File Systems

- **file system:** provides
    - convenient programming interface for disk storage
    - access control
    - file-locking (for file sharing)
- **distributed file system:** emulates non-distributed file system for client programs running on multiple remote computers
- **file service:** allow programs to store and access remote files as they do local ones
    - access files from any computer on intranet
    - hosts providing such services can be optimised for multiple disk drives, and can supply file services for other services (web, email)
    - facilitates backup and archiving
- **files:** data + attributes
- **directory:** file containing list of other files

File System Layers

- **Directory:** relate file names to IDs
- **File:** relate file IDs to particular files
- **Access control:** check permissions for requested operations
- **File access:** read/write file data/attributes
- **Block:** access/allocate disk blocks
- **Device:** disk IO and buffering

UNIX file system operations

- `open`
- `create`
- `close`
- `read`
- `write`
- `lseek`: move read/write pointer to new position in the file
- `link`: add new name for file
- `stat`: get file attributes

Distributed File System Requirements

- e.g. Hadoop

Transparency

- **Access:** clients unaware of distribution of files
  - uniform API for accessing local and remote files
- **Location:** clients see a uniform file name space
  - names of files should be consistent regardless of where the file is physically stored
- **Mobility:** client programs/admin services don't need to change when the files are physically moved
- **Performance:** client programs should perform satisfactorily while the load varies in specified range
- **Scaling:** service can be expanded by incremental growth

Concurrent file updates

- multiple clients' updates should not interfere with each other
- should be able to manage policies

File replication

- multiple copies of files over several servers: better capacity for accessing the file, better fault tolerance

Heterogeneity

- client and server should be able to operate on various hardware/OS

Fault tolerance

- transient communication problems shouldn't result in file corruption
- invocation semantics: can be
  - at-most-once

    - at-least-once: simpler, but requires idempotent operations

- servers can be **stateless** such that there is no recovery required if a server goes down

Consistency

- multiple, concurrent access to file should see consistent representation of the file
- file metadata should be consistent on all clients

Security

- client requests should be authenticated
- data transfer should be encrypted

Efficiency

- comparable to conventional file systems

File Service Architecture

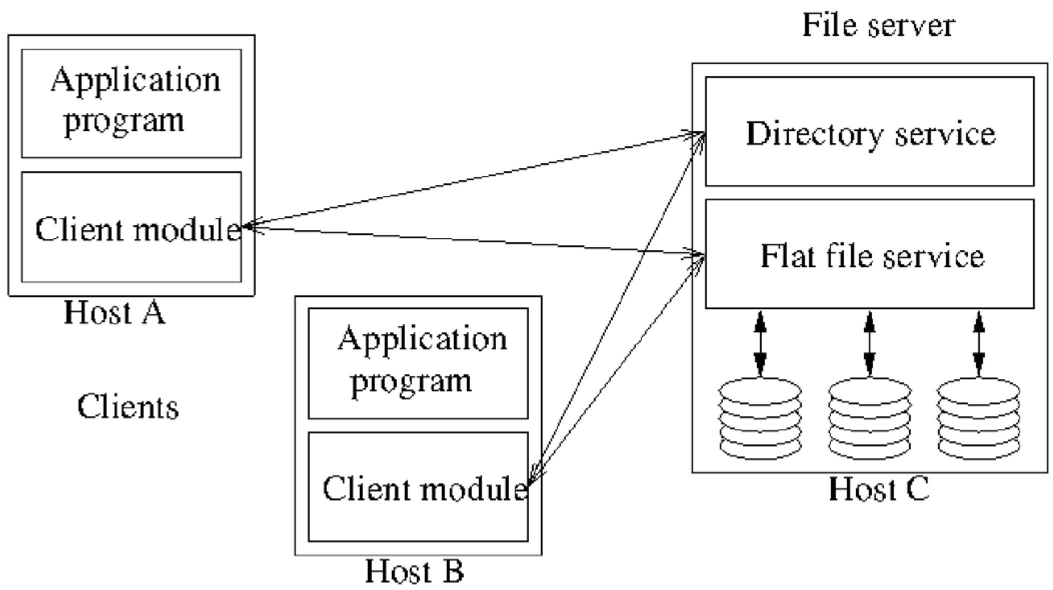

Figure 1: File system architecture

- abstract architecture based on NFS

Flat file service

- implements operations on contents of files
- **UFID, Unique File Identifier** given to flat file service to refer to the file to operate on

Directory service

- mapping between text file names and UFID
- creates directories, and manages files within them
- client of flat file service, as directory files are stored there

Client module

- integrates directory service and flat file service to provide API expected by client applications
- client maintains a list of available file servers
- caching to improve performance

Flat file service interface

- UNIX interface requires the filesystem to maintain state (in the **file pointer**), which is manipulated during read/write
- flat file service differs from UNIX interface for **fault tolerance**
  - **repeatable operations:** except for `Create`, operations are idempotent, permitting at-least-once RPC semantics
  - **stateless server:** flat file service doesn't need to maintain state. Can be restarted after failure and resume operation without need for clients/server to restore any state
  - files can be accessed immediately, c.f. UNIX where they first need to be opened

RPC Calls

- `Read`
- `Write`
- `Create`
- `Delete`

- `GetAttributes`
- `SetAttributes`

Flat file service access control

- authenticate RPC caller
- prevent illegal operations: e.g. legal UFIDs, enforce access privileges
- cannot store access control state: would break idempotency
- options:
  - access check made whenever file name is converted to UFID, and results encoded as a capability returned to client for submission to flat file server
  - user ID can be submitted for every request, with access checks performed by flat file server for each file operation

Directory service interface

- translation from file name to UFID
- abstract directory service interface
- 0 TODO: <22-10-20, yourname> 0

File Group

- collection of files on a given server
- server may hold several file groups, and file groups can be moved between servers
- files cannot change file group
- permits file service to be implemented across several servers
- files given UFIDs that ensure uniqueness across different servers
  - e.g. concatenate server IP address with a date the file was created
  - permits files in a group (i.e. files with common *file group id*) to be relocated to a different server without conflicting with files already on the server
- mapping of UFIDs to servers can be cached at client module

Sun Network File System

- uses architecture described above

- many implementations of NFS following NFS protocols, using a set of RPCs that provide means for the client to perform operations on the remote file store
- NFS client makes requests to NFS server to access files

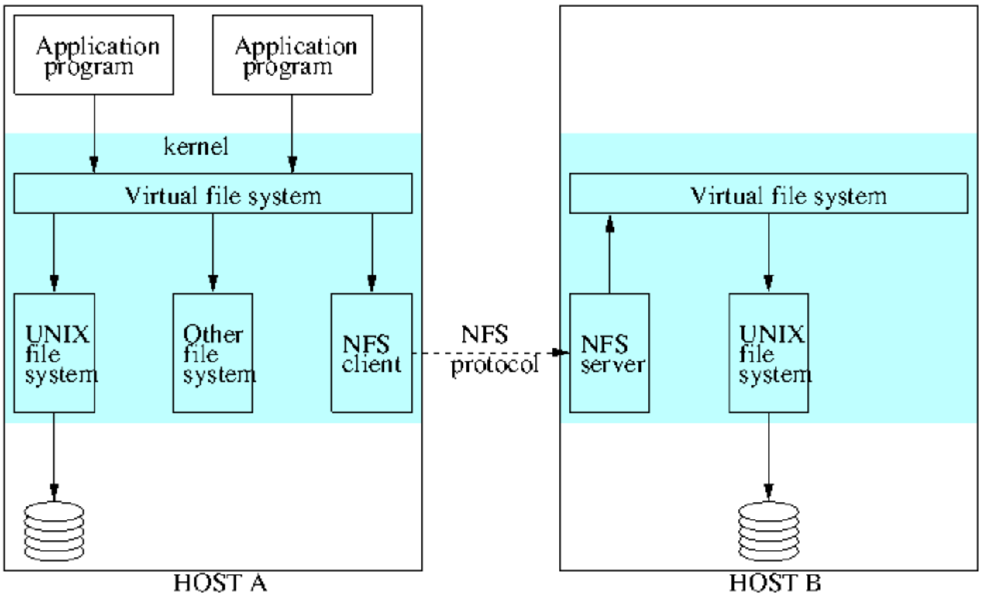

Figure 2: Sun NFS

Virtual File System

- VFS used by UNIX to provide transparent access to any number of different file systems, combining remote and local file systems into a single filesystem
  - maintains VFS structure for each filesystem in use
  - maintains **v-node** for each open file, which records whether file is local/remote
    * if local, v-node contains reference to `i-node` on UNIX file system
    * if remote, v-node contains reference to files NFS **file handle**, a combo of **filesystem identifier, i-node number** and any other identifying info
- NFS integrated in the same way

Client Integration

Server Interface

Mount Service

-

Server caching

- conventional UNIX systems: data read from disk/pages are retained in main memory buffer cache, and evicted when buffer space is needed. Accesses to the cache do not require disk access
  - **read-ahead:** anticipates read accesses, fetches pages following those recently read
  - **delayed-write/write-back:** optimises writes to disk by only writing pages when both modified and evicted
    * UNIX `sync` flushes modified pages every 30s
  - works for conventional filesystem on single host, because there is only 1 cache and file accesses cannot bypass it
- use of cache at server for client reads introduces no problems
- use of cache for writes requires special care: client needs to be confident writes are persistent if server crashes
- options: cache policies used by the server
  - **Write-through**: data written to cache and directly to disk
    * increases disk I/O and latency for write
    * operation completes when the data has been written to disk
    * poor when server receives large number of write requests for the same data
    * saves network bandwidth
  - **Commit**: data is written to cache and is written to disk when a commit operation is received for the data
    * reply sent when data has been written to disk
    * uses more network bandwidth
    * may lead to uncommitted data being lost
    * receives full benefit of cache

Client Caching

- NFS Client caches data reads, writes, attributes and directory operations to reduce network IO
- caching at the client: problem for cache consistency, as different caches on multiple clients, and the server
- reading and writing are both problems: a `write` on another client between two `read`s will lead to the second `read` being incorrect
- NFS clients poll the server for updates
- T_c: time when a cache block was last validated by the client
- T_m: time when a block was last modified
- cache block is valid at time T if
  - $T - T_c < t$ where t is a freshness interval, or
  - $T_{m,client} = T_{m,server}$
- small value for t leads to close approximation of **one-copy consistency**, but costs greater network IO
- in Sun Solaris clients t is set adaptively (3-30s) depending on file update frequency
- validity check is made on each access to a cache block
  - first half of check requires no network IO

NFS Summary

- ✓ access transparency: applications are usually unaware files are remote
- $\times$ location transparency: not enforced; no global namespace as different clients can mount filesystems at different points
- $\times$ mobility transparency: if server changes, the client must be updated
- $\sim$ scalability: good, can be better. System can grow to accommodate more servers as needed. Bottlenecks when many processes access a single file.
- $\times$ file replication: not supported for updates. Additional services can be added to do this
- ✓ Hardware/OS heterogeneity: NFS implemented on most OS and hardware platforms
- ✓ fault tolerance: acceptable. NFS is stateless, idempotent. Options to handle failures
- ✓ consistency: tunable. not recommended for close synchronisation between processes
- ✓ security: Kerberos is integrated with NFS. Secure RPC also an option
- ✓ efficiency: acceptable, can be tuned.