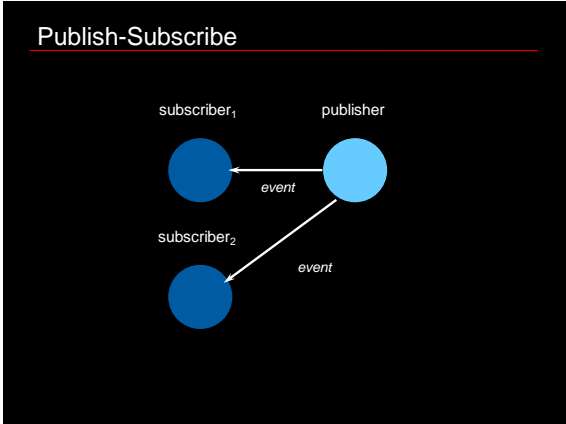
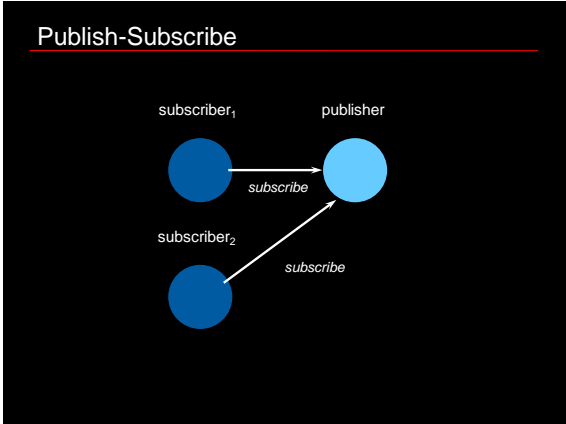
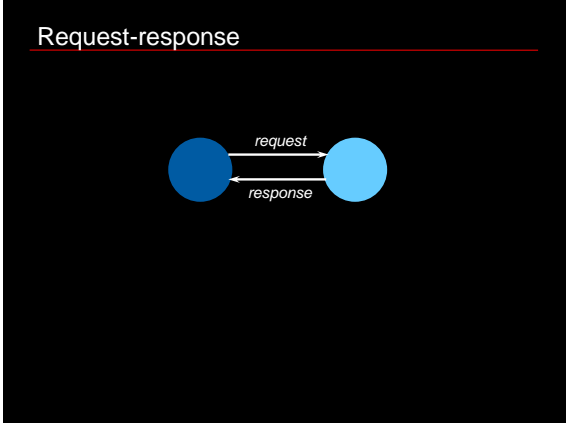
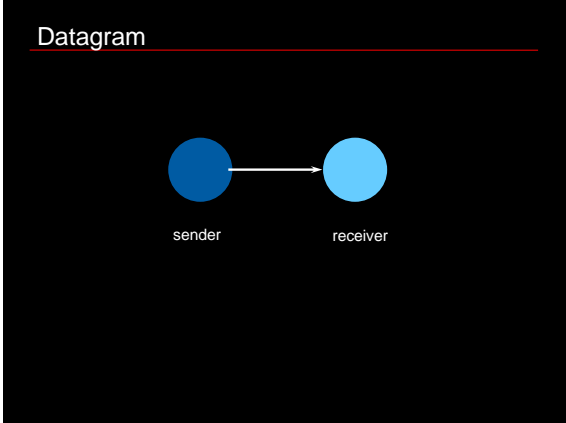
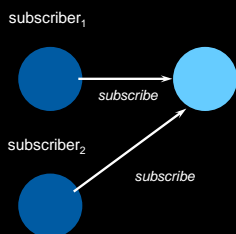


Distributed Systems  
 8. Remote Procedure Calls  
 Paul Krzyzanowski  
 pxk@cs.rutgers.edu

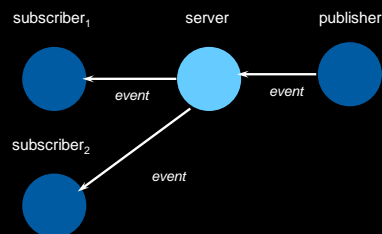
Communication models



### Publish-Subscribe with coordinator



### Publish-Subscribe with coordinator



Publisher does not have to be aware of how many apps are interested

### Publish-Subscribe Types

- List-based
  - List of topics (subjects) and their subscribers
- Broadcast-based
  - Broadcast to all listening nodes; nodes filter
- Content-based
  - Similar to list-based but match one or more message fields or sets of values

### 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

### RPC

1984: Birrell & Nelson

- Mechanism to call procedures on other machines

#### **Remote Procedure Call**

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

### Regular procedure calls

You write:

```
x = f(a, "test", 5);
```

The compiler parses this and generates code to:

- a. Push the value 5 on the stack
- b. Push the address of the string "test" on the stack
- c. Push the current value of a on the stack
- d. Generate a call to the function f

In compiling *f*, the compiler generates code to:

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

### 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**

The compiler creates code to send messages to invoke remote functions

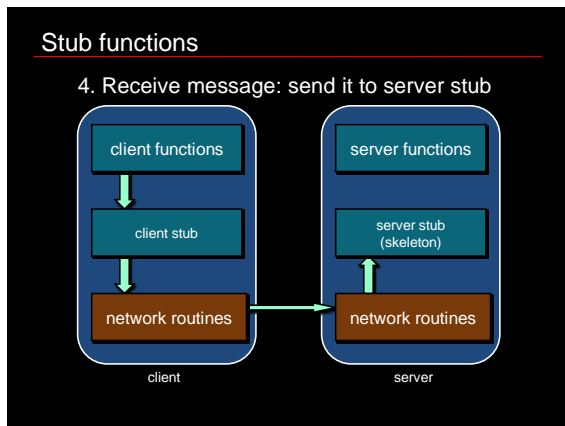
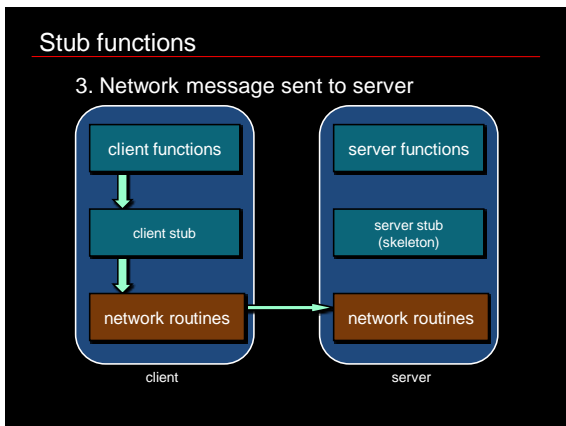
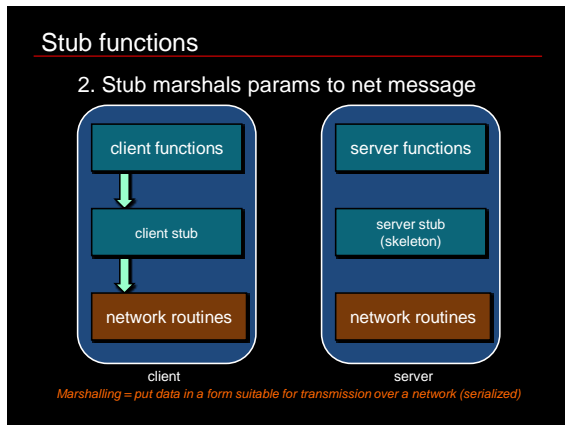
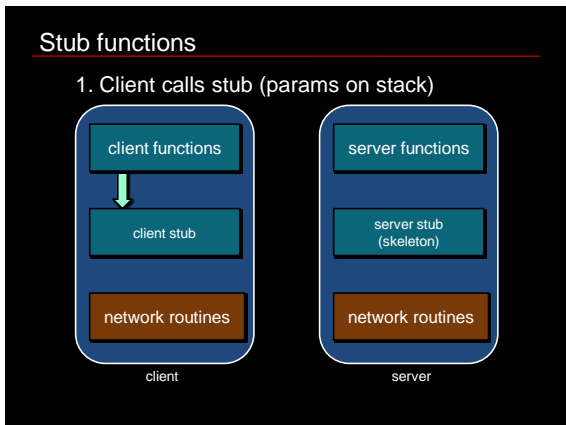
The OS gives us sockets

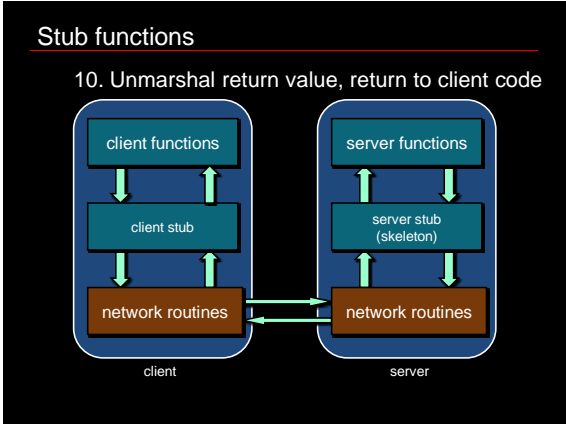
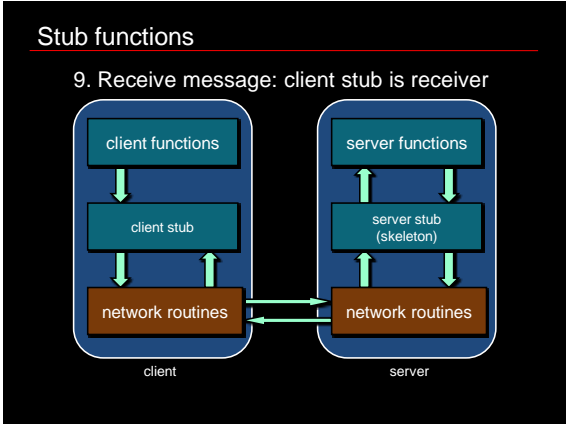
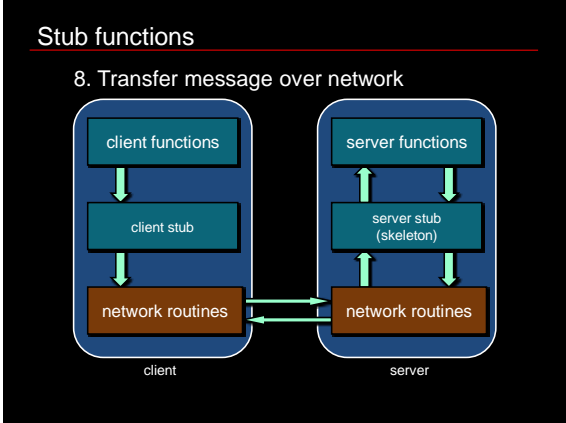
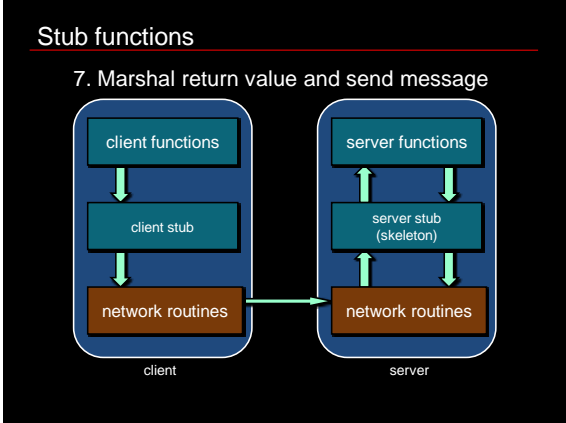
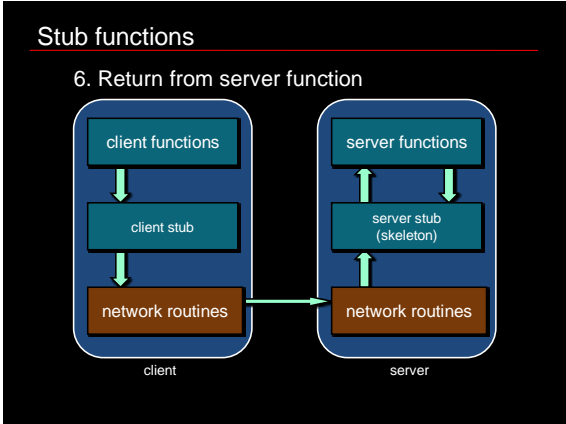
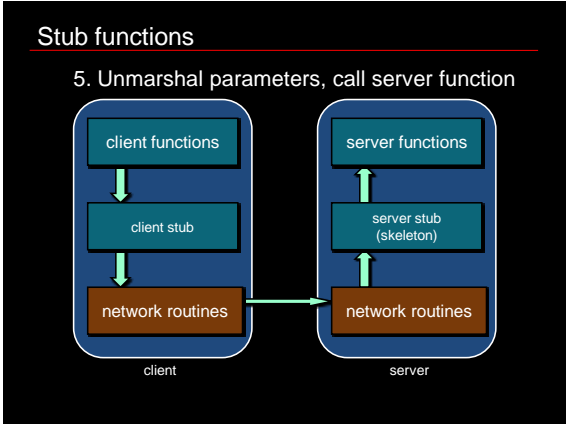
### Implementing RPC

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





## Benefits

- Procedure call interface
- Writing applications is simplified
  - RPC hides all network code into stub functions
  - 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: Marshalling/data representation

## RPC has challenges

## 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

- Big endian = most significant byte in low memory
  - Sparc, 680x0, IBM z/ series
- Little endian = least significant byte in low memory
  - Intel IA-32, Intel/AMD x86-64, Alpha
- Bi-endian = software selectable endianness
  - ARM, PowerPC, Sparc v9, MIPS, IA-64 (Intel Itanium)

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

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

Output with little endian:  
44, 33, 22, 11

Output with big endian:  
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)
- ASN.1 (ISO Abstract Syntax Notation)

## 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)*

## Where to bind? – Solution 2

---

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

Solution 1 is problematic for Sun NFS – identical file servers serve different file systems

## 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
- More 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 times: server crashed or server 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
  - or *at most once* semantics
- Understand application:
  - **idempotent** functions: may be run any number of times without harm
  - **non-idempotent** functions: those with 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

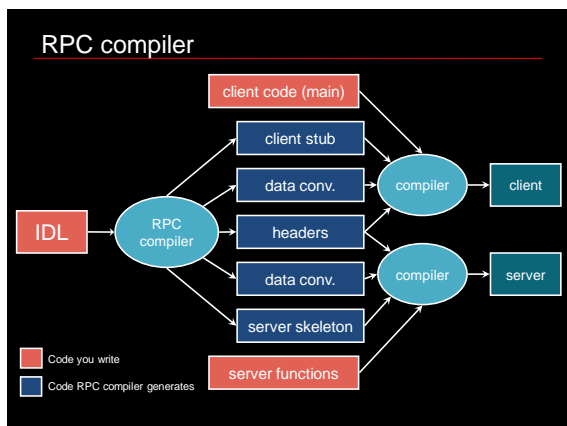
- Most programming languages (C, C++, Java, ...) have no concept of remote procedure calls
- Language compilers will not generate client and server stubs

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:
  - Marshalling code
  - Unmarshalling code
  - Network transport routines
  - Conform to defined interface
- Similar to function prototypes



### 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 of 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 (often the *main* program on the server)

### RPC API

What kind of services does an RPC system need?

- **Security operations**
  - Authenticate client/server
- **Internationalization operations** (possibly)
- **Marshalling/data conversion operations**
- **Stub memory management**
  - Dealing with "reference" data, temporary buffers
- **Program ID operations**
  - Allow applications to access IDs of RPC interfaces
  - Can you pass references to remote functions to other processes?

The End