To infinity, and beyond!

Kiyan Ahmadizadeh
CS 614 - Fall 2007
LRPC - Motivation

- Small-kernel operating systems used RPC as the method for interacting with OS servers.
- Independent threads, exchanging (large?) messages.
- Great for protection, bad for performance.
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>
Where’s the problem?

- RPC implements cross-domain calls using cross-machine facilities.
  - Stub, buffer, scheduling, context switch, and dispatch overheads.
- This overhead on every RPC call diminishes performance, encouraging developers to sacrifice safety for efficiency.
- Solution: optimize for the common case.
What’s the common case?

Most RPCs are cross-domain and have small arguments.
LRPC Binding

Client

Kernel

Server's Clerk

Kernel Memory

Shared Memory
LRPC Binding

Kernel

Server’s Clerk

PDL
PD:
Entry Addr
Sim Call Limit
A-Stack Size
PD...
...

Shared Memory

Kernel Memory

Client
LRPC Binding

Client

Kernel

PDL
PD:
Entry Addr
Sim Call Limit
A-Stack Size
PD...
...

Server’s Clerk

Shared Memory

Kernel Memory

A-Stack
A-Stack
A-Stack
A-Stack
A-Stack
LRPC Binding

Kernel

Server’s Clerk

PDL
PD:
Entry Addr
Sim Call Limit
A-Stack Size
PD...
...

Client

Kernel Memory

Shared Memory

Linkage Record

A-Stack

A-Stack

A-Stack

A-Stack
LRPC Binding

- Kernel
  - PDL
  - PD:
    - Entry Addr
    - Sim Call Limit
  - Binding Object
    - PD
    - A-Stack List...
  - A-Stack Size

- Server's Clerk

- Shared Memory
  - A-Stack
  - A-Stack

- Client

- Kernel Memory
  - Linkage Record
LRPC Binding

- Client
- Kernel
- Server’s Clerk

Kernel Memory:
- Linkage Record
- A-Stack List...
- Binding Object

Shared Memory:
- A-Stack

Kernel:
- PDL
- PD:
  - Entry Addr
  - Sim Call Limit
  - A-Stack Size
- PD...
- ...

Linkage Record
A-Stack List...
Binding Object
Client calls client stub with procedure arguments, A-Stack List, and Binding Object. If call is cross-machine, stub takes traditional RPC path.

Otherwise, client stub finds next A-Stack for this procedure and pushes procedure’s arguments.

A-Stack, Binding Object, and Procedure Identifier addresses placed in registers.

Kernel trap.
LRPC Calls - The Kernel

- Kernel executes in client’s context.
- Verifies binding object. Finds the linkage record linked with the A-Stack.
- Place caller’s return address and stack pointer in linkage record. Push linkage onto TCB.
Kernel finds new \textit{E-Stack} in server’s domain. The thread’s SP is updated to point to this stack.

- Processor’s virtual memory registered loaded with the server’s domain.
- Control transferred to server stub’s entry address from process descriptor.
- Server puts results on A-Stack, traps to kernel. Kernel uses linkage record to return to client.
Major Advantage: Copy Reduction

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>

**Code**

- A: Copy from client stack to message (or A-stack)
- B: Copy from sender domain to kernel domain
- C: Copy from kernel domain to receiver domain
- D: Copy from sender/kernel space to receiver/kernel domain
- E: Copy from message (or A-stack) into server stack
- F: Copy from message (or A-stack) into client’s results
Issues / Optimizations

- What about large arguments of variable size? What if A-Stack size cannot be determined in advance?
- Stub generator generates stubs in assembly language. Generator must be ported from machine to machine.
- Multiprocessor systems can use idle processors to eliminate context switch cost.
## 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>

Averaged over 100,000 runs on the C-VAX Firefly
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>

A 307 microsecond improvement over Taos.
Less contention over shared resources increases throughput.
For small messages in a LAN, processing overhead dominates network latency.

New applications demand high bandwidth and low latencies for small messages.

Remote file systems, RPC, object-oriented technologies, distributed systems, etc.
Is this possible on traditional UNIX?

- Protocol stack is in the kernel:
  - Increased overhead when sending messages (especially from copies)
  - New protocols have to be built on top of protocols kernel provides. Bad for efficiency and optimizing buffer management.
U-Net’s Solution

- Move the *entire* protocol stack into user space. Applications access the network interface directly.
- Network must be multiplexed among processes.
- Processes cannot interfere with each other.
• Processes wishing to use the network create an endpoint, and associate a communication segment, send queue, receive queue, and free queue with it.
Sending a message

recv_queue
free_queue
communication_segment
send_queue

Endpoint

Process
Sending a message

recv queue
free queue
send queue

communication segment

Message Data

Endpoint

Process
Sending a message

Process -> Endpoint

recv queue
free queue
communication segment
send queue
Message Data
Sending a message

Process

Endpoint

recv queue
free queue

communication segment

Message Data

send queue

NI
Sending a message

- recv queue
- free queue
- send queue

communication segment

Message Data

Endpoint

Process
Sending a message

recv queue
free queue
communication segment
send queue

Endpoint

Process
Sending a message

recv queue  free queue
communication segment
send queue

Endpoint

Process
Receiving a message

- Much the same. U-Net demultiplexes messages, transferring data to the correct communication segment.

- Space in segment found using free queue. Message descriptor placed in receive queue.

- Process can poll the receive queue, block, or U-Net can perform upcall on two events.

- Receive queue non-empty and almost full.
Multiplexing

- Process calls OS to create communication channel based on destination. Uses this in sends and receives.

- On send, OS maps communication channel to a message tag (such as ATM virtual channel identifier). This tag is placed on message.

- Incoming message’s tag mapped to channel identifier: message delivered to endpoint indicated by identifier.
Base-level U-Net

- Communication segments are pinned to physical memory so network interface can access them.
- Buffers and segments can be scarce resources. Kernel-emulated U-Net endpoints can be used: application endpoints are multiplexed into a single real endpoint.
- Represents zero-copy, which is really one copy (from process address space to communication segment)
Let communication segment span entire address space! Network interface can transfer data directly into data structures (true zero-copy).

But then NI needs to understand virtual memory, and needs enough I/O bus address lines to reach all of physical memory.
Two Implementations

- Implemented using SPARCstations and two Fore Systems ATM interfaces.
- SBA-100 implemented with loadable device driver and user-level library.
- SBA-200 firmware rewritten to implement U-Net directly. The interface’s processor and DMA capability make this possible.
Performance - Round Trip Times

Small round-trip times for messages under 1-cell in size. This case is optimized in the firmware.
Figure 4: U-Net bandwidth as a function of message size. The AAL-5 limit curve shows the maximum bandwidth achievable. The Raw U-Net curve demonstrates the bandwidth performance of the raw U-Net protocol. The UAM store/get curve illustrates the performance when using the UAM protocol for store and get operations. The AAL5 limit curve suggests that the existing buffer size of 4160 bytes affects the performance at 4164 bytes, leading to a drop in bandwidth.
Split-C Benchmarks

Graph normalized to execution time of CM-5.
Split-C Benchmarks

Graph normalized to execution time of CM-5.
Saw-tooth effect caused by Fore’s buffering restrictions.

U-Net buffers are in user-space, relaxing size restriction on socket receive buffer.
Figure 8: TCP bandwidth as a function of data generation by the application.
U-Net and Fore Latencies

![Graph showing TCP bandwidth as a function of data generation by the system. The graph compares U-Net, Fore TCP, and Fore UDP latencies.](image)
Some things to consider...

- Is this really implemented on “off-the-shelf” hardware?
  - Firmware customizations.
- Memory requirements for end-points. Pages getting pinned into memory.
- Virtual Interface Architecture (VIA) heavily influenced by U-Net.
Summary

- LRPC and U-Net seek to speed up applications by optimizing the common case.
- Both cases eliminated unneeded processing overheads, boosting efficiency.