Remote Procedure Calls

Problems with sockets

Sockets interface is straightforward
- [connect]
- read/write
- [disconnect]

Forces read/write mechanism
- *Not* how we generally program
- We usually use a **procedure call**

To make distributed computing look more like centralized:
- I/O is not the way to go

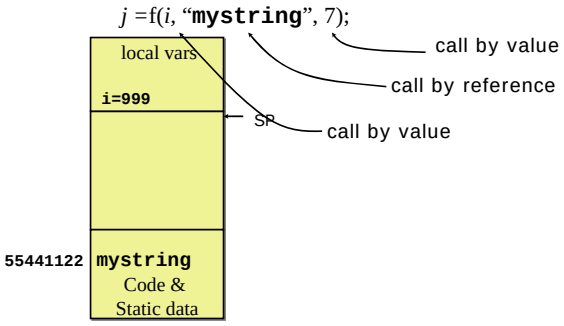
RPC

1984: Birrell & Nelson
- Mechanism to call procedures on other machines
- Process on machine A can call procedure on machine B
  - *A* is suspended
  - Execution continues on *B*
  - When *B* returns, control passed back to A

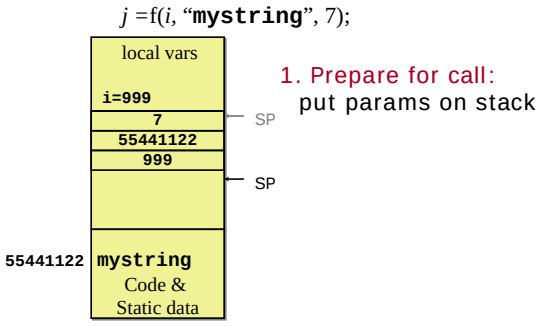
Remote Procedure Call

Goal: it should appear to the programmer that a normal call is taking place

Digression: local procedure calls

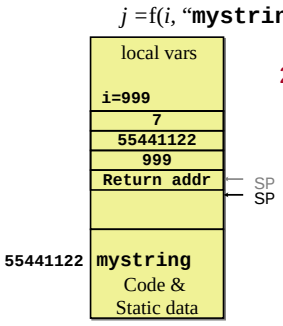


Digression: local procedure calls



1. Prepare for call:
   put params on stack

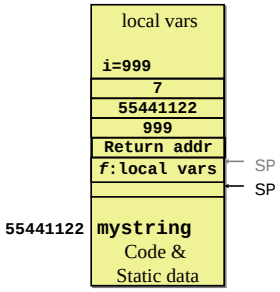
Digression: local procedure calls



2. Call:
   call f

Digression: local procedure calls

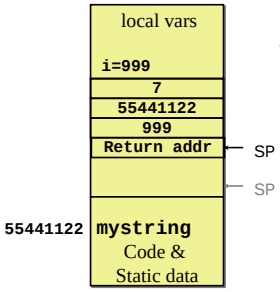
j = f(*i*, "**mystring**", 7);



3. On entry to f:
   adjust SP to allocate space for locals

Digression: local procedure calls

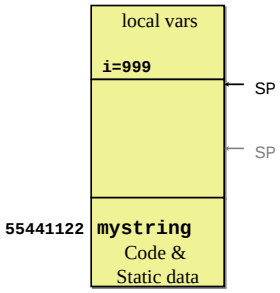
j = f(*i*, "**mystring**", 7);



4. Prepare to return:
   - return value in register
   - adjust SP
   - return

Digression: local procedure calls

j = f(*i*, "**mystring**", 7);



5. Return:
   caller cleans up parameters

Implementing RPC

No architectural support for remote procedure calls

Simulate it with tools we have (local procedure calls)

Simulation makes RPC a
language-level construct
instead of an
operating system construct

Implementing RPC

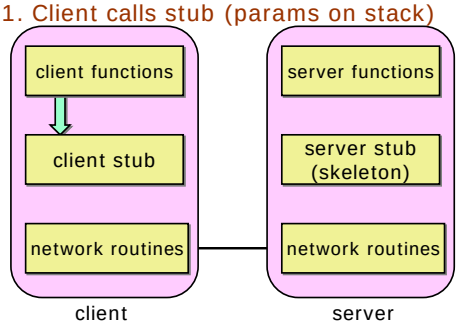
The trick:

Create **stub functions** to make it appear to the user that the call is local

Stub function contains the function's interface

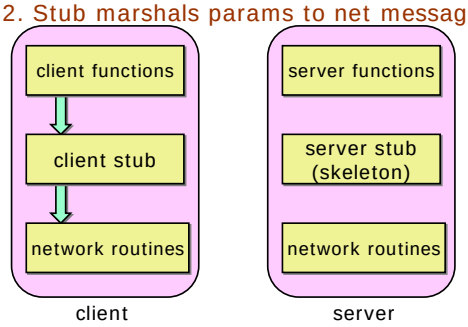
Stub functions

1. Client calls stub (params on stack)



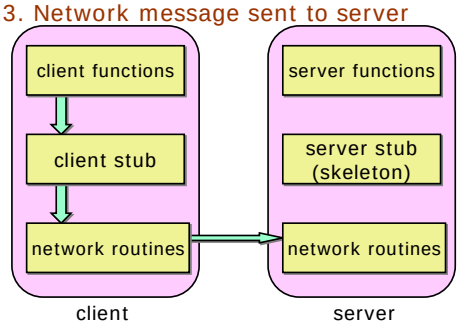
Stub functions

2. Stub marshals params to net message



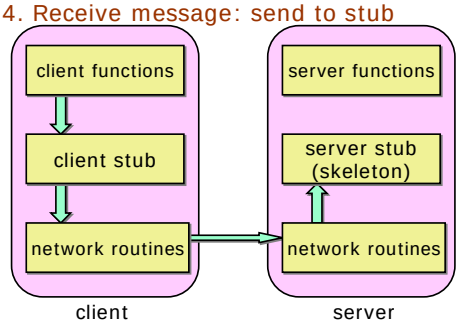
Stub functions

3. Network message sent to server



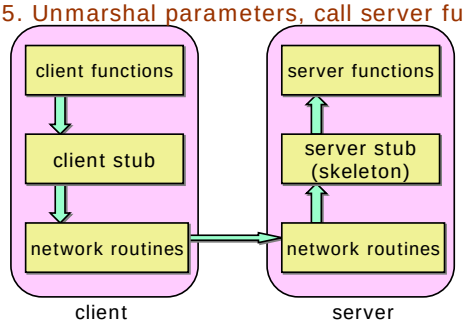
Stub functions

4. Receive message: send to stub



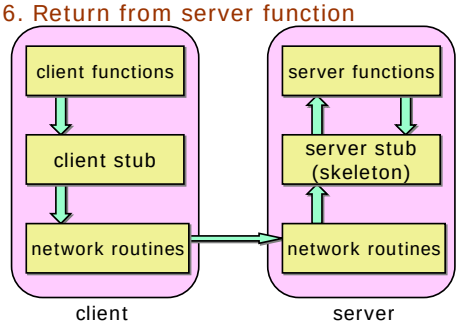
Stub functions

5. Unmarshal parameters, call server func



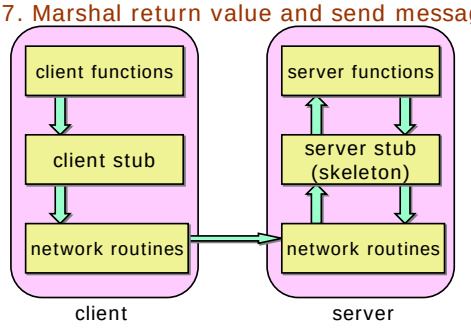
Stub functions

6. Return from server function



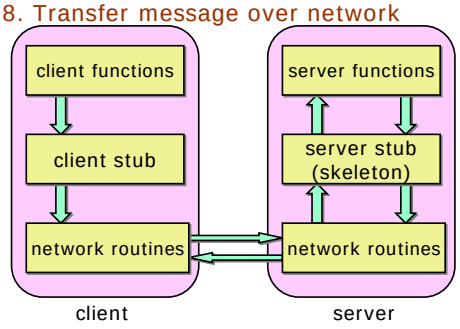
Stub functions

7. Marshal return value and send message



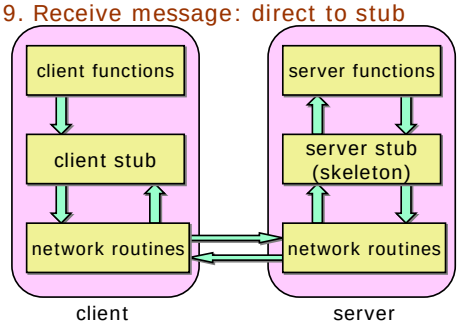
Stub functions

8. Transfer message over network



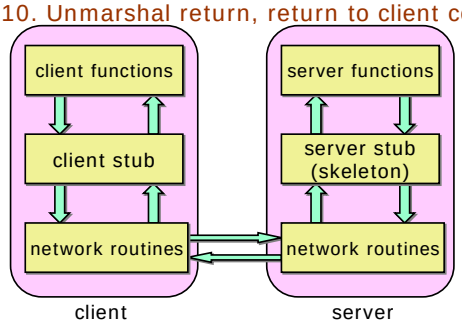
Stub functions

9. Receive message: direct to stub



Stub functions

10. Unmarshal return, return to client code



Benefits

- Procedure call interface
- Writing applications simplified
  - RPC hides all network code into stub functions
  - Application programmers don't have to worry about details
    - Sockets, port numbers, byte ordering
- RPC: presentation layer in OSI model

RPC has issues

Parameter passing

Pass by value
- Easy: just copy data to network message

Pass by reference
- Makes no sense without shared memory

Pass by reference?

1. Copy items referenced to message buffer
2. Ship them over
3. Unmarshal data at server
4. Pass *local* pointer to server stub function
5. Send new values back

To support complex structures

- Copy structure into pointerless representation
- Transmit
- Reconstruct structure with local pointers on server

Representing data

No such thing as ***incompatibility problems*** on local system

Remote machine may have:

- Different byte ordering
- Different sizes of integers and other types
- Different floating point representations
- Different character sets
- Alignment requirements

Representing data

IP (headers) forced all to use **big endian** byte ordering for 16 and 32 bit values

- Most significant byte in low memory
  - Sparc, 680x0, MIPS, G5
  - x86/Pentiums use little endian

```
main() {
    unsigned int n;
    char *a = (char *)&n;

    n = 0x11223344;
    printf("%02x, %02x, %02x, %02x\n",
            a[0], a[1], a[2], a[3]);
}
```

Output on a Pentium:
44, 33, 22, 11

Output on a G4:
11, 22, 33, 44

Representing data

Need standard encoding to enable communication between heterogeneous systems

- e.g. Sun's RPC uses XDR (eXternal Data Representation)

Representing data

Implicit typing

- only values are transmitted, not data types or parameter info
- e.g., Sun XDR

Explicit typing

- Type is transmitted with each value
- e.g., ISO's ASN.1, XML

Where to bind?

Need to locate host and correct server process

Where to bind? – Solution 1

Maintain centralized DB that can locate a host that provides a particular service
(Birrell & Nelson's 1984 proposal)

- Server sends message to central authority stating its willingness to accept certain remote procedure calls (and sends port number)
- Clients then contact this authority when they need to locate a service

Where to bind? – Solution 2

- Require client to know which host it needs to contact
- A server on that host maintains a DB of *locally* provided services
- Solution 1 is problematic for Sun NFS – identical file servers serve different file systems

Transport protocol

Which one?

- Some implementations may offer only one (e.g. TCP)
- Most support several
  - Allow programmer (or end user) to choose.

When things go wrong

- Local procedure calls do not fail
  - If they core dump, entire process dies
- More opportunities for error with RPC:
  - Server could generate error
  - Problems in network
  - Server crash
  - Client might disappear while server is still executing code for it
- Transparency breaks here
  - Applications should be prepared to deal with RPC failure

When things go wrong

- Semantics of remote procedure calls
  - Local procedure call: ***exactly once***
- Exactly once may be difficult to achieve with RPC
- A remote procedure call may be called:
  - 0 times: server crashed or server process died before executing server code
  - 1 time: everything worked well
  - 1 or more: excess latency or lost reply from server and client retransmission

RPC semantics

- Most RPC systems will offer either:
  - ***at least once*** semantics
  - or ***at most once*** semantics
- Understand application:
  - **idempotent** functions: may be run any number of times without harm
  - **non-idempotent** functions: side-effects

More issues

Performance

– RPC is slower … a lot slower

Security

– messages visible over network
– Authenticate client
– Authenticate server

Programming with RPC

Language support

– Most programming languages (C, C++, Java, …) have no concept of remote procedure calls
– Language compilers will not generate client and server stubs

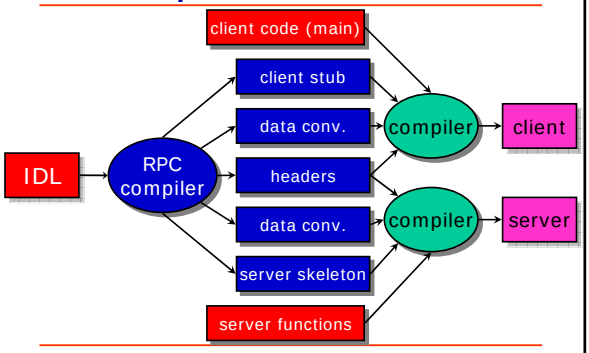
Common solution:

– Use a separate compiler to generate stubs (pre-compiler)

Interface Definition Language

- Allow programmer to specify remote procedure interfaces
  (names, parameters, return values)
- Pre-compiler can use this to generate client and server stubs:
  – Marshaling code
  – Unmarshaling code
  – Network transport routines
  – Conform to defined interface
- Similar to function prototypes

RPC compiler



Writing the program

Client code has to be modified

– Initialize RPC-related options
  • Transport type
  • Locate server/service
– Handle failure of remote procedure call

Server functions

– Generally need little or no modification

RPC API

What kind of services does an RPC system need?

- Name service operations
  – Export/lookup binding information (ports, machines)
  – Support dynamic ports
- Binding operations
  – Establish client/server communications using appropriate protocol (establish endpoints)
- Endpoint operations
  – Listen for requests, export endpoint to name server

RPC API

What kind of services does an RPC system need?

- Security operations
  - Authenticate client/server
- Internationalization operations
- Marshaling/data conversion operations
- Stub memory management
  - Dealing with “reference” data, temporary buffers
- Program ID operations
  - Allow applications to access IDs of RPC interfaces

To be continued …