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

Transparency
Concurrent Updates
Replication
Fault Tolerance
Consistency
Platform Independence
Security
Efficiency

Illusion that local/remote 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

Transparency
Concurrent Updates
Replication
Fault Tolerance
Consistency
Platform Independence
Security
Efficiency

- 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

Transparency
Concurrent Updates
Replication
Fault Tolerance
Consistency
Platform Independence
Security
Efficiency

- 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 other copies
- More complicated solution
  - Multiple copies can be updated independently at same time needs finer grained refresh and/or merge

7

Fault Tolerance

Transparency
Concurrent Updates
Replication
Fault Tolerance
Consistency
Platform Independence
Security
Efficiency

- 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

Transparency
Concurrent Updates
Replication
Fault Tolerance
Consistency
Platform Independence
Security
Efficiency

- 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

Transparency
Concurrent Updates
Replication
Fault Tolerance
Consistency
Platform Independence
Security
Efficiency

- 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 (protocol)

10

Security

Transparency
Concurrent Updates
Replication
Fault Tolerance
Consistency
Platform Independence
Security
Efficiency

- 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

Transparency
Concurrent Updates
Replication
Fault Tolerance
Consistency
Platform Independence
Security
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

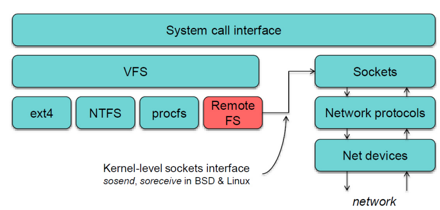
Read returns result of last write
- Easily achieved if
  - Only one server
  - Clients do not cache data
- But
  - Performance problems if no cache
  - Can instead write-through
    - Must notify clients holding copies
    - Requires extra state, generates extra traffic

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)

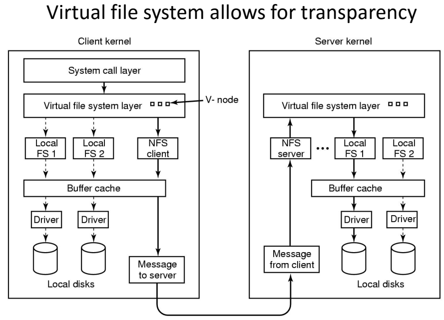
- For transparency, implement client as module under Virtual File System (VFS)



(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 what files each client has open and cached
- 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)
  - First was 1970's Data Access Protocol by DEC
  - But NFS first to be used as product
  - Developed in conjunction with Sun RPC
- Made interfaces in public domain
  - Request For Comment (RFC) by Internet Engineering Task Force (IETF) – technical development of Internet standards
  - 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 setups 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 NFS ran over UDP using Sun RPC
- Why UDP?
  - Slightly faster than TCP
  - No connection to maintain (or lose)
  - Reliable send not issue
    - NFS is designed for Ethernet LAN (relatively reliable)
  - UDP has error detection but no correction
    - NFS retries requests upon error/timeout

NFS Protocols

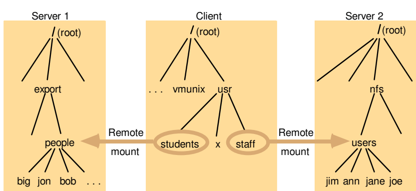
- Since clients and servers can be implemented for different platforms, need *well-defined* way to communicate → Protocol
  - *Protocol* – agreed upon set of requests and responses between client and servers
- Once agreed upon, Apple Mac NFS client can talk to a Sun Solaris NFS server
- NFS has two main 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 #, inode #, instance #
- Client
  - Create in-memory VFS inode at mount point
  - Internally point to r-node (for remote/RPC) 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

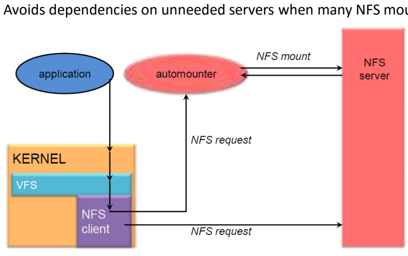
- File stored on server, typically `/etc/exports`

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

- Share folder `/public`
- Restrict to `192.168.1.0/24` Class C subnet
  - Use '*' for wildcard/any
- Give read/write access (`rw`)
- Allow root user to connect as root (`no_root_squash`)

NFS Automounter

- *Automounter* – only mount when access empty NFS-specified dir
  - Attempt unmount every 5 minutes
  - Conserve local resources if users don't need
  - Avoids dependencies on unneeded servers when many NFS mounts



NFS Access Protocol

- Most file operations supported from client to server (e.g., `read()`, `write()`, `getattr()`)
  - But doesn't support `open()` and `close()`
- First, client performs lookup RPC
  - Gets RPC handle for connection/return call
  - Successful call gets file handle (UFID) and attributes
  - Note, not like `open()` since no information stored on server
- On, e.g., `read()` client sends RPC handle, UFID and offset
- Allows server to be *stateless*, not remember connections
  - Better for scaling and robustness
- However, typical Unix file system can lock file on `open()`, unlock on `close()`
  - If doing with NFS must run separate lock daemon

NFS Access Operations

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

null lookup	link symlink readlink	getattr setattr
create remove rename	mkdir rmdir readdir	statfs
read write		

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
- Server supports *write-through* (data to disk immediately when client asks)
  - Performance can suffer, so another option only when file closed, called *commit*
- Delayed write – only put data on disk in batch when using memory cache
  - Typically every 30 seconds

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 (local `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 "typical" default (that used to be large)
  - Common Linux default 32K
- Read-ahead
  - Optimize for sequential file access
  - Send requests to read disk blocks before requested by process
- Generally → tune NFS performance
  - Many possibilities - server threads, network timeout, cache write, cache sizes, server disk layout …
  - "Best" depends upon system and workload

Problems with NFS

- File consistency (if 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 (errors) on WANs or wireless
  - Realized all traffic from one client to server can be multiplexed on one connection
    - Minimizes connection setup cost
- 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 in 1980's (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 for 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
  - Distributed file systems generally have 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" in that written to local disk, so still there if rebooted

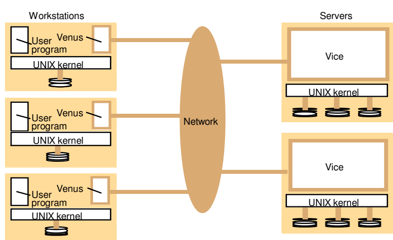
AFS Example

- Process at client issues `open()` system call
- Check if local cached copy
  - Yes? then use. Done.
  - No? then proceed to next step.
- Send request to server
- Server sends back entire copy
- Client opens file (normal Unix file descriptor, local access)
- `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 clients?
- How does AFS ensure cached copies are up-to-date since may be updated by several clients?

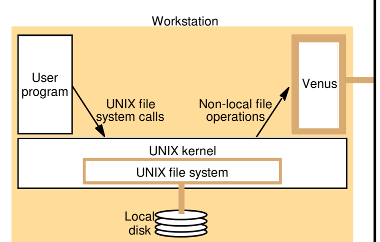
AFS Architecture



- *Vice* – implements flat file system on server
- *Venus* – intercepts remote requests, pass to vice
  - Vice provides for directory structure, relative location, working directory

System Call Interception in AFS

- Kernel mod to `open()` and `close()`
- If remote, pass to *Venus*



Cache Consistency

- Vice issues *callback promise* with file
- If server copy changes, it "calls back" to Venus processes, cancelling file
  - Note, change only happens on close of whole file
- If Venus process re-opens file, must fetch copy from server
  - Note, if client already had open, will still proceed
- If reboot, cannot be sure callbacks are all correct (may have missed some)
  - Checks with server for each open
- Note, versus traditional cache checking, AFS far less communication for non-shared, read-only files

(Flow diagram next slide)

Implementation of System Calls in AFS

User process	*UNIX kernel*	*Venus*	*Net*	*Vice*
open(FileName, mode)	If *FileName* refers to a file in shared file space, pass the request to Venus.	Check list of files in local cache. If not present or there is no valid *callback promise*, send a request for the file to the Vice server that is custodian of the volume containing the file.	→	Transfer a copy of the file and a *callback promise* to the workstation. Log the callback promise.
		Place the copy of the file in the local file system, enter its local name in the local cache list and return the local name to UNIX.	←	
	Open the local file and return the file descriptor to the application.			
read(FileDescriptor, Buffer, length)	Perform a normal UNIX read operation on the local copy.			
write(FileDescriptor, Buffer, length)	Perform a normal UNIX write operation on the local copy.			
close(FileDescriptor)	Close the local copy and notify Venus that the file has been closed.	If the local copy has been changed, send a copy to the Vice server that is the custodian of the file.	→	Replace the file contents and send a *callback* to all other clients holding *callback promises* on the file.

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 (i.e., generally none, unless processes explicitly lock)

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 first journaling 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)

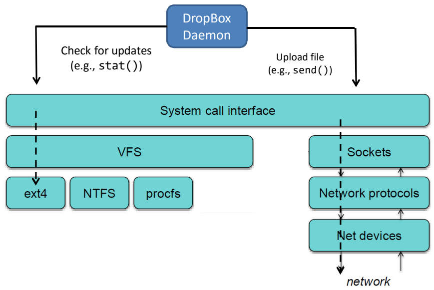
Dropbox Overview (1 of 3)

- 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 Overview (2 of 3)

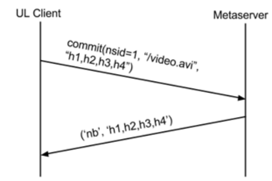
- Motivation most Web apps high read/write
  - e.g., Twitter, Facebook, reddit 100:1, 1000:1, +
- Everyone's computer has complete copy of Dropbox
  - Run daemon on computer to track "Sync" folder
- Traffic only when changes occur
  - Results in file upload : file download about 1:1
  - Huge number of uploads compared to traditional service
- Uses compression to reduce traffic

Dropbox Overview (3 of 3)



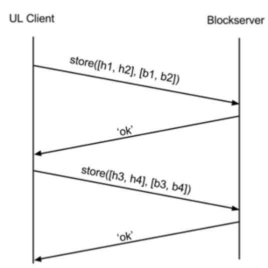
Dropbox Upload (1 of 3)

- Client attempts to "commit" new file
  - Breaks file into blocks, computes hashes
  - Contacts Metaserver
- Metaserver checks if hashes known
- If not, Metaserver returns that it "needs blocks" (nb)



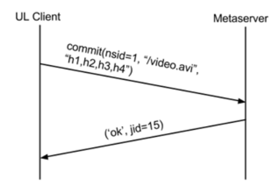
Dropbox Upload (2 of 3)

- Client talks to Blockserver to add needed blocks
- Limit bytes/request (typically 8 MB), so may be multiple requests



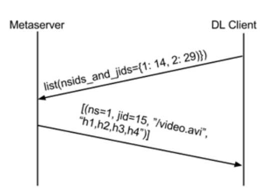
Dropbox Upload (3 of 3)

- Client commits again
  - Contacts Metaserver with same request
- This time, ok



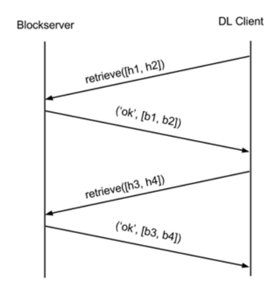
Dropbox Download (1 of 2)

- Client periodically polls Metaserver
  - Lists files it "knows about"
- Metaserver returns information on new files



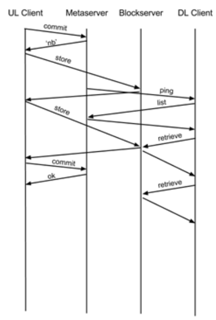
Dropbox Download (2 of 2)

- Client checks if blocks exist
  - For new file, this fails
- Retrieve blocks
- Limit bytes/request (typically 8 MB), so may be multiple requests
- When done, reconstruct and add to local file system
  - Using local filesystem system calls (e.g., `open()`, `write()`…)



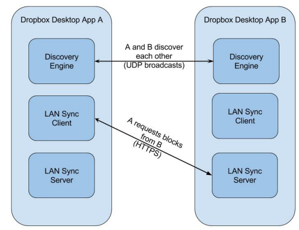
Dropbox Misc – Streaming Sync

- Normally, cannot download to another until upload complete
  - For large files, takes time "sync"
- Instead, enable client to start download when some blocks arrive, before commit
  - *Streaming Sync*



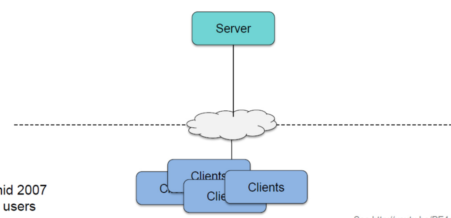
Dropbox Misc – LAN Sync

- *LAN Sync* – download from other clients
- Periodically broadcast on LAN (via UDP)
- Response to get TCP connection to other clients
- Pull blocks over HTTP



Dropbox Architecture – v1

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

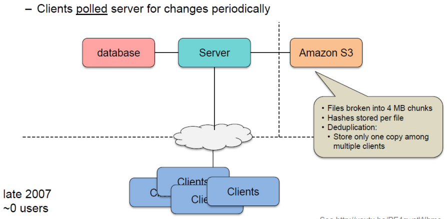


mid 2007
0 users

See http://youtu.be/PE4gwstWhmc

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

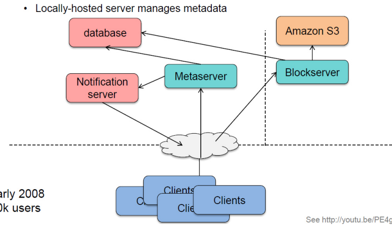


late 2007
~0 users

See http://youtu.be/PE4gwstWhmc

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

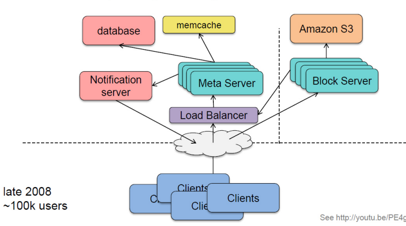


early 2008
50k users

See http://youtu.be/PE4gwstWhmc

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

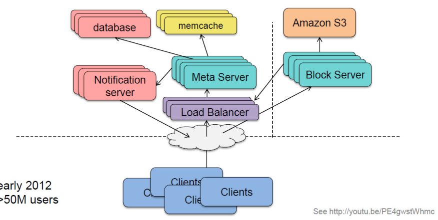


late 2008
~100k users

See http://youtu.be/PE4gwstWhmc

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



early 2012
>50M users

Bit Bucket

File System Functions

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

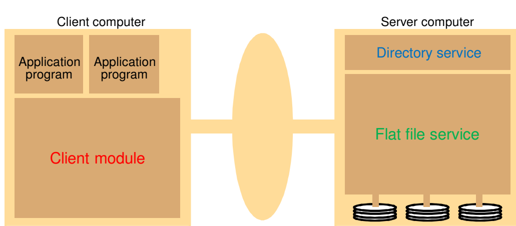
Directory module:	relates file names to file IDs
File module:	relates file IDs to particular files
Access control module:	checks permission for operation requested
File access module:	reads or writes file data or attributes
Block module:	accesses and allocates disk blocks
Device module:	disk I/O and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

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

UNIX File System Operations

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

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