Distributed File Systems

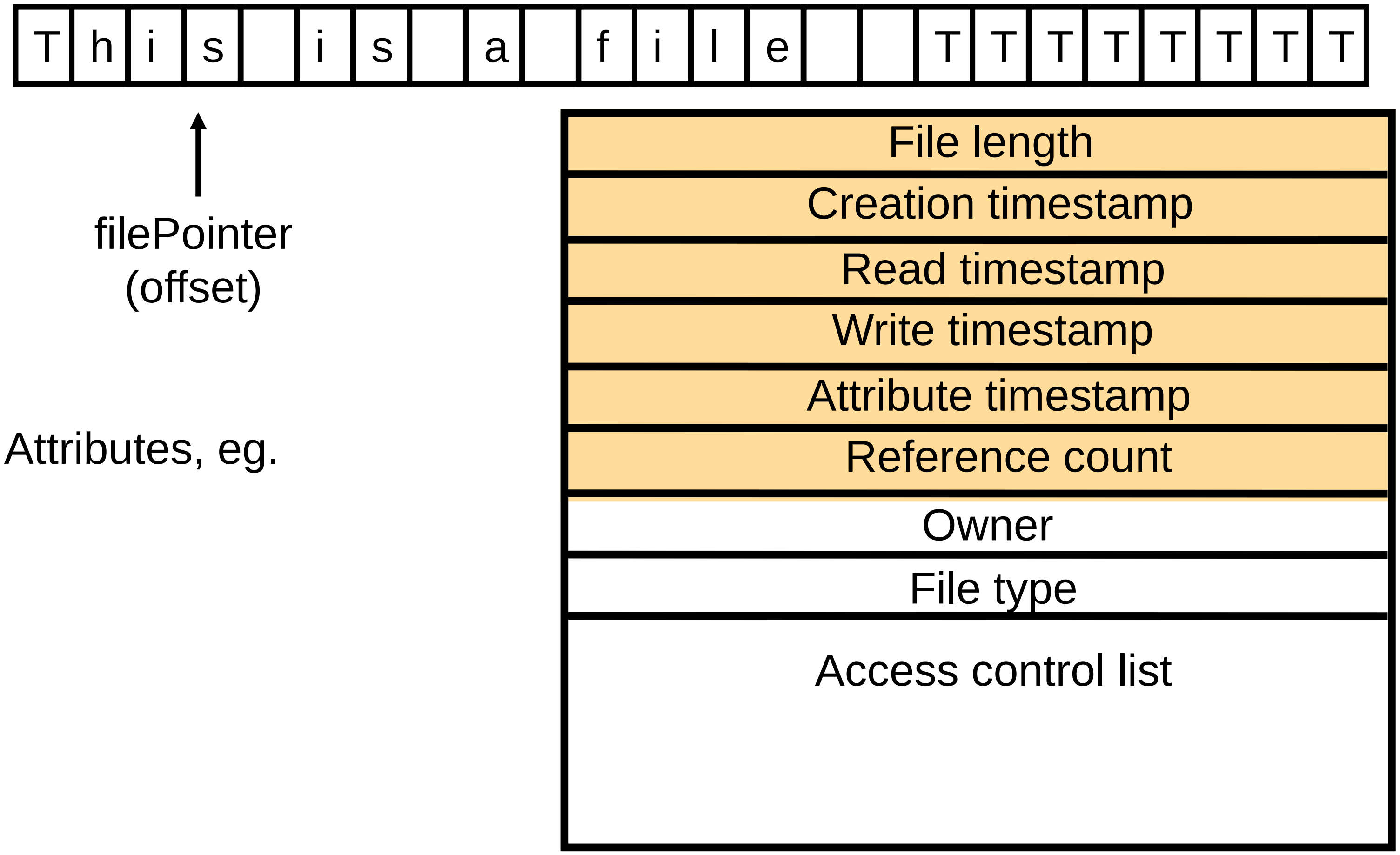
Brian Nielsen
bnielsen@cs.aau.dk

Distributed filesystems

- The most important intranet distributed application
  - Sharing of data (cscw) and programs
  - Easy management and backup, economy
  - Fast, reliable file-server HW (eg RAID)
  - Infrastructure for print+naming
  - User mobility
  - Security
- High transparency requirements
- High performance requirements
- Today:
  - Basic Distributed FS (emulate ordinary FS for clients on different computers)
  - No replication

Files

- Unix Style: sequence of bytes+meta-data



UNIX file system operations

filedes = open(name, mode)	Opens an existing file with the given *name*.
filedes = creat(name, mode)	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)	Transfers *n* bytes from the file referenced by *filedes* to *buffer*.
count = write(filedes, buffer, n)	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*.

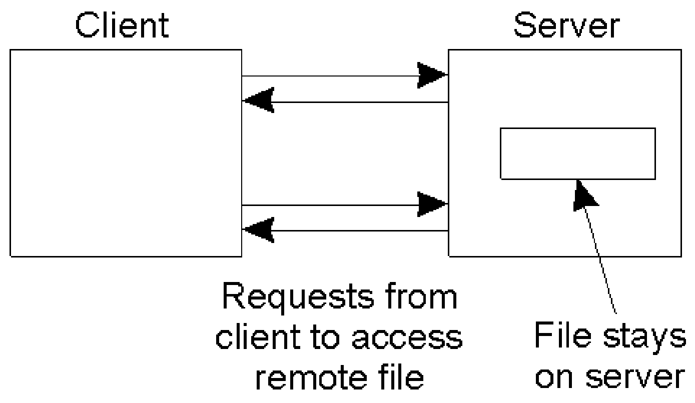
Semantics of File Sharing

Method	**Comment**
UNIX semantics	Every operation on a file is instantly visible to all processes: a **read** operation returns the effect of the last **write** operation Can only be implemented for remote access models in which there is only a **single copy** of the file
Session semantics	No changes are visible to other processes until the file is closed. The effects of **read** and **write** operations are seen only to the client that has opened (a local copy) of the file. When the file is closed, only one client's writes remain
Immutable files	No updates are possible; simplifies sharing and replication
Transaction semantics	All changes occur atomically. The file system supports transactions on a *single* file Issue: how to allow concurrent access to a physically distributed file

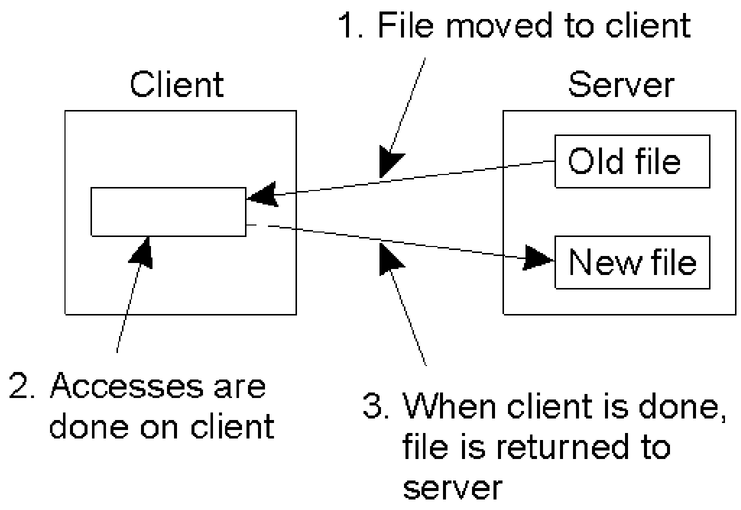
- Four ways of dealing with the shared files in a distributed system.

File System Models

Remote access model



Upload/download model


1. File moved to client
Client
Server
Old file
New file
2. Accesses are done on client
3. When client is done, file is returned to server

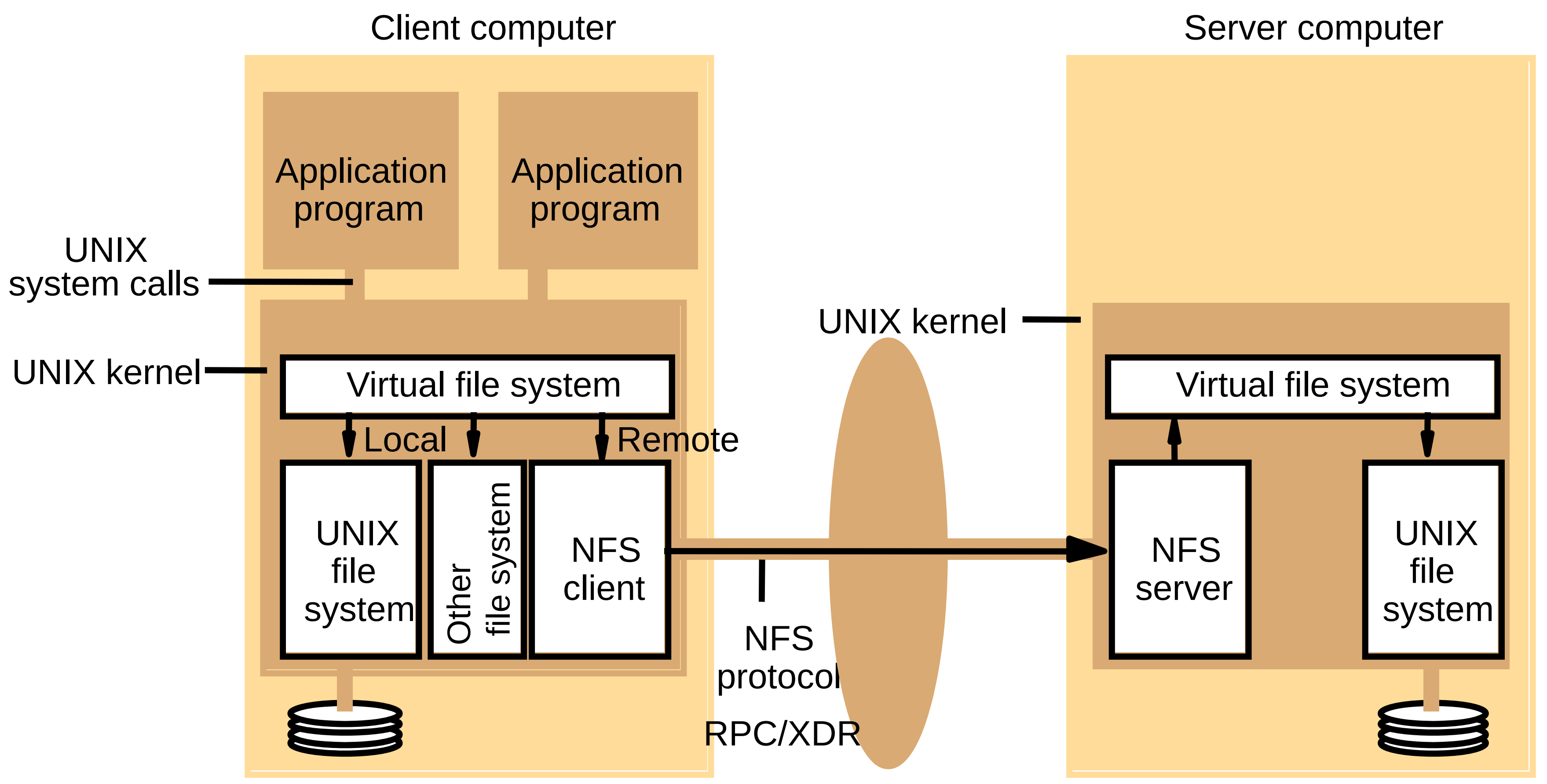
The Sun Network File System (NFS)

- An implementation and a specification (RFC) of a software system for accessing remote files across LANs (or WANs)
- SUN 1985
- RPC/XDR based protocol
- Goals
  - Access transparency
  - Heterogeneous,
  - OS Independent
- Mounting and the actual remote-file-access are distinct services

NFS Protocol

- Provides a set of remote procedure calls for remote file operations.
  - searching for a file within a directory
  - reading a set of directory entries
  - manipulating links and directories
  - accessing file attributes
  - reading and writing files
- NFS servers are **stateless**; each request has to provide a full set of arguments
  (NFS V4 is becoming available – very different, stateful)
- The NFS protocol does not provide concurrency-control mechanisms

NFS architecture



•**Virtual File System** (VFS) provides a standard file system interface that hides the difference between accessing local or remote file systems.

•V-node = virtual file identifier (remote/local, ID)

- ID= i-node number, if local
- ID=fileHandle, if remote (File-Sys id, i-node, i-node-generation)

NFS server operations (simplified) – 1

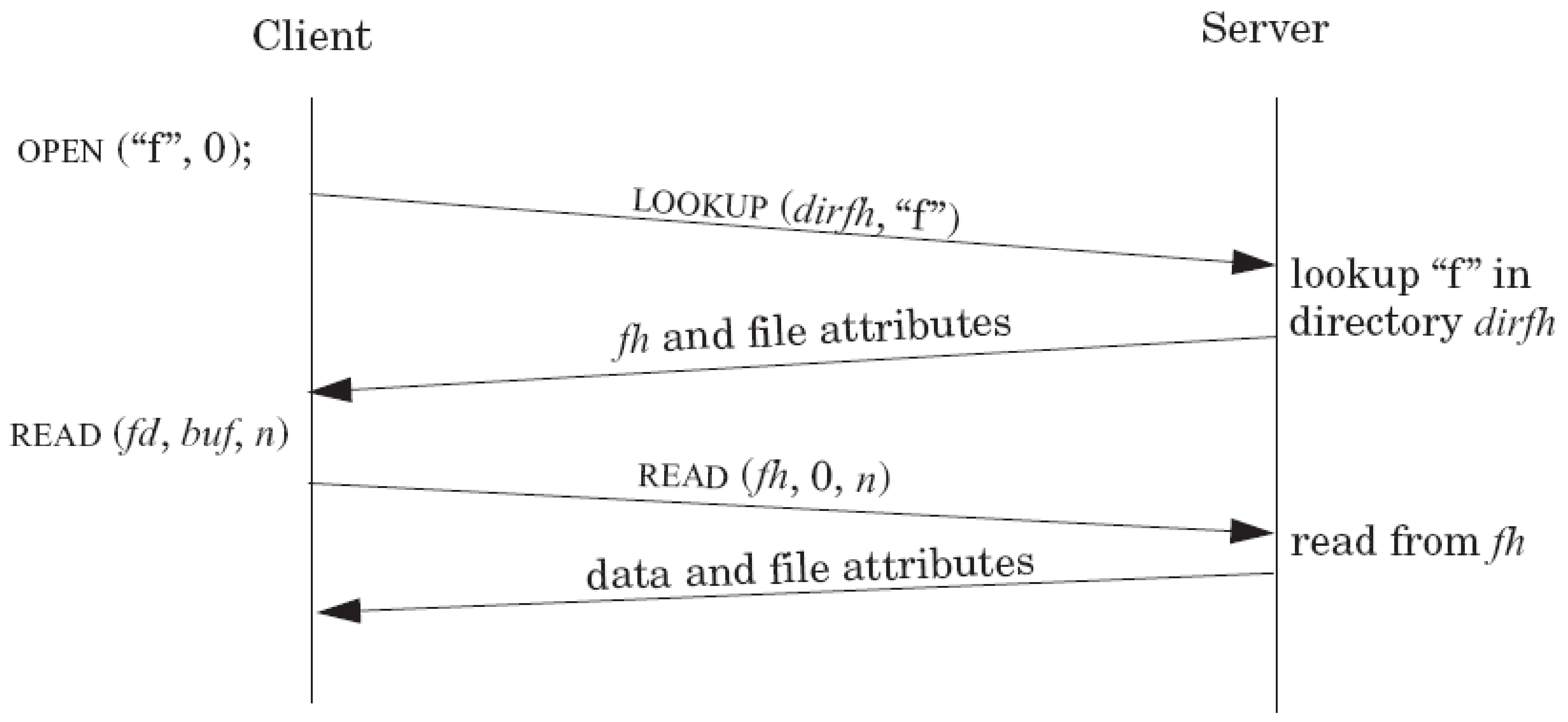
lookup(dirfh, name) → fh, attr	Returns file handle and attributes for the file *name* in the directory *dirfh*.
create(dirfh, name, attr) → newfh, attr	Creates a new file name in directory *dirfh* with attributes *attr* and returns the new file handle and attributes.
remove(dirfh, name) → status	Removes 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*.

Continues on next slide .

NFS server operations (simplified) – 2

symlink(newdirfh, newname, string) $\rightarrow$ *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) $\rightarrow$ *string*	Returns the string that is associated with the symbolic link file identified by *fh*.
mkdir(dirfh, name, attr) $\rightarrow$ *newfh, attr*	Creates a new directory *name* with attributes *attr* and returns the new file handle and attributes.
rmdir(dirfh, name) $\rightarrow$ *status*	Removes the empty directory *name* from the parent directory *dirfh*. Fails if the directory is not empty.
readdir(dirfh, cookie, count) $\rightarrow$ *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* is 0, reads from the first entry in the directory.
statfs(fh) $\rightarrow$ *fsstats*	Returns file system information (such as block size, number of free blocks and so on) for the file system containing a file *fh*.

Simple Example: NFS RPCs for Reading a File



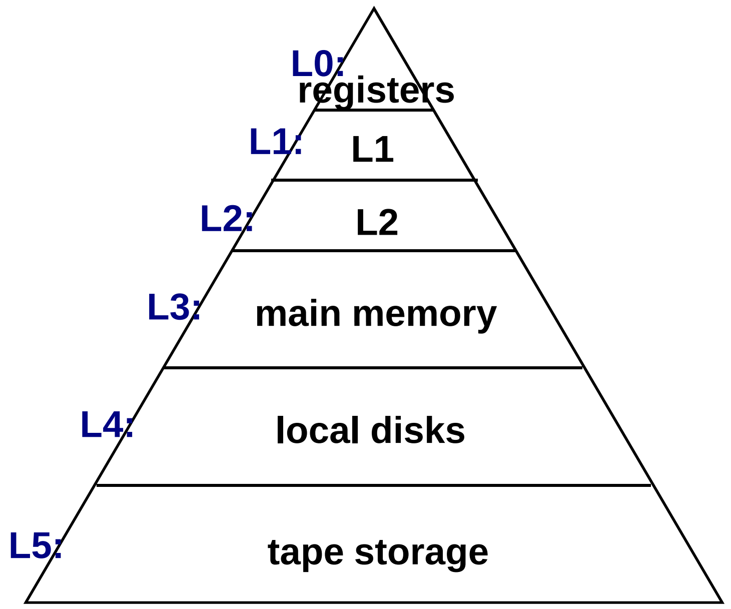
- Where are RPCs for close()?
- File Pointer supplied at each R/W operation?

Fault-Tolerance

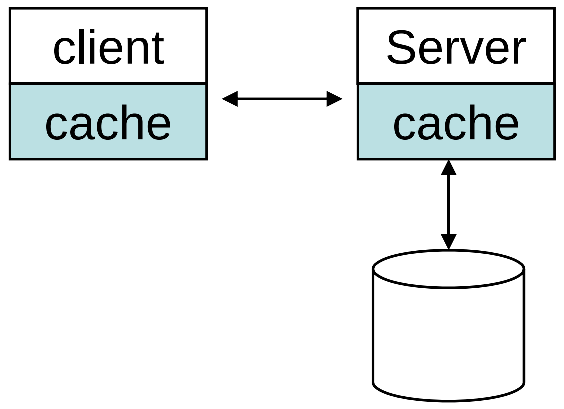
- No open / close!
- File-pointer supplied at each invocation
- Operations are ***Idempotent***
  - Repeated invocations leaves server in same state
- Server is ***State-less!***
  - Server crash: Client can continue unaffected when server recovers
  - Client crash: No state to be cleaned up at server

Caching

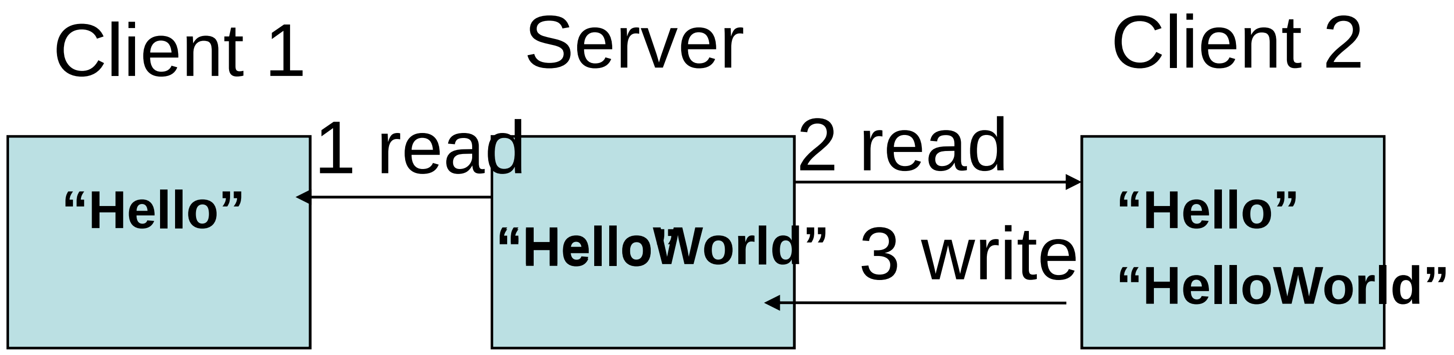
- Store recently accessed **disk-blocks** locally in main memory
- Needed for good performance
  - disk access time
  - network latency,
  - bandwidth
- Exploit memory hierarchy
  - locality-of-reference
  - local access is fast(er)
- Caching in Normal Unix FS
  - Read-ahead
  - Delayed-write (write dirty blocks every 30s)



Caching in NFS



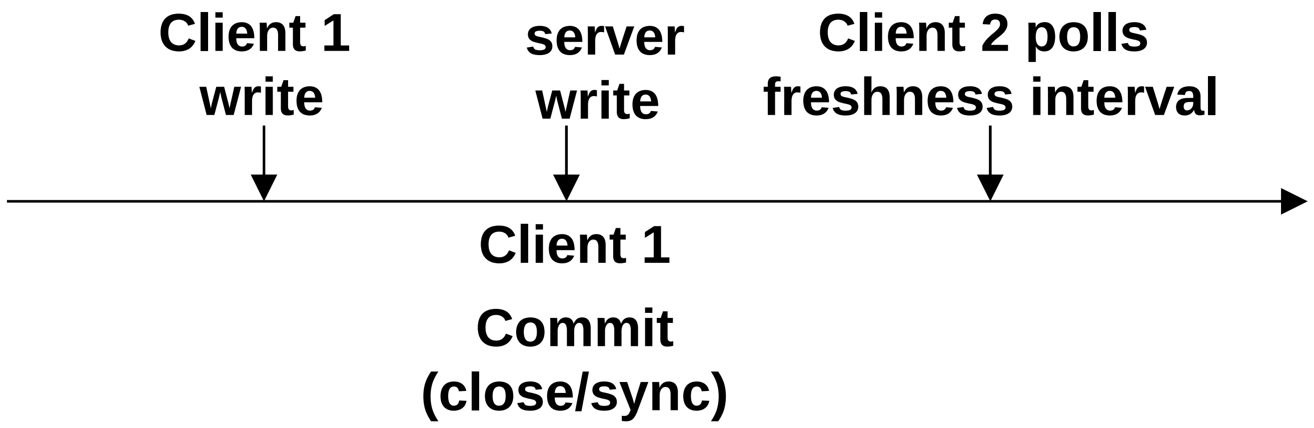
- Server-side caching
  - Read operations: easy.
  - Write operations:
    - Write-through, or
    - Delayed-write: flush on commit operation (+file close)
- Client-side caching
  - Consistency problems when several clients holds copies of the same blocks



Client cache check in NFS

- Time stamps based validation
- Client validation before use of cache contents
  - T_C is the time of the last validation of cached block
    - $T_{m\text{-}server}$ is the modification timestamp stored at server
    - $T_{m\text{-}client}$ is the modification timestamp stored at client
  - T=current time
  - t is the freshness interval
- $(T - T_C < t)$ or $(T_{m\text{-}client} = T_{m\text{-}server})$
  - T_m obtained through getattr polling before cache entry is used
  - t is 3-30s adaptive (compromise between consistency and efficiency)

Inconsistency Time



- Optional block I/O daemon perform commit and read-ahead

NFS Goals

- Access transparency : yes
- Location transparency : yes, (dependent on mounting)
- Failure transparency : partial
- Mobility transparency : yes, (with update of mount tables)
- Replication transparency : no
- HW/SW heterogeneity: Yes
- Consistency: approximation to one-copy semantics (3 sec lag)
- Scalability : no

Performance

- Early experiences
  - Getattr polling (many optimizations needed)
    - Piggy-backing on every operation
    - Apply attributes to all cached blocks
  - Write-through cache at server (no commit)
  - Few writes
- LADDIS Benchmark
- Effective in LAN intranets

The Andrews file system (AFS)

- A distributed computing environment under development since 1983 at Carnegie-Mellon University
- AFS 1, AFS 2, AFS-3
- Available today eg. from www.openafs.org/
- Design objectives
  - Highly scalable: targeted to span over 5000 workstations.
  - Secure: Little discussed here (see the above paper)
- **Whole-file-serving**
- **Whole-file-caching (on client's disk)**
- Shared vs. private files
- Clients more independent of server than NFS

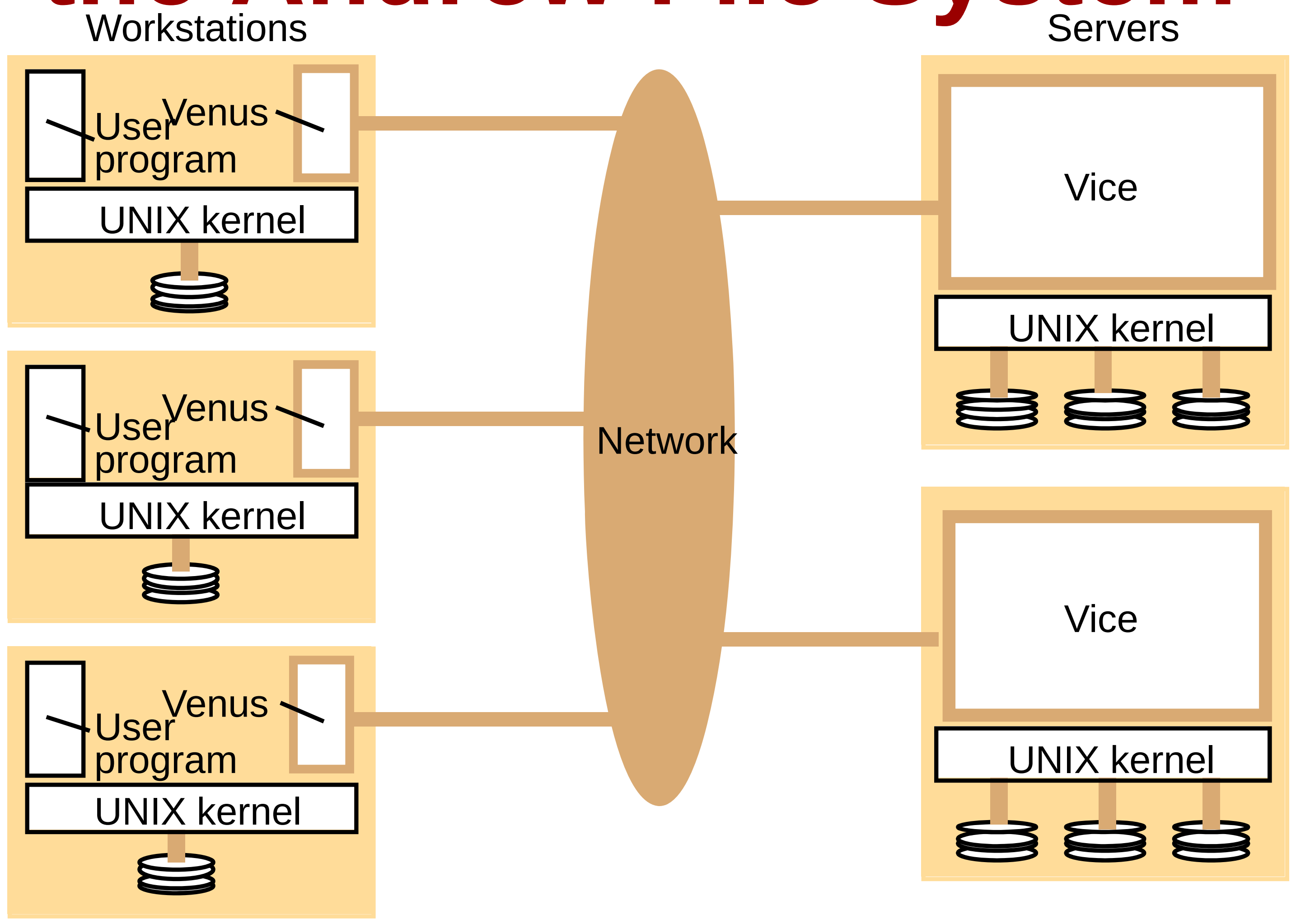
Basic idea

- A user process issues an ***open*** operation on a shared files not in the local cache. The client requests a copy of the file
- The copy is *cached on the local file system*, it is opened, and the user process can continue.
- ***Read*** and ***write*** operations are performed on the local copy
- When the user process performs a ***close*** operation, and if the file has been modified, it is copied back to the server. The server installs the new version of the file, and updates the last modified timestamp for the file.

Why AFS

- For infrequently updated files, the cached copies remain valid for long periods (e.g. system binaries)
- Large caches are possible
- The following observations: (Unix Workload)
  - Files are small (often less than 10Kb)
  - Reads are more common than writes
  - Sequential access is common
  - Most files are read and written by only one user
  - When a file is shared it is often only one user who modifies it
  - Files are referenced in bursts.

Distribution of processes in the Andrew File System



The main components of the Vice service interface

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

Implementation of calls in AFS

User process	*UNIX kernel*	*Venus (client)*	*Net*	*Vice (server)*
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.

Cache Consistency 1

- **“call-back promise”** is a token representing a promise made by server that it will notify the client when the cached file is modified by other clients
- Stored in client disk-cache
- States: **valid or cancelled**
  - Moves from valid to cancelled state when callback is received
  - Client access to file with cancelled call-back promise => fetch fresh copy from server
  - Client access to file with valid call-back promise => use local copy

Cache Consistency 2

- Client Crash: missed callbacks!
  - State of callbacks uncertain
  - First use after restart: send **cache validation request** to server to check **timestamp**
- Communication Failures
  - No communication with server for **T minutes**:
  - Renew callback (leasing principle)
- Server Crash (State-full)
  - List of clients with callback promises stored on disk
  - With atomic update

Update Semantics

- Unix
  - **one-copy semantics**
    - there is one copy of the file, and each write is destructive (*i.e.*, “last write wins”)
- NFS
  - one-copy semantics, except:
    - clients may have out-of-date cache entries for brief periods of time when files are shared
    - this can lead to invalid writes at the server
- AFS
  - one-copy semantics, except:
    - if a callback message is lost, a client will continue working with an out-of-date copy for at most T minutes
    - If two clients writes to the same file concurrently => last to close wins (Use locking if needed)

Failure Performance

- When an NFS server fails, everything fails
  - all accesses have apparent local semantics (except for “soft mounts”)
  - when a server fails, it is as though the local disk has become unobtainable
  - since authentication files are often stored on NFS servers, this brings down the entire system
- When an AFS server fails, life (partly) goes on
  - all locally cached files remain available
  - work is still possible, though there is a higher chance of conflict for shared files

END