

## Distributed Computing Systems

### Distributed File Systems

## Distributed File Systems

- Early networking and files
  - Had FTP to transfer files
  - Telnet to remote login to other systems with files
- But want more transparency!
  - local computing with remote file system
- Distributed file systems → One of earliest distributed system components
- Enables programs to access remote files as if local
  - Transparency
- Allows sharing of data and programs
- Performance and reliability comparable to local disk

## Outline

- Overview (done)
- Basic principles (next)
  - Concepts
  - Models
- Network File System (NFS)
- Andrew File System (AFS)
- Dropbox

## Concepts of Distributed File System

- Transparency
- Concurrent Updates
- Replication
- Fault Tolerance
- Consistency
- Platform Independence
- Security
- Efficiency

## Transparency

Illusion that all files are similar. Includes:

- *Access transparency* — a single set of operations. Clients that work on local files can work with remote files.
- *Location transparency* — clients see a uniform name space. Relocate without changing path names.
- *Mobility transparency* —files can be moved without modifying programs or changing system tables
- *Performance transparency* —within limits, local and remote file access meet performance standards
- *Scaling transparency* —increased loads do not degrade performance significantly. Capacity can be expanded.

5

## Concurrent Updates

- Changes to file from one client should not interfere with changes from other clients
  - Even if changes at same time
- Solutions often include:
  - File or record-level locking

6

## Replication

- File may have several copies of its data at different locations
  - Often for performance reasons
  - Requires update other copies when one copy is changed
- Simple solution
  - Change master copy and periodically refresh the other copies
- More complicated solution
  - Multiple copies can be updated independently at same time needs finer grained refresh and/or merge

7

## Fault Tolerance

- Function when clients or servers fail
- Detect, report, and correct faults that occur
- Solutions often include:
  - Redundant copies of data, redundant hardware, backups, transaction logs and other measures
  - Stateless servers
  - Idempotent operations

8

## Consistency

- Data must always be complete, current, and correct
- File seen by one process looks the same for all processes accessing
- Consistency special concern whenever data is duplicated
- Solutions often include:
  - Timestamps and ownership information

9

## Platform Independence

- Access even though hardware and OS completely different in design, architecture and functioning, from different vendors
- Solutions often include:
  - Well-defined way for clients to communicate with servers

10

## Security

- File systems must be protected against unauthorized access, data corruption, loss and other threats
- Solutions include:
  - Access control mechanisms (ownership, permissions)
  - Encryption of commands or data to prevent “sniffing”

11

## Efficiency

- Overall, want same power and generality as local file systems
- Early days, goal was to share “expensive” resource → the disk
- Now, allow convenient access to remotely stored files

12

## Outline

- Overview (done)
- Basic principles (next)
  - Concepts
  - Models
- Network File System (NFS)
- Andrew File System (AFS)
- Dropbox

## File Service Models

### Upload/Download Model

- Read file: copy file from server to client
- Write file: copy file from client to server
- Good
  - Simple
- Bad
  - Wasteful – what if client only needs small piece?
  - Problematic – what if client doesn't have enough space?
  - Consistency – what if others need to modify file?

### Remote Access Model

- File service provides functional interface
  - Create, delete, read bytes, write bytes, ...
- Good
  - Client only gets what's needed
  - Server can manage coherent view of file system
- Bad
  - Possible server and network congestion
    - Servers used for duration of access
    - Same data may be requested repeatedly

## Semantics of File Service

### Sequential Semantics

*Read returns result of last write*

- Easily achieved if
  - Only one server
  - Clients do not cache data
- But
  - Performance problems if no cache
  - Can instead write-through
    - Must notify clients holding copies
    - Requires extra state, generates extra traffic

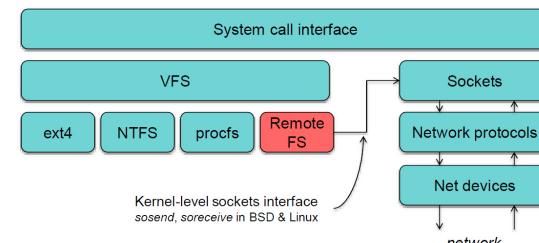
### Session Semantics

*Relax sequential rules*

- Changes to open file are initially visible only to process that modified it
- Last process to modify file “wins”
- Can hide or lock file under modification from other clients

## Accessing Remote Files (1 of 2)

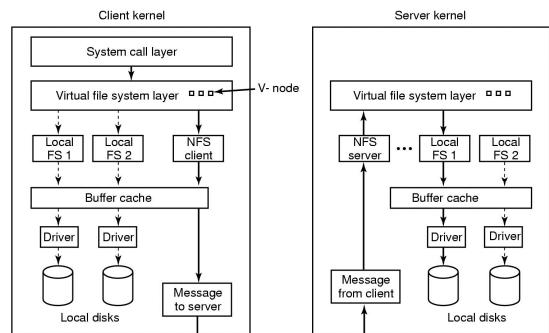
- For transparency, implement client as module under VFS



(Additional picture next slide)

## Accessing Remote Files (2 of 2)

Virtual file system allows for transparency



## Stateful or Stateless Design

### Stateful

*Server maintains client-specific state*

- Shorter requests
- Better performance in processing requests
- Cache coherence possible
  - Server can know who's accessing what
- File locking possible

### Stateless

*Server maintains no information on client accesses*

- Each request must identify file and offsets
- Server can crash and recover
  - No state to lose
- No open/close needed
  - They only establish state
- No server space used for state
  - Don't worry about supporting many clients
- Problems if file is deleted on server
- File locking not possible

## Caching

- Hide latency to improve performance for repeated accesses
- Four places:
  - Server's disk
  - Server's buffer cache (memory)
  - Client's buffer cache (memory)
  - Client's disk
- Client caches risk cache consistency problems

## Concepts of Caching (1 of 2)

### Centralized control

- Keep track of who has what open and cached on each node
- Stateful file system with signaling traffic

### Read-ahead (pre-fetch)

- Request chunks of data before needed
- Minimize wait when actually needed
- But what if data pre-fetched is out of date?

## Concepts of Caching (2 of 2)

### Write-through

- All writes to file sent to server
  - What if another client reads its own (out-of-date) cached copy?
- All accesses require checking with server
- Or ... server maintains state and sends invalidations

### Delayed writes (write-behind)

- Only send writes to files in batch mode (i.e., buffer locally)
- One bulk write is more efficient than lots of little writes
- Problem: semantics become ambiguous
  - Watch out for consistency – others won't see updates!

### Write on close

- Only allows session semantics
- If lock, must lock whole file

## Outline

- Overview (done)
- Basic principles (done)
- Network File System (NFS) (next)
- Andrew File System (AFS)
- Dropbox

## Network File System (NFS)

- Introduced in 1984 (by Sun Microsystems)
- Not first made, but first to be used as product
- Made interfaces in public domain
  - Allowed other vendors to produce implementations
- Internet standard is NFS protocol (version 3)
  - [RFC 1913](#)
- Still widely deployed, up to v4 but maybe too bloated so v3 widely used

## NFS Overview

- Provides transparent access to remote files
  - Independent of OS (e.g., Mac, Linux, Windows) or hardware
- Symmetric – any computer can be server *and* client
  - But many institutions have dedicated server
- Export some or all files
- Must support diskless clients
- Recovery from failure
  - Stateless, UDP, client retries
- High performance
  - Caching and read-ahead

## Underlying Transport Protocol

- Initially NSF ran over UDP using Sun RPC
- Why UDP?
  - Slightly faster than TCP
  - No connection to maintain (or lose)
  - NFS is designed for Ethernet LAN
    - Relatively reliable
  - Error detection but no correction
    - NFS retries requests

## NSF Protocols

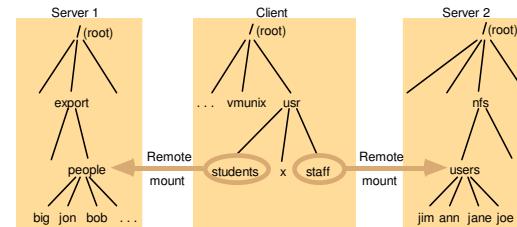
- Since clients and servers can be implemented for different platforms, need *well-defined* way to communicate → Protocol
  - Protocol – agreed upon set of requests and responses between client and servers
- Once agreed upon, Apple implemented Mac NFS client can talk to a Sun implemented Solaris NFS server
- NFS has two main protocols
  - Mounting Protocol*: Request access to exported directory tree
  - Directory and File Access Protocol*: Access files and directories (read, write, mkdir, readdir ... )

## NFS Mounting Protocol

- Request permission to access contents at pathname
- Client
  - Parses pathname
  - Contacts server for file handle
- Server
  - Returns file handle: file device #, i-node #, instance #
- Client
  - Create in-memory VFS i-node at mount point
  - Internally point to r-node for remote files
    - Client keeps state, not server
- Soft-mounted* – if client access fails, throw error to processes. But many do not handle file errors well
- Hard-mounted* – client blocks processes, retries until server up (can cause problems when NFS server down)

## NFS Architecture

- In many cases, on same LAN, but not required
  - Can even have client-server on same machine
- Directories available on server through `/etc/exports`
  - When client mounts, becomes part of directory hierarchy



File system mounted at `/usr/students` is sub-tree located at `/export/people` in Server 1, and file system mounted at `/usr/staff` is sub-tree located at `/nfs/users` in Server 2

## Example NFS exports File

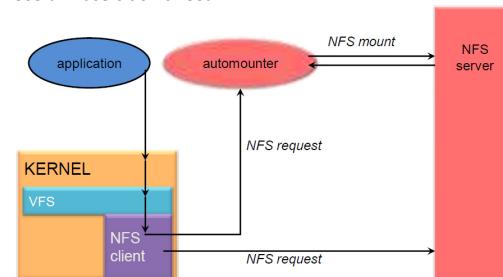
- File stored on server, typically /etc/exports

```
# See exports(5) for a description.
/public 192.168.1.0/255.255.255.0 (rw,no_root_squash)
```

- Share the folder /public
- Restrict to 192.168.1.0/24 Class C subnet
  - Use '\*' for wildcard/any
- Give read/write access
- Allow the root user to connect as root

## NFS Automounter

- Automounter – only mount when access empty NFS-specified dir
  - Attempt unmount every 5 minutes
  - Avoids long start up costs when many NFS mounts
  - Useful if users don't need

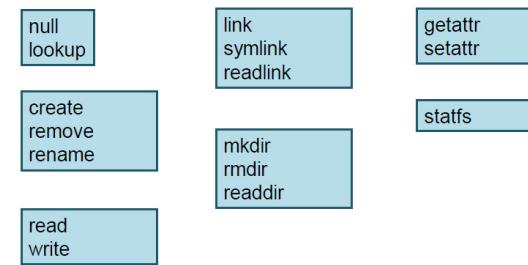


## NFS Access Protocol

- Most file operations supported from server to client (e.g., `read()`, `write()`, `getattr()`)
- First, perform lookup RPC
  - Returns file handle and attributes (not like `open()` since no information stored on server)
- Doesn't support `open()` and `close()`
  - Instead on, say, `read()`, client sends RPC handle, UFID and offset
- Allows server to be *stateless*, not remember connections
  - Better for scaling and robustness
- However, typical Unix can lock file on `open()`, unlock on `close()`
  - NFS must run separate lock daemon

## NFS Access Operations

- NFS has 16 functions (v2, v3 added six more)



## NFS Caching - Server

- Keep file data in memory as much as possible (avoid slow disk)
- *Read-ahead* – get subsequent blocks (typically 8 KB chunk) before needed
- Delayed write – only put data on disk when memory cache (typically every 30 seconds)
- Server responds by *write-through* (data to disk when client asks)
  - Performance can suffer, so another option only when file closed, called *commit*

## NFS Caching - Client

- Reduce number of requests to server (avoid slow network)
- Cache – `read`, `write`, `getattr`, `readdir`
- Can result in different versions at client
  - Validate with timestamp
  - When contact server (`open()` or new block), invalidate block if server has newer timestamp
- Clients responsible for polling server
  - Typically 3 seconds for file
  - Typically 30 seconds for directory
- Send written (dirty) blocks every 30 seconds
  - Flush on `close()`

## Improve Read Performance

- Transfer data in large chunks
  - 8K bytes default (that used to be large)
- Read-ahead
  - Optimize for sequential file access
  - Send requests to read disk blocks before requested by process

## Problems with NFS

- File consistency (client caches)
- Assumes clocks are synchronized
- No locking
  - Separate lock manager needed, but adds state
- No reference count for open files
  - Could delete file that others have open!
- File permissions may change
  - Invalidating access

## NFS Version 3

- TCP support
  - UDP caused more problems on WANs (errors)
  - All traffic can be multiplexed on one connection
    - Minimizes connection setup
- Large-block transfers
  - Negotiate for optimal transfer size
  - No fixed limit on amount of data per request

## NFS Version 4

- Adds state to system
- Supports `open()` operations since can be maintained on server
- Read operations not absolute, but relative, and don't need all file information, just handle
  - Shorter messages
- Locking integrated
- Includes optional security/encryption

## Outline

- Overview (done)
- Basic principles (done)
- Network File System (NFS) (done)
- Andrew File System (AFS) (next)
- Dropbox

## Andrew File System (AFS)

- Developed at CMU (hence the “Andrew” from “Andrew Carnegie”)
  - Commercialized through IBM to OpenAFS (<http://openafs.org/>)
- Transparent access to remote files
- Using Unix-like file operations (`creat`, `open`, ...)
- But AFS differs markedly from NFS in design and implementation...

## General Observations Motivating AFS

- For Unix users
  - Most files are small, less than 10 KB in size
  - `read()` more common than `write()` - about 6x
  - Sequential access dominates, random rare
  - Files referenced in bursts – used recently, will likely be used again
- Typical scenarios for most files:
  - Many files for one user only (i.e., not shared), so no problem
  - Shared files that are infrequently updated to others (e.g., code, large report) no problem
- Local cache of few hundred MB enough of a working set for most users
- What doesn't fit? → databases – updated frequently, often shared, need fine-grained control
  - Explicitly, AFS *not* for databases

## AFS Design

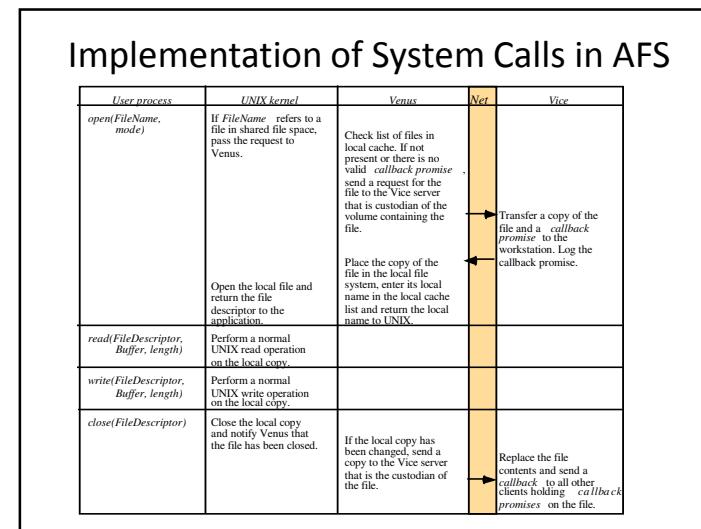
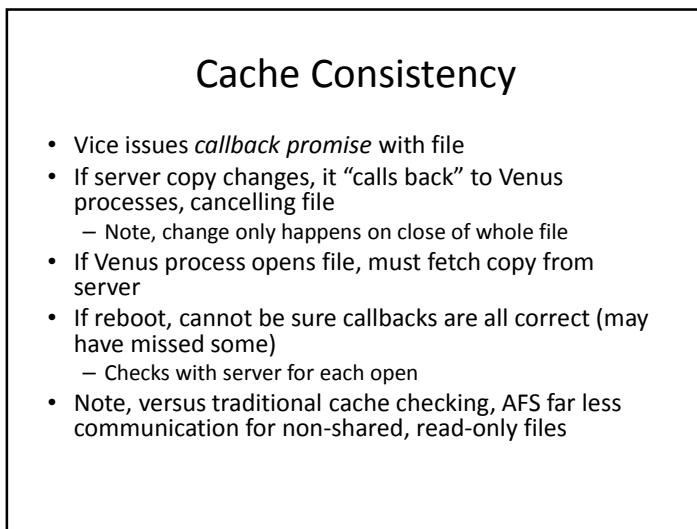
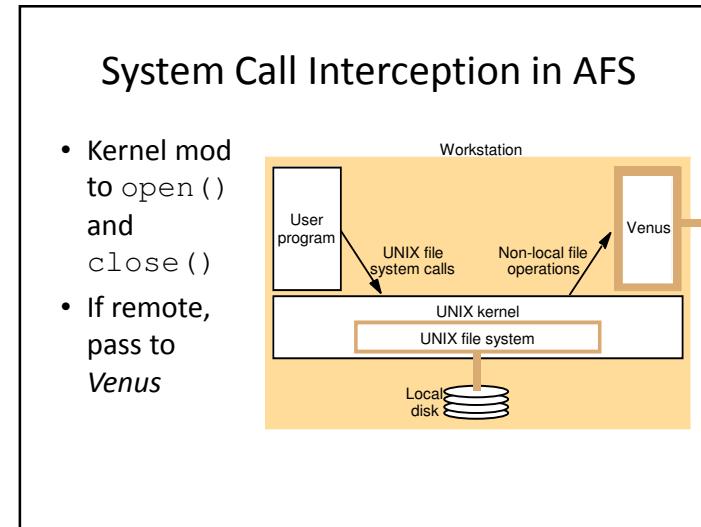
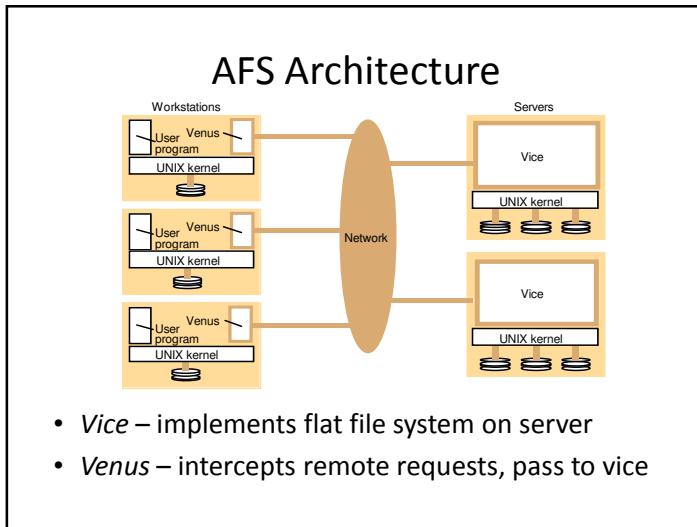
- Scalability is most important design goal
  - More users than other distributed systems
- Key strategy is caching of whole files at clients
  - *Whole-file serving* – entire file and directories
  - *Whole-file caching* – clients store cache on disk
    - Typically several hundred
    - Permanent so still there if rebooted

## AFS Example

- Process at client issues `open()` system call
- Check if local cached copy (Yes, then use. No, then next step.)
- Send request to server
- Server sends back entire copy
- Client opens file (normal Unix file descriptor)
- `read()`, `write()`, etc. all apply to copy
- When `close()`, if local cached copy changed, send back to server

## AFS Questions

- How does AFS gain control on `open()` or `close()`?
- What space is allocated for cached files on workstations?
- How does FS ensure cached copies are up-to-date since may be updated by several clients?



## Update Semantics

- No other access control mechanisms
- If several workstations `close()` file after writing, only last file will be written
  - Others silently lost
- Clients must implement concurrency control separately
- If two processes on same machine access file, local Unix semantics apply

## AFS Misc

- 1989: Benchmark with 18 clients, standard NFS load
  - Up to 120% improvement over NFS
- 1996: Transarc (acquired by IBM) Deployed on 1000 servers over 150 sites
  - 96-98% cache hit rate
- Today, some AFS cells up to 25,000 clients (Morgan Stanley)
- OpenAFS standard: <http://www.openafs.org/>

## Other Distributed File Systems

- *SMB*: Server Message Blocks, Microsoft (*Samba* is a free re-implementation of SMB). Favors locking and consistency over client caching.
- *CODA*: AFS spin-off at CMU. Disconnection and fault recovery.
- *Sprite*: research project in 1980's from UC Berkeley, introduced log file system.
- *Amoeba Bullet File Server*: Tanenbaum research project. Favors throughput with atomic file change.
- *xFS*: SGI serverless file system by distributing across multiple machines for Irix OS.

## Outline

- Overview (done)
- Basic principles (done)
- Network File System (NFS) (done)
- Andrew File System (AFS) (done)
- Dropbox (next)

## File Synchronization

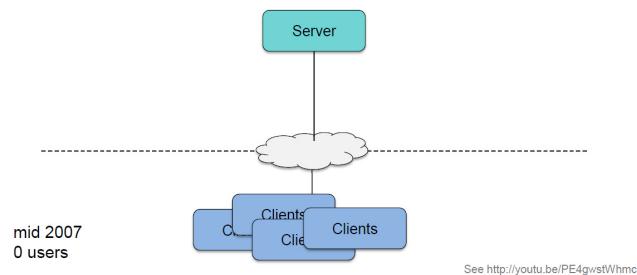
- Client runs on desktop
- Copies changes to local folder
  - Uploaded automatically
  - Downloads new versions automatically
- Huge scale – 100+ million users, 1 billion files/day
- Design
  - Small client, few resources
  - Possibility of low-capacity network to user
  - Scalable back-end
  - (99% of code in Python)

## Dropbox Differences

- Most Web apps high read/write
  - e.g., twitter, facebook, reddit 100:1, 1000:1, +
- Dropbox
  - Everyone's computer has complete copy of dropbox
  - Traffic only when changes occur
  - File upload : file download about 1:1
    - Huge number of uploads compared to traditional service
- Guided by ACID requirements (from DB)
  - Atomic – don't share partially modified files
  - Consistent – operations have to be in order, reliable; cannot delete file in shared folder and have others see
  - Durable – files cannot disappear

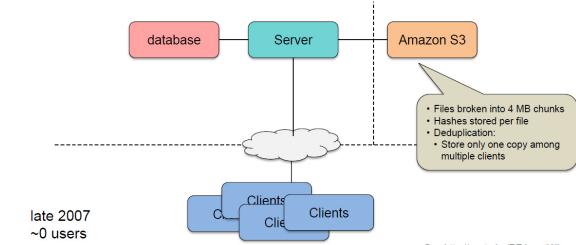
## Dropbox Architecture – v1

- One server: web server, app server, mySQL database, sync server



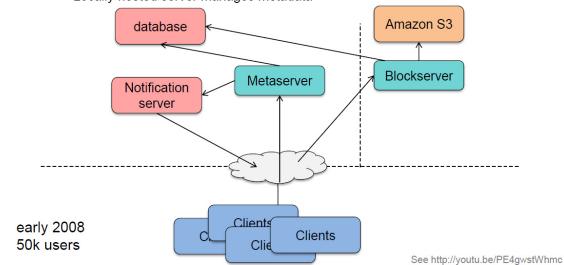
## Dropbox Architecture – v2

- Server ran out of disk space: moved data to Amazon S3 service (key-value store)
- Servers became overloaded: moved mySQL DB to another machine
- Clients polled server for changes periodically



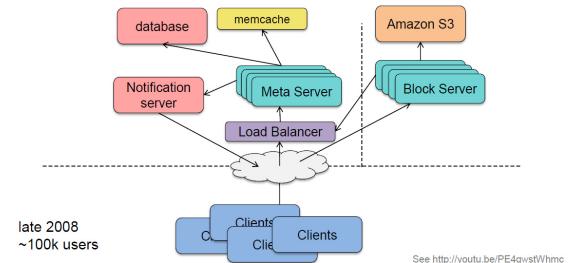
## Dropbox Architecture – v3

- Move from polling to notifications: add notification server
- Split web server into two:
  - Amazon-hosted server hosts file content and accepts uploads (stored as blocks)
  - Locally-hosted server manages metadata



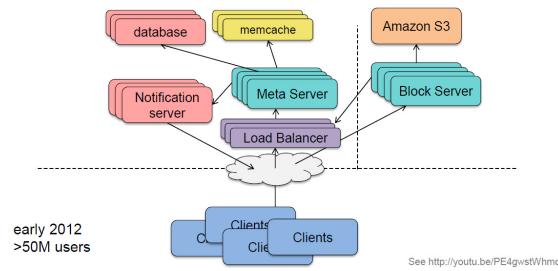
## Dropbox Architecture – v4

- Add more metaservers and blockservers
- Blockservers do not access DB directly; they send RPCs to metaservers
- Add a memory cache (memcache) in front of the database to avoid scaling



## Dropbox Architecture – v5

- 10s of millions of clients – Clients have connect before getting notifications
- Add 2-level hierarchy to notification servers: ~1 million connections/server



## Bit Bucket

## File System Functions

- Abstraction of file system functions that apply to distributed file systems

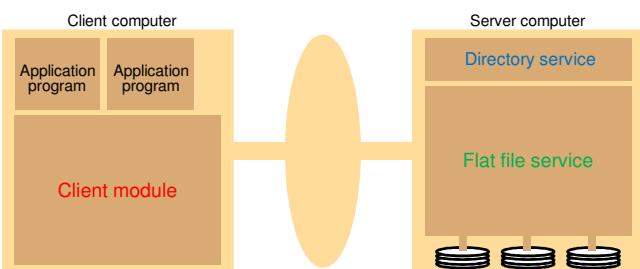
Directory module:	relates file names to file IDs
File module:	relates file IDs to particular files
Access control module:	checks permission for operation requested
File access module:	reads or writes file data or attributes
Block module:	accesses and allocates disk blocks
Device module:	disk I/O and buffering

- Most use set of functions derived from Unix (next slide)

## UNIX File System Operations

<code>filedes = open(name, mode)</code>	Opens an existing file with given <i>name</i> . Creates a new file with given <i>name</i> . Both operations deliver file descriptor referencing open file. The <i>mode</i> is <i>read</i> , <i>write</i> or both.
<code>status = close(filedes)</code>	Closes open file <i>filedes</i> .
<code>count = read(filedes, buffer, n)</code>	Transfers <i>n</i> bytes from file referenced by <i>filedes</i> to <i>buffer</i> . Transfers <i>n</i> bytes to file referenced by <i>filedes</i> from <i>buffer</i> . Both operations deliver number of bytes actually transferred and advance read-write pointer.
<code>count = write(filedes, buffer, n)</code>	Moves read-write pointer to offset (relative or absolute, depending on <i>whence</i> ).
<code>pos = lseek(filedes, offset, whence)</code>	Removes file <i>name</i> from directory structure. If file has no other names, it is deleted.
<code>status = unlink(name)</code>	Adds new name ( <i>name2</i> ) for file ( <i>name1</i> ). Gets file attributes for file <i>name</i> into <i>buffer</i> .
<code>status = link(name1, name2)</code>	
<code>status = stat(name, buffer)</code>	

## File Service Architecture



## File Service Architecture

- Flat file service**
  - Implement operations on the files
  - Manage unique file identifiers (UFIDs) – create, delete
- Directory service**
  - Mapping between text names and UFIDs
  - Create, delete new directories and entries
  - Ideally, hierarchies, as in Unix/Windows
- Client module**
  - Integrate flat file and directory services under single API
  - Make available to all computers