requires crossing a domain boundary, performance has been a problem, both in hardware [16, 11] and software based implementations [17].

Clouds is a distributed operating system that is, like Lipto, based on the object-thread model [5]. However, a Clouds object is persistent and resides in its own protection domain. Consequently, Clouds objects are "heavyweight" and they do not support fine-grained decomposition.

Choices is an object-oriented operating system [4]. It has a fine-grained, modular structure based on the encapsulation, subtyping, and inheritance mechanisms of its implementation language, C++. However, the entire system is contained in the kernel domain. The system's application interface is exported to user-level domains using proxy objects, but the internal interfaces are strictly local.

Several distributed programming systems, notably Emerald [12] and Hermes [15], provide uniform, location transparent invocation mechanisms that include efficient local invocations. However, both systems provide a single, high-level language environment, which is not appropriate for the implementation of a general-purpose operating system.

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 the USENIX Summer '86 Conference*, July 1986.

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

[3] A. Birrell and B. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, Feb. 1984.

[4] 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.

[5] P. Dasgupta, R. J. LeBlanc Jr., M. Ahamad, and U. Ramachandran. The Clouds distributed operating system. *IEEE Computer*. To appear.

[6] P. Druschel and L. L. Peterson. High-performance cross-domain data transfer. Technical Report TR 92-11, Department of Computer Science, University of Arizona, Tucson, Ariz., Mar. 1992.

[7] P. Druschel, L. L. Peterson, and N. C. Hutchinson. Service composition in Lipto. In *1991 Workshop on Object Orientation in Operating Systems*, pages 108–111, 1991.

[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 of the USENIX Summer '90 Conference*, pages 63–71, June 1990.

[9] 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.

[10] 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 12th ACM Symposium on Operating Systems Principles*, pages 91–101, Dec. 1989.

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

[12] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, Feb. 1988.

[13] 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.

[14] 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, Boston, Mass., May 1986.

[15] R. E. Strom. Hermes: An integrated language and system for distributed programming. In *1990 Workshop on Experimental Distributed Systems*, Oct. 1990.

[16] M. Wilkes and R. Needham. *The Cambridge CAP Computer and Its Operating System*. North Holland, Reading, MA, 1979.

[17] 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.

	MC68020	SPARC	MIPS R2000
System Call	26	15.2	9
No Arguments	57	22	15
One IOData Argument	51	19	13
Two `int` Arguments, `int` result	69	24	16

Table 2: UKRPC Performance (μsec)

	MC68020	SPARC	MIPS R2000
function call	4	0.3	0.5
local invoc.	7	0.9	0.8

Table 1: Local Call Performance (μsec)

The overhead is mainly due the indirection caused by the late binding, and the fact that the object pointer is passed as an argument. Note that our version of the GNU compiler does not use register windows on the SPARC. Using register windows, we suspect that the invocation cost on the SPARC would be within twice the cost of a function call, as is the case on the other processors. This cost—paid for location transparency and late binding—is small enough not to stand in the way of a fine-grained decomposition of the system.

Second, we measured the round-trip cost for an object in a user address space calling an object in the kernel address space (UKRPC). The results for various arguments and results are given in Table 2. The row labeled *system call* shows the cost of a kernel trap plus invocation of an empty C function in the kernel address space for each of the machines. It provides a reference value to determine the added overhead of a UKRPC invocation. Note that an invocation with one IOData argument is faster than a call with no arguments. This is because no argument message has to be created when one of the arguments is of type IOData; this IOData object is used as the argument. Also note that the cost of such an invocation is independent of the size of the IOData object. This is because no data is being copied.

We found that on the MC68020, a user-to-kernel object invocation costs approximately twice as much as a simple system call, but on a RISC processor, the additional penalty is on the order of only 33 to 50 percent.

Third, we measured the round-trip cost of a kernel-to-user object invocation with various arguments and results. The results are presented in Table 3. One observation is that upcalls are an order of magnitude more expensive than their counterpart downcalls. This is in part because of the lack of hardware support (i.e., a counterpart to a system trap) available on the Sun 3/75. Note that by combining UKRPC with KURPC, one gets a user-to-user RPC (LRPC) time of approximately 244 μsec in the null argument case, which is comparable to LRPC times reported elsewhere in the literature [2].

	MC68020
No Arguments	187
One IOData (128 bytes) Argument	240
Two `int` Arguments, `int` result	247

Table 3: KURPC Performance (μsec)

Finally, we measured the round-trip performance of Lipto's network RPC service suite. This is interesting for two reasons. First, this suite implements one of the three basic RPC cases identified in Section 3. Second, our network RPC is in fact implemented by a stack of modules that invoke each other using Lipto's location transparent invocation mechanism. We have explicitly compared the performance of this service suite with a corresponding suite implemented in the *x*-kernel [9], which does not provide a location transparent invocation mechanism. As illustrated in Table 4, the performance of this suite in the two systems is almost identical. Note that KURPC and UKRPC were used to cross the user/kernel boundary, which in this example was between an application object and the topmost module of the RPC protocol stack. The numbers reported are in milliseconds.

	Lipto RPC	*x*-Kernel
0 Bytes	2.6	2.6
128 Bytes	2.9	2.9
1k Bytes	5.2	5.1
2k Bytes	11.8	11.7
3k Bytes	15.0	14.8
4k Bytes	17.0	16.7
8k Bytes	29.2	29.2
15k Bytes	50.0	50.7

Table 4: NRPC Performance (msec)

In summary, Lipto's location transparent invocation

ence passed as an argument/result, into a reference that is meaningful in the recipient's domain. There are subtle issues concerning references and object identity, the details of which are omitted for the sake of brevity; for a discussion of these issues, see [12]. Lipto avoids overheads associated with reference translation by allowing more than one proxy object (in the same domain) to represent a given remote object. This approach makes it harder to determine object identity, but our experience suggests that operations whose performance suffer from this are few and infrequently invoked. Alternatively, Lipto's proxy compiler could be extended to generate proxies that employ different policies for each reference type, according to information in the service class definition.

Tasks (4) and (5) require special attention because they do not have to be performed in all cases. For example, data presentation is not an issue when the invocation is between objects that reside on machines of the same architecture; argument/result validation can be omitted if they are supplied by an object in a trusted domain; and, argument/result marshaling can be simplified in the case of an invocation between objects on the same machine. Our compiler-generated proxy objects take advantage of these optimizations and perform these functions only when required.

3.3.2 Data transfer and the IOData type: The data transfer component of object invocation is handled in the implementation of the abstract data type IOData. An IOData instance contains raw (untyped) data; its implementation avoids physical copying of data whenever possible. Abstractly, an IOData object is a string of untyped, immutable data. It supports a number of operations such as concatenate, fragment, truncate, prepend and append, to facilitate copy-free processing of the data by network protocols. Users of an IOData object cannot access its data directly; data must be explicitly embedded into and retrieved from the object. The implementation of the IOData type is integrated with virtual memory management to permit efficient data transfer across protection domains [6].

3.3.3 Control transfer: We now turn our attention to the various RPC protocols used to transfer control across protection boundaries. UKRPC implements the case of an object in a user-level address space calling an object in the kernel address space. As the reader is probably already well aware, UKRPC implements a simple system call disguised as an RPC service. KURPC is responsible for handling calls from the kernel address space to an object in a user-level address space. Thus, it implements what is commonly called a user-space *upcall*, again disguised as an RPC service. What has become known as *light-weight* RPC (LRPC) is in our implementation simply a combination of UKRPC and KURPC.

For cross-machine invocations, we use a decomposed version of the conventional Birrell-Nelson RPC protocol [3, 10]. This RPC service is implemented as three independent modules. Notice that this service, while it is used as a communication service for object invocation, could itself be configured with a domain boundary between any two component modules. In fact, the communication service used between two proxy objects can be arbitrarily complex. Consequently, the client and server proxy can transparently span heterogeneous systems and networks.

4 Performance

We ran a series of tests to evaluate the performance of our object invocation mechanisms. In each test, we measured the elapsed time for $1,000,000$ object invocations performed in a tight loop. From this number, we subtracted the elapsed time for $1,000,000$ iterations of an empty loop to compensate for the cost of the loop, and divided the result by $1,000,000$.

Most of the tests were run on a Sun 3/75, which uses a MC68020 microprocessor running at 16.67MHz. The processor has a 256 byte on-chip instruction cache. Our measurements suggested that in the test case for the invocation of an object within the same address space, all instructions of the test loop were held in the i-cache after the first iteration. In all other test cases, the instructions of the test loop exceeded the capacity of the i-cache. In order to make the numbers comparable, we included code into the test loop that ensures that every iteration starts with a cold i-cache.

Although ports of Lipto to RISC-based machines are not complete, we have measured individual components of our invocation mechanisms on both a Sun SPARCstation 1 (SPARC processor at 20 MHz) and an IRIS 4D/25 (MIPS R2000 at 20MHz). We then calculated the expected performance of the simplest invocation cases from these measurements. Note that the measurements on the RISC machines were made with warm caches, which seems reasonable given their larger instruction caches. The following reports on four separate experiments.

First, we measured the round-trip cost of a local (intra-domain) object invocation, which corresponds to a C++ virtual function call. Table 1 quantifies the overhead imposed by Lipto's location transparent object invocation in the local case by comparing a local invocation to an ordinary function call. The numbers presented in this table are in microseconds, and were measured using the GNU C++ compiler, version 1.37.1.

Object implementations are responsible for generating the operation table at compile time. To perform an invocation, the appropriate tuple is fetched from the table using the operation number as an index. Then, the offset is added to the value of the object reference and passed to the function, along with the operation's explicit arguments.

Our implementation of object references and invocations is identical to the way many C++ compilers implement object references and invocations. This is not accidental: Since C++ is our primary implementation language for system modules, and we are using the GNU C++ compiler, it was convenient to choose our convention to conform to this compiler's implementation. This allows modules implemented in C++ to invoke objects implemented in other modules as if they were C++ objects.

3.3 Non-local invocation

Whenever a reference is created to a non-local object, a *client proxy* object is instantiated in the local domain, and a reference to the proxy object is returned to the client. This proxy object supports the appropriate interface, and is responsible for forwarding invocations to a *server proxy object* in the callee object's protection domain. Upon receiving an invocation, this server proxy in turn invokes the callee object and passes any results of the invoked operation back to the client proxy, which eventually returns control to the calling object. The invocations from caller to client proxy, and from server proxy to callee are handled as local invocations, as described above. The caller is unaware of the fact that the callee is remote, just as the callee is unaware of the fact that it is being invoked by a remote object. The pair of proxy objects forward invocations between client and server object using a *remote procedure call* service. The specific RPC service used depends on the location of the server object with respect to the client object.

Figure 2 illustrates the basic components involved in a cross-domain call. Note that the "protection boundary" in this figure could correspond to a domain boundary as well as a machine boundary. The RPC service module may rely on a lower-level communication service to accomplish its task.

It is useful to isolate the orthogonal issues involved in location transparent object invocation. We have identified three such issues: (1) distribution hiding, (2) data transfer, and (3) control transfer. These three concerns are explicitly separated from one another in our implementation. First, distribution hiding is the responsibility of the proxy objects. Second, the data transfer mechanism is hidden inside the abstract data type IOData. The

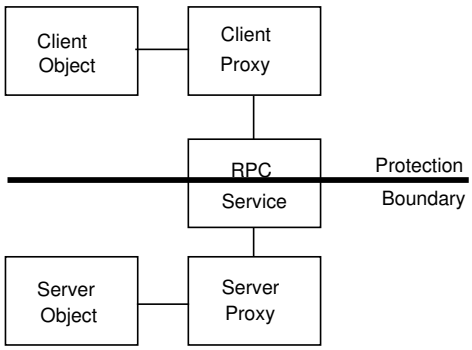

Figure 2: Proxy Objects and RPC

IOData data type encapsulates data as it is passed between the client proxy and server proxy. Finally, control transfer is handled by a set of highly specialized RPC services: a new protocol called *user-kernel RPC* (UKRPC) for the case of a user-level object invoking a kernel object; a new protocol called *kernel-user RPC* (KURPC) for the opposite case; LRPC and URPC protocols for invocations among user-level objects on the same machine; and Birrell-Nelson RPC for cross-machine invocations. The following subsections consider the three components of location transparent object invocation, in turn.

3.3.1 Distribution hiding and proxy objects: The client proxy and server proxy hide distribution and communication issues from the client and server objects. The proxy pair must perform several tasks: (1) binding and authentication; (2) failure handling; (3) object reference management; (4) argument/result marshaling; (5) argument/result validation; (6) performance enhancement (caching); and (7) replication management.

The proxy object implementations generated from an interface definition by the proxy-compiler handle the first five tasks. Tasks (6) and (7) are related to performance and fault-tolerance; handling of these issues is optional. The compiler-generated proxy object implementations can be manually augmented to employ performance improving techniques such as caching, and reliability-related techniques such as transparent server replication based on an underlying group communication service.

Binding and authentication are performed during the instantiation of a pair of proxy objects; each proxy is passed an *address object* that refers to its peer. The address object encapsulates an RPC address and authentication credentials; it is used by the proxies to establish a connection.

Object reference management (3) is concerned with the mechanisms and policies used to translate an object refer-

erations through this reference are forwarded; the object is not moved nor copied. The only exception to this rule applies to the system-defined object type *IOData*. The IOData type encapsulates untyped, immutable data. When an instance of type IOData is passed as an argument/result, it is moved. Instances of type IOData are typically used as containers for raw data that is passed through the layers of the I/O subsystem.

In Lipto, we have found the parameter passing modes `val` (for simple data types) and `ref` (for object types) to be sufficient in almost all cases. This is because complex arguments are usually encapsulated in objects, and bulk data is passed in IOData objects. However, to provide efficient passing modes for special situations, and to ease the transition of non-object-oriented interfaces, the additional passing modes `in`, `out` and `inout` are provided, which can be applied to arbitrary non-object data types. In the non-local case, these modes cause parameters to be copied in the usual way, but in the local case, they are implemented by passing a pointer. This eliminates parameter copying costs in the case of a local invocation, which could otherwise impose a performance penalty in the local case due to the lack of a by-reference parameter passing mode.

In order to ensure location transparency, modules that use `in`, `out`, or `inout` passing modes must obey three restrictions. In particular, modules *must not* (1) pass aliases as actuals to `in`, `out`, or `inout` formal parameters, (2) reference an `out` formal parameter before it is set, or (3) modify an `in` formal parameter. In languages where these constraints can be expressed, the proxy compiler generates appropriate code as part of the interface definition. For modules implemented in languages such as C, the programmer is responsible for verifying these constraints. Alternatively, one could devise language specific type checkers that warn about potentially location dependent code.

3.1.3 Failure handling: We next consider the issue of invocation failure handling. Our approach is to give the interface designer some flexibility. One choice is to reserve a special result value for each operation that indicates an invocation failure. The pitfall of this method is that the programmer has to include a test for failure after each invocation. Alternatively, a module interface specification can require each client object to support an *upcall* interface that includes an operation for failure notification. With this approach, clients can handle failures as exceptional conditions, and no explicit test is necessary after an invocation.

Finally, the interface designer is free to provide no failure notification at all in an interface. When an invocation fails, the system takes the default action of terminating the client's protection domain. This method effectively eliminates the potential for independent failure of client and server, and thus obviates the need for failure handling. Note that this may be a reasonable approach for many modules whose access scope is limited to the same physical machine. In general, location transparent invocation introduces complexity in failure handling, due to the potential for independent failure of modules. We believe that our design provides some flexibility, and minimizes the impact of this issue on the feasibility of fine-grained decomposition.

3.1.4 Performance: The final issue is that of non-local invocation performance. Our invocation mechanism relies on a configurable RPC service, which allows the dynamic substitution of the most appropriate RPC mechanism. As a consequence, RPC protocols can be used that take advantage of locality or special hardware support. In current hardware architectures, non-local invocations, even on the same machine, are an order of magnitude more costly than a procedure call. Thus, there is an inherent tradeoff between fine-grained protection and performance. Lipto's location transparent invocation system provides flexibility in dealing with this tradeoff. First, the dynamic configuration of RPC services allows the use of protocols that provide performance close to the architectural limits in each case. Second, proxy objects can employ caching mechanisms to reduce the performance hit of remote invocations. Third, by decoupling modularity and protection, the architecture allows the adjustment of the granularity of protection according to the needs of a particular installation and its users.

3.2 Object references and local invocation

Now consider the implementation of object references and local invocation. From the module/object implementor's perspective, all invocations are local; either the invocation is to a local server object, or to a proxy object. Thus, the implementation of object references and the conventions for local invocation entirely define the interface between module implementations and the module/object infrastructure.

An object reference is a pointer to a structure that represents the state of the referent object. This state structure is object implementation dependent. The architecture defines only the first member: A pointer to a table of $< functionpointer, offset >$ tuples, one for each operation that the object supports, indexed by the operation number. The *functionpointer* refers to a function that implements the operation. The *offset* is added to the value of the object reference, and the resulting value is passed as a hidden first argument to the function.

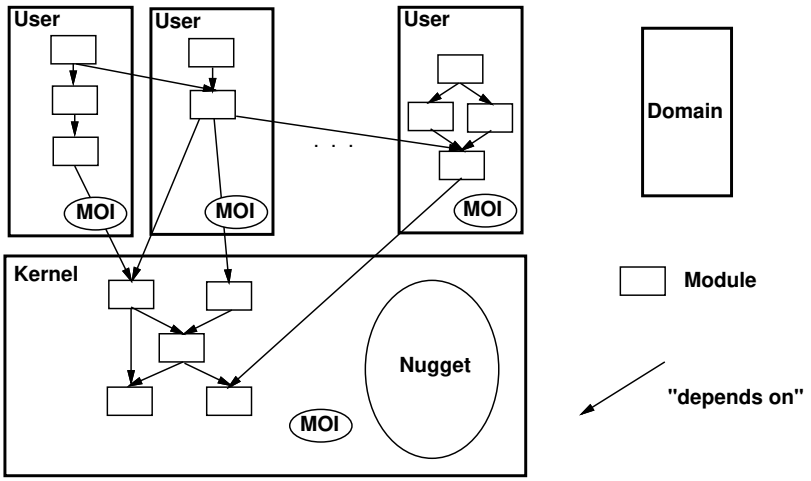

Figure 1: Lipto Architecture

efficient implementation of local invocations. This technique is "optimized" for local invocation, which is the most common case. It is the task of the proxy objects to handle the remote case, hiding the associated complexity from both caller and callee.

Several problems arise in the attempt to provide location transparency while retaining local procedure call semantics: remote procedure calls can fail; it is difficult to provide pass-by-reference parameter passing semantics; binding a caller and callee involves complex naming, protection, and authentication issues; and the performance of local, same-machine, and cross-machine invocations each differ by an order of magnitude. Most existing remote invocation systems don't attempt to hide the differences between local and remote calls, and require the caller to explicitly handle all the issues of a remote invocation, such as binding, authentication and failure handling. This approach conflicts with our goals since it exposes the full complexity of remote invocation to the caller, which would discourage a fine-grained decomposition.

The complexity of certain mechanisms used in the module/object infrastructure, and their associated costs, depend on the scale of the client-server distribution. Binding, authentication and failure detection are relatively simple between domains on a shared-memory machine or between nodes in a tightly coupled multiprocessor. They are more complex among machines connected by a LAN, and require elaborate protocols when the machines are connected by a WAN. Note that in a fine-grained modular system many modules need only be accessed in a limited scope, e.g., within the same machine. Only modules that represent entire subsystems must be accessible on a larger scale. Consequently, the most common cases can be handled using simple and efficient mechanisms. A module's access scope is determined when it is registered with the system directory, and is transparent to the module implementor. We proceed with a discussion of the issues in location transparency from the perspective of the module implementor.

3.1.1 Binding and authentication: Before invoking any operations on an object, a client has to obtain a reference for that object. This hides binding and authentication inside the mechanism for reference creation, thereby shielding the client from these issues. Further, the cost of binding, authentication and resource allocation is associated with reference creation, which allows a more efficient implementation of invocations because a "pseudo-connection" between client and server object is established. The implicit assumption is that a client that obtains a reference to an object will invoke several operations on that object before relinquishing the reference, so that the cost of reference creation is amortized over several calls. For cases where this is not a reasonable assumption, the proxy compiler can generate proxy object implementations for individual object types that delay connection establishment until the first invocation occurs.

3.1.2 Parameter passing: Lipto's invocation mechanism provides the parameter passing methods `val`, `ref`, `in`, `out` and `inout`. `Val` is the default mode for data types. Since it always causes parameters to be copied, it is only used for simple data types. `Ref` is the method used for arguments of an object type. A reference for the object is passed and invocations of the object's op-

ing modes that provide the efficiency of pass-by-reference in the case of a local (same domain) invocation, yet can be implemented in a straightforward and efficient way in the case of non-local invocations.

2 Architectural model

We have implemented and evaluated our proposed architecture in the context of an experimental operating system called Lipto. We focus here on those features of Lipto that pertain to the subject of this paper. A more comprehensive description of Lipto's design and motivation can be found elsewhere [7].

The components of Lipto's architecture are a fixed nugget, a module/object infrastructure, and a configurable collection of modules. The nugget consists of the lowest-level resource management, such as memory management, processor allocation and exception handling. The nugget is a truly minimal kernel; it includes functionality that *must* be executed with the machine in kernel mode, but not functionality that *may* be put into the kernel domain for performance.

The module/object infrastructure (MOI) provides location transparency and dynamic binding at the granularity of modules. All functionality not found in either the nugget or the module/object infrastructure is implemented in the form of configurable modules, which can be linked into any protection domain, including the privileged kernel domain of each machine. The basic Lipto architecture is illustrated in Figure 1.

A module provides the implementation for one or more types of *objects*. For example, a module that implements the top layer of a file system might provide two object types: a service object of type *file manager* and a resource object of type *file*. A module is the unit of configuration and distribution; it provides implementations of object types that represent a service or resource. Objects are passive run-time entities; they represent services and resources. An object's code is executed by independent threads of control upon invocation of one of the object's operations.

The architecture places three constraints on the implementation of objects: (1) An object must export a procedural interface consisting of a set of functions with well-defined signatures, (2) an object must not share data with any object implemented in a different module, and (3) the object invocation mechanism must conform to the architecture's specification. The module implementor is free to use any programming language and methodology, as long as these conditions are satisfied.

Encapsulation of objects with respect to objects implemented in other modules can be achieved in two ways. One is to use a programming language that enforces encapsulation through a safe type system. Unfortunately, such languages are currently not widely used in the implementation of operating system software. Without language support, encapsulation has to rely on convention. This has two consequences: first, each module should be configured into a separate protection domain during the testing phase to detect violations of encapsulation. Second, only modules that enjoy mutual trust can be configured into a common domain. Otherwise, an untrusted module could maliciously violate encapsulation to gain unauthorized access to information or resources.

The task of Lipto's module/object infrastructure is to provide location transparency. It consists of a service called the *system directory* that maps global object identifiers to object references, and a location transparent object invocation mechanism.

The system directory allows an authorized client object to obtain a *reference* to a server object using a global name for that object. This reference can be used to invoke the operations defined for the server object's interface. We omit a description of the mechanism for naming and locating objects, since it is not relevant to Lipto's architecture, which is the focus of this paper. The design and implementation of the invocation mechanism is described in detail in the next section.

Lipto includes a proxy compiler that takes as input an interface definition for a class of modules that export the same interface. It produces target language specific interface definition files and source code that implements proxy objects. Our proxy compiler initially supports C and C++ as target languages for the implementation of modules. We plan to provide support for other languages, for example Modula-3.

3 Object invocation mechanism

This section describes in some detail the design and implementation of Lipto's object invocation mechanism. We start with the overall design issues and proceed in the following subsections with a description of individual components.

3.1 Design issues

The main goal in the design of Lipto's invocation mechanism is: (1) to provide location transparency while closely resembling the semantics of a local procedure call, and (2) to retain the efficiency of a local procedure call in the intra-domain case. As stated earlier, this is essential for decomposing the system at a fine grain. The technique of proxy objects permits the representation of all object references as local pointers, and thus a very

allows *modular decomposition without concern for cross-domain communication costs*, and (2) the *partitioning of functions into protection domains becomes a matter of configuration rather than design*. These two facts have several important consequences, which are explored below.

First, while configuring a given system, the granularity of protection can be adjusted from one resembling a capability-system [17] (with each module in a separate domain), through the granularity found in micro-kernel based systems, up to no protection at all (with all modules in a single domain). The partitioning of modules into domains can be adjusted according to their stage in the software life-cycle, and/or the requirements in a particular installation of the system. For instance, a subsystem consisting of a set of modules can be configured with each of its modules in a separate domain for ease of fault detection and debugging during its testing phase, and later, in the post-release phase, combined into a single domain (or even the kernel domain) for performance. In other words, during the validation phase, the chosen granularity of protection is such that encapsulation is enforced. Once in the post-release phase, the granularity of protection is reduced to the point where only modules with different levels of trust are separated by domain boundaries.

Second, determining the set of functions to be included in the kernel domain—the subject of an ongoing debate in the micro-kernel based OS community—becomes a matter of configuration. The kernel module, which we call the *nugget*, can be reduced to include only functionality that *must* be provided in the privileged domain—management of hardware resources. All other services are implemented in separate modules which may or may not be configured into the kernel domain. This is important because new kinds of servers built on top of the kernel, new hardware platforms, and new OS technology will continue to shift the "right" set of functions to be included in the kernel domain.

Third, a system built in this manner can be extended vertically through layered services without imposing cross-domain invocation costs at each layer [8, 9]. Moreover, since interfaces are provided at module boundaries rather than domain boundaries, a layered service can be accessed at each level. That is, applications can access a layered system service at the level of abstraction most appropriate to their needs. This eliminates the need for unstructured, extensible interfaces like the UNIX `ioctl` call, which are used to allow the access of low-level services through an abstract, high-level interface provided at the domain boundary.

Note that most of the advantages of micro-kernel based systems (when compared to monolithic systems) are a result of the modularity, the dynamic binding of clients and servers, and the location transparent communication mechanism. Most of the overhead associated with micro-kernel based design, on the other hand, is a result of the more fine-grained protection. The proposed approach offers the advantages of fine-grained modularity, dynamic binding and location transparency without the cost of equally fine-grained protection. Decoupling modularity and protection turns the tradeoff between *modularity* and performance (which governs micro-kernel design) into a tradeoff between *protection* and performance.

1.2 How to separate modularity and protection

An architecture that decouples modularity and protection must provide location transparency and dynamic binding between *modules* instead of merely between protection domains. Moreover, the module invocation mechanisms must be designed both for efficient intra-domain invocation, and cross-domain communication.

The requirements for a suitable location transparent invocation mechanism are somewhat different from those found in micro-kernel based systems. The key difference is that many more interfaces exist (because the system is decomposed at a finer granularity), and as a consequence, the most common case is an invocation between modules in the same domain. A suitable invocation mechanism should provide semantics similar to a local procedure call and deliver performance similar to a local procedure call when the invocation is local. This is necessary in order to encourage and permit a fine-grained decomposition of the system.

One key issue is how *object references* are implemented. Most location transparent invocation systems, including those found in micro-kernel based systems, use global identifiers—such as ports or capabilities—to refer to all objects. With this approach, local invocations require the translation of a global identifier into a local object pointer. This design is pessimistic; it assumes that most object invocations are non-local. In the proposed architecture, most object references are to local objects. Consequently, an optimistic design that uses local pointers to refer to all objects is more appropriate, where the local pointer is translated in the case of a non-local invocation.

Another important issue is how to approximate the semantics of a local procedure call. The general approach is to shield the client and server from the complex issues of remote invocation. For example, binding and authentication are performed implicitly during object reference creation. Our design, which is discussed in detail in Section 3, employs the technique of *proxy objects* [14] to provide local object references and to shield client and server from the complexity of remote invocation. Finally, an appropriate invocation facility must offer a set of parameter pass-

Beyond Micro-Kernel Design: Decoupling Modularity and Protection in Lipto

Peter Druschel and Larry L. Peterson*

Department of Computer Science
University of Arizona
Tucson, AZ 85721

Norman C. Hutchinson

Department of Computer Science
University of British Columbia
Vancouver, B.C. V6T 1Z2

Abstract

This paper argues that a modular operating system architecture should provide support for modularity independent of protection domains. Given such support, modules and interfaces can be designed according to sound software engineering principles, without concern for cross-domain invocation costs. The partitioning of modules into domains (and across machines) becomes a matter of configuration, rather than design. Current micro-kernel based architectures do not sufficiently address this issue since their communication mechanisms are designed for the non-local, i.e., cross-domain case. We propose an architecture that provides location transparent binding and access of modules optimized for the local case, thereby decoupling the orthogonal concepts of modularity and protection.

1 Introduction

The benefits of a modular operating system design are well-known: The system is more easily configured, scaled, extended, ported, maintained, verified, and distributed across multiple processors. Growing support for modular OS design has recently popularized the idea of *micro-kernel* based systems [1, 13]—a small kernel that implements the most basic abstractions, and a small number of user-level *servers* that provide the functionality of a specific abstract machine. While micro-kernel based systems are an improvement over kernelized, monolithic operating systems like UNIX, they fall short of satisfying a number of demands placed on modern operating systems.

The problem with micro-kernel based systems is that they tightly couple modularity and protection—servers are implemented in separate protection domains, and consequently, the communication mechanisms are designed for the cross-domain case. A number of limitations arise as a result of this coupling. First, the modularity supported by these systems is very coarse-grained. Since each module is implemented in a separate domain, concern for cross-domain communication costs prevents a fine-grained decomposition of the system[1]. For the same reason, it is difficult to extend the system vertically—i.e., through stackable services—because each layer adds communication overhead that degrades performance. Second, the partitioning of functionality into servers—and in particular determining what functionality should be provided by the kernel—is static and part of an early design decision. Consequently, it is difficult to reconfigure functionality among servers and between kernel and servers. Such reconfigurations are desirable to satisfy the needs of applications, to match the characteristics of a variety of hardware architectures, and to integrate new technology as the system evolves.

The solution we propose is to provide architectural support for modules that is independent of protection domains. The key is a location transparent invocation mechanism that (1) has semantics similar to a local procedure call, (2) delivers performance close to that of an ordinary procedure call when the invocation is between modules in the same domain, and (3) allows the use of the most appropriate IPC mechanism in the cases of cross-domain and cross-machine invocations. Given such support, the decomposition of a system into modules can be guided by sound software engineering principles, with the modules distributed across protection domains and machines at configuration time based on criteria such as trust, security, fault-tolerance and performance. Put another way, protection and modularity are decoupled.

1.1 Why separate modularity and protection?

The fundamental reasons for providing support for modularity that is independent of protection are that (1) it

*This work supported in part by National Science Foundation Grant CCR-9102040, DARPA Contract DABT63-91-C-0030, and Sun Microsystems Inc.

[1]Note that even light-weight RPC is an order of magnitude more costly than local procedure calls