sharing resources inside the kernel
“protection domains”
like address spaces; may or may not be address spaces
remote procedure call

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

treat local as a special case of remote

treat atomic parameters as a special case of structured data
instead, make the common case fast
most calls are local

even in distributed systems
<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
Fig. 1. RPC size distribution.
sources of overhead

- stubs
- message copying
- sender validation
- message queueing
- thread scheduling
- context switching
- thread dispatch
sender validation

when should the kernel check permissions?
(capability systems)

single address space for parameter passing
fine-grained: token ≈ pointer + permissions
principle of least privilege
existing optimizations

memory mapping
handoff scheduling
global message buffers
LRPC’s four techniques
I. simple control transfer

look up addresses once, at discovery time (clerks and bindings)

server uses caller’s thread

return directly to caller

cache execution stacks for reuse
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
3. simple stubs

server stub branches straight to procedure
generate simple assembly for common cases
exceptions, large parameters fall back to RPC
4. design for concurrency

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

remote vs. local
activation stack size limits
domain termination
results
Table IV. LRPC Performance of Four Tests (in microseconds)

<table>
<thead>
<tr>
<th>Test</th>
<th>Description</th>
<th>LRPC/MP</th>
<th>LRPC</th>
<th>Taos</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>

Table V. Breakdown of Time (in microseconds) for Single-Processor Null LRPC

<table>
<thead>
<tr>
<th>Operation</th>
<th>Minimum</th>
<th>LRPC overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modula2+ procedure call</td>
<td>7</td>
<td>—</td>
</tr>
<tr>
<td>Two kernel traps</td>
<td>36</td>
<td>—</td>
</tr>
<tr>
<td>Two context switches</td>
<td>66</td>
<td>—</td>
</tr>
<tr>
<td>Stubs</td>
<td>—</td>
<td>21</td>
</tr>
<tr>
<td>Kernel transfer</td>
<td>—</td>
<td>27</td>
</tr>
<tr>
<td>Total</td>
<td>109</td>
<td>48</td>
</tr>
</tbody>
</table>
Fig. 2. Call throughput on a multiprocessor. LRPC measured calls versus number of processors.

- LRPC Optimal
- LRPC Measured
- RPC Optimal
- RPC Measured

Y-axis: Calls per Second
X-axis: Number of Processors