Lightweight Remote Procedure Call (LRPC)
Overview

Observations
Performance analysis of RPC
Lightweight RPC for *local* communication
Performance
Remote Procedure Call

An IPC mechanism for invoking a subroutine in another address space
- The receiver might reside on the same physical system or over a network

Provides a large-grained protection model

The call semantics make it appear as though only a normal procedure call was performed
- Stubs interface to a runtime environment, which handles data marshalling
- OS handles low-level IPC
- Protection domain boundaries hidden by stubs
Steps in a Traditional RPC

- Client Application
- Client Stub
- Client Runtime Library
- Client Kernel

- Server Application
- Server Stub
- Server Runtime Library
- Server Kernel

Potentially shared in a single-system RPC

transport layer
RPC in Microkernel Systems

Small-kernel systems use RPC

- Separate components are placed in disjoint address spaces (protection domains)
- Communication between components uses RPC (message passing)

Advantages: modularity, simplification, failure isolation, and distribution transparency

But the control transfer facilities of RPC are not optimized for same-machine control transfer

- This leads to inefficiency
Monolithic Kernel based Operating System

- Application
- System Call
- VFS
- IPC, File System
- Scheduler, Virtual Memory
- Device Drivers, Dispatcher, ...

Microkernel based Operating System

- Application IPC
- UNIX Server
- Device Driver
- File Server
- Basic IPC, Virtual Memory, Scheduling

Hardware
The Use of RPC Systems

The common case for RPC:
- is cross-domain (not cross-machine)
- involves relatively simple parameters
- can be optimized

Frequency of Cross-Machine Activity

<table>
<thead>
<tr>
<th>Operation 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>
Parameter Size and Complexity

- 1,487,105 cross-domain procedure RPCs observed during one four-day period
- 95% were to 10 procedures; 75% were to 3 procedures
- None of them involved complex arguments
- Most involve a relatively small amount of data transfer

![RPC size distribution](image-url)

Fig. 1. RPC size distribution.
Performance of Cross-Domain RPC

The theoretical minimum time for a null cross-domain operation includes time for
- Two procedure calls
- Two traps
- Two virtual memory context switches

The cross-domain performance, measured across six systems using the Null RPC, varies from over 300% to over 800% of the theoretical minimum.
Overhead in Cross-Domain RPC

**Stub Overhead**: stubs allow cross-machine RPC, but inefficient for the common case of local RPC calls

**Message Buffer Overhead**: client/kernel, kernel/server, server/kernel, kernel/client

**Access Validation**: the kernel must validate the message sender on call and again in return

**Message Transfer**: messages are enqueued by the sender and dequeued by the receiver

**Scheduling**: separate, concrete threads run in client and server domains

**Context Switch**: in going from client to server

**Dispatch**: the server must receive and interpret the message
Lightweight RPC (LRPC)

LRPC aims to improve the performance of cross-domain communication relative to RPC. The execution model is borrowed from a protected procedure call:
- Control transfer proceeds by way of a kernel trap; the kernel validates the call and establishes a linkage.
- The client provides an argument stack and its own concrete thread of execution.

The programming semantics and large-grained protection model are borrowed from RPC:
- Servers execute in private protection domains.
- Each one exports a specific set of interfaces to which clients may bind.
- By allowing a binding, the server authorizes a client to access its procedures.
LRPC High-Level Design

Client Context

Control Flow
- thread

Virtual Memory
- A-stack

Server Context

Control Flow
- E-stack

Physical Memory

Kernel

Sending path
- return path
Implementation Details

Execution of the server procedure is made by way of a kernel trap.
The client provides the server with an argument stack and its own concrete thread of execution.
The argument stacks (A-stacks) are shared between client and server; the execution stacks (E-stacks) belong exclusively in the server domain.
- A-stacks and E-stacks are associated at call time.
- Each A-stack queue is guarded by a single lock.
The client must bind to an LRPC interface before using it; binding:
  - establishes shared segments between client and server.
  - allocates bookkeeping structures in the kernel.
  - returns a non-forgeable binding object to the client, which serves as the key for accessing the server (recall capability systems).

On multiprocessors, domains are cached on idle processors (to reduce latency).
Performance

The measurements below were taken across 100,000 cross-domain calls in a tight loop.
LRPC/MP uses the domain-caching optimization for multiprocessors.
LRPC performs a context switch on each call.

<table>
<thead>
<tr>
<th>Test</th>
<th>Description</th>
<th>LRPC/MP</th>
<th>LRPC</th>
<th>Tao</th>
</tr>
</thead>
<tbody>
<tr>
<td>Null</td>
<td>The Null cross-domain call</td>
<td>125</td>
<td>157</td>
<td>464</td>
</tr>
<tr>
<td>Add</td>
<td>A procedure taking two 4-byte arguments and returning one 4-byte argument</td>
<td>130</td>
<td>164</td>
<td>480</td>
</tr>
<tr>
<td>BigIn</td>
<td>A procedure taking one 200-byte argument</td>
<td>173</td>
<td>192</td>
<td>539</td>
</tr>
<tr>
<td>BigInOut</td>
<td>A procedure taking and returning one 200-byte argument</td>
<td>219</td>
<td>227</td>
<td>636</td>
</tr>
</tbody>
</table>
Discussion Items

When the client thread is executing an LRPC, does the scheduler know it has changed context?

What is a context switch?