Computer Science 425
Distributed Systems

CS 425 / ECE 428

Fall 2013

Indranil Gupta (Indy)

December 3, 2013

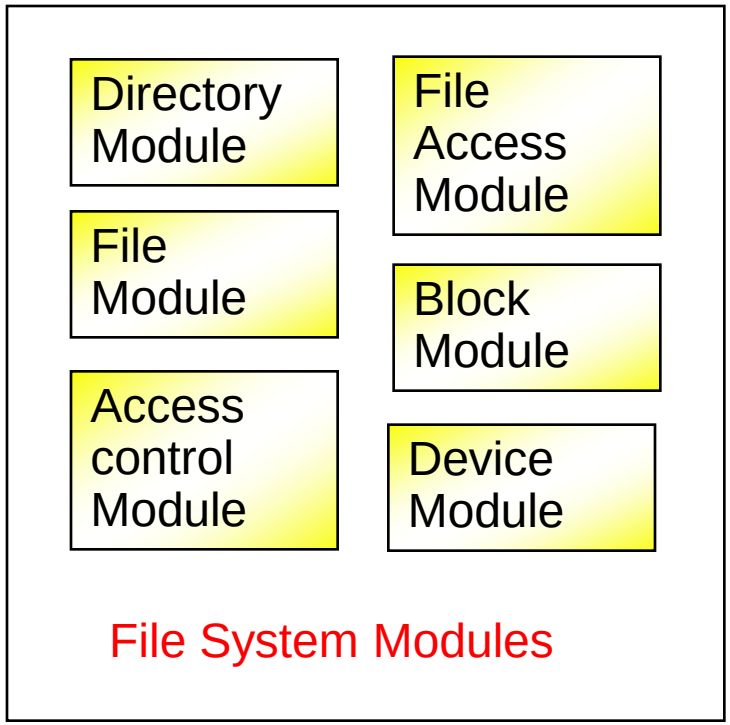
Lecture 27

Distributed File Systems

Chapter 12 (relevant parts)

File Attributes & System Modules

- length
- creation timestamp
- read timestamp
- write timestamp
- attribute timestamp
- reference count
- file type
- ownership
- access control list



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 links to it, it is deleted from disk.
status = link(name1, name2)	Creates a new link (*name2*) for a file (*name1*).
status = stat(name, buffer)	Gets the file attributes for file *name* into *buffer*.

Distributed File System (DFS) Requirements

- **Transparency : server-side changes should be invisible to the client-side.**
  - ***Access transparency*: A single set of operations is provided for access to local/remote files.**
  - ***Location Transparency:* All client processes see a uniform file name space.**
  - ***Migration Transparency*: When files are moved from one server to another, users should not see it.**
  - ***Scaling and Performance Transparency***
- **File Replication**
  - **A file may be represented by several copies for read/write efficiency and fault tolerance.**
- **Concurrent File Updates**
  - **Changes to a file by one client should not interfere with the operation of other clients simultaneously accessing the same file.**

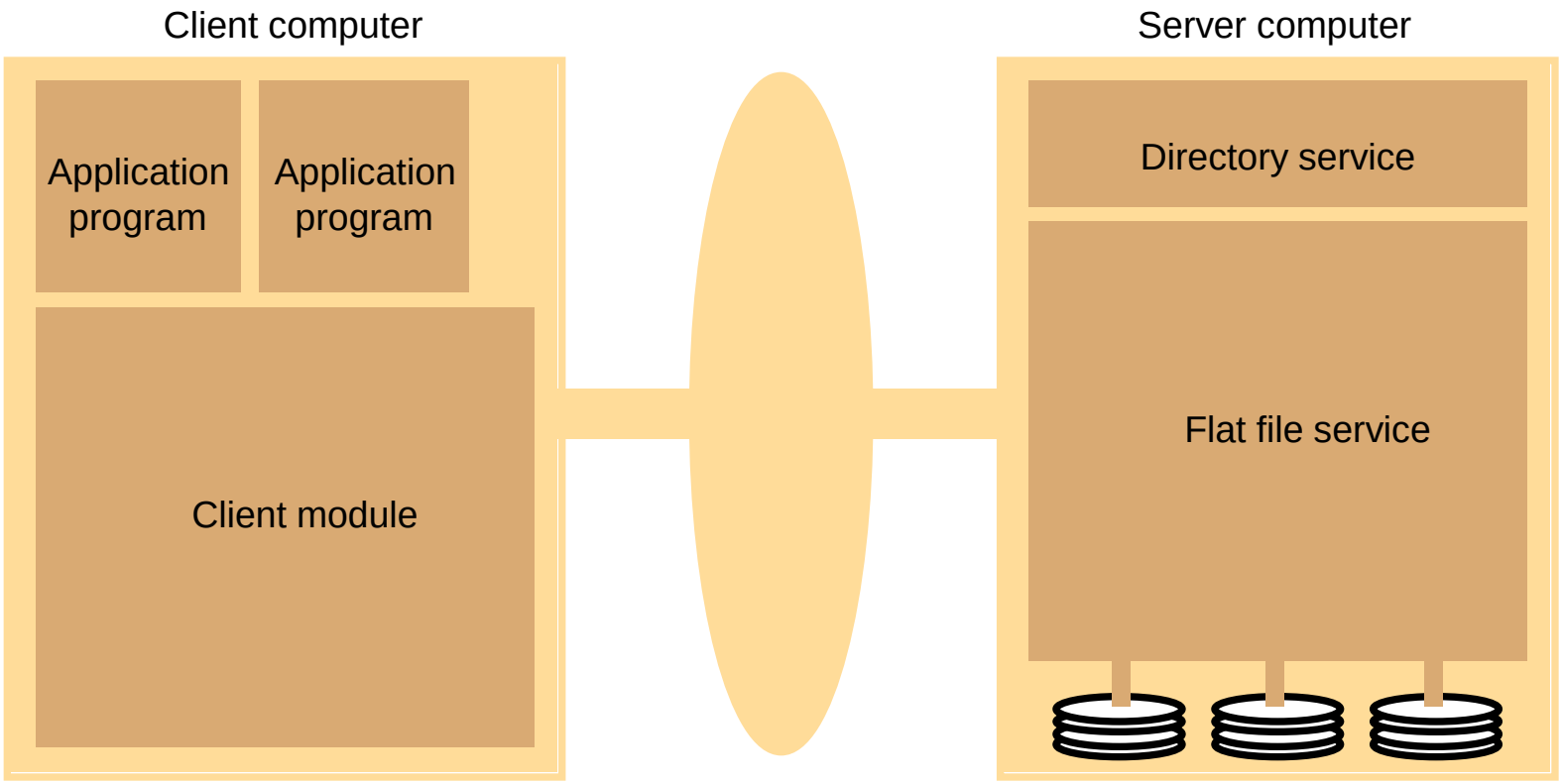
DFS Requirements (2)

- **Concurrent File Updates**
  - **One-copy update semantics: the file contents seen by all of the clients accessing or updating a given file are those they would see if only a single copy of the file existed.**
- **Fault Tolerance**
  - **At most once invocation semantics, e.g., append to file**
  - **At least once semantics. OK for a server protocol designed for idempotent operations (i.e., duplicated requests do not result in invalid updates to files), e.g., read at a position in the file**
- **Security**
  - **Access Control list = per object, list of allowed users and access allowed to each**
  - **Capability list = per user, list of objects allowed to access and type of access allowed (could be different for each (user,obj))**
  - **User Authentication: need to authenticate requesting clients so that access control at the server is based on correct user identifiers.**
- **Efficiency**
  - **Whole file vs. block transfer**

Basic File Service Model

- E.g., SUN NFS (Network File System) and AFS (Andrew File System)
- An abstract model (Our "Vanilla" Model):
  - Flat file service
    - implements create, delete, read, write, get attribute, set attribute and access control operations.
  - Directory service: is itself a client of (i.e., uses) flat file service.
    - Creates and updates directories (hierarchical file structures) and provides mappings between user names of files and the unique file ids in the flat file structure.
  - Client service/module: A client of directory and flat file services
    - Runs in each client computer, integrating and expanding flat file and directory services to provide a unified API (e.g., the full set of UNIX file operations).
    - Holds information about the locations of the flat file server and directory server processes.

File Service Architecture



Flat File Service Operations

Read(FileId, i, n) -> Data — throws*BadPosition*	If $1 \leq i \leq Length(File)$: Reads a sequence of up to *n* items from a file starting at item *i* and returns it in *Data*.
Write(FileId, i, Data) — throws*BadPosition*	If $1 \leq i \leq Length(File)+1$: Writes a sequence of *Data* to a file, starting at item *i*, extending the file size if necessary.
Create() -> FileId	Creates a new file of length 0 and delivers a UFID for it.
Delete(FileId)	Removes the file from the file store.
GetAttributes(FileId)->Attr	Returns the file attributes for the file.
SetAttributes(FileId, Attr)	Sets the file attributes

(1) Repeatable operation: No read-write pointer. Except for Create and delete, the operations are idempotent, allowing the use of at least once RPC semantics.

(2) Stateless servers: No file descriptors. Stateless servers can be restarted after a failure and resume operation without the need to restore any state.

In contrast, the UNIX file operations are neither idempotent nor stateless.

Access Control

- In UNIX, the user's access rights are checked against the access mode requested in the open call and the file is opened only if the user has the appropriate rights.
- In a distributed file system (DFS), a user identity has to be passed with requests – server first authenticates the user.
  - An access check is made whenever a file name is converted to a UFID (unique file id), and the results are encoded in the form of a capability which is returned to the client for future access.
    - » Capability = per (user, obj) list of allowed operations. A signed certificate.

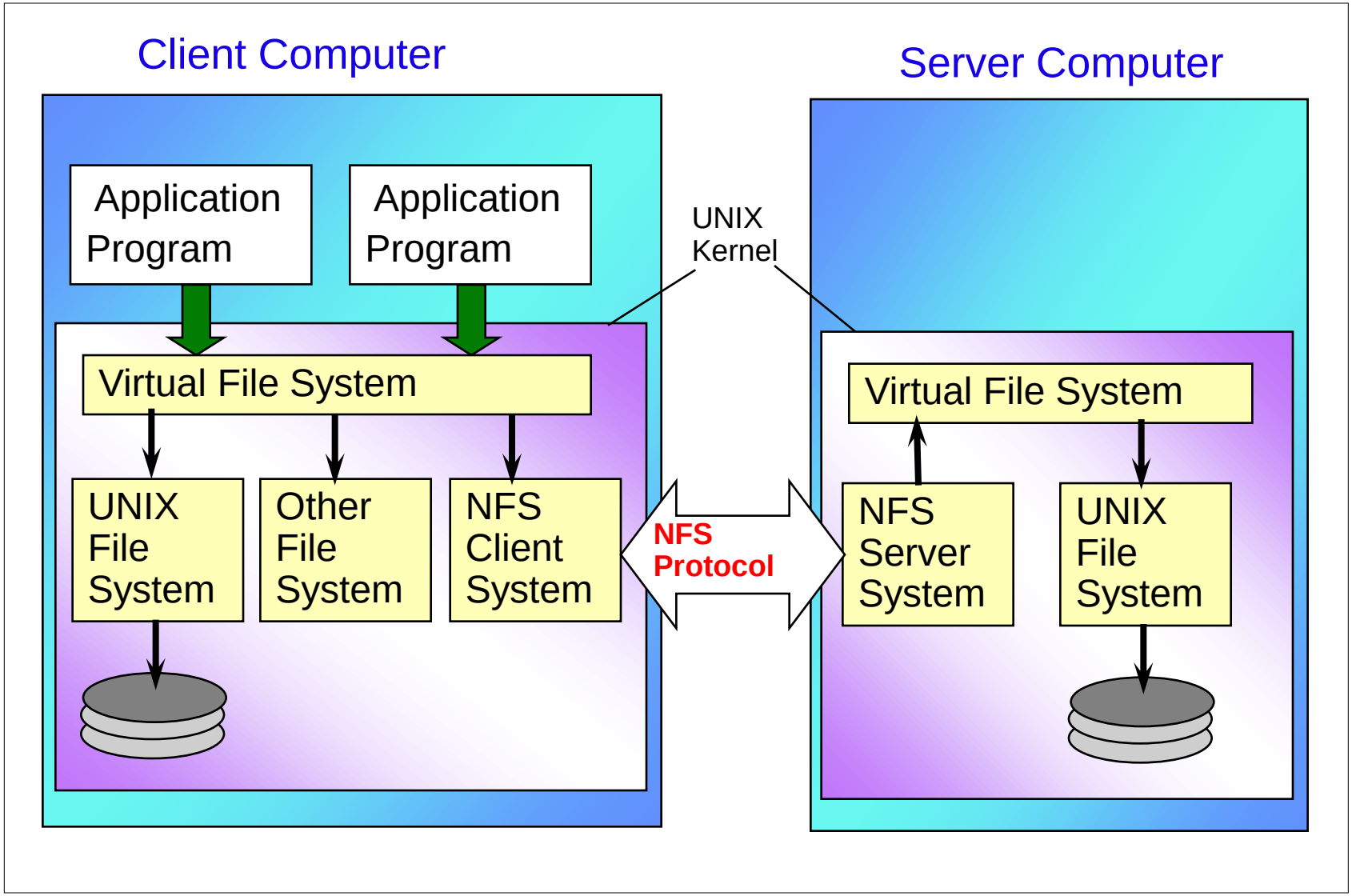
Directory Service Operations

Lookup(Dir, Name) -> FileId — throws *NotFound*	Locates the text name in the directory and returns the relevant UFID. If *Name* is not in the directory, throws an exception.
AddName(Dir, Name, File) — throws *NameDuplicate*	If *Name* is not in the directory, adds (*Name*, *File*) to the directory and updates the file's attribute record. If *Name* is already in the directory: throws an exception.
UnName(Dir, Name) — throws *NotFound*	If *Name* is in the directory: the entry containing *Name* is removed from the directory. If *Name* is not in the directory: throws an exception.
GetNames(Dir, Pattern)->NameSeq	Returns all the text names in the directory that match the regular expression *Pattern*. Like *grep*.

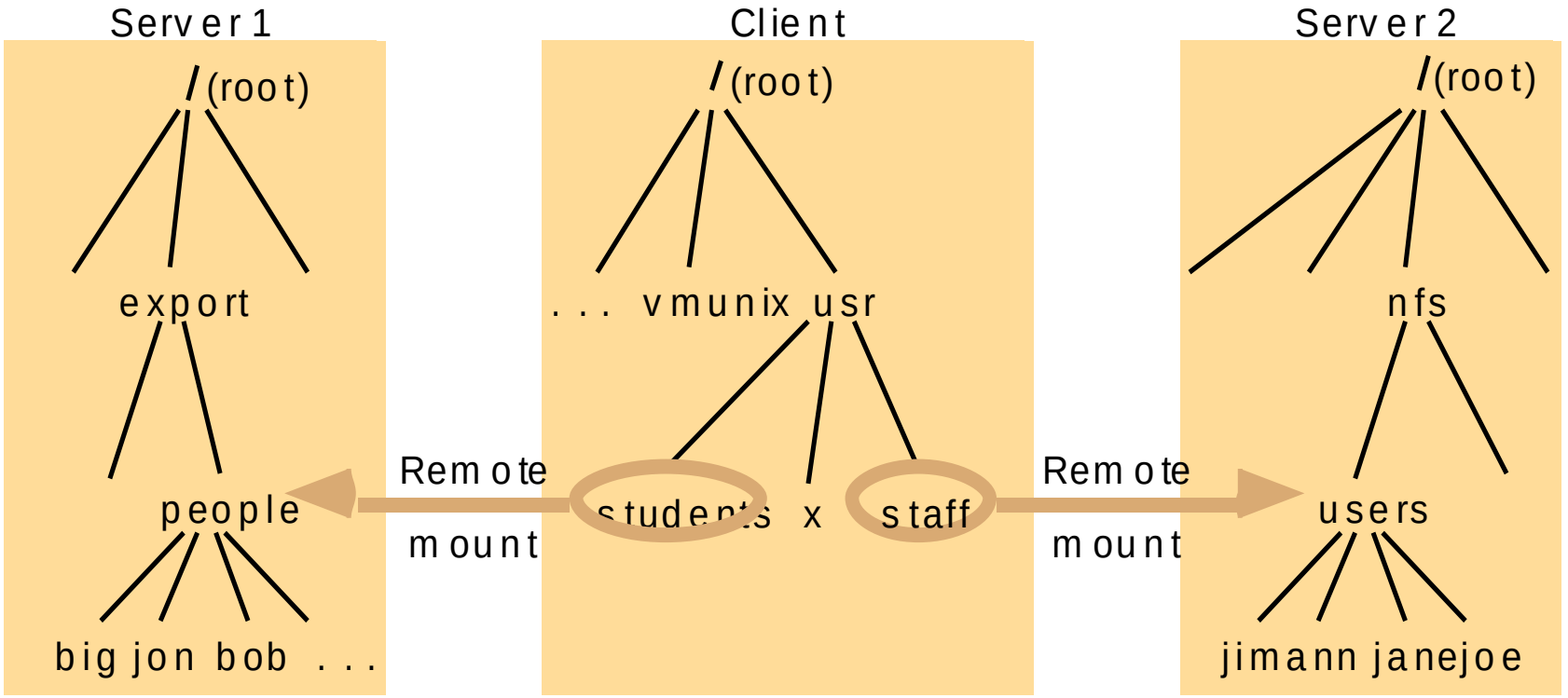
(1) Hierarchical file system: The client module provides a function that gets the UFID of a file given its pathname. The function interprets the pathname starting from the root, using *Lookup* to obtain the UFID of each directory in the path.

(2) Each server may hold several *file groups,* each of which is a collection of files located on the server. A file group identifier consists of IP address + date, and allows (i) file groups to migrate across servers, and (ii) clients to access file groups.

Network File System (NFS)



Local and Remote File Systems Accessible on an NFS client



Note: The filesystem 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.

Hard mounting (retry f.s. request on failure) vs. Soft mounting (return error on f.s. access failure) – Unix is more compatible with hard mounting

NFS Client and Server

- **Client**
  - **Plays the role of the client module from our vanilla model.**
  - **Integrated with the kernel, rather than being supplied as a library.**
  - **Transfers blocks of files to and from server via RPC. Caches the blocks in the local memory.**
  - **May support file descriptors**
- **Server**
  - **Provides a conventional RPC interface at a well-known port on each host.**
  - **Plays the role of file and directory service modules in our vanilla model.**
  - **Mounting of sub-trees of remote filesystems by clients is supported by a separate mount service process on each NFS server.**

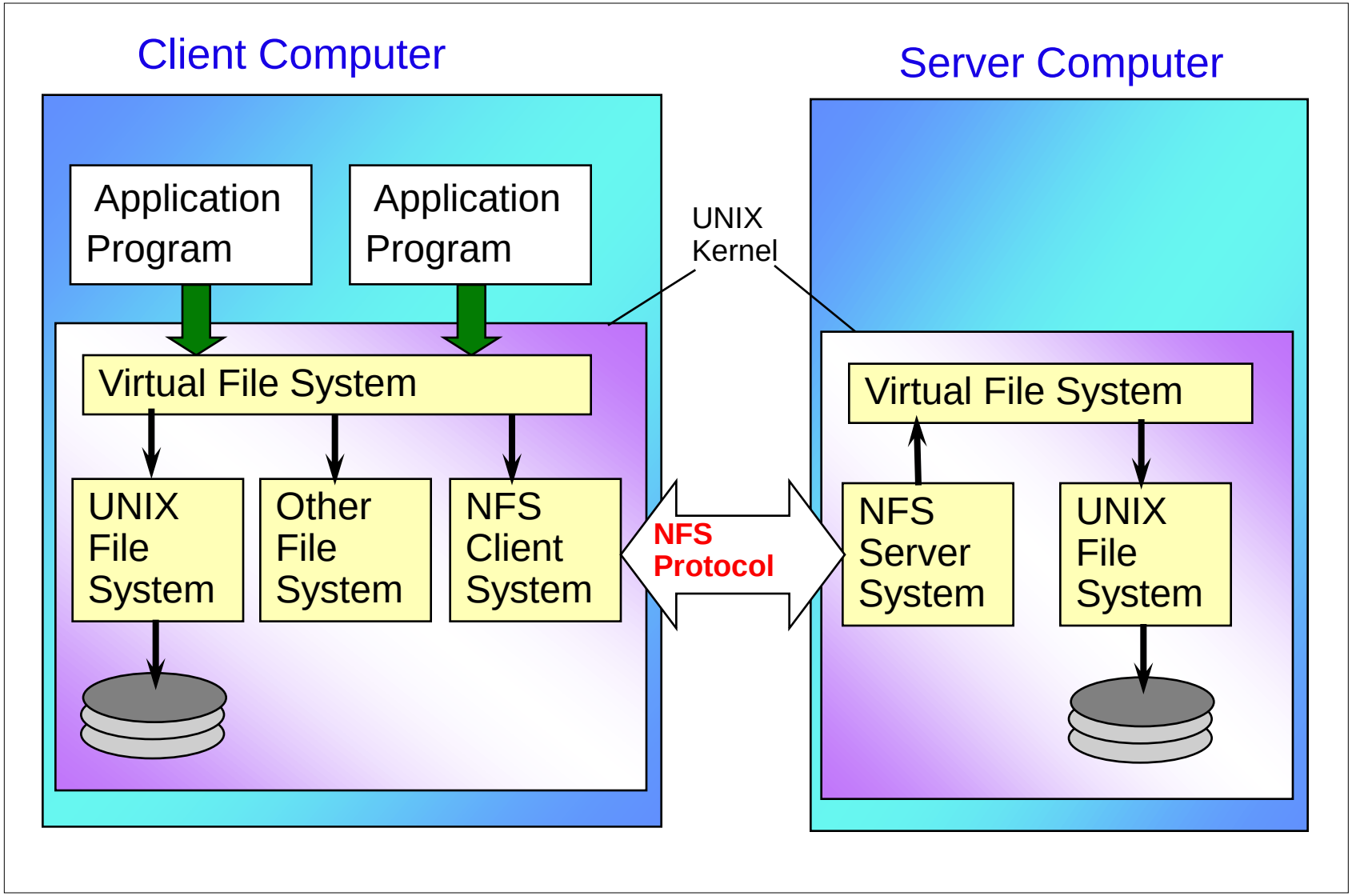
NFS Server Operations (simplified) – 1

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 and 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*.

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* parameter is 0, it 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*.

Network File System (NFS)



NFS Architecture -- VFS

- **Virtual file system module**
  - **Translates between NFS file identifiers and other file systems's (e.g., UNIX) identifiers.**
    - » **The NFS file identifiers are called *file handle*s.**
    - » **File handle = *Filesystem/file group* identifier + *i-node* number of file + i-node generation number.**
  - **Keeps track of filesystems (i.e., NFS file groups, different from a "file system") that are available locally and remotely.**
    - » **The client obtains the first file handle for a remote filesystem when it first *mounts* the filesystem. File handles are passed from server to client in the results of lookup, create, and mkdir operation.**
  - **Distinguishes between local and remote files.**

NFS Architecture – VFS (2)

- **Virtual file system module**
  - **(contd.)**
  - **Distinguishes between local and remote files.**
    - » **VFS keeps one VFS structure for each mounted filesystem and one v-node per open file.**
      - **A VFS structure relates a remote filesystem to the local directory on which it is mounted.**
      - **A v-node contains an indicator to show whether a file is local or remote.**
        - **If the file is local, it contains a reference to the i-node.**
        - **Otherwise if the file is remote, it contains the file handle of the remote file.**

Server Caching

- File pages, directories and file attributes that have been read from the disk are retained in a *main memory buffer cache.*
- *Read-ahead* anticipates read accesses and fetches the pages following those that have most recently been read.
- In *delayed-write*, when a page has been altered, its new contents are written back to the disk only when the buffered page is required for another client.
  - In comparison, Unix *sync* operation writes pages to disk every 30 seconds
- In *write-through,* data in write operations is stored in the memory cache at the server immediately *and* written to disk before a reply is sent to the client.
  - *Better strategy to ensure data integrity even when server crashes occur. But more expensive. (remember CAP theorem?)*

Client Caching

- **A timestamp-based method is used to validate cached blocks before they are used.**
- **Each data item in the cache is tagged with**
  - ***Tc*: the time when the cache entry was last validated.**
  - ***Tm*: the time when the block was last modified at the server.**
  - **A cache entry at time T is valid if**

    $(T-Tc < t)$ or $(Tm_{client} = Tm_{server})$.
  - ***t=freshness interval***
    - » ***Compromise between consistency and efficiency***
    - » ***Sun Solaris: t is set adaptively between 3-30 seconds for files, 30-60 seconds for directories***

Client Caching (Cont'd)

- When a cache entry is read, a validity check is performed.
  - If the first half of validity condition (previous slide) is true, the the second half need not be evaluated.
  - If the first half is not true, Tm_{server} is obtained (via *getattr()* to server) and compared against Tm_{client}
- When a cached page (not the whole file) is modified, it is marked as dirty and scheduled to be flushed to the server.
  - Modified pages are flushed when the file is closed or a *sync* occurs at the client.
- Does not guarantee one-copy update semantics.
- More details in textbook

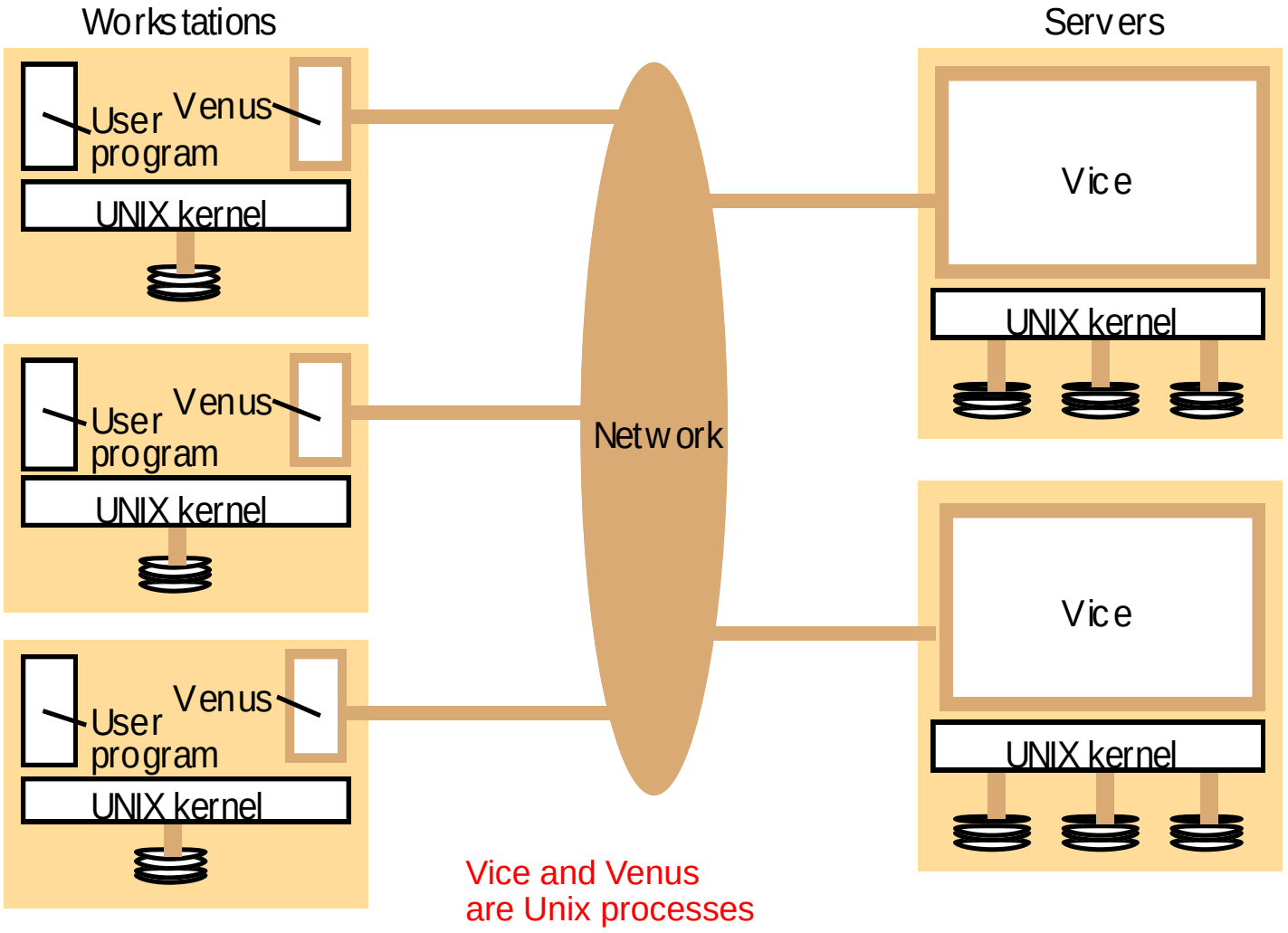
Andrew File System (AFS)

- **Two unusual design principles:**
  - **Whole file serving**
    - » **Not in blocks**
  - **Whole file caching**
    - » **Permanent cache, survives reboots**

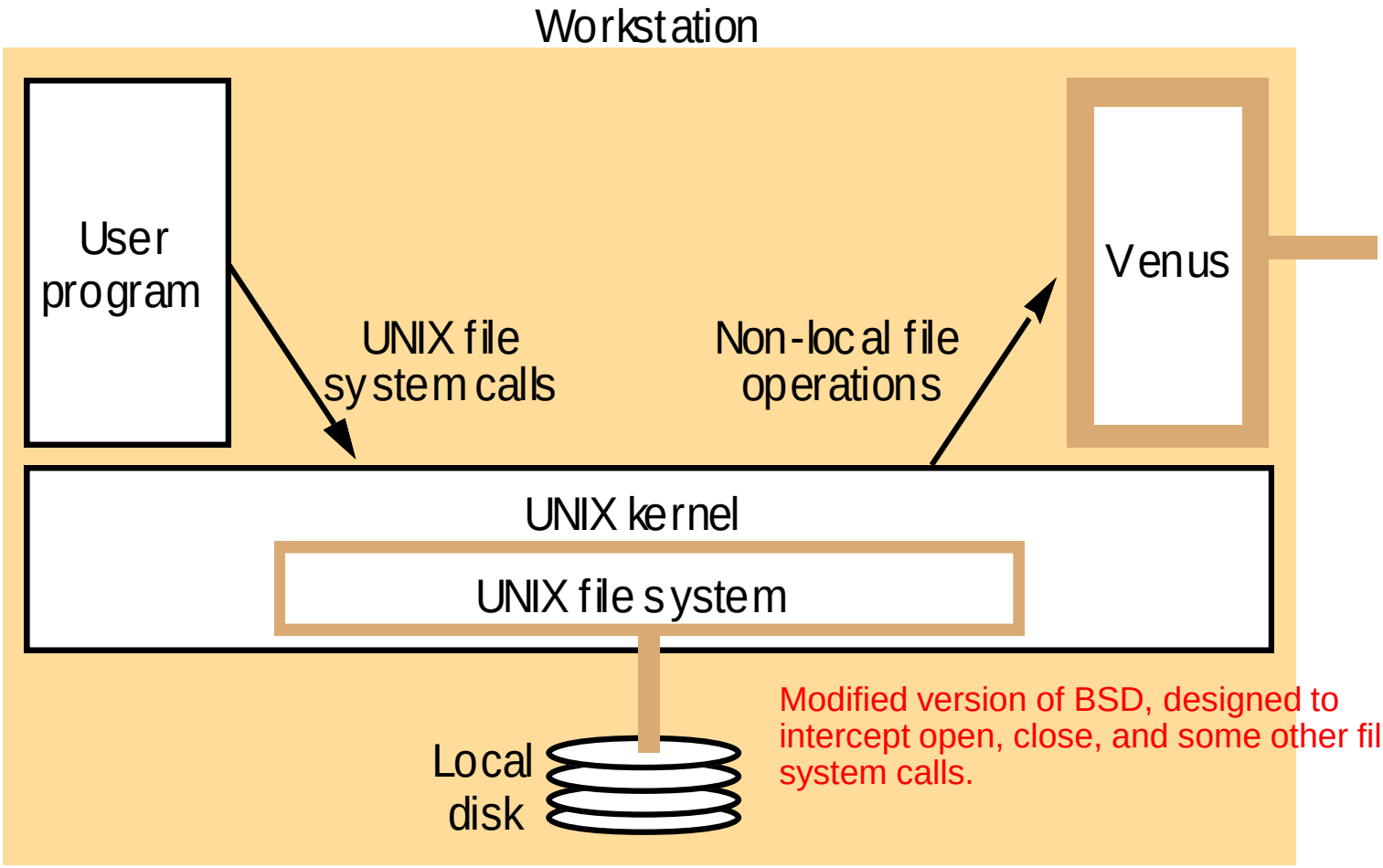
- **Based on (validated) assumptions that**
  - **Most file accesses are by a single user**
  - **Most files are small**
  - **Even a client cache as "large" as 100MB is supportable (e.g., in RAM)**
  - **File reads are much more often that file writes, and typically sequential**

- **We'll see overview only**

Distribution of Processes in the Andrew File System



System Call Interception in AFS



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.

Callback promise= Server will call client if there is a change in the file. Will set its state to *canceled*.

Callback promise state (token) for file is binary: *valid* or *canceled*.

Summary

- Distributed File systems design
- Vanilla file system
- NFS
- AFS

Reminders

- **HW4 due this Thursday**
- **MP4 due this Sunday (demos on Monday)**
- **Mandatory to attend next Tuesday's lecture: semester's last lecture**
- **Final exam posted on Course Schedule**
- **Conflict exam**
  - Please email course staff email by this Thursday (Dec 5) if you feel you might need to take a conflict exam