Performance of Firefly RPC

- Contribution: “account precisely for all measured latency” in an RPC implementation

Background

- A frequently heard question: how could it possibly be so slow?
  - substitute “it” with your favorite target: your program, Java, the file server, …
- Answering the question is one of the jobs of system researchers
  - back of envelop calculations
    - What is the maximum RPC throughput you can get on a 10Mb/s Ethernet?
  - Profiling tools: gprof, top, vmstat,netstat, kernel

Why RPC Performance?

- RPC is the primary communication mechanism in LAN distributed systems
- If it is too slow, people won’t use it
- Focus: inter-machine RPC
  - Same machine RPC is shown to be 20-60 times the cost of a procedural call

System Configuration

- Firefly multiprocessor machine: 5 1MIPS CPU, 16MB memory
- 1 processor connects to Qbus I/O bus, which then connects to a 10Mb/s Ethernet via DEQNA device controller
- device controller has access to ~16Mb/s of Qbus bandwidth
- True SMP-capable kernel

Review of RPC Implementation

- Caller stub
  - allocates and prepares a call packet
  - marshal interface and procedure definition and arguments
  - call transport mechanism to send the packet, block
  - ...
- Server stub
  - get the packet from the transport mechanism
  - unmarshal arguments and call the procedure
  - …

Measuring RPC Performance

- Designing the tests
  - max packet size on the Ethernet is 1514bytes
  - Procedure: Null()
  - Procedure: MaxResult(VAR OUT buf: ARRAY of CHAR)
  - Procedure: MaxArg(VAR IN buf: ARRAY of CHAR)

Latency and Throughput

- Time for 10K RPC calls
  - Should the results be affected by the number of threads?
- Throughput

Measuring Marshalling Time

- Marshalling code generated by Modula-2
- How to measure it
  - Why measure on same-machine RPC?
- Types of arguments to measure
  - Integers
  - Fixed-length array
  - variable length array
  - Text.T string argument

Steps in RPC

- Fast path, via UDP/IP:
  - no thread allocation overhead, no retransmission/ack overhead, etc.
- Caller stub
  - Obtain a packet buffer *with partially filled-in header*
  - Copy the arguments to the packet
  - Transmit the packet and wait for return packet
  - copy the result packet data to caller's variables
  - return the result packet to the free pool

Steps in RPC

- Server stub: once receive a call packet
  - process arguments depending on the type: either copy to local stack variable, or copy into storage, or left in the call packet
  - Call server procedure
  - copy results in the saved call packet, which then becomes the result packet

Steps in RPC

- Transport mechanism
  - Fill in the RPC header, the UDP, IP, Ethernet headers
  - UDP checksum
  - interrupt CPU0 to run Ethernet driver, CPU0 prob Ethernet controller into action
  - Ethernet controller reads the packet over Qbus and transmits it to server
  - server Ethernet controller writes the packet over Qbus and issues packet arrival interrupt

Steps in RPC

- Server's Ethernet interrupt routine validates headers, verify UDP checksums, and attempts to hand the packet directly to a waiting server thread
- Server stub inspects the RPC header and calls the interface stub, which then calls the procedure

Optimizations

- no library procedure for marshalling common argument types
- No table lookup for calling Starter, Ender, Transporter
- Directly awaken the thread in the Ethernet interrupt routine (1 wakeups)
- Buffer management tricks: packet buffers reside in shared memory

Breaking Down the Latency

- Send+Receive operations
- User space procedures
- Total

Effects of Various Optimizations

- Write it using assembly
- Different network controllers
- Faster networks
- Faster CPUs
- Omit UDP checksum
- Redesign RPC protocol
- Omitting the IP/UDP headers
- Busy wait
- Write RPC run-time routine in assembly