Locking in IDS

Rolf Krahl

ICAT Meeting @ 11th NOBUGS, Copenhagen, October 2016

Introduction and Rationale

- Remember Issue # 47: “Race condition: DsWriteThenArchiver may start while put call is in process.”
- Cause was concurrency of a background task and an incoming web service call. Impact was possible loss of data.

Introduction and Rationale

- Remember Issue # 47: "Race condition: DsWriteThenArchiver may start while put call is in process."
- Cause was concurrency of a background task and an incoming web service call. Impact was possible loss of data.

Observations from discussion in the issue report:

- FiniteStateMachine takes care that never more then one background task operates on the same dataset at any time.
- Less protecion from conflicts between background tasks and incoming web service calls.
- Other cases of conflicting access found, but most of them mostly harmless. Not sure to have checked all possible cases though.
- Different mechanisms to prevent conflicts in place. Impression: code already too complicated yet still not complete.
- Review of locking in ids.server might be advisable.

Analysis

- Consider a two level storage with write access and storage unit Dataset.
- Separate threads are launched in IDS for:
  - web service calls that may come in at any time,
  - deferred operations run by the DsProcessQueue within the FiniteStateMachine,
  - maintenance tasks: FileChecker and Tidier.
- Conflicts may arise when more then one thread accesses the same dataset in main storage at the same time.
- No conflicts in archive storage. FileChecker and Tidier don't seem to cause conflicts either.
- Current mechanisms to prevent conflicts:
  - Deferred operations are queued, no more then one may operate on the same dataset at any time.
  - Some locking to hold back certain actions, see next slide.

Analysis: Locking

- The FiniteStateMachine maintains two sets of locks, `deleteLocks` and `archiveLocks`. By design, locking does only protect against two specific actions, the delete web service call and starting the DsArchiver.
- Design pattern: "Check lock before action". This is a potential race condition.
- Only the web service calls delete, getData, and put set a lock. Deferred operations are not protected.
- Locks are set on selections, not on individual datasets. This makes checking for a lock complicated.

Proposal: Standard Resource Locking

- Keep the DsProcessQueue as is.
- Replace the current locking by standard resource locking:
  - Locks are created on Datasets.
  - One singleton to obtain the locks from.
  - Two types of locks, `shared` and `exclusive`, with the obvious semantics.
  - Only one method, `lock(DsInfo dsInfo, boolean shared)`. It returns an `AutoCloseable` that releases the lock on close.
  - `lock()` should be non-blocking. It should throw an exception if the lock is not available.
  - No `isLocked()` method!
  - (Almost) any operation acting on a dataset in main storage should acquire a lock.
- For convenience, a method to acquire a lock on a DataSelection rather then on a single Dataset may be coded on top of that.

Proposal: Standard Resource Locking

- The FiniteStateMachine acquires a lock before starting the thread for a deferred operation. The lock is handed over to the thread and closed by it when done. If acquiring the lock fails, the thread is not started and the operation remains in the queue.
- Web service calls mail fail if they can't aquire a lock. A new `DataTemporaryNotAvailableException` will be thrown in this case. (Or even a `DataNotOnlineException`.)

Locking in the File System and Direct File Access

Further option that could build on top of the proposal:

Talk on direct file access to storage at last F2F meeting

- With sufficient precautions, direct read only file access to the main storage concurrently to the IDS server can be made safe.
- Somewhat inefficient: acquiring and releasing the lock for each single file access from IDS.
- Further improvement would require changes in the IDS server and to the plugin API.

- The present proposal on internal locking in IDS can be combined with file system locking in the storage plugin.
- Requires one additional plugin call.

Optional Storage Plugin Call

- Add a call `lock(DsInfo dsInfo, boolean shared)` to the main storage plugin. It will be called each time a lock is about to be acquired and should return an `AutoCloseable`. It will be closed by the close method of the lock. It should not block, but may throw an exception if the lock could not be acquired at storage level.
- Storage plugins not willing to implement this may add a dummy implementation that simply returns `null` instead.

Conclusion

- Resource based locking is a well established design pattern that has proven to be reliable in many applications if implemented correctly.
- It could replace the current action based internal locking in ids.server.
- This would simplify the code and fix remaining race conditions that are still present.
- As a further option, this could be combined with file system locking in the storage plugin to allow external processes to access the main storage in a safe way.