Distributed File Service: Case Studies

- NFS (Sun)
- Andrew (CMU)
- Coda (CMU)

NFS

Sun Microsystems' *Network File System* (NFS) introduced in 1985

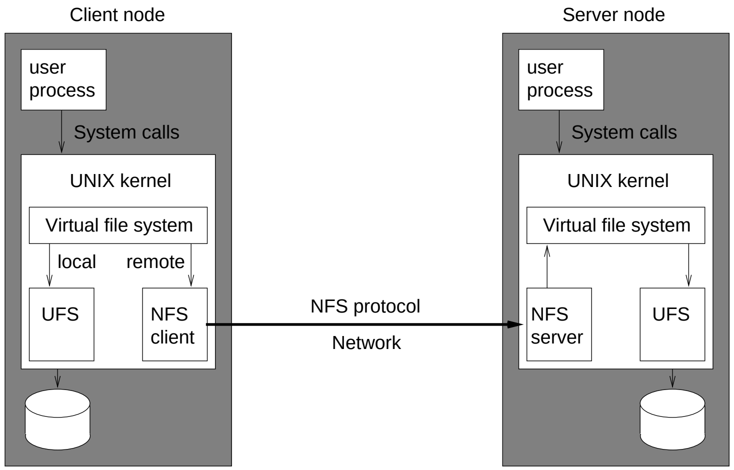
- design goals:
  - product
  - support heterogeneity
- interface definition released into public domain in 1989
- quickly established as an *de-facto* standard since,
  supported by virtually all vendors

R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon: The design and implementation of the Sun Network File System. In *Proc. 1985 USENIX Conf.*, Portland, OR, USA, 1985.

Features

- each node can be client and server
  however, usually configure some machines as servers
- integration of dissimilar platforms
  - UNIX systems on various hardware
  - non-UNIX systems like VMS, PCs, Macs
- remote files added to local file name space
- access is via normal local OS interface (fully compatible)
- *stateless servers* with *idempotent* operations
- extensive caching makes cost of remote access comparable to local
- *mounting* of remote file systems supports replication and migration (at the file system level)

NFS architecture



Implementation

- UNIX kernel extended by three components:
  - *virtual file system* (VFS)
  - *NFS client module*, and
  - *NFS server module*
- VFS redirects network file requests to remote nodes via NFS modules
- NFS client and NFS server modules communicate via RPC

VFS

- uses a form of UIDs called *file handles.* UNIX implementation:

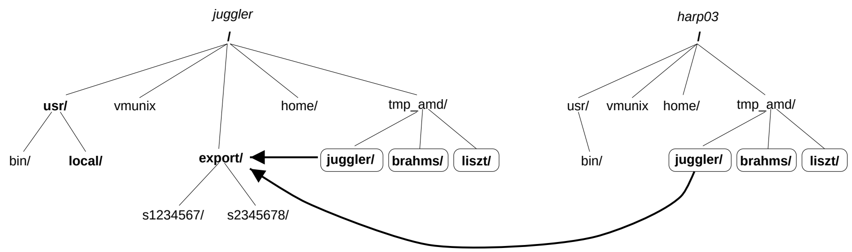
  <FS Id ; i-node number ; i-node generation number >

  generation number makes i-node unique; stored with file
- maps between file handles (for NFS requests) and UNIX i-nodes (for UFS requests)
- for each open file, VSF maintains a descriptor, called called *v-node*
  - for local files, contains i-node
  - for remote files, contains NFS file handle

Client module

- implements exact UFS semantics
- shares UFS buffer cache
- communicates with server via RPC
- RPC uses DES encryption for authentication
- access control based on usual UNIX user/group IDs
- stateless server requires authenticating each request

Path name translation



File system hierarchy in CSE (simplified)

- any directory may be a *mount point* of a remote FS

⇒ must translate path component-by-component

- client caches translations

Mount service

- each server maintains *export list* containing
  - a list of file systems available for remote mounting
  - an *access control list* for each of them
- clients authorised to do so can *mount* an exported FS (or any subtree thereof) at an arbitrary *mount point* in their local file name space
  - *hard mounts* block clients when remote service is unavailable
  - *soft mounts* (or *auto mounts*) reject request when resource unavailable

Automount

- user-level process on client node
- invoked by client module when resolving a name containing an automount point
- automounter checks availability of server, if successful invokes mount service to mount remote FS
- actual mount is at a different mount point, pointed to by a symbolic link installed by automounter
- once mounted, automounter is not involved in further name resolutions (as link is in place)
- automounter checks link periodically, if unused for a while, removes link and unmounts remote FS

Caching: Server

Server uses caching mostly like normal UFS:

- LRU buffer cache in physical memory
- read-ahead
- **however**, write-through required to prevent loss of updates on server failure

⇒ use of NVRAM allows *delayed write* to disk

Caching: Client

Client caches:

- name resolutions
- file blocks (read-ahead and delayed-write)
- cached copies may be out-of-date

⇒ use time stamping:
  - file modification time stamp (from server) held with cached blocks, validated when
    - new `open` operation on file
    - reading an uncached block from server
    - when cache is too old (e.g. 3 sec for files, 30 sec for dirs)

⇒ inconsistent caches unlikely, but not impossible

Does NFS match the goals?

match?	*goal*	*description*
	diversity of application	support all applications
	ease of use	
	matching requirements	don't pay for what you don't use
	consistent naming	same name for all clients
	location transparence	moving file shouldn't change its name
	access transparent	no difference between local and remote files
	data consistency	provide locking, transactions
	information protection	authorised access only
	user mobility	same (efficient) access from anywhere
	performance	not worse than centralised system
	reliability/fault tolerance	crashes rare, recovery possible
	hardware independence	accommodate different media
	scalability	accommodate growth of system

Andrew

Carnegie-Mellon University's *Andrew File System* (AFS) [Satyanarayanan et al. 1983]

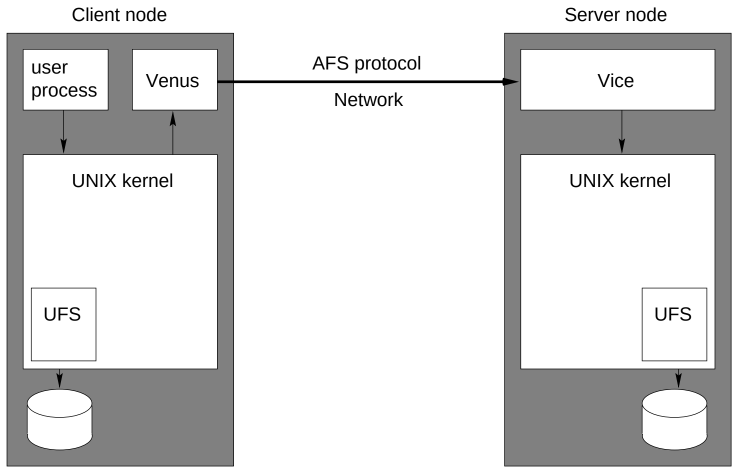
- design goals:
  - scalability
  - security
  - high reliability
- implemented on top of BSD 4.3 UFS
- available for UNIX, Mach
- compatible with NFS

J. Morris, M. Satyanarayanan, M.H. Conner, J.H. Howard, D.S. Rosenthal, and F.D. Smith: Andrew: A distributed personal computing environment. *Comm. ACM*, 29(3):184–201, 1986.

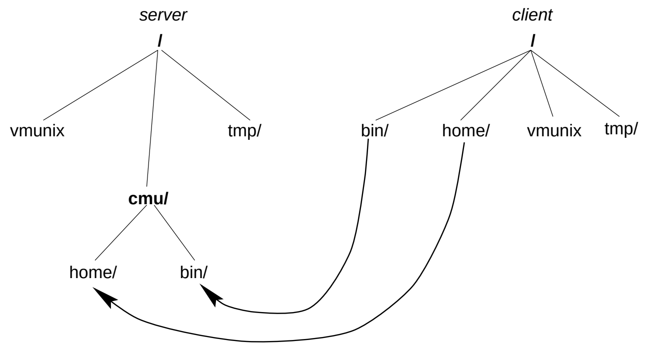
Features

- highly reliable *servers*
  - running AFS component *Vice* as user-level process
  - never run any actual user processes
  - physically safe
- *workstations* (i.e. clients)
  - running AFS component *Venus* as user-level process
  - running user code
  - untrusted
  - autonomous (i.e. fully independent of each other)
- assumes reliable network

AFS architecture



AFS name space



All files are either *local* or *shared*

- local files (mostly OS and /tmp) reside on local node
- shared files (almost all) reside on server

Strategy

- only user-level processes, minimal kernel modifications
- dedicated servers ⇒ reliability
- clients cache whole files (AFS-2) or large (64k) sections (AFS-3)
- cache coherency achieved via *callback* mechanism
- all client access to data is via normal UNIX system calls (to the cached copy)
- (almost) stateless server using encryption for authentication
- design optimised towards usual UNIX file access patterns

Implementation

- minimal modifications to UNIX kernel:
  - intercept `open`, `close` calls on shared files
    invoke Venus to handle
  - use AFS file identifiers (*fids*) instead of UNIX file descriptors
- path names translated to fids for communication
  between Venus and Vice
- use local partition as LRU file cache for shared files
- network transfers bundled into 64k chunks

Vice

- implements *flat* file service based on *fids*
  < volume number ; file handle ; uniquifier >
- *volumes* are groupings of files: common subtrees within a partition
- volumes can be read-only
- volumes can be migrated, replicated
- Venus keeps maps of volume locations

Cache Coherency

- AFS-1 used a timestamp scheme similar to NFS
- AFS-2 uses a *write invalidate* protocol called *callback*
  - when a client obtains a file, a *callback* is logged at the server
  - when a file is closed after writing,
    the client sends a updated copy to the server
  - the server sends a *callback* message to all clients holding copies
  - clients invalidate cached copy for future opens
    (but continue using cached copy for already opened files)

  ⇒ no *strict* cache coherence,
  but good approximation allowing efficient implementation

AFS-3

- supports a global (intercontinental!) name space
- system decomposed into autonomous "cells"
- performance improvements:
  - Venus in kernel
  - use vnodes
  - cache 64k chunks rather than whole files

Does Andrew match the goals?

match?	*goal*	*description*
	diversity of application	support all applications
	ease of use	
	matching requirements	don't pay for what you don't use
	consistent naming	same name for all clients
	location transparence	moving file shouldn't change its name
	access transparent	no difference between local and remote files
	data consistency	provide locking, transactions
	information protection	authorised access only
	user mobility	same (efficient) access from anywhere
	performance	not worse than centralised system
	reliability/fault tolerance	crashes rare, recovery possible
	hardware independence	accommodate different media
	scalability	accommodate growth of system

Coda

CMU's *Coda* file system [Satyanarayanan et al. 1990]
is based on AFS-2

- design goals:
  - increased fault tolerance
  - support for disconnected operation
    Idea: use cached data for temporary autonomous operation
    fix inconsistencies on re-integration

M. Satyanarayanan, J.J. Kistler, P. Kumar, M.R. Okasaki, E.H. SIegel, and D.C. Steere: Coda: A highly available file system for a distributed workstation environment. *IEEE Trans. Computers*, 39:447–459, 1990.

Replication

- data are replicated on
  - multiple servers
  - local cache
- set of servers replicating a certain volume is *volume storage group* (VSG)
- presently reachable servers comprise *available VSG* (AVSG)
- *write-through* updates to AVSG (*read-one/write-all*)
- *call-back* mechanism to notify holders of copies
- requests serviced off cache without contacting server
- *optimistic* coherency protocol guarantees *detection* not *prevention* of inconsistencies

Support for disconnected operation

- three modes of operation:
  - *hoarding*: normal connected operation
  - *emulation*: disconnected operation
  - *reintegration*: exchanging updates when reconnected

Hoarding

- Accumulates "interesting" files in cache; residence determined based on combination of
  - reference history
  - user-supplied *hoard profiles*
- system facilitates construction of hoard profiles by recording reference patterns

Emulation

When disconnected

- lives off cached data
- Venus emulates file server actions:
  - assigns temporary fids
  - perform updates on cached copies
  - return error on cache miss
  - perform transactional updates of metadata (using *RVM*); keep copies of "deleted" files
  - maintain logs of changes

Reintegration

On reconnect:

- propagate changes to AVSG
- done a volume at a time, while activities on volume frozen
- update done by servers in a single, distributed transaction
- on collisions:
  - signal consistency problem with data
  - automatically repair (if possible) collisions in meta-data
- Coda provides tools supporting selective replay of logs and manual construction of consistent versions

Does Coda match the goals?

match?	*goal*	*description*
	diversity of application	support all applications
	ease of use	
	matching requirements	don't pay for what you don't use
	consistent naming	same name for all clients
	location transparence	moving file shouldn't change its name
	access transparent	no difference between local and remote files
	data consistency	provide locking, transactions
	information protection	authorised access only
	user mobility	same (efficient) access from anywhere
	performance	not worse than centralised system
	reliability/fault tolerance	crashes rare, recovery possible
	hardware independence	accommodate different media
	scalability	accommodate growth of system

Summary (Andrew and Coda)

Good performance achieved by

- keeping number of servers small
- letting workstations do most the work
- extensive caching
- optimised towards typical UNIX usage patterns

[1] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. The design and implementation of the Sun Network File System. In *Proc. 1985 USENIX Conf.*, Portland, OR, USA, 1985.

[2] J. Morris, M. Satyanarayanan, M.H. Conner, J.H. Howard, D.S. Rosenthal, and F.D. Smith. Andrew: A distributed personal computing environment. *Comm. ACM*, 29(3):184–201, 1986.

[3] M. Satyanarayanan, J.J. Kistler, P. Kumar, M.R. Okasaki, E.H. Siegel, and D.C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Trans. Computers*, 39:447–459, 1990.