RFC: Fine grained capability reporting by HDF5 VOL connectors

Neil Fortner

The HDF Group

VOL connectors for HDF5 are user-defined plugins that essentially reimplement much of the HDF5 library, however most VOL connectors do not support the entire feature set enabled by the HDF5 API. While there is an existing mechanism to query which features are supported, it is insufficient to cover every possible case. This RFC describes possible solutions for VOL connectors to be able to report, at invocation time, when an operation is unsupported.

Contents

1 Introduction

VOL connectors are a powerful mechanism that allows developers to implement HDF5 routines in any way that is needed. Due to the wide array of features supported by the HDF5 API, developing a VOL connector for the HDF5 library can be a very demanding task. To mitigate this and make developing a VOL connector more approachable, we allow connectors to only implement a subset of HDF5 features. The VOL interface allows VOL connectors to report which features they do and do not support via the VOL capability flags facility. However, these flags are not comprehensive, and it would not be practical to list every possible supported feature or combination of features. This RFC lists alternative ways for VOL connectors to indicate that an operation is not supported, and, similarly, ways for the HDF5 library to indicate to the application that the VOL connector does not support the operation.

2 Motivation

The motivation for this feature primarily comes from testing. We maintain a thorough HDF5 API test suite that is meant to be used to validate VOL connectors. However, since not all VOL connectors support all features, it can be difficult for this test suite to know when a feature is not supported by a particular VOL connector and should therefore not report a test failure. While the test suite does leverage the VOL capability flags feature, and does not test features that are reported to be unsupported, the VOL capability flags facility simply cannot hope to ever cover all possible situations where a feature may or may not be supported by a VOL connector, as there are almost countless possibilities. For example, a VOL connector may only support variable length data with an atomic parent type, or may only support attributes with a scalar dataspace, or may only support absolute paths for soft links.

Therefore, we want to add a mechanism for the VOL connector to report to the HDF5 library that it cannot complete an operation at the time the operation is invoked, rather than attempting to enumerate all possibilities beforehand, and ways for that same information to be passed from HDF5 to the application. We will, of course, retain the existing VOL capability flags for coarse-grained feature support queries. While this is primarily intended for testing purposes, it could also be used by applications that are designed to gracefully handle certain operations being unsupported, possibly by using an alternate method to store the data in such a case.

This information needs to be ultimately passed from the VOL connector to the application. However, since there are two public interfaces between the VOL connector and the application, we must define the method of passing this information across each interface. These methods do not necessarily need to be the same or even similar, so we will treat each interface separately.

3 Approaches to VOL Interface Extension

The VOL connector needs a way to report to the top level of the HDF5 library that the requested operation cannot be executed because it is not supported. This is different from an operation that cannot be completed due to invalid input, a system error, or an internal VOL connector error. The reason it is not supported may be apparent purely from the parameters passed to the callback (or even the entire callback may not be supported), or it may arise from a combination of the parameters with preexisting state in the library or the file. There are several ways this can be signaled to the HDF5 library. Some possibilities are listed here:

3.1 Approach 1: Return Value

One possibility is to expand the meaning of the return values in the `herr_t` type. This is possible because currently `herr_t` is defined to be a signed integer where a non-negative value indicates success, while a negative value indicates failure. In practice, an error is always indicated with `-1`, and the FAIL macro is defined to `-1`, leaving all other negative values unused and available for use as a way to indicate more about the cause of the error (i.e. an error code). We could therefore define, perhaps, `-2`, to indicate that the operation is supported by the HDF5 API but not by the underlying VOL connector. VOL connectors would then return `-2` in this situation, allowing the top level of the HDF5 library to report this to the application, the method of which will be discussed in the next section.

Advantages to this approach include its simplicity, and its compatibility with existing code. If the VOL connector is not updated to support this, then all that happens is unsupported actions are still reported as normal errors, so the existing behavior is not changed. There is a small chance that existing VOL connectors are returning `-2` for some errors, but that seems unlikely as there is currently no reason to do so. In general, it may not be preferable to change the semantics of the return value without changing the signature, as this can cause problems to occur elsewhere as other functions' assumptions about its behavior are violated, but in this case it is unlikely to cause any such problems.

Some VOL callbacks, primarily those that create or open HDF5 objects, return a pointer to that object instead of an `herr_t` value. This presents an obvious difficulty for this approach. One solution would be to change the signatures of these callbacks to instead return `herr_t` as the return value and return the pointer through a parameter. This is the cleanest solution but would require VOL developers to update their code, and add precompiler directives if they want to support both versions of HDF5. Another solution would be to provide a special pointer through the public API for callbacks to return to indicate an unsupported operation. This pointer would be initialized at library initialization time with a memory address that cannot be used by the application (possibly the address of the pointer itself). This is slightly strange but avoids breaking existing code.

3.2 Approach 2: VOL API Call

Another possibility is to add a new H5VL developer API, which a VOL connector callback would invoke before returning to indicate to the top-level HDF5 library that the current callback cannot be executed. The callback would then return a return value indicating success (`0`), and the library would then report the fact that the operation is unsupported to the application. The callback must return success because if a true failure occurs after the new H5VL routine is invoked, the callback needs to be able to report a true failure.

This has the advantage of not changing the form or semantics of any existing interfaces. It can also be easily extended in the future to allow more fully featured and flexible error-type reporting. However, it could be considered slightly clumsier for VOL connectors to implement than a special return value. In addition, the HDF5 library will need to take care that, when noting the fact that the current operation is unsupported, it is stored in the correct API context (library state), and is only associated with the current thread. There is already machinery in place to handle this, though, so this should not be a major concern. Care will also need to be taken that passthrough VOL connectors can handle this correctly, and that passthrough VOL connectors that have not been updated to support this do not accidentally return (a supported) success to the top layer if the underlying VOL connector returns unsupported. This can be accomplished by ensuring the public H5VL* routines return a negative return value (while following the rest of the normal API rules, as discussed in the next section).

3.3 Approach 3: Exposed Global Variable

It would also be possible for the HDF5 library to expose a global variable that a VOL connector could modify in order to signal the cause of an error returned. This is analogous to the `errno` facility in C, but has some problems. Primarily, in order to handle the multithreaded case correctly it will be necessary to make this a thread-specific variable. Since we don't require `pthread` support, it will then be necessary to create public versions of H5TS_key_create() and H5TS_key_delete(), though these could be hidden from the VOL connector by use of a macro, similarly to `errno`. In addition, it will be possible for passthrough VOL connectors to accidentally overwrite errors reported by the terminal VOL connector if they aren't careful. This approach is conceptually similar to the new VOL API call outlined in section 3.2, but since the new API routine does not require exposing the internal thread specific variable abstraction and should be able to handle multiple errors correctly without relying on the VOL connector, the global variable approach is probably not the best.

3.4 Approach 4: New Parameter

Another approach is to add a new parameter to all VOL callbacks to allow the return of more detailed error information. Alternatively, we could modify the type of the return value for these callbacks to be a new type defined by HDF5, and add library calls to encode error information in this type. This approach has the advantage that, similarly to the new API call idea, it is very flexible and can easily be extended to more thorough error reporting. It also does not change the semantics of existing parameters and may be seen as a more natural option in terms of the code flow in the VOL connector. However, it will require all VOL callbacks to change their signature and, therefore, will create a heavier burden on VOL connector maintainers.

3.5 Approach 5: Error Stack Parsing

One last possibility is to require the VOL connector to use the public H5E interface to populate an error stack, then the HDF5 library would either parse the stack using internal H5E routines, or would capture the text printed to stderr. In either case, the librarywould look for a specific major or minor error code or a specific text string. This has many problems and is probably not the best option, but is included here for completeness. First, many VOL connectors do not use the HDF5 error stack mechanism and imposing this requirement would require substantial work on the part of VOL connector developers. Also, parsing the stack directly would require the VOL connector to register its stack with the library in a special way, and the library would need to print this stack on error after parsing it. Capturing the stderr output may be different on different architectures, adding to the complexity of the code and configuration, and parsing the raw text runs the risk of unintended matches.

Finally, this approach would proliferate the use of major and minor error codes within the library, which may not be preferred since the removal of these codes entirely has been suggested in the past.

4 Approaches to Top Level API Extension

Once the HDF5 library is informed that the VOL connector does not support an operation, it needs a way to signal this status to the application that invoked the HDF5 library. This will then allow the application to take appropriate action depending on whether the error was due to an unsupported feature or a system error.

4.1 Approach 1: Return Value

If we decide to extend the meaning of the herr_t return value, as outlined in section 3.1, it is only natural to apply this to the top-level HDF5 API as well. As with the VOL interface, this should not break compatibility with existing code that follows existing guidelines, that state that any negative return value is considered an error. However, there may be application code that inappropriately checks for an error return from HDF5 API calls by comparing the return value to `-1` (or `FAIL`). In this case, if the code is not updated, returning `-2` to indicate the operation was not completed because it was unsupported would cause the application to behave as if the operation was completed successfully, causing problems later. While this approach is more likely to cause issues at the top level than at the VOL layer, it should still be a workable approach since it does not violate existing guidelines.

Other return value types, such as `hid_t` and `htri_t,` can be handled similarly, since they are also signed integers that have multiple possible negative values where only one is currently used. There are no public API functions that pass through to the VOL layer that return pointers, so there should be no need to modify any function signatures or provide a special pointer return value, as is the case for the approach outlined in section 3.1 for the VOL interface.

4.2 Approach 2: HDF5 API Call

Another approach to pass this error information to the application is to add a new public HDF5 API routine that, when invoked, would give more information on the cause of the last error returned by HDF5 in the current thread. As with the approach outlined in section 3.2, this would enable greater flexibility in error reporting as it would not be limited to a single number, but would require care to ensure it behaves appropriately in multi-threaded contexts. This can also be seen as an inelegant solution from a coding perspective, as it adds a hidden state on the library instead of simply returning the information immediately and clearing the state.

4.3 Approach 3: Exposed Global Variable

Similarly to the approach outlined in section 3.3, we could expose a global variable analogous to `errno`. While this is similarly inelegant to the API call described in section 4.2, it is a familiar programming pattern and does not have the disadvantages related to passthrough VOL connectors as approach 3.3. As with that approach, we would need to expose H5TS thread-specific variable facilities to enable thread safety, though we could use a macro to hide this complexity from the application.

4.4 Approach 4: New Parameter

Analogously to the approach is section 3.4, it would be possible to add a new parameter to every public API routine that can go through the VOL layer to return detailed error information, though this is not practical and would likely cause a great disruption in the HDF5 community, as all codes would need to update every line that includes an HDF5 API call (or they would need to make use of the versioning macros and set the API version appropriately).

4.5 Approach 5: Error Stack Parsing

Finally, it would be possible to define a certain string or major or minor error class as indicating a lack of support in the VOL connector and leave it up to the application to parse the error output. This is clumsy for the application but makes no changes to the API and, therefore, preserves full compatibility. Similarly to approach 3.5, it adds a slight chance that this condition could be triggered inadvertently, though this chance can be minimized with the appropriate selection of the magic string. The application would need to either capture stderr or use H5E facilities to register its own error stack handler.

5 Other Considerations

5.1 Error Stack Printing

We may want to add an option to disable printing of the error stack when an unsupported feature is invoked to keep output clean. This is especially desirable for automated testing. This option could be added as a public H5 API routine or possibly an H5E routine. This option would likely not be necessary if we go with the approaches outlined in sections 3.5 and 4.5.

5.2 Partial Support

If only part of an operation is supported but part is unsupported, for example in a multi dataset I/O operation, the VOL connector needs to decide if it should do what it can and return unsupported, or do nothing and return unsupported. It is probably best to do nothing so the file is not left in an ambiguous state, but an argument could be made that it should do as much as possible. We should make a decision on this and issue guidance.

5.3 Asynchronous Operations

Asynchronous operations have their own unique difficulties with error reporting, and we must make sure to support indicating unsupported operations, even for operations executed asynchronously. This can be accomplished by adding this information to the `H5ES_err_info_t` struct (and versioning the struct). The exact way this information should be added depends on the approach taken for the top level API in section 4, and if we go with the error stack parsing approach, no change is needed to `H5ES_err_info_t`.