High level design of a Colibri Object Index

By Nathan Rutman <nathan_rutman@xyratex.com>
Date: 2010/10/19
Revision: 1.0

This document presents a high level design (HLD) of an Object Index for Colibri C2 core. 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

The following color marking is used in this document: incomplete or todo item, possible design extension or future directions.]

The object index performs the function of a metadata layer on top of C2 storage objects. A C2 storage object (*stob*) is a flat address space where one can read from or write on, with block size granularity. Stobs have no metadata associated with them. To be used as files (or components of files, aka stripes), additional metadata must be associated with the stobs:

- namespace information: parent object id, name, links
- file attributes: owner/mode/group, size, m/a/ctime, acls
- fol reference information: log sequence number (*lsn*), version counter

Metadata must be associated with both *component objects (cobs)*, and *global objects*. Global objects would be files striped across multiple component objects. Ideally global objects and component objects should reuse the same metadata design (a cob can be treated as a gob with a local layout).

1. Definitions

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

2. Requirements

The following requirements from the Summary requirements table are relevant to Object Index:

- [R.C2.BACK-END.OBJECT-INDEX]: an object index allows back-end to locate an object by its fid
- [R.C2.BACK-END.INDEXING]: back-end has mechanisms to build meta-data indices
- [R.C2.LAYOUT.BY-REFERENCE]: file layouts are stored in file attributes by reference
- [R.C2.BACK-END.FAST-STAT]: back-end data-structures are optimized to make stat(2) call fast
- [R.C2.DIR.READDIR.ATTR]: readdir should be able to return file attributes without additional IO
- [R.C2.FOL.UNDO]: FOL can be used for undo-redo recovery
- [R.C2.CACHE.MD]: meta-data caching is supported

3. Design highlights

- The file operation log will reference particular *versions* of cobs (or gobs). The version information enables undo and redo of file operations.
- cob metadata will be stored in database tables.
- The database tables will be stored persistently in a metadata container.
- There may be multiple *cob domains* with distinct tables inside a single container.

4. Functional specification

Cob code:

- provides access to file metadata via fid lookup
- provides access to file metadata via namespace lookup
- organizes metadata for efficient filesystem usage (esp. stat() calls)
- allows creation and destruction of component objects
- facilitates metadata modification under a user-provided transaction

5. Logical specification

5.a Structures

Three database tables are used to capture cob metadata:

- object-index table
  - key is {child_fid, link_index} pair
  - record is {parent_fid, filename}
- namespace table
  - key is {parent_fid, filename}
  - record is {child_fid, nlink, attrs}
  - if nlink > 0, attrs = {size, mactime, omg_ref, nlink}; else attrs = {}
  - multiple keys may point to the same record for hardlinks, if the database can support this. Otherwise, we store the attrs in one of the records only (link number 0). (This leads to a long sequence of operations to delete a hardlink, but straightforward.)
- fileattr_basic table
  - key is {child_fid}
  - record is {layout_ref, version, lsn, acl} (version and lsn are updated at every fop involving this fid)

link_index is an ordinal number distinguishing between hardlinks of the same fid. E.g. file a/b with fid 3 has a hardlink c/d. In the object index table, the key {3,0} refers to a/b, and {3,1} refers to c/d.
omg_ref and layout_ref refer to common owner/mode/group settings and layout definitions; these will frequently be cached in-memory and referenced by cobs in a many-to-one manner. Exact specification of these is beyond the scope of this document.

References to the database tables are stored in a cob_domain in-memory structure. The database

contents are stored persistently in a metadata container.

There may be multiple cob_domains within a metadata container, but the usual case will be 1 cob_domain per container. A cob_domain may be identified by an ordinal index inside a container. The list of domains will be created at container ingest.

```
struct c2_cob_domain {
        cob_domain_id  cd_id   /* domain identifier  */
            c2_list_link    cd_domain_linkage
            c2_dbenv     *cd_dbenv
            c2_table       *cd_obj_idx_table
            c2_table       *cd_namespace_table
            c2_table       *cd_file-attr-basic_table
            c2_addb_ctx  cd_addb

}
```

A c2_cob is an in-memory structure, instantiated by the method cob_find and populated as needed from the above database tables. The c2_cob may be cached and should be protected by a lock.

```
struct c2_cob {
            fid                  co_fid;
            c2_ref               co_ref;     /* refcounter for caching cobs */
        struct c2_stob   *co_stob;    /* underlying storage object */
            struct c2_rwlock co_guard;  /* lock on cob manipulation */
        c2_fol_obj_ref    co_lsn;
        u64                  co_version
        struct namespace_rec    *co_ns_rec;
            struct fileattr_basic_rec *co_fab_rec;
            struct object_index_rec *co_oi_rec;             /* pfid, filename */
```

};The *_rec members are pointers to the records from the database tables. These records may or may not be populated at various stages in cob life.

The co_stob reference is also likely to remain unset, as metadata operations will not frequently affect the underlying storage object (and, indeed, the storage object is likely to live on a different node).

5.b Usage

c2_cob_domain methods locate the database tables associated with a container. These methods are called at container discovery/setup.

c2_cob methods are used to create, find, and destroy in-memory and on-disk cobs. These might be:

- cob_locate: find an object via a fid using the object_index table.
- cob_lookup: find an object via a namespace lookup (namespace table).
- cob_create: add a new cob to the cob_domain namespace
- cob_remove: remove the object from the namespace
- cob_get/put: take references on the cob. At last put cob may be destroyed.

c2_cob_domain methods are limited to initial setup and cleanup functions, and are called during

container setup/cleanup.

Simple mapping functions from the fid to stob:so_id and to the cob_domain:cd_id are assumed to be available.

5.1. Conformance

- [I.C2.BACK-END.OBJECT-INDEX]: object-index table facilitates lookup by fid
- [I.C2.BACK-END.INDEXING]: new namespace entries are added to the db table
- [I.C2.LAYOUT.BY-REFERENCE]: layouts are referenced by layout ID in fileattr_basic table.
- [I.C2.BACK-END.FAST-STAT]: stat data is stored adjacent to namespace record in namespace table.
- [I.C2.DIR.READDIR.ATTR]: namespace table contains attrs
- [I.C2.FOL.UNDO]: versions and lsn's are stored with metadata for recovery
- [I.C2.CACHE.MD]: c2_cob is refcounted and locked

5.2. Dependencies

- [R.C2.FID.UNIQUE]: uses; fids can be used to uniquely identify a stob
- [R.C2.CONTAINER.FID]: uses; fids indentify the cob_domain via the container
- [R.C2.LAYOUT.LAYID]: uses; reference stored in fileattr_basic table

5.3. Security model

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

5.4. Refinement

[This sub-section enumerates design level requirements introduces 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

Scenario	QA.schema.op
Relevant quality attributes	variability, re-usability, flexibility, modifiability
Stimulus	a Request Handler invokes back-end as part of file system operation processing
Stimulus source	a file system operation request originating from protocol translator, native C2 client or storage application
Environment	normal operation
Artifact	a series of Schema accesses
Response	Meta-data back-end contains enough information to handle file system operation request. This information includes: • standard file attributes as defined by POSIX, including access control related information; • description of file system name-space, including directory structure, hard-links and symbolic links; • references to remote parts of file-system namespace; • file data allocation information
Response Measure	
Questions and issues	

Scenario	QA.schema.stat
Relevant quality attributes	usability
Stimulus	a stat(2) request arrives to a Request Handler
Stimulus source	a user application
Environment	normal operation
Artifact	a back-end query to locate the file and fetch its basic attributes
Response	Schema must be structured so that stat(2) processing can be done quickly without extract index lookups and associated storage accesses
Response Measure	• an average number of schema operations necessary to complete stat(2) processing; • an average number of storage accesses during stat(2) processing
Questions and issues	

Scenario	QA.schema.duplicates
Relevant quality attributes	usability
Stimulus	a new file is created
Stimulus source	protocol translator, native C2 client or storage application
Environment	normal operation
Artifact	a records, describing new file are inserted in various schema indices
Response	records must be small. Schema must exploit the fact that in a typical file system, certain sets of file attributes have much fewer different values than combinatorially possible. Such sets of attributes are stored by reference, rather than by duplicating the same values in multiple records. Examples of such sets of attributes are: • {file owner, file group, permission bits} • {access control list} • {file layout formula}
Response Measure	• average size of data that is added to the indices as a result of file creation • attribute and attribute set sharing ratio
Questions and issues	

Scenario	QA.schema.simple
Relevant quality attributes	re-usability, variability
Stimulus	
Stimulus source	
Environment	
Artifact	
Response	Schema can be described and implemented in terms of a limited repertoire of standard operations: • index lookup, • index modification, • index iteration Assuming fairly standard transactional capabilities and usual locking primitives
Response Measure	
Questions and issues	

Scenario	QA.schema.index
Relevant quality attributes	variability, extensibility, re-usability
Stimulus	storage application wants to maintain additional meta-data index
Stimulus source	storage application
Environment	normal operation
Artifact	index creation operation
Response	schema allows dynamic index creation

Response Measure

Questions and issues

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

[1] Summary requirements table
[2] C2 glossary

11. Inspection process data

11.1. Logt

	Task	Phase	Part	Date	Planned time (min.)	Actual time (min.)	Comments
N. Rutman	object-index	HLD	1		180		
Huang Hua		HLDINSP	1	2010-10-28	180	180	inline

11.2. Logd

No.	Task	Summary	Reported by	Date reported	Comments
1					