NSF

How to build a distributed file system?

COM S 352

Iowa State University

Matthew Tancreti

Sun's Network File System (NFS)

"the network is the computer" - John Gage, Sun's fifth employee, in 1984

NFS defines an open standard client/server protocol for making a distributed file system

Clients exist for many platforms Unix, Linux, MacOS, Windows…

Server can be built on top of most traditional file systems

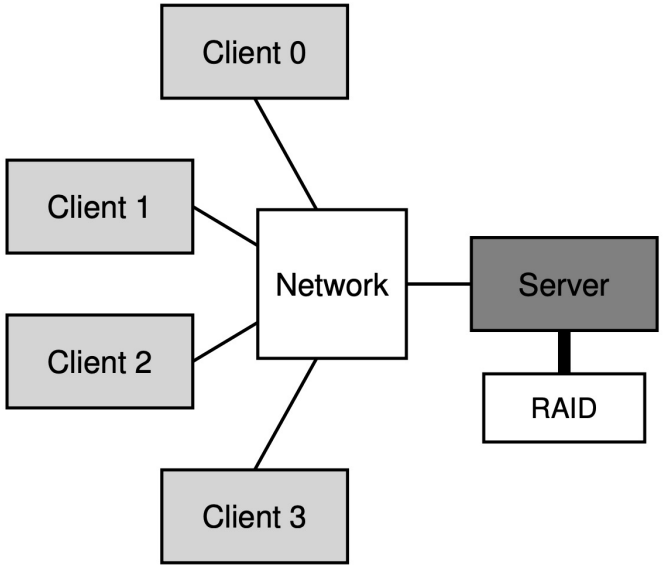
How to build a distributed file system?

Co-founders of Sun Microsystems [source]

Generic Distributed File System

Advantages

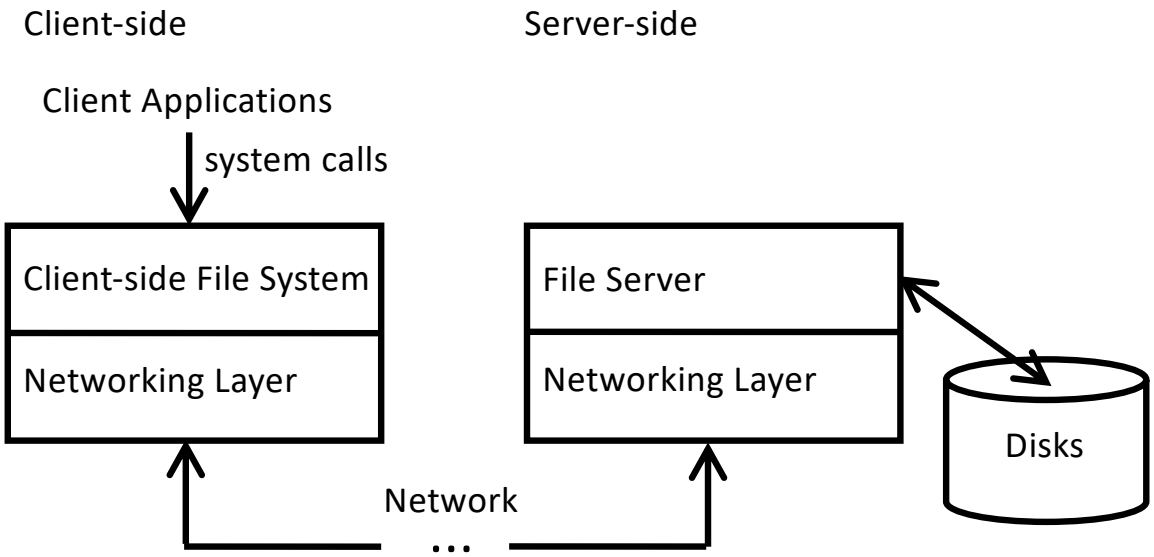
- Provide easy sharing of data across clients
- Centralized administration of file system



Architecture

Distinguish between **client-side** and **server-side** components

System call interface is identical to local file systems, provides client with **transparent** access



Stateful Server

A **stateful** server means the server maintains information about the client's state (e.g., what files the client has open, location (lseek) of next read…), a server without client stat is **stateless**

Example of stateful server

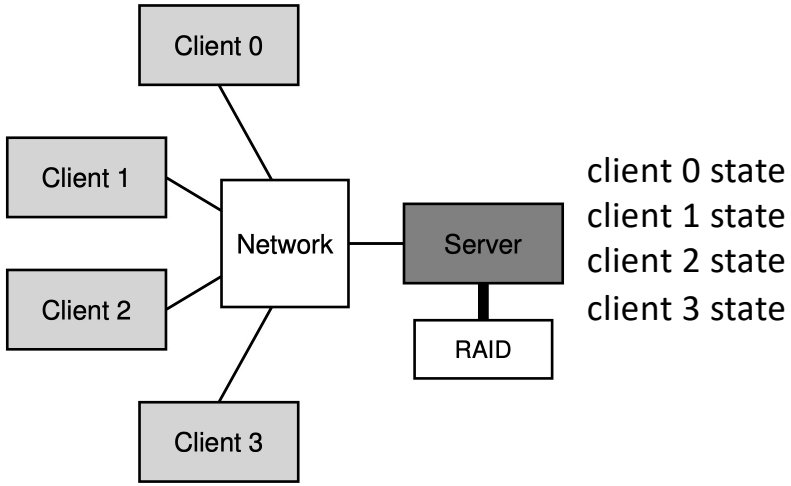
Application	Server State
int fd = open("foo", O_RDONLY);	fd: inode=x, open_instace_cout=1, lseek=0…
read(fd, buffer, 1);	fd: inode=x, open_instace_cout=1, lseek=1…
read(fd, buffer, 1);	fd: inode=x, open_instace_cout=1, lseek=2…
read(fd, buffer, 1);	fd: inode=x, open_instace_cout=1, lseek=3…
close(fd);	fd: valid=false

What is Wrong with Stateful Server?

What happens when server crashes?

Need to reestablish connections with all clients and request their state to rebuild the complete state on server

Slow recovery, extra complexity for client



Fast Server Crash Recovery

In distributed system we expect servers to crash, design goal is for server to **recover** quickly

What if server is stateless? All recovery is local

NFS design is a **stateless protocol**, server doesn't remember anything from previous client requests

Client/Server Protocol

A **file handle** has *volume identifier*, *inode number* and *generation number*

Generation number identifies the version of the inode

Common Commands sent from client to server

- LOOKUP – obtain file handle
- READ – read from file at specified location a number of bytes
- WRITE – write to file at specified location a number of bytes
- GETATTR – get the attributes for a file (e.g., time of last modify)

Generation Number

Unix filesystems often allow reusing inode numbers after a file has been deleted

Consider

- Client0 gets file handle for “/foo.txt” which has inode=100
- Client1 also gets file handle for “/foo.txt”
- Client1 sends command to delete “/foo.txt”
- Client1 opens a new file “/bar.txt” which has the reused inode=100
- Client0 writes using file handle with inode=100 (it is writing to the wrong file!)

Opening a File

Application calls open resulting in client sending LOOKUP

Server replies with a file handler (FH)

Client	Server
fd = open("/foo", ...);	
Send LOOKUP (rootdir FH, "foo")	
	Receive LOOKUP request look for "foo" in root dir return foo's FH + attributes
Receive LOOKUP reply allocate file desc in open file table store foo's FH in table store current file position (0) return file descriptor to application	

Reading a File

read(fd, buffer, MAX);
Index into open file table with fd
get NFS file handle (FH)
use current file position as offset
Send READ (FH, offset=0, count=MAX)

Receive READ request
use FH to get volume/inode num
read inode from disk (or cache)
compute block location (using offset)
read data from disk (or cache)
return data to client

Receive READ reply
update file position (+bytes read)
set current file position = MAX
return data/error code to app

---

read(fd, buffer, MAX);
Same except offset=MAX and set current file position = 2*MAX

---

read(fd, buffer, MAX);
Same except offset=2*MAX and set current file position = 3*MAX

Closing File

close(fd);

Just need to clean up local structures
Free descriptor "fd" in open file table
(No need to talk to server)

Retry on Failure

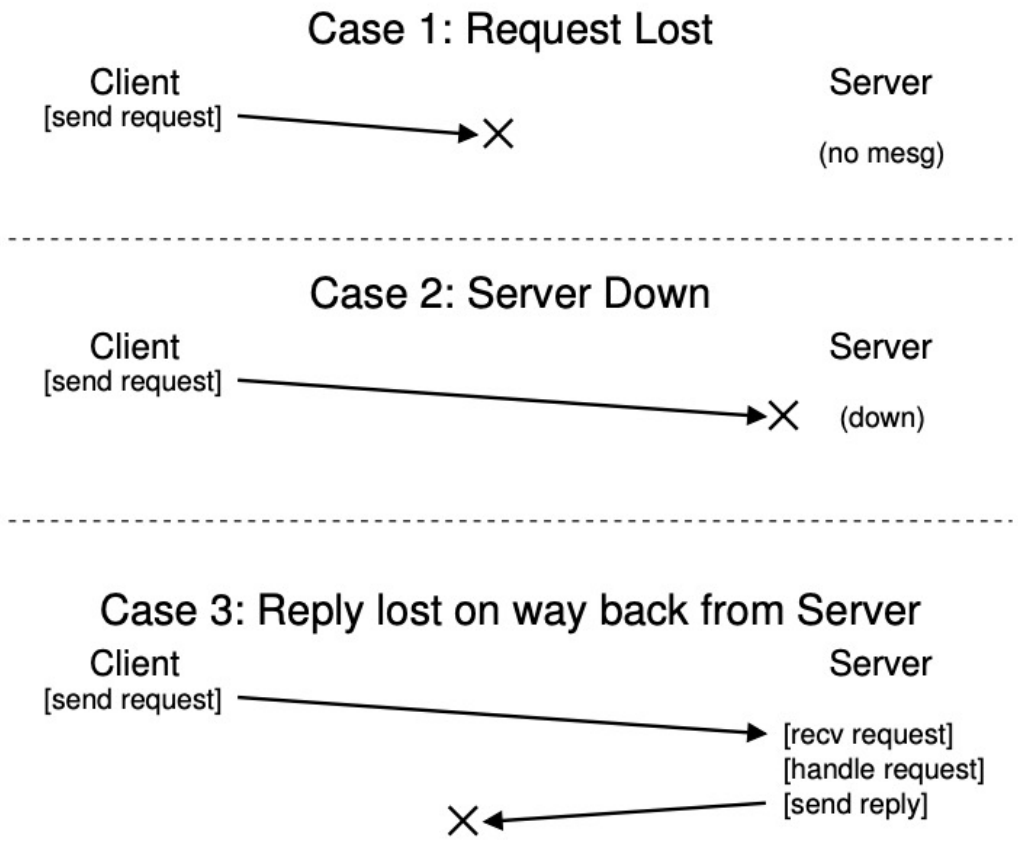
What to do when there is not a quick response from the server?

- network failure
- server crashed and is rebooting
- server is under heavy load and is slow completing the operation

Could implement a sophisticated recovery protocol, however, simple solution is client **retires** the request after a timeout

But, if client resends message server might end up performing operation twice (e.g., server will append to file two times)!

Three Types of Message Loss



Idempotency

Idempotency is the principle that performing an operation multiple times is equivalent to the effect of performing the operation a single time

Which of the following are idempotent operations?

- LOOKUP
- READ
- WRITE
- APPEND
- GETATTR

Idempotency

Which of the following are idempotent operations?

- LOOKUP
- READ
- WRITE
- ~~APPEND~~ not an actual NFS command because it is not idempotent
- GETATTR

NFS only has idempotent commands

Not perfect, as consequence appending must be performed by multiple operations (e.g., GETATTR to determine file size and WRITE to write past end of file) which is not atomic

NFS sacrifices correctness of some corner cases in the name of simplicity, efficiency and salability

Client-Side Caching

Sending every read and write request over network has big performance penalty, orders of magnitude slower than a local file system

Locality observed in typical file accesses, therefore obvious solution is to add a **cache** on the client

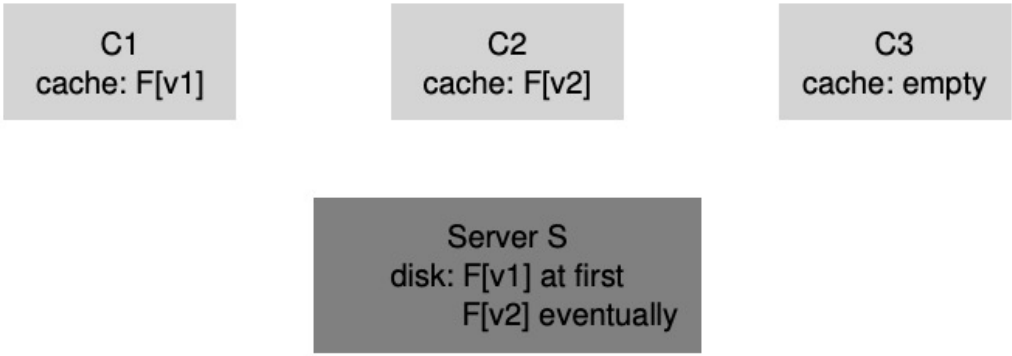
Recently accessed file data is keep it client cache so it can be quickly read again

Write buffering means write goes to cache first and then later the changes are pushed to the server

Advantage: client responds quickly to a write system call, doesn't need to block application for network operation

Cache Consistency Problem

Big problem: **cache consistency**



Example 1:

C1 reads file F

C2 overwrites file F

C3 reads file F

What version of F does C3 get?

When client can't get most recent version of file from server it is an **update visibility** cache consistency problem

Example 2:

C1 reads file F

C2 overwrite file F

C2 flushes cache to the server

C1 reads again from file F

What version of F does C1 read the second time?

When client reads from out-of-date cache it is a **stale cache** consistency problem

Addressing Update Visibility

Flush-on-close semantics means cache is always flushed when the application closes a file

Ensures that subsequent opens from another node will see the latest file version

Not perfect solution, update visibility problem still exists, but is mitigated for common file usage patterns

Addressing Stale Cache

Check if file has changed before using cached contents of file

The GETATTR command will indicate time of last modification to file

Results in a flood of GETATTR commands, solution is to add a local **attribute cache** that updates contents only after a timeout