Configurare il servizio MPLS L3VPN sul router PE utilizzando REST-API (IOS-XE)

Sommario

Introduzione

Questo documento dimostra l'uso della programmazione Python per effettuare il provisioning di una VPN MPLS L3VPN su un router Service Provider Edge (PE) utilizzando l'API REST. In questo esempio vengono utilizzati i router Cisco CSR1000v (IOS-XE) come router PE.

Contributo di: Anuradha Perera

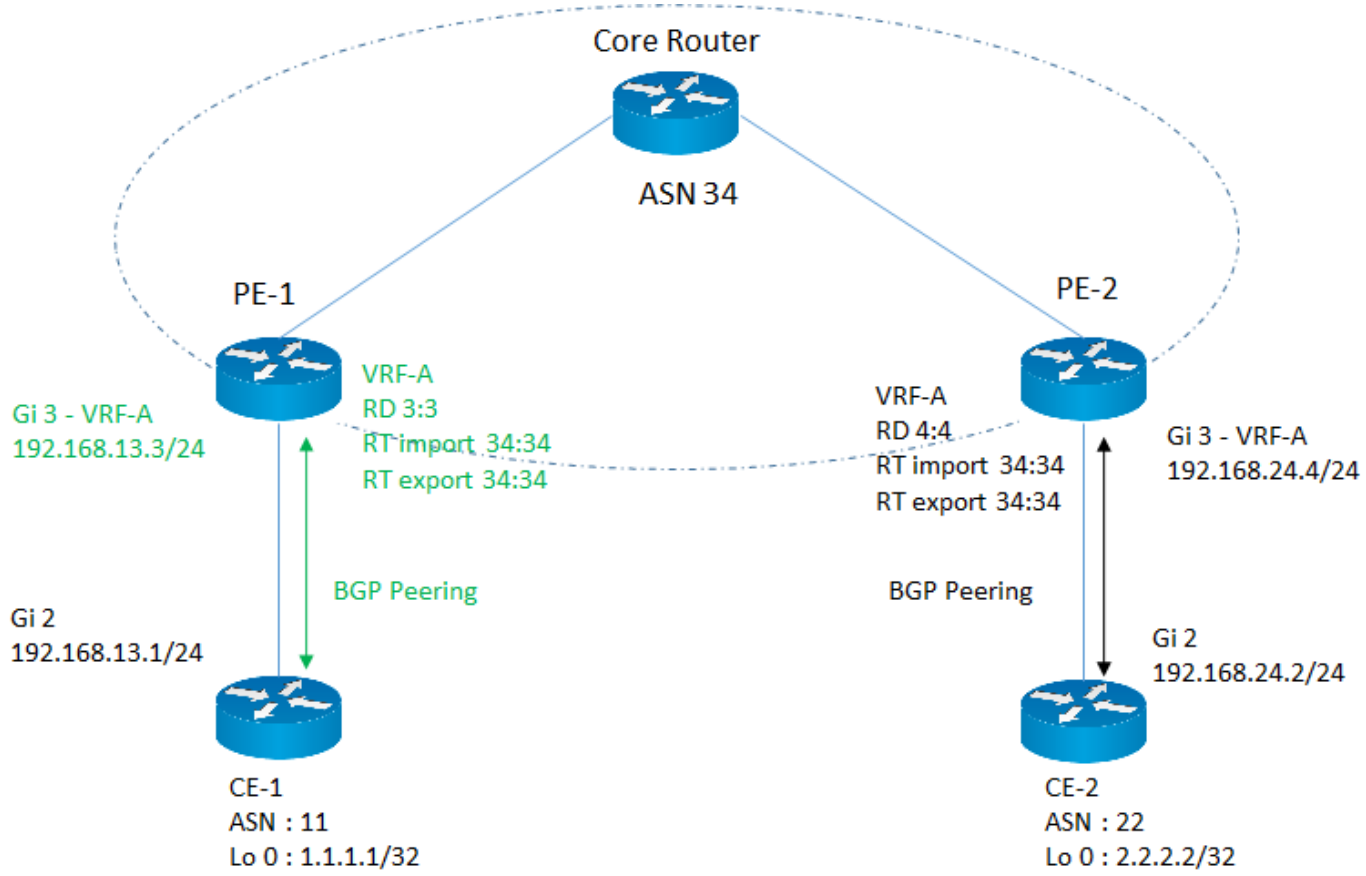
Modificato da: Kumar Sridhar

Prerequisiti

- Accesso di gestione dell'API REST ai router CSR1000v (fare riferimento ai riferimenti alla fine di questo documento).
- Python (versione 2.x o 3.x) e libreria Python "Richieste" installata sul computer utilizzato per la configurazione dei router.
- Alcune nozioni di base sulla programmazione Python.

Configurazione

Esempio di rete



In questo esempio viene descritta la configurazione dei parametri del servizio MPLS L3VPN richiesti sul router PE-1, evidenziati in rosa.

Procedura di configurazione

L'attività Configurazione è suddivisa in più sottoattività e ogni sottoattività è implementata in una funzione definita dall'utente. In questo modo le funzioni possono essere riutilizzate quando necessario.

Tutte le funzioni utilizzano la libreria "Requests" per accedere alle API REST sul router e il formato dei dati è JSON. Nelle richieste HTTP il parametro "verify" è impostato su "False" per ignorare la convalida del certificato SSL.

1. Recupera token-id

Prima di procedere con la configurazione su un router, è necessario avere un token-id valido ottenuto dal router. Questa funzione avvia una richiesta HTTP per autenticare e ottenere un token ID in modo che possa richiamare altre API utilizzando questo token. La risposta a questa richiesta include un token-id.

N.—

```
def getToken (ip, porta, nome utente, password):

richieste di importazione

importa base64

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

 headers = {

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

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

```
    'cache-control': "no-cache"
  }
  response = request.request("POST", url, headers=headers, verify=False )
if response.status_code == 200:
  return response.json()['token-id']
Altrimenti:
  ritorno "non riuscito"
N.—
```

2. Creare VRF

Questa funzione crea il VRF sul router PE con il necessario identificatore di percorso (RD) e gli oggetti di percorso di importazione/esportazione (RT)

```
N.—
def createVRF (ip, porta, tokenID, vrfName, RD, importRT, exportRT):
richieste di importazione
  url = "https://" + ip + ":"+ porta + "/api/v1/vrf"
  headers = {
    'content-type': "application/json",
    'X-auth-token': tokenID,
    'cache-control': "no-cache"
  }
  dati = {
    'name': vrfName
    'rd': RD
    'route-target': [
       {
          'azione' : "importazione",
```

```
            'community': importRT
        },
        {
            'azione' : "esportazione",
            'community': exportRT
        }
    ]
}
 response = request.request("POST", url, headers=headers, json=data, verify=False )
if response.status_code == 201:
    ritorno "riuscito"
Altrimenti:
    ritorno "non riuscito"
N.—
```

3. Spostamento dell'interfaccia in un VRF

Questa funzione sposta una determinata interfaccia in un VRF.

```
N.—
def addInterfaceVRF (ip, port, tokenID, vrfName, interfaceName, RD, importRT, exportRT):
richieste di importazione
 url = "https://" + ip + ":"+ porta + "/api/v1/vrf/" + vrfName
 headers = {
    'content-type': "application/json",
    'X-auth-token': tokenID,
    'cache-control': "no-cache"
    }
 dati = {
```

```
  'rd': RD
  'forwarding': [ nomeInterfaccia ],
  'route-target': [
     {
        "azione" : "importazione",
        'community': importRT
     },
     {
        "azione" : "esportazione",
        'community': exportRT
     }
  ]
}
 response = request.request("PUT", url, headers=headers, json=data, verify=False )


if response.status_code == 204:
  ritorno "riuscito"
Altrimenti:
  ritorno "non riuscito"
```

N.—

4. Assegna indirizzo IP all'interfaccia

Questa funzione assegna l'indirizzo IP all'interfaccia.

N.—

```
def assignInterfaceIP (ip, port, tokenID, interfaceName, interfaceIP, interfaceSubnet):
richieste di importazione
 url = "https://" + ip + ":"+ porta + "/api/v1/interfaces/" + interfaceName
 headers = {
```

```
  'content-type': "application/json",
  'X-auth-token': tokenID,
  'cache-control': "no-cache"
  }
 dati = {
  'tipo': "ethernet",
  'if-name': interfaceName;
  'ip-address': interfaceIP,
  'subnet-mask': interfaceSubnet
  }
 response = request.request("PUT", url, headers=headers, json=data, verify=False )
if response.status_code == 204:
  restituisci "riuscito"
Altrimenti:
  restituisci "non riuscito"
N.—
```

5. Creare bgp con riconoscimento VRF

In questo modo verrà abilitata la famiglia di indirizzi VRF ipv4.

```
N.—
def createVrfBGP (ip, porta, tokenID, vrfName, ASN):
richieste di importazione
 url = "https://" + ip + ":"+ porta + "/api/v1/vrf/" + vrfName + "/routing-svc/bgp"
 headers = {
  'content-type': "application/json",
  'X-auth-token': tokenID,
  'cache-control': "no-cache"
  }
```

```
 dati = {
   'routing-protocol-id': ASN
   }
 response = request.request("POST", url, headers=headers, json=data, verify=False )
if response.status_code == 201:
   ritorno "riuscito"
Altrimenti:
   ritorno "non riuscito"
```

N.—

6. Definire il sistema BGP adiacente nella famiglia di indirizzi VRF

Questa funzione definirà il router adiacente BGP nella famiglia di indirizzi VRF IPV4.

N.—

```
def defineVrfBGPNighbor (ip, porta, tokenID, vrfName, ASN, neighbourIP, remoteAS):
richieste di importazione
 url = "https://" + ip + ":"+ porta + "/api/v1/vrf/" + vrfName + "/routing-svc/bgp/" + ASN +"/neighbors"
 headers = {
   'content-type': "application/json",
   'X-auth-token': tokenID,
   'cache-control': "no-cache"
   }
 dati = {
   'routing-protocol-id': ASN,
   'indirizzo': neighborsIP,
   'remote-as': remoteAS
   }
 response = request.request("POST", url, headers=headers, json=data, verify=False )
 if response.status_code == 201:
```

```
    ritorno "riuscito"
Altrimenti:
    ritorno "non riuscito"
N.—
```

Descrizione e valori dei parametri input

```
ip = "10.0.0.1 "                          # indirizzo ip del router
port = porta "55443"                      # Porta API REST su router
username = "cisco"                        # nome utente per l'accesso.Deve essere configurato con
il livello di privilegio 15.
password = "cisco"                        # password associata al nome utente
tokenID = <valore restituito> # Token ID ottenuto dal router tramite la funzione getToken
vrfName = "VRF-A" # nome del VRF
RD = "3:3" # Distintore di route per VRF
importRT = "34:34" # Destinazione route di importazione
exportRT = "34:34" # destinazione route di esportazione
interfaceName = "Gigabit Ethernet3" # nome dell'interfaccia lato cliente (CE)
interfaceIP = "192.168.13.3" # indirizzo IP dell'interfaccia CE
interfaceSubnet = "255.255.255.0" # subnet dell'interfaccia di sfacciatura CE
ASN = "34" # BGP AS numero di router PE
neighbourIP = "192.168.13.1" # IP di peering BGP del router CE
remoteAS = "11" # Numero AS del router CE
```

In tutte le funzioni precedenti, sono state chiamate API dedicate per ogni configurazione. L'esempio seguente mostra come passare la CLI di IOS-XE, in generale, nel corpo della chiamata API REST. Questa opzione può essere utilizzata come soluzione per automatizzare il processo nel caso in cui una particolare API non sia disponibile. Nelle funzioni precedenti 'content-type' è impostato su 'application/json', ma nell'esempio seguente 'content-type' è impostato su 'text/plain' come viene analizzato l'input CLI standard.

Nell'esempio viene definita la descrizione dell'interfaccia per l'interfaccia Gigabit Ethernet3. La configurazione può essere personalizzata modificando il parametro "cliInput".

```
N.—
def passCLIInput (ip, porta, tokenID):
```

```
richieste di importazione

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

 headers = {

   'tipo di contenuto': "testo/semplice",

  'X-auth-token': tokenID,

   'cache-control': "no-cache"

   }

 line1 = "Interfaccia Gigabit Ethernet 3"

 line2 = "description Interfaccia a faccia cliente"

 cliInput = riga1 + "\r\n" + riga2

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

 stampa(response.text)

if response.status_code == 204:

   ritorno "riuscito"

Altrimenti:

   ritorno "non riuscito"
```

N.—

Riferimenti

- Guida alla configurazione di Cisco CSR serie 1000v Cloud Services Router

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

- Guida di riferimento per la gestione dell'API REST Cisco IOS XE

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

Acronimi usati:

MPLS - Multi Protocol Label Switching

L3 - Layer 3

VPN - Rete privata virtuale

VRF - Inoltro route virtuale

BGP - Border Gateway Protocol

REST - Trasferimento stato di rappresentanza

API - Interfaccia del programma applicativo

JSON - Notazione oggetto script Java

HTTP - Protocollo di trasferimento Hyper-Text

Informazioni su questa traduzione

Cisco ha tradotto questo documento utilizzando una combinazione di tecnologie automatiche e umane per offrire ai nostri utenti in tutto il mondo contenuti di supporto nella propria lingua. Si noti che anche la migliore traduzione automatica non sarà mai accurata come quella fornita da un traduttore professionista. Cisco Systems, Inc. non si assume alcuna responsabilità per l’accuratezza di queste traduzioni e consiglia di consultare sempre il documento originale in inglese (disponibile al link fornito).