Remote Procedure Call

Remote Procedure Call

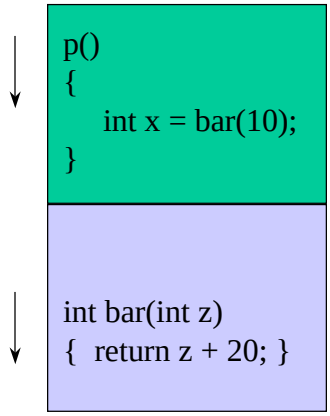
- Integrate network communication with programming language
- Procedure call is well understood
  - implementation
  - use
- Control transfer
- Data transfer

Goals

- Easy
  - make it look like PC at all costs
- Simple
  - make sure it is implementable
- Fast
  - optimize ruthlessly for the common case

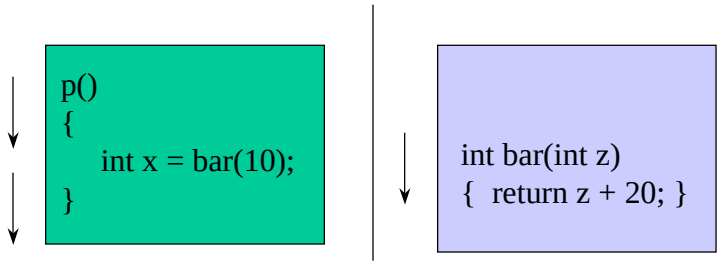
The Procedure Call Model

- Caller and callee run in the same address space
- Caller is suspended
- Data passes from caller to callee
- Callee executes procedure
- Data passes from callee to caller
- Caller is resumed
- Compiler takes care of the dirty work



The RPC Model

- Caller and callee run on different machines
- Caller is suspended
- Message passes from caller to callee
- Callee executes procedure
- Message passes from callee to caller
- Caller is resumed
- RPC system takes care of the dirty work



Why not messages?

- sendMessage(p,M); recvMessage(p, M)
  - UDP sockets, Mach messages
  - options aplenty
    - synchronous vs. asynchronous
    - reliable vs. unreliable
    - ordered vs. unordered
- Bottom line is that messages do the right thing, but are hard to use.

Why not Shared Memory?

- Initial RPC implementation discounted possibility of using shared memory
  - it was hard
    - Spector's remote reference was too slow
    - page-based DSM was not invented yet
    - language/compiler support for objects not yet discovered
  - it was more difficult
    - direct exposure of synchronization
    - interface at much too low a level

Problems with Shared Memory

- LOAD/STORE interface
- much much slower than it "looks"
- limited protection levels
- anonymous accesses (passive)
- sharing granularity is large (address space)
- no access control (eg, enforced locking)
- uncontained failure (both between and within a program)

The do's and the don't's

RPC DOES

- Simplify construction of distributed programs
- Hides many details of communication and failure

RPC DOES NOT

- Make it trivial to build distributed programs
- Hide all the details of communication, errors, and failure.

Even though RPC does not do everything, it's an incredibly useful tool.

Notable RPC Systems

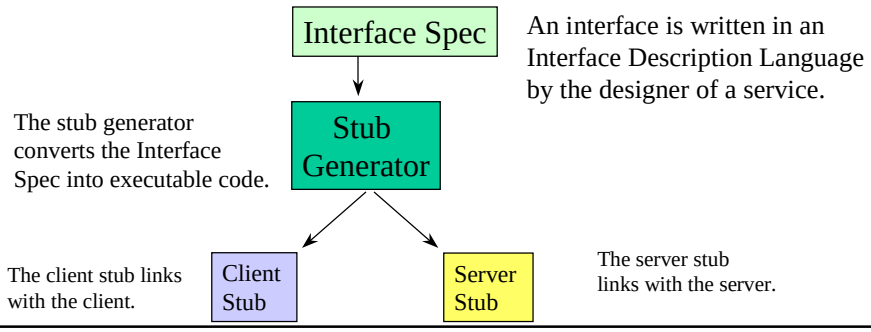
- Nelson's PhD Thesis
- Courier
- Cedar RPC
- SUN RPC
- Mach RPC
- DEC SRC RPC
- CORBA
- JAVA RMI

Interfaces

- Key assumption of all RPC systems is that the interface between client and server is well defined.
- An interface contains
  - the names and arguments of all exported procedures
  - the types of all arguments
- Type information allows the RPC system to serialize (*marshall*) arguments

Stubs

- Turn procedure arguments/return values into/out of messages
- Interfaces are statically defined.

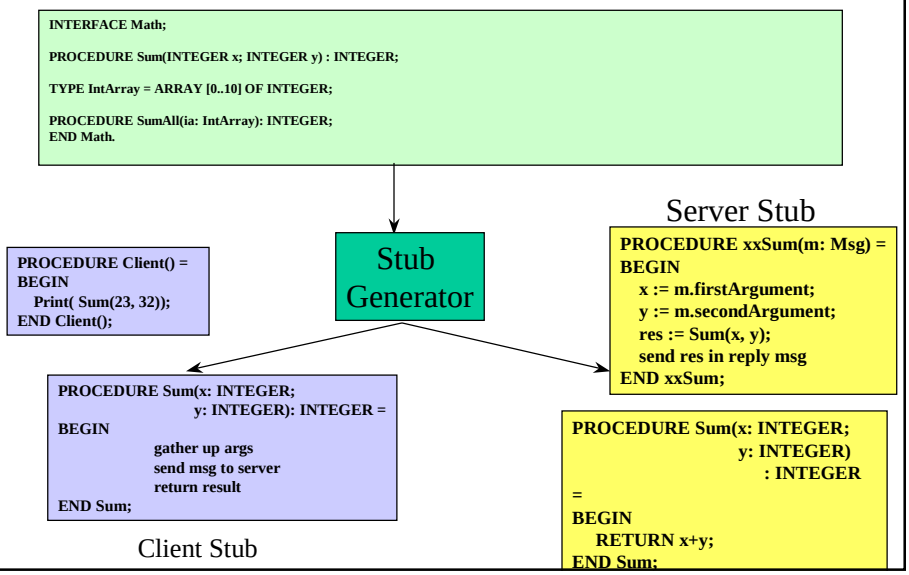


Example

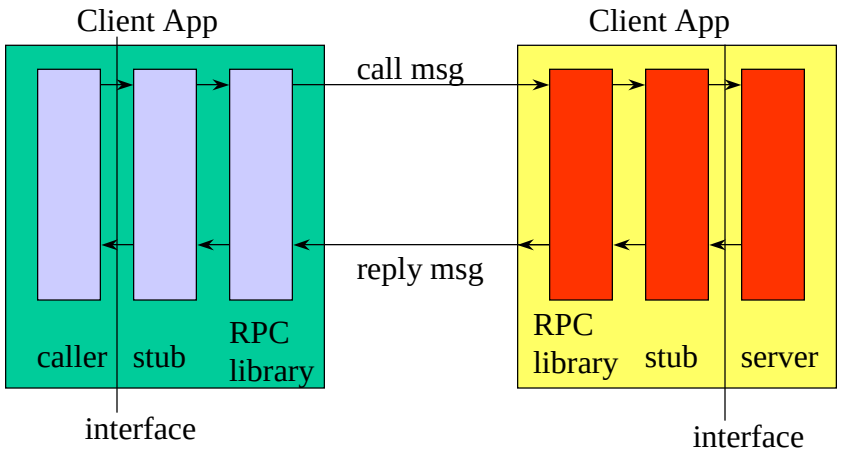
```
INTERFACE Math;

PROCEDURE Sum(INTEGER x; INTEGER y) : INTEGER;

TYPE IntArray = ARRAY [0..10] OF INTEGER;

PROCEDURE SumAll(ia: IntArray): INTEGER;
END Math.
```

- An interface completely defines an exported service.
  - Limits access to service
  - Enables access to service

Interfaces and Stubs



The Big Picture



Who does what?

- To the client, the client stub looks like the server
- To the server, the server stub looks like the client
- Stubs marshall and unmarshall arguments and results
- RPC libraries handle reliable messaging and conceal the network

Problems with stubs

- Large parameters must be marshalled apriori
- Cyclic structures hard to deal with
- Hard to pass procedure parameters
- Call by value semantics not always what we want
- No global variables

Transport Protocol

- Any will do, as RPC runtime specifies the only visible network interface
  - TCP, UDP
- Simple request/reply is best
  - Goal is to minimize number of messages
  - Leverage communication patterns for reliability
  - Bulk transfer with multiple threads
  - Consider TCP/IP vs UDP

Why not streaming protocols?

- Streaming protocols intended for bulk transfer
- Feel around for good bandwidth.
  - adapt slowly to improvements and quickly to degradation
- Large setup time, teardown, and connection state overhead
- You need connection state information in RPC layer anyway

The Birrell and Nelson Protocol

- Reliable
- At-most-once semantics
- Optimized for simple (small, short lived) calls
- Complex calls work, but are slow

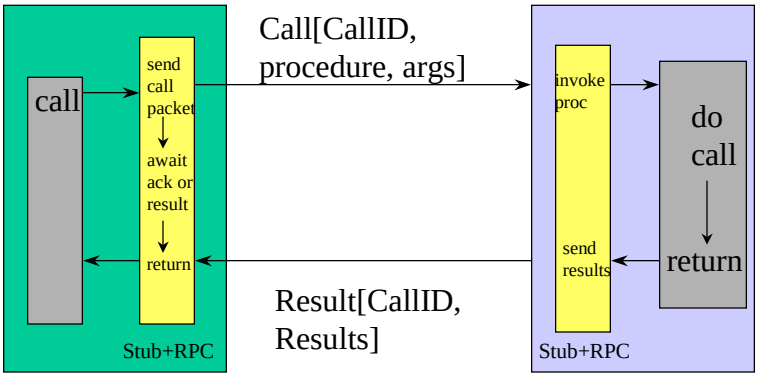
Strategy

- Sender always retains last sent packet until ack is received
  - acks can be explicit (ACK) or implicit (next call in sequence)
- Key concept is CallID
  - ([MachineID, Process], SeqNo) -> (activity, event in activity)
  - activity can have one call outstanding
    - stream with window size of one.
    - easy duplicate suppression
      - broken interface, gateway
      - delayed initial message. Rexmit.
      - delayed response message. Rexmit.

Duplicate Suppression

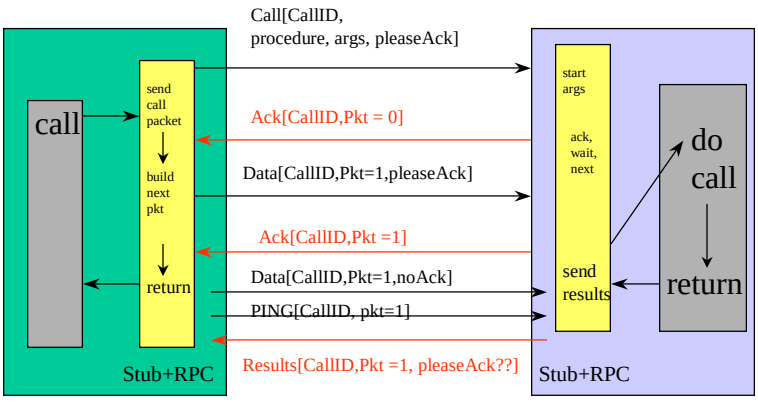
- each new call includes a callID at least one more than the last
- on receipt of a new packet, if callID > lastCallID, ok. Else, is duplicate
- on receipt of a reply packet, if callID == lastCallID, ok, else duplicate.
- servers can flush call tables after a few minutes?
  - really, consider debug.

The Simple B&N Protocol



The result message serves as the call's acknowledgement

The Not So Simple B&N Protocol



For big or long calls

Implicit Acks

- Server can avoid work if not necessary
  - client is down, or running on slower processor
- Client can implement its own timeout policies
- Still need support for client ping if server ack is lost
  - increases delay until client can determine server failure

Transparency Issues

- Goal is to make semantics of remote call match local call
- Regular procedure call
  - at most once semantics
- RPC must deal with
  - communication and site failure
  - really want zero-or-one semantics but hard to implement efficiently
    - see transactions

Binding Issues

- Question is when does client “connect” to server?
- In local case, binding is simple and implicit
  - at link time, or program instantiation time.
  - failure is not an issue
- In remote case, binding must be made explicit
  - servers can move or have multiple instatiations
- Failure at bind time is easy to deal with

Binding and Naming

- Servers export interfaces through a name server
- Clients import interfaces through a name server
- Name server maps service names into network addresses
  - service names are generally text strings
- Services (including the name server) may be replicated for availability

Heterogeneity

- How to deal with client and server being of different types?
  - architecture, OS, programming languages?
- Fortunately, the interface specifies at a high level what the relationship is.
- Static IDL solves many problems
  - procedures and types are pre-declared
  - client/server can negotiate type formats
  - standard wire format or tagged arguments

Concurrency

- Caller is suspended while RPC executes
- Single threaded systems are a problem for client and server
  - SunOS, BSD
  - heavyweight OS processes for concurrency
- Threads naturally complement RPC systems
  - one call per thread
  - one service request per thread

Performance

- Fast RPC is now well understood
- Overhead is about 5% on top of what you would get if you rolled your own protocol
- The bottlenecks are not in the stubs
  - Network and host interface
- Although this is changing with faster networks
  - Calls for higher performance request/reply services

Avoiding Process Overhead

- Processes are expensive
  - create, destroy, switch to
- Avoid doing process management
  - cache idle processes on server
  - include process ID as a “hint” to dispatcher
  - do direct dispatch from device driver
- Result
  - for simple calls, no new processes are created
  - four context switches
    - fewer if client or server are otherwise idle

What about the paper?

- Possible to precisely account for latency
  - we can bicker over the strategy…
  - but a computer's just a big clock
    - nothing magic

- Buffer management is critical
  - so critical you are allowed to cheat
- Assembly language is faster than not assembly language
- Network controller counts
- IP and UDP layers not totally useless
- Spare processors are always a good thing to have.

Other Issues