15-440 Distributed Systems

Lecture 8 – Distributed File Systems 2

1

Review of Last Lecture

- Distributed file systems functionality
- Implementation mechanisms example
  - Client side: VFS interception in kernel
  - Communications: RPC
  - Server side: service daemons
- Design choices
  - Topic 1: client-side caching
    - NFS and AFS

2

Today's Lecture

- DFS design comparisons continued
  - Topic 2: file access consistency
    - NFS, AFS
  - Topic 3: name space construction
    - Mount (NFS) vs. global name space (AFS)
  - Topic 4: Security in distributed file systems
    - Kerberos
- Other types of DFS
  - Coda – disconnected operation
  - LBFS – weakly connected operation

3

Topic 2: File Access Consistency

- In UNIX local file system, concurrent file reads and writes have “sequential” consistency semantics
  - Each file read/write from user-level app is an atomic operation
    - The kernel locks the file vnode
  - Each file write is immediately visible to all file readers
- Neither NFS nor AFS provides such concurrency control
  - NFS: “sometime within 30 seconds”
  - AFS: session semantics for consistency

4

Session Semantics in AFS v2

- What it means:
  - A file write is visible to processes on the same box immediately, but not visible to processes on other machines until the file is closed
  - When a file is closed, changes are visible to new opens, but are not visible to “old” opens
  - All other file operations are visible everywhere immediately
- Implementation
  - Dirty data are buffered at the client machine until file close, then flushed back to server, which leads the server to send “break callback” to other clients

5

AFS Write Policy

- Writeback cache
  - Opposite of NFS “every write is sacred”
  - Store chunk back to server
    - When cache overflows
    - On last user close()
  - ...or don't (if client machine crashes)
- Is writeback crazy?
  - Write conflicts “assumed rare”
  - Who wants to see a half-written file?

6

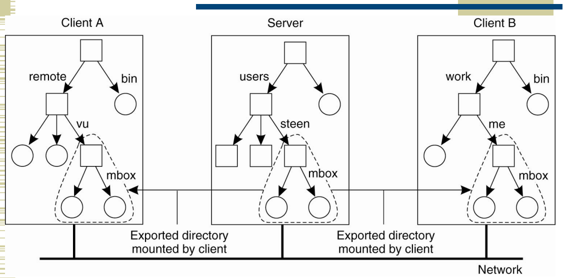
Results for AFS

- Lower server load than NFS
  - More files cached on clients
  - Callbacks: server not busy if files are read-only (common case)
- But maybe slower: Access from local disk is much slower than from another machine's memory over LAN
- For both:
  - Central server is bottleneck: all reads and writes hit it at least once;
  - is a single point of failure.
  - is costly to make them fast, beefy, and reliable servers.

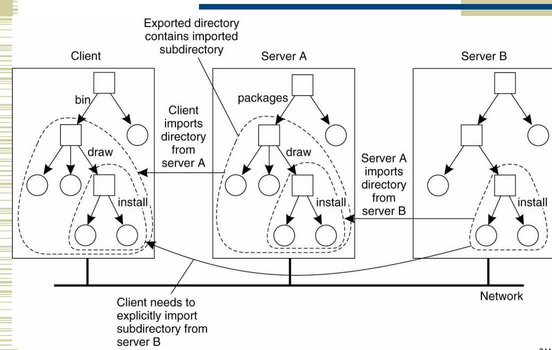
Topic 3: Name-Space Construction and Organization

- NFS: per-client linkage
  - Server: export /root/fs1/
  - Client: mount server:/root/fs1 /fs1
- AFS: global name space
  - Name space is organized into Volumes
    - Global directory /afs;
    - /afs/cs.wisc.edu/vol1/…; /afs/cs.stanford.edu/vol1/…
  - Each file is identified as fid = <vol_id, vnode #, unique identifier>
  - All AFS servers can access "volume location database", which is a table of vol_id→ server_ip mappings

Naming in NFS (1)



Naming in NFS (2)



Implications on Location Transparency

- NFS: no transparency
  - If a directory is moved from one server to another, client must remount
- AFS: transparency
  - If a volume is moved from one server to another, only the volume location database on the servers needs to be updated

Topic 4: User Authentication and Access Control

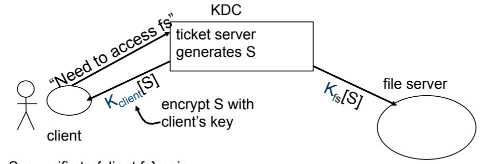
- User X logs onto workstation A, wants to access files on server B
  - How does A tell B who X is?
  - Should B believe A?
- Choices made in NFS V2
  - All servers and all client workstations share the same <uid, gid> name space → B send X's <uid,gid> to A
    - Problem: root access on any client workstation can lead to creation of users of arbitrary <uid, gid>
  - Server believes client workstation unconditionally
    - Problem: if any client workstation is broken into, the protection of data on the server is lost;
    - <uid, gid> sent in clear-text over wire → request packets can be faked easily

User Authentication (cont'd)

- How do we fix the problems in NFS v2
  - Hack 1: root remapping → strange behavior
  - Hack 2: UID remapping → no user mobility
  - Real Solution: use a centralized Authentication/ Authorization/Access-control (AAA) system

A Better AAA System: Kerberos

- Basic idea: shared secrets
  - User proves to KDC who he is; KDC generates shared secret between client and file server



S: specific to {client,fs} pair;
"short-term session-key"; expiration time (e.g. 8 hours)

Key Lessons

- Distributed filesystems almost always involve a tradeoff: consistency, performance, scalability.
- We'll see a related tradeoff, also involving consistency, in a while: the CAP tradeoff. Consistency, Availability, Partition-resilience.

More Key Lessons

- Client-side caching is a fundamental technique to improve scalability and performance
  - But raises important questions of cache consistency
- Timeouts and callbacks are common methods for providing (some forms of) consistency.
- AFS picked close-to-open consistency as a good balance of usability (the model seems intuitive to users), performance, etc.
  - AFS authors argued that apps with highly concurrent, shared access, like databases, needed a different model

Today's Lecture

- DFS design comparisons continued
  - Topic 2: file access consistency
    - NFS, AFS
  - Topic 3: name space construction
    - Mount (NFS) vs. global name space (AFS)
  - Topic 4: AAA in distributed file systems
    - Kerberos
- Other types of DFS
  - Coda – disconnected operation
  - LBFS – weakly connected operation

Background

- We are back to 1990s.
- Network is slow and not stable
- Terminal → "powerful" client
  - 33MHz CPU, 16MB RAM, 100MB hard drive
- Mobile Users appeared
  - 1st IBM Thinkpad in 1992
- We can do work at client without network

CODA

- Successor of the very successful Andrew File System (AFS)
- AFS
  - First DFS aimed at a campus-sized user community
  - Key ideas include
    - open-to-close consistency
    - callbacks

Hardware Model

- CODA and AFS assume that client workstations are personal computers controlled by their user/ owner
  - ***Fully autonomous***
  - ***Cannot be trusted***
- CODA allows owners of laptops to operate them in ***disconnected mode***
  - ***Opposite of ubiquitous connectivity***

Accessibility

- Must handle two types of failures
  - ***Server failures:***
    - Data servers are ***replicated***
  - ***Communication failures*** and ***voluntary disconnections***
    - Coda uses ***optimistic replication*** and ***file hoarding***

Design Rationale

- Scalability
  - Callback cache coherence (inherit from AFS)
  - Whole file caching
  - Fat clients. (security, integrity)
  - Avoid system-wide rapid change
- Portable workstations
  - User’s assistance in cache management

Design Rationale –Replica Control

- Pessimistic
  - Disable all partitioned writes

  \- Require a client to acquire control of a cached object ***prior*** to disconnection
- Optimistic
  - Assuming no others touching the file

  \- conflict detection

  \+ fact: low write-sharing in Unix

  \+ high availability: access anything in range

What about Consistency?

- ***Pessimistic replication control protocols*** guarantee the consistency of replicated in the presence of ***any non-Byzantine failures***
  - Typically require a quorum of replicas to allow access to the replicated data
  - Would **not** support disconnected mode

Pessimistic Replica Control

- Would require client to acquire ***exclusive*** (RW) or ***shared*** (R) control of cached objects before accessing them in disconnected mode:
  - Acceptable solution for voluntary disconnections
  - Does not work for involuntary disconnections
- What if the laptop remains disconnected for a long time?

Leases

- We could grant exclusive/shared control of the cached objects for a ***limited amount of time***
- Works very well in ***connected mode***
  - Reduces server workload
  - Server can keep leases in volatile storage as long as their duration is shorter than boot time
- Would only work for very short disconnection periods

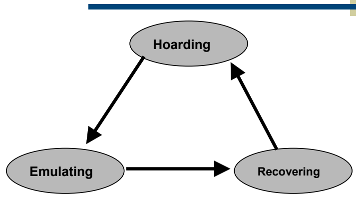
Optimistic Replica Control (I)

- ***Optimistic replica control*** allows access in ***every*** disconnected mode
  - Tolerates temporary inconsistencies
  - Promises to detect them later
  - Provides ***much higher data availability***

Optimistic Replica Control (II)

- Defines an ***accessible universe:*** set of filesthat the user can access
  - Accessible universe varies over time
- At any time, user
  - Will read from the latest file(s) in his accessible universe
  - Will update all files in his accessible universe

Coda States



1. ***Hoarding:***
   Normal operation mode
2. ***Emulating:***
   Disconnected operation mode
3. ***Reintegrating:***
   Propagates changes and detects inconsistencies

Hoarding

- Hoard useful data for disconnection
- Balance the needs of connected and disconnected operation.
  - Cache size is restricted
  - Unpredictable disconnections
- Uses user specified preferences + usage patterns to decide on files to keep in hoard

Prioritized algorithm

- User defined hoard priority *p*: how important is a file to you?
- Recent Usage *q*
- Object priority = f(p,q)
- Kick out the one with lowest priority

\+ Fully tunable

Everything can be customized

\- Not tunable (?)

- No idea how to customize

31

Hoard Walking

- Equilibrium – uncached obj < cached obj
  - Why it may be broken? Cache size is limited.
- Walking: restore equilibrium
  - Reloading HDB (changed by others)
  - Reevaluate priorities in HDB and cache
  - Enhanced callback
- Increase scalability, and availability
- Decrease consistency

32

Emulation

- In emulation mode:
  - Attempts to access files that are not in the client caches appear as failures to application
  - All changes are written in a persistent log, the client modification log (CML)
  - Coda removes from log all obsolete entries like those pertaining to files that have been deleted

33

Persistence

- Coda keeps its cache and related data structures in non-volatile storage
- All Venus metadata are updated through ***atomic transactions***
  - Using a lightweight ***recoverable virtual memory*** (RVM) developed for Coda
  - Simplifies Venus design

34

Reintegration

- When workstation gets reconnected, Coda initiates a ***reintegration process***
  - Performed one volume at a time
  - Venus ships replay log to all volumes
  - Each volume performs a log replay algorithm
- Only care about write/write confliction
  - Conflict resolution succeeds?
    - Yes. Free logs, keep going…
    - No. Save logs to a tar. Ask for help
- In practice:
  - No Conflict at all! Why?
  - Over 99% modification by the same person
  - Two users modify the same obj within a day: <0.75%

35

Coda Summary

- Puts scalability and availability before data consistency
  - Unlike NFS
- Assumes that inconsistent updates are very infrequent
- Introduced disconnected operation mode and file hoarding

36

Remember this slide?

- We are back to 1990s.
- Network is slow and not stable
- Terminal → "powerful" client
  - 33MHz CPU, 16MB RAM, 100MB hard drive
- Mobile Users appear
  - 1st IBM Thinkpad in 1992

What's now?

- We are in 2000s now.
- Network is fast and reliable in LAN
- "powerful" client → very powerful client
  - 2.4GHz CPU, 4GB RAM, 500GB hard drive
- Mobile users everywhere
- Do we still need support for disconnection?
  - WAN and wireless is not very reliable, and is slow

Today's Lecture

- DFS design comparisons continued
  - Topic 2: file access consistency
    - NFS, AFS
  - Topic 3: name space construction
    - Mount (NFS) vs. global name space (AFS)
  - Topic 4: AAA in distributed file systems
    - Kerberos
- Other types of DFS
  - Coda – disconnected operation
  - LBFS – weakly connected operation

Low Bandwidth File System Key Ideas

- A network file systems for slow or wide-area networks
- Exploits similarities between files or versions of the same file
  - Avoids sending data that can be found in the server's file system or the client's cache
- Also uses conventional compression and caching
- Requires 90% less bandwidth than traditional network file systems

Working on slow networks

- Make local copies
  - Must worry about update conflicts
- Use remote login
  - Only for text-based applications
- Use instead a LBFS
  - Better than remote login
  - Must deal with issues like auto-saves blocking the editor for the duration of transfer

LBFS design

- LBFS server divides file it stores into chunks and indexes the chunks by hash value
- Client similarly indexes its file cache
- Exploits similarities between files
  - LBFS never transfers chunks that the recipient already has

Indexing

- Uses the SHA-1 algorithm for hashing
  - It is collision resistant
- Central challenge in indexing file chunks is keeping the index at a reasonable size while dealing with shifting offsets
  - Indexing the hashes of fixed size data blocks
  - Indexing the hashes of all overlapping blocks at all offsets

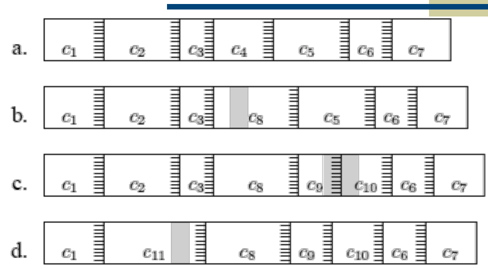
43

LBFS chunking solution

- Considers only non-overlapping chunks
- Sets chunk boundaries based on file contents rather than on position within a file
- Examines every overlapping 48-byte region of file to select the boundary regions called *breakpoints* using Rabin fingerprints
  - When low-order 13 bits of region's fingerprint equals a chosen value, the region constitutes a breakpoint

44

Effects of edits on file chunks



- Chunks of file before/after edits
  - Grey shading show edits
- Stripes show regions with magic values that creating chunk boundaries

45

More Indexing Issues

- Pathological cases
  - Very small chunks
    - Sending hashes of chunks would consume as much bandwidth as just sending the file
  - Very large chunks
    - Cannot be sent in a single RPC
- LBFS imposes minimum and maximum chuck sizes

46

The Chunk Database

- Indexes each chunk by the first 64 bits of its SHA-1 hash
- To avoid synchronization problems, LBFS always recomputes the SHA-1 hash of any data chunk before using it
  - Simplifies crash recovery
- Recomputed SHA-1 values are also used to detect hash collisions in the database

47

Conclusion

- Under normal circumstances, LBFS consumes 90% less bandwidth than traditional file systems.
- Makes transparent remote file access a viable and less frustrating alternative to running interactive programs on remote machines.

48