Cloud computing and distributed systems

Distributed file systems

Zeynep Yücel

Ca' Foscari University of Venice
zeynep.yucel@unive.it
yucelzeynep.github.io

Introduction

- Resource sharing goals.
- Data sharing forms:
  - Web servers: limited sharing.
  - Large-scale challenges: balancing, availability.
  - Peer-to-peer solutions.
  - Replicated storage: reliability.
- Basic distributed systems:
  - Mimics non-distributed systems.
  - Remote client programs.
  - Properties:
    - No multiple file copies.
    - No bandwidth guarantees.
    - Foundational for intranet computing.

Introduction

- Remote files accessed as if local.
- Emulates non-distributed functionalities.
- Local networks require persistent storage.

Introduction

History and evolution

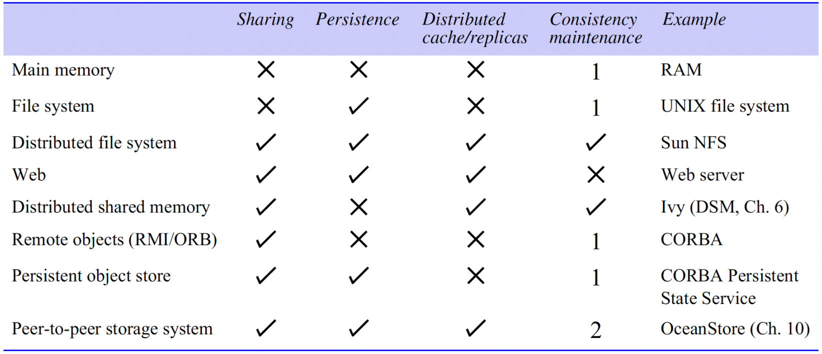
- Centralized system and desktop computer storage management.
- Evolved features (e.g. access control)
- Distributed systems share file and hardware resources.
- Good file service: performance, reliability comparable to local disk.
- Optimized for local networks.
- Effective shared storage in intranets.

Introduction

- File service enables remote storage and access.
    - Centralized storage reduces local space.
    - Web servers utilize distributed systems.
- Demand for persistent storage.
    - Serialization impractical for changes.
    - Some technologies lack persistence, replication.

Introduction

Overview of storage system types



	Sharing	*Persistence*	*Distributed cache/replicas*	*Consistency maintenance*	*Example*
Main memory	✗	✗	✗	1	RAM
File system	✗	✓	✗	1	UNIX file system
Distributed file system	✓	✓	✓	✓	Sun NFS
Web	✓	✓	✓	✗	Web server
Distributed shared memory	✓	✗	✓	✓	Ivy (DSM, Ch. 6)
Remote objects (RMI/ORB)	✓	✗	✗	1	CORBA
Persistent object store	✓	✓	✗	1	CORBA Persistent State Service
Peer-to-peer storage system	✓	✓	✓	2	OceanStore (Ch. 10)

Types of consistency:

1: strict one-copy ✓: slightly weaker guarantees 2: considerably weaker guarantees

Introduction

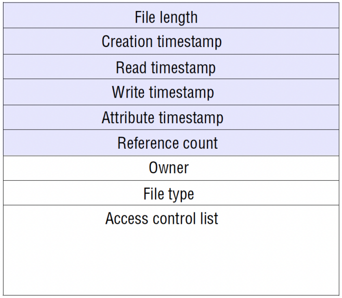
Overview of storage system types - Discussion of consistency and caching

- Consistency column shows data consistency maintenance.
- Non-distributed systems: strict consistency.
- Distributed systems: strict consistency is harder.
- Sun NFS: near strict consistency.
- Web caching: manual update checks.
- Persistent systems: different methods.

Introduction

Characteristics of file systems

- File systems: organize, store, retrieve etc. files.
- Programming interface for storage management.
- Data and attributes.
  - Data: sequence of bytes.
  - Attributes: metadata details (e.g. owner).



File length
Creation timestamp
Read timestamp
Write timestamp
Attribute timestamp
Reference count
Owner
File type
Access control list

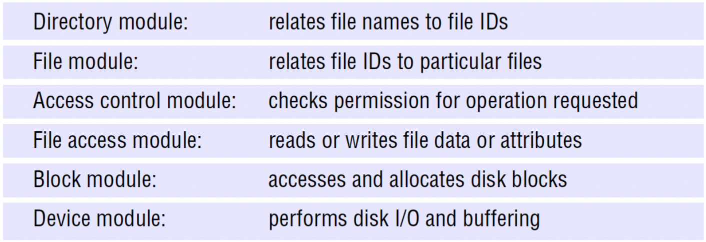
Introduction

Characteristics of file systems - Directories

- File systems: create, name, etc. files with directory organization.
  - Directory: maps names to identifiers.
  - Directories: hierarchical structure.
- File systems: control access based on authorization.
- Metadata: additional file management info.

Introduction

Characteristics of file systems - Directories



Directory module:	relates file names to file IDs
File module:	relates file IDs to particular files
Access control module:	checks permission for operation requested
File access module:	reads or writes file data or attributes
Block module:	accesses and allocates disk blocks
Device module:	performs disk I/O and buffering

- Non-distributed: layered structure.
- Distributed systems: require components for
  - Client-server communication
  - File location

Introduction

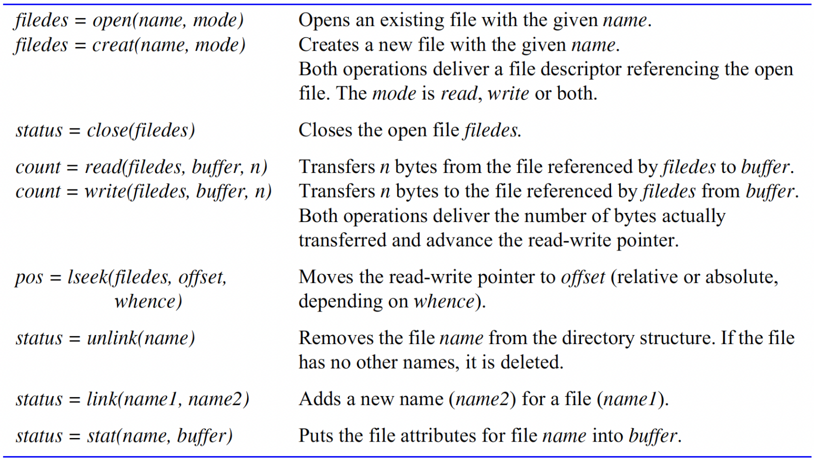
File system operations - UNIX file system operations

- UNIX systems provide main file operations.
- Programmers access access system calls via libraries.
- File system: manages access control.
- Matching rights: open file access.

Introduction

File system operations - UNIX file system operations

Figure 12.4 UNIX file system operations



filedes = open(name, mode)	Opens an existing file with the given *name*.
filedes = creat(name, mode)	Creates a new file with the given *name*. Both operations deliver a file descriptor referencing the open file. The *mode* is *read*, *write* or both.
status = close(filedes)	Closes the open file *filedes*.
count = read(filedes, buffer, n)	Transfers *n* bytes from the file referenced by *filedes* to *buffer*.
count = write(filedes, buffer, n)	Transfers *n* bytes to the file referenced by *filedes* from *buffer*. Both operations deliver the number of bytes actually transferred and advance the read-write pointer.
pos = lseek(filedes, offset, whence)	Moves the read-write pointer to *offset* (relative or absolute, depending on *whence*).
status = unlink(name)	Removes the file *name* from the directory structure. If the file has no other names, it is deleted.
status = link(name1, name2)	Adds a new name (*name2*) for a file (*name1*).
status = stat(name, buffer)	Puts the file attributes for file *name* into *buffer*.

Introduction

Distributed file system requirements and Transparency

- File service design: supports transparency.
  - Access Transparency: clients access local and remote files seamlessly.
  - Location Transparency: uniform file name space.
  - Mobility Transparency: files can be moved without changes on client applications.
  - Performance Transparency: no significant degradation with increasing load.
  - Scaling Transparency: allows incremental growth.

Introduction

Concurrent file updates and File replication

- Concurrent File Updates:
  - Concurrency control: no disruption between clients.
  - Widely needed, but costly.
- File Replication:
  - Multiple copies: load sharing, fault tolerance.
  - Few services offer full replication support.

Introduction

Hardware and operating system heterogeneity, Fault tolerance

- Hardware, OS Compatibility
- Fault Tolerance: operational despite failures.
  - Stateless servers: restart without restore.
  - Replication: complex fault tolerance.

Introduction

Consistency, Security

- Consistency: One-copy update.
  - All processes uniform.
  - Update delays issues.
- Security: Access control lists.
  - Client request authentication.
  - Protection via encryption.
- Efficiency: Match conventional performance.

Introduction

Case studies, File service architecture

- Abstract model and case study of Sun NFS.
- Three modules: client, server, files.
  - Client mimics traditional system.
  - Server handles file operations.
  - File module storage.
- Stateless server implementation.

Introduction

SUN NSF

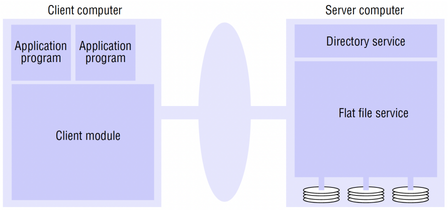
- Introduced in 1985, widely adopted.
- First product-designed file service.
- Any computer can serve.
  - Common practice: dedicated servers.
- Supports multiple hardware and OS.

Introduction

Andrew File System (AFS)

- Developed at CMU.
- Distributed computing environment.
- Minimizes communication with file caching.

File service architecture



- Abstract model with a clear separation, three components.
  - Flat file service.
  - Directory service.
  - Client module.
- Provide interfaces for clients.
  - Comprehensive file access.
- Unified programming interface.
- Supports diverse hardware setups.

File service architecture

Flat file service and Directory service

- Flat File Service:
  - Manages file operations.
  - Uses Unique File Identifiers (UFIDs).
  - Generates new UFID at creation.
- Directory Service:
  - Maps names to UFIDs.
  - Clients retrieve UFIDs via names.
  - Supports directory management.
  - Stores in flat file service.
  - Manages hierarchic naming schemes.

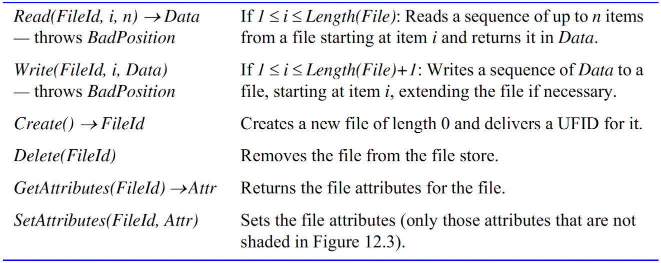
File service architecture

Client module

- Operates on each client.
- Access user-level file operations.
  - Emulates UNIX file operations.
- Stores network location info.
- Improves performance via caching.

File service architecture

Flat file service interface

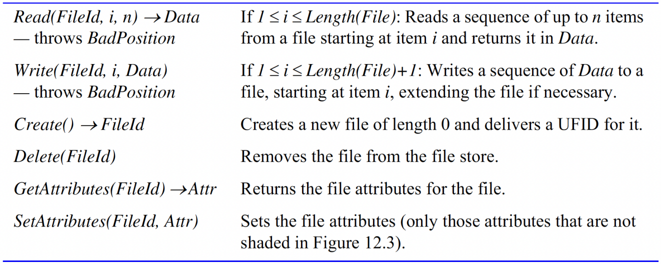


Read(FileId, i, n) $\rightarrow$ *Data* — throws *BadPosition*	If $1 \le i \le Length(File)$: Reads a sequence of up to *n* items from a file starting at item *i* and returns it in *Data*.
Write(FileId, i, Data) — throws *BadPosition*	If $1 \le i \le Length(File)+1$: Writes a sequence of *Data* to a file, starting at item *i*, extending the file if necessary.
Create() $\rightarrow$ *FileId*	Creates a new file of length 0 and delivers a UFID for it.
Delete(FileId)	Removes the file from the file store.
GetAttributes(FileId) $\rightarrow$ *Attr*	Returns the file attributes for the file.
SetAttributes(FileId, Attr)	Sets the file attributes (only those attributes that are not shaded in Figure 12.3).

- Sample flat file service interface.
- Invalid FileId, if file is missing or access permission is not not sufficient.
- Most procedures throw exceptions for invalid FileId.

File service architecture

Flat file service interface - Key operations include:



Read(FileId, i, n) $\rightarrow$ *Data* — throws *BadPosition*	If $1 \leq i \leq Length(File)$: Reads a sequence of up to n items from a file starting at item i and returns it in *Data*.
Write(FileId, i, Data) — throws *BadPosition*	If $1 \leq i \leq Length(File)+1$: Writes a sequence of *Data* to a file, starting at item i, extending the file if necessary.
Create() $\rightarrow$ *FileId*	Creates a new file of length 0 and delivers a UFID for it.
Delete(FileId)	Removes the file from the file store.
GetAttributes(FileId) $\rightarrow$ *Attr*	Returns the file attributes for the file.
SetAttributes(FileId, Attr)	Sets the file attributes (only those attributes that are not shaded in Figure 12.3).

- Read: Copies data items.
- Write: Replaces existing content.
- Create: Generates new file.
- Delete: Removes specified file.
- GetAttributes: Access file attributes.
- SetAttributes: Modify file attributes.

File service architecture

Comparison with UNIX

- Interface vs. UNIX.
- Functionally equivalent, but
  - No Open/Close operations.
  - Read/Write starting point.
  - UNIX uses read-write pointer.
- Designed for stateless servers.
  - All operations idempotent.
  - UNIX not idempotent.
  - Eliminating pointer reduces state.

File service architecture

Access control

- UNIX access rights checked during opening.
- Non-distributed case: verify UID at login.
  - Rights retained until file isclosed.
- Distributed case:check at server.
  - Access checks not stateless.
  - Two access check methods:
    - Access check during name conversion.
    - Send user identity with every request.
  - Both methods allow stateless implementation.
  - Neither method prevents forging.

File service architecture

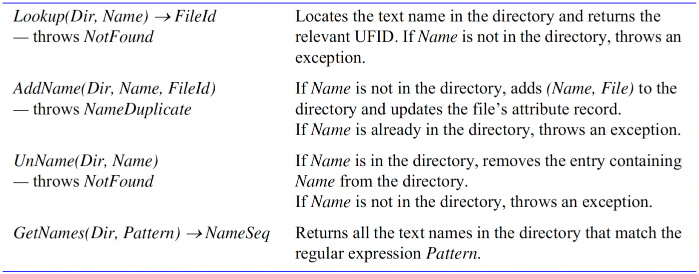
Directory service interface

- Directory service translates names to UFIDs.
  - Maintains directory mappings.
  - Each directory is a conventional file.

File service architecture

Directory service interface, Operations

Figure 12.7 Directory service operations



Lookup(Dir, Name) → FileId — throws *NotFound*	Locates the text name in the directory and returns the relevant UFID. If *Name* is not in the directory, throws an exception.
AddName(Dir, Name, FileId) — throws *NameDuplicate*	If *Name* is not in the directory, adds *(Name, File)* to the directory and updates the file's attribute record. If *Name* is already in the directory, throws an exception.
UnName(Dir, Name) — throws *NotFound*	If *Name* is in the directory, removes the entry containing *Name* from the directory. If *Name* is not in the directory, throws an exception.
GetNames(Dir, Pattern) → NameSeq	Returns all the text names in the directory that match the regular expression *Pattern*.

- Define operations on directories.
- Lookup translates Name to UFID.
- Two operations for altering directories:
  - AddName adds directory entry.
  - UnName removes directory entry.
- GetNames views directory contents.

File service architecture

Hierarchic file system

- Hierarchic file system in a tree structure.
- Each directory contains files and other directories.
- Accessed via pathname.
- Tree structure with directories and files.
- Root directory identified by a well-known UFID.
- Supports multiple names for files.
- File attributes indicate type.

File service architecture

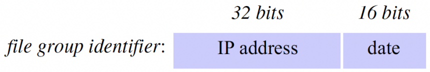
File groups

- File Groups: Collections of files on a server.
- Multiple Groups: Server can hold several groups.
  - Groups movable between servers.
  - Files not movable between groups.
- Filesystem similar to UNIX.
  - "Filesystem" = files on storage.
  - "file system" = access software.
- Distributed File Service: Organizes files on servers.

File service architecture

File groups

- Unique Identifiers: Each group has unique ID.
- Identifier Generation: Distinct identifiers.
  E.g. unique IDs by combining IP address and date integer.



  - Groups can be moved; IPs change.
  - Maintain mapping of identifiers.

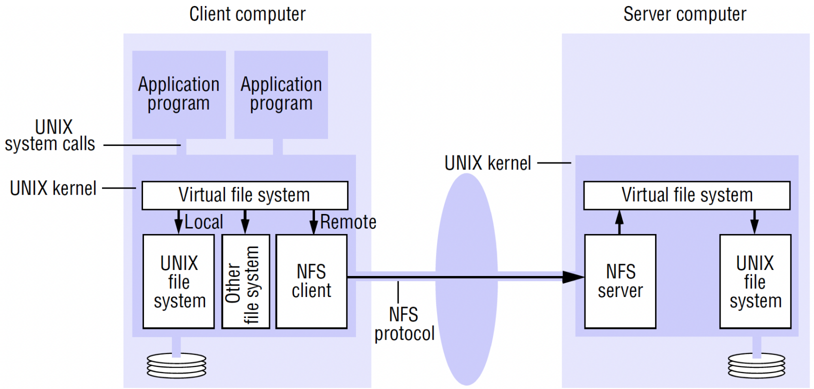
Case study: Sun Network File System

- Sun NFS architecture.
- Client/server communication with RPCs.
  - Supports NFS protocol.
  - Protocol is OS-independent.
- NFS server module in kernel.
- Client requests transformed to NFS operations.
- RPC uses UDP/TCP.

Case study: Sun Network File System

Virtual file system

- NFS provides access transparency: local/remote files without distinction.



Case study: Sun Network File System

Virtual file system

- VFS integrated into UNIX kernel.
  - Mediates between user applications and file systems.
- Files can be local/remote.
  - VFS identifies file location.
- NFS uses own identifiers.
  - VFS translates identifiers.
- Tracks local/remote filesystems.

Case study: Sun Network File System

Virtual file system

- NFS uses file handles.
- File handles derived from i-node number.
  - i-node number identifies file.
- File handle includes filesystem identifier, i-node number, and generation number:

File handle:	Filesystem identifier	i-node number of file	i-node generation number

  - Filesystem identifier is unique.
  - i-node generation number prevents confusion in case of reuse.
    - Tracks reused i-node numbers.

Case study: Sun Network File System

Client integration

- NFS client module: client interface.
- Mimics UNIX functions.
  - Programs access files via UNIX calls.
  - Single client module for all processes.
  - Encryption key in kernel for security.

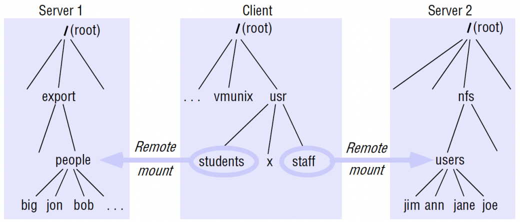
Case study: Sun Network File System

Access control and authentication, NFS server interface

- Access control and authentication
  - Stateless servers
    - Identity check required at every request.
    - Client authentication info in requests.
- NFS server interface
  - RPC-based.

Case study: Sun Network File System

Mount service



Note: The file system mounted at */usr/students* in the client is actually the subtree located at */export/people* in Server 1; the filesystem mounted at */usr/staff* in the client is actually the subtree located at /nfs/users in Server 2.

- Mount service for filesystems.
- File listing available for access.
- Access list for hosts.
- Client requests remote mounts.
- Remote directory can be any subtree.

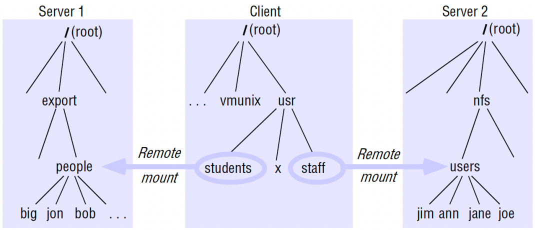
Case study: Sun Network File System

Mount service

- Mount protocol for communication.
- Client receives file handle.
- Filesystems: hard/soft-mounted.
  - Hard-mounted: processes suspend.
  - Soft-mounted: returns failure.
- Prefer hard-mounting.

Case study: Sun Network File System

Pathname translation



- UNIX translates pathnames to i-node references.
- Iterative parsing by client.
- Multipart path home/user/student/bob.txt:
  - Start at root.
  - Lookup each part.
  - Continue to final file.

Case study: Sun Network File System

Pathname translation

- Different parts can be on different servers.
- Separate lookup requests needed for each part.
- Each lookup retrieves a file handle.
- Handles used in the next lookup.
- Caching improves efficiency.

Case study: Sun Network File System

Automounter

- Dynamically mount remote directories.
  - Keeps a table of mount points.
  - Acts like a local NFS server on client.
  - NFS sends client request to automounter.
  - Automounter look up filesystem and probes servers.
  - First responding server mounts filesystem.
  - Symbolic link for future access.

Case study: Sun Network File System

Server caching

- Caching enhances performance at both server and client.
- Buffer Cache: File pages in memory.
- No disk access for cached pages.
- Read and Write Optimization:
  - Read-ahead reduces disk access.
  - Delayed-write saves changes.
    - UNIX sync flushes every 30 seconds.

Case study: Sun Network File System

Client caching

- Client caches results of operations.
- Possibly outdated files.
- Clients are responsible for polling the server to check.
- Caches are validated with timestamps:
  - Tc: Last validation time.
  - Tm: Last modification time.
- Validity Condition:
  - $T - Tc < t$ (freshness interval t), or
  - $Tm_{client} = Tm_{server}$ (no server modification).
- t balances consistency and efficiency.
- Reduces server calls.

Discussion topic

Why is there no open or close operation in the interface to the flat file service or the directory service. What are the differences between our directory service Lookup operation and the UNIX open?

Discussion topic

How many lookup calls are needed to resolve a 5-part pathname (for example, /usr/users/jim/code/ xyz.c) for a file that is stored on an NFS server? What is the reason for performing the translation step-by-step?