목차

Octavia LB 구현 및 분석 3
Octavia 3
기본 객체 개념 3
기본 사용 과정 4
소프트웨어 아키텍처 11
서비스 프로세스 목록 12
코드 구조 12
LoadBalaner 프로세스 분석 14
network_tasks.AllocateVIP 21
network_tasks.PlugVIP 22
Amphora 24
Amphora Agent 25
AmphoraAPIClient 26
AmphoraePostVIPPlug 26
Keepalived 서비스 프로세스 시작 28
리스너 생성 프로세스 분석 31
haproxy 서비스 프로세스 시작 32
풀 생성 프로세스 분석 34
멤버 프로세스 분석 만들기 37
CalculateDelta 38
HandleNetworkDeltas 39
AmphoraePostNetworkPlug 39
ListenersUpdate 41
L7policy, L7rule 및 Health Monitor의 생성 프로세스 분석 42
예시 1. 기본 풀로 전송 44
예시 2. 공유 풀로 전송 45
Amphora의 보안 통신 구현 47
자체 CA를 통해 구현된 SSL 통신 47
Amphora Agent 인증서 로딩 49
AmphoraAPIClient가 인증서 요청을 전송 51
Amphora의 장애 조치(페일오버) 구현 52
Health Manager 52
Amphora의 상태 모니터링 52
failover(장애 조치) 57
장애 조치 테스트 61
Neutron-lbaas vs. LBaaS v2 API vs. Octavia vs. Octavia v2 API 64
참조링크 64

Octavia LB 구현 및 분석

Octavia

Octavia는 OpenStack과 함께 작동하도록 설계된 오픈 소스, 운영자 규모 로드 밸런싱 솔루션입니다.

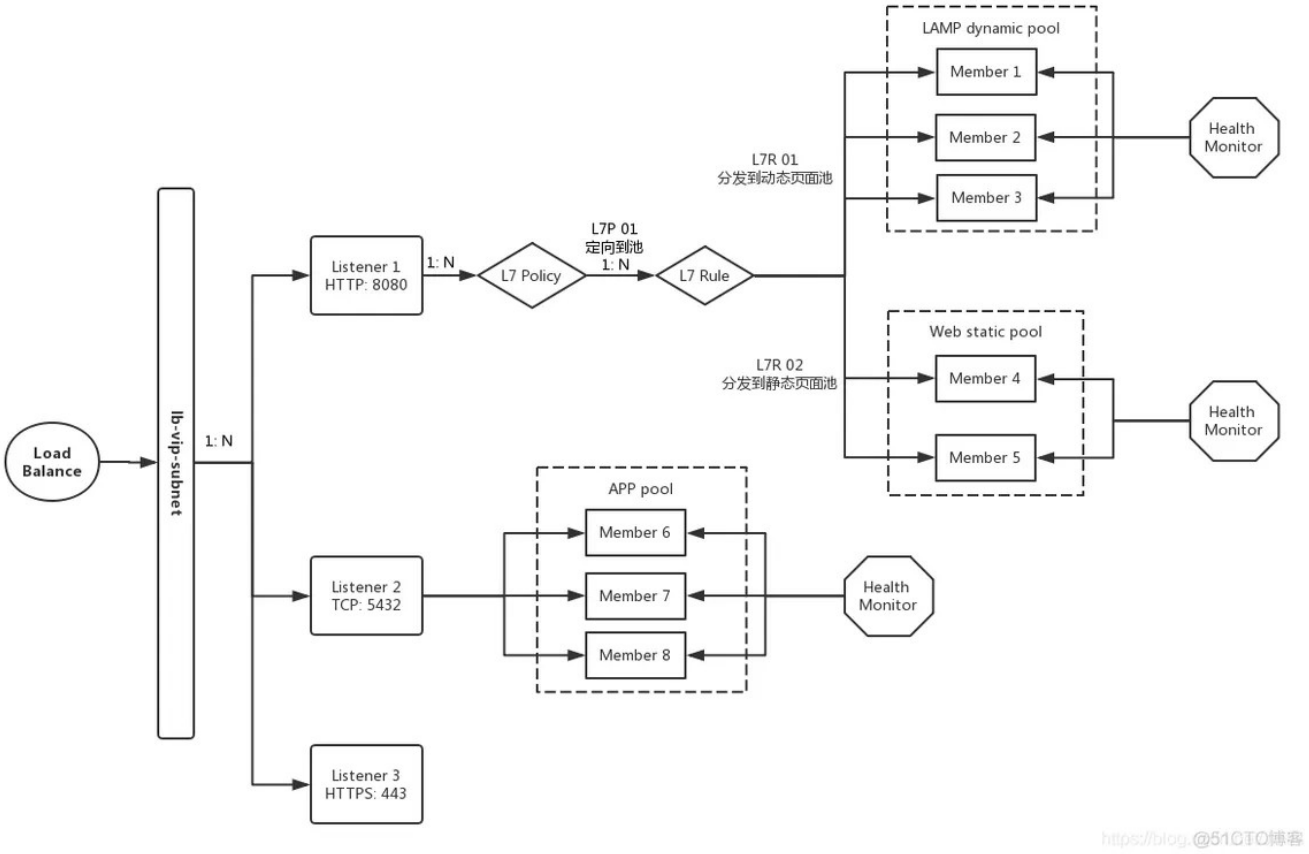
Pike 이후 OpenStack은 서비스로서의 로드 밸런싱을 위한 기본 솔루션으로 neutron-lbaas Extension 대신 Octavia를 사용할 것을 권장하고 Queens에서 neutron-lbaas를 더 이상 사용되지 않는 것으로 표시합니다. *Neutron-lbaas is now deprecated.*

커뮤니티에서 Octavia를 추천하는 데에는 neutron-lbaas가 남긴 역사적 문제를 해결하고 독립적이고 안정적인 API(Neutron/LBaaS/Deprecation)를 외부 세계에 제공할 수 있는 여러 가지 이유가 있습니다. 간단히 말해서, 커뮤니티는 neutron-lbaas가 Neutron의 프로젝트 관리를 장기화하고 LBaaS가 독립적인 프로젝트로 개발되어야 한다고 믿고 있으며 실제로 그렇습니다.

이 기사는 Rocky를 기반으로 OpenStack LBaaS로서의 Octavia의 추상 설계, 개발 설계 및 코드 구현을 기록하고 분석하며 Octavia에 대한 커뮤니티 개발자의 신뢰를 느낍니다.

기본 객체 개념

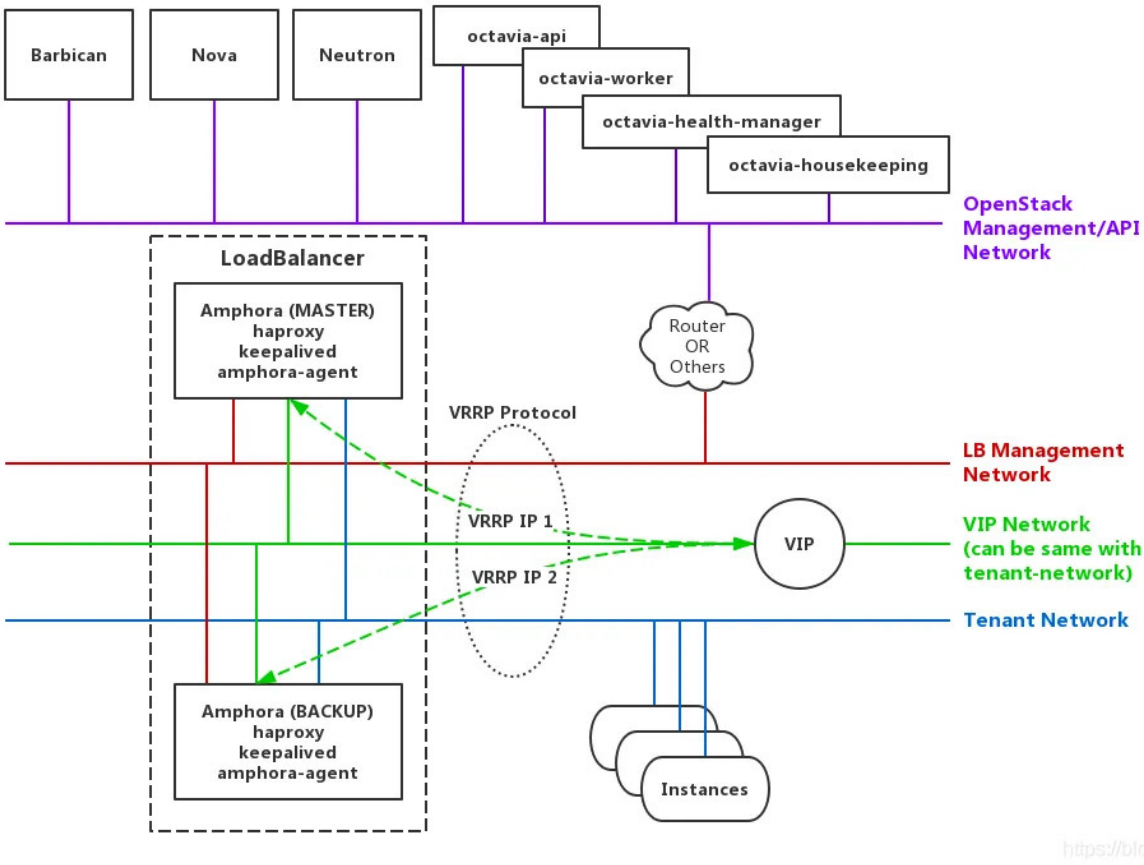
- LBaaS : OpenStack 플랫폼의 경우 LB(로드 밸런싱)가 사용자에게 서비스로 제공되며, 사용자는 필요에 따라 언제든지 구성 가능한 비즈니스 로드 밸런싱 솔루션을 얻을 수 있습니다. 이를 서비스형 로드 밸런싱이라고 합니다.
- loadbalancer : 로드 밸런싱 서비스의 루트 객체로, 이를 기반으로 로드 밸런싱에 대한 사용자 정의, 구성 및 동작이 이루어집니다.
- VIP : 로드 밸런서와 연결된 IP 주소입니다. 각 로드 밸런서에는 백엔드 비즈니스 클러스터에 대한 표준 외부 액세스 입구 역할을 하는 VIP가 하나 이상 있습니다.
- 리스너 : 로드밸런서의 하위 리스너로, VIP에 대한 외부 접속의 리스닝 유형(예: 프로토콜, 포트)을 구성할 수 있습니다.
- Pool : 백엔드의 실제 비즈니스 클라우드 호스트 클러스터 도메인. 일반적으로 사용자는 클라우드 호스트의 비즈니스 유형에 따라 구분합니다.
- 구성원(Member) : Pool에 종속된 비즈니스 클라우드 호스트로서 기존 로드 밸런싱 시스템의 Real Server에 해당합니다.
- Health Monitor : Pool에 연결되어 Pool 내 멤버에 대한 Health Check를 주기적으로 수행합니다.
- L7 정책 : 패킷 전달 작업을 설명하는 레이어 7 전달 정책(예: 풀로 전달, URL로 전달, 전달 거부)
- L7 규칙 : L7 정책에 종속된 레이어 7 전달 규칙은 데이터 패킷 전달을 위한 일치 도메인을 설명합니다(예: 풀의 웹 서버로 시작하는 모든 구성원에게 전달됨).



위 그림은 이러한 개념과 개인과 전체 간의 관계를 이해하는 데 도움이 되는 간단한 활성 및 정적 페이지 분리 로드 밸런싱 애플리케이션 아키텍처입니다.

이 시점에서 우리는 다음과 같은 질문을 하려고 합니다. 왜 이러한 객체를 추상화하는가?

기본 사용 과정



사용의 관점에서 Octavia가 어떤 모습인지 계속해서 이해하세요.

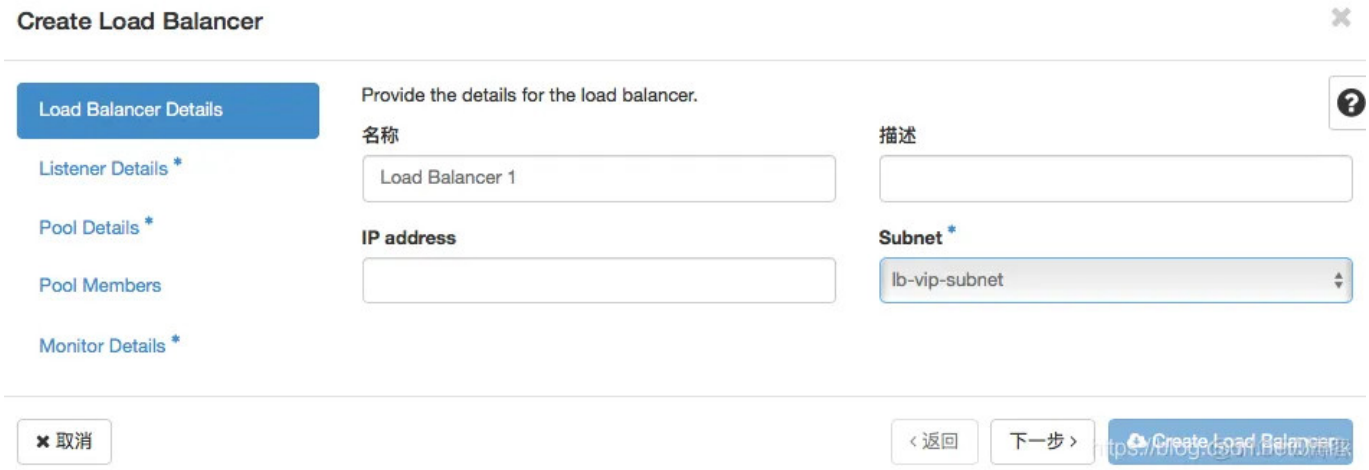
위 그림은 다음을 포함하는 **표준 Octavia 네트워크 아키텍처**입니다.

- `Amphora(e)` : 엔터티는 로드 밸런서의 캐리어 역할을 하며 Octavia의 기본 로드 밸런서 공급자이기도 한 클라우드 호스트입니다.
- `lb-mgmt-net` : OpenStack Management/API Network에 연결된 네트워크로 프로젝트 관리자에게 공개되며 동쪽은 Amphora 인스턴스에 연결되고 서쪽은 Octavia 서비스 프로세스에 연결됩니다.
- `tenant-net` : 비즈니스 클라우드 호스트가 위치한 네트워크
- `vip-net` : VIP 주소 풀을 제공하는 네트워크

참고: vip-net과 테넌트-넷은 동일한 네트워크일 수 있지만 프로덕션 환경에서는 보다 표적화된 방식으로 보안 정책을 적용하고 네트워크 보안 격리 도메인을 서로 다른 수준에서 나누기 위해 분리하는 것이 좋습니다.



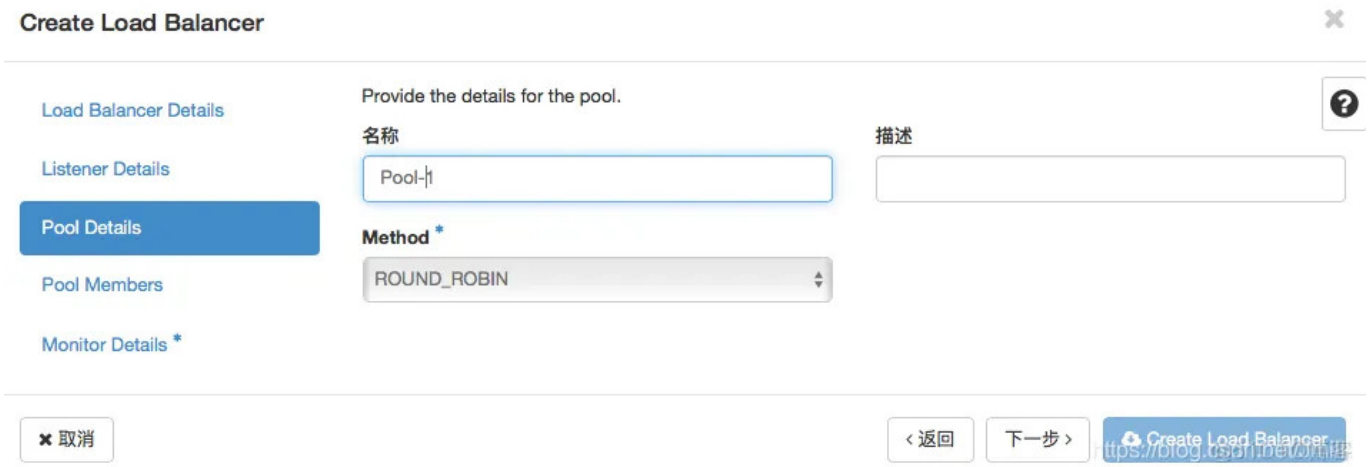
1단계. 로드밸런서의 VIP를 설정합니다. VIP는 직접 지정하거나 DHCP를 통해 할당할 수 있습니다.



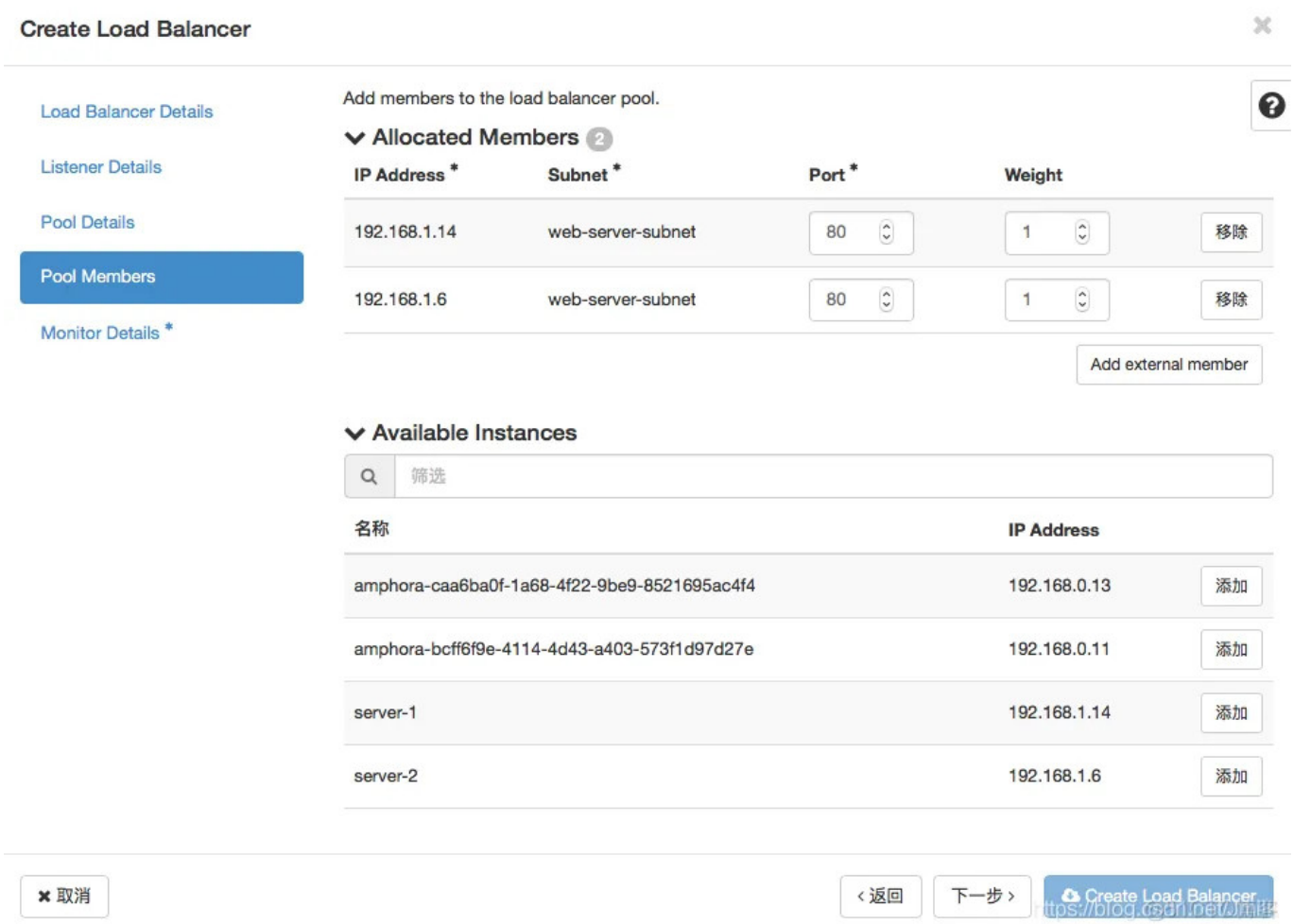
2단계. 리스너가 수신할 프로토콜과 포트를 설정합니다. 외부 액세스를 모니터링합니다 `http://<VIP>:8080/`.



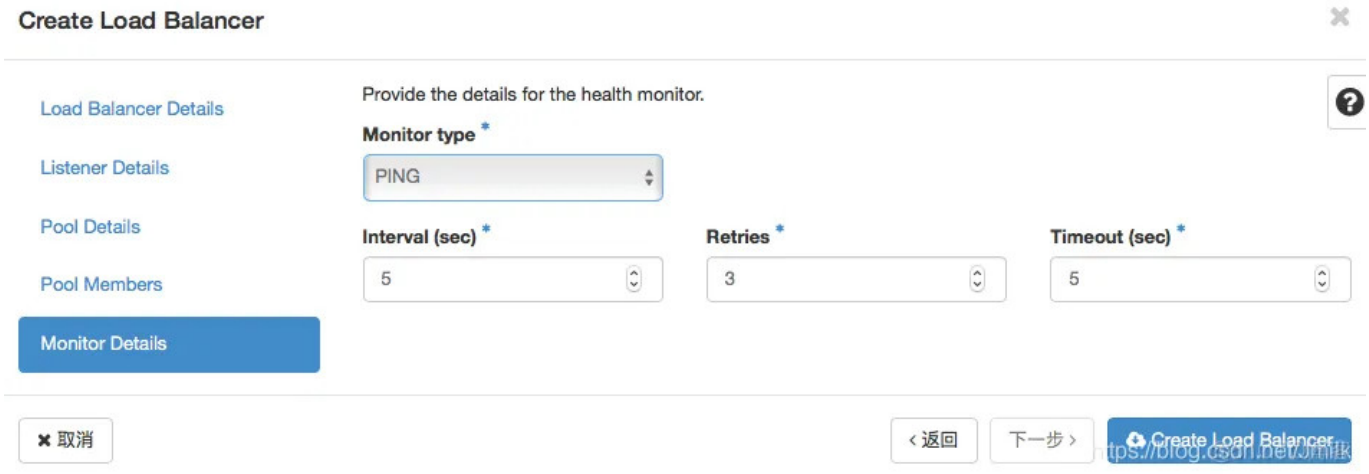
3단계. 풀의 로드 밸런싱 알고리즘을 설정합니다. 여기서는 RR 폴링 분포 알고리즘이 선택됩니다.



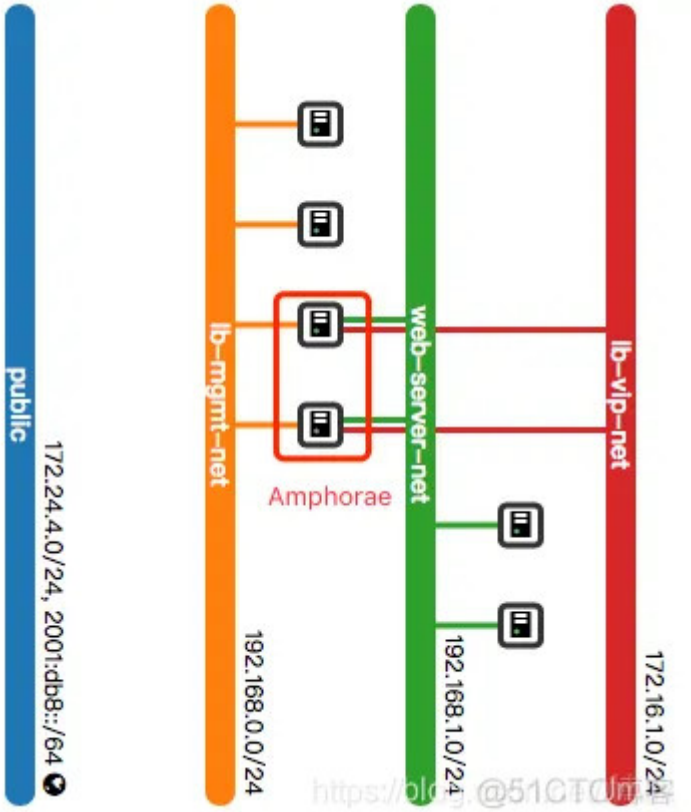
4단계. 풀의 구성원을 설정합니다. 멤버를 설정하려면 포트와 가중치를 지정해야 합니다. 전자는 데이터 전달을 허용하는 소켓을 나타내고 후자는 배포 우선 순위를 나타냅니다.



5단계. 상태 모니터의 상태 확인 규칙을 설정합니다. 구성원의 PING이 다른 경우 결함이 있는 것으로 표시되고 더 이상 배포가 허용되지 않습니다.

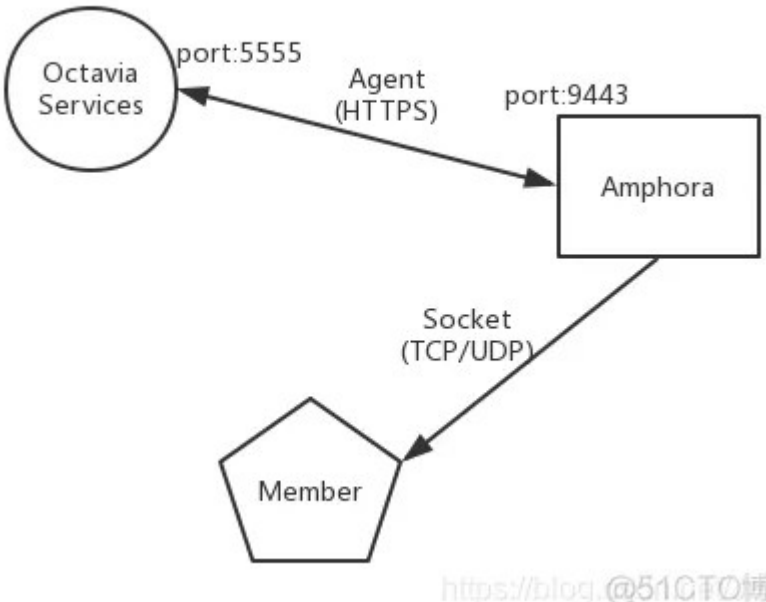


현재 네트워크 토폴로지 변경 사항은 다음과 같습니다. Amphorae가 포트 마운팅을 사용하여 세 가지 다른 네트워크에서 VIP, Member 및 Octava 서비스 프로세스를 연결하는 것을 볼 수 있습니다. 이름.



이제 Octavia Amphora Provider의 디자인 아이디어를 간략하게 검토해 보겠습니다.

- Amphora는 로드 밸런서 소프트웨어(HAProxy) 및 고가용성 지원(Keepalived)을 실행하는 캐리어 역할을 하며 에이전트를 통해 Octavia 서비스 프로세스와 통신합니다.
- Octavia 서비스 프로세스는 사용자의 로드 밸런서 및 VIP 구성 매개변수를 수신하고 에이전트를 통해 haproxy 및 keepalived 구성 파일을 동적으로 수정합니다.
- Member가 속한 Subnet을 Amphora에 연결하면 Amphora는 Member Socket(IP, Port)을 통해 요청 패킷을 배포합니다.



여기에는 Octavia와 관련된 이미지 및 보안 그룹의 내용을 추가하겠습니다. Amphora 인스턴스는 특정

이미지를 사용하여 시작됩니다. Octavia는 centos 및 ubuntu 운영 체제를 지원하는 특수 이미지 생성 스크립트를 제공합니다. 그러나 프로덕션 환경에서 로그인하려면 키 쌍을 사용하는 것이 좋습니다. 보안 그룹의 경우, Amphora의 보안 그룹은 적어도 두 가지 ingress 규칙(UDP/5555 및 egress:TCP/9443)을 충족해야 함을 위 그림에서 볼 수 있습니다.

앰포라 이미지를 사용하는 단계 :

1단계. 앰포라 이미지 업로드

```
$ /opt/rocky/octavia/diskimage-create/diskimage-create.sh -i ubuntu

$ openstack image create amphora-x64-haproxy \
  --public \
  --container-format=bare \
  --disk-format qcow2 \
  --file /opt/rocky/octavia/diskimage-create/amphora-x64-haproxy.qcow2 \
  --tag amphora
```

2단계. 앰포라 이미지 구성 앰포라 이미지를 업로드한 후 용도를 지정 하도록 구성해야 합니다.
`[controller_worker] amp_image_owner_id, amp_image_tag`

```
[controller_worker]
amp_image_owner_id = 9e4fe13a6d7645269dc69579c027fde4
amp_image_tag = amphora
...
```

amphora 보안 그룹을 사용하는 단계 :

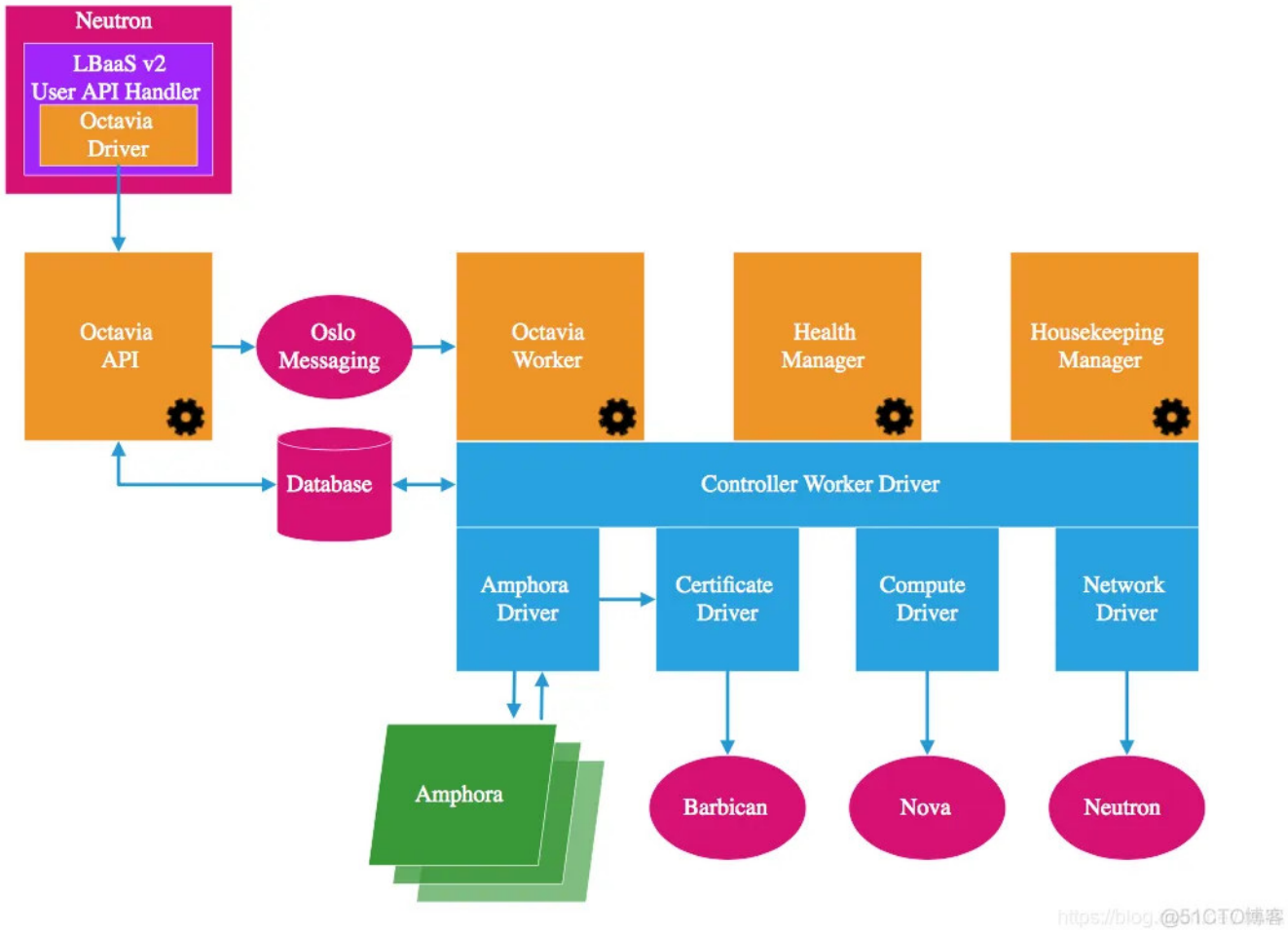
1단계. amphora에서 사용하는 보안 그룹 생성

```
$ openstack security group create amphora-sec-grp --project <admin project
id>
$ openstack security group rule create --remote-ip "0.0.0.0/0" --dst-port
9443 --protocol tcp --ingress --ethertype IPv4 --project <admin project id>
amphora-sec-grp
$ openstack security group rule create --remote-ip "0.0.0.0/0" --dst-port
5555 --protocol udp --egress --ethertype IPv4 --project <admin project id>
amphora-sec-grp
```

2단계. amphora 보안 그룹 구성

```
[controller_worker]
amp_secgroup_list = <amphora-sec-grp id>
...
```

소프트웨어 아키텍처



(참고: 사진은 Octavia 공식 문서에서 가져온 것입니다 .)

Octavia의 소프트웨어 아키텍처 디자인은 여전히 일반적인 “ 생산자-소비자 ” 모델입니다. API는 작업자와 분리되어 있으며 MessageQueens를 통해 통신합니다.

- `Octavia API` : 표준 RESTful API, Octavia v2 API(기본적으로 활성화됨)는 LBaaS v2 API의 상위 집합이며 이전 버전과 완전히 호환됩니다. 따라서 뒤떨어진 버전의 OS 플랫폼도 Neutron Octavia Driver를 통해 통합할 수 있습니다.
- `Octavia Controller Worker` : Octavia의 핵심입니다. 하단 계층은 Driver & Plugin을 사용하여 OS 플랫폼의 개방성을 나타내며 상위 계층에서 구현되는 세 가지 구성 요소를 지원합니다.
  - `Octavia Worker` : API 요청 완료를 담당하며 Octavia의 주요 기능을 실행합니다.
  - `Health Manager` : 로드밸런서의 고가용성을 보장하는 역할을 담당합니다.
  - `Housekeeping Manager` : 옥타비아의 건전한 운영을 보장하는 진정한 하우스키핑 서비스입니다. SpaceAmphora, DatabaseCleanup 및 CertRotation을 구현했습니다.

참고: 아키텍처 다이어그램에는 하나의 LB 공급자인 Amphora만 표시되어 있지만 Octavia의 드라이버 설계는 실제로 여러 LB 공급자(예: F5)를 지원합니다. 실제로 커뮤니티에서는 항상 openstack/neutron-lbaas 저장소에 구현된 드라이버를 Octavia로 마이그레이션할 계획을 세웠지만 이를 수행할 사람이 부족했습니다.

서비스 프로세스 목록

서비스 목록은 소프트웨어 아키텍처를 구체적으로 표현한 것입니다.

- Octavia API
- Octavia Worker
- Octavia Health Manager
- Octavia Housekeeping

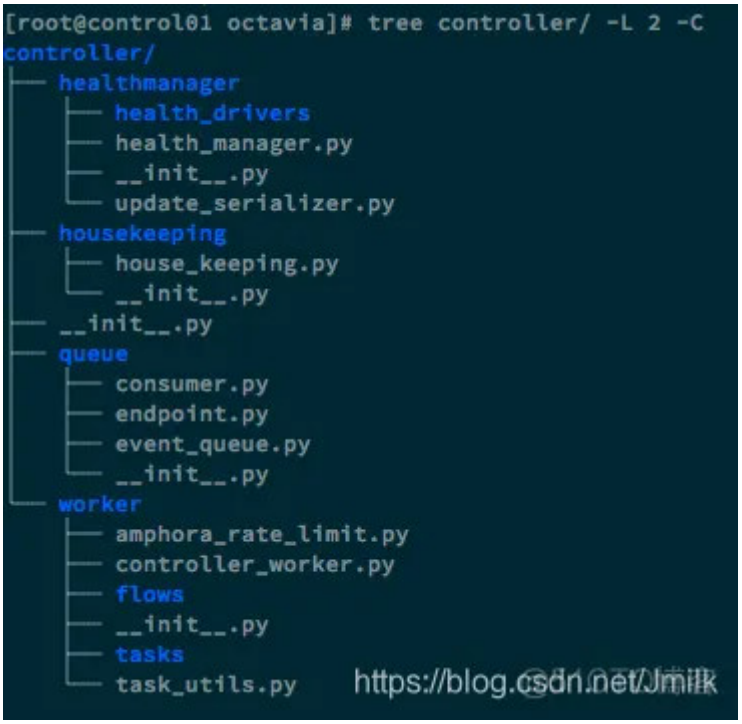
코드 구조



다음은 몇 가지 주요 디렉터리입니다.

- `amphora` : amphora Rest API 및 amphora-agent 구현
- `api` : Octavia API 구현
- `certificates` : CA 인증 구현, amphora와 Octavia Worker 간 HTTPS 통신 및 TLS 기능 지원
- `Compute` : Compute Driver의 추상화와 novaclient의 캡슐화를 구현합니다.
- `network` : 네트워크 드라이버의 추상화와 neutronclient의 캡슐화를 구현합니다.
- `db` : ORM 구현
- `policies` : API 요청에 대한 인증 정책을 정의합니다.

계속해서 컨트롤러 디렉터리를 확장합니다.



- `healthmanager` : Health Manager 구현
- `housekeeping` : HouseKeeping 구현
- `queue` : cotyledon 프레임워크 및 oslo_messaging을 사용하여 내부 RPC 통신 구현
  - `producer`:api/handlers/queue/producer.py
  - `consumer`:controller/queue/consumer.py
- `worker` : 작업 흐름 프레임워크를 사용하여 Octavia Worker 구현
  - `flow` : 작업 흐름을 캡슐화하여 각 작업을 흐름으로 정의합니다.
  - `task` : 태스크를 캡슐화하고 태스크 로직을 추상화하여 태스크의 재사용성을 높입니다.

추신: cotyledon은 oslo.service를 대체하기 위해 커뮤니티에서 개발한 타사 오픈 소스 라이브러리입니다.

Cotyledon은 장기 실행 서비스 정의를 위한 프레임워크를 제공하며 Unix 신호 처리, 작업자 생성, 하위 프로세스 감독, 데몬 다시 로드, sd-notify, 작업자 생성 속도 제한 등을 제공합니다.

이 라이브러리는 주로 OpenStack Telemetry 프로젝트에서 oslo.service를 대체하여 사용됩니다. 그러나 oslo.service는 eventlet에 의존하므로 애플리케이션이 Python 표준 라이브러리를 Monkeypatch하지 않는 경우에는 다른 라이브러리가 필요합니다. 더 이상 Greenlet은 시기적절하지 않습니다. 이로 인해 Tooz 또는 oslo.messaging과 같은 다른 라이브러리가 하트비트 시스템과 함께 실패하게 되었습니다. 또한 Greenpipe가 처리되지 않기 때문에 프로세스가 예상대로 존재하지 않습니다. ——Cotyledon 공식 문서에서 발췌.

OpenStack의 독립 프로젝트인 Octavia의 아키텍처 설계를 요약하면 일관되게 우수한 개방형 설계 아이디어를 계승하고 있으며 Driver 클래스는 LB Provider, Certificates Driver, Compute Driver 및 Network Driver와 같은 외부 지원 노드에서 고도로 추상화되어 Vendor를 만듭니다. 사용자는 기존 인프라에 더 쉽게 연결할 수 있습니다. 이는 의심할 여지 없이 Octavia와 OpenStack이 인기를 끄는 이유 중 하나입니다. 또한 위에서 제기한 질문에 대한 한 가지 측면에 대한 답변이기도 합니다. 왜 이러한 객체를 추상화해야 할까요?

LoadBalaner 프로세스 분석

가장 일반적인 Octavia 구현 사양은 로드 밸런서 생성 프로세스입니다. 우리는 이것을 시작점으로 사용하고 UML 다이어그램의 도움으로 Octavia의 코드 구현을 계속해서 탐구할 것입니다.

CLI:

```
$ openstack loadbalancer create --vip-subnet-id lb-vip-subnet --name lb1
```

API:

```
POST /v2.0/lbaas/loadbalancers
```

요청 본문:

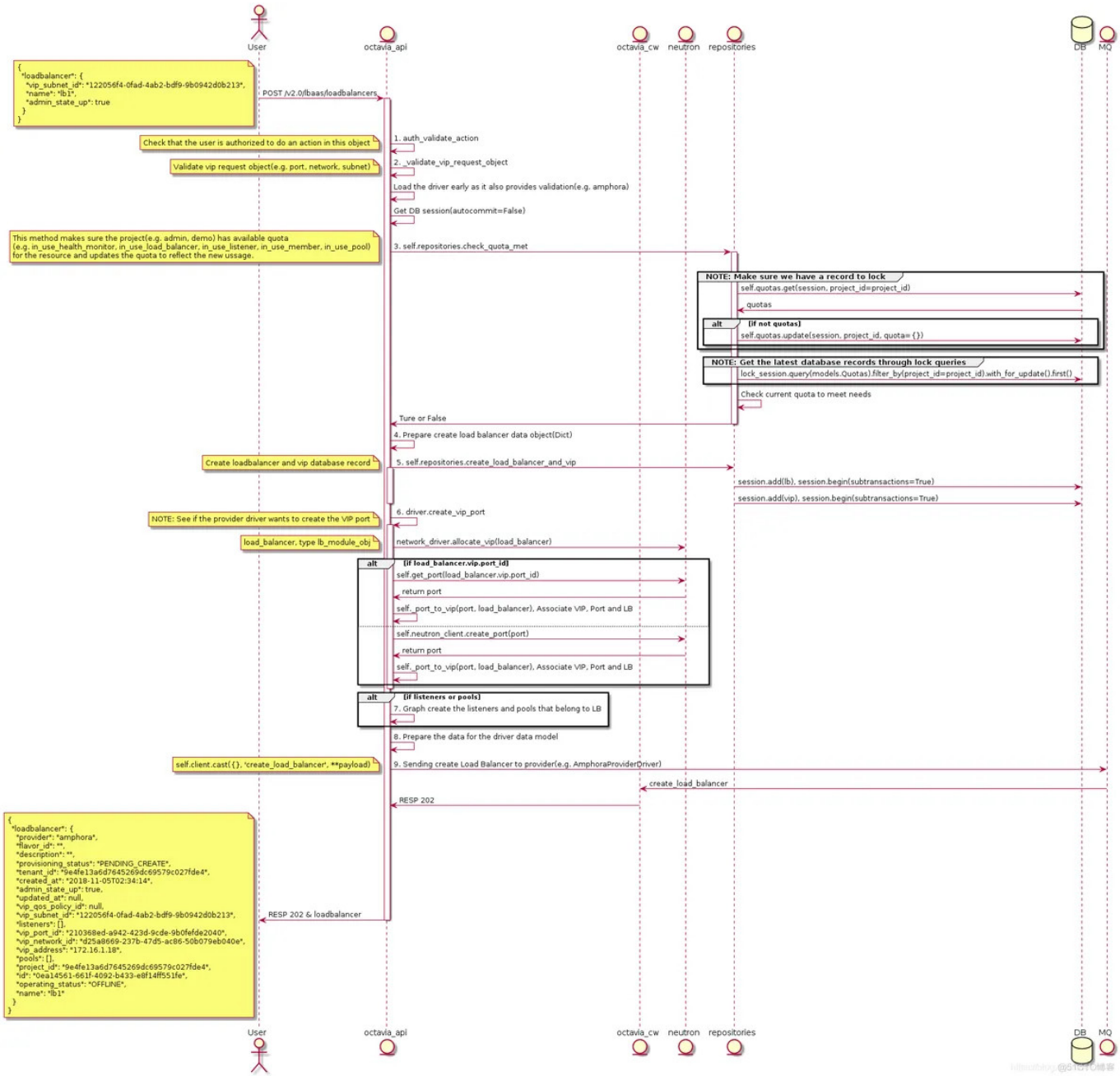
```
{
  "loadbalancer": {
    "vip_subnet_id": "c55e7725-894c-400e-bd00-57a04ae1e676",
    "name": "lb1",
    "admin_state_up": true
  }
}
```

응답:

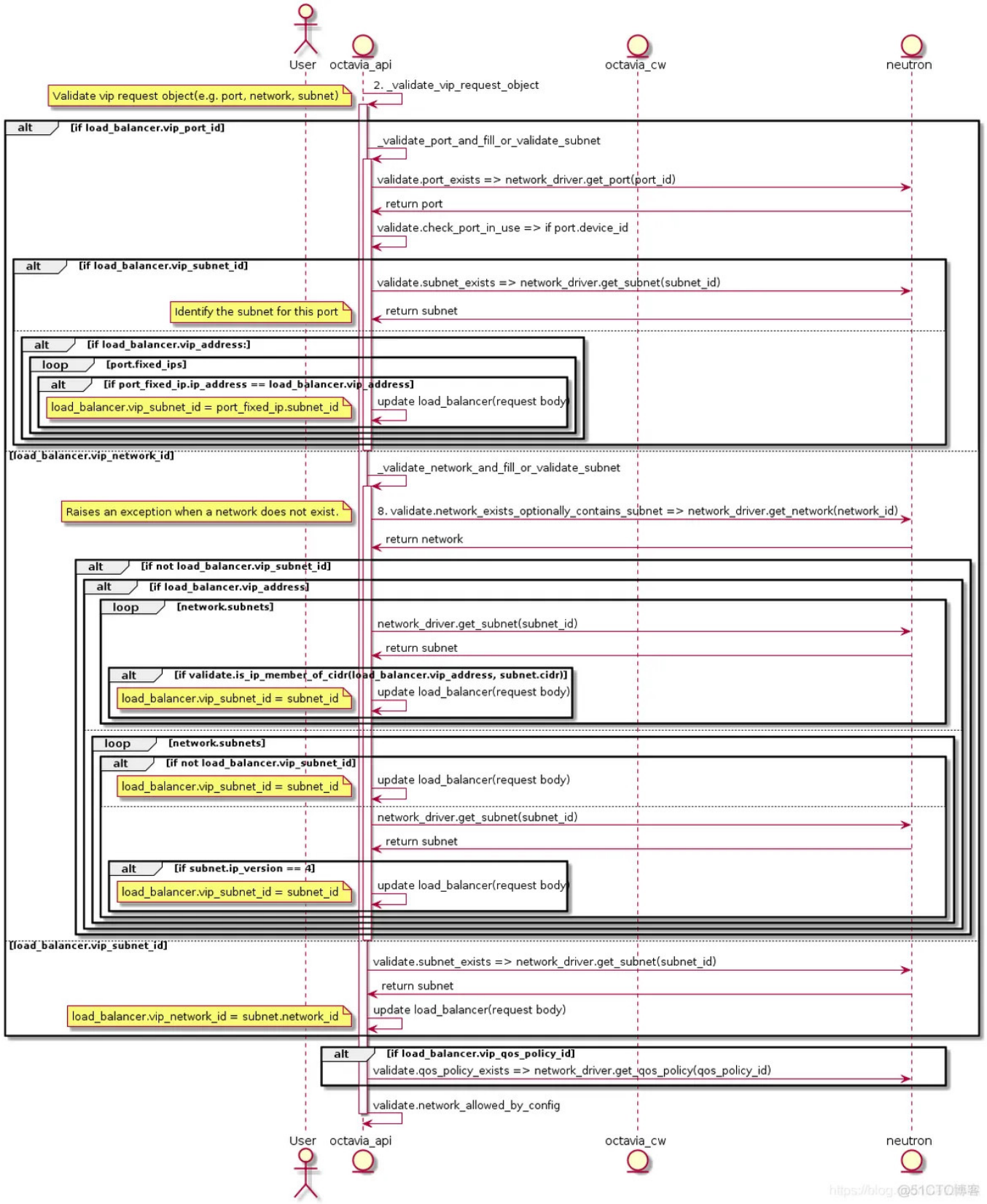
```
{
  "loadbalancer": {
    "provider": "octavia",
    "flavor_id": "",
    "description": "",
    "provisioning_status": "PENDING_CREATE",
    "created_at": "2018-10-22T02:52:04",
    "admin_state_up": true,
    "updated_at": null,
    "vip_subnet_id": "c55e7725-894c-400e-bd00-57a04ae1e676",
    "listeners": [],
    "vip_port_id": "6629fef4-fe14-4b41-9b73-8230105b2e36",
    "vip_network_id": "1078e169-61cb-49bc-a513-915305995be1",
    "vip_address": "10.0.1.7",
    "pools": [],
    "project_id": "2e560efadb704e639ee4bb3953d94afa",
    "id": "5bcf8e3d-9e58-4545-bf80-4c0b905a49ad",
    "operating_status": "OFFLINE",
    "name": "lb1"
```

```
    }
}
```

Create LB의 Octavia API UML 다이어그램 :



2. _validate_vip_request_object의 UML 다이어그램을 확장합니다 .



요청을 받은 `POST /v2.0/lbaas/loadbalancers`후 `octavia-api` 서비스가 처리하는 작업 요약:

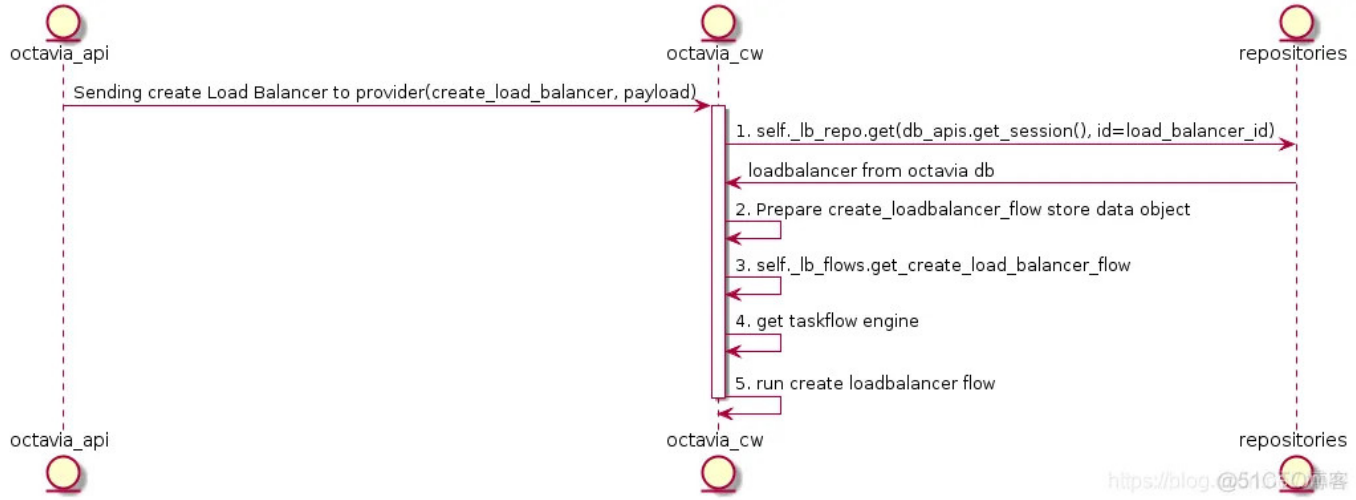
1. 사용자에게 로드밸런서를 생성할 수 있는 권한이 있는지 확인하기 위해 인증을 요청합니다.
2. VIP 및 관련 개체(예: 포트, 서브넷, 네트워크)를 사용할 수 있는지 확인합니다. 여기서 `VIP`를 생성할 때 특정 네트워크 개체 유형을 허용/허용하지 않도록 `config secition [networking]` 구성 할 수 있습니다.
3. 사용자 프로젝트의 LB 할당량을 확인합니다. `config section [quotas]`이를 통해 기본 할당

량을 구성할 수 있습니다(예: Project1을 지정하면 로드 밸런서는 3개만 생성할 수 있습니다).
4. 테이블 load_balancer 및 vip에 대한 데이터베이스 레코드를 생성합니다.
5. Amphora 드라이버(기본 lb 공급자)를 호출하여 VIP에 해당하는 포트를 생성하고 Port, VIP 및 LB의 데이터베이스 레코드를 연결합니다.
6. 그래프 플로우 형태로 로드밸런서에 종속된 리스너와 풀을 생성합니다.
7. create_loadbalancer_flow에 전달된 저장소를 준비합니다.
8. octavia-worker 서비스를 비동기식으로 호출하여 create_loadbalancer_flow를 실행합니다.

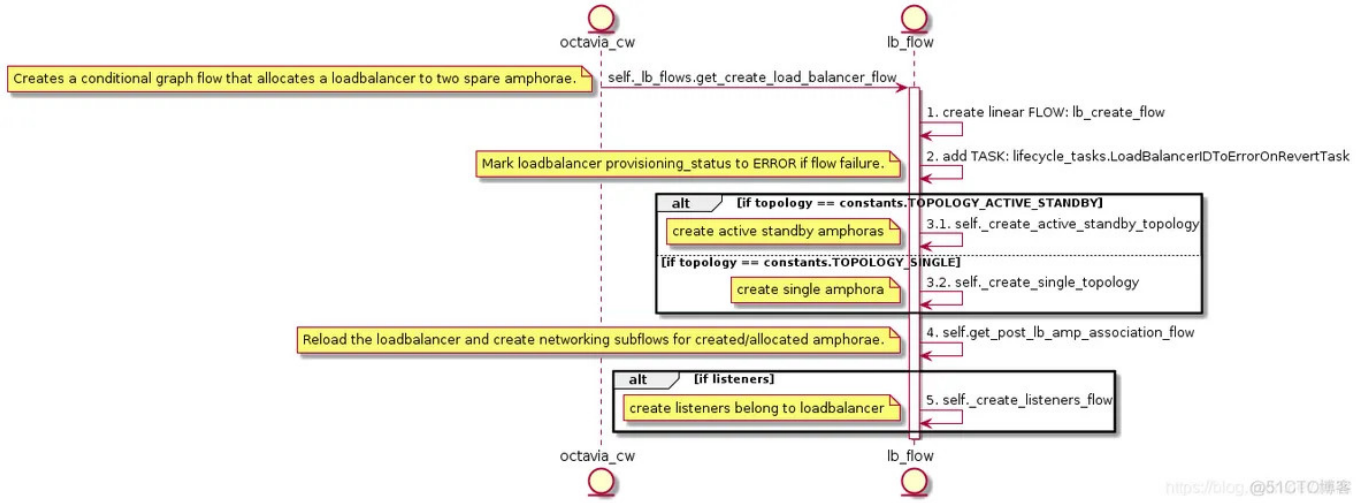
주목할 만한 몇 가지 사항이 있습니다.

- 로드 밸런서 할당량은 여전히 명령을 통해 설정됩니다openstack quota set.
- openstack loadbalancer create 지시어는 로드밸런서와 동시에 하위 리스너와 풀을 생성하는 --listeners 또는 --pools와 같은 옵션을 제공하지 않지만, POST /v2.0/lbaas/loadbalancers는 두 속성을 모두 받을 수 있습니다.따라서 대시보드의 UI/UX를 이에 맞게 최적화할 수 있습니다.
- 지정된 VIP 포트가 존재하지 않는 경우 octavia-api 서비스는 먼저 neutronclient를 호출하여 포트를 생성하고 이름을 loadbalancer-<load_balancer_id> 로 지정 하므로 vip-net에서 이러한 유형의 포트를 볼 수 있습니다.
- VIP는 네트워크, 서브넷, 포트 등 모든 방법을 통한 생성을 지원하며 VIP QoS 설정도 지원합니다.

Create LB의 Octavia Controller Worker UML 다이어그램 :



3. get_create_load_balancer_flow의 UML 다이어그램을 확장합니다 .

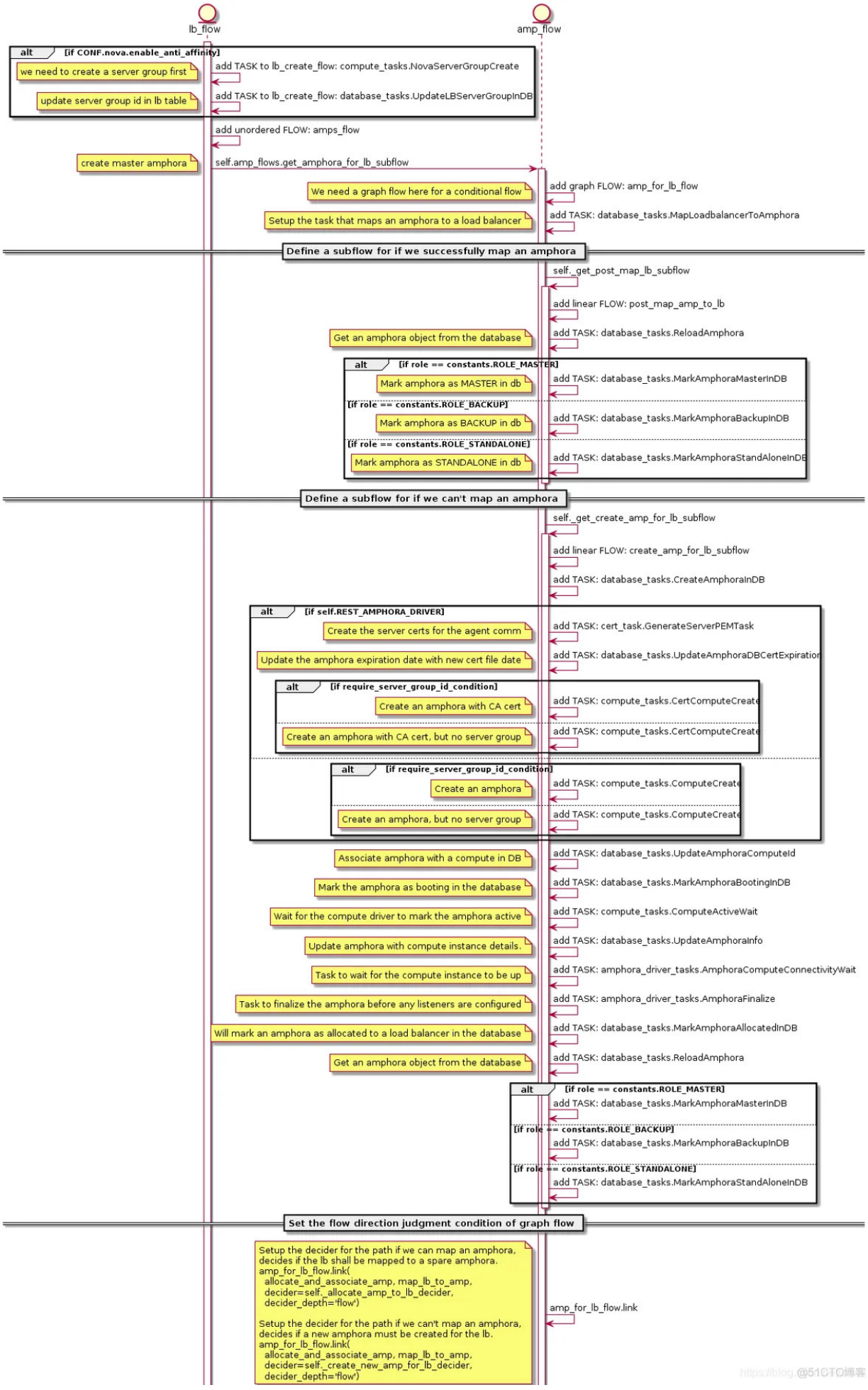


로드 밸런서 흐름 생성에는 두 가지 주요 사항이 있음을 알 수 있습니다.

- 로드 밸런서 토폴로지 생성
- amphora(e)에 대한 네트워킹 생성

먼저 첫 번째 요점을 설명합니다. 소위 로드 밸런서 토폴로지는 본질적으로 amphorae의 고가용성 토폴로지를 의미합니다. SINGLE과 ACTIVE_STANDBY의 두 가지 유형을 지원합니다. 이름에서 알 수 있듯이 SINGLE은 가용성이 높지 않으며 프로덕션 환경에서 사용하지 않는 것이 권장되는 단일 노드 앰포라입니다. 반면 ACTIVE_STANDBY는 Keepalived 마스터/백엔드 마스터-슬레이브 모드에 의존하는 이중 앰포라를 구현합니다. 따라서 이 문서에서는 SINGLE 토폴로지에 대해서는 다루지 않습니다.

Amphora 토폴로지 만들기의 UML 다이어그램 :


lb_flow
amp_flow
alt [if CONF.nova.enable_anti_affinity]
we need to create a server group first
add TASK to lb_create_flow: compute_tasks.NovaServerGroupCreate
update server group id in lb table
add TASK to lb_create_flow: database_tasks.UpdateLBServerGroupInDB
add unordered FLOW: amps_flow
create master amphora
self.amp_flows.get_amphora_for_lb_subflow
We need a graph flow here for a conditional flow
add graph FLOW: amp_for_lb_flow
Setup the task that maps an amphora to a load balancer
add TASK: database_tasks.MapLoadbalancerToAmphora
Define a subflow for if we successfully map an amphora
self._get_post_map_lb_subflow
add linear FLOW: post_map_amp_to_lb
Get an amphora object from the database
add TASK: database_tasks.ReloadAmphora
alt [if role == constants.ROLE_MASTER]
Mark amphora as MASTER in db
add TASK: database_tasks.MarkAmphoraMasterInDB
[if role == constants.ROLE_BACKUP]
Mark amphora as BACKUP in db
add TASK: database_tasks.MarkAmphoraBackupInDB
[if role == constants.ROLE_STANDALONE]
Mark amphora as STANDALONE in db
add TASK: database_tasks.MarkAmphoraStandAloneInDB
Define a subflow for if we can't map an amphora
self._get_create_amp_for_lb_subflow
add linear FLOW: create_amp_for_lb_subflow
add TASK: database_tasks.CreateAmphoraInDB
alt [if self.REST_AMPHORA_DRIVER]
Create the server certs for the agent comm
add TASK: cert_task.GenerateServerPEMTask
Update the amphora expiration date with new cert file date
add TASK: database_tasks.UpdateAmphoraDBCertExpiration
alt [if require_server_group_id_condition]
Create an amphora with CA cert
add TASK: compute_tasks.CertComputeCreate
Create an amphora with CA cert, but no server group
add TASK: compute_tasks.CertComputeCreate
alt [if require_server_group_id_condition]
Create an amphora
add TASK: compute_tasks.ComputeCreate
Create an amphora, but no server group
add TASK: compute_tasks.ComputeCreate
Associate amphora with a compute in DB
add TASK: database_tasks.UpdateAmphoraComputeId
Mark the amphora as booting in the database
add TASK: database_tasks.MarkAmphoraBootingInDB
Wait for the compute driver to mark the amphora active
add TASK: compute_tasks.ComputeActiveWait
Update amphora with compute instance details.
add TASK: database_tasks.UpdateAmphoraInfo
Task to wait for the compute instance to be up
add TASK: amphora_driver_tasks.AmphoraComputeConnectivityWait
Task to finalize the amphora before any listeners are configured
add TASK: amphora_driver_tasks.AmphoraFinalize
Will mark an amphora as allocated to a load balancer in the database
add TASK: database_tasks.MarkAmphoraAllocatedInDB
Get an amphora object from the database
add TASK: database_tasks.ReloadAmphora
alt [if role == constants.ROLE_MASTER]
add TASK: database_tasks.MarkAmphoraMasterInDB
[if role == constants.ROLE_BACKUP]
add TASK: database_tasks.MarkAmphoraBackupInDB
[if role == constants.ROLE_STANDALONE]
add TASK: database_tasks.MarkAmphoraStandAloneInDB
Set the flow direction judgment condition of graph flow
Setup the decider for the path if we can map an amphora, decides if the lb shall be mapped to a spare amphora.
amp_for_lb_flow.link(
allocate_and_associate_amp, map_lb_to_amp,
decider=self._allocate_amp_to_lb_decider,
decider_depth='flow')
Setup the decider for the path if we can't map an amphora, decides if a new amphora must be created for the lb.
amp_for_lb_flow.link(
allocate_and_associate_amp, map_lb_to_amp,
decider=self._create_new_amp_for_lb_decider,
decider_depth='flow')
amp_for_lb_flow.link

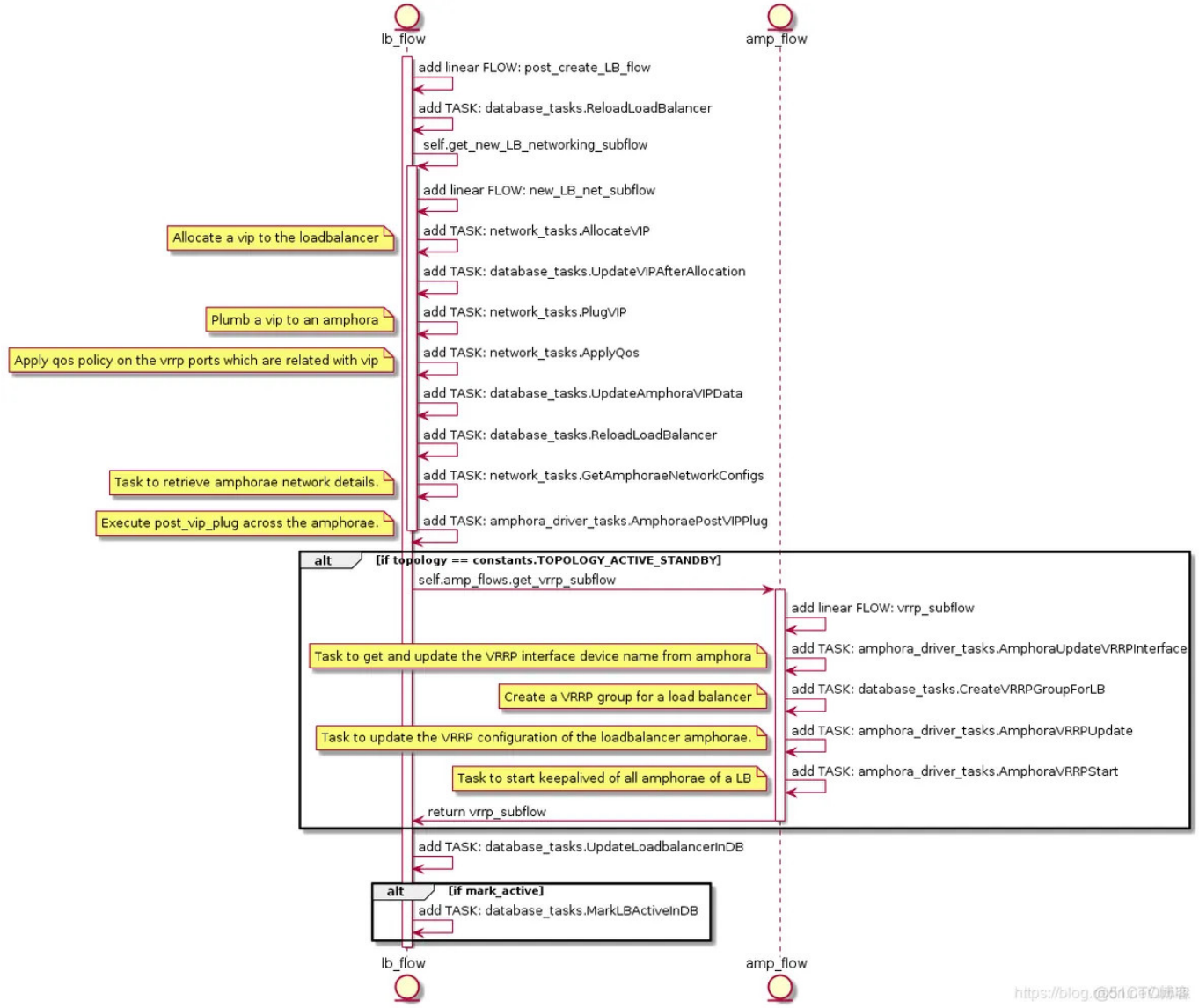
저는 특히 몇 가지 세부 사항을 강조합니다.

- 로드 밸런서 토폴로지가 ACTIVE_STANDBY인 경우 [nova] enable_anti_affinity = True 설정을 통해 Nova의 반친화성 메커니즘을 적용하여 고가용성을 더욱 향상시키도록 할 수도 있습니다.
- 로드밸런서에 대한 암포라 준비는 즉석에서 새로운 암포라를 생성하여 시간 낭비 없이 space amphora pool에서 직접 수행할 수 있습니다. amphora for lb flow는 먼저 space amphora pool에 로드밸런서에 매핑할 수 있는 여유 암포라가 있는지 확인하고, 있으면 직접 매핑합니다. space amphora pool은 Housekeeping Manager 메커니즘에 의해 유지 관리됩니다. 있는 경우 직접 매핑되며, 그렇지 않은 경우 새 앰포라 작업 흐름 생성을 활성화해야 합니다. space amphora pool은 Housekeeping Manager 메커니즘에 의해 유지되며 풀 크기는 구성에 따라 설정됩니다 [house_keeping] spare_amphora_pool_ size=2는 pool size를 설정합니다.
- amphora for lb flow는 그래프 흐름 방식을 사용하며, 이는 무방향성이며 흐름 방향을 사용자 정의할 수 있습니다. 개발자는 사용자 정의 판단 조건(amp_for_lb_flow.link)을 통해 작업 흐름 방향을 제어할 수 있습니다. 이 흐름에 정의된 판단 조건은 다음과 같습니다.

```
if loadbalancer mapping Amphora instance SUCCESS:
    Upload database associations for loadbalancer and amphora
else:
    Create amphora first
    Upload database associations for loadbalancer and amphora
```

두 번째는 amphora가 처음에는 lb-mgmt-net에만 연결되어 있다가 loadbalancer를 할당받은 후에는 vip-net에도 amphora를 연결해야 한다는 점입니다. 이때는 octavia-api 단계에서 vip-net에 생성한 port:loadbalancer-<load_balancer_id>가 이 시점에서 사용됩니다. 또한 ACTIVE_STANDBY 토폴로지를 사용하는 경우 Keepalived VIP 드리프트의 캐리어로 두 개의 암포라 각각에 마운트하기 위해 vip-net에 두 개의 VRRP_port (octavia-lb-vrrp-<amphora_id>)가 생성됩니다.

amphora(e)에 대한 네트워킹 생성을 위한 UML 다이어그램 :



Amphora 네트워킹과 관련된 몇 가지 주요 작업을 나열하십시오.

- `network_tasks.AllocateVIP`
- `network_tasks.PlugVIP`
- `amphora_driver_tasks.AmphoraePostVIPPlug`
- `amphora_driver_tasks.AmphoraVRRPUpdate`
- `amphora_driver_tasks.AmphoraVRRPStart`

실제로는 `Octavia Networking`이 구현의 초점이지만 문제 발생률이 높기 때문에 기본 구현을 마스터해야만 문제를 더 잘 파악할 수 있다고 생각하기 때문에 이러한 구현에 집중해야 합니다.

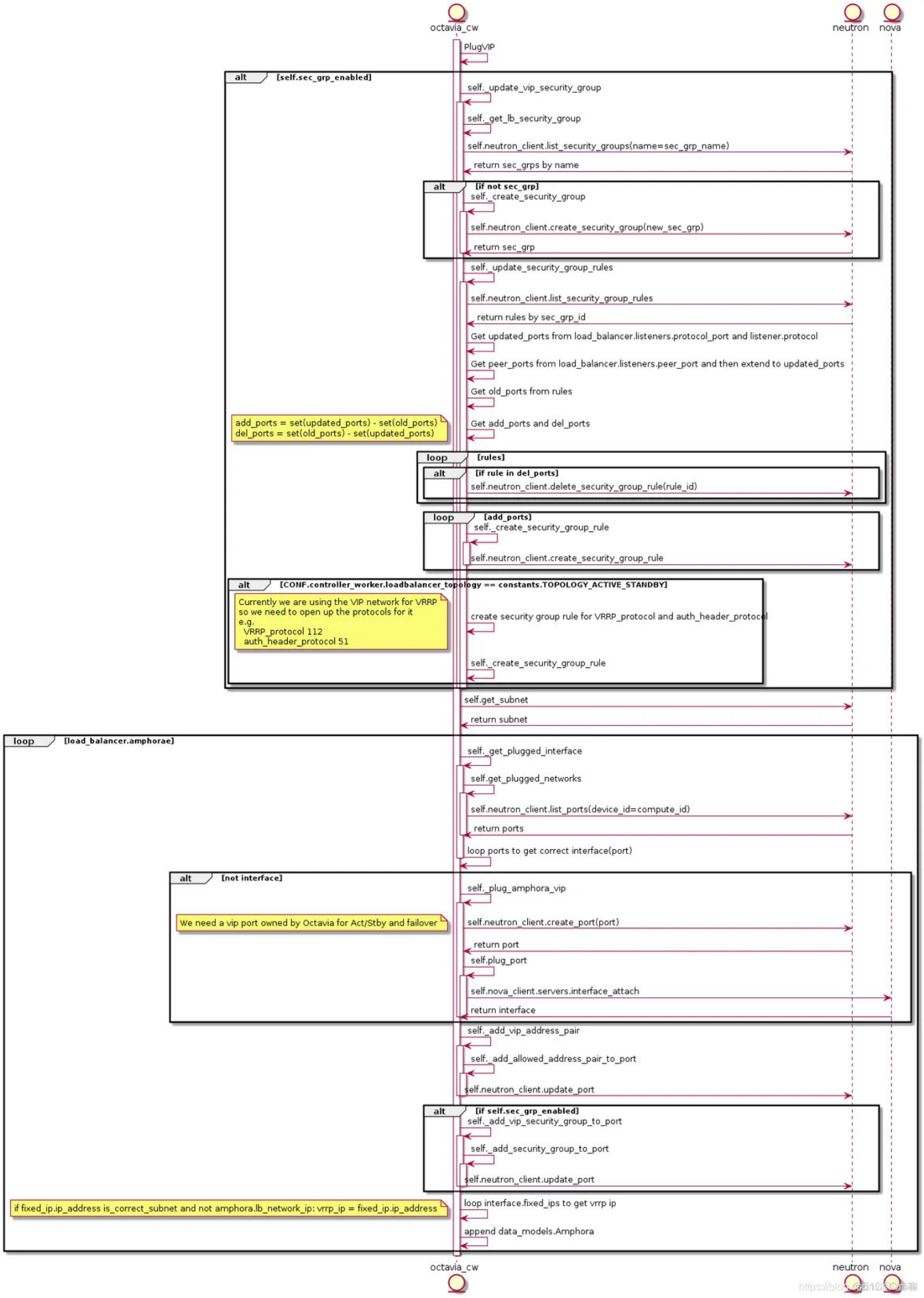
network_tasks.AllocateVIP

`AllocateVIP`는 `VIP`의 포트가 존재하는지 확인하고 `Port`, `VIP` 및 `LB`를 연결하는 `data_models.Vip` 객체를 반환하는 `Neutron`의 인터페이스 래퍼 `AllowedAddressPairsDriver.allocate_vip` 메서드를 호출합니다. 이 메서드는 `octavia-api`에서 한 번 호출되므로 애플리케이션이 `octavia-worker`로 이동할 때 `VIP`의 포트는 이미 생성되어 있고, `data_models.Vip` 라이브러리는 `Task:UpdateAmphoraVIPData`에 의해 유지됩니다.

network_tasks.PlugVIP

AllocateVIP 로드는 Neutron에서 VIP를 할당하고 PlugVIP는 Amphora에 VIP를 삽입하는 역할을 담당합니다.

PlugVIP의 UML 다이어그램



PlugVIP의 로직 구현에는 크게 두 가지 영역이 있습니다.

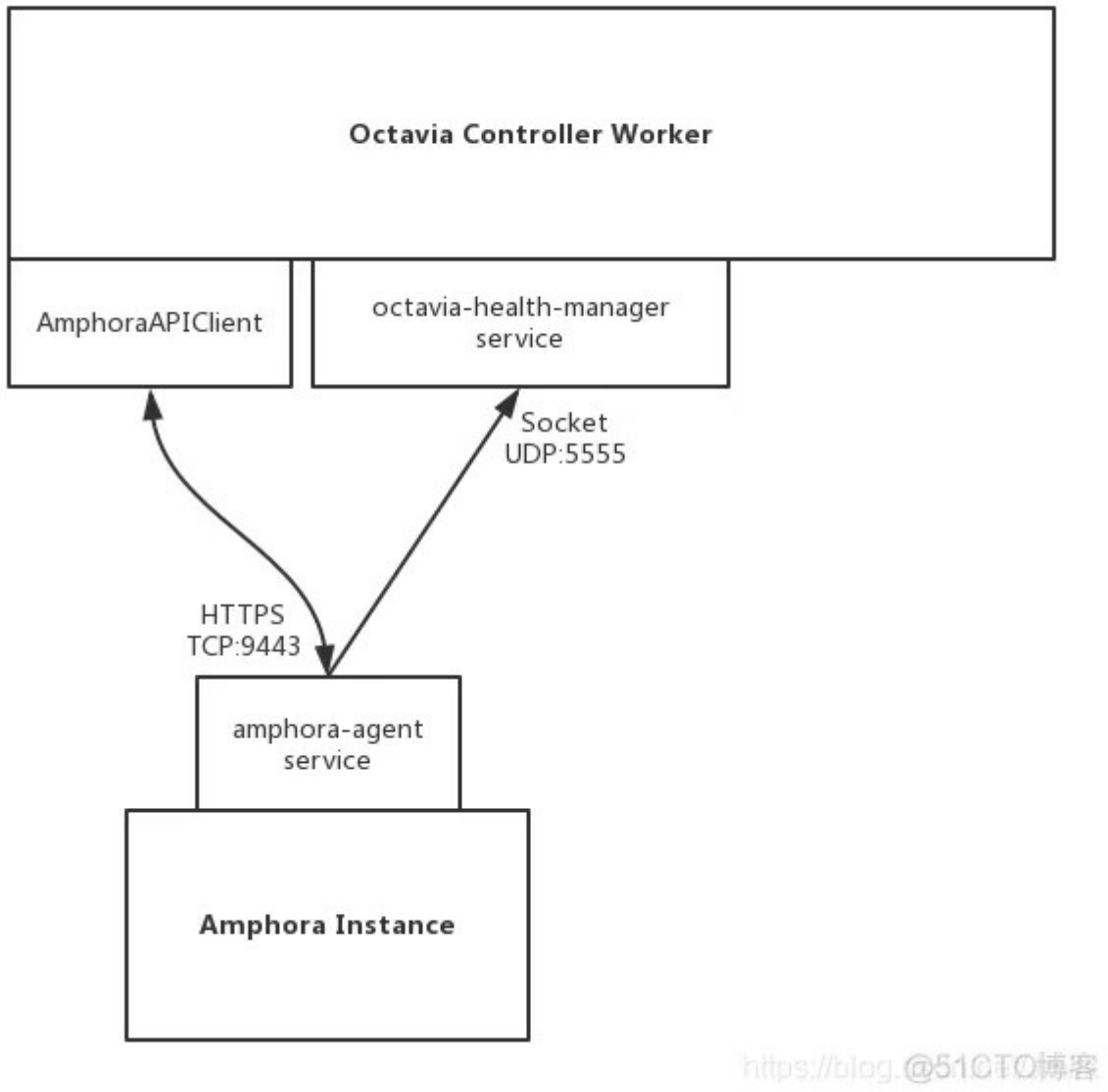
1. VIP 포트의 security_group_rules을 업데이트합니다. 서비스에 대한 외부 액세스는 VIP를 통해 이루어지고 리스너도 VIP에 종속되기 때문에 실제로 VIP의 보안 그룹 규칙은 동적입니다. 예를 들어 로드밸런서에 HTTP:8080 리스너를 추가하면 해당 VIP에 HTTP:8080 규칙이 업로드됩니다.
2. 모든 암포라에 대해 로드밸런서를 폴링하고, 암포라에 필요한 모든 포트가 있는지 확인하고, 없는 경우 Neutron API를 호출하여 포트를 생성한 다음 Nova API를 조사하여 암포라 인스턴스에 마운트합니다.

TASK:AllocateVIP 및 TASK:PlugVIP 이후의 create lb flow은 기본적으로 Amphora의 외부 리소스를 준비하는 것으로 완료되며, 다음 흐름은 Amphora의 내부 구현으로 이동합니다. 흐름의 다음 단계는 Amphora의 내부 구현이 될 것인데, Octavia Controller Worker와 Amphora 에이전트가 어떻게 서로 안전하게 통신할 수 있는지에 대한 문제가 남아 있기 때문입니다. 따라서 Amphora Agent와 AmphoraAPIClient 간의 통신 구현에 대해 논의한 다음, 우리가 이야기하지 않은 나머지 세 가지 작업으로 돌아가 보겠습니다.

Amphora

위에서 언급했듯이 Amphora는 본질적으로 HAProxy와 Keepalived의 런타임 캐리어 역할을 하는 인스턴스입니다. 제 생각에 Amphora는 매우 고전적인 프록시와 유사한 구현으로, '프록시가 어떻게 옥타비아 컨트롤러 워커와 안전하게 통신해야 하는가'라는 문제를 해결하는 데 매우 유용합니다. 하트비트 프로토콜을 사용자 정의하는 방법은 무엇인가요? 호스트의 운영 환경에 미치는 영향을 줄이는 방법은 무엇인가요? 이는 이러한 문제에 대한 훌륭한 데모이며 배우고 연구할 가치가 있습니다!

amphora-agent와 Octavia Controller Worker 간의 통신 모델 다이어그램



먼저, amphora-agent가 AmphoraAPIClient와 통신을 설정하는 방법을 살펴보겠습니다.

Amphora Agent

amphora-agent 서비스 프로세스는 Launch Amphora과 함께 시작되며, 전자는 웹 애플리케이션을 제공하고 후자는 WSGI HTTP 서버 역할을 하는 Flask & gunicorn 구현을 사용합니다. 서비스 프로세스의 주요 기능은 from octavia.cmd.agent import main에서 가져옵니다.

```
# file:
/opt/rocky/octavia/octavia/amphorae/backends/agent/api_server/server.py

class Server(object):
    def __init__(self):
        self.app = flask.Flask(__name__)
        ...
        self.app.add_url_rule(rule=PATH_PREFIX +
'/listeners/<amphora_id>/<listener_id>/haproxy',
```

```
                                view_func=self.upload_haproxy_config,
                                methods=['PUT'])
        ...
```

위의 서버 클래스는 경량 Flask 프레임워크 래퍼 구현인 amphora-agent API의 경로 정의와 보기 기능을 완성하며, 앱 객체는 결국 gunicorn에 의해 로드되어 실행됩니다. 각 route_url의 의미를 이해할 수 있는 공식 문서 Octavia HAProxy Amphora API를 참조하시기 바라며, 여기서는 반복하지 않겠습니다.

AmphoraAPIClient

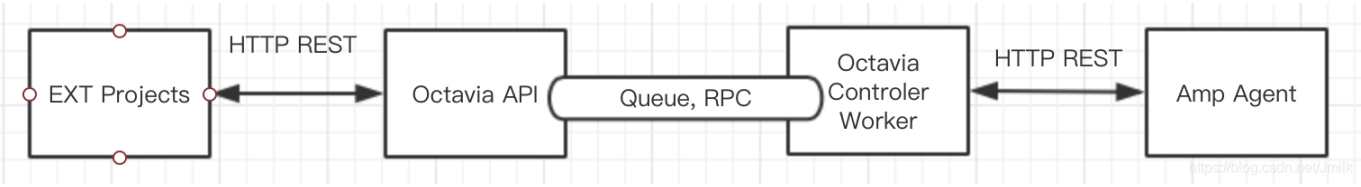
AmphoraAPIClient는 amphora-agent REST API의 클라이언트 측 구현으로, 상위 계층 서비스 호출을 위한 Octavia HAProxy Amphora API에 대한 모든 URL 요청을 캡슐화합니다.

```
# file:
/opt/rocky/octavia/octavia/amphorae/drivers/haproxy/rest_api_driver.py

class AmphoraAPIClient(object):
    def __init__(self):
        super(AmphoraAPIClient, self).__init__()
        self.secure = False
        ...
```

옥타비아의 커뮤니케이션 아키텍처를 돌아보기

- Octavia API: 외부 REST API 통신을 제공합니다.
- Queue: 내부 RPC 통신을 제공합니다.
- Amphora agent: Amphora와 Octavia Controller Worker 간의 REST API 통신을 제공합니다.



AmphoraePostVIPPlug

TASK:AmphoraePostVIPPlug 구현으로 돌아가서, AmphoraePostVIPPlug는 모든 Amphorae를 개별적으로 폴링하여 AmphoraAPIClient를 호출하여 PUT plug/vip/{vip} 요청을 amphora-agent에 전송하여 VM의NIC 구성 파일을 업데이트하고 라우팅 규칙을 추가합니다. 네트워크의 주소 덮어쓰기를 방지하고 Amphora 운영 체제를 깨끗하게 유지하기 위해 AmphoraePostVIPPlug는 Amphora 액세스 lb-mgmt-net을 제외한 모든 NIC가 나뉘어져 있는 네트워크 네임스페이스를 생성합니다.

보시다시피 AmphoraePostVIPPlug의 의미는 VIP용 NIC 장치 파일을 생성하고 여기에 vip-net 포트의 네트워크 정보를 주입하는 것입니다. 구현은 Plug:plug_vip 메서드이며, 다음은 이 작업이 수행되

는 방법을 보여줍니다.

초기 상태의 Amphora에는 lb-mgmt-net과 통신할 수 있는 포트가 하나만 있습니다.

```
root@amphora-cd444019-ce8f-4f89-be6b-0edf76f41b77:~# ifconfig
ens3      Link encap:Ethernet  HWaddr fa:16:3e:b6:8f:a5
          inet addr:192.168.0.9  Bcast:192.168.0.255  Mask:255.255.255.0
          inet6 addr: fe80::f816:3eff:feb6:8fa5/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1450  Metric:1
          RX packets:19462 errors:14099 dropped:0 overruns:0 frame:14099
          TX packets:70317 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1350041 (1.3 MB)  TX bytes:15533572 (15.5 MB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

Amphora가 로드밸런서에 할당되면 vrrp_port 유형의 포트가 추가됩니다. vrrp_port는 Keepalived 가상 경로의 NIC 역할을 하며 네임스페이스(일반적으로 eth1)에 주입됩니다.

```
root@amphora-cd444019-ce8f-4f89-be6b-0edf76f41b77:~# ip netns exec amphora-
haproxy ifconfig
eth1      Link encap:Ethernet  HWaddr fa:16:3e:f4:69:4b
          inet addr:172.16.1.3  Bcast:172.16.1.255  Mask:255.255.255.0
          inet6 addr: fe80::f816:3eff:fef4:694b/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1450  Metric:1
          RX packets:12705 errors:0 dropped:0 overruns:0 frame:0
          TX packets:613211 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:762300 (762.3 KB)  TX bytes:36792968 (36.7 MB)

eth1:0    Link encap:Ethernet  HWaddr fa:16:3e:f4:69:4b
          inet addr:172.16.1.10  Bcast:172.16.1.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1450  Metric:1
```

VRRP IP: 172.16.1.3 및 VIP: 172.16.1.10은 모두 lb-vip-network의 DHCP에 의해 할당되며 포트 octavia-lb-vrrp-<amphora_uuid> 및 octavia-lb-<loadbalancer_uuid>에 해당합니다. 여기서 인터페이스 eth1은 다음과 같이 구성됩니다.

```
root@amphora-cd444019-ce8f-4f89-be6b-0edf76f41b77:~# ip netns exec amphora-
haproxy cat /etc/network/interfaces.d/eth1
```

```
auto eth1
iface eth1 inet dhcp
root@amphora-cd444019-ce8f-4f89-be6b-0edf76f41b77:~# ip netns exec amphora-
haproxy cat /etc/network/interfaces.d/eth1.cfg

# Generated by Octavia agent
auto eth1 eth1:0
iface eth1 inet static
address 172.16.1.3
broadcast 172.16.1.255
netmask 255.255.255.0
gateway 172.16.1.1
mtu 1450

iface eth1:0 inet static
address 172.16.1.10
broadcast 172.16.1.255
netmask 255.255.255.0
# Add a source routing table to allow members to access the VIP
post-up /sbin/ip route add 172.16.1.0/24 dev eth1 src 172.16.1.10 scope link
table 1
post-up /sbin/ip route add default via 172.16.1.1 dev eth1 onlink table 1
post-down /sbin/ip route del default via 172.16.1.1 dev eth1 onlink table 1
post-down /sbin/ip route del 172.16.1.0/24 dev eth1 src 172.16.1.10 scope
link table 1
post-up /sbin/ip rule add from 172.16.1.10/32 table 1 priority 100
post-down /sbin/ip rule del from 172.16.1.10/32 table 1 priority 100
post-up /sbin/iptables -t nat -A POSTROUTING -p udp -o eth1 -j MASQUERADE
post-down /sbin/iptables -t nat -D POSTROUTING -p udp -o eth1 -j MASQUERADE
```

Keepalived 서비스 프로세스 시작

고가용성 서비스 제공을 위해 loadbalancer_topology = ACTIVE_STANDBY일 때만 Keepalived 시작 프로세스가 수행되며, TASK:AmphoraVRRPUpdate와 TASK:AmphoraVRRPStart는 각각 Keepalived 구성 파일의 내용 편집과 Keepalived 서비스 프로세스 시작을 담당합니다.

TASK:AmphoraVRRPUpdate의 로직은 비교적 간단한데, amphora topology의 VIP port, VRRP_ports의 네트워크 정보를 keepalived.conf 설정 파일의 Jinja 템플릿에 렌더링한 후, AmphoraAPIClient를 통해 amphora-agent에 PUT vrrp/upload 요청을 보내 Keepalived 설정 파일의 내용을 업데이트하는 것입니다.

TASK:AmphoraVRRPStart는 AmphoraAPIClient에서 PUT vrrp/start 요청을 전송하여 amphora-agent의 view_func:manage_service_vrrp(action=start)를 실행합니다.

```
# file:
/opt/rocky/octavia/octavia/amphorae/backends/agent/api_server/keepalived.py

    def manager_keepalived_service(self, action):
```

```
        action = action.lower()
        if action not in [consts.AMP_ACTION_START,
                          consts.AMP_ACTION_STOP,
                          consts.AMP_ACTION_RELOAD]:
            return webob.Response(json=dict(
                message='Invalid Request',
                details="Unknown action: {0}".format(action)), status=400)

        if action == consts.AMP_ACTION_START:
            keepalived_pid_path = util.keepalived_pid_path()
            try:
                # Is there a pid file for keepalived?
                with open(keepalived_pid_path, 'r') as pid_file:
                    pid = int(pid_file.readline())
                os.kill(pid, 0)

                # If we got here, it means the keepalived process is
running.
                # We should reload it instead of trying to start it again.
                action = consts.AMP_ACTION_RELOAD
            except (IOError, OSError):
                pass

        cmd = ("/usr/sbin/service octavia-keepalived {action}".format(
            action=action))

        try:
            subprocess.check_output(cmd.split(), stderr=subprocess.STDOUT)
        except subprocess.CalledProcessError as e:
            LOG.debug('Failed to %s octavia-keepalived service: %s %s',
                      action, e, e.output)
            return webob.Response(json=dict(
                message="Failed to {0} octavia-keepalived service".format(
                    action), details=e.output), status=500)

        return webob.Response(
            json=dict(message='OK',
                      details='keepalived
{action}ed'.format(action=action)),
            status=202)
```

분명히 amphora-agent는 /usr/sbin/service octavia-keepalived start 명령을 실행하여 keepalived서비스 프로세스를 시작합니다. octavia-keepalived.service의 내용을 살펴보세요:

```
# file: /usr/lib/systemd/system/octavia-keepalived.service

[Unit]
Description=Keepalive Daemon (LVS and VRRP)
After=network-online.target .service
```

```
Wants=network-online.target
Requires=.service

[Service]
# Force context as we start keepalived under "ip netns exec"
SELinuxContext=system_u:system_r:keepalived_t:s0
Type=forking
KillMode=process

ExecStart=/sbin/ip netns exec amphora-haproxy /usr/sbin/keepalived  -D -d -f
/var/lib/octavia/vrrp/octavia-keepalived.conf -p
/var/lib/octavia/vrrp/octavia-keepalived.pid

ExecReload=/bin/kill -HUP $MAINPID
PIDFile=/var/lib/octavia/vrrp/octavia-keepalived.pid

[Install]
WantedBy=multi-user.target
```

위 내용에 따르면

- 실제 keepalived 서비스 프로세스는 namespace amphora-haproxy에서 시작됩니다.
- keepalived 구성 파일은 /var/lib/octavia/vrrp/octavia-keepalived.conf입니다.

view_func:manage_service_vrrp는 시작 외에도 중지 및 다시 로드 작업을 지원하며, keepalived 구성 파일에 대한 업데이트는 view_func:upload_keepalived_config에서 처리합니다.

keepalived 설정 파일의 내용으로 넘어가 보겠습니다.

```
# file: /var/lib/octavia/vrrp/octavia-keepalived.conf

vrrp_script check_script {
  script /var/lib/octavia/vrrp/check_script.sh   # VRRP check
  interval 5
  fall 2
  rise 2
}

vrrp_instance 01197be798d5440da846cd70f52dc503 { # VRRP instance name is
loadbalancer UUID
  state MASTER                                   # Master router
  interface eth1                                 # VRRP IP device
  virtual_router_id 1                            # VRID
  priority 100
  nopreempt
  garp_master_refresh 5
  garp_master_refresh_repeat 2
  advert_int 1
```

```
  authentication {
    auth_type PASS
    auth_pass b76d77e
  }

  unicast_src_ip 172.16.1.3                        # VRRP IP
  unicast_peer {
    172.16.1.7                                     # Backup router VRRP IP
  }

  virtual_ipaddress {
    172.16.1.10                                    # VIP address
  }
  track_script {
    check_script
  }
}
```

보시다시피, keepalived는 eth1을 VRRP IP와 VIP를 위한 인터페이스로 사용하며, 또한 eth1은 이미 TASK:AmphoraePostVIPPlug에서 namespace amphora에 준비되어 있습니다.

check_script.sh 스크립트는 VIP 드리프트를 결정하기 위한 기준으로 각 Amphorae의 HAProxy의 상태를 확인하는 데 사용됩니다.

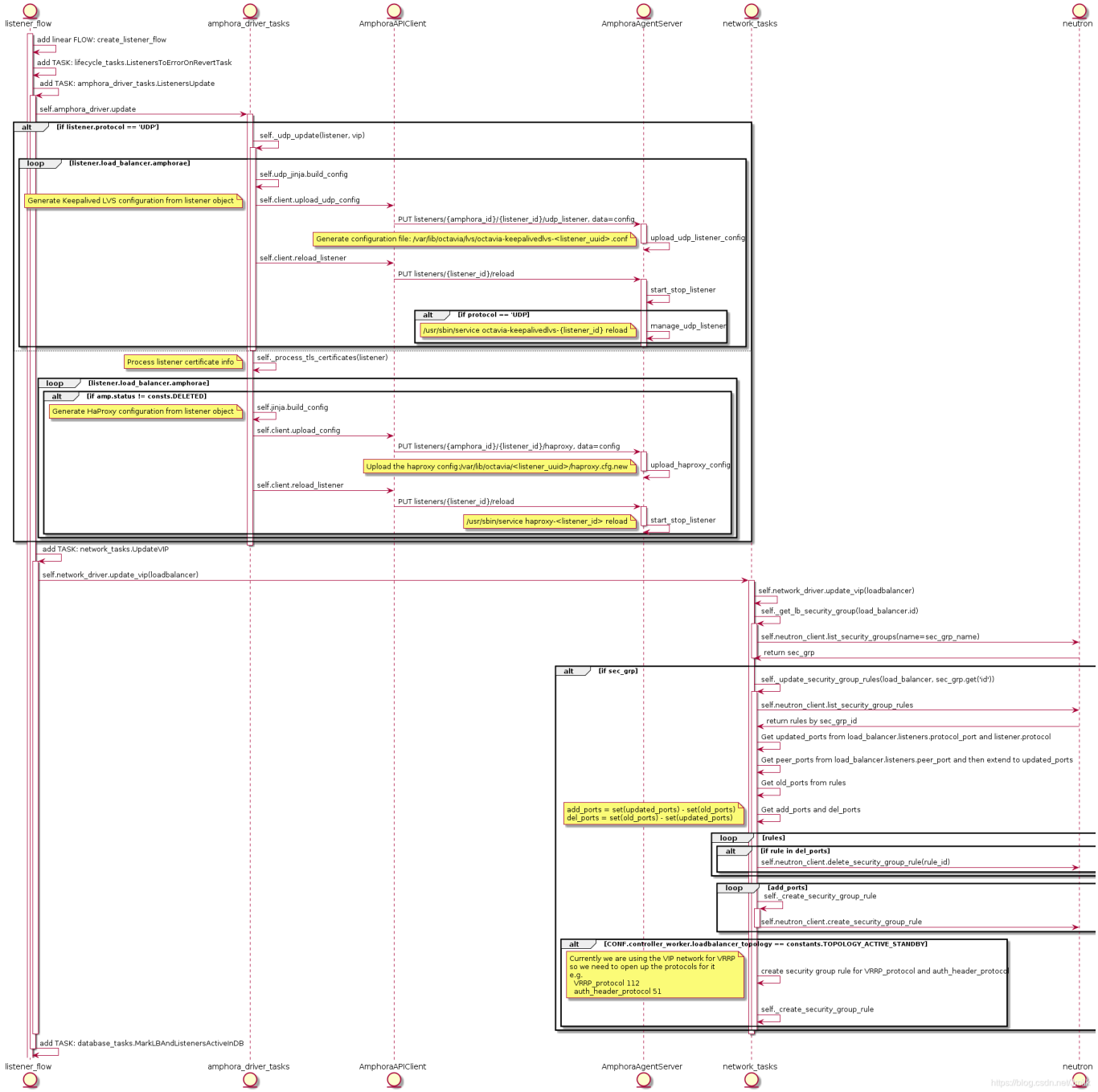
```
root@amphora-caa6ba0f-1a68-4f22-9be9-8521695ac4f4:~# cat
/var/lib/octavia/vrrp/check_scripts/haproxy_check_script.sh
haproxy-vrrp-check /var/lib/octavia/d367b5ec-24dd-44b3-b947-
e0ff72c75e66.sock; exit $?
```

Amphora Instance는 amphora-agent와 keepalived도 실행하지만 리스너가 생성될 때만 시작되는 haproxy도 실행합니다. haproxy는 리스너가 생성된 후에야 시작되므로 리스너 웨어러 프로세스를 분석할 때까지 기다리겠습니다.

로드밸런서를 만드는 과정을 분석해보니, 간단히 말해 amphorae를 준비하고 amphorae를 vip-net에 연결하는 것이지만 그 사이에는 음미할 만한 세부 사항이 많이 있습니다.

리스너 생성 프로세스 분석

리스너 흐름 생성 UML 도표



위 그림에서 볼 수 있듯이, `openstack loadbalancer listener create –protocol HTTP –protocol-port 8080 lb-1` 명령을 실행하여 리스너를 생성하면 `Task:ListenersUpdate`로 실행되며, 여기서 `AmphoraAPIClient`가 호출됩니다:

- `PUT listeners/{amphora_id}/{listener_id}/haproxy`: `haproxy` 구성 파일 업데이트
- `PUT listeners/{listener_id}/reload` : `haproxy` 서비스 프로세스 재시작

따라서 `haproxy` 서비스 프로세스는 로드밸런서에 대한 리스너가 생성될 때만 시작됩니다. 또한 리스너가 생성될 때 Listener에 포함된 프로토콜 및 포트 정보를 VIP의 보안 그룹 규칙에서 업데이트해야 하기 때문에 `Task:UpdateVIP`도 실행됩니다.

haproxy 서비스 프로세스 시작

`amphora`에 로그인하여 `haproxy` 구성 파일을 확인합니다.

```
# file: /var/lib/octavia/1385d3c4-615e-4a92-aea1-c4fa51a75557/haproxy.cfg,
Listener UUID: 1385d3c4-615e-4a92-aea1-c4fa51a75557

# Configuration for loadbalancer 01197be7-98d5-440d-a846-cd70f52dc503
global
    daemon
    user nobody
    log /dev/log local0
    log /dev/log local1 notice
    stats socket /var/lib/octavia/1385d3c4-615e-4a92-aea1-c4fa51a75557.sock
mode 0666 level user
    maxconn 1000000

defaults
    log global
    retries 3
    option redispatch

peers 1385d3c4615e4a92aea1c4fa51a75557_peers
    peer l_Ustq0qE-h-_Q1dlXLXBAiWR8U 172.16.1.7:1025
    peer O08zAgUhIv9TEXhyYZf2iHdxOkA 172.16.1.3:1025


frontend 1385d3c4-615e-4a92-aea1-c4fa51a75557
    option httplog
    maxconn 1000000
    bind 172.16.1.10:8080
    mode http
    timeout client 50000
```

리스너는 HTTP 프로토콜 및 포트 8080에서 수신하도록 지정하므로 fronted section도 bind 172.16.1.10:8080 및 mode http 구성 항목으로 렌더링됩니다.

Amphora 운영 체제에서 시작된 haproxy 프로세스는 haproxy-1385d3c4-615e-4a92-aea1-c4fa51a75557.service(ListenerUUID:1385d3c4-615e-4a92-aea1-c4fa51a75557)입니다. 이 프로세스에 대한 service 구성 보기

```
# file: /usr/lib/systemd/system/haproxy-1385d3c4-615e-4a92-aea1-
c4fa51a75557.service

[Unit]
Description=HAProxy Load Balancer
After=network.target syslog.service amphora-netns.service
Before=octavia-keepalived.service
Wants=syslog.service
Requires=amphora-netns.service

[Service]
# Force context as we start haproxy under "ip netns exec"
```

```
SELinuxContext=system_u:system_r:haproxy_t:s0

Environment="CONFIG=/var/lib/octavia/1385d3c4-615e-4a92-aea1-
c4fa51a75557/haproxy.cfg" "USERCONFIG=/var/lib/octavia/haproxy-default-user-
group.conf" "PIDFILE=/var/lib/octavia/1385d3c4-615e-4a92-aea1-
c4fa51a75557/1385d3c4-615e-4a92-aea1-c4fa51a75557.pid"

ExecStartPre=/usr/sbin/haproxy -f $CONFIG -f $USERCONFIG -c -q -L
O08zAgUhIv9TEXhyYZf2iHdxOkA

ExecReload=/usr/sbin/haproxy -c -f $CONFIG -f $USERCONFIG -L
O08zAgUhIv9TEXhyYZf2iHdxOkA
ExecReload=/bin/kill -USR2 $MAINPID

ExecStart=/sbin/ip netns exec amphora-haproxy /usr/sbin/haproxy-systemd-
wrapper -f $CONFIG -f $USERCONFIG -p $PIDFILE -L O08zAgUhIv9TEXhyYZf2iHdxOkA

KillMode=mixed
Restart=always
LimitNOFILE=2097152

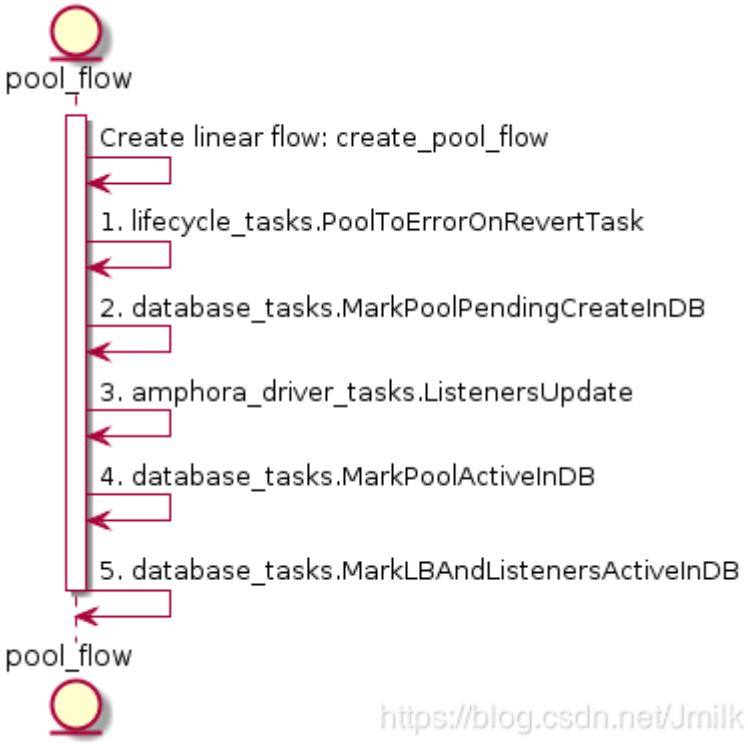
[Install]
WantedBy=multi-user.target
```

구성에서 실제 서비스가 시작된 것은 `/usr/sbin/haproxy-systemd-wrapper`이며, namespace `amphora-haproxy`에서도 실행되고 있음을 알 수 있으며, 로그에서 `/usr/sbin/haproxy` 지시어를 호출하는 것만 볼 수 있습니다.

```
Nov 15 10:12:01 amphora-cd444019-ce8f-4f89-be6b-0edf76f41b77 ip[13206]:
haproxy-systemd-wrapper: executing /usr/sbin/haproxy -f
/var/lib/octavia/1385d3c4-615e-4a92-aea1-c4fa51a75557/haproxy.cfg -f
/var/lib/octavia/haproxy-default-user-group.conf -p
/var/lib/octavia/1385d3c4-615e-4a92-aea1-c4fa51a75557/1385d3c4-615e-4a92-
aea1-c4fa51a75557.pid -L O08zAgUhIv9TEXhyYZf2iHdxOkA -Ds
```

리스너 외에도 풀, 멤버, L7정책, L7규칙, `Health Monitor`와 같은 개체를 만들면 `haproxy` 구성 변경에도 영향을 미칩니다.

풀 생성 프로세스 분석



풀 플로우 생성 UML 도표

`create pool flow`하는 데 가장 중요한 작업은 `haproxy` 구성 파일을 업데이트하는 `Task:ListenersUpdate`입니다.

`openstack loadbalancer pool create –protocol HTTP –lb-algorithm ROUND_ROBIN –listener 1385d3c4-615e-4a92-aea1-c4fa51a75557` 명령이 리스너에 대해 실행될 때 리스너에 대한 `default pool`이 생성되면 `haproxy.cfg`는 `backend section`을 추가하고 명령으로 전달된 파라미터에 따라 `backend mode http` 및 `balance roundrobin`을 렌더링합니다.

```
# Configuration for loadbalancer 01197be7-98d5-440d-a846-cd70f52dc503
global
    daemon
    user nobody
    log /dev/log local0
    log /dev/log local1 notice
    stats socket /var/lib/octavia/1385d3c4-615e-4a92-aea1-c4fa51a75557.sock
mode 0666 level user
    maxconn 1000000

defaults
    log global
    retries 3
    option redispatch

peers 1385d3c4615e4a92aea1c4fa51a75557_peers
    peer l_Ustq0qE-h-_Q1dlXLXBAiWR8U 172.16.1.7:1025
    peer O08zAgUhIv9TEXhyYZf2iHdxOkA 172.16.1.3:1025


frontend 1385d3c4-615e-4a92-aea1-c4fa51a75557
    option httplog
```

```
    maxconn 1000000
    bind 172.16.1.10:8080
    mode http
    default_backend 8196f752-a367-4fb4-9194-37c7eab95714       # UUID of
pool
    timeout client 50000

backend 8196f752-a367-4fb4-9194-37c7eab95714
    mode http
    balance roundrobin
    fullconn 1000000
    option allbackups
    timeout connect 5000
    timeout server 50000
```

풀을 생성할 때 `listener uuid` 또는 `loadbalancer uuid`를 지정할 수 있는데, 전자를 지정하면 리스너에 대한 `default pool`이 지정되고 리스너는 하나의 `default pool`만 가질 수 있으며 `default pool`을 반복 지정하면 예외가 트리거되고 `loadbalancer uuid`를 지정하면 `shared pool`이 생성된다는 점에 유의할 필요가 있습니다. `shared pool`은 동일한 로드밸런서 아래의 모든 리스너가 공유할 수 있으며 종종 `l7policy` 기능의 구현을 지원하는 데 사용됩니다. 공유 풀은 리스너의 l7policy 동작이 " 다른 풀로 전달 " 로 설정된 경우 선택할 수 있으며, 공유 풀은 동일한 로드밸런서 아래의 모든 리스너의 전달 요청을 수락할 수 있습니다. 다음 명령을 실행하여 공유 풀을 생성합니다.

```
$ openstack loadbalancer pool create --protocol HTTP --lb-algorithm
ROUND_ROBIN --loadbalancer 01197be7-98d5-440d-a846-cd70f52dc503
+---------------------+--------------------------------------+
| Field               | Value                                |
+---------------------+--------------------------------------+
| admin_state_up      | True                                 |
| created_at          | 2018-11-20T03:35:08                  |
| description         |                                      |
| healthmonitor_id    |                                      |
| id                  | 822f78c3-ea2c-4770-bef0-e97f1ac2eba8 |
| lb_algorithm        | ROUND_ROBIN                          |
| listeners           |                                      |
| loadbalancers       | 01197be7-98d5-440d-a846-cd70f52dc503 |
| members             |                                      |
| name                |                                      |
| operating_status    | OFFLINE                              |
| project_id          | 9e4fe13a6d7645269dc69579c027fde4     |
| protocol            | HTTP                                 |
| provisioning_status | PENDING_CREATE                       |
| session_persistence | None                                 |
| updated_at          | None                                 |
+---------------------+--------------------------------------+
```

리스너에 바인딩하지 않고 단순히 공유 풀을 만들면 `haproxy.cfg` 구성 파일이 즉시 변경되지 않는다

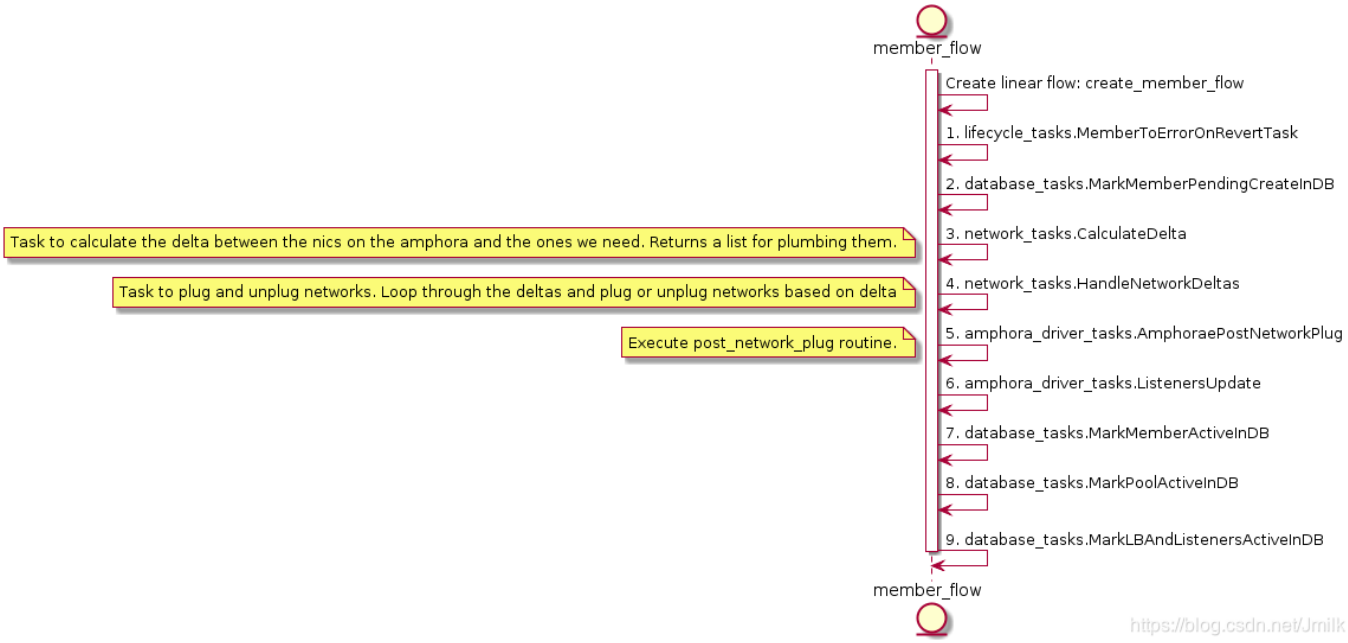
는 점에 유의하세요.

멤버 프로세스 분석 만들기

다음 명령을 사용하여 클라우드 호스트가 위치한 서브넷, IP 주소 및 수신된 데이터가 전달되는 protocol-port를 지정하는 옵션과 함께 기본 풀에 구성원을 만듭니다.

```
[root@control01 ~]# openstack loadbalancer member create --subnet-id
2137f3fb-00ee-41a9-b66e-06705c724a36 --address 192.168.1.14 --protocol-port
80 8196f752-a367-4fb4-9194-37c7eab95714
+---------------------+--------------------------------------+
| Field               | Value                                |
+---------------------+--------------------------------------+
| address             | 192.168.1.14                         |
| admin_state_up      | True                                 |
| created_at          | 2018-11-20T06:09:58                  |
| id                  | b6e464fd-dd1e-4775-90f2-4231444a0bbe |
| name                |                                      |
| operating_status    | NO_MONITOR                           |
| project_id          | 9e4fe13a6d7645269dc69579c027fde4     |
| protocol_port       | 80                                   |
| provisioning_status | PENDING_CREATE                       |
| subnet_id           | 2137f3fb-00ee-41a9-b66e-06705c724a36 |
| updated_at          | None                                 |
| weight              | 1                                    |
| monitor_port        | None                                 |
| monitor_address     | None                                 |
| backup              | False                                |
+---------------------+--------------------------------------+
```

octavia-api 수준에서는 먼저 CONF.networking.reserved_ips를 구성하여 멤버의 ipaddress를 사용할 수 있는지, 멤버의 서브넷이 존재하는지 확인한 다음 octavia-worker로 이동합니다.



몇 가지 주요 작업은 다음과 같이 확장됩니다.

CalculateDelta

TASK:CalculateDelta는 로드밸런서에서 Amphora를 폴링하여 Amphora의 기존 NIC 세트와 필요한 예상 NIC 세트 간의 " 차이 " 를 계산하는Task:CalculateAmphoraDelta를 실행합니다. "

```
# file: /opt/rocky/octavia/octavia/controller/worker/tasks/network_tasks.py


class CalculateAmphoraDelta(BaseNetworkTask):

    default_provides = constants.DELTA

    def execute(self, loadbalancer, amphora):
        LOG.debug("Calculating network delta for amphora id: %s",
amphora.id)

        # Figure out what networks we want
        # seed with lb network(s)
        vrrp_port = self.network_driver.get_port(amphora.vrrp_port_id)
        desired_network_ids = {vrrp_port.network_id}.union(
            CONF.controller_worker.amp_boot_network_list)

        for pool in loadbalancer.pools:
            member_networks = [
                self.network_driver.get_subnet(member.subnet_id).network_id
                for member in pool.members
                if member.subnet_id
            ]
            desired_network_ids.update(member_networks)

        nics = self.network_driver.get_plugged_networks(amphora.compute_id)
        # assume we don't have two nics in the same network
        actual_network_nics = dict((nic.network_id, nic) for nic in nics)

        del_ids = set(actual_network_nics) - desired_network_ids
        delete_nics = list(
            actual_network_nics[net_id] for net_id in del_ids)

        add_ids = desired_network_ids - set(actual_network_nics)
        add_nics = list(n_data_models.Interface(
            network_id=net_id) for net_id in add_ids)
        delta = n_data_models.Delta(
            amphora_id=amphora.id, compute_id=amphora.compute_id,
            add_nics=add_nics, delete_nics=delete_nics)
        return delta
```

간단히 말해, 먼저 필요할 것으로 예상되는 desired_network_ids와 이미 존재하는

actual_network_nics를 가져온 다음, 삭제할 delete_nics와 추가할 add_nics를 계산하고 마지막으로 Delta data models을 Task:HandleNetworkDeltas에 반환하면 Amphora NIC의 실제 마운팅 및 마운트 해제가 수행됩니다.

HandleNetworkDeltas

Task:HandleNetworkDelta는 Amphora Delta를 기반으로 네트워크 마운트 및 마운트 해제를 로드합니다.

```
# file: /opt/rocky/octavia/octavia/controller/worker/tasks/network_tasks.py


class HandleNetworkDelta(BaseNetworkTask):
    """Task to plug and unplug networks

    Plug or unplug networks based on delta
    """

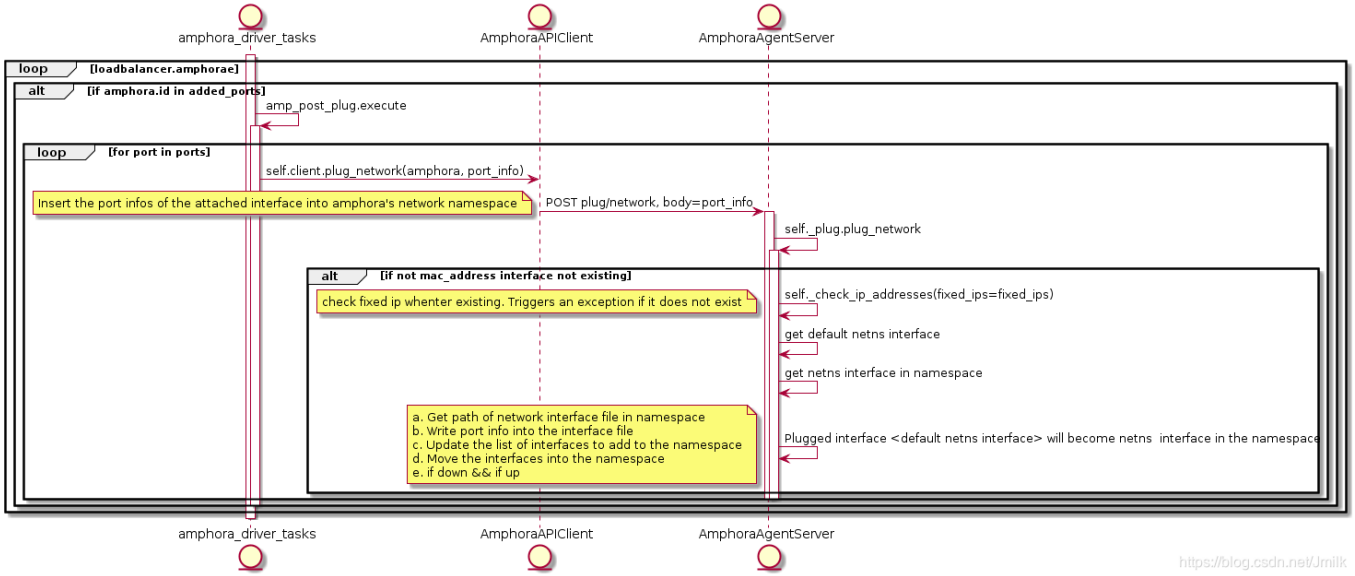
    def execute(self, amphora, delta):
        """Handle network plugging based off deltas."""
        added_ports = {}
        added_ports[amphora.id] = []
        for nic in delta.add_nics:
            interface = self.network_driver.plug_network(delta.compute_id,
                                                         nic.network_id)
            port = self.network_driver.get_port(interface.port_id)
            port.network = self.network_driver.get_network(port.network_id)
            for fixed_ip in port.fixed_ips:
                fixed_ip.subnet = self.network_driver.get_subnet(
                    fixed_ip.subnet_id)
            added_ports[amphora.id].append(port)
        for nic in delta.delete_nics:
            try:
                self.network_driver.unplug_network(delta.compute_id,
                                                   nic.network_id)
            except base.NetworkNotFound:
                LOG.debug("Network %d not found ", nic.network_id)
            except Exception:
                LOG.exception("Unable to unplug network")
        return added_ports
```

마지막으로, added_port return을 후속 TASK:AmphoraePostNetworkPlug에 전달하여 사용합니다.

AmphoraePostNetworkPlug

Task: AmphoraePostNetworkPlug는 member가 속한 네트워크의 port 정보를 network namespace에 주입하는 역할을 담당합니다. AmphoraePostNetworkPlug와 AmphoraePostVIPPlug를 구분해야 합니다. 전자는 create member flow에서 작동하며, member tenant-net에 연결된 인터페이스를 추가하는 데 사용됩니다. 후자는 create lb flow에서 작동하

며, vip-net에 연결된 인터페이스를 추가하는 데 사용됩니다. 물론, member와 VIP가 동일한 네트워크에 속한 경우에는 amphora에 새로운 인터페이스를 추가할 필요가 없습니다.



Member를 추가한 후 Amphora의 네트워크 상태를 다시 확인합니다

```
root@amphora-cd444019-ce8f-4f89-be6b-0edf76f41b77:~# ip netns exec amphora-
haproxy ifconfig
eth1      Link encap:Ethernet  HWaddr fa:16:3e:f4:69:4b
          inet addr:172.16.1.3  Bcast:172.16.1.255  Mask:255.255.255.0
          inet6 addr: fe80::f816:3eff:fef4:694b/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1450  Metric:1
          RX packets:12705 errors:0 dropped:0 overruns:0 frame:0
          TX packets:613211 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:762300 (762.3 KB)  TX bytes:36792968 (36.7 MB)

eth1:0    Link encap:Ethernet  HWaddr fa:16:3e:f4:69:4b
          inet addr:172.16.1.10  Bcast:172.16.1.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1450  Metric:1

eth2      Link encap:Ethernet  HWaddr fa:16:3e:18:23:7a
          inet addr:192.168.1.3  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::f816:3eff:fe18:237a/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1450  Metric:1
          RX packets:8 errors:2 dropped:0 overruns:0 frame:2
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:2156 (2.1 KB)  TX bytes:808 (808.0 B)
```

구성 파일은 다음과 같습니다

```
# Generated by Octavia agent
auto eth2
```

```
iface eth2 inet static
address 192.168.1.3
broadcast 192.168.1.255
netmask 255.255.255.0
mtu 1450
post-up /sbin/iptables -t nat -A POSTROUTING -p udp -o eth2 -j MASQUERADE
post-down /sbin/iptables -t nat -D POSTROUTING -p udp -o eth2 -j MASQUERADE
```

ListenersUpdate

최종적으로 haproxy 구성 변경은 Task:ListenersUpdate에 의해 완료됩니다

```
# Configuration for loadbalancer 01197be7-98d5-440d-a846-cd70f52dc503
global
    daemon
    user nobody
    log /dev/log local0
    log /dev/log local1 notice
    stats socket /var/lib/octavia/1385d3c4-615e-4a92-aea1-c4fa51a75557.sock
mode 0666 level user
    maxconn 1000000

defaults
    log global
    retries 3
    option redispatch

peers 1385d3c4615e4a92aea1c4fa51a75557_peers
    peer l_Ustq0qE-h-_Q1dlXLXBAiWR8U 172.16.1.7:1025
    peer O08zAgUhIv9TEXhyYZf2iHdxOkA 172.16.1.3:1025

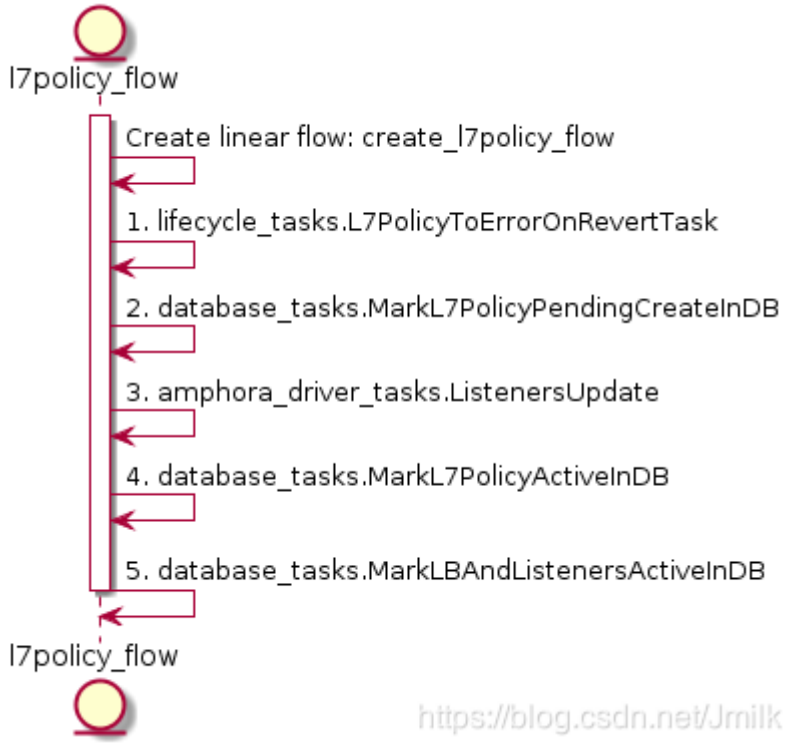

frontend 1385d3c4-615e-4a92-aea1-c4fa51a75557
    option httplog
    maxconn 1000000
    bind 172.16.1.10:8080
    mode http
    default_backend 8196f752-a367-4fb4-9194-37c7eab95714
    timeout client 50000

backend 8196f752-a367-4fb4-9194-37c7eab95714
    mode http
    balance roundrobin
    fullconn 1000000
    option allbackups
    timeout connect 5000
    timeout server 50000
    server b6e464fd-dd1e-4775-90f2-4231444a0bbe 192.168.1.14:80 weight 1
```

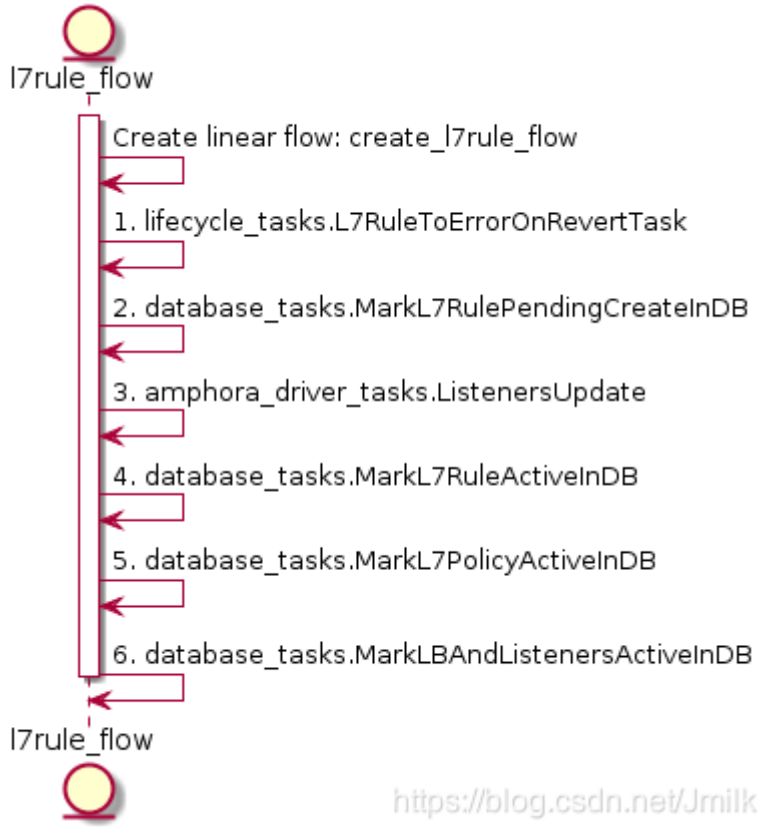
실제로, member를 추가하는 것은 `backend(default pool)`에서 `server <member_id> 192.168.1.14:80 weight 1` 항목을 추가하는 것으로, 해당 클라우드 호스트가 기본 풀의 일부가 되었음을 나타냅니다

L7policy, L7rule 및 Health Monitor의 생성 프로세스 분석

L7policy 객체의 의미는 전송 동작 유형(예: pool로 전송, URL로 전송 또는 전송 거부)을 설명하는 데 사용되며, L7rule의 컨테이너로서 Listener에 종속됩니다.

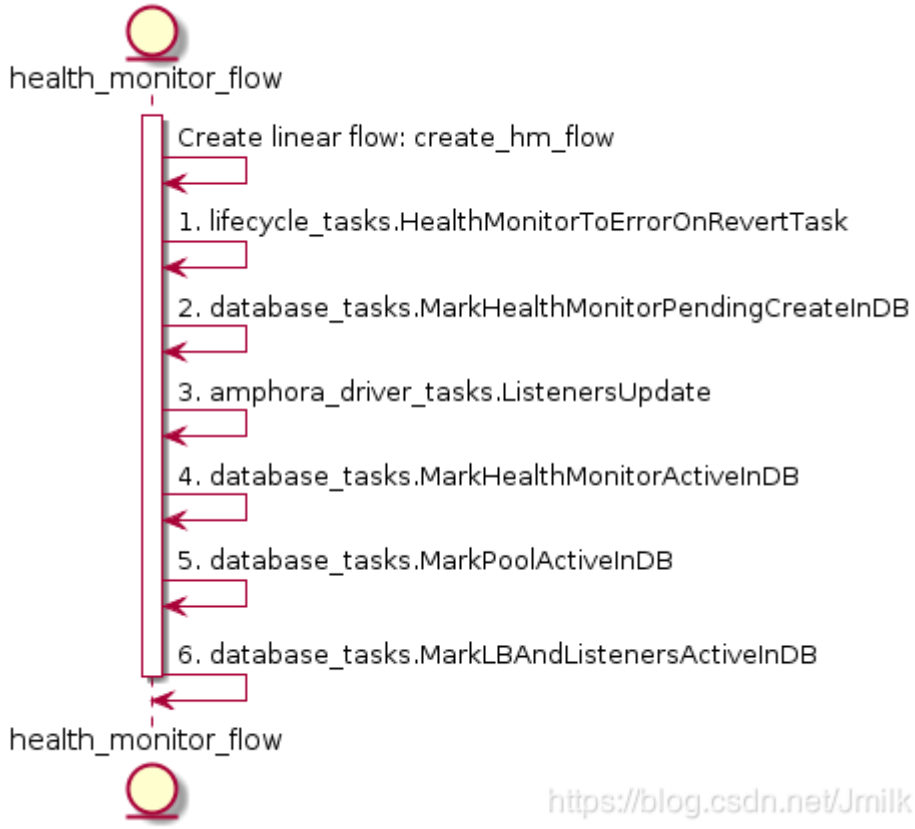


L7Rule 객체의 의미는 데이터 전송의 매칭 필드를 나타내며, 전송 라우팅 관계를 설명하고 L7policy에 종



속됩니다.

Health Monitor 객체는 Pool 내 Member의 상태를 모니터링하는 데 사용되며, 본질적으로는 건강 검사의 규칙을 설명하는 데이터베이스 기록이며 Pool에 종속됩니다.



왜 이 세 가지(L7policy, L7rule, Health Monitor)를 함께 분석하는가? 위의 세 가지 UML 다이어그램에서 볼 수 있듯이, L7policy, L7rule, Health Monitor 및 Pool 생성 과정은 실제로 매우 유사합니다. 핵심은 모두 **TASK:ListenersUpdate**가 haproxy 구성 파일을 업데이트하는 것에 있습니다. 따라서 우리는 주로 몇 가지 예를 통해 haproxy 구성 파일의 변경 규칙을 관찰하면 됩니다.

예시 1. 기본 플로 전송

```
$ openstack loadbalancer healthmonitor create --name healthmonitor1 --type
PING --delay 5 --timeout 10 --max-retries 3 8196f752-
a367-4fb4-9194-37c7eab95714

$ openstack loadbalancer l7policy create --name l7p1 --action
REDIRECT_TO_POOL --redirect-pool 8196f752-a367-4fb4-9194-37c7eab95714
1385d3c4-615e-4a92-aea1-c4fa51a75557

$ openstack loadbalancer l7rule create --type HOST_NAME --compare-type
STARTS_WITH --value "server" 87593985-e02f-4880-b80f-22a4095c05a7
```

haproxy.cfg

```
# Configuration for loadbalancer 01197be7-98d5-440d-a846-cd70f52dc503
global
    daemon
    user nobody
    log /dev/log local0
    log /dev/log local1 notice
    stats socket /var/lib/octavia/1385d3c4-615e-4a92-aea1-c4fa51a75557.sock
mode 0666 level user
    maxconn 1000000
    external-check

defaults
    log global
    retries 3
    option redispatch

peers 1385d3c4615e4a92aea1c4fa51a75557_peers
    peer l_Ustq0qE-h-_Q1dlXLXBAiWR8U 172.16.1.7:1025
    peer O08zAgUhIv9TEXhyYZf2iHdxOkA 172.16.1.3:1025


frontend 1385d3c4-615e-4a92-aea1-c4fa51a75557
    option httplog
    maxconn 1000000
    # frontend http://172.16.1.10:8080
    bind 172.16.1.10:8080
    mode http
        # ACL 전송규칙
        acl 8d9b8b1e-83d7-44ca-a5b4-0103d5f90cb9 req.hdr(host) -i -m beg
server
    # if ACL 8d9b8b1e-83d7-44ca-a5b4-0103d5f90cb9 충족되면 backend 8196f752-
a367-4fb4-9194-37c7eab95714
    use_backend 8196f752-a367-4fb4-9194-37c7eab95714 if 8d9b8b1e-83d7-44ca-
```

```
a5b4-0103d5f90cb9
   # 어떤 ACL 규칙도 일치하지 않으면 backend 8196f752-a367-4fb4-9194-37c7eab95714
로 전송
    default_backend 8196f752-a367-4fb4-9194-37c7eab95714
    timeout client 50000

backend 8196f752-a367-4fb4-9194-37c7eab95714
    # http 프로토콜 사용
    mode http
    # RR 알고리즘 사용
    balance roundrobin
    timeout check 10s
    option external-check
    # ping-wrapper.sh 스크립트를 사용하여 server 상태 모니터링
    external-check command /var/lib/octavia/ping-wrapper.sh
    fullconn 1000000
    option allbackups
    timeout connect 5000
    timeout server 50000
   # 뒷단의 실제 서버(real server)서비스포트 80, 모니터링규칙 inter 5s fall 3 rise
3
    server b6e464fd-dd1e-4775-90f2-4231444a0bbe 192.168.1.14:80 weight 1
check inter 5s fall 3 rise 3
```

Health Check Script (ping-wrapper.sh)는 우리가 설정한 대로 PING 방식을 사용합니다.

```
#!/bin/bash
if [[ $HAPROXY_SERVER_ADDR =~ ":" ]]; then
    /bin/ping6 -q -n -w 1 -c 1 $HAPROXY_SERVER_ADDR > /dev/null 2>&1
else
    /bin/ping -q -n -w 1 -c 1 $HAPROXY_SERVER_ADDR > /dev/null 2>&1
fi
```

예시 2. 공유 풀로 전송

```
$ openstack loadbalancer healthmonitor create --name healthmonitor1 --type
PING --delay 5 --timeout 10 --max-retries 3 822f78c3-ea2c-4770-bef0-
e97f1ac2eba8

$ openstack loadbalancer l7policy create --name l7p1 --action
REDIRECT_TO_POOL --redirect-pool 822f78c3-ea2c-4770-bef0-e97f1ac2eba8
1385d3c4-615e-4a92-aea1-c4fa51a75557

$ openstack loadbalancer l7rule create --type HOST_NAME --compare-type
STARTS_WITH --value "server" fb90a3b5-c97c-4d99-973e-118840a7a236
```

haproxy.cfg

```
# Configuration for loadbalancer 01197be7-98d5-440d-a846-cd70f52dc503
global
    daemon
    user nobody
    log /dev/log local0
    log /dev/log local1 notice
    stats socket /var/lib/octavia/1385d3c4-615e-4a92-aea1-c4fa51a75557.sock
mode 0666 level user
    maxconn 1000000
    external-check

defaults
    log global
    retries 3
    option redispatch

peers 1385d3c4615e4a92aea1c4fa51a75557_peers
    peer l_Ustq0qE-h-_Q1dlXLXBAiWR8U 172.16.1.7:1025
    peer O08zAgUhIv9TEXhyYZf2iHdxOkA 172.16.1.3:1025


frontend 1385d3c4-615e-4a92-aea1-c4fa51a75557
    option httplog
    maxconn 1000000
    bind 172.16.1.10:8080
    mode http
        acl 8d9b8b1e-83d7-44ca-a5b4-0103d5f90cb9 req.hdr(host) -i -m beg
server
    use_backend 8196f752-a367-4fb4-9194-37c7eab95714 if 8d9b8b1e-83d7-44ca-
a5b4-0103d5f90cb9
        acl c76f36bc-92c0-4f48-8d57-a13e3b1f09e1 req.hdr(host) -i -m beg
server
    use_backend 822f78c3-ea2c-4770-bef0-e97f1ac2eba8 if
c76f36bc-92c0-4f48-8d57-a13e3b1f09e1
    default_backend 8196f752-a367-4fb4-9194-37c7eab95714
    timeout client 50000

backend 8196f752-a367-4fb4-9194-37c7eab95714
    mode http
    balance roundrobin
    timeout check 10s
    option external-check
    external-check command /var/lib/octavia/ping-wrapper.sh
    fullconn 1000000
    option allbackups
    timeout connect 5000
    timeout server 50000
    server b6e464fd-dd1e-4775-90f2-4231444a0bbe 192.168.1.14:80 weight 1
check inter 5s fall 3 rise 3
```

```
backend 822f78c3-ea2c-4770-bef0-e97f1ac2eba8
    mode http
    balance roundrobin
    timeout check 10s
    option external-check
    external-check command /var/lib/octavia/ping-wrapper.sh
    fullconn 1000000
    option allbackups
    timeout connect 5000
    timeout server 50000
    server 7da6f176-36c6-479a-9d86-c892ecca6ae5 192.168.1.6:80 weight 1
check inter 5s fall 3 rise 3
```

볼 수 있듯이, listener에 공유 풀을 추가한 후 `shared pool 822f78c3-ea2c-4770-bef0-e97f1ac2eba8`에 해당하는 `backend section`이 하나 더 추가됩니다.

Amphora의 보안 통신 구현

자체 CA를 통해 구현된 SSL 통신

계속해서 `amphora-agent`와 Octavia Controller Worker가 어떻게 안전한 통신을 구축하는지 살펴보겠습니다. 먼저 Octavia가 왜 자체 CA 인증서를 필요로 하는지 생각해 봅시다.

참고: 실제 운영 환경에서는 클라이언트 인증서를 발급하는 CA와 서버 인증서를 발급하는 CA가 달라야 합니다. 그래야 해킹된 amphora의 서버 인증서를 사용해 다른 인스턴스를 제어하지 못하게 할 수 있습니다.

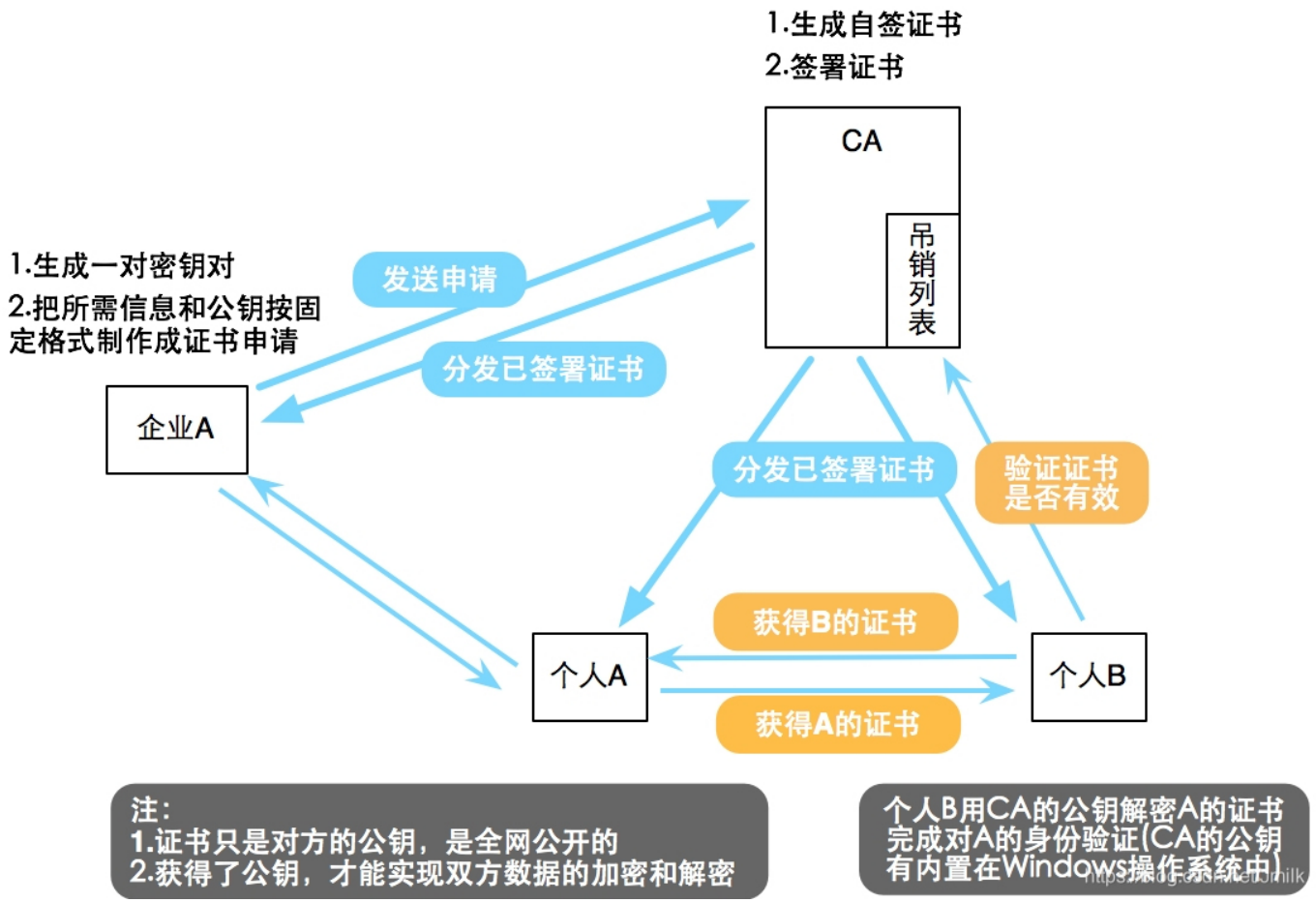
Octavia와 Dashboard가 동일한 인증서를 사용한다면 OpenStack 관리/API 네트워크를 공개하는 것과 다름없습니다. 간단히 말해서, Octavia가 자체 CA 인증서를 사용하는 데에는 두 가지 중요한 이유가 있습니다:

1. `amphora-agent`는 인증 메커니즘이 없으므로, API의 안전성을 보장하기 위해 인증서가 필요합니다.
2. 악의적인 사용자가 amphora를 '좀비 서버'로 이용해 OpenStack 내부 네트워크를 공격하는 것을 방지합니다.

Octavia는 또한 OpenSSL을 사용하여 CA 센터를 생성하는 자동화 스크립트를 제공합니다. 다음 명령어를 실행하면 완료됩니다.

```
$ source /opt/rocky/octavia/bin/create_certificates.sh /etc/octavia/certs/
/opt/rocky/octavia/etc/certificates/openssl.cnf
```

CA 센터에 대해 좀 더 설명하자면, 이른바 CA는 외적으로 다양한 유형의 인증서를 포함한 디렉터리로 나타납니다. 내적으로는 인증서 발급 및 관리 서비스를 제공하는 제3자 신뢰 기관을 의미하며, 비대칭 암호화 시스템에서 중간자 공격 문제를 효과적으로 해결할 수 있습니다. 자세한 내용은 “OpenSSL을 사용해 자체 CA를 구축하고 인증서를 발급하는 방법 ” 에서 확인할 수 있으며, 여기서는 더 이상 자세히 설명하지 않겠습니다.



Octavia가 자체적으로 구축한 CA 센터

```
$ ll /etc/octavia/certs/
total 44
-rw-r--r-- 1 stack stack 1294 Oct 26 12:51 ca_01.pem
-rw-r--r-- 1 stack stack  989 Oct 26 12:51 client.csr
-rw-r--r-- 1 stack stack 1708 Oct 26 12:51 client.key
-rw-r--r-- 1 stack stack 4405 Oct 26 12:51 client-.pem
-rw-r--r-- 1 stack stack 6113 Oct 26 12:51 client.pem
-rw-r--r-- 1 stack stack   71 Oct 26 12:51 index.txt
-rw-r--r-- 1 stack stack   21 Oct 26 12:51 index.txt.attr
-rw-r--r-- 1 stack stack    0 Oct 26 12:51 index.txt.old
drwxr-xr-x 2 stack stack   20 Oct 26 12:51 newcerts
drwx------ 2 stack stack   23 Oct 26 12:51 private
-rw-r--r-- 1 stack stack    3 Oct 26 12:51 serial
-rw-r--r-- 1 stack stack    3 Oct 26 12:51 serial.old
```

- `newcerts` dir: CA에서 서명(발급)한 디지털 인증서를 저장
- `private` dir: CA의 개인 키를 저장
- `serial` file: 인증서 일련번호를 저장(e.g. 01), 새로운 인증서를 만들 때마다 일련번호가 자동으로 1씩 증가
- `index.txt` file: 인증서 정보를 저장
- `ca_01.pem` PEM file: CA 인증서 파일
- `client.csr` file: 서버의 CSR(인증서 서명 요청) 파일

- client.key file: 서버의 개인 키 파일
- client-.pem: PEM 인코딩된 서버 인증서 파일
- client.pem: client-.pem과 client.key를 결합한 파일

아래는 CA 인증과 관련된 설정 항목들을 나열한 것입니다.

```
# create new amphora flow에 적용되는 **TASK:GenerateServerPEMTask**는 amphora 서버 인증서를 생성합니다.
[certificates]
ca_private_key_passphrase = foobar
ca_private_key = /etc/octavia/certs/private/cakey.pem
ca_certificate = /etc/octavia/certs/ca_01.pem

# AmphoraAPIClient에 적용되며, client.pem(서버 인증서와 서버 개인 키 포함)과 CA 인증서(공개 키)를 사용하여 amphora-agent에 SSL 통신을 요청합니다.
[haproxy_amphora]
server_ca = /etc/octavia/certs/ca_01.pem
client_cert = /etc/octavia/certs/client.pem

# Task:CertComputeCreate에 적용되며, CA 인증서의 경로를 지정합니다.
[controller_worker]
client_ca = /etc/octavia/certs/ca_01.pem
```

먼저 SSL 통신을 설정하는 과정을 간단히 정리한 후, 구체적인 구현을 자세히 살펴보겠습니다:

1. Amphora를 생성할 때 CA 센터에 서버 인증서 서명을 요청하며, amphora-agent 서비스 프로세스가 시작될 때 Flask 앱이 이 인증서를 로드하여 HTTPS 프로토콜을 활성화합니다.
2. AmphoraAPIClient가 처음으로 amphora-agent에 요청을 보낼 때, CA 인증서를 사용하여 서버 인증서를 검증하고, 검증이 완료되면 서버의 공개 키를 받아 SSL 통신을 설정합니다.

Amphora Agent 인증서 로딩

먼저 amphora에 대한 인증서 생성 구현을 살펴봅니다.

```
# file: /opt/rocky/octavia/octavia/controller/worker/tasks/cert_task.py

class GenerateServerPEMTask(BaseCertTask):
    """Create the server certs for the agent comm

    Use the amphora_id for the CN
    """

    def execute(self, amphora_id):
        cert = self.cert_generator.generate_cert_key_pair(
            cn=amphora_id,
            validity=CERT_VALIDITY)

        return cert.certificate + cert.private_key
```

Octavia Certificates는 `local_cert_generator`(기본값)와 `anchor_cert_generator` 두 가지 인증서 생성기를 제공하며, `[certificates] cert_generator` 설정을 통해 선택할 수 있습니다.

```
# file: /opt/rocky/octavia/octavia/certificates/generator/local.py

    @classmethod
    def generate_cert_key_pair(cls, cn, validity, bit_length=2048,
                               passphrase=None, **kwargs):
        pk = cls._generate_private_key(bit_length, passphrase)
        csr = cls._generate_csr(cn, pk, passphrase)
        cert = cls.sign_cert(csr, validity, **kwargs)
        cert_object = local_common.LocalCert(
            certificate=cert,
            private_key=pk,
            private_key_passphrase=passphrase
        )
        return cert_object
```

위의 **LocalCertGenerator.generate_cert_key_pair**의 의미는 다음과 같습니다:

1. Amphora 개인 키 생성
2. Amphora 인증서 서명 요청(CSR) 생성
3. CA 센터에 Amphora 서버 인증서 서명을 요청

이는 일반적인 인증서 생성 절차에 속하며, **create_certificates.sh** 스크립트와의 차이점은 Octavia Certificates가 **cryptography** 라이브러리를 사용하여 구현되었다는 점입니다.

TASK:GenerateServerPEMTask는 최종적으로 Amphora의 개인 키와 인증서를 반환하며, 이후 **TASK:CertComputeCreate**가 Nova의 userdata와 Nova Store metadata on a configuration drive 메커니즘을 통해 이러한 파일들을 Amphora 인스턴스에 주입합니다. Amphora에 로그인하면 이러한 파일을 확인할 수 있으며, 파일 경로는 설정 파일에 기록되어 있습니다.

```
# file: /etc/octavia/amphora-agent.conf

[amphora_agent]
agent_server_ca = /etc/octavia/certs/client_ca.pem
agent_server_cert = /etc/octavia/certs/server.pem
```

Gunicorn HTTP 서버가 시작될 때 인증서 파일을 로드하며, 인증서를 로드하는 옵션은 다음과 같습니다.

```
options = {
        'bind': bind_ip_port,
        'workers': 1,
        'timeout': CONF.amphora_agent.agent_request_read_timeout,
        'certfile': CONF.amphora_agent.agent_server_cert,
        'ca_certs': CONF.amphora_agent.agent_server_ca,
```

```
        'cert_reqs': True,
        'preload_app': True,
        'accesslog': '/var/log/amphora-agent.log',
        'errorlog': '/var/log/amphora-agent.log',
        'loglevel': 'debug',
    }
```

- `key:certfile`: Amphora-agent의 개인 키와 인증서를 업로드.
- `key:ca_certs`: Amphora-agent의 CA 인증서를 업로드.

AmphoraAPIClient가 인증서 요청을 전송

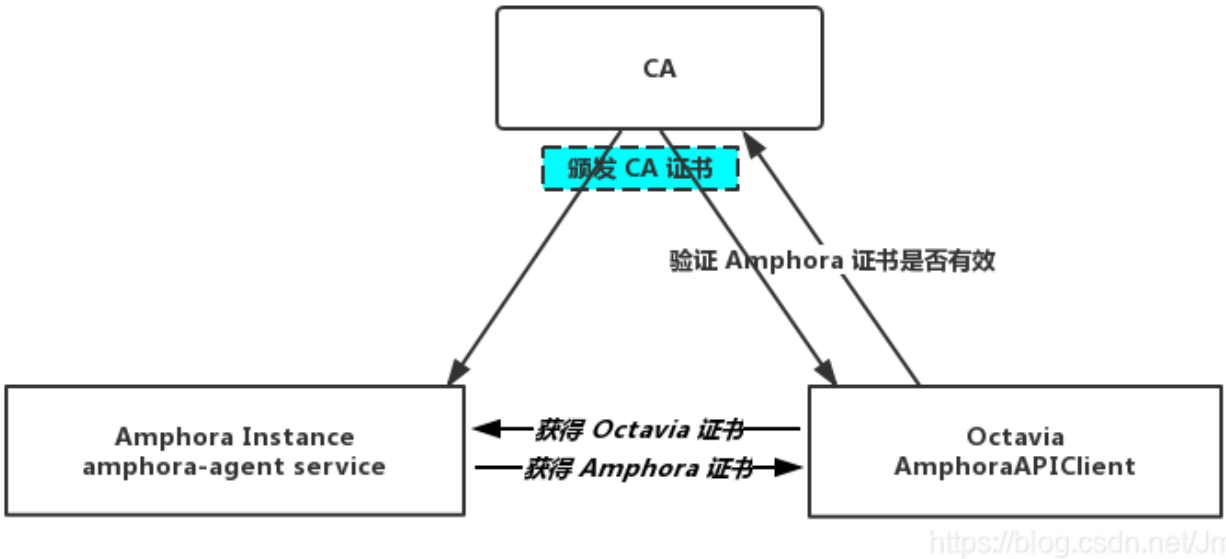
```
class AmphoraAPIClient(object):
    def __init__(self):
        super(AmphoraAPIClient, self).__init__()
        ...
        self.session = requests.Session()
        self.session.cert = CONF.haproxy_amphora.client_cert
        self.ssl_adapter = CustomHostNameCheckingAdapter()
        self.session.mount('https://', self.ssl_adapter)
        ...

    def request(self, method, amp, path='/', timeout_dict=None, **kwargs):
        ...
        LOG.debug("request url %s", path)
        _request = getattr(self.session, method.lower())
        _url = self._base_url(amp.lb_network_ip) + path
        LOG.debug("request url %s", _url)
        reqargs = {
            'verify': CONF.haproxy_amphora.server_ca,
            'url': _url,
            'timeout': (req_conn_timeout, req_read_timeout), }
        reqargs.update(kwargs)
        headers = reqargs.setdefault('headers', {})
        ...
```

위의 코드는 **requests** 라이브러리를 사용하여 HTTPS 요청을 실행하는 일반적인 구현입니다:

1. `self.session.cert`: Octavia(AmphoraAPIClient)의 개인 키와 인증서를 업로드.
2. `reqargs = {'verify': CONF.haproxy_amphora.server_ca, …}`: CA 인증서를 첨부한 요청을 전송.

마지막으로, Octavia가 자체 CA를 통해 Amphora와 Octavia Controller Worker 간에 HTTPS 통신을 구현하는 과정을 간단히 정리하면 다음과 같습니다: AmphoraAPIClient가 처음으로 amphora-agent에 요청을 보낼 때, AmphoraAPIClient는 먼저 amphora-agent의 인증서를 다운로드하고, 이를 로컬의 CA 인증서와 비교하여 검증합니다. amphora-agent 인증서도 CA 개인 키로 암호화되어 있기 때문에 CA 인증서로 이를 복호화할 수 있습니다. 검증이 완료되면 amphora-agent의 공개 키를 얻고, amphora-agent가 업로드한 개인 키와 비교하여 SSL 통신을 설정합니다.



Amphora의 장애 조치(페일오버) 구현

Health Manager

Health Manager - 이 하위 구성 요소는 개별 amphora를 모니터링하여 정상적으로 작동하고 건강한 상태인지 확인합니다. 또한 amphora가 예상치 못하게 실패할 경우 장애 조치(페일오버) 이벤트를 처리합니다.

간단히 말해, Health Manager는 각 amphora의 상태를 모니터링하고, amphora에 장애가 발생하면 장애 조치(페일오버) 프로세스를 시작하여 로드 밸런서의 고가용성을 보장합니다.

따라서 Health Manager Service를 이해하려면 먼저 이 서비스가 amphora의 상태를 어떻게 모니터링하는지 파악한 후, 장애 조치 프로세스의 세부 사항을 알아야 합니다.

Amphora의 상태 모니터링

먼저 프로그램의 진입점(**octavia/cmd/health_manager.py**)에서 시작하여, **octavia-health-manager** 서비스를 시작하면 **UDPStatusGetter.check()** 및 **HealthManager.health_check()** 두 가지 메서드가 로드됩니다. 먼저 **UDPStatusGetter.check()**의 구현을 살펴보겠습니다.

```
# file: /opt/rocky/octavia/octavia/amphorae/drivers/health/heartbeat_udp.py


class UDPStatusGetter(object):
    """This class defines methods that will gather heatbeats

    The heartbeats are transmitted via UDP and this class will bind to a
port
    and absorb them
    """
```

```
    def __init__(self):
        self.key = cfg.CONF.health_manager.heartbeat_key
        self.ip = cfg.CONF.health_manager.bind_ip
        self.port = cfg.CONF.health_manager.bind_port
        self.sockaddr = None
        LOG.info('attempting to listen on %(ip)s port %(port)s',
                 {'ip': self.ip, 'port': self.port})
        self.sock = None
        self.update(self.key, self.ip, self.port)

        self.executor = futures.ProcessPoolExecutor(
            max_workers=cfg.CONF.health_manager.status_update_threads)
        self.repo = repositories.Repositories().amphorahealth

    def update(self, key, ip, port):
        """Update the running config for the udp socket server

        :param key: The hmac key used to verify the UDP packets. String
        :param ip: The ip address the UDP server will read from
        :param port: The port the UDP server will read from
        :return: None
        """
        self.key = key
        for addrinfo in socket.getaddrinfo(ip, port, 0, socket.SOCK_DGRAM):
            ai_family = addrinfo[0]
            self.sockaddr = addrinfo[4]
            if self.sock is not None:
                self.sock.close()
            self.sock = socket.socket(ai_family, socket.SOCK_DGRAM)
            self.sock.settimeout(1)
            self.sock.bind(self.sockaddr)
            if cfg.CONF.health_manager.sock_rlimit > 0:
                rlimit = cfg.CONF.health_manager.sock_rlimit
                LOG.info("setting sock rlimit to %s", rlimit)
                self.sock.setsockopt(socket.SOL_SOCKET, socket.SO_RCVBUF,
                                     rlimit)
            break  # just used the first addr getaddrinfo finds
        if self.sock is None:
            raise exceptions.NetworkConfig("unable to find suitable socket")
```

Class:UDPStatusGetter는 **octavia-health-manager** 서비스 내에서 **amphora**에서 전송된 heartbeats(심장 박동 패킷)를 수신하는 역할을 합니다. 이 heartbeats 데이터를 준비하고 이를 데이터베이스에 영구적으로 저장합니다. **init()** 메서드를 보면, **amphora**와 **octavia-health-manager** 서비스 간의 통신은 UDP 소켓으로 구현되었으며, 소켓은 **(CONF.health_manager.bind_ip, CONF.health_manager.bind_port)**로 설정됩니다.

참고: 여기서 **amphora**와 **octavia-health-manager** 서비스 간의 네트워크 토폴로지 세부 사항을 강조해야 합니다.

- Octavia를 배포할 때, **ext-net**을 직접 **octavia**의 “lb-mgmt-net”으로 사용하는 경우, **CONF.health_manager.bind_ip**는 물리 호스트의 IP 주소가 되어야 하며, **amphora**와 **octavia-**

health-manager 서비스는 **OpenStack Management Network**를 통해 직접 통신합니다. 그러나 이 방식은 **amphora**가 **ext-net**의 고정 IP를 차지하게 되므로, 실제 운영 환경에서는 권장되지 않습니다.
- Octavia 배포 시, 별도로 생성된 **tenant network**를 **lb-mgmt-net**으로 사용하는 경우, **CONF.health_manager.bind_ip**는 **lb-mgmt-net** IP 풀 내의 주소여야 합니다. 이 경우 **lb-mgmt-net**과 **OpenStack Management Network** 간의 통신 문제를 해결해야 합니다. **devstack**에서는 **lb-mgmt-net**의 포트 하나를 **ex-int**에 연결하여, **lb-mgmt-net** 내의 **amphora**가 이 포트를 통해 물리 호스트에서 실행 중인 **octavia-health-manager** 서비스와 통신할 수 있도록 설정합니다. 실제 운영 환경에서는 현장 네트워크 환경에 맞추어 네트워크 관리자가 이를 구성해야 합니다.

Devstack에서 로컬 네트워크를 연결하는 명령어

```
$ neutron port-create --name octavia-health-manager-standalone-listen-port \
  --security-group <lb-health-mgr-sec-grp> \
  --device-owner Octavia:health-mgr \
  --binding:host_id=<hostname> lb-mgmt-net \
  --tenant-id <octavia service>

$ ovs-vsctl --may-exist add-port br-int o-hm0 \
  -- set Interface o-hm0 type=internal \
  -- set Interface o-hm0 external-ids:iface-status=active \
  -- set Interface o-hm0 external-ids:attached-mac=<Health Manager Listen
Port MAC> \
  -- set Interface o-hm0 external-ids:iface-id=<Health Manager Listen Port
ID>
# /etc/octavia/dhcp/dhclient.conf
request subnet-mask,broadcast-address,interface-mtu;
do-forward-updates false;

$ ip link set dev o-hm0 address <Health Manager Listen Port MAC>
$ dhclient -v o-hm0 -cf /etc/octavia/dhcp/dhclient.conf


o-hm0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1450
        inet 192.168.0.4  netmask 255.255.255.0  broadcast 192.168.0.255
        inet6 fe80::f816:3eff:fef0:b9ee  prefixlen 64  scopeid 0x20<link>
        ether fa:16:3e:f0:b9:ee  txqueuelen 1000  (Ethernet)
        RX packets 1240893  bytes 278415460 (265.5 MiB)
        RX errors 0  dropped 45  overruns 0  frame 0
        TX packets 417078  bytes 75842972 (72.3 MiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

다시 주제로 돌아가서, **UDPStatusGetter.check()**의 구현은

```
    def check(self):
        try:
            obj, srcaddr = self.dorecv()
```

```
        except socket.timeout:
            # Pass here as this is an expected cycling of the listen socket
            pass
        except exceptions.InvalidHMACException:
            # Pass here as the packet was dropped and logged already
            pass
        except Exception as e:
            LOG.warning('Health Manager experienced an exception processing
a'
                        'heartbeat packet. Ignoring this packet. '
                        'Exception: %s', e)
        else:
            self.executor.submit(update_health, obj, srcaddr)
            self.executor.submit(update_stats, obj, srcaddr)
```

- `self.dorecv()`을 호출하여 데이터를 수신
- `self.executor.submit(update_health, obj, srcaddr)`을 호출하여 health를 **amphora_health** 테이블에 영구 저장
- `self.executor.submit(update_stats, obj, srcaddr)`을 호출하여 stats를 **listener_statistics** 테이블에 영구 저장

계속해서 **amphora**가 어떻게 heartbeats를 전송하는지 살펴보겠습니다.

```
# file: /opt/rocky/octavia/octavia/cmd/agent.py

def main():
    # comment out to improve logging
    service.prepare_service(sys.argv)

    gmr.TextGuruMeditation.setup_autorun(version)

    health_sender_proc = multiproc.Process(name='HM_sender',
                                           target=health_daemon.run_sender,
                                           args=(HM_SENDER_CMD_QUEUE,))
    health_sender_proc.daemon = True
    health_sender_proc.start()

    # Initiate server class
    server_instance = server.Server()

    bind_ip_port = utils.ip_port_str(CONF.haproxy_amphora.bind_host,
                                     CONF.haproxy_amphora.bind_port)
    options = {
        'bind': bind_ip_port,
        'workers': 1,
        'timeout': CONF.amphora_agent.agent_request_read_timeout,
        'certfile': CONF.amphora_agent.agent_server_cert,
        'ca_certs': CONF.amphora_agent.agent_server_ca,
        'cert_reqs': True,
        'preload_app': True,
```

```
        'accesslog': '/var/log/amphora-agent.log',
        'errorlog': '/var/log/amphora-agent.log',
        'loglevel': 'debug',
    }
    AmphoraAgent(server_instance.app, options).run()
```

amphora-agent 서비스 프로세스가 시작될 때, **health_daemon.run_sender**가 로드되며, 이것이 **amphora**가 **octavia-health-manager** 서비스에 heartbeats패킷을 전송하는 구현입니다.

```
# file:
/opt/rocky/octavia/octavia/amphorae/backends/health_daemon/health_daemon.py

def run_sender(cmd_queue):
    LOG.info('Health Manager Sender starting.')
    sender = health_sender.UDPStatusSender()

    keepalived_cfg_path = util.keepalived_cfg_path()
    keepalived_pid_path = util.keepalived_pid_path()

    while True:

        try:
            # If the keepalived config file is present check
            # that it is running, otherwise don't send the health
            # heartbeat
            if os.path.isfile(keepalived_cfg_path):
                # Is there a pid file for keepalived?
                with open(keepalived_pid_path, 'r') as pid_file:
                    pid = int(pid_file.readline())
                os.kill(pid, 0)

            message = build_stats_message()
            sender.dosend(message)

        except IOError as e:
            # Missing PID file, skip health heartbeat
            if e.errno == errno.ENOENT:
                LOG.error('Missing keepalived PID file %s, skipping health '
                          'heartbeat.', keepalived_pid_path)
            else:
                LOG.error('Failed to check keepalived and haproxy status due
'
                          'to exception %s, skipping health heartbeat.', e)
        except OSError as e:
            # Keepalived is not running, skip health heartbeat
            if e.errno == errno.ESRCH:
                LOG.error('Keepalived is configured but not running, '
                          'skipping health heartbeat.')
```

```
            else:
                LOG.error('Failed to check keepalived and haproxy status due
'
                          'to exception %s, skipping health heartbeat.', e)
        except Exception as e:
            LOG.error('Failed to check keepalived and haproxy status due to
'
                      'exception %s, skipping health heartbeat.', e)

        try:
            cmd = cmd_queue.get_nowait()
            if cmd == 'reload':
                LOG.info('Reloading configuration')
                CONF.reload_config_files()
            elif cmd == 'shutdown':
                LOG.info('Health Manager Sender shutting down.')
                break
        except queue.Empty:
            pass
        time.sleep(CONF.health_manager.heartbeat_interval)
```

run_sender 함수는 **build_stats_message()**를 호출하여 heartbeats를 구성한 다음, **UDPStatusSender.dosend()**를 호출하여 데이터를 전송합니다. 중요한 점은, **keepalived** 서비스 프로세스가 정상적으로 실행되지 않으면 heartbeats를 전송하지 않는다는 것입니다. 즉, **keepalived**가 비정상인 **amphora**는 장애가 발생한 **amphora**로 처리됩니다. 데이터 전송은 여전히 UDP 소켓을 사용하며, 대상 URL은 **CONF.health_manager.controller_ip_port_list**에 의해 설정됩니다.

```
# file: /etc/octavia/octavia.conf

[health_manager]
bind_port = 5555
bind_ip = 192.168.0.4
controller_ip_port_list = 192.168.0.4:5555
```

간단히 말하면, **octavia-health-manager**와 **amphora-agent**는 주기적인 heartbeats 프로토콜을 구현하여 **amphora**의 건강 상태를 모니터링합니다.

failover(장애 조치)

장애 조치 메커니즘은 **health_manager.HealthManager.health_check()**에 의해 주기적으로 모니터링되고 트리거됩니다.

health_check 메서드는 주기적으로 **amphora_health** 테이블에서 “stale amphora” 기록을 가져오며, 이는 heartbeats 보고가 없어서 장애로 판단된 **amphora**입니다.

```
# file: /opt/rocky/octavia/octavia/db/repositories.py

    def get_stale_amphora(self, session):
```

```
        """Retrieves a stale amphora from the health manager database.

        :param session: A Sql Alchemy database session.
        :returns: [octavia.common.data_model]
        """

        timeout = CONF.health_manager.heartbeat_timeout
        expired_time = datetime.datetime.utcnow() - datetime.timedelta(
            seconds=timeout)

        amp = session.query(self.model_class).with_for_update().filter_by(
            busy=False).filter(
            self.model_class.last_update < expired_time).first()

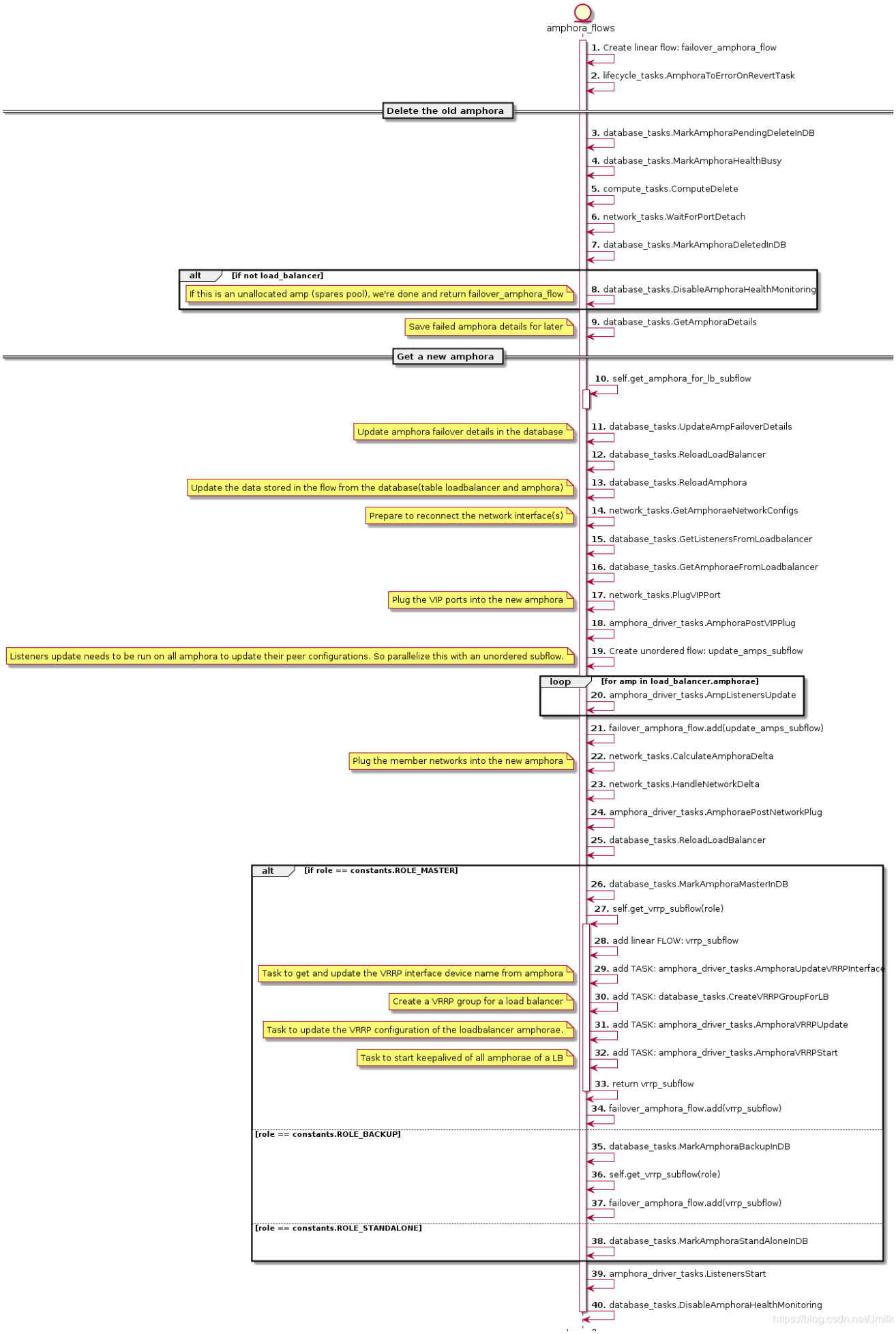
        if amp is None:
            return None

        amp.busy = True

        return amp.to_data_model()
```

만약 **stale amphora**가 존재하고 loadbalancer의 상태가 **PENDING_UPDATE**가 아니면, **failover amphora** 프로세스에 진입하게 됩니다. **failover amphora**의 taskflow는 **self._amphora_flows.get_failover_flow**입니다.

failover의 UML 다이어그램


amphora_flows
1. Create linear flow: failover_amphora_flow
2. lifecycle_tasks.AmphoraToErrorOnRevertTask
Delete the old amphora
3. database_tasks.MarkAmphoraPendingDeleteInDB
4. database_tasks.MarkAmphoraHealthBusy
5. compute_tasks.ComputeDelete
6. network_tasks.WaitForPortDetach
7. database_tasks.MarkAmphoraDeletedInDB
alt [if not load_balancer]
If this is an unallocated amp (spares pool), we're done and return failover_amphora_flow
8. database_tasks.DisableAmphoraHealthMonitoring
Save failed amphora details for later
9. database_tasks.GetAmphoraDetails
Get a new amphora
10. self.get_amphora_for_lb_subflow
Update amphora failover details in the database
11. database_tasks.UpdateAmpFailoverDetails
12. database_tasks.ReloadLoadBalancer
Update the data stored in the flow from the database(table loadbalancer and amphora)
13. database_tasks.ReloadAmphora
Prepare to reconnect the network interface(s)
14. network_tasks.GetAmphoraeNetworkConfigs
15. database_tasks.GetListenersFromLoadbalancer
16. database_tasks.GetAmphoraeFromLoadbalancer
Plug the VIP ports into the new amphora
17. network_tasks.PlugVIPPort
18. amphora_driver_tasks.AmphoraPostVIPPlug
Listeners update needs to be run on all amphora to update their peer configurations. So parallelize this with an unordered subflow.
19. Create unordered flow: update_amps_subflow
loop [for amp in load_balancer.amphorae]
20. amphora_driver_tasks.AmpListenersUpdate
21. failover_amphora_flow.add(update_amps_subflow)
Plug the member networks into the new amphora
22. network_tasks.CalculateAmphoraDelta
23. network_tasks.HandleNetworkDelta
24. amphora_driver_tasks.AmphoraePostNetworkPlug
25. database_tasks.ReloadLoadBalancer
alt [if role == constants.ROLE_MASTER]
26. database_tasks.MarkAmphoraMasterInDB
27. self.get_vrrp_subflow(role)
28. add linear FLOW: vrrp_subflow
Task to get and update the VRRP interface device name from amphora
29. add TASK: amphora_driver_tasks.AmphoraUpdateVRRPInterface
Create a VRRP group for a load balancer
30. add TASK: database_tasks.CreateVRRPGroupForLB
Task to update the VRRP configuration of the loadbalancer amphorae.
31. add TASK: amphora_driver_tasks.AmphoraVRRPUpdate
Task to start keepalived of all amphorae of a LB
32. add TASK: amphora_driver_tasks.AmphoraVRRPStart
33. return vrrp_subflow
34. failover_amphora_flow.add(vrrp_subflow)
[role == constants.ROLE_BACKUP]
35. database_tasks.MarkAmphoraBackupInDB
36. self.get_vrrp_subflow(role)
37. failover_amphora_flow.add(vrrp_subflow)
[role == constants.ROLE_STANDALONE]
38. database_tasks.MarkAmphoraStandAloneInDB
39. amphora_driver_tasks.ListenersStart
40. database_tasks.DisableAmphoraHealthMonitoring
https://blog.csdn.net/Jmilk

분명히, 전체 **failover_flow**는 “ 기존amphora 삭제 ” 와 “ 새로운amphora 획득 ” 의 두 가지 큰 부분으로 나뉩니다. 대부분의 TASK는 이전에 분석했으므로, 아래에서는 각 작업의 기능을 간단히 나열하겠습니다.

- delete old amphora
  - MarkAmphoraPendingDeleteInDB
  - MarkAmphoraHealthBusy
  - ComputeDelete: amphora 삭제
  - WaitForPortDetach: amphora에 연결된 포트(들) 분리
  - MarkAmphoraDeletedInDB

참고: 만약 장애가 발생한 amphora가 **free amphora**라면, 바로 삭제하면 됩니다.

- get a new amphora
  - **get_amphora_for_lb_subflow**: 사용 가능한 **free amphora**를 획득
  - **UpdateAmpFailoverDetails**: **old amphora**의 정보를 **new amphora**로 업데이트 (테이블 amphora)
  - **ReloadLoadBalancer & ReloadAmphora**: 데이터베이스에서 loadbalancer와 amphora의 기록을 가져와 flow에 **stores**로 전달
  - **GetAmphoraeNetworkConfigs & GetListenersFromLoadbalancer & GetAmphoraeFromLoadbalancer**: listener, amphora 및 네트워크 정보를 가져와 flow에 **stores**로 전달, amphora 네트워크 모델 재구성 준비
  - **PlugVIPPort**: amphora에 **keepalived**의 VIP NIC 설정
  - **AmphoraPostVIPPlug**: amphora의 VIP NIC를 네트워크 네임스페이스에 주입
  - **update_amps_subflow\AmpListenersUpdate**: listener 데이터를 기반으로 amphora의 haproxy 구성 파일 업데이트, 이 flow는 unordered 유형으로, 여러 listener가 있으면 병렬로 실행
  - **CalculateAmphoraDelta**: amphora에 필요한 NIC와 이미 존재하는 NIC의 차이를 계산
  - **HandleNetworkDelta**: 위에서 계산된 차이에 따라 NIC 추가 또는 제거
  - **AmphoraePostNetworkPlug**: **member**가 속한 서브넷에 포트를 추가하여 연결
  - **ReloadLoadBalancer**
  - **MarkAmphoraMasterInDB**
  - **AmphoraUpdateVRRPInterface**: amphora의 역할에 따라 테이블 amphora의 VRRP 인터페이스 이름(필드: vrrp_interface)을 가져와 업데이트
  - **CreateVRRPGroupForLB**: amphora의 역할에 따라 loadbalancer의 주/종 amphorae 그룹 업데이트
  - **AmphoraVRRPUpdate**: amphora의 역할에 따라 **keepalived** 서비스의 VRRP 구성 업데이트
  - **AmphoraVRRPStart**: **keepalived** 서비스 시작
  - **ListenersStart**: **haproxy** 서비스 시작
  - **DisableAmphoraHealthMonitoring**: 해당 **amphora_health** 데이터베이스 기록 삭제

간단히 정리하면, **amphora failover**의 흐름은 먼저 장애가 발생한 **old amphora**를 삭제한 후, 사용 가능한 **new amphora**를 확보하여, **old amphora**의 관련 데이터(e.g., 데이터베이스) 및 객체(e.g., 네트워크 모델)를 **new amphora**로 전환하는 것입니다.

주의해야 할 점:

직관적으로는 **old amphora**를 삭제하기 전에 **new amphora**를 부팅하는 것이 더 나아 보일 수 있지만, 이는 복잡한 문제입니다. 만약 타겟 호스트가 자원 제한(anti-affinity 규칙 때문에)을 겪고 있다면, **old amphora** 삭제 후 부팅은 성공할 수 있지

만, 먼저 부팅하려고 하면 실패할 수 있습니다. 이로 인해 비동기 API와의 통신에서 **LB**가 **ERROR** 상태로 끝날 수 있지만, **amphora**는 여전히 살아 있게 됩니다. 향후에는 이 문제를 처리하기 위해 실패 시 재시도 흐름을 고려하거나, 업그레이드 장애 조치를 API와 동기화하도록 수정할 필요가 있습니다. 현재로서는 **spares pool**과 **act/stdby** 구성이 이러한 지연을 완화할 수 있습니다.

비록 장애 조치는 **old amphora**를 삭제한 후 **new amphora**를 얻는 것이지만, 실제 과정은 복잡합니다. 예를 들어, **old amphora**를 성공적으로 삭제한 후 **new amphora**를 생성하려고 할 때 자원 제한으로 인해 실패할 수 있습니다. 또 다른 예로, 비동기 API 호출로 인해 **new amphora**가 성공적으로 생성되었지만, **loadbalancer**의 상태는 이미 **ERROR**로 변경될 수 있습니다. 비동기 API 문제는 앞으로 동기 API를 사용하여 해결할 가능성이 있지만, 현재로서는 **space amphora**를 통해 비동기 생성의 지연 문제를 완화하는 데 더 의존하고 있습니다.

장애 조치 테스트

MASTER amphora의 전원을 끄면 **octavia-health-manager** 서비스가 **amphora failover**를 트리거합니다.

```
Nov 22 11:22:31 control01 octavia-health-manager[29147]: INFO
octavia.controller.healthmanager.health_manager [-] Stale amphora's id is:
cd444019-ce8f-4f89-be6b-0edf76f41b77
Nov 22 11:22:31 control01 octavia-health-manager[29147]: INFO
octavia.controller.healthmanager.health_manager [-] Waiting for 1 failovers
to finish
```

old amphorae

```
 2ddc4ba5-b829-4962-93d8-562de91f1dab |
amphora-4ff5d6fe-854c-4022-8194-0c6801a7478b | ACTIVE | lb-mgmt-
net=192.168.0.23                                                    |
amphora-x64-haproxy      | m1.amphora |
| b237b2b8-afe4-407b-83f2-e2e60361fa07 | amphora-bcff6f9e-4114-4d43-
a403-573f1d97d27e | ACTIVE | lb-mgmt-net=192.168.0.11
| amphora-x64-haproxy      | m1.amphora |
| 46eccf47-be10-47ec-89b2-0de44ea3caec | amphora-cd444019-ce8f-4f89-
be6b-0edf76f41b77 | ACTIVE | lb-mgmt-net=192.168.0.9; web-server-
net=192.168.1.3; lb-vip-net=172.16.1.3  | amphora-x64-haproxy      |
m1.amphora |
| bc043b23-d481-45c4-9410-f7b349987c98 | amphora-a1c1ba86-6f99-4f60-b469-
a4a29d7384c5 | ACTIVE | lb-mgmt-net=192.168.0.3; web-server-
net=192.168.1.12; lb-vip-net=172.16.1.7 | amphora-x64-haproxy      |
m1.amphora |
```

new amphoras

```
| 712ff785-c082-4b53-994c-591d1ec0bf7b | amphora-
caa6ba0f-1a68-4f22-9be9-8521695ac4f4 | ACTIVE | lb-mgmt-net=192.168.0.13
| amphora-x64-haproxy      | m1.amphora |
```

```
| 2ddc4ba5-b829-4962-93d8-562de91f1dab |
amphora-4ff5d6fe-854c-4022-8194-0c6801a7478b | ACTIVE | lb-mgmt-
net=192.168.0.23; web-server-net=192.168.1.4; lb-vip-net=172.16.1.3 |
amphora-x64-haproxy      | m1.amphora |
| b237b2b8-afe4-407b-83f2-e2e60361fa07 | amphora-bcff6f9e-4114-4d43-
a403-573f1d97d27e | ACTIVE | lb-mgmt-net=192.168.0.11
| amphora-x64-haproxy      | m1.amphora |
| bc043b23-d481-45c4-9410-f7b349987c98 | amphora-a1c1ba86-6f99-4f60-b469-
a4a29d7384c5 | ACTIVE | lb-mgmt-net=192.168.0.3; web-server-
net=192.168.1.12; lb-vip-net=172.16.1.7 | amphora-x64-haproxy      |
m1.amphora |
```

new amphora haproxy config

```
# Configuration for loadbalancer 01197be7-98d5-440d-a846-cd70f52dc503
global
    daemon
    user nobody
    log /dev/log local0
    log /dev/log local1 notice
    stats socket /var/lib/octavia/1385d3c4-615e-4a92-aea1-c4fa51a75557.sock
mode 0666 level user
    maxconn 1000000
    external-check

defaults
    log global
    retries 3
    option redispatch

peers 1385d3c4615e4a92aea1c4fa51a75557_peers
    peer 3dVescsRZ-RdRBfYVLW6snVI9gI 172.16.1.3:1025
    peer l_Ustq0qE-h-_Q1dlXLXBAiWR8U 172.16.1.7:1025


frontend 1385d3c4-615e-4a92-aea1-c4fa51a75557
    option httplog
    maxconn 1000000
    bind 172.16.1.10:8080
    mode http
        acl 8d9b8b1e-83d7-44ca-a5b4-0103d5f90cb9 req.hdr(host) -i -m beg
server
    use_backend 8196f752-a367-4fb4-9194-37c7eab95714 if 8d9b8b1e-83d7-44ca-
a5b4-0103d5f90cb9
        acl c76f36bc-92c0-4f48-8d57-a13e3b1f09e1 req.hdr(host) -i -m beg
server
    use_backend 822f78c3-ea2c-4770-bef0-e97f1ac2eba8 if
c76f36bc-92c0-4f48-8d57-a13e3b1f09e1
```

```
    default_backend 8196f752-a367-4fb4-9194-37c7eab95714
    timeout client 50000

backend 8196f752-a367-4fb4-9194-37c7eab95714
    mode http
    balance roundrobin
    timeout check 10s
    option external-check
    external-check command /var/lib/octavia/ping-wrapper.sh
    fullconn 1000000
    option allbackups
    timeout connect 5000
    timeout server 50000
    server b6e464fd-dd1e-4775-90f2-4231444a0bbe 192.168.1.14:80 weight 1
check inter 5s fall 3 rise 3

backend 822f78c3-ea2c-4770-bef0-e97f1ac2eba8
    mode http
    balance roundrobin
    timeout check 10s
    option external-check
    external-check command /var/lib/octavia/ping-wrapper.sh
    fullconn 1000000
    option allbackups
    timeout connect 5000
    timeout server 50000
    server 7da6f176-36c6-479a-9d86-c892ecca6ae5 192.168.1.6:80 weight 1
check inter 5s fall 3 rise 3
```

new amphora keepalived config

```
vrrp_script check_script {
  script /var/lib/octavia/vrrp/check_script.sh
  interval 5
  fall 2
  rise 2
}

vrrp_instance 01197be798d5440da846cd70f52dc503 {
  state MASTER
  interface eth1
  virtual_router_id 1
  priority 100
  nopreempt
  garp_master_refresh 5
  garp_master_refresh_repeat 2
  advert_int 1
  authentication {
    auth_type PASS
```

```
    auth_pass b76d77e
  }

  unicast_src_ip 172.16.1.3
  unicast_peer {
    172.16.1.7
  }

  virtual_ipaddress {
    172.16.1.10
  }
  track_script {
    check_script
  }
}
```

new amphora의 구성 파일과 네트워크 설정이 **old amphora**와 동일하며, 이로 인해 마이그레이션이 성공적으로 완료됩니다.

Neutron-lbaas vs. LBaaS v2 API vs. Octavia vs. Octavia v2 API

사용자들이 가장 많이 묻는 질문 중 LBaaS v2 API와 Octavia v2 API를 혼동하는 것이 단연 1위입니다. 여기서 이 개념들을 간단히 구분해 보겠습니다.

- `Neutron-lbaas`: Neutron의 확장 프로젝트로, 초기 LBaaS 구현을 제공합니다.
- `LBaaS v2 API`: LBaaS API의 v2 버전은 Neutron-lbaas에서 처음 도입되어, 로드밸런서, 리스너, 풀 등의 리소스 객체를 구현했습니다.
- `Octavia`: OpenStack의 독립 프로젝트로, 최신 LBaaS 권장 솔루션입니다.
- `Octavia v2 API`: Octavia API의 v2 버전은 LBaaS v2 API의 확장판으로, Neutron-lbaas의 octavia driver와 호환 가능합니다.

참조링크

- https://www.cnblogs.com/jmilkfan-fanguiju/p/10589749.html
- https://blog.csdn.net/jmilk
- https://blog.51cto.com/u_15301988/3126511

From:
https://atl.kr/dokuwiki/ - **AllThatLinux!**

Permanent link:
https://atl.kr/dokuwiki/doku.php/octavia_lb_%EA%B5%AC%ED%98%84_%EB%B0%8F_%EB%B6%84%EC%84%9D

Last update: **2024/10/10 06:29**