Mero Object Store Architecture: Technical

Dr. Nikita Danilov
Nathan Rutman
Networked Storage Solutions
http://goo.gl/3eaAlh

Issue v0.7 | 2015

Revision History

Revision	Date	Author	Subject
0.5	2014-04-25	N. Rutman	Fork from Mero Architecture WP
0.6	2014-05-06	N. Rutman	Rework to technical version
0.7	2015-08-28	N. Rutman	Seagate branding

Table of Contents

---

1. Abstract

Today's large-scale storage architectures are typically divided into three classes, each addressed by a different storage infrastructure. Big Data Analytics systems perform searches and syntheses of collected, unstructured data. Cloud-based storage systems use stateless RESTful APIs to store static content. High-Performance Computing systems use POSIX with “defensive IO” for long-running simulations.

While the storage requirements of these areas differ, there is also utility in overcoming the ‘siloing’ of data. For example, data gathered via a Cloud system or generated by an HPC simulation should be available for efficient analysis by an BDA application. A “horizontally” extensible system in which applications can interact with a universal storage system using their preferred semantics could allow access via multiple APIs and prevent this siloing.

Simultaneously, new hardware solutions are becoming commercially viable. Flash and other non-volatile memory technologies offer much faster access than traditional rotational storage,

but at a higher price point. CPU speeds are decelerating and core counts proliferating. As software designs increasingly reflect the reality that hardware failures are unavoidable, software must compensate for degraded componentry. A “vertically” extensible storage system could take advantage of non-uniform hardware technologies by eliminating the layering abstractions imposed by today’s software stacks.

To respond to these technology trends, and to meet the challenges posed by ongoing scaling up of storage requirements, we introduce a new object-based storage architecture named Mero. This paper focuses on the technical features of Mero; motivation and markets are addressed in a separate whitepaper.

2. Introduction

This whitepaper describes the architecture of Mero — a storage core capable of deployment for a wide range of large-scale storage regimes, from cloud and enterprise systems to exascale HPC installations.

The document serves two principal purposes:

- Describes the fundamental principles underlying the Mero architecture; specifically, the basis for the architecture and the challenges it attempts to address
- Discusses how to use the Mero architecture and the hardware and software technologies it enables

3. Background

Xyratex's ClusterStor division delivers high-performance, networked storage solutions for HPC environments. ClusterStor systems sold today integrate Lustre open-source distributed file system servers, embedded in high-density disk drive enclosures, and assembled as modular units into storage racks. A multi-rack ClusterStor system presents an extremely large, single-namespace file system offering state-of-the-art performance and storage capacity. ClusterStor systems were the first to reach the threshold of 1 TB/sec of continuous I/O and have scaled to store 30+ PBs of data[1].

ClusterStor's design goals include maximal storage density (based mainly on the drive enclosure hardware design) and best-in-class performance, based on carefully balanced component bandwidths. To leverage Lustre's failover mechanism (enabling data availability in

[1] https://cug.org/proceedings/cug2013_proceedings/includes/files/pap188.pdf

the event of server loss), ClusterStor systems use dual-ported drives that provide a data path to each server in an embedded server pair, which normally run in an active-active configuration. RAID software runs locally on each server and provides data redundancy in a RAID6 8+2 pattern; offering protection against the loss of 2 drives out of 8 (20% space overhead). The RAID system automatically allocates available spare drives and, when necessary, rebuilds data on them. In addition to Lustre's failover functionality, a separate high-availability (HA) software package runs on ClusterStor systems, enabling services to restart automatically on alternate hardware, if required. Additionally, Xyratex's proprietary GEM software administers servers, provides visibility into hardware health and performance, and collects diagnostic information when problems occur.

While ClusterStor systems work well for present-day, top-scale storage, this style of storage architecture show signs of reaching its design boundaries. Lustre software continues to reveal inherent limitations, typically in the form of bugs, as larger-scale systems are deployed. Many of these bugs are only revealed at higher scale, as greater pressure is exerted on system resources (thread counts, memory, network buffers, etc.). Another area of concern is the scalability of Lustre's timeout-based failover mechanism, which requires significant time to recover after a node failure, as system timeout thresholds must naturally increase with scale to cover broader variances. Additionally, relatively poor visibility into system activity makes it difficult for system administrators and developers to diagnose performance and stability issues.

To overcome the difficulties posed by using Lustre at scale limits and to address emerging trends in storage hardware (non-volatile memory, CPU speeds, core counts) and software infrastructures (MapReduce, cloud storage, burst buffers), we believe an entirely new exascale storage architecture, designed from the ground up, is needed. Mero, a new storage software technology under active development by Xyratex, embodies this new architecture. Mero is not a file system per se, but rather a flexible storage system base designed to meet the challenges outlined in this paper.

4. Quality Attributes

Mero was conceived as a storage system for supercomputers operating in the exascale[2] range, two decimal orders of magnitude beyond existing (2014) sizes of required data rates and storage capacities. The authors do not believe any other storage system existing today or expected in the near term (extrapolating from ongoing development) is capable of achieving either of these requirements.

Notably, many of the features required for exascale storage translate extremely well to less demanding environments: maximal data availability, low power consumption, hardware

[2] Exascale Computing. In *Wikipedia*: http://en.wikipedia.org/wiki/Exascale_computing

awareness, visibility into system behavior, and excellent performance and scalability are generally desirable features for many enterprise-class storage systems.

The essential qualities required for Mero were derived from two complementary sources that together have motivated development of an entirely new storage model:

- **Implementor view**
  Architects' long experience with state-of-the-art storage systems (dramatically different sizes and scales) and knowledge of their limits, bottlenecks and problems

- **User view**
  Requirements and use cases obtained from users (including application developers, system administrators and end users) collected systematically from 2009–2012 at Quality Attribute Workshops[3] in Paris, FR and Portland, OR. These workshops yielded participant consensus regarding the most important quality attributes[4] of a new storage architecture.

The latter source provides the fundament for the functional architecture specification or the "what" of the architecture. The former source determines the "how" of the implementation. In this section we describe the driving quality attributes for Mero.

4.1. Scalability

Storage system scalability can be classified along two dimensions: horizontal and vertical.

Aggregate system parameters (capacity, throughput, operation rates, number of objects, etc.) grow with the number of network-interconnected nodes. *Horizontal scalability* provides a uniform method to build systems of varying scale and of gradual system growth, but it is often limited by various single-node bottlenecks and hotspots. For example, a communication bottleneck occurs if every I/O operation requires interaction with a metadata or lock service (as in early NFS versions); a processing and storage bottleneck occurs if the system supports only a single metadata service, like Lustre (before version 2.4).

System parameters also grow in proportion to the capabilities of individual nodes, that is, system capabilities scale with node CPU cycles, memory, bus bandwidth, etc. Limits to *vertical scalability*, the ability of the system to take full advantage of an individual node's resources, are often imposed by concurrency control (locking), data structures and algorithms, data copying, and operating system limits. Vertical scalability becomes more important with the advent of

[3] Quality Attribute Workshops (QAWs) provide a method to identify a system's architecture-critical quality attributes, Software Engineering Institute, Carnegie Mellon University.
[4] Barbacci Mario, et al. *Quality Attributes* (CMU/SEI-95-TR-021). Software Engineering Institute, Carnegie Mellon University, 1995. http://www.sei.cmu.edu/reports/95tr021.pdf

modern platforms that feature very large numbers of processor cores, multiple levels of memory hierarchies, and non-uniform components: CPU vs. GPU, NUMA, various degrees of primary store persistency and so on. Frequently, systems become gated by the single slowest component and suffer a relatively high amount of inefficient idle time in other elements.

4.2. Availability and Reliability, Fault Tolerance

Business and mission-critical systems commonly must provide uninterrupted access to data. To address this requirement, many storage systems implement data redundancy and recovery processes. However, as systems scale, individual component failures become more common requiring either more frequent recovery events, resulting in lower system availability, or additional redundancy paths, increasing costs and complexity. For example, a common design to handle failures relies on switching to a dedicated "recovery mode" in which the system manages a particular failure while postponing normal processing. This design suffers when the Mean Time Between Failures (MTBF) becomes comparable with the recovery duration - the system will essentially always be in a recovery mode. Similarly, the widespread tactic of HPC applications storing checkpoints to recover from crashes (“defensive I/O”) becomes ineffective when the MTBF falls below the time required to make a checkpoint.

Mero attempts to avoid this problem by eliminating this recovery mode from normal operations. Mero uses a distributed erasure coding algorithm (SNS) to provide data redundancy; this redundancy is used to actively recreate data on the fly for reads in the event of a temporary server or drive loss. For writes, Mero uses a feature called non-blocking availability based on a strong layout mechanism which redirects writes to other available drives if an in-use server or drive fails.

Frequently, fault-tolerance mechanisms are implemented on a per-subsystem basis - timeout-retry on a network, failover-restart of a server. Lacking a global perspective, correct identification of the cause of a failure is error-prone and responses therefore suboptimal. Because failure paths are rarely exercised and the combinatorial nature of the failure state of large distributed systems makes exhaustive testing impossible, typical testing methods are unlikely to uncover latent defects in fault tolerance code.

Mero implements a consensus-based high-availability subsystem designed to scalably maintain a single globally consistent view of the cluster state. Because this system is capable of distinguishing between local events (e.g. no network traffic on this NIC) and broader events (e.g. no network traffic from any server connected to this router), it is able to make more appropriate responses.

Additionally, no matter how well-designed a system is or how many redundant copies of data it maintains, it is possible for a system to end up in an inconsistent state - either because

hardware or other software behaved contrary to the assumed model or because of software defects. In this situation, the storage system must recover as much of the survived state as possible. This task is performed by a consistency check process (`fsck`, scavenger, etc.). For systems requiring strong consistency (e.g POSIX), traditional `fsck` methods take a system offline (to ensure data does not change) for a period of time proportional to the total storage space; in a large data center, this activity may significantly impact availability. Via its FDMI interface, Mero can implement an on-line consistency check process to verify system state without impacting system performance.

4.3. Observability

Even when a storage system is working correctly, users and administrators need tools to inspect and understand internal system behavior: to predict how the system would react to new workloads; to optimize existing applications (and the system itself); or to plan future installations, upgrades or configuration changes.

When the system behaves unexpectedly, the need to analyze system behavior is more pronounced. Distributed applications are notoriously difficult to debug as multiple distributed jobs in a large cluster tend to interact in unintuitive ways. Critical system parameters related to availability and reliability should be easily accessible for analysis.

Existing systems provide little help to achieve this observability, as they typically offer tools in a form of logs and ad-hoc runtime metrics. At the QAW meetings described above (2. Design Requirements), users gave consistently high rankings to these attributes: tools to inspect, analyze and model system behavior, and tools to determine the cause(s) of malfunction and identify appropriate corrective actions. Behavioral indicators should consist of first-class data, which can be analyzed either online or post factum. This data should be comprehensive enough to re-create an issue with sufficient fidelity to be able to test candidate fixes.

To address this requirement, Mero implements an Analysis and Diagnostics Database (ADDB), a first-class mechanism to efficiently track, collect, analyze, and simulate system activity. ADDB logs can be gathered and shipped, without including any customer data, to provide a complete record of system activity for off-site analyses.

4.4. Implementation Complexity

Experience with the development of the Lustre file system provides the insight that accumulated complexity, more than any other factor, hinders the rate of software progress. High rates of innovation and rapid code development quickly devolve into a slow pace of progress as originally expedient solutions reveal their limitations, requiring rework, refactoring and

workarounds. Over time, code growth (Lustre is now over 250,000 lines of code[5]) leads to such complexity that simply understanding the system in its entirety becomes impossible. This situation can only be avoided by implementing an extensive architectural design which focuses primarily on simplicity, layering and reusability, *before* any code is written. Additionally, a strong quality process must be used in all phases of development, from architectural design through coding and test, to minimize the number of latent defects.

For Mero, we spent more than a year working on the architectural design, before a single line of code was written, to identify commonalities in feature requirements. A small number of key, flexible mechanisms were designed with clear roles, boundaries, interfaces, and utility.

4.4.1. Abstraction Unification

Mero is designed to use global, standardized mechanisms wherever possible. This design principle is reflected in the highly "generic" nature of the code modules: software architecture is identical between client code and server code, data and metadata are handled via the same mechanisms (for example, in the transport layer), resource borrowers can, in turn, re-lend borrowed resources, local and remote caches are treated similarly, proxy servers and back-end servers provide the same services with the same code, etc. Obvious benefits to this unified approach include reduced line count, fewer corner cases to hide bugs, greater developer familiarity with common mechanisms and more extensive use of code paths.

Unification was achieved by analyzing a number of critical use cases and resulted in the identification of several key abstractions (resource, transaction, container, layout and so forth) that capture the desired behavior.

Examples of code reduction owing to unification include:

- Client and server have the same organization with some shared code (operations log, request handler and resource management).
- Kernel and user code are identical, with differences hidden in the portability library.
- No separate data and metadata servers. Mero containers can store data blocks or an entire metadata database.
- All resources are abstracted and handled uniformly via the same code:
    - File identifiers
    - Quotas
    - Cache memory
    - Locks on file extents
    - Locks on metadata
    - Device space grants
    - Network and storage bandwidth allocations

[5] Barton, E. (2010). *Lustre Development* [Presentation slides] . Retrieved from http://wiki.lustre.org/index.php/File:Lustre_Development_Barton_LUG_2010.pdf

    - Configuration parameters, etc.

  The global resource manager code manages ownership and resolves conflicts.
- Processing paths are implemented as non-blocking state machines, to minimize the number of threads and eliminate the possibilities for deadlocks.
- Generic "Loom" pipeline (composed of a number of distributed state machines) performs distributed data reconstruction for all forms of internal moving data: erasure code computations, data migration, replication, layout flattening and proxy re-integration.
- Same notion of layout encompasses various methods of storing data and metadata. Common layout manager code is shared by parity de-clustered network RAID, metadata mirroring, encryption, compression, de-duplication, etc.
- Distributed transaction mechanism is used to define consistency points and to recover from various transient failures. This mechanism is reused by data and metadata operations requested by users as well as by internal operations during network RAID repair or FDMI.
- Containers are used for data organisation and migration. Examples of containers include:
    - Volatile cache on a diskless client
    - Persistent cache on a proxy-server
    - Storage device on a server

  Generic container manager code handles container identification, location, migration (for example, moving a drive from one server to another) and merging (when a cache is re-integrated).

4.4.2. Coding Patterns

To help mandate implementation quality, Mero code layout uses highly standardized patterns and methods: for example, every major module and data structure has "init" and "fini" setup and teardown functions; every function has pre- and post-condition checks to ensure entrance and exit requirements are met; functions are meticulously documented using Doxygen; even function names follow strict rules. Adhering to these standards results in more readable code and fewer programmer mistakes.

Each module is equipped with a set of unit tests, which must pass before code is committed to the central repository. Mero uses a test-carrying code technique, whereby data structures are associated with extensive invariant functions and checked at runtime to detect violations as soon as possible.

4.5. Extensibility

Extensibility is the ability of a system to track evolving requirements; adding new features and capabilities in a timely manner without sacrificing system stability.

Similar to scalability (2.1 Scalability, above), extensibility can be roughly classified along horizontal and vertical axes. A system is *horizontally extensible* when functionality can be added to it without adversely affecting the system's existing functionality. A system is *vertically extensible* when it can be integrated with (or, ideally, obviate the need for) different layers of software stacks “above” or “below” the system.

4.5.1. Vertical Extensibility

A traditional approach to decreasing complexity is to divide software into layers that progressively abstract low-level commands to upper-level functions. However, today's distributed storage systems are burdened with a layering structure that was developed for local storage: device driver, block device, RAID, local file system, distributed file system server, network layer, distributed file system client, kernel virtual file system and POSIX system call. While each layer may provide some functionality, imposing this layering structure on distributed file systems introduces inefficiencies and complexities rather than removes them. For example, a network-distributed erasure coding system does not fit neatly into any of these software layers.

The creators of ZFS made a similar observation: "...the standard layering of the storage stack induces a surprising amount of unnecessary complexity and duplicated logic."[6] Similar to ZFS, Mero decides where and when to write to storage devices itself, rather than relying on additional layers in the software stack.

The storage stack itself is just one component in a complete software stack, with multiple layers of system libraries, cluster middleware, application-specific libraries and application code above it. As systems grow in scale and the sophistication of their services increases, the additional burdens begin to reveal the limitations of the storage system API. For example, on very large systems, even basic applications such as “backup modified files” cannot be efficiently implemented over the standard POSIX interface. As a result, users have to code around the interface's limitations, giving rise to a cottage industry of various libraries that attempt to compensate for the deficiencies of the interface (for example, data containers like HDF5, “middleware”-like burst buffers, out-of-band I/O scheduling like MPI/IO, etc.). Invariably, these libraries are suboptimal for a number of reasons:

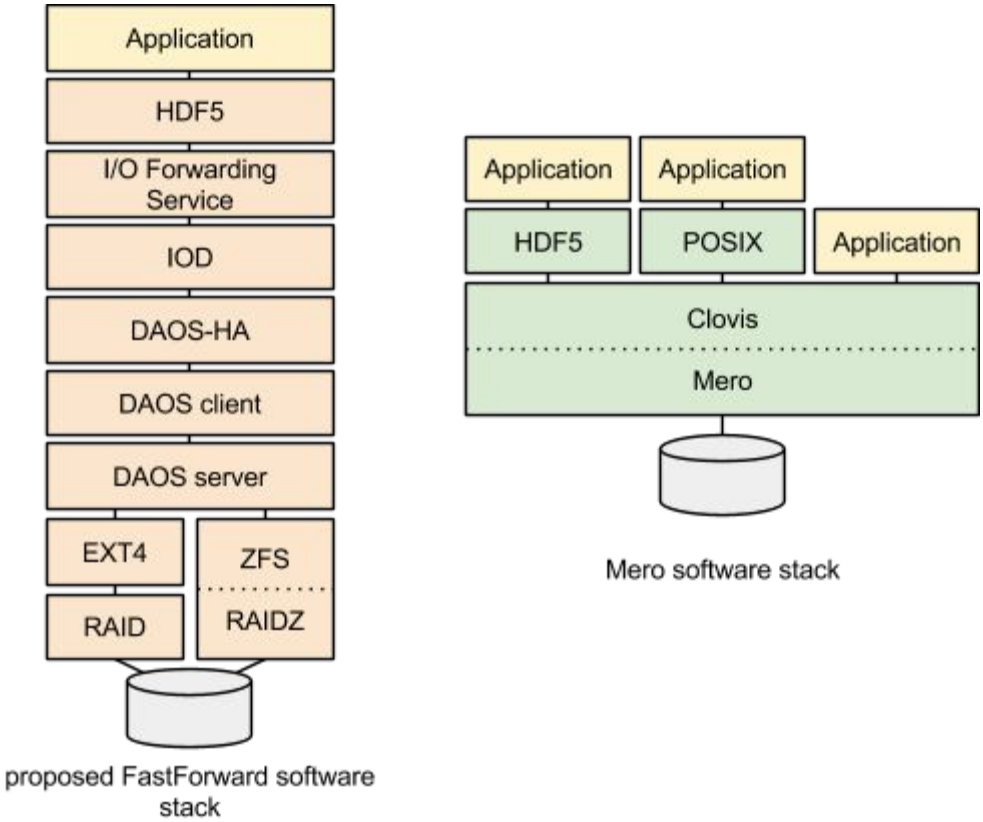
- They cannot use important primitives, such as transactions and distributed locking, which are not exported through the API and which the storage system implements internally. To compensate for this limitation, these primitives have to be re-implemented on top of the interface, which is inefficient and often incorrect.
- They have no access to global workload characteristics (only available at the lower layers) and, hence, cannot efficiently utilize shared resources (such as a common cache).

[6] Bonwick, J. *Rampant Layering Violation?* (2007)
https://blogs.oracle.com/bonwick/entry/rampant_layering_violation

- They are not integrated with consistency- and coherency-protecting mechanisms within the storage system, resulting in ill-defined behavior during failures.

Mero removes many of the traditional layers of storage systems, eliminating the associated limitations and bottlenecks. Mero implements its own containerized storage format directly on storage devices (e.g. block devices), removing the "backing filesystem" traditionally used by systems like Lustre (Ext4 or ZFS), Ceph (BTRFS), or Swift (XFS). Mero also implements network-distributed erasure coding, removing the traditional RAID stack. The distributed server, network transport, and distributed client are all included, eliminating e.g. NFS and CIFS overheads.

More importantly, at the "top end" of the storage stack, Mero provides a rich transactional API named Clovis. While at first we expect the majority of usage to take place via more traditional storage APIs (POSIX, S3, CDMI) implemented on Clovis, the most demanding applications can write against Clovis directly (or use e.g. exascale libraries), and can therefore optimize the IO performance for their particular consistency requirements at a granular level.



Mero offers *vertical extensibility* into applications[7]

[7] FastForward stack from Barton, Eric: Exascale Computing Vision

4.5.2. Horizontal Extensibility

Today's storage systems are typically designed for a particular usage pattern: clustered HPC POSIX or RESTful object storage or MapReduce at-compute locality. Historically, these interfaces tend to dictate the storage architecture, so a system designed for one pattern is generally not flexible enough to perform well in another domain. As a result, it becomes difficult for data ingested or created via one realm to be easily accessed from another application; for example, data is frequently copied from a site-wide HPC POSIX filesystem into a Hadoop cluster for Map/Reduce analyses. Partitioned access becomes more problematic as the size of data grows and the types of analytics systems multiply. A horizontally extensible system should provide multiple access methods.

Mero offers access to storage through the unified Clovis API, upon which arbitrary plug-in access methods may be built. The key to this capability is the exposure of user-defined distributed transactions through the interface; depending on the needs of the user, the interface may be stateless (e.g. RESTful), synchronous, asynchronous, local, distributed, or epoch-like (see DTM). The Mero IO interface can be horizontally extended with new APIs as desired.

Similarly, another problematic area of today's storage systems is the difficulty of extending an integrated feature set. When a new feature (for example an audit log, an indexing system, or a consistency checker) is needed, it is generally integrated directly into the storage system code, resulting in long development and test times and increased system complexity. Invariably, significant extensions depend on internal system interfaces and, therefore, are rarely ever portable. (For example, the idea of a portable file system checker is completely alien to contemporary storage system architectures.)

Instead, a fixed, public interface able to integrate with storage system operations would simultaneously isolate the storage core while providing opportunity for future feature additions. The component of Mero that provides this horizontal feature extensibility is called the File Data Manipulation Interface, or FDMI. FDMI users can receive file operations records or inject file operation requests transactionally into the processing streams.

4.6. Data Locality

Existing storage system architectures usually assume a fixed scheme for the placement of resources, most importantly, system storage and compute power. For example, HPC installations historically use a client-server model with a static, global configuration consisting of a compute cluster with diskless clients that send requests to a separate storage cluster comprised of dedicated servers with attached storage devices. Alternatively, fully-distributed systems, such as MapReduce, attempt to distribute storage and compute cycles among all nodes participating in an operation. However, both of these models have significant limitations

inherent in their designs. Client-server models must transfer large numbers of messages (and large amounts of data) over networks, and introduce significant latency when storage resources are contended. Distributed models impose restrictions on the placement of data and compute jobs, and handle contended resources even more poorly than client-server models or not at all.

A Mero storage system is formed from fully symmetric instances that are capable of serving local resources and, at the same time, are fully optimized for distributed communication, including both RDMA data transfers and cross-node distributed transaction control. A Mero instance acting as a traditional filesystem client communicates with other instances on other nodes providing distributed Mero services; this “client instance” in turn may be acting as a server to other instances. This approach enables Mero to act not only as a traditional HPC system or a fully-distributed Hadoop-style system, but also allows for entirely new system capabilities, such as BitTorrent-like dynamic distribution of data and proxy servers providing close-proximity caching as well as support for WANs and disconnected operations.

5. Architectural Components

This section describes the major structural components of Mero and describes how they satisfy the quality attributes defined above.

5.1. Overview

Mero is based on an internal core set of components and services that together provide the capabilities to build scalable, distributed storage solutions. This core does not depend on any other software (local filesystems, RAID layers, web services, HA systems) other than the operating system and device drivers. The core provides for availability, performance, scalability, observability, and storage efficiency. The core components include:

- Storage Object (stob) - Module that abstracts details of the underlying bulk storage
- Key-Value Store (KVS) - Module that implements local transactional key-value store for small data (e.g. metadata)
- Distributed Transaction Manager (DTM) - Provides consistency in the face of transient network and node failure
- Resource Manager (RM) - Uniformly controls and distributes arbitrary types of resources and arbitrate conflicts
- Loom - Performs internal distributed cooperative data movement and transformation, including recovery from permanent storage failures.
- Layout Manager - Specifies how logical entities (objects and metadata structures) are mapped to lower-level storage.
- Server Network Striping (SNS) - Network-distributed erasure coding scheme

- Non-Blocking Availability (NBA) - Automatic storage redirection for writes
- Container Manager - Organizes data and metadata for migration and placement
- Analytics and Diagnostics Database (ADDB) - Collects information about system behavior for analysis and monitoring
- File Operation Machine (FOM) - Non-blocking state machine that executes storage operations
- File Operation Log (FOL) - Log that records executed operations
- High-Availability system (Hask) - Quorum-based, consistent view of cluster state

Upon this core two interfaces are built. The first, Clovis, provides the external data interface upon which other common interfaces or libraries can be built. Internally, Mero implements a generalized, distributed object storage service to persistently and reliably store large amounts of user data, optimized for *throughput*, and a distributed key-value store designed to hold arbitrary small bits of data (typically, metadata related to objects in the object store), optimized for *minimal latency*.
Possible interfaces to Mero, implemented on top of Clovis, might include (for example) POSIX, pNFS, CIFS, HDF5, CDMI, S3, or Swift.

The second interface, FDMI, is build around the core and allows for horizontally extending the features and capabilities of the system in a scalable and reliable manner. Extensions among various dimensions are possible: information lifecycle management (auditing, accounting, quotas, replication, indexing, backup), security (encryption, authentication), compression (deduplication, bespoke algorithms), availability (self-consistency check, background scrub), etc.

Figure 1 represents this structure graphically; the storage core, the Clovis interface on top providing vertical integration, and the FDMI interface around providing horizontal extensibility.

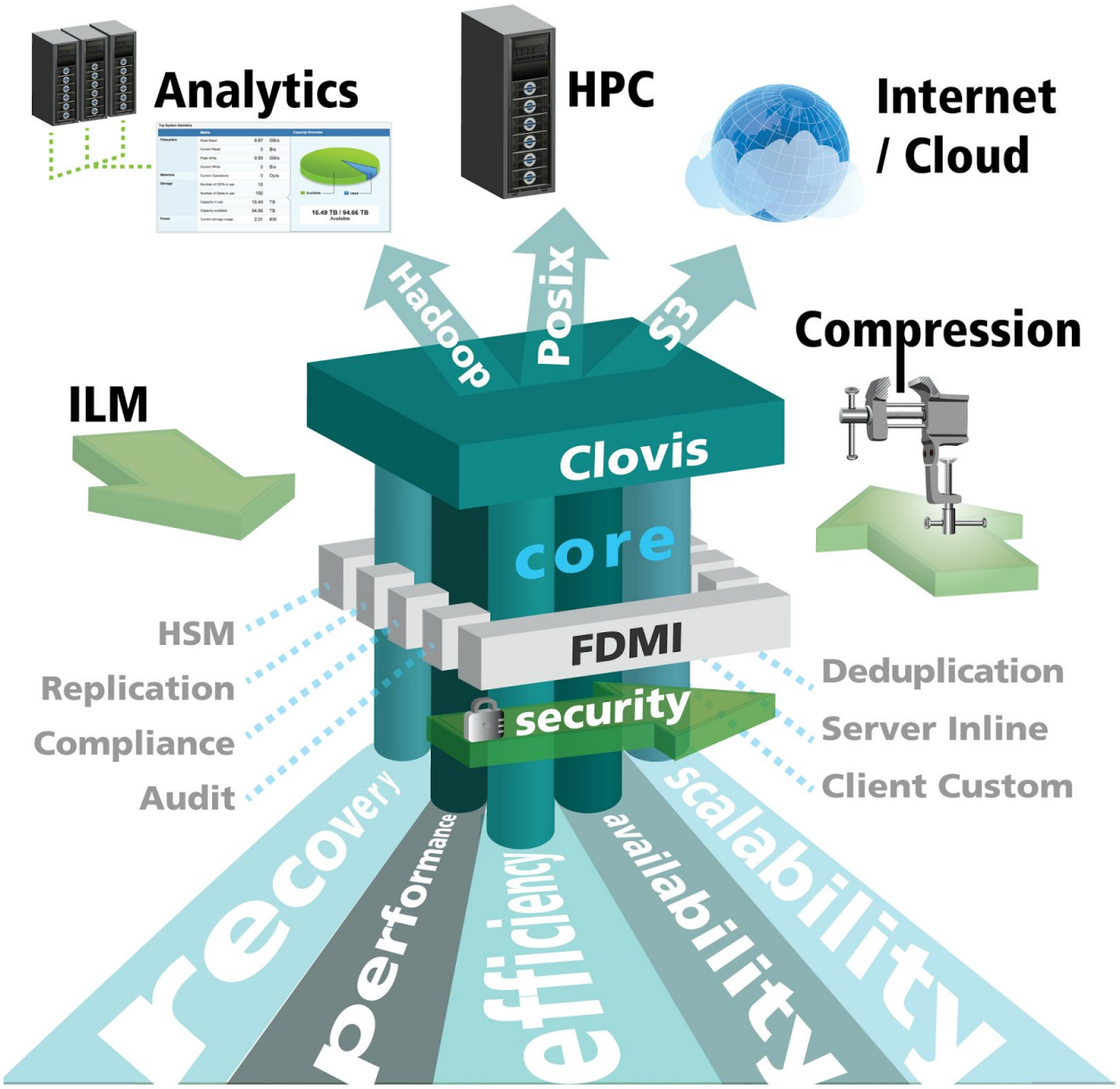


Figure 1: Mero core and interfaces

The following sections provide detailed descriptions of several key components.

5.2. Key-Value Store

Fast key-value store based on transactional memory

Mero uses an internal, RAM-based key-value store (KVS) to provide extremely fast access to object descriptors. Mero stores object metadata such as data layouts, access control

information, security features and other internally-used metadata in a compact, efficient form in KVS. Mero also provides a user API (via Clovis) for KVS, enabling storage systems users (such as a POSIX file system) to use KVS for arbitrary small metadata, for example, inode modification time. To provide the fastest access possible to this type of data, KVS operates in RAM.

The problems with RAM are well known, including limited capacities and volatility. To address RAM volatility, Mero has implemented a recoverable virtual memory system. Changes to KVS are logged transactionally to a persistent local device (disk or flash, if available) using the DTM. Until a key-value change has been persisted in the log, the DTM retains the ability to undo a transaction and any subsequent, dependent transactions. Logging is a serializing event, so optimal sequential I/O patterns can be used on the persistent device. To provide improved capacity, Mero's KVS is fully distributed across all nodes participating in the storage system; it scales horizontally with the cluster. (If the metadata load exceeds the cluster's memory limits, a paging mechanism can be used at the cost of some metadata performance.) The DTM is used to ensure that distributed metatdata operations behave atomically.

Additionally, KVS provides a straightforward integration path for non-volatile memory types, such as phase change memory. Because KVS operates directly on memory regions, bypassing all block and file system abstractions, it can easily leverage memory-like storage technologies.

Figure 7 is a schematic representation of how KVS maps storage objects (stobs) into regions of process address space (called segment).

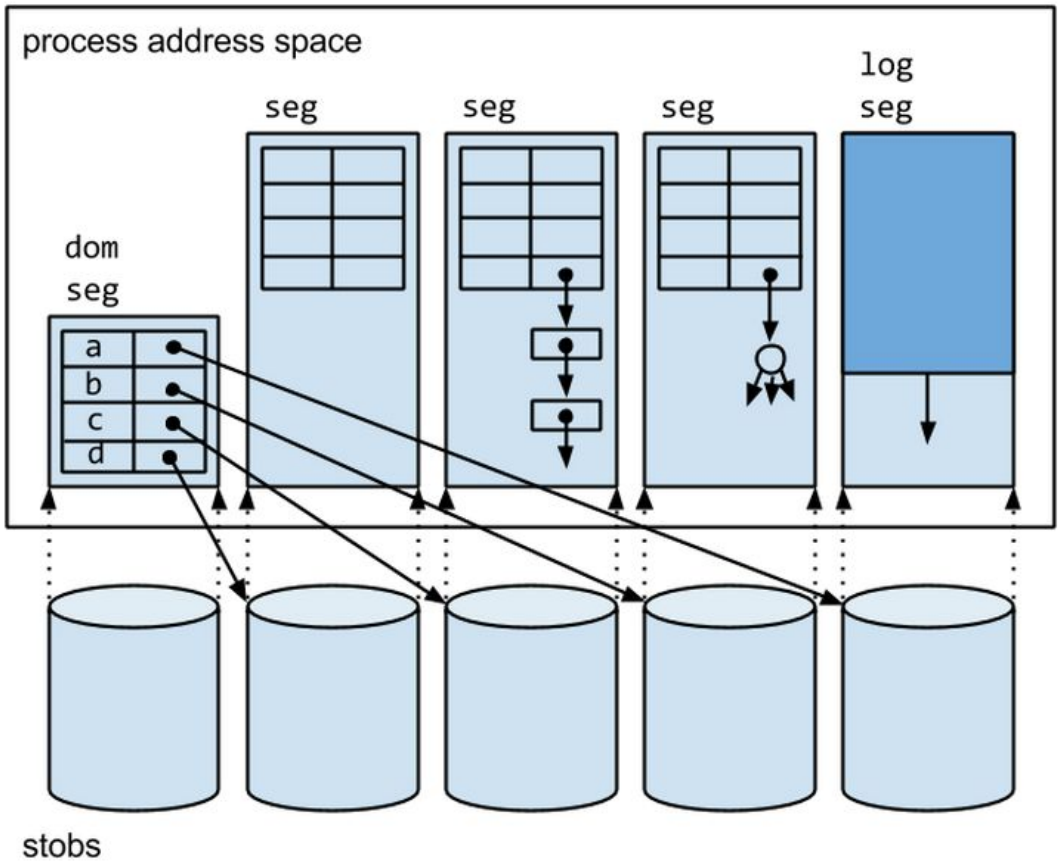


Figure 7: KVS data-structures and mapping from storage objects to the process address space

5.3. Distributed Transaction Manager

Provides consistency in the face of transient failures

The Distributed Transaction Manager (DTM) groups storage operations (writes to objects, metadata updates, etc.) in *transactions*. Transactions are atomic, that is, if a transient failure such as a node restart or a network partition occurs, either all or none of the operations in a transaction survive the failure. Transactions are the cornerstone of a manageable and reliable storage stack, because they isolate users from myriad failure scenarios and encapsulate error handling in one place. The key to usable transactions are strong atomicity guarantees that do not sacrifice scalability. To this end, Mero uses a patented distributed transaction algorithm, based on asynchronous background stabilization, that does not block ongoing operations.

This algorithm implements *epochs*, which are logical (Lamport) clock values causally ordering events in the distributed system. By comparing epochs, potentially dependent events can be determined quickly and without unnecessary communication.

Each operation executed by a Mero instance is placed transactionally in the instance's FOL. In the event of a failure, remaining Mero instances poll their FOLs to find the latest fully preserved epoch. The instances then roll back to this epoch by going through the FOLs in the reverse chronological order and undoing each operation from incomplete epochs in turn.

An epoch is deemed *stable* when all events in the epoch and in all previous epochs made it to the persistent store. The *stabilization coordinator* uses a scalable tree communication protocol to detect epoch stabilization and prune FOLs.

5.4. Resource Manager

Manages any "ownable" part of the system

A large number of superficially incompatible components are treated by Mero as manifestations of the same concept of *resource*. A resource is something that one system component can request, hold locally (cache), use, or grant to other parts of the system. Generic Resource Manager (RM) code manages the details of resource location, tracking, accounting and conflict detection. Examples of resources include:

- File identifiers
- Quotas
- Cache memory
- Locks on file extents
- Locks on metadata
- Device space grants
- Network and storage bandwidth allocations
- Configuration parameters
- Object data layouts

Representing system functionality as a collection of resources results in a significant economy of concepts and implementation effort. Improvements to central RM functionality are immediately applicable to each resource type.

From the RM's point of view, Mero is a set of nodes which caches resources, uses them locally and reintegrates changes between caches by exchanging updates grouped into transactions.

Figure 6 shows a typical RM use case. To speed metadata operations, clients perform metadata operations, such as object creation, locally in their caches without exchanging messages with mdservice for each operation. To assign unique non-conflicting file identifiers (FIDs) to objects, a client acquires ownership of an extent of FIDs. Such an extent is treated as a resource.

Initially, the entire FID namespace is owned by the (redundant) FID allocation service. When an mdservice is initialized, it acquires a large extent of FIDs and later, sublets sub-extents of this extent to the clients as they connect to the service. In this case, certain resource ownership records are persistent (on the FID allocation service and mdservices) and others are volatile (on the clients).

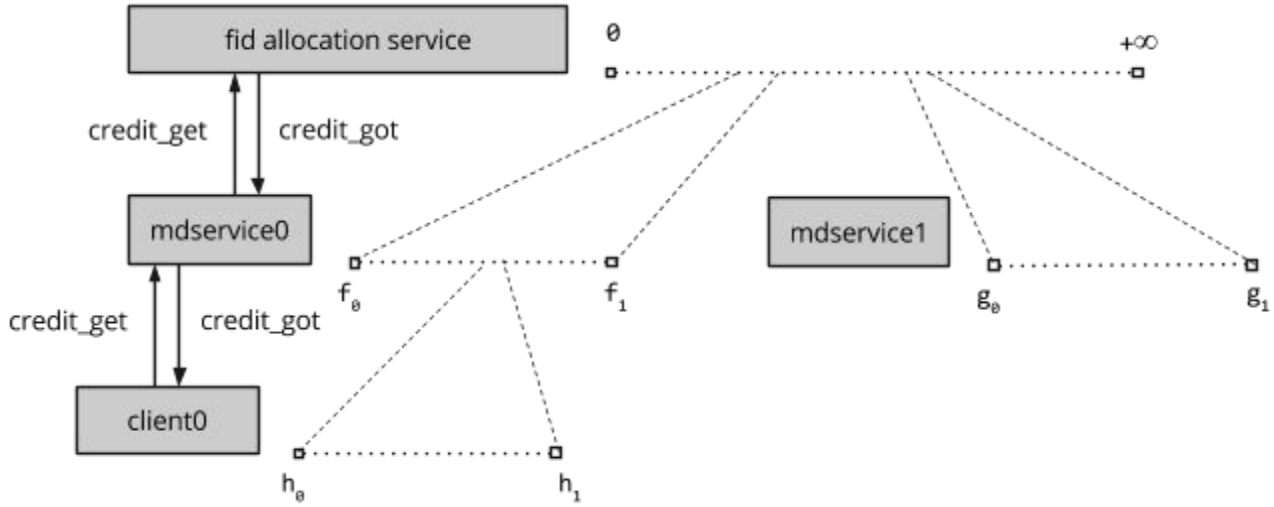


Figure 6: Use of RM for FID allocation

Figure 6 illustrates that the same representation can be applied to file extents or user quotas, etc. to provide scalable and efficient resource control without involving centralized authorities.

5.5. Layout Manager

Provides a flexible mapping of object data to a persistent location

The basic underlying structure of Mero is a distributed caching object store, where referenced objects have no predetermined location but may instead be migrated, striped, replicated or cached. Mero's Layout Manager (LM) is unlike any other currently-available storage system, most of which use precalculated (Ceph's CRUSH, Swift), or allocated-on-demand (Lustre) object layouts. Instead, Mero provides a flexible, hierarchical description to map object subcomponents to physical locations. This mapping allows for compact formulaic expressions, like CRUSH, as well as data transformations, such as erasure coding, deduplication, encryption and compression. This layout also describes data redundancy models, like simple replication or SNS.

Layouts can also describe aggregations of other layouts in a recursive manner, allowing for arbitrarily rich complexity. For example, an object layout may be described by an SNS redundancy model, except for the beginning of the file which is stored in a RAID1 manner for

faster, frequent access. Rich, hierarchical layouts are required for the NBA feature.

Layouts are treated as resources and allocated to objects by the RM. For example, a set of layouts in a parity de-clustered group may be allocated to a user for local assignment, so the user process does not have to check with a central authority for contention on the storage location.

5.6. Server Network Striping

Scalable and fault-tolerant object layout based on erasure codes

At the heart of Mero's I/O subsystem lies a *parity de-clustered* I/O layout algorithm called Server Network Striping (SNS), which is designed to meet these goals:

- Configurable resilience
  An object stored according to a parity de-clustered layout is guaranteed to be accessible, via data reconstruction, if preconfigured storage devices or servers fail (up to a certain number of failures). Resilience varies from "none" (in which case the parity de-clustered layout degenerates to RAID0) to N-way mirroring with a spectrum of intermediate possibilities (for example, RAID6 8+2).

- Storage hardware flexibility
  Neither multi-ported storage devices nor RAID hardware are necessary. Moreover, there are no restrictions on the number of devices in a storage pool. If any device or server fails, data remains accessible from other locations across the network.

- Distributed spare space
  To optimally utilize available hard drive arms, spare space is used to reconstruct lost devices scattered across all devices.

- Erasure codes
  Parity de-clustering redundancy is based on industry standard Reed-Solomon erasure codes, for which a host of well-known optimizations exist.

- Fast repair
  Because data and spare space are uniformly distributed across all devices, typical repair bottlenecks for RAID arrays are eliminated and repair speed is proportional to the total throughput of the entire pool. As a result, the larger a pool, the faster it repairs. SNS repair is performed by a distributed Loom state machine.

Figure 8 shows the flow of data between instances (replicas) of a Loom state machine

executing SNS repair.

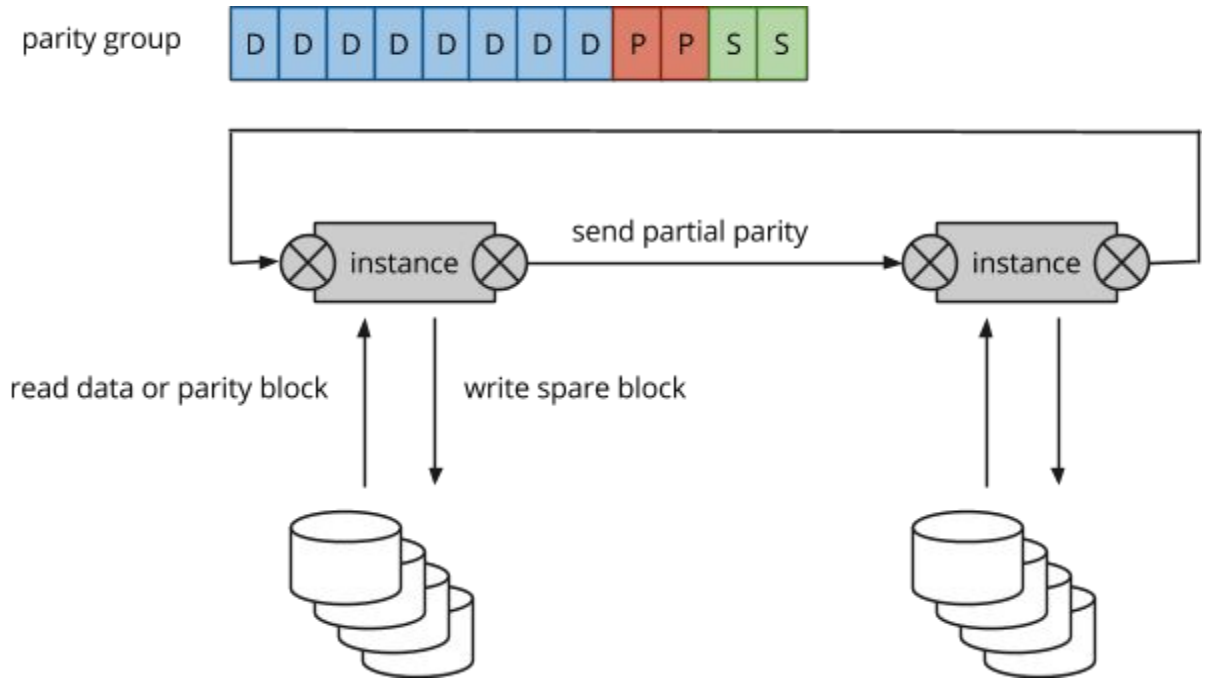


Figure 8: Loom performing an SNS repair

- Limited I/O degradation
  A longstanding problem with RAID is repairing a failed device requires all remaining devices to be read in their entirety. As a result, either the device repair proceeds at full speed, consuming 100% of available device bandwidth (and halting concurrent user I/O) or the repair operation is throttled, which makes it slower and increases the window of vulnerability when a secondary failure cannot be tolerated. Parity de-clustering decouples parity group size (the number of data and parity blocks in a stripe) from pool width, using a combinatorial layout based on pseudo-random permutations. Using this approach, only a limited fraction of each surviving device needs to be read to reconstruct the failed device data, effectively capping the total fraction of bandwidth used by repair; user I/O proceeds unimpeded. At the same time, since all devices are utilized for the repair operation, repair speed is proportional to the entire pool width.

Figure 9 shows how the parity de-clustered layout works. Columns correspond to *targets*, which can be thought of as component objects, stored on separate storage devices on separate nodes in a pool.

The upper half of the figure is an un-permuted *tile*, representing a contiguous region of object and covered back to back by the *parity groups* each consisting of data blocks (five, in this

example) and parity blocks (two, in this example). Obviously, no two blocks of the same group belong to the same target.

The parity de-clustered layout permutes columns in the tile (see the lower half of the figure). Still no two blocks of a parity group belong to the same target. A different permutation is selected by a deterministic algorithm for each tile in the object. As a result, parity blocks are uniformly scattered asymptotically across all targets, achieving SNS goals described above.

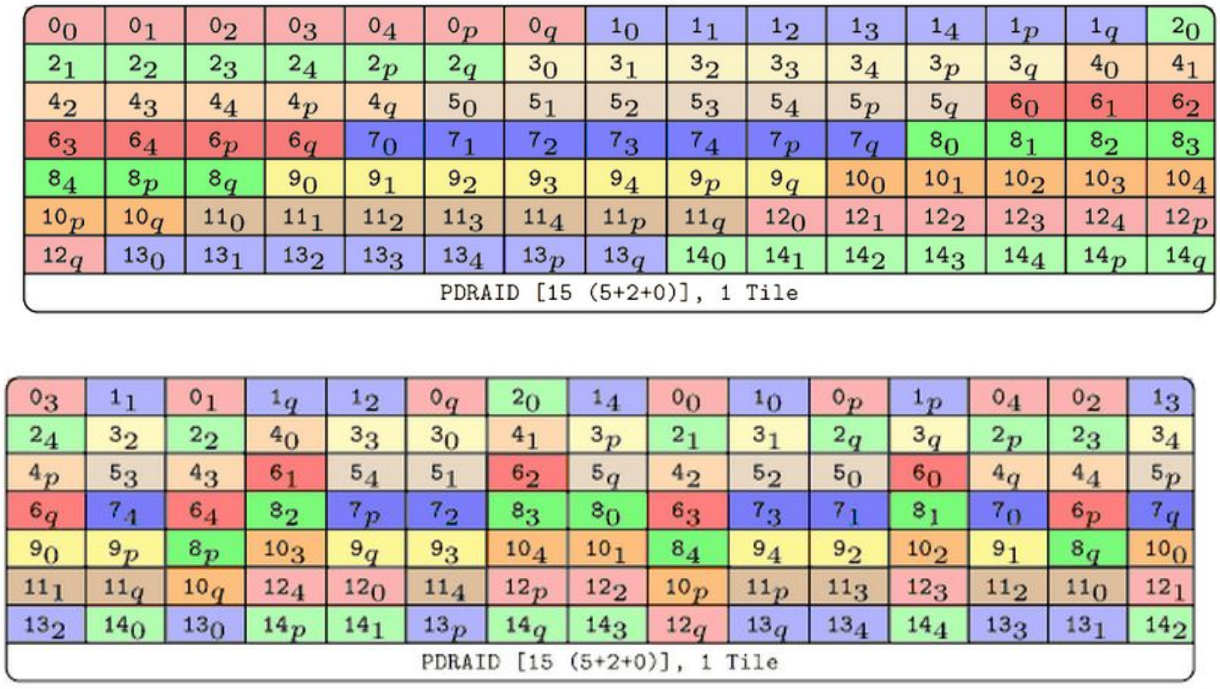


Figure 9: Un-permuted (top) and permuted (bottom) tiles used by the parity de-clustered layout

5.7. Non-Blocking Availability

Provides sustained I/O in the face of storage device failure

While SNS and the Loom provide the ability to reconstruct data on the fly to immediately compensate for failed devices during read operations, a different mechanism is required to ensure durable and reliable write operations in the event of device failure.

Ideally, applications writing data should not be affected by the availability (or non-availability) of a particular storage device in the underlying storage system, as long as another device is available. Mero's architecture includes the Non-Blocking Availability (NBA) feature which, after the failure of a storage device, enables data to be written continuously without any degradation

in I/O performance. When a device failure is detected, NBA causes an immediate change in the layout of an object, resulting in the data stream being redirected to a new set of storage devices. Older parts of the object retain their original layout, so the entire object has a layout consisting of two different sub-layouts. As usual for Mero, the elements are generic, so a sub-layout can, in turn, have its own sub-sub-layouts which enable the deft handling of arbitrary numbers of device failures.

The sequence of events involved in NBA is depicted in Figures 10a–10d.

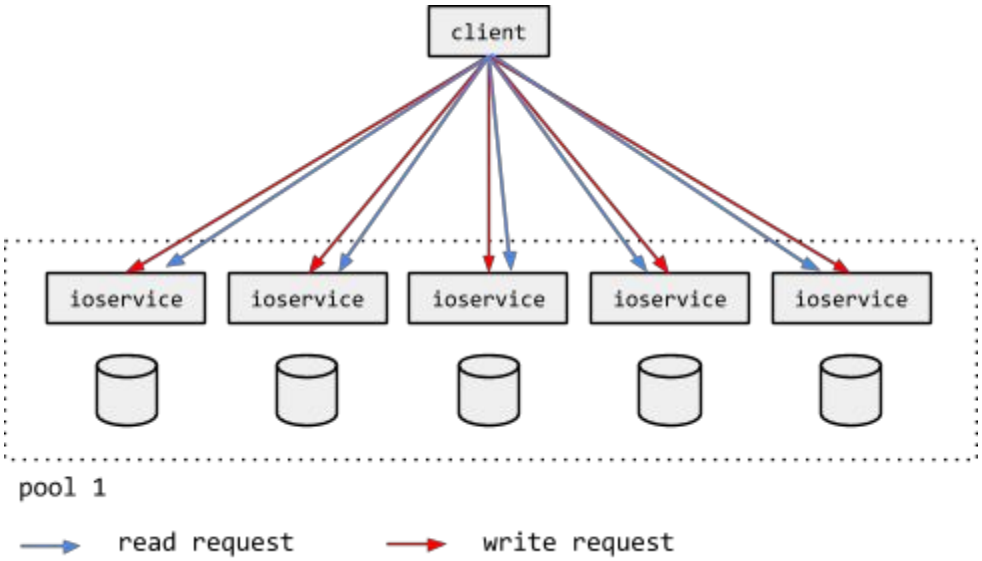


Figure 10a: NBA: The client is doing normal read and write operations to objects stored in Pool 1.

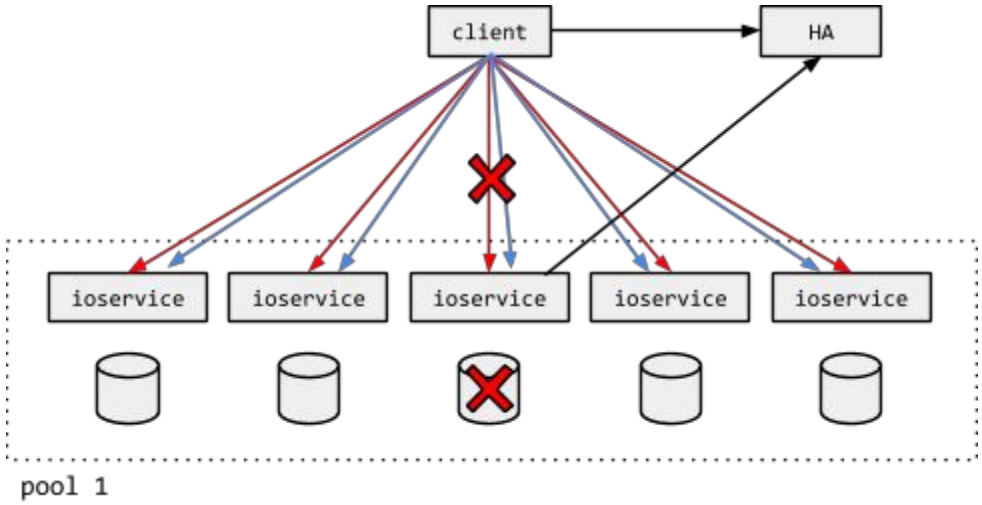


Figure 10b: NBA: One of the storage devices or services in Pool 1 fails. The client experiences

errors or timeouts. The client (and possibly the service) notify HA about the failure.

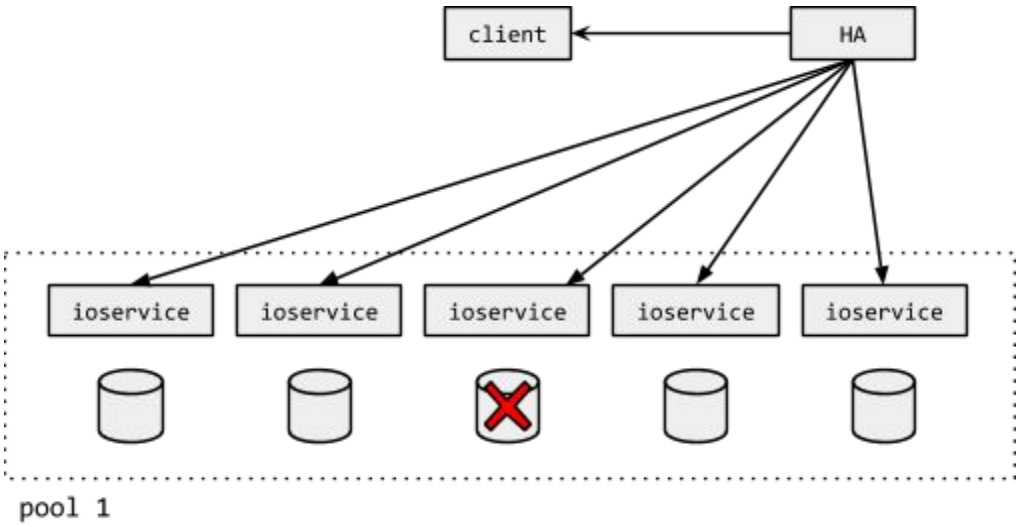


Figure 10c: NBA: HA notifies the pool that repair should start and notifies the client that Pool 1 is in a degraded mode.

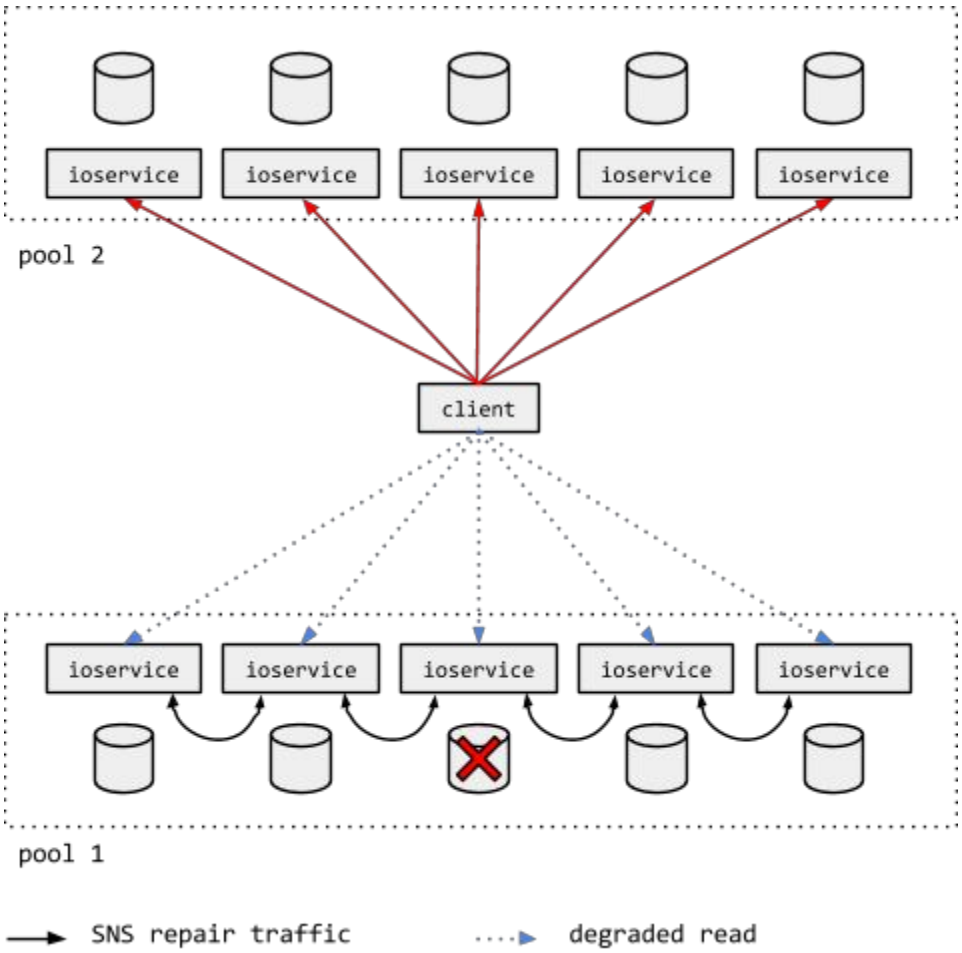


Figure 10d: NBA: Pool 1 starts the repair process. The client uses the degraded mode for read operations and uses a different pool (Pool 2 in this example) for write operations.

5.8. Analytics and Diagnostic Database

Systematically collects and organizes performance measurement data

When Mero executes an operation on behalf of a user or system, it accumulates auxiliary information about the operation's execution, for example, time elapsed for network communication, time elapsed for storage I/O, lock contention, queue depth, cache hit or miss, etc. These performance measurements, called *data-points*, are collected in the records of the Analytics and Diagnostics Database (ADDB). An ADDB record contains information that can be used to analyze the execution of operations and detect exceptional conditions. Other ADDB records contain system data-points, such as free memory or provide notifications for exceptions like device failures or timeouts.

Diskless Mero instances (clients) forward ADDB records across the network. Instances with

persistent storage store ADDB records, in a documented format, on the storage device. Similarly to the FOL, ADDB records can be mined using a MapReduce-style mechanism, either online (as the records are produced) or offline (from storage). While this mechanism can collect millions of records per second, it is designed to be "lossy" to avoid negatively impacting data performance.

Figure 4 shows subcomponents of the ADDB subsystem.

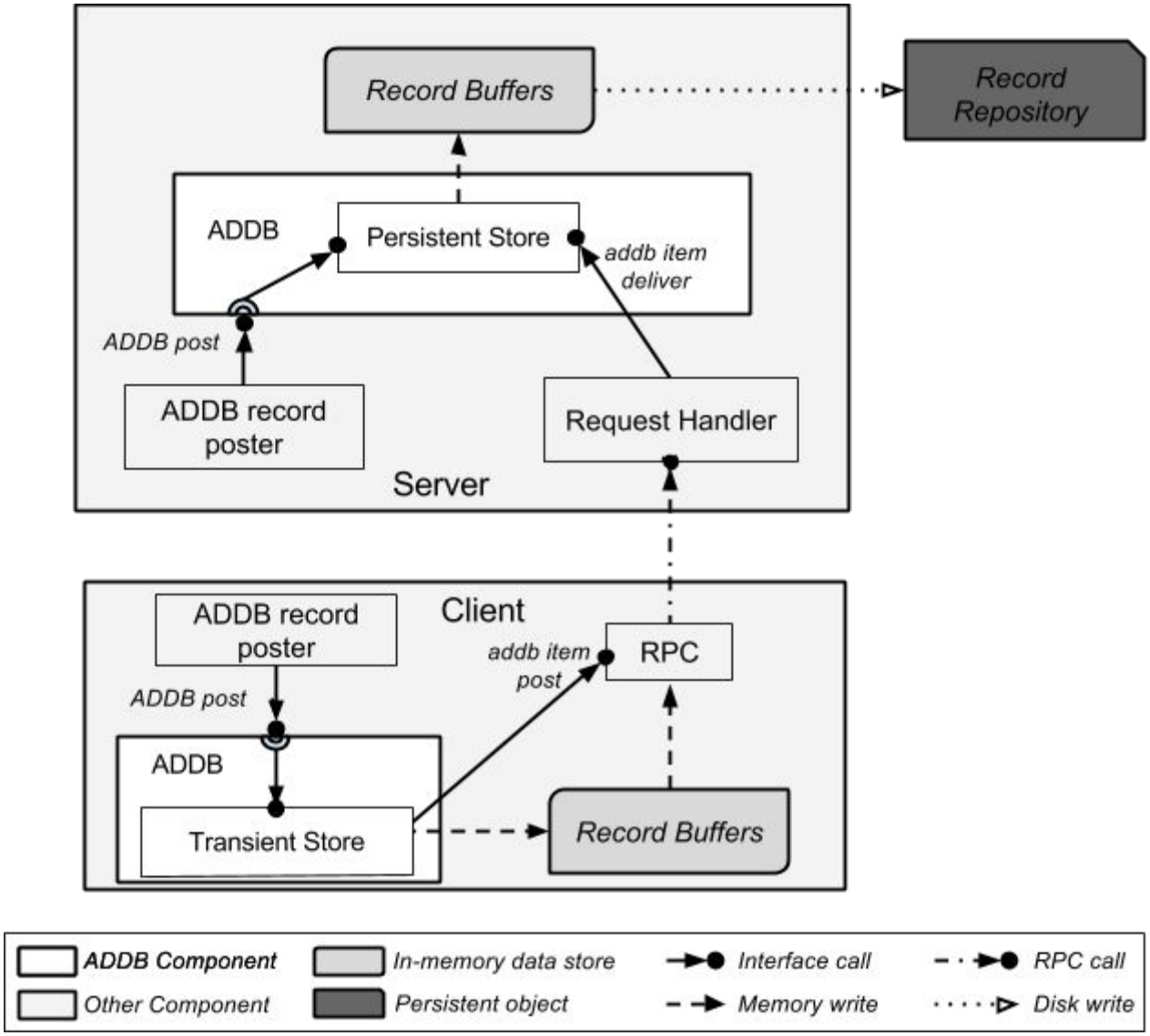


Figure 4: ADDB components and control flow

Mero code is systematically instrumented to produce ADDB records. Each record contains *context information*, which identifies system activity to which the record pertains. Mapping the

context of multiple related records enables, for example, records produced by multiple Mero services while executing the same read request, to be identified.

This *structured data* approach enables the collection of more efficient, clear and complete records of system behavior (ADDB is always on) as compared to bulk statistics or traditional unstructured system logs, which are either not sufficiently detailed or must be turned off to reduce overhead.

ADDB records are intended to be used for error reporting, debugging, health monitoring, performance measurements and visualizations of data flows and system hotspots.

In a non-trivial storage stack, multiple layers should be able to report events of interest that can be correlated with events in different layers and on different nodes. Data mining consolidates these data (on a live basis and post-factum) for system analysis. Valuable first-find-fix diagnostics should be available along with theoretical system modeling to answer "what-if" questions and simulate application behavior in proposed configurations.

Figure 5 shows the possible uses of ADDB.

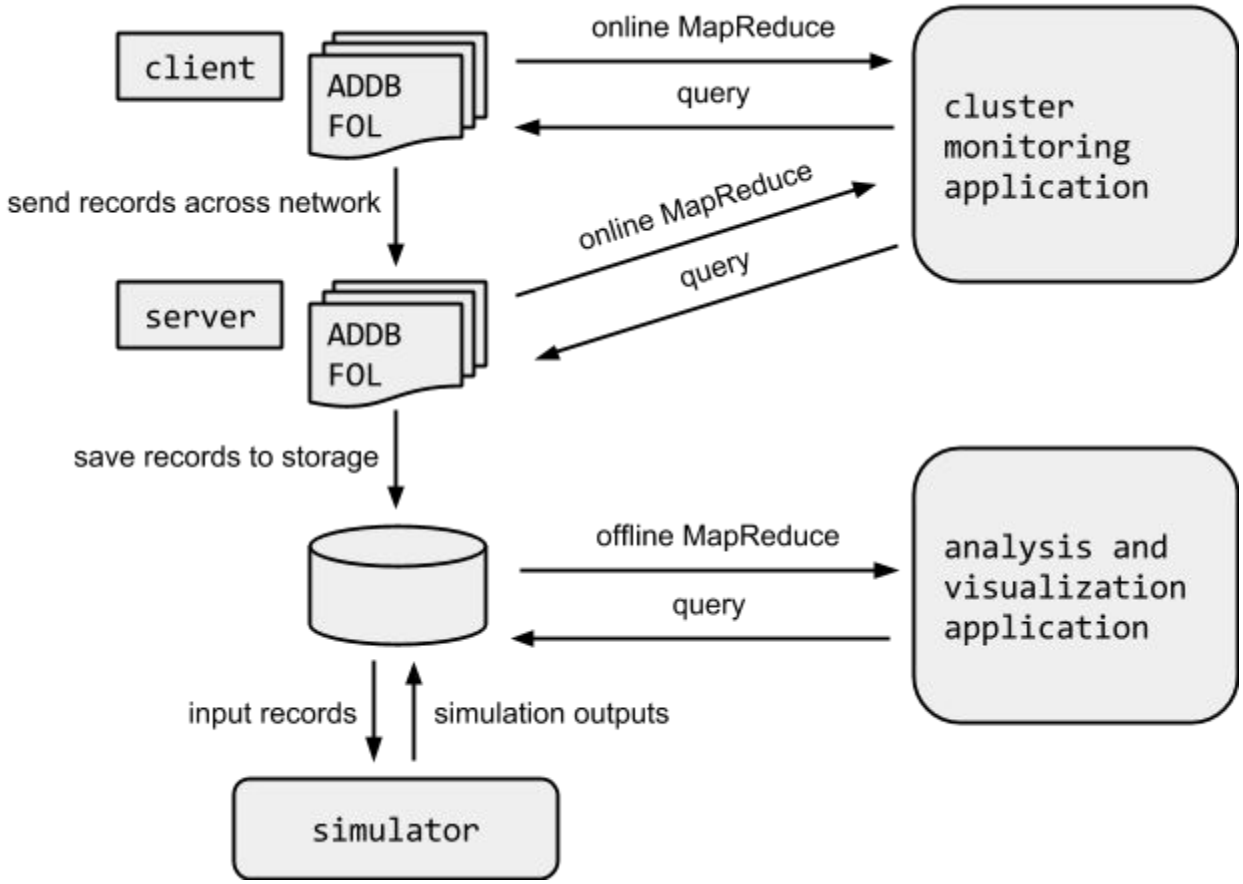


Figure 5: Interaction of ADDB with monitoring, visualisation and simulation applications

5.8.1. Monitoring

The simplest ADDB use case is cluster monitoring. If an ADDB record, produced by a Mero node, matches a filter, specified by a cluster monitoring application, the record is forwarded to the application (subject to batching and aggregation on the intermediate nodes). The application, which can also be an FDMI subscriber, uses incoming records to display statistics of interest: operation rate or throughput, memory usage, processor utilization, etc. all of which can be calculated cluster-wide, per server, per client or per job.

By manipulating ADDB and FDMI filters, a monitoring application can *query* Mero system about more detailed aspects of its behavior. For example, it is possible to request records to get answers to the questions like the following:

- Which processes of a particular MPI job do random IO?
- What clients read a given range of a given file?
- Does any device in a pool have abnormal service latency?

Internally, Mero uses the same ADDB mechanism to send information about exceptional conditions to the HA subsystem.

5.8.2. Analysis

All ADDB records generated are eventually stored on persistent storage and kept there subject to an administrative purge policy.

An analysis and visualization application mines stored records to analyze system behavior post factum, much like the monitoring application is doing online. The application makes it possible to analyze past events, inspect details of a job execution post mortem and provides invaluable help to find, fix and track.

5.8.3. Modeling

ADDB data is sufficiently detailed to allow the simulation of file system behavior.

A simulator uses a subset of stored ADDB records as the input for the simulation. ADDB records, among other things, provide accurate traces of all calls made by applications to Mero. The simulator "re-executes" these traced calls. Additionally, other subsets of ADDB can be used to drive the simulation, for example, to provide information about characteristics of storage devices, network connections or failures.

The most basic use of such incoming traces to is calibrate the simulator until the output produced, in the form of ADDB records, matches actual system behavior.

Once the simulator is properly calibrated, it can be used to test what-if scenarios on a system:

- What would be the completion time of a given job if more memory were installed on a server?
- How would a job (with a known ADDB trace) be affected by concurrent execution of a background backup job (with a known ADDB trace)?
- How would a particular HA algorithm scale to a system 10x larger than an existing one?

With a simulator, simulations can be run without access to actual customer data or customer hardware, providing the ability to remotely debug or model the impact of changes to a system.

5.9. File Operation Machine

Non-blocking, threadless state machine for executing operations

Standard server architecture employs a thread-per-request model, in which a thread from a pool takes a request from the incoming queue and runs it to completion.

Figure 12 depicts a typical thread-per-request server in which a thread from the thread pool takes a request from the incoming queue and executes the request to completion, going through computational phases intermixed with wait periods.

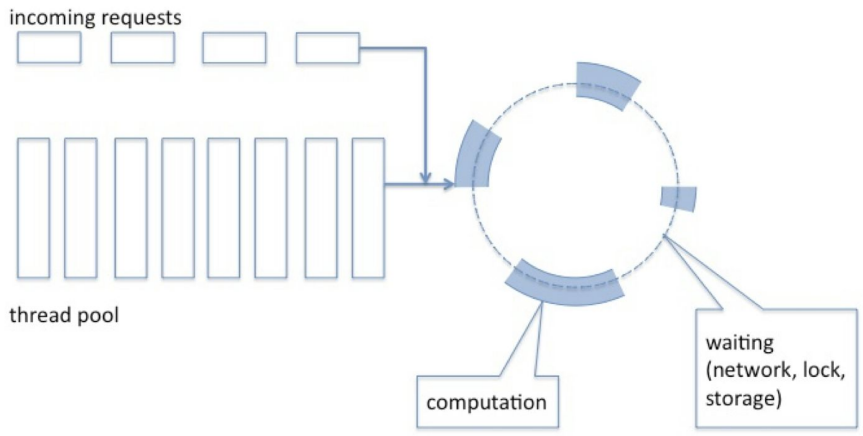


Figure 12: Typical thread-per-request server

This model has well-known scalability problems:

- Threads often block waiting for resources and events. To fully utilize CPUs, an ever growing number of threads is needed. For example, high-end Lustre servers run thousands of threads.

- Threads are expensive; they consume user and kernel resources. CPU caches and TLB must be flushed, and stacks must be saved off and restored.

- Hordes of threads are difficult to control: their scheduling is unpredictable, resulting in increased contention, overhead and reduced temporal locality of reference.

- Thread schedulers are optimized for very different workloads.

To avoid these problems, Mero uses a *threadless* execution model. A server is partitioned in *localities*. Each locality is, effectively, a small server containing a single processor core. A File Operation Machine (FOM) is associated with a locality and parked on one of the locality queues, either the run queue or the wait queue. A locality has a *single thread* that continuously takes a FOM from the run queue, executes a non-blocking state transition for the FOM (that is, some part of operation execution that does not require waiting) and places the FOM into the wait queue when it pauses for a pending event.

Figure 13 depicts the threadless Mero server design. A locality (CPU core) runs a single handler thread that takes a ready FOM from the ready queue, executes a non-blocking state transaction and puts the FOM into the wait queue.

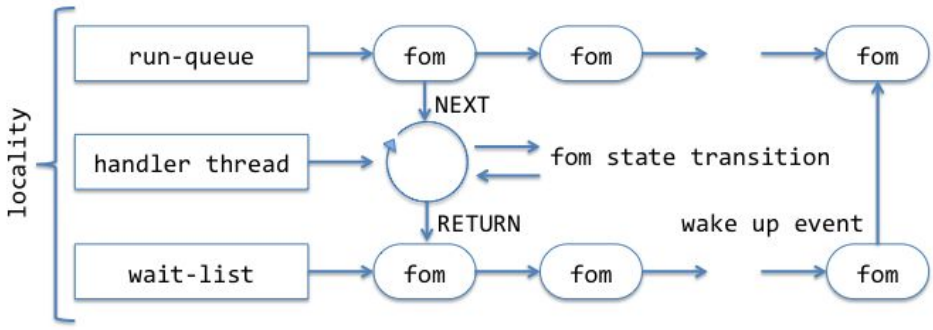


Figure 13: Mero's threadless server design using FOM

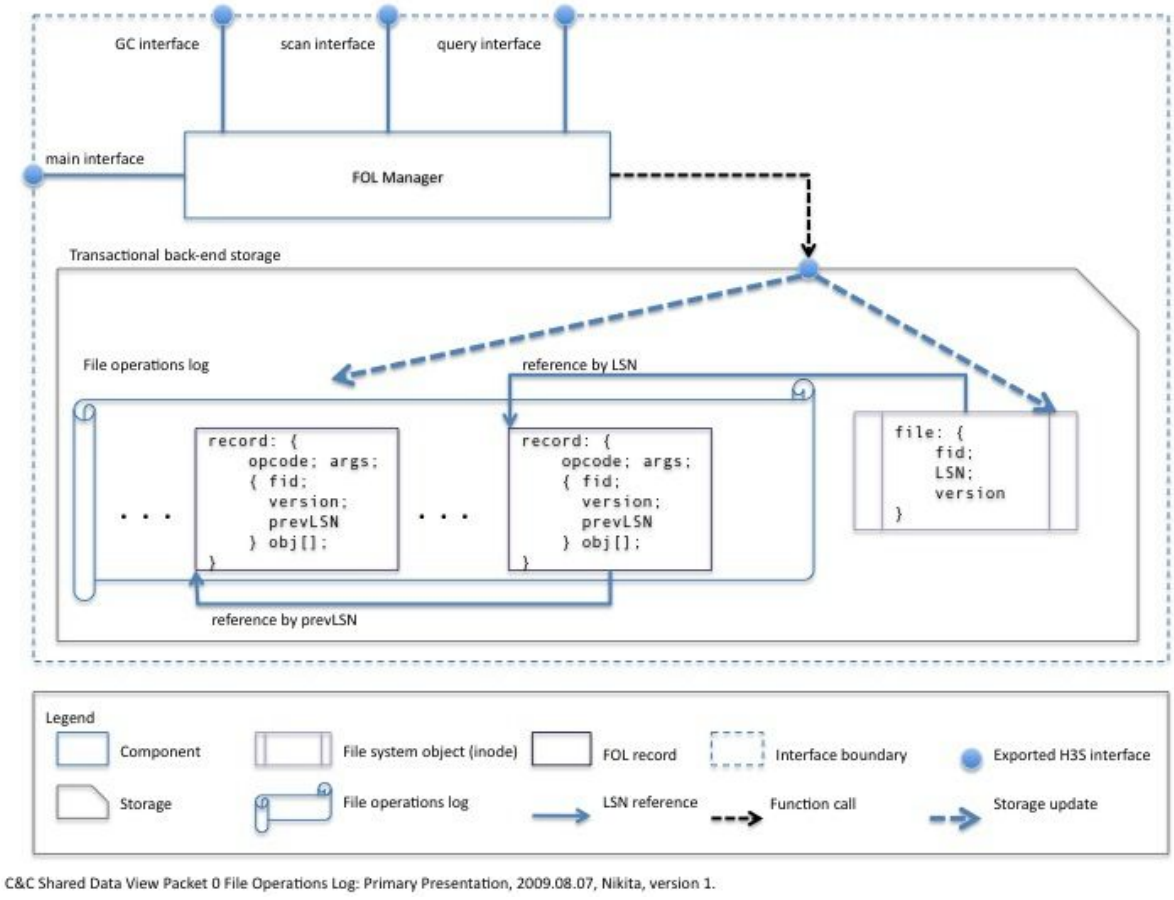
Effectively, a locality is a FOM scheduler that is aware of the semantics of its executed operations, enabling it to make scheduling decisions that optimize resource utilization.

5.10. File Operation Log

Records all storage operations, including parameters and dependencies, in a Mero instance

The File Operation Log (FOL) is an ordered sequence of records that precisely describe all storage operations, together with dependencies, executed by Mero. To avoid scalability problems, each Mero instance maintains its own FOL, where all operations executed by the instance are recorded, rather than maintain the FOL in a single centralized location. Each FOL record contains all parameters of the executed operation, together with links to earlier operations upon which the storage operation depends.

Figure 2 shows components in the FOL subsystem.



C&C Shared Data View Packet 0 File Operations Log: Primary Presentation, 2009.08.07, Nikita, version 1.

Figure 2: FOL interfaces and components

FOL is a critical part of the Mero architecture and is used for variety of purposes:

- KVS uses the FOL to recover consistency by replaying operations after a node failure
- DTM uses the FOL to determine transaction boundaries and restore system consistency in the event of a failure
- FOL detects previously executed operations that are duplicated by the network
- FOL reintegrates cached changes from a client to a server or upstream from a proxy server
- FDMI uses the FOL as the source of recorded operations

5.11. High Availability

Provides a global, consistent view of cluster status

Most storage systems are designed based on the assumption that failures are rare events, such that they are not accounted for at all or, when they occur, the system enters a time-consuming recovery/reconstruction phase. Typically, failure detection is left to secondary mechanisms or a reliance on timeouts which must be extended to cover transient hardware interrupts. By contrast, Mero implements an embedded high availability (HA) subsystem to optimize problem detection and deliver the fastest response, based on the assumption that failures are a standard part of system behavior and not unusual occurrences.

5.11.1. Quorum-Based Global State

With increasing system scale, the frequency of component failures increases. An HA system used to detect and recover from failure must scale with component count. Fundamentally, a storage system designed for exascale requires an HA system that is also designed for exascale. Mero includes a built-in HA subcomponent designed to gather and incorporate the state of every node into an authoritative view of cluster status. A rule-based recovery coordinator responds to state changes and broadcasts changes to the cluster. Communication is hierarchical for scalability and redundant for reliability. Because the global state is continuously updated, it is not necessary to wait for large timeouts to decide component disposition; a consensus view of the failure is rapidly established.

Figure 11 illustrates communication between Mero processes and the HA subsystem. Through ADDB, a Mero process notifies its local HA about exceptional situations, like communication timeouts and storage I/O errors. HA analyzes the pattern of exceptions, detects failures and announces them back to Mero.

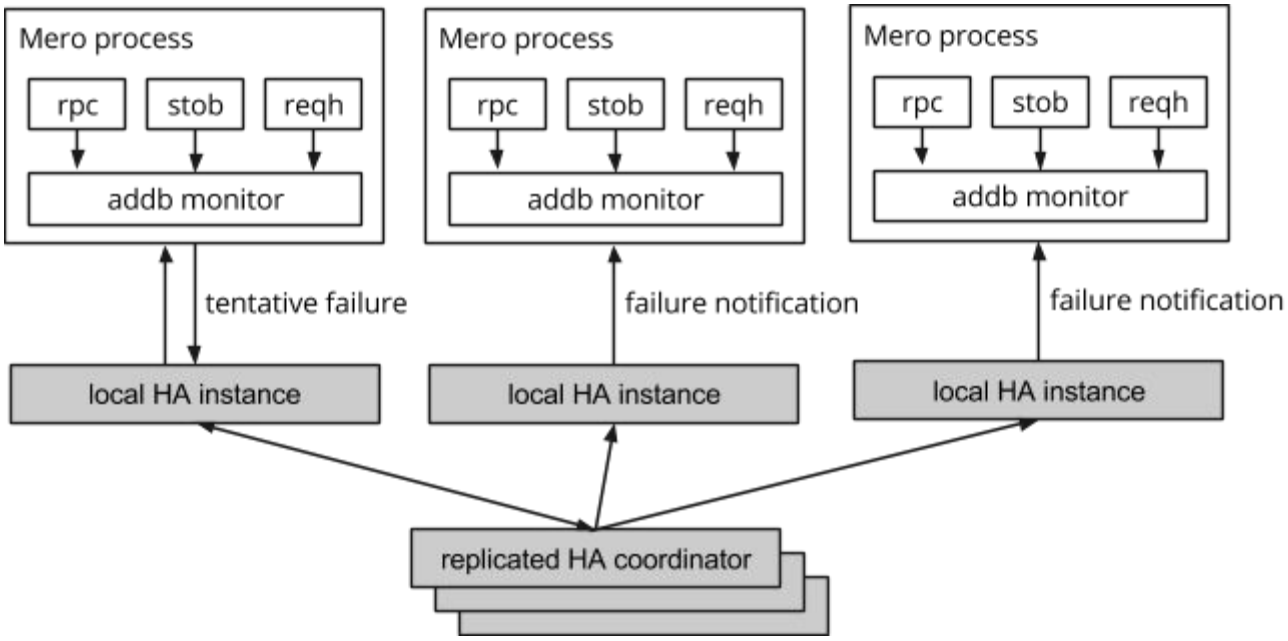


Figure 11: Data flow between Mero processes and the HA subsystem

5.11.2. Continuous Availability

Unlike traditional "failover/recovery" models (integrated into many storage systems) which enter a special "recovery mode" and postpone normal processing while resolving a particular failure, Mero simply redirects data to alternate locations—via NBA (for writing) or via SNS (for reconstruction on the fly) as needed. This approach avoids most of the timeout problems that plague large-scale systems.

5.11.3. Continuous Consistency

To avoid taking the system offline for a traditional file system consistency check (fsck), an FDMI plugin can be developed for Mero to continuously check the distributed consistency. Optimally, such a check would never be needed, but it is prudent to allow for the possibility and ensure that the design of the Mero architecture and FDMI enables this option with minimal impact on data availability.

5.12. File Data Manipulation Interface

Provides a horizontally extensible API for storage features (e.g. compliance, retention, search, audit, security)

The File Data Manipulation Interface (FDMI) is a publish-subscribe interface designed to receive records about operations executed in the Mero system or inject file operation requests. The

FDMI is designed for use with external plugins (storage applications).

FDMI allows users to subscribe to a subset of FOL records that are specified by a filter. Mero uses a MapReduce-style algorithm to continuously monitor new records added to the FOL across all Mero instances, collect records that match the filter, batch the filtered records and forward them to subscribed users. In other words, FDMI allows users to "listen" to interesting events that occur in the Mero system.

Figure 3 shows the flow of FOL records from the nodes where they are generated to FDMI subscribers.

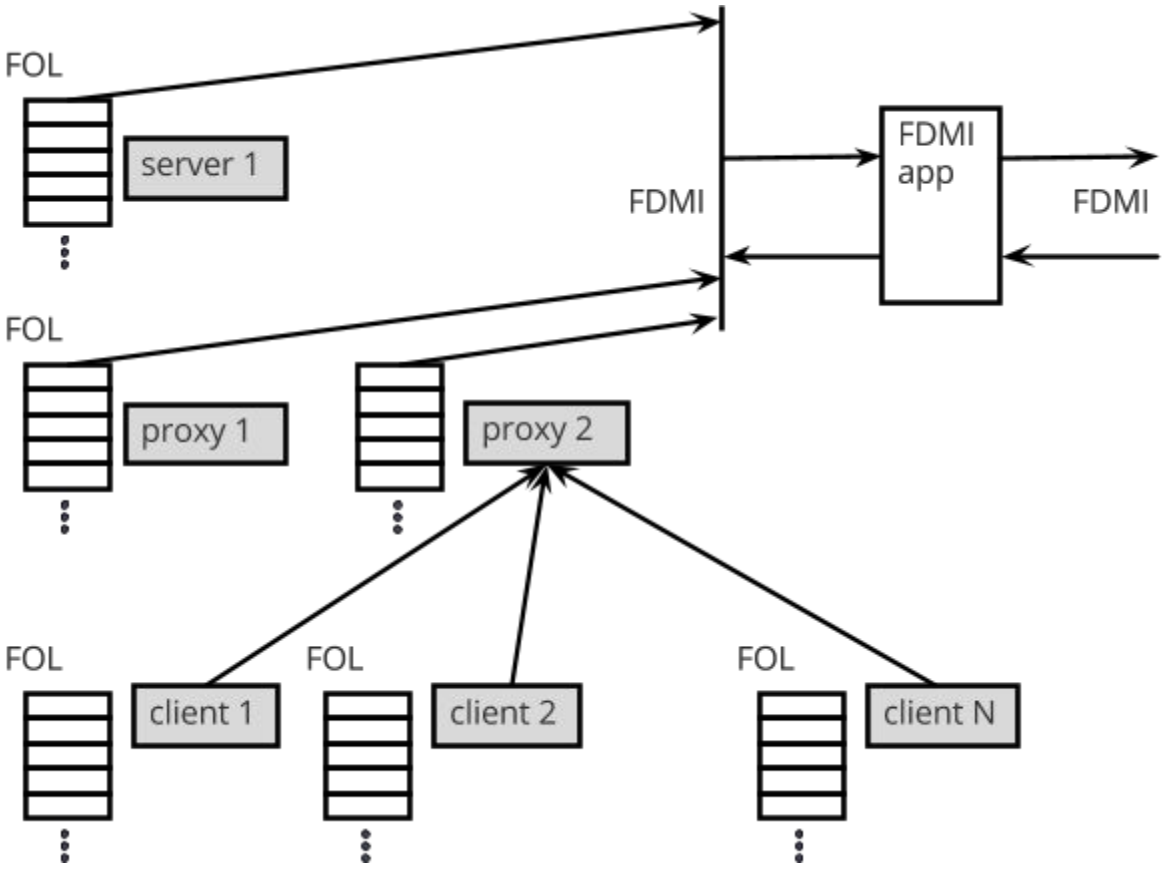


Figure 3: Data flow of FDMI's MapReduce-style algorithm

As a simplified example, consider an FDMI-based application running on a dedicated server, connected to two Mero object stores (source and target) and subscribed to all operations in the source. When the FDMI user receives an operation from the source, it re-executes it in the target. This approach provides a simple and scalable replicator that does not require scanning of the source for updates and changes. If failures occur, the DTM guarantees that the source and that target remain consistent.

As a key element of Mero's scalability architecture, FDMI allows Mero's core to remain simple, clear and focused, while providing a framework to support future features and capabilities, in which they act as tightly integrated components. Core Mero will not have to change to add new features, such as later-developed functionality to support HSM; instead, an FDMI plugin will be written. The ability to utilize additional nodes to run FDMI plugins is important for horizontal scalability and compares favorably with the traditional architectures, where new features have to be added to existing servers.

5.12.1. FDMI vs. POSIX

As a *storage-level* interface to data, FDMI overcomes the inherent limitations of POSIX, by enabling plugins to directly manipulate the storage infrastructure. This direct interaction allows storage applications to implement changelogs, mirrors, snapshots, indexers and other complex data management solutions.

As scale increases, ILM features (backup, migration, HSM, replication, etc.) hit a POSIX "bottleneck". POSIX is designed for per-file lookups via string-based hierarchical namespace traversals; this is an extremely inefficient method to operate on an entire file system. POSIX forces inherently storage-centric utilities like HSM, replication, backup, and search to use the same stat/open/read/write/close interface to data as applications use, causing generally poor performance, scalability, recovery, and administration. In other words, the FDMI approach to publishing batched records to subscribed users is singularly effective because no form of scanning works at exascale.

Another problem that emerges at scale is storage application behavior in the face of failure, either in the storage system it operates upon or in the application itself. To avoid arbitrary inconsistencies, scalable solutions to ILM requirements must be tightly integrated with the storage system implementation, in particular by relying on logging changes and transactional mechanisms capable of resuming interrupted data management processes.

On a higher level, there is a striking contrast between the file system and database worlds. In the latter, the domain-specific SQL language underlies most applications, making them portable (either without or with a modest effort) between various RDBMS systems. For file systems, the only common language is POSIX and portable backup or HSM applications are unheard of (except for POSIX-based solutions, such as tar and rsync, which are not scalable and generally incorrect). The development of FDMI creates the possibility of an entirely new market for portable storage applications that are designed and developed independently from storage systems. To this end, FDMI is portable and can be implemented on non-Mero systems.

5.13. Clovis

Mero interface exported for object store access

With Mero, we have defined a rich, transactional storage API, known as Clovis, that can be used directly by user applications and can also be layered with traditional interfaces such as POSIX and RESTful APIs, much as libRados is the interface upon which the CephFS (POSIX), RadosGW (S3), and RBD (block device) interfaces are built.

Figure 14 shows Clovis in a larger system context.

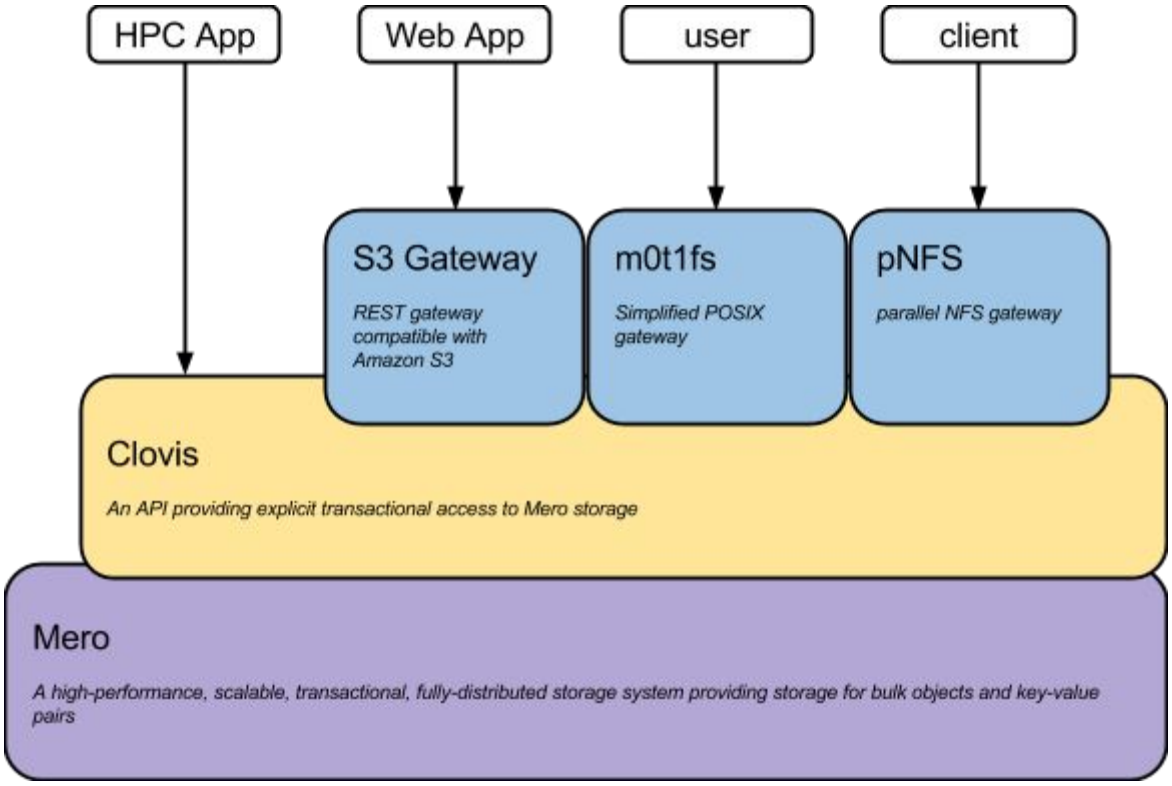


Figure 14: Clovis contextual diagram

Details on the Clovis interface are available below.

5.14. Mero's Data Path

To help concretize the abstract conceptions of our new storage architecture, this section provides a detailed description of Mero's data path (auxiliary metadata operations are ignored).

While Mero has no distinct client and server code, we refer to a Mero "client" as the *role* of a local provider of data to an application. Ultimately, this data may reside on any other Mero instance (effectively, running in a *server* role). Keep in mind that this "server" may actually be

local as well or may be a proxy for another remote service.

An application on a cluster compute node writes data to a new Mero object through an API based on the Clovis interface, for example, an object-based zero-copy API. The local Mero instance (the Mero “client”) assigns a layout to the object; the layout is, for example, a parameterized expression of an SNS parity de-clustered RAID layout with 8+2 parity. The application passes a description of the data buffers to the client, which then divides the data into striping chunks. At this point, the client may compress or encrypt the chunks. The client then calculates 2 parity chunks for every 8 data chunks, calculates chunk checksums, and the 10 data chunks are queued for network transfer to 10 different “servers” (nodes running Mero services with storage). The next set of 10 data chunks are sent to 10 other servers, and so on, eventually covering the entire storage pool. Each target node has a separate queue; RPCs in each queue are coalesced, if possible, into larger units to be transferred over the network. As target buffers come available, RDMA network transfers are initiated by the receiver.

Once a data chunk arrives on the “server”, the checksum may be verified, optionally, and metadata including the checksum, a chunk identifier, and other attributes are stored in the server’s KVS. The data is stored as a component object, which may eventually grow with additional data chunks. The data is stored directly on a block device (with no intervening file system) at a location recorded in the KVS. Modified data is written to new locations on the block device's copy-on-write (COW) style to allow for recovery. The entire write operation for a parity group of stripe chunks is atomic; component objects and KVS changes are guaranteed to all be written successfully using “redo” semantics if sufficient information is available after a node failure or all will be undone if the operation cannot be reconstructed.

5.15. Meeting the Challenge

Table 1 summarizes how the Mero architecture reflects the quality attributes of an optimal storage system and meets the challenge of overcoming the inherent limitations of existing systems, as outlined above (in 2. Motivation).

Quality Attribute	Mero Feature
Horizontal scalability	• SNS provides a data layout in which a large number of devices can be used to achieve desired throughput and redundancy. • FDMI allows services to be offloaded to additional standalone nodes, eliminating server-side bottlenecks. • KVS scales the size of metadata storage with the number of servers, removing metadata server bottlenecks. • RM distributes sub-pools of resources throughout the cluster, reducing single-point contention and inter-node communication. • Client-server integration facilitates server proxies for hierarchical deployments.

Vertical scalability	• FOM server design avoids context switching overhead, reduces memory requirements, and optimizes multi-core NUMA CPU. • KVS can utilize high-speed, low-latency non-volatile storage. • Simplified layering reduces latencies and interface bottlenecks.
Availability, Reliability and Fault Tolerance	• DTM hides transient failures. • SNS repairs persistent storage failures. • HA provides a consensus view of failures. • NBA allows continuous writing in the face of device failure. • Layouts allow flexible redundancy models.
Observability	• ADDB provides information about system behavior that can be used for analysis, debugging and simulation.
Quality of Implementation	• Abstraction unification, rigorous development process, and systematic testing improve code quality. • Simplified layering reduces server stack complexity. • Client-server integration reduces code size.
Horizontal Extensibility	• Clovis interface provide a rich transactional API upon which to build new interfaces and meet existing standards. • FDMI is used to create new storage plugins to extend system capabilities.
Vertical Extensibility	• Clovis interface can provide distributed transaction control from within applications • Layout capabilities can be extended with new patterns or transformations. • Hardware awareness and elimination of abstraction layers allows for new efficiencies.
Deployment Architectures	• SNS eliminates requirements for drive multi-porting and RAID hardware. • Client-server integration enables local resource utilization and simplifies code.
Data Locality	• RM provides for ubiquitous, distributed resource caches. • FOM allows for CPU- and cache-line affinity.

Table 1: Mero architecture's features by quality attribute

6. Applications

Mero technology is intended to be broadly applicable and can be used effectively in a wide variety of distributed storage environments.

6.1. Application Areas

Today, three major areas exist with large-scale distributed storage requirements, broadly

classified as High-Performance Computing, Analytics, and Cloud applications. Mero has the potential to deliver significant advantages in each of these areas.

6.1.1. High-Performance Computing

High-performance computing (HPC) applications impose the most demanding requirements on storage systems. Such systems encompass the largest and fastest file systems in the world, the most complex distributed file locking tasks and the highest component failure rates, owing to the large number of system components.

Typically, HPC applications involve very large numbers of client nodes (tens of thousands) working communally on single, computationally-intensive simulations or analyses. A backing storage system stores input data, output data and, typically, expanded temporary (working) data sets. Extreme requirements for storage capacity, data I/O rates and metadata rates (object creation, deletion, naming, etc.) are implied by the large numbers of client nodes. Fully parallel storage systems are capable of sustained speeds of more than 1 TB/sec. However, many use cases experience data transfer rates far below optimal speeds, due to inefficiencies in the I/O patterns requested by the HPC application and the overhead required to resolve contention for storage resources among nodes. Contention and synchronization problems are not easily resolved using POSIX semantics, so a consortium of HPC users developed a separate message-passing library standard (MPI) to help with these tasks.

Mero addresses the challenges of HPC in a number of ways. Most directly, Mero implements a much richer transactional API (Clovis) that allows application developers to specify transaction boundaries and ensure their data is coherent to the required degree, without excessively burdening the system with locking overhead. For example, an application that periodically checkpoints independent data segments only needs to ensure that transaction boundaries are placed around each "epoch" and not around each write system call.

Clovis also provides for guided interfaces, enabling hints to be passed to the storage system about the intended use of data, such as short-term paging data, that does not need to be transmitted remotely or stored redundantly or the precious results of a long-running compute job that is intended to be archived redundantly and will not be used again in the short-term. The storage system can use these hints strategically - to decide *which* data should be cached and *where* it should be located, *when* data should be transferred across the network, and *how* and *where* data should be stored.

Beyond providing improved interfaces, Mero also offers a number of solutions to address present-day HPC problems. Mero’s RM is a highly flexible component that can grant resources in a hierarchical manner and provide much greater scalability than the traditional client-server HPC model in which every client talks to the same set of servers to control resource contention. RM's hierarchical design allows any Mero instance to act as a proxy for the others; a proxy can

become locally authoritative for cached data and provide a simple path to utilize local flash devices for temporarily buffering I/O from checkpoint-restart applications.

Other architectural changes within Mero can improve the operation of HPC applications. Within Mero, communication paths have been organized into parallel streams between arbitrary endpoints, eliminating deadlocks and increasing utilization. Thread-based, context-switching data handling flows have been replaced with non-blocking state machines organized around individual cores—positioning Mero to take advantage of more powerful CPUs as core counts increase. Memory-based, fully distributed metadata improves metadata performance both vertically and horizontally. Mero's comprehensive emphasis on locality, caching and event-driven processing can result in lower power requirements.

Finally, significant improvements in availability are provided by Mero's robust feature set. SNS provides storage redundancy to protect against drive failure in a storage-efficient manner. The Loom can reconstruct data on the fly in the face of temporary or permanent loss of access to drives. NBA transparently redirects written data to available storage devices. An integrated, scalable HA mechanism monitors the health of every node in the system, providing quick and deterministic responses to component failures. Complete undo/redo recovery within the DTM maintains system consistency and coherency.

6.1.2. Analytics

In this paper, we refer to "Analytics" as the analysis and manipulation of large data sets involving unstructured data. Whereas HPC compute jobs typically operate on data with a known structure, Analytics operate on problems without predefined data models.

As a general rule, Analytics applications can operate on segmentable data sets, in which data segments are treated as independent units. MapReduce-style computations do most of their work using multiple nodes that operate on locally stored data and the (mostly) columnar databases built for Analytics are also designed to operate in parallel on smaller data segments. Mero can be deployed to provide local storage, but it offers the additional advantage of enabling quick and efficient access to remote data. This functionality should allow Hadoop, for example, to operate more quickly (when remote data is needed) even if the storage redundancy / location model is fully duplicated in Mero.

More importantly, Mero does not require triplication to ensure data availability, but instead uses highly efficient networked erasure codes—enabling a Mero-based Analytics installation to offer substantial savings in storage costs compared to other existing systems like Hadoop or HBase. For example, Mero can provide trebled redundant access to data for a 20% RAID 8+2 overhead versus a 300% triplication overhead.

Ultimately, Analytics applications must become location-agnostic, as HPC file systems are

today. As the number of concurrent component failures grows in an increasingly large system, the greater the likelihood of losing all copies of an object (with ever expanding copy redundancy an untenable option); this reality implies we will reach an eventual limit to Analytics scalability based on today's storage system designs. To continue to scale beyond these boundaries, Analytics applications must divorce themselves from intimate knowledge of where data lives and must instead rely on a very fast, distributed storage system to provide access to data as required by the application. It should be the responsibility of the storage system, not the application, to ensure the reliability, integrity and availability of data and to manage data layout and location. If this architectural approach is taken, the flexibility of the compute system is increased, storage efficiency is increased and application writers are freed to concentrate on their analyses, rather than the backing storage system.

In terms of Analytics applications, the Mero infrastructure can provide direct and tangible benefits. For example, frequently, the value of Analytics applications lies in indexing of large-scale data sets, not in storage of the data sets themselves. Exploring the metadata about the data provides the value, tagging interesting events and tracking patterns of interest. Mero does not predefine the metadata associated with objects; arbitrary, user-defined metadata can be stored and accessed extremely quickly in the RAM-based KVS. Should the intended amount of metadata exceed the capacity of the KVS, metadata containers can be constructed and treated as bulk data, shipped and stored quickly and efficiently. Storage layouts for metadata containers would be optimized for random-access patterns, for example, mirrored and stored preferentially on SSDs.

Additionally, with FDMI, application-specific data indexing can be implemented as a file system primitive, allowing dual-use modes, such as HPC analyses with "automatic" Analytics processing in the background. The DTM can ensure that data, the FDMI indexing application and any key-value generated metadata are added atomically and are recoverable.

6.1.3. Cloud

Generally, cloud-based storage for web applications or custom storage solutions places a premium on cost per megabyte, wide-area access and simple interfaces, with data reliability and availability assumed. Because Mero implements an extremely scalable object storage system, it is well-suited for storing large amounts of data remotely, that is accessible to large numbers of network-connected computers. Mero's flexible erasure-coded layouts provide for arbitrary redundancy choices, allowing per-object tuning for cost versus availability. For additional cost savings at scale, Mero uses a copy-on-write storage technique that is compatible with lower-cost shingled drives. Additionally, FDMI can be used for transactional, policy-based replication to remote sites.

Typical data I/O patterns for cloud storage involve large files, sequential I/O and static content. These types of patterns are especially efficient for Mero to use when calculating parity

information, parallelizing object stripes and transferring data over networks using RDMA techniques.

Mero can also extend the standard capabilities assumed with cloud storage. The RM and DTM allow for much stronger consistency guarantees than competitive systems. For example, Amazon's S3 provides for loose eventual consistency of data and metadata changes—a change made on one client may not be seen by other clients for some period of time. Mero could (but need not) guarantee that all nodes see the same state; the choice could be made by the cloud user.

6.2. Comparing Mero to Other Systems

Table 2 summarizes the differences (based on key attributes) between Mero and two other file systems in use today, Ceph and Lustre.

Table 2: Mero storage architecture attributes compared to Ceph and Lustre

	Ceph	**Lustre**	**Mero**
Inception date	2007	1999	2011
Design process	ad hoc	ad hoc	SEI TSP
Interface	librados: S3, POSIX, RBD block	POSIX	Clovis: CDMI, POSIX
Environment	userspace only (FUSE)	kernel only	kernel or userspace
Structure	client-server	client-server	symmetric
Processing		thread per request	non-blocking state machines
Network RAID level	RAID 10 1+2 (triplication), Erasure Coding as of v0.8 release	RAID 0	Reed-Solomon erasure coding, N+M
Availability	synchronous redundancy	recovery model, redo	continuous availability
HA	embedded Paxos	external	embedded Paxos
Metadata	dedicated MDSs	dedicated (few) MDSs	fully distributed
Monitoring / Diagnostics	unstructured logging	unstructured logging	distributed, structured analytics
Disk format	layered (btrfs)	layered (ext4, zfs)	direct
I/O	synchronous	asynchronous	asynchronous

Designed scale	PBs	10s of PBs	EBs
Features	snapshots, shared key security, open source	open source, HSM, full POSIX	SNS, distributed metadata
Status	"reasonably stable"	production	in development

6.2.1. Mero vs. Lustre

Because Mero is intended to address the most demanding storage requirements in the future, it is illustrative to compare the design of Mero against the design of Lustre, the incumbent top-end distributed filesystem. This section presents a detailed comparison of some of the key differences between Mero and Lustre. (Bear in mind that the design of Mero was heavily influenced by what we perceived as shortcomings in Lustre, so it should not be surprising that this comparison tends to show Mero at an advantage.)

6.2.1.1. Code Complexity

Since its inception in 1999, Lustre has seen significant functional improvements released on a consistent basis. Although it is a mature product, Lustre development has not wanted, as one might expect of an established product. Instead, an increase in performance and scaling requirements and the demand for new features have provided steady work for the Lustre team and contributed to the consistent growth in Lustre's Lines of Code (LOC) and code/file changes, as shown in Figures 15 and 16.

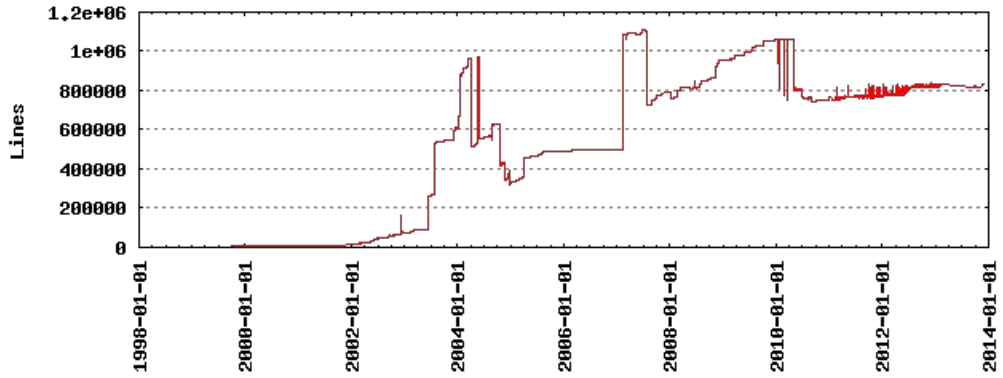


Figure 15: Lustre LOC over time (1999-2014)

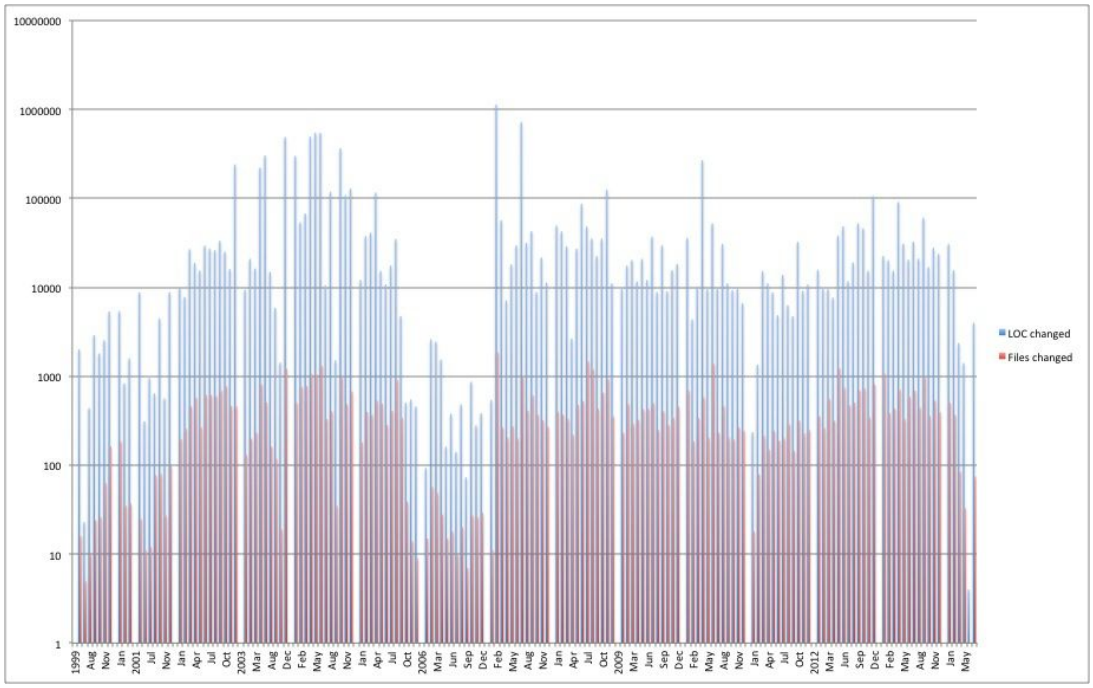


Figure 16: Changes in Lustre over time (LOC and files)

The increasing size of Lustre's code base and continuous code churn have resulted in an extremely complex and brittle system that very few people understand. Lustre's code complexity makes it susceptible to changes by subdomain experts that may end up negatively impacting other parts of the code.

The authors of this paper aim to avoid a similar outcome with the Mero storage architecture by implementing a two-fold approach—using simple core components and implementing FDMI. Mero's core components are straightforward and widely used, improving their reliability. Additional Mero features are intended to be implemented via FDMI, preventing changes in the code core and ensuring that it remains stable. This strategy, to add additional architectural capabilities in a modular, conservative manner is the genesis of the requirement for "horizontal extensibility".

6.2.1.2. Quality

Lustre is the highest performant, most scalable file system in the world that is used by over 60% of TOP100 sites,[8] however, it is far from being bug-free. Lustre's authors (along with many others) have spent many years developing Lustre, adding features and debugging problems along the way. It is fair to say, that at each new increasingly large Lustre deployment, new problems have been revealed, requiring significant debugging efforts. Due to the size and complexity of Lustre's code base, it has often been the case that implementing a fix for one

[8] *OpenSFS*. Retrieved from http://opensfs.org/lustre/

issue caused problems elsewhere in the product.

This is one of the major promises of Mero—the opportunity to learn from the long experience of Lustre development and start from a clean slate, in terms of the architecture and the implementation. Mero's architectural focus on simple, core mechanisms reflects our experience, as well as the “walled garden” approach of the FDMI, the choice of user space servers, and the structured data ADDB monitoring mechanisms discussed in this paper. Regarding implementation, a strict quality process (SEI's TSP) was chosen with the intention that the quality of the code was measurable and predictable. (A similar effort to impose a robust software development process failed for Lustre because it was imposed very late in the product lifecycle, long after the horse "left the barn”.) In addition to multiple design and code review phases, Mero's development process includes internal unit tests for each component, exercised automatically at every check-in event, and live assertion-based testing of parameter preconditions and postconditions for each function.

6.2.1.3. Threads Per Request

Thread-based request handling models process a single request from a queue, moving it to completion while waiting for prerequisites to be met: obtaining a lock, waiting on a related request or obtaining notification of persistence or acknowledgement of a message. This model requires large numbers of threads to be instantiated to ensure that CPUs have sufficient non-sleeping threads, which consumes significant memory resources. Context switches are common, where stacks need to be rebuilt and cache lines flushed. Related requests may vie for contended resources and deadlocks may occur in unanticipated use cases.

Mero avoids these problems by using a state-machine processing model. A small pool of processing threads is dedicated to each CPU core. Each thread pulls a request that is ready to transition, that is, the request has already met any required preconditions. The request is moved from state to state until it cannot be progressed, at which point it is dropped and the processing thread picks up another pending request. Threads are always busy, reducing context switches and keeping cache lines hot. Inter-thread deadlocks due to reverse dependencies are eliminated. Less memory is consumed, less data is moved, cores are used more efficiently and less power is required.

6.2.1.4. HA Integration

HA can be decomposed into two functions: detection of problems and remediation actions. Lustre detects failures by using various timeouts; significant effort is spent automatically adjusting timeouts to measured system conditions using features like Adaptive Timeouts. A ping-based system between servers and service clients is used to detect dead clients and servers. Problematically, as system size scales, ping traffic increases and becomes a significant issue. A more difficult, inherent problem is that timeouts on larger systems must become larger, due to an increased "standard deviation" of normative behavior.

By contrast, Mero employs a scalable, dedicated HA subsystem (called HAsk) that uses a quorum-based eternal group to maintain a universal, consistent view of the entire cluster. Status events are grouped and forwarded to the quorum, which acts as the central authority on node and service state. This approach eliminates non-scalable ping traffic and reduces reliance on system timeouts. Discovery of node events may use arbitrary local functions, such as hardware monitors, and forsake timeouts completely.

Lustre recovery is relegated to special-purpose, external HA subsystems like Corosync, which offer only one remediative action: power down the server and restart the service on a failover node. Alternately, HAsk allows for complex rules that can result in arbitrary remediations, from changing global state to executing driver hardware resets via local processes. HAsk can also be used outside of Mero itself, for example, RAS rules could email reports of surpassed temperature thresholds.

6.2.1.5. Locks vs. Resources

Lustre uses an efficient Distributed Lock Manager (LDLM) to protect cached resources, for example to ensure “ownership” of a particular piece of a file for read or write operations. The LDLM ensures the coherency of caches on the distributed clients. This approach works reasonably well for smaller systems, as server-managed locks are lightweight and special-case optimizations can improve performance. However, as system scale increases so does lock contention, and fine-grained locks are required in more and more areas. For highly contended resources, lock traffic can mire Lustre servers.

Mero seeks to avoid this situation by re-envisioning locks as a case of general resource management. As described above (in 3.9 Resource Manager), usage rights can be handled in a resource hierarchy. For example, a file’s extents may be owned fully by one client, perhaps the node that created the file. A second node that wants to write to the file borrows the second half of the file extents. In turn, a third node borrows part of the file from the second node. In this way, lock management becomes decentralized and scalable. Local resource control allows asynchronous operations on the resource subtree.

6.2.1.6. Debugging

While Lustre operates in a very large scale distributed asynchronous environment, a cohesive plan for distributed data collection and analysis was never designed. It is not surprising that debugging Lustre is a complex, time-consuming and interactive affair that requires the collection of large numbers of server and client system logs, memory dumps, requests to reproduce the problem and "experimental patches".

Mero’s ADDB, on the other hand, aims to address this by incorporating an internal distributed data collection mechanism, designed to scale via map/reduce style collection points. The data

collected is structured (i.e. in a machine-readable format) and is intended to provide information sufficient to simulate system behavior for “first incident” debugging.

6.2.1.7. User Space vs. Kernel Space

Lustre server components are written as kernel modules for a number of reasons, the most significant being that it must interact with the kernel filesystem drivers. Since Mero incorporates it’s own disk format, it can run outside the Linux kernel. This allows developers to use comprehensive userspace debugging tools (e.g. symbolic debugging, automatic crashdumps), and results in more complete information and quicker analysis. Mero is also not dependent on kernel version changes, thereby reducing code churn and customization requirements, and enables important features like rapid kernel security updates.

Perhaps most importantly, in contrast to the total system shutdown due to a kernel panic, userspace operation enables storage servers to continue to operate in the event of a storage system bug, allowing auxiliary software stacks to continue to function. In this situation, other Mero services may continue to run uninterrupted on the same server; Mero’s high-availability processes can more easily identify a particular failed service; restarting a service is a simple respawn instead of a complex powercycle procedure; and associated restart times drop from minutes to seconds.

6.2.1.8. Layouts

Layouts in both Lustre and Mero refer to the description of the location of a file’s component objects. Lustre has a design that allows a selection of layout types, but as of Lustre 2.5 only the “RAID 0” (striped) layout has ever been used. (There are high-level designs to allow for RAID 1 (mirrored) layouts.) Because Mero implements its own erasure coding, layouts are significantly more complex. Looking upon this as an opportunity rather than a problem, Mero implements a pluggable, hierarchical layout schema allowing for complex representations of data location; see Layout Manager for detail.

6.2.1.9. Failover Model

With Lustre, an external HA mechanism must detect (using ad-hoc methods) a server “failure”, and execute a “failover” which involves cutting power to the failed node, and restarting the failed service on a failover partner, which must have dual-path access to the original server’s storage devices. The new server then enters a special "recovery mode", postponing normal processing while the Lustre clients reconnect to the new server, then replay in-progress requests. This process can be very time-consuming for large systems (an hour is not unheard of).

Mero does not employ this “failover/recovery” model - Mero simply redirects data to (or from) alternate locations—via NBA (for writing) or via SNS (for reconstruction on the fly) as needed. This approach avoids most of the timeout problems that plague large-scale systems. Additionally, because access to the original storage devices is not needed, there are no

requirements to dual-port the storage, restart/reassemble RAID arrays, or replay backing filesystem journals. The Hask HA subsystem makes a global determination that a server or device is unavailable, and all clients redirect based on this consensus decision.

7. References

Mero has been under development for several years and a number of resources have been gathered and written during that time. If access is restricted to any of the references listed below, please contact the document owner for permission.

[1] Mero in prose

[2] 2009-08 Paris Requirements

[3] Original—EIOW QAS (Portland, 2012.04.14)

[4] EIOW-QAW2 Worksheet - Portland

[5] Business Goals & Architecture: Colibri ClusterStor Storage Software

[6] Horizontal Scale Storage Software (H3S)

[7] Clovis Presentation

[8] Mero Technical Presentation

8. Addendum

This Addendum contains additional technical details related to the Mero architecture.

8.1. Clovis

Clovis is Mero's primary I/O interface. It is intended for use as a base upon which other, more traditional storage interfaces are built. Alternatively, high-end applications could be written directly to the Clovis interface for extreme customized efficiency. Clovis is discussed thoroughly above (in 3.16. Clovis). Clovis provides a small repertoire of abstractions.

8.1.1. Object

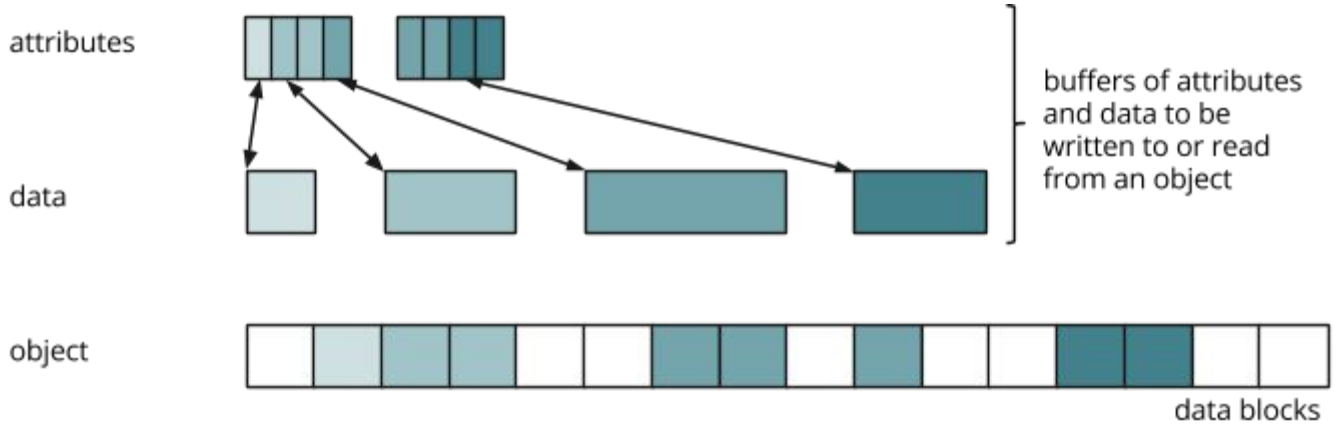
Externally, a Clovis object looks like a very large array of an array of data blocks of fixed size with indices in the [0, 2^{64}) range. Initially, every block is a *hole* that returns zeroes on read operations. Individual blocks can be read and written. All access to an object is with block granularity. I/O operations on objects are fully scatter-gather-scatter.

Internally, an object is stored in multiple component objects according to object's layout, for example, parity de-clustered layout (see 3.12 Server Network Striping).

Characteristically, Clovis objects have no usual metadata (size, modification time, attributes, etc.). If a user application wants metadata, arbitrary metadata can be implemented separately, using the Clovis's index interface; such an implementation would normally be wrapped in a reusable library.

Instead of per-object metadata, Clovis provides *block attributes*, which are associated with each block in the object. Block attributes can be used for checksums, encryption keys, hash fingerprints, etc.

Figure A-1 illustrates the object I/O operation interface. Data and block attributes are transferred between object blocks and buffers provided by the user as operation arguments. Note that number of extents accessed in the object can be different from the number of buffers, only their total size must match.




```
clovis_read(op, obj, data_buf_vec, attr_buf_vec, extent_vec)
clovis_write(op, obj, tx, data_buf_vec, attr_buf_vec, extent_vec)
clovis_alloc(op, obj, tx, extent_vec)
clovis_free(op, obj, tx, extent_vec)
```

Figure A-1: Clovis object I/O

8.1.2. Index

The Clovis index is a key-value store. Keys and values are arbitrary byte strings without internal structure. Clovis does not interpret values or keys in any way. The index comes with a standard

key-value set of operations: `lookup`, `insert`, `delete` and `next`. The `next` operation returns the key, following a given key in lexicographic order; other operations should be obvious. All operations are fully vectored and scatter-gather-scatter, that is, a single call to a Clovis index entry point can insert multiple key-value pairs with keys and values stored in multiple user provided buffers.

Internally, an index is stored in a collection of component indices, provided by the KVS (see 3.10 Key-Value Store).

Figure A-2 (below) depicts a particular operation against an index. The index is presented as an associative array.

The second row from the bottom represents the namespace of index keys. Cells in this row, from which no arrow goes down, correspond to unused keys (ones which were never used as an argument to clovis_insert()). Cells with downward arrows represent used keys. In a typical situation absolute majority of keys are unused, that is, this row is very very sparse.

The bottom row represents values associated with the keys. The downward arrow from key namespace row connect a (used) key to its value. Variable size of rectangles in the value row represents that values in an index can be of variable size. Actually, keys can be of variable size too, but that would be rather confusing to show.

Colored keys and values represent index elements affected by the particular operation.

The topmost row depicts the list of keys involved in the operation. Keys are stored in buffers. The slide shows that keys can be grouped in buffers: the first buffer contains 4 keys, the second buffer contains 2 keys.

The second from the top row shows value buffers of the operation. For clovis_insert and clovis_update, these buffers contain data to be stored as values in the index. For clovis_lookup and clovis_next, these buffers receive values read from the index. Again, the slide attempts to show that values can be grouped in buffers. Compare this with scatter-gather-scatter IO presented on the "Object IO" slide.

The shade of turquoise corresponds to a (key, value) pair used in the operation. Parts of user buffers containing key and value and the corresponding cells in index key and value rows are colored with this shade. Also, note that size of value (in the second row) matches the size of the connected value in the bottom row.

Supported operations are:

- clovis_lookup(): an array of keys (stored in top row buffers) as input, for each input locate the corresponding value in the index and store it in the corresponding part of the value buffers in the second row.

- clovis_insert(): an array of keys (stored in top row buffers) and an array or values (stored in second row buffers) as input. For each (key, value) part in the input, update corresponding index value.

- clovis_next(): ar array of keys as input. For each key, find the next used key in the index. Set the key in the input buffer to that next used key and place the value corresponding to the next used key in the value buffer.

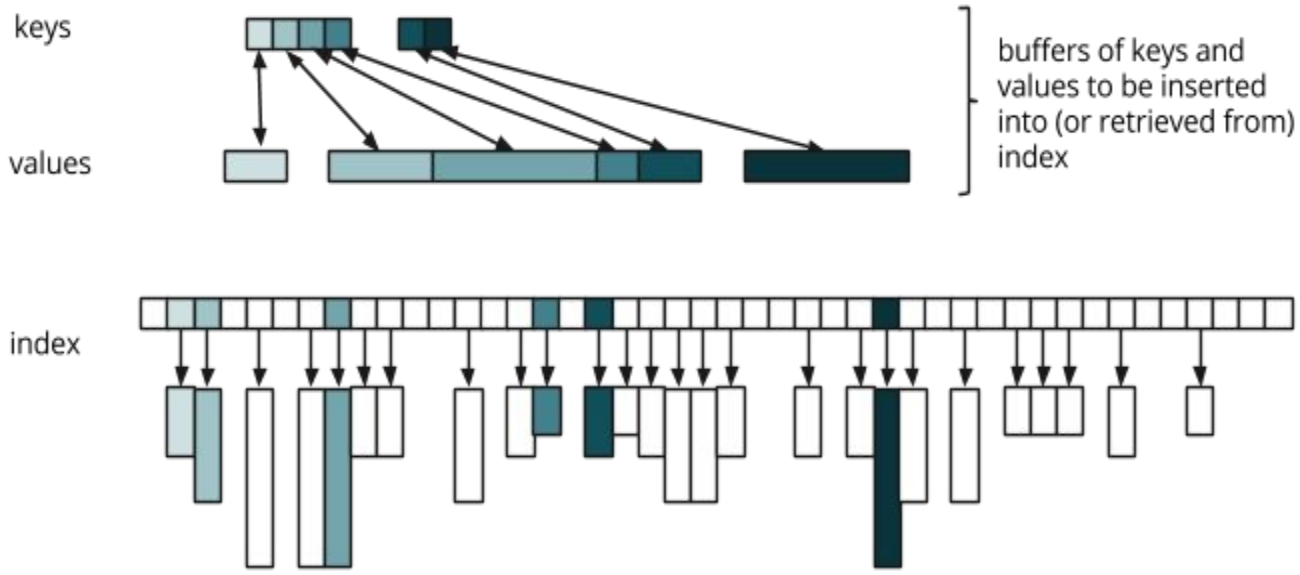


Figure A-2: Clovis index operations

8.1.3. Operation

A Clovis operation is a state machine that represents an ongoing call to Clovis. Operation state machines are needed because all Clovis calls are asynchronous. For example, a call to clovis_insert() returns immediately without waiting for a (potentially long) insertion, involving network communication and storage I/O operations on the remote end to complete. Instead, the caller provides an operation state machine, which is notified when the call execution state changes.

Operation state machines decouple Clovis concurrency from the user, so a user is free to use any threading model, including fully synchronous computation.

Clovis provides an interface to group operations together, wait for their state transitions, etc. With these interfaces, various caching and aggregation policies can be implemented on top of Clovis.

Figure A-3 shows the Clovis operation state transition diagram.

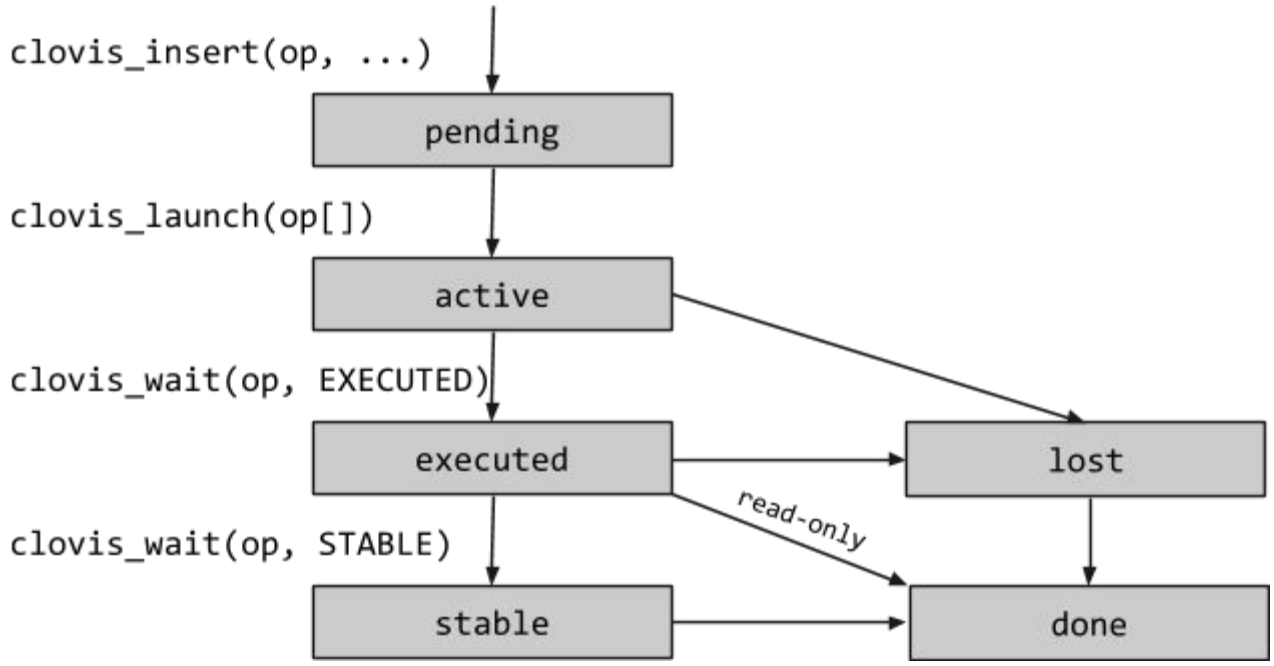


Figure A-3: Clovis operation state machine transition

8.1.4. Realm

A Clovis realm is a collection of Clovis objects and indices, together with updates to their state. A realm is a generalization of both a (distributed) container, which is an isolated collection of objects and indices, and a transaction, which is a collection of updates.

Instances of realms include:

- Read-only snapshot
- Writeable storage system clone
- Distributed transaction
- Client or proxy cache
- Distributed container

A realm is created from its parent realm, from which the former inherits initial structure of objects and indices.

Figure A-4 shows a possible realm parent-child relationship.

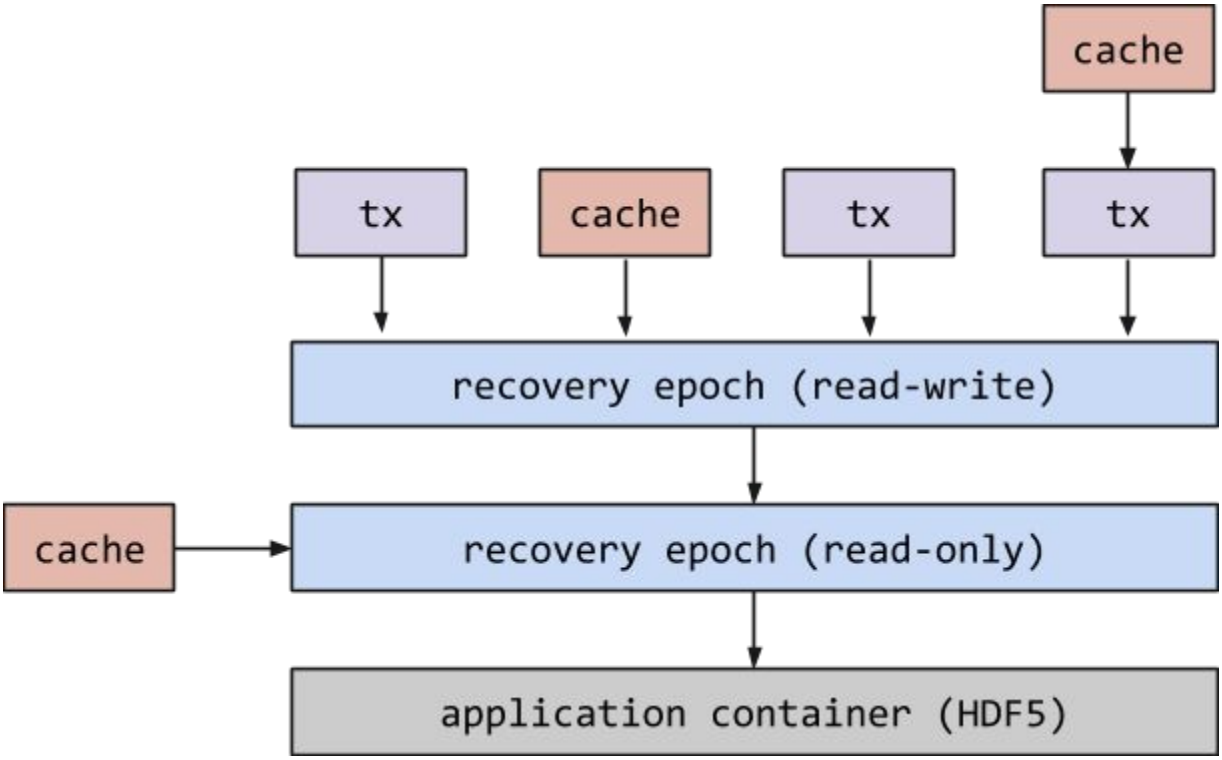


Figure A-4: Collection of Clovis realms, each pointing to its parent realm