Network File System: NFS and Coda

Presented by Hakim Weatherspoon

Evolution of networked file systems

- NFS: 1984, now in its 4th version. Developed by Sun.
- AFS: 1987, developed at CMU.
- Coda: post-AFS, also developed at CMU.

Design and Implementation of the Sun Network File System

Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon

Appears in USENIX Annual Technical Conference 1985

Goals

- Designed in early/mid 80s.
  - Before: each computer had its own private file disk + file system.
    - Fine for expensive central time-sharing.
    - Awkward for individual workstation.
- New model: One server, LAN full of client workstations. Not WAN.
  - Allow users to share files easily.
  - Allow a user to sit at any workstation.
  - Diskless workstations could save money

Implementation Goals

- Had to work with existing applications.
- Had to be easy to retro-fit into UNIX O/S.
- Had to implement same semantics as local UNIX FFS.
- Had to be not too UNIX specific
  - work w/ DOS, for example.
- Had to be fast enough to be tolerable
  - (but willing to sacrifice some).

Kernel FS Structure *before* NFS

- Specialized to local file system called Fast File System (FFS).
- Disk inodes.
- O/S keeps in-core copies of inodes that are in use.
  - File descriptors, current directories, executing programs.
- File system system calls used inodes directly.
  - To find e.g. disk addresses for read().
- Disk block cache.
  - Indexed by disk block #.

Why not a Network Disk (ND)?

- What was ND?
  - Server supplied a block store, with JUST read/write block RPCs.
  - This makes read/write sharing awkward.
  - Clients would have to carefully lock disk data structures.
- How is NFS different from ND?
  - Moves complex operations to server.
- Why might NFS be slower than ND?
  - Have to worry about consistency between multiple clients

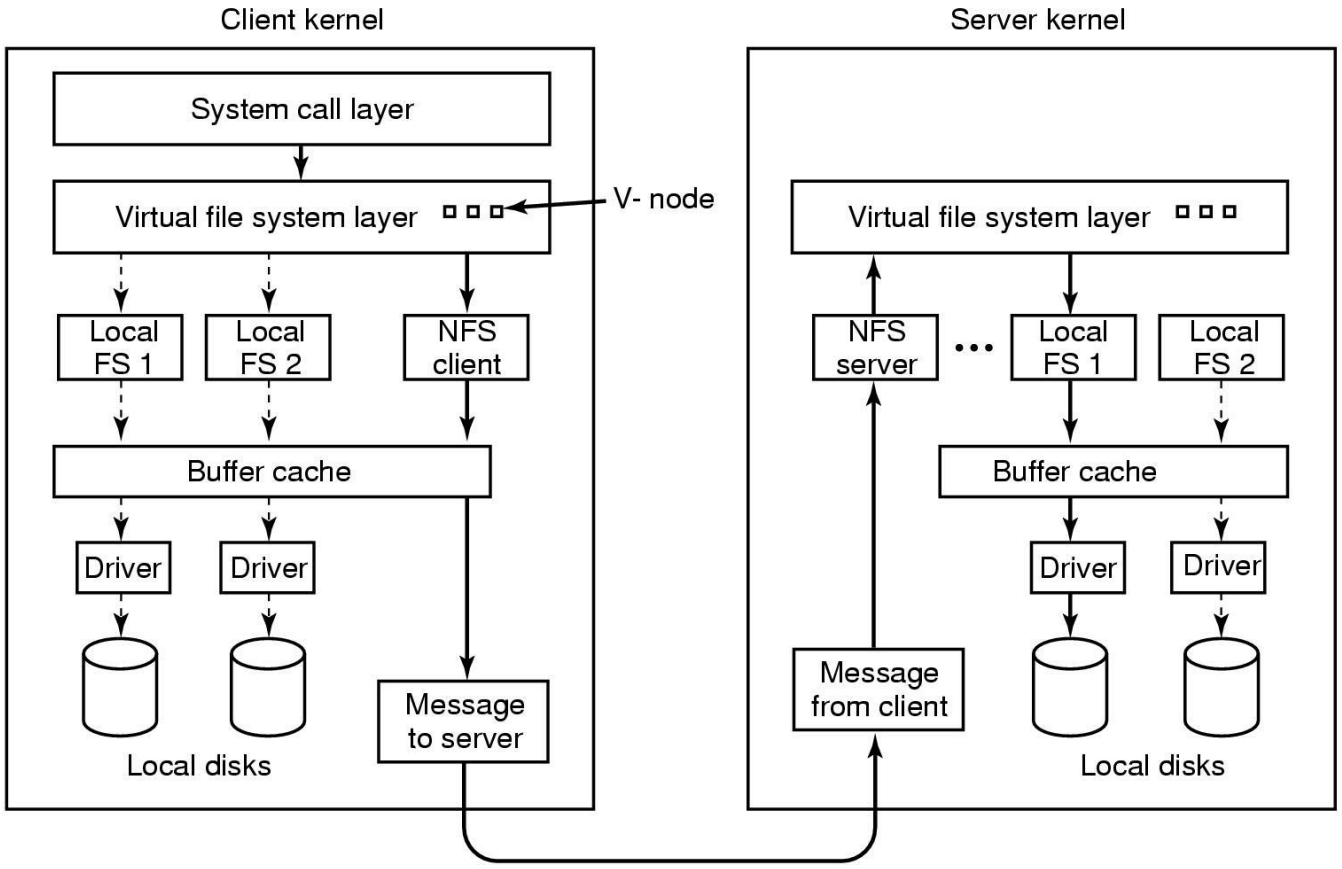
Virtual File System (VFS) Interface

- New "vnode" plan, invented to support NFS.
- Need a layer of indirection, to hide implementation.
  - A file might be FFS, NFS, or something else.
  - Replace inode with vnode object.
  - vnode has lots of methods:
    - open, close, read, remove.
  - Each file system type has its own implementation methods.
- What about disk cache?
  - Replaced with per-vnode list of cached blocks.

NFS Implementation

- Three main layers:
- System call layer:
  - Handles calls like open, read and close
- Virtual File System Layer:
  - Maintains table with one entry (v-node) for each open file
  - v-nodes indicate if file is local or remote
    - If remote it has enough info to access them
    - For local files, FS and i-node are recorded
- NFS Service Layer:
  - This lowest layer implements the NFS protocol

NFS Layer Structure



NFS Client/Server Structure

- Client programs have file descriptors, current directory, &c.
- Inside kernel, these refer to vnodes of type NFS.
- When client programs make system calls:
  - NFS vnode implementation sends RPC to server.
  - Kernel half of that program waits for reply.
  - So we can have one outstanding RPC per program.

NFS Client/Server Structure

- Server kernel has NFS threads, waiting for incoming RPCs.
  - NFS thread acts a lot like user program making system call.
  - Find *vnode* in server corresponding to client's vnode.
  - Call that vnode's relevant method.
  - Server vnodes are typically of type FFS.
  - This saves a lot of code in the server.
  - This means NFS will work with different local file systems.
  - This means files are available on server in the ordinary way.
  - NFS server thread blocks when needed.

How does NFS RPC designate file to read?

- E.g. a read RPC
- Could use file name.
  - Client NFS vnode would contain name, send in rpc.
  - Easy to implement in the server.
- Why doesn't this work?
  - Doesn't preserve UNIX file semantics.
    - Client 1: chdir("dir1");
      fd = open("file");
    - Client 2: rename("dir1", "dir2");
      rename("dir3", "dir1");
    - Client 1: read(fd, buf, n);
  - Does client read current dir1/file, or dir2/file?
  - UNIX says dir2/file.

The NFS File Handle

- What else goes in file handle, besides i-number?
  - File system ID - because server may server multiple file systems
  - Generation number - what is this?
    - Suppose an inode gets deleted & recycled while file/dir still open
    - Don't want old file descriptor referring to new file--very dangerous
    - Solution: Store generation number in inode on disk, change when recycled
    - What happens to read/write of old handle when generation number changes?
    - NFS stale file handle error
    - Sometimes: Inode of export point
  - So server can disallow lookup ("..") past export point
    - (not secure)

The NFS File Handle

- File systems already need a way to name files/inodes
  - E.g., How do directory entries refer to files?
    - Map name -> i-node number ("i-number")
- Don't want to expose i-node number details to client
  - I.e. client should never have to make up a file reference.
- So file handles are opaque.
  - Client sees them as 32-byte blob.
  - Client gets all file handles from the server.
  - Every client NFS vnode contains the file's handle.
  - Client sends back same handle to server.

NFS RPCs

- lookup
- read
- write
- getattr
- create
- remove, setattr, rename, readlink, link, symlink, mkdir, rmdir, readdir
- ***(no open, close, chdir)*** – why not?
  - Requires state to be maintained at server

Example

- `fd = open("./notes", O_RDONLY);`
- `read(fd, buf, n);`
- Client process has a reference to current directory's vnode.
- Sends `LOOKUP(dir-vnode, "notes")` to server.
- Server extracts i-number from file handle.
- Asks local file system to turn that into a local vnode.
- Every local file system must support file handles...
- Calls the local vnode's lookup method.
- dir->lookup("notes") returns "notes" vnode.
- NFS server code extracts i-number from vnode, creates new file handle.
- Server returns new file handle to client.
- Client creates new vnode, sets its file handle.
- Client creates new file descriptor pointing to new vnode.
- Client app issues read(fd, ...).
- Results in READ(file-handle, ...) being sent to server.

Where does first File Handle come from?

- Every NFS RPC has to be contain a file handle
  - A valid file handle
- Server's mount daemon maps file system name to root file handle.
- Client kernel marks mount point on local file system as special.
- Remembers vnode (and thus file handle) of remote file system.

Crash Recovery

- Suppose server crashes and reboots.
- Clients might not even know.
- File handles held by clients must still work!
- That's why file handle holds i-number, which is basically a disk address.
  - Rather than, say, server NFS code creating an arbitrary map.
  - That is, server is stateless!

What if open file gets deleted by different client?

- UNIX semantics
  - file still exists until I stop using it.
- Would require server to keep reference count per file.
  - Would require open() and close() RPCs to help mantain that count.
- Which would have to persist across server reboots.
- So NFS just does the wrong thing!
- RPCs will fail if some other client deletes a file I have open.
  - this is part of the reason why there is no open() or close() RPC.

What about performance?

- Does *every* program system call go over the wire to the server?
- No: client cache for better performance.
- Per-vnode block cache, name->file handle cache, attribute cache.
- Can satisfy read()s, for example, from block cache.

What about consistency?

- Is it enough to make the data cache write-through?
- No: I read a file, another client writes it, I read it again.
- How do I realize my cache is stale?

Consistency

- What are the semantics of read and write system calls?
- One possibility:
  - read() sees data from most recent write().
  - This is what local UNIX file systems implement.
- How to implement these strong semantics?
  - Turn off client caching altogether.
  - Or have clients check w/ server before every read.
  - Or have server notify clients when other clients write.

NFS chooses poor consistency

- In V2 implementation described, very poor consistency
- In newer implementations, close-to-open consistency
- If I write() and then close(), then you open() and read(), you see my data.
- Otherwise you may see stale data.
- How to implement these strong semantics?
  - Writing client must force dirty blocks during close().
  - Reading client must check w/ server during open().
  - Ask if file has been modified since data were cached.
  - This is much less expensive than strong consistency.
  - Though maybe not very scalable: every open() produces an

Optimizations

- NFS server and block I/O daemons
- Caching!
  - Client-side buffer cache
    - (write-behind w. flush-on-close)
  - Client-side attribute cache
  - Name cache
- XDR directly to/from mbufs
- Fill-on-demand clustering, swap in small programs

What about Security?

- Server has list of IP addresses.
- Fully trusts any client with that address.
- Client O/S expected to enforce user IDs, send to server.

Other issues

- soft vs hard mounts
- replay cache.
- I can execute files I can't read.
- what if I open(), then chmod u=?
  - owner always allowed to read/write…
- dump+restore may wreck client file handles.

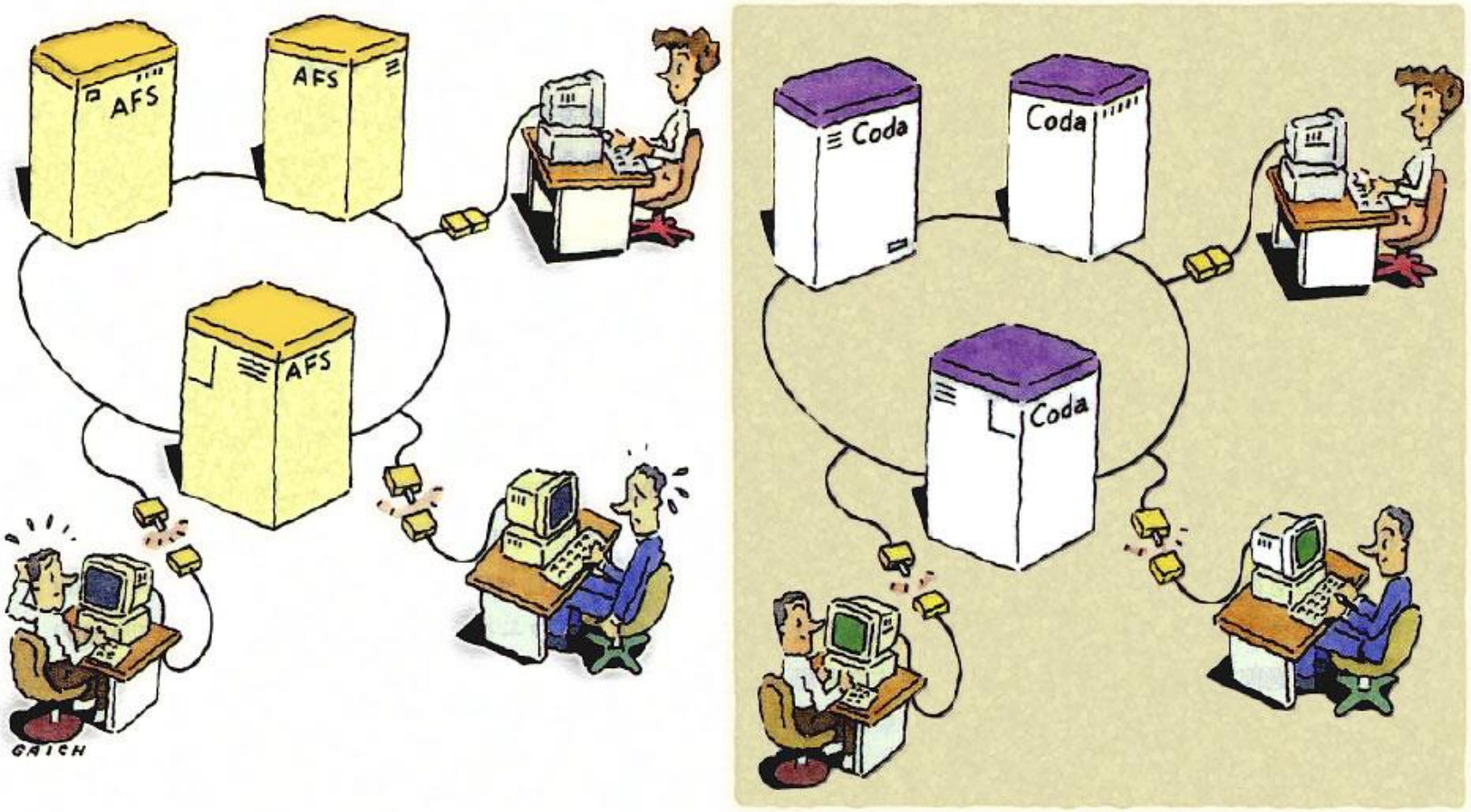
Coda: A Highly Available File System for a Distributed Workstation Environment

Mahadev Sayyanarayanan (“Satya”), James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere

Motivation for Coda

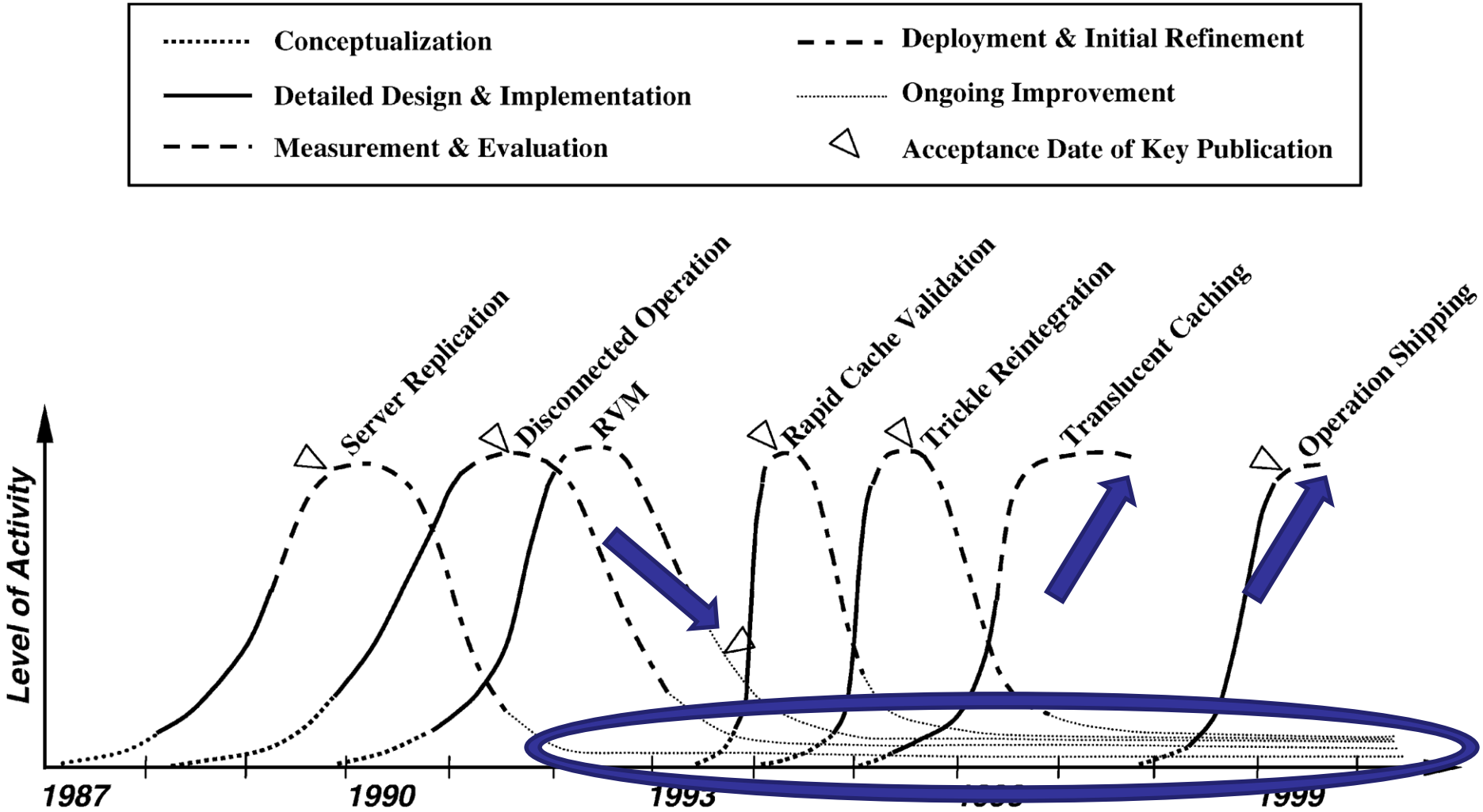
- Epilogue to the Andrew File System (AFS)
- AFS was found to be vulnerable to ***server and network failures***
  - Not that different from NFS
  - Limits scalability of AFS
- Coda provides high data availability
  - ***Server replication*** (optimistic replication)
  - ***Disconnected operation***

Coda Cartoon



http://www.cs.cmu.edu/~coda/images/nikkei-PAGE-213.JPG

Timeline


Conceptualization
Detailed Design & Implementation
Measurement & Evaluation
Deployment & Initial Refinement
Ongoing Improvement
Acceptance Date of Key Publication
Server Replication
Disconnected Operation
RVM
Rapid Cache Validation
Trickle Reintegration
Translucent Caching
Operation Shipping
Level of Activity
1987
1990
1993
1999

Lessons Learned from 20 years of Coda

- Optimistic replication can work
  - Must use for performance
- Real systems research needs
  - Real system artifacts
  - Real users
- Timing
  - Need to be lucky
- Research vs product
  - Long ‘product’ tail
- Moores law
  - Worked then. Does it work now?

Lessons Learned from 20 years of Coda

- Code reuse is a double edged sword
  - Good initially, but locks you into a particular regime
- Need system admins
  - Deeply held secret
- Small projects never die
  - Also small features hard to remove

Server Replication: 1987-1991

- ***Optimistic replication control protocols*** allow access in disconnected mode
  - Tolerate temporary inconsistencies
  - Promise to detect them later
  - Provide ***much higher data availability***
- Optimistic replication control requires a reliable tool for detecting inconsistencies among replicas
  - Better than LOCUS tool

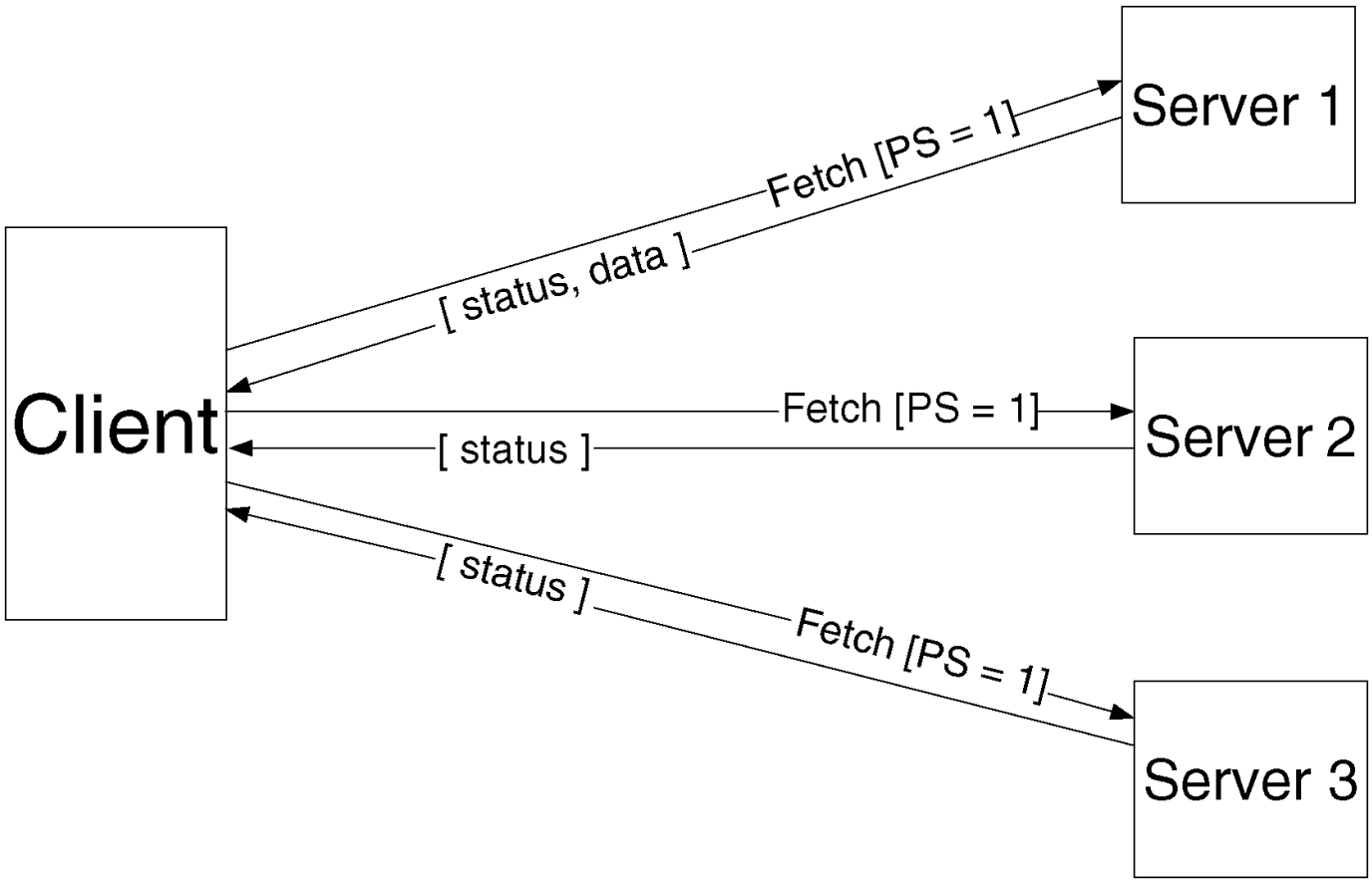
Server Replication

- Unit of replication is ***volume*** (subtree of files)
- Set of servers containing replicas of a volume is ***volume storage group*** (VSG)
- Currently accessible subset of VSG is ***accessible volume storage group*** (AVSG)
  - Tracked by cache manager of client (***Venus***):

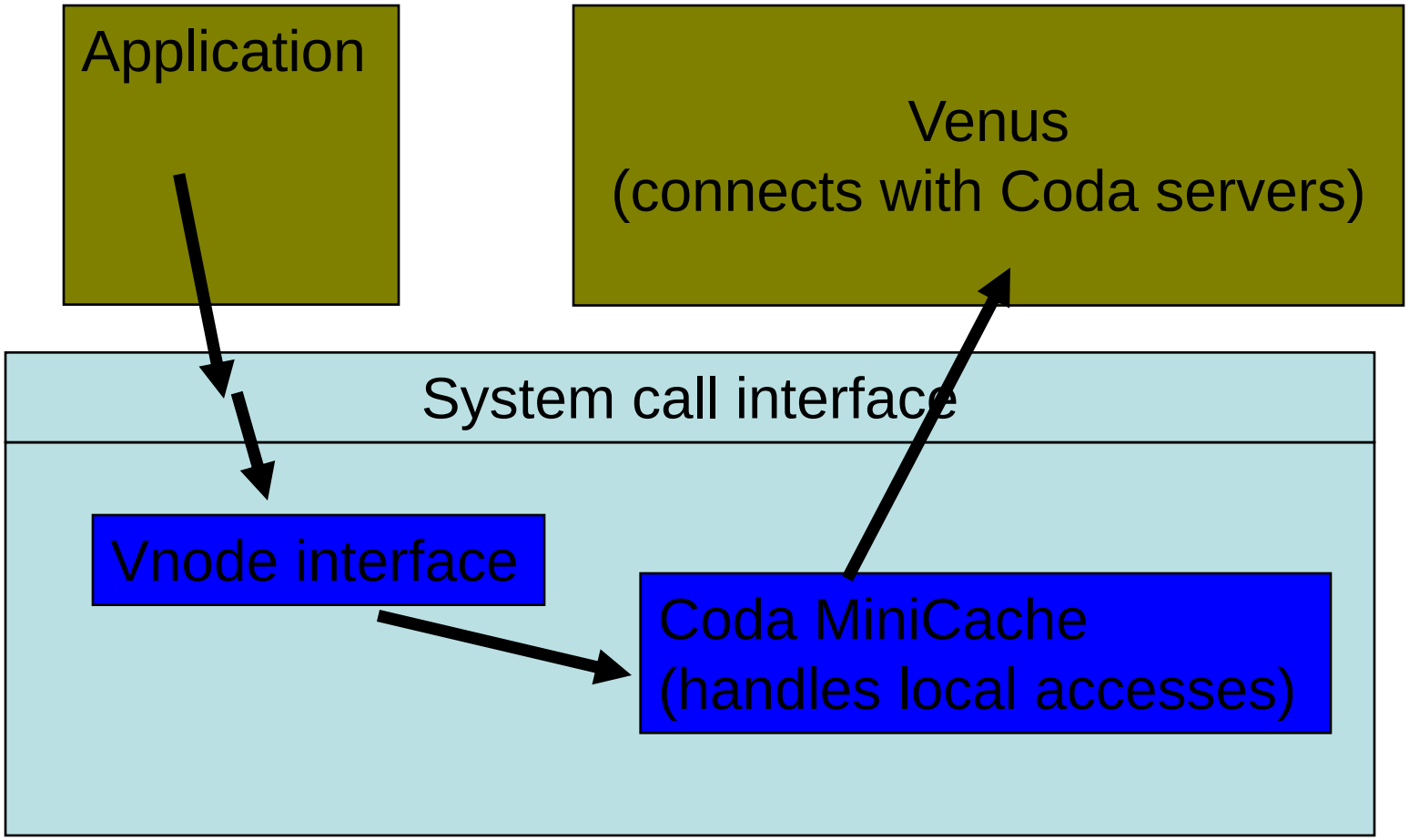
Read protocol

- ***Read-one-data, read-all-status, write-all***
- Each client
  - Has a ***preferred server (VS)***
  - Still checks with other servers to find which one has the latest version of a file
- Reads are aborted if a conflict is detected
- Otherwise a callback is established with all servers in AVSG

Read protocol



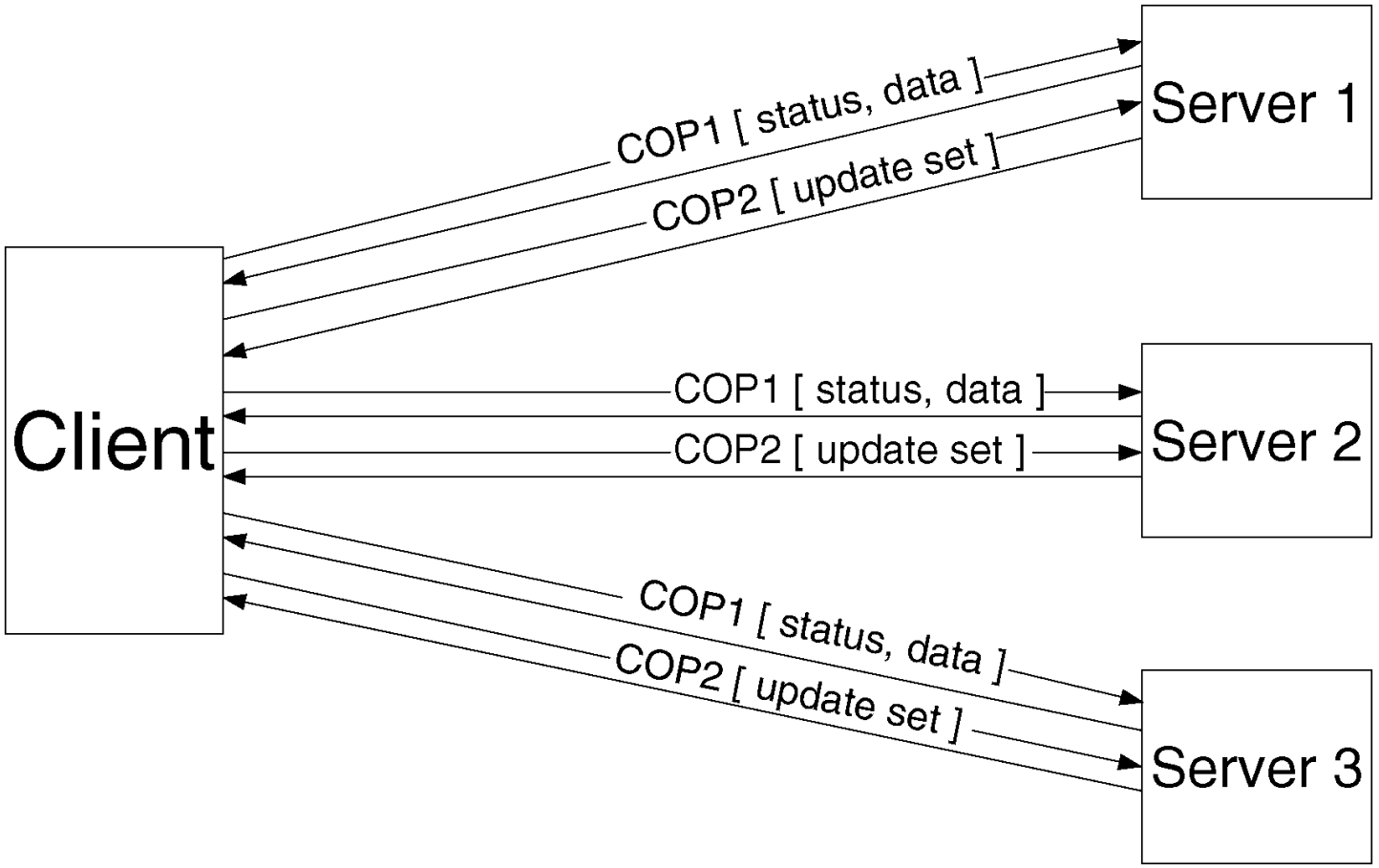
Client Structure



Update protocol

- When a file is closed after modification, updated file is transferred in parallel to all members of the AVSG
- Directory updates are also written through to all members of AVSG
- Coda checks for replica divergence before and after each update
- Update protocol is ***non-blocking***

Update protocol



Consistency model

- Client keeps track of subset ***s*** of servers it was able to connect the last time it tried
- Updates ***s*** at least every τ seconds
- At **open time**, client ***checks*** it has the ***most recent copy*** of file among all servers in ***s***
  - Guarantee weakened by use of ***callbacks***
  - Cached copy can be ***up to*** τ ***minutes behind*** the server copy

Fault-tolerance

- Correctness of update protocol requires atomicity and permanence of metadata updates
- Used first Camelot transaction management system:
  - Too slow and Mach-specific
- Coda uses instead its own ***recoverable virtual memory*** (RVM)
  - Implemented as a library

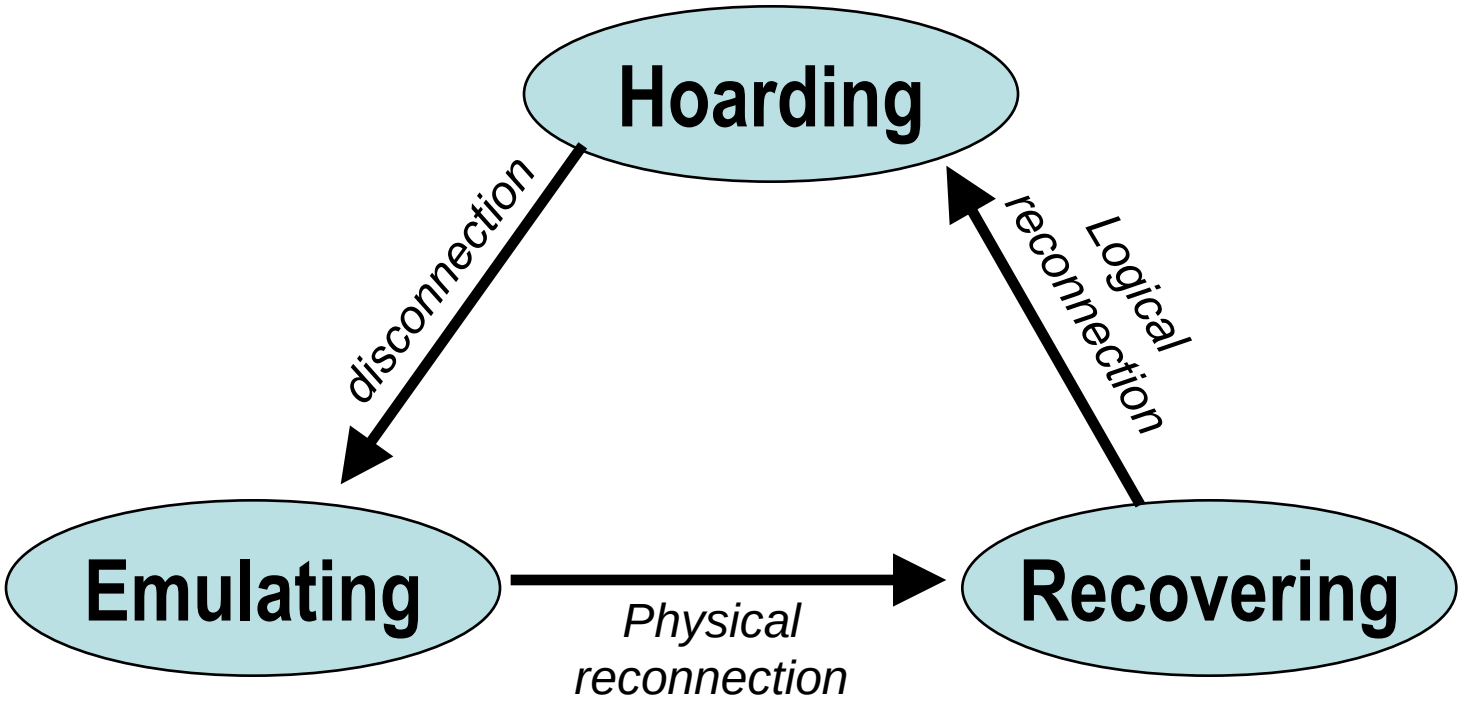
Disconnected Operation: 1988-1993

- Network isolation
  - Correlated server crashes
  - Overloaded or faulty routers
- Caching
  - Optimistic replication
  - Improve availability

Disconnected Operation

- Complimentary to server replication
  - Server replicas are *first-class replicas*
  - Cached replicas are *secondary replicas*

Implementation



Hoarding

- Implicit / Explicit sources of implementation
- Hoard database (HDB)
  - Specifies files to be cached
  - Users update directly or via command scripts
  - Venus periodically reevaluates every ten minutes

Emulation / Reintegration

- While disconnected, All changes are written to a log, Client modification log (CML)
- After reconnection, perform reintegration for each volume independently.
  - Venus sends CML to all volumes
  - Each volume performs a log replay algorithm

Conflict Resolution

- **Syntactic approach** to detect absence of conflicts.
  - Server replication
  - Version checks during the hoarding and reintegration
- **Semantic approach** to resolve conflicts
  - Different control for directory and file resolution

Conflict Resolution: Objectives

- *No updates should ever be lost without explicit user approval*
- *The common case of no conflict should be fast*
- *Conflicts are ultimately an application-specific concept*
- *The buck stops with the user*

Directory Resolution

- Automatic resolution by Coda
- After disconnected operation
  - Apply the CML
- During connected operation
  - Resolution log
  - Recovery protocol locks the replicas merges the logs and distributes the merged logs.

Application-Specific File resolution

- ***Application-specific resolvers*** (ASRs) are executed entirely on clients.

Conflict representation

```
$ ls -l
total 364
-rw-r--r--   1 satya    4976 Jun 28 08:42 cevol99.aux
lr--r--r--   1 root       27 Jun 29 11:08 cevol99.bib -> @7f0004e6.0000a52c.0000b7dc
-rw-r--r--   1 satya     528 Jun 28 08:42 cevol99.err
-rw-r--r--   1 satya   87070 Jun 28 08:41 cevol99.mss
-rw-r--r--   1 satya    6937 Jun 28 08:42 cevol99.otl
-rw-r--r--   1 satya  267914 Jun 28 08:42 cevol99.ps
```

(a) Before repair

```
$ls -lR cevol99.bib
total 75
-rw-r--r--   1 satya   26290 Jun 29 11:04 marais.coda.cs.cmu.edu
-rw-r--r--   1 satya   20286 Jun 29 11:03 mozart.coda.cs.cmu.edu
-rw-r--r--   1 satya   26290 Jun 29 11:04 verdi.coda.cs.cmu.edu
```

(b) During repair

```
$ ls -l
total 390
-rw-r--r--   1 satya    4976 Jun 28 08:42 cevol99.aux
-rw-r--r--   1 ras     26290 Jun 29 11:09 cevol99.bib
-rw-r--r--   1 satya     528 Jun 28 08:42 cevol99.err
-rw-r--r--   1 satya   87070 Jun 28 08:41 cevol99.mss
-rw-r--r--   1 satya    6937 Jun 28 08:42 cevol99.otl
-rw-r--r--   1 satya  267914 Jun 28 08:42 cevol99.ps
```

(c) After repair

Weakly Connected Operation: 1993-1996

- Broad principles
  - Do not punish strongly connected clients
  - Do not make life worse when disconnected
  - Do it in the background if you can
- Rapid validation of cache
- Trickle Reintegration

Lessons Learned from 20 years of Coda

- Optimistic replication can work
  - Must use for performance
- Real systems research needs
  - Real system artifacts
  - Real users
- Timing
  - Need to be lucky
- Research vs product
  - Long ‘product’ tail
- Moores law
  - Worked then. Does it work now?

Lessons Learned from 20 years of Coda

- Code reuse is a double edged sword
  - Good initially, but locks you into a particular regime
- Need system admins
  - Deeply held secret
- Small projects never die
  - Also small features hard to remove

Next Time

- Read and write review. *Turn into CMS before class*:
  - ***Mach: A new kernel foundation for UNIX development***, Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. *Proceedings of the USENIX Summer Conference*, Atlanta, GA, 1986, pages 93--112.
  - ***The Performance of µ-Kernel-based Systems***. Härtig et al. *16th SOSP*, Oct 1997.
- Check website for updated schedule
- Sign up to present