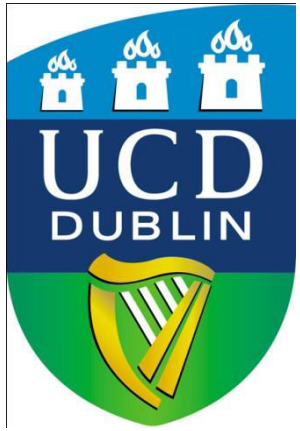
Distributed Systems: Distributed File Systems

Anca Jurcut
E-mail: anca.jurcut@ucd.ie

School of Computer Science and Informatics
University College Dublin
Ireland



From Previous Lecture…

- Sharing of resources is a key goal for distributed systems
- This can be done in many ways:
  - Databases, the Web
  - Distributed Shared Memory
  - Remote Objects
  - .....
- To do this, we need a File System!
- A DFS aims to make remote files accessible to a local user
- DFS Components: File Servers, Clients, File Services
- Example of DFS: *Novell Netware, Windows File Sharing, ...*

From Previous Lecture…

- **Naming and Transparency**
  - Naming schemes enforce a separation of concerns between the logical objects that we work with and the corresponding physical objects
  - Location transparency
  - Location independence
- **Access Transparency :** requires that a common set of operations be used for both local and remote file management
- **Global Naming Scheme**: DFS Naming Schemes is how to provide a global context for every file name
- **Remote File Access:** two basic approaches
  - Upload/Download Model.
  - Remote Access Model

Outline

- Introduction
- DFS Issues
  - Naming and Transparency
  - Remote File Access
  - Stateful versus Stateless
  - File Replication
  - Security
- Example Systems

Stateful File Service

- The File Server keeps a copy of a requested file in memory until the client has finished with it.
  - 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  - Server must reclaim the main-memory space used by clients who are no longer active.

- The principle advantage of stateful file services is increased performance:
  - Fewer disk accesses.
  - Stateful server knows if a file was opened for sequential access and can thus read ahead the next blocks.

- Example: Andrew File System

Stateless File Server

- The File Server treats each request for a file as self-contained.
  - Each request must identify both the file name to be accessed and position of the data (in the file) that is to be accessed.
  - This approach removes the need to establish and terminate a connection by open and close operations.
  - Reads and writes take place as remote messages (or cache lookups).
- NFS is an example of a stateless file service.

Stateful versus Stateless Services

- 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 failures and recovery are almost unnoticeable.
  - A newly reincarnated server can respond to a self-contained request without any difficulty.

Stateful versus Stateless Services

- Penalties for using the robust stateless service:
  - longer request messages
  - slower request processing

- 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.

Outline

- Introduction
- DFS Issues
  - Naming and Transparency
  - Remote File Access
  - Stateful versus Stateless
  - File Replication
  - Security
- Example Systems

File Replication

- File Replication
  - Multiple physical copies of a single logical file.
  - The Naming Service maintains references to each of the copies.
  - The existence of the copies is hidden from the user (replication transparency)

- This is not possible in systems such as NFS and Windows!
  - To maintain multiple copies of a single file requires location independence!

- See Andrew File System and its successor CODA

File Replication

- Why Replicate?
  - **Performance Enhancement**: Requests for accessing the file can be spread over multiple File Servers, or can be retrieved from the “closest server”
  - **Increased Availability**: File still accessible in the event of server failures, or communication disruption
- When to Replicate?
  - What are the conditions under which a file should be replicated?
  - When are there too many copies?

File Replication Issues

- The Update Problem:
  - Replicas of a file denote the same logical entity, and thus an update to any replica must be reflected on all other replicas.
  - This can be expensive (in terms of bandwidth)!
- Demand Replication:
  - Reading a non-local replica causes it to be cached locally, thereby generating a new non-primary replica.
  - What do we do with this new copy?
- Concurrency Problem:
  - Two users concurrently try to update a replica.
  - Whose update is kept, and why?

Outline

- Introduction
- DFS Issues
  - Naming and Transparency
  - Remote File Access
  - Stateful versus Stateless
  - File Replication
  - Security
- Example Systems

Security

- Security is key to DFS:
  - It controls access to the data stored in the files
  - It deals with authentication of client requests.

- Most file systems provide access control mechanisms based on the use of access control lists.

- UNIX performs an access rights check against the access mode whenever an open operation is performed.
  - The access rights for the user are then maintained until the file is closed.
  - The rights check is based on the UID which was authenticated when the user logged on.

- Unfortunately, for Distributed File Systems, this is not enough!

Security

- In DFS, security checks must be carried out before each operation is carried out.
  - Otherwise the server is left unprotected

- Two approaches exist:
  - Name Resolution Security Check.
    - Whenever a name is resolved to a UFID (Unique File ID), a check is performed.
    - If the user passes the check, then a capability is returned along with the results of the security check.
    - This capability is passed to the server with all subsequent RPCs (remote procedure calls).
  - RPC Check
    - User Identity submitted with every RPC.
    - Send as an encrypted digital signature.
    - Access checks are carried out by the server for every operation.

- This approach does not cater for identity theft!

Security Examples

- Basic Sun NFS RPC requires an unencrypted 16-bit userID and groupID with each request.
  - This is checked against access permissions in the file attributes
- In its simplest form, the userID and groupID are not encrypted!
  - This means that anybody who knows a valid userID and groupID can perform file operations!!!
- This problem has been solved through a revision of the Sun RPC protocol to include DES encryption.
- Another extension has seen NFS integrated with the Kerberos security system.

Distributed File Systems: Case Studies

Comparing DFSs

- **Objective**. What are the design objectives underlying the solution?
- **Security**. How is access to the File Service controlled?
- **Naming**. What kind of naming scheme is used?
  - i.e. Location Transparency, Location Independence
- **File Access**. How are the files accessed?
  - Upload/Download, Remote Access

Comparing DFS

- **Cache Update Policy**. Is caching used? What update policy is applied?
- **Consistency**. How does the DFS ensure consistency of files?
- **Service Type**. Is the DFS a stateful or stateless service?
- **Replication**. Is replication supported? What mechanism is used?

Distributed Systems: Flat File System

File Server Architecture

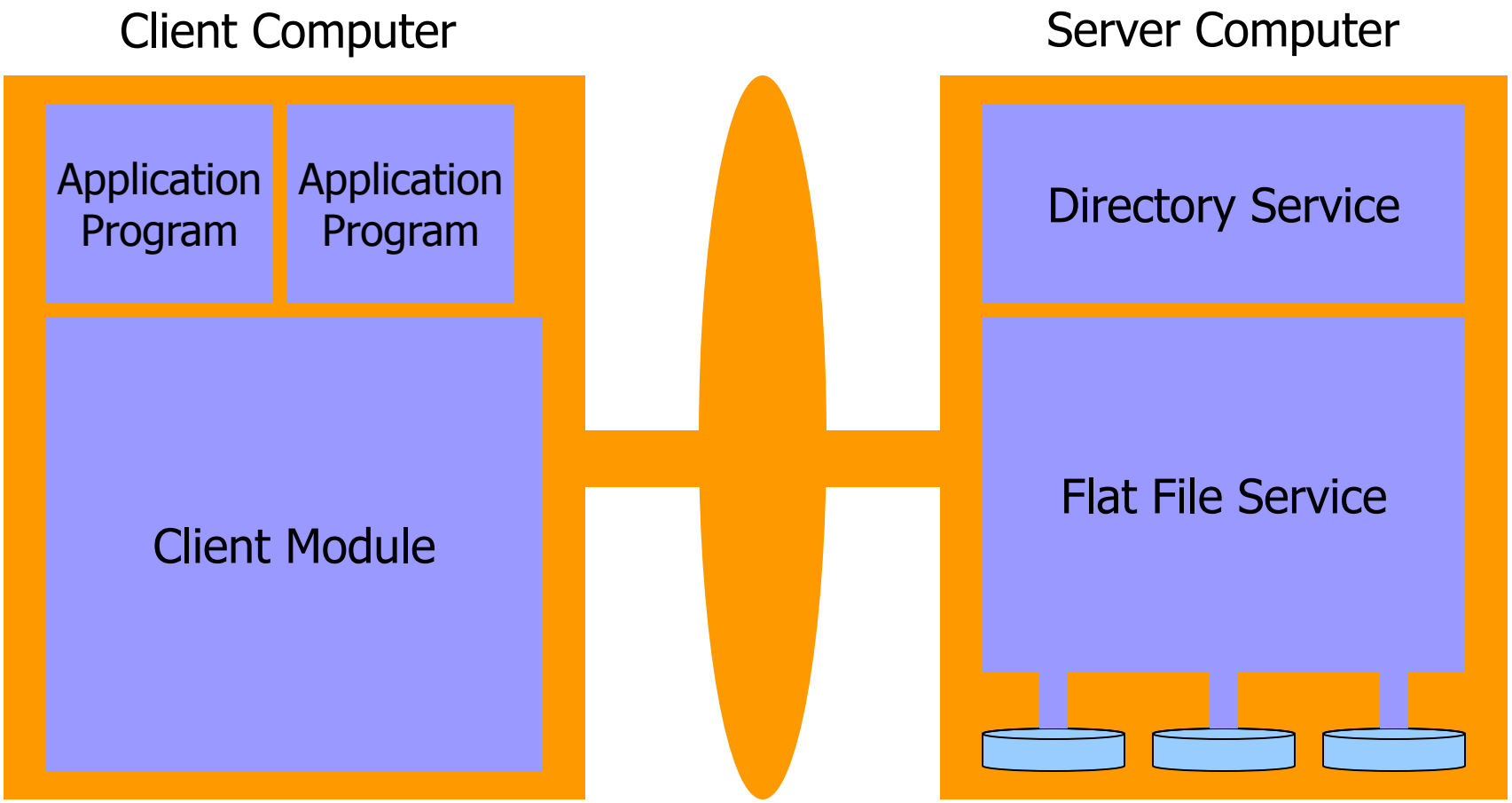
- Abstract DFS Architecture
  - Common framework that underpins both NFS and AFS
  - Designed as a stateless implementation

- Consists of three core components:
  - Flat File Service
  - Directory Service
  - Client Module

- The two Service components each specify export an interface that, together with relevant RPC interfaces, provide a complete set of DFS operations.

- In addition, the Client provides a standardised interface that can easily be adapted to a given OS.

File Service Architecture



Flat File Service

- Concerned with implementing operations on the contents of files.

- Unique File Identifiers (UFIDs) are used to refer to files in all requests for flat file service operations.
  - UFIDs are a long sequence of bits chosen so that each file has a UFID that is unique to the DFS.

- When the FFS receives a request for a new file:
  - It generates a new UFID, and
  - Returns the UFID to the requester

Directory Service

- Provides a mapping between text names for files and UFIDs.
- Clients can obtain a UFID by quoting the text name
- This service provides support for:
  - Generating new directories
  - Adding new file names to directories
  - Removing UFIDs from directories

Client Module

- One instance per computer

- Integrates and extends the operations of the flat file service and the directory service under a single API
  - This is available to user-level programs on the client
  - Includes a cache of recently used file blocks at the client
  - A Write-Through cache update policy is employed.

- For example:
  - When installed on a UNIX platform, the client module would be modified to emulate the full set of UNIX file operations.
  - This would include the interpretation of multi-part file names by iterative requests to the Directory Service.

Client Module

- Access to the Remote File System is through two additional commands:
  - **ffsmount <local-mount-point> <server address>**
  - Mounts the remote file system specified by the server address on the local computer with the root of the remote file system set to the local mount point.
  - **ffsunmount <local-mount-point>**
  - Unmounts the remote file system that was previously mounted at the specified local mount point.

- The Client Module maintains a list (stored in a local file) of current mount points.
  - If the local machine crashes, then any mount points is this file are automatically remounted as the system is rebooted.
  - Cached data is flushed on reboot.

Flat File Service Interface

- FFS Interface defines six operations:
  - ReadFile(FileId, i, n) -> Data
  - WriteFile(FileId, i, Data)
  - Create() -> FileId
  - Delete(FileId)
  - GetAttributes(FileId) -> Attr
  - SetAttributes(FileId, Attr)

- While this offers only a subset of the UNIX operations (see earlier), we can emulate those operations using the ones above!

- The above operations have been chosen because they allow:
  - Repeatable Operations (uses the at-least-once RPC semantics)
  - Stateless Servers (no state need be maintained – e.g. read-write pointer)

Directory Service Interface

- Another set of RPCs that allow for resolution of file names to UFIDs.
  - File name mappings are organised into directory files
  - Each directory is stored in a conventional file that recognised by the File System.

- DS Interface:
  - Lookup(Dir, Name) -> FileId
  - AddName(Dir, Name, FileId)
  - UnName(Dir, Name)
  - GetNames(Dir, Pattern) -> NameSeq

- The interface operations work only on a single directory – recursive directory searching is not supported.
  - Hierarchical Name Structures can be delivered on top of this basic service.

File Groups

- A *file group* is a collection of files located on a server.
  - A server may have many file groups but files cannot change the group to which they belong.
  - File groups are used to support the allocation of files to file servers in larger logical units.
- Where a DFS supports file groups, the UFID must be extended to include a file group identifier (IP address + Timestamp)
- The IP address cannot be used for the purpose of locating the file group since it may move to other servers.
  - The file service must maintain a mapping between file groups and servers.

FFS Review

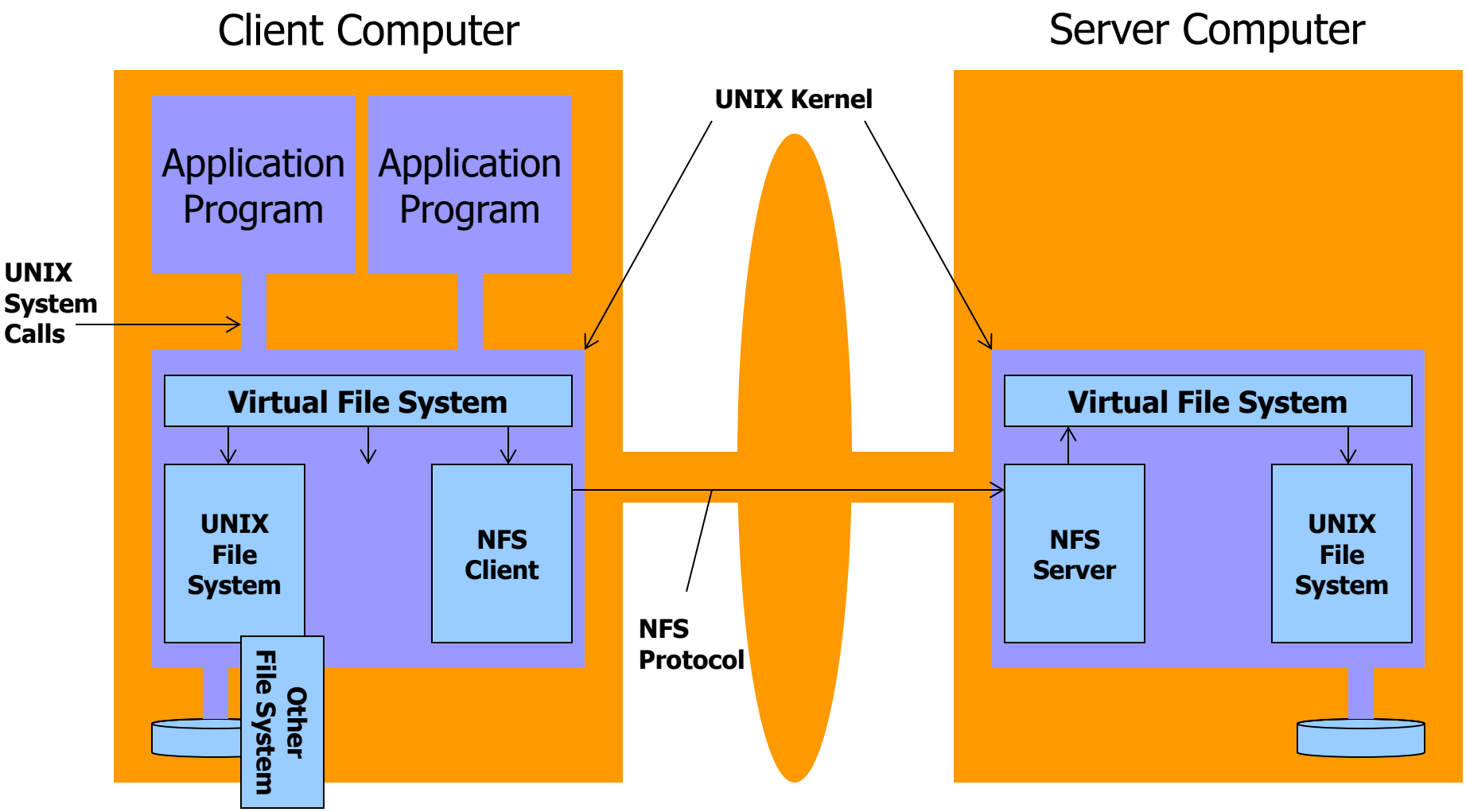
	FFS
Objective	Reference Architecture
Security	Directory Service/RPC
	Encryption
Naming	Location Transparency
File Access	Remote Access Model + Cache
Cache-Update Policy	Write-Through
	N/A
Consistency	N/A
Service Type	Stateless
Replication	N/A

Distributed Systems: Sun NFS

Sun's Network File System (NFS)

- Developed to support file system sharing between networked workstations using a client-server model.
  - Each work station may be both a client and a server.
- Client is Integrated into the kernel:
  - No recompilation necessary
  - Single client module serving all user-level processes
- Employs 2 Key Protocols (RFC 1813):
  - The Mount Protocol
  - The NFS Protocol
- Employs Stateless operations

NFS Architecture



Virtual File System (VFS)

- Key component of the UNIX kernel.
  - Supports access transparency through separation of generic file system operations from implementation.

- Keeps track of what file systems are currently available both locally and remotely.
  - Responsible for invoking the relevant file system module.

- Based on file implementation structure called a *vnode* that combines an indicator that determines whether the file is local or remote with a unique file reference:
  - For local files the file reference is an *inode* identifier
  - For remote files, the file reference is a *NFS file handle*

NFS File Handles

- Files in NFS are referenced by a unique **file handle:**

- The **file system number** is a unique number that is assigned to each device / partition.
  - NFS Mounts are treated as a filesystem

- The **inode generation number** is needed because UNIX reuses inodes after a file is deleted.
  - Updating of this number is managed at the VFS layer

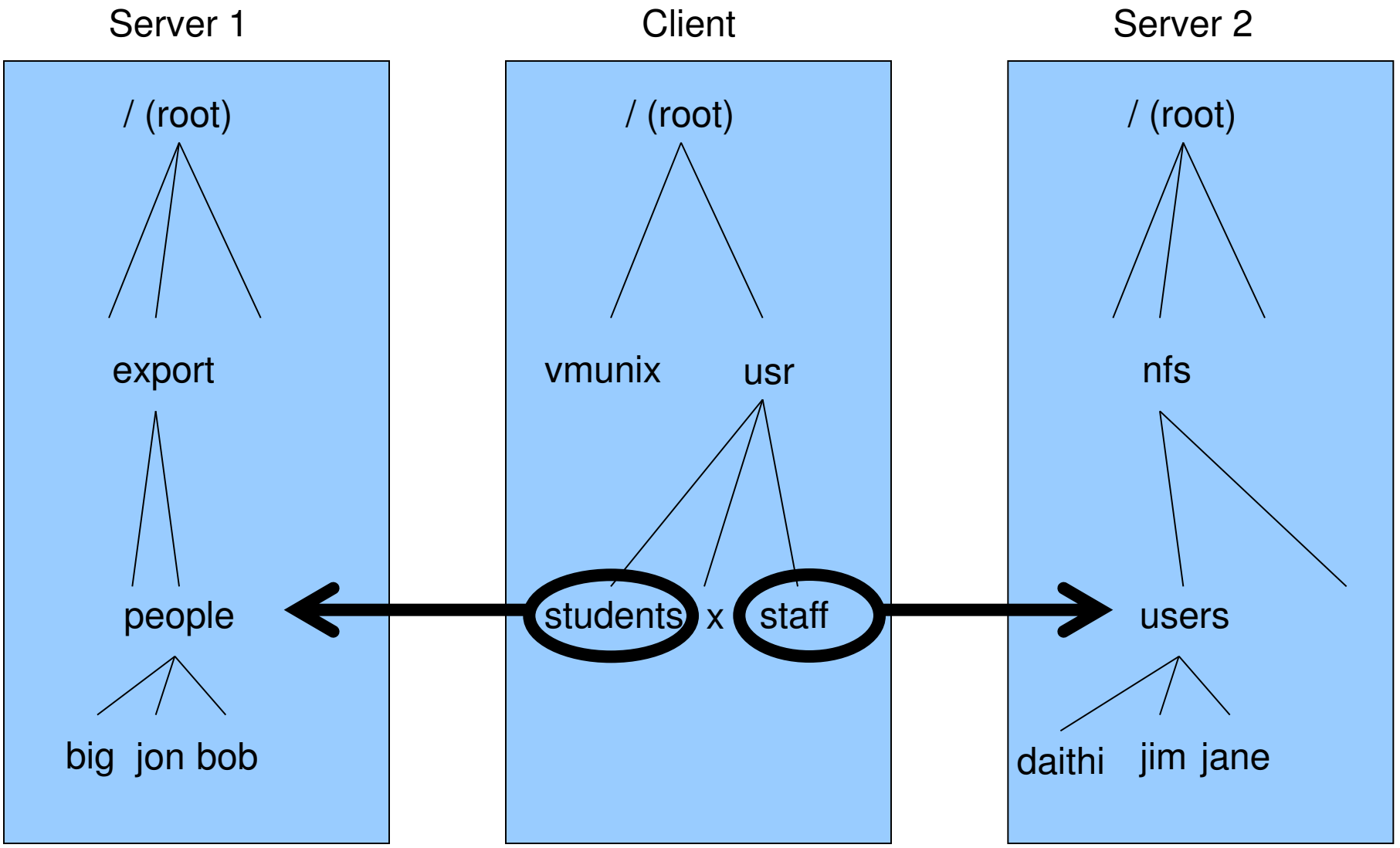
The NFS Protocol

- Set of RPCs for remote file operations (RFC 1813):
  - Lookup(dirfh, name) -> fh, attr
  - Create(dirfh, name, attr) -> newfh, attr
  - Remove(dirfh, name) -> status
  - Read(fh, offset, count) -> attr, data
  - Write(fh, offset, count, data) -> attr
  - …
- No Open or Close operations!
- Modified data must be committed to the server's disk before the call returns results to the client.
- No Concurrency control mechanism
  - Users coordinate access outside the NFS mechanism.

The Mount Protocol

- Clients initiate a connection to an NFS server via the UNIX mount command:
  - e.g. `mount krytan.ucd.ie:/home/daithi /mnt/shared`
- This command invokes an RPC provided by the NFS Client via the VFS.
  - This RPC transmits a request to the target machine requesting that the specified directory be mounted.
  - The Mount Service validates the request and responds with the file handle of the specified directory if successful.
- Shared files are listed in the “/etc/exports” file and access lists are provided to further constrain which hosts can mount the filesystem.

Mounting


Server 1
Client
Server 2
/ (root)
/ (root)
/ (root)
export
vmunix
usr
nfs
people
students
x
staff
users
big
jon
bob
daithi
jim
jane

Hard and Soft Mounting

- *Hard Mounted Filesystems:*
  - User level processes must be suspended until the request is completed
  - Requests are retried until they succeed
  - Failure causes the client process to “hang”

- *Soft Mounted Filesystems:*
  - NFS Client performs a small number of retries, and then returns a failure code.
  - Processes that do not detect this code exhibit unpredictable behaviour after failure…

- Many applications do not detect this failure code, resulting in most NFS mounts being hard mounted in practice.

Pathname Translation

- Pathnames may be a combination of local and remote directories.
  - E.g. /usr/staff/daithi

- VFS breaks the path into component names and does a lookup for every components name using its parents vnode.
  - Local component inode identifiers are located using the local file system
  - Remote component file handles are located using the NFS lookup RPC.
  - A directory name lookup cache at the client holds the vnodes for remote directory names.

Server Caching

- NFS Server maintains an in-memory buffer cache.
  - Contains file pages and directory attributes and file attributes.
  - Pages are maintained until buffer capacity reached.
  - Buffered pages are transmitted without disk access.
  - Read-ahead is used to anticipate future reads.
  - Delayed-write is used to write buffer data to the hard disk.
  - The buffer also flushes buffered pages every 30 seconds (sync).

- Supported Cache Update Strategies include:
  - Write-through cache updates.
  - Write-on-commit cache updates.

Client Caching

- NFS Client caches the results of the read, write, getattr and lookup operations.

- Timestamps are used to validate cached data:
  - A cache entry is valid at time T for a given timestamp (Tc) if

$$(T-Tc) < t$$

where t is a **freshness interval**.

  - In addition, periodic checks are carried out where the client polls the server to check whether Tc still matches the relevant timestamp on the server (Tm).

- To improve efficiency, t is set adaptively for individual files based on usage.

Client Caching

- Consistency is client-initiated:
- Validity must be checked before every NFS file operation.
- The adopted policy is:
  - If the freshness property is true, then attempt the operation
  - If the freshness property is false, retrieve the server timestamp (Tm) and perform a timestamp check.
  - If the timestamp check fails, then retrieve an up-to-date copy of the cached file page
- This does not solve the problem where cached data is modified!
- As a result, the client cache uses write-through to minimise the potential for consistency problems.

Security

- NFS is stateless – each RPC must include authentication information.

- (Recap) By default, the Sun RPC Protocol requires a user id and group id
  - This has been extended to support encryption
  - Also, can be integrated with Kerberos (as with FFS)

- Any security system that can be applied to the Sun RPC Protocol can be used with NFS!

Sun NFS Review

	FFS	Sun NFS
Objective	Reference Architecture	Simple and Reliable
Security	Directory Service/RPC	Sun RPC-based
	Encryption	Kerberos Integration
Naming	Location Transparency	Location Transparency
File Access	Remote Access Model + Cache	Remote Access Model + Cache
Cache-Update Policy	Write-Through	Periodic Flush w/ Disk Sync
	N/A	Write-Through / Write on Commit
Consistency	N/A	Timestamp-based
Service Type	Stateless	Stateless
Replication	N/A	N/A

Next week attraction... Andrew File System