Telemetry Configuration Guide for Cisco NCS 5000 Series Routers, IOS XR Release 6.0.x

First Published: 2015-12-23

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

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

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

The Cisco implementation of TCP header compression is an adaptation of a program developed by the University of California, Berkeley (UCB) as part of UCB's public domain version of the UNIX operating system. All rights reserved. Copyright © 1981, Regents of the University of California.

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

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

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

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

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

Cisco and the Cisco logo are trademarks or registered trademarks of Cisco and/or its affiliates in the U.S. and other countries. To view a list of Cisco trademarks, go to this URL: www.cisco.com go trademarks. Third-party trademarks mentioned are the property of their respective owners. The use of the word partner does not imply a partnership relationship between Cisco and any other company. (1721R)

© 2016 Cisco Systems, Inc. All rights reserved.

- 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.

© 2016 Cisco Systems, Inc. All rights reserved.

CONTENTS

CHAPTER 1

New and Changed Feature Information

This section lists all the new and changed features for the *Telemetry Configuration Guide for Cisco NCS 5000 Series Routers* .

New and Changed Telemetry Features

Table 1: Telemetry Features Added or Modified in IOS XR Release 6.0.x

Feature	Description	Changed in Release	Where Documented
Allowed list entries in the policy files to explicitly specify a list of fields to include in the streamed output.	This feature was introduced.	Release 6.0.2	*Configure Policy-based Telemetry* chapter Create Policy File, on page 8
Improved GPB workflow to eliminate the need to generate .map file.	This feature was introduced.	Release 6.0.2	*Configure Policy-based Telemetry* chapter Stream Telemetry Data, on page 3

CHAPTER 2

Stream Telemetry Data

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

Telemetry - Scope, Need, and Benefits

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.

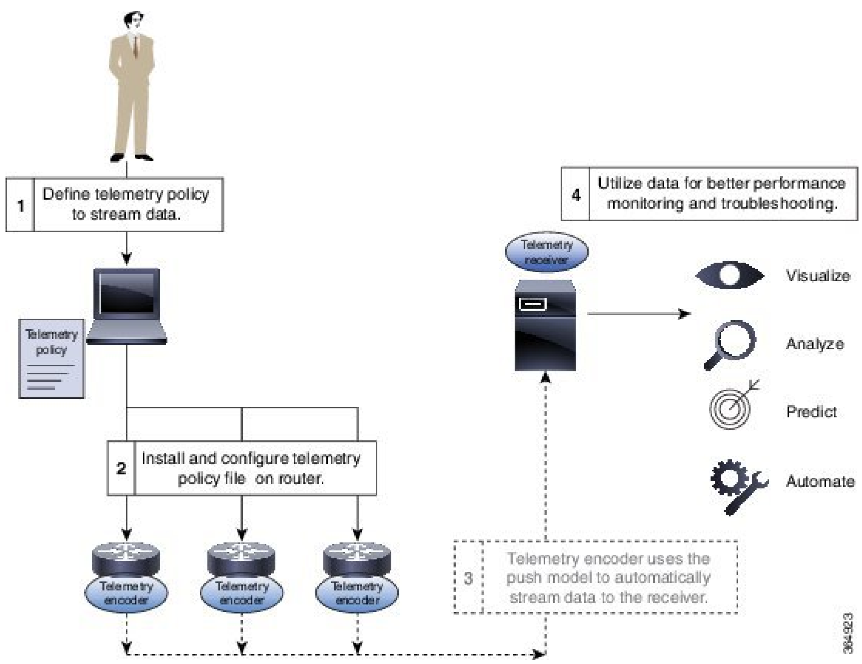
The process of streaming telemetry data uses three core components:

- **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 13

Figure 1 shows the core components used in streaming telemetry data.

Figure 1: Streaming Telemetry Data



Note The data is continuously streamed out of the router and the monitoring systems act on the data. The data at any point is real-time and no history of the data is stored.

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.

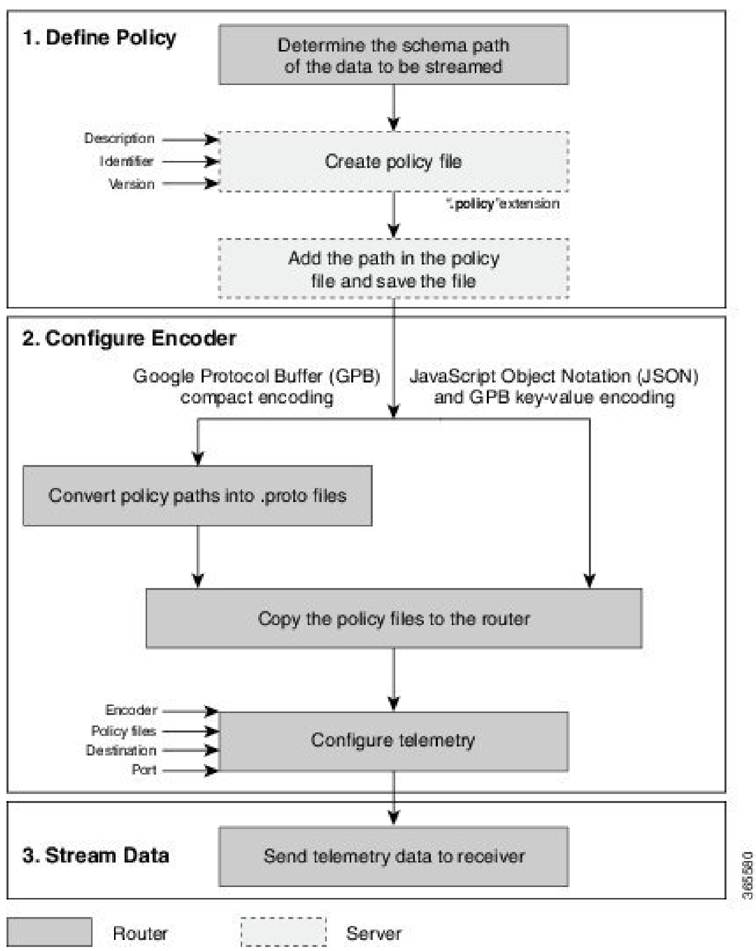
Policy-based Telemetry Process

The process of streaming real-time data using telemetry involves:

- Defining streaming frequency
- Specifying user-defined format for data collection
- Transmitting collected data to a user-specified receiver
- Viewing and analyzing the data

Figure 2 shows the tasks involved in the process of streaming telemetry data:

Figure 2: Telemetry Process



CHAPTER 3

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.

ASR9000 series routers and CRS routers do not support PBT.

The process of streaming telemetry data uses three core components:

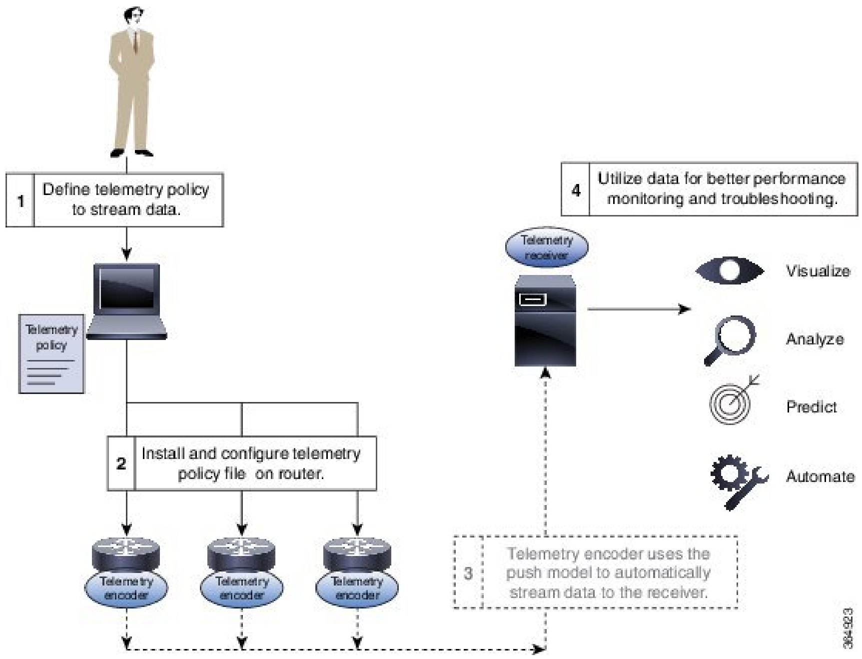
- **Telemetry Policy File** specifies the kind of telemetry data to be generated, at a specified frequency.
- **Telemetry Encoder** encapsulates the generated data into the desired format and transmits to the receiver.
- **Telemetry Receiver** is the remote management system that stores the telemetry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

For more information about the three core components, see Core Components of Policy-based Telemetry Streaming, on page 13.

Note Model-driven telemetry supersedes policy-based telemetry.

Figure 1 shows the core components used in streaming telemetry data.

Figure 3: Streaming Telemetry Data



Note The data is continuously streamed out of the router and the monitoring systems act on the data. The data at any point is real-time and no history of the data is stored.

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

- Create Policy File, on page 8
- Copy Policy File, on page 10
- Configure Encoder, on page 10
- Verify Policy Activation, on page 12

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 14.

For more information on policy files, see Telemetry Policy File, on page 13.

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 8.

For more information about the message format of JSON encoder, see JSON Message Format, on page 17

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 19

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 22.

CHAPTER 4

Core Components of Policy-based Telemetry Streaming

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

Telemetry Policy File

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

A policy file:

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

Policy file syntax

The following example shows a sample policy file:

```
{
"Name": "NameOfPolicy",
 "Metadata": {
     "Version": 25,
     "Description": "This is a sample policy to demonstrate the syntax",
     "Comment": "This is the first draft",
     "Identifier": "<data that may be sent by the encoder to the mgmt stn"
},
 "CollectionGroups": {
     "FirstGroup": {
         "Period": 10,
         "Paths": [
```

```
            "RootOper.MemorySummary.Node",
            "RootOper.RIB.VRF",
            "..."
        ]
    },
    "SecondGroup": {
        "Period": 300,
        "Paths": [
            "RootOper.Interfaces.Interface"
        ]
    }
 }
}
```

The syntax of the policy file includes:

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

  Schema path:

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

  YANG path:

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

  Allowed list entry:

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

Schema Paths

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

Table 2: 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 16.

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=1.1.1.1, Port=2000) would be represented as follows:

```
{
  "Foo":
    {
    "Destination": [
      {
        "IPAddress": 1.1.1.1,
        "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=1.1.1.1, Port=2000), would be represented by nesting each key in order:

```
{
  "Foo":
    {
    "Destination": {
        1.1.1.1: {
          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 11.1.1.1 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 at https://github.com/cisco/bigmuddy-network-telemetry-collector.

A copy of the `cisco.proto` file is required to compile code for a GPB receiver. The `cisco.proto` file is available at http://github.com/cisco/logstash-codec-bigmuddy-network-telemetry-gpb/tree/master/resources/xr6.0.0.

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.