CS 153
Design of Operating Systems

Winter 2016

Lecture 23: Inter-Process Communication (IPC) and Remote Procedure Call (RPC)

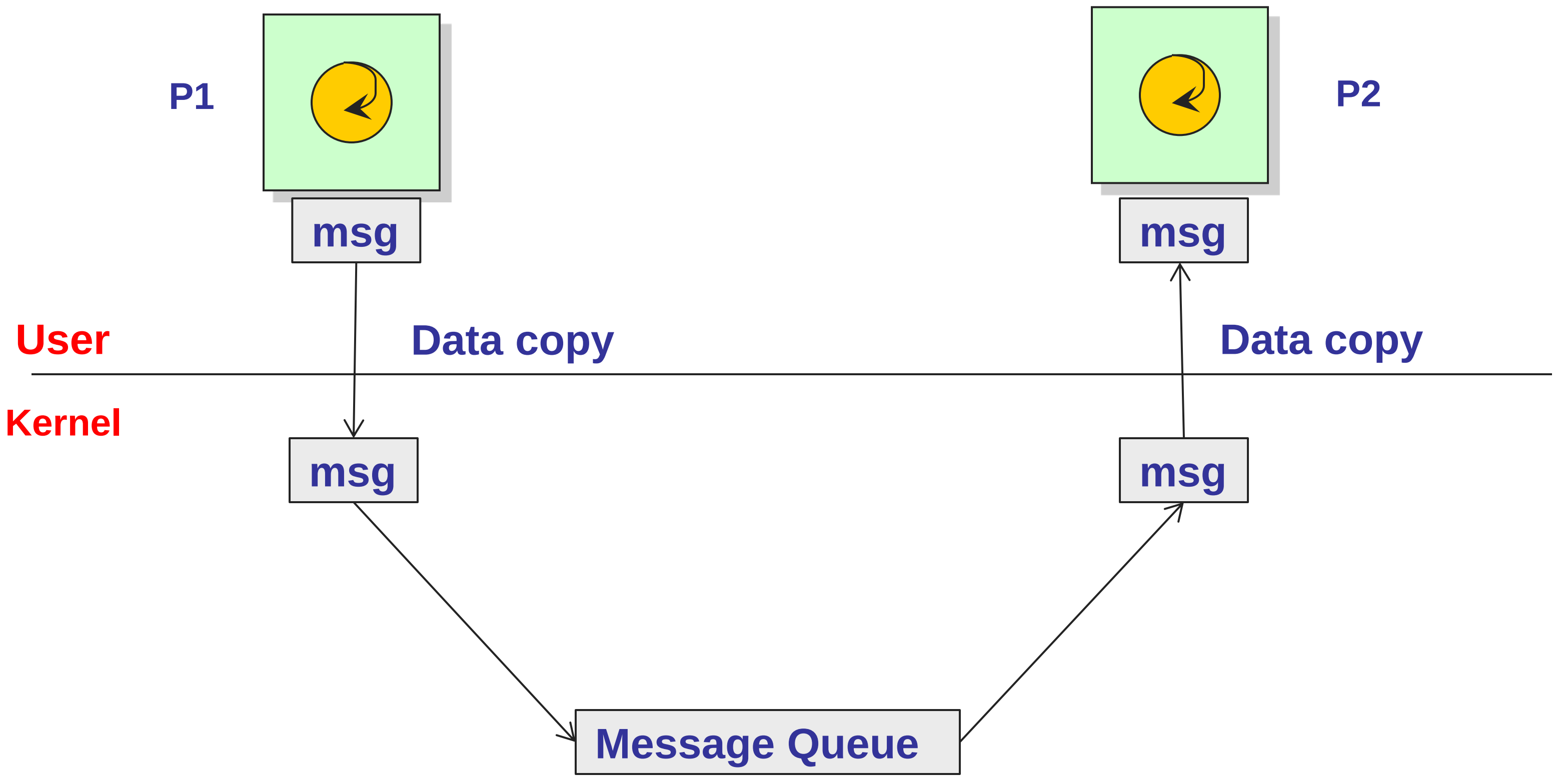
Inter-Process Communication

- **Exchange of data** between two or more separate, independent processes.
- Operating systems provide facilities/resources for inter-process communications (**IPC**), such as message queues, semaphores, and shared memory.
- Similar concept exists for network communications (processes that reside on different physical hosts).

Inter-Process Communication

- Two main types
  - Message queues (heavy kernel involvement)
  - Shared memory (discussed earlier, minimal kernel involvement)

Message Queues



Message Queues

Two basic operations:

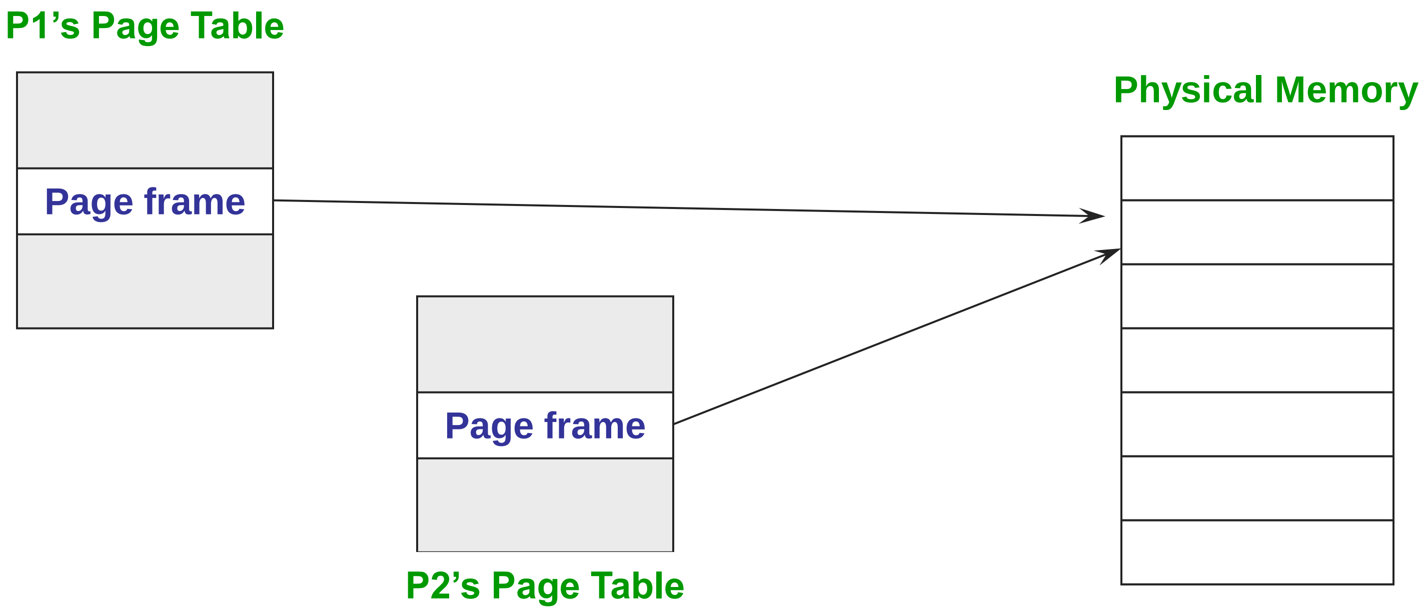
send(message) and **receive**(message)

- message contents can be anything mutually comprehensible
  - » data, remote procedure calls, executable code etc.
- usually contains standard fields
  - » destination process ID, sending process ID for any reply
  - » message length
  - » data type, data etc.

- UNIX
  - System V Message Queues
  - UNIX domain socket, local TCP/UDP socket

Shared Memory

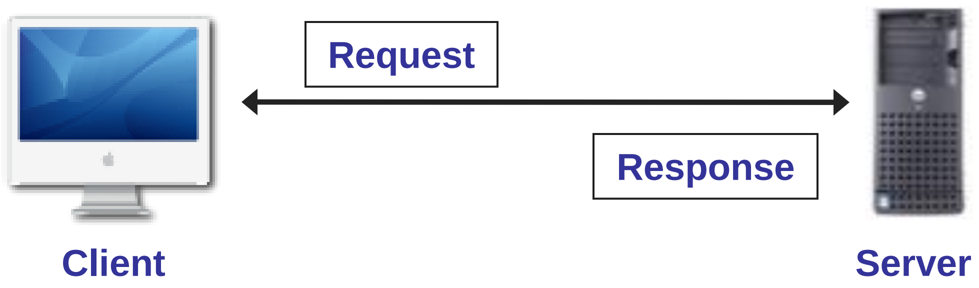
- Once page table mapping setup, no syscall or additional copying necessary



Network Communication

- By in large similar to IPC in terms of the interface
- **send**(message) and **receive**(message)
  - message contents can be anything mutually comprehensible
    - data, remote procedure calls, executable code etc.
  - usually contains standard fields
    - destination server IP, sending client IP
    - message length
    - data type, data etc.

Client-Server Communication



1. Client connects to the server (locates it and establishes a connection to it)
2. Client sends a request to the server
3. Server performs some action
4. Server sends a response back

OS Support for Network

- OS includes implementations of network protocols
  - For example: TCP, UDP, ICMP, etc.

- How should applications use these protocols for network communication?
  - Open network connection to the server
  - Hand-code messages to send requests and receive responses
  - For example, request/response for weather service:
    - » *(Date: 01/27/2014, City: Riverside, State: CA)*
    - » *(Temperature: 70, Chance of rain: 20%)*

Messages: A Bad Abstraction

- Hand-coding messages gets tiresome
  - Need to worry about message formats
  - Have to pack and unpack data from messages
  - Servers have to decode and dispatch messages to handlers
  - Messages are often asynchronous

- Messages are not a very natural programming model (still heavily used nevertheless)
  - Think about web browsing

Procedure Calls

- Procedure calls are a more natural way to communicate
  - Every language supports them
  - Semantics are well-defined and understood
  - Natural for programmers to use
- Idea: Have servers export a set of procedures that can be called by client programs
  - For example: *GetWeather(Date, City, State)*
  - Similar to module interfaces, class definitions, etc.
- Clients just do a procedure call as if they were directly linked with the server
  - Under the covers, the procedure call is converted into a message exchange with the server

Remote Procedure Calls

- So, we would like to use procedure call as a model for distributed (remote) communication
- Remote Procedure Call (RPC) is used both by operating systems and applications
  - NFS is implemented as a set of RPCs
  - DCOM, CORBA, Java RMI, etc., are all basically just RPC
- Lots of issues
  - How do we hide the details from the programmer?
  - What are the semantics of parameter passing?
  - How do we (locate, connect) to servers?
  - How do we support heterogeneity (OS, arch, language)?
  - How do we make it perform well?

RPC Model

- A server defines the server's interface using an interface definition language (IDL)
  - The IDL specifies the names, parameters, and types for all client-callable server procedures
- A stub compiler reads the IDL and produces two stub procedures for each server procedure (client and server)
  - The server programmer implements the server procedures and links them with the server-side stubs
  - The client programmer implements the client program and links it with the client-side stubs
  - The stubs are responsible for managing all details of the remote communication between client and server

RPC Example

Server Interface:

```
int Add(int x, int y);
```

Client Program:

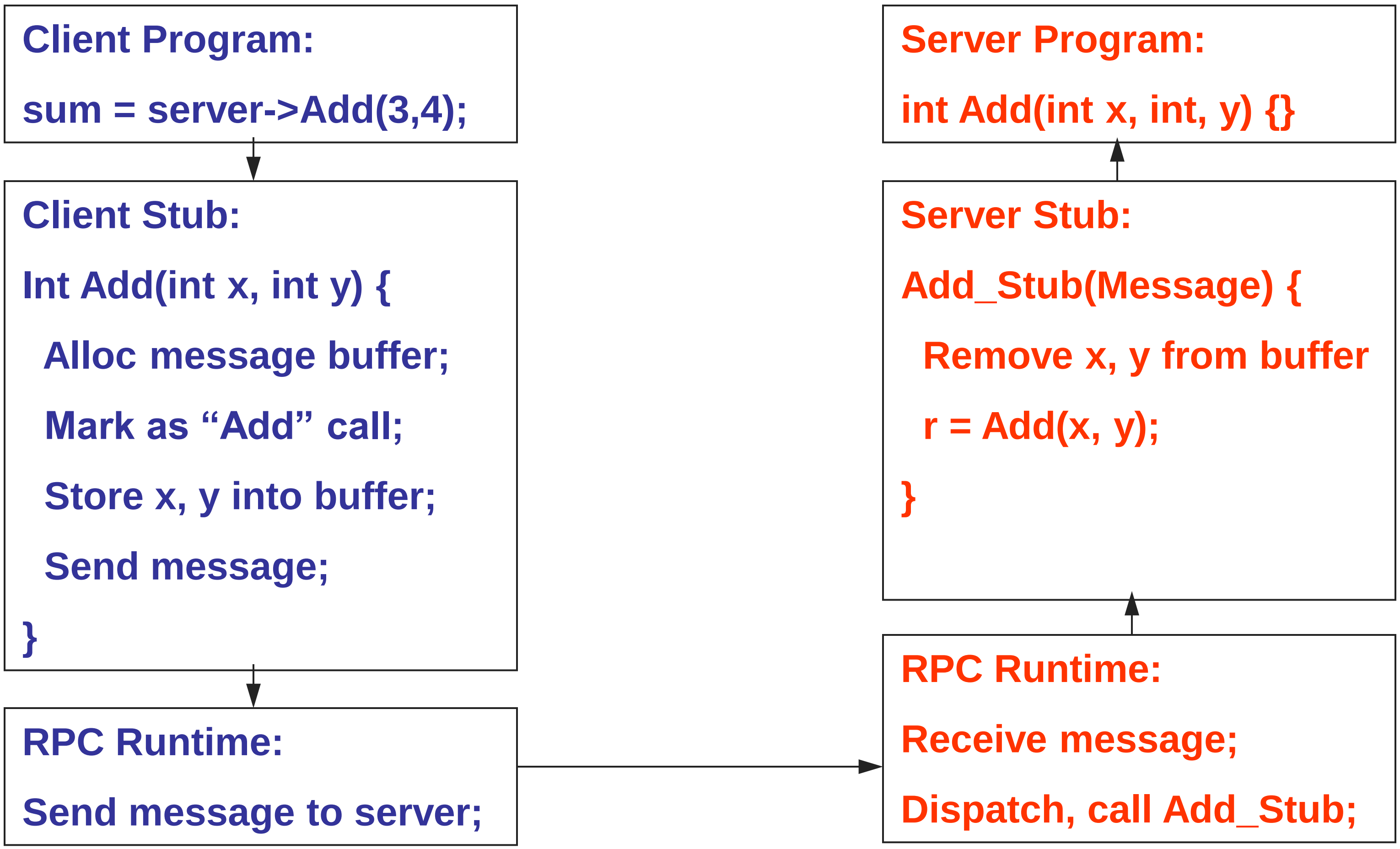
```
…
sum = server->Add(3,4);
…
```

Server Program:

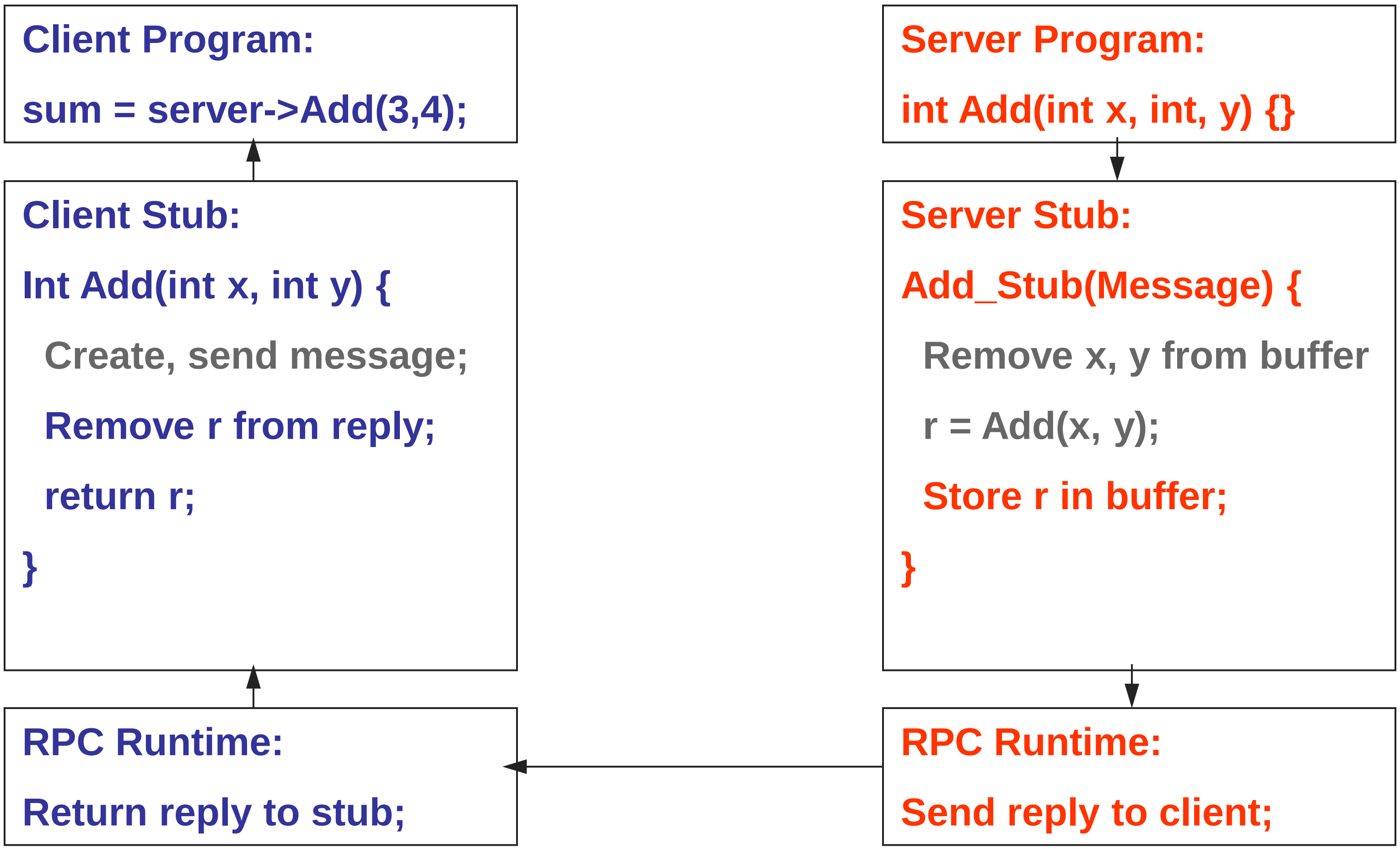
```
int Add(int x, int, y) {
  return x + y;
}
```

- If the server were just a library, then Add would just be a procedure call

RPC Example: Call



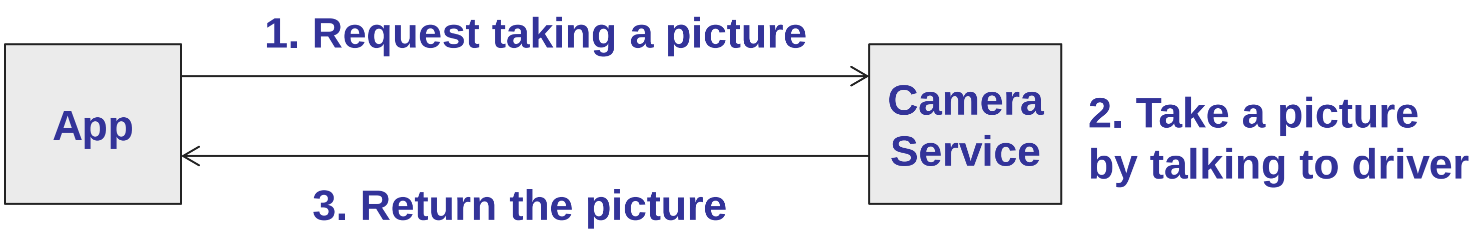
RPC Example: Return



RPC Summary

- RPC is the most common model for communication in distributed applications
  - “Cloaked” as DCOM, CORBA, Java RMI, etc.
  - Also used on same node between applications
- RPC is language support for distributed programming
- RPC relies upon a stub compiler to automatically generate client/server stubs from the IDL server descriptions
  - These stubs do the marshalling/unmarshalling, message sending/receiving/replying
- NFS uses RPC to implement remote file systems
  - Statelessness makes it easy to implement, but introduces consistency issues

Android IPC and AIDL



Why do we introduce a service process acting as a proxy to access the driver?

Android IPC and AIDL Example

Server Interface:

```
Pic takePic();
```

Client Process:

```
…
pic = server->takePic();
…
```

Server Process:

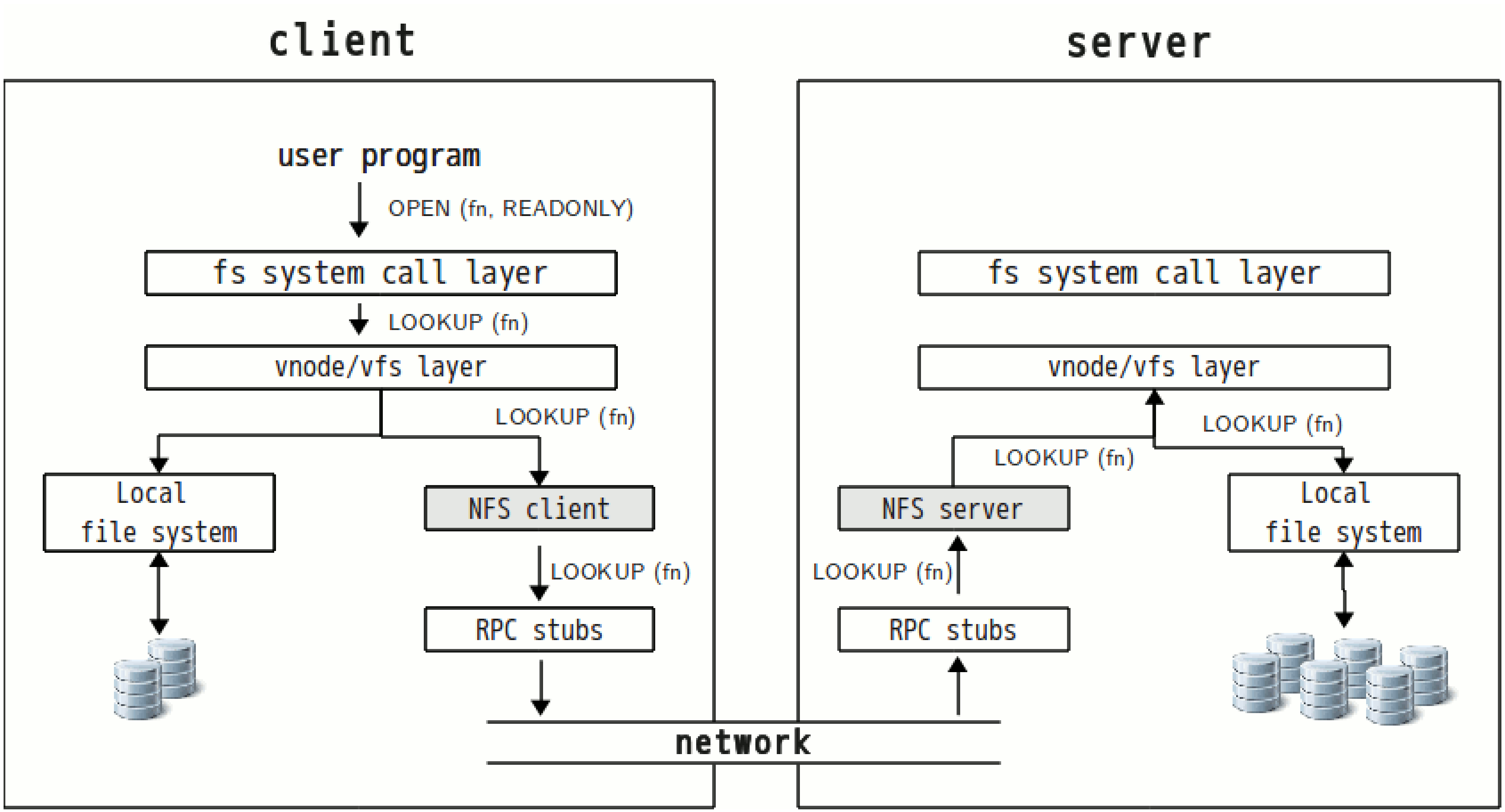
```
Pic takePic() {
  // talk to driver
  return pic;
}
```

- Why do we need this on a single host? Isn't RPC designed for remote network communication?

Network File System

- We have talked about file systems and RPC
- We'll now look at a file system that uses RPC
- Network File System (NFS)
  - Protocol for remote access to a file system
    - » Does not implement a file system per se
    - » Remote access is transparent to applications
  - File system, OS, and architecture independent
    - » Originally developed by Sun
    - » Although Unix-y in flavor, explicit goal to work beyond Unix
  - Client/server architecture
    - » Local file system requests are forwarded to a remote server
    - » These requests are implemented as RPCs

Architecture



Mounting

- Before a client can access files on a server, the client must mount the file system on the server
  - The file system is mounted on an empty local directory
  - Same way that local file systems are attached
  - Can depend on OS (e.g., Unix dirs vs NT drive letters)
  - E.g., Lab machines mount home directory from NFS servers
  - **"mount backend:/home/csgrads /home/csgrads"**
- Servers maintain ACLs of clients that can mount their directories
  - When mount succeeds, server returns a file handle
  - Clients use this file handle as a capability to do file operations
- Mounts can be cascaded
  - Can mount a remote file system on a remote file system

NFS Protocol

- The NFS protocol defines a set of operations that a server must support
  - Reading and writing files
  - Accessing file attributes
  - Searching for a file within a directory
  - Reading a set of directory links
  - Manipulating links and directories
- These operations are implemented as RPCs
  - Usually by daemon processes (e.g., nfsd)
  - A local operation is transformed into an RPC to a server
  - Server performs operation on its own file system and returns

Statelessness

- Note that NFS has no open or close operations
- NFS is stateless
  - An NFS server does not keep track of which clients have mounted its file systems or are accessing its files
  - Each RPC has to specify all information in a request
    - » Operation, FS handle, file id, offset in file, sequence #
    - » How is this good or bad?
- Robust
  - No reconciliation needs to be done on a server crash/reboot
  - Clients detect server reboot, continue to issue requests
- Writes must be synchronous to disk, though
  - Clients assume that a write is persistent on return
  - Servers cannot cache writes

Consistency

- Since NFS is stateless, consistency is tough
  - What do we mean by consistency?
  - NFS can be (mostly) consistent, but limits performance
  - NFS assumes that if you want consistency, applications will use higher-level mechanisms to guarantee it
- Writes are supposed to be atomic
  - But performed in multiple RPCs (larger than a network packet)
  - Simultaneous writes from clients can interleave RPCs (bad)
- Server caching
  - Can cache for reads, but we saw that it cannot cache writes

Consistency (2)

- Client caching can lead to consistency problems
  - Caching a write on client A will not be seen by other clients
  - Cached writes by clients A and B are unordered at server
  - Since sharing is rare, though, NFS clients usually do cache
- NFS statelessness is both its key to success and its Achilles' heel
  - NFS is straightforward to implement and reason about
  - But limitations on caching can severely limit performance
    - » Dozens of network file system designs and implementations that perform much better than NFS
  - But note that it is still the most widely used remote file system protocol and implementation