

Junos® OS Evolved

gRPC Network Services User Guide

Published
2023-12-15

Juniper Networks, Inc.
1133 Innovation Way
Sunnyvale, California 94089
USA
408-745-2000
www.juniper.net

Juniper Networks, the Juniper Networks logo, Juniper, and Junos are registered trademarks of Juniper Networks, Inc. in the United States and other countries. All other trademarks, service marks, registered marks, or registered service marks are the property of their respective owners.

Juniper Networks assumes no responsibility for any inaccuracies in this document. Juniper Networks reserves the right to change, modify, transfer, or otherwise revise this publication without notice.

Junos® OS Evolved gRPC Network Services User Guide
Copyright © 2023 Juniper Networks, Inc. All rights reserved.

The information in this document is current as of the date on the title page.

YEAR 2000 NOTICE

Juniper Networks hardware and software products are Year 2000 compliant. Junos OS has no known time-related limitations through the year 2038. However, the NTP application is known to have some difficulty in the year 2036.

END USER LICENSE AGREEMENT

The Juniper Networks product that is the subject of this technical documentation consists of (or is intended for use with) Juniper Networks software. Use of such software is subject to the terms and conditions of the End User License Agreement ("EULA") posted at <https://support.juniper.net/support/eula/>. By downloading, installing or using such software, you agree to the terms and conditions of that EULA.

Table of Contents

About This Guide | vi

1

gRPC Services Overview

Understanding gRPC Services for Managing Network Devices | 2

Benefits of gRPC Network Services | 2

Understanding OpenConfig | 2

gRPC-based Services Overview | 3

gNMI, gNOI, and gRIBI Overview | 3

2

Configure gRPC Services

Configure gRPC Services | 8

Understanding Authentication and Authorization for gRPC-based Services | 9

Obtain X.509 Certificates | 11

Load the gRPC Server's Local Certificate in the Junos PKI | 14

Enable gRPC Services | 15

Configure Mutual (Bidirectional) Authentication for gRPC Services | 17

Configure the User Account for gRPC Services | 21

Configure gRPC RPC Authorization | 22

Configure gNOI Services | 24

Set up the gRPC Client | 25

Compile the Proto Definition Files | 26

Create gNOI Applications | 28

Execute the Application | 32

3

gNOI Services

gNOI Services Overview | 34

Supported Services Overview | 34

CertificateManagement Service | 35

Diagnostic (Diag) Service | 37

File Service | 37

Operating System (OS) Service | 38

System Service | 39

gNOI Certificate Management Service | 41

Overview | 42

Supported RPCs | 45

Network Device Configuration | 46

Install a Certificate | 47

Rotate a Certificate | 57

Revoke a Certificate | 68

gNOI Diagnostic (Diag) Service | 73

Overview | 73

Supported RPCs | 75

Network Device Configuration | 75

Example: Run a BERT | 76

gNOI File Service | 86

Supported RPCs | 87

Network Device Configuration | 87

Example: Get File | 88

Example: Put File | 92

gNOI Operating System (OS) Service | 97

Supported RPCs | 98

Network Device Configuration | 98

Example: Install and Activate | 99

gNOI System Service | 105

Overview | 105

Network Device Configuration | 106

Ping and Traceroute | 106

Reboot a Device | 114

Upgrade Software | 120

Routing Engine Switchover | 125

4

gRIBI

gRIBI | 131

Supported RPCs | 132

Network Device Configuration | 132

Modify Routes | 135

Get Routes | 144

Flush Routes | 145

5

Configuration Statements and Operational Commands

Junos CLI Reference Overview | 147

About This Guide

Use this guide to remotely operate supported Junos devices using gRPC services, including gRPC Network Operations Interface (gNOI) services and gRPC Routing Information Base Interface (gRIBI) services.

1

CHAPTER

gRPC Services Overview

Understanding gRPC Services for Managing Network Devices | 2

Understanding gRPC Services for Managing Network Devices

SUMMARY

gRPC client applications can use the gRPC network services defined in the OpenConfig network interfaces (gNMI, gNOI, gRIBI) to manage supported network devices.

IN THIS SECTION

- [Benefits of gRPC Network Services | 2](#)
- [Understanding OpenConfig | 2](#)
- [gRPC-based Services Overview | 3](#)
- [gNMI, gNOI, and gRIBI Overview | 3](#)

Benefits of gRPC Network Services

- Provide simple, vendor agnostic interfaces for managing network devices.
- Enable you to more easily manage multivendor networks on a large scale.
- Use the gRPC Remote Procedure Call framework for transport and Protocol Buffers for service definitions and encoding, which provide efficient transport and fast processing.

Understanding OpenConfig

OpenConfig is a collaborative effort in the networking industry to move toward a more dynamic, programmable method for configuring and managing multivendor networks. OpenConfig supports the use of vendor-neutral service definitions and data models to manage and configure the network. The service definitions define common operations executed on network devices, and the data models define the configuration and operational state of network devices for common network protocols or services.

Operators with a multivendor network benefit greatly from using industry standard models and specifications. The goal of OpenConfig is for operators to be able to use a single set of data models and operations to configure and manage all network devices that support the OpenConfig initiative. The OpenConfig working group has developed specifications for gRPC-based interfaces for managing the configuration, operations, and telemetry streams on network devices, which provide advantages over other traditional network management protocols.

gRPC-based Services Overview

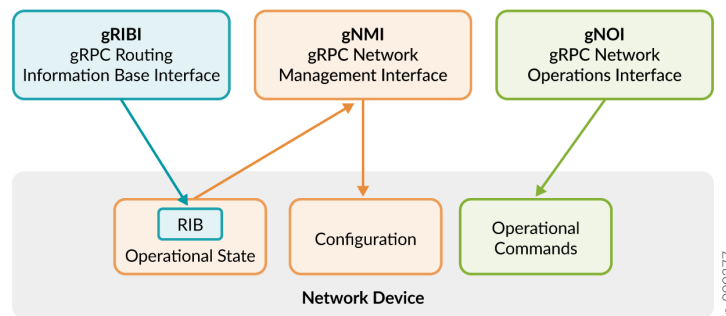
gRPC is an open source Remote Procedure Call (RPC) framework that was originally developed by Google. gRPC uses HTTP/2 for transport and supports modern security mechanisms and bidirectional streaming. gRPC uses the Protocol Buffers data format for defining services and encoding data. Protocol Buffers is language agnostic and supports bindings for many different languages, which enables operators to easily integrate gRPC-based services into existing management applications.

The OpenConfig working group has defined specifications for gRPC-based network management protocols. The gRPC-based network services include:

- gRPC Network Management Interface (gNMI)—Single service for configuration management and streaming telemetry.
- gRPC Network Operations Interface (gNOI)—Suite of microservices for operations management.
- gRPC Routing Information Base Interface (gRIBI)—Service that enables external applications to programmatically add or remove entries in a routing table on the target device.

Figure 1 on page 3 illustrates the scope of the different services.

Figure 1: gRPC Network Interfaces



gNMI, gNOI, and gRIBI Overview

gNMI provides a single service for state management of network elements. gNMI uses vendor-neutral data models that define the configuration and operational state of network devices for common network protocols or services. gNMI clients can retrieve and modify the configuration of a network device as well as stream operational data. Operators monitor the network by subscribing to the specific

data objects of interest. gNMI also supports *on-change* streaming, which is critical for time-sensitive operations.

For more information about using gNMI for configuration management and telemetry streaming on Junos devices, see:

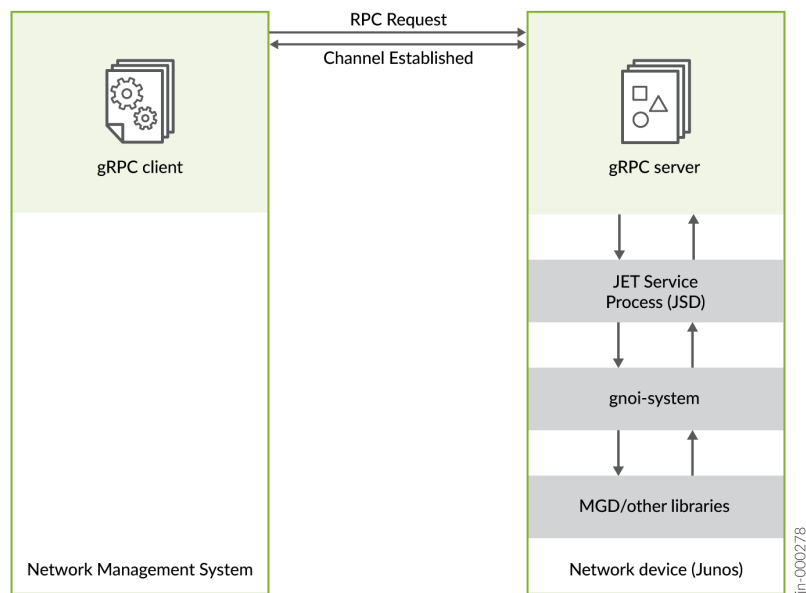
- [OpenConfig User Guide](#)
- [Junos Telemetry Interface User Guide](#)

Whereas gNMI handles state management, gNOI handles operations management. gNOI is a collection of gRPC-based microservices for performing common operations on network devices. Each service definition defines remote procedure calls (RPCs) that management applications can execute on a device to perform a set of operations, for example, rebooting a device or upgrading the software. For a list of supported gNOI services, see "[gNOI Services Overview](#)" on page 34.

gRIBI provides a single service for managing the network device's routing information base (RIB, also known as a routing table) and forwarding information base (FIB, also known as a forwarding table). Management applications can execute gRIBI RPCs on a device to retrieve, add, modify, or delete routes from the device's RIB or FIB. For more information about supported gRIBI RPCs, see "[gRIBI](#)" on page 131.

gNMI, gNOI, and gRIBI use gRPC for transport, and the connection between the gRPC server and the gRPC client is over an SSL-encrypted gRPC session. For example, [Figure 2 on page 5](#) illustrates a simple connection between a gRPC client and server. Juniper Networks supports both server-only authentication and mutual authentication for the gRPC session, which uses X.509 certificates to authenticate the device or application. Certificates can be signed by a certificate authority or self-signed.

Figure 2: gRPC Server and Client Interaction



gNMI, gNOI, and gRIBI define services for managing network devices. Each service definition defines the operations (RPCs) and data structures (messages) for that specific service in proto definition files. The data structures are defined using Protocol Buffers, which is an open-source, language-neutral data format for serializing structured data. You use protoc or an equivalent compiler to compile the proto files for your language of choice. Management applications can use the code in the compiled files to perform the requested operations on network devices. [Table 1 on page 5](#) outlines the GitHub repositories containing the proto definition files for each service interface.

Table 1: gNMI, gNOI, and gRIBI GitHub Repositories

Interface	GitHub Repository
gNMI	openconfig/gnmi
gNOI	openconfig/gnoi
gRIBI	openconfig/gribi

The gRPC-based network services provide alternatives to and advantages over other network management protocols like NETCONF and RESTCONF. Because the proto definition files can be compiled for many different supported languages, the services can be easily integrated with existing management applications to manage multivendor networks. Additionally, the use of Protocol Buffers for

data serialization generally provides more efficient transport and faster processing over other serialization formats such as JSON and XML.

2

CHAPTER

Configure gRPC Services

[Configure gRPC Services](#) | 8

[Configure gNOI Services](#) | 24

Configure gRPC Services

SUMMARY

Configure the gRPC server to enable a client to use gRPC services on the network device, including: gRPC Network Operations Interface (gNOI) services, gRPC Network Management Interface (gNMI) services, and gRPC Routing Information Base Interface (gRIBI) services.

IN THIS SECTION

- [Understanding Authentication and Authorization for gRPC-based Services | 9](#)
- [Obtain X.509 Certificates | 11](#)
- [Load the gRPC Server's Local Certificate in the Junos PKI | 14](#)
- [Enable gRPC Services | 15](#)
- [Configure Mutual \(Bidirectional\) Authentication for gRPC Services | 17](#)
- [Configure the User Account for gRPC Services | 21](#)
- [Configure gRPC RPC Authorization | 22](#)

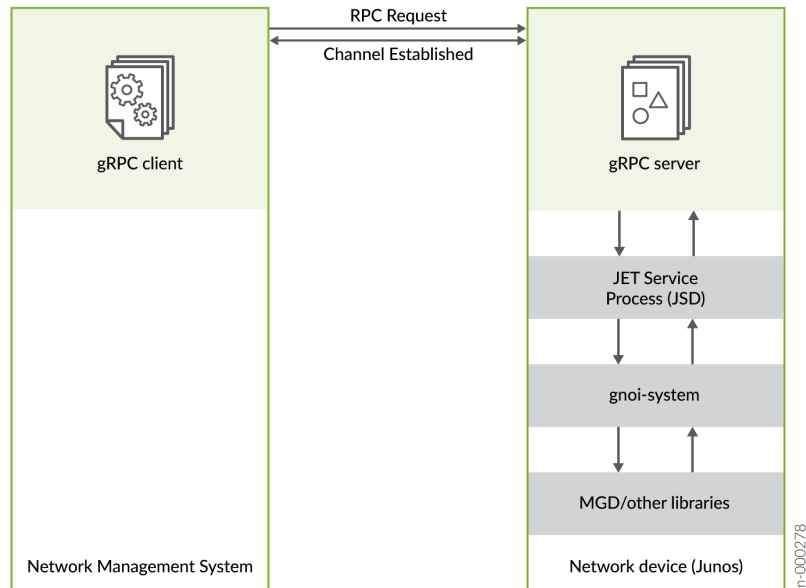
This topic discusses how to configure gRPC services on Junos devices, including the options for authentication and how to configure each option. Before the server and client can establish a gRPC session, you must satisfy the requirements discussed in the following sections:

- ["Understanding Authentication and Authorization for gRPC-based Services" on page 9](#)
- ["Obtain X.509 Certificates" on page 11](#)
- ["Load the gRPC Server's Local Certificate in the Junos PKI" on page 14](#)
- ["Enable gRPC Services" on page 15](#)
- ["Configure Mutual \(Bidirectional\) Authentication for gRPC Services" on page 17 \(Optional\)](#)
- ["Configure the User Account for gRPC Services" on page 21](#)
- ["Configure gRPC RPC Authorization" on page 22 \(Optional\)](#)

Understanding Authentication and Authorization for gRPC-based Services

The gNOI, gNMI, and gRIBI interfaces use the gRPC Remote Procedure Call framework for transport. The gRPC server runs on the network device and listens for connection requests on a specified port. The gRPC client application runs on a remote network management system (NMS) and establishes a gRPC channel with the server on the specified host and port. The client executes RPCs through the SSL-encrypted gRPC session to perform network service operations. [Figure 3 on page 9](#) illustrates a simple connection between a gRPC client and server.

Figure 3: gRPC Server and Client Interaction



gRPC channels use channel credentials to handle authentication between the server and the client. Standard channel credentials use X.509 digital certificates for authenticating the server and the client. A digital certificate provides a way of authenticating users through a trusted third-party called a *certificate authority* or *certification authority (CA)*. The CA verifies the identity of a certificate holder and “signs” the certificate to attest that it has not been forged or altered. The X.509 standard defines the format for the certificate. Digital certificates can be used to establish a secure connection between two endpoints through certificate validation. To establish a gRPC channel, each endpoint (device or application) that requires authentication must supply an X.509 certificate in the exchange.

Junos devices support both server-only authentication as well as mutual authentication for SSL-based gRPC sessions. When server-only authentication is configured, the server provides its public key certificate when the channel is established. The client uses the server's Root CA certificate to

authenticate the server. When mutual authentication is configured, the client also provides its certificate when it connects to the server, and the server validates the certificate. If the certificate validation is successful, the client is allowed to make calls. We recommend that you configure mutual authentication and use CA-signed certificates for the strongest security, although self-signed certificates are accepted.

A public key infrastructure (PKI) supports the distribution and identification of public encryption keys, enabling users to both securely exchange data over networks such as the Internet and verify the identity of the other party. For gRPC-based services, the Junos PKI must contain the certificate for the local device acting as the gRPC server. If you use mutual authentication, the Junos PKI must also contain the Root CA certificates required to validate the certificates of any gRPC clients that connect to the device.

[Table 2 on page 10](#) outlines the general requirements for server-only authentication and mutual authentication when a gRPC client connects to the device to perform gRPC-based services. The gRPC server's certificate must define either the server's hostname in the Common Name (CN) field, or it must define the server's IP address in the Subject Alternative Name (subjectAltName or SAN) IP Address field. The client application must use the same value to establish the connection to the server. If the certificate defines the SubjectAltName IP Address field, the Common Name field is ignored during authentication.

Table 2: Requirements for Server-Only and Mutual Authentication for gRPC Sessions

Requirements	Server-only Authentication	Mutual Authentication
Certificates	<p>The server must have an X.509 public key certificate.</p> <p>If the client connects to the server's IP address instead of the hostname, the server's certificate must include the subjectAltName (SAN) IP address extension field with the IP address of the server.</p>	<p>The server and client must each have an X.509 public key certificate.</p> <p>If the client connects to the server's IP address instead of the hostname, the server's certificate must include the subjectAltName (SAN) IP address extension field with the IP address of the server.</p>
Junos PKI	The server's local certificate must be loaded in the Junos PKI.	The server's local certificate and each client's Root CA certificate must be loaded in the Junos PKI.
Channel credentials	The client must pass in the server's Root CA certificate when the gRPC channel is established.	The client must pass in their certificate and key and the server's Root CA certificate when the gRPC channel is established.

Channel credentials are attached to the gRPC channel and enable the client application to access the service. Call credentials, on the other hand, are attached to a specific service operation (RPC request) and provide information about the person who is using the client application. Call credentials are sent

per request, that is, for each RPC call. To execute gRPC-based operations on Junos devices, you must provide call credentials in the request. The user must either have a user account defined locally on the device, or the user must be authenticated by a TACACS+ server, which then maps the user to a user template account that is defined locally on the device. You can provide the call credentials (username and password) in the RPC's `metadata` argument. If authentication is successful, the Junos device executes the RPC request using the account privileges of the specified user.

NOTE: As an alternative to passing in call credentials for every RPC executed on a Junos device, you can use the Juniper Extension Toolkit [jnx_authentication_service API](#) to log in to the device once at the start of the gRPC session, and all subsequent RPCs executed in the channel are authenticated. You can download the JET Client IDL library from the [Juniper Networks download site](#).

By default, Junos devices authorize an authenticated gRPC client to execute all gRPC RPCs. You can optionally configure a gRPC user's login class to explicitly allow or deny specific gRPC RPCs. To specify the RPCs, you configure the `allow-grpc-rpc-regexps` and `deny-grpc-rpc-regexps` statements and define regular expressions that match the RPCs. See ["Configure gRPC RPC Authorization" on page 22](#) for more information.

Obtain X.509 Certificates

The SSL-encrypted gRPC session uses X.509 public key certificates to authenticate the gRPC server and client. For server-only authentication, the gRPC server must have a certificate. For mutual authentication, both the gRPC server and client must have certificates. The requirements for the certificates are:

- The certificate can be signed by a certificate authority (CA) or self-signed.
- The certificate must be PEM-encoded.
- The gRPC server's certificate must define either the gRPC server's hostname in the Common Name (CN) field, or it must define the gRPC server's IP address in the SubjectAltName (SAN) IP Address field. The gRPC client must use the same value to establish the connection to the server. If the certificate defines the SubjectAltName IP Address, the Common Name field is ignored during authentication.

To use OpenSSL to obtain the gRPC server's certificate:

1. Generate a private key, and specify the key length in bits.

```
user@nms:~$ openssl genrsa -out server.key 4096
Generating RSA private key, 4096 bit long modulus (2 primes)
...++++
.....++++
e is 65537 (0x010001)
```

2. If the gRPC client connects to the gRPC server's IP address, update your **openssl.cnf** or equivalent configuration file to define the `subjectAltName=IP` extension with the gRPC server's IP address.

```
user@nms:~$ cat openssl.cnf
# OpenSSL configuration file.
...
extensions          = v3_sign
...
[v3_sign]
subjectAltName=IP:10.53.52.169
```

3. Generate a certificate signing request (CSR), which contains the entity's public key and information about their identity.

```
user@nms:~$ openssl req -new -key server.key -out server.csr
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:CA
Locality Name (eg, city) []:Sunnyvale
Organization Name (eg, company) [Internet Widgits Pty Ltd]: Acme
Organizational Unit Name (eg, section) []: testing
Common Name (e.g. server FQDN or YOUR name) []:gnoi-server.example.com
Email Address []:
Please enter the following 'extra' attributes
to be sent with your certificate request
```

```
A challenge password []:
An optional company name []:
```

Alternatively, you can provide the CSR information in a single command, for example:

```
user@nms:~$ openssl req -new -key server.key -out server.csr -subj "/C=US/ST=CA/L=Sunnyvale/
O=Acme/OU=testing/CN=gnoi-server.example.com"
```

4. Generate the certificate by doing one of the following:

- Send the CSR to a certificate authority to request an X.509 certificate, and provide the configuration file to include any additional extensions.
- Sign the CSR with a CA to generate the certificate, and include the `-extfile` option if you need to reference your configuration file and extensions.

```
user@nms:~$ openssl x509 -req -in server.csr -CA /etc/pki/certs/ServerRootCA.crt -
CAkey /etc/pki/certs/ServerRootCA.key -set_serial 0101 -out server.crt -days 365 -sha256 -
extfile openssl.cnf
Signature ok
subject=C = US, ST = CA, L = Sunnyvale, O = Acme, OU = testing, CN = gnoi-
server.example.com
Getting Private key
```

- Sign the CSR with the server key to generate a self-signed certificate, and include the `-extfile` option if you need to reference your configuration file and extensions.

```
user@nms:~$ openssl x509 -req -in server.csr -signkey server.key -out server.crt -days
365 -sha256 -extfile openssl.cnf
Signature ok
subject=C = US, ST = CA, L = Sunnyvale, O = Acme, OU = testing, CN = gnoi-
server.example.com
Getting Private key
```

5. Verify that the certificate's Common Name (CN) field and extensions, if provided, are correct.

```
user@nms:~$ openssl x509 -text -noout -in server.crt
Certificate:
    Data:
        Version: 3 (0x2)
        ...
```

```

    Subject: C = US, ST = CA, L = Sunnyvale, O = Acme, OU = testing, CN = gnoi-
server.example.com
    ...
    X509v3 extensions:
        X509v3 Subject Alternative Name:
            IP Address:10.53.52.169
    ...

```

For mutual authentication, repeat the previous steps with the information for the gRPC client to generate the client's key and certificate. The client certificate does not require the SAN IP extension field.

Load the gRPC Server's Local Certificate in the Junos PKI

The network device running the gRPC server must have an X.509 certificate that identifies the device to gRPC clients. To perform gRPC-based services on the Junos device, you must load the public key certificate and key for the local network device in the Junos PKI. After you load the certificate and perform the initial configuration, gRPC clients can then use any micro-service to update the certificate. For example, a gRPC client can use the gNOI CertificateManagement service to install a new certificate or replace an existing certificate.

To load the local device's certificate and key in the PKI:

1. Download the certificate and key for the device that is acting as the gRPC server to that device.
2. In operational mode, define an identifier and load the local device's certificate and key into the PKI.

```

user@host> request security pki local-certificate load certificate-id gnoi-server
filename /var/tmp/server.crt key /var/tmp/server.key
Local certificate loaded successfully

```

3. (Optional) Verify the certificate is present in the PKI database.

```

user@host> show security pki local-certificate certificate-id gnoi-server
LSYS: root-logical-system
Certificate identifier: gnoi-server
    Issued to: gnoi-server.example.com, Issued by: C = US, ST = CA, O = serverRootCA0rg, CN =
serverRootCA
    Validity:
        Not before: 04-13-2022 18:15 UTC
        Not after: 04-13-2023 18:15 UTC

```

```
Public key algorithm: rsaEncryption(4096 bits)
Keypair Location: Keypair generated locally
```

Enable gRPC Services

gRPC-based services use an API connection setting based on Secure Socket Layer (SSL) technology. For an SSL-based connection, you must specify a local certificate that identifies the gRPC server.

After you enable gRPC services and specify a local certificate, the network device uses server-only authentication. You can then optionally configure mutual authentication by completing the steps described in ["Configure Mutual \(Bidirectional\) Authentication for gRPC Services" on page 17](#).

To configure your network device for gRPC services and specify the local certificate used for server authentication:

1. Navigate to the SSL-based API connection settings for gRPC services.

```
[edit]
user@host# edit system services extension-service request-response grpc ssl
```

2. Configure the port to use for gRPC services.

```
[edit system services extension-service request-response grpc ssl]
user@host# set port port-number
```

For example:

```
[edit system services extension-service request-response grpc ssl]
user@host# set port 50051
```

3. Specify a local certificate name.

Enter the identifier for the local certificate that you previously loaded into the Junos PKI with the `request security pki local-certificate load operational mode` command.

- If you are updating the Junos configuration hierarchy, add the server certificate as follows:

```
[edit system services extension-service request-response grpc ssl]
user@host# set local-certificate certificate-id
```

The following example configures the local certificate `gnoi-server`:

```
[edit system services extension-service request-response grpc ssl]
user@host# set local-certificate gnoi-server
```

- Alternatively, if you are updating the OpenConfig configuration hierarchy instead, add the server certificate as follows:

```
[edit]
user@host# set system schema openconfig unhide
user@host# set openconfig-system:system openconfig-system-grpc:grpc-servers grpc-server
jsd config certificate-id certificate-id
```

4. Configure the device to use the PKI database for certificates.

```
[edit system services extension-service request-response grpc ssl]
user@host# set use-pki
```

5. Enable the device to reload certificates without terminating the gRPC session.

```
[edit system services extension-service request-response grpc ssl]
user@host# set hot-reloading
```

6. (Optional) Specify an IP address to listen to for incoming connections.

```
[edit system services extension-service request-response grpc ssl]
user@host# set ip-address address
```

For example:

```
[edit system services extension-service request-response grpc ssl]
user@host# set ip-address 192.168.2.1
```

NOTE: If you do not specify an IP address, the default address of `::` is used to listen for incoming connections.

7. (Optional) Configure tracing for extension services to debug any issues that might arise.

```
[edit]
user@host# top
user@host# set system services extension-service traceoptions file jsd
user@host# set system services extension-service traceoptions flag all
```

NOTE: To view Junos OS Evolved trace files for extensions services, use the `show trace application jsd` and `show trace application jsd live` operational mode commands.

8. Commit the configuration.

```
user@host# commit
```

To configure mutual authentication instead of server-only authentication, you must also complete the steps in ["Configure Mutual \(Bidirectional\) Authentication for gRPC Services" on page 17](#).

Configure Mutual (Bidirectional) Authentication for gRPC Services

IN THIS SECTION

- [Configure Mutual Authentication in the Device Configuration | 18](#)
- [Configure Mutual Authentication Using the gNOI CertificateManagement Service | 20](#)

You can configure mutual (bidirectional) authentication for gRPC sessions, which authenticates both the network device as the gRPC server and the network management system as the gRPC client using SSL certificates. The Junos device uses the credentials provided by the external client to authenticate the client and authorize a connection.

You can configure mutual authentication on Junos devices using one of the following options:

- Configure the mutual authentication settings directly under the `[edit system services extension-service request-response grpc ssl mutual-authentication]` hierarchy level.

- Set up server-only authentication initially, and then use the gNOI CertificateManagement service to load the necessary CA certificates on the device.

If you configure mutual authentication directly in the device configuration, the device configuration takes precedence over any setup done using the gNOI services.

Before you begin:

- Load the certificate and key for the network device acting as the gRPC server into the device's PKI as described in ["Load the gRPC Server's Local Certificate in the Junos PKI" on page 14](#).
- Enable gRPC services and configure the local server authentication as described in ["Enable gRPC Services" on page 15](#).

The following sections discuss the different methods for configuring mutual authentication. You can use whichever method works best for your environment.

Configure Mutual Authentication in the Device Configuration

To configure authentication for the gRPC client directly in the network device configuration:

1. Download the root CA certificate that will be used to validate the client's certificate to the local device acting as the gRPC server.
2. Configure the certificate authority profile for the client certificate's root CA at the [edit security pki] hierarchy.

```
[edit security pki]
user@host# set ca-profile ca-profile-name ca-identity ca-identifier
```

For example:

```
[edit security pki]
user@host# set ca-profile gnoi-client ca-identity clientRootCA
```

3. Commit the configuration.

```
[edit]
user@host# commit and-quit
```


4. In operational mode, load the root CA certificate that will be used to verify the client's certificate into the Junos PKI. Specify the `ca-profile` identifier that you configured in the previous steps.

```
user@host> request security pki ca-certificate load ca-profile ca-profile filename cert-path
```

For example:

```
user@host> request security pki ca-certificate load ca-profile gnoi-client filename /var/tmp/
clientRootCA.crt
Fingerprint:
  00:2a:30:e9:59:94:db:f1:a1:5c:d1:c9:d4:5f:db:8f:f1:f0:8d:c4 (sha1)
  02:3b:a0:b8:95:0c:a2:fa:15:18:57:3d:a3:10:e9:ac (md5)

69:97:90:39:de:75:a0:1d:94:1e:06:a8:be:8c:66:e5:41:95:fd:dc:14:8a:e7:3a:e0:42:9e:f9:f7:dd:c8:c
2 (sha256)
Do you want to load this CA certificate ? [yes,no] (no) yes

CA certificate for profile gnoi-client loaded successfully
```

TIP: To load a CA certificate bundle, issue the `request security pki ca-certificate ca-profile-group load ca-group-name ca-group-name filename bundle-path` command.

After loading the certificate, enter configuration mode and continue configuring mutual authentication.

5. Enable mutual authentication and specify the requirements for client certificates.

```
[edit system services extension-service request-response grpc ssl]
user@host# set mutual-authentication client-certificate-request requirement
```

For example, to specify the strongest authentication, which requires a certificate and its validation, use `require-certificate-and-verify`.

```
[edit system services extension-service request-response grpc ssl]
user@host# set mutual-authentication client-certificate-request require-certificate-and-verify
```

NOTE: The default is no-certificate. The other options are: request-certificate, request-certificate-and-verify, require-certificate, require-certificate-and-verify.

We recommend that you use the no-certificate option in a test environment only.

6. Specify the certificate authority profile that will be used to verify the client certificate.

The certificate authority profile was configured in step "2" on page 18.

```
[edit system services extension-service request-response grpc ssl]
user@host# set mutual-authentication certificate-authority certificate-authority
```

For example, to specify the certificate authority profile named gnoi-client:

```
[edit system services extension-service request-response grpc ssl]
user@host# set mutual-authentication certificate-authority gnoi-client
```

7. Commit the configuration.

```
[edit]
user@host# commit and-quit
```

Configure Mutual Authentication Using the gNOI CertificateManagement Service

You can use the gNOI CertificateManagement service to set up mutual authentication between the gRPC client and gRPC server instead of configuring the settings directly in the device configuration. You initially set up server-only authentication and then use the gNOI CertificateManagement service RPCs to load the client CA certificates. See ["gNOI Certificate Management Service" on page 41](#) for information about loading the certificates using the gNOI CertificateManagement service.

The gRPC server supports only one global CA certificate bundle for gNOI services. When you use the gNOI CertificateManagement service to load the CA certificate bundle, the device implicitly uses mutual authentication. However, you should take note of the following:

- The CertificateManagement service always loads the CA certificate bundle using the ca-profile-group reserved identifier gnoi-ca-bundle.

- If you use the CertificateManagagment service to load the CA certificate bundle, the device implicitly uses mutual authentication and assumes the following configuration, even though it is not explicitly configured on the device.

```
[edit system services extension-service request-response grpc ssl]
mutual-authentication {
    certificate-authority gnoi-ca-bundle;
    client-certificate-request require-certificate-and-verify;
}
```

- If the CertificateManagagment service sends a request to load a new CA certificate bundle, the server clears the certificates for the previous CA bundle from the device and loads the new ones.
- If you use the CertificateManagagment service to load a CA certificate bundle and you also configure the [edit system services extension-service request-response grpc ssl mutual-authentication] statement hierarchy, then the configured statements take precedence.

Configure the User Account for gRPC Services

Channel credentials are attached to the gRPC channel and enable the client application to access the service. Call credentials are attached to a specific RPC request and provide information about the user who is using the client application. You must provide call credentials in each RPC request, which requires a user account for the network device. The user must have a user account defined locally on the network device, or the user must be authenticated by a TACACS+ server, which then maps the user to a user template account that is defined locally on the device.

To create a user account:

1. Configure the user statement with a unique username, and include the class statement to specify a login class that has the permissions required for all actions to be performed by the user. For example:

```
[edit system login]
user@host# set user gnoi-user class super-user
```

2. For local user accounts, configure the user's password.

You can omit the password for local user template accounts because the user is authenticated through a remote authentication server.

```
[edit system login]
user@host# set user gnoi-user authentication plain-text-password
New password:
Retype new password:
```

3. (Optional) Configure the `full-name` statement to specify the user's name.

```
[edit system login]
user@host# set user gnoi-user full-name "gNOI client"
```

4. Commit the configuration to activate the user account on the device.

```
[edit system login]
user@host# commit
```

5. Repeat the preceding steps on each network device where the gRPC client will execute RPCs in a gRPC session.

Configure gRPC RPC Authorization

By default, Junos devices authorize an authenticated gRPC client to execute all gRPC RPCs. You can configure a Junos login class to explicitly allow or deny gRPC RPCs. To specify the RPCs, you configure the `allow-grpc-rpc-regexps` and `deny-grpc-rpc-regexps` statements and define regular expressions that match the RPCs. If there are conflicting expressions in the allow and deny lists, the deny list takes precedence. If an RPC does not match either list, the RPC is allowed by default.

Junos devices use the following syntax for specifying gRPC RPCs:

```
/package.service/rpc
```

where *package*, *service*, and *rpc* are the names defined in the respective statement in that service's proto definition file. For example:

```
/gnmi.gNMI/Get
/gnoi.certificate.CertificateManagement/Rotate
```

```
/gnoi.system.System/Reboot
/gnoi.system.System/RebootStatus
/gribi.gRIBI/.*
```

You can configure multiple `allow-grpc-rpc-regexps` and `deny-grpc-rpc-regexps` statements with one or more expressions. Enclose each expression within quotation marks (" "). Enclose multiple expressions in square brackets [], and separate the expressions with a space.

```
allow-grpc-rpc-regexps ["regex1" "regex2" ... ]
allow-grpc-rpc-regexps "regex3"
```

To create a login class that defines authorization for gRPC RPCs:

1. Configure the login class name and permissions.

```
[edit system login]
user@host# set class class-name permissions [permission1 permission2 ...]
```

For example:

```
[edit system login]
user@host# set class grpc-operator permissions all
```

2. Within the class, configure regular expressions for the RPCs that the class allows.

```
[edit system login class class-name]
user@host# set allow-grpc-rpc-regexps ["regex1" "regex2" ... ]
user@host# set allow-grpc-rpc-regexps "regex3"
```

For example, the following statement allows the gNMI Get() RPC and all gNOI System service RPCs.

```
[edit system login class grpc-operator]
user@host# set allow-grpc-rpc-regexps ["/gnmi.gNMI/Get" "/gnoi.system.System/.*"]
```

3. Configure regular expressions for the RPCs that the class denies.

```
[edit system login class class-name]
user@host# set deny-grpc-rpc-regexps ["regex1" "regex2" ... ]
```

For example, the following statements deny the gNMI Set() RPC and also deny all RPCs for the gRIBI service as well as the gNOI CertificateManagement service.

```
[edit system login class grpc-operator]
user@host# set deny-grpc-rpc-regexps ["/gnmi.gNMI/Set" "/gribi.gRIBI/.*"]
user@host# set deny-grpc-rpc-regexps "/gnoi.certificate.CertificateManagement/.*"

```

4. Assign the login class to the appropriate gRPC users.

```
[edit system login]
user@host# set user username class class-name

```

For example, the following statement assigns the grpc-operator class to the grpc-user user.

```
[edit system login]
user@host# set user grpc-user class grpc-operator

```

After enabling gRPC services on the network device, set up the remote network management system as a gRPC client. To enable the client to execute gNOI operations, configure the client as outlined in ["Configure gNOI Services" on page 24](#).

Configure gNOI Services

SUMMARY

Configure the remote network management system as a gRPC client that can execute gNOI operations on network devices.

IN THIS SECTION

- [Set up the gRPC Client | 25](#)
- [Compile the Proto Definition Files | 26](#)
- [Create gNOI Applications | 28](#)
- [Execute the Application | 32](#)

The gRPC Network Operations Interface (gNOI) uses the gRPC Remote Procedure Call (gRPC) framework to perform operations on a network device. The network management system must have the gRPC stack installed.

OpenConfig defines proto definition files for [gNOI](#) services. Proto definition files define the operations (RPCs) and the data structures (messages) for a given service. The definitions are language agnostic. gRPC supports using many different languages to execute service operations. You must compile the proto definition files for your language of choice. You then create applications that use the objects (classes, functions, methods, etc) in the compiled files to connect to the gRPC server on the network device and execute the desired operations.

For information about using gRPC with the different supported languages, consult the [gRPC documentation](#). The following sections provide sample commands for setting up a gRPC client and downloading and compiling the gNOI proto definition files for Python. You must use the commands that are appropriate for your operating system, environment, and gRPC language of choice.

Before you configure the gRPC client, configure the gRPC server as defined in "[Configure gRPC Services](#)" on page 8.

Set up the gRPC Client

gNOI uses the gRPC framework to perform operations on a network device. gRPC supports using many different languages. Before you can perform gNOI operations using your language of choice, you must install the gRPC stack on the network management system.

For example, to install the gRPC stack for Python on a network management system running Ubuntu 20.04 LTS (use `sudo` where appropriate):

1. Install pip for Python 3.

```
user@nms:~$ sudo apt install python3-pip
```

2. Install the `grpcio` package for Python.

```
user@nms:~$ sudo pip3 install grpcio
```

3. Install the `grpcio-tools` package for Python.

```
user@nms:~$ sudo pip3 install grpcio-tools
```

Compile the Proto Definition Files

gRPC supports using many languages. In order to perform gRPC operations on network devices, you must compile the respective proto definition files for your language of choice. OpenConfig provides the necessary proto definition files in the [OpenConfig GitHub repository](#). You use the protocol buffer compiler (protoc or equivalent application) to compile the .proto files.

For this setup, we execute a script that copies all the desired .proto files into a directory, updates the files to use relative import statements, and then compiles the files.

To download and compile the gNOI proto definition files for Python:

1. Create the temporary directory for the original source files.

```
user@nms:~$ mkdir -p src/github.com/openconfig/gnoi
```

2. Clone the gNOI GitHub repository to the local device.

```
user@nms:~$ git clone https://github.com/openconfig/gnoi.git src/github.com/openconfig/gnoi
```

3. Compile the .proto files for your language, which in this example is Python.

The sample shell script performs the following operations:

- Creates the **src/proto** directory.
- Copies the desired proto files into the new directory.
- Updates the import statement in each proto file to use a relative path.
- Compiles each proto file in the specified list for use with Python.

```
user@nms:~$ cat compile-proto.sh
```

```
#!/usr/bin/env bash

src=src/proto

gnoiFileList="types common cert diag file os system"

echo "Updating proto file source location and import statements"
mkdir -p $src
```



```

for p in $gnoiFileList; do
    cp src/github.com/openconfig/gnoi/$p/$p.proto $src
    python3 -c "
import re
with open('$src/$p.proto', 'r') as fd:
    data = fd.read()
data1 = re.sub(r'import \"(common|types)\\/((common|types).proto)\\\"', r'import \"\2\\\"', data)
with open('$src/$p.proto', 'w') as fd:
    fd.write(data1)    "
done

echo "Compiling proto files"
for p in $gnoiFileList; do
    python3 -m grpc_tools.protoc --proto_path=$src --python_out=$src --grpc_python_out=$src
    $p.proto
done

```

To compile the files using the script, execute the script from the parent directory that contains the `src` directory.

```

user@nms:~$ sh compile-proto.sh
Updating proto file source location and import statements
Compiling proto files

```

NOTE: The script compiles only the subset of proto files (as defined in `gnoiFileList`) that have services supported on Junos devices. To compile other services as they become available, you can update the variables to include those services.

4. Verify that the proto files are compiled by viewing the output files in the target directory.

The file list should include the compiled files, which for Python have **_pb2** and **pb2_grpc** in the output filenames.

```

user@nms:~$ ls src/proto

```

```

user@nms:~$ ls src/proto
cert_pb2_grpc.py    common.proto    file_pb2.py    system_pb2_grpc.py  types.proto

```

cert_pb2.py	diag_pb2_grpc.py	file.proto	system_pb2.py
cert.proto	diag_pb2.py	os_pb2_grpc.py	system.proto
common_pb2_grpc.py	diag.proto	os_pb2.py	types_pb2_grpc.py
common_pb2.py	file_pb2_grpc.py	os.proto	types_pb2.py

Create gNOI Applications

IN THIS SECTION

- [grpc_channel.py | 28](#)
- [gnoi_connect_cert_auth_mutual.py | 29](#)

After you compile the proto definition files, you create applications that use the objects in the compiled files. The applications connect to the gRPC server on the network device and perform the desired operations. This section provides two sample Python modules, which are described in their respective sections.

grpc_channel.py

The `grpc_channel.py` Python module provides sample functions that create a gRPC channel using the arguments provided for the selected method of authentication, server-only or mutual.

```
import grpc
from os.path import isfile

def grpc_authenticate_channel_mutual(server, port, root_ca_cert="", client_key="",
client_cert=""):
    if not isfile(root_ca_cert):
        raise Exception("Error: root_ca_cert file does not exist")
    if (client_key == "") or (not isfile(client_key)):
        raise Exception(
            "Error: client_key option is missing or target file does not exist")
    elif (client_cert == "") or (not isfile(client_cert)):
        raise Exception(
```

```

        "Error: client_cert option is empty or target file does not exist")

    print("Creating channel")
    creds = grpc.ssl_channel_credentials(open(root_ca_cert, 'rb').read(),
                                         open(client_key, 'rb').read(),
                                         open(client_cert, 'rb').read())
    channel = grpc.secure_channel('%s:%s' % (server, port), creds)

    return channel

def grpc_authenticate_channel_server_only(server, port, root_ca_cert=""):
    if isfile(root_ca_cert):
        print("Creating channel")
        creds = grpc.ssl_channel_credentials(open(root_ca_cert, 'rb').read(),
                                              None,
                                              None)
        channel = grpc.secure_channel('%s:%s' % (server, port), creds)
        return channel
    else:
        raise Exception("root_ca_cert file does not exist")

```

gnoi_connect_cert_auth_mutual.py

The gnoi_connect_cert_auth_mutual.py Python application establishes a gRPC channel with the given gRPC server and executes a simple gNOI System service operation. The user provides the necessary connection and mutual authentication information as input to the application. The application invokes the appropriate function in the grpc_channel.py module to establish the gRPC channel between the client and the server. If the application successfully establishes a gRPC channel, it then executes a simple system service RPC to retrieve the time from the network device.

```

"""gRPC gNOI Time request utility."""

from __future__ import print_function
import argparse
import logging
from getpass import getpass

import system_pb2
import system_pb2_grpc

```

```

from grpc_channel import grpc_authenticate_channel_mutual

def get_args(parser):
    parser.add_argument('--server',
                        dest='server',
                        type=str,
                        default='localhost',
                        help='Server IP or name. Default is localhost')

    parser.add_argument('--port',
                        dest='port',
                        nargs='?',
                        type=int,
                        default=50051,
                        help='The server port. Default is 50051')

    parser.add_argument('--client_key',
                        dest='client_key',
                        type=str,
                        default='',
                        help='Full path of the client private key. Default ""')

    parser.add_argument('--client_cert',
                        dest='client_cert',
                        type=str,
                        default='',
                        help='Full path of the client certificate. Default ""')

    parser.add_argument('--root_ca_cert',
                        dest='root_ca_cert',
                        required=True,
                        type=str,
                        help='Full path of the Root CA certificate.')

    parser.add_argument('--user_id',
                        dest='user_id',
                        required=True,
                        type=str,
                        help='User ID for RPC call credentials.')

    args = parser.parse_args()
    return args

```

```

def send_rpc(channel, metadata):
    stub = system_pb2_grpc.SystemStub(channel)
    print("Executing GNOI::System::Time RPC")
    req = system_pb2.TimeRequest()
    try:
        response = stub.Time(request=req, metadata=metadata, timeout=60)
    except Exception as e:
        logging.error('Error executing RPC: %s', e)
        print(e)
    else:
        logging.info('Received message: %s', response)
        return response

def main():
    parser = argparse.ArgumentParser()
    args = get_args(parser)

    grpc_server_password = getpass("gRPC server password for executing RPCs: ")
    metadata = [('username', args.user_id),
                ('password', grpc_server_password)]

    try:
        # Establish grpc channel to network device
        channel = grpc_authenticate_channel_mutual(
            args.server, args.port, args.root_ca_cert, args.client_key, args.client_cert)
        response = send_rpc(channel, metadata)
        print("Response received: time since last epoch in nanoseconds is ", str(response))
    except Exception as e:
        logging.error('Received error: %s', e)
        print(e)

if __name__ == '__main__':
    logging.basicConfig(filename='gnoi-testing.log',
                        format='%(asctime)s %(levelname)-8s %(message)s',
                        level=logging.INFO,
                        datefmt='%Y-%m-%d %H:%M:%S')

    main()

```

Execute the Application

After you create applications to perform gNOI service operations, you execute the applications and provide any necessary arguments. The following example uses the scripts provided in the previous section to connect to the gRPC server on the network device and request the time. The gRPC server is configured to require and verify the client's certificate.

- For mutual authentication, the client provides their own key and X.509 public key certificate in PEM format in addition to the server's IP address, gRPC port, and root CA certificate. The client also provides the credentials for RPC calls: the `user_id` argument supplies the username, and the application prompts for the user password.

```
lab@gnoi-client:~/src/proto$ python3 gnoi_connect_cert_auth_mutual.py --server 10.53.52.169 --  
port 50051 --root_ca_cert /etc/pki/certs/serverRootCA.crt --client_key /home/lab/certs/  
client.key --client_cert /home/lab/certs/client.crt --user_id gnoi-user  
gRPC server password for executing RPCs:  
Creating channel  
Executing GNOI::System::Time RPC  
Response received: time since last epoch in nanoseconds is  time: 1650061065769701762
```

3

CHAPTER

gNOI Services

[gNOI Services Overview](#) | 34

[gNOI Certificate Management Service](#) | 41

[gNOI Diagnostic \(Diag\) Service](#) | 73

[gNOI File Service](#) | 86

[gNOI Operating System \(OS\) Service](#) | 97

[gNOI System Service](#) | 105

gNOI Services Overview

SUMMARY

Junos devices support the gNOI services and RPCs outlined in this topic.

IN THIS SECTION

- Supported Services Overview | 34
- CertificateManagement Service | 35
- Diagnostic (Diag) Service | 37
- File Service | 37
- Operating System (OS) Service | 38
- System Service | 39

The gRPC Network Operations Interface (gNOI) defines services for performing operational commands on network devices. OpenConfig defines the operations (RPCs) and data structures (messages) for each service in proto definition files. The proto files with the full list of gNOI RPCs are located in the OpenConfig gNOI GitHub repository at <https://github.com/openconfig/gnoi>. Junos devices support a subset of the services and RPCs as described in the following sections.

Supported Services Overview

Table 3 on page 34 outlines the gNOI services supported on Junos devices. The table lists the release in which support starts for the given version of the proto file.

Table 3: Supported gNOI Services

Service	Release Starting Support	Proto File Version
CertificateManagement	Junos OS Evolved Release 22.2R1	0.2.0
Diag	Junos OS Evolved Release 22.2R1	0.1.0
File	Junos OS Evolved Release 22.2R1	0.1.0

Table 3: Supported gNOI Services (*Continued*)

Service	Release Starting Support	Proto File Version
OS	Junos OS Evolved Release 22.2R1	0.1.1
System	Junos OS Evolved Release 22.2R1	1.0.0

CertificateManagement Service

Table 4: Supported cert.proto RPCs

RPC	Description	Introduced in Release
CanGenerateCSR()	<p>Query the target device to determine if it can generate a certificate signing request (CSR) with the specified key type, key size, and certificate type. Supported values:</p> <ul style="list-style-type: none"> Key type: KT_RSA Key sizes: 1024, 2048, 4096 Certificate type: CT_X509 <p>Returns True if the gNOI server supports the specific key type, key size, and certificate type.</p>	Junos OS Evolved 23.1R1
GenerateCSR()	Generate and return a certificate signing request (CSR).	Junos OS Evolved 22.2R1
GetCertificates()	Return the local certificates loaded on the target device.	Junos OS Evolved 22.2R1

Table 4: Supported cert.proto RPCs *(Continued)*

RPC	Description	Introduced in Release
Install()	Load a new certificate on the target device by creating a CSR request, generating a certificate based on the CSR, and loading the certificate using a new certificate ID.	Junos OS Evolved 22.2R1
LoadCertificate()	Load a certificate signed by a Certificate Authority (CA) on the target device.	Junos OS Evolved 22.2R1
LoadCertificateAuthorityBundle()	Load a CA certificate bundle on the target device.	Junos OS Evolved 22.2R1
RevokeCertificates()	Revoke the certificates with the specified certificate IDs on the target device.	Junos OS Evolved 23.1R1
Rotate()	Replace an existing certificate on the target device by creating a CSR request, generating a certificate based on the CSR, and loading the certificate using an existing certificate ID.	Junos OS Evolved 22.2R1

Diagnostic (Diag) Service

Table 5: Supported diag.proto RPCs

RPC	Description	Introduced in Release
StartBERT()	<p>Start a BERT on a set of ports. Junos devices support the following PRBS patterns for gNOI BERTs:</p> <ul style="list-style-type: none"> • PRBS7 • PRBS9 • PRBS15 • PRBS23 • PRBS31 	Junos OS Evolved 22.2R1
StopBert()	Stop an already in-progress BERT on a set of ports.	Junos OS Evolved 22.2R1
GetBERTResult()	Get BERT results during the BERT or after it completes.	Junos OS Evolved 22.2R1

File Service

Table 6: Supported file.proto RPCs

RPC	Description	Introduced in Release
Get()	<p>Read and stream the contents of a file from the target.</p> <p>The file is streamed by sequential messages, each message containing up to 64KB of data. A final message is sent prior to closing the stream that contains the hash of the data sent. The operation returns an error if the file does not exist or there is an error reading the file.</p>	Junos OS Evolved 22.2R1

Table 6: Supported file.proto RPCs (*Continued*)

RPC	Description	Introduced in Release
Put()	<p>Stream data to a file on the target.</p> <p>The file is sent in sequential messages, each message containing up to 64KB of data. A final message must be sent that includes the hash of the data.</p> <p>The operation returns an error if the location does not exist or there is an error writing the data. If no checksum is received, the target removes the partially transmitted file. A failure will not alter any existing file of the same name.</p>	Junos OS Evolved 22.2R1
Remove()	Remove the specified file from the target. The operation returns an error if the file does not exist, if the file path resolves to a directory, or if the remove operation encounters an error.	Junos OS Evolved 22.2R1
Stat()	Return metadata about a file on the target device. The operation returns an error if the file does not exist or there is an error accessing the metadata.	Junos OS Evolved 22.2R1

Operating System (OS) Service

Table 7: Supported os.proto RPCs

RPC	Description	Introduced in Release
Activate()	<p>Set the OS version that is used at the next reboot.</p> <p>Activate() reboots the target if the no_reboot flag is omitted or set to False. If the reboot fails to boot the requested OS version, the target rolls back to the previous OS package.</p> <p>NOTE: Junos devices do not support the standby_supervisor field in the ActivateRequest message.</p>	Junos OS Evolved 22.2R1

Table 7: Supported os.proto RPCs (*Continued*)

RPC	Description	Introduced in Release
Install()	Transfer a software image onto the target. NOTE: Junos devices do not support the standby_supervisor field in the TransferRequest message.	Junos OS Evolved 22.2R1
Verify()	Check the running OS version. This RPC may be called multiple times while the target boots until successful. NOTE: Junos devices do not support verify_standby for VerifyResponse.	Junos OS Evolved 22.2R1

System Service

IN THIS SECTION

- [System Service \(Ping and Traceroute\) | 39](#)
- [System Service \(Reboot\) | 40](#)
- [System Service \(Software Upgrade\) | 41](#)
- [System Service \(Routing Engine Switchover\) | 41](#)

System Service (Ping and Traceroute)

Table 8: Supported system.proto RPCs for Troubleshooting the Network

RPC	Description	Introduced in Release
Ping()	Ping a device. The Ping() RPC supports IPv4 and IPv6 pings. This RPC streams back the results of the ping after the ping is complete. Default number of packets: 5	Junos OS Evolved 22.2R1

Table 8: Supported system.proto RPCs for Troubleshooting the Network *(Continued)*

RPC	Description	Introduced in Release
Traceroute()	<p>Execute the traceroute command on the target device and stream back the results.</p> <p>Default hop count: 30</p>	Junos OS Evolved 22.2R1

System Service (Reboot)

Table 9: Supported system.proto RPCs for Rebooting

RPC	Description	Introduced in Release
Reboot()	<p>Reboot the target. You can only execute one reboot request on a target at a time.</p> <p>You can optionally configure a delay to reboot in the future, reboot subcomponents individually, and add a message when the reboot initiates. The delay is configured in nanoseconds.</p> <p>Junos devices support the following reboot methods:</p> <ul style="list-style-type: none"> • COLD (1) • POWERDOWN (2) • HALT (3) • POWERUP (7) 	Junos OS Evolved 22.2R1
RebootStatus()	Return the status of the reboot.	Junos OS Evolved 22.2R1
CancelReboot()	Cancel a pending reboot request.	Junos OS Evolved 22.2R1

System Service (Software Upgrade)

Table 10: Supported system.proto RPCs for Software Upgrades

RPC	Description	Introduced in Release
SetPackage()	Install a software image on the target device.	Junos OS Evolved 22.2R1

System Service (Routing Engine Switchover)

Table 11: Supported system.proto RPCs for Routing Engine Switchover

RPC	Description	Introduced in Release
SwitchControlProcessor()	<p>Switch from the current Routing Engine to the specified Routing Engine. If the current and specified Routing Engines are the same, it is a NOOP. If the target does not exist, the RPC returns an error.</p> <p>NOTE: Junos devices do not support control_processor for SwitchControlProcessorResponse.</p>	Junos OS Evolved 22.2R1

gNOI Certificate Management Service

SUMMARY

Use the gNOI CertificateManagement service to manage certificates on the target network element.

IN THIS SECTION

- [Overview | 42](#)
- [Supported RPCs | 45](#)
- [Network Device Configuration | 46](#)
- [Install a Certificate | 47](#)
- [Rotate a Certificate | 57](#)

Overview

The gNOI CertificateManagement service in the `gnoi.certificate` package handles certificate management on the target network element. The proto definition file is located at <https://github.com/openconfig/gnoi/blob/master/cert/cert.proto>.

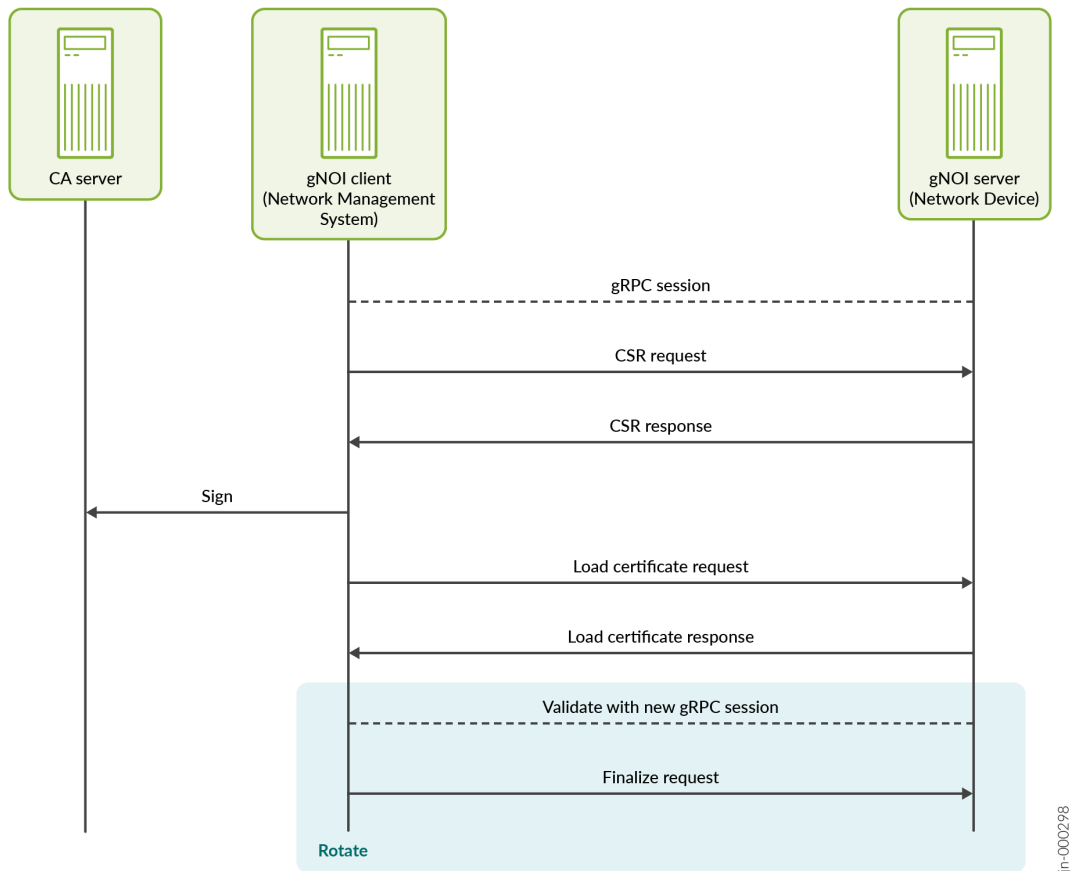
A public key infrastructure (PKI) supports the distribution and identification of public encryption keys, enabling users to both securely exchange data over networks such as the Internet and verify the identity of the other party. The Junos PKI enables you to manage public key certificates on Junos devices, including downloading, generating, and verifying certificates. The gNOI CertificateManagement service defines operations for certificate management, which is through the Junos PKI. The two main operations are:

- **Install**—Install a new certificate using a new certificate ID on the target network device. If the certificate ID already exists, the operation returns an error.
- **Rotate**—Replace an existing certificate, which already has an existing certificate ID, on the target network device. If the stream is broken or any steps fail during the process, then the device rolls back to the original certificate.

Figure 4 on page 43 outlines the workflow for the `Install()` and `Rotate()` operations. For both operations, the client can generate the certificate signing request (CSR) itself or request the target to generate the CSR. In either case, the client forwards the CSR to a certificate authority (CA) to request a digital certificate. The client then loads the certificate on the target, either with a new certificate ID for `Install()` operations or with an existing certificate ID for `Rotate()` operations. For `Rotate()` operations, the client should also validate any replacement certificates and finalize or cancel the `Rotate()` operation based on the success or failure of the validation. If the client cancels the operation, the server rolls back the certificate, key pair, and any CA bundle, if it is present in the request.

Starting in Junos OS Evolved Release 23.1R1, during an `Install()`, `Rotate()`, or `LoadCertificate()` operation, the gNOI server verifies the new end entity certificate using the corresponding CA certificate. Thus, the gNOI server's PKI must include the root CA certificate that verifies the new certificate. You can load the required CA certificate as part of the gNOI CA bundle, or you can load it separately. If the verification fails, the device does not install the new certificate.

Figure 4: gNOI CertificateManagement Service Install and Rotate Operations



The gNOI server supports only one global CA certificate bundle for gNOI services. When you use the gNOI CertificateManagement service to load the CA bundle, the following statements are applicable:

- The CertificateManagagment service always loads the CA certificate bundle using the ca-profile-group reserved identifier `gnoi-ca-bundle`.
- If you use the CertificateManagagment service to load the CA certificate bundle, the device implicitly uses mutual authentication and assumes the following configuration, even though it is not explicitly configured on the device.

```

[edit system services extension-service request-response grpc ssl]
mutual-authentication {
    certificate-authority gnoi-ca-bundle;
    client-certificate-request require-certificate-and-verify;
}

```

- If the CertificateManagagment service sends a request to load a new CA certificate bundle, the server clears the certificates for the previous CA bundle from the device and loads the new ones.
- If you use the CertificateManagagment service to load a CA certificate bundle and you also configure the [edit system services extension-service request-response grpc ssl mutual-authentication] statement hierarchy, then the configured statements take precedence.

Thus you can initially set up server-only authentication on the gNOI server and then use the Install() RPC to load the CA certificates. When you use gNOI to load the initial CA certificate bundle, the device performs the following steps:

- Adds the CA certificates in the Junos PKI.
- Automatically configures the gNOI CA certificate bundle at the [edit security pki] hierarchy level using the ca-profile-group identifier gnoi-ca-bundle.
- Switches from server-only authentication to mutual authentication.

```
[edit]
security {
  pki {
    ca-profile gnoi-ca-bundle_1 {
      ca-identity gnoi-ca-bundle_1;
    }
    ca-profile gnoi-ca-bundle_2 {
      ca-identity gnoi-ca-bundle_2;
    }
    ca-profile-group gnoi-ca-bundle {
      cert-base-count 2;
    }
  }
}
```

The Rotate() RPC does not support switching between authentication modes during the rotate operation. Thus, Rotate() does not support loading the CA certificate bundle on the gNOI server for the first time because it causes the device to switch from server-only authentication to mutual authentication during the operation. When the authentication mode changes, the network device must restart the gRPC stack and the connection is lost. If the stream is broken, the client cannot finalize the rotate request, and the device would roll back to the certificates that were in place before the Rotate() request was initiated.

NOTE: The hot-reloading statement at the [edit system services extension-service request-response grpc ssl] hierarchy level only maintains the gRPC session during a certificate update when the authentication mode remains unchanged during the operation. If the authentication mode switches, for example, from server-only to mutual authentication or vice versa, the client disconnects.

Supported RPCs

Table 12 on page 45 outlines the CertificateManagement service RPCs supported on Junos devices.

Table 12: Supported cert.proto RPCs

RPC	Description	Introduced in Release
CanGenerateCSR()	<p>Query the target device to determine if it can generate a certificate signing request (CSR) with the specified key type, key size, and certificate type. Supported values:</p> <ul style="list-style-type: none"> • Key type: KT_RSA • Key sizes: 1024, 2048, 4096 • Certificate type: CT_X509 <p>Returns True if the gNOI server supports the specific key type, key size, and certificate type.</p>	Junos OS Evolved 23.1R1
GenerateCSR()	Generate and return a certificate signing request (CSR).	Junos OS Evolved 22.2R1
GetCertificates()	Return the local certificates loaded on the target device.	Junos OS Evolved 22.2R1

Table 12: Supported cert.proto RPCs (*Continued*)

RPC	Description	Introduced in Release
Install()	Load a new certificate on the target device by creating a CSR request, generating a certificate based on the CSR, and loading the certificate using a new certificate ID.	Junos OS Evolved 22.2R1
LoadCertificate()	Load a certificate signed by a Certificate Authority (CA) on the target device.	Junos OS Evolved 22.2R1
LoadCertificateAuthorityBundle()	Load a CA certificate bundle on the target device.	Junos OS Evolved 22.2R1
RevokeCertificates()	Revoke the certificates with the specified certificate IDs on the target device.	Junos OS Evolved 23.1R1
Rotate()	Replace an existing certificate on the target device by creating a CSR request, generating a certificate based on the CSR, and loading the certificate using an existing certificate ID.	Junos OS Evolved 22.2R1

Network Device Configuration

To use gNOI certificate management services on Junos devices, you must configure the `use-pki` and `hot-reloading` statements for gRPC extensions services. In most cases, you configure the `use-pki` statement when you configure gRPC services on the network device. The `hot-reloading` statement is required to maintain the gRPC session when updating certificates that affect the session.

Before you begin:

- Configure gRPC services on the network device as described in ["Configure gRPC Services" on page 8](#).
- Configure the network management system to support gNOI operations as described in ["Configure gNOI Services" on page 24](#).

To configure the network device for CertificateManagement service operations:

1. Configure the device to use the PKI database for local certificates.

```
[edit system services extension-service request-response grpc ssl]  
user@host# set use-pki
```

2. Enable the device to reload certificates without terminating the gRPC session.

```
[edit system services extension-service request-response grpc ssl]  
user@host# set hot-reloading
```

3. Commit the configuration.

```
[edit system services extension-service request-response grpc ssl]  
user@host# commit
```

Install a Certificate

IN THIS SECTION

- [Example: Install a Certificate | 48](#)

You can use the `CertificateManagement` service `Install()` RPC to load a new certificate on the target device. When you install a new certificate using the `Install()` operation, you must specify a new certificate ID that does not already exist on the target device. You can also optionally load a CA certificate bundle as part of the `Install()` operation.

As part of the `Install()` operation, the device verifies the new certificate. Therefore, the Junos PKI must have the root CA certificate that verifies the new certificate. You can load the required CA certificate as part of the `Install()` operation, or you can load it separately, prior to the operation, if it is not already in the PKI.

If you install a new local certificate that will be used for gRPC session authentication, you must also update the gRPC server configuration on the device to use the new certificate ID.

Example: Install a Certificate

In this example, the gNOI server has been initially configured with a local certificate only and has not been configured to use mutual authentication. The gNOI client uses the `Install()` RPC to load a new local certificate and a CA certificate bundle on the device. After the CA bundle is loaded on the gNOI server, the server uses mutual authentication by default. The CA bundle includes the root CA certificate for the client certificate as well as the root CA certificate for the new server certificate.

The client executes the `gnoi_cert_install_certificate_csr.py` Python application, which performs the following operations:

- Requests the target to generate a CSR.
- Gets a signed certificate based on the CSR.
- Loads the new server certificate, the server's new root CA certificate, and the client's root CA certificate on the target network device.

The application uses the `InstallCertificateRequest` message with the appropriate parameters to define the requests for generating the CSR and loading the certificates. For each request, the application uses the `Install()` RPC to send the requests to the network device.

The `gnoi_cert_install_certificate_csr.py` application imports the `grpc_channel` module to establish the channel. The `grpc_channel` module is described in ["Configure gNOI Services" on page 24](#). The application's arguments are stored in the `args_cert_install_csr.txt` file. The application and argument files are presented here.

`gnoi_cert_install_certificate_csr.py`

```
"""gNOI Install Certificate utility."""

from __future__ import print_function
from __future__ import unicode_literals
import argparse
import logging
import re
from getpass import getpass
from subprocess import call

import cert_pb2
import cert_pb2_grpc
from grpc_channel import grpc_authenticate_channel_server_only
```

```

def get_args(parser):
    parser.add_argument('--server',
                        dest='server',
                        type=str,
                        default='localhost',
                        help='Server IP or name. Default is localhost.')

    parser.add_argument('--port',
                        dest='port',
                        nargs='?',
                        type=int,
                        default=50051,
                        help='Server port. Default is 50051')

    parser.add_argument('--client_key',
                        dest='client_key',
                        type=str,
                        default='',
                        help='Full path of the client private key. Default is ".")

    parser.add_argument('--client_cert',
                        dest='client_cert',
                        type=str,
                        default='',
                        help='Full path of the client certificate. Default is ".")

    parser.add_argument('--root_ca_cert',
                        dest='root_ca_cert',
                        required=True,
                        type=str,
                        help='Full path of the Root CA certificate.')

    parser.add_argument('--user_id',
                        dest='user_id',
                        required=True,
                        type=str,
                        help='User ID for RPC call credentials.')

    parser.add_argument('--type',
                        dest='type',
                        type=int,
                        default='1',
                        help='Certificate Type. Default is 1. Valid value is 1 (1 is CT_X509);

```

Invalid value is 0 (0 is CT_UNKNOWN).')

```

parser.add_argument('--min_key_size',
                    dest='min_key_size',
                    type=int,
                    default='2048',
                    help='Minimum key size. Default is 2048.')

parser.add_argument('--key_type',
                    dest='key_type',
                    type=int,
                    default='1',
                    help='Key Type. Default is 1 (KT_RSA); 0 is KT_UNKNOWN.')

parser.add_argument('--common_name',
                    dest='common_name',
                    type=str,
                    default='',
                    help='CN of the certificate')

parser.add_argument('--country',
                    dest='country',
                    type=str,
                    default='US',
                    help='Country name')

parser.add_argument('--state',
                    dest='state',
                    type=str,
                    default='CA',
                    help='State name')

parser.add_argument('--city',
                    dest='city',
                    type=str,
                    default='Sunnyvale',
                    help='City name')

parser.add_argument('--organization',
                    dest='organization',
                    type=str,
                    default='Acme',
                    help='Organization name')

```



```

parser.add_argument('--organizational_unit',
                    dest='organizational_unit',
                    type=str,
                    default='Test',
                    help='Organization unit name')

parser.add_argument('--ip_address',
                    dest='ip_address',
                    type=str,
                    default='',
                    help='IP address on the certificate')

parser.add_argument('--email_id',
                    dest='email_id',
                    type=str,
                    default='',
                    help='Email id')

parser.add_argument('--certificate_id',
                    dest='certificate_id',
                    required=True,
                    type=str,
                    help='Certificate id.')

parser.add_argument('--server_cert_private_key',
                    dest='server_cert_private_key',
                    type=str,
                    default='',
                    help='Server certificate private key')

parser.add_argument('--server_cert_public_key',
                    dest='server_cert_public_key',
                    type=str,
                    default='',
                    help='Server certificate public key')

parser.add_argument('--server_cert',
                    dest='server_cert',
                    type=str,
                    default='server_cert',
                    help='Server certificate')

```

```

parser.add_argument('--server_root_ca1',
                    dest='server_root_ca1',
                    type=str,
                    default='server_root_ca1',
                    help='Server Root CA')

parser.add_argument('--server_root_ca2',
                    dest='server_root_ca2',
                    type=str,
                    default='server_root_ca2',
                    help='Server Root CA')

parser.add_argument('--client_root_ca1',
                    dest='client_root_ca1',
                    type=str,
                    default='client_root_ca1',
                    help='Client Root CA')

parser.add_argument('--client_root_ca2',
                    dest='client_root_ca2',
                    type=str,
                    default='client_root_ca2',
                    help='Client Root CA')

parser.add_argument('--client_root_ca3',
                    dest='client_root_ca3',
                    type=str,
                    default='client_root_ca3',
                    help='Client Root CA')

parser.add_argument('--client_root_ca4',
                    dest='client_root_ca4',
                    type=str,
                    default='client_root_ca4',
                    help='Client Root CA')

args = parser.parse_args()
return args

def install_cert(channel, metadata, args):
    try:
        stub = cert_pb2_grpc.CertificateManagementStub(channel)

```

```

print("Executing GNOI::CertificateManagement::Install")

# Create request to generate certificate signing request (CSR)
it = []
req = cert_pb2.InstallCertificateRequest()

req.generate_csr.csr_params.type = args.type
req.generate_csr.csr_params.min_key_size = args.min_key_size
req.generate_csr.csr_params.key_type = args.key_type
req.generate_csr.csr_params.common_name = args.common_name
req.generate_csr.csr_params.country = args.country
req.generate_csr.csr_params.state = args.state
req.generate_csr.csr_params.city = args.city
req.generate_csr.csr_params.organization = args.organization
req.generate_csr.csr_params.organizational_unit = args.organizational_unit
req.generate_csr.csr_params.ip_address = args.ip_address
req.generate_csr.csr_params.email_id = args.email_id
req.generate_csr.certificate_id = args.certificate_id
it.append(req)

# Send request to generate CSR
for csr_rsp in stub.Install(iter(it), metadata=metadata, timeout=180):
    logging.info(csr_rsp)

# Write CSR to a file
with open('/home/lab/certs/server_temp.csr', "wb") as file:
    file.write(csr_rsp.generated_csr.csr.csr)

# If client connects to server IP address
# update openssl.cnf template to include subjectAltName IP extension
with open('/etc/pki/certs/openssl.cnf', 'r') as fd:
    data = fd.read()
data1 = re.sub(r'(subjectAltName=IP:).*',
               r'\g<1>'+args.ip_address, data)
with open('/home/lab/certs/openssl_temp.cnf', 'w') as fd:
    fd.write(data1)

# Generate certificate with v3 extensions
cmd = "openssl x509 -req -days 365 -in /home/lab/certs/server_temp.csr -CA /etc/pki/
certs/serverRootCA.crt -CAkey /etc/pki/certs/serverRootCA.key -CAcreateserial -out /home/lab/
certs/server_temp.crt -extensions v3_sign -extfile /home/lab/certs/openssl_temp.cnf -sha384"
decrypted = call(cmd, shell=True)

```

```

# Create request to install node certificate and CA certificates
print("\nExecuting GNOI::CertificateManagement::Install")
it = []
req = cert_pb2.InstallCertificateRequest()

# Import certificate and add to request
cert_data = bytearray(b'')
with open("/home/lab/certs/server_temp.crt", "rb") as file:
    cert_data = file.read()
req.load_certificate.certificate.type = args.type
req.load_certificate.certificate_id = args.certificate_id
req.load_certificate.certificate.certificate = cert_data

# Add client and server CA certificates to request
ca1 = req.load_certificate.ca_certificates.add()
ca1.type = args.type
ca1.certificate = open(args.client_root_ca1, 'rb').read()
ca2 = req.load_certificate.ca_certificates.add()
ca2.type = args.type
ca2.certificate = open(args.server_root_ca1, 'rb').read()

it.append(req)

# Send request to install node certificate and CA bundle
for rsp in stub.Install(iter(it), metadata=metadata, timeout=180):
    logging.info("Installing certificates: %s", rsp)
print("Install complete.")

except Exception as e:
    logging.error('Certificate install error: %s', e)
    print(e)

def main():
    parser = argparse.ArgumentParser(fromfile_prefix_chars='@')
    args = get_args(parser)

    grpc_server_password = getpass("gRPC server password for executing RPCs: ")
    metadata = [('username', args.user_id),
                ('password', grpc_server_password)]

    try:
        # Establish grpc channel to network device

```

```

        channel = grpc_authenticate_channel_server_only(
            args.server, args.port, args.root_ca_cert)
        install_cert(channel, metadata, args)
    except Exception as e:
        logging.error('Received error: %s', e)
        print(e)

if __name__ == '__main__':
    logging.basicConfig(filename='gnoi-testing.log',
                        format='%(asctime)s %(levelname)-8s %(message)s',
                        level=logging.INFO,
                        datefmt='%Y-%m-%d %H:%M:%S')

    main()

```

args_cert_install_csr.txt

```

--server=10.53.52.169
--port=50051
--root_ca_cert=/etc/pki/certs/serverRootCA.crt
--user_id=gnoi-user
--type=1
--min_key_size=2048
--key_type=1
--common_name=gnoi-server.example.com
--country=US
--state=CA
--city=Sunnyvale
--organization=Acme
--organizational_unit=testing
--ip_address=10.53.52.169
--email_id=test@example.com
--certificate_id=gnoi-server1
--client_root_ca1=/etc/pki/certs/clientRootCA.crt
--server_root_ca1=/etc/pki/certs/serverRootCA1.crt

```

Execute the Application

When the client executes the application, the application requests the CSR, gets the signed certificate, and loads the new server certificate and CA certificates on the target network device.

```
lab@gnoi-client:~/src/gnoi/proto$ python3 gnoi_cert_install_certificate_csr.py
@args_cert_install_csr.txt
gRPC server password for executing RPCs:
Creating channel
Executing GNOI::CertificateManagement::Install
Signature ok
subject=CN = gnoi-server.example.com, C = US, ST = CA, O = Acme, OU = testing
Getting CA Private Key

Executing GNOI::CertificateManagement::Install
Install complete.
```

After you install the new server certificate, you must configure the server to use that certificate ID for gRPC session authentication, as shown here. In addition, because the `Install()` operation loaded new CA certificates, the device implicitly uses mutual authentication. As a result, all subsequent gRPC sessions must include the client's certificate and key when establishing the channel.

```
user@gnoi-server> show configuration system services extension-service request-response grpc ssl
port 50051;
local-certificate gnoi-server1;
hot-reloading;
use-pki;
```

If you execute the application and provide a certificate ID that already exists on the server, the application returns an `ALREADY_EXISTS` error, because the `Install()` operation requires a new certificate ID.

```
lab@gnoi-client:~/src/gnoi/proto$ python3 gnoi_cert_install_certificate_csr.py
@args_cert_install_csr.txt
gRPC server password for executing RPCs:
Creating channel
Executing GNOI::CertificateManagement::Install
Signature ok
subject=CN = gnoi-server.example.com, C = US, ST = CA, O = Acme, OU = testing
Getting CA Private Key
```

```

Executing GNOI::CertificateManagement::Install
<_MultiThreadedRendezvous of RPC that terminated with:
    status = StatusCode.ALREADY_EXISTS
    details = ""
    debug_error_string = "{\"created\":\"@1652241881.676147097\",\"description\":\"Error received
from peer ipv4:10.53.52.169:50051\",\"file\":\"src/core/lib/surface/
call.cc\",\"file_line\":903,\"grpc_message\":\"\",\"grpc_status\":6}"

```

Rotate a Certificate

IN THIS SECTION

- [Example: Rotate a Certificate | 57](#)

You can use the `CertificateManagement` service `Rotate()` RPC to replace an existing certificate on the target device. When you replace an existing certificate using the `Rotate()` operation, you must load the certificate using the certificate ID that already exists on the target device. You can also optionally replace the existing gNOI CA certificate bundle as part of the `Rotate()` operation.

The `Rotate()` operation is similar to the `Install()` operation except that the `Rotate()` operation replaces an existing certificate instead of installing a new certificate. In addition, the client must validate that the updated certificate works and then finalize or cancel the `Rotate()` request based on the success or failure of the certificate validation.

As part of the `Rotate()` operation, the device verifies the new certificate. Therefore, the Junos PKI must have the root CA certificate that verifies the new certificate. You can load the required CA certificate as part of the `Rotate()` operation, or you can load it separately, prior to the operation, if it is not already in the PKI.

Example: Rotate a Certificate

In this example, the client executes the `gnoi_cert_rotate_certificate_csr.py` Python application, which performs the following operations:

- Requests the target to generate a CSR.
- Gets a signed certificate based on the CSR

- Replaces the node certificate and the gNOI CA bundle on the target network device.
- Validates the new certificate.
- Finalizes the Rotate operation.

The application uses the `RotateCertificateRequest` message with the appropriate parameters to define the requests for generating the CSR and loading the certificate and CA bundle. For each request, the application uses the `Rotate()` RPC to send the request to the network device. To enable the target device to verify the new node certificate, the application replaces the existing CA bundle with a new CA bundle. The bundle includes both the client CA certificate and the CA certificate required to verify the node certificate.

The application validates that the new certificate works by creating a new gRPC session with the network device and executing a simple `Time()` RPC, although you can test the session authentication with any RPC. The application finalizes the rotate request if the session is successfully established and cancels the rotate request if the session authentication fails.

The `gnoi_cert_rotate_certificate_csr.py` application imports the `grpc_channel` module to establish the channel. The `grpc_channel` module is described in ["Configure gNOI Services" on page 24](#). The application's arguments are stored in the `args_cert_rotate_csr.txt` file. The application and argument files are presented here.

`gnoi_cert_rotate_certificate_csr.py`

```
"""gNOI Rotate Certificate utility."""

from __future__ import print_function
from __future__ import unicode_literals
import argparse
import logging
import time
import re
import grpc
from getpass import getpass
from subprocess import call

import cert_pb2
import cert_pb2_grpc
import system_pb2
import system_pb2_grpc
from grpc_channel import grpc_authenticate_channel_mutual
```



```

def get_args(parser):
    parser.add_argument('--server',
                        dest='server',
                        type=str,
                        default='localhost',
                        help='Server IP or name. Default is localhost.')

    parser.add_argument('--port',
                        dest='port',
                        nargs='?',
                        type=int,
                        default=50051,
                        help='Server port. Default is 50051')

    parser.add_argument('--client_key',
                        dest='client_key',
                        type=str,
                        default='',
                        help='Full path of the client private key. Default is ".")

    parser.add_argument('--client_cert',
                        dest='client_cert',
                        type=str,
                        default='',
                        help='Full path of the client certificate. Default is ".")

    parser.add_argument('--root_ca_cert',
                        dest='root_ca_cert',
                        required=True,
                        type=str,
                        help='Full path of the Root CA certificate.')

    parser.add_argument('--user_id',
                        dest='user_id',
                        required=True,
                        type=str,
                        help='User ID for RPC call credentials.')

    parser.add_argument('--type',
                        dest='type',
                        type=int,
                        default='1',

```

```

        help='Certificate Type. Default is 1. Valid value is 1 (1 is CT_X509);  
Invalid value is 0 (0 is CT_UNKNOWN).')

```

```

parser.add_argument('--min_key_size',
                    dest='min_key_size',
                    type=int,
                    default='2048',
                    help='Minimum key size. Default is 2048.')

```

```

parser.add_argument('--key_type',
                    dest='key_type',
                    type=int,
                    default='1',
                    help='Key Type. Default is 1 (KT_RSA); 0 is KT_UNKNOWN.')

```

```

parser.add_argument('--common_name',
                    dest='common_name',
                    type=str,
                    default='',
                    help='CN of the certificate')

```

```

parser.add_argument('--country',
                    dest='country',
                    type=str,
                    default='US',
                    help='Country name')

```

```

parser.add_argument('--state',
                    dest='state',
                    type=str,
                    default='CA',
                    help='State name')

```

```

parser.add_argument('--city',
                    dest='city',
                    type=str,
                    default='Sunnyvale',
                    help='City name')

```

```

parser.add_argument('--organization',
                    dest='organization',
                    type=str,
                    default='Acme',

```

```

        help='Organization name')

parser.add_argument('--organizational_unit',
                    dest='organizational_unit',
                    type=str,
                    default='Test',
                    help='Organization unit name')

parser.add_argument('--ip_address',
                    dest='ip_address',
                    type=str,
                    default='',
                    help='IP address on the certificate')

parser.add_argument('--email_id',
                    dest='email_id',
                    type=str,
                    default='',
                    help='Email id')

parser.add_argument('--certificate_id',
                    dest='certificate_id',
                    required=True,
                    type=str,
                    help='Certificate id.')

parser.add_argument('--server_cert_private_key',
                    dest='server_cert_private_key',
                    type=str,
                    default='',
                    help='Server certificate private key')

parser.add_argument('--server_cert_public_key',
                    dest='server_cert_public_key',
                    type=str,
                    default='',
                    help='Server certificate public key')

parser.add_argument('--server_cert',
                    dest='server_cert',
                    type=str,
                    default='server_cert',
                    help='Server certificate')

```

```

parser.add_argument('--server_root_ca1',
                    dest='server_root_ca1',
                    type=str,
                    default='server_root_ca1',
                    help='Server Root CA')

parser.add_argument('--server_root_ca2',
                    dest='server_root_ca2',
                    type=str,
                    default='server_root_ca2',
                    help='Server Root CA')

parser.add_argument('--client_root_ca1',
                    dest='client_root_ca1',
                    type=str,
                    default='client_root_ca1',
                    help='Client Root CA')

parser.add_argument('--client_root_ca2',
                    dest='client_root_ca2',
                    type=str,
                    default='client_root_ca2',
                    help='Client Root CA')

parser.add_argument('--client_root_ca3',
                    dest='client_root_ca3',
                    type=str,
                    default='client_root_ca3',
                    help='Client Root CA')

parser.add_argument('--client_root_ca4',
                    dest='client_root_ca4',
                    type=str,
                    default='client_root_ca4',
                    help='Client Root CA')

parser.add_argument("--client_key_test",
                    dest='client_key_test',
                    type=str,
                    default='',
                    help='Full path of the test client private key. Default ""')

```

```

parser.add_argument("--client_cert_test",
                    dest='client_cert_test',
                    type=str,
                    default='',
                    help='Full path of the test client certificate. Default ""')

args = parser.parse_args()
return args

def rotate_cert(channel, metadata, args):
    try:
        result = ''
        stub = cert_pb2_grpc.CertificateManagementStub(channel)
        print("Executing GNOI::CertificateManagement::Rotate")

        # Create request to generate certificate signing request (CSR)
        it = []
        req = cert_pb2.RotateCertificateRequest()

        req.generate_csr.csr_params.type = args.type
        req.generate_csr.csr_params.min_key_size = args.min_key_size
        req.generate_csr.csr_params.key_type = args.key_type
        req.generate_csr.csr_params.common_name = args.common_name
        req.generate_csr.csr_params.country = args.country
        req.generate_csr.csr_params.state = args.state
        req.generate_csr.csr_params.city = args.city
        req.generate_csr.csr_params.organization = args.organization
        req.generate_csr.csr_params.organizational_unit = args.organizational_unit
        req.generate_csr.csr_params.ip_address = args.ip_address
        req.generate_csr.csr_params.email_id = args.email_id
        req.generate_csr.certificate_id = args.certificate_id
        it.append(req)

        # Send request to generate CSR
        print('Sending request for CSR')
        for csr_rsp in stub.Rotate(iter(it), metadata=metadata, timeout=30):
            logging.info(csr_rsp)

        # Write CSR to a file
        with open('/home/lab/certs/server_temp.csr', "wb") as file:
            file.write(csr_rsp.generated_csr.csr.csr)

```

```

# If client connects to server IP address
# update openssl.cnf template to include subjectAltName IP extension
with open('/etc/pki/certs/openssl.cnf', 'r') as fd:
    data = fd.read()
data1 = re.sub(r'(subjectAltName=IP:).*',
               r'\g<1>'+args.ip_address, data)
with open('/home/lab/certs/openssl_temp.cnf', 'w') as fd:
    fd.write(data1)

# Generate certificate with v3 extensions
cmd = "openssl x509 -req -days 365 -in /home/lab/certs/server_temp.csr -CA /etc/pki/
certs/serverRootCA.crt -CAkey /etc/pki/certs/serverRootCA.key -CAcreateserial -out /home/lab/
certs/server_temp.crt -extensions v3_sign -extfile /home/lab/certs/openssl_temp.cnf -sha384"
decrypted = call(cmd, shell=True)

# Create request to rotate node certificate and CA certificates
print("\nExecuting GNOI::CertificateManagement::Rotate")
it = []
req = cert_pb2.RotateCertificateRequest()

# Import certificate and add to request
with open("/home/lab/certs/server_temp.crt", "rb") as file:
    cert_data = file.read()
req.load_certificate.certificate.type = args.type
req.load_certificate.certificate.certificate = cert_data
req.load_certificate.certificate_id = args.certificate_id

# Add client and server CA certificates to request
ca1 = req.load_certificate.ca_certificates.add()
ca1.type = args.type
ca1.certificate = open(args.client_root_ca1, 'rb').read()
ca2 = req.load_certificate.ca_certificates.add()
ca2.type = args.type
ca2.certificate = open(args.server_root_ca1, 'rb').read()

it.append(req)

# Send request to replace node certificate and CA bundle
for rsp in stub.Rotate(iter(it), metadata=metadata, timeout=60):
    logging.info("Rotating certificates. %s", rsp)

# Validate certificates
print("Validating certificates")

```

```

time.sleep(5)
validate_rc = True

try:
    validate_channel = grpc_authenticate_channel_mutual(
        args.server, args.port, args.server_root_ca1, args.client_key_test,
        args.client_cert_test)
    validate_stub = system_pb2_grpc.SystemStub(validate_channel)
    validate_rsp = validate_stub.Time(
        request=system_pb2.TimeRequest(), metadata=metadata, timeout=60)
except grpc.RpcError as e:
    print("Validation failed with error:", e)
    validate_rc = False
    pass

if validate_rc:
    print("Finalizing certificate rotation.")
    it = []
    req = cert_pb2.RotateCertificateRequest()
    req.finalize_rotation.SetInParent()
    it.append(req)
    for rsp in stub.Rotate(iter(it), metadata=metadata, timeout=30):
        logging.info("Finalizing rotate. %s", rsp)
    logging.info(
        "Certificate validation succeeded. Certificate rotation finalized.")
    result = "Certificate rotation finalized."
else:
    print("Rolling back certificates.")
    logging.info(
        "Certificate validation failed. Rolling back to original certificates.")
    rsp.cancel()

except Exception as e:
    logging.error('Certificate rotate error: %s', e)
    print(e)
else:
    return result

def main():
    parser = argparse.ArgumentParser(fromfile_prefix_chars='@')
    args = get_args(parser)

```

```

grpc_server_password = getpass("gRPC server password for executing RPCs: ")
metadata = [('username', args.user_id),
            ('password', grpc_server_password)]

try:
    # Establish grpc channel to network device
    channel = grpc_authenticate_channel_mutual(
        args.server, args.port, args.root_ca_cert, args.client_key, args.client_cert)
    response = rotate_cert(channel, metadata, args)
    print(response)
except Exception as e:
    logging.error('Error: %s', e)
    print(e)

if __name__ == '__main__':
    logging.basicConfig(filename='gnoi-testing.log',
                        format='%(asctime)s %(levelname)-8s %(message)s',
                        level=logging.INFO,
                        datefmt='%Y-%m-%d %H:%M:%S')

    main()

```

args_cert_rotate_csr.txt

```

--server=10.53.52.169
--port=50051
--root_ca_cert=/etc/pki/certs/serverRootCA.crt
--client_key=/home/lab/certs/client.key
--client_cert=/home/lab/certs/client.crt
--user_id=gnoi-user
--type=1
--min_key_size=2048
--key_type=1
--common_name=gnoi-server.example.com
--country=US
--state=CA
--city=Sunnyvale
--organization=Acme
--organizational_unit=testing
--ip_address=10.53.52.169
--email_id=test@example.com

```



```
--certificate_id=gnoi-server
--client_root_ca1=/etc/pki/certs/clientRootCA.crt
--server_root_ca1=/etc/pki/certs/serverRootCA.crt
--client_key_test=/home/lab/certs/client.key
--client_cert_test=/home/lab/certs/client.crt
```

It's important to note that the `root_ca_cert` argument is the server's root CA certificate required for the initial channel credentials. The `server_root_ca1` argument is the root CA certificate corresponding to the server's new certificate. The Junos PKI must have the new root CA certificate in order to verify the new local certificate during the `Rotate()` operation. In addition, the channel credentials for the gRPC session that validates the new certificate use this root CA certificate. Although this example uses the same root CA certificate for the new and old server certificates, these might differ for another case.

Execute the Application

When the client executes the application, the application requests the CSR, gets the signed certificate, and loads the replacement certificate and CA bundle on the target network device. The application then validates the replacement certificate with a new gRPC session that executes a simple `Time()` RPC. Upon successful validation, the client finalizes the rotate request.

```
lab@gnoi-client:~/src/gnoi/proto$ python3 gnoi_cert_rotate_certificate_csr.py
@args_cert_rotate_csr.txt
gRPC server password for executing RPCs:
Creating channel
Executing GNOI::CertificateManagement::Rotate
Sending request for CSR
Signature ok
subject=CN = gnoi-server.example.com, C = US, ST = CA, O = Acme, OU = testing
Getting CA Private Key

Executing GNOI::CertificateManagement::Rotate
Validating certificates
Creating channel
Finalizing certificate rotation.
Certificate rotation finalized.
```

Revoke a Certificate

IN THIS SECTION

- [Example: Revoke a Certificate](#) | 68

A gNOI client can use the `RevokeCertificates()` RPC to remove one or more certificates from the target device. The client includes a `RevokeCertificatesRequest` message with the list of certificate IDs to revoke.

When the gNOI server receives the `RevokeCertificates()` request, it processes each certificate ID in the list as follows:

- If the certificate is present and the revocation is successful, the device removes the certificate from the file system and the Junos PKI and adds the certificate ID to the list of successfully revoked certificates.
- If the certificate is present and the revocation fails, the device includes the certificate ID and the reason for the failure in the certificate revocation error list.
- If the certificate is not present, the device considers the revocation operation successful and adds the certificate ID to the list of successfully revoked certificates.

NOTE: If the request revokes the certificate used for the current session, the session is not affected.

After processing the request, the gNOI server returns a `RevokeCertificatesResponse` message that includes:

- A list of successfully revoked certificate IDs.
- A list of revocation errors containing the certificate ID and the reason for the failure.

Example: Revoke a Certificate

In this example, the client executes the `gnoi_cert_revoke_certificates.py` Python application, which revokes two certificates on the server. The first certificate ID is a valid identifier on the device. The second certificate ID is an identifier that does not exist on the device.

The application uses the `RevokeCertificatesRequest` message with the appropriate parameters to define the request. The application sends the `RevokeCertificates()` RPC to the network device to perform the operation.

The `gnoi_cert_revoke_certificates.py` application imports the `grpc_channel` module to establish the channel. The `grpc_channel` module is described in ["Configure gNOI Services" on page 24](#). The application's arguments are stored in the `args_cert_revoke_certificates.txt` file. The application and argument files are presented here.

`gnoi_cert_revoke_certificates.py`

```
"""gNOI Revoke Certificates utility."""
from __future__ import print_function
from __future__ import unicode_literals
import argparse
import logging
from getpass import getpass

import cert_pb2
import cert_pb2_grpc
from grpc_channel import grpc_authenticate_channel_mutual

def get_args(parser):
    parser.add_argument('--server',
                        dest='server',
                        type=str,
                        default='localhost',
                        help='Server IP or name. Default is localhost.')

    parser.add_argument('--port',
                        dest='port',
                        nargs='?',
                        type=int,
                        default=50051,
                        help='Server port. Default is 50051')

    parser.add_argument('--client_key',
                        dest='client_key',
                        type=str,
                        default='',
                        help='Full path of the client private key. Default is ""')
```

```

parser.add_argument('--client_cert',
                    dest='client_cert',
                    type=str,
                    default='',
                    help='Full path of the client certificate. Default is ".")

parser.add_argument('--root_ca_cert',
                    dest='root_ca_cert',
                    required=True,
                    type=str,
                    help='Full path of the Root CA certificate.')

parser.add_argument('--user_id',
                    dest='user_id',
                    required=True,
                    type=str,
                    help='User ID for RPC call credentials.')

parser.add_argument('--certificate_id1',
                    dest='certificate_id1',
                    required=True,
                    type=str,
                    help='Certificate id.')

parser.add_argument('--certificate_id2',
                    dest='certificate_id2',
                    type=str,
                    help='Certificate id.')

args = parser.parse_args()
return args

def revoke_cert(channel, metadata, args):
    try:
        stub = cert_pb2_grpc.CertificateManagementStub(channel)
        print("Executing GNOI::CertificateManagement::RevokeCertificates")

        # Create request to revoke certificates
        req = cert_pb2.RevokeCertificatesRequest()
        req.certificate_id.append(args.certificate_id1)
        req.certificate_id.append(args.certificate_id2)

```

```

        # Send request to revoke certificates
        logging.info("Sending RevokeCertificates request.")
        rsp = stub.RevokeCertificates(req, metadata=metadata, timeout=60)
        logging.info(rsp)
        print("rsp:\n%s" %rsp)

    except Exception as e:
        logging.error('Error: %s', e)
        print(e)

def main():
    parser = argparse.ArgumentParser(fromfile_prefix_chars='@')
    args = get_args(parser)

    grpc_server_password = getpass("gRPC server password for executing RPCs: ")
    metadata = [('username', args.user_id),
                ('password', grpc_server_password)]

    try:
        # Establish grpc channel to network device
        channel = grpc_authenticate_channel_mutual(
            args.server, args.port, args.root_ca_cert, args.client_key, args.client_cert)

        revoke_cert(channel, metadata, args)
    except Exception as e:
        logging.error('Received error: %s', e)
        print(e)

if __name__ == '__main__':
    logging.basicConfig(filename='gnoi-testing.log',
                        format='%(asctime)s %(levelname)-8s %(message)s',
                        level=logging.INFO,
                        datefmt='%Y-%m-%d %H:%M:%S')

    main()

```

args_cert_revoke_certificates.txt

```

--server=10.53.52.169
--port=50051

```

```
--root_ca_cert=/etc/pki/certs/serverRootCA.crt
--client_key=/home/lab/certs/client.key
--client_cert=/home/lab/certs/client.crt
--user_id=gnoi-user
--certificate_id1=gnoi-server
--certificate_id2=id-does-not-exist
```

Execute the Application

When the client executes the application, the application instructs the target device to revoke the specified certificates. The device returns a list of successfully revoked certificates and any errors. The device deems the operation successful for both the valid certificate ID as well as for the certificate ID that does not currently exist on the device.

```
lab@gnoi-client:~/src/gnoi/proto$ python3 gnoi_cert_revoke_certificates.py
@args_cert_revoke_certificates.txt
gRPC server password for executing RPCs:
Creating channel
Executing GNOI::CertificateManagement::RevokeCertificates
rsp:
revoked_certificate_id: "gnoi-server"
revoked_certificate_id: "id-does-not-exist"
```

Release History Table

Release	Description
23.1R1-EVO	Starting in Junos OS Evolved Release 23.1R1, the Install(), Rotate(), and LoadCertificate() operations verify the new certificate as part of the operation.

gNOI Diagnostic (Diag) Service

SUMMARY

Use the gNOI diagnostic (Diag) service to test the reliability of a link between two devices.

IN THIS SECTION

- [Overview | 73](#)
- [Supported RPCs | 75](#)
- [Network Device Configuration | 75](#)
- [Example: Run a BERT | 76](#)

Overview

Use the Diag service RPCs to perform a bit error rate test (BERT) on a pair of connected ports. The Diag service proto definition file is located at <https://github.com/openconfig/gnoi/blob/master/diag/diag.proto>.

A BERT, also known as a pseudo-random binary sequence (PRBS) test, tests the reliability of a link. The `StartBERT()` gNOI RPC initiates a bidirectional BERT on a pair of connected physical interfaces. The devices exchange a set pattern of 1s and 0s across the link. The devices compare the received message to the sent message and count the number of errors. The lower the number of errors, the higher the quality of the link.

You must run a gNOI BERT on both sides of the link so the devices can compare results. The link you are testing goes down during the BERT and comes back up after the BERT ends. However, if one of the devices where you are running the BERT reboots, the link remains down unless you stop the BERT on the other device.

You can choose the test pattern from several predetermined types. BERT or PRBS test patterns are titled in the form PRBSx, where x is a number. Junos devices support the following test patterns for gNOI BERTs:

- PRBS7
- PRBS9
- PRBS15
- PRBS23
- PRBS31

You must give each gNOI BERT a unique operation ID. The RPCs to start the BERT, stop the BERT, and get the BERT results are linked by the BERT operation ID. When you run a new BERT, you must change the operation ID to a new string. Because the RPCs identify each BERT by its operation ID, you can run multiple BERTs on different interfaces with the same ID.

The device keeps the results of the last 5 BERT operations. However, the saved BERT results are not persistent. They are lost if the system reboots.

To view the result of a specific saved BERT operation, send the `GetBERTResultRequest` message for the desired BERT operation ID and set the `result_from_all_ports` field to `False`. To view all request results for different IDs, set the `result_from_all_ports` field in the `GetBERTResultRequest` message to `True`.

When you run the `GetBERTResult()` RPC on a device, the RPC displays the number of mismatched bits that particular device detected during the BERT. Since the RPC does not have pass or fail criteria configured, it is up to the user to evaluate the results. You might see a high number of errors for several reasons, including:

- The quality of the link is poor.
- One of the devices went offline during the BERT.
- The BERT only ran on one device.
- The BERT did not start and stop on both devices simultaneously.

To avoid the last error, we recommend sending the `StartBERT()` RPC to both devices simultaneously. If you start a BERT on one device before the other, the first device doesn't receive a response until the BERT starts on the other device. The first device records the lack of response as mismatched bits. The first device continues to report errors until BERT starts on the second device. If it is not possible to start the BERT simultaneously, we recommend running the `GetBERTResult()` RPC on the device that started the BERT last. Since the BERT was already running on the first device, the second device should not report any false missing bits.

Supported RPCs

Table 13: Supported diag.proto RPCs

RPC	Description	Introduced in Release
StartBERT()	<p>Start a BERT on a set of ports. Junos devices support the following PRBS patterns for gNOI BERTs:</p> <ul style="list-style-type: none"> • PRBS7 • PRBS9 • PRBS15 • PRBS23 • PRBS31 	Junos OS Evolved 22.2R1
StopBert()	Stop an already in-progress BERT on a set of ports.	Junos OS Evolved 22.2R1
GetBERTResult()	Get BERT results during the BERT or after it completes.	Junos OS Evolved 22.2R1

Network Device Configuration

- Configure gRPC services on the network device as described in ["Configure gRPC Services" on page 8](#).
- Configure the network management system to support gNOI operations as described in ["Configure gNOI Services" on page 24](#).
- For the link you want to run the BERT on, configure the server and peer interfaces speeds to match. BERT only runs if the interface speeds match.

Example: Run a BERT

IN THIS SECTION

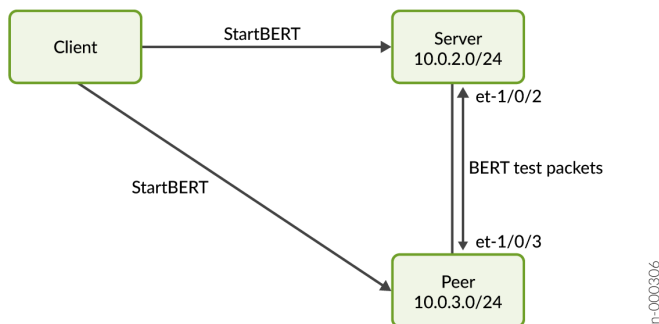
- [gnoi_bert_client.py | 77](#)
- [input_bert_start.json | 81](#)
- [input_bert_get.json | 82](#)
- [Execute the Application | 82](#)

NOTE: While a BERT is in progress on an interface, the physical link on that interface goes down.

After you have configured the gNOI client and server, you are ready to write and execute your application to run the BERT. In this example, the client executes the `gnoi_bert_client.py` Python application to test a link between the server and a peer device. The `gnoi_bert_client.py` application can start the BERT, stop the BERT, or get the BERT results depending on the arguments.

First, the client uses `gnoi_bert_client.py` to send the `StartBERT()` RPC to start the BERT on the server and the peer. While the BERT is running, the server and peer exchange BERT test packets across the link between the `et-1/0/2` and `et-1/0/3` interfaces.

Figure 5: Network Topology During the BERT



The BERT ends after the set time expires. Then the client executes the application a second time with the `GetBERTResult()` RPC to get the BERT results from the server.

The parameters for the StartBERTRequest message are stored in the **input_bert_start.json** JSON file. This file specifies that the BERT should run for 60 seconds using PRBS pattern 31. The parameters for the GetBERTResultRequest message are stored in the **input_bert_get.json** JSON file. The `result_from_all_ports` field is set to False, so the GetBERTResult() RPC only retrieves the result for this particular BERT from this port. The BERT operation ID is BERT-operation id 1 in both JSON files.

The application imports the `grpc_channel` module to establish the channel. The `grpc_channel` module is described in ["Configure gNOI Services" on page 24](#). The application file and JSON files are presented here.

gnoi_bert_client.py

```
"""gRPC gNOI BERT utility."""

from __future__ import print_function
import argparse
import json
import sys
import logging
from getpass import getpass

import diag_pb2
import diag_pb2_grpc

from grpc_channel import grpc_authenticate_channel_mutual
from google.protobuf.json_format import MessageToJson
from google.protobuf.json_format import ParseDict

def get_args(parser):
    # Main arguments
    parser.add_argument('--server',
                        dest='server',
                        type=str,
                        default='localhost',
                        help='Server IP or name. Default is localhost')

    parser.add_argument('--port',
                        dest='port',
                        nargs='?',
                        type=int,
                        default=50051,
```

```

        help='The server port. Default is 50051')

parser.add_argument('--client_key',
                    dest='client_key',
                    type=str,
                    default='',
                    help='Full path of the client private key. Default ""')

parser.add_argument('--client_cert',
                    dest='client_cert',
                    type=str,
                    default='',
                    help='Full path of the client certificate. Default ""')

parser.add_argument('--root_ca_cert',
                    dest='root_ca_cert',
                    required=True,
                    type=str,
                    help='Full path of the Root CA certificate.')

parser.add_argument('--user_id',
                    dest='user_id',
                    required=True,
                    type=str,
                    help='User ID for RPC call credentials.')

# BERT arguments
parser.add_argument('--input_file',
                    dest='input_file',
                    type=str,
                    default=None,
                    help='Input JSON file to convert to a Message Object. Default NULL
string')

parser.add_argument('--output_file',
                    dest='output_file',
                    type=str,
                    default=None,
                    help='Output file. Default NULL string')

parser.add_argument('--message',
                    dest='message',
                    type=str,

```

```

        default=None,
        help='The type of Message Object. Must correspond to input file JSON.
Default NULL string')

    args = parser.parse_args()
    return args

def check_inputs(args):
    # Check each of the default=None arguments
    if args.server is None:
        print('\nFAIL: --server is not passed in\n')
        return False
    if args.port is None:
        print('\nFAIL: server port (--port) is not passed in\n')
        return False
    if args.input_file is None:
        print('\nFAIL: --input_file is not passed in\n')
        return False
    if args.output_file is None:
        print('\nFAIL: --output_file is not passed in\n')
        return False
    if args.message is None:
        print('\nFAIL: --message is not passed in\n')
        return False

    return True

# Create a dictionary where top-level keys match what is passed in via args.message
# The values are pointers to the relevant classes and method names needed to build/send message
objects
MESSAGE_RELATED_OBJECTS = {
    'StartBERTRequest': {
        'msg_type': diag_pb2.StartBERTRequest,
        'grpc': diag_pb2_grpc.DiagStub,
        'method': 'StartBERT'
    },
    'StopBERTRequest': {
        'msg_type': diag_pb2.StopBERTRequest,
        'grpc': diag_pb2_grpc.DiagStub,
        'method': 'StopBERT'
    },
}

```

```

'GetBERTResultRequest': {
    'msg_type': diag_pb2.GetBERTResultRequest,
    'grpc': diag_pb2_grpc.DiagStub,
    'method': 'GetBERTResult'
}
}

def send_rpc(channel, metadata, args):
    if not check_inputs(args):
        print('\nFAIL: One of the inputs was not as expected.\n')
        return False

    print('\nMessage Type is {}'.format(args.message))

    # Message objects to send
    msg_object_list = []

    with open(args.input_file) as json_file:
        user_input = json.load(json_file)

    # Choose the Request Message Object type based on the --message type passed
    request_message = MESSAGE_RELATED_OBJECTS[args.message]['msg_type']()

    # Convert the dictionary to the type of message object specified by request_message
    try:
        msg_object_list.append(ParseDict(user_input, request_message))
    except Exception as error:
        print('\n\nError:\n{}'.format(error), file=sys.stderr)
        raise

    # Assemble callable object to use for sending, e.g. diag_pb2_grpc.DiagStub(channel).StartBERT
    method = MESSAGE_RELATED_OBJECTS[args.message]['method']
    send_message = getattr(
        MESSAGE_RELATED_OBJECTS[args.message]['grpc'](channel), method)

    # send the Request Object(s)
    for msg_object in msg_object_list:
        resp = send_message(msg_object, metadata=metadata)
        print('\n\nResponse:\n{}'.format(resp))
        print('=====')

    resp_json = MessageToJson(resp)

```

```

print('\n\nResponse JSON:\n{}'.format(resp_json))

with open(args.output_file, 'w') as data:
    data.write(str(resp_json))

return True

def main():
    parser = argparse.ArgumentParser()
    args = get_args(parser)

    grpc_server_password = getpass("gRPC server password for executing RPCs: ")
    metadata = [('username', args.user_id),
                ('password', grpc_server_password)]

    try:
        # Establish grpc channel to network device
        channel = grpc_authenticate_channel_mutual(
            args.server, args.port, args.root_ca_cert, args.client_key, args.client_cert)
        response = send_rpc(channel, metadata, args)
    except Exception as e:
        logging.error('Received error: %s', e)
        print(e)

if __name__ == '__main__':
    logging.basicConfig(filename='gnoi-testing.log',
                        format='%(asctime)s %(levelname)-8s %(message)s',
                        level=logging.INFO,
                        datefmt='%Y-%m-%d %H:%M:%S')

    main()

```

input_bert_start.json

```

{
  "bert_operation_id": "BERT-operation id 1",
  "per_port_requests": [
    {
      "interface": {
        "origin": "origin",

```

```

        "elem": [
            {"name": "interfaces"},
            {"name": "interface", "key": {"name": "et-1/0/2"}}
        ],
        "prbs_polynomial": "PRBS_POLYNOMIAL_PRBS31",
        "test_duration_in_secs": "60"
    }
]
}

```

input_bert_get.json

```

{
    "bert_operation_id": "BERT-operation id 1",
    "result_from_all_ports": false,
    "per_port_requests": [
        {
            "interface": {
                "origin": "origin",
                "elem": [
                    {"name": "interfaces"},
                    {"name": "interface", "key": {"name": "et-1/0/2"}}
                ]
            }
        }
    ]
}

```

Execute the Application

1. From the client, run the `gnoi_bert_client.py` application to start the BERT on the peer (not shown). Then run the `gnoi_bert_client.py` application to start the BERT on the server (shown below). To start the BERT, set message to `StartBERTRequest` and set `input_file` to the **input_bert_start.json** file path. For each device, the input file should specify the interface tested on that device. The `BERT_STATUS_OK` status indicates that the BERT started successfully.

```

lab@gnoi-client:~/src/gnoi/proto$ python3 gnoi_bert_client.py --server 10.0.2.1 --port 50051
--root_ca_cert /etc/pki/certs/serverRootCA.crt --client_key /home/lab/certs/client.key --

```



```
client_cert /home/lab/certs/client.crt --user_id gnoi-user --message StartBERTRequest --
input_file diag/input_bert_start.json --output_file diag/output/bert-start-resp1.json
gRPC server password for executing RPCs:
```

Message Type is StartBERTRequest

Response:

```
bert_operation_id: "BERT-operation id 1"
per_port_responses {
  interface {
    origin: "origin"
    elem {
      name: "interfaces"
    }
    elem {
      name: "interface"
      key {
        key: "name"
        value: "et-1/0/2"
      }
    }
  }
  status: BERT_STATUS_OK
}
```

=====

Response JSON:

```
{
  "bertOperationId": "BERT-operation id 1",
  "perPortResponses": [
    {
      "interface": {
        "origin": "origin",
        "elem": [
          {
            "name": "interfaces"
          },
          {
            "name": "interface",
            "key": {
```

```

        "name": "et-1/0/2"
    }
}
],
{
    "status": "BERT_STATUS_OK"
}
]
}

```

2. (Optional) While you are running the BERT, use the `show interfaces` command on the server or peer device to view the ongoing BERT results. When a BERT is running, the PRBS Mode is Enabled. The output in this example has been truncated for clarity.

```

user@server> show interfaces
et-1/0/2
Physical interface: et-1/0/2, Enabled, Physical link is Down
  Interface index: 1018, SNMP ifIndex: 534
[...]
  PRBS Mode : Enabled
  PRBS Pattern : 31
  PRBS Statistics
    Lane 0 : Error Bits : 0 Total Bits : 200000000000 Monitored Seconds : 8
    Lane 1 : Error Bits : 0 Total Bits : 200000000000 Monitored Seconds : 8
    Lane 2 : Error Bits : 0 Total Bits : 200000000000 Monitored Seconds : 8
    Lane 3 : Error Bits : 0 Total Bits : 200000000000 Monitored Seconds : 8
  Interface transmit statistics: Disabled
  Link Degradate :
    Link Monitoring : Disable
[...]

```

3. After the BERT is finished, run the `gnoi_bert_client.py` application again with `message` set to `GetBERTResultRequest` and `input_file` set to the `input_bert_get.json` file path to get the results of the test. In this example, the BERT found zero errors during a one-minute test.

```

lab@gnoi-client:~/src/gnoi/proto$ python3 gnoi_bert_client.py --server 10.0.2.1 --port 50051
--root_ca_cert /etc/pki/certs/serverRootCA.crt --client_key /home/lab/certs/client.key --
client_cert /home/lab/certs/client.crt --user_id gnoi-user --message GetBERTResultRequest --
input_file diag/input_bert_get.json --output_file diag/output/bert-get-resp1.json
gRPC server password for executing RPCs:

```

Message Type is GetBERTResultRequest

Response:

```
per_port_responses {
  interface {
    origin: "origin"
    elem {
      name: "interfaces"
    }
    elem {
      name: "interface"
      key {
        key: "name"
        value: "et-1/0/2"
      }
    }
  }
}
status: BERT_STATUS_OK
bert_operation_id: "BERT-operation id 1"
prbs_polynomial: PRBS_POLYNOMIAL_PRBS31
last_bert_start_timestamp: 1652379568178
last_bert_get_result_timestamp: 1652379688037
peer_lock_established: true
error_count_per_minute: 0
total_errors: 0
}
```

=====

Response JSON:

```
{
  "perPortResponses": [
    {
      "interface": {
        "origin": "origin",
        "elem": [
          {
            "name": "interfaces"
          },
          {
            "name": "interface",
```

```

        "key": {
            "name": "et-1/0/2"
        }
    },
    "status": "BERT_STATUS_OK",
    "bertOperationId": "BERT-operation id 1",
    "prbsPolynomial": "PRBS_POLYNOMIAL_PRBS31",
    "lastBertStartTimestamp": "1652379568178",
    "lastBertGetResultTimestamp": "1652379688037",
    "peerLockEstablished": true,
    "errorCountPerMinute": [
        0
    ],
    "totalErrors": "0"
}
]
}

```

The BERT completed successfully and shows the quality of the link is good.

gNOI File Service

SUMMARY

Use the gNOI File service to manage files on a network device.

IN THIS SECTION

- Supported RPCs | 87
- Network Device Configuration | 87
- Example: Get File | 88
- Example: Put File | 92

Use the File service RPCs to transfer and delete files or retrieve information about files. The proto definition file is located at <https://github.com/openconfig/gnoi/blob/master/file/file.proto>.

Supported RPCs

Table 14: Supported file.proto RPCs

RPC	Description	Introduced in Release
Get()	<p>Read and stream the contents of a file from the target.</p> <p>The file is streamed by sequential messages, each message containing up to 64KB of data. A final message is sent prior to closing the stream that contains the hash of the data sent. The operation returns an error if the file does not exist or there is an error reading the file.</p>	Junos OS Evolved 22.2R1
Put()	<p>Stream data to a file on the target.</p> <p>The file is sent in sequential messages, each message containing up to 64KB of data. A final message must be sent that includes the hash of the data.</p> <p>The operation returns an error if the location does not exist or there is an error writing the data. If no checksum is received, the target removes the partially transmitted file. A failure will not alter any existing file of the same name.</p>	Junos OS Evolved 22.2R1
Remove()	<p>Remove the specified file from the target. The operation returns an error if the file does not exist, if the file path resolves to a directory, or if the remove operation encounters an error.</p>	Junos OS Evolved 22.2R1
Stat()	<p>Return metadata about a file on the target device. The operation returns an error if the file does not exist or there is an error accessing the metadata.</p>	Junos OS Evolved 22.2R1

Network Device Configuration

Before you begin:

- Configure gRPC services on the network device as described in ["Configure gRPC Services" on page 8](#).
- Configure the network management system to support gNOI operations as described in ["Configure gNOI Services" on page 24](#).

To perform file operations on the target device, the client must have the appropriate permissions to interact with the file system.

Example: Get File

IN THIS SECTION

- [gnoi_file_get.py | 88](#)
- [args_file_get.txt | 91](#)
- [Execute the Application | 92](#)

This example provides a simple Python application, `gnoi_file_get.py`, to download a file from the target device to the local network management system.

The application imports the `grpc_channel` module to establish the channel. The `grpc_channel` module is described in ["Configure gNOI Services" on page 24](#). The application's arguments are stored in the `args_file_get.txt` file. The application and argument files are presented here.

`gnoi_file_get.py`

```
"""gNOI Get File utility."""

from __future__ import print_function
import argparse
import hashlib
import logging
from getpass import getpass

import file_pb2
import file_pb2_grpc
from grpc_channel import grpc_authenticate_channel_mutual

def get_args(parser):
    parser.add_argument('--server',
                        dest='server',
                        type=str,
                        default='localhost',
                        help='Server IP or name. Default is localhost')

    parser.add_argument('--port',
```

```

        dest='port',
        nargs='?',
        type=int,
        default=50051,
        help='The server port. Default is 50051')

parser.add_argument('--client_key',
                    dest='client_key',
                    type=str,
                    default='',
                    help='Full path of the client private key. Default ""')

parser.add_argument('--client_cert',
                    dest='client_cert',
                    type=str,
                    default='',
                    help='Full path of the client certificate. Default ""')

parser.add_argument('--root_ca_cert',
                    dest='root_ca_cert',
                    required=True,
                    type=str,
                    help='Full path of the Root CA certificate.')

parser.add_argument('--user_id',
                    dest='user_id',
                    required=True,
                    type=str,
                    help='User ID for RPC call credentials.')

parser.add_argument('--dest_file',
                    dest='dest_file',
                    type=str,
                    default='',
                    help='Full path for destination file. Default ""')

parser.add_argument('--source_file',
                    dest='source_file',
                    type=str,
                    default='',
                    help='Full path of source file to retrieve. Default ""')

args = parser.parse_args()

```

```

return args

def send_rpc(channel, metadata, args):
    stub = file_pb2_grpc.FileStub(channel)
    print("Executing GNOI::File::Get")

    # Prepare hash generator
    gen_hash = hashlib.sha256()

    # Get File
    req = file_pb2.GetRequest()
    req.remote_file = args.source_file

    hashvalue = None
    hm = None
    count = 1
    with open(args.dest_file, "wb") as file:
        # Read data in 64 KB chunks and calculate checksum and data messages
        print("Retrieving file")
        try:
            for msg in stub.Get(req, metadata=metadata, timeout=120):
                if msg.WhichOneof('response') == "contents":
                    count = count + 1
                    file.write(msg.contents)
                    gen_hash.update(msg.contents)
                else:
                    hashvalue = msg.hash.hash
                    hm = msg.hash.method
            print("File transfer complete: ", args.dest_file)
        except Exception as e:
            logging.error("Get() operation error. %s", e)
            print(e)
        else:
            file.close()
            ehashvalue = gen_hash.hexdigest().encode()
            if (ehashvalue != hashvalue):
                raise ValueError(
                    'Hash value mismatch. Expected "%s", got "%s"' % (ehashvalue, hashvalue))
            if (hm != 1):
                raise ValueError(
                    'Hash method mismatch. Expected "1", got "%s"' % (hm))
            logging.info("Downloaded file: %s", args.dest_file)

```



```

def main():
    parser = argparse.ArgumentParser(fromfile_prefix_chars='@')
    args = get_args(parser)

    grpc_server_password = getpass("gRPC server password for executing RPCs: ")
    metadata = [('username', args.user_id),
                ('password', grpc_server_password)]

    try:
        # Establish grpc channel to network device
        channel = grpc_authenticate_channel_mutual(
            args.server, args.port, args.root_ca_cert, args.client_key, args.client_cert)
        send_rpc(channel, metadata, args)
    except Exception as e:
        logging.error('Received error: %s', e)
        print(e)

if __name__ == '__main__':
    logging.basicConfig(filename='gnoi-testing.log',
                        format='%(asctime)s %(levelname)-8s %(message)s',
                        level=logging.INFO,
                        datefmt='%Y-%m-%d %H:%M:%S')

    main()

```

args_file_get.txt

```

--server=10.53.52.169
--port=50051
--root_ca_cert=/etc/pki/certs/serverRootCA.crt
--client_key=/home/lab/certs/client.key
--client_cert=/home/lab/certs/client.crt
--user_id=gnoi-user
--source_file=/var/log/messages
--dest_file=downloads/10.53.52.169-messages

```

Execute the Application

When the client executes the application, it transfers the specified file from the target device to the local device.

```
lab@gnoi-client:~/src/gnoi/proto$ python3 gnoi_file_get.py @args_file_get.txt
gRPC server password for executing RPCs:
Creating channel
Executing GNOI::File::Get
Retrieving file
File transfer complete:  downloads/10.53.52.169-messages
```

Example: Put File

IN THIS SECTION

- [gnoi_file_put.py | 92](#)
- [args_file_put.txt | 96](#)
- [Execute the Application | 97](#)

This example provides a simple Python application, `gnoi_file_put.py`, to upload a file from the local network management system to the target device.

The application imports the `grpc_channel` module to establish the channel. The `grpc_channel` module is described in ["Configure gNOI Services" on page 24](#). The application's arguments are stored in the `args_file_put.txt` file. The application and argument files are presented here.

`gnoi_file_put.py`

```
"""gNOI Put File utility."""

from __future__ import print_function
import argparse
import hashlib
import logging
```

```

import sys
from functools import partial
from getpass import getpass

import file_pb2
import file_pb2_grpc
from grpc_channel import grpc_authenticate_channel_mutual

MAX_BYTES = 65536

def get_args(parser):
    parser.add_argument('--server',
                        dest='server',
                        type=str,
                        default='localhost',
                        help='Server IP or name. Default is localhost')

    parser.add_argument('--port',
                        dest='port',
                        nargs='?',
                        type=int,
                        default=50051,
                        help='The server port. Default is 50051')

    parser.add_argument('--client_key',
                        dest='client_key',
                        type=str,
                        default='',
                        help='Full path of the client private key. Default ""')

    parser.add_argument('--client_cert',
                        dest='client_cert',
                        type=str,
                        default='',
                        help='Full path of the client certificate. Default ""')

    parser.add_argument('--root_ca_cert',
                        dest='root_ca_cert',
                        required=True,
                        type=str,
                        help='Full path of the Root CA certificate.')

```

```

parser.add_argument('--user_id',
                    dest='user_id',
                    required=True,
                    type=str,
                    help='User ID for RPC call credentials.')

parser.add_argument('--dest_file',
                    dest='dest_file',
                    type=str,
                    default='',
                    help='Full path for destination file. Default ""')

parser.add_argument('--dest_file_mode',
                    dest='dest_file_mode',
                    type=int,
                    default=600,
                    help='Destination file mode (file permissions in octal). Default 600')

parser.add_argument('--hash_method',
                    dest='hash_method',
                    type=str,
                    default='unspecified',
                    help='Hash method. Valid values are md5, sha256, sha512, or
unspecified. Default: unspecified')

parser.add_argument('--source_file',
                    dest='source_file',
                    type=str,
                    default='',
                    help='Full path of source file to transfer. Default ""')

args = parser.parse_args()
return args

def send_rpc(channel, metadata, args):
    stub = file_pb2_grpc.FileStub(channel)
    print("Executing GNOI::File::Put")

    # Prepare hash generator
    if args.hash_method == "sha256":
        gen_hash = hashlib.sha256()
        hm = 1

```

```

elif args.hash_method == "sha512":
    gen_hash = hashlib.sha512()
    hm = 2
elif args.hash_method == "md5":
    gen_hash = hashlib.md5()
    hm = 3
else:
    print("Unsupported hash method:", args.hash_method)
    sys.exit(1)

# Put File
req = file_pb2.PutRequest()
req.open.remote_file = args.dest_file
req.open.permissions = args.dest_file_mode
it = []
it.append(req)

# Read source file and add to request
with open(args.source_file, "rb") as file:
    # Read data in 64 KB chunks and calculate checksum and data messages
    for data in iter(partial(file.read, MAX_BYTES), b''):
        req = file_pb2.PutRequest()
        req.contents = data
        it.append(req)
        gen_hash.update(data)

# Checksum message
req = file_pb2.PutRequest()
req.hash.hash = gen_hash.hexdigest().encode()
req.hash.method = hm
it.append(req)

# Send PutRequest
try:
    print("Sending file.")
    response = stub.Put(iter(it), metadata=metadata, timeout=120)
except Exception as e:
    logging.error("Error uploading source file %s to %s. Error: %s",
                  args.source_file, args.dest_file, e)
    print(e)
else:
    print("File transfer complete: ", args.dest_file)
    logging.info("Uploaded file: %s", args.dest_file)

```

```

def main():
    parser = argparse.ArgumentParser(fromfile_prefix_chars='@')
    args = get_args(parser)

    grpc_server_password = getpass("gRPC server password for executing RPCs: ")
    metadata = [('username', args.user_id),
                ('password', grpc_server_password)]

    try:
        # Establish grpc channel to network device
        channel = grpc_authenticate_channel_mutual(
            args.server, args.port, args.root_ca_cert, args.client_key, args.client_cert)
        send_rpc(channel, metadata, args)
    except Exception as e:
        logging.error('Received error: %s', e)
        print(e)

if __name__ == '__main__':
    logging.basicConfig(filename='gnoi-testing.log',
                        format='%(asctime)s %(levelname)-8s %(message)s',
                        level=logging.INFO,
                        datefmt='%Y-%m-%d %H:%M:%S')

    main()

```

args_file_put.txt

```

--server=10.53.52.169
--port=50051
--root_ca_cert=/etc/pki/certs/serverRootCA.crt
--client_key=/home/lab/certs/client.key
--client_cert=/home/lab/certs/client.crt
--user_id=gnoi-user
--source_file=scripts/op/ospf-summary.slax
--dest_file=/var/db/scripts/op/ospf-summary.slax
--dest_file_mode=644
--hash_method=sha256

```

Execute the Application

When the client executes the application, it transfers the specified file from the local device to the target device and sets the file permissions according to the `dest_file_mode` value.

```
lab@gnoi-client:~/src/gnoi/proto$ python3 gnoi_file_put.py @args_file_put.txt
gRPC server password for executing RPCs:
Creating channel
Executing GNOI::File::Put
Sending file.
File transfer complete: /var/db/scripts/op/ospf-summary.slax
```

gNOI Operating System (OS) Service

SUMMARY

Use the gNOI operating system (OS) service to upgrade the software on the target network device.

IN THIS SECTION

- [Supported RPCs | 98](#)
- [Network Device Configuration | 98](#)
- [Example: Install and Activate | 99](#)

You can use the gNOI OS service to upgrade software on the target device. The proto definition file is located at <https://github.com/openconfig/gnoi/blob/master/os/os.proto>.

Software installation has three main steps, which correspond to the OS service RPCs.

- Install
- Activate
- Verify

The `Install()` RPC transfers the specified image to the `/var/tmp/` directory on the target device. The destination filename is the value defined in the `TransferRequest` message's `version` field. The `Activate()` RPC installs the image and reboots the device to activate the newly installed image. The `Verify()` RPC validates the OS version on the device.

NOTE: You can also use the gNOI system service `SetPackage()` RPC to install software on a device. For more information, see ["gNOI System Service" on page 105](#).

Supported RPCs

Table 15: Supported os.proto RPCs

RPC	Description	Introduced in Release
Activate()	<p>Set the OS version that is used at the next reboot.</p> <p>Activate() reboots the target if the <code>no_reboot</code> flag is omitted or set to <code>False</code>. If the reboot fails to boot the requested OS version, the target rolls back to the previous OS package.</p> <p>NOTE: Junos devices do not support the <code>standby_supervisor</code> field in the <code>ActivateRequest</code> message.</p>	Junos OS Evolved 22.2R1
Install()	<p>Transfer a software image onto the target.</p> <p>NOTE: Junos devices do not support the <code>standby_supervisor</code> field in the <code>TransferRequest</code> message.</p>	Junos OS Evolved 22.2R1
Verify()	<p>Check the running OS version. This RPC may be called multiple times while the target boots until successful.</p> <p>NOTE: Junos devices do not support <code>verify_standby</code> for <code>VerifyResponse</code>.</p>	Junos OS Evolved 22.2R1

Network Device Configuration

Before you begin:

- Configure gRPC services on the network device as described in ["Configure gRPC Services" on page 8](#).

- Configure the network management system to support gNOI operations as described in ["Configure gNOI Services" on page 24](#).

No additional configuration is required to use the OS service RPCs.

Example: Install and Activate

In this example, the client executes the `gnoi_os_install_activate.py` Python application, which performs the following operations:

- Copies the software package from the local network management system to the network device.
- Installs the package on the network device.
- Reboots the network device, thus activating the new software image.

The application calls the `Install()` RPC with the `InstallRequest()` message to transfer the file. The application tracks the progress of the file transfer by emitting progress messages at each 10 percent transfer completion interval. If the file transfer is successful, the application then calls the `Activate()` RPC to install the image and reboot the target.

The application imports the `grpc_channel` module to establish the channel. The `grpc_channel` module is described in ["Configure gNOI Services" on page 24](#). The application's arguments are stored in the `args_os_install_activate.txt` file. The application and argument files are as follows:

`gnoi_os_install_activate.py`

```
"""gRPC gNOI OS Install, Activate utility."""

from __future__ import print_function
import argparse
import logging
import os
from functools import partial
from getpass import getpass

import os_pb2
import os_pb2_grpc
from grpc_channel import grpc_authenticate_channel_mutual

MAX_BYTES = 65536
```

```

def get_args(parser):
    parser.add_argument('--server',
                        dest='server',
                        type=str,
                        default='localhost',
                        help='Server IP or name. Default is localhost')

    parser.add_argument('--port',
                        dest='port',
                        nargs='?',
                        type=int,
                        default=50051,
                        help='The server port. Default is 50051')

    parser.add_argument('--client_key',
                        dest='client_key',
                        type=str,
                        default='',
                        help='Full path of the client private key. Default ""')

    parser.add_argument('--client_cert',
                        dest='client_cert',
                        type=str,
                        default='',
                        help='Full path of the client certificate. Default ""')

    parser.add_argument('--root_ca_cert',
                        dest='root_ca_cert',
                        required=True,
                        type=str,
                        help='Full path of the Root CA certificate.')

    parser.add_argument('--user_id',
                        dest='user_id',
                        required=True,
                        type=str,
                        help='User ID for RPC call credentials.')

    parser.add_argument('--no_reboot',
                        dest='no_reboot',
                        type=int,
                        default=0,

```

```

        help='Reboot immediately or not. Default 0 (Reboot immediately)')

    parser.add_argument('--source_package',
                        dest='source_package',
                        type=str,
                        default='',
                        help='Full path of the source file. Default ""')

    parser.add_argument('--timeout',
                        dest='timeout',
                        type=int,
                        default=600,
                        help='Timeout in seconds. Default 600')

    parser.add_argument('--version',
                        dest='version',
                        type=str,
                        default='',
                        help='OS version to activate. Default is ""')

    args = parser.parse_args()
    return args

def send_rpc(channel, metadata, args):
    print("Executing GNOI::OS::Install")
    stub = os_pb2_grpc.OSStub(channel)
    it = []

    # Create file transfer request
    req = os_pb2.InstallRequest()
    req.transfer_request.version = args.version
    it.append(req)

    # Read source package and add to request
    source_package_bytes = os.path.getsize(args.source_package)
    with open(args.source_package, "rb") as file:
        # Read data in 64 KB chunks and calculate checksum and data messages
        for data in iter(partial(file.read, MAX_BYTES), b''):
            req = os_pb2.InstallRequest()
            req.transfer_content = data
            it.append(req)

```

```

req = os_pb2.InstallRequest()
req.transfer_end.SetInParent()
it.append(req)

next_pct = 0
transfer_percent = 0
validated = False
activated = False

try:
    responses = stub.Install(
        iter(it), metadata=metadata, timeout=args.timeout)
    print("OS Install start\n")

    for response in responses:
        rsp_type = response.WhichOneof('response')
        if rsp_type == 'install_error':
            print("%s: %s -- %s\n" % (rsp_type,
                response.install_error.type, response.install_error.detail))
            raise Exception("Install Error")
        elif rsp_type == 'transfer_ready':
            print("%s: %s\n" % (rsp_type, response.transfer_ready))
        elif rsp_type == 'transfer_progress':
            transfer_percent = int(float(
                response.transfer_progress.bytes_received) / float(source_package_bytes) *
100)

            if 0 == (transfer_percent % 10) and transfer_percent != next_pct:
                next_pct = transfer_percent
                print("Transfer percent complete: %s%%" % transfer_percent)
                logging.info('Transferring file %s%%', transfer_percent)
            elif rsp_type == 'validated':
                print("%s: %s -- %s\n" % (rsp_type,
                    response.validated.version, response.validated.description))
                logging.info('Validated: %s', response.validated.version)
                validated = True

    if transfer_percent > 0 and validated:
        print("Executing GNOI::OS::Activate")
        req = os_pb2.ActivateRequest()
        req.version = '/var/tmp/' + args.version
        req.no_reboot = args.no_reboot
        activate_response = stub.Activate(
            req, metadata=metadata, timeout=args.timeout)

```

```

        rsp_type = activate_response.WhichOneof('response')
        if rsp_type == 'activate_ok':
            activated = True

    except Exception as e:
        logging.error('Error installing package: %s', e)
        print(e)
    else:
        if activated:
            logging.info('Installation complete: %s', args.version)
            print('Installation complete for %s' % args.version)

def main():
    parser = argparse.ArgumentParser(fromfile_prefix_chars='@')
    args = get_args(parser)

    grpc_server_password = getpass("gRPC server password for executing RPCs: ")
    metadata = [('username', args.user_id),
                ('password', grpc_server_password)]

    try:
        # Establish grpc channel to network device
        channel = grpc_authenticate_channel_mutual(
            args.server, args.port, args.root_ca_cert, args.client_key, args.client_cert)
        response = send_rpc(channel, metadata, args)
    except Exception as e:
        logging.error('Received error: %s', e)
        print(e)

if __name__ == '__main__':
    logging.basicConfig(filename='gnoi-install.log',
                        format='%(asctime)s %(levelname)-8s %(message)s',
                        level=logging.INFO,
                        datefmt='%Y-%m-%d %H:%M:%S')

    main()

```

args_os_install_activate.txt

```
--root_ca_cert=/etc/pki/certs/serverRootCA.crt
--client_key=/home/lab/certs/client.key
--client_cert=/home/lab/certs/client.crt
--server=10.53.52.169
--port=50051
--user_id=gnoi-user
--source_package=/home/lab/images/junos-evo-install-ptx-x86-64-22.3R1.9-EVO.iso
--timeout=1800
--version=junos-evo-install-ptx-x86-64-22.3R1.9-EVO
```

Execute the Application

When the client executes the application, the application copies the package from the local device to the `/var/tmp` directory on the network device, installs the package, and then reboots the device to complete the installation.

```
lab@gnoi-client:~/src/gnoi/proto$ python3 gnoi_os_install_activate.py
@args_os_install_activate.txt
gRPC server password for executing RPCs:
Creating channel
Executing GNOI::OS::Install
OS Install response

transfer_ready:

Transfer percent complete: 10%
Transfer percent complete: 20%
Transfer percent complete: 30%
Transfer percent complete: 40%
Transfer percent complete: 50%
Transfer percent complete: 60%
Transfer percent complete: 70%
Transfer percent complete: 80%
Transfer percent complete: 90%
Transfer percent complete: 100%
validated: junos-evo-install-ptx-x86-64-22.3R1.9-EVO -- Use Activate to Validate and Integrity
Check the installed Image
```

```
Executing GNOI::OS::Activate  
Installation complete for junos-evo-install-ptx-x86-64-22.3R1.9-EVO
```

gNOI System Service

SUMMARY

Use the gNOI System service to perform system operations on the target network device, including rebooting the device, upgrading software, and troubleshooting the network.

IN THIS SECTION

- Overview | 105
- Network Device Configuration | 106
- Ping and Traceroute | 106
- Reboot a Device | 114
- Upgrade Software | 120
- Routing Engine Switchover | 125

Overview

The gNOI System service provides RPCs to perform a number of different system operations on a network device, including the following operations:

- Reboot a device
- Execute ping and traceroute commands to troubleshoot the network
- Upgrade software
- Perform a Routing Engine switchover

The proto definition file is located at <https://github.com/openconfig/gnoi/blob/master/system/system.proto>.

NOTE: The `gnoi-system` process restarts in the event of a system failure. To restart it manually, use the `restart gnoi-system` command.

Network Device Configuration

Before you begin:

- Configure gRPC services on the network device as described in ["Configure gRPC Services" on page 8](#).
- Configure the network management system to support gNOI operations as described in ["Configure gNOI Services" on page 24](#).

No additional configuration is required to use the `System` service RPCs.

Ping and Traceroute

IN THIS SECTION

- [Example: Ping | 107](#)

You can execute ping and traceroute commands on the network device to troubleshoot issues on your network.

Table 16: Supported `system.proto` RPCs for Troubleshooting the Network

RPC	Description	Introduced in Release
Ping()	<p>Ping a device. The <code>Ping()</code> RPC supports IPv4 and IPv6 pings. This RPC streams back the results of the ping after the ping is complete.</p> <p>Default number of packets: 5</p>	Junos OS Evolved 22.2R1

Table 16: Supported system.proto RPCs for Troubleshooting the Network *(Continued)*

RPC	Description	Introduced in Release
Traceroute()	Execute the traceroute command on the target device and stream back the results. Default hop count: 30	Junos OS Evolved 22.2R1

Example: Ping

In this example, the client executes the `gnoi_ping_request.py` Python application. The application sends the `Ping()` RPC to the network device, which then pings the specified device on the network.

The `gnoi_ping_request.py` application imports the `grpc_channel` module to establish the channel. The `grpc_channel` module is described in ["Configure gNOI Services" on page 24](#). The application's arguments are stored in the `gnoi_ping_request_args.txt` file. The application and argument files are presented here.

`gnoi_ping_request.py`

```

"""gRPC gNOI ping request utility."""

from __future__ import print_function
import argparse
import logging
from getpass import getpass

import system_pb2
import system_pb2_grpc

from grpc_channel import grpc_authenticate_channel_mutual

def get_args(parser):
    parser.add_argument('--server',
                        dest='server',
                        type=str,
                        default='localhost',
                        help='Server IP or name. Default is localhost')

    parser.add_argument('--port',

```

```

        dest='port',
        nargs='?',
        type=int,
        default=50051,
        help='The server port. Default is 50051')

parser.add_argument('--client_key',
                    dest='client_key',
                    type=str,
                    default='',
                    help='Full path of the client private key. Default ""')

parser.add_argument('--client_cert',
                    dest='client_cert',
                    type=str,
                    default='',
                    help='Full path of the client certificate. Default ""')

parser.add_argument('--root_ca_cert',
                    dest='root_ca_cert',
                    required=True,
                    type=str,
                    help='Full path of the Root CA certificate.')

parser.add_argument('--user_id',
                    dest='user_id',
                    required=True,
                    type=str,
                    help='User ID for RPC call credentials.')

# Ping request arguments
parser.add_argument('--destination',
                    dest='destination',
                    type=str,
                    default=None,
                    help='Destination IP. Default None')

parser.add_argument('--source',
                    dest='source',
                    type=str,
                    default=None,
                    help='Source IP. Default None')

```

```

parser.add_argument('--count',
                    dest='count',
                    type=int,
                    default=None,
                    help='Count of packets. Default None')

parser.add_argument('--interval',
                    dest='interval',
                    type=int,
                    default=None,
                    help='Interval of packets in nanoseconds. Default None')

parser.add_argument('--wait',
                    dest='wait',
                    type=int,
                    default=None,
                    help='Wait of packets in nanoseconds. Default None')

parser.add_argument('--size',
                    dest='size',
                    type=int,
                    default=None,
                    help='Size of packets. Default None')

parser.add_argument('--dnfragment',
                    dest='dnfragment',
                    type=int,
                    default=0,
                    help='Do not fragment. Default 0 (False)')

parser.add_argument('--dnresolve',
                    dest='dnresolve',
                    type=int,
                    default=0,
                    help='Do not resolve. Default 0 (False)')

parser.add_argument('--l3protocol',
                    dest='l3protocol',
                    type=int,
                    default=None,
                    help='L3 protocol (1=ipv4,2=ipv6). Default None')

parser.add_argument('--timeout',

```

```

        dest='timeout',
        type=int,
        default=30,
        help='Timeout for ping. Default: 30 seconds')

args = parser.parse_args()
return args

def send_rpc(channel, metadata, args):
    stub = system_pb2_grpc.SystemStub(channel)
    print("Executing GNOI::System::Ping Request RPC")
    req = system_pb2.PingRequest()

    if args.count != None:
        req.count = args.count
    if args.source != None:
        req.source = args.source
    if args.destination != None:
        req.destination = args.destination
    if args.interval != None:
        req.interval = args.interval
    if args.wait != None:
        req.wait = args.wait
    if args.size != None:
        req.size = args.size
    if args.dnfragment != 0:
        req.do_not_fragment = args.dnfragment
    if args.dnresolve != 0:
        req.do_not_resolve = args.dnresolve
    if args.l3protocol != None:
        req.l3protocol = args.l3protocol

    try:
        print("Ping Request Response starts\n")
        count = 1
        for ping in stub.Ping(request=req, metadata=metadata, timeout=args.timeout):
            print("Response Source%s: %s " % (count, ping.source))
            print("Time%s: %s" % (count, ping.time))
            print("Sent%s: %s" % (count, ping.sent))
            print("Receive%s: %s" % (count, ping.received))
            print("Mintime%s: %s" % (count, ping.min_time))
            print("Avgtime%s: %s" % (count, ping.avg_time))

```

```

        print("Stddev%s: %s" % (count, ping.std_dev))
        print("Bytes%s: %s" % (count, ping.bytes))
        print("Sequence%s: %s" % (count, ping.sequence))
        print("TTL%s: %s" % (count, ping.ttl))
        count += 1

    print("Ping Request Response ends\n")
except Exception as e:
    logging.error('Error: %s', e)
    print(e)

def main():
    parser = argparse.ArgumentParser(fromfile_prefix_chars='@')
    args = get_args(parser)

    grpc_server_password = getpass("gRPC server password for executing RPCs: ")
    metadata = [('username', args.user_id),
                ('password', grpc_server_password)]

    try:
        # Establish grpc channel to network device
        channel = grpc_authenticate_channel_mutual(
            args.server, args.port, args.root_ca_cert, args.client_key, args.client_cert)
        response = send_rpc(channel, metadata, args)
    except Exception as e:
        logging.error('Received error: %s', e)
        print(e)

if __name__ == '__main__':
    logging.basicConfig(filename='gnoi-testing.log',
                        format='%(asctime)s %(levelname)-8s %(message)s',
                        level=logging.INFO,
                        datefmt='%Y-%m-%d %H:%M:%S')

    main()

```

gnoi_ping_request_args.txt

```

--server=10.0.2.1
--port=50051
--root_ca_cert=/etc/pki/certs/serverRootCA.crt

```

```
--client_key=/home/lab/certs/client.key
--client_cert=/home/lab/certs/client.crt
--user_id=gnoi-user
--destination=10.0.3.1
--source=10.0.2.1
--count=5
```

Execute the Application

On the client, execute the application, which prompts for the server's password for the RPC call credentials. The `PingResponse` indicates that the device sent five pings. The final response includes the summary statistics for the ping request, which shows that the device sent five pings and received five responses.

```
lab@gnoi-client:~/src/gnoi/proto$ python3 gnoi_ping_request.py @gnoi_ping_request_args.txt
gRPC server password for executing RPCs:
Creating channel
Executing GNOI::System::Ping Request RPC
Ping Request Response starts

Response Source1: 10.0.3.1
Time1: 741000
Sent1: 0
Receive1: 0
Mintime1: 0
Avgtime1: 0
Stddev1: 0
Bytes1: 64
Sequence1: 1
TTL1: 59
Response Source2: 10.0.3.1
Time2: 734000
Sent2: 0
Receive2: 0
Mintime2: 0
Avgtime2: 0
Stddev2: 0
Bytes2: 64
Sequence2: 2
TTL2: 59
Response Source3: 10.0.3.1
```

```
Time3: 704000
Sent3: 0
Receive3: 0
Mintime3: 0
Avgtime3: 0
Stddev3: 0
Bytes3: 64
Sequence3: 3
TTL3: 59
Response Source4: 10.0.3.1
Time4: 767000
Sent4: 0
Receive4: 0
Mintime4: 0
Avgtime4: 0
Stddev4: 0
Bytes4: 64
Sequence4: 4
TTL4: 59
Response Source5: 10.0.3.1
Time5: 800000
Sent5: 0
Receive5: 0
Mintime5: 0
Avgtime5: 0
Stddev5: 0
Bytes5: 64
Sequence5: 5
TTL5: 59
Response Source6: 10.0.3.1
Time6: 4111000000
Sent6: 5
Receive6: 5
Mintime6: 704000
Avgtime6: 749000
Stddev6: 32000
Bytes6: 0
Sequence6: 0
TTL6: 0
Ping Request Response ends
```

Reboot a Device

IN THIS SECTION

- [Example: Reboot | 115](#)

Use the `System` service RPCs to remotely reboot a device, check the status of the reboot, and cancel the reboot if needed. You can execute these RPCs on the device or on specific subcomponents. Junos devices support the following reboot methods:

- COLD (1): Available for all types of reboots.
- POWERDOWN (2): Use for FPC reboots.
- HALT (3): Use for active Control Processor reboots.
- POWERUP (7): Use for FPC reboots.

Table 17: Supported `system.proto` RPCs for Rebooting

RPC	Description	Introduced in Release
Reboot()	<p>Reboot the target. You can only execute one reboot request on a target at a time.</p> <p>You can optionally configure a delay to reboot in the future, reboot subcomponents individually, and add a message when the reboot initiates. The delay is configured in nanoseconds.</p> <p>Junos devices support the following reboot methods:</p> <ul style="list-style-type: none">• COLD (1)• POWERDOWN (2)• HALT (3)• POWERUP (7)	Junos OS Evolved 22.2R1
RebootStatus()	Return the status of the reboot.	Junos OS Evolved 22.2R1

Table 17: Supported system.proto RPCs for Rebooting *(Continued)*

RPC	Description	Introduced in Release
CancelReboot()	Cancel a pending reboot request.	Junos OS Evolved 22.2R1

Example: Reboot

In this example, the client executes the `gnoi_reboot_request.py` Python application. The application sends the reboot request and then checks the status of the reboot.

The application lets the user set the reboot delay in seconds. Since the `RebootRequest()` interprets the delay in nanoseconds, the application converts the user input into nanoseconds for the request. In this example, the client specifies a 60-second delay for the reboot operation.

The `gnoi_reboot_request.py` application imports the `grpc_channel` module to establish the channel. The `grpc_channel` module is described in ["Configure gNOI Services" on page 24](#). The application's arguments are stored in the `reboot_status_request_args.txt` file. The application and argument files are presented here.

`gnoi_reboot_status_request.py`

```
"""gRPC gNOI reboot request and reboot status utility."""

from __future__ import print_function
import argparse
import logging
from getpass import getpass

import types_pb2
import system_pb2
import system_pb2_grpc

from grpc_channel import grpc_authenticate_channel_mutual

def get_args(parser):
    parser.add_argument('--server',
                        dest='server',
                        type=str,
```

```

        default='localhost',
        help='Server IP or name. Default is localhost')

parser.add_argument('--port',
                    dest='port',
                    nargs='?',
                    type=int,
                    default=50051,
                    help='The server port. Default is 50051')

parser.add_argument('--client_key',
                    dest='client_key',
                    type=str,
                    default='',
                    help='Full path of the client private key. Default ""')

parser.add_argument('--client_cert',
                    dest='client_cert',
                    type=str,
                    default='',
                    help='Full path of the client certificate. Default ""')

parser.add_argument('--root_ca_cert',
                    dest='root_ca_cert',
                    required=True,
                    type=str,
                    help='Full path of the Root CA certificate.')

parser.add_argument('--user_id',
                    dest='user_id',
                    required=True,
                    type=str,
                    help='User ID for RPC call credentials.')

# Arguments for RebootRequest
parser.add_argument('--method',
                    dest='method',
                    type=int,
                    default=1,
                    help='Reboot method. Valid value: 0 (UNKNOWN), 1 (COLD), 2 (POWERDOWN),
3 (HALT), 6 (reserved), 7 (POWERUP). Default 1')

parser.add_argument('--delay',

```

```

        dest='delay',
        type=int,
        default=None,
        help='Delay in seconds before rebooting. Default 0')

parser.add_argument('--message',
                    dest='message',
                    type=str,
                    default=None,
                    help='Message for rebooting.')

parser.add_argument('--force',
                    dest='force',
                    type=int,
                    default=None,
                    help='Force reboot. Valid value 0|1. Default 0')

parser.add_argument('--subcomponents',
                    dest='subcomponents',
                    type=str,
                    default='',
                    help='Subcomponents to reboot. Valid value re0,re1,fpc0,fpc8,etc.
Default ""')

args = parser.parse_args()
return args

def send_rpc(channel, metadata, args):
    # RebootRequest
    stub = system_pb2_grpc.SystemStub(channel)
    print("Executing GNOI::System::Reboot RPC")
    req = system_pb2.RebootRequest()

    # Add RebootRequest arguments
    req.method = args.method
    if args.delay != None:
        # gNOI delay is in nanoseconds. Convert from seconds to nanoseconds.
        req.delay = args.delay*(10**9)
    if args.message != None:
        req.message = args.message
    if args.force != None:
        req.force = args.force

```

```

for subcomponent in args.subcomponents.split(","):
    if subcomponent == "":
        continue
    elem_key = {}
    elem_key["%s" % subcomponent] = subcomponent
    path_elem = [types_pb2.PathElem(
        name="%s" % subcomponent, key=elem_key)]
    path = types_pb2.Path(origin="origin", elem=path_elem)
    req.subcomponents.extend([path])

# RebootStatus
print("Executing GNOI::System::Reboot Status RPC")
req_status = system_pb2.RebootStatusRequest()

try:
    reboot_response = stub.Reboot(
        request=req, metadata=metadata, timeout=60)
    status_response = stub.RebootStatus(
        request=req_status, metadata=metadata, timeout=60)
    print("Reboot status response received. %s" % status_response)
except Exception as e:
    logging.error('Error: %s', e)
    print(e)
else:
    logging.info('Received reboot status: %s', status_response)

def main():
    parser = argparse.ArgumentParser(fromfile_prefix_chars='@')
    args = get_args(parser)

    grpc_server_password = getpass("gRPC server password for executing RPCs: ")
    metadata = [('username', args.user_id),
                ('password', grpc_server_password)]

    try:
        # Establish grpc channel to network device
        channel = grpc_authenticate_channel_mutual(
            args.server, args.port, args.root_ca_cert, args.client_key, args.client_cert)
        send_rpc(channel, metadata, args)
    except Exception as e:
        print(e)
        logging.error('Received error: %s', e)

```

```

if __name__ == '__main__':
    logging.basicConfig(filename='gnoi-testing.log',
                        format='%(asctime)s %(levelname)-8s %(message)s',
                        level=logging.INFO,
                        datefmt='%Y-%m-%d %H:%M:%S')

    main()

```

reboot_status_request_args.txt

```

--server=10.0.2.1
--port=50051
--root_ca_cert=/etc/pki/certs/serverRootCA.crt
--client_key=/home/lab/certs/client.key
--client_cert=/home/lab/certs/client.crt
--user_id=gnoi-user
--message="Testing gNOI reboot"
--delay=60

```

Execute the Application

When the client executes the application, the application prompts for the server's password for the RPC call credentials. The application then reboots the server after a 60 second delay and returns the applicable reboot status messages. The message set under reason also appears on the server immediately before the server reboots. In this example, any user logged into the server sees "Testing gNOI reboot" immediately before it reboots.

```

lab@gnoi-client:~/src/gnoi/proto$ python3 gnoi_reboot_status_request.py
@reboot_status_request_args.txt
gRPC server password for executing RPCs:
Creating channel
Executing GNOI::System::Reboot RPC
Executing GNOI::System::Reboot Status RPC
Reboot status response received! active: true
wait: 59266969677
when: 1651788480000000000
reason: "Testing gNOI reboot"
count: 5

```

Upgrade Software

IN THIS SECTION

- [Example: Install a Software Package | 120](#)

Table 18 on page 120 lists the **System** service RPCs that support software upgrades.

Table 18: Supported system.proto RPCs for Software Upgrades

RPC	Description	Introduced in Release
SetPackage()	Install a software image on the target device.	Junos OS Evolved 22.2R1

You can use the SetPackage() RPC to copy a software image to the target device and install it. The source software image must reside on the local network management system. If the file copy operation is successful, and a file of the same name already exists at the target location, the file is overwritten. The RPC returns an error if the target location does not exist or if there is an error writing the data.

By default, SetPackage() does not reboot the device and activate the software. You must explicitly set the activate option to 1 in the SetPackageRequest message to activate the new software. If you activate the software, the device reboots and uses the new software image. If you do not activate the software, you must reboot the relevant nodes to complete the installation and activate the new software image.

Example: Install a Software Package

In this example, the client executes the gnoi_system_set_package.py Python application, which performs the following operations:

- Copies the software package from the local network management system to the network device.
- Installs the package on the network device.
- Reboots the network device, thus activating the new software image.

The application constructs the SetPackageRequest message with the appropriate parameters to define the request for the copy and install operations. The application then calls the SetPackage() RPC to send the request to the network device. The SetPackageRequest message contains the following components:

- An initial Package message containing the path and file information for the software image. The activate argument is set to 1 (True) to reboot the device and activate the software.
- A stream of the software image file contents in sequential messages that do not exceed 64KB.
- A final message with the file checksum to verify the integrity of the file contents.

The `gnoi_system_set_package.py` application imports the `grpc_channel` module to establish the channel. The `grpc_channel` module is described in ["Configure gNOI Services" on page 24](#). The application's arguments are stored in the `args_system_set_package.txt` file. The application and argument files are as follows:

`gnoi_system_set_package.py`

```
"""gRPC gNOI OS Upgrade Utility."""

from __future__ import print_function
import argparse
import hashlib
import logging
from functools import partial
from getpass import getpass

import system_pb2
import system_pb2_grpc
from grpc_channel import grpc_authenticate_channel_mutual

MAX_BYTES = 65536

def get_args(parser):
    parser.add_argument('--server',
                        dest='server',
                        type=str,
                        default='localhost',
                        help='Server IP or name. Default is localhost')

    parser.add_argument('--port',
                        dest='port',
                        nargs='?',
                        type=int,
                        default=50051,
                        help='The server port. Default is 50051')
```

```

parser.add_argument('--client_key',
                    dest='client_key',
                    type=str,
                    default='',
                    help='Full path of the client private key. Default ""')

parser.add_argument('--client_cert',
                    dest='client_cert',
                    type=str,
                    default='',
                    help='Full path of the client certificate. Default ""')

parser.add_argument('--root_ca_cert',
                    dest='root_ca_cert',
                    required=True,
                    type=str,
                    help='Full path of the Root CA certificate.')

parser.add_argument('--user_id',
                    dest='user_id',
                    required=True,
                    type=str,
                    help='User ID for RPC call credentials.')

parser.add_argument('--activate',
                    dest='activate',
                    type=int,
                    default=0,
                    help='Reboot and activate the package. Default: 0 (Do not reboot/
activate). Valid value: 1 (Reboot/activate).')

parser.add_argument('--filename',
                    dest='filename',
                    type=str,
                    default='',
                    help='Destination path and filename of the package. Default ""')

parser.add_argument('--source_package',
                    dest='source_package',
                    type=str,
                    default='',
                    help='Full path of the source file to send. Default ""')

```



```

parser.add_argument('--timeout',
                    dest='timeout',
                    type=int,
                    default=None,
                    help='Timeout in seconds.')

parser.add_argument('--version',
                    dest='version',
                    type=str,
                    default='',
                    help='Version of the package. Default ""')

args = parser.parse_args()
return args

def send_rpc(channel, metadata, args):
    stub = system_pb2_grpc.SystemStub(channel)

    print("Executing GNOI::System::SetPackage")

    # Create request
    # Add file information to request
    req = system_pb2.SetPackageRequest()
    req.package.activate = args.activate
    req.package.filename = args.filename
    it = []
    it.append(req)

    # Prepare hash generator
    gen_hash = hashlib.sha256()

    # Read source package and add to request
    with open(args.source_package, "rb") as fd:
        # Read data in 64 KB chunks and calculate checksum and data messages
        for data in iter(partial(fd.read, MAX_BYTES), b''):
            req = system_pb2.SetPackageRequest()
            req.contents = data
            it.append(req)
            gen_hash.update(data)

    # Add checksum to request
    req = system_pb2.SetPackageRequest()

```

```

req.hash.hash = gen_hash.hexdigest().encode()
req.hash.method = 1
it.append(req)

# Install the package
try:
    logging.info('Installing package %s', args.source_package)
    print('SetPackage start.')
    response = stub.SetPackage(
        iter(it), metadata=metadata, timeout=args.timeout)
    print('SetPackage complete.')
except Exception as e:
    logging.error('Software install error: %s', e)
    print(e)
else:
    logging.info('SetPackage complete.')
    return response

def main():
    parser = argparse.ArgumentParser(fromfile_prefix_chars='@')
    args = get_args(parser)

    grpc_server_password = getpass("gRPC server password for executing RPCs: ")
    metadata = [('username', args.user_id),
                ('password', grpc_server_password)]

    try:
        # Establish grpc channel to network device
        channel = grpc_authenticate_channel_mutual(
            args.server, args.port, args.root_ca_cert, args.client_key, args.client_cert)
        response = send_rpc(channel, metadata, args)
    except Exception as e:
        logging.error('Error: %s', e)
        print(e)

if __name__ == '__main__':
    logging.basicConfig(filename='gnoi-install.log',
                        format='%(asctime)s %(levelname)-8s %(message)s',
                        level=logging.INFO,

```

```

                                datefmt='%Y-%m-%d %H:%M:%S')
main()

```

args_system_set_package.txt

```

--server=10.0.2.1
--port=50051
--root_ca_cert=/etc/pki/certs/serverRootCA.crt
--client_key=/home/lab/certs/client.key
--client_cert=/home/lab/certs/client.crt
--user_id=gnoi-user
--activate=1
--filename=/var/tmp/junos-evo-install-ptx-x86-64-22.2R1.13-EV0.iso
--source_package=/home/lab/images/junos-evo-install-ptx-x86-64-22.2R1.13-EV0.iso
--timeout=1800

```

Execute the Application

When the client executes the application, the application copies the package from the local device to the network device, installs it, and then reboots the device to complete the installation.

```

lab@gnoi-client:~/src/gnoi/proto$ python3 gnoi_system_set_package.py
@args_system_set_package.txt
gRPC server password for executing RPCs:
Creating channel
Executing GN0I::System::SetPackage
SetPackage start.
SetPackage complete.

```

Routing Engine Switchover

IN THIS SECTION

- [Example: Routing Engine Switchover | 126](#)

You can use the `SwitchControlProcessor()` RPC to perform a Routing Engine switchover.

NOTE: Successive Routing Engine switchover events must be a minimum of 400 seconds apart after both Routing Engines have come up.

Table 19: Supported system.proto RPCs for Routing Engine Switchover

RPC	Description	Introduced in Release
<code>SwitchControlProcessor()</code>	<p>Switch from the current Routing Engine to the specified Routing Engine. If the current and specified Routing Engines are the same, it is a NOOP. If the target does not exist, the RPC returns an error.</p> <p>NOTE: Junos devices do not support <code>control_processor</code> for <code>SwitchControlProcessorResponse</code>.</p>	Junos OS Evolved 22.2R1

Example: Routing Engine Switchover

In this example, the gNOI client executes the `gnoi_system_switch_control_processor.py` application to perform a Routing Engine switchover. The client specifies which switch control processor, or Routing Engine, should be the primary Routing Engine by including the `control_processor` argument. If the target Routing Engine does not exist, the RPC returns an `INVALID_ARGUMENT` error.

The application imports the `grpc_channel` module to establish the channel. The `grpc_channel` module is described in ["Configure gNOI Services" on page 24](#).

`gnoi_system_switch_control_processor.py`

```
"""gNOI Routing Engine switchover request utility."""

from __future__ import print_function
import argparse
import logging
from getpass import getpass

import system_pb2
import system_pb2_grpc
import types_pb2
```

```

from grpc_channel import grpc_authenticate_channel_mutual

def get_args(parser):
    parser.add_argument('--server',
                        dest='server',
                        type=str,
                        default='localhost',
                        help='Server IP or name. Default is localhost')

    parser.add_argument('--port',
                        dest='port',
                        nargs='?',
                        type=int,
                        default=50051,
                        help='The server port. Default is 50051')

    parser.add_argument('--client_key',
                        dest='client_key',
                        type=str,
                        default='',
                        help='Full path of the client private key. Default ""')

    parser.add_argument('--client_cert',
                        dest='client_cert',
                        type=str,
                        default='',
                        help='Full path of the client certificate. Default ""')

    parser.add_argument('--root_ca_cert',
                        dest='root_ca_cert',
                        required=True,
                        type=str,
                        help='Full path of the Root CA certificate.')

    parser.add_argument('--user_id',
                        dest='user_id',
                        required=True,
                        type=str,
                        help='User ID for RPC call credentials.')

    parser.add_argument('--control_processor',
                        dest='control_processor',

```

```

        type=str,
        default='re1',
        help='Control processor that will assume the role of primary. Default is
re1. Valid values are re0,re1.')

args = parser.parse_args()
return args

def send_rpc(channel, metadata, processor):
    stub = system_pb2_grpc.SystemStub(channel)
    print("Executing GNOI::System::SwitchControlProcessor")

    elem_key = {}
    elem_key["%s" % processor] = processor
    path_elem = [types_pb2.PathElem(name="%s" % processor, key=elem_key)]
    path = types_pb2.Path(origin="origin", elem=path_elem)
    req = system_pb2.SwitchControlProcessorRequest(control_processor=path)

    # Send the request
    try:
        response = stub.SwitchControlProcessor(
            req, metadata=metadata, timeout=60)
        print("SwitchControlProcessor response:\n", response)
    except Exception as e:
        logging.error('Switchover error: %s', e)
        print(e)
    else:
        logging.info('SwitchControlProcessor response:\n %s', response)
        return response

def main():
    parser = argparse.ArgumentParser()
    args = get_args(parser)

    grpc_server_password = getpass("gRPC server password for executing RPCs: ")
    metadata = [('username', args.user_id),
                ('password', grpc_server_password)]

    try:
        # Establish grpc channel to network device
        channel = grpc_authenticate_channel_mutual(

```

```

        args.server, args.port, args.root_ca_cert, args.client_key, args.client_cert)
    response = send_rpc(channel, metadata, args.control_processor)
except Exception as e:
    logging.error('Received error: %s', e)
    print(e)

if __name__ == '__main__':
    logging.basicConfig(filename='gnoi-testing.log',
                        format='%(asctime)s %(levelname)-8s %(message)s',
                        level=logging.INFO,
                        datefmt='%Y-%m-%d %H:%M:%S')

    main()

```

Execute the Application

The client executes the application and sets the `control_processor` argument to `re1` so that `re1` becomes the primary Routing Engine.

```

lab@gnoi-client:~/src/gnoi/proto$ python3 gnoi_system_switch_control_processor.py --server
10.0.2.1 --port 50051 --root_ca_cert /etc/pki/certs/serverRootCA.crt --client_key /home/lab/
certs/client.key --client_cert /home/lab/certs/client.crt --user_id gnoi-user --
control_processor re1
gRPC server password for executing RPCs:
Creating channel
Executing GNOI::System::SwitchControlProcessor
SwitchControlProcessor response:
  version: "22.2R1.13-EV0"
  uptime: 1652478709000000000

```

After executing the operation, `re1` is the primary Routing Engine on the target device.

```

{master}
lab@gnoi-server-re1>

```

4

CHAPTER

gRIBI

gRIBI | 131

gRIBI

SUMMARY

gRPC Routing Information Base Interface (gRIBI) is a gRPC service that enables external applications to programmatically add, modify, and remove routes on a network device.

IN THIS SECTION

- [Supported RPCs | 132](#)
- [Network Device Configuration | 132](#)
- [Modify Routes | 135](#)
- [Get Routes | 144](#)
- [Flush Routes | 145](#)

The gRIBI service is an API for adding, modifying, and removing routing entries in a device's routing information base (RIB, also known as a routing table). If the entry is eligible for forwarding, the operating system automatically adds the entry to the device's forwarding information base (FIB, also known as a forwarding table). gRIBI client applications can use any language supported by the Juniper Extension Toolkit (JET). The client application can run on an external network management system or as a local application on the network device.

The gRIBI service proto definition file is located at <https://github.com/openconfig/gribi/blob/master/v1/proto/service/gribi.proto>. The gRIBI messages that are supported by Junos devices are located in the [JET IDL package](#).

The OpenConfig Abstract Forwarding Table (AFT) model is a YANG data model that describes the forwarding entries installed on a network device. gRIBI uses a Protocol Buffer translated version of the OpenConfig AFT model to describe the RIB entries that it can modify. The protobuf representation of the OpenConfig AFT schema is in the proto definition file located at https://github.com/openconfig/gribi/blob/master/v1/proto/gribi_aft/gribi_aft.proto.

Benefits of gRIBI:

- Sends acknowledgments when you program a route.
- Supports hierarchical lookups.
- Supports arbitration when multiple clients are connected to the gRIBI session.

Use the `show route extensive` command to display route data for gRIBI including the client ID and the next hop group ID used by the route.

NOTE: We recommend using either gRIBI or the JET RIB Service API, not both simultaneously, especially for same set of routes.

Supported RPCs

Junos devices support gRIBI service RPCs to remotely retrieve, add, modify, or delete routes from a device's RIB. The RPCs function by modifying or reading the abstract forwarding tables (AFTs) on the device.

Table 20: Supported gribi.proto RPCs

RPC	Definition	Introduced in Release
Modify()	Add, modify, or delete entries from the AFT.	Junos OS Release 19.4R1 Junos OS Evolved Release 20.3R1
Get()	Retrieve the installed entries from the AFT.	Junos OS Evolved 22.2R1
Flush()	Remove all the device's AFT entries that match what is described in the FlushRequest message.	Junos OS Evolved 22.2R1

Network Device Configuration

Before you begin:

- Configure gRPC services on the network device as described in ["Configure gRPC Services" on page 8](#).

To configure the network device for gRIBI:

1. Create a routing instance with filter-based forwarding.

```
[edit]
user@host# set routing-instances routing-instance-name instance-type forwarding
```

2. Configure the routing table(s) you want to use for default, IPv4 family protocol, and IPv6 family protocol route resolution in your routing instance.

```
[edit routing-instances routing-instance-name routing-options resolution]
user@host# set rib routing-table-name-1 resolution-ribs default-routing-table-name
user@host# set rib routing-table-name-1 inet-resolution-ribs IPv4-routing-table-name
user@host# set rib routing-table-name-1 inet6-resolution-ribs IPv6-routing-table-name
```

You can specify up to two routing tables for each protocol family. The route resolution scheme only checks the second routing table if it can't find an entry for the protocol next-hop address in the first routing table.

In this example, the `teVRF.inet.0` is the default routing table. If there is no route for the next-hop address in that routing table, the route solution scheme checks the `inet.3` table.

```
[edit routing-instances teVRF routing-options resolution]
user@host# set rib teVRF.inet.0 resolution-ribs teVRF.inet.0
user@host# set rib teVRF.inet.0 resolution-ribs inet.3
user@host# set rib teVRF.inet.0 inet6-resolution-ribs inet6.3
```

3. Specify import policies for the IPv4 and IPv6 family resolution trees.

```
[edit routing-instances routing-instance-name routing-options resolution]
user@host# set rib routing-table-name-1 inet-import policy-name-IPv4
user@host# set rib routing-table-name-1 inet6-import policy-name-IPv6
```

For example:

```
[edit routing-instances teVRF routing-options resolution]
user@host# set rib teVRF.inet.0 inet6-import mp-resolve
```

4. Configure the two policies: one to handle multipath resolve and one to handle load balancing.

```
[edit policy-options]
user@host# set policy-statement policy-name term t1 then accept
user@host# set policy-statement policy-name term t1 then multipath-resolve
user@host# set policy-statement policy-name-2 then load-balance per-packet
```

In this example, a policy called `mp-resolve` handles multipath resolve. If the resolving route has multiple paths, the resolved route resolves over all the paths. The policy `pplb` tells the Packet Forwarding Engine to load balance the traffic for each packet.

```
[edit policy-options]
user@host# set policy-statement mp-resolve term t1 then accept
user@host# set policy-statement mp-resolve term t1 then multipath-resolve
user@host# set policy-statement pplb then load-balance per-packet
```

5. Configure the routing options to preserve the next hop hierarchy when installing a next hop in the forwarding plane.

```
[edit]
user@host# set routing-options resolution preserve-nexthop-hierarchy
```

6. Configure the routing table(s) you want to use for IPv4 and IPv6 family protocol route resolution and the policy for route resolution at the routing options level. Repeat this configuration for each routing table you configured at the routing instance level.

```
[edit routing-options resolution]
user@host# set rib routing-table-name inet-resolution-ribs routing-table-name-IPv4
user@host# set rib routing-table-name inet6-resolution-ribs routing-table-name-IPv6
user@host# set rib routing-table-name import policy-name-IPv4
user@host# set rib routing-table-name inet-import policy-name-IPv4
user@host# set rib routing-table-name inet6-import policy-name-IPv6
```

For example:

```
[edit routing-options resolution]
user@host# set rib inet.0 inet-resolution-ribs inet.3
user@host# set rib inet.0 inet-resolution-ribs teVRF.inet.0
user@host# set rib inet.0 import mp-resolve
```

```

user@host# set rib inet.0 inet-import mp-resolve
user@host# set rib inet.0 inet6-import mp-resolve
user@host# set rib inet6.3 inet-resolution-ribs inet.3
user@host# set rib inet6.3 import mp-resolve
user@host# set rib inet6.3 inet6-import mp-resolve
user@host# set rib inet.3 inet6-resolution-ribs inet6.3
user@host# set rib inet.3 import mp-resolve
user@host# set rib inet.3 inet6-import mp-resolve

```

7. Set the remnant holdtime to give the system enough time to update the routes. After the system restarts, the rpd process waits until the remnant holdtime expires before cleaning up the routes. The rpd process does not delete any routes that are updated before the wait time expires.

```

[edit]
user@host# set routing-options forwarding-table remnant-holdtime 20

```

Modify Routes

IN THIS SECTION

- [Route Acknowledgments | 136](#)
- [Program an IPv4 Route | 137](#)
- [Program Next Hops and Next Hop Groups | 137](#)
- [Program Next Hops with MAC Addresses | 138](#)
- [Hierarchical Lookups and IP-in-IP Tunneling | 139](#)
- [Arbitration for Multiple Clients | 140](#)
- [Program a Fallback Route in a VRF Instance | 142](#)
- [VRF Instance Selection | 143](#)
- [Policy-Based Forwarding | 144](#)

Use the `Modify()` RPC to install new routes and edit existing routes in the gRIBI server's RIB. Routes are added as static routes.

Modify() is a bidirectional streaming RPC. The client sends a Modify() RPC containing ModifyRequest messages to modify an AFT entry on the server. For each ModifyRequest, the gRIBI server responds to the client with a ModifyResponse message.

The ModifyRequest message comprises one or more AFTOperation messages. Each AFTOperation message defines a request to add, modify, or remove a single AFT entry. The gRIBI server processes the AFT operations in the order that the Modify() RPC streams them.

Junos devices support the following AFT entry types:

- IPv4Entry—Program an IPv4 route.
- NextHopEntry—Program a next hop.
- NextHopGroup—Program a next hop group.

Use the Modify() RPC to perform the following functions:

Route Acknowledgments

The server sends an acknowledgment when you successfully program a route in the Packet Forwarding Engine using the Modify() RPC. If the gRIBI API fails to program a route in the Packet Forwarding Engine within the given timeout period, the server sends an error message. You can configure the length of this timeout. The acknowledgment is only valid for the most recent route. If an older route sends an acknowledgment but the new route does not, the Packet Forwarding Engine records it as an error.

Junos devices support the following values in the entry field of the message AFTOperation:

```
AFTOperation {
  EntryAckType {
    INVALID;
    FIB_ACK;
    RIB_ACK;
  }
  ack_type;
}
```

NOTE: Junos devices do not support the MAC_ENTRY option.

Use the show route extensive command to display the acknowledgment status. The acknowledgment status is persistent across rpd process restarts.

Program an IPv4 Route

To program an IPv4 route, use the IPv4Entry AFT entry. The AFT matches the input packets based on the destination address and maps them to the corresponding next hops. Install the AFT entry on the default VRF instance as well as the traffic engineering VRF instances in your network. To install an AFT entry in a non-default instance, specify the VRF instance in the `network_instance` field of the AFTOperation message. For example:

- Traffic engineering VRF instance: `g_b4_cos1`
- Set the `network_instance` field to: `g_b4_cos1`

The gRIBI client only programs the IPv4Entry AFT entry on the server after it receives acknowledgments from the server that the server received the associated NextHopGroup and NextHop messages. If the client programs the IPv4Entry AFT entry on the server without acknowledgment of the NextHopGroup message, it adds the route to the server as a hidden route.

Program Next Hops and Next Hop Groups

Use the gRIBI `Modify()` RPC to program a next hop or a next hop group on the gRIBI server. The RPC only creates next hops and next hop groups within the default VRF instance.

When there are next hops and next hop groups in the same `ModifyRequest` message, the gRIBI client handles them according to the AFT operation. If the AFT operation adds NextHop and NextHopGroup entries, the client adds all the next hops to the server before adding the next hop groups. If the AFT operation deletes NextHop and NextHopGroup entries, the client processes them in the reverse order: it deletes all the next hop groups before deleting the next hops.

In Junos devices, the RPC instantiates next hops in the `inet6.3` table as `FC01::next_hop_id`, where the next hop ID is in hexadecimal. For example, if the next hop ID is 10, the server installs a route called `FC01::A` in the `inet6.3` table.

Next hop groups appear in the `inet6.3` table as `FC02::next_hop_id`. For example, if the next hop group ID is 100, the server installs a route called `FC02::64` in the `inet6.3` table.

For example, to program a next hop object via a directly reachable interface:

1. Assuming the address 10.0.1.2 is reachable via interface `et-0/0/7.0`, set the following fields in the Afts message, where = means set the field to that value:

```
NextHop {
  ip_address = 10.0.1.2; // Next hop IP address
  InterfaceRef {
    interface = "et-0/0/7";
```

```

        subinterface = 0;
    }
}
NextHopKey {
    index = 1;
}

```

2. Set the AFTOperation message fields as follows:

```

AFTOperation {
    Operation {
        ADD;
    }
    entry {
        next_hop; // NextHopKey object created above
    }
}

```

3. Set the ModifyRequest message to use the AFTOperation defined above.
4. Call the Modify() RPC with the above ModifyRequest message.
5. To confirm the route was programmed successfully, use the `show route programmed` command in the CLI.

Program Next Hops with MAC Addresses

Optionally, you can identify a next hop with its MAC address instead of its IP address. This feature is useful in networks where devices cannot use dynamic Address Resolution Protocol (ARP) or Neighbor Discovery Protocol (NDP) to look up the next hop's MAC address. To use the MAC address, use the `mac_address` field instead of the `ip_address` field in the AFT message.

NOTE: All traffic using this interface uses the static MAC address programmed by the gRIBI service, even traffic on routes not programmed by the gRIBI service.

After you use the gRIBI service to program a MAC address as the next hop on the interface, the device does not use dynamic ARP or NDP for any traffic using this interface. If the programmed gRIBI next hop is deleted or purged when the client disconnects, the device automatically re-enables ARP on the interface and the route continues to function using dynamic ARP.

For example, to program a next hop object with a MAC address via a directly reachable interface:

1. Make sure the interface you want to program with a next hop is a numbered interface.
2. Make sure the IPv6 family is enabled on the interface.
3. Assuming the MAC address 00:00:5E:00:53:00 is reachable via interface et-0/0/7.0, set the following fields in the Afts message, where = means set the field to that value:

```

NextHop {
  mac_address = 00:00:5E:00:53:00; // Next hop MAC address
  InterfaceRef {
    interface = "et-0/0/7";
    subinterface = 0;
  }
}
NextHopKey {
  index = 1;
}

```

4. Set the AFTOperation message fields as follows:

```

AFTOperation {
  Operation {
    ADD;
  }
  entry {
    next_hop; // NextHopKey object created above
  }
}

```

5. Set the ModifyRequest message to use the AFTOperation defined above.
6. Call the Modify() RPC with the above ModifyRequest message.
7. To confirm the route was programmed successfully, use the `show route programmed` command in the CLI.

Hierarchical Lookups and IP-in-IP Tunneling

The Junos implementation of gRIBI supports hierarchical lookups. To configure hierarchical lookups, use the IPv4 AFT to program IP-IP tunnel endpoints and site group virtual IP address routes.

To encapsulate traffic on the ingress node in an IP-in-IP tunnel, set the following fields in the `NextHop` message:

```
NextHop {
  encapsulate_header;
  IpInIp {
    dst_ip; // Destination IP address
    src_ip; // Source IP address
  }
}
```

Arbitration for Multiple Clients

The `Modify()` RPC supports arbitration when multiple clients are connected to the gRIBI server. Arbitration determines which client can perform which operations.

Use the `SessionParameters` message to set the persistence mode and the client redundancy mode for the gRIBI clients. All clients must send the same values of all the attributes of the `SessionParameters` message. `SessionParameters` should be sent only once during the lifetime of the session.

`SessionParameters` must be the first message sent after a reconnect. When a client reconnects, a new session starts. If other clients are already connected, match the `SessionParameters` message values to the values set by existing clients. If all the clients reconnect, you can set the `SessionParameters` message values to different values than the ones used in the previous session.

Junos devices support both `PRESERVE` and `DELETE` persistence modes. If the persistence mode is set to `PRESERVE`, then the server preserves the AFT entries added by the client even after the client disconnects. If the persistence mode is set to `DELETE`, then the server deletes the AFT entries when the client disconnects.

We recommend deleting all routes before changing session parameters. You might see unexpected behavior if you change the session parameters and switch the redundancy mode between `ALL_PRIMARY` and `SINGLE_PRIMARY` after adding routes in the other mode.

When there are multiple clients, you must chose between two client redundancy modes:

All Primary Mode

In `ALL_PRIMARY` redundancy mode:

- Any client can modify routes.
- Multiple clients can add the same AFT entry.

- The gRIBI API maintains a mapping of which clients have added the route.
- The first add operation adds the entry to the RIB. Subsequent add operations for the same entry from a different client adds the client to the list of clients referencing the entry.
- Delete operations remove the client from the list of clients referencing the entry. The entry is only deleted when there are no clients referencing the entry.

NOTE: When FlushRequest is processed, the entries are deleted without any reference count checks.

Use the `show route extensive` command to view the details of the route. Here is an example of what the `show route extensive` command displays in ALL_PRIMARY mode. The output has been shortened for clarity.

```
user@host> show route 10.0.1.1 extensive

b4.inet.0: 2 destinations, 2 routes (2 active, 0 holddown, 0 hidden)
10.0.1.1/32 (1 entry, 1 announced)
TSI:
[...]
Opaque data client: PRPD
Address: ABC123
Opaque-data reference count: 2
Opaque data PRPD: client_num_ids=1,5,6 nh group Id=110
```

Single Primary Mode

In SINGLE_PRIMARY redundancy mode:

- gRIBI clients can have a primary (active) or backup role.
- Only the primary client can perform AFT operations.
- The client with the highest election ID is the primary client. All other clients are backup clients.
- When a backup client becomes the primary client, the routes added by the previous primary client can be modified by the new primary client.

Set the election ID for each device to determine which client is the primary client. You can only set the election ID in SINGLE_PRIMARY redundancy mode. The election ID is preserved even if a client is in the down state. If the primary client disconnects, it remains the primary client until you set the election ID of

another device to be higher. After the election ID is set, the new primary client continues programming the gRIBI entries.

To update the election ID, send the `ModifyRequest` message with the election ID set to its new value. Each client must have a unique election ID. Do not set any other fields of the `ModifyRequest` message when you update the election ID.

The election ID is present in the following messages:

- `ModifyRequest`—Set the election ID for the client. The client with the highest election ID becomes the primary client.
- `AFTOperation`—Determines if the server should process the AFT operation.
- `ModifyResponse`—The server responds with the current highest election ID.

Use the `show programmable-rpd clients detail` command to view the group ID and whether the client has the primary or backup role.

Use the `show route extensive` command to view the details of the route. Here is an example of what the `show route extensive` command displays in `SINGLE_PRIMARY` mode. The output has been shortened for clarity.

```
user@host> show route 10.0.1.1 extensive

b4.inet.0: 2 destinations, 2 routes (2 active, 0 holddown, 0 hidden)
10.0.1.1/32 (1 entry, 1 announced)
TSI:
[...]
Opaque data client: PRPD
Address: ABC123
Opaque-data reference count: 2
Opaque data PRPD: group_num_id=1 nh group Id=110
```

Program a Fallback Route in a VRF Instance

When a next hop becomes unreachable through a static route, the network can reroute the traffic through an alternate route to avoid traffic disruption. This alternate route is called a fallback route. If the traffic was not encapsulated in a tunnel, configure the fallback static route as you usually would using the CLI. However, if the traffic was encapsulated in a tunnel, you can use gRIBI to program a fallback tunnel that includes decapsulation and encapsulation.

You can program the fallback route in the VRF so that the system decapsulates the traffic from the old tunnel and re-encapsulates it in a new tunnel before re-routing the traffic to the next hop. This feature supports IPv4 transport for dynamic IP-IP tunnels with an IPv4 or IPv6 payload.

To program a fallback IP-in-IP tunnel with decapsulation and re-encapsulation capability, set the following fields in the `NextHop` message:

```
NextHop {
  decapsulate_header;
  encapsulate_header;
  network_instance; // VRF instance
  IpInIp {
    dst_ip; // Destination IP address
    src_ip; // Source IP address
  }
}
```

You can use a default route in a traffic engineering virtual routing and forwarding (VRF) instance as the backup route. Add the default route to the VRF first so the future routes you configure in the VRF will use it as a fallback route. To use this default route, set the `decapsulate_header` field to `OPENCONFIGAFTTYPESENCAPSULATION HEADERTYPE_IPV4` and set `network_instance` to `DEFAULT`. This default route has a next hop with decapsulation and looks up routes in the default VRF.

You can also select a backup next hop group to make it easier to configure a fallback route. To do so, set the `backup_next_hop_group` field in the `NextHopGroup` message.

VRF Instance Selection

gRIBI does not support programming routes in a non-default VRF instance. To use a non-default VRF instance, first configure a firewall filter using the CLI. The firewall filter must match the DSCP and IP protocol required. Apply the filter to the interface on which the traffic is expected.

For example, if traffic is on interface `et-0/0/0`:

```
[edit]
user@host# set firewall filter b4-filter term 1 from dscp cs7
user@host# set firewall filter b4-filter term 1 then count b4-count
user@host# set firewall filter b4-filter term 1 then routing-instance b4
user@host# set firewall filter b4-filter term 2 then accept
user@host# set interfaces et-0/0/0 unit 0 family inet filter input b4-filter
```

Policy-Based Forwarding

Use the `PolicyForwardingEntry` message to program policy-based forwarding on the gRIBI server. Policy-based forwarding ensures that traffic moved to the backup tunnel remains in the tunnel regardless of what the routing table says.

To set the match conditions and program a policy for forwarding traffic:

1. Set the following fields in the `Afts` message:

```
PolicyForwardingEntry {
  ip_prefix; // To match the destination IP address
  src_ip_prefix; // To match the source IP address
  next_hop_group;
}
```

2. Set the following fields in the `AFTOperation` message:

```
AFTOperation {
  entry {
    policy_forwarding_entry; // PolicyForwardingEntryKey object created above
  }
}
```

3. Set the `ModifyRequest` message to use the `AFTOperation` defined above.
4. Call the `Modify()` RPC with the above `ModifyRequest` message.

Get Routes

When the client loses the connection to the gRIBI server, any routes that were programmed during the downtime might not be added to the server. After the connection to the server comes back up, use the `Get()` RPC to check that all the routes were added correctly to the server's routing table. The `Get()` RPC is also useful for periodically checking that the routes installed on a server are correct and reconciling any differences.

The `Get()` RPC retrieves the contents of the AFTs installed on the server. When the client sends a `Get()` RPC request, the server responds with the set of currently installed entries using the `GetResponse` stream. The server only responds with the entries that have been acknowledged. After the server sends all the entries to the client, the server closes the RPC.

If graceful Routing Engine switchover (GRES) is configured, the gRIBI server and rpd process also recover routes after the gRIBI server restarts. After the client reconnects to the server, the client automatically sends a gRIBI `Get()` RPC request to the server. If GRES is configured, the client reconciles the routes on the server. If the client sends another `Get()` RPC request, the `GetResponse` stream includes the active reconciled routes on the server. If GRES is configured and non-stop routing is not configured, the gRIBI API also recovers routes after a Routing Engine switchover.

NOTE: Only active routes are recovered when the rpd process restarts.

Flush Routes

The `Flush()` RPC removes all the server's gRIBI programmed routes that match what is described in the `FlushRequest` message. Sending a `FlushRequest` message is a quick and easy way to delete gRIBI programmed routes from the server.

When routes are present in a traffic engineering VRF instance, flush the routes from the VRF instance using the `Flush()` RPC before deleting the VRF instance.

5

CHAPTER

Configuration Statements and Operational Commands

[Junos CLI Reference Overview](#) | 147

Junos CLI Reference Overview

We've consolidated all Junos CLI commands and configuration statements in one place. Learn about the syntax and options that make up the statements and commands and understand the contexts in which you'll use these CLI elements in your network configurations and operations.

- [Junos CLI Reference](#)

Click the links to access Junos OS and Junos OS Evolved configuration statement and command summary topics.

- [Configuration Statements](#)
- [Operational Commands](#)