Announcements

- P1 teams
  - If you still don’t have a teammate: make use of Piazza posts
  - Don’t wait for too long – let us know soon

- Start P1 early!

15-440/640 Distributed Systems

Distributed File Systems

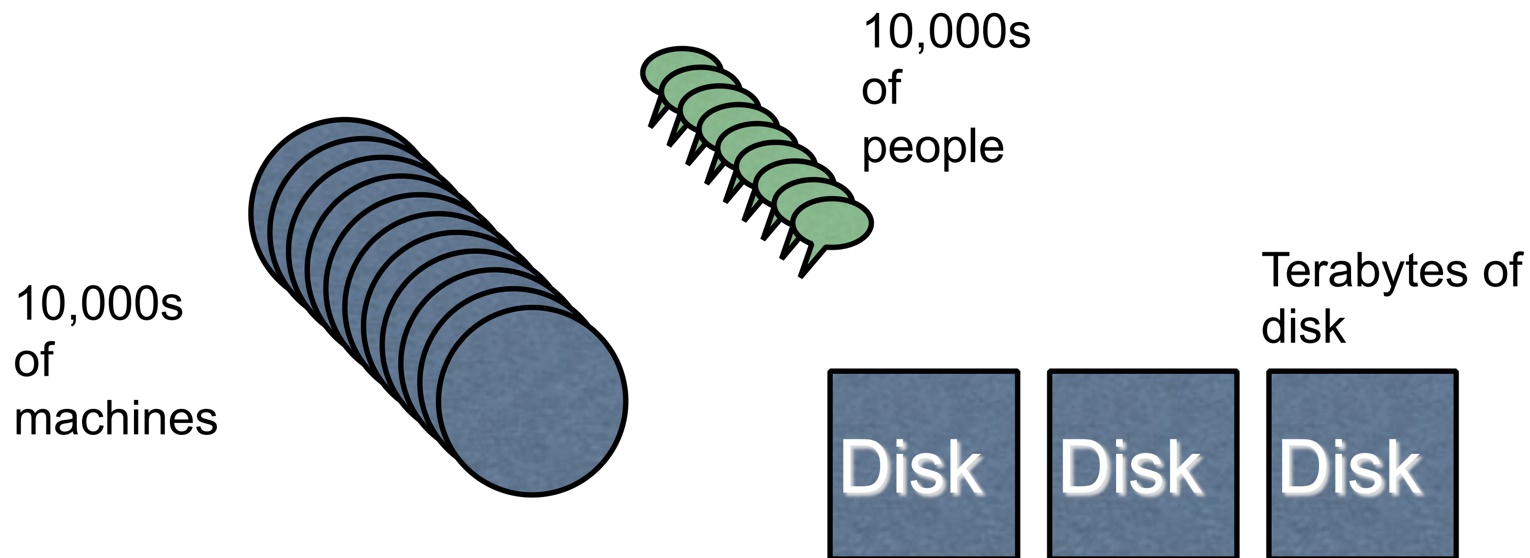
(Two lectures)

Outline

- Why Distributed File Systems?
- Basic mechanisms for building DFSs
  - Using NFS and AFS as examples
- Design choices and their implications
  - Caching
  - Consistency
  - Naming
  - Authentication and Access Control

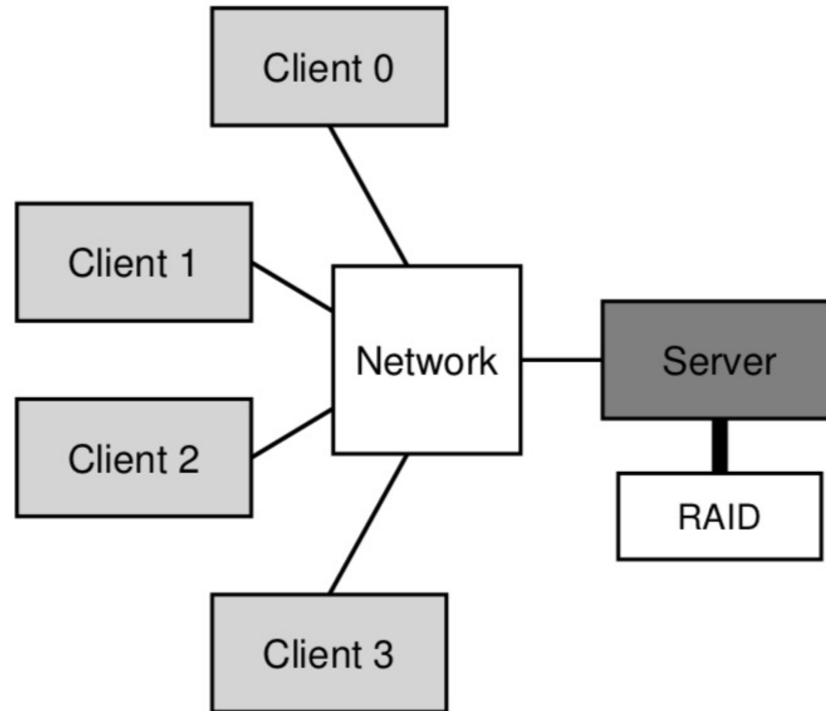
andrew.cmu.edu

- Let's start with a familiar example: andrew



- Have a consistent namespace for files across computers
- Allows any authorized user to access files from any computer

Network File System (NFS)



Picture credit: The Wisconsin OS book

What Distributed File Systems Provide

- Access to data stored at servers using file system interfaces

- What are the file system interfaces?
  - Open a file, check status of a file, close a file
  - Read data from a file
  - Write data to a file
  - Lock a file or part of a file
  - List files in a directory, create/delete a directory
  - Delete a file, rename a file, add a symlink to a file
  - etc

Why are DFSs Useful?

- Data sharing among multiple users
- User mobility
- (File) Location transparency
- Backups and centralized management

Challenges

- Heterogeneity (lots of different computers & users)
- Scale (10s of thousands of users!)
- Security (my files! hands off!)
- Failures
- Concurrency
- oh no... we’ve got ‘em all.

How can we build this??

Prioritized goals? / Assumptions

- Often very useful to have an explicit list of prioritized goals
- Distributed filesystems almost always involve trade-offs

- Scale, scale, scale
- User-centric workloads... how do users use files (vs. big programs?)
  - Most files are personally owned
  - Not too much concurrent access; user usually only at one or a few machines at a time
  - Sequential access is common; reads much more common that writes
  - There is locality of reference (if you've edited a file recently, you're likely to edit again)
- If you change the workload assumptions the design parameters change!

Components in a DFS Implementation

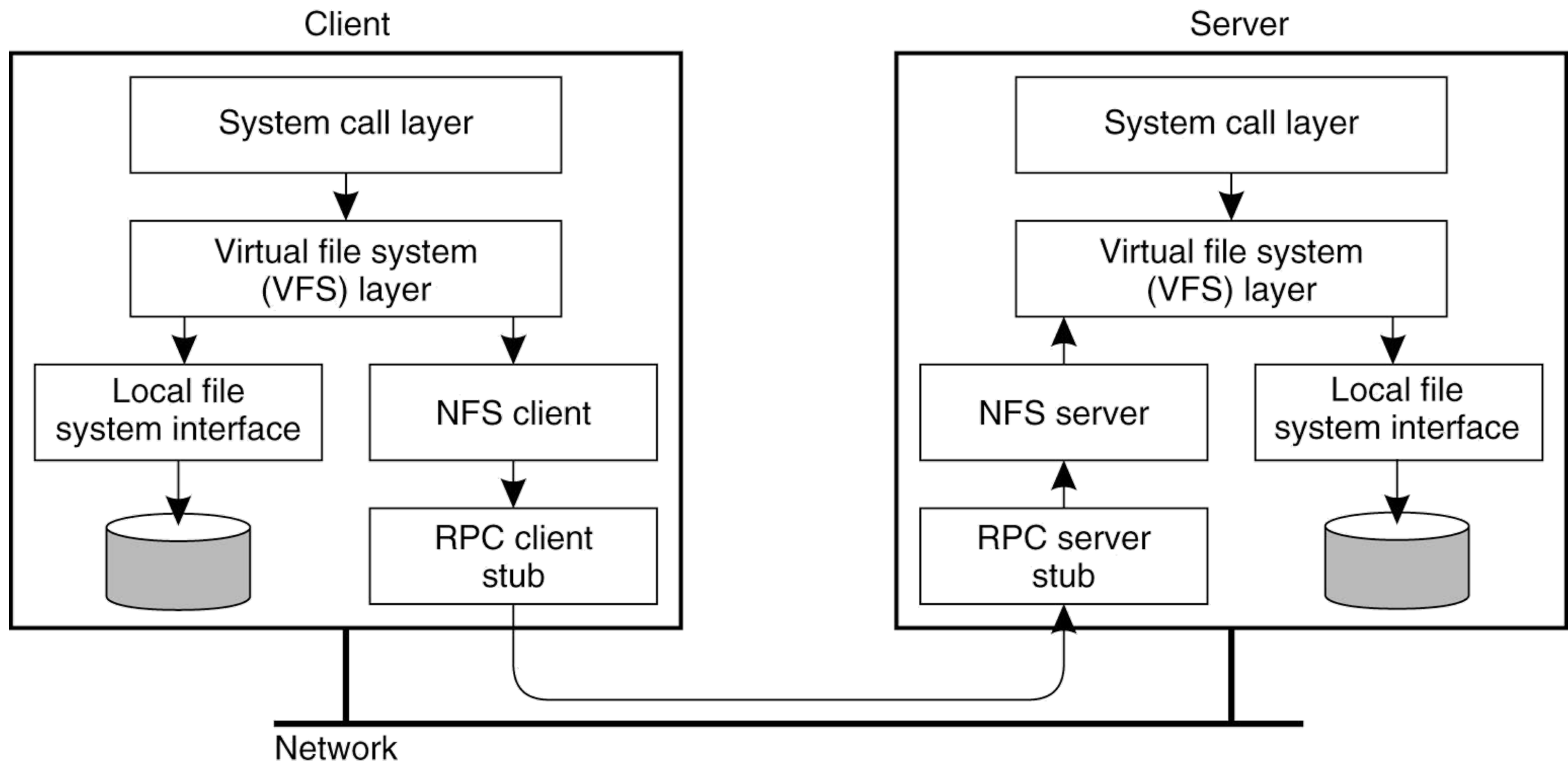
- Client side:
  - What has to happen to enable applications to access a remote file the same way a local file is accessed?
  - Accessing remote files in the same way as accessing local files → requires kernel support

- Communication layer:
  - How are requests sent to server?

- Server side:
  - How are requests from clients serviced?

VFS Interception

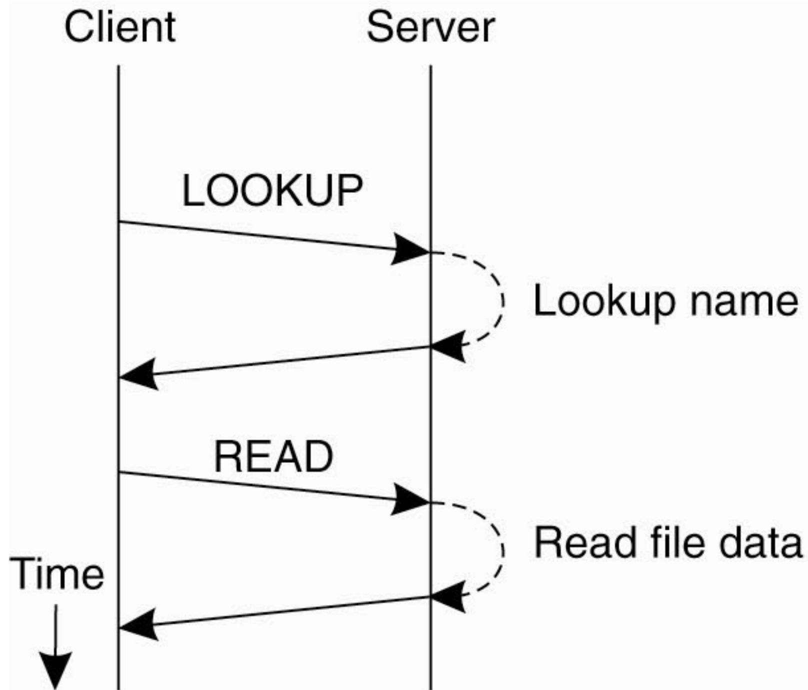
VFS = virtual file system



A Simple Approach

- Use RPC to forward every filesystem operation to the server
  - Server serializes all accesses, performs them, and sends back result.
- Same behavior as if both programs were running on the same local filesystem!

Remote Procedure Calls in NFS



- Reading data from a file
- Lookup takes directory+name and return filehandle

Some NFS v2 RPC Calls

- NFS RPCs using XDR over, e.g., TCP/IP

Proc.	Input args	Results
LOOKUP	dirfh, name	status, fhandle, fattr
READ	fhandle, offset, count	status, fattr, data
CREATE	dirfh, name, fattr	status, fhandle, fattr
WRITE	fhandle, offset, count, data	status, fattr

- fhandle: 32-byte opaque data (64-byte in v3)

Server Side Examples

- **mountd**: provides the initial file handle for the exported directory
  - Client issues nfs_mount request to mountd
  - mountd checks if the pathname is a directory and if the directory should be exported to the client
- **nfsd**: answers the RPC calls, gets reply from local file system, and sends reply via RPC
  - Usually listening at port 2049
- Both mountd and nfsd use underlying RPC implementation

A Simple Approach

- Use RPC to forward every filesystem operation to the server
  - Server serializes all accesses, performs them, and sends back result.
- Great: Same behavior as if both programs were running on the same local filesystem!
- Bad: Performance can stink. Latency of access to remote server often much higher than to local memory.
- In Andrew context: server would get hammered!

A Simple Approach

- Use RPC to forward every filesystem operation to the server
  - Server serializes all accesses, performs them, and sends back result.
- Great: Same behavior as if both programs were running on the same local filesystem!
- Bad: Performance can stink. Latency of access to remote server often much higher than to local memory.
- In Andrew context: server would get hammered!

Lesson 1: Needing to hit the server for every detail **impairs performance and scalability.**

Question: How can we avoid going to the server for everything? What do we lose in the process?

Outline

- Why Distributed File Systems?
- Basic mechanisms for building DFSs
  - Using NFS and AFS as examples
- Design choices and their implications
  - Caching
  - Consistency
  - Naming
  - Authentication and Access Control

Caching

- Client-side caching: Having a copy of the data on client machine
- E.g., AFS, newer versions of NFS

Caching

AFS Goals

- Have a consistent namespace for files across computers
- **LARGE** numbers of clients, servers
  - 1000 machines could cache a single file,
  - Most local, some (very) remote
- Goal: Minimize/eliminate work per client operation

Caching

- AFS uses client-side whole file caching

AFS Assumptions

- Client machines have disks(!!)
  - Can cache whole files over long periods
- Write/write and write/read sharing are rare
  - Most files updated by one user, on one machine

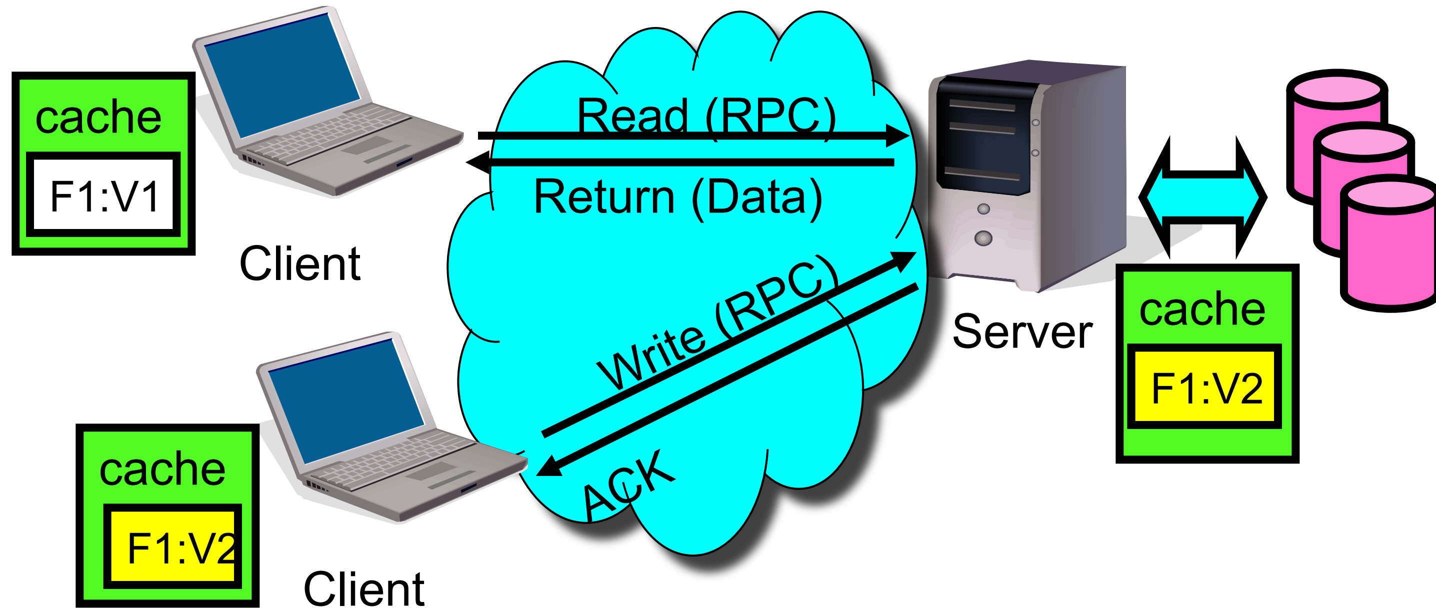
Caching

- So, what do we cache?
- On-demand caching
  - Cache data that was requested by the client
- Pre-fetching
  - Cache data in advance
  - Take latency affects on pre-fetching into account

Caching

read(f1)→V1
read(f1)→V1
read(f1)→V1
read(f1)→?

write(f1)→OK
read(f1)→V2



Caching

- And if we cache... doesn't that risk making things inconsistent?

- Read-only files → easy
- Data that is written by other machines ("cache staleness")→
  - How to know that the data has changed?
  - How to ensure data consistency?
- Data written by the client machine ("update visibility")→
  - When is data written to the server?
  - What happens if the client machine goes down?

Approaches to handle cache staleness

- Update propagation policy
- Lots and lots of research papers on this topic

- We will learn a few key ideas
  - Broadcast invalidations
  - Check on use
  - Callbacks
  - Leases

1. Broadcast invalidations

- When there is an update, every possible cache location is notified
- Used in early (70’s, 80’s, 90’s) multiprocessor caches
- Pros?
  - Simple
  - Can provide strict consistency
- Cons?
  - Can lead to useless network communication
  - Not scalable

2. Check on use

- Client checks with the server before each use
- NFS v2
- Pros?
  - Simple
  - Can provide strict consistency
- Cons?
  - Slow reads
  - Can lead to useless network communication
  - Not scalable: too high load on server

3. Callbacks

- Clients register with server that they have a copy of file
  - Server tells them: "Invalidate!" if the file changes
- This trades server state for improved consistency
- What if server crashes? Lose all callback state!
- Reconstruct callback information from client: go ask everyone "who has which files cached?"
- What if client crashes?
  - Must revalidate any cached content it uses since it may have missed callback
- AFS uses callbacks

4. Leases

- Granting exclusive/shared control of the cached objects for a ***limited amount of time***
- Lease period: duration of the control
- Lease renewal: control expires without renewal
  - Client has to request lease renewal
  - Need to take latency of communication into account

- Read lease vs write lease
  - multiple sites can obtain read lease
  - only one can get write lease

- Clock synchronization
  - Not necessary: absolute time not relevant
  - But clock tick rate matters