

# Messaging HLD for Enstore Small Files Aggregation Project

Alex Kulyavtsev

6/18/2010 v0.10

## 1 Introduction

To implement Small Files Aggregation feature in Enstore we introduce several new components:

### **Disk Mover (DM)**

- cache frontend, the component which reads files from or writes files to cache. In the future it can be any other Data Delivery Service (DDS) component.

### **Policy Engine (PE)**

- the engine to implement business logic, e.g. to reorder requests to group files on write.

### **Migration Dispatcher (MD)**

- central component to dispatch execution of work items (archive file, stage file, ... ) to Migrators and track execution of requests.

### **Migrator**

- one of several workers distributed among Migrator Nodes responsible for execution of one item of work: file aggregation and/or writing and reading data containers to/from tape backend. Migrators execute vanilla `ncp` to perform data transfer to/from tape.

To implement interaction between newly added enstore components for Small Files Aggregation feature we plan to use Advanced Message Queuing Protocol (AMQP). The other possibility is to use enstore UDP to provide communication Migration Dispatcher and Migrator, to be described elsewhere. AMQP standardizes both messaging and wire protocol and provides standard way to implement fast, secure and reliable communication in language independent way. Enstore is written in Python and most Open Source Complex Event Processing Engines to be used as Policy Engine are written in Java.

## 2 Functional specification

AMQP specifies datatypes on the wire as various integer and float data types, timestamp, UUID and also complex data types such as map and list. Complex datatypes can be nested. This allows simple transfer of nested python dictionaries used in enstore messages. On java side python dictionaries are converted transparently to java Maps. Java maps can be used by Esper CEP engine to represent CEP events. The other alternative is to use XML representation both in messages and PE, this will allow message format check and simple message filtering by AMQP brokers. Term “map” used in discussion below can be substituted by “dictionary” when used by python client.

We use standard AMQP features (message routing, reliable delivery, persistence, security, etc). AMQP message consists of *Message Header* and *Message Body*. The *Message Body* is opaque and AMQP does not specify or care about its properties. *Message Header* consists of fields specific for AMQP protocol. *Message Header* has `message_properties.application_headers` map where application can put any information. At present we intend to put Enstore specific message fields

into `application_headers` map. The *Message Body* can be empty or we may use *Message Body* field for bulk messages and/or when fast access to message payload is not required by messaging system to route event. E.g. we may put full encp ticket to message body thus full original ticket is available and we copy few fields of original ticket to message header for quick access by logic.

## 2.1 AMQP Python quick example

The following is quick code example what data structures are available in AMQP and how message is sent :

### Example 1

```
#...
import qpid
from qpid.datatypes import Message
# ...more...

nested_dictionary = {
    "string": "stringValue",
    "int": 1234,
    "long": 2**32,
    "map": {"string": "nested map"},
    "list": [1, "two", 3.0, -4]
}

#...skip ... get amqp session here ...

# Create some messages and put them on the broker.
dp = session.delivery_properties(routing_key="this_message_destination")
message_properties = session.message_properties()
message_properties.content_encoding = "amqp/map"

# set application header :
message_properties.application_headers = {}
message_properties.application_headers["my_nested_dictionary"] = nested_dictionary
message_properties.application_headers["my_tuple"] = (12345, 54321, 'hello!')
# create and send message. Message body can be empty.
msg = Message(message_properties, dp, "this is text message body")
session.message_transfer(destination="amq.direct", message=msg)
```

### 3 Enstore Message Properties

Now we define map (nested dictionary) `enmsg` in `amqp` application header to serve as enstore *event* or *command*. The commands will be represented as constant strings in all capital letters as value.

- `msg_type` - component specific type of the message specifying payload, some kind of *command* or *event* :
  - *command* - command send to the peer.
  - *event* - event generated in response to completed action or change of state
- `msg_ver` - tuple (int major, int minor)

#### Example 2

```
mp = session.message_properties()
mp.application_headers = {}
m = {}
m["msg_type"] = "MD_COMMAND"
m["msg_ver"] = (1, 0)
m["command"] = "MD_ARCHIVE_FILES"
myArgs = {"a1":"v1", "a2":123 }
m["args"] = myArgs
mp.application_headers["enmsg"] = m
```

### 4 Addressing

Policy Engine and Migration Dispatcher are singletons. They read messages from queue bonded to the direct exchange. There are multiple Migrators, probably with different properties. A group of Migrators with similar properties can read work-assigning messages from the same queue to implement load balancing and HA.

The initial command assigning work is sent to direct exchange by message producer. Worker replies are sent directly to sender using request-response mechanism described in “Server Application” section of MRG Tutorial [MRG Tutorial]. Message `routing_key` specifies destination *amqp node* (message consumer process), for example `migration_dispatcher`, `policy_engine`, “any” enstore file cache migrator `fc_migrator` or concrete file cache migrator `fc_migrator.mvr1234` (name includes “.” to separate fields).

## 5 Component Interaction Through AMQP Messaging

Messages specs are grouped below by message destination.

### 5.1 Messages sent to Policy Engine.

Data Delivery Service (enstore Disk Mover, DM) can send messages consumed by Policy Engine (PE). PE reacts on events issued by Migrators.

#### 5.1.1 Description

Event reflects changes in local cache or user namespace.

#### 5.1.2 Parameters

`msg_type` : `FC_EVENT` // File Cache event

*event* :

- `CACHE_WRITTEN` // file replica written to cache by client (enstore Disk Mover)
- `CACHE_MISS` // client attempts to read file from cache and file not found in cache. Triggers restore from tape & unpack (DM)
- `CACHE_PURGED` // file copy removed from cache (Migrator)
- `CACHE_STAGED` // file restored from tape to cache (Migrator)

`msg_type` : `NS_EVENT` // Namespace event

- `FILE_DELETED` // file is deleted in user namespace (Disk Mover)

#### 5.1.3 Detailed Parameters Description

`msg_type` : `FC_EVENT` // file cache event

*event* :

- `CACHE_WRITTEN`
  - meaning: file written to cache, signals end of data write operation.
  - when: `close()` on write
  - action: aggregate write requests and prepare list of file to be written. Send *command* `ARCHIVE_FILES` to MD when needed.
- `CACHE_MISS`
  - when: `open()` on read with cache miss - file not found in cache
  - action: send *command* `STAGE_FILES` to MD to read files from tape and unpack files.

- `CACHE_RESTORED (FILE_MIGRATED ?)`
  - meaning : file restored from tape to cache
  - action: release pending read transfers from file cache to user
- `CACHE_PURGED`
  - meaning: file copy removed from cache
  - action: delete pending requests to store file if any. File can be perged only if it has been written to tape, otherwise this shall generate error reported by monitoring.
- `FILE_DELETED`
  - meaning: file deleted in user namespace
  - action: clear cache entry, delete pending requests to aggregate file. If multiple file aggregation started, mark file as deleted but do not abort aggregating files.

## 5.2 Policy Engine communication with Migration Dispatcher

The folowing are commands executed by Migration Dispatcher and confirmation message sent out by MD to signal operation completion.

### 5.2.1 Parameters

`msg_type` : `MD_COMMAND`

*command* :

- `MD_ARCHIVE_FILES` // Package and Write to tape
- `MD_STAGE_FILES` // Read from tape and Unpack
- `MD_PURGE_FILES` // Purge cache entry

`msg_type` : `MD_REPLY`

*reply* :

- `MD_FILES_ARCHIVED`
- `MD_FILES_STAGED`
- `MD_FILES_PURGED`

### 5.2.2 Description

Policy Engine sends *command* to Migration Dispatcher. Migration Dispatcher gets message and “accepts” the message (sends acknowledge) and then work is executed asynchronously. After completion of the work Migration Dispatcher sends reply message to report operation completion. The

reply message contains message ID of the original message it is reply to.

## 5.3 Migration Dispatcher communication with Migrator

### 5.3.1 Parameters

`msg_type` : MIGRATOR\_COMMAND

*command* :

- `MG_ARCHIVE_FILES` // Package and Write package file to tape
- `MG_STAGE_FILES` // Read from tape and Unpack if needed
- `MG_PURGE_FILES` // Purge cache entry
- `MG_REPORT_PROGRESS` // Direct message - query worker status and transfer progress

### 5.3.2 Description. Commands sent by Migration Dispatcher.

Operations on list of files where list may consist of single file. Migration Dispatcher controls file packing/unpacking and also file transfer operations to/from tape by encp. MD sends commands above to work queue where it read by Migrators. When the message retrieved from queue and work started, Migrator sends message directly to Migrator informing it with direct address for communications. Migrator Dispatcher may send query command to Migrator to check liveness and progress of the transfer and Migrator replies to query command directly to Migration Dispatcher. When transfer finished, Migrator sends final message reporting the end of transfer and error status.

### 5.3.3 Description. Events and replies issued by Tape Backend interface (Migrators)

Migrator sends out event to signal operation completion when the operation is completed with success or error. These events correspond to commands sent by Migration dispatcher to Migrators. The event is sent asynchronously through direct exchange to original command source. The message has reference to original command event and reports error code and error detail of the operation.

### 5.3.4 Parameters

`msg_type` : MIGRATOR\_REPLY

*reply* :

- `MIGRATOR_STATUS` // Direct message. Message reporting progress of work in reply to // command "`MG_REPORT_PROGRESS`"

```
msg_type : MIGRATOR_DONE // Direct message. Final message reporting completion of work.
  event :
  • FILE_ARCHIVED

  • FILE_STAGED

  • FILE_PURGED
```

### 5.3.5 Return value

```
output : tuple error = [ierr, err_msg]
  int ierr // error code ierr == 0 is success.
  string err_msg // error message
```

## 6 Detailed Message Descriptions

### 6.1 CACHE\_WRITTEN and CACHE\_MISS events

CACHE\_WRITTEN and CACHE\_MISS events are used as input for policy decisions and potentially carry more information compared to other messages. These events lead to file archiving or staging. We provide following information in CACHE\_WRITTEN and CACHE\_MISS events to Policy Engine to group and/or prioritize requests. Most of the message fields we use below are based on the fields of enstore ticket. Some fields for write operation are not known at the time when message sent out and thus not set as indicated below, e.g. bfid for write operation. The vanilla encp ticket is included in message body, the following fields are extracted into message header to be easily accessed (see next page):

```

ticket = {
  'fc': {
    'arch': {
      'id': 'cdf',
      'type': 'enstore'
    }
    'ns' : {
      'id': 'cdf',
      'type': 'pnfs',
      'mnt': '/pnfs/fnal.gov'
    }
    'en' : {
      'node': 'cache01',
      'mount': '/mnt/cache/',
      'path': '000E/0000/0000/0000/095D',
      'name': '000E000000000000095D5220',
      'id': '0x12345'
    }
  },

  'file' : {
    'name': '/pnfs/fs/usr/Migration/cms/WAX/11/file.root',
    'id': '000E000000000000095D5220',
    'size': 663748608L,
    'crc_adler32': 3298525413L,
  },

  'enstore' : {
    'bfid': 'CDMS115785728500000',
    'vc': {
      'library': 'CD-LT04G1',
      'storage_group': 'cms',
      'file_family': 'Commissioning08',
      'wrapper': 'cpio_odc',
      'file_family_width': '2',
      'external_label': 'VOM563',
      'volume_family': 'cms.Commissioning08.cpio_odc',
    },
    'location_cookie': '0000_0000000000_0000174'
  }
}

```

Issues :

- there is no crc in enstore write ticket.

## 6.2 CACHE\_PURGED

File has been purged on disk cache.

The ticket is same as CACHE\_WRITTEN event, “enstore” dictionary entry is not required.

## 6.3 CACHE\_STAGED

File has been staged to disk cache. The ticket is same as CACHE\_MISS event, “enstore” dictionary entry is not required.

## References

[MRG Tutorial] Jonathan Robie, "Red Hat Enterprise MRG 1.1 Messaging Tutorial AMQP", Red Hat, Inc, 2008