High level design of the catalogue service

By Nikita Danilov <nikita.danilov@seagate.com>
Date: 2014/10/01
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 the Mero catalogue service. The main purposes of this document are: (i) to be inspected by the 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.]

Catalogue service (cas) is a Mero service exporting key-value catalogues (indices). Users can access catalogues by sending appropriate fops to an instance of the catalogue service. Externally, a catalogue is a collection of key-value pairs, called records. A user can insert and delete records, lookup records by key and iterate through records in a certain order. A catalogue service does not interpret keys or values, (except that keys are ordered as bit-strings)—semantics are left to users.

Catalogues are used by other Mero sub-systems, specifically by Clovis, to store and access meta-data. Distributed meta-data storage is implemented on top of cas.

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.]

- *catalogue*: a container for records. A catalogue is explicitly created and deleted by a user and has an *identifier*, assigned by the user;
- *record*: a key-value pair;
- *key*: an arbitrary sequence of bytes, used to identify a record in a catalogue;
- *value*: an arbitrary sequence of bytes, associated with a key;
- *key order*: total order, defined on keys within a given container. Iterating through the container, returns keys in this order. The order is defined as lexicographical order of keys, interpreted as bit-strings.
- *user*: any Mero component or external application using a cas instance by sending fops to it.

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.]

- [r.cas.persistency]: modifications to catalogues are stored persistently;
- [r.cas.atomicity]: operations executed as part of particular cas fop are atomic w.r.t. service failures. If the service crashes and restarts, either all or none modifications are visible to the future queries;
- [r.cas.efficiency]: complexity of catalogue query and modification is logarithmic in the number of records in the catalogue;
- [r.cas.vectored]: cas operations are vectored. Multiple records can be queried or updated by the same operation;

- [r.cas.scatter-gather-scatter]: input and output parameters (keys and values) of an operation are provided by the user in arbitrary vectored buffers to avoid data-copy;
- [r.cas.creation]: there is an operation to create and delete a catalogue with user provided identifier;
- [r.cas.identification]: a catalogue is uniquely identified by a non-reusable 120-bit identifier;
- [r.cas.fid]: catalogue identifier, together with a fixed 8-bit prefix form the catalogue fid;
- [r.cas.unique-keys]: record keys are unique within a given catalogue;
- [r.cas.variable-size-keys]: keys with different sizes are supported;
- [r.cas.variable-size-values]: values with different sizes are supported;
- [r.cas.data-integrity]: a user can request, at the catalogue creation time, a particular form of data-integrity support. If such support is requested, each key and value are tagged with a checksum that is, depending on the options
    - optionally computed by the user and passed to the service along with data;
    - optionally verified by the service to guard against corrupted network messages;
    - optionally verified by the service to guard against silent media corruption;
    - optionally passed back to the user to be verified end-to-end;
- [r.cas.locality]: the implementation guarantees that spatial (in key order) together with temporal locality of accesses to the same catalogue is statistically optimal. That is, consecutive access to records with close keys is more efficient than random access;
- [r.cas.cookies]: a service returns to the user an opaque cookie together with every returned record, plus a cookie for a newly inserted record. This cookie is optionally passed by the user (along with the key) to later access the same record. The cookie might speed up the access.

Requirements from the Mero Summary Requirements table: none.

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.]

A catalogue, exported by cas is *local*: records of the catalogue are stored in the meta-data back-end (BE) in the instance where cas is running. A catalogue is implemented as a BE b-tree.

New fid type is registered for catalogue fids.

4. Functional specification

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

Catalogue service introduces and accepts the following fop types:

- CREATE: create a catalogue, given user-supplied fid and flags;
- DELETE: delete a catalogue, given its fid. All records are deleted from the catalogue;

In addition, there are fops for operations on catalogues, which all take catalogue fid as a parameter.

- PUT: given a vector of records, insert each record in the catalogue, or update its value if the record with such key already exists;
- GET: given a vector of keys, lookup and return the vector of matching values, together with indicators for the missing keys;
- DEL: given a vector of keys, delete the matching records from the catalogue;
- NEXT: given a vector of keys, lookup next N (in the ascending key order) records for each key and return them.

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.]

Service

Catalogue service is implemented as a standard request handler service. Catalogue service instance startup is regulated by configuration.

Each catalogue is assigned a locality, based on some hash of catalogue fid. cas foms, created to process operations on a catalogue, are executed in the locality assigned to this catalogue. Multiple foms operating on the same catalogue are synchronised by a fom-long-lock, associated with the catalogue.

Meta-catalogue

Catalogue service instance maintains a meta-catalogue, where all catalogues (possibly including the meta-catalogue) are listed. The meta-catalogue is created when the storage is formatted for the service. The fid of the meta-catalogue is exported to the users together with formats of meta-catalogue keys and values, so that the users can query the meta-catalogue to find existing catalogues, subject to access control restrictions. Direct modifications of the meta-catalogue by the users are not allowed, the meta-catalogue is updated as a result of CREATE and DELETE operations. When a new catalogue is created, its cookie, which is the address of the root of the catalogue b-tree is passed to the client and can be used to bypass meta-catalogue lookup on the following operations on the catalogue.

BE interaction

A catalogue (including the meta-catalogue) is implemented as a BE b-tree. Keys and values in the b-tree are supplied by the cas users. Keys are treated as bit-strings for the purpose of ordering. In other words, `memcmp(3)` can be used as key comparison function and variable length keys compare as if right-padded with zeroes.

Operations in cas fops are vectored. In the first version of the implementation, operation in a fop is translated in a sequence of BE b-tree calls executed in the loop. The upcoming re-implementation of b-tree, might provide an interface to insert, lookup or delete multiple records at once.

In any case all operations from a given fop are executed in the same BE transaction.

Cookies

When a cas instance looks up or creates a record on behalf of a user, it constructs a special cookie and returns it to the user. To operate on the same record (*e.g.*, to delete it or update its value), the user passes this cookie back to the service along with the record key. The cookie is used to speed up access to the record. Similar cookies are used for catalogues, which are records in the meta-catalogue.

The implementation and semantics of cookies are internal to the service. To a user, a cookie is an array of bytes. One possible implementation strategy for cookies is based on `m0_cookie` interface. The service might create m0_cookie for the b-tree leaf, in which the record resides. When this cookie is received from the user, it is dereferenced to reach the leaf node directly bypassing top-down tree traversal. The dereference might fail, because due to tree re-balancing the leaf node can be freed, or the record in question can be moved out of the node. In this case, the tree is traversed top-down.

File operation packets

This sub-section describes details of cas fop execution. Cookies, described in the previous sub-section are omitted from fop parameters.

fop type	input parameters (request fop fields)	output parameters (reply fields)
CREATE	cfid	rc

Allocate and initialise new catalogue object and insert it in the meta-catalogue, using `cfid` as the key. If a catalogue with such key already exists, return -EEXIST to the user. In either case return the cookie of the catalogue to the user.

DELETE	cfid	rc

The problem with delete is that deletion of a catalogue with a large number of records might require more meta-data updates than can fit in a single transaction. Because of this a technique similar to handling of truncates for open-unlinked files in a POSIX file system is used. (Also see stob deletion code in ioservice.)

```
bool deathrowed = false;
tx_open(tx);
cat = catalogue_get(req.cfid);
/*
 * First, remove all existing records.
 * Scan the catalogue.
 */
foreach key in cat {
        tree_del(cat.btree, key, tx);
        if (m0_be_tx_should_break(tx, deathrow_credit)) {
            if (!deathrowed) {
              // if the transaction is about to overflow,
              // put the catalogue in a special “dead row” catalogue.
              tree_insert(service.death_row, cfid, tx);
              // do this only in the first transaction
              deathrowed = true;
            }
            /* reopen the transaction, continue with deletions. */
            tx_close(tx);
            tx_open(tx);
        }
}
/* all records removed, delete the catalogue from the deathrow. */
tree_delete(service.death_row, cfid, tx);
/* delete the empty catalogue from the meta-catalogue, etc. */
…
tx_close(tx);
```

Deathrow catalogue contains all large catalogues which are in the process of being deleted. If the service crashes and restarts, it scans the deathrow and completes pending deletions. In other words, deathrow is used for logical logging of catalogue deletions.

GET	cfid, input: array of {key}	rc, output: array of {exists, val}

```
cat = catalogue_get(req.cfid);
foreach key in req.input {
        reply.output[i].val = tree_lookup(cat.btree, key);
}
```

PUT	cfid, input: array of {key, val}	rc, count

```
reply.count = 0;
cat = catalogue_get(req.cfid);
tx_open(tx);
foreach key, val in req.input {
        tree_insert(cat.tree, key, val, tx);
        if (rc in {0, -EEXIST})
                reply.count++;
        else
                break;
}
tx_close(tx);
```

DEL	cfid, input: array of {key}	rc, count

```
count = 0;
cat = catalogue_get(req.cfid);
tx_open(tx);
foreach key in req.input {
        tree_del(cat.tree, key, tx);
        if (rc == 0)
                reply.count++;
        else if (rc == -ENOENT)
                ; /* Nothing. */
        else
                break;
}
tx_close(tx);
```

NEXT	cfid, input: array of {key, nr}	rc, output: array of { rec: array of { key, val } }

```
count = 0;
cat = catalogue_get(req.cfid);
foreach key, nr in req.input {
        cursor = tree_cursor_position(cat.tree, key);
        for (i = 0; i < nr; ++i) {
                reply.output[count].rec[i] = tree_cursor_get(cursor);
                tree_cursor_next(cursor);
        }
        count++;
}
```

To bulk or not to bulk?

The detailed level design of the catalogue service should decide on use of rpc bulk mechanism. Possibilities include:

- do not use bulk, pass all records in fop data. This imposes a limit on total records size in the operation;
- use bulk all the time, do not pass records in fop data. This requires keys and data to be page aligned;
- use fop data up to a certain limit, use bulk otherwise.

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.]

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.]

- reqh service
- fom, fom-long-term-lock
- be (tx, btree)

Security model

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

None at the moment. Security model should be designed for all storage objects together.

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.]

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.]

State invariants

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

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.]

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.]

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

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.]

Other

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

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

Compatibility

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

Network

Persistent storage

Core

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

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.]

Clovis presentation
Meta-data back-end