Web三層モデル(1)

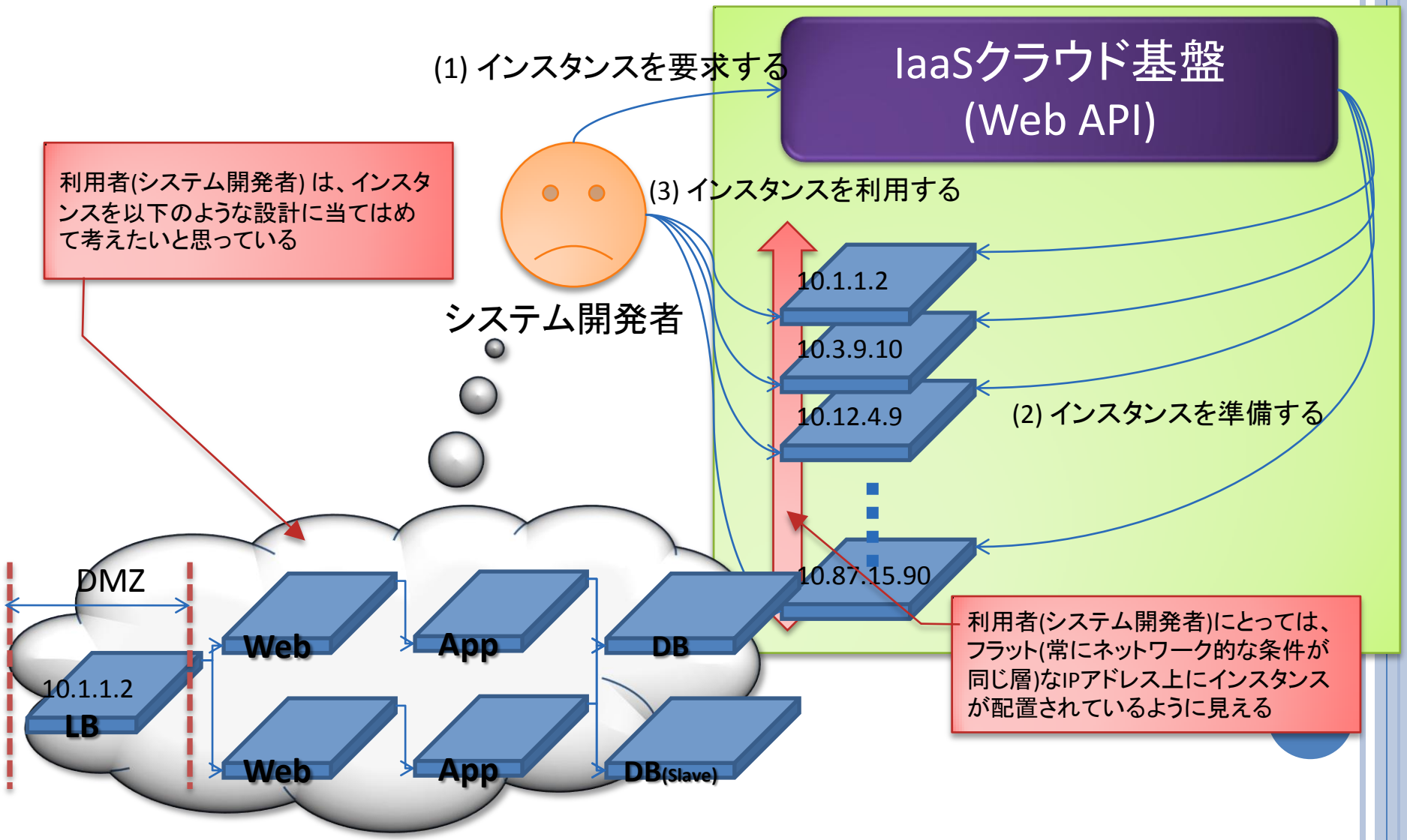
スケールアウトの実現

アジェンダ

1. ネットワークの考え方
    1. フラットなネットワーク構成
    2. セキュリティの仕組み
    3. 構成例
2. スケールアウトとシュリンクイン
    1. インスタンスの増減
    2. プロダクト間の依存関係
    3. サービスの開始と確認
    4. 非同期処理による効率化
3. 演習の準備
    1. スケールアウトをプログラミングに挑戦
    2. 実習
    3. 更なる効率化の議論

クラウド基盤のネットワーク構成と仕組み

フラットなネットワーク構成



Security Groupの考え方

- 起動するインスタンスに対し、ファイアウォールの設定を表現したグループ名を複数指定することで、それぞれのインスタンスにそのファイアウォールが備わる考え方
- 通信の制御は全てSecurity Groupの考え方に基づいて行われる
- スイッチやルータを仮想的にしようとした考え方ではない

EUCALYPTUSでの実装

- iptables + Tagged VLAN
  - L2レベルでTagged VLANを利用してセキュリティグループを実現する
  - Tagged VLANの名前空間(6bits=4096個)の制約がある
  - Cluster Controllerで実現するため、
    シングルポイントで集中的管理、同時反映が可能である

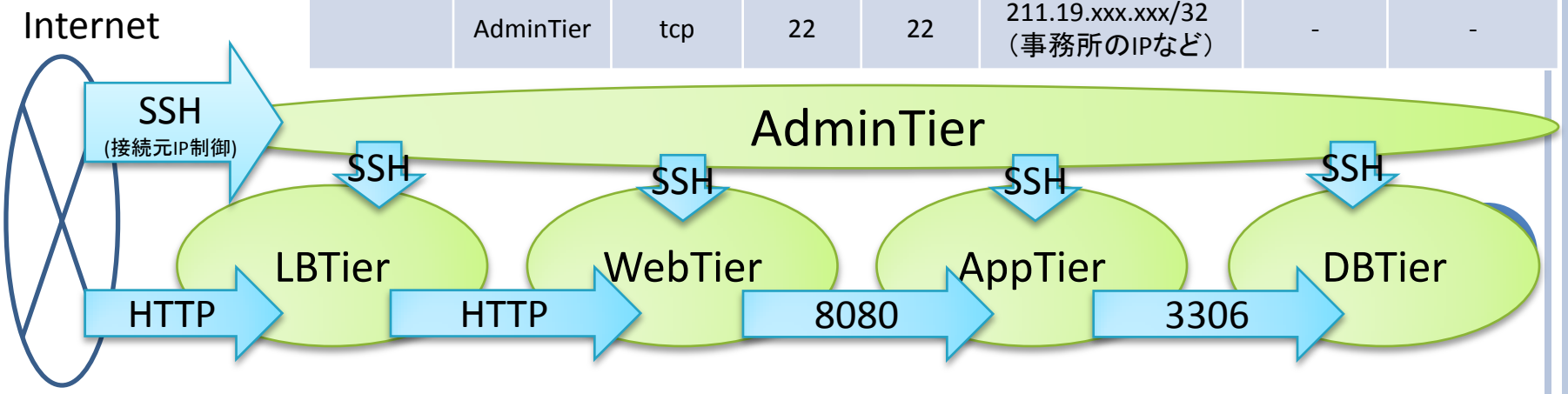
WAKAME-VDCでの実装

- ebtables + iptables
  - インスタンス間のパケットフィルタリングをebtablesやiptablesだけで実現する
  - セキュリティグループへの変更は、ハイパーバイザー層(HVA)でイベンチュアルに実行されていくため一貫性は保証されないが、分散しているのでスケーラブルに処理ができる

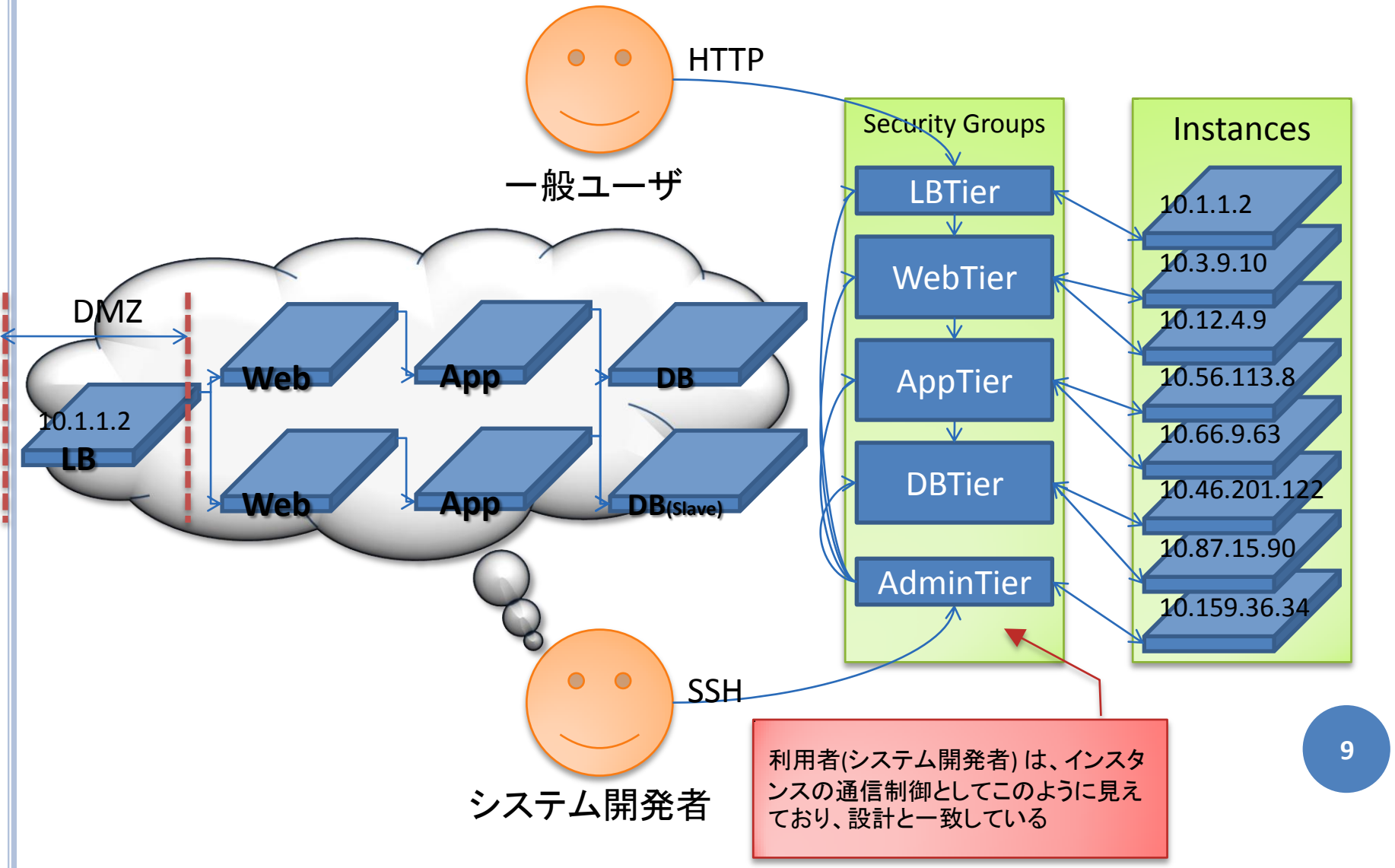
WEBシステムのSECURITY GROUP適用例

※ユーザIDは、
他のユーザのグループと
通信するために用いる

ユーザID	グループ名	Protocol	From Port / ICMP Type	To Port / ICMP Code	接続元許可 ((1)か(2)の一方が必須)		
					(1)CIDR	(2)他のグループ	
						ユーザID	グループ名
123456	LBTier	tcp	22	22	-	123456	AdminTier
		tcp	80	80	0.0.0.0/0 （万人に向ける）	-	-
	WebTier	tcp	22	22	-	123456	AdminTier
		tcp	80	80	-	123456	LBTier
	AppTier	tcp	22	22	-	123456	AdminTier
		tcp	8080	8080	-	123456	WebTier
	DBTier	tcp	22	22	-	123456	AdminTier
		tcp	3306	3306	-	123456	AppTier
	AdminTier	tcp	22	22	211.19.xxx.xxx/32 （事務所のIPなど）	-	-

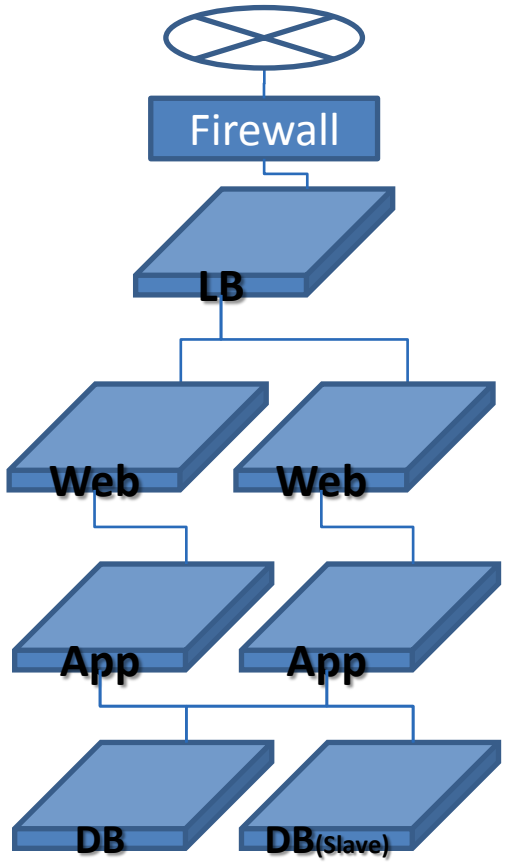


FIREWALLとして機能する



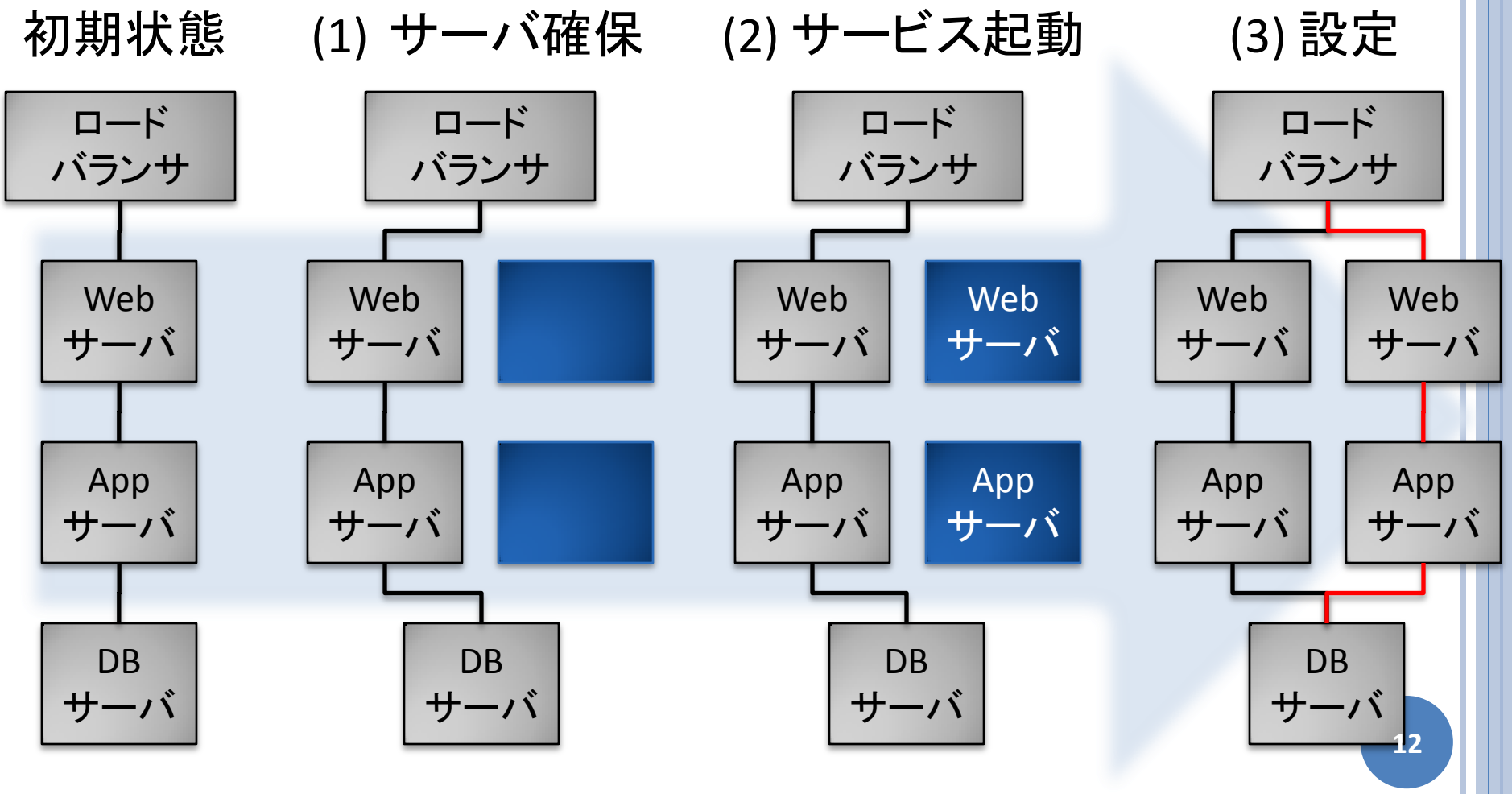
参考: ネットワークそのものの仮想化をしようという動き

- スイッチやルータを仮想化したものを制御に用いようとする動きはある
  - Project Crossbow
  - vyatta



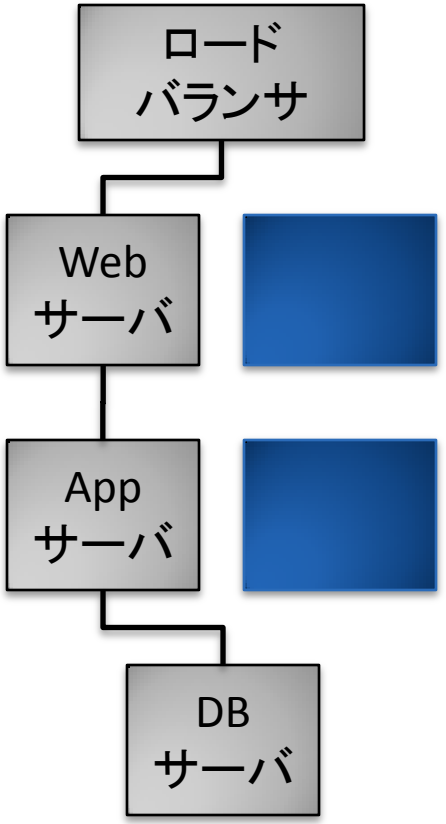
スケールアウトとシュリンクインの実現方法

スケールアウトの概要



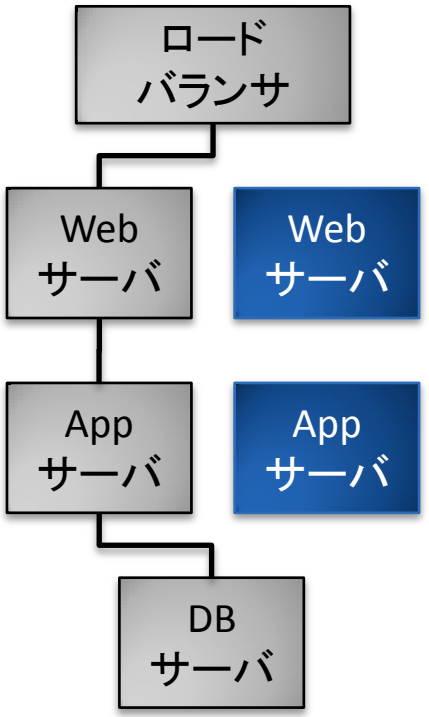
インスタンスを増減させる

- インスタンスを起動する
  - すぐには起動しない
  - PendingからRunningになるまで待つ必要がある



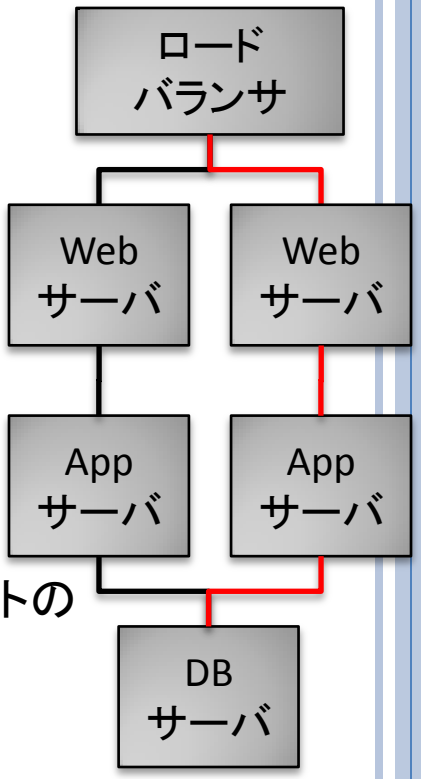
ブートと通信の確認

- インスタンスの設定を変更する
  - 疎通の確認
    - Running直後のインスタンスはブートが始まったばかり
    - SSHが通らないので、チェックする
  - SSHを実行する
    - リモートでコマンドを実行するために使用



プロダクト間の依存関係を解決する

- 全体の起動の場合
  - インターネットから見てネットワーク的に一番遠いものから順に起動するのが良い
  - 例えば全体の起動であれば下記の順が良い
    1. Database
    2. Web Server
    3. Load Balancer
  - 最近のプロダクトは、あまり気にしなくても良くはなってきている。
    - 接続を保持し続けるものが少なくなってきた
    - 切断があっても再接続を試みるのが一般的になった
- スケールアウトの場合
  - 必要であれば、ネットワーク的に関係する前後のプロダクトの設定変更を実施しなければならない
- 設定の反映方法
  - SSHでインスタンス内部のサービスの設定ファイルを変更
  - SSHでGraceful Restartを実施



Web APIを利用する

- インスタンスの起動
  - RunInstances命令で、インスタンスを起動する
  - DescribeInstances命令で、
    起動しているインスタンスの状態をチェックする
    - pengingからrunningになるまで
- インスタンスの終了
  - TerminateInstances命令で、インスタンスを終了する

Web API: HTTP(S)通信と認証アルゴリズム

- Web APIは、HTTPSを用いる
- ユーザの認証アルゴリズムは下記の通り
  - HmacSHA1で下記情報のダイジェスト値を算出する
    - Secret Access Key (ユーザ毎に発行される)
    - Web APIに必要なQueryパラメータ
      - Access Key ID
      - Web APIの情報(バージョン、メソッド名)
      - Web APIの引数
    - その他
  - ダイジェスト値をURLに含めてHTTPリクエストを発行する
- なお、戻り値はXMLになっている

参考: 認証アルゴリズム(RUBY) (1/2)

リスト1

```
require 'openssl'
require 'base64'

class AwsRequestParameter
  @@digest = OpenSSL::Digest::Digest.new("sha1")

  def self.amz_escape(param)
    param.to_s.gsub(/([^a-zA-Z0-9._~-]+)/n) do
      '%' + $1.unpack('H2' * $1.size).join('%').upcase
    end
  end

  # リクエストパラメータへの署名方法 v2
  def self.sign_request_v2(aws_secret_access_key,
                           service_hash,
                           http_verb,
                           host,
                           uri_path)
    # 必須パラメータの追加
    service_hash["Timestamp"] ||= Time.now.utc.strftime("%Y-%m-%dT%H:%M:%S.000Z") unless service_hash["Expires"]
    service_hash["SignatureVersion"] = '2'
    service_hash['SignatureMethod'] = 'HmacSHA1'

    # 署名対象データの正規化
    canonical_string = service_hash.keys.sort.map do |key|
      "#{amz_escape(key)}=#{amz_escape(service_hash[key])}"
    end.join('&')

    # 署名をする
    string_to_sign = "#{http_verb.to_s.upcase}¥n#{host.downcase}¥n#{uri_path}¥n#{canonical_string}"
    signature      = amz_escape(Base64.encode64(OpenSSL::HMAC.digest(@@digest, aws_secret_access_key, string_to_sign)).strip)

    # パラメータとして返す
    "#{canonical_string}&Signature=#{signature}"
  end

end
```

参考: 認証アルゴリズム(RUBY) (2/2)

リスト2 : リスト1の利用例

```
params = AwsRequestParameter.sign_request_v2(
  'AdBcCbDaE9F8G7H6I5J4K3L2M1NOOzPyQxRwSvTu', # Secret Access Keyを転記する
  {
    'AWSAccessKeyId' => 'A1B2C3D4F5G6H7I8J9K0', # Access Key IDを転記する
    'Version' => '2010-06-15',                  # Web API Version
    'Action' => 'RunInstances',                 # インスタンス起動

    'ImageId' => 'ami-84db39ed',                # マシンイメージのID
    'MinCount' => 1,                            # 最小個数
    'MaxCount' => 1,                            # 最大個数
  },
  'GET',                                        # HTTPメソッド
  'ec2.amazonaws.com',                          # 通信先ホスト名
  '/'                                           # URIのパス
)
puts "https://ec2.amazonaws.com/?"+params
```

リスト2の実行結果：認証アルゴリズムによって組み立てられたURLが表示される

```
https://ec2.amazonaws.com/?AWSAccessKeyId=A1B2C3D4F5G6H7I8J9K0&Action=RunInstances&ImageId=ami-84db39ed&MaxCount=1&MinCount=1&SignatureMethod=HmacSHA1&SignatureVersion=2&Timestamp=2010-08-10T12%3A18%3A03.000Z&Version=2010-06-15&Signature=VvU63m2zYFKKbxMyu3cXUpGLmhc%3D
```

```
<?xml version="1.0"?>
<RunInstancesResponse xmlns="http://ec2.amazonaws.com/doc/2010-06-15/">
    <requestId>3e16c4bc-b880-4634-98c1-bf276c112bbd</requestId>
    <reservationId>r-f07e069b</reservationId>
    <ownerId>123456789012</ownerId>
    <groupSet>
        <item>
            <groupId>default</groupId>

        </item>
    </groupSet>
    <instancesSet>
        <item>
            <instanceId>i-59ae2733</instanceId>
            <imageId>ami-84db39ed</imageId>
            <instanceState>
                <code>0</code>
                <name>pending</name>
            </instanceState>
            <privateDnsName/>
            <dnsName/>
            <reason/>
            <amiLaunchIndex>0</amiLaunchIndex>
            <productCodes/>
            <instanceType>m1.small</instanceType>

            <launchTime>2010-08-10T11:43:51.000Z</launchTime>
            <placement>
                <availabilityZone>us-east-1c</availabilityZone>
                <groupName/>
            </placement>
            <kernelId>aki-94c527fd</kernelId>
            <ramdiskId>ari-96c527ff</ramdiskId>

            <monitoring>
                <state>disabled</state>
            </monitoring>
            <stateReason>
                <code>pending</code>
                <message>pending</message>
            </stateReason>

            <rootDeviceType>ebs</rootDeviceType>
            <rootDeviceName>/dev/sda1</rootDeviceName>
            <blockDeviceMapping/>
            <virtualizationType>paravirtual</virtualizationType>
        </item>
    </instancesSet>
</RunInstancesResponse>
```

起動して来る
インスタンスのID

インスタンスの
現在の状態

参考:
URLにアクセスした結果

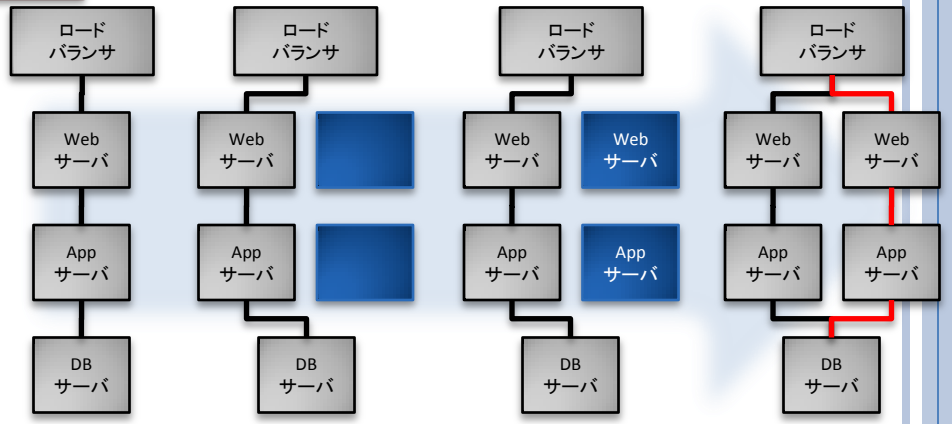
Web APIをWrapした言語別ライブラリ

- 提供されるもの
  - Web APIにアクセスするためのメソッド
  - 応答されたXMLを言語ごとのオブジェクトにマッピングして、使いやすく加工する
- 代表的なもの
  - Java
    - Typica
      - http://code.google.com/p/typica/
      - 今回の演習ではTypicaを用います
  - Ruby
    - RightAWS
      - http://rightaws.rubyforge.org/
  - Python
    - Boto
      - http://code.google.com/p/boto/

非同期処理

サーバ確保を100台同時に行うとどうなるか？

- 100台同時には起動して来ない
- 状態の異なるインスタンスが混在する
  - すぐに利用可能になるインスタンス
  - なかなか利用可能にならないインスタンス



100台全てが起動してくるのを待つのは非効率的。
起動完了した物から先の手続きに進めるなど、
「並列で処理できる部分は並列に実行する」と言う非同期処理の考え方が必要。

マルチプロセス・マルチスレッドプログラミングが重要になる。

WAKAME-FUELの事例

1. Ruby+AMQPの構成で動く
   オープンソースソフトウェア。
   - どなたでも無償である限り無料でご利用可能なもの

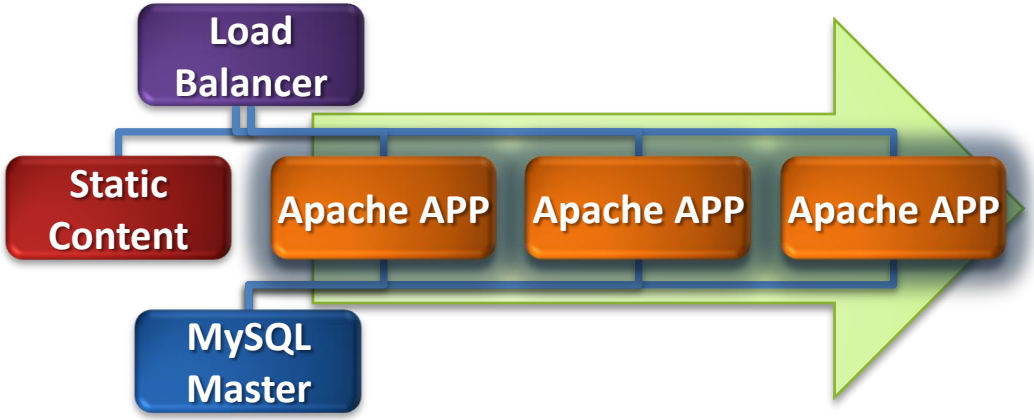
2. 管理すべきマシン全てにインストールし、
   手順を与えることでそれを自動的に実行する。
   - システム管理者の代わりに働く

3. 応用としてシステムの
   スケールアウトが可能になる。
   - IaaSクラウド基盤と組み合わせてご利用いただくと効果的

WAKAME-FUELはコマンド一行で スケールアウトするようになっている

```
# wakameadm propagate_service ¥
  Apache_APP 10
```

コマンド1行で、
後は見ているだけで
10台になる



なお、今回の演習ではJavaプログラムを実行するだけで
スケールアウトするように実装する。

演習：スケールアウトを実践してみる

演習の概要

- 【目的】
  - サンプルプログラム「掲示板」を正しく動作させながら、システムの規模を変化(スケールアウト)させる
- 【方法】
  - 「掲示板」の正しい動作を確認する
    【当演習参考資料1章】
  - スケールアウトのプログラムを作成する
    【当演習参考資料2章】
  - 実行して動作を確認する
    - スケールアウトしたか、分散処理がされているか
    - 動作は正しいか

演習

1. Apache2+Tomcatをスケールアウトさせるコードを書いてみよう
2. スケールアウトを実行してみよう
3. 動作の確認をしよう
4. 【議論】
   - より良い処理にするにはどのような実装が可能か

課題：議論で出たアイディアをプロトタイピングしてみよう

- 提出物
  - 動作するソースコード一式
- 例
  - 数を指定することで同時に並列でスケールアウトさせる
    - Web+Appの準備、設定までを複数スレッドで並列
    - 実行する