Coda & disconnected operation

Note: this is one of 45 papers on the system

Motivation

- AFS caches whole files on the client
- Laptops allow people to take computers away
- Wouldn't it be nice to access files in that cache?
- Or: how to trade consistency for availability
  - key contribution: it is possible (and done windows XP)
- QUESTION: Is this applicable today

Challenges

1. Making sure the files you need are present
   - hoarding + access monitoring + configuration
2. Reintegrating changes when you connect
   - All at once or on demand
3. Dealing with conflicts with two off-line updates
   - Application-specific resolvers
4. Reducing bandwidth of resolution
   - Operation-based resolution (e.g., run *make* instead of copying object files)
5. Dealing with weak connectivity
   - New modes (synchronous download, asynchronous upload, bandwidth metering)
   - *Trickle integration* – upload portion of update log
6. Consistency of multiple updates
   - Isolation-only transactions – make all changes on client, then make all on server if possible

Question: consistency

- What is the right consistency model?
  - How do you trade off availability, consistency, partition tolerance?
  - Disconnected client = partition in this model
- What do clients of a file system expect?
  - What operations could cause problems?
    - Reading stale data
    - Simultaneous update of a file (maybe disconnected, maybe not)
  - Do clients know when they are disconnected and does this matter?

Coda Model

- Replicated file servers with volumes
  - Extends AFS with read/write copies
  - Volume Storage Group (VSG) = set of server containing a volume (group of files)
  - AVSG = set of running servers …
  - Goal: replication for reliability in the presence of **failures**
- First class replication: between servers
  - Reliable, accurate, trusted complete
  - Clients read one server (RO), check all to see if latest
    - If not latest, make the up-to-date one the "preferred"
    - Clients detect stale replicas, notify AVSG members
    - Client gets callback to preferred server (notified if file changes there)
  - Clients write all in AVSG (All Available)

Server Replication Consistency

- Clients establish "preferred server" for a volume, but different clients may have different preferred servers
  - Update on one server may not break callback on another with partitioning/failure
- Solution: volume replicas maintain CVV = coda version vector, is side effect of every modification operation
  - Mismatch in CVV between replicas indicates some are out of date
  - Standard version vector rules
  - Triggered by client inspecting vectors (like Dynamo)

Ensuring server consistency

- On read, clients obtain data from preferred server but versions from whole AVSG
  - If preferred out of date, makes most newest server preferred and retries
- On write, update file to AVSG
  - For reliable servers, ensures all servers have data
- Efficiency: use parallel *multiRPC*
- Servers do no crash recovery
  - Rely on clients to notify them of inconsistency on reads
  - When learn of failure, use *Force* operation to copy data to out-of-date site (one direction anti-entropy)

Comparison to Dynamo

- How different?
  - Client still checks versions, but uses most recent
  - Clients still fix conflicts
    - As part of a system call run handler to fix things up
  - Clients do the write to all replicas, rather than chaining from one
  - Clients do inconsistency detection, not servers
    - No background anti-entropy

Client replication

- Client caches are "second-class replicas"
  - They don't store persistent data,
  - Not involved in all synchronous operations
  - Store subset of all files
- Client replication used for availability
  - Tolerate network failures (disconnections)
- Three client states:
  - hoarding: preparing for disconnection by downloading and saving files
  - Emulation: pretending a server is available when it is not
  - Reintegration: propagating local changes back up to a server

Concurrency control

- What to do about update conflicts?
  - Pessimistic approach: lock all files while cached for modification (exlusive) or read (shared), prevent conflicting accesses
    - Problem: lack of availability with disconnected clients
    - Must know about disconnection in advance to acquire locks
  - Optimistic approach: allow conflicting updates, sort it out during reconciliation and roll things back
- QUESTION: When are these approaches good?
  - Pessimistic often better when there are many conflicts, machine generally available
  - Optimistic better when few conflicts

Hoarding problems

- Identifying files:
  - Human-specified files
    - binaries, configuration files
    - Priority indicates importance
  - LRU files
    - what currently working on
    - Priority based on recency
  - Namespace:
    - Need parent directories of all cached files
  - QUESTION: Will this work? Why not hoard your whole home directory, project directory
  - QUESTION: in the world of large disks, is this a problem
    - What if some files change frequently and require a lot of update traffic?

Updating cached files

- Space management:
  - Want to keep client cache full at all times with useful things
  - On eviction for delete or LRU, want to fetch other files to take up free space
- How do client's know a file has changed?
  - Callback on all cached files
- When do you update?
  - If update on every write, then bursts of writes are expensive
- CODA approach
  - Hoard walking – make sure have the **right** set of files (e.g. highest priority
    - Evict lower priority files to make space for higher priority files
    - (e.g. new file added to directory or old file has increased space)
  - Fetch changed files on hoard walks
  - Keep stale directory data for less-consistent offline use but purge stale files & symlinks
- QUESTION: How compare to anti-entropy & rumor mongering?

Emulation

- Coda emulates existence of a server while disconnected
  - Updates modify cached copy
    - Modified files have high priority so never evicted
  - Logs updates per-volume
    - One update per file (not need old overwrites)

Reintegration

- Goal: execute updates as a transaction
- Step 1: Allocate file IDs for all new objects (so can be agreed upon by server replicas
- Step 2: Coda ships replay log to AVSG
  1. Parse log, identify & lock all modified files
  2. Validate updates to detect conflicts, rule violations (e.g. protection), and execute all directory operations
     1. (create empty shadow files for new data)
  3. Upload data new data and write back
  4. Commit transaction, release locks

Conflict detection

- File/directory changes update an LSID
  - Clients store the LSID when caching a file
  - Clients included LSID when propagating a modification
    - If server LSID != client LSID, file changed, update not done
    - Like a compare-and-swap

Conflict Resolution

- Automatic resolution:
  - directory addition of different files
  - Deletion
- Manual reconciliation:
  - Adding two files with same name
  - updating/Renaming a file and deleting it
- Semi-automated
  - Applications with specific data patterns & conflict semantics
  - e.g. calendar: add non-conflicting events
  - Done with "application-specific resolvers" executed at client during reintegration to create new, unconflicted version (like Dynamo)

Automated reconciliation

- Client can write rules for automatic reconciliation per-directory tree
  - can be per-file type (e.g. delete temporaries, merge calendars)

Client state management

- Client maintains a log of file system operations (CML)
  - File name/id + operation (for directory entries) or contents (for file writes)

Replica management

- Each file update tagged with unique storeid (LSID) = client ID + logical time
  - maintaining history of a file in terms of LSID gives causal order of updates, like vector/lamport clocks
- CVV = update count for a file at all sites (number of changes made there)
  - missing update = entry in CVV is lower
  - Standard VV comparison rules used to detect out-of-date replicas and conflicts

Using CVV

- Client cache maintains LSID + CVV for each file at time downloaded
  - Not modified when changed locally at client
- Sent to server when file updated
  - If LSID new + cvv > server CVV, then no conflict and update is fine (server may be stale)
    - Increment CVV, use new LSID from client
    - Client takes new CVV from server, distribute to other servers (for common knowledge so all can increment CVV)
  - Otherwise, detects update conflict,

Issues

- Should clients control when reintegration happens?
  - Addressed with "trickle integration" – working disconnected but replaying log
- What about reverting updates?
  - Is this a possibility
- Requires file-granularity update
  - E.g. store calendar as a file per day, not a database of all events
- Compare to DropBox – how is it different?
  - Handles all files, not just user data