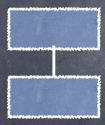
Enforcing modularity

How do modules communicate?

- Within the same address space and protection domain
  - local procedure calls
- Across protection domain
  - system calls
- Over a connection
  - client/server programming model
    - Remote Procedure Call
    - socket (local or across the network)



Modularity: client-server organization

- All communication through messages
  - modules are isolated
  - propagation of errors is reduced
    - cfr. soft modularity in procedure calls
- Encourages a clean design
  - even within a single computer!
    - modules can be thought of on separate computers

Tradeoffs of enforcing modularity

- Module granularity tend to be coarser
  - crossing a layer of abstraction is more expensive!
    - to amortize, each message may elicit significant computation
    - e.g. databases, fileservers
- Many design decisions
  - how much client state should a server keep?
  - how should modules handle their partner's failure?
  - does the server trust the client?
  - does the client trust the server?

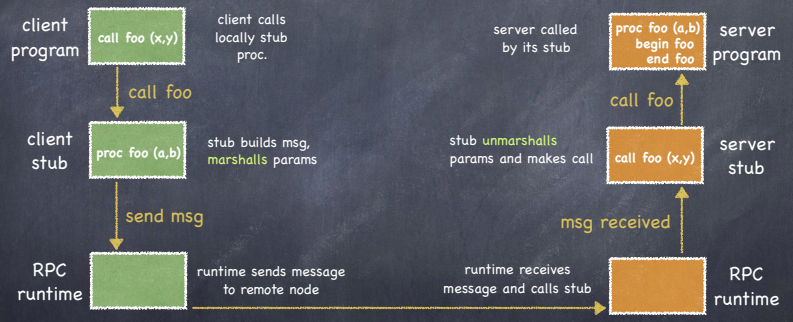
Remote Procedure Call (RPC) Birrell & Nelson, 1984

- Hide message passing I/O from programmer
- Make RPC look as much as possible as PC
- Three components on each side
  - user program (client or server)
  - (compiled) stubs
  - RPC runtime support

Building a server

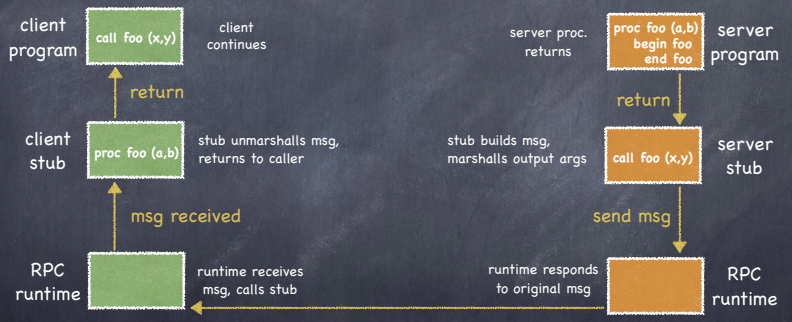
- Server interface defined via an Interface Definition Language
  - specifies names, parameters, types of client-callable procedures
- Stub compiler reads IDL to produce client and server stubs
  - stubs manage all details of remote communication
- Server developer links server code with server-side stub — similarly at the client

RPC Call structure



- Communication on top of XDR
  - these days, on top of web services
    - RESTful APIs, AJAX.

RPC Return structure



- Communication on top of XDR
  - these days, on top of web services
    - RESTful APIs, AJAX.

RPC binding

Server

- On startup, exports its interface
  - identifies itself to a network name server
  - tell local runtime its dispatcher address

Client

- Before issuing any calls, imports server
  - RPC runtime looks up server through name service and contacts it to create a connection

RPC is not PC

- No shared address space
  - no pointers; data structures must be fully marshalled
- Performance
  - cost of procedure call << same machine RPC << network RPC
- Fate sharing
  - client and server fail independently
  - fault-tolerance handled end-to-end
    - three execution semantics
      - "At least once" (SUN RPC)
      - "At most once" (Java RMI, HTTP)
      - "Exactly once"

"At least once": if succeeds: at least one execution; if fails: none, partial, multiple — implemented by retransmission on timeout; works only if operation is idempotent

"At most once": if succeeds: exactly once; if fails: none, — implemented by server filtering duplicate requests

"Exactly once": if succeeds: exactly once; if fails: none, partial, one — Needs updating persistent state at the server on every receipt

What granularity of protection within OS?

- Monolithic kernel
  - no protection within kernel
    - hard to modify, debug, validate
- Fine-grained protection with capabilities
  - objects share address space but have individual protection domain
    - difficult to implement efficiently and develop for
- Small kernel
  - functionalities are in separate, larger domains/address spaces, communicating via RPC
    - better fault-isolation, debugging, easier implementation/maintenance

The catch

- RPC is general, but inefficient

```
procedure Null()
  begin
  return
end Null
```

- Theoretical minimum time { procedure call, kernel trap, VM context switch } X 2
- Cross domain performance over six system is 300% to over 800% that of theoretical min
  - pressure to colocate logically distinct functionalities in single domain

Handle normal case and worst case separately

B. Lampson. "Hints for Computer Systems Design"

1. Few cross-domain calls cross machine boundaries
   - 0.6% (SUN Unix+NFS); 3% (V); 5.3% (Taos)

2. Most calls send few bytes

3. No need for complex marshalling
   - byte copy suffices

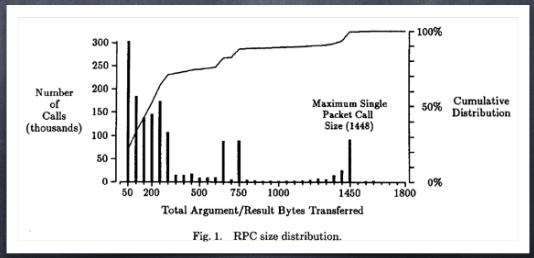


Fig. 1. RPC size distribution.

Lightweight RPC

Bershad, Anderson, Lozowska, Levy 1990

- A combination of protected and remote procedure calls
  - Executions model: protected procedure call
    - call to server made through kernel trap
    - kernel validates caller, creates linkage (?), dispatches client's concrete thread to server
    - upon procedure's completion, kernel returns control to client
  - Semantics: RPC
    - server exports interfaces
    - clients bind interfaces
    - large-grain protection model

LRPC binding

- Conceptually similar to RPC...
  - Server exports interface to name server; client imports (via kernel call)
- ... but different implementation
  - mediated by the kernel
  - server's clerk returns to kernell a procedure descriptor (PD) list
  - one PD per exported procedure

LRPC binding: mechanisms

- Procedure descriptor
  - procedure's entry address in server domain
  - count C of permitted simultaneous invocation
  - size of the procedure's argument stack (A-stack)
- A-stack
  - holds arguments and return values for a call
  - r/w addressable by both client and server
  - kernel allocates C A-stacks
  - upon binding, client learns location of A-stacks for procedures in interface

LRPC binding: mechanisms

- Linkage record
  - stores caller's return address
  - only accessible by kernel
  - one per A-stack, easy to refer to from A-stack
- Binding object
  - returned once binding completes
  - a capability to access the server interface
  - must be presented at each call

LRPC calls

Client's stub

Kernel

LRPC calls

Client's stub

- picks an A-stack
- pushes procedure's arguments
- loads registers with
  - A-stack's address
  - Binding Object
  - ID of procedure
- traps to kernel

Kernel

LRPC calls

Client's stub

- picks an A-stack
- pushes procedure's arguments
- loads registers with
  - A-stack's address
  - Binding Object
  - ID of procedure
- traps to kernel

Kernel

- verifies BO and procedure ID, locates PD
- verifies A-stack; find its linkage
- ensures A-stack, linkage are unused
- stores caller's return address in linkage
- pushes linkage on stack of linkages in TCB
- finds an execution stack (E-stack) in server's domain
- sets threads SP to use E- stack
- changes VM registers
- upcalls in server's stub to procedure's address specified in PD

LRPC returns

- Server's stub traps to the kernel
  - no need to verify rights: they were granted at call time
  - client-specified A-stack contains return values
  - NO EXPLICIT MESSAGE PASSING!

Fun details

- Call by reference
  - objects passed by reference are copied on A-stack
  - server creates a reference to address in A-stack
- E-stack/A-stack pairing
  - dynamic (E-stacks can be large!)
- Leveraging multiprocessors to reduce c-switch overhead
  - Calling thread placed on idle processor in server's domain context (and viceversa on return!)
- Reducing argument copying

client's stub's stack to RPC message
RPC message to kernel
one copy to A-stack
kernel to RPC message
RPC message to server's stub

The uncommon cases

- Call to a truly remote server
  - detected on first stub instruction
    - bit set in the BO
- Size of A-stack
  - easy if arguments known at compile time. If not...
  - ... stub sets size to Ethernet packet
    - can be overridden; can use out-of-band memory segment

The uncommon cases

- Domain termination
  - if server's , client's outstanding call must return
  - if client's, outstanding calls should not return
- The kernel
  - revokes BOs for terminated domain
    - no more in or out calls!
  - client threads with outstanding LRPC are restarted with call-failed exception
  - active linkage records are invalidated
    - invalid linkage record on return raises call-failed exception in caller

Looking back

- Four techniques key to LRPC's performance
  - Simple control transfer
    - client's thread executes requested procedure in server's domain
  - Simple data transfer
    - as in PC, shared argument stack elinates redundant copies
  - Simple stubs
    - because data and cotrol transfers are simple, so are stubs
  - Design for concurrency
    - leverages potential speedup from multiprocessors

NFS
Sun's Network File System

NFS
Sun's Network File System

- Mid 80s — the dawn of a new era!
  - individual workstations
  - potentially diskless
  - NFS aims to provide a shared file system
    - saves money
    - easy to share and backup files
    - RPC is its core mechanism

Sun-1, 1982

Goals

- Compatible with existing applications
  - aim for same semantics as a UNIX file system
- Easy to deploy
  - should retrofit into existing UNIX systems
- Not too UNIX specific
  - should work with DOS
- Efficient enough
  - adequate, but not necessarily as fast as a local file system

Using NFS

- Same user interface as UNIX FS
  - `OPEN ("/users/lorenzo/pong.c", READONLY)`
- Easy to deploy (host : path)
  - % mount -t nfs 10.131.248.8:/nfs/dir /mnt/nfs/dir
    - RPC to host, asking for a file handle for the object identified by path
    - all future RPCs for that object include its file handle

Key design choice

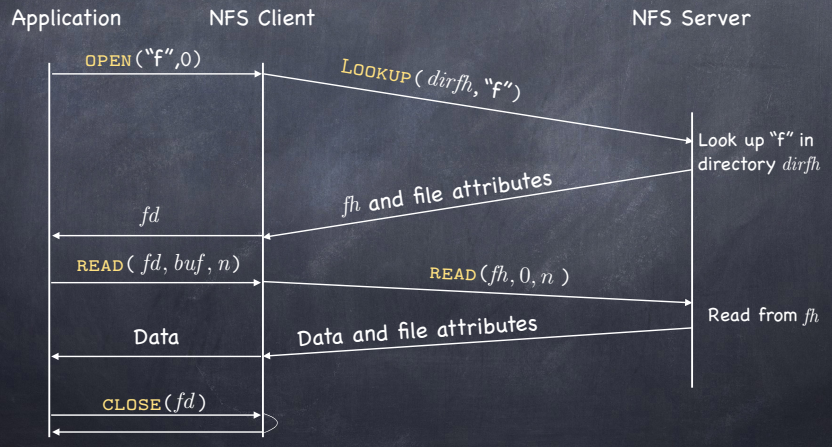
Stateless server

Server maintains no other state than on-disk files

RPC arguments contain all information necessary to carry out request

Example:

```
fd ← OPEN("f", READONLY)
READ(fd, buf, n)
CLOSE(fd)
```



What should a file handle include?

- ~~The path name of the requested object~~

Program 1 on client 1	Program 2 on client 2
1. CHDIR("dir1")	
2. fd ← OPEN("f",READONLY)	
3.	RENAME("dir1", "dir2")
4.	RENAME("dir3", "dir1")
5. READ(fd, buf, n)	

Which file should be read?

In Unix, it would be "dir2/f" — NFS would like to keep that behavior

The file handle

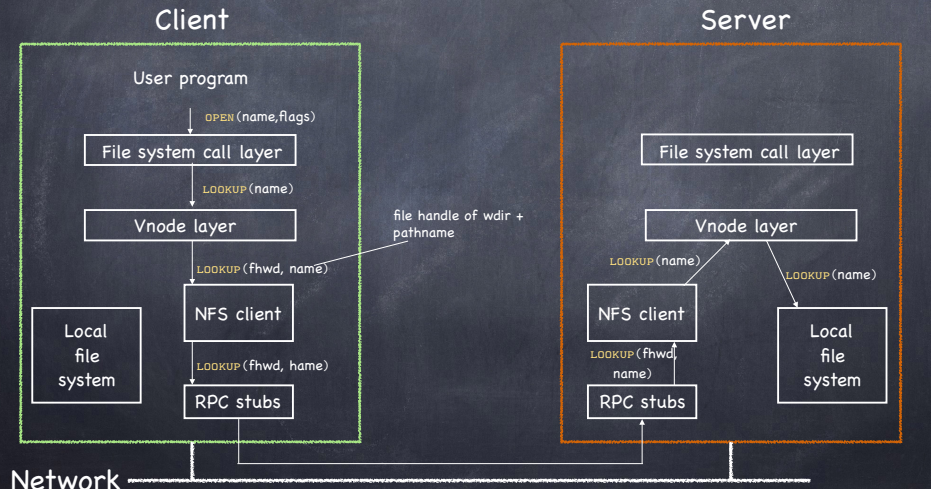
- Inode number
- File system identifier

Program 1 on client 1	Program 2 on client 2
1.	$fd \leftarrow$ OPEN("f",READONLY)
2. UNLINK("f")	
3. $fd \leftarrow$ OPEN("f",CREATE)	
4.	READ(fd, buf, n)

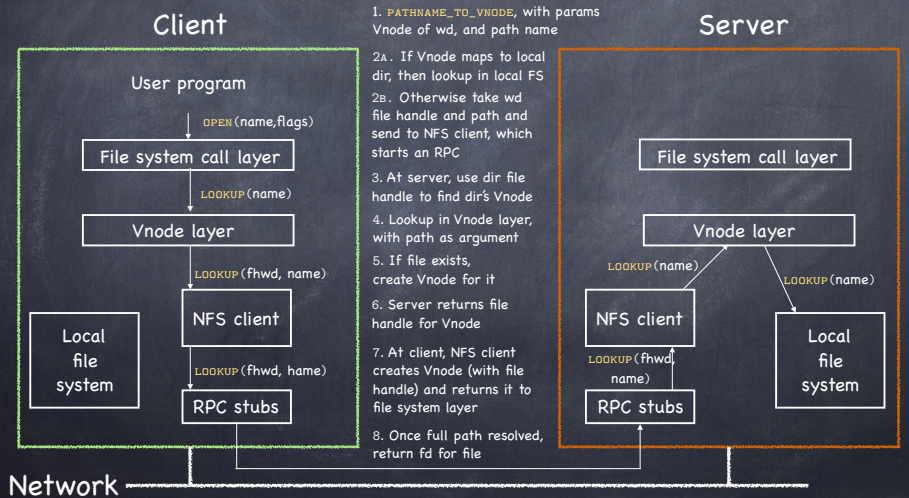
what happens in Unix?

- Generation number
  - incremented every time inode is reused
  - Not quite the same semantics as UNIX

Extending the UNIX FS to support NFS



Extending the UNIX FS to support NFS



1. PATHNAME_TO_VNODE, with params Vnode of wd, and path name

2A. If Vnode maps to local dir, then lookup in local FS

2B. Otherwise take wd file handle and path and send to NFS client, which starts an RPC

3. At server, use dir file handle to find dir's Vnode

4. Lookup in Vnode layer, with path as argument

5. If file exists, create Vnode for it

6. Server returns file handle for Vnode

7. At client, NFS client creates Vnode (with file handle) and returns it to file system layer

8. Once full path resolved, return fd for file

Caching: joys and tribulations

- NFS client caches
  - vnode for every open file
  - recently used vnodes
  - recently used blocks of cached vnodes
  - mapping from path name to vnode
- Lower latency, lower load on server
- What about coherence?

Coherence

- A read returns the result of the latest completed write
  - with caching?
  - write-through?
- NFS implements coherence only for certain operations

Close-to-open consistency

- A client that opens "f", will see the result of all writes to "f" performed by clients who have previously closed "f".
  - blocks cached at client have a timestamp recording last known modification of corresponding Vnode
  - on open, NFS client contacts server to learn current timestamp of Vnode
    - reads cached blocks only if retrieved timestamp is no larger
  - on close, client first sends modified blocks, and waits for acks before returning

What if the server crashes?

"Morto un Papa,
se ne fa un altro"

Italian proverb

Stateless server