Lightweight Remote Procedure Call

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

Presented by Alana Sweat
Outline

• Introduction
  • RPC refresher
  • Monolithic OS vs. micro-kernel OS

• Use and Performance of RPC in Systems
  • Cross-domain vs. cross-machine
  • Problems with traditional RPC used for cross-domain RPC

• Lightweight RPC (LRPC)
  • Implementation
  • Performance

• Conclusion
Introduction
What is an RPC?

An inter-process communication that allows a computer program to cause a subroutine or procedure to execute in another address space without the programmer explicitly coding the details for this remote interaction.


Monolithic kernel & Micro-kernel OSs

http://en.wikipedia.org/wiki/Monolithic_kernel
Monolithic kernel OS

• Advantages
  • All parts of kernel have easy access to hardware
  • Easy communication between kernel threads due to shared address space

• Disadvantages
  • Increasingly complex code as kernel grows, difficult to isolate problems and add/remove/modify code
  • Large amount of code having direct access makes hardware more vulnerable
Micro-kernel OS

• Advantages
  • Since modules are in user space, relatively easy to add/remove/modify functionality to operating system
  • Hardware is only accessed directly by small amount of protected kernel code
  • Completely separate modules helps with isolating problems & debugging
  • Each module in its own “protection domain”, since can only access its own address space

• Disadvantages
  • User-level modules must interact with each other over separate address spaces, difficult to achieve good performance
Use and Performance of RPC in Systems
Cross-domain RPC (local RPC)

• Local remote procedure call
  • Remote since it accessing a “remote” address space, local because it is a procedure call on the same machine

• General RPC model used for inter-process communication (IPC) in micro-kernel systems
Comparatively, how often does a system execute cross-machine RPC vs. cross-domain RPC?

<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>

*Measured over 5-hr period on work day for Taos, over 4 days for Sun workstation*
Size and complexity of cross-domain RPCs

- Survey includes 28 RPC services defining 366 procedures w/ 1000+ parameters over four-day period using SRC RPC on Taos OS
Why not just use standard RPC implementation for cross-domain calls?

<table>
<thead>
<tr>
<th>System</th>
<th>Processor</th>
<th>Null (theoretical minimum)</th>
<th>Null (actual)</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accent</td>
<td>PERQ</td>
<td>444</td>
<td>2,300</td>
<td>1,856</td>
</tr>
<tr>
<td>Taos</td>
<td>Firefly C-VAX</td>
<td>109</td>
<td>464</td>
<td>355</td>
</tr>
<tr>
<td>Mach</td>
<td>C-VAX</td>
<td>90</td>
<td>754</td>
<td>664</td>
</tr>
<tr>
<td>V</td>
<td>68020</td>
<td>170</td>
<td>730</td>
<td>560</td>
</tr>
<tr>
<td>Amoeba</td>
<td>68020</td>
<td>170</td>
<td>800</td>
<td>630</td>
</tr>
<tr>
<td>DASH</td>
<td>68020</td>
<td>170</td>
<td>1,590</td>
<td>1,420</td>
</tr>
</tbody>
</table>
Overhead in cross-domain RPC

• Stub overhead
  • execution path is general, but much code in path is not needed for cross-domain

• Message Buffer management
  • Allocate buffers; copies to kernel and back

• Access validation
  • Kernel validates message sender on call and again on return

• Message transfer
  • Enqueue/dequeue messages

• Scheduling
  • Programmer sees one abstract thread crossing domains; kernel has threads fixed in their own domain signaling each other

• Context switch
  • Swap virtual memory from client’s domain to server’s domain and back

• Dispatch
  • Receiver thread in server domain interprets message and dispatches thread to execute the call
Lightweight RPC (LRPC)
What is LRPC?

- Modified implementation of RPC optimized for cross-domain calls
- Execution model borrowed from protected procedure call
  - Call to server procedure made by kernel trap
  - Kernel validates caller, creates a linkage, dispatches client’s thread directly to server domain
  - Client provides server with argument stack along with thread
- Programming semantics borrowed from RPC
  - Servers execute in private protection domain & export 1+ interfaces
  - Client binds to server interface before starting to make calls
  - Server authorizes client by allowing binding to occur
Implementation Details

• Binding
  • Kernel allocates A-stacks (argument stacks) in both client and server domains for each procedure in the interface which are shared & read/write
  • Procedures can share A-stacks (if of similar size) to reduce storage needs
  • Kernel creates linkage record for each A-stack allocated to record caller’s return address (kernel accessible only)
  • Kernel returns Binding Object containing key for accessing server’s interface & A-stack list (for each procedure) to client
Implementation Details

• Client calls into stub, which:
  • Takes A-stack off of stub-managed A-stack queue & pushes client’s arguments onto it
  • Puts address of A-stack, binding object, & procedure ID into registers
  • Traps to the kernel

• Kernel then:
  • Verifies binding object, procedure ID, A-stack & linkage
  • Records caller’s return address and stack pointer in the linkage
  • Updates thread’s user stack pointer to run off an Execution stack (E-stack) in the server’s domain & reloads processor’s virtual memory registers with those of server domain
  • Does an upcall into the server’s stub to execute the procedure
Implementation Details

• Returning
  • Server procedure returns through its own stub
  • No need to verify Binding Object, procedure identifier, and A-stack (already in the linkage and not changed by server return call)
  • A-stack contains procedure’s return values
Optimizations

• Separate code paths for cross-machine vs. cross domain calls, and distinction made from first instruction executed in stub

• Keep E-stacks allocated and associated with A-stacks, only allocate new E-stack when none unassociated available

• Each A-stack queue (per procedure) has its own lock, so minimum contention in multi-threaded scenario

• In multiprocessor systems, kernel caches domain contexts for idle processors
  • After LRPC call is made, kernel checks for processor idling in context of server domain
  • If found, kernel exchanges processors of calling & idling threads, & server procedure can execute without requiring context switch
A Note about A-stacks and E-stacks

• Modula2+ language has the convention that procedure calls use a separate argument pointer instead of requiring the arguments be pushed onto the execution stack

• Different threads cannot share E-stacks, but because of the convention used it is safe to share A-stacks

• If LRPC was implemented in a language where E-stacks have to contain arguments (such as C), the optimization of shared A-stacks would not be possible (thus arguments would need extra copies)
Performance of LRPC

- Ran on Firefly using LRPC & Taos RPC
- 100,000 cross domain calls in tight loop, averaged time
- LRPC/MP uses idle processor domain caching, LRPC does context switch on every call on single processor

<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>
Conclusion
Conclusion

• Cross-domain RPC calls are significantly more common than cross-machine RPC calls

• Significant amount of extra overhead in standard RPC execution path when used for cross-domain calls

• LRPC eliminates many sources of overhead by creating a separate version of RPC that is optimized for cross-domain calls (arguably the common case of RPC)

• LRPC was shown to improve cross-domain RPC performance by a factor of 3 (in the Firefly/Taos system) over Taos RPC