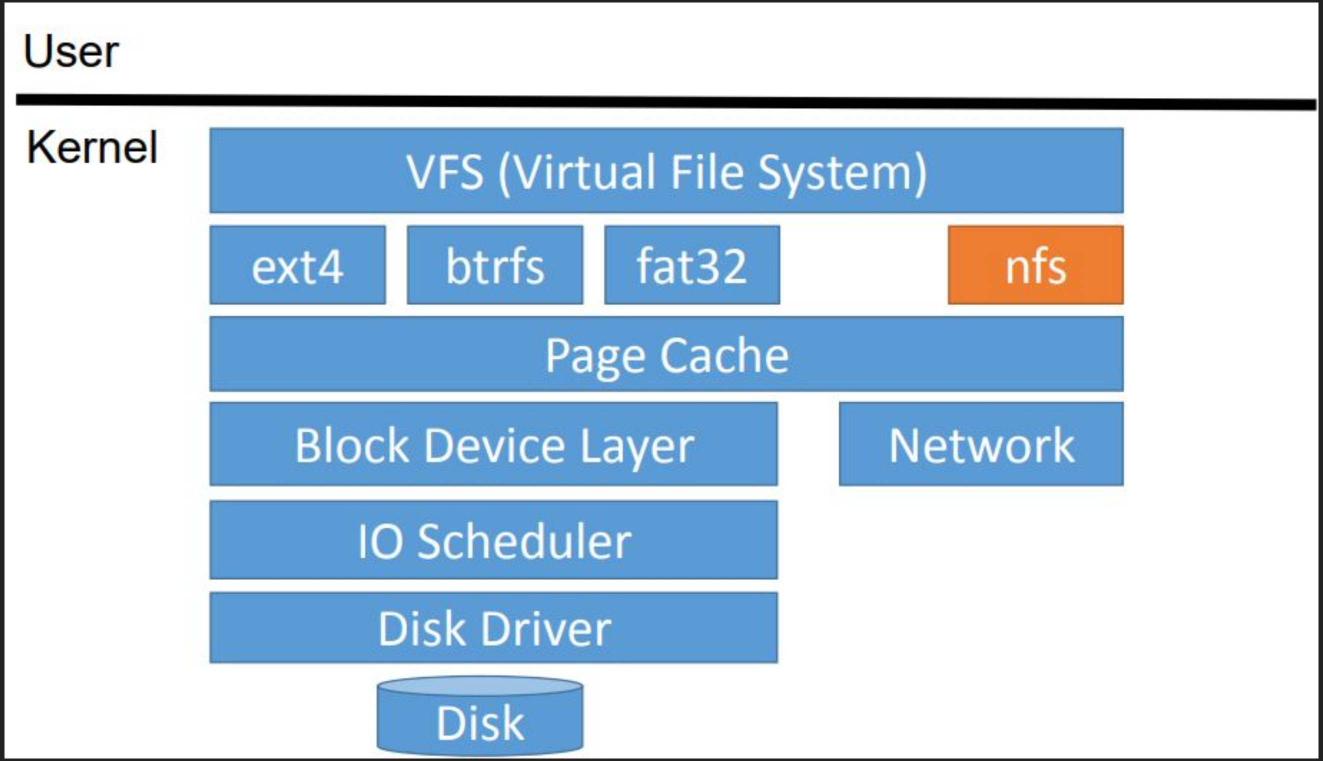
ILLINOIS TECH

College of Computing

CS 450 Operating Systems
Network File System

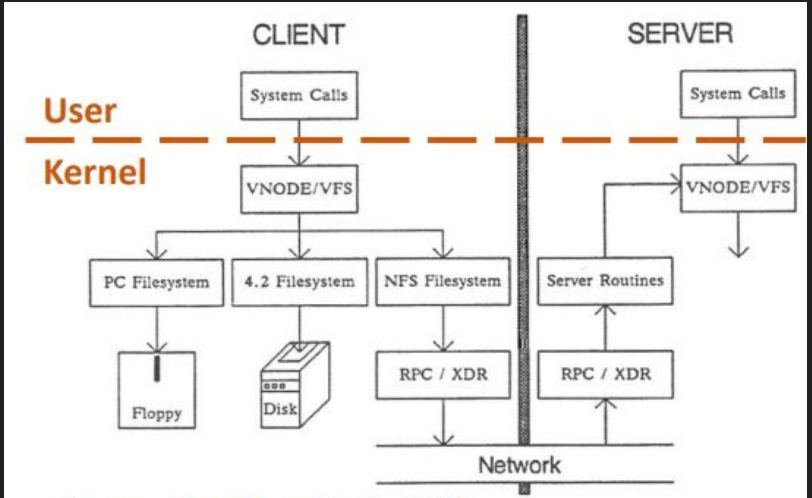
Yue Duan

A Typical Storage Stack (Linux)



Network File System

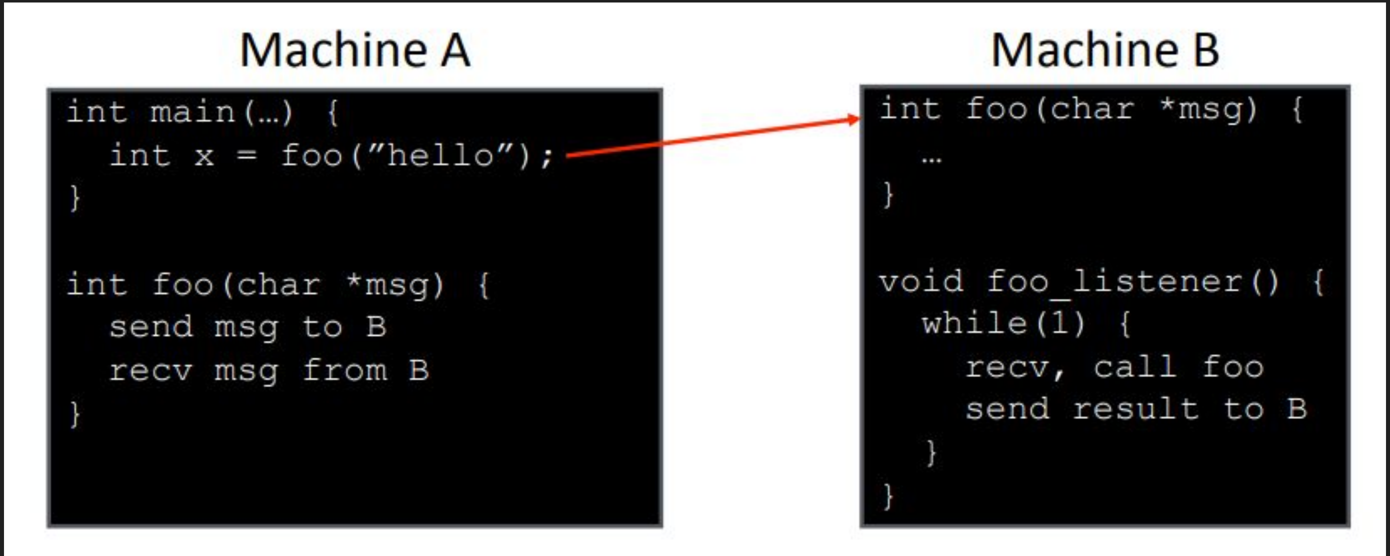
- A client/server system to share the content of a file system over network
- Translate VFS requests into **Remote Procedure Calls** (RPC) to server
  - Instead of translating them into disk accesses



Source: Sandberg et al., 1985

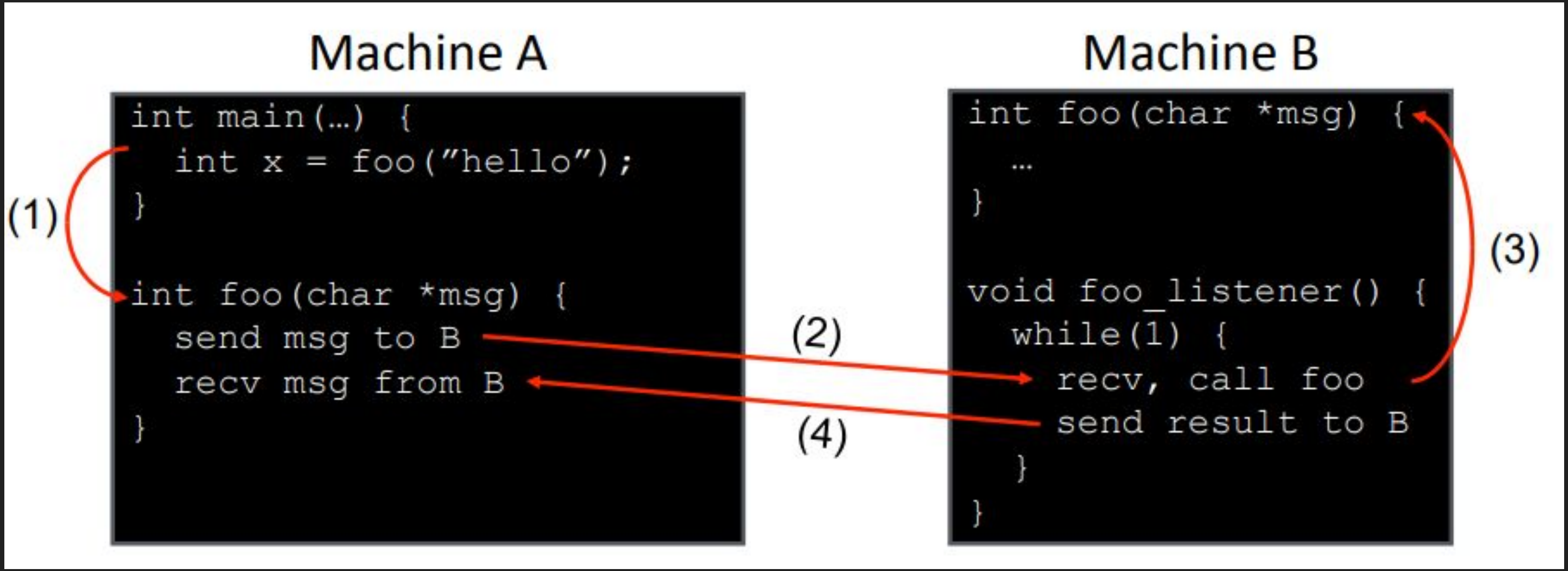
Remote Procedure Call

- Intuition: create wrappers so calling a function on another machine feels just like calling a local function



Remote Procedure Call

- Actual Calls



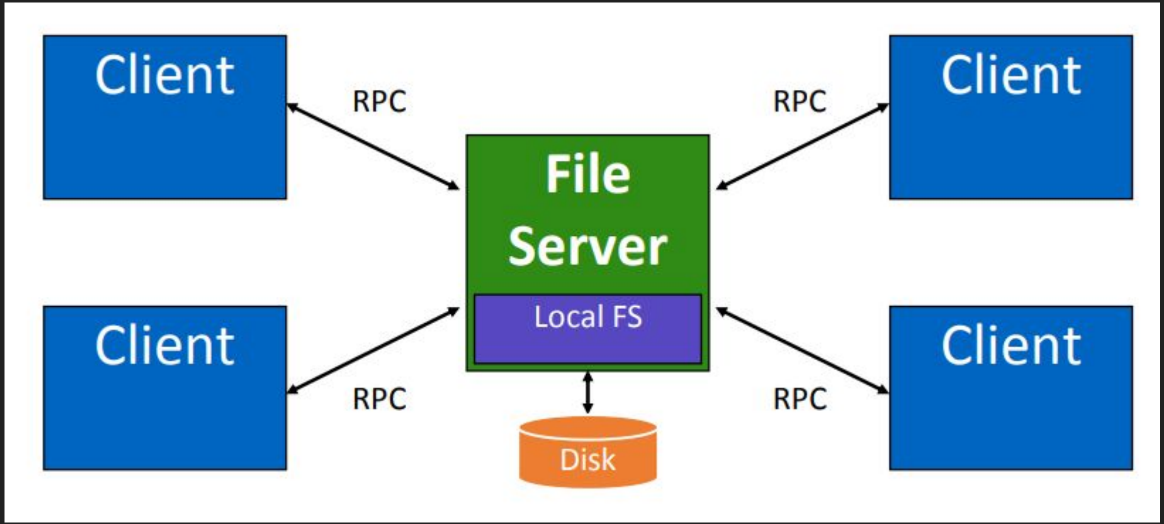
Remote Procedure Call

- There is a pre-assigned **procedure ID** for each remote call
- Client side:
  - 1) Pack procedure ID and all its arguments in an RPC request packet (aka. serialization or marshalling)
  - 2) Send the request to the server
  - 3) Wait for the response
  - 4) unpack results (aka. deserialization or unmarshalling) & return to caller

Remote Procedure Call

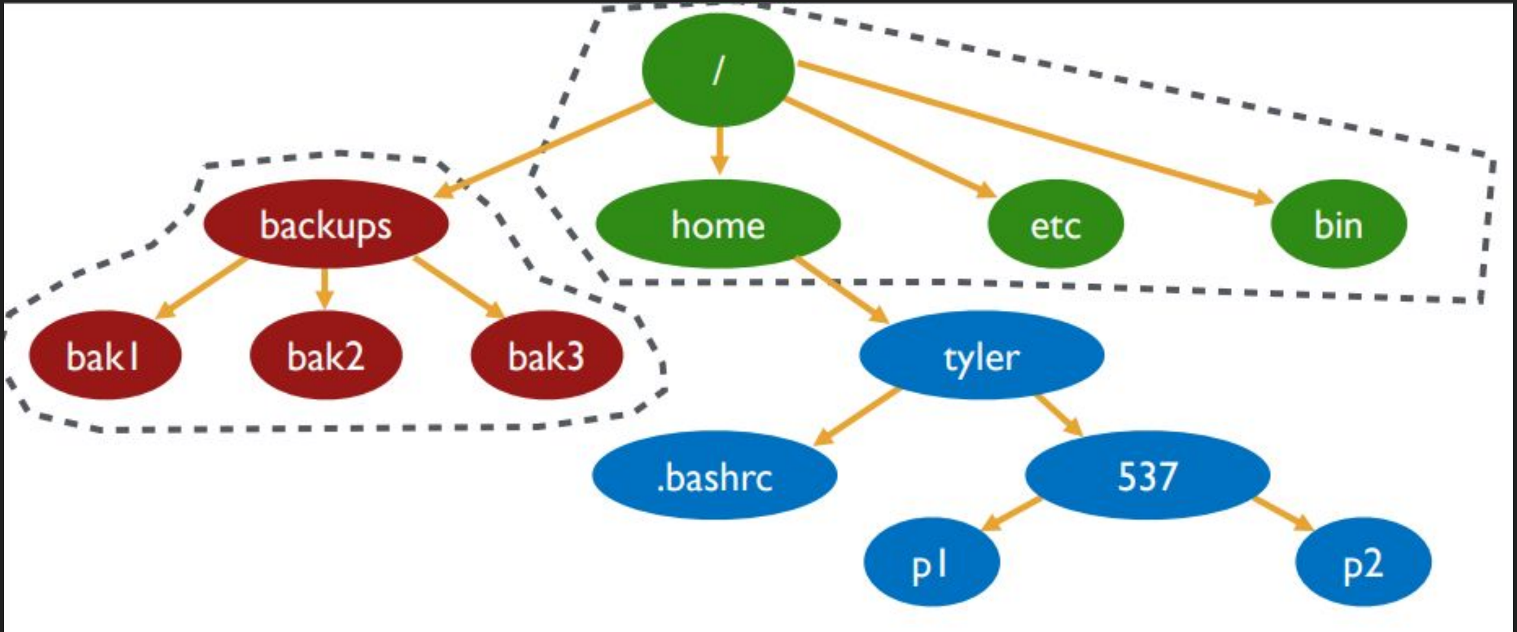
- There is a pre-assigned **procedure ID** for each remote call
- Server side:
  - 1) Wait for and receive the request packet
  - 2) Deserialize the request content (procedure ID and arguments) into appropriate data structures
  - 3) Service the request
  - 4) Serialize results into an RPC response packet and send it to the client

General NFS Architecture



- Server exports the NFS volume
    - Basically assigns a port number to it
- Each client “mounts” the NFS volume somewhere in its directory tree

Example



/dev/sda1 **on** /

/dev/sdb1 **on** /backups

NFS **on** /home

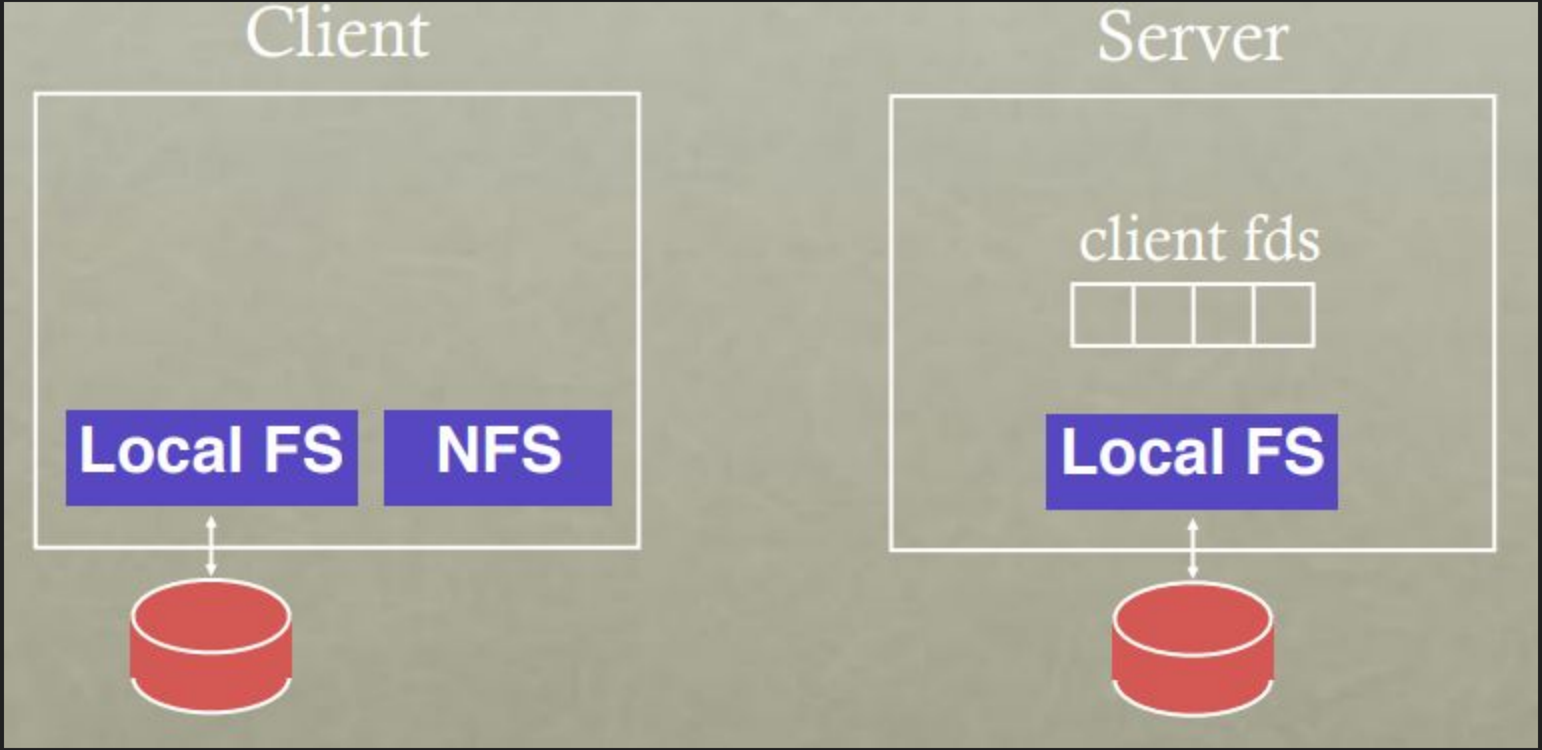
Challenges of NFS Protocol Design

- Both server or client can crash (i.e., lose state)
- Server and client can be temporarily disconnected (e.g., lost or corrupted packets)
- How to coordinate multiple clients actions?
  - Client-side caching
  - inode reuse
- Buffering writes in the server

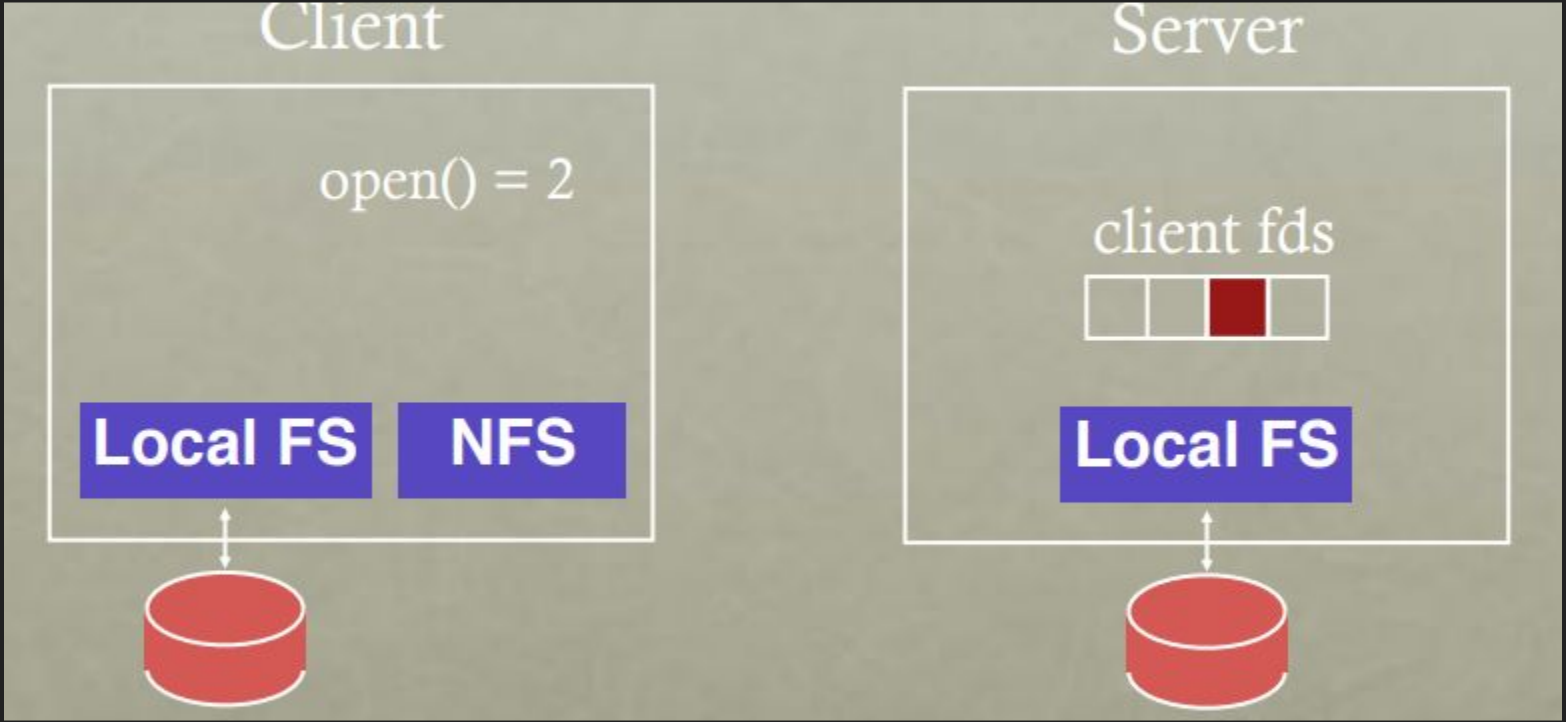
Protocol Design

- Attempt:
  - Wrap regular UNIX system calls using RPC
- open()
  - open() on client calls open() on server
  - open() on server returns fd back to client
- read(fd)
  - read(fd) on client calls read(fd) on server
  - read(fd) on server returns data back to client

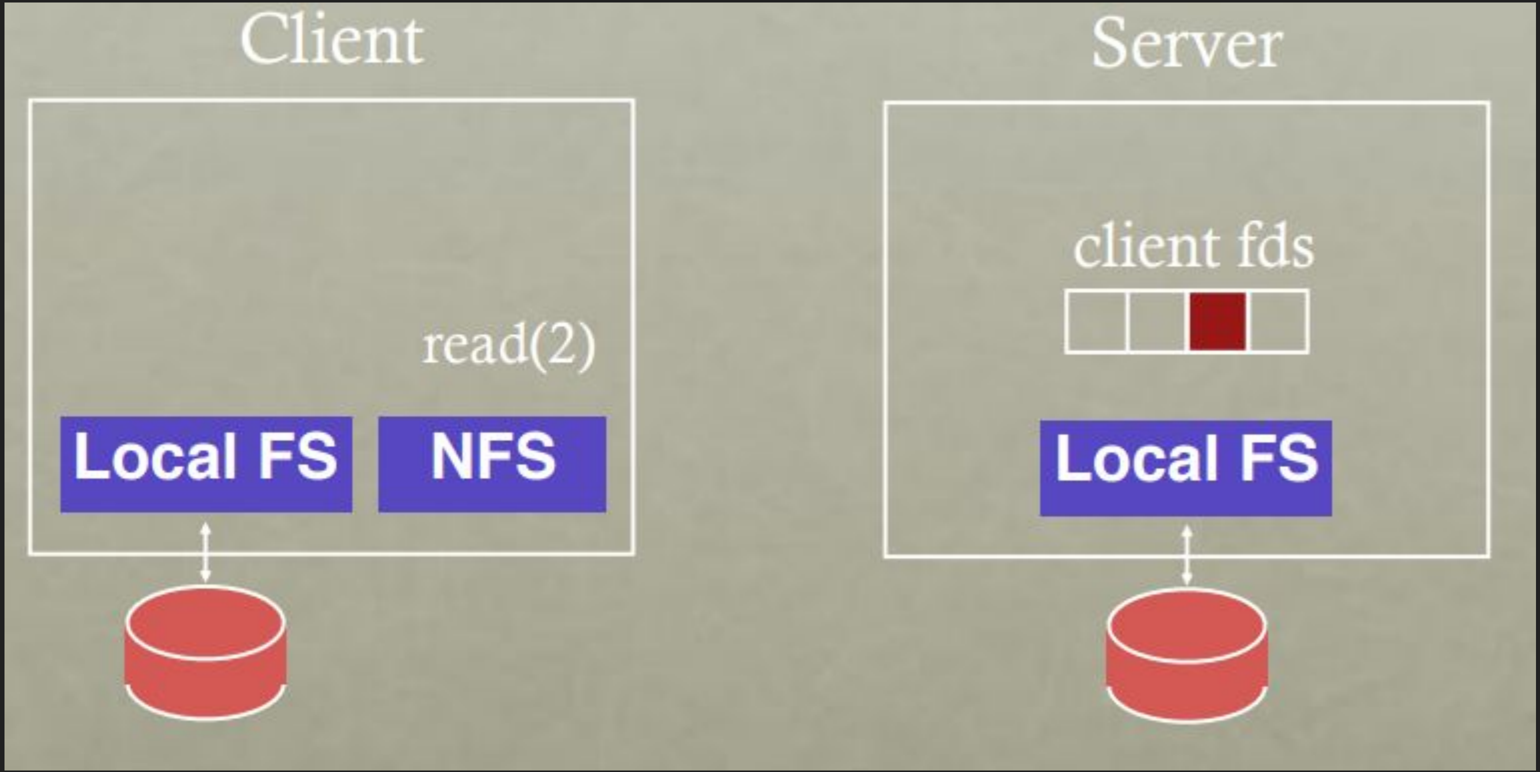
File Descriptors



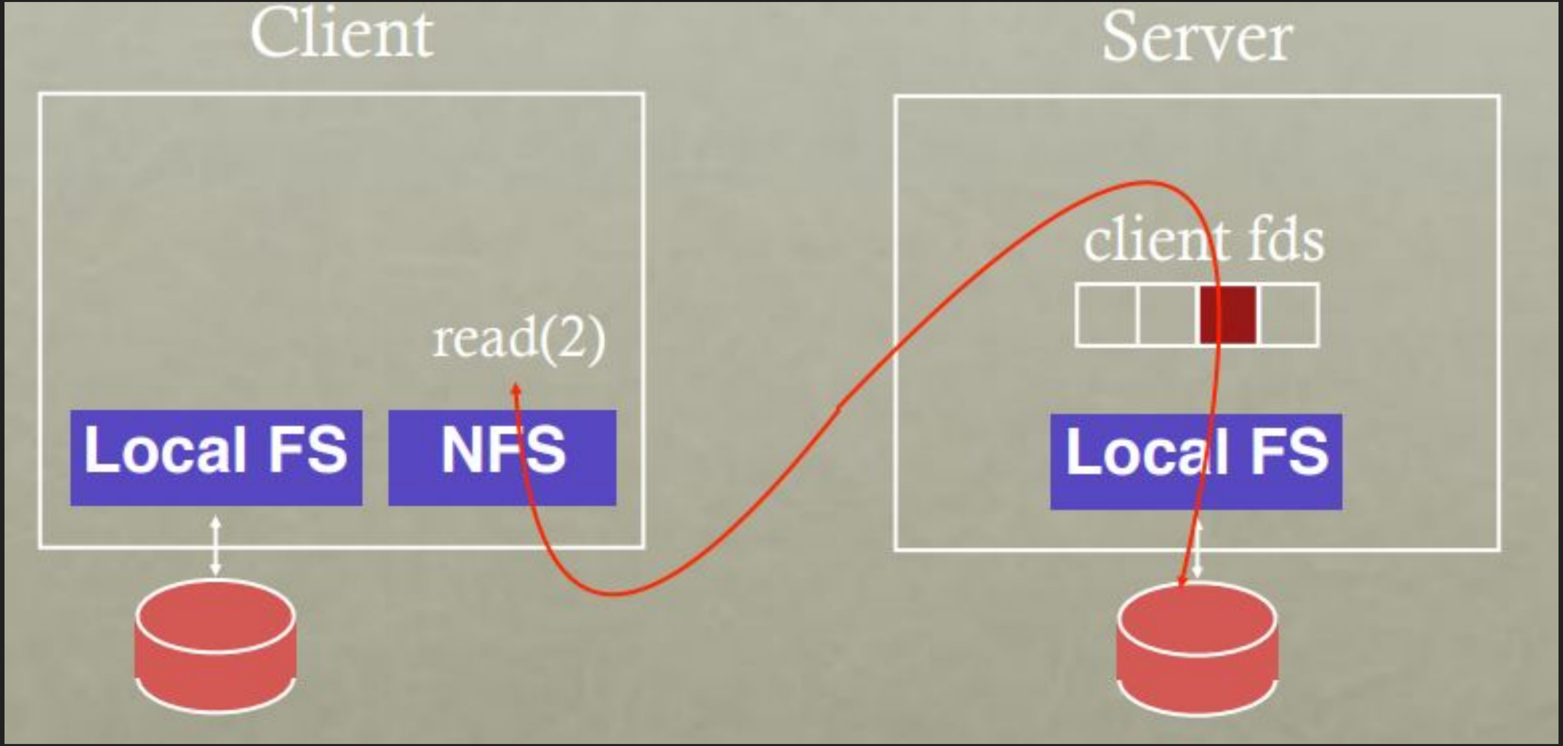
File Descriptors



File Descriptors



File Descriptors



Challenge 1: Dealing w/ Crashes

- What about crashes?

```
int fd = open("foo", O_RDONLY);
read(fd, buf, MAX);
read(fd, buf, MAX);
…
read(fd, buf, MAX);
```

Server crash!

nice if acts like a slow read

- Imagine server crashes and reboots during reads…

Solution: Put All Info in Requests

- Stateful vs. Stateless Protocols
  - **Stateful protocol**: server keeps track of past requests and client states
    - i.e., state persist across requests on the server
    - e.g., keep track of open files and their cursor by each client
  - **Stateless protocol**: server **does not** keep track of past requests
    - client should send all necessary state with a single request
    - e.g., server does not keep track of a client's open file cursor
- NSF uses "stateless" protocol
  - server maintains no state about clients
  - server still keeps other state, of course

Put All Info in Requests

- "Stateless" protocol: server maintains no state about clients
- Need API change. One possibility:
  - **pread**(char *path, buf, size, offset);
  - **pwrite**(char *path, buf, size, offset);
- Specify path and offset each time.
- Server need not remember anything from clients.
- Pros? Cons?
  - Server can crash and reboot transparently to clients.
  - Too many path lookups.

Put All Info in Requests

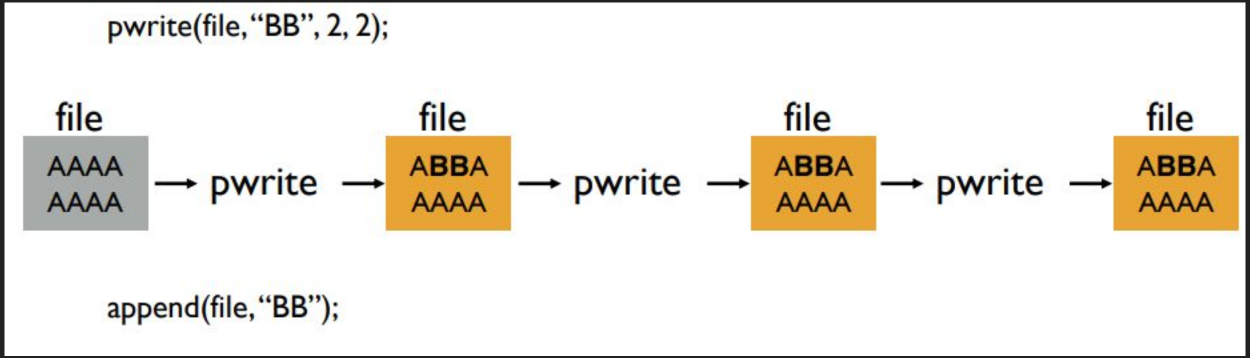
- Every request sends all needed info
  - user credentials (for security checking)
  - file handle and offset
- File operations
  - fh = open(char *path);
  - pread(fh, buf, size, offset);
  - pwrite(fh, buf, size, offset);
- File Handle = <volume ID, inode #, **generation #**>

Challenge 2: Request Timeouts

- Request sent to NFS server, no response received
  - 1) Did the message get lost in the network (UDP)?
  - 2) Did the server die?
  - 3) Is the server slow?
  - 4) Is the response lost or in transit?
- Client has to retry after a timeout
  - okay if (1) or (2)
  - potentially doing things twice if (3) or (4)
  - **problem**?

Can NFS Protocol include Append?

- File operations
  - fh = open(char *path);
  - pread(fh, buf, size, offset);
  - pwrite(fh, buf, size, offset);
  - **append**(fh, buf, size);
- Problem with append()?



Challenge 2: Request Timeouts

- Idea: Make all requests **idempotent**
  - requests should have **same** effect when executed multiple times
- Some requests not easy to make idempotent
  - e.g., deleting a file, making a directory, etc.
  - partial remedy:
    - server keeps a cache of recent requests and ignores duplicates

Challenge 3: inode Reuse

- Process A opens file ‘foo’
  - maps to inode 30
- Process B unlinks file ‘foo’
  - on client, OS holds reference to the client inode alive
  - NFS is stateless, server doesn’t know about open handle
    - the file can be deleted and the server inode reused
    - next request for inode 30 will go to the wrong file
- Idea: **generation #** as part of file handle
  - if server inode is recycled, generation number is incremented

Challenge 4: Client-Side Caching

- Client-side caching is necessary for high-performance
  - otherwise, for every user FS operation, we'll have to go to the server
  - can cause consistency issues when there are multiple copies of data
- Example:
  - Clients A and B have file in their page cache
  - Client A writes to the file
    - data stays in A's cache
    - eventually flushed to the server
  - Client B reads the file
    - Does B see the old content or the new stuff?
    - Who tells B that the cache is stale?
      - server could tell, but only after A actually wrote/flushed the data
      - but this would make the protocol stateful — bad idea!

Consistency/Performance Tradeoff

- Performance: cache always, write when convenient
  - other clients can see old data, or make conflicting updates
- Consistency: write everything to server immediately
  - and tell everyone who may have it cached
    - requires server to know the clients which cache the file (stateful)
  - much more network traffic, lower performance
  - **not good for the common case: accessing an unshared file**

Compromise: Close-to-Open Consistency

- NFS Model: Close-to-Open consistency
- On **close**(), flush all writes to the server
- On **open**(), ask the server for the current timestamp to check the cached version's timestamp
  - if stale, invalidate the cache
  - makes sure you get the latest version on the server when opening a file

Challenge 5: Removal of Open Files

- Recall: Unix allows accessing deleted files if still open
  - reference in in-memory inode prevents cleanup
  - applications expect this behavior; how to deal with it in NFS?
- On client, check if file is open before removing it
  - if yes, rename file instead of deleting it
    - .nfs* files in modern NFS
  - when file is closed, delete temp file
    - if client crashes, garbage file is left over
  - only works if the same client opens and then removes file

Challenge 6: Time Synchronization

- Each CPU’s clock ticks at slightly different rates
  - these clocks can drift over time •
- Tools like ‘make’ use file timestamps
  - clock drift can cause programs to misbehave
  - make[2]: warning: Clock skew detected. Your build may be incomplete.
- Systems using NFS must have clocks synchronized
  - using external protocol like Network Time Protocol (NTP)
    - synchronization depends on unknown communication delay
    - very complex protocol but works pretty well in practice

Challenge 7: Security

- Local UID/GID passed as part of the call
  - UIDs must match across systems
  - yellow pages (yp) service; evolved to NIS
  - replaced with LDAP or Active Directory
- Problem with “root” (User ID 0) : root on one machine becomes root everywhere
- Solution: root squashing – root (UID 0) mapped to “nobody”
  - ineffective security
    - malicious client, can send any UID in the NFS packet

THANK YOU!