Packaging files in enstore (AKA small files). High Level Design.

Table of Contents

1 Introduction

This project is primarily driven by the need to aggregate small files into bigger packages for more efficient use of tape drives.

Tape drive transfer rates depend on the size of the files. If the size of a file is relatively small then overall average data throughput rate is less than for the relatively big file size. There are several reasons for this: seeking to the files position, starts/ stops etc. The optimal size for a file on tape depends on the tape technology and currently is about 1GB with tendency to grow[1]. The upper limit on the small file depends on a particular tape technology and has tendency to grow. For now and foreseeable future the reasonable limitation is 500 MB. Today's large file is tomorrow's small file

Users can not always easily control the size of files they write to tapes, and many storage systems provide transparent aggregation of files for the user. Enstore does not provide this functionality and this project is intended to provide this. It is of particular interest to existing and upcoming Neutrino experiments, whose data files are typically small.

Another fact to take into considerations is that there are already many files with relatively small sizes stored in the Fermilab enstore tape systems. We are migrating from tapes with small files to tapes with much larger capacity, resulting in tapes with tens of thousands of small files on them. The access to such files can be quite slow and inefficient, tying up valuable resources. To optimize access and transfer rates for relatively small files we need to create a mechanism of packaging such files as a single entity (package), stored on tape, while at the same time permitting transparent access to each file in a package.

Packaging files before writing them to tape requires a disk buffer to aggregate files into a package. Unpacking files from their package retrieved from tape before they get delivered to users requires a disk cache. This disk space can not be requested on the user side because users may not be able to give up a part (sometimes substantial) of their disk space for packaging. This disk cache must be an internal to enstore , allowing to optimize packing / unpacking and delivery of individual files.

This cache/buffer will be used by multiple migrators/stagers to transfer files (packaged or not) between disks and tapes in a distributed environment. Files will arrive from client nodes to the caching system and will be stored on its disks. To provide a flexible environment for such files the caching system should provide access to each file in cache from any host involved in transfer, packaging, migration, staging, and unpacking. Thus the main requirement for such a cache is to provide a global access to any file, which can be achieved by using a clustered file system.

2 Structure of enstore caching system.

The possible structure of integrated caching system is illustrated in fig 1. Such a system will easily

1 We consider an optimal file size the size, which provides the throughput not less than 90% of maximum

scale by adding file servers, packagers, migrators, and stagers and by expanding the global file system. The clustered file system should also sustain simultaneous high data throughput to multiple sources and destinations. Here we intentionally have not specified any details of clients, file servers, tape systems, or global file system. As a good candidate for clustered file the Lustre FS could be considered. It has proven to be quite reliable and high performance distributed global file system, which has a good record of utilization in mid to large data storage systems. Standard client/file server pairs, such as ftp client – server, can be used.

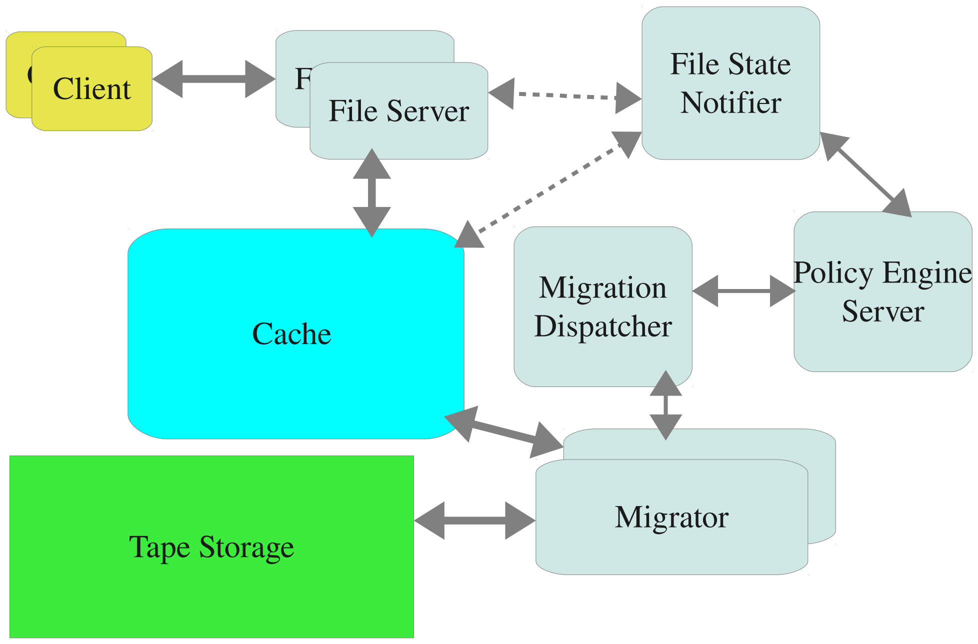

Fig 1. General structure of integrated data caching and tape system

In fig. 1. Data gets transferred between clients and File servers and temporarily stored in cache.

There are 3 types of file cache operations:

2.1 Write file into enstore.

When data is written to cache by File Server the Policy Engine Server (PE Server) receives event from File State Notifier specifying that the **new** file arrived into cache. PE server adds this file into list of files to archive to tape. PE server may have multiple file lists, based on a policy set for each list.

It has 3 types of lists:

1. Archive Lists. List of files to be written to tape.
2. Stage Lists. Lists of files to be staged from tape(s) to cache.
3. Purge Lists. Lists of files to be purged in cache.

Lists may be groups of files belonging to a certain storage group, file family, directory, having certain size limits, time in cache, etc. When policy rule is satisfied for a certain list it gets sent to Migration Dispatcher (MD). Migration Dispatcher distributes file lists between Migrators. Migrator aggregates files into container and writes this container to tape. It then notifies MD that the file was written to tape, which in turn sends the corresponding event to PE Server. PE server adds files written to tape to purge list(s).

2.2 Read file from enstore

When client reads file from enstore, File Server checks if the file is immediately available from cache. If it is available File Server transfers this file to client. If file is not in cache but on tape, File Server contacts Notifier requesting to stage a file and waits for a file. Notifier generates event to PE server. PE Server puts it into stage list. When the rule for this list defines that it is time to stage files, this list is sent to Migration Dispatcher. Migration dispatcher sends list of files to Migrator. Migrator stages requested files from tape to cache. Note, that it will also stage files that happen to be in the package along with requested files. Migrator returns list of staged files to MD, which in turn sends the corresponding event to PE Server. PE Server contacts Notifier with event that requested file was staged. Notifier sends this information back to File Server. File Server transfers the requested file. Policy engine also puts all staged files into purge lists.

2.3 Purge file from cache

When PE Server rule determines that the file in the purge list is to be purged the file gets deleted in cache.

2.4 Structure of enstore caching system with encp clients and disk movers

It was decided to implement enstore caching system with encp clients and disk movers. This implementation allows to reuse reliable data delivery mechanisms already incorporated into enstore, such as CRC calculations, internal retries, etc. The modified data caching system is shown in fig. 2. The combination of Library Manager Director and File Clerk provide for the functionality analogous to File

State Notifier. The Disk mover provides the functionality analogous to File Server from fig. 1.

Communication between enstore caching system components in Fig.2 will be supported by two protocols: Enstore UDP (EUDP) and AMQP. EUDP protocol will be used because some components of enstore caching systems are standard enstore servers and clients. AMQP will be used to communicate with components involving Policy Engine because it is one of its possible communication protocols. For integration of these 2 protocols the EUDP/AMQP Proxy Server will be used. Its role is to receive messages in UDP from existing enstore components and send to new components in AMQP and vice verse. More than one EUDP/AMQP can be configured in the system to scale the system throughput.

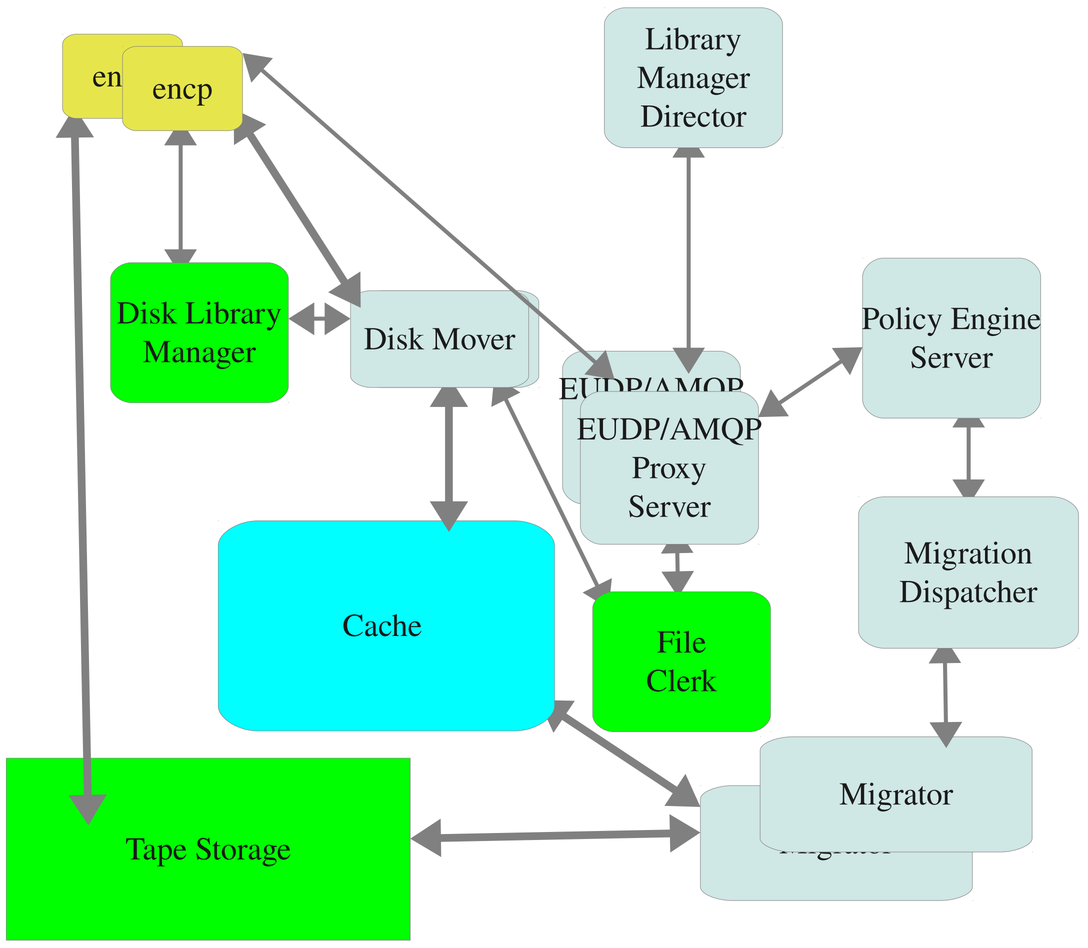

Fig 2. Structure of integrated data caching and tape system using encp and disk movers.

The Library Manager Director (LMD) functionality is to determine whether to send data to tape library directly or to cache first. Encp (client) sends to Library Manager Director request (ticket) containing library name from “library” tag in pnfs directory it tries to write file to. For enstore pnfs tags see ....(ref to doc). Library Manager Director has associated with it Policy Engine with set of rules defining selection of ether tape or disk library manager. The selection rules can be based on different parameters. The main is the file size. Each external encp client must firs contact LMD. Such an approach allows to automatically define which user files may and will go directly to tape and which will be first written to enstore cache.

Very important detail in this structure is that all disk movers are connected to the same clustered file system, so that they all can access any file in cache. Further we will address details of enstore caching system using encp clients and disk movers.

2.5 Hardware for cache

The enstore cache will be implemented as a clustered file system available for access by any migrator. It must be very reliable to reduce to the minimum possible corruption of data for files written to cache and awaiting their migration to tapes. It has been decided to use Oracle ZFS based disk system Advanced HPC File Server Nexenta OS (Open Solaris with Linus user level utilities with 63 TB of disk space, 48 GB memory and 10 Gb Ethernet network interface.

2.6 Files and cache

2.6.1 File names in cache.

Files written into cache by disk mover have their path and names derived from the file pnfs id. Pnfs id consists of 36 hexadecimal digits. To prevent from a single directory in cache from thousand of entries the pnfs file id maps to the corresponding file name according to algorithm:

```
file_id_hex = int("0x"+file_id, 16)
first = "%s"%((file_id_hex & 0xFFF) ^ ( (file_id_hex >> 24) & 0xFFF),)
second = "%s"%((file_id_hex>>12) & 0xFFF,)
path = os.path.join(root, first, second, file_id)
```

So for instance:

```
root = "/data_files"
file_id = "00001E9281CFB7054652B62737ED1ED3B3F6"
return value:
"/data_files/3816/3387/00001E9281CFB7054652B62737ED1ED3B3F6"
```

Thus each path will contain not more than 0xfff (4095) files. And files will be evenly distributed between different directories.

All files get written by disk mover into a temporary directory, with name unique for each disk mover:

/prefix/DN

- where DN is a disk mover name from enstore configuration

When disk mover completes write file transfer into this directory it immediately renames it according to a pnfs id name convention. Only one file can be in this temporary directory for each disk mover.

This approach guaranties that there will be no partially files on disk due to disk mover failures during file transfers (for write operations).

2.6.2 Package file.

Package file contains a files packaged according certain criteria defined by policy and a special file README.1ST with the following format:

List of cached files and original names

cached_file_path_1 file_name_in_the_name_space1 CRC1

cached_file_path_2 file_name_in_the_name_space2 CRC2

....

cached_file_path_N file_name_in_the_name_spaceN CRCN

This file makes package file self describing and allows to recover files from tape to their original locations in cases when enstore database is absent. These cases may be:

1. Loss of the database
2. Transfer of the tape to another system, which may be enstore or some other tape based storage system.

2.6.2.1 Package file name convention

Package files are written and read by encp client, so they are regular enstore and pnfs files.

Package files for archiving (to get written to tape) will be kept in the special archiving area. This file names are:

/<archive>/.package-YYYY-mm-ddT%HH:%MM:%SSZ/.package-YYYY-mm-ddT%HH:%MM:%SSZ.tar

The separate directory for a packaged file is needed to not allow a conflict between package file contents during packaging.

Staged package files will be kept in staging area

/<stage>

where <archive> and <stage> paths to archiving and staging areas.

Archiving and staging areas can be part of enstore cache or just local disk areas on Migrator nodes.

Package files will be kept in the highest pnfs directory common to all files in a package. So if there are 3 files in a package:

/pnfs/d1/d2/d3/f1

/pnfs/d1/d2/d4/f2

/pnfs/d1/d5/d6/f3

Then the package file will be:

/pnfs/d1/.package-YYYY-mm-ddT%HH:%MM:%SSZ/.package-YYYY-mm-ddT%HH:%MM:%SSZ.tar

with time stamp suffix according to ISO8601 time format

Additional entries in **file table of enstore DB**:

packageId - character varying default '' - This entry is a bfid of a package file, indicates that the file is a part of package and can not be accessed as an individual file directly on tape.

PackageFileCounter integer default -1 – counter of active files in package (gets updated only for a package file).

PackageFileNumber integer default 0 – total number of files in a package.

Package can be deleted when ***PackageFileCounter=0.***

2.6.3 Cached file information

If file was written to cache it will have the following additional entries in **file table of enstore DB**:

cache_status character varying default ''
archive_status character varying default ''
cache_mod_time timestamp without time zone
archive_mod_time timestamp without time zone
where:

- cache_status – shows if file is in cache/migration/purge status. It can have the following values:
  - created – file was written to cache
  - purging – file is being purged
  - purged – file was deleted in cache
  - staging – file is being staged to cache from tape
  - cached – file is in cache

  - None – default value
- archive_status – file migration/purge status. It can have the following values:
  - archiving – file is being written to tape. This state is useful for the recovery from failure
  - archived – file was written
  - None – default value
- cache_mod_time – time when cache_status has changed
- archive_mod_time – time when archive_status has changed

New table will be created in file DB to hold a status of files in transition - files_in_transition

bfid ***character varying***

file_status ***character varying default ''***

cache_mod_time timestamp without time zone.

Special purge rules in PE Server define if a certain file needs to get purged.

3 Enstore Caching System Components.

In this section Enstore Section Components shown in Fig. 2 will be described.

3.1 UDP to AMQP proxy servers.

Each enstore UDP to AMQP proxy has two proxy servers: one to convert enstore udp messages to amqp messages and the other to convert amqp replies to enstore UDP reply tickets.

UDP2AMQ proxy server receives enstore UDP messages and uses its payload (enstore ticket) as *content* of AMQP message. The resulting qpid Message is sent to qpid target address "***`target_addr`***" defined in configuration. For instance, messages destined to LMD will be forwarded to qpid queue ***`lmd`***.

The another server component converts qpid replies to enstore UDP messages and sends replies to enstore UDP clients. In the process of original UDP-2-AMQP conversion udp2amq server sets property '***`reply_to`***' of outgoing qpid message to its own qpid queue to receive future qpid reply. The proxy is stateless (as much as enstore UDP server is stateless) and it is expected that qpid reply contains reply address from original enstore ticket ***`ticket['r_a']`*** or ***`ticket['ra']`***. The ***`content`*** of reply qpid ticket is delivered as payload of enstore UDP ticket as is, converting only the dictionary type of ***`ticket['r_a']`*** or/and ***`ticket['ra']`***to tuple as expected by enstore / udp.

3.1.1 UDP2AMQ Proxy Configuration.

Configuration may have one AMQP broker defined as:

```
configdict['amqp_broker'] = {
   'host':enstore_qpid_broker_host,
   'port':5672,
   }
```

'host' – host on which AMQP broker is running

'port' – AMQP broker communication port

Configuration dictionary may have entries for multiple proxy servers. Each entry has description as follows:

```
configdict['my_proxy.proxy_server'] = {
    'host': 'proxyhost.fnal.gov',
    'port' : 7700, # udp server port
    'udp_port': 7710,
    'target_addr':'udp_relay_test'
}
```

'host' – host on which 'my_proxy.proxy_server' is running

'port' - 'my_proxy.proxy_server' communication port as enstore server

'udp_port' – port on which proxy server transfers enstore UDP messages

'target_addr' – queue on which proxy server transfers AMQP messages.

3.1.2 UDP2AMQP Proxy Server enstore commands.

Udp2amqp proxy server is a regular enstore server, monitored by inquisitor and shown on System Status page and Servers page. It can be started and stopped just as other enstore servers using “enstore start/stop” command. It responds to the following commands:

```
[enstore@dmsen02 test_dir]$ enstore udp
Usage:
udp [ -ha --alive --help --retries= --timeout= --usage ] udp_proxy_server
-a, --alive prints message if the server is up or down.
-h, --help prints this message
--retries <ALIVE_RETRIES> number of attempts to resend alive requests
```

--timeout <SECONDS> number of seconds to wait for alive response

--usage prints short help message

3.2 Library Manager Configuration Changes.

Configuration dictionary for each Library Manager may have entry with new key `“use_LMD”` with value containing reference (key) configuration dictionary entry for LMD, e.g.

```
configdict['LTO3.library_manager'] = { …,'use_LMD': 'lmd_proxy',}
```

When `'use_LMD'` entry is present, encp shall contact specified LMD to get configuration entry for Library Manager to be used for transfer. When entry is not present encp acts in a “classical” way.

3.3 Library Manager Director.

The Library Manager Director (LMD) functionality is to determine shall client send data directly to tape library or use file aggregation in cache. External encp clients contact LMD first. LMD decides destination of the transfer: tape or enstore cache. For read operation LMD chooses tape or cache as a future file source. Library Manager Director was conceived to use esper Complex Event Processing engine written in Java. The policy engine has set of rules defined to perform selection of tape or disk based Library Manager. The selection rules can be based on different parameters most importantly file size. Currently Library Manager Director uses simplified policy selector, which prototype was taken from Library Manager. The policy selector has set of rules defined to perform selection of tape or disk based Library Manager.

The LMD is a regular enstore server, monitored by inquisitor and shown on System Status page and Servers page. It can be started and stopped just as other enstore servers using “enstore start/stop” command. The LMD can be used without UDP2AMQP server (see comments to LMD configuration). Its configuration is as:

```
configdict['lm_director'] = {
    'host': 'pmig01.fnal.gov',
    'port': 5602,
    'logname':'LMDSRV',
    'udp_port': 7710, # to replace UDP2AMQP
    'queue_in': 'udp_relay_test', # optional
    'udp_proxy_server': 'lmd.udp_proxy_server', # optional
    'policy_file': "/home/enstore/policy_files/lmd_policy.py"
    }
```

'host' - host on which “lm_director” is running

'port' - “lm_director” communication port as enstore server 'logname' - “lm_director” log name

'udp_proxy_server' – proxy server with which “lm_director” communicates. Do not declare if UDP2AMQP proxy server is not used.

'udp_port' – define this port if UDP2AMQP proxy server is not used.

'queue_in' – AMQP queue name on which “lm_director” communicates with proxy server. Do not declare if UDP2AMQP proxy server is not used.

'policy file' – file containing policies for a given LMD

3.3.1 Policy File Format

Library Manager Director and implemented in Python Policy Engine Server / Migration Dispatcher use the same policy file, although it can be different.

The policy file is a Python dictionary and has a following format:

```
{Library_Manager1: {policy1:
                       {'rule':
                            {'storage_group': 'G1',
                             'file_family': 'F1',
                             'wrapper':'cpio_odc'
                            }
                       'minimal_file_size': 2000000000L
                       'min_files_in_pack': 100,
                       'max_waiting_time': 300,
                       'resulting_library': 'new_library'
                       }
                   {policy2:
                       {
                       .....
                       },
                  ....
                  },
Library_Manager2: { policy1:
                           ....
                         }
```

Here is an example and explanation

```
'LTO3.library_manager':{1: {'rule': {'storage_group': 'G1',
                                  'file_family': 'F1',
                                  'wrapper':'cpio_odc'
                               }
                       'minimal_file_size': 2000000000L
                       'min_files_in_pack': 100,
                       'max_waiting_time': 300,
                       'resulting_library': 'new_library'
                       }
```

'minimal_file_size' - if file is less than this size the file will be aggregated

'min_files_in_pack' - minimal number of files in package,

if total size of files to be aggregated is less than minimal_file_size

and number of files >= min_files_in_pack then files will get packaged

'max_waiting_time' - if time of collection of files for a package exceeds this value (sec),

the files will get packaged

If request comes from encp with library LTO3.library_manager and it satisfies this rule and minimal_file_size conditions (less than the minimum) it will be sent to ' resulting_library'.

3.3.2 Library Manager Director enstore commands

Library Manager Director is a regular enstore server, monitored by inquisitor and shown on System Status page and Servers page. It can be started and stopped just as other enstore servers using "enstore start/stop" command. It responds to the following commands:

```
[enstore@dmsen02 test_dir]$ enstore lmd
Usage: lmd [OPTIONS]...
-a, --alive
```

prints message if the server is up or down.

--do-alarm <DO_ALARM> turns on more alarms

(snip ...)

--load load a new policy file

--retries <ALIVE_RETRIES> number of attempts to resend alive requests

--show - print the current policy in python format

--timeout <SECONDS> number of seconds to wait for alive response

--usage - prints short help message

The options of interest are –load and –show.

3.3.3 Encp Interaction with Library Manager Director.

The new encp switch will be implemented to allow users to select whether they want to use the file aggregation feature: ***enable-redirection.*** The default value of this switch is to not use Library Manager Director to select a library manager. If this switch is specified the encp will try to send a request to a Library Manager Director.

User side encp client extracts library name from `“library”` tag in pnfs directory (namespace EA) where it tries to write file to. For enstore pnfs tags see [to do: ref to doc]. Enstore configuration file will have entry specifying if caching is enabled for concrete tape Library Manager. The configuration dictionary entry for Library Manager may have entry `“use_LMD”`. For backward compatibility, if there is no such entry encp acts in a old way, encp does not contact LMD and it contacts Library Manager directly. When `“use_LMD”` entry is present in configuration its value contains key in configuration dictionary for LMD (see). Encp contacts specified LMD to get configuration entry for Library Manager to be used for transfer. Encp sends ticket to LMD in the same format it usially sends to Library Manager. The ticket contains library name from “library” tag in pnfs directory (namespace EA).

The choice of caching Library Manager is described by LMD Policy Engine rules. As a simple case, each original tape Library Manager has corresponding Caching Library Manager. LMD make decision to cache or not to cache and selects one of LMs from the pair.

The configuration of LMD must have a mapping of original libraries to alternative libraries.

LMD receives ticket from encp, processes the ticket in policy engine, modifies ticket field ***`ticket[“vc”][“library”]`*** if necessary and then sends ticket as a reply back to encp.

LMD sets status filed of the reply. When ticket original ticket is not valid, the ***`ticket['status']`*** contains error `(e_errors.LMD_WRONG_TICKET_FORMAT, detail)` meaning library can not be selected and it is a fatal error.

In the case of success ***`ticket['status']`***is set to `('ok',None`. If caching is not required LMD may leave original library unchanged. Otherwise LMD modifies `ticket['vc']['library']` to contain name of one of enstore Caching Library Managers. In the case of success encp continues transfer with Library Manager specified in the reply ticket.

3.4 Modified File Clerk

Existing File Clerk (FC) must be modified to notify Policy Engine (PE) Server about when files are created in cache and when read file is requested. For the communications with PE Server file clerk will use AMQP API, described in "Messaging HLD". The specific of enstore file write operation is such that the file may not get considered as written into enstore even if mover successfully completed data transfer. File is considered as written into enstore when enstore client (encp) sets the pnfsId of written file in enstore file database by calling a corresponding File Clerk Client (FCC) method, which in its turn send a message to FC. File Clerk will be modified to send CACHE_WRITTEN event to PE Server. A special process (thread) in File Clerk will set timeouts when File Clerk gets a new_bit_file request from disk mover. If set_pnfsid request does not arrive from encp for the corresponding bfId, File Clerk raises an alarm and makes an entry into the list of suspect files on disk. This is needed for the subsequent cleanup of not completed file writes.

For read requests DM checks if requested file is on disk and if not it sends an open request to File Clerk, which sends CACHE_MISS event to PE Server.

Modifications to file clerk.

new_bit_file:

If file was written to disk media:

1. set cache_status=created
2. cache_mod_time=now
3. put bfid, cache_status, cache_mod_time into files_in_transition table
4. notify a timer process (thread) to check this entry in a specified time period

Otherwise:

1. set cache_status="none"

Timer process (thread).

If specified period expires and the corresponding entry is in the table send alarm that file transfer has not completed.

set_pnfsid:

If file was written to disk media:

1. send CACHE_WRITTEN event to PE Server
2. set cache_status=cached
3. notify a timer process that no check for this bfId is needed

4. remove entry from files_in_transition table

Otherwise no changes.

open_bitfile (new method):

1. if cache_status==cached return success
2. otherwise set cache_status=staging
3. send CACHE_MISS event to PE Server
4. return cache_status (and other information in the ticket)

set_cache_status (new method) – modifies cache_status and archive_status

3.5 Policy Engine Server.

Policy Engine Server (PE Server) collects events from other servers (particularly from File Clerk) combines them into the request lists and submits them to Migration Dispatcher. This is why it will be implemented as a part of the process running both PE Server and Migration Dispatcher. The communication between PE Server and Migration Dispatcher will be done via shared memory. The events destined for PE Server are described in the corresponding "Messaging HLD" document. PE Server receives event from File Clerk specifying that the **new** file arrived into cache or the file written to tape needs to get staged from it. PE Server has 3 types of file lists:

1. Archive Lists. List of files to be written to tape.
2. Stage Lists. Lists of files to be staged from tape(s) to cache.
3. Purge Lists. Lists of files to be purged in cache.

Lists may be groups of files belonging to a certain storage group, file family, directory, having certain size limits, time in cache, etc. Lists have states indication what happens with them. This status is:

- filling – list is being populated with new files
- full – list is full
- work – list is sent for the execution
- done – execution was completed successfully
- failed – execution failed.

When policy rule is satisfied for a certain list it gets sent to Migration Dispatcher. This list gets sent to Migration Dispatcher and is marked as "work". Migration Dispatcher sends event to Policy Engine Server, containing file list id and result of the operation: done/failed.

3.5.1 File List Format.

File list format is as follows:

File list ID - unique file list id.

list_item_1[, list_item_2, ... , list_item_N]

where list_item_*i* is:

enstore bit file ID(BFID) – assigned when file gets written into enstore disk cache

name space file id (pnfs Id) – name space (pnfs) id. This allows to avoid contacting file clerk to fetch a file on disk

file path – complete file path. This allows to avoid contacting file clerk.

tape library list – information from pnfs library tag

file CRC

All information about requested file can be obtained from enstore file database referring by BFID.

Tape library list is needed to identify to tape(s) from what library (or libraries for multiple copy) the files will be written of from what tape in what library it will (or may be) staged. The operation type (archive, stage, purge) performed on the list is specified in the message sent to the server.

3.6 Migration Dispatcher.

Migration Dispatcher (MD) receives file lists from Policy Engine Server over shared memory and distributes file lists between Migrators or purges files depending on the operation specified in the list. When migrator replies with result of the work, MD sends corresponding event to Policy Engine Server.

3.6.1 Migration Dispatcher Configuration

Migration Dispatcher configuration is described in enstore configuration file, presents a python dictionary, and contains the following attributes:

'host': - MD host (string)

'port': - MD port (string)

'logname': - MD log name in enstore log (string)

'norestart':' - automatic restart flag (string)

3.6.2 Python implementation of Policy Engine Server and Migration Dispatcher – Dispatcher.

Python implementation of Policy Engine Server and Migration Dispatcher has both these components running in the same process, because they share a lot of information. They run in separate threads.

The Dispatcher is a regular enstore server, monitored by inquisitor and shown on System Status page and Servers page. It can be started and stopped just as other enstore servers using “enstore start/stop” command. Its configuration is as:

```
configdict['dispatcher'] = {
    'host': 'pmig01.fnal.gov',
    'port': 5603,
    'logname':'DISP',
    'queue_work': 'policy_engine',
    'queue_reply': 'file_clerk',
    'migrator_work': 'migrator',
    'migrator_reply': 'migrator_reply',
    'policy_file': "/home/enstore/policy_files/lmd_policy.py",
    'max_time_in_cache': 600,
    'purge_watermarks':(.8, .4),
    }
```

'host': - sever host
'port' - server port

'logname' – server log name

'queue_work': - events (from file clerk) come to this queue

'queue_reply': - replies (if needed) are sent on this queue

'migrator_work': - request lists are sent to this queue

'migrator_reply': - replies from migrators come to this queue

'policy_file': - policy file (same as for Library Manager Director).

'max_time_in_cache': 600 – purge file if it was written max_time_in_cache ago

'purge_watermarks':(.8, .4) – start purging staged files if occupied space is more than .8*Total,

stop purging files if occupied space is less than .4*Total

Dispatcher responds to the following commands:

[enstore@pmig01 src]$ enstore disp

Usage:

disp [OPTIONS]...

```
  -a, --alive              prints message if the server is up or down.
      --do-alarm <DO_ALARM>  turns on more alarms
      --do-log <DO_LOG>     turns on more verbose logging
      --do-print <DO_PRINT>  turns on more verbose output
      --dont-alarm <DONT_ALARM>  turns off more alarms
      --dont-log <DONT_LOG>  turns off more verbose logging
      --dont-print <DONT_PRINT>  turns off more verbose output
      --get-queue          print content of pools
  -h, --help               prints this message
      --load               load a new policy file
      --retries <ALIVE_RETRIES>  number of attempts to resend alive requests
      --show               print the current policy in python format
      --timeout <SECONDS>   number of seconds to wait for alive response
      --usage              prints short help message
```

The options of interest are –load, –show (same as for Library Manager Director) and –get-queue.

Here is an example output of "enstore disp –get" command:

```
{'cache_missed': {},
 'cache_purge': {},
 'cache_written': {},
 'migration_pool': {'005cc2f6-95aa-40d3-9565-013cfbbaf903': {'id': '005cc2f6-95aa-40d3-9565-013cfbbaf903',
                                        'list': [{'bfid': 'GCMS132950908800000',
                                                  'complete_crc': 2548854233,
                                                  'libraries': ['LTO3GS'],
                                                  'nsid': '0000DCDC7B5FC2254F5088630204A8D06406',
                                                  'path': '/pnfs/data2/file_aggregation/LTO3/john_f
amily2/variousSizes-Feb-17/200m.dm9a.copy-0175'},
                                                 {'bfid': 'GCMS132950908900000',
                                                  'complete_crc': 4021464339,
                                                  'libraries': ['LTO3GS'],
                                                  'nsid': '000023351D17A4BF4B90891E2DB44CF43841',
                                                  'path': '/pnfs/data2/file_aggregation/LTO3/john_f
```

```
amily2/variousSizes-Feb-17/613k.dm6a.copy-0116'},
                                                 {'bfid': 'GCMS132950909300000',
                                                  'complete_crc': 3136921648,
                                                  'libraries': ['LTO3GS'],
                                                  'nsid': '000078A625C4D88A422B87E58B51FBF72F00',
                                                  'path': '/pnfs/data2/file_aggregation/LTO3/john_f
amily2/variousSizes-Feb-17/25m.dm9a.copy-0175'},
                                                 {'bfid': 'GCMS132950909400000',
                                                  'complete_crc': 2851503227,
                                                  'libraries': ['LTO3GS'],
                                                  'nsid': '0000795EC4F491F54782B0CE89FA2EFDFB21',
                                                  'path': '/pnfs/data2/file_aggregation/LTO3/john_f
amily2/variousSizes-Feb-17/100m.dm8a.copy-0184'},
                                                 {'bfid': 'GCMS132950910000000',
                                                  'complete_crc': 0,
                                                  'libraries': ['LTO3GS'],
                                                  'nsid': '00002EF339597A3B4FD4BB48F543ACF5B994',
                                                  'path': '/pnfs/data2/file_aggregation/LTO3/john_f
amily2/variousSizes-Feb-17/10m.dm8a.copy-0184'},
                                                 {'bfid': 'GCMS132950910200000',
                                                  'complete_crc': 0,
                                                  'libraries': ['LTO3GS'],
                                                  'nsid': '00003C231D4D72D44251B55A46471AC02A8A',
                                                  'path': '/pnfs/data2/file_aggregation/LTO3/john_f
amily2/variousSizes-Feb-17/500m.dm9a.copy-0175'},
                                                 {'bfid': 'GCMS132950910400000',
                                                  'complete_crc': 0,
                                                  'libraries': ['LTO3GS'],
                                                  'nsid': '0000D7B8FC043D4344C78695D60080810AC8',
                                                  'path': '/pnfs/data2/file_aggregation/LTO3/john_f
amily2/variousSizes-Feb-17/750m.dm6a.copy-0116'},
                                                 {'bfid': 'GCMS132950911100000',
                                                  'complete_crc': 1335146173,
                                'libraries': ['LTO3GS'],
                                'libraries': ['LTO3GS'],
                                'nsid': '000020A8843390D14A37B2AA52A40D607A19',
amily2/variousSizes-Feb-17/50m.dm9a.copy-0175'},
                               {'bfid': 'GCMS132950912500000',
                                'complete_crc': 766913875,
                                'libraries': ['LTO3GS'],
                                'nsid': '00001B0E43C14CED4CA3B857BA2733B89592',
amily2/variousSizes-Feb-17/966k.dm6a.copy-0116'},
                               {'bfid': 'GCMS132950912100000',
                                'complete_crc': 0,
                                'libraries': ['LTO3GS'],
                                'nsid': '0000B1008542814645B4BBB254C890BCB2E9',
amily2/variousSizes-Feb-17/750m.dm9a.copy-0175'}],
                            'time_qd': 'Fri Feb 17 14:05:29 2012',
                            'type': u'CACHE_WRITTEN'},
}
```

3.7 Migrator.

As has been described above Migrator is responsible for moving files between Enstore disk cache and tapes in both directions. When files are being written to Enstore. Migrator receives a list of files from Migration Dispatcher. If this list is a list of files to be written to tape, Migrator aggregates these files into container and writes this container to tape. It then notifies MD that the file was written to tape, for each file in the list. If list, received from Migration Dispatcher is a list of files to be staged from tape Migrator stages requested files from tape to cache. Migrator will also stage files that happen to be in the package along with requested files.

3.7.1 Migrator configuration and enstore commands.

Migrator is a regular enstore server, monitored by inquisitor and shown on System Status page and Servers page. It can be started and stopped just as other enstore servers using “enstore start/stop” command. Migrator configuration is described in enstore configuration file, presents a python dictionary, and contains the following attributes:

'host': - migrator host name (string)

'port': - migrator port number (integer)

'logname': - migrator name in the enstore log file (string) 'migration_dispatcher' – pointer to migration dispatcher structure in enstore configuration (keyword)

'data_area' – disk area where files are stored

'archive_area' – disk area where archive is created during data archiving (string)

'stage_area' – disk area where archived files get staged from tape (string)

'tmp_stage_area' – disk area where staged files are temporarily stored (string)

'packages_dir' - directory in name space for packages

'dismount_delay' – delay for dismounting tape

'check_written_file' - if greater than 0, then randomly check files written using this number as the mean (default value:0 - don't check)

'tar_blocking_factor' – blocking factor for archiver (tar).

Migrator responds to the following enstore commands:

```
[enstore@enmvr005 ~]$ enstore mig
Usage:
     mig [ -ha --alive --help --retries= --timeout= --usage ] migrator_name
  -a, --alive             prints message if the server is up or down.
  -h, --help              prints this message
     --retries <ALIVE_RETRIES>  number of attempts to resend alive requests
     --timeout <SECONDS>   number of seconds to wait for alive response
     --usage              prints short help message
```

Let's consider Migrator functionality for writing cached files to tape and for staging files from tape to cache.

3.7.2 Writing cached files to tape(s).

Migrator creates an archive file with name /<archive>/.package-YYYY-mm-ddT%HH:%MM:%SSZ according to 2.5.2. It then writes this archive file to tape using encp and library defined in name space of library tag with pnfs name defined according to 2.5.2.

After archive file was successfully write Migrator modifies File DB record for each file in archive:

packageId = archive bfId

cache_status=archived
cache_mod_time=NOW
The entry for archive file gets also modified:
PackageFileCounter = *Number of files in archive*

PackageFileNumber =*Number of files in archive*

The entries in file_copies_map get created as follows:

bfid	alt bfid
bfId1	Archive bfId
bfId2	Archive bfId
...	...
bfIdN	Archive bfId

This File Copies Map will be used for staging files from tape.

Reading archived files.

When Migrator receives a list of files to get staged it identifies archive files using File Copies Map. It then stages archive file and marks all files in archive as staged:

archive_status=cached

archive_mod_time=NOW

4 Detailed description of read write request processing.

4.1 Write request processing of files written via cache

1. Encp sends write request to Library Manager Director (LMD) to get library manager to send a request to. The ticket is the one which was was originally sent to Library Manager.
2. LMD defines the library manager to send a request to, according to policy and sends reply to encp with modified (if required by policy) ticket[“vc”][“library”] and the name of the original library in ticket[“vc”][“original_library”]
3. Encp sends write request to Library Manger (LM) defined by ticket[“vc”][“library”]
4. Disk Mover (DM) sends a request for work to LM.
5. LM sends write_to_hsm work to DM
6. DM transfers data from encp to disk.
7. DM creates new bit file calling new_bit_file method of File Clerk Client.
8. File Clerk creates bfId, sets cache_status to created, makes entry in file_in_transition table and starts waiting for create_pnfsId call from encp
9. Encp completes file operations and calls   create_pnfsId FC Client method

10. FC sets pnfsId, sets cache_status to cached, removes entry from file_in_transition table and sends CACHE_WRITTEN event to PE Server
11. PE Server creates lists of of events grouped according policy. The policy can be “group all files with certain volume family if the size of the individual file is less than specified”, “immediately migrate the file”, etc.
12. When list fills in (satisfies fill in criteria, such as size of all files in the list) it gets sent to Migration Dispatcher.
13. Migration dispatcher sends this list to common queue and sets all files' cache_status to “archiving” in enstoreDB. Then it moves the list into active lists internally.
14. Migrator pulls list from common queue, packs files into package if necessary and sends the package to tape using encp. The package gets written to tape according to pnfs tags, common to files in the package (as described in 2.5.2
15. Migrator modifies enstore DB according to 3.7.2

4.2 Read request processing.

1. Encp gets the library manager from the file's record in enstore DB (this is a standard way for encp, no changes are needed). It hen send a request to library manager.
2. If request comes to tape Library Manager, it gets sent to tape mover (when request for work comes from the mover). It then gets transferred to encp (client).
3. If request comes to disk Library Manager, it checks whether the file is in cache using File Clerk information contained in the incoming request.
4. If file is in cache (cache_status == cached) the request gets sent to disk mover (when request for work comes from the mover). It then gets transferred to encp (client).
5. If file is not in cache and its cache_status != cached then there could be the following scenarios:
    1. cache_status == purged: request gets sent to disk mover
    2. cache_status == staging: This means that the file is being processed by one of migrators or is a part of package, one of which files is being processed by migrator. If disk LM can not find this file in at_movers list it moves this file in on_hold list.
        1. When migrator completes file stage from tape it sets cache_status == cached
        2. It then sends the stage confirmation to disk mover if there is one, waiting for a file, and to Disk Library Manager
        3. Disk Library Manager identifies staged file(s) in on_hold list and moves them back to request queue.

5 Reliability issues.

The File Aggregation feature must provide a reliable delivery of files from user to tape and back. One the most important issues here is a guaranty that the file in cache will be written to tape within a certain period of. In this chapter a different aspects of the system reliability will be described.

5.1 Recovery from the failure during migration from cache to file.

This section describes the case when there was a failure of the Policy Engine Server caused by any reason including corruption of write request lists. This recovery will be done by a file clerk resending events based on a file transition table.

6 Administration.

6.1 Enstore cache command line interface

The purpose enstore cache command line interface is to provide administrators of the system with tool allowing to control a file aggregation feature The format of the command is:

enstore cache [OPTIONS [PARAMETERS]]

Available options and parameters:

--archive

This option requires additional OPTIONS:

--all - flush all write lists to tape(s)

--sg STORAGE_GROUP– flush all pending write lists for a specified storage group to tape(s)

--vf VOLUME_FAMILY -- flush all pending write lists for a specified volume family to tape(s)

--stage

This option requires additional OPTIONS:

--all – stage all read lists from tape(s)

--sg STORAGE_GROUP– stage all pending read lists for a specified storage group from tape(s)

--vf VOLUME_FAMILY -- stage all pending read lists for a specified volume family from tape(s)

--flush – both –archive and –stage with the same additional options.

6.2 Emergency data migration to tape.

Some system operations, such as scheduled or unscheduled system shutdowns may require flushing all files not yet written to tape to get written even if criteria defined in the Policy Engine Server were not met. The special enstore command and the corresponding event will be designed for this.

Enstore command:

enstore cache –flush –all - this command instructs the policy engine to send all its current lists for migration to tapes.

6.3 Migration to new media.

In enstore there is a need of consistent migration of data form one tape media to another as the old media and drives get replaced with new technology. The natural way of providing such migration for packaged files is to stage them to cache with subsequent repackaging and archiving to a new tape. This approach does not require any changes in migration scripts.

7 Document Change log

v13	02/16/11	Alex	Updated Caching Components section to describe interaction and configuration changes. Took out operation from File List Format.
V14	03/16/11	Alex	Change order of operations in set_pnfsid() to ensure recovery after crash.
V15	06/29/11	Sasha	1. Changed package file path in cache. 2. Changed file clerk section to distinguish files written via cache or directly to tape. 3. Changed “Writing cached files to tape(s)”.
V16	09/16/11	Sasha	1. Added detailed description of package file 2. Modified “Read request processing” section.
V17	10/06/11	Sasha	1. Added reliability issues. 2. Added administration section 3. Changes according to comments from Gene.
v18		Sasha	
v19		Sasha	
v20		Sasha	
v21	08/27/12	Sasha	Added description for configuration without UDP2AMQP server