Cristina Nita-Rotaru

CS505: Distributed Systems

Distributed file systems.

REQUIRED READING

- Design and Implementation or the Sun Network File System, R. Sandberg , D. Goldberg , S. Kleiman , D. Walsh, B. Lyon, 1985, http://www.stanford.edu/class/cs240/readings/nfs.pdf
- Scale and performance in a distributed file system, J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West, TOCS, 1988
- Scalable, Secure, and Highly Available Distributed File Access
  M. Satyanarayanan, IEEE Computer
  May 1990

1: Distributed file systems design

What is a File System?

- **File system**: a method of **organizing and storing** computer files and the data they contain to make it easy to find and access them.
- File systems are responsible of:
  - Organization
  - Storage
  - Retrieval
  - Naming
  - Sharing
  - Protection

Files and Directories

- **File**: contains data and attributes - file length, timestamps (create, read, write), file type, owner, access control list
- **Directory**: special type of file, provides a mapping between file names and internal file identifiers
- **Metadata**: extra information stored and needed for the management of files (includes file attributes and directories)
- **UNIX file system operations**: open, create, close, read, write, lseek, link, unlink, stat

Distributed File Systems

- **A Distributed File System (DFS) is a file system that supports sharing of files and resources in the form of persistent storage over a network**
- First file servers were developed in the 1970s
- First widely used distributed file system was Sun's Network File System (NFS) introduced in **1985**
- Examples of distributed file systems: Andrew File System (CMU), CODA (CMU), Google File System (Google)

Requirements for Distributed File Systems

- A DFS should appear to its users to be a conventional, centralized file system.
- 1) Transparency
- 2) File replication
- 3) Concurrent file updates
- 4) Hardware and operating systems heterogeneity
- 5) Fault tolerance
- 6) Performance
- 7) Security

Transparency in Distributed Systems

- Access transparency: local and remote resources are accessed using identical operations
- Location transparency: resources are accessed without knowledge of their location
- Concurrency transparency: several processes operate concurrently using shared resources without interference between them
- Replication transparency: multiple replicas are used, users are not aware of the replicas
- Failure transparency: concealment of faults
- Mobility transparency: movement of resources
- Performance transparency: allows system to be reconfigured to improve performance as load vary
- Scaling transparency: system expands in scale without changing the system structure

Transparency and File Systems

- Access transparency: A single set of operations is provided for access to local/remote files.
- Location transparency: Client programs see a uniform file name space. Name of a file doesn't need to be changed when the file's physical location changes.
- Mobility transparency: Neither client programs nor system administration tables in client nodes need to be changed when files are moved.
- Performance transparency: clients should continue to perform satisfactorily while the load on the system varies in a specified range.
- Scaling transparency: service can be expanded by incremental growth with a wide range of loads and network sizes.

More Requirements for DFS

- **2) File Replication**: A file may be represented by several copies for service efficiency and fault tolerance.
- **3) Concurrent File Updates**:
  - Changes to a file by one client should not interfere with the operation of other clients simultaneously accessing the same file.
  - **One-copy update semantics**: the file contents seen by all of the processes accessing or updating a given file are those they would see if only a single copy of the file existed.

More Requirements for DFS

- **4) Hardware and operating systems heterogeneity**: service interface should be defined so that client and server software can be implemented for different operating systems and platforms.
- **5) Fault Tolerance**:
  - At most once invocation semantics.
  - At least once semantics with a server protocol designed with idempotent operations (i.e., duplicated requests do not result in invalid updates to files).

More Requirements for DFS

- **6) Performance**: most important aspect is the time to answer requests
  - Conventional systems: disk-access time and a small amount of CPU-processing time.
  - DFS: additional overhead includes the time to deliver the request to a server, as well as the time to get the response across the network back to the client, CPU overhead of running the communication protocol software.
- The performance of an ideal DFS would be comparable to that of a conventional file system.

More Requirements for DFS

- **7) Security:** for file systems most critical security services are:
  - Access Control: per object, list of allowed users and access allowed to each
  - Client Authentication: need to authenticate client requests so that access control at the server is based on correct client/user identifiers

Semantics of File Sharing

- **UNIX semantics**: value read is the value stored by last write; Writes to an open file are visible immediately to others that have this file opened at the same time.
- **Session semantics**: Write to an open file by a user is not visible immediately by other users that have files opened already; Once a file is closed, the changes made by it are visible by sessions started later.
- **Immutable-Shared-Files semantics**: A sharable file cannot be modified. File names cannot be reused and its contents may not be altered. Simple to implement.
- **Transactions**: All changes have all-or-nothing property.

Stateful vs. Stateless Design

- **Stateful**: server keeps track of info about client requests.
  - what files are opened by a client, connection identifiers, server caches

[redacted]

- **Stateless**: each client request provides complete information needed by the server (i.e., filename, file offset)
  - server can maintain information on behalf of the client, but it is not required

[redacted]

Stateful vs. Stateless Design

- **Stateful**: server keeps track of info about client requests.
  - what files are opened by a client, connection identifiers, server caches

[redacted]

- **Stateless**: each client request provides complete information needed by the server (i.e., filename, file offset)
  - server can maintain information on behalf of the client, but it is not required

[redacted]

Stateful vs. Stateless Design

- **Stateful**: server keeps track of info about client requests.
  - what files are opened by a client, connection identifiers, server caches
  - increased performance
  - if server crashes, it looses all its volatile state information
  - if client crashes, the server needs to know to claim state space
- **Stateless**: each client request provides complete information needed by the server (i.e., filename, file offset)
  - server can maintain information on behalf of the client, but it is not required
  - server failure is identical to slow server (client retries...)
  - each request must be idempotent

Where to Cache?

- **Server's disk**: slow performance
- **Server main memory**: cache management issue, how much to cache, replacement strategy; used in high-performance web-search engine servers
- **Client main memory**: faster to access from main memory than disk; Compete with the virtual memory system for physical memory space
- **Client-cache on a local disk**: large files can be cached; the virtual memory management is simpler; a workstation can function even when it is disconnected from the network

Client Caching

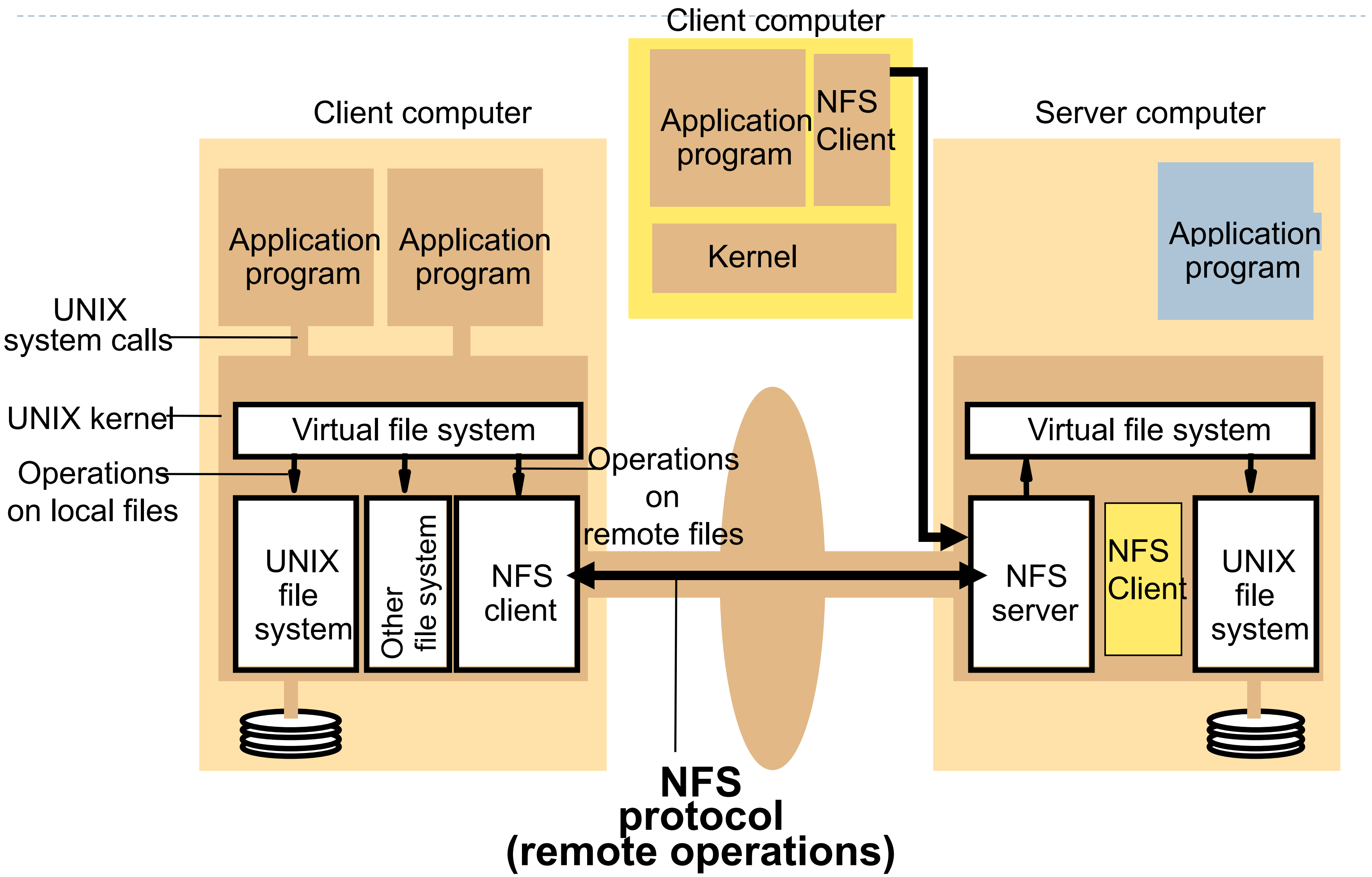
- **Write-through**: all writes are carried out immediately
- **Delayed-write**: delays writing at the server
- **Write-on-close**: delay writing until the file is closed at the client

2: Network File System (NFS)

Network File System (NFS)

- Originally developed by Sun Microsystems, introduced in 1985
- To encourage its adoption, the definition of the key interfaces were placed in the public domain
- Communication:
  - Version 2 of the protocol originally operated entirely over UDP
  - Sun Microsystems added support for TCP as a transport for NFS at the same time it added support for Version 3
- State maintained by the server:
  - Version 2 and 3 stateless, version 4 introduced stateful protocols

NFS Architecture



NFS Architecture Implementation

- Some NFS clients and servers implementation run at application-level as libraries or processes (e.g. early Windows and MacOS implementations)
- Unix kernel implementation have advantages:
  - Binary code compatible - no need to recompile applications; Standard system calls that access remote files can be routed through the NFS client module by the kernel
  - Shared cache of recently-used blocks at client
  - Kernel-level server can access i-nodes and file blocks directly but a privileged (root) application program could do almost the same.

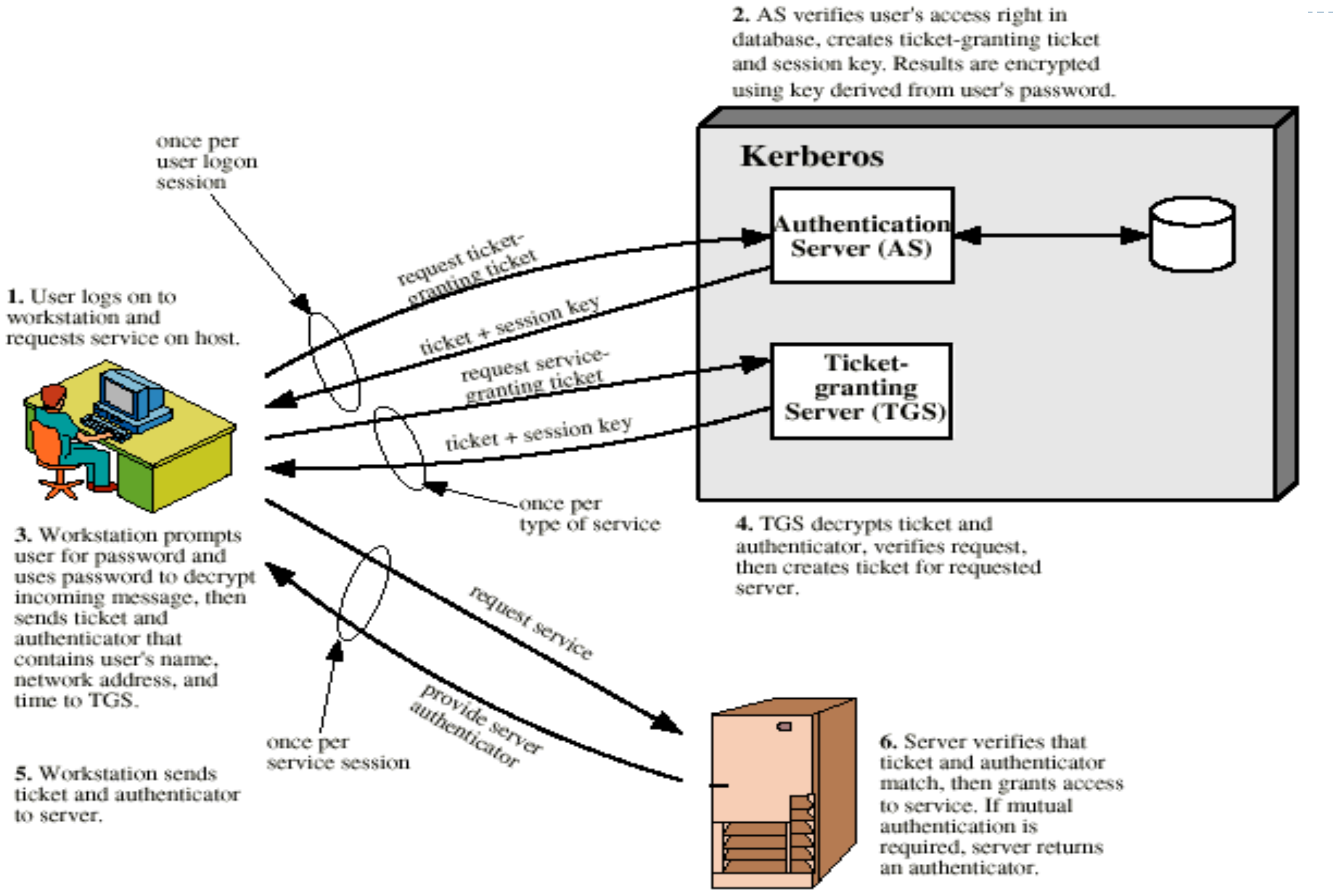
NFS Server Operations (simplified)

- *read(fh, offset, count) -> attr, data*
- *write(fh, offset, count, data) -> attr*
- *create(dirfh, name, attr) -> newfh, attr*
- *remove(dirfh, name) status*
- *getattr(fh) -> attr*
- *setattr(fh, attr) -> attr*
- *lookup(dirfh, name) -> fh, attr*
- *rename(dirfh, name, todirfh, toname)*
- *link(newdirfh, newname, dirfh, name)*
- *readdir(dirfh, cookie, count) -> entries*
- *symlink(newdirfh, newname, string) -> status*
- *readlink(fh) -> string*
- *mkdir(dirfh, name, attr) -> newfh, attr*
- *rmdir(dirfh, name) -> status*
- *statfs(fh) -> fsstats*

NFS Access Control and Authentication

- **Stateless server, so the user's identity and access rights must be checked by the server on each request.**
  - In the local file system they are checked only on *open()*
- Every client request is accompanied by the userID and groupID
- Server is exposed to impersonation attacks unless the userID and groupID are protected by encryption
- Kerberos has been integrated with NFS to provide a stronger and more comprehensive security solution.

Overview of Kerberos


2. AS verifies user's access right in database, creates ticket-granting ticket and session key. Results are encrypted using key derived from user's password.
Kerberos
Authentication Server (AS)
Ticket-granting Server (TGS)
once per user logon session
request ticket-granting ticket
ticket + session key
request service-granting ticket
ticket + session key
1. User logs on to workstation and requests service on host.
once per type of service
4. TGS decrypts ticket and authenticator, verifies request, then creates ticket for requested server.
3. Workstation prompts user for password and uses password to decrypt incoming message, then sends ticket and authenticator that contains user's name, network address, and time to TGS.
request service
provide server authenticator
once per service session
5. Workstation sends ticket and authenticator to server.
6. Server verifies that ticket and authenticator match, then grants access to service. If mutual authentication is required, server returns an authenticator.

V4: Authentication Service Exchange

Goal: Obtain Ticket-Granting Ticket

C $\rightarrow$ AS: $ID_c \,||\, ID_{tgs} \,||\, TS_1$

AS $\rightarrow$ C: $E_{Kc}\,[K_{c,tgs} \,||\, ID_{tgs} \,||\, TS_2 \,||\, Lifetime_2 \,||\, Ticket_{tgs}]$

$Ticket_{tgs} = E_{K_{tgs}}\,[K_{c,tgs} \,||\, ID_C \,||\, AD_C \,||\, ID_{tgs} \,||\, TS_2 \,||\, Lifetime_2]$

ID_{tgs} denotes the identifier of the Ticket Granting Server (TGS)

TS1 and TS2 are timestamps

K_C is the key shared by the AS and client C

$K_{C,\ tgs}$ is the key shared by the TGS and client C

K_{tgs} key known by AS and the TGS

$Ticket_{tgs}$ …is the ticket

Lifetime is the validity of the ticket

AD is address identifier

V4: Ticket-Granting Service Exchange

Goal: Obtain Service-Granting Ticket

$C \rightarrow TGS$: $ID_S \,||\, Ticket_{tgs} \,||\, Authenticator_C$

$TGS \rightarrow C$: $E_{K_{c,tgs}}[\, K_{C,S} \,||\, ID_S \,||\, TS_4 \,||\, Ticket_S \,]$

$Ticket_{tgs} = E_{K_{tgs}}[\, K_{C,tgs} \,||\, ID_C \,||\, AD_C \,||\, ID_{tgs} \,||\, TS_2 \,||\, Lifetime_2 \,]$

$Ticket_S = E_{K_S}[\, K_{C,S} \,||\, ID_C \,||\, AD_C \,||\, ID_s \,||\, TS_4 \,||\, Lifetime_4 \,]$

$Authenticator_C = E_{K_{C,\, tgs}}[\, ID_C \,||\, AD_C \,||\, TS_3 \,]$

K_S is the key shared by the TGS and server S

V4: Client-Server Authentication Exchange

Goal: Obtain Service

$C \rightarrow S$: $Ticket_S \,||\, Authenticator_C$

$S \rightarrow C$: $E_{K_{C,S}}[\, TS_5 + 1 \,]$

$Ticket_S = E_{K_S}[\, K_{C,S} \,||\, ID_C \,||\, AD_C \,||\, ID_s \,||\, TS_4 \,||\, Lifetime_4 \,]$

$Authenticator_C = E_{K_{C,\,S}}[\, ID_C \,||\, AD_C \,||\, TS_5 \,]$

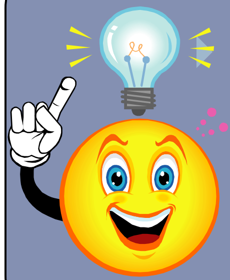
Kerberized NFS

- Kerberos is too costly to apply on each file access request
- Kerberos is used in the mount service to authenticate the user's identity
  - User's UserID and GroupID are stored at the server with the client's IP address
  - For each file request UserID, GroupID and IP address sent must match those stored at the server

What are the disadvantages of this approach?

Kerberized NFS

- Kerberos is too costly to apply on each file access request
- Kerberos is used in the mount service to authenticate the user's identity
  - User's UserID and GroupID are stored at the server with the client's IP address
  - For each file request UserID, GroupID and IP address sent must match those stored at the server
- Disadvantages of this approach:



  - Cannot accommodate multiple users sharing the same client computer
  - All remote filestores must be mounted each time a user logs in

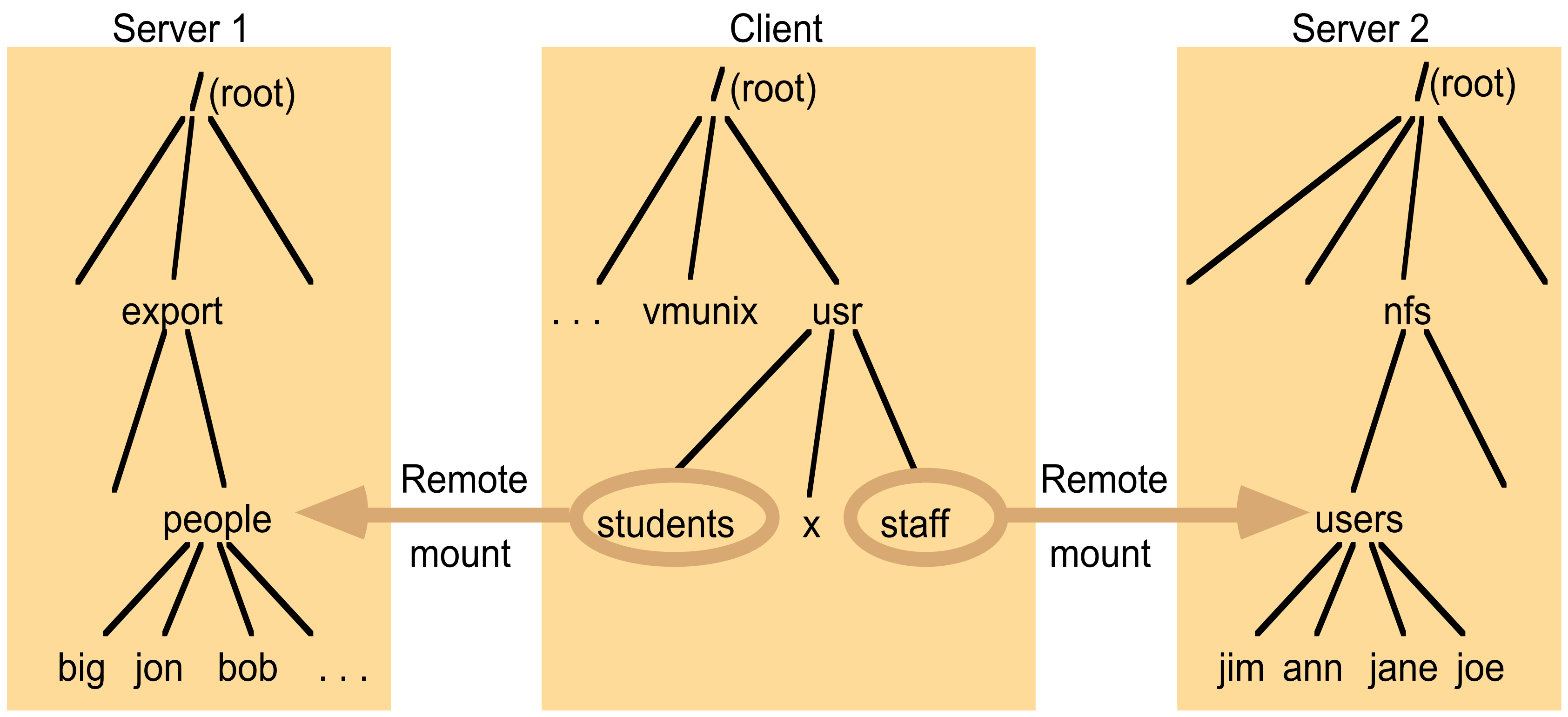
Mount Service

- Clients use the UNIX Mount operation:

  mount(remotehost, remotedirectory, localdirectory)
- The mount command communicates with the mount service process on the remote host via RPC.
  - The RPC operation takes the directory pathname and returns the file handle of the specified directory.
  - The location of the server (IP address and port number) and the file handle for the remote directory are passed on to the VFS module and the NFS client.
- On each server, there is a file with a well-known name (/etc/exports) containing the names of local filesystems that are available for remote mounting.
- Server maintains a table of clients who have mounted filesystems at that server
- Each client maintains a table of mounted file systems holding:

  < IP address, port number, file handle>

Local and Remote File Systems



NFS Server Caching

- Similar to UNIX file caching for local files:
  - Pages (blocks) from disk are held in a main memory buffer cache until the space is required for newer pages. Read-ahead and delayed-write optimizations.
  - For local files, writes are deferred to next sync event (30 second intervals).
  - Works well in local context, where files are always accessed through the local cache, but in the remote case it doesn't offer necessary synchronization guarantees to clients.
- NFS v3 servers offers two strategies for updating the disk:
  - *write-through* - altered pages are written to disk as soon as they are received at the server. When a *write()* RPC returns, the NFS client knows that the page is on the disk.
  - *delayed commit* - pages are held only in the cache until a *commit()* call is received for the relevant file. This is the default mode used by NFS v3 clients. A *commit()* is issued by the client whenever a file is closed.

NFS Client Caching

- Server caching does nothing to reduce traffic between client and server
  - further optimization is essential to reduce server load in large networks
  - NFS client module caches the results of *read*, *write*, *getattr*, *lookup* and *readdir* operations
  - **synchronization of file contents (*one-copy semantics*) is not guaranteed when two or more clients are sharing the same file.**
- Timestamp-based validity check
  - reduces inconsistency, but doesn't eliminate it
  - validity condition for cache entries at the client:

    $(T - Tc < t) \vee (Tm_{client} = Tm_{server})$
  - t is configurable (per file) but is typically set to 3 seconds for files and 30 secs. for directories
  - it remains difficult to write distributed applications that share files with NFS

Automounter

- NFS client catches attempts to access 'empty' mount points and routes them to the Automounter
  - Automounter has a table of mount points and multiple candidate serves for each
  - It sends a probe message to each candidate server and then uses the mount service to mount the filesystem at the first server to respond
- Keeps the mount table small
- Provides a simple form of replication for read-only filesystems
  - E.g. if there are several servers with identical copies of /usr/lib then each server will have a chance of being mounted at some clients.

NFS Summary

- **Access**: Excellent, the API is the UNIX system call interface for both local and remote files.
- **Location**: Not guaranteed but normally achieved; naming of filesystems is controlled by client mount operations, but transparency can be ensured by an appropriate system configuration.
- **Concurrency**: Limited but adequate for most purposes; when read-write files are shared concurrently between clients, consistency is not perfect.
- **Replication**: Limited to read-only file systems; for writable files, the SUN Network Information Service (NIS) runs over NFS and is used to replicate essential system files.

NFS Summary

- **Failure**: Limited but effective; service is suspended if a server fails. Recovery from failures is aided by the simple stateless design.
- **Mobility**: Hardly achieved; relocation of files is not possible, relocation of filesystems is possible, but requires updates to client configurations.
- **Performance**: Good; multiprocessor servers achieve very high performance, but for a single filesystem it's not possible to go beyond the throughput of a multiprocessor server.
- **Scaling**: Good; filesystems (file groups) may be subdivided and allocated to separate servers. Ultimately, the performance limit is determined by the load on the server holding the most heavily-used filesystem (file group).

3: Andrew File System

Andrew File System (AFS)

- Developed by CMU as part of the Andrew Project. It is named for Andrew Carnegie and Andrew Mellon
- Supported and developed as a product by Transarc Corporation (now IBM Pittsburgh Labs).
- IBM branched the source of the AFS product, and made a copy of the source available for community development and maintenance, the release is called OpenAFS.

Andrew File System (AFS)

- Provides access to remote shared files for UNIX, compatible with NFS
- Design motivation:
  1. Most file accesses are by a single user
  2. Most files are small
  3. Even a client cache as “large” as 100MB is supportable (e.g., in RAM)
  4. File reads are much more often that file writes, and typically sequential not random
  5. Files are accessed in bursts

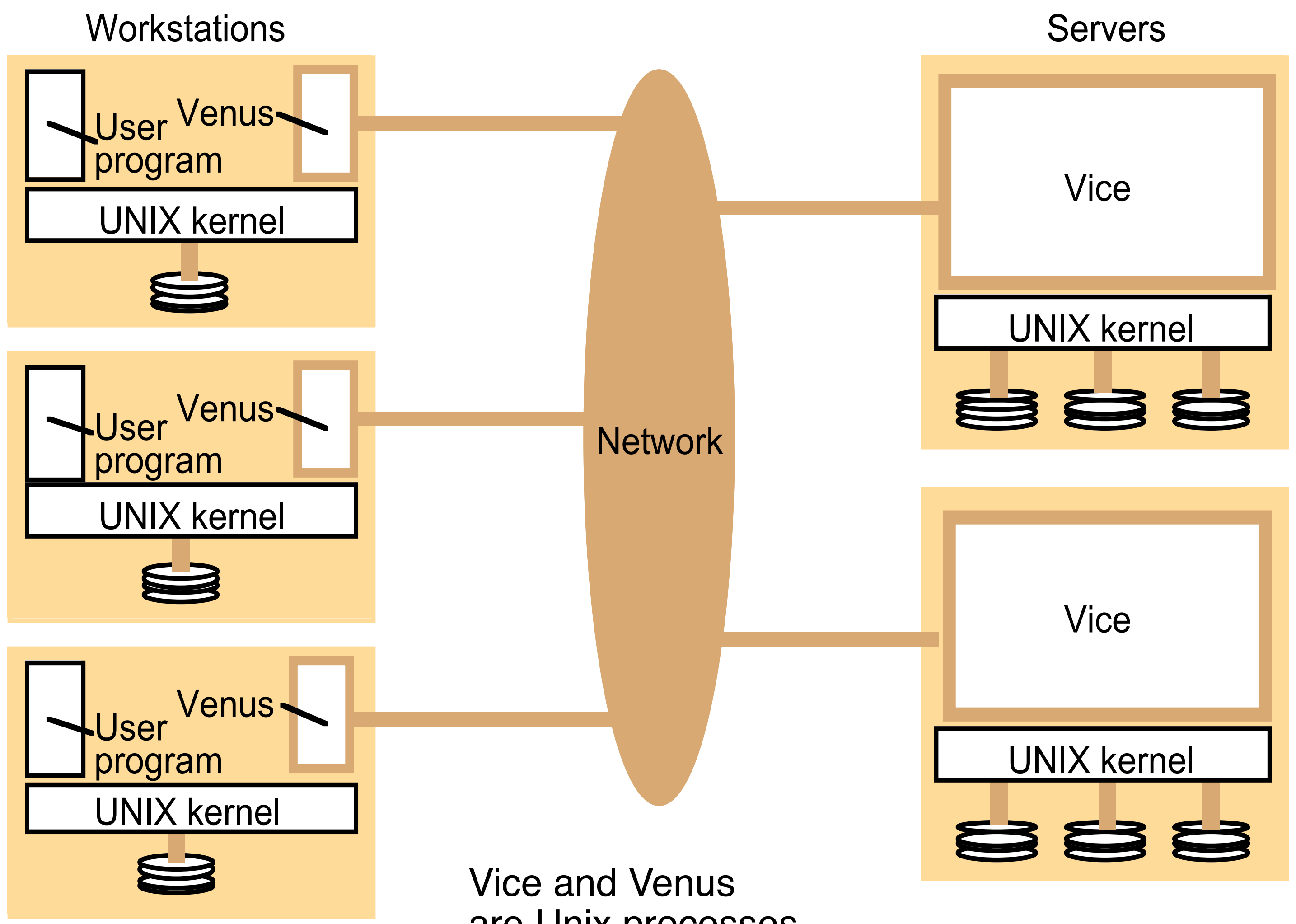
Design Characteristics

- Design and implementation supports information sharing on a large scale (thousands of workstations).
- **Whole-file serving**: entire contents of directories and files are transmitted to client computers by AFS servers
- **Whole-file caching**: once a copy of a file or a chunk has been transferred to a client computer it is stored in a cache or local disk;
  - Cache is permanent (survives reboots) and contains several hundred of the files most recently used from that computer.

Example

- User in a client computer issues an **open** system call:
  - there is no current copy of the file in the local cache
  - the server holding the file is located and is sent a request for a copy of the file
- The copy is stored in the local UNIX file system in the client computer, opened and the corresponding file descriptor is returned to the client
- Subsequent operations on the file by processes in the client computer are applied to the local copy
- User in the client computer issues a **close** system call:
  - If the local copy has been updated its contents are sent back to the server. The server updates the file contents and the timestamps on the file.
  - The copy on the client's local disk us retained in case it is needed again by a user-level process on the same computer

AFS Architecture

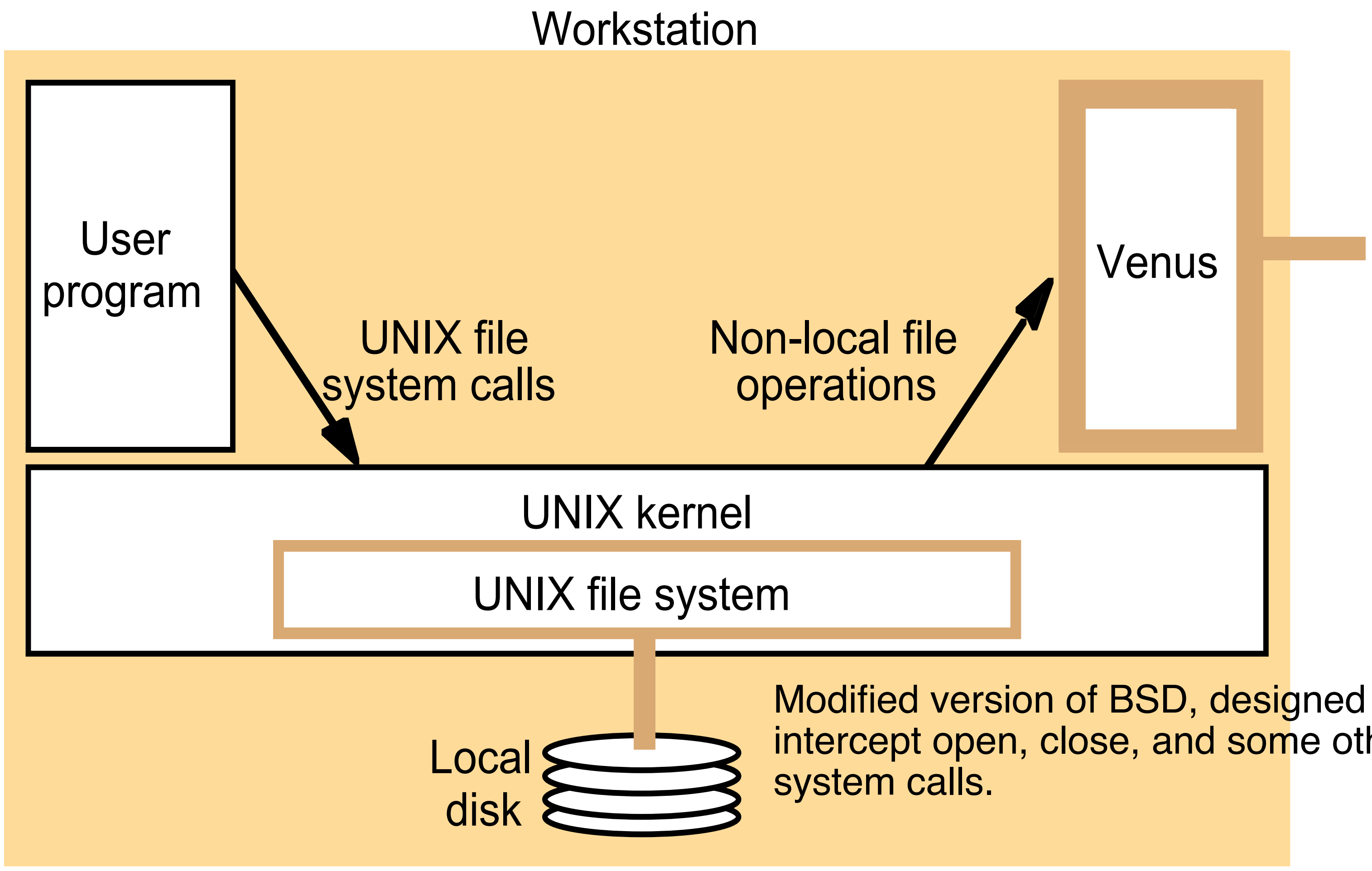


Vice and Venus
are Unix processes

AFS Architecture

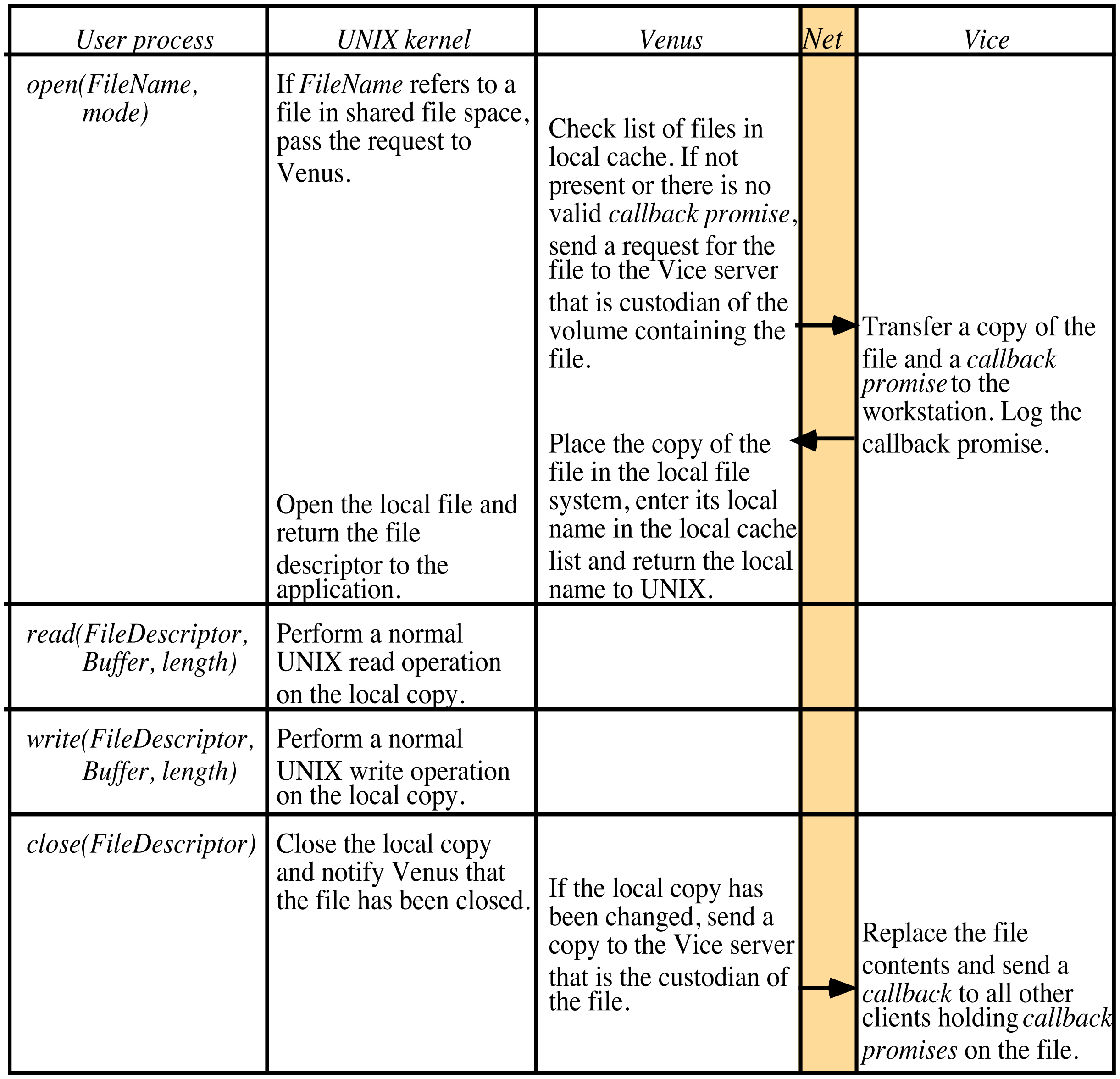
- Clients have a partitioned space of file names:
  a **local name space** and a **shared name space**
- Dedicated servers, called *Vice*, present the shared name space to the clients as an homogeneous, identical, and location transparent file hierarchy
- Workstations, called *Venus*, run the *Virtue* protocol to communicate with Vice.
- Are required to have local disks where they store their local name space
- Servers collectively are responsible for the storage and management of the shared name space

System Call Interception in AFS



Modified version of BSD, designed to intercept open, close, and some other file system calls.

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. 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 *callback promise*, 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.	$\rightarrow$ $\leftarrow$	Transfer a copy of the file and a *callback promise* to the workstation. Log the callback promise.
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.	$\rightarrow$	Replace the file contents and send a *callback* to all other clients holding *callback promises* on the file.

AFS Security

- AFS uses Kerberos for authentication, and implements access control lists on directories for users and groups.

4: CODA

CODA

- Coda is a Distributed File System developed as a research project at Carnegie Mellon University since 1987, descended directly from an older version of AFS (AFS-2)
- Features
  - disconnected operation for mobile computing
  - high performance through client side persistent caching
  - <u>server replication</u>
  - security model for authentication, encryption and access control
  - continued operation during partial network failures in server network
  - network bandwidth adaptation
  - good scalability
  - <u>well defined semantics of sharing, even in the presence of network failures</u>

Access in the Presence of Failures

- Normal operation: a user reads and writes to the file system, the client fetches the data the user wants in the event of network disconnection.
- Network connection lost: the client's local cache serves data from this cache and logs all updates.
- Network reconnection: client transitions from disconnected operation to a transient "reintegration" state where logged updates are sent back to the servers. When all updates are reintegrated, the client transitions back to normal operation mode.

Replication

- AFS: one read/write server receive updates and all other servers act as read-only replicas. Can not handle network partitions.
- CODA: all servers can receive updates, greater availability in the event of network partitions.
- **Local/global conflict**: While disconnected the local updates can potentially clash with other users' updates on the same objects.
- **Server/server conflict**: Optimistic replication can potentially cause concurrent updates to different servers on the same object.
- Coda has extensive repair tools, (manual and automated), to handle and repair conflicts.