Table of Contents

ASO Non-Blocking Rewrite

Introduction

The page covers a proposed rewrite of the TransferWorker component of ASO to introduce non-blocking behavior. A few goals and non-goals:

- A non-goal is to manage concurrency. The TransferWorker should not try to rate limit submissions or throttle the number of transfers in the system (although it may try to rate-limit monitoring). We assume that any concurrency management occurs external to the system.
- We keep the 1-to-1 relationship between active users and TransferWorker processes.
- We aim for FTS3 compatibility and not FTS2.
- We aim to separate serialization (CouchDB) of state from the transfer logic.
- We aim to not block on the completion of transfers, but it is acceptable to block on the interaction with components (such as sending commands to SRM or FTS).
- Each file transfer is managed by at most one TransferWorker. That is, we do not worry about two different TransferWorkers both trying to operate on the same file.
- We aim to give ASO users an indication of progress made on the transfer.

The TransferWorker process will have two main classes:

- TransferAgent: This manages the transfer lifecycle of one or more files. It is responsible for submitting transfers to FTS3, monitoring the status, and performing any post-transfer cleanup.
- TransferState: This manages the state of transfers in the system. It will provide a list of transfers to be performed, serialize the state, and make it possible to recover from a unexpected exit of the process.

TransferState

The TransferState object provides the following methods with the noted semantics:

- *getNewTransfers()*: Returns a list of (source site, source LFN, dest site, dest LFN, ASO transfer hash) tuples. Each represents a new transfer which should enter the system. This represents the beginning of a transaction which is ended with either *setTransferId* or *abortTransfer*; if neither is called within 15 minutes, the transaction is assumed to be aborted.
- *setTransferID(list of (ASO transfer hashes, destination PFN) tuples, FTS job ID)*: Throws an exception if the transfer state cannot be persisted. This records a FTS job ID for a set of ASO transfer hashes. This should be a durable commit.
- *abortTransfer(list of ASO transfer hashes, detail)*: Throws an exception if the transfer cannot be aborted. This indicates that the transfer failed to be submitted to the transfer system; due the fact

that FTS lacks a two-phase commit protocol, it is undefined whether a transfer should be retried in the future. Advice is to wait for 15 minutes before retrying the transfer.
- *setTransientTransferState(list of ASO transfer hashes, detail)*: Throws an exception if the transfer state cannot be persisted. This records a transient transfer state (such as "in progress", "X percent complete", or simply a timestamp). On error, the transfer will not be cancelled by the TransferAgent. The implementation may make a non-durable commit as an optimization.
- *getActiveTransfers(user)*: Returns a list of (list of (ASO transfer hash, destination PFN) tuples, FTS job ID) tuples for all ASO transfers in a non-terminal state.
- *setTerminalState(list of ASO transfer hashes, state)*: Throws an exception if the transfer state cannot be persisted. This persists a terminal transfer state; the implementation should make a durable commit. If this fails, this call will be repeated in the future. Any ASO transfer set to a terminal state will not be returned as a part of a future *getActiveTransfers* result unless it is first returned by *getNewTransfers*.

The TransferState object does not need to be thread-safe or re-entrant.

A first implementation of the TransferState should be in-memory (picking up new transfers from a file on disk) to allow for easy testing of the TransferAgent. The "real" TransferState implementation should persist to the ASO CouchDB. Note that the ASO transfer hashes need to embed the revision for interacting with CouchDB in order to handle CouchDB's eventual consistency model. Finally, any interaction with the ASO should do bulk updates to reduce the number of interactions with cmsweb.

The API calls getNewTransfers / setTransferID / abortTransfer should be treated as a BEGIN TRANSACTION / COMMIT / ABORT, respectively. In particular, getNewTransfers should update a timestamp in CouchDB to denote that the row is locked.

For *getNewTransfers*, *setTransferID*, *abortTransfer* and *setTerminalState*, any failures due to document conflict should be retried. For *setTransientTransferState*, no retry is needed.

It is suggested that any aborted transfer or transfer be retried no sooner than 15 minutes later. Retry policy (number of retries and which transfers should be retried) can be handled separately.

TransferAgent

The TransferAgent will expose one method, *execute*, which performs the logic of transfers. The *execute* method will have a main loop with approximately the following logic; the loop will break whenever there is no transfer being managed, no new transfers available, and all completed transfers have been set to terminal state.

1. **SUBMIT PHASE**: Query the TransferState.getNewTransfers for new transfers. If there are new transfers, group them into lists of transfers for (source site, destination site) pairs. Resolve into

PFNs using PhEDEx data service calls; record failures with TransferState.abortTransfer. For each (source, destination) pair,
   1. Submit a transfer to FTS using `fts-transfer-submit`. Block until this returns.
   2. If successful, record the FTS job ID using TransferState.setTransferID. Otherwise, record the failure with TransferState.abortTransfer.
2. **MONITOR PHASE**: Call TransferState.getActiveTransfers for a list of active transfers. [As an optimization, this may be done once every N loops.]
   1. For each active FTS transfer, call `fts-transfer-status --json` to get the status of the transfer. Block until this returns. If this returns non-zero, continue to the next FTS transfer. Group the results in-memory by state.
   2. Update transfers with TransferState.setTransientTransferState (as many transfers should be done at once as possible).
3. **RECORD PHASE**: For any ASO transfer considered in step (3) which is in a terminal state (failed or successful),
   1. Any completed but failed transfers trigger a `lcg-del` to delete the file at the destination (as many transfers should be done at once as possible).
   2. Any completed transfers are updated with TransferState.setTerminalState (as many transfers should be done at once as possible).
4. If no new transfers were submitter, no in-progress transfers were detected by *getActiveTransfers*, and any calls to *setTerminalState* were successful, break. Otherwise, sleep for 60 seconds and goto 1.

Note that any calls to `fts-transfer-status` or `lcg-del` should be wrapped with timeouts. OTOH, `fts-transfer-submit` should not; any failures in `fts-transfer-submit` should result in the corresponding transfer to be aborted (it's quite unfortunate that FTS has no commit protocol).

Progress monitoring

This rewrite has the side-effect of providing the client with liveness monitoring. All transfer jobs should get updated once every 15 minutes; if no updates are observed, the client on the schedd may decide to give up on the ASO application and resubmit (if necessary, using direct stageout instead of ASO).

This topic: Main > Crab3ASORewrite
Topic revision: r2 - 2014-02-09 - BrianBockelman