Lightweight RPC

Robert Grimm
New York University
Altogether Now:
The Three Questions

- What is the problem?
- What is new or different?
- What are the contributions and limitations?
Monolithic kernels
  - Hard to modify, debug, validate
Fine-grained protection through capabilities
  - Relies on protected procedure calls
  - Is difficult to implement efficiently and program
Large-grained protection through machine boundaries
  - Relies on *remote procedure calls*
Small kernels (think Mach)
  - Rely on user-space servers for most functionality
  - Adopt programming models of distributed computing
Problem Statement and Approach

- Small kernels use distributed programming models
- Common case of communications is not across net but rather across domains on the same machine
- Optimize for the common case
  - Simple control transfer
  - Simple data transfer
  - Simple stubs
  - Design for concurrency
RPC (by Hank Levy)
Remote Procedure Call

- The basic model for Remote Procedure Call (RPC) was described by Birrell and Nelson in 1980, based on work done at Xerox PARC.
- Goals was to make RPC look as much like local PC as possible.
- Used computer/language support.
- There are 3 components on each side:
  - a user program (client or server)
  - a set of stub procedures
  - RPC runtime support
Basic process for building a server:

- Server program defines the server’s interface using an interface definition language (IDL)
- The IDL specifies the names, parameters, and types for all client-callable server procedures
- A stub compiler reads the IDL and produces two stub procedures for each server procedure: a client-side stub and a server-side stub
- The server writer writes the server and links it with the server-side stubs; the client writes her program and links it with the client-side stubs.
- The stubs are responsible for managing all details of the remote communication between client and server.
RPC Stubs

- Basically, a client-side stub is a procedure that looks to the client as if it were a callable server procedure.
- A server-side stub looks to the server as if it’s a calling client.
- The client program thinks it is calling the server; in fact, it’s calling the client stub.
- The server program thinks it’s called by the client; in fact, it’s called by the server stub.
- The stubs send messages to each other to make the RPC happen.
**RPC Call Structure**

- **Client Program**
  - `call foo(x,y)`
  - `call foo` (to stub)
  - `proc foo(a,b)`
  - `send msg`

- **Client Stub**
  - `proc foo(a,b)`
  - `stub builds msg packet, inserts params`

- **RPC Runtime**
  - `runtime sends msg to remote node`

- **Server Program**
  - `proc foo(a,b)`
  - `server is called by its stub`
  - `begin foo...`
  - `end foo`

- **Server Stub**
  - `call foo(x,y)`
  - `stub unpacks params and makes call`

- **RPC Runtime**
  - `runtime receives msg and calls stub`
  - `msg received`
RPC Return Structure

- **client program**
  - `call foo(x,y)`
  - `proc foo(a,b)`
  - `msg received`
  - `runtime receives msg, calls stub`
  - `return`

- **client stub**
  - `proc foo(a,b)`
  - `stub unpacks msg, returns to caller`
  - `return`

- **RPC runtime**
  - `msg received`
  - `runtime receives msg, calls stub`
  - `return`

- **server proc**
  - `proc foo(a,b)`
  - `begin foo...`
  - `stub builds result msg with output args`
  - `stub unpacks msg, returns to caller`

- **server program**
  - `call foo(x,y)`
  - `return`
  - `send msg`

- **RPC runtime**
  - `runtime responds to original msg`
  - `return`
RPC Binding

- Binding is the process of connecting the client and server.
- The server, when it starts up, *exports* its interface, identifying itself to a network name server and telling the local runtime its dispatcher address.
- The client, before issuing any calls, *imports* the server, which causes the RPC runtime to lookup the server through the name service and contact the requested server to setup a connection.
- The *import* and *export* are explicit calls in the code.
RPC Marshalling

- Marshalling is the packing of procedure parameters into a message packet.
- The RPC stubs call type-specific procedures to marshal (or unmarshal) all of the parameters to the call.
- On the client side, the client stub marshalls the parameters into the call packet; on the server side the server stub unmarshalls the parameters in order to call the server’s procedure.
- On the return, the server stub marshalls return parameters into the return packet; the client stub unmarshalls return parameters and returns to the client.
RPC is the most common model now for communications in distributed applications.

RPC is essentially language support for distributed programming.

RPC relies on a stub compiler to automatically produce client/server stubs from the IDL server description.

RPC is commonly used, *even on a single node*, for communication between applications running in different address spaces. *In fact, most RPCs are intra-node.*
Back (Well, Forward) to LRPC
Use of RPC

- Most RPCs are cross domain
  - 3.0% on V, 5.3% on Taos, 0.6% on Sun+NFS
- Most RPCs transfer little data
  - On Taos, 3 procedures account for 75% of all RPCs
    - No complex marshalling, byte copy suffices
Overheads of RPC (When Compared to Null Proc)

- Stub overhead
- Message buffer overhead
- Access validation
- Message transfer
- Scheduling
- Context switch
- Dispatch
- What about SRC RPC?

Table II. Cross-Domain Performance (times are in microseconds)

<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>
Enter LRPC

- Call to server made through kernel trap
- Kernel validates caller, creates a linkage, dispatches concrete thread to server
- Client provides argument stack and thread
- Procedure returns through kernel
LRPC Binding

- Model comparable to regular RPC
  - Server exports interface to name server
  - Clients import interface
- Details differ
  - Procedure descriptor (PD)
    - Entry address, number of simultaneous calls, size of A-stack
    - Pair-wise shared memory for arguments and return values
  - Linkage record
    - Record of caller’s return address
  - Binding object
    - Capability for accessing server’s interface
LRPC Calls

- Client stub sets up A-stack, calls kernel
- Kernel
  - Verifies binding object, procedure identifier, locates PD
  - Verifies A-stack, locates linkage record
  - Ensures that A-stack/linkage record are unused
  - Records caller’s return address in linkage record
  - Pushes linkages record on per-thread stack
  - Locates execution stack ($E$-stack) in server’s domain
  - Updates stack pointer to use E-stack
  - Changes virtual memory registers
  - Performs upcall into server
LRPC Calls (cont.)

- Return through kernel
  - No verification of rights, data structures
  - No explicit message passing
- Call-by-reference requires local reference (why?)
- E-stacks dynamically associated with A-stacks
  - Association performed on first call with given A-stack
  - E-stacks reclaimed when supply runs low
  - Why no static association?
More Details

- Stubs blur boundaries of traditional RPC layers
  - Direct invocation of server stubs, no message dispatch
  - Simple LRPCs require one procedure call, two returns
- LRPC designed for multi-processors
  - Each A-stack queue guarded by its own lock
  - Domains cached on idle processors
    - Processors changed on LRPC (in both directions)
    - Context switch only performed when domain not cached
    - Generalized technique (Amoeba, Taos cache blocked threads)
LRPC Argument Copying

- Copying performed in stubs, not in kernel
  - 4 times for RPC, once for LRPC in common case
- Shared memory allows for asynchronous change
  - Extra copy for immutable parameters
  - Constraint check folded into copy operation

### Table III. Copy Operations for LRPC versus Message-Based RPC

<table>
<thead>
<tr>
<th>Operation</th>
<th>LRPC</th>
<th>Message passing</th>
<th>Restricted message passing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call (mutable parameters)</td>
<td>A</td>
<td>ABCE</td>
<td>ADE</td>
</tr>
<tr>
<td>Call (immutable parameters)</td>
<td>AE</td>
<td>ABCE</td>
<td>ADE</td>
</tr>
<tr>
<td>Return</td>
<td>F</td>
<td>BCF</td>
<td>BF</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code</th>
<th>Copy operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Copy from client stack to message (or A-stack)</td>
</tr>
<tr>
<td>B</td>
<td>Copy from sender domain to kernel domain</td>
</tr>
<tr>
<td>C</td>
<td>Copy from kernel domain to receiver domain</td>
</tr>
<tr>
<td>D</td>
<td>Copy from sender/kernel space to receiver/kernel domain</td>
</tr>
<tr>
<td>E</td>
<td>Copy from message (or A-stack) into server stack</td>
</tr>
<tr>
<td>F</td>
<td>Copy from message (or A-stack) into client’s results</td>
</tr>
</tbody>
</table>
# Performance of LRPC

## 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>
Performance of LRPC (cont.)

- Locking matters!
The Uncommon Cases

- LRPC still supports cross-machine RPC
  - Detected in first instruction of client stub
- A-stacks are either statically sized or size of ethernet packet
  - Stubs use out-of-band memory for larger arguments
- Domain termination integrated with LRPC
  - Binding objects are revoked
  - Threads returned to client domain (with exception)
  - Linkage records of terminating domain invalidated
- Threads can be recreated in client
  - Addresses server capturing a client’s thread
What Do You Think?