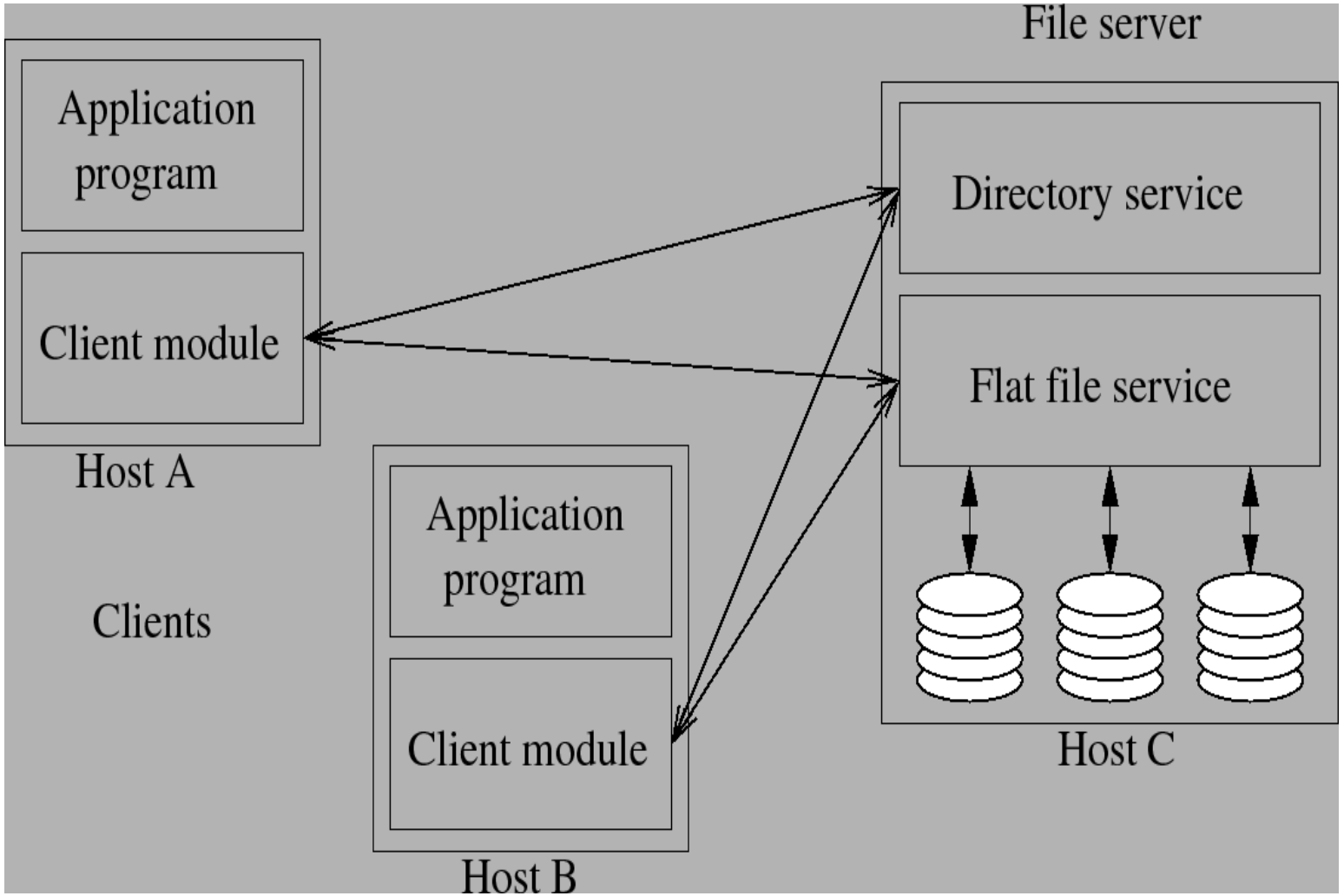
File service architecture



File service architecture

- Flat file service

  The flat file service is concerned with implementing operations on the contents of files. A *unique file identifier* (UFID) is given to the flat file service to refer to the file to be operated on. The UFID is unique over all the files in the distributed system. The flat file service creates a new UFID for each new file that it creates.

- Directory service

  The directory service provides a mapping between text names and their UFIDs. The directory service creates directories and can add and delete files from the directories. The directory service is itself a client of the flat file service since the directory files are stored there.

- Client module
  - integrate/extend the operations of the flat file and directory services
  - provide a common application programming interface (can emulate different file interfaces)
  - stores location of flat file and directory services
  - Though the client module is shown as directly supporting application programs, in practice it integrates into a virtual file system.

Virtual File System

- A virtual file system (VFS) is an abstraction layer on top of a more concrete file system.
- Its purpose is to allow client applications to access different types of concrete file systems in a uniform way.
- A VFS can, for example, be used to access local and network storage devices transparently without the client application noticing the difference.
- Also, it can be used to bridge the differences in Windows, Mac OS and Unix file systems, so that applications can access files on local file systems of those types without having to know what type of file system they're accessing.

Flat file service operations

Any differences between this interface and the UNIX file system primitives?

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 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 (only those attributes that are not shaded in Figure 12.3).

1. Access to the *SetAttributes* operation would normally be restricted to the directory service that provides access to the file.

2. The values of the length and timestamp portions of the attribute record are maintained separately by the flat file service itself.

Differences

- Flat file service has no *open* and *close* operations. Files can be accessed immediately by quoting the appropriate UFID.

- The *Read* and *Write* requests include a parameter specifying a starting point within the file for each transfer, whereas the equivalent UNIX operations do not.

Flat file service interface

- UNIX interface requires that the UNIX file system maintains state, i.e. a file pointer, that is manipulated during reads and writes.
- The flat file service interface differs from the UNIX interface mainly for reasons of fault tolerance:

1. *repeatable operations* - with the exception of Create(), the operations are idempotent, allowing the use of at-least-once RPC semantics. Clients may repeat calls to which they receive no reply.
2. *stateless server* - the flat file service does not need to maintain any state and can be restarted after a failure and resume operation without any need for clients or the server to restore any state.

- Also note that UNIX files require an explicit open command (why?) before they can be accessed, while files in the flat file service can be accessed immediately.

Because the read-write pointer's value has to be retained by the server as long as the relevant file is open.

File service access control

- UNIX checks access rights when a file is opened
  - subsequent checks during read/write are not necessary

- distributed environment
  - server has to check

  Question: Can server store any access control state ?
  - stateless approaches
    1. access check once when UFID is issued
       - client gets an encoded "capability" (who can access and how)
       - capability is submitted with each subsequent request
    2. access check for each request.
  - second is more common (NFS and AFS)

The server cannot store any access control state as this would break the idempotent property.

Directory service interface

- The primary purpose of the directory service is to provide a translation from file names to UFIDs.
- The directory server maintains directory files that contain mappings between text file names and UFIDs. The directory files are stored in the flat file server and so the directory server is itself a client to the flat file server.
- A hierarchical file system can be built up from repeated accesses. E.g., the root directory has name "/" and it contains subdirectories with names "usr", "home", "etc", which themselves contain other subdirectories or files. A client function can make requests for the UFIDs in turn, to proceed through the path to the file or directory at the end.

Directory service operations

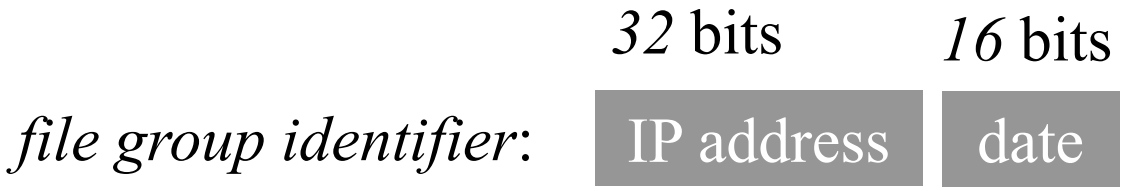
Lookup(Dir, Name) -> FileId — throws *NotFound*	Locates the text name in the directory and returns the relevant UFID. If *Name* is not in the directory, throws an exception.
AddName(Dir, Name, FileId) — throws *NameDuplicate*	If *Name* is not in the directory, adds (*Name*, *File*) to the directory and 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*.

File groups

- A *file group* is a collection of files located on a given server. A server may hold several file groups and file groups can be moved between servers, but a file cannot change file group.
- File groups support the allocation of files to the servers in larger logical units and enable the file service to be implemented over several servers.
- Files are given UFIDs that ensure uniqueness across different servers, e.g. by concatenating the server IP address (32 bits) with a date that the file was created (16 bits). This allows the files in a group, i.e. that have a common part to their UFID called the *file group identifier*, to be relocated to a different server without conflicting with files already there.
- The file service needs to maintain a mapping of UFIDs to servers. This can be cached at the client module.

File group identifier

Can we use the IP address to locate the file group ?

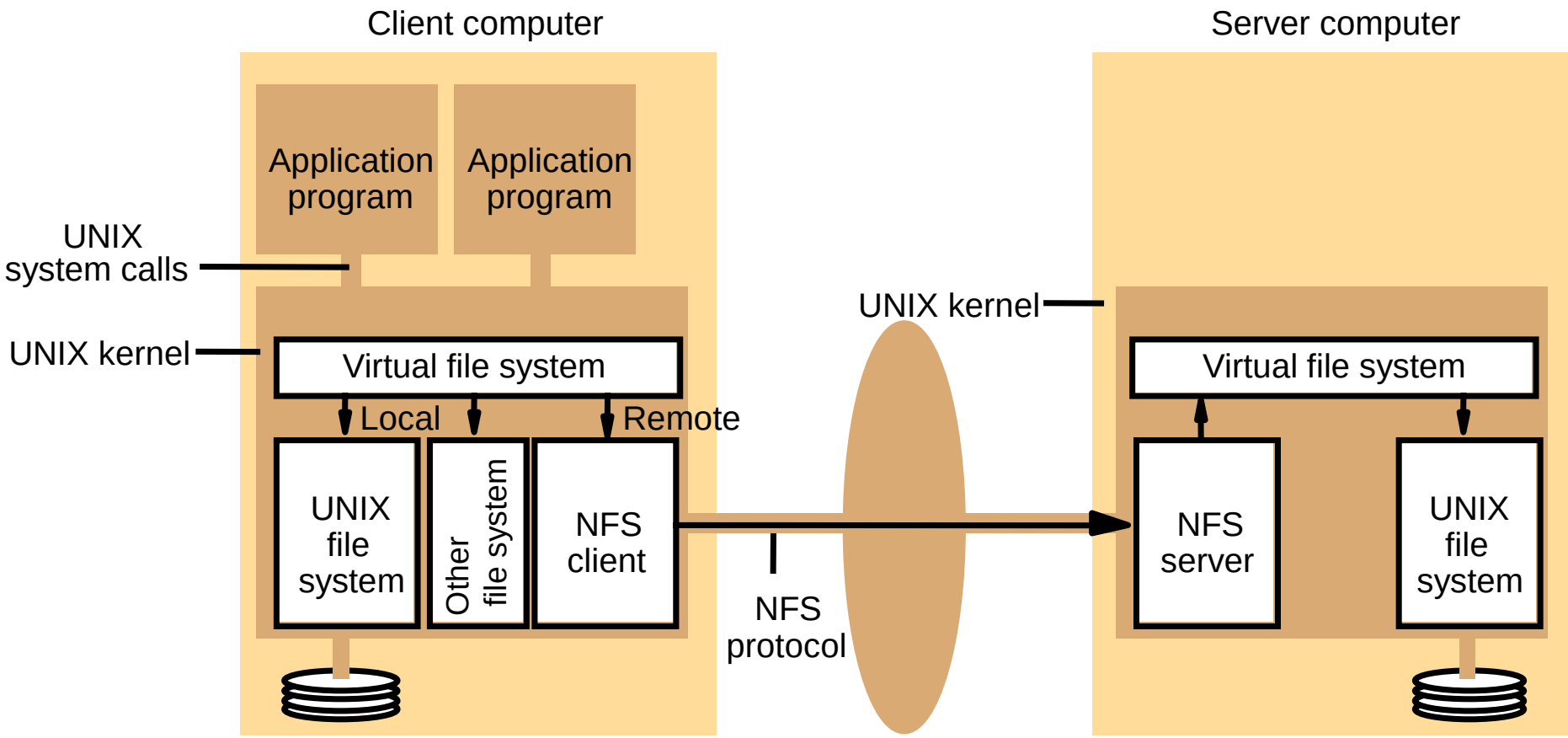


No, because a file group may be moved to another server. Thus, a mapping between group identifiers and servers should be maintained by the file service.

Case Study: Sun Network File System (Sun NFS)

- Industry standard for local networks since the 1980's
- OS independent
- The Sun Network File System (NFS) follows the abstract system shown earlier
- There are many implementations of NFS and they all follow the NFS protocol using a set of RPCs that provide the means for the client to perform operations on the remote file store
- We consider a UNIX implementation

NFS architecture: virtual file system

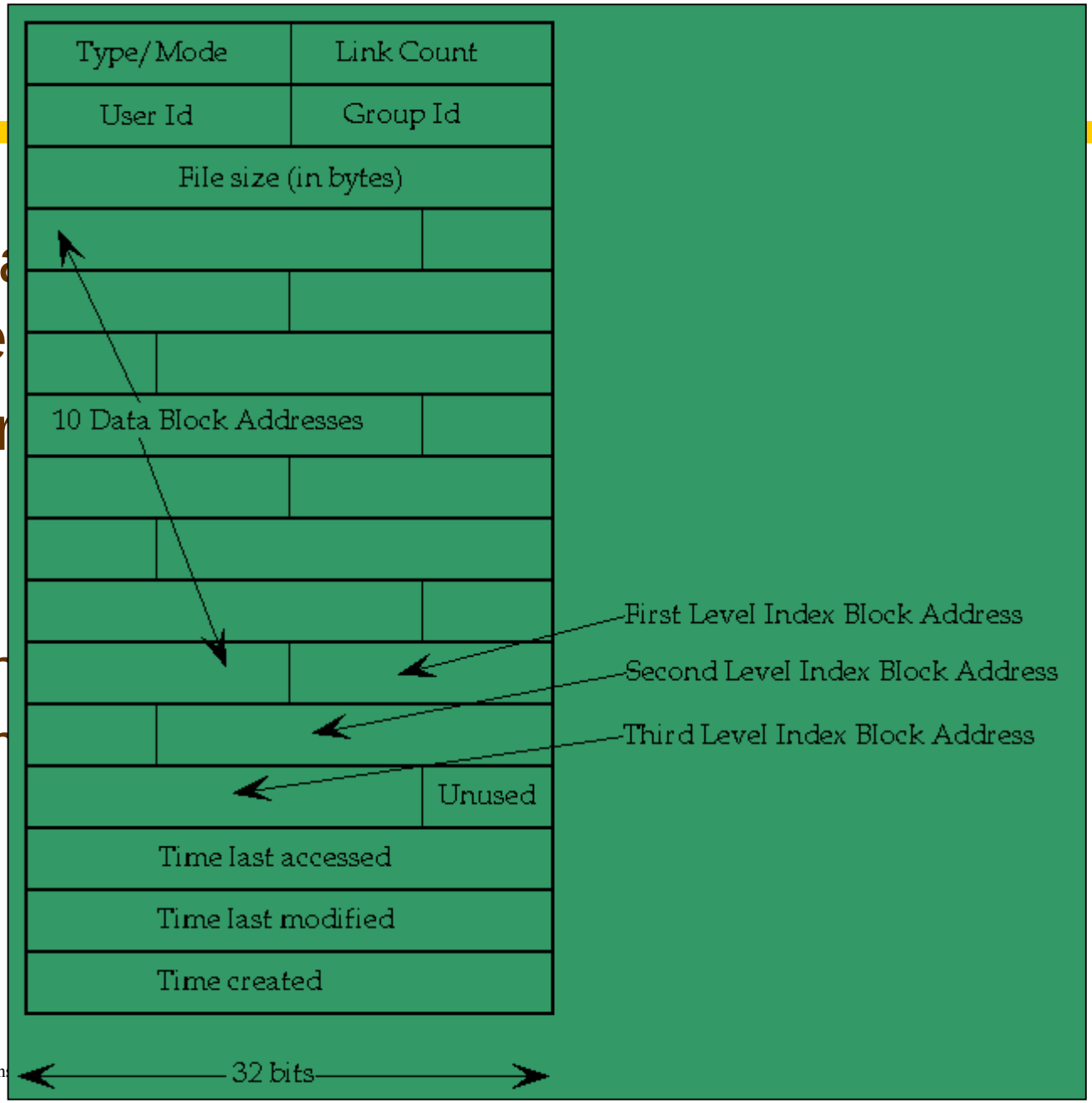


Virtual file system

- UNIX uses a *virtual file system* (VFS) to provide transparent access to any number of different file systems.
- The VFS maintains a VFS structure for each filesystem in use. The VFS structure relates a remote filesystem to the local filesystem.
- The VFS maintains a *v-node* for each open file, and this records an indicator as to whether the file is local or remote.
- If the file is local then the v-node contains a reference to the file's *i-node* on the UNIX file system.
- If the file is remote then the v-node contains a reference to the file's NFS *file handle* which is a combination of *filesystem identifier*, i-node number and whatever else the NFS server needs to identify the file.

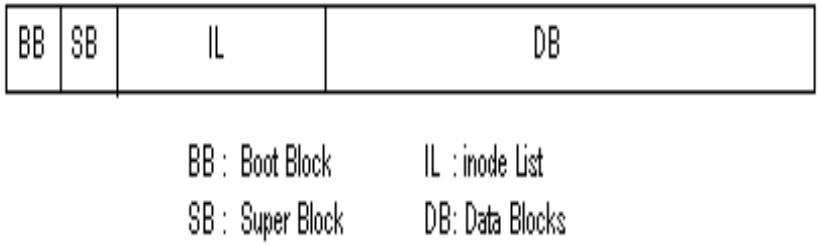
i-node

- An inode is a
  style file syste
- An inode stor
  file, directory,
- Each file has
  number (often
  the file system



Unix file system

- The boot block contains the code to bootstrap the OS.
- The super block contains information about the entire disk.
- The I-node list a list of inodes
- The data blocks contains the actual data in the form of directories and files.



Unix filesystem

- The standard Unix filesystem, generically referred to as the "ufs" filesystem, is arranged on a disk partition using a "linked-list" of pointers to data.
- The structure of a partition begins with, or is defined by, the "superblock."
- The superblock is a data structure which includes information about the: type of filesystem (i.e. "ufs", "ext2fs", etc.), size and modification time of the the filesystem, list of free and allocated blocks and the first inode, which points to (you guessed it) the root directory, "/".
- The superblock is always replicated, to provide fault tolerance against disk failure in the first superblock.

Client integration

The NFS client is integrated within the kernel so that (advantages):

- user programs can access files via UNIX system calls without recompilation or reloading;
- a single client module serves all of the user-level processes, with a shared cache;
- the encryption key used to authenticate user IDs passed to the server can be retained in the kernel, preventing impersonation by user-level clients.

The client transfers blocks of files from the server host to the local host and caches them, sharing the same buffer cache as used for local input-output system. Since several hosts may be accessing the same remote file, caching presents a problem of consistency.

Access control

- The NFS server is stateless and doesn't keep open files for clients
- So the server must check the user's identity against the file's access permission attributes afresh on each request (uid and gid)
- Using DES encryption of the user's authentication information to close a security loophole.

Server interface

- The NFS server interface integrates both the directory and file operations in a single service. The creation and insertion of file names in directories is performed by a single create operation, which takes the text name of the new file and file handle for the target directory as arguments.
- The primitives of the interface largely resemble the UNIX filesystem primitives.

NFS server operations (simplified)

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) -> 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* is 0, 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*.

Mount service(1)

- The process of including a new filesystem is called mounting
- A *mount service* is a separate service process that runs at user level on each NFS server computer.
- On each server, there is a file “/etc/exports”, which has the names of local filesystems that are available for remote mounting
- Clients use a modified mount command to request mounting of a remote filesystem
- hard-mounted
  - user process is suspended until request is successful
  - when server is not responding
  - request is retried until it's satisfied
- soft-mounted
  - if server fails, client returns failure after a small # of retries
  - user process handles the failure

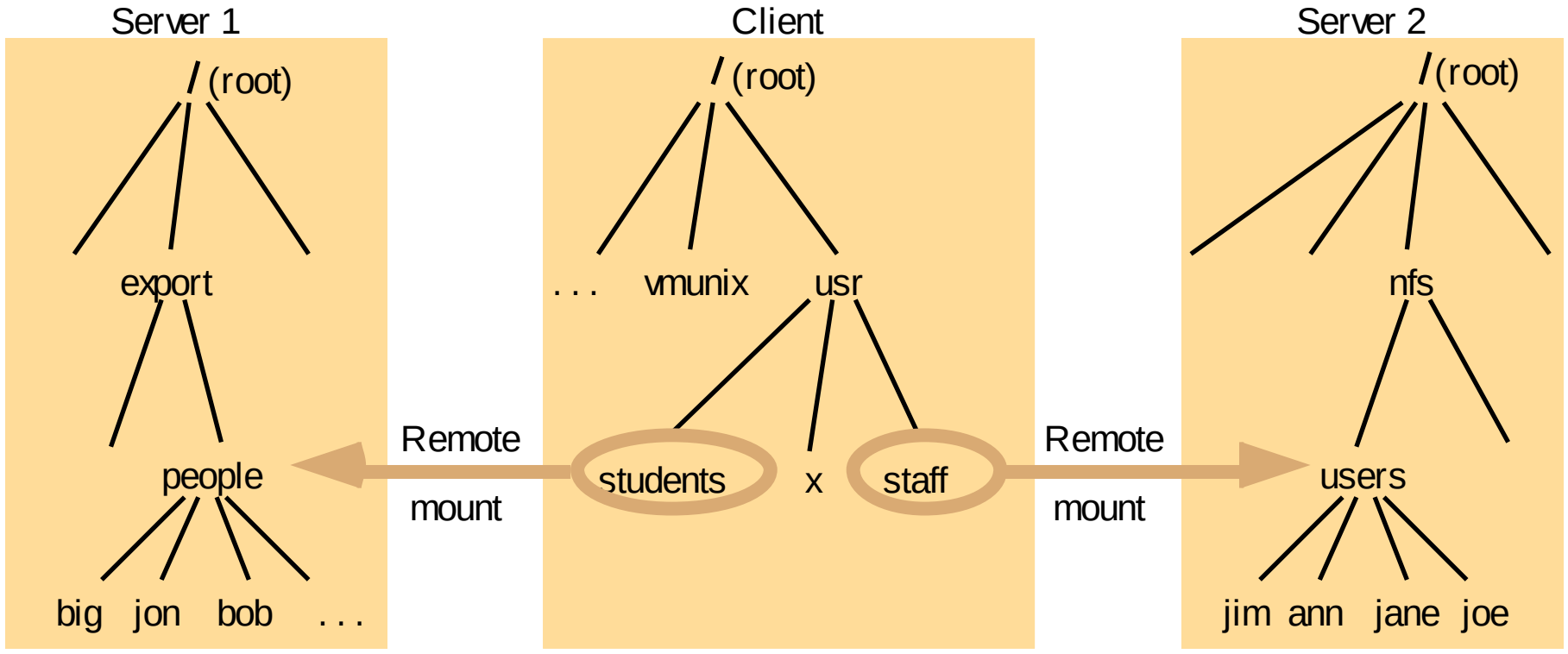
Mount service(2)

- Each server maintains a file that describes which parts of the local filesystems that are available for remote mounting.

```
aharwood@htpc:~$ cat /etc/exports
# /etc/exports: the access control list for
# filesystems which may be exported
# to NFS clients. See exports(5).
/store 192.168.1.0/255.255.255.0(rw)
```

In the above example all hosts on the subnet can mount the filesystem directory /store with read and write access.

Local and remote file systems



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.

Programs running at Client can access files at Server 1 and Server 2 by using pathnames such as /usr/students/jon and /usr/staff/ann.

Server caching(1)

- In conventional UNIX systems, data read from the disk is retained in a main memory buffer cache and are evicted when the buffer space is required for other pages. Accesses to cached data does not require a disk access.
- *Read-ahead* anticipates read accesses and fetches the pages following those that have been recently read.
- *Delayed-write* (or write-back) optimizes writes to the disk by only writing pages when both they have been modified and when they are evicted. A UNIX sync operation flushes modified pages to disk every 30 seconds.

This works for a conventional files system, on a single host, because there is only one cache and all file accesses cannot bypass the cache.

Server caching(2)

Use of the cache at the server for client reads does not introduce any problems. However use of the cache for writes requires special care to ensure that client can be confident that the writes are persistent, especially in the event of a server crash.

- *Write-through* - data is written to cache and also directly to the disk. This increases disk I/O and increases the latency for write operations. The operation completes when the data has been written to disk.
- *Commit* - data is written to cache and is written to disk when a commit operation is received for the data. A reply to the commit is sent when the data has been written to disk.
- Servers use *Write-through* and clients *Commit*
- Pro and Con for each option?

Pros and Cons

- *Write-through* is poor when the server receives a large number of write requests for the same data. It however saves network bandwidth.
- *Commit* uses more network bandwidth and may lead to uncommitted data being lost. However it receives the full benefit of the cache.

Client caching

- Why? In order to reduce the number of requests transmitted to servers
- What? Caches results of read, write, getattr, lookup, readdir

1. Client caching introduces the potential for different versions of files or portions of files to exist in different client nodes.
2. Caching at the client introduces the cache consistency problem since now there is a cache at the client and the server, and there may be more than one client as well, each with its own cache.

- How? Clients responsibility to poll the server for consistency

Client caching: reading (1)

- timestamp-based methods for consistency validation
  - T_c: time when the cache entry was last validated by the client
  - T_m: time when the block was last modified at the server
- cache entry is valid at time T if:
  1. $T - T_c < t$, where t is the freshness interval
     - t is adaptively adjusted:
       - files: 3 to 30 seconds depending on frequency of updates
       - directories: 30 to 60 seconds
  2. *OR* $T_{m_{client}} = T_{m_{server}}$

The data has not been modified at the server since the cache entry was made. Tm client : the value of Tm recorded at the client matches the value of Tm at the server since the cache entry was made.

Client caching: reading (2)

- need validation for all cache accesses
- condition “1" can be determined by the client alone--performed first (does not require network I/O)
- Reducing getattr() to the server [for getting Tm_{server}]
  1. new value of Tm_{server} is received, apply to all cache entries from the same file
  2. piggyback getattr() on file operations
  3. adaptive algorithm for update *t*
- validation doesn't guarantee the same level of consistency as one-copy because recent updates are not always visible to clients sharing a file

Client caching: writing

- dirty: modified page in cache
- flush to disk: file is closed or sync from client
- bio-daemon (*b*lock *i*nput-*o*utput, at client side)
  - read-ahead: after each read request, request the next file block from the server as well
  - delayed write: after a block is filled by a client operation, it's sent to the server
  - reduce the time to wait for read/write

Performance

- overhead/penalty is low
- main problems
  - frequent getattr() for cache validation (piggybacking)
  - relatively poor performance is write-through used on the server
  - write < 5%

Summary for NFS

- *Access transparency*: Yes. Applications programs are usually not aware that files are remote and no changes are need to applications in order to access remote files.
- *Location transparency*: Not enforced. NFS does not enforce a global namespace since client file systems may mount shared file systems at different points. Thus an application that works on one client may not work on another.
- *Mobility transparency*: No. If the server changes then each client must be updated.
- *Scalability*: Good, could be better. The system can grow to accommodate more file servers as needed. Bottlenecks are seen when many processes access a single file.
- *File replication*: Not supported for updates. Additional services can be used to facilitate this.
- *Hardware and operating system heterogeneity*: Good. NFS is implemented on almost every known operating system and hardware platform.
- *Fault tolerance*: Acceptable. NFS is stateless and idempotent. Options exist for how to handle failures.
- *Consistency*: Tunable. NFS is not recommended for close synchronization between processes.
- *Security*: Kerberos is integrated with NFS. Secure RPC is also an option being developed.
- *Efficiency*: Acceptable. Many options exist for tuning NFS.

Assignment#2 (Chapter 12)

- 12.5
- 12.10