Distributed-File Systems

Guihai Chen
Department of Computer Science and Engineering
Shanghai Jiao Tong University
Spring 2020

Distributed-File Systems

Outline of Contents

- Background
- Naming and Transparency
- Remote File Access
- Stateful versus Stateless Service
- File Replication
- An Example: AFS

Red color stuff are appended by Guihai Chen

Chapter Objectives

- To explain the naming mechanism that provides location transparency and independence
- To describe the various methods for accessing distributed files
- To contrast stateful and stateless distributed file servers
- To show how replication of files on different machines in a distributed file system is a useful redundancy for improving availability
- To introduce the Andrew file system (AFS) as an example of a distributed file system

The Ultimate goal is to make DFS look as if it is a conventional file system.

Background

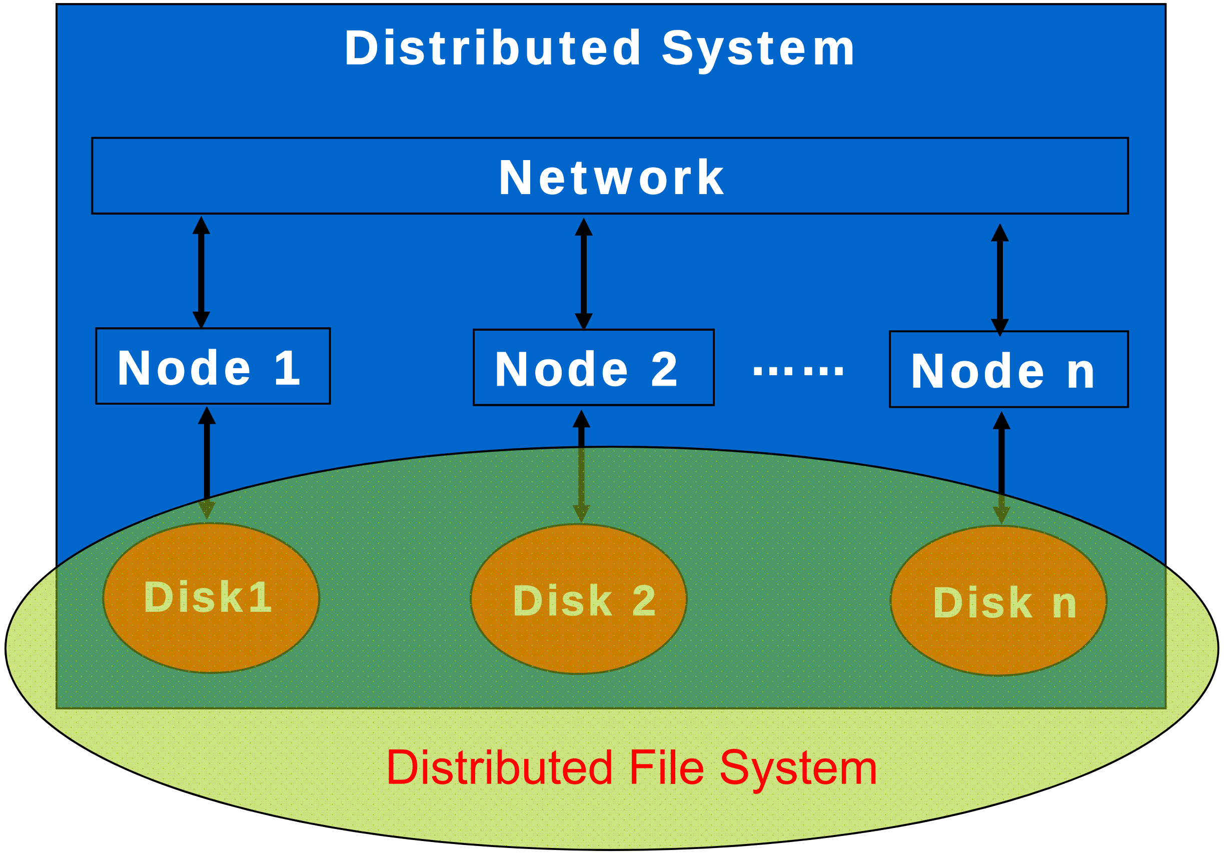
- **Distributed file system** (**DFS**) – a distributed implementation of the classical time-sharing model of a file system, where multiple users share files and storage resources

- A DFS manages set of dispersed storage devices

- Overall storage space managed by a DFS is composed of different, remotely located, smaller storage spaces

- There is usually a correspondence between constituent storage spaces and sets of files

Background (Cont.)


Distributed System
Network
Node 1
Node 2
……
Node n
Disk1
Disk 2
Disk n
Distributed File System

Disk(1~n)➔ many component units ➔ A file system

DFS Structure

- **Service** – software entity running on one or more machines and providing a particular type of function to a priori unknown clients
- **Server** – service software running on a single machine
- **Client** – process that can invoke a service using a set of operations that forms its client interface
- A client interface for a file service is formed by a set of primitive file operations (create, delete, read, write, see next slides)
- Client interface of a DFS should be transparent, i.e., not distinguish between local and remote files

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*.

Naming and Transparency

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

- **Multilevel mapping** – abstraction of a file that hides the details of how and where on the disk the file is actually stored
  - Name to identifier
  - Identifier to blocks and locations (inode)
  - More complicated if there are many replicas of one file

- 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

Naming Structures

- **Location transparency** – file name does not reveal the file's physical storage location

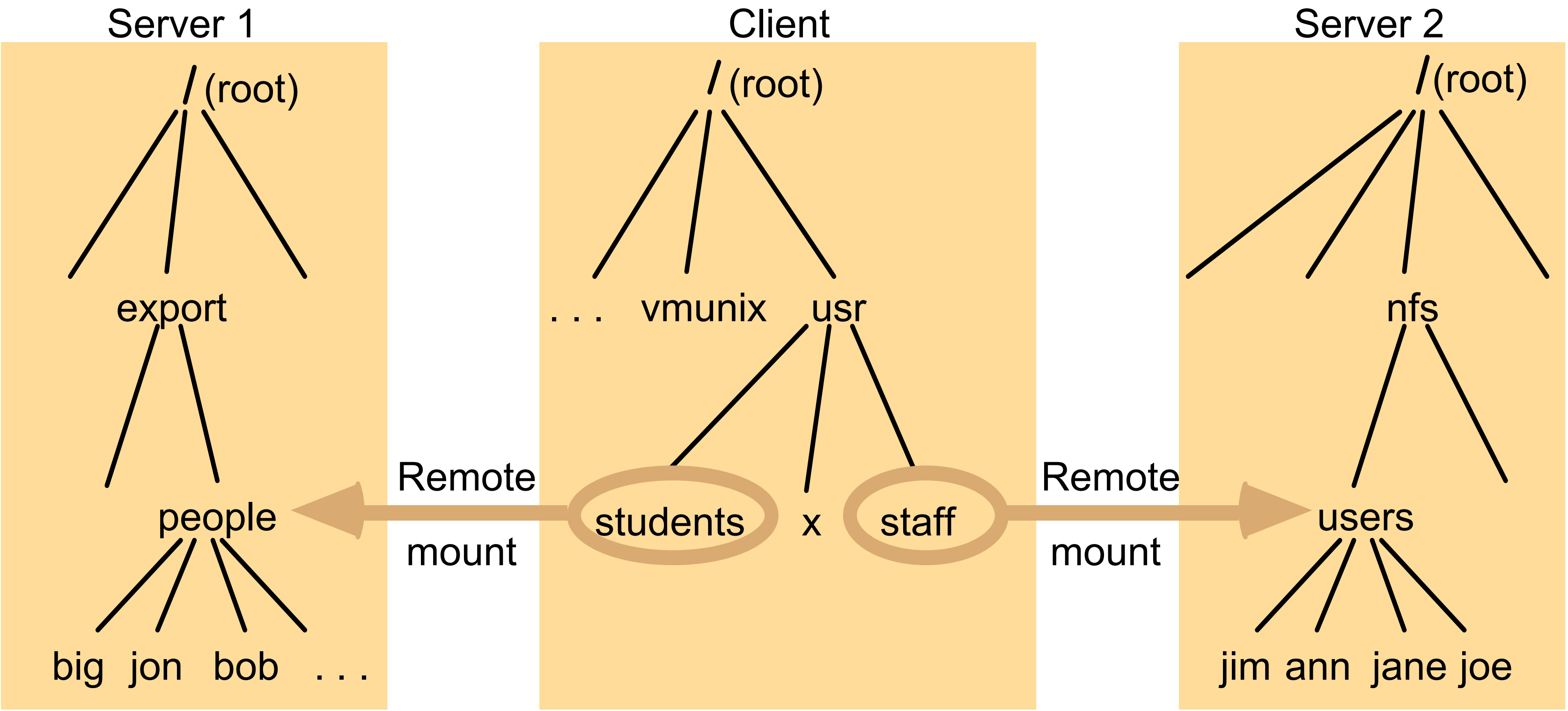
- **Location independence** – file name does not need to be changed when the file's physical storage location changes

- Location independence is a stronger requirement than Location transparency
  - Most DFSs only support Location transparency , but AFS meets both requirements

Naming Schemes — Three Main Approaches

- Files named by combination of their host name and local name; guarantees a unique system-wide name
  - It supports neither Location independence nor Location transparency.
- Attach remote directories to local directories, giving the appearance of a coherent directory tree; only previously mounted remote directories can be accessed transparently
  - UNIX is an example. See next slide.
- 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

Local and Remote File Systems Accessible on An NFS Client


Server 1
/ (root)
export
people
big
jon
bob
. . .
Remote
mount
Client
/ (root)
. . .
vmunix
usr
students
x
staff
Remote
mount
Server 2
/ (root)
nfs
users
jim
ann
jane
joe

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.

Remote File Access

- **Remote-service mechanism** is one transfer approach
  - Use RPC to support remote file access.(see 3.6.2)
- 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 (parts of) the file are scattered in different caches
  - **Cache-consistency problem** – keeping the cached copies consistent with the master file
    - Could be called **network virtual memory**

Cache Location – Disk vs. Main Memory

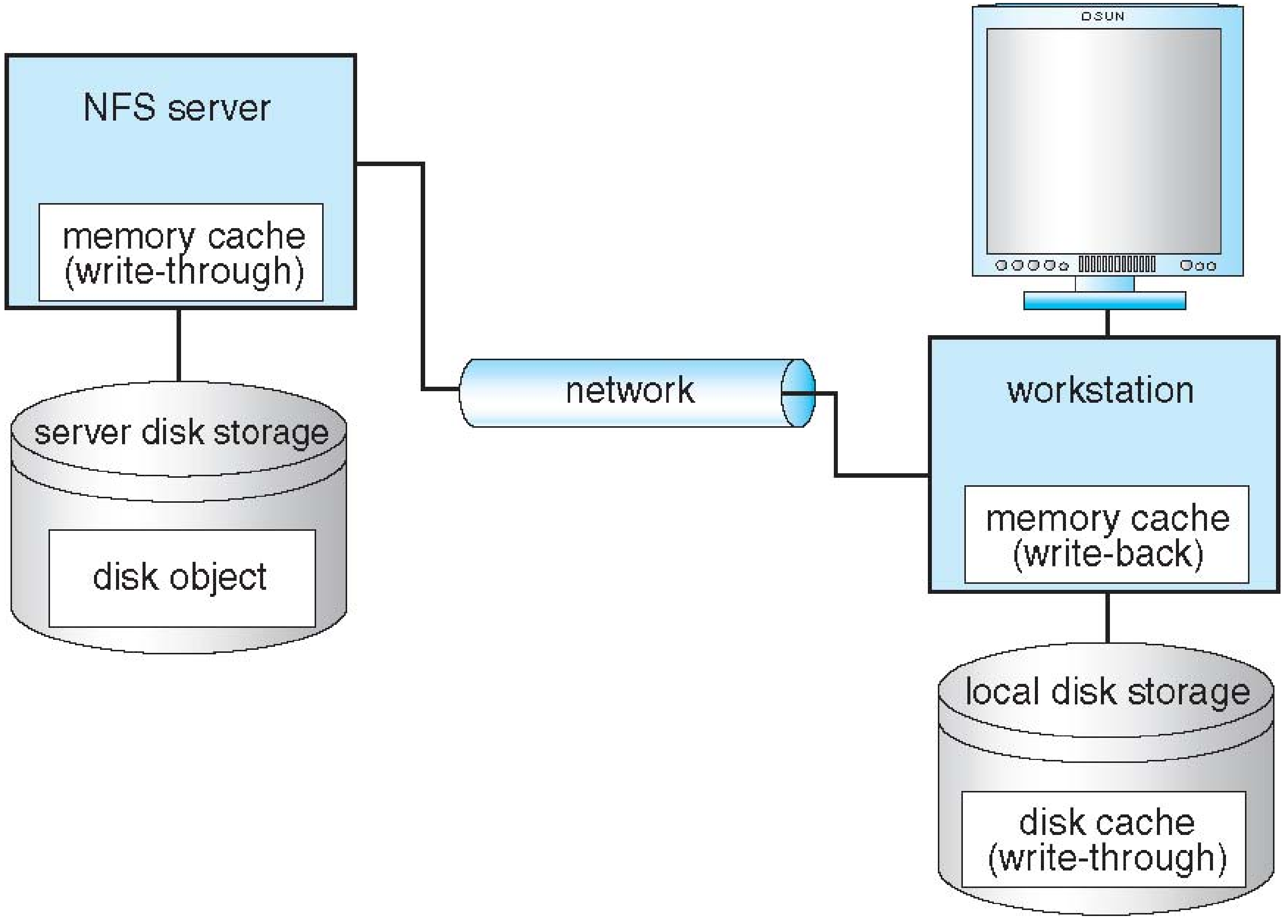
- Advantages of disk caches
  - More reliable
  - Cached data kept on disk are still there during recovery and don't need to be fetched again

- Advantages of main-memory caches:
  - Permit workstations to be diskless
  - Data can be accessed more quickly
  - Performance speedup in bigger memories
  - 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

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** (a.k.a.Write Back)– 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
    - Used in AFS

CacheFS and its Use of Caching


NFS server
memory cache (write-through)
server disk storage
disk object
network
SUN
workstation
memory cache (write-back)
local disk storage
disk cache (write-through)

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
  - Tradeoff between validity check and access performance

- **Server-initiated approach**
  - Server records, for each client, the (parts of) files it caches
  - When server detects a potential inconsistency, it must react

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 contacted 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)

Caching and Remote Service (Cont.)

- Caching is superior in access patterns with infrequent writes
  - With frequent writes, substantial overhead incurred to overcome cache-consistency problem
- Benefit from caching when execution carried out on machines with either local disks or large main memories
- Remote access on diskless, small-memory-capacity machines should be done through remote-service method
- In caching, the lower intermachine interface is different from the upper user interface
- In remote-service, the intermachine interface mirrors the local user-file-system interface

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

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

Distinctions Between Stateful and 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 failure and recovery are almost unnoticeable
    - A newly reincarnated server can respond to a self-contained request without any difficulty

Distinctions (Cont.)

- Penalties for using the robust stateless service:
  - longer request messages
  - slower request processing
  - additional constraints imposed on DFS design

- 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

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
  - But sometimes file replication mechanism is exposed to users for performance purpose like Lucas
- Updates – replicas of a file denote the same logical entity, and thus an update to any replica must be reflected on all other replicas
- Demand replication – reading a nonlocal replica causes it to be cached locally, thereby generating a new nonprimary replica

An Example: AFS

- A distributed computing environment (Andrew) under development since 1983 at Carnegie-Mellon University, purchased by IBM and released as **Transarc DFS**, now open sourced as OpenAFS
- AFS tries to solve complex issues such as uniform name space, location-independent file sharing, client-side caching (with cache consistency), secure authentication (via Kerberos)
  - Also includes server-side caching (via replicas), high availability
  - Can span 5,000 workstations
- See which choice is taken for each mechanism
  - Naming
  - Remote file access
  - Caching
  - consistency

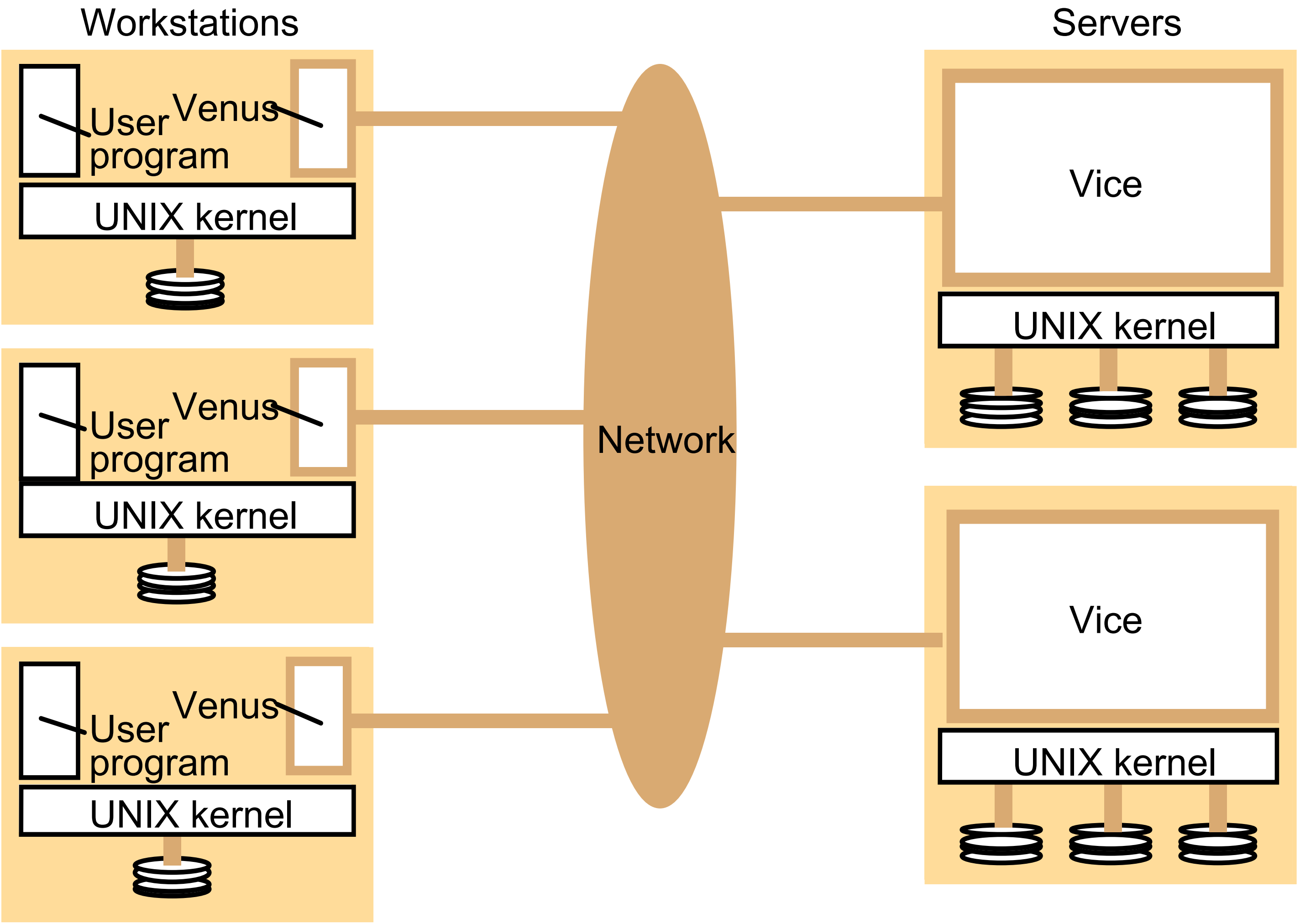
ANDREW (Cont.)

- Clients are presented with 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
- The local name space is the root file system of a workstation, from which the shared name space descends
- Workstations run the *Virtue* protocol to communicate with Vice, and 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

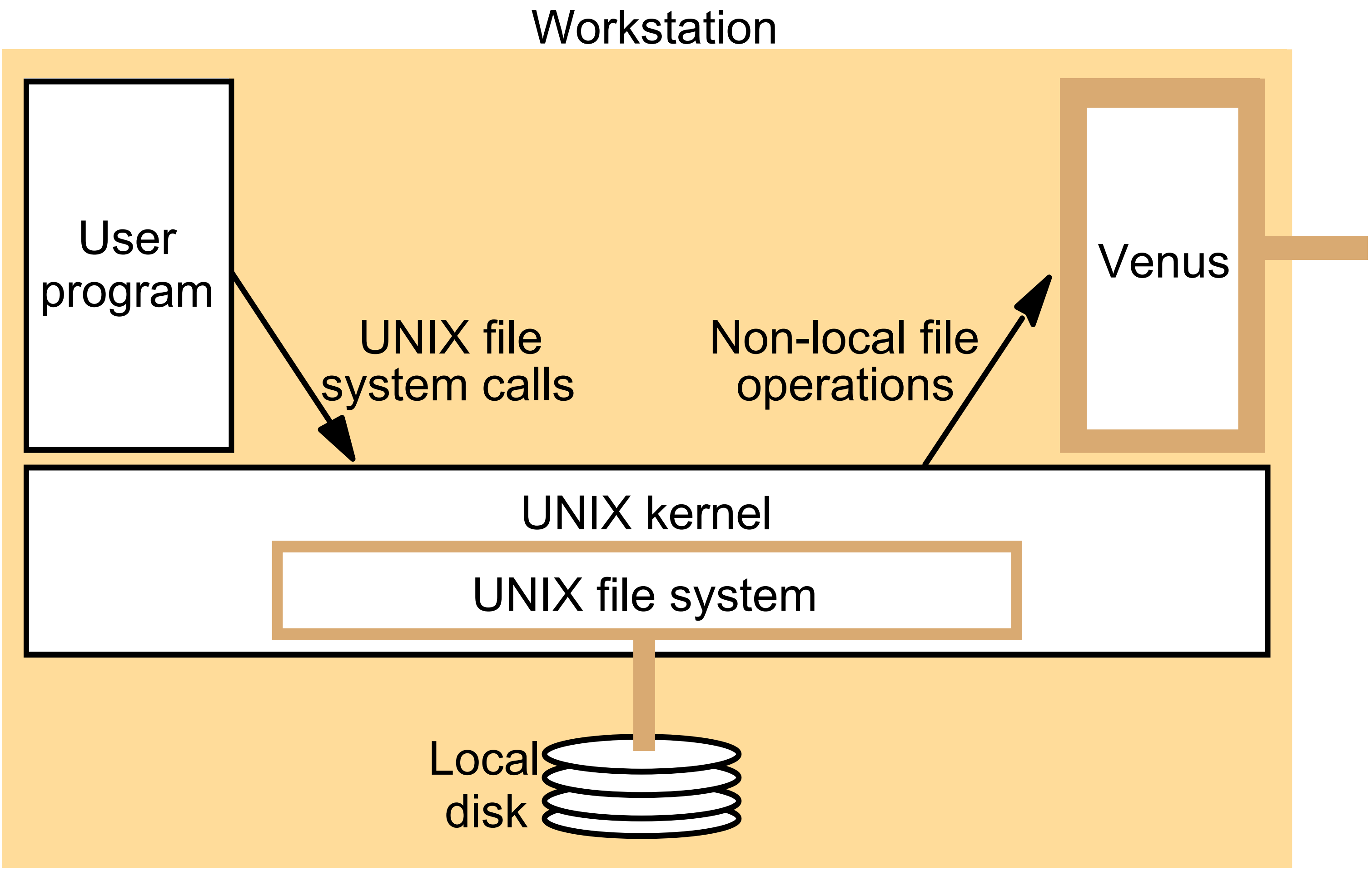
ANDREW (Cont.)

- Clients and servers are structured in clusters interconnected by a backbone LAN
- A cluster consists of a collection of workstations and a cluster server and is connected to the backbone by a router
- A key mechanism selected for remote file operations is whole file caching
  - Opening a file causes it to be cached, in its entirety, on the local disk

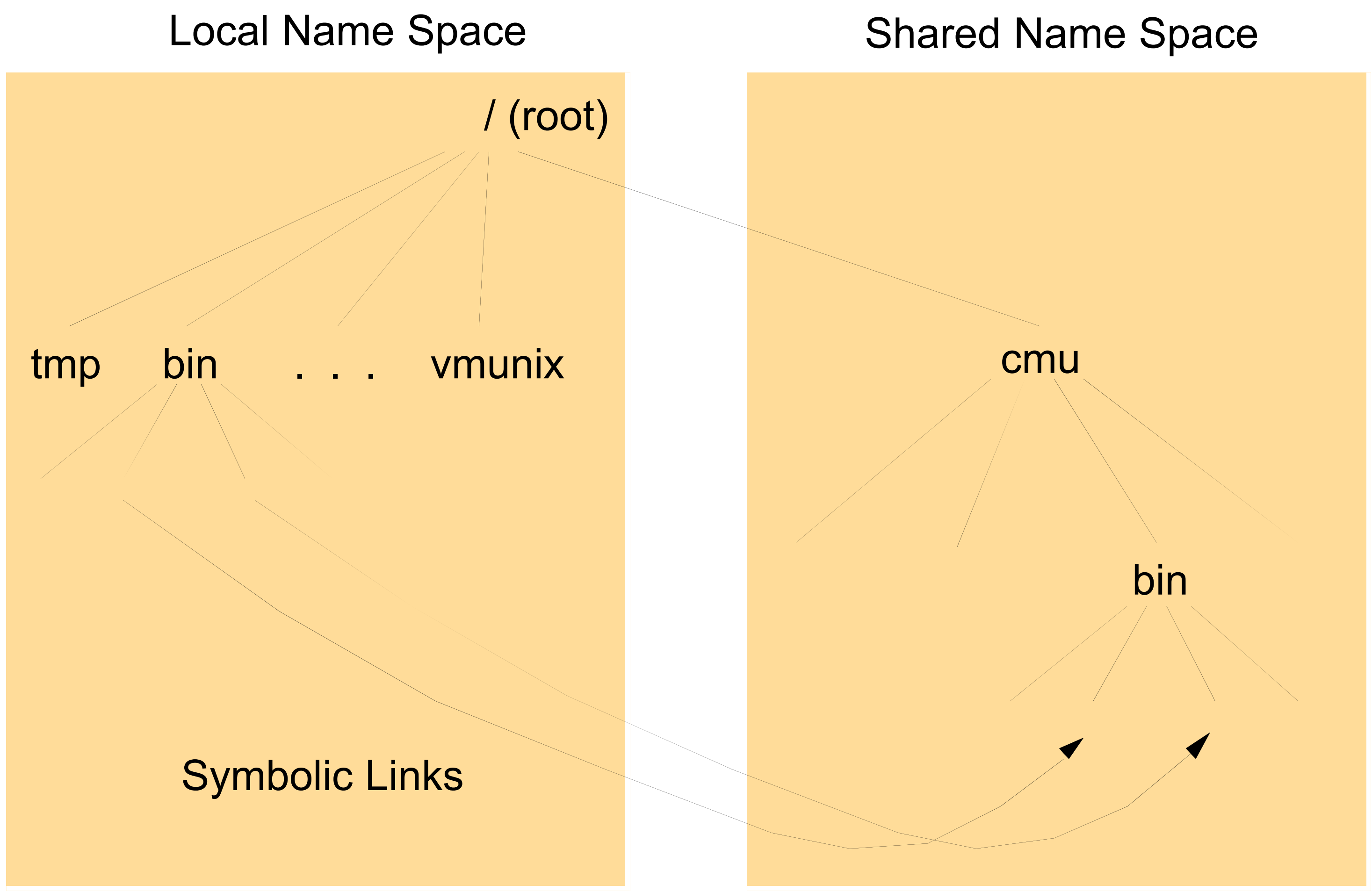
Distribution of Processes in the Andrew File System


Workstations
Servers
User program
Venus
UNIX kernel
User program
Venus
UNIX kernel
User program
Venus
UNIX kernel
Network
Vice
UNIX kernel
Vice
UNIX kernel

System Call Interception in AFS


Workstation
User program
UNIX file system calls
Non-local file operations
Venus
UNIX kernel
UNIX file system
Local disk

File Name Space Seen by Clients of AFS


Local Name Space
Shared Name Space
/ (root)
tmp
bin
. . .
vmunix
cmu
bin
Symbolic Links

ANDREW Shared Name Space

- Andrew's **volumes** are small component units associated with the files of a single client

- A **fid** identifies a Vice file or directory - A fid is 96 bits long and has three equal-length components:
  - volume number
  - **vnode number** – index into an array containing the inodes of files in a single volume
  - **uniquifier** – allows reuse of vnode numbers, thereby keeping certain data structures, compact

- Fids are location transparent; therefore, file movements from server to server do not invalidate cached directory contents

- Location information is kept on a volume basis, and the information is replicated on each server

ANDREW File Operations

- Andrew caches entire files from servers
  - A client workstation interacts with Vice servers only during opening and closing of files
- *Venus* – caches files from Vice when they are opened, and stores modified copies of files back when they are closed
- Reading and writing bytes of a file are done by the kernel without Venus intervention on the cached copy
- Venus caches contents of directories and symbolic links, for path-name translation
- Exceptions to the caching policy are modifications to directories that are made directly on the server responsibility for that directory

ANDREW Implementation

- Client processes are interfaced to a UNIX kernel with the usual set of system calls
- Venus carries out path-name translation component by component
- The UNIX file system is used as a low-level storage system for both servers and clients
  - The client cache is a local directory on the workstation’s disk
- Both Venus and server processes access UNIX files directly by their inodes to avoid the expensive path name-to-inode translation routine

ANDREW Implementation (Cont.)

- Venus manages two separate caches:
  - one for status
  - one for data
- LRU algorithm used to keep each of them bounded in size
- The status cache is kept in virtual memory to allow rapid servicing of `stat()` (file status returning) system calls
- The data cache is resident on the local disk, but the UNIX I/O buffering mechanism does some caching of the disk blocks in memory that are transparent to Venus

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. 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.	→ ←	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.	→	Replace the file contents and send a *callback* to all other clients holding *callback promises* on the file.

Homework

- Reading
    - Chapter 19