Lightweight Remote Procedure Call

Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

Abstract

Lightweight Remote Procedure Call (LRPC) is a communication facility designed and optimized for communication between protection domains on the same machine.

In contemporary small-kernel operating systems, existing RPC systems incur an unnecessarily high cost when used for the type of communication that predominates — between protection domains on the same machine. This cost leads system designers to coalesce weakly-related subsystems into the same protection domain, trading safety for performance. By reducing the overhead of same-machine communication, LRPC encourages both safety and performance.

LRPC combines the control transfer and communication model of capability systems with the programming semantics and large-grained protection model of RPC. LRPC achieves a factor of three performance improvement over more traditional approaches based on independent threads exchanging messages, reducing the cost of same-machine communication to nearly the lower bound imposed by conventional hardware.

LRPC has been integrated into the Taos operating system of the DEC SRC Firefly multiprocessor workstation.

1 Introduction

This paper describes Lightweight Remote Procedure Call (LRPC), a communication facility designed and optimized for communication between protection domains on the same machine.

LRPC combines the control transfer and communication model of capability systems with the programming semantics and large-grained protection model of RPC. For the common case of same-machine communication passing small, simple arguments, LRPC achieves a factor of three performance improvement over more traditional approaches.

The granularity of the protection mechanisms used by an operating system has a significant impact on the system's design and use. Some operating systems [Mealy et al. 66, Ritchie & Thompson 74] have large, monolithic kernels insulated from user programs by simple hardware boundaries. Within the operating system itself, though, there are no protection boundaries. The lack of strong firewalls, combined with the size and complexity typical of a monolithic system, make these systems difficult to modify, debug and validate. Further, the shallowness of the protection hierarchy (typically only two levels) makes the underlying hardware directly vulnerable to a large mass of complicated operating system software.

Capability systems supporting *fine-grained* protection were suggested as a solution to the problems of large-kernel operating systems [Dennis & Van Horn 66]. In a capability system, each fine-grained object exists in its own protection domain, but all live within a single name or address space. A process in one domain can act on an object in another only by making a *protected procedure call*, transferring control to the second domain. Parameter passing is simplified by the existence of a global name space containing all objects. Unfortunately, many found it difficult to efficiently implement and program systems that had such fine-grained protection.

In contrast to the fine-grained protection of capability systems, some distributed computing environments rely on relatively *large-grained* protection mechanisms: protection boundaries are defined by machine boundaries [Redell et al. 80]. Remote Procedure Call (RPC) [Birrell & Nelson 84] facilitates the placement of subsystems onto separate machines. Subsystems present themselves to one another in terms of interfaces implemented by servers. The absence of a global address space is ameliorated by automatic stub generators and sophisticated run-time libraries that can transfer arbitrarily complex arguments in messages. RPC is a system structuring and programming style that has become widely successful, enabling efficient and convenient communication across machine boundaries.

Small-kernel operating systems have borrowed the

This material is based on work supported by the National Science Foundation (Grants CCR-8619663, CCR-8700106 and CCR-8703049), the Naval Ocean Systems Center, U S WEST Advanced Technologies, the Washington Technology Center, and Digital Equipment Corporation (the Systems Research Center and the External Research Program). Anderson was supported by an IBM Graduate Fellowship Award, and Bershad was supported by an AT&T Ph.D. Scholarship.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1989 ACM 089791-338-3/89/0012/0102 $1.50

large-grained protection and programming models used in distributed computing environments and have demonstrated these to be appropriate for managing subsystems, even those not primarily intended for remote operation [Rashid 86]. In these small-kernel systems, separate components of the operating system can be placed in disjoint domains (or address spaces), with messages used for all inter-domain communication. The advantages of this approach include modular structure, easing system design, implementation, and maintenance; failure isolation, enhancing debuggability and validation; and transparent access to network services, aiding and encouraging distribution.

In addition to the large-grained protection model of distributed computing systems, small-kernel operating systems have adopted their control transfer and communication models — independent threads exchanging messages containing (potentially) large, structured values. In this paper, though, we show that most communication traffic in operating systems is (1) between domains on the same machine (*cross-domain*), rather than between domains located on separate machines (*cross-machine*), and (2) simple rather than complex. Cross-domain communication dominates because operating systems — even those supporting distribution — localize processing and resources to achieve acceptable performance at reasonable cost for the most common requests. Most communication is simple because complex data structures are concealed behind abstract system interfaces — communication tends to involve only handles to these structures and small value parameters (booleans, integers, etc.).

Although the conventional message-based approach can serve the communication needs of both local and remote subsystems, it violates a basic tenet of system design by failing to isolate the common case [Lampson 84]. A cross-domain procedure call *can* be considerably less complex than its cross-machine counterpart, yet conventional RPC systems have not fully exploited this fact. Instead, local communication is treated as an instance of remote communication, and simple operations are considered in the same class as complex ones.

Because the conventional approach has high overhead, today's small-kernel operating systems have suffered from a loss in performance or a deficiency in structure or both. Usually structure suffers most; logically separate entities are packaged together into a single domain, increasing its size and complexity. Such aggregation undermines the primary reasons for building a small-kernel operating system. The Lightweight Remote Procedure Call facility that we describe in this paper arises from these observations.

LRPC achieves a level of performance for cross-domain communication that is significantly better than conventional RPC systems while still retaining their qualities of safety and transparency. Four techniques contribute to the performance of LRPC:

- *Simple control transfer:* The client's thread executes the requested procedure in the server's domain.
- *Simple data transfer:* The parameter passing mechanism is similar to that used by procedure call. A shared argument stack, accessible to both client and server, can often eliminate redundant data copying.
- *Simple stubs:* LRPC uses a simple model of control and data transfer, facilitating the generation of highly optimized stubs.
- *Design for concurrency:* LRPC avoids shared data structure bottlenecks and benefits from the speedup potential of a multiprocessor.

We have demonstrated the viability of LRPC by implementing and integrating it into Taos, the operating system for the DEC SRC Firefly multiprocessor workstation [Thacker et al. 88]. The simplest cross-domain call using LRPC takes 157 microseconds on a single C-VAX processor. By contrast, SRC RPC, the Firefly's native communication system [Schroeder & Burrows 89], takes 464 microseconds to do the same call; though SRC RPC has been carefully streamlined and outperforms peer systems, it is a factor of three slower than LRPC. The Firefly virtual memory and trap handling machinery limit the performance of a safe cross-domain procedure call to roughly 109 microseconds; LRPC adds only 48 microseconds of overhead to this lower bound.

The remainder of this paper discusses LRPC in more detail. Section 2 describes the use and performance of RPC in existing systems, offering motivation for a more lightweight approach. Section 3 describes the design and implementation of LRPC. Section 4 discusses its performance, and section 5 addresses some of the concerns that arise when integrating LRPC into a serious operating system.

2 The Use and Performance of RPC Systems

In this section, using measurements from three contemporary operating systems, we show that only a small fraction of RPCs are *truly* remote, and that large or complex parameters are rarely passed during non-remote operations. We also show the disappointing performance of cross-domain RPC in several systems. These results demonstrate that simple, cross-domain calls represent the common case and can be well-served by optimization.

2.1 Frequency of Cross-Machine Activity

We examined three operating systems to determine the relative frequency of cross-machine activity.

- The V System

  In V [Cheriton 88], a highly decomposed system, only the basic message primitives (Send, Receive, etc.) are accessed directly through kernel traps. All other system functions are accessed by sending messages to the appropriate server. Concern for efficiency, though, has forced the implementation of many of these servers down into the kernel.

  In an instrumented version of the V system, Williamson found that 97% of calls crossed protection, but not machine, boundaries [Williamson 89]. Williamson's measurements include message traffic to kernel-resident servers.

- Taos

  Taos, the Firefly operating system, is divided into two major pieces. A medium-sized privileged kernel accessed through traps is responsible for thread scheduling, virtual memory, and device access. A second, multi-megabyte domain accessed through RPC implements the remaining pieces of the operating system (domain management, local and remote file systems, window management, network protocols, etc.). Taos does not cache remote files, but each Firefly node is equipped with a small disk for storing local files to reduce the frequency of network operations.

  We measured activity on a Firefly multiprocessor workstation connected to a network of other Fireflies and a remote file server. During one five-hour work period, we counted 344,888 local RPC calls, but only 18,366 network RPCs. Cross-machine RPCs thus accounted for only 5.3% of all communication activity.

- UNIX+NFS

  In UNIX, a large-kernel operating system, all local system functions are accessed through kernel traps. RPC is used only to access remote file servers. Although a UNIX system call is not implemented as a cross-domain RPC, in a more decomposed operating system most calls would result in at least one such RPC.

  On a diskless Sun 3 workstation running Sun UNIX+NFS [Sandberg et al. 85], during a period of four days we observed over 100 million operating system calls, but fewer than one million RPCs to file servers. Inexpensive system calls, encouraging frequent kernel interaction, and file caching, eliminating many calls to remote file servers, are together responsible for the relatively small number of cross-machine operations.

Table 1 summarizes our measurements of these three systems. Our conclusion is that most calls go to targets on the same node. While measurements of systems taken under different workloads will demonstrate different percentages, we believe that cross-domain activity, rather than cross-machine activity, will dominate. Because a cross-machine RPC is slower than even a slow cross-domain RPC, system builders have an incentive to avoid network communication. This incentive manifests itself in the many different caching schemes used in distributed computing systems.

Operating System	**Percentage of Operations That Cross Machine Boundaries**
V	3%
Taos	5.3%
Sun Unix+NFS	0.6%

Table 1: Frequency of Remote Activity

2.2 Parameter Size and Complexity

The second part of our RPC evaluation is an examination of the size and complexity of cross-domain procedure calls. Our analysis considers both the dynamic and static usage of SRC RPC as used by the Taos operating system and its clients. The size and maturity of the system make it a good candidate for study — our version includes 28 RPC services defining 366 procedures involving over 1000 parameters.

We counted 1,487,105 cross-domain procedure calls during one four-day period. Although 112 different procedures were called, 95% of the calls were to ten procedures, and 75% were to just three. None of the stubs for these three were required to marshal complex arguments — byte copying was sufficient to transfer the data between domains. [1]

In the same four days, we also measured the number of bytes transferred between domains during cross-domain calls. Figure 1, a histogram and cumulative distribution of this measure, shows that the most frequently occurring calls transfer fewer than 50 bytes, and a majority transfer fewer than 200.

Statically, we found that four out of five parameters were of fixed size known at compile time; sixty-five percent were four bytes or fewer. Two-thirds of all procedures passed only parameters of fixed size, and sixty percent transferred 32 or fewer bytes. No data types were recursively defined so as to require recursive marshaling (such as linked lists or binary trees). Recursive types *were* passed through RPC interfaces, but these were marshaled by system library procedures, rather than by machine-generated code.

These observations indicate that simple byte copying is usually sufficient for transferring data across system interfaces, and that the majority of interface procedures move only small amounts of data.

---

[1] SRC RPC maps domain-specific pointers into and out of network-wide unique representations, enabling pointers to be passed back and forth across an RPC interface. The mapping is done by a simple table-lookup, and was necessary for two of the top three procedures.

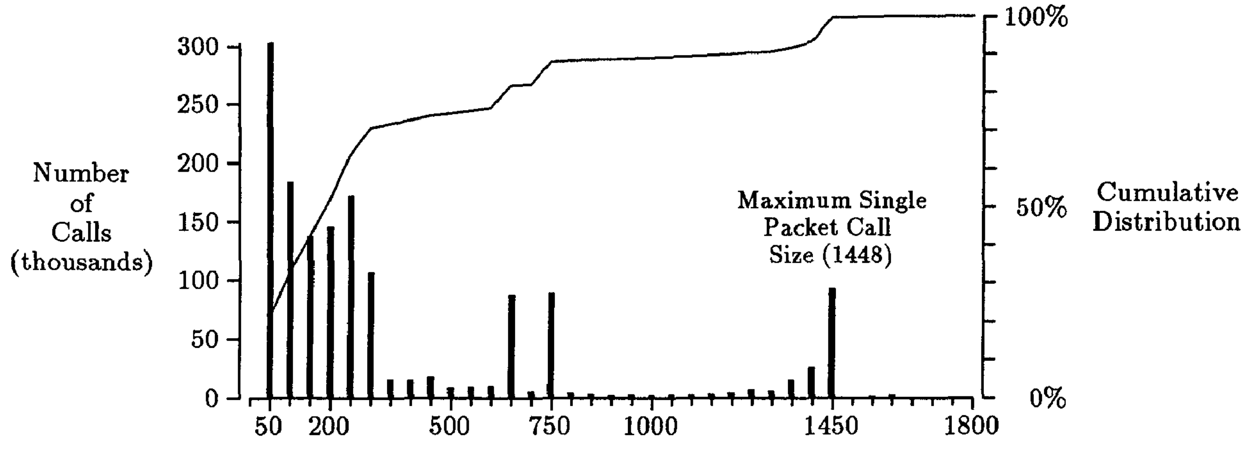

Figure 1: RPC Size Distribution

Others have noticed that most interprocess communication is simple, passing mainly small parameters [Cook 78, Cheriton 88, Karger 89], and some have suggested optimizations for this case. V, for example, uses a message protocol that has been optimized for fixed-sized messages of 32 bytes. Karger describes compiler-driven techniques for passing parameters in registers during cross-domain calls. These optimizations, although sometimes effective, only partially address the performance problems of cross-domain communication.

2.3 The Performance of Cross-Domain RPC

In existing RPC systems, cross-domain calls are implemented in terms of the facilities required by cross-machine ones. Even through extensive optimization, good cross-domain performance has been difficult to achieve. Consider the Null procedure call that takes no arguments, returns no values and does nothing:

```
PROCEDURE Null(); BEGIN RETURN END Null;
```

The theoretical minimum time to invoke Null() as a cross-domain operation involves one procedure call, followed by a kernel trap and change of the processor's virtual memory context on call, and then a trap and context change again on return. The difference between this theoretical minimum call time and the actual Null call time reflects the overhead of a particular RPC system. Table 2 shows this overhead for six systems. The data in Table 2 comes from measurements of our own and from published sources [Fitzgerald 86, Tzou & Anderson 88, van Renesse et al. 88].

The high overheads revealed by Table 2 can be attributed to several aspects of conventional RPC:

- *Stub overhead:* Stubs provide a simple procedure call abstraction, concealing from programs the interface to the underlying RPC system. The distinction between cross-domain and cross-machine calls is usually made transparent to the stubs by lower levels of the RPC system. This results in an interface and execution path that are general but infrequently needed. For example, it takes about 70 microseconds to execute the stubs for the Null procedure call in SRC RPC. Other systems have comparable times.
- *Message buffer overhead:* Messages need to be allocated and passed between the client and server domains. Cross-domain message transfer can involve an intermediate copy through the kernel, requiring four copy operations for any RPC (two on call, two on return).
- *Access Validation:* The kernel needs to validate the message sender on call and then again on return.
- *Message transfer:* The sender must enqueue the message, which must later be dequeued by the receiver. Flow-control of these queues is often necessary.
- *Scheduling:* Conventional RPC implementations bridge the gap between *abstract* and *concrete* threads. The programmer's view is one of a single, abstract thread crossing protection domains, while the underlying control transfer mechanism involves concrete threads fixed in their own domain signalling one another at a rendezvous. This indirection can be slow, as the scheduler must manipulate system data structures to block the client's concrete thread and then select one of the server's for execution.
- *Context switch:* There must be a virtual memory context switch from the client's domain to the server's on call, and then back again on return.

System	Processor	Null (Theoretical Minimum)	Null (Actual)	Overhead
Accent	PERQ	444	2300	1856
Taos	Firefly C-VAX	109	464	355
Mach	C-VAX	90	754	664
V	68020	170	730	560
Amoeba	68020	170	800	630
DASH	68020	170	1590	1420

Table 2: Cross-Domain Performance (times are in microseconds)

- *Dispatch:* A receiver thread in the server domain must interpret the message and dispatch a thread to execute the call. If the receiver is self-dispatching, it must ensure that another thread remains to collect messages that may arrive before the receiver finishes to prevent caller serialization.

RPC systems have optimized some of these steps in an effort to improve cross-domain performance. The DASH system [Tzou & Anderson 88] eliminates an intermediate kernel copy by allocating messages out of a region specially mapped into both kernel and user domains. Mach [Jones & Rashid 86] and Taos rely on *handoff scheduling* to bypass the general, slower scheduling path; instead, if the two concrete threads cooperating in a domain transfer are identifiable at the time of the transfer, a direct context switch can be made. In line with handoff scheduling, some systems pass a few, small arguments in registers, thereby eliminating buffer copying and management. [2]

SRC RPC represents perhaps the most ambitious attempt to optimize traditional RPC for swift cross-domain operation. Unlike techniques used in other systems which provide safe communication between mutually suspicious parties, SRC RPC trades safety for increased performance. To reduce copying, message buffers are globally shared across all domains. A single lock is mapped into all domains so that message buffers can be acquired and released without kernel involvement. Further, access validation is not performed on call and return, simplifying the critical transfer path.

SRC RPC runs much faster than other RPC systems implemented on comparable hardware. Nevertheless, SRC RPC still incurs a large overhead due to its use of heavyweight stubs and run-time support, dynamic buffer management, multi-level dispatch, and interaction with global scheduling state.

3 The Design and Implementation of LRPC

The lack of good performance for cross-domain calls has encouraged system designers to coalesce cooperating subsystems into the same domain. Applications use RPC to communicate with the operating system, ensuring protection and failure isolation for users and the collective system. The subsystems themselves, though, grouped into a single protection domain for performance reasons, are forced to rely exclusively on the thin barriers provided by the programming environment for protection from one another. LRPC solves, rather then circumvents, this performance problem in a way that does not sacrifice safety.

The execution model of LRPC is borrowed from protected procedure call. A call to a server procedure is made by way of a kernel trap. The kernel validates the caller, creates a call linkage, and dispatches the client's concrete thread directly to the server domain. The client provides the server with an argument stack as well as its own concrete thread of execution. When the called procedure completes, control and results return through the kernel back to the point of the client's call.

The programming semantics and large-grained protection model of LRPC are borrowed from RPC. Servers execute in a private protection domain, and each exports one or more interfaces, making a specific set of procedures available to other domains. A client *binds* to a server interface before making the first call. The server, by allowing the binding to occur, authorizes the client to access the procedures defined by the interface.

3.1 Binding

At a conceptual level, LRPC binding and RPC binding are similar. Servers export interfaces and clients bind to those interfaces before using them. At a lower-level, however, LRPC binding is quite different due to the high degree of interaction and cooperation that is required of the client, server and kernel.

A server module exports an interface through a clerk in the LRPC run-time library included in every domain. The clerk registers the interface with a name server and awaits import requests from clients. A client binds to a specific interface by making an import call via the kernel. The importer waits while the kernel notifies the server's waiting clerk.

---

[2] Optimizations based on passing arguments in registers exhibit a performance discontinuity once the parameters overflow the registers. The data in Figure 1 indicates that this can be a frequent problem.

The clerk enables the binding by replying to the kernel with a *procedure descriptor list* (PDL) that is maintained by the exporter of every interface. The PDL contains one *procedure descriptor* (PD) for each procedure in the interface. The PD includes an entry address in the server domain, the number of simultaneous calls initially permitted to the procedure by the client, and the size of the procedure's *argument stack* (A-stack) on which arguments and return values will be placed during a call. For each PD, the kernel pair-wise allocates in the client and server domains a number of A-stacks equal to the number of simultaneous calls allowed. These A-stacks are mapped read-write and shared by both domains.

Procedures in the same interface having A-stacks of similar size can share A-stacks, reducing the storage needs for interfaces with many procedures. The number of simultaneous calls initially permitted to procedures that are sharing A-stacks is limited by the total number of A-stacks being shared. This is only a soft limit, though, and Section 5.2 describes how it can be raised.

The kernel also allocates a *linkage record* for each A-stack that is used to record a caller's return address and is accessible only to the kernel. The kernel lays out A-stacks and linkage records in memory in a way such that the correct linkage record can be quickly located given any address in the corresponding A-stack.

After the binding has completed, the kernel returns to the client a Binding Object. The Binding Object is the client's key for accessing the server's interface and must be presented to the kernel at each call. The kernel can detect a forged Binding Object, so clients cannot bypass the binding phase. In addition to the Binding Object, the client receives an A-stack list for each procedure in the interface giving the size and location of the A-stacks that should be used for calls into that procedure.

3.2 Calling

Each procedure in an interface is represented by a stub in the client and server domains. A client makes an LRPC by calling into its stub procedure which is responsible for initiating the domain transfer. The stub manages the A-stacks allocated at bind time for that procedure as a LIFO queue. At call time, the stub takes an A-stack off the queue, pushes the procedure's arguments onto the A-stack, puts the address of the A-stack, the Binding Object and a procedure identifier into registers, and traps to the kernel. In the context of the client's thread, the kernel

- verifies the Binding and procedure identifier
- verifies the A-stack and locates the corresponding linkage
- ensures that no other thread is currently using that A-stack/linkage pair
- records the caller's return address and current stack pointer in the linkage
- pushes the linkage onto the top of a stack of linkages kept in the thread's control block[3]
- finds an execution stack (*E-stack*) in the server's domain
- updates the thread's user stack pointer to run off of the new E-stack
- reloads the processor's virtual memory registers with those of the server domain
- performs an upcall [Clark 85] into the server's stub at the address specified in the PD for the requested procedure.

Arguments are pushed onto the A-stack according to the calling conventions of Modula2+ [Rovner et al. 85]. Since the A-stack is mapped into the server's domain, the server procedure can directly access the parameters as though it had been called directly. It's important to note that this optimization relies on a calling convention that uses a separate argument pointer. In a language environment that required arguments to be passed on the E-stack, this optimization would not be possible.

The server procedure returns through its own stub, which initiates the return domain transfer by trapping to the kernel. Unlike the call, which required presentation and verification of the Binding Object, procedure identifier and A-stack, this information, contained at the top of the linkage stack referenced by the thread's control block, is implicit in the return. There is no need to verify the returning thread's right to transfer back to the calling domain since it was granted at call time. Further, since the A-stack contains the procedure's return values, and the client specified the A-stack on call, no explicit message needs to be passed back.

If any parameters are passed by reference, the client stub copies the referent onto the A-stack. The server stub creates a reference to the data and places the reference on its private E-stack before invoking the server procedure. The reference must be recreated to prevent the caller from passing in a bad address. The data, though, is not copied and remains on the A-stack.

Privately mapped E-stacks enable a thread to safely cross between domains. Conventional RPC systems provide this safety by implication, deriving separate stacks from separate threads. LRPC excises this level of indirection, dealing directly with less weighty stacks.

A low-latency domain transfer path requires that E-stack management incur little call-time overhead. One way to achieve this is to statically allocate E-stacks at bind time and to permanently associate each with an A-stack. Unfortunately, E-stacks can be large (tens of kilobytes) and must be managed conservatively; otherwise a server's address space could be exhausted by just a few clients.

---

[3] The stack is necessary so that a thread can be involved in more than one cross-domain procedure call at a time.

Rather than statically allocating E-stacks, LRPC delays the A-stack/E-stack association until it is needed; that is, until a call is made with an A-stack not having an associated E-stack. When this happens, the kernel checks if there is an E-stack already allocated in the server domain, but currently unassociated with any A-stack. If so, the kernel associates the E-stack with the A-stack. Otherwise, the kernel allocates an E-stack out of the server domain and associates it with the A-stack. When the call returns, the E-stack and A-stack remain associated with one another so that they might be used together soon for another call (A-stacks are LIFO managed by the client). Whenever the supply of E-stacks for a given server domain runs low, the kernel reclaims those associated with A-stacks that have not been recently used.

3.3 Stub Generation

Stubs bridge the gap between procedure call, the communication model used by the programmer, and domain transfer, the execution model of LRPC. A procedure is represented by a call stub in the client's domain and an entry stub in the server's. Every procedure declared in an LRPC interface defines the terminus of a three-layered communication protocol: end-to-end, described by the calling conventions of the programming language and architecture; stub-to-stub, implemented by the stubs themselves; and domain-to-domain, implemented by the kernel.

LRPC stubs blur the boundaries between the protocol layers to reduce the cost of crossing between them. Server entry stubs are invoked directly by the kernel on a transfer; no intermediate message examination and dispatch is required. The kernel primes E-stacks with the initial call frame expected by the server's procedure, enabling the server stub to branch to the first instruction of the procedure. As a result, a simple LRPC needs only one formal procedure call (into the client stub), and two returns (one out of the server procedure and one out of the client stub).

The LRPC stub generator produces run-time stubs in assembly language directly from Modula2+ definition files. The use of assembly language is possible because of the simplicity and stylized nature of LRPC stubs, which consist mainly of move and trap instructions. The LRPC stubs have shown a factor of four performance improvement over Modula2+ stubs created by the SRC RPC stub generator.

Since the stubs are automatically generated, the only maintenance concerns arising from this use of assembly language are related to the portability of the stub generator (the stubs themselves are not portable, but we don't consider this to be an issue). Porting the stub generator to work on a different machine architecture should be a straightforward task, although we have not yet had any reason to do so.

The stub generator emits Modula2+ code for more complicated, but less frequently traveled execution paths, such as those dealing with binding, exception handling, and call failure. Calls having complex or heavyweight parameters — linked lists or data that must be made known to the garbage collector — are handled with Modula2+ marshaling code. LRPC stubs become more like conventional RPC stubs as the overhead of dealing with the complicated data types increases. This shift occurs at compile-time, eliminating the need to make run-time decisions.

3.4 LRPC on a Multiprocessor

The existence of shared-memory multiprocessors has influenced the design of LRPC. Multiple processors can be used to achieve a higher call throughput and lower call latency than is possible on a single processor.

LRPC increases throughput by minimizing the use of shared data structures on the critical domain transfer path. Each A-stack queue is guarded by its own lock, and queuing operations take less than 2% of the total call time. No other locking occurs, so there is little interference when calls occur simultaneously.

Multiple processors are used to reduce LRPC latency by caching domain contexts on idle processors. As we show in Section 4, the context switch that occurs during an LRPC is responsible for a large part of the transfer time. This time is due partly to the code required to update the hardware's virtual memory registers, and partly to the extra memory fetches that occur as a result of invalidating the translation lookaside buffer (TLB).

LRPC reduces context-switch overhead by caching domains on idle processors. When a call is made, the kernel checks for a processor idling in the context of the server domain. If one is found, the kernel exchanges the processors of the calling and idling threads, placing the calling thread on a processor where the context of the server domain is already loaded; the called server procedure can then execute on that processor without requiring a context switch. The idling thread continues to idle, but on the client's original processor in the context of the client domain. On return from the server, a check is also made. If a processor is idling in the client domain (likely for calls that return quickly), then the processor exchange can be done again.

If no idle domain can be found on call or return, then a single-processor context switch is done. For each domain, the kernel keeps a counter indicating the number of times that a processor idling in the context of that domain was needed but not found. The kernel uses these counters to prod idle processors to spin in domains showing the most LRPC activity.

The high cost of frequent domain crossing can also be reduced by using a TLB that includes a process tag. For multiprocessors without such a tag, domain-caching can often achieve the same result for commonly called servers. Even with a tagged TLB, a single-processor domain switch still requires that hardware mapping registers be modified on the critical transfer path; domain

Operation	LRPC	Message Passing	Restricted Message Passing
call (mutable parameters)	A	ABCE	ADE
call (immutable parameters)	AE	ABCE	ADE
return	F	BCF	BF

Code	Copy Operation
A	copy from client stack to message (or A-stack)
B	copy from sender domain to kernel domain
C	copy from kernel domain to receiver domain
D	copy from sender/kernel space to receiver/kernel domain
E	copy from message (or A-stack) into server stack
F	copy from message (or A-stack) into client's results

Table 3: Copy Operations For LRPC Vs. Message-Based RPC

caching does not. Finally, domain caching preserves per-processor locality across calls — a performance consideration for systems having low tolerance for sudden shifts in locality.

Using idle processors to decrease operating system latency is not a new idea. Both Amoeba and Taos cache recently blocked threads on idle processors to reduce wakeup latency. LRPC generalizes this technique by caching domains, rather than threads. In this way, any thread that needs to run in the context of an idle domain can do so quickly, not just the thread that ran there most recently.

3.5 Argument Copying

Consider the path taken by a procedure's argument during a traditional cross-domain RPC. An argument, beginning with its placement on the stack of the client stub, is copied 4 times — from the stub's stack to the RPC message, from the message in the client's domain to one in the kernel's, from the message in the kernel's domain to one in the server's, and from the message to the server's stack. The same argument in an LRPC can be copied only once: from the stack of the client stub to the shared A-stack from which it can be used by the server procedure.

Pair-wise allocation of A-stacks enables LRPC to copy parameters and return values only as many times as are necessary to ensure correct and safe operation. Protection from third-party domains is guaranteed by the pair-wise allocation that provides a private channel between the client and server. It is still possible for a client or server to asynchronously change the values of arguments in an A-stack once control has transferred across domains. The copying done by message-based RPC prevents such changes, but often at a higher cost than necessary. LRPC, by considering each argument individually, avoids extra copy operations by taking advantage of argument passing conventions, by exploiting a value's correctness semantics, and by combining the copy into a check for the value's integrity.

In most procedure call conventions, the destination address for return values is specified by the caller. During the return from an LRPC, the client stub copies returned values from the A-stack into their final destination. No added safety comes from first copying these values out of the server's domain into the client's, either directly or by way of the kernel.

Parameter copying can also be avoided by recognizing situations in which the actual value of the parameter is unimportant to the server. This occurs when parameters are processed by the server without interpretation. For example, the *Write* procedure exported by a file server takes an array of bytes to be written to disk. The array itself is not interpreted by the server, which is made no more secure by an assurance that the bytes won't change during the call. Copying is unnecessary in this case. These types of arguments can be identified to the LRPC stub generator.

Finally, concern for type safety motivates explicit argument copying in the stubs, rather than wholesale message copying in the kernel. In a strongly-typed language, such as Modula2+, actual parameters must conform to the types of the declared formals; for example, the Modula2+ type CARDINAL is restricted to the set of positive integers — a negative value will result in a run-time error when the value is used. A client could crash a server by passing it an unwanted negative value. To protect itself, the server must check type-sensitive values for conformancy before using them. Folding this check into the copy operation can result in less work than if the value is first copied by the message system and then later checked by the stubs.

Table 3 shows how the use of A-stacks in LRPC can affect the number of copying operations. For calls where parameter immutability is important, and for those where it isn't, we compare the behavior of LRPC against the traditional message-passing approach, and

Test	Description	LRPC/MP	LRPC	Taos
Null	the Null cross-domain call	125	157	464
Add	a procedure taking two 4-byte arguments and returning one 4-byte argument	130	164	480
BigIn	a procedure taking one 200-byte argument	173	192	539
BigInOut	a procedure taking and then returning one 200-byte argument	219	227	636

Table 4: LRPC Performance of Four Tests (in microseconds)

against a more restricted form of message-passing used in the DASH system. In the restricted form, all message buffers on the system are allocated from a specially mapped region that enables the kernel to copy messages directly from the sender's domain into the receiver's, avoiding an intermediate kernel copy.

In Table 3, we assume that the server places the results directly into the reply message. If this isn't the case (i.e., messages are managed as a scarce resource), then one more copy from the server's results into the reply message is needed. Even when the immutability of parameters is important, LRPC performs fewer copies (3) than either message passing (7) or restricted message passing (5).

For passing large values, copying concerns become less important, since by-value semantics can be achieved through virtual memory operations. But, for the more common case of small- to medium-sized values, eliminating copy operations is crucial to good performance when call latency is on the order of only 100 instructions.

LRPC's A-stack/E-stack design offers both safety and performance. While our implementation demonstrates the performance of this design, the Firefly operating system does not yet support pair-wise shared memory. Our current implementation places A-stacks in globally shared virtual memory. Since mapping is done at bind time, an implementation using pair-wise shared memory would have identical performance, but greater safety.

4 The Performance of LRPC

To evaluate the performance of LRPC, we used the four tests shown in Table 4. These tests were run on the C-VAX Firefly using LRPC and Taos RPC. The Null call provides a baseline against which we can measure the added overhead of LRPC. The procedures Add, BigIn, and BigInOut represent calls having "typical" parameter sizes.

Table 4 shows the results of these tests when performed on a single node. The measurements were made by performing 100,000 cross-domain calls in a tight loop, computing the elapsed time, and then dividing by 100,000. The table shows two times for LRPC. The first, listed as "LRPC/MP," uses the idle processor optimization described in Section 3.4. The second, shown as "LRPC," executes the domain switch on a single processor; it is roughly 3 times faster than SRC RPC, which also uses only one processor.

Table 5 shows a detailed cost breakdown for the serial (1-processor) Null LRPC on a C-VAX. This table was produced from a combination of timing measurements and hand calculations of TLB misses. The code to execute a Null LRPC consists of 120 instructions that require 157 microseconds to execute. The column labeled "Minimum" in Table 5 is a timing breakdown for the theoretically minimum cross-domain call (one procedure call, two traps and two context switches). The column labeled "LRPC Overhead" shows the additional time required to execute the call and return operations described in Section 3.2 and is the cost of our implementation. For the Null call, approximately 18 microseconds are spent in the client stub and 3 in the server's. The remaining 27 microseconds of overhead are spent in the kernel, and go towards binding validation and linkage management. Most of this takes place during the call, as the return path is simpler.

Operation	Minimum	LRPC Overhead
Modula2+ Procedure Call	7	
Two Kernel Traps	36	
Two Context Switches	66	
Stubs		21
Kernel Transfer		27
TOTAL	109	48

Table 5: Breakdown of Time (in microseconds) for Single Processor Null LRPC

Approximately 25% of the time used by the Null LRPC is due to TLB misses that occur during virtual address translation. A context switch on a C-VAX requires the invalidation of the TLB, and each subsequent TLB miss increases the cost of a memory reference by about .9 microseconds. Anticipating this, the data structures and control sequences of LRPC were designed to minimize TLB misses. Even so, we estimate that 43 TLB misses occur during the Null call.

Section 3.4 stated that LRPC avoids locking shared data during call and return in order to remove contention on shared-memory multiprocessors. This is demonstrated by Figure 2, which shows call throughput as a function of the number of processors simultaneously making calls. Domain caching was disabled for

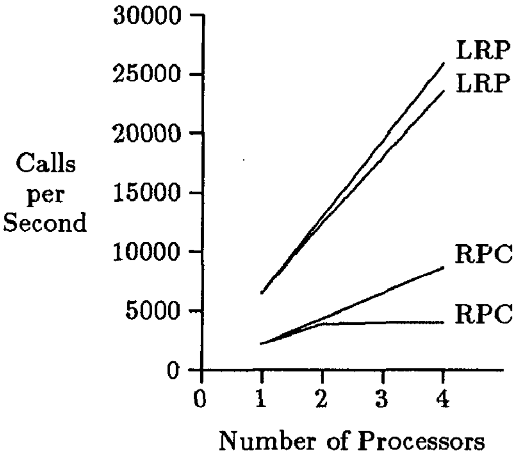

Figure 2: Call Throughput On a Multiprocessor

this experiment — each call required a context switch. A single processor can make about 6300 LRPCs per second, but four processors can make over 23000 calls per second — a speedup of 3.7 and close to the maximum that the Firefly is capable of delivering. These measurements were made on a Firefly having four C-VAX processors and one MicroVaxII I/O processor. Measurements on a five processor MicroVaxII Firefly showed a speedup of 4.3 with 5 processors.

In contrast, the throughput of SRC RPC levels off with two processors at about 4000 calls per second. This limit is due to a global lock that is held during a large part of the RPC transfer path. For a machine like the Firefly, a small scale shared-memory multiprocessor, a limiting factor of two is annoying, but not serious. On shared-memory machines with just a few dozen processors, though, contention on the critical control transfer path would have a greater performance impact.

5 The Uncommon Cases

In addition to working well in the common case, LRPC must work acceptably in the less common ones. This section describes several of these less common cases and explains how they are dealt with by LRPC. This section does not enumerate all possible uncommon cases that must be considered. Instead, by describing just a few, we hope to emphasize that the common-case approach taken by LRPC is flexible enough to accommodate the uncommon cases gracefully.

5.1 Transparency and Cross-Machine Calls

Deciding whether a call is cross-domain or cross-machine is made at the earliest possible moment — the first instruction of the stub. If the call is to a truly remote server (indicated by a bit in the Binding Object), then a branch is taken to a more conventional RPC stub. The extra level of indirection is negligible compared to the overheads that are part of even the most efficient network RPC implementation.

5.2 A-stacks — Size and Number

Procedure Descriptor Lists are defined during the compilation of an interface. The stub generator reads each interface and determines the number and size of the A-stacks for each procedure. The number defaults to five, but can be overridden by the interface writer. When the size of each of a procedure's arguments and return values are known at compile time, the A-stack size can be determined exactly. In the presence of variable sized arguments, though, the stub generator uses a default size equal to the Ethernet packet size (this default also can be overridden). Experience has shown, and Figure 1 confirms, that RPC programmers strive to keep the sizes of call and return parameters under this limit. Most existing RPC protocols are built on simple packet exchange protocols, and multi-packet calls have performance problems. In cases where the arguments are too large to fit into the A-stack, the stubs transfer data in a large out-of-band memory segment. Handling unexpectedly large parameters is complicated and relatively expensive, but infrequent.

A-stacks in a single interface are allocated contiguously at bind time to allow for quick validation during a call (a simple range check guarantees their integrity). If the number of pre-allocated A-stacks proves too few, the client can either wait for one to become available (when an earlier call finishes), or allocate more. Waiting is simple, but may not always be appropriate. When further allocation is necessary, it is unlikely that space contiguous to the original A-stacks will be found, but other space can be used. A-stacks in this space, not in the primary contiguous region, will take slightly more time to validate during a call.

5.3 Domain Termination

A domain can terminate at any time, for reasons such as an unhandled exception or a user action (CTRL-C). When a domain terminates, all resources in its possession (virtual address space, open file descriptors, threads, etc.) are reclaimed by the operating system. If the terminating domain is a server handling an LRPC request, the call, completed or not, must return to the client domain. If the terminating domain is a client with a currently outstanding LRPC request to another domain, the outstanding call, when finished, must not be allowed to return to its originating domain.

When a domain is terminated, each Binding Object associated with that domain (either as client or server) is revoked. This prevents any more out-calls from the domain, and prevents other domains from making any more in-calls. All threads executing within the domain are then stopped, and a kernel collector scans all of the domain's threads looking for any that had been running on behalf of an LRPC call; these threads are restarted in the client with a call-failed exception. Finally, the collector scans all Binding Objects held by the terminating domain and invalidates any active linkage records. When a thread returns from an LRPC call, it follows the stack of linkage records referenced by the thread control block, returning to the domain specified in the first valid linkage record. If any invalid linkage records are found on the way, a call-failed exception is raised in the caller. If the stack contains no valid linkage records, the thread is destroyed.

A terminating domain's outstanding threads are not forced to terminate synchronously with the domain. Doing so would require every server procedure to protect the integrity of its critical data structures from external forces, since a mutating thread could be terminated at any time. More generally, LRPC has no way to force a thread to return from an outstanding call. Taos does have an *alert* mechanism which allows one thread to signal another, but the notified thread may choose to ignore the alert. It is therefore possible for one domain to "capture" another's thread and hold it indefinitely. To address this problem, LRPC enables client domains to create a new thread whose initial state is that of the original captured thread as if it had just returned from the server procedure with a call-aborted exception. The captured thread continues executing in the server domain but is destroyed in the kernel when released.

Traditional RPC does not have these problems because the abstract thread seen by the programmer is provided by two concrete threads, one in each of the client and server domains. Because premature domain and call termination are infrequent, LRPC has adopted a "special case" approach for dealing with them.

6 Summary

This paper has described the motivation, design, implementation, and performance of LRPC, a communication facility that combines elements of capability and RPC systems. Our implementation on the Firefly achieves performance that is close to the minimum round-trip cost of transferring control between domains on conventional hardware.

LRPC adopts a common-case approach to communication, exploiting, whenever possible, simple control transfer, simple data transfer, simple stubs, and multiprocessors. In so doing, LRPC performs well for the majority of cross-domain procedure calls by avoiding needless scheduling, excessive run-time indirection, unnecessary access validation, redundant copying, and lock contention. LRPC, nonetheless, is safe and transparent, and represents a viable communication alternative for small-kernel operating systems.

7 Acknowledgements

We would like to thank Guy Almes, David Anderson, Andrew Birrell, Mike Burrows, Dave Cutler, Roy Levin, Mark Lucovsky, Tim Mann, Brian Marsh, Rick Rashid, Dave Redell, Jan Sanislo, Mike Schroeder, Shin-Yuan Tzou, and Steve Wood for discussing with us the issues raised in this paper. We would also like to thank DEC SRC for building and supplying us with the Firefly. It has been a challenge to improve on the excellent performance of SRC RPC, but one made easier by the Firefly's overall structure. One measure of a system's design is how easily a significant piece of it can be changed. We doubt that we could have implemented LRPC as part of any other system as painlessly as we did on the Firefly.

References

[Birrell & Nelson 84] Birrell, A. D. and Nelson, B. J. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[Cheriton 88] Cheriton, D. R. The V Distributed System. *Communications of the ACM*, 31(3):314–333, March 1988.

[Clark 85] Clark, D. D. The Structuring of Systems Using Upcalls. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 171–180, December 1985.

[Cook 78] Cook, D. *The Evaluation of a Protection System.* PhD dissertation, Cambridge University, Computer Laboratory, April 1978.

[Dennis & Van Horn 66] Dennis, J. B. and Van Horn, E. C. Programming Semantics for Multiprogrammed Computations. *Communications of the ACM*, 9(3):143–155, March 1966.

[Fitzgerald 86] Fitzgerald, R. P. *A Performance Evaluation of the Integration of Virtual Memory Management and Inter-Process Communication in Accent.* PhD dissertation, Carnegie-Mellon University, October 1986.

[Jones & Rashid 86] Jones, M. B. and Rashid, R. F. Mach and Matchmaker: Kernel and Language Support for Object-Oriented Distributed Systems. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 67–77, October 1986.

[Karger 89] Karger, P. A. Using Registers to Optimize Cross-Domain Call Performance. In *Proceedings of the Third Conference on Architectural Support for Programming Languages and Operating Systems*, April 1989.

[Lampson 84] Lampson, B. W. Hints for Computer System Design. *IEEE Software*, 1(1):11–28, January 1984.

[Mealy et al. 66] Mealy, G., Witt, B., and Clark, W. The Functional Structure of OS/360. *IBM Systems Journal*, 5(1):3–51, 1966.

[Rashid 86] Rashid, R. F. From Rig to Accent to Mach: The Evolution of a Network Operating System. In *Proceeding of ACM/IEEE Computer Society Fall Joint Computer Conference*, November 1986.

[Redell et al. 80] Redell, D. D., Dalal, Y. K., Horsley, T. R., Lauer, H. C., Lynch, W. C., McJones, P. R., Murray, H. G., and Purcell, S. C. Pilot: An Operating System for a Personal Computer. *Communications of the ACM*, pages 81–92, February 1980.

[Ritchie & Thompson 74] Ritchie, D. and Thompson, K. The Unix Time-Sharing System. *Communications of the ACM*, 17(7):365–375, July 1974.

[Rovner et al. 85] Rovner, P., Levin, R., and Wick, J. On Extending Modula-2 For Building Large, Integrated Systems. Technical Report # 3, Digital Equipment Corporation Systems Research Center, Palo Alto, California, January 1985.

[Sandberg et al. 85] Sandberg, R., Goldberg, D., Steve Kleiman, D. W., and Lyon, B. Design and Implementation of the SUN Network Filesystem. In *Proceedings of the 1985 USENIX Summer Conference*, pages 119–130, 1985.

[Schroeder & Burrows 89] Schroeder, M. D. and Burrows, M. Performance of Firefly RPC. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, December 1989. To appear in ACM Transactions on Computer Systems, February 1990.

[Thacker et al. 88] Thacker, C. P., Stewart, L. C., and Satterthwaite, Jr., E. H. Firefly: A Multiprocessor Workstation. *IEEE Transactions on Computers*, 37(8):909–920, August 1988.

[Tzou & Anderson 88] Tzou, S.-Y. and Anderson, D. P. A Performance Evaluation of the DASH Message-Passing System. Technical Report UCB/CSD 88/452, Computer Science Division, University of California, Berkeley, October 1988.

[van Renesse et al. 88] van Renesse, R., van Staveren, H., and Tanenbaum, A. S. Performance of the World's Fastest Distributed Operating System. *Operating Systems Review*, 22(4):25–34, October 1988.

[Williamson 89] Williamson, C., January 1989. Personal communication.