# I/O API HLD

Yuriy Umanets

5th October 2005

## 1 Requirements

This work requires to design an I/O API that is usable on the server and the client, and across the net possibly. It should not be an OBD API but an encapsulation of it. So API itself should unify the use of the following subsystems on client llite/lov/osc and ost/obdfilter/lvfs on server respectively, into one new API. It should not be OS dependent to allow porting with minimal changes or without changes at all. This means for instance, that all lustre components except of llite should not operate on OS specific structures or functions.

## 2 Functional specification

### 2.1 I/O types

In order to meet requirements, all involved lustre subsystems should be able to perform all kinds of IO operations used in lustre by only using this unified I/O API. It should be balanced enough to have minimal set of understandable functions.

The following I/O cases should be taken into account:

- synchronous I/O on client.
- asynchronous I/O on client.
- getting metadata from MDS server from client.
- handling client I/O requests on server side.
- llog related I/O operations.
- lvfs related I/O operations.

There will be a lot of changes to API itself. Some changes to protocol are foreseen also, however attempt to minimize them will be taken.

### 2.2 Related structures

There will the set of structures, used for abstracting OS specific structures from lustre ones. One of them is lustre page representation. It has reference to next layer page structure, set of page flags, registered completion handler, etc. It is need to have own page representation on each layer.

## 3 Use cases

There are few types of I/O which should be worked out (see below). However it does not mean that for all of them special API should exist. They all may be handled by the same byte oriented API (see below), which has *read()* and *write()* methods with bunch of params (to meet all requirements) and all the playing with OS pages, etc. is hidden inside.

### 3.1 Synchronous I/O on client

Synchronous I/O implies, that the following actions should be allowed to perform in the same code path:

- wrap OS-specific page into lustre structure
- send it for IO to lower layer
- wait for IO completion

### 3.2 Asynchronous I/O on client

This kind of I/O should allow the following actions being taken:

- wrap OS-specific page into lustre structure
- send it for IO to lower layer
- after I/O is completed, registered I/O completion handler should be called asynchronously, notifying that I/O is finished. This handler may be used also for accounting, etc.

### 3.3 Getting metadata from MDS

Client needs metadata from MDS server(s) when reading directory content. API should allow this kind of I/O as well. This may be using synchronous API.

### 3.4 Handling IO on server side

On server the same set of functions should be used for handling clients IO requests as on client for sending them. This makes API set minimized and well suited for all cases, and thus, easy to maintain and improve.

### 3.5 Working with llog

There are needs to write llog related data sometimes. It has own specific, as I/O should not be page oriented and rather bytes oriented one. So API should have functions like *read()* and *write()* which accept number of bytes to read and pointer to a buffer read data should be stored in.

### 3.6 Working with lvfs bits

Servers need to store different bits of data into EA or to special files like ones used for storing last object id, etc. This is done currently over lvfs interface and should be using the same bytes oriented API as llog.

## 4 Logic specification

The following components of API make it up and should see detailed design:

- all involved structures, for instance lustre page which is wrapper for OS specific pages.
- functions providing live circle for lustre pages. That is those needed for initializing/finalizing struct lustre_page and others.
- completion handler and its registration.
- all functions for all types of API (that is synchronous, asynchronous, etc). Interaction and examples of using lustre IO related structures.

## 5 State management

### 5.1 State sharing between objects. Helper objects

There possibly will be helper objects (asynchronous pages, etc.) which need managing. That is API should have methods for preparing that objects for using with API and also those for finalizing them. That is especially the case about memory allocations. All the cases when user of API is out of control about helper objects, API itself should take care of them.

Also, helper objects change their status while doing IO. For instance, asynchronous page is sent for IO. This is initial state. Then when IO completes page is moved to next state and so on.

### 5.2 State sharing between operations. Composite operations

It is possible that some IO operations consist of few parts. For instance, *prepare_write()* and *commit_write()*. Both they operate on some helper object or set of flags and there is implication that both should be performed. In such a case, possible breaking of such a pare of operations should also be worked out.

### 5.3 Recovery

As for recovery as a lustre process, I/O API should not break it. It may change it (should not do that if only possible). That is almost the only one thing about API related to recovery.

## 6 Focus in inspection

There is one of important ideas of this HLD which is to design minimal and suitable for all IO cases API as only possible. Such a minimization attempt may cause kind of lack of functionality to API.

For instance, if we try to use *read()* and *write()* API for page oriented IO, it may be not optimal, or API will be complicated or another kind of suffering is possible.

This should be taken into account when doing review.