

Failures and Recovery

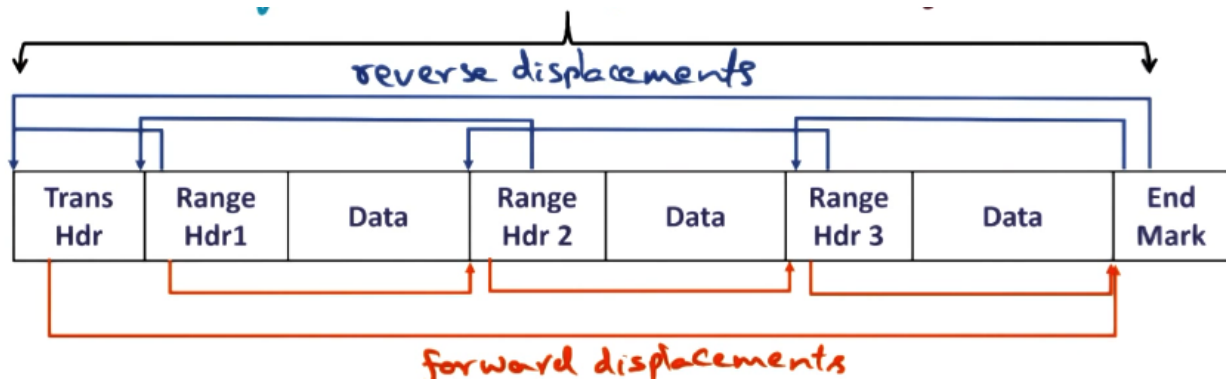
Lightweight Recoverable Virtual Memory

- * Lightweight Recoverable Virtual Memory Introduction
 - How do we build systems that survive crashes?
 - + Hardware, software, power failures
 - LRVM: Persistent memory layer in support of system services
- * Persistence
 - Why?
 - + Need of OS subsystems (filesystem inodes)
 - + Log memory to disk, but need to keep it consistent with changes in memory
 - How?
 - + Make virtual memory persistent
 - + All data structures contained in virtual memory become persistent
 - + Subsystems don't have to worry about flushing to disk if it's handled by the OS
 - Who will use it?
 - + Subsystem designers if it is performant
 - How to make it efficient?
 - + Writing every change in memory to disk incurs significant overhead due to the latency of writing to disk (seek, rotational)
 - + Use persistent logs to record changes to VM (similar to xFS)
 - + Only write logs of the changes, buffer to save them all at once to solve the small write problem
- * Server Design
 - Distinguish between persistent metadata and normal data structures
 - Create external data segments to back persistent data structures
 - + Applications manage their persistence needs
 - Designers can make regions of virtual memory to segments on disk
 - + Designer's choice to use single or multiple data segments
 - Mapping from virtual address to external data segments is 1-to-1
 - + No overlap in occupancy of virtual address space
 - + Simplifies design
 - Application can map and unmap whenever needed
- * Recoverable Virtual Memory Primitives
 - Initialization
 - + initialize(options): Log segment to be used for persistent data
 - + map(region, options): Virtual address to external data segments
 - + unmap(region): Deletes a mapping
 - Body of server code (like a critical section)
 - + begin_xact(tid, restore_mode): Start of changes to log
 - + set_range(tid, addr, size): Only using a portion of address range
 - + end_xact(tid, commit_mode): Log changes to log segment
 - + abort_xact(tid): Discard changes to persistent data structures
 - GC to reduce log space (done by LRVM automatically)
 - + flush(): Provided for application flexibility
 - + truncate(): Provided for application flexibility
 - Miscellaneous
 - + query_options(region)
 - + set_options(options)
 - + create_log(options, len, mode)
 - When a developer is making changes within a transaction, RVM commits

- the changes to a redo log in the log segment
 - + These are only committed to disk if the transaction isn't aborted
 - + Committing to disk happens at opportune times
 - + Truncation: Throwing away the redo log after it's committed
 - + Developer can explicitly manage its redo log as a way to optimize the use of disk space
- Small set of primitives that are easy to use and performant
- Transaction: Intended for recovery management, doesn't need all of the properties associated with typical database transactions (ACID)
 - + Atomicity
 - + Consistency
 - + Isolation
 - + Durability
- RVM doesn't allow for nested transactions or support concurrency control
 - + Developer must implement at a higher level if needed
- * How the Server Uses the Primitives
 - Initialize address space from external segments
 - begin_xact(tid, mode);
 - + set_range(tid, base_addr, #bytes);
 - + write metadata m1; // contained in range
 - + write metadata m2; // contained in range
 - end_xact(tid, mode); // can also be abort
 - When developer calls set_range, LRVM creates an undo record
 - + Copy of virtual address space for the specified number of bytes
 - Mode specifier allows user to specify to RVM whether transaction will ever abort
 - + If developer is certain transaction won't abort, can specify a no_restore mode so RVM knows not to create an undo record
 - When the application is writing metadata, no action is needed by LRVM
 - + Changes happen directly to the virtual address space of that particular process where the in-memory copy of the persistent data structures are living
- * Transaction Optimizations
 - No-restore mode in begin_xact
 - + No need to create in-memory undo record
 - No-flush mode in end_xact
 - + No need to do synchronous flush redo log to disk
 - + Lazy persistence (will be persistent, but not at the point of end_xact)
 - + There's a period of vulnerability between end_xact and flush, but this provides improved speed by removing synchronous I/O
 - + "Transactional systems perform well when you don't use transactions"
 - Use transactions as insurance
- * Implementation
 - Lightweight in terms of transactional properties
 - No undo/redo value logging
 - + Undo log: Creating an undo record of changes only in memory, not on disk (only for duration of transaction)
 - + Redo log: All changes to different regions between begin_xact and end_xact (also in memory, then flushed to disk)
 - On commit: Replace old value record in virtual memory with new value records (automatic based on how RVM works)
 - + Must undo changes if a transaction aborts
 - Creating the undo log is optional if the developer guarantees that the

transaction won't abort

- Writing the redo log to disk can be done lazily (don't block on call to end_xact)
 - + Gives a window of vulnerability where data can be lost
- Can traverse redo log in both directions to provide flexibility



RVM Implementation

* Crash Recovery

- Redo log has transaction header
 - + Between transaction header and end mark contains all of the changes that have been made by a critical section
- Resume from crash
 - + Read redo log from disk
 - + Apply to external data segments in memory
- All of the needed information is contained in the redo log

* Log Truncation

- If time between crashes is long (as we hope), log segments build up
 - + Redo log will also build up
- When a crash happens, apply to redo log to disk to clear it
 - + Clogging disk space, unnecessary overhead
- Truncating the log: Read logs from disk and apply to external data segments so the logs can be discarded
 - + Simply apply crash recovery algorithm (same logic)
 - + Read redo logs into memory and apply to data segment
- Log truncation: Perform in parallel with forward processing
 - + LRVM splits log record into epochs
 - + Truncation epoch: Crash recovery is using this
 - + Current epoch: Server is using this
- Biggest challenge in LRVM is log truncation code
 - + So much coordination involved

* Lightweight Recoverable Virtual Memory Conclusion

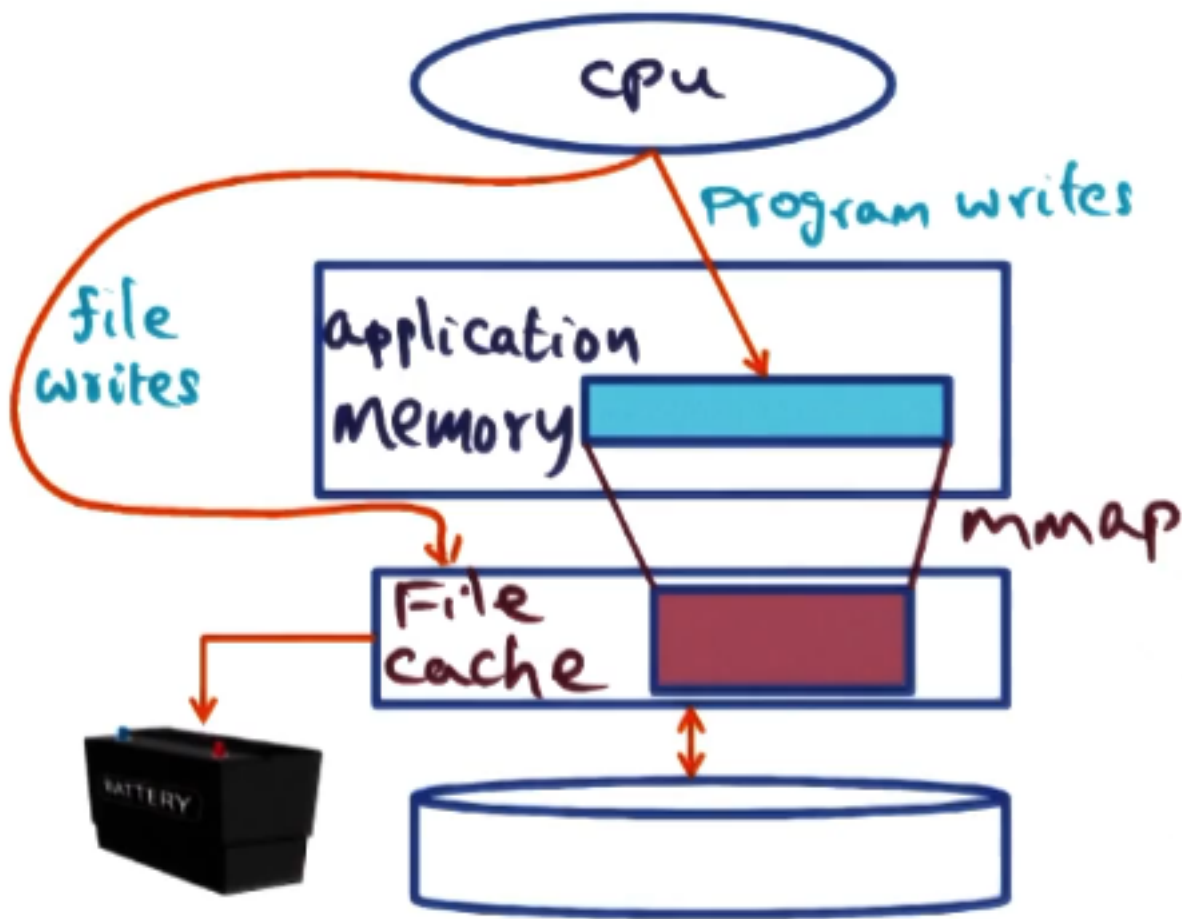
- LRVM is classic systems research example
 - + Understand pain point and create a solution
- Pain point: Managing persistence of critical data structures
- Solution: Lightweight transactions without typical ACID properties

RioVista

* RioVista Introduction

- Synchronous I/O makes transactions heavyweight in LRVM even though the

- semantics of transactions have been simplified considerably
 - RioVista's goal is performance-conscious design of persistent memory
- * System Crash
 - Two problems concerning failure
 - + Power failure: Can we throw some hardware at the problem and make it disappear? (UPS)
 - + Software crash: Reserve a portion of main memory that survives crashes
 - RioVista is only concerned with software crashes
 - + Makes transactions cheaper
- * LRVM Revisited
 1. begin_xact: Memory copy of portion of memory by LRVM
 2. body: Normal program writes
 3. end_xact: Disk copy by LRVM, get rid of undo logs
 4. log truncation: Redo logs -> data, get rid of redo logs by LRVM
 - Upshot: 3 copies by LRVM to persist data
 - + Biggest vulnerability is power failure if using no-flush option
- * Rio File Cache
 - Rio: Using battery-backed DRAM to implement a persistent file cache
 - + File system uses file cache holds data brought from disk
 - + Persistent: Use UPS to persist file cache data
 - OS buffers writes to DRAM and writes to disk at opportune times
 - + User has to use fsync to force writes to disk
 - mmap: Normal application memory becomes persistent
 - + Backed by battery
 - Using battery-backed DRAM means no synchronous writes are required

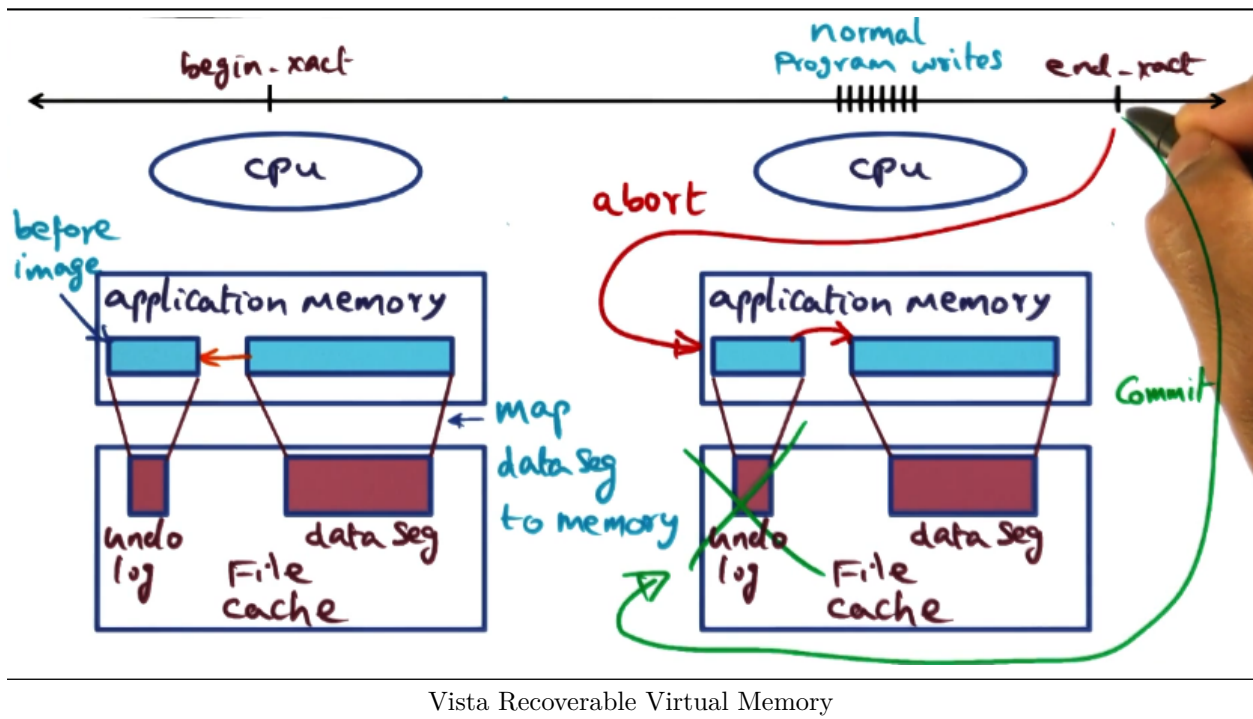


Rio File Cache

* Vista RVM on Top of Rio

- Vista: RVM library on top of Rio
- Semantics are identical to LRVM semantics, but the implementation takes advantage of the fact that it's sitting on top of Rio
- When mapping virtual memory to a data segment, it's mapped to the file cache which is already guaranteed to be persistent
- Specify address range to be persistent at the point of `begin_xact`
 - + Create an in-memory copy of the memory to be modified, which serves as the undo log
 - + Undo log is mapped to file cache (persistent by definition)
 - + Changes during the body of the critical section are already persistent
- At `end_xact`, changes are committed
 - + In Vista, changes are already committed because they're in persistent memory
 - + Faster because there's no need to write to disk synchronously
- If the transaction aborts, the undo record created at the beginning of the transaction is simply copied back to memory
- Rio+Vista is fast because there's no disk I/O
 - + Everything is memory resident and can be written back to disk

asynchronously



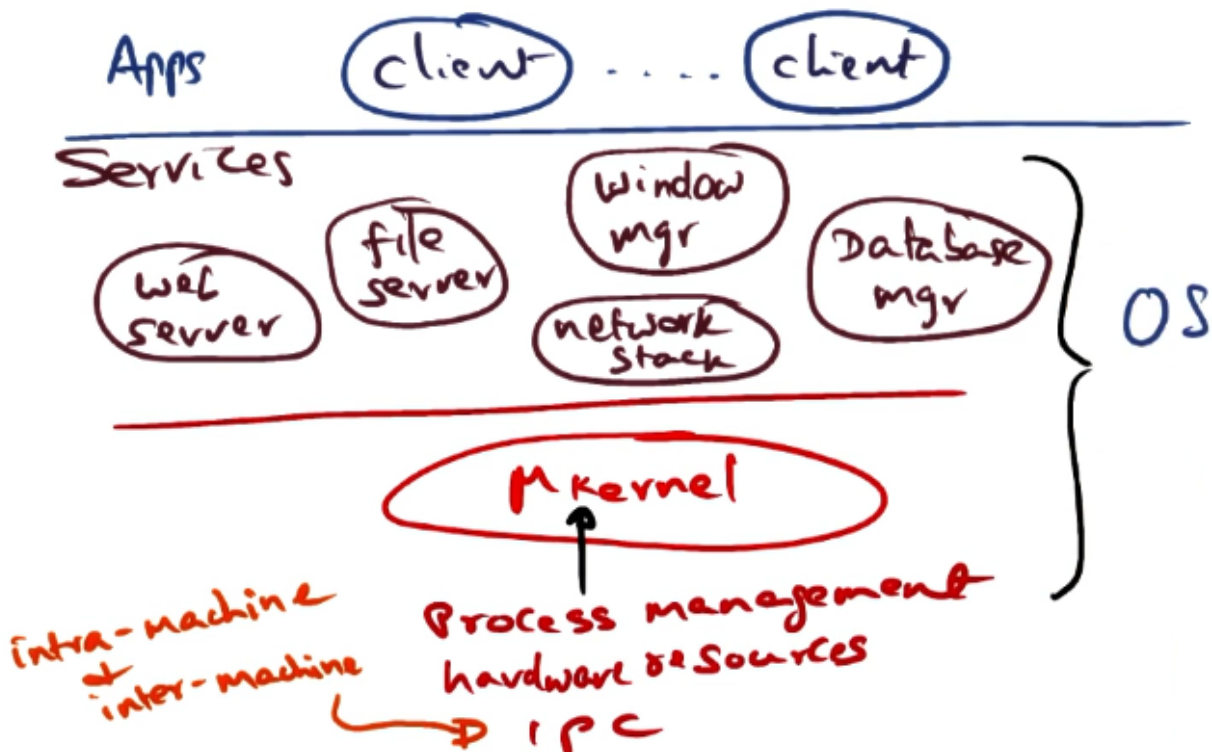
Vista Recoverable Virtual Memory

- * Crash Recovery
 - Treat like an abort
 - + Recover old image from undo log (survives crashes since it's in Rio file cache)
 - Crash during crash recovery?
 - + Idempotency of recovery, so there's no issue
- * Vista Simplicity
 - 700 lines of code in Vista compared to 10000 in LRVM
 - + Performs three orders of magnitude better than LRVM (no disk I/O)
 - Simpler
 - + No redo logs or truncation code
 - + Checkpointing and recovery code is simplified
 - + No group commit optimizations
 - Upshot: Simple like LRVM but performance efficient
- * RioVista Conclusion
 - Shows that by changing the starting assumptions of the problem, you can arrive at a completely different solution

Quicksilver

- * Quicksilver Introduction
 - Making recovery a first class citizen in an OS design, not an afterthought
 - Performance and reliability are generally considered to be opposing concerns
 - Quicksilver's approach is that if recovery is taken seriously at the initial design, you can have both
- * Cleaning up State Orphan Processes
 - "Not responding" windows come from programs not being hygienic

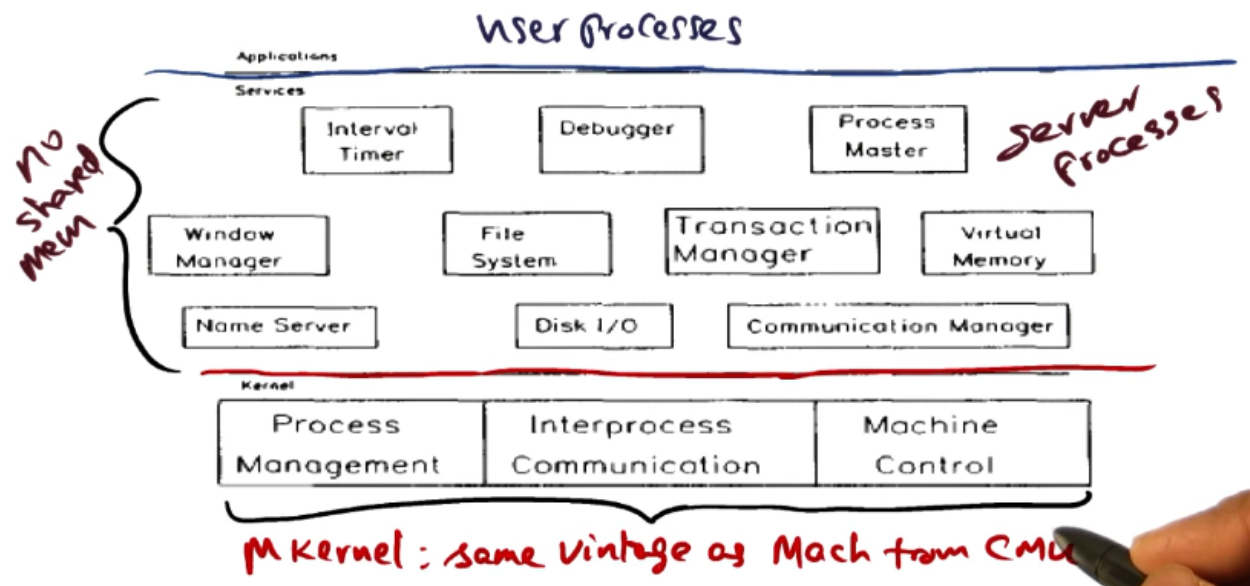
- When an application closes, it needs to clean up any resources it was using
- LRVM and RioVista only target state that needs to be persisted across system crashes
- NFS is stateless; server doesn't maintain any state pertaining to the clients
 - + The server cannot know about what state the client is reliant on when a transaction aborts. This state may live forever if a client application crashes
- * Quicksilver
 - Quicksilver identifies many problems that we face with our everyday computing with orphan windows and such
 - Built by IBM in the early 80s
- * Distributed System Structure
 - Applications interact with system services, which interact directly with the microkernel
 - Structure provides extensibility and high performance



Distributed System Structure

- * Quicksilver System Architecture
 - Quicksilver was architected to be very similar to modern day designs
 - + No shared memory
 - + Server processes all sit above microkernel
 - + Microkernel; same vintage as Mach from CMU
 - 80s were moving from CRTs and mainframe to desktop computers
 - Quicksilver proposes transaction as a unifying concept for recovery management of the servers

- + Transaction manager is service provided by OS to manage this



Quicksilver Architecture

* IPC Fundamental to System Services

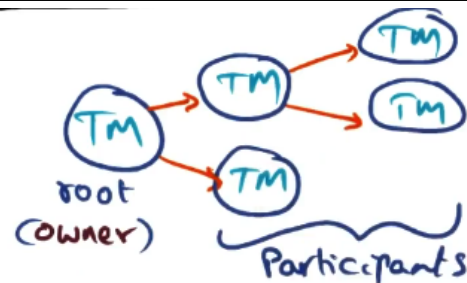
- Service-q used to handle IPC; client adds a request to service-q and initiates an upcall to the server. Server executes request, completion goes back to service-q, and OS gives control back to client
 - + Created by server
 - + Any process can connect and any server can service requests
 - + No loss of requests or duplicate requests
 - + Service-q is globally unique, so client doesn't need to know where in the network the requests is being serviced
 - + Supports synchronous and asynchronous requests
 - + Very similar to RPC (invented around the same time)
- Recovery mechanism is intimately tied to RPC
 - + Client/server interactions must use IPC
 - + Binds recovery with IPC to make it cheaper
- Quicksilver created in early 1980s, paper not published until 1988
 - + Industry only published when the system was finished
 - + Nowadays, everybody publishes often

* Bundling Distributed IPC and Transactions

- Transaction: Secret sauce for recovery management
 - + Lightweight, not heavyweight notion associated with databases
 - + LRVM took their ideas for transactions from Quicksilver
- IPC calls are tagged with transaction headers
 - + Want all failures to be recoverable, clean up state from clients, servers, communication manager, etc.
- Communication manager: One per node, handles IPC
 - + Transaction data is piggybacked on top of regular IPC so there's no additional overhead
- Transaction manager: One per node, handles tracking local changes to commit/abort as part of a transaction later
 - + Creator of transaction is default owner of transaction, coordinator

- Transactions need to clean up windows started by a window manager, file pointers opened by the filesystem, any additional state
 - + Transaction manager needs to track this state across an entire transaction tree

The diagram illustrates a replicated system architecture. At the top, two ovals represent the 'client' and 'server'. Below them are two boxes labeled 'Kernel A' and 'Kernel B'. Each kernel contains a 'CM' (Configuration Manager) box. Above each CM is a 'TM' (Transaction Monitor) box. Arrows show the following connections: Client to both TMs; both TMs to their respective CMs; both CMs to their respective Kernels; and both Kernels to the Server. A double-headed arrow connects the two TMs, labeled 'Participate'. A double-headed arrow connects the two CMs, labeled 'reliable recoverable from link failures'. The TMs are also labeled 'root' and 'Participant'.



```

graph LR
    C1((client)) --> WM((window mgr))
    C2((client)) --> FS((file server))
  
```

Quicksilver IPC and Transactions

- Coordinator can be different from owner
- Client makes a call to a filesystem, filesystem makes call to data server
 - + Client is owner of transaction tree, filesystem and data server are participants
 - + Owner can delegate ownership to someone else
 - + Clients are most fickle; clean up will become difficult if client crashes, so it might delegate to another node
- Heavy lifting done by Quicksilver is facilitating recovery management through the transaction tree

- Transactions are inherently distributed
- Results in a graph structure of the transaction tree
- Different types of failures
 - + Client crashes
 - + Connection failure
 - + Subordinate transaction manager failed to report
- Transaction manager logs periodically to a persistent store to create checkpoint records for the client
 - + Useful for recovery of work
- When a transaction manager terminates a transaction, all of the nodes must clean up all of their state

9

- When a transaction completes, the transaction tree is traversed to clean up any resources created to satisfy the request
 - + Initiated by coordinator for commit or abort
- Down the tree (initiated by coordinator)
 - + Vote request
 - + Abort request
 - + End commit/abort
- Up the tree
 - + Response commensurate with request
- If the transaction tree is representing the client/server relationship for opening a window...
 - + If client crashes, coordinator will send an abort. Window manager will clear window since it's volatile
- If the transaction tree is representing the client/server relationship a writing to a file...
 - + If client crashes, coordinator will send an abort. File system will undo changes to disk using checkpoints
- * Upshot of Bundling IPC and Recovery
 - Reclaim resources: Service can safely collect all breadcrumbs left behind by failed clients and servers
 - + Have a tree of all of the state changes to undo
 - No extra communication due to piggybacking off of IPC
 - Examples
 - + Memory
 - + File handles
 - + Communication handles
 - + Orphan windows
 - Only mechanism in OS, policy up to service
 - + Low overhead mechanisms for simple services (window manager)
 - + Weighty mechanisms for services (file system)
 - In memory logs are written to disk by transaction manager
 - + Frequency is a performance consideration
 - + Can be more opportunistic if failures are infrequent
- * Implementation Notes
 - Log maintenance
 - + Transaction manager write log records for recovering to persistent state
 - + Frequency of "log force" impacts performance
 - Services have to choose mechanisms commensurate with their recovery requirements
 - + Can impact all clients due to synchronous I/O requirements
- * Quicksilver Conclusion
 - Ideas of transactions as a fundamental OS mechanism to bundle in state recovery of OS services found resurgence in the 90s with LRVM
 - + In 2010, used to provide safeguard against system vulnerability and malicious attacks on a system in a research OS called Texas
 - Commercial OS are always focused on performance
 - + Reliability takes a back seat
 - + Writing to a file is only in memory until it's flushed to disk
 - Storage class memory
 - + Nonvolatile, but with latency properties similar to DRAM