LRPC

1. Reading group presentation
    a. Anubhavnidhi
    b. Tithy
    c. VINOTHKUMAR
2. Questions from reviews
    a. What about security?
        i. Answer: locally not need encryption etc.; kernel guarantees who client is
    b. How evaluate security?
        i. Good question!
    c. Why copy arguments at all with shared memory?
    d. What is need for RPC if most have small arguments?
        i. RPC is much more than marshaling
3. Why use RPC for structuring a system
    a. Easy to use compared to alternatives – compiler handles most of details
    b. Easy to build a protected subsystem
    c. Allows moving components out of kernel if fast enough
        i. E.g. singularity, Minix, Mach
    d. Compared to Opal Portals:
        i. Portals are a protection mechanism indicating who can call what (not mentioned in LRPC),
        ii. Very similar mechanisms (same advisor)
    e. More reliable
    f. Easier to extend
    g. Faster RPC makes it possible to structure systems differently; brings up issue of evaluating new capabilities
    h. QUESTION: Are these good reasons?
        i. Do people using RPC intend to later migrate it off machine, so it is only local temporarily?
    i. Examples:
        i. Cross-frame communication in a browser
        ii. Local DNS resolver
        iii. Windows service controller, services
        iv. Programmability layer on top of unix domain sockets
    j. QUESTION: When is RPC not a good mechanism for this?
        i. If want to integrate communication into an event loop (e.g. interactive applications), may want to poll for messages on a socket/pipe (select)
4. Overview

    a. General approach:
        i. Analyze a system
        ii. Find an untapped opportunity; some common behavior that can be optimized
            1. E.g. small arguments
            2. Fixed size arguments
            3. Unstructed arguments (e.g. buffers vs. types needing marshalling)
            4. Unnecessary optimizations (e.g. copying data)
        iii. Measure overhead that you could remove; best case performance
        iv. Build an optimized version that takes advantage of the opportunity
        v. Go on to fame and fortune
        vi. QUESTION: other examples?
            1. Flash memory
            2. Web servers: create “sendfile” api, changing how networking works
    b. Opportunities
        i. RPC used for structuring systems:
            1. Client / server (e.g. Windows services, name server)
            2. NFS file server – used for sending requests to server
            3. Common implementation:
                a. Separate out stubs from communication
                b. Build on existing protocol: pipes, tcp/ip/udp
        ii. QUESTION: they don’t profile RPC to show where it slow, and optimize those parts. Is this important
            1.
        iii. Common case is not remote / large arguments
            1. Common case is local calls when used in systems with micro-kernels (1-6%)
            2. Common case is small, fixed-size arguments
                a. 60% were < 32 bytes
                b. 80% of arguments fixed size at compile time
                c. 2/3 procedures have fixed-size arguments
    c. QUESTIONS: how legitimate is this study? Look both at existing microkernel systems + future use (e.g. system calls)
        i. Note: look at system without RPC where RPC could have been used (V messages, Unix system calls)
    d. QUESTION: why not look at socket applications? Could look at domain sockets (local sockets). Internet applications relying on RFCs aren’t going to convert to RPC
5. How LRPC works
    a. **Big approach:**
        i. **Look at minimum time of things that have to happen**
            1. One procedure call

      2. Trap to kernel
      3. Process context switch (change address space)
      4. Return from Trap in server
      5. Trap to return
      6. Process context switch to client
      7. Return from trap
   ii. **Everything else is overhead!**
b. Approach
   i. Do everything in advance
      1. e.g. allocating stacks
      2. e.g. setting up dispatch (no dynamic dispatch in server)
   ii. Remove unnecessary copies
      1. Memory copies huge cause of performance problems
   iii. Break operation into constituent parts, optimize or remove each
      1. Stub overhead – convert procedure call into message passing; marshall arguments, wait for result
      2. Message buffer overhead – allocating buffers, copying data between domains
      3. Access validation – validate sender's identity on call and return
      4. Message transfer – enqueue/dequeue message, flow control
      5. Scheduling – put client to sleep, wake up server, access system's run queue
      6. Context switch – change page tables
      7. Dispatch – server must interpret message, parse arguments, call destination routine
   iv.
c. Doing things in advance
   i. Bind
      1. **Clerk** thread in server listens to binding calls and accepts them
      2. Create procedure description list with each exported procedure (address, size of args)
      3. Allocate shared a-stacks and corresponding linkage records (for caller's return address) for each procedure. **JUST FOR CLIENT**
         a. QUESTION: Why allocate stacks for all procedures?
            i. ANSWER: want contiguity for easy range checking
         b. QUESTION: how many should be allocated?
            i. ANSWER: # of concurrent clients.
            ii. Why default to 5?

            iii. ANSWER: Had a 6 processor machine
        4. Return **binding object** to client runtime to identify binding
        5. KEY POINT: binding object contains server function address – no need for dispatch in server
            a. Note: is like a file object
    ii. Call
        1. QUESTION: How do they know if call is local or remote?
            a. A: at bind time, cache a bit of information
        2. QUESTION: What is the cost?
            a. A: can be one test & procedure call at the beginning of every stub; already included in local stub cost as shown. Overhead on real RPC would be almost zero given cost of network.

d. Copy avoidance
    i. Client stub grabs A-stack off queue (managed in stub)
    ii. Push arguments on A-stack
        1. No separate copy to kernel; copies directly to server
    iii. Pass a-stack, binding object, procedure identifier in registers to kernel
    iv. Kernel
        1. Verify binding, procedure identifier
        2. Locates procedure description
        3. Verify A-stack & locate linkage for A-stack
        4. Verify ownership of A-stack
        5. Record caller's return address in linkage (means return address stack)
        6. Push linkage onto thread (so can nest calls)
        7. Find execution stack (e-stack) for server to execute (from pool)
            a. NOTE: e-stack is like a normal stack, private to server for local variables
            b. QUESTION: When? Can do on demand (avoid wasting stacks), or in advance.
                i. LRPC: on demand first time, then remember association
        8. Update thread to point at E-stack
        9. Change processor address space
        10. Call into server stub at address in PD
            a. Writes return value back to A-stack
            b. Kernel knows what to do when call returns
    v. Notes:
        1. Avoids runtime: client and server interact with kernel directly

2. One copy – from client to A-stack
3. Can use separate argument stack because language supports in, C would need to copy arguments to E stack
    a. What else could you do? Put a-stack/e-stack on attached pages
4. By-reference objects are copied to A stack by client stub
    a. PRINCIPLE: client does copying work
    b. PRINCIPLE: client stub does work, kernel verifies (e.g. choose A-stack)
    c. QUESTION: What are alternatives to having client stub do copying?
    d. Server stub must create pointer to A-stack data on E stack
5. What about thread-local storage in server? Or thread-init routines for DLLs?

vi. QUESTION: What about writing with shared memory?
    1. A: no isolation

vii. NOTE: server does not create threads; it just creates stacks and reuses client threads
    1. Needs bookkeeping: if RPC thread makes a system call, must create/access objects of server not client process
    2.

e. Writing stubs
    i. Generated in ASM from source
        1. First instruction determines local/remote binding for uncommon case
    ii. Modula 2 code generated automatically for complex pointer-based data structures

f. Optimizations – multiprocessor
    i. Cache processors running in a protection domain
        1. Page table already pointing correctly, TLB already has right contents
        2. Like pipe between two processes on same core vs different cores: cache coherence vs TLB misses
    ii. Do handoff scheduling on call
        1. Client thread migrates to target processor in servers domain
        2. Servers thread takes over client processor
            a. QUESTION:why?
                i. Avoids needing to queue the server thread
        3. Ensure conservation of processors; doesn't impact kernel processor scheduling *much*

        iii.
    g. Copying Safety
        i. Normal RPC makes copy of arguments
            1. Many times – up to 4 times
        ii. QUESTION: What is benefit?
            1. Ensures COW semantics; client changes can't corrupt server
        iii. LRPC uses shared stacks accessible to both processes
            1. Client can overwrite A-stack while server access it
        iv. Solution:
            1. Server can copy/verify data **only if needed**
            2. Destination address for return values private to client; no benefit in having kernel, not server, write to it
                a. Q: what would effect of doing it wrong be?
                b. A: returning incorrect value
            3. Not needed for opaque (e.g. buffers) parameters
            4. More efficient to have stubs to copying than kernel
                a. Server can integrate validity checks with copying
            5. Adds at most one extra copy (on top of initial 1)
            6. COMMENT: More like a system call, where kernel validates parameters
            7. ISSUE: Complicates server; not transparent
        v. QUESTION: What does this mean for safety/security?
    h. Reliability
        i. What if server crashes?
            1. Thread stopped, returns to previous caller
        ii. Client crashes?
            1. On return, skip failed processes or terminate thread if is ultimate client
        iii. What do you do if a server thread hangs?
            1. Question: what is the key problem?
                a. A: client thread has been taken over for the server, can't just timeout because server is actively using it
            2. Solution: duplicate client thread state into a new thread
                a. Can also do this based on timeout
        iv. What if a client process crashes?
            1. Mark linkage as dead, so when server thread returns, it just terminates
6. Evaluation:
    a. Comments: good evaluation explains why the performance is better, doesn't just show it is better
    b. Example: was on a PC meeting, one paper showed a 100x speedup. But, didn't explain it. PC felt that they didn't understand

the system, because the code they explained didn’t justify a 100x increase. Result: paper dinged

  - i. In this paper: didn’t which pieces of RPC were bad
  - ii. Showing the minimum possible gets around this from the other direction

7. Commentary
    - a. Limitations:
        - i. Assumes no per-thread application state
        - ii. Relies on argument stack pointer to avoid copying / changing protection on execution stack
    - b. Idea used in Windows NT
        - i. Dave Cutler drove from MS over to UWash for a meeting
        - ii. Windows version different
            1. No shared stacks
            2. Pre-allocated shared memory if large objects needed
            3. Handoff scheduling for low latency
            4. Still have to copy messages many times
                - a. Into user-mode message
                - b. Directly from client buffer to server buffer
                - c. Onto server stack
            5. Quick LPC:
                - a. Dedicated server thread
                - b. Dedicated shared memory with server thread
                - c. Event pair for signaling message arriving / reply arriving
    - c. How important is fast IPC?
        - i. Systems are never fast enough
        - ii. If code called frequently, always the temptation to move code into the kernel
8. Performance Techniques
    - a. Early binding
    - b. Pre-allocation, pre-association
    - c. Migrating threads
    - d. Make kernel/server to verification only, not the work.
        - i. E.g. choosing an A stack