High level design of Mero HA interface

By Maxim Medved <max.medved@seagate.com>
Date: 2016/04/06
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 the interface between Mero and HA. 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.]

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.]

HA interface — API that allows Halon to control Mero and allows Mero to receive cluster state information from Halon;

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.]

— HA interface and Spiel include all kinds of interaction between Mero and Halon;
— notification/command ordering is enforced by HA interface;
— HA interface is a reliable transport;
— HA interface is responsible for reliability of event/command/notification delivery;
— HA interface is responsible for reconnecting after endpoint on the other end dies;
— Mero can send an event to Halon on error;
— Mero can send detailed information about the event;
— Halon is responsible for decisions about failures (if something is failed or it is not);
— Halon can query a state of notification/command;
— Each pool repair/rebalance operation has cluster-wide unique identifier;
— HA interface is asynchronous;

Desires:

- Ability to include more error information on failures
- Better handling for failures (e.g. can we confirm we have notified all Mero instances)
- Possibly some additional 'context' for notifications - e.g. holding an identifier for pool repair and using that for follow-up responses
- Better handling for 'cascade' failures. E.g. I don't think it makes sense for Halon to mark all drives failed when a controller fails, because this blunts other failures

Do we want an evolution or a complete rewrite?

Unify spiel pool repair work with state set interface? Need shared identifiers across the two.

Halon hosting confd

====

Define Confd interface in terms of FOPs. Halon could implement response to these. This would simplify bootstrap (no need to start up confd and RM servers prior to anything else). Halon would need to take out the write lock, except during bootstrap when this is not necessary.

Commented [1]: This would create a very tight bind between Mero and Halon. It would be better to keep confd separate, so that it can accumulate Mero-private information.

Commented [2]: OK. Then we'll go the way removing confd mkfs.

Metadata drives:

Add as drives and dangle off of MDS as opposed to IOS.

Epochs and related things:

====

Possibly Halon should hold a version number for every entity in confd. We increment this on any set operation. Mero instances can send a partial version vector for each operation, which can ensure that all services involved in an operation share the same version of each affected object.

Version vector should go up the hardware and software trees, rather than the pool tree. (E.g. we should not include version of pool)

For additional failure context, introduce a union `no_context` to `m0_ha_note` - e.g. for drives this could contain info on "Read error whilst reading from sector x"

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.]

4. Functional specification

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

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.]

Consubstantiation, as proposed by D. Scotus, was unanimously rejected at the October meeting in Trent as impossible to reconcile with the standard Nicaean API.

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.]

[0] Mero and Halon Integration
[1]