Silberschatz and Galvin
Chapter 17

Distributed File Systems

Distributed File Systems

- Naming and Transparency
- Remote File Access
- Stateful versus Stateless Service
- File Replication

Terminology

- *Distributed file system (DFS):* a distributed implementation of the classical time-sharing model of a file system, where multiple users share files and storage resources.
  - A DFS manages sets of dispersed storage devices.
  - Overall storage space managed by a DFS is composed of different, remotely located, smaller storage spaces.
  - There is usually a correspondence between constituent storage spaces and sets of files.

Terminology

- *Service* – software entity running on one or more machines and providing a particular type of function to *a priori* unknown clients.
- *Server* – service software running on a single machine.
- *Client* – process that can invoke a service using a set of operations that forms its *client interface.*
  - A client interface for a file service is formed by a set of primitive *file operations* (create, delete, read, write).
  - Client interface of a DFS should be transparent, i.e., not distinguish between local and remote files.
- Key performance measure: time to satisfy service requests

Naming and Transparency

- *Naming* – mapping between logical and physical objects.
  - Example: file names versus physical blocks of data stored on data tracks
- Multilevel mapping – abstraction of a file that hides the details of how and where on the disk the file is actually stored.
- A transparent DFS hides the location where in the network the file is stored.
  - For a file being replicated in several sites, the mapping returns a set of the locations of this file's replicas; both the existence of multiple copies and their location are hidden.

Naming Structures

- **Location transparency** – file name does not reveal the file's physical storage location.
  - File name still denotes a specific, although hidden, set of physical disk blocks.
  - Convenient way to share data.
  - Can expose correspondence between component units and machines.

Naming Structures

- **Location independence** – file name does not need to be changed when the file's physical storage location changes.
  - Better file abstraction.
  - Promotes sharing the storage space itself.
  - Separates the naming hierarchy from the storage-devices hierarchy.

Naming Structures

- Location independence can map same file name to different locations at different times
- Location independence is a stronger property than is location transparency
- However most current DFSs provide location transparency but not file migration; hence location independence is not relevant

Naming Structures

- Separation of name and location enables *diskless* clients
  - rely on servers to provide all files, including the operating system kernel
  - booting requires boot protocol, stored in ROM, and the kernel or boot code stored in a fixed location
  - diskless client advantages: lower cost (diminishing return with lower cost disks), less noise, easier to upgrade OS (update server copy)
  - diskless client disadvantages: added complexity of local protocols; performance loss resulting from use of network, rather than disk.

Naming Schemes

- Three main approaches to naming
  - host name, local name combination
  - attaching remote directories to local directories
  - single global name structure

Naming Schemes: host name/local name

- Files named by a combination of their host name and local name
- Guarantees a unique system-wide name
- Example (as in rcp): *host:localname*
  - dilbert:myfile.txt
  - dilbert:/etc/hosts

Naming Schemes: attach remote directory to local

- Gives the appearance of a coherent directory tree
- Automount feature
  - mounts occur on-demand based on a table of mount points and file structure names
  - previously, remote directories had to be mounted in advance
  - examples include NFS
  - issues: what to do if remote directory is (or becomes) inaccessible? Which machines are allowed to mount directory?

Naming Schemes: total integration

- A single global name structure spans all the files in the system.
- If a server is unavailable; some arbitrary set of directories on different machines also becomes unavailable.
- Special files (e.g., device files and other machine specific files) make true isomorphism difficult

Remote File Access

- Remote-service mechanism to satisfy user requests for access to remote files.
- Analogy between remote service in a DFS (perhaps implemented by RPC) and local service
  - remote service method analogous to performing a disk access for each access request
- Caching: improve performance by reducing both network traffic and also disk I/O

Remote File Access Caching

- 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 needed data not already cached, a copy of data is brought from the server to the user.
  - Accesses are performed on the cached copy.
  - Replacement policy keeps cache size bounded.
  - Files identified with one master copy residing at the server machine, but copies of (parts of) the file are scattered in different caches.

Remote File Access: Caching

- *Cache-consistency problem* – keeping the cached copies consistent with the master file.

Remote File Access: Cache Location

- Cached data can be stored on disk or in memory.
- In practice, though, many are hybrids.
- Advantages of disk caches
  - More reliable.
  - Cached data kept on disk are still there during recovery and don't need to be fetched again.

Remote File Access: Cache Location

- Advantages of main-memory caches:
  - Permit workstations to be diskless.
  - Data can be accessed more quickly.
  - Performance speedup in bigger memories.
  - Server caches (used to speed up disk I/O) are in main memory regardless of where user caches are located; using main-memory caches on the user machine permits a single caching mechanism for servers and users since server caches (e.g., to speed up disk I/O) will be in main memory.

Remote File Access: Cache Update Policy

- *Write-through* – write data through to disk as soon as they are placed on any cache. Reliable, but poor performance.
- *Delayed-write* – modifications written to the cache and then written through to the server later. Write accesses complete quickly; some data may be overwritten before they are written back, and so need never be written at all.
  - Poor reliability; unwritten data will be lost if a user machine crashes
  - Variation – write modified data blocks when ejecting from client's cache. However, some blocks may reside in cache a long time.
  - Variation – scan cache at regular intervals and flush blocks that have been modified since the last scan.
  - Variation – *write-on-close*, writes data back to the server when the file is closed. Best for files that are open for long periods and frequently modified.

Remote File Access: Consistency

- Is locally cached copy of the data consistent with the master copy?
- Client-initiated approach
  - Client initiates a validity check.
  - Server checks whether the local data are consistent with the master copy.
  - May load network and server.
- Server-initiated approach
  - Server records, for each client, the (parts of) files it caches.
  - When server detects a potential inconsistency, it must react (for example, notification)

Remote File Access:
Comparing Caching and Remote Service

- In caching, many remote accesses handled efficiently by the local cache; most remote accesses will be served as fast as local ones.
- Servers are contacted only occasionally in caching (rather than for each access).
  - Reduces server load and network traffic.
  - Enhances potential for scalability.
- Remote server method handles every remote access across the network; penalty in network traffic, server load, and performance.

Remote File Access:
Comparing Caching and Remote Service

- Total network overhead in transmitting big chunks of data (caching) is lower than a series of responses to specific requests (remote-service).
- Caching is superior in access patterns with infrequent writes.
- With frequent writes, substantial overhead incurred to overcome cache-consistency problem.

Remote File Access:
Comparing Caching and Remote Service

- Benefit from caching when execution carried out on machines with either local disks or large main memories.
- Remote access on diskless, small-memory-capacity machines should be done through remote-service method.
- In caching, the lower inter-machine interface is different from the upper user interface (data transferred en masse between server and client)
- In remote-service, the inter-machine interface mirrors the local user-file-system interface (data transferred in response to client's request)

Stateful and Stateless File Service

- Stateful file service: server tracks each file being accessed by each client
- Stateless file service: server simply provides blocks as they are requested by the client without knowledge of the blocks' use

Stateful File Service

- Mechanism.
  - Client opens a file.
  - Server fetches information about the file from its disk, stores it in its memory, and gives the client a connection identifier unique to the client and the open file.
  - Identifier is used for subsequent accesses until the session ends.
  - Server must reclaim the main-memory space used by clients who are no longer active.

Stateful File Service

- Increased performance.
  - Fewer disk accesses because file information is cached in main memory.
  - Stateful server knows if a file was opened for sequential access and can thus read ahead the next blocks.
- Key point: main-memory information is kept by a server about its clients

Stateless File Server

- Avoids state information by making each request self-contained.
- Each request identifies the file and position in the file.
- No need to establish and terminate a connection by open and close operations.

Distinctions between Stateful and Stateless Service

- Failure Recovery.
  - A stateful server loses all its volatile state in a crash.
    - Restore state by recovery protocol based on a dialog with clients, or abort operations that were underway when the crash occurred.
    - Server needs to be aware of client failures in order to reclaim space allocated to record the state of crashed client processes (orphan detection and elimination).
  - With stateless server, the effects of server failures and recovery are almost unnoticeable. A newly reincarnated server can respond to a self-contained request without any difficulty.

Distinctions between Stateful and Stateless Service

- Penalties for using the robust stateless service:
  - longer request messages
  - slower request processing
  - additional constraints imposed on DFS design
    - each request identifies the target file so a uniform, system-wide, low-level naming scheme is required
    - client operations must be *idempotent* since they may be retransmitted
      - idempotent: each operation has the same effect and produces the same output if executed several times consecutively

Distinctions between Stateful and Stateless Service

- Some environments require stateful service.
  - A server employing server-initiated cache validation cannot provide stateless service, since it maintains a record of which files are cached by which clients.
  - UNIX use of file descriptors and implicit offsets is inherently stateful; servers must maintain tables to map the file descriptors to inodes, and store the current offset within a file.

File Replication

- Replicas of the same file on different machines
  - Failure-independent machines (i.e., availability of one replica is independent from availability of others)
  - Improves availability
  - Can shorten service times
- Naming scheme maps a replicated file name to a particular replica.
  - Existence of replicas should be invisible to higher levels.
  - Replicas must be distinguished from one another by different lower-level names.

File Replication

- Updates – replicas of a file denote the same logical entity, and thus an update to any replica must be reflected on all other replicas.
- Demand replication – reading a non-local replica causes it to be cached locally, thereby generating a new non-primary replica.