Implementing Lightweight Remote Procedure Calls in the Mach 3 Operating System

Virgil Bourassa and John Zahorjan

virgil@cs.washington.edu, zahorjan@cs.washington.edu

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

Abstract

The Mach 3 operating system makes extensive use of remote procedure calls (RPCs) to provide services to client applications. Although the existing Mach 3 RPC facility is highly optimized, the incorporation of a Lightweight Remote Procedure Call (LRPC) facility further reduces this critical cost. This paper describes the implementation of LRPC in the Mach 3 operating system, reviewing the original LRPC concept and implementation in the TAOS operating system, the issues involved with the Mach 3 microkernel operating system, and the resulting design of $LRPC_{Mach3}$. Performance measurements indicate that the resulting implementation provides a 31% reduction in latency for a minimal RPC, with even more significant benefits for more complicated RPCs.

1. Introduction

This paper describes the implementation of Lightweight Remote Procedure Calls [3] in the Mach[1] 3 operating system [1][15][16]. This implementation shows that a variation of the original LRPC facility tailored to the Mach 3 operating system delivers significant performance improvements over the existing Mach RPC facility—despite the existing RPC having been optimized for local communications[6][7].

Remote Procedure Calls (RPCs) [21][5] extend the procedure–call mechanism of programming languages to allow execution of procedures by other processes, which may or may not be located on other computers. Remote procedure calls have two layers: the *semantic* layer, corresponding to the interface presented to the programmer; and the *transport* layer, corresponding to the underlying control and data transfer mechanism. Historically, RPCs began by using an underlying message–passing transport layer appropriate for network communications. In time, however, it became clear that the most common use of RPCs was for servers residing on the same machine [2]. The Mach 3 operating system takes advantage of this by providing a relatively fast *local* RPC mechanism that ignores network communication needs. But recent research into *lightweight* remote procedure calls indicates that existing local RPC facilities can gain further performance improvements by using a procedure–call based, rather than a message–passing based, transport mechanism.

The Mach 3 operating system has a strong need for low–cost client–server communications. Organized as a *microkernel*, Mach 3 provides a small set of fundamental kernel services meant to support all other services at the user level. The user–level services are made available to applications through an RPC facility supported directly by the

[1] Mach is a trademark of Carnegie Mellon University.

kernel (see figure 1). While this gives flexibility and modularity to the operating system organization, it imposes more

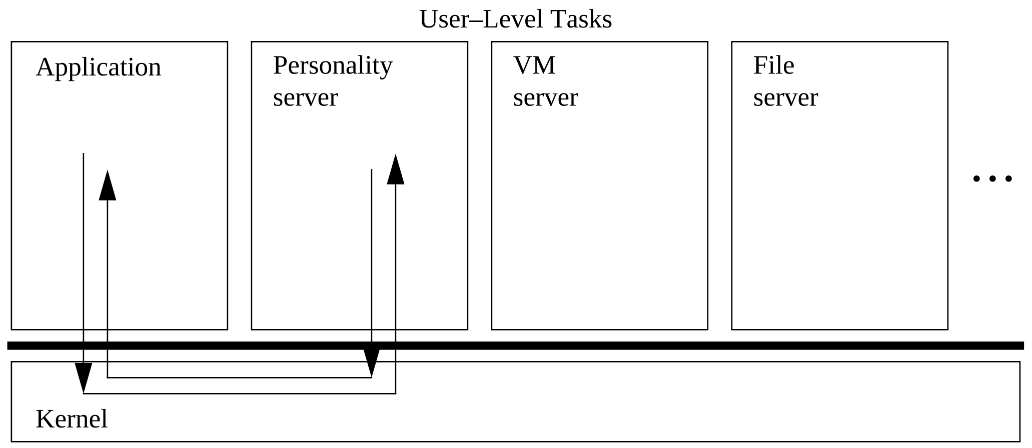


Figure 1. Mach 3's microkernel architecture puts most of the operating system services at the user level, requiring remote procedure call access.

inter–process communication than that of the more traditional monolithic operating system, which provides all services from within the kernel.

$\text{LRPC}_{\text{Mach3}}$ provides an LRPC facility in Mach 3 as a transparent RPC alternative which works within the existing Mach framework and remains consistent with the existing standard RPC interface, MiG (Mach interface Generator). The structure of Mach 3 leads to a different design for its LRPC facility than for the original LRPC facility implemented by Bershad in the TAOS operating system [3].

2. LRPC Concepts & Implementation

Bershad showed in [2] that well over 95% of RPC calls in general settings are to servers on the same computer. This implies that the RPC functionality of providing transparent server access across a network of computers is unused in the common case. In addition, the high cost of network access masks further message–based overhead costs that become bottlenecks when the client and server are known to be on the same machine. LRPC shows how local RPC (see figure 2) can be implemented with minimal overhead, primarily by matching the transport layer to the semantic layer.

The sources of overhead for traditional RPC transport mechanisms include: stub overhead, message buffer overhead, access validation, message transfer, scheduling, processor reallocation, and dispatching. Stubs present the procedure call interface to the underlying RPC message–based transport mechanism, packing and unpacking messages appropriate to the remote procedure being invoked. Message buffers must be allocated and data copied to and from them. The message sender must be validated on both call and return. The message transfer requires enqueuing, dequeuing, and flow control.

Further, the client thread must be blocked until the reply is received, then a server thread must be scheduled for execution. The processor must be reallocated from the client's to the server's address space, then back again. And, finally, the server's receiving thread must dispatch a client's request to itself or to another server thread.

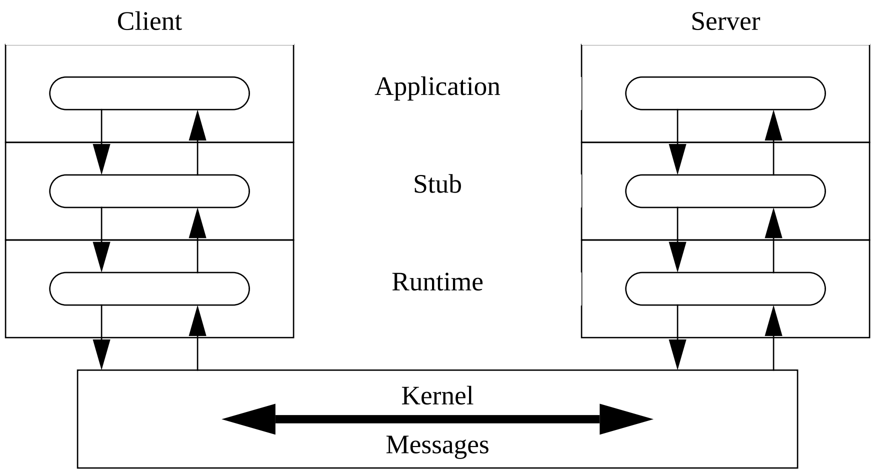


Figure 2. RPC in the local case (from [2], p23).

The costs just described suggest four techniques for the reduction of overhead: simple control transfer, simple data transfer, simple stubs, and design for concurrency. The design goal of these techniques is to emulate, as much as possible, a procedure–call model rather than a message–passing model.

$\text{LRPC}_{\text{TAOS}}$[1] for the Modula2+ [17] programming language in the TAOS operating system on the DEC Firefly [19] achieves a factor of three improvement over the best RPC mechanism available for that environment, namely SRC RPC [18]. In this implementation, the control transfer is made simple by using the client's thread to execute the requested service in the server's address space. The data transfer is made simple by using a shared argument stack between the client and server, providing a parameter passing mechanism similar to that of a procedure call. The simpler control and data transfer allow for the generation of optimized stubs tightly integrated with the programming language. Finally, shared data structures are avoided to minimize interactions among requests.

3. Mach 3

3.1. Kernel Abstractions

The target environment of $\text{LRPC}_{\text{Mach3}}$ is the Mach 3 operating system. The kernel of this operating system is designed to support a minimal set of abstractions. The kernel's role extends to facilitating execution, communication, address spaces, and general resource management. All other operating system services are provided by user–level servers on top of these kernel services. Some important abstractions provided by the Mach kernel are depicted in figure 3. These include tasks, threads, virtual memory (VM), and ports. Tasks serve as containers of resources. These resources include threads of control, a virtual address space, ports, and user–defined resources.

Ports are an amalgam of resource–related concepts. A port acts simultaneously as a reference to a resource, access rights to the resource, and a communications channel for the resource. Exactly one task (possibly the kernel task) at a time has the right to receive messages from a given port. Ports may be distributed freely among tasks, either to give

[1]. "$\text{LRPC}_{\text{TAOS}}$" is used in this paper to refer to the original implementation of LRPC on the TAOS operating system. "LRPC" by itself refers to the conceptual contribution.

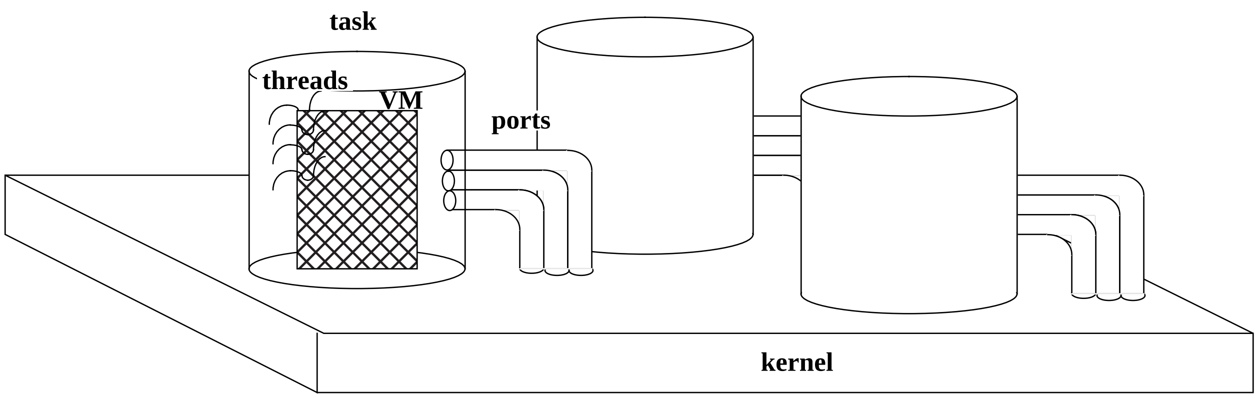
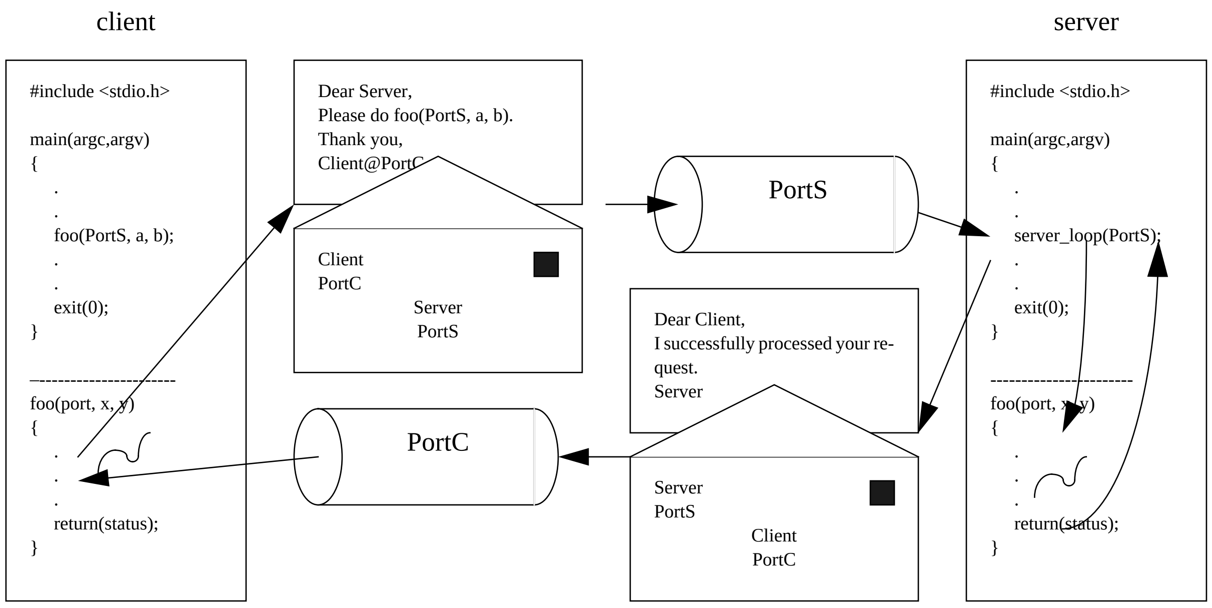


Figure 3. The Mach 3 operating system provides a kernel which directly supports user-level tasks with a few fundamental abstractions.

rights to send to the port, or to transfer the right to receive from a port. The actual reference to the port (called the *port name*) is local to each task it resides in, but resolves to a common underlying resource entity.

3.2. RPC

MiG is a standard Mach facility that aids in the creation of server tasks. MiG produces C language client and server stub code, as well as a server request–processing loop, based on a definition of the server's exported procedures. The stubs take care of building, sending, receiving, and understanding the necessary messages for the kernel's port messaging facility, termed IPC (see figure 4). Clients bind to a server by presenting the server's access name to the



```
#include <stdio.h>

main(argc,argv)
{
    .
    .
    foo(PortS, a, b);
    .
    .
    exit(0);
}

----------------------
foo(port, x, y)
{
    .
    .
    .
    return(status);
}
```

Dear Server,
Please do foo(PortS, a, b).
Thank you,
Client@PortC

Client
PortC
Server
PortS

```
#include <stdio.h>

main(argc,argv)
{
    .
    .
    server_loop(PortS);
    .
    .
    exit(0);
}

----------------------
foo(port, x, y)
{
    .
    .
    .
    return(status);
}
```

Dear Client,
I successfully processed your request.
Server

Server
PortS
Client
PortC

Figure 4. MiG automatically generates RPC interface code for using the IPC port messaging system.

name server, which returns send rights to the server's port. By linking with the MiG–produced client stubs, the client can call the available server procedures.

The underlying transport mechanism, IPC, has been optimized for RPC transactions using an internal “fast path” [6][7]. A direct handoff of both the message and CPU control occurs from the source to the destination if the destination port has a thread waiting to receive from it. This optimization avoids synchronization, queueing operations, and scheduling latency altogether, while also allowing sharing of message parsing and locking and unlocking operations between the sender and receiver. When conditions are appropriate for the fast path, the latency of a null RPC call is cut almost in half.

Note that actual remote host RPC has been implemented on top of the local RPC mechanism in Mach 3 by using proxy ports which transparently provide network communication services, e.g. [14].

4. LRPC$_{\mathbf{Mach3}}$

Section 4.1 describes the design of LRPC$_{\text{Mach3}}$. Section 4.2 discusses how the design decisions reduce the costs of performing an RPC in Mach 3. Performance measurements comparing LRPC$_{\text{Mach3}}$ to MiG/IPC are provided in section 4.3. Section 4.4 describes future plans to more fully develop the LRPC$_{\text{Mach3}}$ facility.

4.1. Design

LRPC$_{\text{Mach3}}$ is designed specifically for Mach 3. It makes use of the concepts and techniques of the original LRPC, but is not a direct port of the original to Mach. The differences between LRPC$_{\text{Mach3}}$ and LRPC$_{\text{TAOS}}$ stem not only from differences in the target operating systems, but also differences in the target languages (C vs. Modula2+) and the desire for consistency with the existing Mach RPC interface, MiG. Nevertheless, LRPC$_{\text{Mach3}}$ addresses the same RPC overhead issues revealed by LRPC.

Like LRPC$_{\text{TAOS}}$ LRPC$_{\text{Mach3}}$ uses simple control and data transfer, simple stubs, and is designed for concurrency to reduce the main sources of overhead. The control transfer is made simple by using a dedicated server thread to execute the requested service in the server’s address space. The data transfer is simplified by using temporary kernel memory to provide a direct parameter passing path between the client and server. The client stub simply builds a heap to handle referenced parameters if necessary, then calls the LRPC$_{\text{Mach3}}$ kernel transport mechanism, while the server stub merely calls the reverse kernel transport mechanism. Finally, requests are handled through dedicated ports minimizing interactions that might inhibit concurrency.

The two most significant differences between the implementations of LRPC$_{\text{Mach3}}$ and LRPC$_{\text{TAOS}}$, are that LRPC$_{\text{Mach3}}$ neither uses the client’s thread to perform the server procedure, nor uses shared memory between the client and server. In particular, threads in Mach 3 are tightly interwoven with the tasks containing them, making it costly to remove the client thread from the client task and insert it into the server task. Doing so also opens up the server task to accidental or intentional assault by entities holding capabilities to the thread. Short of completely redesigning Mach’s thread scheduling, as in [8], thread migration is not currently a practical solution.

Furthermore, establishing a shared region of memory between two distinct tasks is fairly involved in Mach 3 [12], requiring either a) a common ancestor task, b) a common memory–mapped file, or c) privileged access. Additionally, sharing a stack between the client and server puts the server in the position of dealing with data which may be altered by other executing client threads, a safety issue that was not satisfactorily addressed in LRPC$_{\text{TAOS}}$. Yet our results show that the cost of the extra copying incurred without shared memory does not contribute significantly to the latency, so we have chosen not to complicate our implementation with this optimization for now.

To illustrate the design of LRPC$_{Mach3}$, we now examine the steps involved in binding to a server and performing an LRPC$_{Mach3}$ call. As with LRPC$_{TAOS}$, once bound to a server, a client may make requests by making a procedure call to the client stub. This prepares for and makes a kernel call. The kernel then transfers data and control to the server's address space, directly invoking the requested procedure. The server procedure returns to its own return stub which sets up for and makes the reply kernel call. The kernel returns out–going data and control to the client stub which completes by performing any necessary unmarshalling and returning.

When binding to a server, a client receives a number of *channel* ports (see figure 5). A channel port represents a kernel

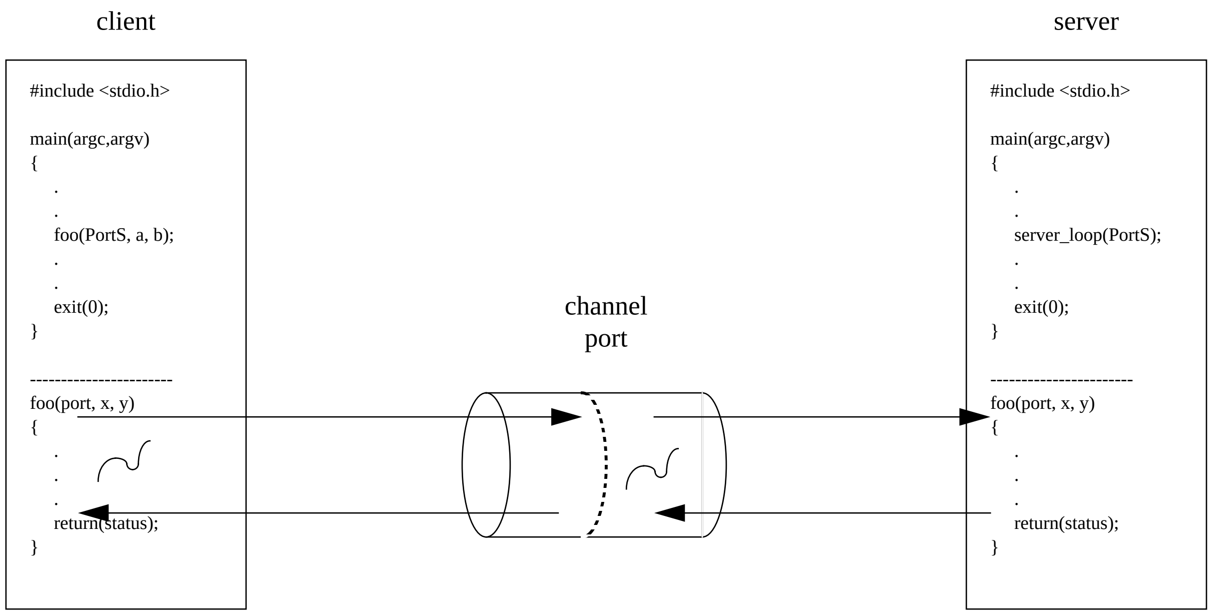


Figure 5. LRPC$_{Mach3}$ uses special channel ports to provide a more direct RPC based on the model of a procedure call.

object connecting the client to the server. The client task is free to manage these channel ports in any way (e.g., one per thread), but each is dedicated to one transaction at a time; additional requests on the same channel port block until the current transaction completes.

The client stubs are written in the C language to support C language calls. There is only a single stack in C, used for both arguments and local variables (unlike Modula2+, which uses separate argument and *execution* stacks). When the client stub is called the parameters are already arranged on the stack exactly as desired by the destined server procedure, so these values are just left in their (known) places. In general, some of these parameters are pointers referring to the actual values of interest. These referred–to values are called *referents*, and are specified in the RPC interface definition to be either *in–only, out–only,* or *in-out* referents, indicating in which direction they must be marshalled. Other parameters may be ports, which must change identity when passing from the client task to the server task. The MiG interface reserves the first parameter to be the server's port through which the RPC is being made (i.e. the channel port in LRPC$_{Mach3}$).

For the case of multiple pointer parameters, the client stub organizes the referents into a contiguous *in–out heap* (see

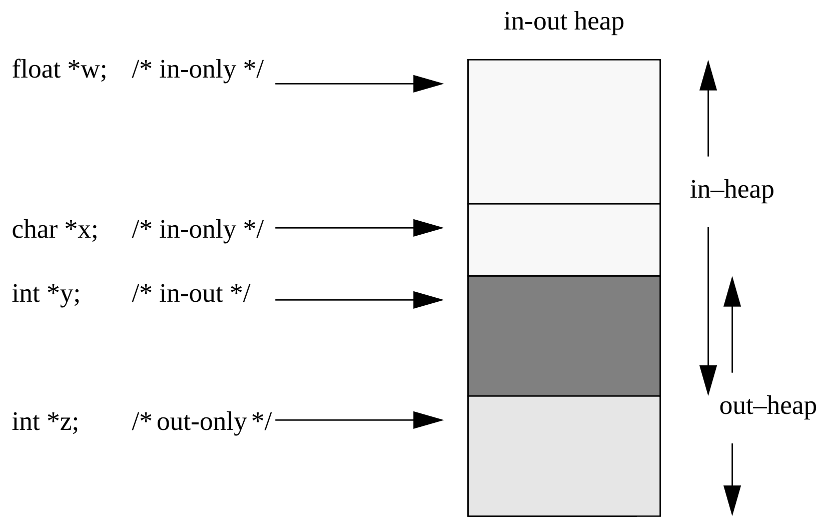


Figure 6. Multiple referents are organized into a single *in-out* heap. In–going referents are copied from the client to the server. Out–going referents are copied from the server to the client.

figure 6). In–only referents are copied to the first section of the heap, followed by in–out referents in the next section, with the final section left as space for the out–only referents. The call frame pointers are changed in place to point to their locations in the in–out heap, with the out–going pointers cached in local variables. As an optimization for the common case of a single pointer parameter, the referent itself serves as the in–out heap, so no allocation, copying, or pointer fix up need be done. Of course, none of this is necessary for procedures with no pointer parameters, but since for compatibility with MiG/IPC the return value is used exclusively to indicate success or failure of the RPC itself, pointer parameters are the only way to get information back from a server procedure [12].

The client stub next performs the forward LRPC$_{\text{Mach3}}$ kernel call, `lrpc_call()`, conveying the identifier of the requested procedure, pointers to the call frame and the in–out heap. To facilitate fast transformations, the client stub also gives the kernel a list of pointers to the pointer parameters on the call frame, and to any port names other than the first parameter, which is known to be the channel port.

In the context of the client (i.e. in the kernel with the client being the *active task*), `lrpc_call()` uses a fast Mach 3 algorithm to verify the channel port and reference the internal channel object. The in–going portion of the in–out heap (the *in–heap*) and the call frame are copied from the client's to the kernel's address space. Since the channel object contains a dedicated server thread with a stack, the kernel uses the location of the server stack in the server's address space to directly translate the call frame pointers while still in the client context. Similarly, by storing the server's task port and local channel port name in the channel object, the ports are also translated directly (the channel port being translated without lookup).

Control is then handed off directly to the channel's server thread (previously blocked in the kernel) changing the context to that of the server's, thus blocking the client thread. The saved user state of the server thread is set to resume at the desired server procedure, with its instruction, stack, and frame pointers set appropriately. The kernel finishes up by copying the call frame and in–heap to the server thread's stack, then exiting—returning to the user–level in the guise of the newly tailored server thread.

The server thread immediately executes the desired procedure upon leaving the kernel. Since there is no server stub wrapping the procedure, the return–path server stub code is in–lined with the procedure. This code simply invokes the reverse LRPC$_{\text{Mach3}}$ kernel call, `lrpc_return()`, conveying the channel port and return status.

`lrpc_return()` again uses the channel port to reference the internal channel object. The out–heap is then copied from the server's to the kernel's address space, and any port names are translated. Control is handed off to the blocked client thread; the kernel copy of the out–heap is copied to the client's out–heap; and the kernel returns control to the client thread.

Finally, the client stub finishes by unmarshalling the out–heap and returning the return status.

4.2. Discussion

LRPC$_{\text{Mach3}}$ addresses the overhead sources of RPC existing in MiG/IPC by several means:

- Stub overhead is reduced by using the existing call frame directly, by building a simple heap when necessary instead of stylized messages, and by directly invoking server procedures and trapping directly upon return, virtually eliminating server stub code. As an added benefit, using a single in–out heap avoids the MiG/IPC necessity of copying in–out referents from a request to a reply buffer in the server stub.
- Message buffer overhead is reduced by avoiding the creation or removal of a return port. The use of separate ports for each request eliminates the need for associated queues. By using free memory space in the kernel's stack and in the pre–allocated server stack, LRPC$_{\text{Mach3}}$ also avoids any buffer creation or management.
- The message transfer cost is reduced by taking advantage of the asymmetry of forward and reverse paths—keeping track of the information required by the reverse path in the channel to keep the reverse call simpler. The message transfer cost is reduced further by pointing out elements requiring transformation, rather than parsing message contents, allowing bulk rather than piece–wise copying. In addition, LRPC$_{\text{Mach3}}$ completely eliminates queueing latency, unlike IPC which only avoids queueing latency when on the "fast path."
- Scheduling costs are reduced and processor reallocation costs eliminated by directly handing off thread scheduling from the client to the server thread. Although this technique is also used by the IPC fast path, having a waiting server thread is a prerequisite; whereas LRPC$_{\text{Mach3}}$ guarantees that each channel has a waiting server thread. (Note that this does not inhibit concurrent execution of other threads within the client and server tasks.)
- Finally, the cost of server thread dispatch is eliminated by pre–allocating dedicated server threads upon creation of the channel ports. Keeping these blocked channel threads "lying around" is not particularly expensive, thanks to the work done to incorporate continuations into Mach 3's kernel (see [7]), relieving each thread of the need for a kernel stack.

Unlike, LRPC$_{\text{TAOS}}$, LRPC$_{\text{Mach3}}$ can't reduce the cost of access validation by not verifying access on the return trip since the channel port is exposed to both the client and the server, which may result in an access violation if the server doesn't leave its port argument intact.

To summarize, $\text{LRPC}_{\text{Mach3}}$ uses procedure–call mechanisms to provide fast local RPC for Mach 3. It provides semantic consistency with MiG[1]—the same server procedure code is used for both MiG/IPC and $\text{LRPC}_{\text{Mach3}}$. Concurrency is facilitated by distributing multiple entry ports, each having a ready thread to service it. And, unlike IPC's "fast path," $\text{LRPC}_{\text{Mach3}}$ doesn't require special circumstances for its performance improvements.

When designing $\text{LRPC}_{\text{Mach3}}$, recent proposed optimizations to the existing Mach 3 RPC mechanism were also reviewed. One such optimization is *port buffers* [10], a technique of grouping multiple RPC calls which can be performed on top of $\text{LRPC}_{\text{Mach3}}$ and benefit from the improved RPC performance. Another Mach 3 RPC optimization reviewed was *in–kernel servers* [11], which reduces RPC call latency by putting servers into the kernel's address space, limiting the facility to those with kernel access and defeating some of the benefits of the microkernel architecture. This work led to the exploration of the use of thread migration to speed up directed control transfer in [8].

4.3. Performance

$\text{LRPC}_{\text{Mach3}}$ is implemented on a Sequent multiprocessor containing 20 Intel i386 microprocessors. A single processor is used for all timing comparisons.

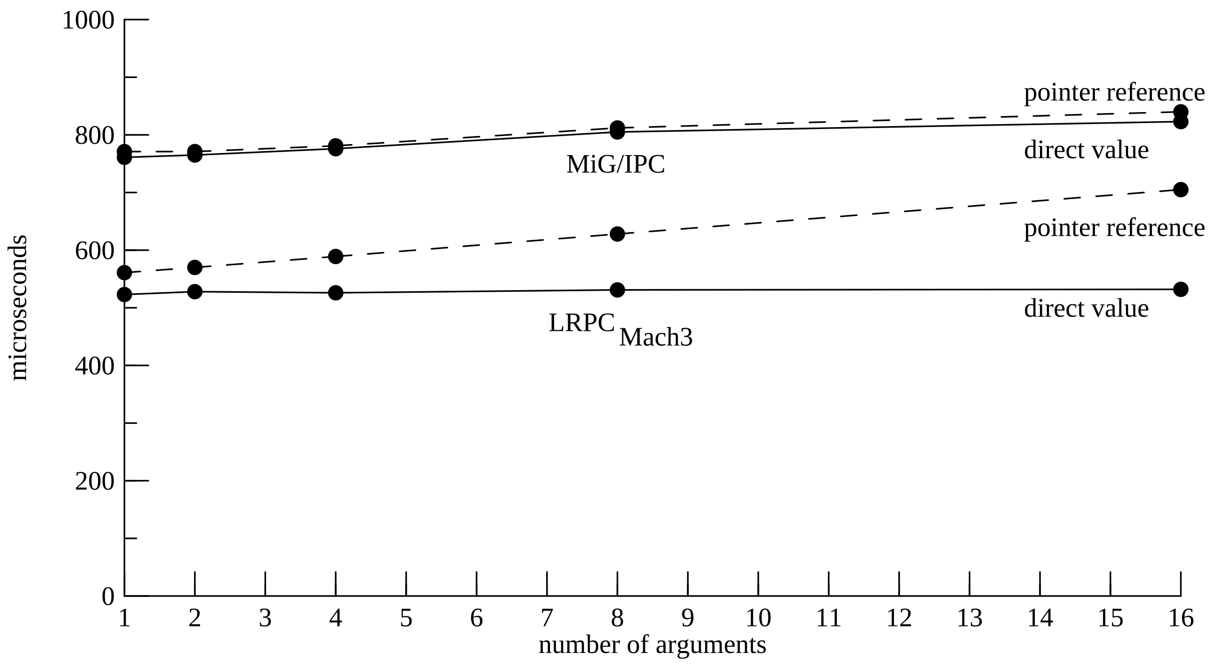


Figure 7. RPC call latency over a range of arguments. Each pointer parameter refers to a single, in-only 4-byte integer.

The fundamental performance measurement is call latency—the time required for a client to invoke and return from a server procedure. Call latency tests are performed for various numbers of direct value and pointer parameters (see

[1]. Specifically, $\text{LRPC}_{\text{Mach3}}$ supports the same RPC interface options as the MiG/IPC "fast path." Large and/or complicated interfaces are handled in both cases by the existing "normal path" of MiG/IPC.

figure 7), and for increasing sizes of single in–only, in–out, and out–only referents (figure 8). For the minimum call of one port argument (see figure 7— direct value), $LRPC_{Mach3}$ takes 523 microsecs versus 761 microsecs for MiG/IPC. This represents the minimum call latency overhead for both facilities. For this base case, $LRPC_{Mach3}$ achieves a 31% cost reduction over MiG/IPC.

The relative improvement of $LRPC_{Mach3}$ over MiG/IPC when using pointer parameters of a single integer referent is not as great, and shows a sharper increase in overhead as more are sent. This reflects the additional costs of marshalling and pointer transformations. MiG/IPC, on the other hand, shows only a slight increase in overhead.

Figure 8 shows how the two facilities are affected by changes in the size of referents. For $LRPC_{Mach3}$, as expected, the

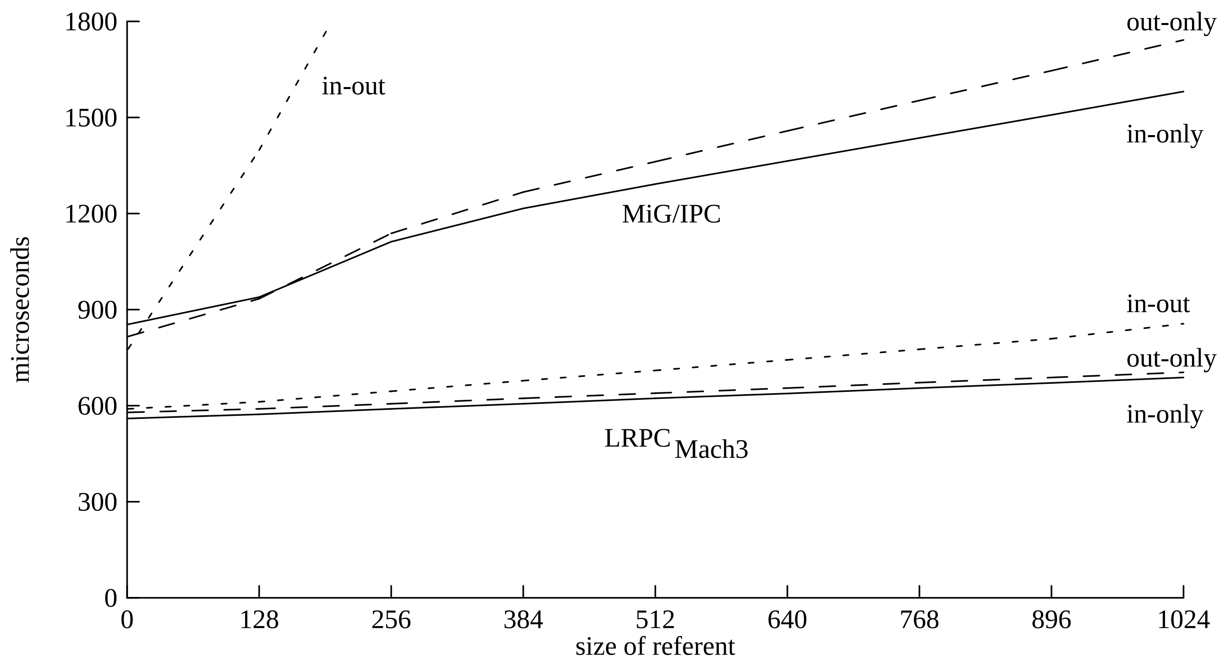


Figure 8. RPC call latency with a single referent of varying size.

rate of increase for in–out referents is very close to the sum of the rates of increase for in–only and out–only referents. On the other hand, the in–out results for MiG/IPC are clearly out of line with those of the in–only and out–only referents individually. This is partly, but not completely, accounted for by an extra server stub copy required to move the referent from the request to the reply message. These measurements show that $LRPC_{Mach3}$ moves additional referent data with a cost of about 125 nanoseconds per byte in either direction, compared to MiG/IPC's 710 nanosecs/byte for in–only referents and 905 nanosecs/byte for out–only referents.

4.4. Future Work

$LRPC_{Mach3}$ is not yet complete. We are currently developing an interface compiler to automatically create stub code and interface headers from existing MiG interface definition files, which we intend to use on existing servers to investigate performance improvements. We also intend to compare the sub-costs of the $LRPC_{Mach3}$ path to those of MiG/IPC to more clearly detail specific areas of improvement. Some bullet–proofing issues must also be more completely addressed, such as exception notification.

5. Conclusion

Mach 3's microkernel organization places increased demands on remote–procedure–call communication. $LRPC_{Mach3}$ successfully outperforms the current, highly–optimized, RPC facility in Mach 3 by a considerable margin. Furthermore, it achieves and sustains its performance advantage without regard to the call–time circumstances or the complexity of the call.

$LRPC_{Mach3}$ captures the essence of LRPC, while being specifically tailored to the Mach 3 operating system. It works seamlessly and cooperatively with the existing RPC mechanism, providing a practical and safe transition path to its incorporation. Above all, the success of $LRPC_{Mach3}$ shows that the performance benefits of LRPC are not just from its individual techniques, but from its view of RPC as primarily a *procedure–call* model, rather than a *message–passing* model.

6. References

[1] Accetta, M., R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevenian, and M. Young, "Mach: A New Kernel Foundation For UNIX Development," *Proceedings of the Summer 1986 USENIX Technical Conference and Exhibition*, June 1986, pp. 93-112.

[2] Bershad, Brian N., *High Performance Cross–Address Space Communication*, Ph.D. dissertation, Department of Computer Science and Engineering, University of Washington, Technical Report No. 90–06–02, June 1990.

[3] Bershad, Brian N., Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy, "Lightweight Remote Procedure Call," *ACM Transactions on Computer Systems*, v8(1), February 1990, pp. 37–55.

[4] Bershad, Brian N., Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy, "User–Level Interprocess Communication for Shared Memory Multiprocessors," *ACM Transactions on Computer Systems*, v9(2), May 1991, pp. 175–198.

[5] Birrell, Andrew D., and Bruce Jay Nelson, "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems*, v2(1), February 1984, pp. 39–59.

[6] Draves, Richard P., "A Revised IPC Interface," *Proceedings of the First Mach USENIX Workshop*, October 1990, pp. 101–121.

[7] Draves, Richard P., Brian N. Bershad, Richard F. Rashid, and Randall W. Dean, "Using Continuations to Implement Thread Management and Communication in Operating Systems," *Proceedings of the 13th ACM Symposium on Operating System Principles*, October 1991, pp. 122–136.

[8] Ford, Bryan, Mike Hibler, and Jay Lepreau, "Notes on Thread Models in Mach 3.0," Department of Computer Science, University of Utah, Technical Report No. UUCS-93-012, Salt Lake City, Utah, April 1993.

[9] Johnson, David B. and Willy Zwaenepoel, "The Peregrine High–performance RPC System," *Software—Practice and Experience*, Vol. 23(2), February 1993, pp. 201–221.

[10] Koontz, Kenneth W., "Port Buffers: A Mach IPC Optimization for Handling Large Volumes of Small Messages," *Proceedings of the USENIX Mach III Symposium*, Sante Fe, New Mexico, U.S.A., Usenix Association, Spring 1993, pp. 89–102.

[11] Lepreau, Jay, Mike Hibler, Bryan Ford, and Jeffrey Law, “In–Kernel Servers on Mach 3.0: Implementation and Performance,” *Proceedings of the USENIX Mach III Symposium*, Sante Fe, New Mexico, U.S.A., Usenix Association, Spring 1993, pp. 39–56

[12] Loepere, Keith, ed., *Mach 3 Kernel Principles*, Revision 2.2, Open Software Foundation, 1992

[13] Loepere, Keith, ed., *Mach 3 Server Writer's Guide*, Revision 2.2, Open Software Foundation, 1992

[14] Orman, Hilarie, Edwin Menze III, Sean O'Malley, and Larry Peterson, “A Fast and General Implementation of Mach IPC in a Network,” *Proceedings of the USENIX Mach III Symposium*, Sante Fe, New Mexico, U.S.A., Usenix Association, Spring 1993, pp. 75–88.

[15] Rashid, Richard F., “Threads of a New System,” *Unix Review*, v4, August 1986, pp. 37–49.

[16] Rashid, Richard F., “From RIG to Accent to Mach: The Evolution of a Network Operating System,” *The Ecology of Computation* (B.A. Huberman, ed.), North Holland, 1988, pp. 207–230.

[17] Rovner, P., R. Levin, and J. Wick, “On Extending Modula–2 for Building Large, Integrated Systems,” Technical Report #3, Digital Equipment Corporation's System Research Center, Palo Alto, California, April 1989.

[18] Schroeder, M. D., and M. Burrows, “Performance of the Firefly RPC,” *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, Litchfield Port, Arizona, U.S.A., ACM, December 3-6, 1989, pp. 83-90.

[19] Thacker, C. P., L. C. Stewart, and E. H. Satterthwaite, Jr., “Firefly: A Multiprocessor Workstation,” *IEEE Transactions on Computers*, v37(8), August 1988, pp. 909–920.

[20] Wahbe, Robert, Steven Lucco, Thomas Anderson, and Susan Graham, “Efficient Software–Based Fault Isolation,” Proceedings of the 14th ACM Symposium on Operating System Principles, December 1993, pp. 203-216.

[21] White, J. E., “A High–level Framework for Network–based Resource Sharing,” *Proceedings of the National Computer Conference*, June 1976.