High level design of Auxiliary Databases for SNS repair

By Carl Braganza <carl_braganza@xyratex.com>
Date: 2011/08/18
Revision: 1.0

This document presents a high level design (HLD) of the auxiliary databases required for SNS repair. 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

SNS repair requires additional meta-data not normally needed for normal ioservice operations. ~~SNS repair requires the ability to locate the *cob_fid* for a given *file_fid* on a given device, and to iterate over all *cob_fids* in *file_fid* order.~~ SNS repair requires the ability to iterate over the *cob_fids* on a given *device*, ordered by their associated *file_fid*.

A more complete description was provided by Nikita Danilov in [1]:

> Formally, this task is to implement any additional meta-data tables that are not needed for normal ioservice operation, but required for SNS repair. At the moment it seems that a single table is necessary. SNS repair proceeds in the *file* fid (also referred to as *global object* fid) order and by a parity group number (that is, effectively in a logical file offset order) within a file. For its normal operation ioservice requires no knowledge of files at all, because it works with *component objects* (cob-s), the translation between user visible (file, offset) *coördinates* to (cob, offset) coördinates is performed by the client through layout mapping function.
>
> The additional table should have (device id, file fid) as the key and cob fid as the corresponding record. The represented (device, file)->cob mapping is 1:1.
>
> The task should define the table and provide wrapper functions to inserts and delete (device_id, file_fid, cob_fid) pairs, lookup cob-fid by device-id and file-fid, and iterate over the table in file-fid order for a given device. In the future, device-id will be generalised to container-id.
>
> This table is used by the object creation path, executed on every ioservice when a file is created. A new file->cob fid mapping is installed at this point. The mapping is deleted by file deletion path (not existing at the moment). During SNS repair it is used by storage agents associated with storage devices. Each agent iterates over file-fid->cob-fid mapping for its device, selecting the next cob to process.

1. Definitions

One new definition is introduced:

- A *cobfid map* is a persistent data structure that tracks the id of *cobs* and their associated *file fid,* contained within other containers, such as a storage object.

Some relevant definitions from related documents are repeated here for convenience:

- A *storage object* (stob) is a basic C2 data structure containing raw data. [3]
- A *component object* (cob) is a component (stripe) of a file, referencing a single storage object and containing metadata describing the object. [3]
- A *global object* (gob) is an object describing a striped file, by referring to a collection of

component objects. [3]

- storage *devices* are attached to data servers; [2]
- *storage objects* provide access to storage device contents by means of a linear name-space associated with an object; [2]
- some of the objects are *containers*, capable of storing other objects. Containers are organized into a hierarchy; [2]
- a *cluster-wide object* is an array of bytes (the object's linear name-space, called cluster-wide object data) and accompanying meta-data, stored in containers and accessed through at least read and write operations (and potentially other operations of POSIX or similar interface). Index in this array is called an offset; A cluster-wide object can appear as a file if it is visible in file system namespace. [2]
- a cluster-wide object is uniquely identified by an identifier, called a *fid*; [2]
- a *parity group* is a collection of data units and their parity units. [2]

2. Requirements

- [**r.container.enumerate**]: it is possible to efficiently iterate through the containers stored (at the moment) on a given storage device.

  This requirement is mentioned as a dependency in [2].

- [**r.container.enumerate.order.fid**]: it is possible to efficiently iterate through the containers stored on a given storage device in priority order of global object fid. This is a special case of [**r.container.enumerate**] with a specific type of ordering.
- [**r.generic-container.enumerate.order.fid**]: it is possible to efficiently iterate through the containers stored in another container in priority order of global object fid. This extends the previous requirement, [**r.container.enumerate.order.fid**], to support enumeration of the contents of arbitrary types of containers, not just storage devices.

3. Design highlights

- A cobfid map is implemented with the C2 database interface which is actually based upon key-value tables implemented using the Oracle Berkeley Database, with one such "table" or "database" per file.
- Interfaces to add and delete such entries are provided for both devices and generic containers.
- Interfaces to iterate through the contents of a device or generic container are provided.

4. Functional specification

Data types

The following data types are used in the interfaces but are not defined by the interfaces:

- Device identifier (uint64_t)
- File fid (global object fid) (struct c2_fid, 128 bit)
- Cob file identifier (struct c2_fid or stobid, both 128 bit)
- Container identifier (generalization of device identifier) (uint64_t)

All these are globally unique integer data types of possibly varying lengths. This specification does not address how these identifiers are created.

The interface defines a structure to represent the cobfid map.

In all cases the invoker of the interface supplies a database environment pointer.

Interfaces

Initialization and termination interfaces

- Create and/or prepare a named cobfid map for use.
- Properly finalize use of the cobfid map.

The same cobfid map can be used for both devices and general containers, though the invoker will have the ability to use separate databases if desired.

Device content tracking

- Associate with a given device (device-id), contents identified by the 2-tuple (cob-fid-id, file-fid), in the cobfid map. The 3-tuple of (device-id, cob-fid, file-fid) is defined to be unique.
- Lookup the cob-fid given the 2-tuple of (device-id, file-fid)
- Remove the 3-tuple (device-id, file-fid, cob-fid) from the cobfid map.
- Enumerate the cob-fids contained within a given device (identified by device-id) in the cobfid map, sorted by ascending order of the file-fid associated with each cob-fid.

  The interface should require a user specified buffer in which to return an array of cob-fids. The trade-offs here are:

  - The time the database transaction lock protecting the cobfid map file is held while the contents are being enumerated
  - The amount of space required to hold the results.

  An implementation could choose to return a unique error code and the actual number of entries in case the buffer is too small, or it could choose to balance both of the above trade-offs by returning data in batches, which each call continuing with next possible file-fid in sequence.

Logical container content tracking

General container tracking will work with the same interface as a device is a special type of container. A general container identifier, a 64 bit unsigned integer, can be provided wherever device-id is mentioned in this specification.

> *The implementation is free to use the more generic term container_id instead of device_id in the interface.*

If general container identifiers may clash with device identifiers, then the invoker has the ability to create separate cobfid maps for each type of object.

Recovery interfaces

An implementation should provide interfaces to aid in the recovery of the map in case of corruption or loss. These interfaces will be required by [Dependency: **r.container.recovery**] and would possibly include

- An interface to determine if the map is corrupt or otherwise irrecoverable
- An interface to initiate the periodic check-pointing of the map
- An interface to restore the map

5. Logical specification

Schema

C2 database tables are really key-value associations, each represented by a `c2_db_pair` structure. A suitable key for device content tracking and the generalized logical container content tracking would contain two fields: (container-id, file-fid), where container-id is a 64 bit unsigned integer. The associated value would contain just

one field, the cob-fid.

Persistent store

The name of the database file containing the cobfid map is supplied during initialization. The current C2 database interface creates files on disk for the tables, but future implementations may use different mechanisms. The location of this disk file is not defined by this document, other than requiring it to be in a "standard" location for Colibri servers. [Dependency: **r.container.enumerate.map-location**]

The invoker can chose to mix both device and container mappings in the same cobfid map, or use separately named maps for each type. The latter would be necessary in case device identifiers could clash with generic container identifiers.

The cobfid map contains information that is critical during repair; a mechanism is required to recover this map should it ever get corrupted or lost. [Dependency: **r.cobfid-map.recovery**]

Insert and delete operations

Insertion involves a `c2_table_insert()` operation or a `c2_table_update()` operation if the record already exists. It is assumed that a cob-fid will be used in a single mapping. [Dependency: **r.cob-fid.usage.unique**]

Deletion of a record involves a `c2_table_delete()` operation. The specific mapping of (device-id, file-fid) to cob-id will be deleted.

Enumeration operation

This will be implemented using the `c2_db_cursor` interfaces. The sequence of operation is as follows:

1. Create a cursor using `c2_db_cursor_init()`.
2. Create an initial positioning key value with the desired device-id and a file fid value of 0, and invoke `c2_db_cursor_get()` to set the initial position and get the first key/value record. This works because this subroutine sets the `DB_SET_RANGE` flag internally, which causes a greater-than-equal-to comparison of the key value when positioning the cursor.
   Note that this does not make 0 an invalid file-fid value.
3. Subsequent records are fetched using c`2_db_cursor_next()`.
4. Traversal ends if at any time the device-id component of the returned key changes from the desired device-id, or we've exhausted all records in the database (`DB_NOTFOUND => -ENOENT`).

As a transaction must be held across the cursor use, the interface will require that the invoker supply a buffer (an array of cob-fid ids) to be filled in by the operation. If the array is too small, the interface should return a distinct error code, and the count of the number of entries found, or return data in batches. If the latter mechanism is used, some contextual data structure should be returned to track the current position.

Recovery

Recovery mechanisms are beyond the scope of this HLD at this time, but they are required by this HLD. [Dependency: **r.cobfid-map.recovery**]

Possible mechanisms could include (but are not limted to)

- Maintaining multiple copies of the map (directly or indirectly via file system level redundancy).
- Recreating the map from meta data stored with the layout manager.
- Periodically saving the map data in the configuration database, and recovering it upon failure. The map data is not expected to change often, relative to the rate of file data I/O.

5.1. Conformance

- [**i.container.enumerate**]: The design provides the means to iterate through the cobs stored in a device or container.
- [**i.container.enumerate.order.fid**]: The iteration would be ordered by file fid.
- [**i.generic-container.enumerate.order.fid**]: Interfaces are provided for generic containers identifiers too.

5.2. Dependencies

- [**r.cobfid-map.recovery**] There must be a mechanism to recover the cobfid map in case it gets corrupted or otherwise rendered inaccessible. This may involve other Colibri components, including those off-host to the IO service.
- [**r.cob-fid.usage.unique**] The mapping of (container-id, file-id) to cob-fid must be unique. This is the responsibility of external components that drive the ioservice in its use of the interfaces described in this document.

5.3. Security model

No special concerns arise here. The mapping file must be protected like any other C2 database.

5.4. Refinement

- [**r.container.enumerate.map-location**] The location of the database file(s) containing the cobfid map(s) remains to be defined by the implementation.

6. State

6.1. States, events, transitions

6.2. State invariants

6.3. Concurrency control

1. The application is responsible for synchronization during creation and finalization of the map.
2. The C2 database operations provide thread safe access to the database.
3. Enumeration represents a case where an application may hold the database transaction for a relatively lengthy period of time. It would be up to the application to minimize the impact by saving off the returned cob-fids for later processing out of this critical section.

7. Use cases

7.1. Scenarios

1. The ioservice creates the cobfid map upon start up, and finalizes it upon termination. During normal operation, it inserts and/or deletes associations into this index as storage for files is allocated or deallocated.
2. During SNS repair a storage-in agent would use this map to drive its operation. See the storage agent algorithm in [2].

7.2. Failures

It is required that the map be recovered if corrupted or lost. [Dependency: **r.cobfid-map.recovery**]

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.]

Normal operation of the ioservice would involve inserting and deleting records when files are created, extended or shrunk and deleted, which is not very often, relative to normal data I/O access. The amount of contention depends upon how concurrent is the ioservice run-time, and the ability to scale depends upon the efficiency of the underlying database engine.

The storage-in agent would necessarily interfere with ongoing activity because it performs traversals. If, however, it minimizes the time spent holding the database lock, then the interference will not be significant.

8.2. Other

8.2. Rationale

9. Deployment

9.1. Compatibility

9.1.1. Network

9.1.2. Persistent storage

The cobfid map is stored in a C2 database on disk.

9.1.3. Core

9.2. Installation

10. References

[1] T1 Task Definitions
[2] HLD of SNS Repair
[3] HLD of Colibri Object Index