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) {};

}

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

message ShowCmdTextReply {
    int64 ResReqId =1;
    string output = 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.

• Configure Interfaces Using Data Models in a gRPC Session, on page 29

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 LLDP feature.

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

<table>
<tr><th>gRPC Request (Client to Router)</th><th>gRPC Response (Router to Client)</th></tr>
<tr><td><pre>rpc GetConfig
  {
   "Cisco-IOS-XR-cdp-cfg:cdp": [
    "cdp": "running-configuration"
    ]
  }

rpc GetConfig
  {
   "Cisco-IOS-XR-ethernet-lldp-cfg:lldp": [
    "lldp": "running-configuration"
    ]
  }</pre></td><td><pre>{
 "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
}
}</pre></td></tr>
</table>

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.

Procedure

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
```

gRPC over UNIX Domain Sockets

Table 2: 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 /var/lib/docker/ems/grpc.sock directory.

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

Procedure

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 =
"/var/lib/docker/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 3: 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
      >
    >
```

Example: gNMI Get Request with Unique Path to a CLI

The following is a sample `Get` request to fetch the system updates through CLI.

```
path: <
  origin: "cli"
  elem: <
    name: "show version"
  >
>
type: ALL
encoding: ASCII
```

The following is a sample `Get` response.

```
path: <
  origin: "cli"
  elem: <
    name: "show version"
  >
>

type: ALL
```

```
...
...

[
  {
    "source": "unix:///var/run/test_env.sock",
    "timestamp": 1730123328800447525,
    "time": "2024-10-28T06:48:48.800447525-07:00",
    "updates": [
      {
        "Path": "show version",
        "values": {
          "show version":
"--------------------------------- show version ----------------------------------
Cisco IOS XR Software, Version 24.4.1.37I
Copyright (c) 2013-2024 by Cisco Systems, Inc.
Build Information:\n Built By     : swtools
Built On     : Mon Oct 21 03:16:32 PDT 2024
Built Host   : iox-lnx-121\n Workspace    :
/auto/iox-lnx-121-san2/prod/24.4.1.37I.SIT_IMAGE/ncs5500/ws
Version      : 24.4.1.37I\n Location     : /opt/cisco/XR/packages/
Label        : 24.4.1.37I-EFT2LabOnly
cisco NCS-5500 () processor
System uptime is 3 days 22 hours 54 minutes\n\n\n"
        }
      }
    ]
  }
]
```

gNMI Bundling of Telemetry Updates

Table 4: 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
  !
 !
!
```

Replace Router Configuration at Sub-tree Level Using gNMI

Table 5: Feature History Table

Feature Name	Release Information	Description
Replace Router Configuration at Sub-tree Level Using gNMI	Release 7.8.1	Using the gNMI `SetRequest` message, you can replace the router's existing configuration with a new set of configurations at the subtree level within the same model. Earlier you could replace router configurations at the data tree root level. To view the specification of gNMI replace, see Github repository.

The gNMI replace feature replaces the existing configuration on the router with the new configuration using a `SetRequest` RPC message. It allows you to specify a `path` (a structured format for path elements, and any associated key values) as the root prompt to perform a `replace` operation. Cisco IOS XR software Release 7.8.1 supports subtree-level replace operation. Prior to this release replace operation was performed at datatree-level.

Replace operation either includes all the path elements which are defined under the root or only few of them. If the omitted path elements are configured with default values, they are reverted to its default values during the replace operation. If the omitted path elements are not configured with default values, they are deleted from the data tree during the replace operation, and returned to its original unconfigured state. Consider the following example:

In the following data tree schema, `b` has a default value of `true` and `c` has no default value. Both `b` and `c` are set as `False`.

```
root +
     |
     + a --+
     |     |
     |     +-- b
     |     |
     |     +-- c
     |
     |
     + d --+
           +-- e
           |
           +-- f
```

When a `replace` operation is performed with `e` and `f` as set, and all other elements are omitted, `b` is reverted to its default setting true, and `c` is deleted from the tree, and returned to its original unconfigured state.

Following example shows the `SetRequest` and `SetResponse` of gNMI replace operation.

gNMI Replace Example

This example shows the gNMI replace request and response messages.

```
Request Message:
replace: <
  path: <
    elem: <
      name: "system"
    >
    elem: <
      name: "config"
    >
    elem: <
      name: "hostname"
    >
  >
  val: <
    json_ietf_val: "\"testing123\""
  >
>

Response Message:
  path: <
    elem: <
      name: "system"
    >
    elem: <
      name: "config"
    >
    elem: <
      name: "hostname"
    >
  >
  op: REPLACE
>
message: <
>
timestamp: 1662873319202107537
```

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 6: Feature History Table

Feature Name	Release Information	Description
gNOI Cert Proto to Load CA Bundle	Release 7.9.1	The RPCs defined in the proto file supports loading a Certificate Authority (CA) bundle to validate a peer's certificate.
gNOI OS Proto	Release 7.9.1	The RPCs defined in the proto file can be used to install the software, activate the software version and verify that the installation is successful.
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.

RPC	Description
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.

RPC	Description
CanGenerateCSR	Asks a target if the certificate can be generated.
LoadCertificateAuthorityBundle	Loads a bundle of CA certificates on the target. This CA certificate bundle is used to verify the client certificate when mutual TLS is enabled.

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.

Feature Name	Description
GetBERTResult	Gets the BERT results during the BERT or after the operation is complete.

Operating System (OS) RPCs

The OS service provides an interface for the OS installation on a target device. The RPCs replace the router software to upgrade the system. No concurrent installation is allowed on the same target. The **os.proto** file is available in the Github repository.

Feature Name	Description
Install	Transfers an OS package onto the target. **Note** Only Golden ISO installation is supported; RPM installation is not supported.
Activate	Sets the requested OS version as the version that is used at the next reboot. If booting up the requested OS version fails, the system recovers by rolling back to the previously running OS package.
Verify	Verifies the running OS version. The following gNOI OS verify information returns based on the install state: • If `success`, verify returns the installed version. • If `failure`, verify the version returned by install and set the activation_fail_message to the error returned by the install. • If `in-progress`, verify returns version returned by install and set the activation_fail_message to `in-progress`. • If the install state was not retrieved, verify that the version returned is `unknown` and set the activaiton_fail_message to `Failed to verify the current version`.

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$
```

gRIBI

Table 7: Feature History Table

Feature Name	Release Information	Description
Traffic engineering using gRIBI RPCs	Release 7.9.1	gRPC Routing Information Base Interface (gRIBI) is a gRPC service that allows an external client to programatically manage the routes in the Routing Information Base (RIB) of the router. With gRIBI RPCs, you can customize traffic engineering decisions based on individual requirement irrespective of the interdependency with the routes configured using other routing protocols. This feature introduces the **show gribi aft** command.

You can use the client application to add, edit or remove the routing entries in the routing table. The client can be local to the router or hosted externally in the network management station.

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

The OpenConfig Abstract Forwarding Table (`openconfig-aft.yang`) data model defines a common abstraction of the RIB information, and describes the forwarding entries installed on a network element. The AFT definitions are auto-generated from the OpenConfig AFT YANG schema. The protocol buffer (protobuf) representation of the OpenConfig AFT schema is available as gribi_aft.proto file in the Github repository. gRIBI leverages this data model and the proto file to manage the RIB entries. This data model supports streaming Event-driven telemetry (EDT) data to check the installed routes in the Forwarding Information Base (FIB).

The routes configured using static configuration have the highest preference, followed by routes configured using gRIBI, and then those configured using other protocols such as BGP or ISIS.

gRIBI RPCs

gRIBI supports the following remote procedure calls (RPCs) to manage the routes in the RIB:

- **Modify Operation:** Provides a bidirectional streaming RPC that is used to issue modifications to the AFT in the form of a `ModifyRequest` message. The network element responds asynchronously with a `ModifyResponse` message based on each request.

  Messages:

  Supports route modifications on IPv4Entry, next hop group (NHG), next hop (NH) key objects. The traffic engineering controller ensures that specific ordering of gRIBI transactions is met—NH and NHG entries are sent before IPv4Entries. The NHGs and NHs are sent in a single `ModifyRequest` as repeated AFTOperation messages. The controller expects that the NHG or NH transactions are acknowledged before programming the corresponding IPv4Entry transactions. In the next hop entries, `decapsulate_header`, `encapsulate_header`, `interface_ref`, `ip_address`, `ip_in_ip` and `network_instance` attributes are supported.

Note IP forwarding, encapsulation and decapsulation operations are supported. MPLS operations are not supported.

  SessionParameters:

  - For client redundancy, only `SINGLE_PRIMARY` is supported. The primary client is designated based on the client with the highest election ID. Each AFTOperation carries an election ID. The server processes the AFTOperation if the election ID is the last advertised ID and is the highest ID on the server. If the election ID is less than the current election ID, the ID is ignored. If the election ID is equal, the client sending the message is accepted as a new master.

    For client persistence, only the `PRESERVE` option is supported, wherein the network device preserves the routes programmed by the gRIBI server's RIB, the system RIB, and the system FIB when the primary client disconnects.

    When gRIBI restarts, the configuration in the gRIBI server's cache are sent to the router to reprogram the RIB. The election ID is reset to 0 upon restart.

  The following table shows the request and response exchanged between the client and server for the `Modify` RPCs:

Operation	Request (Client to Server)	Response (Server to Client)

Session setup	`Message ModifyResponse {`
	` SessionParameters`
	`}`
		`Message ModifyResponse{`
		` SessionParametersResult`
		`}`

Election ID	`Message Modify Request {`
	` election_id (int128)`
	`}`
		`Message Modify Request {`
		` election_id (int128)`
		`}`

Operation	Request (Client to Server)	Response (Server to Client)

AFTOperation	`Message ModifyResponse {`
	` repeated AFTOperation operation`
	`}`
		`Message ModifyResponse{`
		`  repeated AFTResult result`
		`}`

- **Get Operation**: Retrieves the content of the installed AFTs from the gRIBI daemon. The client requests for information using a `GetRequest` message, and the server responds with the set of currently installed entries via the `GetResponse` message. Once all entries have been sent, the server closes the RPC.

  Supports all operations defined in the `gribi.proto` file.

Note IPv4Entry.metdata is supported only in `Get` RPC, and not in AFT telemetry.

The following table shows the request and response exchanged between the client and server for the `Get` RPCs:

Operation	Request (Client to Server)	Response (Server to Client)

Get entries	`Message GetRequest {`
	` network_instance`
	`  name [string]`
	`  all`
	` AFTType aft`
	`}`
		`Message GetResponse {`
		` repeated AFTEntry entry`
		`}`

- **Flush Operation**: Removes all AFT entries that are currently installed on the server using gRIBI RPCs. The client sends a `FlushRequest` message to remove all the entries, and the server responds with a `FlushResponse` message after the operation is complete.

  Supports all operations defined in the `gribi.proto` file.

  The following table shows the request and response exchanged between the client and server for the `Flush` RPCs:

Operation	Request (Client to Server)	Response (Server to Client)

Flush entries	`Message FlushRequest {`
	` election`
	`  id [uint 128]`
	`  override [bool]`
	` network_instance`
	`  name [string]`
	`  all`
	`}`
		`Message FlushResponse {`
		` timestamp [int64]`
		`}`

Configuring gRIBI to Modify Routing Entries

Configure gRIBI protocol to directly interact with the routers' RIB using RPCs. The gRIBI client sends messages to the RIB to add a route, delete a route, register next hop and next hop groups to manage the routes.

Before you begin

Ensure that you have configured the gRIBI client application.

Procedure

Step 1 Enable gRPC protocol on the router.

Example:

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

Step 2 Configure the port number and address family.

Example:

```
Router(config-grpc)#port 57345
Router(config-grpc)#address-family ipv4
```

The port number ranges from 57344 to 57999. The default port is 57400. If a port number is unavailable, an error is displayed.

Step 3 Verify that gRPC is enabled on the router.

Example:

```
Router#show grpc
Thu Feb 2 22:03:17.004 UTC

Address family                : dual
Port                          : 57777
DSCP                          : Default
TTL                           : 64
VRF                           : global-vrf
Server                        : enabled
TLS                           : enabled
TLS mutual                    : disabled
Trustpoint                    : none
Certificate Authentication    : disabled
TLS v1.0                      : enabled
Maximum requests              : 128
Maximum requests per user     : 10
Maximum streams               : 32
Maximum streams per user      : 32

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

  Operational enable          : ecdhe-rsa-chacha20-poly1305
                              : ecdhe-ecdsa-chacha20-poly1305
                              : ecdhe-rsa-aes128-gcm-sha256
                              : ecdhe-ecdsa-aes128-gcm-sha256
                              : ecdhe-rsa-aes256-gcm-sha384
                              : ecdhe-ecdsa-aes256-gcm-sha384
                              : ecdhe-rsa-aes128-sha
                              : ecdhe-ecdsa-aes128-sha
                              : ecdhe-rsa-aes256-sha
                              : ecdhe-ecdsa-aes256-sha
                              : aes128-gcm-sha256
                              : aes256-gcm-sha384
```

```
                            : aes128-sha
                            : aes256-sha
  Operational disable       : none

Listen address suites
  Listen to Address         : ANY
```

Step 4 Manage the routing entries using gRIBI RPCs. In this example, you use the `Modify` RPC to add a next hop entry with IP address 192.0.2.0.

a) Configure the next hop parameters for the AFT message.

Example:

```
NextHop {
  ip_address 192.0.2.0;
  InterfaceRef {
    interface [string]
    subinterface [uint]
  }
  IPnIP {
    dst_ip [string]
    src_ip [string]
  }
}
```

b) Set the next hop entry in the AFTOperation. In this example, you add the next hop IP address.

Example:

```
Message AFTOperation {
 id
 network_instance
 Operation op
  ADD
 entry
  Afts.NextHopKey next_hop
}
```

c) Initiate the `ModifyRequest` RPC using the AFTOperation message.

Example:

```
Message ModifyRequest {
 repeated AFTOperation operation
}
```

The NHs are sent in a single `ModifyRequest` RPC as repeated AFTOperation messages.

d) View that the request is acknowledged in the gRIBI client.

Example:

```
gRIBIClient sent Modify message operation:{id:1 network_instance:"DEFAULT"
op:ADD next_hop:{index:1000 next_hop:{ip_address:{value:"192.0.2.0"}}} election_id:{low:3}}
```

Step 5 Verify the configuration performed using gRIBI RPC.

Example:

In this example, you verify the next hop IP address that you sent to the server through the `Modify` RPC is configured successfully.

```
Router#show gribi aft next-hop-groups
Thu Feb  2 00:34:08.104 UTC
100, Backup NHG: 1111
  [100, 2]: 192.0.2.40
```

```
  [200, 2]: 192.0.2.42
  [1111, 100]: (vrf REPAIR) (!)
1000
  [1100, 30]: 192.0.2.10
  [1200, 10]: 192.0.2.14
  [1000, 60]: 192.0.2.0
1111
  [1111, 100]: (vrf REPAIR)
2000
  [2000, 50]: 192.0.2.18
  [2100, 50]: 192.0.2.22
3000
  [3000, 10]: 192.0.2.26
4000
  [4000, 10]: Decapsulate IPv4(vrf DEFAULT)
```

Configure Interfaces Using Data Models in a gRPC Session

Table 8: Feature History Table

Feature Name	Release Information	Description
Disable TLS 1.0 to avoid security risks	Release 7.9.1	Although Transport Layer Security (TLS) protocol provides secure communication between servers and clients, TLS 1.0 may pose a security threat. This version is deprecated as per RFC 8996. You can now disable TLS 1.0 using the **tlsv1-disable** command. Use TLS 1.2 or later versions.

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.

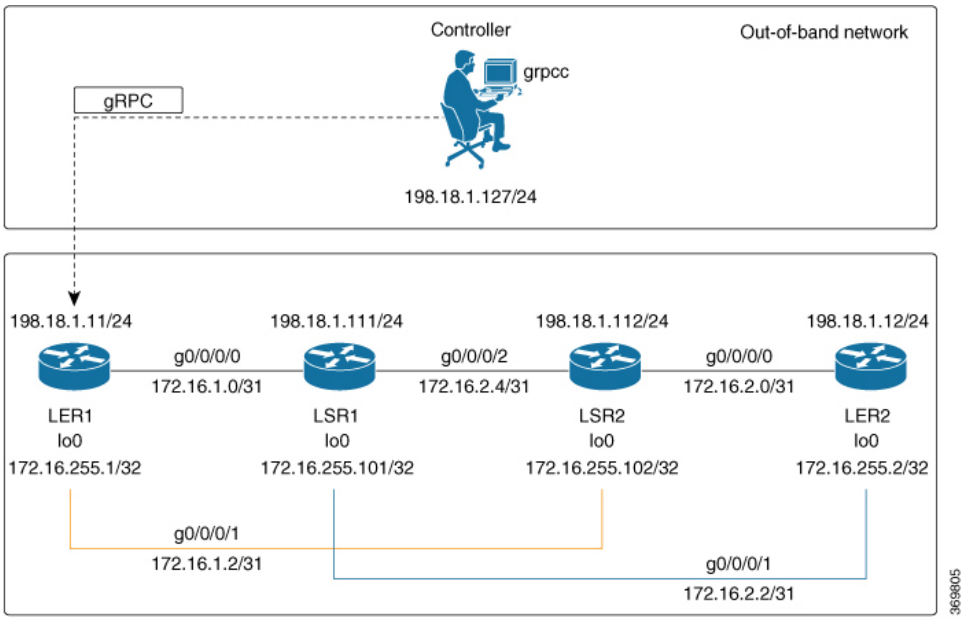
Note Configure AAA authorization to restrict users from uncontrolled access. If AAA authorization is not configured, the command and data rules associated to the groups that are assigned to the user are bypassed. An IOS-XR user can have full read-write access to the IOS-XR configuration through Network 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.

Procedure

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.

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-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.