Remote Procedure Calls

Rong Chen

Institute of Parallel and Distributed Systems

Shanghai Jiao Tong University

http://ipads.se.sjtu.edu.cn/rong_chen

Credits: some slides from P. Krzyzanowski (Rutgers) and et al.

Outline

RPC: Remote Procedure Calls

- Intro
- Implement
- Challenge
- Semantics

Lightweight RPC

Problems with the Sockets API

The sockets interface forces a read/write mechanism

Programming is often easier with a functional interface

To make distributed computing look more like centralized computing

I/O (read/write) is not the way to go

Birth of RPC

1984: Birrell & Nelson

- Mechanism to call procedure on other machines

Remote Procedure Call

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

Andrew Birrell, Bruce Nelson, "***Implementing Remote Procedure Calls***". ACM Transactions on Computer Systems (TOCS), 2(1), **1984**

Regular Procedure Calls

You write: `x = f(a, "test", 5)`

The compiler parses this and gen. code to:

- Push the value 5 on the stack
- Push the addr. of the string "test" on the stack
- Push the current value of a on the stack
- Generate a call to the function f

Regular Procedure Calls

You write: `x = f(a, "test", 5)`

In compiling f, the compiler gen. code to:

- Push registers that will be clobbered on the stack to save the values
- Adjust the stack to make room for local and temporary variables
- Before a return, unadjust the stack, put the return data in a register, and issue a return instruction

Implementing RPC

No architectural support for RPC

Simulate it with tools we have (local proc. calls)

Simulation makes RPC a
language-level construct

Instead of an
operating system construct

Compiler creates code
to send messages to
invoke remote funcs

The OS gives us
sockets API

Implementing RPC

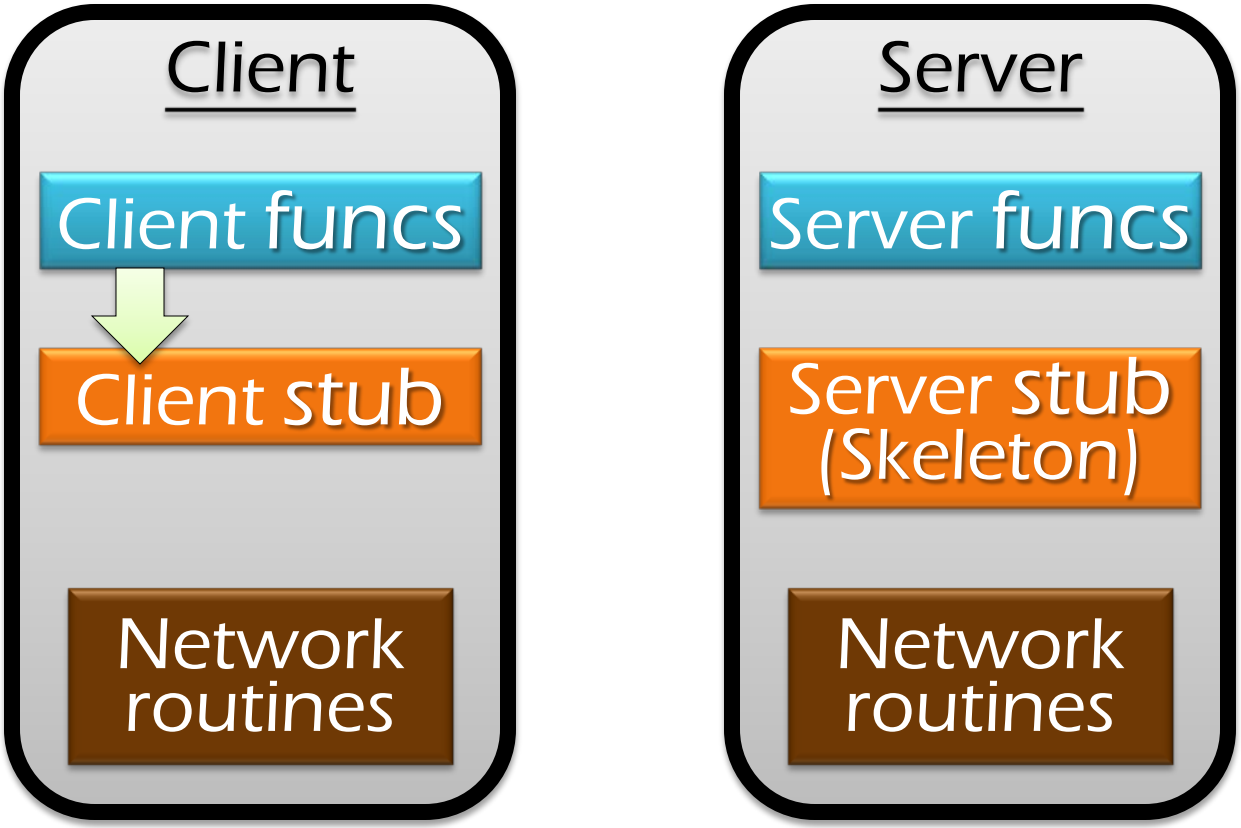
The **trick**:

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

The **stub** function contains the function's interface

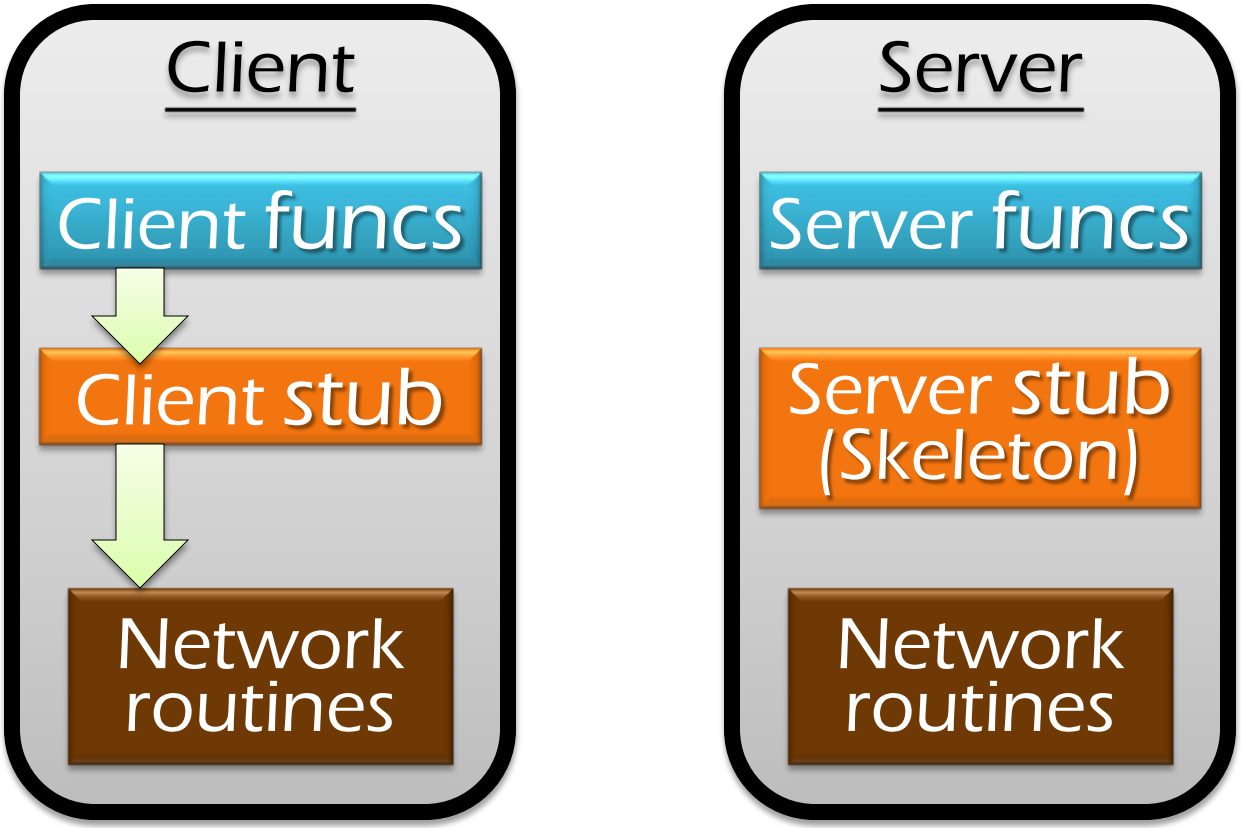
Stub Functions

1. Client calls **stub** (params on stack)



Stub Functions

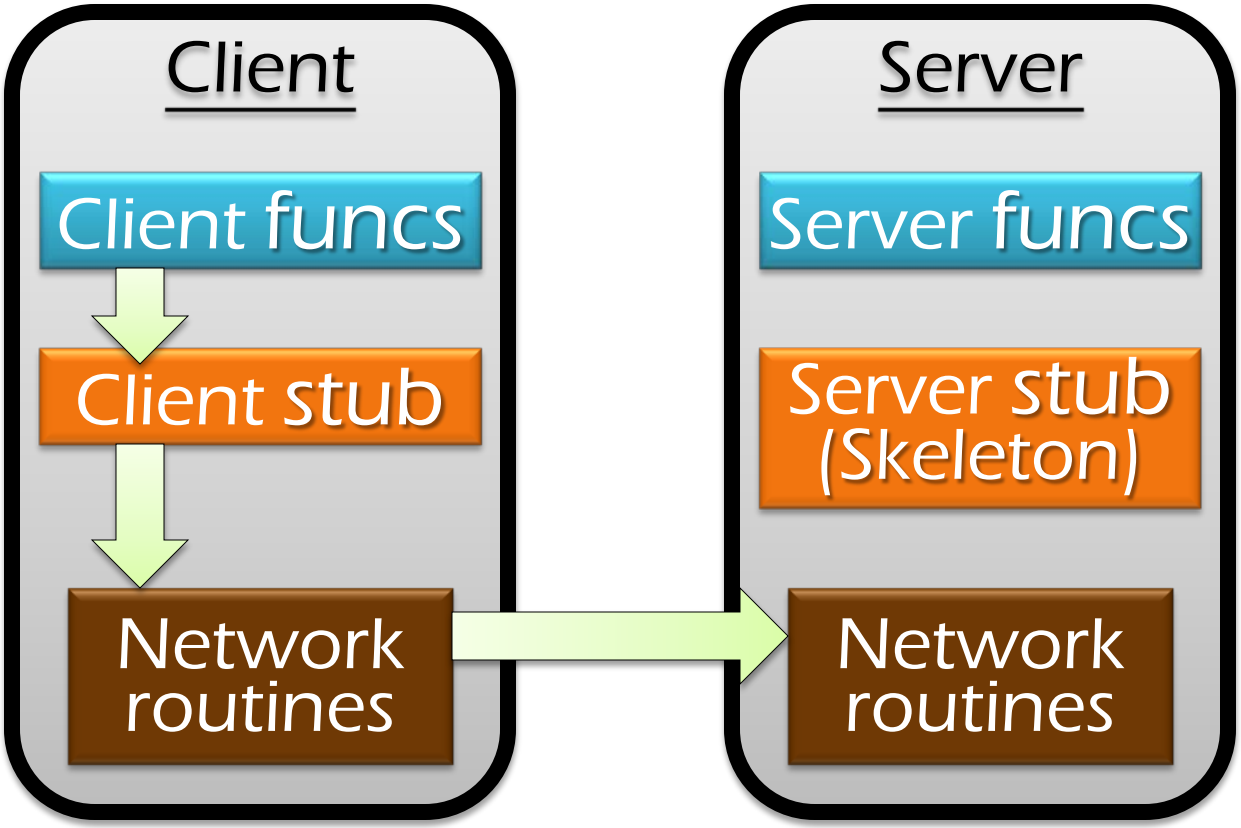
2. **Stub** marshals params to net **message**



Marshals = put data in a form suitable for transmission over a network

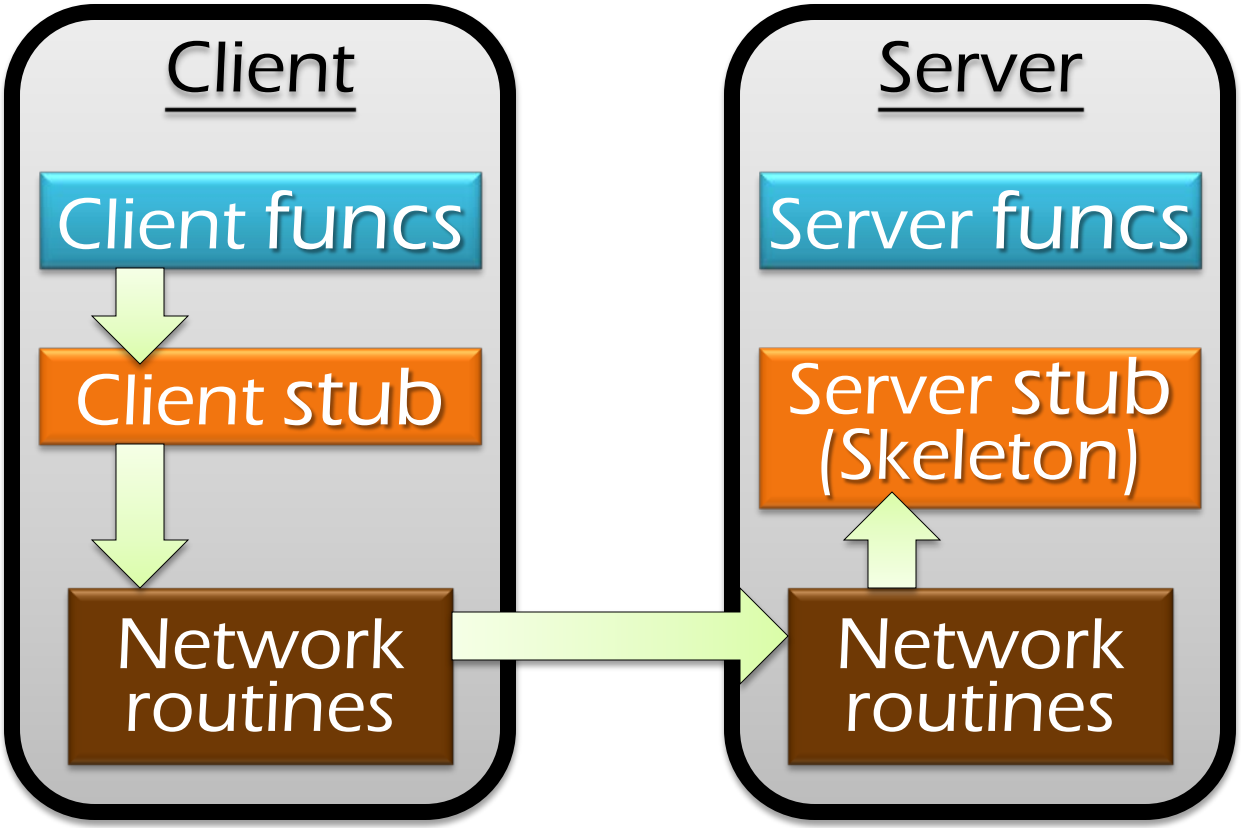
Stub Functions

3. Network routines sends **message** to server



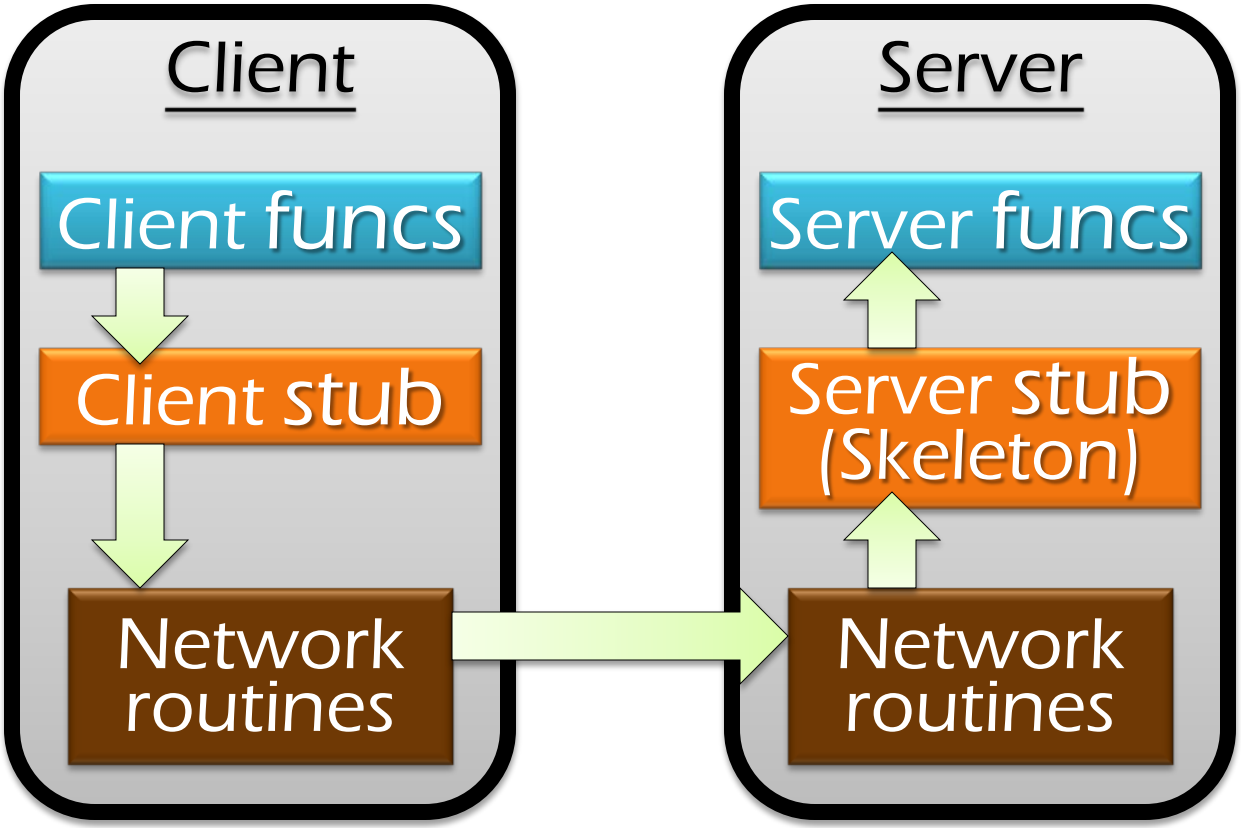
Stub Functions

4. Receive **message**: send it to server **stub**



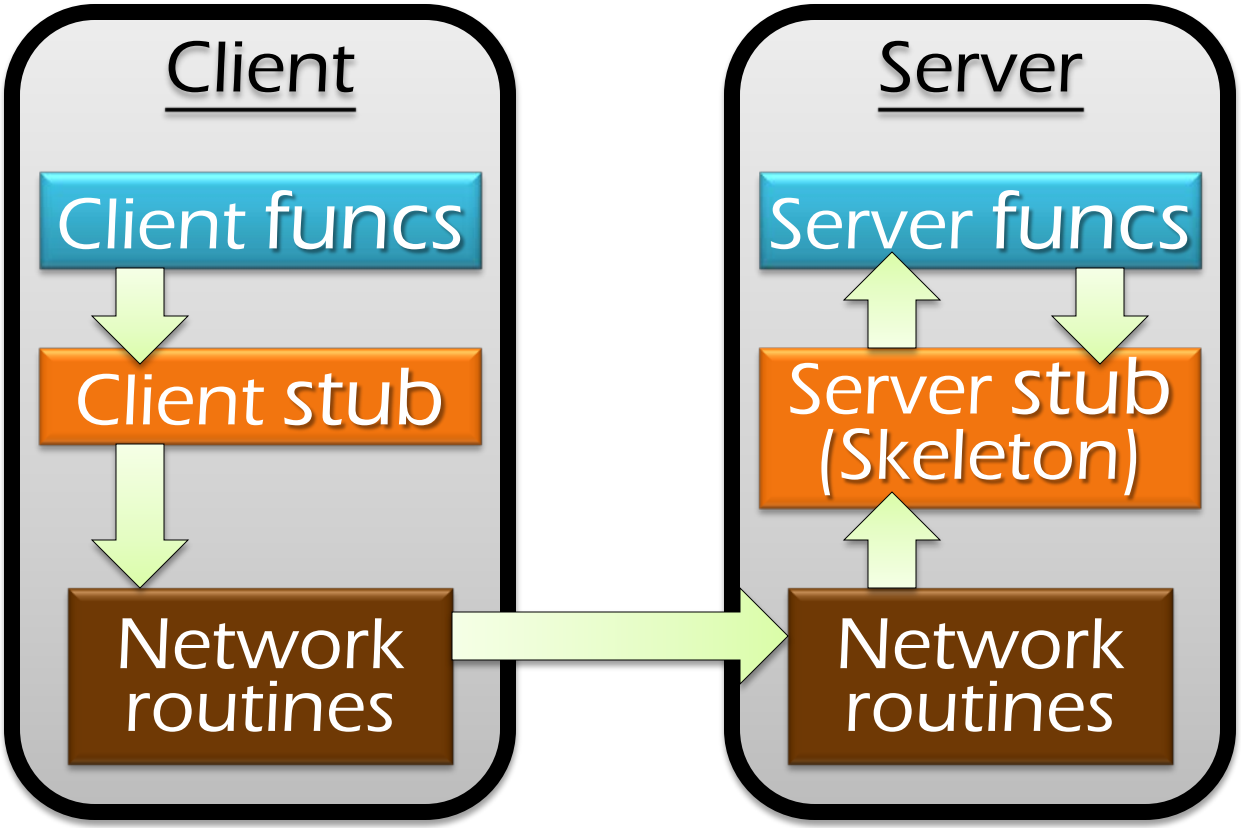
Stub Functions

5. Unmarshal parameters, call server function



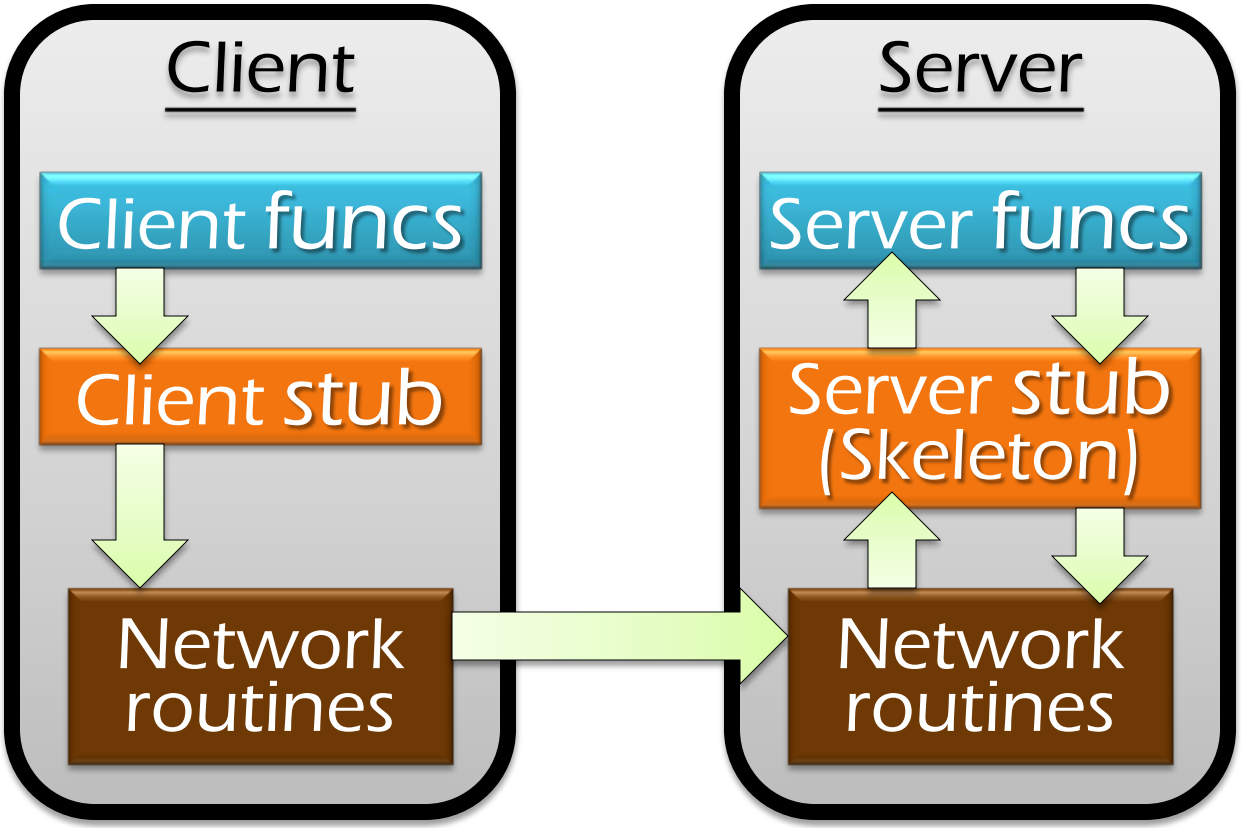
Stub Functions

6. Return from server function



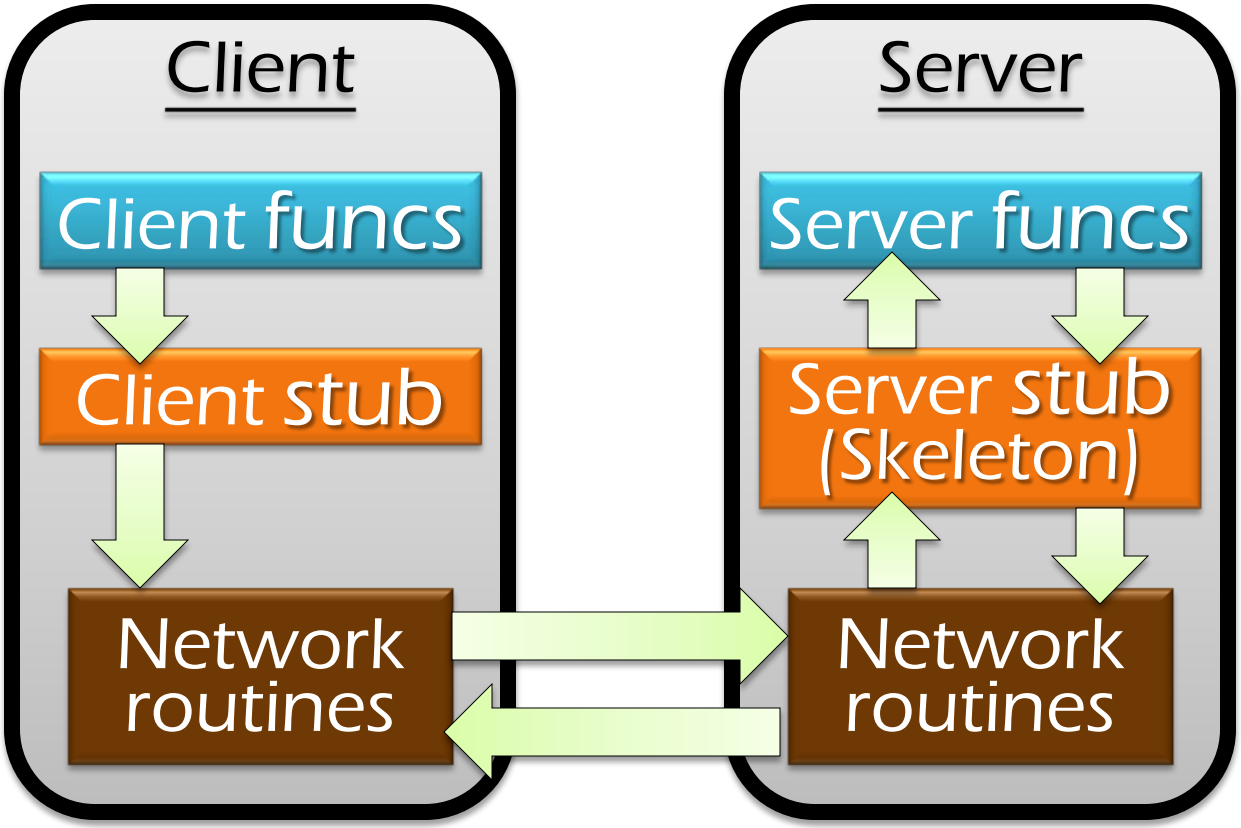
Stub Functions

7. Marshal return value and send message



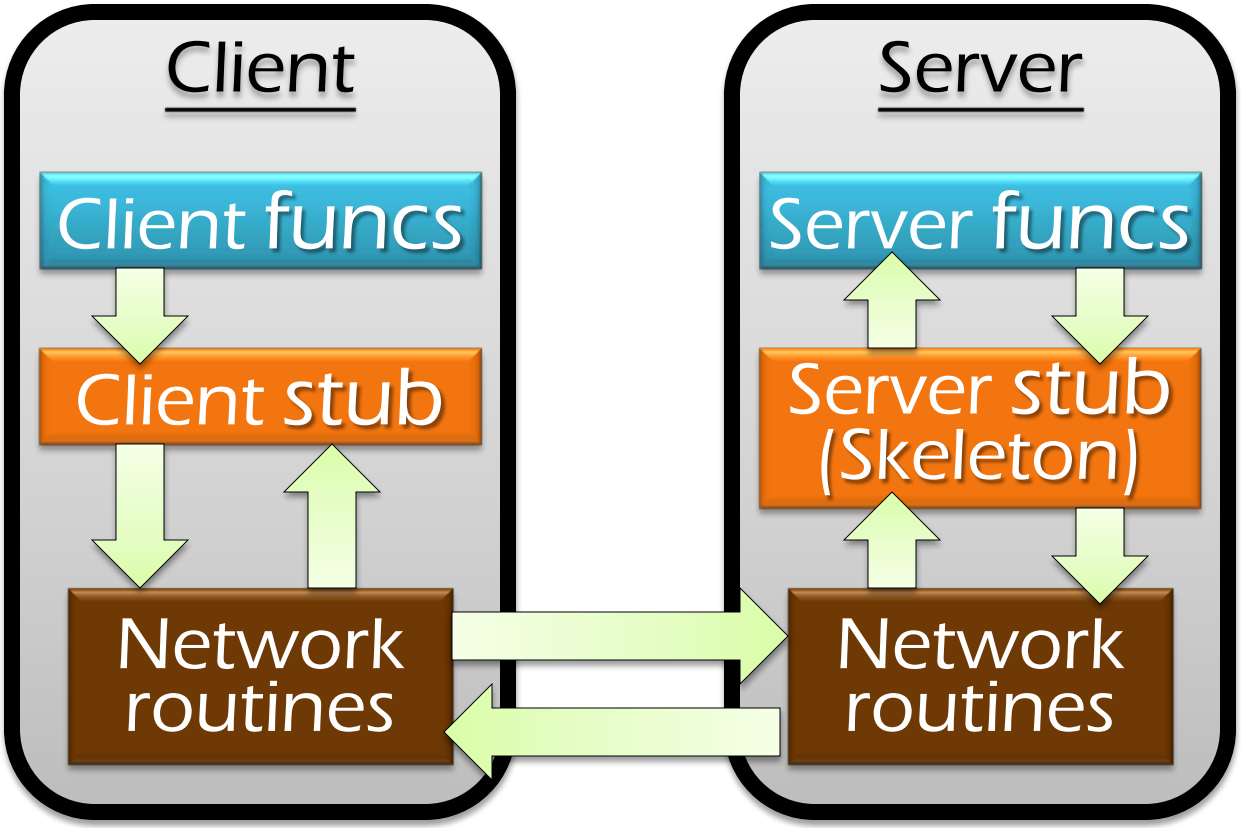
Stub Functions

8. Transfer **message** over network



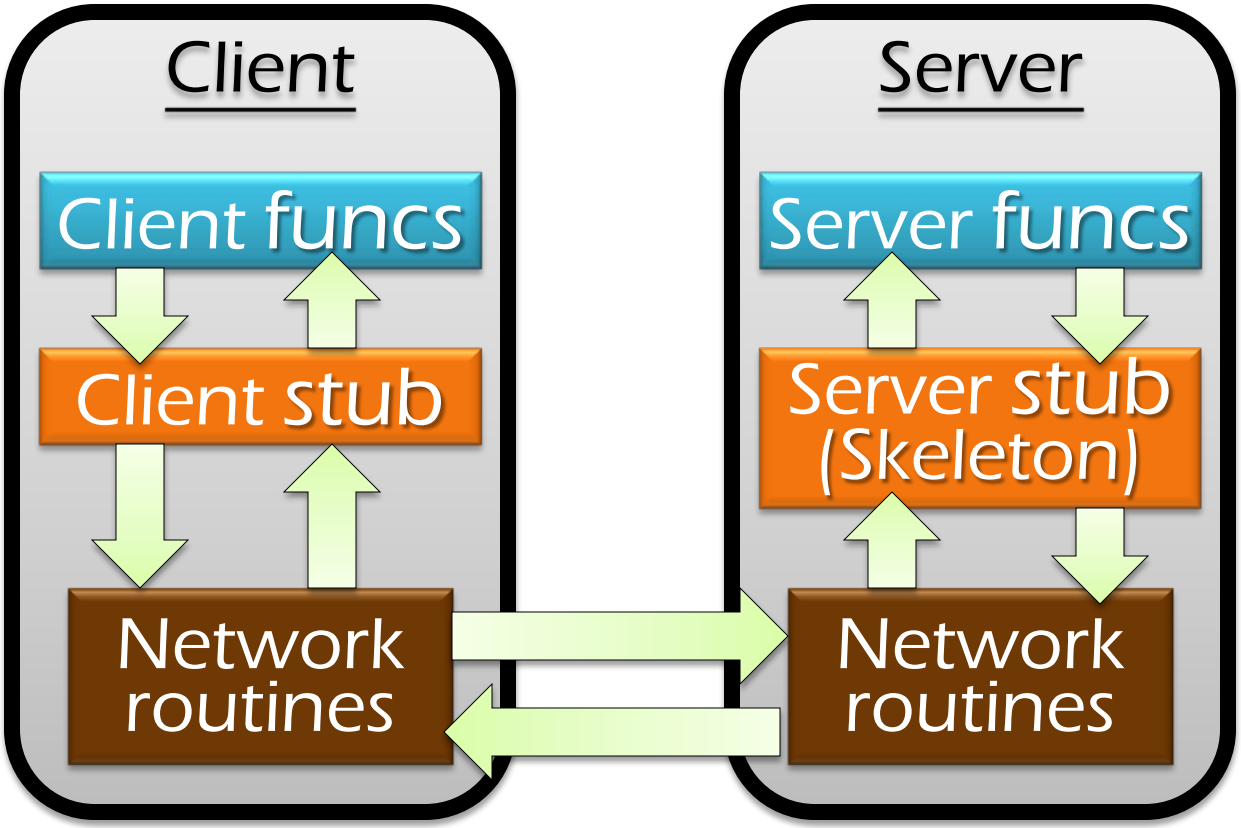
Stub Functions

9. Receive **message**: client **stub** is receiver



Stub Functions

10. Unmarshal return value, return to client code



Benefits

Procedure call interface

Writing application is simplified

- RPC hides all network code into stub function
- Application programmers don't have to worry about details
  - Sockets, port numbers, byte ordering

Where is RPC in the OSI model?

- Layer-5 Session layer: Connection management
- Layer-6 Presentation: Marshaling/data rep.

RPC has challenges

Parameter Passing

Pass by **value**

- Easy: just copy data to network message

Pass by **reference**

- Makes no sense without shared memory

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

Pass by Reference?

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
- Sizes of integers and other types
- Floating point representations
- Character sets
- Alignment requirements

Representing Data

Need standard **encoding** to enable comm. between **heterogeneous** systems

- Sun's RPC uses **XDR** (e**X**ternal **D**ata **R**ep.)
- **ASN.1** (ISO Abstract Syntax Notation)
- **JSON** (JavaScript Object Notation)
- Google **Protocol Buffers**
- W3C XML Schema Language

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, protocol buffers, JSON

Where to Bind?

Need to local host and correct server process

Solution 1:

Maintain **centralized** DB that can locate a host that provides a particular service

– Birrell & Nelson's 1984 proposal

Solution 2:

A server on **each** host maintains a DB of **locally** provided services

Transport Protocol

TCP or UDP: Which one should we use?

Some implementations may offer only one

- e.g., TCP

Most support several

- Allow programmer (or end user) to choose at runtime

When Things Go Wrong

Local procedure calls do not fail

- If they core dump, entire process dies

Most opportunities for error with RPC

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

A remote procedure call may be called:

- 0 time: server crashed or serve process died before executing server code
- 1 time: everything worked well, as expected
- 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
- At-most-once semantics

Simple retransmission leads to "at-least-once"

Birrell's RPC semantics:

- server says OK: executed once
- server says CRASH: zero or one times

much easier than exactly once, more useful than at-least-once

RPC Semantics

Understand application:

- Idempotent: may be run any number of times without harm
- Non-idempotent: those with side-effects

When at-least-once is ever OK?

- if no side effects (e.g., read-only operation)
- if app has its own plan for detecting duplicates

RPC Semantics

Question:

- What is semantics provided by init. code of YFS

weaker than at-least-once ☹

Example: send acquire, lost responds

→ time-out and re-send, then srv will wait forever!

Question+: just don't double-grant

send release, delay resp[1], re-send and received, then acquire again and now resp[1] delivered

→ incorrectly releases lock!

RPC Semantics

Exactly-once semantics:

- Server remember the requests it has **seen** and replies to **executed** RPCs (**across** **reboots**)
- Detect **duplicates**, reqs need **unique** IDs (XIDs)

Assumes: failures are **eventually** repaired and client retries **forever**

How to deal with server crash?

RPC Semantics

What if the server crash?

- Just before or just after call to handler()

After server restart:

- Can't tell whether handler() was called
- Client will re-send req (no resp. due to crash)
- Server must reply with CRASH for re-send

It is just at-most-once *semantics,*
we will return to this problem later in the course

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

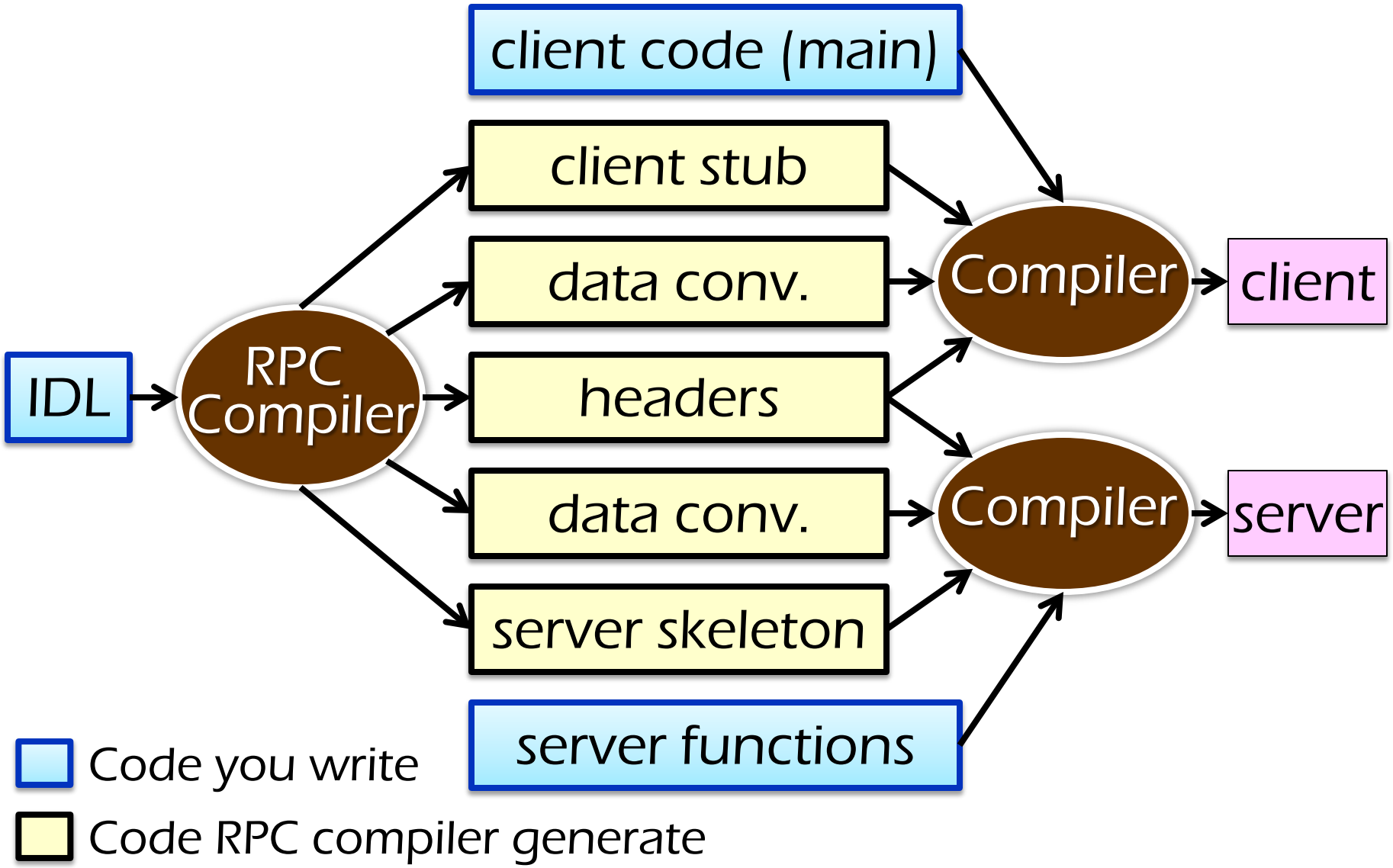
Allow programmer to **specify** remote procedure interface (*names*, *paras*, *return*)

Pre-compiler can use this to generate client and server **stub**

- Marshaling/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 service does an RPC system need?

Name service

- Export/lookup of binding info (ports, machines)
- Support dynamic ports

Binding operations

- Establish client/server communications using appropriate protocol (establish endpoints)

Endpoint operations

- Listen for reqs, export endpoint to name server

RPC API

What kind of service does an RPC system need?

Security operations

- Authenticate client/server

Internationalization operations (possibly)

Marshaling/data conversion operations

Stub memory management

- Dealing with “reference” data
- temporary buffers

Outline

RPC: Remote Procedure Calls

- Intro
- Implement
- Challenge
- Semantics

Lightweight RPC

Why LRPC

Most communication traffic in OS

- Between domains on the same machine
- Simple rather than complex

In conventional RPC systems

- Local comm. → remote communication
- Simple op. → complex operation

Employing RPC for cross-domain comm.

- Loss in performance
- Deficiency in structure

What is LRPC

A facility designed and optimized for comm. between domains in the **same** machine

Execution model is borrowed from protected procedure call (kernel trap)

Programming semantics and **large-grained protection model** are borrowed from RPC

Techniques of LRPC

Simple **control transfer**

- the client's thread executes the requested procedure in the server's domain.

Simple **data transfer**

- Shared argument stack and normal parameter passing mechanism

Simple **stubs**

- facilitating the gen. of highly optimized stubs

Design for **concurrency**

- Avoids shared data structure bottlenecks
- Benefits from speedup of multiprocessor

Control Transfer

Client bind to a server LPRC via kernel

Client → stub → kernel (trap)

- Arguments

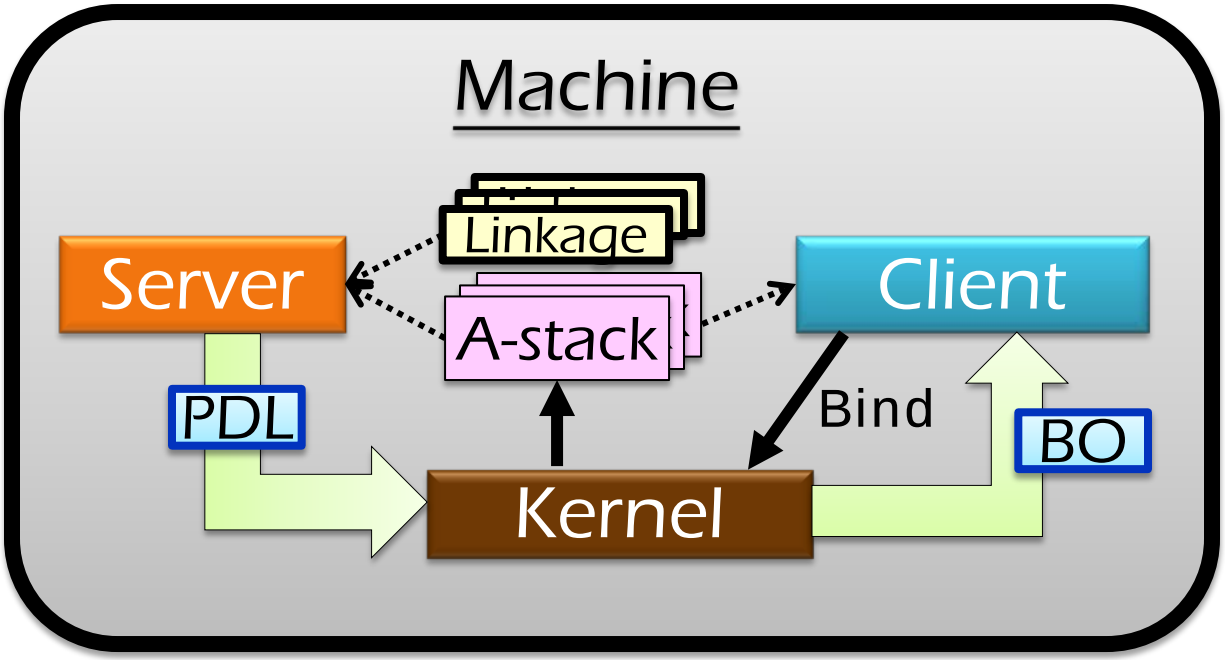
Kernel → server

- Validate
- Upcall to server

Server → kernel (trap) → stub

- Results

Bing LRPC



Calling LRPC

Client stub

- Manage A-stacks as a LIFO
- Take A-stack off the queue, push args on it
- Put the addr. of A-stack, BO and procedure identifier into registers
- Trap to kernel

Calling LRPC

Kernel

- Verify binding, procedure identifier, …
- Verify A-stack and locate corresponding linkage
- Save caller's ret-addr and %esp in the linkage
- prepare user stack and registers for server
- Perform upcall into the server's stub

Data Transfer

Arguments Copying

- RPC: 4 times

  client's stub → RPC message

  message → kernel (client)

  kernel (client) → kernel (server)

  kernel (server) → server's stub

- LPRC: 1 time

  client stub → shared A-stack

Stub Generation

RPC makes the cross-domain/machine calls transparent to lower level stubs

LRPC - simple

- kernel directly invoke server procedure
- only 1 formal procedure call (into client stub) and 2 returns (server procedure and client stub)

LRPC - complex

- Marshaling like conversional PRC

Corner Case

Transparency and Cross-Machine Calls

- Check in the first instruction of stub

A-Stacks: Size and Number

- Variable-sized arguments
- Out-of-band
- Pre-allocation for contiguous memory

Domain Termination

Summary

LRPC adopts a common-case approach to exploit simple control transfer, simple data transfer, simple stubs, and ...

LRPC performs well for the majority of cross- domain procedure calls

Next Lecture

Topic: Distributed Programming

Paper

Jeffrey Dean and Sanjay Ghemawat, "***MapReduce: Simplified Data Processing on Large Clusters***". Symposium on Operating Systems Design and Implementation (OSDI), **2004**

THANKS