A. Arpaci-Dusseau **Department of Computer Science**
CS739: Distributed Systems **University of Wisconsin, Madison**

Disconnected Operation in the Coda File System (SOSP'91)

1 Motivation

- What were the goals of Coda?

  Goal: Enjoy the benefits of a shared data repository but be able to continue critical work when the repository is inaccessible. Be able to work when disconnected from servers, whether voluntary or involuntary.

  Voluntary vs. involuntary? *Involuntary – AFS servers (Vice) would go down all of the time. Rewrite motivation after the fact. Very important in research to get the correct motivation. Doesn't have to be your original motivation.*

  Other goals: Scalability: Experience with AFS convinced them this needed to be considered a priori, not as an afterthought. What is fundamental for scalability? Don't want server to be bottleneck. Whole-file caching. Place functionality on clients, not servers. Avoid system-wide rapid change.

  Transparency? *Transparency to only some extent. User's shouldn't be bothered by disconnected op too much, but might be aware; can't mask unavailability. Sometimes problems with reintegration.*

- What assumptions did Coda make?

  Assumptions: Large collection of untrusted clients with local disk and high bandwidth network to smaller number of trusted servers. Conventional, off-the-shelf hardware. Obviously, a local disk per machine is essential!

- How good of a job did the designers do of predicting technology trends?

  How do wireless networks change the picture?

- Coda developed from AFS. Briefly, how did AFS work with regard to caching files?? What type of data consistency does AFS provide?

  Whole file caching. Bring over copy of file on open and set up a callback on the server (keep state on server of which clients have file cached). When close, flush changes to server. When server sees changes to file, it breaks any outstanding callbacks so that client will re-fetch the file the next time it opens it. Otherwise, if client still has good callback, it doesn't have to re-fetch file on open.

  Close-to-open consistency: client when it calls open will get version of file that was last closed.

- Replication is often used to increase availability, but there are trade-offs that must be considered. Is it possible to simultaneously achieve perfect **consistency** and **availability** when suffering from network partitions? Why or why not? Which does Coda place more emphasis on?

  No.... If can't reach both sides, can either let two replicas diverge, or deny service to one of the groups. Coda chose availability. How? Caching.

- When a network is partitioned, replicas can be controlled with either pessimistic or optimistic replica control. What is *pessimistic* replica control? What are the pros and cons of it? Why don't leases solve the problem?

  Pessimistic: avoid conflict by disallowing operations when partitioned. Two possibilities. Can disallow any writes but let all partitions read (shared access) or can give ownership to a single partition (both reads and writes). Cons: Difficult to acquire exclusive access before partition occurs. Others unhappy that they can't access (entire community at mercy of one client). Pro: Data stays consistent.

  Leases don't work because one expires, disconnected client loses ability to access even if no one else interested. Have to throw away changes when expire. Leases work, as long as don't expire. Leases designed more to handle clients crashes than partitions.

- What is *optimistic* replica control? What are its pros and cons? Why was optimistic replica control chosen in Coda? Can you think of an environment where pessimistic replica control would be more appropriate?

  Optimistic: Permit reads and writes when partition occurs, detect and fix up problems after the fact. Cons: Updates might conflict and system state may no longer be correct. Doesn't work well if much write-sharing in workload.

  Availability as a goal goes well with optimistic; consistency goes with pessimistic.

  Not appropriate for circumstances, like finances where correctness is needed; can't allow ATM withdrawals if don't know how much is really in the bank account.

- Coda performs replication on both the servers (VSG, volume storage group) and clients. What are the differences between these two types of replicas? What does Coda do if some, but not all, servers are available? With a different view of servers, how might you design a file system for disconnected operation?

  Clients can be turned off at any time and for any period, limited disk capacity, may be tampered with; servers are more likely to be always on and have greater disk capacity. The replicas on servers are viewed as first class replicas; they are more persistent, widely known, secure, available, complete and accurate. Clients are second class; inferior along all axes except availability.

  AVSG: Accessible VSG. Writes propagate in parallel to all in AVSG; can read from any.

2 Detailed Design and Implementation

- Clients are managed by a software layer called Venus. How does the state and behavior of Venus change as the client becomes disconnected or connected?

  Venus, running on the Coda client, operates in one of three states: Venus is normally in hoarding*; upon disconnection, it enters* emulation*; upon reconnection, Venus enters* reintegration *and then returns to hoarding.*

- Consider the hoarding state first, in which Venus attempts to hoard useful data in anticipation of disconnection. The challenge for hoarding is that the amount of cache space on the clients is, of course, limited. During hoarding, what tensions must Venus balance in how it manages the client cache? How does Venus decide what is cached? (What information is given infinite priority in the local cache? Why?)

  Tension: For performance (while connected), should cache what is being used now; for availability, should cache what will be absolutely needed when disconnected.

  Decisions based on explicit statements and implicit usage. Explicit made easier with command to record all file usage. Can specify all files in a directory or all descendants (current or forever). Combine, so each file has a dynamic priority; cache the highest priority files.

  Infinite: Directories with cached children, so that can find children; also modified data (but this is in emulation phase).

- Is Venus during the hoarding stage identical to AFS? Why might the performance of Coda Hoarding be worse than AFS?

  Petty similar, but certainly cache slightly different things. Why must Venus sometimes (re-)fetch data from the servers (when AFS would not need to)? When does it do this? (Why doesn't it re-fetch data immediately?)

  Re-fetch if cache not in equilibrium; B is in HDB but not accessed for awhile; A not, but accessed, so replaces B. After some time for decay, PriB ¿ PriA; need to refetch A. Check for equilibrium periodically in "hoard walk".

  Also needed for callback breaks (don't have most up-to-date copy of file). Refetch on hoard walk (not immediately because likely to be changed again by other client).

- Imagine that Venus includes a command so a user can specify that disconnection is about to take place. How should Venus respond?

  If disconnection is looming, perform hoard walk. (Do you think you should you give complete priority to the files explicitly specified in the hoard database at this point???)

  No, not complete priority; the user is accessing some files which are clearly now important to them.

- During emulation, Venus on the client performs many of the actions normally handled by the servers. What types of tasks does this include? How does Venus record enough information to update the servers during reintegration? How does Venus save space? What happens when all space is consumed?

To emulate the behavior of server: create new file ids (preallocate some during hoard phases); manage the cache (give modified objects infinite priority, remove objects from cache when deleted, and return error on cache miss).

Mutating actions are placed in log; certainly does not put the data itself in there; save space by only recording final store (not all intermediate writes) of file; replace previous store records. If consume all log space, can't do anymore mutating ops.

- During reintegration, Venus propagates changes made during emulation to the servers and updates its cache to reflect current server state. What are the steps of reintegration? Under what circumstances will the replay fail? How is failure detected? What happens when the replay fails? Do you think Coda chose the right level of granularity for conflict resolution?

  Obtain permanent FIDs. Ship log to AVSG; each server replays independently and aborts on errors. To replay log: 1) Parse log, lock objects 2) Validate ops and execute; look for conflicts; stores cause an empty shadow file to be setup. 3) Perform back-fetching from client of all stored data. 4) Commit and release all locks.

  (Why is "back fetching" placed in its own step? Because expensive to bring all data over if not necessary.)

  Fail: Write-write conflict. Detected by seeing that the storeid of the object is different. Store-id updated on every successful store.

  On store failure, no operations are performed at all; the replay is atomic at the level of a volume; other systems we will look at later will have finer granularity.

  Directory operations are more flexible (can create/delete different files in same dir).

3 Status and Evaluation

- In their Evaluation, they look at 3 questions. How long does reintegration take? How large a local disk does one need? And, how likely are conflicts? Which question do you think has the most impact on whether or not Coda could be successful?

- Question 1: About how long is reintegration expected to take? Why is the time for this step crucial? How are technology trends likely to impact this time? Is a design change needed?

  About one minute for hours off-line. All update activity suspended. Don't see opportunities for big improvement from technology (but some from faster disks and networks) balanced with longer off-line times. Would like to change design so updates can be concurrent (at least to non-involved files.

- How did they determine the size of a needed local disk? How are technology trends likely to impact this? Is a design change needed?

  How much data was accessed by a Coda workstation over 10 hours? Why does one need a larger local disk than this for successful operation? About 30 MB in worst case. Need more space since don't do perfect caching. Larger local disks in future make this not a huge problem. Don't need to fix anything here.

- How likely is a conflict during reintegration? Will technology trends impact this? Is a design change needed?

  Unlikely. Over 99% of mods are by same user. More likely same user uses multiple machines, which could cause conflicts more readily. Need finer granularity conflict resolution before resorting to manual intervention.

4 Conclusions

- Coda handles both voluntary and involuntary disconnection of a client from the network. Where could Coda have made different (or simpler) decisions if they had handled only voluntary disconnection?

  If only voluntary, then wouldn't need to be performing hoarding in normal case; just do hoard walk right before known disconnection.

  Could change to do pessimistic replica control, but probably not the right solution.

- *Conclusion: Coda is able to improve availability during disconnected operation by performing caching on clients with optimistic replica control. Due to small amount of write-sharing, the number of conflicts that must be resolved when reconnected is relatively small.*

- If you were designing a file system for disconnected operation today, what would you do differently?