Telemetry Configuration Guide for Cisco NCS 540 Series Routers, IOS XR Release 6.6.x

First Published: 2019-05-30

Americas Headquarters

Cisco Systems, Inc.
170 West Tasman Drive
San Jose, CA 95134-1706
USA
http://www.cisco.com
Tel: 408 526-4000
800 553-NETS (6387)
Fax: 408 527-0883

THE SPECIFICATIONS AND INFORMATION REGARDING THE PRODUCTS IN THIS MANUAL ARE SUBJECT TO CHANGE WITHOUT NOTICE. ALL STATEMENTS, INFORMATION, AND RECOMMENDATIONS IN THIS MANUAL ARE BELIEVED TO BE ACCURATE BUT ARE PRESENTED WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED. USERS MUST TAKE FULL RESPONSIBILITY FOR THEIR APPLICATION OF ANY PRODUCTS.

THE SOFTWARE LICENSE AND LIMITED WARRANTY FOR THE ACCOMPANYING PRODUCT ARE SET FORTH IN THE INFORMATION PACKET THAT SHIPPED WITH THE PRODUCT AND ARE INCORPORATED HEREIN BY THIS REFERENCE. IF YOU ARE UNABLE TO LOCATE THE SOFTWARE LICENSE OR LIMITED WARRANTY, CONTACT YOUR CISCO REPRESENTATIVE FOR A COPY.

NOTWITHSTANDING ANY OTHER WARRANTY HEREIN, ALL DOCUMENT FILES AND SOFTWARE OF THESE SUPPLIERS ARE PROVIDED “AS IS" WITH ALL FAULTS. CISCO AND THE ABOVE-NAMED SUPPLIERS DISCLAIM ALL WARRANTIES, EXPRESSED OR IMPLIED, INCLUDING, WITHOUT LIMITATION, THOSE OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OR ARISING FROM A COURSE OF DEALING, USAGE, OR TRADE PRACTICE.

IN NO EVENT SHALL CISCO OR ITS SUPPLIERS BE LIABLE FOR ANY INDIRECT, SPECIAL, CONSEQUENTIAL, OR INCIDENTAL DAMAGES, INCLUDING, WITHOUT LIMITATION, LOST PROFITS OR LOSS OR DAMAGE TO DATA ARISING OUT OF THE USE OR INABILITY TO USE THIS MANUAL, EVEN IF CISCO OR ITS SUPPLIERS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Any Internet Protocol (IP) addresses and phone numbers used in this document are not intended to be actual addresses and phone numbers. Any examples, command display output, network topology diagrams, and other figures included in the document are shown for illustrative purposes only. Any use of actual IP addresses or phone numbers in illustrative content is unintentional and coincidental.

All printed copies and duplicate soft copies of this document are considered uncontrolled. See the current online version for the latest version.

Cisco has more than 200 offices worldwide. Addresses and phone numbers are listed on the Cisco website at www.cisco.com/go/offices.

The documentation set for this product strives to use bias-free language. For purposes of this documentation set, bias-free is defined as language that does not imply discrimination based on age, disability, gender, racial identity, ethnic identity, sexual orientation, socioeconomic status, and intersectionality. Exceptions may be present in the documentation due to language that is hardcoded in the user interfaces of the product software, language used based on standards documentation, or language that is used by a referenced third-party product.

- To receive timely, relevant information from Cisco, sign up at Cisco Profile Manager.
- To get the business impact you’re looking for with the technologies that matter, visit Cisco Services.
- To submit a service request, visit Cisco Support.
- To discover and browse secure, validated enterprise-class apps, products, solutions and services, visit Cisco Marketplace.
- To obtain general networking, training, and certification titles, visit Cisco Press.
- To find warranty information for a specific product or product family, access Cisco Warranty Finder.

Cisco Bug Search Tool

Cisco Bug Search Tool (BST) is a web-based tool that acts as a gateway to the Cisco bug tracking system that maintains a comprehensive list of defects and vulnerabilities in Cisco products and software. BST provides you with detailed defect information about your products and software.

CONTENTS

CHAPTER 1

Stream Telemetry Data

This document will help you understand the process of streaming telemetry data and its core components.

Scope

Streaming telemetry lets users direct data to a configured receiver. This data can be used for analysis and troubleshooting purposes to maintain the health of the network. This is achieved by leveraging the capabilities of machine-to-machine communication.

The data is used by development and operations (DevOps) personnel who plan to optimize networks by collecting analytics of the network in real-time, locate where problems occur, and investigate issues in a collaborative manner.

Need

Collecting data for analyzing and troubleshooting has always been an important aspect in monitoring the health of a network.

IOS XR provides several mechanisms such as SNMP, CLI and Syslog to collect data from a network. These mechanisms have limitations that restrict automation and scale. One limitation is the use of the pull model, where the initial request for data from network elements originates from the client. The pull model does not scale when there is more than one network management station (NMS) in the network. With this model, the server sends data only when clients request it. To initiate such requests, continual manual intervention is required. This continual manual intervention makes the pull model inefficient.

Network state indicators, network statistics, and critical infrastructure information are exposed to the application layer, where they are used to enhance operational performance and to reduce troubleshooting time. A push model uses this capability to continuously stream data out of the network and notify the client. Telemetry enables the push model, which provides near-real-time access to monitoring data.

Streaming telemetry provides a mechanism to select data of interest from IOS XR routers and to transmit it in a structured format to remote management stations for monitoring. This mechanism enables automatic tuning of the network based on real-time data, which is crucial for its seamless operation. The finer granularity and higher frequency of data available through telemetry enables better performance monitoring and therefore, better troubleshooting. It helps a more service-efficient bandwidth utilization, link utilization, risk assessment and control, remote monitoring and scalability. Streaming telemetry, thus, converts the monitoring process into a Big Data proposition that enables the rapid extraction and analysis of massive data sets to improve decision-making.

Benefits

Streamed real-time telemetry data is useful in:

- **Traffic optimization:** When link utilization and packet drops in a network are monitored frequently, it is easier to add or remove links, re-direct traffic, modify policing, and so on. With technologies like fast reroute, the network can switch to a new path and re-route faster than the SNMP poll interval mechanism. Streaming telemetry data helps in providing quick response time for faster traffic.
- **Preventive troubleshooting:** Helps to quickly detect and avert failure situations that result after a problematic condition exists for a certain duration.

Methods of Telemetry

Telemetry data can be streamed using these methods:

- **Model-driven telemetry:** provides a mechanism to stream data from an MDT-capable device to a destination. The data to be streamed is driven through subscription. There are two methods of configuration:
  - **Cadence-based telemetry:** Cadence-based Telemetry (CDT) continuously streams data (operational statistics and state transitions) at a configured cadence. The streamed data helps users closely identify patterns in the networks. For example, streaming data about interface counters and so on.
  - **Event-based telemetry:** Event-driven Telemetry (EDT) optimizes data collected at the receiver by streaming data only when a state transition occurs. For example, stream data only when an interface state transitions, IP route updates and so on.

  Note EDT is supported only for Interface events, Routing state (RIB events) and Syslog events.

- **Policy-based telemetry:** streams telemetry data to a destination using a policy file. A policy file defines the data to be streamed and the frequency at which the data is to be streamed.

Note Model-driven telemetry supersedes policy-based telemetry.

CHAPTER 2

Configure Model-driven Telemetry

Model-driven Telemetry (MDT) provides a mechanism to stream data from an MDT-capable device to a destination. The data to be streamed is defined through subscription.

The data to be streamed is subscribed from a data set in a YANG model. The data from the subscribed data set is streamed out to the destination either at a configured periodic interval or only when an event occurs. This behavior is based on whether MDT is configured for cadence-based telemetry or event-based telemetry (EDT).

The configuration for event-based telemetry is similar to cadence-based telemetry, with only the sample interval as the differentor. Configuring the sample interval value to zero sets the subscription for event-based telemetry, while configuring the interval to any non-zero value sets the subscription for cadence-based telemetry.

The following YANG models are used to configure and monitor MDT:

- **Cisco-IOS-XR-telemetry-model-driven-cfg.yang** and **openconfig-telemetry.yang:** configure MDT using NETCONF or merge-config over grpc.
- **Cisco-IOS-XR-telemetry-model-driven-oper.yang:** get the operational information about MDT.

For the nodes that support event-driven telemetry (EDT), the YANG model is annotated with the statement `xr:event-telemetry`. For example, the interface that supports EDT has an annotation as shown in the following example:

```
leaf interface-name {
      xr:event-telemetry "Subscribe Telemetry Event";
      type xr:Interface-name;
      description "Member's interface name";
    }
```

The process of streaming MDT data uses these components:

- **Destination:** specifies one or more destinations to collect the streamed data.
- **Sensor path:** specifies the YANG path from which data has to be streamed.
- **Subscription:** binds one or more sensor-paths to destinations, and specifies the criteria to stream data. In cadence-based telemetry, data is streamed continuously at a configured frequency. In event-based telemetry, data is streamed only when a change in the state or data for the configured model occurs.
- **Transport and encoding:** represents the delivery mechanism of telemetry data.

For more information about the core components, see Core Components of Model-driven Telemetry Streaming, on page 15.

The options to initialize a telemetry session between the router and destination is based on two modes:

- Dial-out mode: The router initiates a session to the destinations based on the subscription.
- Dial-in mode: The destination initiates a session to the router and subscribes to data to be streamed.

Note Dial-in mode is supported only over gRPC.

Important From Release 6.1.1 onwards, Cisco introduces support for the 64-bit Linux-based IOS XR operating system. The 64-bit platforms, such as NCS5500, NCS5000, ASR9000 support gRPC, UDP and TCP protocols. All 32-bit IOS XR platforms, such as CRS and legacy ASR9000, support only TCP protocol.

Streaming model-driven telemetry data to the intended receiver involves these tasks:

- Configure Dial-out Mode, on page 4
- Configure Dial-in Mode, on page 9

Configure Dial-out Mode

In a dial-out mode, the router initiates a session to the destinations based on the subscription.

All 64-bit IOS XR platforms (except for NCS 6000 series routers) support gRPC , UDP and TCP protocols. All 32-bit IOS XR platforms support only TCP.

For more information about the dial-out mode, see Dial-out Mode, on page 16.

The process to configure a dial-out mode involves:

Create a Destination Group

The destination group specifies the destination address, port, encoding and transport that the router uses to send out telemetry data.

1. Identify the destination address, port, transport, and encoding format.
2. Create a destination group.

```
Router(config)#telemetry model-driven
Router(config-model-driven)#destination-group <group-name>

Router(config-model-driven-dest)#address family ipv4 <IP-address> port <port-number>
Router(config-model-driven-dest-addr)#encoding <encoding-format>
Router(config-model-driven-dest-addr)#protocol <transport>
Router(config-model-driven-dest-addr)#commit
```

Example: Destination Group for TCP Dial-out

The following example shows a destination group `DGroup1` created for TCP dial-out configuration with key-value Google Protocol Buffers (also called self-describing-gpb) encoding:

```
Router(config)#telemetry model-driven
Router(config-model-driven)#destination-group DGroup1
Router(config-model-driven-dest)#address family ipv4 172.0.0.0 port 5432
Router(config-model-driven-dest-addr)#encoding self-describing-gpb
Router(config-model-driven-dest-addr)#protocol tcp
Router(config-model-driven-dest-addr)#commit
```

Example: Destination Group for UDP Dial-out

The following example shows a destination group `DGroup1` created for UDP dial-out configuration with key-value Google Protocol Buffers (also called self-describing-gpb) encoding:

```
Router(config)#telemetry model-driven
Router(config-model-driven)#destination-group DGroup1
Router(config-model-driven-dest)#address family ipv4 172.0.0.0 port 5432
Router(config-model-driven-dest-addr)#encoding self-describing-gpb
Router(config-model-driven-dest-addr)#protocol udp
Router(config-model-driven-dest-addr)#commit
```

The UDP destination is shown as `Active` irrespective of the state of the collector because UDP is connectionless.

Model-driven Telemetry with UDP is not suitable for a busy network. There is no retry if a message is dropped by the network before it reaches the collector.

Create a Sensor Group

The sensor-group specifies a list of YANG models that are to be streamed.

1. Identify the sensor path for XR YANG model.
2. Create a sensor group.

   ```
   Router(config)#telemetry model-driven
   Router(config-model-driven)#sensor-group <group-name>
   Router(config-model-driven-snsr-grp)# sensor-path <XR YANG model>
   Router(config-model-driven-snsr-grp)# commit
   ```

Example: Sensor Group for Dial-out

Note gRPC is supported in only 64-bit platforms.

The following example shows a sensor group `SGroup1` created for dial-out configuration with the YANG model for interface statistics:

```
Router(config)#telemetry model-driven
Router(config-model-driven)#sensor-group SGroup1
Router(config-model-driven-snsr-grp)# sensor-path
Cisco-IOS-XR-infra-statsd-oper:infra-statistics/interfaces/interface/latest/generic-counters
Router(config-model-driven-snsr-grp)# commit
```

What to Do Next:

Create a subscription.

Create a Subscription

The subscription associates a destination-group with a sensor-group and sets the streaming method - cadence-based or event-based telemetry.

A source interface in the subscription group specifies the interface that will be used for establishing the session to stream data to the destination. If both VRF and source interface are configured, the source interface must be in the same VRF as the one specified under destination group for the session to be established.

```
Router(config)#telemetry model-driven
Router(config-model-driven)#subscription <subscription-name>
Router(config-model-driven-subs)#sensor-group-id <sensor-group> sample-interval <interval>

Router(config-model-driven-subs)#destination-id <destination-group>
Router(config-model-driven-subs)#source-interface <source-interface>
Router(config-mdt-subscription)#commit
```

Example: Subscription for Cadence-based Dial-out Configuration

The following example shows a subscription `Sub1` that is created to associate the sensor-group and destination-group, and configure an interval of 30 seconds to stream data:

```
Router(config)#telemetry model-driven
Router(config-model-driven)#subscription Sub1
Router(config-model-driven-subs)#sensor-group-id SGroup1 sample-interval 30000
Router(config-model-driven-subs)#destination-id DGroup1
Router(config-mdt-subscription)# commit
```

Example: Subscription for Event-based Dial-out Configuration

The following example shows a subscription `Sub1` that is created to associate the sensor-group and destination-group, and configure event-based method to stream data:

```
Router(config)#telemetry model-driven
Router(config-model-driven)#subscription Sub1
Router(config-model-driven-subs)#sensor-group-id SGroup1 sample-interval 0
Router(config-model-driven-subs)#destination-id DGroup1
Router(config-mdt-subscription)# commit
```

Example: Configure Event-driven Telemetry for Interface Path

```
telemetry model-driven
 destination-group 1
  address family ipv4 <ip-address> port <port-number>
   encoding self-describing-gpb
   protocol grpc no-tls
  !
 !
 sensor-group 1
  sensor-path
Cisco-IOS-XR-ipv6-ma-oper:ipv6-network/nodes/node/interface-data/vrfs/vrf/global-briefs/global-brief

 !
 sensor-group 2
```

```
  sensor-path Cisco-IOS-XR-pfi-im-cmd-oper:interfaces/interface-xr/interface
 !
 subscription 1
  sensor-group-id 1 sample-interval 0
  sensor-group-id 2 sample-interval 0
  destination-id 1
!
```

What to Do Next:

Validate the configuration.

Validate Dial-out Configuration

Use the following command to verify that you have correctly configured the router for dial-out.

```
Router#show telemetry model-driven subscription <subscription-group-name>
```

Example: Validation for TCP Dial-out

```
Router#show telemetry model-driven subscription Sub1
Thu Jul 21 15:42:27.751 UTC
Subscription:  Sub1                     State: ACTIVE
-------------
  Sensor groups:
  Id                  Interval(ms)        State
  SGroup1             30000               Resolved

  Destination Groups:
  Id                  Encoding            Transport   State   Port    IP
  DGroup1             self-describing-gpb tcp         Active  5432    172.0.0.0
```

Example: Configure Event-driven Telemetry for LLDP

Telemetry supports NETCONF event notifications where the NETCONF client is configured to receive event notifications from a NETCONF server through a subscription. The NETCONF client must subscribe using a `create-subscription` request. Currently, only the events from Link Layer Discovery Protocol (LLDP) is supported. These event notifications are sent until either the NETCONF session or the subscription is terminated.

Note Configuring a sensor group and a subscription is not required for receiving NETCONF notifications. While sensor path and subscription configurations are required for receiving telemetry events, NETCONF `create-subscription` is required for receiving NETCONF notifications.

To generate NETCONF notifications:

1. Enable NETCONF agent and SSH sub system.

   ```
   ssh server netconf
   netconf-yang agent ssh
   ```

2. Enable model-driven telemetry.

   ```
   telemetry model-driven
   ```

3. Enable LLDP.

   ```
   lldp
   ```

This example shows event-driven telemetry fo LLDP configuration data.

1. Create a destination group.

   ```
   grpc
   port 56782
   address-family ipv4
   !
   telemetry model-driven
   destination-group <destination-udp>
     address-family ipv4 <client-ip>1 port <udp port num>
      encoding self-describing-gpb
      protocol udp
     !
   !
   destination-group <destination-tcp>
     address-family ipv4 <client-ip> port <tcp port num>
      encoding gpb
      protocol tcp
     !
   destination-group <destination-grpc>
     address-family ipv4 <grpc client ip>port <grpc port num>
      encoding self-describing-gpb
      protocol grpc no-tls
   ```

2. Create a sensor group.

   ```
   sensor-group <sensor-group-name>
     sensor-path Cisco-IOS-XR-ethernet-lldp-oper:lldp/global-lldp/lldp-info
     sensor-path Cisco-IOS-XR-ethernet-lldp-oper:lldp/nodes/node/interfaces/interface
     sensor-path Cisco-IOS-XR-ethernet-lldp-oper:lldp/nodes/node/neighbors/details/detail

    !
   ```

3. Create a subscription.

   ```
   subscription udp-out
     sensor-group-id <sensor-group-name> sample-interval 0
     destination-id <destination-udp>
   !

   subscription <subscription-name>
     sensor-group-id <sensor-group-name> sample-interval 0
     destination-id <destination-tcp>

   subscription <subscription-name>
     sensor-group-id <sensor-group-name> sample-interval 0
   !
   netconf-yang agent
   ssh
   !
   ```

4. Set the notification to stream data when an event occurs.

   ```
   Router(config-lldp)#timer 12
   Router(config-lldp)#commit

   Router(config-lldp)#holdtime 150
   ```

```
Router (config-lldp)#commit

Router (config-lldp)#exit
#506
<?xml version="1.0"?>
<notification xmlns="urn:ietf:params:xml:ns:netconf:notification:1.0">
 <eventTime>Date-and-Time</eventTime>
 <lldp xmlns="http://cisco.com/ns/yang/Cisco-IOS-XR-ethernet-lldp-oper">
  <global-lldp>
   <lldp-info>
    <chassis-id>000b.1bc9.e700</chassis-id>
    <chassis-id-sub-type>4</chassis-id-sub-type>
    <system-name>ios</system-name>
    <timer>12</timer>
    <hold-time>120</hold-time>
    <re-init>2</re-init>
   </lldp-info>
  </global-lldp>
 </lldp>
</notification>
Ready to send a request.
Paste your request or enter 'get', 'get-config', 'create-sub', or 'bye' to quit):
```

5. Validate response received from NETCONF agent.

```
#506
<?xml version="1.0"?>
<notification xmlns="urn:ietf:params:xml:ns:netconf:notification:1.0">
 <eventTime>Date-and-Time</eventTime>
 <lldp xmlns="http://cisco.com/ns/yang/Cisco-IOS-XR-ethernet-lldp-oper">
  <global-lldp>
   <lldp-info>
    <chassis-id>000b.1bc9.e700</chassis-id>
    <chassis-id-sub-type>4</chassis-id-sub-type>
    <system-name>ios</system-name>
    <timer>12</timer>
    <hold-time>150</hold-time>
    <re-init>2</re-init>
   </lldp-info>
  </global-lldp>
 </lldp>
</notification>
```

Configure Dial-in Mode

In a dial-in mode, the destination initiates a session to the router and subscribes to data to be streamed.

Note Dial-in mode is supported over gRPC in only 64-bit platforms.

For more information about dial-in mode, see *Dial-in Mode*.

The process to configure a dial-in mode involves these tasks:

- Enable gRPC
- Create a sensor group
- Create a subscription

- Validate the configuration

Enable gRPC

Configure the gRPC server on the router to accept incoming connections from the collector.

1. Enable gRPC over an HTTP/2 connection.

```
Router# configure
Router (config)# grpc
```

2. Enable access to a specified port number.

```
Router (config-grpc)# port <port-number>
```

The <port-number> range is from 57344 to 57999. If a port number is unavailable, an error is displayed.

Note When the gRPC server is configured, support for Google network management interface (gNMI) client to request subscriptions to the router is activated. The user can subscribe to oper paths of interest to stream da

3. In the configuration mode, set the session parameters.

```
Router (config)# grpc{ address-family | dscp | max-request-per-user | max-request-total
 | max-streams | max-streams-per-user | no-tls | service-layer | tls-cipher | tls-mutual
 | tls-trustpoint | vrf }
```

where:

- **address-family:** set the address family identifier type
- **dscp:** set QoS marking DSCP on transmitted gRPC
- **max-request-per-user:** set the maximum concurrent requests per user
- **max-request-total:** set the maximum concurrent requests in total
- **max-streams:** set the maximum number of concurrent gRPC requests. The maximum subscription limit is 128 requests. The default is 32 requests
- **max-streams-per-user:** set the maximum concurrent gRPC requests for each user. The maximum subscription limit is 128 requests. The default is 32 requests
- **no-tls:** disable transport layer security (TLS). The TLS is enabled by default.
- **service-layer:** enable the grpc service layer configuration
- **tls-cipher:** enable the gRPC TLS cipher suites
- **tls-mutual:** set the mutual authentication
- **tls-trustpoint:** configure trustpoint
- **server-vrf:** enable server vrf

4. Configure the TLS ciphers.

```
Router(config-grpc)#tls-cipher default <enable | disable>
Router(config-grpc)#tls-cipher disable <cipher1, cipher2, ...>
Router(config-grpc)#tls-cipher enable <cipher1, cipher2, ...>
```

5. Commit the configuration.

```
Router(config-grpc)#commit
```

Note

- If **`tls-cipher default enable`** and **`tls-cipher disable <all the ciphers>`** are configured, all the ciphers are enabled, not one.
- If **`tls-cipher default disable`** and **`tls-cipher enable <>`** is not configured, all the ciphers are enabled, not one.
- A change in the tls-cipher configuration may cause the gRPC server to stop and restart.

Example: gRPC and TLS enabled on the router

The following example shows the output of `show grpc` command. The sample output displays the gRPC configuration when TLS is enabled on the router.

```
Router#show grpc

Address family              : ipv4
Port                        : 57300
VRF                         : global-vrf
TLS                         : enabled
TLS mutual                  : disabled
Trustpoint                  : none
Maximum requests            : 128
Maximum requests per user   : 10
Maximum streams             : 32
Maximum streams per user    : 32

TLS cipher suites
  Default                   : none
  Enable                    : none
  Disable                   : none

  Operational enable        : ecdhe-rsa-chacha20-poly1305
                            : ecdhe-ecdsa-chacha20-poly1305
                            : ecdhe-rsa-aes128-gcm-sha256
                            : ecdhe-ecdsa-aes128-gcm-sha256
                            : ecdhe-rsa-aes256-gcm-sha384
                            : ecdhe-ecdsa-aes256-gcm-sha384
                            : ecdhe-rsa-aes128-sha
                            : ecdhe-ecdsa-aes128-sha
                            : ecdhe-rsa-aes256-sha
                            : ecdhe-ecdsa-aes256-sha
                            : aes128-gcm-sha256
                            : aes256-gcm-sha384
                            : aes128-sha
                            : aes256-sha
  Operational disable       : none
```

What to Do Next:

Create a sensor group.

Create a Sensor Group

The sensor group specifies a list of YANG models that are to be streamed.

1. Identify the sensor path for XR YANG model.
2. Create a sensor group.

```
Router(config)#telemetry model-driven
Router(config-model-driven)#sensor-group <group-name>
Router(config-model-driven-snsr-grp)# sensor-path <XR YANG model>
Router(config-model-driven-snsr-grp)# commit
```

Example: Sensor Group for gRPC Dial-in

The following example shows a sensor group `SGroup3` created for gRPC dial-in configuration with the YANG model for interfaces:

```
Router(config)#telemetry model-driven
Router(config-model-driven)#sensor-group SGroup3
Router(config-model-driven-snsr-grp)# sensor-path openconfig-interfaces:interfaces/interface

Router(config-model-driven-snsr-grp)# commit
```

What to Do Next:

Create a subscription.

Create a Subscription

The subscription associates a sensor-group with a streaming interval. The collector requests the subscription to the sensor paths when it establishes a connection with the router.

```
Router(config)#telemetry model-driven
Router(config-model-driven)#subscription <subscription-name>
Router(config-model-driven-subs)#sensor-group-id <sensor-group> sample-interval <interval>

Router(config-model-driven-subs)#destination-id <destination-group>
Router(config-mdt-subscription)#commit
```

Example: Subscription for gRPC Dial-in

The following example shows a subscription `Sub3` that is created to associate the sensor-group with an interval of 30 seconds to stream data:

```
Router(config)telemetry model-driven
Router(config-model-driven)#subscription Sub3
Router(config-model-driven-subs)#sensor-group-id SGroup3 sample-interval 30000
Router(config-mdt-subscription)#commit
```

What to Do Next:

Validate the configuration.

Validate Dial-in Configuration

Use the following command to verify that you have correctly configured the router for gRPC dial-in.

```
Router#show telemetry model-driven subscription
```

Example: Validation for gRPC Dial-in

```
RP/0/RP0/CPU0:SunC#show telemetry model-driven subscription Sub3
Thu Jul 21 21:32:45.365 UTC
Subscription:  Sub3
-------------
  State:       ACTIVE
  Sensor groups:
  Id: SGroup3
    Sample Interval:      30000 ms
    Sensor Path:          openconfig-interfaces:interfaces/interface
    Sensor Path State:    Resolved

  Destination Groups:
  Group Id: DialIn_1002
    Destination IP:       172.30.8.4
    Destination Port:     44841
    Encoding:             self-describing-gpb
    Transport:            dialin
    State:                Active
    Total bytes sent:     13909
    Total packets sent:   14
    Last Sent time:       2016-07-21 21:32:25.231964501 +0000

  Collection Groups:
  ------------------
    Id: 2
    Sample Interval:      30000 ms
    Encoding:             self-describing-gpb
    Num of collection:    7
    Collection time:      Min:    32 ms Max:    39 ms
    Total time:           Min:    34 ms Avg:    37 ms Max:    40 ms
    Total Deferred:       0
    Total Send Errors:    0
    Total Send Drops:     0
    Total Other Errors:   0
    Last Collection Start:2016-07-21 21:32:25.231930501 +0000
    Last Collection End:  2016-07-21 21:32:25.231969501 +0000
    Sensor Path:          openconfig-interfaces:interfaces/interface
```

CHAPTER 3

Core Components of Model-driven Telemetry Streaming

The core components used in streaming model-driven telemetry data are:

Session

A telemetry session can be initiated using:

Dial-in Mode

In a dial-in mode, an MDT receiver dials in to the router, and subscribes dynamically to one or more sensor paths or subscriptions. The router acts as the server and the receiver is the client. The router streams telemetry data through the same session. The dial-in mode of subscriptions is dynamic. This dynamic subscription terminates when the receiver cancels the subscription or when the session terminates.

There are two methods to request sensor-paths in a dynamic subscription:

- **OpenConfig RPC model:** The `subscribe RPC` defined in the model is used to specify sensor-paths and frequency. In this method, the subscription is not associated with an existing configured subscription. A subsequent `cancel` RPC defined in the model removes an existing dynamic subscription.

  Cisco IOS XR supports Google network management interface (gNMI) protocol. gNMI is a unified mangement protocol for streaming telemetry data using gRPC framework. This framework and protocol do not need explicit configuration, but simplifies telemetry configuration on the router by starting only the gRPC server.

  The RPC is defined in the gNMI proto file. The gNMI specification is published at Github.

- **IOS XR MDT RPC:** IOS XR defines RPCs to subscribe and to cancel one or more configured subscriptions. The sensor-paths and frequency are part of the telemetry configuration on the router. A subscription is identified by its configured subscription name in the RPCs.

Dial-out Mode

In a dial-out mode, the router dials out to the receiver. This is the default mode of operation. The router acts as a client and receiver acts as a server. In this mode, sensor-paths and destinations are configured and bound together into one or more subscriptions. The router continually attempts to establish a session with each destination in the subscription, and streams data to the receiver. The dial-out mode of subscriptions is persistent. When a session terminates, the router continually attempts to re-establish a new session with the receiver every 30 seconds.

Sensor Path

The sensor path describes a YANG path or a subset of data definitions in a YANG model with a container. In a YANG model, the sensor path can be specified to end at any level in the container hierarchy.

An MDT-capable device, such as a router, associates the sensor path to the nearest container path in the model. The router encodes and streams the container path within a single telemetry message. A receiver receives data about all the containers and leaf nodes at and below this container path.

The router streams telemetry data for one or more sensor-paths, at the configured frequency (cadence-based streaming) or when the sensor-path content changes (event-based streaming), to one or more receivers through subscribed sessions.

Subscription

A subscription binds one or more sensor paths and destinations. An MDT-capable device streams data for each sensor path at the configured frequency (cadence-based streaming) or when the sensor-path content changes (event-based streaming) to the destination.

The following example shows subscription `SUB1` that associates a sensor-group, sample interval and destination group.

```
Router(config)#telemetry model-driven
Router(config-model-driven)#subscription SUB1
Router(config-model-driven-subs)#sensor-group-id SGROUP1 sample-interval 10000
Router(config-model-driven-subs)#strict-timer
```

Note With a `strict-timer` configured for the sample interval, the data collection starts exactly at the configured time interval allowing a more deterministic behavior to stream data.

In 32-bit platforms, `strict-timer` can be configured only under the subscription. Whereas, 64-bit platforms support configurtion at global level in addition to the subscription level. However, configuring at the global level will affect all configured subscriptions.

```
Router(config)#telemetry model-driven
Router(config-model-driven)#strict-timer
```

Transport and Encoding

The router streams telemetry data using a transport mechanism. The generated data is encapsulated into the desired format using encoders.

Model-Driven Telemetry (MDT) data is streamed through :

- **Transmission Control Protocol (TCP):** used for only dial-out mode.
- **User Datagram Protocol (UDP):** used for only dial-out mode.

The data to be streamed can be encoded into Google Protocol Buffers (GPB) encoding. In GPB, the encoding can either be compact GPB (for optimising the network bandwidth usage) or self-describing GPB. The encodings supported are:

- **GPB encoding:** configuring for GPB encoding requires metadata in the form of compiled .proto files. A .proto file describes the GPB message format, which is used to stream data. The .proto files are available in the Github repository.
  - **Compact GPB encoding:** data is streamed in compressed and non self-describing format. A .proto file corresponding to each sensor-path must be used by the receiver to decode the streamed data.
  - **Key-value (KV-GPB) encoding:** data of each sensor path streamed is in a self-describing formatted ASCII text. A single .proto file `telemetry.proto` is used by the receiver to decode any sensor path data. Because the key names are included in the streamed data, the data on the wire is much larger as compared to compact GPB encoding.

Note Telemetry data is streamed out of the router using an Extensible Manageability Services Deamon (emsd) process. The data of interest is subscribed through subscriptions and streamed through gRPC, TCP or UDP sessions. However, a combination of gRPC, TCP and UDP sessions with more than 150 active sessions leads to emsd crash or process restart.

CHAPTER 4

Example: Configure Event-driven Telemetry for LLDP

Telemetry supports NETCONF event notifications where the NETCONF client is configured to receive event notifications from a NETCONF server through a subscription. The NETCONF client must subscribe using a `create-subscription` request. Currently, only the events from Link Layer Discovery Protocol (LLDP) is supported. These event notifications are sent until either the NETCONF session or the subscription is terminated.

Note Configuring a sensor group and a subscription is not required for receiving NETCONF notifications. While sensor path and subscription configurations are required for receiving telemetry events, NETCONF `create-subscription` is required for receiving NETCONF notifications.

To generate NETCONF notifications:

1. Enable NETCONF agent and SSH sub system.

```
ssh server netconf
netconf-yang agent ssh
```

2. Enable model-driven telemetry.

```
telemetry model-driven
```

3. Enable LLDP.

```
lldp
```

This example shows event-driven telemetry fo LLDP configuration data.

1. Create a destination group.

```
grpc
port 56782
address-family ipv4
!
telemetry model-driven
destination-group <destination-udp>
  address-family ipv4 <client-ip>1 port <udp port num>
   encoding self-describing-gpb
   protocol udp
  !
!
destination-group <destination-tcp>
  address-family ipv4 <client-ip> port <tcp port num>
   encoding gpb
   protocol tcp
  !
destination-group <destination-grpc>
  address-family ipv4 <grpc client ip>port <grpc port num>
   encoding self-describing-gpb
   protocol grpc no-tls
```

2. Create a sensor group.

```
sensor-group <sensor-group-name>
  sensor-path Cisco-IOS-XR-ethernet-lldp-oper:lldp/global-lldp/lldp-info
  sensor-path Cisco-IOS-XR-ethernet-lldp-oper:lldp/nodes/node/interfaces/interface
  sensor-path Cisco-IOS-XR-ethernet-lldp-oper:lldp/nodes/node/neighbors/details/detail

 !
```

3. Create a subscription.

```
subscription udp-out
  sensor-group-id <sensor-group-name> sample-interval 0
  destination-id <destination-udp>
!

subscription <subscription-name>
  sensor-group-id <sensor-group-name> sample-interval 0
  destination-id <destination-tcp>

subscription <subscription-name>
  sensor-group-id <sensor-group-name> sample-interval 0
!
netconf-yang agent
ssh
!
```

4. Set the notification to stream data when an event occurs.

```
Router(config-lldp)#timer 12
Router(config-lldp)#commit

Router(config-lldp)#holdtime 150
Router (config-lldp)#commit

Router (config-lldp)#exit
#506
<?xml version="1.0"?>
<notification xmlns="urn:ietf:params:xml:ns:netconf:notification:1.0">
 <eventTime>Date-and-Time</eventTime>
 <lldp xmlns="http://cisco.com/ns/yang/Cisco-IOS-XR-ethernet-lldp-oper">
  <global-lldp>
   <lldp-info>
    <chassis-id>000b.1bc9.e700</chassis-id>
    <chassis-id-sub-type>4</chassis-id-sub-type>
    <system-name>ios</system-name>
    <timer>12</timer>
    <hold-time>120</hold-time>
    <re-init>2</re-init>
   </lldp-info>
  </global-lldp>
 </lldp>
</notification>
Ready to send a request.
Paste your request or enter 'get', 'get-config', 'create-sub', or 'bye' to quit):
```

5. Validate response received from NETCONF agent.

```
#506
<?xml version="1.0"?>
<notification xmlns="urn:ietf:params:xml:ns:netconf:notification:1.0">
 <eventTime>Date-and-Time</eventTime>
 <lldp xmlns="http://cisco.com/ns/yang/Cisco-IOS-XR-ethernet-lldp-oper">
  <global-lldp>
   <lldp-info>
    <chassis-id>000b.1bc9.e700</chassis-id>
    <chassis-id-sub-type>4</chassis-id-sub-type>
    <system-name>ios</system-name>
    <timer>12</timer>
    <hold-time>150</hold-time>
    <re-init>2</re-init>
   </lldp-info>
  </global-lldp>
 </lldp>
</notification>
```

- Example: Create and Delete LACP Bundle or Member, on page 21

Example: Create and Delete LACP Bundle or Member

OpenConfig-Link Aggregation Control Protocol (OC-LACP) data model can be used to stream model-driven telemetry (MDT) and event-driven telemetry (EDT) data. Link Aggregation Control Protocol (LACP) controls the bundling of one or more ports together to form a single logical link. This single link provides higher bidirectional bandwidth, redundancy, and load balancing between the routers in the network.

The OC-LACP model defined by the OC community, helps manage LACP-enabled bundles and member interfaces. Cisco IOS XR software supports OC-LACP version 1.0.2. The support is extended to version 1.1.0 from Cisco IOS XR software release 6.6.1.

Note Streaming model-driven or event-driven telemetry data is supported only on operational state parameters.

The support for telemetry is provided only for LACP state data at global, bundle and member level.

Using this model, user can configure LACP parameters on the bundle interface and view state data for LACP-enabled bundle and member interfaces.

The OC LACP yang model is available in the Github repository.

The following logs shows an example of expected telemetry data when a LACP member is added to bundle, and then deleted from the bundle.

Add a LACP Member to Bundle

1. Add an ethernet link bundle with the specified bundle-id.

```
Router#config
Router(config)#interface bundle-ether 4
```

2. Specify the interface details.

```
Router(config)#interface gigabitEthernet 0/2/0/3
```

3. Add the link to the specified bundle. To enable active or passive LACP on the bundle, include the optional `mode active` or `mode passive` keywords in the command string.

```
Router(config-if)#bundle id 4 mode passive
Router(config-if)#no shutdown
```

4. Check the log.

```
Sub_id 200000001, flag 4, len 830
--------
{"node_id_str":"ios","subscription_id_str":"app_TEST_200000001","encoding_path":"openconfig-lacp:lacp",
"collection_id":"5","collection_start_time":"1542876320548","msg_timestamp":"1542876320548","
data_json":[{"timestamp":"1542876320548","keys":[],"content":{"interfaces":{"interface":{"name":"Bundle-Ether4",
"members":{"member":{"interface":"GigabitEthernet0/2/0/3","state":{"interface":"GigabitEthernet0/2/0/3",
"activity":"PASSIVE","timeout":"LONG","synchronization":"OUT_SYNC","aggregatable":true,"collecting":false,
"distributing":false,"system-id":"02-9d-af-84-41-05","oper-key":4,"partner-id":"00-00-00-00-00-00","partner-key":0,
"port-num":0,"partner-port-num":0,"counters":{"lacp-in-pkts":"0","lacp-out-pkts":"0","lacp-rx-errors":"0",
"lacp-unknown-errors":"0","lacp-errors":"0"}}}}}}}}}],"collection_end_time":"1542876320548"}
--------
Sub_id 200000001, flag 4, len 830
--------
{"node_id_str":"ios","subscription_id_str":"app_TEST_200000001","encoding_path":"openconfig-lacp:lacp","collection_id":
"6","collection_start_time":"1542876320569","msg_timestamp":"1542876320569","data_json":[{"timestamp":"1542876320568",
"keys":[],"content":{"interfaces":{"interface":{"name":"Bundle-Ether4","members":{"member":{"interface":
"GigabitEthernet0/2/0/3","state":{"interface":"GigabitEthernet0/2/0/3","activity":"PASSIVE","timeout":"LONG",
"synchronization":"OUT_SYNC","aggregatable":true,"collecting":false,"distributing":false,"system-id":
"02-9d-af-84-41-05","oper-key":4,"partner-id":"00-00-00-00-00-00","partner-key":0,"port-num":4,"partner-port-num":0,
"counters":{"lacp-in-pkts":"0","lacp-out-pkts":"0","lacp-rx-errors":"0","lacp-unknown-errors":"0","lacp-errors":"0"}}}}}}}}}],
"collection_end_time":"1542876320569"}
--------
Sub_id 200000001, flag 4, len 466
--------
{"node_id_str":"ios","subscription_id_str":"app_TEST_200000001","encoding_path":"openconfig-lacp:lacp","collection_id":
"7","collection_start_time":"1542876320570","msg_timestamp":"1542876320570","data_json":[{"timestamp":"1542876320568",
"keys":[],"content":{"interfaces":{"interface":{"name":"Bundle-Ether4","state":{"name":"Bundle-Ether4","interval":"SLOW","lacp-mode":
"PASSIVE","system-id-mac":"02-9d-af-84-41-05","system-priority":32768}}}}}],"collection_end_time":"1542876320570"}
```

Delete a LACP Member from Bundle

1. Delete a LACP member from the bundle, a scenario where the bundle also becomes non-LACP.

```
Router(config)#interface gigabitEthernet 0/2/0/3
Router(config-if)#bundle id 4 mode on
Router(config-if)#commit
```

2. Check the log.

```
--------
Sub_id 200000001, flag 4, len 425
--------
{"node_id_str":"ios","subscription_id_str":"app_TEST_200000001","encoding_path":"openconfig-lacp:lacp",
"collection_id":"8","collection_start_time":"1542876408256","msg_timestamp":"1542876408256","data_json":
[{"timestamp":"1542876408256","delete":true,"keys":[],"content":{"interfaces":{"interface":{"name":
"Bundle-Ether4","members":{"member":{"interface":"GigabitEthernet0/2/0/3"}}}}}}],"collection_end_time":
"1542876408256"}
--------
Sub_id 200000001, flag 4, len 365
--------
```

```
{"node_id_str":"ios","subscription_id_str":"app_TEST_200000001","encoding_path":"openconfig-lacp:lacp",
"collection_id":"9","collection_start_time":"1542876408256","msg_timestamp":"1542876408256","data_json":
[{"timestamp":"1542876408256","delete":true,"keys":[],"content":{"interfaces":{"interface":{"name":
"Bundle-Ether4"}}}}],"collection_end_time":"1542876408256"}
```

CHAPTER 5

Configure Policy-based Telemetry

Policy-based telemetry (PBT) streams telemetry data to a destination using a policy file. A policy file defines the data to be streamed and the frequency at which the data is to be streamed.

The process of streaming telemetry data uses three core components:

- **Telemetry Policy File** specifies the kind of telemetry data to be generated, at a specified frequency.
- **Telemetry Encoder** encapsulates the generated data into the desired format and transmits to the receiver.
- **Telemetry Receiver** is the remote management system that stores the telemetry data.

For more information about the three core components, see Core Components of Policy-based Telemetry Streaming, on page 31.

Note Model-driven telemetry supersedes policy-based telemetry.

Streaming policy-based telemetry data to the intended receiver involves these tasks:

- Create Policy File, on page 25
- Copy Policy File, on page 27
- Configure Encoder, on page 27
- Verify Policy Activation, on page 29

Create Policy File

You define a telemetry policy file to specify the kind of telemetry data to be generated and pushed to the receiver. Defining the policy files requires a path to stream data. The paths can be schemas, native YANG or allowed list entries.

For more information on the schema paths associated with a corresponding CLI command, see Schema Paths, on page 32.

For more information on policy files, see Telemetry Policy File, on page 31.

1. Determine the schema paths to stream data.

   For example, the schema path for interfaces is:

```
RootOper.InfraStatistics.Interface(*).Latest.GenericCounters
```

2. Create a policy file that contains these paths.

Example: Policy File

The following example shows a sample policy file for streaming the generic counters of an interface:

```
{
"Name": "Test",
 "Metadata": {
     "Version": 25,
     "Description": "This is a sample policy",
     "Comment": "This is the first draft",
     "Identifier": "<data that may be sent by the encoder to the mgmt stn"
},
 "CollectionGroups": {
     "FirstGroup": {
         "Period": 10,
         "Paths": [
             "RootOper.InfraStatistics.Interface(*).Latest.GenericCounters"
             ]
     }
 }
}
```

The following example shows the paths with allowed list entries in the policy file. Instead of streaming all the data for a particular entry, only specific items can be streamed using allowed list entries. The entries are allowed using `IncludeFields` in the policy file. In the example, the entry within the IncludeFields section streams only the latest applied AutoBW value for that TE tunnel, which is nested two levels down from the top level of the path:

```
{

        "Name": "RSVPTEPolicy",
        "Metadata": {

                "Version": 1,
                "Description": "This policy collects auto bw stats",
                "Comment": "This is the first draft"
        },

        "CollectionGroups": {
                "FirstGroup": {
                   "Period": 10,
                        "Paths": {
                            "RootOper.MPLS_TE.P2P_P2MPTunnel.TunnelHead({'TunnelName':
'tunnel-te10'})": {
                                "IncludeFields": [{

                                     "P2PInfo": [{
                                         "AutoBandwidthOper": [
                                               "LastBandwidthApplied"
                                               ]
                                         }]
                                    }]
                                }
                        }
                }
        }
}
```

The following example shows the paths with native YANG entry in the policy file. This entry will stream the generic counters of the interface:

```
"Paths": [

“/Cisco-IOS-XR-infra-statsd-oper:infra-statistics/interfaces/interface=*/latest/generic-counters”

 ]
```

What to Do Next:

Copy the policy file to the router. You may copy the same policy file to multiple routers.

Copy Policy File

Run the Secure Copy Protocol (SCP) command to securely copy the policy file from the server where it is created. For example:

```
$ scp Test.policy <ip-address-of-router>:/telemetry/policies
```

For example, to copy the `Test.policy` file to the `/telemetry/policies` folder of a router with IP address 10.0.0.1:

```
$ scp Test.policy cisco@10.0.0.1:/telemetry/policies
cisco@10.0.0.1's password:
Test.policy
100%  779     0.8KB/s   00:00
Connection to 10.0.0.1 closed by remote host.
```

Verify Policy Installation

In this example, the policy is installed in the `/telemetry/policies/` folder in the router file system. Run the **show telemetry policies brief** command to verify that the policy is successfully copied to the router.

```
Router#show telemetry policy-driven policies brief
Wed Aug 26 02:24:40.556 PDT

 Name                       |Active?| Version | Description
----------------------------|-------|---------|------------------------------
 Test                         N        1        This is a sample policy
```

What to Do Next:

Configure the telemetry encoder to activate and stream data.

Configure Encoder

An encoder calls the streaming Telemetry API to:

- Specify policies to be explicitly defined
- Register all policies of interest

Configure the encoder to activate the policy and stream data. More than one policy and destination can be specified. Multiple policy groups can be specified under each encoder and each group can be streamed to multiple destinations. When multiple destinations are specified, the data is streamed to all destinations.

Configure an encoder based on the requirement.

Configure JSON Encoder

The JavaScript Object Notation (JSON) encoder is packaged with the IOS XR software and provides the default format for streaming telemetry data.

To stream data in JavaScript Object Notation (JSON) format, specify the encoder, policies, policy group, destination, and port:

```
Router# configure
Router(config)#telemetry policy-driven encoder json
Router(config-telemetry-json)#policy group FirstGroup
Router(config-policy-group)#policy Test
Router(config-policy-group)#destination ipv4 10.0.0.1 port 5555
Router(config-policy-group)#commit
```

The names of the policy and the policy group must be identical to the policy and its definition that you create. For more information on policy files, see Create Policy File, on page 25.

For more information about the message format of JSON encoder, see JSON Message Format, on page 35

Configure GPB Encoder

Configuring the GPB (Google Protocol Buffer) encoder requires metadata in the form of compiled `.proto` files. A `.proto` file describes the GPB message format, which is used to stream data.

Two encoding formats are supported:

- **Compact encoding** stores data in a compressed and non-self-describing format. A `.proto` file must be generated for each path in the policy file to be used by the receiver to decode the resulting data.
- **Key-value encoding** uses a single `.proto` file to encode data in a self-describing format. This encoding does not require a `.proto` file for each path. The data on the wire is much larger because key names are included.

To stream GPB data, complete these steps:

1. For compact encoding, create `.proto` files for all paths that are to be streamed using the following tool:

   ```
   telemetry generate gpb-encoding path <path> [file <output_file>]
   ```

   or

   ```
   telemetry generate gpb-encoding policy <policy_file> directory <output_dir>
   ```

   Attention A parser limitation does not support the use of quotes within paths in the tool. For example, for use in the tool, change this policy path, `RootOper.InfraStatistics.Interface(*).Latest.Protocol(['IPV4_UNICAST'])` to `RootOper.InfraStatistics.Interface(*).Latest.Protocol.`

2. Copy the policy file to the router.
3. Configure the telemetry policy specifying the encoder, policies, policy group, destination, and port:

```
Router# configure
Router(config)#telemetry policy-driven encoder gpb
Router(config-telemetry-json)#policy group FirstGroup
Router(config-policy-group)#policy Test
Router(config-policy-group)#destination ipv4 10.0.0.1 port 5555
Router(config-policy-group)#commit
```

For more information about the message format of GPB encoder, see GPB Message Format, on page 37

Verify Policy Activation

Verify that the policy is activated using the `show telemetry policies` command.

```
Router#show telemetry policy-driven policies
Wed Aug 26 02:24:40.556 PDT

  Filename:            Test.policy
  Version:              25
  Description:          This is a sample policy to demonstrate the syntax
  Status:               Active
  CollectionGroup: FirstGroup
    Cadence:              10s
    Total collections:    2766
    Latest collection:    2015-08-26 02:25:07
    Min collection time:  0.000s
    Max collection time:  0.095s
    Avg collection time:  0.000s
    Min total time:       0.022s
    Max total time:       0.903s
    Avg total time:       0.161s
    Collection errors:    0
    Missed collections:   0

+--------------------------------------+-------+-------+-----+-----+-----+-----+-----+-----+

    | Path                                                                          | Avg (s)
 | Max (s) | Err  |

+--------------------------------------+-------+-------+-----+-----+-----+-----+-----+-----+

    | RootOper.InfraStatistics.Interface(*).Latest.GenericCounters            |   0.000 |
   0.000 |    0 |

+--------------------------------------+-------+-------+-----+-----+-----+-----+-----+-----+
```

After the policy is validated, the telemetry encoder starts streaming data to the receiver. For more information on the receiver, see Telemetry Receiver, on page 40.

CHAPTER 6

Core Components of Policy-based Telemetry Streaming

The core components used in streaming policy-based telemetry data are:

Telemetry Policy File

A telemetry policy file is defined by the user to specify the kind of telemetry data that is generated and pushed to the receiver. The policy must be stored in a text file with a .policy extension. Multiple policy files can be defined and installed in the `/telemetry/policies/` folder in the router file system.

A policy file:

- Contains one or more collection groups; a collection group includes different types of data to be streamed at different intervals
- Includes a period in seconds for each group
- Contains one or more paths for each group
- Includes metadata that contains version, description, and other details about the policy

Policy file syntax

The following example shows a sample policy file:

```
{
"Name": "NameOfPolicy",
 "Metadata": {
     "Version": 25,
     "Description": "This is a sample policy to demonstrate the syntax",
     "Comment": "This is the first draft",
     "Identifier": "<data that may be sent by the encoder to the mgmt stn"
},
 "CollectionGroups": {
     "FirstGroup": {
         "Period": 10,
         "Paths": [
             "RootOper.MemorySummary.Node",
             "RootOper.RIB.VRF",
             "..."
         ]
     },
     "SecondGroup": {
         "Period": 300,
         "Paths": [
             "RootOper.Interfaces.Interface"
         ]
     }
 }
}
```

The syntax of the policy file includes:

- **Name** the name of the policy. In the previous example, the policy is stored in a file named `NameOfPolicy.policy`. The name of the policy must match the filename (without the .policy extension). It can contain uppercase alphabets, lower-case alphabets, and numbers. The policy name is case sensitive.
- **Metadata** information about the policy. The metadata can include the version number, date, description, author, copyright information, and other details that identify the policy. The following fields have significance in identifying the policy:
    - Description is displayed in the **show policies** command.
    - Version and Identifier are sent to the receiver as part of the message header of the telemetry messages.
- **CollectionGroups** an encoder object that maps the group names to information about them. The name of the collection group can contain uppercase alphabets, lowercase alphabets, and numbers. The group name is case sensitive.
- **Period** the cadence for each collection group. The period specifies the frequency in seconds at which data is queried and sent to the receiver. The value must be within the range of 5 and 86400 seconds.
- **Paths** one or more schema paths, allowed list entries or native YANG paths (for a container) for the data to be streamed and sent to the receiver. For example,

  Schema path:

  ```
  RootOper.InfraStatistics.Interface(*).Latest.GenericCounters
  ```

  YANG path:

  ```
  /Cisco-IOS-XR-infra-statsd-oper:infra-statistics/interfaces/interface=*/latest/generic-counters
  ```

  Allowed list entry:

  ```
  "RootOper.Interfaces.Interface(*)":
  {
                 "IncludeFields": ["State"]
  }
  ```

Schema Paths

A schema path is used to specify where the telemetry data is collected. A few paths are listed in the following table for your reference:

Table 1: Schema Paths

Operation	Path
Interface Operational data	RootOper.Interfaces.Interface(*)
Packet/byte counters	RootOper.InfraStatistics.Interface(*).Latest.GenericCounters
Packet/byte rates	RootOper.InfraStatistics.Interface(*).Latest.DataRate
IPv4 packet/byte counters	RootOper.InfraStatistics.Interface(*).Latest.Protocol(['IPV4_UNICAST'])
MPLS stats	• RootOper.MPLS_TE.Tunnels.TunnelAutoBandwidth • RootOper.MPLS_TE.P2P_P2MPTunnel.TunnelHead • RootOper.MPLS_TE.SignallingCounters.HeadSignallingCounters
QOS Stats	• RootOper.QOS.Interface(*).Input.Statistics • RootOper.QOS.Interface(*).Output.Statistics
BGP Data	RootOper.BGP.Instance({'InstanceName': 'default'}).InstanceActive.DefaultVRF.Neighbor([*])
Inventory data	RootOper.PlatformInventory.Rack(*).Attributes.BasicInfo RootOper.PlatformInventory.Rack(*).Slot(*).Card(*).Sensor(*).Attributes.BasicInfo

Telemetry Encoder

The telemetry encoder encapsulates the generated data into the desired format and transmits to the receiver.

An encoder calls the streaming Telemetry API to:

- Specify policies to be explicitly defined
- Register all policies of interest

Telemetry supports two types of encoders:

- **JavaScript Object Notation (JSON) encoder**

  This encoder is packaged with the IOS XR software and provides the default method of streaming telemetry data. It can be configured by CLI and XML to register for specific policies. Configuration is grouped into policy groups, with each policy group containing one or more policies and one or more destinations. JSON encoding is supported over only TCP transport service.

  JSON encoder supports two encoding formats:

  - **Restconf-style encoding** is the default JSON encoding format.
  - **Embedded-keys encoding** treats naming information in the path as keys.

- **Google Protocol Buffers (GPB) encoder**

  This encoder provides an alternative encoding mechanism, streaming the data in GPB format over UDP or TCP. It can be configured by CLI and XML and uses the same policy files as those of JSON.

Additionally, a GPB encoder requires metadata in the form of compiled .proto files to translate the data into GPB format.

GPB encoder supports two encoding formats:

- **Compact encoding** stores data in a compact GPB structure that is specific to the policy that is streamed. This format is available over both UDP and TCP transport services. A `.proto` file must be generated for each path in the policy file to be used by the receiver to decode the resulting data.
- **Key-value encoding** stores data in a generic key-value format using a single `.proto` file. The encoding is self-describing as the keys are contained in the message. This format is available over UDP and TCP transport service. A `.proto` file is not required for each policy file because the receiver can interpret the data.

TCP Header

Streaming data over a TCP connection either with a JSON or a GPB encoder and having it optionally compressed by zlib ensures that the stream is flushed at the end of each batch of data. This helps the receiver to decompress the data received. If data is compressed using zlib, the compression is done at the policy group level. The compressor resets when a new connection is established from the receiver because the decompressor at the receiver has an empty initial state.

Header of each TCP message:

Type	Flags	Length	Message
4 bytes	4 bytes • **default** - Use 0x0 value to set no flags. • **zlib compression** - Use 0x1 value to set zlib compression on the message.	4 bytes	Variable

where:

- The Type is encoded as a big-endian value.
- The Length (in bytes) is encoded as a big-endian value.
- The flags indicates modifiers (such as compression) in big-endian format.
- The message contains the streamed data in either JSON or GPB object.

Type of messages:

Type	Name	Length	Value
1	Reset Compressor	0	No value
2	JSON Message	Variable	JSON message (any format)
3	GPB compact	Variable	GPB message in compact format

Type	Name	Length	Value
4	GPB key-value	Variable	GPB message in key-value format

JSON Message Format

JSON messages are sent over TCP and use the header message described in TCP Header, on page 34.

The message consists of the following JSON objects:

```
{
"Policy": "<name-of-policy>",
"Version": "<policy-version>",
"Identifier": "<data from policy file>"
"CollectionID": <id>,
"Path": <Policy Path>,
"CollectionStartTime": <timestamp>,
"Data": { … object as above … },
"CollectionEndTime": <timestamp>,
}
```

where:

- `Policy`, `Version` and `Identifier` are specified in the policy file.
- `CollectionID` is an integer that allows messages to be grouped together if data for a single path is split over multiple messages.
- `Path` is the base path of the corresponding data as specified in the policy file.
- `CollectionStartTime` and `CollectionEndTime` are the timestamps that indicate when the data was collected

The JSON message reflects the hierarchy of the router's data model. The hierarchy consists of:

- containers: a container has nodes that can be of different types.
- tables: a table also contains nodes, but the number of child nodes may vary, and they must be of the same type.
- leaf node: a leaf contains a data value, such as integer or string.

The schema objects are mapped to JSON are in this manner:

- Each container maps to a JSON object. The keys are strings that represent the schema names of the nodes; the values represent the values of the nodes.
- JSON objects are also used to represent tables. In this case, the keys are based on naming information that is converted to string format. Two options are provided for encoding the naming information:
    - The default is restconf-style encoding, where naming parameters are contained within the child node to which it refers.
    - The embedded-keys option uses the naming information as keys in a JSON dictionary, with the corresponding child node forming the value.
- Leaf data types are mapped in this manner:

- Simple strings, integers, and booleans are mapped directly.
- Enumeration values are stored as the string representation of the value.
- Other simple data types, such as IP addresses, are mapped as strings.

Example: Rest-conf Encoding

For example, consider the path -

```
Interfaces(*).Counters.Protocols(“IPv4”)
```

This has two naming parameters - the interface name and the protocol name - and represents a container holding leaf nodes which are packet and byte counters. This would be represented as follows:

```
{
  "Interfaces": [
    {
      "Name": "GigabitEthernet0/0/0/1"
      "Counters": {
        "Protocols": [
          {
            "ProtoName": "IPv4",
            "CollectionTime": 12345678,
            "InputPkts": 100,
            "InputBytes": 200,
          }
        ]
      }
    },{
      "Name": "GigabitEthernet0/0/0/2"
      "Counters": {
        "Protocols": [
          {
            "ProtoName": "IPv4",
            "CollectionTime": 12345678,
            "InputPkts": 400,
            "InputBytes": 500,
          }
        ]
      }
    }
  ]
}
```

A naming parameter with multiple keys, for example Foo.Destination(IPAddress=10.1.1.1, Port=2000) would be represented as follows:

```
{
  "Foo":
    {
    "Destination": [
      {
        "IPAddress": 10.1.1.1,
        "Port": 2000,
        "CollectionTime": 12345678,
        "Leaf1": 100,
      }
    ]
  }
}
```

Example: Embedded Keys Encoding

The embedded-keys encoding treats naming information in the path as keys in the JSON dictionary. The key name information is lost and there are extra levels in the hierarchy but it is clearer which data constitutes the key which may aid collectors when parsing it. This option is provided primarily for backwards-compatibility with 6.0.

```
{
  "Interfaces": {
      "GigabitEthernet0/0/0/1": {
        "Counters": {
          "Protocols": {
            "IPv4": {
              "CollectionTime": 12345678,
              "InputPkts": 100,
              "InputBytes": 200,
            }
          }
        }
      },
      "GigabitEthernet0/0/0/2": {
        "Counters": {
          "Protocols": {
            "IPv4": {
              "CollectionTime": 12345678,
              "InputPkts": 400,
              "InputBytes": 500,
            }
          }
        }
      }
    }
  }
}
```

A naming parameter with multiple keys, for example Foo.Destination(IPAddress=10.1.1.1, Port=2000), would be represented by nesting each key in order:

```
{
  "Foo":
    {
    "Destination": {
        10.1.1.1: {
          2000: {
            Leaf1": 100,
          }
        }
      }
    ]
  }
}
```

GPB Message Format

The output of the GPB encoder consists entirely of GPBs and allows multiple tables in a single packet for scalability.

GPB (Google Protocol Buffer) encoder requires metadata in the form of compiled `.proto` files. A `.proto` file describes the GPB message format, which is used to stream data.

For UDP, the data is simply a GPB. Only the compact format is supported so the message can be interpreted as a `TelemetryHeader` message.

For TCP, the message body is either a `Telemetry` message or a `TelemetryHeader` message, depending on which of the following encoding types is configured:

- **Compact GPB format** stores data in a compressed and non-self-describing format. A `.proto` file must be generated for each path in the policy file to be used by the receiver to decode the resulting data.
- **Key-value GPB format** uses a single `.proto` file to encode data in a self-describing format. This encoding does not require a `.proto` file for each path. The data on the wire is much larger because key names are included.

In the following example, the policy group, *alpha* uses the default configuration of compact encoding and UDP transport. The policy group, *beta* uses compressed TCP and key-value encoding. The policy group, *gamma* uses compact encoding over uncompressed TCP.

```
telemetry policy-driven encoder gpb
  policy group alpha
    policy foo
    destination ipv4 192.168.1.1 port 1234
    destination ipv4 10.0.0.1 port 9876
  policy group beta
    policy bar
    policy whizz
    destination ipv4 10.20.30.40 port 3333
    transport tcp
      compression zlib
  policy group gamma
    policy bang
    destination ipv4 10.11.1.1 port 4444
    transport tcp
      encoding-format gpb-compact
```

Compact GPB Format

The compact GPB format is intended for streaming large volumes of data at frequent intervals. The format minimizes the size of the message on the wire. Multiple tables can be sent in in a single packet for scalability.

Note The tables can be split over multiple packets but fragmenting a row is not supported. If a row in the table is too large to fit in a single UDP frame, it cannot be streamed. Instead either switch to TCP, increase the MTU, or modify the .proto file.

The following .proto file shows the header, which is common to all packets sent by the encoder:

```
message TelemetryHeader {
  optional uint32 encoding = 1

  optional string policy_name = 2;
  optional string version = 3;
  optional string identifier = 4;

  optional uint64 start_time = 5;
  optional uint64 end_time = 6;

  repeated TelemetryTable tables = 7;
  }

message TelemetryTable {
  optional string policy_path = 1;
  repeated bytes  row = 2;
  }
```

where:

- encoding is used by receivers to verify that the packet is valid.
- policy name, version and identifier are metadata taken from the policy file.
- start time and end time indicate the duration when the data is collected.
- tables is a list of tables within the packet. This format indicates that it is possible to receive results for multiple schema paths in a single packet.
- For each table:
  - policy path is the schema path.
  - row is one or more byte arrays that represents an encoded GPB.

Key-value GPB Format

The self-describing key-value GPB format uses a generic .proto file. This file encodes data as a sequence of key-value pairs. The field names are included in the output for the receiver to interpret the data.

The following .proto file shows the field containing the key-value pairs:

```
message Telemetry {
  uint64   collection_id = 1;
  string   base_path = 2;
  string   subscription_identifier = 3;
  string   model_version = 4;
  uint64   collection_start_time = 5;
  uint64   msg_timestamp = 6;
  repeated TelemetryField fields = 14;
  uint64   collection_end_time = 15;
}

message TelemetryField {
  uint64         timestamp = 1;
  string         name = 2;
  bool           augment_data = 3;
  oneof value_by_type {
    bytes          bytes_value = 4;
    string         string_value = 5;
    bool           bool_value = 6;
    uint32         uint32_value = 7;
    uint64         uint64_value = 8;
    sint32         sint32_value = 9;
    sint64         sint64_value = 10;
    double         double_value = 11;
    float          float_value = 12;
  }
  repeated TelemetryField fields = 15;
}
```

where:

- collection_id, base_path, collection_start_time and collection_end_time provide streaming details.
- subscription_identifier is a fixed value for cadence-driven telemetry. This is used to distinguish from event-driven data.

- model_version contains a string used for the version of the data model, as applicable.

Telemetry Receiver

A telemetry receiver is used as a destination to store streamed data.

A sample receiver that handles both JSON and GPB encodings is available in the Github repository.

A copy of the `cisco.proto` file is required to compile code for a GPB receiver. The `cisco.proto` file is available in the Github repository.

If you are building your own collector, use the standard protoc compiler. For example, for the GPB compact encoding:

```
protoc --python_out . -I=/sw/packages/protoc/current/google/include/:. generic_counters.proto
 ipv4_counters.proto
```

where:

- --python_out <out_dir> specifies the location of the resulting generated files. These files are of the form `<name>_pb2.py`.
- -I <import_path> specifies the path to look for imports. This must include the location of `descriptor.proto` from Google. (in `/sw/packages`) and `cisco.proto` and the .proto files that are compiled.

All files shown in the above example are located in the local directory.