

Messaging HLD for Enstore Small Files Aggregation Project

Alex Kulyavtsev

6/04/2010 v0.8

1 Introduction

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

Disk Mover

- 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

- central component to dispatch execution of work items (migrate file, restore file, ...) to Migrators (workers) to and track execution of requests.

Migrator

- on 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 standartize 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 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, list and array (`array` will be replaced by `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. In its turn 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, percistence, security, etc). AMPQ 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 intent to put Enstore specific message features into `application_headers` map and do not use message body. We may use *Message Body* field for bulk messages and/or when fast access to message payload is not required be messaging system.

2.1 AMQP Python quick example

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

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

#... get amqp session here ...

# Create some messages and put them on the broker.
dp = session.delivery_properties(routing_key="routing_key")
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_Dict
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 1

```
mp = session.message_properties()
mp.application_headers = {}
m = {}
m["msg_type"] = "MD_COMMAND"
m["msg_ver"] = (1, 0)
m["command"] = "MD_STORE_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 direct exchange. There are multiple Migrators, probably with different properties. A group of Migrator 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. 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* (client process), for example `migration_dispatcher`, `policy_engine`, “some” `fc_mover` or specific mover `fc_mover.mvr1234` (name includes “.” to separate fields).

5 Component Interaction Through AMQP Messaging

5.1 Data Delivery Service (Disk Mover) and Migrator communication with Policy Engine

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 (DM)
- `CACHE_MISS` // client attempts to read file from cache and file not found in cache. Triggers restore from tape & unpack (DM)
- `CACHE_RELEASED` // file copy removed from cache (MG)
- `CACHE_RESTORED` // file restored from tape to cache (MG)

`msg_type` : `NS_EVENT` // namespace event

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

5.1.3 Detailed Parameters Description

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

event :

- `CACHE_WRITTEN`
 - meaning: file written to cache. File can be created in namespace at the beginning of transfer, or it can be writing of the file already existing in name space (such as dcache file).
 - when: `close()` on write
 - action: aggregate write requests and prepare list of file to be written. Send *command* `STORE_FILES` to MD when needed.
- `CACHE_MISS`
 - when: `open()` on read with cache miss - file not found in cache
 - action: send *command* `RESTORE_FILES` to MD to read files from tape and unpack files.

- `CACHE_RESTORED` (`FILE_MIGRATED` ?)
 - meaning : file restored from tape to cache
 - action: release read pending transfers from cache to user
- `CACHE_RELEASED` (`CACHE_PURGED` ?)
 - meaning: file copy removed from cache
 - action: delete pending requests to store file if any. File can be released 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

5.2.1 Parameters

`msg_type` : `MD_COMMAND`

command :

- `MD_STORE_FILES` // Package and Write to tape
- `MD_RESTORE_FILES` // Read from tape and Unpack
- `MD_RELEASE_FILES` // release cache entry

`msg_type` : `MD_REPLY`

reply :

- `MD_FILES_STORED`
- `MD_FILES_RESTORED`
- `MD_FILES_RELEASED`

5.2.2 Description

Policy Engine sends *command* to Migration Dispatcher. Migration Dispatcher gets message and accepts the message (replies with acknowledge) and then work is executed asynchronously. After completion of the work Migration Dispatcher sends 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_STORE_FILES (MIGRATE_FILES ?)` // Package and Write package file to tape
- `MG_RESTORE_FILES (STAGE_FILES?)` // Read from tape and Unpack if needed
- `MG_RELEASE_FILES (PURGE_FILES?)` // Release 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 Migration Dispatcher 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 end of transfer and error status.

5.3.3 Description. Events and replies generated 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 `MG_REPORT_PROGRESS`

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

event :

- FILE_MIGRATED (FILE_STORED ?)
- FILE_STAGED (FILE_RESTORED ?)
- FILE_RELEASED (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 most information compared to other messages. These event lead to file archiving or restore. 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, 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 = {
  'efc': {
    'arc': {
      'id': 'cdf',
      'type': 'enstore'
    }
    'ns' : {
      'id': 'cdf',
      'type': 'pnfs',
      'mnt': '/pnfs/fnal.gov'
    }
    'efc' : {
      'node': 'cache01',
      'mount': '/mnt/cache/',
      'path': '000E/0000/0000/0000/095D',
      'name': '000E0000000000000000095D5220',
      'id': '0x12345'
    }
  },

  'file' : {
    'name': '/pnfs/fs/usr/Migration/cms/WAX/11/file.root',
    'id': '000E0000000000000000095D5220',
    '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_RELEASED

File has been released on disk cache.

The ticket is same as CACHE_WRITTEN event, “enstore” entry is not required.

6.3 CACHE_RESTORED

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

References

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