Lightweight Remote Procedure Call

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

ACM Transactions Vol. 8, No. 1,
February 1990, pp. 37-55

presented by Ian Dees
for PSU CS533, Jonathan Walpole
February 2, 2009

This paper proposes a way for operating systems to take advantage of RPC-style programming techniques inside the kernel.
sharing resources inside the kernel

Modern OSes have mechanisms to protect resources as they’re shared among user apps, or between apps and the kernel. We’ve seen several of those mechanisms in class so far. What this paper addresses is how different parts of the kernel share resources internally.
It’s worth taking a moment to look at the concept of protection domains used in the paper. These are a bit like address spaces; they’re basically the walls behind which you separate the parts of the system you want to protect from each other. Microkernel systems actually do use different address spaces to separate concerns; the subsystems communicate via the same sorts of mechanisms used by distributed systems.
remote procedure call

inspired by distributed computing
separate address spaces
coarse-grained access
messages wrapped in stubs and proxies

By the nature of running in different address spaces (on different machines), distributed systems using RPC must marshal the parameters of a function call into a format that can be transferred from one space to another. Complex data structures with pointers require particular care. Code generation can ease some of this boilerplate programming burden.
applying RPC ideas locally

RPC model “appropriate for managing subsystems, even those not primarily intended for remote operation”

but how to make message passing efficient?

OSes have adopted RPC–like techniques internally, isolating subsystems in separate address spaces and passing parameters around in messages. One big concern with this approach is performance: making a lot of context switches and marshaling a lot of data structures into messages will add overhead. Typical systems sacrifice either performance or purity of the protection scheme.
Where do these inefficiencies from from? It’s tempting to create a system to treat “localhost” as just another networked host, and to treat simple numeric parameters as structs that just happen to have only one member. But such an approach ignores the way real–world OSes run.
Instead, the authors looked at actual communication patterns inside OSes, so that they could propose optimizations for the most common cases.
It turns out that most OS communications occur inside one machine, even on distributed systems.
Here’s a loose confirmation of this distribution based on a survey they did of running OSes.  (They had to wave their hands a bit with the definition of RPC on the UNIX system.)

<table>
<thead>
<tr>
<th>Operating system</th>
<th>Percentage of operations that cross machine boundaries</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>3.0</td>
</tr>
<tr>
<td>Taos</td>
<td>5.3</td>
</tr>
<tr>
<td>Sun UNIX+NFS</td>
<td>0.6</td>
</tr>
</tbody>
</table>
most parameters are simple

even in complex APIs

It also turns out that the majority of parameters passed are simple scalar values like numbers and booleans. Even complex APIs tend to use more and smaller parameters, rather than giant structs.
Here is the distribution of parameter sizes in a run of Taos OS, expressed as the total size of the argument list. As is often the case in a “long-tail”–like Poisson distribution, the majority of calls were to the same few functions, using the same few parameters. 65% of the calls transferred four bytes or fewer.
Identifying the common case allowed the authors to target specific types of overhead to eliminate. The biggest sources of delay were serializing, copying, and queuing parameters; scheduling and context-switching to a new thread; and all the boilerplate stub code that has to run for every call.
sender validation

when should the kernel check permissions?

It’s worth taking a slightly closer look at one of these sources of inefficiency, sender validation. A vital part of transferring control is verifying that the client has permission to access the server, and that the server has permission to transfer control back to the client. This coupling of addressing and permissions is conceptually similar to capability systems.
Capability systems protect resources by requiring all access to be through “protected procedure calls;” think of a POSIX file handle, or a pointer coupled with an extremely specific list of permissions. Some early systems actually used hardware to tag this meta-data onto values. In contrast to blanket read-write access schemes, tokens carry the minimum set of permissions needed to complete a task. Like the valet-only key to a car, capabilities are specific, are persistent, and can be transferred or copied without jeopardizing the resource they protect.
Some operating systems had already made headway against some of the many sources of RPC overhead. DASH still uses marshaling, but at least saves a message copy by mapping the serialized data into both processes’ address spaces. Mach and Taos use the caller’s thread to execute code in the server’s domain, thus saving a context switch. SRC RPC allocates all message buffers out of one global pool.
LRPC’s four techniques

To deal with a broader set of overhead sources, the authors propose a host of related optimizations, which fall into four main classes of techniques.
Rather than go through a broker on every single call, an LRPC client uses an OS-supplied “clerk” to get a function address and argument stack size once, and is then able to reuse this “binding” object (again, these are conceptually related to capabilities) from call to call. When a client makes a cross-domain call, the kernel writes the server’s execution stack and registers into the caller’s thread. Reusing the thread saves scheduler and context-switch overhead (but note that saving/restoring registers is a little like a partial context switch).
2. simple data transfer

map argument stacks into both spaces
aggressively reduce copies
(but call-by-reference involves an extra copy)
fall back on marshaling

Rather than serializing parameters into a message, queueing the message, and then taking it apart again, LRPC maps the argument list into the memory spaces of both client and server. The authors’ choice of the Modula-2+ calling convention (which uses a chunk of memory separate from the execution stack) made this optimization possible. For more complex data structures, they fall back on traditional RPC-style marshaling.
The kernel pre-fills the execution stack for the procedure being called, so the server-side stub merely has to jump to the first instruction. For the simple case of direct invocation using small parameters, the authors generate hand-coded assembly consisting of little more than branch instructions and system calls. More complex cases fall back on the traditional RPC-style stubs to perform marshaling and exception handling.
4. design for concurrency

distribute domains across processors
monitor concurrency misses and wake CPUs
use fine-grained locking of argument stacks

LRPC takes advantage of multiprocessor machines by distributing protection domains across processors. When a client calls a server whose thread is idling on a different processor, the kernel switches the calling thread to the new processor, thus saving register rewrites. Returning to the client can involve a similar jump across CPUs. The kernel can actively monitor processors and domains to optimize at runtime.
There are a few edge cases to consider. Since LRPC assumes local calls are the norm, remote calls now involve one more level of indirection; however, the delay of a single function call and flag check is insignificant in the face of a network round-trip. Parameters too large to fit in the activation stack must be copied into a buffer; this has performance impacts similar to RPC approaches based on message queues. Finally, handoff scheduling means that a terminating server could inadvertently hijack a client’s thread; LRPC allows spinning up an exception-handling thread on the client before the call.
The authors have implemented the ideas behind LPRC on the C-VAX Firefly system, and compared them with Taos RPC on the same system.
On a set of representative functions called in a tight loop, LRPC outperformed traditional RPC across the board, but of particular interest is the common case of a function taking and returning a “small” amount of data (eight bytes in, four bytes out), where LRPC was faster by a factor of three.

It would have been interesting to instrument both systems and let them run a typical set of a few hundred day-to-day computations, rather than a single, repeatedly looped call.
LRPC’s call throughput scaled nearly linearly with the number of processors, which is in line with the authors’ efforts to reduce lock contention for the multiprocessor case.