IDS Exception Handling

Emil Junker

November 5, 2018

1 Introduction

This document describes the way IDS server handles exceptions when calling methods in the storage plugin.

Throughout this document, we use the terms "*after* a service request" and "*during* a deferred operation" to indicate when a particular method is called. This is important, because if a method is called immediately *after* a service request, the client may expect to be informed about possible errors via the HTTP status code. The same is not possible for errors that happen *during* a deferred operation in an asynchronous background thread.

2 Exception handling in IDS for each plugin method

Call context	Details	Exception handling

2.1 `archiveStorage.delete(DsInfo)`

`DsWriter`	Dataset is deleted from archive storage during `write` deferred operation.	Exception is caught and written to log. **No changes needed.**

2.2 `archiveStorage.delete(String)`

`DfDeleter`	Datafile is deleted from archive storage during `delete` deferred operation.	Exception is caught and written to log. **No changes needed.**

2.3 `archiveStorage.get(DsInfo, Path)`

`DsRestorer`	Dataset is copied from archive storage to extract it into main storage during `restore` deferred operation.	Exception is caught and written to log. Dataset is marked as `failure` in Finite State Machine. No possibility for data loss. **No changes needed.**
`FileChecker`	Dataset content is checked by `FileChecker`.	IOException is caught and incident is reported by `FileChecker`. **Could catch other Exceptions as well.**

2.4 `archiveStorage.put(DsInfo, InputStream)`

`DsWriter`	Dataset is written to archive storage during `write` deferred operation.	Exception is caught and written to log. Marker file of dataset does not get deleted which prevents data loss via an `archive` deferred operation. **No changes needed.**

2.5 `archiveStorage.put(InputStream, String)`

`DfWriter`	Datafile is written to archive storage during `write` deferred operation.	Exception is caught and written to log. Marker file of datafile does not get deleted which prevents data loss via an `archive` deferred operation. **No changes needed.**

2.6 `archiveStorage.restore(MainStorageInterface, DfInfos)`

`DfRestorer`	Datafile is copied from archive storage to main storage during `restore` deferred operation.	Exception is caught and written to log. Dataset is *not* marked as `failure` in Finite State Machine. No possibility for data loss, but **should record failure in Finite State Machine.**

2.7 `mainStorage.delete(DsInfo)`

`DsArchiver`	Dataset is deleted from main storage during `archive` deferred operation.	Exception is caught and written to log. **No changes needed.**

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

`DfArchiver`	Datafile is deleted from main storage during `archive` deferred operation.	Exception is caught and written to log. **No changes needed.**

`IdsBean` `delete` request	Datafile is deleted from main storage after `delete` service request	IOException is caught and written to log. `InternalException` is thrown to return HTTP status code 500. **Could catch other Exceptions as well.**
`IdsBean` `put` request	Datafile is deleted from main storage after a `put` service request when registering the datafile object in ICAT failed.	IOException is caught and written to log. Misplaced datafile does *not* get deleted. `InternalException` is thrown to return HTTP status code 500. **Should delete the datafile, and could catch other Exceptions as well.**

2.9 `mainStorage.exists(DsInfo)`

`DsWriter`	Dataset is checked for existence during `write` deferred operation.	Exception is caught and written to log. Marker file of dataset does not get deleted which prevents data loss during a possible `archive` deferred operation. **No changes needed.**
`DsRestorer`	Dataset is checked for existence during `restore` deferred operation.	IOException is caught and written to log. DfRestorer is optimistic and attempts the restoration of the dataset. **No changes needed.**
`IdsBean` `restoreIfOffline` method	Data object is checked for existence after various service requests.	IOException is not caught and left for calling method to deal with. **No changes needed.**

`IdsBean` `delete` request	Data object is checked for existence after `delete` service request.	IOException is caught and written to log. `InternalException` is thrown to return HTTP status code 500. **Could catch other Exceptions as well.**
`IdsBean` `getStatus` request	Data object is checked for existence after `getStatus` service request.	IOException is caught and written to log. `InternalException` is thrown to return HTTP status code 500. **Could catch other Exceptions as well.**
`IdsBean` `write` request	Data object is checked for existence after `write` service request.	IOException is caught and written to log. `InternalException` is thrown to return HTTP status code 500. **Could catch other Exceptions as well.**

2.10 `mainStorage.exists(String)`

`DfRestorer`	Datafile is checked for existence during `restore` deferred operation.	IOException is caught and written to log. `DfRestorer` is optimistic and attempts the restoration of the datafile. **No changes needed.**

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

`IdsBean` `getData` request	Content of datafile or dataset is read from main storage and sent to the client after a `getData` service request.	IOException is caught and written to log. JSON response with error message is sent to the client. **Could catch other Exceptions as well.**

`DsWriter`	Dataset content is read from main storage during `write` deferred operation.	Exception is caught and written to log. **No changes needed.**
`DfWriter`	Datafile content is read from main storage during `write` deferred operation.	Exception is caught and written to log. **No changes needed.**

2.12 `mainStorage.getDatafilesToArchive(long, long)`

`Tidier`	`Tidier` asks main storage for datafiles to archive, in order to free up space.	Throwable is caught and written to log. `Tidier` continues its work. **No changes needed.**

2.13 `mainStorage.getDatasetsToArchive(long, long)`

`Tidier`	`Tidier` asks main storage for datasets to archive, in order to free up space.	Throwable is caught and written to log. `Tidier` continues its work. **No changes needed.**

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

`IdsBean getLink` request	Path of a datafile is determined after a `getLink` service request.	IOException is caught and written to log. `InternalException` is thrown to return HTTP status code 500. **Could catch other Exceptions as well.**

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

`IdsBean` `put` request	Datafile is put into main storage after a `put` service request.	IOException is caught and written to log. `InternalException` is thrown to return HTTP status code 500. **Could catch other Exceptions as well.**

2.16 `mainStorage.put(InputStream, String)`

`DsRestorer`	Datafile is copied to main storage during `restore` deferred operation.	Exception is caught and written to log. **No changes needed.**

2.17 `mainStorage.lock(DsInfo, boolean)`

`LockManager`	`LockManager` asks main storage to acquire a file system level lock for a dataset.	IOException and AlreadyLockedException are caught and then thrown. **No changes needed.**

3 Conclusion

The only time errors within the storage plugin methods have the potential to be harmful, is when they are thrown during deferred operations.

In all other cases, the calling client can be (and is being) informed about the error via the HTTP status code. Also, there is no possibility for data loss in these cases.

As for errors during deferred operations, the only truly potentially critical operations are `restore` and `write`, as they copy data between main and archive storage.

Errors during `restore` deferred operations are currently being protected against by marking them as "failure" or "success". As this information is stored in-memory, this method does not allow for IDS to be restarted. Nevertheless, it is sufficient to protect against unwanted behavior during `restore` deferred operations.

The more critical case are errors during `write` deferred operations, because it is the equivalent of archiving (i.e. "saving") a piece of data from the main storage permanently. Currently, errors during `write` deferred operations are being protected against by creating special "marker" files. These files are being created before the operation is queued, and only being deleted after it has been completed successfully, surviving even a system restart of IDS. The way IDS deals with leftover marker files after a restart might be further investigated and improved, though.