CableLabs®
Proactive Network Maintenance

Combined Common Collection Framework Architecture Technical Report

CL-TR-XCCF-PNM-V01-180814

RELEASED

DISCLAIMER

This document is furnished on an "AS IS" basis and neither CableLabs nor its members provides any representation or warranty, express or implied, regarding the accuracy, completeness, noninfringement, or fitness for a particular purpose of this document, or any document referenced herein. Any use or reliance on the information or opinion in this document is at the risk of the user, and CableLabs and its members shall not be liable for any damage or injury incurred by any person arising out of the completeness, accuracy, or utility of any information or opinion contained in the document.

CableLabs reserves the right to revise this document for any reason including, but not limited to, changes in laws, regulations, or standards promulgated by various entities, technology advances, or changes in equipment design, manufacturing techniques, or operating procedures described, or referred to, herein.

This document is not to be construed to suggest that any company modify or change any of its products or procedures, nor does this document represent a commitment by CableLabs or any of its members to purchase any product whether or not it meets the characteristics described in the document. Unless granted in a separate written agreement from CableLabs, nothing contained herein shall be construed to confer any license or right to any intellectual property. This document is not to be construed as an endorsement of any product, or company or as the adoption or promulgation of any guidelines, standards, or recommendations.

Document Status Sheet

Document Control Number:	CL-TR-XCCF-PNM-V01-180814			
Document Title:	Combined Common Collection Framework Architecture Technical Report			
Revision History:	V01 – 08/14/2018			
Date:	August 14, 2018			
Status:	~~Work in Progress~~	~~Draft~~	**Released**	~~Closed~~
Distribution Restrictions:	~~Author Only~~	~~CL/Member~~	~~CL/ Member/ Vendor~~	**Public**

Trademarks

Contents

Figures

Tables

1 SCOPE

1.1 Introduction and Purpose

This document describes an architecture for the Combined Common Collection Framework (XCCF). This XCCF combines data collection for multiple access network technologies for the cable industry, including DOCSIS®, and several optical technologies, including Passive Optical Network (PON) equipment.

DOCSIS 3.1 technology introduces many new features into the access network: some are built upon what was present in previous versions, and others are new features which are needed as a result of the new OFDM-based PHY layer.

Several use cases for the ongoing management and optimization of DOCSIS 3.1 networks have been identified based on systems and tools described in CableLabs technologies as companions to this new protocol. These include the following CableLabs DOCSIS 3.1 network management programs.

- Proactive Network Maintenance (PNM)
- Profile Management Application (PMA)

Each of these management systems or tools is dependent upon the availability of network data to function. The XCCF described in this document provides a standard platform enabling a structured approach to complex data collection in DOCSIS 3.1 and optical networks.

1.2 Scope

This document describes an architecture for the XCCF and DOCSIS 3.1 Common Collection Framework (DCCF).

In Scope:

- Support for the DOCSIS 3.1 protocol
- Support for TFTP and SNMP data collection methodologies
- Support for RESTful API interfaces

Out of Scope:

- Support for pre-DOCSIS 3.1 protocols
- Support for IPDR protocol as described in DOCSIS 3.1 specifications
- System deployment scenarios and considerations for scalability
- Considerations for security and high availability
- User interfaces, analytics, or any application specific behaviors
- Data persistence methodologies or databases
- Application specifics

1.3 Organization of this Document

This document is organized into the following sections.

- Section 1 describes the scope and organization of this document.
- Section 2 describes informative references relevant to this document.
- Sections 3 and 4 provide terms, definitions, abbreviations, and acronyms used in this document.
- Section 5 provides a document introduction and overview.

- Section 6 is an overview of the combined Common Collection Framework (XCCF).
- Section 7 describes the DCCF architecture.
- Appendix I provides HTTP response codes.
- Appendix II provides TFTP server details.
- Appendix III contains a guide to the API.
- Appendix IV is a driver guide.
- Appendix V contains acknowledgements.

2 INFORMATIVE REFERENCES

This technical report uses the following informative references. References are either specific (identified by date of publication, edition number, version number, etc.) or non-specific. For a non-specific reference, the latest version applies.

[CCAP OSSIv3.1] CCAP™ Operations Support System Interface Specification, CM-SP-CCAP-OSSIv3.1-I12-180509, May 9, 2018, Cable Television Laboratories, Inc.

[CM OSSIv3.1] Cable Modem Operations Support System Interface Specification, CM-SP-CM-OSSIv3.1-I12-180509, May 9, 2018, Cable Television Laboratories, Inc.

[DOCS-IF31-MIB] CableLabs DOCSIS DOCS-IF31-MIB SNMP MIB Module, DOCS-IF31-MIB, http://www.cablelabs.com/MIBs/DOCSIS/.

[DOCS-IF3-MIB] CableLabs DOCSIS DOCS-IF3-MIB SNMP MIB Module, DOCS-IF3-MIB, http://www.cablelabs.com/MIBs/DOCSIS/.

[DOCS-PNM-MIB] CableLabs DOCSIS DOCS-PNM-MIB SNMP MIB Module, DOCS-PNM-MIB, http://www.cablelabs.com/MIBs/DOCSIS/.

[GL-PNMP] PNM Best Practices: HFC Networks (DOCSIS 3.0), CM-GL-PNMP-V03-160725, July 25, 2016, Cable Television Laboratories, Inc.

[MULPIv3.1] MAC and Upper Layer Protocols Interface Specification, CM-SP-MULPIv3.1-I15-180509, May 9, 2018, Cable Television Laboratories, Inc.

[NDT] CableLabs Network Dimensioning Tool, https://nougat.cablelabs.com/PNM_Tools/netdimtool (Note: To access, additional information is required)

[NETCONF] M. Bjorklund, Ed. YANG – A Data Modeling Language for the Network Configuration Protocol (NETCONF). Internet Engineering Task Force, October 2010, RFC-6020.

[PHYv3.1] Physical Layer Specification, CM-SP-PHYv3.1-I14-180509, May 9, 2018, Cable Television Laboratories, Inc.

[PMA] DOCSIS 3.1 Profile Management Application Technical Report, CM-TR-PMA-V01-180530, May 30, 2018, Cable Television Laboratories, Inc.

[RESTCONF] IETF RFC 8040, A. Bierman, M. Bjorklund, K. Watsen. RESTCONF Protocol, January 2017.

[RFC 2863] IETF RFC 2863, The Interfaces Group MIB, June 2000

[RFC 6020] IETF RFC 6020, YANG: A Data Modeling Language for the Network Configuration Protocol (NETCONF), October 2010.

[RFC 7159] IETF RFC 7159, The JavaScript Object Notation (JSON) Data Interchange Format, March 2014.

[SDN-ARCH] SDN Architecture for Cable Access Networks Technical Report, VNE-TR-SDN-ARCH-V01-150625, June 25, 2015, Cable Television Laboratories, Inc.

[TR-181] TR-181, Device Data Model for TR-069, Broadband Forum, http://www.broadband-forum.org..

[WCCF] WI-FI PNM Common Collection Framework (WCCF), WR-TR-PNM-WCCF-V01-171010, October 10, 2017, Cable Television Laboratories, Inc.

2.1 Reference Acquisition

- Cable Television Laboratories, Inc., 858 Coal Creek Circle, Louisville, CO 80027; Phone +1-303-661-9100; Fax +1-303-661-9199; http://www.cablelabs.com.
- Internet Engineering Task Force (IETF) Secretariat, 46000 Center Oak Plaza, Sterling, VA 20166, Phone +1-571-434-3500, Fax +1-571-434-3535, http://www.ietf.org.

3 TERMS AND DEFINITIONS

This document uses the following terms.

CCAP	Converged Cable Access Platform. Often used interchangeably with CMTS, CCAP connotes a CMTS that also includes video capabilities.
CMTS	Cable Modem Termination System, often used interchangeably with CCAP.
XCCF	The combined Common Collection Framework, which joins the Wi-Fi Common Collection Framework (WCCF), Optical Common Collection Framework (OCCF), DOCSIS 3.1 Common Collection Framework (DCCF), and other possible future Common Collection Frameworks.

4 ABBREVIATIONS AND ACRONYMS

This document uses the following abbreviations and acronyms.

ACK	Acknowledgement
ADC	Analog-to-Digital Converter
AP	Access Point
API	Application Programming Interface
APP	XCCF Application
C3	CableLabs Common Code Community
CCAP	Converged Cable Access Platform
CCF	Common Collection Framework
CM	Cable Modem
CMTS	Cable Modem Termination System
CMVA	Cable Modem Validation Application
DCA	Distributed CCAP Architecture
DCCF	DOCSIS Common Collection Framework
DCID	Downstream Channel ID
DLC	DCCF Local Cache
DOCSIS	Data-Over-Cable Service Interface Specification
DP	Driver Pool
DRA	DCCF REST API
DRD	DCCF Request Dispatcher
DS	Downstream
DWC	DCCF Workflow Controller
FDX	Full Duplex DOCSIS
FEC	Forward Error Correction
FFT	Fast Fourier Transform
HFC	Hybrid Fiber Coaxial
I&Q	In-Phase & Quadrature
IFS	Inter-frame Space
IoT	Internet of Things
IPDR	Internet Protocol Detail Record

LC	Local Cache
JSON	JavaScript Object Notation
MER	Modulation Error Ratio
MIB	Management Information Base
MoCA	Multimedia over Coax Alliance
MSO	Multiple System Operator
NB	Northbound
NBI	North Bound Interface
NDT	Network Dimensioning Tool
NOC	Network Operations Center
OCCF	Optical Common Collection Framework
OFDM	Orthogonal Frequency Division Multiplexing
OFDMA	Orthogonal Frequency Division Multiple Access
OLT	Optical Line Termination
OID	Object Identifier
ONU	Optical Network Unit
OPT	OFDM Profile Test
PCCF	Power Common Collection Framework
PHY	Physical Layer of the OSI model
PMA	Profile Management Application
PMD	Protocol Module Drivers
PNM	Proactive Network Maintenance
QAM	Quadrature Amplitude Modulation
RAD	RESTful API and Request Dispatcher
REST	Representational State Transfer
R-PHY	Remote PHY
RX	Receiver
SID	Service ID
SNMP	Simple Network Management Protocol
STA	Station
T&M	Time and Materials
TFTP	Trivial File Transfer Protocol
TP	Translator Pool
US	Upstream
VM	Virtual Machine
VSA	Vector Signal Analyzer
WAC	Workflow Access Controller
WCCF	Wi-Fi Common Collection Framework
XCCF	Combined Common Collection Framework
YANG	Yet Another Next Generation

5 COMBINED COMMON COLLECTION FRAMEWORK (XCCF)

5.1 Overview

CableLabs' Proactive Network Maintenance (PNM) program has been met with measurable success by Cable operators working to enhance best practices for DOCSIS 3.0 network operations. With the introduction of DOCSIS 3.1 technology, it is anticipated that PNM adoption will increase while network data complexity and volumes will rise. We are already seeing more complexity and volume; the deployment of Full Duplex DOCSIS (FDX) will add even more.

In addition, new platforms that enhance visibility into other critical segments within the service provider access network infrastructure, both wireless and wireline, will also be addressed by companion PNM initiatives. This gives the Cable operator a common platform and methodology upon which to build enhanced applications to support current and future operations use cases. Network infrastructures within the operators' management domain that would benefit from common PNM practices could include technologies such as Wi-Fi, MoCA, R-PHY, optical, power, and IoT.

It is expected that multiple network technologies will be under PNM management within an operator's infrastructure at the same time. It is also understood that different network types will expose different forms of operational instrumentation, based on information models inherent to their design and deployment disposition. Over the course of broadband network evolution, a number of different network management protocols have been adopted to manage DOCSIS 3.1 networks.

Though data are collected from the same network for multiple uses, they are often gathered by multiple protocols (SNMP, TFTP) embedded in disparate and closed network management systems. This approach adds to the burden of network data collection, making holistic visibility unattainable. Further, many service providers have multiple collectors hitting the same network elements without coordinating efforts. This situation further reduces the capability of operations support systems.

To motivate wider adoption of PNM practices across current and emerging network technologies, a structured approach to network data collection would accelerate development and deployment by abstracting the complexities of multi-network visibility through the support of standard network information models and protocols. The Combined Common Collection Framework (XCCF) provides a structured approach to the collection of data from standards-based network deployments.

The XCCF is a framework; but it is also an architecture to which the entire cable community can contribute, utilize, copy from, and extend as needed. By following the approaches specified in the XCCF, the entire community can depend on interoperation. As a framework, contributions to the XCCF become part of the reference.

5.2 Architectural Goals

Referring to the conceptual model illustrated in Figure 1, the architectural goals of the XCCF are numerous.

- **Collect once and use many**: The XCCF consolidates data collection and presents results that can be consumed by multiple applications, which all require access to the same network data elements to support their individual use cases. This approach serves to reduce the classic network management problem of multiple applications all collecting the same data from the same network element, which would increase the burden on the network layer by consuming resources sub-optimally. The XCCF proposes to reduce network data collection redundancy by allowing multiple applications to access a single collected data instance.
- **Protocol abstraction and simplicity:** The XCCF abstracts underlying network complexities associated with network data collection, including the protocol type used for collection (e.g., SNMP, TFTP, etc.), and expresses data to applications through common information models and APIs.
- **Common infrastructure:** A single framework is used to collect data from different network technologies (e.g., Wi-Fi, DOCSIS, Optical) using a plurality of supported network layer data collection and standard network management protocols (e.g., SNMP, TFTP).

- **Directly programmable**: Network data collection is controlled through common interfaces that expose powerful data gathering capabilities to all applications with access.
- **Extensible and agile**: Abstracting network data collection, lower level device, and network protocol complexities enables rapid development of dynamic applications. The XCCF will support the addition of new protocols, APIs, and data services through a driver architecture of protocol modules. Also, as the API is common, the extensibility and agility extend to the applications.
- **Programmatically configured**: The XCCF lets network managers collect from complex networks via dynamic, automated PNM programs which they can write themselves because the programs do not depend on proprietary software. Network topologies and device inventories are discovered from collected data to eliminate these common tasks from the application layer.

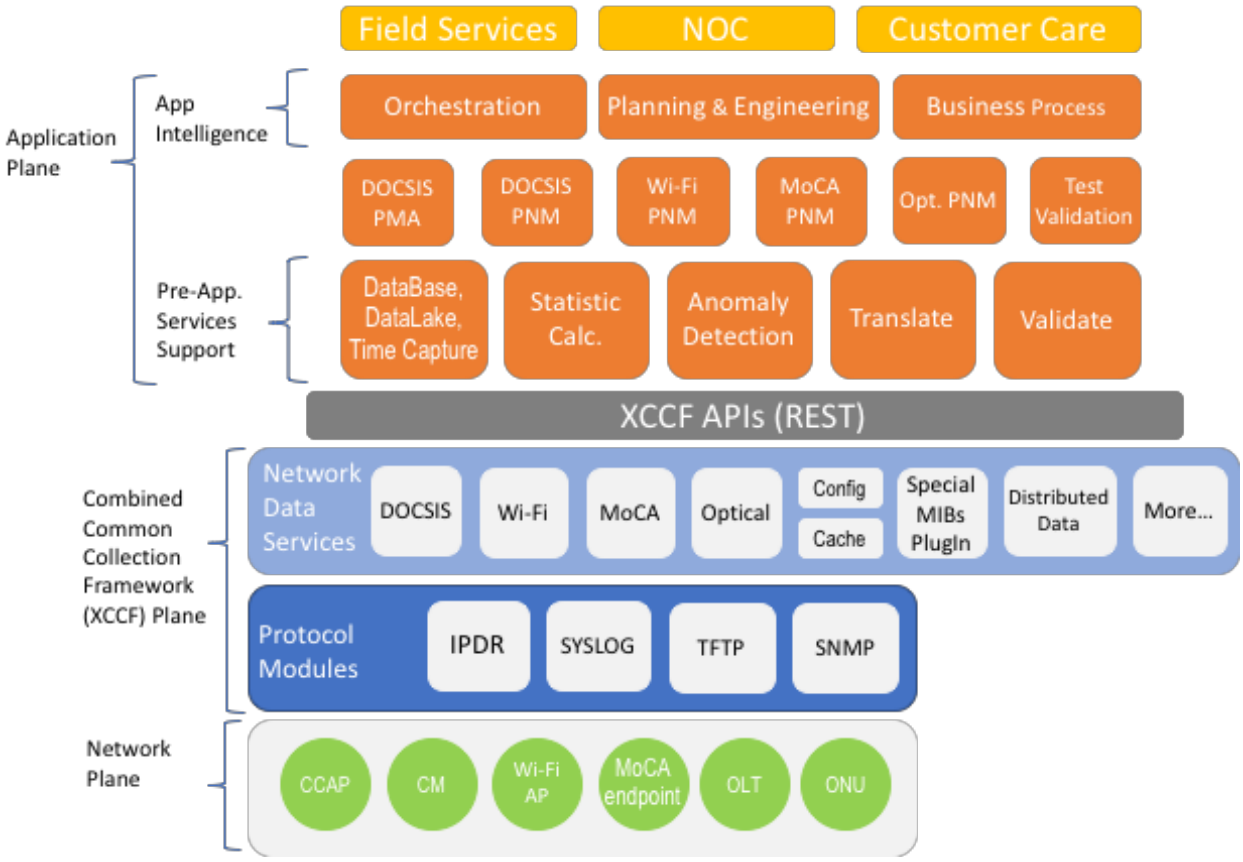

Figure 1: The Combined Common Collection Framework Conceptual Model

- **Open standards-based and vendor-neutral**: When implemented through open standards and open source software, the XCCF simplifies network data collection and application design by automating the tasks of gathering metrics for applications. The XCCF leverages existing standards and data models, including DOCSIS technology and TR-181. With a common API from the collector, used by operators and others in the ecosystem, the XCCF enables a lower barrier for contribution from the community to the C3 for new analytical techniques, accelerating access to innovation across CableLabs members. New protocols can be added to communicate with the network, for the benefit of the whole community.
- **Low level network data processing:** The XCCF includes an environment allowing for the enrichment of collected data, and the generation of low-level derived data products to conform to data model requirements described in the northbound API.

- **Alignment with SDN approach:** Not unlike Software Defined Networking (SDN) concepts specified for the control of network infrastructures such as OpenFlow [SDN-ARCH], the XCCF takes a similar approach, and shares many of the same objectives. In fact, as an upward avenue for data, it would complement many SDN architectures, and provide information by which their control features could be optimized.
- **Rapid deployment of applications:** By providing a flexible platform for data collection, applications can be developed quickly and separately from the data collection burden. By separating the data collection from the application development, all steps in the process are accelerated: architecture, testing, development, prototyping, experimenting.

5.3 Architectural Components

Referring to Figure 1, The XCCF's architectural components are as follows.

- **Application Plane:** The Application plane consists of applications that interact with the XCCF and rely on network data collected and prepared by it. These can be stand-alone, or are often consisting of several supporting capabilities:
  - **App Intelligence:** By adding some form of intelligence to the application space, stand-alone applications, models, work flows, business rules, and other software can be combined to support operations in sophisticated ways.
  - **Pre-App Services Support**: Some applications will have common needs, and an effective operations system will utilize like capabilities when and where possible. By having a pre-app services support layer, such common functions can be developed and maintained consistently.
- **XCCF APIs:** The XCCF exposes an interface for the configuration of data collection, interaction with network topologies, and the execution of data collection and exchange of results. This is presented through standard information models and RESTful APIs that are accessed from the application plane.
- **Network Data Services:** Central to the collector plane, the XCCF implements network data services which are modules managing workflows for data collection and caching, as well as device inventory configuration and topology management.
- **Protocol Modules:** The XCCF supports multiple standard network management and data collection protocols, and is extensible to allow for new protocols to be added to support a plurality of different network architectures.
- **Network Elements:** Network elements are connected devices contained within the network access plane.

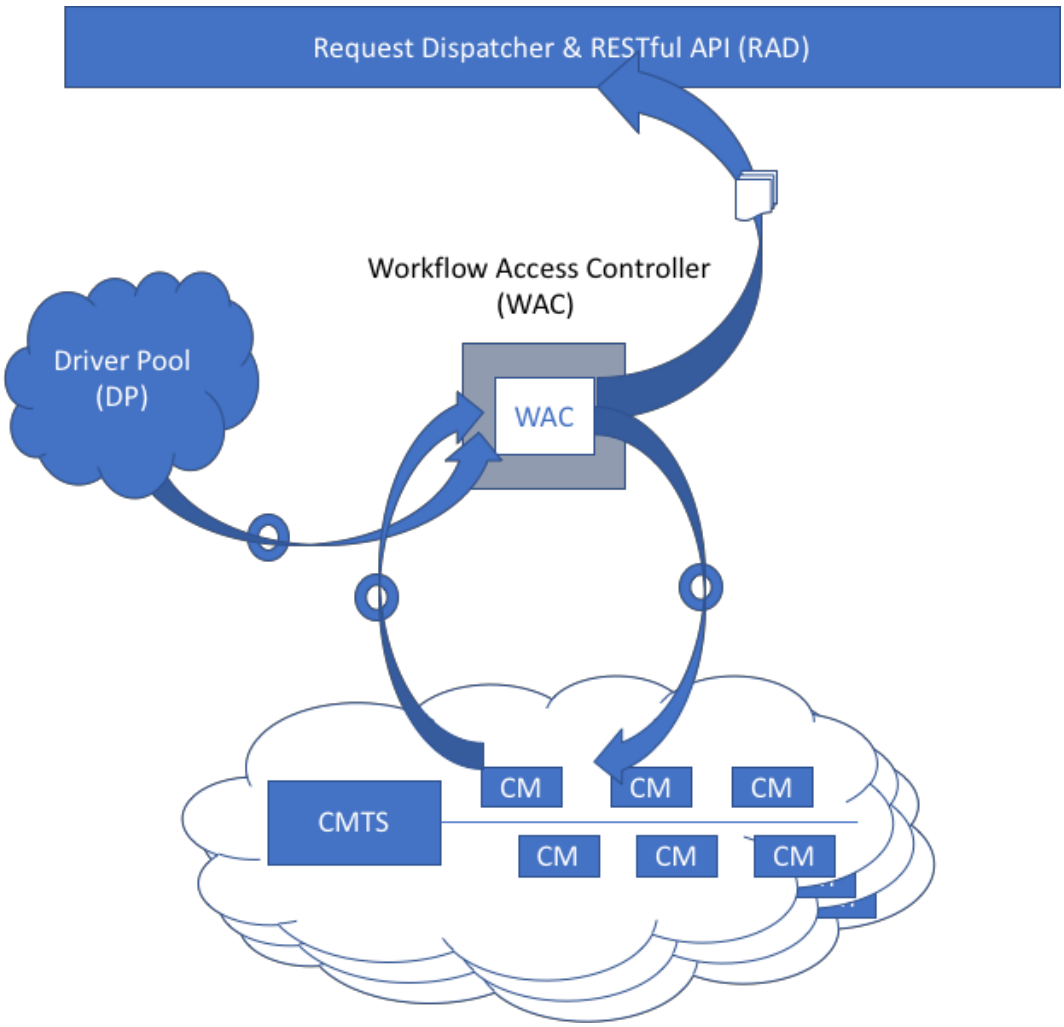

Figure 2: The Combined Common Collection Framework Architecture

Referring to Figure 2, the XCCF's architecture components from a software-oriented view can be catalogued as follows.

- **The RESTful API and Request Dispatcher (RAD):** This section of code forms the northbound interface to applications and services, and communicates southbound to the workflow access controllers. The data are held here in the file structure, or in the TFTP server. The TFTP server, not shown, can be external at the discretion of the user, or the XCCF will create one at the API point. The RAD contains several key functions: mapping requests to the workflow access controllers (WACs), queue handling for WACs, security, file store, inform applications where to receive files and what job IDs to use, and may hold some network inventory information for applications or for XCCF use.
- **Translator Pool (TP):** The translator pool is a set of code objects that can be called upon to translate data formats. The XCCF can be directed by applications to present results in translated forms different from the raw data form, or the raw data can be delivered as well. If an application wants the data translated, then the XCCF will call the correct translator out of the pool to do the job. TP elements are stateless microservices.
- **Workflow Access Controller (WAC):** The workflow access controller manages the work being performed by the XCCF, and the access to the network resources. It decides what metadata needs to be given to drivers, gathers the output from the drivers, keeps track of states and manages them as needed, and keeps track of jobs and exports job identifications to the RAD. Further, it keeps track of CMTS work load and CM work load to

assure overload doesn't happen. It also handles initialization management, and instantiates the drivers from the pool as needed.

- **Driver Pool (DP):** The driver pool contains the drivers that the WAC instantiates to communicate with the network elements. These drivers have several interesting qualities and tasks:
  - Take input from a WAC and translate it into one or more commands to the network for data acquisition.
  - Take responses from the network and present them into a serialized format in standard location for the WAC to accept.
  - Drivers become stateless micro-services.
  - Export metadata like version number, and identify its capabilities to the WAC.
  - Set the interface between the WAC and driver that is a parameter object, and accept the parameter object from the WAC in a set format.
  - It may translate data to a standard format, but not doing the same thing as the translator.
  - Drivers allow outputs to go to alternate destinations; the driver tells the CM where to send the data.
  - The driver closes all loops in requests.
  - Drivers do the communication with the network.

5.3.1 Directories and Files

A data directory is specified in which all data and state information is stored. The datastore is structured as follows.

- A datastore HAS a list of parent nodes.
- A parent node HAS
  - A metadata child node called system
  - A string (usually an address) identifier
  - A list of child nodes.
- A child node IS
  - A metadata node OR
  - A data node
- A metadata node HAS
  - A type
  - - Some arbitrary JSON containing data about the device
- A data node HAS
  - A type
  - A timestamp
  - The actual raw, uninterpreted (by CCF) data

To illustrate, an example directory structure is provided in Figure 3.

Figure 3: XCCF Cache Filesystem Structure, Populated but Redacted to Fit the Page

5.4 Use Case Assumptions

While the XCCF is very flexible, so few assumptions are necessary, we outline here a few assumptions that are helpful to make when applying the XCCF to a deployment.

- A single instance of an XCCF can be used for multiple CMTSs, but a large deployment would benefit from having one XCCF for each CMTS.
- A single WAC for each CMTS is a good approach, even if all WACs point to the same TFTP location, or even if all JSON data are in the same REST. At this time, multiple WACs pointing to the same file system is not tested, but the architecture is built to support it.
- A large deployment should consider a containerized version of XCCF. Lab and test deployments can function well on a reasonably sized VM.
- Assume that the WAC maintains state, and is the only stateful element in the architecture for XCCF.

Several assumptions about the data requests to the XCCF are made:

- All data requests made to the XCCF are for a single CMTS, with one or more CMs assigned to that CMTS, and one or more PNM data elements requested from each CM. Data elements and CMTSs are assumed to be one to one, but multiple modems can be requested using a single data request command.
- If given a list of CMs and a PNM data element are in the same request, interpret the request as a request for the data on all requested CMs.
- For PNM data, the CMTS is identified by IP, the CMs are identified by MAC, and the data elements are identified by PNM type in the string name and are binary data (TFTP).
- For MIB data, the CMTS is identified by IP, the CMs are identified by MAC; the data elements are identified by driver type, and possibly the OIDs, in the string name, and are JSON data (SNMP).
- Any request from an application will be for one identified CMTS. For data requests across multiple CMTSs, use multiple data requests.
- Each CMTS can have only one WAC associated with it, but a WAC could conceivably have more than one CMTS associated with it. This is to support workload management.

5.5 Evolution of the XCCF

It is anticipated that subsystems of the XCCF described in the overall architecture will not be developed simultaneously, but rather introduced incrementally at different times based on priority of use cases and demand.

The first components of the XCCF introduced are those necessary to support DOCSIS 3.1-specific use cases and applications as described in the preceding section. A later section in this document is focused on the architectural description of the DOCSIS 3.1 CCF (DCCF). DOCSIS 3.0 and DOCSIS 2.0 support exists as well. Following that, we discuss extensions.

As of this writing, business needs in the industry are driving us to develop more fully the optical and power versions of the XCCF: OCCF, PCCF. In addition, applications for DOCSIS PNM are the use cases for the XCCF.

6 PROACTIVE NETWORK MAINTENANCE (PNM)

6.1 Overview

DOCSIS 3.1 specifications describe a new network management technique for diagnosing physical layer network issues. The set of new proactive network maintenance features are available to identify performance issues by obtaining data from the following.

- Cable Modem (CM)
- Converged Cable Access Platform (CCAP) or equivalent in Distributed CCAP Architectures (DCAs)
- Both the CM and CCAP collaboratively

These data are obtained using a basic approach of initializing a test at the CM or CCAP using an SNMP set. The CM must execute the test while remaining in service. After test results are obtained, the data are stored in a file on the CM or CCAP. The file can be automatically transferred, or by TFTP to a file server using bulk data file management. The server IP address, directory structure, and filename can be set on the CM or CCAP, or provisioned to the CM or CCAP. This binary test result file can then be decoded and analyzed by a network management system or application. The PNM features along with the process for test initiation and file management are described in [PHYv3.1], [CM OSSIv3.1], [CCAP OSSIv3.1], and [DOCS-PNM-MIB].

The primary diagnostics are intended to provide the same capabilities as found in test and measurement equipment, but with the added benefit of data being available at any network termination point at any time, as shown in Figure 4, excerpted from [PHYv3.1].

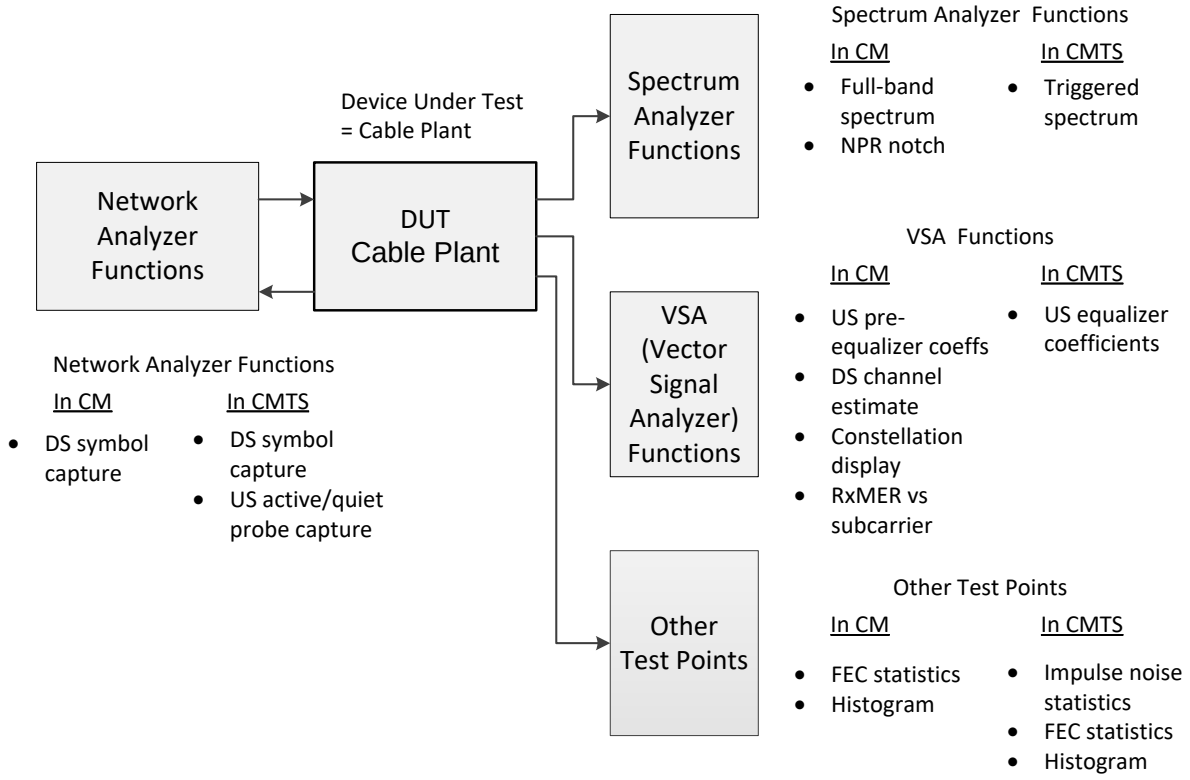

Figure 4: CM and CCAP Test Points Supporting PNM (from [PHYv3.1])

DOCSIS 3.1 PNM specifies a very powerful diagnostic platform for an MSO to identify physical layer issues, and enhance overall access network performance and service quality. For example, user interfaces corresponding to

these tests could be added to the customer support and installation tools. Before sending a technician to a home or a plant maintenance truck roll, these tools can provide diagnostics that are very helpful in determining which type of technician to dispatch and where to dispatch. Many of these functions could address the transient modem diagnostics often challenging for MSOs.

The current set of measurement functions specified by PNM includes those listed in Table 1, Table 2, and Table 3.

6.2 CM PNM Test Functions

Table 1: CM PNM Test Functions

PNM Test Function	Description
`CmDsOfdmRxMer`	At the CM, provides the Modulation Error Ratio per subcarrier for the downstream OFDM channel. This will help determine what modulation orders can be supported by the profiles used by the CM. Several statistical summaries are also provided by the CM's PNM agent. The PNM MIBs include the mean, standard deviation, and X%ile, where X is configurable as an alternative to collecting all values for the CM and calculating from the larger data set. The highest frequency subcarrier with RxMER = to the X threshold value.
`CmDsOfdmRequiredQamMer`	This function provides a target MER value for each downstream OFDM modulation order to be used by the next row function, `CmDsOfdmMerMargin`. The MER is set with quarter Db units with default values shown in [CM OSSIv3.1].
`CmDsOfdmMerMargin`	Provides an estimate of the MER margin available on the downstream data channel with respect to a candidate profile. This is similar to the OPT test message defined in [MULPIv3.1] between the CCAP and CM. The CM calculates the required average MER based on the bit loading for the profile and the `CmDsOfdmRequiredQamMer` in the row above. The CM then measures the RxMER per subcarrier, and calculates the mean for active subcarriers. The CM also counts the number of MER subcarrier values that are below the threshold, and the threshold offset described in the row above for `CmDsOfdmRequiredQamMer`. The threshold offset is configurable.
`CmDsConstDispMeas`	At the CM, provides QAM constellation points for display so technicians can evaluate the impairments on the network. The data are obtained at the slicer input over time, made available for analysis, and normalized to an average constellation power of 1. Each query will include up to 8192 I and Q samples, and additional queries can be used to fill out the display. The constellation is based on the set of profiles received by the CM within the OFDM channel. The modulation order to be captured can be set by the network management system.
`CmSpectrumAnalysisMeas`	Enables a downstream spectrum analyzer within each CM. This function provides the energy content of the signal at each frequency within a specified spectrum band. The frequency is divided into a discrete set of evenly spaced frequency bins across the band. Each bin has an associated amplitude value. The start, stop, span, resolution bandwidth, and bins can be controlled analogous to spectrum analyzer T&M equipment functions.
`CmDsHist`	This function is to provide a measurement of the non-linear distortion effects in the HFC network, such as laser clipping or amplifier compression. The data creates a histogram of time domain samples at the wideband ADC input to the receiver. The distribution is two sided with values from the far negative to the far positive. The bins of the distribution include 255 or 256 bins of equal size. If the end of the bin is truncated or has a spike, it indicates a clipping effect. The CM will report the time frame for the histogram, and its accumulation can be reset.
`CmDsOfdmChanEstimateCoef`	This function captures, at the CM, an estimate of the downstream channel response. The reciprocals of the channel response coefficients are typically used by the CMTS as its frequency-domain downstream equalizer coefficients. The estimate comprises a single complex value per subcarrier. Summary metrics of slope, ripple amplitude, group delay, and mean are also defined and available in the PNM MIB. For example, this function could be used to optimize the channel cyclic prefix setting.
`CmDsOfdmFecSummary`	This object enables collection of codeword error statistics in the downstream OFDM channel. This information is the best indicator of impacted customer experience, as uncorrectable codewords indicate lost packets as described in the F2 section of the SMP. A test can be run for 10 minutes with codeword data obtained every 10 seconds, for a total of 600 measurements; or 24 hours with codeword data collected every 60 seconds, for a total of 1440 measurements. The data are collected for each profile in use by the CM. The FEC data includes the timestamp, total codewords, corrected codewords, and uncorrectable codewords.

PNM Test Function	Description
CmUsPreEq	This function provides access to CM upstream pre-equalizer coefficients. These can be used to identify and locate faults on the network. This is similar data to that described by CableLabs PNM group for DOCSIS 2.0 and DOCSIS 3.0 modems that have proven valuable to service providers. It also includes summary metrics of amplitude ripple, group delay ripple, slope, and mean. This function also provides the status of adjustment requests from the CMTS. These could be used on an OFDMA channel, for example to set the cyclic prefix which affects OFDMA efficiency.

6.3 CCAP PNM Test Functions

Table 2: CCAP PNM Test Functions

PNM Test Function	Description
UpstreamHistogram	This function measures the values into the wideband DAC, and reports the data in 255 or 256 equal sized bins. This approach enables identification of non-linear events, such as laser clipping or amplifier compression. If one end of the histogram is truncated or includes a spike, it indicates an issue such as clipping. The time frame for the histogram is reported with the samples, and the accumulation of samples can be reset.
UsOfdmaActiveAndQuietProbe	This function is used to measure upstream channel response and to view the underlying noise floor by capturing one OFDMA symbol during a scheduled active or quiet probe frame. The active probe frame provides the input and output for a specific CM, identified in the test enabling vector network analyzer functionality. A quiet probe enables observation of the underlying noise and ingress during a quiet period, when no traffic is being transferred. The CMTS returns the captured complex data values scaled to an average power of 1 during the probe frame. The test can be configured to have equalization on or off, and specify the number of OFDMA symbols to capture, which may span a couple of probe frames.
UsImpulseNoise	This function provides statistics on burst and impulse noise in a selected narrow band of unoccupied upstream spectrum. A threshold is set, and energy exceeding the threshold triggers an event. When energy falls below the threshold, the event is ended. The CCAP records and reports the timestamp of the event, the duration of the event, and the average power of the event. Analysis of this data over time can determine the rate and power of the transient noise in the band, which could be used to set interleaver values. The trigger levels and frequency, or the run time (for a threshold of zero) can be configured.
UsOfdmaRxMerPerSubcarrier	This function returns the MER per subcarrier for a specified CM. This data can be used to determine the modulation supported by the channel for each CM and profile. The number of averages and the pre-equalization on/off can be configured. Summary statistics such as mean, standard deviation, and X%ile are provided in the CmtsCmUsStatus table, or can be calculated from the PNM file.
UsSpectrumAnalysis	This function provides a spectrum analyzer view of the upstream that can be triggered to examine specific transmissions, as well as a trigger during a quiet period to measure the noise and interference levels. The measurement can be free running or triggered on a minislot count, MAC address, SID, timestamp, active or quiet probe frame, or an idle (quiet) SID. The frequency and span and number of bins can be specified and configured if in free running mode.
UsOfdmaRxPower	This function provides an estimate of the RX Power for a specific CM on the upstream for a specific OFDMA channel, based on the probe frames. The CCAP measures the average power per subcarrier, and converts that into a power per 1.6 MHz channel. The power can be measured with the pre-equalization on or off, for a configurable number of probes used for the average.

6.4 CM and CCAP joint PNM Test Functions

Table 3: CM and CCAP Joint PNM Test Functions

PNM Test Function	Description
CmSymbolCapture	At the CM, the downstream symbol capture function is to provide analogous functionality to a vector network analyzer in analyzing the frequency response of the cable network from the CM perspective. The complex I&Q time domain samples for a full OFDM symbol before the FFT is captured. The result is a number of samples equal to the FFT length. The CCAP DsOfdmSymbolCapture includes the input symbol that can be used to create the vector network analyzer function.
DsOfdmSymbolCapture	At the CCAP, this is the full OFDM symbol I&Q values before the FFT. These can be used with the CmSymbolCapture at the cable modem to measure the response of the channel.
DsOfdmNoisePowerRatio	This test enables the measurement of the noise, interference, and intermodulation products underneath a portion of the OFDM signal. This is an “out-of-service” test where the CCAP can define an exclusion band of 0 valued subcarriers, which forms a spectral notch for all profiles of a channel. The CM provides its spectrum analysis function described in the table above, or symbol capture functions described in this table. This could be used to observe LTE interference occurring in the OFDM band, or to view non-linear effects. Because the spectrum notch impacts all modems, and will reduce overall capacity, it should only be done during periods of low utilization such as a maintenance window. The duration and start-and-stop frequency can be configured. Because this measure is used in conjunction with the CM functions, it does not by itself return measurement data.
CmSymbolCapture	At the CM, the downstream symbol capture function is to provide analogous functionality to a vector network analyzer in analyzing the frequency response of the cable network from the CM perspective. The complex I&Q time domain samples for a full OFDM symbol before the FFT is captured. The result is a number of samples equal to the FFT length. The CCAP DsOfdmSymbolCapture includes the input symbol that can be used to create the vector network analyzer function.

7 DCCF ARCHITECTURE

7.1 Overview

The remainder of this document describes an architecture for the DOCSIS 3.1 Common Collection Framework (DCCF).

Figure 5 again illustrates the XCCF conceptual architecture, while highlighting the DCCF components from the application, XCCF, and network planes that will be addressed in this section. As the DCCF is considered to be a portion of the overall XCCF, and it encompasses a large portion of what the XCCF does today, this description of the DCCF architecture will be highly leveraged for the other portions of the XCCF.

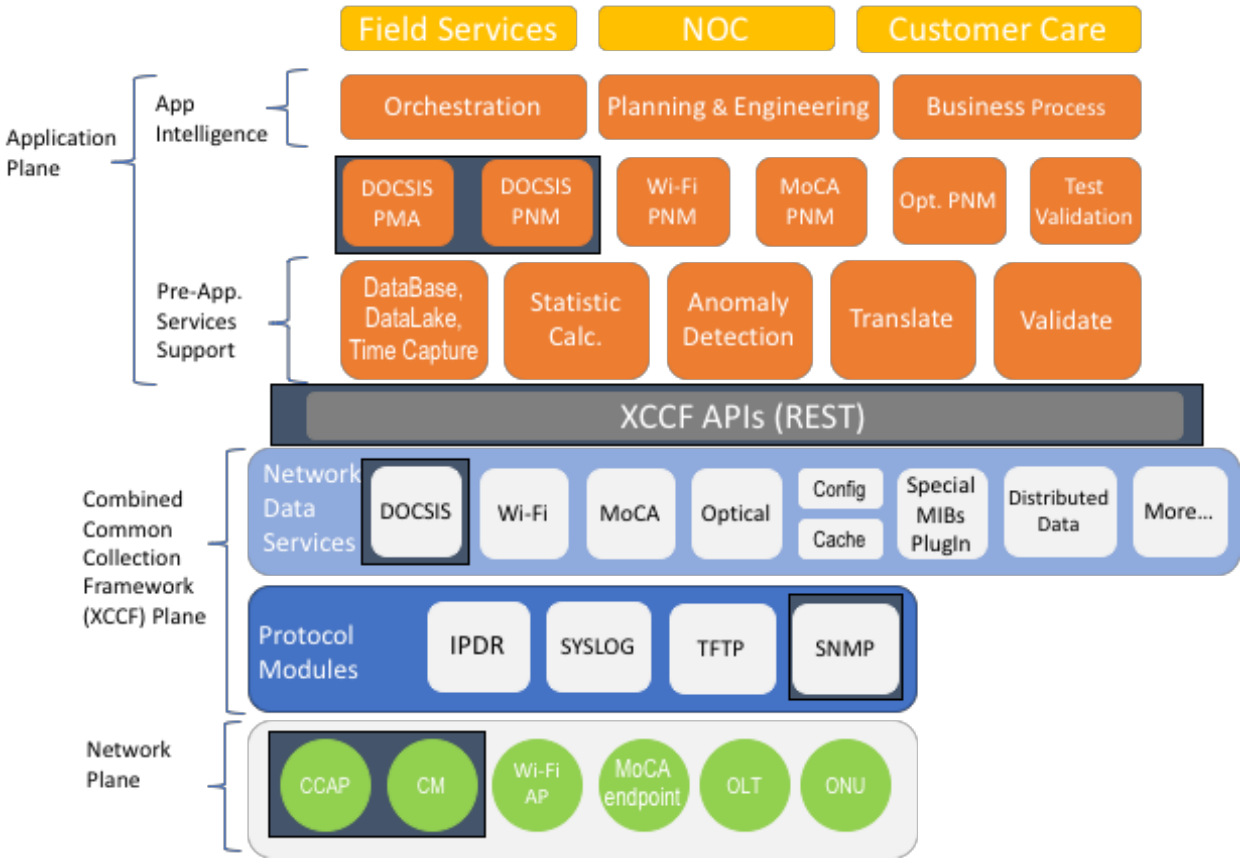

Figure 5: Scope of DCCF within XCCF

As the name indicates, the DCCF is concerned with connecting DOCSIS 3.1 management applications with data instrumented by the underlying DOCSIS 3.1 network plane. This design restricts the DCCF to the following subset of logical components described in the XCCF architecture.

- DOCSIS 3.1 PMA and DOCSIS 3.1 PNM Applications
- DOCSIS 3.1 network data service
- SNMP protocol modules
- DOCSIS 3.1 CCAP and CM devices
- With the inclusion of the ability to poll older devices, we include DOCSIS 3.0 devices as well

The overall DCCF architecture follows the model described for XCCF in the preceding sections.

Note that, while the DCCF is focused on DOCSIS 3.1, it is also DOCSIS 3.0 capable because the XCCF is generally capable.

7.2 DCCF Components

Figure 6 depicts a complex deployment of DCCF with multiple applications and a single RESTful API for multiple WACs and CMTSs. The rest of this section will discuss some of the details in this reference deployment.

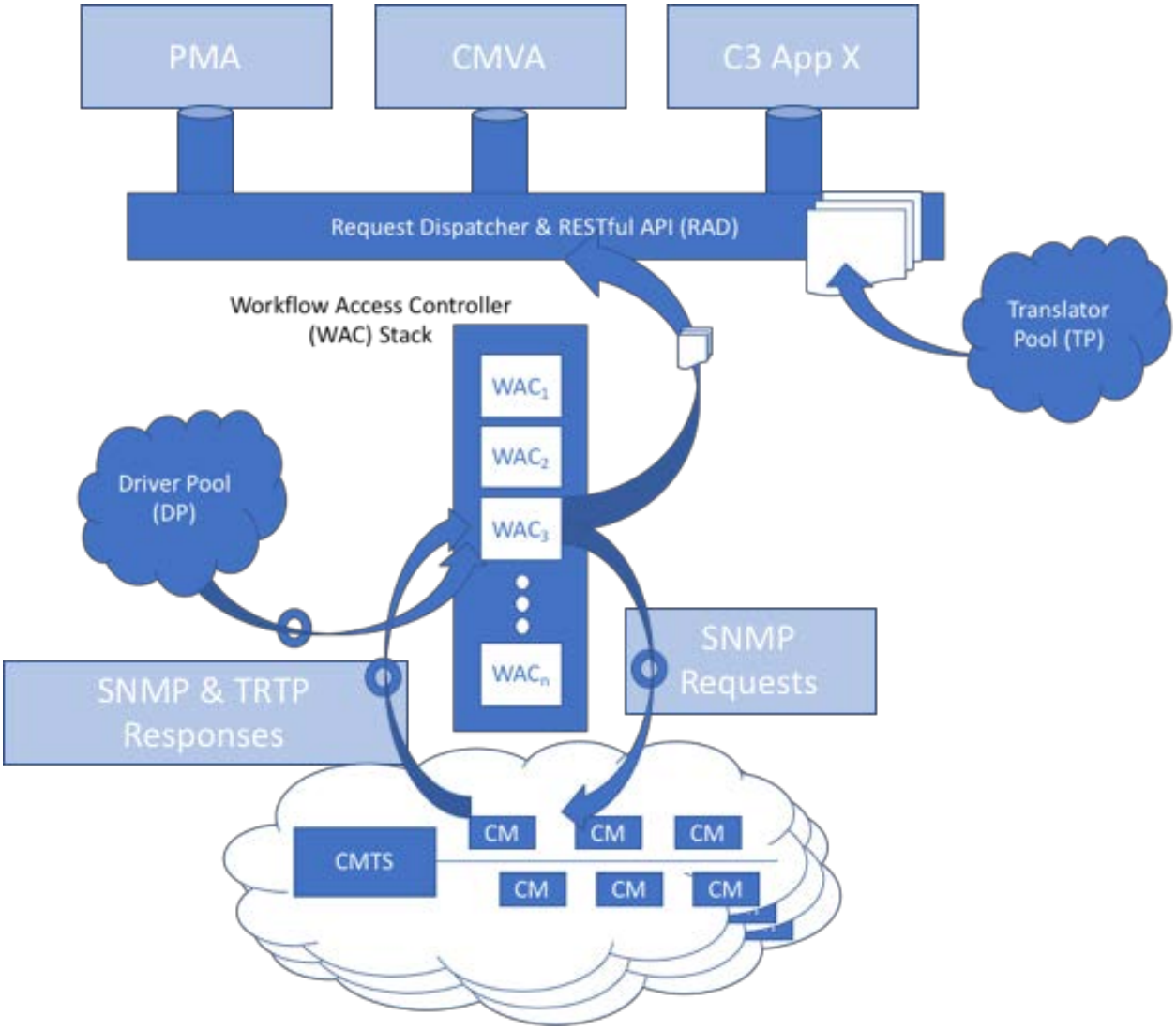

Figure 6: DCCF Component Architecture

7.2.1 Applications

We envision several use cases associated with DCCF (and equivalent use cases with other XCCF components). Each use case classification can drive the development of multiple applications, which may overlap with the applications in other classifications.

- Maintenance
  - Proactive
    - Anomaly Detection

        - Impairment Location
        - Work prioritization and clustering
    - Reactive
        - Fault locating
        - Work prioritization and clustering
        - Quality measurements
- Performance Optimization
    - Profile Management
    - Small Cell Placement
    - Work assignment
- Planning and Engineering
    - Network Dimensioning Tool
    - Plant Health
    - Network Degradation Estimation
- Testing
    - Cable Modem Validation Application
    - CMTS Validation

The overall architecture for applications can vary, but we envision a few possible approaches, and will pursue these for supportive reference designs to some extent. Possibilities include but are not limited to the following.

- Workflow based applications – specific work flows can be supported with custom applications.
- Microservices – application functions and services or objects can be created and combined to support applications.

We expect that each application implements a Representational State Transfer (REST) client conformant with the XCCF RESTful API.

7.2.2 Request Dispatcher and RESTful API (RAD)

The RAD manages all messaging to and from the northbound interface in accordance with the API definition described in Section 7.4.

The RAD is responsible for coordinating interactions between northbound client requests and interactions with services internal to the DCCF. The RAD shares job identifiers to inbound create request operations, and forwards requests and results to and from the RAD.

7.2.3 Workflow Controller (WAC)

The WC coordinates interaction between the RAD and lower-level services that execute either network or storage-based operations.

7.2.3.1 Protocol Module Drivers (PMD)

Protocol module instructions (such as a SNMP GET operation on a CM) are embodied in PMDs under the direction of the workflow controller. Protocol module drivers execute low-level operations by interacting with the underpinning network protocol modules. Multiple PMDs exist, and more can be added to perform different network

tasks. All PMDs are executed by the workflow controller. A trivial example of a PMD would be a script that performs an SNMP GET request to a CM, collecting the agent's instance of sysDescr.

7.2.4 Local Cache (LC)

The DCCF LC (DLC) stores the results of all job operations executed by protocol modules. The DLC uses a simple structured Unix filesystem to organize DCCF data.

The DCCF REST stores the results of data requests. PNM data are delivered to the TFTP server, either your own, or the default provided by the XCCF. SNMP responses arrive as MIB responses, which are held in the JSON file structure.

7.2.5 Protocol Modules

The DCCF supports a total of two (2) standard network management protocols as described in [CCAP OSSIv3.1] and [CM OSSIv3.1]:

- SNMP
- TFTP

7.3 DCCF Example Workflows

7.3.1 Create a New CCAP

In this example workflow, a job request is made by a DCCF Application (APP) to add a CCAP device to a DCCF instance. The job executes a series of tasks including an investigation of the CCAP system, basic topology, discovery of the DOCSIS 3.1 CMs attached to the CCAP, and the configuration of each DOCSIS 3.1 CM found with the necessary PNM TFTP configuration.

Referring to Figure 7, the following sequence of operations is executed.

1. The DCCF application generates an HTTP POST request to the DCCF REST API for a create CCAP job. The POST request includes both the IP address of the CCAP and the SNMP public community string in a JSON attachment.
2. The DCCF REST API immediately returns a "200 OK" to the application with a unique job identifier {job_id}.
3. The DCCF REST API forwards the request for a CCAP creation job to the workflow controller (WC).
4. The WC executes protocol module driver instructions as a series of network initialization and discovery commands. It begins with an SNMP protocol operation to the CCAP requesting system data. The first represents a SNMP GET request to the CCAP to gather system data.
5. The SNMP module executes the GET request, and retrieves system information from the CCAP.
6. Once successfully returned by the SNMP protocol module, the result of the data collection is transformed into a JSON-encoded document, and stored in the DCCF local cache.
7. The next SNMP operation is then executed under the supervision of the WC. In the second job operation, the SNMP protocol module is instructed to gather basic topology information from the CCAP CM registration status table.
8. The SNMP module manages a series of successive GET operations on the CCAP to retrieve the DOCSIS 3.1 CM topology data from the CCAP.
9. Once successfully returned by the SNMP protocol module, the result of the DOCSIS 3.1 CM data collection is transformed into a JSON-encoded topology document, and stored in the DCCF local cache.
10. The final step (optional) of the create CCAP and initialization job is to configure all DOCSIS 3.1 CM devices with the necessary PNM configuration to generate a single PNM files and upload to the DCCF TFTP server.
11. For each CM found in the CM topology discovery step, the CM is configured with the necessary PNM data using SNMP set operations.

12. Upon completion, the results of the transaction and the state of the job are written to the DCCF cache.
13. The DCCF APP which requested the creation of the CCAP requests the status of the job to determine if the CCAP was successfully created and initialized in the DCCF. The APP generates a GET request to the jobs resource of the corresponding {job_id} generated during the POST request and response in the original create CCAP job.
14. The DCCF REST API forwards the request to the RAD, which accesses the cache by corresponding job identifier.
15. The RAD verifies that the job has completed and the initialization data exists.
16. The REST API returns a "200 OK" message to the DCCF APP, including a JSON attachment that describes the status of the job.

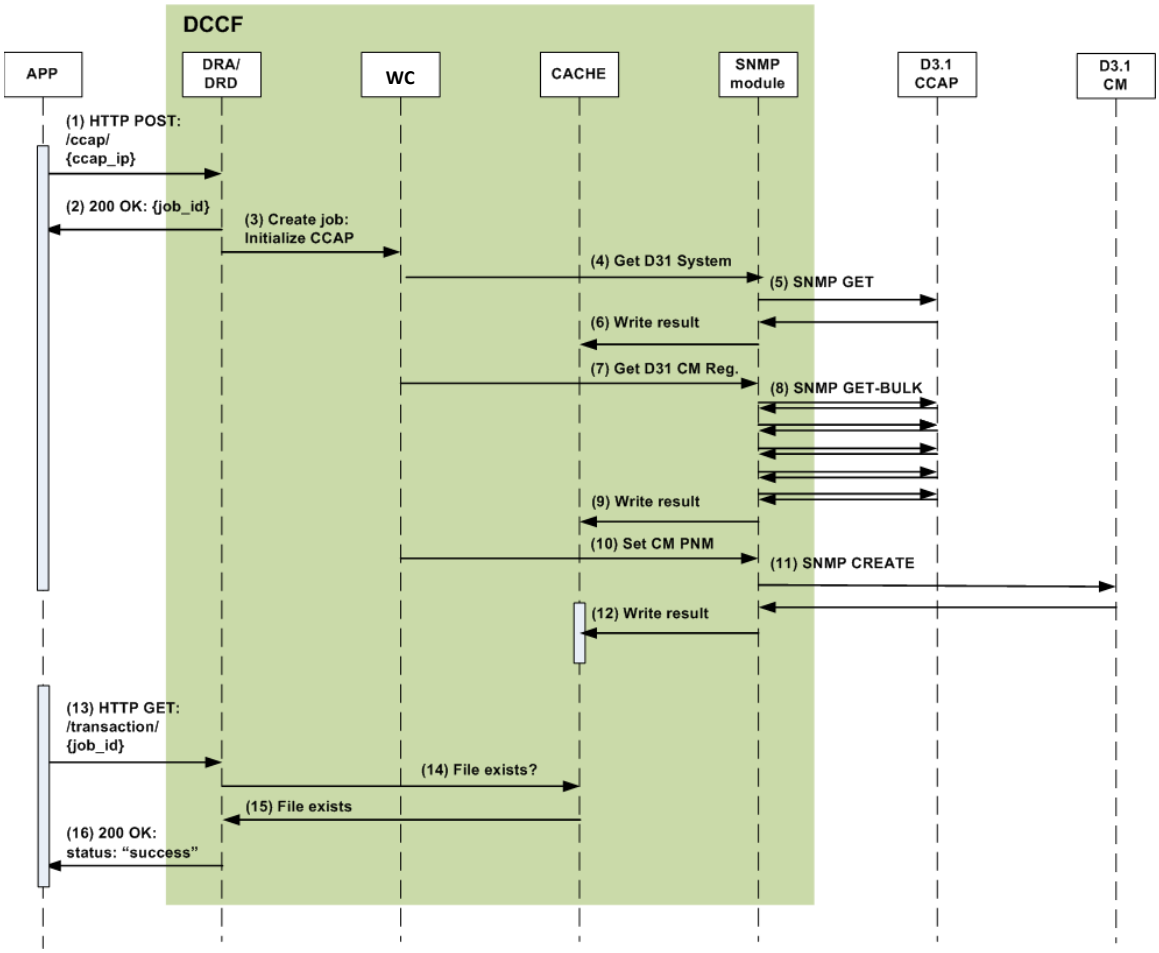

Figure 7: An Example of a Successful CCAP Creation Job Sequence

7.3.2 Get Registered DOCSIS 3.1 CMs

In this workflow, the DCCF APP requests all DOCSIS 3.1 CMs registered on a CCAP. (Note that CMs not registered in DOCSIS 3.1 mode are filtered, and not actively managed by the SNMP protocol driver in this flow; support for DOCSIS 2.0 and DOCSIS 3.0 CMs exists in the most recent version.) The APP generates a total of three requests during the DOCSIS 3.1 CM topology discovery workflow.

Referring to Figure 8, the steps representing these requests are as follows.

1. The DCCF APP generates a POST to create a DOCSIS 3.1 CM Registered job to the DCCF REST API.

2. The REST API RAD returns a status report attachment indicating “success” as an encoded JSON file. The DCCF REST API generates a “200 OK” response to the WCCF APP which includes the job’s identifier {job_id}.
3. The DWC executes the low-level SNMP protocol drivers necessary to perform the DOCSIS 3.1 basic topology collection exercise.
4. The SNMP protocol module interacts with the CCAP to complete the DOCSIS 3.1 basic topology discovery operation.
5. The REST API RAD forwards the create job request to the DWC.
6. Once successfully returned by the SNMP protocol module, the result of the DOCSIS 3.1 CM data collection is transformed into a JSON-encoded topology document and stored in the DCCF local cache.
7. The DCCF APP confirms that the job is completed by generating a request for job status.
8. The DCCF REST API forwards the request to the RAD, which accesses the cache by corresponding job identifier.
9. The RAD verifies that the job has completed and the initialization data (including topology) exists.
10. The DCCF REST API returns a “200 OK” message to the DCCF APP, including a JSON attachment that describes the status of the job.
11. The DCCF APP retrieves the results of the successful job by issuing a GET read request for the CCAP IP to the REST API.
12. The REST API RAD retrieves the job output from the local cache.
13. A “200 OK” response is returned to the DCCF, including an attachment that contains the topology describing the DOCSIS 3.1 CM registered job output.

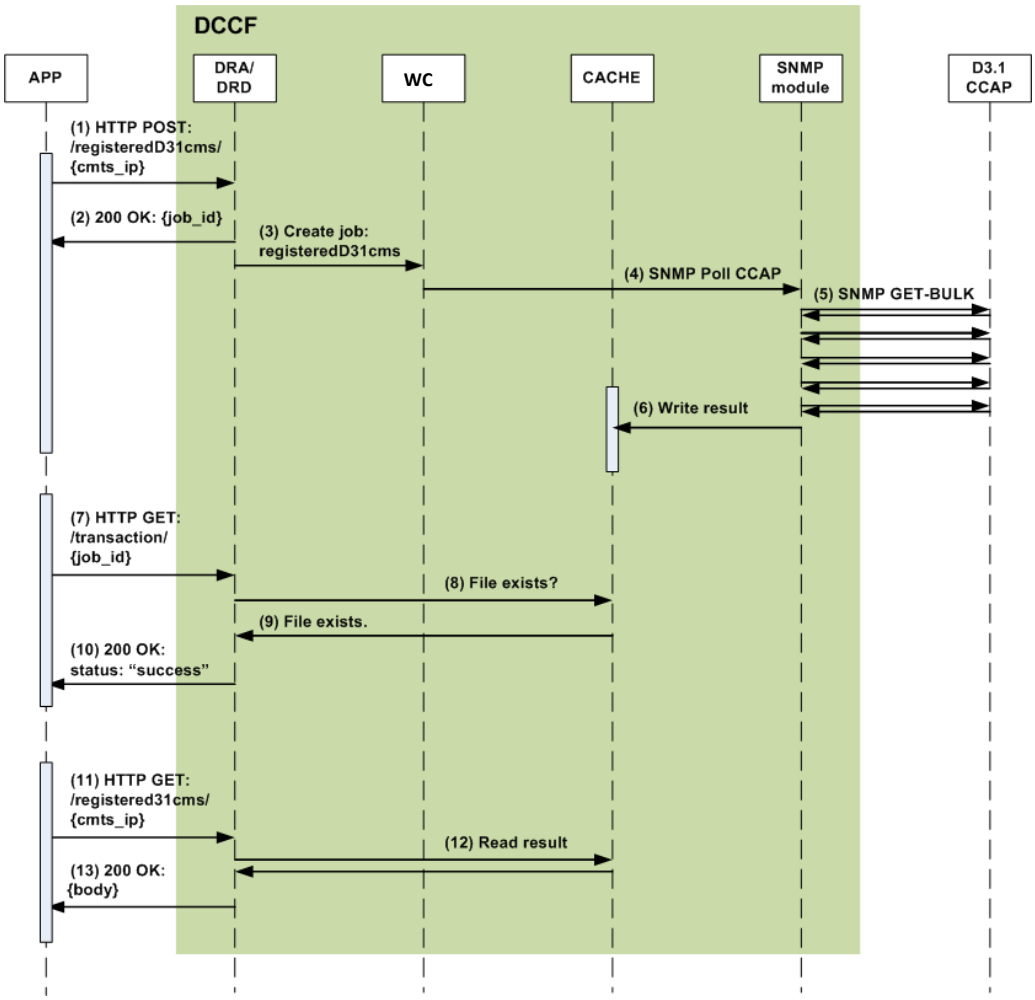

Figure 8: An Example of a Successful D31CmRegistered Job Sequence

7.4 Interfaces

7.4.1 Northbound Interface

7.4.1.1 Protocol Model

The DCCF Northbound Interface (NBI) API is organized around REST. The goal of the API is to expose predictable, resource-oriented URLs, and HTTP response codes to express API errors.

7.4.1.2 Data Interchange Format

The DCCF uses JavaScript Object Notation (JSON) as a lightweight, text-based, language-independent data interchange format. JSON was derived from the ECMAScript Programming Language Standard. JSON defines a small set of formatting rules for the portable representation of structured data [RFC 7159].

All PNM files are received as binary files from the CM or CCAP via TFTP. These binary files are stored and delivered in the original binary format to preserve data integrity. A PNM file dump program is provided to convert these binary files to JSON or CSV formats.

7.4.1.3 REST API Definitions

7.4.1.3.1 Summary of API Calls

The following table is current for version 1.3 of DCCF.

Table 4: List of REST API calls in DCCF v1.3

HTTP Method	Resource	Universal Resource Locator (URL)
POST	CCAP	/dccf/ccaps/(CCAP)/cmPNMDsRxMer
POST		/dccf/ccaps/(CCAP)/cmPNMDsRxMerAll
POST		/dccf/ccaps/(CCAP)/cmPNMFecSum
POST		/dccf/ccaps/(CCAP)/initialize
POST		/dccf/ccaps/(CCAP)/registered31cms
GET		/dccf/ccaps/(CCAP)/registered31cms
POST		/dccf/ccaps/(CCAP)/system
GET		/dccf/ccaps/(CCAP)/system
POST		/dccf/ccaps/(CCAP)/topology
GET	CCAP, CMMAC	/dccf/ccaps/(CCAP)/cms/(CMMAC)/cm
GET		/dccf/ccaps/(CCAP)/cms/(CMMAC)/cmDsFecStats
POST		/dccf/ccaps/(CCAP)/cms/(CMMAC)/cmDsFecStats
POST		/dccf/ccaps/(CCAP)/cmDsFecStats
GET		/dccf/ccaps/(CCAP)/cms/(CMMAC)/cmDsOfdmChanTable
POST		/dccf/ccaps/(CCAP)/cms/(CMMAC)/cmDsOfdmChanTable
POST		/dccf/ccaps/(CCAP)/cmDsOfdmChanTable
GET		/dccf/ccaps/(CCAP)/cms/(CMMAC)/cmDsOfdmChannelPowerTable
POST		/dccf/ccaps/(CCAP)/cms/(CMMAC)/cmDsOfdmChannelPowerTable
POST		/dccf/ccaps/(CCAP)/cmDsOfdmChannelPowerTable
GET		/dccf/ccaps/(CCAP)/cms/(CMMAC)/cmDsOfdmRxMerTable
POST		/dccf/ccaps/(CCAP)/cms/(CMMAC)/cmDsOfdmRxMerTable
POST		/dccf/ccaps/(CCAP)/cmDsOfdmRxMerTable
GET		/dccf/ccaps/(CCAP)/cms/(CMMAC)/cmPNMDsRxMer
POST		/dccf/ccaps/(CCAP)/cms/(CMMAC)/cmPNMDsRxMer
GET		/dccf/ccaps/(CCAP)/cms/(CMMAC)/cmPNMFecStats
POST		/dccf/ccaps/(CCAP)/cms/(CMMAC)/cmPNMFecSum
GET	CCAP, CMMAC, OID	/dccf/ccaps/(CCAP)/cms/(CMMAC)/cmTable/(OID)
POST		/dccf/ccaps/(CCAP)/cms/(CMMAC)/cmTable/(OID)
GET	CCAP, OID	/dccf/ccaps/(CCAP)/ccapTable/(OID)
POST		/dccf/ccaps/(CCAP)/ccapTable/(OID)
GET	JOBID	/dccf/jobs/(JOBID)
GET	Topology	/dccf/ccaps/(CCAP)/topology

In the sections that follow, example calls are documented.

7.4.1.3.2 Initialize CCAP

Title	**Initialize CCAP** Creates and initializes a new CCAP in the DCCF. Triggers initialization for CCAP and all connected CMs.
URL	/dccf/ccaps/(CCAP)/initialize
Method	POST
URL Params	**Required:** ccap_ip

Data Params	Attached JSON file with community string info:
	{
	"commstr": "<community string for CCAP>",
	"cm_commstr": "<community string for CM>",
	"cm_commstr_common": "True"
	}

Success Response	Returns identifier (Job ID) corresponding to the executing job.
Sample Call	curl -X POST -d @/opt/dccf/conf/dccf_ccap_rflab.json -H 'Content-Type: application/json' http://${DCCF_HOST}:8888/dccf/ccaps/${CCAP_IP}/initialize?doPnm=1
Notes	• DRA passes request to WAC • DRA returns JOB ID to requester • WAC executes SNMP GET CCAP System Driver • WAC executes SNMP GET CM Registration Driver • WAC executes SNMP SET PNM for ALL CMs discovered on CCAP

7.4.1.3.3 Discover CCAP Registered DOCSIS 3.1 CMs Job

Title	**Discover CCAP Registered DOCSIS 3.1 CMs** Discover (or rediscover) the DOCSIS 3.1 CMs on a given CCAP. Use this to update the DOCSIS 3.1 CM list of an already discovered CCAP.
URL	/dccf/ccaps/(CCAP)/registered31cms
Method	POST
URL Params	**Required:** ccap_ip
Data Params	n/a
Sample Call	curl -X GET http://${DCCF_HOST}:8888/dccf/ccaps/${CCAP_IP}/registered31cms
Notes	• RAD passes request to WAC • RAD returns job ID to APP • WAC executes SNMP GET CM Registration Driver

7.4.1.3.4 GET CCAP Registered DOCSIS 3.1 CMs Job

Title	**Get CCAP Registered DOCSIS 3.1 CMs Report** Retrieve list of the DOCSIS 3.1 CMs on a given CCAP.
URL	/dccf/ccaps/(CCAP)/registered31cms
Method	GET

URL Params	**Required:** ccap_ip
Data Params	n/a
Sample Call	curl -X GET http://${DCCF_HOST}:8888/dccf/ccaps/${CCAP_IP}/registered31cms

GET Return	{
	"function": "wc_get_ccaps_registered31cms",
	"json_data": [
	[
	"64777D5E5D50",
	"10.0.2.20",
	true,
	"56"
	],
	[
	"1056118E5F69",
	"10.0.3.6",
	true,
	"205"
	],
	[
	"9050CA21B7D0",
	"10.0.2.22",
	true,
	"206"
	],
	[
	"64777DE45830",
	"10.0.3.31",
	true,
	"128"
	],
	[
	"A84E3FCA5B50",
	"10.0.3.8",
	true,
	"222"
	],
	[
	"A84E3F3747B0",
	"10.0.3.5",
	true,
	"145"
	]
	],
	"message": "wc_get_ccaps_registered31cms: Completed on CCAP 065097252066 Status code: 200",
	"results_in": [
	"json_data"
	],
	"status": "OK",
	"status_code": 200
	}

Notes	• RAD passes request to WAC • RAD returns job ID to APP • WAC executes SNMP GET CM Registration Driver

7.4.1.3.5 Generate CCAP Topology

Title	**Generate CCAP Topology** Create a job to execute the deep topology discovery
URL	/dccf/ccaps/(CCAP)/topology
Method	POST
URL Params	**Required:** ccap_ip
Data Params	n/a
Sample Call	curl -X POST http://${DCCF_HOST}:8888/dccf/ccaps/${CCAP_IP}/topology
Notes	1. DRA passes request to WAC 2. DRA returns JOBID to requester 3. WAC executes snmpGetTopology driver

7.4.1.3.6 Get CCAP Topology

Title	**Get CCAP Topology** Create a job to get the deep topology discovery data
URL	/dccf/ccaps/(CCAP)/topology?topo_type=<NAME>
Method	POST, GET
URL Params	**Required:** ccap_ip topo_type <Others based on topo_type, see below>
Data Params	n/a

Sample Call	Return topology information based on parameter topo_type. Additional parameters may be required: **Parameters:** **topo_type** – Required, specified <TYPE> of info to return, see list following for valid values. **md_index** – MAC domain index. Number, e.g. 16000004 **ofdm_index** – OFDM index. Number, e.g. 14000021 **ofdma_index** – OFDMA index. Number, e.g. 20000212 **cm_mac** – MAC address of CM to filter results by. Undelimited, e.g., 1CABCD123456 Where <TYPE> is: - ccap_summary - md_summary (requires md_index) - ofdm_modems (requires ofdm_index) - ofdma_modems (requires ofdma_index) - ofdm_detail (requires ofdm_index) - ofdma_detail (requires ofdma_index) - cms_in_md (requires md_index) - cm_detail (requires md_index and cm_mac) - all_profiles Not all parameters are required, if unrequired parameters are specified they are ignored. Examples: curl -X POST http://10.10.10.10:8888/dccf/ccaps/192.168.1.1/topology curl -X GET http://10.10.10.10:8888/dccf/ccaps/192.168.1.1/topology?topo_type=ccap_summary curl -X GET http://10.10.10.10:8888/dccf/ccaps/192.168.1.1/topology?topo_type=md_summary&md_index=16000004 curl -X GET http://10.10.10.10:8888/dccf/ccaps/192.168.1.1/topology?topo_type=ofdm_modems&ofdm_index=14000021 curl -X GET http://10.10.10.10:8888/dccf/ccaps/192.168.1.1/topology?topo_type=cm_detail&md_index=16000004&cm_mac=1CABCD123456
Notes	1. RAD passes request to WAC 2. RAD returns JOBID to requester 3. WAC executes snmpGetTopology driver

7.4.1.3.7 Get CCAP Topology (Simplified)

Title	**Get CCAP Topology** Create a job to get the deep topology discovery data
URL	/dccf/ccaps/(CCAP)/cmOfdmChanTopo
Method	POST, GET
URL Params	**Required:** ccap_ip
Data Params	n/a

Sample Call	Return topology information based on parameter topo_type. Additional parameters may be required: **Parameters:** **ccap_ip** – IP address of the CCAP. Not all parameters are required, if unrequired parameters are specified they are ignored. Examples: curl -X POST http://10.10.10.10:8888/dccf/ccaps/192.168.1.1/cmOfdmChanTopo curl -X GET http://10.10.10.10:8888/dccf/ccaps/192.168.1.1/cmOfdmChanTopo

GET return	{
	"timestamp": 1532101534.644599,
	"ifindex_topology": {
	"264902": {
	"cm_mac_list": [],
	"ds_ofdm_channel_id": "159"
	},
	"264390": {
	"cm_mac_list": [
	"5CE30E72B249",
	"5CE30E72B505",
	"5CE30E72B0D5",
	"5CE30E72A5AD",
	"5CE30E72B04D",
	"5CE30E72B625"
	],
	"ds_ofdm_channel_id": "159"
	},
	"264903": {
	"cm_mac_list": [],
	"ds_ofdm_channel_id": "160"
	},
	"265414": {
	"cm_mac_list": [
	"5CE30E5C9485",
	"0050F112E00E",
	"5CE30E5CD5E1",
	"5CE30E5CD4B1"
	],
	"ds_ofdm_channel_id": "159"
	},
	"265158": {
	"cm_mac_list": [
	"B0DAF9170215"
	],
	"ds_ofdm_channel_id": "159"
	}
	},
	"cmts_sysdescr": "<*redacted system description*>",
	"cmts_ip": "10.32.40.68"
	}

Notes	1. RAD passes request to WAC 2. RAD returns JOBID to requester 3. WAC executes snmpGetTopology driver

7.4.1.3.8 Create Single CM downstream RX MER Job

Title	**Create single CM DS RX MER Job** Create CM DS FEC statistics job for a single CM.
URL	/dccf/ccaps/(CCAP)/cms/(CMMAC)/cmPNMDsRxMer
Method	POST
URL Params	**Required:** ccap_ip cm_mac
Data Params	n/a
Sample Call	$curl -X POST http://10.10.10.10:8888/dccf/ccaps/10.51.1.3/cms/fc528d5e8f37/cmDsRxMer
Notes	1. Collects DsRxMER PNM file from ONLY the specified CM on CCAP 2. RAD passes request to WAC 3. RAD returns JOB ID to requester 4. Workflow controller executes snmpSetPnm driver for all CMs

Appendix I HTTP RESPONSE Codes

Response status codes are part of the HTTP specification, and there are quite a number of them to address the most common situations. In the spirit of having our RESTful services embrace the HTTP specification, our Web APIs should return relevant HTTP status codes. For example, when a resource is successfully created (e.g. from a POST request), the API should return HTTP status code 201. A list of valid HTTP status codes is available which lists detailed descriptions of each.

Suggested usages for the "Top 10" HTTP Response Status Codes are as follows.

200 OK

General success status code. This is the most common code, and is used to indicate success.

201 CREATED

Successful creation occurred (via either POST or PUT). Set the Location header to contain a link to the newly-created resource (on POST). Response body content may or may not be present.

204 NO CONTENT

Indicates success but nothing is in the response body, often used for DELETE and PUT operations.

400 BAD REQUEST

General error for when fulfilling the request would cause an invalid state. Domain validation errors, missing data, etc. are some examples.

401 UNAUTHORIZED

Error code response for missing or invalid authentication token.

403 FORBIDDEN

Error code for when the user is not authorized to perform the operation or the resource is unavailable for some reason (e.g. time constraints, etc.).

404 NOT FOUND

Used when the requested resource is not found, whether it doesn't exist or if there was a 401 or 403 that, for security reasons, the service wants to mask.

405 METHOD NOT ALLOWED

Used to indicate that the requested URL exists, but the requested HTTP method is not applicable. For example, POST */users/12345* where the API doesn't support creation of resources this way (with a provided ID). The Allow HTTP header must be set when returning a 405 to indicate the HTTP methods that are supported. In the previous case, the header would look like "Allow: GET, PUT, DELETE".

409 CONFLICT

Whenever a resource conflict would be caused by fulfilling the request. Duplicate entries, such as trying to create two customers with the same information, and deleting root objects when cascade-delete is not supported are a couple of examples.

500 INTERNAL SERVER ERROR

Never return this intentionally. This is the general catch-all error when the server-side throws an exception. Use this only for errors that the consumer cannot address from their end.

Appendix II TFTP Server Details

II.1 TFTP Server Configuration

The XCCF doesn't distribute a TFTP server along with the code by default. This is so users can have more flexibility in their choice of TFTP server implementation as well as in deployment configuration. Yet TFTP servers are an important component of the DOCSIS 3.1 PNM measurement collection process. If you're more interested in DOCSIS 3.0 measurements or things that are strictly SNMP, you don't need to read this section of the document. The XCCF, to successfully retrieve DOCSIS 3.1 PNM data from cable modems, needs to be configured with two things: where the TFTP server is on the network, and how to retrieve the files from it to present to the user in the API once the modem has uploaded them.

II.2 Uploading to the TFTP Server

DOCSIS 3.1 PNM measurements are all collected according to the following process. First, check if the modem is ready to do a test by reading an SNMP object on the modem that contains this status. If not ready, stop. Next, do some SNMP sets on the modem telling it what measurement we want, and where, once the measurement is done, to upload via the TFTP protocol, the measurement data.

So, for the XCCF to present users with the results of these measurements, it needs to know the address of a TFTP server, and be able to somehow retrieve those results once they are there. The net result of all this is that you need to run a TFTP server somewhere, and tell XCCF about it.

II.3 Location (IP Address) of the TFTP Server

Per the above process, the XCCF needs to be able to set the modem to upload its measurements to a server. So, the user of an XCCF instance needs to have a TFTP server running. We recommend tftp-hpa, which is available on most Linux distributions. You tell the XCCF where the tftp server is by setting the DCCF_TFTP_SERVER variable in the wc.json configuration file, or whatever config file you pass to the workflow controller. In the below config file snippet, the XCCF will direct modems to upload their results to the TFTP server at the ip address '1.2.3.4'. The user is responsible for making sure a TFTP server is indeed running on this IP.

```
{
  "DCCF_TFTP_SERVER": "1.2.3.4"
}
```

II.4 Instructing XCCF to Present TFTP Server Data

Now that the XCCF has told the modem to upload to a given TFTP server, the XCCF needs to then get that data from the server to present to the user. Users tell the XCCF how to do this via the tftp_type configuration file variable in wc.json, or whichever config file you're passing to the workflow controller. We detail the possible values below.

II.4.1 The 'localfilesystem' Value

You can tell the XCCF that the TFTP server is running on the same machine as the XCCF and it will retrieve a given file from the local file system and put it (the file) in its (XCCF's) datastore accordingly. Optionally, you can tell the XCCF to then delete the TFTP server's copy by setting the 'local_tftp_files_remove' configuration variable to true. An example configuration file snippet might be:

```
{
    ...
    "DCCF_TFTP_SERVER": "<your-server-ip-v4-or-v6-address>",
    "upload_path": "/",
    "tftp_type": "localfilesystem",
```

```
    "local_tftp_server_directory": "/var/lib/tftpboot",
    "local_tftp_files_remove": true,
    ...
}
```

This code will direct the XCCF to look under /var/lib/tftpboot on the local filesystem for the files it expects modems to upload. It will, per the local_tftp_files_remove variable, then remove those files from the /var/lib/tftpboot directory after putting them in the local datastore where they're available through the REST api.

II.4.2 The 'noretrieve' Value

If tftp_type is set to 'noretrieve', the XCCF will make no attempt to retrieve the files once they get uploaded to the TFTP server. It only functions as a way to tell the modems to do a measurement.

```
{
  ...
  "tftp_type": "noretrieve",
  ...
}
```

II.4.3 The 'remoteretrieve' Value

With this value, CCF will wait until the measurement is done, wait for a backoff period, and then attempt to retrieve with TFTP the file that it told the modem to upload from the same server.

```
{
  ...
  "tftp_type": "remoteretrieve",
  "tftp_wait_time": 300,
  ...
}
```

II.5 Example Configs

We have a TFTP server on 10.95.254.177, which happens to be our own local host since we're configured for localfilesystem, and the TFTP server is configured with a root at /var/lib/tftpboot. The modems will use / as the path prefix.

```
{
    "DCCF_TFTP_SERVER":     "10.95.254.177",
    "DCCF_RA_HOST":         "localhost",
    "DCCF_RA_PORT":         8888,
    "DCCF_PROCESS_POOL":    10,
    "SNMP_TIMEOUT":         2,
    "SNMPV3_DH_PRIME":
"0xFFFFFFFFFFFFFFFFC90FDAA22168C234C4C6628B80DC1CD129024E088A67CC74020BBEA63B139B22514A08798E
3404DDEF9519B3CD3A431B302B0A6DF25F14374FE1356D6D51C245E485B576625E7EC6F44C42E9A637ED6B0BFF5CB
6F406B7EDEE386BFB5A899FA5AE9F24117C4B1FE649286651ECE65381FFFFFFFFFFFFFFFF",
    "SNMPV3_MGR_RAND":
"0xC6628B80DC1CD129024E088A67CC7402FFFFFFFFFFFC90FDAA22168C234C4C6628B80DC1CD129024E04DDEF951
9B3CD3A431B302B0A6DF25F14374FE1356D6D51BBEA63B139B2251C1356D6D51C245E485B576625E7EC6F44C42E9A
637ED6B0BFF5CB6F406B7EDEE386BFB5A899FA5AE9F24117C4B1FE649286651ECE65381FF",
    "SNMPV3_USE":           "False",
    "tftp_type": "localfilesystem",
```

```
    "host":         "localhost",
    "port":         8080,
    "upload_path": "/",
    "local_tftp_server_directory": "/var/lib/tftpboot"
}
```

Retrieve the TFTP files from the remote server 10.95.254.188 that we'll tell them to upload to. Wait 300 seconds after the test measurement (not upload) is done on the modem which is an SNMP variable before retrieval.

```
{
    "DCCF_HOME":            "/home/ccf/data",
    "DCCF_CONF":            "DCCF_HOME/dccf_conf",
    "DCCF_DATA_LOCATION":   "DCCF_HOME/data",
    "DCCF_TFTP_SERVER":     "10.95.254.188",
    "DCCF_RA_HOST":         "localhost",
    "DCCF_RA_PORT":         8888,
    "DCCF_PROCESS_POOL":    10,
    "SNMP_TIMEOUT":         2,
    "SNMPV3_DH_PRIME":
"0xFFFFFFFFFFFFFFFFC90FDAA22168C234C4C6628B80DC1CD129024E088A67CC74020BBEA63B139B22514A08798E
3404DDEF9519B3CD3A431B302B0A6DF25F14374FE1356D6D51C245E485B576625E7EC6F44C42E9A637ED6B0BFF5CB
6F406B7EDEE386BFB5A899FA5AE9F24117C4B1FE649286651ECE65381FFFFFFFFFFFFFFFF",
    "SNMPV3_MGR_RAND":
"0xC6628B80DC1CD129024E088A67CC7402FFFFFFFFFFFC90FDAA22168C234C4C6628B80DC1CD129024E04DDEF951
9B3CD3A431B302B0A6DF25F14374FE1356D6D51BBEA63B139B2251C1356D6D51C245E485B576625E7EC6F44C42E9A
637ED6B0BFF5CB6F406B7EDEE386BFB5A899FA5AE9F24117C4B1FE649286651ECE65381FF",
    "SNMPV3_USE":           "False",
    "tftp_type": "remoteretrieve",
    "tftp_wait_time": 300,
    "host":         "localhost",
    "port":         8080,
}
```

Upload all files to 10.95.254.188 for tests, but don't even attempt to retrieve them to present on the api.

```
{
    "DCCF_HOME":            "/home/ccf/data",
    "DCCF_CONF":            "DCCF_HOME/dccf_conf",
    "DCCF_DATA_LOCATION":   "DCCF_HOME/data",
    "DCCF_TFTP_SERVER":     "10.95.254.188",
    "DCCF_RA_HOST":         "localhost",
    "DCCF_RA_PORT":         8888,
    "DCCF_PROCESS_POOL":    10,
    "SNMP_TIMEOUT":         2,
    "SNMPV3_DH_PRIME":
"0xFFFFFFFFFFFFFFFFC90FDAA22168C234C4C6628B80DC1CD129024E088A67CC74020BBEA63B139B22514A08798E
3404DDEF9519B3CD3A431B302B0A6DF25F14374FE1356D6D51C245E485B576625E7EC6F44C42E9A637ED6B0BFF5CB
6F406B7EDEE386BFB5A899FA5AE9F24117C4B1FE649286651ECE65381FFFFFFFFFFFFFFFF",
    "SNMPV3_MGR_RAND":
"0xC6628B80DC1CD129024E088A67CC7402FFFFFFFFFFFC90FDAA22168C234C4C6628B80DC1CD129024E04DDEF951
9B3CD3A431B302B0A6DF25F14374FE1356D6D51BBEA63B139B2251C1356D6D51C245E485B576625E7EC6F44C42E9A
637ED6B0BFF5CB6F406B7EDEE386BFB5A899FA5AE9F24117C4B1FE649286651ECE65381FF",
```

```
        "SNMPV3_USE":           "False",
        "tftp_type": "noretrieve",
        "host":         "localhost",
        "port":         8080,
    }
```

Appendix III Guide to the API

The XCCF is designed to only be a kind of caching and categorization service. That is, the XCCF just runs drivers that obtain data from network devices for you and saves their results. Accordingly, when you ask the XCCF to do something, you specify two pieces of information in the path:

- The addresses needed to get to the network device and to obtain all the metadata about the device needed to access it, e.g., SNMP community strings.
- The type of information you want to access which we call the 'data element'.

Accordingly, an XCCF path looks like this.

/dccf/ccaps/1.1.1.1/initialize

This calls the initialize driver for the ccap 1.1.1.1. You would do a POST to this path and provide some JSON content in the body:

```
{ "commstr": "public",
  "cm_commstr": "public" }
```

This content is some extra metadata that will be cached by the XCCF, which tells it how to access the ccap and its modems. The initialize element collects several different data elements from modems and the ccap specified. You cannot collect the initialize element directly, but you collect the system and cm metadata elements.

Here is another path:

/dccf/ccaps/1.1.1.1/cms/ab:cd:ef:00:11:22/cmPNMDsRxMer

As a POST call, this path asks the XCCF to find a driver that can handle this data element, and then ask the driver to collect it. The XCCF will pass what metadata about the ccap and cm that the driver needs to obtain the information. This information typically entails the SNMP community strings and metadata about the CM and CMTS configuration.

III.1 Oidmap Elements

The oidmap elements all conform to the basic convention outlined above. Oidmaps are just lists of oids that clients might like to collect at the same time. For example,

/dccf/ccaps/1.1.1.1/cms/ab:cd:ef:00:11:22/cmDsFecStats

will get several OIDs at the same time and collect them in the same data element.

Here is a list of DCCF's current out-of-the-box oidmap elements:

- cmDsFecStats
- cmDsOfdmChanTable
- cmDsOfdmChannelPowerTable
- cmDsOfdmRxMerTable

III.2 PNM Elements

PNM elements are a set of elements that directly correspond to PNM files outlined in the [CM OSSIv3.1] spec. These elements are:

- cmPNMDsRxMer
- cmPNMHist
- cmPNMUsPreEq

- cmPNMUsPreEqLast
- cmPNMFecSum
- cmPNMSpecAnData
- cmPNMChEstCoef
- cmPNMSpecAnData

III.3 More Exotic Calls

The call

/dccf/ccaps/1.1.1.1/ccapTable/.1.3.6.1.2.1.1.1

is different from the ones above. It asks the XCCF for the ccapTable data element, but this doesn't completely specify what we want from this kind of data element which is a specific SNMP data element. The extra path component, .1.3.6.1.2.1.1.1, is the extra bit of information we need, the numeric OID of the table we want to collect.

There is a similar call for modems:

/dccf/ccaps/1.1.1.1/cms/ab:cd:ef:00:11:22/cmTable/.1.3.6.1.2.1.1.1

This call will tell the XCCF to collect the sysDescr of the cm ab:cd:ef:00:11:22 under the ccap 1.1.1.1 if POSTed, and as a GET it will ask for the latest information.

III.4 Queries and Criteria

Queries currently are supported as JSON bodies in GET requests and as query strings.

III.4.1 Date Range Query

```
{
 "start_date": "<ccf-timestamp>",
 "end_date": "<ccf-timestamp>"
}
```

This query can be combined with other types of queries. It gets all the data the XCCF has between start date and end date.

III.4.2 CM mac Query

You pass a CM mac query to a path that doesn't have a cm mac, like this.

/dccf/ccaps/1.1.1.1/cmPNMDsRxMer

And in the JSON body of the GET:

```
{
  "cm_mac_list": ["<mac-a>", "<mac-b>" ]
}
```

This command will return a tarball consisting of all CMs that match the criteria.

III.5 Namespacing

Data elements are divided into namespaces. This is because certain kinds of equipment, such as optical, are broadly similar to the way cable networks work, but have certain differences. The first element of the path (above we use dccf) specifies the namespace for the XCCF. The other namespace currently supported is occf, which has a different (smaller) set of elements, not documented here.

III.6 Review

The general format of an XCCF call takes several forms:

1. /{namespace}/{root-node-type}/{root-node-ip}/{data-element}
   - if POSTed asks for a data element to be obtained by the XCCF using the appropriate driver.
   - if GETed asks the XCCF to return in the body either the latest element of that type or a tarball of elements that satisfy the query criteria.
2. /{namespace}/{root-node-type}/{root-node-ip}/cms/{cm-mac}/{data-element}
   - if POSTed asks XCCF to collect a cm-specific data element for that specific CM.
   - if GETed asks XCCF to return in the body either the latest element of that type or a tarball of elements that satisfy query criteria.
3. /{namespace}/{root-node-type}/{root-node-ip}/{data-element}/{parameter}
   - Does the same thing as the previous two except it provides an extra parameter to XCCF to completely resolve the data element.
4. /{namespace}/{root-node-type}/{root-node-ip}/cms/{cm-mac}/{data-element}/{parameter}
   - This item is like the third, but for specific modems.

Appendix IV Driver Guide

IV.1 Driver Definition

In the context of XCCF, 'drivers' are the pieces of software that retrieve data of whatever sort from the network elements queried. These software pieces are where the data collection happens. Your driver might query modems using SNMP (which is the most common use case), or it could for example retrieve JSON from some other network appliance. A driver is not, however, in charge of keeping track of its results for longer than however long it takes to get them.

IV.2 Writing a New Driver

While the support software behind the drivers is complicated, the driver architecture simplifies what the user needs to know to use a driver or write one. Drivers in this XCCF architecture work like this:

First, it reads a JSON dictionary (or object) from STDIN, which has in it several key-value pairs that are the parameters for the driver.

It writes on STDOUT a tarball file or archive, which can have a variable number of files. These files will be stored by the XCCF. See IV.4 for how to retrieve these.

So, the (heavily python-influenced) pseudocode for a driver might be like this.

```
parameters = json.parse(stdin.read())

ip_address = parameters['ip_parameter']

other_parameter = parameters['other_parameter']

raw_result = work_function(ip_address, other_parameter)

write_tarball_to_stdout([Result('output', raw_result)])
```

This hypothetical driver would result in the XCCF storing data known by the type 'output' after calling this driver, obtained from ip_address. The XCCF handles where ip_address comes from. There are some utilities in the new_util.py module to help with this task, the function write_files_to_stdout and the class Result.

Here is a hello world driver, that creates results of type 'hello', always containing the text 'Hello, world!'

```
from dccf.drivers.new_util import Result, write_files_to_stdout

if __name__ == '__main__':

   write_files_to_stdout([Result('hello', 'Hello, world!')])
```

This source code would need to be placed under dccf/drivers/new/ in the source tree to be interpreted by CCF as a driver. But other than that, if you insert that code under the name 'hello.py', you'll have a new REST endpoint (discussed in more detail below), and a new data type (hello) that can be collected.

IV.3 Calling a Driver

To 'call' a driver, you POST to XCCF a URL that essentially specifies what kinds of parameters you want to pass to the driver. For example, the curl call

```
curl -X POST <ip-address>:<port>/dccf/ccaps/<ccap-ip>/cms/<cm-mac>/output
```

would call the driver 'output', after collecting parameter values for the ccap and cm. These values are obtained from the JSON files that were collected during initialization.

IV.4 Retrieving the Results of Your Driver

To retrieve the results, you GET from XCCF a result that contains the path you specified to the driver when you called it along with the 'type' of data the driver produced (which is in the filenames). So, for our example driver, we'd have a query like

curl <ip-address>:<port>/dccf/ccaps/<ccap-ip>/cms/<cm-mac>/output

This query would return a tarball. The tarball would contain the most recent file returned by any driver that had the name 'output', and that had been produced for that particular cable modem. Practically speaking, unless you have a good reason not to, name your file the same as your driver. This way, the GET endpoint is the same as the POST endpoint.

IV.5 Common Parameter Values

key	value description
ip	The ip address of the ccap
param	The value of the extra URL parameter if one was specified
cm_ip	The ip address of the modem, if one was specified in the URL path
cmmac	The mac address of the modem if one was specified in the URL path
cm_commstr	The community string of the cable modem (for SNMP)
commstr	The community string of the CMTS (for SNMP)
cm_is31	A Boolean, true or false for if the CM is a DOCSIS 3.1 modem
cm_usev3	A Boolean, does the CM use SNMPv3?
dh_prime	The DH prime value for SNMPv3 if the modem uses SNMPv3
mgr_rand	Random number for SNMPv3 if the modem uses SNMPv3
drivername	The name of the driver, which could be different from the filename in some cases (not documented here)

Also, all the parameter values that DCCF keeps as configuration are passed as well, and values passed in the query string, or as JSON body parameters.

Appendix V ACKNOWLEDGEMENTS

We wish to thank the following participants contributing directly to this document:

Contributor	Company
Jason Schnitzer	Applied Broadband
Robert Cruickshank, Curtis Knittle, John Phillips, Jay Zhu, Jason Rupe	CableLabs