



Scaling Small Files Aggregation feature up for performance and capacity using cache clusters

Alexander Moibenko

High Level Design for using cache clusters to scale SFA for increasing number of user groups.

Table of Contents

<u>CONFIGURATION WITH SINGLE CACHE SERVER (CURRENT PRODUCTION).</u>	3
<u>CONFIGURATION WITH MULTIPLE CACHE SERVERS.</u>	4

Configuration with single cache server (current production).

There can be 4 types of different Enstore cache areas in the current SFA implementation:

- Write cache – where files are written by disk movers.
- Packaging area – where files from write cache are packaged and packages get written to tapes.
- Stage area – where packaged files get staged from tape.
- Read cache – where unpackaged files get stored for the disk movers read access.

These areas are mounted on servers, hosting disk movers and migrators. The packaging and unpackaging processes can run either on these same servers or on remote servers (which usually are the servers providing storage for Enstore cache) – aggregation and staging host. Currently this storage is accessed over NFS or Lustre, but can be provided over any distributed FS. The corresponding entries in disk mover and migrator configurations are specified as:

- Migrator:
 - 'data_area': write_cache_area,
 - 'archive_area': archive_area,
 - 'stage_area': stage_area,
 - 'tmp_stage_area': tmp_stage_area,
 - 'aggregation_host': aggregation_host,
 - 'staging_host': staging_host,
 -
- Disk mover:
 - 'device': write_cache_area,
 - 'tmp_dir': '%s/tmp'%(write_cache_area,)

Below is the current production SFA configuration.

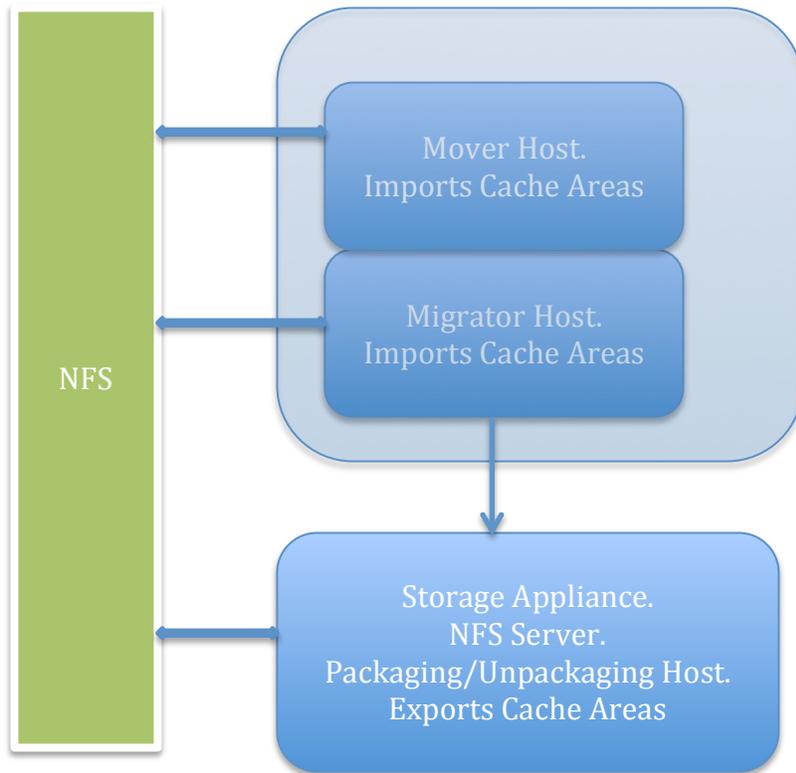


Fig 1. Single cache server configuration.

Configuration with multiple cache servers.

The implementation of disk movers and migrators is such that it allows to have more than one cache area served. There is a problem though – packaging and un-packaging of files is done directly on cache servers for better performance. This also can be done on migrator host over NAS. Currently we use NFS and performance of these operations is much worse comparing with direct packaging and un-packaging. The described design suggests to use cache clusters consisting of disk movers and migrators grouped around a single enstore disk library and using cache provided by single cache server (Fig 2).

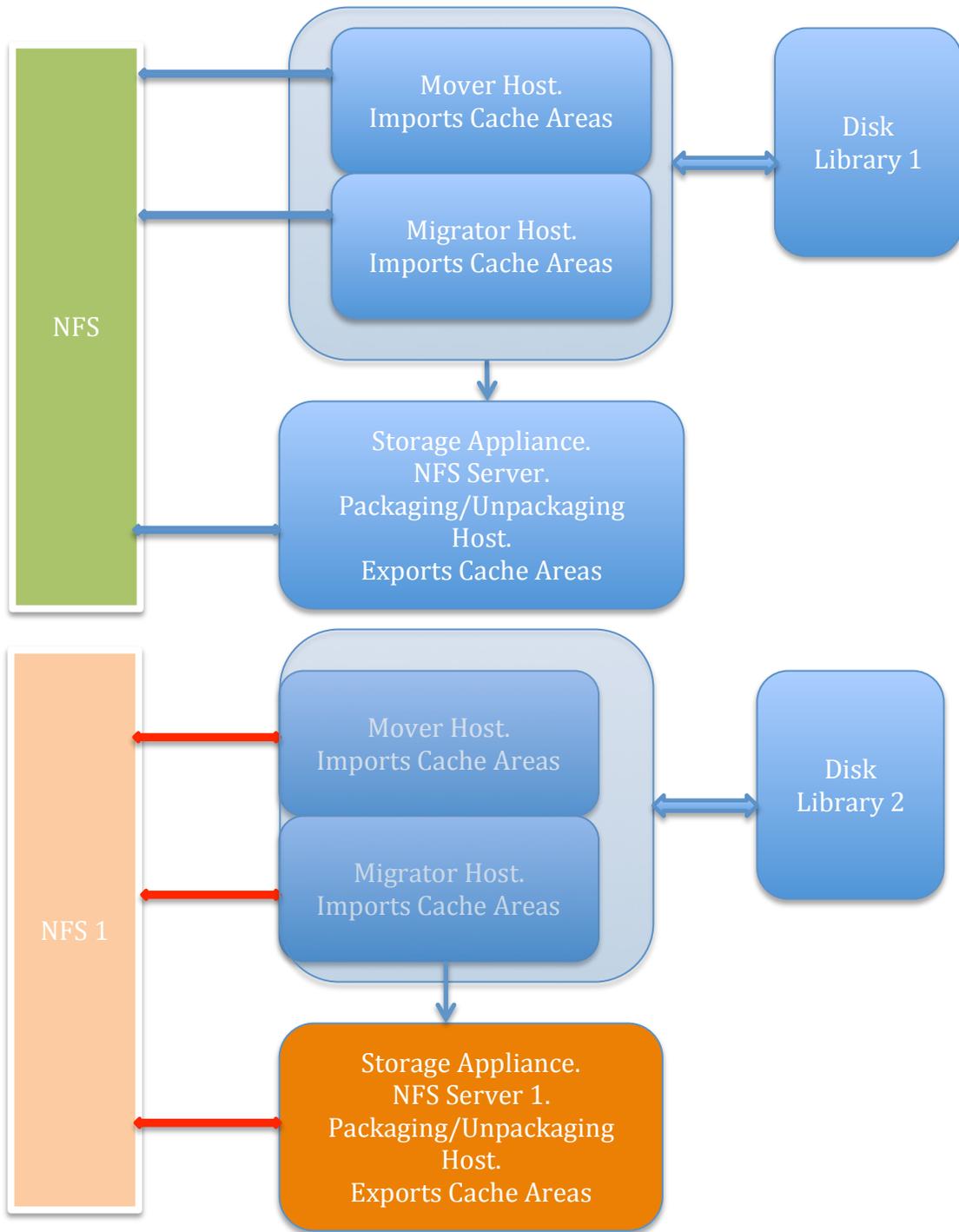


Fig 2. Dual cache server configuration.

More cache servers can be added the same way.

To implement such clustered configuration the library must be added to several SFA components. Below is a functional description of changes.

Write Requests.

As described in [SFA HLD](#) when file gets written into cache by disk mover file clerk generates event "CACHE WRITTEN", this even is sent to dispatcher, which creates and appends a file list based on the values returned by policy. Policy has the information about disk library and was modified to return it. The modified file is `src/lmd_policy_selector.py` (Done). The files then are grouped by dispatcher into file lists to get written to a tape. Such list must also contain the information about disk library. For this reason `cache/messaging/file_list.py` was modified to have disk library as its property.

When file list gets full (according to the corresponding policy) it gets sent by migration dispatcher into a queue in qpid messaging system. In current implementation it is a single queue for all migrators. For the new implementation qpid queues must be different for different cache clusters. To implement such approach migration dispatcher will use qpid exchanges with routing keys. Migrators will use queues named by the name of associated disk library provided in their configurations. These queues will be bound to a migration dispatcher exchange. Below is the illustration.

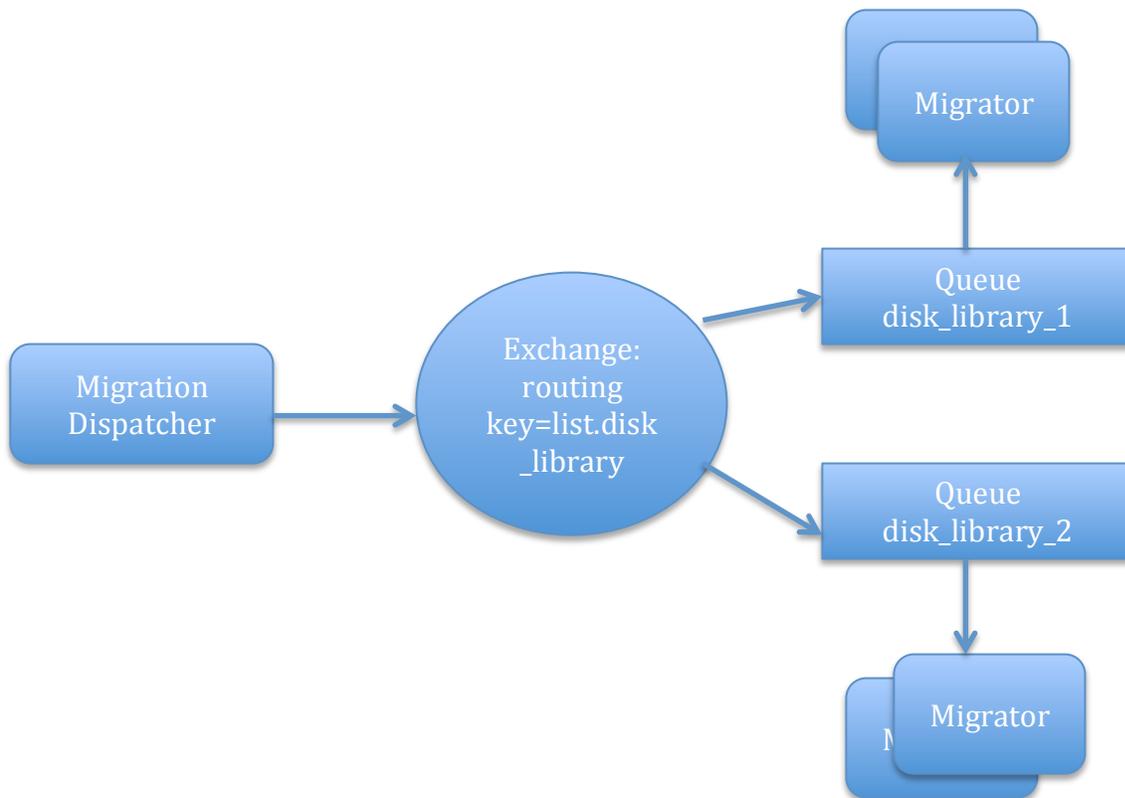


Fig 3. Routing migration dispatcher requests to migrators.

Read requests

If requested file is not in disk cache the corresponding disk library sends `open_bitfile` or `open_bitfile_for_package` request to file clerk. These file clerk methods send `CACHE_MISS` events to dispatcher. To include the library name these file clerk methods will be modified along with `evt_cache_miss_t` in `pe_client.py`. The rest is done as described in “Write requests” (Fig 3).

Purge requests

Purge requests also do not include disk libraries and will be modified to do so. Currently purge requests are sent from migration dispatcher to purge migrators over a separate qpid queue. The new implementation in such case would require to have a separate exchange and queues associated with it for purge migrators just as for write requests. It has been decided to take a different approach and use the same exchange and migrator queues as for write requests (Fig 3).

Resolving migrator performance issues

One of migrator performance and functional issue is: when migrator issues `encp` request it may wait for a very long time for completion of `encp` request. During this time migrator does nothing. To partially resolve this problem the certain amount of migrators per one node can be configured, but better approach is to dynamically “create” a migrator as needed. In such approach only one migrator will be configured per node. The new instance will be spawned as needed. The python multiprocessing module will be used to implement this solution. Multiprocessing in this case is better and easier to use then threading. Here are few reasons:

- processes run in separate name spaces which automatically resolves possible conflicts with variables
- processes are scheduled by OS allowing for effective use of multi-CPU

The main process receives a message from migration dispatcher as described in [“Enstore Small Files Aggregation HLD”](#). It then checks how many processes were spawned. If this number is more than maximum allowed migrator sends back status message `mt.FAILED`. When migration dispatcher receives this message it re-sends it. If mover decides to process received message it starts a separate thread running method `run_mw_request`. This method starts the migration process running method `process_mw_request`. This way of starting processes allows to easily monitor execution of a process by a single thread. Processes communicate with “parent” threads using internal queue. The advantage if queue is that it does not depend on the content of data it transfers. The processes update their states using shared dictionary.

Modifications

Nodes

New disk mover and migrator nodes need to mount a new cache areas.

Enstore configuration

- Leave only one migrator per node (it will dynamically spawn new migrators if needed).
- Remove purging dispatcher (it is no more needed, dispatcher now does all requests).
- Append the information to dispatcher indicating that it serves a clustered cache: "clustered_configuration": True
- Append disk library entries to old migrators: "disk_library": <DISK_LIBRARY>
- Optional migrator parameter: "max_process": <integer number>
- Configure additional migrators for new migrator nodes
- Configure additional disk movers for new disk mover nodes (can be shared with migrators).

Modules

The following modules will be modified.

Module	Status	%% completed
src/cache/messaging/pe_client.py	Done	100
src/cache/messaging/file_list.py	Done	100
src/cache/servers/dispatcher.py	Done	100
src/cache/servers/migration_dispatcher.py	Done	100
src/cache/servers/migrator.py	Done	100
src/file_clerk.py	Done	100
src/lmd_policy_selector.py	Done	100
src/purge_files.py	Done	100