# High level design of a fop object iterator

By Nikita Danilov <nikita_danilov@xyratex.com>
Date: 2010/11/17
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 a fop object iterator. 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

[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.]

This component introduces an infrastructure for "generic fop methods". To recall, a fop[1] is a file operation packet, that contains fields describing a file system operation. Generic fop methods, described in the request handler HLD, allow common tasks such as authorization, authentication, resource pre-allocation, etc. to be done by the generic code (rather than in a per-fop-type way). This can be thought of as a more uniform and generic variant of `HABEO_` flags in Lustre 2.0 MDT.

Fop object iterator is a component that allows generic code to iterate over a list of file system objects affected by a fop.

## 1. Definitions

[Definitions of terms and concepts used by the design go here. The definitions must be as precise as possible. References to the C2 Glossary are permitted and encouraged. Agreed upon terminology should be incorporated in the glossary.]

- a *file operation packet* (fop) is description of file system state modification that can be passed across the network and stored in a data-base;
- a fop belongs to one of *fop types*. *E.g.*, there is a fop type for MKDIR, another for WRITE. A fop, belonging to a particular fop type is called an *instance* of this type;
- structure of data in instances of a fop type is defined by a *fop format*. A format defines the structure of instance in terms of *fop fields*. A data value in a fop instance, corresponding to a fop field in instance's format is called a *fop field instance*.
- for the purposes of the present specification, a *file system object* is something identified by a fid[2]. A fop identifies objects involved in the operation by carrying their fids as field instances. A fid

[1] [r.fop] ST

[2] [r.fid] ST

might be accompanied by a version number, specifying to which version of the object the operation is to be applied.

## 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.fop-object.reqh]: the major requirements for the fop object iterator interface is to allow object handling code to be lifted from per-fop-type code into a generic request hander part;
- [r.fop-object.batch]: fop batching[3] must be supported. For a batched fop, iterator should return the objects in component fops (recursively, if needed);
- [r.fop-object.ordering]: fop object iterator must return objects in a consistent order, so that request handler can deal with objects without risking dead-locks and without additional sorting pass.

## 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 fop format defines the structure of fields within a fop (of a given fop type). For each fop format, a list is maintained, enumerating format's fields identifying file system objects. This list is built statically when the fop format is constructed. A fop object iterator goes through this list, descending into sub-fields and sub-fops.

## 4. Functional specification

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

Fop object iterator component exports a fop object cursor data-type. A cursor is initialized for a given fop instance.

Fop object cursor is used in the following ways (among others):

[3] [r.fop.batching] ST

- by the Network Request Scheduler[4] to asynchronously read-ahead objects[5] involved into queued operations[6];

- by request handler generic code to load all the objects involved in the operation;

- by distributed transaction manager to check versions of objects against the versions specified in the fop.

To support flexible generic operations, a fop cursor returns some additional information associated with a fop field. *E.g.*, a fop object cursor returns bit-flags indicating whether the object should be checked for existence (or non-existence), whether it should be locked.

## 5. Logical specification

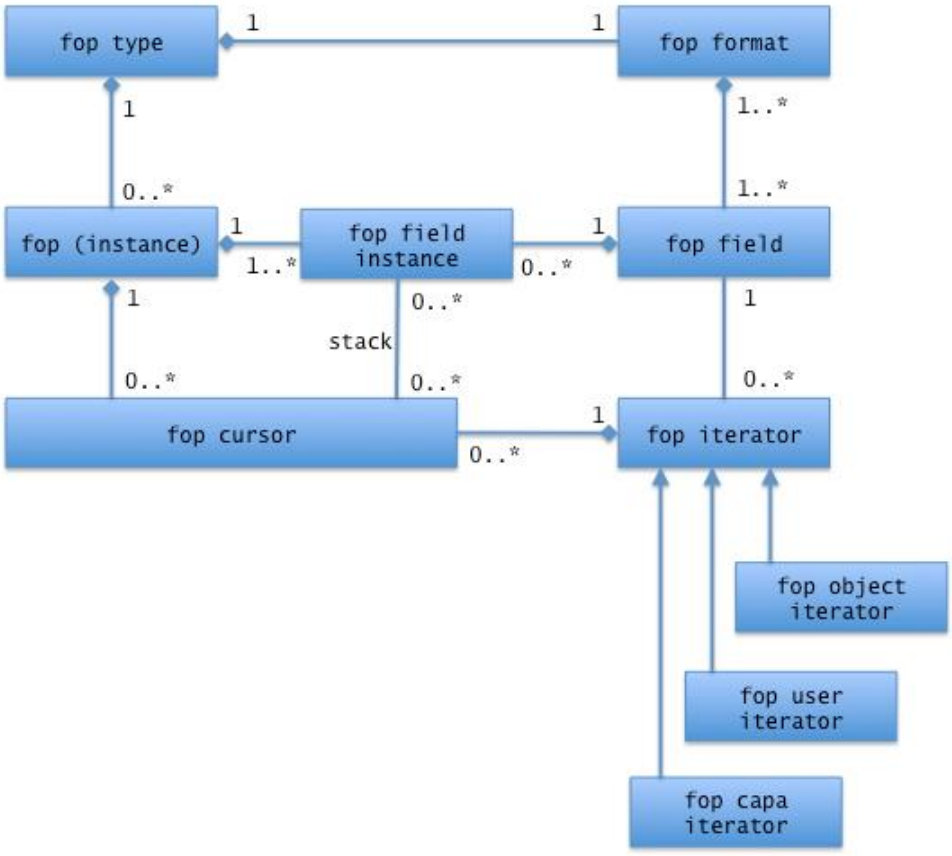
[This section defines a logical structure of the designed component: the decomposition showing *how* the functional specification is met. Subcomponents and diagrams of their interrelations should go in this section.]

This pseudo-UML diagram describes the relationships between fop iterator related entities:

[4][r.nrs] ST

[5][r.md.async]

[6][r.fop.nrs] ST




A fop object cursor keeps a stack of fop field instances, corresponding to the current cursor position in a (potentially multiply nested) batched fop.

Each fop field has associated with it an enumeration of sub-fields identifying file system objects (this enumeration is empty most of the time). To advance a cursor, the position in this enumeration for the top of the cursor field stack is advanced and the top is popped off the stack when end of the enumeration is reached.

### 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.]

- [r.fop-object.reqh]: with the help of fop object iterator, request handler can load all the objects in the generic code. With the addition of other iterators, such as fop user iterator, all authorization checks can be lifted into generic code;

- [r.fop-object.batch]: the design of fop object cursor, supports fop batching by recording a stack

of positions in a cursor;

- [r.fop-object.ordering]: the fop iterator assumes that objects are listed in a fop in the proper order[7].

### 5.2. 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.]

- fop
  - [r.fop.batching]: fops can be batched;
  - [r.fop.nrs]: fops can be used by NRS;
  - [r.fop.ordering]: fields in a fop are ordered so that fids occur in the desired order;
- fid: a file system object is identified by a fid;

### 5.3. Security model

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

This component introduces no additional security constraints.

### 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.]

The detailed level design is straight-forward. No refinement is necessary.

## 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.]

No non-trivial state transitions.

[7] [r.fop.ordering]

## 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.fop-object.reqh] |
| Relevant quality attributes | scalability |
| Stimulus | request handler receives a fop for processing |
| Stimulus source | an client request or a system call on a client |
| Environment | normal operation |
| Artifact | request handler calls fop object iterator to return a list of objects involved in the operation |
| Response | fop object iterator returns a list of objects involved in the operation. Request handler loads, locks and checks the objects before calling into fop-type specific code |
| Response measure | Object loading and checking code is re-used by all fop-types. Fop-type specific code is smaller and addition of a new fop type is simplified |
| Questions and issues | |

| | |
|---|---|
| Scenario | [usecase.fop-object.nrs] |
| Relevant quality attributes | re-usability, extensibility |
| Stimulus | NRS queues a fop in its processing queue |
| Stimulus source | a stream of incoming client request |
| Environment | normal operation of a large cluster |
| Artifact | NRS calls fop object iterator for fops in its read-ahead queue window |
| Response | fop object iterator returns a list of objects involved in the operation |
| Response measure | NRS issues asynchronous load requests for the objects meta-data |
| 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.]

Not applicable.

## 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.]

There are two obvious ways to implement iteration over fop fields:

- iterate over all field instances and skip all instances except for the required ones (*e.g.*, except for field instances representing file system objects), or
- introduce additional data, associated with a fop format, enumerating all required fields.

The first approach allows for a greater degree of code sharing between various iterators (iterators returning lists of file system objects, users, capabilities, *etc.*) because "iterate over all field instances" part is common, but has a drawback of requiring multiple passes over every field instance in a fop. Given that fop iterators are used in the request handler hot code path, the second approach was selected.

## 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.]

## 11. Inspection process data

[The tables below are filled in by design inspectors and reviewers. Measurements and defects are transferred to the appropriate project data-bases as necessary.]

### 11.1. Logt

| | Task | Phase | Part | Date | Planned time (min.) | Actual time (min.) | Comments |
|---|---|---|---|---|---|---|---|
| nathan | fop-object | HLDINSP | 1 | | 50 | 47 | hldit |
| huanghua | fop-object | HLDINSP | 1 | | 50 | 60 | good |