Module 17: Distributed-File Systems

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

Background

- *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.

DFS Structure

- *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.

Naming and Transparency

- *Naming* – mapping between logical and physical objects.
- 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.
- **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 Schemes — Three Main Approaches

- Files named by combination of their host name and local name; guarantees a unique systemwide name.
- Attach remote directories to local directories, giving the appearance of a coherent directory tree; only previously mounted remote directories can be accessed transparently.
- Total integration of the component file systems.
  - 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.

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 needed data not already cached, a copy of data is brought from the server to the user.
  - Accesses are performed on the cached copy.
  - Files identified with one master copy residing at the server machine, but copies of (parts of) the file are scattered in different caches.
- *Cache-consistency problem* – keeping the cached copies consistent with the master file.

Location – Disk Caches vs. Main Memory Cache

- Advantages of disk caches
  - More reliable.
  - Cached data kept on disk are still there during recovery and don't need to be fetched again.
- 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.

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 whenever a user machine crashes.
  - 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.

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.
- Server-initiated approach
  - Server records, for each client, the (parts of) files it caches.
  - When server detects a potential inconsistency, it must react.

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.
- Total network overhead in transmitting big chunks of data (caching) is lower than a series of responses to specific requests (remote-service).

Caching and Remote Service (Cont.)

- Caching is superior in access patterns with infrequent writes. With frequent writes, substantial overhead incurred to overcome cache-consistency problem.
- 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 intermachine interface is different from the upper user interface.
- In remote-service, the intermachine interface mirrors the local user-file-system interface.

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.
- Increased performance.
  - Fewer disk accesses.
  - Stateful server knows if a file was opened for sequential access and can thus read ahead the next blocks.

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 & 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 (Cont.)

- Penalties for using the robust stateless service:
  - longer request messages
  - slower request processing
  - additional constraints imposed on DFS design
- 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 reside on failure-independent machines.
- Improves availability and can shorten service time.
- 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.
- 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 nonlocal replica causes it to be cached locally, thereby generating a new nonprimary replica.

Example Systems

- UNIX United
- The Sun Network File System (NFS)
- Andrew
- Sprite
- Locus

UNIX United

- Early attempt to scale up UNIX to a distributed file system without modifying the UNIX kernel.
- Adds software subsystem to set of interconnected UNIX systems (*component* or *constituent* systems).
- Constructs a distributed system that is functionally indistinguishable from conventional centralized UNIX system.
- Interlinked UNIX systems compose a UNIX United system joined together into a single naming structure, in which each component system functions as a directory.
- The component unit is a complete UNIX directory tree belonging to a certain machine; position of component units in naming hierarchy is arbitrary.

UNIX United (Cont.)

- Roots of component units are assigned names so that they become accessible and distinguishable externally.
- Traditional root directories (e.g., */dev*, */temp*) are maintained for each machine separately.
- Each component system has own set of named users and own administrator (superuser).
- Superuser is responsible for accrediting users of his own system, as well as for remote users.

UNIX United (Cont.)

- The Newcastle Connection – user-level software layer incorporated in each component system. This layer:
  - Separates the UNIX kernel and the user-level programs.
  - Intercepts all system calls concerning files, and filters out those that have to be redirected to remote systems.
  - Accepts system calls that have been directed to it from other systems.

The Sun Network File System (NFS)

- An implementation and a specification of a software system for accessing remote files across LANs (or WANs).
- The implementation is part of the SunOS operating system (version of 4.2BSD UNIX), running on a Sun workstation using an unreliable datagram protocol (UDP/IP protocol) and Ethernet.

NFS (Cont.)

- Interconnected workstations viewed as a set of independent machines with independent file systems, which allows sharing among these file systems in a transparent manner.
  - A remote directory is mounted over a local file system directory. The mounted directory looks like an integral subtree of the local file system, replacing the subtree descending from the local directory.
  - Specification of the remote directory for the mount operation is nontransparent; the host name of the remote directory has to be provided. Files in the remote directory can then be accessed in a transparent manner.
  - Subject to access-rights accreditation, potentially any file system (or directory within a file system), can be mounted remotely on top of any local directory.

NFS (Cont.)

- NFS is designed to operate in a heterogeneous environment of different machines, operating systems, and network architectures; the NFS specification is independent of these media.
- This independence is achieved through the use of RPC primitives built on top of an External Data Representation (XDR) protocol used between two implementation-independent interfaces.
- The NFS specification distinguishes between the services provided by a mount mechanism and the actual remote-file-access services.

NFS Mount Protocol

- Establishes initial logical connection between server and client.
- Mount operation includes name of remote directory to be mounted and name of server machine storing it.
  - Mount request is mapped to corresponding RPC and forwarded to mount server running on server machine.
  - *Export list* – specifies local file systems that server exports for mounting, along with names of machines that are permitted to mount them.
- Following a mount request that conforms to its export list, the server returns a *file handle*—a key for further accesses.
- File handle – a file-system identifier, and an inode number to identify the mounted directory within the exported file system.
- The mount operation changes only the user's view and does not affect the server side.

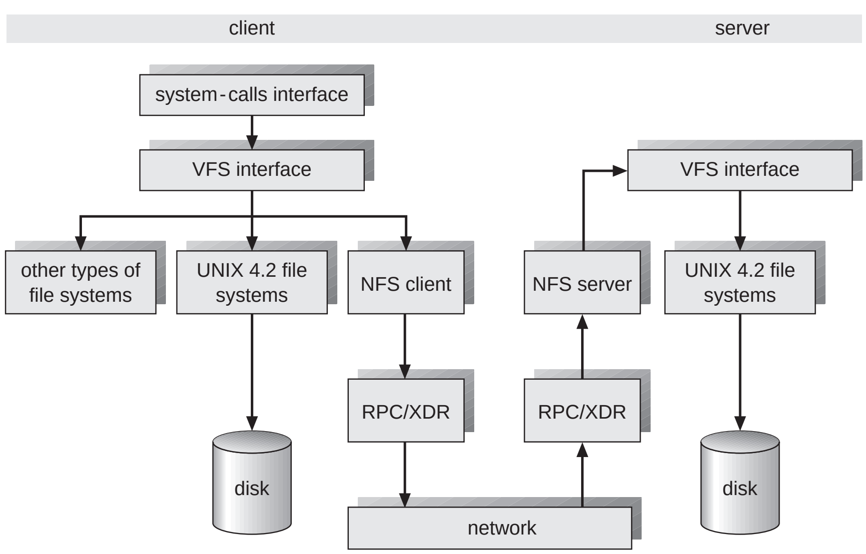
NFS Protocol

- Provides a set of remote procedure calls for remote file operations. The procedures support the following operations:
  - searching for a file within a directory
  - reading a set of directory entries
  - manipulating links and directories
  - accessing file attributes
  - reading and writing files
- NFS servers are *stateless*; each request has to provide a full set of arguments.
- Modified data must be committed to the server's disk before results are returned to the client (lose advantages of caching).
- The NFS protocol does not provide concurrency-control mechanisms.

Three Major Layers of NFS Architecture

- UNIX file-system interface (based on the open, read, write, and close calls, and file descriptors).
- *Virtual File System* (VFS) layer – distinguishes local files from remote ones, and local files are further distinguished according to their file-system types.
  - The VFS activates file-system-specific operations to handle local requests according to their file-system types.
  - Calls the NFS protocol procedures for remote requests.
- NFS service layer – bottom layer of the architecture; implements the NFS protocol.

Schematic View of NFS Architecture



NFS Path-Name Translation

- Performed by breaking the path into component names and performing a separate NFS *lookup* call for every pair of component name and directory vnode.
- To make lookup faster, a directory name lookup cache on the client's side holds the vnodes for remote directory names.

NFS Remote Operations

- Nearly one-to-one correspondence between regular UNIX system calls and the NFS protocol RPCs (except opening and closing files).
- NFS adheres to the remote-service paradigm, but employs buffering and caching techniques for the sake of performance.
- File-blocks cache – when a file is opened, the kernel checks with the remote server whether to fetch or revalidate the cached attributes. Cached file blocks are used only if the corresponding cached attributes are up to date.
- File-attribute cache – the attribute cache is updated whenever new attributes arrive from the server.
- Clients do not free delayed-write blocks until the server confirms that the data have been written to disk.

ANDREW

- A distributed computing environment under development since 1983 at Carnegie-Mellon University.
- Andrew is highly scalable; the system is targeted to span over 5000 workstations.
- Andrew distinguishes between client machines (workstations) and dedicated *server machines*. Servers and clients run the 4.2BSD UNIX OS and are interconnected by an internet of LANs.

ANDREW (Cont.)

- Clients are presented with a partitioned space of file names: a *local name space* and a *shared name space*.
- Dedicated servers, called *Vice*, present the shared name space to the clients as an homogeneous, identical, and location transparent file hierarchy.
- The local name space is the root file system of a workstation, from which the shared name space descends.
- Workstations run the *Virtue* protocol to communicate with Vice, and are required to have local disks where they store their local name space.
- Servers collectively are responsible for the storage and management of the shared name space.

ANDREW (Cont.)

- Clients and servers are structured in clusters interconnected by a backbone LAN.
- A cluster consists of a collection of workstations and a *cluster server* and is connected to the backbone by a *router*.
- A key mechanism selected for remote file operations is *whole file caching*. Opening a file causes it to be cached, in its entirety, on the local disk.

ANDREW Shared Name Space

- Andrew's volumes are small component units associated with the files of a single client.
- A *fid* identifies a Vice file or directory. A fid is 96 bits long and has three equal-length components:
  - *volume number*
  - *vnode number* – index into an array containing the inodes of files in a single volume.
  - *uniquifier* – allows reuse of vnode numbers, thereby keeping certain data structures compact.
- Fids are location transparent; therefore, file movements from server to server do not invalidate cached directory contents.
- Location information is kept on a volume basis, and the information is replicated on each server.

ANDREW File Operations

- Andrew caches entire files from servers. A client workstation interacts with Vice servers only during opening and closing of files.
- *Venus* – caches files from Vice when they are opened, and stores modified copies of files back when they are closed.
- Reading and writing bytes of a file are done by the kernel without Venus intervention on the cached copy.
- Venus caches contents of directories and symbolic links, for path-name translation.
- Exceptions to the caching policy are modifications to directories that are made directly on the server responsible for that directory.

ANDREW Implementation

- Client processes are interfaced to a UNIX kernel with the usual set of system calls.
- Venus carries out path-name translation component by component.
- The UNIX file system is used as a low-level storage system for both servers and clients. The client cache is a local directory on the workstation's disk.
- Both Venus and server processes access UNIX files directly by their inodes to avoid the expensive path name-to-inode translation routine.

ANDREW Implementation (Cont.)

- Venus manages two separate caches:
  - one for status
  - one for data
- LRU algorithm used to keep each of them bounded in size
- The status cache is kept in virtual memory to allow rapid servicing of *stat* (file status returning) system calls.
- The data cache is resident on the local disk, but the UNIX I/O buffering mechanism does some caching of the disk blocks in memory that are transparent to Venus.

SPRITE

- An experimental distributed OS under development at the Univ. of California at Berkeley; part of the Spur project – design and construction of a high-performance multiprocessor workstation.
- Targets a configuration of large, fast disks on a few servers handling storage for hundreds of diskless workstations which are interconnected by LANs.
- Because file caching is used, the large physical memories compensate for the lack of local disks.
- Interface similar to UNIX; file system appears as a single UNIX tree encompassing all files and devices in the network, equally and transparently accessible from every workstation.
- Enforces consistency of shared files and emulates a single time-sharing UNIX system in a distributed environment.

SPRITE (Cont.)

- Uses *backing files* to store data and stacks of running processes, simplifying process migration and enabling flexibility and sharing of the space allocated for swapping.
- The virtual memory and file system share the same cache and negotiate on how to divide it according to their conflicting needs.
- Sprite provides a mechanism for sharing an address space between client processes on a single workstation (in UNIX, only code can be shared among processes).

SPRITE Prefix Tables

- A single file-system hierarchy composed of several subtrees called *domains* (component units), with each server providing storage for one or more domains.
- Prefix table – a server map maintained by each machine to map domains to servers.
- Each entry in a prefix table corresponds to one of the domains. It contains:
  - the name of the topmost directory in the domain (prefix for the domain).
  - the network address of the server storing the domain.
  - a numeric designator identifying the domain's root directory for the storing server.
- The prefix mechanism ensures that the domain's files can be opened and accessed from any machine regardless of the status of the servers of domains above the particular domain.

SPRITE Prefix Tables (Cont.)

- Lookup operation for an absolute path name:
  - Client searches its prefix table for the longest prefix matching the given file name.
  - Client strips the matching prefix from the file name and sends the remainder of the name to the selected server along with the designator from the prefix-table entry.
  - Server uses this designator to locate the root directory of the domain, and then proceeds by usual UNIX path-name translation for the remainder of the file name.
  - If server succeeds in completing the translation, it replies with a designator for the open file.

Cases Where Server Does Not Complete Lookup

- Server encounters an absolute path name in a symbolic link. Absolute path name returned to client, which looks up the new name in its prefix table and initiates another lookup with a new server.
- If a path name ascends past the root of a domain, the server returns the remainder of the path name to the client, which combines the remainder with the prefix of the domain that was just exited to form a new absolute path name.
- If a path name descends into a new domain or if a root of a domain is beneath a working directory and a file in that domain is referred to with a relative path name, a *remote link* (a special marker file) is placed to indicate domain boundaries. When a server encounters a remote link, it returns the file name to the client.

Incomplete Lookup (Cont.)

- When a remote link is encountered by the server, it indicates that the client lacks an entry for a domain — the domain whose remote link was encountered.
- To obtain the missing prefix information, a client broadcasts a file name.
  - *broadcast* – network message seen by all systems on the network.
  - The server storing that file responds with the prefix-table entry for this file, including the string to use as a prefix, the server's address, and the descriptor corresponding to the domain's root.
  - The client then can fill in the details in its prefix table.

SPRITE Caching and Consistency

- Capitalizing on the large main memories and advocating diskless workstations, file caches are stored in memory, instead of on local disks.
- Caches are organized on a block (4K) basis, rather than on a file basis.
- Each block in the cache is virtually addressed by the file designator and a block location within the file; enables clients to create new blocks in the cache and to locate any block without the file inode being brought from the server.
- A delayed-write approach is used to handle file modification.

SPRITE Caching and Consistency (Cont.)

- Consistency of shared files enforced through version-number scheme; a file's version number is incremented whenever a file is opened in write mode.
- Notifying the servers whenever a file is opened or closed prohibits performance optimizations such as name caching.
- Servers are centralized control points for cache consistency; they maintain state information about open files.

LOCUS

- Project at the Univ. of California at Los Angeles to build a full-scale distributed OS; upward-compatible with UNIX, but the extensions are major and necessitate an entirely new kernel.
- File system is a single tree-structure naming hierarchy which covers all objects of all the machines in the system.
- Locus names are fully transparent.
- A Locus file may correspond to a set of copies distributed on different sites.
- File replication increases availability for reading purposes in the event of failures and partitions.
- A primary-copy approach is adopted for modifications.

LOCUS (Cont.)

- Locus adheres to the same file-access semantics as standard UNIX.
- Emphasis on high performance led to the incorporation of networking functions into the operating system.
- Specialized remote operations protocols used for kernel-to-kernel communication, rather than the RPC protocol.
- Reducing the number of network layers enables performance for remote operations, but this specialized protocol hampers the portability of Locus.

LOCUS Name Structure

- Logical filegroups form a unified structure that disguises location and replication details from clients and applications.
- A logical filegroup is mapped to multiple *physical containers* (or *packs*) that reside at various sites and that store file replicas of that filegroup.
- The <logical-filegroup-number, inode number> (the file's *designator*) serves as a globally unique low-level name for a file.

LOCUS Name Structure (Cont.)

- Each site has a consistent and complete view of the logical name structure.
  - Globally replicated logical mount table contains an entry for each logical filegroup.
  - An entry records the file designator of the directory over which the filegroup is logically mounted, and indication of which site is currently responsible for access synchronization within the filegroup.
- An individual pack is identified by pack numbers and a logical filegroup number.
- One pack is designated as the *primary copy*.
  - a file must be stored at the primary copy site
  - a file can be stored also at any subset of the other sites where there exists a pack corresponding to its filegroup.

LOCUS Name Structure (Cont.)

- The various copies of a file are assigned the same inode number on all the filegroup's packs.
  - Reference over the network to data pages use logical, rather than physical, page numbers.
  - Each pack has a mapping of these logical numbers to its physical numbers.
  - Each inode of a file copy contains a version number, determining which copy dominates other copies.
- Container table at each site maps logical filegroup numbers to disk locations for the filegroups that have packs locally on this site.

LOCUS File Access

- Locus distinguishes three logical roles in file accesses, each one potentially performed by a different site:
  - **Using site** (US) – issues requests to open and access a remote file.
  - **Storage site** (SS) – site selected to serve requests.
  - **Current synchronization site** (CSS) – maintains the version number and a list of physical containers for every file in the filegroup.
    - Enforces global synchronization policy for a filegroup.
    - Selects an SS for each open request referring to a file in the filegroup.
    - At most one CSS for each filegroup in any set of communicating sites.

LOCUS Synchronized Accesses to Files

- Locus tries to emulate conventional UNIX semantics on file accesses in a distributed environment.
  - Multiple processes are permitted to have the same file open concurrently.
  - These processes issue read and write system calls.
  - The system guarantees that each successive operation sees the effects of the ones that precede it.
- In Locus, the processes share the same operating-system data structures and caches, and by using locks on data structures to serialize requests.

LOCUS Two Sharing Modes

- A single token scheme allows several processes descending from the same ancestor to share the same position (offset) in a file. A site can proceed to execute system calls that need the offset only when the token is present.
- A multiple-data-tokens scheme synchronizes sharing of the file's in-core inode and data.
  - Enforces a single exclusive-writer, multiple-readers policy.
  - Only a site with the write token for a file may modify the file, and any site with a read token can read the file.
- Both token schemes are coordinated by token managers operating at the corresponding storage sites.

LOCUS Operation in a Faulty Environment

- Maintain, within a single partition, strict synchronization among copies of a file, so that all clients of that file within that partition see the most recent version.
- Primary-copy approach eliminates conflicting updates, since the primary copy must be in the client's partition to allow an update.
- To detect and propagate updates, the system maintains a *commit count* which enumerates each commit of every file in the filegroup.
- Each pack has a *lower-water mark* (*lwm*) that is a commit-count value, up to which the system guarantees that all prior commits are reflected in the pack.