応募区分：研究型論文

Cisco ACI と OpenStack 連携と今後の展望

荒牧　大樹（あらまき　ひろき）
ネットワンシステムズ株式会社第 2 応用技術部

■ 要約

近年 IT システムをクラウドと捉えて、IT システムの利用を簡素化する傾向が主流になって来ている。クラウドの基盤構築を取り扱った多くの例では CPU/メモリ/ディスク容量とクラウドの関係性に付いては言及されているが、NW の視点は抜け落ちている。本論文ではクラウド基盤と NW の関係性について、OpenStack と Cisco ACI を利用して検討を行う。最初に OpenStack と ACI の L2/L3 NW 連携機能について解説を行い、問題点と今後の期待について述べる。さらにクラウド基盤全体の効率的な運用と、基盤上の Application の性能を最大限に引き出すために基盤内で仮想マシンの配置、再配置、Application の拡張を自動的に行う必要がある。NW 視点でこれらの自動化を行うためには NW のステータス取得が必要となる。ACI では NW ステータスを System Health として提供している。この System Health に基づいて、OpenStack 上の仮想マシンの配置、再配置、Application の拡張を自動的に行えないかの実装の検討と検証を行い、更には利用方法の提言を行う。

目次

1. はじめに

近年 IT システムをクラウドと捉えて、IT システムの利用を簡素化する傾向が主流になって来ている。この流れは、社内 IT はもちろんのこと、Service Provider が提供する IT サービスでも同様である。このクラウドの基盤を構築するオープンソースのソフトウェアとして、OpenStack に注目が集まっている。クラウド基盤を構築する場合に仮想化の仕組みは切っても切れない関係性にある。今まで CPU/メモリ/ディスクに関しては仮想マシンの仕組みで提供されてきたが、同様の仕組みを NW にも適用すべく、SDN に注目が集まっている。Cisco はこの SDN をもっと広義の Application Centric Infrastructure と定義して ACI Solution を提供している。今回はこの Cisco ACI と OpenStack の連携でクラウド基盤として提供出来る新しい可能性について検討して行きたい。Cisco ACI と OpenStack の連携で現在実装がほぼ完了している機能としては OpenStack からの指示に基づいて L2/L3 NW を提供する機能がある。今回はこの ACI と OpenStack の実装済みの連携機能の解説と今後の展望を説明する。さらにクラウド基盤にとって、作成された仮想マシンが最適なサーバー上に構築され、必要に応じて再配置され、Application のキャパシティが自動的に拡張、縮小する機能は、基盤全体の効率的な運用と、Application の性能面では非常に重要である。多くの例において、ホストのメモリ・CPU・ディスク容量を考慮して機能を提供する場合はあるが、現状では NW 視点の要素が抜け落ちている。Cisco ACI ではコントロール配下の NW のステータスを System Health として提供する機能がある。今回は ACI の System Health の値に基づいて、OpenStack 上で仮想マシンの最適な配置、再配置、Application の自動拡張・縮小が出来ないかの検討を行う。

2. OpenStack と ACI の連携について

2.1. ACI について

Cisco ACI は Application Centric Infrastructure の名前が示す通り Application を中心に据えた Infrastructure の構築を目指す Solution である。そのコアコンポーネントは APIC(Application Policy Infrastructure Controller)と呼ばれるコントローラーと Nexus9000 シリーズを中心とした NW インフラで実現されている。同様のコントローラーモデルである OpenFlow と異なり APIC が仮に Down したとしても既存の通信には影響を及ぼさない。Nexus9000 を利用した NW インフラ部分に付いては Leaf/Spine の構成で構築される。また、Cisco ACI を大きく特徴付けるのが Policy 制御の機能である。ACI ではサーバーやポートの集合体を EPG(EndPointGroup)として定義して、その関係性を Contract として記述する。それを Application Profile としてまとめることが出来る。

2.2. OpenStack について

OpenStack は IaaS 基盤を提供するためのオープンソースのソフトウェアである。OpenStack リリースはアルファベット順に名前が付けられており、今年前半にリリースされた最新バージョンは Icehouse である。次期リリースは今年後半のリリースが予定されており、Juno となる予定である。今回は Icehouse の機能を中心に議論を行うが、一部 Juno での実装予定の機能に付いても言及している。OpenStack はプロジェクトと呼ばれる機能を提供するモジュール群が集まって提供される。リリ

ース毎にプロジェクトは増えており、各プロジェクトの機能強化と合わせてプロジェクトの追加による新機能の提供も行っている。OpenStack は IaaS の領域にとどまらず DB や Big Data 等の PaaS 領域にも進出を初めている。各プロジェクトは独立して動作する事も可能となっており、必要な機能を個別に抜き出して構成する事も可能となっている。IaaS 基盤を構築する上で代表的なプロジェクトは Nova/Neutron/Cinder/Keystone/Glance/Horizon となっている。Nova は仮想マシンの作成全般を提供し、Neutron は NW、Cinder はブロックストレージ、Keystone は認証、Glance は起動イメージ、Horizon は GUI の提供を行う。この他に Heat/Ceilometer は昨年末のリリースから新しく含まれ Heat は Orchestration、Ceilometer はインフラのデータ収集を担当する。今回は OpenStack のプロジェクトの中でも Neutron/Nova/Ceilometer/Heat が関係する機能について言及する。

2.3. OpenStack の開発の体制について

OpenStack の各プロジェクト内で新しい機能を提供したい場合は Blueprint として新機能が登録される。Blueprint については Launch Pad で管理されており誰でもどのような機能が提案されており、実際に作業が進んでいるのかを確認する事ができる。一定の合意が得られればコードの記述が始まり、コードレビューで問題が無ければ最新の開発ブランチにマージされる。マージされた最新のコードは GitHub 上に公開されており、何時でも参照する事が可能となっている。全く新しい機能はプロジェクトとして組まれ、将来的に正式プロジェクトに昇格して OpenStack として正式に機能提供となる。プロジェクトベースの新機能に付いては OpenStack サイト上の wiki で確認を行うことが出来る。今回は幾つか Blueprint で提案中の機能についても言及を行う。

2.4. OpenStack とマルチハイパーバイザーの対応

OpenStack は KVM/VMware vSphere /Microsoft Hyper-V/Xen と各種ハイパーバイザーに対応している。基本的に Nova が仮想マシンとの接続を行うが、Hypervisor 側でストレージやネットワークの機能の提供を行う場合は Cinder や Neutron も関わってくる。OpenStack では Nova からハイパーバイザーのコントロールを行う場合 KVM と Xen が一番多くの機能を提供している [1]。今回はハイパーバーザーについては OpenStack で利用される事の多い KVM を前提として議論を進める。

2.5. Neutron について

Neutron は NW 機能全般を提供しており、元々は Nova の中にあった Nova Network の機能が切りだされたのが Neutron である。Neutron は比較的新しいプロジェクトであり、Nova の Network 機能である Nova-Network を利用している実例が多いのも現実である。Neutron は L2 NW を提供するコアの機能の他に FW/Load Balancer/VPN/L3 を提供するネットワークサービス機能も提供している。Neutron は L2 NW については Open vSwitch(以下 OVS)/Linux Bridge/IPtables といった機能を活用して NW 機能の提供を行う。基本的にハイパーバイザーが構成されるホスト部分の NW 構成は OVS や Linux Bridge 等の機能を利用する事で実現されている。L3 NW については OVS/Linux Bridge と Network Namespace の機能を組み合わせて仮想ルーターとして提供している。Neutron では外部の物理 NW 部分については各社が提供するドライバーを利用して連携する構成になっている。従来は一つの Neutron インスタンスに付き、一つのドライバーしか利用出来なかったが、現実の環境では複数の仮想スイッチ・物理スイッチが導入されている。その状況に対応するために ML2(Modular Layer2)といった機能が提供された。ML2 では一つの Neutron インスタンスで複数の

メーカーの NW 機器・仮想スイッチや、NW 分割方式を利用することが出来る。

2.6. Modular Layer 2 について

Neutron のドライバーとして Modular Layer 2 ドライバーは標準となることが期待されている。複数の NW をサポートするために ML2 内部で更に各種のドライバーが動作するが、大きくタイプドライバーとメカニズムドライバーの 2 種類のドラバーに分かれている。タイプドライバーは NW の分離の方式を設定する事が出来る。例としては VLAN/VXLAN/GRE 等々である。メカニズムドライバーは各社の複数のドライバーを含む事が出来る。例として、OVS のドライバーと Cisco のスイッチのドライバーのような物がある。今回は Cisco の APIC 用のドライバーを利用した場合の構成について言及する。

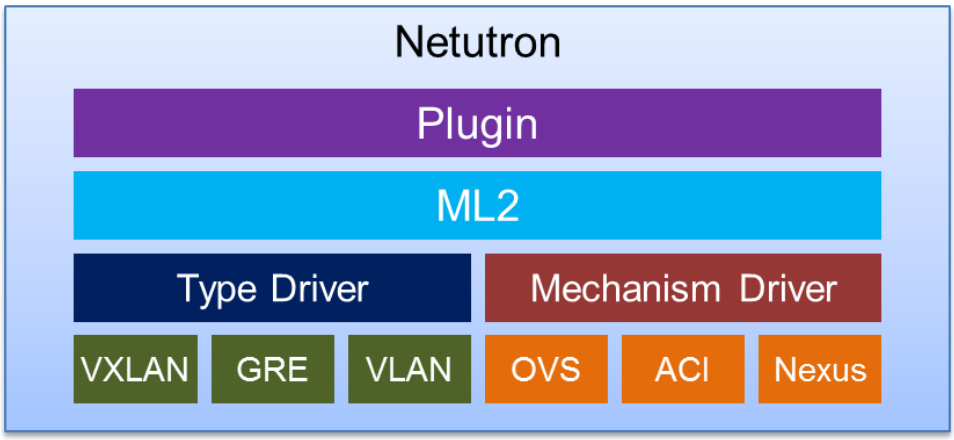

図 1　ML2 の構成

2.7. 実装済み OpenStack の ACI の L2/L3 連携について

OpenStack の ACI 連携用のドライバーは最新の Icehouse リリースでは標準では含まれていない。ACI と OpenStack の L2/L3 連携については今秋リリース予定の Juno に標準で含まれる予定だ。Juno で提供されるネットワーク機能は、L2 部分に関しては、Neutron の ML2 ドライバー、L3 部分は L3 サービスドライバーとして提供される予定である。現在 Github 上に公開されている次期バージョンである Juno 用の master レポジトリでその存在を確認する事が出来る。実際の動作を確認すると図 2 のように OpenStack 上で Network/Router の設定を行うと、ACI 上では図 3 のように EPG/Contract として反映される。現状は VLAN をベースとした構成がサポートされており NW 毎に OpenStack で自動的に割り振られた VLAN のタグがコンピュートノードで添付されて、その後 ACI の NW で適切に処理がされる。ACI を特徴付けている Policy の考え方は適用できず、固定値が割り振られる。この ACI に関連する機能との連携については Juno でのリリースを目標に Group Policy という名前で Neutron への取り込みの作業が進んでいる [2]。

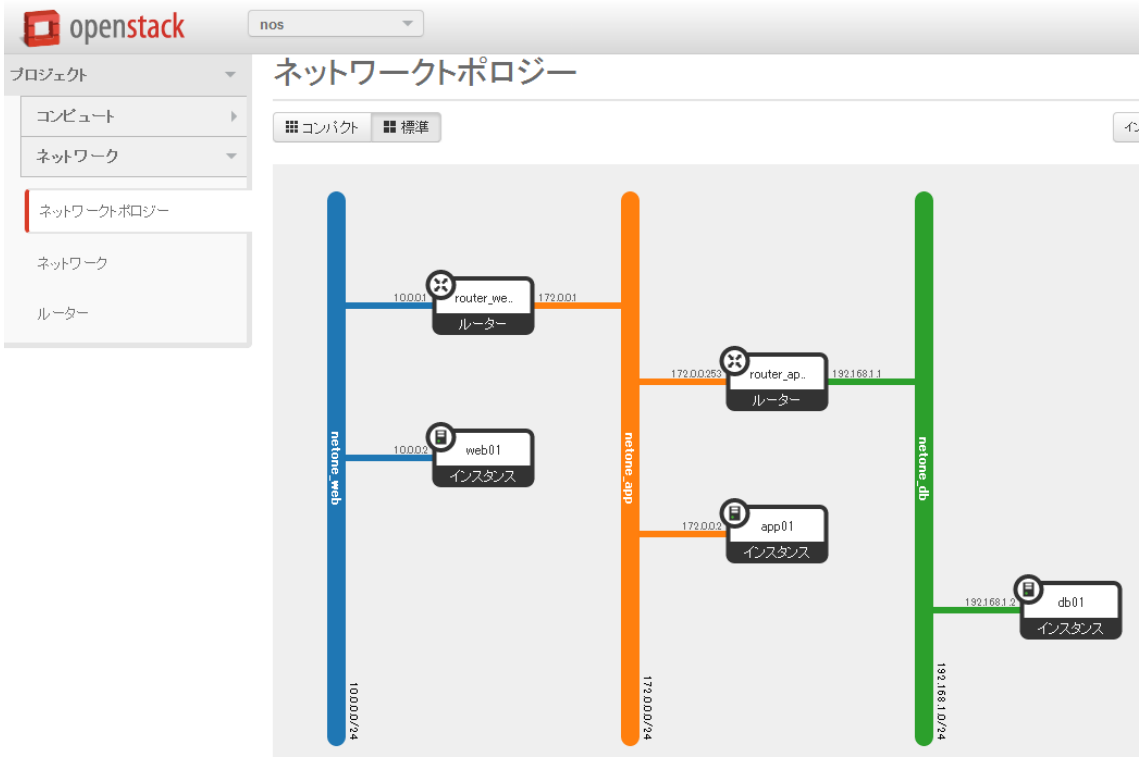

図 2 OpenStack 上の設定

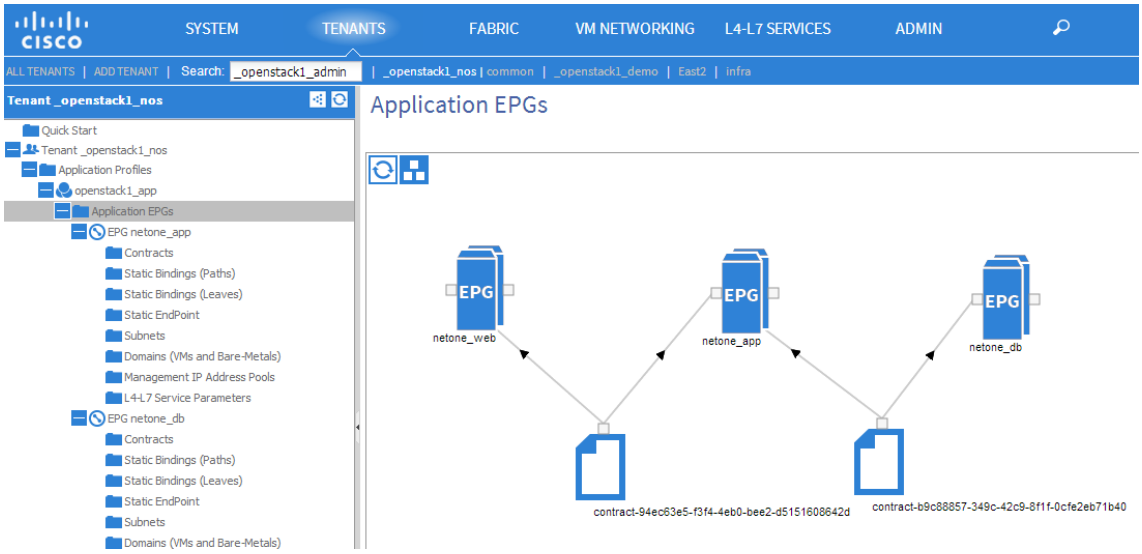

図 3　APIC 上の表示

2.8. ACI と OpenStack 連携の制約

OpenStack と ACI の連携を行うといくつかの制約を確認出来る。1 つ目は Floating IP の割り当てである。Floating IP の利用想定は外部 NW との接続となり、Floating IP を特定の仮想マシンに設定する際に、NAT ルールが追加され、その仮想マシンは外部との双方向の通信が可能となる機能である。Floating IP の問題を回避する方法は APIC ドライバーの設定で外部通信用の NW の名前とスイッチ

のポートを指定しておくとそのポートとネットワークを通じて外部通信が行えるようになる。しかしながら当初の目的である NAT 処理が行えるわけでは無い。2 つ目の制約は Metadata Proxy への対応である。OpenStack では仮想インスタンスが起動すると内部にインストール済みの Cloud-init が"http://169.254.169.254"へのアクセスを試みて、自身の設定情報の Download を行う。Neutron ではこの接続を Metadata Proxy プロキシ機能を使って Nova 側へ転送している。ACI 側で L3 機能が提供されると、この Metadata Proxy が利用出来ない。残念ながら今の所これを解消する手段は無い。3 つ目の制約は VLAN のみで現状 VXLAN を利用出来ない点である。これに関しては将来的なサポートが予定されている。

2.9. Neutron ACI 連携の利用方法と今後の期待

現状の OpenStack と ACI 連携においては L2/L3 機能の提供に留まるため、次のような 3 つの場合に ACI 連携を利用すると考えられる。一つは、物理ファブリックのパフォーマンスを活用し、外部あるいは内部での NW トラフィックが数十 Gbps から数百 Gbps となる場合が考えられる。二つ目に、L3 機能として Dynamic Routing 等を提供したい場合、三つ目として、Default Gateway に完全な冗長化が必要な場合となる。OpenStack と ACI の連携を、より多くのシステムに適応するためにも、L2/L3 利用時の制約の早期の解消が期待される。具体的には Floating IP/VXLAN 対応/Metadata Proxy の問題である。更に、今後 ACI を特徴付けている Policy の考え方が Group Policy として OpenStack 側に実装されると利用する場面も増えてくると考えられる。例えば OpenStack の Heat では仮想マシンから NW まで一連の流れで自動作成できるが、その時に Group Policy も適用できれば Application を提供する仮想マシンと Application を意識した NW がワンクリックで作成出来る事になる。また、OpenStack では Service Chain の機能も実装されているので [3] 、ACI のサービスグラフの機能と連動できれば NFV 等でも様々な利用例が出てくると考えられる。

3. 仮想マシン起動時の仮想マシンの配置と ACI

仮想マシン起動時には複数ある仮想マシンをホストするサーバーから一台のサーバーが選択されて、仮想マシンが配置される。この仮想マシンの作成と配置は Nova が実施している。この仮想マシンの配置を ACI で提供される System Health を参照することで実現できないかの検討を行う。

3.1. Nova について

Nova(Compute)は OpenStack のモジュール内で仮想マシンの作成を担当している。KVM や VMware 等様々な Hypervisor に対応している。OpenStack ではコントローラーノードとコンピュートノードの 2 種類のノードが作成される。仮想マシン作成時に Nova の Scheduler は複数あるコンピュートノードから最適な 1 台を選択する。Nova は標準では Filter Scheduler を利用して新規に仮想マシンを配置する。Filter Scheduler 以外にはランダムにホストを選択する Chance Scheduler も実装されているが今回は検討の対象から外す。

3.2. Nova の Filter Scheduler について

Nova は仮想マシンをどのホストに配置するかの決定を行う場合に Filter Scheduler を利用している。CLI/GUI により仮想マシンの作成が行われると、Filter スケジューラーはまず各種条件でホストをフ

ィルタリングしてホストの候補を絞って行く、絞った後に複数の候補がまだある場合は重み付を行い適切な一台の選択を行う [4]。

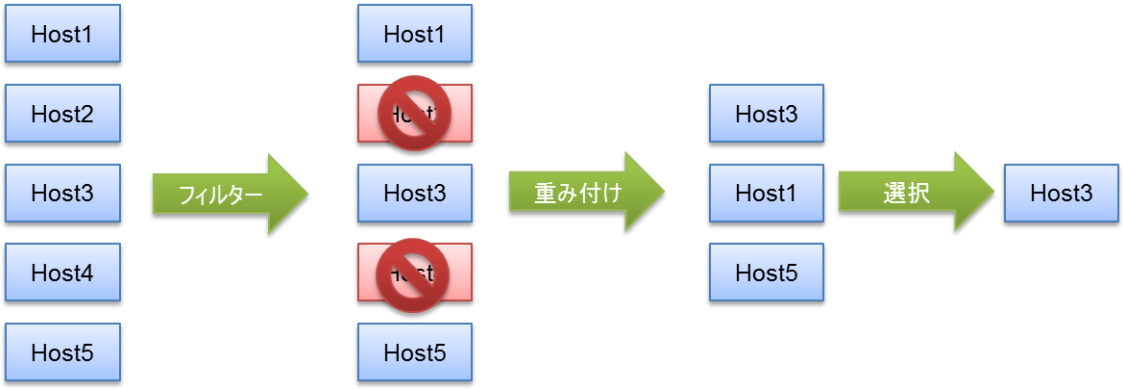

図 4 Filter Scheduler の動作

現状 Filter Scheduler は標準で以下の値を元に判断を行う。

フィルター名	機能
RetryFilter	リトライ数の超えたホストを除外します。Default では 3 回試行します。
AvailabilityZoneFilter	ブート時に設定された AZ によりフィルターを行います。
RamFilter	RAM によりフィルターを行います。Default 値では物理 RAM の 1.5 倍まで許容されます。
ComputeFilter	全ての Compute Host を選択する
ComputeCapabilityFilter	Instance Type に設定された Extra Spec を満たすか確認します。
ImagePropertiesFilter	起動イメージに事前に設定された CPU/RAM 値によりフィルターを行います。
CoreFilter	コア数によるフィルターを行います。Default 値では Core 数の 16 倍まで許容されます。

表 1 Default で設定されているフィルター

これ以外のフィルターもサポートされているが今回は詳細を説明しない。重み付けに関しては RAM の空き容量と、Icehouse から新しくサポートされた Metric 値(Utilization Based Scheduling)がある。

3.3. Utilization Based Scheduling について

Utilization Based Scheduling は Icehouse から導入された仕組みで、ホストの様々なステータスを元に Scheduling を行うことが想定されている。Scheduler で評価されるデータは Compute Node 上で定期的に収集され、JSON 形式でコントローラーノードの SQL 内に保存される。具体的には

compute_nodes テーブルの metrics の項目に JSON 形式で保存される。Icehouse では標準で CPU 利用率の取得が実装されている。Filter Scheduler はこの値を利用することができ、Filter と Weight 両方に適用できる。今回は Metric 値を Weight で利用する。Weight で Metric 値を利用する場合は JSON で保存された値は、設定で倍率を設定することができ、全ての値が合算されて重み付けが行われてホストが選択される。 [5]

3.4. ACI の System Health について

ACI の環境では APIC を通じてファブリックの Health スコアを参照する事が出来る。ヘルススコアはファブリック全体から、各スイッチ、ポートや 3rd Party のコンポーネント等広く様々な場所で確認する事ができる。今回はファブリック内のサーバーホストが接続されているリーフスイッチのポートのヘルススコアを利用する。

3.5. ACI の System Health を利用しての仮想マシンの配置

Compute Node から自身が接続されているスイッチポートの Health Score をレポートしておき、Nova の Filter Scheduler で、この Metric 値を参考に配置を行えば System Health の一番良い値のホストに配置する事が可能となる。ここで問題となるのは Nova でどのような方法で APIC から値を取り出して保存するのかと、Nova の Filter Scheduler で、どのように Metric 値を参照するかとの事である。今回は nova-compute のプロセスの一部として動作する Python Script を作成して定期的に ACI の Health score の値をレポートし、その値と元に仮想マシンの配置が出来るかの検証を行った。

3.6. 検証環境でのテスト

3.6.1. 検証概要

今回は osp5com01.noslab.com と osp5com02.noslab.com の 2 台の Hypervisor を用意した。osp5com02.noslab.com のメモリ搭載量を増やし、RAM Weigh により通常状態では空きメモリの大きい osp5com02.noslab.com が選択されるようにしておく。その状態で osp5com01noslab.com のみ Health Score の値を入力しておく。その状態で Health Score が無いホストにはペナルティが入る状態にしておくと osp5com01.noslab.com が選択されると正しく Health Score が参照されている事がわかる。この動作を図示したのが図 5 になる。

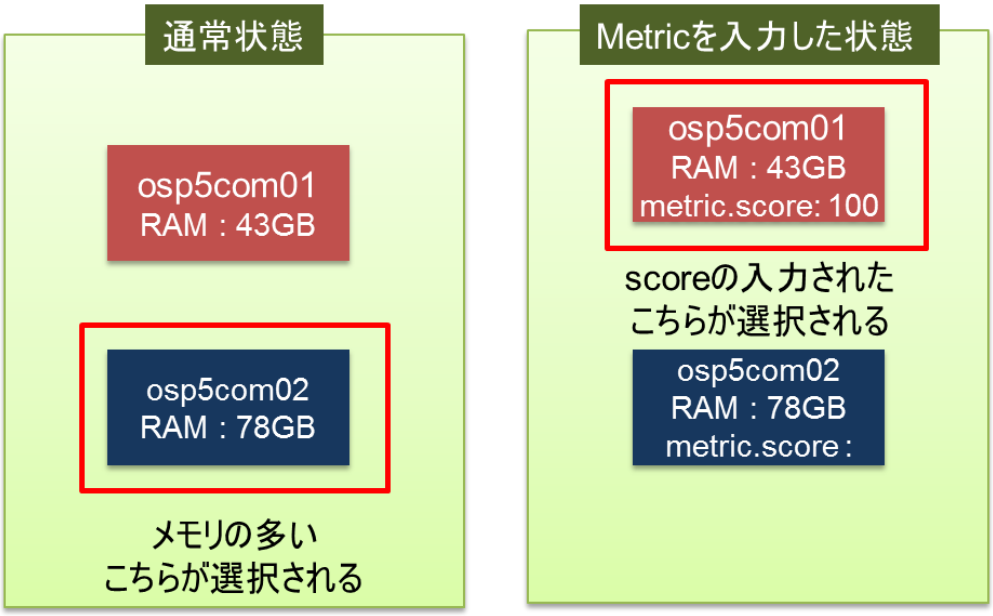

図 5 Metric 値の参照

3.6.2. Metric 値の取得と保存

Metric 値は Compute Node が定期的に APIC に対して、自身が接続されたポートの Health Score を取得する。その値は Controller Node にレポートされ、最終的に Database に保存される。

図 6 動作概要

APIC の Health Score を Metric 値として定期的にレポートするために、APIC から取得した Port の Health 値を送付する Agent を Python で作成した。Agent のコードは付録 1 として添付した。この Agent を参照するために以下通り nova.conf の設定を行った。

```
#
# Options defined in nova.compute.monitors
#

# Monitor classes available to the compute which may be
# specified more than once. (multi valued)
compute_available_monitors=nova.compute.monitors.all_monitors

# A list of monitors that can be used for getting compute
# metrics. (list value)
compute_monitors=MetricMonitor
```

取得した値が正しく反映されているかは SQL の nova データーベース上の compute_nodes の metric の値を見ると確認できる。図 7 では Score が 100.0 として 2 台ある Hypervisor の内一台から Metric

値が返されている事がわかる。

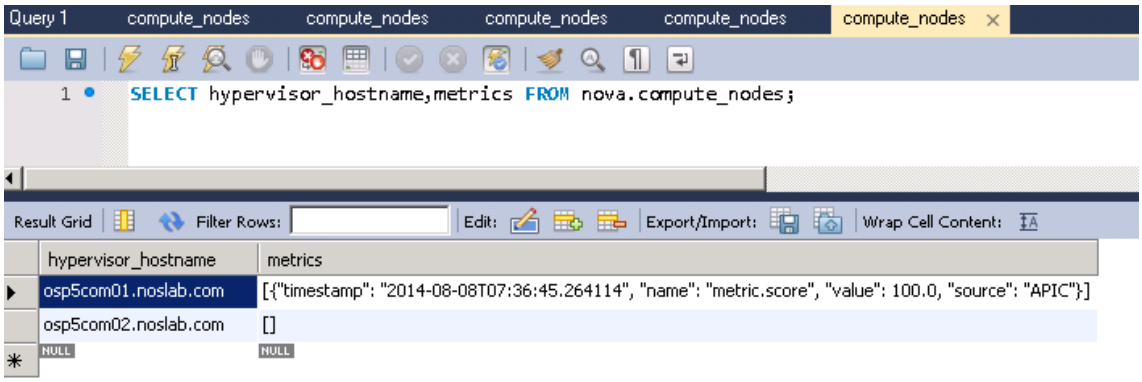

図 7　SQL 内のメトリック値の参照

3.6.3. Scheduler の設定と動作確認

Filter Scheduler の設定で metric を参照するように設定を行った。Metric は Filter と Weight に適用可能だが、今回は Weight で参照する設定を行った。設定は以下の通りで System Health の Score がレポートされない場合はペナルティが課される設定を行った。

```
[metrics]
# Multiplier used for weighing metrics. (floating point value)
weight_multiplier=1.0
# metric 名と Weight の設定
weight_setting=metric.score=1.0
# false とすると値が無い場合にはペナルティが課される。True とすると Error となり動作しない
required=false
# 値が無い場合に返される Weight
weight_of_unavailable=-10000.0
```

この状態で新規仮想マシンを作成した。以後 Scheduler の動作をログから確認する。Scheduler のログを見ると、空きメモリは osp5com02.noslab.com の方が大きいため通常は osp5com02.noslab.com が選択される。

```
Filtered [(osp5com01.noslab.com, osp5com01.noslab.com) ram:38977 disk:8192 io_ops:0 instances:3,
(osp5com02.noslab.com, osp5com02.noslab.com) ram:77769 disk:1878016 io_ops:0 instances:1]
```

ここに Metric の考え方を適用すると、System Health の Metric 値を参考にして Weight は osp5com01.noslab.com の方が大きくなり、osp5com01.noslab.com が選択された。

```
Weighed [WeighedHost [host: osp5com01.noslab.com, weight: 1.50118941995], WeighedHost [host:
osp5com02.noslab.com, weight: 1.0]]
```

これにより ACI の Health Score を元に Nova Scheduler は仮想マシンを配置している事がわかる。

3.7. ACI Health Score と仮想マシン配置の連携の利用例

今回 APIC から Switch のポートステータスを抜き出して、その値を仮想マシンの配置に利用出来る事を確認した。これにより、NW を意識した仮想マシンの配置が出来る事がわかった。今回はホストの接続されたポートを対象にしたが、その上位のリーフスイッチの Health Score 等様々な場所の値を利用可能となる。これにより、例えば信頼性が必要とされるアプリケーションが稼働する仮想マシンは、より Health Score の高いスイッチに配置したり、ある一定値以下の Health Score となっているスイッチにはなるべく仮想マシンを配置しない等の様々な利用方法が考えられる。

4. OpenStack の Auto Scaling と ACI

Auto scaling については OpenStack のプロジェクトの中で Heat と Ceilometer が連携することで機能を提供している。Ceilometer が仮想マシンの CPU 利用率等の各種値を監視して、閾値以上になると Heat に Push して Auto Scaling が実行される。Heat では仮想マシン数の増減をサポートしているが、仮想マシン単体の vCPU やメモリを追加するようなスケールアップ・ダウンに付いてはサポートしていない。

4.1. Heat について

OpenStack の中で Heat はオーケストレーション機能を提供する。AWS の Cloud Formation に対応した JSON 形式と、Heat 用の YAML 書式の HOT テンプレートの 2 つをサポートしている。このテンプレートを元に起動された一連の動作をスタックと呼ぶ。テンプレートの中では NW の作成、削除や、仮想マシンを起動と作成、Auto-Scaling についての設定を行うことが出来る。

4.2. Ceilometer とデータ収集について

Ceilometer は OpenStack 内の情報の収集、蓄積、取り出しを担当している。統計情報の取り出しも可能なので課金等での利用が見込まれている。Ceilometer でのデータ収集は BUS/Push/Polling と 3 つの方式をサポートしている。 [6] BUS に関してはリソースが MQ を通じて Ceilometer にデータを送付する手法となる。Ceilometer が余裕のある時に MQ からデータを取り込んでデーターベースに取り込むため、Ceilometer の負荷を最小限に抑える事が出来、この手法が推奨されている。OpenStack に標準で用意されている Nova Agent 等はこの方式を採用している。次に推奨されているのは PUSH の方式となる。REST による PUSH(POST)によって、Ceilometer 側にデータを送信する。非推奨となっているのは Ceilometer から対象リソースへの Pooling となる。Ceilometer 自身が Polling 動作を行うため Ceilometer の処理に大きな影響を与えてしまうため非推奨となっている。この Ceilometer に ACI の System Health の値を定期的に保存しておけば Heat で利用可能となる。

4.3. Heat と Ceilometer の連携

Heat と Ceilometer を連携すると Auto-Scaling を提供出来る。Ceilometer で取得される値に対して、サーバーの増減が開始される。Heat はあらかじめ設定された数の仮想マシンを増やしたり、減らしたりの操作を行う。実際は CPU が 80%以上になれば、仮想マシンを増やして 50%以下になれば増やした仮想マシンを減らして行くような動作が可能となっている。利用方法としては、Web サーバーの

負荷が増えた時に Web サーバーの仮想マシンを増やして Load Balancer と連動して自動的に負荷分散を行う動作が可能となる。

4.4. Auto-scaling と APIC

Heat は Ceilometer 上に設定される閾値により Auto scaling を行うことが出来るため、APIC 側から System Health を Ceilometer に代入しておけば System Health の値によって Auto-Scale が可能となる。CPU 使用率やメモリ使用率と共に System Health が閾値以下/以上になればシステムの能力低下を検知して Auto-Scaling を行えると考えられる。

4.5. 検証環境でのテスト

4.5.1. Ceilometer へのデータのレポート

まずは Ceilometer に対して、APIC から取得した Score をレポートする。Agent を作成して定期的に APIC から Health score を取得して、Ceilometer に対して Health score の書き込みを行う。

図 8APIC と Ceilometer

今回付録 2 のようなサンプルコードを記述した。それぞれの APIC/OpenStack 両方で認証後に APIC から情報を REST で GET して、REST の POST でデータの書き込みを行っている。Ceilometer に正しくデータが反映されているかに関して、以下の用に Ceilometer のコマンドにより確認する事ができる。今回は 2 つのホストの情報が 100.0 として保存されていることがわかる。

```
[root@LBaaS-SR12 ~(keystone_admin)]# ceilometer sample-list -m healthscore
+----------------------+-------------+-------+--------+------+----------------------------+
| Resource ID          | Name        | Type  | Volume | Unit | Timestamp                  |
+----------------------+-------------+-------+--------+------+----------------------------+
| osp5com01.noslab.com | healthscore | gauge | 100.0  | %    | 2014-08-12T04:34:35.561000 |
| osp5com01.noslab.com | healthscore | gauge | 100.0  | %    | 2014-08-12T04:33:35.812000 |
| rhelosp01            | healthscore | gauge | 100.0  | %    | 2014-08-12T04:09:28.956000 |
| rhelosp01            | healthscore | gauge | 100.0  | %    | 2014-08-12T00:47:15.030000 |
+----------------------+-------------+-------+--------+------+----------------------------+
```

図 9 データの表示

4.5.2. Heat での Auto Scaling の起動

Heat で作成した、仮想マシンの CPU 使用率等を監視して Auto Scale を実行する仕組みはあるが、今の所 Heat で最初に作成される仮想マシンの配置されたホストを割り出して、自動的に検出出来る仕組みが無い。このため Health score が悪化した場合に Heat でスケール Up させる事が出来ないことがわかった。

4.6. ACI Health score と Auto-Scaling の連携時の利用例

Heat でホストと仮想マシンの関係性を確認する方法が無いため Health Score を元にして Auto-Scale が出来ない事がわかった。今後対応した場合は次のような利用方法が考えられる。例えば Health

Score が悪化すると新規仮想マシンを作成する。この場合は同一のホストに配置しては意味がないので前出の Scheduler と連動できれば新規仮想マシンは Health Score の良いホストに配置する事が可能である。さらに LB との連携も行えるのでトラッフィクの配信の調整も行えばシステム全体としては NW の問題点の影響を最小限に抑えてサービスの提供が可能となる。

5. 仮想マシンの再配置と ACI について

現状仮想マシンの再配置は CLI/GUI での手動での再配置のみサポートされている。この再配置が自動化された場合に ACI の System Health の値が利用出来ないか検討する。

5.1. OpenStack と仮想マシンの再配置について

OpenStack 環境で仮想マシンの再配置を行う場合は、Nova の Migration の機能を利用する事となる。KVM 環境の OpenStack では Migration と Live Migration がサポートされている。Migration は Shutdown 済みの仮想マシンに対してホストの移動を可能としている。Live Migration は起動中の仮想マシンに対して、ホスト間の移動を可能としている。共有ストレージを持たない場合でも Live Migration は—block-migrate のオプションを利用することでストレージの Migration と仮想マシンの Migration を同時に行う事が可能となっている。

5.2. 仮想マシンの再配置の自動化と Congress について

OpenStack の仮想マシンの再配置の自動化に関してはまだ標準の機能としては実装されていない。再配置を自動的に行う事を目的としている Congress [7]というプロジェクトが立ち上がっており議論が進んでいる。Congress は Neutron で実装が進んでいる Group Policy の考え方を IT 全体に広げたもので各種 IT のシステムの Policy の定義を行う事を目的として開発が進められている。動作としては Policy を設定しておくと、Policy に従って動作し、何らかの Policy 違反が発生するとその問題点を自動的に修正されることが期待されている。この定義を Nova に適用すると Policy 違反が発生した場合に一つの選択肢として仮想マシンの再配置が行われる。

5.3. ACI Health score を元とした仮想マシン再配置の利用例

再配置に関してはまだ、実装が進んでいる途中なので今回は動作の検証を行わなかった。Congress と連携した場合 ACI の Health Score に関しての Policy を設定しておけば、Health Score の値がある値以下になると仮想マシンの再配置が行えるようになると考えられる。仮にトラフィック集中や、リンクダウン等でホストの状況が悪化した場合に自動的に仮想マシンの再配置を行う事が可能となる。

6. まとめ

今回は OpenStack と ACI の連携の様々な可能性について検討した。ACI と OpenStack の L2/L3 連携は NW の基本的な動作はすでに実装が進んでいる。今後 Group Policy との連携機能や、提示した問題点の解消が期待される。ACI の Health Score の活用方法として 3 種類の方法を提示した。仮想マシンの配置に付いては Metric 値として Health Score をレポート出来るようにしておけば簡単に Nova の Scheduler から利用できることがわかった。Auto Scale に関しては、Ceilometer への値の保

存は実装可能な事は分かったが、Heat に関してはホストを意識した動作の実装が期待される。Auto-Scale 対応の第一歩として APIC から Ceilometer に対応して Health Score をレポート出来るようになれば様々な活用方法が今後生まれると考える。仮想マシンの再配置に関しては OpenStack 側の実装の問題でまだまだ出来ない事が多いが、今後順次機能が追加されていくと考えられる。以上、OpenStack と ACI の連携について様々な検討を行ったが、今後も両者の連携の強化を進めて行けば純粋な L2/L3 の NW 機能に留まらず NW の状態を意識した様々な機能をユーザーに提供可能なことがわかる。

7. 引用文献

1. OpenStackHypervisorMatrix. (オンライン)
https://wiki.openstack.org/wiki/HypervisorSupportMatrix.
2. Neutron/GroupPolicy. (オンライン) https://wiki.openstack.org/wiki/Neutron/GroupPolicy.
3. Neutron Service chaining. (オンライン)
https://blueprints.launchpad.net/neutron/+spec/neutron-services-insertion-chaining-steering.
4. Filter Scheduler. (オンライン)
http://docs.openstack.org/developer/nova/devref/filter_scheduler.html.
5. Utilization aware scheduling. (オンライン)
https://blueprints.launchpad.net/nova/+spec/utilization-aware-scheduling.
6. Ceilometer Architecture. (オンライン)
http://docs.openstack.org/developer/ceilometer/architecture.html#how-is-data-collected.
7. Congress. (オンライン) https://wiki.openstack.org/wiki/Congress.

8. 付録 1

```
import json
import requests
from nova.compute import monitors


class APICController(object):
    def __init__(self, apic_ip):
        self.base_url = 'https://%s/api' % (apic_ip)
    def login(self, name, pwd):
        name_pwd = json.dumps(
            {'aaaUser': {'attributes': {'name': name, 'pwd': pwd}}})
        login_url = '%s/aaaLogin.json' % (self.base_url)
        #need login error handling
        post_response = requests.post(login_url, data=name_pwd, verify=False)
        # get token from login response structure
        auth = json.loads(post_response.text)
        login_attributes = auth['imdata'][0]['aaaLogin']['attributes']
        return login_attributes['token']
    def get_port_health(self, auth_token, switch):
        cookies = {}
        # set cookie
        cookies['APIC-cookie'] = auth_token
        # create health score URI
        request_url = '%s/mo/topology' % (self.base_url)
        if 'pod' in switch:
            request_url = '%s/pod-%s' % (request_url,switch['pod'])
        if 'node' in switch:
            request_url = '%s/node-%s' % (request_url,switch['node'])
        if 'slot' in switch:
            request_url = '%s/sys/ch/lcslot-%s' % (request_url,switch['slot'])
        if 'port' in switch:
            request_url = '%s/lc/leafport-%s' % (request_url,switch['port'])
        request_url = '%s/health.json' % (request_url)
        get_response = requests.get(request_url, cookies=cookies, verify=False)
        # return json data
        return get_response.json()


class MyDriver(object):
    # Example driver to get Leaf switch health

    def get_metric_score(self, **kwargs):
        apic_info={'ip':IPADDRESS,'user':ADMIN,'pwd':PASSWORD}
        switch = {'pod': '1', 'node': '101', 'slot': '1', 'port': '1'}
        apic = APICController(apic_info['ip'])
        token = apic.login(apic_info['user'],apic_info['pwd'])
        json_data = apic.get_port_health(token, switch)
        score = json_data["imdata"][0]["healthInst"]["attributes"]["twScore"]
        #return health score as a floating value
        return float(score)


class MetricMonitor(monitors.ResourceMonitorBase):

    def __init__(self, parent):
        super(MetricMonitor, self).__init__(parent)
        self.source = "APIC"
        self.mdriver = MyDriver()

    def _get_metric_score(self, **kwargs):
        # Return health score and its timestamps.
        return self.mdriver.get_metric_score(), None
```

9. 付録2

```
import json
import requests

class APICController(object):
    def __init__(self, apic_ip):
        self.base_url = 'https://%s/api' % (apic_ip)
    def login(self, name, pwd):
        name_pwd = json.dumps(
            {'aaaUser': {'attributes': {'name': name, 'pwd': pwd}}})
        login_url = '%s/aaaLogin.json' % (self.base_url)
        #need login error handling
        post_response = requests.post(login_url, data=name_pwd, verify=False)
        # get token from login response structure
        auth = json.loads(post_response.text)
        login_attributes = auth['imdata'][0]['aaaLogin']['attributes']
        return login_attributes['token']
    def get_port_health(self, auth_token, switch):
        cookies = {}
        # set cookie
        cookies['APIC-cookie'] = auth_token
        # create health score URI
        request_url = '%s/mo/topology' % (self.base_url)
        if 'pod' in switch:
            request_url = '%s/pod-%s' % (request_url,switch['pod'])
        if 'node' in switch:
            request_url = '%s/node-%s' % (request_url,switch['node'])
        if 'slot' in switch:
            request_url = '%s/sys/ch/lcslot-%s' % (request_url,switch['slot'])
        if 'port' in switch:
            request_url = '%s/lc/leafport-%s' % (request_url,switch['port'])
        request_url = '%s/health.json' % (request_url)
        get_response = requests.get(request_url, cookies=cookies, verify=False)
        # return json data
        return get_response.json()


class OpenStackController(object):
    def __init__(self, os_ip):
        self.base_url='http://%s' % (os_ip)
    def login(self, name, pwd):
        login_url='%s:5000/v2.0/tokens/' % (self.base_url)
        name_pwd= {'auth': {'tenantName': 'admin', 'passwordCredentials': {'username': name,
'password': pwd}}}
        json_credentials = json.dumps(name_pwd)
        headers= {'content-type': 'application/json','accept':'application/json'}
        login_response = requests.post(login_url, data=json_credentials, headers=headers)
        auth = json.loads(login_response.text)
        token = auth['access']['token']['id']
        return token

    def post_sample(self,token,sample):
        post_headers={'X-Auth-Token':token,'content-type':'application/json','
            accept':'application/json'}
        json_data = json.dumps(sample)
        post_url = '%s:8777/v2/meters/healthscore' % (self.base_url)
        post_response = requests.post(post_url,data=json_data,headers=post_headers)
        return post_response.json()


def main():
    apic_info={'ip':IPADDRESS,'user':USER,'pwd':PASSWORD}
    switch= {'pod':'1','node':'101','slot':'1','port':'1'}
```

```
    os_info={'ip':IPADDRESS,'user':USER,'pwd':PASSWORD}
    apic= APICController(apic_info['ip'])
    token= apic.login(apic_info['user'],apic_info['pwd'])
    json_data= apic.get_port_health(token,switch)
    score= json_data["imdata"][0]["healthInst"]["attributes"]["twScore"]
    openstack= OpenStackController(os_info['ip'])
    token= openstack.login(os_info['user'],os_info['pwd'])
    sample= [{'counter_name': 'healthscore','counter_type': 'gauge',
              'counter_unit': '%','counter_volume': float(score),
              'project_id': '9f8e3513a1fd4d9e9c60201d8b30e4d6',
              'resource_id': 'osp5com01.noslab.com',
              'user_id':'4a5e1a2207924fe7b4803e2d32d2aaa1'}]
    response= openstack.post_sample(token,sample)

if __name__ == "__main__":
    main()
```