|  |  |  |
| --- | --- | --- |
| Cisco-logo-picture | Document Number | EDCS-1448321 |
| Based on Template | EDCS-189230 Rev 24  EDCS-228142 Rev 18 |
| Created By | E-NXOS Model Driven Programmability Team |

E-NXOS Model Driven Programmability  
Software Functional and Design Specification

Adopting DME model-driven framework to programmability across NXOS.

Reviewers

|  |  |
| --- | --- |
| Department | Name/Title |
| Development Engineers | Jag Tangirala, Jeffrey Liang, Saravanan Kannan, Donny Chan, Sharmila Palani, Minjie Qiu, Peter Kwok, Nitin Mathur, Junekang Yang, King Sun |
| Dev Mgr | Guhan Raman, Parag Deshpande (padeshpa) |
| Dev-Test Mgr | Bhavana Anklesaria |
|  |  |

Modification History

|  |  |  |  |
| --- | --- | --- | --- |
| Revision | Date | Originator | Comments |
| 1 | Oct 6, 2014 | E-NXOS Model Driven Programmability Team | Pre-draft |
| 2 | Oct 11, 2014 | Sharmila Palani | Added Section: 5.4, 5.4.1  Mapping of Bgp Config CLIs to DNs/MOs |
| 3  4. | Oct 21, 2014  Oct 29, 2014 | Sharmila Palani  Sharmila Palani | Added BGP commands/CLIs for objectifying bgp component in Section 3.1  Added Design flow for BGP component in Section 4. |
| 5 | Nov 19, 2014 | Nitin Mathur | Added Section 4.4 for Transaction Module |
| 6. | Apr 16, 2015 | Sharmila Palani | Updated BGP design to latest for DTHO |

Table of Contents

1 Problem Definition 5

2 Software Architecture 5

2.1 CLI 5

2.1.1 Configuration 5

2.1.2 Operational 11

2.1.3 CLI Metadata 12

2.1.4 CLI Compiler 12

2.1.5 VSH 12

2.2 Programmatic Agents 12

2.2.1 Configuration 13

2.2.2 Operational 14

2.3 Advanced Configuration 14

2.3.1 Configuration Replace 14

2.4 AAA 17

2.4.1 Authentication 17

2.4.2 Authorization (RBAC) 17

2.5 NE/DME 17

2.5.1 Configuration CLI 17

2.5.2 Configuration Replace 18

2.5.3 NE 18

2.6 DME Access Lib C++ 18

2.7 HA 18

2.8 ISSU 18

2.9 Applications 19

2.10 Startup Configuration 19

3 Software Requirements 19

3.1 BGP command sets 19

4 Design and System Flow 20

4.1 Design Flow for BGP 20

4.2 Advantages of using CLIS process for BGP 21

4.3 Code flow 22

4.4 BGP Operational data flow 29

4.5 Design Flow for Transaction Module 29

4.5.1 Overview 29

4.5.2 Architecture 30

4.5.3 Special Case 38

4.5.4 Limitation 38

5 Functional Structure 38

5.1 CLI Metadata 38

5.1.1 DME Handler directive 39

5.1.2 New token type 39

5.2 CLI Compiler 44

5.2.1 Load the DME handler directive into parse tree 44

5.2.2 Parsing of the new data type 44

5.3 VSH 45

5.3.1 cli\_dme\_handler class 45

5.3.2 Integrate cli\_dme\_handler into vsh infra 45

5.4 Bgp application capabilities 45

5.4.1 Mapping of Config CLIs to DNs/MOs 46

5.4.2 Mapping of show CLIs to DNs/MOs for rendering 58

6 System Flow 58

7 Data Structures 58

8 Description of Algorithms 58

8.1 Advanced Configuration 58

8.2 Configuration Replace 58

9 Interface Design 61

10 Memory and Performance Impact 62

11 Packaging Considerations 62

12 End User Interface/User Experience 62

12.1 REST API 63

12.1.1 REST API for Configuration Replace 64

13 Configuration and Restrictions 64

14 Software Restrictions and Considerations 64

15 Firmware Restrictions and Considerations 65

16 Hardware Restrictions and Considerations 65

17 External Restrictions and Configuration 65

18 Source Code 65

19 Testing Considerations 65

20 Development Unit Testing 66

21 Initiative, Legal, & Regulatory 66

22 Requirements Traceability Considerations 67

23 References 67

24 Glossary 67

25 Appendix 68

25.1 BGP Object Model 68

26 Attachments 70

26.1 Review Action Items 70

# Problem Definition

Adopting DME model-driven framework to programmability across NXOS.

# Software Architecture

E-NXOS Model Driven Programmability Architecture is captured in EDCS-1457089

## CLI

The command-line interface (CLI) is a method by which users can access the device. It provides a set of commands that one uses to monitor and configure the device.

The CLI is a straightforward command interface. CLI commands are typed on a single line, and the commands are executed when pressed Enter.

The NXOS CLI implementation is VSH, which communicates directly with the applications and that the applications are responsible for their configurations. But with the introduction of DME, this is changing and that is described below.

### Configuration

For DME-based applications, configuring via CLI goes over DME. The main considerations of this are:

* DME as the single source of truth.
* Multiple feature interactions are handled in a DME way.
* Northbound interfaces (CLI, REST, NETCONF, etc.) are handled in a DME way.

**Transaction Model (Existing)**

Clients of DME send request messages called *stimuli*. Each stimulus is processed by DME in the context of a transaction and a response is generated back to the client. The response indicates the transaction success or failure. Note that the transaction only operates on the model, i.e. only access the object-store. A failed transaction leaves the object-store unchanged.

Applications consumption of the configuration occurs outside of the transaction and only occurs after the transaction has been committed. Failures of applications in applying the configuration are not reflected in the response. User is informed of these by means of faults and associated notification mechanisms (e.g. Syslog).

Asynchronous event notification mechanism allows clients to be informed of changes posted to the object-store. The notifications are produced at the end of each transaction and there is one notification per object that changes.

|  |
| --- |
| NE  DME  Async Deployment (Not in Transaction)  Async Object Change Notification  Async Object Change Notification  REST  NGINX  Reply  e  Doer  Transactor  tr  API  Request  API  Reply  REST Client  External Request Stimulus  Changer  tr  External Request Stimulus  App  Object  Store |

**CLI Transaction Model**

CLI is a human-machine interface. Requirements of DME, to provide CLI backward compatibility, are:

* A CLI command failure is, as it never happened.
  + DME does undo (e.g. rolls back changes to object-store) if application returns an error.
  + DME does not produce asynchronous object change notifications and any other external notifications (e.g. syslog).
  + DME suppresses the faults resulted from processing this transaction.
* A response includes success or failure of the application. If failure, this needs to be replied to the CLI client of the correct session and synchronously.
  + DME supports an external request of *synchronous* stimulus. DME processing of this stimulus includes the application response and that it blocks the caller awaiting a response.
* A means of returning application specific error message (to be printed on VTY as same as today).
* A CLI command is transacted as a whole. That is, a CLI command, in the form of objects/stimuli, is sent to DME as a whole and that the application receives this CLI command as a whole.

*NOTE: DME will support both transaction models.*

**CLI Configuration Flow**

|  |
| --- |
| 1 Config  VTY  3 Act on config  Legacy App  4 Set  4 Sync response  2 Sync request  7 Act on config  7 Act on config  6 Read  8 Drop MTS message after consumption  DME  3 Validate against model  NE  10 Sync response  2 Sync stimulus  VSH  9 All drop notification  Object  Store  8 Drop MTS message after consumption  App  5 MTS primary chunk update with all drop notification  App  Map config cmd  to DN/MO  no  DME?  DME Access  Lib (C++)  yes |

**CLI Configuration Flow for Legacy (and non-DME based) Applications**

1. A CLI configuration command enters into VSH. VSH processes and validates that the command is syntactically valid.
2. After the command is validated, command data and context are retrieved and sent to the application synchronously to be applied.
3. Application validates that the command is semantically valid and then applies the configuration.
4. Application responses to VSH with success or failure. If a failure, application also returns an application specific error message to VSH to the external client in the response.

*NOTE: This is an existing flow and remains unchanged, for VSH directly to applications.*

**CLI Configuration Flow for DME based Applications**

1. A CLI configuration command enters into VSH:
   * VSH processes and validates that the command is syntactically valid.
   * VSH transforms the CLI configuration command to the corresponding DNs/MOs according to the CLI metadata specified in the .cmd definition files. For complex scenarios, special logics might be required for transforming. *See subsequent sections below for details.*
   * VSH creates a stimulus of the DNs/MOs (representing the CLI configuration command).

**Example:**

|  |
| --- |
| User enters  switch(config)# router bgp **100**  Where value 100 is the asn (Autonomous System Number).  VSH maps this command to  DN: sys/bgp/inst  Property name: asn  Property value: 100  Class name: bgpInst  Class name is composed of the package name and the MO name (refer to MO XML models). In code, class name is used as input to instantiate such an object.  Once object is instantiated sets property asn of object to value 100. Then creates a stimulus of this DN/MO to send to the NE/DME. |

1. VSH sends the stimulus as synchronous to NE/DME, via the DME access lib as a means:
   * To indicate that the transaction when NE/DME processes this stimulus includes the applications. *In the normal (asynchronous) case, applications deployment occurs outside the scope of this transaction*.
   * To block VSH awaiting a response.
2. NE/DME validates the object change within the stimulus against the MO XML models.
3. After the object change is validated, NE/DME transacts the object to object-store.

*NOTE: Since there could be application failure, do not produce asynchronous object change notifications at this time.*

|  |
| --- |
| DESIGN:  NE/DME invokes response required variant object-store API to transact the object to object-store. The response required variant of the object-store add/mod/delChunks() APIs have following additional properties to the normal ones:   * Creates a transient transaction handle (which holds the source information) and generates a transaction ID to correspond to the transaction. * MTS notification payload carries this transaction ID. * MTS message sends with MTS\_MTSBUF\_HEADER\_OPTION\_ALL\_DROP option. The significance of this option is that a local MTS event called MTS\_LOCAL\_EVENT\_ALL\_DROP will be generated with the original MTS payload once all the consumers have dropped the MTS message. |

1. Notifies applications of object change.

|  |
| --- |
| DESIGN:  Triggers MTS primary chunk update notification *with ALL DROP option* to applications. |

1. Applications read the configuration chunk for the object change.
2. Applications determine and apply the object change. If error, application undoes application changes as if though never happened.
3. Applications reply to NE/DME of done with status.

If status of failure, NE/DME

* + Undoes what were done (e.g. rolls back object change made to object-store, to replica, to persistent storage).
  + Suppresses faults raised by application, if any.

*NOTE: In the case of non-CLI, e.g. REST, path no rolling back of object-store and no suppressing of faults.*

If status of success, NE/DME

* Produces asynchronous object change notifications.

|  |
| --- |
| DESIGN:  Application replies to NE/DME of this by means of drop MTS message:   * If an application is successful in applying the object change, it silently drops the MTS message. * If an application fails to successfully process a particular property, it updates the appropriate bit in the configFailedStateBmp, returns an application specific error message, if applicable, and then drops the message.   In configFailedStateBmp, there is one bit per configuration property. This is only for MOs where attribute called config-control in the chunk definitions. Every secondary chunk where this attribute is present gets its own cfgFailedStateBmp. |

1. NE/DME constructs response. If an application error, NE/DME does undo.

|  |
| --- |
| DESIGN:  Once all the applications drop the message, system generates the MTS\_LOCAL\_EVENT\_ALL\_DROP event. NE/DME in receiving this event looks at the payload and retrieves the transaction ID and then finds the corresponding transaction. It constructs a success response if the configFailedStateBmp is zero. Otherwise, a failure response with the configFailedStateBmp and an application specific error message, if any.  The ALL\_DROP message will have the original payload that will be used to correlate what transaction this response is for. |

1. NE/DME responses to VSH. VSH processes the response. In case of error, VSH provides the application specific error message up the call chain, as today.

Typically, in the CLI path a configuration command is to a single process. All second order processes that need to react to the configuration, if any, have a private contract with the first order process, which owns that piece of configuration.

*NOTE: If a CLI configuration command maps to multiple DNs/MOs and these are to different processes, how should this be handled?* ***TBD****.*

### Operational

<Body>

*Note: Phase approach –* ***Explain rationale***

Show command goes directly to the backend of each process.

Each backend retrieves its operational data from LPSS either *directly* or from *object store* (depending on whether it is non-MO-ed config or MO-ed config).

Issues:

How to provide/guarantee consistency of information (“same” information available) on CLI agent and programmatic agents?

|  |
| --- |
| VTY  1 Show  4,5 Response  e  5 Response  e  1 Get  3 Read  REST  REST Client  VSH  Legacy App  App  3 Read  Object  Store  NGINX  4 Response  2 Get  DME  NE  4 Show response  2 Show request  2 Show request  3 Show response |

#### Operational Eventual

<Body>

‘show’ (includes show run) goes via DME.

Application operational data is then retrieved via DME (by accessing the data store).

Issues:

Large scale – with DME

|  |
| --- |
| 1 Show  4,5 Response  e  VTY  4 Response  2 Get  2 Get  Push operational data  App  5 Response  e  1 Get  3 Read  REST  REST Client  VSH  Object  Store  NGINX  4 Response  DME  NE  2 Show request  Legacy App  3 Show response |

### CLI Metadata

### CLI Compiler

### VSH

## Programmatic Agents

Programmatic agents support only over DME and follow the DME *eventual* deployment paradigm.

The programmatic agent is the REST interface, from iNX, initially. Then possibly next to be supported are RESTCONF agent and NETCONF agent. RESTCONF is currently an IETF draft described in <http://tools.ietf.org/html/draft-ietf-netconf-restconf-01>. NETCONF is described in RFC-6241.

Though how these agents would be supported is out of scope of this document, infrastructure enhancements would be needed to support, for examples:

* Candidate datastore
* Configuration lock
* Confirmed commit
* Additional operations
* YANG defined RPCs
* YANG defined notifications

### Configuration

For programmatic agents, content is DME based applications configuration only.

**Programmatic Agent Configuration Flow (REST Interface)**

1. REST client sends an XML payload over HTTPS.
2. NGINX server sends the payload in the form of stimulus to NE/DME via a REST lib.
3. NE/DME validates the object change within the stimulus against the MO XML models.
4. After the object change is validated, NE/DME transacts the object to object-store. The transaction only operates on the model, i.e. only access the object-store.
5. NE/DME generates a response, success or failure, to the client.
6. If transaction is committed successfully, notifies applications of object change.
7. Applications read the configuration chunk for the object change.
8. Applications determine and apply the object change. If error, applications raise faults and reflect failures in the model as operational state.

*NOTE: This is an existing flow and remains unchanged.*

|  |
| --- |
| REST  1 config (https)  6 response  6 MTS primary chunk update  8 Act on config  8 Act on config  App  App  3 Validate against model  Object  Store  6 response  2 config  5 config ack/nack  5 config ack/nack  5 config ack/nack  2 config  2 config  6 response  1 config (ssh)  1 config (https)  7 Read  4 Set  NE  DME  if restconf  NETCONF  Agent  RESTCONF  Agent  NETCONF Client  RESTCONF Client  REST Client  NGINX |

### Operational

<Body>

## Advanced Configuration

Advanced configuration support only on DME paths for programmatic agents.

### Configuration Replace

The configuration replace feature provides the capability to replace the current running configuration with the incoming configuration.

The granularity of configuration replace is an object subtree (specified in the URI if RESTful).

Use cases:

* If the root of the object subtree specified is root of MIT, then replaces the entire system configuration. *Where* ***system means DME based configuration only****, not the whole system configuration. That is until all applications configuration is DME based.*
* If the root of the object subtree specified is root of a feature (e.g. BGP, OSPF, ISIS), then replaces any configuration at the feature level.
* If the root of the object subtree specified is root of an instance (e.g. BGP peer, OSPF neighbor), then replaces any configuration at the specific instance.

|  |
| --- |
| This request emerges from a strong requirement that the configuration system running off the box is the master and must be able to put the system into known good states. Performing a diff operation to determine how to merge the master view into the device view is not deemed reliable. |

When this operation is performed, the current running configuration is compared with the specified replacement configuration and a set of diffs is generated. Only the diffs are applied, avoiding potential service disruption from deleting and reapplying configuration that already exists in the current running configuration.

|  |
| --- |
| Current running configuration (MIT)  Apply  diff  Replace diff  Calculate differential  Replacement configuration  Apply all necessary additions, deletions, and modifications.  Objects could be added, deleted, or modified. Properties of an object could be added, deleted, or modified. |

Requirements of DME:

* Support an external request stimulus that contains the entire incoming configuration to DME.
* Calculate differential between what is currently configured and what is incoming.
* Apply the configuration changes.
* Process the incoming configuration as a transaction.
* Support the level of granularity for configuration replace described above.
* Validate that the user is authorized to perform this operation. This includes being authorized for the objects being deleted as a result.

**Programmatic Agent Configuration Replace Flow (REST Interface)**

1. REST client sends an XML payload over HTTPS.
2. NGINX server sends the payload (entire replace request) in the form of objects to NE/DME via a REST lib.
3. NE/DME adds the input objects to a scratch pad (a temporary buffer) and organizes the objects hierarchically similar to MIT for ease of traversing and comparing for calculating differential.
4. NE/DME validates the input objects against the MO XML models.
5. After the objects are validated, NE/DME calculates differential of input objects and MIT/object-store to create an object replace diff. Validates that the user is authorized to delete the objects, if any, in the replace diff.
6. NE/DME transacts the object replace diff to object-store. The transaction only operates on the model, i.e. only access the object-store.
7. NE/DME generates a response, success or failure, to the client.
8. If transaction is committed successfully, notifies applications of object changes.
9. Applications read the configuration chunks for the object changes.
10. Applications determine and apply the object changes. If error, applications raise faults and reflect failures in the model as operational state.

|  |
| --- |
| 10 Act on config  3 Add  NE  5 Create object replace diff  4 Validate against model  9 Read  App  6 Set  Scratch pad  Replace diff  Undo object log  Object  Store  REST Client  1 ‘replace’ request (https)  1 ‘replace’ request (ssh)  8 response  REST  8 MTS primary chunk update  10 Act on config  App  8 response  2 config  7 config ack/nack  7 config ack/nack  7 config ack/nack  2 config  2 config  8 response  DME  if restconf  NETCONF  Agent  RESTCONF  Agent  NETCONF Client  RESTCONF Client  NGINX  1 ‘replace’ request (https) |

## AAA

### Authentication

To incorporate [ENXOS Authentication: EDCS-1453219](http://wwwin-eng.cisco.com/cgi-bin/edcs/edcs_info?EDCS-1453219)into this document

### Authorization (RBAC)

To incorporate [ENXOS RBAC: EDCS-1452803](http://wwwin-eng.cisco.com/cgi-bin/edcs/edcs_info?EDCS-1452803)into this document

## NE/DME

### Configuration CLI

Please refer to section *Configuration* [2.1.1] under CLI above.

### Configuration Replace

Please refer to section *Configuration* *Replace* [2.3] above.

### NE

The Network Element (NE) is an instance of DME with distinct code that is the configuration broker and configuration owner and is for *standalone* mode.

The Policy Element (PE) is specific for *fabric* mode and APIC solution. Where there are logical models and concrete models that PE understands and does fabric specific business logic.

In standalone mode, these logical models and fabric specific logic are not applicable.

The concrete models would be kept common, with fabric specific fields, if any, be hidden from external users in standalone mode. And the models would be enhanced as appropriate for features parity.

In code base, both PE and NE coexist under dme/svc/policyelem and dme/svc/networkelem, respectively. When building an image for fabric mode, only include PE; when building for standalone mode, only include NE.

In the generated code, “Policyelem” is part of names of some definitions and functions. And other code could be referring to these. Therefore, making this name change is not desirable. That is implementation as NE but still called “Policyelem”.

In the code base, there are places hardcoded with the service ID of PE. TBD: identify and evaluate impact of these.

Alternatively being looked at is top-down approach of starting with current PE and removing /disabling stuffs not applicable.

## DME Access Lib C++

<Body>

## HA

<Body>

## ISSU

To incorporate [ENXOS Upgrade/Downgrades and Model Change Issues: EDCS-1452805](http://wwwin-eng.cisco.com/cgi-bin/edcs/edcs_info?EDCS-1452805)into this document

## Applications

<Body>

Process restart

## Startup Configuration

<Body>

# Software Requirements

## BGP command sets

BGP commands must be supported through model-driven (DME) for BGP component:

1. Supported Config Commands:

Please refer to [MDP Models to CLI Definition: EDCS-1459693](http://wwwin-eng.cisco.com/cgi-bin/edcs/edcs_info?3868301)for complete set of commands.

2. Operational data:

Please refer to Mapping of bgp cli command and snmp to MO properties: **EDCS: 1485650**

# Design and System Flow

The following depicts the end-to-end flow of BGP process:

**DME Access**

**Lib (C++)**

**DME?**

**Map config cmd**

**to DN/MO**

**CLI/VTY**

**Object**

**Store**

NE

**DME**

**REST Client**

**NGINX**

**CLIServer**

**(CLIS)**

**BGP**

**1. Config Request**

**no**

**yes**

**2. Sync stimulus**

**14. Sync response**

**15. Config Response**

**2. Sync request**

**(Legacy flow)**

**3. Invoke BGP CLI handler to act on Config**

**4. Sync response (Legacy flow)**

**1. Config Async Request**

**4. set**

**3. PE + BI logic to Validate against MODEL(s)**

**5. DME MTS notification (add/update/delete)**

**6. Read data from objstore**

**7. MTS CLI handler request**

**10. CLI handler response**

**8. Act on Config**

**12. Drop MTS msg after consumption**

**13. Gets transaction response**

**5. Done**

**show**

**CLIs**

**9. Oper update**

**VSH**

Nvdb pss

**11. nvgen**

**Diagram: Design Flow**

## Design Flow for BGP

The following flow explains the internal details from CLIServer to BGP process on processing the request and response.

**CLIServer Process:**

Upon enabling of “feature bgp” command, as BGP is licensed based:

1. Initializes the Object store and Opens BGP/PE DB in read/write mode.
2. Registers for all the BGP’s MTS OPCODES to receive notifications of MO Additions/Updates/Deletion.
3. Waits on CLIS’s MTS queue to receive BGP related objstore MTS messages, which are transactional based.
4. Transactional Engine collects all the MOs for that transactional id.
5. For each of the MO notification, the corresponding handler gets the MTS payload from the message.
6. Reads the configuration chunk data associated with the MO object change.
7. Associate each of the MO (or more) notification data to a CLI command.
8. And sends a MTS message to invoke the Bgp application backend handler.
9. Wait for the processing response/status.

**BGP process:**

1. Waits on MTS queue to receive notifications from CLIServer.
2. Upon notification, gets the MTS payload from the message.
3. Executes the logic in the existing CLI handler code.
4. If it fails to successfully process, sends the error code and error string to CLIS via MTS message.
5. On success, updates the MO with operational data, if any. Please refer to section for operational data update.
6. Drops the MTS message.

**CLIServer process**

Upon return from bgp process,

1. If the application returns error code, it updates the appropriate bit in the configFailedStateBmp in the secondary chunk. Also sets application specific error message.
2. Updates its nvdb with bgp config for nvgenning
3. Drops the MTS message.

The flow for REST from #1 to #5 in diagram is asynchronous. But however, BGP component src code behaves the same for all agents (CLI as well as REST).

For transaction details, please refer to Section 4.4

Finally, CLIServer process does:

19. Transaction engine replies to DME of transaction done status by doing ALL\_DROP.

## Advantages of using CLIS process for BGP

1. Less code churn to the backend bgp component

[TBD: except for updating the operational data and fault error in objstore]

1. Reuse of CLIS nvdb storage for nvgen commands:

* show running-config
* show startup-config
* show running-config all
* show running-config bgp all
* copy run start

1. Faster replay of commands from CLIS nvdb during system boot (ascii).
2. All the external dependency logic is already maintained by CLIS. For example, bgp is dependent on l3vm, route-map, interfaces, …

## Code flow

Step 1: **Add handler directive to the command syntax**

In bgp.cmds:

command "launcher\_router\_cmd"

syntax '[no] router bgp <as>'

keyword no "Negate a comment, a command or set a command's default"

keyword router "Enable a routing process"

keyword bgp "Border Gateway Protocol (BGP)"

integer <as> 'Autonomous system number'

1 65535

key <as>

mode "/exec/configure"

chmode "/exec/configure/router-bgp" <as>

nvgen\_index 1300

**handler dme 'sys/bgp/inst.asn=<as>’**

end

**Step 2: CLIS - Initialize the Object Store**

In /feature/vsh/clis/bgp/bgp\_objstore\_mts.c

syserr\_t

bgp\_objstore\_init (mts\_q\_handle\_t mts\_qh)

{

syserr\_t rv = SUCCESS;

rv = oipcRegisterMtsHandle(mts\_qh);

…

rv = objStoreInit();

…

/\*

\* Open the DME PE DB if not already done

\*/

rv = objStoreOpen(DME\_DB\_NAME,

IM\_PROC\_ID\_POLICYELEM,

&dme\_obj\_db);

…

/\*

\* Open BGP DB if not already done. In read mode.

\*/

rv = objStoreOpen(BGP\_DB\_NAME,

IM\_PROC\_ID\_BGP,

&bgp\_obj\_db);

…

/\* Register for objstore MTS opcodes \*/

rv = bgp\_mts\_opcodes\_register(mts\_qh);

…

}

**Step 3:** **Register for the MO change notification**

In /feature/vsh/clis/bgp/bgp\_objstore\_mts.c,

static

bgp\_mts\_opcodes\_t mts\_opcodes[] =

{

/\* Inst ADD/MODIFY/DELETE MO \*/

{

MTS\_OPC\_CREATE\_ImBgpInstPolicyelem,

MTS\_OPTION\_PERSISTENT | MTS\_OPTION\_REGISTER

},

{

MTS\_OPC\_MODIFY\_ImBgpInstPolicyelem,

MTS\_OPTION\_PERSISTENT | MTS\_OPTION\_REGISTER

},

{

MTS\_OPC\_DELETE\_ImBgpInstPolicyelem,

MTS\_OPTION\_PERSISTENT | MTS\_OPTION\_REGISTER

}

};

/\*

\* bgp\_mts\_opcodes\_register

\*

\* Initialize the MTS opcode options

\* Opcodes are present in /include/isan/mts/include/Mts\_dme6\_opc.h

\*/

syserr\_t

bgp\_mts\_opcodes\_register (mts\_q\_handle\_t mts\_qh)

{

for (i = 0; i < BGP\_MSG\_ARRAY\_SZ ; i++) {

bgp\_mts\_opcodes\_t\* msg = &mts\_opcodes[i];

if (mts\_options\_set(mts\_qh,

msg->opcode,

msg->options) != SUCCESS) {

…

}

}

return (ret);

}

**/\***

**\* bgp\_mts\_dispatch()**

**\***

**\* Function to dispatch the notification to the right handler.**

**\***

**\*/**

syserr\_t

bgp\_mts\_dispatch (mts\_msg\_ref\_t \*mts\_msg, mts\_opc\_t opc)

{

syserr\_t ret = SUCCESS;

ObjStoreNotifyPld \*msg = NULL;

msg = (ObjStoreNotifyPld \*)mts\_payload\_get(mts\_msg);

switch (opc) {

case MTS\_OPC\_CREATE\_ImBgpInstPolicyelem:

/\* Action for router bgp <asn> command \*/

ret = mts\_bgp\_inst\_notif(msg,

BGP\_OBJSTORE\_CFG\_ADD);

break;

case MTS\_OPC\_MODIFY\_ImBgpInstPolicyelem:

ret = mts\_bgp\_inst\_notif(msg,

BGP\_OBJSTORE\_CFG\_UPDATE);

break;

default:

…

}

**Step 4: Wait for the MO Change Notification**

In feature/vsh/clis/clis\_mts.cc

syserr\_t

clis\_mts\_msg(…)

{

…

case MTS\_OPC\_CREATE\_ImBgpInstPolicyelem:

case MTS\_OPC\_MODIFY\_ImBgpInstPolicyelem:

case MTS\_OPC\_DELETE\_ImBgpInstPolicyelem:

bgp\_mts\_dispatch(ref, opc);

break;

…

}

**Step 5: Handler for command ‘router bgp <asn>’**

In /feature/vsh/clis/bgpbgp\_objstore\_mts\_handlers.c

/\*

\* mts\_bgp\_inst\_notif

\*

\* Function/Handler for Inst MO notification

\*/

syserr\_t

mts\_bgp\_inst\_notif (ObjStoreNotifyPld \*msg,

bgp\_cfg\_opers\_t action)

{

ObjStoreNotify \*notify = NULL;

ImBgpInstPolicyelem chunk = { 0 };

PriKey \*priKey = NULL;

uint64\_t chg\_bmp = 0;

cli\_cmd\_t \*cli\_cmd = NULL;

**…**

/\* Fetch the chunk \*/

notify = msg->objStoreNotify;

size = sizeof(chunk);

bzero(&priKey, sizeof(priKey));

ret = bgp\_get\_prichunk\_from\_notify(notify, &priKey, &chunk, &size);

chg\_bmp = (action == BGP\_OBJSTORE\_CFG\_ADD)? 0 :

getPropChgBmpFromNotification(notify);

**/\* Compose the CLI based on the property in BGP Inst MO \*/**

cli\_cmd = bgp\_inst\_cmds(&chunk, action, chg\_bmp);

**/\* Execute the cli command \*/**

ret = clis\_objstore\_cmds(cli\_cmd);

…

}

/\*

\* bgp\_inst\_cmds

\*

\* Function to read the data from objstore and compose a CLI.

\*

\* Returns: cli\_cmd\_t structure containing

\* - Command (eg, router bgp 10)

\* - Fully Qualified command (eg, ^ex^conf^router bgp 1)

\* - Current mode (eg, /exec/configure/router-bgp)

\* - Command id (launcher\_router\_cmd)

\* Note: All the above information is defined in .cmd for that command syntax.

\*/

static cli\_cmd\_t\*

bgp\_inst\_cmds (ImBgpInstPolicyelem \*chunk,

bgp\_cfg\_opers\_t action,

uint64\_t chg\_bmp)

{

cli\_cmd\_t \*cmd = NULL;

uint32\_t asn = 0;

uint32\_t len = 0;

char command[COMMAND\_MAX\_SIZE] = { 0 };

char mode[32] = "/exec/configure";

char cmd\_id[32] = "launcher\_router\_cmd";

…

/\* Retrieve data relevant for the CLI \*/

asn = getImBgpInstPolicyelemasn(chunk);

…

switch (action) {

case BGP\_OBJSTORE\_CFG\_ADD:

snprintf(command, COMMAND\_MAX\_SIZE,

"router bgp %d", asn);

cli\_create\_fq\_command(cmd->fq\_command,

FQ\_COMMAND\_MAX\_SIZE,

"%s", (char \*)command);

strncpy(cmd->command, command, strlen(command));

break;

case BGP\_OBJSTORE\_CFG\_UPDATE:

if (isDirtyImBgpInstPolicyelemasn(&chg\_bmp)) {

snprintf(command, sizeof(command),

"router bgp %d", asn);

cli\_create\_fq\_command(cmd->fq\_command,

FQ\_COMMAND\_MAX\_SIZE,

"%s", (char \*)command);

strncpy(cmd->command, command, strlen(command));

}

break;

case BGP\_OBJSTORE\_CFG\_DELETE:

snprintf(command, sizeof(command),

"no router bgp %d", asn);

cli\_create\_fq\_command(cmd->fq\_command,

FQ\_COMMAND\_MAX\_SIZE,

"%s", (char \*)command);

strncpy(cmd->command, command, strlen(command));

break;

…

}

}

int

clis\_objstore\_cmds (cli\_cmd\_t \*cli\_cmd)

{

…

/\* Populate the msg structure \*/

msg.set(CLI\_FQ\_PARENT\_FIELD, fq\_parent);

msg.set(CLI\_FQ\_COMMAND\_FIELD, cli\_cmd->fq\_command);

msg.set(CLI\_COMMAND\_LINE\_FIELD, command\_line);

msg.set(CLI\_MODE\_FIELD, mode\_name);

username.format("admin");

msg.set(CLI\_SESSION\_USERNAME\_FIELD, username);

msg.set(CLI\_SESSION\_PRIVILEGE\_FIELD, CLI\_PRIVILEGE\_LEVEL\_MAX);

msg.set(CLI\_COMPONENT\_FIELD, cmd->m\_component\_name);

mode\_stack.set\_config\_mode();

mode\_stack.populate\_from\_fqcmd(cli\_cmd->fq\_command, ctx);

mode\_stack.add\_stack\_commands(clis\_g, msg);

msg.set\_cmd\_args(ctx.m\_args);

msg.m\_send\_args = ctx.m\_args;

msg.set(CLI\_ORIGINAL\_COMMAND\_FIELD, cmd\_field);

msg.set(CLI\_COMMAND\_FIELD, cmd\_field);

msg.set(CLI\_REASON\_FIELD, CLI\_CMD\_EXECUTE);

return (**clis\_proxy\_standard**(cmd\_field, &msg, mode, cmd));

}

**Step 6: Update the Fault error status in the MO Instance**

Wait for the mts response from bgp process send to CLIS.

* Upon error, update the secondary chunk for fault.
* Update the transactional mo for error string/reason.

syserr\_t

bgp\_objstore\_upd\_inst\_sec\_chunk\_errs (uint64\_t inst\_oper\_error)

{

syserr\_t rv = SUCCESS;

ImBgpInstBgp chunk\_data = { 0 };

uint32\_t chunk\_size = sizeof(chunk\_data);

if ((rv = lookupChunkbyPriKey(bgp\_obj\_db,…) {

…

}

setImBgpInstBgpoperErr(getImBgpInstBgpoperErr(&chunk\_data) | inst\_oper\_error,

&chunk\_data);

if (((rv = bgp\_add\_sec\_chunk(&obj\_prikey,

getImBgpInstBgpChunkId(),

&chunk\_data) != SUCCESS))) {

…

return (rv);

}

## BGP Operational data flow

For non-real time operational data to DME objstore, bgp uses Lazy MO Queue update library to update their operational data. This mechanism achieves asynchronous, non-blocking service.

Lazy Managed Object Update Queue has the following aspects/attributes:

1. Setup multiple queues upto MAX\_OBJ\_UPD\_QUEUE limit of 8.
2. Each queue can have varying priority.
3. Client must provide an unique queue id, to identify the queue.
4. Priority of each queue need not be unique.
5. The queues are sorted from higher priority to lowest.
6. Work units (ie, number of MOs to be processed/updated from the queue during one cycle) is per queue basis.
7. Facility of merging MOs with unique key can be achieved through Coalesce flag set to true.
8. Thread setup with the callback facility/handler provided by the client. The client can use pthread or procket’s pthread.
9. Thread initialization with scheduling policy of SCHED\_IDLE for low priority updates, or default of SCHED\_OTHER.
10. It uses Pthread which is preemptive multitasking, with a time sliced environment, provides locking mechanish, and can be awakened by conditional signal.

## Design Flow for Transaction Module

### Overview

In NXOS, configuring a feature/functionality from the VSH/CLI involves interaction with the backend application that must respond to the CLI request with either a success or failure code.This makes VSH/CLI request as blocking/waiting for the backend response, which is different from REST interface.

Transaction Module is an intermediate layer, which will abstract if the request is coming from VSH/CLI or REST and will inform the CLIS MTS Handler if to return an Error code as response back or raise a Fault respectively.

### Architecture

**Prerequisites and Guidelines for upcoming Flow diagrams**

* CLIS module has registered for Transaction Begin and Transaction End and all the other needed objectStore needed MTS op-codes.
* Transaction Module and CLIS MTS Handler is part of CLIS process.
* returnStruct Pseudo code

typedef struct returnStruct\_

{

// Response code returned by backend to VSH/CLI.

uint64\_t respStatusCode;

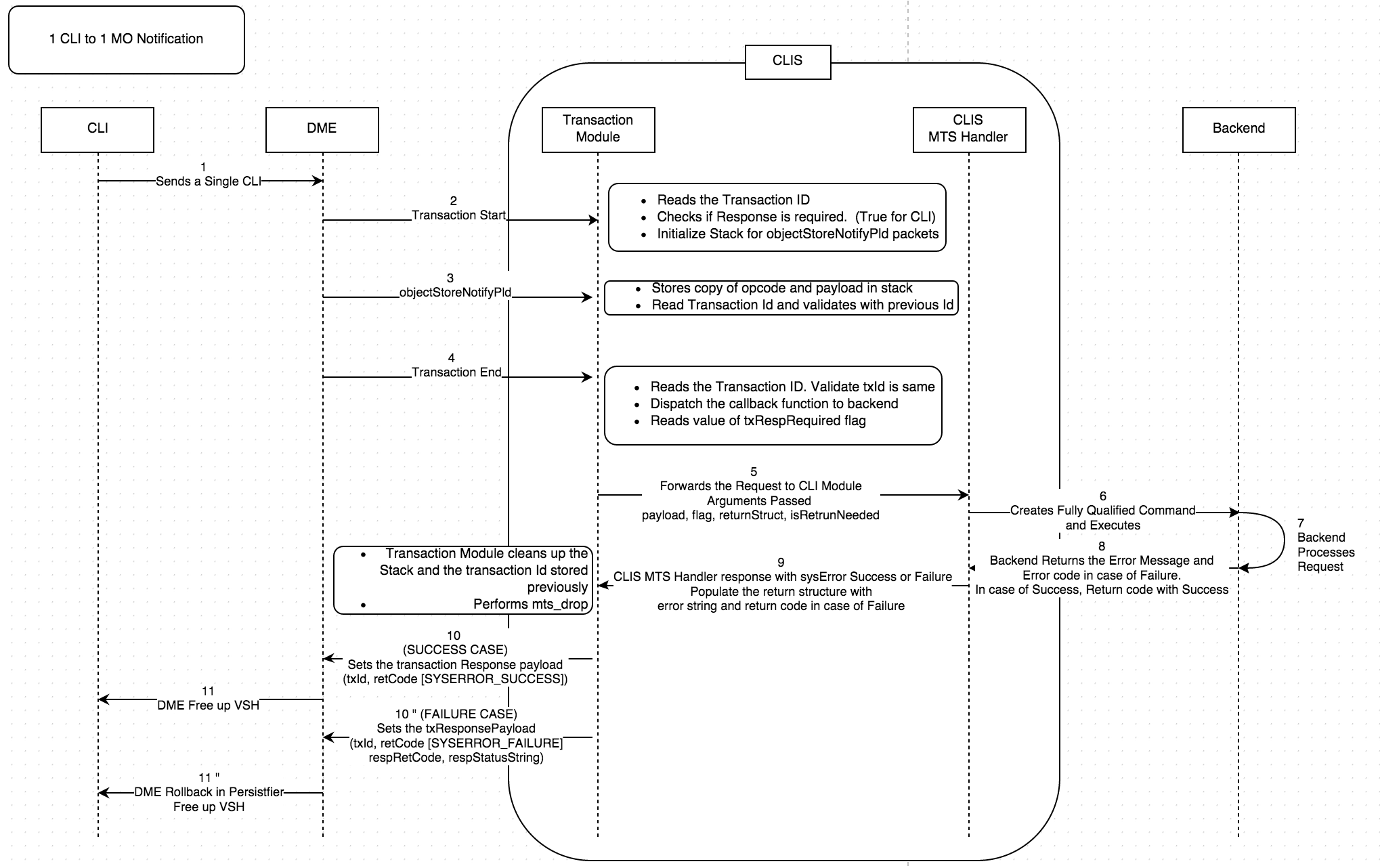
// Response string returned by backend to VSH/CLI

char respStatusString[MAX\_APP\_RESP\_ERROR\_STRING\_SIZE];

} returnStruct;

**FLOW DIAGRAM 1**

The following diagram provides a high level flow if a request is coming from VSH/CLI. Here the single CLI corresponds to a single MO modification.



**Detail Description of the Flow diagram**

1. The request is generated by VSH/CLI and is passed on to DME in a single transaction.
2. DME sends a Transaction Start op-code (MTS\_OPC\_DME\_TX\_BEGIN) to Transaction Module.
   1. The Transaction Model will read the Transaction id from the payload and stores it.
   2. Will check if the Response required flag is set to TRUE or FALSE. Note**:-** in case of request coming from VSH/CLI it will be TRUE.
   3. Initialize Queue to store the next coming request (objectStoreNotifyPld and opcode)
3. DME sends objectStoreNotifyPld MTS request to Transaction Module.
   1. Transaction model reads the Transaction Id, validates it with the previously received Transaction Id.
   2. Transaction model will store all consecutive incoming objectStoreNotifyPld request till the Transaction End opcode is received.
4. DME sends a Transaction End op-code (MTS\_OPC\_DME\_TX\_END) to Transaction Module.
   1. Transaction Model reads the Transaction Id, validates it with the previously received Transaction Id.
   2. Transaction Model will dispatch the request to CLIS MTS Handler to process the objectStore request.
   3. Transaction Module reads if txRespRequired flag is set or not.

**Note:-** In case of VSH/CLI this flag is always being set.

1. Transaction Model forwards the request to CLIS MTS Handler with 4 arguments, mts payload, mts opcode, returnStruct and isReturnStructNeeded.

**Note:-** In case if the request comes from VSH/CLI, the isReturnStructNeeded flag is set to TRUE.

1. CLIS MTS Handler will decode the payload, creates fully qualified command and executes it.
2. The Backend consumes the fully qualified command and tries to process the backend functionality.
3. Backend returns the response to CLIS MTS Handler as SUCCESS or FAILURE. In case of FAILURE, it passes an Error message and Error code.
4. CLIS MTS Handler checks if isReturnStruct is set to TRUE. (In case of VSH/CLI it is set to TRUE.)
   1. In case of FAILURE, CLIS MTS Handle populates the returnStruct.
   2. In case of SUCCESS, returnStruct is not populated.

CLIS MTS Handler passes control back to Transaction Module with a function return as SUCCESS or FAILURE.

* 1. In case if function returns as Success, Transaction Module will create the Transaction Response Payload. Transaction Module sets the transaction Id and retCode as SYSERROR\_SUCCESS.
  2. In case if function return as Failure, Transaction Module will create the Transaction Response Payload. Transaction Module sets the transaction Id, retCode as SYSERROR\_FAILURE. Transaction module also sets the respReturnCode and respStatusString as received from CLIS MTS Handler in previous step.

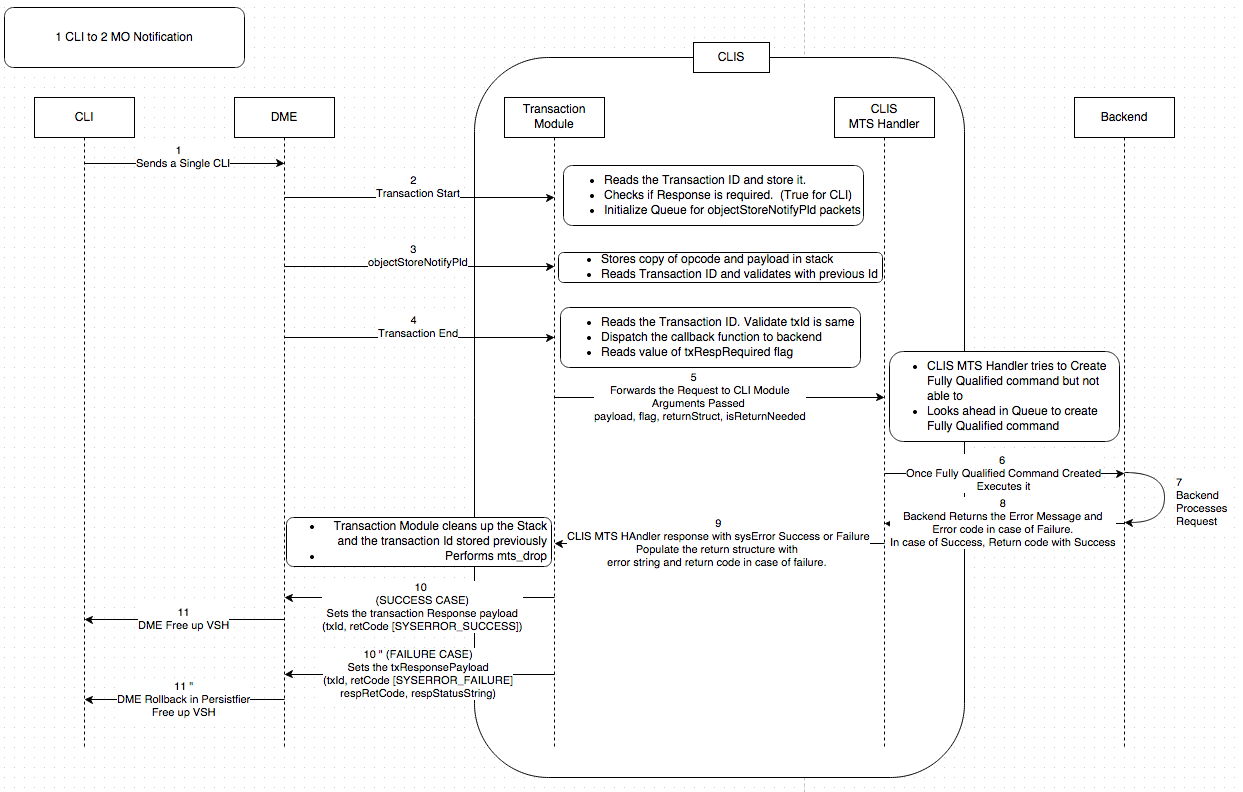
Transaction Module cleans up the stack and previously stored transaction Id. Transaction Module also preforms mts\_drop for the opcodes.

* 1. In case of retCode set to SYSERROR\_SUCCESS, DME frees up VSH/CLI.
  2. In case of retCode set to SYSERROR\_FAILURE, DME rollbacks Persistifier and frees up VSH/CLI.

**FLOW DIAGRAM 2**

The following diagram provides a high level flow if a request is coming from VSH/CLI. Here the single CLI corresponds to 2 MO modifications.

**Note:-** There is no current example for this case.



**Detail Description of the Flow diagram**

1. The request is generated by VSH/CLI and is passed on to DME in a single transaction.
2. DME sends a Transaction Start op-code (MTS\_OPC\_DME\_TX\_BEGIN) to Transaction Module.
   1. The Transaction Model will read the Transaction id from the payload and stores it.
   2. Will check if the Response required flag is set to TRUE or FALSE. Note**:-** in case of request coming from VSH/CLI it will be TRUE.
   3. Initialize Queue to store the next coming request (objectStoreNotifyPld and opcode)
3. DME sends objectStoreNotifyPld MTS request to Transaction Module.
   1. Transaction model reads the Transaction Id, validates it with the previously received Transaction Id.
   2. Transaction model will store all consecutive incoming objectStoreNotifyPld request till the Transaction End opcode is received.
4. DME sends a Transaction End op-code (MTS\_OPC\_DME\_TX\_END) to Transaction Module.
   1. Transaction Model reads the Transaction Id, validates it with the previously received Transaction Id.
   2. Transaction Model will dispatch the request to CLIS MTS Handler to process the objectStore request.
   3. Transaction Module reads if txRespRequired flag is set or not.

**Note:-** In case of VSH/CLI this flag is always being set.

1. Transaction Model forwards the request to CLIS MTS Handler with 4 arguments, mts payload, mts opcode, returnStruct and isReturnStructNeeded.

**Note:-** In case if the request comes from VSH/CLI, the isReturnStructNeeded flag is set to TRUE.

1. CLIS MTS Handler will decode the payload and tries to create fully qualified command. If it cannot create a fully qualified command CLI MTS Handler will look ahead in queue.
2. The Backend consumes the fully qualified command and tries to process the backend functionality.
3. Backend returns the response to CLIS MTS Handler as SUCCESS or FAILURE. In case of FAILURE, it passes an Error message and Error code.
4. CLIS MTS Handler checks if isReturnStruct is set to TRUE. (In case of VSH/CLI it is set to TRUE.)
   1. In case of FAILURE, CLIS MTS Handle populates the returnStruct.
   2. In case of SUCCESS, returnStruct is not populated.

CLIS MTS Handler passes control back to Transaction Module with a function return as SUCCESS or FAILURE.

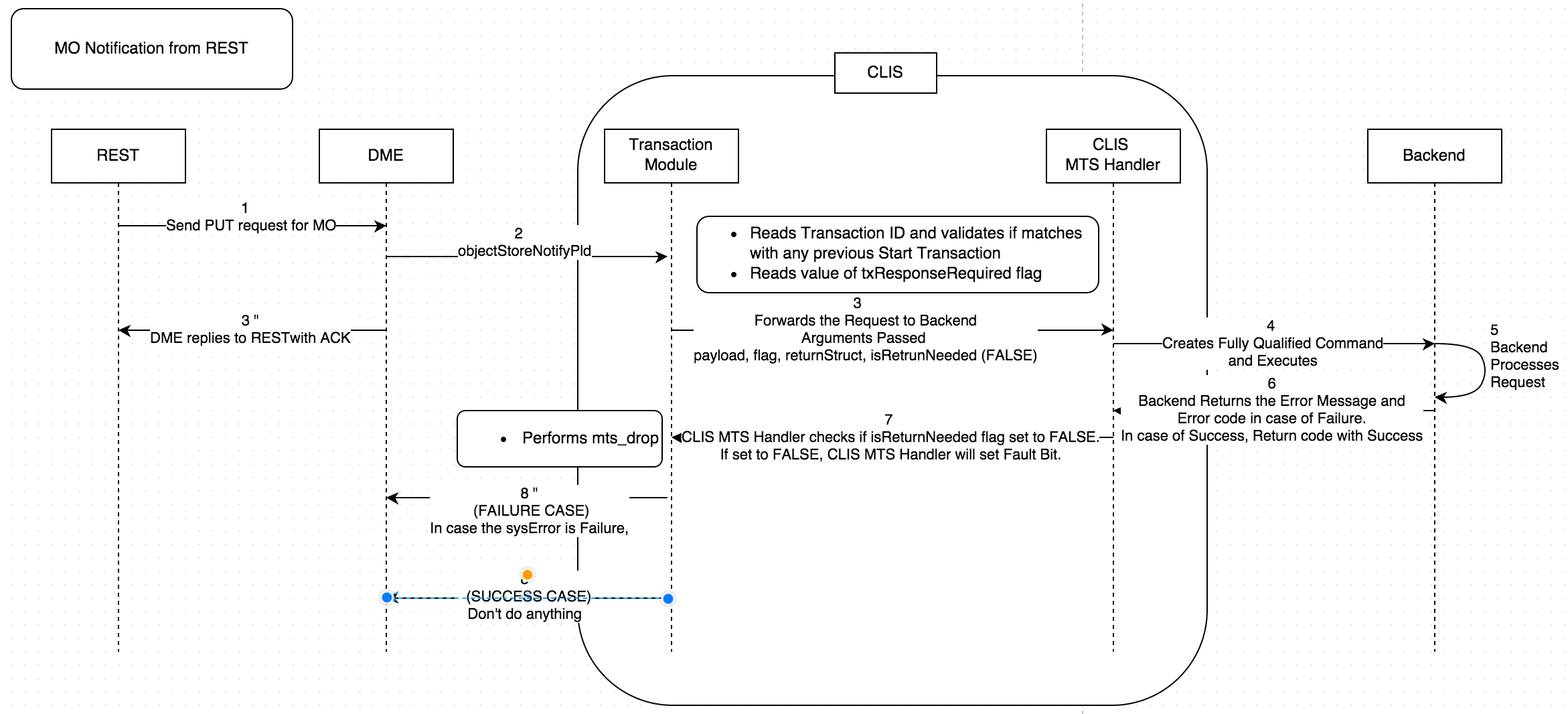
* 1. In case if function returns as Success, Transaction Module will create the Transaction Response Payload. Transaction Module sets the transaction Id and retCode as SYSERROR\_SUCCESS.
  2. In case if function return as Failure, Transaction Module will create the Transaction Response Payload. Transaction Module sets the transaction Id, retCode as SYSERROR\_FAILURE. Transaction module also sets the respReturnCode and respStatusString as received from CLIS MTS Handler in previous step.

Transaction Module cleans up the stack and previously stored transaction Id. Transaction Module also preforms mts\_drop for the opcodes.

* 1. In case of retCode set to SYSERROR\_SUCCESS, DME frees up VSH/CLI.
  2. In case of retCode set to SYSERROR\_FAILURE, DME rollbacks Persistifier and frees up VSH/CLI.

**FLOW DIAGRAM 3**

The following diagram provides a high level flow if a request is coming from REST.



**Detail Description of the Flow diagram**

1. The request is generated by REST and is passed on to DME.
2. DME sends objectStoreNotifyPld MTS request to Transaction Module.
   1. Transaction model reads the Transaction Id, validates it with the previously received Transaction Id, if any.
   2. Transaction model reads value of txResponseRequired flag.
3. Transaction Model forwards the request to CLIS MTS Handler with 4 arguments, mts payload, mts opcode, returnStruct and isReturnStructNeeded.

**Note:-** In case if the request comes from REST, the isReturnStructNeeded flag is set to FALSE.

1. CLIS MTS Handler will decode the payload, creates fully qualified command and executes it.
2. The Backend consumes the fully qualified command and tries to process the backend functionality.
3. Backend returns the response to CLIS MTS Handler as SUCCESS or FAILURE. In case of FAILURE, it passes an Error message and Error code.
4. CLIS MTS Handler checks if isReturnStructNeeded flag is set to FALSE. (In case of REST it is set to FALSE.)
   1. In case of FAILURE, CLIS MTS Handle will mark the Fault Bit.
   2. In case of SUCCESS, will not do anything.

CLIS MTS Handler passes control back to Transaction Module with a function return as SUCCESS or FAILURE.

1. Transaction Module also preforms mts\_drop for the opcode.

**FLOW DIAGRAM 4**

**Scenario:-**

switch# sh run | sec bgp

feature bgp

router bgp 100

vrf foo

address-family ipv4 unicast

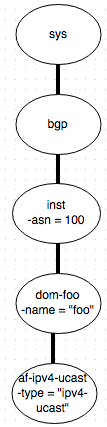
switch# config t

Enter configuration commands, one per line. End with CNTL/Z.

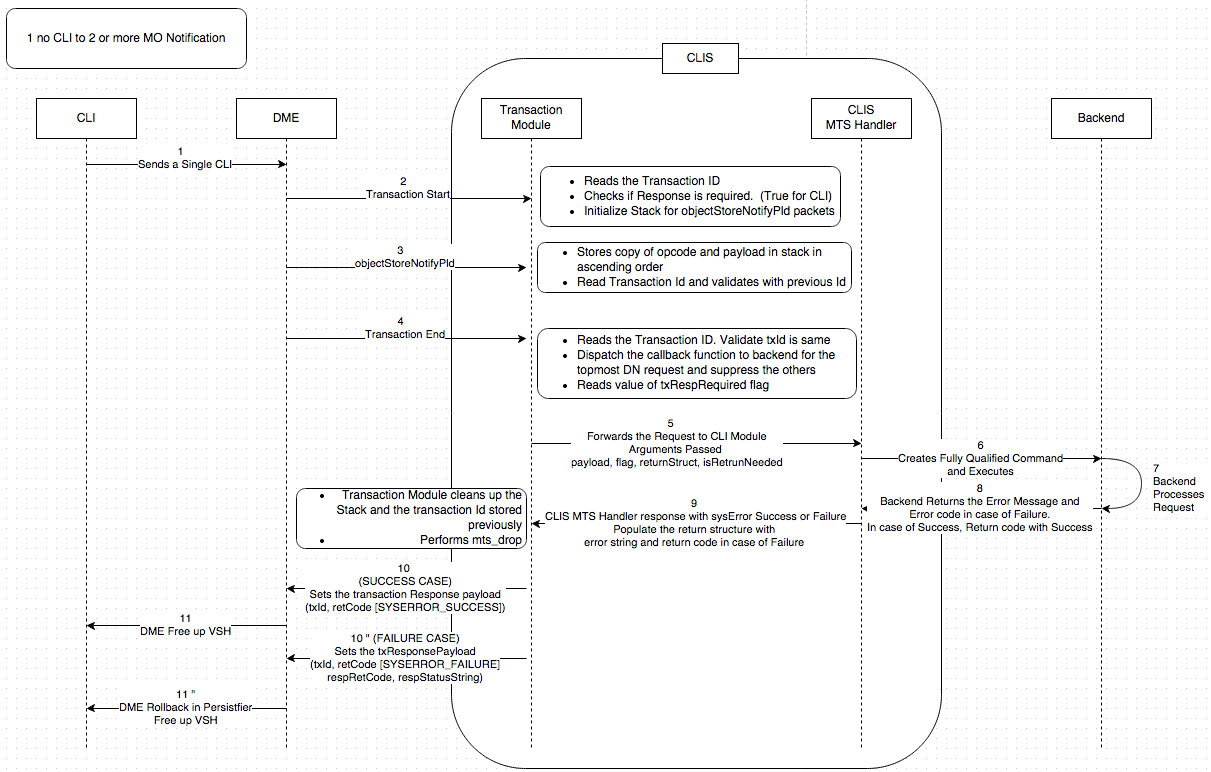
switch(config)# router bgp 100

switch(config-router)# **no vrf foo**

**MIT TREE:-**

****

The following diagram provides a high level flow for a no command.



**Detail Description of the Flow diagram**

1. The request is generated by VSH/CLI and is passed on to DME in a single transaction.
2. DME sends a Transaction Start op-code (MTS\_OPC\_DME\_TX\_BEGIN) to Transaction Module.
   1. The Transaction Model will read the Transaction id from the payload and stores it.
   2. Will check if the Response required flag is set to TRUE or FALSE. Note**:-** in case of request coming from VSH/CLI it will be TRUE.
   3. Initialize Queue to store the next coming request (objectStoreNotifyPld and opcode)
3. DME sends objectStoreNotifyPld MTS request to Transaction Module.
   1. Transaction model reads the Transaction Id, validates it with the previously received Transaction Id.
   2. Transaction model will store all consecutive incoming objectStoreNotifyPld request in ascending order till the Transaction End opcode is received.
4. DME sends a Transaction End op-code (MTS\_OPC\_DME\_TX\_END) to Transaction Module.
   1. Transaction Model reads the Transaction Id, validates it with the previously received Transaction Id.
   2. Transaction Model will dispatch the top most DN request to CLIS MTS Handler to process.
   3. Transaction Module reads if txRespRequired flag is set or not.

**Note:-** In case of VSH/CLI this flag is always being set.

1. Transaction Model forwards the request to CLIS MTS Handler with 4 arguments, mts payload, mts opcode, returnStruct and isReturnStructNeeded.

**Note:-** In case if the request comes from VSH/CLI, the isReturnStructNeeded flag is set to TRUE.

1. CLIS MTS Handler will decode the payload, creates fully qualified command and executes it.
2. The Backend consumes the fully qualified command and tries to process the backend functionality.
3. Backend returns the response to CLIS MTS Handler as SUCCESS or FAILURE. In case of FAILURE, it passes an Error message and Error code.
4. CLIS MTS Handler checks if isReturnStruct is set to TRUE. (In case of VSH/CLI it is set to TRUE.)
   1. In case of FAILURE, CLIS MTS Handle populates the returnStruct.
   2. In case of SUCCESS, returnStruct is not populated.

CLIS MTS Handler passes control back to Transaction Module with a function return as SUCCESS or FAILURE.

* 1. In case if function returns as Success, Transaction Module will create the Transaction Response Payload. Transaction Module sets the transaction Id and retCode as SYSERROR\_SUCCESS.
  2. In case if function return as Failure, Transaction Module will create the Transaction Response Payload. Transaction Module sets the transaction Id, retCode as SYSERROR\_FAILURE. Transaction module also sets the respReturnCode and respStatusString as received from CLIS MTS Handler in previous step.

Transaction Module cleans up the stack and previously stored transaction Id. Transaction Module also preforms mts\_drop for the opcodes.

* 1. In case of retCode set to SYSERROR\_SUCCESS, DME frees up VSH/CLI.
  2. In case of retCode set to SYSERROR\_FAILURE, DME rollbacks Persistifier and frees up VSH/CLI.

### Special Case

If the command coming in is a range command (e.g. - int ethernet1/1-20; ip address 1.1.1.1 255.255.255.0). VSH/CLI will split this command in multiple transactions and pass it on to DME as individual transaction. Thus for Transaction Module, CLIS MTS Handler and Backed it will be processed as individual transactions. This consideration is made as in current behavior this configuration is considered as “apply partial config if possible”. This way Transaction Module, CLIS MTS Handler and Backend will not have to remember the current state and rollback in case of Failure.

### Limitation

This document does not support if 2 MO creates 1 CLI and the request comes from REST. Currently the thought process is to store the request in a Queue if we cannot create a CLI. For any of the later request that comes from REST, we will look at this Queue to create the Fully Qualified command.

This approach will not be able to support Bulk commits if the requirement comes in the future meaning “all or nothing”. For that we will have to enhance the Backend to remember the current configuration and rollback to it if Transaction Module asks to. Since bulk commit is not part of requirement for camden, this is not thought about.

# Functional Structure

## CLI Metadata

NXOS CLI metadata are defined in files with extension “.cmd”. The grammar of the current .cmd files needs to be extended in order to support the interaction with the NE/DME.

### DME Handler directive

Each CLI definition contains a “handler” directive that defines the how this CLI should be handled. So far, NXOS command can have 3 types of handlers: function, SAP and procket. A new handler is needed for commands that need to be processed through DME. It is defined as handler “dme” followed by list of DNs/MOs that need to be added or updated:

handler dme <dn-value>+

The “<dn-value>” argument is list of DN/MO name value pair that is separated by space. Below is an example of how command “bgp <as>” is defined using the new handler directive:

command ‘some\_command’

syntax bgp <as>

keyword bgp "BGP protocol"

string <as>" “Autonomous-system number in asdot format”

"

mode CFG\_BGP\_SUBMODE

handler dme 'sys/bgp/inst.asn=<as>’

end

In the above example, the DN property “sys/bgp/inst.asn” is set to the <as> number upon command execution.

The mapping between CLI and DN properties is one to many. That is, one CLI may result in one or more DN or DN properties be updated.

### New token type

Each CLI token is represented by a unique token type, which includes keyword, string, integer, ipaddress and etc. Not all DME data type has a corresponding token type that can be used in the command definition. Therefore new token types are need to be created for those that are defined in DME but not in NXOS CLI.

The subsequent subsections gives list of base DME data types as well as list of token types supported in NXOS CLI. The new token types that need to be created are marked as red in the DME data type table.

#### Token types in NXOS CLI

typedef enum {

TOKEN\_TYPE\_UNSET = 0,

TOKEN\_TYPE\_KEYWORD,

TOKEN\_TYPE\_ENUM,

TOKEN\_TYPE\_BOOTVAR,/\* sanos specific \*/

TOKEN\_TYPE\_BOOTVAR\_KS\_SYS,/\* sanos specific \*/

TOKEN\_TYPE\_LC\_PARTNER,/\* sanos specific \*/

TOKEN\_TYPE\_LC\_PARTNER\_CLEAR,/\* sanos specific \*/

TOKEN\_TYPE\_LC\_PARTNER\_DEBUG,/\* sanos specific \*/

TOKEN\_TYPE\_LC\_PARTNER\_APPID,/\* sanos specific \*/

TOKEN\_TYPE\_LC\_DEV\_NAME,/\* sanos specific \*/

TOKEN\_TYPE\_ILC\_APPL\_NAME,/\* sanos specific \*/

TOKEN\_TYPE\_ILC\_APPL\_NAME\_EXT,/\* sanos specific \*/

TOKEN\_TYPE\_INTERFACE,

TOKEN\_TYPE\_INTERFACE\_RANGE,

TOKEN\_TYPE\_INTERFACE\_MRANGE,/\* sanos specific \*/

TOKEN\_TYPE\_ETHERNET,

TOKEN\_TYPE\_USERATHOST,

TOKEN\_TYPE\_INTEGER,

TOKEN\_TYPE\_INTEGER\_RANGE,/\* sanos specific \*/

TOKEN\_TYPE\_INTEGER\_MRANGE,/\* sanos specific \*/

TOKEN\_TYPE\_UINTEGER,

TOKEN\_TYPE\_LONGLONG,

TOKEN\_TYPE\_VSAN,/\* sanos specific \*/

TOKEN\_TYPE\_VSAN\_RANGE,/\* sanos specific \*/

TOKEN\_TYPE\_VSAN\_MRANGE,/\* sanos specific \*/

TOKEN\_TYPE\_FCID,/\* sanos specific \*/

TOKEN\_TYPE\_WWN,/\* sanos specific \*/

TOKEN\_TYPE\_LUN,/\* sanos specific \*/

TOKEN\_TYPE\_URI,/\* sanos specific \*/

TOKEN\_TYPE\_UNUSED, /\* this can be used later \*/

TOKEN\_TYPE\_IPADDR,

TOKEN\_TYPE\_IPPREFIX,

TOKEN\_TYPE\_IPV6,

TOKEN\_TYPE\_IPV6\_PREFIX,

TOKEN\_TYPE\_HEX,

TOKEN\_TYPE\_HEX\_RANGE,/\* sanos specific \*/

TOKEN\_TYPE\_HEX\_MRANGE,/\* sanos specific \*/

TOKEN\_TYPE\_SID,

TOKEN\_TYPE\_NID,

TOKEN\_TYPE\_LID,

TOKEN\_TYPE\_NET,

TOKEN\_TYPE\_COMMUNITY,

TOKEN\_TYPE\_PCL\_EXPRESSION,

TOKEN\_TYPE\_QOS\_EXPRESSION,

TOKEN\_TYPE\_BIT\_EXPRESSION,

TOKEN\_TYPE\_PASSWORD,

TOKEN\_TYPE\_REMAINDER,

TOKEN\_TYPE\_COMMAND,

TOKEN\_TYPE\_STRING,

TOKEN\_TYPE\_FILENAME,

TOKEN\_TYPE\_UINTEGER\_RANGE,

TOKEN\_TYPE\_ILC\_APPL\_NAME\_ALL,/\* sanos specific \*/

TOKEN\_TYPE\_CFS\_APPL\_NAME,/\* sanos specific \*/

TOKEN\_TYPE\_XBAR\_STR,/\* sanos specific \*/

TOKEN\_TYPE\_XBAR\_DRV,/\* sanos specific \*/

TOKEN\_TYPE\_LC\_PARTNER\_SYSTEM,/\* sanos specific \*/

TOKEN\_TYPE\_USERDEF,/\* user defined (dlopened function) \*/

TOKEN\_TYPE\_VLAN,

TOKEN\_TYPE\_VLAN\_MRANGE,

TOKEN\_TYPE\_VRF,

TOKEN\_TYPE\_BOOL\_ENABLED,

TOKEN\_TYPE\_BOOL\_INSERVICE,

TOKEN\_TYPE\_ASN, /\* BGP's autonomous system nb: <u16>.<u16> \*/

TOKEN\_TYPE\_CMD, /\* an encapsulated cli command (see alias command) \*/

TOKEN\_TYPE\_DATE, /\* a date (1970.12.31-24:60:60) \*/

TOKEN\_TYPE\_LDATE, /\* a long date (1970.12.31-24:60:60-999:999) \*/

TOKEN\_TYPE\_DURATION, /\* a duration (seconds or microseconds) \*/

TOKEN\_TYPE\_LDURATION, /\* a duration (seconds + microseconds) \*/

TOKEN\_TYPE\_BITMAP, /\* 1,5,10-20 \*/

TOKEN\_TYPE\_FLOAT, /\* 1.123 \*/

TOKEN\_TYPE\_TIME, /\* time (24:60:60) \*/

TOKEN\_TYPE\_NAMEORIP, /\*name or ip \*/

TOKEN\_TYPE\_IPMASKORLEN, /\*ipaddress and/or subnet len \*/

TOKEN\_TYPE\_BINARY, /\* array of uint8 (cli equivalent of void\*) \*/

TOKEN\_TYPE\_INTEGER\_LIST, /\* array of uint32 (like snmp OIDs) \*/

TOKEN\_TYPE\_IPV46NAME, /\* union of ipv4 ipv6 dns-name \*/

TOKEN\_TYPE\_BD, /\* bridge-domain \*/

TOKEN\_TYPE\_BD\_MRANGE, /\* bridge-domain multi-range (encoded as bitmap) \*/

TOKEN\_TYPE\_MAX

} cli\_token\_type\_g;

#### Base DME data type

Only the types that can be used in the information model need to be mapped.

|  |  |  |  |
| --- | --- | --- | --- |
| **Base Type Name in the Information Model** | **Base Type** | **C++ Type** | **Indicates if the type can be used in the Information Model** |
| address:BDF | BDF | uint32\_t |  |
| address:GUID | GUID | base::String |  |
| address:Ipv4 | Ipv4 | uint32\_t |  |
| address:Ipv6 | Ipv6 | base::String |  |
| **address:MAC** | **MAC** | **uint64\_t** |  |
| address:Subnet | SUBNET | uint32\_t |  |
| **address:UUID** | **UUID** | **base::UUID** |  |
| **address:UUIDPrefix** | **UUID\_PREFIX** | **base::UUIDPrefix** |  |
| address:UUIDSuffix | UUID\_SUFFIX | base::UUIDSuffix |  |
| **address:WWN** | **WWN** | **uint64\_t** |  |
| binary:Buffer | BINBUFFER | base::Buffer |  |
| collection:List | LIST | std::vector< $VALUE > |  |
| collection:Map | MAP | std::map< $KEY, $VALUE > |  |
| config:ClassIdSet | CLASSID\_SET | method::ClassIdSet |  |
| config:Config | CONFIG | method::Config |  |
| config:Context | CONTEXT | uint64\_t |  |
| config:DnSet | DN\_SET | method::DnSet |  |
| config:Filter | FILTER | method::Filter |  |
| config:IdSet | ID\_SET | method::IdSet |  |
| config:List | CONFIG\_LIST | method::ConfigList |  |
| config:Map | CONFIG\_MAP | method::ConfigMap |  |
| config:NameSet | NAME\_SET | method::NameSet |  |
| config:Set | CONFIG\_SET | method::ConfigSet |  |
| config:Value | VALUE | method::Value |  |
| config:ValueMap | VALUE\_MAP | method::ValueMap |  |
| config:ValuePair | VALUE\_PAIR | method::ValuePair |  |
| extpol:ConnType | ConnType | uint32\_t |  |
| method:MethodArgId | ARG\_ID | uint32\_t |  |
| method:MethodCollection | METHOD\_SET | method::MethodSet |  |
| method:MethodContainer | METHOD | method::Holder |  |
| method:MethodId | METHOD\_ID | uint32\_t |  |
| mo:MoClassId | CLASS\_ID | uint32\_t |  |
| mo:PropId | PROP\_ID | uint32\_t |  |
| mo:StatsClassId | STATS\_CLASS\_ID | uint32\_t |  |
| mo:StatsPropId | STATS\_PROP\_ID | uint32\_t |  |
| proc:ApplicationId | APP\_ID | uint32\_t |  |
| proc:PkgingId | PACKAGING\_ID | uint32\_t |  |
| proc:ServiceId | SVC\_ID | uint32\_t |  |
| **reference:Object** | **OBJECT\_REFERENCE** | **mo::Dn** |  |
| reference:RN | RN\_REFERENCE | mo::Rn |  |
| reference:Type | TYPE\_REFERENCE | base::String |  |
| **scalar:Bitmask16** | **BITMASK16** | **uint16\_t** |  |
| **scalar:Bitmask32** | **BITMASK32** | **uint32\_t** |  |
| **scalar:Bitmask64** | **BITMASK64** | **uint64\_t** |  |
| **scalar:Bitmask8** | **BITMASK8** | **uint8\_t** |  |
| **scalar:Bool** | **BOOL** | **bool** |  |
| **scalar:Char** | **CHAR** | **uint8\_t** |  |
| **scalar:Date** | **DATE** | **uint64\_t** |  |
| **scalar:Double** | **DOUBLE** | **int64\_t** |  |
| **scalar:Enum16** | **ENUM16** | **uint16\_t** |  |
| **scalar:Enum32** | **ENUM32** | **uint32\_t** |  |
| **scalar:Enum64** | **ENUM64** | **uint64\_t** |  |
| **scalar:Enum8** | **ENUM8** | **uint8\_t** |  |
| **scalar:Float** | **FLOAT** | **float** |  |
| **scalar:Sbyte** | **SBYTE** | **int8\_t** |  |
| **scalar:Seconds** | **SECONDS** | **uint64\_t** |  |
| **scalar:Sint16** | **SINT16** | **int16\_t** |  |
| **scalar:Sint32** | **SINT32** | **int32\_t** |  |
| **scalar:Sint64** | **SINT64** | **int64\_t** |  |
| **scalar:Time** | **TIME** | **uint64\_t** |  |
| **scalar:Ubyte** | **UBYTE** | **uint8\_t** |  |
| **scalar:Uint16** | **UINT16** | **uint16\_t** |  |
| **scalar:Uint32** | **UINT32** | **uint32\_t** |  |
| **scalar:Uint64** | **UINT64** | **uint64\_t** |  |
| **stream:Biovec** | **BIOVEC** | **base::Biovec** |  |
| **string:Basic** | **BASIC\_STRING** | **base::String** |  |
| **string:CharBuffer** | **STRBUFFER** | **base::Buffer** |  |
| **string:Password** | **PASSWORD** | **base::String** |  |
| UNKNOWN | UNKNOWN | int32\_t |  |

## CLI Compiler

### Load the DME handler directive into parse tree

The CLI compiler is enhanced to parse the new syntax “handler dme <dn-value>+”. Upon loading the parse tree, DN property and value pairs are saved into two lists defined in command\_token\_data\_c:

class command\_token\_data\_c : public token\_data\_c

{

…..

string\_array\_c m\_dn\_property\_list;

string\_array\_c m\_dn\_value\_list;

}

There is one to one relationship between the elements in the above two lists, which indicates which value corresponds to which DN property.

### Parsing of the new data type

In addition to the changes to process the dme handler directive, code needs to be added to support the token types identified in the previous section.

The implementation of each new token type is similar. The following are what needs to be added for each token type:

* Define the corresponding syntax for token parsing and implement the parsing of the new syntax in cli\_read\_command\_files.cc.
* Add a C++ class to store token data. This class is inherited from class token\_data\_c.
* Add a C++ class to handle the parsing of this token. This class is inherited from class token\_data\_handler\_c. Define and implement the match() method. It method will be invoked by vsh during command parsing. If the match() method returns TRUE, it means this token is accepted and vsh moves on to parse the next token.

The integer types ranges from int8\_t, uint8\_t to int64\_t and uint64\_t. According to the above design that each token type involves creating two C++ class, one for token data and the other for token parsing, the integer types will result in 16 C++ classes. What is being considered is to consolidate the processing of all integers into one token data class and one token parsing class instead of creating separated C++ classes for each one of them.

## VSH

In the existing vsh, CLIs are parsed and translated into TLVs, then sent to backend to be processed and stored. With the DME architecture, vsh does not interface with backend directly. It interfaces with DME access library to persist DN/MO.

The CLI configuration flow for both DME based applications and legacy applications are already described in section 2.1.1. This section focuses on the design detail of vsh enhancements.

### cli\_dme\_handler class

Define a new C++ class cli\_dme\_handler to include operations that interfaces with DME C++ library:

* Create stimulus of the DNs/MOs
* Sends the stimulus as synchronous to NE/DME via the DME access lib
* Process response from NE/DME
* Convert CLI TLV to data types supported by DME

[Details of the actual class definition will be added later.]

### Integrate cli\_dme\_handler into vsh infra

After the parsing of CLI syntax, vsh invokes the handler function in command\_dispatcher\_c::send\_command(). This is where cli\_dme\_handler is instantiated and the DNs/MOs are sent to DME.

## Bgp application capabilities

BGP is the routing protocol commonly used to exchange neighbor reachability information between autonomous systems. This application/feature is taken for demonstrating the capability to integrate with DME model and for end-to-end flow in this project.

The following list some of the bgp feature’s configuration commands (CLI) supported on the switch:

router bgp <asn>

shutdown

router-id <ip-address>

no graceful-restart

confederation identifier <id>

confederation peers <asn> <asn> <asn> <asn>

suppress-fib-pending

neighbor-down fib-accelerate

log-neighbor-changes

address-family <type>

maximum-paths <num>

neighbor <ip-address> remote-as <asn>

no shutdown

password <text>

update-source <interface>

address-family <type>

maximum-prefix <number>

vrf name

neighbor 1.2.3.4 remote-as 4

### Mapping of Config CLIs to DNs/MOs

These CLI configuration commands are mapped to the corresponding DNs in Managed Objects (MOs)

#### [no] router bgp <asn>

**Config**: router bgp 1

**Purpose**: To handle creation/deletion of a new bgp instance as specified by the

Policyelem

|  |  |  |  |
| --- | --- | --- | --- |
| CLI | MO | MTS | Handler |
| router bgp <asn> | MO: <mo name=”Inst”>  **DN**: sys/bgp/inst  **Property name**: asn  **Property value**: <num>  **Class name**: bgpInst  **File**: inst.xml |  |  |
| ADD MO operation |  | MTS\_OPC\_CREATE\_ImBgpInstPolicyelem | bgp\_process\_inst\_notif(  &mts\_msg, BGP\_OBJSTORE\_CFG\_ADD);   * bgp\_obj\_inst\_create() * bgp\_obj\_inst(); * bgp\_obj\_update\_inst(); * bgp\_objstore\_update\_inst()   Property name/value update:   * createTs * activateTs * waitDoneTs * memAlert |
| MODIFY operation |  | MTS\_OPC\_MODIFY\_ImBgpInstPolicyelem | bgp\_process\_inst\_notif(  &mts\_msg,  BGP\_OBJSTORE\_CFG\_UPDATE);  - bgp\_obj\_inst()   * bgp\_objstore\_update\_inst() |
| DELETE operation |  | MTS\_OPC\_DELETE\_ImBgpInstPolicyelem | bgp\_process\_inst\_delete\_notif(&mts\_msg);  - bgp\_obj\_inst\_delete()  -bgp\_objstore\_update\_inst()  - objStoreDeleteSubtree()  (Recursively delete children MOs from objstore) |

#### [no] router-id <ip-address>

**Config**:

router bgp 1

vrf vrf-name

router-id 100.112.255.1

**Purpose**: To handle creation/deletion of ipaddress associated to the bgp instance as specified by the policyelem.

|  |  |  |  |
| --- | --- | --- | --- |
| CLI | MO | MTS | Handler |
| router-id <ip-address> | **MO**: <mo name=”Dom”>  **DN**: sys/bgp/inst/dom-[name]  **Property name**: rtrId  **Property value**: <ip>  **Class name**: bgpDom  **File**: dom.xml |  |  |
| Add operation |  | MTS\_OPC\_CREATE\_ImBgpDomPolicyelem | bgp\_process\_dom\_notif  &mts\_msg,  BGP\_OBJSTORE\_CFG\_ADD);    -bgp\_obj\_dom\_create()  -bgp\_cfg\_vrf\_rtrId()  Get the vrf name:  vrf\_name = getImBgpDomPolicyelemname(elem); |
| Modify operation |  | MTS\_OPC\_MODIFY\_ImBgpDomPolicyelem | bgp\_process\_dom\_notif(&mts\_msg,  BGP\_OBJSTORE\_CFG\_UPDATE);  - getPropChgBmpFromNotification();  - bgp\_cfg\_vrf\_rtrId() |
| Delete operation |  | MTS\_OPC\_DELETE\_ImBgpDomPolicyelem | * bgp\_process\_dom\_delete\_notif(&mts\_msg);   - getImBgpDomNameFromDnBgpDom(&binDn, &vrf\_name);  - bgp\_cfg\_vrf\_rtrId\_no(); |

#### [no] graceful-restart

**Config**:

router bgp 1

vrf vrf-name

no graceful-restart

**Purpose**: To handle graceful restart associated to the bgp instance in domain.

|  |  |  |  |
| --- | --- | --- | --- |
| CLI | MO | MTS | Handler |
| graceful-restart | **MO**: <mo name=”Gr”>  **DN**: sys/bgp/inst/dom-[name]/gr  **Property name**: ctrl  **Property value**: <GrCtrl>  **Class name**: ImBgpGr  **File**: gr.xml |  |  |
| Add operation |  | MTS\_OPC\_CREATE\_ImBgpGrPolicyelem | bgp\_process\_gr\_notif(  &mts\_msg,  BGP\_OBJSTORE\_CFG\_ADD);  -bgp\_obj\_gr\_create()  - bgp\_cfg\_lookup\_vrf\_ctx(vrf\_name);  bgp\_cfg\_vrf\_gr();  Property update name/value:   * ctrl * restartIntvl * staleIntvl |
| Modify operation |  | MTS\_OPC\_MODIFY\_ImBgpGrPolicyelem | bgp\_process\_gr\_notif(  &mts\_msg,  BGP\_OBJSTORE\_CFG\_UPDATE);  isDirtyImBgpGrPolicyelemctrl(&propChgBmp); |
| Delete operation |  | MTS\_OPC\_DELETE\_ImBgpGrPolicyelem | bgp\_process\_gr\_delete\_notif(&mts\_msg);  - Update the ctrl property to “false”;  - Update vrf context’s graceful\_restart to false. |

#### [no] neighbor <ip-address> [remote-as <asn>]

**Config**:

router bgp 1

vrf vrf-name

neighbor 101.0.0.1 remote-as 8101

**Purpose**: To handle creation/update/deletion of BGP peers as specified for domain

|  |  |  |  |
| --- | --- | --- | --- |
| CLI | MO | MTS | Handler |
| neighbor <ip-address>  remote-as <asn> | **MO**: <mo name=”Peer”>  **DN**: sys/bgp/inst/dom-[name]/peer-[[addr]]  **Property name**: addr  **Property value**: <address:Ip>  **Property name**: asn  **Property value**: <rtcom:AS>  **Class name**: ImBgpPeer  **File**: peer.xml |  |  |
| Add operation |  | MTS\_OPC\_CREATE\_ImBgpPeerPolicyelem | bgp\_process\_peer\_notif(&mts\_msg,  BGP\_OBJSTORE\_CFG\_ADD);  getImBgpDomNameFromDnBgpPeer();  -bgp\_obj\_peer\_create()  -bgp\_cfg\_peer\_ipaddr(); |
| Modify operation |  | MTS\_OPC\_MODIFY\_ImBgpPeerPolicyelem | bgp\_process\_peer\_notif(&mts\_msg,  BGP\_OBJSTORE\_CFG\_UPDATE);  - bgp\_cfg\_peer\_ipaddr();  - isDirtyImBgpPeerPolicyelemasn();  - bgp\_objstore\_update\_cfgpeer();  -bgp\_change\_neighbor\_as(); |
| Delete operation |  | MTS\_OPC\_DELETE\_ImBgpPeerPolicyelem | bgp\_process\_peer\_delete\_notif( &mts\_msg);   * getImBgpDomNameFromDnBgpPeer(); * getImBgpPeerAddrFromDnBgpPeer();   -bgp\_obj\_peer\_delete(); |

#### : [no] shutdown

**Config**:

router bgp 1

vrf vrf-name

neighbor 101.0.0.1 remote-as 8101

no shutdown

**Purpose**: To handle administrative operational status of BGP peers as specified for neighbor/peer in that domain.

|  |  |  |  |
| --- | --- | --- | --- |
| CLI | MO | MTS | Handler |
| shutdown | **MO**: <mo name=”Peer”>  **DN**: sys/bgp/inst/dom-[name]/peer-[[addr]]  **Property name**: adminSt  **Property value**: < nw:AdminSt>  **Class name**: ImCPeer  **File**: peer.xml |  |  |
| Add operation |  | MTS\_OPC\_CREATE\_ImBgpPeerPolicyelem | bgp\_process\_peer\_notif(&mts\_msg,  BGP\_OBJSTORE\_CFG\_ADD); |
| Modify operation |  | MTS\_OPC\_MODIFY\_ImBgpPeerPolicyelem | bgp\_process\_peer\_notif (&mts\_msg, BGP\_OBJSTORE\_CFG\_UPDATE);  - getPropChgBmpFromNotification();  - - bgp\_cfg\_peer\_attributes(); |
| Delete operation |  | MTS\_OPC\_DELETE\_ImBgpPeerPolicyelem | bgp\_process\_peer\_delete\_notif (&mts\_msg); |

#### [no] update-source <interface>

**Config**:

router bgp 1

vrf vrf-name

neighbor 101.0.0.1 remote-as 8101

update-source Ethernet3/1

**Purpose**: To handle neighbor/peer source interface info in that domain

|  |  |  |  |
| --- | --- | --- | --- |
| CLI | MO | MTS | Handler |
| update-source ethernet 3/1 | **MO**: <mo name=”Peer”>  **DN:** sys/bgp/inst/dom-[name]/peer-[[addr]]  **Property name**: srcIf  **Property value**: <nw:IfId>  **Class name**: ImCPeer  **File**: peer.xml |  |  |
| Add operation |  | MTS\_OPC\_CREATE\_ImBgpPeerPolicyelem | bgp\_process\_peer\_notif(&mts\_msg,  BGP\_OBJSTORE\_CFG\_ADD); |
| Modify operation |  | MTS\_OPC\_MODIFY\_ImBgpPeerPolicyelem | bgp\_process\_peer\_notif (&mts\_msg, BGP\_OBJSTORE\_CFG\_UPDATE);  - getPropChgBmpFromNotification();  - - bgp\_cfg\_peer\_attributes(); |
| Delete operation |  | MTS\_OPC\_DELETE\_ImBgpPeerPolicyelem | bgp\_process\_peer\_delete\_notif (&mts\_msg); |

#### [no] password <text>

**Config**:

router bgp 1

vrf vrf-name

neighbor 101.0.0.1 remote-as 8101

password 3 857d737b69d69744

**Purpose**: To handle add/modify/delete of password setting for neighbor/peer source in that domain

|  |  |  |  |
| --- | --- | --- | --- |
| CLI | MO | MTS | Handler |
| password <LINE> | **MO**: <mo name=”Peer”>  **DN**: sys/bgp/inst/dom-[name]/peer-[[addr]]  **Property name**: password  **Property value:** < string:Password>  **Class name:** ImCPeer  **File**: peer.xml |  |  |
| Add operation |  | MTS\_OPC\_CREATE\_ImBgpPeerPolicyelem | bgp\_process\_peer\_notif(&mts\_msg,  BGP\_OBJSTORE\_CFG\_ADD); |
| Modify operation |  | MTS\_OPC\_MODIFY\_ImBgpPeerPolicyelem | bgp\_process\_peer\_notif (&mts\_msg, BGP\_OBJSTORE\_CFG\_UPDATE);  - getPropChgBmpFromNotification();  - - bgp\_cfg\_peer\_attributes(); |
| Delete operation |  | MTS\_OPC\_DELETE\_ImBgpPeerPolicyelem | bgp\_process\_peer\_delete\_notif (&mts\_msg); |

#### [no] address-family ipv4 unicast

**Config**:

router bgp 1

vrf vrf-name

address-family ipv4 unicast

**Purpose**: To handle creation/update/deletion of new BGP domain AF objects under **vrf submode.**

|  |  |  |  |
| --- | --- | --- | --- |
| CLI | MO | MTS | Handler |
| * address-family ipv4 unicast * address-family ipv4 multicast * address-family ipv6 unicast * address-family ipv6 multicast | MO: <mo name=”Af”>  **DN:** sys/bgp/inst/dom-[name]/af-[type]  **Property name**: type  **Property value:** <AfT>  **Class name**: ImCDomAf  **File**: af.xml |  |  |
| Add operation |  | MTS\_OPC\_CREATE\_ImBgpDomAfPolicyelem | bgp\_process\_domaf\_notif(&mts\_msg,  BGP\_OBJSTORE\_CFG\_ADD);  -getImBgpDomNameFromDnBgpDomAf();  getImBgpDomAfPolicyelemtype();  -bgp\_obj\_domaf\_create(); |
| Modify operation |  | MTS\_OPC\_MODIFY\_ImBgpDomAfPolicyelem | bgp\_process\_domaf\_notif(&mts\_msg, BGP\_OBJSTORE\_CFG\_UPDATE);  - getPropChgBmpFromNotification();  - bgp\_objstore\_update\_domaf (); |
| Delete operation |  | MTS\_OPC\_DELETE\_ImBgpDomAfPolicyelem | bgp\_process\_domaf\_delete\_notif (&mts\_msg);   * getImBgpDomNameFromDnBgpDomAf * getImBgpDomAfTypeFromDnBgpDomAf   - bgp\_obj\_domaf\_delete(); |

#### [no] address-family ipv4 unicast

**Config**:

router bgp 1

vrf vrf-name

neighbor 101.0.0.1 remote-as 8101

address-family ipv4 unicast

**Purpose**: To handle creation/update/deletion of BGP neighbor/peer AF objects in a domain under **neighbor submode**.

|  |  |  |  |
| --- | --- | --- | --- |
| CLI | MO | MTS | Handler |
| address-family <ip-address> | **MO:** <mo name=”PeerAf”>  **DN:** sys/bgp/inst/dom-[name]/peer-[[addr]]/af-[type]  **Property name:** type  **Property value:** <AfT>  **Class name:** ImCPeerAf  **File:** peer.xml |  |  |
| Add operation |  | MTS\_OPC\_CREATE\_ImBgpPeerAfPolicyelem | bgp\_process\_peeraf\_notif(&mts\_msg,  BGP\_OBJSTORE\_CFG\_ADD);  -getImBgpPeerAddrFromDnBgpPeerAf();  getImBgpDomNameFromDnBgpPeerAf();  - bgp\_obj\_peeraf\_create (); |
| Modify operation |  | MTS\_OPC\_MODIFY\_ImBgpPeerAfPolicyelem | bgp\_process\_peeraf\_notif (&mts\_msg, BGP\_OBJSTORE\_CFG\_UPDATE);  - getPropChgBmpFromNotification();  - - bgp\_cfg\_peer\_attributes(); |
| Delete operation |  | MTS\_OPC\_DELETE\_ImBgpPeerAfPolicyelem | bgp\_process\_peeraf\_delete\_notif (&mts\_msg);   * getImBgpDomNameFromDnBgpPeerAf (); * getImBgpPeerAddrFromDnBgpPeerAf(); * getImBgpPeerAfTypeFromDnBgpPeerAf();   - bgp\_obj\_peeraf\_delete(); |

#### [no] maximum-prefix <max limit number>

**Config**:

router bgp 1

vrf vrf-name

neighbor 101.0.0.1 remote-as 8101

address-family ipv4 unicast

maximum-prefix 10

**Purpose**: To handle creation/update/deletion of maximum prefix defined for a peer BGP neighbor/peer AF objects in a domain under **address family submode**

|  |  |  |  |
| --- | --- | --- | --- |
| CLI | MO | MTS | Handler |
| maximum-prefix <num> | **MO:** <mo name=”MaxPfxP”>  **DN:** sys/bgp/inst/dom-[name]/peer-[[addr]]/af-[type]/maxpfxp  **Property name**: maxPfx  **Property value**: <MaxPfx>  **Class name**: ImCMaxPfxP  **File**: peer.xml |  |  |
| Add operation |  | MTS\_OPC\_CREATE\_ImBgpMaxPfxPPolicyelem | bgp\_process\_maxpfx\_notif (&mts\_msg,  BGP\_OBJSTORE\_CFG\_ADD); |
| Modify operation |  | MTS\_OPC\_MODIFY\_ImBgpMaxPfxPPolicyelem | bgp\_process\_maxpfx\_notif (&mts\_msg, BGP\_OBJSTORE\_CFG\_UPDATE); |
| Delete operation |  | MTS\_OPC\_DELETE\_ImBgpMaxPfxPPolicyelem | bgp\_process\_maxpfx\_delete\_notif (&mts\_msg);  Upon deletion, the default of 10 is set. |

### Mapping of show CLIs to DNs/MOs for rendering

# System Flow

Provide a high level description of how the software modules interrelate. Include diagrams and descriptions of how the modules interact during operation, including, as applicable, flow control diagrams, data flow diagrams, state transitions diagrams, timing diagrams, priorities among modules, interrupt handling, timing/sequencing relationships, error and exception handling, concurrent execution, dynamic creation/deletion of objects, processes, and other aspects of dynamic behavior.

<Body - DSpec>

# Data Structures

Define the format of new data structures and extensions or modifications to existing data structures

<Body - DSpec>

# Description of Algorithms

Detailed description of new and modified code modules. The use of pseudocode, flow charts, and English language descriptions are encouraged. Detail should include, as applicable, the following:

* algorithm(s) used
* constraints, dependencies, limitations, or unusual features in the software’s design
* expected conditions of the software when execution begins
* conditions under which control is passed to other components
* error and exception handling

<Body - DSpec>

## Advanced Configuration

## Configuration Replace

**Example of replaces an entire configuration**

|  |  |  |
| --- | --- | --- |
| Current configuration | Incoming configuration | Resultant configuration |
| "**bgp**" : {  "**inst**" : {  "asn" : "13979"  "snmpTrapSt" : "disable"  "syslogLvl" : "err"  }  } | "**bgp**" : {  "**inst**" : {  "asn" : "13979"  "snmpTrapSt" : "enable"  "**dom-green**" : {  "holdIntvl" : "200"  "kaIntvl" : "100"  "name" : "green"  }  }  } | "**bgp**" : {  "**inst**" : {  "asn" : "13979"  "snmpTrapSt" : "enable"  "**dom-green**" : {  "holdIntvl" : "200"  "kaIntvl" : "100"  "name" : "green"  }  }  } |

*REFERENCE: Appendix section “BGP Object Model”.*

**Example of replaces portions of a configuration**

|  |  |  |
| --- | --- | --- |
| Current configuration | Incoming configuration  (Replace dom-blue) | Resultant configuration |
| "**bgp**" : {  "**inst**" : {  "asn" : "13979"  "snmpTrapSt" : "disable"  "syslogLvl" : "err"  "**dom-green**" : {  "holdIntvl" : "300"  "name" : "green"  "rtrId" : "12.1.8.1"  }  "**dom-blue**" : {  "holdIntvl" : "300"  "name" : "blue"  "rtrId" : "10.10.8.1"  }  }  } | "**bgp**" : {  "**inst**" : {  "**dom-blue**" : {  "name" : "blue"  "rtrId" : "2.1.8.1"  }  }  } | "**bgp**" : {  "**inst**" : {  "asn" : "13979"  "snmpTrapSt" : "disable"  "syslogLvl" : "err"  "**dom-green**" : {  "holdIntvl" : "300"  "name" : "green"  "rtrId" : "12.1.8.1"  }  "**dom-blue**" : {  "name" : "blue"  "rtrId" : "2.1.8.1"  }  }  } |

*REFERENCE: Appendix section “BGP Object Model”.*

**Determine replace diff of an input configuration object and the corresponding MIT MO**

1. Walk the configuration properties of the configuration object and that of the MIT MO
   1. If property exists in both,
      1. If property is a naming property, mark this property as CREATE. *Note that the values of the naming property are the same here.*
      2. Else, mark this property as MODIFY.
   2. If property exists in configuration object but not in MIT MO, then mark this property as CREATE.
   3. If property exists in MIT MO but not in configuration object, then mark this property as DELETE.

|  |
| --- |
| NOTE: MIT MO might contain properties with default values set internally. Mark them as DELETE and have the framework sets them with default values again. This avoids check both the property name and the property value here. [To confirm that this is fine]. |

**MO bgp:Inst**

|  |  |  |  |
| --- | --- | --- | --- |
| Meta | MIT MO | Config object | Object replace diff |
| asn  syslogLvl  snmpTrapSt | asn  syslogLvl  --- | asn  ---  snmpTrapSt | asn CREATE  syslogLvl DELETE  snmpTrapSt CREATE |

*NOTE: DN of this is sys/bgp/inst*

**Determine MIT objects replace diff**

The Management Information Tree (MIT) consists of all MO instances organized in a single tree structure. Each object (other than top:Root) has a parent object, a relative name (RN) that uniquely identifies the object among its siblings, and a distinguished name (DN) that uniquely identifies the object globally. The DN is the concatenation of the RN along the path from the root to the MO, with RNs separated by “/”. The RN is immutable (set once and for all at MO creation time).

|  |
| --- |
| NEXT: For an object that is marked as MODIFY, calculate object differential, i.e. property changes, if any, of an object.  ADD  DELETE  MODIFY  **INPUT**  dom-blue  dom-red  inst-200  inst-100  ospf  dom-green  inst  bgp  sys  sys  **MIT**  inst-100  inst-200  ospf  bgp  inst  dom-green  dom-blue  dom-red  dom-green  inst  inst-200  ospf  bgp  sys |

# Interface Design

Define the interfaces between modules and those external interfaces used by other computer programs & hardware. Include interface parameter information such as:

* parameter name
* purpose
* data type
* size and format
* range or enumeration of values (e.g., 0-99)

Public API: Examine each of your provided APIs from a robustness perspective and consider how the API will react if passed incorrect output. For good APIs you should never crash the device upon invalid input. Instead, an error should be returned. Indicate how you will use the API robustness testing to validate all the interfaces. Indicate if there are specific interfaces for which you will or will not use this technique, and why.

Include the granularity at which you will do this (function, logical component, component package

Include any security and privacy considerations as well as any dependencies on the other products’ interfaces for the new capabilities. Identify for each interface the sources (i.e., the setting/sending entities) and the recipients (i.e., the using/receiving entities). Include OIR interface if applicable.

<Body - DSpec>

# Memory and Performance Impact

Estimate the additional memory requirements due to this feature.

* Identify whether there is a performance impact, either positive or negative, due to this feature.
* Specify if there is a particular performance requirement or limitation for this feature.
* Assess the complexity of the feature, and any necessary impact on existing features and user interface, which might affect backwards compatibility.

**<<TL 9000: ADDITIONAL REQUIREMENT FOR TL 9000 COMPLIANCE>>**

If relevant, consider impact on CPU and disk utilization. Also, consider impact on I/O channels and bandwidth

**<<ISO requirement>>**

<Body>

# Packaging Considerations

Describe the image packaging considerations and list the feature packages.

Describe the impact on software subset feature packages.

**<<ISO requirement>>**

<Body>

# End User Interface/User Experience

Describe how end users will interact with the product to accomplish tasks.

* Define the user interface commands, parameters, screen formats, and error messages of this feature, along with sample screen shots.
* Include end user performance metrics (e.g. time on task, success rates, etc.), security considerations, and command/error extensibility as appropriate
* For routing and bridging protocols –
  + Define how are addresses assigned
  + Define the addresses format
* Review and approve any addition or change in Error Messages by following the guidelines in ["Writing Cisco IOS System Messages", EDCS-77462](http://wwwin-eng.cisco.com/protected-cgi-bin/edcs/edcs_attr_search.pl?doc_num=EDCS-77462)

**<<ISO requirement>>**

<Body>

## REST API

According to the *Cisco APIC REST API User Guide*

The API supports HTTP POST, GET, and DELETE request methods as follows:

* An API command to create or update an MO, or to execute a method, is sent as an HTTP POST message.
* An API query to read the properties and status of an MO, or to discover objects, is sent as an HTTP GET message.
* An API command to delete an MO is sent as either an HTTP POST or DELETE message. In most cases, you can delete an MO by setting its status to *deleted* in a POST operation.

A uniform resource identifier (URI) of this form for an operation on a managed object (MO):

{**http| https**}**://***host*[ *:port* ]**/api/mo/***dn***.**{**json| xml**}[**?***options*]

Use this form for an operation on an object class:

{**http| https**}**://***host*[ *:port* ]**/api/class/***className***.**{**json| xml**}[**?***options*]

The components of the URI are as follows:

* *http://* or *https://* - Specifies HTTP or HTTPS. By default, only HTTPS is enabled. HTTP or HTTP-to-HTTPS redirection, if desired, must be explicitly enabled and configured. HTTP and HTTPS can coexist.
* *host* - Specifies the hostname or IP address of the APIC.
* :*port* - Specifies the port number for communicating with the APIC. If your system uses standard port  numbers for HTTP (80) or HTTPS (443), you can omit this component.
* /api/ - Specifies that the message is directed to the API.
* mo | class - Specifies whether the target of the operation is an MO or an object class.
* *dn* - Specifies the distinguished name (DN) of the targeted MO.
* *className* - Specifies the name of the targeted class. This name is a concatenation of the package name of the object queried and the name of the class queried in the context of the corresponding package.  For example, the class aaa:User results in a *className* of aaaUser in the URI.
* json | xml - Specifies whether the encoding format of the command or response HTML body is JSON  or XML.
* ?*options* - (Optional) Specifies one or more filters, selectors, or modifiers to a query. Multiple option statements are joined by an ampersand (&).

The API operates in forgiving mode, which means that missing attributes are substituted with default values (if applicable) that are maintained in the internal data management engine (DME). The DME validates and rejects incorrect attributes. The API is also **atomic**.

### REST API for Configuration Replace

The API to support HTTP PUT request method as follows:

* Creates or replaces the specified resource with the data that is included in the PUT operation.
* The PUT operation is used to replace or modify an existing resource. If no such resource exists, it is created.
* The request body of a PUT operation must contain the complete representation of the mandatory attributes of the resource.
* If the PUT method succeeds, a "200 OK" Status-Line is returned, and there is no response message body.
* If the user is not authorized to replace the specified resource an error response containing a "403 Forbidden" Status-Line is returned to the client.
* TBD - Additional responses

This API (operation) is **atomic**.

**[TBD – Add examples]**

# Configuration and Restrictions

Identify the known hardware, firmware, software, and external configuration requirements.

Explain what needs doing to start the process or enable the function:

* What other tasks can, or need, to be done?
* When and or why would the user need to perform these tasks?
* Explain if the function is enabled during the setup facility.
* Identify the known limitations or restrictions of the implementation, e.g. do we meet the specification/standard, are there any workarounds or other configuration tips the user need be aware of?

**<<ISO requirement>>**

<Body>

# Software Restrictions and Considerations

Identify the known restrictions and limitations of this feature, including any capability that is not supported by a configuration or any combination of features that cannot be supported concurrently. Identify dependencies with other software products and operating systems, as applicable. Include any restrictions and considerations for assistive technologies and making the product usable by people with disabilities.

<Body - DSpec>

# Firmware Restrictions and Considerations

Define any restrictions in firmware interoperability or configurations. Firmware is defined here to mean a combination of a hardware device and computer instructions and data residing as read-only software on that device. Include any restrictions and considerations for assistive technologies and making the product usable by people with disabilities.

<Body - DSpec>

# Hardware Restrictions and Considerations

Define the minimum, target, and theoretical maximum hardware configurations. Identify hardware that is supported but cannot be fully utilized. . Include any restrictions and considerations for assistive technologies and making the product usable by people with disabilities.

<Body - DSpec>

# External Restrictions and Configuration

Identify dependencies on external products, configurations or other limitation.

<Body - DSpec>

# Source Code

This section is optional. It provides a place for the designer to describe high-level data structures, the Application Programming Interface, and/or pseudo-code for algorithms.

<Body - DSpec>

# Testing Considerations

Identify the special test equipment needs, if not readily available or if an external test suite required.

* Describe test needs for subset feature packages
* Identify the reference test configuration and environment in which the feature will be unit tested
* Include a high-level overview of the types of unit testing that will be done
  + This should include black box as well as white box testing
  + Include the harnesses that will be leveraged
  + Mention if an early field test is to be done

Note that testing accessibility features may require special test equipment, test software, test environment, Cisco’s Accessibility Lab, and/or assistive technologies.

For more information on white box unit test strategy, see the following: [Unit Test Framework - Unit Test Framework - Confluence](http://zed.cisco.com/confluence/display/CTF/Unit+Test+Framework).

For additional information, see Testing Tools & Resources on the Accessibility website at: <http://iwe.cisco.com/web/accessibility/resources>

**<<ISO requirement>>**

<Body>

# Development Unit Testing

Define the approach used in developing the unit test cases. Include a high level overview of the types of unit testing that will be done. This should include black box as well as white box testing. Include the harnesses that will be leveraged and how the testing will map to each included feature. For more info on white box unit test strategy, go to <http://zed.cisco.com/confluence/display/CTF/Home>.

* List the test cases that comprise the unit test plan (or reference the SW unit test plan).
* Identify who executes the tests.
* Define the unit test completion criteria to meet before feature integration into the development trunk and hand-off to Development Test Engineering.
* Identify any special test equipment needs.

<Body - DSpec>

# Initiative, Legal, & Regulatory

<CPDM requirement>

*Product managers are required to document all initiative, legal, & regulatory requirements in the PRD and SFS. The engineering Lead(s) will respond to these requirements in this section. Refer to* [*Engineering Development Directive Job Aid (EDCS – 1278872)*](http://wwwin-eng.cisco.com/cgi-bin/edcs/edcs_info?3470007) *for any additional actions associated with these requirements.*

*If all the initiative, legal, & regulatory product specific requirements have been addressed elsewhere in this document, write in” N/A, addressed elsewhere in this document” in this section – do not delete*

<Body>

# Requirements Traceability Considerations

**<<TL 9000: ADDITIONAL REQUIREMENT FOR TL 9000 COMPLIANCE>>**

Requirements traceability is required for TL 9000 registration. Traceability may be performed manually, or with assistance of automated software tools.

If the requirements in the SFS are not of sufficient detail to support traceability to test cases, you may need to trace the product requirements to the more detailed functional requirements specified in this SW functional specification. In this situation, provide unique identifiers for all functional requirements and trace product requirements to functional requirements consistent with your organization’s strategy for requirements traceability. If the project has a separate traceability matrix provide a link to that document.

Refer to [Common Requirements Traceability Process Handbook, EDCS-400506](http://wwwin-eng.cisco.com/protected-cgi-bin/edcs/edcs_attr_search.pl?doc_num=EDCS-400506), for a thorough description of traceability and some examples of a manual implementation

<Body>

# References

* *NETCONF*, RFC-6241
* *Object Store Software Design Specification*, EDCS-XXX (Currently, not in EDCS)
* *RESTCONF*, IETF draft <http://tools.ietf.org/html/draft-ietf-netconf-restconf-01>
* *UCS Manager Developer’s Guide*, EDCS-849998
* *Standalone DME Infrastructure Software Function Specification,* EDCS-1459142

# Glossary

The following list describes acronyms and definitions for terms used throughout this document:

* **Data Management Engine (DME):** An entity that organizes run-time management information in a MIT and provides API-based, transactional, role-based access to managed objects.
* **Deployment:** Applying changes implied by the information model mutation to applications.
* **Distinguished Name (DN):** A name that uniquely identifies the object in the MIT. It is the concatenation of the RN along the path from the root to the MO, with RNs separated by “/”.
* **Management Information Tree (MIT):** A hierarchical repository of managed objects.
* **Network Element (NE):** An instance of DME with distinct code that is the configuration broker and owner and works in standalone mode.
* **Relative Name (RN):** A name that uniquely identifies the object among its siblings.
* **Stimulus:** Object described in the information model; input to NE/DME.

# Appendix

## BGP Object Model

A look at part of a BGP object model:

|  |
| --- |
| NOTE: “bgp” is name of the package. Followed by name of the MO.  \*  1  bgp:Gr  1  \*  bgp:Peer  \*  1  bgp:DomAf  bgp:Dom  \*  1  bgp:Inst  1  1  bgp:Entity  1  1  top:System |

|  |  |  |
| --- | --- | --- |
| **Object** | | **Description** |
| **bgp:Entity** | BGP control plane entity information | |
| **bgp:Inst** | Global BGP configuration and operation state | |
| **bgp:Dom** | Per VRF configuration and operation state | |
| **bgp:DomAf** | Per VRF per address family configuration and operation state | |
| **bgp:Gr** | Per VRF per address family graceful restart configuration and operation state | |
| **bgp:Peer** | Per VRF per peer configuration and operation state | |

**bgp:Entity**

|  |  |  |
| --- | --- | --- |
| **Property** | **Description** | **Values** |
| **adminSt** | Administrative state | States are enabled (default) and disabled. |

dn : "sys/bgp"

rn : "bgp"

**bgp:Inst**

|  |  |  |
| --- | --- | --- |
| **Property** | **Description** | **Values** |
| **adminSt** | Administrative state | States are enabled (default) and disabled |
| **asn** | Autonomous system number | Range 1..0xffffffff. |
| **snmpTrapSt** | Administrative port snmp trap state | States are disable and enable (default) |
| **syslogLvl** | Syslog level | Levels are alert, crit, debug, emerg, err (default) inform, notif, and warn. |
| **ver** | BGP version | Version is v4 (default) |

dn : "sys/bgp/inst"

rn : "inst"

*NOTE: There is only instance supported in BGP. “asn” is not a naming property.*

**bgp:Dom**

|  |  |  |
| --- | --- | --- |
| **Property** | **Description** | **Values** |
| **mode** | Mode | Modes are external and fabric (default). |
| **rtrId** | Router ID | BGP router ID. |
| **clusterId** | Route reflector ID (cluster ID) | BGP route reflector ID. |
| **kaIntvl** | Keepalive interval | Range 0..3600 in seconds. Default 60. |
| **holdIntvl** | Hold interval | Range 0..3600 in seconds. Default 180. |
| **name** | BGP domain name | Up to 64 alphanumeric characters. Cannot change this name after the object has been saved. |
| **rd** | Route distinguisher | A unique value that extends the formal network address for all routes in a particular VRF. |

dn : "sys/bgp/inst/dom-[name]"

rn : "dom-[name]"

Where [name] is replaced with the input BGP domain name.

# Attachments

As appropriate, attach log sheets, diagrams, schematics, usability research, examples of forms, or other pieces of information used in or generated in the production of the document.

## Review Action Items

Use this section to log meeting minutes from the review of this document and to track review action items to closure. Relevant data includes meeting attendees, issues, and action items. Action item data includes description and owner, status (Open or Closed), and closure date.

In lieu of keeping the action item log here, this section may reference external review records, which capture and track the action items to closure. Examples of these external review records include Review Minutes checked into EDCS and review data captured via Peer Review Request Queue Tool:   
<http://wwwin-tools.cisco.com/prrq/welcome.do>

**<<ISO requirement>>**

<Body>

For guidelines about creating documents using this template, see the [Template Instructions](http://wwwin-eng.cisco.com/cgi-bin/edcs/edcs_info?EDCS-700169). For template comments, contact the [[cpdm-input](mailto:cpdm-input@cisco.com)](mailto:cpdm-input@cisco.com) alias.

End of Document