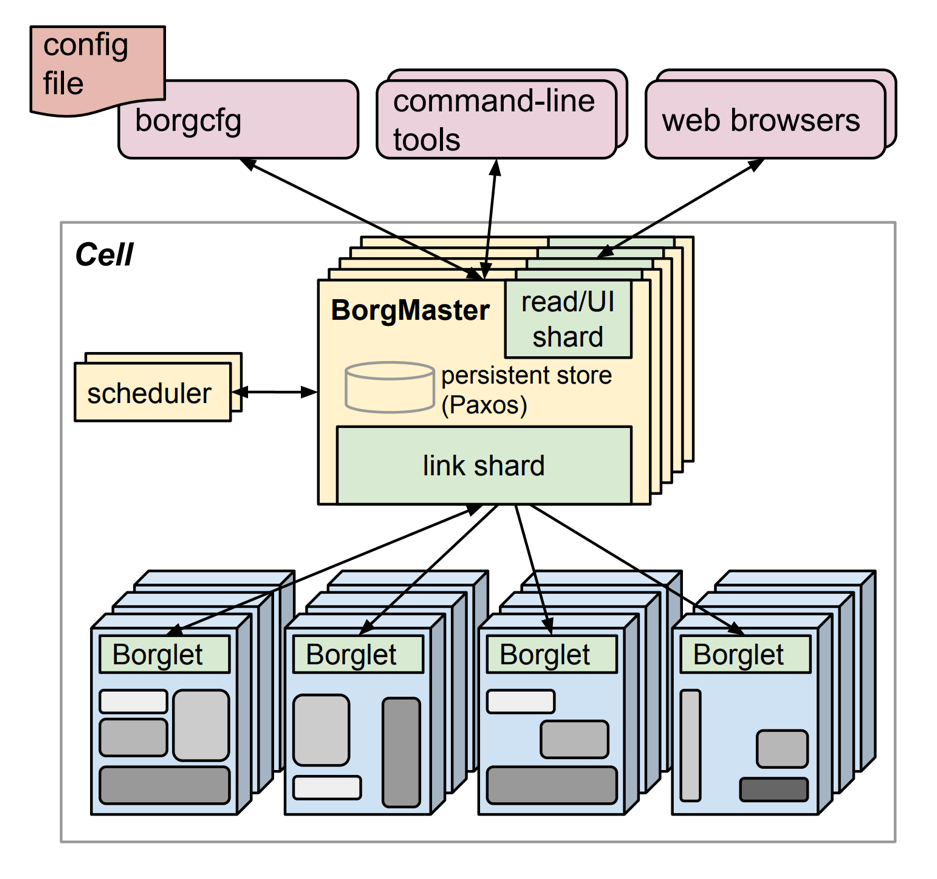
**Kubernetes 란?**

컨테이너화된 워크로드와 서비스를 관리하기 위한 이식성이 있고, 확장 가능한 오픈소스 플랫폼

쿠버네티스는 구글이 2014년 6월 7일 발표한 오픈소스 컨테이너 오케스트라제이션 시스템이다.

최신 버전은 1.18 버전이며, Go로 작성되었다.

쿠버네티스는 구글의 내부 서버 관리 시스템에서 시작되었다. 구글은 기존에 Borg라는 매니저를 사용하고 있었다. BorgManager와 Borgiet으로 구성되어 작동하는데, Docker와 Kubernetes와 매우 유사한 구조이다. Borg는 주로 물리 서버를 직접 사용했는데, 이는 구글이 가상화할때 발생하는 손실을 감당하기 싫어서이다.



Borg에서 구동하는 대부분의 작업은 성능 지표나 현재 상태를 보여주는 HTTP 서버를 포함했다.

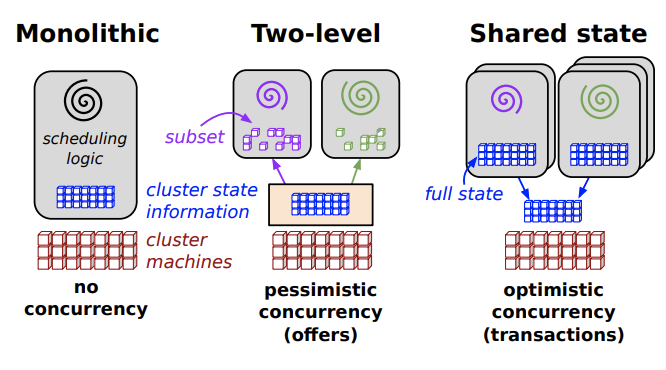
대부분의 Borg 개발자는 Kubernetes를 개발 중이다. 즉, Borg는 Kubernetes로 대체되었다.

Borg는 다음과 같은 문제들을 가지고 있었다.

* **작업을 하나의 엔티티처럼 실행할 수 있는 방법을 기본으로 지원하지 않았다.** 도커는 이를 docker-compose로 지원한다.
* **한 머신 당 1개의 IP만 할당할 수 있었다.** 쿠버네티스는 모든 포드에 하나의 IP를 할당한다.
* **무거운 작업을 위해서 설계되었기에, 가벼운 작업을 효율적으로 실행할 수 없었다.**

구글은 Omega라는 컨테이너 클러스터 매니저도 만들었다. 완전히 새롭게 개발되었지만, Borg와 유사한 구조를 가지고 있으며, Borg보다 약간 더 빠르다.

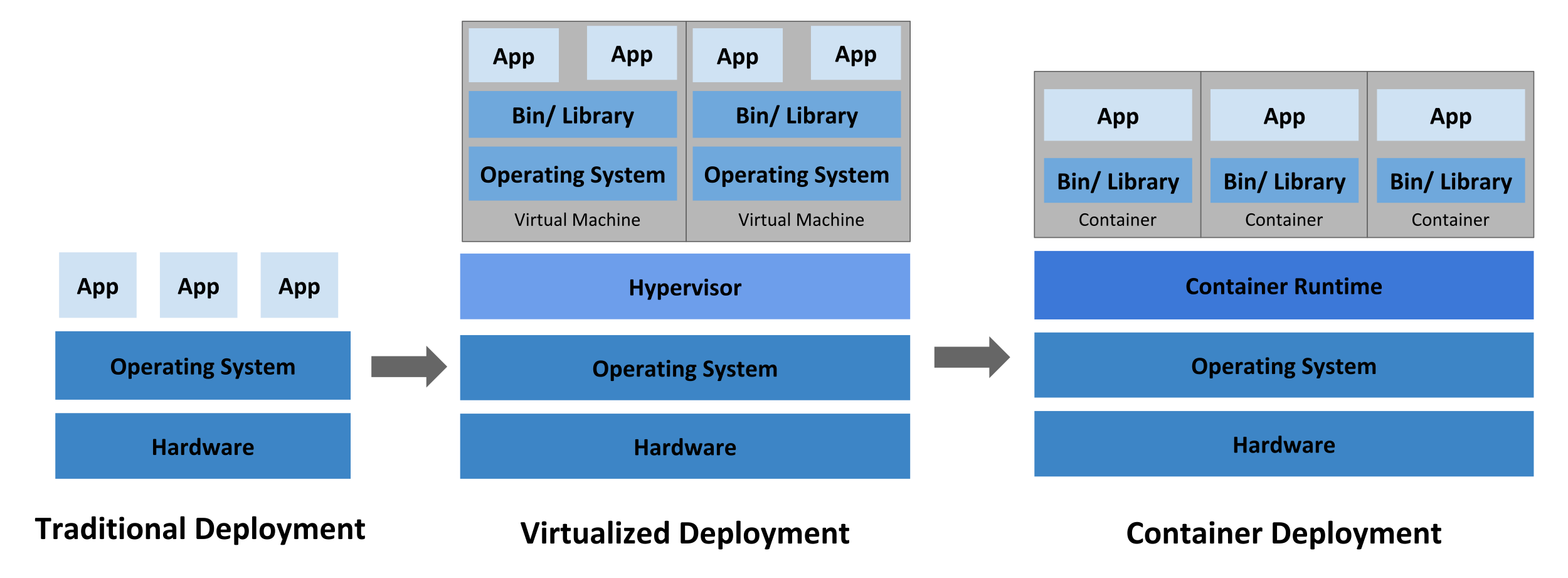
Omega는 모놀리틱한 Borg와는 다르게, 컨테이너들이 공유하는 state가 있었고, 더 나은 스케쥴링 아키텍쳐를 제공했다.

텍스트, 지도이(가) 표시된 사진

자동 생성된 설명

여기서 찾은 단점을 해결하기 위하여 구글은 Kubernetes를 개발하게 된다.

쿠버네티스란 명칭은 키잡이(helmsman)나 파일럿을 뜻하는 그리스어에서 유래했다.



기존의 개발 방식에서는 애플리케이션을 물리 서버에서 실행했었다. 한 물리 서버에서 여러 애플리케이션의 리소스 한계를 정의할 방식이 없었기에, 리소스 할당의 문제가 발생했다. 즉, 물리 서버 하나에서 여러 애플리케이션을 실행할 때 리소스를 많이 사용하는 애플리케이션이 있으면 다른 애플리케이션의 성능이 저하될 수 있었다. 이는 여러 물리 서버에서 각 애플리케이션을 실행하는 방법으로 해결할 수 있지만 리소스가 낭비되는 결과를 가져오며, 물리 서버를 유지하기 위해서 많은 비용을 써야하는 문제도 있었다.

위의 문제를 해결하기 위해서 가상화가 도입되었다. 이는 단일 물리 서버 CPU에서 여러 VM을 실행할 수 있게 해 주었다. 가상화를 사용하면 애플리케이션을 VM에서 사용하게 되므로 애플리케이션이 다른 애플리케이션의 정보를 자유롭게 접촉할 수 없게 되면서 일정 수준의 보안성이 제공된다.

VM을 사용 시, 물리 서버에서 리소스를 보다 효율적으로 사용할 수 있으며, 쉽게 애플리케이션을 변경할 수 있고 하드웨어 비용을 절감할 수 있어 더 나은 확장성을 제공한다. 가상화를 통해 물리 서버에서 리소스를 보다 효율적으로 활용할 수 있으며, 쉽게 애플리케이션을 추가하거나 업데이트할 수 있고 하드웨어 비용을 절감할 수 있어 더 나은 확장성을 제공한다.

각 VM은 가상화된 하드웨어 상에서 자체 운영체제를 포함한 모든 구성 요소를 실행한다.

가상 하드웨어는 곧 컨테이너로 발전하게 된다. 컨테이너는 VM과 유사하지만 격리 속성을 완화하여 애플리케이션 간에 운영체제(OS)를 공유한다. 그러므로 컨테이너는 가볍다고 여겨진다. VM과 마찬가지로 컨테이너에는 자체 파일 시스템, CPU, 메모리, 프로세스 공간 등이 있다. 기본 인프라와의 종속성을 끊었기 때문에, 클라우드나 OS 배포본에 모두 이식할 수 있다.

컨테이너는 다음과 같은 추가적인 혜택을 제공하기 때문에 인기가 있다.

* 기민한 애플리케이션 생성과 배포: VM 이미지를 사용하는 것에 비해 컨테이너 이미지 생성이 보다 쉽고 효율적임.
* **컨테이너 이미지의 안정적이고 주기적인 빌드/배포와 빠르고 쉬운 롤백**
* 개발과 운영의 관심사 분리: 배포 시점이 아닌 빌드/릴리스 시점에 애플리케이션 컨테이너 이미지를 만들기 때문에, 애플리케이션이 인프라스트럭처에서 분리된다. 가시성은 OS 수준의 정보와 메트릭에 머무르지 않고, 애플리케이션의 헬스와 그 밖의 시그널을 볼 수 있다.
* 개발, 테스팅 및 운영 환경에 걸친 일관성: 랩탑에서도 클라우드에서와 동일하게 구동된다.
* 클라우드 및 OS 배포판 간 이식성: Ubuntu, RHEL, CoreOS, 온-프레미스, 주요 퍼블릭 클라우드와 어디에서든 구동된다.
* 애플리케이션 중심 관리: 가상 하드웨어 상에서 OS를 실행하는 수준에서 논리적인 리소스를 사용하는 OS 상에서 애플리케이션을 실행하는 수준으로 추상화 수준이 높아진다.
* 느슨하게 커플되고, 분산되고, 유연하며, 자유로운 마이크로서비스: 애플리케이션은 단일 목적의 머신에서 모놀리식 스택으로 구동되지 않고 보다 작고 독립적인 단위로 쪼개져서 동적으로 배포되고 관리될 수 있다.
* 리소스 격리: 애플리케이션 성능을 예측할 수 있다.
* 리소스 사용량: 고효율 고집적.

컨테이너는 애플리케이션을 포장하고 실행하는 좋은 방법이다. 프로덕션 환경에서는 애플리케이션을 실행하는 컨테이너를 관리하고 가동 중지 시간이 없는지 확인해야한다. 쿠버네티스는 이 역할을 대신해준다. 쿠버네티스는 분산 시스템을 탄력적으로 실행하기 위한 프레임 워크를 제공한다. 애플리케이션의 확장과 장애 조치를 처리하고, 배포 패턴 등을 제공한다.

* 서비스 디스커버리와 로드 밸런싱: 쿠버네티스는 DNS 이름을 사용하거나 자체 IP 주소를 사용하여 컨테이너를 노출할 수 있다. 컨테이너에 대한 트래픽이 많으면, 쿠버네티스는 네트워크 트래픽을 로드밸런싱하고 배포하여 배포가 안정적으로 이루어질 수 있다.
* 스토리지 오케스트레이션: 쿠버네티스를 사용하면 로컬 저장소, 공용 클라우드 공급자 등과 같이 원하는 저장소 시스템을 자동으로 탑재 할 수 있다.
* 자동화된 롤아웃과 롤백: 쿠버네티스를 사용하여 배포된 컨테이너의 원하는 상태를 서술할 수 있으며 현재 상태를 원하는 상태로 설정한 속도에 따라 변경할 수 있다. 예를 들어 쿠버네티스를 자동화해서 배포용 새 컨테이너를 만들고, 기존 컨테이너를 제거하고, 모든 리소스를 새 컨테이너에 적용할 수 있다.
* 자동화된 빈 패킹(bin packing): 컨테이너화된 작업을 실행하는데 사용할 수 있는 쿠버네티스 클러스터 노드를 제공한다. 각 컨테이너가 필요로 하는 CPU와 메모리(RAM)를 쿠버네티스에게 지시한다. 쿠버네티스는 컨테이너를 노드에 맞추어서 리소스를 가장 잘 사용할 수 있도록 해준다.
* 자동화된 복구(self-healing): 쿠버네티스는 실패한 컨테이너를 다시 시작하고, 컨테이너를 교체하며, ‘사용자 정의 상태 검사’에 응답하지 않는 컨테이너를 죽이고, 서비스 준비가 끝날 때까지 그러한 과정을 클라이언트에 보여주지 않는다.
* 시크릿과 구성 관리: 쿠버네티스를 사용하면 암호, OAuth 토큰 및 SSH 키와 같은 중요한 정보를 저장하고 관리 할 수 있다. 컨테이너 이미지를 재구성하지 않고 스택 구성에 시크릿을 노출하지 않고도 시크릿 및 애플리케이션 구성을 배포 및 업데이트 할 수 있다.

쿠버네티스는 전통적인 PaaS가 아니다. 쿠버네티스는 하드웨어 수준보다는 컨테이너 수준에서 운영되기 때문에, PaaS가 일반적으로 제공하는 배포, 스케일링, 로드 밸런싱, 로깅 및 모니터링과 같은 기능에서 공통점이 있기도 하다. 하지만, 쿠버네티스는 모놀리식(monolithic)이 아니어서, 이런 기본 솔루션이 선택적이며 추가나 제거가 용이하다. 쿠버네티스는 개발자 플랫폼을 만드는 구성 요소를 제공하지만, 필요한 경우 사용자의 선택권과 유연성을 지켜준다.

* stateless, stateful, 데이터 처리 워크로드를 포함하는 극단적으로 다양한 워크로드를 지원하는 걸 목표로 한다. 애플리케이션이 컨테이너에서도 잘 돌아가면, 쿠버네티스에서도 잘 동작한다.
* 소스 코드를 배포하지 않으며 애플리케이션을 빌드하지 않는다. 지속적인 통합과 전달과 배포, 곧 CI/CD 워크플로우는 조직 문화와 취향에 따를 뿐만 아니라 기술적인 요구사항으로 결정된다.
* 애플리케이션 레벨의 서비스를 제공하지 않는다. 애플리케이션 레벨의 서비스에는 미들웨어(예, 메시지 버스), 데이터 처리 프레임워크(예, Spark), 데이터베이스(예, MySQL), 캐시 또는 클러스터 스토리지 시스템(예, Ceph) 등이 있다. 이런 컴포넌트는 쿠버네티스 상에서 구동될 수 있고, 쿠버네티스 상에서 구동 중인 애플리케이션이 [Open Service Broker](https://openservicebrokerapi.org/) 와 같은 이식 가능한 메커니즘을 통해 접근할 수도 있다. 즉, 이런 컴포넌트는 사용자가 직접 컨테이너를 통해 추가해야 한다.
* 로깅, 모니터링 또는 경보 솔루션을 포함하지 않는다. 개념 증명을 위한 일부 통합이나, 메트릭을 수집하고 노출하는 메커니즘을 제공한다.
* 기본 설정 언어/시스템(예, Jsonnet)을 제공하거나 요구하지 않는다. 선언적 명세의 임의적인 형식을 목적으로 하는 선언적 API를 제공한다.
* 포괄적인 머신 설정, 유지보수, 관리, 자동 복구 시스템을 제공하거나 채택하지 않는다.
* 추가로, 쿠버네티스는 단순한 오케스트레이션 시스템이 아니다. 사실, 쿠버네티스는 오케스트레이션의 필요성을 없애준다. 오케스트레이션의 기술적인 정의는 A를 먼저 한 다음, B를 하고, C를 하는 것과 같이 정의된 워크플로우를 수행하는 것이다. 반면에, 쿠버네티스는 독립적이고 조합 가능한 제어 프로세스들로 구성되어 있다. 이 프로세스는 지속적으로 현재 상태를 입력받은 의도한 상태로 나아가도록 한다. 프로세스가 어떻게 동작했는지 상관이 없다. 중앙화된 제어도 필요치 않다. 이로써 시스템이 보다 더 사용하기 쉬워지고, 강력해지며, 견고하고, 회복력을 갖추게 되며, 확장 가능해진다.

**Reference**

* **https://kubernetes.io/ko/docs/concepts/overview/what-is-kubernetes/**
* **https://en.wikipedia.org/wiki/Kubernetes**
* **https://pdos.csail.mit.edu/6.824/papers/borg.pdf**
* **https://research.google/pubs/pub41684/**
* **https://medium.com/containermind/a-new-era-of-container-cluster-management-with-kubernetes-cd0b804e1409**