CS 202: Advanced Operating Systems

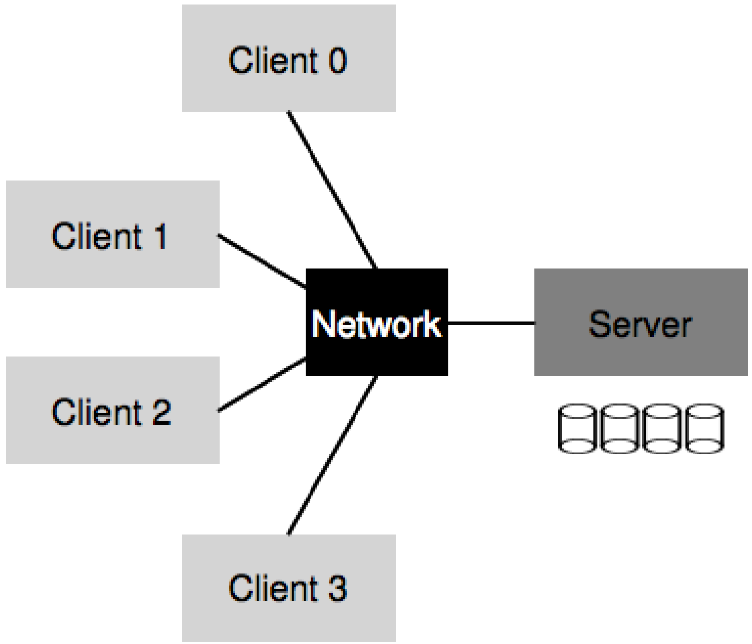
Distributed Filesystems

Credit: Uses some slides by Jehan-Francois Paris, Mark Claypool and Jeff Chase

DESIGN AND IMPLEMENTATION OF THE SUN NETWORK FILESYSTEM

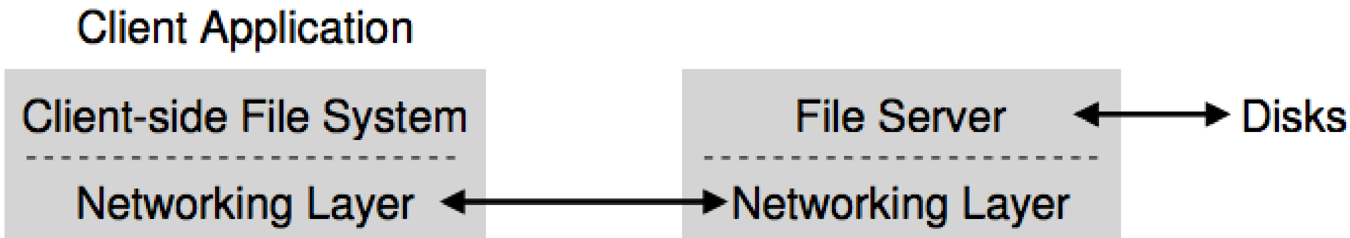
R. Sandberg, D. Goldberg
S. Kleinman, D. Walsh, R. Lyon
Sun Microsystems

What is NFS?



- First commercially successful network file system:
  - Developed by Sun Microsystems for their diskless workstations
  - Designed for robustness and "adequate performance"
  - Sun published all protocol specifications
  - Many many implementations

Overview and Objectives



- Fast and efficient crash recovery
  - Why do crashes occur?
- To accomplish this:
  - NFS is stateless – key design decision
    - All client requests must be self-contained
  - The virtual filesystem interface
    - VFS operations
    - VNODE operations

Additional objectives

- ***Machine and Operating System Independence***
  - Could be implemented on low-end machines of the mid-80's
- ***Transparent Access***
  - Remote files should be accessed in exactly the same way as local files
- *UNIX semantics should be maintained on client*
  - Best way to achieve transparent access
- *"Reasonable" performance*
  - Robustness and preservation of UNIX semantics were much more important

Example

```
char buffer[MAX];
int fd = open("foo", O_RDONLY); // get descriptor "fd"
read(fd, buffer, MAX);          // read MAX bytes from foo (via fd)
read(fd, buffer, MAX);          // read MAX bytes from foo
...
read(fd, buffer, MAX);          // read MAX bytes from foo
close(fd);                      // close file
```

- What if the client simply passes the open request to the server?
  - Server has state
  - Crash causes big problems
- Three important parts
  - The protocol
  - The server side
  - The client side

The protocol (I)

- Uses the Sun RPC mechanism and Sun eXternal Data Representation (XDR) standard
- Defined as a set of remote procedures
- Protocol is ***stateless***
  - Each procedure call contains ***all the information necessary to complete the call***
  - Server maintains no “between call” information

Advantages of statelessness

- Crash recovery is very easy:
  - When a server crashes, client just resends request until it gets an answer from the rebooted server
  - Client cannot tell difference between a server that has crashed and recovered and a slow server
- Client can always ***repeat any request***

NFS as a “Stateless” Service

- A classical NFS server maintains no in-memory hard state.
  - The only hard state is the stable file system image on disk.
  - no record of clients or open files
  - no implicit arguments to requests
    - *E.g., no server-maintained file offsets*: **read** and **write** requests must explicitly transmit the byte offset for each operation.
  - no write-back caching on the server
  - no record of recently processed requests
  - etc., etc....
- *Statelessness makes failure recovery simple and efficient.*

Consequences of statelessness

- Read and writes must specify their start offset
  - Server does not keep track of current position in the file
  - User still use conventional UNIX reads and writes
- Open system call translates into several lookup calls to server
- No NFS equivalent to UNIX close system call

Important pieces of protocol

```
NFSPROC_GETATTR
  expects: file handle
  returns: attributes
NFSPROC_SETATTR
  expects: file handle, attributes
  returns: nothing
NFSPROC_LOOKUP
  expects: directory file handle, name of file/directory to look up
  returns: file handle
NFSPROC_READ
  expects: file handle, offset, count
  returns: data, attributes
NFSPROC_WRITE
  expects: file handle, offset, count, data
  returns: attributes
NFSPROC_CREATE
  expects: directory file handle, name of file, attributes
  returns: nothing
NFSPROC_REMOVE
  expects: directory file handle, name of file to be removed
  returns: nothing
NFSPROC_MKDIR
  expects: directory file handle, name of directory, attributes
  returns: file handle
NFSPROC_RMDIR
  expects: directory file handle, name of directory to be removed
  returns: nothing
NFSPROC_READDIR
  expects: directory handle, count of bytes to read, cookie
  returns: directory entries, cookie (to get more entries)
```

From protocol to distributed file system

- Client side translates user requests to protocol messages to implement the request remotely
- Example:

Client	Server
fd = open("/foo", ...); Send LOOKUP (rootdir FH, "foo")	
	Receive LOOKUP request look for "foo" in root dir return foo's FH + attributes
Receive LOOKUP reply allocate file desc in open file table store foo's FH in table store current file position (0) return file descriptor to application	

The lookup call (I)

- Returns a **file handle** instead of a file descriptor
  - File handle specifies unique location of file
    - Volume identifier, inode number and generation number
- **lookup(dirfh, name)** *returns* **(fh, attr)**
  - Returns file handle **fh** and attributes of named file in directory **dirfh**
  - Fails if client has no right to access directory **dirfh**

The lookup call (II)

- One single open call such as

  fd = open("/usr/joe/6360/list.txt")

  will be result in several calls to lookup

  lookup(rootfh, "usr") returns (fh0, attr)
  lookup(fh0, "joe") returns (fh1, attr)
  lookup(fh1, "6360") returns (fh2, attr)
  lookup(fh2, "list.txt") returns (fh, attr)

- Why all these steps?
  - Any of components of /usr/joe/6360/list.txt could be a *mount point*
  - Mount points are *client dependent* and mount information is kept above the lookup() level

Server side (I)

- Server implements a write-through policy
  - Required by statelessness
  - Any blocks modified by a write request (including i-nodes and indirect blocks) must be written back to disk before the call completes

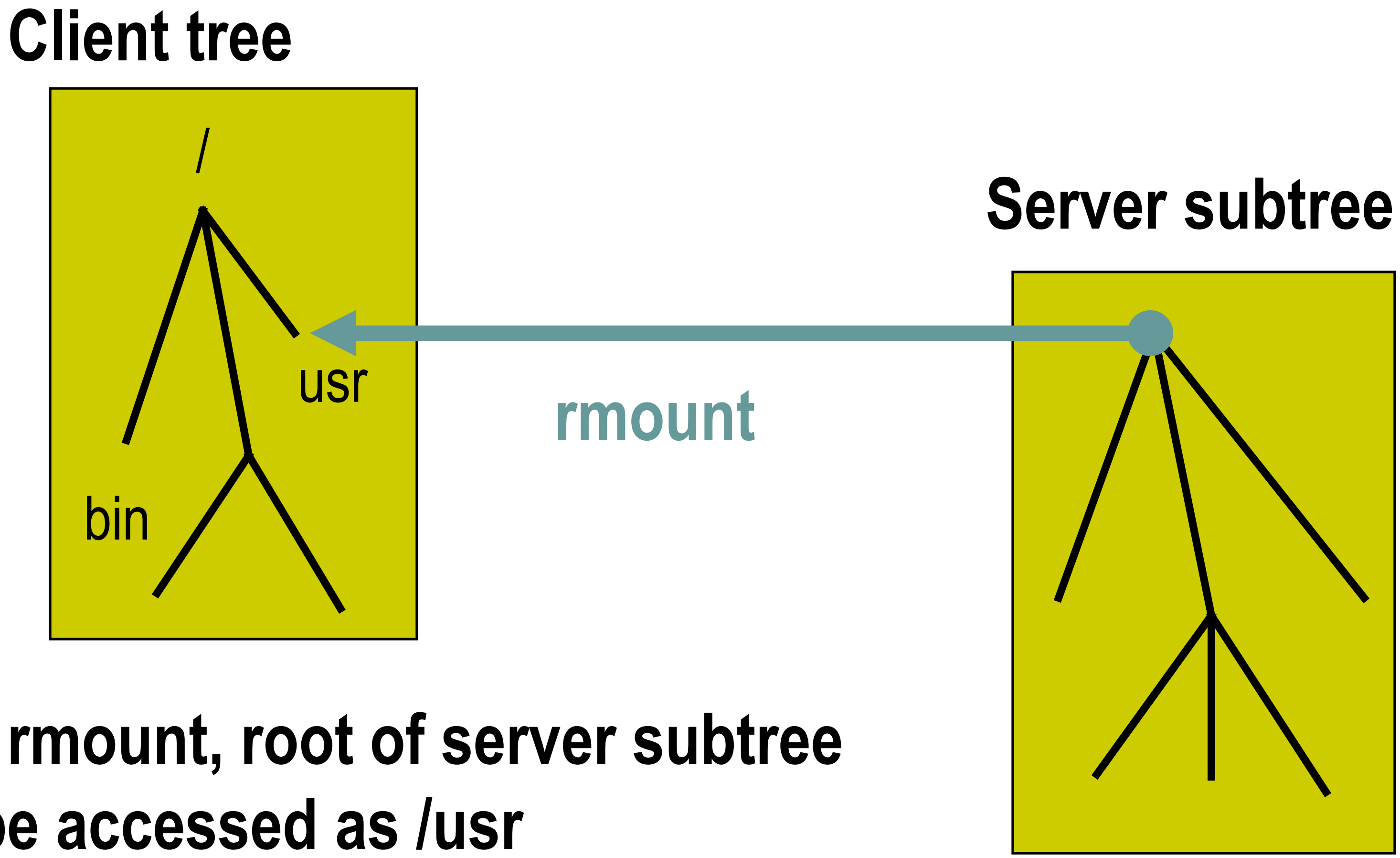
Server side (II)

- File handle consists of
  - Filesystem id identifying disk partition
  - I-node number identifying file within partition
  - Generation number changed every time i-node is reused to store a new file
- Server will store
  - Filesystem id in filesystem superblock
  - I-node generation number in i-node

Client side (I)

- Provides transparent interface to NFS
- Mapping between remote file names and remote file addresses is done a server boot time through ***remote mount***
  - Extension of UNIX mounts
  - Specified in a ***mount table***
  - Makes a remote subtree appear part of a local subtree

Remote mount

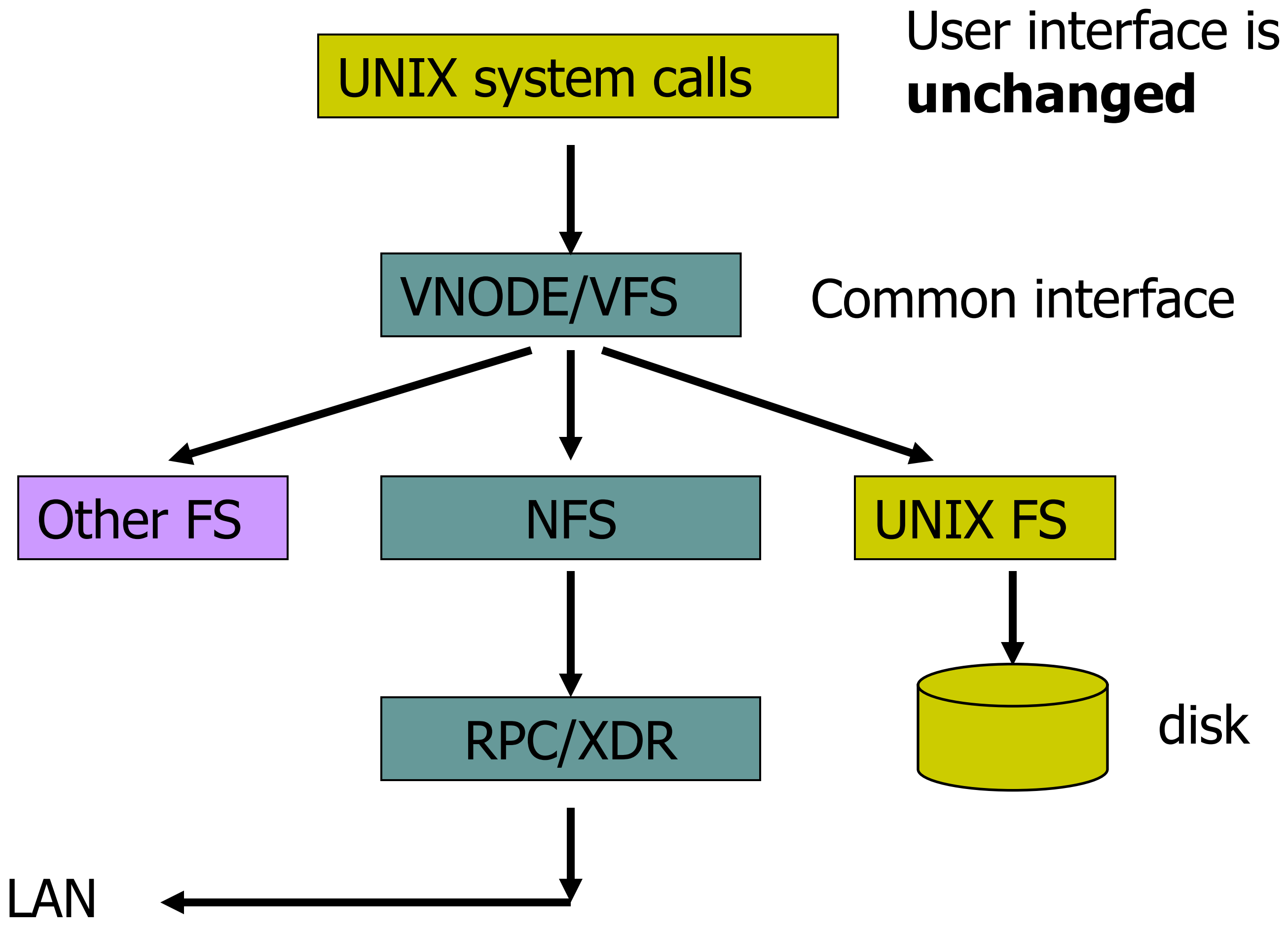


After rmount, root of server subtree can be accessed as /usr

Client side (II)

- Provides transparent access to
  - NFS
  - Other file systems (including UNIX FFS)
- New virtual filesystem interface supports
  - VFS calls, which operate on whole file system
  - VNODE calls, which operate on individual files
- Treats all files in the same fashion

Client side (III)



More examples

read(fd, buffer, MAX);

Index into open file table with fd
  get NFS file handle (FH)
  use current file position as offset
Send READ (FH, offset=0, count=MAX)

Receive READ request
  use FH to get volume/inode num
  read inode from disk (or cache)
  compute block location (using offset)
  read data from disk (or cache)
  return data to client

Receive READ reply
  update file position (+bytes read)
  set current file position = MAX
  return data/error code to app

Continued

read(fd, buffer, MAX);
Same except offset=MAX and set current file position = 2*MAX

---

read(fd, buffer, MAX);
Same except offset=2*MAX and set current file position = 3*MAX

---

close(fd);
Just need to clean up local structures
Free descriptor "fd" in open file table
(No need to talk to server)

Handling server Failures

- Failure types:

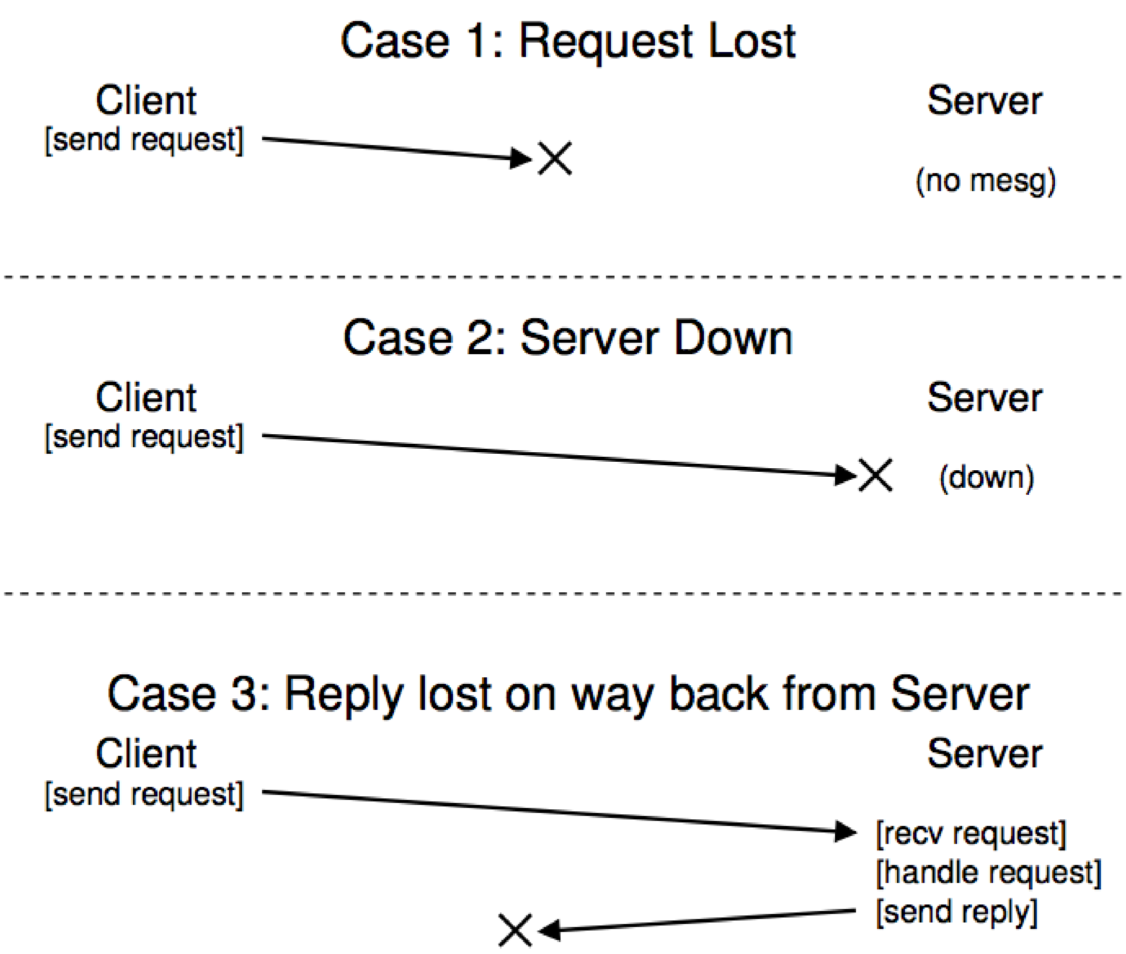


Figure 48.6: **The Three Types of Loss**

Idempotency

- A client handles all these failures by simply retrying the request
- Why can this approach work? These operations are idempotent:
  - performing an operation multiple times is equivalent to performing it one time
- Lookup, read, write are obviously idempotent
- What about delete, mkdir, exclusive create, append-mode write?

Client-side Caching

- Can greatly improve the performance
- But what about cache consistency?

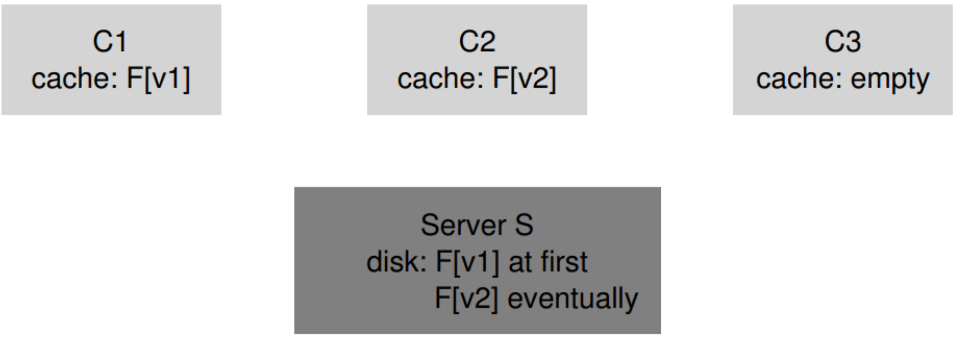


Figure 49.7: **The Cache Consistency Problem**

- Solution:
  - flush-on-close (a.k.a, close-to-open)
  - GETATTR (with an attribute cache)
  - What do we sacrifice?

Discussion

- Throughput
- Latency
- Scalability
- Crash Recovery
- Fault Tolerance