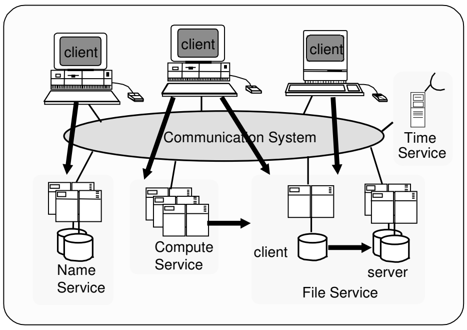
06-06798 Distributed Systems

Lecture 7: Distributed File Systems

Overview

- Requirements for distributed file systems
  - transparency, performance, fault-tolerance, ...
- Design issues
  - possible options, architectures
  - file sharing, concurrent updates
  - caching
- Example
  - Sun NFS

Distributed Service



Distributed file service

- Basic service
  - persistent file storage of data and programs
  - operations on files (create, open, read,…)
  - multiple remote clients, within intranet
  - file sharing
  - typically one-copy update semantics, over RPC
- Many new developments
  - persistent object stores (storage of objects)
    - Persistent Java, Corba, …
  - replication, whole-file caching
  - distributed multimedia (Tiger video file server)

Characteristics of file systems

- Operations on files (=data + attributes)
  - create/delete
  - query/modify attributes
  - open/close
  - read/write
  - access control
- Storage organisation
  - directory structure (hierarchical, pathnames)
  - metadata (file management information)
    - file attributes
    - directory structure info, etc

File attributes

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

User controlled

Layered structure of file system

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:	disk I/O and buffering

Concentrate on higher levels.

Distributed file system requirements

- Transparency (clients unaware of the distributed nature)
  - access transparency (client unaware of distribution of files, same interface for local/remote files)
  - location transparency (uniform file name space from any client workstation)
  - mobility transparency (files can be moved from one server to another without affecting client)
  - performance transparency (client performance not affected by load on service)
  - scaling transparency (expansion possible if numbers of clients increase)

Distributed file system requirements

- Concurrent file updates (changes by one client do not affect another)
- File replication (for load sharing, fault-tolerance)
- Heterogeneity (interface platform-independent)
- Fault-tolerance (continues to operate in the face of client and server failures)
- Consistency (*one-copy-update* semantics or slight variations)
- Security (access control)
- Efficiency (performance comparable to conventional file systems)

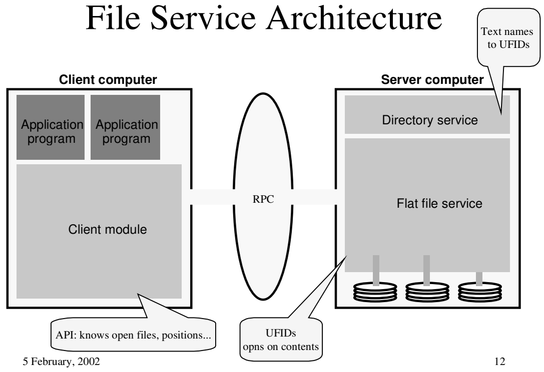
File Service Design Options

- Stateful
  - server holds information on open files, current position, file locks
  - open before access, close after
  - better performance - shorter message, read-ahead possible
  - server failure - lose state
  - client failure - tables fill up
  - can provide file locks

File Service Design Options

- Stateless
  - no state information held by server
  - file operations idempotent, must contain all information needed (longer message)
  - simpler file server design
  - can recover easily from client or server crash
  - locking requires extra lock server to hold state

File Service Architecture



File server architecture

Components (for openness):

- Flat file service
  - operations on file contents
  - unique file identifiers (UFIDs)
  - translates UFIDs to file locations
- Directory service
  - mapping between text names to UFIDs
- Client module
  - API for file access, one per client computer
  - holds state: open files, positions
  - knows network location of flat file & directory server

5 February, 2002 13

Flat file service RPC interface

- Used by client modules, not user programs
  - *FileId* (UFID) uniquely identifies file
  - invalid if file not present or inappropriate access
  - *Read/Write*; *Create/Delete*; *Get/SetAttributes*
- No open/close! (unlike UNIX)
  - access immediate with *FileId*
  - *Read/Write* identify starting point
- Improved fault-tolerance
  - operations idempotent except Create, can be repeated (at-least-once RPC semantics)
  - stateless service

5 February, 2002 14

Flat file service operations

Read(FileId, i, n) -> 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() -> 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) -> Attr	Returns the file attributes for the file.
SetAttributes(FileId, Attr)	Sets the file attributes (only owner, file type and access control list; rest maintained by flat file service).

5 February, 2002 15

Access control

- In UNIX file system
  - access rights are checked against the access mode (read, write, execute) in open
  - user identity checked at login time, cannot be tampered with
- In distributed systems
  - access rights must be checked at server
    - RPC unprotected
    - forging identity possible, a security risk
  - user id typically passed with every request (e.g. Sun NFS)
  - stateless

5 February, 2002 16

Directory structure

- Hierarchical
  - tree-like, pathnames from root
  - (in UNIX) several names per file (*link* operation)
- Naming system
  - implemented by client module, using directory service
  - root has well-known UFID
  - locate file following path from root

5 February, 2002 17

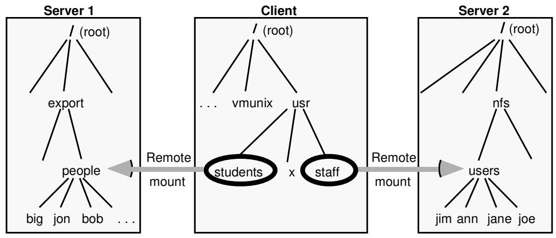
File names

Text name (=directory pathname+file name)

- hostname:local name
  - not mobility transparent
- uniform name structure (same name space for all clients)
- remote mount (e.g. Sun NFS)
  - remote directory inserted into local directory
  - relies on clients maintaining consistent naming conventions across all clients
    - all clients must implement same local tree
    - must mount remote directory into the same local directory

5 February, 2002 18

Remote mount



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

Directory service

- Directory
  - conventional file (client of the flat file service)
  - mapping from text names to UFIDs
- Operations
  - require *FileId*, machine readable UFID as parameter
  - locate file (*LookUp*)
  - add/delete file (*AddName/UnName*)
  - match file names to regular expression (*GetNames*)

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, File) — 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: the entry containing *Name* is removed 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*.

File sharing

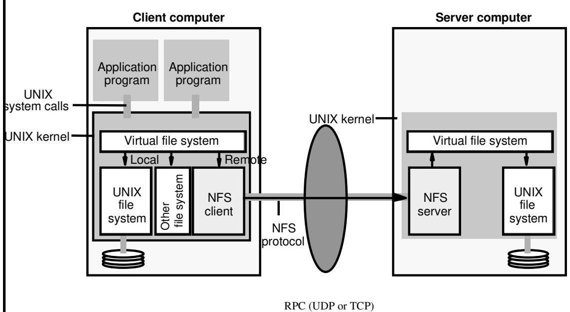
Multiple clients share the same file for read/write access.

- One-copy update semantics
  - every read sees the effect of all previous writes
  - a write is immediately visible to clients who have the file open for reading
- Problems!
  - caching: maintaining consistency between several copies difficult to achieve
  - serialise access by using file locks (affects performance)
  - trade-off between consistency and performance

Example: Sun NFS (1985)

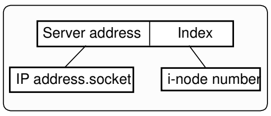
- Structure of flat file & client & directory service
- NFS protocol
  - RPC based, OS independent (originally UNIX)
- NFS server
  - stateless (no open/close)
  - no locks or concurrency control
  - no replication with updates
- Virtual file system, remote mount
- Access control (user id with each request)
  - security loophole (modify RPC to impersonate user…)
- Client and server caching

NFS architecture



File identifier (*FileId*)

Simple Solution
– i-node (number identifying file within file system)
– file migration requires finding and changing all *FileId*s
– UNIX reuses i-node numbers after file deleted (i-node gen. no)



NFS file handle
Virtual file system uses i-node if local, file handle if remote.

File handle

File system identifier	i-node no.	i-node gener. no.

5 February, 2002 25

Caching in NFS

- Indispensable for performance
- Caching
  - retains recently used data (file pages, directories, file attributes) in cache
  - updates data in cache for speed
  - block size typically 8kbytes
- Server caching
  - cache in server memory (UNIX kernel)
- Client caching
  - cache in client memory, local disk

5 February, 2002 26

Server caching

- Store data in server memory
- Read-ahead: anticipate which pages to read
- Delayed write
  - update in cache; write to disk periodically (UNIX *sync* to synchronise cache) or when space needed
  - which contents seen by users depends on timing
- Write through
  - cache and write to disk (reliable, poor performance)
- Write on close
  - write to disk only when commit received (fast but problems with files open for a long time)

5 February, 2002 27

Client caching

- Potential consistency problems!
  - different versions, portions of files, … since writes delayed
  - clients poll server to check if copy still valid
- Timestamp method
  - tag with latest time of validity check and modification time
  - copy valid if time since last check less than freshness interval, or modification time on server the same
  - choose freshness interval adaptively, 3-30 sec for files, 30-60 sec for directories
  - for small freshness interval, potential heavy load on network

5 February, 2002 28

Client caching

- Reads
  - perform validity check whenever cache entry used
  - if not valid, request data from server
  - several optimisations to reduce traffic
  - recent updates not always visible (timing!)
- Writes
  - when page modified, marked as dirty
  - dirty pages flushed asynchronously, periodically (client's synch) and on close
- Not truly *one-copy update* semantics...

5 February, 2002 29

New developments

- **AFS**, Andrew file system (CMU 1986)
  - whole-file serving (64kbytes), whole-file caching (on local client disk, 100s of recently used files)
- **NFS** protocol (precise one-copy update semantics)
  - Spritely NFS: extend with open/close with state info held at server, add server callbacks to notify about cache entries
- **WebNFS** (Internet programs can interact with NFS server directly, bypassing mount)
- **xFS** (serverless network, file serving responsibility distributed across LAN)

5 February, 2002 30

Summary

- File service
  - crucial to the running of a distributed system
  - performance, consistency and easy recovery essential
- Design issues
  - separate flat file service from directory service and client module
  - stateless for performance and fault-tolerance
  - caching for performance
  - concurrent updates difficult with caching
  - approximation of one-copy update semantics