Lightweight RPC

Robert Grimm
New York University
The Three Questions

- What is the problem?
- What is new or different?
- What are the contributions and limitations?
The Structure of Systems

* Monolithic kernels
  * Hard to modify, debug, validate
* Alternative: Fine-grained protection with capabilities
  * Relies on protected procedure calls
  * Is difficult to implement efficiently and to develop for
* Alternative: Small kernels (think Mach)
  * Rely on user-space servers for most functionality
  * But how to communicate between different parts?
* Large-grained protection through machine boundaries
  * Relies on *remote procedure calls* ➔ use with small kernels!
Small kernels use distributed programming models

But common case of communication is not across net

Rather across domains on the same machine

Optimize for the common case

Simple control transfer: use the same thread

Simple data transfer: use shared argument stack

Simple stubs: optimize for local transfer

Design for concurrency: avoid shared data structures
Backgrounder: RPC

By Hank Levy (UW)
Remote Procedure Call

- The basic model for Remote Procedure Call (RPC) described by Birrell and Nelson in 1980
  - Based on work done at Xerox PARC
- Goal was to make RPC look as much like local PC as possible
- Used computer/language support
- There are three 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)
- IDL specifies 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 RPC happen.
RPC Call Structure

- **Client Program**: call `foo(x,y)`
  - Client makes local call to stub proc.

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

- **RPC Runtime**: send msg
  - Runtime sends msg to remote node

- **Server**: proc `foo(a,b)`
  - Server is called by its stub
  - Begin foo...

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

- **Runtime**: msg received
  - Runtime receives msg and calls stub

- **RPC Runtime**: call
  - Runtime sends msg to remote node

**Diagram Notes**
- Client program makes a call to `foo(x,y)`
- Client stub builds a message packet and inserts parameters
- Message is sent to the server
- Server is called by its stub
- Server stub unpacks parameters and makes the call
- Message is received and processed by the runtime
- Runtime sends the message to the remote node
- Client program receives the response
RPC Return Structure

Client program calls `foo(x,y)`

Client continues

Server program calls `foo(a,b)`

Server returns

Client Program:
- `call foo(x,y)`
- `begin foo... end foo`

Server Program:
- `proc foo(a,b)`
- `stub builds result msg with output args`
- `return`

Client Stub:
- `proc foo(a,b)`
- `stub unpacks msg, returns to caller`
- `stub unpacks msg, returns to caller`
- `msg received`

RPC Runtime:
- `msg received`
- `runtime receives msg, calls stub`
- `runtime receives msg, calls stub`
- `runtime responds to original msg`

Server Stub:
- `call foo(x,y)`
- `send msg`
- `send 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 set up a connection

 The *import* and *export* operations are explicit calls in the code
Marshalling is the packing of procedure parameters into a message packet.

The RPC stubs call type-specific procedures to marshall (or unmarshall) 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 to call the server's procedure.

On return, the server stub marshalls return parameters into the return packet; the client stub unmarshalls return parameters and returns to client.
RPC is the most common model now for communication 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 Lightweight RPC
Most RPCs are cross-domain but not cross-machine
- 97% on V, 94.7% on Taos, 99.4% 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 (vs. 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>
LRPC: Combination of protected and remote proc calls

- Execution model: protected procedure call
  - Call to server made through kernel trap
  - Kernel validates caller, records return address, dispatches concrete thread to server
  - Procedure returns through kernel

- Semantics: remote procedure call
  - Servers export interfaces
  - Clients bind interfaces
  - Clients and servers reside in large-grained protection domains

What is the trade-off when compared to regular RPC?
LRPC Binding

- Overall comparable to regular RPC
  - Server exports interface to name server
  - Client imports interface
- Details differ due to high degree of cooperation
  - Procedure descriptor (PD)
    - Entry address, number of simultaneous calls, size of A-stack
    - Pair-wise shared memory for arguments and return values
  - Binding object
    - Capability for accessing the server
  - Linkage record
    - Record of caller's return address — why not on A-stack?
LRPC Calls

- Client stub sets up data on 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 linkage record on per-thread stack — why?
  - Locates execution stack (E-stack) in server's domain
  - Updates thread's 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?
- 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
  - Protection domains (processes) cached on idle processors
    - Processors changed on LRPC (in both directions)
    - Context switch only performed when domain not cached
      - Forced context switches are counted to help with scheduling
  - 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
  * But shared memory allows for asynchronous changes

* Extra copy for immutable parameters
* Constraint checks folded into copy operation

| Table III. Copy Operations for LRPC versus Message-Based RPC |
|------------------|------------------|
| Operation          | LRPC | Message passing | Restricted message passing |
| Call (mutable parameters) | A    | ABCE           | ADE             |
| Call (immutable parameters) | AE   | ABCE           | ADE             |
| Return             | F    | BCF            | BF              |

<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>
Why does RPC level off at 2 processors?
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 (when args are variably sized)
  - Stubs use out-of-band memory for larger arguments
- Domain termination integrated with LRPC
  - Binding objects are revoked
  - Threads returns to client domain (raising exception)
  - Linkage records of terminating domain invalidated
- Threads can be recreated in client
  - Addresses server capturing a client's thread
What Do You Think?