

# Enstore Small Files Aggregation Messaging DLD

Alex Kulyavtsev

3/16/2011 v0.13

## 1 Introduction

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

### **Library Manager Dispatcher (LMD)**

- Library Manager Dispatcher selects Library Manager (file cache or tape bound) to be used by encp.

### **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 busines logic, e.g. to reorder requests to group files on write.

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

- central component to dispatch execution of work items (archive, stage or purge file list ) to Migration Workerrrs and track execution of requests.

### **Migrator, or Migration Worker (MW)**

- 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 encp to perform data transfer to/from tape.

To impement interaction between newly added enstore components for Small Files Aggregation feature we plan to use Adavance Message Queing Protocol (AMQP). AMQP standartizes 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.

Standard Enstore uses UDP to provide communication between enstore components. To simplify interaction between legacy enstore components and new components using AMQP we introduce UDP to AMP proxy servers converting Enstore UDP tickets to AMQP messages and converting AMQP replies back to enstore UDP tickets.

## 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 Low Level Python API Quick Example

The following is quick code example what data structures are available in qpid 0.6 AMQP implementation and how the 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 [Qpid Pr][Qpid UG]. 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.mgr1234` (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

Namespace event:

msg\_type :

- FILE\_DELETED – file is deleted in user namespace (Disk Mover)

File Cache events:

msg\_type :

- CACHE\_WRITTEN – file replica written to cache by client (enstore Disk Mover)
- CACHE\_MISSED – 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 staged from tape to cache (Migrator)

#### 5.1.3 Detailed Parameters Description

msg\_type :

- 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\_MISSED
  - 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\_STAGED
  - 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 purged 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 following are commands executed by Migration Dispatcher and confirmation message sent out by MD to signal operation completion.

### 5.2.1 Parameters

Commands:

msg\_type :

- MDC\_ARCHIVE – Package file list and Write to tape
- MDC\_STAGE – Read from tape and Unpack file list
- MDC\_PURGE – Purge cache entries (file list)

Replies:

msg\_type :

- MDR\_ARCHIVED – result of execution of MDC\_ARCHIVE, files are archived or archive failed.
- MDR\_STAGED – result of execution of MDC\_STAGE, files are staged or stage failed.
- MDR\_PURGED – result of execution of MDC\_PURGE, files are staged or purge failed.

### 5.2.2 Description

Policy Engine sends *command* to Migration Dispatcher. Migration Dispatcher fetches the message and stores the message on hash, “*acknowledges*” the original message and then work is executed

asynchronously. Migration Dispatcher sends message to Migrators (Migration Workers) and they actually execute the work. Migration Dispatcher tracks worklist execution and may request worker status of request processing. After completion of the work Migration Dispatcher sends reply message to PE 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

Migration Worker commands:

msg\_type:

- MDW\_ARCHIVE – Package file list and Write to tape
- MDW\_STAGE – Read from tape and Unpack file list
- MDW\_PURGE – Purge cache entries (file list)
- MDW\_STATUS – query worker status and transfer progress (sent directly to worker)

### 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. Migration 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

Migration Worker replies.

Final message reporting completion of work:

`msg_type`:

- `MDR_ARCHIVED` – file list archival completed or failed
- `MDR_STAGED` – file list staged
- `MDR_PURGED` – file list purged

Message reporting progress of work :

- `MDR_STATUS` – worker status and transfer progress

### 5.3.5 Return value

`output : list status = [err, err_msg]`

`string err` – error code, from `e_errors`

`string err_msg` – error message

## 6 Detailed Message Descriptions

### 6.1 CACHE\_WRITTEN and CACHE\_MISSED events

`CACHE_WRITTEN` and `CACHE_MISSED` events are used as input for policy decisions. These events cause file archiving or staging. We provide following information in `CACHE_WRITTEN` and `CACHE_MISSED` events to Policy Engine to group and/or prioritize requests. Most of the message fields 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 these entries are 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).

Issues :

- there is no `crc` in enstore write ticket.

```

ticket_missed = {
  'cache': {
    # -- File Cache specific fields
    'arch': {
      # archive (tape backend)
      'id': 'cdf', # archive name
      'type': 'enstore' # archive type
    },
    'ns': {
      # user namespace (frontend)
      'id': 'cdf', # namespace identification
      'type': 'pnfs', # user namespace description
      'mnt': '/pnfs/fnal.gov' # mount point in global namespace
    }
  },
  'file': {
    # -- data file specific fields
    # the original file name :
    'name': '/pnfs/fs/usr/Migration/cms/WAX/11/file.root',
    # file ID in user namespace. String.
    'id': '000E0000000000000095D5220'
    'size': 663748608L, # file size in bytes
    # Checksum type (key) and value :
    'crc_adler32': 3298525413L, # (RD only)
  },
  'enstore': {
    # -- Enstore backend specific fields
    'bfid': 'CDMS115785728500000', # (RD only)
    'vc': {
      # from Volume Clerk :
      'library': 'CD-LT04G1', #
      'storage_group': 'cms', #
      'file_family': 'Commissioning08', #
      'wrapper': 'cpio_odc', #
      'file_family_width': '2', #
      'external_label': 'VOM563' # (RD only)
      'volume_family': 'cms.Commissioning08.cpio_odc', # NA for W
    },
    # file location on tape from FC, (RD only)
    'location_cookie': '0000_000000000_0000174'
  }
}
}

```

```

ticket_written = {
  'cache': {
    # -- File Cache specific fields
    'arch': {
      # archive (tape backend)
      'id': 'cdf', # archive name
      'type': 'enstore' # archive type
    }
    'ns' : {
      # user namespace (frontend)
      'id': 'cdf', # namespace identification
      'type': 'pnfs', # user namespace description
      'mnt': '/pnfs/fnal.gov' # mount point in global namespace
    }
    'en' : {
      # enstore file cache
      'fsfn' : "node:/mount/path/filename" # fully specified file
      # name constructed from items below
      'node': 'cache01', # node name / address
      'mount': '/mnt/cache', # mount point of enstore cache
      'path': '000E/0000/0000/0000/095D', # cached file subpath
      'name': '000E00000000000000095D5220', # file name in cache
      'id': '0x12345' # file handle in cache
    }
  },

  'file' : {
    # -- data file specific fields
    # the original file name :
    'name': '/pnfs/fs/usr/Migration/cms/WAX/11/file.root',
    # file ID in user namespace. String.
    'id': '000E00000000000000095D5220'
    'size': 663748608L, # file size in bytes
  },

  'enstore' : {
    # -- Enstore backend specific fields
    'vc': {
      # from Volume Clerk :
      'library': 'CD-LT04G1', #
      'storage_group': 'cms', #
      'file_family': 'Commissioning08', #
      'wrapper': 'cpio_odc', #
      'file_family_width': '2', #
    },
  }
}

```

## 6.2 CACHE\_PURGED

File has been purged on disk cache.

Issue: is it actually the same as Migration Dispatcher reply MDR\_PURGED?

CACHE\_PURGED ticket is similar to CACHE\_WRITTEN event, enstore dictionary entry “enstore” is not required:

```
ticket_purged = {
  'cache': {
    # -- File Cache specific fields
    'arch': {
      # archive (tape backend)
      'id': 'cdf', # archive name
      'type': 'enstore' # archive type
    }
    'ns' : {
      # user namespace (frontend)
      'id': 'cdf', # namespace identification
      'type': 'pnfs', # user namespace description
      'mnt': '/pnfs/fnal.gov' # mount point in global namespace
    }
    'en' : {
      # enstore file cache
      'fsfn' : "node:/mount/path/filename" # fully specified file
      # name constructed from items below
      'node': 'cache01', # node name / address
      'mount': '/mnt/cache', # mount point of enstore cache
      'path': '000E/0000/0000/0000/095D', # cached file subpath
      'name': '000E00000000000000095D5220', # file name in cache
      'id': '0x12345' # file handle in cache
    }
  },
  'file' : {
    # -- data file specific fields
    # the original file name :
    'name': '/pnfs/fs/usr/Migration/cms/WAX/11/file.root',
      # file ID in user namespace. String.
    'id': '000E00000000000000095D5220'
    'size': 663748608L, # file size in bytes
  }
}
```

### 6.3 CACHE\_STAGED

File has been staged to enstore file cache on disk.

Issue: is it actually the same as Migration Dispatcher reply MDR\_STAGED?

The ticket is similar to CACHE\_WRITTEN event, enstore dictionary entries d['enstore']['vc'] and d['enstore']['location\_cookie'] are not required:

```
ticket_staged = {
  'cache': {
    # -- File Cache specific fields
    'arch': {
      # archive (tape backend)
      'id': 'cdf', # archive name
      'type': 'enstore' # archive type
    }
    'ns' : {
      # user namespace (frontend)
      'id': 'cdf', # namespace identification
      'type': 'pnfs', # user namespace description
      'mnt': '/pnfs/fnal.gov' # mount point in global namespace
    }
    'en' : {
      # enstore file cache
      'fsfn' : "node:/mount/path/filename" # fully specified file
      # name constructed from items below
      'node': 'cache01', # node name / address
      'mount': '/mnt/cache', # mount point of enstore cache
      'path': '000E/0000/0000/0000/095D', # cached file subpath
      'name': '000E00000000000000095D5220', # file name in cache
      'id': '0x12345' # file handle in cache
    }
  },
  'file' : {
    # -- data file specific fields
    # the original file name :
    'name': '/pnfs/fs/usr/Migration/cms/WAX/11/file.root',
    # file ID in user namespace. String.
    'id': '000E00000000000000095D5220'
    'size': 663748608L, # file size in bytes
  },
  'enstore' : {
    # -- Enstore backend specific fields
    'bfid': 'CDMS11578572850000', # (RD only)
  }
}
```

## 7 Enstore High Level Messaging API

We build enstore messaging API on qpid python High Level API. It became fully available in qpid v0.8 [Qpid Pr].

### 7.1 Modules

enstore/src package cache/messaging has modules as follows:

client.py	– base class EnqClient for Enstore qpid clients
enq_message.py	– base class for all enstore qpid messages
pe_client.py	– messages and replies processed by Policy Engine and base class for Policy Engine client
md_client.py	– messages and replies processed by Migration Dispatcher and base class for Migration Dispatcher client
mw_client.py	– messages and replies processed by Migration Worker and base class for Migration Worker client

cache/servers/ - enstore cache servers implementation

cache/stub/ - pilot implementation for server stubs

cache/test/ - test components

### 7.2 AMQP Broker

AMQP (qpid) broker is identified in configuration by host name where it runs and port where it listens.

### 7.3 Exchanges and Queues

Enstore components use direct exchange `enstore.fcache`. Each server component has input queue defined with name matching components name. More precisely, the routing key of the message is the destination component name and by default message routing key is bonded to the queue with the same name. The queue to process replies has “\_reply” appended to name of the queue.

The following queues are defined:

lmd	- Library Manager Dispatcher input queue
pe	- policy engine commands and events queue
pe_reply	- pe accepts replies on commands sent to MD
md	- Migration Dispatcher command queue
md_reply	- Migration Dispatcher waits replies here on commands it sent to Migration Workers
mw	- TBD. Migrator worker queue.

### 7.3 Enstore Qpid Client

Enstore qpid client is a base class encapsulating qpid connection, session(s), sender(s) and receiver(s). In constructor we create one sender and one receiver, more can be added. Sender and receiver each have address specified: the sender has destination address where messages will be delivered to, and receiver has address of the node where from messages are fetched. “Vanilla” enstore qpid client has to have provided *broker address* (host,port), *target address* used by default sender and specifying where to messages will be sent and the address “*myaddr*” used by default receiver to fetch commands (or in another instance, to fetch replies).

E.g. **on low level** PE component connecting to qpid broker on local host, port=5672 can create client with receiver to fetch commands from qpid queue named “pe” and sending commands to the output queue “md”:

```
import cache.messaging.client as cmc
qc = cmc.EnQpidClient(("localhost",5672),myaddr="pe", target="md")
qc.start()
while True:
    msg = qc.rcv.fetch()
    # ... process message, create file list l ...
    cmd = MDCArchive( file_list = l )
    qc.send(cmd)
```

**On “higher” level** each cache server component has corresponding client class *having* one or more EnQpidClient instances, e.g. MD class to implement Migration Dispatcher functionality will instantiate two EnQpidClient : one client to read events and send commands to MD, the other to read MD replies.

The component class (MD class here) is further customized in the server level component class (here MDs) class by injecting knowledge of enstore configuration. MDs communicates with enstore configuration server to get broker configuration, names of input/output queues and creates MD class with proper queues specification.

Here is example of Migration Dispatcher server MDs instantiating MD class:

```
# get configuration dictionary from Enstore configuration server
self.conf = self.csc.get(MY_NAME)
self.amqp_broker_conf = self.csc.get("amqp_broker")

# get broker address and message queue names from configuration
brk = self.amqp_broker_conf['host_port']
queue_in = self.conf['queue_in'] # this is 'md' (migrator disp. command queue)
queue_out = self.conf['queue_out'] # this is 'mw' (migrator worker queue)

# instantiate file cache component:
self.md_srv = md.MD(amqp_broker=brk, myaddr=queue_in, target=queue_out)
```

On application layer server enstore server instantiate top level component such as LMDs and it communicate with configuration server and instantiate 'component level' client with proper configuration.

### 7.3.1 EnqClient methods

**\_\_init\_\_(self, broker\_host\_port, myaddr=None, target=None)** – constructor.

**broker\_host\_port:** (host,port) of the qpid broker

**myaddr:** qpid address (queue name) where from default receiver fetches messages

**target:** qpid address (queue name) where to default senders sends messages

**configure()** – hook for subclass to provide extra configuration

**start()** – create connection to qpid broker, session, default receiver and sender.

**stop()** – close session and connection.

**receiver(self, addr, name)** – create additional receiver named “name” to fetch messages from remote node “addr”

addr: string, source address (queue name)

name: string, receiver name

**sender(self, addr, name)** – create additional sender named “name” to send messages to remote node “addr”

addr: string, destination address (queue name)

name: string, sender name

**send(self, msg, \*args, \*\*kwargs)** – send message msg through default sender and to associated address.

msg: qpid message. Message to send.

args, kwargs – extra arguments for qpid Sender

**fetch(self, \*args, \*\*kwargs)** – return message fetched from default receiver rev (connected to address set in constructor)

args, kwargs – extra arguments for qpid Receiver

returns: qpid message

## 7.4 Enstore Qpid Messages

EnqMessage is a base class for all Enstore qpid Messages. It is defined in module cache/messaging/enq\_message.py. EnqMessage inherits from qpid Message class, thus all enstore messages are qpid amqp messages. EnqMessage provides necessary qpid message customization by setting values message fields like id, correlation\_id and by setting message properties, where we set enstore message type, message version.

The following message classes derived from EnqMessage:

*class PEEvent(EnqMessage) – base class for all Policy Engine events*

*class MDCommand(EnqMessage) – base class for all MD commands*

*class MDReply(EnqMessage) – base class for all replies processed by MD*

*class MWCommand(EnqMessage) – base class for all MW commands*

*class MWReply(EnqMessage) – base class for all replies processed by MW*

Classes for concrete command or reply derived from proper class above, e.g. Migration Dispatcher ARCHIVE command is derived from generic MDCCommand class:

```
class MDCArchive(MDCCommand)
```

Under the hood, valid message types are defined in class MSG\_TYPES in module *cache.messaging.messages*. For instance, *MDCArchive* sets message type to *cache.messaging.messages.MDC\_ARCHIVE*, which has value of string “MDC\_ARCHIVE.” Strong typing enables early code checking in IDE and code integrity, message type representation as a string simplifies debugging. Class MSG\_TYPES also contains names for common operations (ARCHIVE, DELETE, PACK, PURGE, STAGE, UNPACK), states (ARCHIVED, DELETED, PACKED, PURGED, MISSED, STAGED, WRITTEN, CREATED, UNPACKED) and status (STATUS) defined as strings.

Quick example, create MD archive file list command and reply to the command:

```
>>> import cache.messaging.md_client as mdc
>>> f1=[1001,0x1,"file_A", "lib_A"]
>>> f2=[1002,0x2,"file_B", "lib_B"]
>>> l=[f1,f2]
>>> m=mdc.MDCArchive(l)
>>> r=mdc.MDRArchived(m,l)
>>> m
Message(correlation_id=UUID('315457aa-e4b8-4a97-83c6-6d3f9c514588'),
properties={'version': (0, 1), 'type': 'MDC_ARCHIVE'}, content=[[1001, 1, 'file_A',
'lib_A'], [1002, 2, 'file_B', 'lib_B']])
>>> r
Message(correlation_id=UUID('315457aa-e4b8-4a97-83c6-6d3f9c514588'),
properties={'version': (0, 1), 'type': 'MDR_ARCHIVED'}, content=[[1001, 1,
'file_A', 'lib_A'], [1002, 2, 'file_B', 'lib_B']])
```

### 7.4.1. Policy Engine Events.

Policy Engine events are defined in module *cache/messaging/pe\_client.py*

*PEEvent(EnqMessage)* – base class for all Policy Engine events

Events defined as classes derived from class *PEEvent*:

*EvtFileDeleted()* File deleted in Namespace

*EvtCacheMissed()* Client attempted to read file from cache and file not found in cache.

*EvtCachePurged()* File replica removed from cache (Migrator)

*EvtCacheWritten()* File replica written to cache by client (enstore Disk Mover)

*EvtCacheStaged()* File staged from tape to cache (Migrator)

## 7.4 Migration Dispatcher Commands and Replies.

Migration Dispatcher messages are defined in module *cache/messaging/md\_client.py*

Migration Dispatcher command messages inherit from generic MD command - base class *MDCCommand*.

Migration Dispatcher reply messages inherit from generic MD reply – base class **MDReply**. All Migration Dispatcher commands and replies inherit from Enstore Qpid Message class **EnqMessage**.

```
class MDCommand(EnqMessage): # base class for MD commands
```

```
class MDReply(EnqMessage):# base class for replies processed by MD
```

Class named to have two first letters matching component name, third letter is 'C' for Command and or 'R' for Reply, the rest is action or status name:

```
MDCArchive( file_list = l )
```

```
MDCPurge( file_list = l )
```

```
MDCStage( file_list = l )
```

```
MDRArchived( file_list = l )
```

```
MDRPurged( file_list = l )
```

```
MDRStaged( file_list = l )
```

## 7.5 Migration Worker Commands and Replies.

Migration Worker messages are defined in module *cache/messaging/mw\_client.py*

Migration Worker command messages inherit from generic MW command - base class **MWCommand**.

Migration Worker reply messages inherit from generic MW reply – base class **MWReply**. All Migration Worker commands and replies inherit from Enstore Qpid Message class **EnqMessage**.

```
class MWCommand(EnqMessage): # base class for MW commands
```

```
class MWReply(EnqMessage):# base class for replies processed by MW
```

Class named to have two first letters matching component name, third letter is 'C' for Command or or 'R' for Reply, the rest is action or status name:

```
MWCArchive( file_list = l )
```

```
MWCPurge( file_list = l )
```

```
MWCStage( file_list = l )
```

```
MWCStatus( request_id = id )
```

```
MWRArchived( file_list = l )
```

```
MWRPurged( file_list = l )
```

```
MWRStaged( file_list = l )
```

```
MWRStatus( content )
```

## References

[Qpid Pr] Jonathan Robie, Chuck Rolke, Alison Young, "Red Hat Enterprise MRG 1.3. Programming in Apache Qpid", Red Hat, Inc, 2011

[Qpid UG] Lana Brindley, Alison Young, "Red Hat Enterprise MRG 1.3. Messaging User Guide", Red Hat, Inc, 2011