HEWLETT PACKARD

DataMesh architecture 1.0

Chia Chao, Robert English, David Jacobson, Bart Sears,
Alexander Stepanov, and John Wilkes
Computer Systems Laboratory

HPL–92–153
December, 1992

HP Laboratories Technical Report

DataMesh is a storage architecture designed for parallel operation. This document describes the first version of the DataMesh architecture—that intended to provide a flexible, scalable, parallel block service.

The DataMesh architecture defines a set of components that can execute on one or more processing and storage nodes, together with a data model that allows for high performance, reliability, and recovery. The components are designed to balance the load across different components of the same type and to take centralized action only when the operations themselves require it. The data model attaches tags to data and expresses most low-level operations through the selection and manipulation of tags.

The primary contributions of this architecture are the description of a system structure that can scale to take advantage of high degrees of physical parallelism; and a new model for associative indexing of data that provides considerable expressive power through a simple, straightforward model that is also fairly easy to implement efficiently.

Previously published as Concurrent Systems Project technical report HPL–CSP–92–23, June 1992.

1 Introduction

This document defines the DataMesh block server architecture for implementors and potential customers. It focuses more on description than motivation; more detailed design and implementation documents will discuss the details of the algorithms and internal structures, as well as implementation and prototyping plans.

The architecture is presented here in a generally top-down manner, starting with a summary of the external model, followed by a description of the major system components. Detailed descriptions of both components and external and internal interfaces are left for a second pass.

> *Occasional sections of text (like this one) are not part of the architecture proper, but provide explanatory material or amplify our intentions. Although this document is primarily an architecture document, it is not practical, nor even desirable, to completely avoid all implementation issues. There are many places where the document mentions the anticipated usage patterns, such as the expected number of entries in some table or the intended use of an abstraction. These expected usage passages are to be considered first class parts of the document. Failure to observe them risks poor performance and possible failure due to resource exhaustion. For example, if the expected number of entries in a table is 50 to 100, it tells the designer something about how many hash buckets might be necessary, and it warns the user that inserting 5000 entries is risking severe performance problems.*

2 Overview

DataMesh is a parallel, scalable storage server, intended to be connected to one or more *host* computers that will use it to provide persistent storage. DataMesh phase 1 provides support for operations on *blocks*—fixed-size arrays of bytes—each of which has an associated *tag*, which is a small amount of additional state tightly bound to the block and used for indexing purposes.

> *In the simplest manifestations of a DataMesh the tag is just the host-specified block address, as on a regular SCSI disk. In more elaborate versions, support is provided for a number of operations that can be used to retrieve and operate on data using the tag contents as a rich indexing mechanism.*

DataMesh can provide a range of performance, reliability, and cost. For example:

- resilience to failures of different kinds (e.g., none, power fail, head crash);
- optimization choices (e.g., between small writes and large sequential reads);
- addressing models: data addresses can be *compact* (contiguous) or *sparse*. Sparse addressing is similar to virtual memory: the number of data blocks that can be addressed is much larger than the number of blocks that can be stored.

The modules that implement these choices will be described in detail elsewhere; this document focuses on the interfaces between the various modules that go to make up a DataMesh.

2.1 Physical components

> *The DataMesh architecture was developed with a particular physical manifestation in mind. We refer to this as the **research model**. While this hardware design has motivated some of the architectural decisions, the architecture can also be implemented effectively on more conventional systems, for example, ones that collapse multiple distinct components in the research model onto single computational elements.*

A research model DataMesh is physically composed of a number of *nodes* linked by a high-performance interconnection fabric. In DataMesh 1, there are three kinds of nodes: port nodes, memory nodes, and disk nodes. Each memory and disk node can belong to at most one volume. All nodes contain an embedded processor, and additional hardware that defines their function:

- *Port nodes* provide connectivity to the external world to which a DataMesh is a server. (Examples of port nodes include SCSI ports, FDDI ports, various kinds of gigabit link ports, and backplane bus ports.)

- *Memory nodes* contain no I/O component: instead, they provide more primary RAM than could conveniently be fit onto a disk or port node. Memory nodes may contain volatile and/or non-volatile RAM. They could also be used as computation nodes, if so desired.
- *Disk nodes* contain a physical disk, an associated processor, and some RAM. Ultimately, all data stored on a DataMesh ends up on one of these nodes. A typical disk node might contain 16MB RAM, and a 1–2GB disk.

The nodes are connected together by a high-performance local-area network, whose primary characteristics are low latency (< 10µs), moderately high bandwidth (≥10MB/s per node), and sufficient redundancy that partitions never occur.

2.2 Logical components

To orchestrate the physical resources, DataMesh phase 1 uses a two-level internal software structure. Figure 1 shows its major components.

Multiple *host* computers can be connected to a single DataMesh through one or more port nodes. *Clients* on the host communicate with the DataMesh through *channels*: port-to-host links. A typical DataMesh client might be a file system or database. Communications between hosts and a DataMesh are managed by a software component called a *spigot*, which handles striping, routing of individual requests across one or more channels, and authentication.

Volumes provide access to storage. The functions supported by a volume, and the interface to them, determine the *type* of a volume. There can be several volumes with the same type in a single DataMesh, each using different physical storage resources. Storage is not shared between volumes. Each volume has an identifier unique to the DataMesh in which it resides: the *volumeID*, which is typically mapped onto some externally visible identifier (e.g. a SCSI logical unit number). Volumes also provide *bindings* between the command-set used by the client across the channel and the operations available within volumes.

Finally, *banks* manage hardware resources in the DataMesh such as memory, processor cycles, and nodes. Since this document is primarily concerned with the operation of a statically configured DataMesh, bank interfaces and operations are not specified.

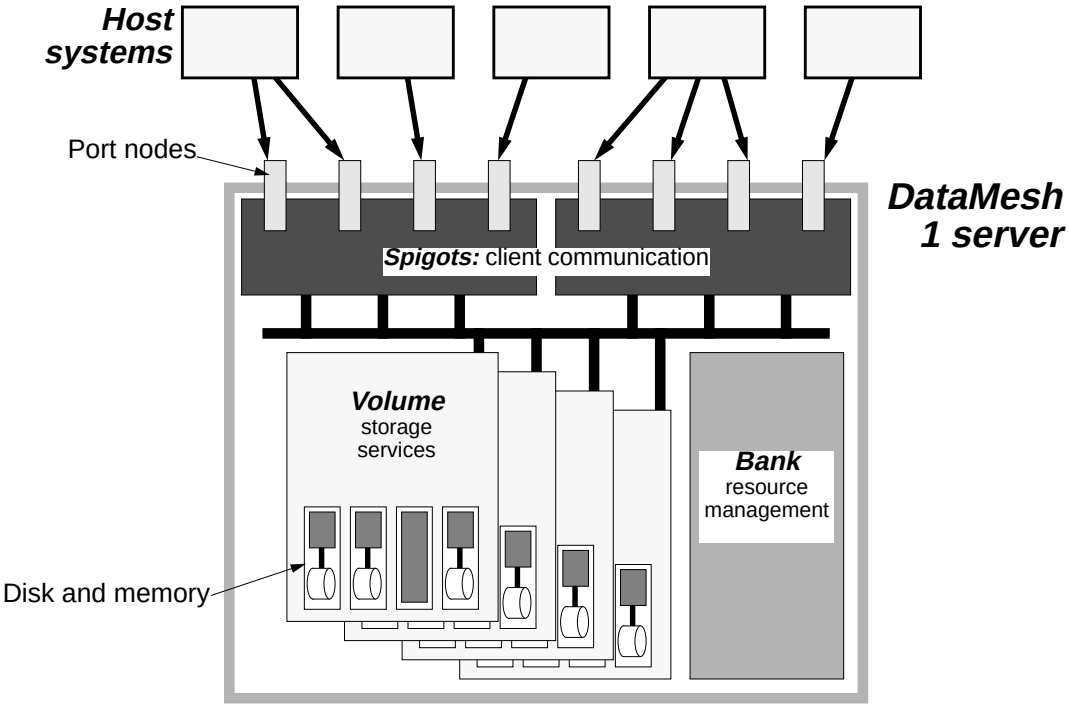

Figure 1. DataMesh 1 overall system structure.

2.3 Blocks, tags and predicates

DataMesh stores data in *packets* with two parts: the *block* and the *tag*. The block contains the data issued by the client. The tag contains metadata associated with the block, either explicitly by the client or implicitly by DataMesh. Among other things, a tag can contain addresses, timestamps, protection and lock information, information pertaining to file system structures, or recovery information. In general, tags will be small—a few bytes of tag per kilobyte of data.

> *Tags are essential to the DataMesh architecture, expressing many functions that other storage systems describe with distinct operations. Commit and abort, for example, are not considered separate actions, but modifications of tag values and deletion of tags respectively (there will be examples of this in the discussion below).*

Each tag can have many named *tag fields* (often referred to simply as *fields*), any or all of which can be interpreted or ignored by DataMesh, and hidden from or made visible to the client. A tag field can either be given a value explicitly or allowed to default—to be treated as if it had the current default value for the field. All tags in a given volume contain the same set of fields, although fields with the default value need not be explicitly stored.

Individual tag fields can either be *sparse* or *compact*. A compact field is one in which the entire range of legal tag values is present in the DataMesh at any given time (e.g., in a conventional disk, the logical block addresses form a compact tag field). A sparse tag field is one that is not compact—for example, timestamps associated with individual atomic writes for a volume that records such things.

A *tag predicate* specifies a set of tag values on which an operation is to be performed. A tag is said to *match* a tag predicate if the tag value is contained within the specified set. A tag predicate may be either partially or totally ordered, in which case the ordering is used to specify the order in which the matching takes place. This is important for operations which return a fixed number tags or packets to the client (such as a read), since the operation terminates after the needed packets are found. Tag predicates are constructed from an order list of *tag field predicates*, just as tags are constructed from a set of fields.

2.4 Volumes

A volume is the unit of storage management in a DataMesh. Volumes control a set of storage resources and offer services to clients. The internal structure of a volume is shown in Figure 1.

Incoming requests are handled as follows:

- They go first to a *dealer*, which performs a fast load balancing function based on the tags or tag predicates in the request. The outcome of the load-balancing is that the dealer forwards the request to one or more *decks* for processing. (The dealer function is designed to be easy to replicate widely so that it can be performed

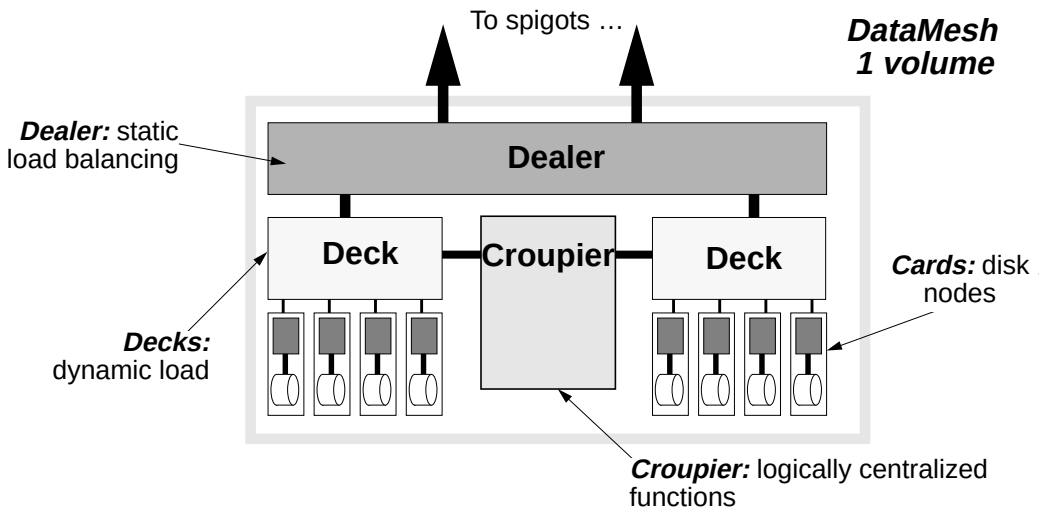

Figure 2. DataMesh 1 volume internal structure.

very quickly—indeed, we anticipate that there will be a copy of a volume's dealer at every port node. Static allocation policies are one example that do not need a dealer to converse with its peers.)

- The deck first ensures that it is indeed the correct recipient of the request: if it is, and it can process the request on its own, it does so. If it is not the correct recipient (e.g., the data has migrated to another deck), the request is forwarded to one or more other decks. (We allow such occurrences to keep the dealers fast in the common case. If they make sufficiently many bad forwarding decisions, this may be used as the signal to recompute the dealer forwarding heuristic.)

  Decks interpret tag predicates in the requests they are given, and select or create the packets on which to operate accordingly. Using this information, a deck converts a request into one or more operations on storage locations, together with appropriate internal data structure modifications.

  If the deck needs to interact with other decks, it invokes the services of a *croupier*, which performs logically centralized functions for the volume. (Examples include volume-wide request sequencing, atomicity coordination across multiple decks, and volume-level locking.)

- Operations on particular storage locations are passed on to the appropriate *cards* (disk nodes) to perform the associated physical I/O operation.

The following table shows how choices at different layers in this structure can be used to create a wide range of different volume types. Note that a single component change can affect a considerable change in behavior (e.g., the switch from a simple striped disk array to a RAID array is effected by changing only the deck module).

Table 1: Sample DataMesh volume policies.

	Plain	*RAID[1-6]*	*Loge*[a]	*MultiLoge*	*Sparse MultiLoge*
Dealer	pass through	pass through	pass through	pass through	hash on block number
Deck	pass through	parity, recovery	indirection table gives block #	indirection table gives block #, card #	as in MultiLoge
Card	pass through	pass through	manage free map	manage free map	manage free map

a. Loge, MultiLoge, and Sparse Loge are all described in [English92].

2.5 Ordering of operations

The ordering of operations with respect to one another, and with respect to failures, is volume specific, as is the mechanism by which it is specified. The order in which responses to commands are sent back through a single spigot will represent a possible serial schedule for the actual operations.

> *It is possible, for example, to guarantee prefix ordering with regard to the update command stream (as in Mime), but a volume with static data placement policies might relax this constraint to achieve better performance. In both cases, however, a host system that is a client of DataMesh will see requests acknowledged in the order in which they complete, assuming the host–spigot channel preserves ordering of messages.*

3 Tags

Tags constitute the address space of DataMesh. Each volume in a DataMesh stores sets of packets, which are <data, tag> tuples. Data blocks are named and accessed through their tags. Tags provides support for a very wide range of facilities, such as transaction-like visibility rules for changes (all or none of a set of writes can be made visible to other entities). These functions are provided through the operations on the tags themselves.

The operations described in this section are primitives and should be viewed as such. In a typical implementation, it will be possible to bind one or more of these primitives into a single external command. Thus, while a SCSI `write` command might translate into a `write-barrier-sync` sequence inside DataMesh, this is not considered a part of the DataMesh architecture. As a result, bindings are relegated to appendices to the architecture proper. For example, the architected DataMesh functions are defined in terms of read and write operations against single blocks, while a binding to SCSI would include multi-block reads and writes.

While bindings themselves are not part of the architecture, the architecture depends on them to provide certain types of functionality. A binding, for example, can limit the types and ranges of tag predicates, so that tag fields can be used for protection.

The data in a packet with a particular tag can be read and written. A `write` operation takes a tag and a block of data, and stores them for later retrieval. A `read` operation takes a tag predicate and returns any data associated with packets that have tags matching the predicate. The volume will also return the tag(s), but they may or not make it past the binding layer.

Packets can be explicitly discarded with the `free` operation, which takes as argument a tag predicate. A list of the tags that match a tag predicate can be obtained with a `ReadTags` call.

3.1 Creating tag fields

Before a portion of the tag address space can be used, it must be allocated: the field must exist and the set of values it can take must be defined. (Note that no packets are created as a result of any of these operations.)

Extending the address space is done by creating a new tag field for a volume. This also creates a new tag field identifier that can be used to name the field in a tag predicate; this identifier is unique across the lifetime of a volume.

When a field is created it is given a type (e.g., `int`, `double`, `string`); and a default value. At first, the only legal value for the field is the default one. The default value for a field cannot be changed after it is created. All existing tags are implicitly extended to have the new field with its default value. The valid range of a field can be changed using an operation that specifies the legal values for a field, in addition to the default value. If a volume contains tags outside the new range, the operation will fail.

A field may be deleted, after which it will be ignored in tag predicates directed to this volume.

Some volumes may be able to detect that all the old values have been explicitly deleted, some may not. The behavior may even be field-specific: deleting the field may be easy if it is stored in an in-memory data structure, hard if it has been written to disk in each packet.

Specifying a tag predicate in future operations that lies outside the current tag address space will induce a volume-specific response.

3.2 Tag predicates

A tag predicate is a partially ordered set of tags. It is used by clients to refer to packets, and by triggers to detect events. A tag predicate is specified by an ordered list of *tag field predicates*, in the same way that a tag is specified by a set of tag fields.

Tag field predicates are sets of tag field values, and can be specified either as lists of explicit values, or by the use of expressions. For example:

- {1, 2, 3, 4} — a set of values;
- 0 < **N** < 5 — an expression; the symbol **N** is intended to imply the set of natural numbers.
- **X** < 2745 — the largest existing value smaller than a given one (used in persistence operations).

The ordering of the tag field predicates is used to determine overall ordering. The first tag field predicate is the most significant. Each tag field predicate has a flag to indicate whether it should be treated as an ordered list or an unordered set of values. Unordered fields are given a non-deterministic, volume-specific ordering, which may change between invocations.

A tag field is said to *match* a tag field predicate if the predicate contains the tag field. A tag matches a tag predicate if all the fields in the tag match all the corresponding fields in the tag predicate. A tag predicate may match several tags. If this produces more tags than the operation supports (e.g., a two-packet read with a predicate that matches a dozen tags), the needed tags are used in order and the excess ignored. Matching fewer tags than the number required to complete the operation is an error.

The elements in tag field predicate expressions may be constants, global values, and "self" values—references to values associated with the operation, such as an operation sequence number assigned in the DataMesh. Tag predicates cannot express combinations of tag field values other than that provided by the matching of individual field values. This means, for example, that there is no way to express a join.

3.3 Preservations and packet deletion

Packets are protected against deletion through the use of *preservations*. A preservation specifies that packets matching a particular tag predicate should not be deleted. By extension, any packet that does not match an existing preservation may be deleted.

There is also a command to discard a preservation. Although this command has no direct effect on packet operations, it can effect the results of tag predicates by either deleting packets or by protecting them from deletion.

> *Preservations are used to control storage reuse. Since DataMesh stores a tuple space, all packets are potentially valid. There is no intrinsic difference, for example, between two packets with the same logical block number, but different item stamps, and thus no reason to reuse the space occupied by an old packet when a new one is written. The simplest block store requires a preservation of the form "the latest timestamp with a given block number," which specifies implicitly that all packets with earlier time stamps should be reused. This same mechanism extends in a straightforward way to allow consistent data snapshots, by specifying preservation of the most recent packets created before a particular time.*

3.4 Tag remapping

A volume can modify the tag fields of a set of packets through the use of the `MapTags` call. This call acts against a tag predicate and changes the values of all matching tags according to some transformation rules. The call is atomic, with respect to both failures and other operations within DataMesh. It is the basic unit of atomic operation within DataMesh and is used to perform higher-level operations such as commit and abort.

> *This operation can be used to perform multi-block atomic operations: for example, writes to a Mime device could be performed against group_id2, a barrier operation would be performed by mapping all packets with group_id2 to packets with the same values, but with group_id1. Commit would map all packets with group_id1 and group_id2 to the default group value, making them visible to all update streams. MapTags can also be used to implement locking by changing the value of a lock field to a private value.*

3.5 Stability control

DataMesh allows clients limited control over the movement of data from volatile to stable storage (e.g., volatile RAM to disk). In particular, the `sync(predicate)` command instructs DataMesh to insure that all packets matching the given tag predicate have reached the volume's stable storage. The definition of "stable" is volume-specific. For volumes providing redundancy, "stable" will imply redundant storage. For volumes providing only persistence, it will merely imply that the designated packets have reached non-volatile media.

> *We have chosen not to support intermediate levels of stability in this release of the architecture document, although we anticipate that we will revisit this in the future—for example, for volumes that separate recording the initial copy of a write from the provision of redundancy for it.*

4 Logical components

This section describes the functionality of the internal software components in greater detail.

4.1 Spigots

Spigots provide peer-to-peer communication channels to the DataMesh host(s). Each spigot has associated port(s) on which commands are given to the DataMesh, and results are returned. Incoming requests are forwarded by the spigot to the volume to which they are addressed. A port node belongs to a single spigot; a single spigot can use multiple port nodes to manage its traffic. A host-to-port connection is termed a *channel*; a single host may be connected to a DataMesh through multiple channels.

Spigots are responsible for the lower levels of the protocol used over the port. The host counterpart of a spigot is a (physical) device adapter driver, such as a (shared) device driver for all the devices on a SCSI string attached to an I/O card. The disk driver itself communicates across the channels to volumes—the spigot layer is an intermediary. As part of its communications responsibilities, the spigot may attach a sequence number and/or a time stamp to all incoming requests.

> *We believe that additional performance can be obtained through the use of two techniques for increasing bandwidth and reducing request queueing delays across multiple channels to a single host. The spigot layer is responsible for implementing these functions:*
>
> – *Channel striping—sending parts of a single I/O request or response across multiple channels at once (modulo queueing effects). In the absence of queueing, striping will reduce the latency for completing the data transfer phase of individual large I/O requests.*
>
> – *Multichannelling—using more than one channel to communicate between a host and a DataMesh, but continuing to send each individual request in its entirety across a single channel. Fixed multichannelling requires the response to return on the same channel as the request; split-mode multichannelling lifts this restriction. Multichannelling should improve the latencies for small requests (fewer queueing delays), and increase the overall bandwidth to the host, although individual large requests will complete no faster.*

Spigots are also responsible for *request ordering*—the mechanism by which hosts arrange that the sequence in which they issue requests is reflected in the DataMesh. Spigots present commands to volumes in the order in which the requests arrive, though a volume is free to execute commands in any order consistent with its policies. The precise rules by which "the order in which the requests arrive" is determined may vary—in particular, two requests arriving at different ports (even for the same spigot) may be labelled with a different arrival order than expected if they arrive within some (small) interval of one another; and/or requests that arrive at least this far apart in time may be guaranteed to be sequenced correctly. Such policies are spigot-specific.

Commands sent to different spigots may be sequenced externally, either through the use of a compact sequence of integers, or by sending a second command only after the response to the first has been received.

4.2 Bindings

Bindings map the protocol and command set used by the clients (e.g., IPI-3, SCSI-2) into the set of architected operations defined in this document. Bindings are associated with spigot/volume pairs, and logically part of a volume, although they may be located at port nodes. Bindings can either be simple and stateless (as in SCSI) or complex and stateful (as in Mime). Some bindings may use the (stable) storage provided by their volumes to record information that must survive across failures.

4.3 Volumes

Volumes encapsulate a set of storage resources, a set of operations defined against that storage, and a set of properties that the storage provides. Volumes are independent: there is no sharing of data across volumes, and there are no multi-volume operations.

4.3.1 Dealers

Dealers distribute work across decks. The dealer looks at a request, and chooses the appropriate deck to send the request to based on information in the tag predicate. Hashing on a block address, or a portion of a block address, is the archetypal dealer function. It is stateless—requiring almost no interactions between dealers—and small—the only data involved is the hash function itself.[1] Another possible dealer mapping is to divide the volume into contiguous ranges. While this second example does have shared data, the amount of data is small (equal to the number of dealers), and as long as the update rate is small, will not represent a load on the system.

> *To minimize communication overheads, dealers are designed to be heavily replicated. We expect that there will be an instance of a volume's dealer at every port through which the volume is communicated. As a result, the primary design criteria for dealers are compactness and stability: dealer data structures should be small and change very slowly compared to the overall load on the system.*

4.3.2 Decks

Decks are responsible for matching tag predicates to packets. Unlike dealers, decks have highly dynamic, storage-intensive mapping functions. The archetypal deck function is an indirection table, where each block is mapped to a storage location independently and where mappings can change with every operation. The storage-intensive nature of the deck's mapping function places practical limits on the size of a single deck, and requires the dealer function described above to allow larger systems. The number of cards under the control of a single deck is limited primarily by the size of the deck's data structures.

> *In Mime, the RAM requirements of a deck grow linearly with the number of data blocks stored on its cards: as the number of cards grows, so does the number of available blocks, and the deck storage required to manage them. On a Sparse Mime volume, for example, each additional 1GB of storage requires about 4MB of RAM at the deck. If decks are limited to 64MB, a single deck can only service 16GB of storage, or about eight cards. Since the main benefits the system receives by increasing the size of the deck are an increase in parallelism and improved load balancing, the most significant improvements will be achieved at relatively small deck sizes, and the size limitation should not present a performance problem.*

The following types of functions are executed at the deck level:

- Indexing—mapping tags into physical storage locations;
- Space balancing—equalizing data across cards.
- Data redundancy—both on and across cards, for both reliability and performance; examples of redundancy policies include full replication (mirroring), partial redundancy (various kinds of RAID arrays), and on-card replication for better rotation latency on reads.
- Local I/O pattern assessment and prediction, which is used to provide hints to cards about prefetching or caching, as well as to perform data shuffling (moving blocks around to improve performance) and dynamic load balancing (moving blocks between cards to equalize the dynamic load—again, to increase performance).
- Packet-level protection and/or locking.
- Atomic update primitives and recovery mechanisms.

Since packet reclamation is expressed in terms of tag predicates, decks are also responsible for reclaiming storage. The actual tables representing physical storage are kept at the cards, but the deck is responsible for notifying the cards that a particular packet is no longer needed and can be reclaimed.

For the most part, decks operate independently. If a volume requires global synchronization or an atomic operation that spans multiple decks, decks coordinate through croupiers.

4.3.3 Croupiers

Croupiers perform logically centralized functions across the entire volume. One example is the coordination of operations across multiple decks (for example, multi-deck atomic write operations). A typical croupier function is to act as a coordinator for a two-phase commit protocol. The following operations are performed specifically at the

1. Changes in configuration, such as bringing a new deck on line, or a failure of an existing one, will require exchange of information between dealers to modify the hash functions.

croupier level. As usual, provision of these facilities is a function of the volume semantics: some or all may not be present.

- Resource management: allocating resources to particular functions within a volume, and the interface to the DataMesh-wide configuration manager (the bank).
- Volume identification: providing information about what a volume can do, such as its capacity, performance and availability characteristics.
- I/O pattern assessment and prediction (e.g., for caching hints to be passed down to the decks/cards).
- Multi-deck atomic operations.
- Global operation ordering.

  Every effort should be made to limit the number of actions that require centralized functions at the croupier level during normal operation. While this level is the only one capable of performing global actions across the entire volume, such actions are inherently non-scalable. One example that we currently know of occurs when a volume must generate a global sequence number to assure atomic update and prefix semantics.

4.4 Cards

Cards provide persistent storage. Different cards may have different performance or reliability characteristics, either because of differences in the underlying hardware of because of differences in the way the hardware is used. One node might have non-volatile RAM buffers to accelerate writes, for example, while another might have data placement policies that favor sequential read performance over write performance. Cards provide a sets of fixed-sized slots into which packets may be written. Once a packet has been written into a slot it is persistent.

Cards specify the atomicity guarantees they provide on store operations. All cards support read and update-in-place store operations. Some cards (such as those derived from Loge) support shadow writes, which are used in preference to update. Such cards maintain a list of free slots available for writing and select one when a write request arrives. The slot number is returned to the caller, and the slot is removed from the free list until it is explicitly freed.

> *Typical values for the atomicity guarantees would be none, valid/invalid sector (a write either completes at a sector boundary or trashes a sector in a detectable way: this is typical behavior of a regular disk drive), full-sector (writes always complete to a sector boundary), or whole-slot.*

Other card-level functions are:

- Packet-level cache management, using hints provided by the deck and/or built in policy decisions (such as “predict sequential read-ahead if accesses are to consecutive addresses”).
- Request execution scheduling, to maximize performance, possibly subject to card/volume-specific sequencing constraints.This means that out-of-order operation is possible, as is concurrent request processing. (Examples: 2D scheduling, SCSI command queueing, no write-reordering, explicit barriers, partial operation orderings.)
- Bad-sector/track sparing.

5 Operations and interfaces

The preceding sections have presented the outline of the components in the system. This section presents descriptions of the interfaces of each of the logical components. The operation descriptions are presented in the form of tables describing the effects that are caused by the invocation of a particular function at the indicated level.

Table 2: Operations on volumes

Operation	*Effect*
read (tag_predicate) → packet	Returns the first packet found matching the tag predicate.
read_tag (tag_predicate) → list of tags	Returns a list of all tags matching the predicate.
write (packet) → tag	Store a packet and return the resulting tag value.
free (tag_predicate)	Delete the tags corresponding to the specified tag range.
preserve (tag_predicate)	Prevents packets matching the predicate from being re-used.
release (tag_predicate)	Reduces the range of preserved values. May cause packets to be re-used.
sync(tag_predicate)	Causes all packets matching the predicate, and any internal information necessary to preserve semantic integrity, to be written to persistent media.
map (tag_predicate, transform)	Change all tags matching the tag_predicate according to the transform function. This function is atomic.

Table 3: Operations on dealers

operation	*effect*
deck (tag_predicate) → list of decks	maps tag predicates onto decks *Note that this may be only partially successful, and that a dealer may return a list of all decks which must then be searched.*

Table 4. Operations on decks.

operation	*effect*
read (tag_predicate) → (card#,slot#)	Translate tag_predicate into card and slot address, pass on to card. Return error if packet not found. Return first match if more than one possibility.
read_tag (tag_predicate) → list of tags	Return a list of tags to the caller. If necessary, assemble the information from cards.
write (packet) → card#	Choose a card to send a data packet to. Send the packet to that card to be stored.
free (tag_predicate)	Delete the tag predicates from the deck. Remove references to associated storage.
map(tag_predicate, transform) → present	Perform an atomic transformation against a set of tags. "Present" is true if this deck contains packets matching the predicate.
sync (tag_predicate)	Make any preceding deck operations on these tags non-volatile, as well as any "operation log" that is needed to record additional operations (such as free and map).
preserve (tag_predicate) → preservation	Prevents packets matching the predicate from being re-used.
release (preservation)	The indicated preservation no longer ensures the existence of packets.

Table 5. Operations on cards.

operation	*effect*
read (slot#) → packet	Retrieve data from a slot
write (packet) → slot#	Writes a packet to a free slot and returns its address.
free (slot#)	Frees a slot.
update (slot#, packet)	stores a <tag,data> pair into the given slot.
sync (tag predicate)	Upon return, guarantees that all packets matching the predicate have reached stable storage.
read_tag (slot range) → set of tags	read the tags associated with a set of slots

Appendix A: Constructing Mime in DataMesh

An important test of the DataMesh software architecture is its ability to cover implementations we have already designed and are in the process of building. The most complicated of these at the present time is Mime [Chao92], and since Mime subsumes all of the Loge work, it is sufficient to cover Loge. This description will assume a generalized Mime, with several decks and several cards per deck.

The main areas to be covered are the binding of Mime's external operations to the DataMesh primitives and the implementation of low-level storage operations from the decks to the cards. We will also cover the implementation of multi-deck atomic operation, though that is not at the present covered by Mime. We will not discuss the implications of multi-spigot prefix semantics, but will for the present assume that all operations come through a single spigot, which assigns them an ascending sequence number.

This section assumes familiarity with the Mime and Loge work described in [English92] and [Chao92].

A.1 Expressing Mime operations in DataMesh

The external operations available in Mime are summarized in the following table:

Table 6: Mime external interface

Operation	*Description*
read (block#, group id) → data	Returns a block of data relative to a group.
write (block#, group id)	Stores a block of data relative to a group.
delete (block#, group id)	Deletes a block of data.
read_map (block range, group id) → list of blocks	Returns the list of valid blocks in the specified range.
new group () → group id	Allocates a new group id.
group status (group id) → status	Determines whether a group is active, recently committed, or aborted.
barrier (group id)	Creates a *savepoint* within a group.
sync (group id)	Causes all outstanding operations to the group to be written to non-volatile media.
commit (group id)	Causes all updates within the group to become part of the default (permanent) group. Eliminates the group.
abort (group id)	Eliminates the group without moving the updates to the default group.

Mime requires a set of tag fields to be associated with each data block. These are listed in Table 7.

Mime's *groups* and *group operations* can be described quite simply in terms of tags and tag operations. In DataMesh, group IDs are simply a tag field, and operations with respect to groups are operations on tags and tag predicates. Group management (`new group`, `group status`, `commit`, and `abort`) is handled by the binding, which keeps a record of outstanding commit groups and allocates `group id`'s for the client. Each group id is translated into a pair of tag field values, a *temporary* one for writes as they occur, and a *protected* one to record writes protected by a barrier. Reads are performed against an ordered tag predicate that specifies the temporary tag value first, the barrier-protected tag value second, and the default (permanent) tag value last. Deletes are performed by setting *data*

Table 7. Fields in Mime tags.

field in tag	*Storage*	*purpose*
block number	Card, deck	names the block a segment's data belongs to
sequence number	Card	monotonic counter used to determine the most recent version of a block with multiple copies
group id	Card, Deck	Marks packet as part of an atomic group.
commit	Card	Indicates that all prior operations are permanent.
data present	Deck	Indicates that valid data is associated with the tag.

present to false (this allows deletes within atomic groups to propagate into the base group). Syncs are translated directly into the corresponding DataMesh sync operation.

Barriers and commits are performed with the DataMesh `map` function, which performs an atomic change of tag values across a range. In the case of a barrier operation, the function transforms all tags with the temporary group id field value into tags with the protected group id field value. In the case of a commit, all tags matching either the temporary or the protected value are transformed into the permanent group value. Aborts are merely a special case of delete, where the entire range of tags matching the temporary or protected group id values are deleted.

These mappings can be tabulated as follows:

Table 8: Mime operations expressed in DataMesh

Operation	*Mapping*
read (block#, group id)	read (block==block#, group = {temp_group_id, prot _group_id, perm})
write (block#, data)	write (block==block#, group:= temp_group_id)
delete (block#, group id)	write (block==block#, group := temp_group_id, data present := false)
read_map (block range, group id) → list of blocks	read_tags (first ≤ block ≤ last, group = {temp_group_id, prot_group_id, perm})
new group () → group id	preserve (all blocks, temp_group_id:=group_id++); preserve (all blocks, prot_group_id:=group_id++).
group status (group id) → status	Check list of active groups at volume. The volume can reconstruct this list by analyzing the list of preservations.
barrier (group id)	maptags (all blocks, temp_group_id, id := prot_group_id)
sync (group id)	sync (all blocks, (temp_group_id, prot_group_id))
commit (group id)	map ((all blocks, (temp_group_id, prot_group_id)), id := permanent)
abort (group id)	release (all blocks, (temp_group_id, prot_group_id))

Note that the top level operations are all converted into operations relative to various group identifiers, and that the creation and deletion of groups is accomplished by preserving and releasing the relevant tag ranges.

A.2 Card functions

Mime requires only a few card functions to operate correctly. These are summarized in Table 5. All of these operations have immediate parallels in DataMesh.

Table 9. Operations on Mime cards.

operation	*effect*
read (slot#) → packet	retrieve data from a segment
write (packet) → slot#	writes a <data,tag> pair in a free segment (as marked in the free map) and returns its address. The segment is removed from the free map
free (slot#)	marks the free map entry for a segment so that it can be written. A card never marks a segment free—only the deck
update (slot#, packet)	stores a <data,tag> pair into the given segment (only used for writing out deck data structure checkpoints)
read_tag (set of slots) → set of tags	read the tags associated with a set of segments