VM Ensembles

A unit or group of complementary parts that contribute to a single effect

Gary Kotton[1](VMware) and Gilad Zlotkin[2](Radware)

Introduction

This document introduces the concept of a VM ensemble or VM group into Nova. An ensemble will provide the tenant the ability to group together VMs that provide a certain service or part of the same application. It enables configuring scheduling policies per group. This will in turn allow for a more robust and resilient service. Specifically, it will allow a tenant to deploy a multi-VM application that is designed for VM fault tolerance in a way that application availability is actually resilient to physical host failure. It will also allow tenant to request that VMs will be placed in close proximity to each other to optimize performance.

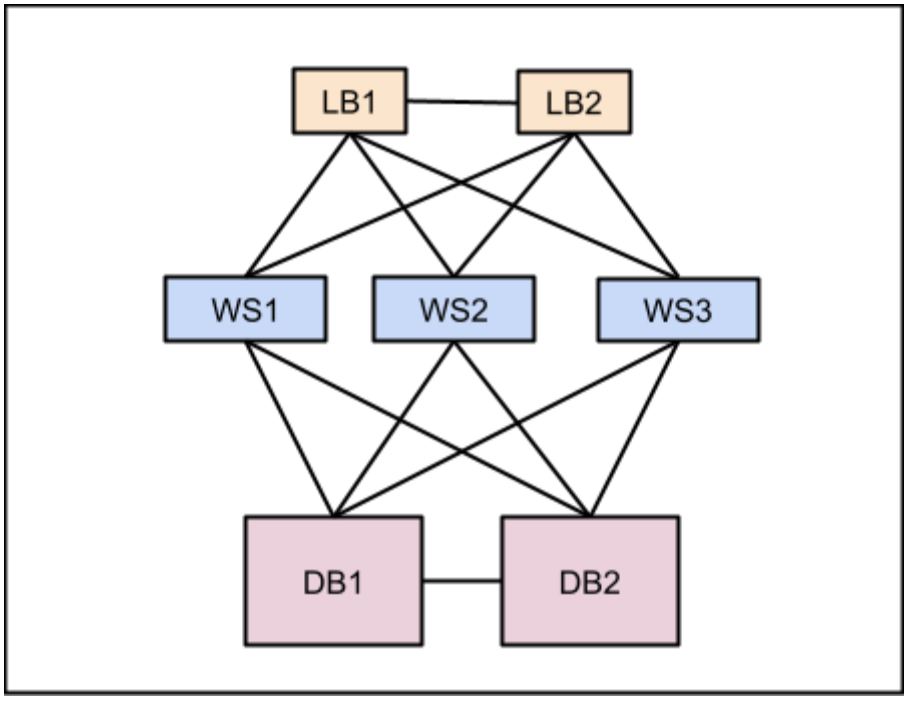
Let us consider the example below.

A tenant is deploying a WEB service. This service has the following components in a fault resilient layout:

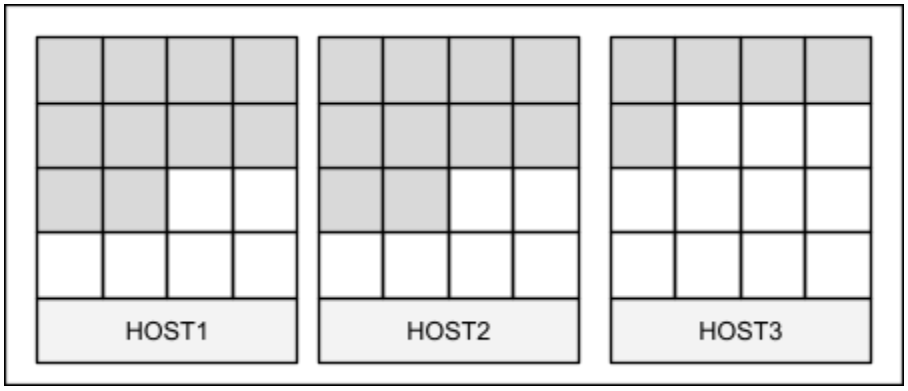
- Load balancers - these VM's will be used to distribute the load amongst the WEB servers. This provides fault resiliency for the WEB servers. (LB***i***). Each load balancer requires 1 capacity unit.
- WEB servers (WS***i***). Each web server requires 2 capacity units.
- Back end databases (DB***i***). Each database requires 4 capacity units.

[1] Gary Kotton is a Staff Engineer at VMware - gkotton@vmware.com

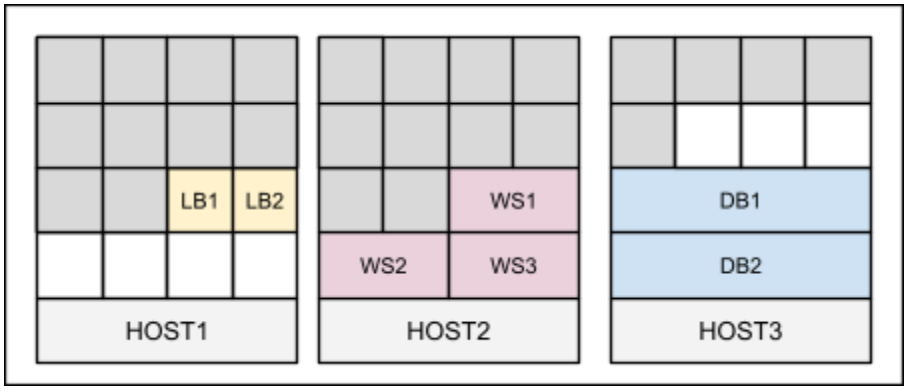
[2] Gilad Zlotkin is a VP Virtualization and Management at Radware - giladz@radware.com



The tenant is able to make use of host aggregates, zones or availability zones for different placement strategies for the VM's. Lets take for example a target host aggregate with 3 Hosts: HOST1 has 6 free compute units, HOST2 has 6 free compute units and HOST3 has 11 free compute units.



The placement of the VMs could be as follows:

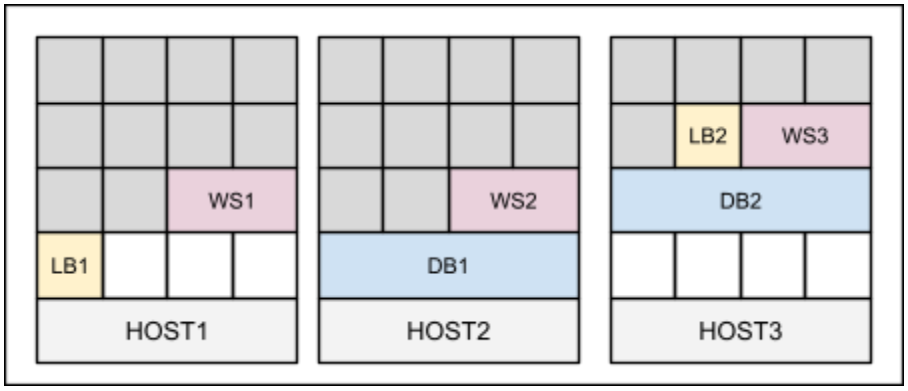


The problem is that if any of the hosts go down, then the tenant's service will no longer be up and accessible.

This is where the notion of the ensembles comes into play. The tenant will be able to define the placement strategies for the groups, for example:

- LB group = Anti Affinity
- WS group = Anti Affinity
- DB group = Anti Affinity

In the example above the placements of the VM's will now be:



This means that the service is actually fault tolerant as designed, i.e. if one of the HOSTs goes down then the service will still be up and running. The ensembles will enable a service to be tolerant to host failures in an availability zone or a host aggregate.

Host Fault Tolerance versus Site Failure Recovery

Host server failure tolerance is different from site failure recovery. Availability zone is a mechanism for managing site failure recovery, i.e. redirecting application clients from the failed primary instance of the application to another "stand by" application instance that is deployed in a different availability zone. This typically requires re-establishing of all client sessions. From an

application client perspective, a site failure recovery (also called “Disaster Recovery”) event will be experienced as a (potentially short) downtime that will require re-establishing the session. The complexity of re-establishing an application client session differs from one application to the next. For example, an online navigation application may require the user to resubmit the target destination and wait for the service to re-compute a route. This may be a challenging when the application user is driving with “real time” directions. For other applications, it can be, as simple as, reloading a web-page.

Host failure resiliency, on the other hand, is designed to minimize or even completely avoid any application client impact in the case of faults. In the example above, the two load balancing instances are continuously synchronizing states with each other, similarly with the two database instances. Such that, both load balancing and database service layers support stateful failover, maintaining all sessions intact. Losing an application server instance (WSi) may result in losing all sessions processed by it only if session states are not stored in the database. Otherwise, the load balancer will be able to redirect the session to one of the surviving application server instances that will be able to call the full session state out from the database, such that the session will continue intact.

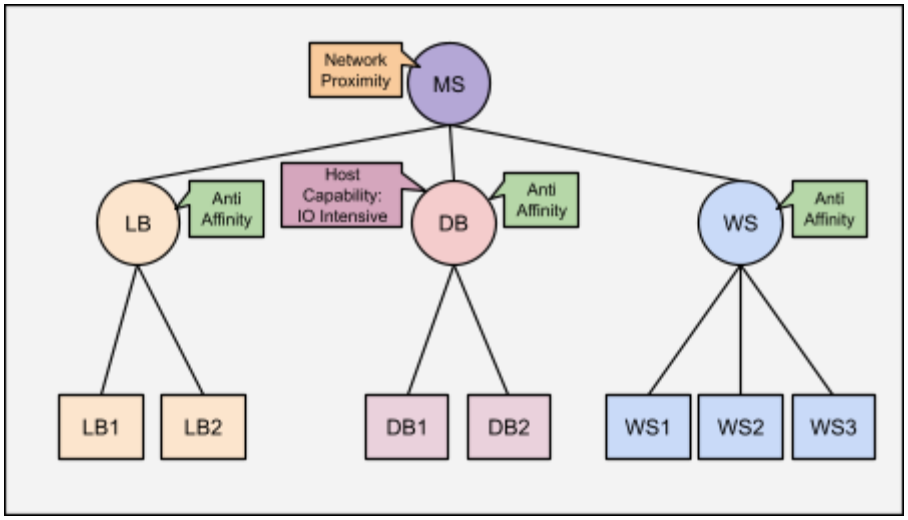
Host failure is a more common event in the cloud than an availability-zone/site failure. Mission critical application deployments in the cloud can be designed to be resilient for either or both types of failures.

VM Group Scheduling Policies

As we have described above Anti-Affinity is an important VM group scheduling policy to ensure host fault resiliency. There are other VM group scheduling policies to be focused on performance.

- Availability:
    - **Anti-Affinity**: Group member VMs should be placed on different Hosts to ensure that any single host failure will affect at most one of the VM group members to support fault tolerance application architectures.
- Performance:
    - **Network-Proximity**: Group member VMs should be placed as close as possible to each other on the network (in terms of hop count and latency) to optimize application performance.

Going back to the example above, where we have 3 different sub-groups: LB, WS and DB groups. As we saw above each group requires an Anti-Affinity scheduling policy to ensure that a single host fault will kill at most one member VM in each group ensuring fault tolerance.



The entire application (the MS supergroup) requires a Network-Proximity scheduling policy to ensure best performance connectivity between the load balancers and the application servers, between applications servers and the database servers, as well as, between the two database servers themselves and the between the two load balancers themselves for state synchronization.

Implementing Anti-Affinity

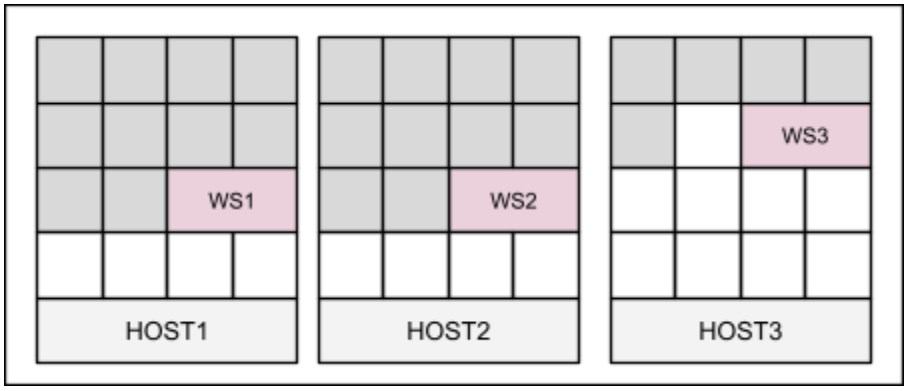
The current Nova scheduler is admitting one VM. There is an option for passing group scheduler hints, which make use of a scheduling filter *GroupAntiAffinityFilter*[3]. In addition to this one can deploy same VM N times. The scheduler hint *group* will indicate that the *GroupAntiAffinityFilter* will be used as part of the scheduler filtering. VM ensembles will need to be scheduled as a group. Otherwise, if we try to schedule them one VM at a time and even one subgroup at the time, then we are risking a false negative admission decision.

To demonstrated this, let us schedule VMs from the group one subgroup at the time.

Starting by scheduling the 3 WS VMs, the Nova scheduler will take the anti-affinity policy into account and will place them as follows:

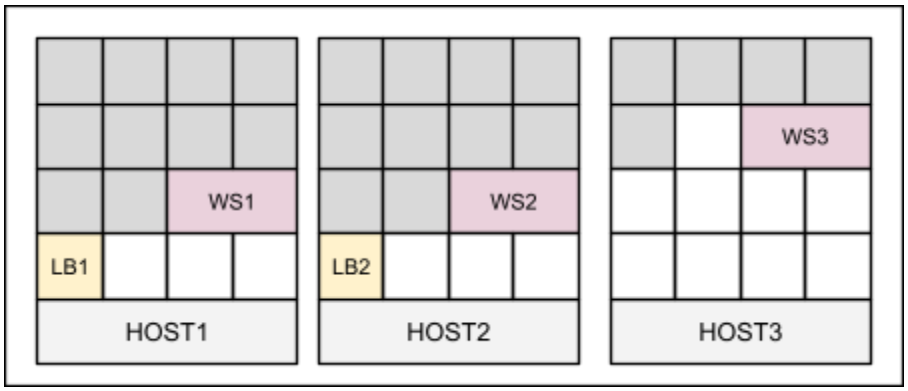
nova boot --hint group=WS[:anti-affinity] --image ws.img --flavor 2 --num 3 WS1

[3] The support was added in https://review.openstack.org/#/c/21070/
The current default support is anti-affinity.



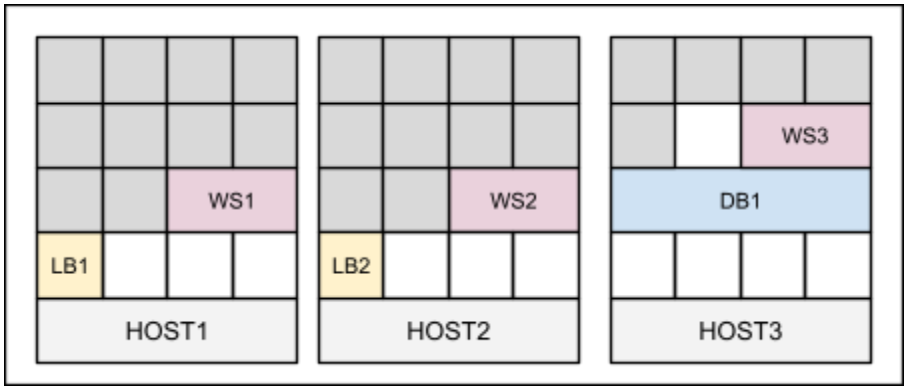
Scheduling the two LB VMs next, will get the Nova Scheduler to do the following placements:

nova boot --hint group=LB[:anti-affinity]--image lb.img --flavor 1 --num 2 LB1



Trying to schedule the DB1 VM next will succeed:

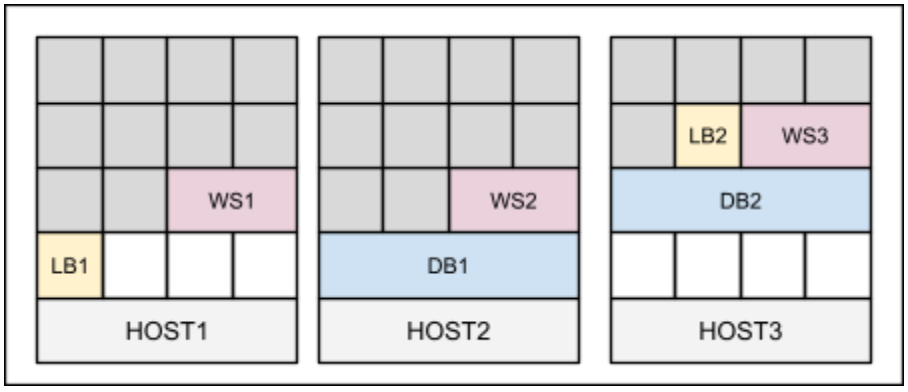
nova boot --hint group=DB[:anti-affinity] --image db.img --flavor 3 --num 2 DB1



However, DB2 cannot be admitted, as the only HOST3 has 4 free slots. But, that would violate the anti-affinity constraints with DB1.

Therefore, the above call will fail and the Nova scheduler will reject admitting the DB VMs, which practically means rejecting the admission of the entire multi-VM application.

This would be a false negative admission decision, as the application can be admitted as follows:

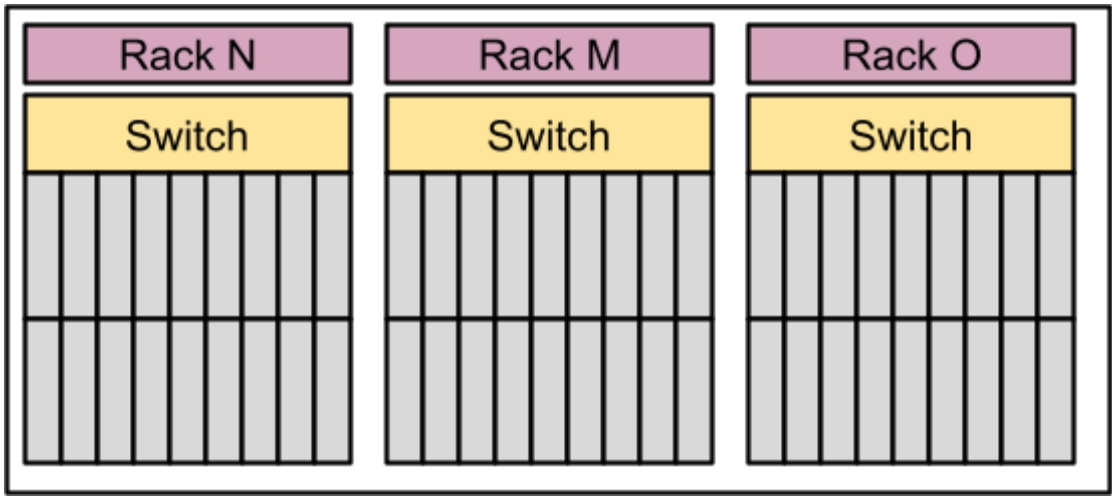


As the cloud compute capacity allocation percentage is higher, more likely we will suffer from false admission control decisions. Unless, we schedule the entire VM ensemble as a group. Cloud service providers are striving to maximize compute capacity percentage allocated to cloud tenants. For example, AWS is offering a special EC2 Spot Instances (http://aws.amazon.com/ec2/spot-instances/) to utilize any available capacity unit, such that capacity allocation will be maximized. Please note that even when capacity allocation on a given host is maximized, it does not mean that CPU utilization is maximized, since not all VMs on that host are likely to utilize their vCPU quota at the maximum level at the same time. Therefore, it is safe to drive capacity allocation percentage to the 90%-100% level.

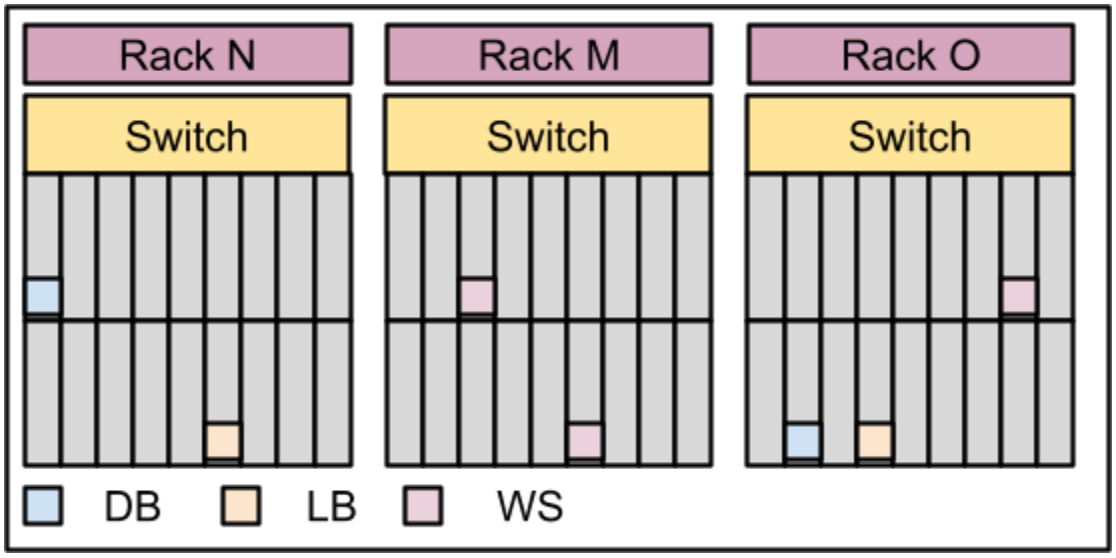
Conclusion: The Nova scheduler will need to schedule VM ensembles as a group. The next version of the ensemble API will enable the user to send an entire group for scheduling. In addition to this it will include an option to perform a rollback in the event that the group scheduling does not succeed.

Implementing Network-Proximity

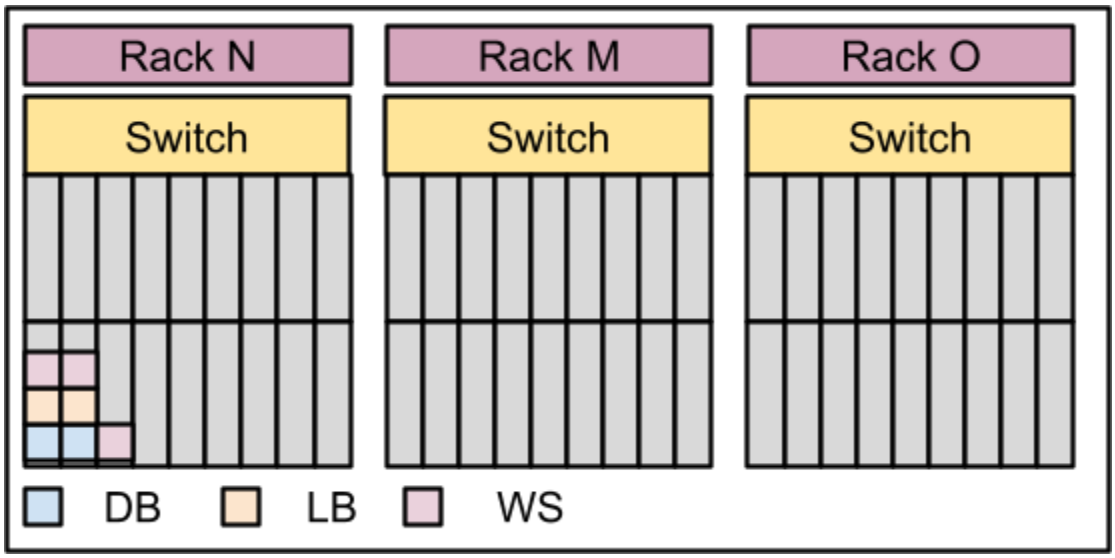
The current Nova scheduler has a filter *SimpleCIDRAffinityFilter*. This filter looks at the subnet mask and investigates if the network address of the current host is in the same subnetwork as it was defined in the request. If the hosts in a rack are not on the same subnet then the placement strategy when using the aforementioned filter can be suboptimal. For example if we a number of racks with blade servers:



The **Anti-Affinity** placement could lead to the following:



Where ideally the best placement would be as follows:



In order to be able to perform the above we will need a scheduling filter that would take the proximity of the hosts, more specifically the latency and the number of hops between the hosts into account.

Interfaces

Nova Ensemble Support

An API will be added to Nova to enable the user to pass a ensemble to be booted. This will be achieved by adding in a extension call *Ensemble_create* similar to *Multiple_create*. This will enable the tenant to pass in the ensemble information. The API will enable the user to pass the following information through to the scheduler. This will be a dictionary (format defined below). The Nova CLI will need to be updated to enable the user to create a Ensemble.

Create Ensemble

Verb	URI	Description
POST	/ensembles	Creates a VM ensemble

Normal Response Code(s): 202

Error Response Code(s): computeFault (400, 500, …), serviceUnavailable (503), unauthorized (401), forbidden (403), badRequest (400), badMethod (405), overLimit (413), itemNotFound (404), badMediaType (415), serverCapacityUnavailable (503)
This operation asynchronously provisions a VM ensemble.

The following table describes the required and optional attributes that you can specify in the request body:

Name	Description	Required
ensemble	A unique name/ID for the ensemble.	Yes
policy	The scheduling policy for the ensemble. This is an array of types. The possible values are: anti-affinity and network-proximity. The default is anti-affinity. Examples: policy: [‘anti-affinity’] policy: [‘anti-affinity’, network-proximity] policy: [‘network-proximity’]	No
rollback	Perform a rollback if the ensemble deployment fails. The default value is True. Example: rollback: False rollback: True	No
sub-groups	A list of subgroups. This can be one of more sub-groups. The table below will contain the sub-group fields.	Yes

The following table describes the required and optional attributes that you can specify in the request body of a sub-group:

Name	Description	Required
name	A unique name/ID for the ensemble.	Yes.
policy	The scheduling policy for the subgroup. This is an array of types. If this is not defined then it is inherited from the parent.	No

server	This is the current server details that exist for the creation of an instance	Yes

Example:

The example below will demonstrate the usage of the API for the service discussed above. More specifically it will be deploying a ensemble, called MS, consisting of:

- 2 load balancers, called LBG, using image lb.img
- 3 web servers, called WSG, using image ws.img
- 2 database servers, called DBG, using image db.img

```
{'ensemble': 'MS',
 'policy': ['network-proximity'],
 'sub-groups': [
       {'name': 'LBG',
        'policy': ['anti-affinity'],
        'server': {
              'name': 'LB'
              'imageRef': 'http://ensemble.example.org/images/lb.img',
              'flavorRef': '52415800-8b69-11e0-9b19-734f6f006e54',
              'min_count': 2,
              …}
       },
       {'name': 'DBG',
        'policy': ['anti-affinity'],
        'server': {
              'name': 'DB'
              'imageRef': 'http://ensemble.example.org/images/db.img',
              'flavorRef': '52415800-8b69-22e0-9b19-734f6f006e33',
              'min_count': 2,
              …}
       },
       {'name': 'WSG',
        'policy': ['anti-affinity'],
        'server': {
              'name': 'WS'
              'imageRef': 'http://ensemble.example.org/images/ws.img',
              'flavorRef': '52415800-8b69-44e0-9b19-734f6f006e44',
              'min_count': 3,
              …}
       }]
}
```

The command will return a list of the servers created.

Network Proximity

An API will be added to Quantum to provide the “network proximity” between hosts or compute nodes. This will be achieved by adding in a extension called *NetworkProximity*. The API will enable the user to get the number of hops and latency between two hosts. This will be a

dictionary (format defined below). A Quantum CLI command will also be provided to get the details.

Network Proximity get

Verb	URI	Description
GET	/networkproximity	Gets the network proximity between two hosts

Normal Response Code(s): 202

Error Response Code(s): computeFault (400, 500, …), serviceUnavailable (503), unauthorized (401), forbidden (403), badRequest (400), badMethod (405), overLimit (413), itemNotFound (404), badMediaType (415), serverCapacityUnavailable (503)

This operation returns the proximity between two hosts.

The following table describes the required and optional attributes that you can specify in the request body:

Name	Description	Required
host1	This is the host ID of the first host.	Yes
host2	This is the host ID of the second host.	Yes
network_id	The ID of the Quantum network	Yes

Example:

The example below will demonstrate the usage of the API for the service discussed above. More specifically it will be get the proximity between A and B on network ID X.

```
{‘host1’: ‘ID1’,
 ‘host2’: ‘ID2’,
 ‘network_id’: ‘X’
}
```

The command will return a dictionary that has:

{‘number_of_hops’: <number of hops between A and B>,

‘latency’: <latency in milliseconds between A and B>}