# High level design of a layout schema

By Nikita Danilov <nikita_danilov@xyratex.com>
Huang Hua <hua_huang@xyratex.com>
Date: 2011/07/28
Revision: 0.1

[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 a layout schema of 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

Very broadly, a layout is something that determines where a particular piece of data or meta-data is located in a storage system, including location of data and meta-data that are about to be added to the system. The layout is by itself a piece of meta-data and has to be stored somewhere. This HLD of layout schema is to design the organization of the layout meta-data stored in database.

## 1. Definitions

- A *layout* is a map determining where file data and meta-data are located. The layout is by itself a piece of meta-data and has to be stored somewhere.
- A layout is identified by *layout identifier* uniquely.
- This *map* is composed of *sub-maps*, and a sub-map may still be composed of sub-maps, till direct mapping to some known underlying location, i.e. block number on physical device.
- A layout *schema* is a way to store the layout information in data base. The schema describes the organization for the layout meta-data.

## 2. Requirements

- [r.layout.schema.layid] layout identifiers. Layout identifiers are unique globally in the system, and persistent in the life cycle.
- [r.layout.schema.types] multiple layout types. There are multiple layout types for different purposes: SNS, block map, local raid, de-dup, encryption, compression, etc.
- [r.layout.schema.formulae] layout formulae (future)
  - parameters. Layout may contain sub-map information. Layout may contain some formula, and its parameters and real mapping information should be calculated from the formula and its parameters.
  - garbage collection. If some objects are deleted from the system, their associated layout may still be left in the system, with zero reference count. This layout can be re-used, or be garbage collected in some time.
- [r.layout.schema.sub-layouts] sub-layouts (future). ?

## 3. Design highlights

Lustre stores layout (LOVEA) as an extended attribute of file. Every file has its own layout stored in EA, even though they have similar striping pattern. This does not only waste precious meta-data storage, but also impact performance, because to access a file, a separate EA has to be loaded from disk and network. In Colibri, layout is a meta-data that is stored separately. It can be transferred from meta-data server to clients or other servers. It is used to locate file data or meta-data according to the offset of desired data. Layout in Colibri are generalized to locate data and meta-data for all objects: file, dir, encrypted file, compressed file, de-dup files, etc.

A layout describes a mapping in term of sub-maps. A sub-map can be specified in a one of the following ways:

- a layout id (layid) of a layout that implements a sub-map;

- a directly embedded layout that implements a sub-map;
- a fid of a file (a component file) whose file layout implements a sub-map;
- a storage address (a block number usually).

The layout schema is to organize these layout data structures in memory and in database and store them in database.

## 4. Functional specification

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

### 4.1 Layout types

Layout is used to locate data and meta-data for various types of files/objects in Colibri. The following layout will be defined:

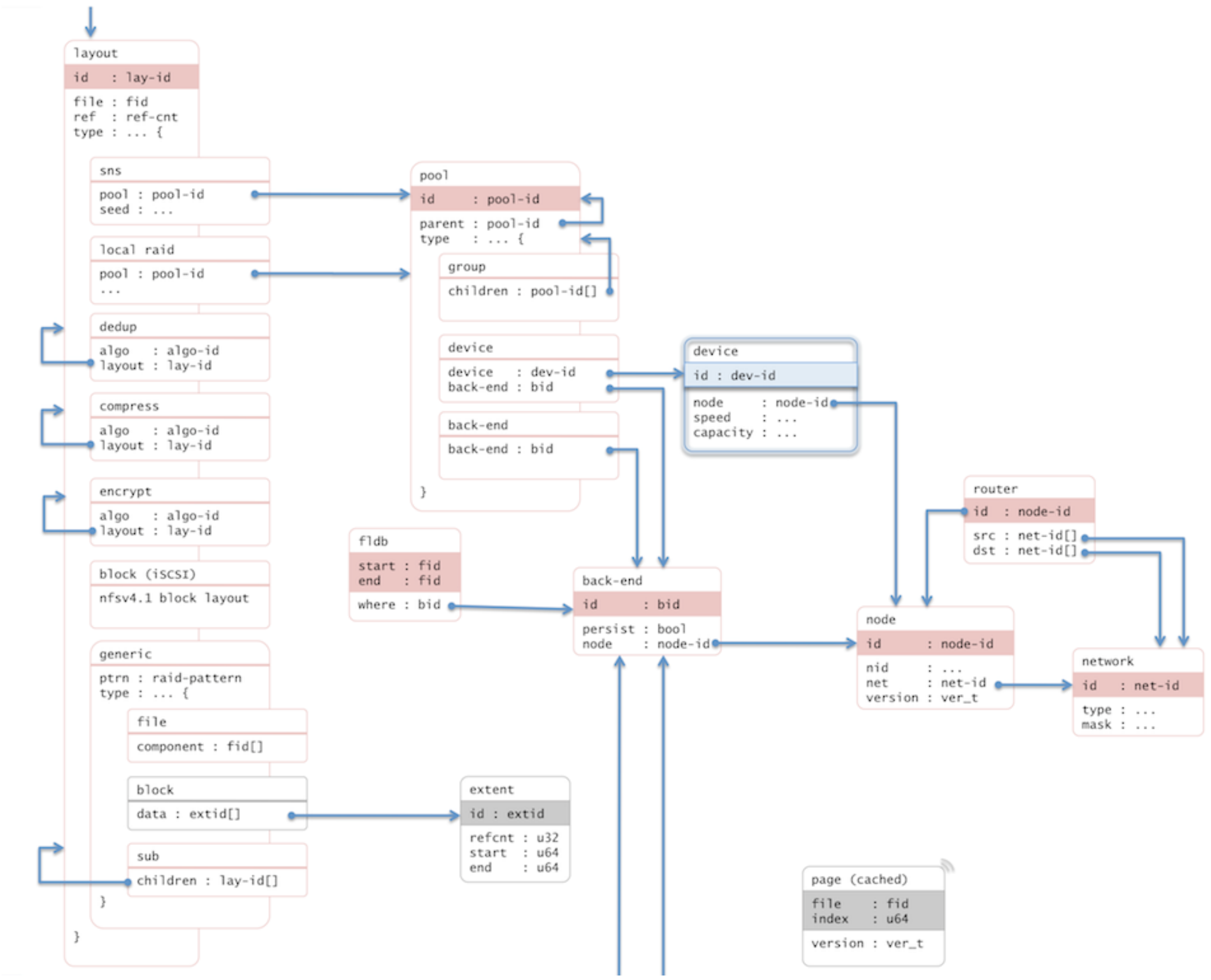
- SNS. File data or metadata will be striped over multiple devices within a pool.
- local raid. Device data will span over multiple local disks.
- Generic. File data or meta-data is striped with some pre-defined RAID pattern. File is striped within component file; block device is striped over extents; And also the sub-map can be another layout.
- Other types of layout are also supported( in the future), such as de-dup, encryption, compression.

### 4.2 Layout hierarchy

Layout is organized as sub-maps in a hierarchy. The sub-map should be resolved until some preliminary address (e.g. block number of physical device) is reached.

### 4.3 Layout schema

Layout schema is the organization of layout data structures used for database storing. The following figure depicts the schema.



### 4.4 Layout owner and references

Layout is assigned to some Colibri object, that is its owner. Layout is stored separated from its owner's meta-data. Sometimes multiple objects share the same layout, .e.g. in de-dup, or snapshots. Layout has persistent reference count to address this requirement.

### 4.5 Layout operations

Layout will support the following operations:

- Layout creation. This will create a layout and store it in database.
- Layout update. This will update an existing layout of its content, e.g. its sub maps.
- Layout query. This will retrieve layout content by its layout id.
- Layout references get/put. This will increase/decrease reference count for layout.
- Layout deletion. This will delete a layout specified by its layout id.

### 4.6 Layout policy

Layout is created for new file when new file is created. Layout is updated for file when a file is in NBA mode, or is compressed, or de-dup, etc. Layout policy makes the decision on how to choose a suitable layout. Layout policy is affected by:

- SNS[2] arguments: stripe size, stride size, stripe count, etc. This may derived from system default, or from parent directory, or specified explicitly upon creation.
- system running state. This include system load, disk/container usage, quota, etc.
- PDRAID requirements. This insures the data availablity in face of failure, I/O performance.

On principle, this topic is not covered by this task. It will be discussed by layout policy task in the future.

## 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.]

A layout is a map determining where file data and meta-data are located. This map is composed of sub-maps. The layout is by itself a piece of meta-data and has to be stored somewhere. The naïve solution (used by Lustre) is to store a layout for a file as a file attribute. This leaves much to be desired, because a large number of files usually have very similar layouts, wasting precious attribute space. Instead of storing every file layout individually, a layout can be parametrized to describe whole group of file layouts, so that an individual file layout can be obtained by substituting a parameter. As an example, instead of file layouts "a file RAID0 striped across component files with fids A0, ..., AN" and "a file RAID0 striped across component files with fids B0, ..., BN" a parametrized layout "a file RAID0 striped across component files with fids F + i * STEP, i = 0, ..., N" can be used. A particular file layout can be obtained by substituting proper values of F and STEP---a representation much more compact than original file layouts.

A layout describes a mapping in term of sub-maps. A sub-map can be specified in a one of the following ways:

- a layout id (layid) of a layout that implements a sub-map;
- a directly embedded layout that implements a sub-map;
- a fid of a file (a component file) whose file layout implements a sub-map;
- a storage address (a block number usually).

SNS and local raid layouts include a number of sub-maps, indeed a rather large number in case of wide-striping. Moreover, an overwhelming majority of files in a typical system would have such layouts. This makes an ability to represent a collection of sub-maps compactly an important optimization. To achieve it, two types of sub-map collections are introduced:

- a regular collection is one where all sub-maps belong to the same type (*i.e.*, all are given as fids, or all are given as layids, *etc.*), and the identifier of i-th member of collection is (I0 + STEP*i). Such a collection can be represented by a triple (I0, STEP, N), where N is a number of elements in the collection;
- a collection is irregular otherwise.

To support NBA, mixed layout is needed. In a mixed layout, the original layout is kept unchanged, but some part of it is superseded by the new layout.

The simplest layout schema is to implement a regular collection. In this task, we will first implement the following sub-maps:

- a fid of a file.

### 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

Colibri db interfaces is a dependency. This is already implemented.

### 5.3. Security model

N/A

### 5.4. Refinement

N/A

## 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.]

## 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] On layouts
[1] Meta-data schema
[2] HLD of SNS Repair