Sviluppo, debug e installazione dell'applicazione Python NX-SDK negli switch Nexus 3000/9000

Sommario

Introduzione

Questo documento fornisce un flusso di lavoro per lo sviluppo di applicazioni Python con Cisco NX-Software Development Kit (SDK) e Cisco NX-OS sulle piattaforme Nexus 3000 e Nexus 9000.

Prerequisiti

Requisiti

Nessun requisito specifico previsto per questo documento.

Componenti usati

Le informazioni fornite in questo documento si basano sulle seguenti versioni software e hardware:

- Questo documento utilizza NX-SDK v1.0.0 e NX-SDK v1.5.0
- NX-SDK v1.0.0 può essere utilizzato sulla piattaforma Nexus 9000 a partire da NX-OS versione 7.0(3)I6(1) e dalla piattaforma Nexus 3000 a partire da NX-OS versione 7.0(3)I7(1)
- NX-SDK v1.5.0 può essere utilizzato sia sulla piattaforma Nexus 9000 che sulla piattaforma Nexus 3000 a partire da NX-OS versione 7.0(3)I7(3)

Le informazioni discusse in questo documento fanno riferimento a dispositivi usati in uno specifico ambiente di emulazione. Su tutti i dispositivi menzionati nel documento la configurazione è stata ripristinata ai valori predefiniti. Se la rete è operativa, valutare attentamente eventuali conseguenze derivanti dall'uso dei comandi.

Premesse

Cisco NX-SDK consente lo sviluppo di applicazioni personalizzate che possono essere eseguite in modo nativo in Cisco NX-OS sulle piattaforme Nexus 9000 e Nexus 3000. NX-SDK offre ai clienti la possibilità di creare comandi e output CLI personalizzati, generare syslog personalizzati in risposta a eventi specifici, telemetria personalizzata in streaming e molto altro ancora.

NX-SDK dispone di un'API C++, che è tradotta in altri linguaggi con l'uso di Wrapper semplificato e di Interface Generator (SWIG). Questo consente al cliente di utilizzare NX-SDK in qualsiasi lingua. Questo documento dimostra l'implementazione delle funzioni comuni di NX-SDK in Python, oltre a fornire un flusso di lavoro per i clienti per sviluppare le proprie applicazioni Python di NX-SDK.

Sviluppo di un'applicazione Python con NX-SDK

Abilita NX-SDK

Affinché un'applicazione NX-SDK possa essere eseguita, è necessario prima abilitare la funzionalità NX-SDK sul dispositivo:

```
switch(config)# feature nxsdk
```

Creazione di un file Python

È possibile creare e modificare un file Python utilizzando la shell Bash di NX-OS. Per poter utilizzare la shell Bash, è necessario prima abilitarla sul dispositivo:

```
switch(config)# feature bash-shell
```

Immettere la shell Bash e utilizzare l'editor di testo vi per creare e modificare il file Python:

```
switch(config)# run bash
bash-4.2$ vi /isan/bin/nxsdk-app.py
```

> **Nota:** È buona norma creare i file Python nella directory **/isan/bin/**. I file Python necessitano di autorizzazioni di esecuzione per poter essere eseguiti - non collocare i file Python nella

directory **/bootflash** o in una delle sue sottodirectory.

Nota: Non è necessario creare e modificare file Python tramite NX-OS. Lo sviluppatore può creare l'applicazione utilizzando il proprio ambiente locale e trasferire i file completati nel dispositivo utilizzando un protocollo di trasferimento file di sua scelta. Tuttavia, potrebbe essere più efficiente per lo sviluppatore eseguire il debug e risolvere i problemi relativi allo script utilizzando le utilità NX-OS.

Implementazione componenti NX-SDK

Si consiglia agli sviluppatori di iniziare a creare l'applicazione Python NX-SDK dal modello customCliPyApp su Cisco DevNet NX-SDK GitHub. Nelle sezioni seguenti di questo documento vengono illustrati i componenti necessari con l'utilizzo dei relativi nomi all'interno del modello.

Per un'applicazione Python NX-SDK sono necessari quattro componenti principali:

1. NX-SDK deve essere importato nell'applicazione tramite l'istruzione **import nx_sdk_py**.
2. Funzione (in genere denominata **sdkThread**) che avvia l'applicazione NX-SDK e modifica diverse opzioni correlate all'applicazione.
3. Creazione di comandi CLI personalizzati e definizione della sintassi dei comandi CLI personalizzati all'interno della funzione **sdkThread**.
4. Una classe denominata **pyCmdHandler** con un metodo denominato **postCliCb**, che elabora i comandi CLI personalizzati aggiunti dall'applicazione NX-SDK.

Funzione sdkThread

La funzione sdkThread inizializza, aggiunge funzionalità e avvia l'applicazione NX-SDK. La funzione non richiede il passaggio di parametri. Tutte le applicazioni Python NX-SDK richiedono tre metodi dalla libreria nx_sdk_py da chiamare:

1. **nx_sdk_py.NxSdk.getSdkInst(len(sys.argv), sys.argv)** restituisce un oggetto istanza SDK o None. Se viene restituito un oggetto istanza SDK, l'applicazione NX-SDK è stata correttamente registrata con l'infrastruttura NX-OS. Se viene restituito Nessuno, gli errori si verificano al momento della registrazione e nel syslog della periferica viene visualizzata una voce del registro errori. Questo metodo è documentato qui.

   Nota: A partire da NX-SDK v1.5.0, è possibile passare un terzo parametro booleano al metodo **NxSdk.getSdkInst**, che attiva Eccezioni avanzate quando è True e disattiva Eccezioni avanzate quando è False. Questo metodo è documentato qui .

1. **sdk.startEventLoop()**, dove **sdk** è l'oggetto istanza SDK restituito dal metodo **nx_sdk_py.NxSdk.getSdkInst()**. Questo metodo consente di avviare l'applicazione NX-SDK e di interagire con l'infrastruttura NX-OS. Questo metodo è documentato qui .
2. **nx_sdk_py.NxSdk.__swig_destroy_(sdk)**, dove **sdk** è l'oggetto istanza SDK restituito dal

metodo **nx_sdk_py.NxSdk.getSdkInst()** descritto in precedenza. Questo metodo, posizionato alla fine della funzione **sdkThread**, consente l'uscita graduale dell'applicazione NX-SDK.

Di seguito sono riportati alcuni metodi comunemente utilizzati.

- **sdk.getTracer()**, dove **sdk** è l'oggetto istanza SDK restituito dal metodo **nx_sdk_py.NxSdk.getSdkInst()**. Questo metodo restituisce un oggetto NxTrace che può essere utilizzato per generare syslog personalizzati, nonché per registrare eventi ed errori nella cronologia eventi dell'applicazione. I syslog personalizzati verranno visualizzati nel syslog del dispositivo (visibile tramite il comando **show logging logfile**), mentre gli eventi registrati nella cronologia eventi dell'applicazione sono visibili tramite i comandi **show <application-name> nxsdk event-history** o **show <application-name> nxsdk event-history**. Questo metodo è documentato qui. L'oggetto NxTrace restituito da questo metodo e i relativi metodi associati sono documentati qui. Ad esempio, è possibile visualizzare la cronologia degli eventi di un'applicazione denominata **Transceiver_DOM.py** tramite i comandi **show Transceiver_DOM.py nxsdk event-history** e **show Transceiver_DOM.py nxsdk event-history errors**.
- **sdk.getCliParser()**, dove **sdk** è l'oggetto istanza SDK restituito dal metodo **nx_sdk_py.NxSdk.getSdkInst()**. Questo metodo restituisce un oggetto NxCliParser, che può essere utilizzato per eseguire i comandi CLI già esistenti tramite Python e per creare comandi CLI personalizzati. Questo metodo è documentato qui. L'oggetto NxCliParser restituito da questo metodo e i relativi metodi associati sono documentati qui.
- **cliP.execShowCmd("cmd", tipo_restituito)**, dove **cliP** è l'oggetto NxCliParser restituito dal metodo **sdk.getCliParser()**, **cmd** è il comando show che si desidera eseguire incapsulato tra virgolette e **tipo_restituito** è il formato di dati da restituire. Il formato dati può essere **R_TEXT**, **R_JSON** o **R_XML**. Questo metodo è documentato qui .

  Nota: I formati di dati **R_JSON** e **R_XML** funzionano solo se il comando supporta l'output in tali formati. In NX-OS è possibile verificare se un comando supporta l'output in un particolare formato dati reindirizzando l'output al formato dati richiesto. Se il comando piped restituisce un output significativo, il formato dei dati è supportato. Ad esempio, se si esegue **show mac address-table dynamic | json** in NX-OS restituisce l'output JSON, quindi il formato dati R_JSON è supportato anche in NX-SDK.

- **cliP.execConfigCmd(cmd_filename)**, dove **cliP** è l'oggetto NxCliParser restituito dal metodo **sdk.getCliParser()** e **cmd_filename** è il percorso assoluto di un file contenente comandi separati da righe. Questo metodo restituisce una stringa che indica la riuscita dell'esecuzione del comando. Se nella stringa è presente "SUCCESS", tutti i comandi vengono eseguiti correttamente. In caso contrario, la stringa contiene l'eccezione che descrive il motivo per cui l'esecuzione dei comandi non è riuscita. Questo metodo è documentato qui.

Alcuni metodi facoltativi che possono essere utili sono:

- **sdk.setAppDesc('description string')**, dove **sdk** è l'oggetto istanza SDK restituito dal metodo **nx_sdk_py.NxSdk.getSdkInst()**. Questo metodo imposta la descrizione dell'applicazione NX-SDK. La descrizione viene visualizzata nel menu della guida contestuale di NX-OS, a cui si accede tramite un punto interrogativo sulla CLI. Questo metodo è documentato qui. Ad esempio, un'applicazione denominata **Transceiver_DOM.py**, una descrizione dell'applicazione di Restituisce tutte le interfacce con ricetrasmettitori compatibili con DOM inseriti, viene visualizzata nella Guida sensibile al contesto di NX-OS nel modo seguente:

```
 N9K-C93180LC-EX# show Tra?
track Tracking information
Transceiver_DOM.py Returns all interfaces with DOM-capable transceivers inserted
```

- **sdk.getAppName()**, dove **sdk** è l'oggetto istanza SDK restituito dal metodo **nx_sdk_py.NxSdk.getSdkinst()**. Questo metodo restituisce il nome dell'applicazione Python. Questo metodo è documentato qui.

Creazione di comandi CLI personalizzati

In un'applicazione Python con l'utilizzo di NX-SDK, i comandi CLI personalizzati vengono creati e definiti all'interno della funzione sdkThread. Esistono due tipi di comandi: **Mostra** comandi e **Config** comandi.

1. I comandi **show** visualizzano informazioni sul dispositivo, la sua configurazione o il suo ambiente.
2. I comandi **Config** modificano la configurazione del dispositivo, modificando la modalità di reazione del dispositivo alla rete circostante.

I due metodi seguenti consentono di creare rispettivamente i comandi show e config:

- **cliP.newShowCmd("cmd_name", "syntax")**, dove **cliP** è l'oggetto NxCliParser restituito dal metodo **sdk.getCliParser()**, **cmd_name** è un nome univoco per il comando interno all'applicazione NX-SDK personalizzata e **syntax** descrive le parole chiave e i parametri che è possibile utilizzare nel comando. Questo metodo restituisce un oggetto NxCliCmd, come illustrato qui. Questo metodo è documentato qui.

  Nota: Questo comando è una sottoclasse di cliP.newCliCmd("cmd_type", "cmd_name", "syntax") dove cmd_type è CONF_CMD o **SHOW_CMD** (a seconda del tipo di comando configurato), **cmd_name** è un nome univoco per il comando interno all'applicazione NX-SDK personalizzata e **syntax** descrive le parole chiave e i parametri che possono essere utilizzati nel comando. Per questo motivo, la documentazione API per questo comando potrebbe essere più utile come riferimento.

- **cliP.newConfigCmd("cmd_name", "syntax")**, dove **cliP** è l'oggetto NxCliParser restituito dal metodo **sdk.getCliParser()**, **cmd_name** è un nome univoco per il comando interno all'applicazione NX-SDK personalizzata e **syntax** descrive le parole chiave e i parametri che è possibile utilizzare nel comando. Questo metodo restituisce un oggetto NxCliCmd, come documentato qui . Questo metodo è documentato qui .

  Nota: Questo comando è una sottoclasse di cliP.newCliCmd("cmd_type", "cmd_name", "syntax") dove cmd_type è CONF_CMD o **SHOW_CMD** (dipende dal tipo di comando configurato), **cmd_name** è un nome univoco per il comando interno all'applicazione NX-SDK personalizzata e **syntax** descrive le parole chiave e i parametri che è possibile utilizzare nel comando. Per questo motivo, la documentazione API per questo comando potrebbe essere più utile come riferimento.

Entrambi i tipi di comandi hanno due componenti diversi: Parametri e parole chiave:

1. **I parametri** sono valori utilizzati per modificare i risultati del comando. Ad esempio, nel comando **show ip route 192.168.1.0**, è presente una parola chiave **route** seguita da un parametro che accetta un indirizzo IP, il quale specifica che devono essere visualizzate solo le route che

includono l'indirizzo IP fornito.

2. **Le parole chiave** modificano i risultati del comando esclusivamente tramite la loro presenza. Ad esempio, nel comando **show mac address-table dynamic**, è presente una parola chiave **dynamic** che specifica che devono essere visualizzati solo gli indirizzi MAC appresi in modo dinamico.

Entrambi i componenti vengono definiti nella sintassi di un comando NX-SDK al momento della creazione. Esistono metodi per l'oggetto NxCliCmd per modificare l'implementazione specifica di entrambi i componenti.

- **nx_cmd.updateParam("<parametro>", "help_str", tipo), dove nx_cmd è l'oggetto NxCliCmd restituito dai metodi cliP.newShowCmd() o cliP.newConfigCmd(), <parametro>** è il nome del parametro del comando che può essere modificato racchiuso tra parentesi quadre (**<>**), **help_str** imposta la stringa della guida del comando personalizzato e viene visualizzato nella Guida sensibile al contesto di NX-OS tramite un punto interrogativo sulla CLI e **type** è il tipo di parametro. Ulteriori parametri facoltativi per questo metodo sono disponibili e documentati qui. Di seguito sono riportati i tipi validi per i parametri che è possibile specificare nell'argomento Type:

  P_INTEGER - specifica qualsiasi numero intero**P_STRING** - specifica qualsiasi stringa**P_INTERFACE** - specifica qualsiasi interfaccia di rete**P_IP_ADDR** - specifica qualsiasi indirizzo IP**P_MAC_ADDR** - specifica qualsiasi indirizzo MAC**P_VRF** - specifica qualsiasi istanza VRF (Virtual Routing and Forwarding)
- **nx_cmd.updateKeyword("keyword", "help_str", is_key), dove nx_cmd è l'oggetto NxCliCmd restituito dai metodi cliP.newShowCmd() o cliP.newConfigCmd(), keyword** è il nome della parola chiave del comando che si desidera modificare, **help_str** imposta la stringa della guida del comando personalizzato e viene visualizzato nel menu della guida sensibile al contesto di NX-OS accessibile tramite un punto interrogativo CLI e **is_key** è un valore booleano facoltativo il cui valore predefinito è False. Se **is_key** è True, la configurazione univoca creata dal comando tramite questa parola chiave non sovrascrive le altre configurazioni univoche create dal comando. Se **is_key** è False, la configurazione creata dal comando tramite questa parola chiave sovrascrive l'altra configurazione creata dal comando. Questo metodo è documentato qui .

Per visualizzare esempi di codice dei componenti dei comandi di uso comune, consultare la sezione Esempi di comandi CLI personalizzati in questo documento.

Dopo aver creato i comandi CLI personalizzati, è necessario creare un oggetto dalla classe **pyCmdHandler** descritta più avanti in questo documento e impostarlo come oggetto gestore di callback CLI per l'oggetto NxCliParser. La dimostrazione è la seguente:

```
 cmd_handler = pyCmdHandler()
cliP.setCmdHandler(cmd_handler)
```

Quindi, è necessario aggiungere l'oggetto NxCliParser all'albero del parser CLI di NX-OS in modo che i comandi CLI personalizzati siano visibili all'utente. A tale scopo, è necessario utilizzare il comando **cliP.addToParseTree()**, dove **cliP** è l'oggetto NxCliParser restituito dal metodo **sdk.getCliParser()**.

Esempio di funzione sdkThread

Di seguito è riportato un esempio di una tipica funzione sdkThread con l'utilizzo delle funzioni descritte in precedenza. Questa funzione (tra le altre all'interno di una tipica applicazione Python NX-SDK personalizzata) utilizza variabili globali, che vengono istanziate sull'esecuzione dello script.

```
cliP = ""
sdk = ""
event_hdlr = ""
tmsg = ""

def sdkThread():
    global cliP, sdk, event_hdlr, tmsg

    sdk = nx_sdk_py.NxSdk.getSdkInst(len(sys.argv), sys.argv)
    if not sdk:
       return

    sdk.setAppDesc("Returns all interfaces with DOM-capable transceivers inserted")

    tmsg = sdk.getTracer()
    tmsg.event("[{}] Started service".format(sdk.getAppName()))

    cliP = sdk.getCliParser()

    nxcmd = cliP.newShowCmd("show_port_bw_util_cmd", "port bw utilization [<port>]")
    nxcmd.updateKeyword("port", "Port Information")
    nxcmd.updateKeyword("bw", "Port Bandwidth Information")
    nxcmd.updateKeyword("utilization", "Port BW utilization in (%)")
    nxcmd.updateParam("<port>", "Optional Filter Port Ex) Ethernet1/1", nx_sdk_py.P_INTERFACE)

    nxcmd1 = cliP.newConfigCmd("port_bw_threshold_cmd", "port bw threshold <threshold>")
    nxcmd1.updateKeyword("threshold", "Port BW Threshold in (%)")

    int_attr = nx_sdk_py.cli_param_type_integer_attr()
    int_attr.min_val = 1;
    int_attr.max_val = 100;
    nxcmd1.updateParam("<threshold>", "Threshold Limit. Default 50%", nx_sdk_py.P_INTEGER,
int_attr, len(int_attr))

    mycmd = pyCmdHandler()
    cliP.setCmdHandler(mycmd)

    cliP.addToParseTree()

    sdk.startEventLoop()

    # If sdk.stopEventLoop() is called or application is removed from VSH...
    tmsg.event("Service Quitting...!")

    nx_sdk_py.NxSdk.__swig_destroy__(sdk)
```

Classe pyCmdHandler

La classe **pyCmdHandler** viene ereditata dalla classe **NxCmdHandler** all'interno della libreria nx_sdk_py. Il metodo **postCliCb(self, clicmd)** definito nella classe **pyCmdHandler** viene chiamato ogni volta che i comandi CLI provengono da un'applicazione NX-SDK. Di conseguenza, il metodo **postCliCb(self, clicmd)** è il punto in cui si definisce il comportamento sul dispositivo dei comandi CLI personalizzati definiti nella funzione sdkThread.

La funzione **postCliCb(self, clicmd)** restituisce un valore booleano. Se viene restituito **True**, si presume che il comando sia stato eseguito correttamente. **Il valore False** deve essere restituito se il comando non è stato eseguito correttamente per qualsiasi motivo.

Il parametro **clicmd** utilizza il nome univoco definito per il comando al momento della creazione nella funzione sdkThread. Ad esempio, se si crea un nuovo comando **show** con un nome univoco **show_xcvr_dom**, è consigliabile fare riferimento a questo comando con lo stesso nome nella funzione **postCliCb(self, clicmd)** dopo aver verificato se il nome dell'argomento clicmd contiene **show_xcvr_dom**. La dimostrazione è la seguente:

```
def sdkThread():
    <snip>
    sh_xcvr_dom = cliP.newShowCmd("show_xcvr_dom", "dom")
    sh_xcvr_dom.updateKeyword("dom", "Show all interfaces with transceivers that are DOM-
capable")
    </snip>

class pyCmdHandler(nx_sdk_py.NxCmdHandler):
    def postCliCb(self, clicmd):
        if "show_xcvr_dom" in clicmd.getCmdName():
            get_dom_capable_interfaces()
```

Se viene creato un comando che utilizza parametri, è molto probabile che sia necessario utilizzare tali parametri in un determinato punto della funzione **postCliCb(self, clicmd)**. A tale scopo, è possibile utilizzare il metodo **clicmd.getParamValue("<parameter>")**, dove **<parameter>** è il nome del parametro del comando di cui si desidera ottenere il valore racchiuso tra parentesi angolari (**<>**). Questo metodo è documentato qui. Tuttavia, il valore restituito da questa funzione deve essere convertito nel tipo necessario. A tale scopo, è possibile utilizzare i metodi seguenti:

- **nx_sdk_py.void_to_int** converte un valore in un tipo integer.
- **nx_sdk_py.void_to_string** converte un valore in un tipo stringa.

La funzione **postCliCb(self, clicmd)** (o qualsiasi funzione successiva) si trova generalmente anche dove l'output del comando show viene stampato sulla console. Questa operazione viene eseguita con il metodo **clicmd.printConsole()**.

> **Nota:** Se l'applicazione rileva un errore, un'eccezione non gestita o si chiude improvvisamente, l'output della funzione **clicmd.printConsole()** non verrà visualizzato. Per questo motivo, quando si esegue il debug dell'applicazione Python, è consigliabile registrare i messaggi di debug nel syslog utilizzando un oggetto NxTrace restituito dal metodo **sdk.getTracer()** oppure utilizzare le istruzioni print ed eseguire l'applicazione tramite il binario **/isan/bin/python della** shell Bash.

Esempio di classe pyCmdHandler

Il codice seguente funge da esempio per la classe **pyCmdHandler** descritta in precedenza. Questo codice è tratto dal file ip_move.py nell'applicazione ip-move NX-SDK disponibile qui. Lo scopo di questa applicazione è tenere traccia dello spostamento di un indirizzo IP definito dall'utente tra le interfacce di un dispositivo Nexus. A tale scopo, il codice trova l'indirizzo MAC dell'indirizzo IP immesso tramite il parametro **<ip>** nella cache ARP del dispositivo, quindi verifica la VLAN in cui risiede tale indirizzo MAC utilizzando la tabella degli indirizzi MAC del dispositivo. Utilizzando questo indirizzo MAC e la VLAN, il comando **show system internal l2fm l2dbg macdb address**

<mac> vlan <vlan> visualizza un elenco di indici dell'interfaccia SNMP a cui questa combinazione è stata recentemente associata. Il codice utilizza quindi il comando **show interface snmp-ifindex** per convertire gli indici recenti dell'interfaccia SNMP in nomi di interfaccia leggibili.

```
class pyCmdHandler(nx_sdk_py.NxCmdHandler):
    def postCliCb(self, clicmd):
        global cli_parser

        if "show_ip_movement" in clicmd.getCmdName():
            target_ip = nx_sdk_py.void_to_string(clicmd.getParamValue("<ip>"))

            target_mac = get_mac_from_arp(cli_parser, clicmd, target_ip)
            mac_vlan = ""
            if target_mac:
                mac_vlan = get_vlan_from_cam(cli_parser, clicmd, target_mac)
                if mac_vlan:
                    find_mac_movement(cli_parser, clicmd, target_mac, mac_vlan)
                else:
                    print("No entires in MAC address table")
                    clicmd.printConsole("No entries in MAC address table for
{}".format(target_mac))
            else:
                clicmd.printConsole("No entries in ARP table for {}".format(target_ip))
        return True

def get_mac_from_arp(cli_parser, clicmd, target_ip):
    exec_cmd = "show ip arp {}".format(target_ip)
    arp_cmd = cli_parser.execShowCmd(exec_cmd, nx_sdk_py.R_JSON)
    if arp_cmd:
        try:
            arp_json = json.loads(arp_cmd)
        except ValueError as exc:
            return None
        count = int(arp_json["TABLE_vrf"]["ROW_vrf"]["cnt-total"])
        if count:
            intf = arp_json["TABLE_vrf"]["ROW_vrf"]["TABLE_adj"]["ROW_adj"]
            if intf.get("ip-addr-out") == target_ip:
                target_mac = intf["mac"]
                clicmd.printConsole("{} is currently present in ARP table, MAC address
{}\n".format(target_ip, target_mac))
                return target_mac
            else:
                return None
        else:
            return None
    else:
        return None

def get_vlan_from_cam(cli_parser, clicmd, target_mac):
    exec_cmd = "show mac address-table address {}".format(target_mac)
    mac_cmd = cli_parser.execShowCmd(exec_cmd, nx_sdk_py.R_JSON)
    if mac_cmd:
        try:
            cam_json = json.loads(mac_cmd)
        except ValueError as exc:
            return None
        mac_entry = cam_json["TABLE_mac_address"]["ROW_mac_address"]
        if mac_entry:
            if mac_entry["disp_mac_addr"] == target_mac:
                egress_intf = mac_entry["disp_port"]
                mac_vlan = mac_entry["disp_vlan"]
                clicmd.printConsole("{} is currently present in MAC address table on interface
```

```
{}, VLAN {}\n".format(target_mac, egress_intf, mac_vlan))
                return mac_vlan
            else:
                return None
        else:
            return None
    else:
        return None

def find_mac_movement(cli_parser, clicmd, target_mac, mac_vlan):
    exec_cmd = "show system internal l2fm l2dbg macdb address {} vlan {}".format(target_mac,
mac_vlan)
    l2fm_cmd = cli_parser.execShowCmd(exec_cmd)
    if l2fm_cmd:
        event_re = re.compile(r"^\s+(\w{3}) (\w{3}) (\d+) (\d{2}):(\d{2}):(\d{2}) (\d{4})
(0x\S{8}) (\d+)\s+(\S+) (\d+)\s+(\d+)\s+(\d+)")
        unique_interfaces = []
        l2fm_events = l2fm_cmd.splitlines()
        for line in l2fm_events:
            res = re.search(event_re, line)
            if res:
                day_name = res.group(1)
                month = res.group(2)
                day = res.group(3)
                hour = res.group(4)
                minute = res.group(5)
                second = res.group(6)
                year = res.group(7)
                if_index = res.group(8)
                db = res.group(9)
                event = res.group(10)
                src=res.group(11)
                slot = res.group(12)
                fe = res.group(13)
                if "MAC_NOTIF_AM_MOVE" in event:
                    timestamp = "{} {} {} {}:{}:{} {}".format(day_name, month, day, hour,
minute, second, year)
                    intf_dict = {"if_index": if_index, "timestamp": timestamp}
                    unique_interfaces.append(intf_dict)
        if not unique_interfaces:
            clicmd.printConsole("No entries for {} in L2FM L2DBG\n".format(target_mac))
        if len(unique_interfaces) == 1:
            clicmd.printConsole("{} has not been moving between
interfaces\n".format(target_mac))
        if len(unique_interfaces) > 1:
                clicmd.printConsole("{} has been moving between the following interfaces, from
most recent to least recent:\n".format(target_mac))
                unique_interfaces = get_snmp_intf_index(unique_interfaces)
                clicmd.printConsole("\t{} - {} (Current interface)\n".format(unique_interfaces[-
1]["timestamp"], unique_interfaces[-1]["intf_name"]))
                for intf in unique_interfaces[-2::-1]:
                    clicmd.printConsole("\t{} - {}\n".format(intf["timestamp"],
intf["intf_name"]))
def get_snmp_intf_index(if_index_dict_list): global cli_parser snmp_ifindex =
cli_parser.execShowCmd("show interface snmp-ifindex", nx_sdk_py.R_JSON) snmp_ifindex_json =
json.loads(snmp_ifindex) snmp_ifindex_list =
snmp_ifindex_json["TABLE_interface"]["ROW_interface"] for index_dict in if_index_dict_list:
index = index_dict["if_index"] for ifindex_json in snmp_ifindex_list: if index ==
ifindex_json["snmp-ifindex"]: index_dict["intf_name"] = ifindex_json["interface"] return
if_index_dict_list
```

Esempi di sintassi dei comandi CLI personalizzati

In questa sezione vengono illustrati alcuni esempi di parametri di sintassi utilizzati quando si creano comandi CLI personalizzati con i metodi **cliP.newShowCmd()** o **cliP.newConfigCmd()**, dove **cliP** è l'oggetto NxCliParser restituito dal metodo **sdk.getCliParser()**.

> **Nota:** Il supporto della sintassi con parentesi di apertura e chiusura ("(" e ")") è introdotto in NX-SDK v1.5.0, incluso in NX-OS release 7.0(3)I7(3). Si presume che l'utente utilizzi NX-SDK v1.5.0 quando segue uno di questi esempi forniti che includono la sintassi che utilizza le parentesi di apertura e chiusura.

Parola chiave singola

Questo comando show accetta un singolo mac di parole chiave e aggiunge alla parola chiave una stringa di supporto Mostra tutti gli indirizzi MAC non programmati sul dispositivo.

```
 nx_cmd = cliP.newShowCmd("show_misprogrammed", "mac")
nx_cmd.updateKeyword("mac", "Shows all misprogrammed MAC addresses on this device")
```

Parametro singolo

Questo comando show accetta un singolo parametro **<mac>**. Le parentesi angolari che racchiudono la parola mac indicano che si tratta di un parametro. Al parametro viene aggiunta una stringa di supporto dell'indirizzo MAC per verificare la presenza di errori di programmazione. Il parametro **nx_sdk_py.P_MAC_ADDR** nel metodo **nx_cmd.updateParam()** viene utilizzato per definire il tipo del parametro come indirizzo MAC, impedendo l'input dell'utente finale di un altro tipo, ad esempio una stringa, un numero intero o un indirizzo IP.

```
 nx_cmd = cliP.newShowCmd("show_misprogrammed_mac", "<mac>")
nx_cmd.updateParam("<mac>", "MAC address to check for misprogramming", nx_sdk_py.P_MAC_ADDR)
```

Parola chiave opzionale

Il comando show accetta facoltativamente una singola parola chiave **[mac]**. Le parentesi che racchiudono la parola mac indicano che questa parola chiave è facoltativa. Alla parola chiave viene aggiunta una stringa di supporto Mostra tutti gli indirizzi MAC non programmati nel dispositivo.

```
 nx_cmd = cliP.newShowCmd(  "show_misprogrammed_mac"  ,   "[mac]"  )
nx_cmd.updateKeyword( "mac" , "Shows all misprogrammed MAC addresses on this device" )
```

Parametro facoltativo

Questo comando show accetta un singolo parametro **[<mac>]**. Le parentesi che racchiudono la parola < mac > indicano che questo parametro è facoltativo. Le parentesi angolari che racchiudono la parola mac indicano che si tratta di un parametro. Al parametro viene aggiunta una stringa di supporto dell'indirizzo MAC per verificare la presenza di errori di programmazione. Il parametro **nx_sdk_py.P_MAC_ADDR** nel metodo **nx_cmd.updateParam()** viene utilizzato per definire il tipo del parametro come indirizzo MAC, impedendo l'input dell'utente finale di un altro tipo, ad esempio una stringa, un numero intero o un indirizzo IP.

```
 nx_cmd = cliP.newShowCmd("show_misprogrammed_mac", "[<mac>]")
nx_cmd.updateParam("<mac>", "MAC address to check for misprogramming", nx_sdk_py.P_MAC_ADDR)
```

Parola chiave e parametro singoli

Il comando show accetta una singola parola chiave mac seguita immediatamente dal parametro **<mac-address>**. Le parentesi angolari che racchiudono la parola mac-address indicano che si tratta di un parametro. Alla parola chiave viene aggiunta una stringa di supporto di Controlla indirizzo MAC per errori di programmazione. Al parametro viene aggiunta una stringa di supporto dell'indirizzo MAC per verificare la presenza di errori di programmazione. Il parametro **nx_sdk_py.P_MAC_ADDR** nel metodo **nx_cmd.updateParam()** viene utilizzato per definire il tipo del parametro come indirizzo MAC, che impedisce l'input dell'utente finale di un altro tipo, ad esempio una stringa, un numero intero o un indirizzo IP.

```
 nx_cmd = cliP.newShowCmd("show_misprogrammed", "mac <mac-address>")
nx_cmd.updateKeyword("mac", "Check MAC address for misprogramming")
nx_cmd.updateParam("<mac-address>", "MAC address to check for misprogramming",
nx_sdk_py.P_MAC_ADDR)
```

Più parole chiave e parametri

Il comando show può accettare una di due parole chiave, entrambe con due parametri diversi. La prima parola chiave mac ha un parametro **<mac-address>**, mentre la seconda parola chiave ip ha un parametro **<ip-address>**. Le parentesi angolari che racchiudono le parole mac-address e ip-address indicano che si tratta di parametri. Alla parola chiave mac viene aggiunta una stringa di supporto di Controlla indirizzo MAC per errori di programmazione. Al parametro **<mac-address>** viene aggiunta una stringa di supporto dell'indirizzo MAC per verificare la presenza di errori di programmazione. Il parametro **nx_sdk_py.P_MAC_ADDR** nel metodo **nx_cmd.updateParam()** viene utilizzato per definire il tipo di parametro **<mac-address>** come indirizzo MAC, che impedisce l'input dell'utente finale di un altro tipo, ad esempio una stringa, un numero intero o un indirizzo IP. Alla parola chiave ip viene aggiunta una stringa di supporto di Verifica indirizzo IP per errori di programmazione. Al parametro **<ip-address>** viene aggiunta una stringa di supporto dell'indirizzo IP per verificare la presenza di errori di programmazione. Il parametro **nx_sdk_py.P_IP_ADDR** nel metodo **nx_cmd.updateParam()** viene utilizzato per definire il tipo del parametro **<ip-address>** come indirizzo IP, che impedisce l'input dell'utente finale di un altro tipo, ad esempio una stringa, un numero intero o un indirizzo IP.

```
 nx_cmd = cliP.newShowCmd("show_misprogrammed", "(mac <mac-address> | ip <ip-address>)")
nx_cmd.updateKeyword("mac", "Check MAC address for misprogramming")
nx_cmd.updateParam("<mac-address>", "MAC address to check for misprogramming",
nx_sdk_py.P_MAC_ADDR)
nx_cmd.updateKeyword("ip", "Check IP address for misprogramming")
nx_cmd.updateParam("<ip-address>", "IP address to check for misprogramming",
nx_sdk_py.P_IP_ADDR)
```

Parole chiave e parametri multipli con parola chiave opzionale

Il comando show può accettare una di due parole chiave, entrambe con due parametri diversi. La prima parola chiave mac ha un parametro **<mac-address>**, mentre la seconda parola chiave ip ha un parametro **<ip-address>**. Le parentesi angolari che racchiudono le parole mac-address e ip-address indicano che si tratta di parametri. Alla parola chiave mac viene aggiunta una stringa di supporto di Controlla indirizzo MAC per errori di programmazione. Al parametro **<mac-address>** viene aggiunta una stringa di supporto dell'indirizzo MAC per verificare la presenza di errori di

programmazione. Il parametro **nx_sdk_py.P_MAC_ADDR** nel metodo **nx_cmd.updateParam()** viene utilizzato per definire il tipo di parametro **<mac-address>** come indirizzo MAC, che impedisce l'input dell'utente finale di un altro tipo, ad esempio una stringa, un numero intero o un indirizzo IP. Alla parola chiave ip viene aggiunta una stringa di supporto di Verifica indirizzo IP per errori di programmazione. Al parametro **<ip-address>** viene aggiunta una stringa di supporto dell'indirizzo IP per verificare la presenza di errori di programmazione. Il parametro **nx_sdk_py.P_IP_ADDR** nel metodo **nx_cmd.updateParam()** viene utilizzato per definire il tipo del parametro **<ip-address>** come indirizzo IP, che impedisce l'input dell'utente finale di un altro tipo, ad esempio una stringa, un numero intero o un indirizzo IP. Per questo comando show è possibile usare la parola chiave [clear]. A questa parola chiave opzionale viene aggiunta una stringa di supporto che cancella gli indirizzi rilevati come non programmati.

```
 nx_cmd = cliP.newShowCmd("show_misprogrammed", "(mac <mac-address> | ip <ip-address>) [clear]")
nx_cmd.updateKeyword("mac", "Check MAC address for misprogramming")
nx_cmd.updateParam("<mac-address>", "MAC address to check for misprogramming",
nx_sdk_py.P_MAC_ADDR)
nx_cmd.updateKeyword("ip", "Check IP address for misprogramming")
nx_cmd.updateParam("<ip-address>", "IP address to check for misprogramming",
nx_sdk_py.P_IP_ADDR)
nx_cmd.updateKeyword("clear", "Clears addresses detected to be misprogrammed")
```

Parole chiave e parametri multipli con parametro facoltativo

Il comando show può accettare una di due parole chiave, entrambe con due parametri diversi. La prima parola chiave mac ha un parametro **<mac-address>**, mentre la seconda parola chiave ip ha un parametro **<ip-address>**. Le parentesi angolari che racchiudono le parole mac-address e ip-address indicano che si tratta di parametri. Alla parola chiave mac viene aggiunta una stringa di supporto di Controlla indirizzo MAC per errori di programmazione. Al parametro **<mac-address>** viene aggiunta una stringa di supporto dell'indirizzo MAC per verificare la presenza di errori di programmazione. Il parametro **nx_sdk_py.P_MAC_ADDR** nel metodo **nx_cmd.updateParam()** viene utilizzato per definire il tipo di parametro **<mac-address>** come indirizzo MAC, che impedisce l'input dell'utente finale di un altro tipo, ad esempio una stringa, un numero intero o un indirizzo IP. Alla parola chiave ip viene aggiunta una stringa di supporto di Verifica indirizzo IP per errori di programmazione. Al parametro **<ip-address>** viene aggiunta una stringa di supporto dell'indirizzo IP per verificare la presenza di errori di programmazione. Il parametro **nx_sdk_py.P_IP_ADDR** nel metodo **nx_cmd.updateParam()** viene utilizzato per definire il tipo del parametro **<ip-address>** come indirizzo IP, che impedisce l'input dell'utente finale di un altro tipo, ad esempio una stringa, un numero intero o un indirizzo IP. Questo comando show accetta facoltativamente un parametro **[<module>]**. Una stringa di supporto Cancella solo indirizzi nel modulo specificato viene aggiunta a questo parametro facoltativo.

```
 nx_cmd = cliP.newShowCmd("show_misprogrammed", "(mac <mac-address> | ip <ip-address>)
[<module>]")
nx_cmd.updateKeyword("mac", "Check MAC address for misprogramming")
nx_cmd.updateParam("<mac-address>", "MAC address to check for misprogramming",
nx_sdk_py.P_MAC_ADDR)
nx_cmd.updateKeyword("ip", "Check IP address for misprogramming")
nx_cmd.updateParam("<ip-address>", "IP address to check for misprogramming",
nx_sdk_py.P_IP_ADDR)
nx_cmd.updateParam("<module>", "Clears addresses detected to be misprogrammed",
nx_sdk_py.P_INTEGER)
```

Debug di un'applicazione Python con NX-SDK

Una volta creata un'applicazione Python NX-SDK, sarà spesso necessario eseguirne il debug. NX-SDK informa l'utente in caso di errori di sintassi nel codice, ma poiché la libreria Python NX-SDK utilizza SWIG per convertire le librerie C++ in librerie Python, qualsiasi eccezione rilevata al momento dell'esecuzione del codice genera un dump del core dell'applicazione simile al seguente:

```
 terminate called after throwing an instance of 'Swig::DirectorMethodException'
what(): SWIG director method error. Error detected when calling 'NxCmdHandler.postCliCb'
Aborted (core dumped)
```

A causa della natura ambigua di questo messaggio di errore, è consigliabile eseguire il debug delle applicazioni Python registrando i messaggi di debug nel syslog con l'utilizzo di un oggetto NxTrace restituito dal metodo **sdk.getTracer()**. La dimostrazione è la seguente:

```
#! /isan/bin/python

tracer = 0

def evt_thread():
    <snip>
    tracer = sdk.getTracer()
    tracer.event("[NXSDK-APP][INFO] Started service")
<snip>
class pyCmdHandler(nx_sdk_py.NxCmdHandler):
    def postCliCb(self, clicmd):
        global tracer
        tracer.event("[NXSDK-APP][DEBUG] Received command: {}".format(clicmd))
        if "show_test_command" in clicmd.getCmdName():
            tracer.event("[NXSDK-APP][DEBUG] `show_test_command` recognized")
```

Se la registrazione dei messaggi di debug nel syslog non è un'opzione, un metodo alternativo consiste nell'utilizzare le istruzioni print ed eseguire l'applicazione tramite il binario **/isan/bin/python della** shell Bash. Tuttavia, l'output di queste istruzioni di stampa sarà visibile solo se eseguite in questo modo: l'esecuzione dell'applicazione tramite la shell VSH non produce alcun output. Di seguito è riportato un esempio di utilizzo dei rendiconti di stampa:

```
#! /isan/bin/python

tracer = 0

def evt_thread():
    <snip>
    print("[NXSDK-APP][INFO] Started service")
<snip>
class pyCmdHandler(nx_sdk_py.NxCmdHandler):
    def postCliCb(self, clicmd):
        print("[NXSDK-APP][DEBUG] Received command: {}".format(clicmd))
        if "show_test_command" in clicmd.getCmdName():
            print("[NXSDK-APP][DEBUG] `show_test_command` recognized")
```

Distribuire un'applicazione Python con NX-SDK

Una volta che un'applicazione Python è stata completamente testata nella shell Bash ed è pronta per la distribuzione, l'applicazione deve essere installata in produzione tramite VSH. In questo modo, l'applicazione può essere mantenuta anche quando il dispositivo viene ricaricato o quando il sistema viene sostituito in uno scenario con due supervisori. Per distribuire un'applicazione

tramite VSH, è necessario creare un pacchetto RPM con l'utilizzo di un ambiente di generazione NX-SDK e ENXOS SDK. Cisco DevNet fornisce un'immagine Docker che consente la semplice creazione di pacchetti RPM.

> **Nota:** Per assistenza nell'installazione di Docker su un sistema operativo specifico, consultare la documentazione relativa all'installazione di Docker.

Su un host compatibile con Docker, trascinare la versione dell'immagine desiderata con il comando **docker pull dockercisco/nxsdk:<tag>**, dove **<tag>** è il tag della versione dell'immagine scelta. Qui è possibile visualizzare le versioni delle immagini disponibili e i tag corrispondenti. Questa condizione viene dimostrata con il tag **v1** riportato di seguito:

```
docker pull dockercisco/nxsdk:v1
```

Avviare un contenitore denominato **nxsdk** da questa immagine e collegarlo. Se il tag scelto è diverso, sostituire il tag con **v1**:

```
docker run -it --name nxsdk dockercisco/nxsdk:v1 /bin/bash
```

Aggiornare alla versione più recente di NX-SDK e passare alla directory di **NX-SDK**, quindi estrarre i file più recenti da git:

```
cd /NX-SDK/
git pull
```

Se è necessario utilizzare una versione precedente di NX-SDK, è possibile clonare il ramo di NX-SDK utilizzando il relativo tag di versione con il comando **git clone -b v<version>** https://github.com/CiscoDevNet/NX-SDK.git, dove **<version>** è la versione di NX-SDK necessaria. Ciò è dimostrato in NX-SDK v1.0.0:

```
cd /
rm -rf /NX-SDK
git clone -b v1.0.0 https://github.com/CiscoDevNet/NX-SDK.git
```

Quindi, trasferire l'applicazione Python al contenitore Docker. Ci sono diversi modi per farlo.

- Uscire dal contenitore Docker (che arresta il contenitore e richiede di riavviarlo), trasferire l'applicazione Python all'host Docker, quindi utilizzare il comando **docker cp** per copiare l'applicazione dall'host al contenitore. Ciò viene dimostrato qui, partendo dal presupposto che l'applicazione Python sia stata trasferita all'host Docker all'indirizzo **/app/python_app.py**.

```
root@2dcbe841742a:~# exit
[root@localhost ~]# docker cp /app/python_app.py nxsdk:/root/
[root@localhost ~]# docker start nxsdk
nxsdk
[root@localhost ~]# docker attach nxsdk
root@2dcbe841742a:/# ls /root/
python_app.py
```

- Copiare il contenuto dell'applicazione Python negli Appunti di sistema, quindi incollare il contenuto in un file creato nel contenitore Docker con l'utilizzo di vim.

A questo punto, utilizzare lo script **rpm_gen.py** presente in **/NX-SDK/scripts/**per creare un pacchetto RPM dall'applicazione Python. Questo script ha un argomento obbligatorio e due

opzioni obbligatorie:

- Il nome file dell'applicazione Python. Ad esempio, un'applicazione Python in un file denominato **python_app.py** produrrebbe un argomento di **python_app.py**. Questo nome file verrà utilizzato successivamente come nome dell'applicazione per NX-SDK e anche da NX-OS per fare riferimento ai comandi creati da questa applicazione.

  Nota: Non è necessario che il nome file contenga estensioni, ad esempio **.py**. In questo esempio, se il nome del file è **python_app** anziché **python_app.py**, il pacchetto RPM verrà generato senza problemi.

- L'opzione -s accetta un argomento per il percorso di file assoluto che porta alla posizione del nome di file sopra indicato. Ad esempio, se **python_app.py** si trova in **/root/**, l'argomento corretto sarà **-s /root/**.
- L'opzione -u indica che il nome del file di origine è uguale al nome del file eseguibile.

Di seguito è illustrato l'utilizzo dello script **rpm_gen.py**.

```
root@7bfd1714dd2f:~# python /NX-SDK/scripts/rpm_gen.py test_python_app -s /root/ -u
################################################################################################
####
Generating rpm package...
<snip>
RPM package has been built
################################################################################################
####

SPEC file: /NX-SDK/rpm/SPECS/test_python_app.spec
RPM file : /NX-SDK/rpm/RPMS/test_python_app-1.0-1.0.0.x86_64.rpm
```

Il percorso del file del pacchetto RPM è indicato nella riga finale dell'output dello script **rpm_gen.py**. Questo file deve essere copiato dal contenitore Docker sull'host in modo che possa essere trasferito al dispositivo Nexus su cui si desidera eseguire l'applicazione. Dopo aver chiuso il contenitore Docker, è possibile eseguire facilmente il comando cp docker <container>:<container_filepath> <host_filepath>, dove <container> è il nome del contenitore Docker NX-SDK (in questo caso, nxsdk), <container_filepath> è il percorso completo del pacchetto RPM all'interno del contenitore (in questo caso, **/NX-SDK/rpm/RPMS/test_python_app-1.0-1.0.0.x86_64.rpm**). **<host_filepath>** è il percorso file completo sull'host Docker in cui deve essere trasferito il pacchetto RPM (in questo caso, **/root/**). Di seguito viene riportata la dimostrazione di questo comando:

```
root@7bfd1714dd2f:/# exit
[root@localhost ~]# docker cp nxsdk:/NX-SDK/rpm/RPMS/test_python_app-1.0-1.0.0.x86_64.rpm /root/
[root@localhost ~]# ls /root/
anaconda-ks.cfg             test_python_app-1.0-1.0.0.x86_64.rpm
```

Trasferire il pacchetto RPM al dispositivo Nexus utilizzando il metodo di trasferimento file preferito. Una volta che il pacchetto RPM si trova sul dispositivo, deve essere installato e attivato in modo simile a un SMU. Questa condizione viene dimostrata nel modo seguente, presupponendo che il pacchetto RPM sia stato trasferito al bootflash del dispositivo.

```
N9K-C93180LC-EX# install add bootflash:test_python_app-1.0-1.0.0.x86_64.rpm
[####################] 100%
Install operation 27 completed successfully at Tue May  8 06:40:13 2018
```

```
N9K-C93180LC-EX# install activate test_python_app-1.0-1.0.0.x86_64
[####################] 100%
Install operation 28 completed successfully at Tue May  8 06:40:20 2018
```

> **Nota:** Quando si installa il pacchetto RPM con il comando **install add**, includere il dispositivo di storage e il nome file esatto del pacchetto. Quando si attiva il pacchetto RPM dopo l'installazione, non includere il dispositivo di storage e il nome del file. Utilizzare il nome del pacchetto. È possibile verificare il nome del pacchetto con il comando **show install inactive**.

Una volta attivato il pacchetto RPM, è possibile avviare l'applicazione con NX-SDK con il comando di configurazione **nxsdk service <application-name>**, dove **<application-name>** è il nome del nome file Python (e, successivamente, dell'applicazione) definito quando lo script rpm_gen.py è stato utilizzato in precedenza. La dimostrazione è la seguente:

```
N9K-C93180LC-EX# conf
Enter configuration commands, one per line. End with CNTL/Z.
N9K-C93180LC-EX(config)# nxsdk service-name test_python_app
% This could take some time. "show nxsdk internal service" to check if your App is Started &
Runnning
```

Èpossibile verificare che l'applicazione sia attiva e avviata con il comando **show nxsdk internal service**:

```
N9K-C93180LC-EX# show nxsdk internal service

NXSDK Started/Temp unavailabe/Max services : 1/0/32
NXSDK Default App Path         : /isan/bin/nxsdk
NXSDK Supported Versions       : 1.0

Service-name              Base App        Started(PID)      Version    RPM Package
------------------------- --------------- ----------------- ---------- -----------------------
test_python_app           nxsdk_app4      VSH(23195)        1.0        test_python_app-1.0-
1.0.0.x86_64
```

Èinoltre possibile verificare che i comandi CLI personalizzati creati da questa applicazione siano accessibili in NX-OS:

```
N9K-C93180LC-EX# show test?
  test_python_app    Nexus Sdk Application
```

Informazioni correlate

- NX-SDK GitHub
- Cisco Nexus serie 9000 NX-OS Programmability Guide, versione 7.x
- Cisco Nexus serie 3000 NX-OS Programmability Guide, versione 7.x
- Cisco Nexus serie 3500 NX-OS Programmability Guide, versione 7.x
- White paper sulla programmabilità e l'automazione della rete con gli switch Cisco Nexus serie 9000
- Programmabilità e automazione con Cisco Open NX-OS (PDF)
- Documentazione e supporto tecnico – Cisco Systems