Flush Dependency Testing

Testing Architecture

The flush dependency testing harness will be designed to exercise, as the name implies, the flush dependencies between the various metadata cache items. Unlike the acceptance test nature of the existing SWMR tests, the flush dependency tests are designed to act more like unit tests, with much more fine-grained control over the states of both the reader and the writer. This architecture will also serve as the core of the future debugging harness, which will allow for more efficient debugging of SWMR failures.

The testing architecture, depicted in figure 1, consists of a single SWMR writer, one or more SWMR readers, and a centralized controller, each of which will be created as a separate process. In this setup, the controller contains the test logic while the reader and writer processes simply execute various test harness I/O calls as directed. Communication between the processes will take place via messages passed over TCP/IP. A mechanism for launching the various processes on local/remote machines will be provided and this will be configurable to match the user's environment. Additionally, logging functions will also be added to the existing metadata cache code in the library so that the order of cache flushes can be verified.

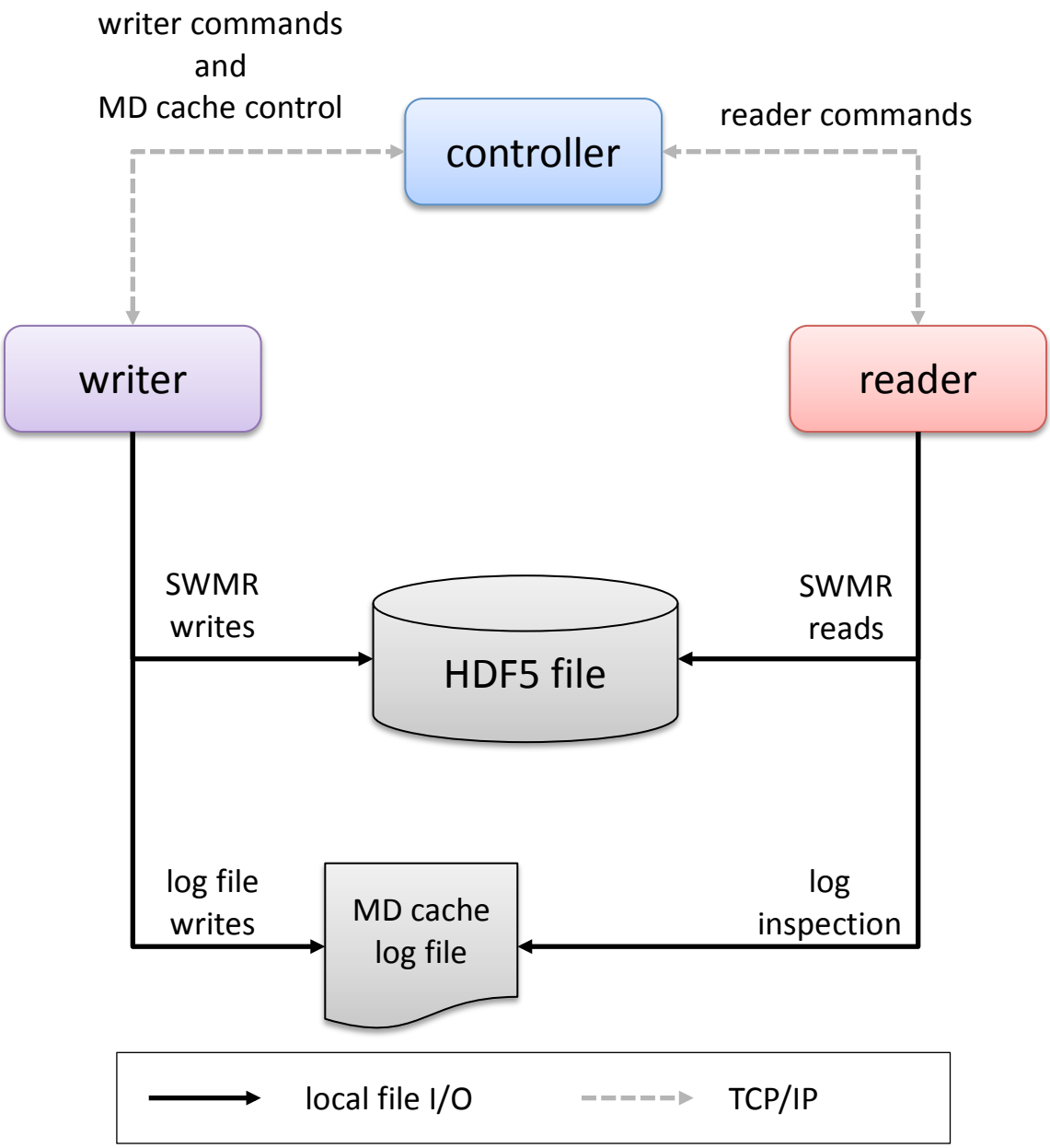

Figure 1 – Flush dependency test architecture schematic.

The overall test plan is that each data structure is tested under any condition that creates a new type of flush dependency. For example, the fixed array chunk index would be tested under both paged and unpaged configurations[1]. Additionally, data structures that can be used in multiple contexts will be tested individually and in each context in which they are used. As an example, the fractal heap would be tested on its own as well as in the context of its uses for dense attribute storage and dense group storage.

[1] See the HDF5 file format and SWMR design documents for details about these data structures.

For each test, the controller will invoke a test-specific function that will send messages to the writer and readers, causing them to perform operations on the common file. For some data structures, these reader and writer functions will use the internal HDF5 C API in order to generate specific states that can be tested. Since these states will be somewhat artificial (i.e. they may not be realizable via the current C API), all data structures will also be tested via the external HDF5 C API. As an example, for low-level extensible array testing purposes we may run through several configurations of data blocks, super blocks, index block, etc. which would then be flushed to ensure that they arrive on the disk in the correct order. These configurations would be set up via internal API calls and some may be theoretical, but important to test for completeness. Additionally, the extensible array would also be tested via standard H5D API calls. This "two level" coverage is to ensure that the flush dependencies are very thoroughly tested.

For each test, success will be verified by the reader finding correct file objects and data after each writer flush, as well as by finding the correct flush order in the log file.

At this time, the flush dependency test harness has not been implemented and its architecture is subject to change.

Adding a Test

Writing a new flush dependency test will require adding functions to all components:

1. **Controller**
   Write a new test-specific function in the controller that spells out the required order of writer, reader, and writer metadata cache commands. This may also require adding new TCP/IP messages to the test architecture.

2. **Writer**
   Map any new or modified TCP/IP messages to callback functions that implement the data structure or metadata cache changes. You will also need to write these new callback functions.

3. **Reader**
   Map any new or modified TCP/IP messages to callback functions that verify the file objects and inspect the log. You will also need to write these new callback functions.

All components of this testing harness will be written in C to avoid adding extra dependencies to HDF5 users. Invocation of the controller, reader(s), and writer could initially be via a shell script but will need to be something more portable in the future[2].

[2] This is mainly a Windows issue. We could either write a Powershell script or use a small embedded language like Lua to control invocation (Lua is very small and easy to build on systems where it doesn't exist).

A few words about forcing cache flushes

Flush dependencies are implemented by pinning the parent in the relationship so that it cannot be evicted before the child. When a chain of flush dependencies exist (e.g., A --> B, B --> C, and C --> D), this will result in only the ultimate child (e.g., D) being unpinned. During normal cache operation, as the cache approaches capacity, metadata elements will be evicted via a cyclical LRU-like algorithm, where parents are unpinned after children are flushed. This will cause the elements to be flushed in reverse order (D, C, B, and A in our example).

In order to realistically emulate the behavior of the cache, this behavior needs to be simulated. This could be done either with a simple flush call that writes out all cache objects or a more complicated function that puts pressure on the cache in a more realistic manner. For the purposes of this document, the writer's cache action is simply listed as 'flush'.

Specific Conditions Tested (by data structure)

This section lists the basic conditions that should trigger the creation of a new type of flush dependency. Since this document only pertains to phase I of the project, which only supports dataset appends, only chunk index data structures are considered here.

B-tree (version 1)

The version 1 B-tree will not be supported under SWMR due to the lack of a node checksum, which does not allow readers to detect torn writes.

B-tree (version 2)

- After initial construction
- After adding an element
- After adding enough elements to split the root
- After adding elements such that the nodes rebalance
- After adding enough elements to split the root again (longer node chain)

Extensible Array

- After initial construction
- After adding an element (stored in index block)
- After adding more elements (stored in data block)
- After adding still more elements (stored in data block pages via super blocks)

Fixed Array

NOTE: The both paged and unpaged fixed arrays must be tested.

- After initial construction
- After adding an element
- After adding an element on a different page (if a paged array)