オンプレミスとAWSのハイブリッド構成による Webサーバーのオフロード事例

オンプレミスでWebサービスを運用していると、突発的な高負荷時にすぐに追加のサーバーを用意できない、想定される最大負荷に応じてサーバーをあらかじめ用意しておくにはコストがかかりすぎるなど、スケールに関する問題に直面する場合があります。本資料では、オンプレミスとアマゾン ウェブ サービス（AWS）によるクラウド環境をネットワーク接続し、オンプレミスのWebサーバーをAWSにオフロードすることで必要なときに必要な台数のWebサーバーをAWS上で確保する事例をご紹介します。

INDEX

●第1部：基本構成 …… 1p
●第2部：自動スケールアウト …… 5p
●第3部：自動スケールイン …… 8p

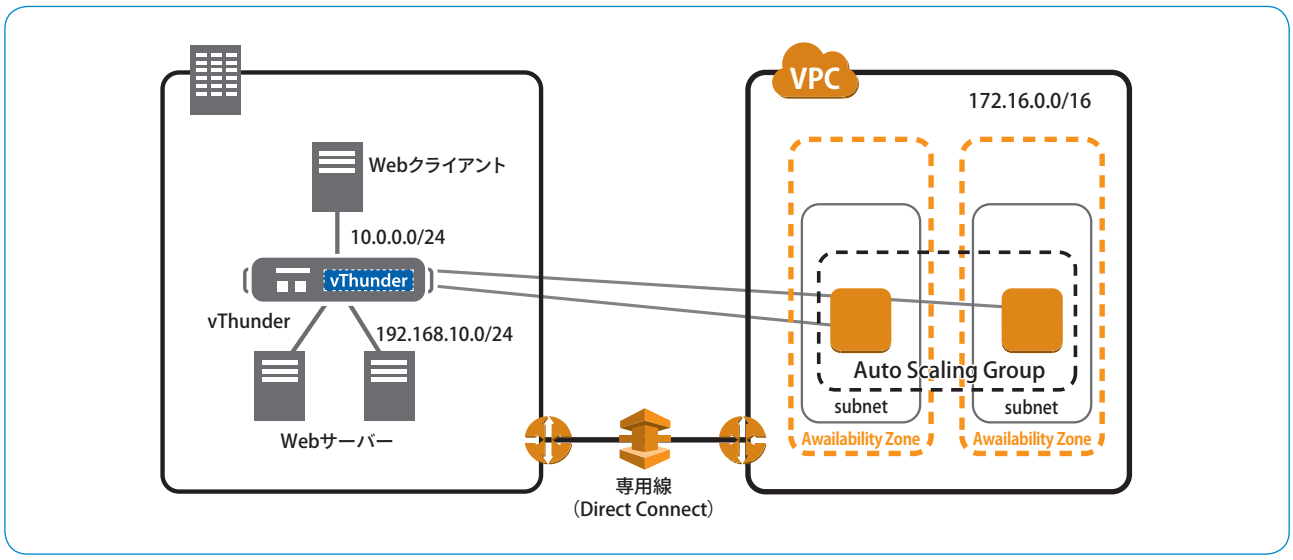
第1部：基本構成

本章では、AWS Direct Connect を利用したハイブリッド構成で、オンプレミスの Web サーバーを AWS にオフロードする基本構成をご紹介します。

■ 構成

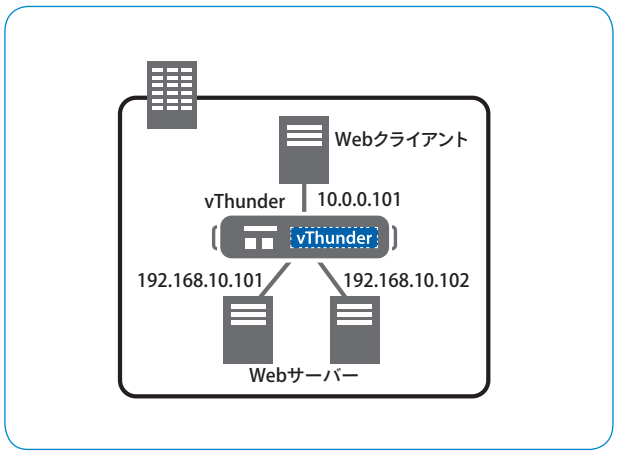
オンプレミスからAWSへのネットワーク接続は、専用線サービス「AWS Direct Connect」を利用します。オンプレミス上にはWebサーバーが常時2台稼働していますが、追加のWebサーバーが必要になった場合はAWSのVPC上にインスタンスを起動し、ロードバランサーに追加します。別途、動作確認用にWebクライアントをロードバランサーの上位に設置します 。

オンプレミス上のロードバランサーは、A10ネットワークスの提供するアプリケーションデリバリーコントローラー（ADC）「Thunder® ADC」のソフトウェア版「vThunder® ADC」を使用しています。



■ オンプレミス環境の準備

まずはオンプレミス上のADCとWebサーバー2台から構成されるWebサービスの環境を構築します。



仮想ADC vThunderの基本設定は以下の通りです。

```
interface ethernet 1
 description "WAN interface"
 ip address 10.0.0.100 255.255.255.0
!
interface ethernet 2
 description "LAN interface"
 ip address 192.168.10.20 255.255.255.0
!
health monitor http_index interval 3 timeout 1
 method http url GET /index.html expect response-code
200
!
slb server websv01 192.168.10.101
  health-check http_index
  port 80  tcp
!
slb server websv02 192.168.10.102
  health-check http_index
  port 80  tcp
!
slb service-group websvgrp tcp
   member websv01:80
   member websv02:80
!
slb virtual-server webservice 10.0.0.101
  disable when-all-ports-down
  port 80  tcp
    service-group websvgrp
```

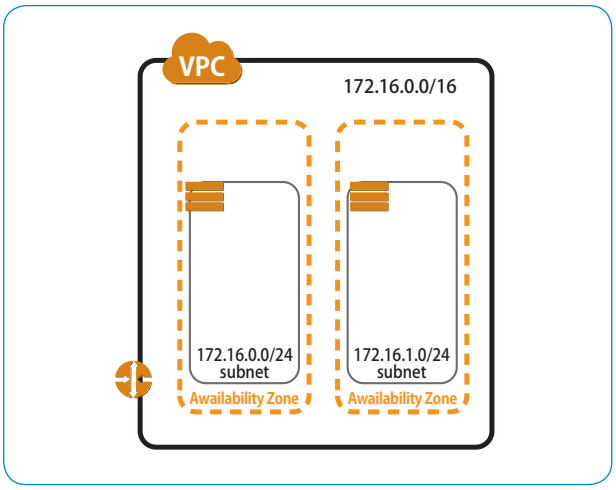
この状態で、"show slb virtual-server"コマンドなどでVIPがUPしているか確認し、かつWebクライアントからVIP10.0.0.101にアクセスできることを確認します。

```
$ curl -I http://10.0.0.101/
HTTP/1.1 200 OK
...
```

オンプレミス上でWebサービスを提供する準備ができました。

■ AWSのVPC設定

次に、AWS上でWebサーバーを起動するために、図のようなネットワーク環境を準備します。今回はAWS CLIを利用しました。



まずはVPCを作成します。

```
$ aws ec2 create-vpc --cidr-block 172.16.0.0/16
```

次に、Subnetを作成します。Subnetは2つのAvailability Zoneにそれぞれ1つずつ作成します。

```
$ aws ec2 create-subnet --vpc-id vpc-12345678
--availability-zone ap-northeast-1a --cidr-block
172.16.0.0/24
$ aws ec2 create-subnet --vpc-id vpc-12345678
--availability-zone ap-northeast-1c --cidr-block
172.16.1.0/24
```

Direct Connectを終端するために、VGW（Virtual Private Gateway）を作成し、VPCにアタッチします。

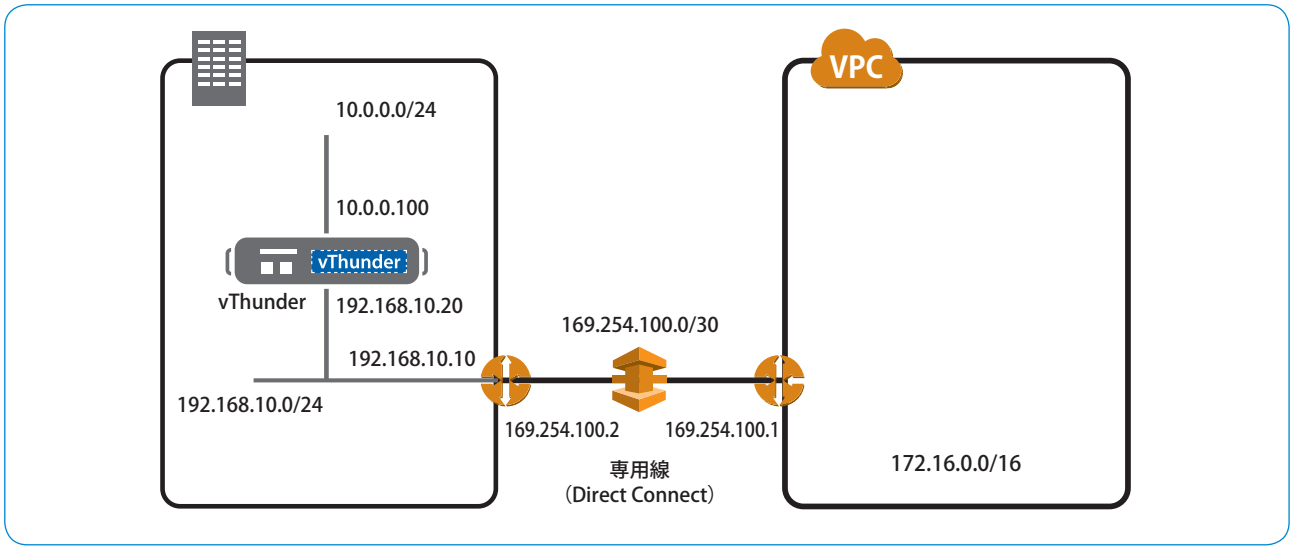
```
$ aws ec2 create-vpn-gateway --type ipsec.1
$ aws ec2 attach-vpn-gateway --vpn-gateway-id vgw-
12345678 --vpc-id vpc-12345678
```

新規でルーティングテーブルを作成し、それぞれのSubnetにアソシエートします。オンプレミスと通信するため、192.168.10.0/24宛のネットワークをVGWへ向けるようにルーティングエントリーを追加します。

```
$ aws ec2 create-route-table --vpc-id vpc-12345678
$ aws ec2 create-route --route-table-id rtb-12345678
--destination-cidr-block 192.168.10.0/24 --gateway-id
vgw-12345678
$ aws ec2 associate-route-table --route-table-id rtb-
12345678 --subnet-id subnet-12345678
$ aws ec2 associate-route-table --route-table-id rtb-
12345678 --subnet-id subnet-abcdefgh
```

■ Direct Connectの設定

オンプレミスとAWSをネットワーク接続するために、Direct Connectを設定します。オンプレミスとVPCのルーティングにはBGP（Border Gateway Protocol）を利用します。



Direct Connectの利用手続きは完了しているものとし、Virtual Interfaceの作成から解説します。

VLAN番号、AS番号、IPアドレスなど、接続に必要なパラメーターを指定し、Virtual Interfaceを作成します。前項で作成したVGWも同様に指定します。

```
$ aws directconnect create-private-virtual-interface \
--connection-id dx-xxxxxxx \
--new-private-virtual-interface virtualInterfaceName=vif,v
lan=100,asn=65001,authKey=aws123,amazonAddress=1
69.254.0.1/30,customerAddress=169.254.0.2/30,virtualGa
tewayId=vgw-12345678
```

オンプレミス側のDirect Connectを終端するルーターには、右記のようにインターフェース、ルーティング設定を行います（今回はCisco CSR1000Vを使用）。

```
!
interface GigabitEthernet1
 no ip address
 negotiation auto
!
interface GigabitEthernet1.100
 description "Direct Connect"
 encapsulation dot1Q 100
 ip address 169.254.100.2 255.255.255.252
!
interface GigabitEthernet2
 description "LAN"
 ip address 192.168.10.20 255.255.255.0
 negotiation auto
!
router bgp 65001
 bgp log-neighbor-changes
 network 192.168.10.0
 neighbor 169.254.100.1 remote-as 10124
 neighbor 169.254.100.1 password aws123
```

仮想ADC vThunderに、VPCに対するルーティング設定を行います。VPCのCIDR（172.16.0.0/16）宛に、Cisco CSR1000Vをネクストホップとしてstatic routeを設定します。

```
ip route 172.16.0.0 /16 192.168.10.10
```

vThuderはBGPに対応しているため、Direct Connectを直接終端することも可能です。

■ EC2インスタンスの起動/登録

AWS上でEC2™インスタンスを起動します。セキュリティグループはADCからのHTTPトラフィックを許可します。また、Webサーバーとして稼働するためにHTTPサーバーのインストールやコンテンツの配置も行います。

```
$ aws ec2 create-security-group --group-name sg_websv
--description "Security group for Web servers"
$ aws ec2 describe-security-groups --group-ids sg-
12345678
$ aws ec2 authorize-security-group-ingress --group-id sg-
12345678 --protocol tcp --port 80 --cidr 192.168.10.10/32
$ aws ec2 authorize-security-group-ingress --group-id sg-
12345678 --protocol tcp --port 22 --cidr 192.168.10.0/24
$ aws ec2 run-instances --image-id ami-xxxxxxxx --count 1
--instance-type t1.micro --key-name keyname --security-
group-ids sg-12345678 --subnet-id subnet-12345678
```

vThunderにEC2インスタンスを登録します。

```
slb server ec2websv01 172.16.0.101
 health-check http_index
 port 80 tcp
```

確認用のサーバーからvThunderのVIP宛にアクセスできるか確認してみます。オンプレミスで稼働しているWebサーバーを一旦サービスから外しておくと、EC2インスタンスへのアクセスのみを確認することができます。

```
$ curl -I http://10.0.0.101/
HTTP/1.1 200 OK
...
```

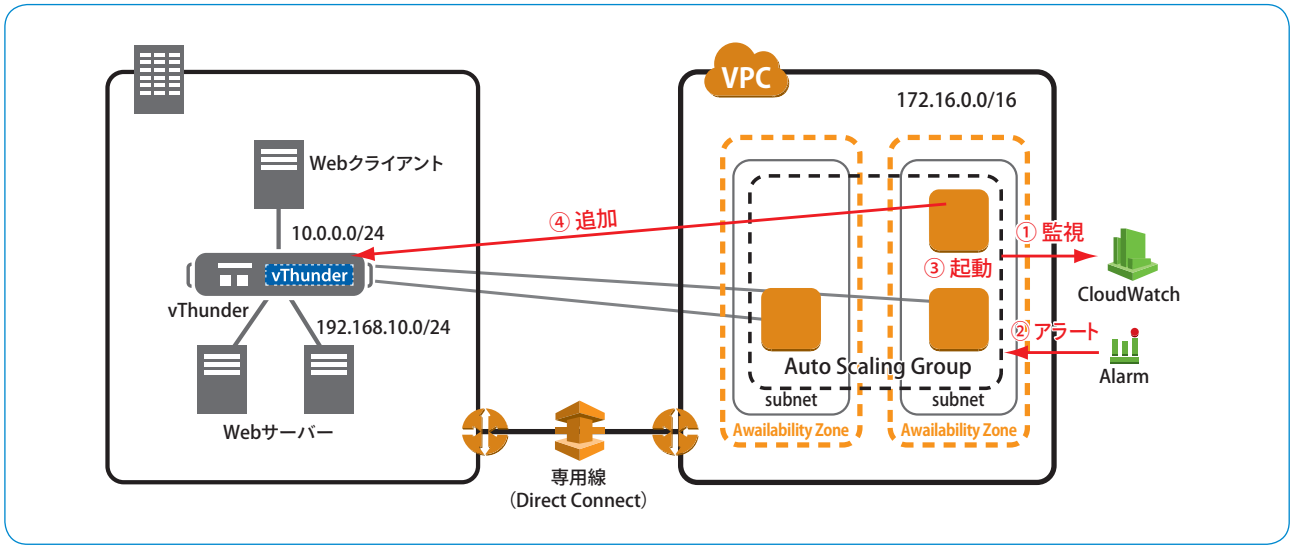
以上の手順により、AWS上のEC2インスタンスもWebサーバーとしてオンプレミスのADCから利用できることが確認できました。

第2部：自動スケールアウト

本章では突発的な負荷が発生した場合に備え、Auto Scalingを使って自動的にスケールアウトを行う方法をご紹介します。Auto Scaling Group内のEC2インスタンスの平均CPU使用率が70%を超えた場合にAWSのVPC上にインスタンスを追加し、仮想ADC vThunderに自分自身をWebサーバーとして登録します。CPU使用率の監視にはAmazon CloudWatchを利用します。

処理の流れは以下のとおりです。

1. CloudWatchでAuto Scaling Group内のインスタンスのCPU使用率を監視
2. CPU使用率が70%を超えるとアラートを生成
3. あらかじめ設定されたAuto Scalingにより、新規でインスタンスが起動
4. 新規で起動したインスタンスが自分自身をADCのサービスへ追加



■ API（aXAPI）によるロードバランサーの設定

vThunder仮想アプライアンス は aXAPI と呼ばれるRESTful APIを搭載しており、これを利用することによりリモートからコマンドを実行することができます。今回は、EC2インスタンスのメタデータから自身のプライベートIPアドレスを取得し、aXAPIを利用して自身をADCに追加するサンプルスクリプトを準備しました。

```
#!/bin/bash

a10_ip=192.168.10.10        #vThunderのIPアドレス
service_group=websvgrp      #vThunderで設定されている
サービスグループ
username=********           #vThunderのログインユーザー名
password=********           #vThunderのログインパスワード
```

```
url="https://$a10_ip/services/rest/V2/"

# TLSv1と証明書チェックをスキップするためのオプション
cmd_options='--tlsv1.0 --insecure'

# セッションIDの取得
session_id=`curl -v $cmd_options $url \
 -d method=authenticate \
 -d username=******** \
 -d password=******** | \
 sed -n -e 's/.*<session_id>\(.*\)<\/session_id>.*/\1/p'`

#自分のプライベートIPアドレス取得
my_ip=`curl http://169.254.169.254/latest/meta-data/
```

```
local-ipv4`
my_instanceid=`curl http://169.254.169.254/latest/meta-data/instance-id`

# サーバー登録
curl -v -X POST $cmd_options $url \
  -d session_id=$session_id \
  -d method=slb.server.create \
  -d name=$my_instanceid \
  -d host=$my_ip \
  -d status=1 \
  -d health_monitor=http_index \
  -d port_list=port1 \
  -d port1=port_num,protocol \
  -d port_num=80 \
  -d protocol=2

# サービスグループへメンバー登録
curl -v -X POST $cmd_options $url \
  -d session_id=$session_id \
  -d method=slb.service_group.member.create \
  -d name=$service_group \
  -d member=server,port \
  -d server=$my_ip \
  -d port=80
```

実際の運用時は、エラー処理などを考慮してスクリプトを作成することを推奨します。Pythonコードについては、弊社Webサイトをご参考ください。

後のステップで、作成したスクリプトをインスタンス起動時に実行するためUserData に登録します。

■ AMI化

Auto Scalingでは、自動的に起動するインスタンスのテンプレートイメージにAMI（Amazon Machine Image）を利用します。前項で作成したインスタンスからAMIを作成します。

```
$ aws ec2 create-image --instance-id i-12345678 --name "ec2websv" --description "AMI for web server running on EC2"
```

■ Auto Scalingの設定

まずはLaunch Configurationで、起動するWebサーバーを定義します。前項で作成したAMIを利用し、UserDataでは起動時にロードバランサーに追加するスクリプトを登録します。

```
$ aws autoscaling create-launch-configuration --launch-configuration-name ec2websv_launch_conf --image-id ami-14c6d315 --key-name key4tokyo --security-groups sg-7e4fe91b --user-data file:///filepath/userdata.txt --instance-type t2.micro --no-associate-public-ip-address
```

作成したLaunchConfigurationを利用し、2つのサブネット上に最小2台、最大4台までWebサーバーを起動するようにAutoSacling Groupを設定します。

```
$ aws autoscaling create-auto-scaling-group --auto-scaling-group-name ec2websv_asgrp --launch-configuration-name ec2websv_launch_conf --min-size 2 --max-size 4 --vpc-zone-identifier subnet-33946c44,subnet-451df91c
```

AutoScaling Group内のインスタンスの平均CPU使用率が70%を超えるとインスタンスを1つ追加するようにScaling PolicyとCloudWatchを設定します。CloudWatchの設定にはScaling PolicyのARNが必要なため、コマンドの結果を控えておきます。

```
$ aws autoscaling put-scaling-policy --policy-name ec2websv_scaleout --auto-scaling-group-name ec2websv_asgrp --scaling-adjustment 1 --adjustment-type ChangeInCapacity
{
    "PolicyARN": "arn:aws:autoscaling:ap-northeast-1:<AWS Account ID>:scalingPolicy:xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxxxxx:autoScalingGroupName/ec2websv_asgrp:policyName/ec2websv_scaleout"
}
```

```
$ aws cloudwatch put-metric-alarm  \
--period 300 \
--alarm-name scalout_alerm \
--dimensions Name=AutoScalingGroupName,Value=ec2
websv_asgrp \
--namespace "AWS/EC2" \
--metric-name CPUUtilization \
--evaluation-periods 1 \
--statistic Average \
--threshold 70 \
--comparison-operator GreaterThanThreshold \
--alarm-actions arn:aws:autoscaling:ap-northeast-1:<AWS
AccountID>:arn:aws:autoscaling:ap-northeast-1:<AWS
Account ID>:scalingPolicy:xxxxxxxx-xxxx-xxxx-xxxx-
xxxxxxxxxxxxxxx:autoScalingGroupName/ec2websv_
asgrp:policyName/ec2websv_scaleout
```

■ 動作確認

実際に負荷をかけてみて、AWS上のWebサーバー用のインスタンスが自動的に起動するか、起動したインスタンスがADC上にメンバーとして追加されているかを確認します。Webクライアントから以下のabコマンドで負荷をかけてみます。

```
$ ab -c 3 -n 1000 http://10.0.0.101/
```

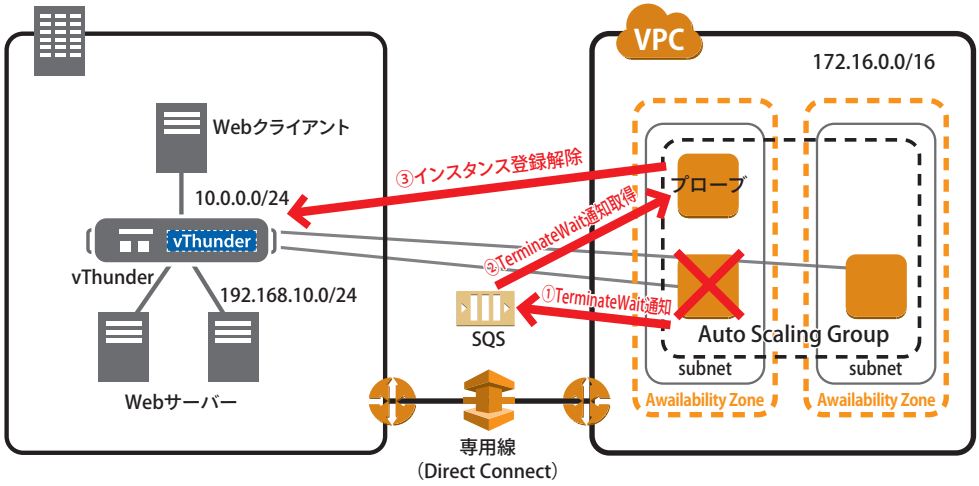
コマンド実行後しばらくすると、EC2インスタンスが追加で2台起動し、合計4台となっていることが確認できます。

第3部：自動スケールイン

本章では、負荷が落ち着いた際に追加した分のサーバーを自動的に削除するスケールインの動作についてご紹介します。Auto Scalingのスケールイン動作においては、CPU負荷のしきい値を下回った場合などのスケールインイベントが発生すると、対象となるインスタンスを自動的にTerminateします。しかし、ADCに追加されたままの状態でインスタンスをTerminateしてしまうと、ADCのヘルスチェック機能によりサービスから除外されるまでのわずかな間も該当のインスタンスへのトラフィックが転送され続けるため、アクセスエラーが一部発生します。エラーを避けるため、インスタンスのTerminateの前にADC上で無効化することにします。

Auto Scalingではスケールアウト・スケールインのイベントからインスタンスがStartまたはTerminateされる前に、何らかの処理を実行するために一時的に状態をフックするLifecycle Hook機能があります。
この機能を利用して、スケールイン対象となるインスタンスをADCから自動的に削除する機能を実装します。処理の流れは右記のとおりです。

1. スケールインイベント発生。Terminateが開始されるがTerminate状態を一時的にフックしSQSへ通知
2. プローブがSQSのキューを定期的にチェックしており、1で通知されたメッセージを受信
3. ADCへ該当のインスタンスを無効化・削除するためにAPIを実行



■ 事前準備

Auto Scalingのスケールインのイベントが発生した後、インスタンスが"Terminating"状態になったときにSQSに通知するようにLifecycle Hookの設定を行います。

まず最初に、Lifecycle Hookの通知を受け取るSQSのQueueを作成します。

```
$ aws sqs create-queue --queue-name websvtermination
```

Lifecycle Hookが利用するIAM Roleを設定します。フック時にSQSへメッセージをpublishする権限を付与します。Lifecycle設定が完了したらRoleのARNも控えてください。

```
{
 "Version": "2012-10-17",
 "Statement": [
  {
   "Effect": "Allow",
   "Action": [
    "sqs:SendMessage",
    "sqs:GetQueueUrl",
    "sns:Publish"
   ],
   "Resource": "*"
  }
 ]
}
```

次に、Lifecycle Hookの設定を行います。WebサーバーのAuto Scaling Groupを指定し、Terminating状態をフックするように設定します。

```
$ aws autoscaling put-lifecycle-hook \
 --lifecycle-hook-name hookTermination \
 --auto-scaling-group-name ec2websv_asgrp \
  --lifecycle-transition autoscaling:EC2_INSTANCE_
TERMINATING \
  --notification-target-arn arn:aws:sqs:ap-northeast-
1:123456789012:websvtermination \
  --role-arn arn:aws:iam::123456789012:role/
lifecyclehookRole
```

■ スケールインのイベント発生時の動作

実際にスケールインのイベントが発生すると、前項で設定したlifecycle Hookの設定により、SQSへ以下のようなメッセージがpublishされます。aws cliで中身を確認します。

```
$ aws sqs receive-message \
--queue-url="https://ap-northeast-1.queue.amazonaws.
com/123456789012/websvtermination"

{
 "Messages": [
  {
   "Body": "{
   \"AutoScalingGroupName\":\"ec2websv_asgrp\",
   \"Service\":\"AWS Auto Scaling\",
   \"Time\":\"2015-01-14T08:57:13.131Z\",
   \"AccountId\":\"123456789012\",
    \"LifecycleTransition\":\"autoscaling:EC2_INSTANCE_
TERMINATING\",
   \"RequestId\":\"xxxxxxxxxxxxxxxxxxxxxxxx\",
   \"LifecycleActionToken\":\"xxxxxxxxxxxxxxxxxxxxxxxx\",
    \"EC2InstanceId\":\"i-12345678\",\"LifecycleHookNam
e\":\"hookTermination\"}",
   "ReceiptHandle": "xxxxxxxxxxxxxxxxxxxxxxxx",
   "MD5OfBody": "xxxxxxxxxxxxxxxxxxxxxxxx",
   "MessageId": "xxxxxxxxxxxxxxxxxxxxxxxx"
  }
 ]
}
```

メッセージの各アイテムをパースし、ADCの削除を実行します。vThunder仮想アプライアンスには、サーバー名としてインスタンスIDが登録されているため、インスタンスIDをキーにしてADCからサーバーを削除します。

以下はvThunderのサーバー登録部分のコンフィグレーションです。

```
slb server i-12345678 172.16.1.55
  health-check http_index
  conn-limit 8000000 no-logging
  port 80  tcp
    health-check http_index
!
slb service-group websvgrp tcp
   member i-12345678:80
   member i-abcdefgh:80
!
```

ADCの設定からサーバーを削除するAPIを実行します。サーバーに接続中のコネクションを保護するため、サーバーをdisableにした後、コネクションが0の場合のみサーバーを削除することにします。こちらのスニペットをご参考ください。

```
url="https://$a10_ip/services/rest/V2/"

# TLSv1と証明書チェックをスキップするためのオプション
cmd_options='--tlsv1.0 --insecure'

# セッションIDの取得
session_id=`curl -v $cmd_options $url \
 -d method=authenticate \
 -d username=admin \
 -d password=a10 | \
 sed -n -e 's/.*<session_id>\(.*\)<\/session_id>.*/\1/p'`

# ADCに登録されているIPアドレス取得
ip=`curl -v -X GET $cmd_options $url \
   -d session_id=$session_id \
   -d method=slb.server.search \
   -d name=$instance_name | \
   sed -n -e 's/.*<host>\(.*\)<\/host>.*/\1/p'`

# サーバーをdisable
curl -v -X POST $cmd_options $url \
   -d session_id=$session_id \
   -d method=slb.server.update \
   -d name=$instanceid \
   -d host=$ip \
   -d status=0 \
   -d port_list=port1 \
   -d port1=port_num,protocol \
   -d port_num=80 \
   -d protocol=2

# 接続中のコネクション数を確認
conn=`curl -v -X GET $cmd_options $url \
   -d session_id=$session_id \
   -d method=slb.server.fetchStatistics \
   -d name=$instanceid |\
   sed -n -e 's/.*<cur_conns>\(.*\)<\/cur_conns>.*/\1/p'`

# 接続数が0ではない場合は終了
if [ $conn -ne 0 ]; then
   exit 1
fi

# サーバー削除
curl -v -X POST $cmd_options $url \
   -d session_id=$session_id \
   -d method=slb.server.delete \
   -d name=$instanceid
```

ADCから削除された後にフックされた状態を解除し、インスタンスのTerminateを再開する必要があります。

```
$ aws autoscaling complete-lifecycle-action --lifecycle-name hookTermination \
   --auto-scaling-group-name ec2websv_asgrp \
   --life-cycle-action-token xxxxxxxxxxxxxxxxxxxx \
   --life-cycle-action-result CONTINUE
```

■ スケールイン動作の自動化

以上の一連の処理をシェルスクリプトにまとめました。定期的にプローブで実行するように設定します。今回はEC2インスタンスをプローブとして設定しています。

SQSのメッセージ受信およびLifecycle Hookのコマンド実行権限を持ったIAM Roleをプローブのインスタンスに割り当てます。

こちらのスクリプトをプローブ上で定期的に実行するように設定します。Lifecycle Hookのメッセージを受信するためのSQSへのポーリングですが、ポーリングによる負荷を下げるため、ロングポーリング（コマンドでは--wait-time-secondsオプション）を付与しました。

Amazon SQS ロングポーリング

```
#!/bin/bash

a10_ip=192.168.10.10        #vThunderのIPアドレス
service_group=websvgrp      #vThunderで設定されている
サービスグループ
username=********           #vThunderのログインユーザー名
password=********           #vThunderのログインパスワード
url="https://$a10_ip/services/rest/V2/"

# SQSのQueue URL
queue_url="https://ap-northeast-1.queue.amazonaws.
com/123456789012/websvtermination"

# Queueのメッセージを取得
message=`aws sqs receive-message --queue-
url=$queue_url --wait-time-seconds=20 --output=text |
awk '{print $2}' | sed -n -e 's/.*{\(.*\)}.*/\1/p'`

# Queueにメッセージが無かったら終了
if [ -z "$message" ] then
   exit 0
```

```
fi

# メッセージからAutoscalingグループ名、LifecycleActionトークンとEC2InstanceIDを取得
ag_name=`echo $message | awk -F',' '{print $1}' | awk -F':' '{print $2'} | sed -e s/\"//g`
token=`echo $message | awk -F',' '{print $7}' | awk -F':' '{print $2'} | sed -e s/\"//g`
instanceid=`echo $message | awk -F',' '{print $8}' | awk -F':' '{print $2'} | sed -e s/\"//g`
lifecycle_name=`echo $message | awk -F',' '{print $9}' | awk -F':' '{print $2'} | sed -e s/\"//g`

# TLSv1と証明書チェックをスキップするためのオプション
cmd_options='--tlsv1.0 --insecure'

# セッションIDの取得
session_id=`curl -v $cmd_options $url \
 -d method=authenticate \
 -d username=admin \
 -d password=a10 | \
 sed -n -e 's/.*<session_id>\(.*\)<\/session_id>.*/\1/p'`

# ADCに登録されているIPアドレス取得
ip=`curl -v -X GET $cmd_options $url \
   -d session_id=$session_id \
   -d method=slb.server.search \
   -d name=$instance_name | \
   sed -n -e 's/.*<host>\(.*\)<\/host>.*/\1/p'`

# サーバー削除
curl -v -X POST $cmd_options $url \
   -d session_id=$session_id \
   -d method=slb.server.delete \
   -d name=$instanceid

# インスタンス終了をAuto Scalingへ通知
aws autoscaling complete-lifecycle-action --lifecycle-name $lifecycle_name \
   --auto-scaling-group-name $ag_name \
   --life-cycle-action-token $token \
   --life-cycle-action-result CONTINUE

# QueueからReceit Handleを取得し、Queueの中のメッセージを削除
receipt_handle=`echo $message | awk -F'\t' '{print $5}'`
aws sqs delete-message --queue-url $queue_url --receipt-handle $receipt_handle
```

本手順により、Auto Scalingのスケールイン時に vThunderに登録されていたインスタンスを自動的に無効化・削除することができます。

■ スケールアウト時のLifecycle Hook利用

第2部では、スケールアウト時にADCへインスタンスを登録する際にUserDataを利用しました。今回利用したLifecycle Hookを利用して、起動状態をフックし、ロードバランサーに登録する方法もあります。スケールイン、スケールアウトそれぞれLifecycle Hookで統一する方が管理上好ましい場合もあります。

■ さいごに

本資料では、3章にわたりオンプレミスで稼働しているWebサービスをAWS上にオフロードする事例をご紹介しました。AWS Direct Connectを利用することでオンプレミスの機器とAWSのリソースをシームレスに組み合わせて一つのシステムを構成することができます。vThunderのようにAPIを利用して設定ができるアプライアンスであれば、AWSのAPIと組み合わせてオペレーションを自動化することも可能です。ぜひご自分の環境でもお試しください。

A10 Networks/A10ネットワークス株式会社について

A10 Networks（NYSE: ATEN）はアプリケーションネットワーキング分野におけるリーダーとして、高性能なアプリケーションネットワーキングソリューション群を提供しています。
世界中で数千社にのぼる大企業やサービスプロバイダー、大規模Webプロバイダーといったお客様のデータセンターに導入され、アプリケーションとネットワークを高速化し安全性を確保しています。A10 Networksは2004年に設立されました。米国カリフォルニア州サンノゼに本拠地を置き、世界各国の拠点からお客様をサポートしています。
A10ネットワークス株式会社はA10 Networksの日本子会社であり、お客様の意見や要望を積極的に取り入れ、革新的なアプリケーションネットワーキングソリューションをご提供することを使命としています。
詳しくはホームページをご覧ください。
www.a10networks.co.jp
Facebook：http://www.facebook.com/A10networksjapan

本資料は、アマゾン データ サービス ジャパン株式会社が運営する「AWS Solutions Architect ブログ」に掲載された記事を許諾を得て再構成したものです。Amazon Web Services、アマゾン ウェブ サービス、AWS、AWS Direct Connect、Amazon EC2、Amazon Machine Image、Amazon CloudWatchおよびPowered by Amazon Web Servicesロゴは、Amazon.com, Inc.またはその関連会社の商標です。

A10ネットワークス株式会社
〒105-0001
東京都港区虎ノ門4-3-20
神谷町MTビル16階
TEL : 03-5777-1995
FAX: 03-5777-1997
jinfo@a10networks.com
www.a10networks.co.jp

海外拠点
北米（A10 Networks本社）
sales@a10networks.com
ヨーロッパ
emea_sales@a10networks.com
南米
latam_sales@a10networks.com
中国
china_sales@a10networks.com
香港
HongKong@a10networks.com
台湾
taiwan@a10networks.com
韓国
korea@a10networks.com
南アジア
SouthAsia@a10networks.com
オーストラリア/ニュージーランド
anz_sales@a10networks.com

お客様のビジネスを強化するA10のアプリケーションサービスゲートウェイ、Thunderの詳細は、A10ネットワークスのWebサイト**www.a10networks.co.jp**をご覧になるか、A10の営業担当者にご連絡ください。

Part Number: A10-SB-90005-JA-01
June 2015

©2015 A10 Networks, Inc. All rights reserved. A10 Networks、A10ロゴ、A10 Lightning、A10 Thunder、aCloud、ACOS、ACOS Policy Engine、ACOS Synergy、Affinity、aFleX、aFlow、aGalaxy、aVCS、AX、aXAPI、IDaccess、IDsentrie、IP-to-ID、SoftAX、SSL Insight、Thunder、Thunder TPS、UASG、VirtualN、Virtual ChassisおよびvThunderは米国およびその他各国におけるA10 Networks, Inc. の商標または登録商標です。その他上記の全ての商品およびサービスの名称はそれら各社の商標です。その他の商標はそれぞれの所有者の資産です。A10 Networksは本書の誤りに関して責任を負いません。A10 Networksは、予告なく本書を変更、修正、譲渡、および改訂する権利を留保します。製品の仕様や機能は、変更する場合がございますので、ご注意ください。