Distributed File Systems

B.Ramamurthy

Introduction

- Distributed file systems support the sharing of information in the form of files throughout the intranet.
- A distributed file system enables programs to store and access remote files exactly as they do on local ones, allowing users to access files from any computer on the intranet.
- Recent advances in higher bandwidth connectivity of switched local networks and disk organization have lead high performance and highly scalable file systems.

Storage systems and their properties

	Sharing	*Persis-tence*	*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 (Ch. 16)
Remote objects (RMI/ORB)	✓	×	×	1	CORBA
Persistent object store	✓	✓	×	1	CORBA Persistent Object Service
Persistent distributed object store	✓	✓	✓	✓	PerDiS, Khazana

File system modules

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

File attribute record structure

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

UNIX file system operations

filedes = open(name, mode) *filedes = creat(name, mode)*	Opens an existing file with the given *name*. 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) *count = write(filedes, buffer, n)*	Transfers *n* bytes from the file referenced by *filedes* to *buffer*. 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)	Gets the file attributes for file *name* into *buffer*.

Distributed File System Requirements

- Many of the requirements of distributed services were lessons learned from distributed file service.
- First needs were: access transparency and location transparency.
- Later on, performance, scalability, concurrency control, fault tolerance and security requirements emerged and were met in the later phases of DFS development.

Transparency

- Access transparency: Client programs should be unaware of the the distribution of files.
- Location transparency: Client program should see a uniform namespace. Files should be able to be relocated without changing their path name.
- Mobility transparency: Neither client programs nor system admin program tables in the client nodes should be changed when files are moved either automatically or by the system admin.
- Performance transparency: Client programs should continue to perform well on load within a specified range.
- Scaling transparency: increase in size of storage and network size should be transparent.

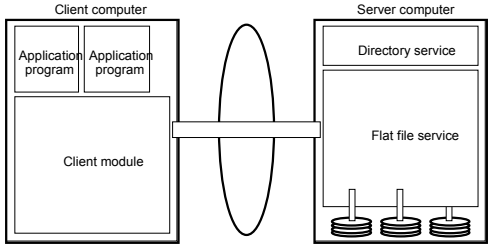
Other Requirements

- Concurrent file updates is protected (record locking).
- File replication to allow performance.
- Hardware and operating system heterogeneity.
- Fault tolerance
- Consistency : Unix uses on-copy update semantics. This may be difficult to achieve in DFS.
- Security
- Efficiency

General File Service Architecture

- The responsibilities of a DFS are typically distributed among three **modules:**
  - **Client module** which emulates the conventional file system interface
  - **Server modules**(2) which perform operations for clients on **directories** and on **files**.
- Most importantly this architecture enables stateless implementation of the server modules.

File service architecture



Flat file service Interface

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 those attributes that are not shaded in).

Primary operations are reading and writing.

Directory service Interface

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*.

Primary purpose is to provide a service for translation text names to UFIDs.

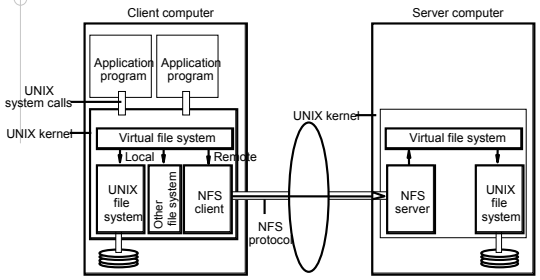
Case Studies in DFS

- We will look into architecture and operation of SUN's Network File System (NFS) and CMU's Andrew File System (AFS).

Network File System

- The Network File System (NFS) was developed to allow machines to mount a disk partition on a remote machine as if it were on a local hard drive. This allows for fast, seamless sharing of files across a network.

NFS architecture



NFS server operations (simplified) – 1

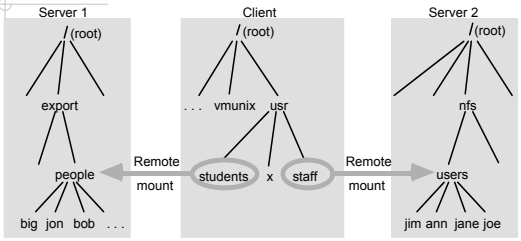
lookup(dirfh, name) -> fh, attr	Returns file handle and attributes for the file *name* in the directory *dirfh*.
create(dirfh, name, attr) -> newfh, attr	Creates a new file name in directory *dirfh* with attributes *attr* and returns the new file handle and attributes.
remove(dirfh, name) status	Removes file name from directory *dirfh*.
getattr(fh) -> attr	Returns file attributes of file *fh*. (Similar to the UNIX *stat* system call.)
setattr(fh, attr) -> attr	Sets the attributes (mode, user id, group id, size, access time and modify time of a file). Setting the size to 0 truncates the file.
read(fh, offset, count) -> attr, data	Returns up to *count* bytes of data from a file starting at *offset*. Also returns the latest attributes of the file.
write(fh, offset, count, data) -> attr	Writes *count* bytes of data to a file starting at *offset*. Returns the attributes of the file after the write has taken place.
rename(dirfh, name, todirfh, toname) -> status	Changes the name of file *name* in directory *dirfh* to *toname* in directory to *todirfh*
link(newdirfh, newname, dirfh, name) -> status	Creates an entry *newname* in the directory *newdirfh* which refers to file *name* in the directory *dirfh*.

Continues on next slide

NFS server operations (simplified) – 2

symlink(newdirfh, newname, string) -> status	Creates an entry *newname* in the directory *newdirfh* of type symbolic link with the value *string*. The server does not interpret the *string* but makes a symbolic link file to hold it.
readlink(fh) -> string	Returns the string that is associated with the symbolic link file identified by *fh*.
mkdir(dirfh, name, attr) -> newfh, attr	Creates a new directory *name* with attributes *attr* and returns the new file handle and attributes.
rmdir(dirfh, name) -> status	Removes the empty directory *name* from the parent directory *dirfh*. Fails if the directory is not empty.
readdir(dirfh, cookie, count) -> entries	Returns up to *count* bytes of directory entries from the directory *dirfh*. Each entry contains a file name, a file handle, and an opaque pointer to the next directory entry, called a *cookie*. The *cookie* is used in subsequent *readdir* calls to start reading from the following entry. If the value of *cookie* is 0, reads from the first entry in the directory.
statfs(fh) -> fsstats	Returns file system information (such as block size, number of free blocks and so on) for the file system containing a file *fh*.

Local and remote file systems accessible on an NFS client



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

NFS Revisited

- From A.Tannenbaum's text
- Three aspects of NFS are of interest: the architecture, the protocol, and the implementation.

NFS Architecture

- Allows an arbitrary collection of clients and servers to share a common file system.
- In many cases all servers and clients are on the same LAN but this is not required.
- NFS allows every machine to be a client and server at the same time.
- Each NFS server exports one or more directories for access by remote clients.
- See example enclosed.

NFS Protocol

- One of the goals o NFS is to support a heterogeneous system, with clients and servers running different operating systems on different hardware. It is essential the interface between clients and server be well defined.
- NFS accomplishes this goal by defining two client-server protocol: one for handling mounting and another for directory and file access.
- Protocol defines requests by clients and responses by servers.

Mounting

- Client requests a directory structure to be mounted, if the path is legal the server returns file handle to the client.
- Or the mounting can be automatic by placing the directories to mounted in the /etc/rc: automounting.

File Access

- NFS supports most unix operations except open and close. This is to satisfy the "statelessness" on the server end. Server need not keep a list of open connections. See the operations listed in slides 17, 18.
- (On the other hand consider your database connection… you create an object, connection is opened etc.)

Implementation

- After the usual system call layer, NFS specific layer Virtual File System (VFS) maintains an entry per file called vnode (virtual I-node) for every open file.
- Vnode indicate whether a file is local or remote.
  - For remote files extra info is provided.
  - For local file, file system and I-node are specified.
  - Lets see how to use v-nodes using a mount, open, read system calls from a client application.

Vnode use

- To mount a remote file system, the sys admin (or /etc/rc) calls the mount program specifying the remote directory, local directory in which to be mounted, and other info.
- If the remote directory exist and is available for mounting, mount system call is made.
- Kernel constructs vnode for the remote directory and asks the NFS-client code to create a r-node (remote I-node) in its internal tables. V-node in the client VFS will point to local I-node or this r-node.

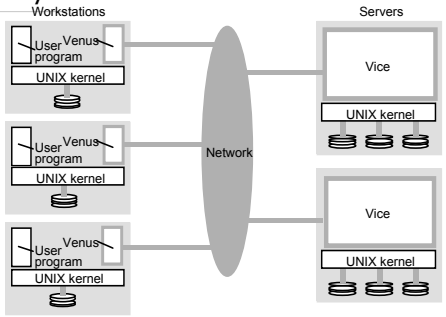
Remote File Access

- When a remote file is opened by the client, it locates the r-node.
- It then asks NFS Client to open the file. NFS file looks up the path in the remote file system and return the file handle to VFS tables.
- The caller (application) is given a file descriptor for the remote file. No table entries are made on the server side.
- Subsequent reads will invoke the remote file, and for efficiency sake the transfers are usually in large chunks (8K).

Server Side of File Access

- When the request message arrives at the NFS server, it is passed to the VFS layer where the file is probably identified to be a local or remote file.
- Usually a 8K chunk is returned. **Read ahead and caching** are used to improve efficiency.
- Cache: server side for disk accesses, client side for I-nodes and another for file data.
- Of course this leads to cache consistency and security problem which ties us into other topics we are discussing.

Distribution of processes in the Andrew File System



Summary

- Study Andrew Files System (AFS): how?
- Architecture
- APIs for operations
- Protocols for operations
- Implementation details