

Outline for Today's Lecture

Administrative:

- Problems posted
- Due date Sunday night. Demos Mon-Wed.

Objective:

- Prediction for prefetching
- Atomic transactions in a distributed system

Prefetching

- To avoid the access **latency** of moving the data in for that first cache miss.
- **Prediction!** "Guessing" what data will be needed in the future. How?
 - It's not for free:
 - Consequences of guessing wrong
 - Overhead – removal of useful stuff, disk bandwidth consumed

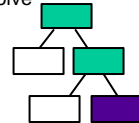
Intrafile prediction

- Sequential access suggests prefetching block $n+1$ when n is requested
- Upon seek (sequentiality is broken)
 - Stop prefetching
 - Detect a "stride" or pattern automatically
 - Depend on hints from program
 - Compiler generated "prefetch" statements
 - User supplied
 - How often is this issue relevant? Big files, nonsequential files, predictable accesses.

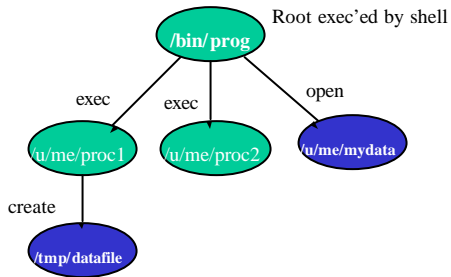
Interfile prediction

Related files – what does that mean?

- Directory nodes that are ancestors of a cached object must be cached in order to resolve pathname.
- Detection of "file working sets"
 - Trees representing program executions are constructed
- Capture semantic relationships among files in "semantic distance" measure – SEER system



Program Execution Tree



Generalize at Hoard Time

- Continuous monitoring and logging file access patterns
- To determine all of a program's dependencies
- Essentially union all saved trees (a limited number based on LRU) from previous executions of this program
- Per-application not per-user basis (global)
- Set of heuristics to fix over-generalization (if requested)
 - differentiate between data (private?) and application files (e.g. extensions, directory, time)

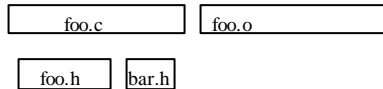
SEER's Hoarding Scheme: Semantic Distance

- **Observer** monitors user access patterns, classifying each access by type.
- **Correlator** calculates semantic distance among files
- **Clustering algorithm** assign each file to one or more **projects**
- Only entire projects are hoarded.

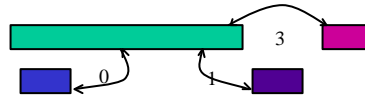
Defining Semantic Distance

- **Temporal semantic distance** - elapsed time between two file references
Time scale effects :-()
- **Sequence-based semantic distance** - number of intervening file references between 2, of interest. At what point? Open? Close?
- **Lifetime semantic distance** - accounts for concurrently open files - overlapping lifetimes

Example of Lifetime Distance



Distance is 0 if A not closed before B opened (Overlap)
intervening opens including itself otherwise
foo.c -> foo.h 0
foo.c -> bar.h 0
foo.c -> foo.o 3



- How to turn semantic distance between two references into semantic distance between files? Summarize - geometric mean.
- Using months of data. Only store n nearest neighbors for each file and files within distance M
- *External investigators* can incorporate some extra info (e.g. heuristics used by Tait, makefile)

Real World Complications

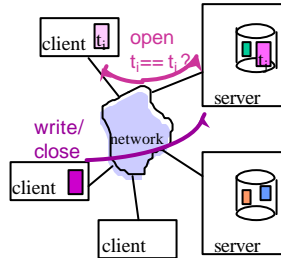
- Meaningless clutter in the reference stream (e.g. find command)
- Shared libraries - an apparent link between unrelated files - want to hoard but not use in distance calculations and clustering
- Rare but critical files, temp files, directories
- Multi-tasking clutter
- Delete and recreate by same filename.

Reliability Issues

- Server crashes
 - State (if any kept) lost, reconstruct upon recovery (dialog with clients?)
 - Stateless server - all requests from clients are self-contained
- Network partitions
 - Client response - optimistic (continue to use what's in cache) or pessimistic (conservative)

Sun NFS Cache Consistency

- Server is **stateless**
- Requests are self-contained.
- **Blocks** are transferred and cached in memory.
- Timestamp of last known mod kept with cached file, compared with "true" timestamp at server on Open. (Good for an interval)
- Updates delayed but flushed before Close ends.



30

NFS as a "Stateless" Service

The NFS server maintains no transient information about its clients; there is no state other than the file data on disk.

Makes failure recovery simple and efficient.

- **no record of open files**
- **no server-maintained file offsets:** read and write requests must explicitly transmit the byte offset for the operation.
- **no record of recently processed requests:** retransmitted requests may be executed more than once.
 - Requests are designed to be **idempotent** (repeatable) whenever possible (e.g., no append mode for writes, no exclusive create)

Drawbacks of a Stateless File Service

The stateless nature of NFS has compelling design advantages (simplicity), but also some key drawbacks:

- Update operations are disklimited because they *must be committed synchronously* at the server. Otherwise, live clients can "see" data loss if server crashes.
- NFS cannot (quite) preserve local *single-copy semantics*.
 - Files may be removed while they are open on the client.
 - Idempotent operations cannot capture full semantics of Unix FS.
- Retransmissions can lead to correctness problems and can quickly saturate an overloaded server.
- Server keeps no record of blocks held by clients, so cache consistency is problematic.

The Synchronous Write Problem

Stateless NFS servers must commit each operation to stable storage before responding to the client.

- Interferes with FS optimizations, e.g., clustering, LFS, and disk write ordering (seek scheduling).
 - Damages bandwidth and scalability.
- Imposes disk access latency for each request.
 - Not so bad for a logged write; much worse for a complex operation like an FFS file write.

The synchronous update problem occurs for any storage service with reliable update (*commit*).

"Committing" a Transaction

Begin Transaction

lots of reads and writes

Commit or Abort Transaction

"Committing" a Transaction

Begin Transaction

Withdraw \$1000 from savings account

Deposit \$1000 to checking account

Commit or Abort Transaction

Atomic Transactions

ACID property - data is recoverable.

- Atomicity - a transaction must be *all-or-nothing*.
- Consistency - a transaction takes system from one consistent state to another
- Isolation - No intermediate effects are *visible* to others - *serializability*
- Durability - the effects of a committed transaction are *permanent*

Implementation Mechanisms

- Stable storage
- Shadow blocks
- Logging

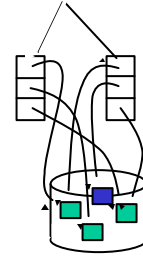
Stable Storage

We need to be able to trust something not to be corrupted or destroyed

- Mirrored disks. Always write disk 1, verify, then write disk 2.
 - If crash, compare disks, disk 1 “wins”
 - If bad checksum, use other disk block.
- Battery backed up RAM

Private Workspace

- Create a shadow data structure
- On commit, make the shadow the real one.
 - One pointer change to exchange indices allows this to be atomic.

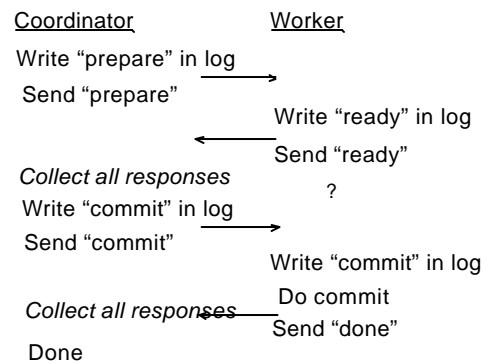


Logging

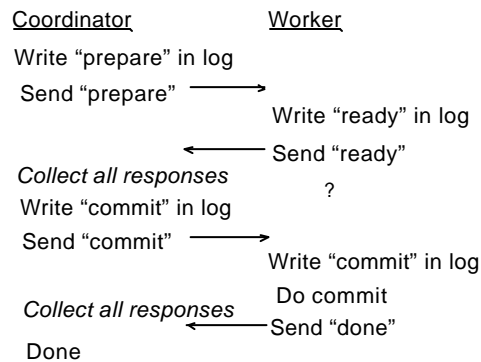
- Intentions list
- Do/undo log
- Log is written to stable storage. Rollback, if abort. Completion, if commit.

Savings	\$5K/\$4K
Checking	\$100/\$1100
Commit	

2 Phase Commit



2 Phase Commit



Explicit First-class Replication

- File name maps to set of replicas, one of which will be used to satisfy request
 - Goal: availability
- Update strategy
 - Atomic updates - all or none
 - Primary copy approach
 - Voting schemes
 - Optimistic, then detection of conflicts

Multiple Copy Schemes

- Primary Copy
 - Of all copies, there is one which is "primary" to which updates are sent. Secondary copies *eventually* get updates. Reads can go to any copy. How to take over when primary gone?
- Voting
 - An operation must acquire locks on some subset of copies (overlapping read or write quorums)
- Optimistic
 - Act as though no conflicts, when possible, compare replicas for conflicts (version vector) and resolve.