Configurer le service L3VPN MPLS sur un routeur PE à l'aide de REST-API (IOS-XE)

Table des matières

Introduction

Ce document démontre l'utilisation de la programmation Python pour provisionner un MPLS L3VPN sur un routeur de périphérie de fournisseur de services (PE) en utilisant l'API REST. Cet exemple utilise des routeurs Cisco CSR1000v (IOS-XE) comme routeurs PE.

Contribution : Anuradha Perera

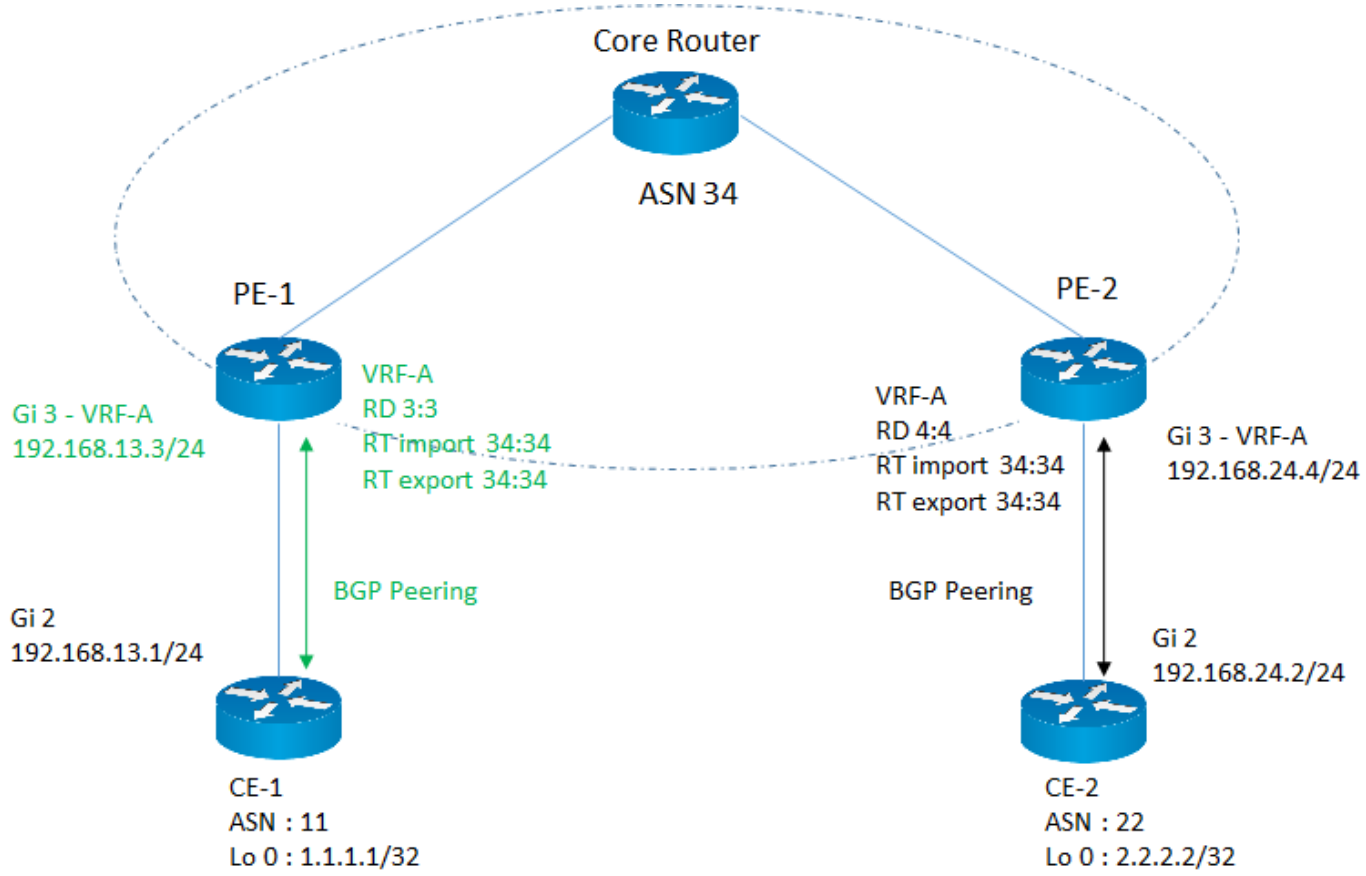
Edité par : Kumar Sridhar

Conditions préalables

- Accès de gestion de l'API REST aux routeurs CSR1000v (reportez-vous aux références à la fin de ce document).
- Python (Version 2.x ou 3.x) et "Requests" Bibliothèque Python installée sur l'ordinateur utilisé pour configurer les routeurs.
- Quelques connaissances de base en programmation Python.

Configuration

Diagramme du réseau



Dans cet exemple, l'accent est mis sur la configuration des paramètres de service MPLS L3VPN requis sur un routeur PE-1, qui sont mis en surbrillance en couleur rose.

Procédure de configuration

La tâche de configuration est divisée en un certain nombre de sous-tâches et chaque sous-tâche est mise en oeuvre sous une fonction définie par l'utilisateur. De cette manière, les fonctions peuvent être réutilisées si nécessaire.

Toutes les fonctions utilisent la bibliothèque « Requests » pour accéder aux API REST sur le routeur et le format des données est JSON. Dans les requêtes HTTP, le paramètre « verify » est défini sur « False » pour ignorer la validation du certificat SSL.

1. Récupérer l'ID de jeton

Avant de procéder à une configuration sur un routeur, vous devez obtenir un ID de jeton valide auprès du routeur. Cette fonction lance une requête HTTP pour authentifier et obtenir un ID de jeton afin de pouvoir appeler d'autres API utilisant ce jeton. La réponse à cette demande inclut un ID de jeton.

```
#----------------------------------

def getToken (ip, port, nom d'utilisateur, mot de passe) :

demandes d'importation

import base64

 url = "https://" + ip + ":"+ port + "/api/v1/auth/token-services"

 en-têtes = {

   'content-type' : "application/json",

   'autorisation' : "Basic " + base64.b64encode((username + ":" + password).encode('UTF-8')).decode('ascii'),
```

```
    'cache-control' : "no-cache"
 }
 response = requests.request("POST", url, headers=headers, verify=False )
if response.status_code == 200 :
 return response.json()['id_jeton']
autres :
 réapparaître "échec"
#---------------------------------
```

2. Créer un VRF

Cette fonction crée le VRF sur le routeur PE avec le marqueur de route (RD) et les cibles de route d'importation/exportation (RT) requis

```
#---------------------------------
def createVRF (ip, port, tokenID, vrfName, RD, importRT, exportRT) :
demandes d'importation
 url = "https://" + ip + ":"+ port + "/api/v1/vrf"
 en-têtes = {
   'content-type': "application/json",
   'X-auth-token': tokenID,
   'cache-control': "pas de cache"
 }
 données = {
   'nom': vrfName,
   'rd': RD,
   'route-target' : [
      {
          'action' : "importer",
```

```
            'community' : importRT
        },
        {
            'action' : "exporter",
            'community' : exportRT
        }
    ]
}
 response = requests.request("POST", url, headers=headers, json=data, verify=False )
if response.status_code == 201 :
    réapparaître "réussi"
autres :
    réapparaître "échec"
#----------------------------------
```

3. Déplacer l'interface dans un VRF

Cette fonction déplace une interface donnée dans un VRF.

```
#----------------------------------
def addInterfacetoVRF (ip, port, tokenID, vrfName, interfaceName, RD, importRT, exportRT) :
demandes d'importation
 url = "https://" + ip + ":"+ port + "/api/v1/vrf/" + vrfName
 en-têtes = {
    'content-type': "application/json",
    'X-auth-token' : tokenID,
    'cache-control': "pas de cache"
    }
 données = {
```

```
  'rd': RD,
  'transmission' : [ nomInterface ],
  'route-target' : [
     {
        'action' : "importer",
        'community' : importRT
     },
     {
        'action' : "exportation",
        'community' : exportRT
     }
  ]
}
 response = requests.request("PUT", url, headers=headers, json=data, verify=False )

if response.status_code == 204 :
  réapparaître "réussi"
autres :
  réapparaître "échec"
#---------------------------------
```

4. Attribuer une adresse IP à l'interface

Cette fonction attribuera une adresse IP à l'interface.

```
#---------------------------------
def assignInterfaceIP (ip, port, tokenID, interfaceName, interfaceIP, interfaceSubnet) :
demandes d'importation
 url = "https://" + ip + ":"+ port + "/api/v1/interfaces/" + interfaceName
 en-têtes = {
```

```
  'content-type': "application/json",
  'X-auth-token': tokenID,
  'cache-control': "pas de cache"
  }
 données = {
  'type': "ethernet",
  'if-name': interfaceName,
  'ip-address': interfaceIP,
  'masque-sous-réseau': interfaceSous-réseau
  }
 response = requests.request("PUT", url, headers=headers, json=data, verify=False )
if response.status_code == 204 :
  return "successful"
autres :
  return "failed"
#---------------------------------
```

5. Créer un bgp compatible VRF

Cela active la famille d'adresses VRF ipv4.

```
#---------------------------------
def createVrfBGP (ip, port, tokenID, vrfName, ASN) :
demandes d'importation
 url = "https://" + ip + ":"+ port + "/api/v1/vrf/" + vrfName + "/routing-svc/bgp"
 en-têtes = {
  'content-type': "application/json",
  'X-auth-token': tokenID,
  'cache-control': "pas de cache"
  }
```

```
 données = {
   'routing-protocol-id': ASN
   }
 response = requests.request("POST", url, headers=headers, json=data, verify=False )
if response.status_code == 201 :
   réapparaître "réussi"
autres :
   réapparaître "échec"
#----------------------------------
```

6. Définir le voisin BGP sous la famille d'adresses VRF

Cette fonction définit le voisin BGP sous la famille d'adresses VRF IPV4.

```
#----------------------------------
def defineVrfBGPNneighbor (ip, port, tokenID, vrfName, ASN, neighborIP, remoteAS) :
demandes d'importation
 url = "https://" + ip + ":"+ port + "/api/v1/vrf/" + vrfName + "/routing-svc/bgp/" + ASN +"/neighbors"
 en-têtes = {
   'content-type': "application/json",
   'X-auth-token': tokenID,
   'cache-control': "pas de cache"
   }
 données = {
   'routing-protocol-id': ASN,
   'address' : adresseIP du voisin,
   'remote-as': remoteAS
   }
 response = requests.request("POST", url, headers=headers, json=data, verify=False )
 if response.status_code == 201 :
```

```
   réapparaître "réussi"

autres :

   réapparaître "échec"

#---------------------------------
```

Description et valeurs des paramètres d'entrée

```
ip = "10.0.0.1"                          # adresse ip du routeur

port = "55443"                           # Port API REST sur le routeur

nom d'utilisateur = "cisco"                    # username to login. Ce nom doit être configuré
avec le niveau de privilège 15.

mot de passe = "cisco"                       # mot de passe associé au nom d'utilisateur

tokenID = <valeur renvoyée> # ID de jeton obtenu du routeur à l'aide de la fonction getToken

vrfName = "VRF-A" # nom du VRF

RD = "3:3" # Distincteur de route pour VRF

importRT = "34:34" # Import Route Target

exportRT = "34:34" # export Route Target

interfaceName = "GigabitEthernet3" # nom de l'interface faisant face à la périphérie du client (CE)

interfaceIP = "192.168.13.3" # Adresse IP de l'interface orientée CE

interfaceSubnet = "255.255.255.0" # subnet de l'interface CE

ASN = "34" # BGP Numéro de système autonome du routeur PE

neighborIP = "192.168.13.1" # IP d'appairage BGP du routeur CE

remoteAS = "11" # Numéro de système autonome du routeur CE
```

Dans toutes les fonctions ci-dessus, des API dédiées ont été appelées pour chaque étape de configuration. L'exemple ci-dessous montre comment passer l'interface de ligne de commande IOS-XE, en général, dans le corps de l'appel de l'API REST. Cela peut être utilisé comme solution de contournement pour automatiser si une API particulière n'est pas disponible. Dans les fonctions ci-dessus, « content-type » est défini sur « application/json », mais dans l'exemple ci-dessous, « content-type » est défini sur « text/plain » car il analyse l'entrée CLI standard.

Cet exemple montre comment définir la description de l'interface GigabitEthernet3. La configuration peut être personnalisée en modifiant le paramètre « cliInput ».

```
#---------------------------------

def passCLInput (ip, port, tokenID) :
```

```
demandes d'importation

 url = "https://" + ip + ":"+ port + "/api/v1/global/running-config"

 en-têtes = {

   'content-type': "text/plain",

  'X-auth-token': tokenID,

   'cache-control': "pas de cache"

   }

 line1 = "Interface GigabitEthernet 3"

 line2 = "description Interface client"

 cliInput = line1 + "\r\n" + line2

 response = requests.request("PUT", url, headers=headers, data=cliInput, verify=False )

 print(response.text)

if response.status_code == 204 :

   réapparaître "Réussite"

autres :

   réapparaître "échec"

#---------------------------------
```

Références

- Guide de configuration logicielle des routeurs de services cloud Cisco CSR 1000v

https://www.cisco.com/c/en/us/td/docs/routers/csr1000/software/configuration/b_CSR1000v_Configuration_Guide /b_CSR1000v_Configuration_Guide_chapter_01101.html

- Guide de référence de gestion de l'API REST Cisco IOS XE

https://www.cisco.com/c/en/us/td/docs/routers/csr1000/software/restapi/restapi.html

Acronymes utilisés :

MPLS - Commutation multiprotocole par étiquette

L3 - Couche 3

VPN - Réseau privé virtuel

VRF - Transfert de route virtuelle

BGP - Border Gateway Protocol

REST - Transfert d'état représentatif

API - Interface de programme d'application

JSON - Notation d'objet Java Script

HTTP - Hyper Text Transfer Protocol

À propos de cette traduction
Cisco a traduit ce document en traduction automatisée vérifiée par une personne dans le cadre d'un service mondial permettant à nos utilisateurs d'obtenir le contenu d'assistance dans leur propre langue.

Il convient cependant de noter que même la meilleure traduction automatisée ne sera pas aussi précise que celle fournie par un traducteur professionnel.