NFS

CS 537: Introduction to Operating Systems

Louis Oliphant & Tej Chajed

University of Wisconsin - Madison

Spring 2024

Administrivia

- Project 7 due April 30th @ 11:59pm
- Final Exam:
  - **Cumulative, focusing on new material**
  - Lec 1 - May 8th, 12:25-2:25 (Biochem 1125)
  - Lec 2 - May 6th, 2:45-4:45 (Sterling Hall 1310)
  - McBurney: May 6th, 2:40-6:50 (Nancy Nicholas Hall 1135)
  - If you can't take the exam for a *legitimate reason* at your designated time, please fill out the alternate exam form to take the exam with the other lecture. Legitimate Reasons include:
    - Another exam at the same time, Religious conflict, University Sanctioned conflict, Scheduled Medical conflict, Civic Duty (e.g. jury duty), Military Service, Family Caregiving Responsibility, Family Emergency, Serious Illness, 3 or more exams scheduled during a 24 hour period

Review: distributed systems

- Transparent: UDP (unreliable), TCP (reliable stream)
- RPC abstraction and library
  - has a stub generator and run-time library
  - handles issues like fragmentation and byte ordering
  - Typically synchronous calls (wait for completion)

Quiz 23 Distributed Systems

https://tinyurl.com/cs537-sp24-q23

Distributed file systems

Local FS: processes on one machine access shared files

Network FS: processes on multiple machines access shared files

Goals for distributed file systems

Transparent access: don't change applications

Crash recovery: both clients and file server may crash

Reasonable performance?

Network File System (NFS)

Protocol for sharing files

Many independent implementations: Oracle, NetApp, Windows, Linux

Note: this lecture is NFSv3, NFSv4 has many changes

NFS architecture

File server + many independent clients

Communicate via RPCs

Note: client goes through an NFS file system in the kernel

NFS overview

- Architecture
- Stateless network API
- Caching and cache coherency

Protocol design: failed attempt 1

First thought for protocol: same as UNIX system calls

```
int fd = open("foo", O_RDONLY);
read(fd, buf, MAX);
... // server crash here!
read(ff, buf, MAX);
```

Attempt 2: put all info in requests

```
pread(char *path, buf, size, offset)
pwrite(char *path, buf, size, offset)
```

Stateless: server maintain no state about clients

- Pros: server can crash and reboot, no state lost
- Cons: slow, server must look up path on each request

NFSv3: file handles

```
open(char *path) -> file_handle
pread(fh, buf, size, offset)
pwrite(fh, buf, size, offset)
```

file handle = <volume ID, inode number, generation number>

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	

What about append?

```
open(char *path) -> file_handle
pread(fh, buf, size, offset)
pwrite(fh, buf, size, offset)

append(fh, buf, size) // what if we add this?
```

Idempotent operations

Design API so no harm to executing an RPC more than once

If `f()` is idempotent, then `f()` as same effect as `f(); f()`

If `f()` is idempotent and server doesn't respond, client can safely retry

What operations are idempotent?

Idempotent: read, pwrite

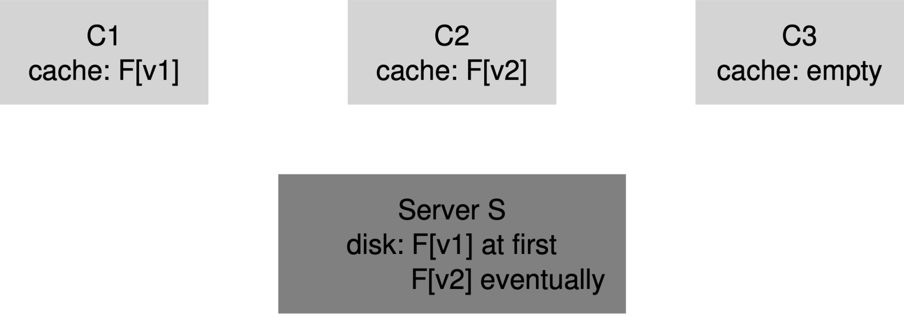
Not idempotent: append

Sort of: mkdir, create

Cache consistency

NFS clients cache data to avoid server requests

Multiple clients means **cache consistency** issues



Oddities caused by caching

1. **Update visibility**: when one client updates a file, when are they propagated to the server/other clients?
2. **Stale cache**: when a client has a cached file, when is it invalidated due to changes by other clients?

Provide "close-to-open" consistency ("flush-on-close"): flush on `close`, so `open` sees any writes before `close`

NFSv2 solution: GETATTR

Clients track the modified time of files

Cache modified time and update periodically (e.g., every 3 seconds)

(NFSv3 has a slightly better solution called “weak cache consistency”)

NFS summary

- Architecture: many clients accessing one server
- Stateless protocol design
- Caching and cache consistency issues