Telemetry Configuration Guide for Cisco NCS 1000 Series

First Published: 2018-03-29

Last Modified: 2024-07-23

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

- 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** 1

Scope 1

Need 1

Benefits 2

Methods of Telemetry 2

CHAPTER 2 **Configure Model-driven Telemetry** 3

Configure Dial-out Mode 4

Create a Destination Group 4

Create a Sensor Group 6

Create a Subscription 6

Validate Dial-out Configuration 8

Configure Dial-in Mode 11

Enable gRPC 11

Create a Sensor Group 13

Create a Subscription 13

Validate Dial-in Configuration 14

Event-driven Telemetry for Terminal-device Models 15

Streaming Event-Driven Telemetry for Online Insertion and Removal of Pluggables 16

gRPC Network Management Interface 27

2s Telemetry Based on GNMI Subscribe 28

gNMI Heartbeat Interval 29

CHAPTER 3 **Core Components of Model-driven Telemetry Streaming** 31

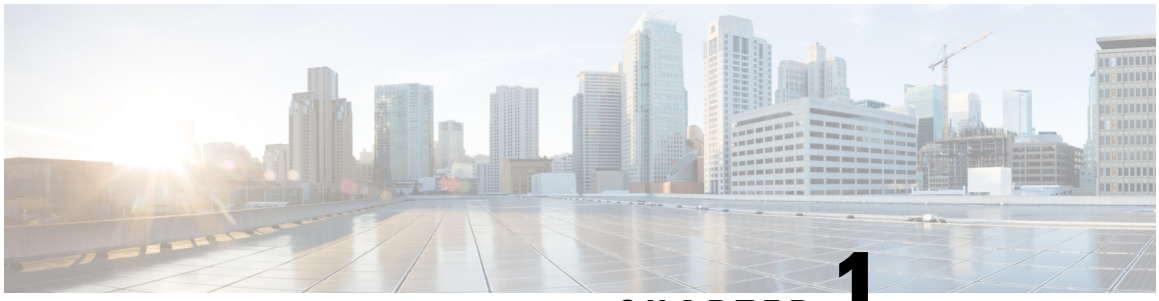
Session 31

Dial-in Mode 31

Dial-out Mode 32
Sensor Path 32
Sensor Paths Supported for EDT in NCS 1001 32
OpenConfig Sensor Paths Supported for MDT in NCS 1001 33
Sensor Paths Supported in NCS 1004 33
Sensor Paths Supported in NCS 1010 and NCS 1020 36
Subscription 39
Transport and Encoding 40

CHAPTER 4 **Configure Policy-based Telemetry** 43
Create Policy File 43
Copy Policy File 45
Configure Encoder 45
Configure JSON Encoder 46
Configure GPB Encoder 46
Verify Policy Activation 47

CHAPTER 5 **Core Components of Policy-based Telemetry Streaming** 49
Telemetry Policy File 49
Schema Paths 50
Telemetry Encoder 51
TCP Header 52
JSON Message Format 53
GPB Message Format 55
Telemetry Receiver 58



CHAPTER 1

Stream Telemetry Data

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

- Scope, on page 1
- Need, on page 1
- Benefits, on page 2
- Methods of Telemetry, on page 2

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

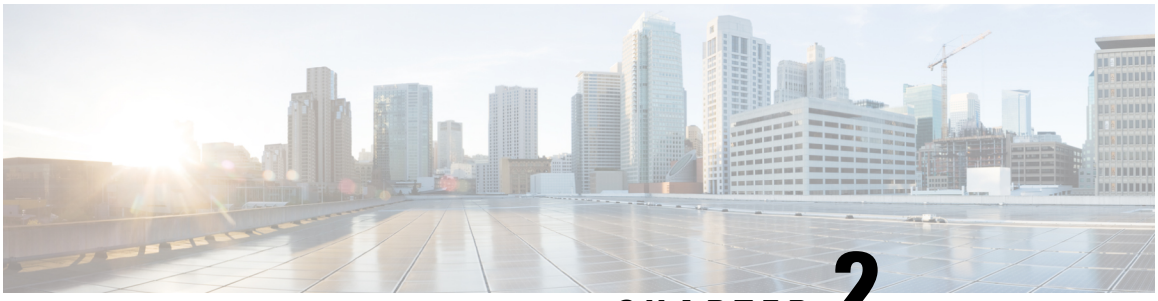
Telemetry data can be streamed using these methods:

- **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.

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.

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 11
- Event-driven Telemetry for Terminal-device Models, on page 15
- Streaming Event-Driven Telemetry for Online Insertion and Removal of Pluggables, on page 16
- gRPC Network Management Interface, on page 27
- 2s Telemetry Based on GNMI Subscribe, on page 28
- gNMI Heartbeat Interval, on page 29

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 32.

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.

Example: Destination Group for gRPC Dial-out

Note gRPC is supported in only 64-bit platforms.

gRPC protocol supports TLS and model-driven telemetry uses TLS to dial-out by default. The certificate must be copied to `/misc/config/grpc/dialout/`. To by-pass the TLS option, use `protocol grpc no-tls`.

The following is an example of a certificate to which the server certificate is connected:

```
RP/0/RP0/CPU0:ios#run

Wed Aug 24 05:05:46.206 UTC
[xr-vm_node0_RP0_CPU0:~]$ls -l /misc/config/grpc/dialout/
total 4
-rw-r--r-- 1 root root 4017 Aug 19 19:17 dialout.pem
[xr-vm_node0_RP0_CPU0:~]$
```

The CN (CommonName) used in the certificate must be configured as `protocol grpc tls-hostname <>`.

The following example shows a destination group `DGroup2` created for gRPC dial-out configuration with key-value GPB encoding, and with tls disabled:

```
Router(config)#telemetry model-driven
Router(config-model-driven)#destination-group DGroup2
Router(config-model-driven-dest)#address family ipv4 172.0.0.0 port 57500
Router(config-model-driven-dest-addr)#encoding self-describing-gpb
Router(config-model-driven-dest-addr)#protocol grpc no-tls
Router(config-model-driven-dest-addr)#commit
```

The following example shows a destination group `DGroup2` created for gRPC dial-out configuration with key-value GPB encoding, and with tls hostname:

```
Configuration with tls-hostname:
Router(config)#telemetry model-driven
```

```
Router(config-model-driven)#destination-group DGroup2
Router(config-model-driven-dest)#address family ipv4 172.0.0.0 port 57500
Router(config-model-driven-dest-addr)#encoding self-describing-gpb
Router(config-model-driven-dest-addr)#protocol grpc tls-hostname hostname.com
Router(config-model-driven-dest-addr)#commit
```

Note If only the **protocol grpc** is configured without tls option, tls is enabled by default and tls-hostname defaults to the IP address of the destination.

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 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 optics controller:

```
Router(config)#telemetry model-driven
Router(config-model-driven)#sensor-group SGroup1
Router(config-model-driven-snsr-grp)# sensor-path
Cisco-IOS-XR-controller-optics-oper:optics-oper/optics-ports/optics-port/optics-info
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: Configure Event-driven Telemetry for Optics Controller and Performance Monitoring

```
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-controller-optics-oper:optics-oper/optics-ports/optics-port/optics-info
 !
 sensor-group 2
  sensor-path Cisco-IOS-XR-pmengine-oper:performance-management-history/global/periodic/
  optics-history/optics-port-histories/optics-port-history/optics-second30-history
 !
 subscription 1
  sensor-group-id 1 sample-interval 0
  sensor-group-id 2 sample-interval 0
  destination-id 1
!
```

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
```

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: Validation for gRPC Dial-out

Note gRPC is supported in only 64-bit platforms.

```
Router#show telemetry model-driven subscription Sub2
Thu Jul 21 21:14:08.636 UTC
Subscription:  Sub2                     State: ACTIVE
-------------
  Sensor groups:
  Id                Interval(ms)        State
  SGroup2           30000               Resolved

  Destination Groups:
  Id                Encoding            Transport   State   Port    IP
  DGroup2           self-describing-gpb grpc        ACTIVE      57500   172.0.0.0
```

The telemetry data starts steaming out of the router to the destination.

Example: Configure model-driven telemetry with different sensor groups

```
RP/0/RP0/CPU0:ios#sh run telemetry model-driven

Wed Aug 24 04:49:19.309 UTC

telemetry model-driven
 destination-group 1
  address family ipv4 10.1.1.1 port 1111
   protocol grpc
  !
 !

 destination-group 2
  address family ipv4 10.2.2.2 port 2222
  !
 !
```

```
 destination-group test
  address family ipv4 172.0.0.0 port 8801
   encoding self-describing-gpb
   protocol grpc no-tls
  !
  address family ipv4 172.0.0.0 port 8901
   encoding self-describing-gpb
   protocol grpc tls-hostname chkpt1.com
  !
 !

 sensor-group 1
  sensor-path
Cisco-IOS-XR-controller-optics-oper:optics-oper/optics-ports/optics-port/optics-info
 !

 sensor-group mdt
  sensor-path Cisco-IOS-XR-pmengine-oper:performance-management-history/global/periodic/
  optics-history/optics-port-histories/optics-port-history/optics-second30-history
 !

 sensor-group generic
  sensor-path Cisco-IOS-XR-pmengine-oper:performance-management-history/global/periodic/
  optics-history/optics-port-histories/optics-port-history/optics-minute15-history
 !

 sensor-group if-oper
  sensor-path Cisco-IOS-XR-pmengine-oper:performance-management-history/global/periodic/
  optics-history/optics-port-histories/optics-port-history/optics-hour24-history
 !

 subscription mdt
  sensor-group-id mdt sample-interval 10000
 !

 subscription generic
  sensor-group-id generic sample-interval 10000
 !

 subscription if-oper
  sensor-group-id if-oper sample-interval 10000
  destination-id test
 !
!
```

A sample output from the destination with TLS certificate `chkpt1.com`:

```
RP/0/RP0/CPU0:ios#sh telemetry model-driven dest

Wed Aug 24 04:49:25.030 UTC
  Group Id        IP              Port    Encoding            Transport   State

  ------------------------------------------------------------------------------
  1               10.1.1.1         1111    none                grpc        ACTIVE

      TLS:10.1.1.1
  2               10.2.2.2         2222    none                grpc        ACTIVE

      TLS:10.2.2.2
  test            172.0.0.0   8801    self-describing-gpb grpc        Active
  test            172.0.0.0   8901    self-describing-gpb grpc        Active
      TLS:chkpt1.com
```

A sample output from the subscription:

```
RP/0/RP0/CPU0:ios#sh telemetry model-driven subscription

Wed Aug 24 04:49:48.002 UTC
Subscription:  mdt                     State: ACTIVE
-------------
  Sensor groups:
  Id                Interval(ms)        State
  mdt               10000               Resolved

Subscription:  generic                 State: ACTIVE

-------------
  Sensor groups:
  Id                Interval(ms)        State
  generic           10000               Resolved

Subscription:  if-oper                 State: ACTIVE
-------------
  Sensor groups:
  Id                Interval(ms)        State
  if-oper           10000               Resolved

  Destination Groups:
  Id                Encoding            Transport   State   Port    IP
  test              self-describing-gpb grpc        ACTIVE  8801    172.0.0.0

    No TLS :

  test              self-describing-gpb grpc        Active  8901    172.0.0.0
    TLS :             chkpt1.com


RP/0/RP0/CPU0:ios#sh telemetry model-driven subscription if-oper

Wed Aug 24 04:50:02.295 UTC
Subscription:  if-oper
-------------
  State:       ACTIVE
  Sensor groups:
  Id: if-oper
    Sample Interval:      10000 ms
    Sensor Path:
Cisco-IOS-XR-pmengine-oper:performance-management-history/global/periodic/
    optics-history/optics-port-histories/optics-port-history/optics-hour24-history
    Sensor Path State:    Resolved

  Destination Groups:
  Group Id: test
    Destination IP:       172.0.0.0
    Destination Port:     8801
    Encoding:             self-describing-gpb
    Transport:            grpc
    State:                ACTIVE
    No TLS
    Destination IP:       172.0.0.0
    Destination Port:     8901
    Encoding:             self-describing-gpb
    Transport:            grpc
    State:                ACTIVE
    TLS :                 chkpt1.com
    Total bytes sent:     120703
    Total packets sent:   11
    Last Sent time:       2016-08-24 04:49:53.52169253 +0000

  Collection Groups:
```

```
------------------
  Id: 1
  Sample Interval:      10000 ms
  Encoding:             self-describing-gpb
  Num of collection:    11
  Collection time:      Min:    69 ms Max:    82 ms
  Total time:           Min:    69 ms Avg:    76 ms Max:    83 ms
  Total Deferred:       0
  Total Send Errors:    0
  Total Send Drops:     0
  Total Other Errors:   0
  Last Collection Start:2016-08-24 04:49:53.52086253 +0000
  Last Collection End:  2016-08-24 04:49:53.52169253 +0000
  Sensor Path:
Cisco-IOS-XR-controller-optics-oper:optics-oper/optics-ports/optics-port/optics-info
```

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.

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. Commit the configuration.

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

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
```

Event-driven Telemetry for Terminal-device Models

In R6.5.2, event-driven telemetry is supported for terminal-device models. When an alarm is received, the alarm is immediately sent through the telemetry system. The event-driven telemetry is enabled by setting the sample interval value to 0 in the subscription configuration.

Example: Configure Event-driven Telemetry for Terminal-device Models

RP/0/RP0/CPU0:ios# show running-config telemetry model-driven

```
Wed Sep 19 13:57:41.418 IST
telemetry model-driven
destination-group pipeline_test
  address-family ipv4 198.51.100.3 port 5890
   encoding self-describing-gpb
   protocol tcp
  !
!

 sensor-group gkl_30seconds
  sensor-path openconfig-system:system
!
subscription gkl_30seconds
  sensor-group-id gkl_30seconds sample-interval 0
  destination-id pipeline_test
!
!
```

sensor-path openconfig-system:system means open config sensor path (global).

sample-interval 0 means telemetry is performed instantly for alarm occurrence and clearance.

Verify the Resolution of Sensor Path

Use the following show command to verify whether the sensor path is resolved.

RP/0/RP0/CPU0:ios# show telemetry model-driven subscription gkl_30s$

```
Wed Sep 26 09:59:48.326 IST
Subscription:  gkl_30seconds
-------------
  State:       Paused
  Sensor groups:
  Id: gkl_30seconds
    Sample Interval:      0 ms
    Sensor Path:          openconfig-system:system
    Sensor Path State:    Resolved

  Destination Groups:
  Group Id: pipeline_test
    Destination IP:       10.77.132.122
    Destination Port:     5900
    Encoding:             self-describing-gpb
    Transport:            tcp
    State:                NA
    No TLS

  Collection Groups:
  ------------------
```

```
Id: 1
Sample Interval:      0 ms
Encoding:             self-describing-gpb
Num of collection:    24
Collection time:      Min:     7 ms Max:    15 ms
Total time:           Min:     1 ms Avg:     5 ms Max:    15 ms
Total Deferred:       9
Total Send Errors:    0
Total Send Drops:     0
Total Other Errors:   0
No data Instances:    0
Last Collection Start:2018-09-24 18:22:36.1991344829 +0530
Last Collection End:  2018-09-25 12:39:56.3406424389 +0530
Sensor Path:          openconfig-system:system
```

Streaming Event-Driven Telemetry for Online Insertion and Removal of Pluggables

Table 1: Feature History

Feature Name	Release	Description
Event Driven Telemetry Support for Online Insertion and Removal (OIR) of Pluggables	Cisco IOS XR Release 7.8.1	A new sensor path in the OpenConfig model type is introduced to support EDT in NCS 1004 during OIR of the pluggables. It triggers telemetry data such as form factor, SONET-SDH compliance code, FEC corrected bits during removal, and state, channel data during insertion of the NCS 1004 chassis. This telemetry data helps you to track the pluggables present in the NCS 1004 chassis.

Event-driven telemetry in NCS 1004 streams operational data that are related to each lane that is configured for pluggables when OIR of pluggables occurs. In this section, the output examples show the operational data for pluggables that are configured with a single lane and four lanes. The `openconfig-platform-transceiver:transceiver` sensor path in the OpenConfig RPC model provides telemetry data of NCS 1004 pluggables that are removed or added in the NCS 1004 chassis.

Enabling EDT for OIR of Pluggables

To enable the event-driven telemetry for the OIR of pluggables, perform the following steps in order.

1. Use the `no no-tls` command in the gRPC configuration mode to enable the event-driven telemetry.
2. Run the subscription configuration file and input file together in the following format. Use the following command in your local machine to which you want to stream the event-driven telemetry data for pluggables that you remove or add.

```
<local-file-path>/<client-file> -a <IPv4-address>:<gRPC-portnumber> -insecure
-insecure_username <username> -insecure_password <password> -<encoding> "$(cat
<subscription-config file)" -dt <display-type-string>
```

Table 2: Attribute Description

Attribute	Data Type	Description
<local-file-path>	file path	Local file path of the client and subscription configuration file in your machine.
<client-file>	String	Name of the client file to enable EDT
<IPv4-address>	Decimal	IPv4 address of the NCS 1004 chassis
<gRPC-portnumber>	Integer	Port number of the gRPC port
<username>	String	Name of the admin user
<password>	Alphanumeric	User password to access the NCS 1004 chassis.
<encoding>	String	Type of the encoding format
<subscription-config file>	String	Name of the configuration file to enable EDT subscription
<display-type-string>	Character	Format of the output display

The following sample command executes the subscription file to stream the telemetry data for the OIR of the pluggables.

```
/ws/achakali-bgl/bh_final/bh_devtest/bh_auto/bh_automation/generated_files/gnmi_cli_latest
 -a 10.127.60.146:57400 -insecure -insecure_username test2 -insecure_password cisco123
-proto "$(cat transceiver_input)" -dt p
```

Subscription Configuration File

The following sample configuration file is based on gNMI specifications. It uses the openconfig sensor path and enables the EDT in NCS 1004 for the OIR of pluggables.

Note Event-driven telemetry is enabled by setting the sample interval value to **0** and mode to **ON_CHANGE** in the subscription configuration.

```
subscribe: <
    prefix: <
    >
    subscription: <
      path: <
            elem: <
                name: "openconfig-platform:components"
            >
            elem: <
                name: "component/openconfig-platform-transceiver:transceiver"
            >
          >
```

```
      mode: ON_CHANGE
      sample_interval: 0
    >
      mode: STREAM
      encoding: PROTO
>
```

The sensor path **component/openconfig-platform-transceiver:transceiver** enables the streaming of transceiver data when the pluggable is inserted or removed. The encoding format **proto** displays the streamed telemetry data in the **.proto** format. The mode **STREAM** enables the stream subscription for the set of sensory paths. For more information on the **gNMI** specifications, refer to gRPC Network Management Interface (gNMI).

Verify the Sensor Path

Use the following command to verify whether the event-driven telemetry sensor path for the OIR of pluggables in NCS 1004 is enabled.

```
RP/0/RP0/CPU0:ios#show telemetry model-driven internal subscription
```

The following output shows the **component/openconfig-platform-transceiver:transceiver** sensor path is active.

```
Fri Oct 21 15:32:28.785 IST
Subscription:  GNMI__10083376335112435231
-------------
  State:       ACTIVE
  Sensor groups:
  Id: GNMI__10083376335112435231_0
    Sample Interval:      0 ms
    Heartbeat Interval:   NA
    Sensor Path:
openconfig-platform:components/component/openconfig-platform-transceiver:transceiver
    Sensor Path State:    Resolved

  Destination Groups:
  Group Id: GNMI_1001
    Destination IP:       198.51.100.3
    Destination Port:     60058
    Encoding:             gnmi-proto
    Transport:            dialin
    State:                Active
    TLS :                 True
    Total bytes sent:     309553
    Total packets sent:   179
    Last Sent time:       2022-10-21 15:32:15.2304030650 +0530

  Collection Groups:
  ------------------
    Id: 1
    Sample Interval:      0 ms
    Heartbeat Interval:   NA
    Heartbeat always:     False
    Encoding:             gnmi-proto
    Num of collection:    1
    Incremental updates:  0
    Collection time:      Min:   709 ms Max:   709 ms
    Total time:           Min:   716 ms Avg:   716 ms Max:   716 ms
    Total Deferred:       1
    Total Send Errors:    0
    Total Send Drops:     0
    Total Other Errors:   0
```

```
    No data Instances:    0
    Last Collection Start:2022-10-21 15:32:14.2303313823 +0530
    Last Collection End:  2022-10-21 15:32:15.2304030650 +0530
    Sensor Path:
openconfig-platform:components/component/openconfig-platform-transceiver:transceiver

      Sysdb Path:
/oper/overlays/gl/oc_transceiver/openconfig-platform/components/component__list_S/*/transceiver/physical-channels/channel__list_u_bag_overlay_oc_transceiver_channel/*

      Count:          1 Method: FINDDATA Min: 709 ms Avg: 709 ms Max: 709 ms
      Item Count:     132 Status: Eventing Active
      Missed Collections:0  send bytes: 286891 packets: 130 dropped bytes: 0
      Missed Heartbeats: 0  Filtered Item Count: 0
                      success         errors          deferred/drops
      Gets            0               0
      List            0               0
      Datalist        0               0
      Finddata        3               0
      GetBulk         0               0
      Encode                          0               1
      Send                            0               0

    Id: 2
    Sample Interval:      0 ms
    Heartbeat Interval:   NA
    Heartbeat always:     False
    Encoding:             gnmi-proto
    Num of collection:    1
    Incremental updates:  0
    Collection time:      Min:   691 ms Max:   691 ms
    Total time:           Min:   694 ms Avg:   694 ms Max:   694 ms
    Total Deferred:       0
    Total Send Errors:    0
    Total Send Drops:     0
    Total Other Errors:   0
    No data Instances:    0
    Last Collection Start:2022-10-21 15:32:14.2302824953 +0530
    Last Collection End:  2022-10-21 15:32:15.2303519103 +0530
    Sensor Path:
openconfig-platform:components/component/openconfig-platform-transceiver:transceiver

      Sysdb Path:
/oper/overlays/gl/oc_transceiver/openconfig-platform/components/component__list_S/*/transceiver/state_bag_overlay_oc_transceiver_state

      Count:          1 Method: FINDDATA Min: 691 ms Avg: 691 ms Max: 691 ms
      Item Count:     49 Status: Eventing Active
      Missed Collections:0  send bytes: 22662 packets: 48 dropped bytes: 0
      Missed Heartbeats: 0  Filtered Item Count: 0
                      success         errors          deferred/drops
      Gets            0               0
      List            0               0
      Datalist        0               0
      Finddata        2               0
      GetBulk         0               0
      Encode                          0               0
      Send                            0               0

RP/0/RP0/CPU0:ios#
```

EDT Output for the OIR of Pluggables

Output for Inserted Pluggable

The following example shows the telemetry data for the addition of a single lane FR-S pluggable optic transceiver in port 13 of a 1.2T card in slot 1 in **.proto** format. The following output is the same for a single lane LR-S pluggable optic transceiver.

```
update: <
    path: <
      elem: <
        name: "state"
      >
      elem: <
        name: "present"
      >
    >
    val: <
      string_val: "PRESENT"
    >
  >
>

update: <
  timestamp: 1667367012319000000
  prefix: <
    origin: "openconfig-platform"
    elem: <
      name: "components"
    >
    elem: <
      name: "component"
      key: <
        key: "name"
        value: "Optics0_1_0_13"
      >
    >
    elem: <
      name: "openconfig-platform-transceiver:transceiver"
    >
  >
  update: <
    path: <
      elem: <
        name: "physical-channels"
      >
      elem: <
        name: "channel"
        key: <
          key: "index"
          value: "1"
        >
      >
      elem: <
        name: "state"
      >
      elem: <
        name: "index"
      >
    >
    val: <
      uint_val: 1
    >
  >
>
```

The following table describes the highlighted parameters in the preceding example.

Table 3: Parameters Description

Parameters	Description
name: "openconfig-platform-transceiver:transceiver"	Subscribed sensor path
update:	Addition of the transceiver pluggable
string_val: "PRESENT"	Indicates the availability of the pluggable
value: "Optics0_1_0_13"	FR pluggable inserted in slot 1 port 13 of 1.2T card

Output for Removed Pluggable

The following example shows the telemetry data for the removal of a single lane LR-S pluggable optic transceiver in port 13 of a 1.2T card in slot 1 in **.proto** format. The following output is the same for a single lane FR-S pluggable optic transceiver.

```
update: <
  timestamp: 1667367004302000000
  prefix: <
    origin: "openconfig-platform"
    elem: <
      name: "components"
    >
    elem: <
      name: "component"
      key: <
        key: "name"
        value: "Optics0_1_0_13"
      >
    >
    elem: <
      name: "openconfig-platform-transceiver:transceiver"
    >
  >
  delete: <
    elem: <
      name: "state"
    >
    elem: <
      name: "fec-mode"
    >
  >

.
.
output snipped
.
.
  delete: <
    elem: <
      name: "state"
    >
    elem: <
      name: "fec-uncorrectable-words"
    >
  >
  delete: <
    elem: <
      name: "state"
```

```
    >
    elem: <
      name: "fec-corrected-bits"
    >
  >
>

update: <
  timestamp: 1667367004303000000
  prefix: <
    origin: "openconfig-platform"
    elem: <
      name: "components"
    >
    elem: <
      name: "component"
      key: <
        key: "name"
        value: "Optics0_1_0_13"
      >
    >
    elem: <
      name: "openconfig-platform-transceiver:transceiver"
    >
  >
  delete: <
    elem: <
      name: "physical-channels"
    >
    elem: <
      name: "channel"
      key: <
        key: "index"
        value: "1"
      >
    >
    elem: <
      name: "state"
    >
    elem: <
      name: "index"
    >
  >
  delete: <
    elem: <
      name: "physical-channels"
    >
    elem: <
      name: "channel"
      key: <
        key: "index"
        value: "1"
      >
    >
    elem: <
      name: "state"
    >
    elem: <
      name: "description"
    >
  >
  delete: <
    elem: <
      name: "physical-channels"
```

```
    >
    elem: <
      name: "channel"
      key: <
        key: "index"
        value: "1"
      >
    >
    elem: <
      name: "state"
    >
>
```

The following table describes the highlighted parameters in the preceding example.

Table 4: Parameters Description

Parameters	Description
name: "openconfig-platform-transceiver:transceiver"	Subscribed sensor path
delete:	Removal of the transceiver pluggable
value: "Optics0_1_0_13"	LR pluggable removed in slot 1 port 13 of 1.2T card
key: < key: "index" value: "1"	Indicates the deleted lane number

The following example shows the telemetry data for the removal of a four-lane LR4-S pluggable optic transceiver in port 5 of a 1.2T card in slot 0 in **.proto** format.

```
update: <
  timestamp: 1667367249665000000
  prefix: <
    origin: "openconfig-platform"
    elem: <
      name: "components"
    >
    elem: <
      name: "component"
      key: <
        key: "name"
        value: "Optics0_0_0_5"
      >
    >
    elem: <
      name: "openconfig-platform-transceiver:transceiver"
    >
  >
  delete: <
    elem: <
      name: "state"
    >
    elem: <
      name: "fec-mode"
    >
  >
.
.
output snipped
```

```
.
.
  delete: <
    elem: <
      name: "state"
    >
    elem: <
      name: "fec-uncorrectable-words"
    >
  >
  delete: <
    elem: <
      name: "state"
    >
    elem: <
      name: "fec-corrected-bits"
    >
  >
>

update: <
  timestamp: 1667367249666000000
  prefix: <
    origin: "openconfig-platform"
    elem: <
      name: "components"
    >
    elem: <
      name: "component"
      key: <
        key: "name"
        value: "Optics0_0_0_5"
      >
    >
    elem: <
      name: "openconfig-platform-transceiver:transceiver"
    >
  >
  delete: <
    elem: <
      name: "physical-channels"
    >
    elem: <
      name: "channel"
      key: <
        key: "index"
        value: "1"
      >
    >
    elem: <
      name: "state"
    >
    elem: <
      name: "index"
    >
  >
.
.
.output snipped
.
.
>

update: <
```

```
  timestamp: 1667367249667000000
  prefix: <
    origin: "openconfig-platform"
    elem: <
      name: "components"
    >
    elem: <
      name: "component"
      key: <
        key: "name"
        value: "Optics0_0_0_5"
      >
    >
    elem: <
      name: "openconfig-platform-transceiver:transceiver"
    >
  >
  delete: <
    elem: <
      name: "physical-channels"
    >
    elem: <
      name: "channel"
      key: <
        key: "index"
        value: "2"
      >
    >
    elem: <
      name: "state"
    >
    elem: <
      name: "index"
    >
  >
.
.
output snipped
.
.
>

update: <
  timestamp: 1667367249669000000
  prefix: <
    origin: "openconfig-platform"
    elem: <
      name: "components"
    >
    elem: <
      name: "component"
      key: <
        key: "name"
        value: "Optics0_0_0_5"
      >
    >
    elem: <
      name: "openconfig-platform-transceiver:transceiver"
    >
  >
  delete: <
    elem: <
      name: "physical-channels"
    >
```

```
    elem: <
      name: "channel"
      key: <
        key: "index"
        value: "3"
      >
    >
    elem: <
      name: "state"
    >
    elem: <
      name: "index"
    >
.
.
output snipped
.
.
  >
>

update: <
  timestamp: 1667367249670000000
  prefix: <
    origin: "openconfig-platform"
    elem: <
      name: "components"
    >
    elem: <
      name: "component"
      key: <
        key: "name"
        value: "Optics0_0_0_5"
      >
    >
    elem: <
      name: "openconfig-platform-transceiver:transceiver"
    >
  >
  delete: <
    elem: <
      name: "physical-channels"
    >
    elem: <
      name: "channel"
      key: <
        key: "index"
        value: "4"
      >
    >
    elem: <
      name: "state"
    >
    elem: <
      name: "index"
    >
  >
.
.
output snipped
.
.
>
```

The following table describes the highlighted parameters in the preceding example.

Table 5: Parameters Description

Parameters	Description
name: "openconfig-platform-transceiver:transceiver"	Subscribed sensor path
delete:	Removal of the transceiver pluggable
value: "Optics0_0_0_5"	LR4 pluggable removed in slot 0 port 5 of 1.2T card
key: < key: "index" value: "4"	Indicates the deleted lane number 4

gRPC Network Management Interface

gRPC Network Management Interface is an interface for a network management system to interact with a network element.

gNMI Services

- Get - Used by the client to retrieve configuration data on the target.
- Set - Used by the client to modify configuration data of the target.
- Telemetry - Used by the client to control subscriptions to the data on the target.

Example for GET:

Syntax:

```
$./gnmi_cli -get --address=mrstn-5502-2.cisco.com:57344 \
  -proto "$(cat test.proto)" \
  -with_user_pass \
  -insecure \
  -ca_crt=ca.cert \
  -client_crt=ems.pem \
  -client_key=ems.key \
  -timeout=5s
```

Example for SET:

Syntax:

```
$ ./gnmi_cli -set --address=mrstn-5502-2.cisco.com:57344 \
  -proto "$(cat test.proto)" \
  -with_user_pass \
  -insecure \
  -ca_crt=ca.cert \
  -client_crt=ems.pem \
  -client_key=ems.key \
  -timeout=5s
```

Example for Subscribe:

Syntax:

```
$ ./gnmi_cli --address=mrstn-5502-2.cisco.com:57344 \
  -proto "$(cat test.proto)" \
  -with_user_pass \
  -insecure \
  -ca_crt=ca.cert \
  -client_crt=ems.pem \
  -client_key=ems.key \
  -timeout=5s
  -display_type string (g, group, s, single, p, proto). (default "group")
```

Subscription Mode

Subscription Mode is the mode of the subscription, specifying how the target must return values in a subscription.

Modes of the subscription:

- STREAM = 0
- ONCE = 1
- POLL = 2

ONCE Subscriptions: A subscription operating in the ONCE mode acts as a single request/response channel. The target creates the relevant update messages, transmits them, and subsequently closes the RPC.

STREAM Subscriptions: Stream subscriptions are long-lived subscriptions which continue to transmit updates relating to the set of paths that are covered within the subscription indefinitely.

2s Telemetry Based on GNMI Subscribe

gRPC Network Management Interface (GNMI) is a network management protocol used for configuration management and telemetry. gNMI provides the mechanism to install, manipulate, and delete the configuration of network devices, and to view operational data. The content provided through gNMI can be modeled using YANG.

Typically, GNMI client is configured to receive telemetry reports for every 30 seconds. The user can configure GNMI client with a sample interval of two seconds. However, the system cannot manage this delay between two collections. Hence, a characterization has been done to evaluate the actual system performance.

The characterization started to identify the maximum system load corresponding to the following scenario:

- The node is configured as a section protection node (EDFA, PSM, and EDFA modules on the three slots).
- Both the EDFA modules are configurated with grid mode=50Ghz.

  The grid-mode configuration creates up to 96 additional OTS-OCH controllers for each ots 0/slot/0/0 and ots 0/slot/0/1. The grid-mode 50GHz configuration adds 96 * 2 (number of slots with EDFA module for section protection) * 2 (bidirectional ports having OTS-OCH controllers for each EDFA module) = 384 controllers to the system.

Measurements have been performed for maximum load and for no_grid mode.

The following sensor paths are supported for telemetry testing in NCS 1001. GNMI client is configured for the following sensor paths to receive telemetry reports for every two seconds.

Sensor Path	Description
Cisco-IOS-XR-controller-optics-oper:optics-oper/optics-ports/optics-port/optics-info	Provides the data of all the OTS and OTS-OCH controllers for the system.
Cisco-IOS-XR-controller-optics-oper:optics-oper/optics-ports/optics-port/optics-info Cisco-IOS-XR-alarm-server-oper:alarm-short/alarm-additional-location-short/location/active Characterization is performed with line-rx and com-rx disconnected on both the EDFA modules. OTS-OCH controller is in maintenance state by default.	Provides the controller information of OTS and OTS-OCH controllers and active alarms.

The above sensor paths must belong to the same sensor group for which the minimum subscription interval is measured. Other sensor paths can belong to other sensor groups with subscriptions greater than 30 seconds.

The characterization performed found a value for which a sample interval for telemetry is suggested to 10 seconds for maximum load, whereas subscription time is 5 seconds for grid_mode=no grid.

gNMI Heartbeat Interval

Table 6: Feature History

Feature Name	Release Information	Feature Description
gNMI Heartbeat Interval	Cisco IOS XR Release 7.3.2	The gNMI Heartbeat Interval feature allows you to send ON_CHANGE subscription data for each heartbeat interval regardless of change in value. This feature enables you to enhance the network management system.

The gNMI heartbeat interval must be specified along with **ON_CHANGE** subscription. In this case, the value of the data items must be resent once for each heartbeat interval regardless of change in value.

Limitation

Enabling the gNMI heartbeat interval using Open Config is not supported.

Examples

The following example uses the openconfig sensor path and enables the gNMI heartbeat interval value of one hour in nanoseconds.

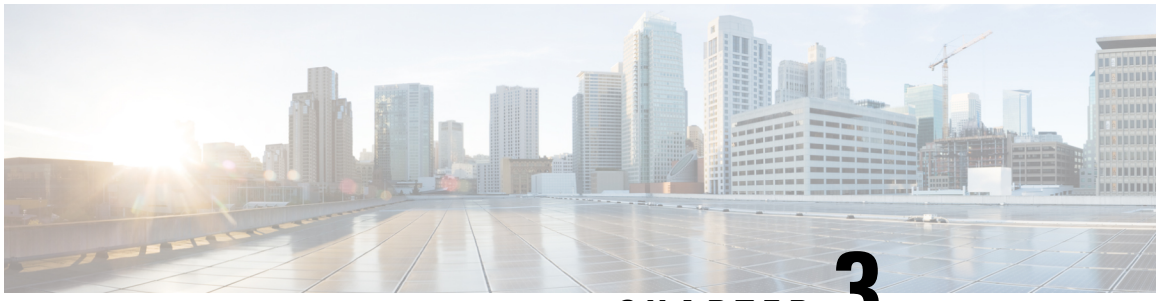
```
subscribe: <
    prefix: <
    >
    subscription: <
      path: <
            origin: "openconfig-system"
            elem: <
                name: "system"
```

```
            >
            elem: <
                name: "alarms"
            >
          >
      mode: ON_CHANGE
      heartbeat_interval: 3600000000000
    >
        mode: STREAM
        encoding: PROTO
>
```

The following example shows the enabled gNMI heartbeat interval.

```
RP/0/RP0/CPU0:ios#show run telemetry model-driven subscription sub-1
Thu Jun 17 08:41:52.400 UTC
telemetry model-driven
subscription sub-1
  sensor-group-id group1 sample-interval 0
  sensor-group-id group1 heartbeat interval 3600000000000
  sensor-group-id group1 heartbeat always
!
!
```

The **interval** attribute sends subscription data for each heartbeat interval when no events have occurred within the interval. The **always** attribute sends subscription data for each heartbeat interval even if events have occurred within the interval. The **sample-interval** attribute is enabled only with event-driven telemetry. This attribute value must be set to 0 to enable event-driven telemetry.



CHAPTER 3

Core Components of Model-driven Telemetry Streaming

The core components used in streaming model-driven telemetry data are described in this chapter.

- Session, on page 31
- Sensor Path, on page 32
- Sensor Paths Supported for EDT in NCS 1001, on page 32
- OpenConfig Sensor Paths Supported for MDT in NCS 1001, on page 33
- Sensor Paths Supported in NCS 1004, on page 33
- Sensor Paths Supported in NCS 1010 and NCS 1020, on page 36
- Subscription, on page 39
- Transport and Encoding, on page 40

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.
- **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.

Sensor Paths Supported for EDT in NCS 1001

The following sensor paths are supported for Event-based telemetry in NCS 1001.

EDT Sensor Path	Description
`Cisco-IOS-XR-controller-optics-oper:optics-oper/optics-ports/optics-port/optics-info`	This event is triggered when the configuration changes for optics/ots controller (say shutdown / no shutdown) or when the configuration changes for Transport Admin State (say sec-admin-state maintenance).
`Cisco-IOS-XR-pmengine-oper:performance-management-history/global/periodic/optics-history/optics-port-histories/optics-port-history/optics-second30-history`	This event is triggered when the 30 seconds historical PM is completed. It returns latest bucket for all optics/ots controllers.
`Cisco-IOS-XR-pmengine-oper:performance-management-history/global/periodic/optics-history/optics-port-histories/optics-port-history/optics-minute15-history`	This event is triggered when the 15 minutes historical PM is completed. It returns latest bucket for all optics/ots controllers.
`Cisco-IOS-XR-pmengine-oper:performance-management-history/global/periodic/optics-history/optics-port-histories/optics-port-history/optics-hour24-history`	This event is triggered when the 24 hours historical PM is completed. It returns latest bucket for all optics/ots controllers.

OpenConfig Sensor Paths Supported for MDT in NCS 1001

The following OpenConfig sensor paths are supported for Model-based telemetry in NCS 1001.

MDT Sensor Path	Description
`openconfig-optical-amplifier:optical-amplifier/amplifiers/amplifier`	Sensor path related to EDFA objects (ots controllers)
`openconfig-transport-line-protectionaps/aps-modules/aps-module`	Sensor path related to PSM objects (ots controllers)
`openconfig-channel-monitor:channel-monitors/channel-monitor/channels`	Sensor path related to EDFA objects (ots-och controllers and spectrum information)

Sensor Paths Supported in NCS 1004

The following sensor paths are supported in NCS 1004.

Table 7:

Model Type	Sensor Path	Description
Native	Cisco-IOS-XR-show-fpd-loc-ng-oper:show-fpd/hw-module-fpd	Provides the details of the FPGA versions of various hardware components and the packaged FPGAs with the ISO such as, BP_FPGA, XGE_FLASH.
Native	Cisco-IOS-XR-mediasvr-linux-oper:media-svr/all	Provides details of available space and occupied space in the various directory structures.
Native	Cisco-IOS-XR-alarmgr-server-oper:alarms/brief/brief-system/active	Provides the list of all active system alarms on the node.
Native	Cisco-IOS-XR-alarmgr-server-oper:alarms/brief/brief-system/suppressed	Provides the list of all suppressed system alarms on the node.
Native	Cisco-IOS-XR-alarmgr-server-oper:alarms/brief/brief-system/conditions	Provides the list of all conditional system alarms on the node.

Model Type	Sensor Path	Description
Native	Cisco-IOS-XR-controller-optics-oper:optics-oper/optics-ports	Provides the details of all the trunk or client ports of optics controller such as Baud rate, TX-RX power admin state, and LED state
Native	Cisco-IOS-XR-pmengine-oper:performance-management/otu/otu-ports/otu-port/otu-current/otu-second30/otu-second30fecs	Provides the details of OTU FEC PM counters for 30 second bucket such as, OSNR, PDL, PSR.
Native	Cisco-IOS-XR-pmengine-oper:performance-management/otu/otu-ports/otu-port/otu-current/otu-second30/otu-second30otns	Provides the details of OTU OTN PM counters for 30 second bucket such as, BBER-FE, FC-FE.
Native	Cisco-IOS-XR-pmengine-oper:performance-management/optics/optics-ports/optics-port/optics-current/optics-second30/optics-second30-optics	Provides the details of Optics PM counters for 30 second bucket such as, LB+E4C, OPT, OPR.
Native	Cisco-IOS-XR-pmengine-oper:performance-management/ethernet/ethernet-ports/ethernet-port/ethernet-current/ethernet-second30/second30-ethers	Provides the details of Ethernet PM counters for 30 second bucket such as, STAT-PKT, TX-PKT.
Native	Cisco-IOS-XR-wdsysmon-fd-oper:system-monitoring/cpu-utilization	Provides the snapshot of current CPU utilization of the node.
Native	Cisco-IOS-XR-nto-misc-oper:memory-summary/nodes/node/summary	Provides the snapshot of memory utilization of the node.
Native	Cisco-IOS-XR-osa-oper:osa/node-ids/node-id/mxponder-slices	Provides the details of muxponder slices that are configured on the node.
Native	Cisco-IOS-XR-spirit-install-instmgr-oper:software-install/active	Provides the details of the current active ISO and RPMs on the node.

Model Type	Sensor Path	Description
OpenConfig	openconfig-system:system	Checks for the alarms, host name, SSH configuration, and gRPC configuration.
OpenConfig	openconfig-platform:components/component	Checks for the inventory of the node such as subcomponents and field replaceable units such as, QSFP-100G-LR4-S, QSFP-100G-CWDM4-S.
OpenConfig	openconfig-platform:components/component/openconfig-platform-transceiver:transceiver	Provides operational data for each configured lane of pluggables that is added or removed from the chassis, such as FEC corrected bits, form factor during removal and state, channel data during insertion.
OpenConfig	openconfig-terminal-device:terminal-device	Provides the supported operational modes of the OC terminal device configuration.
OpenConfig	openconfig-terminal-device:terminal-device/logical-channels/channel/otn/state	Provides the details of PM counters for 10 second history bucket for OTN/ODU logical channels trunk ports such as, ES-NE, ESR-NE, BBE-FE.
OpenConfig	openconfig-terminal-device:terminal-device/logical-channels/channel/ethernet/state	Provides the details of PCS counters for 10 second history bucket and all the other packet counters as cumulative for the ethernet logical channel client ports such as, STAT-MULTICAST-PKT, TX-PKT, IN-MCAST.

Sensor Paths Supported in NCS 1010 and NCS 1020

Table 8: Feature History

Feature Name	Release	Description
Sensor paths supported for EDT	Cisco IOS XR Release 7.9.1	New native YANG model sensor paths and an OpenConfig sensor path are introduced for Event Driven Telemetry (EDT) in NCS 1010. EDT streams data only when a state transition occurs and thus avoids excess data collection at the receiver. EDT streams data about interface state transitions, controller shutdown, and failure, removal and insertion of the components such as power module, fan, line card, and passive modules into the NCS 1010 chassis.
Sensor paths supported for MDT	Cisco IOS XR Release 7.9.1	Model Driven Telemetry (MDT) is now supported by NCS 1010 for the native YANG models and OpenConfig sensor paths. MDT performs continuous data streaming and provides near real-time access to operational statistics.

Sensor Paths Supported for EDT in NCS 1010 and NCS 1020

These sensor paths are supported for event-based telemetry in NCS 1010 and NCS 1020.

Model Type	EDT Sensor Path	Description
Native	`Cisco-IOS-XR-platform-oper:platform/racks/rack/slots/slot/state`	This event is triggered whenever the state changes in the inventory modules. It returns the admin-state and oper-state status.
Native	`Cisco-IOS-XR-platform-oper:platform/racks/rack/slots/slot/instances/instance/state`	This event is triggered whenever the state changes in the inventory modules. It returns the admin-state and oper-state status per instance.

Model Type	EDT Sensor Path	Description
Native	Cisco-IOS-XR-alarmgr-server-oper:alarms/brief/alarm-id/active-alarms/active-alarm	This event is triggered whenever a new alarm is generated in the system. It returns the alarm, alarm-id, and a few other status values whenever a node turns active.
Native	Cisco-IOS-XR-pmengine-oper:performance-management-history/global/periodic/optics-history/optics-port-histories/optics-port-history/optics-minute15-history/optics-minute15-optics-histories/optics-minute15-optics-history/	This event is triggered when the 15-minutes historical PM is completed. It returns the latest bucket for all optics/ots controllers.
Native	Cisco-IOS-XR-pmengine-oper:performance-management-history/global/periodic/optics-history/optics-port-histories/optics-port-history/optics-second30-history	This event is triggered when the 30-minutes historical PM is completed. It returns the latest bucket for all optics/ots controllers.
Native	Cisco-IOS-XR-pmengine-oper:performance-management-history/global/periodic/optics-history/optics-port-histories/optics-port-history/optics-hour24-history	This event is triggered when the 24-hours historical PM is completed. It returns the latest bucket for all optics/ots controllers.
Native	Cisco-IOS-XR-pfi-im-cmd-ctrlr-oper:controllers/controllers/controller	This event is triggered when there is a change in state in the optical controllers for shutdown or no-shutdown states, such as in OTS, OTS-OCH, OMS,OCH, OSC and DFB controllers.
Native	Cisco-IOS-XR-ethernet-lldp-oper:lldp/nodes/node/neighbors	This event is triggered when the state of any neighboring node changes in the system. It returns state values like UP or DOWN during peer interface state changes.

Model Type	EDT Sensor Path	Description
OpenConfig	`openconfig-platform:/components/component/state`	This event is triggered when there is an online insertion and removal of any component, such as the power module, fan, and line card (OLT or ILA cards), and the NCS 1000 passive modules (NCS 1000 Breakout modules, and NCS 1000 32-Channel mux/demux patch panel). This event is also triggered whenever there is a change in the state of the component, such as failure of the power module and fan.

Sensor Paths Supported for MDT in NCS 1010 and NCS 1020

These sensor paths are supported for model-based telemetry in NCS 1010 and NCS 1020.

Model Type	MDT Sensor Path	Description
Native	`Cisco-IOS-XR-controller-ots-oper:ots-oper/ots-ports/ots-port`	Provides the details of the ots controllers such as ingress/egress gain or tilt values along with others.
Native	`Cisco-IOS-XR-controller-ots-och-oper:ots-och-oper/ots-och-ports/ots-och-port`	Provides the details of all the ots-och controllers, such as total TX-RX power, add/drop channel and other states.
Native	`Cisco-IOS-XR-controller-osc-oper:osc-oper/osc-ports/osc-port/osc-info`	Provides the details of all the osc controllers, such as total TX-RX power and other states.
Native	`Cisco-IOS-XR-controller-dfb-oper:dfb-oper/dfb-ports/dfb-port/dfb-info`	Provides the details of all the DFB controllers, such as total TX-RX power states.
Native	`Cisco-IOS-XR-controller-oms-oper:oms-oper/oms-ports/oms-port/oms-info`	Provides the details of all the OMS controllers, such as total TX-RX power and other states.
Native	`Cisco-IOS-XR-controller-och-oper:och-oper/och-ports/och-port/och-info`	Provides the details of all the och controllers, such as total TX-RX power and other states.

Model Type	MDT Sensor Path	Description
Native	Cisco-IOS-XR-pmengine-oper:performance-management/optics/optics-ports/optics-port/optics-current/optics-second30	Provides the details of optical controllers such as ots, ots-och, osc, dfb, oms, and och PM counters for the 30-seconds bucket.
Native	Cisco-IOS-XR-platform-oper:platform/racks/rack/slots/slot	Provides the details of all the inventory modules that are connected to NCS1010.
Native	Cisco-IOS-XR-alarmgr-server-oper:alarms/brief/brief-system/active	Provides the list of all active system alarms on a node.
Native	Cisco-IOS-XR-olc-oper:olc	Provides the details of optical applications such as raman-tuning, span-loss, PSD, and others.
OpenConfig	openconfig-optical-amplifierr:optical-amplifier/amplifiers	Provides the details of ingress/egress gain, tilt, gain-range, and OSRI.
Native	Cisco-IOS-XR-invmgr-oper:inventory/entities/entity	
Native	Cisco-IOS-XR-platform-oper:platform/racks/rack/slots/slot/instances/instance/state	
Native	Cisco-IOS-XR-infra-syslog-oper	
Native	Cisco-IOS-XR-envmon-oper:environmental-monitoring Cisco-IOS-XR-envmon-oper:power-management	
Native	Cisco-IOS-XR-invmgr-diag-oper:diag/racks/rack	
Native	Cisco-IOS-XR-ledmgr-oper:led-management/locations	

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 these supported transport mechanisms:

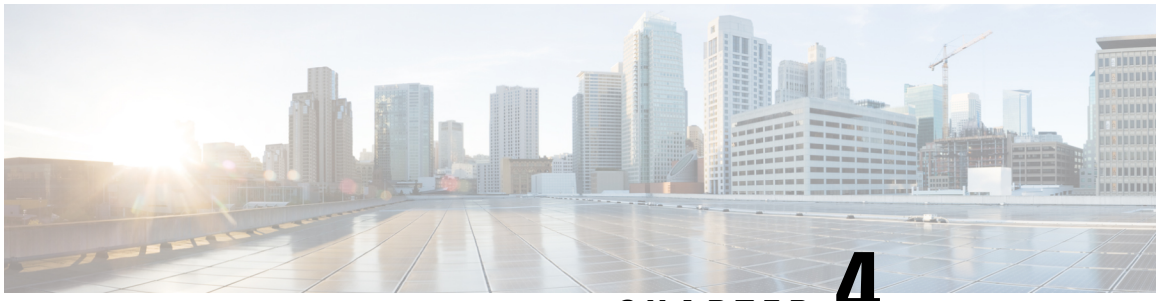
- **Google Protocol RPC (gRPC):** used for both dial-in and dial-out modes.
- **Transmission Control Protocol (TCP):** used for only dial-out mode.
- **User Datagram Protocol (UDP):** used for only dial-out mode.

Note UDP for Telemetry is not recommended for production networks. It doesn't support models that send messages larger than the UDP size limit of 65507 bytes.

The data to be streamed can be encoded into Google Protocol Buffers (GPB) or JavaScript Object Notation (JSON) 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.
- **JSON 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

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 49.

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 43
- Copy Policy File, on page 45
- Configure Encoder, on page 45
- Verify Policy Activation, on page 47

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 50.

For more information on policy files, see Telemetry Policy File, on page 49.

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 43.

For more information about the message format of JSON encoder, see JSON Message Format, on page 53

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 55

Verify Policy Activation

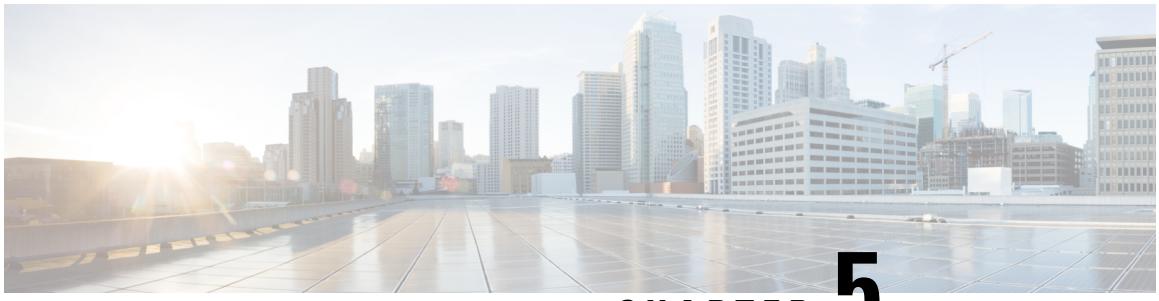
Verify that the policy is activated using the `show telemetry policies` command.

```
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 58.



CHAPTER 5

Core Components of Policy-based Telemetry Streaming

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

- Telemetry Policy File, on page 49
- Telemetry Encoder, on page 51
- Telemetry Receiver, on page 58

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 9: 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 52.

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.