Distributed File Systems

Case Studies:

Sprite

Coda

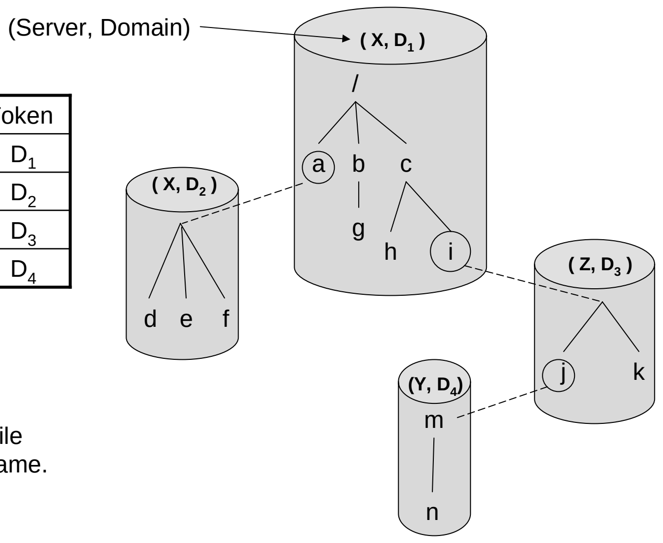
Sprite (SFS)

- Provides identical file hierarchy to all users
- Location transparency
- Pathname lookup using a prefix table
  - Lookup simpler and more efficient
  - Allows dynamic reconfiguration
- Caching
  - Client-caching in main memory
  - Delayed write policy

Name Lookup in Sprite

Prefix Table

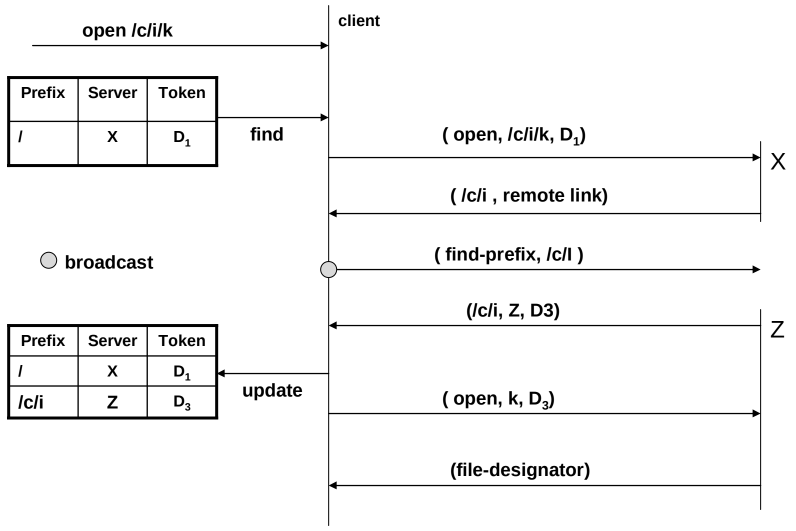
Prefix	Server	Token
/	X	D_1
/a	X	D_2
/c/i	Z	D_3
/c/i/j	Y	D_4



● Remote link

A specially marked file containing its own name.

Constructing the Prefix Tables



Prefix	Server	Token
/	X	D_1

Prefix	Server	Token
/	X	D_1
/c/i	Z	D_3

Prefix Table Advantages

- Efficient name lookup (in comparison to component-at-a-time lookup as in NFS)
- Added fault tolerance (once an entry for a domain is loaded in the prefix table of a client, that client can access files in the domain regardless of failures to other servers)
- Allows dynamic reconfiguration (if a known server stops responding, broadcast the path again to find its new location)
- Permits private domains (a client adds to its prefix table the path to the root of the private subtree and refuses to respond to broadcast requests for that path name)

Caching in Sprite

- Client memory cache of accessed disk blocks
- Empirical observations
  - 20-30% of new data is deleted within 30 s
  - 75% of files are open for less than 0.5 s
  - 90% of all files are open for less than 10 s
- Delayed write policy
  - Check by daemon every 5 s
  - Changed blocks not accessed for 30 s are written back to server (or if ejected from cache by LRU policy)
  - Transfer from server cache to server disk in 30 s to 60 s

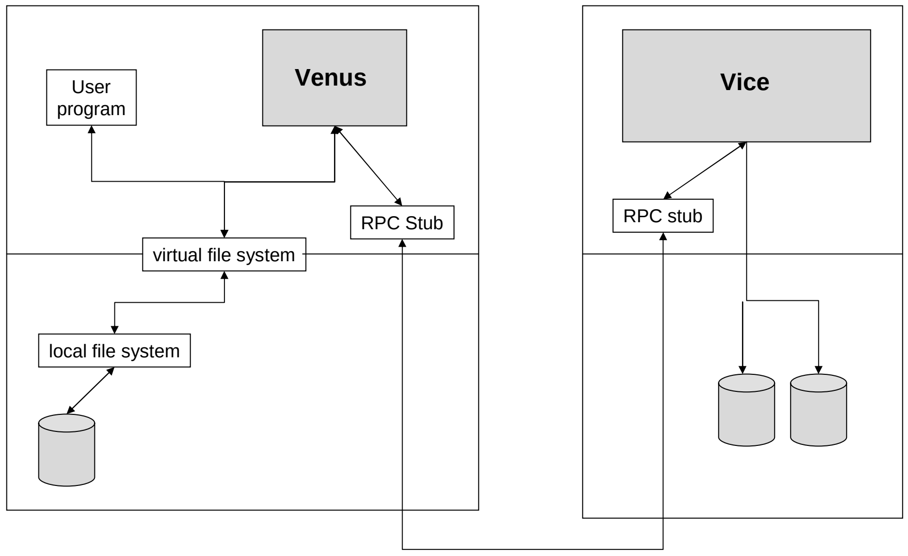
Cache Consistency

- Server-initiated invalidation
- Concurrent write sharing
  - Detected at open of second write
  - Server notifies client with write access to flush all modified blocks to server
  - Server notifies all clients that the file is no longer cachable
- Sequential write sharing
  - Each file has a version number incremented at each open for write access
  - Version number allows client to detect outdated blocks
  - Server maintains identify of last client with write access
  - When file is opened, last writer is asked to flush to the server any modified blocks

CODA

- Derived from Andrew File System (AFS)
- Single location-transparent UNIX file system
- Scalability in CODA
  - Small set of trusted servers used for file storage/management
  - Caching; cache coherence through callbacks
  - Whole-file philosophy
    - Entire file is transfered to client on open
    - Entire file is cached in client
    - Infrequent updating of shared files
    - Working set of typical user fits into cache
- Additional CODA goals
  - Support for disconnected operations
  - Greater reliability/availability vs. AFS
  - Relaxed emulation of UNIX semantics

CODA Architecture



Opening a File

- User process issues open(FileName, mode) call
- UNIX kernel passes request to Venus.
- Venus check if file is in cache. If not, or no valid callback promise, retrieve file from Vice
- Vice copies file to Venus, with a callback promise. Logs callback promise.
- Venus places copy of file in local cache.
- Unix kernel opens file and returns file descriptor to application.

Volumes and Replication

- Volume
  - Directory sub-tree
  - Unit of replication
  - Volume storage group (VSG) – set of servers hosting a given volume
  - Accessible VSG (AVSG) – currently accessible subset of VSG
  - Expansion/contraction of AVSG detected by periodic probes
  - The AVSG for each cached file is recorded by client
- File identifier
  - Unique internal identifier for each file/directory
  - FID = (volume#, vnode#, uniquifier)
  - Does not contain location information
  - Replicas of a file have the same file identifier
  - Directory entry: <name, FID>
- Volume location database
  - Replicated on each server
  - Used to locate volumes/files

Replication and Caching

- Actions on a cache-miss
  - Retrieve data from a preferred server (PS) in AVSG
  - Collect status/version information from all servers in AVSG
  - If replicas are in conflict – abort
  - If some replicas are stale – notify AVSG asynchronously
  - If PS is stale – select new PS
- When file is returned
  - Cache file on client
  - Cache location information
  - Establish callback on server
- On close after modification
  - Transfer file to all members of AVSG

Replica Management

- A storeid = <client-id, timestamp> is associated with each file modification that the client performs on a server
- Each server conceptually maintains an update history of storeids
- The most recent storeid is the lastest storeid (LSID)
- Replicas on A and B are:
  - Equal: if $LSID_A = LSID_B$
  - A dominates B: LSID's are different and $LSID_B$ is in A's history
  - A is submissive to B: LSID's are different and $LSID_A$ is in B's history
  - A and B are inconsistent, otherwise

History Approximation

- It is impractical to maintain the entire history
- The history of each replica is represented by the history's length
- Each replica maintains a vector (CVV – coda version vector) recording the length of each replica's history
- Two replicas are compared as follows:
  - Strong equality: $LSID_A = LSID_B$ and $CVV_A = CVV_B$
  - Weak equality: $LSID_A = LSID_B$ and CVV_A != CVV_B
  - Dominance/submission: $LSID_A$!= $LSID_B$ and CVV_A >= CVV_B
  - Inconsistent: otherwise