CHAPTER 16: DISTRIBUTED-SYSTEM STRUCTURES

- Network-Operating Systems
- Distributed-Operating Systems
- Remote Services
- Robustness
- Design Issues

Network-Operating Systems – users are aware of multiplicity of machines. Access to resources of various machines is done explicitly by:

- Remote logging into the appropriate remote machine.
- Transferring data from remote machines to local machines, via the File Transfer Protocol (FTP) mechanism.

Distributed-Operating Systems – users not aware of multiplicity of machines. Access to remote resources similar to access to local resources.

- Data Migration – transfer data by transferring entire file, or transferring only those portions of the file necessary for the immediate task.
- Computation Migration – transfer the computation, rather than the data, across the system.
- Process Migration – execute an entire process, or parts of it, at different sites.
  - Load balancing – distribute processes across network to even the workload.
  - Computation speedup – subprocesses can run concurrently on different sites.
  - Hardware preference – process execution may require specialized processor.
  - Software preference – required software may be available at only a particular site.
  - Data access – run process remotely, rather than transfer all data locally.

Remote Services

- Requests for access to a remote file are delivered to the server. Access requests are translated to messages for the server, and the server replies are packed as messages and sent back to the user.

- A common way to achieve this is via the *Remote Procedure Call* (RPC) paradigm.

- Messages addressed to an RPC daemon listening to a *port* on the remote system contain the name of a process to run and the parameters to pass to that process. The process is executed as requested, and any output is sent back to the requester in a separate message.

- A *port* is a number included at the start of a message packet. A system can have many ports within its one network address to differentiate the network services it supports.

The RPC scheme requires binding client and server port.

- Binding information may be predecided, in the form of fixed port addresses.
  - At compile time, an RPC call has a fixed port number associated with it.
  - Once a program is compiled, the server cannot change the port number of the requested service.
- Binding can be done dynamically by a rendezvous mechanism.
  - Operating system provides a rendezvous daemon on a fixed RPC port.
  - Client then sends a message to the rendezvous daemon requesting the port address of the RPC it needs to execute.

- A distributed file system (DFS) can be implemented as a set of RPC daemons and clients.
  - The messages are addressed to the DFS port on a server on which a file operation is to take place.
  - The message contains the disk operation to be performed (i.e., **read**, **write**, **rename**, **delete**, or **status**).
  - The return message contains any data resulting from that call, which is executed by the DFS daemon on behalf of the client.

Threads

- Threads can send and receive messages while other operations within the task continue asynchronously.

- *Pop-up thread* – created on ‘‘as needed’’ basis to respond to new RPC.
  - Cheaper to start new thread than to restore existing one.
  - No threads block waiting for new work; no context has to be saved, or restored.
  - Incoming RPCs do not have to be copied to a buffer within a server thread.

- RPCs to processes on the same machine as the caller made more lightweight via shared memory between threads in different processes running on same machine.

The DCE thread calls

1. Thread-management:

   create, exit, join, detach

2. Synchronization:

   mutex_init, mutex_destroy, mutex_lock, mutex_trylock, mutex_unlock

3. Condition-variable:

   cond_init, cond_destroy, cond_wait, cond_signal, cond_broadcast

4. Scheduling:

   setscheduler, getscheduler, setprio, getprio

5. Kill-thread:

   cancel, setcancel

Robustness

To ensure that the system is robust, we must :

- *Detect* failures.
  - link
  - site

- *Reconfigure* the system so that computation may continue.

- *Recover* when a site or a link is repaired.

Failure Detection – To detect link and site failure, we use a *handshaking* procedure.

- At fixed intervals, sites A and B send each other an *I-am-up* message. If site *A* does not receive this message within a predetermined time period, it can assume that site *B* has failed, that the link between *A* and *B* has failed, or that the message from *B* has been lost.

- At the time site A sends the *Are-you-up?* message, it specifies a time interval during which it is willing to wait for the reply from *B*. If A does not receive B's reply message within the time interval, A may conclude that one or more of the following situations has occurred:
  1. Site *B* is down.
  2. The direct link (if one exists) from *A* to *B* is down.
  3. The alternative path from *A* to *B* is down.
  4. The message has been lost.

Reconfiguration – Procedure that allows the system to reconfigure and to continue its normal mode of operation.

- If a direct link from *A* to *B* has failed, this information must be broadcast to every site in the system, so that the various routing tables can be updated accordingly.

- If it is believed that a site has failed (because it can no longer be reached), then every site in the system must be so notified, so that they will no longer attempt to use the services of the failed site.

Recovery from Failure – When a failed link or site is repaired, it must be integrated into the system gracefully and smoothly.

- Suppose that a link between *A* and *B* has failed. When it is repaired, both *A* and *B* must be notified. We can accomplish this notification by continuously repeating the handshaking procedure.

- Suppose that site *B* has failed. When it recovers, it must notify all other sites that it is up again. Site *B* then may have to receive from the other sites various information to update its local tables.

Design Issues

- Transparency and locality – distributed system should look like conventional, centralized system and not distinguish between local and remote resources.
- User mobility – brings user's environment (i.e., home directory) to wherever the user logs in.
- Fault tolerance – system should continue functioning, perhaps in a degraded form, when faced with various types of failures.
- Scalability – system should adapt to increased service load.
- Large-scale systems – service demand from any system component should be bounded by a constant that is independent of the number of nodes.
- Process structure of the server – servers should operate efficiently in peak periods; use of lightweight processes or threads.

CHAPTER 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.

Comparison of 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 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 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.

- 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.

- 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.

- 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.

- 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 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.

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.
  - A mount request is mapped to the corresponding RPC and forwarded to the mount server running on the 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* that serves as the 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 (the advantages of caching are lost).

- The NFS protocol does not provide concurrency-control mechanisms.

Three Major Layers of NFS Architecture

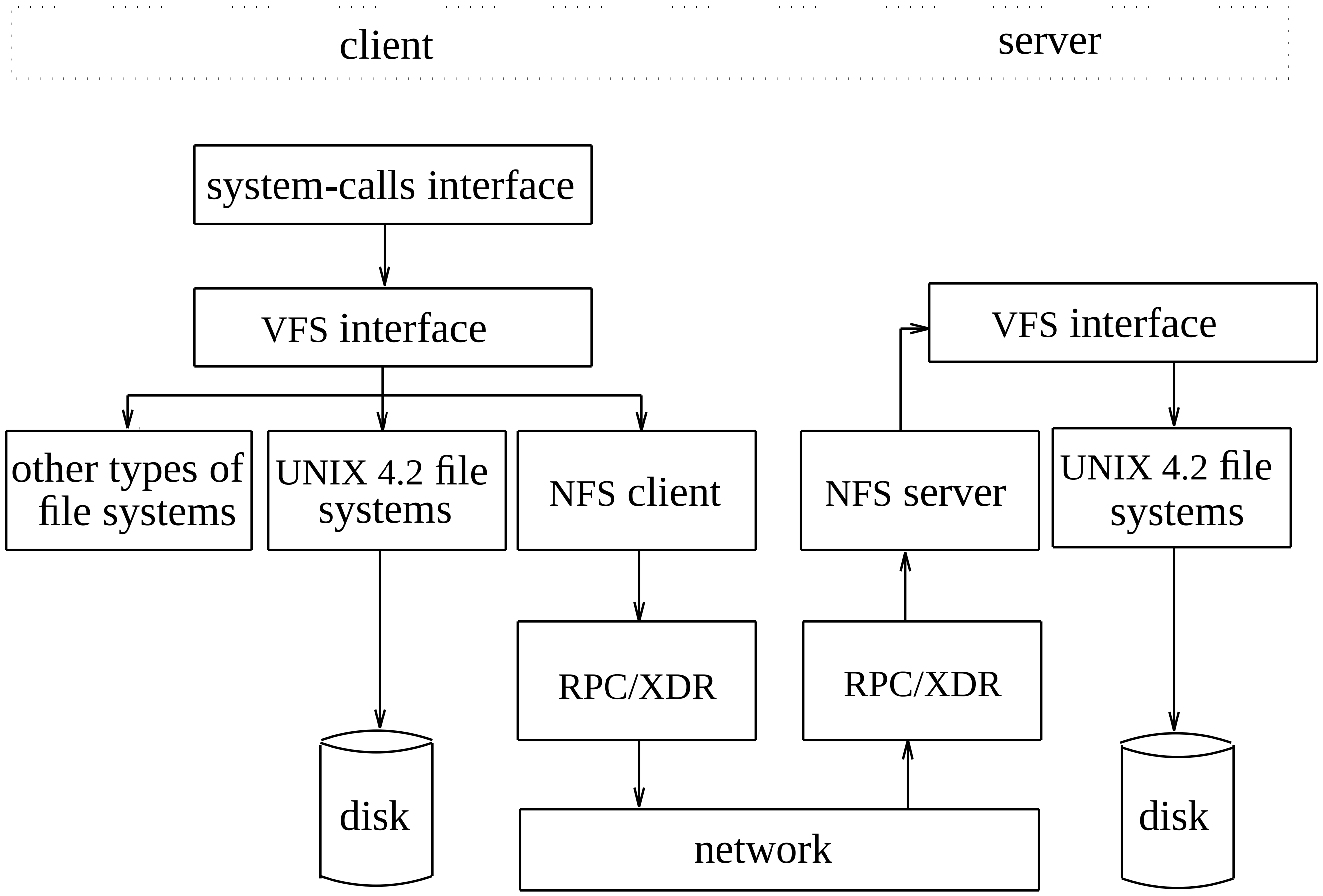
1. UNIX file-system interface (based on the open, read, write, and close calls, and file descriptors).

2. *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, and calls the NFS protocol procedures for remote requests.

3. NFS service layer – bottom layer of the architecture; implements the NFS protocol.

Schematic View of the NFS Architecture



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.

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.

- 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.

- 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.

Shared Name Space – *volumes*

- 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.

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.

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.

- 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.

- 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).

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.

- Lookup operation for an absolute path name:

  1. Client searches its prefix table for the longest prefix matching the given file name.

  2. 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.

  3. 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.

  4. If server succeeds in completing the translation, it replies with a designator for the open file.

- Cases where the server does not complete the 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.

- 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.

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.

- 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 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.

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.
- 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, and can be stored also at any subset of the other sites where there exists a pack corresponding to its filegroup.
- 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 distinguishes three logical roles in file accesses, each one potentially performed by a different site:

1. **Using site** (US) – issues requests to open and access a remote file.

2. **Storage site** (SS) – site selected to serve requests.

3. **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.

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.

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.

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.

CHAPTER 18: DISTRIBUTED COORDINATION

- Event Ordering
- Mutual Exclusion
- Atomicity
- Concurrency Control
- Deadlock Handling
- Election Algorithms
- Reaching Agreement

Event Ordering

- *Happened-before* relation (denoted by $\rightarrow$).

  1. If A and B are events in the same process, and A was executed before B, then $A \rightarrow B$.

  2. If A is the event of sending a message by one process and B is the event of receiving that message by another process, then $A \rightarrow B$.

  3. If $A \rightarrow B$ and $B \rightarrow C$ then $A \rightarrow C$.

- Implementation of $\rightarrow$
  - Associate a *timestamp* with each system event. Require that for every pair of events A and B, if $A \rightarrow B$, then the timestamp of A is less than the timestamp of B.
  - Within *each* process P_i a *logical clock*, LC_i is associated. The logical clock can be implemented as a simple counter that is incremented between any two successive events executed within a process.
  - A process advances its logical clock when it receives a message whose timestamp is greater than the current value of its logical clock.
  - If the timestamps of two events A and B are the same, then the events are concurrent. We may use the process identity numbers to break ties and to create a total ordering.

Distributed Mutual Exclusion

- Assumptions
  - The system consists of n processes; each process P_i resides at a different processor.
  - Each process has a critical section that requires mutual exclusion.
- Requirement
  - If P_i is executing in its critical section, then no other process P_j is executing in its critical section.
- We present two algorithms to ensure the mutual exclusion execution of processes in their critical sections.

Centralized Approach

- One of the processes in the system is chosen to coordinate the entry to the critical section.
- A process that wants to enter its critical section sends a *request* message to the coordinator.
- The coordinator decides which process can enter the critical section next, and it sends that process a *reply* message.
- When the process receives a *reply* message from the coordinator, it enters its critical section.
- After exiting its critical section, the process sends a *release* message to the coordinator and proceeds with its execution.
- This scheme requires three messages per critical-section entry:
  - request
  - reply
  - release

Fully Distributed Approach

- When process P_i wants to enter its critical section, it generates a new timestamp, *TS*, and sends the message *request*(P_i, *TS*) to all other processes in the system.

- When process P_j receives a *request* message, it may reply immediately or it may defer sending a reply back.

- When process P_i receives a *reply* message from all other processes in the system, it can enter its critical section.

- After exiting its critical section, the process sends *reply* messages to all its deferred requests.

Fully Distributed Approach (continued)

- The decision whether process P_j replies immediately to a *request*(P_i, *TS*) message or defers its reply is based on three factors:

  1. If P_j is in its critical section, then it defers its reply to P_i.
  2. If P_j does *not* want to enter its critical section, then it sends a *reply* immediately to P_i.
  3. If P_j wants to enter its critical section but has not yet entered it, then it compares its own request timestamp with the timestamp *TS*.
     - If its own request timestamp is greater than *TS*, then it sends a *reply* immediately to P_i (P_i asked first).
     - Otherwise, the reply is deferred.

Fully Distributed Approach – desirable behavior:

- Freedom from deadlock is ensured.
- Freedom from starvation is ensured, since entry to the critical section is scheduled according to the timestamp ordering. The timestamp ordering ensures that processes are served in a first-come, first-served order.
- The number of messages per critical-section entry is $2 \times (n - 1)$. This is the minimum number of required messages per critical-section entry when processes act independently and concurrently.

Fully Distributed Approach – three undesirable consequences:

1. The processes need to know the identity of all other processes in the system, which makes the dynamic addition and removal of processes more complex.

2. If one of the processes fails, then the entire scheme collapses. This can be dealt with by continuously monitoring the state of all the processes in the system.

3. Processes that have not entered their critical section must pause frequently to assure other processes that they intend to enter the critical section. This protocol is therefore suited for small, stable sets of cooperating processes.

Atomicity

- Either all the operations associated with a program unit are executed to completion, or none are performed.

- Ensuring atomicity in a distributed system requires a *transaction coordinator*, which is responsible for the following:

  - Starting the execution of the transaction.

  - Breaking the transaction into a number of subtransactions, and distributing these subtransactions to the appropriate sites for execution.

  - Coordinating the termination of the transaction, which may result in the transaction being committed at all sites or aborted at all sites.

Two-Phase Commit Protocol (2PC)

- Assumes fail-stop model.
- Execution of the protocol is initiated by the coordinator after the last step of the transaction has been reached.
- When the protocol is initiated, the transaction may still be executing at some of the local sites.
- The protocol involves all the local sites at which the transaction executed.

- Let T be a transaction initiated at site S_i, and let the transaction coordinator at S_i be C_i.

- Phase 1: Obtaining a decision
  - C_i adds <prepare T> record to the log.
  - C_i sends <prepare T> message to all sites.
  - When a site receives a <prepare T> message, the transaction manager determines if it can commit the transaction.
    - If no: add <no T> record to the log and respond to C_i with <abort T>.
    - If yes:
      - add <ready T> record to the log.
      - force *all log records* for T onto stable storage.
      - transaction manager sends <ready T> message to C_i.

Phase 1 (cont’d)

- Coordinator collects responses
  - All respond ‘‘ready’’,

    decision is *commit.*
  - At least one response is ‘‘abort’’,

    decision is *abort.*
  - At least one participant fails to respond within timeout period,

    decision is *abort.*

- Phase 2: Recording the decision in the database
  - Coordinator adds a decision record (<abort *T*> or <commit *T*>) to its log and forces record onto stable storage.
  - Once that record reaches stable storage it is irrevocable (even if failures occur).
  - Coordinator sends a message to each participant informing it of the decision (commit or abort).
  - Participants take appropriate action locallyl

Failure Handling in 2PC

- Site failure
  - The log contains a <commit *T*> record. In this case, the site executes **redo**(*T*).
  - The log contains an <abort *T*> record. In this case, the site executes **undo**(*T*).
  - The log contains a <ready *T*> record; consult C_i. If C_i is down, site sends **query-status** *T* message to the other sites.
  - The log contains no control records concerning *T*. In this case, the site executes **undo**(*T*).

- Coordinator C_i failure

  - If an active site contains a <commit T> record in its log, then T must be committed.

  - If an active site contains an <abort T> record in its log, then T must be aborted.

  - If some active site does *not* contain the record <ready T> in its log, then the failed coordinator C_i cannot have decided to commit T. Rather than wait for C_i to recover, it is preferable to abort T.

  - All active sites have a <ready T> record in their logs, but no additional control records. In this case we must wait for the coordinator to recover. *Blocking* problem – T is blocked pending the recovery of site S_i.

Concurrency Control

- Modify the centralized concurrency schemes to accommodate the distribution of transactions.
- Transaction manager coordinates execution of transactions (or subtransactions) that access data at local sites.
- Local transaction only executes at that site.
- Global transaction executes at several sites.

Locking Protocols

- Can use the two-phase locking protocol in a distributed environment by changing how the lock manager is implemented.

- Nonreplicated scheme – each site maintains a local lock manager which administers lock and unlock requests for those data items that are stored in that site.
  - Simple implementation involves two message transfers for handling lock requests, and one message transfer for handling unlock requests.
  - Deadlock handling is more complex.

- Single-coordinator approach – a single lock manager resides in a single chosen site; all lock and unlock requests are made at that site.

  - Simple implementation
  - Simple deadlock handling
  - Possibility of bottleneck
  - Vulnerable to loss of concurrency controller if single site fails.
  - *Multiple-coordinator approach* distributes lock-manager function over several sites.

- Majority protocol – avoids drawbacks of central control by dealing with replicated data in a decentralized manner.
  - More complicated to implement
  - Deadlock-handling algorithms must be modified; possible for deadlock to occur in locking only one data item.

- Biased protocol – similar to majority protocol, but requests for shared locks prioritized over requests for exclusive locks.
  - Less overhead on **read** operations than in majority protocol; but has additional overhead on **writes**.
  - Like majority protocol, deadlock handling is complex.

- Primary copy – one of the sites at which a replica resides is designated as the *primary site*. Request to lock a data item is made at the primary site of that data item.

  - Concurrency control for replicated data handled in a manner similar to that for unreplicated data.

  - Simple implementation, but if primary site fails, the data item is unavailable, even though other sites may have a replica.

Timestamping

- Generate unique timestamps in distributed scheme:
  - Each site generates a unique local timestamp.
  - The global unique timestamp is obtained by concatenation of the unique local timestamp with the unique site identifier.
  - Use a *logical clock* defined within each site to ensure the fair generation of timestamps.
- Timestamp-ordering scheme – combine the centralized concurrency control timestamp scheme (Section 6.9) with the 2PC protocol to obtain a protocol that ensures serializability with no cascading rollbacks.

Deadlock Prevention

- Resource-ordering deadlock-prevention – define a *global* ordering among the system resources.
  - Assign a unique number to all system resources.
  - A process may request a resource with unique number *i* only if it is not holding a resource with a unique number greater than *i*.
  - Simple to implement; requires little overhead.

- Banker's algorithm – designate one of the processes in the system as the process that maintains the information necessary to carry out the Banker's algorithm.
  - Also implemented easily, but may require too much overhead.

- Timestamped deadlock-prevention scheme

  - Each process P_i is assigned a unique priority number.

  - Priority numbers are used to decide whether a process P_i should wait for a process P_j. P_i can wait for P_j if P_i has a higher priority than P_j; otherwise P_i is rolled back.

  - The scheme prevents deadlocks. For every edge $P_i \rightarrow P_j$ in the wait-for graph, P_i has a higher priority than P_j. Thus, a cycle cannot exist.

- Wait-die scheme – based on a nonpreemptive technique.
  - If P_i requests a resource currently held by P_j, P_i is allowed to wait only if it has a smaller timestamp than does P_j (P_i is older than P_j). Otherwise, P_i is rolled back (dies).
  - Example: Suppose that processes P_1, P_2, and P_3 have timestamps 5, 10, and 15, respectively.
    - If P_1 requests a resource held by P_2, then P_1 will wait.
    - If P_3 requests a resource held by P_2, then P_3 will be rolled back.

- Wound-wait scheme – based on a preemptive technique; counterpart to the wait-die system.
  - If P_i requests a resource currently held by P_j, P_i is allowed to wait only if it has a larger timestamp than does P_j (P_i is younger than P_j). Otherwise, P_j is rolled back (P_j is *wounded* by P_i).
  - Example: Suppose that processes P_1, P_2, and P_3 have timestamps 5, 10, and 15, respectively.
    - If P_1 requests a resource held by P_2, then the resource will be preempted from P_2 and P_2 will be rolled back.
    - If P_3 requests a resource held by P_2, then P_3 will wait.

Deadlock Detection – Centralized Approach

- Each site keeps a *local* wait-for graph. The nodes of the graph correspond to all the processes that are currently either holding or requesting any of the resources local to that site.

- A global wait-for graph is maintained in a *single* coordination process. This global graph is the union of all the local wait-for graphs.

- There are three different options (points in time) when the wait-for graph may be constructed:
  1. Whenever a new edge is inserted or removed in one of the local wait-for graphs.
  2. Periodically, when a number of changes have occurred in a wait-for graph.
  3. Whenever the coordinator needs to invoke the cycle-detection algorithm.

- Unnecessary rollbacks may occur as a result of *false cycles*.

Detection algorithm based on option 3.

- Append unique identifiers (timestamps) to requests from different sites.

- When process P_i, at site *A*, requests a resource from process P_j, at site *B*, a request message with timestamp *TS* is sent.

- The edge $P_i \rightarrow P_j$ with the label *TS* is inserted in the local wait-for of *A*. This edge is inserted in the local wait-for graph of *B* only if *B* has received the request message and cannot immediately grant the requested resource.

The algorithm:

1. The controller sends an initiating message to each site in the system.

2. On receiving this message, a site sends its local wait-for graph to the coordinator.

3. When the controller has received a reply from each site, it constructs a graph as follows:

   a. The constructed graph contains a vertex for every process in the system.

   b. The graph has an edge $P_i \rightarrow P_j$ if and only if (1) there is an edge $P_i \rightarrow P_j$ in one of the wait-for graphs, or (2) an edge $P_i \rightarrow P_j$ with some label *TS* appears in more than one wait-for graph.

   If the constructed graph contains a cycle $\Rightarrow$ deadlock.

Fully Distributed Approach

- All controllers share equally the responsibility for detecting deadlock.
- Every site constructs a wait-for graph that represents a part of the total graph.
- We add one additional node P_{ex} to each local wait-for graph.
- If a local wait-for graph contains a cycle that does not involve node P_{ex}, then the system is in a deadlock state.
- A cycle involving P_{ex} implies the *possibility* of a deadlock. To ascertain whether a deadlock does exist, a distributed deadlock-detection algorithm must be invoked.

Election Algorithms – determine where a new copy of the coordinator should be restarted.

- Assume that a unique priority number is associated with each active process in the system, and assume that the priority number of process P_i is i.

- Assume a one-to-one correspondence between processes and sites.

- The coordinator is always the process with the largest priority number. When a coordinator fails, the algorithm must elect that active process with the largest priority number.

- Two algorithms, the bully algorithm and a ring algorithm, can be used to elect a new coordinator in case of failures.

Bully Algorithm – applicable to systems where every process can send a message to every other process in the system.

- If process P_i sends a request that is not answered by the coordinator within a time interval *T*, assume that the coordinator has failed; P_i tries to elect itself as the new coordinator.

- P_i sends an election message to every process with a higher priority number, P_i then waits for any of these processes to answer within *T*.

- If no response within *T*, assume that all processes with numbers greater than *i* have failed; P_i elects itself the new coordinator.

- If answer is received, P_i begins time interval *T'*, waiting to receive a message that a process with a higher priority number has been elected.

- If no message is sent within *T'*, assume the process with a higher number has failed; P_i should restart the algorithm.

- If P_i is not the coordinator, then, at any time during execution, P_i may receive one of the following two messages from process P_j:

  1. P_j is the new coordinator ($j > i$). P_i, in turn, records this information.

  2. P_j started an election ($j < i$). P_i sends a response to P_j and begins its own election algorithm, provided that P_i has not already initiated such an election.

- After a failed process recovers, it immediately begins execution of the same algorithm.

- If there are no active processes with higher numbers, the recovered process forces all processes with lower numbers to let it become the coordinator process, even if there is a currently active coordinator with a lower number.

Ring Algorithm – applicable to systems organized as a ring (logically or physically).

- Assumes that the links are unidirectional, and that processes send their messages to their right neighbors.

- Each process maintains an *active list*, consisting of all the priority numbers of all active processes in the system when the algorithm ends.

   1. If process P_i detects a coordinator failure, it creates a new active list that is initially empty. It then sends a message *elect*(i) to its right neighbor, and adds the number i to its active list.

2. If P_i receives a message *elect*(j) from the process on the left, it must respond in one of three ways:

    a. If this is the first *elect* message it has seen or sent, P_i creates a new active list with the numbers i and j. It then sends the message *elect*(i), followed by the message *elect*(j).

    b. If $i \neq j$, then P_i adds j to its active list and forwards the message to its right neighbor.

    c. If $i = j$, then the active list for P_i now contains the numbers of all the active processes in the system. P_i can now determine the largest number in the active list to identify the new coordinator process.

Reaching Agreement

- There are applications where a set of processes wish to agree on a common "value."

- Such agreement may not take place due to:
  - Faulty communication medium
  - Faulty processes
    - ○ Processes may send garbled or incorrect messages to other processes.
    - ○ A subset of the processes may collaborate with each other in an attempt to defeat the scheme.

Faulty Communications

- Process P_i at site *A*, has sent a message to process P_j at site *B*, and needs to know whether P_j has received the message in order to decide how to proceed.

- Detect failures using a *time-out* scheme.

  - When P_i sends out a message, it also specifies a time interval during which it is willing to wait for an acknowledgment message from P_j.

  - When P_j receives the message, it immediately sends an acknowledgment to P_i.

  - If P_i receives the acknowledgment message within the specified time interval, it concludes that P_j has received its message. If a time-out occurs, P_i needs to retransmit its message and wait for an acknowledgment.

  - Continue procedure until P_i either receives an acknowledgment, or is notified by the system that *B* is down.

- Suppose that P_j also needs to know that P_i has received its acknowledgment message, in order to decide on how to proceed.
  - In the presence of failure, it is not possible to accomplish this task.
  - It is not possible in a distributed environment for processes P_i and P_j to agree completely on their respective states.

Faulty processes (*Byzantine generals problem*)

- Communication medium is reliable, but processes can fail in unpredictable ways.
- Consider a system of n processes, of which no more than m are faulty. Suppose that each process P_i has some private value of V_i.
- Devise an algorithm that allows each nonfaulty P_i to construct a vector $X_i = (A_{i,1}, A_{i,2}, ..., A_{i,n})$ such that:
  1. If P_j is a nonfaulty process, then $A_{i,j} = V_j$.
  2. If P_i and P_j are both nonfaulty processes, then $X_i = X_j$.
- Solutions share the following properties.
  - A correct algorithm can be devised only if $n \geq 3 \times m + 1$.
  - The worst-case delay for reaching agreement is proportionate to $m + 1$ message-passing delays.

- An algorithm for the case where $m = 1$ and $n = 4$ requires two rounds of information exchange:
  1. Each process sends its private value to the other three processes.
  2. Each process sends the information it has obtained in the first round to all other processes.

- If a faulty process refuses to send messages, a nonfaulty process can choose an arbitrary value and pretend that that value was sent by that process.

- After the two rounds are completed, a nonfaulty process P_i can construct its vector $X_i = (A_{i,1}, A_{i,2}, A_{i,3}, A_{i,4})$ as follows:
  1. $A_{i,i} = V_i$.
  2. For $j \neq i$, if at least two of the three values reported for process P_j agree, then the majority value is used to set the value of $A_{i,j}$. Otherwise, a default value (*nil*) is used.