gRPC Agent

- gRPC Agent, on page 1
- Ephemeral Data, on page 17

gRPC Agent

About the gRPC Agent

The Cisco NX-OS gRPC protocol defines a mechanism through which a network device can be managed and its configuration data can be retrieved and installed. The protocol exposes a complete and formal Application Programming Interface (API) that clients can use to manage device configurations.

The Cisco NX-OS gRPC protocol uses a remote procedure call (RPC) paradigm where an external client manipulates device configurations utilizing Google Protocol Buffer (GPB)-defined API calls along with their service- specific arguments. These GPB-defined APIs transparently cause an RPC call to the device that return replies in the same GPB-defined API context.

The gRPC Agent provides a secure transport through TLS and user authentication and authorization through AAA.

The functional objective of the Cisco NX-OS gRPC protocol is to mirror that provided by NETCONF, particularly in terms of both stateless and stateful configuration manipulation for maximum operational flexibility.

The Cisco NX-OS gRPC Agent supports the following protocol operations:

- Get
- GetConfig
- GetOper
- EditConfig
- StartSession
- CloseSession
- KillSession

The gRPC Agent supports two types of operations:

- **Stateless operations** are performed entirely within a single message without creating a session.
- **Stateful operations** are performed using multiple messages. The following is the sequence of operations that are performed:
  1. Start the session. This action acquires a unique session ID.
  2. Perform session tasks using the session ID.
  3. Close the session. This action invalidates the session ID.

The following are the supported operations. See the Appendix for their RPC definitions in the **.proto** file that is exported by the gRPC Agent.

Operation	Description
StartSession	Starts a new session between the client and server and acquires a unique session ID.
EditConfig	Writes the specified YANG data subset to the target datastore.
GetConfig	Retrieves the specified YANG configuration data subset from the source datastore.
GetOper	Retrieves the specified YANG operational data from the source datastore.
Get	Retrieves the specified YANG configuration and operational data from the source datastore.
KillSession	Forces the termination of a session.
CloseSession	Requests graceful termination of a session.

GetConfig, GetOper, and Get are stateless operations so don’t require a session ID.

EditConfig can be either stateless or stateful. For a stateless operation, specify the SessionID as 0. For a stateful operation, a valid (nonzero) SessionID is required.

The gRPC Agent supports timeout for the sessions. The idle timeout for sessions can be configured on the device, after which idle sessions are closed and deleted.

Guidelines and Limitations for gRPC

The gRPC Agent has the following guideline and limitation:

- Beginning with Cisco NX-OS Release 9.3(3), if you have configured a custom gRPC certificate, upon entering the **reload ascii** command the configuration is lost. It will revert to the default day-1 certificate. After entering the **reload ascii** command, the switch will reload. Once the switch is up again, you need to reconfigure the gRPC custom certificate.
- gRPC does not support enhanced Role-Based Access Control (RBAC) as specified in RFC 6536. Only users with a "network-admin" role are granted access to the gRPC agent.
- gRPC traffic destined for a Nexus device will hit the control-plane policer (CoPP) in the default class. To limit the possibility of gRPC drops, configure a custom CoPP policy using the gRPC configured port in the management class.

- Beginning with NX-OS 9.3(1), gRPC `Get` and `GetConfig` requests from the gRPC client to the switch must contain an explicit namespace and filter. This requirement affects requests to the OpenConfig YANG and Device models. If you see a message that is similar to the following, the requests are not carrying a namespace:

  ```
  Request without namespace and filter is an unsupported operation
  ```

  The following example shows a `Get` request and response with the behavior before this change. This example shows the error message that is caused by behavior which is no longer supported.

  Request:

  ```
  client-host % cat payload.json {
  }
  ```

  Response:

  ```
  {"rpc-reply":
      {"rpc-error":{ … Request without filtering is an unsupported operation" …}}
  }
  ```

  The following example shows a `Get` request and response with the correct behavior in NX-OS release 9.3(1) and later.

  Request:

  ```
  client-host % cat payload.json {
      "namespace": "http://cisco.com/ns/yang/cisco-nx-os-device",
      "System": {}
  }
  ```

  Response:

  ```
  {"rpc-reply":
      {"data":{"System": ...}}
  }
  ```

Configuring the gRPC Agent for Cisco NX-OS Release 9.3(3) and Later

The gRPC Agent supports the following configuration commands:

Parameter	Description
grpc port	Configure the port number. The range of *port-id* is from 1024 to 65535. 50051 is the default.

Parameter	Description
grpc certificate	Specify the certificate trustpoint ID. For more information, see the Installing Identity Certificates section of the Cisco Nexus 9000 Series NX-OS Security Configuration Guide, Release 9.3(x) for importing the certificate to the switch. **Note** This command is available beginning with Cisco NX-OS Release 9.3(3). **Note** When the certificate is invalid and after five consecutive failed attempts, the server is put on hold with the following syslog: `2020 May 17 00:35:44 n9k %NXSDK-4-WARNING_MSG: grpc (5459) Reached maximal mgt server retry. Verify certificate and port config are valid.`

The gRPC Agent supports the following configuration parameters under the **[grpc]** section in the configuration file (**/etc/mtx.conf**).

Parameter	Description
idle_timeout	(Optional) Specifies the timeout in minutes after which idle client sessions are disconnected. The default timeout is 5 minutes. A value of 0 disables timeout.
limit	(Optional) Specifies the number of maximum simultaneous client sessions. The default limit is 5 sessions. The range is from 1 through 50.
security	Specifies the type of secure connection. Valid choices are: • **TLS** for TLS • **NONE** for an insecure connection

For the modified configuration file to take effect, you must either restart or reload the gRPC Agent using the following commands:

Action	Command
Restart the agent	**grpcctl restart** or **grpcctl stop** **grpcctl start**
Reload the configuration	**grpcctl reload**

About Certificates

For new switch deployments with Cisco NX-OS 9.3(1), the switch generates a one-day temporary certificate to allow you enough time to install your custom certificate. This temporary one-day certificate replaces the hard-coded default certificate that was generated in previous versions of Cisco NX-OS. The temporary one-day certificate gets regenerated when a switch reload or system switchover event occurs.

If your switch already has a custom certificate that is installed, for example, if you are upgrading from a previous NX-OS version to NX-OS 9.3(1), your existing certificate is retained and used after upgrade.

The following is an example of the **[grpc]** section in the configuration file:

```
[grpc]
mtxadapter=/opt/mtx/lib/libmtxadaptergrpc.1.0.1.so
idle_timeout=10
limit=1
lport=50051
security=TLS
cert=/etc/grpc.pem
key=/etc/grpc.key
```

Beginning with Cisco NX-OS Release 9.3(3) and later, **lport=50051** is replaced by the **grpc port** command. **cert=/etc/grpc.pem** and **key=/etc/grpc.key** are no longer needed and are replaced by the **grpc certficate** command.

Configuring gRPC

Configure the gRPC feature through the **grpc** commands.

To import certificates used by the **grpc certificate** command onto the switch, see the Installing Identity Certificates section of the Cisco Nexus 9000 Series NX-OS Security Configuration Guide, Release 9.3(x).

Note When modifying the installed identity certificates or **grpc port** and **grpc certificate** values, the gRPC server might restart to apply the changes. When the gRPC server restarts, any active subscription is dropped and you must resubscribe.

Procedure

	Command or Action	Purpose

Step 1	**configure terminal**
	Example:
	`switch# configure terminal`
	`switch-1(config)#`
		Enters global configuration mode.

Step 2	**feature grpc**
	Example:
	`switch# feature grpc`
	`switch-1(config)#`
		Enables the gRPC agent, which supports the gNMI interface for dial-in.

Step 3	**grpc port** *port-id* **Example:** `switch(config)# grpc port 50051`	Configure the port number. The range of *port-id* is from 1024 to 65535. 50051 is the default. **Note** This command is available beginning with Cisco NX-OS Release 9.3(3).
Step 4	**grpc certificate** *certificate-id* **Example:** `switch-1(config)# grpc certificate cert-1`	Specify the certificate trustpoint ID. For more information, see the Installing Identity Certificates section of the Cisco Nexus 9000 Series NX-OS Security Configuration Guide, Release 9.3(x) for importing the certificate to the switch. **Note** This command is available beginning with Cisco NX-OS Release 9.3(3).

Configuring the gRPC Agent for Cisco NX-OS Release 9.3(2) and Earlier

The gRPC Agent supports the following configuration parameters under the **[grpc]** section in the configuration file (**/etc/mtx.conf**).

Parameter	Description
idle_timeout	(Optional) Specifies the timeout in minutes after which idle client sessions are disconnected. The default timeout is 5 minutes. A value of 0 disables timeout.
limit	(Optional) Specifies the number of maximum simultaneous client sessions. The default limit is 5 sessions. The range is from 1 through 50.

Parameter	Description
lport	(Optional) Specifies the port number on which the gRPC Agent listens. The default port is 50051.
key	Specifies the key file location for TLS authentication. The default location is **/opt/mtx/etc/grpc.key**.
cert	Specifies the certificate file location for TLS authentication. The default location is **/opt/mtx/etc/grpc.pem**. Beginning with NX-OS release 9.3(1), some changes are made to the certificate for gRPC Agent. See "About Certificates" below.
security	Specifies the type of secure connection. Valid choices are: • **TLS** for TLS • **NONE** for an insecure connection

For the modified configuration file to take effect, you must either restart or reload the gRPC Agent using the following commands:

Action	Command
Restart the agent	**grpcctl restart** or **grpcctl stop** **grpcctl start**
Reload the configuration	**grpcctl reload**

About Certificates

For new switch deployments with Cisco NX-OS 9.3(1), the switch generates a one-day temporary certificate to allow you enough time to install your custom certificate. This temporary one-day certificate replaces the hard-coded default certificate that was generated in previous versions of Cisco NX-OS. The temporary one-day certificate gets regenerated when a switch reload or system switchover event occurs.

If your switch already has a custom certificate that is installed, for example, if you are upgrading from a previous NX-OS version to NX-OS 9.3(1), your existing certificate is retained and used after upgrade.

The following is an example of the **[grpc]** section in the configuration file:

```
[grpc]
mtxadapter=/opt/mtx/lib/libmtxadaptergrpc.1.0.1.so
idle_timeout=10
```

```
limit=1
lport=50051
security=TLS
cert=/etc/grpc.pem
key=/etc/grpc.key
```

Note For the following, if you have a hard-coded key, you must delete it and install your custom key in a persistent location.

Delete the Hard-Coded Certificate

If your switch was deployed with a version of NX-OS earlier than 9.3(1), you must delete the hard-coded certificate.

Change the Certificate Location

The previous default location for the certificate (`/etc/mtx.conf file`) is no longer persistent.

By default, the `/etc/mtx.conf.user` file is not present, so you will need to create it and specify the new location of the key and certificate so that the change is persistent. For example:

```
[grpc]
cert="new location"
key="new location"
```

Note For switches that have dual supervisor modules, you must upload the certificate files and modify the `/etc/mtx.conf.user` file on both the Active and Standby supervisors. If you do not do this on both supervisors, the certificate files are not persistent after a switchover.

Replace an Expired Certificate

If your temporary, one-day certificate is expired, delete it and replace it with a new certificate.

Reload the Configuration File

Use the appropriate option, depending on whether you want to reload the switch:

- If you are restarting the gRPC agent to use the new certificate without reloading the switch:
  1. Add the certificate and key to `/etc/mtx.conf`.
  2. For the modified configuration file to take effect, issue **no feature grpc** to disable the gRPC Agent, then **feature grpc** to reenable it.
- If you want to reload the switch, the changes to `/etc/mtx.conf.user` are sufficient. You do not need to modify `/etc/mtx.conf`.

Using the gRPC Agent

General Commands

You can enable or disable the gRPC Agent by issuing the [**no**] **feature grpc** command.

General Commands

The available control commands for the gRPC agent are:

grpcctl { status | start | restart | reload | stop }

Viewing the Agent Status

```
bash-4.2# grpcctl status
xosdsd is stopped
grpcctl is stopped
```

Starting the Agent

```
bash-4.2# grpcctl start
Starting Grpc Agent: [OK]
```

Example: A Basic Yang Path in JSON Format

```
client-host % cat payload.json
{
  "namespace": "http://cisco.com/ns/yang/cisco-nx-os-device",
  "System": {
    "bgp-items": {
      "inst-items": {
        "dom-items": {
          "Dom-list": {
            "name": "default",
            "rtrId": "7.7.7.7",
            "holdIntvl": "100"
          }
        }
      }
    }
  }
}
```

Note The JSON structure has been pretty-formatted here for readability.

Sending an EditConfig Request to the Server

```
client-host % ./grpc_client -username=admin -password=cisco -operation=EditConfig
-e_oper=Merge -def_op=Merge -err_op=stop-on-error -infile=payload.json -reqid=1
-source=running -tls=true -serverAdd=192.0.20.123 -lport=50051

#######################################################
Starting the client service
#######################################################
TLS set true for client requests1ems.cisco.com
TLS FLAG:1
192.0.20.123:50051
All the client connections are secured
```

```
Sending EditConfig request to the server
sessionid is
0
reqid:1
{"rpc-reply":{"ok":""}}
```

Sending a GetConfig Request to the Server

```
client-host % ./grpc_client -username=admin -password=cisco -operation=GetConfig
-infile=payload.json -reqid=1 -source=running -tls=true -serverAdd=192.0.20.123 -lport=50051

#####################################################
Starting the client service
#####################################################
TLS set true for client requests1ems.cisco.com
TLS FLAG:1
192.0.20.123:50051
All the client connections are secured
Sending GetConfig request to the server
in get config
Got the response from the server
########################################
Yang Json is:
########################################
{"rpc-reply":{"data":{"System":{"bgp-items":{"inst-items":{"dom-items":{"Dom-list":{"name":"default","rtrId":"7.7.7.7","holdIntvl":"100"}}}}}}}}
########################################
client-host %
```

Troubleshooting the gRPC Agent

Troubleshooting Connectivity

- From a client system, verify that the agent is listening on the port. For example:

```
client-host % nc -z 192.0.20.222 50051
Connection to 192.0.20.222 50051 port [tcp/*] succeeded!
client-host % echo $?
0
client-host %
```

- In the NX-OS, check the gRPC agent status by issuing **show feature | grep grpc**.

gRPC Protobuf File

The gRPC Agent exports the supported operations and data structures in the proto definition file at `/opt/mtx/etc/nxos_grpc.proto`. The file is included in the gRPC Agent RPM. The following shows the definitions:

```
// Copyright 2016, Cisco Systems Inc.
// All rights reserved.

syntax = "proto3";
```

```
package NXOSExtensibleManagabilityService;

// Service provided by Cisco NX-OS gRPC Agent
service gRPCConfigOper {

    // Retrieves the specified YANG configuration data subset from the
    // source datastore
    rpc GetConfig(GetConfigArgs) returns(stream GetConfigReply) {};

    // Retrieves the specified YANG operational data from the source datastore
    rpc GetOper(GetOperArgs) returns(stream GetOperReply) {};

    // Retrieves the specified YANG configuration and operational data
    // subset from the source datastore
    rpc Get(GetArgs) returns(stream GetReply){};

    // Writes the specified YANG data subset to the target datastore
    rpc EditConfig(EditConfigArgs) returns(EditConfigReply) {};

    // Starts a new session between the client and server and acquires a
    // unique session ID
    rpc StartSession(SessionArgs) returns(SessionReply) {};

    // Requests graceful termination of a session
    rpc CloseSession(CloseSessionArgs) returns (CloseSessionReply) {};

    // Forces the termination of a session
    rpc KillSession(KillArgs) returns(KillReply) {};

// Unsupported; reserved for future
    rpc DeleteConfig(DeleteConfigArgs) returns(DeleteConfigReply) {};

    // Unsupported; reserved for future
    rpc CopyConfig(CopyConfigArgs) returns(CopyConfigReply) {};

    // Unsupported; reserved for future
    rpc Lock(LockArgs) returns(LockReply) {};

    // Unsupported; reserved for future
    rpc UnLock(UnLockArgs) returns(UnLockReply) {};

    // Unsupported; reserved for future
    rpc Commit(CommitArgs) returns(CommitReply) {};

    // Unsupported; reserved for future
    rpc Validate(ValidateArgs) returns(ValidateReply) {};

    // Unsupported; reserved for future
    rpc Abort(AbortArgs) returns(AbortReply) {};

}

message GetConfigArgs
{
    // JSON-encoded YANG data to be retrieved
    string YangPath = 1;

    // (Optional) Specifies the request ID.  Default value is 0.
    int64 ReqID = 2;

    // (Optional) Specifies the source datastore; only "running" is supported.
    // Default is "running".
    string Source = 3;
}
```

```
message GetConfigReply
{
    // The request ID specified in the request.
    int64 ReqID = 1;

    // JSON-encoded YANG data that was retrieved
    string YangData = 2;

    // JSON-encoded error information when request fails
    string Errors = 3;
}

message GetOperArgs
{
    // JSON-encoded YANG data to be retrieved
    string YangPath = 1;

    // (Optional) Specifies the request ID.  Default value is 0.
    int64 ReqID = 2;
}

message GetOperReply
{
    // The request ID specified in the request.
    int64 ReqID = 1;

    // JSON-encoded YANG data that was retrieved
    string YangData = 2;

    // JSON-encoded error information when request fails
    string Errors = 3;
}

message GetArgs
{
    // JSON-encoded YANG data to be retrieved
    string YangPath=1;

    // (Optional) Specifies the request ID.  Default value is 0.
    int64 ReqID = 2;
}

message GetReply
{
    // The request ID specified in the request.
    int64 ReqID = 1;

    // JSON-encoded YANG data that was retrieved
    string YangData = 2;

    // JSON-encoded error information when request fails
    string Errors = 3;
}

message EditConfigArgs
{
    // JSON-encoded YANG data to be edited
    string YangPath = 1;

    // Specifies the operation to perform on teh configuration datastore with
    // the YangPath data.  Possible values are:
    //   create
    //   merge
```

```
    //   replace
    //   delete
    //   remove
    // If not specified, default value is "merge".
    string Operation = 2;

    // A unique session ID acquired from a call to StartSession().
    // For stateless operation, this value should be set to 0.
    int64 SessionID = 3;

    // (Optional) Specifies the request ID.  Default value is 0.
    int64 ReqID = 4;

    // (Optional) Specifies the target datastore; only "running" is supported.
    // Default is "running".
    string Target = 5;

    // Specifies the default operation on the given object while traversing
    // the configuration tree.
    // The following operations are possible:
    //   merge:     merges the configuration data with the target datastore;
    //              this is the default.
    //   replace:   replaces the configuration data with the target datastore.
    //   none:      target datastore is unaffected during the traversal until
    //              the specified object is reached.
    string DefOp = 6;

    // Specifies the action to be performed in the event of an error during
    // configuration.  Possible values are:
    //   stop
    //   roll-back
    //   continue
    // Default is "roll-back".
    string ErrorOp = 7;
}

message EditConfigReply
{
    // The request ID specified in the request.
    int64 ReqID = 1;

    // If EditConfig is successful, YangData contains a JSON-encoded "ok" response.
    string YangData = 2;

    // JSON-encoded error information when request fails
    string Errors = 3;
}

message DeleteConfigArgs
{
    // A unique session ID acquired from a call to StartSession().
    // For stateless operation, this value should be set to 0.
    int64 SessionID = 1;

    // (Optional) Specifies the request ID.  Default value is 0.
    int64 ReqID = 2;

    // (Optional) Specifies the target datastore; only "running" is supported.
    // Default is "running".
    string Target = 3;
}

message DeleteConfigReply
{
```

```
    // The request ID specified in the request.
    int64 ReqID = 1;

    // If DeleteConfig is successful, YangData contains a JSON-encoded "ok" response.
    string YangData = 2;

    // JSON-encoded error information when request fails
    string Errors = 3;
}

message CopyConfigArgs
{
    // A unique session ID acquired from a call to StartSession().
    // For stateless operation, this value should be set to 0.
    int64 SessionID = 1;

    // (Optional) Specifies the request ID.  Default value is 0.
    int64 ReqID = 2;

    // (Optional) Specifies the source datastore; only "running" is supported.
    // Default is "running".
    string Source = 3;

    // (Optional) Specifies the target datastore; only "running" is supported.
    // Default is "running".
    string Target = 4;
}

message CopyConfigReply
{
    // The request ID specified in the request.
    int64 ReqID = 1;

    // If CopyConfig is successful, YangData contains a JSON-encoded "ok" response.
    string YangData = 2;

    // JSON-encoded error information when request fails
    string Errors = 3;
}

message LockArgs
{
    // A unique session ID acquired from a call to StartSession().
    int64 SessionID = 1;

    // (Optional) Specifies the request ID.  Default value is 0.
    int64 ReqID=2;

    // (Optional) Specifies the target datastore; only "running" is supported.
    // Default is "running".
    string Target = 3;
}

message LockReply
{
    // The request ID specified in the request.
    int64 ReqID = 1;

    // If Lock is successful, YangData contains a JSON-encoded "ok" response.
    string YangData = 2;

    // JSON-encoded error information when request fails
    string Errors = 3;
}
```

```
message UnLockArgs
{
    // A unique session ID acquired from a call to StartSession().
    int64 SessionID = 1;

    // (Optional) Specifies the request ID.  Default value is 0.
    int64 ReqID = 2;

    // (Optional) Specifies the target datastore; only "running" is supported.
    // Default is "running".
    string Target = 3;
}

message UnLockReply
{
    // The request ID specified in the request.
    int64 ReqID = 1;

    // If UnLock is successful, YangData contains a JSON-encoded "ok" response.
    string YangData = 2;

    // JSON-encoded error information when request fails
    string Errors = 3;
}

message SessionArgs
{
    // (Optional) Specifies the request ID.  Default value is 0.
    int64 ReqID = 1;
}

message SessionReply
{
    // The request ID specified in the request.
    int64 ReqID = 1;
    int64 SessionID = 2;

    // JSON-encoded error information when request fails
    string Errors = 3;
}

message CloseSessionArgs
{
    // (Optional) Specifies the request ID.  Default value is 0.
    int64 ReqID = 1;

    // A unique session ID acquired from a call to StartSession().
    int64 SessionID = 2;
}

message CloseSessionReply
{
    // The request ID specified in the request.
    int64 ReqID = 1;

    // If CloseSession is successful, YangData contains a JSON-encoded "ok" response.
    string YangData = 2;

    // JSON-encoded error information when request fails
    string Errors = 3;
}

message KillArgs
```

```
{
    // A unique session ID acquired from a call to StartSession().
    int64 SessionID = 1;

    int64 SessionIDToKill = 2;

    // (Optional) Specifies the request ID.  Default value is 0.
    int64 ReqID = 3;
}

message KillReply
{
    // The request ID specified in the request.
    int64 ReqID = 1;

    // If Kill is successful, YangData contains a JSON-encoded "ok" response.
    string YangData = 2;

    // JSON-encoded error information when request fails
    string Errors = 3;
}

message ValidateArgs
{
    // A unique session ID acquired from a call to StartSession().
    int64 SessionID = 1;

    // (Optional) Specifies the request ID.  Default value is 0.
    int64 ReqID = 2;
}

message ValidateReply
{
    // The request ID specified in the request.
    int64 ReqID = 1;

    // If Validate is successful, YangData contains a JSON-encoded "ok" response.
    string YangData = 2;

    // JSON-encoded error information when request fails
    string Errors = 3;
}

message CommitArgs
{
    // A unique session ID acquired from a call to StartSession().
    int64 SessionID = 1;

    // (Optional) Specifies the request ID.  Default value is 0.
    int64 ReqID = 2;
}

message CommitReply
{
    // (Optional) Specifies the request ID.  Default value is 0.
    int64 ReqID = 1;

    // If Commit is successful, YangData contains a JSON-encoded "ok" response.
    string YangData = 2;

    // JSON-encoded error information when request fails
    string Errors = 3;
}
```

```
message AbortArgs
{
    // A unique session ID acquired from a call to StartSession().
    int64 SessionID = 1;

    // (Optional) Specifies the request ID.  Default value is 0.
    int64 ReqID = 2;
}

message AbortReply
{
    // (Optional) Specifies the request ID.  Default value is 0.
    int64 ReqID = 1;

    // If Abort is successful, YangData contains a JSON-encoded "ok" response.
    string YangData = 2;

    // JSON-encoded error information when request fails
    string Errors = 3;
}
```

Ephemeral Data

About Ephemeral Data in gRPC

Beginning with Cisco NX-OS Release 9.3(3), this feature provides access to ephemeral data. Ephemeral data is high volume data. DME provides a batching mechanism to retrieve the data so that each batch is of a manageable size in term of memory usage. The size of the batch is the number of MOs to be retrieved.

The batching mechanism is not exposed via gRPC. Batching is handled internally. You do not use gRPC batching explicitly. You can use batching while trying to access directly from DME via REST API.

You can find information about which data is ephemeral by the comment "Ephemeral data" in the published YANG file.

The output from ephemeral data is returned, if and only if, the filter in the request points to:

- A leaf from ephemeral data
- A container or list with ephemeral data children
- An empty container that is used to wrap a list that had direct ephemeral data children

System level GET queries do not return ephemeral data.

gRPC Ephemeral Data Example

This is a gRPC example of retrieving ephemeral data.

```
ayyim-lnx.cisco.com:200> cat urib.mgmt.json
{"namespace":"http://cisco.com/ns/yang/cisco-nx-os-device",
    "System": {
        "urib-items":{
            "table4-items":{
                    "Table4-list":{
         "vrfName":"management",
```

```
          "route4-items":{
             "Route4-list":{
             }
          }
                     }
             }
          }
       }
}

ayyim-lnx.cisco.com:395> ./grpc_client -username=admin -password=C\!sco123 -operation=Get
-reqid=1 -serverAdd=172.23.167.216 -lport=50051 -tls=true -infile=payloads/urib.mgmt.json
-cafile=grpc.pem.n9kv-blade2

####################################################
Starting the client service
####################################################
TLS set true for client requests1ems.cisco.com
TLS FLAG:1
172.23.167.216:50051
All the client connections are secured
Sending Get request to the server
sending get request
Got the response from the server
########################################
Yang Json is:
########################################
{
   "rpc-reply" : {
      "data" : {
         "System" : {
            "xmlns" : "http://cisco.com/ns/yang/cisco-nx-os-device",
            "urib-items" : {
               "table4-items" : {
                  "Table4-list" : [
                     {
                        "vrfName" : "management",
                        "route4-items" : {
                           "Route4-list" : [
                              {
                                 "prefix" : "172.23.167.255/32",
                                 "flags" : "0",
                                 "mBestNextHopCount" : "0",
                                 "nh4-items" : {
                                    "NextHop4-list" : [
                                       {
                                          "id" : "0",
                                          "address" : "172.23.167.255",
                                          "bindinglabel" : "0",
                                          "encapType" : "none",
                                          "interfaceName" : "mgmt0",
                                          "isBest" : "true",
                                          "metric" : "0",
                                          "owner" : "broadcast",
                                          "preference" : "0",
                                          "routeType" : "unknown",
                                          "segidType" : "null",
                                          "segmentId" : "0",
                                          "tag" : "0",
                                          "tunnelId" : "0",
                                          "uptime" : "00:18:18",
                                          "vrf" : "management"
                                       }
                                    ]
```

```
      },
      "pendingHw" : "false",
      "pendingUfdm" : "false",
      "sortKey" : "0",
      "uBestNextHopCount" : "1"
   },
   {
      "prefix" : "172.23.167.216/32",
      "flags" : "0",
      "mBestNextHopCount" : "0",
      "nh4-items" : {
         "NextHop4-list" : [
            {
               "id" : "0",
               "address" : "172.23.167.216",
               "bindinglabel" : "0",
               "encapType" : "none",
               "interfaceName" : "mgmt0",
               "isBest" : "true",
               "metric" : "0",
               "owner" : "local",
               "preference" : "0",
               "routeType" : "unknown",
               "segidType" : "null",
               "segmentId" : "0",
               "tag" : "0",
               "tunnelId" : "0",
               "uptime" : "00:18:18",
               "vrf" : "management"
            }
         ]
      },
      "pendingHw" : "false",
      "pendingUfdm" : "false",
      "sortKey" : "0",
      "uBestNextHopCount" : "1"
   },
   {
      "prefix" : "172.23.167.20/32",
      "flags" : "0",
      "mBestNextHopCount" : "0",
      "nh4-items" : {
         "NextHop4-list" : [
            {
               "id" : "0",
               "address" : "172.23.167.20",
               "bindinglabel" : "0",
               "encapType" : "none",
               "interfaceName" : "mgmt0",
               "isBest" : "true",
               "metric" : "0",
               "owner" : "am",
               "preference" : "250",
               "routeType" : "unknown",
               "segidType" : "null",
               "segmentId" : "0",
               "tag" : "0",
               "tunnelId" : "0",
               "uptime" : "00:03:06",
               "vrf" : "management"
            }
         ]
      },
      "pendingHw" : "false",
```

```
      "pendingUfdm" : "false",
      "sortKey" : "0",
      "uBestNextHopCount" : "1"
   },
   {
      "prefix" : "172.23.167.8/32",
      "flags" : "0",
      "mBestNextHopCount" : "0",
      "nh4-items" : {
         "NextHop4-list" : [
            {
               "id" : "0",
               "address" : "172.23.167.8",
               "bindinglabel" : "0",
               "encapType" : "none",
               "interfaceName" : "mgmt0",
               "isBest" : "true",
               "metric" : "0",
               "owner" : "am",
               "preference" : "250",
               "routeType" : "unknown",
               "segidType" : "null",
               "segmentId" : "0",
               "tag" : "0",
               "tunnelId" : "0",
               "uptime" : "00:02:23",
               "vrf" : "management"
            }
         ]
      },
      "pendingHw" : "false",
      "pendingUfdm" : "false",
      "sortKey" : "0",
      "uBestNextHopCount" : "1"
   },
   {
      "prefix" : "0.0.0.0/0",
      "flags" : "0",
      "mBestNextHopCount" : "0",
      "nh4-items" : {
         "NextHop4-list" : [
            {
               "id" : "0",
               "address" : "172.23.167.1",
               "bindinglabel" : "0",
               "encapType" : "none",
               "interfaceName" : "N/A",
               "isBest" : "true",
               "metric" : "0",
               "owner" : "static",
               "preference" : "1",
               "routeType" : "",
               "segidType" : "null",
               "segmentId" : "0",
               "tag" : "0",
               "tunnelId" : "0",
               "uptime" : "00:18:18",
               "vrf" : "management"
            }
         ]
      },
      "pendingHw" : "false",
      "pendingUfdm" : "false",
      "sortKey" : "0",
```

```
   "uBestNextHopCount" : "1"
},
{
   "prefix" : "172.23.167.0/24",
   "flags" : "0",
   "mBestNextHopCount" : "0",
   "nh4-items" : {
      "NextHop4-list" : [
         {
            "id" : "0",
            "address" : "172.23.167.216",
            "bindinglabel" : "0",
            "encapType" : "none",
            "interfaceName" : "mgmt0",
            "isBest" : "true",
            "metric" : "0",
            "owner" : "direct",
            "preference" : "0",
            "routeType" : "",
            "segidType" : "null",
            "segmentId" : "0",
            "tag" : "0",
            "tunnelId" : "0",
            "uptime" : "00:18:18",
            "vrf" : "management"
         }
      ]
   },
   "pendingHw" : "false",
   "pendingUfdm" : "false",
   "sortKey" : "0",
   "uBestNextHopCount" : "1"
},
{
   "prefix" : "172.23.167.1/32",
   "flags" : "0",
   "mBestNextHopCount" : "0",
   "nh4-items" : {
      "NextHop4-list" : [
         {
            "id" : "0",
            "address" : "172.23.167.1",
            "bindinglabel" : "0",
            "encapType" : "none",
            "interfaceName" : "mgmt0",
            "isBest" : "true",
            "metric" : "0",
            "owner" : "am",
            "preference" : "250",
            "routeType" : "unknown",
            "segidType" : "null",
            "segmentId" : "0",
            "tag" : "0",
            "tunnelId" : "0",
            "uptime" : "00:04:07",
            "vrf" : "management"
         }
      ]
   },
   "pendingHw" : "false",
   "pendingUfdm" : "false",
   "sortKey" : "0",
   "uBestNextHopCount" : "1"
},
```

```
                                    {
                                       "prefix" : "172.23.167.0/32",
                                       "flags" : "0",
                                       "mBestNextHopCount" : "0",
                                       "nh4-items" : {
                                          "NextHop4-list" : [
                                             {
                                                "id" : "0",
                                                "address" : "172.23.167.0",
                                                "bindinglabel" : "0",
                                                "encapType" : "none",
                                                "interfaceName" : "Null0",
                                                "isBest" : "true",
                                                "metric" : "0",
                                                "owner" : "broadcast",
                                                "preference" : "0",
                                                "routeType" : "unknown",
                                                "segidType" : "null",
                                                "segmentId" : "0",
                                                "tag" : "0",
                                                "tunnelId" : "0",
                                                "uptime" : "00:18:18",
                                                "vrf" : "management"
                                             }
                                          ]
                                       },
                                       "pendingHw" : "false",
                                       "pendingUfdm" : "false",
                                       "sortKey" : "0",
                                       "uBestNextHopCount" : "1"
                                    }
                                 ]
                              }
                           }
                        ]
                     }
                  }
               }
            }
         }
      }
   }
```