11장.
쿠버네티스 내부 이해

방재근

개요

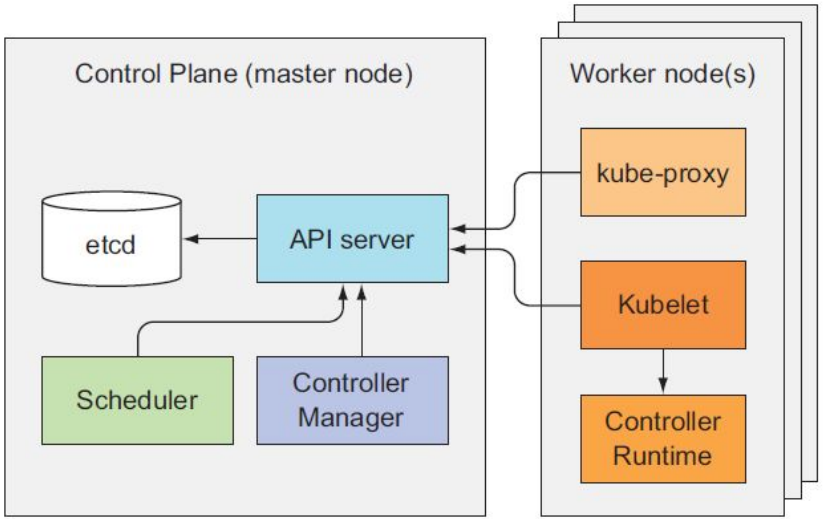
지금까지 쿠버네티스가 제공하는 것, 각 무엇을 하는지에 대해 다루었음.

- 파드, 레플리케이션 컨트롤러, 레플리카셋, 서비스, 볼륨, 디폴로이먼트 등등

해당 Chapter는 쿠버네티스 내부 시스템의 동작 방법에 대해 자세히 다뤄볼것.

- 클러스터 구성 요소
- 구성 요소의 기능 및 동작 방법
- 파드의 동작 과정 (실행 과정, 네트워킹)
- 서비스 동작 방법
- 고가용성 실현 방법

아키텍쳐 이해



- 컨트롤 플레인 (마스터 노드)
  - etcd, API 서버, 스케줄러, 컨트롤 매니저
- 워커 노드
  - Kubelet, Kube-proxy, 컨테이너 런타임
- 애드온 구성 요소
  - DNS 서버, 웹 대시보드, 인그레스 컨트롤러 등

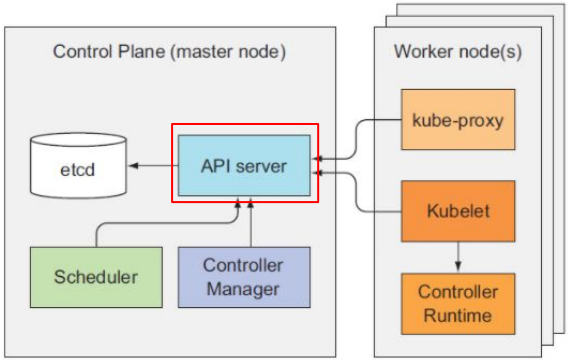
쿠버네티스 구성 요소의 분산 특성

- 컨트롤 플레인 구성 요소 상태 확인
  - API 서버는 컨트롤 플레인 구성 요소 상태를 표시하는 ComponentStatus API를 제공함
  - kubectl 명령을 통해 조회할 수 있음

```
$ kubectl get componentstatuses
NAME                 STATUS    MESSAGE              ERROR
scheduler            Healthy   ok
controller-manager   Healthy   ok
etcd-0               Healthy   {"health": "true"}
```

- 컨트롤 플레인은 여러 서버에 걸쳐 실행될 수 있음
  - **etcd, API 서버**는 여러 인스턴스를 동시에 활성해 병렬로 수행함
  - **스케줄러, 컨트롤러 매니저**는 하나의 인스턴스만 활성화, 나머지는 대기 상태로 있음

쿠버네티스 구성 요소의 분산 특성



- 구성 요소들끼리 오로지 API 서버하고만 통신함
  - 각 구성요소들은 직접 통신하지 않음

- kubelet은 무조건 시스템 구성 요소 (데몬)로 실행 되어야 함
  - 나머지 구성 요소들은 모두 파드로 실행됨
  - 컨트롤 플레인에도 Kubelet이 배포되어, 각 구성 요소를 실행함

```
$ kubectl get po -o custom-columns=POD:metadata.name,NODE:spec.nodeName
➥ --sort-by spec.nodeName -n kube-system
POD                                 NODE
kube-controller-manager-master      master
kube-dns-2334855451-37d9k           master
etcd-master                         master
kube-apiserver-master               master
kube-scheduler-master               master
kube-flannel-ds-tgj9k               node1
kube-proxy-ny3xm                    node1
kube-flannel-ds-0eek8               node2
kube-proxy-sp362                    node2
kube-flannel-ds-r5yf4               node3
kube-proxy-og9ac                    node3
```

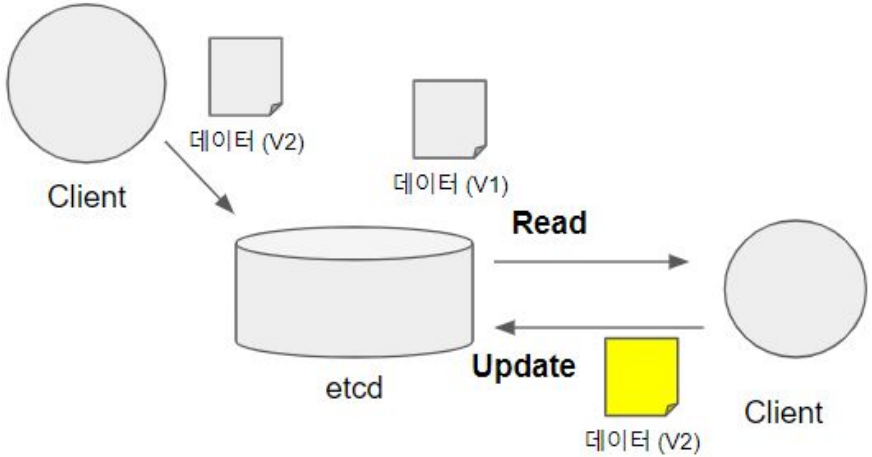
etcd, API server, Scheduler, Controller Manager, and the DNS server are running on the master.

The three nodes each run a Kube Proxy pod and a Flannel networking pod.

컨트롤 플레인 (etcd)

- 쿠버네티스 클러스터 데이터를 저장하기 위한 일관성, 고가용성 키-값 저장소
  - 클러스터 상태, 메타데이터를 저장하기 위함
  - 참고 자료(http://play.etcd.io/play)

- 강력한 낙관적 잠금 기능 (낙관적 동시성 제어)을 제공함



1. Client가 데이터를 읽는다. (V1)
2. Client가 데이터를 수정후 Update를 한다.
3. etcd에 기존 데이터가 V1인지 체크한다.
4. V1이면, Update 성공
5. V1이 아니면, Update 실패
6. 실패시 다시 데이터를 읽고, 새로 Update 요청이 필요

컨트롤 플레인 (etcd)

- 저장소 조회

```
$ etcdctl ls /registry
/registry/configmaps
/registry/daemonsets
/registry/deployments
/registry/events
/registry/namespaces
/registry/pods
...
```

모든 데이터는 /registry 안에 저장됨

```
$ etcdctl ls /registry/pods
/registry/pods/default
/registry/pods/kube-system
```

파드 조회

```
$ etcdctl ls /registry/pods/default
/registry/pods/default/kubia-159041347-xk0vc
/registry/pods/default/kubia-159041347-wt6ga
/registry/pods/default/kubia-159041347-hp2o5
```

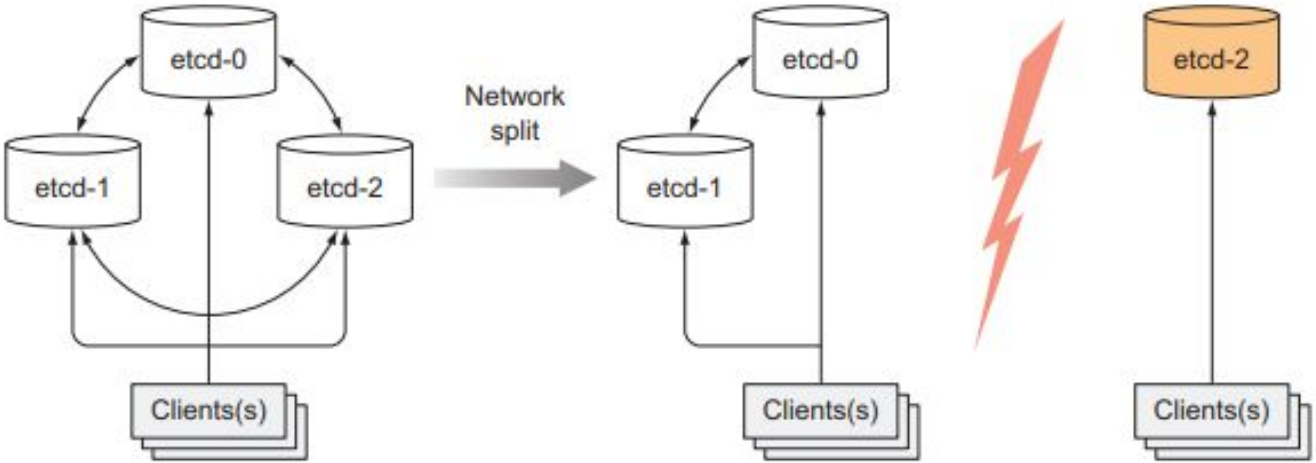
네임 스페이스 조회

```
$ etcdctl get /registry/pods/default/kubia-159041347-wt6ga
{"kind":"Pod","apiVersion":"v1","metadata":{"name":"kubia-159041347-wt6ga",
"generateName":"kubia-159041347-","namespace":"default","selfLink":...
```

파드의 세부 정보 조회 (JSON 형식)

컨트롤 플레인 (etcd)

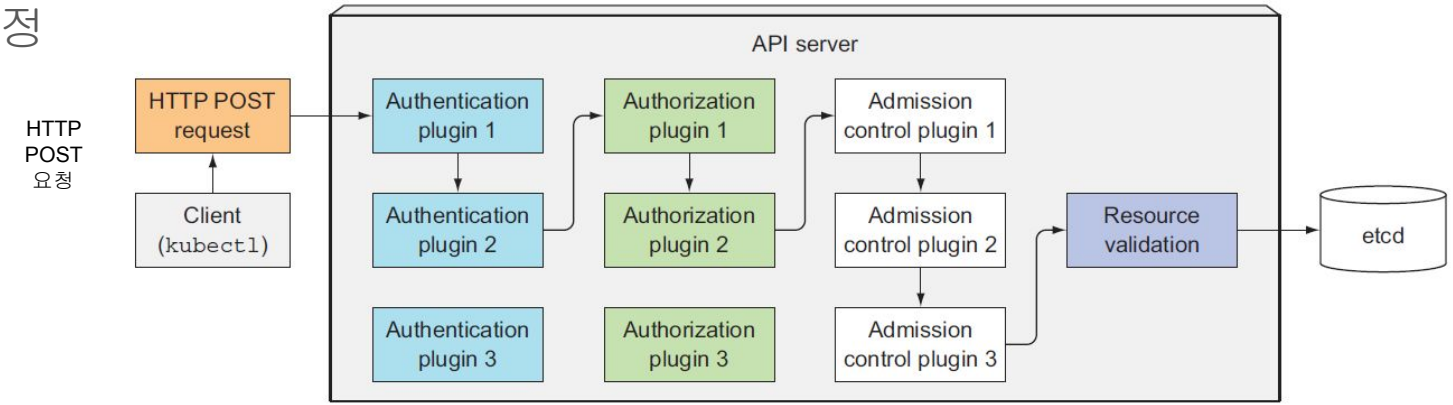
- 고가용성 보장
  - 여러 etcd 인스턴스는 일관성을 유지해야 하며, 이를 위해 RAFT 합의 알고리즘을 사용함.
    - 참고 자료 (http://thesecretlivesofdata.com/raft/)
  - 합의를 위해 과반수 (=쿼럼)가 필요함. (홀수 3, 5, 7대로 운영해야함)
  - 노드 3대는 1대 장애 대응, 5대는 2대 장애 대응, 7대는 3대 장애 대응이 가능함



컨트롤 플레인 (API 서버)

- 다른 모든 구성요소들이 통신하기 위한 중심 구성 요소
  - 클러스터 상태를 조회, 변경하기 위해 Restful API (CRUD)를 제공함
  - 모든 상태는 etcd 안에 저장함

- 동작과정



인증: HTTP 헤더에서 클라이언트 이름, ID, 그룹을 추출해 정상적인 사용자인지 인증함

인가: 인증된 사용자에 한해, 요청한 리소스를 Client가 사용할 수 있는지 판별함

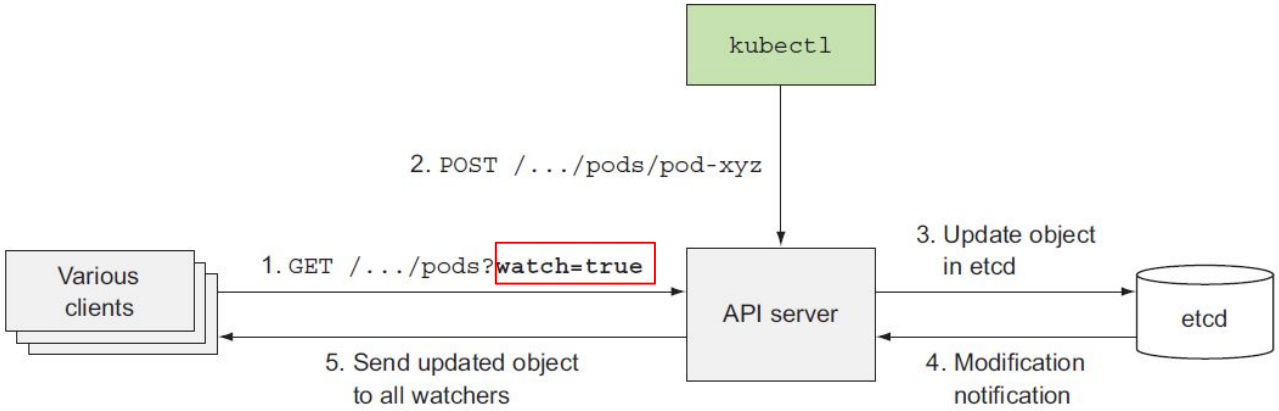
어드미션: 리소스 생성, 수정, 삭제 요청인 경우 어드미션 컨트롤로 보냄 (읽기는 어드미션을 거치지 않음)

컨트롤 플레인 (API 서버)

- 어드미션 컨트롤 플러그인 종류
  - AlwaysPullImages
    - 파드가 배포될때마다 이미지를 항상 강제로 가져오도록 재정의함
  - ServiceAccount
    - 서비스를 명시적으로 지정하지 않을경우, default 서비스 어카운트를 적용함
  - NamespaceLifecycle
    - 삭제되는 과정에 네임스페이스와 존재하지 않는 네임스페이스 안에 파드 생성을 방지함
  - ResourceQuota
    - 특정 네임스페이스 안에 있는 파드가 해당 네임스페이스에 할당된 CPU, 메모리만 사용하도록 강제함
  - 더 많은 어드미션 컨트롤 플러그인 (https://kubernetes.io/docs/admin/admission-controllers/)

컨트롤 플레인 (API 서버)

- 리소스 변경의 통보 방법
  - API 서버는 파드 생성 요청이 오면, 파드를 만들지도 않고, 다른 구성요소에 직접 전달도 안함
  - 단지, 다른 구성요소들이 배포된 리소스의 변경 사항을 관찰 (=감시)하고 있음
    - Client는 API 서버에 HTTP 연결을 맺고 변경 사항을 감지할 수 있음



컨트롤 플레인 (API 서버)

- API 서버 감시 예제

```
$ kubectl get pods --watch
NAME                    READY   STATUS              RESTARTS   AGE
kubia-159041347-14j3i   0/1     Pending             0          0s
kubia-159041347-14j3i   0/1     Pending             0          0s
kubia-159041347-14j3i   0/1     ContainerCreating   0          1s
kubia-159041347-14j3i   0/1     Running             0          3s
kubia-159041347-14j3i   1/1     Running             0          5s
kubia-159041347-14j3i   1/1     Terminating         0          9s
kubia-159041347-14j3i   0/1     Terminating         0          17s
kubia-159041347-14j3i   0/1     Terminating         0          17s
kubia-159041347-14j3i   0/1     Terminating         0          17s
```

파드에 변화가 발생할때,
관련 내용을 전달받음

컨트롤 플레인 (스케줄러)

- 파드를 특정 노드에 할당해주는 역할
  - 스케줄러가 직접 할당하는 것은 아니고, API 서버의 감시 메커니즘을 사용함
  - 새로 생성된 파드가 있다면, 스케줄링 알고리즘을 통해 특정 노드를 선정함
  - API 서버로 파드 정의를 갱신하여 요청을 함
  - 이를 감시하던 Kubelet이 파드의 컨테이너를 생성하고 실행함

- 기본 스케줄링 알고리즘

컨트롤 플레인 (스케줄러)

- 수용 가능한 노드를 찾기 위한 조건은 책 참고 (매우 많음)

- 가장 적합한 노드 선택
  - 상황에 따라 노드를 선택하는 것이 달라질 수 있음
  - ex) 10개 파드를 실행중인 A 노드, 0개 파드를 실행중인 B 노드라면, 당연 B 노드를 선택할것
  - ex) 클라우드를 사용할 경우, 비용 절감을 위해 B 노드를 반환하고 A 노드를 선택할것

- 고급 파드 스케줄링 (16장)
  - 어피니티, 안티-어피니티 규칙을 정의해 클러스터 전체 퍼지거나, 가깝게 유지하도록 강제함

- 다중 스케줄러 사용
  - 여러 개 스케줄러를 사용해, 파드 정의 안에서 schedulerName 속성에 스케줄러를 지정함

컨트롤 플레인 (컨트롤러 매니저)

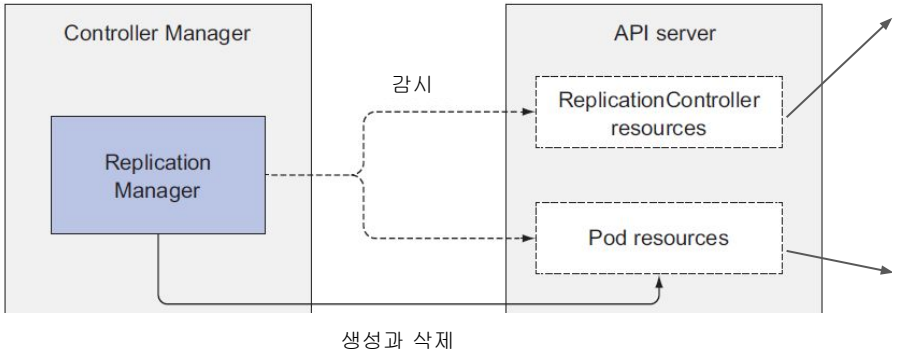
- 컨트롤러를 생성하고 구동하기 위한 구성 요소
  - 컨트롤러는 API 서버에서 리소스 (디폴로이먼트, 서비스)가 변경되는 것을 감시함
  - 이후, 각 변경 작업 (새로운 오브젝트 생성, 이미 있는 오브젝트 갱신 or 삭제)를 수행함

- 컨트롤러 종류
  - 레플리케이션 매니저 (레플리케이션컨트롤러 리소스의 컨트롤러)
  - 레플리카셋, 데몬셋, 잡 컨트롤러
  - 디플로이먼트 컨트롤러
  - 스테이트풀셋 컨트롤러
  - 노드 컨트롤러
  - ...

- 생성할 수 있는 거의 모든 리소스 컨트롤러가 있음

컨트롤 플레인 (컨트롤러 매니저)

- 레플리케이션 매니저 (나머지 컨트롤러도 똑같이 동작함)
  - 레플리케이션 컨트롤러 리소스를 활성화함
  - 실제 작업을 수행하는 것은 레플리케이션 컨트롤러가 아닌 레플리케이션 매니저임
  - 변경된 파드 정의를 API 서버에 게시해 Kubelet이 컨테이너를 생성하고 실행하도록 함




```
apiVersion: v1
kind: ReplicationController
metadata:
  name: kubia
spec:
  replicas: 3
  selector:
    app: kubia
```

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: kubia
spec:
  replicas: 5
  selector:
    app: kubia
```

Replicas 수를 3에서 5로 변경

파드 리소스가 3개에서 5개로 변경

```
$ etcdctl ls /registry/pods/default
/registry/pods/default/kubia-159041347-xk0vc
/registry/pods/default/kubia-159041347-wt6ga
/registry/pods/default/kubia-159041347-hp2o5
/registry/pods/default/kubia-159041347-wt6g
/registry/pods/default/kubia-159041347-hp2o
```

2개 추가

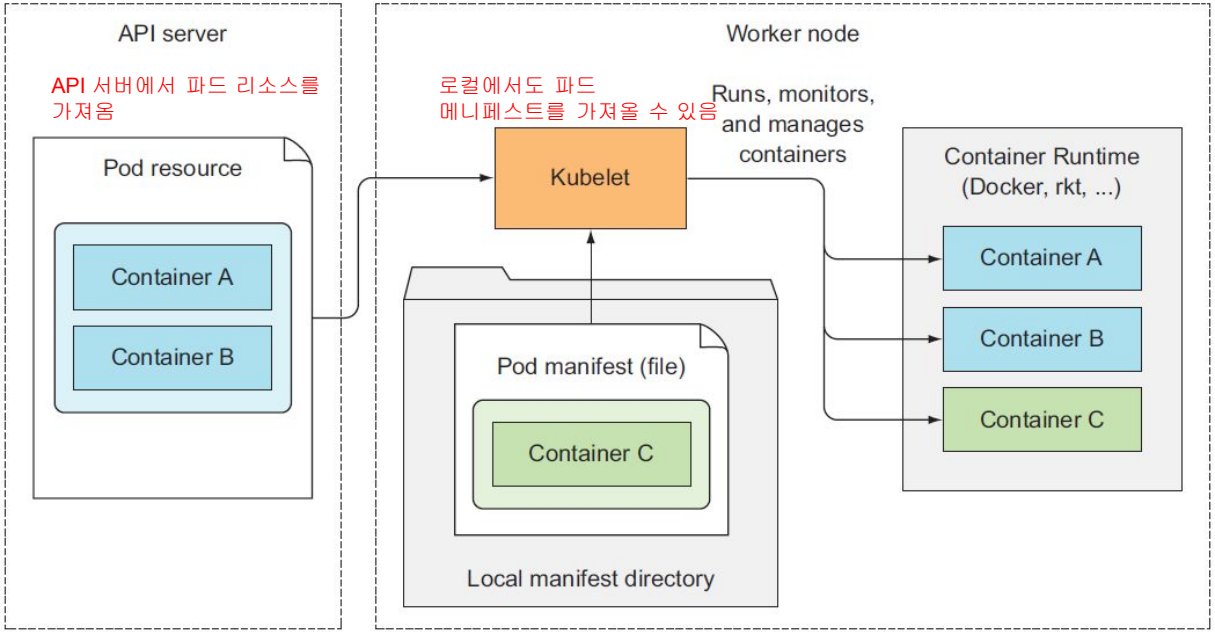
워커 노드 (Kubelet)

- Kubelet의 작업 이해
  - 워커 노드에서 실행하는 모든 것을 담당하는 구성 요소로 아래와 같이 동작함

- 동작 과정
  - Kubelet이 실행 중인 노드를 노드 리소스로 만들어 API 서버에 등록함
    - 이후 파드가 스케줄링이 되면, 파드의 컨테이너를 시작함
    - 설정된 컨테이너 런타임에 지정된 컨테이너 이미지로 컨테이너를 실행하도록 지시함
  - 실행중인 컨테이너를 계속 모니터링하여, 상태, 이벤트, 리소스 사용량을 API 서버에 보고함
  - 컨테이너 라이브니스 프로브를 실행하는 구성요소
    - 프로브가 실패할 경우, 컨테이너를 다시 실행함
  - API 서버에 파드가 삭제되면, 컨테이너를 정지하고, 파드 종료된 것을 API 서버에 보고함

워커 노드 (Kubelet)

- API 서버 없이 정적 파드 실행

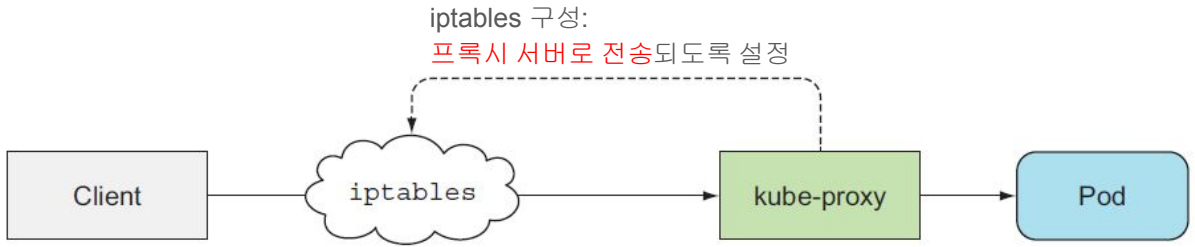


컨테이너화된 버전으로 컨트롤 플레인 구성요소를 파드로 실행하는데 사용됨

워커 노드 (Kube-Proxy)

- Client가 쿠버네티스 API로 정의한 서비스에 연결할 수 있도록함
  - 서비스의 IP/PORT로 들어온 접속을 서비스가 지원하는 파드 중 하나와 연결시켜줌
  - 서비스가 둘 이상의 파드에서 지원되는 경우 파드간 로드밸런싱을 수행함

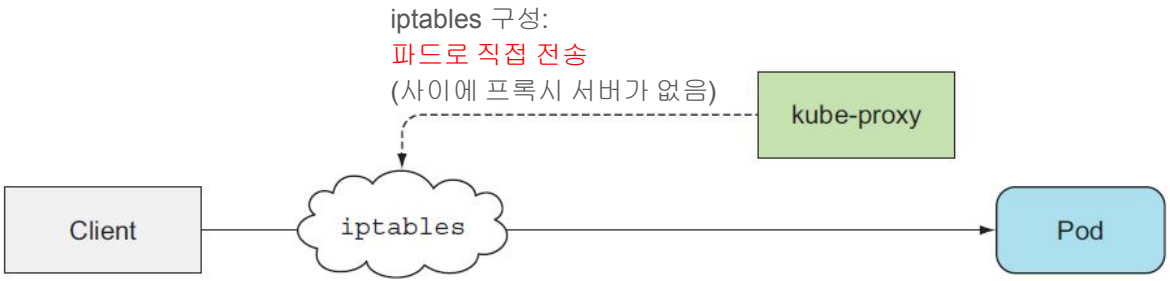
- 초기 구현은 사용자 공간 (userspace)에서 동작하는 프록시
  - 실제 서버 프로세스가 연결을 수락하고 이를 파드로 전달했음
  - 프록시는 iptables* 규칙을 설정했으며, Clinet 요청을 프록시 서버로 전송했음
  - 실제 프록시 역할을 했기에 kube-proxy 라는 이름을 얻었음



*iptables 도구는 리눅스 커널의 패킷 필터링 기능을 관리함

워커 노드 (Kube-Proxy)

- iptables 규칙만 사용해 프록시 서버를 거치지 않는 iptables 프록시 모드
  - 앞선 방법과의 큰 차이는 사용자 공간에서 처리되는지, 커널 공간에서 처리되는지의 여부
  - 또한, userspace 프록시 모드는 라운드 로빈을 통해 파드 간 연결이 균형을 이룸
  - iptables 프록시 모드는 임의 파드로 선택되어 균형이 맞지 않을 수 있음
    - 하지만, 클라이언트나 파드 수가 많다면 문제가 두드러지지 않을것



애드온 구성 요소

- 항상 필요치는 않지만, 활성화 할수 있는 여러 기능들이 있음
  - DNS 조회
  - 여러 HTTP 서비스를 단일 외부 IP 주소로 노출하는 기능 (인그레스 컨트롤러)
  - 쿠버네티스 웹 대시보드

- 애드온 배포 방식
  - YAML 메니페스트를 API 서버에 게시해 파드로 배포함
  - Minikube는 인그레스 컨트롤러, 대시보드 애드온이 레플리케이션컨트롤러로 배포되어 있음

```
$ kubectl get rc -n kube-system
NAME                        DESIRED   CURRENT   READY     AGE
default-http-backend        1         1         1         6d
kubernetes-dashboard        1         1         1         6d
nginx-ingress-controller    1         1         1         6d
```

애드온 구성 요소

- 애드온 배포 방식
  - DNS 애드온은 디플로이먼트로 배포되어 있음

```
$ kubectl get deploy -n kube-system
NAME       DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
kube-dns   1         1         1            1           6d
```

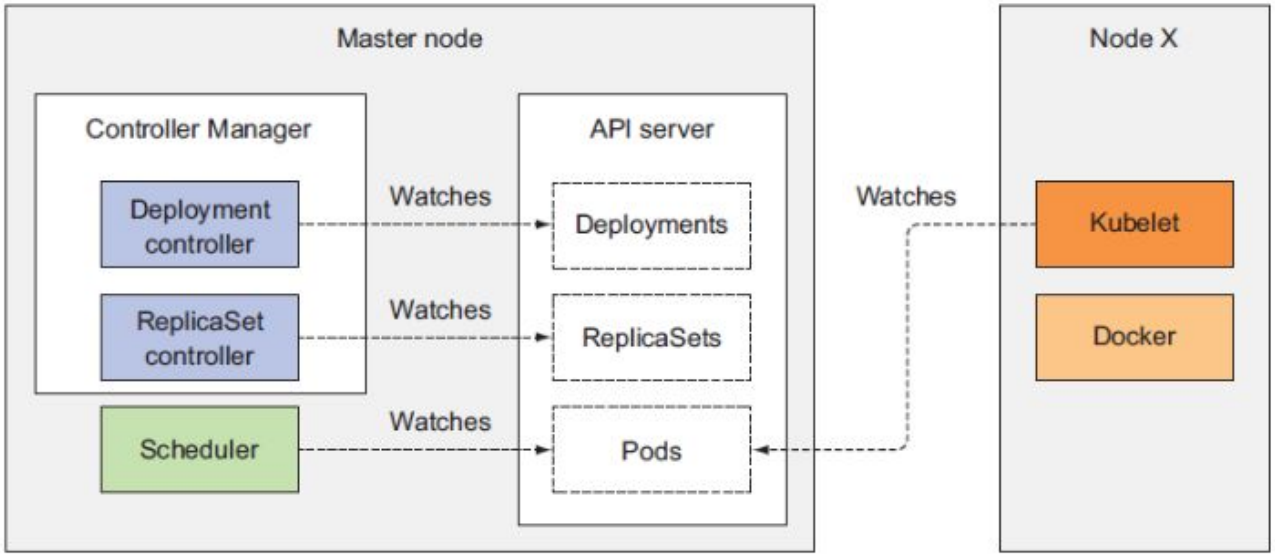
애드온 구성 요소

- DNS 서버 동작 방식
  - 클러스터 내 모든 파드는 클러스터 내부 DNS 서버를 사용하도록 설정됨
    - 이를 통해 파드는 쉽게 서비스 이름을 찾고, 헤드리스 서비스 파드인 경우에 해당 파드 IP 주소를 조회할 수 있음
  - DNS 서버 파드는 kube-dns 서비스로 노출됨
  - 해당 서비스의 IP 주소는 클러스터 내 배포된 모든 컨테이너가 가진 /etc/resolv.conf에 저장

- 인그레스 컨트롤러 동작 방식
  - 리버시 프록시 서버 (ex. Nginx)를 실행하고 클러스터에 정의된 인그레스, 서비스, 엔드포인트 리소스 설정을 유지함
  - 컨트롤러는 리소스를 API 서버를 통해 감시하여 변경이 일어날때 마다 프록시 서버 설정을 변경함

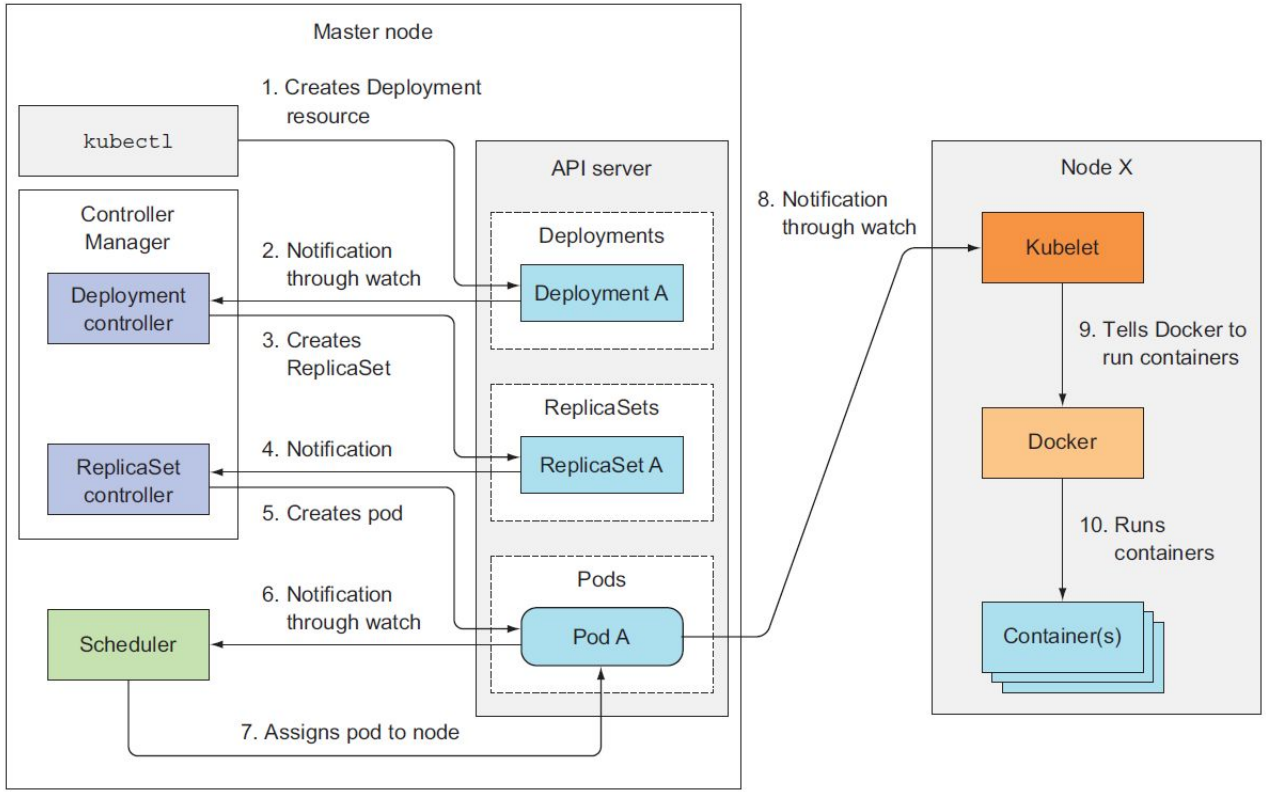
컨트롤러 협업 방법

- 개요
  - 앞서 설명했듯이, 컨트롤러 매니저, 스케줄러, kubelet은 API 서버에서 각 리소스 유형의 변경을 감시함



컨트롤러 협업 방법

- 이벤트 체인



컨트롤러 협업 방법

- 클러스터 이벤트 관찰

```
$ kubectl get events --watch
    NAME              KIND         REASON              SOURCE
... kubia             Deployment   ScalingReplicaSet   deployment-controller
                      ➥ Scaled up replica set kubia-193 to 3
... kubia-193         ReplicaSet   SuccessfulCreate    replicaset-controller
                      ➥ Created pod: kubia-193-w7ll2
... kubia-193-tpg6j   Pod          Scheduled           default-scheduler
                      ➥ Successfully assigned kubia-193-tpg6j to node1
... kubia-193         ReplicaSet   SuccessfulCreate    replicaset-controller
                      ➥ Created pod: kubia-193-39590
... kubia-193         ReplicaSet   SuccessfulCreate    replicaset-controller
                      ➥ Created pod: kubia-193-tpg6j
... kubia-193-39590   Pod          Scheduled           default-scheduler
                      ➥ Successfully assigned kubia-193-39590 to node2
... kubia-193-w7ll2   Pod          Scheduled           default-scheduler
                      ➥ Successfully assigned kubia-193-w7ll2 to node2
... kubia-193-tpg6j   Pod          Pulled              kubelet, node1
                      ➥ Container image already present on machine
```

실행중인 파드의 이해

- 하나의 컨테이너를 가진 파드를 실행했을 때,

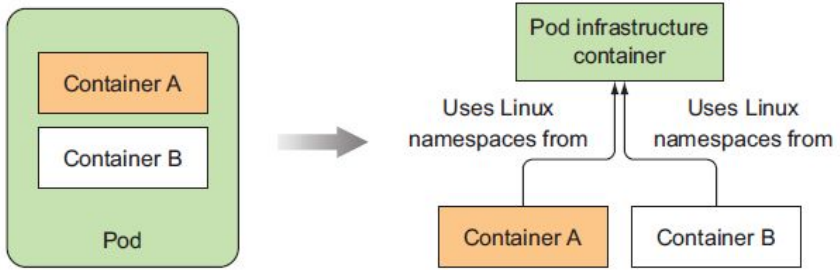
```
$ kubectl run nginx --image=nginx
deployment "nginx" created
```

- 실제 노드 접속 후 docker ps를 하면 아래와 같이 나타남

```
docker@minikubeVM:~$ docker ps
CONTAINER ID    IMAGE                   COMMAND                  CREATED
c917a6f3c3f7    nginx                   "nginx -g 'daemon off"   4 seconds ago
98b8bf797174    gcr.io/.../pause:3.0    "/pause"                 7 seconds ago
```

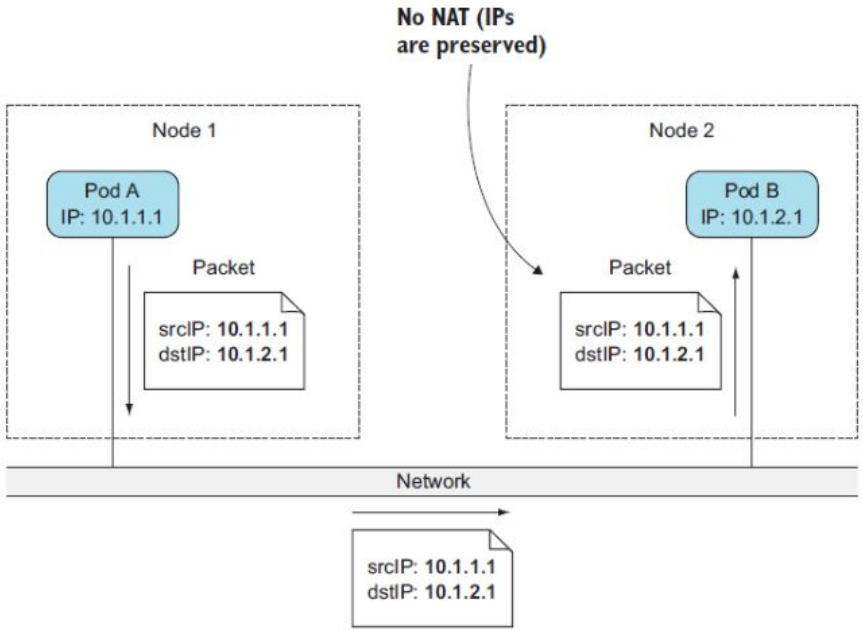
퍼즈 컨테이너 (=인프라스트럭처 컨테이너)라 함

- 파드 내 모든 컨테이너는 동일한 네트워크, 리눅스 네임스페이스를 공유함
- 이러한 네임스페이스를 모두 보유하는 유일한 목적을 가짐
- 파드의 컨테이너는 인프라스트럭처 컨테이너의 네임스페이스를 사용함



파드 간 네트워킹

- 파드는 고유 IP 주소를 가져, 다른 파드와 NAT없이 플랫 네트워크 통신이 가능
  - 파드가 보는 자신의 IP가 다른 해당 파드 주소를 찾을때 정확히 동일해야 함

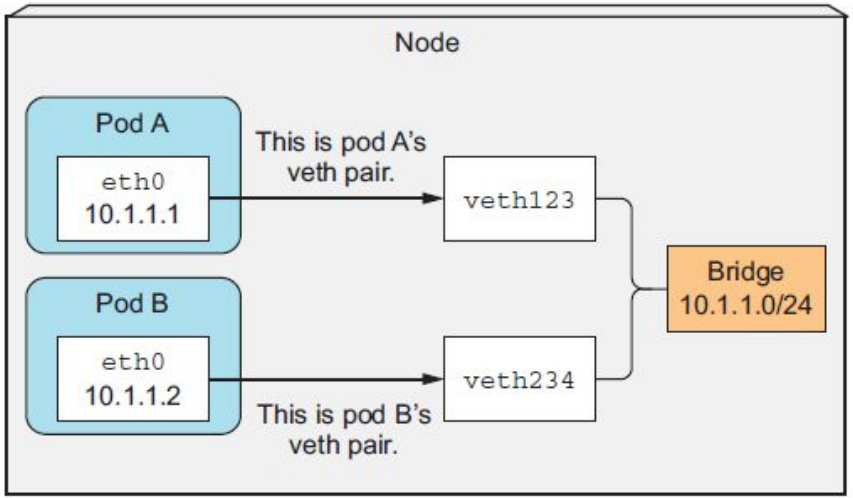


단, 인터넷에 있는 서비스와 통신할 때는, 출발지 IP의 변경이 필요함

- 파드의 IP는 사설 IP이기 때문에, 외부로 나갈때는 호스트 워커 노드의 IP로 변경됨

파드 간 네트워킹

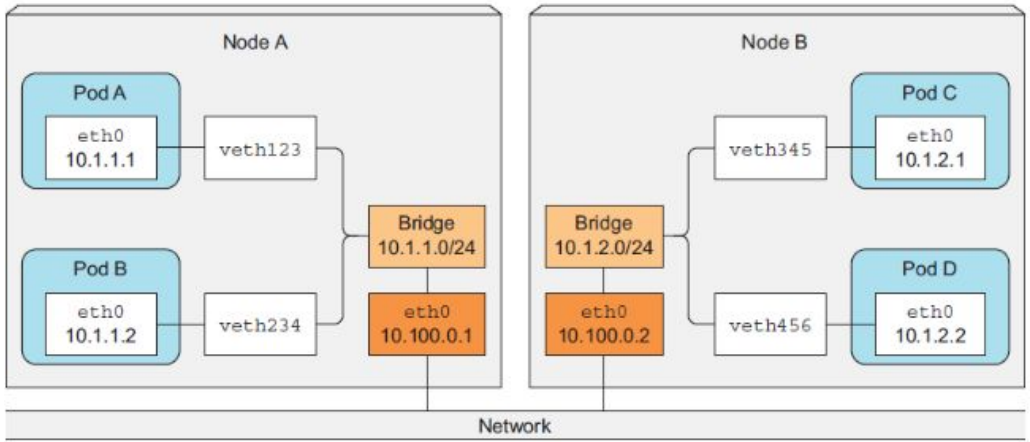
- 네트워킹 동작 방식
  - 파드의 컨테이너는 인프라스트럭쳐 컨테이너에서 설정한 네트워크 인터페이스를 사용함



1. 인프라스트럭쳐 컨테이너가 생성 전, 컨테이너를 위한 가상 이더넷 인터페이스 쌍 (veth 쌍)이 생성됨
   a. 이쌍의 한쪽은 호스트 네트워크 네임스페이스 (vethxxx),
      다른쪽은 컨테이너 네트워크 안으로 옮겨져 eth0으로 변경
   b. 이후 호스트 네트워크 네임스페이스는 네트워크 브리지에 연결됨
2. 컨테이너 내부에 있는 eth0은 각 브리지의 주소 범위 안에 있는 IP를 할당 받음
3. 노드 내 있는 모든 컨테이너는 같은 브리지에 연결되어 서로 통신이 가능함
4. 하지만, 노드 간 통신을 위해 노드 사이의 브리지가 어떤 형태로든 연결되어야 함

파드 간 네트워킹

- 다른 노드에서 파드 간 통신



서로 다른 노드 간 브리지 연결을 위해,
오버레이, 언더레이, 일반 계층 3라우팅이 있음

- 참고로 파드 IP는 전체 클러스터 내 유일해야함
- 그림에서 보듯이
  노드 A는 10.1.1.0/24,
  노드 B는 10.1.2.0/24 를 사용해 충돌 발생을 안함

일반 계층 3 네트워킹으로, 두 노드에서 노드간 통신을 위해 물리 네트워크 인터페이스도 브리지에 연결되어야 함

- 여러 컨테이너 네트워크 인터페이스가 있음 (Calico, Flannel, Romana 등)
  - 네트워크 플러그인 설치를 위해 데몬셋과 다른 지원 리소스를 가지고 있는 YAML을 배포함

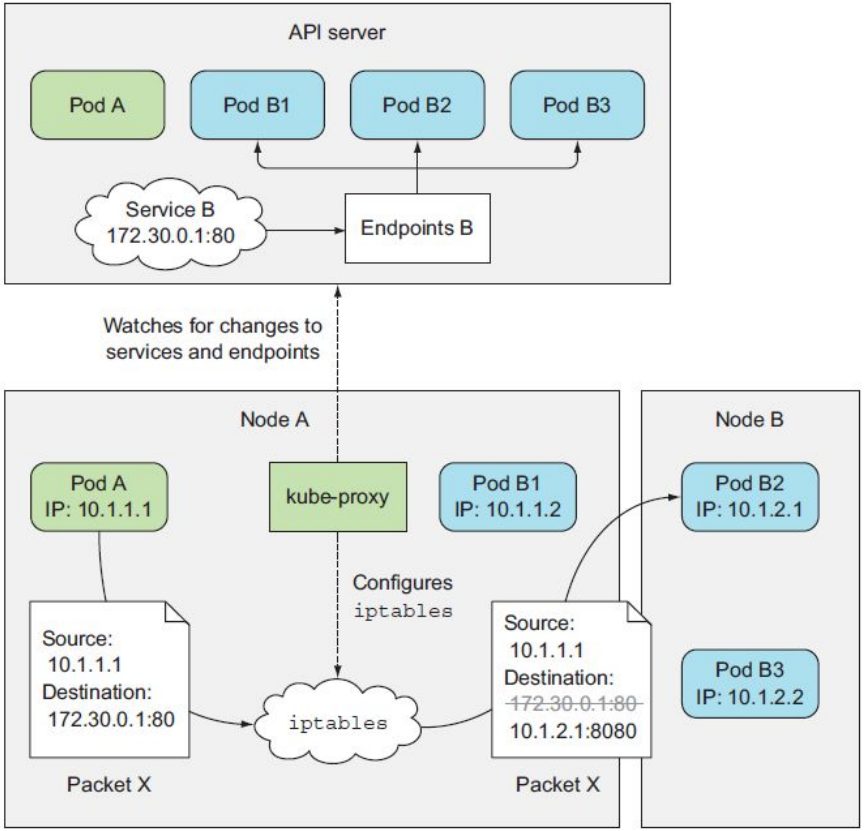
서비스 구현 방식

- 서비스는 파드 집합을 길게 지속되는 안정적인 IP/PORT로 노출시키기 위함

- kube-proxy 소개
  - 서비스와 관련된 모든 것은 각 노드에서 동작하는 kube-proxy 프로세스에 의해 처리됨
  - 서비스 IP는 가상 IP이며, 실제 어떠한 네트워크 인터페이스에 할당되지 않음
    - 즉, 서비스 IP만으로는 아무것도 나타내지 않으며, 서비스에 핑도 날릴 수 없음

- kube-proxy가 iptables를 사용하는 방법
  - API 서버에서 서비스를 생성하면, 가상 IP가 할당되어, 해당 내용을 각 노드 kube-proxy에 통보
    - kube-proxy는 각 서비스 주소로 접근할 수 있도록 만듬
  - 이를 통해, 서비스의 IP/PORT 쌍으로 향하는 패킷을 가로채, 목적지 주소를 변경함
  - 몇 개 iptables 규칙을 설정하여, 여러 파드 중 하나로 리다이렉션 함

서비스 구현 방식



1. 처음 패킷의 목적지는 서비스의 IP/PORT로 지정됨 (**172.30.0.1 : 80**)
2. 패킷이 네트워크로 전송되기 전 노드 A의 커널이 노드에 설정된 iptables 규칙에 따라 먼저 처리함
3. 커널은 패킷이 iptables 규칙 중 일치하는게 있는지 검사함
   a. 규칙 중 목적지 IP가 **172.30.0.1 : 80**가 있다면?
   b. 서비스와 연동된 임의로 선택된 파드의 IP/PORT로 교체돼야 한다고 알림
4. 해당 예제는 B2가 임의로 선택되어, **10.1.2.1:8000**으로 변경됨
5. 이후 Client는 서비스를 통하지 않고 바로 패킷을 B2 파드로 전송함

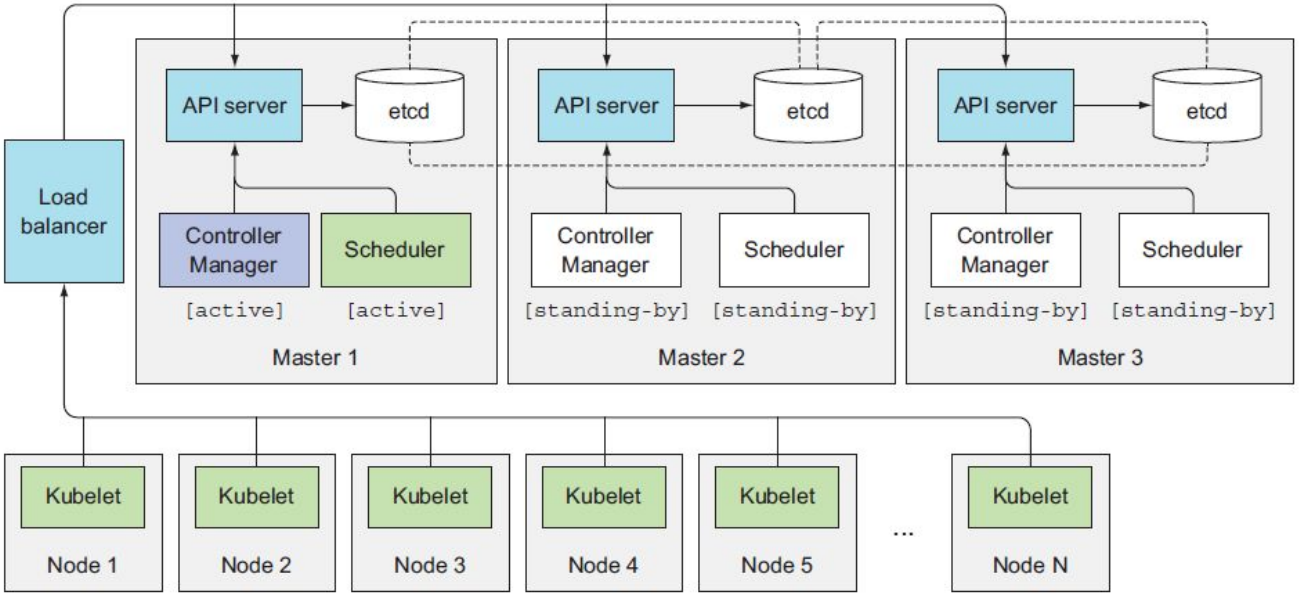
고가용성 클러스터

- 서비스를 중단 없이 계속 실행하기 위해, 구성 요소들은 항상 동작해야 함

- 고가용성 높이기
  - 가동 중단 시간을 줄이기 위해 다중 인스턴스 실행
    - 가동 중단을 줄이기 위해 어플리케이션을 수평 확장해야 함
    - 그렇지 않다면, 레플리카 수를 1로 지정된 디플로이먼트를 사용함
    - 레플리카 장애가 발생시 새로운 레플리카로 교체할 것 (중단 시간이 발생할 것)
  - 수평 스케일링이 불가능한 어플리케이션은 리더 선출 메커니즘 사용
    - 누가 리더가 될지 합의해야하며, (리더는 단 하나) 여러 형태가 있음
    - ex) 리더가 모든 작업을 수행, 나머지는 리더가 실패할 경우를 기다림
    - ex) 리더는 Write 가능, 나머지는 Read만 가능

고가용성 클러스터

- 컨트롤 플레인 가용성 향상



etcd, API 서버는 수평 확장

컨트롤러 매니저, 스케줄러는 Active-Standby 구조

*여기서 Standby 인스턴스는 리더가 되는 것을 기다리는 것 외에 아무것도 하지 않음

고가용성 클러스터

- 리더 선출을 위한 메커니즘
  - 마스터 노드에 각 구성 요소에서 리더를 선출하기 위해 서로 직접 대화할 필요가 없음
    - API 서버에 오브젝트를 생성하는 것만으로 완전히 동작함 (누가 리더인지 기록하면 됨)
  - 리더 선출을 위해 엔드포인트 오브젝트를 사용하는데, 특별한 이유는 없음 (곧 ConfigMap 사용)

```
$ kubectl get endpoints kube-scheduler -n kube-system -o yaml
apiVersion: v1
kind: Endpoints
metadata:
  annotations:
    control-plane.alpha.kubernetes.io/leader: '{"holderIdentity":
      "minikube","leaseDurationSeconds":15,"acquireTime":
      "2017-05-27T18:54:53Z","renewTime":"2017-05-28T13:07:49Z",
      "leaderTransitions":0}'
  creationTimestamp: 2017-05-27T18:54:53Z
  name: kube-scheduler
  namespace: kube-system
  resourceVersion: "654059"
  selfLink: /api/v1/namespaces/kube-system/endpoints/kube-scheduler
  uid: f847bd14-430d-11e7-9720-080027f8fa4e
subsets: []
```

holderIdentity: 리더 이름 필드

*해당 필드의 이름은 처음 성공한 인스턴스가 리더가 됨

요약

- 쿠버네티스 클러스터를 이루는 구성 요소와 역할
- API 서버, 스케줄러, 컨트롤러 매니저 안에서 실행되는 컨트롤러 소개
- Kubelet이 함께 동작하여 파드를 실행하는 방법
- 인프라스트럭쳐 컨테이너가 파드 내 모든 컨테이너를 하나로 묶는 방법
- 네트워크 브리지를 통해 같은 노드 내 파드 통신, 서로 다른 노드 내 파드 통신 방법
- kube-proxy가 노드에 iptables 규칙을 설정해 같은 서비스 내 파드 사이 로드 밸런싱을 수행하는 방법
- 클러스터 가용성을 위해 각 구성 요소 인스턴스를 여러 개 실행하는 방법