컨테이너 오케스트레이션

도커

쿠버네티스

Open Shift

CONTENTS

01. 도커의 등장 배경

도커 이전의 서버 관리

1. 문서화

- 서버 환경/버전 정보를 문서화시켜 관리

- 문서를 보고 환경 세팅 시 문제 발생 가능성 有

- 여러 사람이 수정 시 버전 정보 오기입 가능성 有

2. 서버 관리 도구

- CHEF, PUPPET, ANSIBLE 등 서버 관리 도구 등장

- 사용법이 어려워 서버 관리의 또다른 불편함 야기

3. 가상 머신

- 가상 머신 위에서 서버 관리

- 해당 가상 머신에 종속적이게 되어 유연성이 떨어짐

4. 리눅스 자원 격리 기술

- 리눅스 기술을 활용해, 프로세스, 파일, 디렉토리 등을 가상으로 분리

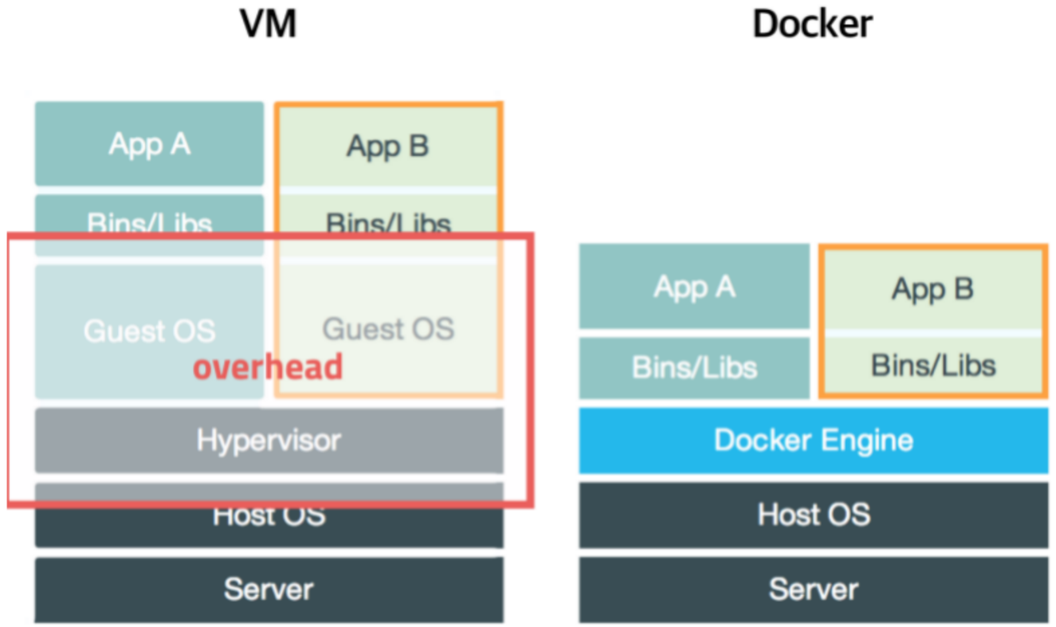
- cpu, memory, I/O 그룹별로 제한하여 분리된 환경에서 서버 관리 구현

- 리눅스 기술을 이용한 빠르고 효율적인 서버 관리가 가능하지만 사용하기 어려움

=> 자원 격리 기술을 쉽고 편하게 사용할 수 있게 하는 컨테이너 기술 등장

02. 도커의 특징

컨테이너 환경에서의 서버 관리



1 가벼운 가상화 기술

- 가상화와 비교하면 컨테이너는 OS 없이 프로세스와 파일 시스템을 분리하는 형태로 크기가 작고 가벼움
- 기타 애플리케이션으로부터 논리적으로 분리된 환경을 제공

2 높은 이동성

- 로컬에서 개발한 컨테이너를 그대로 다른 PC 환경 / 클라우드에서 실행 가능
- 서비스 단위로 유연한 확장이 가능

3 배포 편이성

- 애플리케이션과 종속 항목을 하나의 패키지로 묶어 버전 제어를 쉽게 만듦
- 컨테이너 이미지를 이용한 배포와 롤백이 간단

02. 도커의 특징

도커 이미지

도커 이미지란?

- 레이어드 파일 시스템을 기반으로 프로세스가 실행되는 파일들의 집합(혹은 환경)
- 레이어란 기존 이미지에 추가적인 파일이 필요할 때 다시 다운로드 받는 방법이 아닌 해당 파일을 추가하기 위한 개념
- 기존 이미지에 새로운 파일을 추가하여 새로운 이미지를 만들 수 있음

이미지 특징

- 불변성 : 한번 생성된 이미지는 수정이 불가능 하며
- 영구성 : 이미지를 기반으로 구동한 컨테이너가 삭제되어도 이미지는 삭제되지 않음. 단, DB 내역과 같이 컨테이너 내부 저장 데이터는 삭제
- (상대적) 적은 용량 : 보통 수백MB ~ 수GB의 용량을 가지지만, 가상머신 이미지와 비교하면 굉장히 적은 용량

버전 관리

- 이미지들은 github과 유사한 서비스인 DockerHub를 통해 배포 및 버전 관리 용이

=> 이미지를 통한 더 쉽게 조립하고, 유지관리하고, 이동시킬 수 있는 애플리케이션 제작 환경

03. 컨테이너 오케스트레이션

도커의 한계

1. 배포 과정 작업
   - 컨테이너의 증감 및 각각의 컨테이너 버전 update를 수작업으로 up/down을 해줘야 함

2. 서비스 검색
   - 서버 부하 & MSA로 인해 서버의 수가 많아지면 등록해야 하는 Load Balancer 및 서버 수 증가

3. 서비스 노출
   - 등록해야 하는 도메인 多

4. 모니터링
   - 이상 발견 & 조치의 수작업
   - 관리 대상 증가 시 작업 부담 증가

=> 개별 컨테이너 관리의 유연성은 확보했지만, 다중 컨테이너 관리의 유연성은 부족

=> 복잡한 컨테이너 환경을 효과적으로 관리하기 위한 도구의 필요성 대두

03. 컨테이너 오케스트레이션

컨테이너 오케스트레이션의 동작

1. 클러스터

- 중앙제어 : 노드들을 추상화해서 클러스터 단위로 관리
- 네트워킹 : 노드들끼리 네트워크 통신

2. 상태 관리

- 컨테이너 증감 자동화
- 컨테이너 이상 시, 셧다운 & 개설 자동화

3. Scheduling

- 배포 관리 : 리소스 상황에 맞춰 서버 증감 관리

4. ROLLOUT/ROLLBACK

- 중앙에서의 배포 버전 관리

5. SERVICE DISCOVERY

- 서비스 등록 및 조회 : 서비스 등록에 맞춰 PROXY의 설정 변경

6. SERVICE DISCOVERY

- 각 컨테이너 별 볼륨 관리를 중앙 설정으로 관리
- 볼륨 : 데이터 영속성을 위한 컨테이너에 연결된 별도 스토리지

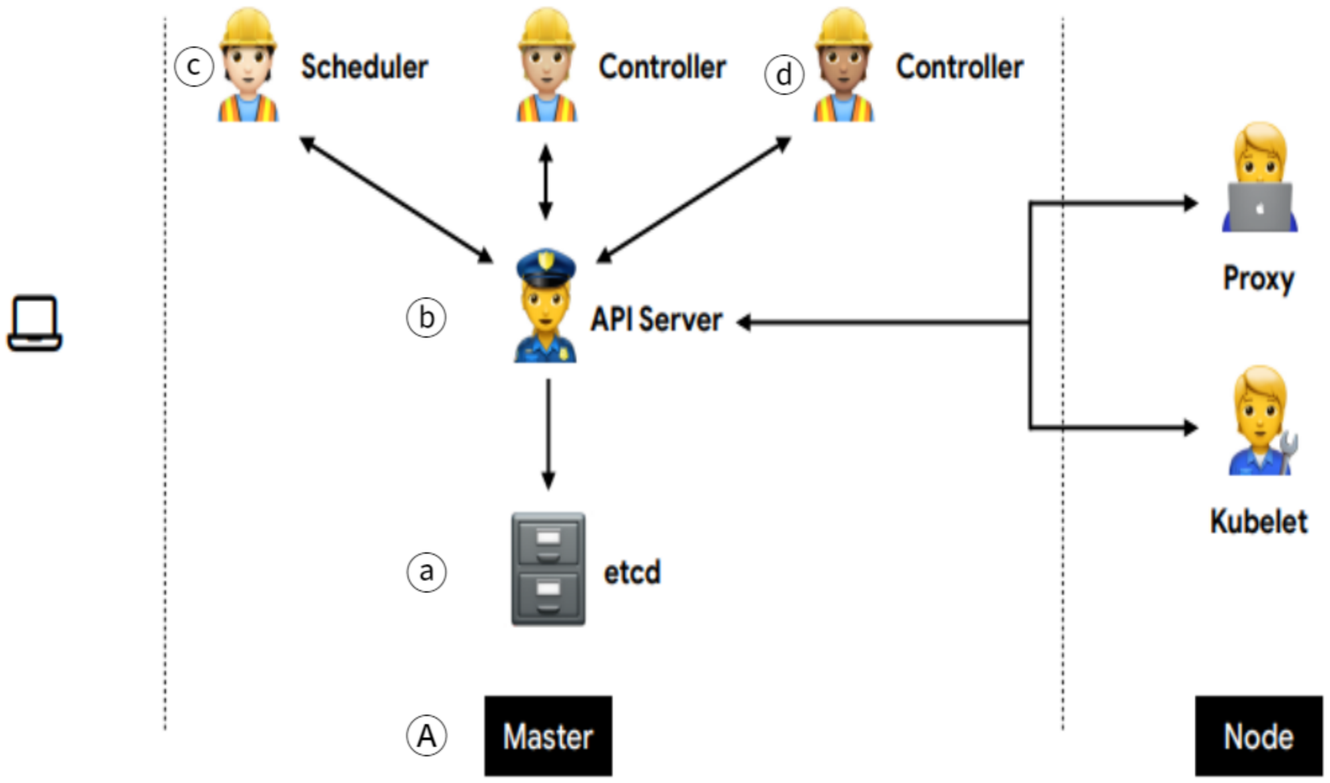
04. 쿠버네티스 아키텍처

쿠버네티스 기본 동작

원하는 상태 변경 => 상태 체크 => 차이점 발견 => 조치 의 반복을 통해 현재 상태 == 원하는 상태 를 유지함

04. 쿠버네티스 아키텍처

쿠버네티스 아키텍처



Ⓐ 마스터 노드

- 쿠버네티스 클러스터 전체를 컨트롤

ⓐ etcd

- 모든 상태와 데이터 저장
- 분산 시스템 => 안정적이며 가볍고 빠름

ⓑ API Server

- etcd와 유일하게 통신하며 상태를 바꾸거나 조회 요청
- 권한 체크 => 요청 허용 / 차단

ⓒ Scheduler

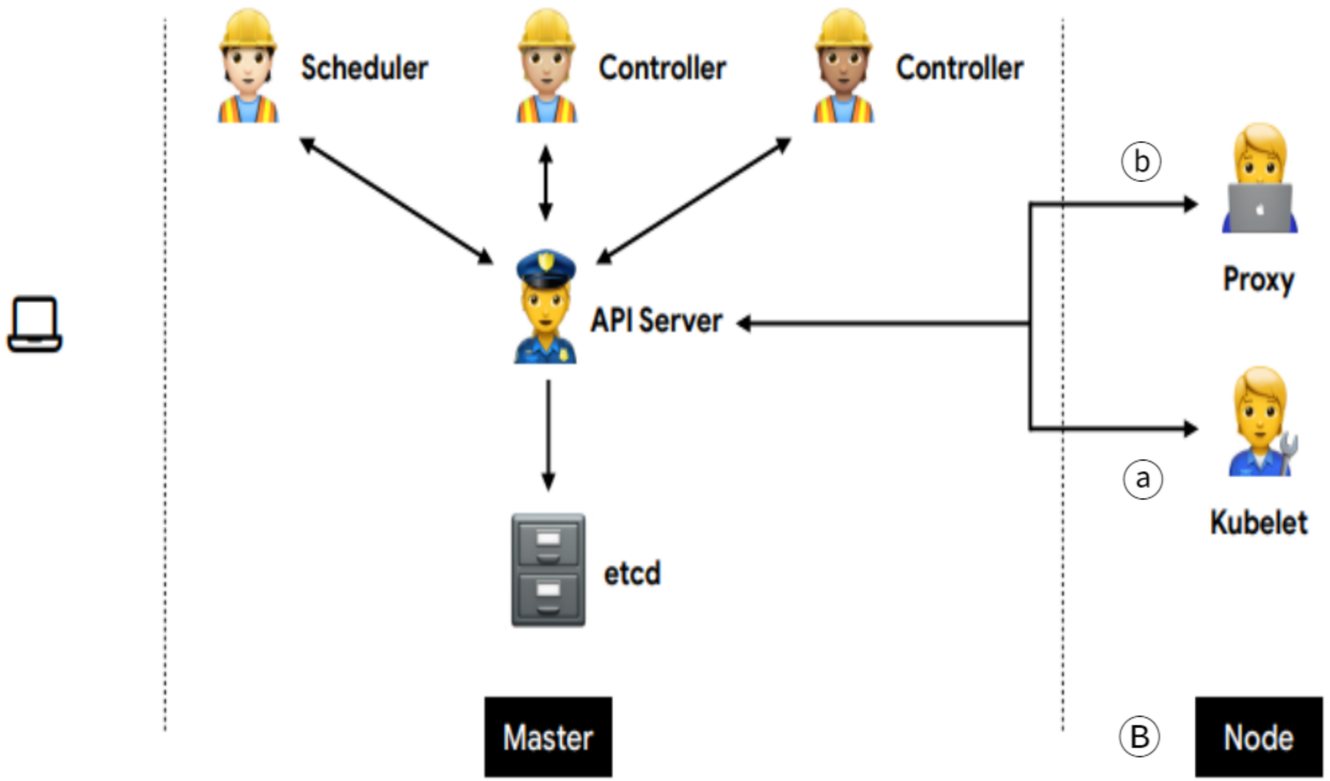
- 새로 생성된 Pod 감지 및 실행할 노드 선택
- 노드의 현재 상태와 Pod의 요구사항 체크

ⓓ Controller

- 끊임없이 쿠버네티스 오브젝트들의 상태 체크 및 원하는 상태 유지
- 논리적으로 다양한 컨트롤러 존재

04. 쿠버네티스 아키텍처

쿠버네티스 아키텍처



Ⓑ (워커) 노드

- 마스터에 의해 명령을 받고 실제 컨테이너가 생성되는 서버 영역

ⓐ Kubelet

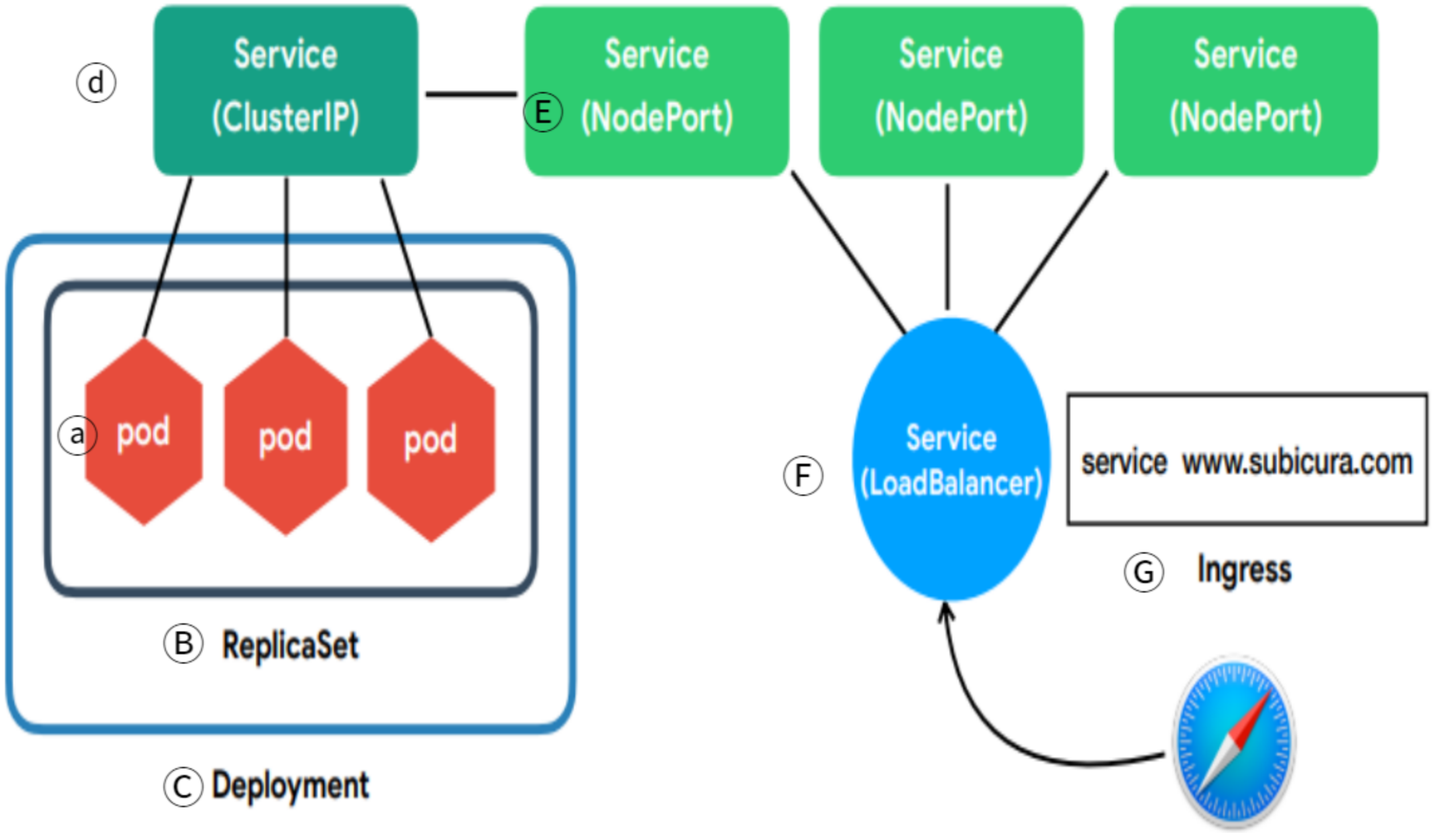
- Container Runtime Interface
- 각 노드에서 Pod의 실행 / 중지 및 상태 체크 실행

ⓑ Proxy

- 네트워크 프록시와 부하 분산 역할

05. 쿠버네티스 오브젝트

쿠버네티스의 일반적인 구성



ⓐ Pod

- 쿠버네티스의 가장 작은 배포 단위
- 한 개 이상의 컨테이너가 하나의 pod에 속함

Ⓑ ReplicaSet

- 여러 개의 pod을 관리
- 신규 Pod을 생성하거나 기존 Pod을 제거하여 원하는 수(Replicas)를 유지

Ⓒ Deployment

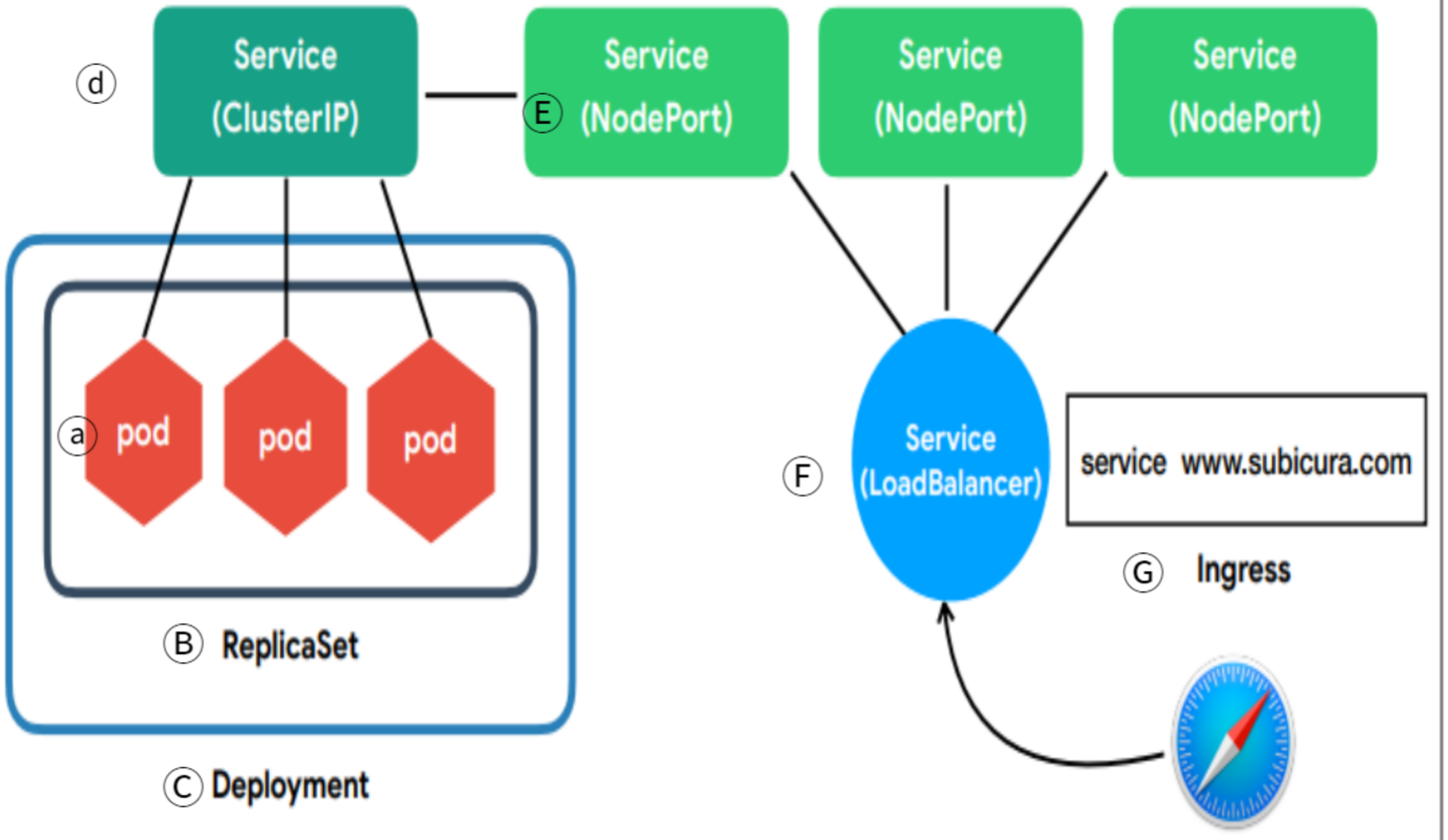
- 배포 버전을 관리
- 파드를 기존 레플리카셋에서 새로운 레플리카셋으로 속도를 제어하며 이동하는 것을 관리

ⓓ cluserIp

- pod는 동적이지만 서비스는 고유 IP 가짐
- pod의 생성과 삭제에 무관한 서비스 단위의 내부 Ip 주소
- 여러 개의 Pod 바라보는 로드밸런서 기능

05. 쿠버네티스 오브젝트

쿠버네티스의 일반적인 구성



Ⓔ NodePort

- 노드를 port에 연결시켜 외부에서 접근 가능한 서비스
- 노드는 각각 고유한 ip 주소를 가지지만
  같은 서비스끼리는 동일판 포트 번호 생성

Ⓕ LoadBalancer

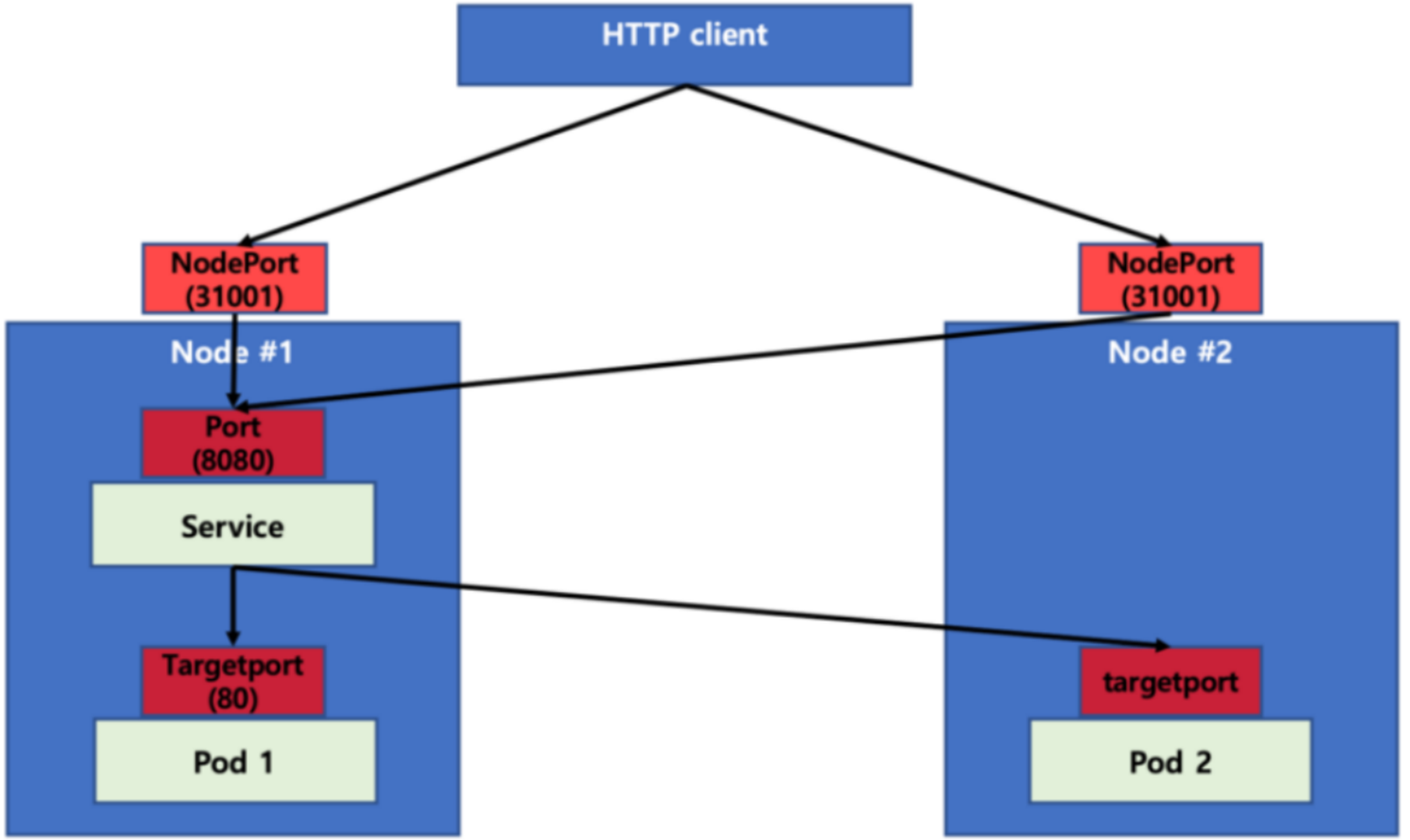
- 불변적인 노드를 대신하여 하나의 ip 주소를 외부에 노출
- Service당 각각의 LoadBalancer 등록

Ⓖ Ingress

- DNS을 통한 외부에서의 접근에 대해 설정한 Rule에
  따라 맵핑되는 Service에게 Routing

05. 쿠버네티스 오브젝트

참고. NodePort 구조



06.OpenShift

OpenShift의 개념 및 이점

OpenShift : 레드햇에서 만든 컨테이너 기반 소프트웨어의 배포 및 관리 플랫폼으로, 쿠버네티스의 배포판 역할 수행

1. 배포 자동화

쿠버네티스 배포 절차

: 코드 작성 => 이미지 생성 => k8s 파일 작성

=> repo 저장 => 빌드

OpenShift 배포 절차

: 코드 작성 => git 등 push

- Jenkins pipeline을 통해서 빌드 / 배포 자동화

c.f. k8s : 쿠버네티스

2. 고가용성

시스템에 장애가 생기더라도 빠르게 복구

기본적으로 2 이상의 replicas 운영

=> Pod에 장애가 생기더라도 router가 빠르게 대처

3. Auto-Scaling

쿠버네시트

: 설정한 replicaset 수 만큼 유지

Openshift

: Pod의 최소 ~ 최대 수 설정

=> 자원 사용에 따른 자동 증감

=> 모든 호스트에서 여유공간이 없을 때 새로운 호스트 생성 및 작업 자동화

4. 보안

이미지에 대한 까다로운 요구사항과 암호화 서비스에 솔루션 제공

=> 더 높은 수준의 보안 유지