Distributed File System

Sharing

- Location-transparent sharing of data among workstations.
- A distributed file system subsumes fast and reliable underlying network connection, so that remote files appear local.
- Sharing of files is achieved by distinguishing two components:
  - **Client**
  - **Server**

Distributed File System

Motivation

Goal

- To support efficient and transparent access to shared files maintaining data consistency.

Problems

- Large number of mobile computers.
- Limited resources of mobile computers.
- Low and variable network bandwidth.
- Intermittent connection.
- Heterogeneity of h/w and s/w.
- Conventional RPC (ok for NFS) will not work.

Solutions

- Replication of data (cloning, copying and caching)
- Data collection in anticipation of use (hoarding, prefetching)

Distributed File System

Consistency

Problem of consistency

- In a loosely coupled distributed system all views on data will not be same.
- How view change should be propagated and to which users?
- Strong consistency (atomic update) out of question.

Weak consistency

- Conflict resolution applied gradually afterwards to bring back consistency.
- Invalidation of data in mobile cache is problematic (disconnected mobiles).

Distributed File System

Update conflicts

Conflict detection techniques

- Content independent: version numbering, timestamping
- Content dependent: dependency graph.

Symmetry

- P2P relationship between client and server.
- Support in fixed n/w and/or mobile devices.
- One file or several file systems
- One or several namespaces for the files

Distributed File System

Connectedness

Transparency

- Hides mobility support: applications do not require re-engineering.
- Users are oblivious of mobility.

Consistency model

- Optimistic or pessimistic

Caching and pre-fetching

- Single files, directories, subtree, partitions
- Permanent or only at certain points in time.

Distributed File System

Connectedness

Data management

- Management of buffered data and copies.
- Requests for updates, validity of data.
- Detection of changes in data

Conflict solution

- Application specific or general
- Errors.

Distributed File System

Storage systems for mobiles

- Storage systems: Coda, Bayou, and Rover
  - Bayou and Rover both support client mobility and weak network connectivity.
- Both employ replications, caching, and weak data consistency.
- **Bayou's focus**: application specific mechanism to detect conflict and resolve them.
- **Rover's focus**: both mobility-transparent and mobility-aware applications.
- **Coda's focus**: only application-transparent mobility.

Distributed File System

Goals of CODA

- High availability (n/w partitions, disconnected operations)
- Scalability
- Transparency
- Consistency
- Conflicting detection and resolution.
- Security

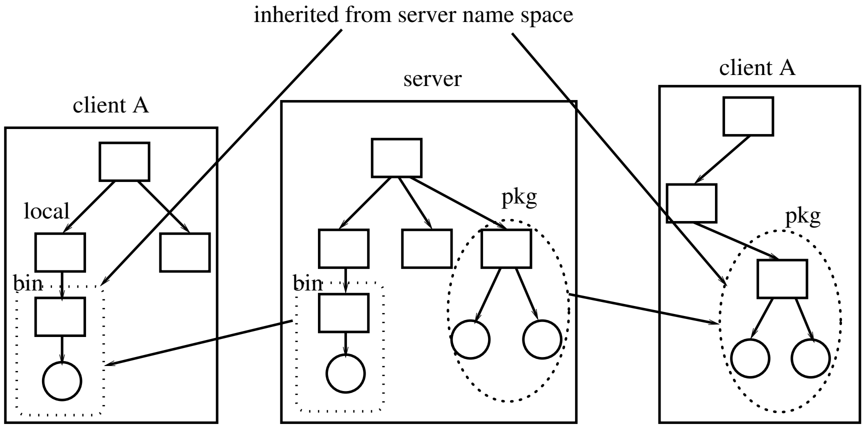
Distributed File System

Coda File System

- A client-side cache manager coupled with hoarding.
- Supports disconnected operation
- Makes sharing and collaborative computation easy in mobile computing environment.

Overview of Coda

Naming



Overview of Coda

Naming

- Files are grouped into volumes which is similar to unix partition but of smaller granularity.
- Basic units are volumes from which entire name space constructed.
- The construction takes place by mounting volumes at mount points.
- Mount points in Coda is a leaf node of one volume which becomes root node of another.

Overview of Coda

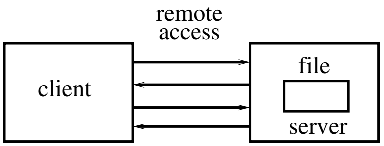
NFS and Coda Comparison

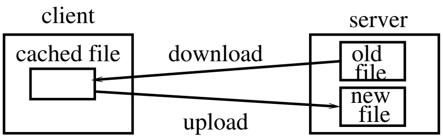
- A client uses mount commands on per server basis.
- File system should be exported.
- Based on RPC.
- Uses strong consistency.

- Knowing mount point of `/coda` suffices.
- Client-side component fetchs required file from appropriate server.
- Based on RPC2.
- Weak consistency.

Overview of Coda

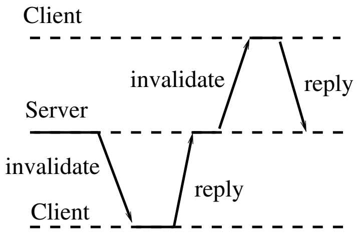
File access and update methods

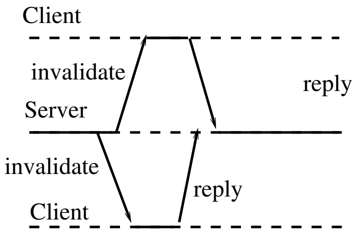




Overview of Coda

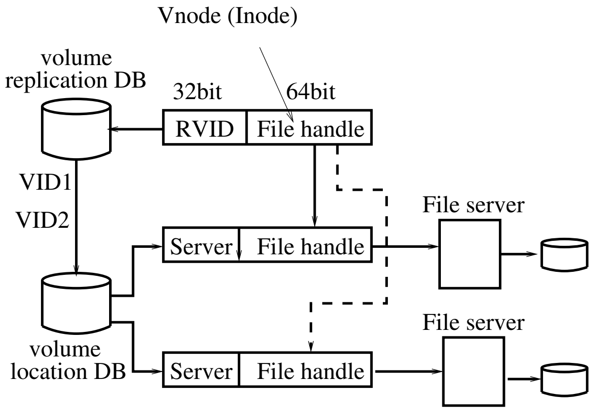
File access and update methods





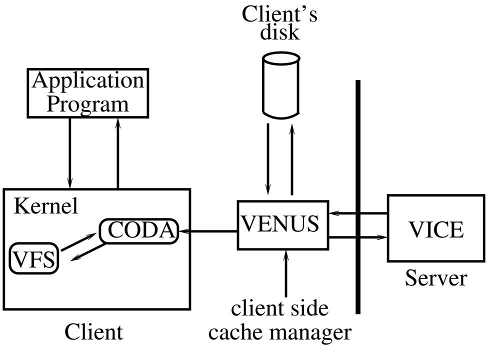
Overview of Coda

Fetching files from server



Overview of Coda

Components of Coda



Overview of Coda

Processing coda system call

- Any operation requiring access of shared objects is in granularity of files.
- User may be **connected** to network or operating in **disconnected** mode.
- Consider display of contents of a file in `/coda` directory.
- `cat <file>` generates a sequence of **system calls** with respect to the required files.
  - `open`, `read`, `close`.

Overview of Coda

Processing system calls

- The kernel at client requires VFS to open a file.
- VFS does the following:
  - Recognizes file argument is a coda resource.
  - Informs the kernel module for coda file system.
  - This module works basically a **in-kernel mini-cache**.

Overview of Coda

Role of in-kernel mini-cache

- Mini-cache is a cache of recently answered requests (for coda files) from VFS.
- If the file is locally available in cache area within the local coda file system, then VFS services the call.
- If file not locally available the mini-cache passes the request a user level program called **Venus**.
- Venus checks local disk, in the case of a cache miss contacts the server to locate the file.
- When file is located, Venus responds to kernel (mini-cache) which in turn services the client's request.

Design of Coda

Volume

- A client views coda as a single location independent Unix like file system.
- At the individual file server coda is mapped at the granularity of subtrees called **volumes**.
- A volume is a group of files and directories. located at one server
- The volume mappings are cached by Venus at the individual clients.

Design of Coda

VSG and AVSG

- Coda uses replications and callbacks for high availability and data consistency.
- The set of sites where replicas of file system volumes are placed called VSG.
- The subset of a VSG accessible to a client is called its AVSG.
- The overhead for maintaining replications at client is controlled by callback based cache coherence protocol.

Design of Coda

Replica

- Coda distinguishes between the replicas
  - **First class**: file copy at the server as
  - **second class**: cached at clients as
- If change occurs in server's replica then Venus notifies invalidation of cached copies.
- On server side updates are propagated to AVSGs then to missing VSGs.

Design of Coda

Disconnected operation

- When a client operates in disconnected mode: its AVSG is a null set.
- Clients can operate with the cached copies even when the AVSG is empty.
- Venus services the file system calls from its cache.
- The updates made by a client on the cached copies
- If cache miss occurs then failure reported.
- Update transmitted to server on reconnection.
- Venus recharges cache with server replication.

Scalability

Requirements

- The scalability requirements:
  - Ratio of clients to servers should be as large as possible.
  - Most of the load to be borne by clients.
  - Adding more clients does not significantly increase load on servers.
- OS running on a client filters out all the system calls that do not need interaction with servers.

Scalability

Filtering system calls

- Only `open` and `close` are forwarded to cache manager Venus.
- After a successful `open`, system calls `read`, `write`, `seek` can bypass Venus.
- Since servers are trusted and not loaded too much, they are responsibile for preserving integrity and security of data.
- Most obvious approach to scale up is to employ caching at the clients.

Replication and Availability

Increasing availability

- Server replications increases availability of shared data.
- When a client is connected to at least one server it depends on the server replication.
- With many server replications, the period of disconnection gets minimized.
- Disconnected clients can switch to cached copies of shared data.
- But the difficulty is in preserving data consistency.

Replication and Availability

Consistency Versus Availability

- Maintaining consistency of cached replica at clients with the corresponding server replica which receive updates, should be done cleverly.
- So, a cache coherence protocol that combines consistency (first class with second class replicas) with scaling is required.
- Updates are generated by clients, so, consistency maintenance is complex.
- Several clients may use same replicas simultaneously and without each other's knowledge.

Replication and Availability

Restrictions

- Any rapid system-wide change is disallowed.
- Updates notified only if callback is registered. All the clients are not informed.
- There is no additional update overheads – like voting, quorum eliminated completely.
- So, consensus is sacrificed in preference to efficiency.

Replication and Availability

Restrictions

- The clients cache in granularity of entire files.
  - Files once opened successfully are assured to be available in local cache.
  - Subsequent operations on cached files are protected from network failures.
  - The idea of caching file in whole compatible with disconnected mode of operation.

Replication and Availability

Caching for availability

- Caching provides client autonomy to some extent.
- The files are cached in *whole*.
- Simplifies the failure model and makes disconnected mode of operation transparent to application.
- A cache miss can occur only at `open`.
- No cache miss for `read`, `write`, `seek` or `close`.

Replication and Availability

Caching for availability

- An application running disconnected client unaware of disconnection when the required file replicas are cached.
- Coda uses additional technique called *hoarding* to pre-cache files from the server.
- Hoarding is a large grain pre-fetching.

Location Transparency

Callbacks and cache coherency

- NFS does not provide location transparency.
- In AFS (on which coda is based), files are grouped into units or volumes that reside in a single server.
- Each volume contains mount points into other volumes on different servers.
- Mount point is a key (for look up in a distributed database) valid throughout an administrative domain called cell.
- So, with callback based cache coherence movement of files between servers could be possible.
- The client is required to know mapping of files to servers and to cache this dynamically changing map.

Location Transparency

Disconnection and failures

- Goal is to make client as much autonomous as possible, so that it becomes resilience to whole range failures.
- Complete autonomy means – personal computers operating in isolation.
- No sharing of data, antithetical to very purpose of distributed file system.
- Therefore, the coda goals were two-fold
  - using shared storage repository, and
  - continuance of client's operation even in the case of network failures.

Replica Maintenance

Replica types

- The server replicas represent the **first class** replicas.
- The client replicas represent the **second class** replicas.
- Second class replicas are known only to the client where they are located.
- Second class replicas increase overall data availability.
- Second class replicas must be periodically revalidated by synching.

Replica Maintenance

Replica control strategy

- The strategy may be either *pessimistic*, or *optimistic*.
- In pessimistic control strategy update is allowed only by one node in a unique n/w partition.
  - Prior to switching into disconnected mode, exclusive/shared lock on the cached object is to be acquired.
  - Lock relinquished only after reconnection.
  - Exclusive lock disallows read or write at any other replica.
  - Shared lock allows reads but disallows writes.

Replica Maintenance

Problems with locks

- Lock is restrictive, and infeasible in context of mobile clients as disconnection can be involuntary.
- Contention for lock can be a real problem even in voluntary disconnection.
- The duration of disconnection is unpredictable.
- Allowing exclusive lock on some objects only for a brief period of disconnection is acceptable.

Replica Maintenance

Lease lock

- Lease lock can solve problems in releasing locks by disconnected clients.
- But the purpose is defeated.
  - Disconnected client may be in the midst of update when it looses the lease.
  - Then all updates made by disconnected client have to be discarded.
  - It is also possible no other client is interested for the object when the disconnected client looses the lease for the lock.

Replica Maintenance

Optimistic control

- No exclusive lock is allowed, updates performed on cached objects.
- System has to be sophisticated to detect and resolve the update conflicts in reported client updates.
- Automatic conflict resolution is not only difficult but sometimes infeasible.
- It is possible to localize the conflicts and preserve evidences for a manual repair in extreme cases.

Replica Maintenance

Optimistic control

- The strategy provides a uniform model for every client when it operates in disconnected mode.
- The fact that the cached objects are in granularity of files also helps.
- In Unix like shared file system, usually the write access is with one user, other users share the file only for read accesses.
- The client with write access updates the cached file without any creating any update conflicts.
- The clients with read access may possibly get slightly outdated values, but applications can be tuned to handle such eventualities.

Replica Maintenance

Mixed control strategy

- Employ optimistic control strategy for disconnected mode and pessimistic strategy in connected mode.
- Requires users to be adaptive to the two conflicting approaches.
- Therefore, for transparent operation in disconnected mode, a uniform model of replica control is required.

Replica Maintenance

Visibility of updates

- In Unix, any modification to a file becomes immediately visible to all users.
- Preserving this semantics is incompatible with scalability and impractical when users operate in disconnected mode.
- So, coda settled for a diluted requirement of the visibility of updates under Unix file operation semantics.

Replica Maintenance

Visibility of updates

- The difference between a cached copy at a disconnected client from its latest copy depends on time difference between the last update of the copy and the current time.
- So, can be measured as a distance function of time difference.
- Revalidating cached file at the time of opening, and propagating updates at the time closing should be adequate for most applications.
- A successful `open` means the file being accessed is the latest replication available at the server from where the client fetched it.

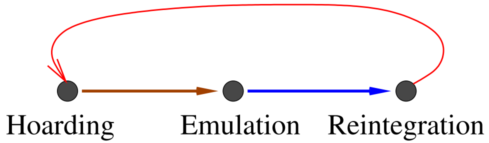
Replica Maintenance

Visibility of updates

- A server itself may have a stale copy.
- So, revalidation at the time of opening actually means fetching the latest of the replica at the accessible server site.
- Currency guarantee can be increased by querying the replica custodians on the fixed network when file opening is attempted at a client.
- This query is initiated by the server on which Venus dockets the open call.

Operation of Venus

States of operation



- Venus fetches objects from accessible servers holding replicas of requested file objects.
- It propagates updates to the servers.
- Operates in three states

Hoarding and Hoard Walk

Factors Influencing Hoarding

- User's file reference behaviour in distant future.
  - Specifying reference profile can be helpful.
- Disconnection and reconnection are purely random in time.
- What is cost of cache miss during disconnection?
- Can cache size influences in balancing caching strategy?

Hoarding and Hoard Walk

Hoard Walk

- Dynamic nature of the above factors necessitates periodic re-evaluation of priorities.
- Limited cache size disturbs equilibrium as a result of normal activities.
  - Hoard walk should be performed periodically.
- Triggering hoard walk at the time of voluntary disconnection can be specified in profile.

Emulation

Venus works as sever's proxy

- Checks the access control.
- Generates *tfid*s for newly created objects.
- Evaluates priorities of mutating objects.
- Reports or blocks a failures when a cache miss occurs.
- Records information to let the update activities done during disconnected operation to be *replayed* on reintegration.

Emulation

Cache management

- Caching is done using same priority algorithm as that for hoarding.
- Mutating objects directly update cache entries of involved objects.
- Cache entries of deleted objects are freed immediately.
- Cache entries of modified objects given infinite priorities.
- Cache miss by default reported unless explicitly masked.

Emulation

Replay log

- Per volume log is kept.
- All the operations during emulation state are logged.
- Operations replayed during the reintegration.
- Only updates are logged.
- Therefore, it is also known as change modify log (CML).
- Length of CML is reduced by applying optimization.

Emulation

Log optimization

- Operations followed by their inverses are purged from CML.
- A sequence of operations often replaced a single logical operation.
- For example, a `close` after an `open` installs a completely new file.
- `open`, the sequence of intervening `writes`, followed by a `close` replaced by a single `store`.
- `store` points to the cached copy of the file.
- A `store` invalidates the previous `stores` if any.
- So, recording the latest `store` is sufficient.

Emulation

Recovery guarantee

- In emulation state Venus must provides persistence guarantees:
  - A disconnected client to restart after a shutdown and continue from where it left off.
  - In the event of a crash while operating in disconnected state, the amount of data loss should not be more than that for an identical crash occurring in a connected state.

Emulation

RVM

- The *meta-data* used by Venus consists of
  - Cached directories and symbolic link contents,
  - Status blocks of cached objects of all types,
  - Replay logs, and
  - Hoard database (*hoard profile*) for the client.
- Meta data are mapped into address space of Venus as the RVM.
- The contents of cached files are stored as regular Unix files on client's local disk.

Emulation

RVM

- Transactional operation guarantees consistency.
- Crash recovery is ensured by log of the operation.
- Only non-nested local transactions possible.
- To reduce commit latency commits are labelled as `no-flush`
- To ensure persistence of no-flush commits log is periodically flushed.
- So RVM provides bounded persistence
  - Where bound is the period of between two flushes.

Emulation

Resource exhaustion

- Non-volatile storage may exhaust during emulation.
    - File cache become full with modified files.
    - RVM space for log becomes full.
- In case of file cache space is freed by deleting or truncating modified files.
- When log space is full no mutation is allowed until reintegration.
- Non mutating operations are allowed.

Reintegration

Role of Venus

- It changes from pseudo server state to cache manager.
- Changes during emulation state is propagated.
- Cache is updated.
- Reintegration is performed on a volume at a time.
- All update activities on volume is suspended until completion.

Reintegration

Replay algorithm

- First step is to allocate permanent fids.
  - Can be avoided if a small supply of fids cached in advance during hoarding state.
- In second step replay log is tranferred in parallel to AVSG and execute independently.
- Every server will execute replay log in one transaction.

Reintegration

Replay algorithm

- Consists of 4 phases: log parsing, validation, backfetching and commit
- Each operation is validate and executed.
  - Consists of integrity, conflict detection, protection and disk space checks.
  - Except for `store` all other operations are executed directly.
- For `store` a shadow file is created and meta data updates done.
  - Data transfer defered to phase 3.

Reintegration

Replay algorithm

- Third phase is backfetching: for data transfer.
- Final phase is releasing all locks.
- If reintegration succeeds replay log is freed.
- If reintegration fails replay log is written to a local replay file and cache entries are purged.
- So subsequent references fetches current contents of AVSG.
- Manual tool is provided for inspection of replay file.
- Selective replay is possible.