Distributed File Systems

- Issues in Distributed File Service
- Case Studies:
  - Sun Network File System
  - Coda File System
  - Web
- *Reading:*
  - *Coulouris: Distributed Systems, Addison Wesley, Chapters 7,8*
  - *Tanenbaum/van Steen: Distributed Systems, Prentice Hall, 2002, Chapter 10*
  - *A.S. Tanenbaum: Distributed Operating Systems, Prentice Hall, 1995, Chapter 5*

File Service Components

- File Service
  - Operations on individual files
- Directory Service
  - Manage directories
- Naming Service
  - Location independence: files can be moved without their names being changed.
  - Common approaches to file and directory naming:
    - Machine + path naming, e.g. */machine/path* or *machine:path*
    - Mounting remote file systems onto the local file hierarchy
    - A single name space that looks the same on all machines
  - Two-level naming: symbolic names as seen by user *vs.*binary names as seen by system.

Requirements

- Transparency:
  - Access transparency
  - Location transparency
  - Concurrency transparency
  - Failure transparency
  - Performance transparency
  - Replication transparency
  - Migration transparency
- Others:
  - Heterogeneity
  - Scalability
  - Support for fine-grained distribution of data
  - Partitions & disconnected operation

File Sharing

- What is the semantics of file operations in a distributed system? What is the problem?
- "Unix" semantics: the system enforces absolute time ordering on all operations and always returns the most recent value.
  - Straightforward for system with single server and no caching.
  - What about multiple servers or caching clients?
  - Relax semantics of file sharing.
- Session semantics:
  - Changes to an open file are initially visible only to the process that modified the file. Changes are propagated only when the file is closed.
  - What if two processes cache and modify the file?
- Immutable files:
  - Files are created and replaced, not modified.
  - Problem of concurrent operations simply disappears.
- Atomic Transactions:
  - BEGIN TRANSACTION / END TRANSACTION.
  - Transactions are executed indivisbly.

File Servers: System Structure

Separation of File Clients and File Servers?

Separation of File Service and Directory Service?

Where is State Information to be maintained?

stateless servers	*vs.*	"stateful" servers.
fault tolerance		shorter request messages
no OPEN/CLOSE calls		better performance
no server space wasted on tables		readahead possible
no limits on number of open files		idempotency easier
no problems if a client crashes		file locking possible

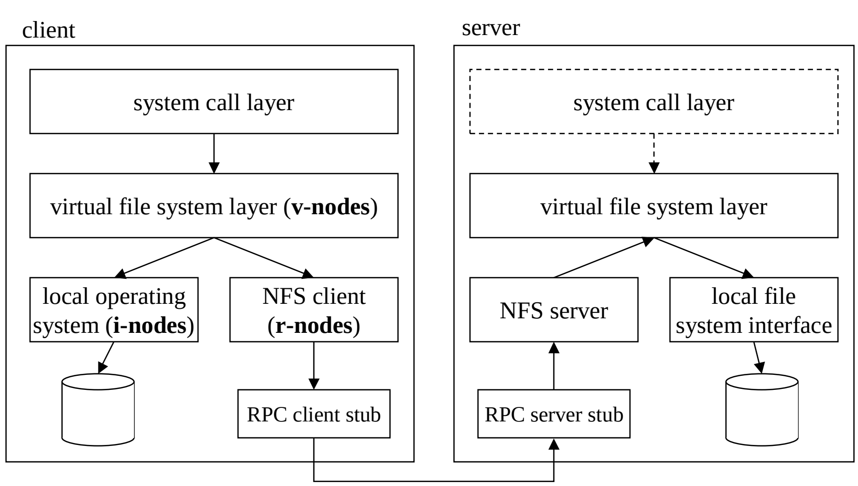
Aspects of Distributed File Systems

- Communication
- Processes
- Naming
- Synchronization
- Fault Tolerance
- Caching and Replication
- Security

Sun's Network File System (NFS)

- Architecture:
  - NFS as collection of protocols the provide clients with a distributed file system.
  - **Remote Access Model** (as opposed to **Upload/Download Model**)
  - Every machine can be both a client and a server.
  - Servers export directories for access by remote clients (defined in the `/etc/exports` file).
  - Clients access exported directories by mounting them remotely.
- Protocols:
  - mounting
    - Client sends a path name and server returns a file handle.
    - Static mounting (at boot-up) *vs.* automounting.
    - Hard mounting *vs.* soft mounting
  - file and directory access
    - Servers are stateless (no `OPEN`/`CLOSE` calls)

NFS: Basic Architecture



NFS Implementation: Issues

- File handles:
  - specify *filesystem* and *i-node number* of file
  - sufficient?
- Integration:
  - where to put NFS on client?
  - on server?
- Server caching:
  - *read-ahead*
  - *write-delayed* with periodic *sync* *vs.* *write-through*
- Client caching:
  - timestamps with validity checks

NFS: File System Model

- File system model similar to UNIX file system model
  - Files as uninterpreted sequences of bytes
  - Hierarchically organized into naming graph
  - NSF supports **hard links** and **symbolic links**
  - Named files, but access happens through **file handles**.

- File system operations
  - NFS Version 3 aims at statelessness of server
  - NFS Version 4 is more relaxed about this

NFS: File System Operations

Operation	v3	v4	Description
Create	Yes	No	Create a regular file
Create	No	Yes	Create a nonregular file
Link	Yes	Yes	Create a hard link to a file
Symlink	Yes	No	Create a symbolic link to a file
Mkdir	Yes	No	Create a subdirectory in a given directory
Mknod	Yes	No	Create a special file
Rename	Yes	Yes	Change the name of a file
Remove	Yes	Yes	Remove a file from a file system
Rmdir	Yes	No	Remove an empty subdirectory from a directory
Open	No	Yes	Open a file
Close	No	Yes	Close a file
Lookup	Yes	Yes	Look up a file by means of a file name
Readdir	Yes	Yes	Read the entries in a directory
Readlink	Yes	Yes	Read the path name stored in a symbolic link
Getattr	Yes	Yes	Get the attribute values for a file
Setattr	Yes	Yes	Set one or more attribute values for a file
Read	Yes	Yes	Read the data contained in a file
Write	Yes	Yes	Write data to a file

Figure 10-3. An incomplete list of file system operations supported by NFS.

NFS: Communication

- OS independence achieved through use of RPC.
- Every NFS operation can be implemented through separate RPC call.
  - e.g. lookup / read in Version 3
- **Compound procedures** in Version 4
  - e.g. lookup / open / read can be combined in single request/reply.
- Compound procedures have no transactional semantics.
  - IOWs: No measures are taken to avoid conflicts by concurrent operations from other clients.

NFS: Processes

- Client – Server
- Stateless servers in Version 3
  - File locking?
    - Separate **Lock Manager**
  - Authentication?
  - Caching?
- Version 4: stateless approach abandoned

NFS: File Locking

- Version 3: locking handled by separate (stateful) **lock manager**.
  - What if clients or servers fail while locks are being held?
  - Need proper recovery schemes.
- Version 4: Locking integrated into file access protocol:
  - Operations: lock, lockt, locku, renew
  - Nonblocking lock; requires polling, but can ask to temporarily keep request in FIFO queue at server.
  - Locks are granted for a specific time (lease); simplifies recovery.
- Share Reservation in NFS for Window-based systems

NFS: Client Caching

- Potential for inconsistent versions at different clients.
- Solution approach:
  - Whenever file cached, timestamp of last modification on server is cached as well.
  - Validation: Client requests latest timestamp from server (*getattributes*), and compares against local timestamp. If fails, all blocks are invalidated.
- Validation check:
  - at file open
  - whenever server contacted to get new block
  - after timeout (3s for file blocks, 30s for directories)
- Writes:
  - block marked dirty and scheduled for flushing.
  - flushing: when file is closed, or a *sync* occurs at client.
- Time lag for change to propagate from one client to other:
  - delay between write and flush
  - time to next cache validation

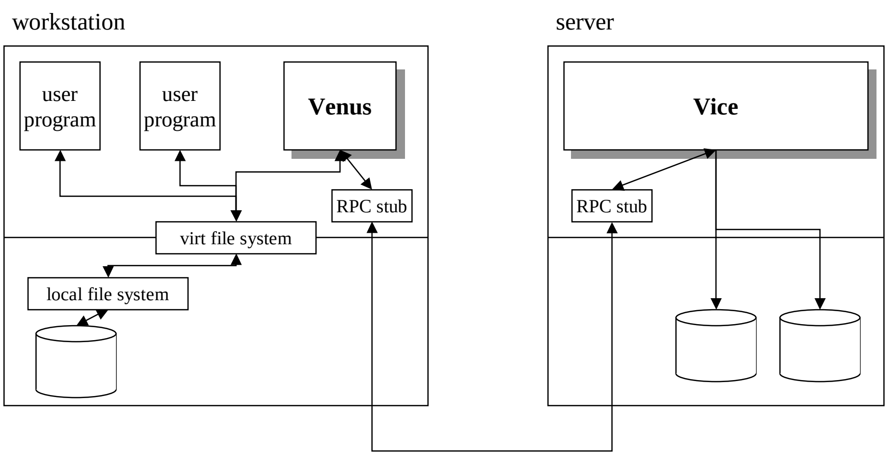
NFS: Fault Tolerance

- RPC Failures:
  - When reply is lost, retransmission may trigger multiple invocations of requests.
  - Problem solved with **duplicate-request cache** and **transaction identifiers**.
- Fault tolerance becomes an issue when servers start becoming stateful in Version 4.
- File Locking Failures:
  - Client crashes: associate **lease** with locks.
    - Locks can only held until lease expires. Leases can be renewed by server.
    - After recovery, leases may only be renewed during a grace period; no new leases are given out.
  - False removal of leases due to network partitions (unaddressed)
    - Lease renewals don’t make it to the lock holder.

The Coda File System

- Descendant of CMU's Andrew File System (AFS)
- AFS' Design for Scalability
  - Whole-file serving:
    - on opening a file, the entire file is transferred to client
  - Whole-file caching:
    - persistent cache contains most recently used files on that computer.
  - Observations:
    - shared files updated infrequently
    - working set of single user typically fits into cache on local machine
    - file access patterns
    - what about transactional data (databases)
- Coda/AFS Architecture:
  - Small number of dedicated **Vice** file servers.
  - Much larger collection of **Virtue** workstations give users and processes access to the file system.
  - Coda provides globally shared name space.

Internal Organization



CODA: Communication

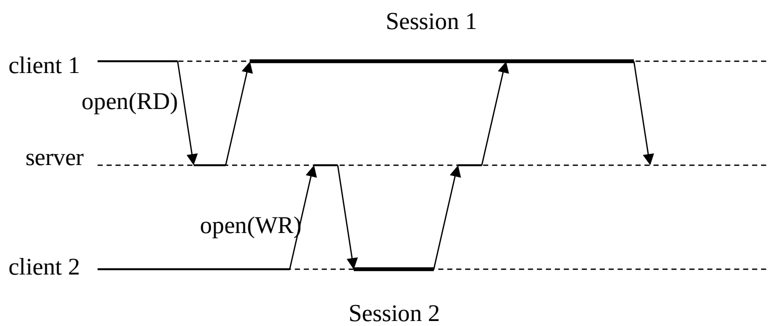
- Interprocess communication using RCP2
  (http://www.coda.cs.cmu.edu/doc/html/rpc2_manual.html)
- RPC2 provides reliable RPC over UDP.
- Support for **Side Effects**
  - RPC connections may be associated with *Side-Effects* to allow application-specific network optimizations to be performed. An example is the use of a specialized protocol for bulk transfer of large files. Detailed information pertinent to each type of side effect is specified in a *Side Effect Descriptor*.
  - Adding support for a new type of side effect is analogous to adding a new device driver in Unix. To allow this extensibility, the RPC code has hooks at various points where side-effect routines will be called. Global tables contain pointers to these side effect routines. The basic RPC code itself knows nothing about these side-effect routines.
- Support for MultiRPC (enables for parallel calls, e.g. invalidations)

Coda: Processes

- Clear distinction between client and server processes
- Venus processes represent clients.
- Vice processes represent servers.
- All processes realized as collection of user-level threads.
- Additional low-level thread handles I/O operations (why?)

Coda: Synchronization

- Attempt to provide transactional semantics (weaker than normally supported by transactions)
- Problem: Continue to provide uninterrupted file service when servers are temporarily unavailable (failure, partition, disconnection)



Opening a File in AFS

- User process issues `open(FileName, mode)` call.
- UNIX kernel passes request to Venus if file is shared.
- Venus checks if file is in cache. If not, or no valid *callback promise*, gets file from Vice.
- Vice copies file to Venus, with a *callback promise*. Logs callback promise.
- Venus places copy of file in local cache.
- UNIX kernel opens file and returns file descriptor to application.

Cache Coherency

- Callback promise:
  - Token from Vice server.
  - Guarantee that Venus will be notified if file is modified.
- 2 states:
  - valid:callback promise as received from server upon open call.
  - cancelled: callback was issued when somebody else issued an update to file (callback break).
- Callback promise is checked whenever client opens file in cache.
- What about callbacks that are lost?
- Callback renewals with current timestamp of file.