High level design of the distributed indexing

By Nikita Danilov <nikita.danilov@seagate.com>
Date: 2016/05/01
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 distributed indexing. 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.

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.]

Distributed indexing is a Mero component that provides key-value indices distributed over multiple storage devices and network nodes in the cluster for performance, scalability and fault tolerance.

Distributed indices are exported via Clovis interface and provide applications with a method to store application-level meta-data. Distributed indices are also used internally by Mero to store certain internal meta-data. Distributed indexing is implemented on top of "non-distributed indexing" provided by the catalogue service (cas).

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.]

- definitions from the high level design of catalogue service are included: *catalogue*, *identifier*, *record*, *key*, *value*, *key order*, *user*;
- definitions from the high level design of parity de-clustered algorithm are included: *pool*, *P*, *N*, *K*, *spare slot*, *failure vector*;
- a *distributed index*, or simply *index* is an ordered container of key-value records;
- a *component catalogue* of a distributed index is a non-distributed catalogue of key-value record, provided by the catalogue service, in which the records of the distributed index are stored.

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.idx.entity]: an index is a Clovis entity with a fid. There is a fid type for distributed indices;
- [r.idx.layout]: an index has a layout attribute, which determines how the index is stored in non-distributed catalogues;
- [r.idx.pdclust]: an index is stored according to a parity de-clustered layout with N = 1, *i.e.*, some form of replication. The existing parity de-clustered code is re-used;
- [r.idx.hash]: partition of index records into parity groups is done via key hashing. The hash of a key determines the parity group (in the sense of parity de-clustered layout algorithm) and, therefore, the location of all replicas and spare spaces;
- [r.idx.hash-tune]: the layout of an index can specify one of the pre-determined hash functions and specify the part of the key used as the input for the hash function. This provides an application with some degree of control over locality;
- [r.idx.cas]: indices are built on top of catalogues and use appropriately extended catalogue service;
- [r.idx.repair]: distributed indexing sub-system has a mechanism of background

repair. In case of a permanent storage failure, index repair restores redundancy by generating more replicas in spare space;

- [r.idx.re-balance]: distributed indexing sub-system has a mechanism of background re-balance. When a replacement hardware element (a device, a node, a rack) in added to the system, re-balance copies appropriate replicas from the spare space to the replacement unit;
- [r.idx.repair.reuse]: index repair and re-balance, if possible, are built on top of copy machine abstraction used by the SNS repair and re-balance;
- [r.idx.degraded-mode]: access to indices is possible during repair and re-balance;
- [r.idx.root-index]: a *root index* is provided, which has a known built-in layout and, hence, can be accessed without learning its layout first. The root index is stored in a pre-determined pool, specified in the configuration data-base. The root index contains a small number of global records;
- [r.idx.layout-index]: a *layout index* is provided, which contains `(key: index-fid, value: index-layout-id)` records for all indices except the root index, itself and other indices mentioned in the root index. The layout of the layout index is stored in the root index. Multiple indices can use the same layout-id;
- [r.idx.layout-descr]: a *layout descriptor index* is provided, which contains `(key: index-layout-id, value: index-layout-descriptor)` records for all indices;
- [r.idx.data-integrity]: Clovis data-integrity interface is supported for indices;

Relevant requirements from the Mero Summary Requirements table:

- [r.m0.cmd]: clustered meta-data are supported;
- [r.m0.layout.layid]: a layout is uniquely identified by a layout id (layid);
- [r.m0.layout.meta-data]: layouts for meta-data are supported.

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.]

An index is stored in a collection of catalogues, referred to as component catalogues (similarly to a component object, cob), distributed across the pool according to the index layout. Individual component catalogues are either created during explicit index creation operation or created lazily on the first access.

To access the index record with a known key, the hash of the key is calculated and used as the data unit index input of parity de-clustered layout algorithm. The algorithm outputs the locations of N+K component catalogues, where the replicas of the record are located and S component catalogues that hold spare space for the record. Each component catalogue stores a subset of records of the index without any transformation of keys or values.

Iteration through an index from a given starting key is implementing by querying all component catalogues about records following the key and merge-sorting the results. This requires updating catalogue service to correctly handle NEXT operation with a non-existent starting key.

Commented [1]: what's the current status of this?

New fid type is registered for index fids.

Functional specification

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

Indices are available through Clovis interface.

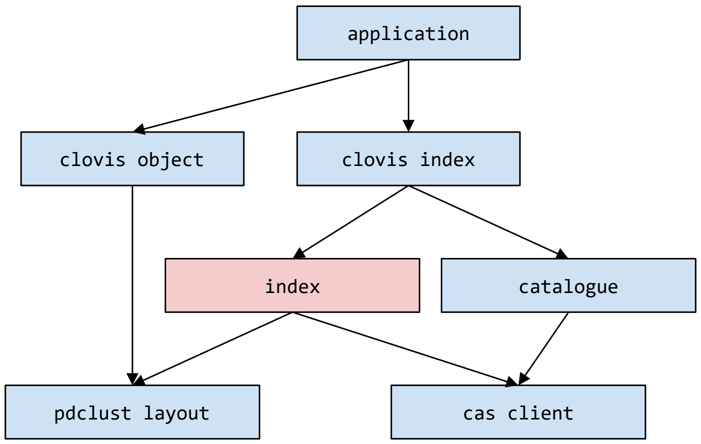
Clovis interface is extended to indicate the pool in which new entity is created. Until this is done, indices are created in the default pool.

Spiel and HA interfaces are extended to control repair and re-balance of indices.

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.]

The following context diagram shows the relations between distributed index sub-system and other components.



Index layout

Index layouts are based on N+K+S parity de-clustered layout algorithm, with the following modifications:

- N = 1. The layout provides (K+1)-way replication;
- parity de-clustered layouts for data objects come with unit size as a parameter. Unit size is used to calculate parity group number, which is an essential input to the parity de-clustered layout function. For indices there is no natural way to partition key-space into units, so the implementation should provide some flexibility to select suitable partitioning. One possible (but not mandated) design is calculate unit number by specifying an *identity mask* within a key:
    - identity mask is a sequence of ranges of bit positions in index key (keys are considered as bit-strings): $[S_0, E_0]$, $[S_1, E_1]$, ..., $[S_m, E_m]$, here S_i and E_i are bit-offsets counted from 0. The ranges can be empty, overlapping and are not necessarily monotone offset-wise;
    - given a key bit-string X, calculate its *seed* as

      $$\texttt{seed} = X[S_0, E_0]::X[S_1, E_1]:: \ldots :: X[S_m, E_m],$$

      where `::` is bit-string concatenation;
    - if the layout is hashed (a Boolean parameter), then the key belongs to the parity group `hash(seed)`, where hash is some fixed hash function, otherwise (not hashed), the parity group number equals `seed`, which must not exceed 64 bits;

    - the intention is that if it is known that some parts of keys of a particular index have good statistical properties, *e.g.*, are generated as a sequential counter, these parts of the key can be included in the identity mask of a non-hashed layout. In addition, some parts of a key can be omitted from the identity mask to improve locality of reference, so that "close" keys are stored in the same component catalogue, increasing the possibility of network aggregation. Note that a user can always use a hash function tailored for a particular index by appending arbitrary hash values to the keys.

A few special cases require special mention:

- redundant, but not striped layout is a layout with the empty identity mask. In an index with such layout, all records belong to the same parity group. As a result, the index is stored in (K+1) component catalogues. The location of the next record is known in advance and iteration through the index can be implemented without broadcasting all component catalogues. The layout provides fault-tolerance, but doesn't provide full scalability within a single index, specifically the total size of an index is bound by the size of the storage controlled by a single catalogue service. Note, however that different indices with the same layout will use different sets of services;
- fully hashed layout is a `layout with infinite identity mask [0, +∞]` and with "hashed" attribute true. Records of an index with such layout are uniformly distributed across the entire pool. This layout is the default layout for "generic" indices;
- fid index layout. It is expected that there will be many indices using fids as keys. The default hash function should work effectively in this case. Similarly for the case of an index, where a 64-bit unsigned integer is used as a key;
- *lingua franca* layout is the layout type optimised for storing lingua franca namespace indices, by hashing filename part of the key and omitting attributes from the hash.

Layout descriptor is the set of parameters necessary to do index operations. Layout descriptor consists of:

- storage pool version fid. Component catalogues of an index using the layout are stored in the pool version. Pool version object is stored in confc and contains, among other attributes, N, K, and P;

- identity mask, as described above;
- hashed flag, as described above (or the identifier of a hash function to use for this layout, if multiple hash functions are supported);
- for uniformity, layout descriptors are also defined for catalogues (*i.e.*, non-distributed indices). A catalogue layout descriptor consists of the fid of the service hosting the catalogue.

Typically a layout descriptor will be shared by a large number of indices. To reduce the amount of meta-data, a level of indirection is introduced, see the Internal meta-data sub-section below.

In-memory representation of a clovis index includes index fid and index layout descriptor.

Internal meta-data

Index sub-system maintains some internal meta-data summarised in the table below.

Commented [2]: reference for max, dima, me.
root:
 - 'index name' -> <index FID, index layout descriptor>

layout:
 - index FID -> layout identifier (uint64_t) | layout DESC

layout-descr:
 - layout identifier (uint64_t) -> layout DESC

name	purpose	key	value	layout	access
root	top-level index to find all other indices	string label of a global index, *e.g.*, "layout"	global piece of meta-data	hard-coded	can be accessed immedia *via* the hard-coded layout
layout-descr	holds layout descriptors of "normal" indices	64-bit layout identifier	layout descriptor (plus, maybe a reference counter)	stored in the root index	*via* root, key "layout-descr"
layout	maps indices to layouts	index fid	layout identifier, or layout descriptor (union)	stored in the root index	*via* root, key "layout"
catalogue-index	a catalogue created locally by each cas to map indices to component catalogues	index fid	catalogue fid and index layout descriptor	a local catalogue, doesn't need a layout	a record is added to catalogue-index whenever a component catalogue is created by the cas

The root index is intended to be a small index containing some small number of rarely updated global meta-data. As the root index is small and rarely updated it can be stored in highly replicated default pool (specified in confd), that can remain unchanged as system configuration changes over time.

Layout and layout-descr indices collectively provide layouts to indices. Separation between layout and layout-desc allows layout descriptors to be shared between indices. A record in the layout index can contain as value either a layout identifier (usual case) or full layout descriptor (special case). Because layout-descr and especially layout indices can grow very large, it is not possible to store them once and for all in the original default pool. Instead, the layout descriptors of the layout and layout-descr indices are stored in the root index. When system grows layout index can be migrated to a larger pool and its layout descriptor in the root index updated. Layout and layout-descr indices will be used, in the future, to store layouts of data objects.

A catalogue-index is a local (non-distributed) catalogue maintained by the index sub-system on each node in the pool. When a component catalogue is created for a distributed index, a record mapping the catalogue to the index is inserted in the catalogue-index. This record is used by the index repair and re-balance to find locations of other replicas.

Client

Commented [3]: dix_request lifecycle?

Initialisation:

- find default index pool in confc
- construct root index layout descriptor
- fetch layout and layout-descr layouts from the root index

Index creation:

- construct layout descriptor
- cas-client: send CREATE to all casses holding the component catalogues

Index open:

- cas-client: look up in layout, given index fid, get layout-id or layout descriptor
- if got identifier, lookup descriptor in layout-descr

Index operation (get, put, next):

- use layout descriptor to find component catalogues
- cas-client: operate on the component catalogues

Operation concurrent with repair or re-balance:

- use spare components;
- for PUT, use *overwrite* flag (see below), when updating the spare;
- for re-balance, update correct replicas, spares and re-balance target (use *overwrite*

flag);

- for DEL, delete from spares, re-balance target and correct replicas;
- DEL is 2 phase:
  - use cas-client to update correct replicas, get reply
  - use cas-client to update spares and re-balance target

  This avoids a possible race, where repair send old value to the spares concurrently with a client update.

Service

Catalogue service (cas) implementation is extended in the following ways:

- a record is inserted in the meta-index, when a component catalogue is created. The key is the catalogue fid, the value is `(tree, index-fid, pos, layout-descr)`, where
  - `tree` is the b-tree as for a catalogue,
  - `index-fid` is the fid of the index this catalogue is a component of,
  - `pos` is the position of the catalogue within the index layout, from 0 to P;
  - `layout-descr` is the layout descriptor of the index;
- values in the meta-index can be distinguished by their size;
- when a catalogue with the fid `cat-fid` is created as a component of an index with the fid `idx-fid`, the record `(key: idx-fid, val: cat-fid)` is inserted in the catalogue-index;
- NEXT operation accepts a flag parameter (*slant*), which allows iteration to start with the smallest key following the start key;
- PUT operation accepts a flag (*create*) instructing it to be a no-op if the record with the given key already exists;
- PUT operation accepts a flag (*overwrite*) instructing it to silently overwrite existing record with the same key, if any;
- before executing operations on component catalogues, cas checks that the index fid and layout descriptor, supplied in the fop match contents of the meta-index record.

Repair

Commented [4]: What is it for? Why can't we store all its values in catalogue-index? The relation seems to be 1-to-1.

Index repair service is started along with every cas, similarly to SNS repair service being started along with every ios.

When index repair is activated (by Halon by using spiel), index repair service goes through catalogue-index catalogue in index fid order. For each index, repair fetches layout descriptor from the meta-index, uses it to calculate the spare space location and invokes cas-client to do the copy. The copy should be done with the *create* flag to preserve updates to spares made the clients.

Commented [5]: Update documentation according to repair/rebalance context everywhere, inside HLD, DLD, code.

Re-balance

Similar to repair.

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.]

- cas: add "flags" field to cas record structure, with the following bits:
    - *slant*: allows NEXT operation to start with a non-existent key;
    - *overwrite*: PUT operation discards existing value of the key;
    - *create*: PUT operation is a successful no-op if the key already exists;
- conf: assign devices to cas services (necessary to control repair and re-balance)
- spiel: support for repair and re-balance of indices
- halon: interact with catalogue services (similarly to io services)
- halon: introduce "global mkfs" phase of cluster initialisation, use it to call a script to create global meta-data

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.]

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.]

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.]

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.]

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.]

Implementation plan

- implement basic pdclust math, hashing, identity mask;
- implement layout descriptors in memory
- implement subset of clovis sufficient to access the root index, *i.e.*, without
    - fetching layouts from network
    - catalogue-index
- add unit tests, working with the root index
- implement layout index and layout-descr index
- more UT
- system tests?
- implement catalogue-index
- modify conf schema to record devices used by cas
    - deployment?
- implement index repair
- implement index re-balance
- halon controls for repair and re-balance
- system tests with halon, repair and re-balance
- implement concurrent repair, re-balance and index access
- system tests (acceptance, S3)

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.]

HLD of catalogue service
HLD of parity de-clustered algorithm
HLD of SNS repair