CSE 486/586 Distributed Systems
Distributed File Systems

Steve Ko
Computer Sciences and Engineering
University at Buffalo

Recap

- Optimistic quorum
- Distributed transactions with replication
  - One copy serializability
  - Primary copy replication
  - Read-one/write-all replication
  - Active copies replication

Local File Systems

- File systems provides file management.
  - Name space
  - API for file operations (create, delete, open, close, read, write, append, truncate, etc.)
  - Physical storage management & allocation (e.g., block storage)
  - Security and protection (access control)
- Name space is usually hierarchical.
  - Files and directories
- File systems are mounted.
  - Different file systems can be in the same name space.

Traditional Distributed File Systems

- Goal: emulate local file system behaviors
  - Files not replicated
  - No hard performance guarantee
- But,
  - Files located remotely on servers
  - Multiple clients access the servers
- Why?
  - Users with multiple machines
  - Data sharing for multiple users
  - Consolidated data management (e.g., in an enterprise)

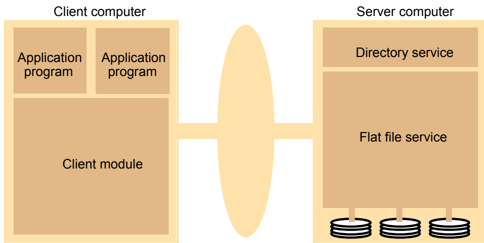
Requirements

- Transparency: a distributed file system should appear as if it's a local file system
  - Access transparency: it should support the same set of operations, i.e., a program that works for a local file system should work for a DFS.
  - (File) Location transparency: all clients should see the same name space.
  - Migration transparency: if files move to another server, it shouldn't be visible to users.
  - Performance transparency: it should provide reasonably consistent performance.
  - Scaling transparency: it should be able to scale incrementally by adding more servers.

Requirements

- Concurrent updates should be supported.
- Fault tolerance: servers may crash, msgs can be lost, etc.
- Consistency needs to be maintained.
- Security: access-control for files & authentication of users

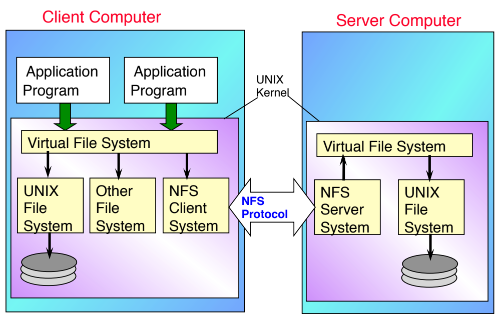
File Server Architecture



Components

- Directory service
  - Meta data management
  - Creates and updates directories (hierarchical file structures)
  - Provides mappings between user names of files and the unique file ids in the flat file structure.
- Flat file service
  - Actual data management
  - File operations (create, delete, read, write, access control, etc.)
- These can be independently distributed.
  - E.g., centralized directory service & distributed flat file service

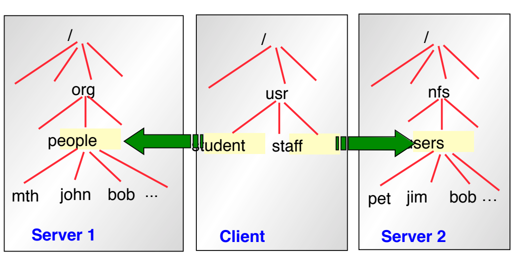
Sun NFS



VFS

- A translation layer that makes file systems pluggable & co-exist
  - E.g., NFS, EXT2, EXT3, ZFS, etc.
- Keeps track of file systems that are available locally and remotely.
- Passes requests to appropriate local or remote file systems
- Distinguishes between local and remote files.

NFS Mount Service



Remote Mount

Each server keeps a record of local files available for remote mounting. Clients use a *mount* command for remote mounting, providing name mappings

NFS Basic Operations

- Client
  - Transfers blocks of files to and from server via RPC
- Server
  - Provides a conventional RPC interface at a well-known port on each host
  - Stores files and directories
- Problems?
  - Performance
  - Failures

Improving Performance

- Let's cache!
- Server-side
  - Typically done by OS & disks anyway
  - A disk usually has a cache built-in.
  - OS caches file pages, directories, and file attributes that have been read from the disk in a *main memory buffer cache.*
- Client-side
  - On accessing data, cache it locally.
- What's a typical problem with caching?
  - Consistency: cached data can become stale.

(General) Caching Strategies

- Read-ahead (prefetch)
  - Read strategy
  - Anticipates read accesses and fetches the pages following those that have most recently been read.
- Delayed-write
  - Write strategy
  - New writes stored locally.
  - Periodically or when another client accesses, send back the updates to the server
- Write-through
  - Write strategy
  - Writes go all the way to the server's disk
- This is not an exhaustive list!

NFS Client-Side Caching

- Write-through, but only at close()
  - Not every single write
  - Helps performance
- Other clients periodically check if there's any new write (next slide).
- Multiple writers
  - No guarantee
  - Could be any combination of writes
- Leads to inconsistency

Validation

- A client checks with the server about cached blocks.
- Each block has a timestamp.
  - If the remote block is new, then the client invalidates the local cached block.
- Always invalidate after some period of time
  - 3 seconds for files
  - 30 seconds for directories
- Written blocks are marked as "dirty."

Failures

- Two design choices: stateful & stateless
- Stateful
  - The server maintains all client information (which file, which block of the file, the offset within the block, file lock, etc.)
  - Good for the client-side process (just send requests!)
  - Becomes almost like a local file system (e.g., locking is easy to implement)
- Problem?
  - Server crash → lose the client state
  - Becomes complicated to deal with failures

Failures

- Stateless
  - Clients maintain their own information (which file, which block of the file, the offset within the block, etc.)
  - The server does not know anything about what a client does.
  - Each request contains complete information (file name, offset, etc.)
  - Easier to deal with server crashes (nothing to lose!)
- NFS's choice
- Problem?
  - Locking becomes difficult.

NFS

- Client-side caching for improved performance
- Write-through at close()
  - Consistency issue
- Stateless server
  - Easier to deal with failures
  - Locking is not supported (later versions of NFS support locking though)
- Simple design
  - Led to simple implementation, acceptable performance, easier maintenance, etc.
  - Ultimately led to its popularity

CSE 486/586 Administrivia

- Midterm scoring mostly done.
  - 26 rejected, will need some manual verification.
  - Will post the result by Friday.
- PA3
  - Will post the result by Friday.
- PA4
  - Will be released tonight.
  - Tester might not be ready though ☹

New Trends in Distributed Storage

- Geo-replication: replication with multiple data centers
  - Latency: serving nearby clients
  - Fault-tolerance: disaster recovery
- Power efficiency: power-efficient storage
  - Going green!
  - Data centers consume lots of power

Power Consumption

- eBay: 16K servers, ~0.6 * 10^5 MWh, ~$3.7M
- Akamai: 40K servers, ~1.7 * 10^5 MWh, ~$10M
- Rackspace: 50K servers, ~2 * 10^5 MWh, ~$12M
- Microsoft: > 200K servers, > 6 * 10^5 MWh, > $36M
- Google: > 500K servers, > 6.3 * 10^5 MWh, > $38M
- USA (2006): 10.9M servers, 610 * 10^5 MWh, $4.5B
- Year-to-year: 1.7%~2.2% of total electricity use in US
- http://ccr.sigcomm.org/online/files/p123.pdf
- Question: can we reduce the energy footprint of a distributed storage while preserving performance?

One Extreme Design Point: FAWN

- Fast Array of Wimpy Nodes
  - Andersen et al. (CMU & Intel Labs)
- Coupling of low-power, efficient embedded CPUs with flash storage
  - Embedded CPUs are more power efficient.
  - Flash is faster than disks, cheaper than memory, consumes less power than either.
- Performance target
  - Not just queries (requests) per second
  - Queries per second per Watt (queries per Joule)

Embedded CPUs

- Observation: many modern server storage workloads do not need fast CPUs
  - Not much computation necessary, mostly just small I/O
  - I.e., mostly I/O bound, not CPU bound
  - E.g., 1 KB values for thumbnail images, 100s of bytes for wall posts, twitter messages, etc.
- (Rough) Comparison
  - Server-class CPUs (superscalar quad-core): 100M instructions/Joule
  - Embedded CPUs (low-frequency, single-core): 1B instructions/Joule

Flash (Solid State Disk)

- Unlike magnetic disks, there's no mechanical part
  - Disks have motors that rotate disks & arms that move and read.
- Efficient I/O
  - Less than 1 Watt consumption
  - Magnetic disks over 10 Watt
- Fast random reads
  - << 1 ms
  - Up to 175 times faster than random reads on magnetic disks

Flash (Solid State Disk)

- The smallest unit of operation (read/write) is a page
  - Typically 4KB
  - Initially all 1
  - A write involves setting some bits to 0
  - A write is fundamentally constrained.
- Individual bits cannot be reset to 1.
  - Requires an erasure operation that resets all bits to 1.
  - This erasure is done over a large block (e.g., 128KB), i.e., over multiple pages together.
  - Typical latency: 1.5 ms
- Blocks wear out for each erasure.
  - 100K cycles or 10K cycles depending on the technology.

Flash (Solid State Disk)

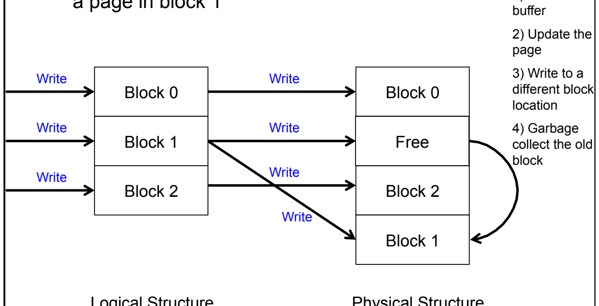
- Early design limitations
  - Slow write: a write to a random 4 KB page → the entire 128 KB erase block to be erased and rewritten → write performance suffers
  - Uneven wear: imbalanced writes result in uneven wear across the device
- Any idea to solve this?

Flash (Solid State Disk)

- Recent designs: log-based
- The disk exposes a logical structure of pages & blocks (called *Flash Translation Layer*).
  - Internally maintains remapping of blocks.
- For rewrite of a random 4KB page:
  - Read the surrounding entire 128KB erasure block into the disk's internal buffer
  - Update the 4KB page in the disk's internal buffer
  - Write the entire block to a new or previously erased physical block
  - Additionally, carefully choose this new physical block to minimize uneven wear

Flash (Solid State Disk)

- E.g. sequential write till block 2, then random read of a page in block 1



FAWN Design

- Wimpy nodes based on PCEngine Alix 3c2
  - Commonly used for thin clients, network firewalls, wireless routers, etc.
- Single-core 500 MHz AMD Geode LX
- 256MB RAM at 400 MHz
- 100 MBps Ethernet
- 4 GB Sandisk CompactFlash
- Power consumption
  - 3W when idle
  - 6W under heady load

Summary

- NSF
  - Caching with write-through policy at close()
  - Stateless server
- One power efficient design: FAWN
  - Embedded CPUs & Flash storage

Acknowledgements

- These slides contain material developed and copyrighted by Indranil Gupta (UIUC).