|  |  |  |
| --- | --- | --- |
|  | Document Number | EDCS-XXXXXX |
| Based on Template | [EDCS-228142 Rev 1](http://wwwin-eng.cisco.com/cgi-bin/edcs/edcs_info?EDCS-228142)7 |
| Created By | Sergey Timokhin |

RESTful Services  
Software Design Specification

This document describes the high level architectural framework and functionality of the RESTful services provided by IFC

Reviewers

*It is the responsibility of the project team to determine the document reviewers and approvers. Once you have completed this document, it will require appropriate review and approval prior to use. Recommend reviewing EDCS -1161410 for guidance on review and approval..*

|  |  |
| --- | --- |
| Department | Name/Title |
|  |  |
|  |  |
|  |  |
|  |  |

*The departments and/or individuals listed above should be notified in advance and given a sufficient time period to review this document*

Modification History

*Modification history should be maintained for this document. EDCS maintains this history, the modification history table may be removed and the following note inserted where the table was. “For Modification History see EDCS.”*

|  |  |  |  |
| --- | --- | --- | --- |
| Revision | Date | Originator | Comments |
| 1 |  |  |  |
|  |  |  |  |

Table of Contents

[1 Problem Definition 4](#_Toc329262541)

[2 Design Considerations 4](#_Toc329262542)

[2.1 Feature highlights 4](#_Toc329262543)

[2.1.1 Direct queries for GET operations 4](#_Toc329262544)

[2.1.2 Access local and remote DME for POST/CREATE 4](#_Toc329262545)

[2.1.3 Support XML/JSON 5](#_Toc329262546)

[2.1.4 Support filtering 5](#_Toc329262547)

[2.2 Public API 5](#_Toc329262548)

[2.2.1 Guiding principles of the interface 5](#_Toc329262549)

[2.2.2 RESTful web services 6](#_Toc329262550)

[2.3 Third Party Relationships – NGINX 6](#_Toc329262551)

[2.3.1 Architecture highlights 7](#_Toc329262552)

[2.3.2 Master and workers 7](#_Toc329262553)

[2.3.3 External modules 8](#_Toc329262554)

[2.4 Security Considerations 8](#_Toc329262555)

[2.4.1 DME Authentication library 8](#_Toc329262556)

[2.4.2 Authentication flow 8](#_Toc329262557)

[3 Functional Structure 9](#_Toc329262558)

[4 System Flow 10](#_Toc329262559)

[4.1 Big picture diagram 10](#_Toc329262560)

[4.2 Forwarder module 10](#_Toc329262561)

[4.2.1 Execution steps 11](#_Toc329262562)

[4.2.2 Processing thread 12](#_Toc329262563)

[4.3 Meta 12](#_Toc329262564)

[4.3.1 URL parser 13](#_Toc329262565)

[4.3.2 Meta API 13](#_Toc329262566)

[5 Data Structures 13](#_Toc329262567)

[6 Description of Algorithms 13](#_Toc329262568)

[7 Interface Design – REST API 14](#_Toc329262569)

[8 End User Interface 16](#_Toc329262570)

[9 Software Restrictions and Considerations 16](#_Toc329262571)

[10 Firmware Restrictions and Considerations 16](#_Toc329262572)

[11 Hardware Restrictions and Considerations 17](#_Toc329262573)

[12 External Restrictions and Configuration 17](#_Toc329262574)

[13 Source Code 17](#_Toc329262575)

[14 Development Unit Testing 17](#_Toc329262576)

[15 Patentability Considerations 18](#_Toc329262577)

[16 References 18](#_Toc329262578)

[17 Glossary 18](#_Toc329262579)

[18 Attachments 18](#_Toc329262580)

[18.1 Review Action Items 18](#_Toc329262581)

List of Tables and/or List of Figures are optional and may be deleted.

List of Tables

[Table 2.x: <name of table> **Error! Bookmark not defined.**](#_Toc15281175)

List of Figures

[Figure 1: <name of figure> **Error! Bookmark not defined.**](#_Toc14144807)

# Problem Definition

Insieme Fabric Controller (IFC) is a distributed system for managing fabric configuration and running state. IFC is deployed as a cluster of multiple appliances (IFCA), each of which contains a set of control functions (DME-based controllers) and management console processes

1. Cluster Controller – responsible for formation of control cluster
2. Policy Repository / Endpoint Registry
3. Observer
4. Topology Manager
5. Boot Director
6. Object Reader – provides read access into local datastore

Each component on every IFCA and switch in the fabric can be accessed and configured from the web server using representational state transfer (REST) software architecture.

This document describes the API and functional description of the solution behind the API.

# Design Considerations

## Feature highlights

### Direct queries for GET operations

Each IFC and ToR will have a reader process that will assist with read-only requests. The idea is not to interact with controllers for read requests to unload the DME processing.

* ToR reader will be able to access only local data. ToR will not be able to re-route requests to other components of the system.
* IFC reader will be able to communicate to remote readers and to run queries in parallel fashion using concepts similar to map reduce.

To improve performance, local reading will be done directly from web server process.

### Access local and remote DME for POST/CREATE

Similar to read requests, NGINX will be able to communicate only to local DMEs.

* ToR will be able to process only local requests.
* IFC will be able to process local requests and to communicate to remote IFCs/ToRs to complete the request.

### Support XML/JSON

REST API will support requests and produce responses in XML and JSON formats.

All REST requests will have information about the output format (.xml/.json)

### Support filtering

Filtering allows defining filter object from URL. Filter object will be passed along with context.

## Public API

Comprehensive RESTful API will be supported with XML and JSON encoding bindings. Both class-level and tree-oriented data access will be provided.

Representational state transfer (REST) is a style of software architecture for distributed systems such as the World Wide Web. REST has emerged over the past few years as a predominant Web service design model. REST has increasingly displaced other design models such as SOAP and WSDL due to its simpler style.

### Guiding principles of the interface

The uniform interface that any REST interface must provide is considered fundamental to the design of any REST service.

* **Identification of resources**

Individual resources are identified in requests, for example using URIs in web-based REST systems. The resources themselves are conceptually separate from the representations that are returned to the client. For example, the server does not send its database, but rather, perhaps, some HTML, XML or JSON that represents some database records expressed.

* **Manipulation of resources through these representations**

When a client holds a representation of a resource, including any metadata attached, it has enough information to modify or delete the resource on the server, provided it has permission to do so.

* **Self-descriptive messages**

Each message includes enough information to describe how to process the message. Responses also explicitly indicate their cacheability.

An important concept in REST is the existence of resources (sources of specific information), each of which is referenced with a global identifier (e.g., a URI in HTTP). In order to manipulate these resources, components of the network (user agents and origin servers) communicate via a standardized interface (e.g., HTTP) and exchange representations of these resources (the actual documents conveying the information).

Any number of connectors (e.g., clients, servers, caches, tunnels, etc.) can mediate the request, but each does so without "seeing past" its own request (referred to as "layering," another constraint of REST and a common principle in many other parts of information and networking architecture). Thus, an application can interact with a resource by knowing two things: the identifier of the resource and the action required—it does not need to know whether there are caches, proxies, gateways, firewalls, tunnels, or anything else between it and the server actually holding the information. The application does, however, need to understand the format of the information (representation) returned, which is typically an HTML, XML or JSON document of some kind, although it may be an image, plain text, or any other content.

### RESTful web services

A RESTful web service (also called a RESTful web API) is a web service implemented using HTTP and the principles of REST. It is a collection of resources, with four defined aspects:

* the base URI for the web service, such as http://example.com/resources/
* the Internet media type of the data supported by the web service. This is often JSON, XML or YAML but can be any other valid Internet media type.
* the set of operations supported by the web service using HTTP methods (e.g., GET, PUT, POST, or DELETE).

The PUT and DELETE methods are idempotent methods. The GET method is a safe method (or nullipotent), meaning that calling it produces no side-effects.

Unlike SOAP-based web services, there is no "official" standard for RESTful web services. This is because REST is an architecture, unlike SOAP, which is a protocol. Even though REST is not a standard, a RESTful implementation such as the Web can use standards like HTTP, URI, XML, etc.

## Third Party Relationships – NGINX

NGINX is a free, open-source, high-performance HTTP server and reverse proxy. It is known for its high performance, stability, rich feature set, simple configuration, and low resource consumption.

Unlike traditional servers, NGINX doesn't rely on threads to handle requests. Instead it uses a much more scalable event-driven (asynchronous) architecture. This architecture uses small, but most importantly, predictable amounts of memory under load. Even if one doesn’t expect to handle thousands of simultaneous requests, one can still benefit from NGINX's high-performance and small memory footprint.

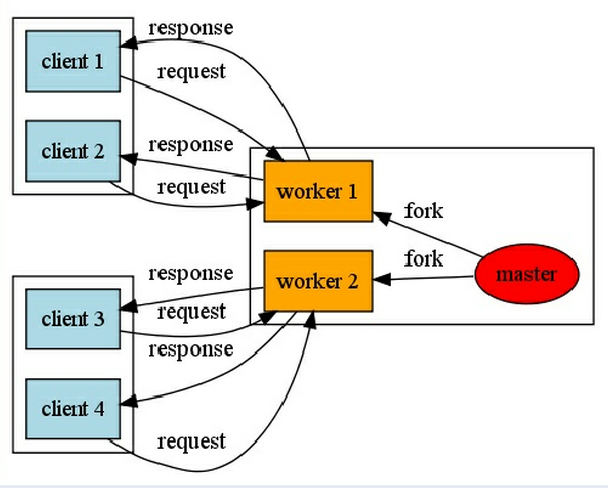
Useful links to documentation and guides:

<http://nginx.org/en/docs/>

[http://wiki.nginx.org](http://wiki.nginx.org/)

<http://www.evanmiller.org/nginx-modules-guide.html>

<http://www.slideshare.net/joshzhu/nginx-internals>



### Architecture highlights

* Non-blocking
* Even-driven
* Single-threaded
* One master process and several worker processes
* Resource efficient
* Highly modular
* Uses memory pool to avoid leaks, defragmentation and alloc/dealloc

### Master and workers

* Master
  + Monitors workers and restarts when worker dies
  + Handle signals and notify workers
* Worker
  + Process client requests
  + Handle connections
  + Get commands from master

### External modules

Core of the NGINX is very thin and most of functionality is provided using specialized modules. There are standard set of modules that is prepackaged with the source code and there is an infrastructure in place that allows adding custom modules with standard interface.

External modules are statically compiled in to the NGINX build.

NGINX modules have three roles:

* Handlers process a request and produce output
* Filters manipulate the output produced by the handler
* Load-balancers choose a backend server

## Security Considerations

### DME Authentication library

DME authentication is a library that encapsulates the user authentication logic. Below are key functions that will be provided by the library.

1. **Login**

* Runs expiration procedure for the user (cleanup expired sessions)
* Check session count
* Create session object in DME and get session UUID

1. **Validate session**

* Check session timeout
* Returns the context object

1. **Refresh session**

* No credentials, session should be valid
* Update the timeout for the session

1. **Logout**

* Decrement session count
* Delete session object in DME

### Authentication flow

DME will be a single place where the session information will be stored and only a unique reference will be passed to the user when the session is started.

User Request with cookie

State and context

Validate session

Request

Authentication

IFC Request and context

Send cookie

Send session UUID

Login

Client

NGINX PAM Wrapper

DME Auth

User Credentials

NGINX Forwarder

IFC

XML/JSON reply

IFC response

**Time**

Data flow:

* User credentials are used to login and create new session using DME authentication library.
* If successful, retuned value of login function is a unique session ID.
* Session UID will be used as a user cookie, all following user requests will have a cookie.
* When request comes in, cookie will be validated using the DME authentication library.
* If session UID validation is successful, function will return roles and domains associated with the user/session.
* Context will be created using destination from meta, user information and request.

For more information about authentication process, please refer to the Insieme Fabric Controller Security documentation.

# Functional Structure

Describe the software modules defining the feature. State each module’s purpose. Describe the major code modules modified, in sufficient detail to analyze for potential impact and testing resources.

<body>

# System Flow

## Big picture diagram

**HTTP**

**IFC**

NGINX Master

NGINX

Worker

NGINX

Worker

META

META

IFC

Reader

IFC

Writer 1

IFC

Writer M

**NGINX Processes**

**DME Processes**

Local Data

Data flow:

* Northbound HTTP request received and processed by one of the child processes of NGINX
* Based on the nature of the request, META layer will create and post the request to the IFC/ToR or access local data directly
* META may need to send the request to multiple DMEs/readers
* After receiving response(s), META will prepare the reply and pass it back to the child processes where it will be packaged and sent back.

## Forwarder module

External forwarder module will be invoked by NGINX for every HTTP request that satisfies the following format:

<*system*>/api/……

Execution of the request will be done in asynchronous fashion because of the single threaded/event-driven nature of NGINX.

Main purpose of the forwarder module:

* Proxy requests to the appropriate DME or set of DMEs
* Handle responses to northbound API.

### Execution steps

**N/B Reply**

**N/B Request**

Return Control

Finalize results

Result ready handler

Result ready event

IFC reply

Request body arrived

Request Body

HTTP Request

Send IFC request

**Time**

NGINX

FORWARDER

IFC

1. New request:
   1. When northbound request comes in, module is invoked without waiting for the request body to arrive.
   2. Module will post a callback to the main process that will be called as soon as the body of the request will arrive.
2. Request body:
   1. Prepare southbound (IFC) request and send it for execution.
3. Reply:
   1. IFC reply arrives and a separate result thread will receive notification.
   2. Packaging of the northbound reply is done in the result thread.
4. NGINX reply:

Request finalization can only be done in the main worker process; therefore the result thread should notify the worker. There are two ways to do that:

* 1. Periodically poll the result structure from the worker using timer subscription mechanism.
  2. Generate the event when the result comes and have a event handler that will be called from the worker.

This flow description is valid for POST requests. For the GET requests it will be simplified since there is no request body.

### Processing thread

To handle asynchronous requests, processing thread will be created as part of the forwarder module. Processing thread will keep track of multiple northbound requests.

Main flow of the thread:

* Accept request packets from NGINX worker thread
* Prepare DME context
  + Parse XML load
  + Query destination from the meta based on the URL
  + Create context from XML and set the destination
* Local read request
  + Perform read using local data store
* Remote read or any write request
  + Send context to one or more destinations using communication channel
* Accept reply callbacks from the lower level communication channel
* Prepare HTTP response packet
  + Read the response from the communication channel
  + Prepare XML or JSON in the NGINX structure that is associated with current response
* Notify the NGINX worker thread when packet is ready

NGINX worker thread will receive notification or will poll the processing thread and will handle the event by sending or finalizing the packets.

In case of a class query, there may be more than one response. Additional logic required to handle the multi response case using approach similar to map reduce concept.

* Send queries to list of destinations identified by the meta.
* Accept each reply individually, controlling the timeouts.
* Send northbound response
  + Do not wait for all the destinations to reply
  + Send blocks back using chunking mechanism

## Meta

The purpose of the meta is to provide location information in the fabric based on the REST API URL.

Also, XML/JSON parser and generator will be part of the Meta functionality.

Meta library will be shared between REST and CLI clients; therefore it will be necessary to parse the URL before using the library.

### URL parser

REST API URL structure is described in the later chapter, for now we can assume that the URL will represent the path to the component and will have options specified after the question mark symbol, including filtering.

URL parser will be generic and will not assume any knowledge about the underlying component structure.

NGINX PRCE or similar module will be used to achieve required functionality.

### Meta API

* **Component type**

Returns component type based on the DN/class of an object

* **Context name**

Returns context name based on the DN/class of an object

* **List of destinations**

Returns list of destinations based on the component type and appliance/node vector

* **XML/JSON parser**

Parses XML/JSON input into the object representation

* **XML/JSON generator**

Produces XML or JSON string based on the binary data

# Data Structures

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

<body>

# 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>

# Interface Design – REST API

DME’s hierarchical object model approach is a very good fit for RESTful interface as URL/URIs map directly into Distinguished Names identifying objects on the tree, and any data on the Managed Information Tree can be described as a self-contained structured text tree document encoded in XML or JSON. Like CMIP, and other X.500 invariants, DME was designed to allow the control of managed resources by presenting their manageable characteristics as object properties. The objects have parent-child relationships that are identified using distinguished names and properties, which are read and modified by a set of CRUD (create, read, update, delete) operations.

Rest API will use standard HTTP commands for retrieval and manipulation of IFC data. The URL format used in the API will be represented as follows:

<*system*>/api/<*component*>/[*mo***|***class***]/[***dn***|***class***][***:method***]**.**[***xml***|***json***]?**{*options*}

* ***system*** – system identifier, an IP address or DNS-resolvable host name
* ***component*** – IFC component (policy manager, observer, node, …)
* ***mo | class*** – indicates if this is an mo/tree (mit) or class level query
* ***class*** – managed object class (as specified in the information model) of the objects queried. Class Name is represented as <pkgName><ManagedObjectClassName>
* ***dn*** – distinguished name (unique hierarchical name of the object on the MIT tree) of the object queried
* ***method*** – optional indication of what method is being invoked on the object, applies only to HTTP POST requests.
* ***xml|json*** – encoding format
* ***options*** – query options, filter, arguments

For example, *iLeaf-55.insiemenetworks.com/node/api/mo/system/card-5/port-4.xml* globally identifies port 4 of card 5 of system iLeaf55.

Class-level URL Format

<*system*>/api/<*component*>/*class***/<pkgName><ClassName>**.**[***xml***|***json***]?**{*options*}

* **pkgName** – represents the package name of the object queried.
* **className** – represents the name of the class queried in the context of the corresponding package.

For example, iLeaf-55.insiemenetworks.com/node/api/class/etherPort.xml will result in resolution of all Ethernet ports in the system, encode in xml format.

Mo-level URL Format:

<*system*>/api/<*component*>/*mo***[/RN]\***.**[***xml***|***json***]?**{*options*}

* **RN** – collection of relative names forming a full distinguished name identifying the path to the managed object on managed information tree

The following common scoping **options** will be supported for queries:

* ***query-target=[\_self\_|children|subtree]*** – specifies whether the object, itself or children, or subtree are to be retrieved
* ***target-subtree-class=[mo-class\*]*** – specify object classes to be retrieved if query-target is other than self
* ***query-target-filter=[****FILTER****]--*** – specify object filters to be retrieved if query-target is other than self
* ***rsp-subtree=[\_no\_|children|full]*** – for objects returned, indicates whether subtree information should be included
* ***rsp-subtree-include=[\_no\_transients\_|no\_faults|no\_stats|config\_only]***
* ***rsp-prop-include=[\_all\_|naming-only|config-explicit|config-all|oper]*** – what type of properties to include into the result
* ***rsp-subtree-filter=[*** *FILTER****]***

These options will apply to all HTTP requests: GET, POST, CREATE, but DELETE.

Standard HTTP Requests will have the following semantics:

* GET – retrieval of objects in scope of the URL and the options provided
* POST – idempotent replace/clobber, with additional per-object/subtree instructions on how to deal with child nodes. Target DN/class and options specify root(s) of subtree(s) where modifications will be applied. Data carried in the request will be in the form of N structured text (XML or JSON) subtrees. It is, therefore, possible to specify a parent tree node target, and affect N subtrees under that node. POST is also capable of executing “methods” on objects in scope.
* CREATE – subtree creation. Subtree targeting, scoping as well as data format is identical to POST.

Just like in CMIP, the notable non-restful aspect of the API is “methods” or “action” operations. Methods provide a way for IFC API to specify functionality that is not possible to implement through generic data queries described above. Methods will be handled as part of HTTP Post request, and will largely have identical object targeting/scoping scheme and options. However, the data payload of the request will be able to carry a set of arguments required for execution of the method. General format of the URL for methods, will be as follows:

<*system*>/api/<*comp*>/[*mo***|***class***]/[***dn***|***class***]*:<method-name>*.[***xml***|***json***]?**{*options*}

Note that since some methods do not apply to subtrees or classes, object targeting/scoping and targeting options can be skipped, reducing request URL to:

<*system*>/api/<*comp*>***:<method-name>*.[***xml***|***json***]?**{*options*}

[*FILTER*] proposed format:

FILTER = OPERATOR(parameter|(FILTER)[/parameter|(FILTER)|value[,parameter|(FILTER)|value]…])

Comma is currently a delimiter, it will be finalized later.

Supported operators

* eq – equality
* ne – inequality
* lt – less than
* gt – greater than
* le – less or equal
* ge – greater or equal
* bw – between
* Logical operators: not, and, or, xor, true, false
* anybit – true if at least one bit is set
* allbits – true if all bits are set
* wcard – wildcard
* pholder – property holder
* passive – passive holder

Example:

We are looking at all fabric ports that are failed in the date range.

query-target-filter = “and(eq(faultevent:type,failed),eq(faultevent:object,fabric\_port), bw(faultevent:timestamp,06-15-12,06-30-12))”

Filter string will be inflated into the DME filtering object and passed along with the request.

# End User Interface

# 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>

# 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>

# 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>

# External Restrictions and Configuration

Identify dependencies on external products, configurations or other limitations. Include any restrictions and considerations for assistive technologies and making the product usable by people with disabilities.

<body>

# 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>

# 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.
* Guidance on how to test for Accessibility Design Requirements that are implemented with this software can be found in the details for each ADR ID located at <http://wwwin.cisco.com/accessibility/requirements/>**.**
* Testing accessibility features may require special test equipment, test software, test environment, Cisco’s Accessibility Lab, and/or assistive technologies. For additional information, see Testing Tools & Resources page on the Accessibility website at: <http://wwwin.cisco.com/accessibility/design_testing/testing_tools_resources/index.shtml>**.**
* Training on test procedures for accessibility features can be found at <http://wwwin.cisco.com/accessibility/training/>
* Record and review the unit test results

<body>

# Patentability Considerations

(may reference SW FS)

Identify any aspects of the system that may be patentable. For guidance on patentable ideas and the patent process, see

<http://wwwin-eng.cisco.com/protected-cgi-bin/cpol/patent.cgi.>

.

<body>

# References

* <*Name of document in italics*>, <document info in plain text>
* <*Name of document in italics*>, <document info in plain text>

# Glossary

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

* **Term 1 <in bold>**: **<**definition in plain text**>**
* **Term 2 <in bold>**: **<**definition in plain text**>**

# 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 Tool: [Peer Review Request Queue](http://wwwin-tools.cisco.com/prrq/welcome.do)

<body>

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

End of Document