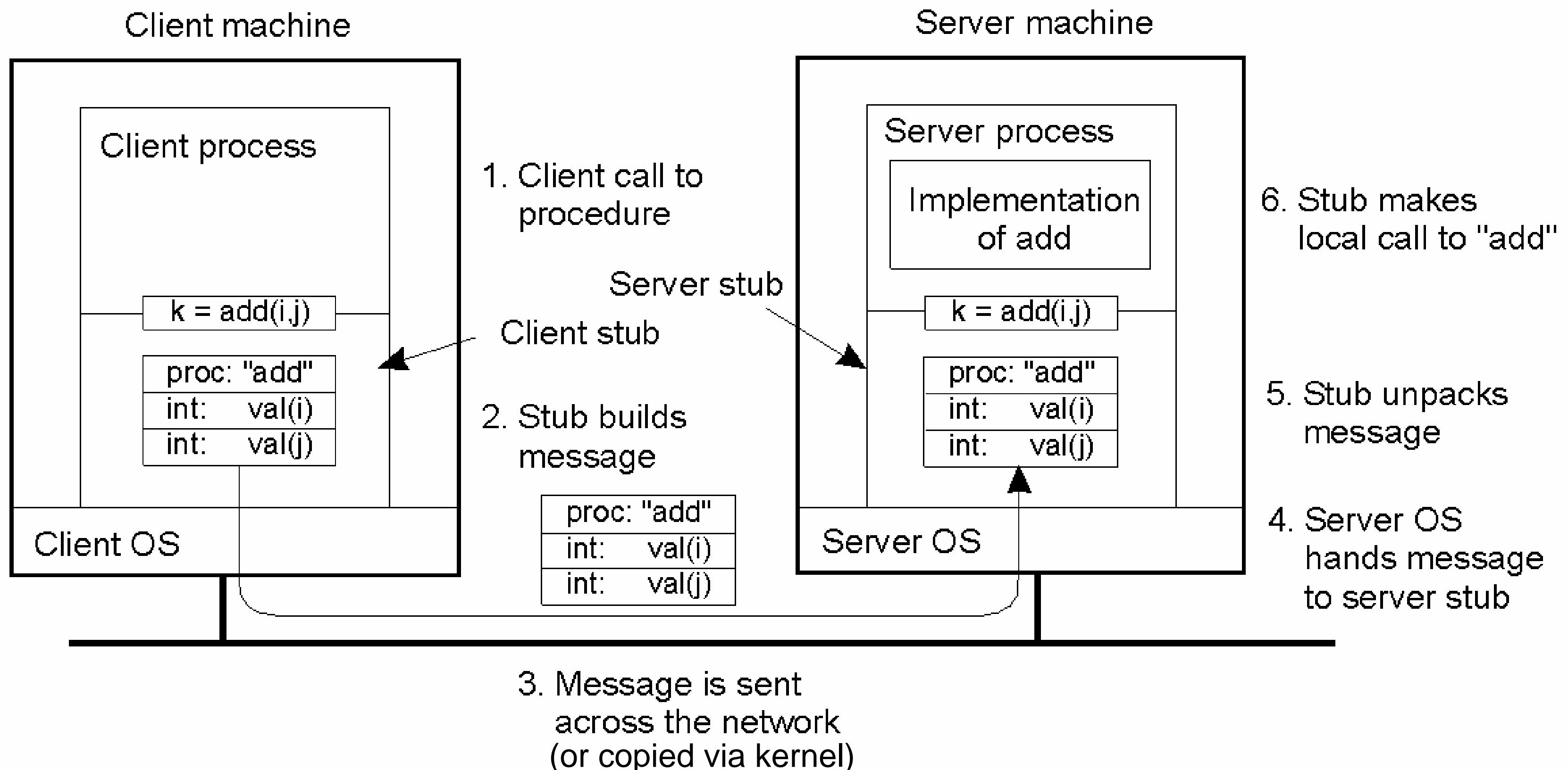
Lightweight RPC

Brian Bershad, Thomas Anderson, Ed Lazowska, Henry Levy, UW

Remote Procedure Calls

- Structure applications in separate domains, separate locations… yet maintain familiar procedure call interface
- Promotes failure isolation, modularity, maintenance, location transparency (to a degee)…



Reference paper

- More details on RPC implementation
- Some special optimizations
  - Reuse and recycling of packets
  - Shared RPC packets buffers
- More general, cross-machine implementation
- This paper focus on optimizing RPC for cross-domain calls

Important to minimize overheads

- Sources of overheads
  - Stub overhead
    - Must make it general, but what if there are no parameters?
  - Message buffer overhead
    - 4 copies on call for cross-domain call on single machine
  - Access validation
    - Is sender allowed to perform op? Are results returned to correct sender?
  - Message transfer
    - Some queue, flow control mechanism…
  - Scheduling
    - 1 abstract thread vs. 2 concrete threads -> scheduler invoked
  - Context switch
    - May be necessary to perform 2
  - Dispatch
    - “listener thread” vs. “worker thread” in server domain

Some common techniques

- Alloc messages in regions mapped to kernel and user
- Use shared memory region (if shm is an option) for all RPC messages (safety?)
  - Or pairwise shared memory (this is what paper has)
- Pass few arguments in registers (we already saw this in exokernel, tornado…)
  - Use combination of registers, copy, and mmap to pass data
- Assume certain trust level and simplify/ignore access control
- Handoff scheduling – if ‘partner’ thread known, bypass scheduler and directly context switch
- Many others…

Objectives of Lightweight RPC

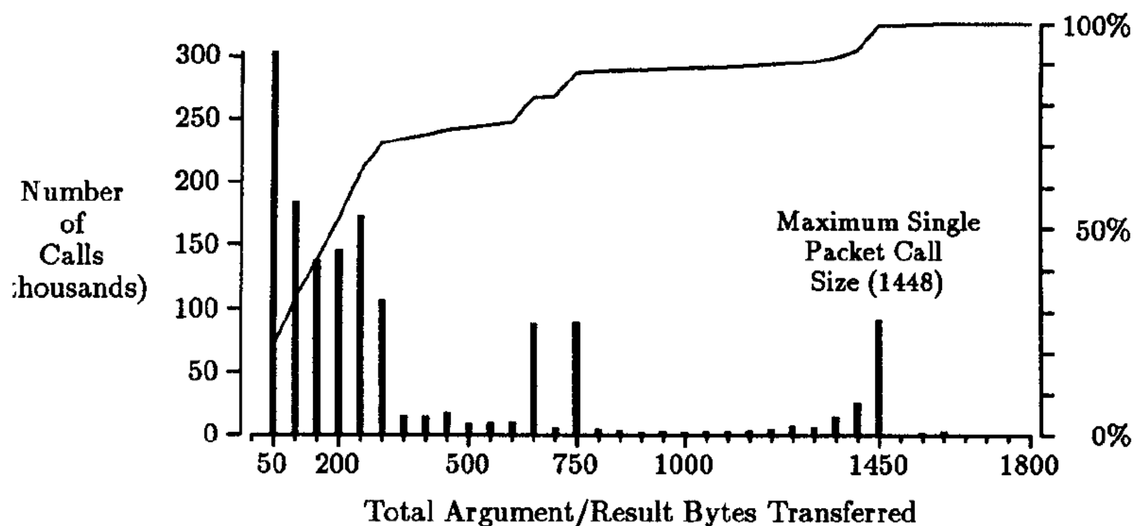
- Keep RPC semantics
- Lightweight – as close to theoretical minimum
- Specialize
  - for cross-domain calls (single machine, shared memory)
  - for calls with few arguments
  - Why?

RPC common case

- V system – OS services in separate servers
- Taos – microkernel + large server for all other services
- UNIX + NFS – RPC for files, but potentially monolithic UNIX can be decomposed, so count system calls

Table I. Frequency of Remote Activity

Operating system	Percentage of operations that cross machine boundaries
V	3.0
Taos	5.3
Sun UNIX+NFS	0.6



What is (was) cost of NULL call

- Should be procedure call + 2 kernel traps + 2 ctx switched

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

System	Processor	Null (theoretical minimum)	Null (actual)	Overhead
Accent	PERQ	444	2,300	1,856
Taos	Firefly C-VAX	109	464	355
Mach	C-VAX	90	754	664
V	68020	170	730	560
Amoeba	68020	170	800	630
DASH	68020	170	1,590	1,420

Lightweight RPC design

- Summary
  - Client makes procedure call and traps into kernel
  - Kernel validates caller and creates a linkage
  - Kernel dispatches client thread into server address space
  - Client and server share argument stack, but each has its own execution stack
- Leverages Modula2+ language environment
  - Arguments on stack, share stack
  - Separate argument and execution stack

Binding

- Server exports list of procedure descriptors for each procedure in its interface
- Each PD – entry address, stack size, num of concurrent calls
- On client bind
  - for each PD kernel creates a number of shared A-stacks
  - For each A-stack kernel creates a linkage record at location which can be easily determined
  - Each A-stack has private lock
- Client receives an unforgeable Binding Object and list of A-stacks for each of the procedures in the server interface

Calling

- Stub takes an A-stack, pushes args on it, puts A-stack addr, Binding Obj and proc ID into registers and traps
- Kernel verifies info in registers, finds and fills out linkage record and places linkage record on linkage record stack for client thread
  - On return, TCB points to linkage record, so can return to caller without extra checks
- Get an E-stack from server domain
  - E-stacks private to server, but may be associated with A-stack (given availability)
- Set thread registers to server domain/E-stack and upcall at PD address
- Parameter passing
  - On stack
  - If immutable then copy made
  - If by reference client copies on stack, server creates local reference to stack location
  - For large values – mmap
  - Mark arguments which should not be copied
- Leverage type-safe language for value correctness checks

Argument handling

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

Code	*Copy operation*
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

Stubs

- Fast path code in assembly
  - Efficient, but non-portable
- Complex marshaling code or binding / exception handling /etc. – rpcgen generates Modula2+ code
- Choice of fast vs. slow code made at compile time
- Direct dispatch to server stub, E-stacks prepped appropriately

Multiprocessor optimization

- Avoid context switching
- Cache domains on idle processors
  - On call (and return) swap calling and idling thread
  - Keep heuristics to determine which server domains to cache on idle processors
  - Has benefits even for tagged TLBs
  - What's the difference between caching threads vs. caching domains?

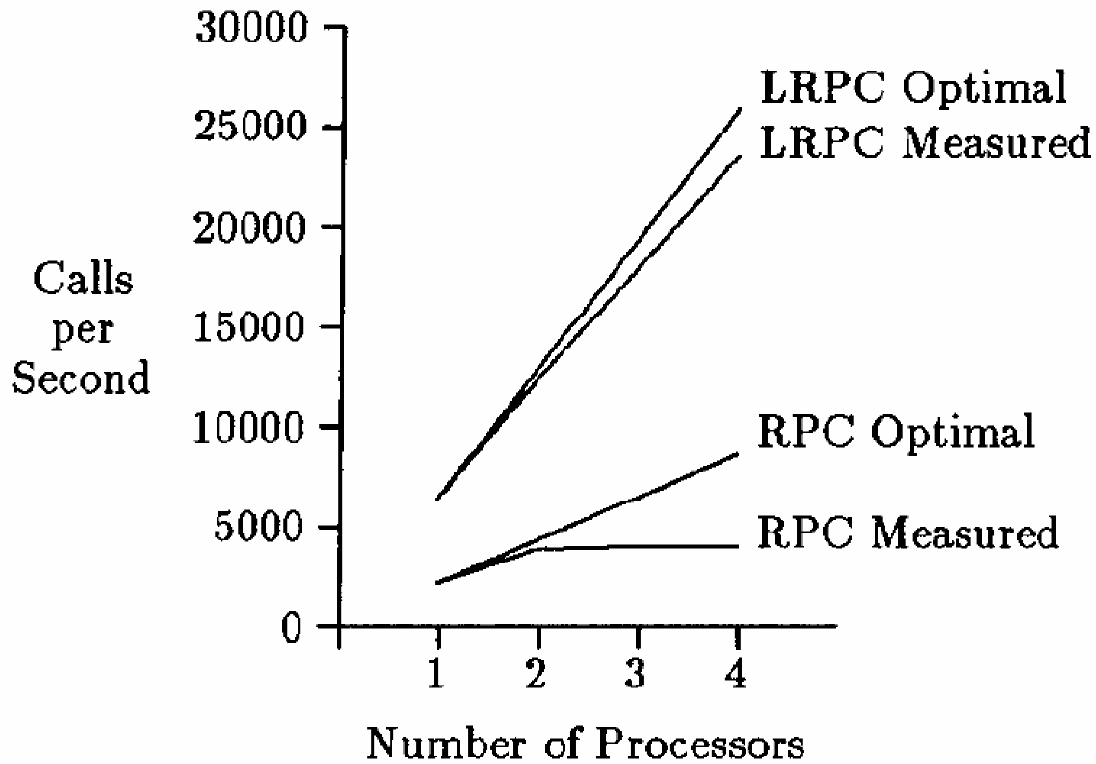
Table IV. LRPC Performance of Four Tests (in microseconds)

Test	Description	LRPC/MP	LRPC	Taos
Null	The Null cross-domain call	125	157	464
Add	A procedure taking two 4-byte arguments and returning one 4-byte argument	130	164	480
BigIn	A procedure taking one 200-byte argument	173	192	539
BigInOut	A procedure taking and returning one 200-byte argument	219	227	636

Table V. Breakdown of Time (in microseconds) for Single-Processor Null LRPC

Operation	Minimum	LRPC overhead
Modula2+ procedure call	7	—
Two kernel traps	36	—
Two context switches	66	—
Stubs	—	21
Kernel transfer	—	27
Total	109	48

Benefits of fine grain locking



Other issues

- Binding object has a bit to denote true remote RPC
- Uncommon cases of args don't fit on A-stack, or need more A-stacks, handled at a higher cost, but don't occur frequently
- Domain termination
  - Revoke Binding Objects
  - Threads within domain running LRPC restarted in client domain at call-fail exception
  - Invalidate linkage records -> will raise call-fail exception
  - If thread 'captured' can create new one with call-aborted exception
  - Issue in LRPC because caller thread is thread executing in other domain