HDF5 Virtual Object Layer (VOL) Connector Author Guide

The HDF Group

27th October 2019

Contents

1 Introduction **1**

2 New VOL Code **1**

3 The VOL Property in the File Access Property List **1**

4 Changes in API Routines Implementation **2**

5 Object Identifiers **3**

6 Using `H5I_object()`, `H5I_object_verify()`, and `H5I_iterate()` **4**

7 Native Plugin Implementation **5**

8 The Named Datatype Pickle **6**

9 Adding New Features to the Library **7**

1 Introduction

The Virtual Object Layer (VOL) is an abstraction layer in the HDF5 library that intercepts all API calls that could potentially access objects in an HDF5 container and forwards those calls to plugin "object drivers". The plugins could store the objects in variety of ways. A plugin could, for example, have objects be distributed remotely over different platforms, provide a raw mapping of the model to the file system, or even store the data in other file formats (like native netCDF or HDF4 format). The user still gets the same data model where access is done to a single HDF5 "container"; however the plugin object driver translates from what the user sees to how the data is actually stored. Having this abstraction layer maintains the object model of HDF5 and would allow HDF5 developers or users to write their own plugins for accessing HDF5 data.

This guide is for HDF5 library developers responsible for maintaining the HDF5 C library. It details the changes made to the library to implement the VOL, and the necessary requirements for adding new features to the library to work with the Virtual object layer. Readers of this document are expected to be quite familiar with the VOL high level architecture presented in [1] and the VOL user's guide[2].

2 New VOL Code

The VOL captures all API routines that could potentially access the HDF5 file and routes them through the VOL intermediate layer which in-turn calls the plugin callback that implements the API function.

The VOL code additions have been added to the library similar to any new interface introduced. The library's acronym to all VOL related variables and functions is `H5VL`. The public VOL definitions including the VOL class and callbacks are in `H5VLpublic.h`. Private routines are in H5VLprivate.h. The actual implementation for public and private routines are in `H5VL.c` and `H5VLint.c` respectively.

3 The VOL Property in the File Access Property List

A new property is added for file access property lists (FAPL) that indicates what VOL plugin should be used when accessing a file with the given FAPL. The property `H5F_ACS_VOL_ID_NAME` maps to the plugin identifier to use. If the plugin selected requires information to be passed down from the application, another property is used, `H5F_ACS_VOL_INFO_NAME`, which translates to a `void` pointer to the information needed by specific the VOL plugin. The property setting are analogous to the Virtual File Driver properties for selecting and using file drivers with the native HDF5 library. For more information on selecting VOL plugins refer to the VOL user guide.

The default settings for the FAPL properties for the plugin identifier is the global identifier for the native plugin. Since the native plugin does not require any extra information from the application, the info property is set to `NULL` by default. The global identifier for the native plugin is initialized when the HDF5 library is initialized by registering a global identifier with the native plugin structure.

Whenever a plugin is set to be used with a FAPL or when copying a FAPL, the library calls:

```
herr_t H5VL_fapl_open(H5P_genplist_t *plist, hid_t vol_id, const void
    *vol_info);
```

which increments the reference count on the plugin, calls the plugin specific `info_copy` callback if the `vol_info` parameter is not `NULL`, and sets the VOL properties for the VOL identifier and info that are described above.

When a FAPL is closed, the library calls:

```
herr_t H5VL_fapl_close(hid_t vol_id, void *vol_info)
```

which calls the `info_free` callback of the VOL plugin with the info if `vol_info` is not `NULL`, and decrements the reference count on the VOL identifier.

4 Changes in API Routines Implementation

The VOL is implemented as an abstraction layer in the HDF5 C library. All API calls have been modified to call into the intermediate VOL layer instead of executing the operation directly with the HDF5 native implementation. For example, the function `H5Fcreate()` was implemented as:

```
    /* Check arguments */
    ...
    /*
     * Create a new file or truncate an existing file.
     */
    if(NULL == (new_file = H5F_open(filename, flags, fcpl_id, fapl_id,
        H5AC_dxpl_id)))
       HGOTO_ERROR(H5E_FILE, H5E_CANTOPENFILE, FAIL, "unable to create
           file")
```

where `H5F_open()` is an internal HDF5 function that creates/opens a native HDF5 file. With the VOL changes introduced, the implementation will change to:

```
    /* Check arguments */
    ...
    /* create a new file or truncate an existing file through the VOL */
    if(NULL == (file = H5VL_file_create(vol_cls, filename, flags,
        fcpl_id, fapl_id, H5AC_dxpl_id, H5_REQUEST_NULL)))
```

```
        HGOTO_ERROR(H5E_FILE, H5E_CANTOPENFILE, FAIL, "unable to create
            file")
```

where `H5VL_file_create()` is a private function defined in the intermediate VOL layer that calls the the create callback in the file class of the VOL plugin selected in the file access property list.

This change is introduced for all such API routines that could potentially access the file. Multiple API routines could map to the same VOL callback. For example `H5Acreate()`, `H5Acreate_by_name()`, and `H5Acreate_by_idx()` all map to the attribute create intermediate routine which calls the attribute create callback in the VOL class. For an exact mapping of API routines to VOL intermediate functions and callbacks, please refer to the VOL user guide or the VOL header file in `src/H5VLpublic.h`. All the intermediate routines are implemented in `H5VLint.c`.

5 Object Identifiers

All functions that create or open objects in an HDF5 file return an identifier of type `hid_t` for that object. This identifier could used by the application to access other objects or operate on the object itself. Internally the HDF5 library maintains a mapping of that identifier to the private data structure of the object that it represents. For example, `H5Fcreate()` returns an identifier that internally maps to the private `H5F_t` structure of that file; `H5Dcreate()` returns an identifier that maps to the `H5D_t` private structure of the dataset, etc...

With the introduction of the VOL, the current mapping between the identifiers and the private structures is not possible anymore for two main reasons. The first being that the private structures that the identifiers point to now are specific to the native HDF5 implementation. Other plugins will have different structures that they implement. Thus at the API and VOL intermediate layer, those objects are accessible only through a `void` pointer with an unknown structure type to cast it to. The second reason is that the internal structures (`H5F_t, H5D_t, H5G_t, etc ...`) do not contain a pointer to the plugin class used to create the object. This is problematic because subsequent access on those object through the identifiers would not allow us to determine the VOL class to forward the call to.

To get around the two issues explained above, we introduce a private object structure for all File, Attribute, Dataset, Group, and Named Datatype objects. Those set of objects are the ones currently stored in the HDF5 file. Other object types (dataspace, property lists, etc...) are memory space objects that are not stored or accessed through an HDF5 file. All the API routines that create/open objects will create this "wrapper" object around the object pointer returned from the intermediate VOL function. The new object is defined in H5VLprivate.h as:

```
/* The internal vol object structure returned to the API */
```

```
typedef struct H5VL_object_t {
    void             *vol_obj;      /* pointer to object created by
        plugin */
    H5VL_t           *vol_info;     /* pointer to VOL info struct */
} H5VL_object_t;
```

where `vol_obj` is a pointer to the plugin created object. In the native plugin this would be a pointer to the private structures of that object (`H5F_t, H5D_t, H5G_t, etc ...`). The `vol_info` contains the information about the plugin that created the object and is defined as:

```
/* Internal struct to track VOL information with objects */
typedef struct H5VL_t {
    const H5VL_class_t *vol_cls;     /* constant plugin class info */
    int                nrefs;        /* number of references by objects
        using this struct */
    hid_t              vol_id;       /* identifier for the VOL class */
} H5VL_t;
```

`vol_cls` is a pointer to the plugin class. `nrefs` indicates how many object currently point to the `H5VL_t` structure. `vol_id` is the global identifier for the VOL plugin used. Usually this structure is created in `H5Fcreate()` or `H5Fopen()` with `nrefs = 1`, then `nrefs` is incremented by 1 on any subsequent object creation that stored a copy of that pointer and decremented when the object is closed. When `nrefs` becomes zero, the structure is freed.

As a convenience, a new private routines is provided to create the wrapper object for the underlying plugin specific object pointer, manage the VOL info structure, and register an identifier to be returned to the user:

```
H5VL_register_id(H5I_type_t type, void *object, H5VL_t *vol_plugin,
    hbool_t app_ref);
```

This routine is implemented in `H5VLint.c`.Instead of calling `H5I_register()` to create an identifier, developers should now use the above routine to get back an `hid_t` for objects stored in HDF5 files.

6 Using `H5I_object()`, `H5I_object_verify()`, and `H5I_iterate()`

The change in the mapping of identifiers to the corresponding object pointer and introducing the new wrapper `H5VL_object_t` structure, means that any usage in the library of `H5I_object(), H5I_object_verify(), and H5I_iterate()` expecting to get back a pointer to the native structure of the object is broken, since those routines return a pointer to the object without unwrapping it from the `H5VL_object_t` structure. To fix the above problem, we introduced two private convenience functions:

```
void *H5VL_object(hid_t id);
void *H5VL_object_verify(hid_t id, H5I_type_t obj_type);
```

which do the same thing as their `H5I` counterpart but return a pointer to the actual VOL specific object (`vol_obj`) instead of the unwrapped object pointer that the identifier points to. The objects identifiers must be either files, datasets, attributes, named datatypes, or groups; otherwise the function will return `NULL` for erroneous usage. We replaced all usage of `H5I_object()` and `H5I_object_verify()` in the library as appropriate for the above listed object types. Developers should use the `H5VL` functions above in the native library and tests when expecting to get back the plugin specific objects.

As for `H5I_iterate()`, we update the implementation of this function internally to unwrap the object being iterated over if it is a file, dataset, group, attribute, or named datatype, and return the plugin specific object to the iterate callback. We saw this as a better approach than requiring a change in all the iteration callbacks to check for the object type and unwrap it accordingly. With the current solution, no code changes are required in the internal usage of `H5I_iterate()`.

7 Native Plugin Implementation

The native HDF5 plugin implements the VOL callbacks with the native HDF5 file format. This is basically all what the library currently does when implementing HDF5 features. The native VOL plugin is implemented in `H5VLnative.c`. The callbacks of the VOL class are implemented here similarly to how the API routines were mapped directly to the internal HDF5 implementation before the VOL was introduced. However, since there isn't a one to one mapping from every API routine to a VOL callback, there is an additional level of processing parameters passed in from the VOL intermediate layer. The protocol for parameter processing is explained in detail in the VOL user guide.

There are places in the library where we need to register an identifier to pass directly to the application (for example, creating a group identifier and calling the user defined link iteration callback). To do that, we need to create the high level object wrapper and attach the native plugin information before registering that ID. This can be done by calling this private and native specific routine:

```
hid_t H5VL_native_register(H5I_type_t type, void *obj, hbool_t app_ref);
```

which does all the steps required for creating an identifier for the native object. That identifier has to be freed with:

```
herr_t H5VL_native_unregister(hid_t obj_id);
```

8 The Named Datatype Pickle

Datatypes in HDF5 exist in two states, transient and committed. When a user creates an HDF5 datatype, it is in the transient state, i.e. in memory space. It does not exist in the file; in fact, an HDF5 program could work with datatypes without even opening or accessing any HDF5 file. HDF5 provides a way for users to write (or commit) a datatype to the HDF5 file. When a user calls `H5Tcommit()` on a transient type, that datatype is written to the HDF5 file under the location specified in the commit call. The datatype then becomes a named/committed datatype. Since the datatype is not stored in the file, the use can open that datatype from the file later to retrieve it and work with it in a similar way like other objects that are stored in the file (groups, datasets, and attributes).

There are a large set of routines that operate on datatypes in general. For a complete list, look at the reference manual for the `H5T` interface. As far as the user is concerned, it does not matter whether the datatype is committed or not. Those routines work the same way for transient or named datatypes. With the addition of the VOL, a datatype class is added to handle committing and opening a named datatype, in addition to retrieving some metadata stored with the named datatype. However other operations that work with datatypes in both its transient or committed state do not pass through the VOL, since they do not access the datatype in the container. At that point the datatype should already by open and loaded into memory space for those operations to work. We chose not to handle those operations through the VOL even if that datatype is committed, to relieve VOL plugin developers from implementing a large set of datatype routines that is already available in the HDF5 library and is likely to be the same across all VOL plugins.

The design decision above does have some implications on the internal handling of named datatypes with the VOL changes. The higher level datatype APIs expect that the identifier for dataypes (named and transient) to map to the private `H5T_t` structure. However; as we explained earlier in this document, the object that we get back from the VOL plugins when committing a datatype to the file is a `void` pointer that the library just needs to keep track. For the native plugin we do know that the pointer is actually pointing to the internal `H5T_t` structure for the name datatype, but we can not make that assumption for other plugins that will return a pointer to their own implementation of named datatypes. Furthermore we need to wrap whatever object returned from the plugin when creating/committing a named datatype with the higher level object structure that contains the VOL plugin information used for creating the datatype. This mismatch between what the VOL expects and what the `H5T` API functions expect will require us to do some special handling for committing and opening named datatypes:

- Committing a transient datatype: At this point the identifier for the datatype exists and points to the private transient `H5T_t` structure. The difference between other objects and named datatypes is that the identifier for the named datatype should always dereference to the private `H5T_t` structure, so instead of updating the identifier's underlying object after committing the datatype to the newly created `H5VL_object_t` structure

for the VOL object of the named datatype, we store that object pointer in a newly added field to the private `H5T_t` structure and use that field to determine if that datatype is committed or not. At this point we satisfied the requirement for the public `H5T` functions that do not care about the state of the datatype and can continue to work with the datatype identifier since it still dereferences to the same `H5T_t` structure, but with an additional field. The additional field contains the VOL object that should be used instead of the `H5T_t` structure when operating on that datatype through the VOL.

- Opening a named datatype: The open operation for a named datatype is forwarded to the VOL that returns the VOL specific pointer to the datatype that is opened. The open operation for datatypes should return an identifier that dereferences to an `H5T_t` structure for the reasons we stated earlier. However at this point we do not have the `H5T_t` structure for that datatype. For this reason we require all VOL plugins that support named datatypes to store a serialized copy of the HDF5 datatype somewhere in the underlying container when committing a named datatype. After opening the datatype through the VOL, we ask the VOL plugin to return the serialized form of the datatype which we can call `H5T_decode()` on to create the private H5T_t structure which is a transient form of the datatype. Then we store the object created from the open operation in the new `vol_obj` field of the `H5T_t` structure we just created and atomize it to return to the user.

HDF5 datatypes proved to be quite a challenge with the VOL due to the two states that they exist in unlike any other HDF5 object. However we believe the design decisions we made and explained above are a compromise to ease the work for plugin developers and HDF5 library maintainers at the same time.

9 Adding New Features to the Library

It is expected that new features and routines will be added to the HDF5 library after the VOL has been introduced. New functionality added that accesses data in the file has to go through the VOL the same way that any other function does. There are several ways to handle extensions to the VOL.

- Adding a new VOL class for the new feature. This usually means that a new object type is added to the HDF5 library or a new service is introduced that is expected to be general in the sense that most plugins should be implemented. For example, consider the case when a new HDF5 object is added to the library that allows users to create and use a KV store in the HDF5 container. Adding a new class means that all existing plugins have to be modified to cope with the change in the VOL public structure. Changing the VOL public structure should be done only in major HDF5 releases, and not in maintenance releases.
- Adding a new callback to a VOL class. This is done when a new API function is added to a particular class of operations (Files, Datasets, Groups, etc...) and is quite important that it can't be "hidden" in the `specific`

callback of that class. This change in the public VOL class also requires all existing plugins to be updated and should be done in major releases only.

- Adding a new operation type to the `get, specific` or `optional` callbacks of a sub-class in the VOL public structure. When a new operation type to retrieve data from a particular object or to execute some new operation on that object, whether that operation is general among all plugins or plugin specific, adding a new type to the `enum` of types for the three callbacks present in every sub-class should be enough to support the new operation. For more information on how this works, please consult with the VOL user guide.
- Adding a new operation type to the `optional` callback in the VOL public class for services. New HDF5 functionality that is added specifically for certain plugins and would not make sense to promote to an actual class since it isn't general enough to be implemented by most plugins or would not be used a lot by applications should follow this option.

References

[1] M. Chaarawi and Q. Koziol, "RFC: Virtual Object Layer," 2014, available at http://svn.hdfgroup.uiuc.edu/hdf5doc/trunk/RFCs/HDF5/VOL/RFC/.

[2] T. H. Group, "User Guide for Developing a Virtual Object Layer Plugin," 2014, available at http://svn.hdfgroup.uiuc.edu/hdf5doc/trunk/RFCs/HDF5/VOL/user_guide/.