Use gRPC Protocol to Define Network Operations with Data Models

XR devices ship with the YANG files that define the data models they support. Using a management protocol such as NETCONF or gRPC, you can programmatically query a device for the list of models it supports and retrieve the model files.

gRPC is an open-source RPC framework. It is based on Protocol Buffers (Protobuf), which is an open source binary serialization protocol. gRPC provides a flexible, efficient, automated mechanism for serializing structured data, like XML, but is smaller and simpler to use. You define the structure using protocol buffer message types in `.proto` files. Each protocol buffer message is a small logical record of information, containing a series of name-value pairs.

gRPC encodes requests and responses in binary. gRPC is extensible to other content types along with Protobuf. The Protobuf binary data object in gRPC is transported over HTTP/2.

gRPC supports distributed applications and services between a client and server. gRPC provides the infrastructure to build a device management service to exchange configuration and operational data between a client and a server. The structure of the data is defined by YANG models.

Note All 64-bit IOS XR platforms support gRPC and TCP protocols. All 32-bit IOS XR platforms support only TCP protocol.

Cisco gRPC IDL uses the protocol buffers interface definition language (IDL) to define service methods, and define parameters and return types as protocol buffer message types. The gRPC requests are encoded and sent to the router using JSON. Clients can invoke the RPC calls defined in the IDL to program the router.

The following example shows the syntax of the proto file for a gRPC configuration:

```
syntax = "proto3";

package IOSXRExtensibleManagabilityService;

service gRPCConfigOper {

    rpc GetConfig(ConfigGetArgs) returns(stream ConfigGetReply) {};

    rpc MergeConfig(ConfigArgs) returns(ConfigReply) {};

    rpc DeleteConfig(ConfigArgs) returns(ConfigReply) {};
```

```
    rpc ReplaceConfig(ConfigArgs) returns(ConfigReply) {};

    rpc CliConfig(CliConfigArgs) returns(CliConfigReply) {};

    rpc GetOper(GetOperArgs) returns(stream GetOperReply) {};

    rpc CommitReplace(CommitReplaceArgs) returns(CommitReplaceReply) {};
}
message ConfigGetArgs {
     int64 ReqId = 1;
     string yangpathjson = 2;
}

message ConfigGetReply {
    int64 ResReqId = 1;
    string yangjson = 2;
    string errors = 3;
}

message GetOperArgs {
     int64 ReqId = 1;
     string yangpathjson = 2;
}

message GetOperReply {
    int64 ResReqId = 1;
    string yangjson = 2;
    string errors = 3;
}

message ConfigArgs {
    int64 ReqId = 1;
    string yangjson = 2;

}

message ConfigReply {
    int64 ResReqId = 1;
    string errors = 2;
}

message CliConfigArgs {
    int64 ReqId = 1;
    string cli = 2;
}

message CliConfigReply {
    int64 ResReqId = 1;
    string errors = 2;
}

message CommitReplaceArgs {
     int64 ReqId = 1;
     string cli = 2;
     string yangjson = 3;
}

message CommitReplaceReply {
     int64 ResReqId = 1;
     string errors = 2;
}
```

Example for gRPCExec configuration:

```
service gRPCExec {
    rpc ShowCmdTextOutput(ShowCmdArgs) returns(stream ShowCmdTextReply) {};
    rpc ShowCmdJSONOutput(ShowCmdArgs) returns(stream ShowCmdJSONReply) {};
    rpc ActionJSON(ActionJSONArgs) returns(stream ActionJSONReply) {};
}

message ShowCmdArgs {
     int64 ReqId = 1;
     string cli = 2;
}

message ShowCmdTextReply {
    int64 ResReqId =1;
    string output = 2;
    string errors = 3;
}

message ActionJSONArgs {
 int64 ReqId = 1;
 string yangpathjson = 2;
}

message ActionJSONReply {
 int64 ResReqId = 1;
 string yangjson = 2;
 string errors = 3;
}
```

Example for OpenConfiggRPC configuration:

```
service OpenConfiggRPC {
     rpc SubscribeTelemetry(SubscribeRequest) returns (stream SubscribeResponse) {};
     rpc UnSubscribeTelemetry(CancelSubscribeReq) returns (SubscribeResponse) {};
     rpc GetModels(GetModelsInput) returns (GetModelsOutput) {};
}

message GetModelsInput {
     uint64  requestId   = 1;
     string  name        = 2;
    string  namespace   = 3;
    string  version     = 4;
    enum MODLE_REQUEST_TYPE {
         SUMMARY = 0;
         DETAIL  = 1;
     }
     MODLE_REQUEST_TYPE requestType = 5;
}

message GetModelsOutput {
     uint64  requestId   = 1;
     message ModelInfo {
         string  name        = 1;
         string  namespace   = 2;
        string  version     = 3;
         GET_MODEL_TYPE  modelType = 4;
         string modelData = 5;
     }
     repeated ModelInfo models = 2;
     OC_RPC_RESPONSE_TYPE responseCode = 3;
     string msg = 4;
}
```

This article describes, with a use case to configure interfaces on a router, how data models helps in a faster programmatic and standards-based configuration of a network, as comapared to CLI.

gRPC Operations

The following are the defined manageability service gRPC operations for Cisco IOS XR:

gRPC Operation	Description
GetConfig	Retrieves the configuration from the router.
GetModels	Gets the supported Yang models on the router
MergeConfig	Merges the input config with the existing device configuration.
DeleteConfig	Deletes one or more subtrees or leaves of configuration.
ReplaceConfig	Replaces part of the existing configuration with the input configuration.
CommitReplace	Replaces all existing configuration with the new configuration provided.
GetOper	Retrieves operational data.
CliConfig	Invokes the input CLI configuration.
ShowCmdTextOutput	Returns the output of a show command in the text form
ShowCmdJSONOutput	Returns the output of a show command in JSON form.

gRPC Operation to Get Configuration

This example shows how a gRPC GetConfig request works for CDP feature.

The client initiates a message to get the current configuration of CDP running on the router. The router responds with the current CDP configuration.

gRPC Request (Client to Router)	gRPC Response (Router to Client)

rpc GetConfig
{
"Cisco-IOS-XR-cdp-cfg:cdp": [
"cdp": "running-configuration"
]
}
 	
rpc GetConfig
{
"Cisco-IOS-XR-ethernet-lldp-cfg:lldp": [
"lldp": "running-configuration"
]
}
	{
	"Cisco-IOS-XR-cdp-cfg:cdp": {
	"timer": 50,
	"enable": true,
	"log-adjacency": [
	null
	],
	"hold-time": 180,
	"advertise-v1-only": [
	null
	]
	}
	}
	
	{
	"Cisco-IOS-XR-ethernet-lldp-cfg:lldp": {
	"timer": 60,
	"enable": true,
	"reinit": 3,
	"holdtime": 150
	}
	}

gRPC Authentication Modes

gRPC supports the following authentication modes to secure communication between clients and servers. These authentication modes help ensure that only authorized entities can access the gRPC services, like gNOI, gRIBI, and P4RT. Upon receiving a gRPC request, the device will authenticate the user and perform various authorization checks to validate the user.

The following table lists the authentication type and configuration requirements:

Table 1: gRPC Authentication Modes and Configuration Requirements

Type	Authentication Method	Authorization Method	Configuration Requirement	Requirement From Client
Metadata with TLS	username, password	username	**grpc**	username, password, and CA
Metadata without TLS	username, password	username	**grpc no-tls**	username, password
Metadata with Mutual TLS	username, password	username	**grpc tls-mutual**	username, password, client certificate, client key, and CA
Certificate based Authentication	client certificate's common name field	username from client certificate's common name field	**grpc tls-mutual** and **grpc certificate authentication**	client certificate, client key, and CA

Certificate based Authentication

In Extensible Manageability Services (EMS) gRPC, the certificates play a vital role in ensuring secure and authenticated communication. The EMS gRPC utilizes the following certificates for authentication:

```
/misc/config/grpc/ems.pem
/misc/config/grpc/ems.key
/misc/config/grpc/ca.cert
```

Note For clients to use the certificates, ensure to copy the certificates from **/misc/config/grpc/**

Generation of Certificates

These certificates are typically generated using a Certificate Authority (CA) by the device. The EMS certificates, including the server certificate (**ems.pem**), public key (**ems.key**), and CA certificate (**ca.cert**), are generated with specific parameters like the common name **ems.cisco.com** to uniquely identify the EMS server and placed in the **/misc/config/grpc/** location.

The default certificates that are generated by the server are Server-only TLS certificates and by using these certificates you can authenticate the identity of the server.

Usage of Certificates

These certificates are used for enabling secure communication through Transport Layer Security (TLS) between gRPC clients and the EMS server. The client should use **ems.pem** and **ca.cert** to initiate the TLS authentication.

To update the certificates, ensure to copy the new certificates that has been generated earlier to the location and restart the server.

Custom Certificates

If you want to use your own certificates for EMS gRPC communication, then you can follow a workflow to generate a custom certificates with the required parameters and then configure the EMS server to use these custom certificates. This process involves replacing the default EMS certificates with the custom ones and ensuring that the gRPC clients also trust the custom CA certificate. For more information on how to customize the **common-name**, see *Certificate Common-Name For Dial-in Using gRPC Protocol*.

Authenticate gRPC Services

Note Typically, gRPC clients include the username and password in the gRPC metadata fields.

Use any one of the following configuration type to authenticate any gRPC service.

- **Metadata with TLS**

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

- **Metadata without TLS**

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

```
Router(config-grpc)#no-tls
Router(config-grpc)#commit
```

- **Metadata with Mutual TLS**

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

- **Certificate based Authentication**

```
Router(config)#grpc
Router(config-grpc)#tls-mutual
Router(config-grpc)#certificate-authentication
Router(config-grpc)#commit
```

Certificate Common-Name For Dial-in Using gRPC Protocol

Table 2: Feature History Table

Feature Name	Release Information	Description
Certificate Common-Name For Dial-in Using gRPC Protocol	Release 24.1.1	Introduced in this release on: NCS 5500 fixed port routers; NCS 5700 fixed port routers; NCS 5500 modular routers (NCS 5500 line cards; NCS 5700 line cards [Mode: Compatibility; Native]) You can now specify a **common-name** for the certificate generated by the router while using gRPC dial-in. Earlier, the **common-name** in the certificate was fixed as *ems.cisco.com* and was not configurable. Using a specified **common-name** avoids potential certification failures where you may specify a hostname different from the fixed common name to connect to the router. The feature introduces these changes: **CLI:** • **grpc certificate common-name** **YANG Data Model:** • New XPath for `Cisco-IOS-XR-um-grpc-cfg.yang` • New XPath for `Cisco-IOS-XR-man-ems-cfg` (see GitHub, YANG Data Models Navigator)

When using gRPC dial-in on Cisco IOS-XR router, the **common-name** associated with the certificate generated by the router is fixed as *ems.cisco.com* and this caused failure during certificate verification.

From Cisco IOS XR Release 24.1.1, you can now have the flexibility of specifying the common-name in the certificate using the **grpc certifcate common-name** command. This allows gRPC clients to verify if the domain name in the certificate matches the domain name of the gRPC server being accessed.

Configure Certificate Common Name For Dial-in

Configure a common name to be used in EMSD certificates for gRPC dial-in.

Step 1 Configure a common name.

Example:

```
Router#config
Router(config)#grpc
Router(config-grpc)#certificate common-name cisco.com
Router(config-grpc)#commit
```

Use the show command to verify the common name:

```
Router#show grpc
Certificate common name                : cisco.com
```

Note For the above configuration to be successful, ensure to regenerate the certificate. so that the new EMSD certificates include the configured common name.

To **regenerate** the self-signed certificate, perform the following steps.

Step 2 Remove the certificates: `/misc/config/grpc/ems.pem`, `/misc/config/grpc/ems.key`, and `/misc/config/grpc/ca.cert` from `/misc/config/grpc` file.

Example:

```
Router#run ls -ltr /misc/config/grpc/

total 16
drwx------. 2 root root 4096 Feb 14 09:17 dialout
-rw-rw-rw-. 1 root root 1505 Feb 14 10:58 ems.pem
-rw-------. 1 root root 1675 Feb 14 10:58 ems.key
-rw-r--r--. 1 root root 1505 Feb 14 10:58 ca.cert

Router#run rm -rf /misc/config/grpc/ems.pem /misc/config/grpc/ems.key

Router#run ls -ltr /misc/config/grpc/

total 8
drwx------. 2 root root 4096 Feb 14 09:17 dialout
-rw-r--r--. 1 root root 1505 Feb 14 10:58 ca.cert
```

Step 3 Restart gRPC server by toggling the TLS configuration.

Configure gRPC with non TLS and then re-configure with TLS.

Example:

```
Router#config
Router(config)#grpc
Router(config-grpc)#no-tls
Router(config-grpc)#commit

Router#run ls -ltr /misc/config/grpc/

total 8
drwx------. 2 root root 4096 Feb 14 09:17 dialout
-rw-r--r--. 1 root root 1505 Feb 14 10:58 ca.cert

Router#config
Router(config)#grpc
Router(config-grpc)#no no-tls
Router(config-grpc)#commit

Router#run ls -ltr /misc/config/grpc/

total 16
```

```
drwx------. 2 root root 4096 Feb 14 09:17 dialout
-rw-rw-rw-. 1 root root 1505 Feb 14 14:23 ems.pem
-rw-------. 1 root root 1675 Feb 14 14:23 ems.key
-rw-r--r--. 1 root root 1505 Feb 14 14:23 ca.cert
```

Copy the newly generated `/misc/config/grpc/ems.pem` certificate in this path (from the device) to the gRPC client.

gRPC over UNIX Domain Sockets

Table 3: Feature History Table

Feature Name	Release Information	Description
gRPC Connections over UNIX domain sockets for Enhanced Security and Control	Release 7.5.1	This feature allows local containers and scripts on the router to establish gRPC connections over UNIX domain sockets. These sockets provide better inter-process communication eliminating the need to manage passwords for local communications. Configuring communication over UNIX domain sockets also gives you better control of permissions and security because UNIX file permissions come into force. This feature introduces the **grpc local-connection** command.

You can use local containers to establish gRPC connections via a TCP protocol where authentication using username and password is mandatory. This functionality is extended to establish gRPC connections over UNIX domain sockets, eliminating the need to manage password rotations for local communications.

When gRPC is configured on the router, the gRPC server starts and then registers services such as gRPC Network Management Interface and gRPC Network Operations Interface . After all the gRPC server registrations are complete, the listening socket is opened to listen to incoming gRPC connection requests. Currently, a TCP listen socket is created with the IP address, VRF, or gRPC listening port. With this feature, the gRPC server listens over UNIX domain sockets that must be accessible from within the container via a local connection by default. With the UNIX socket enabled, the server listens on both TCP and UNIX sockets. However, if disable the UNIX socket, the server listens only on the TCP socket. The socket file is located at /misc/app_host/ems/grpc.sock directory.

The following process shows the configuration changes required to enable or disable gRPC over UNIX domain sockets.

Step 1 Configure the gRPC server.

Example:

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

To disable the UNIX socket use the following command.

```
Router(config-grpc)#no local-connection
```

The gRPC server restarts after you enable or disable the UNIX socket. If you disable the socket, any active gRPC sessions are dropped and the gRPC data store is reset.

The scale of gRPC requests remains the same and is split between the TCP and Unix socket connections. The maximum session limit is 256, if you utilize the 256 sessions on Unix sockets, further connections on either TCP or UNIX sockets is rejected.

Step 2 Verify that the local-connection is successfully enabled.

Example:

```
Router#show grpc status
Thu Nov 25 16:51:30.382 UTC
*************************show gRPC status**********************
----------------------------------------------------------------
transport                       :     grpc
access-family                   :     tcp4
TLS                             :     enabled
trustpoint                      :
listening-port                  :     57400
local-connection                :     enabled
max-request-per-user            :     10
max-request-total               :     128
max-streams                     :     32
max-streams-per-user            :     32
vrf-socket-ns-path              :     global-vrf
min-client-keepalive-interval   :     300
________________________________________________________________
```

A gRPC client must dial into the socket to send connection requests.

The following is an example of a Go client connecting to UNIX socket:

```
const sockAddr = "/misc/app_host/ems/grpc.sock"

...
func UnixConnect(addr string, t time.Duration) (net.Conn, error) {
    unix_addr, err := net.ResolveUnixAddr("unix", sockAddr)
    conn, err := net.DialUnix("unix", nil, unix_addr)
    return conn, err
}


func main() {
...
    opts = append(opts, grpc.WithTimeout(time.Second*time.Duration(*operTimeout)))
    opts = append(opts, grpc.WithDefaultCallOptions(grpc.MaxCallRecvMsgSize(math.MaxInt32)))
    ...
    opts = append(opts, grpc.WithDialer(UnixConnect))
    conn, err := grpc.Dial(sockAddr, opts...)
    ...
}
```

gRPC Network Management Interface

gRPC Network Management Interface (gNMI) is a gRPC-based network management protocol used to modify, install or delete configuration from network devices. It is also used to view operational data, control and

generate telemetry streams from a target device to a data collection system. It uses a single protocol to manage configurations and stream telemetry data from network devices.

The subscription in a gNMI does not require prior sensor path configuration on the target device. Sensor paths are requested by the collector (such as pipeline), and the subscription mode can be specified for each path. gNMI uses gRPC as the transport protocol and the configuration is same as that of gRPC.

gNMI Wildcard in Schema Path

Table 4: Feature History Table

Feature Name	Release Information	Description
Use gNMI Get Request With Wildcard Key to Retrieve Data	Release 7.5.2	You use a gRPC Network Management Interface (gNMI) `Get` request with wildcard key to retrieve the configuration and operational data of all the elements in the data model schema paths. In earlier releases, you had to specify the correct key to retrieve data. The router returned a JSON error message if the key wasn't specified in a list node. For more information about using wildcard search in gNMI requests, see the Github repository.

gNMI protocol supports wildcards to indicate all elements at a given subtree in the schema. These wildcards are used for telemetry subscriptions or gNMI `Get` requests. The encoding of the path in gNMI uses a structured format. This format consists of a set of elements such as the path name and keys. The keys are represented as string values, regardless of their type within the schema that describes the data. gNMI supports the following options to retrieve data using wildcard search:

- **Single-level wildcard:** The name of a path element is specified as an asterisk (*). The following sample shows a wildcard as the key name. This operation returns the description for all interfaces on a device.

```
path {
  elem {
    name: "interfaces"
  }
  elem {
    name: "interface"
    key {
     key: "name"
      value: "*"
    }
  }
  elem {
    name: "config"
  }
  elem {
    name: "description"
  }
}
```

- **Multi-level wildcard:** The name of the path element is specified as an ellipsis (…). The following example shows a wildcard search that returns all fields with a description available under `/interfaces` path.

```
path {
  elem {
    name: "interfaces"
  }
  elem {
    name: "..."
  }
  elem {
    name: "description"
  }
}
```

Example: gNMI Get Request with Unique Path to a Leaf

The following is a sample `Get` request to fetch the operational state of `GigabitEthernet0/0/0/0` interface in particular.

```
path: <
    origin: "Cisco-IOS-XR-pfi-im-cmd-oper"
        elem: <
            name: "interfaces"
        >
        elem: <
            name: "interface-xr"
        >
        elem: <
            name: "interface"
            key: <
                key: "interface-name"
                value: "\"GigabitEthernet0/0/0/0\""
            >
        >
        elem: <
            name: "state"
        >
>
type: OPERATIONAL
encoding: JSON_IETF
```

The following is a sample `Get` response:

```
notification: <
  timestamp: 1597974202517298341
  update: <
    path: <
      origin: "Cisco-IOS-XR-pfi-im-cmd-oper"
       elem: <
          name: "interfaces"
       >
       elem: <
          name: "interface-xr"
       >
       elem: <
          name: "interface"
          key: <
            key: "interface-name"
            value: "\"GigabitEthernet0/0/0/0\""
           >
        >
        elem: <
```

```
            name: "state"
           >
       >
       val: <
          json_ietf_val: im-state-admin-down
       >
    >
>
error: <
>
```

Example: gNMI Get Request Without a Key Specified in the Schema Path

The following is a sample `Get` request to fetch the operational state of all interfaces.

```
path: <
    origin: "Cisco-IOS-XR-pfi-im-cmd-oper"
        elem: <
            name: "interfaces"
        >
        elem: <
            name: "interface-xr"
        >
        elem: <
            name: "interface"
        >
        elem: <
            name: "state"
        >
>
type: OPERATIONAL
encoding: JSON_IETF
```

The following is a sample `Get` response:

```
path: <
    origin: "Cisco-IOS-XR-pfi-im-cmd-oper"
        elem: <
            name: "interfaces"
        >
        elem: <
            name: "interface-xr"
        >
        elem: <
            name: "interface"
        >
        elem: <
            name: "state"
        >
>
type: OPERATIONAL
encoding: JSON_IETF
  notification: <
  timestamp: 1597974202517298341
  update: <
    path: <
      origin: "Cisco-IOS-XR-pfi-im-cmd-oper"
      elem: <
        name: "interfaces"
      >
      elem: <
        name: "interface-xr"
      >
      elem: <
```

```
      name: "interface"
      key: <
        key: "interface-name"
        value: "\"GigabitEthernet0/0/0/0\""
      >
    >
    elem: <
      name: "state"
    >
  >
  val: <
    json_ietf_val: im-state-admin-down
  >
>
update: <
  path: <
    origin: "Cisco-IOS-XR-pfi-im-cmd-oper"
    elem: <
      name: "interfaces"
    >
    elem: <
      name: "interface-xr"
    >
    elem: <
      name: "interface"
      key: <
        key: "interface-name"
        value: "\"GigabitEthernet0/0/0/1\""
      >
    >
    elem: <
      name: "state"
    >
  >
  val: <
    json_ietf_val: im-state-admin-down
  >
>
update: <
  path: <
    origin: "Cisco-IOS-XR-pfi-im-cmd-oper"
    elem: <
      name: "interfaces"
    >
    elem: <
      name: "interface-xr"
    >
    elem: <
      name: "interface"
      key: <
        key: "interface-name"
        value: "\"GigabitEthernet0/0/0/2\""
      >
    >
    elem: <
      name: "state"
    >
  >
  val: <
    json_ietf_val: im-state-admin-down
  >
>
update: <
  path: <
```

```
    origin: "Cisco-IOS-XR-pfi-im-cmd-oper"
    elem: <
      name: "interfaces"
    >
    elem: <
      name: "interface-xr"
    >
    elem: <
      name: "interface"
      key: <
        key: "interface-name"
        value: "\"MgmtEth0/RP0/CPU0/0\""
      >
    >
    elem: <
      name: "state"
    >
  >
  val: <
    json_ietf_val: im-state-admin-down
  >
>
```

gNMI Bundling of Telemetry Updates

Table 5: Feature History Table

Feature Name	Release Information	Description
gNMI Bundling Size Enhancement	Release 7.8.1	With gRPC Network Management Interface (gNMI) bundling, the router internally bundles multiple gNMI `Update` messages meant for the same client into a single gNMI `Notification` message and sends it to the client over the interface. You can now optimize the interface bandwidth utilization by accommodating more gNMI updates in a single notification message to the client. We have now increased the gNMI bundling size from 32768 to 65536 bytes, and enabled gNMI bundling size configuration through Cisco native data model. Prior releases allowed only a maximum bundling size of 32768 bytes, and you could configure only through CLI. The feature introduces new XPaths to the `Cisco-IOS-XR-telemetry-model-driven-cfg.yang` Cisco native data model to configure gNMI bundling size. To view the specification of gNMI bundling, see Github repository.

To send fewer number of bytes over the gNMI interface, multiple gNMI `Update` messages pertained to the same client are bundled and sent to the client to achieve optimized bandwidth utilization.

The router internally bundles multiple gNMI `Update` messages in a single gNMI `Notification` message of gNMI `SubscribeResponse` message. Cisco IOS XR software Release 7.8.1 supports gNMI bundling size up to 65536 bytes.

Router bundles multiple instances of the same client. For example, a router bundles interfaces `MgmtEth0/RP0/CPU0/0`, `FourHundredGigE0/0/0/0`, `FourHundredGigE0/0/0/1`, and so on, of the following path.

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

Router does not bundle messages of different client in a single gNMI `Notification` message. For example,

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

  Data under the container of the client path cannot be split into different bundles.

The gNMI `Notification` message contains a timestamp at which an event occurred or a sample is taken. The bundling process assigns a single timestamp for all bundled `Update` values. The notification timestamp is the first message of the bundle.

Note

- ON-CHANGE subscription mode does not support gNMI bundling.
- Router does not enforce bundling size in the following scenarios:
  - At the end of (N-1) message processing, if the notification message size is less than the configured bundling size, router allows one extra instance which could result in exceeding the bundling size.
  - Data of a single instance exceeding the bundling size.
- The XPath: `network-instances/network-instance/afts` does not support bundling.

Configure gNMI Bundling Size

gNMI bundling is disabled by default and the default bundling size is 32,768 bytes. gNMI bundling size ranges from 1024 to 65536 bytes. Prior to Cisco IOS XR software Release 7.8.1 the range was 1024 to 32768 bytes. You can enable gNMI bundling to all gNMI subscribe sessions and specify the bundling size.

Configuration Example

This example shows how to enable gNMI bundling and configure bundling size.

```
Router# configure
Router(config)# telemetry model-driven
Router(config-model-driven)# gnmi
Router(config-gnmi)# bundling
Router(config-gnmi-bdl)# size 2000
Router(config-gnmi-bdl)# commit
```

Running configuration

This example shows the running configuration of gNMI bundle.

```
Router# show running-config
telemetry model-driven
 gnmi
  bundling
   size 2000
```

```
  !
 !
!
```

OpenConfig Metadata for Configuration Annotations

Table 6: Feature History Table

Feature Name	Release	Description
OpenConfig Metadata for Configuration Annotations	Release 7.10.1	Introduced in this release on: NCS 5500 fixed port routers; NCS 5700 fixed port routers; NCS 5500 modular routers (NCS 5500 line cards; NCS 5700 line cards [Mode: Compatibility; Native]) You can annotate the **OpenConfig-metadata** as part of the OpenConfig **edit-config** request to the Cisco IOS XR router and later fetch using the OpenConfig **get-config** request or delete through gNMI request only. The `Set` or `Get` operations can be performed through gNMI only; not through Netconf RPCs.

In the Cisco IOS XR Software Release 7.10.1, the feature supports a specific RFC7952 based **OpenConfig-metadata** annotation. Here, **root** level node contains the **OpenConfig-metadata**, which you can set or delete through gNMI request only and can be read back while retrieving or verifying the device configuration. Netconf RPC requests are not supported.

Note The usage guidelines in this document provides the OpenConfig YANG support for a specific metadata annotation based on RFC7952 requirements for configuration commits only.

This solution is intended for the requirements of the **OpenConfig-metadata** annotation use case only and not intended to be changed for any other use beyond the scope of this document.

Following is an example for the item:

```
{
  "@": {
   "openconfig-metadata:config-metadata": "xyz" // xyz is base64 encoded string per RFC7951
 encoding rules
  }
  // Rest of configurations
}
```

The **OpenConfig-metadata** annotation is persistent across system restart. The latest **OpenConfig-metadata** annotation is preserved and it overwrites all the previous data. Also, the previous or old **OpenConfig-metadata** annotations cannot be retrieved with any operation (including configuration rollback). If the commit action fails, then the **OpenConfig-metadata** annotation is not updated. During startup failures resulting in removal of running configurations, the **OpenConfig-metadata** annotation at the time of last commit shall persist.

Example: Set Request

The following is a sample `Set` request for **OpenConfig-metadata**:

```
Request:
-------
update: {
  path: {
  }
  val: {
    json_ietf_val: "{\"openconfig-lldp:lldp\":{\"config\":{
\"enabled\":true,\"system-description\":\"test-replace\"}},
\"@\":{\"openconfig-metadata:protobuf-metadata\":
\"012345678901234567890123456789012345678901234567890
123456789012345678901234567890123456789012345678901234567890\"}}"
  }
}

Response:
--------

response: <
  path: <
  >
  op: UPDATE
>
message: <
>
timestamp: 1662150302538441219
```

Example: Get Request

The following is a sample `Get` request for **OpenConfig-metadata**:

```
Request:
-------
path:  {
  elem:  {
    name:  "@"
  }
  elem: {
    name:  "protobuf-metadata"
  }
}
type:  CONFIG
encoding:  JSON_IETF

Response:
--------

notification: <
  timestamp: 1662869232324390815
  update: <
    path: <
      origin: "openconfig"
      elem: <
        name: "@"
      >
      elem: <
        name: "protobuf-metadata"
      >
    >
    val: <
      json_ietf_val: "\"012345678901234567890123456789012345678
```

```
12345678901234567890123456789012345678901234567890
1234567890123456789\""
    >
  >
>
error: <
>
```

Verification

The **OpenConfig-metadata** annotations are stored persistently in the router and are opaque (not visible) to the IOS XR routers. However, the **show** command displays the presence and size of the **OpenConfig-metadata** annotation.

The following example displays the **show** command output:

```
Router#show cfgmgr commitdb
.
.
.
last-commit-metadata-len
[UINT32] 100000 (Ox186A0)
.
.
.
```

Note The **show** command displays only the presence and size of the **OpenConfig-metadata** annotation. If there is no **OpenConfig-metadata** annotation stored in the persistent database, then the output of the **show** command will not contain this entry.

gRPC Network Operations Interface

gRPC Network Operations Interface (gNOI) defines a set of gRPC-based microservices for executing operational commands on network devices. These services are to be used in conjunction with gRPC network management interface (gNMI) for all target state and operational state of a network. gNOI uses gRPC as the transport protocol and the configuration is same as that of gRPC. For more information about gNOI, see the Github repository.

gNOI RPCs

To send gNOI RPC requests, you need a client that implements the gNOI client interface for each RPC.

All messages within the gRPC service definition are defined as protocol buffer (.proto) files. gNOI OpenConfig proto files are located in the Github repository.

Table 7: Feature History Table

Feature Name	Release Information	Description
gNOI System Proto	Release 7.8.1	You can now avail the services of `CancelReboot` to terminate outstanding reboot request, and `KillProcess` RPCs to restart the process on device.

gNOI supports the following remote procedure calls (RPCs):

System RPCs

The RPCs are used to perform key operations at the system level such as upgrading the software, rebooting the device, and troubleshooting the network. The **system.proto** file is available in the Github repository.

RPC	Description
Reboot	Reboots the target. The router supports the following reboot options: • COLD = 1; Shutdown and restart OS and all hardware • POWERDOWN = 2; Halt and power down • HALT = 3; Halt • POWERUP = 7; Apply power
RebootStatus	Returns the status of the target reboot.
SetPackage	Places a software package including bootable images on the target device.
Ping	Pings the target device and streams the results of the ping operation.
Traceroute	Runs the traceroute command on the target device and streams the result. The default hop count is 30.
Time	Returns the current time on the target device.
SwitchControlProcessor	Switches from the current route processor to the specified route processor. If the target does not exist, the RPC returns an error message.
CancelReboot	Cancels any pending reboot request.
KillProcess	Stops an OS process and optionally restarts it.

File RPCs

The RPCs are used to perform key operations at the file level such as reading the contents if a file and its metadata. The **file.proto** file is available in the Github repository.

RPC	Description
Get	Reads and streams the contents of a file from the target device. The RPC streams the file as sequential messages with 64 KB of data.
Remove	Removes the specified file from the target device. The RPC returns an error if the file does not exist or permission is denied to remove the file.
Stat	Returns metadata about a file on the target device.
Put	Streams data into a file on the target device.
TransferToRemote	Transfers the contents of a file from the target device to a specified remote location. The response contains the hash of the transferred data. The RPC returns an error if the file does not exist, the file transfer fails or an error when reading the file. This is a blocking call until the file transfer is complete.

Certificate Management (Cert) RPCs

The RPCs are used to perform operations on the certificate in the target device. The **cert.proto** file is available in the Github repository.

RPC	Description
Rotate	Replaces an existing certificate on the target device by creating a new CSR request and placing the new certificate on the target device. If the process fails, the target rolls back to the original certificate.
Install	Installs a new certificate on the target by creating a new CSR request and placing the new certificate on the target based on the CSR.
GetCertificates	Gets the certificates on the target.
RevokeCertificates	Revokes specific certificates.
CanGenerateCSR	Asks a target if the certificate can be generated.

Interface RPCs

The RPCs are used to perform operations on the interfaces. The **interface.proto** file is available in the Github repository.

RPC	Description
SetLoopbackMode	Sets the loopback mode on an interface.
GetLoopbackMode	Gets the loopback mode on an interface.
ClearInterfaceCounters	Resets the counters for the specified interface.

Layer2 RPCs

The RPCs are used to perform operations on the Link Layer Discovery Protocol (LLDP) layer 2 neighbor discovery protocol. The **layer2.proto** file is available in the Github repository.

Feature Name	Description
ClearLLDPInterface	Clears all the LLDP adjacencies on the specified interface.

BGP RPCs

The RPCs are used to perform operations on the Link Layer Discovery Protocol (LLDP) layer 2 neighbor discovery protocol. The **bgp.proto** file is available in the Github repository.

Feature Name	Description
ClearBGPNeighbor	Clears a BGP session.

Diagnostic (Diag) RPCs

The RPCs are used to perform diagnostic operations on the target device. You assign each bit error rate test (BERT) operation a unique ID and use this ID to manage the BERT operations. The **diag.proto** file is available in the Github repository.

Feature Name	Description
StartBERT	Starts BERT on a pair of connected ports between devices in the network.
StopBERT	Stops an already in-progress BERT on a set of ports.
GetBERTResult	Gets the BERT results during the BERT or after the operation is complete.

gNOI RPCs

The following examples show the representation of few gNOI RPCs:

Get RPC

Streams the contents of a file from the target.

```
RPC to 10.105.57.106:57900
RPC start time: 20:58:27.513638
---------------------File Get Request---------------------
RPC start time: 20:58:27.513668
remote_file: "harddisk:/giso_image_repo/test.log"

---------------------File Get Response---------------------
RPC end time: 20:58:27.518413
contents: "GNOI \n\n"

hash {
method: MD5
```

```
hash: "D\002\375h\237\322\024\341\370\3619k\310\333\016\343"
}
```

Remove RPC

Remove the specified file from the target.

```
RPC to 10.105.57.106:57900
RPC start time: 21:07:57.089554
--------------------File Remove Request---------------------
remote_file: "harddisk:/sample.txt"

---------------------File Remove Response---------------------
RPC end time: 21:09:27.796217
File removal harddisk:/sample.txt successful
```

Reboot RPC

Reloads a requested target.

```
RPC to 10.105.57.106:57900
RPC start time: 21:12:49.811536
--------------------Reboot Request---------------------
RPC start time: 21:12:49.811561
method: COLD
message: "Test Reboot"
subcomponents {
origin: "openconfig-platform"
elem {
name: "components"
}
elem {
name: "component"
key {
key: "name"
value: "0/RP0"
}
}
elem {
name: "state"
}
elem {
name: "location"
}
}
---------------------Reboot Request---------------------
RPC end time: 21:12:50.023604
```

Set Package RPC

Places software package on the target.

```
RPC to 10.105.57.106:57900
RPC start time: 21:12:49.811536
--------------------Set Package Request---------------------
RPC start time: 15:33:34.378745
Sending SetPackage RPC
package {
filename: "harddisk:/giso_image_repo/<platform-version>-giso.iso"
activate: true
}
method: MD5
```

```
hash: "C\314\207\354\217\270=\021\341y\355\240\274\003\034\334"
RPC end time: 15:47:00.928361
```

Reboot Status RPC

Returns the status of reboot for the target.

```
RPC to 10.105.57.106:57900
RPC start time: 22:27:34.209473
--------------------Reboot Status Request---------------------
subcomponents {
origin: "openconfig-platform"
elem {
name: "components"
}
elem {
name: "component"
key {
key: "name"
value: "0/RP0"
}
}
elem {
name: "state"
}
elem
name: "location"
}
}

RPC end time: 22:27:34.319618

--------------------Reboot Status Response---------------------
Active : False
Wait : 0
When : 0
Reason : Test Reboot
Count : 0
```

CancelReboot RPC

Cancels any outstanding reboot

```
Request :
CancelRebootRequest
subcomponents {
origin: "openconfig-platform"
elem {
name: "components"
}
elem {
name: "component"
key {
key: "name"
value: "0/RP0/CPU0"
}
}
elem {
name: "state"
}
elem {
name: "location"
}
}
```

```
CancelRebootResponse

(rhel7-22.24.10) -bash-4.2$
```

KillProcess RPC

Kills the executing process. Either a PID or process name must be specified, and a termination signal must be specified.

```
KillProcessRequest
pid: 3451
signal: SIGNAL_TERM

KillProcessResponse
-bash-4.2$
```

gRPC Network Security Interface

Table 8: Feature History Table

Feature Name	Release Information	Feature Description
gRPC Network Security Interface	Release 7.11.1	Introduced in this release on: NCS 5500 fixed port routers; NCS 5700 fixed port routers; NCS 5500 modular routers (NCS 5500 line cards; NCS 5700 line cards [Mode: Compatibility; Native]) This release implements authorization mechanisms to restrict access to gRPC applications and services based on client permissions. This is made possible by introducing an authorization protocol buffer service for gRPC Network Security Interface (gNSI). Prior to this release, the gRPC services in the gNSI systems could be accessed by unauthorized users. This feature introduces the following change: **CLI**: • **gnsi load service authorization policy** • **show gnsi service authorization policy** To view the specification of gNSI, see Github repository.

gRPC Network Security Interface (gNSI) is a repository which contains security infrastructure services necessary for safe operations of an OpenConfig platform. The services such as authorization protocol buffer manage a network device's certificates and authorization policies.

This feature introduces a new authorization protocol buffer under gRPC gNSI. It contains gNSI.authz policies which prevent unauthorized users to access sensitive information. It defines an API that allows the configuration of the RPC service on a router. It also controls the user access and restricts authorization to update specific RPCs.

By default, gRPC-level authorization policy is provisioned using Secure ZTP. If the router is in zero-policy mode that is, in the absence of any policy, you can use gRPC authorization policy configuration to restrict access to specific users. The default authorization policy at the gRPC level can permit access to all RPCs except for the gNSI.authz RPCs.

If there is no policy specified or the policy is invalid, the router will fall back to zero-policy mode, in which the default behavior allows access to all gRPC services to all the users if their profiles are configured. If an invalid policy is configured, you can revert it by loading a valid policy using exec command **gnsi load service authorization policy.** For more information on how to create user profiles and update authorization policy for these user profiles, see How to Update gRPC-Level Authorization Policy, on page 27. Using **show gnsi service authorization policy** command, you can see the active policy in a router.

We have introduced the following commands in this release :

- **gnsi load service authorization policy**: To load and update the gRPC-level authorization policy in a router.
- **show gnsi service authorization policy**: To see the active policy applied in a router.

Note When both gNSI and gNOI are configured, gNSI takes precedence over gNOI. If niether gNSI nor gNOI is configured, then tls trsutpoint's data is considered for certificate management.

The following RPCs are used to perform key operations at the system level such as updating and displaying the current status of the authorization policy in a router.

Table 9: Operations

RPC	Description
gNSI.authz.Rotate()	Updates the gRPC-level authorization policy.
gNSI.authz.Probe()	Verifies the authenticity of a user based on the defined policy of the gRPC-level authorization policy engine.
gNSI.authz.Get()	Shows the current instance of the gRPC-level authorization policy, including the version and date of creation of the policy.

How to Update gRPC-Level Authorization Policy

gRPC-level authorization policy is configured by default at the time of router deployment using secure ZTP. You can update the same gRPC-level authorization policy using any of two the following methods:

- Using gNSI Client.

- Using exec command.

Updating the gRPC-Level Authorization Policy in the Router Using gNSI Client

Before you start

When a router boots for the first time, it should have the following prerequisites:

- The gNSI.authz service is up and running.
- The default gRPC-level authorization policy is added for all gRPC services.
- The default gRPC-level authorization policy allows access to all RPCs.

The following steps are used to update the gRPC-level authorization policy:

1. Initiate the **gNSI.authz.Rotate()** streaming RPC. This step creates a streaming connection between the router and management application (client).

Note Only one `gNSI.authz.Rotate()` must be in progress at a time. Any other RPC request is rejected by the server.

2. The client uploads new gRPC-level authorization policy using the **UploadRequest** message.

Note
- There must be only one gRPC-level authorization policy in the router. All the policies must be defined in the same gRPC-level authorization policy which is being updated. As `gNSI.authz.Rotate()` method replaces all previously defined or used policies once the **finalize** message is sent.
- The upgrade information is passed to the version and the created_on fields. These information are not used by the gNSI.authz service. It is designed to help you to track the active gRPC-level authorization policy on a particular router.

3. The router activates the gRPC-level authorization policy.
4. The router sends the `UploadResponse` message back to the client after activating the new policy.
5. The client verifies the new gRPC-level authorization policy using separate **gNSI.authz.Probe()** RPCs.
6. The client sends the **FinalizeRequest** message, indicating the previous gRPC-level authorization policy is replaced.

Note It is not recommended to close the stream without sending the **finalize** message. It results in the abandoning of the uploaded policy and rollback to the one that was active before the `gNSI.authz.Rotate()` RPC started.

Below is an example of a gRPC-level authorization policy that allows admins, V1,V2,V3 and V4, access to all RPCs that are defined by the gNSI.ssh interface. All the other users won't have access to call any of the gNSI.ssh RPCs:

```
{
  "version": "version-1",
  "created_on": "1632779276520673693",
  "policy": {
    "name": "gNSI.ssh policy",
    "allow_rules": [{
      "name": "admin-access",
      "source": {
        "principals": [
          "spiffe://company.com/sa/V1",
          "spiffe://company.com/sa/V2"
        ]
      },
      "request": {
        "paths": [
          "/gnsi.ssh.Ssh/*"
        ]
      }
    }],
    "deny_rules": [{
      "name": "sales-access",
      "source": {
        "principals": [
          "spiffe://company.com/sa/V3",
          "spiffe://company.com/sa/V4"
        ]
      },
      "request": {
        "paths": [
          "/gnsi.ssh.Ssh/MutateAccountCredentials",
          "/gnsi.ssh.Ssh/MutateHostCredentials"
        ]
      }
    }]
  }
}
```

Updating the gRPC-Level Authorization Policy file Using Exec Command

Use the following steps to update the authorization policy in the router.

1. Create the users profiles for the users who need to be added in the authorization policy. You can skip this step if you have already defined the user profiles.

   The following example creates three users who are added in the authorization policy.

   ```
   Router(config)#username V1
   Router(config-un)#group root-lr
   Router(config-un)#group cisco-support
   Router(config-un)#secret x
   Router(config-un)#exit
   Router(config)#username V2
   Router(config-un)#group root-lr
   Router(config-un)#password x
   Router(config-un)#exit
   Router(config)#username V3
   Router(config-un)#group root-lr
   Router(config-un)#password x
   Router(config-un)#commit
   ```

2. Enable **tls-mutual** to establish the secure mutual between the client and the router.

```
Router(config)#grpc
Router(config-grpc)#port 0
Router(config-grpc)#tls-mutual
Router(config-grpc)#certificate-authentication
Router(config-grpc)#commit
```

3. Define the gRPC-level authorization policy.

   The following sample gRPC-level authorization policy defines authorization policy for the users V1, V2 and V3.

```
{
    "name": "authz",
    "allow_rules": [
        {
            "name": "allow all gNMI for all users",
            "source": {
                "principals": [
                    "*"
                ]
            },
            "request": {
                "paths": [
                    "*"
                ]
            }
        }
    ],
    "deny_rules": [
         {
            "name": "deny gNMI set for oper users",
            "source": {
                "principals": [
                    "V1"
                ]
            },
            "request": {
                "paths": [
                    "/gnmi.gNMI/Get".
                ]
            }
        },

        {
            "name": "deny gNMI set for oper users",
            "source": {
                "principals": [
                    "V2"
                ]
            },
            "request": {
                "paths": [
                    "/gnmi.gNMI/Get"
                ]
            }
        },
        {
            "name": "deny gNMI set for oper users",
            "source": {
                "principals": [
                    "V3"
                ]
```

```
            },
            "request": {
                "paths": [
                    "/gnmi.gNMI/Set"
                ]
            }
        }
    ]
}
```

4. Copy the gRPC-level authorization policy to the router.

   The following example copies the gNSI Authz policy to the router:

```
-bash-4.2$ scp test.json V1@192.0.2.255:/disk0:/
Password:
test.json
                                      100%  993   161.4KB/s   00:00
-bash-4.2$
```

5. Activate the gRPC-level authorization policy to the router.

   The following example loads the policy to the router.

```
Router(config)#gnsi load service authorization policy /disk0:/test.json
Successfully loaded policy
```

Verification

Use the **show gnsi service authorization policy** to verify if the policy is active in the router.

```
Router#show gnsi service authorization policy
Wed Jul 19 10:56:14.509 UTC{
    "version": "1.0",
    "created_on": 1700816204,
    "policy": {
        "name": "authz",
        "allow_rules": [
            {
                "name": "allow all gNMI for all users",
                "request": {
                    "paths": [
                        "*"
                    ]
                },
                "source": {
                    "principals": [
                        "*"
                    ]
                }
            }
        ],
        "deny_rules": [
            {
                "name": "deny gNMI set for oper users",
                "request": {
                    "paths": [
                        "/gnmi.gNMI/*"
                    ]
                },
                "source": {
                    "principals": [
```

```
                        "User1"
                    ]
                }
            }
        ]
    }
}
```

In the following example, User1 user tries to access the **get** RPC request for which the permission is denied in the above authorization policy.

```
bash-4.2$ ./gnmi_cli -address 198.51.100.255 -ca_crt
certs/certs/ca.cert -client_crt certs/certs/User1.pem -client_key
certs/certs/User1.key -server_name ems.cisco.com -get -proto get-oper.proto
```

Output

```
E0720 14:49:42.277504   26473 gnmi_cli.go:195]
target returned RPC error for Get("path:{origin:"openconfig-interfaces"
elem:{name:"interfaces"}
 elem:{name:"interface" key:{key:"name" value:"HundredGigE0/0/0/0"}}}
 type:OPERATIONAL encoding:JSON_IETF"):
rpc error: code = PermissionDenied desc = unauthorized RPC request rejected
```

gNSI Acctz Logging

Table 10: Feature History Table

Feature Name	Release Information	Feature Description
gNSI Acctz Logging	Release 24.3.1	Introduced in this release on: NCS 5500 fixed port routers; NCS 5700 fixed port routers; NCS 5500 modular routers (NCS 5500 line cards; NCS 5700 line cards [Mode: Compatibility; Native]) You can now log and monitor AAA (Authentication, Authorization, and Accounting) accounting of gRPC operations and CLI accounting data through gNSI Acctz for effective management of network for better performance and resource utilization. You can also configure the number of gNSI accounting records that can be streamed. Previously, you could monitor the AAA accounting data through syslog only. The feature introduces these changes: **CLI:** • **grpc aaa accounting queue-size** • **show gnsi acctz statistics** To view the specification of gNSI Accounting (Acctz) RPCs and messages, see the Github repository.

gNSI Acctz Data Logging

The gNSI accounting (Acctz) is a gNSI accounting protocol that collects and transfers accounting records from a router to a remote collection service over a gRPC transport connection.

Starting from Release 24.3.1, you can log gRPC AAA accounting data through gNSI accounting (Acctz). The gNSI Acctz data is logged, stored in accounting records, and send to gNSI client for monitoring purposes. These gNSI Acctz accounting records contain

- users' login or logout times,
- network access resources such as interface IP and port, and
- duration of each session.

The gNSI Acctz logging can be done using the RecordSubscribe() gRPC request to a router. For more information on the RecordSubscribe() RPC, see the GitHub repository.

gNSI Acctz Logging Stream Capacity

The gNSI Acctz logs are recorded in a queue, maintaining a history of the 10 most recent records. When the accounting queue is full and no gNSI Acctz collectors are connected, the stream drops the records. Besides the 10 records stored for streaming, up to 512 additional records are stored during processing. As new records arrive, the data stream continues until the gNSI session ends or an error occurs, such as a client disconnection due to network issues or the server going down. If the server's output buffer remains full for an extended period, new records are dropped until the collector starts receiving them.

When the queue reaches its full capacity, the system automatically replaces the oldest records with the newest ones. The router then transmits this logged information through gNSI to gNSI client for real-time monitoring purposes. You can configure the queue size using the **grpc aaa accounting queue-size** command.

Supported Records for gNSI Acctz Logging

gNSI Acctz logging system supports Command and gRPC service records.

Table 11: CLI and gRPC Accounting Records

Command Services Accounting Records	gRPC Services Accounting Records
The command accounting records are generated for the commands executed in CLI mode and sent to gNSI Acctz collectors. The details logged include: • **Session Info:** remote/local IP addresses, remote/local ports, and channel ID. • **Authentication details**: Identity, privilege level, authentication status (PERMIT/DENY), and the cause of denial (if applicable). • **Command** and **Command status**: authentication status (PERMIT/DENY). • **Timestamp:** The time when the event was generated.	The gRPC accounting records are generated for the RPCs executed by gRPC services and sent to gNSI Acctz collectors. The details logged include: • **Session Info**: remote/local IP addresses, remote/local ports, and channel ID. • **Authentication details**: Identity and privilege level. • **RPC Service Request**: Service type, RPC name, payload, and configuration metadata. • **gRPC Service Status**: PERMIT/DENY. • **Timestamp:** The time at which the event was generated.

Default Behavior and Verification of gNSI Acctz Logging

By default, gNSI Acctz records are logged when the configuration is enabled. You can verify the gNSI Acctz using **show gnsi state**, **show gnsi acctz statistics**, and **show aaa accounting statistics** commands.

Configure gNSI Acctz Logging

Monitor AAA information through gNSI Acctz logs.

Step 1 Monitor gNSI state in the router.

Example:

```
Router# show gnsi state
Wed Jun 26 09:26:39.035 UTC
---------------GNSI state--------------
Global:
    Main Thread cerrno              : Success
    Acctz Thread cerrno             : Success
    State                           : Active
    RDSFS State                     : Active
```

Step 2 Obtain gRPC port number.

Example:

```
show grpc
Tue Aug 13 14:21:50.995 IST

Server name                             : DEFAULT
Address family                          : dual
Port                                    : 57400

Service ports
  gNMI                                  : none
  P4RT                                  : none
  gRIBI                                 : none

DSCP                                    : Default
TTL                                     : 64
VRF                                     :
Server                                  : enabled
TLS                                     : disabled
TLS mutual                              : disabled
Trustpoint                              : none
Certificate Authentication              : disabled
Certificate common name                 : ems.cisco.com
TLS v1.0                                : disabled
Maximum requests                        : 128
Maximum requests per user               : 10
Maximum streams                         : 32
Maximum streams per user                : 32
Maximum concurrent streams              : 32
Memory limit (MB)                       : 1024
Keepalive time                          : 30
Keepalive timeout                       : 20
Keepalive enforcement minimum time      : 300

TLS cipher suites
  Default                               : none
  Default TLS1.3                        : aes_128_gcm_sha256
                                        : aes_256_gcm_sha384
```

```
                                        : chacha20_poly1305_sha256

  Enable                                : none
  Disable                               : none

  Operational enable                    : none
  Operational disable                   : none
Listen addresses                        : ANY
```

Step 3 Configure gNSI queue size.

Example:

```
Router# configure
Router(config)# grpc aaa accounting queue-size 30
Router(config)# end
```

Step 4 Monitor gNSI Acctz statistics in the router.

Example:

```
Router# show gnsi acctz statistics
Tue Aug 13 05:57:24.210 UTC
SentToAAA Queue:
  Grpc services:
    GNMI:    4998 sent, 0 dropped
    GNOI:    0 sent, 0 dropped
    GNSI:    2 sent, 0 dropped
    GRIBI:   0 sent, 0 dropped
    P4RT:    0 sent, 0 dropped
    UNSPECIFIED: 0 sent, 0 dropped
  Stats:
    Total Sent: 5000
    Total Drops: 0

Streams:
  Grpc services:
    GNMI:    4996 sent, 2 dropped
    GNOI:    0 sent, 0 dropped
    GNSI:    1 sent, 0 dropped
    GRIBI:   0 sent, 0 dropped
    P4RT:    0 sent, 0 dropped
    UNSPECIFIED: 0 sent, 0 dropped
  Stats:
    Total Sent: 4997
    Total Drops: 2
  Cmd services:
    CLI:     3 sent, 0 dropped
  Stats:
    Total Sent: 3
    Total Drops: 0
Router#
```

Step 5 Provide port and IP address to the Acctz gNSI client.

Example:

```
acctz_collector -server_addr 192.0.2.111:57400 -username <user name> -password <passwod> -dieafter
600

--------------- gSNI Remote Collector ---------------
2024/08/25 22:59:13    Connecting to gNSI Server.
2024/08/25 22:59:13    gNSI Server connected.
2024/08/25 22:59:13    Started new acctz client.
2024/08/25 22:59:13    Initiate Acctz RecordSubscribe with server .
```

```
2024/08/25 22:59:13    Stream started
2024/08/25 22:59:13 Waiting for response from server.
```

Step 6 Verify the accounting record from the router.

Example:

gNSI Acctz RPC RecordSubscribe() response to the Acctz gRPC client

```
session_info:
{
  local_address:"192.0.2.111"
  local_port:57400
  remote_address:"192.0.2.1"
  remote_port:44374
  ip_proto:6
  user:
    {
      identity:"lab"
    }
}
timestamp:
{
  seconds:1718971022 nanos:105825300
}
grpc_service:
{
  service_type:GRPC_SERVICE_TYPE_GNSI
  rpc_name:"/gnsi.acctz.v1.AcctzStream/RecordSubscribe" payload_istruncated:true
  authz:
  {
    status:AUTHZ_STATUS_PERMIT
  }
}
```

AAA Accounting Statistics

```
Router# show aaa accounting statistics
Sat Aug 17 17:10:43.055 UTC
Successfully logged events:
Total events: 0
XR CLI: 0
XR SHELL: 0
GRPC:
GNMI: 0
GNSI: 2
GNOI: 0
GRIBI: 0
P4RT: 0
SLAPI: 0
NETCONF: 0
SysAdmin:
CLI: 0
SHELL: 0
Host:
SHELL: 0

Errors:
Invalid requests: 0

Max. records in buffer: 100
```

```
Total records in buffer: 0
Router#
```

IANA Port Numbers For gRPC Services

Table 12: Feature History Table

Feature Name	Release Information	Description
IANA Port Numbers For gRPC Services	Release 24.1.1	Introduced in this release on: NCS 5500 fixed port routers; NCS 5700 fixed port routers; NCS 5500 modular routers (NCS 5500 line cards; NCS 5700 line cards [Mode: Compatibility; Native]). You can now efficiently manage and customize port assignments for gNMI, gRIBI, and P4RT services without port conflicts. This is possible because Cisco IOS XR now supports the Internet Assigned Numbers Authority (IANA)-assigned specific ports for P4RT (Port 9559), gRIBI (Port 9340), and gNMI (Port 9339). You can now use both IANA-assigned and user-specified ports for these gRPC services across any specified IPv4 or IPv6 addresses. As part of this support, a new submode for gNMI in gRPC is introduced. This feature introduces the following changes: **CLI:** • **port (gRPC)** • **gnmi**

IANA (Internet Assigned Numbers Authority) manages the allocation of port numbers for various protocols. These port numbers help in distinguishing different services on a network. Service names and port numbers are used to distinguish between different services that run over transport protocols such as TCP, UDP, DCCP, and SCTP. Port numbers are assigned in various ways, based on three ranges: System Ports (0-1023), User Ports (1024-49151), and the Dynamic and/or Private Ports (49152-65535).

Earlier, the gRPC server configuration on IOS-XR allowed a usable port range of 10000-57999, with a default listening port of 57400 and all services registered to the gRPC server utilized this port for connectivity. Service-based filtering of requests on any of the ports was unavailable. Hence, the request for a specific service sent on a port designated to another service (for example, gRIBI request on gNMI port) was accepted.

From Cisco IOS XR Release 24.1.1, a new submode for gNMI is introduced in the configuration model to allow for service-level port customization. The existing gRPC configuration model includes submodes for P4RT and gRIBI. This submode will enable you to configure specific ports for gNMI, gRIBI, and P4RT services independently. You can configure gNMI, gRIBI, and P4RT services using the gRPC submode

command to set the default port for each service. The **port** command under service submode, allows you to modify the port as needed, while adhering to the defined port range.

Disabling the **port** command will cause the service to use the default or IANA port.

You can set custom ports for gNMI, gRIBI, and P4RT services within the defined range, including default IANA ports like 9339, 9340, and 9559 (respectively). The gRPC service will continue to maintain its default port within the specified range (57344-57999). Any changes made to the gRPC default port will not impact the service port configurations for gNMI, gRIBI, and P4RT. Requests which are sent on a port designated for a specific service (example, gRIBI request on gNMI port) will be accepted. This flexibility allows for seamless communication across different service ports and the general gRPC port.

Starting from Release 24.2.1, the allowed port range is 1024-65535.

Configure gRPC Service-Level Port

To configure a default listening port for the gRPC services such as gNMI, gRIBI, and P4RT, use the respective service command (**gnmi**, **gribi**, or **p4rt**) under the gRPC configuration mode.

To specify a port number for gRPC, gNMI, gRIBI, and P4RT services within the defined range, use the **port** command under respective submodes.

Note XR Ephemeral port range: 15232–57343

If the configured port is in the range of IANA registered ports (1024-49151) or XR ephemeral ports (15232-57343), a syslog is generated with a NOTICE to warn the user for a possible application conflict.

Resetting the port reverts to the default service port, and disabling the service stops listening on that port.

Configure the port number for a service.

The following examples display the service-level port configurations.

- **For gNMI service:**

  This configuration creates a gRPC listener with the default or IANA ratified gNMI port of 9339.

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

  Verify the listening port created for gNMI service.

```
Router#show running-config grpc
grpc
  gnmi
!
```

  The **port** command under gNMI submode allows the port to be modified in the port range or IANA ratified port.

```
Router(config-grpc)#gnmi
Router(config-grpc-gnmi)#port 9339
Router(config-grpc-gnmi)#commit
```

  Verify the port number.

```
Router#show running-config grpc
grpc
  gnmi
```

```
    port 9339
!
```

- **For P4RT service:**

  This configuration creates a gRPC listener with the default or IANA ratified P4RT port of 9559.

```
Router(config-grpc)#p4rt
Router(config-grpc-p4rt)#commit
```

  Verify the listening port created for P4RT service.

```
Router#show running-config grpc
grpc
  p4rt
!
```

  The **port** command under P4RT submode allows the port to be modified in the port range or IANA ratified port.

```
Router(config-grpc)#p4rt
Router(config-grpc-p4rt)#port 9559
Router(config-grpc-p4rt)#commit
```

  Verify the port number.

```
Router#show running-config grpc
grpc
  p4rt
    port 9559
!
```

- **For gRIBI service:**

  This configuration creates a gRPC listener with the default or IANA ratified gRIBI port of 9340.

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

  Verify the listening port created for gRIBI service.

```
Router#show running-config grpc
grpc
  gribi
!
```

  The **port** command under gRIBI submode allows the port to be modified in the port range or IANA ratified port.

```
Router(config-grpc)#gribi
Router(config-grpc-gribi)#port 9340
Router(config-grpc-gribi)#commit
```

  Verify the port number.

```
Router#show running-config grpc
grpc
  gribi
    port 9340
!
```

Unconfiguring the port command in a service

and

Unconfiguring a service under gRPC

- Unconfiguring the **port** command results in using the default port for the respective service.

  Example:

Unconfiguring the **port** command will result in a gNMI service using the default gNMI port.

```
Router(config-grpc)#gnmi
Router(config-grpc-gnmi)#no port
Router(config-grpc-gnmi)#commit
```

Verify the service port configuration.

```
Router#show running-config grpc
grpc
  gnmi
!
```

- Unconfiguring a service removes the listener for the respective port and no requests will be accepted on that port.

  Example:

  Unconfiguring gNMI disables the requests on port 9339.

```
Router(config-grpc)#no gnmi
Router(config-grpc-gnmi)#commit
```

  Verify the port configuration.

```
Router#show running-config grpc
grpc
!
```

Configure Interfaces Using Data Models in a gRPC Session

Table 13: Feature History Table

Feature Name	Release Information	Description
Set Limit on Concurrent Streams for gRPC Server	Release 24.1.1	You can prevent potential security attacks by disallowing any single gRPC server client on Cisco IOS XR from consuming excessive resources and monopolizing connection resources, both of which can be potential attack vectors. Such prevention is possible because you now have the option to configure the gRPC server to limit the number of concurrent streams per gRPC connection. The feature introduces the **grpc max-concurrent-streams** command. **YANG Data Models:** • `Cisco-IOS-XR-man-ems-oper.yang` • `Cisco-IOS-XR-man-ems-cfg.yang` (see GitHub, YANG Data Models Navigator)

Google-defined remote procedure call () is an open-source RPC framework. gRPC supports IPv4 and IPv6 address families. The client applications use this protocol to request information from the router, and make configuration changes to the router.

The process for using data models involves:

- Obtain the data models.
- Establish a connection between the router and the client using gRPC communication protocol.
- Manage the configuration of the router from the client using data models.

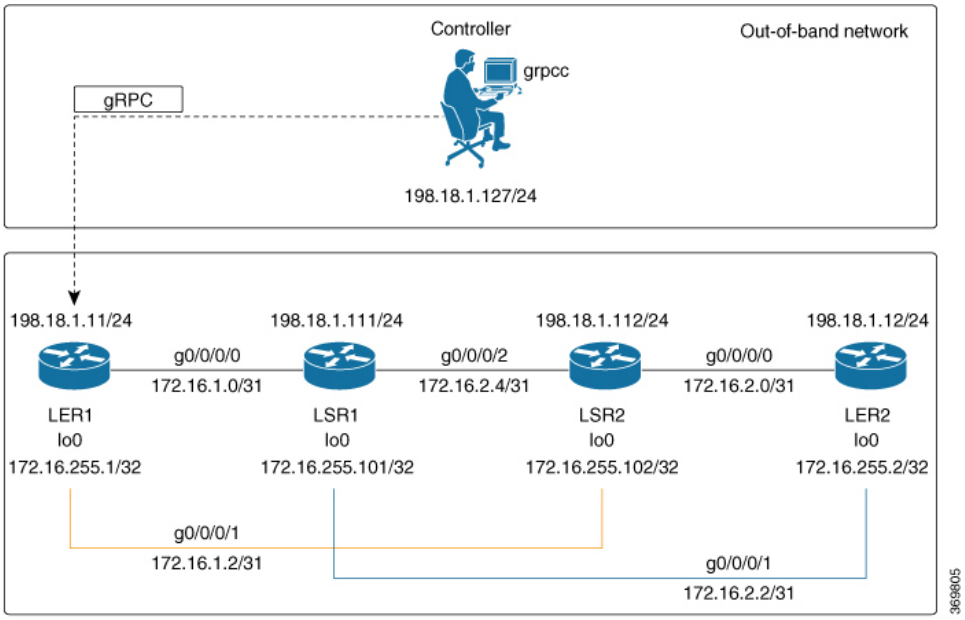
Note Configure AAA authorization to restrict users from uncontrolled access. If AAA authorization is not configured, the command and data rules associated to the groups that are assigned to the user are bypassed. An IOS-XR user can have full read-write access to the IOS-XR configuration through Network Configuration Protocol (NETCONF), google-defined Remote Procedure Calls (gRPC) or any YANG-based agents. In order to avoid granting uncontrolled access, enable AAA authorization using **aaa authorization exec** command before setting up any configuration. For more information about configuring AAA authorization, see the *System Security Configuration Guide*.

In this section, you use native data models to configure loopback and ethernet interfaces on a router using a gRPC session.

Consider a network topology with four routers and one controller. The network consists of label edge routers (LER) and label switching routers (LSR). Two routers LER1 and LER2 are label edge routers, and two routers LSR1 and LSR2 are label switching routers. A host is the controller with a gRPC client. The controller communicates with all routers through an out-of-band network. All routers except LER1 are pre-configured with proper IP addressing and routing behavior. Interfaces between routers have a point-to-point configuration with `/31` addressing. Loopback prefixes use the format `172.16.255.x/32`.

The following image illustrates the network topology:

Figure 1: Network Topology for gRPC session



You use Cisco IOS XR native model `Cisco-IOS-XR-ifmgr-cfg.yang` to programmatically configure router LER1.

Before you begin

- Retrieve the list of YANG modules on the router using NETCONF monitoring RPC. For more information
- Configure Transport Layer Security (TLS). Enabling gRPC protocol uses the default HTTP/2 transport with no TLS. gRPC mandates AAA authentication and authorization for all gRPC requests. If TLS is not configured, the authentication credentials are transferred over the network unencrypted. Enabling TLS ensures that the credentials are secure and encrypted. Non-TLS mode can only be used in secure internal network.

Step 1 Enable gRPC Protocol

To configure network devices and view operational data, gRPC proptocol must be enabled on the server. In this example, you enable gRPC protocol on LER1, the server.

Note Cisco IOS XR 64-bit platforms support gRPC protocol. The 32-bit platforms do not support gRPC protocol.

a) Enable gRPC over an HTTP/2 connection.

Example:

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

The port number ranges from 57344 to 57999. If a port number is unavailable, an error is displayed.

Starting Release 24.1.1, you can now configure IANA port numbers for specified gRPC services. To see the port numbers for the various gRPC services, see *Support IANA Port Numbers*.

b) Set the session parameters.

Example:

```
Router(config)#grpc {address-family | certificate-authentication | dscp | max-concurrent-streams
 | max-request-per-user | max-request-total | max-streams |
max-streams-per-user | no-tls | tlsv1-disable | tls-cipher | tls-mutual | tls-trustpoint |
service-layer | vrf}
```

where:

- `address-family`: set the address family identifier type.
- `certificate-authentication`: enables certificate based authentication
- `dscp`: set QoS marking DSCP on transmitted gRPC.
- `max-concurrent-streams`: set the limit on the maximum concurrent streams per gRPC connection to be applied on the server.
- `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
- `tlsv1-disable`: disable TLS version 1.0
- `service-layer`: enable the grpc service layer configuration.

  This parameter is not supported in Cisco ASR 9000 Series Routers, Cisco NCS560 Series Routers, , and Cisco NCS540 Series Routers.
- `tls-cipher`: enable the gRPC TLS cipher suites.
- `tls-mutual`: set the mutual authentication.
- `tls-trustpoint`: configure trustpoint.
- `server-vrf`: enable server vrf.

After gRPC is enabled, use the YANG data models to manage network configurations.

Step 2 Configure the interfaces.

In this example, you configure interfaces using Cisco IOS XR native model `Cisco-IOS-XR-ifmgr-cfg.yang`. You gain an understanding about the various gRPC operations while you configure the interface. For the complete list of operations, see gRPC Operations, on page 4. In this example, you merge configurations with `merge-config` RPC, retreive operational statistics using `get-oper` RPC, and delete a configuration using `delete-config` RPC. You can explore the structure of the data model using YANG validator tools such as pyang.

LER1 is the gRPC server, and a command line utility `grpcc` is used as a client on the controller. This utility does not support YANG and, therefore, does not validate the data model. The server, LER1, validates the data mode.

Note The OC interface maps all IP configurations for parent interface under a VLAN with index 0. Hence, do not configure a sub interface with tag 0.

a) Explore the XR configuration model for interfaces and its IPv4 augmentation.

Example:

```
controller:grpc$ pyang --format tree --tree-depth 3 Cisco-IOS-XR-ifmgr-cfg.yang
Cisco-IOS-XR-ipv4-io-cfg.yang
module: Cisco-IOS-XR-ifmgr-cfg
    +--rw global-interface-configuration
    | +--rw link-status? Link-status-enum
    +--rw interface-configurations
        +--rw interface-configuration* [active interface-name]
            +--rw dampening
            | ...
            +--rw mtus
            | ...
            +--rw encapsulation
            | ...
            +--rw shutdown? empty
            +--rw interface-virtual? empty
            +--rw secondary-admin-state? Secondary-admin-state-enum
            +--rw interface-mode-non-physical? Interface-mode-enum
            +--rw bandwidth? uint32
            +--rw link-status? empty
            +--rw description? string
            +--rw active Interface-active
            +--rw interface-name xr:Interface-name
            +--rw ipv4-io-cfg:ipv4-network
            | ...
            +--rw ipv4-io-cfg:ipv4-network-forwarding ...
```

b) Configure a loopback0 interface on LER1.

Example:

```
controller:grpc$ more xr-interfaces-lo0-cfg.json
{
 "Cisco-IOS-XR-ifmgr-cfg:interface-configurations":
  { "interface-configuration": [
    {
      "active": "act",
      "interface-name": "Loopback0",
      "description": "LOCAL TERMINATION ADDRESS",
      "interface-virtual": [
        null
       ],
       "Cisco-IOS-XR-ipv4-io-cfg:ipv4-network": {
        "addresses": {
            "primary": {
              "address": "172.16.255.1",
              "netmask": "255.255.255.255"
       }
      }
     }
    }
   ]
 }
}
```

c) Merge the configuration.

Example:

```
controller:grpc$ grpcc -username admin -password admin -oper merge-config
-server_addr 198.18.1.11:57400 -json_in_file xr-interfaces-gi0-cfg.json
```

```
emsMergeConfig: Sending ReqId 1
emsMergeConfig: Received ReqId 1, Response '
'
```

d) Configure the ethernet interface on LER1.

Example:

```
controller:grpc$ more xr-interfaces-gi0-cfg.json
{
 "Cisco-IOS-XR-ifmgr-cfg:interface-configurations": {
   "interface-configuration": [
    {
     "active": "act",
     "interface-name": "GigabitEthernet0/0/0/0",
     "description": "CONNECTS TO LSR1 (g0/0/0/0)",
     "Cisco-IOS-XR-ipv4-io-cfg:ipv4-network": {
        "addresses": {
         "primary": {
            "address": "172.16.1.0",
            "netmask": "255.255.255.254"
      }
     }
    }
   }
  ]
 }
}
```

e) Merge the configuration.

Example:

```
controller:grpc$ grpcc -username admin -password admin -oper merge-config
-server_addr 198.18.1.11:57400 -json_in_file xr-interfaces-gi0-cfg.json
emsMergeConfig: Sending ReqId 1
emsMergeConfig: Received ReqId 1, Response '
'
```

f) Enable the ethernet interface `GigabitEthernet 0/0/0/0` on LER1 to bring up the interface. To do this, delete `shutdown` configuration for the interface.

Example:

```
controller:grpc$ grpcc -username admin -password admin -oper delete-config
-server_addr 198.18.1.11:57400 -yang_path "$(< xr-interfaces-gi0-shutdown-cfg.json )"
emsDeleteConfig: Sending ReqId 1, yangJson {
 "Cisco-IOS-XR-ifmgr-cfg:interface-configurations": {
   "interface-configuration": [
   {
    "active": "act",
    "interface-name": "GigabitEthernet0/0/0/0",
    "shutdown": [
      null
    ]
   }
  ]
 }
}
emsDeleteConfig: Received ReqId 1, Response ''
```

Step 3 Verify that the loopback interface and the ethernet interface on router LER1 are operational.

Example:

```
controller:grpc$ grpcc -username admin -password admin -oper get-oper
-server_addr 198.18.1.11:57400 -oper_yang_path "$(< xr-interfaces-briefs-oper-filter.json )"
emsGetOper: Sending ReqId 1, yangPath {
  "Cisco-IOS-XR-pfi-im-cmd-oper:interfaces": {
    "interface-briefs": [
      null
     ]
  }
}
{ "Cisco-IOS-XR-pfi-im-cmd-oper:interfaces": {
  "interface-briefs": {
    "interface-brief": [
    {
      "interface-name": "GigabitEthernet0/0/0/0",
      "interface": "GigabitEthernet0/0/0/0",
      "type": "IFT_GETHERNET",
      "state": "im-state-up",
      "actual-state": "im-state-up",
      "line-state": "im-state-up",
      "actual-line-state": "im-state-up",
      "encapsulation": "ether",
      "encapsulation-type-string": "ARPA",
      "mtu": 1514,
      "sub-interface-mtu-overhead": 0,
      "l2-transport": false,
      "bandwidth": 1000000
     },
     {
      "interface-name": "GigabitEthernet0/0/0/1",
      "interface": "GigabitEthernet0/0/0/1",
      "type": "IFT_GETHERNET",
      "state": "im-state-up",
      "actual-state": "im-state-up",
      "line-state": "im-state-up",
      "actual-line-state": "im-state-up",
      "encapsulation": "ether",
      "encapsulation-type-string": "ARPA",
      "mtu": 1514,
      "sub-interface-mtu-overhead": 0,
      "l2-transport": false,
      "bandwidth": 1000000
     },
     {
      "interface-name": "Loopback0",
      "interface": "Loopback0",
      "type": "IFT_LOOPBACK",
      "state": "im-state-up",
      "actual-state": "im-state-up",
      "line-state": "im-state-up",
      "actual-line-state": "im-state-up",
      "encapsulation": "loopback",
      "encapsulation-type-string": "Loopback",
      "mtu": 1500,
      "sub-interface-mtu-overhead": 0,
      "l2-transport": false,
      "bandwidth": 0
   },
   {
      "interface-name": "MgmtEth0/RP0/CPU0/0",
      "interface": "MgmtEth0/RP0/CPU0/0",
      "type": "IFT_ETHERNET",
      "state": "im-state-up",
```

```
      "actual-state": "im-state-up",
      "line-state": "im-state-up",
      "actual-line-state": "im-state-up",
      "encapsulation": "ether",
      "encapsulation-type-string": "ARPA",
      "mtu": 1514,
      "sub-interface-mtu-overhead": 0,
      "l2-transport": false,
      "bandwidth": 1000000
   },
   {
      "interface-name": "Null0",
      "interface": "Null0",
      "type": "IFT_NULL",
      "state": "im-state-up",
      "actual-state": "im-state-up",
      "line-state": "im-state-up",
      "actual-line-state": "im-state-up",
      "encapsulation": "null",
      "encapsulation-type-string": "Null",
      "mtu": 1500,
      "sub-interface-mtu-overhead": 0,
      "l2-transport": false,
      "bandwidth": 0
    }
   ]
  }
 }
}
emsGetOper: ReqId 1, byteRecv: 2325
```


In summary, router LER1, which had minimal configuration, is now programmatically configured using data models with an ethernet interface and is assigned a loopback address. Both these interfaces are operational and ready for network provisioning operations.