

Filesystems 2

Administrative

- P1.1 due today (but you know this already)
- 14 students attended IRC office hours last night - do people like this format? Suggestions for alternatives? Need pointers to IRC clients?

Filesystems

- Last time: Looked at how we could use RPC to split filesystem functionality between client and server
- But pretty much, we didn't change the design
- We just moved the entire filesystem to the server
 - and then added some caching on the client in various ways

Reprise of last time

- Client - VFS layer ----RPC---->Server
- + Local cache
- + Leases and/or timeouts for consistency

- How do you scale something like NFS or AFS? A: Partitioning
- What handles do we have to partition on? The namespace.
/afs/andrew.cmu.edu/usr23/dga2 [...usr1, usr2, ..., usrN]

Partitioning / Sharding

- Very common technique in scaling distributed systems
- Key question: What do you partition based upon? Drawbacks and advantages?
 - “Username” (AFS): Makes sense, good for access locality.
 - But what if one user has a 200TB quota, or really hammers the filesystem? Load balance may not be as great.
 - File or chunk: Very fine-grained load balance; - may require accessing LOTS of servers. (Panasas NFS striping)

You can go farther...

- But it requires ripping apart the filesystem functionality into modules
- and placing those modules at different computers on the network
- So now we need to ask... what does a filesystem do, anyway?

- Well, there’s a disk...
 - disks store bits. in fixed-length pieces called sectors or blocks
- but a filesystem has ... files. and often directories. and maybe permissions. creation and modification time. and other stuff *about* the files. (“metadata”)

Filesystem functionality

- Directory management (maps entries in a hierarchy of names to files-on-disk)
- File management (manages adding, reading, changing, appending, deleting) individual files
- Space management: *where* on disk to store these things?
- Metadata management

Conventional filesystem

- Wraps all of these up together
- Useful concepts: [pictures]
 - “Superblock” -- well-known location on disk where top-level filesystem info is stored (pointers to more structures, etc.)
 - “Free list” or “Free space bitmap” -- data structures to remember what’s used on disk and what’s not. Why? Fast allocation of space for new files.
 - “inode” - short for index node - stores all metadata about a file, plus information pointing to where the file is stored on disk
 - Small files may be referenced entirely from the inode; larger files may have some indirection to blocks that list locations on disk
 - Directory entries point to inodes
 - “extent” - a way of remembering where on disk a file is stored. Instead of listing all blocks, list a starting block and a range. More compact representation, but requires large contiguous block allocation.

Filesystem “VFS” ops

- VFS: (‘virtual filesystem’): common abstraction layer inside kernels for building filesystems -- interface is common across FS implementations
 - Think of this as an abstract data type for filesystems
 - has both syntax (function names, return values, etc) and semantics (“don’t block on this call”, etc.)
- One key thing to note: The VFS itself may do some caching and other management...
 - in particular: often maintains an inode cache

Directory operations

- `readdir(path)` - return directory entries for each file in the directory
- `mkdir(path)` -- create a new directory
- `rmdir(path)` -- remove the named directory

File operations

- `mknod(path, mode, dev)` -- create a new “node” (generic: a file is one type of node; a device node is another)
- `unlink(path)` -- remove link to inode, decrementing inode’s reference count
 - many filesystems permit “hard links” -- multiple directory entries pointing to the same file
- `rename(path, newpath)`
- `open` -- open a file, returning a file handle
- `, read, write`
- `truncate` -- cut off at particular length
- `flush` -- close one handle to an open file
- `release` -- completely close file handle

Metadata ops

- `getattr(path)` -- return metadata struct
- `chmod / chown` (ownership & perms)

Back to goals of DFS

- Users should have same view of system, be able to share files
- Last time:
 - Central fileserver handles *all* filesystem operations -- consistency was easy, but overhead high, scalability poor
 - Moved to NFS and then AFS: Added more and more caching at client; added cache consistency problems
 - Solved using timeouts or callbacks to expire cached contents

Protocol & consistency

- Remember last time: NFS defined operations to occur on unique inode #s instead of names... why? *idempotency*. Wanted operations to be unique.
 - Related example for today when we're considering splitting up components: moving a file from one directory to another
 - What if this is a complex operation ("remove from one", "add to another"), etc.
 - Can another user see intermediate state?? (e.g., file in both directories or file in neither?)
- Last time: Saw issue of *when* things become consistent
 - Presented idea of close-to-open consistency as a compromise

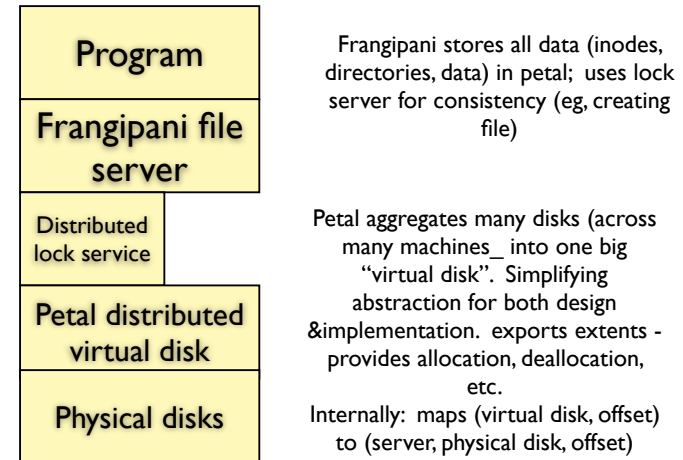
Scaling beyond...

- What happens if you want to build AFS for all of CMU? More disks than one machine can handle; more users than one machine can handle
- Simplest idea: Partition users onto different servers
 - How do we handle a move across servers?
 - How to divide the users? Statically? What about load balancing for operations & for space? Some files become drastically more popular?

“Cluster” filesystems

- Lab inspired by Frangipani, a scalable distributed filesystem.
- Think back to our list of things that filesystems have to do
 - Concurrency management
 - Space allocation and data storage
 - Directory management and naming

Frangipani design



Consequential design

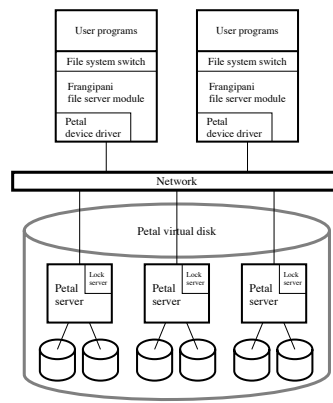


Figure 2: Frangipani structure. In one typical Frangipani configuration, some machines run user programs and the Frangipani file server module; others run Petal and the distributed lock service. In other configurations, the same machines may play both roles.

Compare with NFS/ AFS

- In NFS/AFS, clients just relay all FS calls to the server; central server.
- Here, clients run enough code to know *which* server to direct things to; are active participants in filesystem.
- (n.b. -- you could, of course, use the Frangipani/Petal design to build a scalable NFS server -- and, in fact, similar techniques are how a lot of them actually are built. See upcoming lecture on RAID, though: replication and redundancy management become key)