High level design of a Colibri lostore module

By Nikita Danilov <nikita_danilov@xyratex.com>

Date: 2011/06/02
Revision: 1.0

[Text in square brackets and with a light-green background is a commentary explaining the structure of a design document. The rest is a fictional design document, used as a running example.]

This document presents a high level design (HLD) of a lower store (lostore) module of Colibri core. The main purposes of this document are: (i) to be inspected by C2 architects and peer designers to ascertain that high level design is aligned with C2 architecture and other designs, and contains no defects, (ii) to be a source of material for Active Reviews of Intermediate Design (ARID) and detailed level design (DLD) of the same component, (iii) to serve as a design reference document.

The intended audience of this document consists of C2 customers, architects, designers and developers.

0. Introduction

[This section succinctly introduces the subject matter of the design. 1--2 paragraphs.]

The following color marking is used in this document: incomplete or todo item, possible design extension or future directions.]

Iostore module implements functionality common between various Colibri stores: mdstore and iostore. It provides an interface to the underlying data-base, while hiding details of particular data-base implementation.

1. Definitions

[Definitions of terms and concepts used by the design go here. The definitions must be as precise as possible. References to the C2 Glossary are permitted and encouraged. Agreed upon terminology should be incorporated in the glossary.]

- A *table* is a collection of *pairs*, each consisting of a *key* and a *record*. Keys and records of pairs in a given table have the same structure as defined by the table *type*. "Container" might be a better term for collection of pairs (compare with various "container libraries"), but this term is already used by Colibri;
- records consist of fields and some of fields can be *pointers* to pairs in (the same or other) table;
- a table is *ordered*, if a total ordering is defined on possible keys of its records. For an ordered table an interface is defined to iterate over existing keys in order;
- a *sack* is a special type of table, where key carries no semantic information and acts purely as an opaque record identifier, called record *address*. Addresses are assigned to records by the Iostore module;
- a table is *persistent*, if its contents survives a certain class of failures, *viz*. "power failures". A table can be created persistent or non-persistent (*volatile*);
- tables are stored in *segments*. A segment is an array of *pages*, backed by either persistent or volatile storage. Each page can be assigned to a particular table. A table can be stored in the multiple segments and a segment can store pages belonging to the multiple table. Assignment of pages to tables is done by the Iostore module.
- updates to one of more tables can be grouped in a *transaction*, which is a set of updates atomic with respect to a failure (from the same class as used in the definition of persistent table). By abuse of terminology, an update of a volatile table can also be said to belong to a transaction;
- after a transaction is *opened*, updates can be added to the transaction, until the transaction is *closed* by either *committing* or *aborting*. Updates added to an aborted transaction are reverted (*rolled-back* or *undone*). In the absence of failures, a committed transaction

eventually becomes *persistent*, which means that its updates will survive any further power failures. On a recovery from a power failure, committed, but not yet persistent transactions are rolled back by the lostore implementation;

- a function call is *blocking*, if before return it waits for
    - a completion of a network communication, or
    - a completion of a storage transfer operation, or
    - a *long-term* synchronisation event, where the class of long-term events is to be defined later.

  Otherwise a function call is called *non-blocking*.

2. Requirements

[This section enumerates requirements collected and reviewed at the Requirements Analysis (RA) and Requirements Inspection (RI) phases of development. References to the appropriate RA and RI documents should go here. In addition this section lists architecture level requirements for the component from the Summary requirements table and appropriate architecture documentation.]

General requirement for lostore is to provide non-blocking access to an optionally persistent transactional key-value store. Specific requirements for lostore module are enumerated in the *Colibri store change overview* document:

- R.C2.MDSTORE.BACKEND.VARIABILITY: Supports various implementations: db5 and RVM
- R.C2.MDSTORE.SCHEMA.EXPLICIT: Entities and their relations are explicitly documented. "Foreign key" following access functions provided.
- R.C2.MDSTORE.SCHEMA.STABLE: Resistant against upgrades and interoperable
- R.C2.MDSTORE.SCHEMA.EXTENSIBLE: Schema can be gradually updated over time without breaking interoperability.
- R.C2.MDSTORE.SCHEMA.EFFICIENT: Locality of reference translates into storage locality. Within blocks (for flash) and across larger extents (for rotating drives)
- R.C2.MDSTORE.PERSISTENT-VOLATILE: Both volatile and persistent entities are supported
- R.C2.MDSTORE.ADVANCED-FEATURES: renaming symlinks, changelog, parent pointers
- R.C2.REQH.DEPENDENCIES   mkdir a; touch a/b
- R.C2.DTM.LOCAL-DISTRIBUTED: the same mechanism is used for distributed transactions and local transactions on multiple cores
- R.C2.MDSTORE.PARTIAL-TXN-WRITEOUT: transactions can be written out partially (requires optional undo logging support)
- R.C2.MDSTORE.NUMA: allocator respects NUMA topology
- R.C2.REQH.10M: performance goal of 10M transactions per second on a 16 core system with a battery backed memory.
- R.C2.MDSTORE.LOOKUP: Lookup of a value by key is supported
- R.C2.MDSTORE.ITERATE: Iteration through records is supported.
- R.C2.MDSTORE.CAN-GROW: The linear size of the address space can grow dynamically
- R.C2.MDSTORE.SPARSE-PROVISIONING: including pre-allocation
- R.C2.MDSTORE.COMPACT, R.C2.MDSTORE.DEFRAGMENT: used container space can be compacted and de-fragmented

- R.C2.MDSTORE.FSCK: scavenger is supported
- R.C2.MDSTORE.PERSISTENT-MEMORY: The log and dirty pages are (optionally) in a persistent memory
- R.C2.MDSTORE.SEGMENT-SERVER-REMOTE: backing containers can be either local or remote

- R.C2.MDSTORE.ADDRESS-MAPPING-OFFSETS: offset structure friendly to container migration and merging
- R.C2.MDSTORE.SNAPSHOTS: snapshots are supported
- R.C2.MDSTORE.SLABS-ON-VOLUMES: slab-based space allocator
- R.C2.MDSTORE.SEGMENT-LAYOUT: Any object layout for a meta-data segment is supported
- R.C2.MDSTORE.DATA.MDKEY: Data objects carry meta-data key for sorting (like reiser4 key assignment does).
- R.C2.MDSTORE.RECOVERY-SIMPLER: There is a possibility of doing a recovery twice. There is also a possibility to use either object level mirroring or a logical transaction mirroring.
- R.C2.MDSTORE.CRYPTOGRAPHY: optionally meta-data records are encrypted
- R.C2.MDSTORE.PROXY: proxy meta-data server is supported. A client and a server are almost identical.

3. Design highlights

[This section briefly summarises key design decisions that are important for understanding of the functional and logical specifications and enumerates topics that a reader is advised to pay special attention to.]

Key problem of lostore interface design is to accommodate for different implementations, *viz*., a db5-based one and RVM-based. To address this problem, keys, records, tables and their relationships are carefully defined in a way that allows different underlying implementations without impairing efficiency.

lostore transaction provide very weak guarantees, compared with the typical ACID transactions:

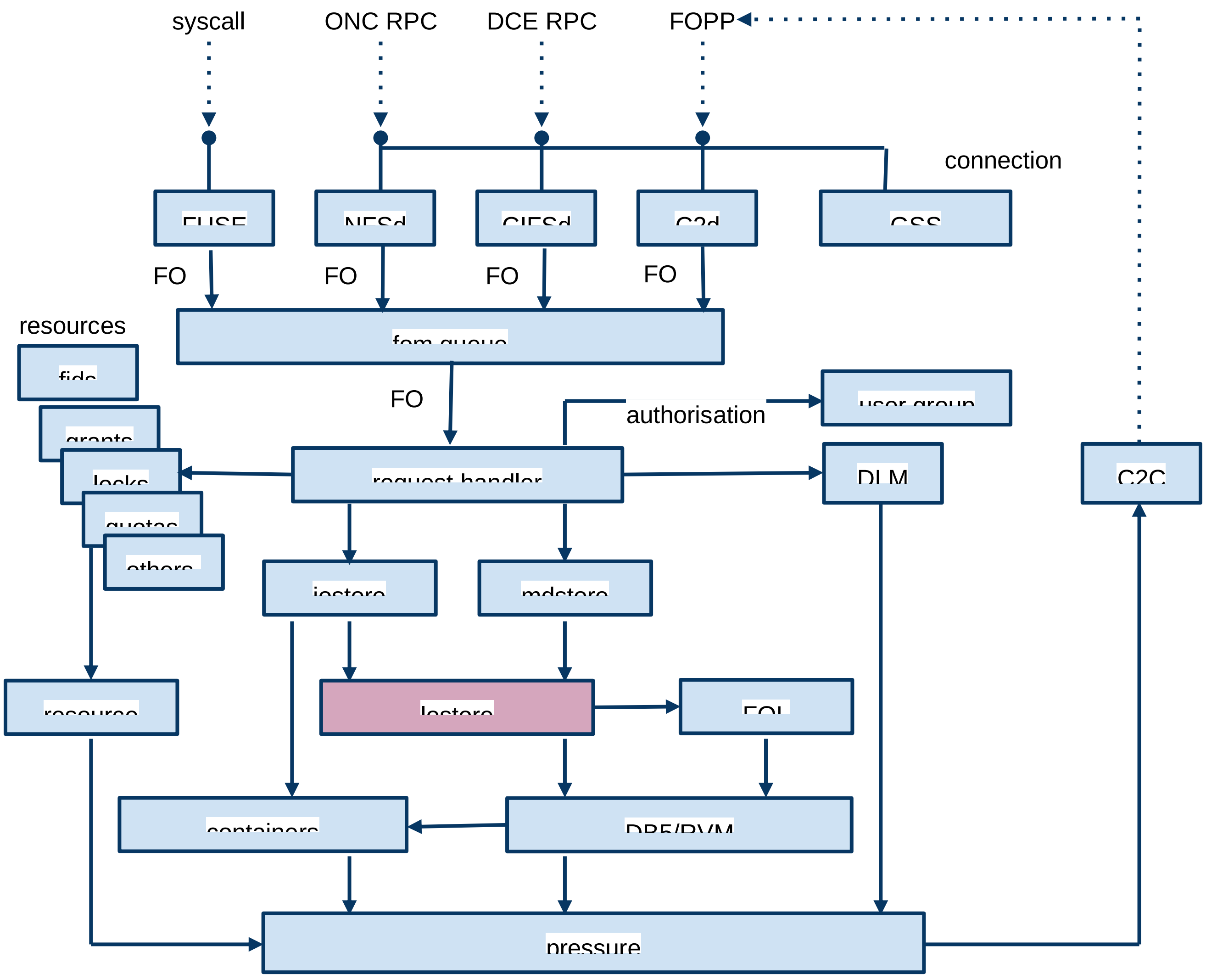
- no isolation: lostore transaction engine does not guarantee transaction serialisability. User has to implement any concurrency control measure necessary. The reason for this decision is that transaction used by Colibri are much shorter and smaller than typical transactions in a general purpose RDBMS and a user is better equipped to implement a locking protocol that guarantees consistency and deadlock freedom. Note, however, that lostore transactions can be aborted and this restricts the class of usable locking protocols;
- no durability: lostore transactions are made durable asynchronously, see the Definitions section above.

The requirement of non-blocking access to tables implies that access is implemented as a state machine, rather than a function call. The same state machine is used to iterate over tables.

4. Functional specification

[This section defines a functional structure of the designed component: the decomposition showing *what* the component does to address the requirements.]

The diagram below shows the system context of the lostore module.



mdstore and iostore use lostore to access a data-base, where meta-data are kept. Meta-data are orgianised according to a *meta-data schema*, which is defined as a set of lostore tables referencing each other together with consistency constraints.

lostore public interface consists of the following major types of entities:

- table type: defines common characteristics of all table of this type, including:
    - structure of keys and records,
    - optional key ordering,
    - usage hints (*e.g.*, how large is the table? Should it be implemented as a b-tree or a hash table?)
- table: defines table attributes, such as persistence, name;
- segment: segment attributes such as volatility or persistency. A segment can be local or

remote;

- transaction: transaction object can be created (opened) and closed (committed or aborted). Persistence notification can be optionally delivered, when the transaction becomes persistent;
- table operation, table iterator: a state machine encoding the state of table operation;
- domain: a collection of tables sharing underlying data-base implementation. A transaction is confined to a single domain.

4.1. Tables

For a table type, a user has to define operations to encode and decode records and keys (unless the table is sack, in which case key encoding and decoding functions are provided automatically) and optional key comparison function. It's expected that encoding and decoding functions will often be generated automatically in a way similar to fop encoding and decoding functions.

Following functions are defined on tables:

- create: create a new instance of a table type. The table created can be either persistent or volatile, as specified by the user. Backing segment can be optionally specified;
- open: open an existing table by name;
- destroy: destroy a table;
- insert: insert a pair into a table;
- lookup: find a pair given its key;
- delete: delete a pair with a given key;
- update: replace pair's record with a new value;
- next: move to the pair with the next key;
- follow: follow a pointer to a record.

4.2. Transactions

The following operations are defined for transactions:

- open: start a new transaction. Transaction flags can be specified, *e.g.*, whether the transaction can be aborted, whether persistence notification is needed.
- add: add an update to the transaction. This is internally called as part of any table update operation (insert, update, delete);
- commit: close the transaction;
- abort: close the transaction and roll it back;
- force: indicate that transaction should be made persistent as soon as possible.

4.3. Segments

The following operations are defined for segments:

- create a segment backed up by a storage object (note that because the meta-data describing the storage object are accessible through lostore, the boot-strapping issues have to be addressed);
- create a segment backed up by a remote storage object;

- destroy a segment, none of which pages are assigned to tables.

5. Logical specification

[This section defines a logical structure of the designed component: the decomposition showing *how* the functional specification is met. Subcomponents and diagrams of their interrelations should go in this section.]

Internally, a lostore domain belongs to one of the following types:

- a db5 domain, where tables are implemented as db5 databases and transactions as db5 transactions;
- an rvm domain, where tables are implemented as hash tables in the RVM segments and transactions as RVM transactions;
- a light domain, where tables are implemented as linked lists in memory and transactions calls are ignored.

5.1. Conformance

[For every requirement in the Requirements section, this sub-section explicitly describes how the requirement is discharged by the design. This section is part of a requirements tracking mechanism, so it should be formatted in some way suitable for (semi-)automatic processing.]

5.2. Dependencies

[This sub-section enumerates other system and external components the component depends on. For every dependency a type of the dependency (uses, generalizes, *etc.*) must be specified together with the particular properties (requirements, invariants) the design depends upon. This section is part of a requirements tracking mechanism.]

5.3. Security model

[The security model, if any, is described here.]

5.4. Refinement

[This sub-section enumerates design level requirements introduced by the design. These requirements are used as input requirements for the detailed level design of the component. This sub-section is part of a requirements tracking mechanism.]

6. State

[This section describes the additions or modifications to the system state (persistent, volatile) introduced by the component. As much of component behavior from the logical specification should be described as state machines as possible. The following sub-sections are repeated for every state machine.]

6.1. States, events, transitions

[This sub-section enumerates state machine states, input and output events and state transitions incurred by the events with a table or diagram of possible state transitions. UML state diagrams can be used here.]

6.2. State invariants

[This sub-section describes relations between parts of the state invariant through the state modifications.]

6.3. Concurrency control

[This sub-section describes what forms of concurrent access are possible and what forms on concurrency control (locking, queuing, *etc.*) are used to maintain consistency.]

7. Use cases

[This section describes how the component interacts with rest of the system and with the outside world.]

7.1. Scenarios

[This sub-section enumerates important use cases (to be later used as seed scenarios for ARID) and describes them in terms of logical specification.]

Scenario	[usecase.component.name]
Relevant quality attributes	[*e.g.*, fault tolerance, scalability, usability, re-usability]
Stimulus	[an incoming event that triggers the use case]
Stimulus source	[system or external world entity that caused the stimulus]
Environment	[part of the system involved in the scenario]
Artifact	[change to the system produced by the stimulus]
Response	[how the component responds to the system change]
Response measure	[qualitative and (preferably) quantitative measures of response that must be maintained]
Questions and issues	

[UML use case diagram can be used to describe a use case.]

7.2. Failures

[This sub-section defines relevant failures and reaction to them. Invariants maintained across the failures must be clearly stated. Reaction to Byzantine failures (*i.e.*, failures where a compromised component acts to invalidate system integrity) is described here.]

8. Analysis

8.1. Scalability

[This sub-section describes how the component reacts to the variation in input and configuration parameters: number of nodes, threads, requests, locks, utilization of resources (processor cycles, network and storage bandwidth, caches), *etc.* Configuration and work-load parameters affecting component behavior must be specified here.]

8.2. Other

[As applicable, this sub-section analyses other aspects of the design, *e.g.*, recoverability of a distributed state consistency, concurrency control issues.]

8.2. Rationale

[This sub-section describes why particular design was selected; what alternatives (alternative designs and variations of the design) were considered and rejected.]

9. Deployment

9.1. Compatibility

[Backward and forward compatibility issues are discussed here. Changes in system invariants (event ordering, failure modes, *etc.*)]

9.1.1. Network

9.1.2. Persistent storage

9.1.3. Core

[Interface changes. Changes to shared in-core data structures.]

9.2. Installation

[How the component is delivered and installed.]

10. References

[References to all external documents (specifications, architecture and requirements documents, *etc.*) are placed here. The rest of the document cites references from this section. Use Google Docs bookmarks to link to the references from the main text.]