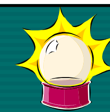
Implementing Remote Procedure Calls

Birrell and Nelson

Presentation By:
Khawaja Shams

The Idea



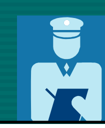
- Extend procedure calls to enable transfer of control & data across a network
- High Level overview:
  - Pass the parameters across the network to invoke procedure remotely
  - Pass the result back to caller

Why Procedures

- Clean and Simple Semantics
- Efficiency through simplicity
- Generality

Issues to Consider

- Precise semantics of calls
- Semantics of address containing arguments w/o shared address space
- Integration with existing/future programming languages
- Binding
- Suitable protocols for data/control transfer between caller/callee
- Data integrity and security



Components

- Cedar
- Dorados
- Ethernet
  - 3 Mbps and 10Mbps
- Assumes most communication will happen on local ethernet

Aims

- Make distributed computation easy
- Provide communication with as much ease as local procedure calls
  - No added complexities to the programmer (timeout, etc)
- Encourage people to build more distributed systems

Secondary Aims

- Make RPC communication efficient
  - Within a factor of 5 beyond necessary transmission times of a network
- Secure Communications
  - Few distributed systems attempted this in the past
  - No RPC systems had attempted this

Decisions

- Procedure calls vs. Message Passing
- Parallel Paradigm
- Shared Address space between computers
  - Hard to integrate with the programming language
  - Could exploit address mapping mechanisms of virtual memory
  - Conclusion: Feasible, but does not seem like it is worth the extra work

RPC Structure

1. User
2. User stub
   1. Places specs of target procedure and arguments into packets
3. RPC Communications
   - Reliably passed packets to the callee
4. Server stub
5. Server

Simple Call over RPC

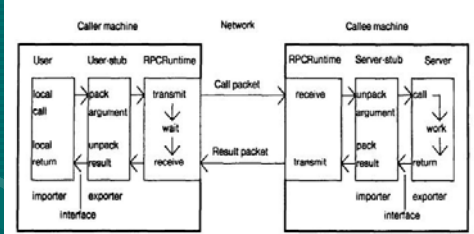


Fig. 1. The components of the system, and their interactions for a simple call.

Binding

- Naming: What?
  - Type
  - Instance
- Location: Where?
  - Statically include location in app
    - Binds too early
  - Broadcast
    - Too much interference for bystanders
    - Not convenient for binding to machines not on the local network
  - Distributed database

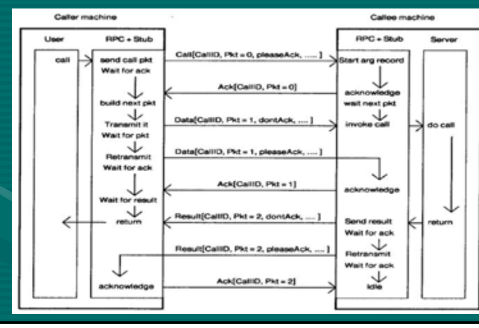
Binding – cont.

- Grapevine
  - Distributed database that is widely and reliably available
  - Replicates data
  - Entries are of two types:
    - Groups (corresponds to Types)
    - Individuals (corresponds to Instance)

Binding on the Callee

- Exports maintained in a table
- For each export table contains:
  - Interface name
  - Dispatcher procedure from server stub
  - 32 bit machine relative unique id of the export
- When caller imports:
  - Give it 1) the 32 bit id, and 2) the index
  - This servers as a verification scheme

Binding: In Action



A few words on …

- No heavy state management
  - With many imports
  - With idle users (we'll get to this)
- Authentication
  - Through Grapevine
- When to bind
  - User specifies only type and not instance
  - Instance is an RName (so lookup at bind type)
  - Specify a network address in the app
  - Dynamically instantiate interfaces & import them.

PAUSE

- Is RPC hiding too many details from the programmers?
- Distributed Communication with No Timeout?
- RPC vs. Local Procedure Calls
  - Looks like
  - Feels like it (maybe…)
  - However, it is not a local call!
- Paper claims that it removes unnecessary difficulties, but it leaves real difficulties of distributed systems.
  - These difficulties can be easily overlooked with this level of abstraction

Transport Protocol

- Existing Protocols
  - Work, but not fast enough
  - Meant for bulk data transfer
- Custom Protocols
  - Potentially 10 X faster
  - Focus on minimizing elapsed time b/w request and response
  - Quick setup and teardown
  - Minimal state
  - Make servers scalable

Guarantees

- If call returns:
  - Procedure executed exactly once
- Else
  - Exception reported to user
  - Procedure invoked 0 or 1 times
  - No upper bound on how long to wait for results
    - Unless communication breakdown
    - No time out if server deadlocks
    - Argument: make it as similar to local calls as possible

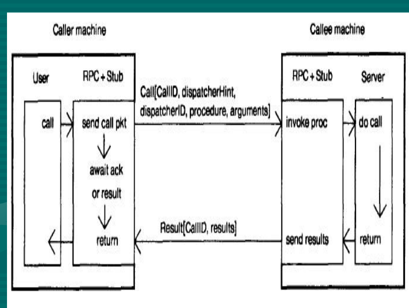
Simple Calls

- Focus on efficiency for calls where all the arguments/results fit in one package each
- Reduce explicit ACKS for quick/frequent requests
  - Response can be the ack for request
  - Next request can be the ack for response

The Call Identifier

- <Machine Id, PID, seq #>
- Allows:
  - eliminations of duplicate calls packets
  - Caller to determine if the result packet is the result of the current request
- Activity = <Machine_id, Process>
  - Nice property: each activity can have at most one procedure executing remotely
  - Call disregarded if the seq # is not greater than the last seq # used for activity

Simple Call Illustrated



Connection Maintenance

- During Idle Connections:
  - Only state on callee is an entry in the table <activity, seq#>
  - Caller only has the machine wide counter
  - No pings required to maintain this state
- No Teardown!

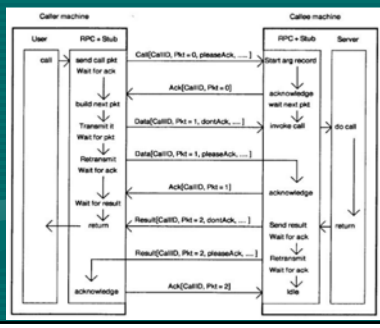
Complicated Calls

- If the calls takes too long:
  - Caller periodically sends a probe packet to callee that the callee acks
  - First probe sent after ~ 1 RTT. Duration of subsequent probe transmission increases gradually until there is one probe/5 minutes
  - Probes retransmitted n times if no ack
  - Notice how this ONLY detects communication failure
- Could have also done this on the callee side: transmit a keep alive if response is not generated quickly

Complicated Calls – Cont.

- What if the arguments don't fit in one packet?
  - Sent in multiple packets, each requesting an explicit ack except the last one
  - Caller and Callee send packts alternatively
- Problem: only one packet in the buffer
  - Really bad for networks with high bandwidth, but high latency
  - Remember: this was optimized for simple calls
  - Mitigated by multiplexing through multiple processes
- Future research: automatically switch to a bulk data transfer protocol for large requests

Complicated Calls – Cont.



Exception Handling

- Callee can return exception packet to caller
  - Packet handled by RPCRuntime on caller
  - RPCRuntime raises the exception in the proper process
  - If there is a catch phrase, it is executed and results are passed to callee. If catch phrase jumps out, callee is notified
  - RPCRuntime can also raise exception during communication failure

Security

- Grapevine is the authentication/ key distribution service
- Full End to End encryption
  - Confidentiality and integrity
- Complex security details left for another paper

Performance Results

- 2 Dorados connected by 3Mbps Ethernet
- Lightly loaded network, 5-10% capacity
- Accomplished 2Mbps on the 3Mbps network through parallel calls
- Did not measure cost of export/import

Performance Results

Table I. Performance Results for Some Examples of Remote Calls

Procedure	Minimum	Median	Transmission	Local-only
no args/results	1059	1097	131	9
1 arg/result	1070	1105	142	10
2 args/results	1077	1127	152	11
4 args/results	1115	1171	174	12
10 args/results	1222	1278	239	17
1 word array	1069	1111	131	10
4 word array	1106	1153	174	13
10 word array	1214	1250	239	16
40 word array	1643	1695	566	51
100 word array	2915	2926	1219	98
resume except'n	2555	2637	284	134
unwind except'n	3374	3467	284	196

Performance of FireFly RPC

By Shroeder and Burrows

Presentation by Khawaja Shams
Draws From Presentation by
Swati Agarwal

Goals

- Make RPC much faster
  - Otherwise, developers may be tempted to design their own communication protocols
- Good same machine performance
  - FireFly RPC costs 60x a local procedure call
  - Compatible with Bershad's system: 20x latency

Hardware

- Multiple VAX processors can access a shared memory system through a coherent cache
- 5 Micro VAX II CPUs: one connected to a QBus I/O bus
- Network access through a DEQNA device controller that connects QBus to a 10 Mbps ethernet.

Measurements

- PROCEDURE: Null( ) ;
  - Measures base latency of RPC
- PROCEDURE: MaxResult(VAR OUT buf: ARRAY OF CHAR)
  - Measurers server to caller throughput
  - No argument, 1440 byte array result
- PROCEDURE: MaxArg(VAR IN buf: ARRAY of CHAR)
  - Measures caller to server throughput
  - 1440 byte array as arg, no result
- Notice the array size : 1440
  - Considering the max ethernet packet size of 1514, and the 74 byte header, this fills the packet completely

Results

Table I. Time for 10,000 RPCs

Number of caller threads	Calls to Null( )		Calls to MaxResult(b)	
	s	RPCs/s	s	Mbits/s
1	26.61	375	63.47	1.82
2	16.80	595	35.28	3.28
3	16.26	615	27.28	4.25
4	15.45	647	24.93	4.65
5	15.11	662	24.69	4.69
6	14.69	680	24.65	4.70
7	13.49	741	24.72	4.69
8	13.67	732	24.68	4.69

Comments on Result

- Base latency of RPC 2.6ms
- Notice how the throughput almost doubles when going from single thread to two threads
- Max bandwidth results with 6 threads: 4.7 Mbps
- 7 threads accomplish 741 RPCs/s
- Are these tests really a true indicator of RPC throughput?

Notes on VARs

- Var Out: If results fits in a single packet, then the server stub passed address of result in the result packet buffer to the server procedure, so that server can directly write to the buffer (avoids copy)

Marshalling Time

Marshalling time scales linearly with size for simple args

Table III. Fixed-Length Array, Passed by VAR OUT

Array size (bytes)	Marshaling time (μs)
4	20
400	140

Table IV. Variable-Length Array, Passed by VAR OUT

Array size (bytes)	Marshaling time (μs)
1	115
1440	550

Table V. Text.T Argument

Array size (bytes)	Marshaling time (μs)
NIL	89
1	378
128	659

Notice the huge latency due to memory allocation and invoking library functions

Steps in RPC

- Caller
  - Obtain a packet buffer with partially filled in buffer, Marshal arguments into the packet, Transmit, Unmarshal result from results packet, Send packet back to pool
- Server
  - Unmarshall the call's argument, hand the argument to the server component, when the server component returns, marshall the results in the result packet (the saved call packet)

Transporter Mechanism

- Transporter
  - Fill RPC header in call packet
  - Call Sender - fills in other headers
  - Send packet on Ethernet (queue it, notify Ethernet controller)
  - Register outstanding call in RPC call table, wait for result packet (not part of RPC fast path)
- Packet-arrival interrupt on server
- Wake server thread - Receiver
- Return result (send+receive)

Reducing Latency

- Usage of direct assignments rather than calling library procedures for marshalling
- Starter, Transporter and Ender through procedure variables not through table lookup
- Interrupt routine wakes up correct thread
  - OS doesn't de-multiplex incoming packet
    - For Null(), going through OS (for both call and result) takes 4.5 ms
- Server reuses call packet for result
- RPC packet buffers reside in shared memory to eliminate the need for extra address mapping operations or copying when doing RPC.
  - Can be a security issue in a time shared environment

Accounting for the time

Table VII. Latency of Stubs and RPC Run Time

Machine	Procedure	μs
Caller	Calling program (loop to repeat call)	16
	Calling stub (call and return)	90
	Starter	128
	Transporter (send call packet)	27
Server	Receiver (receive call packet)	158
	Server stub (call and return)	68
	Null (the server procedure)	10
	Receiver (send result pkt)	27
Caller	Transporter (receive result pkt)	49
	Ender	33
Total		606

Table VIII. Calculation of Latency for RPC to Null( ) and MaxResult(b)

Procedure	Action	μs
Null( )	Caller, server, stubs, RPC run time	606
	Send+receive 74-byte call packet	954
	Send+receive 74-byte result packet	954
	Total	2514
MaxResult(b)	Caller, server, stubs, RPC run time	606
	Marshall 1440-byte result packet	550
	Send+receive 74-byte call packet	954
	Send+receive 1514-byte result pkt	4414
	Total	6524

- Write fast path code in assembly not in Modula2+
  - Speeded up by a factor of 3
  - Application behavior unchanged

Table IX. Execution Time for the Main Path of the Ethernet Interrupt Routine

Version	Time (μs)
Original Modula2+	758
Best Modula2+	547
Assembly language	177

Improvement Speculation

- Different Network Controller
  - Save 11 % on Null() and 28 % on MaxResult
- Faster Network – 100 Mbps Ethernet
  - Null – 4 %, MaxResult – 18%
- Faster CPUs
  - Null – 52 %, MaxResult – 36 %
- Omit UDP checksums
  - Ethernet controller occasionally makes errors
- Redesign RPC Protocol

- Omit layering on IP and UDP
- Busy Wait – caller and server threads
  - Time for wakeup can be saved
- Recode RPC run-time routines

Varying Numbers of Processors

Table X. Calls to Null( ) from One Thread with Varying Numbers of Processors

Caller processors	Server processors	Number of seconds for 1000 calls
5	5	2.69
4	5	2.73
3	5	2.85
2	5	2.98
1	5	3.96
1	4	3.98
1	3	4.13
1	2	4.21
1	1	4.81

Table XI. Throughput in Mbits/s of MaxResult(b) with Varying Numbers of Processors and Threads

Caller processors:	5	1	1
Server processors:	5	5	1
1 caller thread	2.0	1.5	1.3
2 caller threads	3.4	2.3	2.0
3 caller threads	4.6	2.7	2.4
4 caller threads	4.7	2.7	2.5
5 caller threads	4.7	2.7	2.5

Comments

- Reducing caller processors from 5 to 2 increases the latency by only 10%
- Sharp jump in latency for uni-processor caller
- Uni processors are not the focus of Firefly RPC
- Fast path is abandoned on a Uni Processor often by lock conflicts

Conclusion

- Main Objective:
  - Make RPC primary communication mechanism between address spaces (inter machine and same machine)
- Make RPC fast so developers have no excuse not to use it
- Functions and characteristics of RPC maintained
- Not the faster RPC in the game