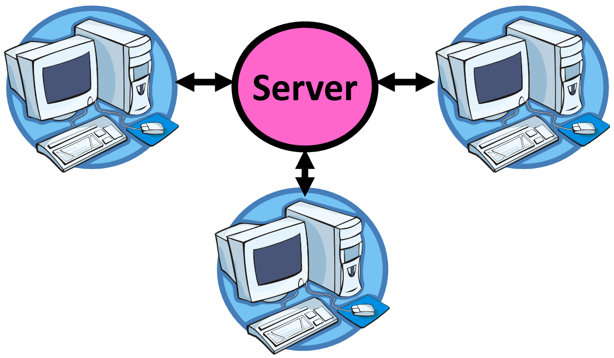
Distributed File Systems

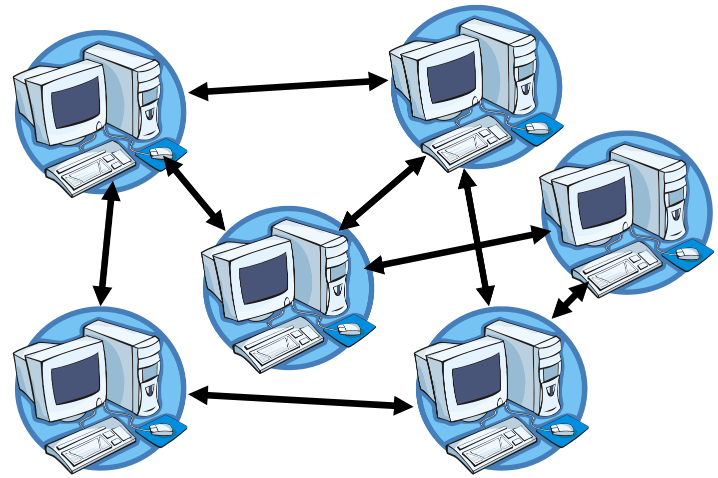
Prof. T. Venkatesh

Dept of CSE, IIT Guwahati

Centralized vs Distributed



Client/Server Model



Peer-to-Peer Model

- **Centralized System:** Major functions performed on one physical computer
  - Many cloud services logically centralized, but not physically so

- **Distributed System:** Physically separate computers working together to perform a single task

Distributed Systems: Motivation

- Why do we want distributed systems?
  - Cheaper and easier to build lots of simple computers
  - Easier to add power incrementally
  - Users can have complete control over some components
  - Collaboration: much easier for users to collaborate through network resources (such as network file systems)
- The *promise* of distributed systems:
  - Higher availability: one machine goes down, use another
  - Better durability: store data in multiple locations
  - More security: each piece easier to make secure

Distributed Systems: Reality

- Reality has been disappointing
  - Worse availability: depend on every machine being up
    - Lamport: "a distributed system is one where I can't do work because some machine I've never heard of isn't working!"
  - Worse reliability: can lose data if any machine crashes
  - Worse security: anyone in world can break into system

- Coordination is more difficult
  - Must coordinate multiple copies of shared state information (using only a network)
  - What would be easy in a centralized system becomes a lot more difficult

Distributed Systems: Goals/Requirements

- Transparency: the ability of the system to mask its complexity behind a simple interface
- Possible transparencies:
  - Location: Can’t tell where resources are located
  - Migration: Resources may move without the user knowing
  - Replication: Can’t tell how many copies of resource exist
  - Concurrency: Can’t tell how many users there are
  - Parallelism: System may speed up large jobs by splitting them into smaller pieces
  - Fault Tolerance: System may hide various things that go wrong
- Transparency and collaboration require some way for different processors to communicate with one another

Distributed Systems: Characteristics

- **Resource sharing**
  - Sharing and printing files at remote sites
  - Processing information in a distributed database
- **Computation speedup** – **load sharing** or **job migration**
- **Reliability** – detect and recover from functional failure
- Communication – **message** passing

Network Operating Systems

- ❖Users are aware of multiplicity of machines
- ❖Access to resources of various machines is done explicitly by:
  - ❖Remote logging into the appropriate remote machine (telnet, ssh)
  - ❖Remote Desktop (Microsoft Windows)
  - ❖Transferring data from remote machines to local machines, via the File Transfer Protocol (FTP) mechanism
- ❖Users must change paradigms – establish a **session**, give network-based commands
  - ❖More difficult for users

Distributed Operating Systems

- Users not aware of multiplicity of machines
  - Access to remote resources similar to access to local resources
- **Data Migration** – transfer data by transferring entire file, or transferring only those portions of the file necessary for the immediate task
- **Computation Migration** – transfer the computation, rather than the data, across the system
  - Via remote procedure calls (RPCs)
  - Via messaging system

Distributed Operating Systems

- **Process Migration** – run an entire process, or parts of it, at different sites
  - **Load balancing** – distribute processes across network to even the workload
  - **Computation speedup** – subprocesses can run concurrently on different sites
  - **Hardware preference** – process execution may require specialized processor
  - **Software preference** – required software may be run at only a particular site
  - **Large Data** – run process remotely, rather than transfer all data locally

A Few Definitions

- Availability: the probability that the system can accept and process requests
  - Often measured in “nines” of probability. So, a 99.9% probability is considered “3-nines of availability”
  - Key idea here is independence of failures
- Durability: the ability of a system to recover data despite faults
  - This idea is fault tolerance applied to data
  - Doesn’t necessarily imply availability
- Reliability: the ability of a system or component to perform its required functions under stated conditions for a specified period of time (IEEE definition)
  - Usually stronger than simply availability: means that the system is not only “up”, but also working correctly
  - Includes availability, security, fault tolerance/durability
  - Must make sure data survives system crashes, disk crashes, etc

How to Make File System Durable?

- Disk blocks contain Reed-Solomon error correcting codes (ECC) to deal with small defects in disk drive
  - Can allow recovery of data from small media defects
- Make sure writes survive in short term
  - Either abandon delayed writes or
  - Use special, battery-backed RAM (called non-volatile RAM or NVRAM) for dirty blocks in buffer cache
- Make sure that data survives in long term
  - Need to replicate! More than one copy of data!
  - Important element: independence of failure
    - Could put copies on one disk, but if disk head fails…
    - Could put copies on different disks, but if server fails…
    - Could put copies on servers in different continents…

RAID: Redundant Arrays of Inexpensive Disks

- Data stored on multiple disks (redundancy)
- Either in software or hardware
    - In hardware case, done by disk controller; file system may not even know that there is more than one disk in use
- Initially, five levels of RAID (more now)

File System Reliability

- What can happen if disk loses power or software crashes?
  - Some operations in progress may complete
  - Some operations in progress may be lost
  - Overwrite of a block may only partially complete
- Having RAID doesn’t necessarily protect against all such failures
  - No protection against writing bad state
  - What if one disk of RAID group not written?
- File system needs durability (as a minimum!)
  - Data previously stored can be retrieved (maybe after some recovery step), regardless of failure

Storage Reliability Problem

- Single logical file operation can involve updates to multiple physical disk blocks
  - inode, indirect block, data block, bitmap, …
  - With sector remapping, single update to physical disk block can require multiple (even lower level) updates to sectors
- At a physical level, operations complete one at a time
  - Want concurrent operations for performance
- How do we guarantee consistency regardless of when crash occurs?

Threats to Reliability

- Interrupted Operation
  - Crash or power failure in the middle of a series of related updates may leave stored data in an *inconsistent state*
  - Example: What if fund transfer is interrupted after withdrawal and before deposit?

- Loss of stored data
  - Failure of non-volatile storage media may cause previously stored data to disappear or be corrupted

Reliability Approach #1: Careful Ordering

- Sequence operations in a specific order
  - Careful design to allow sequence to be interrupted safely
- Post-crash recovery
  - Read data structures to see if there were any operations in progress
  - Clean up/finish as needed
- Approach taken by
  - FAT and FFS (`fsck`) to protect filesystem structure/metadata
  - Many app-level recovery schemes (e.g., Word, emacs autosaves)

FFS: Create a File

Normal operation:

- Allocate data block
- Write data block
- Allocate inode
- Write inode block
- Update bitmap of free blocks and inodes
- Update directory with file name $\rightarrow$ inode number
- Update modify time for directory

Recovery:

- Scan inode table
- If any unlinked files (not in any directory), delete or put in lost & found dir
- Compare free block bitmap against inode trees
- Scan directories for missing update/access times

Time proportional to disk size

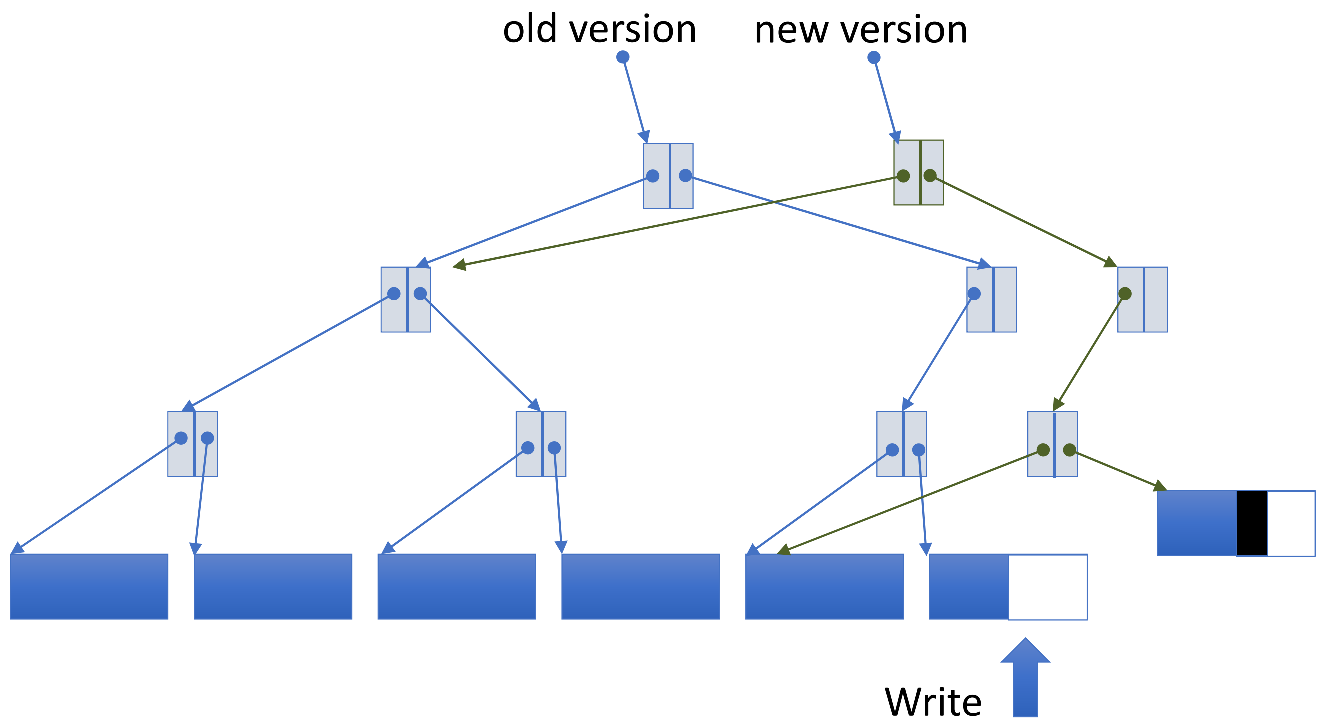
Reliability Approach #2: Copy on Write File Layout

- To update file system, write a new version of the file system containing the update
  - Never update in place
  - Reuse existing unchanged disk blocks

- Seems expensive! But
  - Updates can be batched
  - Almost all disk writes can occur in parallel

- Approach taken in network file server appliances
  - NetApp's Write Anywhere File Layout (WAFL)
  - ZFS (Sun/Oracle) and OpenZFS

COW with Smaller-Radix Blocks



- If file represented as a tree of blocks, just need to update the leading fringe

ZFS and OpenZFS

- Variable sized blocks: 512 B – 128 KB
- Symmetric tree
  - Know if it is large or small when we make the copy
- Store version number with pointers
  - Can create new version by adding blocks and new pointers
- Buffers a collection of writes before creating a new version with them
- Free space represented as tree of extents in each block group
  - Delay updates to freespace (in log) and do them all when block group is activated

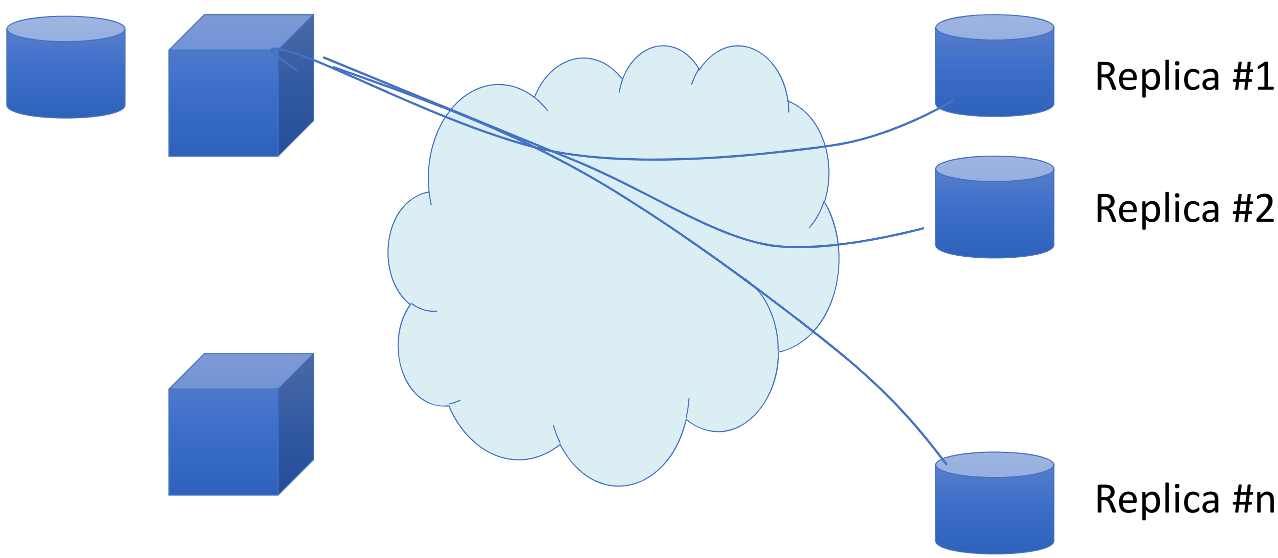
More General Reliability Solutions

- Use *Transactions* for atomic updates
  - Ensure that multiple related updates are performed atomically
  - i.e., if a crash occurs in the middle, the state of the systems reflects either *all or none* of the updates
  - Most modern file systems use transactions internally to update filesystem structures and metadata
  - Many applications implement their own transactions

- Provide *Redundancy* for media failures
  - Redundant representation on media (Error Correcting Codes)
  - Replication across media (e.g., RAID disk array)

Higher Durability/Reliability through Geographic Replication

- Highly durable – hard to destroy all copies
- Highly available for reads – read any copy
- Low availability for writes
  - Can't write if any one replica is not up
  - Or – need relaxed consistency model



Building Distributed Applications

- How do you actually program a distributed application?
  - Multiple threads, running on different machines
    - How do they coordinate and communicate
  - send/receive messages
    - Already atomic: no receiver gets portion of a message and two receivers cannot get same message
- Interface:
  - Mailbox: temporary holding area for messages
    - Includes both destination location and queue
  - `Send(message,mbox)`
    - Send message to remote mailbox identified by `mbox`
  - `Receive(buffer,mbox)`
    - Wait until `mbox` has message, copy into buffer, and return
    - If threads sleeping on this mbox, wake up one of them

Challenge of Coordination

- Components communicate over the network
  - Send messages between machines
- Need to use messages to **agree on system state**
  - in a centralized system the center “rules”

Distributed Systems – Message Passing

- Distributed systems use a variety of messaging frameworks to communicate:
  - TCP/UDP
  - 2PC for transaction processing
  - HTTP GET and POST
- Disadvantages of message passing:
  - Complex, stateful protocols, versions, feature creep
  - Need error recovery, data protection, etc.
  - Ad-hoc checks for message integrity
  - Resources consumed on server between messages (DoS risk)
  - Need to program for different OSes, target languages,…
- Want a higher-level abstraction that addresses these issues, but whose effects are application-specific

Remote Procedure Call (RPC)

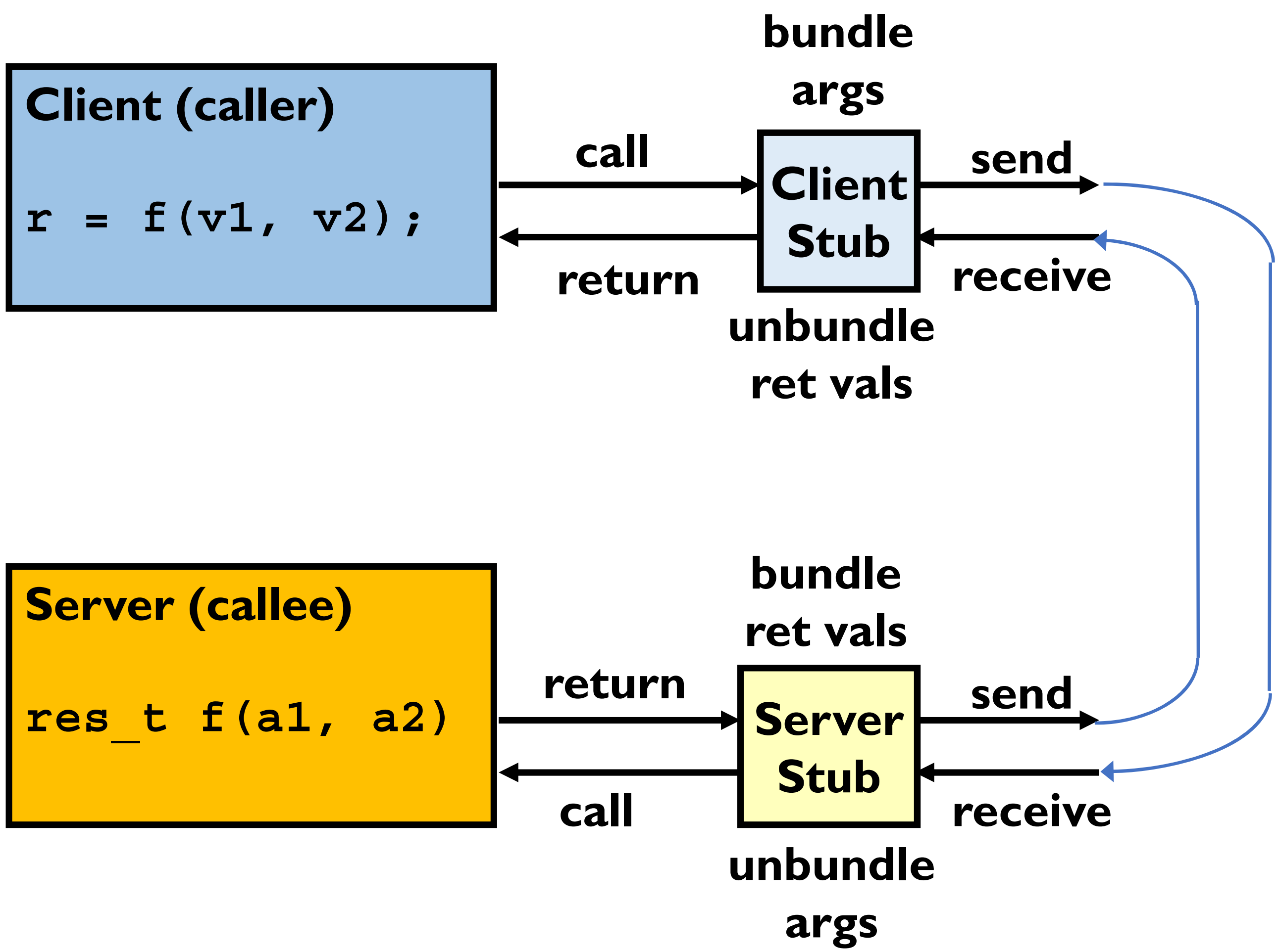
- Raw messaging is a bit too low-level for programming
  - Must wrap up information into message at source
  - Must decide what to do with message at destination
  - May need to sit and wait for multiple messages to arrive
  - And – what about machines with different byte order (“BigEndian” vs “LittleEndian”)

- Another option: Remote Procedure Call (RPC)
  - Calls a procedure on a remote machine
  - Client calls:
    `remoteFileSystem→Read("resource");`
  - Translated automatically into call on server:
    `fileSys→Read("resource");`

RPC Implementation

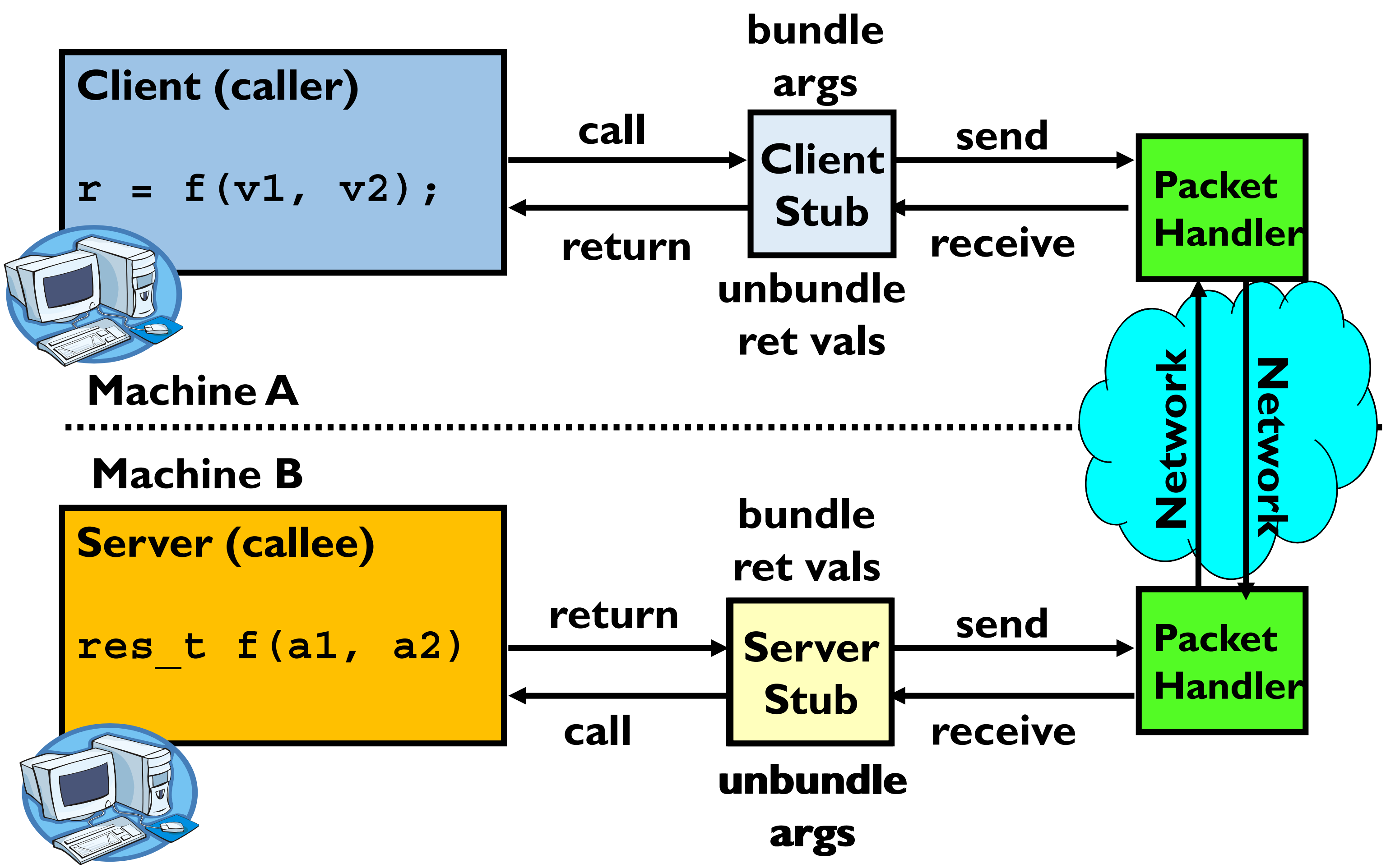
- Request-response message passing (under covers!)
- "Stub" provides glue on client/server
  - Client stub is responsible for "marshalling" arguments and "unmarshalling" the return values
  - Server-side stub is responsible for "unmarshalling" arguments and "marshalling" the return values.
- Marshalling involves (depending on system)
  - Converting values to a canonical form, serializing objects, copying arguments passed by reference, etc.

RPC Concept



Client (caller)
r = f(v1, v2);
call
return
bundle
args
Client
Stub
unbundle
ret vals
send
receive
Server (callee)
res_t f(a1, a2)
return
call
bundle
ret vals
Server
Stub
unbundle
args
send
receive

RPC Information Flow



RPC Details (1/3)

- Equivalence with regular procedure call
  - Parameters ⇔ Request Message
  - Result ⇔ Reply message
  - Name of Procedure: Passed in request message
  - Return Address: c_mbox (client return mail box)

- Stub generator: Compiler that generates stubs
  - Input: interface definitions in an "interface definition language (IDL)"
    - Contains, among other things, types of arguments/return
  - Output: stub code in the appropriate source language
    - Code for client to pack message, send it off, wait for result, unpack result and return to caller
    - Code for server to unpack message, call procedure, pack results, send them off

RPC Details (2/3)

- Cross-platform issues:
  - What if client/server machines are different architectures/ languages?
    - Convert everything to/from some canonical form
    - Tag every item with an indication of how it is encoded (avoids unnecessary conversions)

- How does client know which mbox (destination queue) to send to?
  - Need to translate name of remote service into network endpoint (Remote machine, port, possibly other info)
  - Binding: the process of converting a user-visible name into a network endpoint
    - This is another word for “naming” at network level
    - Static: fixed at compile time
    - Dynamic: performed at runtime

RPC Details (3/3)

- Dynamic Binding
  - Most RPC systems use dynamic binding via name service
    - Name service provides dynamic translation of service $\rightarrow$ mbox
  - Why dynamic binding?
  - Access control: check who is permitted to access service
    - Fail-over: If server fails, use a different one

- What if there are multiple servers?
  - Could give flexibility at binding time
    - Choose least loaded server for each new client
  - Could provide same mbox (router level redirect)
    - Choose least loaded server for each new request
    - Only works if no state carried from one call to next

- What if multiple clients?
  - Pass pointer to client-specific return mbox in request

Structure of RPC

- A language for describing remote interfaces of the procedure calls
- Stubs- entities that work like an object that has multiple procedures and transform the data in the system into the shared data types
- Run-time library for generic support of the operations that solve the problems mentioned
- A name manager to locate the servers

Cross-Domain Communication/Location Transparency

- RPC's can be used to communicate between address spaces on different machines or the same machine
  - Services can be run wherever it's most appropriate
  - Access to local and remote services looks the same
- Examples of RPC systems:
  - CORBA (Common Object Request Broker Architecture)
  - DCOM (Distributed COM)
  - RMI (Java Remote Method Invocation)
  - gRPC (Google RPC)

Problems with RPC: Non-Atomic Failures

- Before RPC: whole system would crash/die
- After RPC: One machine crashes/compromised while others keep working
- Can easily result in inconsistent view of the world
  - Did my cached data get written back or not?
  - Did server do what I requested or not?
- Answer? Distributed transactions/Byzantine Commit

Problems with RPC: Performance

- RPC is *not* performance transparent:
  - Cost of Procedure call « same-machine RPC « network RPC
  - Overheads: Marshalling, Stubs, Kernel-Crossing, Communication
- Programmers must be aware that RPC is not resilient
  - Caching can help, but may make failure handling complex

Distributed Systems and the CAP Theorem

- Consistency:
  - Changes appear to everyone in the same serial order
- Availability:
  - Can get a result at any time
- Partition-Tolerance
  - System continues to work even when network becomes partitioned
- Consistency, Availability, Partition-Tolerance (CAP) Theorem: Cannot have all three at same time
  - Otherwise known as “Brewer’s Theorem”
- Used in design of NoSQL DBs: AP (Cassandra), or CP (MongoDB)

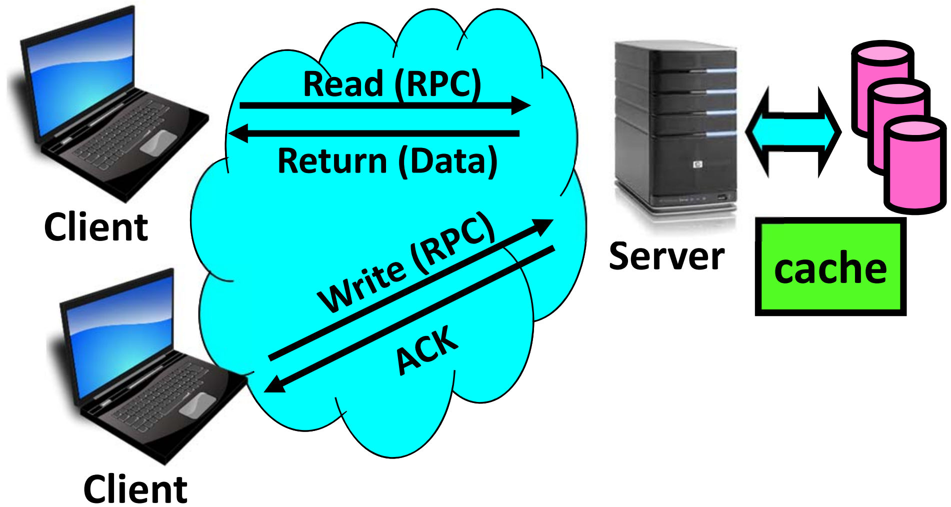
Distributed File Systems

- **Distributed file system** (**DFS**) – a distributed implementation of the classical model of a file system, where multiple users share files and storage resources
- A DFS manages set of dispersed storage devices
- Overall storage space managed by a DFS is composed of different, remotely located, smaller storage spaces
- There is usually a correspondence between constituent storage spaces and sets of files
- Challenges include:
  - Naming and Transparency
  - Remote File Access
- Key-Value Stores are special type of DFS

Distributed File Systems

- **Service** – software entity running on one or more machines and providing a particular type of function to a priori unknown clients
- **Server** – service software running on a single machine
- **Client** – process that can invoke a service using a set of operations that forms its client interface
- A client interface for a file service is formed by a set of primitive file operations (create, read, update, delete (CRUD))
- Client interface of a DFS should be transparent, i.e., not distinguish between local and remote files
  - e.g., `/home/oksi/162/` on laptop actually refers to `/users/oski` on remote file server

Simple Distributed File System



- Remote Disk: Reads and writes forwarded to server
  - Use Remote Procedure Calls (RPC) to translate file system calls into remote requests
  - No local caching, can be cached at server-side
- Advantage: Server provides completely consistent view of file system to multiple clients
- Problems? Performance!
  - Going over network is slower than going to local memory
  - Lots of network traffic/not well pipelined
  - Server can be a bottleneck

Dealing with Failures

- What if server crashes? Can client wait until it comes back and just continue making requests?
  - Changes in server's cache but not in disk are lost

- What if there is shared state across RPC's?
  - Client opens file, then does a seek
  - Server crashes
  - What if client wants to do another read?
- Similar problem: What if client removes a file but server crashes before acknowledgement?

Remote File Service

- **Remote-service mechanism -** reduce network traffic by retaining recently accessed disk blocks in a cache, so that repeated accesses to the same information can be handled locally
  - If data not already cached, a copy of data is brought from the server to the user
  - Accesses are performed on the cached copy
  - Files identified with one master copy residing at the server machine, but copies of (parts of) the file are scattered in different caches
  - **Cache-consistency problem** – keeping the cached copies consistent with the master file

Cache Update Policy

- **Write-through** – write data through to disk as soon as they are placed on any cache
- **Delayed-write** (**write-back**) – modifications written to the cache and then written through to the server later
  - Write accesses complete quickly; some data may be overwritten before they are written back, and so need never be written at all
  - Poor reliability; unwritten data will be lost when a user machine crashes
  - Variation – scan cache at regular intervals and flush blocks that have been modified since the last scan
  - **write-on-close**, writes data back to the server when the file is closed
    - Best for files that are open for long periods and frequently modified

Consistency

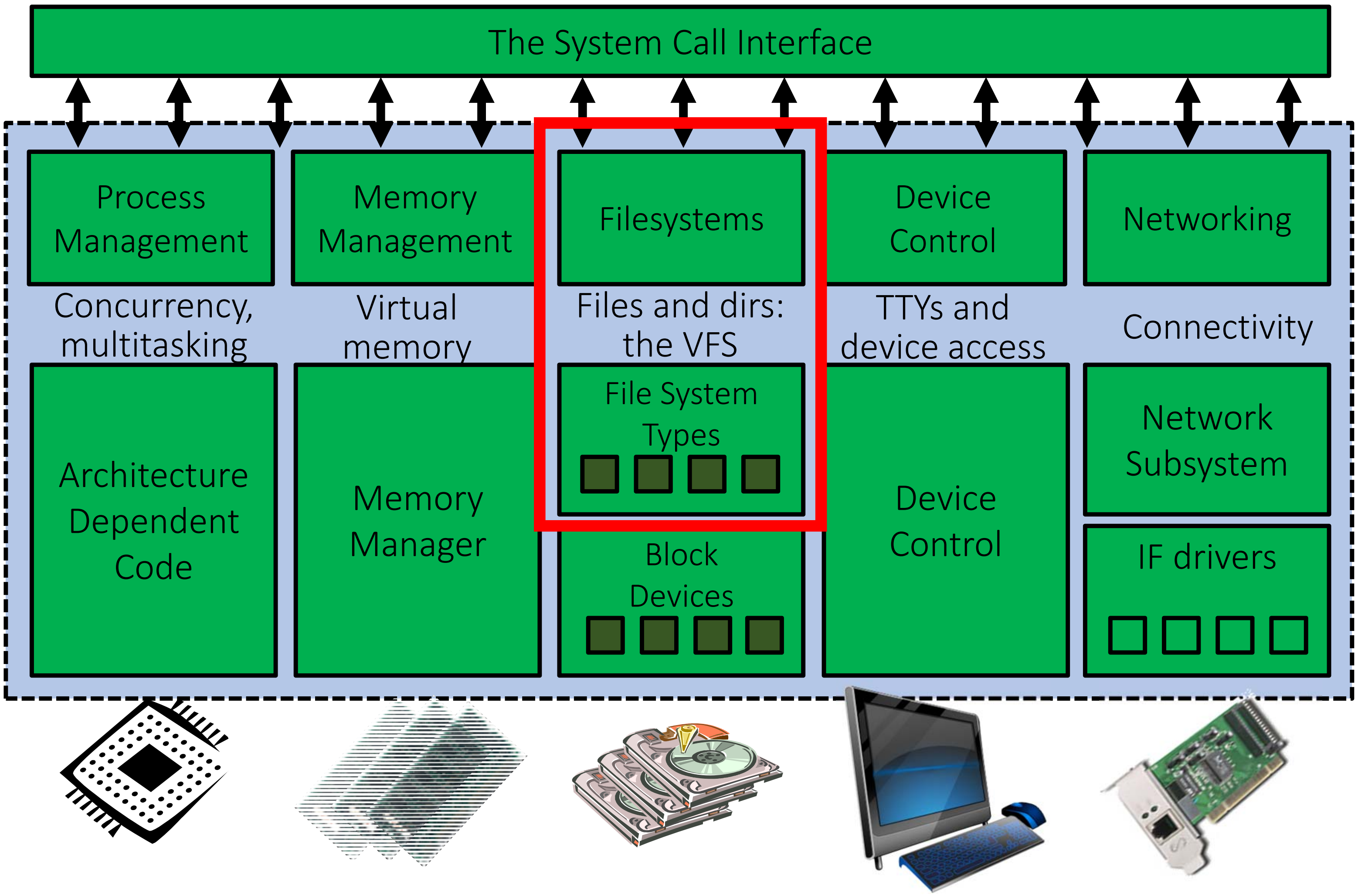
- Is locally cached copy of the data consistent with the master copy?
- **Client-initiated approach**
  - Client initiates a validity check
  - Server checks whether the local data are consistent with the master copy
- **Server-initiated approach**
  - Server records, for each client, the (parts of) files it caches
  - When server detects a potential inconsistency, it must react

Stateless Protocol

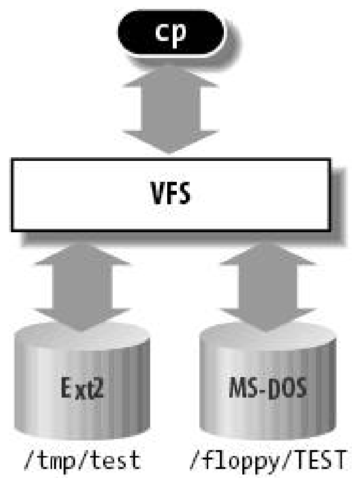
- A protocol in which all information required to service a request is included with the request
- Even better: Idempotent Operations – repeating an operation multiple times is same as executing it just once (e.g., storing to a mem addr.)
- Client: timeout expires without reply, just run the operation again (safe regardless of first attempt)

- Recall HTTP: Also a stateless protocol
  - Include cookies with request to simulate a session

Enabling Design: VFS



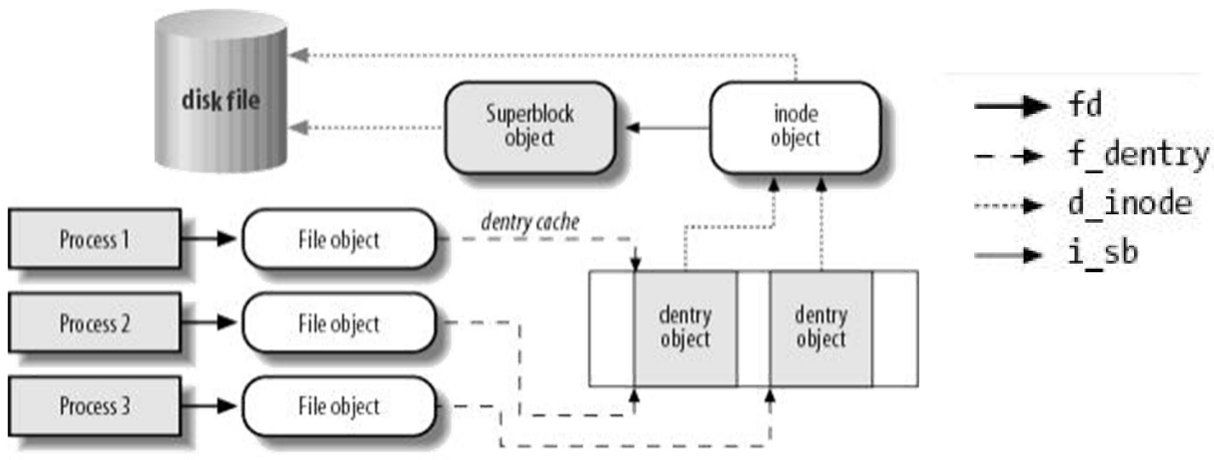
Virtual Filesystem Switch




```
inf = open("/floppy/TEST", O_RDONLY, 0);
outf = open("/tmp/test",
       O_WRONLY|O_CREAT|O_TRUNC, 0600);
do {
    i = read(inf, buf, 4096);
    write(outf, buf, i);
} while (i);
close(outf);
close(inf);
```

- VFS: Virtual abstraction similar to local file system
  - Provides virtual superblocks, inodes, files, etc
  - Compatible with a variety of local and remote file systems
    - provides object-oriented way of implementing file systems
- VFS allows the same system call interface (the API) to be used for different types of file systems
  - The API is to the VFS interface, rather than any specific type of file system

VFS Common File Model in Linux

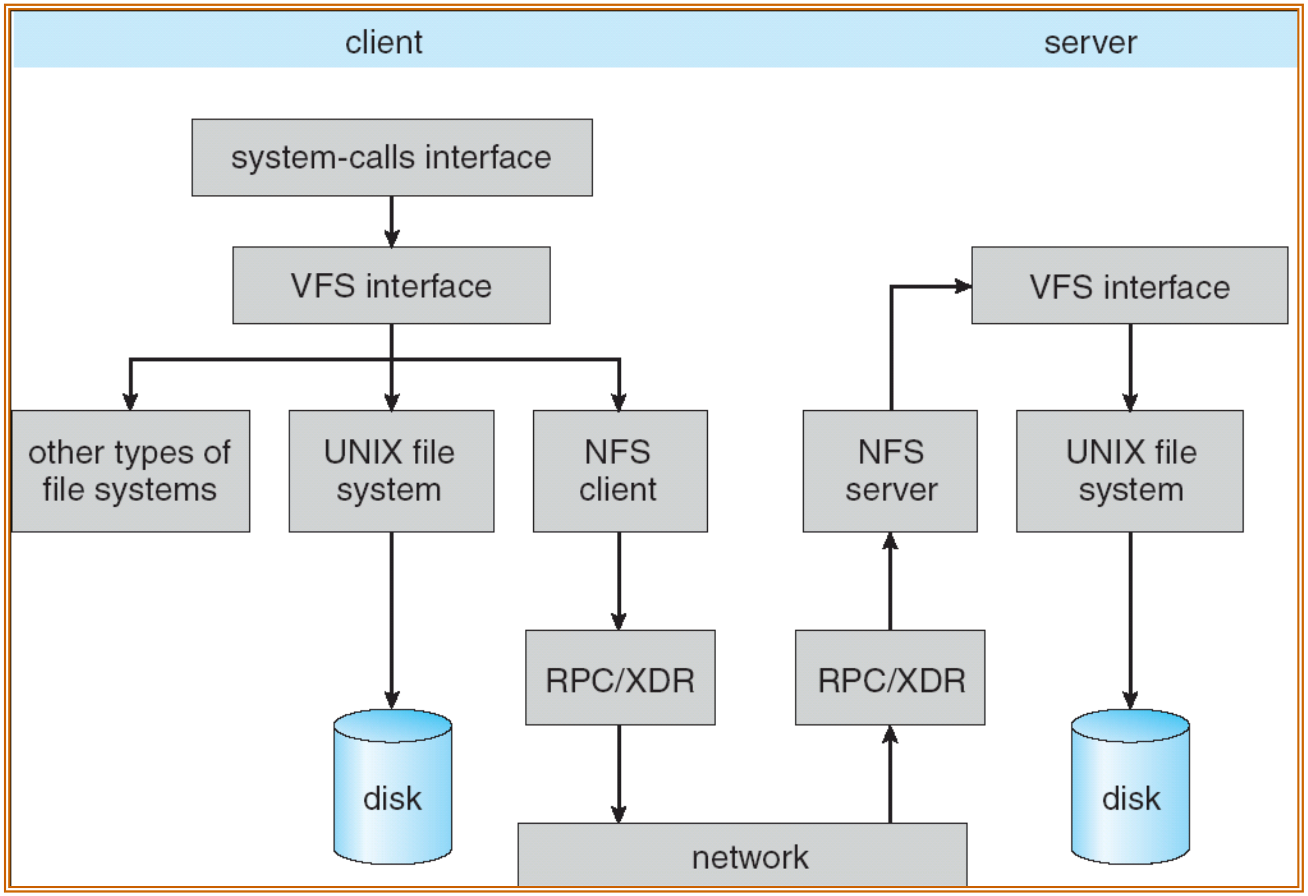


- Four primary object types for VFS:
  - superblock object: represents a specific mounted filesystem
  - inode object: represents a specific file
  - dentry object: represents a directory entry
  - file object: represents open file associated with process
- There is no specific directory object (VFS treats directories as files)
- May need to fit the model by faking it
  - Example: make it look like directories are files
  - Example: make it look like have inodes, superblocks, etc.

Network File System (Sun)

- Defines an RPC protocol for clients to interact with a file server
  - E.g., read/write files, traverse directories, …
  - Stateless to simplify failure cases
- Keeps most operations idempotent
  - Even removing a file: Return advisory error second time
- Don't buffer writes on server side cache
  - Reply with acknowledgement only when modifications reflected on disk

NFS Architecture



Network File System (NFS)

- Three Layers for NFS system
  - UNIX file-system interface: open, read, write, close calls + file descriptors
  - VFS layer: distinguishes local from remote files
    - Calls the NFS protocol procedures for remote requests
  - NFS service layer: bottom layer of the architecture
    - Implements the NFS protocol
- NFS Protocol: RPC for file operations on server
  - Reading/searching a directory
  - manipulating links and directories
  - accessing file attributes/reading and writing files
- Write-through caching: Modified data committed to server's disk before results are returned to the client
  - lose some of the advantages of caching
  - time to perform write() can be long
  - Need some mechanism for readers to eventually notice changes!

NFS Continued

- NFS servers are stateless; each request provides all arguments require for execution
  - E.g. reads include information for entire operation, such as **`ReadAt(inumber,position)`**, not **`Read(openfile)`**
  - No need to perform network open() or close() on file – each operation stands on its own
- File operations are idempotent: Performing requests multiple times has same effect as performing it exactly once
  - Example: Server crashes between disk I/O and message send, client resend read, server does operation again
  - Every request has sequence number, lets server determine sequence and duplicate

NFS Cache consistency

- NFS protocol: weak consistency
  - Client polls server periodically to check for changes
    - Polls server if data hasn't been checked in last 3-30 seconds (exact timeout is tunable parameter).
    - Thus, when file is changed on one client, server is notified, but other clients use old version of file until timeout.
    - If read starts more than 30 seconds after write, get new copy; otherwise, could get partial update

  - What if multiple clients write to same file?
    - In NFS, can get either version (or parts of both)
    - Completely arbitrary!