Recovery Management in QuickSilver.

Haskin88: Roger Haskin, Yoni Malachi, Wayne Sawdon, Gregory Chan, ACM Trans. On Computer Systems, vol 6, no 1, Feb 1988.

Microkernels & Database OSs

- Stonebraker81
  - OS/FS in the way of smarter DBMS, & besides, DBMS is an OS anyway
- Accetta86: Mach
  - Rapid development and customization of OS hampered by bundling of most services in OS core
  - Core should be messaging (IPC = LPC + RPC), threads, HW mgmt only
  - e.g.: Mac OS X is Mach 2.5 + FreeBSD 4
  - Virtual Machine Hypervisor is current vision

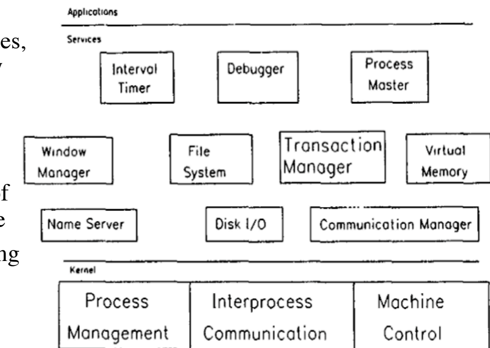

Fig. 1. QuickSilver system structure.

Very different philosophies

- Database folks:
  - OS, get out of the way!
  - Want raw access to everything, "we can do it better"
- Exokernel folks:
  - Get the OS out of the way!
- Microkernel folks:
  - Get the OS functions into user-space out of kernel trust domain
  - Sometimes with an eye towards extensibility, sometimes not
- QuickSilver folks:
  - Transactions as the basis for everything for safe, easy multi-node scaling
  - Emph: This is a distributed OS!

DB folks: Stonebraker81

- "Operating System Support for Database Management", M. Stonebraker, CACM 24(7) 1981
- OSes do
  - Buffer pool mamagement
  - File system / layout / etc. (and buffering)
  - Scheduling
  - Virtual memory / paging / swapping
- Consider buffer mgmt (like in Exokernel argument)
  - OS provides ~LRU buffer mgmt
  - Prefetching for sequential access
  - Transparent (sometimes controllable - madvise)

Multiple Buffering

- DB maintains its own buffer cache
  - Often knows better what will be ‘hot’ - e.g., internal B-tree nodes, etc. May know what it will need to re-visit during query.
- Replacement policy mismatch
  - Sequential scan: MRU/Random >> LRU sometimes
  - Looping scan: fix N (as many as can) pages
  - Biased random accesses: LRU

Prefetching?

- OS’s only guess: sequential
  - Wasted effort
  - Prefetched pages may kick out cached, useful ones!
- But DB usually *knows* what it wants next

Crash Recovery

- Saw last time interaction between buffer mgmt and recovery
  - DB may want control over buffering/eviction to make recovery simpler
  - DB *does* need control over forcing data to disk
    - At least for the log
    - Must control order of physical writes to disk for atomicity

QuickSilver is Distributed DB-OS

- OS Transactions (atomicity)
  - OS (nonDB too) services in transactions (Tid) defined by (reliable, inorder) messaging
  - Incl. window mgmt, virtual terminal service, nameservice & other less-durable services than DB, FS
  - Worry about overhead
  -

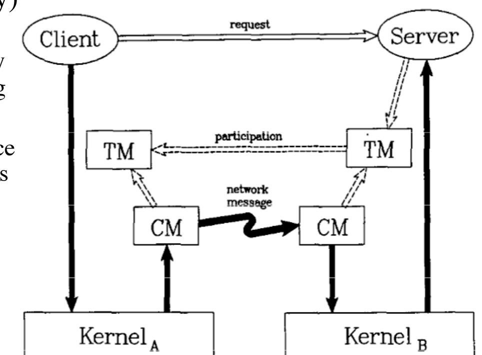

Fig. 2. QuickSilver distributed IPC.

Arguments for transactions

- Timeouts are hard to program
  - Complicated client logic
  - Too long - slow recovery; too short - false crash detection. Slow server vs crash? Quick-response heartbeats + timeouts? starting to get complicated.
- Want uniform recovery code
- Stateless services are too constraining; some actions can't be made idempotent (e.g., locks)

Examples

- Saving a file atomically in an editor (no temporary file; no need for link-rename tricks)
- Compiling - either .o file is complete, or doesn't exist, despite failure of other nodes involved in compilation
- Easier dist FS operation -- separate metadata/data servers, etc.

This is complex

- Transaction could cross multiple serviers
  - That's the whole point! :) Need 2-phase commit
  - Recursive transactions -- server issues RPCs in order to serve client
  - Some servers can't provide 2PC - "volatile" state may be lost (window state, etc.) across reboot.

Quicksilver Recovery

- Transaction Manager
  - Primary focus of paper is distributed commit for atomicity of complex distributed operations in OS
  - Specialized to lighterweight/less-durable services to save overhead
- Log Manager
  - General service for write-ahead logging, used by commit processing & each service's recovery scheme, archived if full
  - Intermingled for all services on a node, group commit by Tid
- Deadlock Detector
  - Detect resource deadlock via lock cycles & abort some transaction
  - Not implemented as OS generally doesn't (ad hoc isolation)
    - Serializability is rarely part of OS too, left to application services
- Connection Manager
  - Provides exactly once, src-dest in-order delivery of IPC/RPC (synch=original RPC, asynch=polling/callback, message=1-way)

Classes of Services

- Recoverable (traditional)
  - Classical DB services, file systems using transactions
- Volatile/non-durable (lightweight)
  - Simple services whose state does not survive failure
  - Window or virtual terminal manager
- Replicated, volatile/non-durable
  - Recovery done with pure replication, with custom recovery protocols, and slow all-failed recovery
  - Failure of one of these services does not necessarily fail transaction
- Long running app “services”
  - Checkpoint services for app restartability

- Servers declare as stateless, volatile or recoverable

Distributed Transaction Commit

- Basic idea: 2 phase commit
  - Root/coordinator initiates commit processing with vote requests through graph
  - Each node does its own logging as needed, propagates voting & waits for all child nodes to vote before it becomes “prepared” and votes
  - When root is prepared, result is determined (phase 1)
  - Result is communicated in “end” round (phase 2), triggering UNDO if result is abort & terminating transaction when all have accepted result
  - note hang in phase 1 if coordinator is partitioned

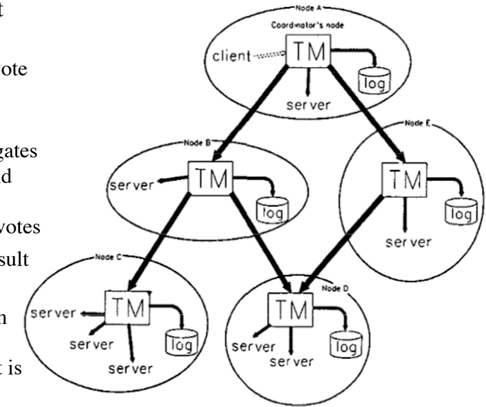

Fig. 3. Structure of a distributed transaction.

Failures

- Participant before prepare:
  - Likely abort (it hasn’t sync’d volatile state)
  - After prepare, before end-commit:
    - When rebooted, must find out final status and either REDO or UNDO - can do so b/c state was saved
- TM? Uhh. Guess it better restart
    - Tx probably aborts then
    - “Coordinator failure ... can cause resources to be locked indefinitely.” (ow)
- Coordinator migration: clients are least reliable nodes for coordinator
  - Rotate graph so more appropriate (fault tolerant, stateful) node is coordinator
- Coordinator replication
  - Coordinator is single point of failure for service commit
  - Can interpose mirror processing for replicating coordinator to reduce risk

Commit Specializations

- 1 phase commit services
  - Simple volatile services get only 1 message: end
- Truncated 2nd phases based on vote
  - Commit-RO if nothing to recover & no interest in 2nd phase
- Transaction graph cycles
  - First invocation votes for real, rest give Commit-RO
- Transactions continuing after commit
  - Real systems have timeout, poll, user input etc messaging that are not stateful/part of last transaction; cannot cause abort (always votes yes)
  - These requests are identified & allowed after commit starts; otherwise subsequent “voting” of committed transaction is to abort subsequent work
  -

Eval

- Describes a real system in detail; an experience report mostly
  - On the cusp of acceptable in 1988, but this is a big, novel system so it got slack
- Small amount of microbenchmark measurements
  - Hard to do better without accepted benchmarks, but that is the standard today
  - Notice how expensive the transaction-based services are relative to underlying OS services -- distributed database services are usually avoided if possible because of this -- generality and uniformity are not compelling enough

?s

- Structuring *everything* this way: heavy!
- Are there non-transactional services?
  - Window manager?
  - Output to user? Input from user? Network traffic to/from external hosts?
- Effect on software? What do clients and servers have to do differently? Is it likely to meet goal of reducing recovery code? Is it actually useful?

Three Phase Commit (non-blocking)

- Skeen83
- Blocking case is coordinator partition after begin-commit
  - Add another phase for "all have agreed"
  - Stuck in w2 can timeout & abort
  - extra messaging!