Distributed File Systems

- Issues in Distributed File Service
- Case Studies:
  - Sun Network File System
  - CMU Andrew File System
  - Coda File System
  - Web?
- *Reading:*
  - *Coulouris: Distributed Systems, Addison Wesley, Chapters 7,8*
  - *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

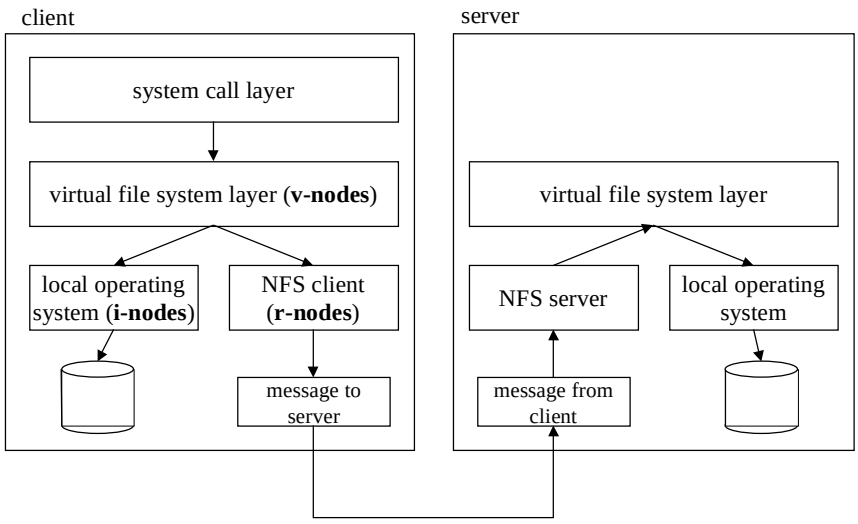
- Separation of file clients and file servers?
- Separation of file service and directory service?
- Where is state information to be maintained?

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

Sun's Network File System (NFS)

- Architecture:
  - Arbitrary collection of clients and servers share a common file system.
  - 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 Implementation



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 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

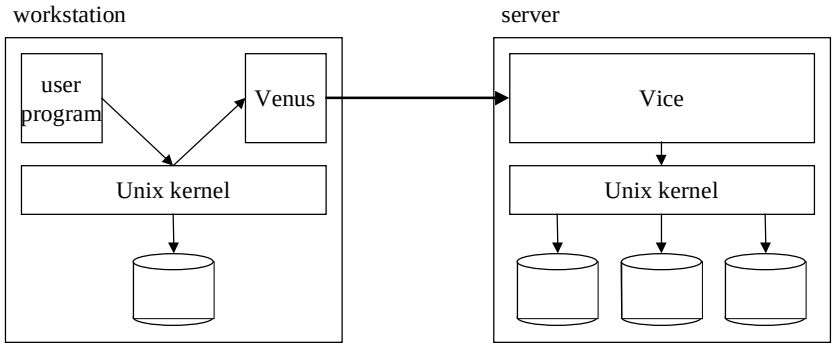
Andrew File System (AFS)
Design for Scalability

- 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)

Andrew File System (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)

AFS Implementation



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 issued an update to file.
- Callback promise is checked whenever client opens file in cache.
- What about callbacks that are lost?
- Callback renewals with current timestamp of file.