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.

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

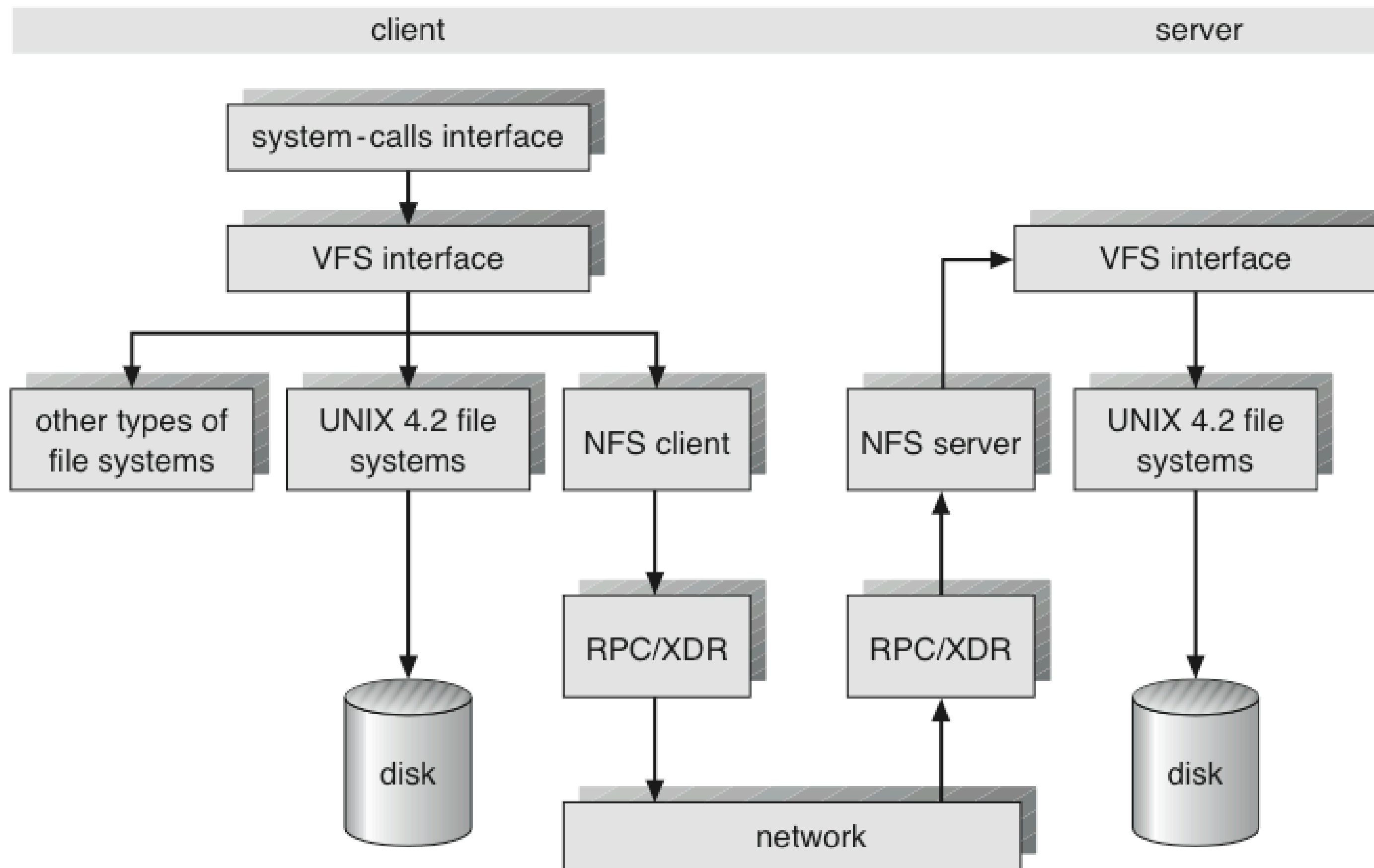
- UNIX file-system interface (based on the open, read, write, and close calls, and file descriptors).

- *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.

Distributed Coordination

- Event Ordering
- Mutual Exclusion
- Atomicity
- Deadlock Handling
- Election Algorithms

Event Ordering

- *Happened-before* relation (denoted by $\rightarrow$).
  - If A and B are events in the same process, and A was executed before B, then $A \rightarrow B$.
  - 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 $A \rightarrow B$.
  - 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 (DME)

- 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.

DME: 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

DME: 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.

DME: Fully Distributed Approach (Cont.)

- The decision whether process P_j replies immediately to a $request(P_i,\ TS)$ message or defers its reply is based on three factors:
  - If P_j is in its critical section, then it defers its reply to P_i.
  - If P_j does not want to enter its critical section, then it sends a $reply$ immediately to P_i.
  - 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.

Desirable Behavior of Fully Distributed Approach

- 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

$$\boxed{2 \times (n-1).}$$

  This is the minimum number of required messages per critical-section entry when processes act independently and concurrently.

Three Undesirable Consequences

- 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.
- 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.
- 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.
- Example: 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.)

- 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 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 locally.

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).

Failure Handling in 2PC – 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.

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 graph is the union of all 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.

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.
- 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.

Ring Algorithm (Cont.)

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

1. 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 $elect(j)$.
2. If $i \neq j$, then P_i adds j to its active list and forwards the message to its right neighbor.
3. 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.

Protection

- Goals of Protection
- Domain of Protection
- Access Matrix
- Implementation of Access Matrix
- Revocation of Access Rights
- Capability-Based Systems
- Language-Based Protection

Protection

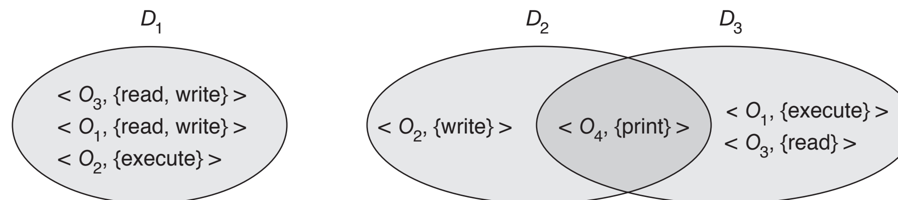
- Operating system consists of a collection of objects, hardware or software.
- Each object has a unique name and can be accessed through a well-defined set of operations.
- Protection problem – ensure that each object is accessed correctly and only by those processes that are allowed to do so.

Domain Structure

- Access-right = <object-name, rights-set>

  Rights-set is a subset of all valid operations that can be performed on the object.

- Domain = set of access-rights



Access Matrix

- Rows – domains
- Columns – domains + objects
- Each entry – Access rights

Operator names

	object $\rightarrow$			
domain $\downarrow$	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

Use of Access Matrix

- If a process in Domain D_i tries to do "op" on object O_j, then "op" must be in the access matrix.

- Can be expanded to dynamic protection.
  - Operations to add, delete access rights.
  - Special access rights:
    * $owner$ of O_i
    * $copy$ op from O_i to O_j
    * $control$ – D_i can modify D_js access rights
    * $transfer$ – switch from domain D_i to D_j

Use of Access Matrix (Cont.)

- Access matrix design separates mechanism from policy.
  - Mechanism
    * Operating system provides Access-matrix + rules.
    * It ensures that the matrix is only manipulated by authorized agents and that rules are strictly enforced.
  - Policy
    * User dictates policy.
    * Who can access what object and in what mode.

Implementation of Access Matrix

- Each column = Access-control list for one object
  Defines who can perform what operation.

  Domain 1 = Read,Write
  Domain 2 = Read
  Domain 3 = Read
  ⋮

- Each Row = Capability List (like a key)
  For each domain, what operations allowed on what objects.

  Object 1 – Read
  Object 4 – Read,Write,Execute
  Object 5 – Read,Write,Delete,Copy

Revocation of Access Rights

- Access List – Delete access rights from access list.
  - Simple
  - Immediate

- Capability List – Scheme required to locate capability in the system before capability can be revoked.
  - Reacquisition
  - Back-pointers
  - Indirection
  - Keys

Security

- The Security Problem
- Authentication
- Program Threats
- System Threats
- Threat Monitoring
- Encryption

The Security Problem

- Security must consider external environment of the system, and protect it from:
  - unauthorized access.
  - malicious modification or destruction.
  - accidental introduction of inconsistency.

- Easier to protect against accidental than malicious misuse.

Authentication

- User identity most often established through *passwords*, can be considered a special case of either keys or capabilities.

- Passwords must be kept secret.

  – Frequent change of passwords.
  – Use of "non-guessable" passwords.
  – Log all invalid access attempts.

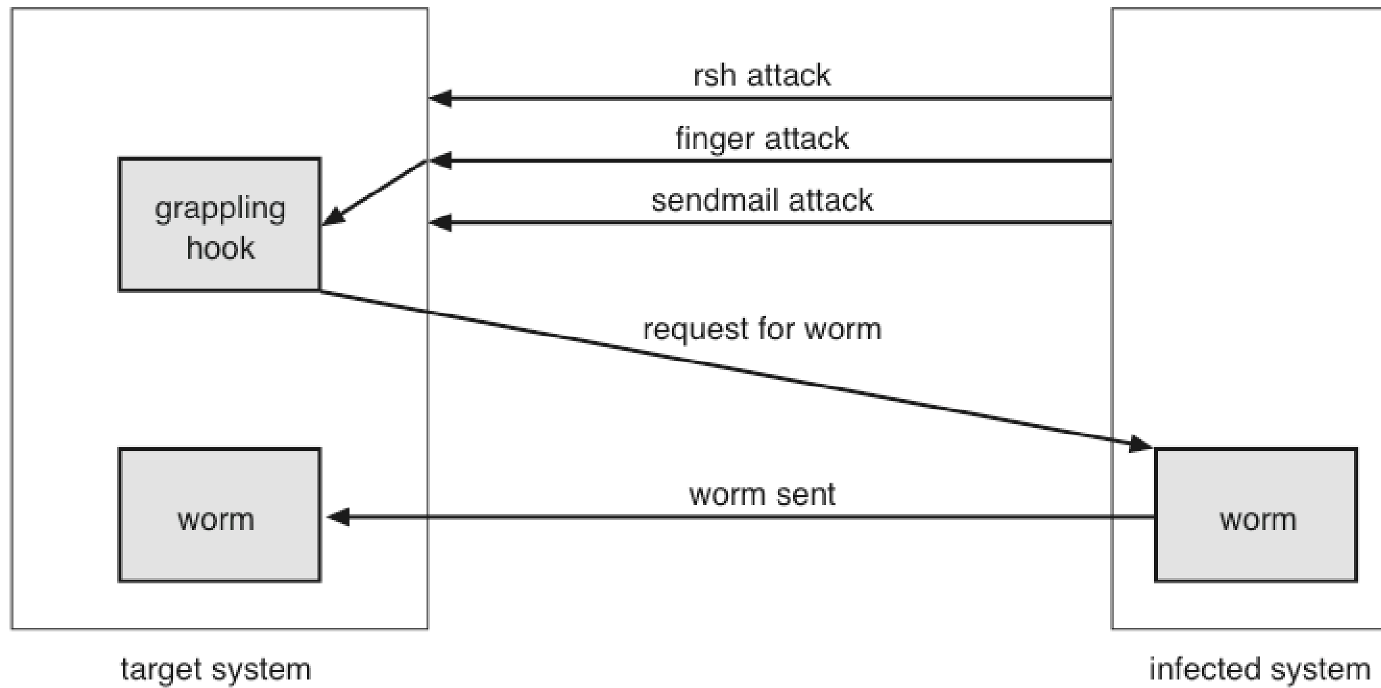
Program Threats

- Trojan Horse
  - Code segment that misuses its environment.
  - Exploits mechanisms for allowing programs written by users to be executed by other users.
- Trap Door
  - Specific user identifier or password that circumvents normal security procedures.
  - Could be included in a compiler.

System Threats

- Worms – use spawn mechanism; standalone program.
- Internet worm
  - Exploited UNIX networking features (remote access) and bugs in *finger* and *sendmail* programs. (*buffer overflows* non controllati, dabbenaggine . . .)
  - Grappling hook program uploaded main worm program.
- Viruses – fragment of code embedded in a legitimate program.
  - Mainly effect microcomputer systems.
  - Downloading viral programs from public bulletin boards or exchanging floppy disks containing an infection.
  - *Safe computing.*

Un worm famoso di R. Morris, 1988



Threat Monitoring

- Check for suspicious patterns of activity – i.e., several incorrect password attempts may signal password guessing.

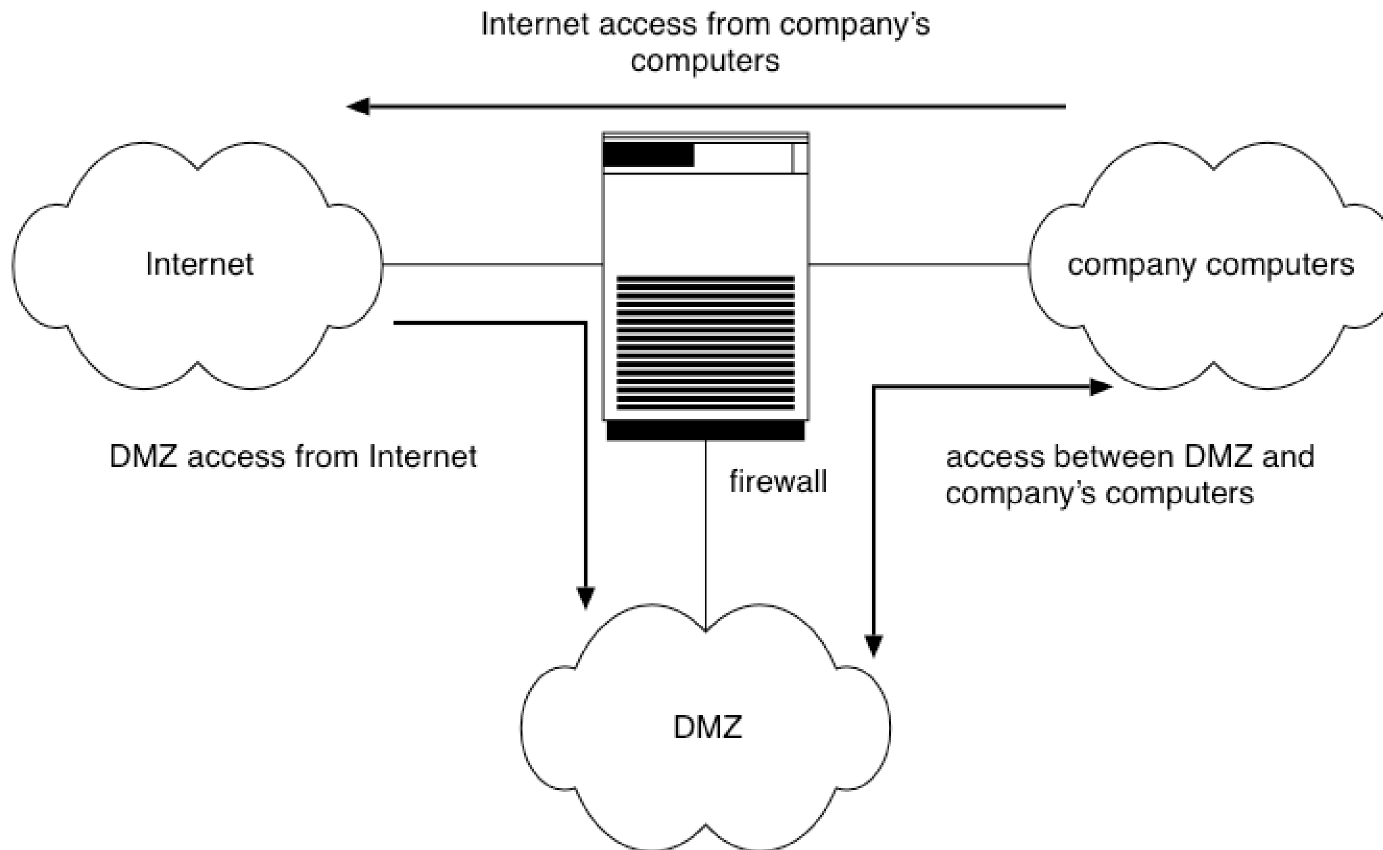
- Audit log – records the time, user, and type of all accesses to an object; useful for recovery from a violation and developing better security measures.

- Scan the system periodically for security holes; done when the computer is relatively unused.

Threat Monitoring (Cont.)

- Check for:
  - Short or easy-to-guess passwords
  - Unauthorized set-uid programs
  - Unauthorized programs in system directories
  - Unexpected long-running processes
  - Improper directory protections
  - Improper protections on system data files
  - Dangerous entries in the program search path (Trojan horse)
  - Changes to system programs; monitor checksum values

Firewalls e Zone Smilitarizzate



Encryption

- Encrypt *clear text* into *cipher text.*

- Properties of good encryption technique:
  - Relatively simple for authorized users to encrypt and decrypt data.
  - Encryption scheme depends not on the secrecy of the algorithm but on a parameter of the algorithm called the *encryption key*.
  - Extremely difficult for an intruder to determine the encryption key.

- *Data Encryption Standard* substitutes characters and rearranges their order on the basis of an encryption key provided to authorized users via a secure mechanism. Scheme only as secure as the mechanism.

Encryption (cont.)

- *Public-key encryption* based on each user having two keys:
  - *public key* – published key used to encrypt data.
  - *private key* – key known only to individual user used to decrypt data.

- Must be an encryption scheme that can be made public without making it easy to figure out the decryption scheme.
  - Efficient algorithm for testing whether or not a number is prime.
  - No efficient algorithm is known for finding the prime factors of a number. (esiste se $P \neq NP$)

Encryption (cont.)

- public key $= (e, n)$
- private key $= (d, n)$
- $n := pq$ con p, q primi
- d deve essere preso coprimo con $(p-1)(q-1)$
- $e := d^{-1}$ in $\mathbb{Z}_{(p-1)(q-1)}$
- $E(m) := m^e \, mod \, n$
- $D(c) := c^d \, mod \, n$
- $D \circ E = id$