Lightweight Remote Procedure Call


Presented by Tim Chevalier
for CS 533, Concepts in Operating Systems
April 22, 2009
RPC is a protection mechanism

Design systems by putting different subsystems in different protection domains

RPC was originally used in large-kernel OSes

Different protection domains on different machines

Machines communicate with each other over a network using RPC
Unintended consequences

- RPC caught on in small-kernel OSes, too
  - Different subsystems still lived in different protection domains
    - But all protection domains lived on the same machine
  - RPC as way to enforce protection vs. RPC as means for communicating over a network
RPC is overkill

RPC was designed for communication over a network

So for calls within the same machine, it suffers from overly general and thus overly expensive machinery

As a result, system designers used RPC only for communication between applications and kernel, and put all kernel subsystems into the same protection domain

Want to make RPC fast enough that different kernel subsystems can talk to each other through RPC, too
The common case

Small arguments
Fixed-size arguments
Simple arguments
Calls: Traditional RPC

1. Client domain
2. Kernel
3. Server domain
4. Server stack

RPC messages

Client stack
Sources of overhead

<table>
<thead>
<tr>
<th>In theory:</th>
<th>In reality:</th>
</tr>
</thead>
<tbody>
<tr>
<td>- One procedure call</td>
<td>Stub overhead</td>
</tr>
<tr>
<td>- Two kernel traps</td>
<td>Message buffers</td>
</tr>
<tr>
<td>- Two VM context switches</td>
<td>Access validation</td>
</tr>
<tr>
<td></td>
<td>Message transfer</td>
</tr>
<tr>
<td></td>
<td>Scheduling</td>
</tr>
<tr>
<td></td>
<td>VM context switches</td>
</tr>
<tr>
<td></td>
<td>Dispatch</td>
</tr>
</tbody>
</table>
Traditional RPC: interfaces

A server exports an interface

A client binds to that interface

An interface lists all the procedures that the server wants to let clients call
LRPC: binding
Calling

Client stub manipulates these

Arg stack

Execution stack

Server stub runs here

Client domain

Kernel

Server domain

Return address

Linkage
Argument copying

Only 1 copy!

Client Stub's stack

arg

Shared A-stack

Client's domain

Server's domain
Multiprocessors

Exploit multiprocessors to minimize context switching

Minimize shared data structures, thus maximizing concurrency

Cache domain contexts with multiple processors
Stubs

1. Kernel trap
2. Direct jump
3. Client stub returns

Client stub
Client code

Kernel

Server stub
Linkage record
Avoid needless scheduling

Running client thread in server domain
Processor switching avoids CPU context switches
Avoid needless indirection

Simple stub design lets the kernel jump to the server entry stub directly, avoiding the indirection of an extra procedure call

The argument and execution stacks are the key to avoiding this indirection: threads and stacks are decoupled.
Avoid redundant access validation

Binding Object bypasses extra validation after the initial binding

Stacks avoid need to verify thread’s right to return to caller
Avoid needless copying

Pairwise (shared by client and server) allocation of A-stacks avoids copying

Server and client share a private communication channel

Client also copies results directly off A-stack
Avoid needless contention

No shared data structures except for A-stack queues
One A-stack queue for each procedure in each binding
Corner cases

Expect arguments to be small, and handle larger ones specially
Corner cases

Servers may terminate at any time

Linkage stacks allow pending calls to server to return to their caller
Summary

Performance of LRPC is close to theoretical minimum

LRPC optimizes for common cases and minimizes work:
  simple calls
  simple stubs
  good use of multiprocessors

  minimal argument copying
  minimizes thread context switches
  avoids indirection
  minimizes access validation
  avoids contention

Does well for the majority of cases.