User–Level Interprocess Communication for Shared Memory Multiprocessors

Brian Bershad, Thomas Anderson, Edward Lazowska, and Henry Levy
Presented by: Byron Marohn
Published: 1991
Outline

- Background
- Remote procedure call (RPC)
  - Fundamentals
  - Previous Implementations
- User level remote procedure call (URPC)
  - Implementation
  - Assumptions
  - Performance
- Conclusion
Background

- Threads – Procedure oriented parallelism
  - An Introduction to Programming with Threads (1989)
- Events – Message passing
  - Why Threads Are a Bad Idea (for most purposes) (1995)
- Interprocess Communication
Remote Procedure Call (RPC)

- Introduced in this class by:
- Defined as:
  Synchronous language-level transfer of control between programs in disjoint address spaces whose primary communication medium is a narrow channel
Previous RPC Implementations

- Until now, need OS kernel to copy data between distinct address spaces
- Calling thread blocks, kernel reschedules CPU
- Requires a trap, and multiple context switches

Questions:
- How can we reduce context switching?
- What if we could transfer data directly?
Reduced Context Switching

- Transfer control to process already running in the desired address space
  - Lightweight Remote Procedure Call (1990)
- User level threads
Data transfer

- Message passing systems
  - SEDA: An Architecture for Well-Conditioned, Scalable Internet Services (2001)

- Shared memory?
User Level Remote Procedure Call (URPC)

- Use shared memory to pass messages between address spaces, bypassing kernel
- Combine cross-address space communication with lightweight user-level threads, so blocking is inexpensive
- Send/receive messages between already-running server/client processes, reducing the need for context switches
Set up a pair of bidirectional queues between client and server, with locks at each end.

- Client → server, server → client queues
- To send a message, acquire the lock, copy message into memory, release lock.
- To receive a message, acquire lock and remove message from queue.
When a call is made, user–level scheduler is invoked
Scheduler queues message, and immediately schedules another user level thread to run.
Requires only an inexpensive user–level reallocation (save/restore execution state, and cache / TLB penalties)
  ◦ 15 µsecs vs 55 µsecs for processor reallocation
URPC also supports transferring an execution context/real processor between client and server address spaces

Kernel does the processor reallocation

Load balancing technique, either side can detect if other side is busy or not by checking the number of items pending in the queue.

Problem: What if processor is not given back?
URPC Structure

Client

- Application
  - Stubs
  - URPC
  - FastThreads

Server

- Application
  - Stubs
  - URPC
  - FastThreads

Message Channel

Reallocate Processor

Kernel
Example

- 2 CPUs
- 2 Client Threads
- Editor (client)
- WinMgr (server)
- FCMgr (server)
  - FC = File Cache

---

```
T1 Call
Send WinMgr
Recv WinMgr
Context Switch
T2 Call
Send FCMgr
Recv FCMgr
Context Switch
T1 Call
Send FCMgr
Recv FCMgr
Reallocate Processor to FCMgr
Context Switch
Terminate T2
Context Switch
Terminate T1
```

**Editor**

<table>
<thead>
<tr>
<th>WinMgr</th>
<th>FCMgr</th>
</tr>
</thead>
<tbody>
<tr>
<td>(scanning channels)</td>
<td>(has no processors)</td>
</tr>
</tbody>
</table>

Fig. 1. URPC timeline.
URPC Assumptions

- “Optimistic reallocation policy”
- Client has other work to do (runnable threads), and a further delay for an already blocked call will not impact performance much.
- Server has capacity (or will soon) to process messages/RPC calls.

- Problem: What about when latency is important?
URPC Performance

- All subsequent performance data gathered on experimental device called “Firefly”, with 6 processors and a dedicated I/O microprocessor.
- Fastthreads as the user-level threading implementation, Taos is the underlying operating system kernel.
# URPC Performance

## Table I. Comparative Performance of Thread Management Operations

<table>
<thead>
<tr>
<th>Test</th>
<th>URPC FastThreads ($\mu$secs)</th>
<th>Taos Threads ($\mu$secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROCEDURE CALL</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>FORK</td>
<td>43</td>
<td>1192</td>
</tr>
<tr>
<td>FORK;JOIN</td>
<td>102</td>
<td>1574</td>
</tr>
<tr>
<td>YIELD</td>
<td>37</td>
<td>57</td>
</tr>
<tr>
<td>ACQUIRE,RELEASE</td>
<td>27</td>
<td>27</td>
</tr>
<tr>
<td>PINGPONG</td>
<td>53</td>
<td>271</td>
</tr>
</tbody>
</table>

## Table II. Component Breakdown of a URPC

<table>
<thead>
<tr>
<th>Component</th>
<th>Client ($\mu$secs)</th>
<th>Server ($\mu$secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>send</td>
<td>18</td>
<td>13</td>
</tr>
<tr>
<td>poll</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>receive</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>dispatch</td>
<td>20</td>
<td>25</td>
</tr>
<tr>
<td>Total</td>
<td>54</td>
<td>53</td>
</tr>
</tbody>
</table>
URPC Latency

C = Client Processors
S = Server Processors

Call Latency (μsecs)

Number of Client Threads (T)
URPC Throughput

C = Client Processors
S = Server Processors

Graph showing throughput of URPC as a function of the number of client threads (T) with different combinations of client and server processors.
Desire to isolate different functionality within domain boundaries.
Processes need to communicate across these domains -> Remote Procedure Calls

Unfortunately:
Switching physical processor contexts is slow

To remedy this:
For communication between programs on shared memory multiprocessor, can use user-space memory to pass data between already running processes
Conclusion

General theme: Reduce kernel context switching to improve performance

Performance:
- Throughput better than other implementations
- Only on shared memory multiprocessor
- Latency increases with load