

---

# CSC 369

Week 11/12: Distributed File Systems &  
RPC

Reading: Text, Chapter 8.2.4, 8.3, 10.6.4

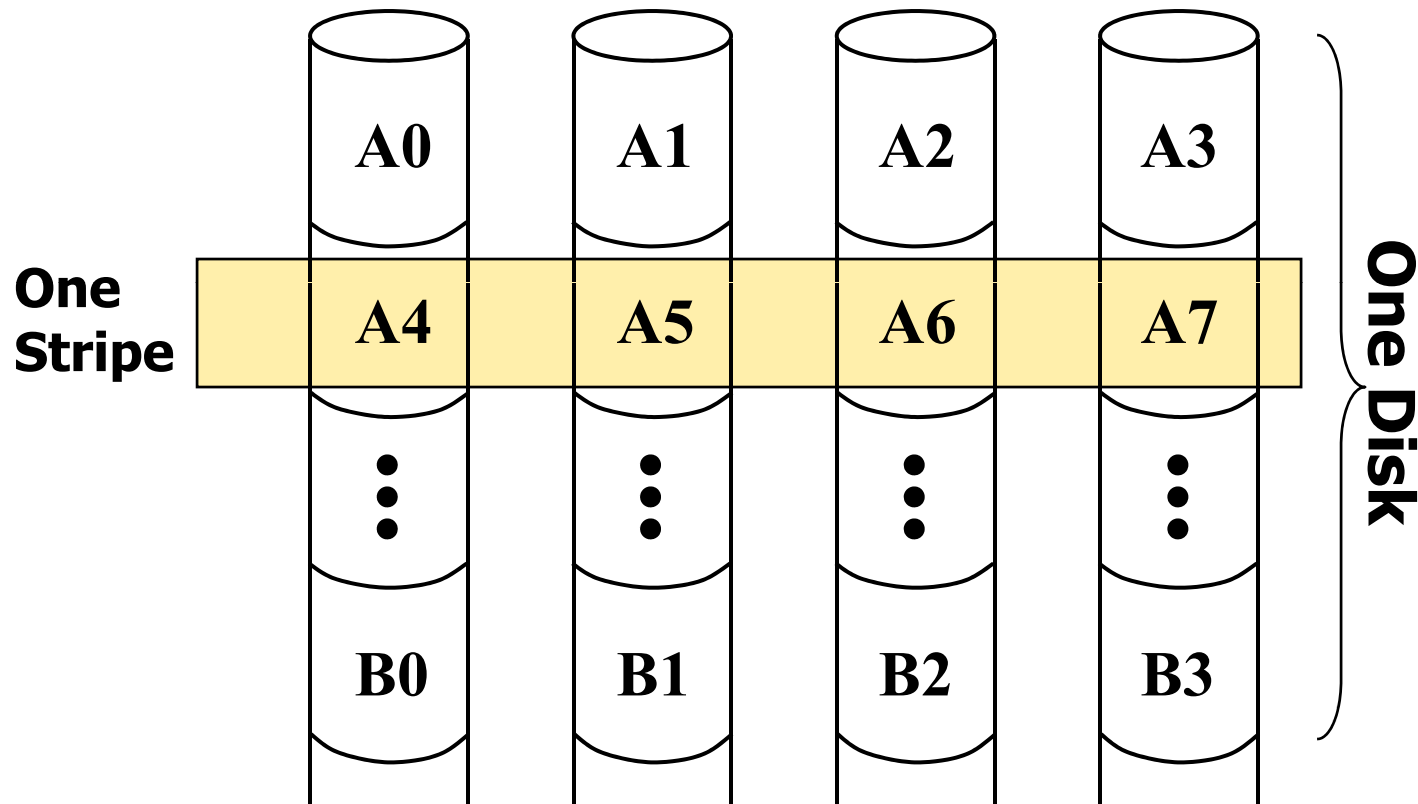
# Announcements

- Assignment 3 (last one!) out tomorrow
  - Add buffer cache to SFS (simple file system)
  - Implement file and file system related system calls
- CSSU elections on Monday, April 2<sup>nd</sup>
  - <http://www.cdf.toronto.edu/~vote>
- TODAY: Storage Systems & Dist. Systems

# RAID

- Redundant Array of Inexpensive Disks (RAID)
  - A storage system, not a file system
  - Patterson, Katz, and Gibson (Berkeley, '88)
- Idea: Use many disks in parallel to increase storage bandwidth, improve reliability
  - Files are striped across disks
  - Each stripe portion is read/written in parallel
  - Bandwidth increases with more disks
  - Better throughput for large requests

# RAID Level 0: Disk Striping



# Disk striping details

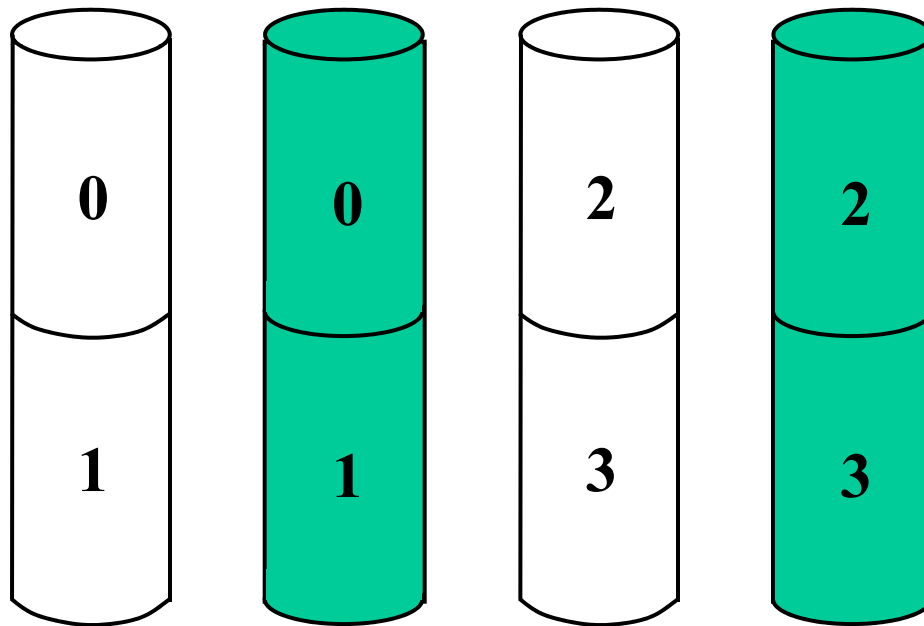
- Break up total space into fixed-size **stripe units**
- Distribute the stripe units among disks in round-robin fashion
- Compute location of block #B as follows
  - $\text{disk\#} = B \% N$  (%=modulo, N = # of disks)
  - $\text{LBN\#} = B / N$  (computes the LBN on given disk)
- Key design decision: picking the stripe unit size
- Improves throughput for large or multiple requests
- No redundancy

# RAID 0 Challenge

- Reliability
  - More disks increases the chance of media failure (MTBF)
- Turn reliability problem into a feature
  - Use one disk to store parity data
    - XOR of all data blocks in stripe
  - Can recover any data block from all others + parity block
  - Hence "redundant" in name
  - Introduces overhead, but, hey, disks are "inexpensive"

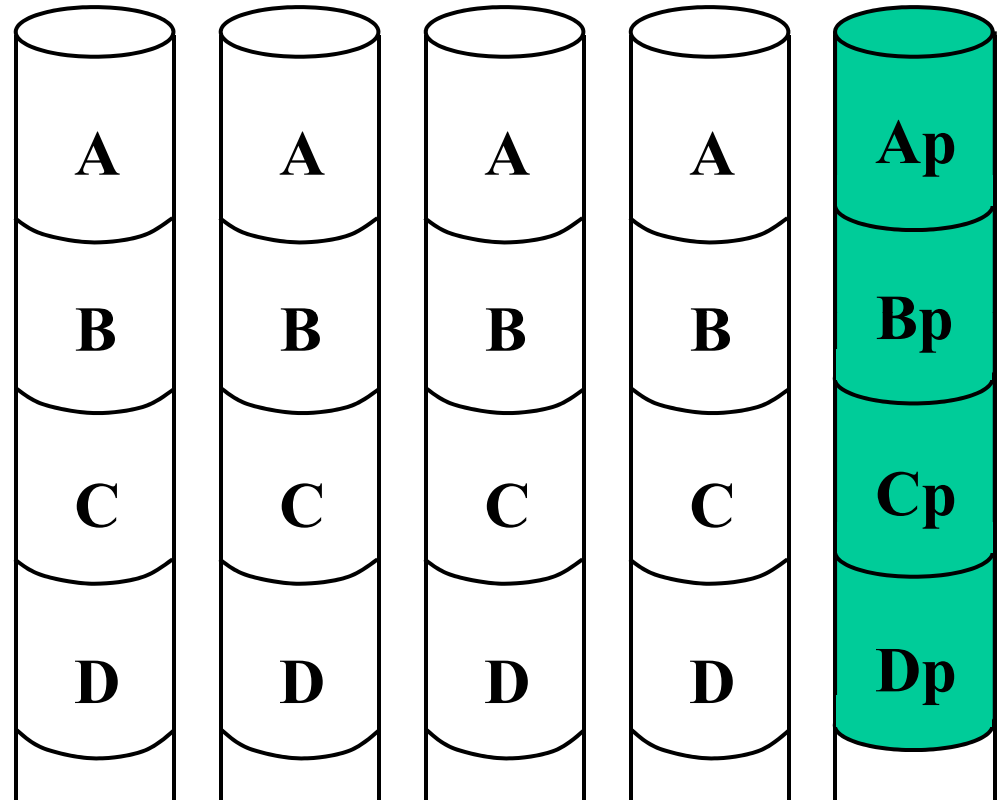
# RAID Level 1: Mirroring

- Redundancy via replication, two (or more) copies
  - mirroring, shadowing, duplexing, etc.
- Write both, read either



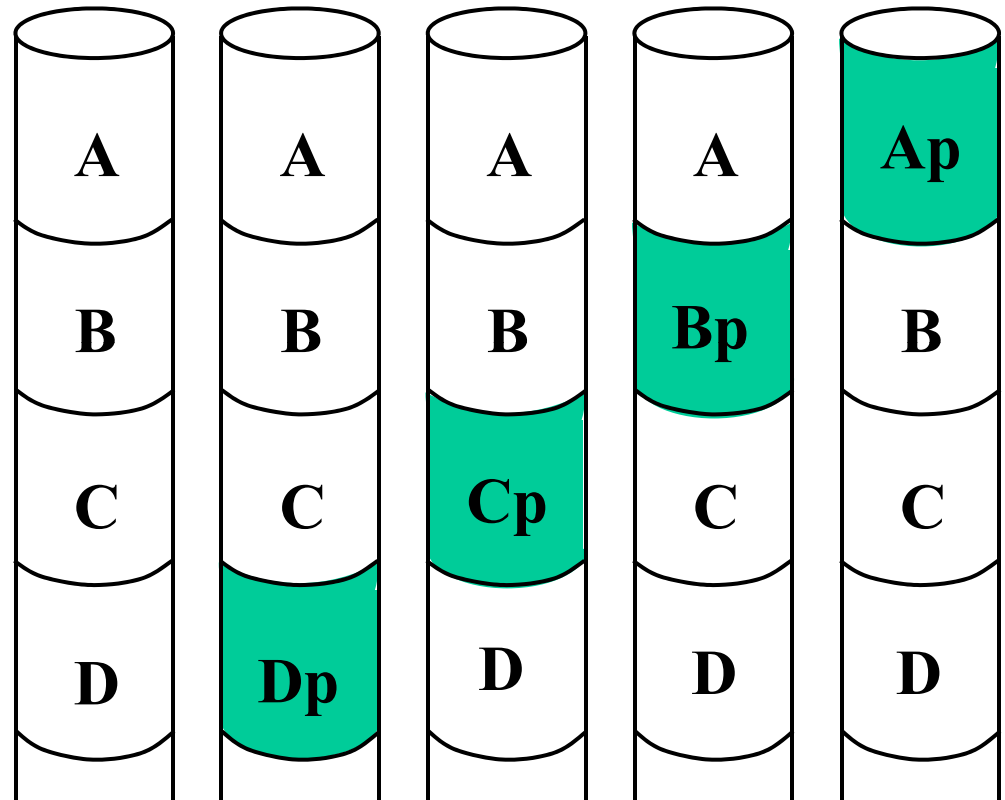
# RAID Level 2&3: Parity disks

- Both:
  - very small stripe unit (single byte or word)
  - All writes update parity disk(s)
  - Can correct single-bit errors
- Level 2:
  - #parity disks =  $\log_2(\#data\ disks)$
  - overkill
- Level 3:
  - One extra disk



# RAID Levels 4-6

- Levels 4-6 introduce *independence*:
  - Each disk may be writing different data
- Level 4 is similar to Level 3
- Level 5 removes parity disk bottleneck →
  - Distributes parity bits across all data disks
- Level 6 tolerates 2 errors



# What is a Distributed System?

- “You know you have one when the crash of a computer you’ve never heard of stops you from getting any work done.” - Leslie Lamport, 1987
- A collection of (perhaps) heterogeneous nodes connected by one or more interconnection networks which provides access to system-wide shared resources and services.
- A collection of independent computers that appears to its users as a single coherent system.

# Key Features

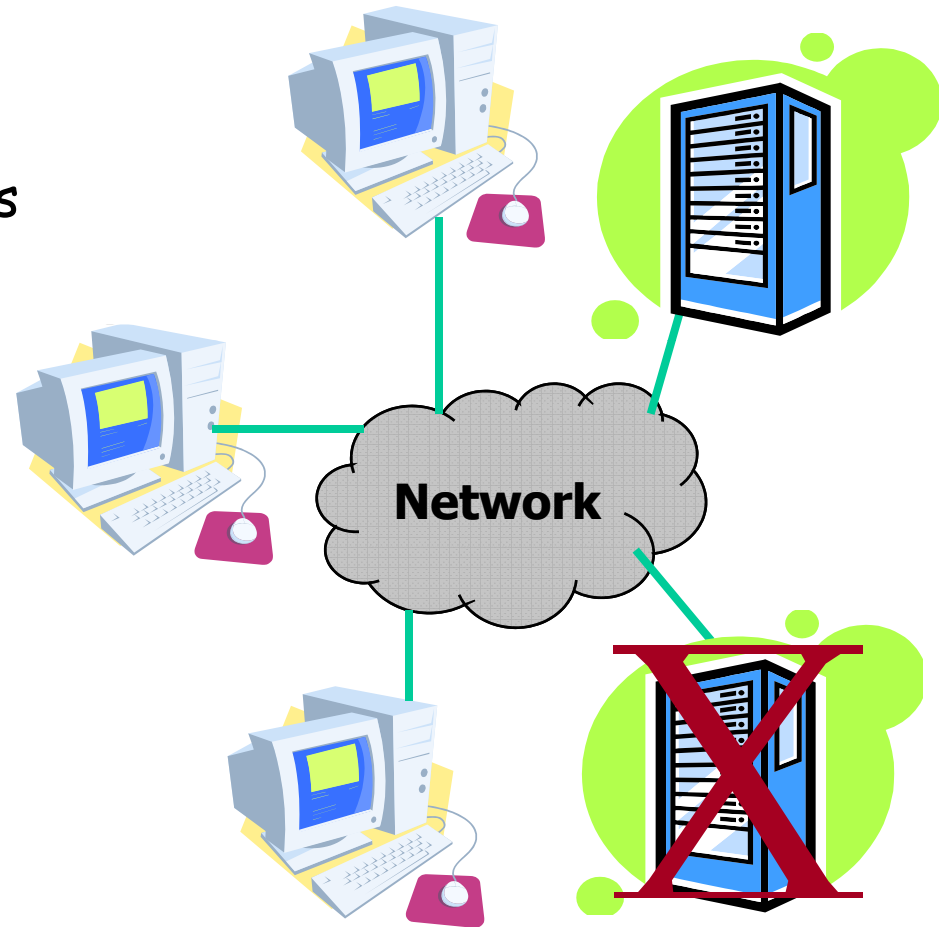
- *Multiple* computers
  - May be heterogenous, or homogeneous
  - May be controlled by a single organization or by distinct organizations or individuals
- Connected by a *communication network*
  - Typically a general-purpose network, not dedicated to supporting the distributed system
- *Co-operating* to share resources and services
  - Application processing occurs on more than one machine

# Technology Trends

- Parallel processing - divide computation among multiple physical processors
  - 2 schemes for building such systems
    - **multiprocessor system**: tightly coupled; the processors share memory and a clock; communication usually takes place through shared memory; dedicated internal communication network
    - **distributed system**: loosely coupled; processors do not share memory or a clock; communication usually takes place through external general-purpose networks
- Cheap, powerful desktop computers
- High performance networks

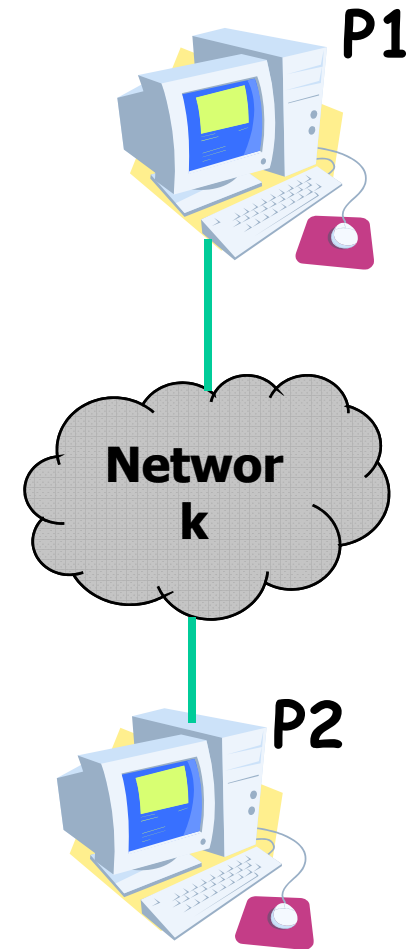
# Motivation for Dist. Systems

- Resource sharing
- Computation speedup
  - can allow subcomputations to be run concurrently
  - load sharing
- Reliability
  - if enough redundancy exists, the system can continue operation, even with some sites failing
- Communication
  - support for group collaboration applications



# Terminology

- From the viewpoint of a specific processor in a Distributed System, the rest of the processors and their resources are *remote*, and its own are *local*
- Computing entities in the DS are referred to by many names
  - *sites, nodes, computers, machines, hosts,...*
- Server host, client host, peer-to-peer



# Operating System Options

- **Network OS**

- Each computer has its own private OS
- OS includes network services to allow access to remote resources
- User is aware of multiple computers and must deal with them explicitly.

- **Distributed OS**

- OS's on each computer cooperate with each other to provide user with a "single-system" image
- User is unaware of multiple computers

- **Middleware**

- A layer of software between OS and application that supports distributed computing

# Network Operating Systems

- Typical examples are the Unix  $r^*$  commands
- remote login (rlogin)
  - transparent, bi-directional link
- remote file transfer (rcp)
  - file location is not transparent to the user
  - no real file sharing
- Rsh, rexec, etc.
- services are implemented by daemon processes on the remote site watching for connection requests
  - Local process is spawned to handle each request

# Distributed Operating Systems

- Built-in support for *migration*
  - *data migration*
    - either whole file transfer from site B to site A, or the transfer of those portions of the file that are actually necessary for the immediate task
  - *computation migration*
    - maybe more efficient to transfer computation, rather than the data, across the system
    - E.g. send an executable function to the machine where the data is stored, rather than sending the data to the machine where the process is running
  - *process migration*
    - a process is not always executed at a site in which it is initiated (for load-balancing or reliability for example)

# Client/Server Computing

- Application tasks are split between *client* machines and *server* machines.
  - Typically servers are specialized in some way that would be difficult to provide directly on a large number of clients
    - May have more memory, more or faster processors, more storage, licences for expensive software, etc.
  - Choosing where to split the application depends on characteristics of the client, server, application itself, communication network, and expected system load

# What are some examples?

- Distributed databases
- World-wide-web
- Give me some more... (there's lots!)
- What about napster? gnutella? Kazaa?
  - Many "peer-to-peer" systems present hybrid architectures

# Communication in Dist. Sys.

- No physically shared memory, must send messages
- Basic primitives are `Send()` and `Receive()`
  - Client `Send()`'s a request for a service, waits to `Receive()` a reply. Server `Receive()`'s a request, processes it and `Send`'s a reply.
  - All data has to be packaged as part of the message body
- Message facility may be reliable or unreliable (if unreliable, higher level has to check for successful transmission)
- Message primitives may be blocking or non-blocking
- May provide specialized communication protocol, or build on general-purpose protocol like TCP/IP

# Distributed File Systems

- a file system whose clients, servers, and storage devices are dispersed among the machines of a Distributed System
- the multiplicity and dispersion of its servers and storage devices should be made transparent
- important performance metric is the amount of time to satisfy service requests
  - overhead associated with distributed structure
  - relates closely to the *scalability* of the dist. file system

# Naming & Location

- A key problem in distributed file systems is locating the physical storage for a logical file name
  - Actually, this is a key problem in local file systems too
  - *Naming* refers to the mapping between logical and physical objects
  - Local file systems have multiple layers of name mapping and translation, hiding physical location from users (sometimes to varying degrees):
    - E.g.) read file from CD-ROM in Windows/MS-DOS)
      - D:\README.txt
    - E.g.) read file from CD-ROM in Unix local FS
      - /cdrom/README.txt
  - Note that in Windows, "name" of file includes partition information

# Name Transparency

- *Transparency* hides the structure of the file system
  - Windows is not transparent - name mapping does not include partition identifier, user must specify
- We can have the same options in dist. file systems
  - mapping **does not** include identity of machine that stores file (no name transparency)
    - E.g. my home dir on cdf could be `homesrv:/h/u1/demke`
  - mapping **does** includes identity of machine that stores file
    - E.g. my home dir on cdf is simply `/h/u1/demke`
    - In this case, another layer must map this filename to `homesrv:/h/u1/demke`

# Aspects of Transparency

- **location transparency:** the name of a file does not reveal any hint of the file's physical storage location
- **location independence:** the name of a file does not need to be changed when the file's physical storage location change
- Location-independent naming is a dynamic mapping, so location independence is a stronger property than location transparency
  - Location transparency implies only that the users do not know the location of files based on their names
  - File system itself may permanently map names to storage locations

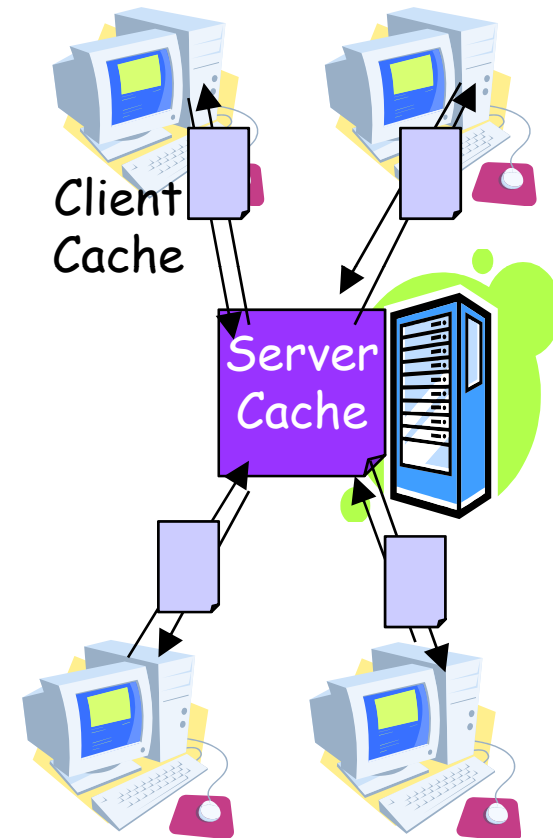
# Locating files in Unix

- Once the separation of name and location has been completed, files residing on remote server systems may be accessed by various clients
  - "Remote" file systems are "glued" into the clients local namespace
  - Managed by the `/etc/fstab` file

```
# /etc/fstab: static file system information.
#
#<file sys> <mount point> <type> <options> <dump> <pass>
/dev/sda5 / ext3 rw,errors=remount-ro 0 1
/dev/sda6 none swap rw 0 0
none /proc proc defaults 0 0
/dev/hda /cdrom auto ro,user,nohide,noauto 0 0
homesrv:/h/u1 /h/u1 nfs rw,bg,rsize=8192,wsiz=8192 0 0
```

# Remote File Access

- Data transfer achieved through a remote-service mechanism (and RPC)
  - requests for access are delivered to the server, the server machine performs the accesses, and the results are forwarded back to the user
- To ensure reasonable performance, use **caching**
  - local FS: reduce disk I/O
  - DFS: reduce both network traffic and disk I/O
- Caching may occur both at the clients and at the server
  - Server caching reduces access time for files that are commonly used by different clients

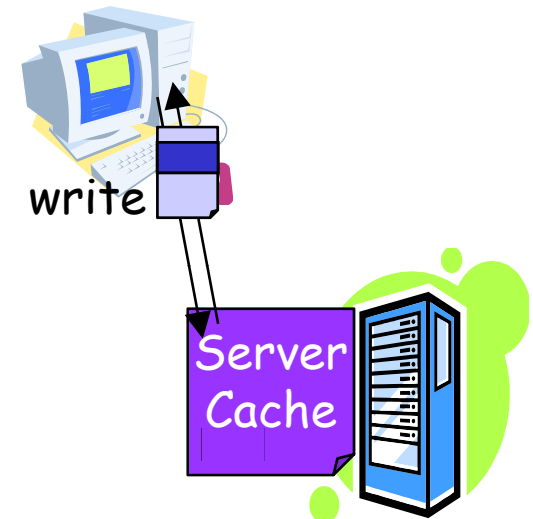


# Basic Client Caching

- keep a copy of the access requests on the client when they come from the server
  - May be cached strictly in client memory or on the client's local disk
- client performs subsequent accesses on the cached copy (no added network traffic)
- As always, need a replacement algorithm
- Server always stores "master" copy
- Granularity: may cache whole files or individual blocks
- Cache consistency problem
  - Server should return most recent version of file to client
  - master copy must be consistent with the cached copies

# Cache Update Policy

- Defn: the policy for writing modified data blocks back to the server's master copy
- **write-through:**
  - Write data through to the server disk as soon as it appears in a cache
  - Simple
  - Reliable
  - Every write access must wait for server reply
  - Equivalent to remote service for write accesses and caching for read accesses



# More Cache Update Policies

- **Delayed-write:**

- Modifications are written to the cache and then are written through to the server at a later time
- Writes complete more quickly than with write-through
  - Application returns from syscall when local write is complete
- Less data may need to be transferred (some writes may overwrite others before updates are transferred to the server)
- Less reliable - data isn't stable until it is written to server
  - New error conditions are possible (e.g. "out of space" on server long after application was told write succeeded)

# Deciding when to send writes

- If writes are delayed, when should they be sent to the server?
  - When a block is about to be ejected from the client cache
  - At regular intervals: flush blocks that have been modified since the last scan (length of interval determines window of vulnerability to crashes)
  - When the file is closed on the client
  - When another client reads the file/block
- What support is needed from the server for each of these options?

# Cache Consistency

- Client must decide if cached data is valid (i.e., consistent with the master copy at the server)
  - If not, then a fresh copy has to be fetched from the server
- Two basic options:
  - **Client-initiated** consistency checks - client contacts the server before using cached data to serve a request (can be on every data access, on file open, or at fixed intervals)
  - **Server-initiated** - server records for each client the parts of the file that it has cached, notifies client that cached data is invalid when server sees a write for that data

# Stateful vs. Stateless Service

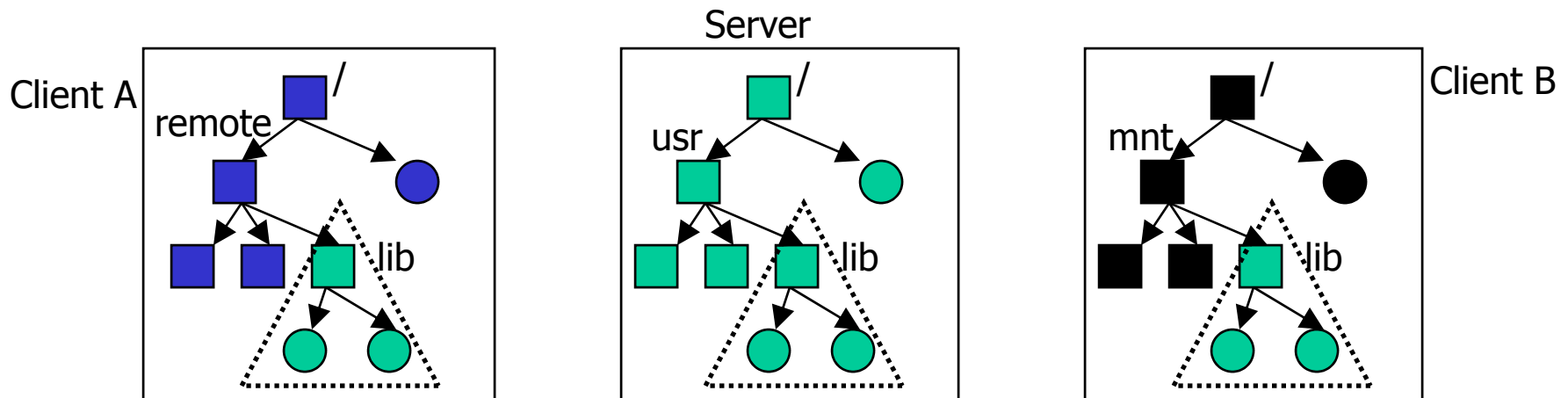
- Server-initiated cache consistency requires server to maintain *state* about each client
  - Server may keep other state, such as result of access permission checks, as well
- Alternative is *stateless* server - each client request is self-contained
- Keeping state improves performance in normal operation, but makes recovering from server crashes more difficult
  - What about effect on scalability?

# Example Dist. File System: NFS

- Sun Microsystems' Network File System, ~1985
- Really, a protocol for remote file access
  - Lookup file (pathname translation)
  - Read directory entries
  - Modify links and directories
  - Get/set file attributes
  - Read/write/create/delete files
- Note no open/close operation!

# Features of NFS

- Naming:
  - NFS servers *export* one or more directories for access by remote clients
  - NFS clients *mount* exported file systems that they want to access, essentially "gluing" the remote file system into the local naming tree



# NFS Client Caching

- Two types of caching at client
  - Attribute cache (metadata - size, time, etc.)
    - Kernel checks with server on open() syscall to see if attributes are still valid
    - Attributes are discarded from cache after 60 seconds
  - Data block cache
    - If attributes still valid at open(), requests can use cached file data blocks
    - Delayed-write strategy is used, with dirty blocks flushed after 30 seconds (tunable value)
    - Dirty blocks are not discarded from client until server confirms write has completed
- What sort of consistency semantics does NFS provide?

# Server-side of accesses

- State
  - Originally designed to be *stateless*
  - For performance, server may keep some state, but not required for correctness
  - server can recover rapidly from a crash
  - All operations must be *idempotent* (repeatable)
- Access
  - Clients obtain a file handle via a lookup request
    - Server checks permissions using a UID provided by the client
  - Subsequent requests pass the file handle to the server
  - File handle must contain enough information to uniquely identify file on server (even if server has crashed and rebooted between the lookup and the request)

# Example Dist. File System: AFS

- Andrew File System, CMU 1985
- Naming:
  - Dedicated servers cooperate to present a shared namespace to the clients
  - Clients perform most of the processing to open a file locally
- Caching:
  - Clients cache whole files on open requests
  - What implications does this have for client file cache sizes?
  - Files are written back to the server on close
  - Server tracks which clients have which files open so invalidations can be issued
    - Known as a *callback* in AFS
- Designed for scalability

# AFS: Shared Name Space

- Basic component is a *volume* (as opposed to disk partition)
  - A single disk partition may hold many volumes
  - Try to store files for a single client in one volume
  - *Volume-location database* records location information on a per-volume basis
    - Replicated at each server
  - AFS directory entries map a file name to a *fid*
    - Volume number, vnode number, uniquifier
    - Location transparent, volumes can move to different servers

# Remote Procedure Calls

- Remote Procedure Call (RPC) is the most common means for remote communication
- It 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
- Someday you will most likely have to write an application that uses remote communication (maybe you already have)
  - You will most likely use some form of RPC for that remote communication
  - So it's good to know how all this RPC stuff works
    - "Debunking the magic"

# Messages

- Initially with network programming, people hand-coded messages to send requests and responses
- 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
  - Could encapsulate messaging into a library
  - Just invoke library routines to send a message
  - Which leads us to RPC...

# 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
  - 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

# Extra Motivation

- From Birrell & Nelson ACM TOCS paper, 1984:

“RPC will, we hope, remove unnecessary difficulties, leaving only the fundamental difficulties of building distributed systems: timing, independent failure of components, and the coexistence of independent execution environments.”

# Remote Procedure Calls

- So, we would like to use procedure call as a model for distributed (remote) communication
- Lots of issues
  - How do we make this invisible to the programmer?
  - What are the semantics of parameter passing?
  - How do we bind (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 Stubs

- A client-side stub is a procedure that looks to the client as if it were a callable server procedure
- A server-side stub looks to the server as if a client called it
- The client program thinks it is calling the server
  - In fact, it's calling the client stub
- The server program thinks it is called by the client
  - In fact, it's called by the server stub
- The stubs send messages to each other to make the RPC happen "transparently"

# RPC Example

## Client Program:

```
...  
sum = server->Add(3,4);  
...
```

## Server Interface:

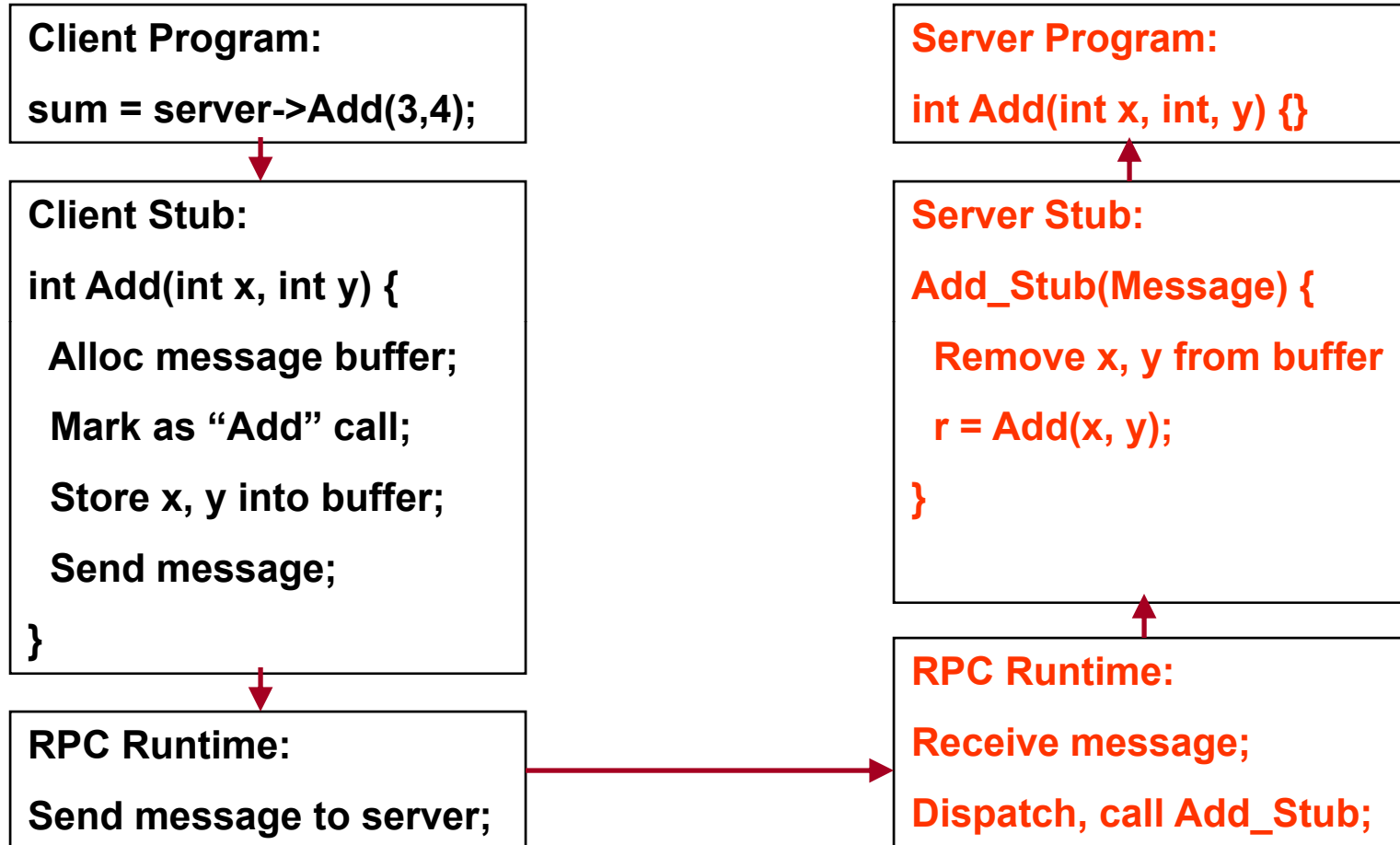
```
int Add(int x, int y);
```

## Server Program:

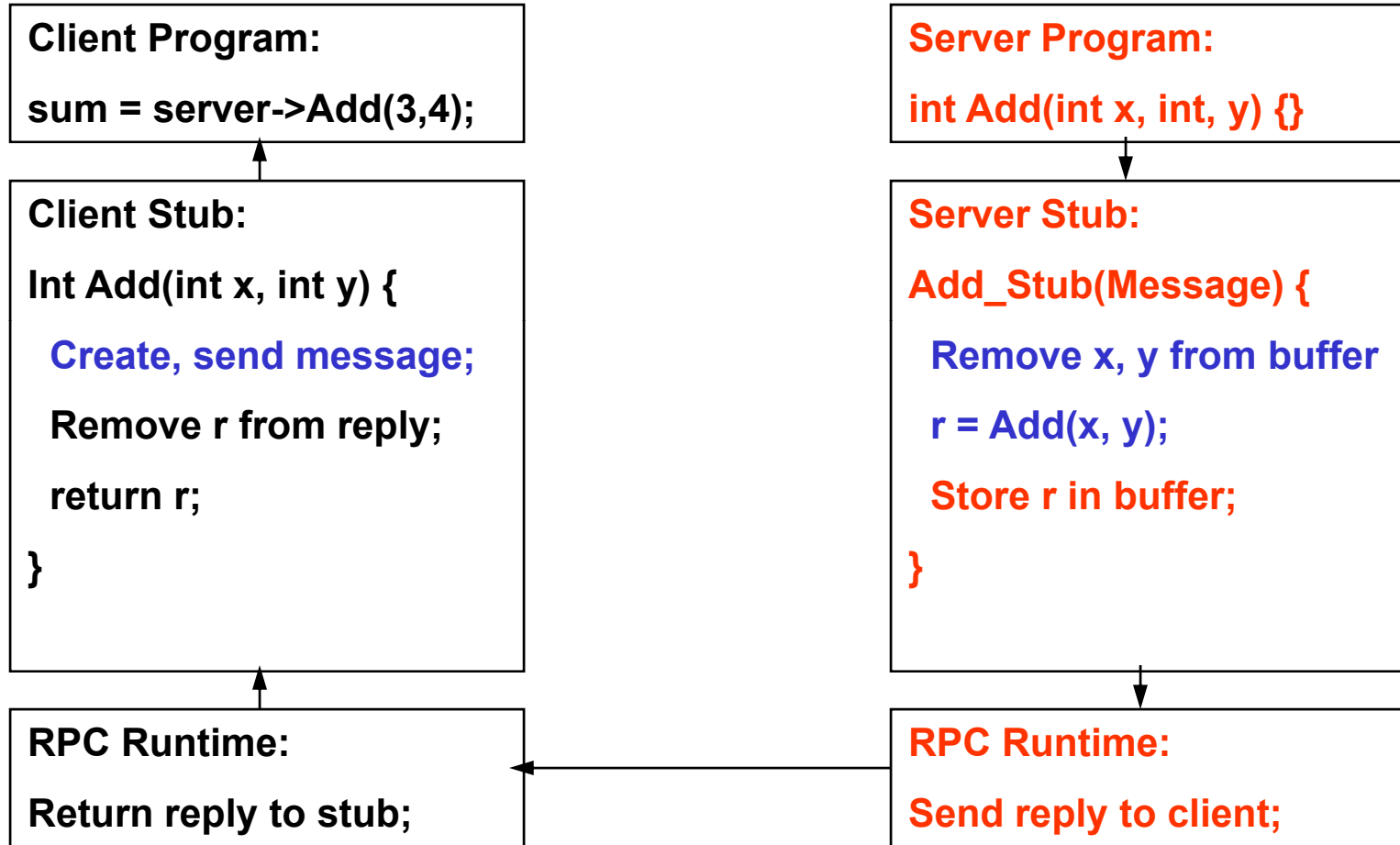
```
int Add(int x, int, y) {  
    return x + y;  
}
```

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

# RPC Example: Call



# RPC Example: Return



# RPC Marshalling

- **Marshalling** is the packing of procedure parameters into a message packet
- The RPC stubs call type-specific procedures to marshal (or unmarshal) the parameters to a call
  - The client stub marshals the parameters into a message
  - The server stub unmarshals parameters from the message and uses them to call the server procedure
- On return
  - The server stub marshals the return parameters
  - The client stub unmarshals return parameters and returns them to the client program

# RPC Binding

- **Binding** is the process of connecting the client to the server
- The server, when it starts up, exports its interface
  - Identifies itself to a network name server
  - Tells RPC runtime its alive and ready to accept calls
- The client, before issuing any calls, imports the server
  - RPC runtime uses the name server to find the location of a server and establish a connection
- The import and export operations are explicit in the server and client programs
  - Breakdown of transparency

# RPC Transparency

- One goal of RPC is to be as transparent as possible
  - Make remote procedure calls look like local procedure calls
- We have seen that binding breaks transparency
- What else?
  - Failures - remote nodes/networks can fail in more ways than with local procedure calls
    - Need extra support to handle failures well
  - Performance - remote communication is inherently slower than local communication
    - If program is performance-sensitive, could be a problem

# 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 essentially 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