

## CS 5523 Lecture 23: NFS and Andrew

---

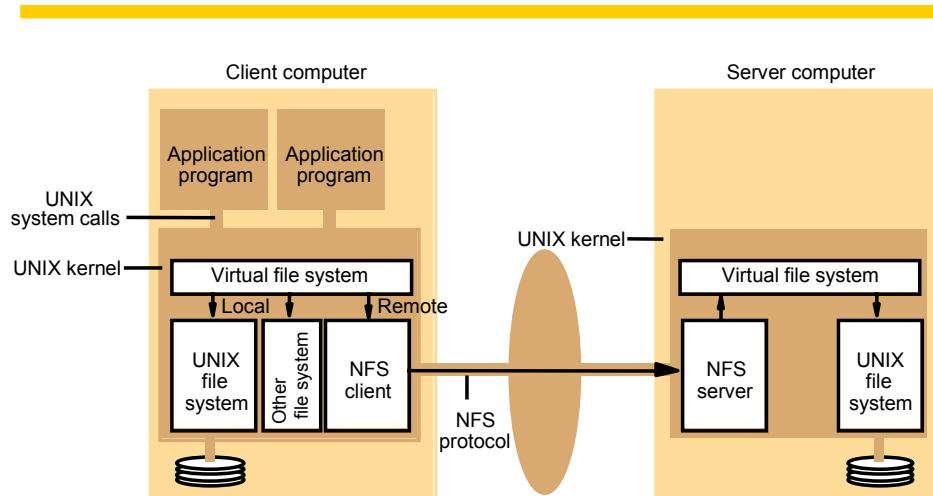
- *Hand back and discuss Laboratory 3*
- *NFS architecture*
- *NFS implementation*
- *NFS scorecard*
- *Andrew architecture*
- *Andrew implementation*
- *Andrew scorecard*

## NFS Overview (Sun OS implementation)

---

- *NFS Protocol – set of remote procedure calls for clients to perform operations on remote files*
- *NFS server resides in kernel and responds to RPC requests*
- *Based on SUN RPC protocol*
- *Portmapper enables clients to bind to services in a given host by name*
- *Any process can send requests, but must provide valid user credentials*

Figure 8.8  
NFS architecture



Instructor's Guide for Coulouris, Dollimore and Kindberg Distributed Systems: Concepts and Design Edn. 3  
© Addison-Wesley Publishers 2000

Figure 8.9  
NFS server operations (simplified) – 1

<code>lookup(dirfh, name) -&gt; fh, attr</code>	Returns file handle and attributes for the file <i>name</i> in the directory <i>dirfh</i> .
<code>create(dirfh, name, attr) -&gt; newfh, attr</code>	Creates a new file <i>name</i> in directory <i>dirfh</i> with attributes <i>attr</i> and returns the new file handle and attributes.
<code>remove(dirfh, name) status</code>	Removes file <i>name</i> from directory <i>dirfh</i> .
<code>getattr(fh) -&gt; attr</code>	Returns file attributes of file <i>fh</i> . (Similar to the UNIX <i>stat</i> system call.)
<code>setattr(fh, attr) -&gt; attr</code>	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.
<code>read(fh, offset, count) -&gt; attr, data</code>	Returns up to <i>count</i> bytes of data from a file starting at <i>offset</i> . Also returns the latest attributes of the file.
<code>write(fh, offset, count, data) -&gt; attr</code>	Writes <i>count</i> bytes of data to a file starting at <i>offset</i> . Returns the attributes of the file after the write has taken place.
<code>rename(dirfh, name, todirfh, toname) -&gt; status</code>	Changes the name of file <i>name</i> in directory <i>dirfh</i> to <i>toname</i> in directory <i>todirfh</i> .
<code>link(newdirfh, newname, dirfh, name) -&gt; status</code>	Creates an entry <i>newname</i> in the directory <i>newdirfh</i> which refers to file <i>name</i> in the directory <i>dirfh</i> .

Continues on next slide ...

Instructor's Guide for Coulouris, Dollimore and Kindberg Distributed Systems: Concepts and Design Edn. 3  
© Addison-Wesley Publishers 2000

Figure 8.9  
NFS server operations (simplified) – 2

---

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

---

Instructor's Guide for Coulouris, Dollimore and Kindberg Distributed Systems: Concepts and Design Edn. 3  
© Addison-Wesley Publishers 2000

## Virtual file system

---

- *part of Unix kernel*
- *makes access to local and remote files transparent*
- *translates between Unix file identifiers and NFS file handles*
- *keeps track of filesystems that are currently available both locally and remotely*
- *NFS file handles:*
  - *filesystem ID*
  - *i-node number*
  - *i-node generation number*
- *filesystems are mounted*
- *The VFS keeps a structure for each mounted file system*

## NFS Client Module

---

- *part of Unix kernel*
- *allows user programs to access files via UNIX system calls without recompilation or reloading*
- *one module serves all user-level processes*
- *a shared cache holds recently used blocks*
- *the encryption key for authentication of user IDs is kept in the kernel*

## Access control and authentication

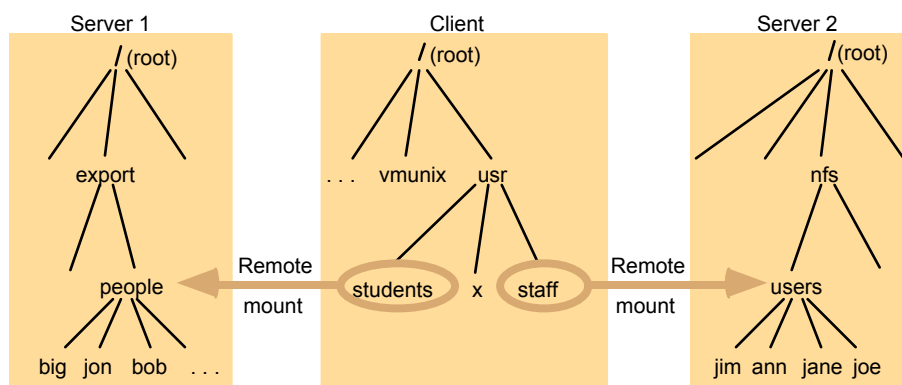
---

- *The NFS server is stateless*
- *User identity is checked with each request*
- *Because the underlying RPC protocol uses a well-known port, any one can impersonate anyone else*
- *DES encryption of user security information can be used*
- *Kerberos has also been integrated recently*

## NFS mount service

- remote file systems can be mounted in directory trees of clients
- each system has an `/etc/exports` listing filesystems available for remote mounting with list of allowed hosts
- clients use modified mount protocol
- with hard-mounted filesystems, user process is suspended until request completes
- with soft-mounted filesystems, NFS returns a failure indication after a small number of retries

Figure 8.10  
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 1; the file system mounted at `/usr/staff` in the client is actually the sub-tree located at `/nfs/users` in Server 2.

## NFS path translation

---

- *pathnames are translated in a step by step procedure by the client*
- *the file handle used for one step is used as a parameter at next lookup*

*What type of navigation does this correspond to?*

## Server caching

---

- *Read-ahead – fetches the pages following those that have been recently read*
- *Delayed-write – doesn't write out disk blocks until the cache buffer is needed for something else*
- *The UNIX sync flushes altered pages to disk every 30 seconds*
- *NFS commit operation forces the blocks of a file to be written in delayed-write mode*
- *NFS also offers write-through caching – block is written to disk before the reply is sent back to client*

*What problems occur with delayed-write?*

*What problems occur with write-through?*

## Client caching (reads)

---

- *Client caching can result in inconsistent files. Why?*
- *NFS uses timestamped validation of cache blocks:*
  - *T<sub>c</sub> is time block last validated*
  - *T<sub>m</sub> is time when block was last modified at the server*
  - *t is the freshness interval (set adaptively for individual files 3 to 30 secs)*
  - *T is current time*

*If  $(T - T_c < t)$  or  $(T - T_c \geq t$  and  $T_m \text{ client} = T_m \text{ server})$ , file is okay*
- *Validation check is made a client with each access*
- *When a new value to T<sub>m</sub> is received for a file, it is applied to all blocks*
- *Current attributes are piggy-backed on other requests*

## Client caching (writes)

---

- *Modified pages are marked as dirty and flushed at next sync*
- *Bio-daemons (block input-output) perform read-ahead and delayed-write*
  - *notified when client reads a block to get next blocks*
  - *notified when client fills a block then writes it out*

## NFS optimizations

---

- *Use UNIX BSD Fast File Systems with 8K blocks*
- *Use UDP with 8K blocks*
- *Allows clients and servers to negotiate larger block sizes*

## NFS scorecard?

---

- *Transparency*
  - *access transparency*
  - *location transparency*
  - *mobility transparency*
  - *performance transparency*
  - *scaling transparency*
- *Concurrency control possibility with support for transactions*
- *Replication*
- *Fault tolerance*
- *Heterogeneity*
- *Security*
- *Consistency*

## Andrew

---

- *Based on whole-file serving*
- *Supports whole-file caching*
- *Designed to support a large number of simultaneous users*
- *Based on the following assumptions about access:*
  - *Most files are small (< 10 K)*
  - *Reads are more common than writes (6:1)*
  - *Sequential access is common*
  - *Most files are accessed by only one user*
  - *Files are referenced in bursts*

*What are the implications of these characteristics for performance of whole-file serving and caching?*

*What kinds of files do not have these characteristics?*

## Andrew scenario

---

- *User process issues an open and there is not a current copy of the file in the client cache.*
- *The client sends a request to the server for the whole file and stores it as a local file in a local file system (client cache).*
- *An open is then performed on the local file*
- *Subsequent operations work on the local file*
- *When the client issues a close, the entire file is written back, but the copy is still kept on the client machine.*

## Andrew

- Based on whole-file serving
- Supports whole-file caching
- Designed to support a large number of simultaneous users
- Based on the following assumptions about access:
  - Most files are small (< 10 K)
  - Reads are more common than writes (6:1)
  - Sequential access is common
  - Most files are accessed by only one user
  - Files are referenced in bursts

What are the implications of these characteristics for performance of whole-file serving and caching?

What kinds of files do not have these characteristics?

Figure 8.11  
Distribution of processes in the Andrew File System

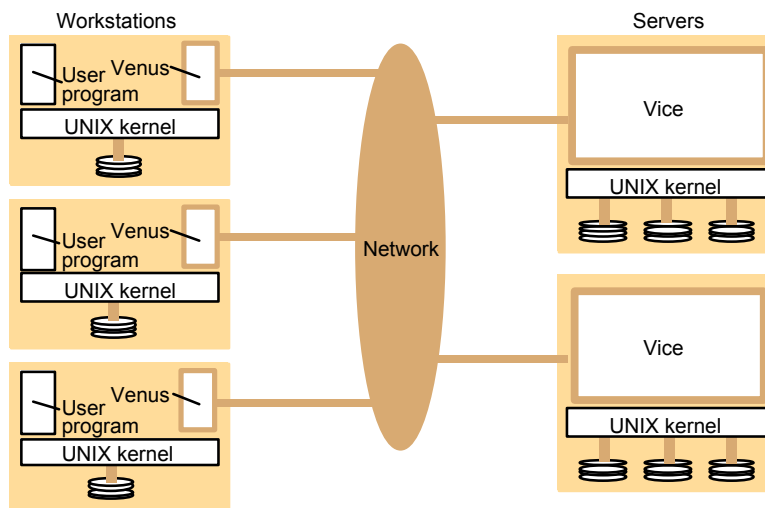
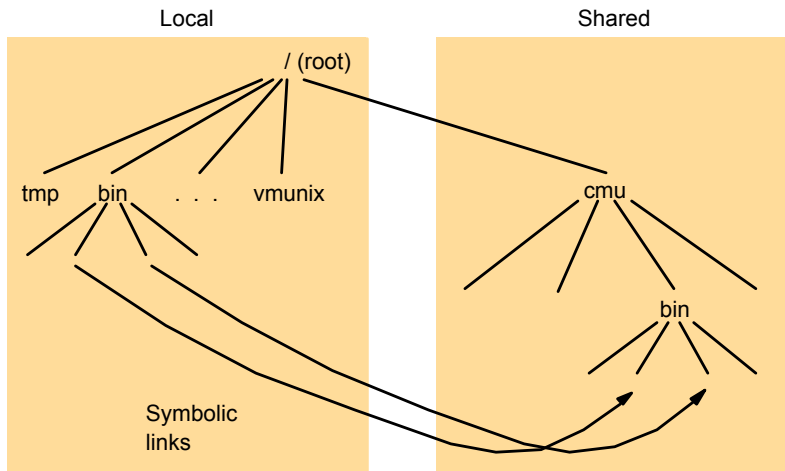
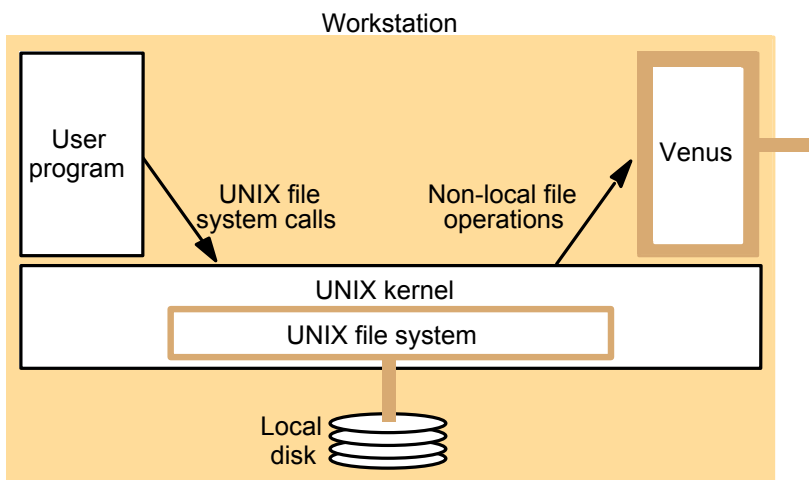


Figure 8.12  
File name space seen by clients of AFS



Instructor's Guide for Coulouris, Dollimore and Kindberg Distributed Systems: Concepts and Design Edn. 3  
© Addison-Wesley Publishers 2000

Figure 8.13  
System call interception in AFS



Instructor's Guide for Coulouris, Dollimore and Kindberg Distributed Systems: Concepts and Design Edn. 3  
© Addison-Wesley Publishers 2000

## Venus and Vice

- Venus – a user-level process that runs on each client
- Venus manages the cache (using a LRU strategy)
- Vice runs on the server and translates client requests into a flat file system
- UFIDs uniquely identify the file:
  - volume number
  - file handle
  - uniquifier
- Venus translates pathnames into UFIDs.

Figure 8.14  
Implementation of file system calls in AFS

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

Figure 8.15  
The main components of the Vice service interface

---

<i>Fetch(fid) -&gt; attr, data</i>	Returns the attributes (status) and, optionally, the contents of file identified by the <i>fid</i> and records a callback promise on it.
<i>Store(fid, attr, data)</i>	Updates the attributes and (optionally) the contents of a specified file.
<i>Create() -&gt; fid</i>	Creates a new file and records a callback promise on it.
<i>Remove(fid)</i>	Deletes the specified file.
<i>SetLock(fid, mode)</i>	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.
<i>ReleaseLock(fid)</i>	Unlocks the specified file or directory.
<i>RemoveCallback(fid)</i>	Informs server that a Venus process has flushed a file from its cache.
<i>BreakCallback(fid)</i>	This call is made by a Vice server to a Venus process. It cancels the callback promise on the relevant file.

---

Instructor's Guide for Coulouris, Dollimore and Kindberg Distributed Systems: Concepts and Design Edn. 3  
© Addison-Wesley Publishers 2000

## Cache consistency

---

- *Vice provides a callback promise with the file*
- *Callback promises a guarantees that server will notify client if someone else modifies the file*
- *Callbacks are either valid or cancelled*
- *When a file is modified, Vice makes a callback to each client cancelling the callback*
- *Venus checks the callback token whenever it opens a file*
- *If the callback token is cancelled, Venus must refetch the file.*
- *If a workstation crashes (or disconnects) musts revalidate callback promises on reconnect.*
- *Callbacks must be renewed if a long time has elapsed since last communication with server.*

## AFS-1 semantics

---

- *After successful open - latest(F, S)*
- *After failed open – failure(S)*
- *After successful close – updated (F,S)*
- *After failed close – failed (S)*

*F = file on client, S = server*

## AFS-2 semantics

---

*AFS-2 requires Vice to maintain state on behalf of Venus clients. The lists of Venus clients are maintained over server failures and updated using atomic operations:*

- *After successful open - latest(F, S)*
- *After failed open – failure(S)*
- *After successful close – updated (F,S)*
- *After failed close – failed (S)*

*F = file on client, S = server*

## AFS cache consistency

---

- *Cache consistency checks between workstations only done on open and close*
- *Callbacks don't provide true consistency in a genuinely concurrent environment (why?)*
- *Consistency is maintained among processes on same workstation (why?)*

## Andrew scorecard?

---

- *Transparency*
  - *access transparency*
  - *location transparency*
  - *mobility transparency*
  - *performance transparency*
  - *scaling transparency*
- *Concurrency control possibility with support for transactions*
- *Replication*
- *Fault tolerance*
- *Heterogeneity*
- *Security*
- *Consistency*

For next time:

---

- *Read CDK 8.3-8.5*