はじめに

DockerSwarmユーザガイド～基礎編

このガイドについて

Docker ドキュメント（https://docs.docker.com/）の日本語翻訳版（http://docs.docker.jp/）をもとに、Docker Swarm に関連する箇所を電子データ化しました。

免責事項

現時点ではベータ版であり、内容に関しては無保証です。PDF の評価用として公開しました。Docker は活発な開発と改善が続いています。同様に、ドキュメントもオリジナルですら日々書き換えが進んでいますので、あらかじめご了承願います。

このドキュメントに関する問題や翻訳に対するご意見がありましたら、GitHub リポジトリ上の Issue までご連絡いただくか、pull request をお願いいたします。

- https://github.com/zembutsu/docs.docker.jp

履歴

- 2016 年 6 月 2 日　Swarm ユーザガイド v1.1 beta1 を公開

目次

1章

DockerSwarm概要

1.1 概要

Docker Swarm（スウォーム）は Docker に対応するネイティブなクラスタリング用ツールです。Docker Swarm は標準 Docker API で操作できます。そのため、Docker ホスト群を集め、1 つの仮想 Docker ホストとして扱えます。既に Docker デーモンと通信可能なツールであれば、Swarm を使うだけで、意識せずに複数のホストにスケール可能になります。以下のツールをサポートしていますが、これだけに限りません。

- Dokku
- Docker Compose
- Krane
- Jenkins

そしてもちろん、Swarm は Docker クライアントでの操作もサポートします。

他の Docker プロジェクトと同様に、Docker Swarm は“ swap, plug, and play ”（交換して、取り付けて、実行）の原理に従います。開発初期においては、API はバックエンドを取り替え可能（pluggable（プラガブル））となるよう開発する方針に落ち着きました。つまり、Docker Swarm が利用するスケジューリングのバックエンドとは、任意に置き換え可能であるのを意味します。Swarm の取り替え可能な設計により、多くの場面でスムーズかつ独創的な体験を提供します。また、Mesos（メソス）のような、より強力なバックエンドに取り替えれば、大規模なプロダクション（本番環境）へのデプロイも可能となります。

1.1.1 Swarm クラスタ作成を理解

自分のネットワーク上で **Swarm クラスタ**（訳注；” Swarm ”=群れ、の意味）を形成するには、まず Docker Swarm イメージを取得します。それから Docker で Swarm マネージャを設定し、Docker Swarm を実行したい全てのノードを設定します。この作業には以下の手順が必要です。

- Swarm マネージャと各々のノードと通信ができるよう TCP ポートを開く
- 各々のノードに Docker をインストールする
- クラスタを安全にするため、TLS 証明書を作成・管理する

管理者が経験を積み始めるため、または、プログラマが Docker Swarm に貢献し始めるためには、手動でのインストールが最適な方法です。あるいは `docker-machine` を使って Swarm をインストールする方法もあります。

Docker Machine を使えば、Docker Swarm をクラウド・プロバイダや自分のデータセンタに素早くインストールできます。ローカルのマシン上に VirtualBox をインストールしていれば、ローカル環境上で Docker Swarm を素早く構築し、試すことができます。Docker Machine はクラスタを安全にするために、証明書を自動生成します。

初めて Docker Swarm を使うのであれば、Docker Machine を使う方法が一番です。推奨する方法で進めるには「Swarm を検証環境で試すには」のセクションをお読みください。

手動でのインストールや開発に対する貢献に興味があれば、「プロダクション用の Swarm クラスタ構築」のセクションをご覧ください。

1.1.2 ディスカバリ・サービス

コンテナ内のサービスを動的に設定・管理するには、Docker Swarm とディスカバリ用のバックエンドを使います。利用可能なバックエンドに関する情報は、「ディスカバリ・サービス」のセクションをお読みください。

1.1.3 高度なスケジューリング

高度なスケジューリングについては、「strategies（ストラテジ）」と「filters（フィルタ）」の各セクションをお読みください。

1.1.4 Swarm API

Docker Swarm API は Docker リモート API と互換性があります。そのため、新しいエンドポイントの追加時には、Swarm も同時にエンドポイントを拡張します。

1.1.5 ヘルプを得るには

Docker Swarm はまだ開発途上であり、活発に開発中です。ヘルプが必要な場合、貢献したい場合、あるいはプロジェクトの同志と対話したい場合のため、私たちは多くのコミュニケーションのためのチャンネルを開いています。

- バグ報告や機能リクエストは、GitHub の issue トラッカー[*1] をご利用ください。
- プロジェクトのメンバとリアルタイムに会話したければ、IRC の`#docker-swarm` チャンネルにご参加ください。
- コードやドキュメントの変更に貢献したい場合は、GitHub でプルリクエスト[*2] をお送りください。

より詳細な情報やリソースについては、私たちのヘルプ用ページ[*3] をご覧ください。

*1 https://github.com/docker/swarm/issues

*2 https://github.com/docker/swarm/pulls

*3 https://docs.docker.com/project/get-help/

2章

セットアップ・導入ガイド

2.1 Docker Swarm の入手方法

Docker Swarm クラスタを作成する方法は、swarm が実行可能なイメージをコンテナとして使うか、あるいは、実行可能な swarm バイナリをシステム上にインストールする方法があります。このページは２つの方法を紹介し、それぞれの賛否を議論します。

2.1.1 インタラクティブなコンテナでクラスタを作成

クラスタの作成には、Docker Swarm 公式イメージを使えます。イメージは Docker 社が構築したものであり、適切な自動構築を通して定期的に更新しています。イメージを使うには、Docker Engine の docker run コマンドを通してコンテナを実行します。Swarm クラスタを作成・管理するため、イメージには複数のオプションとサブコマンドがあります。

イメージを初めて使う時、Docker Engine はイメージが自分の環境に既に存在しているかどうか確認します。Docker はデフォルトで swarm:latest バージョンを実行しますが、latest 以外のタグも指定できます。イメージがローカルにダウンロード済みでも、Docker Hub 上に新しいバージョンが存在していれば、Docker Engine は新しいイメージをダウンロードします。

コンテナで Swarm イメージを実行

1. Docker Engine を実行しているホスト上のターミナルを開きます。

2. Mac か Windows を使っている場合は、Docker Machine コマンドで Docker Engine ホストを起動し、ターミナルの環境を対象ホストに向ける必要があります。動作確認は次のようにします。

```
$ docker-machine ls
NAME      ACTIVE   URL          STATE     URL                         SWARM   DOCKER    ERRORS
default   *        virtualbox   Running   tcp://192.168.99.100:2376           v1.9.1
```

これは Docker Engine ホスト上で動いている default 仮想マシンの環境を指し示しています。

2. swarm イメージを使ってコマンドを実行します。

最も簡単なコマンドはイメージのヘルプ表示です。次のコマンドはイメージで利用可能なオプションの全てを表示します。

```
$ docker run swarm --help
Unable to find image 'swarm:latest' locally
latest: Pulling from library/swarm
d681c900c6e3: Pull complete
188de6f24f3f: Pull complete
90b2ffb8d338: Pull complete
237af4efea94: Pull complete
3b3fc6f62107: Pull complete
7e6c9135b308: Pull complete
986340ab62f0: Pull complete
a9975e2cc0a3: Pull complete
Digest: sha256:c21fd414b0488637b1f05f13a59b032a3f9da5d818d31da1a4ca98a84c0c781b
Status: Downloaded newer image for swarm:latest
使い方: swarm [オプション] コマンド [引数...]

Docker ネイティブのクラスタリング・システム

Version: 1.0.1 (744e3a3)

オプション:
  --debug            デバッグ・モード [$DEBUG]
  --log-level, -l "info"    ログレベル (選択肢: debug, info, warn, error, fatal, panic)
  --help, -h             ヘルプ表示
  --version, -v          バージョンを表示

コマンド:
  create, c クラスタの作成
  list, l   クラスタの一覧
  manage, m docker クラスタの管理
  join, j   docker クラスタへ参加
  help, h   コマンド一覧、あるいは特定のコマンドのヘルプを表示

コマンド 'swarm コマンド名 --help' で詳細情報を表示。
```

この例では swarm イメージが Engine ホスト上に存在していないため、Engine はイメージをダウンロードします。ダウンロード後、イメージは help サブコマンドを実行し、ヘルプ・テキストを表示します。ヘルプを表示した後、swarm イメージが終了し、ターミナル上のコマンドラインに戻ります。

3. Engine ホスト上で実行しているコンテナ一覧を表示します。

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
PORTS               NAMES
```

Swarm は動作していません。swarm イメージはコマンドを実行して終了したからです。

なぜイメージを使うのですか？

Swarm コンテナを使う方法は、他の手法と比べて 3 つの重要な利点があります。

- イメージを使えば、システム上にバイナリのインストールが不要。

- `docker run` コマンドを実行するだけで、常に最新バージョンのイメージを毎回取得。
- コンテナはホスト環境と Swarm を分離。シェル上のパスや環境変数の指定・調整が不要。

Swarm イメージの実行は Swarm クラスタの作成・管理のために推奨されている方法です。Docker の全ドキュメントおよびチュートリアルで使われている方法がこちらです。

2.1.2 Swarm バイナリの実行

ホストのオペレーティング・システム（OS）上で直接 Swarm バイナリを実行する前に、ソースコードからバイナリをコンパイルするか、信頼できる別の場所からコピーする必要があります。その作業の後、Swarm のバイナリを実行します。

ソースコードから Swarm をコンパイルするには、CONTRIBUTING.md[*1] の手順をご覧ください。

なぜバイナリを使うのですか？

他の方法に比べ、Swarm バイナリには利点が 1 つあります。もしあなたが swarm プロジェクトに貢献している開発者であれば、「コンテナ化」したバイナリを実行しなくても、コードに対する変更をテスト可能です。

ホスト OS 上で Swarm バイナリを実行する場合は、不利な点が 3 つあります。

- ソースからコンパイルする手間。
- Docker コンテナによってもたらされる隔離などの利点は、バイナリには無い。
- 大部分の Docker ドキュメントやチュートリアルは、バイナリで実行する方法では説明していない。

加えて、Swarm ノードは Engine を使いませんので、ノード上では Docker Engine CLI のような Docker ベースのソフトウェア・ツールで扱えません。

*1 http://github.com/docker/swarm/blob/master/CONTRIBUTING.md

2.2 Swarm を検証環境で試すには

このセクションでは、Docker Swarm という Docker 用のネイティブなクラスタリング・ツールを使う方法を紹介します。

作業では、Docker Machine がインストールされた Docker Toolbox と、コンピュータ上の他のツールをいくつか使います。Docker Machine は Docker Engine ホスト群をプロビジョン（自動構築）するために使います。そして、Docker クライアントはホストに接続するために使います。ここでのホストとは、ディスカバリ・トークンを作成する場所であり、Swarm マネージャとノードでクラスタを作成・管理するための場所もあります。

準備が完了したら、ローカルの Mac か Windows コンピュータで動く VirtualBox 上で Docker Swarm を起動します。この Swarm 環境は個人的な開発環境（サンドボックス）として使えます。

Docker Swarm を Linux 上で使う場合は、「プロダクション用の Swarm クラスタ構築」のセクションをご覧ください。

2.2.1 Docker Toolbox のインストール

Docker Toolbox[*1] のダウンロードとインストールをします。

Toolbox はローカルの Windows や Mac OS X コンピュータ上に便利なツールをインストールします。この練習では、３つのツールを使います。

- Docker Machine：Docker Engine を実行する仮想マシンをデプロイします。
- VirtualBox：Docker Machine を使い、仮想マシンのホストをデプロイします。
- Docker クライアント：ローカルのコンピュータから仮想マシン上の Docker Engine に接続します。また、docker コマンドで Swarm クラスタを作成します。

以下のセクションでは各ツールの詳細を説明します。以降は仮想マシン（Virtual Machine）を VM と略します。

2.2.2 Docker Engine で３つの VM を作成

ここでは Docker Machine を使い、Docker Engine を動かす３つの VM を作成（プロビジョン）します。

1. 自分のコンピュータ上のターミナルを開きます。Docker Machine で VirtualBox 上の VM 一覧を表示します。

```
$ docker-machine ls
NAME        ACTIVE   DRIVER       STATE     URL                         SWARM
default     *        virtualbox   Running   tcp://192.168.99.100:2376
```

2. オプション：システム・リソースを節約するために、使っていない仮想マシンを停止します。例えば、`default` という名前の VM を停止するには、次のようにします。

```
$ docker-machine stop default
```

3. `manager`（マネージャ）という名前の VM を作成・実行します。

```
$ docker-machine create -d virtualbox manager
```

*1 https://www.docker.com/docker-toolbox

4. agent1（エージェント１）という名前の VM を作成・実行します。

```
$ docker-machine create -d virtualbox agent1
```

5. agent2（エージェント２）という名前の VM を作成・実行します。

```
$ docker-machine create -d virtualbox agent2
```

各 create コマンドの実行時、boot2docker.iso と呼ばれる VM イメージの最新版がローカルにコピーされているかどうか（自動的に）確認します。ファイルが存在しないか最新版でなければ、Docker Machine は Docker Hub からイメージをダウンロードします。それから Docker Machine は boot2docker.iso を使い、Docker Engine を自動的に実行する VM を作成します。

トラブルシューティング：コンピュータやホストが Docker Hub にアクセスできなければ、docker-machine や docker run コマンドは失敗します。そのような場合、サービスが利用可能かどうか Docker Hub ステータス・ページ[*1] を確認します。その次は、自分のコンピュータがインターネットに接続しているか確認します。あるいは VirtualBox のネットワーク設定で、ホストがインターネット側に接続可能かどうかを確認してください。

2.2.3 Swarm ディスカバリ・トークンの作成

ここでは Docker Hub 上にある**ディスカバリ・バックエンド**を使い、自分のクラスタ用のユニークな**ディスカバリ・トークン**を作成します。このディスカバリ・バックエンドは、小規模の開発環境やテスト目的のためであり、プロダクション向けではありません。Swarm マネージャとノードを起動後、ディスカバリ・バックエンドにノードをクラスタのメンバとして登録します。クラスタとユニークなトークンを結び付けるのが、このバックエンドの役割です。ディスカバリ・バックエンドはクラスタのメンバのリストを最新情報に更新し続け、その情報を Swarm マネージャと共有します。Swarm マネージャはこのリストを使いノードに対してタスクを割り当てます。

1. コンピュータ上の Docker クライアントを Docker Engine が動いている manager に接続します[*2]。

```
$ eval $(docker-machine env manager)
```

以降の手順では、クライアント側の docker コマンドは manager 上の Docker Engine に送信します。

2. Swarm クラスタに対するユニーク ID を作成します。

```
$ docker run --rm swarm create
.
.
.
Status: Downloaded newer image for swarm:latest
0ac50ef75c9739f5bfeeaf00503d4e6e
```

docker run コマンドは最新（latest）の swarm を取得し、コンテナとして実行します。引数 create は Swarm コ

*1 http://status.docker.com/

*2 訳者注：環境を指し示す状況であり、実際には接続を維持しません。コマンド実行時に、都度、アクセスします。

ンテナを Docker Hub ディスカバリ・サービスに接続し、ユニークな Swarm ID を取得します。この ID を「ディスカバリ・トークン」(discovery token) と呼びます。トークンは出力（アウトプット）されるだけであり、ホスト上のファイルには保管されません。 --rm オプションは自動的にコンテナを削除するものです。コンテナが終了したら、コンテナのファイルシステムを自動的に削除します。

トークンを利用しなければ、およそ一週間後にディスカバリ・サービスによって削除されます。

3. 先ほどの出力されたディスカバリ・トークンを安全な場所にコピーします。

2.2.4 Swarm マネージャとノードの作成

ここでは、各ホストに接続し、Swarm マネージャまたはノードを作成します。

1. 3 つの VM の IP アドレスを取得します。例：

```
$ docker-machine ls
NAME      ACTIVE   DRIVER       STATE     URL                         SWARM   DOCKER   ERRORS
agent1    -        virtualbox   Running   tcp://192.168.99.102:2376           v1.9.1
agent2    -        virtualbox   Running   tcp://192.168.99.103:2376           v1.9.1
manager   *        virtualbox   Running   tcp://192.168.99.100:2376           v1.9.1
```

2. クライアントは manager を実行する Docker Engine を指し示しているままでしょう。次の構文は manager 上で Swarm コンテナをプライマリ Swarm マネージャとして実行します。

```
$ docker run -d -p <your_selected_port>:3376 -t -v /var/lib/boot2docker:/certs:ro swarm manage \
    -H 0.0.0.0:3376 --tlsverify --tlscacert=/certs/ca.pem --tlscert=/certs/server.pem \
    --tlskey=/certs/server-key.pem token://<cluster_id>
```

例：

```
$ docker run -d -p 3376:3376 -t -v /var/lib/boot2docker:/certs:ro swarm manage \
    -H 0.0.0.0:3376 --tlsverify --tlscacert=/certs/ca.pem --tlscert=/certs/server.pem \
    --tlskey=/certs/server-key.pem swarm manage token://0ac50ef75c9739f5bfeeaf00503d4e6e
```

-p オプションは、コンテナのポート 3376 をホスト上の 3376 に割り当てています。-v オプションは TLS 証明書が入っているディレクトリ（manager VM 上の/var/lib/boot2docker）をマウントします。これは Swarm マネージャの中では読み込み専用（read-only）モードで扱われます。

3. Docker クライアントを agent1 に接続します。

```
$ eval $(docker-machine env agent1)
```

4. 次の構文は agent1 上で Swarm コンテナをエージェントとして起動します。IP アドレスは VM のものに書き換えます。

```
$ docker run -d swarm join --addr=<ノードの ip>:<
```

例：

```
$ docker run -d swarm join --addr=192.168.99.102:2376 token://0ac50ef75c9739f5bfeeaf00503d4e6e
```

5. Docker クライアントを agent2 に接続します。

```
$ eval $(docker-machine env agent2)
```

6. agent2 上で Swarm コンテナをエージェントとして起動します。

```
$ docker run -d swarm join --addr=192.168.99.103:2376 token://0ac50ef75c9739f5bfeeaf00503d4e6e
```

2.2.5 Swarm クラスタを管理

ここではクラスタに接続し、Swarm マネージャとノードの情報を見ていきます。Swarm に対してコンテナ実行を命令し、どのノードで動作しているかを確認します。

1. Docker クライアントを Swarm に接続するため、DOCKER_HOST 環境変数を更新します。

```
$ DOCKER_HOST=<manager_ip>:<your_selected_port>
```

この例では manager の IP アドレスは 192.168.99.100 です。Swarm マネージャ用のポートは 3376 を選びました。

```
$ DOCKER_HOST=192.168.99.100:3376
```

Docker Swarm は標準 Docker API を使うため、Docker クライアントで接続できます。他にも Docker Compose や、Dokku、Jenkins、Krane などのツールが利用できます。

2. Swarm に関する情報を取得します。

```
$ docker info
```

実行したら、Swarm 上にあるマネージャ 1 つと、エージェント・ノード 2 つの情報を表示します。

3. Swarm 上で実行中のイメージを確認します。

```
$ docker ps
```

4. Swarm 上でコンテナを実行します。

```
$ docker run hello-world
Hello from Docker.
.
.
.
```

5. docker ps コマンドを使い、どのノードでコンテナが実行されているかを確認します。実行例：

```
$ docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
          PORTS               NAMES
0b0628349187        hello-world         "/hello"                 20 minutes ago      Exited (0) 20
minutes ago                         agent1
.
.
.
```

この例では、swarm1 上で hello-world が動いていました。

Docker Swarm がコンテナをどのノードで実行するかを決めるには、デフォルトでは「spread」（スプレッド）ストラテジを使います。複数のコンテナを実行する場合、スプレッド・ストラテジはコンテナの実行数が最も少ないノードに対してコンテナを割り当てます。

2.2.6 更に詳しく

ここまでは次の作業を行いました。

- Swarm ディスカバリ・トークンの作成
- Docker Machine を使って Swarm ノードを作成
- Swarm を使ってコンテナを実行
- Swarm に関連する概念と技術を学んだ

この他にも Docker Swarm には多くの特徴や能力があります。より詳しい情報は、Swarm のランディング・ページ[1]（英語）や引き続き Swarm ドキュメントをご覧ください。

*1 https://www.docker.com/docker-swarm

2.3 プロダクションで Swarm 利用時の考慮

このセクションは Docker Swarm クラスタを計画・デプロイ・管理の手助けとなるガイダンスを提供します。想定しているのは、ビジネスにおけるクリティカルなプロダクション環境です。次のハイレベルな項目を扱います。

- セキュリティ
- 高可用性（HA）
- 性能
- クラスタの所有

2.3.1 セキュリティ

Docker Swarm クラスタを安全にする様々な方法があります。このセクションは以下の内容を扱います。

- TLS を使った認証
- ネットワークのアクセス制御

これらのトピックだけでは完全ではありません。広範囲にわたるセキュリティ・アーキテクチャにおける一部分です。セキュリティ・アーキテクチャに含まれるのは、セキュリティ・パッチ、強力なパスワード・ポリシー、ロールをベースとしたアクセス制御、SELinux や AppArmor のように厳格な監査を行う技術、等々です。

Swarm 用の TLS 設定

Swarm クラスタ内にある全てのノードは、Docker Engine デーモンの通信用にポートを開く必要があります。そのため、中間者攻撃（man-in-the-middle attacks）のように、通常のネットワークに関連するセキュリティの問題をもたらします。対象のネットワークがインターネットのような信頼できない環境であれば、これらの危険性が倍増します。危険性を逓減するために、Swarm と Engine は TLS 認証（Transport Layer Security）をサポートしています。

Engine デーモンおよび Swarm マネージャは、TLS を使う設定をすることで、署名された Docker Engine クライアントからのみ通信を受け付けるようにできます。Engine と Swarm は組織内部の認証局（CA; Certificate Authorities）だけでなく、外部のサード・パーティー製による認証局もサポートしています。

Engine と Swarm が TLS に使うデフォルトのポート番号：

- Engine デーモン：2376/tcp
- Swarm マネージャ：3376/tcp

Swarm に TLS 設定を行うための詳しい情報は、「Swarm と TLS の概要」セクションをご覧ください。

ネットワークのアクセス制御

プロダクションにおけるネットワークは複雑であり、通常は特定のトラフィックのみがネットワーク上に流れるように固定します。以下のリストは Swarm クラスタの各コンポーネントが公開しているポート情報です。ファイアウォールや、他のネットワーク・アクセス管理リストの設定に、これらが使えるでしょう。

Swarm マネージャ：

- `Inbound 80/tcp (HTTP)`：`docker pull` コマンドが動作するために使います。Docker Hub からイメージを取得するためには、インターネット側のポート 80 を通す通信の許可が必要です。

- Inbound 2375/tcp：Docker Engine CLI が Engine デーモンと直接通信します。
- Inbound 3375/tcp：Docker Engine CLI が Swarm マネージャと通信します。
- Inbound 22/tcp：SSH を経由したリモート管理を行います。

サービス・ディスカバリ：

- Inbound 80/tcp (HTTP)：docker pull コマンドが動作するために使います。Docker Hub からイメージを取得するためには、インターネット側のポート 80 を通す通信の許可が必要です。
- Inbound （ディスカバリ・サービス用のポート番号)：バックエンド・ディスカバリ・サービス（consul、etcd、zookeeper）が公開するポートの設定が必要です。
- Inbound 22/tcp：SSH を経由したリモート管理を行います。

Swarm ノード ：

- Inbound 80/tcp (HTTP)：docker pull コマンドが動作するために使います。Docker Hub からイメージを取得するためには、インターネット側のポート 80 を通す通信の許可が必要です。
- Inbound 2375/tcp：Docker Engine CLI が Engine デーモンと直接通信します。
- Inbound 22/tcp：SSH を経由したリモート管理を行います。

その他、ホスト横断コンテナ・ネットワーク ：

- Inbound 7946/tcp：他のコンテナ・ネットワークから発見（ディスカバリ）されるために必要です。
- Inbound 7946/udp：他のコンテナ・ネットワークから発見（ディスカバリ）されるために必要です。
- Inbound /tcp：キーバリュー・ストアのサービス用ポートに接続します。
- 7489/udp：コンテナのオーバレイ・ネットワーク用です。

もしもファイアウォールがネットワーク・デバイスとの接続状態を検出（state aware）すると、TCP 接続を確立するための応答を許可します。デバイスが検出できなければ、23768 〜 65525 までのエフェメラル・ポート（訳者注：短時間のみ利用するポート）を自分でオープンにする必要があります。エフェメラル・ポートのルールという、セキュリティ設定の追加のみが、既知の Swarm デバイス上のインターフェースからの接続を受け付けるようにします。

Swarm クラスタが TLS 用の設定を行っている場合、2375 は 2376 に、3375 は 3376 に置き換えます。

クラスタ作成やクラスタ管理、クラスタ上でコンテナをスケジューリングといった Swarm クラスタの操作には、先ほどのリストにあるポートの調整が必要です。アプリケーションに関連する通信のために、追加で通信用ポートの公開も必要になる場合があります。

Swarm クラスタのコンポーネントは、別のネットワークに接続する可能性があります。例えば、多くの組織が管理用のネットワークとプロダクション用のネットワークを分けています。ある Docker Engine クライアントは管理ネットワーク上に存在しており、Swarm マネージャ、ディスカバリ・サービス用インスタンスやノードが 1 つまたは複数のネットワークにあるかもしれません。ネットワーク障害を埋め合わせるために、Swarm マネージャ、ディスカバリ・サービス、ノードが複数のプロダクション用ネットワークを横断することも可能です。先ほどのポート番号の一覧は、皆さんのネットワーク基盤チームがネットワークを効率的・安全に設定するために役立つでしょう。

2.3.2 高可用性（HA）

全てのプロダクション環境は高可用性（HA; Highly available）であるべきでしょう。つまり、長期間にわたる継続的な運用を意味します。高可用性を実現するのは、個々のコンポーネントで障害が発生しても切り抜ける環境です。

回復力のある高可用性 Swarm クラスタを構築するために、以下のセクションでは、いくつかの技術やベストプラクティスについて議論します。これらクラスタは、要求の厳しいプロダクションにおけるアプリケーションやワークロードで利用可能です。

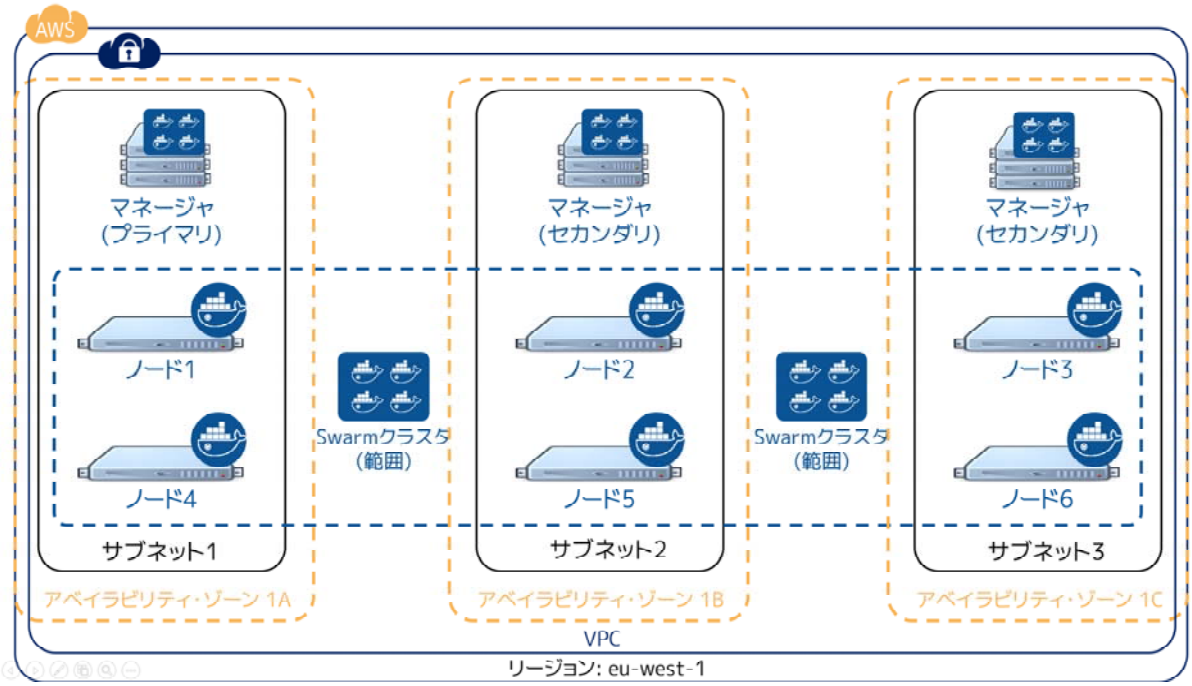
Swarm マネージャ HA

Swarm マネージャは Swarm クラスタに対する全ての命令を受け付ける責任を持ちます。それと、クラスタ内のリソースをスケジューリングする役割があります。もしも Swarm マネージャが利用不可能になれば、再び Swarm マネージャが使えるようになるまでクラスタに対する操作が不可能になります。これは大きくスケールするビジネスにおいては致命的なシナリオであり、許されません。

Swarm が提供する HA 機能は、Swarm マネージャで発生しうる障害を緩和します。クラスタ上に複数の Swarm マネージャを設定することで、Swarm の HA 機能を利用できます。３つの Swarm マネージャがアクティブ／パッシブ（活動中／受け身）を形成します。この時、１つのマネージャが プライマリ であり、残りの全てがセカンダリになります。

Swarm のセカンダリ・マネージャは ウォーム・スタンバイ として扱われます。つまり、プライマリ Swarm マネージャのバックグラウンドで動作することを意味します。セカンダリ Swarm マネージャはオンラインのままであり、プライマリ Swarm マネージャと同様、クラスタに対するコマンドを受け付けます。しかしながら、セカンダリが受信したコマンドはプライマリに転送され、その後に実行されます。プライマリ Swarm マネージャが落ちたとしても、残ったセカンダリの中から新しいプライマリが選出されます。

HA Swarm マネージャの作成時は、障害範囲（failure domains)の影響を受けないよう、可能な限り分散するよう注意を払う必要があります。障害範囲とは、デバイスまたはサービスに対する致命的な問題が発生すると影響があるネットワーク区分です。仮にクラスタが Amazon Web Services のアイルランド・リージョン（eu-west-1）で動いているとします。３つの Swarm マネージャを設定するにあたり（１つはプライマリ、２つはセカンダリ)、次の図のように各アベイラビリティ・ゾーンに置くべきでしょう。



この設定であれば、Swarm クラスタは２つのアベイラビリティ・ゾーンが失われても稼働し続けられます。あなたのアプリケーションが障害を乗り越えるためには、アプリケーションの障害範囲も重複しないよう設計する必要があります。

事業部で需要の高いアプリケーション向けに Swarm クラスタを使う場合は、３つ以上の Swarm マネージャを準備すべきです。そのように設定しておけば、１つのマネージャがメンテナンスのために停止しても、あるいは障害に直面したとしても、クラスタを管理・運用し続けられます。

ディスカバリ・サービス HA

ディスカバリ・サービスは Swarm クラスタにおける重要なコンポーネントです。ディスカバリ・サービスが使えなくなると、適切なクラスタ操作ができなくなります。例えば、ディスカバリ・サービスが動作しなくなったら、クラスタに新しいノードの追加といった操作や、クラスタ設定に関する問い合わせに失敗します。これはビジネスにおけるクリティカルなプロダクション環境では許容できません。

Swarm は 4 つのバックエンド・ディスカバリ・サービスをサポートしています。

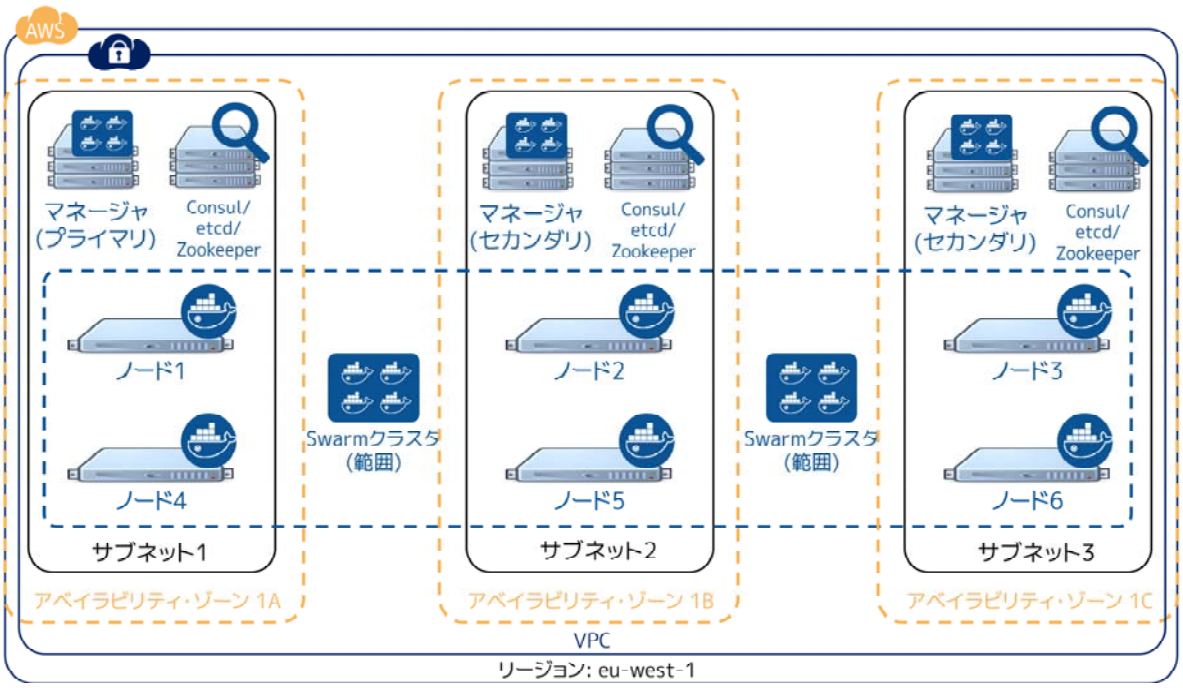
- ホステッド（プロダクション向けではない）
- Consul
- etcd
- Zookeeper

Consul、etcd、Zookeeper はどれもプロダクションにふさわしく、高可用性のために設定されるべきです。HA 向けのベスト・プラクティスを設定するためには、これら各サービスのツールを使うべきでしょう。

事業部で高い需要のアプリケーション向けに Swarm を使う場合は、5 つ以上のディスカバリ・サービス・インスタンスの用意を推奨します。これはレプリケーション/HA で用いられている（Paxos や Raft のような）技術が強力なクォーラム（quorum）を必要とするためです。5 つのインスタンスがあれば、1 つがメンテナンスや予期しない障害に直面しても、強力なクォーラムを形成し続けられます。

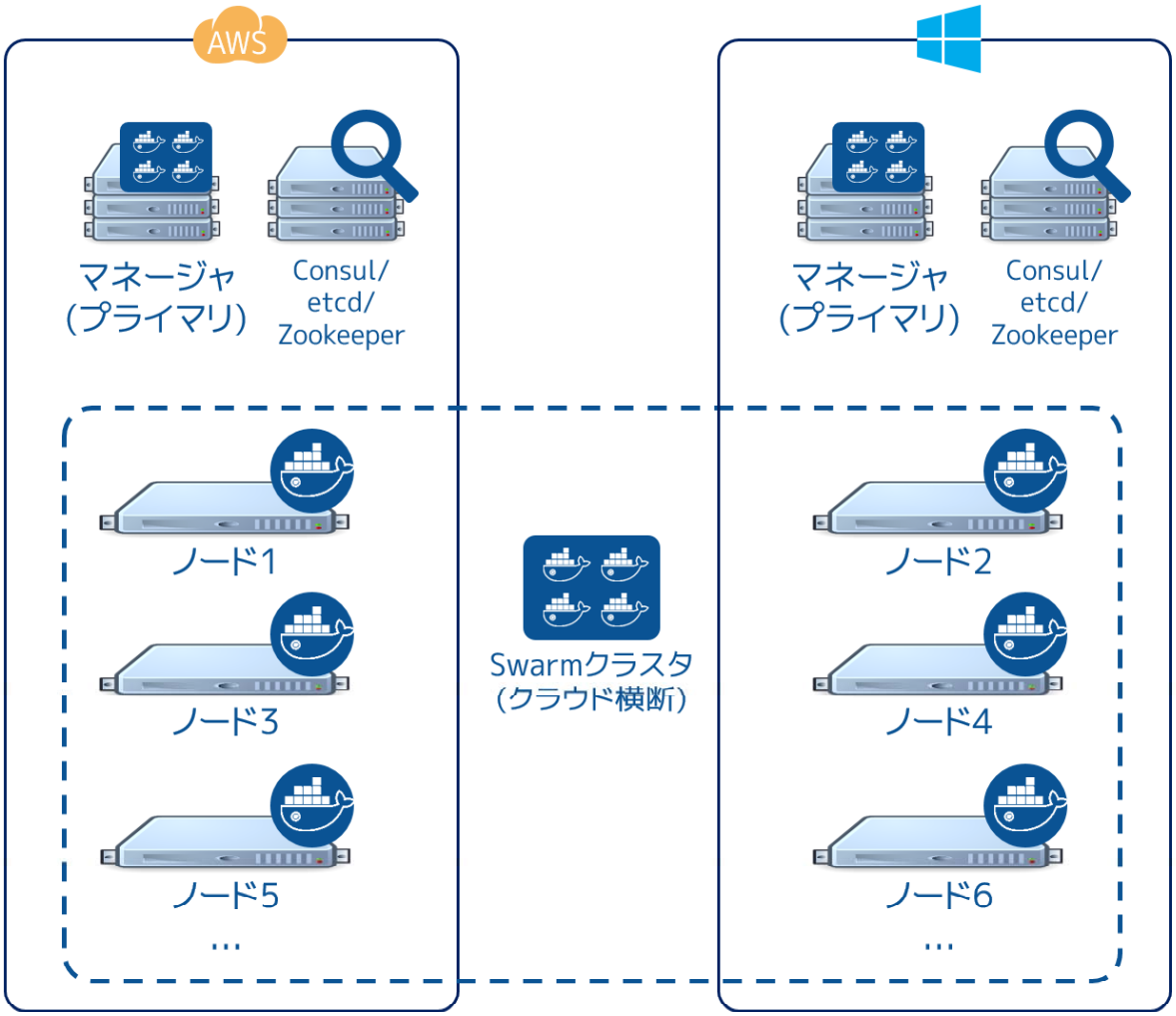
高い可用性を持つ Swarm ディスカバリ・サービスを作成する場合には、各ディスカバリ・サービス・インスタンスを可能な限り障害範囲を重複しないようにすべきでしょう。例えば、クラスタを Amazon Web Service のアイルランド・リージョン（eu-west-1）で動かしているとします。3 つのディスカバリ・サービス用インスタンス設定する時、それぞれを各アベイラビリティ・ゾーンに置くべきです。

次の図は HA を設定した Swarm クラスタ設定を表しています。3 つの Swarm マネージャと 3 つのディスカバリ・サービス・インスタンスが 3 つの障害領域（アベイラビリティ・ゾーン）に展開してます。また、Swarm ノードは 3 つの障害領域を横断しています。2 つのアベイラビリティ・ゾーンで障害が発生したとしても、Swarm クラスタは停止しない設定を表しています。



複数のクラウド

Swarm クラスタを複数のクラウド・プロバイダを横断するよう設計・構築できます。これはパブリック・クラウドでも、オンプレミスの基盤でもです。次の図は Swarm クラスタを AWS と Azure に横断しています。



このアーキテクチャは究極の可用性を提供しているように見えるかもしれませんが、考慮すべき複数の要素があります。ネットワークのレイテンシ（応答遅延）は問題になりがちです。パーティショニング（分割）も問題になりうるでしょう。クラウド・プラットフォームにおいて信頼性、高スピード、低いレイテンシを実現する技術の考慮が必要となるでしょう。例えば AWS ダイレクト・コネクトや Azure ExpressRoute といった技術です。

このように、プロダクションを複数のインフラに横断する検討する場合は、あなたのシステム全体にわたるテストを確実に行うべきでしょう。

プロダクション環境の分離

開発、ステージング、プロダクションのような複数の環境を、1 つの Swarm クラスタ上で動かせるでしょう。そのためには Swarm ノードをタグ付けし、`production` や `staging` 等のようにタグ付けされたコンテナを制約フィルタ（constraint filter）で使う方法があります。しかしながら、これは推奨しません。ビジネスにおけるクリティカルなプロダクション環境において高いパフォーマンスが必要な時は、エアギャップ・プロダクション環境の手法を推奨します。

例えば、多くの会社では、プロダクション用に分離された専用環境にデプロイするでしょう。専用環境とは、ネットワーク、ストレージ、計算資源、その他のシステムです。デプロイは別の管理システムやポリシーで行われます。その結果、プロダクション・システム等にログインするために、別のアカウント情報を持つ必要があります。

この種の環境では、プロダクション専用の Swarm クラスタへデプロイする義務があるでしょう。プロダクションのハードウェア基盤で Swarm クラスタを動かし、そこでプロダクションにおける管理・監視・監査・その他のポリシーに従うことになります。

オペレーティング・システムの選択

Swarm 基盤が依存するオペレーティング・システムの選択には重要な考慮をすべきです。考慮こそがプロダクション環境における核心となります。

開発環境とプロダクション環境でオペレーティング・システムを変えて使う会社は珍しくありません。よくあるのが、開発環境では CentOS を使いますが、プロダクション環境では Red Hat Enterprise Linux (RHEL)を使う場合です。しばしコストとサポートのバランスが決め手になります。CentOS Linux は自由にダウンロードして利用できますが、商用サポートのオプションは僅かなものです。一方の RHEL であればサポートに対してライセンスのコストが想定されますが、Red Hat による世界的な商用サポートが受けられます。

プロダクション向けの Swarm クラスタで使うオペレーティング・システムの選定にあたっては、開発環境とステージング環境で使っているものに近いものを選ぶべきでしょう。コンテナが根本となる OS を抽象化するといえども、避けられない課題があるためです。例えば、Docker コンテナのネットワークには Linux カーネル 3.16 以上が必要です。開発・ステージング環境でカーネル 4.x 系を使っていても、プロダクションが 3.14 であれば何らかの問題が発生します。

他にも考慮すべき点として、手順、デプロイの順序、プロダクション用オペレーティング・システムへのパッチ適用の可能性があるでしょう。

2.3.3 性能

重要な商用アプリケーションを扱う環境にとって、性能（パフォーマンス）が非常に重要です。以下のセクションでは高性能な Swarm クラスタを構築する手助けとなるような議論と手法を紹介します。

コンテナ・ネットワーク

Docker Engine のコンテナ・ネットワークがオーバレイ・ネットワークであれば、複数の Engine ホスト上を横断して作成可能です。そのためには、コンテナ・ネットワークがキーバリュー（KV）・ストアを必要とします。これは Swarm クラスタのディスカバリ・サービスで情報を共有するために使います。しかしながら、最高の性能と障害の分離のためには、コンテナ・ネットワーク用と Swarm ディスカバリ用に別々の KV インスタンスをデプロイすべきでしょう。特に、ビジネスにおけるクリティカルなプロダクション環境においては重要です。

Engine のコンテナ・ネットワークは Linux カーネルの 3.16 以上を必要とします。より高いカーネル・バージョンの利用が望ましいのですが、新しいカーネルには不安定さというリスクが増えてしまいます。可能であれば、皆さんがプロダクション環境で既に利用しているカーネルのバージョンを使うべきです。もしも Linux カーネル 3.16 以上をプロダクションで使っていなければ、可能な限り早く使い始めるべきでしょう。

スケジューリング・ストラテジ

スケジューリング・ストラテジとは、Swarm がコンテナを開始する時に、どのノードか、どのクラスタ上で実行するかを決めるものです。

- spread
- binpack
- random （プロダクション向けではありません）

自分自身で書くこともできます。

`spread` （スプレッド）はデフォルトのストラテジです。クラスタ上の全てのノードにわたり、均一な数のコンテ

ナになるようバランスを取ろうとします。高い性能を必要とするクラスタでは良い選択肢です。コンテナのワークロードをクラスタ全体のリソースに展開するからです。リソースには CPU、メモリ、ストレジ、ネットワーク帯域が含まれます。

もし Swarm ノードで障害が発生したら、Swarm は障害領域を避けてコンテナを実行するようにバランスを取ります。しかしながら、コンテナの役割には注意が払われないため、関係なく展開されます。そのため、サービス展開先を複数の領域に分けたくても、Swarm は把握できません。このような操作を行うには、タグと制限（constraint）を使うべきです。

`binpack`（ビンパック）ストラテジは、ノードに次々とコンテナをスケジュールするのではなく、可能な限り 1 つのノード上にコンテナを詰め込もうとします。

つまり、binpack はクラスタを使い切るまで全てのクラスタ・リソースを使いません。そのため、binpack ストラテジの Swarm クラスタ上で動作するアプリケーションによっては、性能が出ないかもしれません。しかしながら、binpack は必要なインフラとコストの最小化のために良い選択肢です。例えば 10 ノードのクラスタがあり、それぞれ 16 CPU・128 GB のメモリを持っていると想像してみましょう。コンテナのワークロードが必要になるのは、6 CPU と 64 GB のメモリとします。spread ストラテジであれば、クラスタ上の全てのノードにわたってバランスを取ります。一方、binpack ストラテジであれば、コンテナが 1 つのノード上を使い切ります。そのため、追加ノードを停止することで、コストの節約ができるかもしれません。

2.3.4 クラスタの所有

所有者が誰なのかというのは、プロダクション環境において極めて重要です。Swarm クラスタでプロダクションの計画、ドキュメントか、デプロイに至る全てにおける熟慮と合意が重要になります。

- プロダクションの Swarm 盤に対し、誰が予算を持っているのか？
- プロダクションの Swarm ラスタを誰が管理・運用するのか？
- プロダクションの Swarm 盤に対する監視は誰の責任か？
- プロダクションの Swarm 基盤のパッチあてやアップグレードは誰の責任か？
- 24 時間対応やエスカレーション手順は？

このリストは完全ではありません。何が答えなのかは、皆さんの組織やチーム構成によって様々に依存します。ある会社は DevOps の流れに従うかもしれませんし、そうではない場合もあるでしょう。重要なのは、皆さんの会社がどのような状況なのかです。プロダクション用 Swarm クラスタの計画、デプロイ、運用管理に至るまで、全ての要素の検討が重要です。

2.4 プロダクション用の Swarm クラスタ構築

高い可用性を持つ Docker Swarm クラスタのデプロイ方法を紹介します。この例では Amazon Web Services (AWS)をプラットフォームとして使いますが、他のプラットフォーム上でも Docker Swarm クラスタを同じようにデプロイできます。この例では、以下の手順で進めます。

- 動作条件の確認
- 基本的なネットワーク・セキュリティの確保
- ノードの作成
- 各ノードに Engine をインストール
- ディスカバリ・バックエンドの設定
- Swarm クラスタの作成
- Swarm との通信
- Swarm マネージャの高可用性試験
- 追加情報

一般的な Swarm の導入方法については「2.2 Swarm を検証環境で試すには」をご覧ください。

2.4.1 動作条件

- Amazon Web Services (AWS) アカウント
- 以下の AWS 機能やツールに慣れている：
    - エラスティック・クラウド（EC2）ダッシュボード
    - バーチャル・プライベート・クラウド（VPC）ダッシュボード
    - VPC セキュリティ・グループ
    - EC2 インスタンスに SSH で接続

2.4.2 手順

ステップ１：ネットワーク・セキュリティのルールを追加

AWS は VPC ネットワークに対して許可するネットワーク通信を「セキュリティ・グループ」で指定します。初期状態の `default` セキュリティ・グループは、インバウンドの通信を全て拒否し、アウトバンドの通信を全て許可し、インスタンス間の通信を許可するルール群です。

ここに SSH 接続とコンテナのイメージを取得（インバウンド）する２つのルールを追加します。このルール設定は Engine、Swarm、Consul のポートを少々は守ります。プロダクション環境においては、セキュリティ度合いによって更に制限するでしょう。Docker Engine のポートを無防備にしないでください。

AWS のホーム・コンソール画面から、以下の作業を進めます：

1. ［VPC - 独立したクラウドリソース］をクリックします。

VPC ダッシュボードが開きます。

2. ［セキュリティグループ］をクリックします。

3. ［default］セキュリティ・グループを選び、default VPC にセキュリティ・グループを関連付けます。

4. 以下の２つのルールを追加します。

タイプ	プロトコル	ポート範囲	送信元
SSH	TCP	22	0.0.0.0/0
HTTP	TCP	80	0.0.0.0/0

SSH 接続はホストに接続するためです。HTTP はコンテナ・イメージをホストにダウンロードするためです。

ステップ２：インスタンス作成

このステップではデフォルトのセキュリティ・グループで５つの Linux ホストを作成します。作業は３種類のノードをデプロイする例です

ノードの説明	名前
Swarm のプライマリとセカンダリ・マネージャ	manager0 , manager1
Swarm ノード	node0 , node1
ディスカバリ・バックエンド	consul0

インスタンスの作成は、以下の手順で進めます。

1. EC2 ダッシュボードを開き、各 EC2 インスタンスを同時に起動します。

- ［ステップ１］では：Amazon マシン・イメージ (AMI)を選択します。「Amazon Linux AMI」を探します。
- ［ステップ５］では：インスタンスにタグを付けます。各インスタンスの Value に名前を付けます。
    - manager0
    - manager1
    - consul0
    - node0
    - node1
- ［ステップ６］では：セキュリティ・グループを設定します。既存のセキュリティグループから「default」を探します。

2. インスタンスの起動を確認します。

ステップ３：各ノードに Engine をインストール

このステップでは、各ノードに Docker Engine をインストールします。Engine をインストールすることで、Swarm マネージャは Engine CLI と API を経由してノードを割り当てます。

SSH で各ノードに接続し、以下の手順を進めます。

1. yum パッケージを更新します。

「y/n/abort」プロンプトに注意します。

```
$ sudo yum update
```

2. インストール用スクリプトを実行します。

```
$ curl -sSL https://get.docker.com/ | sh
```

3. Docker Engine が Swarm ノードのポート 2375 で通信可能な指定で起動します。

```
$ sudo docker daemon -H tcp://0.0.0.0:2375 -H unix:///var/run/docker.sock
```

4. Docker Engine が正常にインストールされたことを確認します。

```
$ sudo docker run hello-world
```

「Hello World」メッセージが画面に表示され、エラーではない文字列が表示されます。

5. ec2-user に root 権限を与えます。

```
$ sudo usermod -aG docker ec2-user
```

6. logout を実行します。

トラブルシューティング

- docker コマンドを実行してもホスト上で docker が動作しているかどうか訊ねる表示が出るのは、ユーザが root 権限を持っていない可能性があります。そうであれば、sudo を使うか、ユーザに対して root 権限を付与します。

- この例では、Docker Engine が動作するインスタンスを元にして、他のインスタンス用に使う AMI イメージを作成しません。作成したとしても問題になるでしょう。

- ホスト上で docker run コマンドを実行しても Docker Hub に接続できない場合は、コンテナ・イメージの取得に失敗するでしょう。そのような場合、VPC に関連付けられているセキュリティ・グループのルールを参照し、インバウンドの通信（例：HTTP/TCP/80/0.0.0.0/0）が許可されているか確認します。また、Docker Hub ステータス・ページ でサービスが利用可能かどうか確認します。

ステップ４：ディスカバリ・バックエンドのセットアップ

ここでは最小のディスカバリ・バックエンドを作成します。Swarm マネージャとノードは、このバックエンドをクラスタ上のメンバを認識するために使います。また、Swarm マネージャはコンテナを実行可能なノードがどれかを識別するためにも使います。

簡単さを保つために、Swarm マネージャが動いているホストのうちどれか 1 つで consul デーモンを起動します。

1. 始めるにあたり、以下のコマンドをテキスト・ファイルにコピーしておきます。

```
$ docker run -d -p 8500:8500 --name=consul progrium/consul -server -bootstrap
```

2. SSH で manager0 と consul0 インスタンスに接続します。

```
$ ifconfig
```

3. 出力結果から inet addr の eth0 にある IP アドレスをコピーします。

4. SSH で manager0 と consul0 インスタンスに接続します。

5. 手順 1 で実行したコマンドを、コマンドラインに貼り付けます。

```
$ docker run -d -p 8500:8500 --name=consul progrium/consul -server -bootstrap
```

consul ノードを立ち上げ実行することで、クラスタ用のディスカバリ・バックエンドを提供します。このバックエンドの信頼性を高めるには、3つの consul ノードを使った高可用性クラスタを作成する方法があります。詳細情報へリンクを、このセクションの最後をご覧ください（consul ノードのクラスタを作成する前に、VPC セキュリティ・グループに対し、必要なポートに対するインバウンド通信を許可する必要があります)。

ステップ５：Swarm クラスタの作成

ディスカバリ・バックエンドを作った後は、Swarm マネージャを作成できます。このステップでは高い可用性を持つ設定のため、2つの Swarm マネージャを作成します。1つめのマネージャを Swarm の**プライマリ・マネージャ (primary manager)** とします。ドキュメントのいくつかはプライマリを「マスタ」と表現していますが、置き換えてください。2つめのマネージャは**レプリカ（replica）**を提供します。もしもプライマリ・マネージャが利用できなくなれば、クラスタはレプリカからプライマリ・マネージャを選出します。

1. 高可用性 Swarm クラスタのプライマリ・マネージャを作成するには、次の構文を使います。

```
$ docker run -d -p 4000:4000 swarm manage -H :4000 --replication --advertise <manager0_ip>:4000
consul://<consul_ip>
```

特定のマネージャは「manager0 & consul0」インスタンスの consul ノードでもあるので、 <manager0_ip>と<consul_ip> を同じ IP アドレスに書き換えます。例：

```
$ docker run -d -p 4000:4000 swarm manage -H :4000 --replication \
     --advertise 172.30.0.161:4000  consul://172.30.0.161:8500
```

2. docker ps を入力します。

出力結果から Swarm クラスタと consul コンテナが動いているのを確認します。それから manager0 と consul0 インスタンスから切断します。

3. manager1 インスタンスに接続し、ifconfig で IP アドレスを取得します。

```
$ ifconfig
```

4. 以下のコマンドを実行し、セカンダリ Swarm マネージャを起動します。

コマンドを実行前に <manager1_ip> の部分は書き換えてください。実行例：

```
$ docker run -d swarm manage -H :4000 --replication \
    --advertise <manager1_ip>:4000  consul://172.30.0.161:8500
```

5. docker ps を実行し、swarm コンテナの実行を確認します。

6. node0 と node1 に接続し、それぞれをクラスタに追加（join）します。

   a. ifconfig コマンドでノードの IP アドレスを確認します。

   b. 各コンテナで、次の構文を使って Swarm コンテナを起動します。

   ```
   $ docker run -d swarm join --advertise=<node_ip>:2375 consul://<consul_ip>:8500
   ```

   実行例：

   ```
   $ docker run -d swarm join – advertise=172.30.0.69:2375 consul://172.30.0.161:8500
   ```

あなたの小さな Swarm クラスタが起動し、複数のホスト上で実行中になりました。信頼性や収容能力を高めるには、Swarm マネージャやノードを更に追加し、ディレクトリ・バックエンドの可用性を高めることも可能です。

ステップ６：Swarm と通信

Swarm API を使って Swarm と通信し、マネージャとノードに関する情報を取得できます。Swarm API はスタンダード Docker API とよく似ています。この例では SSL を使って manager0 と consul0 ホストに再び接続します。そしてコマンドを Swarm マネージャに対して割り当てます。

1. クラスタ内のマスタとノードの情報を取得します。

   ```
   $ docker -H :4000 info
   ```

出力結果から、マスタの役割がプライマリ（Role: primary）であることと、各ノードの情報が分かります。

2. Swarm 上でアプリケーションを実行します。

   ```
   $ docker -H :4000 run hello-world
   ```

3. Swarm ノード上でアプリケーションが動いているのを確認します。

   ```
   $ docker -H :4000 ps
   ```

ステップ７：Swarm のフェイルオーバをテスト

レプリカ・インスタンスへの継承を確認するために、プライマリ・マネージャをシャットダウンします。これが選出のきっかけとなり、レプリカがプライマリ・マネージャになります。停止したマネージャを再び起動したら、今度はこちらがレプリカになります。

1. manage0 インスタンスに SSH 接続します。

2. swarm コンテナのコンテナ ID もしくはコンテナ名を取得します。

   ```
   $ docker ps
   ```

3. プライマリ・マスタをシャットダウンするため、 <id_name> の部分をコンテナ ID あるいはコンテナ名に置き換えます（例：「8862717fe6d3」または「trusting_lamarr」）。

```
$ docker rm -f <id_name>
```

4. swarm マスタを起動します。例：

```
$ docker run -d -p 4000:4000 swarm manage -H :4000 --replication --advertise 172.30.0.161:4000
consul://172.30.0.161:237
```

5. Engine デーモンのログを確認します。 <id_name> は新しいコンテナ ID かコンテナ名に置き換えます。

```
$ sudo docker logs <id_name>
```

出力から次のような２つのエントリが確認できます。

```
time="2016-02-02T02:12:32Z" level=info msg="Leader Election: Cluster leadership lost"
time="2016-02-02T02:12:32Z" level=info msg="New leader elected: 172.30.0.160:4000"
```

6. クラスタのマスタとノードに関する情報を取得するには、次のように実行します。

```
$ docker -H :4000 info
```

master1 ノードに接続し、info や logs コマンドを実行できます。そうすると、新しいリーダーが適切なエントリを返します。

2.5 Docker Machine で Swarm クラスタ構築

Docker Machine を使って Docker Swarm クラスタをプロビジョン（自動構築）できます。Docker Machine とは Docker のプロビジョニング・ツールです。Machine はホストをプロビジョンし、そこに Docker Engine をインストールし、Docker CLI クライアント用の設定を行います。Machine で Swarm 用のオプションを指定したら、プロビジョニングの過程で Swarm クラスタ用の設定も迅速に行えます。

このセクションでは、Machine で基本的な Swarm クラスタを構築するために必要なコマンドを紹介します。ローカルの Mac もしくは Windows 上に環境を構築します。流れを理解してしまえば、クラウド・プロバイダ上や、あるいは会社のデータセンタ内にも Swarm クラスタをセットアップできるようになるでしょう。

Swarm クラスタの構築が初めてであれば、まず Swarm について学び、「2.2 Swarm を検証環境で試すには」や「2.4 プロダクション用の Swarm クラスタ構築」を読んでおく必要があります。Machine を使うのが初めてであれば、Machine を使う前に概要の理解が望ましいでしょう。

2.5.1 何が必要ですか？

Mac OS X や Windows で Docker Toolbox を使ってインストールしていた場合は、既に Docker Machine がインストール済みです。インストールの必要があれば、Mac OS X または Windows のページをご覧ください。

Machine を使ったインストールをサポートしているのは、AWS、Digital Ocean、Google Cloud Platform、IBM SoftLayer、Microsoft Azure、Hyper-V、OpenStack、RackSpace、VirtualBox、VMware Fusion、vCloud Air、vSphere です。このページの例では VirtualBox 上で `boot2docker.iso` イメージを使った仮想マシンをいくつか起動します。このイメージは Docker Engine を実行するための最小ディストリビューションです。

Toolbox をインストールしたら、VirtualBox と必要に応じて `boot2docker.iso` イメージが用意されます。また、Machine がサポートするあらゆるシステム上へプロビジョン可能です。

この例では Mac OS X もしくは Windows の利用を想定していますが、Docker Machine は Linux システム上も直接インストール可能です 。

2.5.2 Swarm トークンをホスト上で生成

Swarm の設定を始める前に、Docker Engine の動くホストをプロビジョニングする必要があります。Machine をインストールしたホスト上のターミナルを開きます。それから `local` という名称のホストをプロビジョニングするため、次のように実行します。

```
docker-machine create -d virtualbox local
```

この例では VirtualBox を指定していますが、Digital Ocean やデータセンタ内のホストでも簡単に作成できます。`local` の値はホスト名です。作成したら、自分のターミナル上で `local` ホストと通信できるように、環境変数を指定します。

```
eval "$(docker-machine env local)"
```

各 Swarm ホストでは、Engine の設定時にトークンも指定します。このトークンは Swarm ディスカバリ・バックエンドが Swarm クラスタの適切なノードであることを認識するために必要です。クラスタ用のトークンを作成するには、`swarm` イメージを実行します。

```
docker run swarm create
```

```
Unable to find image 'swarm' locally
1.1.0-rc2: Pulling from library/swarm
892cb307750a: Pull complete
fe3c9860e6d5: Pull complete
cc01ef3f1fbc: Pull complete
b7e14a9c9c72: Pull complete
3ec746117013: Pull complete
703cb7acfce6: Pull complete
d4f6bb678158: Pull complete
2ad500e1bf96: Pull complete
Digest: sha256:f02993cd1afd86b399f35dc7ca0240969e971c92b0232a8839cf17a37d6e7009
Status: Downloaded newer image for swarm
0de84fa62a1d9e9cc2156111f63ac31f ←この文字列がトークン
```

swarm create コマンドの出力結果がクラスタ用のトークンです。このトークンを安全な場所にコピーして覚えておきます。このトークンは、Swarm ノードのプロビジョニング時や、そのノードをクラスタに追加する時のクラスタ ID として使います。トークンは、この後で環境変数 SWARM_CLUSTER_TOKEN として参照します。

2.5.3 Swarm ノードのプロビジョン

クラスタの全てのノードは Engine をインストールしている必要があります。Machine で SWARM_CLUSTER_TOKEN を使えば、Machine でコマンドを 1 つ実行するだけで、Engine のホストをプロビジョニングし、Swarm のノードとして設定された状態にします。新しい仮想マシンを Swarm マネージャ・ノードの swarm-manager として作成します。

```
docker-machine create \
    -d virtualbox \
    --swarm \
    --swarm-master \
    --swarm-discovery token://SWARM_CLUSTER_TOKEN \
    swarm-manager
```

次に追加用のノードをプロビジョニングします。ここでも SWARM_CLUSTER_TOKEN を指定する必要があります。そして、各ホストには HOST_NODE_NAME でユニークな名前を付ける必要があります。

```
docker-machine create \
    -d virtualbox \
    --swarm \
    --swarm-discovery token://SWARM_CLUSTER_TOKEN \
    HOST_NODE_NAME
```

例えば、HOST_NODE_NAME には node-01 のような名前を指定するでしょう。

ここまで実行したコマンドは Docker Hub が提供している Docker Swarm のホステッド・ディスカバリ・サービスに依存しています。もしも Docker Hub あるいはネットワークに問題があれば、これらのコマンド実行に失敗するでしょう。サービスが利用可能かどうか、Docker Hub ステータス・ページをご確認ください。Docker Hub で問題がある場合は復旧まで待つか、あるいは、別のディスカバリ・バックエンドの設定をご検討ください。

2.5.4 Machine でノード環境に接続

Machine 接続先のホストを環境変数で指定するには、env サブコマンドの利用が一般的です。

```
eval "$(docker-machine env local)"
```

Docker Machine には、env コマンドで Swarm ノードに接続するための、特別な --swarm フラグがあります。

```
docker-machine env --swarm HOST_NODE_NAME
export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://192.168.99.101:3376"
export DOCKER_CERT_PATH="/Users/mary/.docker/machine/machines/swarm-manager"
export DOCKER_MACHINE_NAME="swarm-manager"
# Run this command to configure your shell:
# eval $(docker-machine env --swarm HOST_NODE_NAME)
```

シェル上の操作を swarm-manager という名称の Swarm ノードに切り替えるには、次のように実行します。

```
eval "$(docker-machine env --swarm swarm-manager)"
```

これで Docker CLI を使ってクラスタと相互に通信できるようになりました。

```
docker info
Containers: 2
Images: 1
Role: primary
Strategy: spread
Filters: health, port, dependency, affinity, constraint
Nodes: 1
 swarm-manager: 192.168.99.101:2376
   └ Status: Healthy
   └ Containers: 2
   └ Reserved CPUs: 0 / 1
   └ Reserved Memory: 0 B / 1.021 GiB
   └ Labels: executiondriver=native-0.2, kernelversion=4.1.13-boot2docker,
operatingsystem=Boot2Docker 1.9.1 (TCL 6.4.1); master : cef800b - Fri Nov 20 19:33:59 UTC 2015,
provider=virtualbox, storagedriver=aufs
CPUs: 1
Total Memory: 1.021 GiB
Name: swarm-manager
```

3章

Swarm のスケール

ここで扱うサンプルは、Swarm クラスタ上に投票アプリケーションをデプロイします。例では典型的な開発プロセスを使って説明します。インフラを構築すると、Swarm クラスタの作成やクラスタ上にアプリケーションをデプロイできるようになります。

投票アプリケーションの構築と手動でデプロイした後、Docker Compose ファイルを作成します。あなた（もしくは誰か）はこのファイルを使い、アプリケーションの更なるデプロイやスケールが可能になります。

このサンプルは、新人ネットワーク管理者向けに書かれています。基本的な Linux システムのスキルを持ち、ssh 経験、Amazon が提供する AWS サービスに対する理解があることでしょう。また、Git のような知識があれば望ましいのですが、必須ではありません。以下の手順でサンプルを進めていくのに、約 1 時間ほどかかります。

3.1 アプリケーションのアーキテクチャを学ぶ

このセクションでは、Swarm をスケールさせるサンプルについて学びます。

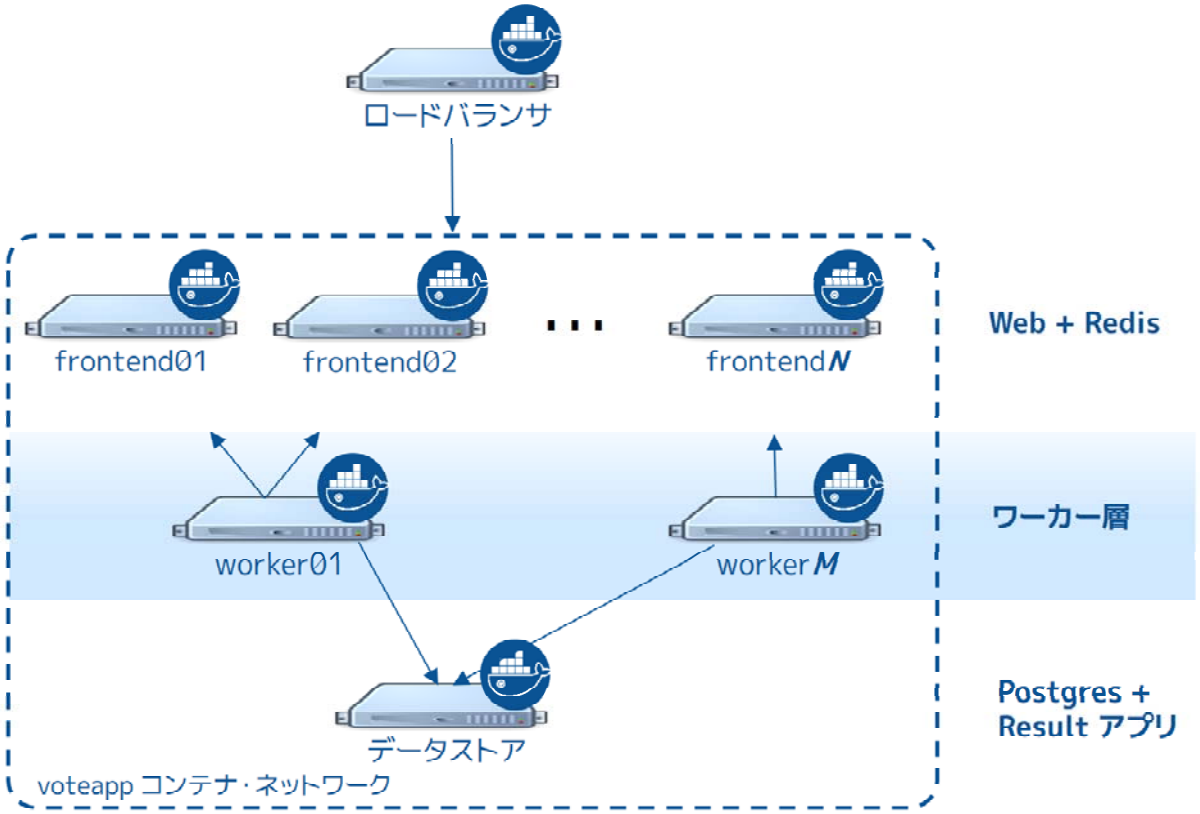
3.1.1 サンプルの背景を学ぶ

あなたの会社はペットフード会社であり、スーパーボウルのコマーシャル枠を購入しようとしています。コマーシャルでは、視聴者に対して調査のために犬か猫かの投票を呼びかけます。あなたはウェブ投票システムを開発します。

この調査では 100 万人もの人々が投票してもウェブサイトが止まらないようにする必要があります。結果をリアルタイムで知る必要は無く、結果は会社のプレスリリースで公開します。しかし、どれだけ投票されたかは、投票の度に確実に把握する必要があります。

3.1.2 アプリケーションのアーキテクチャを理解

投票アプリケーションは複数のマイクロサービスで構成されます。並列なウェブ・フロントエンドを使い、ジョブを非同期のバックグラウンド・ワーカに送ります。アプリケーションは任意に大きくスケール可能な設計です。次の図はアプリケーションのハイレベルなアーキテクチャです。



全てのサーバで Docker Engine が動いています。アプリケーション全体は完全に Docker 化（Dockerized）しており、全てのサービスをコンテナ内で実行します。

フロントエンドはロードバランサと N 台のフロントエンド・インスタンスで構成します。各フロントエンドはウェブ・サーバと Redis キューで構成します。ロードバランサは任意の数のウェブ・コンテナを背後で扱えます（`frontend01` ～ `frontendN`）。Web コンテナは２つの選択肢から投票するシンプルな Python アプリケーションです。キューに入った投票は datasotre 上の Redis コンテナに送られます。

フロントエンドの背後にはワーカ層があり、別々のノードが動いています。この層は次の機能があります。

- Redis コンテナをスキャン
- 投票のキューを回収
- 重複投票を防ぐために投票結果を複製
- 結果を Postgres データベースにコミットする

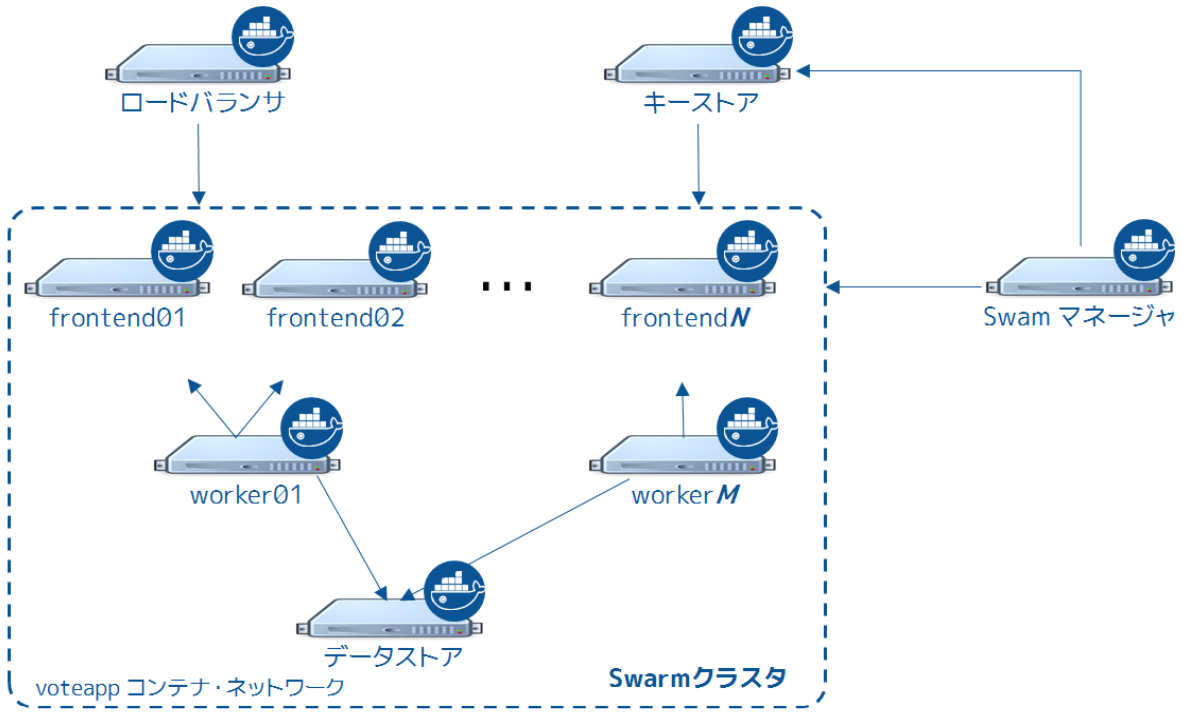
フロントエンドと同様に、ワーカ層も任意にスケールできます。ワーカの数とフロントエンドの数は、お互い独立しています。

Docker 化したマイクロサービスのアプリケーションを、コンテナ・ネットワークにデプロイします。コンテナ・ネットワークは Docker Engine の機能です。これは複数の Docker ホスト上を横断して複数のコンテナ間で通信を可能にします。

3.1.3 Swarm クラスタのアーキテクチャ

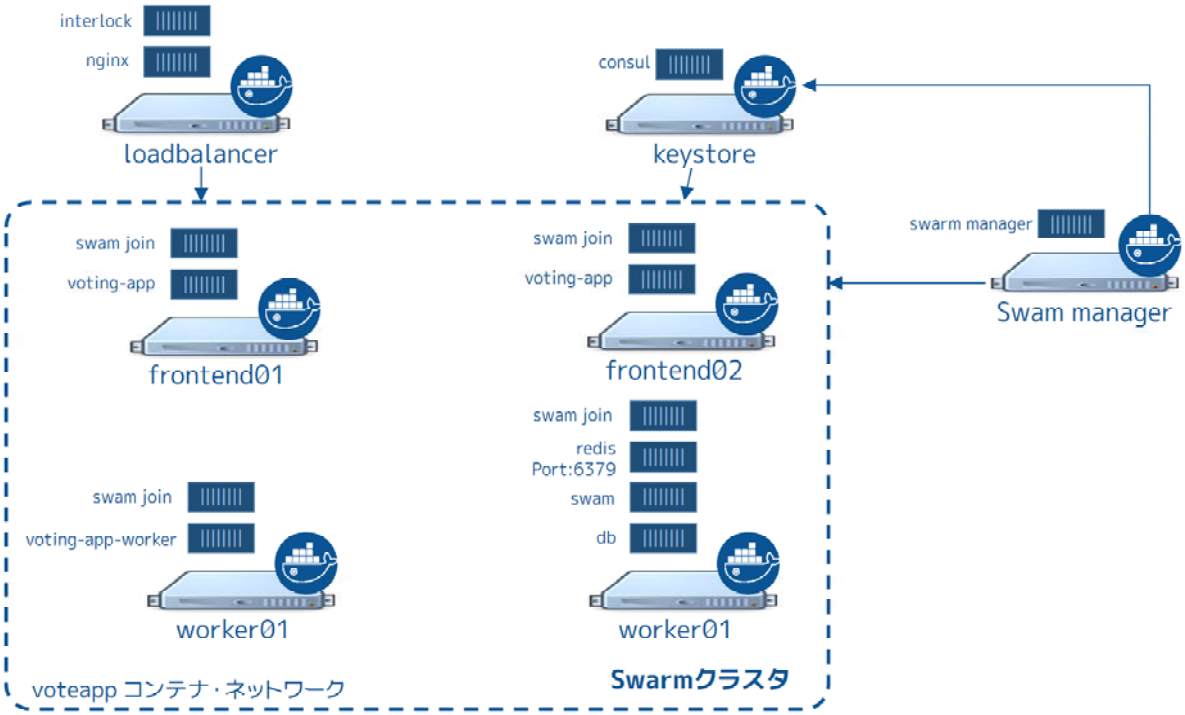
アプリケーションをサポートするのは、次の図のように、１つの Swarm マネージャと４つのノードで構成する

設計の Swarm クラスタです。



クラスタの４つのノード全てで Docker デーモンが動作します。Swarm マネージャと ロードバランサも同様です。Swarm マネージャはクラスタの一部であり、アプリケーションの範囲外であると考えます。１つのホスト上で Consul サーバはキーストア（keystore）として動作します。これは Swarm ディスカバリ用と、コンテナ・ネットワーク用の両方のためです。ロードバランサはクラスタ内に設置可能ですが、今回のサンプルでは扱いません。

サンプルとアプリケーションのデプロイを完了したら、皆さんの環境は下図のようになります。



この図にあるように、クラスタの各ノードでは次のコンテナを実行します。

frontend01：

- コンテナ：voitng-app（投票アプリ）
- コンテナ：Swarm エージェント

frontend02：

- コンテナ：voitng-app（投票アプリ）
- コンテナ：Swarm エージェント

worker01：

- コンテナ：voitng-app-worker（投票ワーカ・アプリ）
- コンテナ：Swarm エージェント

dbstore：

- コンテナ：voting-app-result-app（投票結果用アプリ）
- コンテナ：db (Postgres 9.4)
- コンテナ：redis
- コンテナ：Swarm エージェント

アプリケーションのデプロイ時、ローカル・システムを設定したら、ローカルのブラウザ上でアプリケーションをテスト可能です。プロダクションでは、もちろんこの手順は不要です。

次のステップ

これでアプリケーションのアーキテクチャを理解しました。デプロイするにあたり、どのようなネットワーク設定をサポートする必要があるのか理解したと思います。次のステップでは、このサンプルが使うネットワーク・インフラをデプロイします。

3.2 インフラのデプロイ

このステップでは、複数の Docker ホストを作成し、そこでアプリケーション・スタックを実行します。進む前に、アプリケーション・アーキテクチャを学ぶ必要があります。

3.2.1 構築手順について

以降のサンプルを実行する想定システムは、Mac あるいは Windows です。システム上で Docker Machine を経由して VirtualBox 仮想マシンをローカルにプロビジョニングし、Docker Engine の `docker` コマンドを使えるようにします。作業は 6 つの VirtualBox 仮想マシンをインストール予定です。

今回のサンプルでは Docker Machine を使いますが、任意のインフラ上で実行できます。希望するインフラ上であれば、どこでも環境を構築できる設計です。例えば、Azure や Digital Ocean などのようなパブリックのクラウド・プラットフォーム上で実行できるだけでなく、データセンタ上のオンプレミスや、ノート PC 上のテスト環境ですら動かせます。

なお、これらの手順では複数の値を設定するにあたり、一般的な `bash` コマンド代入技術を使います。次のコマンドを例に考えましょう。

```
$ eval $(docker-machine env keystore)
```

Windows 環境では、このような代入指定に失敗します。Widows 上で実行する場合は、この`$(docker-machine env keystore)`を実際の値に置き換えてください。

3.2.2 手順

タスク 1 ：keystore（キーストア）サーバの作成

Docker コンテナ・ネットワークと Swarm ディスカバリを有効化するために、キーバリュー・ストアのデプロイが必要です。キーストアはディスカバリ・バックエンドとして、Swarm マネージャが使うクラスタのメンバ一覧を常に更新し続けます。Swarm マネージャはこの一覧を使い、ノードにタスクを割り当てます。

オーバレイ・ネットワークはキーバリュー・ストアが必要です。キーバリュー・ストアはネットワーク状態を保持するために使います。ネットワーク状態には、ディスカバリ、ネットワーク、エンドポイント、IP アドレス等を含みます。

様々なバックエンドをサポートしています。今回のサンプルでは Consul コンテナ[*1] を使います。

1. `keystore` という名称の「マシン」を作成します。

```
$ docker-machine create -d virtualbox --virtualbox-memory "2000" \
--engine-opt="label=com.function=consul"  keystore
```

Engine デーモンにオプションを指定するには `--engine-opt` フラグを使います。Engine インスタンスをラベル付けするのに使います。

2. ローカルのシェルを `keystore` Docker ホストに接続します。

```
$ eval $(docker-machine env keystore)
```

*1 https://www.consul.io/

3. consul コンテナを起動します。

```
$ docker run --restart=unless-stopped -d -p 8500:8500 -h consul progrium/consul -server -bootstrap
```

-p フラグはコンテナ上のポート 8500 を公開します。これは Consul サーバがリッスンするためです。また、サーバ上では他のポートも公開します。確認するには docker ps コマンドを使います。

```
$ docker ps
CONTAINER ID        IMAGE               ...       PORTS
                            NAMES
372ffcbc96ed        progrium/consul     ...       53/tcp, 53/udp, 8300-8302/tcp, 8400/tcp,
8301-8302/udp, 0.0.0.0:8500->8500/tcp   dreamy_ptolemy
```

4. curl コマンドを使い、ノードが応答するかテストします。

```
$ curl $(docker-machine ip keystore):8500/v1/catalog/nodes
[{"Node":"consul","Address":"172.17.0.2"}]
```

タスク２：Swarm マネージャの作成

このステップでは、Swarm マネージャを作成し、keystore インスタンスに接続します。Swarm マネージャ・コンテナは Swarm クラスタの心臓部です。Docker コマンドを受け取り、クラスタに送り、クラスタ間のスケジューリングをする役割を持ちます。実際のプロダクションへのデプロイでは、高可用性(HA)のためにセカンダリの Swarm レプリカ・マネージャを設定すべきでしょう。

--eng-opt フラグを使い cluster-store と cluster-advertise オプションが keystore サーバを参照するようにします。これらのオプションは後にコンテナ・ネットワークの作成時に使います。

1. manager ホストを作成します。

```
$ docker-machine create -d virtualbox --virtualbox-memory "2000" \
  --engine-opt="label=com.function=manager" \
  --engine-opt="cluster-store=consul://$(docker-machine ip keystore):8500" \
  --engine-opt="cluster-advertise=eth1:2376" manager
```

デーモンに対して manager ラベルも指定します。

2. ローカルのシェルを manager Docker ホストに向けます。

```
$ eval $(docker-machine env manager)
```

3. Swarm マネージャのプロセスを開始します。

```
$ docker run --restart=unless-stopped -d -p 3376:2375 \
  -v /var/lib/boot2docker:/certs:ro \
  swarm manage --tlsverify \
  --tlscacert=/certs/ca.pem \
  --tlscert=/certs/server.pem \
  --tlskey=/certs/server-key.pem \
  consul://$(docker-machine ip keystore):8500
```

このコマンドは boot2docker.iso あるいはマネージャ用の TLS 証明書を作成します。これはクラスタ上の他マ

シンにマネージャが接続する時に使います。

4. ホスト上で Docker デーモンのログを参照し、正常に動いているか確認します。

```
$ docker-machine ssh manager
<-- 出力を省略 -->
docker@manager:~$ tail /var/lib/boot2docker/docker.log
time="2016-04-06T23:11:56.481947896Z" level=debug msg="Calling GET /v1.15/version"
time="2016-04-06T23:11:56.481984742Z" level=debug msg="GET /v1.15/version"
time="2016-04-06T23:12:13.070231761Z" level=debug msg="Watch triggered with 1 nodes" discovery=consul
time="2016-04-06T23:12:33.069387215Z" level=debug msg="Watch triggered with 1 nodes" discovery=consul
time="2016-04-06T23:12:53.069471308Z" level=debug msg="Watch triggered with 1 nodes" discovery=consul
time="2016-04-06T23:13:13.069512320Z" level=debug msg="Watch triggered with 1 nodes" discovery=consul
time="2016-04-06T23:13:33.070021418Z" level=debug msg="Watch triggered with 1 nodes" discovery=consul
time="2016-04-06T23:13:53.069395005Z" level=debug msg="Watch triggered with 1 nodes" discovery=consul
time="2016-04-06T23:14:13.071417551Z" level=debug msg="Watch triggered with 1 nodes" discovery=consul
time="2016-04-06T23:14:33.069843647Z" level=debug msg="Watch triggered with 1 nodes" discovery=consul
```

出力内容から consul と manager が正常に通信できているのが分かります。

5. Docker ホストから抜けます。

```
docker@manager:~$ exit
```

タスク３：ロードバランサの追加

Interlock[*1] アプリケーションと Nginx をロードバランサとして使います。ロードバランサ用のホストを作る前に、Nginx で使う設定を作成します。

1. ローカルホスト上に config ディレクトリを作成します。

2. config ディレクトリに変更します。

```
$ cd config
```

3. Swarm マネージャ・ホストの IP アドレスを取得します。

例：

```
$ docker-machine ip manager
192.168.99.101
```

4. 任意のエディタで config.toml ファイルを作成し、次の内容をファイルに書き込みます。

*1 https://github.com/ehazlett/interlock

```
ListenAddr = ":8080"
DockerURL = "tcp://SWARM_MANAGER_IP:3376"
TLSCACert = "/var/lib/boot2docker/ca.pem"
TLSCert = "/var/lib/boot2docker/server.pem"
TLSKey = "/var/lib/boot2docker/server-key.pem"

[[Extensions]]
Name = "nginx"
ConfigPath = "/etc/conf/nginx.conf"
PidPath = "/etc/conf/nginx.pid"
MaxConn = 1024
Port = 80
```

5. 設定ファイルにおいて、SWARM_MANAGE_IP は手順３で取得した manager の IP アドレスに書き換えてください。

この値はロードバランサがマネージャのイベント・ストリームを受信するために使います。

6. config.toml ファイルを保存して閉じます。

7. ロードバランサ用にマシンを作成します。

```
$ docker-machine create -d virtualbox --virtualbox-memory "2000" \
  --engine-opt="label=com.function=interlock" loadbalancer
```

8. 環境を loadbalancer に切り替えます。

```
$ eval $(docker-machine env loadbalancer)
```

9. interlock コンテナを起動します。

```
$ docker run \
    -P \
    -d \
    -ti \
    -v nginx:/etc/conf \
    -v /var/lib/boot2docker:/var/lib/boot2docker:ro \
    -v /var/run/docker.sock:/var/run/docker.sock \
    -v $(pwd)/config.toml:/etc/config.toml \
    --name interlock \
    ehazlett/interlock:1.0.1 \
    -D run -c /etc/config.toml
```

このコマンドは現在のディレクトリにある config.toml ファイルを読み込みます。コマンド実行後、イメージを実行しているのを確認します。

```
$ docker ps
CONTAINER ID        IMAGE                         COMMAND                  CREATED             STATUS
          PORTS                      NAMES
d846b801a978        ehazlett/interlock:1.0.1   "/bin/interlock -D ru"   2 minutes ago       Up 2 minutes
        0.0.0.0:32770->8080/tcp   interlock
```

イメージが実行中でなければ、docker ps -a を実行してシステム上で起動した全てのイメージを表示します。そして、コンテナが起動に失敗していれば、ログを取得できます。

```
$ docker logs interlock
INFO[0000] interlock 1.0.1 (000291d)
DEBU[0000] loading config from: /etc/config.toml
FATA[0000] read /etc/config.toml: is a directory
```

このエラーであれば、通常は config.toml ファイルがある同じ config ディレクトリ内で docker run を実行したのが原因でしょう。コマンドを実行し、次のような衝突が表示する場合があります。

```
docker: Error response from daemon: Conflict. The name "/interlock" is already in use by container
d846b801a978c76979d46a839bb05c26d2ab949ff9f4f740b06b5e2564bae958. You have to remove (or rename) that
container to be able to reuse that name.
```

このような時は、docker rm interlock で interlock コンテナを削除し、再度試みてください。

10. ロードバランサ上で nginx コンテナを起動します。

```
$ docker run -ti -d \
  -p 80:80 \
  --label interlock.ext.name=nginx \
  --link=interlock:interlock \
  -v nginx:/etc/conf \
  --name nginx \
  nginx nginx -g "daemon off;" -c /etc/conf/nginx.conf
```

タスク４：他の Swarm ノードを作成

Swarm クラスタのホストを「ノード」と呼びます。既にマネージャ・ノードを作成しました。ここでの作業は、各ノード用の仮想ホストを作成します。３つのコマンドが必要です。

- Docker Machine でホストを作成
- ローカル環境から新しい環境に切り替え
- ホストを Swarm クラスタに追加

Mac あるいは Windows 以外で構築している場合、swarm ノードに追加するには join コマンドを実行するだけです。それだけで Consul ディスカバリ・サービスに登録します。また、ノードの作成時には次の例のようにラベルを付けます。

```
--engine-opt="label=com.function=frontend01"
```

これらのラベルはアプリケーション・コンテナを開始した後に使います。以降のコマンドで、各ノードに対してラベルを適用します。

1. frontend01 ホストを作成し、Swarm クラスタに追加します。

```
$ docker-machine create -d virtualbox --virtualbox-memory "2000" \
  --engine-opt="label=com.function=frontend01" \
  --engine-opt="cluster-store=consul://$(docker-machine ip keystore):8500" \
  --engine-opt="cluster-advertise=eth1:2376" frontend01
$ eval $(docker-machine env frontend01)
$ docker run -d swarm join --addr=$(docker-machine ip frontend01):2376 \
```

```
    consul://$(docker-machine ip keystore):8500
```

2. frontend02 仮想マシンを作成します。

```
$ docker-machine create -d virtualbox --virtualbox-memory "2000" \
  --engine-opt="label=com.function=frontend02" \
  --engine-opt="cluster-store=consul://$(docker-machine ip keystore):8500" \
  --engine-opt="cluster-advertise=eth1:2376" frontend02
$ eval $(docker-machine env frontend02)
$ docker run -d swarm join --addr=$(docker-machine ip frontend02):2376 \
  consul://$(docker-machine ip keystore):8500
```

3. worker01 仮想マシンを作成します。

```
$ docker-machine create -d virtualbox --virtualbox-memory "2000" \
  --engine-opt="label=com.function=worker01" \
  --engine-opt="cluster-store=consul://$(docker-machine ip keystore):8500" \
  --engine-opt="cluster-advertise=eth1:2376" worker01
$ eval $(docker-machine env worker01)
$ docker run -d swarm join --addr=$(docker-machine ip worker01):2376 \
  consul://$(docker-machine ip keystore):8500
```

4. dbstore 仮想マシンを作成します。

```
$ docker-machine create -d virtualbox --virtualbox-memory "2000" \
  --engine-opt="label=com.function=dbstore" \
  --engine-opt="cluster-store=consul://$(docker-machine ip keystore):8500" \
  --engine-opt="cluster-advertise=eth1:2376" dbstore
$ eval $(docker-machine env dbstore)
$ docker run -d swarm join --addr=$(docker-machine ip dbstore):2376 \
  consul://$(docker-machine ip keystore):8500
```

5. 動作確認をします。

この時点では、アプリケーションが必要なインフラをデプロイ完了しました。テストは、次のようにマシンが実行しているか一覧表示します。

```
$ docker-machine ls
NAME           ACTIVE   DRIVER       STATE     URL                         SWARM   DOCKER    ERRORS
dbstore        -        virtualbox   Running   tcp://192.168.99.111:2376           v1.10.3
frontend01     -        virtualbox   Running   tcp://192.168.99.108:2376           v1.10.3
frontend02     -        virtualbox   Running   tcp://192.168.99.109:2376           v1.10.3
keystore       -        virtualbox   Running   tcp://192.168.99.100:2376           v1.10.3
loadbalancer   -        virtualbox   Running   tcp://192.168.99.107:2376           v1.10.3
manager        -        virtualbox   Running   tcp://192.168.99.101:2376           v1.10.3
worker01       *        virtualbox   Running   tcp://192.168.99.110:2376           v1.10.3
```

6. Swarm マネージャが全てのノードを一覧表示するのを確認します。

```
$ docker -H $(docker-machine ip manager):3376 info
Containers: 4
 Running: 4
 Paused: 0
 Stopped: 0
```

```
Images: 3
Server Version: swarm/1.1.3
Role: primary
Strategy: spread
Filters: health, port, dependency, affinity, constraint
Nodes: 4
 dbstore: 192.168.99.111:2376
   └ Status: Healthy
   └ Containers: 1
   └ Reserved CPUs: 0 / 1
   └ Reserved Memory: 0 B / 2.004 GiB
   └ Labels: com.function=dbstore, executiondriver=native-0.2, kernelversion=4.1.19-boot2docker,
operatingsystem=Boot2Docker 1.10.3 (TCL 6.4.1); master : 625117e - Thu Mar 10 22:09:02 UTC 2016,
provider=virtualbox, storagedriver=aufs
   └ Error: (none)
   └ UpdatedAt: 2016-04-07T18:25:37Z
 frontend01: 192.168.99.108:2376
   └ Status: Healthy
   └ Containers: 1
   └ Reserved CPUs: 0 / 1
   └ Reserved Memory: 0 B / 2.004 GiB
   └ Labels: com.function=frontend01, executiondriver=native-0.2, kernelversion=4.1.19-boot2docker,
operatingsystem=Boot2Docker 1.10.3 (TCL 6.4.1); master : 625117e - Thu Mar 10 22:09:02 UTC 2016,
provider=virtualbox, storagedriver=aufs
   └ Error: (none)
   └ UpdatedAt: 2016-04-07T18:26:10Z
 frontend02: 192.168.99.109:2376
   └ Status: Healthy
   └ Containers: 1
   └ Reserved CPUs: 0 / 1
   └ Reserved Memory: 0 B / 2.004 GiB
   └ Labels: com.function=frontend02, executiondriver=native-0.2, kernelversion=4.1.19-boot2docker,
operatingsystem=Boot2Docker 1.10.3 (TCL 6.4.1); master : 625117e - Thu Mar 10 22:09:02 UTC 2016,
provider=virtualbox, storagedriver=aufs
   └ Error: (none)
   └ UpdatedAt: 2016-04-07T18:25:43Z
 worker01: 192.168.99.110:2376
   └ Status: Healthy
   └ Containers: 1
   └ Reserved CPUs: 0 / 1
   └ Reserved Memory: 0 B / 2.004 GiB
   └ Labels: com.function=worker01, executiondriver=native-0.2, kernelversion=4.1.19-boot2docker,
operatingsystem=Boot2Docker 1.10.3 (TCL 6.4.1); master : 625117e - Thu Mar 10 22:09:02 UTC 2016,
provider=virtualbox, storagedriver=aufs
   └ Error: (none)
   └ UpdatedAt: 2016-04-07T18:25:56Z
Plugins:
 Volume:
 Network:
Kernel Version: 4.1.19-boot2docker
Operating System: linux
Architecture: amd64
CPUs: 4
Total Memory: 8.017 GiB
Name: bb13b7cf80e8
```

このコマンドは Swarm ポートに対して処理しているため、クラスタ全体の情報を返します。操作対象は Swarm マネージャあり、ノードではありません。

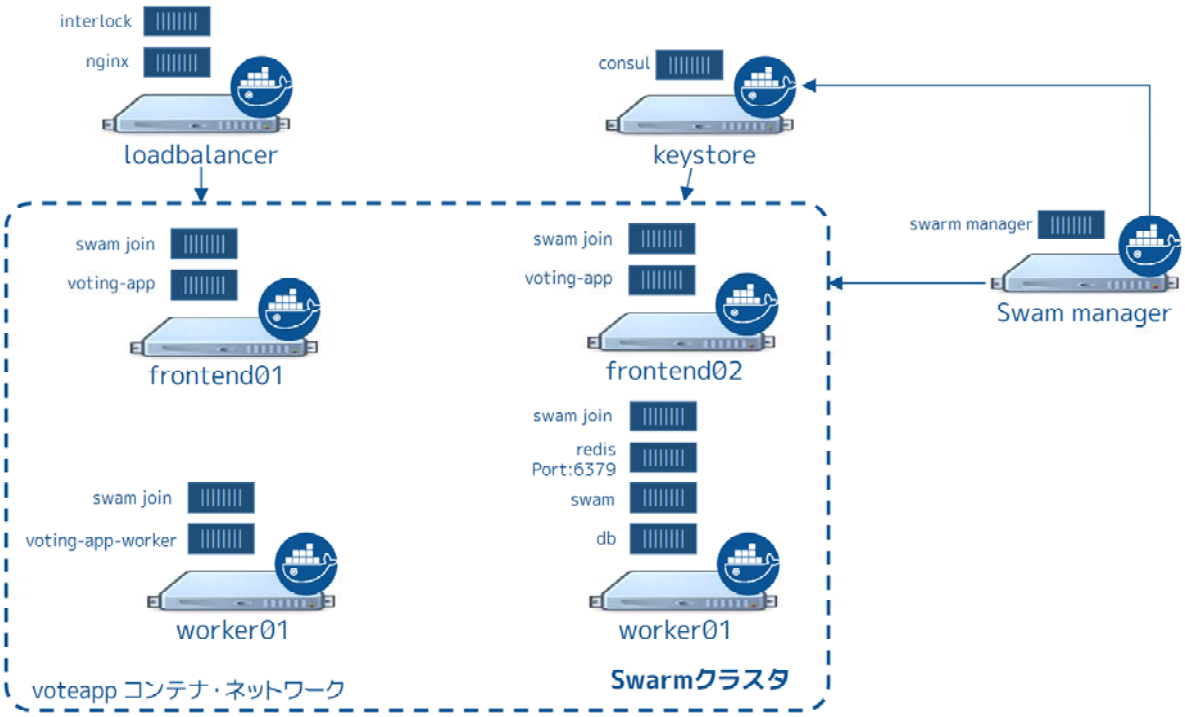
次のステップ

キーストア、ロードバランサ、Swarm クラスタのインフラが動きました。これで投票アプリケーションの構築と実行ができます。

3.3 アプリケーションのデプロイ

これまでロードバランサ、ディスカバリ・バックエンド、Swarm クラスタをデプロイ しました。次は投票アプリケーションをデプロイしましょう。ここで「Docker 化したアプリケーション」を起動し始めます。

次の図は最終的なアプリケーション設定であり、voteapp オーバレイ・コンテナ・ネットワークも含みます。



この手順ではコンテナをネットワークに接続します。この voteapp ネットワークは Consul ディスカバリ・バックエンドを使う全ての Docker ホスト上で利用可能です。interlock、nginx、consul、swarm manager コンテナは voteapp オーバレイ・コンテナ・ネットワークの一部なのでご注意ください。

3.3.1 手順

タスク１：ボリュームとネットワークのセットアップ

このアプリケーションはオーバレイ・コンテナ・ネットワークとコンテナ・ボリュームに依存します。Docker Engine は２つの機能を提供します。Swarm manager インスタンス上でどちらも作成可能です。

1. ローカル環境を Swarm manager ホストに向けます。

```
$ eval $(docker-machine env manager)
```

クラスタ・ノード上でネットワークを作成したら、ネットワーク全体で参照可能になります。

2. voteapp コンテナ・ネットワークを作成します。

```
$ docker network create -d overlay voteapp
```

3. db ストアに切り替えます。

```
$ eval $(docker-machine env dbstore)
```

4. db ストア・ノードで新しいネットワークを確認します。

```
$ docker network ls
NETWORK ID          NAME                DRIVER
e952814f610a        voteapp             overlay
1f12c5e7bcc4        bridge              bridge
3ca38e887cd8        none                null
3da57c44586b        host                host
```

5. db-data という名称のコンテナ・ボリュームを作成します。

```
$ docker volume create --name db-data
```

タスク２：コンテナ化したマイクロサービスの起動

この時点で、マイクロサービスのコンポーネントを起動し、アプリケーションを起動する準備が整いました。アプリケーション・コンテナによっては、Docker Hub にある既存イメージを直接ダウンロードして実行できます。その他、自分でカスタマイズしたイメージを実行したい場合は、構築する必要があります。以下はどのコンテナがカスタム・イメージを使っているか、使っていないかの一覧です。

- ロードバランサ・コンテナ：既存イメージ（ehazlett/interlock）
- Redis コンテナ：既存イメージ（公式 redis イメージ）
- Postgres (PostgreSQL)コンテナ：既存イメージ（公式 postgres イメージ）
- Web コンテナ：カスタム構築イメージ
- Worker コンテナ：カスタム構築イメージ
- Results コンテナ：カスタム構築イメージ

このセクションではクラスタ上のホストに対して、コマンドでこれらコンテナを起動します。 Swarm マネージャに対して命令するためには、各コマンドで -H フラグを使います。

コマンドには -e も含みます。これは Swarm に**制限（constraint）**を指定するためです。制限はマネージャに対して、function（機能）のラベルに一致するノードの指定で使います。ラベルはノードを作成時に設定します。以降のコマンド実行時に、制約の値を確認します。

1. Postgres データベース・コンテナを起動します。

```
$ docker -H $(docker-machine ip manager):3376 run -t -d \
  -v db-data:/var/lib/postgresql/data \
  -e constraint:com.function==dbstore \
  --net="voteapp" \
  --name db postgres:9.4
```

2. Redis コンテナを起動します。

```
$ docker -H $(docker-machine ip manager):3376 run -t -d \
  -p 6379:6379 \
  -e constraint:com.function==dbstore \
  --net="voteapp" \
  --name redis redis
```

redis の名前は重要なため、変更しないでください。

3. ワーカ・アプリケーションを起動します。

```
$ docker -H $(docker-machine ip manager):3376 run -t -d \
  -e constraint:com.function==worker01 \
  --net="voteapp" \
  --net-alias=workers \
  --name worker01 docker/example-voting-app-worker
```

4. results アプリケーションを起動します。

```
$ docker -H $(docker-machine ip manager):3376 run -t -d \
  -p 80:80 \
  --label=interlock.hostname=results \
  --label=interlock.domain=myenterprise.com \
  -e constraint:com.function==dbstore \
  --net="voteapp" \
  --name results-app docker/example-voting-app-result-app
```

5. 各フロントエンド・ノード上に、２つの投票アプリケーションを起動します。

```
$ docker -H $(docker-machine ip manager):3376 run -t -d \
  -p 80:80 \
  --label=interlock.hostname=vote \
  --label=interlock.domain=myenterprise.com \
  -e constraint:com.function==frontend01 \
  --net="voteapp" \
  --name voting-app01 docker/example-voting-app-voting-app
```

そして、別のフロントエンド・ノード上で実行します。

```
$ docker -H $(docker-machine ip manager):3376 run -t -d \
  -p 80:80 \
  --label=interlock.hostname=vote \
  --label=interlock.domain=myenterprise.com \
  -e constraint:com.function==frontend02 \
  --net="voteapp" \
  --name voting-app02 docker/example-voting-app-voting-app
```

タスク３：作業内容の確認と/etc/hosts の更新

このステップでは、Nginx コンテナの設定が適切に行われているかを確認します。ロードバランサの動作確認のため、ローカルの /etc/hosts ファイルを変更します。

1. loadbalancer ノードに変更します。

```
$ eval $(docker-machine env loadbalancer)
```

2. nginx の設定を表示し、内容を確認します。

```
$ docker exec interlock cat /etc/conf/nginx.conf
```

```
... 出力を省略 ...

upstream results.myenterprise.com {
    zone results.myenterprise.com_backend 64k;

    server 192.168.99.111:80;

}
server {
    listen 80;

    server_name results.myenterprise.com;

    location / {
        proxy_pass http://results.myenterprise.com;
    }
}
upstream vote.myenterprise.com {
    zone vote.myenterprise.com_backend 64k;

    server 192.168.99.109:80;
    server 192.168.99.108:80;

}
server {
    listen 80;

    server_name vote.myenterprise.com;

    location / {
        proxy_pass http://vote.myenterprise.com;
    }
}

include /etc/conf/conf.d/*.conf;
}
```

http://vote.myenterprise.com サイトの設定は、どちらかのフロントエンド・ノードを指し示します。http://results.myenterprise.com にリクエストしたら、example-voting-app-result-app が稼働している dbstore ノードに移動します。

1. ローカルホスト上で /etc/hosts ファイルを編集し、これらサイトの名前解決の行を追加します。

2. /etc/hosts ファイルを保存して閉じます。

3. nginx コンテナの再起動。

現在の Interlock サーバの設定が Nginx の設定を反映していません。そのため、手動で再起動の必要があります。

```
$ docker restart nginx
```

タスク４：アプリケーションのテスト

これでアプリケーションをテストできます。

1. ブラウザを開き、サイト http://vote.myenterprise.com に移動します。

投票ページ「Cats vs Dogs!」が画面に表示されます。

2. ２つの選択肢のうち、どちらかに投票します。

3. サイト http://results.myenterprise.com に移動し、結果を表示します。

4. 他の選択肢に投票します。

投票した結果が画面上に表示されます。

追加作業：Docker Compose でデプロイ

これまでは、各アプリケーションのコンテナを個々に起動しました。しかし、複数コンテナの起動や依存関係の順番に従った起動は、とても煩雑です。例えば、データベースはワーカが起動する前に動いているべきでしょう。

Docker Compose はマイクロサービス・コンテナと依存関係を Compose ファイルで定義します。そして、Compose ファイルを使って全てのコンテナを一斉に起動します。これは追加作業（extra credit）です。

1. 始める前に、起動した全てのコンテナを停止します。

    a. （作業対象の）ホストをマネージャに向けます。

        $ DOCKER_HOST=$(docker-machine ip manager):3376

    b. Swarm 上のアプリケーション全てを一覧します。

    c. 各コンテナを停止・削除します。

2. このチュートリアルに従って、自分で Compose ファイルの作成を試みます。

Compose ファイルはバージョン２形式を使うのがベストです。各 docker run コマンドを docker-compose.yml ファイル内のサービスに置き換えます。例えば、次のコマンドがあります。

```
$ docker -H $(docker-machine ip manager):3376 run -t -d \
  -e constraint:com.function==worker01 \
  --net="voteapp" \
  --net-alias=workers \
  --name worker01 docker/example-voting-app-worker
```

これは、次の Compose ファイルに書き換え可能です。

```
  worker:
    image: docker/example-voting-app-worker
    networks:
      voteapp:
        aliases:
        - workers
```

通常、Compose はファイルに現れる逆順でサービスの起動を試みます。そのため、あるサービスを他のサービスよりも前に実行するには、ファイル中の最後尾にサービスを記述する必要があります。アプリケーションがボリュームやネットワークを使う場合は、ファイルの末尾で宣言します。

3. 結果がファイル[*1] と一致しているか確認します。

4. 問題が無ければ、システム上に docker-compose.yml ファイルを保存します。

5. DOCKER_HOST を Swarm マネージャに向けます。

```
$ DOCKER_HOST=$(docker-machine ip manager):3376
```

6. docker-compose.yml と同じディレクトリで、サービスを起動します。

```
$ docker-compose up -d
Creating network "scale_voteapp" with the default driver
Creating volume "scale_db-data" with default driver
Pulling db (postgres:9.4)...
worker01: Pulling postgres:9.4... : downloaded
dbstore: Pulling postgres:9.4... : downloaded
frontend01: Pulling postgres:9.4... : downloaded
frontend02: Pulling postgres:9.4... : downloaded
Creating db
Pulling redis (redis:latest)...
dbstore: Pulling redis:latest... : downloaded
frontend01: Pulling redis:latest... : downloaded
frontend02: Pulling redis:latest... : downloaded
worker01: Pulling redis:latest... : downloaded
Creating redis
Pulling worker (docker/example-voting-app-worker:latest)...
dbstore: Pulling docker/example-voting-app-worker:latest... : downloaded
frontend01: Pulling docker/example-voting-app-worker:latest... : downloaded
frontend02: Pulling docker/example-voting-app-worker:latest... : downloaded
worker01: Pulling docker/example-voting-app-worker:latest... : downloaded
Creating scale_worker_1
Pulling voting-app (docker/example-voting-app-voting-app:latest)...
dbstore: Pulling docker/example-voting-app-voting-app:latest... : downloaded
frontend01: Pulling docker/example-voting-app-voting-app:latest... : downloaded
frontend02: Pulling docker/example-voting-app-voting-app:latest... : downloaded
worker01: Pulling docker/example-voting-app-voting-app:latest... : downloaded
Creating scale_voting-app_1
Pulling result-app (docker/example-voting-app-result-app:latest)...
dbstore: Pulling docker/example-voting-app-result-app:latest... : downloaded
```

*1 https://docs.docker.com/swarm/swarm_at_scale/docker-compose.yml

```
frontend01: Pulling docker/example-voting-app-result-app:latest... : downloaded
frontend02: Pulling docker/example-voting-app-result-app:latest... : downloaded
worker01: Pulling docker/example-voting-app-result-app:latest... : downloaded
Creating scale_result-app_1
```

7. docker ps コマンドで Swarm クラスタ上のコマンドを確認します。

```
$ docker -H $(docker-machine ip manager):3376 ps
CONTAINER ID        IMAGE                                      COMMAND                  CREATED
  STATUS              PORTS                                NAMES
b71555033caa        docker/example-voting-app-result-app   "node server.js"         6 seconds ago
 Up 4 seconds        192.168.99.104:32774->80/tcp     frontend01/scale_result-app_1
cf29ea21475d        docker/example-voting-app-worker       "/usr/lib/jvm/java-7-"   6 seconds ago
 Up 4 seconds                                         worker01/scale_worker_1
98414cd40ab9        redis                                  "/entrypoint.sh redis"   7 seconds ago
 Up 5 seconds        192.168.99.105:32774->6379/tcp   frontend02/redis
1f214acb77ae        postgres:9.4                           "/docker-entrypoint.s"   7 seconds ago
 Up 5 seconds        5432/tcp                         frontend01/db
1a4b8f7ce4a9        docker/example-voting-app-voting-app   "python app.py"          7 seconds ago
 Up 5 seconds        192.168.99.107:32772->80/tcp     dbstore/scale_voting-app_1
```

サービスを手動で起動した時は、voting-app インスタンスは２つのフロントエンド・ノード上で動作していました。今回はいくつ起動していますか？

8. アプリケーションをスケールするため、voting-app インスタンスを追加します。

```
$ docker-compose scale voting-app=3
Creating and starting 2 ... done
Creating and starting 3 ... done
```

スケールアップ後は、クラスタ上のコンテナ一覧を再び表示します。

9. loadbalancer ノードに変更します。

```
$ eval $(docker-machine env loadbalancer)
```

10. Nginx サーバを再起動します。

```
$ docker restart nginx
```

11. http://vote.myenterprise.com と http://results.myenterprise.com を再び表示して、投票の動作を確認します。

12. 各コンテナのログを表示できます。

```
$ docker logs scale_voting-app_1
 * Running on http://0.0.0.0:80/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger pin code: 285-809-660
192.168.99.103 - - [11/Apr/2016 17:15:44] "GET / HTTP/1.0" 200 -
192.168.99.103 - - [11/Apr/2016 17:15:44] "GET /static/stylesheets/style.css HTTP/1.0" 304 -
192.168.99.103 - - [11/Apr/2016 17:15:45] "GET /favicon.ico HTTP/1.0" 404 -
```

```
192.168.99.103 - - [11/Apr/2016 17:22:24] "POST / HTTP/1.0" 200 -
192.168.99.103 - - [11/Apr/2016 17:23:37] "POST / HTTP/1.0" 200 -
192.168.99.103 - - [11/Apr/2016 17:23:39] "POST / HTTP/1.0" 200 -
192.168.99.103 - - [11/Apr/2016 17:23:40] "POST / HTTP/1.0" 200 -
192.168.99.103 - - [11/Apr/2016 17:23:41] "POST / HTTP/1.0" 200 -
192.168.99.103 - - [11/Apr/2016 17:23:43] "POST / HTTP/1.0" 200 -
192.168.99.103 - - [11/Apr/2016 17:23:44] "POST / HTTP/1.0" 200 -
192.168.99.103 - - [11/Apr/2016 17:23:46] "POST / HTTP/1.0" 200 -
```

このログは、ある投票アプリケーション・コンテナの状況を表示しています。

次のステップ

おめでとうございます。マイクロサービスをベースとしたアプリケーションを Swarm クラスタ上に手動でデプロイできました。もちろん、全てが上手く行くとは限りません。どのようにスケールするアプリケーションをデプロイするかを学びましたので、次は Swarm クラスタ上で大規模アプリケーション実行時のトラブルシューティングを学ぶべきでしょう。

3.4 アプリケーションのトラブルシュート

何事も失敗するのが、人生における現実です。これを前提に、障害が発生したらどのように対処すべきか考えるのは重要です。以下のセクションでは、いくつかの障害シナリオを扱います。

- Swarm マネージャ障害
- Consul（ディスカバリ・バックエンド）障害
- Interlock ロードバランサ障害
- ウェブ（web-vote-app）障害
- Redis 障害
- ワーカ（vote-worker）障害
- Postgres 障害
- results-app 障害
- インフラ障害

3.4.1 Swarm マネージャ障害

現在の設定では、Swarm クラスタには 1 つのホスト上で 1 つのマネージャ用コンテナしかありません。コンテナが終了したりノード障害が発生したりしますと、クラスタを管理できなくなるだけでなく、修復や入れ替えも不可能になります。

Swarm マネージャ・コンテナが予期せず終了して障害になった場合、Docker は自動的にコンテナの再起動を試みます。これはコンテナ起動時に `--restart=unless-stopped` に切り替える設定をしたからです。

Swarm マネージャが利用不可能になれば、アプリケーションは現状の設定で実行し続けます。しかし、Swarm マネージャが使えるようになるまで、ノードやコンテナをプロビジョンできなくなります。

Docker Swarm は Swarm マネージャの高可用性をサポートしています。そのため、1 つのクラスタ上に 2 つ以上のマネージャを追加可能です。あるマネージャがプライマリ・マネージャとして選ばれたら、その他のものはセカンダリになります。プライマリ・マネージャで障害が発生したら、他のセカンダリから新しいプライマリ・マネージャが選び出され、クラスタの操作をし続けることが可能になります。高可用性に対応した Swarm マネージャを複数デプロイする場合は、インフラ上で複数の領域を横断する障害発生を考慮したほうが良いでしょう。

3.4.2 Consul（ディスカバリ・バックエンド）障害

この Swarm クラスタは、クラスタのディスカバリ・サービスとして、1 つのノード上で Consul コンテナを 1 つデプロイしました。このセットアップ方法では、もし Consul コンテナを終了するかノード障害が発生しても、現在の状態のままアプリケーションを実行できるかもしれません。しかしながら、クラスタ管理に関する処理は障害になります。障害範囲は、クラスタ上に新しいコンテナの登録ができなくなり、クラスタに対して設定確認も行えません。

`consul` コンテナが予期せず終了して障害になった場合、Docker は自動的にコンテナの再起動を試みます。これはコンテナ起動時に `--restart=unless-stopped` に切り替える設定をしたからです。

`consul`、`etcd`、`Zookeeper` の各ディスカバリ・サービス・バックエンドは、様々な高可用性のオプションをサポートしています。これらには Paxos/Raft クォーラムが組み込まれています。高可用性に対応した設定をするには、あなたが選んだディスカバリ・サービス・バックエンドに対する既存のベスト・プラクティスを確認すべきです。高可用性のために複数のディスカバリ・サービス・バックエンドをデプロイするのであれば、インフラ上で複数の領域に横断した障害発生への対処を考えるべきです。

1つのディスカバリ・バックエンド・サービスで Swarm クラスタを操作する場合、このサービスが停止すると修復不可能になります。そのような場合は、新しいディスカバリ・バックエンド用のインスタンスを起動し直し、クラスタの各ノード上で Swarm エージェントを実行し直す必要があります。

3.4.3 障害の取り扱い

コンテナの障害には様々な理由が考えられます。しかしながら、Swarm はコンテナで障害が発生しても再起動を試みません。

コンテナの障害発生時、自動的に再起動する方法の1つは、コンテナ起動時に `--restart=unless-stopped` フラグを付けることです。これはローカルの Docker デーモンに対して、コンテナで不意な終了が発生した場合に再起動するよう命令します。これが正常に機能するのは、コンテナを実行するノードと Docker デーモンが正常に稼働し続ける状況のみです。コンテナを実行するホスト自身で障害が発生したら、コンテナを再起動できません。あるいは、Docker デーモン自身が障害となっているでしょう。

別の方法としては、外部のツール（クラスタ外にあるツール）を使ってアプリケーションの状態を監視し、適切なサービス・レベルを維持する方法があります。サービス・レベルとは「少なくともウェブサーバのコンテナ 10 動かす」といったものです。このシナリオでは、ウェブ・コンテナの実行数が 10 以下になれば、ツールが何らかの方法で足りない数だけコンテナの起動を試みます。

今回のサンプル投票アプリケーションでは、フロントエンドはロードバランサがあるためスケーラブル（スケール可能）です。2つのウェブ・コンテナで障害が発生すると（あるいは実行している AWS ホスト自身での障害が起これば)、ウェブ・コンテナに対するリクエストがあってもロードバランサは障害対象へのルーティングを停止し、別の経路に振り分けできます。ロードバランサの背後に n 個のウェブ・コンテナを起動できますので、この方法は高い拡張性を持つと言えるでしょう。

3.4.4 Interlock ロードバランサ障害

今回の例では、1つのノード上で1つの interlock ロードバランサを実行する環境を構築しました。このセットアップ方法では、コンテナが終了するかノード障害が発生したら、アプリケーションはサービスに対するリクエストを受け付けできなくなり、アプリケーションが利用不可能になります。

`interlock` コンテナが不意に終了すると障害になり、Docker は自動的に再起動を試みます。これはコンテナ起動時に `--restart=unless-stopped` フラグを付け付けたからです。

高可用性のある Interlock ロードバランサを構築可能です。複数のノード上に複数の Interlock コンテナを実行する方法があります。後は DNS ラウンドロビンの使用や、その他の技術により、Interlock コンテナに対するアクセスを負荷分散します。この方法であれば、1つの Interlock コンテナやノードがダウンしたとしても、他のサービスがリクエストを処理し続けます。

複数の Interlock ロードバランサをデプロイする場合は、インフラ上で複数の領域に横断した障害発生への対処を考えるべきです。

3.4.5 ウェブ（web-vote-app）障害

今回の環境では、2つのノードで2つのウェブ投票用コンテナを実行するように設定しました。これらは Interlock ロードバランサの背後にあるため、受信した接続は両者にまたがって分散されます。

もし1つのウェブコンテナもしくはノードで障害が発生しても、ロードバランサは生存しているコンテナに全てのトラフィックを流し続けますので、サービスは継続します。障害のあったインスタンスが復旧するか、あるいは追加した所に切り替えれば、受信したリクエストを適切に処理するようロードバランサの設定を変更します。

最も高い可用性を考えるのであれば、2つのフロントエンド・ウェブ・サービス（`frontend01` と `frontend02`）をインフラ上の異なった障害ゾーンへデプロイすることになるでしょう。あるいは、更なるデプロイの検討も良いかもしれません。

3.4.6 Redis 障害

redis コンテナで障害が発生したら、一緒に動作している web-vote-app コンテナも正常に機能しなくなります。一番良い方法は対象インスタンスの正常性を監視するよう設定し、各 Redis インスタンスに対して正常な書き込みができるかどうか確認することです。もし問題のある redis インスタンスが発見されれば、web-vote-app と redis の連係を切り離し、復旧作業にあたるべきです。

3.4.7 ワーカ（vote-worker）障害

ワーカ・コンテナが終了するか、実行しているノードで障害が発生したら、redis コンテナは worker コンテナが復旧するまで投票キューを保持します。ワーカが復旧するまでその状態が維持され、投票も継続できます。

もし worker01 コンテナが不意に停止して障害になれば、Docker は自動的に再起動を試みます。これはコンテナ起動時に --restart=unless-stopped フラグを付けたからです。

3.4.8 Postgres 障害

今回のアプリケーションでは HA や Postgres のレプリケーションを実装していません。つまり Postgres コンテナの喪失とは、アプリケーションの障害だけでなく、データの損失または欠損を引き起こす可能性があります。何らかの Postgres HA やレプリケーションのような実装をすることが望ましい解決策です。

3.4.9 results-app 障害

results-app コンテナが終了したら、コンテナが復旧するまで結果をブラウザで表示できなくなります。それでも投票データを集めてカウントを継続できるため、復旧は純粋にコンテナを立ち上げるだけで済みます。

results-app コンテナは起動時に --restart=unless-stopped フラグを付けています。つまり Docker デーモンは自動的にコンテナの再起動を試みます。たとえそれが管理上の停止だったとしてもです。

3.4.10 インフラ障害

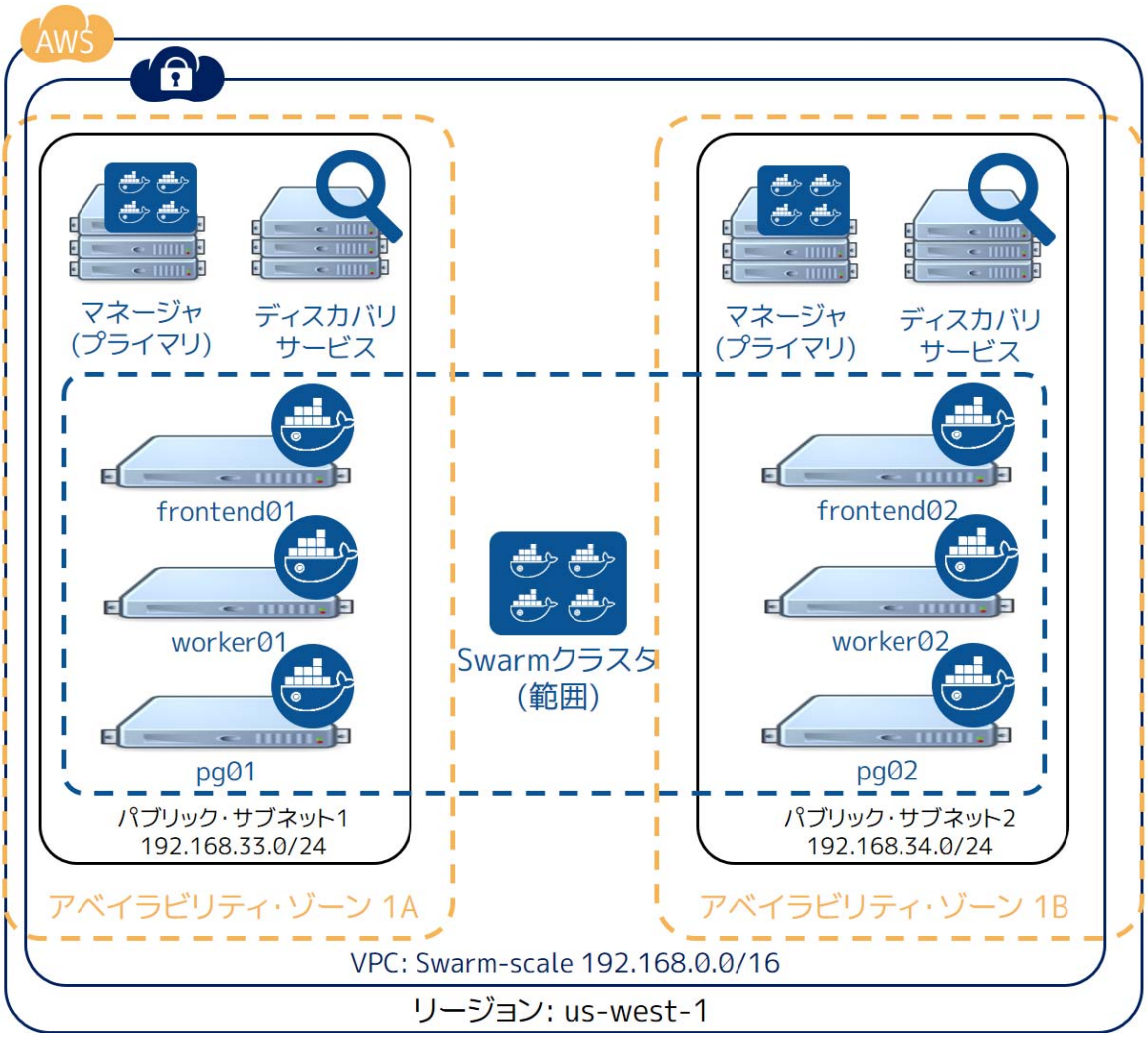
アプリケーションの障害は、その支えとなるインフラによって様々な要因があります。しかしながら、いくつかのベストプラクティスは移行の手助けや障害の緩和に役立つでしょう。

方法の 1 つは、可能な限り多くの障害領域にインフラのコンポーネントを分けてデプロイします。AWS のようなサービスでは、しばしインフラの分散や、複数のリージョン AWS アベイラビリティ・ゾーン（AZ）を横断することです。

Swarm クラスタのアベイラビリティ・ゾーンを増やすには：

- HA 用の Swarm マネージャを、異なった AZ にある H ノードにデプロイ
- HA 用の Consul ディスカバリ・サービスを、異なった AZ にある HA ノードにデプロイ
- 全てのスケーラブルなアプリケーションのコンポーネントを、複数の AZ に横断させる

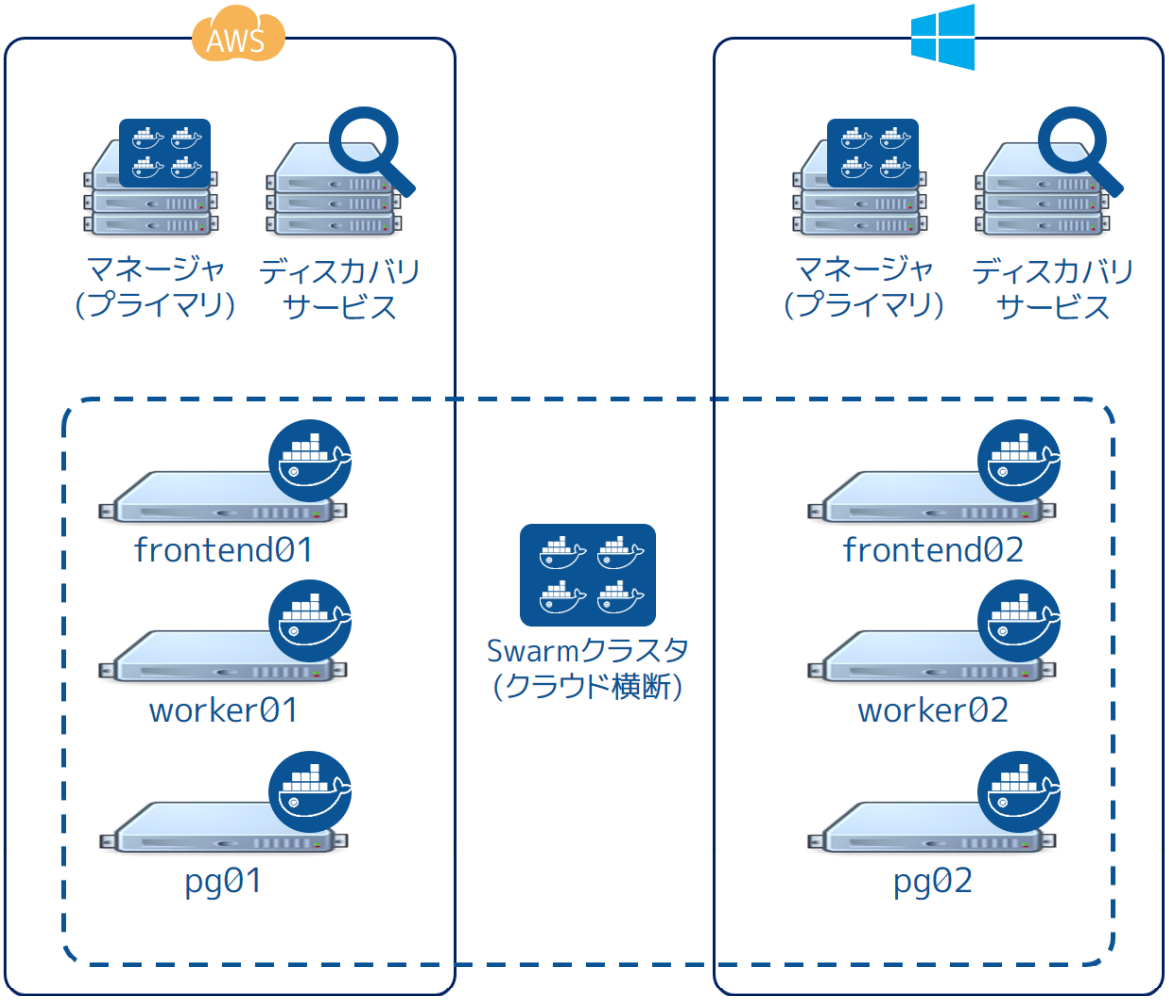
この設定を反映したものが、次の図です。



この手法であれば AZ 全体を喪失しても、クラスタとアプリケーションを処理可能です。

しかし全く止まらない訳ではありません。アプリケーションによっては AWS リージョンを横断して分散されているかもしれません。私たちのサンプルでは、クラスタとアプリケーションを `us-west-1` リージョンにデプロイし、データを `us-east-1` に置いています。この状態から、更にクラウド・プロバイダを横断するデプロイや、あるいはパブリック・クラウド・プロバイダや自分のデータセンタにあるオンプレミスに対して分散することもできるでしょう！

以下の図はアプリケーションとインフラを AWS と Microsoft Azure にデプロイしたものです。ですが、クラウドプロバイダはデータセンタにあるオンプレミスに置き換えても構いません。これらのシナリオでは、ネットワークのレイテンシと信頼性がスムーズに動作させるための鍵となります。


AWS
マネージャ
(プライマリ)
ディスカバリ
サービス
マネージャ
(プライマリ)
ディスカバリ
サービス
frontend01
worker01
pg01
Swarmクラスタ
(クラウド横断)
frontend02
worker02
pg02

4章

管理とネットワーク

4.1 Docker Swarm の高可用性

Docker Swarm の **Swarm マネージャ**は、クラスタ全体に対する責任を持ち、スケール時は複数 Docker ホストのリソースを管理します。もし Swarm マネージャが停止したら、新しいマネージャを作成し、サービス中断に対処しなくてはいけません。

高可用性 (High Availability) 機能により、Docker Swarm は管理インスタンスのフェイルオーバを丁寧に処理します。この機能を使うには、**プライマリ・マネージャ (primary manager)** を作成し、複数の**レプリカ (replica)**インスタンスを作成できます。

プライマリ・マネージャは、Docker Swarm クラスタとの主な接点です。また、バックアップに用いるレプリカ・インスタンスの作成・通信もできます。レプリカにリクエストしたら、プライマリ・マネージャを自動的にプロキシします。プライマリ・マネージャで障害が起これば、レプリカが主導権を取ります。このような方法で、クラスタと通信し続けられます。

4.1.1 プライマリとレプリカのセットアップ

このセクションは、複数のマネージャを使って Docker Swarm をセットアップする方法を説明します。

前提条件

ここでは `Consul`、`etcd`、`Zookeeper` クラスタのいずれかが必要です。今回の手順では、`Consul` サーバが `192.168.42.10:8500` で動作しているものと想定します。サンプルの Swarm は3台のマシンで構成されているものとします。

- `manager-1` は `192.168.42.200` 上で動作
- `manager-2` は `192.168.42.201` 上で動作
- `manager-3` は `192.168.42.202` 上で動作

プライマリ・マネージャの作成

`swarm manager` コマンドで `--replication` と `--advertise` フラグを指定し、プライマリ・マネージャを作成します。

```
user@manager-1 $ swarm manage -H :4000 <tls-config-flags> --replication \
  --advertise 192.168.42.200:4000 consul://192.168.42.10:8500/nodes
INFO[0000] Listening for HTTP addr=:4000 proto=tcp
INFO[0000] Cluster leadership acquired
INFO[0000] New leader elected: 192.168.42.200:4000
[...]
```

--replication フラグは、Swarm に対して複数のマネージャ設定における一部であると伝えます。また、このプライマリ・マネージャは、他のプライマリの役割を持つマネージャ・インスタンスと競合します。プライマリ・マネージャとは、クラスタ管理の権限を持ち、ログを複製し、クラスタ内で発生したイベントを複製します。

--advertise オプションは、プライマリ・マネージャのアドレスを指定します。ノードがプライマリに選ばれた時、Swarm はこのアドレスをクラスタの通知用（advertise）に使います。先ほどのコマンド出力から分かるように、新しく選ばれたプライマリ・マネージャは指定したアドレスを使います。

2つのレプリカ（複製）を作成

プライマリ・マネージャを作ったら、次はレプリカを作成できます。

```
user@manager-2 $ swarm manage -H :4000 <tls-config-flags> --replication \
  --advertise 192.168.42.201:4000 consul://192.168.42.10:8500/nodes
INFO[0000] Listening for HTTP                                 addr=:4000 proto=tcp
INFO[0000] Cluster leadership lost
INFO[0000] New leader elected: 192.168.42.200:4000
[...]
```

このコマンドは 192.168.42.201:4000 上にレプリカ・マネージャを作成します。これは 192.168.42.200:4000 をプライマリ・マネージャとみなしています。

追加で、 3つめのマネージャ・インスタンスを作成します。

```
user@manager-3 $ swarm manage -H :4000 <tls-config-flags> --replication
  --advertise 192.168.42.202:4000 consul://192.168.42.10:8500/nodes
INFO[0000] Listening for HTTP                                 addr=:4000 proto=tcp
INFO[0000] Cluster leadership lost
INFO[0000] New leader elected: 192.168.42.200:4000
[...]
```

プライマリ・マネージャとレプリカを構成した後は、通常通りに **Swarm エージェント**を作成できます。

クラスタ内のマシン一覧

docker info を実行したら、次のような出力が得られます。

```
user@my-machine $ export DOCKER_HOST=192.168.42.200:4000 #  manager-1 を指し示す
user@my-machine $ docker info
Containers: 0
Images: 25
Storage Driver:
Role: Primary  <--------- manager-1 is the Primary manager
Primary: 192.168.42.200
Strategy: spread
Filters: affinity, health, constraint, port, dependency
Nodes: 3
 swarm-agent-0: 192.168.42.100:2375
  └ Containers: 0
```

```
    └ Reserved CPUs: 0 / 1
    └ Reserved Memory: 0 B / 2.053 GiB
    └ Labels: executiondriver=native-0.2, kernelversion=3.13.0-49-generic, operatingsystem=Ubuntu
14.04.2 LTS, storagedriver=aufs
 swarm-agent-1: 192.168.42.101:2375
    └ Containers: 0
    └ Reserved CPUs: 0 / 1
    └ Reserved Memory: 0 B / 2.053 GiB
    └ Labels: executiondriver=native-0.2, kernelversion=3.13.0-49-generic, operatingsystem=Ubuntu
14.04.2 LTS, storagedriver=aufs
 swarm-agent-2: 192.168.42.102:2375
    └ Containers: 0
    └ Reserved CPUs: 0 / 1
    └ Reserved Memory: 0 B / 2.053 GiB
    └ Labels: executiondriver=native-0.2, kernelversion=3.13.0-49-generic, operatingsystem=Ubuntu
14.04.2 LTS, storagedriver=aufs
Execution Driver:
Kernel Version:
Operating System:
CPUs: 3
Total Memory: 6.158 GiB
Name:
ID:
Http Proxy:
Https Proxy:
No Proxy:
```

この情報は manager-1 が現在のプライマリであると示しています。そして、このプライマリへ接続するのに使うアドレスが表示されています。

4.1.2 フェイルオーバ動作のテスト

フェイルオーバ動作をテストするには、特定のプライマリ・マネージャを停止します。Ctrl-C や kill を実行したら、現在のプライマリ・マネージャ（manager-1）は停止します。

自動フェイルオーバを待つ

直後に、他のインスタンスが障害を検出し、レプリカがプライマリ・マネージャの主導権を得ます。

例えば、manager-2 のログを確認します。

```
user@manager-2 $ swarm manage -H :4000 <tls-config-flags> --replication --advertise 192.168.42.201:4000
consul://192.168.42.10:8500/nodes
INFO[0000] Listening for HTTP                            addr=:4000 proto=tcp
INFO[0000] Cluster leadership lost
INFO[0000] New leader elected: 192.168.42.200:4000
INFO[0038] New leader elected: 192.168.42.201:4000
INFO[0038] Cluster leadership acquired                   <--- 新しいプライマリ・マネージャに選出された
[...]
```

これはプライマリ・マネージャ manager-1 で障害が発生しました。その後、192.168.42.201:4000 のアドレスを持つ manager-2 のレプリカが障害を検出したため、主導権を（manager-1 から）取り上げてリーダーに選出されました。理由は manager-2 が十分な速さで、プライマリ・マネージャとして選出手続きを実質的に行ったからです。その結果、manager-2 がクラスタ上のプライマリ・マネージャになりました。

manager-3 を見れば、次のようなログが表示されるでしょう。

```
user@manager-3 $ swarm manage -H :4000 <tls-config-flags> --replication \
  --advertise 192.168.42.202:4000 consul://192.168.42.10:8500/nodes
INFO[0000] Listening for HTTP                            addr=:4000 proto=tcp
INFO[0000] Cluster leadership lost
INFO[0000] New leader elected: 192.168.42.200:4000
INFO[0036] New leader elected: 192.168.42.201:4000   <--- manager-2 が新しいプライマリ・マネージャに
[...]
```

この時点で、新しい DOCKER_HOST の値を指定する必要があります。

プライマリに切り替え

DOCKER_HOST をプライマリとしての manager-2 に切り替えるには、次のようにします。

```
user@my-machine $ export DOCKER_HOST=192.168.42.201:4000 # manager-2 を指定
user@my-machine $ docker info
Containers: 0
Images: 25
Storage Driver:
Role: Replica  <--------- manager-2 はレプリカ
Primary: 192.168.42.200
Strategy: spread
Filters: affinity, health, constraint, port, dependency
Nodes: 3
```

docker コマンドは Docker Swarm プライマリ・マネージャ、あるいは、あらゆるレプリカ上で実行できます。

好みによって、 何らかの仕組みを使うことにより、DOCKER_HOST が現在のプライマリ・マネージャを常に示すようにも可能です。そうしておけば、フェイルオーバ発生の度に、Docker Swarm に対する接続を失うことは無いでしょう。

4.2 Swarm とコンテナのネットワーク

Docker Swarm は Docker のネットワーク機能と完全な互換性があります。互換性の中には複数のホストに対するマルチホスト・ネットワーク機能も含まれます。これは複数の Docker ホストを横断するカスタム・コンテナ・ネットワークを作成する機能です。

Swarm をカスタム・ネットワークで使う前に、Docker コンテナ・ネットワークの概念に関する情報をお読みください。また、「マルチホスト・ネットワーク機能を始める」セクションのサンプルも試すべきでしょう。

4.2.1 Swarm クラスタにカスタム・ネットワークを作成

マルチホスト・ネットワーク機能を使うにはキーバリュー・ストアが必要です。キーバリュー・ストアはネットワークの情報を保持する場所です。ここにはディスカバリ情報、ネットワーク、エンドポイント、IP アドレス等が含まれます。Docker の libkv プロジェクトの成果により、Docker は Consul、Etcd、ZooKeeper の各キーバリュー・ストア・バックエンドをサポートします。詳細は libkv プロジェクト[*1] をご覧ください。

カスタム・ネットワークを作成するには、キーバリュー・ストア・バックエンドを選択し、自分のネットワーク上に実装する必要があります。それから、Docker Engine デーモンの設定を変更し、キーバリュー・ストアにデータを保管できるようにします。キーバリュー・ストア用のサーバを参照するには `--cluster-store` と `--cluster-advertise` という 2 つのパラメータが必要です。

Swarm の各ノード上にあるデーモンの設定変更・再起動を行えば、ネットワーク作成の準備が整います。

4.2.2 ネットワークの一覧

以下は、クラスタ上に 2 のノード `node-0` と `node-1` がある場合の例です。Swarm ノードからネットワーク一覧を表示しています。

```
$ docker network ls
NETWORK ID          NAME                DRIVER
3dd50db9706d        node-0/host         host
09138343e80e        node-0/bridge       bridge
8834dbd552e5        node-0/none         null
45782acfe427        node-1/host         host
8926accb25fd        node-1/bridge       bridge
6382abccd23d        node-1/none         null
```

このように、各ネットワーク名の接頭語がノード名になっていることが分かります。

4.2.3 ネットワークの作成

デフォルトの Swarm クラスタは、ネットワーク全体を範囲とする `overlay` ネットワーク・ドライバを使います。ネットワーク全体を範囲とするドライバを使えば、Swarm クラスタ全体を横断するネットワークを作成できます。Swarm で `overlay` ネットワーク作成時は `-d` オプションを省略できます。

```
$ docker network create swarm_network
42131321acab3233ba342443Ba4312
$ docker network ls
NETWORK ID          NAME                DRIVER
3dd50db9706d        node-0/host         host
09138343e80e        node-0/bridge       bridge
```

*1 https://github.com/docker/libkv

```
8834dbd552e5        node-0/none             null
42131321acab        node-0/swarm_network    overlay
45782acfe427        node-1/host             host
8926accb25fd        node-1/bridge           bridge
6382abccd23d        node-1/none             null
42131321acab        node-1/swarm_network    overlay
```

ここで表示されているように、2つのノード上に node-0/swarm_network と node-1/swarm_network という同じ ID を持つネットワークがあります。これはクラスタに作成したネットワークであり、全てのノード上でアクセス可能なものです。

ローカルな範囲でネットワークを作成したい場合は (例えば、ブリッジ・ドライバを使いたい時)、 <ノード名>/<名前> の形式でなければ、ランダムに選んだノード上でネットワークを作成します。

```
$ docker network create node-0/bridge2 -b bridge
921817fefea521673217123abab223
$ docker network create node-1/bridge2 -b bridge
5262bbfe5616fef6627771289aacc2
$ docker network ls
NETWORK ID          NAME                    DRIVER
3dd50db9706d        node-0/host             host
09138343e80e        node-0/bridge           bridge
8834dbd552e5        node-0/none             null
42131321acab        node-0/swarm_network    overlay
921817fefea5        node-0/bridge2          bridge
45782acfe427        node-1/host             host
8926accb25fd        node-1/bridge           bridge
6382abccd23d        node-1/none             null
42131321acab        node-1/swarm_network    overlay
5262bbfe5616        node-1/bridge2          bridge
```

4.2.4 ネットワークの削除

ネットワークの削除は、ネットワーク ID かネットワーク名を使えます。異なる2つのネットワークが同じ名前の場合は、 <ノード名>/<名前> を使えます。

```
$ docker network rm swarm_network
42131321acab3233ba342443Ba4312
$ docker network rm node-0/bridge2
921817fefea521673217123abab223
$ docker network ls
NETWORK ID          NAME                    DRIVER
3dd50db9706d        node-0/host             host
09138343e80e        node-0/bridge           bridge
8834dbd552e5        node-0/none             null
45782acfe427        node-1/host             host
8926accb25fd        node-1/bridge           bridge
6382abccd23d        node-1/none             null
5262bbfe5616        node-1/bridge2          bridge
```

swarm_network は各ノードから削除されましたが、bridge2 は node-0 からのみ削除されました。

4.3 Docker Swarm ディスカバリ

Docker Swarm は複数のディスカバリ・バックエンドに対応しています。Docker Swarm は**ホステット・ディスカバリ・サービス（hosted discovery service）**が利用可能です。このサービスはクラスタ上の IP アドレスの一覧を保持します。このセクションでは利用可能な様々なホステット・ディスカバリを紹介します。

4.3.1 分散キーバリュー・ストアの利用

Swarm でノードをディスカバリ（発見）するのに推奨される方法は、Docker による libkv プロジェクトの利用です。libkv プロジェクトとは既存の分散キーバリュー・ストア上の抽象化レイヤです。この原稿を書いている時点で、プロジェクトがサポートしているのは次の通りです。

- Consul 0.5.1 以上
- Etcd 2.0 以上
- ZooKeeper 3.4.5 以上

libkv についてやサポートしているバックエンドに対する技術的な詳細は、libkv プロジェクト[*1] をご覧ください。

ホステット・ディスカバリ・キーストアを使用

1. 各ノードで Swarm エージェントを起動します。

ノードの IP アドレスは Swarm マネージャがアクセス可能であれば十分であり、パブリックな IP アドレスを持つ必要はありません。大きなクラスタになれば、Swarm に対するノードの参加が、ディスカバリ時に過負荷となる可能性があります。例えば、沢山のノードをスクリプトで登録する場合や、ネットワーク障害から復旧する時です。この影響によりディスカバリが失敗するかもしれません。そのような場合は、 `--delay` オプションで遅延上限を指定できます。そうすると、Swarm への登録がランダムに遅延して行われますが、指定した時間を上回ることはありません。

Etcd:

```
swarm join --advertise=<node_ip:2375> etcd://<etcd_addr1>,<etcd_addr2>/<optional path prefix>
```

Consul:

```
swarm join --advertise=<node_ip:2375> consul://<consul_addr>/<optional path prefix>
```

ZooKeeper:

```
swarm join --advertise=<node_ip:2375> zk://<zookeeper_addr1>,<zookeeper_addr2>/<optional path prefix>
```

2. Swarm マネージャをサーバもしくはノート PC 上で起動します。

Etcd:

*1 https://github.com/docker/libkv

```
swarm manage -H tcp://<swarm_ip:swarm_port> etcd://<etcd_addr1>,<etcd_addr2>/<optional path prefix>
```

Consul:

```
swarm manage -H tcp://<swarm_ip:swarm_port> consul://<consul_addr>/<optional path prefix>
```

ZooKeeper:

```
swarm manage -H tcp://<swarm_ip:swarm_port> zk://<zookeeper_addr1>,<zookeeper_addr2>/<optional path
prefix>
```

3. 通常の Docker コマンドを実行します。

```
docker -H tcp://<swarm_ip:swarm_port> info
docker -H tcp://<swarm_ip:swarm_port> run ...
docker -H tcp://<swarm_ip:swarm_port> ps
docker -H tcp://<swarm_ip:swarm_port> logs ...
...
```

4. クラスタ上のノード一覧を表示します。

Etcd:

```
swarm list etcd://<etcd_addr1>,<etcd_addr2>/<optional path prefix>
<node_ip:2375>
```

Consul:

```
swarm list consul://<consul_addr>/<optional path prefix>
<node_ip:2375>
```

ZooKeeper:

```
swarm list zk://<zookeeper_addr1>,<zookeeper_addr2>/<optional path prefix>
<node_ip:2375>
```

分散キーバリュー・ディスカバリに TLS を使う

分散キーバリュー・ストアと安全に通信できるようにするため、TLS を利用できます。ストアへ安全に接続するには、Swarm クラスタにノードが join（参加）する時に使う証明書を生成しなくてはいけません。証明書に対応しているのは Consul と Etcd のみです。以下は Consul を使う例です。

```
swarm join \
    --advertise=<node_ip:2375> \
    --discovery-opt kv.cacertfile=/path/to/mycacert.pem \
    --discovery-opt kv.certfile=/path/to/mycert.pem \
    --discovery-opt kv.keyfile=/path/to/mykey.pem \
    consul://<consul_addr>/<optional path prefix>
```

これは Swarm の manage と list コマンドを使う場合も同様です。

4.3.2 静的なファイルまたはノード・リスト

ディスカバリ・バックエンドとして静的なファイルもしくはノードのリストを使えます。このファイルは Swarm マネージャがアクセス可能なホスト上に置く必要があります。あるいは、Swarm 起動時にオプションでノードのリストを指定することもできます。

静的なファイルあるいは nodes オプションは IP アドレスの範囲指定をサポートしています。特定のパターンで範囲を指定するには、例えば 10.0.0.[10:200] を指定したら、10.0.0.10 から 10.0.0.200 までのノードを探そうとします。以下は file（ファイル）ディスカバリ手法を使う例です。

```
$ echo "10.0.0.[11:100]:2375"   >> /tmp/my_cluster
$ echo "10.0.1.[15:20]:2375"    >> /tmp/my_cluster
$ echo "192.168.1.2:[2:20]375"  >> /tmp/my_cluster
```

あるいはノードの直接指定でディスカバリするには、次のように実行します。

```
swarm manage -H <swarm_ip:swarm_port> "nodes://10.0.0.[10:200]:2375,10.0.1.[2:250]:2375"
```

ファイルを作成する場合

1. ファイルを編集し、各行にノードの情報を追加します。

```
echo <node_ip1:2375> >> /opt/my_cluster
echo <node_ip2:2375> >> /opt/my_cluster
echo <node_ip3:2375> >> /opt/my_cluster
```

この例では /opt/my_cluster というファイルを作成しています。任意のファイル名を指定できます。

2. Swarm マネージャを何らかのマシン上で実行します。

```
swarm manage -H tcp://<swarm_ip:swarm_port> file:///tmp/my_cluster
```

3. 通常の Docker コマンドを使います。

```
docker -H tcp://<swarm_ip:swarm_port> info
docker -H tcp://<swarm_ip:swarm_port> run ...
docker -H tcp://<swarm_ip:swarm_port> ps
docker -H tcp://<swarm_ip:swarm_port> logs ...
...
```

4. クラスタ上のノード一覧を表示します。

```
$ swarm list file:///tmp/my_cluster <node_ip1:2375> <node_ip2:2375> <node_ip3:2375>
```

ノード・リストを指定する場合

1. マシンもしくはノート PC 上でマネージャを起動します。

```
swarm manage -H <swarm_ip:swarm_port> nodes://<node_ip1:2375>,<node_ip2:2375>
```

あるいは、次のように実行します。

```
swarm manage -H <swarm_ip:swarm_port> <node_ip1:2375>,<node_ip2:2375>
```

2. 通常の Docker コマンドを実行します。

```
docker -H <swarm_ip:swarm_port> info
docker -H <swarm_ip:swarm_port> run ...
docker -H <swarm_ip:swarm_port> ps
docker -H <swarm_ip:swarm_port> logs ...
```

3. クラスタ上のノード一覧を表示します。

```
$ swarm list file:///tmp/my_cluster
<node_ip1:2375>
<node_ip2:2375>
<node_ip3:2375>
```

Docker Hub のホステッド・ディスカバリ

【警告】Docker Hub ホステット・ディスカバリ・サービスはプロダクションでの利用が**推奨されていません**。これはテストや開発環境での利用を想定しています。プロダクション環境においては、ディスカバリ・バックエンドの項目をご覧ください。

この例は Docker Hub のホステッド・ディスカバリ・サービスを使います。Docker Hub のホステッド・ディスカバリ・サービスを使うには、インターネットに接続している必要があります。次のようにして Swarm クラスタを作成します。

1. まずクラスタを作成します。

```
# クラスタを作成
$ swarm create
6856663cdefdec325839a4b7e1de38e8 # <- これが各自の <クラスタ ID> です
```

2. 各ノードを作成し、クラスタに追加します。

各ノードで Swarm エージェントを起動します。Swarm Manager がアクセス可能であれば、<node_ip> はパブリックである必要はありません（例：192.168.0.x）。

```
$ swarm join --advertise=<node_ip:2375> token://<cluster_id>
```

3. Swarm マネージャを起動します。

これはあらゆるマシン上だけでなく、自分のノート PC 上でも実行できます。

```
$ swarm manage -H tcp://<swarm_ip:swarm_port> token://<cluster_id>
```

4. 通常の Docker コマンドでクラスタと通信します。

```
docker -H tcp://<swarm_ip:swarm_port> info
docker -H tcp://<swarm_ip:swarm_port> run ...
docker -H tcp://<swarm_ip:swarm_port> ps
docker -H tcp://<swarm_ip:swarm_port> logs ...
...
```

5. クラスタのノード情報一覧を表示します。

```
swarm list token://<cluster_id>
<node_ip:2375>
```

4.3.3 新しいディスカバリ・バックエンドに貢献

あなたも Swarm 向けの新しいディスカバリ・バックエンドに貢献できます。どのようにするかは、Docker Swarm リポジトリにある discovery README[*1] をお読みください。

*1 https://github.com/docker/swarm/blob/master/discovery/README.md

5章

セキュリティ

5.1 Swarm と TLS の概要

Swam クラスタの全てのノードでは、それぞれの Docker デーモンが通信用のポートを公開（バインド）する必要があります。そのため、これがセキュリティ上の懸念となるのは明らかです。インターネットのような信頼を疑うべきネットワークにおいて、懸念は倍増します。これらのリスクを減らすため、Docker Swarm と Docker Engine のデーモンは **TLS（Transport Layer Security；トランスポート・レイヤ・セキュリティ）**をサポートしています。

TLS は SSL（Secure Sockets Layer）の後継であり、この 2 つは相互に互換性があります。この記事の中では、Docker は TLS を使います。

5.1.1 TLS の概念を学ぶ

先へ進む前に、TLS と **PKI (Public Key Infrastructure；公開鍵基盤)** の基本概念を理解しておくことが重要です。

公開鍵基盤はセキュリティに関連する技術・ポリシー・手続きを組み合わせたものです。これらが電子証明書の作成や管理に使われます。認証や暗号化のような仕組みにおいて、これらの証明書とインフラの安全なデジタル通信が使われます。

もしかすると、次の例えが分かりやすいかもしれません。個人を認識する手段としてパスポートを使うことは一般的です。通常のパスポートには個人を特定するための写真や生体情報が記録されています。また、パスポートには有効な国名だけでなく、有効・無効に関する日付も記録されています。この仕組みと電子証明は非常に似ています。ある電子証明書を展開したのが、次のテキストです。

```
Certificate:
Data:
    Version: 3 (0x2)
    Serial Number: 9590646456311914051 (0x8518d2237ad49e43)
Signature Algorithm: sha256WithRSAEncryption
    Issuer: C=US, ST=CA, L=Sanfrancisco, O=Docker Inc
    Validity
        Not Before: Jan 18 09:42:16 2016 GMT
        Not After : Jan 15 09:42:16 2026 GMT
    Subject: CN=swarm
```

この証明書で swarm という名称のコンピュータを識別します。証明書の有効期間は 2016 年 1 月から 2026 年 1 月までです。そしてこれを発行したのは米国カリフォルニア州を拠点としている Docker, Inc.です。

パスポートを使った個人認証が使われるのは、飛行機の搭乗や、板を囲まれた税関においてです。電子証明書はネットワーク上のコンピュータを認証するために使います。

公開鍵基盤（PKI）とは、電子証明書を有効に機能させるため、その背後で使われる技術・ポリシー・手順の組み合わせなのです。PKI によって提供される技術・ポリシー・手続きには、以下の項目が含まれます。

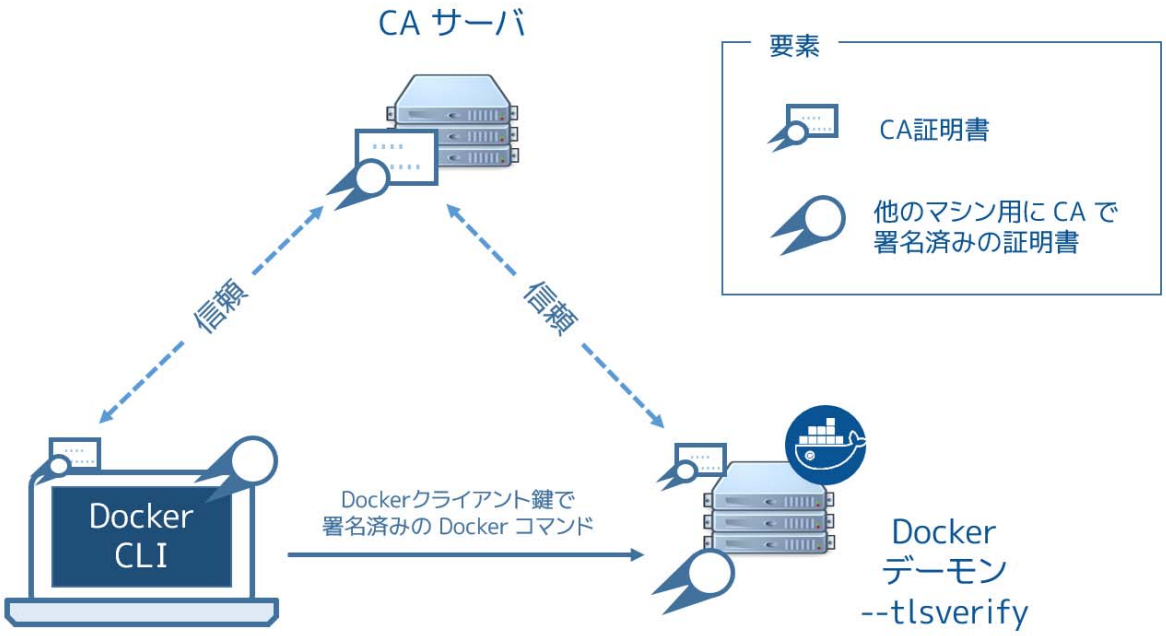
- 証明書を安全に要求するためのサービス
- 要求された証明書が、実在しているかどうか確認するための手順
- 証明書の実体が、適格かどうか決めるための手順
- 証明書を発行する技術と手順
- 証明書を破棄する技術と手順

5.1.2 Docker Engine で TLS 認証を使うには

このセクションは Docker Engine と Swarm のセキュリティを高めるために、 PKI と証明書を使う方法を学びます。

TLS 認証を使うためには、Docker Engine CLI と Docker Engine デーモンの両方で設定が必要です。TLS の設定を行うというのは、Docker Engine CLI と Docker Engine デーモン間の全ての通信を、信頼のある電子証明書で署名された状態で行うことを意味します。Docker Engine CLI は Docker Engine デーモンと通信する前に、電子証明書の提出が必要です。

また、Docker Engine デーモンも Docker Engine CLI が使う証明証を信頼する必要があります。信頼とは、通常は第三者の信頼機関によって担保されます。下図は Docker Engine CLI とデーモンが TLS 通信に必要となる設定です。



図中における信頼できる第三者とは**認証局（CA; Certificate Authority）**サーバです。認証局（CA）とは、パスポートを例にすると国に相当します。認証局は証明証を作成・署名・発行・無効化します。Docker Engine デーモンを実行するホスト上では、信頼を確立するために、認証局のルート証明書をインストールします。Docker Engine

CLI は認証局のサーバに対して証明書を要求します。認証局サーバはクライアントに対して証明書の署名・発行を行います。

Docker Engine CLI はコマンドを実行する前に、この（認証局で署名された）証明書を Docker Engine デーモンに送ります。デーモンは証明書を調査します。その証明書がデーモンの信頼する認証局が署名したものであれば、デーモンは自動的に信頼します。証明書が適切であるとみなすと（証明書が有効期間内であり、破棄されたものでないと分かれば)、Docker Engine デーモンは信頼できる Engine CLI からの要求とみなしコマンドを受け付けます。

Docker Engine CLI はシンプルなクライアントです。Docker Engine デーモンと通信するために Docker リモート API を使います。Docker リモート API を利用可能なクライアントであれば、どれも TLS が使えます。例えば、TLS をサポートしている Docker ユニバーサル・コントロール・プレーン（UCP） に他のクライアントからもアクセス可能です。他のクライアントとは、Docker リモート API を使うサードパーティー製のプロダクトでも、同様に設定ができます。

5.1.3 Docker と Swarm の TLS モード

Docker Engine デーモンが認証に使う証明書について学んできました。重要なのは、Docker Engine デーモンとクライアントで利用できる TLS 設定には 3 種類あることに注意すべきです。

- 外部のサードパーティー製の証明局(CA)
- 社内にある証明局(CA)
- 証明書に対する自己署名

どの証明局(CA) を使うかにより、実際の設定内容が異なります。

外部のサードパーティー製の証明局

外部の証明局とは、信頼できる第三者による会社を指します。そこが証明書の作成、発行、無効化、その他の管理を行います。信頼されているという言葉が意味するのは、高いレベルのセキュリティを実現・維持し、ビジネスに対する成功をもたらすものです。また、外部の認証局のルート証明書をインストールすることにより、皆さんのコンピュータやサーバも信頼できうるものとします。

外部のサードパーティー認証局を使うえば、その認証局によって、皆さんの証明書が作成・署名・発行・無効化など管理が行われます。通常はサービスの利用に料金が発生します。しかし、エンタープライズ・クラスの安定したソリューションを考慮した、高度な信頼をもたらすでしょう。

社内にある証明局

多くの組織で、その組織内で認証局や PKI を運用することが選ばれています。そのために OpenSSL もしくは Microsoft Active Directory を使うのが一般的な例です。このような場合、皆さんの会社自身が自信で証明機関を運用しています。この利点は、自分自身が証明局ですので、更なる PKI を管理できる点です。

外部のサードパーティー認証局が提供するサービスを使い、自身の認証局や PKI を必要に応じて運用できます。これには証明書の作成・発行・破棄などの管理が含まれています。全てを自分たちで運用するとコストやオーバヘッドが必要となるでしょう。しかし、大規模な企業であれば、全てサードパーティーによるサービスを使うよりはコストを削減できるかもしれません。

自分たち自身で認証局や PKI の内部運用・管理を考えているのであれば、企業における認証局を実現するため、高い可用性や高いセキュリティについて考慮が必要になるでしょう。

自己署名した証明書

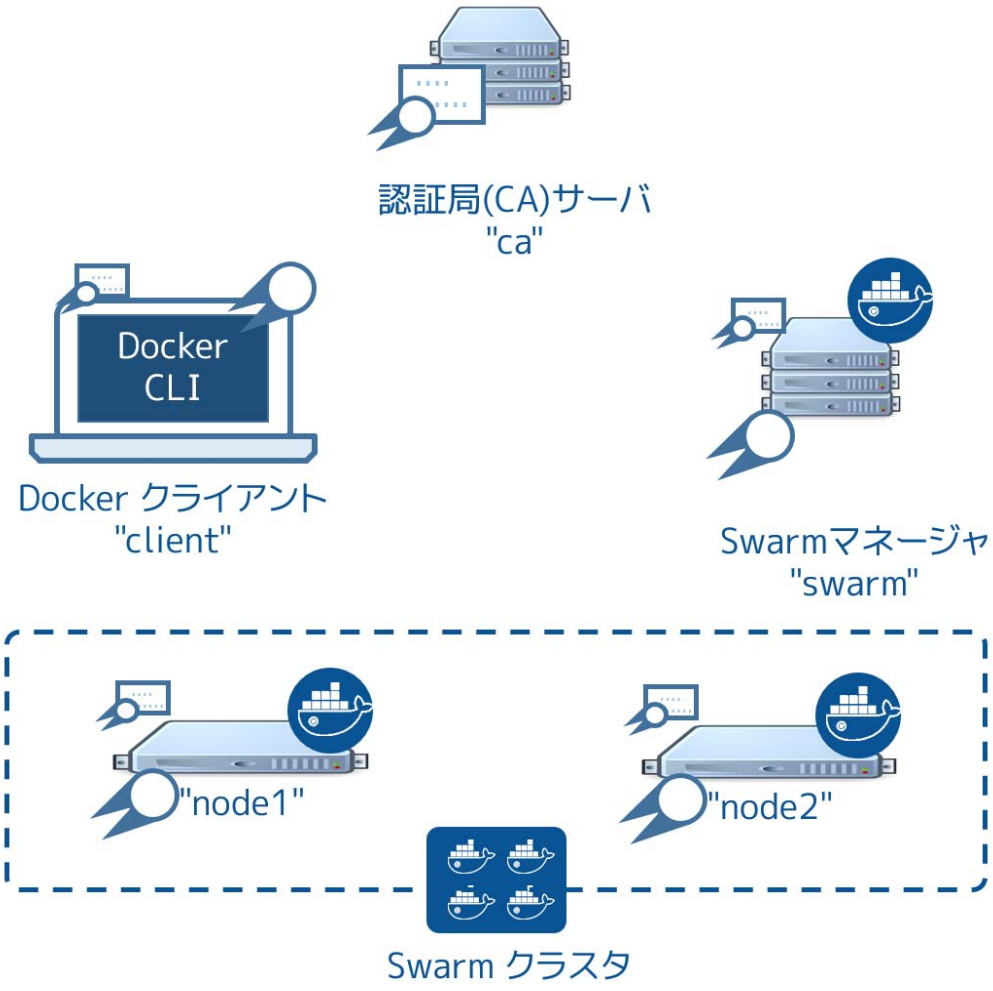
その名前の通り、自己署名した証明書とは、信頼できる認証局の代わりに、自分自身の秘密鍵で署名するものです。これは低いコストかつ簡単に使えるものです。もし自分自身で署名した証明書を適切に運用したいのであれば、

証明書を使わないのも良い方法かもしれません。

なぜならば、自己署名した証明書が本来の PKI を損ねる可能性があるためです。この手法はスケールしませんし、他の選択肢に比べますと、多くの点で不利です。不利な点の 1 つに、自分自身で自己署名した証明書を無効化できません。これだけでなく、他にも制限があるため、自己署名の証明書は、この 3 つの選択肢の中で最低のセキュリティと考えられます。信頼できないネットワーク上でプロダクション用のワークロードを公開する必要があれば、自己署名の証明書の利用は推奨されません。

5.2 Docker Swarm の TLS 設定

この手順では下図のように、Swarm クラスタに Swarm マネージャと認証局 (CA) の２つのノードを作成します。全ての Docker Engine ホスト（`client`、`swarm`、`node1`、`node2`）は、認証局の証明書のコピーと、自分自身で認証局の署名をしたキーペアのコピーも持ちます。



以下の手順で作業を進めていきます。

- ステップ１：動作環境のセットアップ
- ステップ２：認証局(CA)サーバの作成
- ステップ３：鍵の作成と署名
- ステップ４：鍵のインストール
- ステップ５：Engine デーモンに TLS 設定
- ステップ６：Swarm クラスタの作成
- ステップ７：TLS を使う Swarm マネージャの作成
- ステップ８：Swarm マネージャの設定を確認
- ステップ９：TLS を使う Engilne CLI の設定

5.2.1 始める前に

このセクションは OpenSSL で自分自身で認証局(CA)を作成する手順を含みます。これは簡単な社内の認証局や PKI と似ています。しかしながら、プロダクション級の内部の認証局・PKI としては**使うべきではありません**。以降の手順は検証用（デモンストレーション）目的のみです。つまり、皆さんが既に適切な証明局や証明書をお持ちであれば、Docker Swarm で TLS を利用する際には置き換えてお読みください。

5.2.2 手順

ステップ１：動作環境のセットアップ

この手順を進めるには、５つの Linux サーバの起動が必要です。これらのサーバは物理と仮想を組み合わせても構いません。以下の表はサーバ名と役割の一覧です。

サーバ名	説明
ca	認証局(CA)サーバとして動作
swarm	Swarm マネージャとして動作
node1	Swarm ノードとして動作
node2	Swarm ノードとして動作
client	リモートの Docker Engine クライアントとして動作

５台全てのサーバに SSH 接続が可能なのを確認し、DNS の名前解決でお互いに通信できるようにします。特に、次の２点に気を付けます。

- Swarm マネージャと Swarm ノード間は TCP ポート 2376 を開く
- Docker Engine クライアントと Swarm マネージャ間は TCP ポート 3376 を開く

既に使用中であれば、他のポートも選べます。この例ではこれらのポートを使う想定です。

各サーバは Docker Engine と互換性のあるオペレーティング・システムを実行します。簡単にするため、以降のステップでは全てのサーバを Ubuntu 14.04 LTS で動かすと想定します。

ステップ２：認証局(CA)サーバの作成

既に認証局にアクセス可能で証明書があるならば、それらを使ったほうが便利です。その場合、次のステップにスキップしてください。

このステップでは Linux サーバを認証局として設定します。認証局は鍵の作成と署名に使います。読者が既存の（外部または企業の）認証局へのアクセスや証明書が無くても、このステップでは必要な環境のインストールと証明書を使えるようにします。しかし、プロダクションへのデプロイには適切では「ない」モデルです。

1. 認証局サーバのターミナルに入り、root に昇格します。

```
$ sudo su
```

2. 認証局用の秘密鍵 ca-priv-key.pem を作成します。

```
# openssl genrsa -out ca-priv-key.pem 2048
Generating RSA private key, 2048 bit long modulus
.....................................................+++
```

```
.....+++
e is 65537 (0x10001)
```

3. 認証局用の公開鍵 ca.pem を作成します。

公開鍵の作成は、直前の手順で作成した秘密鍵を元にします。

```
# openssl req -config /usr/lib/ssl/openssl.cnf -new -key ca-priv-key.pem -x509 -days 1825 -out ca.pem
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
<output truncated>
```

公開鍵・秘密鍵のペアを持つ認証局のサーバを設定しました。

```
# openssl rsa -in ca-priv-key.pem -noout -text
```

公開鍵（認証済み）を調べるには、次のようにします。

```
# openssl x509 -in ca.pem -noout -text`
```

次のコマンドは、認証局の公開鍵情報を一部表示します。

```
# openssl x509 -in ca.pem -noout -text
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 17432010264024107661 (0xf1eaf0f9f41eca8d)
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: C=US, ST=CA, L=Sanfrancisco, O=Docker Inc
        Validity
            Not Before: Jan 16 18:28:12 2016 GMT
            Not After : Jan 13 18:28:12 2026 GMT
        Subject: C=US, ST=CA, L=San Francisco, O=Docker Inc
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
                Modulus:
                    00:d1:fe:6e:55:d4:93:fc:c9:8a:04:07:2d:ba:f0:
                    55:97:c5:2c:f5:d7:1d:6a:9b:f0:f0:55:6c:5d:90:
<output truncated>
```

後ほど、他のインフラ上にあるサーバの鍵に対する署名で使います。

ステップ３：鍵の作成と署名

これで認証局が動きました。次は Swarm マネージャ、Swarm ノード、リモートの Docker Engine クライアント用の鍵ペアを作成する必要があります。鍵ペア作成の命令と手順は、全てのサーバで同一です。次の鍵を作成します。

ca-priv-key.pem	認証局の秘密鍵であり、安全に保つ必要があります。後ほど環境上にある他ノード用の新しい鍵の署名で使います。ca.pem ファイルと認証局の鍵ペアを構成します。
ca.pem	認証局の公開鍵であり、証明書（certificate）とも呼ばれます。このファイルは環境上全てのノード上にインストールします。つまり、全てのノードは認証局が署名した信頼できる鍵を持っています。ca-priv-key.pem ファイルと認証局の鍵ペアを構成します。
node.csr	証明書署名要求（certificate signing request；CSR）です。認証局に対して個々のノードごとに新しい鍵ペアを作成時 CSR を効率的に使います。認証局は指定した CSR から情報を取得し、ノード用の公開鍵と秘密鍵の鍵ペアを作成します。
node-priv.key	認証局で署名した秘密鍵。ノードはリモートの Docker Engine との認証に使います。node-cert.pem ファイルとノードの鍵ペアを構成します。
node-cert.pem	認証局で署名した証明書。今回のサンプルでは使いません。node-priv.key ファイルとノードの鍵ペアを構成します。

以下で紹介するのは、ノード全てに対する鍵を作成するコマンドの使い方です。認証局サーバ上のディレクトリで、この手順を進めます。

1. 認証局サーバのターミナルにログインし、root に昇格します。

```
$ sudo su
```

2. Swarm マネージャ用の秘密鍵 swarm-priv-key.pem を作成します。

```
# openssl genrsa -out swarm-priv-key.pem 2048
Generating RSA private key, 2048 bit long modulus
.....................................................+++
........+++
e is 65537 (0x10001)
```

3. 証明書署名要求(CSR) swarm.csr を作成します。

```
# openssl req -subj "/CN=swarm" -new -key swarm-priv-key.pem -out swarm.csr
```

この手順はデモンストレーション目的専用です。ご注意ください。実際のプロダクション環境における CSR 作成手順とは若干異なります。

4. 前のステップで作成した CSR を元に、証明書 swarm-cert.pem を作成します。

```
# openssl x509 -req -days 1825 -in swarm.csr -CA ca.pem -CAkey ca-priv-key.pem -CAcreateserial \
  -out swarm-cert.pem -extensions v3_req -extfile /usr/lib/ssl/openssl.cnf
<省略>
# openssl rsa -in swarm-priv-key.pem -out swarm-priv-key.pem
```

これで Swarm マネージャの鍵ペアを作成しました。

5. これまでのステップを各インフラ上（node1、node2、client）で繰り返します。

各ノードで鍵ペアの作成時は、 swarm の値を各ノードのものへ置き換えてください。

サーバ名	秘密鍵	CSR	証明書
swarm	node1-priv-key.pem	node1.csr	node1-cert.pem
node1	node2-priv-key.pem	node2.csr	node2-cert.pem
node2	client-priv-key.pem	client.csr	client-cert.pem

6. 自分の作業用ディレクトリ上に、以下のファイルがあるのを確認します。

```
# ls -l
total 64
-rw-r--r-- 1 root   root   1679 Jan 16 18:27 ca-priv-key.pem
-rw-r--r-- 1 root   root   1229 Jan 16 18:28 ca.pem
-rw-r--r-- 1 root   root     17 Jan 18 09:56 ca.srl
-rw-r--r-- 1 root   root   1086 Jan 18 09:56 client-cert.pem
-rw-r--r-- 1 root   root    887 Jan 18 09:55 client.csr
-rw-r--r-- 1 root   root   1679 Jan 18 09:56 client-priv-key.pem
-rw-r--r-- 1 root   root   1082 Jan 18 09:44 node1-cert.pem
-rw-r--r-- 1 root   root    887 Jan 18 09:43 node1.csr
-rw-r--r-- 1 root   root   1675 Jan 18 09:44 node1-priv-key.pem
-rw-r--r-- 1 root   root   1082 Jan 18 09:49 node2-cert.pem
-rw-r--r-- 1 root   root    887 Jan 18 09:49 node2.csr
-rw-r--r-- 1 root   root   1675 Jan 18 09:49 node2-priv-key.pem
-rw-r--r-- 1 root   root   1082 Jan 18 09:42 swarm-cert.pem
-rw-r--r-- 1 root   root    887 Jan 18 09:41 swarm.csr
-rw-r--r-- 1 root   root   1679 Jan 18 09:42 swarm-priv-key.pem
```

それぞれの鍵の内容を自分で確認できます。秘密鍵を調べるには、次のようにします。

```
openssl rsa -in <key-name> -noout -text
```

公開鍵の確認は、次のようにします。

```
openssl x509 -in <key-name> -noout -text
```

次のコマンドは、Swarm マネージャ公開鍵 swarm-cert.pem の内容を表示する一部です。

```
# openssl x509 -in ca.pem -noout -text
Certificate:
Data:
    Version: 3 (0x2)
    Serial Number: 9590646456311914051 (0x8518d2237ad49e43)
Signature Algorithm: sha256WithRSAEncryption
    Issuer: C=US, ST=CA, L=Sanfrancisco, O=Docker Inc
    Validity
        Not Before: Jan 18 09:42:16 2016 GMT
        Not After : Jan 15 09:42:16 2026 GMT
    Subject: CN=swarm

<出力を省略>
```

ステップ４：鍵のインストール

このステップは、インフラ上の各サーバに鍵をインストールします。各サーバは３つのファイルが必要です。

- 認証局公開鍵（ca.pem）のコピー
- 自分の秘密鍵
- 自分の公開鍵（証明書）

以下の手順では、認証局サーバから各サーバに scp を使い、３つのファイルをコピーします。コピーの段階で、各ノードごとにファイル名を変更します。

オリジナル名	コピー名
ca.pem	ca.pem
<サーバ名>-cert.pem	cert.pem
<サーバ名>-priv-key.pem	key.pem

1. 認証局サーバのターミナルにログインし、root に昇格します。

```
$ sudo su
```

2. Swarm マネージャ上で ~/.certs ディレクトリを作成します。

```
$ ssh ubuntu@swarm 'mkdir -p /home/ubuntu/.certs'
```

3. 認証局から Swarm マネージャ・サーバに鍵をコピーします。

```
$ scp ./ca.pem ubuntu@swarm:/home/ubuntu/.certs/ca.pem
$ scp ./swarm-cert.pem ubuntu@swarm:/home/ubuntu/.certs/cert.pem
$ scp ./swarm-priv-key.pem ubuntu@swarm:/home/ubuntu/.certs/key.pem
```

scp コマンドの動作には認証情報の指定が必要になるかもしれません。例えば、AWS EC2 インスタンスは証明書ベースでの認証を使います。公開鍵 nigel.pem を関連付けている EC2 インスタンスにファイルをコピーするには、scp コマンドを次のように変更します。

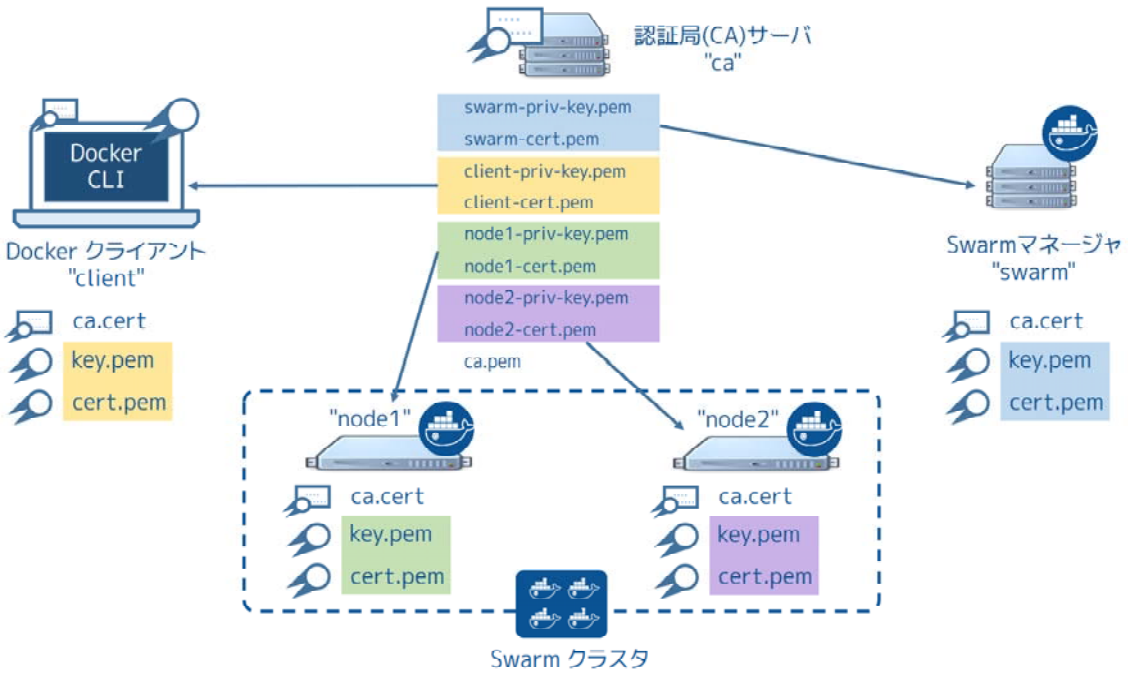
```
scp -i /path/to/nigel.pem ./ca.pem ubuntu@swarm:/home/ubuntu/.certs/ca.pem
```

4. インフラ上の各サーバに対して２つの手順を繰り返します。

- node1
- node2
- client

5. 動作確認をします。

コピーが完了したら、各マシンは以下の鍵を持ちます。



インフラ上の各ノードでは、 /home/ubuntu/.certs/ ディレクトリに次のファイルがあるでしょう。

```
# ls -l /home/ubuntu/.certs/
total 16
-rw-r--r-- 1 ubuntu ubuntu 1229 Jan 18 10:03 ca.pem
-rw-r--r-- 1 ubuntu ubuntu 1082 Jan 18 10:06 cert.pem
-rw-r--r-- 1 ubuntu ubuntu 1679 Jan 18 10:06 key.pem
```

ステップ５：Engine デーモンに TLS 設定

先ほどのステップでは、各 Swarm ノードで必要な鍵をインストールしました。このステップでは、ネットワーク上で通信可能に調整し、TLS を使う通信のみ受け付けるようにします。このステップが終われば、Swarm ノードは TCP ポート 2376 をリッスンし、TLS を使う接続のみ受け付けます。

node1 と node2（Swarm ノード）上で以下の作業を行います。

1. node1 のターミナルを開き、root に昇格します。

```
$ sudo su
```

2. Docker Engine 設定ファイルを編集します。

以降の手順を Ubuntu 14.04 LTS で進めるのであれば、設定ファイルは /etc/default/docker です。Docker Engine の設定ファイルは、お使いの Linux ディストリビューションに依存します。

3. DOCKER_OPTS 行に以下のオプションを追加します。

```
-H tcp://0.0.0.0:2376 --tlsverify --tlscacert=/home/ubuntu/.certs/ca.pem
--tlscert=/home/ubuntu/.certs/cert.pem --tlskey=/home/ubuntu/.certs/key.pem ←１行で入力
```

4. Docker Engine デーモンを再起動します。

```
$ service docker restart
```

5. node2 でも同様の設定を繰り返します。

ステップ６：Swarm クラスタの作成

次は Swarm クラスタを作成します。以降の手順では、２つのノードを持つ Swarm クラスタを、デフォルトのホステッド・ディスカバリ・バックエンドで作成します。デフォルトのホステッド・ディスカバリは Docker Hub を使います。また、プロダクション環境での利用は非推奨です。

1. Swarm マネージャ用ノードのターミナルにログインします。

2. TOKEN 環境変数にユニークな ID を取り込み、クラスタを作成します。

```
$ sudo export TOKEN=$(docker run --rm swarm create)
Unable to find image 'swarm:latest' locally
latest: Pulling from library/swarm
d681c900c6e3: Pulling fs layer
<省略>
986340ab62f0: Pull complete
a9975e2cc0a3: Pull complete
Digest: sha256:c21fd414b0488637b1f05f13a59b032a3f9da5d818d31da1a4ca98a84c0c781b
Status: Downloaded newer image for swarm:latest
```

3. node1 をクラスタに追加します。

TCP ポート 2376 を指定します。2375 ではありません。

```
$ sudo docker run -d swarm join --addr=node1:2376 token://$TOKEN
7bacc98536ed6b4200825ff6f4004940eb2cec891e1df71c6bbf20157c5f9761
```

4. node2 をクラスタに追加します。

```
$ sudo docker run -d swarm join --addr=node2:2376 token://$TOKEN
db3f49d397bad957202e91f0679ff84f526e74d6c5bf1b6734d834f5edcbca6c
```

ステップ７：TLS を使う Swarm マネージャの作成

1. TLS を有効にした新しいコンテナを起動します。

```
$ docker run -d -p 3376:3376 -v /home/ubuntu/.certs:/certs:ro swarm manage \
  --tlsverify --tlscacert=/certs/ca.pem --tlscert=/certs/cert.pem --tlskey=/certs/key.pem \
  --host=0.0.0.0:3376 token://$TOKEN
```

このコマンドは swarm イメージを元にした新しいコンテナを起動します。そして、サーバ側のポート 3376 をコンテナ内のポート 3376 に割り当てます。コンテナは Swarm の manage プロセスを実行し、オプションとして --tlsverify、--tlscacert、--tlscert、--tlskey を指定します。これらのオプションは TLS 認証を強制するものであり、Swarm マネージャの TLS 鍵の場所を指定します。

2. docker ps コマンドを実行し、Swarm マネージャ用コンテナが起動して実行中かを確認します。

```
$ docker ps
CONTAINER ID   IMAGE               COMMAND                  CREATED          STATUS          PORTS
                    NAMES
035dbf57b26e   swarm               "/swarm manage --tlsv"   7 seconds ago    Up 7 seconds    2375/tcp,
0.0.0.0:3376->3376/tcp   compassionate_lovelace
```

これで Swarm クラスタが TLS を使う設定になりました。

ステップ８：Swarm マネージャの設定を確認

TLS を使う Swarm クラスタを構築しました。次は、Docker Engine CLI で動作するかを確認します。

1. client サーバのターミナルを開きます。

2. docker version コマンドを実行します。

コマンドの実行には、クライアント証明書の場所指定が必須です。

```
$ sudo docker --tlsverify --tlscacert=/home/ubuntu/.certs/ca.pem  \
  --tlscert=/home/ubuntu/.certs/cert.pem --tlskey=/home/ubuntu/.certs/key.pem -H swarm:3376 version
Client:
 Version:      1.9.1
 API version:  1.21
 Go version:   go1.4.2
 Git commit:   a34a1d5
 Built:        Fri Nov 20 13:12:04 UTC 2015
 OS/Arch:      linux/amd64

Server:
 Version:      swarm/1.0.1
 API version:  1.21
 Go version:   go1.5.2
 Git commit:   744e3a3
 Built:
 OS/Arch:      linux/amd64
```

Server バージョンの出力は “ swarm/1.0.1 ” を表示します。つまり、Swarm マネージャに対するコマンドの実行が成功したのを意味します。

3. TLS の指定が無くてもコマンドが動作するか確認します。

今回は Swarm マネージャ用の証明書を指定しません。

```
$ sudo docker -H swarm:3376 version
:
 Version:      1.9.1
 API version:  1.21
 Go version:   go1.4.2
 Git commit:   a34a1d5
 Built:        Fri Nov 20 13:12:04 UTC 2015
 OS/Arch:      linux/amd64
```

```
Get http://swarm:3376/v1.21/version: malformed HTTP response "\x15\x03\x01\x00\x02\x02".
* Are you trying to connect to a TLS-enabled daemon without TLS?
```

サーバ側のコマンドを拒否したと表示されます。つまり、サーバ（Swarm マネージャ）と通信できるのは TLS を用いるクライアントのみです。

ステップ９：TLS を使う Engilne CLI の設定

コマンド実行時に TLS オプションを指定しなくても良いよう、Engine 側に設定できます。設定のためには、Docker Engine クライアントがデフォルトで TLS を使うように、Docker Engine のホストの設定をします。

そのためには、クライアントの鍵を自分の ~/.docker 設定ディレクトリに置きます。システム上で他にも Engine コマンドを使っているユーザがいる場合は、それぞれのアカウントでも同様に ~/.docker の設定が必要です。以降は、ubuntu ユーザで Docker Engine クライアントを使う手順です。

1. client サーバのターミナルを開きます。

2. ubuntu ユーザのホームディレクトリに .docker ディレクトリが存在しなければ作成します。

```
$ mkdir /home/ubuntu/.docker
```

3. /home/ubuntu/.certs にある Docker Engine クライアントの鍵を、 /home/ubuntu/.docker にコピーします。

```
$ cp /home/ubuntu/.certs/{ca,cert,key}.pem /home/ubuntu/.docker
```

4. アカウントの ~/.bash_profile を編集します。

5. 以下の環境変数を指定します。

変数	説明
DOCKER_HOST	全ての Engine 用コマンドが送信する Docker ホストと TCP ポートを指定します。
DOCKER_TLS_VERIFY	Engine に TLS を使うと伝えます。
DOCKER_CERT_PATH	TLS 鍵の場所を指定します。

例：

```
export DOCKER_HOST=tcp://swarm:3376
export DOCKER_TLS_VERIFY=1
export DOCKER_CERT_PATH=/home/ubuntu/.docker/
```

6. ファイルを保存して閉じます。

7. 新しい環境変数をファイルから読み込みます。

```
$ source ~/.bash_profile
```

8. docker version コマンドを実行して動作確認します。

```
$ docker version
Client:
 Version:      1.9.1
 API version:  1.21
```

```
 Go version:   go1.4.2
 Git commit:   a34a1d5
 Built:        Fri Nov 20 13:12:04 UTC 2015
 OS/Arch:      linux/amd64

Server:
 Version:      swarm/1.0.1
 API version:  1.21
 Go version:   go1.5.2
 Git commit:   744e3a3
 Built:
 OS/Arch:      linux/amd64
```

コマンド実行結果のサーバ情報にある部分から、Docker クライアントは TLS を使う Swarm マネージャに命令していると分かります。

おつかれ様でした。これで TLS を使う Docker Swarm クラスタができました。

6章

高度なスケジューリング

高度なスケジューリングを学ぶためには、 ストラテジとフィルタのドキュメントをご覧ください。

6.1 フィルタ

フィルタ（filter）とは Docker Swarm スケジューラに対して、ノードを使ってコンテナの作成・実行をするか伝えます。

6.1.1 設定可能なフィルタ

フィルタは**ノード・フィルタ（node filter）**と**コンテナ設定フィルタ（container configuration filter）**の２種類に分けられます。ノード・フィルタは Docker ホストの特徴、あるいは Docker デーモンの設定によって処理します。コンテナ設定フィルタはコンテナの特徴、あるいはホスト上で利用可能なイメージによって処理します。

各フィルタには名前があります。ノード・フィルタは、以下の２つです。

- constraint（ノードの制限）
- health（ノードが正常かどうか）

コンテナ設定フィルタは以下の通りです。

- affinity（親密さ）
- dependency（依存関係）
- port（ポート）

swarm manage コマンドで Swarm マネージャの起動する時、全てのフィルタが指定可能です。もしも Swarm に対して利用可能なフィルタを制限したい場合は、 --filter フラグと名前のサブセットを指定します。

```
$ swarm manage --filter=health --filter=dependency
```

コンテナ設定フィルタに一致するのは全てのコンテナが対象でです。フィルタが適用されるのは停止しているコンテナも含みます。コンテナによって使用されているノードを解放するには、ノード上からコンテナを削除する必要があります。

6.1.2 ノード・フィルタ

コンテナ作成時とイメージ構築時に、constraint か health フィルタを使い、コンテナをスケジューリングするノード群を選択できます。

constraint（制限）フィルタを使う

ノード制限（constraint；制限・制約の意味）は Docker のデフォルトのタグやカスタム・ラベルを参照します。デフォルトのタグとは docker info の情報を元にします。しばし Docker ホストの設定状態に関連付けられます。現在以下の項目をデフォルト・タグとして利用できます。

- node ノードを参照するための ID もしくは名前
- storagedriver
- executiondriver
- kernelversion
- operatingsystem

カスタム・ノード・ラベルは docker daemon 起動時に追加できます。実行例：

```
$ docker daemon --label com.example.environment="production" --label
com.example.storage="ssd"
```

そして、クラスタ上でコンテナの起動時に、これらのデフォルト・タグかカスタム・ラベルを使って制限（constraint）を指定可能です。Swarm スケジューラはクラスタ上に条件が一致するノードを探し、そこでコンテナを起動します。この手法は、いくつもの実践的な機能になります。

- ホスト・プロパティを指定した選択（storage=ssd のように、特定のハードウェアにコンテナをスケジュールするため）
- ノードの基盤に、物理的な場所をタグ付けする（region=us-east のように、指定した場所でコンテナを強制的に実行）
- 論理的なクラスタの分割（environment=production のように、プロパティの違いによりクラスタを複数のサブクラスタに分割）

ノード制限の例

ノードに対してカスタム・ラベルを指定するには、docker 起動時に --label オプションのリストを指定します。例として、node-1 に storage=ssd ラベルを付けて起動します。

```
$ docker -d --label storage=ssd
$ swarm join --advertise=192.168.0.42:2375 token://XXXXXXXXXXXXXXXXXX
```

node-2 を storage=disk としても起動できます。

```
$ docker -d --label storage=disk
```

```
$ swarm join --advertise=192.168.0.43:2375 token://XXXXXXXXXXXXXXXXXX
```

ノードがクラスタに登録されたら、Swarm マネージャは個々のタグを取得します。マネージャは新しいコンテナをスケジューリングする時に、ここで取得したタグの情報を使って処理します。

先ほどのサンプルを例に進めましょう。クラスタには node-1 と node-2 があります。このクラスタ上に MySQL サーバ・コンテナを実行できます。コンテナの実行時、constraint（制限）を使い、データベースが良い I/O 性能を得られるようにできます。そのためには、フラッシュ・ドライブを持つノードをフィルタします。

```
$ docker tcp://<manager_ip:manager_port>  run -d -P -e constraint:storage==ssd --name db mysql
f8b693db9cd6

$ docker tcp://<manager_ip:manager_port>  ps
CONTAINER ID        IMAGE               COMMAND             CREATED                  STATUS
  PORTS                           NAMES
f8b693db9cd6        mysql:latest        "mysqld"            Less than a second ago   running
 192.168.0.42:49178->3306/tcp    node-1/db
```

この例では、マネージャは全てのノードの中から storage-ssd 制限に一致するノードを探し、そこに対してリソース管理を適用します。ここではホストがフラッシュ上で動いている node-1 のみが選ばれました。

クラスタのフロントエンドとして Nginx の実行をお考えでしょうか。この例では、フロントエンドはディスクのログを記録するだけですので、フラッシュ・ドライブを使いたくないでしょう。

```
$ docker tcp://<manager_ip:manager_port> run -d -P -e constraint:storage==disk --name frontend nginx
963841b138d8

$ docker tcp://<manager_ip:manager_port> ps
CONTAINER ID        IMAGE               COMMAND             CREATED                  STATUS
  PORTS                           NAMES
963841b138d8        nginx:latest        "nginx"             Less than a second ago   running
 192.168.0.43:49177->80/tcp      node-2/frontend
f8b693db9cd6        mysql:latest        "mysqld"            Up About a minute        running
  192.168.0.42:49178->3306/tcp    node-1/db
```

スケジューラは storage=disk ラベルを付けて起動済みの node-2 で起動します。

最後に、docker build の構築時の引数としてもノード制限を利用できます。今度もフラッシュ・ドライブを避けてみましょう。

```
$ mkdir sinatra
$ cd sinatra
$ echo "FROM ubuntu:14.04" > Dockerfile
$ echo "MAINTAINER Kate Smith <ksmith@example.com>" >> Dockerfile
$ echo "RUN apt-get update && apt-get install -y ruby ruby-dev" >> Dockerfile
$ echo "RUN gem install sinatra" >> Dockerfile
$ docker build --build-arg=constraint:storage==disk -t ouruser/sinatra:v2 .
Sending build context to Docker daemon 2.048 kB
Step 1 : FROM ubuntu:14.04
 ---> a5a467fddcb8
Step 2 : MAINTAINER Kate Smith <ksmith@example.com>
 ---> Running in 49e97019dcb8
 ---> de8670dcf80e
Removing intermediate container 49e97019dcb8
Step 3 : RUN apt-get update && apt-get install -y ruby ruby-dev
 ---> Running in 26c9fbc55aeb
```

```
 ---> 30681ef95fff
Removing intermediate container 26c9fbc55aeb
Step 4 : RUN gem install sinatra
 ---> Running in 68671d4a17b0
 ---> cd70495a1514
Removing intermediate container 68671d4a17b0
Successfully built cd70495a1514

$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
dockerswarm/swarm   manager             8c2c56438951        2 days ago          795.7 MB
ouruser/sinatra     v2                  cd70495a1514        35 seconds ago      318.7 MB
ubuntu              14.04               a5a467fddcb8        11 days ago         187.9 MB
```

health フィルタを使う

ノード health フィルタは障害の発生したノードにコンテナをスケジュールするの防ぎます。対象のノードはダウンしているか、クラスタ・ストアとの通信ができないことが考えられます。

6.1.3 コンテナ・フィルタ

コンテナの作成時、３種類のコンテナ・フィルタを使えます。

- affinity（親密さ）
- dependency（依存関係）
- port（ポート）

アフィニティ（親密さ）フィルタを使う

アフィニティ（affinity ＝親密さ）フィルタを使えば、コンテナ間を「集めて」作成できます。例えばコンテナを実行する時に、次の３つの親密さを元にして Swarm に対してスケジュールできます。

- コンテナ名か ID
- イメージのあるホスト
- コンテナに適用したカスタム・ラベル

これらのアフィニティ（親密さ）とは、コンテナを同じネットワーク・ノード上で実行することです。それぞれどのノード上で実行しているかどうか、知る必要がありません。

名前アフィニティの例

新しいコンテナを、既存のコンテナ名や ID を元にしてスケジューリングできます。例えば、frontend という名前のノードで nginx を実行します。

```
$ docker tcp://<manager_ip:manager_port>  run -d -p 80:80 --name frontend nginx
87c4376856a8

$ docker tcp://<manager_ip:manager_port> ps
CONTAINER ID        IMAGE               COMMAND             CREATED                  STATUS
  PORTS                            NAMES
87c4376856a8        nginx:latest        "nginx"             Less than a second ago   running
 192.168.0.42:80->80/tcp          node-1/frontend
```

それから、 -e affinity:container==frontend フラグを使い、２つめのコンテナを frontend の隣にスケジュールします。

```
$ docker tcp://<manager_ip:manager_port> run -d --name logger -e affinity:container==frontend logger
87c4376856a8

$ docker tcp://<manager_ip:manager_port> ps
CONTAINER ID        IMAGE               COMMAND             CREATED                  STATUS
  PORTS                              NAMES
87c4376856a8        nginx:latest        "nginx"             Less than a second ago   running
 192.168.0.42:80->80/tcp         node-1/frontend
 963841b138d8        logger:latest       "logger"            Less than a second ago   running
                                    node-1/logger
```

コンテナ名のアフィニティ指定によって、logger コンテナは frontend コンテナと同じ node-1 コンテナで実行されることになります。frontend という名前だけでなく、次のように ID を使った指定もできます

```
docker run -d --name logger -e affinity:container==87c4376856a8
```

イメージ・アフィニティの例

コンテナを起動する時、特定のイメージをダウンロード済みのノードのみにスケジュールすることができます。例えば、２つのホストに redis イメージをダウンロードし、３つめのホストに mysql イメージをダウンロードしたい場合があるでしょう。

```
$ docker -H node-1:2375 pull redis
$ docker -H node-2:2375 pull mysql
$ docker -H node-3:2375 pull redis
```

node-1 と node-3 のみが redis イメージを持っています。 -e affinity:image==redis フィルタを使い、これらのノード上でスケジュールします。

```
$ docker tcp://<manager_ip:manager_port> run -d --name redis1 -e affinity:image==redis redis
$ docker tcp://<manager_ip:manager_port> run -d --name redis2 -e affinity:image==redis redis
$ docker tcp://<manager_ip:manager_port> run -d --name redis3 -e affinity:image==redis redis
$ docker tcp://<manager_ip:manager_port> run -d --name redis4 -e affinity:image==redis redis
$ docker tcp://<manager_ip:manager_port> run -d --name redis5 -e affinity:image==redis redis
$ docker tcp://<manager_ip:manager_port> run -d --name redis6 -e affinity:image==redis redis
$ docker tcp://<manager_ip:manager_port> run -d --name redis7 -e affinity:image==redis redis
$ docker tcp://<manager_ip:manager_port> run -d --name redis8 -e affinity:image==redis redis

$ docker tcp://<manager_ip:manager_port> ps
CONTAINER ID        IMAGE               COMMAND             CREATED                  STATUS
  PORTS                              NAMES
87c4376856a8        redis:latest        "redis"             Less than a second ago   running
                                   node-1/redis1
1212386856a8        redis:latest        "redis"             Less than a second ago   running
                                   node-1/redis2
87c4376639a8        redis:latest        "redis"             Less than a second ago   running
                                   node-3/redis3
1234376856a8        redis:latest        "redis"             Less than a second ago   running
                                   node-1/redis4
86c2136253a8        redis:latest        "redis"             Less than a second ago   running
```

```
                                node-3/redis5
87c3236856a8        redis:latest        "redis"             Less than a second ago   running
                                node-3/redis6
87c4376856a8        redis:latest        "redis"             Less than a second ago   running
                                node-3/redis7
963841b138d8        redis:latest        "redis"             Less than a second ago   running
                                node-1/redis8
```

ここで見えるように、コンテナがスケジュールされるのは redis イメージを持っているノードのみです。イメージ名に加えて、特定のイメージ ID も指定できます。

```
$ docker images
REPOSITORY                          TAG                 IMAGE ID            CREATED
VIRTUAL SIZE
redis                               latest              06a1f75304ba        2 days ago
111.1 MB

$ docker tcp://<manager_ip:manager_port> run -d --name redis1 -e affinity:image==06a1f75304ba redis
```

ラベル・アフィニティの例

ラベル・アフィニティによって、コンテナのラベルで引き寄せたセットアップが可能です。例えば、nginx コンテナに com.example.type=frontend ラベルを付けて起動します。

```
$ docker tcp://<manager_ip:manager_port> run -d -p 80:80 --label com.example.type=frontend nginx
87c4376856a8

$ docker tcp://<manager_ip:manager_port> ps  --filter "label=com.example.type=frontend"
CONTAINER ID        IMAGE               COMMAND             CREATED                  STATUS
  PORTS                           NAMES
87c4376856a8        nginx:latest        "nginx"             Less than a second ago   running
 192.168.0.42:80->80/tcp         node-1/trusting_yonath
```

それから、 -e affinity:com.example.type==frontend を使って、com.example.type==frontend ラベルを持つコンテナの隣にスケジュールします。

```
$ docker tcp://<manager_ip:manager_port> run -d -e affinity:com.example.type==frontend logger
87c4376856a8

$ docker tcp://<manager_ip:manager_port> ps
CONTAINER ID        IMAGE               COMMAND             CREATED                  STATUS
  PORTS                           NAMES
87c4376856a8        nginx:latest        "nginx"             Less than a second ago   running
 192.168.0.42:80->80/tcp         node-1/trusting_yonath
963841b138d8        logger:latest       "logger"            Less than a second ago   running
                                node-1/happy_hawking
```

logger コンテナは、最終的に node-1 に置かれます。これはアフィニティに com.example.type==frontend ラベルを指定しているからです。

dependency フィルタを使う

コンテナの**依存関係（dependency）フィルタ**は、既にスケジューリング済みのコンテナと同じ場所でスケジューリングするという依存関係をもたらします。現時点では、以下の依存関係を宣言できます。

- --volumes-from=dependency(共有ボリューム)
- --link=dependency:alias(リンク機能)
- --net=container:dependency(共有ネットワーク)

Swarm は依存関係のあるコンテナを同じノード上に置こうとします。もしそれができない場合（依存関係のあるコンテナが存在しない場合や、ノードが十分なリソースを持っていない場合)、コンテナの作成を拒否します。

必要であれば、複数の依存関係を組み合わせることもできます。例えば、 --volumes-from=A --net=container:B は、コンテナ A と B を同じノード上に置こうとします。しかし、これらのコンテナが別々のノードで動いているなら、Swarm はコンテナのスケジューリングを行いません。

port フィルタを使う

ポート (port) フィルタが有効であれば、コンテナのポート利用がユニークになるよう設定します。Docker Swarm は対象のポートが利用可能であり、他のコンテナのプロセスにポートが専有されていないノードを選びます。ホスト側にポート番号を割り当てたい場合や、ホスト・ネットワーキング機能を使っている場合は、対象ポートの明示が必要になるかもしれません。

ブリッジ・モードでの例

デフォルトでは、コンテナは Docker のブリッジ・ネットワーク上で動作します。ブリッジ・ネットワーク上で port フィルタを使うには、コンテナを次のように実行します。

```
$ docker tcp://<manager_ip:manager_port> run -d -p 80:80 nginx
87c4376856a8

$ docker tcp://<manager_ip:manager_port> ps
CONTAINER ID    IMAGE              COMMAND         PORTS                          NAMES
87c4376856a8    nginx:latest       "nginx"         192.168.0.42:80->80/tcp
node-1/prickly_engelbart
```

Docker Swarm はポート 80 が利用可能であり他のコンテナ・プロセスに専有されていないノードを探します。この例では node-1 にあたります。ポート 80 を使用する他のコンテナを起動しようとしても、Swarm は他のノードを選択します。理由は node-1 では既にポート 80 が使われているからです。

```
$ docker tcp://<manager_ip:manager_port> run -d -p 80:80 nginx
963841b138d8

$ docker tcp://<manager_ip:manager_port> ps
CONTAINER ID        IMAGE          COMMAND        PORTS                          NAMES
963841b138d8        nginx:latest   "nginx"        192.168.0.43:80->80/tcp        node-2/dreamy_turing
87c4376856a8        nginx:latest   "nginx"        192.168.0.42:80->80/tcp
node-1/prickly_engelbart
```

同じコマンドを繰り返しますと node-3 が選ばれます。これは node-1 と node-2 の両方でポート 80 が使用済みのためです。

```
$ docker tcp://<manager_ip:manager_port> run -d -p 80:80 nginx
963841b138d8

$ docker tcp://<manager_ip:manager_port> ps
CONTAINER ID   IMAGE              COMMAND        PORTS                          NAMES
f8b693db9cd6   nginx:latest       "nginx"        192.168.0.44:80->80/tcp
```

```
node-3/stoic_albattani
963841b138d8   nginx:latest        "nginx"        192.168.0.43:80->80/tcp         node-2/dreamy_turing
87c4376856a8   nginx:latest        "nginx"        192.168.0.42:80->80/tcp
node-1/prickly_engelbart
```

最終的に、Docker Swarm は他のコンテナがポート 80 を要求しても拒否するでしょう。クラスタ上の全てのノードでポートが使えないためです。

```
$ docker tcp://<manager_ip:manager_port> run -d -p 80:80 nginx
2014/10/29 00:33:20 Error response from daemon: no resources available to schedule container
```

各ノード中のポート 80 は、各コンテナによって専有されています。これはコンテナ作成時からのものであり、コンテナを削除するとポートは解放されます。コンテナが exited（終了）の状態であれば、まだポートを持っている状態です。もし node-1 の prickly_engelbart が停止したとしても、ポートの情報は削除されないため、node-1 上でポート 80 を必要とする他のコンテナの起動を試みても失敗します。nginx インスタンスを起動するには、prickly_engelbart コンテナを再起動するか、あるいは prickly_engelbart コンテナを削除後に別のコンテナを起動します。

ホスト・ネットワーキング機能とノード・ポート・フィルタを使う

コンテナ実行時に --net=host を指定したら、デフォルトの bridge モードとは違い、host モードはどのポートも拘束しません。そのため、host モードでは公開したいポート番号を明示する必要があります。このポート公開には Dockerfile で EXPOSE 命令を使うか、コマンドラインで --expose を指定します。Swarm は host モードで新しいコンテナを作成しようとする時にも、これらの情報を利用します。

例えば、以下のコマンドは３つのノードのクラスタで nginx を起動します。

```
$ docker tcp://<manager_ip:manager_port> run -d --expose=80 --net=host nginx
640297cb29a7
$ docker tcp://<manager_ip:manager_port> run -d --expose=80 --net=host nginx
7ecf562b1b3f
$ docker tcp://<manager_ip:manager_port> run -d --expose=80 --net=host nginx
09a92f582bc2
```

docker ps コマンドを実行してもポートをバインド(拘束)している情報が表示されないのは､全てのノードで host ネットワークを利用しているためです。

```
$ docker tcp://<manager_ip:manager_port> ps
CONTAINER ID        IMAGE               COMMAND                 CREATED                  STATUS
     PORTS               NAMES
640297cb29a7        nginx:1             "nginx -g 'daemon of   Less than a second ago   Up 30 seconds
                      box3/furious_heisenberg
7ecf562b1b3f        nginx:1             "nginx -g 'daemon of   Less than a second ago   Up 28 seconds
                      box2/ecstatic_meitner
09a92f582bc2        nginx:1             "nginx -g 'daemon of   46 seconds ago           Up 27 seconds
                       box1/mad_goldstine
```

４つめのコンテナを起動しようとしても、Swarm は処理を拒否します。

```
$  docker tcp://<manager_ip:manager_port> run -d --expose=80 --net=host nginx
FATA[0000] Error response from daemon: unable to find a node with port 80/tcp available in the Host mode
```

しかしながら、例えばポート 81 のように、異なった値のポートをバインドするのであれば、コマンドを実行で

きます。

```
$ docker tcp://<manager_ip:manager_port> run -d -p 81:80 nginx:latest
832f42819adc
$ docker tcp://<manager_ip:manager_port> ps
CONTAINER ID        IMAGE               COMMAND                  CREATED                  STATUS
          PORTS                                  NAMES
832f42819adc        nginx:1             "nginx -g 'daemon of    Less than a second ago   Up Less than a
second    443/tcp, 192.168.136.136:81->80/tcp   box3/thirsty_hawking
640297cb29a7        nginx:1             "nginx -g 'daemon of    8 seconds ago            Up About a minute
                                            box3/furious_heisenberg
7ecf562b1b3f        nginx:1             "nginx -g 'daemon of    13 seconds ago           Up About a minute
                                            box2/ecstatic_meitner
09a92f582bc2        nginx:1             "nginx -g 'daemon of    About a minute ago       Up About a minute
                                            box1/mad_goldstine
```

6.1.4 フィルタ表現の書き方

ノード constraint やコンテナ affinity フィルタをノードに適用するには、コンテナがフィルタ表現を使うため環境変数の指定が必要です。例：

```
$ docker tcp://<manager_ip:manager_port> run -d --name redis1 -e affinity:image==~redis redis
```

表現は次のような記述方式です。

```
<フィルタ・タイプ>:<キー><演算子><値>
```

<フィルタ・タイプ> は affinity か constraint のキーワードのどちらかです。使いたいフィルタのタイプによって異なります。

<キー> は英数字のパターンであり、先頭はアルファベットかアンダースコアです。 <キー> に相当するのは以下の条件です。

- container キーワード
- node キーワード
- デフォルト・タグ（node 制限）
- カスタム・メタデータ・ラベル（node あるいは containers）

<オペレータ>（演算子）は == か != のどちらかです。デフォルトではフィルタ処理はハード・エンフォース（hard enforced:強制）です。指定した表現に一致しなければ、マネージャはコンテナをスケジュールしません。 ~ （チルダ）を使い 「ソフト」表現を作成できます。こちらはフィルタ条件に一致しなくても、スケジューラ自身のストラテジに従ってコンテナを実行します。

<値> は英数字、ドット、ハイフン、アンダースコアと、以下を組み合わせせた文字列です。

- 部分一致、例えば abc*。
- /regexp/ 形式の正規表現。Go 言語の正規表現構文をサポート。

現時点の Swarm は、以下のような命令をサポートしています。

- constraint:node==node1 は、ノード node1 に一致。
- constraint:node!=node1 は、node1 をのぞく全てのノードに一致。
- constraint:region!=us* は、us が付いているリージョン以外のノードに一致。
- constraint:node==/node[12]/ は、node1 と node2 に一致。
- constraint:node==/node\d/ は、node + 10 進数の 1 文字に一致。
- constraint:node!=/node-[01]/ は、node-0 と node-1 以外の全てのノードに一致。
- constraint:node!=/foo\[bar\]/ は、foo[var] 以外の全てのノードに一致。
- constraint:node==/(?i)node1/ は、大文字・小文字を区別しない node1 に一致。そのため、NoDe1 や NODE1 も一致する。
- affinity:image==~redis は、redis に一致する名前のイメージがあるノード上でコンテナを実行。
- constraint:region==~us* は、*us に一致するリージョンのノードを探す。
- affinity:container!=~redis* は、 redis* という名前を持つコンテナが動いていないノードで node5 コンテナをスケジュール。

6.2 ストラテジ

Docker Swarm スケジューラは、複数の**ストラテジ（strategy；方針）機能**でノードを順位付けします。選択したストラテジによって、Swarm が順位を算出します。Swarm で新しいコンテナを作成する時は、選択したストラテジに従って、コンテナを置くために最も順位の高いノードを算出します。

順位付けのストラテジを選択するには、swarm manage コマンドに --strategy フラグを使い、ストラテジ値を指定します。現時点でサポートしている値は次の通りです。

- spread（スプレッド）
- binpack（ビンパック）
- random（ランダム）

spread と binpack ストラテジは、ノードで利用可能な CPU、RAM、実行中のコンテナ数から順位を算出します。random ストラテジは計算をしません。ノードをランダムに選択するもので、主にデバッグ用に使います。

あなたの会社の必要性に従ってクラスタを最適化するのが、ストラテジを選択する目的（ゴール）です。

spread ストラテジの下では、Swarm はノードで実行中のコンテナ数に応じて最適化します。pinback ストラテジは利用するノードが最も少なくなるよう最適化します。random ストラテジでは、利用可能な CPU やメモリに関わらずランダムにノードを選びます。

spread ストラテジを使えば、結果として多くのマシンに幅広く展開します。このストラテジの利点は、ノードがダウンしても、失われるのは小数のコンテナだけです。

binpack は分散しないストラテジです。これは、使っていないマシンに大きなコンテナを動かす余地を残すためです。この binpack ストラテジの利点は、できるだけ少ないマシンしか使わないよう、Swarm はノードに多くのコンテナを詰め込みます。

もし --strategy を指定しなければ、Swarm はデフォルトで spread を使います

6.2.1 Spread ストラテジの例

この例では、Swarm は spread ストラテジでノードを最適化し、多くのコンテナを起動してみます。このクラスタは、node-1 と node-2 は 2GB のメモリ、2CPU であり、他のノードではコンテナは動いていません。このストラテジでは node-1 と node-2 は同じ順位です。

新しいコンテナを実行する時は、クラスタ上で同じランキングのノードが存在しますので、そこからランダムに node-1 をシステムが選びます。

```
$ docker tcp://<manager_ip:manager_port> run -d -P -m 1G --name db mysql
f8b693db9cd6

$ docker tcp://<manager_ip:manager_port> ps
CONTAINER ID        IMAGE               COMMAND             CREATED                  STATUS
  PORTS                           NAMES
f8b693db9cd6        mysql:latest        "mysqld"            Less than a second ago   running
 192.168.0.42:49178->3306/tcp    node-1/db
```

次は別のコンテナで 1GB のメモリを使います。

```
$ docker run tcp://<manager_ip:manager_port> -d -P -m 1G --name frontend nginx
963841b138d8

$ docker tcp://<manager_ip:manager_port> ps
```

```
CONTAINER ID        IMAGE               COMMAND             CREATED                  STATUS
  PORTS                           NAMES
963841b138d8        nginx:latest        "nginx"             Less than a second ago   running
 192.168.0.42:49177->80/tcp      node-2/frontend
f8b693db9cd6        mysql:latest        "mysqld"            Up About a minute        running
  192.168.0.42:49178->3306/tcp    node-1/db
```

コンテナ frontend は node-2 で起動します。これは直前に実行したノードだからです。もし２つのノードがあって、同じメモリや CPU であれば、spread ストラテジは最後にコンテナを実行したノードを選びます。

2.6.2 BinPack ストラテジの例

この例では、node-1 と node-2 いずれも 2GB のメモリを持ち、コンテナを実行していないとします。ノードが同じ時、コンテナの実行は、今回はクラスタから node-1 が選ばれたとします

```
$ docker run tcp://<manager_ip:manager_port> -d -P -m 1G --name db mysql
f8b693db9cd6

$ docker tcp://<manager_ip:manager_port> ps
CONTAINER ID        IMAGE               COMMAND             CREATED                  STATUS
  PORTS                           NAMES
f8b693db9cd6        mysql:latest        "mysqld"            Less than a second ago   running
 192.168.0.42:49178->3306/tcp    node-1/db
```

これで再び、1GB のメモリを使う別のコンテナを起動してみましょう。

```
$ docker run tcp://<manager_ip:manager_port> -d -P -m 1G --name frontend nginx
963841b138d8

$ docker tcp://<manager_ip:manager_port> ps
CONTAINER ID        IMAGE               COMMAND             CREATED                  STATUS
  PORTS                           NAMES
963841b138d8        nginx:latest        "nginx"             Less than a second ago   running
 192.168.0.42:49177->80/tcp      node-1/frontend
f8b693db9cd6        mysql:latest        "mysqld"            Up About a minute        running
  192.168.0.42:49178->3306/tcp    node-1/db
```

システムは node-1 上で新しい frontend コンテナを起動します。これはノードは既に集約するようになっているためです。これにより、2GB のメモリが必要なコンテナは node-2 で動きます。

もし２つのノードが同じメモリと CPU であれば、binpack ストラテジは最もコンテナが実行しているノードを選択します。

6.3 再スケジュール・ポリシー

Docker Swarm で再スケジュール・ポリシーを設定できます。再スケジュール・ポリシーとは、コンテナを実行中のノードが落ちた時、再起動するかどうかを決めます。

6.3.1 再スケジュール・ポリシー

コンテナ起動時に**再スケジュール・ポリシー（reschedule policy）**を設定できます。設定をするには `reschedule` 環境変数を指定するか、`com.docker.swarm.reschedule-policy` ラベルを指定します。ポリシーを指定しなければ、再スケジュール・ポリシーは `off` になります。つまり、ノードで障害が発生しても Swarm はコンテナを再スケジュールしません（再起動しません）。

`on-node-failure` ポリシーを環境変数 `reschedule` で設定するには、

```
$ docker run -d -e reschedule:on-node-failure redis
```

`com.docker.swarm.reschedule-policy` ラベルで同じポリシーを設定するには、

```
$ docker run -d -l 'com.docker.swarm.reschedule-policy=["on-node-failure"]' redis
```

6.3.2 再スケジュール・ログの確認

`docker logs` コマンドを使って再スケジュールしたコンテナの動作を確認できます。確認するには、次の構文を使います。

```
docker logs SWARM マネージャのコンテナ ID
```

コンテナの再スケジュールに成功したら、次のようなメッセージを表示します。

```
Rescheduled container 2536adb23 from node-1 to node-2 as 2362901cb213da321
Container 2536adb23 was running, starting container 2362901cb213da321
```

同様に、新しいノード上で新しいコンテナの起動に失敗したら、ログには次のように表示します。

```
Failed to start rescheduled container 2362901cb213da321
```

7章

リファレンス

7.1 Swarm コマンドライン・リファレンス

swarm - Docker ネイティブのクラスタリング・システム

swarm コマンドは Docker Engine のホスト上に Swarm コンテナを起動し、指定されたサブコマンドに応じたタスクを処理します。

swarm は次の構文で使います。

```
$ docker run swarm [オプション] コマンド [引数...]
```

例えば、swarm で manage サブコマンドを使えば Swarm マネージャを作成します。この時、クラスタ上にある他のマネージャと可用性を持たせるには：

```
$ docker run -d -p 4000:4000 swarm manage -H :4000 --replication \
  --advertise 172.30.0.161:4000 consul://172.30.0.165:8500
```

オプション

swarm コマンドには以下のオプションがあります。

- --debug：デバッグ・モードを有効にします。Swarm ノードのデバッグ用に使うメッセージを表示します。例： time=“ 2016-02-17T17:57:40Z ” level=fatal msg=“ discovery required to join a cluster. See ‘ swarm join – help ’.” 。このオプションは環境変数 ${DEBUG} でも指定可能です。
- --log-level "<値>" または -l "<値>"：ログレベルを指定します。 <値> に入るのは debug、info、warn、error、fatal、panic です。デフォルト値は info です。
- --experimental：実験的機能を有効にします。
- --help または -h：ヘルプを表示します。
- --version または -v：バージョン情報を表示します。例： $ docker run swarm – version swarm version 1.1.0 (a0fd82b)

コマンド

`swarm` コマンドには以下のサブコマンドがあります。

- create, c : ディスカバリ・トークンの作成
- list, l : Docker クラスタのノード一覧を表示
- manage, m : Swarm マネージャの作成
- join, j : Swarm ノードの作成
- help, h : Swarm コマンドの一覧を表示するか、各コマンドに対するヘルプを表示。

create - ディスカバリ・トークンの作成

create コマンドを実行したら、Docker Hub ホステット・ディスカバリ・バックエンドを使い、クラスタ用のユニークなディスカバリ・トークンを作成します。

```
$ docker run --rm  swarm create
86222732d62b6868d441d430aee4f055
```

後は Swarm マネージャやノード作成時に manage あるいは join コマンドで、このディスカバリ・トークンを <discovery> の引数として使います（例： token://86222732d62b6868d441d430aee4f055 ）。ディスカバリ・サービス・バックエンドを通して、Swarm マネージャはクラスタ上のノードを認識するために利用します。

このディスカバリ・トークンは、ドキュメントによっては clouster_id と記述されているかもしれません。

【警告】Docker Hub のホステッド・ディスカバリ・バックエンドはプロダクション環境での利用が推奨されていません。純粋にテスト・開発目的での利用を意図しています。

具体的な利用方法や他のバックエンドに関する情報は「4.3 Docker Swarm ディスカバリ」をご覧ください。

list - クラスタ上のノード一覧

list を使うとクラスタ上のノード一覧を表示します。
クラスタのノードを表示するには、次の構文を使います。

```
docker run swarm list [オプション] <discovery>
```

次の例は <discovery> 引数を使う構文の例です。

etcd:

```
swarm list etcd://<etcd_addr1>,<etcd_addr2>/<optional path prefix> <node_ip:port>
```

Consul:

```
swarm list consul://<consul_addr>/<optional path prefix> <node_ip:port>
```

ZooKeeper:

```
swarm list zk://<zookeeper_addr1>,<zookeeper_addr2>/<optional path prefix> <node_ip:port>
```

引数

list コマンドは引数が 1 つだけあります。

<discovery> - ディスカバリ・バックエンド

list コマンドの使用時、 <discovery> 引数を使って以下のバックエンドを指定可能です。

- token://<token>
- consul://<ip1>/<path>
- etcd://<ip1>,<ip2>,<ip2>/<path>
- file://<path/to/file>
- zk://<ip1>,<ip2>/<path>
- [nodes://]<iprange>,<iprange>

それぞれの項目は：

- <token> は Docker Hub のホステッド・ディスカバリ・サービスによって生成されるトークンです。このトークンを作成するには create コマンドを使います。

【警告】Docker Hub のホステッド・ディスカバリ・バックエンドは、プロダクション環境での利用が推奨されていません。テストもしくは開発用での利用を想定しています。

- ip1、ip2、ip3 はディスカバリ・バックエンド用ノードの IP アドレスとポート番号を指定します。

- path（オプション）はディスカバリ・バックエンドのキーバリュー・ストアのパスを指定します。複数のクラスタを 1 つのバックエンドで管理する場合は、各クラスタごとにキーバリューのペアを記述する必要があります。

- path/to/file は Swarm マネージャとクラスタのメンバであるノード情報の一覧と、それぞれの静的な IP アドレスのリストを指定したファイルのパスを指定します。

- iprange は特定のポート番号を利用する IP アドレスの範囲を指定します。

実行例：

- ディスカバリ・トークン： token://0ac50ef75c9739f5bfeeaf00503d4e6e
- Consul ノード： consul://172.30.0.165:8500

<discovery> は環境変数 $SWARM_DISCOVERY でも指定可能です。

より詳しい情報やサンプルについては、「4.3 Docker Swarm ディスカバリ」をご覧ください。

オプション

list コマンドには以下のオプションがあります。

--timeout - タイムアウト期間

--timeout="<期間>秒" はタイムアウトまでの感覚を秒で指定します。これは、ディスカバリ・バックエンドが一覧を返すまで待つ時間です。デフォルトの間隔は 10s です。

--discovery-opt - ディスカバリ・オプション

ディスカバリ・オプションに --discovery-opt <value> を使い、 TLS 設定で使用するファイル（CA 公開鍵証明書、証明書）のパスや、分散キーバリュー・ストアのディスカバリ・バックエンドを指定します。このオプションは複数回利用可能です。例：

```
--discovery-opt kv.cacertfile=/path/to/mycacert.pem \
--discovery-opt kv.certfile=/path/to/mycert.pem \
--discovery-opt kv.keyfile=/path/to/mykey.pem \
```

より詳しい情報は「4.3 Docker Swarm ディスカバリ」をご覧ください。

manage - Swarm マネージャの作成

動作条件：Swarm マネージャで manage を使う前に、ディスカバリ・バックエンドの構築が必要です。

manage コマンドは Swarm マネージャを作成します。マネージャはクラスタに対するコマンド受信と、Swarm ノードにコンテナを割り当てる役割があります。高可用性クラスタ用に複数の Swarm マネージャの作成も可能です。

Swarm マネージャを作成するには、以下の構文を使います。

```
$ docker run swarm manage [オプション] <discovery>
```

例えば、Swarm マネージャを manage で作成時に、他のマネージャと高可用性クラスタを形成するには、次のように実行します。

```
$ docker run -d -p 4000:4000 swarm manage -H :4000 --replication \
  --advertise 172.30.0.161:4000 consul://172.30.0.165:8500
```

あるいは、Swarm マネージャ作成時、Docker クライアントと Swarm ノード間で TLS 認証を有効にするには、次のように実行します。

```
$ docker run -d -p 3376:3376 -v /home/ubuntu/.certs:/certs:ro swarm manage \
  --tlsverify --tlscacert=/certs/ca.pem --tlscert=/certs/cert.pem --tlskey=/certs/key.pem \
  --host=0.0.0.0:3376 token://$TOKEN
```

引数

manage コマンドは引数を 1 つだけ指定できます。

<discovery> - ディスカバリ・バックエンド

Swarm マネージャを作成する前に、ディスカバリ・トークンの作成とディスカバリ・バックエンドのセットアップが必要です。

Swarm ノードの作成時、 <discovery> 引数を使って以下のバックエンドを指定可能です。

- token://<token>
- consul://<ip1>/<path>
- etcd://<ip1>,<ip2>,<ip2>/<path>
- file://<path/to/file>
- zk://<ip1>,<ip2>/<path>
- [nodes://]<iprange>,<iprange>

それぞれの項目は：

- <token> は Docker Hub のホステッド・ディスカバリ・サービスによって生成されるトークンです。このトークンを作成するには create コマンドを使います。
- ip1、ip2、ip3 はディスカバリ・バックエンド用ノードの IP アドレスとポート番号を指定します。
- path（オプション）はディスカバリ・バックエンドのキーバリュー・ストアのパスを指定します。複数のクラスタを 1 つのバックエンドで管理する場合は、各クラスタごとにキーバリューのペアを記述する必要があります。

- path/to/file は Swarm マネージャとクラスタのメンバであるノード情報の一覧と、それぞれの静的な IP アドレスのリストを指定したファイルのパスを指定します。

- iprange は特定のポート番号を利用する IP アドレスの範囲を指定します。

以下は <discovery> 引数の指定例です：

- ディスカバリ・トークン： token://0ac50ef75c9739f5bfeeaf00503d4e6e
- Consul ノード： consul://172.30.0.165:8500

<discovery> は環境変数$SWARM_DISCOVERY でも指定可能です。

より詳しい情報やサンプルについては、「4.3 Docker Swarm ディスカバリ」をご覧ください。

オプション

manage コマンドには以下のオプションがあります：

strategy - スケジュール先のストラテジ

--strategy "<値>" を使い、Docker Swarm スケジューラに対して何のストラテジを使うか指定します。

"値" の場所には：

- spread - 最も利用可能なリソースが多い Swarm ノードに対し、各コンテナを割り当てます。
- binpack - 割り当てられた Swarm ノードのリソースが溢れる前に、別のノードに割り当てます。
- random - ランダムな Swarm ノードにコンテナを割り当てます。

デフォルトでは、スケジューラは spread ストラテジを使います。

より詳しい情報はや例は「4.3 Docker Swarm ストラテジ」をご覧ください。

--filter 、 -f - スケジューラ・フィルタ

--filter <値> もしくは -f <値> で、コンテナを作成・実行時、どのノードを使うかを Docker Swarm スケジューラに対して指定します。

<値> の場所には：

- health - ディスカバリ・バックエンドと通信可能な実行中ノードを使います。
- port - コンテナにポートを割り当てるために、適切なポート番号が利用可能なノード（つまり、他のコンテナやプロセスにポートが専有されていない環境）を使います。
- dependency - 依存関係を宣言しているコンテナの場合、依存関係のあるコンテナが起動しているノードを使います。
- affinity - アフィニティが宣言されたコンテナの場合は、アフィニティが同一のノードを使います。

- constraint - 制約（constraint）が宣言されたコンテナの場合は、同一の制約を持つノードを使います。

複数のスケジューラ・フィルタを使うには、次のようにします。

```
--filter <value> --filter <value>
```

より詳しい情報や例は「6.1 フィルタ」をご覧ください。

--host , -H - リッスンする IP／ポート

--host <IP>:<ポート> もしくは -H <ip>:<ポート> を使い、マネージャがメッセージを受信するための IP アドレスとポート番号を指定します。ip の部分に 0 を使うか省略したら、manager はデフォルトのホスト IP を使います。例： --host=0.0.0.0:3376 または -H :4000 。

--host は環境変数$SWARM_HOST でも指定できます。

--replication - Swarm マネージャ複製の有効化

高可用性クラスタでは、プライマリとセカンダリ・マネージャ間で、Swarm マネージャの複製（レプリケーション）を可能にします。プライマリからセカンダリにクラスタ情報のミラーを複製します。つまりプライマリ・マネージャで障害が起これば、セカンダリがプライマリ・マネージャになれます。

--replication-ttl - リーダー障害発生時のロック解除時間

--replication-ttl "<遅延>s" を使い、遅延時間を秒で指定します。これはセカンダリ・マネージャがプライマリ・マネージャがダウンまたは到達可能と通知する時間です。この通知をトリガとして、セカンダリ・マネージャの誰がプライマリ・マネージャになるのか選出されます。デフォルトの遅延は 15 秒です。

--advertise , --addr - Docker Engine のアドバタイズ用 IP とポート番号

--advertise <ip>:<ポート> か --addr <ip>:<ポート> を使い Docker Engine のアドバタイズ(Advertise; 周知用) IP アドレスとポート番号を指定します。例： --advertise 172.30.0.161:4000 。他の Swarm マネージャは、対象の Swarm マネージャに接続するため、ここで指定した IP アドレスとポート番号を使う必要があります。

--advertise は環境変数$SWARM_ADVERTISE でも指定できます。

--tls - TLS の有効化

--tls を使い TLS（トランスポート・レイヤ・セキュリティ）を有効化します。 --tlsverify を使う場合は --tls の使用は不要です。

--tlscacert - 証明局(CA) の公開鍵ファイルのパス

--tlscacert=<path/file> を使い証明局（CA）用の公開鍵（証明書）のパスとファイル名を指定します。例： --tlscacert=/certs/ca.pem 。指定したら、マネージャが信頼するのは、同じ証明局で署名された証明書を使っているリモート環境のみです。

--tlscert - ノードの TLS 証明書ファイルのパス

--tlskey を使い、マネージャの証明書（CA によって署名済み）のファイル名とパスを指定します。例： --tlskey=/certs/key.pem 。

--tlskey - ノードの TLS 鍵ファイルのパス

--tlskey を使いマネージャの秘密鍵（CA によって署名済み）のファイル名とパスを指定します。例：--tlskey=/certs/key.pem 。

--tlsverify - TLS を使いリモート環境を確認

--tlsverify を使い TLS 通信を有効化し、同一の証明局（CA）で署名された証明書を持っているマネージャ、ノード、クライアントのみ通信を許可します。 --tlsverify を使えば、 --tls を使う必要はありません。

--engine-refresh-min-interval - Engine の最小リフレッシュ間隔を指定

--engine-refresh-min-interval "<間隔>s" を使い、Engine を例フレッシュするまでの最小間隔を秒単位で指定します。デフォルトでは、この間隔は 30 秒です。

プライマリ・マネージャが Engine をリフレッシュするというのは、クラスタ上にある Engine の情報を更新することです。マネージャはこの情報を Engine が正常（healthy）かどうか決めるために使います。接続できなければ、マネージャは対象ノードを障害（unhealthy）とみなします。マネージャは指定した間隔ごとに再度 Engine 情報の更新を試みます。規定回数の再試行して Engine が応答するのであれば、再び Engine を正常（healthy）とみなします。もしそうでなければ、マネージャは再試行を停止し、対象の Engine を無視します。

--engine-refresh-max-interval - Engine 最大リフレッシュ間隔を指定

--engine-refresh-max-interval "<間隔>秒" を使い、リフレッシュまでの最大間隔を秒単位で指定します。デフォルトでは、この間隔は 60 秒です。

--engine-failure-retry - Engine のリトライ失敗回数

--engine-failure-retry "数値" を使い、Engine が障害とみなすまでの再試行の回数を指定します。デフォルトでは、3 回再試行します。

--engine-refresh-retry - 廃止予定

廃止予定； --engine-failure-retry "数値" の代わりに --engine-failure-retry を使います。デフォルトは 3 です。

--heartbeat - ハートビート間隔

--heartbeat "<間隔>s" を使い、マネージャとプライマリ・マネージャ間のハードビート間隔を秒単位で指定します。ハードビートとはマネージャが正常で到達可能であるかを確認します。デフォルトでは、この間隔は 60 秒です。

--api-enable-cors , --cors - リモート API の CORS ヘッダを有効化

--api-remote-cors か --cors を使い CORS（cross-origin resource sharing)ヘッダをリモート API に入れます。

--cluster-driver , -c - 使用するクラスタ・ドライバ

--cluster-driver "ドライバ" か -c "<ドライバ>" を使い、使用するクラスタ・ドライバを指定します。 <ドライバ> に指定できるのは、以下のどちらかです。

- swarm は Docker Swarm ドライバです。
- mesos-experimental は Mesos クラスタ・ドライバです。

デフォルトは swarm ドライバです。

Mesos ドライバの利用に関する詳しい情報は、Using Docker Swarm and Mesos[*1] をご覧ください。

--cluster-opt - クラスタ・ドライバのオプション

複数のクラスタ・ドライバのオプションを --cluster-opt <値> --cluster-opt <値> の形式で指定できます。

<値> の場所に入る項目は以下の通りです：

- swarm.overcommit=0.05 - リソースをオーバー・コミットする割合（パーセント）を指定します。デフォルト値は 0.05 であり、5 パーセントを意味します。
- swarm.createretry=0 - コンテナ作成に何度失敗すると障害とみなすかを指定します。デフォルトの値は 0 回の再試行です。
- mesos.address= - バインドする Mesos のアドレスを指定します。このオプションは環境変数 $SWARM_MESOS_ADDRESS でも指定できます。
- mesos.checkpointfailover=false - Mesos のチェックポインティング（checkpointing）を有効化します。これは、以前まで使っていたエクゼキュータの状態が復旧したら、スレーブが再接続できるようにします。この時、ディスク I/O を消費します。このオプションは環境変数$SWARM_MESOS_CHECKPOINT_FAILOVER でも指定できます。デフォルト値は false（無効）です。
- mesos.port= - Mesos がバインドするポートを指定します。このオプションは環境変数$SWARM_MESOS_PORT でも指定できます。
- mesos.offertimeout=30s - Mesos がタイムアウトと判断する秒を指定します。このオプションは環境変数 $SWARM_MESOS_OFFER_TIMEOUT でも指定できます。デフォルトの値は 30s です。
- mesos.offerrefusetimeout=5s - Mesos がリソースの再利用ができないと判断する秒を指定します。このオプションは環境変数$SWARM_MESOS_OFFER_REFUSE_TIMEOUT でも指定できます。デフォルトの値は 5s です。
- mesos.tasktimeout=5s - Mesos のタスク作成までのタイムアウトを秒で指定します。このオプションは環境変数$SWARM_MESOS_TASK_TIMEOUT でも指定できます。デフォルトの値は 5s です。
- mesos.user= - Mesos フレームワークのユーザ名を指定します。このオプションは環境変数 $SWARM_MESOS_USER でも指定できます。

--discovery-opt - ディスカバリ・オプション

--discovery-opt <値> を使いディスカバリオプションを指定します。これには Consul や etcd ディスカバリ・サービスが使用する TLS ファイル（CA 公開鍵証明書、証明書、プライベート鍵）の指定も含みます。ディスカバリ・オプションは何度も指定できます。例：

```
--discovery-opt kv.cacertfile=/path/to/mycacert.pem \
--discovery-opt kv.certfile=/path/to/mycert.pem \
--discovery-opt kv.keyfile=/path/to/mykey.pem \
```

より詳しい情報は「4.3 Docker Swarm ディスカバリ」をご覧ください。

*1 https://github.com/docker/swarm/blob/master/cluster/mesos/README.md

join - Swarm ノードの作成

動作条件：Swarm マネージャで manage を使う前に、ディスカバリ・バックエンドの構築が必要です。

join コマンドは Swarm ノードを作成します。Swarm ノードはクラスタ内でコンテナを実行する場所という役割があります。一般的なクラスタは複数の Swarm ノードを持ちます。

Swarm ノードを作成するには、次の構文を使います。

```
$ docker run swarm join [OPTIONS] <discovery>
```

例えば、Swarm ノードを高可用性クラスタ対応マネージャに接続するには、次のように実行します。

```
$ docker run -d swarm join --advertise=172.30.0.69:2375 consul://172.30.0.161:8500
```

あるいは、Swarm ノード作成時、Docker Swarm ノード間で TLS 認証の有効化は、次のように実行します。

```
$ sudo docker run -d swarm join --addr=node1:2376 token://86222732d62b6868d441d430aee4f055
```

引数

join コマンドは引数を 1 つだけ指定できます。

<discovery> - ディスカバリ・バックエンド

Swarm マネージャを作成する前に、ディスカバリ・トークンの作成とディスカバリ・バックエンドのセットアップが必要です。

Swarm ノードの作成時、 <discovery> 引数を使って以下のバックエンドを指定可能です。

- token://<token>
- consul://<ip1>/<path>
- etcd://<ip1>,<ip2>,<ip2>/<path>
- file://<path/to/file>
- zk://<ip1>,<ip2>/<path>
- [nodes://]<iprange>,<iprange>

それぞれの項目は：

- <token> は Docker Hub のホステッド・ディスカバリ・サービスによって生成されるトークンです。このトークンを作成するには create コマンドを使います。
- ip1、ip2、ip3 はディスカバリ・バックエンド用ノードの IP アドレスとポート番号を指定します。
- path（オプション）はディスカバリ・バックエンドのキーバリュー・ストアのパスを指定します。複数のクラスタを 1 つのバックエンドで管理する場合は、各クラスタごとにキーバリューのペアを記述する必要があります。
- path/to/file は Swarm マネージャとクラスタのメンバであるノード情報の一覧と、それぞれの静的な IP アドレスのリストを指定したファイルのパスを指定します。
- iprange は特定のポート番号を利用する IP アドレスの範囲を指定します。

以下は <discovery> 引数の指定例です：

- ディスカバリ・トークン： token://0ac50ef75c9739f5bfeeaf00503d4e6e
- Consul ノード： consul://172.30.0.165:8500

<discovery> は環境変数$SWARM_DISCOVERY でも指定可能です。
より詳しい情報やサンプルについては、「4.3 Docker Swarm ディスカバリ」をご覧ください。

オプション

join コマンドには以下のオプションがあります：

--advertise , --addr - Docker Engine のアドバタイズ用 IP とポート番号

--advertise <ip>:<ポート> か --addr <ip>:<ポート> を使い Docker Engine のアドバタイズ（Advertise; 周知用）IP アドレスとポート番号を指定します。例： --advertise 172.30.0.161:4000 。他の Swarm マネージャは、対象の Swarm マネージャに接続するため、ここで指定した IP アドレスとポート番号を使う必要があります。
--advertise は環境変数$SWARM_ADVERTISE でも指定できます。

--heartbeat - ハートビート間隔

--heartbeat "<間隔>s" を使い、マネージャとプライマリ・マネージャ間のハードビート間隔を秒単位で指定します。ハードビートとはマネージャが正常で到達可能であるかを確認します。デフォルトでは、この間隔は 60 秒です。

--ttl - エフェメラル・ノードの期限を設定

--ttl "<間隔>s" を使い、エフェメラル・ノードに対する TTL（time-to-live）間隔を秒数で指定します。デフォルトの間隔は 180s です。

--delay - 一斉登録しないようランダム遅延を [0s,delay] で指定

--delay "間隔<s>" の形式で、ディスカバリバックエンドがノードを登録するまで、ランダムに遅延させる最大秒数を指定します。多数のノードをデプロイする時、ランダムに登録間隔の遅延を発生することで、ディスカバリ・バックエンドが飽和しないように（応答不能にならないように）します。

--discovery-opt - ディスカバリ・オプション

--discovery-opt <値> を使いディスカバリオプションを指定します。これには Consul や etcd ディスカバリ・サービスが使用する TLS ファイル（CA 公開鍵証明書、証明書、プライベート鍵）の指定も含みます。ディスカバリ・オプションは何度も指定できます。例：

```
--discovery-opt kv.cacertfile=/path/to/mycacert.pem \
--discovery-opt kv.certfile=/path/to/mycert.pem \
--discovery-opt kv.keyfile=/path/to/mykey.pem \
```

より詳しい情報は「4.3 Docker Swarm ディスカバリ」をご覧ください。

help - コマンドに関する情報の表示

help コマンドは各コマンドの使い方に関する情報を表示します。
例えば、Swarm のオプション一覧を表示するには、次のように実行します：

```
$ docker run swarm --help
```

特定の Swarm コマンドのオプションに対する引数一覧を確認するには、次のように実行します。

```
$ docker run swarm <コマンド> --help
```

実行例：

```
$ docker run swarm list --help
Usage: swarm list [OPTIONS] <discovery>

List nodes in a cluster

Arguments:
   <discovery>    discovery service to use [$SWARM_DISCOVERY]
                   * token://<token>
                   * consul://<ip>/<path>
                   * etcd://<ip1>,<ip2>/<path>
                   * file://path/to/file
                   * zk://<ip1>,<ip2>/<path>
                   * [nodes://]<ip1>,<ip2>

Options:
   --timeout "10s"                          timeout period
   --discovery-opt [--discovery-opt option --discovery-opt option]  discovery options
```

7.2 Docker Swarm API

Docker Swarm API は Docker リモート API と広範囲の互換性があります。このセクションは、Swarm API と Docker リモート API 間の違いに関する概要を説明します。

エンドポイントが無い場合

いくつかのエンドポイントは未実装のため、その場合は 404 エラーを返します。

```
POST "/images/create" : "docker import" flow not implement
```

異なる動作をするエンドポイント

- GET "/containers/{name:.*}/json" : Node 追加時の新しいフィールドです。

```
"Node": {
     "Id": "ODAI:IC6Q:MSBL:TPB5:HIEE:6IKC:VCAM:QRNH:PRGX:ERZT:OK46:PMFX",
     "Ip": "0.0.0.0",
     "Addr": "http://0.0.0.0:4243",
     "Name": "vagrant-ubuntu-saucy-64",
    },
```

- GET "/containers/{name:.*}/json" : HostIP が 0.0.0.0 の場合、実際のノード IP アドレスに置き換えます。
- GET "/containers/json" : コンテナ名の前にノード名が付きます。
- GET "/containers/json" : HostIP が 0.0.0.0 の場合、実際のノード IP アドレスに置き換えます。
- GET "/containers/json" : 公式 swarm イメージを使ってコンテナを起動した場合、デフォルトでは表示しません。表示するには all-1 を使います。
- GET "/images/json" : – filter node=<ノード名> を使うことで、特定のノードのイメージ情報を表示します。
- POST "/containers/create" : HostConfig の CpuShares 設定で、コンテナに対する CPU コアの割当数を指定します。