応募区分：研究型論文

Cisco ACI 連携による Microservice ネットワークサービスの充実と今後の課題

田村　勝（たむら　まさる）
ネットワンシステムズ株式会社
ビジネス推進本部 第２応用技術部

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
ビジネス推進本部 第２応用技術部

■ 要約

サービスを小さく分けたタスクを各ノードに分散させ、これらを協調させて一つのサービスとして提供するマイクロサービスというアーキテクチャが最近提案されている。マイクロサービスアーキテクチャを考えるに当たり、Mesos や Consul といったノード管理、リソース管理については検討されてきたが、ネットワークについてはフラットなネットワーク上に展開され、サービス間の通信やセキュリティといった面はあまり意識されていなかった。本論文ではマイクロサービスにおけるネットワークについての課題を取り上げ、さらに Cisco ACI を利用した課題解決を検討する。
最初に Cisco Microservice についての解説を行い、ACI との連携について検討し、OpenStack と ACI が連携している環境で Mesos を使用する場合の実現性と課題を述べる。次に、OpenStack 上の仮想マシン環境を Docker に置き換えた場合を想定し、ACI と Docker の連携について検証し、マイクロサービスインフラストラクチャを Nova-Docker 上に展開する場合の実現性と課題を述べる。

目次

1. はじめに

近年、ビジネスの変化が激しく、それに伴いインフラに対するワークロードの変化も多様化している。急激なワークロードの増加への対応、ワークロードが定常化している際のコスト効率化、最新テクノロジに追従するための頻繁なアップデートも含めた全体的なインフラの可用性の向上、これらの多様な要件にインフラが対応する必要がある。このような背景のもと、サービスを小さく分割し、タスクを各ノードに分散することで全体的なサービスを提供するマイクロサービスの考え方が登場している。このマイクロサービスを実現する基盤としてマイクロサービスインフラストラクチャがある。各ノードを束ねてコントロールし、適切なノードにタスクを割り振るソフトウェアとして、Apache Mesos や Hadoop に注目が集まっている。また、ノード自体を管理する仕組みとして、OpenStack があり、ACI と連携することにより、ネットワークを自動的に構成することができる。OpenStack でデプロイされた各インスタンス上で、Mesos から割り当てられたタスクを実行する必要があるが、実行時の必要なファイルなどを一括で展開し、簡単に実行できるように Docker を利用する。本論文では、現状のマイクロサービスの実装状況を解説し、マイクロサービスとして使用するために Mesos を取り上げその課題を述べたうえで、Cisco ACI を使った解決方法を検討する。そして、Micro Service のネットワークとして ACI を使う場合の考慮点、課題をまとめる。

2. マイクロサービスについて

一つのアプリケーションを小さなサービス単位に分けて、サービス間を協調して動作させるのがマイクロサービスである。これにより、疎結合されたインフラを実現し、柔軟な拡張・縮小、更なる可用性の向上を構成することが可能となる。このマイクロサービスを支えるインフラとしてマイクロサービスインフラストラクチャの考え方が出てきた。従来のモノリシック(全部入り)のアプリケーションであれば、多くても 3 tier の考え方で 3 台程度のサーバーが必要とされこれは、仮想マシンスタイルでの管理で十分だった。しかしながら、マイクロサービスを前提とすると、アプリケーションが細かく分割され、従来の管理方法では管理出来ない。これを管理するのが新しいマイクロサービスインフラストラクチャとなる。サービスを小さな単位に分割すると、サービス単位で容易に拡張と縮小が可能となり、システム全体の拡縮も自由に出来るようになり、マイクロサービスインフラストラクチャはこの管理も行う。マイクロサービスインフラストラクチャに必要とされるコンポーネントはノード管理、サービス管理、リソース管理の 3 つである。

2.1. Cisco Microservice

Cisco Microservice Infrastructure は Cisco によって開発されているマイクロサービス用のインフラである。Ansible の Playbook の集合体で構成されており、指定のクラウド上にマイクロサービスインフラストラクチャを構築する事が可能となっている。対応するクラウドは、OpenStack/Cisco Cloud Service/AWS/Google Compute Engine/Digital Ocean と多岐にわたっている。構築されるインフラは

仮想マシンをベースとして複数台で構成される。仮想インスタンスの内部には Mesos と Marathon がインストールされ、各仮想マシンのリソース管理が行われる。Consul を利用して、各 Datacenter 間で協調して動作が行われ、サービス管理を実現し、実際のアプリケーションは各ノード内の Docker 内部で動作する。

2.2. 今回使用するコンポーネント

今回は、Cisco Microservice を基本として、マイクロサービスについて説明、検証をする。Cisco Microservice ではノード管理に OpenStack/Cisco Cloud Service/AWS/Google Compute Engine/Digital Ocean 、サービス管理に Consul、リソース管理に Mesos を使用している。本論文ではノード管理に OpenStack を選択した。また、サービス管理とリソース管理については Cisco Microservice と同様に、それぞれ Consul と Mesos を使用した。これらコンポーネントの詳細を以下に説明する。

2.3. 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の順に名前が付けられており、今年春にリリースされた最新バージョンは Kilo 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、Liberty となる予定である。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 周り、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。今回は OpenStack のプロジェクトの中でも Neutron/Nova が関係する機能について言及する。

2.4. Consul/Terraform について

Consul と Terraform は Hashi corp が主に開発しているオープンソースソフトウェアである [1]。Consul は各ノードで提供されるサービスの検知、監視機能を備えたソフトウェアとなり、各ノードで動くサービスを監視し、その結果を KVS として提供することができる。アプリケーションは KVS にアクセスし、現在のノードの状態を確認することで、新しいノードを検知し、適切にノードに処理を振り分けることができる。DNS のインターフェイスも備えており、サービスごとに名前解決をすると、稼働しているノードの IP アドレスを返すこともできる。Consul はマスター、スレーブ構成となっており、サービスを提供しているノード上でスレーブを動作させ、マスターに対して状態を報告している。Terraform は、アプリケーション基盤を安全、効率的に構築、変更、バージョン管理をするためのソフトウェアである [1]。デプロイ先は AWS, DigitalOcean, GCE, OpenStack などをサポー

トしており、これらのクラウド上にインスタンスの作成や、ネットワークの作成をすることができる。これらデプロイ先で行う操作や値を設定ファイルとして記述し、実行することで、構成の自動化と再現性をインフラにもたらすことができる。

2.5. Mesos/Marathon について

Apache Mesos は The Apache Software Foundation の元で開発が進められているオープンソースのクラウド管理ソフトウェアである。通常の OS がプロセスを CPU に割り振るように、Mesos は Mesos-master と呼ばれるマスターがタスクを配下の Mesos-slave に割り振り、スレーブ上でタスクを実行する。Mesos はクラスタ内の各ノードの CPU、メモリ、ストレージなどのリソースを抽象化し、効率よくタスクを分散できるようになっている。 [2]

各ノードの CPU やメモリの状況により、タスクが動的に各スレーブに割り当てられるため、各ノードは割り当てられる可能性のあるどのタスクでも実行できるように準備しておく必要がある。この準備の中には、各タスクが参照するライブラリや、データファイルも当てはまる。そこで、各タスクをインスタンス化し、タスクが割り当てられたタイミングでそのインスタンスを実行するという方法が考えられる。Docker はタスクごとにコンテナとして用意することが容易であり、さらに仮想マシンとしてタスクを準備する方法と比べ、様々な観点でオーバーヘッドが非常に小さい。そのため Docker と連携することでスレーブの準備を大幅に簡易化することができる。具体的にはタスクの実行をコンテナの実行とみなし、タスクを割り振られたノードは Docker のレポジトリから指定されたコンテナイメージを pull し、そのコンテナを実行する。

しかしながら、Apache Mesos はスケジューリングの機能を提供しているに過ぎないため、タスクの永続化に必要となるような、サービスの停止や監視の機能を具備していない。Marathon はクラスタ内で使用する、init プログラムの役割をなすソフトウェアとなり、Mesos をバックエンドとして使用し、各サービスの開始や停止、さらに各サービスの多重度を変更してスケールアウト/スケールダウンをさせることができる。本論文では、このようなサービス監視に Marathon を使用する。

2.6. Docker について

Docker は Docker 社が開発している、オープンソースのコンテナテクノロジである。カーネルを共有し、ネットワークやプロセス空間をネームスペースで他のコンテナと隔離する。

カーネルを複数のゲストで共有するため、ゲストがそれぞれの仮想ハードウェアやカーネルを持つ仮想マシン型と比べ、オーバーヘッドが小さくなる。仮想マシンに必要なカーネルやドライバなどもホストが提供しているものを共有するため、ディスクイメージに含める必要がなく、インスタンスのディスク容量も少なくて済む。

上記のようにコンテナテクノロジはオーバーヘッドが非常に小さく、仮想マシン型のように従来であれば仮想マシン内部で複数稼働する必要があった各サービスを、コンテナを使用することで、サービスごとに独立させ稼働することが可能である。そのため、コンテナ単位で一つのサービスを動かし、複数のコンテナを集めてアプリケーションを構成するマイクロサービス型との親和性が高い。

また Docker hub と呼ばれるリポジトリも用意されており、利用者が作成したコンテナイメージをリ

ポジトリにアップロードすれば、Docker コンテナを動作する環境として、どこからでもコンテナイメージをダウンロードして動かすことができる。
Docker で使用できるネットワークとしては、ホスト上で独立したネットワークを作成し、アドレス変換を使用して外部にサービス提供する方法や、ブリッジを使用して外部とコンテナが直接通信する方法がある。しかし、コンテナごとに独立したネットワークに接続しマルチテナント環境で使用する環境は未だ一般的では無い。

2.7. Cisco ACI について

ACI(Application Centric Infrastructure)は Cisco Systems が開発した、ネットワークファブリックのソリューションであり、特徴としては、ネットワークをアプリケーションの視点から設計することができる。アプリケーション内の各コンポーネントをグループ化した EPG(End Point Group)と、そのグループ間のポリシーを定義したコントラクトからなるアプリケーションプロファイルを作成することでネットワークを設計する。ACI は APIC(Application Policy Infrastructure Controller)と呼ばれるコントローラと、データプレーンでパケットを転送する Nexus9000 シリーズのスイッチから構成されている。ACI への設定はコントローラで行うが、すべての設定は API が用意されているため、ネットワークの自動化が非常に行いやすいことも特徴のひとつである。また、ファブリックは自律して動作しており、一度設定が反映された後は、例えすべてのコントローラがダウンしてもファブリックとしての動作を継続することができる。

3. マイクロサービスと ACI との連携

Mesos/Marathon/Consul/Terraform/Docker を利用したマイクロサービスインフラストラクチャの実現を考慮する際に、ネットワークの機能がまだ十分では無いと考えられる。そのため、補完的にマイクロサービスインフラストラクチャに対して、ACI を組み合わせる事でアプリケーションを考慮した基盤として動作させる検討を行う。

3.1. Mesos とネットワークについて

Mesos、Docker、Marathon を組み合わせて構成する場合、2 つのネットワークの接続方法から選択できる。1 つ目は Mesos-slave に割り当てられた IP アドレスを共有する方法である。この方法では外部から各コンテナに直接アクセスすることはできない。2 つ目は、Mesos-slave 内に bridge を作り、各コンテナが独立した IP アドレスを持つ方法である。この方法の場合、各コンテナが IP アドレスを持っているため、前述した Consul の DNS と組み合わせて利用しやすいという利点がある。どちらの方法も現在はフラットなネットワークが必要とされており、タスク間のセキュリティなどは考慮されていない。そのため、同じクラスタにデプロイされたタスクはお互いに通信ができる状態となっており、アプリケーションによっては不都合が生まれる可能性もある。ACI は同様の属性を持つエンドポイントをグループ化し、グループ間のセキュリティを定義することができる。そこで今回は Mesos のネットワークを ACI と連携させることで Mesos クラスタのネットワークセキュリティを検討する。

3.2. ACI と Mesos

現在 ACI と Mesos が直接的に連携するソリューションはない。そこで、今回は簡易的なソリューションとし、用途ごとにクラスタを複数デプロイする方法を検討した。同一クラスタ内のタスクはお互いに通信できてしまうことは先に説明したが、この方法は、通信させたくないタスクを別のクラスタにデプロイする。これは各クラスタをそれぞれの EPG に割り当てることで実現できる。

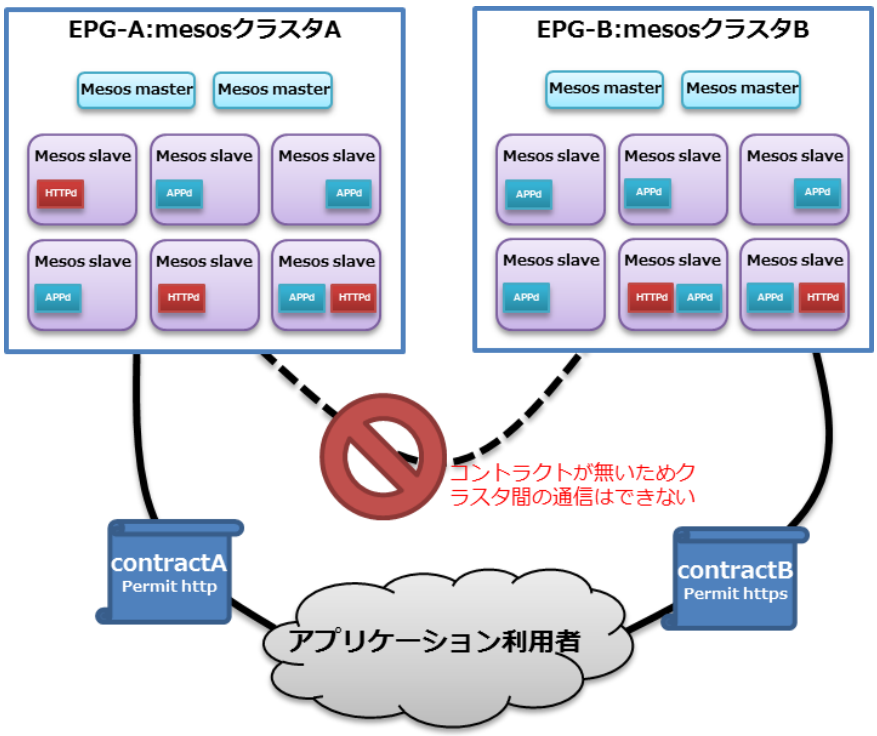

図 1 クラスタレベルでの隔離

3.3. OpenStack と ACI の連携について

ACI の L2 転送機能と L3 ルーティング機能を OpenStack から使用することができるように、Neutron ML2 プラグインが用意されている。プラグインを使用することにより、ACI に用意されている API を使用し ACI 上のネットワークの設定を OpenStack から行うことができる。OpenStack 上のネットワークが ACI での EPG に、OpenStack 上のルーターが ACI 上のコントラクトに相当する。OpenStack の各コンピュートノードは LLDP を使用することで ACI の leaf スイッチとの接続を認識するため、Access policy などの設定も自動で行われる。OpenStack においてインスタンスが作成されると、ACI 上ではそのインスタンスが稼働しているホストが接続されているポートに Static Binding（path）を設定し、仮想マシンと EPG の紐付けを自動的に実施している。

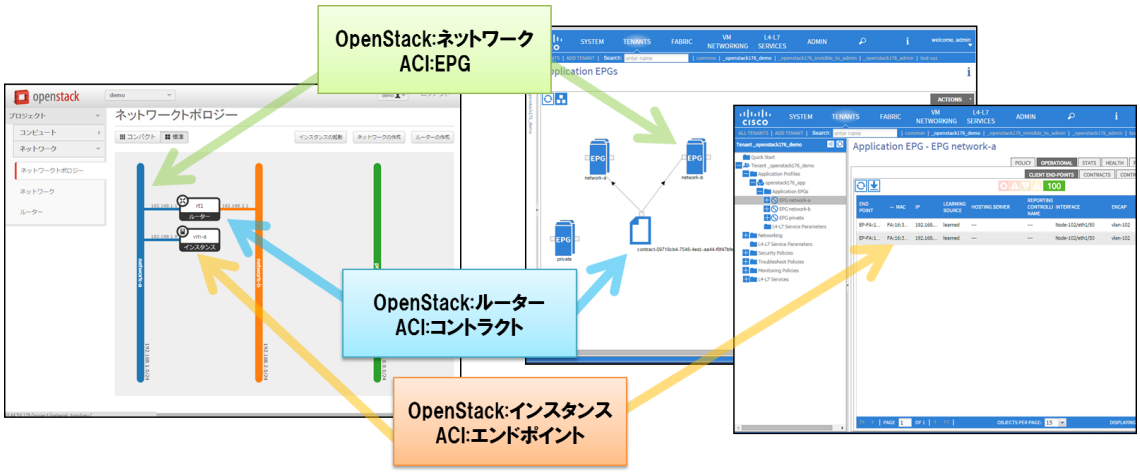

図 2ACI と OpenStack のオブジェクトの対応

注意すべき点としては、ACI が NAT に対応していないため、floating IP を使用することが出来ない点である。また、Nova の metadata proxy の機能も利用できない。また、マイクロサービスの観点から、サービスのレイヤーでは Consul などを使用することで連携するノードを検知することができるが、Consul のマスターに対して、追加インスタンスの登録を実施する場合などは、あらかじめインスタンスのイメージに IP アドレスを埋め込むなどの操作が必要である。尚、GBP に対応したプラグインを使うことにより、ACI 上のコントラクトを OpenStack から詳細に設定することが可能となるため、より柔軟な連携も可能となる。

3.4. 検証環境でのテスト

3.4.1. 検証の概要

ここでは先で提案した、クラスタ構成を用途別に分けることで Mesos クラスタ構成に対して、ある程度のセキュリティを付与する方法が実現できることを示すため、ACI と Mesos、そして OpenStack の連携が動作することを確認する。今回はコントローラとして cont.toshin.lab 、またコンピュートノードとして com01.toshin.lab 、com02.toshin.lab の 2 台を用意した。本章では OpenStack 環境上にコンテナとして Mesos クラスタを構築し、同様の構成を作成した場合の考慮点を述べる。

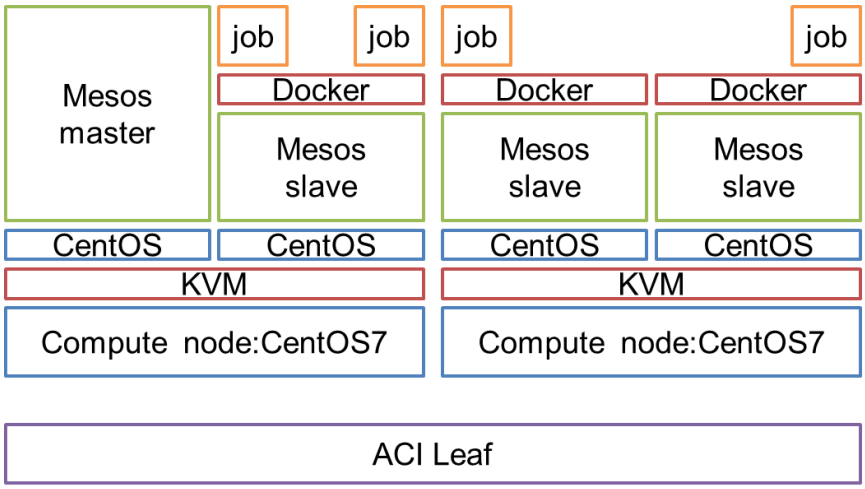

図 3 検討構成

3.4.2. クラスタ構成のデプロイ

Mesos を使用したクラスタ構成を作成するためには、最小で Mesos-master と Mesos-slave を稼働する各インスタンスが必要になる。これらのインスタンスを OpenStack 上にデプロイするために Terraform を使用した。ネットワークに関しては、クラスタごとに L2 のフラットなネットワークをデプロイする。Terraform は OpenStack Neutron と連携するため Neutron 経由で ACI 上にネットワークを構築する。Terraform を使用することで、クラスタ専用のネットワークの作成から、クラスタで使用する IP アドレスの定義、複数のインスタンスのデプロイまでを記述しておき、再現可能な状態で環境を構築することができる。

3.4.3. 考察

OpenStack と ACI を連携させ、Mesos クラスタを ACI の EPG にマッピングさせることで各 Mesos クラスタが別々のネットワーク上にデプロイされ、それぞれ隔離できることが確認できた。しかしこの方法でもまだ、同一クラスタ内に存在する、別々の役割を提供するタスク間の疎通が自由にとれるような状態である。たとえば 3tier のアプリケーションをデプロイする場合、各 Tier 間で通信を制限したいが、今回の方法では制限できない。そのため Mesos のネットワークセキュリティについては引き続き検討が必要である。

また、ACI と連携した場合 Metadata proxy を使用することができない。そのため、今回は Terraform の Provisioner 機能を使用して Slave のデプロイ後に Master の IP アドレスを通知するという方法を取った。ただし、この方法では Terraform と、デプロイしたクラスタ環境が直接通信できる必要がある。

クラスタを複数作成する場合、それぞれに Mesos-master が作成される。また、slave についても集約度が下がってしまう。そのため、少しでもオーバーヘッドを少なくしリソースを効率的に使用する方法が求められる。Metadata-proxy が使えないことによるデプロイ時の課題と、パフォーマンスの課題については次の章で再度検討する。

4. OpenStack を利用した Cisco ACI とコンテナ連携

4.1. 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に対応している。OpenStack ではインスタンスの作成や管理に伴う操作は Nova が行う。そのため各種ハイパーバイザのインターフェイスを Nova が標準化していると考えることができる。また、Nova はインスタンスの作成先をハイパーバイザ上に制限しているわけではない。たとえば Docker のインスタンスを管理するためのプロジェクトも公開されている。この場合も Nova が Docker のインターフェイスを標準化しているため、Horizon や Neutron などのコンポーネントはインスタンスの作成先が Docker であるか、KVM であるかを意識しない。
本章ではコンテナと ACI の連携のために Nova-Docker を利用することにした。

4.2. Nova-Docker について

Nova-Docker は OpenStack Nova 向けの Docker ドライバである [3]。コンピュートノードにセットアップされた Docker デーモンと連携し、OpenStack のインスタンスを Docker 上にコンテナとしてデプロイできる。ストレージとしては Glance を使用するため、Docker コンテナイメージを一度 glance へ登録する必要がある。Nova の metadata は Docker コンテナ起動時に環境変数としてコンテナに渡すことができる。
ネットワークについては Neutron に対応するため、KVM などの上にデプロイされた仮想マシンと同様に Neutron で提供されるネットワークを使用することができる。

4.3. Nova-Docker と ACI

Nova-Docker でデプロイされたインスタンスは Neutron で管理されるネットワークに接続することができる。そのため、Neutron ML2 plugin を使用して ACI 連携をしたネットワークを使用できる。つまり、現在 Docker が直接 ACI と連携するパスは存在していないが、OpenStack と連携することにより、Docker 上のインスタンスが、ACI の提供するハードウェア転送による低遅延、広帯域ネットワークやルーティングの機能を使用することができる。

4.4. 検証環境でのテスト

4.4.1. 検証概要

今回はコントローラとして cont.toshin.lab 、またコンピュートノードとして com01.toshin.lab 、com02.toshin.lab の 2 台を用意した。それぞれのノードでは CentOS7 上で DevStack を使用し、OpenStack、Nova-Docker、apicapi、APIC 連携用のプラグインが含まれる networking-cisco を github 上の master ブランチをダウンロードして使用している。ACI は 1.1(1j)を使用した。尚、Nova-Docker については instance の起動時にコンテナに対して特権を付与することができない [4]。本環境では、Mesos ノードを Nova-Docker からコンテナとして稼働し、その中で更にアプリケーションの稼働用途としてコンテナを稼働させる。そのため、Docker in Docker を使用するが、その場合、特権が必要

になるため今回は Nova-Docker が使用するライブラリを修正し、常に起動するコンテナに特権を付与するようにして検証した。また、デフォルトである、iptables を使用したセキュリティグループの場合、インスタンスからの通信は Nova が割り当てた MAC/IP アドレスを使用する必要がある。今回 Docker in Docker を bridge モードで動作させるために、NoopFirewallDriver を使用した [5]。

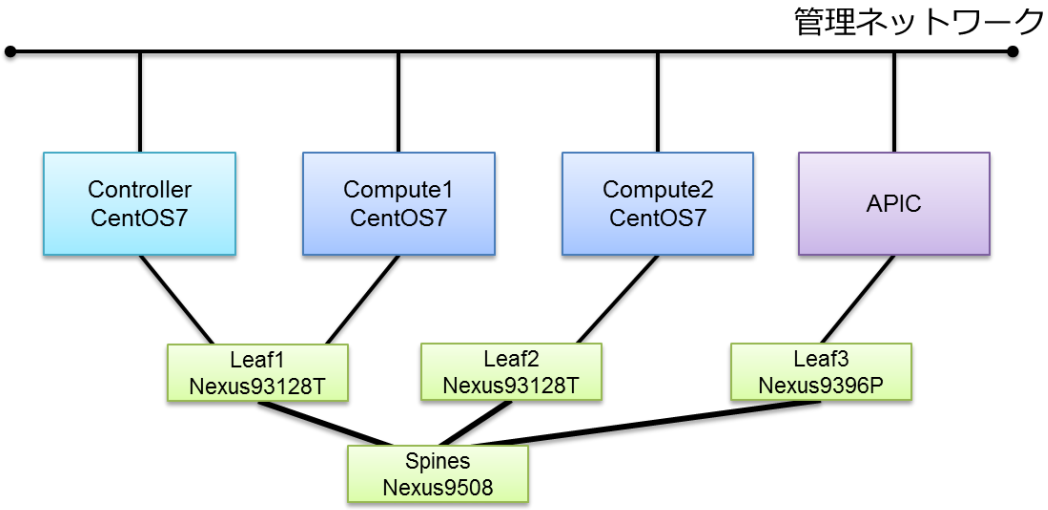

図 4 Nova-Docker-ACI 連携の検証構成

4.4.2. ACI 連携の確認

本環境でコンテナを各コンピュートノードに３つずつデプロイし、それぞれのコンテナを別のネットワークにデプロイした場合に隔離されているか、また、同じネットワークにデプロイした場合に通信ができるかを確認した。その結果、特に問題が見られなかった。

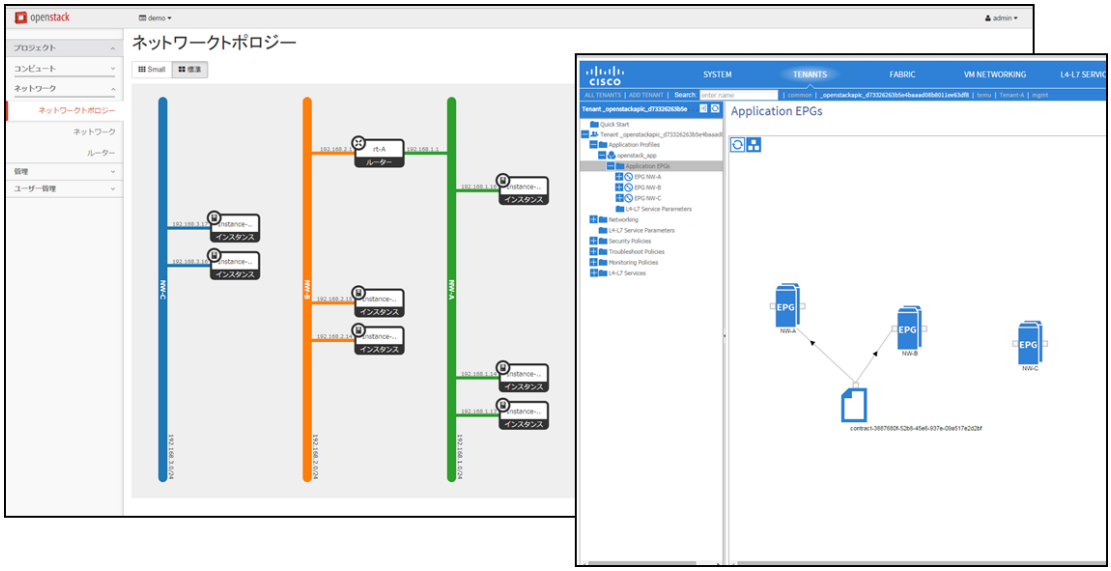

図 5 検証構成

表 1 Nova-Docker ACI 連携の通信確認

通信内容		期待	結果	備考
同一ホスト内 別セグメント (router 接続なし)	Instance-A(com1,NW-A) Instance-B(com1,NW-C)	×	×	EPG 間に Contract が無いため通信不可
同一ホスト内 別セグメント (router 接続あり)	Instance-A(com1,NW-A) Instance-C(com1,NW-B)	○	○	ACI Leaf 折り返し
同一ホスト内 同一セグメント	Instance-A(com1,NW-A) Instance-D(com1,NW-A)	○	○	ホスト内折り返し
同一ホスト内 別セグメント (router 接続なし)	Instance-A(com1,NW-A) Instance-E(com2,NW-C)	×	×	EPG 間に Contract が無いため通信不可
同一ホスト内 別セグメント (router 接続あり)	Instance-A(com1,NW-A) Instance-F(com2,NW-B)	○	○	ACI Leaf 折り返し
同一ホスト内 同一セグメント	Instance-A(com1,NW-A) Instance-G(com2,NW-A)	○	○	ACI Leaf 折り返し

4.4.3. Docker in Docker 環境での ACI 連携

次に Nova-Docker 上で Docker を起動できることを確認した。前述したライブラリを修正したことにより、Nova でデプロイされたコンテナに特権が付与されていることが確認できた。

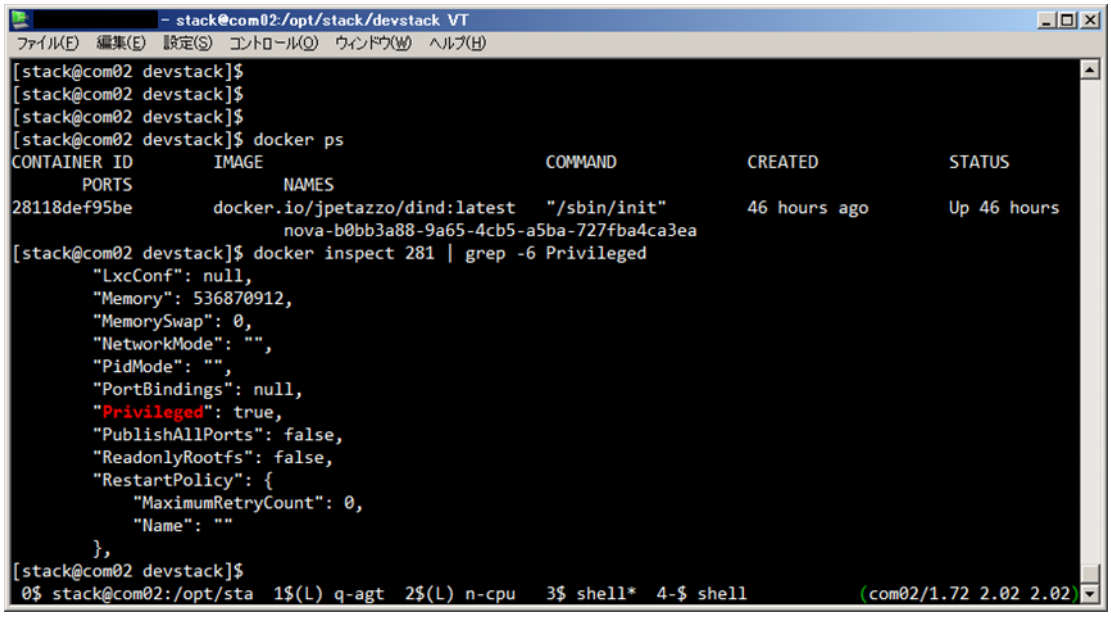

図 6 特権プロパティの確認

また、Nova-Docker でデプロイされたインスタンスの中で Docker を起動し、ネストされたコンテナを起動することができることを確認した。また、NoopFirewallDriver を使用することによりインスタンス内に用意した Linux bridge を使用しネストされたコンテナが ACI を通過して通信ができることを確認した。本構成での各論理インターフェイス間の接続を図 7 に示す

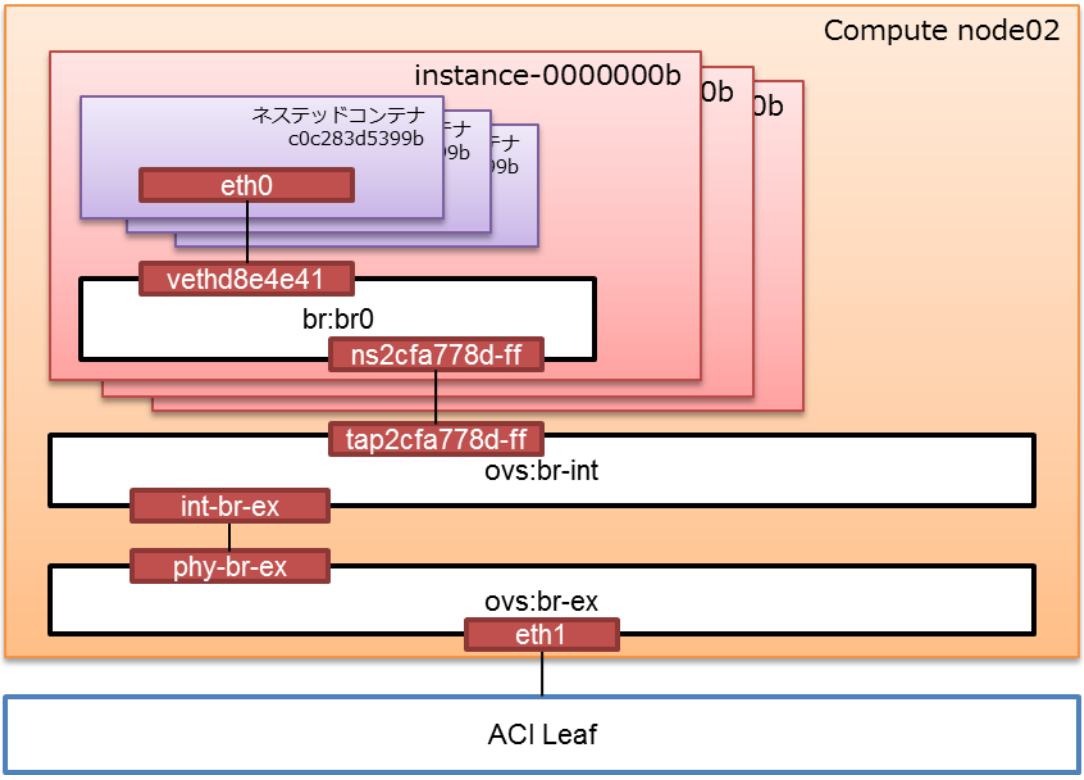

図 7 仮想インターフェイス間の接続関係

4.4.4. OpenStack Nova-Docker の使用

3 章では、OpenStack Nova のドライバとして KVM のドライバを使用した。しかし KVM はハイパーバイザ型のため、コンテナ型と比べオーバーヘッドが大きく、また本環境では ACI 連携をしているため metadata-proxy を使用できないという課題があった。ここでは KVM の代わりに Docker を使用するが、Mesos-slave は起動時に Mesos-master の IP アドレスが必要になる。Nova-Docker は起動時に meta 情報をインスタンスに環境変数として渡すことができるため、手動設定を排除し、自動的に設定することが可能となる。今回は Terraform で instance の作成時に得られる戻り値の中の IP アドレスを使用し、事前に Mesos-master を作成することで、その IP アドレスを Mesos-slave に渡した。

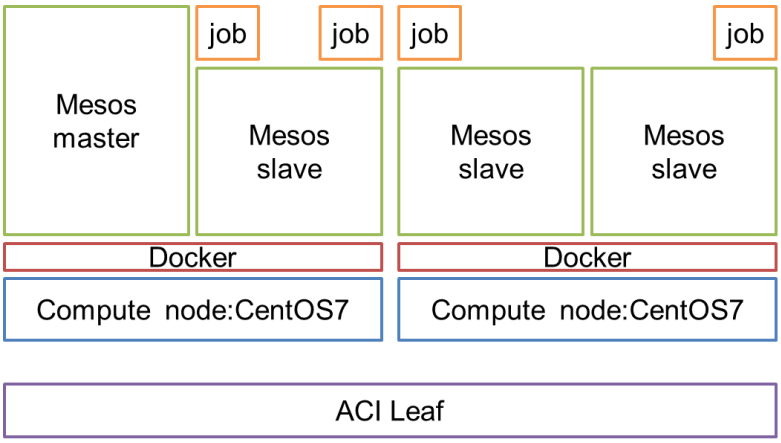

図 8 Nova-Docker を使用した場合の stack 構成

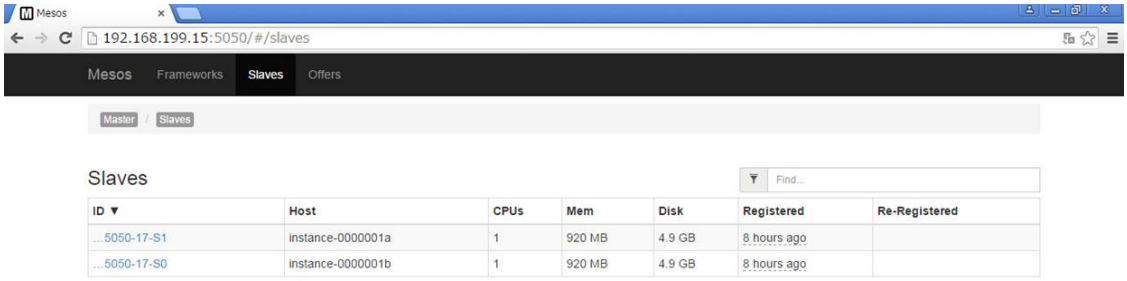

図 9 Mesos-slave が自動的に登録される

4.4.5. 考察

OpenStack Nova-Docker を使用することで、コンテナ環境でのネットワークを OpenStack API を介して、ACI にオフロードできる。特に、既存の Docker のネットワーク環境は NAT により、ホストの IP アドレスでコンテナにアクセスするか、もしくは bridge を作成し、ホストごとにネットワークを管理する必要があった。このような状況に対し、Ambassador パターンなどを使用し異なるホスト上のコンテナ間を接続する方法 [6]も考えられているが、すでに広く使用されている Neutron と組み合わせ、さらに ACI によるハードウェアスイッチングを提供できる Nova-Docker のほうが簡単であり、利用しやすい。そのため、コンテナ環境におけるネットワーク制御の方策として、Nova-Docker を基盤としたインフラを実現すべく、Nova-Docker 上に nested 環境も作成したが、こちらにはいくつか課題が存在する。

1 つめは Nova-Docker の実装についてである。今回、nested 環境を作成するために Docker コンテナの起動時に Privileged フラグを設定する必要があったが、このフラグを Nova から設定することができない。この件については 2 年ほど前から指摘されているがまだ修正されていない [4]。今回の検証では前述したようにすべてのインスタンスの作成時に該当のフラグを設定するように修正したが、この方法では必要のないインスタンスにまで設定するためセキュリティ上好ましくない。やはり Nova

インスタンス作成時に選択できる形の実装が待たれる。

もう一点はOpenStackのセキュリティグループについてである。今回nestedされたコンテナからの通信は、インスタンス内のbridgeを経由し、nestedインスタンスのMAC/IPで外部と通信される。しかしデフォルトのポリシーでは通信ができないことをすでに述べた。そこで今回はNoopFirewallDriverを利用したが、こちらはすべてのセキュリティグループが無効になってしまうため、セキュリティ上のリスクが大きい。KiloリリースからML2/Open vSwitch port-securityがサポートされた[7]。この機能を使うことにより、特定のポートのみ設定を外すことができるようにできるため、こちらを使用すべきであろう。

また、Nova-Dockerを使用する利点の一つとしてインスタンスへ外部から情報を渡すことができるため、metadata-proxyを使用する必要がなくなるという点があげられる。ACI OpenStack連携の課題として、metadata proxyを使用することができないということが議論されていたが、Nova-Dockerの場合、Dockerインスタンスに環境変数を渡すため、今回のようなインスタンス間の連携を起動時に自動的に組み上げるような場合に特に有用となる。

本章ではACI + Nova-Docker連携したプライベートクラウド上でMesosクラスタをデプロイし、クラスタ環境を容易に構成することができることを示した。今後、MesosとDockerを連携させることで、より容易にアプリケーションをデプロイすることができるようになると考えられる。また、MarathonとConsulを加えることでアプリケーションの拡張のためのインスタンスの増減も容易に行うことができると考えられる。

5. まとめ

今回はMesosを使用したマイクロサービス環境を説明し、今まで抜け落ちていたネットワークについて検討を行い、マルチクラスタ構成であれば、ACIと連携し、クラスタ間のセキュリティを保つことができると確認できた。しかし、クラスタ内のセキュリティまで考慮することはできず、課題があることを示した。
また、マイクロサービスデプロイを行うためのインフラとして、ACI、OpenStack Nova-Docker連携が有効なことを示した。特に、KVMなど他のハイパーバイザ型のNova driverを使用している場合、ACI連携をしているとmetadata proxyが使用できないため、インスタンス間の連携が困難となるがNova-Dockerの場合、環境変数としてインスタンスにオプションを簡単に渡すことができるのは大きな利点である。
今後大規模アプリケーションの設計思想としてのMicroservice Architectureの利用や、モノのインターネットから収集された大規模なデータの解析プラットフォームとして、Mesosクラスタが選択されることが増えてくると考えられる。今回検討した内容がACIと組み合わせて使用し、マイクロサービスインフラストラクチャの実現の手掛かりとなることを期待する。

6. 引用文献

1. Introduction - Terraform by HashiCorp. (オンライン) https://terraform.io/intro/index.html.
2. Apache Mesos. (オンライン) http://mesos.apache.org/.
3. stackforge/nova-docker. (オンライン) https://github.com/stackforge/nova-docker.
4. Docker Driver Support To Enable Privileged Mode：Blueprints：OpenStack Compute (nova). (オンライン) https://blueprints.launchpad.net/nova/+spec/docker-privileged-mode-support.
5. Howto disable OpenStack firewalls. (オンライン) https://gist.github.com/djoreilly/db9c2d32a473c6643551.
6. マルチホスト Docker ネットワーキング（１）. (オンライン) http://tech-sketch.jp/2015/04/multi-host-docker-1.html.
7. What’s Coming in OpenStack Networking for the Kilo Release | Red Hat Stack. (オンライン) http://redhatstackblog.redhat.com/2015/05/11/whats-coming-in-openstack-networking-for-the-kilo-release/.

7. 付録

7.1. Mesos クラスタデプロイ用 terraform ファイル

```
provider "openstack" {
    user_name  = "admin"
    tenant_name = "demo"
    password  = "password"
    auth_url  = "http://CONTROLLER-ADDRESS:5000/v2.0"
}

resource "openstack_networking_network_v2" "mesos-cluster" {
  name = "mesos-cluster"
  admin_state_up = "true"
  region = "RegionOne"
}

resource "openstack_networking_subnet_v2" "subnet_1" {
  network_id = "${openstack_networking_network_v2.mesos-cluster.id}"
  cidr = "192.168.198.0/24"
  ip_version = 4
}

resource "openstack_networking_router_v2" "rt1" {
  region = "RegionOne"
  name = "rt1"
}

resource "openstack_networking_router_interface_v2" "router_interface_1" {
  region = "RegionOne"
  router_id = "${openstack_networking_router_v2.rt1.id}"
  subnet_id = "${openstack_networking_subnet_v2.subnet_1.id}"
}

resource "openstack_compute_instance_v2" "messos-master" {
  name = "master"
```

```
  image_id ="518fceda-02bd-4ae6-b82e-6d37c64fcaa0"
  flavor_id = "2"
  security_groups = ["default"]
  region = "RegionOne"
  network {
    uuid = "${openstack_networking_network_v2.mesos-cluster.id}"
  }
}

resource "openstack_compute_instance_v2" "messos-slave" {
  name = "slave"
  image_id ="2ac160fe-d613-4b3f-b03f-bfc0d9d0cfe5"
  flavor_id = "2"
  security_groups = ["default"]
  region = "RegionOne"
  count = 2
  network {
    uuid = "${openstack_networking_network_v2.mesos-cluster.id}"
  }
  provisioner "remote-exec" {
    connection {
      user = "centos"
      key_file = "~/.ssh/mesos.pem.dec"
    }
    inline = [
      "sudo bash start_mesos.sh ${openstack_compute_instance_v2.messos-master.access_ip_v4}"
    ]
  }
}
```

7.2. Nova-Docker 使用時の Mesos クラスタデプロイ用 terraform ファイル

```
provider "openstack" {
    user_name  = "admin"
    tenant_name = "demo"
    password  = "password"
    auth_url  = "http://CONTROLLER-ADDRESS:5000/v2.0"
}

resource "openstack_networking_network_v2" "mesos-cluster" {
  name = "mesos-cluster"
  admin_state_up = "true"
  region = "RegionOne"
}

resource "openstack_networking_subnet_v2" "subnet_1" {
  network_id = "${openstack_networking_network_v2.mesos-cluster.id}"
  cidr = "192.168.199.0/24"
  ip_version = 4
}

resource "openstack_networking_router_v2" "rt1" {
  region = "RegionOne"
  name = "rt1"
}

resource "openstack_networking_router_interface_v2" "router_interface_1" {
  region = "RegionOne"
  router_id = "${openstack_networking_router_v2.rt1.id}"
  subnet_id = "${openstack_networking_subnet_v2.subnet_1.id}"
}

resource "openstack_compute_instance_v2" "messos-master" {
  name = "messos-master"
  image_id ="443befb6-9d0b-409c-9387-5753538fc170"
  flavor_id = "34af1383-3836-4815-8060-503bc5d1b61b"
```

```
  security_groups = ["default"]
  region = "RegionOne"
  network {
    uuid = "${openstack_networking_network_v2.mesos-cluster.id}"
  }
}

resource "openstack_compute_instance_v2" "messos-slave" {
  name = "messos-slave"
  image_id ="af8a06dc-c357-4bad-8290-b0d8f8ab1c07"
  flavor_id = "34af1383-3836-4815-8060-503bc5d1b61b"
  security_groups = ["default"]
  region = "RegionOne"
  count = 3
  network {
    uuid = "${openstack_networking_network_v2.mesos-cluster.id}"
  }
  metadata {
    master = "${openstack_compute_instance_v2.messos-master.access_ip_v4}"
  }
}
```