Disconnected Operation in the Coda File System

Robert Grimm
New York University

The Three Questions

- What is the problem?
- What is new or different?
- What are the contributions and limitations?

Remember AFS?

- Files organized into volumes (i.e., partial subtrees)
  - Identified by FIDs (instead of pathnames)
    - Name resolution performed on clients
  - Cached in entirety on client
    - Updates propagated on close
  - Kept up-to-date through callbacks from server(s)
- Volume cloning as central manageability mechanism
  - Provides consistent copy-on-write snapshot
  - Used for moving, read-only replication, backup

Coda Environment

- Almost the same target environment as AFS
  - Trusted Unix servers, untrusted Unix clients
  - Academic and research workloads
    - Very limited concurrent, fine-grained write sharing
  - But also *mobile computers*
    - Osborne 1 in '81
    - Compaq Portable in '83
    - Apple PowerBook in '91
      - Keyboard placement, room for palmrest, built-in pointing device

Coda Design Overview

- Two techniques for availability
    - Server replication across volume storage groups (VSGs)
        - Similar to AFS
            - Files cached on clients
            - Updates propagated in parallel to accessible VSG
            - Consistency ensured by callbacks
    - Client replication through local caches
        - When disconnected, FS continues to run from local cache
            - Cache misses result in failures, are reflected to application/users
- One replication strategy: optimistic replication
    - Allow copies to diverge, detect and resolve conflicts

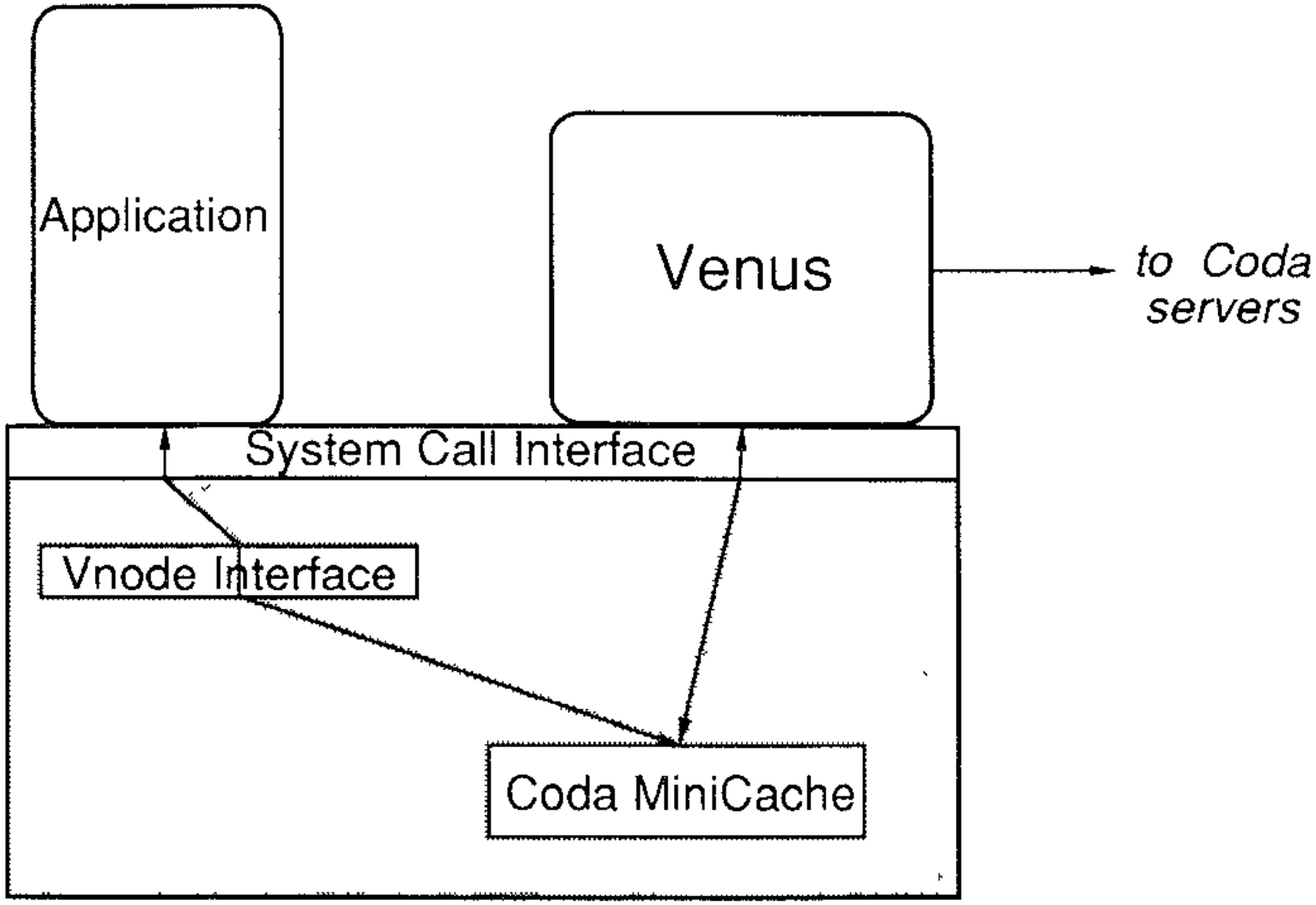
Design Rationale

- Place functionality on clients for *scalability*
  - Callbacks, whole-file caching, name resolution, ...
- Replicate across servers for *performance, scalability,* and *availability*
  - Servers have higher quality (space, security, maintenance)
- Replicate across clients for (more) *availability*
  - Continue to work across failures and intentional disconnection
    - But do not trust results of disconnected operation

Design Rationale (cont.)

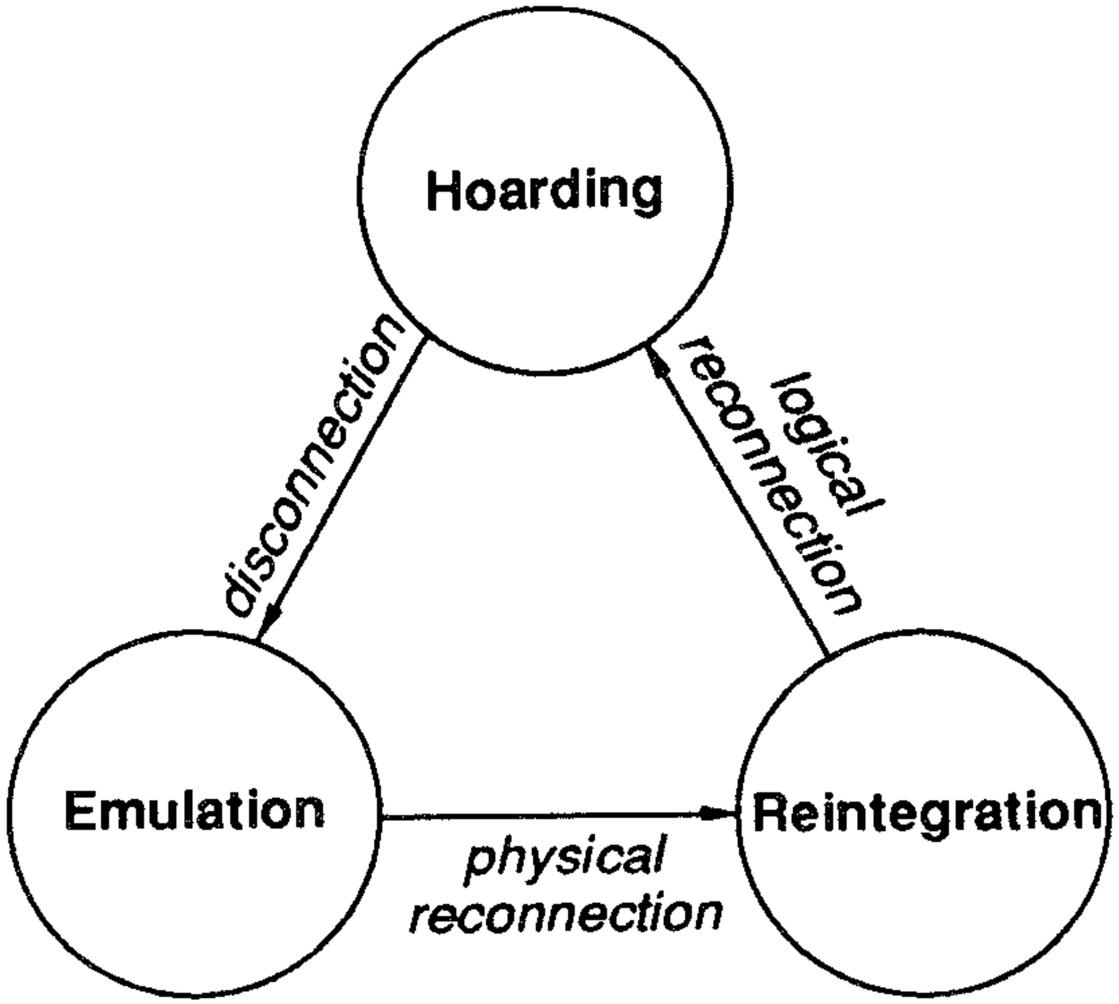
- Use optimistic replica control to make it all work
  - Locks and leases are too limiting
    - Especially when disconnection is "involuntary"
    - Locks may reserve resource for too long
    - Leases may reserve resource for not long enough
  - But optimistic control may result in (write/write) conflicts
    - Need to be automatically detected and resolved
    - But also are uncommon for typical Unix environments
  - Combination of approaches might be feasible
    - Pessimistic replication for servers, optimistic for clients
    - But may result in confusing behavior — how so?

Client Structure



- Most functionality located in user-level server
  - With small in-kernel component
- What changed re AFS? What are the trade-offs?

The Venus State Machine



- Hoarding prepares for possible disconnection
- Emulation services requests from local cache
- Reintegration propagates changes back to servers

Hoarding

- Balance current working set against future needs
  - Hoarding DB provides user-specified list of files
    - Prioritized, may include directories and their descendants
  - Cache organized hierarchically
    - After all, name resolution is performed on client (!)
  - Hoard walk maintains equilibrium
    - Goal: No uncached object has higher priority than cached ones
    - Operation
      - Re-evaluate name bindings to identify all children
      - Computer priorities for cache, HDB; evict and fetch as needed
    - What about broken callbacks?
      - Purge files, symlinks immediately; delay directory operations (?)

Emulation

- Local Venus pretends to be the server
  - Manages cache with same algorithm as during hoarding
    - Modified objects assume infinite priority — why?
  - Logs operations for future reintegration
    - Replay log (metadata, HDB) accessed through RVM — why?
      - Flushed infrequently when hoarding, frequently when emulating
    - Contains store operations, but not individual writes — why?
    - Removes previous store records on repeated close
    - Should remove stores after unlink or truncate
    - Should remove log records that are inverted
      - E.g., mkdir vs. rmdir on the same directory

Emulation (cont.)

- What to do when disk space runs out?
  - Currently: Manually delete files from cache
    - No more modifications if log is full
  - In the future
    - Compress file cache and RVM log
    - Allow user to selectively back out of updates
    - Back up cache, RVM log on removable media

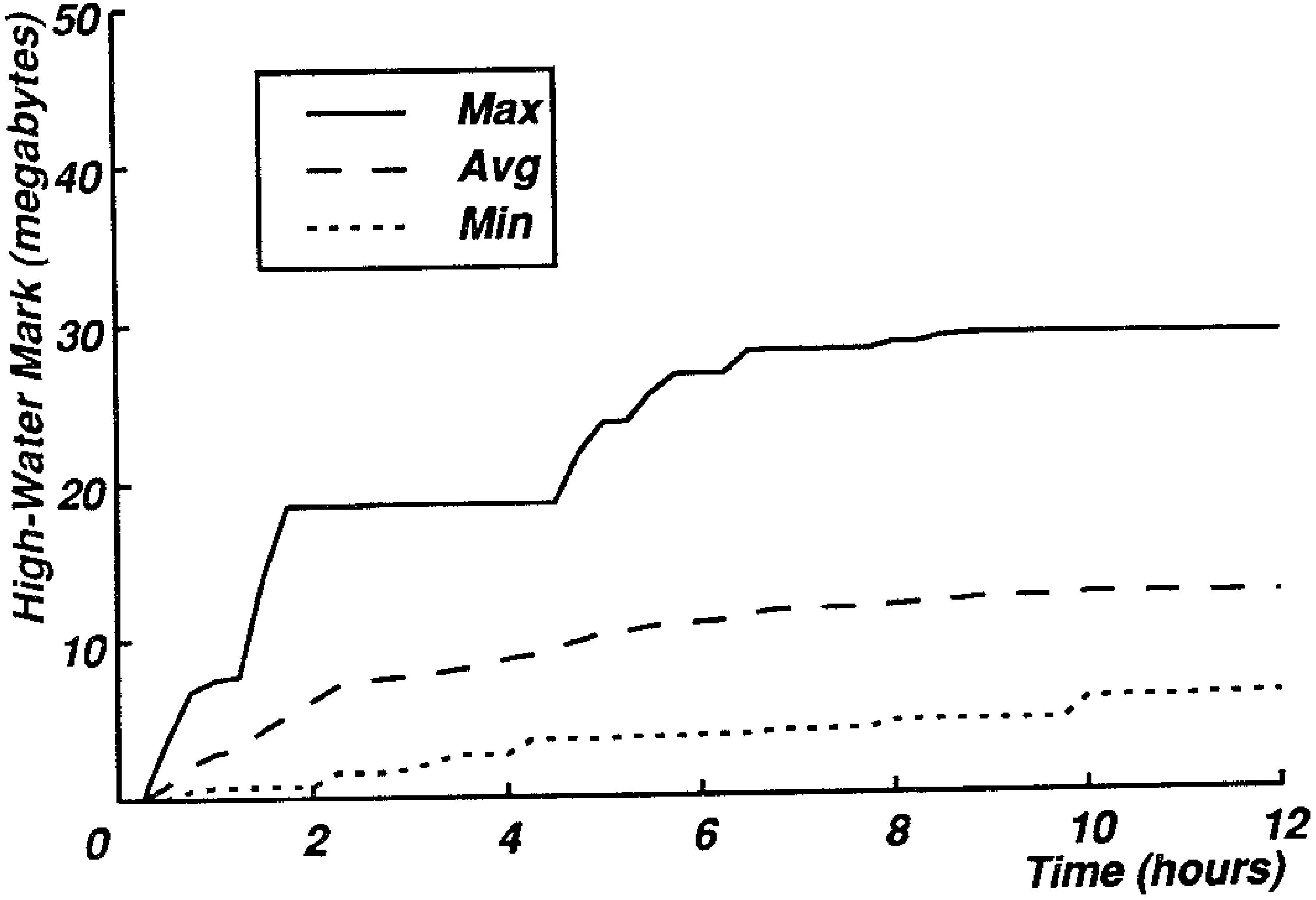
Reintegration

- Replay operations on (A)VSG
  - Obtain permanent FIDs for new objects
    - Block of preallocated FIDs is usually enough
  - Ship log to all servers in AVSG
    - Begin transaction, lock all referenced objects
    - Validate each operation and then execute it
      - Check for conflicts, file integrity, protection, disk space
      - Use monotonically increasing store IDs to identify conflicts
    - Perform data transfers (back-fetching)
    - Commit transaction, unlock all objects
    - On error, abort and create replay file (superset of tar file)

Some Experimental Results

- Reintegration timings
  - Andrew benchmark: 288 s execution, 43 s reintegration
    - Makes extensive use of file system (remember: *load unit*)
  - Venus make: 3,271 s execution, 52 s reintegration
- Likelihood of conflicts
  - Based on AFS traces — why?
  - 99% of all modifications by previous writer
    - Two users modifying same object <1 day apart: 0.75%
  - Without system files: 99.5% modification by previous writer
    - Two users modifying same object <1 day apart: 0.4%

How to Size the Cache?



- Based on trace data for Coda, AFS, local FS on 5 nodes
  - Simulated by Venus itself

What Do You Think?