DISTRIBUTED FILE SYSTEMS

DEFINITIONS:

- A **Distributed File System** (DFS) is simply a classical model of a file system (as discussed before) distributed across multiple machines. The purpose is to promote sharing of dispersed files.

- This is an area of active research interest today.

- The resources on a particular machine are **local** to itself. Resources on other machines are **remote**.

- A file system provides a service for clients. The server interface is the normal set of file operations: create, read, etc. on files.

- Clients, servers, and storage are dispersed across machines. Configuration and implementation may vary -

  a) Servers may run on dedicated machines, OR

  b) Servers and clients can be on the same machines.

  c) The OS itself can be distributed (with the file system a part of that distribution.

  d) A distribution layer can be interposed between a conventional OS and the file system.

- Clients should view a DFS the same way they would a centralized FS; the distribution is hidden at a lower level.

- Performance is concerned with throughput and response time.

NAMING AND TRANSPARENCY:

- **Naming** is the mapping between logical and physical objects.
- Example: A user filename maps to <cylinder, sector>.
- In a conventional filesystem, it's understood where the file actually resides; the system and disk are known.
- In a **transparent** DFS, the location of a file, somewhere in the network, is hidden.
- **File replication** means multiple copies of a file; mapping returns a SET of locations for the replicas.

- **Location transparency -**

  a) The name of a file does not reveal any hint of the file's physical storage location.

  b) File name still denotes a specific, although hidden, set of physical disk blocks.

  c) This is a convenient way to share data.

  d) Can expose correspondence between component units and machines.

- **Location independence -**

  a) The name of a file doesn't need to be changed when the file's physical storage location changes. Dynamic, one-to-many mapping.

  b) Better file abstraction.

  c) Promotes sharing the storage space itself.

  d) Separates the naming hierarchy from the storage devices hierarchy.

- **Most DFSs today:**

  a) Support location transparent systems.

  b) Do NOT support **migration**; (automatic movement of a file from machine to machine.)

  c) Files are permanently associated with specific disk blocks.

- **The ANDREW DFS:**

  a) Is location independent.

  b) Supports file mobility.

- Separation of FS and OS allows for disk-less systems. These have lower cost and convenient system upgrades. The performance is not as good.

NAMING SCHEMES:

There are three main approaches to naming files:

1. Files are named with a **combination** of host and local name.

   - This guarantees a unique name. NOT location transparent NOR location independent.
   - Same naming works on local and remote files. The DFS is a loose collection of independent file systems.

2. Remote directories are **mounted** to local directories.

   - So a local system seems to have a coherent directory structure.
   - The remote directories must be explicitly mounted. The files are location independent.
   - SUN NFS is a good example of this technique.

3. A **single global name structure** spans all the files in the system.

   - The DFS is built the same way as a local filesystem. Location independent.

IMPLEMENTATION TECHNIQUES:

- Can Map directories or larger aggregates rather than individual files.
- A **non-transparent** mapping technique:

  name ----> < system, disk, cylinder, sector >

- A **transparent** mapping technique:

  name ----> file_identifier ----> < system, disk, cylinder, sector >

- So when changing the physical location of a file, only the file identifier need be modified. This identifier must be "unique" in the universe.

REMOTE FILE ACCESS:

- Reduce network traffic by retaining recently accessed disk blocks in a cache, so that repeated accesses to the same information can be handled locally.
- If required data is not already cached, a copy of data is brought from the server to the user.
- Perform accesses on the cached copy.
- Files are identified with one master copy residing at the server machine,
- Copies of (parts of) the file are scattered in different caches.
- **Cache Consistency Problem** -- Keeping the cached copies consistent with the master file.
  - A remote service ((RPC) has these characteristic steps:

    a) The client makes a request for file access.

    b) The request is passed to the server in message format.

    c) The server makes the file access.

    d) Return messages bring the result back to the client.

  - This is equivalent to performing a disk access for each request.

CACHE LOCATION:

- Caching is a mechanism for maintaining disk data on the local machine. This data can be kept in the local memory or in the local disk. Caching can be advantageous both for read ahead and read again.

- The cost of getting data from a cache is a few HUNDRED instructions; disk accesses cost THOUSANDS of instructions.

- The master copy of a file doesn’t move, but caches contain replicas of portions of the file.

- Caching behaves just like "networked virtual memory".

- What should be cached? << blocks <---> files >>. Bigger sizes give a better hit rate; smaller give better transfer times.

- Caching on disk gives:
  Better reliability.

- Caching in memory gives:

  The possibility of diskless work stations,

  Greater speed,

- Since the server cache is in memory, it allows the use of only one mechanism.

UPDATE POLICY:

- A **write through** cache has good reliability. But the user must wait for writes to get to the server. Used by NFS.

- **Delayed write** - write requests complete more rapidly. Data may be written over the previous cache write, saving a remote write. Poor reliability on a crash.

- Flush sometime later tries to regulate the frequency of writes.

- Write on close delays the write even longer.

- Which would you use for a database file? For file editing?

CACHE CONSISTENCY:

- The basic issue is, how to determine that the client-cached data is consistent with what's on the server.

- **Client - initiated approach -**

  The client asks the server if the cached data is OK. What should be the frequency of "asking"? On file open, at fixed time interval, ...?

- **Server - initiated approach -**

  Possibilities: A and B both have the same file open. When A closes the file, B "discards" its copy. Then B must start over.

  The server is notified on every open. If a file is opened for writing, then disable caching by other clients for that file.

  Get read/write permission for each block; then disable caching only for particular blocks.

COMPARISON OF CACHING AND REMOTE SERVICE:

1. Many remote accesses can be handled by a local cache. There's a great deal of locality of reference in file accesses. Servers can be accessed only occasionally rather than for each access.

2. Caching causes data to be moved in a few big chunks rather than in many smaller pieces; this leads to considerable efficiency for the network.

3. Disk accesses can be better optimized on the server if it's understood that requests are always for large contiguous chunks.

4. Cache consistency is the major problem with caching. When there are infrequent writes, caching is a win. In environments with many writes, the work required to maintain consistency overwhelms caching advantages.

5. Caching works best on machines with considerable local store - either local disks or large memories. With neither of these, use remote-service.

6. Caching requires a whole separate mechanism to support acquiring and storage of large amounts of data. Remote service merely does what's required for each call. As such, caching introduces an extra layer and mechanism and is more complicated than remote service.

STATEFUL VS. STATELESS SERVICE:

Stateful: A server keeps track of information about client requests.

- It maintains what files are opened by a client; connection identifiers; server caches.
- Memory must be reclaimed when client closes file or when client dies.

Stateless: Each client request provides complete information needed by the server (i.e., filename, file offset).

- The server can maintain information on behalf of the client, but it's not required.
- Useful things to keep include file info for the last N files touched.

Performance is better for stateful.

- Don't need to parse the filename each time, or "open/close" file on every request.
- Stateful can have a read-ahead cache.

Fault Tolerance: A stateful server loses everything when it crashes.

- Server must poll clients in order to renew its state.
- Client crashes force the server to clean up its encached information.
- Stateless remembers nothing so it can start easily after a crash.

FILE REPLICATION:

- Duplicating files on multiple machines improves availability and performance.

- Placed on failure-independent machines (they won't fail together).

  Replication management should be "location-opaque".

- The main problem is consistency - when one copy changes, how do other copies reflect that change? Often there is a tradeoff: consistency versus availability and performance.

- Example:

  "Demand replication" is like whole-file caching; reading a file causes it to be cached locally. Updates are done only on the primary file at which time all other copies are invalidated.

- Atomic and serialized invalidation isn't guaranteed (message could get lost / machine could crash.)

SUN NETWORK FILE SYSTEM:

OVERVIEW:

- Runs on SUNOS - NFS is both an implementation and a specification of how to access remote files. It's both a definition and a specific instance.

- The goal: to share a file system in a transparent way.

- Uses client-server model (for NFS, a node can be both simultaneously.) Can act between any two nodes (no dedicated server.) Mount makes a server file-system visible from a client.

    mount   server:/usr/shared   client:/usr/local

- Then, transparently, a request for /usr/local/dir-server accesses a file that is on the server.

- The mount is controlled by: (1) access rights, (2) server specification of what's mountable.

- Can use heterogeneous machines - different hardware, operating systems, network protocols.

- Uses RPC for isolation - thus all implementations must have the same RPC calls. These RPC's implement the mount protocol and the NFS protocol.

THE MOUNT PROTOCOL:

- The following operations occur:

- The client's request is sent via RPC to the mount server (on server machine.)

- Mount server checks export list containing

  (a) file systems that can be exported,
  (b) legal requesting clients.
  (c) It's legitimate to mount any directory within the legal filesystem.

- Server returns "file handle" to client.

- Server maintains list of clients and mounted directories -- this is state information! But this data is only a "hint" and isn't treated as essential.

- Mounting often occurs automatically when client or server boots.

THE NFS PROTOCOL:

- RPC’s support these remote file operations:

  a) Search for file within directory.

  b) Read a set of directory entries.

  c) Manipulate links and directories.

  d) Read/write file attributes.

  e) Read/write file data.

- Open and close are conspicuously absent from this list. NFS servers are **stateless**. Each request must provide all information. With a server crash, no information is lost.

- Modified data must actually get to server disk before client is informed the action is complete. Using a cache would imply state information.

- A single NFS write is **atomic**. A client write request may be broken into several atomic RPC calls, so the whole thing is NOT atomic. Since lock management is stateful, NFS doesn't do it. A higher level must provide this service.

NFS ARCHITECTURE:

- Follow local and remote access through this figure: **<<< Figure 17.6 >>>**
- UNIX filesystem layer - does normal open / read / etc. commands.
- Virtual file system (VFS) layer -

  (a) Gives clean layer between user and filesystem.

  (b) Acts as deflection point by using global vnodes.

  (c) Understands the difference between local and remote names.

  (d) Keeps in memory information about what should be deflected (mounted directories) and how to get to these remote directories.

- System call interface layer -

  (a) Presents sanitized validated requests in a uniform way to the VFS.

PATH-NAME TRANSLATION:

- Break the complete pathname into components.
- For each component, do an NFS lookup using the

  component name + directory vnode.

- After a mount point is reached, each component piece will cause a server access.
- Can’t hand the whole operation to server since the client may have a second mount on a subsidiary directory (a mount on a mount).
- A directory name cache on the client speeds up lookups.

CACHES OF REMOTE DATA:

- The client keeps:

  File block cache - (the contents of a file)
  File attribute cache - (file header info (inode in UNIX)).

- The local kernel hangs on to the data after getting it the first time.

- On an open, local kernel, it checks with server that cached data is still OK.

- Cached attributes are thrown away after a few seconds.

- Data blocks use read ahead and delayed write.

- Mechanism has:

  Server consistency problems.
  Good performance.

ANDREW:

A distributed environment at CMU. Strongest characteristic is scalability.

OVERVIEW:

- Machines are either servers or clients.
- Clients see a local name space and a shared name space.
- **Servers**

  run vice which presents a homogeneous, location transparent directory structure to all clients.

- **Clients** (workstations):

  Run virtue protocol to communicate with vice.

  Have local disks (1) for local name space, (2) to cache shared data.

- For scalability, off load work from servers to clients. Uses whole file caching.
- NO clients or their programs are considered trustworthy.

SHARED NAME SPACE:

- The server file space is divided into volumes. Volumes contain files of only one user. It’s these volumes that are the level of granularity attached to a client.
- A vice file can be accessed using a fid = <volume number, vnode >. The fid doesn’t depend on machine location. A client queries a volume-location database for this information.
- Volumes can migrate between servers to balance space and utilization. Old server has "forwarding" instructions and handles client updates during migration.
- Read-only volumes (system files, etc.) can be replicated. The volume database knows how to find these.

FILE OPERATIONS AND CONSISTENCY SEMANTICS:

- If a file is remote, the client operating system passes control to a client user-level process named Venus.
- The client talks to Vice server only during open/close; reading/writing are only to the local copy.
- A further optimization - if data is locally cached, it's assumed to be good until the client is told otherwise.
- A client is said to have a callback on a file.
- When a client encaches a file, the server maintains state for this fact.
- Before allowing a write to a file, the server does a callback to anyone else having this file open; all other cached copies are invalidated.
- When a client is rebooted, all cached data is suspect.
- If too much storage used by server for callback state, the server can break some callbacks.
- The system clearly has consistency concerns.

IMPLEMENTATION:

- Deflection of open/close:
- The client kernel is modified to detect references to vice files.
- The request is forwarded to Venus with these steps:
- Venus does pathname translation.
- Asks Vice for the file
- Moves the file to local disk
- Passes inode of file back to client kernel.
- Venus maintains caches for status (in memory) and data (on local disk.)
- A server user-level process handles client requests.
- A lightweight process handles concurrent RPC requests from clients.
- State information is cached in this process.
- Susceptible to reliability problems.