Développement, débogage et déploiement d'applications Python NX-SDK dans les commutateurs Nexus 3000/9000

Contenu

Introduction

Ce document fournit un workflow pour le développement d'applications Python avec le kit de développement logiciel Cisco NX (SDK) avec l'utilisation de Cisco NX-OS sur les plates-formes Nexus 3000 et Nexus 9000.

Conditions préalables

Conditions requises

Aucune spécification déterminée n'est requise pour ce document.

Components Used

Les informations contenues dans ce document sont basées sur les versions de matériel et de logiciel suivantes :

- Ce document utilise NX-SDK v1.0.0 et NX-SDK v1.5.0
- NX-SDK v1.0.0 peut être utilisé sur la plate-forme Nexus 9000 à partir de la version 7.0(3)I6(1) de NX-OS et de la plate-forme Nexus 3000 à partir de la version 7.0(3)I7(1) de NX-OS
- NX-SDK v1.5.0 peut être utilisé sur la plate-forme Nexus 9000 et la plate-forme Nexus 3000 à partir de la version 7.0(3)I7(3) de NX-OS

The information in this document was created from the devices in a specific lab environment. All of the devices used in this document started with a cleared (default) configuration. If your network is live, make sure that you understand the potential impact of any command.

Informations générales

Cisco NX-SDK permet le développement d'applications personnalisées pouvant s'exécuter nativement dans Cisco NX-OS sur les plates-formes Nexus 9000 et Nexus 3000. NX-SDK permet aux clients de créer leurs propres commandes et sorties CLI, de générer des syslog personnalisés en réponse à des événements spécifiques, de diffuser des données télémétriques personnalisées, etc.

NX-SDK dispose d'une API C++, qui est traduite dans d'autres langues avec l'utilisation de l'outil simplifié Wrapper and Interface Generator (SWIG). Cela permet au client d'utiliser NX-SDK dans n'importe quelle langue de son choix. Ce document présente la mise en oeuvre de fonctions NX-SDK courantes en Python, ainsi qu'un workflow permettant aux clients de développer leurs propres applications Python NX-SDK.

Développer une application Python avec NX-SDK

Activer NX-SDK

Pour que toute application NX-SDK puisse s'exécuter, la fonctionnalité NX-SDK doit d'abord être activée sur le périphérique :

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

Créer un fichier Python

Un fichier Python peut être créé et modifié à l'aide de l'interpréteur de commandes NX-OS Bash. Pour que le shell Bash soit utilisé, il doit d'abord être activé sur le périphérique :

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

Entrez le shell Bash et utilisez l'éditeur de texte vi afin de créer et modifier le fichier Python :

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

Note: La meilleure pratique est de créer des fichiers Python dans le répertoire **/isan/bin/**. Les fichiers Python ont besoin d'autorisations d'exécution pour s'exécuter - ne placez pas les fichiers Python dans le répertoire **/bootflash** ou dans aucun de ses sous-répertoires.

Note: Il n'est pas nécessaire de créer et de modifier des fichiers Python via NX-OS. Le développeur peut créer l'application à l'aide de son environnement local et transférer les fichiers terminés vers le périphérique à l'aide d'un protocole de transfert de fichiers de son choix. Cependant, il peut être plus efficace pour le développeur de déboguer et de dépanner son script à l'aide des utilitaires NX-OS.

Implémenter les composants NX-SDK

Il est recommandé aux développeurs de commencer à créer leur application Python NX-SDK à partir du modèle customCliPyApp sur le Cisco DevNet NX-SDK GitHub. Ces sections de ce document font référence aux composants nécessaires avec l'utilisation de leurs noms dans ce modèle.

Quatre composants principaux sont nécessaires pour une application Python NX-SDK :

1. NX-SDK doit être importé dans l'application via l'instruction **import nx_sdk_py**.
2. Fonction (généralement nommée **sdkThread**) qui démarre l'application NX-SDK et modifie diverses options liées à l'application.
3. La création de commandes CLI personnalisées ainsi que la définition de la syntaxe de commande CLI personnalisée dans la fonction **sdkThread**.
4. Classe nommée **pyCmdHandler** avec une méthode nommée **postCliCb**, qui traite les commandes CLI personnalisées ajoutées par l'application NX-SDK.

sdkThread, fonction

La fonction sdkThread initialise, ajoute des fonctionnalités et démarre l'application NX-SDK. La fonction n'a pas besoin de paramètres pour être passée dedans. Toutes les applications Python NX-SDK nécessitent trois méthodes de la bibliothèque nx_sdk_py à appeler :

1. **nx_sdk_py.NxSdk.getSdkInst(len(sys.argv), sys.argv)** retourne un objet d'instance SDK ou Aucun. Si un objet d'instance SDK est retourné, l'application NX-SDK s'est correctement enregistrée avec l'infrastructure NX-OS. Si Aucun est renvoyé, des erreurs se produisent au moment de ce processus d'enregistrement et une entrée de journal des erreurs apparaît dans le syslog du périphérique. Cette méthode est documentée ici.

   Note: À partir de NX-SDK v1.5.0, un troisième paramètre booléen peut être passé dans la méthode **NxSdk.getSdkInst**, qui active les exceptions avancées quand elles ont la valeur True et désactive les exceptions avancées quand elles ont la valeur False. Cette méthode est documentée ici .

1. méthode **sdk.startEventLoop()**, où **sdk** est l'objet d'instance SDK retourné par la méthode **nx_sdk_py.NxSdk.getSdkInst()**. Cette méthode démarre l'application NX-SDK et lui permet d'interagir avec l'infrastructure NX-OS. Cette méthode est documentée ici .
2. méthode **nx_sdk_py.NxSdk.__swig_Destruc_(sdk)**, où **sdk** est l'objet instance SDK retourné par la **méthode nx_sdk_py.NxSdk.getSdkInst()** expliquée précédemment. Cette méthode placée à la fin de la fonction **sdkThread** permet la sortie gracieuse de l'application NX-SDK.

Voici quelques méthodes couramment utilisées :

- **sdk.getTracer()**, où **sdk** est l'objet d'instance SDK retourné par la méthode **nx_sdk_py.NxSdk.getSdkInst()**. Cette méthode renvoie un objet NxTrace qui peut être utilisé pour générer des syslogs personnalisés, ainsi que des événements et des erreurs de journal dans l'historique des événements de l'application. Les Syslogs personnalisés apparaîtront dans le syslog du périphérique (visible via la commande **show logging logfile**) tandis que les événements consignés dans l'historique des événements de l'application sont visibles via les commandes **show <nom_application> nxsdk event-history events** ou **show <nom_application> nxsdk event-history errors**. Cette méthode est documentée ici. L'objet NxTrace retourné par cette méthode et ses méthodes associées sont documentés ici. Par exemple, vous pouvez afficher l'historique des événements d'une application nommée **Transceiver_DOM.py** via les commandes **show Transceiver_DOM.py nxsdk event-history events** et **show Transceiver_DOM.py nxsdk event-history errors**.
- **sdk.getCliParser()**, où **sdk** est l'objet d'instance SDK retourné par la méthode **nx_sdk_py.NxSdk.getSdkInst()**. Cette méthode retourne un objet NxCliParser, qui peut être utilisé pour exécuter les commandes CLI qui existent déjà via Python, ainsi que pour créer des commandes CLI personnalisées. Cette méthode est documentée ici. L'objet NxCliParser retourné par cette méthode et ses méthodes associées sont documentés ici .
- **cliP.execShowCmd(« cmd », return_type)**, où **cliP** est l'objet NxCliParser retourné par la **méthode sdk.getCliParser()**, **cmd** est la commande show que vous voulez exécuter encapsulée en guillemets, et **return_type est la commande format à afficher.** Le format des données peut être **R_TEXT**, **R_JSON** ou **R_XML**. Cette méthode est documentée ici .

  Note: Les formats de données **R_JSON** et **R_XML** ne fonctionnent que si la commande prend en charge la sortie dans ces formats. Dans NX-OS, vous pouvez vérifier si une commande prend en charge la sortie dans un format de données particulier en envoyant la sortie au format de données demandé. Si la commande piped renvoie une sortie significative, ce format de données est pris en charge. Par exemple, si vous exécutez **show mac address-table dynamic | json** dans NX-OS retourne la sortie JSON, puis le format de données R_JSON est également pris en charge dans NX-SDK.

- **cliP.execConfigCmd(cmd_filename)**, où **cliP** est l'objet NxCliParser retourné par la **méthode sdk.getCliParser()** et **cmd_filename** est le chemin de fichier absolu vers un fichier contenant des commandes séparées par des lignes. Cette méthode retourne une chaîne qui indique le succès de l'exécution de la commande - si « SUCCESS » est dans la chaîne, alors toutes les commandes sont exécutées avec succès. Sinon, la chaîne contient l'exception qui explique pourquoi les commandes n'ont pas pu s'exécuter. Cette méthode est documentée ici.

Voici quelques méthodes facultatives qui peuvent être utiles :

- **sdk.setAppDesc('chaîne de description')**, où **sdk** est l'objet d'instance SDK retourné par la méthode **nx_sdk_py.NxSdk.getSdkInst()**. Cette méthode définit la description de l'application NX-SDK. La description s'affiche dans le menu d'aide contextuel NX-OS accessible via un

point d'interrogation de l'interface de ligne de commande. Cette méthode est documentée ici. Par exemple, une application nommée **Transceiver_DOM.py**, une description d'application de Renvoie toutes les interfaces avec des émetteurs-récepteurs compatibles DOM insérés apparaît dans l'aide contextuelle de NX-OS comme suit :

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

- **sdk.getAppName()**, où **sdk** est l'objet d'instance SDK retourné par la méthode **nx_sdk_py.NxSdk.getSdkinst()**. Cette méthode retourne le nom de l'application Python. Cette méthode est documentée ici.

Créer des commandes CLI personnalisées

Dans une application Python avec l'utilisation de NX-SDK, des commandes CLI personnalisées sont créées et définies dans la fonction sdkThread. Il existe deux types de commandes : **Afficher** les commandes et **configurer** les commandes.

1. **Les** commandes **show** affichent des informations sur le périphérique, sa configuration ou son environnement.
2. **Les** commandes **Config** modifient la configuration du périphérique, ce qui modifie la manière dont le périphérique réagit au réseau environnant.

Ces deux méthodes permettent de créer respectivement des commandes show et config :

- **cliP.newShowCmd(« cmd_name », « syntaxe »)**, où **cliP** est l'objet NxCliParser retourné par la **méthode sdk.getCliParser()**, **cmd_name** est un nom unique pour la commande interne à l'application NX-SDK personnalisée, et **syntaxe** quels mots clés et paramètres peuvent être utilisés dans la commande. Cette méthode retourne un objet NxCliCmd, qui est documenté ici. Cette méthode est documentée ici.

  Note: Cette commande est une sous-classe de cliP.newCliCmd(« cmd_type », « cmd_name », « syntaxe ») où cmd_type est CONF_CMD ou **SHOW_CMD** (selon le type de commande configuré), **cmd_name** est un nom unique pour la commande interne à NSDX-personnalisé. L'application K et la **syntaxe** décrivent les mots clés et les paramètres qui peuvent être utilisés dans la commande. Pour cette raison, la documentation de l'API pour cette commande peut être plus utile pour référence.

- **cliP.newConfigCmd(« cmd_name », « syntaxe »)**, où **cliP** est l'objet NxCliParser retourné par la **méthode sdk.getCliParser()**, **cmd_name** est un nom unique pour la commande interne à l'application NX-SDK personnalisée, et **syntaxe**. décrit les mots clés et les paramètres pouvant être utilisés dans la commande. Cette méthode retourne un objet NxCliCmd, qui est documenté ici . Cette méthode est documentée ici .

  Note: Cette commande est une sous-classe de cliP.newCmd(« cmd_type », « cmd_name », « syntaxe ») où cmd_type est CONF_CMD ou **SHOW_CMD** (cela dépend du type de commande configuré), **cmd_name** est un nom unique pour la commande interne à la personnalisation L'application X-SDK et la **syntaxe** décrivent les mots clés et les paramètres pouvant être utilisés dans la commande. Pour cette raison, la documentation de l'API pour cette commande peut être plus utile pour référence.

Les deux types de commandes ont deux composants différents : Paramètres et mots clés :

1. **Les paramètres** sont des valeurs utilisées pour modifier les résultats de la commande. Par exemple, dans la commande **show ip route 192.168.1.0**, il y a un mot clé **route** suivi d'un paramètre qui accepte une adresse IP, qui spécifie que seules les routes qui incluent l'adresse IP fournie doivent être affichées.

2. **Les mots-clés** modifient les résultats de la commande par leur seule présence. Par exemple, dans la commande **show mac address-table dynamic**, il y a un mot clé **dynamique**, qui spécifie que seules les adresses MAC apprises dynamiquement doivent être affichées.

Les deux composants sont définis dans la syntaxe d'une commande NX-SDK lors de sa création. Des méthodes pour l'objet NxCliCmd existent pour modifier l'implémentation spécifique des deux composants.

- **nx_cmd.updateParam("<paramètre>« , « help_str », type), où nx_cmd est l'objet NxCliCmd retourné par cliP.newShowCmd() ou les méthodes cliP.newConfigCmd(), <paramètre>** est le nom du paramètre de commande qui peut être modifié entre crochets angulaires (<>), **help_str** définit la chaîne d'aide de la commande personnalisée et s'affiche dans le menu d'aide contextuelle NX-OS accessible via un point d'interrogation de l'interface de ligne de commande, et **type** est le type du paramètre. D'autres paramètres facultatifs pour cette méthode sont disponibles et documentés ici. Il s'agit de types valides pour les paramètres qui peuvent être spécifiés dans l'argument de type :

  P_INTEGER - spécifie tout entier**P_STRING** - spécifie toute chaîne**P_INTERFACE** - spécifie toute interface réseau**P_IP_ADDR** - spécifie toute adresse IP**P_MAC_ADDR** - spécifie toute adresse MAC**P_VRF** - spécifie toute instance VRF (Virtual Routing and Forwarding).
- **nx_cmd.updateKeyword(« mot clé », « help_str », is_key), où nx_cmd est l'objet NxCliCmd retourné par cliP.newShowCmd() ou les méthodes cliP.newConfigCmd(), mot clé** est le nom du mot clé de commande que vous **help_str** définit la chaîne d'aide de la commande personnalisée et s'affiche dans le menu d'aide contextuel NX-OS accessible via un point d'interrogation de l'interface de ligne de commande. **is_key** est une valeur booléenne facultative qui prend la valeur False par défaut. Si **is_key** a la valeur True, la configuration unique créée par la commande à l'aide de ce mot clé ne remplace pas une autre configuration unique créée par la commande. Si **is_key** a la valeur False, la configuration créée par la commande à l'aide de ce mot clé remplace l'autre configuration créée par la commande. Cette méthode est documentée ici .

Afin d'afficher des exemples de code de composants de commande couramment utilisés, affichez la section Exemples de commandes de l'interface de ligne de commande personnalisée de ce document.

Une fois les commandes CLI personnalisées créées, un objet de la classe **pyCmdHandler** décrite plus loin dans ce document doit être créé et défini comme objet gestionnaire de rappel CLI pour l'objet NxCliParser. Ceci est démontré comme suit :

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

Ensuite, l'objet NxCliParser doit être ajouté à l'arborescence de l'analyseur CLI NX-OS afin que les commandes CLI personnalisées soient visibles par l'utilisateur. Ceci est fait avec la commande

cliP.addToParseTree(), où **cliP** est l'objet NxCliParser retourné par la méthode **sdk.getCliParser()**.

Exemple de fonction sdkThread

Voici un exemple d'une fonction sdkThread typique avec l'utilisation des fonctions expliquées précédemment. Cette fonction (entre autres dans une application Python NX-SDK personnalisée standard) utilise des variables globales, qui sont instanciées lors de l'exécution de 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)
```

pyCmdHandler, classe

La classe **pyCmdHandler** est héritée de la classe **NxCmdHandler** dans la bibliothèque nx_sdk_py. La méthode **postCliCb(self, clicmd)** définie dans la classe **pyCmdHandler** est appelée chaque fois que des commandes CLI provenant d'une application NX-SDK sont appelées. Ainsi, la méthode

postCliCb(self, clicmd) est l'endroit où vous définissez comment les commandes CLI personnalisées définies dans la fonction sdkThread se comportent sur le périphérique.

La fonction **postCliCb(self, clicmd)** retourne une valeur booléenne. Si **True** est renvoyé, il est présumé que la commande a été exécutée avec succès. **False** doit être retourné si la commande n'a pas été exécutée correctement pour une raison quelconque.

Le paramètre **clicmd** utilise le nom unique défini pour la commande lors de sa création dans la fonction sdkThread. Par exemple, si vous créez une nouvelle commande **show** avec un nom unique **show_xcvr_dom**, il est recommandé de faire référence à cette commande par le même nom dans la **fonction postCliCb(self, clicmd)** après avoir vérifié si le nom de l'argument clicmd contient **show_xcvr_dom.** Il est démontré ici :

```
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()
```

Si une commande qui utilise des paramètres est créée, vous devrez probablement utiliser ces paramètres à un moment donné dans la fonction **postCliCb(self, clicmd)**. Cela peut être fait avec la méthode **clicmd.getParamValue("<paramètre>")**, où **<paramètre>** est le nom du paramètre de commande que vous souhaitez placer entre crochets angulaires (**<>**). Cette méthode est documentée ici. Cependant, la valeur retournée par cette fonction doit être convertie en type dont vous avez besoin. Pour ce faire, procédez comme suit :

- **nx_sdk_py.void_to_int** convertit une valeur en un type entier.
- **nx_sdk_py.void_to_string** convertit une valeur en type chaîne.

La fonction **postCliCb(self, clicmd)** (ou toute fonction ultérieure) sera généralement utilisée pour imprimer la sortie de la commande show sur la console. Ceci est fait avec la méthode **clicmd.printConsole()**.

> **Note**: Si l'application rencontre une erreur, une exception non gérée, ou se retire soudainement, alors la sortie de la **fonction clicmd.printConsole()** ne sera pas affichée du tout. Pour cette raison, la meilleure pratique lorsque vous déboguez votre application Python est soit de consigner les messages de débogage dans le syslog avec l'utilisation d'un objet NxTrace retourné par la méthode **sdk.getTracer()**, soit d'utiliser des instructions d'impression et d'exécuter l'application via le binaire du shell Bash **/isan/bin/bin/python**.

pyCmdHandler, classe, exemple

Le code suivant sert d'exemple pour la classe **pyCmdHandler** décrite ci-dessus. Ce code provient du fichier ip_move.py de l'application ip-mouvement NX-SDK disponible ici. L'objectif de cette application est de suivre le déplacement d'une adresse IP définie par l'utilisateur sur les interfaces d'un périphérique Nexus. Pour ce faire, le code recherche l'adresse MAC de l'entrée d'adresse IP via le paramètre **<ip>** dans le cache ARP du périphérique, puis vérifie le VLAN dans lequel réside

cette adresse MAC à l'aide de la table d'adresses MAC du périphérique. À l'aide de cette adresse MAC et de ce VLAN, la commande **show system internal l2fm l2dbg macdb address <mac> vlan <vlan>** affiche la liste des index d'interface SNMP auxquels cette combinaison a récemment été associée. Le code utilise ensuite la commande **show interface snmp-ifindex** pour traduire les index d'interface SNMP récents en noms d'interface lisibles par l'homme.

```
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
```

Exemples de syntaxe de commande CLI personnalisée

Cette section présente quelques exemples du paramètre de syntaxe utilisé lorsque vous créez des commandes CLI personnalisées avec les méthodes **cliP.newShowCmd()** ou **cliP.newConfigCmd()**, où **cliP** est l'objet NxCliParser retourné par la **méthode sdk.getCliParser()**.

> **Note**: La prise en charge de la syntaxe avec parenthèses ouvrantes et fermantes ("(" et ")") est introduite dans NX-SDK v1.5.0, incluse dans NX-OS version 7.0(3)I7(3). Il est supposé que l'utilisateur utilise NX-SDK v1.5.0 lorsqu'il suit l'un de ces exemples donnés qui incluent la syntaxe utilisant des parenthèses ouvrantes et fermantes.

Mot clé unique

Cette commande show prend un seul mot clé mac et ajoute une chaîne d'aide de Shows all bad programmed MAC address on this device au mot clé.

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

Paramètre unique

Cette commande show prend un seul paramètre **<mac>**. Les crochets d'angle autour du mot mac signifient qu'il s'agit d'un paramètre. Une chaîne d'aide d'adresse MAC pour vérifier la mauvaise programmation est ajoutée au paramètre. Le paramètre **nx_sdk_py.P_MAC_ADDR.** dans la **méthode nx_cmd.updateParam()** sert à définir le type du paramètre en tant qu'adresse MAC, ce qui empêche l'entrée de l'utilisateur final d'un autre type, comme une chaîne, un entier ou une adresse 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)
```

Mot-clé facultatif

Cette commande show peut éventuellement prendre un seul mot clé **[mac]**. Les crochets entourant le mot mac signifient que ce mot clé est facultatif. Une chaîne d'aide Affiche toutes les adresses MAC mal programmées sur ce périphérique est ajoutée au mot clé.

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

Paramètre facultatif

Cette commande show peut éventuellement prendre un seul paramètre **[<mac>]**. Les crochets entourant le mot < mac > signifient que ce paramètre est facultatif. Les crochets d'angle autour du mot mac signifient qu'il s'agit d'un paramètre. Une chaîne d'aide d'adresse MAC pour vérifier la mauvaise programmation est ajoutée au paramètre. Le paramètre **nx_sdk_py.P_MAC_ADDR.** dans la **méthode nx_cmd.updateParam()** sert à définir le type du paramètre en tant qu'adresse MAC, ce qui empêche l'entrée de l'utilisateur final d'un autre type, comme une chaîne, un entier ou

une adresse 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)
```

Mot clé et paramètre uniques

Cette commande show prend un seul mot clé mac immédiatement suivi du paramètre **<mac-address>**. Les crochets d'angle autour du mot mac-address signifient qu'il s'agit d'un paramètre. Une chaîne d'aide de Check MAC address pour détecter une mauvaise programmation est ajoutée au mot clé. Une chaîne d'aide d'adresse MAC pour vérifier la mauvaise programmation est ajoutée au paramètre. Le paramètre **nx_sdk_py.P_MAC_ADDR.** dans la **méthode nx_cmd.updateParam()** est utilisé afin de définir le type du paramètre en tant qu'adresse MAC, ce qui empêche l'entrée de l'utilisateur final d'un autre type, comme une chaîne, un entier ou une adresse 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)
```

Mots clés et paramètres multiples

Cette commande show peut prendre l'un des deux mots clés, qui ont deux paramètres différents après eux. Le premier mot clé mac a un paramètre **<mac-address>**, et le second mot clé ip a un paramètre **<ip-address>**. Les crochets d'angle autour des mots mac-address et ip-address signifient qu'il s'agit de paramètres. Une chaîne d'aide de Check MAC address pour détecter une mauvaise programmation est ajoutée au mot clé mac. Une chaîne d'aide d'adresse MAC pour vérifier la mauvaise programmation est ajoutée au paramètre **<mac-address>**. Le paramètre **nx_sdk_py.P_MAC_ADDR** de la **méthode nx_cmd.updateParam()** est utilisé pour définir le type du **<mac-address>** comme adresse MAC, ce qui empêche l'utilisateur final de saisir un autre type, comme une chaîne, un entier ou une adresse IP. Une chaîne d'aide de Check IP address pour détecter une mauvaise programmation est ajoutée au mot clé ip. Une chaîne d'aide d'adresse IP permettant de vérifier la mauvaise programmation est ajoutée au paramètre **<ip-address>**. Le paramètre **nx_sdk_py.P_IP_ADDR** de la **méthode nx_cmd.updateParam()** est utilisé pour définir le type du **<adresse-ip>** comme adresse IP, ce qui empêche l'entrée de l'utilisateur final d'un autre type, comme une chaîne, un entier ou une adresse 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)
```

Mots clés et paramètres multiples avec mot clé facultatif

Cette commande show peut prendre l'un des deux mots clés, qui ont deux paramètres différents après eux. Le premier mot clé mac a un paramètre **<mac-address>**, et le second mot clé ip a un paramètre **<ip-address>**. Les crochets d'angle autour des mots mac-address et ip-address signifient qu'il s'agit de paramètres. Une chaîne d'aide de Check MAC address pour détecter une mauvaise programmation est ajoutée au mot clé mac. Une chaîne d'aide d'adresse MAC pour

vérifier la mauvaise programmation est ajoutée au paramètre **<mac-address>**. Le paramètre **nx_sdk_py.P_MAC_ADDR** de la **méthode nx_cmd.updateParam()** est utilisé pour définir le type du **<mac-address>** comme adresse MAC, ce qui empêche l'utilisateur final de saisir un autre type, comme une chaîne, un entier ou une adresse IP. Une chaîne d'aide de Check IP address pour détecter une mauvaise programmation est ajoutée au mot clé ip. Une chaîne d'aide d'adresse IP permettant de vérifier la mauvaise programmation est ajoutée au paramètre **<ip-address>**. Le paramètre **nx_sdk_py.P_IP_ADDR** de la **méthode nx_cmd.updateParam()** est utilisé pour définir le type du **<adresse-ip>** comme adresse IP, ce qui empêche l'entrée de l'utilisateur final d'un autre type, comme une chaîne, un entier ou une adresse IP. Cette commande show peut éventuellement prendre un mot clé [clear]. Une chaîne d'aide Efface les adresses détectées comme étant mal programmées est ajoutée à ce mot clé facultatif.

```
 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")
```

Mots clés et paramètres multiples avec paramètre facultatif

Cette commande show peut prendre l'un des deux mots clés, qui ont deux paramètres différents après eux. Le premier mot clé mac a un paramètre **<mac-address>**, et le second mot clé ip a un paramètre **<ip-address>**. Les crochets d'angle autour des mots mac-address et ip-address signifient qu'il s'agit de paramètres. Une chaîne d'aide de Check MAC address for misprogrammingis est ajoutée au mot clé mac. Une chaîne d'aide d'adresse MAC pour vérifier la mauvaise programmation est ajoutée au paramètre **<mac-address>**. Le paramètre **nx_sdk_py.P_MAC_ADDR** de la **méthode nx_cmd.updateParam()** est utilisé pour définir le type du **<mac-address>** comme adresse MAC, ce qui empêche l'utilisateur final de saisir un autre type, comme une chaîne, un entier ou une adresse IP. Une chaîne d'aide de Check IP address pour détecter une mauvaise programmation est ajoutée au mot clé ip. Une chaîne d'aide d'adresse IP permettant de vérifier la mauvaise programmation est ajoutée au paramètre **<ip-address>**. Le paramètre **nx_sdk_py.P_IP_ADDR** de la **méthode nx_cmd.updateParam()** est utilisé pour définir le type du **<adresse-ip>** comme adresse IP, ce qui empêche l'entrée de l'utilisateur final d'un autre type, comme une chaîne, un entier ou une adresse IP. Cette commande show peut éventuellement prendre un paramètre **[<module>]**. Chaîne d'aide Seules les adresses claires sur le module spécifié sont ajoutées à ce paramètre facultatif.

```
 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)
```

Déboguer une application Python avec NX-SDK

Une fois qu'une application Python NX-SDK a été créée, elle devra souvent être déboguée. NX-SDK vous informe en cas d'erreur de syntaxe dans votre code, mais parce que la bibliothèque Python NX-SDK utilise SWIG pour traduire les bibliothèques C++ en bibliothèques Python, toutes les exceptions rencontrées au moment de l'exécution du code entraînent un vidage du noyau d'application similaire à ceci :

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

En raison de la nature ambiguë de ce message d'erreur, la meilleure pratique pour déboguer les applications Python est de consigner les messages de débogage dans le syslog avec l'utilisation d'un objet NxTrace retourné par la méthode **sdk.getTracer()**. Ceci est démontré comme suit :

```
#! /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")
```

Si la journalisation des messages de débogage dans le syslog n'est pas une option, une autre méthode consiste à utiliser des instructions d'impression et à exécuter l'application via le binaire **/isan/bin/python du** shell Bash. Cependant, la sortie de ces instructions d'impression ne sera visible que si elle est exécutée de cette manière. L'exécution de l'application via le shell VSH ne génère aucune sortie. Voici un exemple d'utilisation des instructions d'impression :

```
#! /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")
```

Déployer une application Python avec NX-SDK

Une fois qu'une application Python a été entièrement testée dans le shell Bash et est prête pour le déploiement, l'application doit être installée en production via VSH. Cela permet à l'application de persister lorsque le périphérique se recharge ou lorsque la commutation du système se produit

dans un scénario de double superviseur. Pour déployer une application via VSH, vous devez créer un package RPM à l'aide d'un environnement de build NX-SDK et ENXOS SDK. Cisco DevNet fournit une image Docker qui facilite la création de packages RPM.

> **Note**: Pour obtenir de l'aide pour installer Docker sur votre système d'exploitation spécifique, reportez-vous à la documentation d'installation de Docker.

Sur un hôte compatible Docker, tirez la version d'image de votre choix à l'aide de la commande **docker pull dockercisco/nxsdk:<tag>**, où **<tag>** est la balise de la version d'image de votre choix. Vous pouvez voir les versions d'image disponibles et leurs balises correspondantes ici. Ceci est démontré avec la balise **v1** ici :

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

Démarrez un conteneur nommé **nxsdk** à partir de cette image et joignez-le. Si la balise de votre choix est différente, remplacez **v1** par votre balise :

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

Mettre à jour la dernière version de NX-SDK et accéder au répertoire **NX-SDK**, puis extraire les derniers fichiers de git :

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

Si vous avez besoin d'utiliser une version antérieure de NX-SDK, vous pouvez cloner la branche NX-SDK avec l'utilisation de la balise de version respective avec la commande **git clone -b v<version>** https://github.com/CiscoDevNet/NX-SDK.git, où **<version>** est la version de NX-SDK dont vous avez besoin. Ceci est démontré ici avec NX-SDK v1.0.0 :

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

Ensuite, transférez votre application Python dans le conteneur Docker. Il y a quelques façons différentes de le faire.

- Quittez le conteneur Docker (qui arrête le conteneur et vous demande de le redémarrer), transférez l'application Python vers l'hôte Docker, puis utilisez la commande **docker cp** afin de copier l'application de l'hôte vers le conteneur. Ceci est démontré ici, en supposant que l'application Python a été transférée à l'hôte Docker à l'adresse **/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
```

- Copiez le contenu de l'application Python dans votre presse-papiers système, puis collez le contenu dans un fichier créé dans le conteneur Docker avec l'utilisation de vim.

Ensuite, utilisez le script **rpm_gen.py** situé dans **/NX-SDK/scripts/** afin de créer un paquet RPM à

partir de l'application Python. Ce script comporte un argument obligatoire et deux commutateurs requis :

- Nom de fichier de l'application Python. Par exemple, une application Python dans un fichier nommé **python_app.py** résulterait en un argument de **python_app.py**. Ce nom de fichier sera ensuite utilisé comme nom d'application pour NX-SDK, et sera également utilisé par NX-OS afin de faire référence aux commandes créées par cette application.

  Note: Le nom du fichier ne doit contenir aucune extension de fichier, telle que **.py**. Dans cet exemple, si le nom du fichier était **python_app** au lieu de **python_app.py**, le package RPM serait généré sans problème.

- Le commutateur -s prend un argument pour le chemin de fichier absolu qui conduit à l'emplacement du nom de fichier mentionné ci-dessus. Par exemple, si **python_app.py** se trouve dans **/root/**alors l'argument correct sera **-s /root/**.
- Le commutateur -u indique que le nom de fichier source est identique au nom de fichier exécutable.

L'utilisation du script **rpm_gen.py** est présentée ici.

```
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
```

Le chemin d'accès au paquet RPM est indiqué dans la ligne finale du script **rpm_gen.py**. Ce fichier doit être copié hors du conteneur Docker sur l'hôte afin qu'il puisse être transféré au périphérique Nexus sur lequel vous souhaitez exécuter l'application. Une fois que vous avez quitté le conteneur Docker, vous pouvez le faire facilement avec la commande docker cp <container>:<container_filepath> <host_filepath>, où <container> est le nom du conteneur Docker NX-SDK (dans ce cas, nxsdk), <container_filepath> est le chemin de fichier complet du package RPM à l'intérieur du conteneur (dans ce cas, **/NX-SDK/rpm/RPMS/test_python_app-1.0-1.0.0.x86_64.rpm**) et **<host_filepath>** est le chemin de fichier complet sur notre hôte Docker où le paquet RPM doit être transféré (dans ce cas, **/root/**). Cette commande est présentée ici :

```
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
```

Transférez ce package RPM vers le périphérique Nexus en utilisant la méthode de transfert de fichiers que vous préférez. Une fois que le package RPM est sur le périphérique, il doit être installé et activé de la même manière qu'une SMU. Ceci est démontré comme suit, en supposant que le paquet RPM a été transféré au bootflash du périphérique.

```
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
```

> **Note**: Lorsque vous installez le package RPM à l'aide de la commande **install add**, incluez le périphérique de stockage et le nom de fichier exact du package. Lorsque vous activez le package RPM après l'installation, n'incluez pas le périphérique de stockage et le nom de fichier - utilisez le nom du package lui-même. Vous pouvez vérifier le nom du package à l'aide de la commande **show install inactive**.

Une fois le package RPM activé, vous pouvez démarrer l'application avec NX-SDK avec la commande de configuration **nxsdk service <nom-application>**, où **<nom-application>** est le nom du fichier Python (et, par la suite, de l'application) qui a été défini lorsque le script rpm_gen.py a été utilisé précédemment. Ceci est démontré comme suit :

```
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
```

Vous pouvez vérifier que l'application est active et a commencé à s'exécuter à l'aide de la commande **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
```

Vous pouvez également vérifier que les commandes CLI personnalisées créées par cette application sont accessibles dans NX-OS :

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

Informations connexes

- Concentrateur GitHub NX-SDK
- Guide de programmabilité NX-OS de la gamme Cisco Nexus 9000, version 7.x
- Guide de programmabilité NX-OS de la gamme Cisco Nexus 3000, version 7.x
- Guide de programmabilité NX-OS de la gamme Cisco Nexus 3500, version 7.x
- Livre blanc sur la programmabilité et l'automatisation des réseaux avec les commutateurs Cisco Nexus 9000
- Programmabilité et automatisation avec Cisco Open NX-OS (PDF)
- Support et documentation techniques - Cisco Systems