Overview of IDS Internals

Rolf Krahl

October 9, 2018

1 Introduction

The ICAT Data Service (IDS) operates with delayed actions, internal states, and multiple threads. This makes it somewhat difficult to understand from mere reading the sources what actions may be performed under certain conditions. The knowledge of the internal processes in IDS may be needed to properly implement a storage plugin in a non-trivial setup. In particular, one might need to know the context that each of the plugin's methods may be called in. This text shall provide a reference to accommodate this need. It is however only an overview. In order to keep the representation clear, many details have been omitted. Furthermore it is somewhat biased towards the point of view of a storage plugin developer and it is restricted to the case that the IDS is configured as a two level storage. The text is based on IDS server 1.9.0.

When working with data objects in the storage, IDS can be configured to use either one of two different storage units: datasets or datafiles. This text tries to describe the behavior in both cases.

The workflow in the IDS can be sketched as follows: the IDS waits for incoming service requests coming from the user. In some cases, these requests may be completed immediately. In other cases, a deferred operation that will be processed in the background later is queued.

Before describing this workflow in detail, we give the definition of the status of a data object in Section 2. Section 3 describes the different deferred operations and what is being done for each of them. The various operations that are queued for any given data object are kept track of in a finite state machine. This is detailed in Section 4. Section 5 lists the available service request. Besides processing the service requests, there are some maintenance tasks running regularly in the background. These tasks are described in Section 6. Finally, we provide an index of the context that each storage plugin method is called from in Section 7.

2 Internal states

The processing of a data object is influenced by its internal state and what operations have been queued for them. We say that an operation is *in process* on a data object

if this operation is either in the queue or if it is currently being executed for that data object.

The status of a data object may either be `ONLINE`, `ARCHIVED`, or `RESTORING`. In this context, `ARCHIVED` refers to files being available only in the archive storage, while `ONLINE` refers to the files being available in the main storage as well. The `RESTORING` state describes the process of making files from the archive storage available in the main storage.

A data object is `RESTORING` if a `RESTORE` operation on that data object is in process.

If this is not the case, the status is `ARCHIVED`

- if any operation other than `WRITE` is in process for the data object, or
- for a datafile, if it does not exist in the main storage, or
- for a dataset, if it is not empty (e.g. related datafiles exist), but the dataset directory does not exist in the main storage.

In all other cases, the status is `ONLINE`.

3 Deferred operations

When the queue of deferred operations is processed, a new thread is started to execute each of them. This section describes what these threads do in each case.

3.1 `ARCHIVE`

Delete the data object from the main storage. This leaves only the files in the archive storage.

3.2 `RESTORE`

Make the data object from the archive storage available in the main storage:

- For a datafile, get the file from the archive storage and copy it to the main storage.
- For a dataset, get the ZIP file of the dataset from the archive storage, extract it, and store the contained datafiles in a dataset directory in the main storage.

3.3 `WRITE`

If the data object files do not exist in the main storage, delete the data object from the archive storage. Otherwise, overwrite the file in the archive storage with the latest version:

- For a datafile, get the file from the main storage and copy it to the archive storage.
- For a dataset, get all datafiles that belong to the dataset (according to ICAT) from the main storage, package them as a ZIP file, and store it in the archive storage.

3.4 `DELETE`

This operation is only available for datafiles, not for datasets.

Delete the datafile from the archive storage.

for datafiles	ARCHIVE	RESTORE	WRITE	DELETE
none	ARCHIVE	RESTORE	WRITE	DELETE
ARCHIVE	ARCHIVE	none	ARCHIVE	DELETE
RESTORE	ARCHIVE	RESTORE	RESTORE	DELETE
WRITE	WRITE_THEN_ARCHIVE	WRITE	WRITE	none
DELETE	DELETE	DELETE	DELETE	DELETE
WRITE_THEN_ARCHIVE	WRITE_THEN_ARCHIVE	WRITE	WRITE_THEN_ARCHIVE	none

for datasets	ARCHIVE	RESTORE	WRITE
none	ARCHIVE	RESTORE	WRITE
ARCHIVE	ARCHIVE	RESTORE	WRITE_THEN_ARCHIVE
RESTORE	ARCHIVE	RESTORE	WRITE
WRITE	WRITE_THEN_ARCHIVE	WRITE	WRITE
WRITE_THEN_ARCHIVE	WRITE_THEN_ARCHIVE	WRITE	WRITE_THEN_ARCHIVE

Table 1: Update matrix for queued operations. Matrix entries are the result if a new operation (column) is queued while a previous one (row) is still waiting in the queue.

4 Finite state machine

The finite state machine manages the queue of deferred operations for the data objects. For each data object, only one operation can be queued at a time. Thus, the finite state machine sometimes uses the `WRITE_THEN_ARCHIVE` state, to remember to queue up an `ARCHIVE` deferred operation after the `WRITE` deferred operation is done. The queued operation of a data object may be changed if another operation is queued for it while the previous one is still waiting in the queue, see Tab. 1 for the update matrix.

The queue is processed regularly by a timer task that starts a new thread to execute each of the pending operations.

`WRITE` operations for datasets are processed with a delay that is fixed when the operation is queued. Any subsequent `WRITE` operation queued for the same dataset pushes this delay further.

5 Service requests

In this section, we consider each IDS request and describe in detail what is done in each case. We consider only requests that interact in any way with the storage plugin. The requests `getApiVersion`, `getIcatUrl`, `getServiceStatus`, `getSize`, `isReadOnly`, `isTwoLevel`, `ping`, and `version` are thus skipped.

Several service calls that deal with a selection of data objects in the storage expect lists of datafile, dataset, and investigation ids as parameter. As most internal processing is done at the level of the configured storage unit, it only matters which datafiles or datasets, respectively, are concerned. In these cases, we use the terms *selected datafiles*

and *selected datasets* regardless whether the data objects have actually been selected by datafile, dataset, or investigation id.

Another way to select data objects is to specify a *prepared id*. This can be done only after making a `prepareData` service call (Sec. 5.8), and it automatically selects all data objects that where previously prepared.

5.1 `archive`

Queue an `ARCHIVE` deferred operation for each of the selected data objects.

5.2 `delete`

If any of the selected data objects is not `ONLINE` (Sec. 2), queue a `RESTORE` deferred operation for them and throw a `DataNotOnlineException`.

Otherwise delete the selected data objects from ICAT and from the main storage. Finally queue a `WRITE` deferred operation for each of the selected data objects in order to get them deleted from the archive storage as well.

5.3 `getData`

If any of the selected data objects is not `ONLINE` (Sec. 2), queue a `RESTORE` deferred operation for them and throw a `DataNotOnlineException`.

Otherwise get the selected data objects from the main storage and stream their content to the client.

5.4 `getDatafileIds`

Return a list of all datafile ids that are part of the set of selected data objects.

5.5 `getLink`

If the selected datafile is not `ONLINE` (Sec. 2), queue a `RESTORE` deferred operation for it and throw a `DataNotOnlineException`.

Otherwise get the path of the file from main storage, set an ACL to grant read permission to the user on the file, and return a link to the file.

5.6 `getStatus`

Return `ONLINE` if all selected data objects are `ONLINE` (Sec. 2). Else, if at least one data object is archived, return `ARCHIVED`, otherwise return `RESTORING`.

5.7 `isPrepared`

If any of the selected data objects from a previously prepared data selection (Sec. 5.8) is not yet `ONLINE` (Sec. 2), queue a `RESTORE` deferred operation for them and return `false`.

Otherwise return `true`.

5.8 `prepareData`

If any of the selected data objects is not `ONLINE` (Sec. 2), queue a `RESTORE` deferred operation for them. Store the data selection and other parameter of the request and return the *prepared id* for later referral.

5.9 `put`

If the storage unit is set to dataset and the dataset referenced in the request is not `ONLINE` (Sec. 2), queue a `RESTORE` deferred operation for it and throw a `DataNotOnlineException`.

Otherwise store the uploaded datafile in the main storage, create the datafile object in ICAT, and queue a `WRITE` deferred operation for the referenced data object.

5.10 `reset`

Reset the status flag of the selected data objects so that they can be queried again after a failed `restore` operation.

5.11 `restore`

Queue a `RESTORE` deferred operation for each of the selected data objects.

5.12 `write`

Queue a `WRITE` deferred operation for each of the selected data objects.

6 Maintenance tasks

There are some maintenance tasks running in the background independently from user actions.

6.1 `FileChecker`

The File Checker is only available for the storage unit dataset.

Iterate over all datasets in the ICAT including their datafiles. For each dataset, get the ZIP file from the archive storage and inspect it to check that the list of datafiles in the ZIP file matches the list of datafiles in ICAT and that length and checksum matches for each of the datafiles.

6.2 `Tidier`

Query the main storage plugin for a list of data objects to remove in order to get the overall size of the main storage below the configured limit. Queue an `ARCHIVE` deferred operation for each of the returned data objects.

7 Index of plugin method calls

In this section, we list for each of the storage plugin methods the context that it is called from. Methods not listed here are never called in the considered configuration.

7.1 `archiveStorage.delete(DsInfo)`

Called from the `DsWriter` (Sec. 3.3) if the dataset directory does not exist in the main storage.

7.2 `archiveStorage.delete(String)`

Called from the `DfDeleter` (Sec. 3.4) to delete the datafile from the archive storage.

7.3 `archiveStorage.get(DsInfo, Path)`

Called from the `DsRestorer` (Sec. 3.2) to extract the dataset's ZIP file from the archive storage into the main storage.

Called from the `FileChecker` (Sec. 6.1) to inspect the ZIP file.

7.4 `archiveStorage.put(DsInfo, InputStream)`

Called from the `DsWriter` (Sec. 3.3) to store the ZIP file in the archive storage.

7.5 `archiveStorage.put(InputStream, String)`

Called from the `DfWriter` (Sec. 3.3) to store the file in the archive storage.

7.6 `archiveStorage.restore(MainStorageInterface, DfInfos)`

Called from the `DfRestorer` (Sec. 3.2) to copy the datafile from the archive storage to the main storage.

7.7 `mainStorage.delete(DsInfo)`

Called from the `DsArchiver` (Sec. 3.1) to delete the dataset from the main storage.

7.8 `mainStorage.delete(String, String, String)`

Called from the `DfArchiver` (Sec. 3.1) to delete the datafile from the main storage.

Called from `IdsBean` while processing a `delete` request (Sec. 5.2) to delete the selected datafiles from the main storage.

Called from `IdsBean` while processing a `put` request (Sec. 5.9) to delete the uploaded file from the main storage again in the case that an error occurred while creating the corresponding datafile object in ICAT.

7.9 `mainStorage.exists(DsInfo)`

Called from `IdsBean` while processing various service requests (Sec. 5) to determine the status of a dataset.

Called from the `DsWriter` (Sec. 3.3) and the `DsRestorer` (Sec. 3.2) to check if the dataset directory exists.

7.10 `mainStorage.exists(String)`

Called from the `DfRestorer` (Sec. 3.2) to check if the datafile exists.

7.11 `mainStorage.get(String, String, String)`

Called from the `DsWriter` (Sec. 3.3) to get the datafiles and add them to the ZIP archive.

Called from the `DfWriter` (Sec. 3.3) to get the datafile and copy it to the archive storage.

Called from `IdsBean` while writing data into a ZIP file.

7.12 `mainStorage.getDatafilesToArchive(long, long)`

Called from the `Tidier` (Sec. 6.2) to query the main storage plugin for a list of datafiles to remove.

7.13 `mainStorage.getDatasetsToArchive(long, long)`

Called from the `Tidier` (Sec. 6.2) to query the main storage plugin for a list of datasets to remove.

7.14 `mainStorage.getPath(String, String, String)`

Called from `IdsBean` while processing a `getLink` request (Sec. 5.5) to get the path of the file from main storage.

7.15 `mainStorage.put(DsInfo, String, InputStream)`

Called from `IdsBean` while processing a `put` request (Sec. 5.9) to store the uploaded file in the main storage.

7.16 `mainStorage.put(InputStream, String)`

Called from the `DsRestorer` (Sec. 3.2) to store the files extracted from a ZIP archive in the main storage.