Internet Week 2014 【S4】
ようこそ、ネットワーク運用自動化の世界へ！

1) ネットワーク機器のソフトウェア制御

湯澤民浩（ゆざわ　たみひろ）

2007年　さくらインターネット株式会社入社
　　技術部ネットワークチーム配属
2012年　基盤戦略部バックボーンチーム配属

- データセンタ事業会社でサービス用ネットワークバックボーンの設計、運用、監視業務に携わって8年目.
- 前職では業務系ネットサービス事業会社に6年間在籍し、開発や運用業務を担当.

機器単体を管理、制御する処理の自動化は、構成管理を自動化する仕組みの最小単位のひとつです.

代表的な機器へのアクセス手法を比較しながら、いくつかの手法についてPythonを使った実装の一例を紹介します.

- 機器の構成を管理・制御する方法あれこれ
- CLIとAPI
- さくらインターネットのL2スイッチ設定自動化の事例紹介
- 制御手法の実装例
- ソフトウェアでの制御を実践するにあたっての注意点
- ここまでのまとめ

機器の構成を管理・制御する方法あれこれ

リモートから機器にアクセスして操作する手法

メッセージフォーマット	標準化	トランザクション	プロトコル	インターフェイスとして使用するPythonモジュール
CLIコマンド	RFC	no *(1)	Telnet,ssh	pexpect
構造化テキスト (MIB,YANG, ..) (BER,XML,JSON,..)	RFC	no	SNMP	pysnmp
構造化テキスト (MIB,YANG, ..) (BER,XML,JSON,..)	RFC	yes *(2)	NETCONF	ncclient ↔ py-junos-eznc *(3)
構造化テキスト (MIB,YANG, ..) (BER,XML,JSON,..)	ベンダー提供API	no	Arista eAPI	json, urllib2 （標準ライブラリ）
構造化テキスト (MIB,YANG, ..) (BER,XML,JSON,..)	ベンダー提供API	yes	Cisco NX-API	
構造化テキスト (MIB,YANG, ..) (BER,XML,JSON,..)	ベンダー提供API	yes	Viyatta REST API	
			….	

(1) 機器(OS)ごとの実装に依存　(2) オプショナル　(3) JUNOS専用API(後述)

機器上で直接制御命令を実行できるものもある

ベンダーが独自に提供する機能の一例

- Cisco Embedded Event Manager (Tcl)
- 装置上で直接つかえるPython実行環境
  Cisco Nexus Python API、 Alaxala 運用支援機能
  - CLIのプロンプトからPythonインタープリタを実行できる
  - 機器のオペレーションを制御するためのライブラリが用意されている

今後も機器のAPI実装がトレンドになる(と期待)

近年市場に投入されている、モダンなosを搭載する製品では、マネジメントプレーンでのデータモデル化への対応が整備されてきている(はず)

- 構成情報が構造化されたオブジェクトとして扱われる.
- コントロールプレーンから取得するステータスや学習情報も構造化されたオブジェクトとして扱われる.

CLIとAPI

CLIは人間と計算機のインターフェイス

```
admin@R2> show route terse

inet.0: 15 destinations, 15 routes (15 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

A Destination        P Prf   Metric 1   Metric 2  Next hop         AS path
* 0.0.0.0/0          S   5                        >192.168.11.1
* 10.1.2.0/30        D   0                        >ge-0/0/2.14
* 10.1.2.1/32        L   0                         Local
* 10.1.4.0/30        O  10        101             >10.1.2.2
* 10.1.6.1/32        L   0                         Reject
* 10.1.8.0/30        D   0                        >ge-0/0/4.0
* 10.1.8.1/32        L   0                         Local
* 10.1.9.0/30        O  10      10001             >10.1.2.2
* 10.1.10.0/30       O  10       1001             >10.1.2.2
* 10.1.255.3/32      O  10         11             >10.1.2.2
* 10.1.255.6/32      D   0                        >lo0.0
* 192.168.11.0/24    D   0                        >me0.0
* 192.168.11.101/32  L   0                         Local
* 224.0.0.5/32       O  10          1              MultiRecv
* 224.0.0.22/32      I   0                         MultiRecv
```

情報にアクセスする人間にとって扱いやすいのができの良いCLI

APIはプログラムどうしのインターフェイス

```
<route-table><table-name>inet.0</table-name><destination-count>15</destination-count><total-route-count>15</total-
route-count><active-route-count>15</active-route-count><holddown-route-count>0</holddown-route-count><hidden-
route-count>0</hidden-route-count><rt junos:style="terse"><rt-destination>0.0.0.0/0</rt-destination><rt-
entry><active-tag>*</active-tag><current-active/><last-active/><protocol-name>Static</protocol-
name><preference>5</preference><nh><selected-next-hop/><to>192.168.11.1</to></nh></rt-entry></rt><rt
junos:style="terse"><rt-destination>10.1.2.0/30</rt-destination><rt-entry><active-tag>*</active-tag><current-
active/><last-active/><protocol-name>Direct</protocol-name><preference>0</preference><nh><selected-next-hop/
><via>ge-0/0/2.14</via></nh></rt-entry></rt><rt junos:style="terse"><rt-destination>10.1.2.1/32</rt-
destination><rt-entry><active-tag>*</active-tag><current-active/><last-active/><protocol-name>Local</protocol-
name><preference>0</preference><nh-type>Local</nh-type></rt-entry></rt><rt junos:style="terse"><rt-
destination>10.1.4.0/30</rt-destination><rt-entry><active-tag>*</active-tag><current-active/><last-active/
><protocol-name>OSPF</protocol-name><preference>10</preference><metric>101</metric><nh><selected-next-hop/
><to>10.1.2.2</to></nh></rt-entry></rt><rt junos:style="terse"><rt-destination>10.1.6.1/32</rt-destination><rt-
entry><active-tag>*</active-tag><current-active/><last-active/><protocol-name>Local</protocol-name><preference>0</
preference><nh-type>Reject</nh-type></rt-entry></rt><rt junos:style="terse"><rt-destination>10.1.8.0/30</rt-
destination><rt-entry><active-tag>*</active-tag><current-active/><last-active/><protocol-name>OSPF</protocol-
name><preference>10</preference><metric>1001</metric><nh><selected-next-hop/><to>10.1.2.2</to></nh></rt-entry></
rt><rt junos:style="terse"><rt-destination>10.1.9.0/30</rt-destination><rt-entry><active-tag>*</active-
tag><current-active/><last-active/><protocol-name>OSPF</protocol-name><preference>10</preference><metric>10001</
metric><nh><selected-next-hop/><to>10.1.2.2</to></nh></rt-entry></rt><rt junos:style="terse"><rt-
destination>10.1.10.0/30</rt-destination><rt-entry><active-tag>*</active-tag><current-active/><last-active/
><protocol-name>OSPF</protocol-name><preference>10</preference><metric>1001</metric><nh><selected-next-hop/
><to>10.1.2.2</to></nh></rt-entry></rt><rt junos:style="terse"><rt-destination>10.1.255.3/32</rt-destination><rt-
entry><active-tag>*</active-tag><current-active/><last-active/><protocol-name>OSPF</protocol-name><preference>10</
preference><metric>11</metric><nh><selected-next-hop/><to>10.1.2.2</to></nh></rt-entry></rt>...
```

情報にアクセスする人間にとっての扱いやすさは必要ではない
（プログラムをつくる人間にとっての扱いやすさ、とは別）

APIがない機器をプログラムで制御するには...

必要な情報を取得するための処理を追加する

(Expect hell!)

```
telnet@R1#show access-list name SNMP-ACCESS

Standard IP access list SNMP-ACCESS : 4 entries
 sequence 10 permit 172.25.8.0 0.0.0.255
 sequence 20 permit 172.31.30.0 0.0.0.255
 sequence 30 permit 192.168.11.0 0.0.0.255
 sequence 40 permit host 10.0.0.1
```

```
>>> acl = list()
>>> cmd = "show access-list name SNMP-ACCESS | inc ^_+sequence"
>>> child.sendline(cmd)
>>> child.expect(config_mode and config_prompt or priv_prompt)
>>> for l in child.before.split(linebreak):
...   if l.strip() == cmd.strip() or len(l) == 0: continue
...   m = re.match(r"^\s*sequence\s+\d+\s+permit\s+(?:host\s+)?([\d.]+)(?:\s+([\d.]+))?
\s*$", l)
...   if m:
...     if m.group(2):
...       acl.append(IPv4Network("%s/%s" % m.groups()))
...     else:
...       acl.append(IPv4Network("%s" % m.group(1)))
...
>>> acl
[IPv4Network('172.25.8.0/24'), IPv4Network('172.31.30.0/24'),
IPv4Network('192.168.11.0/24'), IPv4Network('10.0.0.1/32')]
```

APIが提供されている場合は...
(機器のAPIを制御するクライアントライブラリを利用できる場合)

```
>>> PrefListTable(dev).get()
PrefListTable:192.168.11.101: 2 items
>>>
>>> json.loads(PrefListTable(dev).get().to_json())['SNMP-ACCESS']
{u'name': u'SNMP-ACCESS', u'entries': {u'172.31.30.0/24': {u'prefix':
u'172.31.30.0/24'}, u'10.0.0.1/32': {u'prefix': u'10.0.0.1/32'}, u'192.168.11.0/24':
{u'prefix': u'192.168.11.0/24'}, u'172.25.8.0/24': {u'prefix': u'172.25.8.0/24'}}}
>>>
```

必要な情報にアクセスしてから
型変換してるだけ

```
>>>
>>> acl = list()
>>> if [ p for p in PrefListTable(dev).get() if p.name == "SNMP-ACCESS" ]:
...   acl = map(IPv4Network, p.entries.keys())
...
>>> acl
[IPv4Network('10.0.0.1/32'), IPv4Network('172.25.8.0/24'),
IPv4Network('172.31.30.0/24'), IPv4Network('192.168.11.0/24')]
```

プログラム内部ではオブジェクトとして処理する

```
>>> p = IPv4Network('210.224.160.0/19')
>>> type(p)
<class 'ipaddr.IPv4Network'>
>>> str(p)
'210.224.160.0/19'
>>> int(p)
3537936384
>>> p.numhosts
8192
>>> p.broadcast
IPv4Address('210.224.191.255')
>>> p.netmask
IPv4Address('255.255.224.0')
>>> p.hostmask
IPv4Address('0.0.31.255')
>>> p.subnet(2)
[IPv4Network('210.224.160.0/21'), IPv4Network('210.224.168.0/21'),
IPv4Network('210.224.176.0/21'), IPv4Network('210.224.184.0/21')]

>>> IPv4Address('210.224.191.1') in p
True
>>> IPv4Address('210.224.191.1') + 255 in p
False
```

出力の時点で適切な型に変換する
プログラム側でAPIを用意すると、他のプログラムからアクセスが可能になる

APIが出力する構造化データはリクエストされた情報の中身だけではない

受け取るリクエストそのものの情報や、実行ステータスについての情報も構造化されたデータとして出力する.

```
S2#get ip access-lists SNMP-ACCESS
% Invalid input
S2#
```

```
>>> cmd = {'jsonrpc': '2.0', 'method': 'runCmds',
...        'params': {'format': 'json', 'version': 1,
...                   'cmds': ['enable', 'get ip access-lists SNMP-ACCESS', ]},
...        'id': 1, }
>>> res = urllib2.urlopen( urllib2.Request(url, json.dumps(cmd),
...                         {'content-type': 'application/json', } ))
>>>
>>> json.loads(res.readline()).get('error')
{u'message': u"CLI command 2 of 2 'get ip access-lists SNMP-ACCESS' failed: invalid
command", u'code': 1002, u'data': [{}, {u'errors': [u"Invalid input (at token 0:
'get')"]}]}
>>>
```

さくらインターネットのL2スイッチ設定自動化の事例紹介

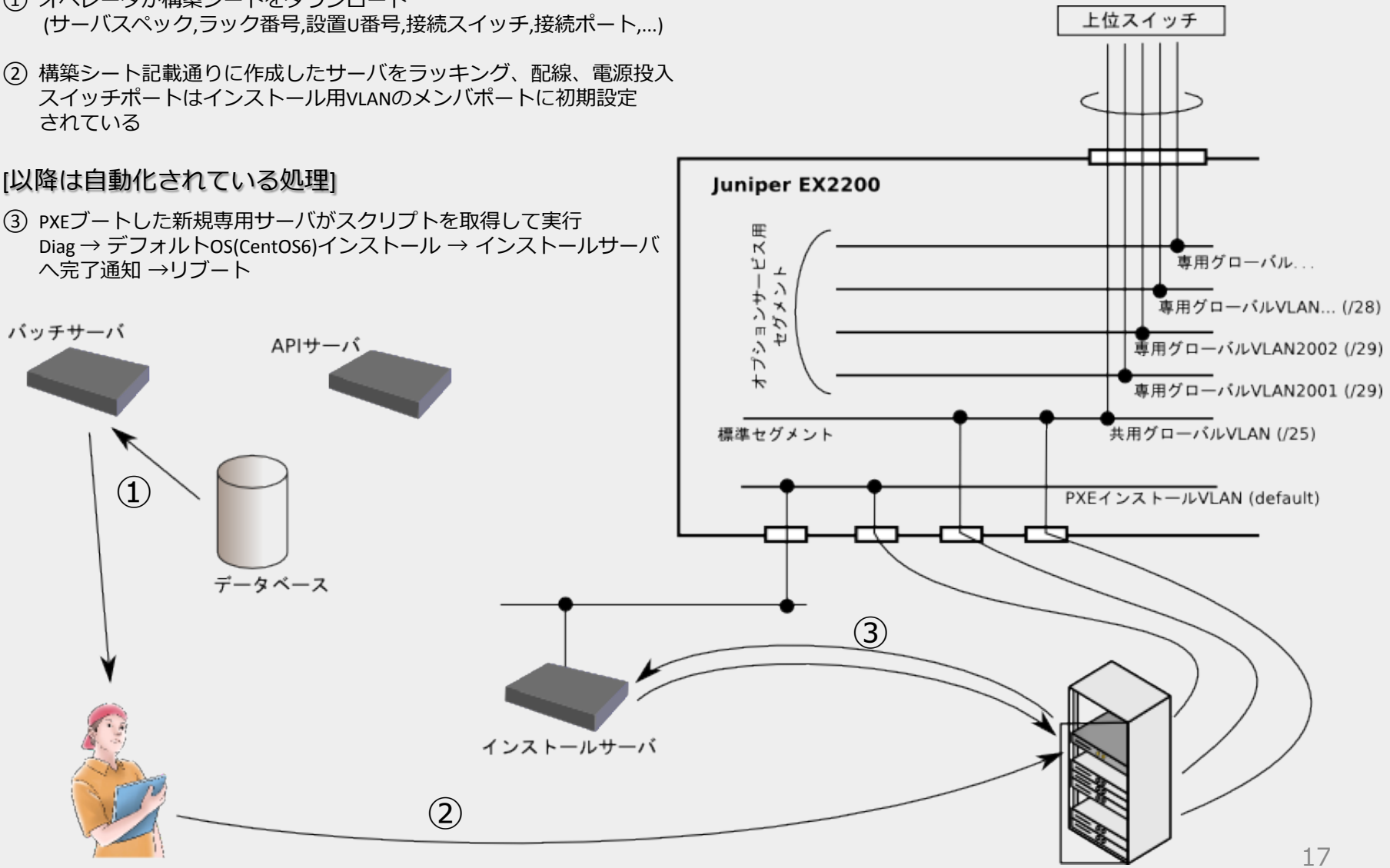
さくらの専用サーバを構築、出荷するまでのフロー

① オペレータが構築シートをダウンロード
(サーバスペック,ラック番号,設置U番号,接続スイッチ,接続ポート,...)

② 構築シート記載通りに作成したサーバをラッキング、配線、電源投入
スイッチポートはインストール用VLANのメンバポートに初期設定
されている

[以降は自動化されている処理]

③ PXEブートした新規専用サーバがスクリプトを取得して実行
Diag → デフォルトOS(CentOS6)インストール → インストールサーバ
へ完了通知 →リブート



さくらの専用サーバを構築、出荷するまでのフロー

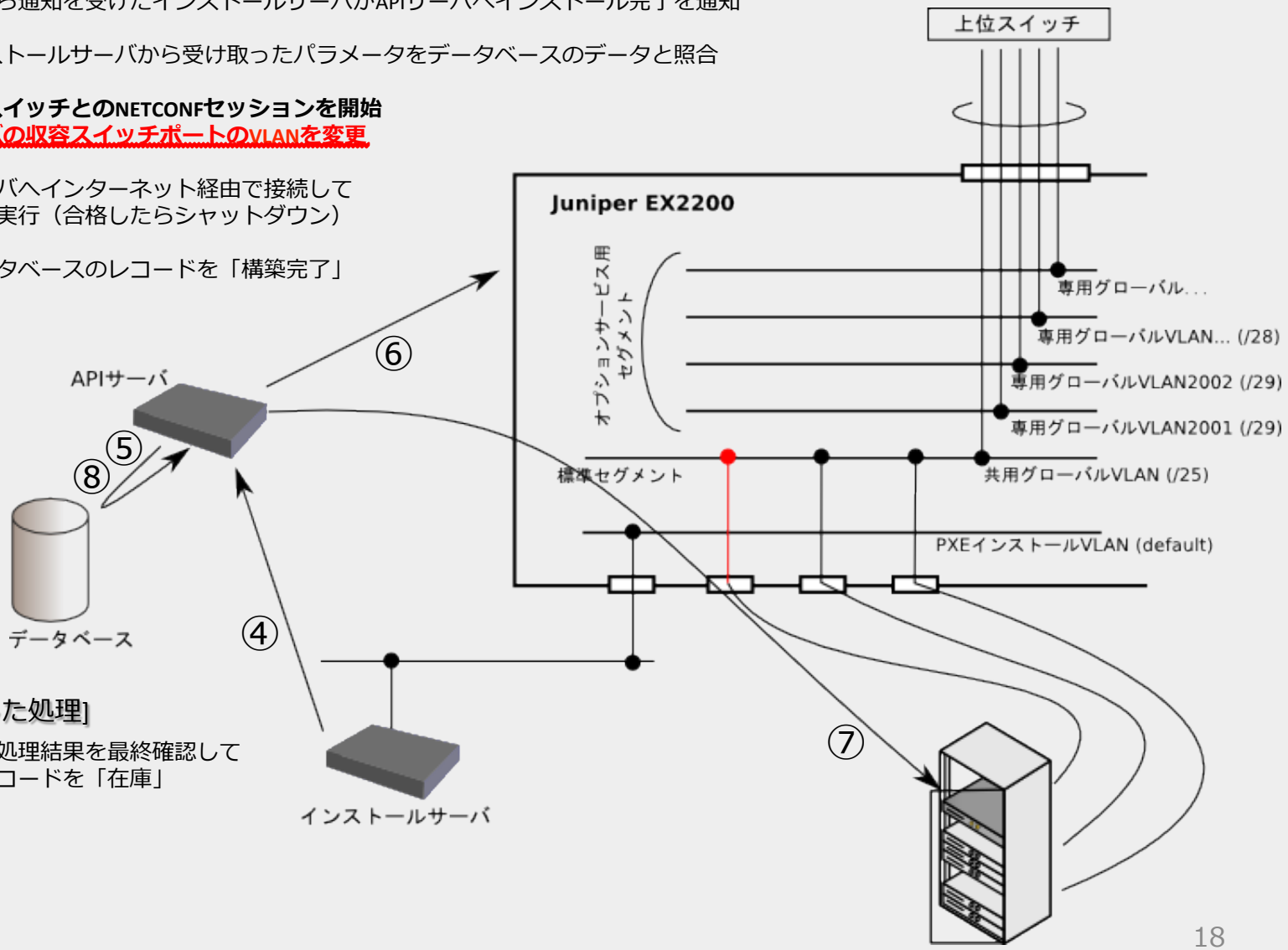
④ 新規専用サーバから通知を受けたインストールサーバがAPIサーバへインストール完了を通知

⑤ APIサーバはインストールサーバから受け取ったパラメータをデータベースのデータと照合

⑥ APIサーバは**収容スイッチとのNETCONFセッションを開始**
→ **新規専用サーバの収容スイッチポートのVLANを変更**

⑦ リブートしたサーバへインターネット経由で接続して
チェックアップを実行（合格したらシャットダウン）

⑧ 正常確認後、データベースのレコードを「構築完了」
ステータスに更新



[最後に人手を介した処理]

⑨ オペレータが自動処理結果を最終確認して
データベースのレコードを「在庫」
ステータスに更新

専用グローバルセグメントへの切り替え処理の自動化

① お客様が会員メニューからオプション申し込み、入金手続き

② 5分ごとの定期自動処理で、プールされたセグメントから決済確認できたお客様へ割り当て
お客様はコントロールパネルから切り替え可能な状態になる

③ お客様がコントロールパネルから切り替え

④ コントロールパネルUIサーバからAPIサーバへ通知

⑤ APIサーバはデータベースと照合してから
収容スイッチとのNETCONFセッションを開始
→ **該当サーバの収容スイッチポートのVLANを変更**

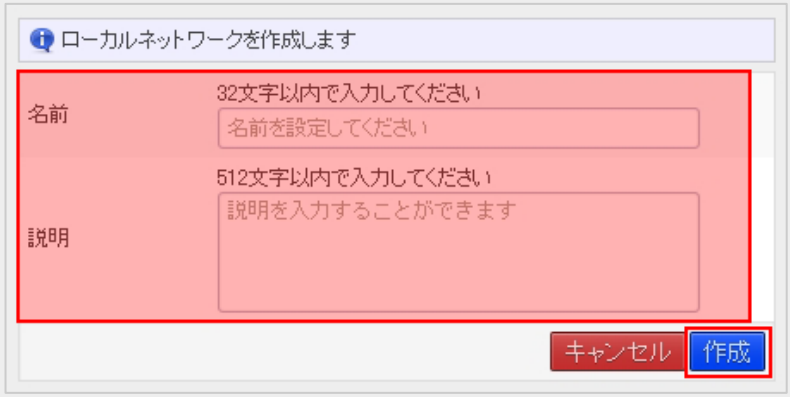
⑥ データベースのレコードを更新

※お客様はIPMIコンソールからネットワーク設定を更新

さくらの専用サーバローカル接続開通処理の自動化

① 在庫ステータスのサーバのLAN側インターフェイスを収容するスイッチポートはすべてディセーブル設定.
お客様からみると、利用開始時点ではリンクダウン状態.

② お客様が専用サーバコントロールパネルから「スイッチ」を作成.
ひとつの会員IDから基本料金内で最大３スイッチまで作成可能.



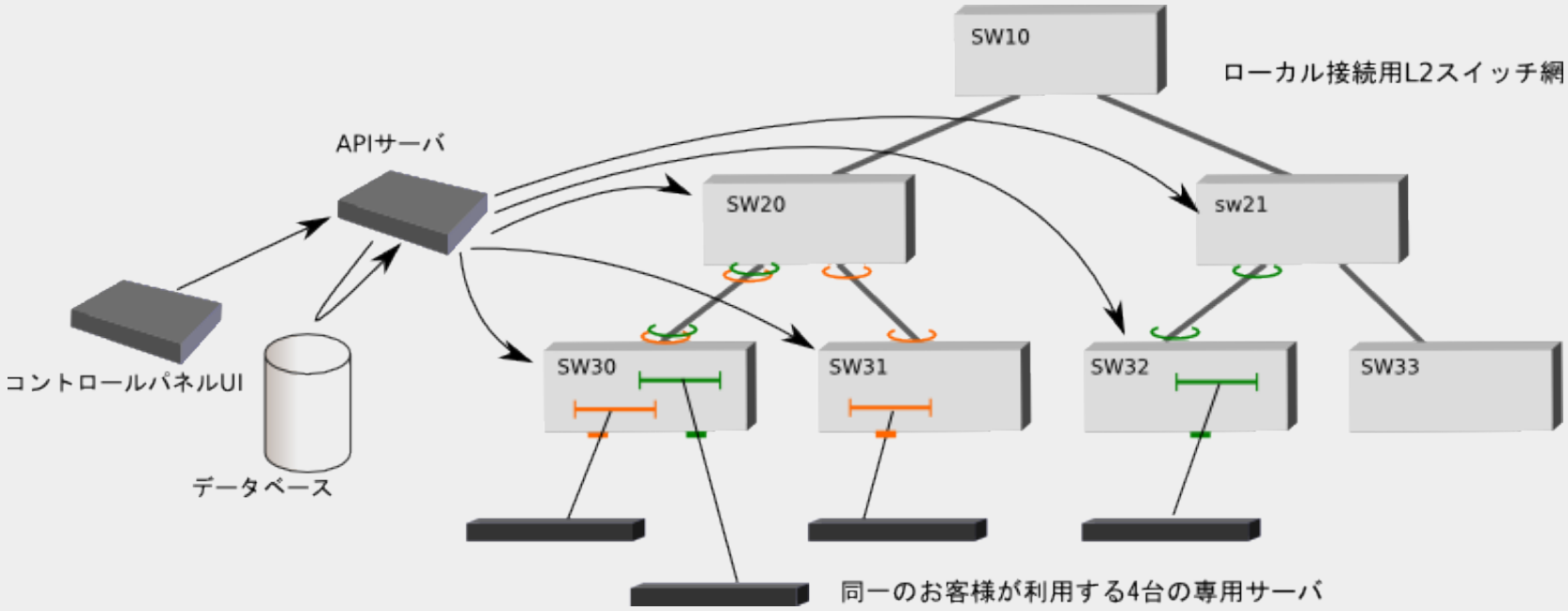
③ スイッチを作成したら、サーバごとに接続先スイッチを選択して接続.

さくらの専用サーバローカル接続開通処理の自動化

④ お客様からスイッチ接続のリクエストを受けたコントロールパネルUIサーバが、APIサーバへリクエストを送信.

⑤ APIサーバはデータベースと照合してから、**サーバ収容スイッチとその上位スイッチと順次NETCONFセッションを開始**

下図のように４台の契約サーバを２台の「スイッチ」に接続するお客様については、以下のように構成を更新します.
- お客様の各スイッチ(オレンジと緑)に該当するVLANを作成、
- 各サーバ収容スイッチ(SW30,SW31,SW32)の収容ポートにアクセスVLANを設定、
- アクセススイッチと上位スイッチ間(SW20-SW30,SW20-SW31,SW21-SW32)の集約論理ポートにVLANを追加



さくらの専用サーバL2設定自動化処理構築担当者の談話

- 2012年2月29日石狩IDCで「さくらの専用サーバ」提供開始以降、
  2012年4月12日コンパネのローカル接続開通機能開始、
  2012年7月31日専用グローバルネットワーク提供開始、
  と段階的にサービスを追加する過程で運用自動化を実装してきました.
- 機能追加の過程でテスト手法も継続的に更新しており、またスイッチ網全体のVLAN整合性の定期監視を実行しています.
- 2014年10月ローカルセグメント10G接続オプションの提供を開始しました.
  - **収容スイッチ選定にあたっては**<u>**API提供が条件**</u>**となりました**.
  - Arista7050を選定、EOS-API(eAPI)を利用してJuniperEXと同様の自動処理を実装しています.

※ 契約終了時の廃止処理におけるスイッチポート閉塞、VLAN削除の自動化についての説明は割愛しております

制御手法の実装例

自動処理デモ用プログラムの概要

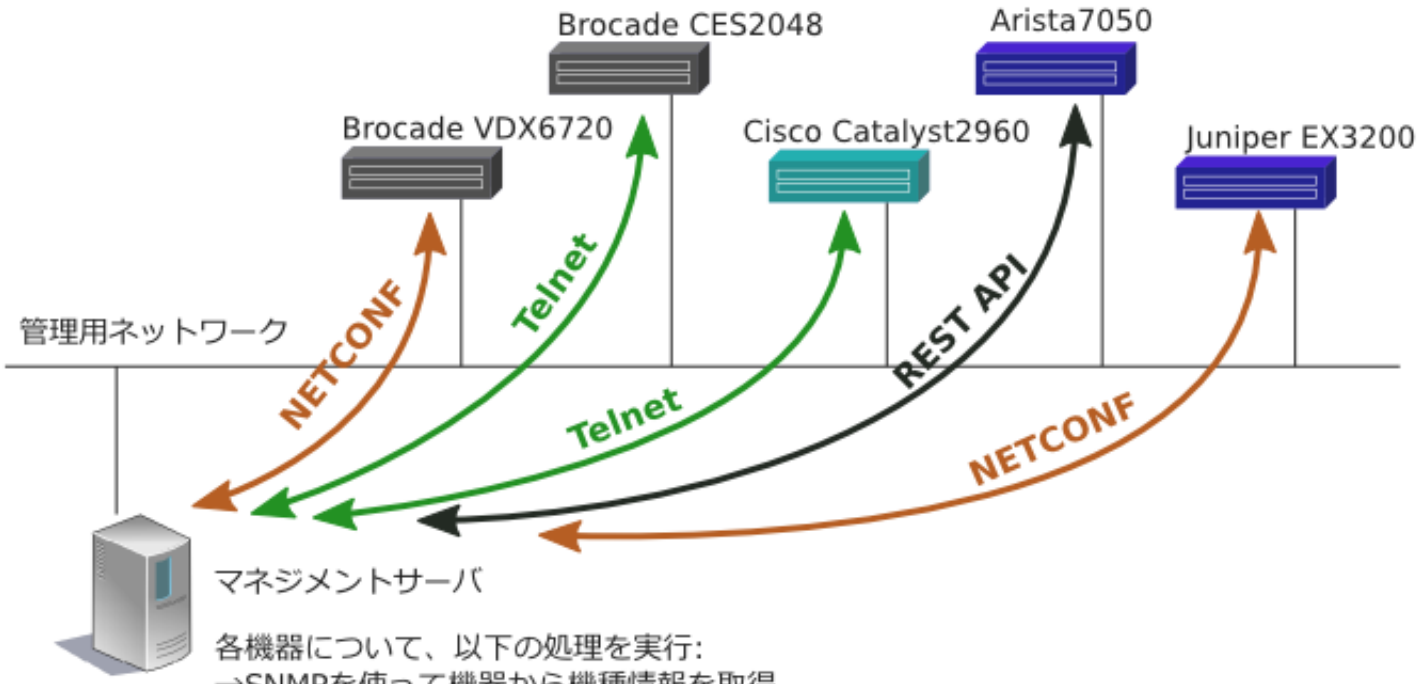
https://github.com/tamihiro/cm_demo

以降のスライドで説明する手法を使って、機種の異なる5台の機器を対象に、順次機種を特定してから、機器へのSNMPクエリを許可する接続元のリストを更新し、startup-configに保存します。

	機種を特定する手法	構成情報を変更する手法
Cisco Catalyst2960	SNMP (共通)	Telnet
Brocade NI CES2048		
Brocade VDX6720		NETCONF
Juniper EX3200		
Arista 7050T		REST API

自動処理デモ用プログラムの概要 (cont.)

マネージメントサーバで実行するプログラムがやること



各機器について、以下の処理を実行:
→SNMPを使って機器から機種情報を取得、
→機種に応じた方法で、running-configを更新するための接続を開始、
→running-configから機器へのSNMP接続を許可するアクセスリストを取得、
→アクセスリストを更新(すべての機器で共通のアクセスリストにする)、
→running-configを保存

自動処理デモ用プログラムの実行結果

```
$ cat update_snmp_acl.log
2014-11-18 13:28:32,809 update_snmp_acl: [INFO] 開始します.
2014-11-18 13:28:37,282 update_snmp_acl: [INFO] NetconfJuniperSess: 192.168.11.101 (juniper): 接続しました.
2014-11-18 13:28:37,666 update_snmp_acl: [INFO] 192.168.11.101: 変更前のACL: 10.0.0.0/24, 172.25.8.0/24, 172.31.30.0/24, 192.168.11.0/24
2014-11-18 13:28:38,486 update_snmp_acl: [INFO] 192.168.11.101: 変更後のACL: 10.0.0.1/32, 172.25.8.0/24, 172.31.30.0/24, 192.168.11.0/24
2014-11-18 13:28:44,047 update_snmp_acl: [DEBUG] NetconfJuniperSess: 192.168.11.101: コミットしました.
2014-11-18 13:28:44,197 update_snmp_acl: [DEBUG] NetconfJuniperSess: 192.168.11.101: セッションを閉じました.
2014-11-18 13:28:45,379 update_snmp_acl: [INFO] NetconfVdxSess: 192.168.11.209 (brocade_vdx): 接続しました.
2014-11-18 13:28:45,650 update_snmp_acl: [INFO] 192.168.11.209: 変更前のACL: 10.0.0.0/25, 172.25.8.0/24, 172.31.30.0/24, 192.168.11.0/24
2014-11-18 13:28:46,503 update_snmp_acl: [INFO] 192.168.11.209: 変更後のACL: 10.0.0.1/32, 172.25.8.0/24, 172.31.30.0/24, 192.168.11.0/24
2014-11-18 13:28:59,800 update_snmp_acl: [DEBUG] NetconfVdxSess: 192.168.11.209: startupを更新しました.
2014-11-18 13:28:59,967 update_snmp_acl: [DEBUG] NetconfVdxSess: 192.168.11.209: セッションを閉じました.
2014-11-18 13:29:00,082 update_snmp_acl: [INFO] EapiHttpSess: 192.168.11.207 (arista): 接続します.
2014-11-18 13:29:00,494 update_snmp_acl: [INFO] 192.168.11.207: 変更前のACL: 10.0.0.0/26, 172.25.8.0/24, 172.31.30.0/24, 192.168.11.0/24
2014-11-18 13:29:01,184 update_snmp_acl: [INFO] 192.168.11.207: 変更後のACL: 10.0.0.1/32, 172.25.8.0/24, 172.31.30.0/24, 192.168.11.0/24
2014-11-18 13:29:02,879 update_snmp_acl: [DEBUG] EapiHttpSess: 192.168.11.207: コンフィグ保存しました.
2014-11-18 13:29:02,879 update_snmp_acl: [DEBUG] EapiHttpSess: 192.168.11.207: セッションを閉じました.
2014-11-18 13:29:04,943 update_snmp_acl: [INFO] TelnetSess: 192.168.11.102 (brocade_netiron): ログインしました.
2014-11-18 13:29:05,142 update_snmp_acl: [INFO] 192.168.11.102: 変更前のACL: 10.0.0.0/27, 172.25.8.0/24, 172.31.30.0/24, 192.168.11.0/24
2014-11-18 13:29:06,397 update_snmp_acl: [INFO] 192.168.11.102: 変更後のACL: 10.0.0.1/32, 172.25.8.0/24, 172.31.30.0/24, 192.168.11.0/24
2014-11-18 13:29:06,779 update_snmp_acl: [DEBUG] TelnetSess: 192.168.11.102: コンフィグ保存しました.
2014-11-18 13:29:07,111 update_snmp_acl: [DEBUG] TelnetSess: 192.168.11.102: セッションを閉じました.
2014-11-18 13:29:08,422 update_snmp_acl: [INFO] TelnetSess: 192.168.11.106 (cisco): ログインしました.
2014-11-18 13:29:08,951 update_snmp_acl: [INFO] 192.168.11.106: 変更前のACL: 10.0.0.0/28, 172.25.8.0/24, 172.31.30.0/24, 192.168.11.0/24
2014-11-18 13:29:10,432 update_snmp_acl: [INFO] 192.168.11.106: 変更後のACL: 10.0.0.1/32, 172.25.8.0/24, 172.31.30.0/24, 192.168.11.0/24
2014-11-18 13:29:12,625 update_snmp_acl: [DEBUG] TelnetSess: 192.168.11.106: コンフィグ保存しました.
2014-11-18 13:29:14,171 update_snmp_acl: [DEBUG] TelnetSess: 192.168.11.106: セッションを閉じました.
2014-11-18 13:29:14,171 update_snmp_acl: [INFO] 終了しました.
```

SNMP

- 1980年代から標準化されている機器の監視と制御のための仕組み.
  - マネージャからエージェント(監視対象機器)宛GET、GETNEXT、SETメッセージ
  - エージェントからマネージャ宛RESPONSE
  - エージェントからマネージャ宛TRAPメッセージ
- 機能面、セキュリティの改善を反映してv2c→v3へバージョンアップ.
- プロトコルとは独立して規定されたメッセージ体系(MIB)は、標準版のほかベンダーによる拡張も可能.
- パフォーマンス、ステータスや、トポロジー情報の取得に使われるケースがほとんどで、機器制御の手段として使われるケースは限定的.

SNMP (cont.)

- 機器制御の手段としては不向きとされる理由
  - セッションレスで信頼性の確保がむずかしい.
  - 機器によってサポートするMIB(標準、拡張とも)の範囲のばらつきが大きい.
    - write memoryの処理は拡張MIBに依存.
  - そもそもどのオブジェクトがread-writeなのかMIBを参照しないと分からない.
    - JUNOSでsetをサポートするオブジェクトは極端に少ない.
  - 構成情報モデルとデータモデルのマッピングの把握も大変.

```
vlan 1004 name VLAN-TEST
 untagged ethe 1/2
!
interface ethernet 1/2
 port-name IF-TEST
 enable
 ip address 10.0.0.5/30
!
```

```
.3.6.1.2.1.17.7.1.4.2.1.5.0.1004 = Hex-STRING: 40 00 00 00 00 00
 00 00 00 00 00 00 00 00 00 00
.3.6.1.2.1.17.7.1.4.3.1.1.1004 = STRING: "VLAN-TEST"
.3.6.1.2.1.2.2.1.1.2 = INTEGER: 2
.3.6.1.2.1.31.1.1.1.1.2 = STRING: ethernet1/2
.3.6.1.2.1.31.1.1.1.18.2 = STRING: IF-TEST
.3.6.1.2.1.17.7.1.4.5.1.1.2 = Gauge32: 1004
.1.3.6.1.2.1.4.20.1.2.10.0.0.5 = INTEGER: 2
.1.3.6.1.2.1.4.20.1.1.10.0.0.5 = IpAddress: 10.0.0.5
.1.3.6.1.2.1.4.20.1.3.10.0.0.5 = IpAddress: 255.255.255.252
.1.3.6.1.2.1.2.2.1.7.2 = INTEGER: up(1)
```

rw

ro

```
$ snmpset -v2c -c secret 192.168.11.102 .1.3.6.1.2.1.2.2.1.7.2 i 2
IF-MIB::ifAdminStatus.2 = INTEGER: down(2)
```

SNMP (cont.)

- よい点
  - ネットワーク機器もサーバも、ほぼ全てのノードでプロセスが動いている.
  - メッセージベース(PDU)のプロトコルなのでサクッとやりとりできる.
  - インターフェイスカウンタ取得などの常套手段だけでなく、監視システムでのちょっとした情報収集に便利(LAGメンバポートのifIndex、BGPピアダウン時のASN、etc.) .
  - Net-SNMPエージェントの拡張コマンドは便利.
- 注意点
  - ファームウェアバージョンアップによって、ベンダ拡張MIBツリーの構成が変わってしまうケースがある.
  - 取得するメッセージのサイズが大きくなると、機器によっては負荷と時間がかかる.

SNMP (cont.)

ちょっとした情報収集の例 (1)
操作対象機器のリストを機種別に分割せずに、 SNMPでsysDescrを取得して判別

```
# 設定変更対象機器のIPアドレス
agent_ipaddrs = ('192.168.11.101', '192.168.11.209', '192.168.11.207',
    '192.168.11.102', '192.168.11.106', )

def get_agent(ipaddr):
  """ipaddrからSNMPで取得するsysDescrを使って機種を判別
 """
  m = re.search(r'(arista|brocade\s+(netiron|vdx)|cisco|juniper)',
    snmpget_sysdescr(ipaddr), re.I)
  if m:
    # Arista、BrocadeNetiron、BrocadeVdx、Cisco、Juniper いずれかのオブジェクトを返す
   return getattr(cm_agent, ''.join(m.group(1).lower().title().split()))(ipaddr)
  else:
    raise ValueError("%s: 機種を特定できませんでした." % (ipaddr))
```

SNMP (cont.)

ちょっとした情報収集の例 (2)
2つのインターフェイスのin/outパケットカウントの1秒デルタを出力して
トラフィック迂回作業時に経路の遷移を確認

```
$ ./count.py -1 rt1 2 -2 rt2 2

interfaces to monitor:
    rt1 : ethernet1/2
    rt2 : ethernet1/2

ok? [Y/n]
+--------+-----------------------------------+-----------------------------------+
|        |                In                 |                Out                |
|  time  |         rt1         rt2       total|         rt1         rt2       total|
|        |      eth1/2      eth1/2            |      eth1/2      eth1/2            |
+--------+-----------------------------------+-----------------------------------+
|00:22:47|       35297           0       35297|       16521       15213       31734|
|00:22:48|       66913           0       66913|       32712        9122       41834|
|00:22:49|       63004           0       63004|       28931       16610       45541|
|00:22:50|       31874           0       31874|       12954        7876       20830|
|00:22:51|       62182           0       62182|       30325       13528       43853|
|00:22:52|       30075           0       30075|       13695       16612       30307|
|00:22:53|       64748           0       64748|       30485        7992       38477|
....
```

NETCONF

- ネットワーク(機器の集合)の高度な管理をめざしているプロトコル.
- セッションベースのプロトコル、RPCでメッセージ交換.

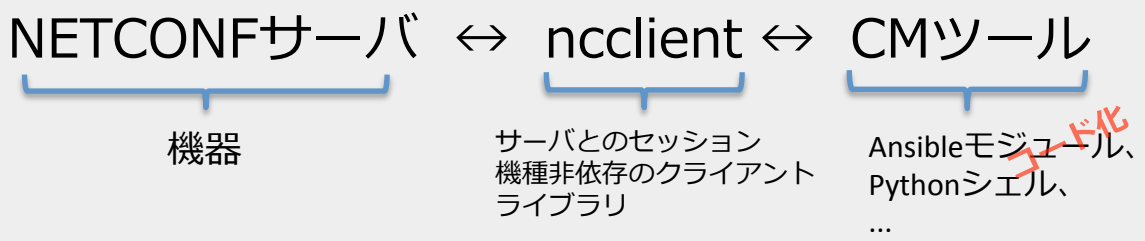
```
<rpc>
<get-interface-information>
<terse/>
<interface-name>ge-0/0/2.0</interface-name>
</get-interface-information>
</rpc>]]>]]>
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" xmlns:junos="http://xml.juniper.net/junos/
12.1R3/junos">
<interface-information xmlns="http://xml.juniper.net/junos/12.1R3/junos-interface" junos:style="terse">
<logical-interface>
<name>
ge-0/0/2.0
</name>
<admin-status>
up
</admin-status>
<oper-status>
up
</oper-status>
<filter-information>
</filter-information>
</logical-interface>
</interface-information>
</rpc-reply>
]]>]]>
```

リクエスト

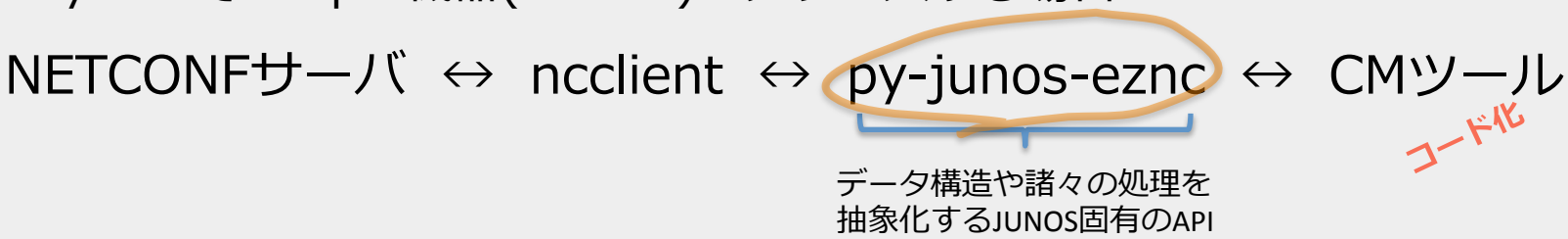
リプライ

NETCONF (cont.)

- 運用者視点で分りやすくつっこんだ解説 (by小島さん)：
  http://codeout.hatenablog.com/entry/2014/10/24/230013
- サーバ機能、データモデル実装度合いはベンダーや機種で異なる.
- クライアント側もいろいろな言語の実装がある.
  - PythonでBrocade VDX(NOS)にアクセスする場合：



  - PythonでJuniper機器(JUNOS)にアクセスする場合：



py-junos-ezncを使う

Juniperがオープンソースで公開した自動処理をサポートするPythonライブラリ

https://techwiki.juniper.net/Automation_Scripting/010_Getting_Started_and_Reference/Junos_PyEZ

✓機器の構成情報やステータス情報をPythonオブジェクトとして直観的に操作できる.

```
>>>
>>> from jnpr.junos import Device
>>> dev = Device(
...             host='192.168.11.101',
...             user='admin',
...             password='iw2014s4',
...             )
>>> dev.open()
>>> dev.connected
True
```

接続時にrpcを実行して収集する機器情報をDeviceオブジェクトの属性として保持する.
(デフォルトの挙動)

```
>>> pp.pprint(dev.facts)
{'2RE': False,
 'HOME': '/var/home/admin',
 'RE0': {'last_reboot_reason':
'0x2:watchdog ',
         'mastership_state': 'master',
         'model': 'EX3200-48T, 8 POE',
         'status': 'OK',
         'up_time': '546 days, 8 hours, 56
minutes, 59 seconds'},
 'domain': None,
 'fqdn': 'R2',
 'hostname': 'R2',
 'ifd_style': 'SWITCH',
 'master': 'RE0',
 'model': 'EX3200-48T',
 'personality': 'SWITCH',
 'serialnumber': '********',
 'switch_style': 'VLAN',
 'vc_capable': False,
 'version': '12.1R3.5',
 'version_info':
junos.version_info(major=(12, 1), type=R,
minor=3, build=5)}
```

py-junos-ezncを使う(cont.)

- ✓ 機器のステータスや学習情報については、YAMLスキーマがライブラリに含まれている.
  https://github.com/Juniper/py-junos-eznc/tree/master/lib/jnpr/junos/op
- ✓ 構成情報についても、自前でYAMLスキーマを作成すればXMLパーズする手間をかけずに簡単にアクセスできる.
  https://github.com/tamihiro/cm_demo/tree/master/cm_sess/netconf_yaml

```
>>>
>>> from cm_sess.netconf_yaml.preflist import *
>>> plt = PrefListTable(dev).get()
>>> json.loads(plt.to_json())['SNMP-ACCESS']
{u'name': u'SNMP-ACCESS', u'entries':
{u'172.31.30.0/24': {u'prefix': u'172.31.30.0/24'},
u'10.0.0.1/32': {u'prefix': u'10.0.0.1/32'},
u'192.168.11.0/24': {u'prefix': u'192.168.11.0/24'},
u'172.25.8.0/24': {u'prefix': u'172.25.8.0/24'}}}
```

```
PrefListTable:
  rpc: get-configuration
  item: policy-options/prefix-list
  key: name
  view: PrefListView

PrefListView:
  fields:
    name: name
    entries: _PrefListEntryTable

_PrefListEntryTable:
  item: prefix-list-item
  view: _PrefListItemView

_PrefListItemView:
  fields:
    prefix: name
```

py-junos-ezncを使う(cont.)

✓set形式コマンドのテンプレートを用意すると、データを構成情報としてロードできる.
https://github.com/tamihiro/cm_demo/tree/master/cm_sess/netconf_templates/preflist_tmpl.set

```
>>> from jnpr.junos.utils.config import Config
>>> cfg = Config(dev)
>>> vars = dict()
>>> vars['acl_dict'] = dict()
>>> vars['acl_dict']['SNMP-ACCESS'] = ['10.0.0.0/24', '10.10.0.0/24', ]
>>> cfg.load(template_path='./cm_sess/netconf_templates/preflist_tmpl.set',
...          template_vars=vars, )
<Element load-configuration-results at 0x104a294d0>
>>> cfg.pdiff()

[edit policy-options prefix-list SNMP-ACCESS]
+    10.0.0.0/24;
-    10.0.0.1/32;
+    10.10.0.0/24;
-    172.25.8.0/24;
-    172.31.30.0/24;
-    192.168.11.0/24;

>>> cfg.commit()
True
```

```
{% for acl_name, acl in acl_dict.iteritems() %}
delete policy-options prefix-list {{ acl_name }}
{% for n in acl %}
set policy-options prefix-list {{ acl_name }} {{ n }}
{% endfor %}
{% endfor %}
```

ncclientでVDXを操作する

VDX(Network OS)でNETCONFを使用する際の制限：

candidateデータストア(非アクティブな設定情報)をサポートしない、ゆえに、

- commit、rollbackはできない.
- lock、unlockもできない.
- 構成情報の変更(edit-config)リクエストはCLIの1コマンド分ずつしか受けとらない.

copy-configリクエストでrunningからstartupへのコピーができない

- Brocade独自のRPCを発行する必要がある.

getリクエストで構成情報を取得できるけどステータス情報を取得できない

- Brocade独自のRPCを発行する必要がある.

ncclientでVDXを操作する (cont.)

VDXのNETCONF実装が限定的であることに加えて、py-junos-ezncに該当するライブラリなしにncclientを使って操作する必要があったため、Juniper用のコード(netconf_juniper_sess.py)のように簡潔にはならなかった.

https://github.com/tamihiro/cm_demo/tree/master/cm_sess/netconf_vdx_sess.py

✓ Brocade固有のRPCを発行する処理は、ncclientで用意されている仕組みを使える

```
# startup更新リクエスト
class VdxSaveConfig(RPC):
  def request(self):
    node = new_ele_no_ns("bna-config-cmd", {'xmlns': brocade_mgmt_urn, }, )
    src = etree.Element('src')
    src.text = 'running-config'
    node.append(src)
    dest = etree.Element('dest')
    dest.text = 'startup-config'
    node.append(dest)
    return self._request(node)

# ncclient.manager内で宣言されているVENDOR_OPERATIONSに追加しておく
manager.VENDOR_OPERATIONS['vdx_save_config'] = VdxSaveConfig
```

NETCONF (cont.)

- よい点
  - プログラマビリティ.
- 注意点
  - get-config や edit-config の処理等を py-junos-eznc のように透過的に実行できるライブラリがない環境では、それなりに頑張ってコーディングすることになる.
    - それでも、再利用可能なコードを蓄積していけば、将来的に自動化のメリットを享受できるはず.
  - NETCONF実装のレベル(capabilities)が機器によって異なるため、扱う機器の対応状況を確認する必要がある.
  - 必要な情報をget-configで取得する際にはきちんとフィルタしないと取得までに時間を要する. (クライアント処理のオーバーヘッドが影響している可能性もあり)
  - 機器のsshホスト鍵を無条件でインポートするようにしておかないと、新規にインストールした機器とのsshトランスポートセッションを開始できない.
  - 取得するメッセージのサイズが大きくなると、機器によっては負荷と時間がかかる.

REST API

- HTTPを使ったシンプルなRPC.
- メッセージフォーマット等の具体的な実装は機器ごとに異なる.
  - Aristaの場合は、リクエストメッセージにCLIのコマンドシーケンスをセットする.

```
>>> req_show = urllib2.Request('http://192.168.11.207/command=api',
...                            json.dumps({
...                                'jsonrpc': '2.0',
...                                'method': 'runCmds',
...                                'params': {'format': 'json',
...                                           'version': 1,
...                                           'cmds': ['enable', 'show ip
access-lists SNMP-ACCESS', ], },
...                                'id': 1, } ),
...                            {'content-type': 'application/json',}, )
>>> req_conf = urllib2.Request('http://192.168.11.207/command=api',
...                            json.dumps({
...                                'jsonrpc': '2.0',
...                                'method': 'runCmds',
...                                'params': {'format': 'json',
...                                           'version': 1,
...                                           'cmds': ['enable', 'configure',
'ip access-lists SNMP-ACCESS', 'no permit x.x.x.x/x', 'permit x.x.x.x/x', ], },
...                                'id': 2, } ),
...                            {'content-type': 'application/json',}, )
```

REST API (cont.)

- レスポンスのボディには、各リクエストコマンド実行結果のシーケンスが含まれる
  （enableやコンフィグモードで実行した５つのコマンドが正常に実行された結果、
  空のデータが５つ返ってくる）

```
>>> res = urllib2.urlopen(req_conf)
>>> res.getcode()
200
>>> res.headers.values()
['60', 'BaseHTTP/0.3 Python/2.7', 'close', 'no-cache', 'no-store, no-cache, must-
revalidate, max-age=0, pre-check=0, post-check=0', 'Tue, 28 Oct 2014 11:50:17 GMT',
'application/json']
>>>
>>> json.loads(res.read())
{u'jsonrpc': u'2.0', u'result': [{}, {}, {}, {}, {}], u'id': 2}
```

REST API (cont.)

- よい点
  - プログラマビリティ.
  - とっかかりやすい.
    シンプルなプロトコルで、スキーマに縛られず、いきなりJSONで受け取れる.
- 注意点
  - 少なくともAristaのeAPIではリクエストしたコマンドセットの途中でエラーが発生した時点で処理を中止 (NETCONFのstop-on-errorと同等)、ロールバックしないので、実行結果のチェックとエラー処理を適宜行う.
  - メッセージフォーマットなど、標準化されているわけではなく各機器ベンダーの任意となるので、将来変更される可能性に留意する.

```
>>> pp.pprint(json.loads(res.read()))
{   u'id': 1,
    u'jsonrpc': u'2.0',
    u'result': [   {   },
                   {   u'aclList': [ ... ],
                       u'warnings': [   u"Model 'AclList' is not a public model and is
subject to change!"]}]}
```

Telnet,ssh (スクリーンスクレイピング)

- リモートから仮想ターミナル越しに実行するCLIと同じコマンドと出力のシーケンスを自動化する.

  ... ログイン → showコマンド、configモードでの構成変更 → 保存 → ログアウト

- Tclの拡張として開発されたExpectと同等の機能が他の多くの言語で実装されています.
- よい点
  - すべての機器で使える (今でも多くの機種でこれ以外にアクセス手段がない)
- 問題点
  - コマンド実行のつど、出力される文字列を情報に変換する必要がある.
    (CLIの実行結果を見た人間の脳内で実行される処理をプログラム化する)
    - 制御リクエスト実行の成否判別
    - showコマンドの出力文字列のデータ化

Telnet,ssh (cont.)

はまりやすいトラブルと回避策

1. コマンド出力結果を正規表現でチェックするだけでは成否の確認が困難
   - 可能なかぎり制御リクエスト実行後はshowコマンドで成否を確認する

```
def update_snmp_acl(self, acl_dict, **kw):
    """ SNMPアクセスリストを更新
  """
    # コンフィグモード
    self.start_config()
    # ACL設定モード
    if hasattr(self, 'config_acl_cmd'):
      self.sendline(self.config_acl_cmd)
      self.child.expect(self.config_prompt)
    # ACL変更
    for which, n in [ (which, n) for which in acl_dict for n in acl_dict[which]]:
      self.sendline(getattr(self, which + '_acl_cmd')(n))
      self.child.expect(self.config_prompt)

    # 変更後に再度showコマンドで取得したACLを返す
    return self.get_snmp_acl(config_mode=True)
```

Telnet,ssh (cont.)

はまりやすいトラブルと回避策

2. プロンプト文字列が出力文字列内の一部とはからずもマッチしてしまう
   - プロンプト文字列に直前の改行コードも含める

```
class TelnetSess(SessBase):
  """telnetセッション用クラス
 """
  def __init__(self, device, pass_login, pass_enable, logger_name,
               user_login=None, telnet_port=23, telnet_timeout=8,
               screen_dump=None):
    ...
    self.unpriv_prompt = "\r\n[-\w]+>\s*$"
    self.priv_prompt = "\r\n[-\w]+#\s*$"
    self.config_prompt = "\r\n[-\w]+\(config.*\)#\s*$"
```

Telnet,ssh (cont.)

はまりやすいトラブルと回避策

3. ページャを無効にできないケースとか制御文字が出力されちゃう場合とか
   - 16進ダンプとってマッチパターンを特定する

```
00000450: 6361 6c36 2069 6e66 6f0d 0a0d 2d2d 2d20  cal6 info...--- 
00000460: 6d6f 7265 202d 2d2d 200d 2020 2020 2020  more --- .      
00000470: 2020 2020 2020 2020 2020 2020 200d 666c                 .fl
00000480: 6f6f 6469 6e67 206c 696d 6974 2062 6c6f  ooding limit blo
00000490: 636b 2031 2031 3030 0d0a 666c 6f6f 6469  ck 1 100..floodi
```

```
lines = re.sub(r'\r\n\r-+\s+more\s+-+\r\s+\r', '\r\n', output).split('\r\n')
```

4. とはいっても想定外の出力文字列は対処しようがない.
   - 想定外の出力には対処しようとしない.
   - 限界を意識して、想定できるパターン以外はエラー扱いにする.
   - show系のコマンドは、ありったけの機材で試してパターンを把握する.

機器ごと手法ごとの処理時間の比較

Brocade NetIron MLXとJuniper EX3200から、10,000前後のARPエントリを取得するまでの所要時間を手法別に比較：

	CLI	SNMP	NETCONF
Brocade NI MLX	00m17s	03m20s	n/a
Juniper EX3200	01m32s	00m30s	01m54s

※クライアント処理のオーバーヘッドもあるので、純粋に機器側の処理だけの所要時間ではありません.

– 1分以上要したケースではいずれもCPUフル使用状態が継続.

– より負荷の少ない手法が機器によって異なることに留意する.

ソフトウェアでの制御を実践するにあたっての注意点

- 認証情報の管理
  - 制御するプログラムへインプットするログインパスワード、特権パスワード、SNMPコミュニティ文字列等の機密を確保する手段は、制御処理の実行方法、実行するサーバへの不正侵入リスク、処理するシステム全体の構成、などに左右され、唯一の正解はない.
  - ターミナルから実行するプログラムであれば、パスワードのハッシュのみ保持して実行時に入力するパスワード文字列と比較できるので、平文パスワードを保持する必要はありません.

```
>>> import hashlib
>>> import getpass
>>> pw_hash = hashlib.md5('naishodayo').hexdigest()
>>> pw_hash
'45acd9f6910a96018aeaf55bcba2df71'
>>>
>>> def auth():
...   pw_plain = getpass.getpass().strip()
...   print hashlib.md5(pw_plain).hexdigest() == pw_hash and "正解！" or "はずれ"
...
>>> auth()
Password:
正解！
```

- テスト
  - インフラエンジニア(われわれ)でも容易に重要性を認識できる.
  - これから本格的にコーディングしようとする人が、ソフトウェア開発の世界で浸透しているテストファーストの手法をいきなり実践しようとすると、敷居がたかくなってしまうかも.
  - それでも、いつの時点からおぼえはじめても有益なので、利用する言語になじんできたら、提供されているユニットテストのフレームワークをぜひ活用してみる.
- ロギング
  - 処理の各ステップの実行結果や、処理中に発生するイベントをもれなく記録すること.
  - 実装言語が備えるロギングの仕組みを活用する.
  - フォアグラウンドで動くプログラムを常時実行する場合は、daemontoolsとmultilogがおすすめです.
    http://cr.yp.to/daemontools.html

- 機器のCPUや制御処理を実行するサーバの負荷に留意
  - 機器の制御処理がサーバの負荷となるケースはあまりないが、
    塵も積もれば山となるので、不要に負荷をかける処理は回避する.
    (不要にサブプロセスを起動せず単一プロセス内で処理する、とか)
- ベンダーロックインのリスクについて
  - (APIに限ったことではありませんが)
    機器固有のAPIに依存するサービス実装を検討するにあたっては、
    将来起こりうる諸々を考慮する.

ここまでのまとめ

ネットワーク機器を制御するいくつかの手法について、それぞれの特徴(長所や制約) を説明してきました．業務での実践のヒントにしてください.

自社で使われている機器について、制御するためのAPIが実装されていたら、まずはLABテストして使い勝手を実感してみてください. バグがあればベンダーに改善要望しましょう.

本セッションでの機器の制御手法についての解説は、ここまでとなります．ただし、、

機器をソフトウェアで制御できるようになっただけでは、ネットワーク運用業務の効率改善を達成できません．日頃CLIで機器を操作するのは、われわれの業務のほんの一部です．

ほかの多くの業務にも自動化が可能な領域があるはずです．むしろそれらの中には、機器制御の自動化達成のおかげで、はじめて自動化が可能になることもあります．

個々の自動化処理の連携、それを使う運用者の取り組み、さらにその先の話へと続きます．