High level design of meta-data back-end

By Anatoliy Bilenko <anatoliy_bilenko@xyratex.com>,
Nikita Danilov <nikita_danilov@xyratex.com>,
Maxim Medved <max_medved@xyratex.com>,
Andriy Tkachuk <andriy_tkachuk@xyratex.com>,
Yuriy Umanets <yuriy_umanets@xyratex.com>,
Valery Vorotyntsev <valery_vorotyntsev@xyratex.com>,

Date: 2013/08/12
Revision: 1.0

[Text in square brackets and with a light-green background is a commentary explaining the structure of a design document.]

This document presents a high level design (HLD) of meta-data back-end for Mero. The main purposes of this document are: (i) to be inspected by Mero architects and peer designers to ascertain that high level design is aligned with Mero 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 Mero 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.]

Meta-data back-end (BE) is a module presenting an interface for a transactional local meta-data storage. BE users manipulate and access meta-data structures in memory. BE maps this memory to persistent storage. User groups meta-data updates in transactions. BE guarantees that transactions are atomic in the face of process failures. BE provides support for a few frequently used data-structures: double linked list, B-tree and extmap.

1. Definitions

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

- *a storage object* (*stob*) is a container for unstructured data, accessible through m0_stob interface. BE uses stobs to store meta-data on persistent store. BE accesses persistent store only through m0_stob interface and assumes that every completed stob write survives any node failure. It is up to a stob implementation to guarantee this;

- *a segment* is a stob mapped to an extent in process address space. Each address in the extent uniquely corresponds to the offset in the stob and *vice versa*. Stob is divided into *blocks* of fixed size. Memory extent is divided into *pages* of fixed size. Page size is a multiple of block size (it follows that stob size is a multiple of page size). At a given moment in time, some pages are *up-to-date* (their contents is the same as of the corresponding stob blocks) and some are *dirty* (their contents was modified relative to the stob blocks). In the initial implementation all pages are up-to-date, when the segment is opened. In the later versions, pages will be loaded dynamically on demand. The

memory extent to which a segment is mapped is called *segment memory*;

- *a region* is an extent within segment memory. *A (meta-data) update* is a modification of some region;

- *a transaction* is a collection of updates. User adds an update to a transaction by *capturing* the update's region. User explicitly closes a transaction. BE guarantees that a closed transaction is atomic with respect to process crashes that happen after transaction close call returns. That is, after such a crash, either all or none of transaction updates will be present in the segment memory when the segment is opened next time. If a process crashes before a transaction closes, BE guarantees that none of transaction updates will be present in the segment memory;

- *a credit* is a measure of a group of updates. A credit is a pair `(nr, size)`, where `nr` is the number of updates and `size` is total size in bytes of modified regions.

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.]

See here and here.

- 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.]

BE transaction engine uses write-ahead redo-only logging. Concurrency control is delegated to BE users.

4. Functional specification

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

BE provides interface to make in-memory structures transactionally persistent. A user opens a (previously created) segment. An area of virtual address space is allocated to the segment. The user then reads and writes the memory in this area, by using BE provided interfaces together with normal memory access operations. When memory address is read for the first time, its contents is loaded from the segment (initial BE implementation loads the entire segment stob in memory when the segment is opened). Modifications to segment memory are grouped in transactions. After a transaction is closed, BE asynchronously writes updated memory to the segment stob.

When a segment is closed (perhaps implicitly as a result of a failure) and re-opened again, the same virtual address space area is allocated to it. This guarantees that it is safe to store pointers to segment memory in segment memory. [Future versions of BE might load segment at a different address providing a transparent relocation mechanism.] Because of this property, a user can place in segment memory in-memory structures, relying on pointers: linked lists, trees, hash-tables, strings, *etc*. Some in-memory structures, notably locks, are meaningless on storage, but for simplicity (to avoid allocation and maintenance of a separate set of volatile-only objects), can nevertheless be placed in the segment. When such a structure is modified (*e.g.*, a lock is taken or released), the modification is not captured in any transaction and, hence, is not written to the segment stob.

BE-exported objects (domain, segment, region, transaction, linked list and b-tree) support Mero non-blocking server architecture.

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.]

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.]