CS 423
Operating System Design: Persistence: NFS/AFS
05/02

Ram Alagappan

NFS Distributed File System

NFS: more of a protocol than a particular file system

Many companies have implemented NFS: Oracle/Sun, NetApp, EMC, IBM

Recap

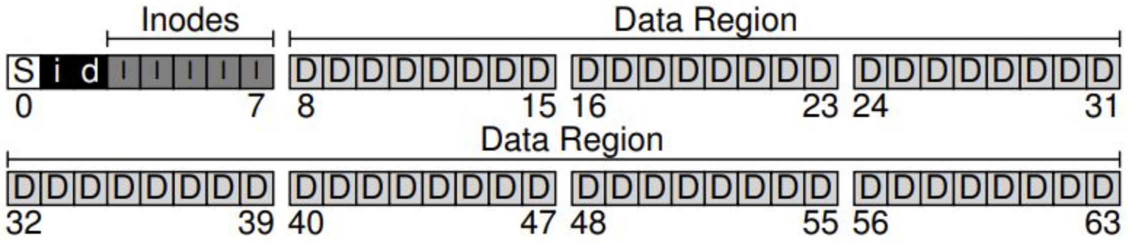
So far, local file systems:

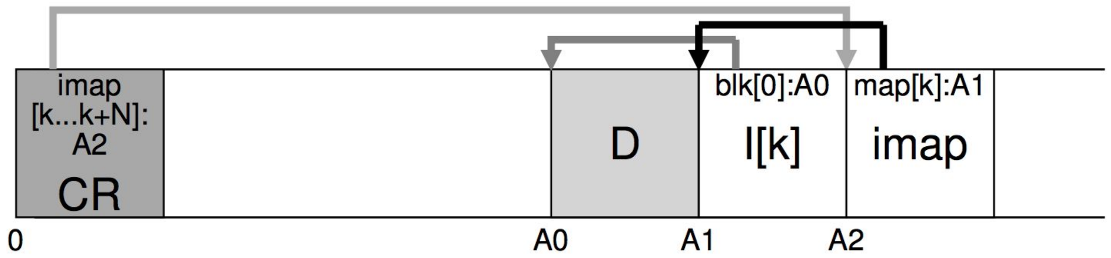
VSFS – in-place update file system

LFS – always append to the end of the log

Everything including the inode, imap, data blocks live on the log

The log is the file system…


Inodes
Data Region
S
i
d
I
D
0
7
8
15
16
23
24
31
Data Region
32
39
40
47
48
55
56
63


imap [k...k+N]: A2
CR
D
blk[0]:A0
I[k]
map[k]:A1
imap
0
A0
A1
A2

Questions (2 min)

What is the biggest downside of LFS over VSFS (or in general in-place update file systems?)

What is one workload that will perform almost the same in both?

Questions (2 mins)

Can you think of one workload where LFS will be much better than VSFS?

Can you think of one workload where VSFS will be much better than LFS?

What is a useful feature that LFS can provide that VSFS cannot (at least efficiently)?

End Recap

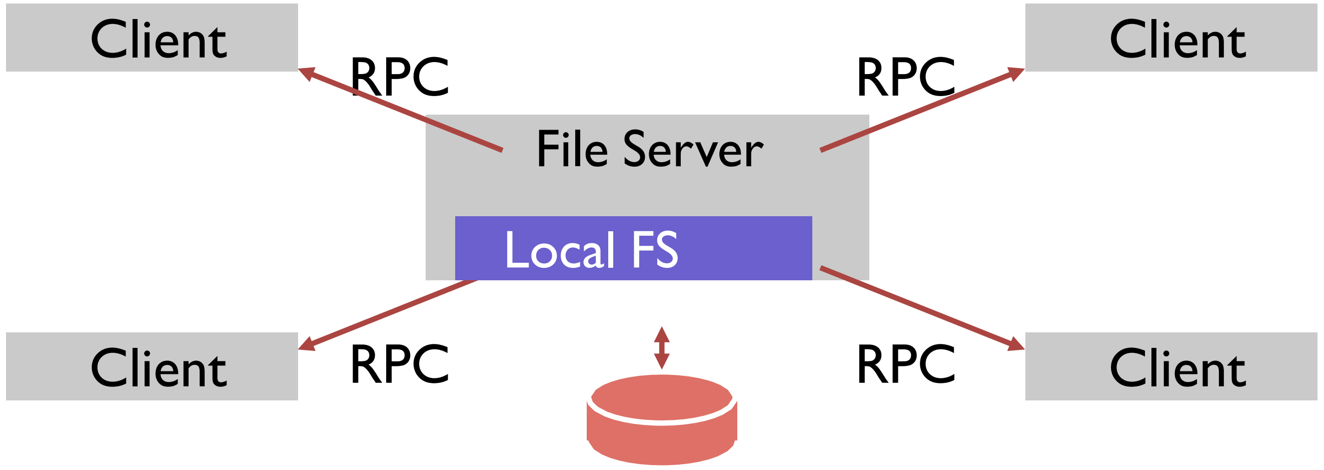
Today:

Switching from local FS to distributed file systems

Will look into two popular ones:

- NFS (network FS)
- AFS (Andrew FS)

NFS Arch


Client
Client
RPC
RPC
File Server
Local FS
Client
RPC
RPC
Client

Benefits?

Sharing across machines

Central admin

Goal: Simple Server Crash Recovery

Why do servers crash?

Goal: Simple Crash Recovery

Strategy-1: server returns fd upon open, client passes fd on each call

```
intfd =  open(“foo”, O_RDONLY);
read(fd,      buf,    MAX);
read(fd,      buf,    MAX);
…
read(fd,      buf,    MAX);
```

Goal: Simple Crash Recovery

Strategy-1: server returns fd upon open, client passes fd on each call

```
intfd =  open(“foo”, O_RDONLY);
read(fd,     buf,   MAX);
read(fd,     buf,   MAX);
…
read(fd,     buf,   MAX);
```

Server crash!

Problems

Complicates crash recovery.

Server crash – what happens? What must client do?

Client crashes – what happens?

General idea: Statelessness

Server keeps no state: no fd □ file map, no file position pointer

Server doesn’t keep any state about a client

Client passes all info needed in each call to server

Advantage:

- no special crash recovery - the server just starts running again
- client might have to retry a request

Pass all info – option-1

Stateless protocol: server maintains no state about clients

Need API change. One possibility:

read(char *path, buf, size, offset)

Specify path and offset each time

Pros? Cons?

Pass all info – option-2

Stateless protocol: server maintains no state about clients

Use file handles

```
fh =   open(char *path);
pread(fh, buf,   size,  offset);
pwrite(fh,   buf,   size,  offset);
```

File Handle = <volume ID, inode #>

Think of volumeID as the file system ID, for simplicity assume only one FS

Pass all info – option-2

What is the problem with using the inode number as the file handle?

What can go wrong?

Pass all info – option-2

Stateless protocol: server maintains no state about clients

Use file handles

```
fh =   open(char *path);
pread(fh, buf,   size,  offset);
pwrite(fh,   buf,   size,  offset);
```

File Handle = <volume ID, inode #, generation#>

Opaque to client, purpose of generation#? when incremented?

Some NFS calls

Lookup – notice no open (open == series of lookups)

GetAttr

Read

Write

Who keeps the fd to fh mapping?

Virtual File System

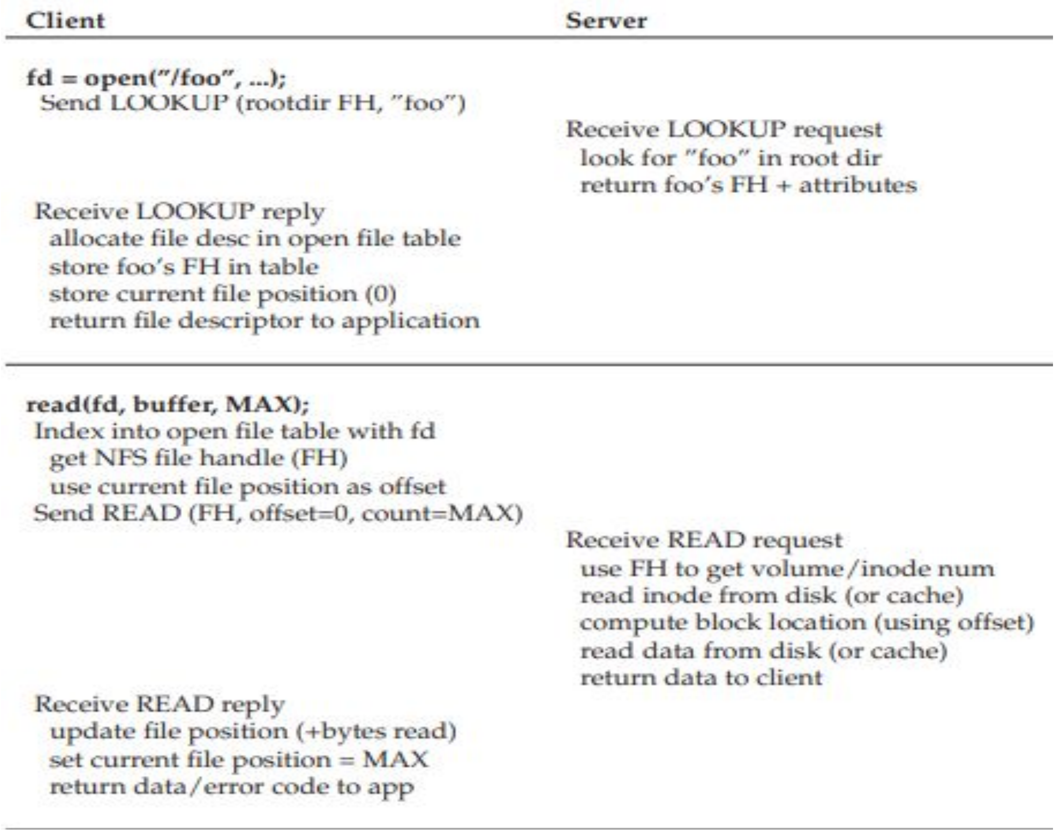
Allows different FSes to be plugged into the OS

Acts a common thin layer above all file systems

Can also hide distributed file systems!

NFS engineers designed VFS to make using NFS easier but today VFS is used for all FS in Linux

Reading a File on NFS



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	
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	

Close() a file?

What happens?

No server communication

Failures

What do clients do when they don’t get a response?

Request lost

Server down

Reply lost

Simplifying Recovery with Idempotency

All cases are handled uniformly

read

write

mkdir

creat

Client-side caching

Cache data for performance

For both reads and writes

What are the problems?

P1: update visibility

Scenario: edit a file and move on to a different workstation

Solution: flush-on-close

Drawbacks?

P2: stale cache

Cached content could be old

Solution: getattr

What problems will this introduce?

Write buffering on server

Can server buffer writes?

BB-Ram to make writes fast with failure-resilience

NFS - a huge success! most possibly due to “open market” approach

NetApp, EMC, IBM all sell NFS servers

AFS

Main goal: scale

Better cache consistency semantics

Successful system from CMU

- many novel ideas
- many institutions use AFS

AFSv1

Whole-file caching instead of block caching

Keep file on disk (not memory like NFS)

Open a file "/path/to/file"– fetch it from the server

Writes and reads – no interaction with server… purely local

Close a file – send it to the server

Next open – use TestAuth message to check if file is up-to-date, if yes
use cached copy else fetch again

AFSv1 Problems

Path traversal costs – too much CPU to convert paths to inodes

Too many TestAuth message – most unnecessary…why?

Load imbalance across servers

Server process/threads structure

AFSv2 - Callbacks

How to reduce TestAuth messages

Callback – promise from server to client that it will inform when the file changes

Implications – ?

AFSv2 - FIDs

How to reduce CPU cost for path resolution

Use an FID

FID = <volume_ID, inode, uniqifier> - similar to NFS

AFSv2

fd = open(/foo/bar.txt, …)	
fetch(/ FID, “foo”)	Look for “foo” in /
	create callback for foo for client
	return foo’s content and FID
write foo to disk	
record callback of foo	
fetch(foo FID, “bar.txt”)	Look for “bar.txt” in foo
	create callback for bar.txt for client
	return bar’s content and FID
write bar.txt to disk	
record callback of bar.txt	
local open cached bar.txt	
return fd to app	

AFSv2

read(fd…) – local

write(fd…) – local

close(fd) – flush to server using Store message – send whole file

Next open…

```
fd = open("/foo/bar.txt", ...);
if (callback(foo) == VALID)
        use local copy foo
else Fetch
if (callback(bar.txt) == VALID)
        open local cached copy
        return file descriptor
else Fetch then open and return fd
```

Cache consistency

Update visibility: when?

Processes on different machines

- upon close, flush to server
- visible to other clients now

Processes on the same machine

- immediately visible

Last closer wins – one consistent version uploaded – compare to NFS?

Cache consistency

Invalidate stale cache: when?

after server gets latest, breaks callback

Is it possible for a client to use an older copy after the file has been changed at the server?

Is it possible for a client to open an older copy?

Crash Recovery

Client crashes – how to handle?

Treat cache as suspect – should throw cached files?

Server crashes

Keeps callback state in memory – can lose, how to handle?

use heartbeats … if not heard from server, suspect cache

Performance Comparison

N = blocks, Lnet, Lmem, Ldisk – latency of net, mem, and disk

	NFS	AFS
Small file, sequential read		
Small file, sequential re-read		
Large file, sequential read		
Large file, sequential re-read		
Large file, single read		
Small file, sequential write (new file)		
Large file, sequential write (new file)		
Large file, sequential overwrite		
Large file, single overwrite		