CSci 5105

Introduction to Distributed Systems

Communication: RPC

Today

- Remote Procedure Call
- Chapter 4 TVS

Last Time

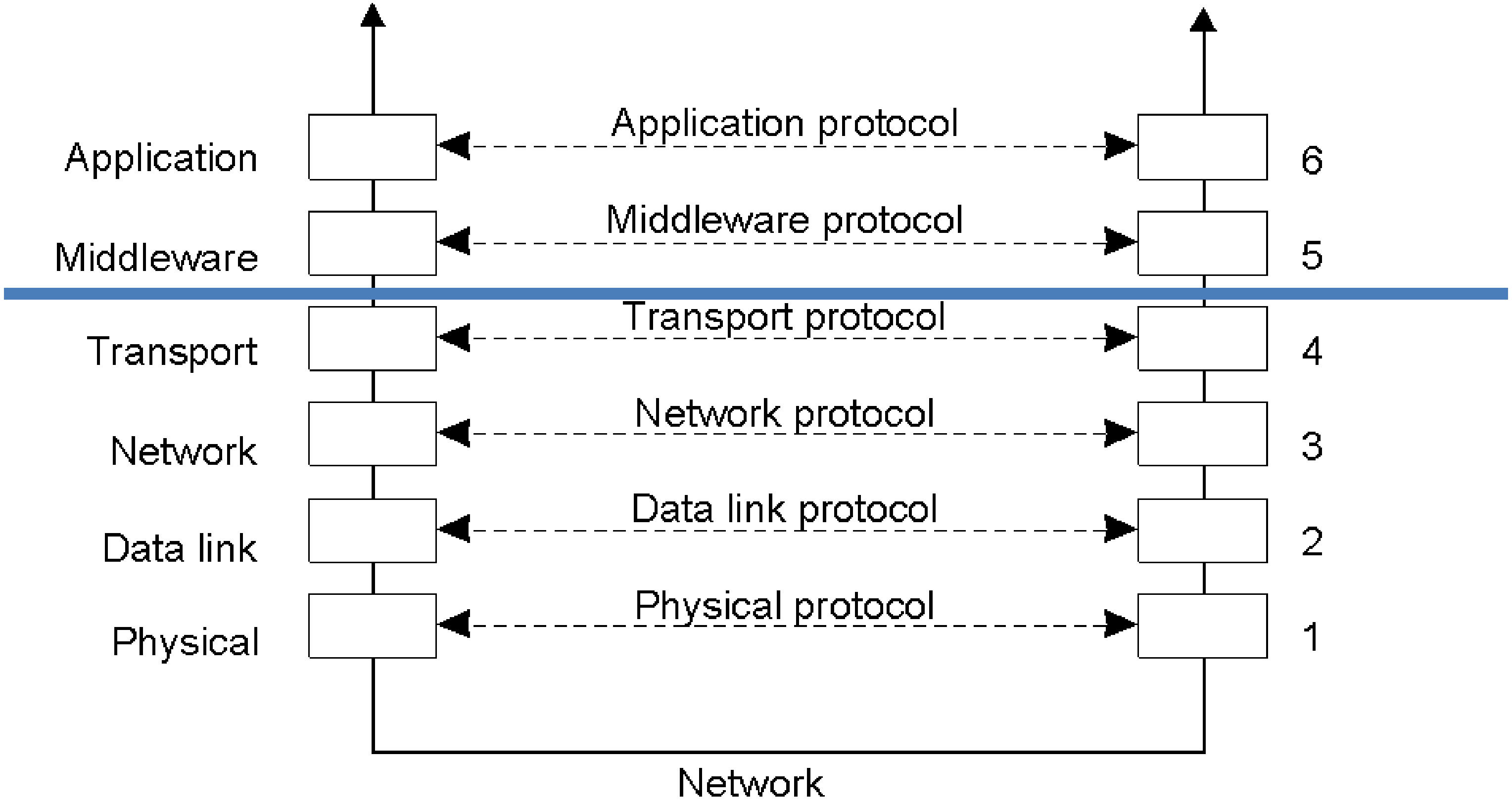
- Architectural styles

- RPC “generally” mandates client-server but not always

Interprocess Communication

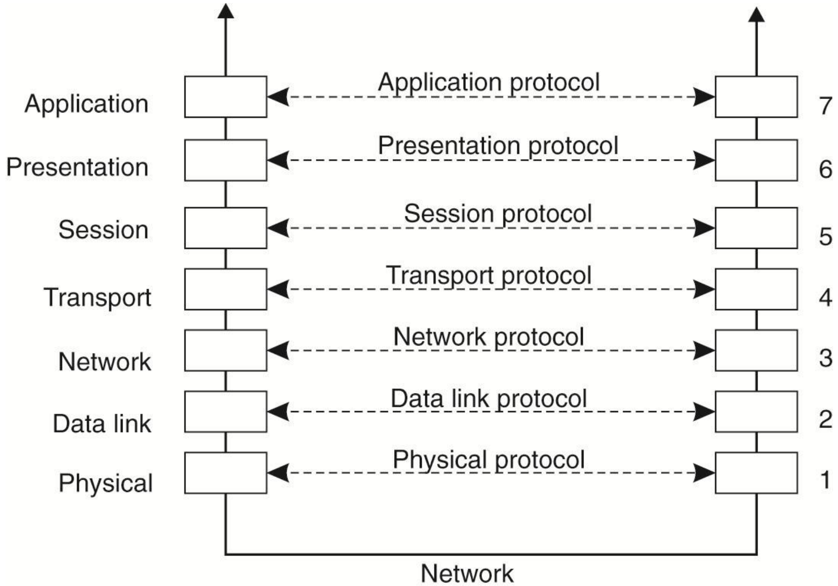
- Building block for centralized, network, and distributed OS
- Low-level primitives
  - shared-memory: same machine, 4061, message-passing: 4061/5103

Layering


Application
Middleware
Transport
Network
Data link
Physical
Application protocol
Middleware protocol
Transport protocol
Network protocol
Data link protocol
Physical protocol
6
5
4
3
2
1
Network

Protocol

- In force at every layer
- Protocol Details
  - data format <GET> <PATH> <VERSION>
  - exchange sequence
  - both sides must speak same "language"



Transport Layer

- TCP
  - connection-oriented; reliable 2-way stream, (http, ftp, …) built on it
- UDP
  - connectionless, unreliable (corruption, collisions, buffer space, out of order arrival, …)
  - every packet carries destination address

both layered on IP (network layer)

Message Passing

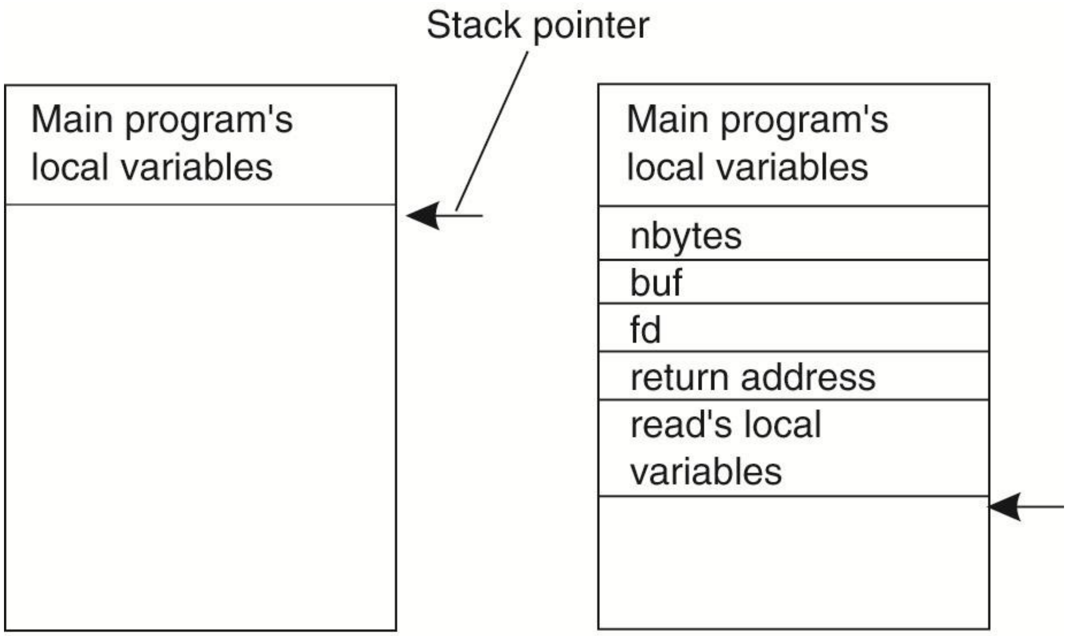
- sender/receiver in different address space (same or different machine)
- send: user msg (**contiguous**) copied into OS network buffer, then OS sends it
- receive: msg received from network by OS into network buffer, then copied to user msg
- blocking vs. non-blocking
- Low-level: addresses, special primitives

Higher-level IPC

- Hide more details …
- Remote procedure call (RPC)
- Message-bus
- Events
- Streams

Remote Procedure Call

- Let’s look at a regular procedure call


Stack pointer
Main program's local variables
Main program's local variables
nbytes
buf
fd
return address
read's local variables

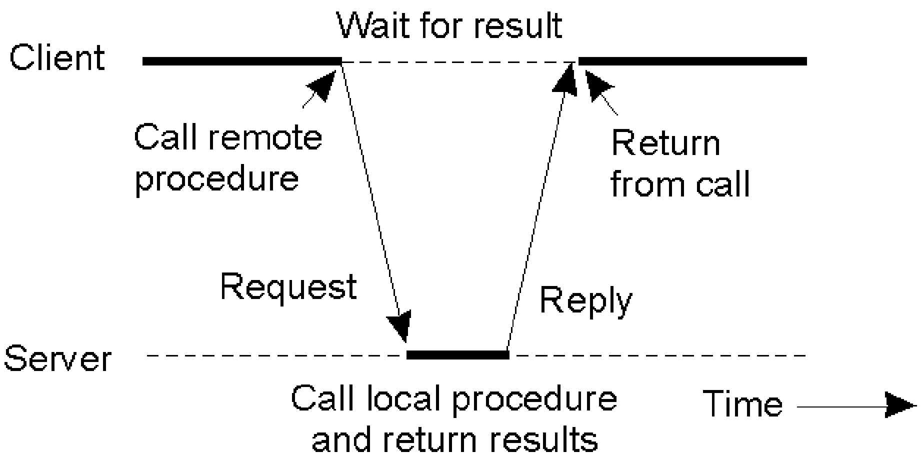
Key idea: use a stack to transfer information; same address space

Remote Procedure Call

- Retain procedure call “look and feel”
  - send/receive are too low level: no transparency

```
    main () {
          send (dest_addr, …); …}
```

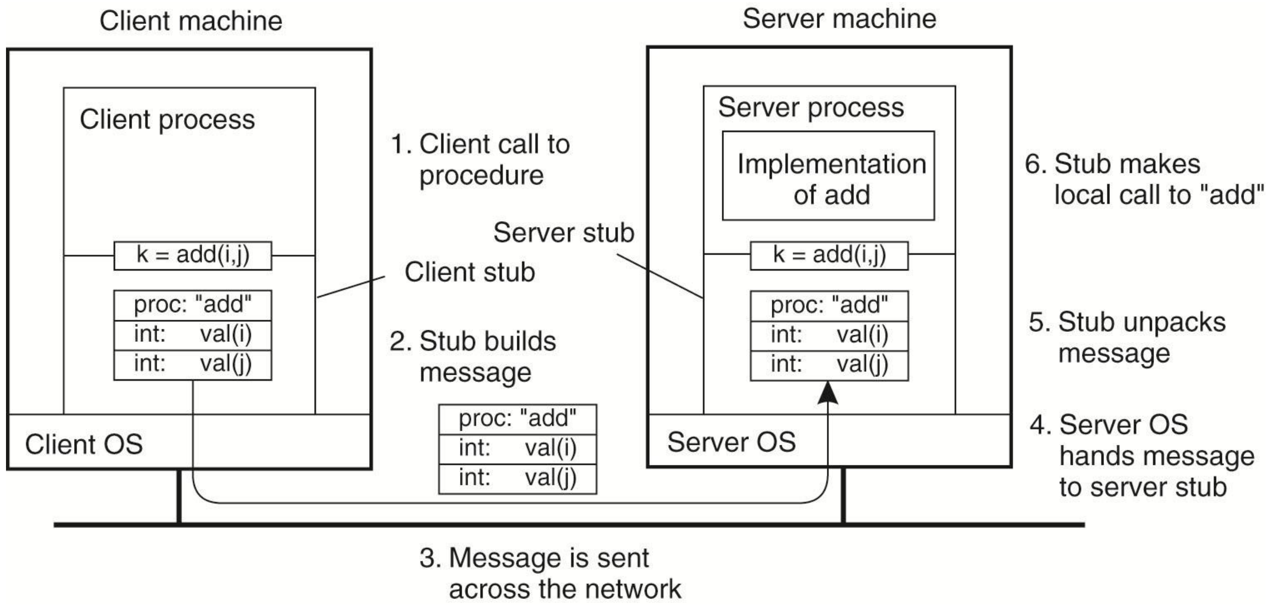
- Synchronous: request-reply communication



Remote Procedure Call

- Transparency

```
do_RPC (add, param1, param2)
add (param1, param2)
```



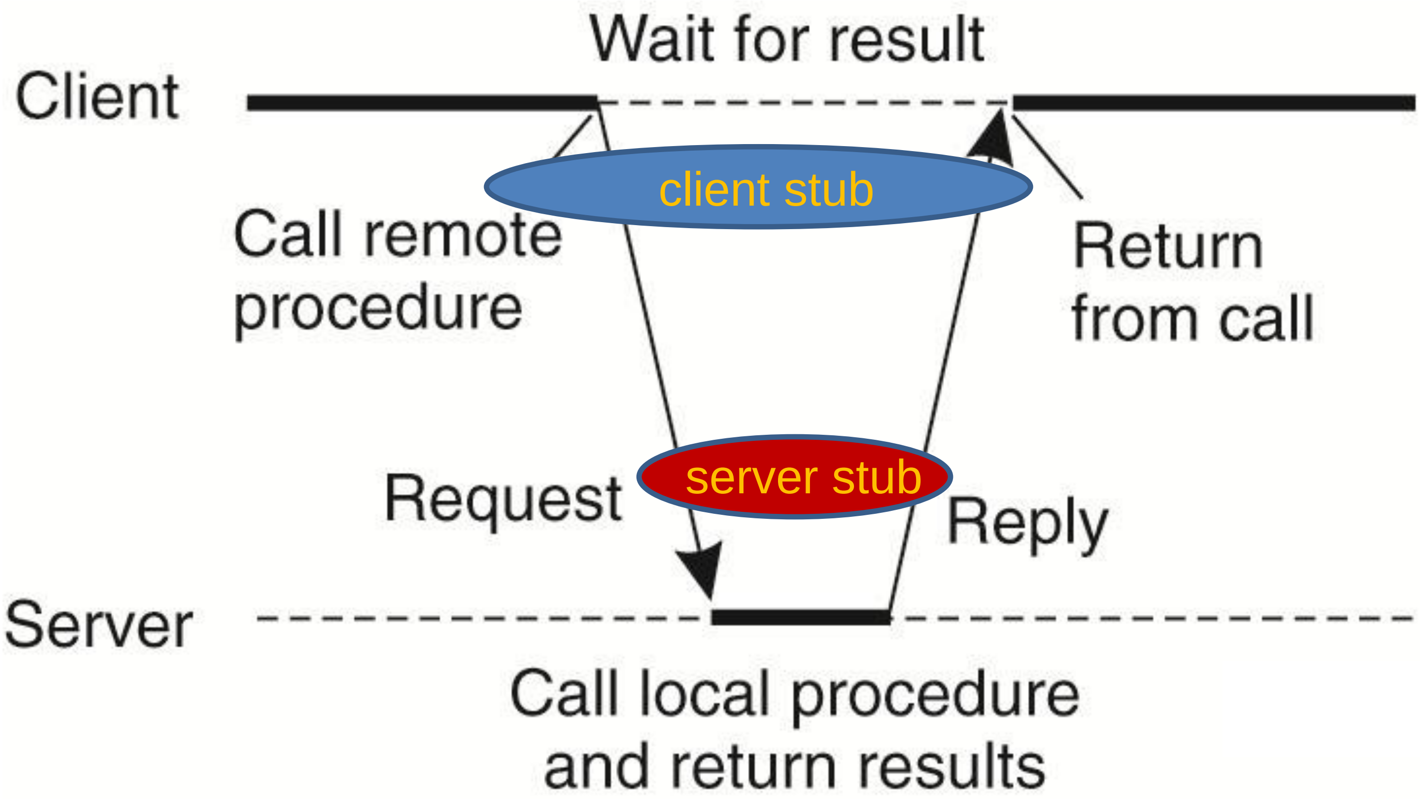
Remote Procedure Call

- RPC: procedure can be on same or different machine (diff. address space)
  - no shared stack/memory
- Client and server stubs are needed for transparency
  - `add` must be a local procedure … why?
  - `add` is a stub … stubs within both client and server processes

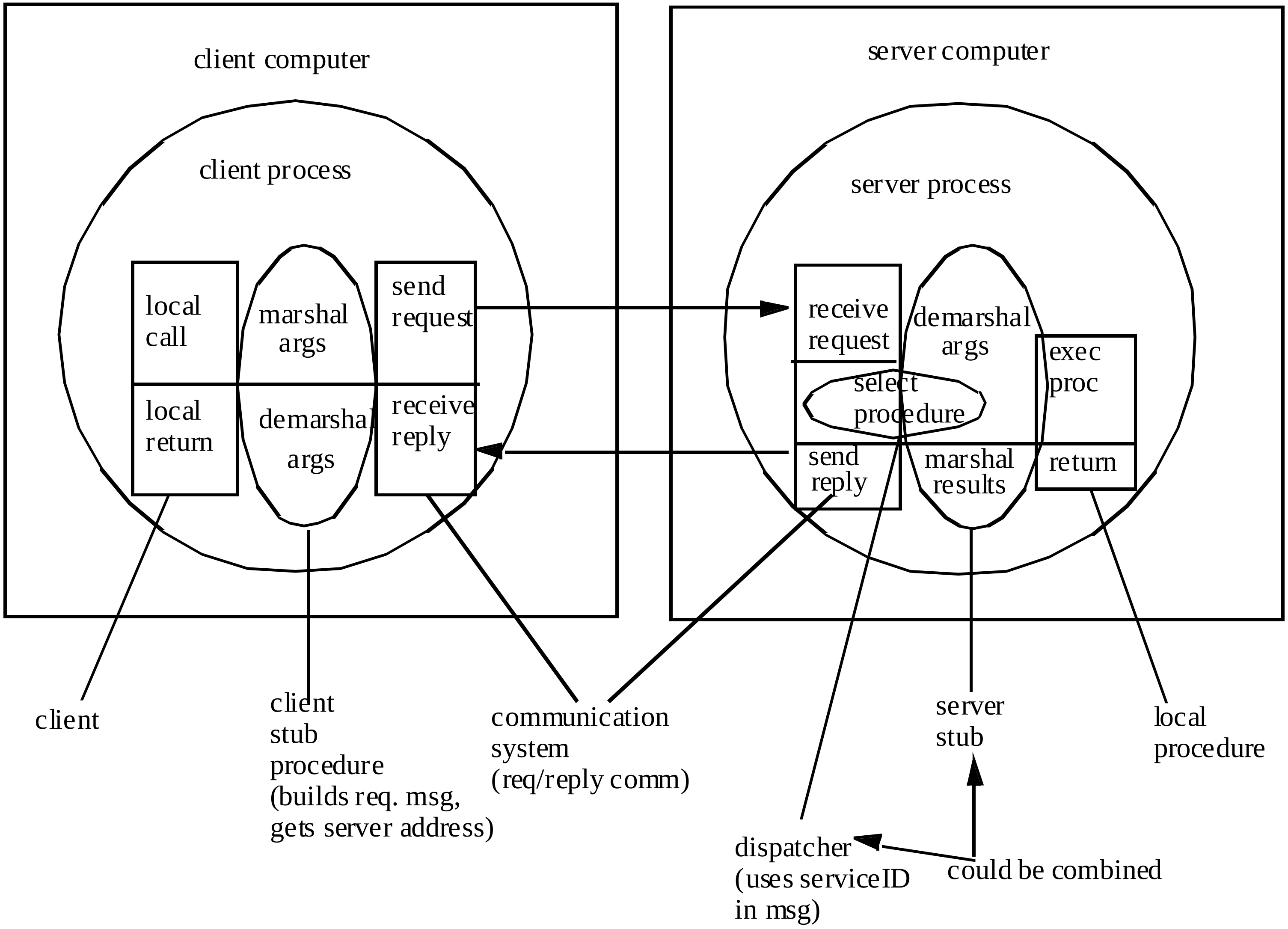
Binding

- Now does stub know/obtain server name/address?
  - full discussion of binding issue is deferred until naming discussion

Client and Server Stubs



The Plumbing



RPC Recipe: 10 step program

1. Client procedure calls client stub in normal way
2. Client stub builds message, calls local OS
3. Client's OS sends message to remote OS
4. Remote OS gives message to server stub (skeleton)
5. Server stub unpacks parameters, calls server
6. Server does work, returns result to the stub
7. Server stub packs it in message, calls local OS
8. Server's OS sends message to client's OS
9. Client's OS gives message to client stub
10. Stub unpacks result, returns to client

Marshaling

- Needed to ensure send is contiguous
- Encoded remote procedure and flatten arguments; must know sizes
- Decode on the server side
- Issues?

Transparency

- RPCs are transparent to the caller
  - Is this really true. In what ways are transparency violated?

RPC Interface

- Who generates the stubs?
- Integrated in host programming language
  - inherits semantics of PL, greater transparency, but client/server written in same PL
    - Java RMI
- Compiler generated stubs: IDL
  - IDL (signatures) compiled into native languages: client/server can be written in different PL
    - Sun RPC

Failure

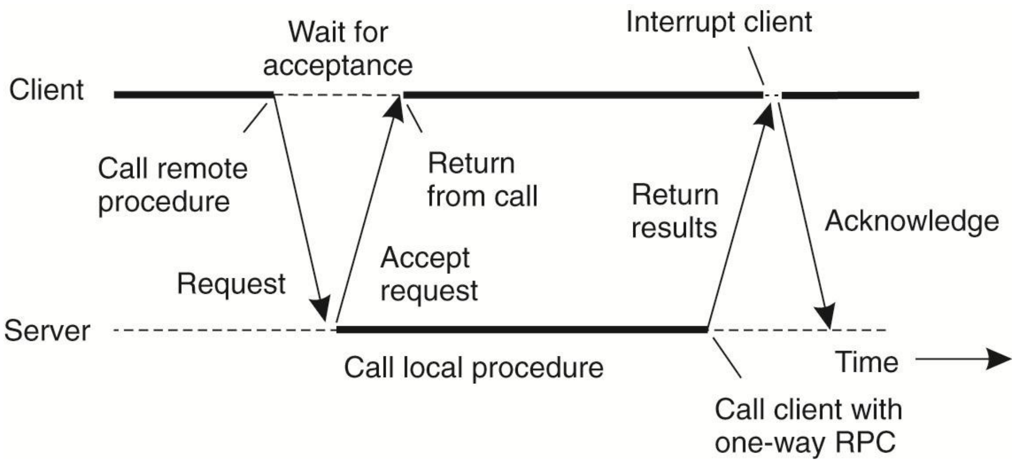
- Maybe-call: no guarantees; caller can’t tell if remote procedure was executed
  - was request or reply lost?
- At-least-once: client will continually try
  - no duplicate filtering – request may execute multiple times: issues?
- At-most-once:
  - filter duplicates
- Idempotent server: the effect of executing a request N times is the same as 1 time

RPC Optimizations

- Asynchronous RPC
  - no return value/1-way RPC -> don’t wait at all, possible wait for server acceptance of request
  - examples?

Asynchronous RPC

- Solution 1. Two RPCs



- Solution 2. Deferred blocking

```
A = op_1 (….);
// some code that doesn’t need A;
// now I need A, block somehow
B = A + 1;
```

Q: What is the price of this kind of optimization?

RPC Optimizations

- Chaining
  - A = op_1 (B, op_2 (X)); // no need for op_2 to return its value to main program
- Light-weight RPC
  - most RPCs are actually done to server on same machine! , e.g. Window Manager
  - Most RPCs are very small (< 200 bytes)
  - optimize this case: LRPC or upcalls
- Clumping
  - combine RPCs from same client to same server

Object-based RPC

- Objects
  - state, methods, and interface
  - passive objects (C++ or Java objects), basically just data-structures+operations
  - these objects have no thread of control
- Distributed/Active/Remote object
  - has a thread of control and two parts
  - interface proxy at the client side and the actual object at the server side

Object-based RPC

- Client binds to a distributed object
  - proxy gets loaded into client address space
  - statically or dynamically
- Compile-time vs. runtime objects
  - compile-time: language-specific C++/Java
  - runtime: any language, use a "wrapper" (object adapter)

Comparison

- Object-based vs. Procedural RPC
  - object reference is a first-class entity – can be passed – stubs cannot!
  - why do this? issues?

Next Time

Next topic: Java RMI, Unix RPC

Read COM papers on line