Distributed File Systems

Operating Systems

1

Distributed-File Systems

- Background
- Naming and Transparency
- Remote File Access
- Stateful versus Stateless Service
- File Replication
- Example Systems
  - NFS
  - AFS
- DFS project

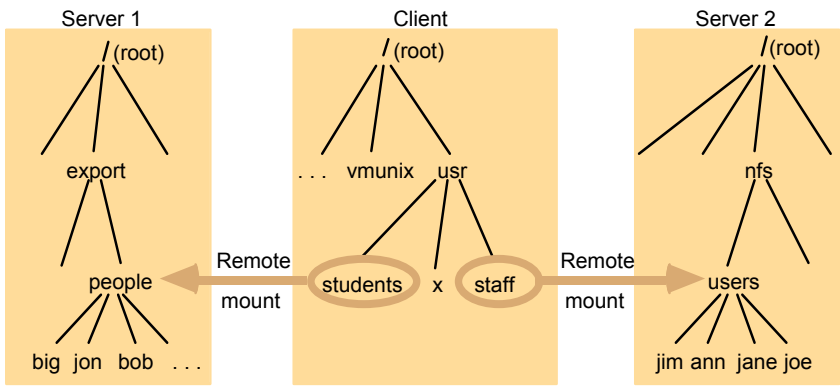
2

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 set of dispersed storage devices
- Client-Server architecture
  - 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.

3

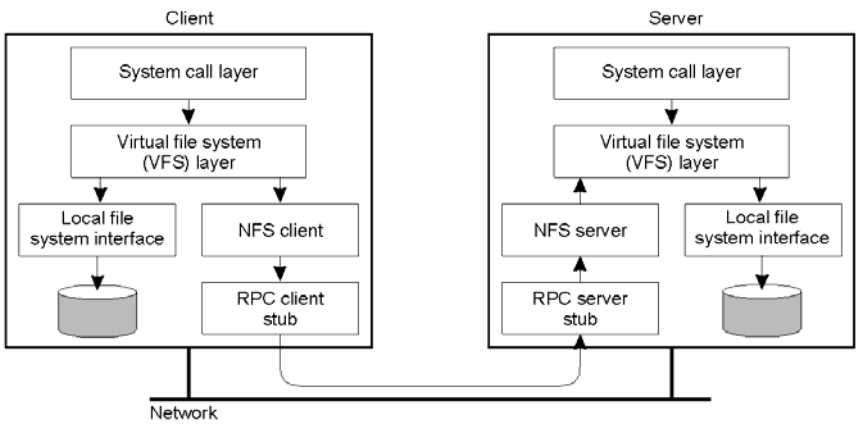
Local and remote file systems accessible on an NFS client



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.

4

NFS Architecture



5

Naming and Transparency

- *Naming* – mapping between logical and physical objects.
- 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.

6

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.
  - 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.
  - Separates the naming hierarchy from the storage-devices hierarchy.
- Most commercial DFS provide location transparency but not location independence

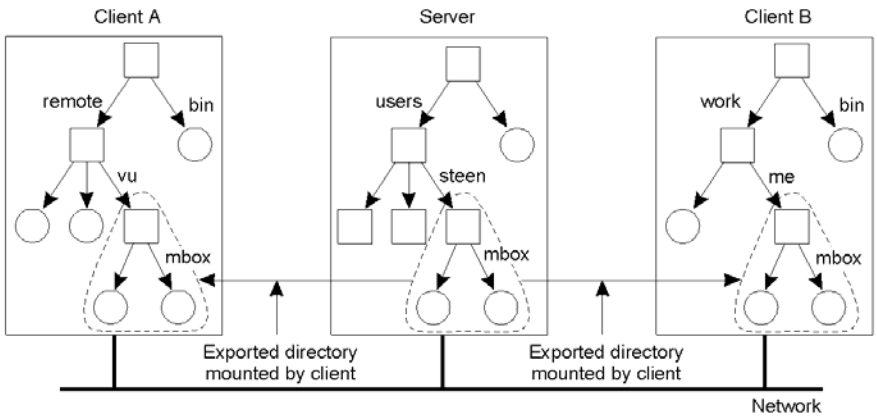
7

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.
  - E.g. NFS, AFS
- 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.
  - Implemented in some experimental file systems, e.g, Sprite from UC Berkeley

8

Naming in NFS



9

Accessing Remote Files

- Two choices:
  - Remote File Access – operations on remote files and directories are always sent to remote machine
  - Caching – remote files & directories are cached on client

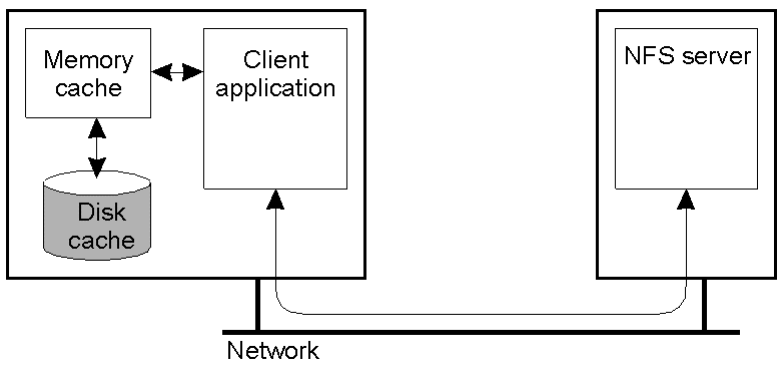
10

Caching

- 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 the file (or parts of the file) are scattered in different caches.
  - *Cache-consistency* problem – keeping the cached copies consistent with the master file.

11

Client Caching



12

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 contracted 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).

13

Issues in File Caching

- Cache Location – on disk or main memory
- (Complete) File caching vs block caching
- Cache consistency
- File Semantics
- Stateful vs Stateless servers

14

Cache Location – Disk vs. Main Memory

- 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
  - 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.

15

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 1 – scan cache at regular intervals and flush blocks that have been modified since the last scan.
  - Variation 2 – *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.

16

Consistency

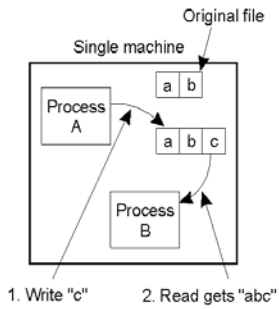
- Is locally cached copy of the data consistent with the master copy?
- Client-initiated approach
  - Client initiates a validity check.
  - Server checks whether the local data are consistent with the master copy.
- Server-initiated approach
  - Server records, for each client, the (parts of) files it caches => Stateful server
  - When server detects a potential inconsistency, it must react.

17

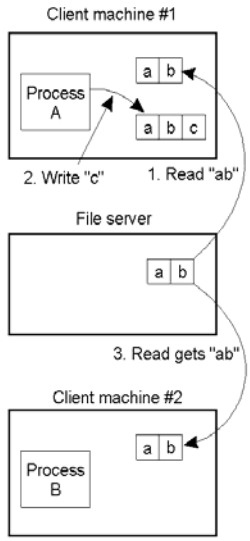
Semantics of File Sharing (1)

a) On a single processor, when a *read* follows a *write*, the value returned by the *read* is the value just written.

b) In a distributed system with caching, obsolete values may be returned.



(a)



(b)

Semantics of File Sharing (2)

Four ways of dealing with the shared files in a distributed system.

Method	Comment
UNIX semantics	Every operation on a file is instantly visible to all processes
Session semantics	No changes are visible to other processes until the file is closed
Immutable files	No updates are possible; simplifies sharing and replication
Transaction	All changes occur atomically

19

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.
- Necessary for server to maintain state if providing UNIX file semantics

20

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.

21

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.

22

Distinctions (Cont.)

- Penalties for using the robust stateless service:
  - longer request messages
  - slower request processing
  - Difficulty in providing UNIX file semantics
- 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.

23

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.

24

Example Distributed File Systems

- NFS (Network File System)
  - Caching
    - Main memory, Block cache
  - UNIX file semantics (approximate)
  - Stateless server
- AFS (Andrew File System)
  - Caching
    - Disk, Complete file caching
  - Session semantics
  - Stateful server

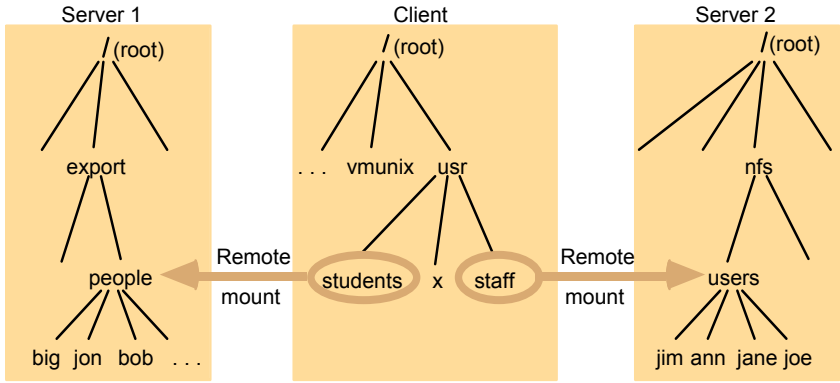
25

NFS

- 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.

26

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.

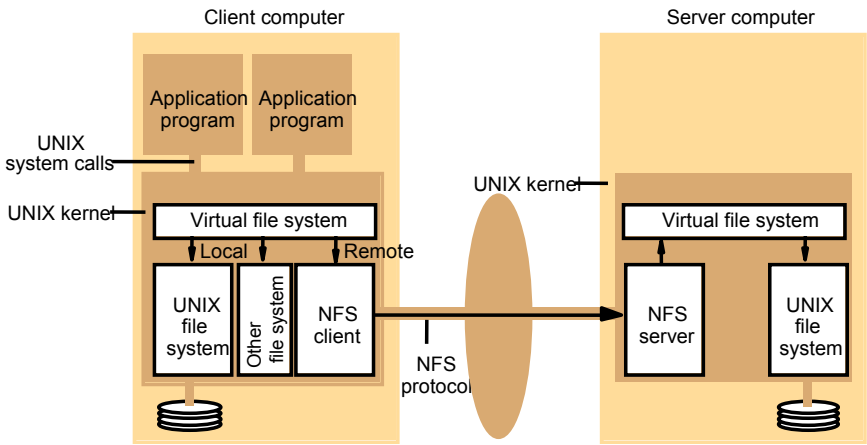
27

NFS (Cont.)

- NFS is designed to operate in a heterogeneous environment of different machines, operating systems, and network architectures; the NFS specifications are 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.
  - Separate protocols for mounting remote filesystems and remote filesystem access

28

NFS architecture



29

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.

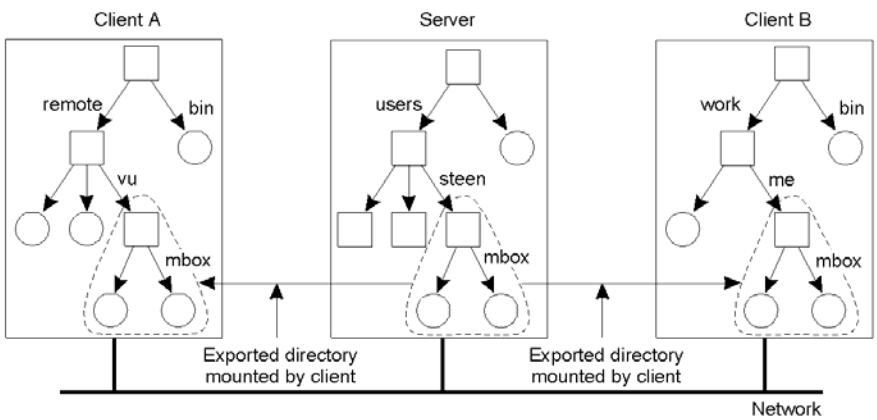
30

NFS Mount Protocol cont'd

- 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.

31

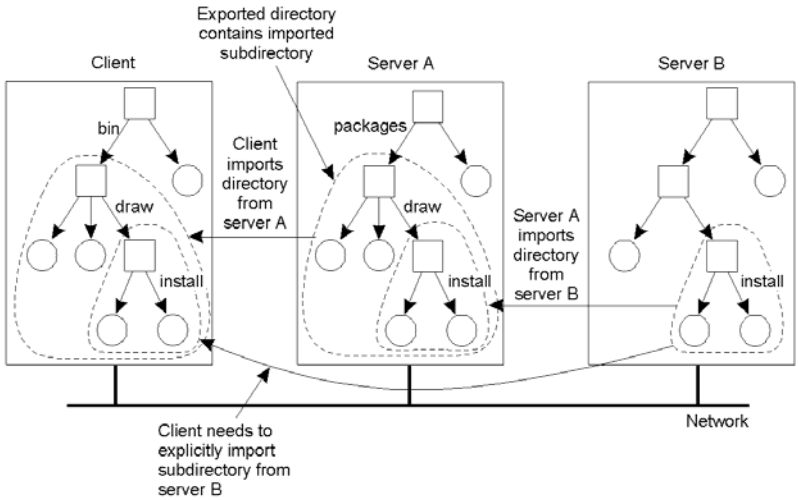
Naming (1)



32

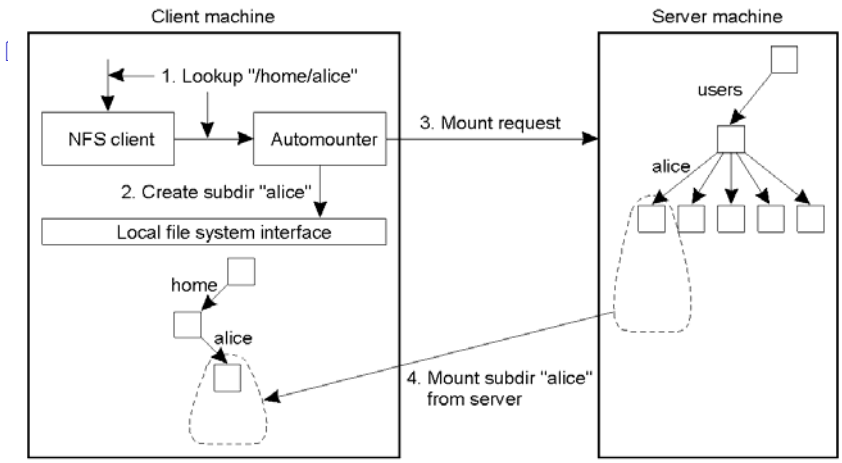
Naming (2)

- Mounting nested directories from multiple servers in NFS.



33

Automounting



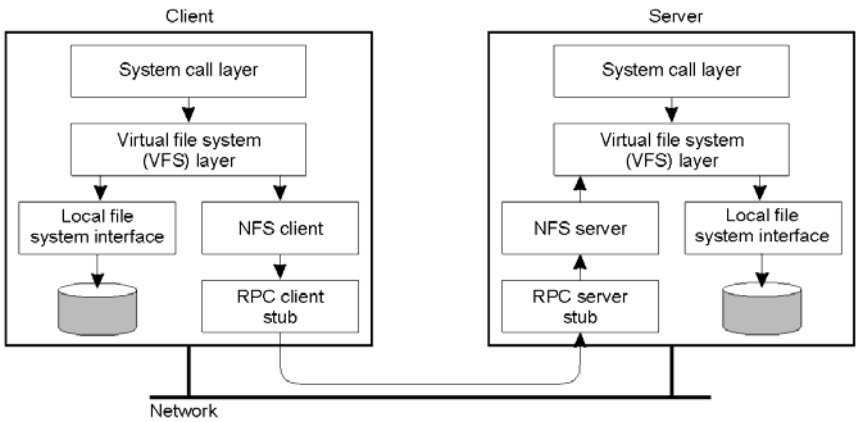
34

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.
- The original NFS protocol did not provide concurrency-control mechanisms.
  - Newer versions have support for file locking

35

NFS Architecture



36

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.

37

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.

38

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.

39

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, 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 .

40

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*.

41

Caching in NFS

- Client Caching
  - NFS client caches results of read, write, getattr, lookup, and readdir operations
  - NFS clients shares same buffer cache as local file system
  - Timestamp stored along with each cached item
    - Cached blocks are assumed to be valid for fixed time (default for directories 30 seconds, for files 3 seconds)
    - Reads after this time has elapsed result in a getattr call
  - Writes are propagated to the server either when file is closed or on sync (according to delayed write policy)
- Server cache
  - Buffer cache at server also used for caching blocks requested by remote clients
  - Server's write operations are performed using a write-through policy

42

NFS cont'd

- Biod (block input-output daemon) for read-ahead and asynchronous write
- Performance Issues (previous versions)
  - Frequent use of getattr
  - Write-through at server
- Unix file semantics not guaranteed
- Several optimizations in newer versions of NFS

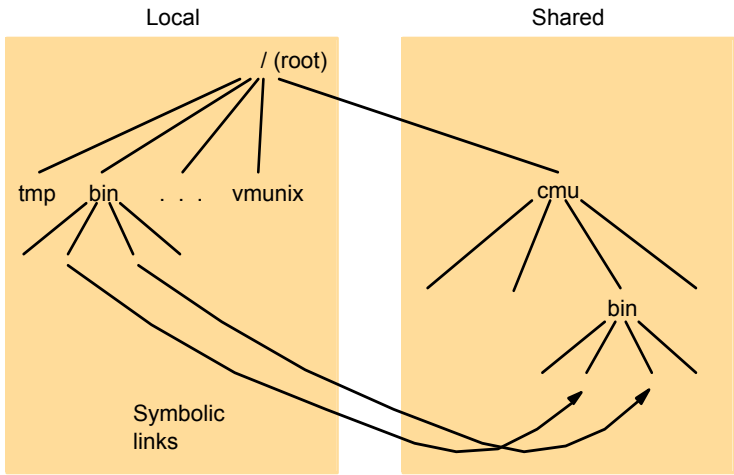
43

AFS

- Caching
  - Whole files are cached on the disk of the client
- Session semantics
  - Scalability
- Server maintains state for callbacks

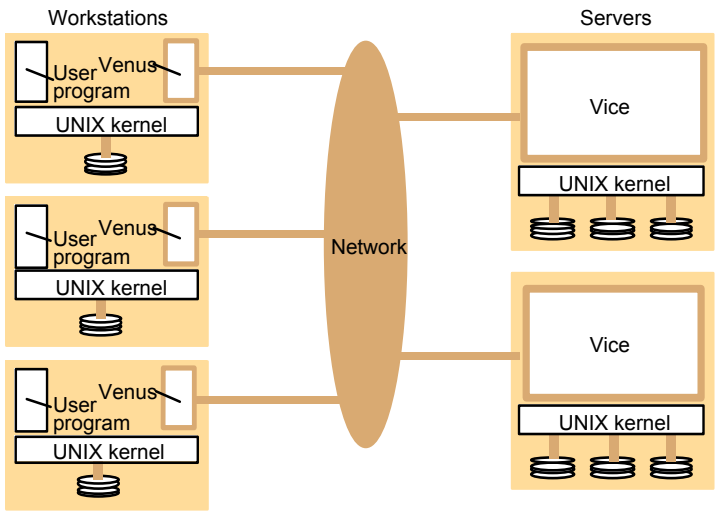
44

File name space seen by clients of AFS



45

Distribution of processes in the Andrew File System



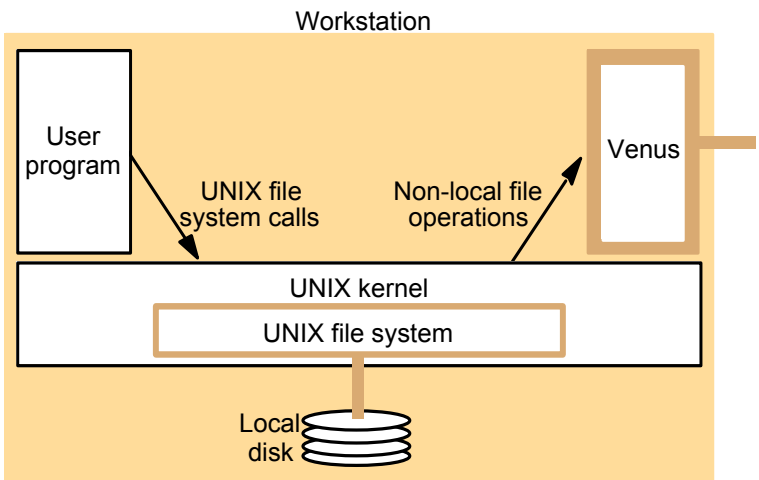
46

The main components of the Vice service interface

Fetch(fid) -> attr, data	Returns the attributes (status) and, optionally, the contents of file identified by the *fid* and records a callback promise on it.
Store(fid, attr, data)	Updates the attributes and (optionally) the contents of a specified file.
Create() -> fid	Creates a new file and records a callback promise on it.
Remove(fid)	Deletes the specified file.
SetLock(fid, mode)	Sets a lock on the specified file or directory. The mode of the lock may be shared or exclusive. Locks that are not removed expire after 30 minutes.
ReleaseLock(fid)	Unlocks the specified file or directory.
RemoveCallback(fid)	Informs server that a Venus process has flushed a file from its cache.
BreakCallback(fid)	This call is made by a Vice server to a Venus process. It cancels the callback promise on the relevant file.

47

System call interception in AFS



48

Implementation of file system calls in AFS

User process	*UNIX kernel*	*Venus*	*Net*	*Vice*
open(FileName, mode)	If *FileName* refers to a file in shared file space, pass the request to Venus.	Check list of files in local cache. If not present or there is no valid *callback promise*, send a request for the file to the Vice server that is custodian of the volume containing the file.	→	Transfer a copy of the file and a *callback promise* to the workstation. Log the callback promise.
	Open the local file and return the file descriptor to the application.	Place the copy of the file in the local file system, enter its local name in the local cache list and return the local name to UNIX.	←	
read(FileDescriptor, Buffer, length)	Perform a normal UNIX read operation on the local copy.			
write(FileDescriptor, Buffer, length)	Perform a normal UNIX write operation on the local copy.			
close(FileDescriptor)	Close the local copy and notify Venus that the file has been closed.	If the local copy has been changed, send a copy to the Vice server that is the custodian of the file.	→	Replace the file contents and send a *callback* to all other clients holding *callback promises* on the file.

49

DFS Project (tricky issues)

- Design
  - Implementing a hierarchical file system
    - Parse each component of the path separately at the client, OR
    - Parse the pathname at the server
  - Stateful vs stateless server
    - Client checks (with server) before opening cached file, OR
    - Server “callbacks”
- Implementation
  - Assume files are smaller than 16 KB
  - SUNRPC + multi-threading? Or use your own RPC or even TCP/IP directly
  - Callbacks
  - EDIT command – invoke an editor using “system” system call
    - Java Native Interface?

50