|  |
| --- |
|  |

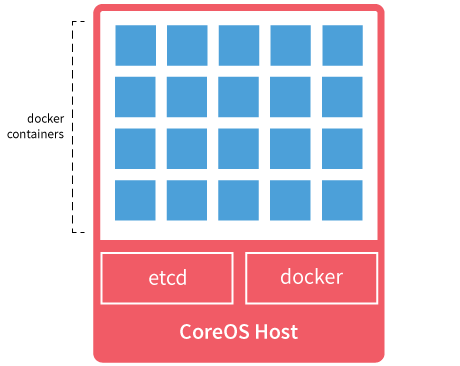
CoreOS는 크게 etcd, systemd, fleet 세 가지 컴포넌트로 구성되어 있습니다

**1.2 대규모 서버 배포를 위한 리눅스 (Linux for Massive Server Deployments)**

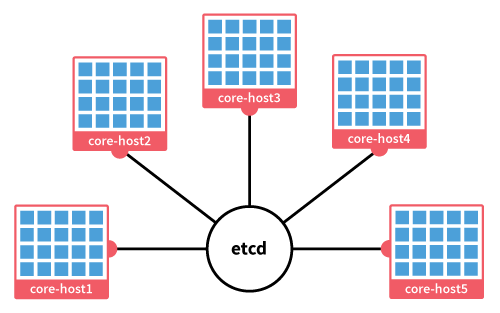
홈페이지를 방문하면 큼지막하게 위와 같은 모토를 확인할 수 있습니다. 이를 위해 CoreOS는 몇 가지 독자적인 콤포넌트를 제공합니다. 각각에 대해서는 후술하겠지만, 먼저 간략히 소개하자면 다음과 같습니다.

* fleet: 클러스터 레벨의 init 시스템
* etcd: fleet을 위한 분산 저장소
* docker: 리눅스 컨테이너 관리 시스템

개별 시스템(노트)의 그림을 그리자면 아래와 같습니다.



그리고 이런 다수개의 노드로 구성된 클러스터의 그림을 그리자면 아래와 같습니다.



즉, 분산 저장소인 etcd를 중심으로 개별 시스템의 fleet이 연결되고, 클러스터가 구성됩니다.  
fleet은 docker를 이용하여 애플리케이션을 구동하게 됩니다.

**2. 기술적 배경**

이해를 돕기 위해 CoreOS를 이루는 기술들을 나열해보도록 하겠습니다.

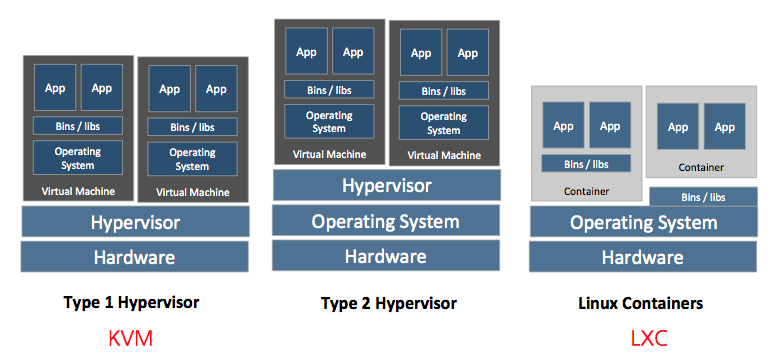
**2.1 Chrome OS**

CoreOS는 Chrome OS를 기본 베이스로 파생된 리눅스 배포판입니다.  
OS 업데이트에 사용되는 Ohama 프로토콜도 여기서 따 온 것으로 보입니다.

**2.2 Docker**

Docker는 오픈소스 리눅스 컨테이너 플랫폼입니다.  
먼저 컨테이너에 대해 짚고 넘어가야할 것 같은데요. 컨테이너는 쉽게 말해 ‘가벼운 가상 머신’이라고 생각하시면 될 것 같습니다.

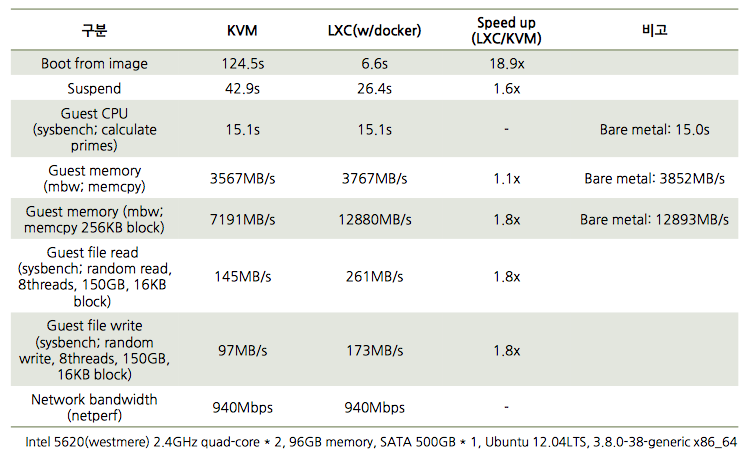
아래 그림은 일반적인 가상머신과 컨테이너와 차이를 보여주고 있습니다.



예컨대 OpenStack에서 주로 많이 사용하는 KVM이나 Xen의 경우 위 그림에서는 Type 1 Hypervisor를 사용하는 VM입니다.  
virtualbox나 VMWare fusion, parallels와 같은 솔루션은 Type 2 Hypervisor VM인데요. 두 타입 모두 가상 장비 개개별로, 즉 guest VM마다 OS 커널이 동작합니다.

lxc와 같은 컨테이너는 이와 달리 guest마다 OS 커널이 필요 없습니다. host의 OS를 공유하기 때문입니다. 때문에 일반적인 VM의 자원 분할과 같은 장점을 가지면서 개별 이미지 크기를 줄일 수 있습니다. 또한 guest 마다 별도의 OS 기동이 필요하지 않으므로, 순식간에 guest를 구동할 수 있는 장점이 있습니다. 하나의 계층을 줄여 성능을 높였지만, 물론 일반적인 VM만큼의 자원 격리(isolation)은 이루어지지 않습니다.

아래는 kvm과 lxc간 성능을 간단히 표로 정리한 것입니다.

  
(출처: <http://www.slideshare.net/BodenRussell/kvm-and-docker-lxc-benchmarking-with-openstack>)

CoreOS에서는 모든 애플리케이션이 Docker를 통해 구동됩니다. Docker 컨테이너 이미지 자체가 실행될 애플리케이션이기 때문에, CoreOS는 별도의 패키지 관리자를 가지고 있지 않습니다.

대신, 예를 들어 웹 서버인 nginx를 CoreOS에서 구동하려면, 저장소(private 저장소 or Docker public 저장소)에서 nginx Docker 이미지를 다운로드합니다. 그리고 이를 구동하여 실행합니다. 이 작업은 fleet을 통해 클러스터 스케일로 확장 가능한데요. 이는 이후 fleet에서 간단히 살펴보겠습니다.

**2.3 systemd**

systemd는 기존 init을 대체하는 리눅스의 새로운 PID 1 프로세스 시스템입니다. 그 역할은 모든 프로세스를 생성하는, 모든 프로세스의 부모입니다. 현재 빠르게 대부분 배포판의 PID 1 자리를 꿰어 차고 있습니다. CentOS/RHEL(7부터), Debian/Ubuntu와 같은 굵직한 배포판에서 대체되거나 대체 계획이 발표되고 있습니다.

기존 init과 다르게 자원 분할을 해주는 cgroups과 일종의 API로 UDS나 D-Bus 인터페이스도 제공하고 있습니다. 또, 프로세스의 헬스 체크가 가능하고, 실패한 서비스는 재구동하는 기능도 갖추고 있습니다.

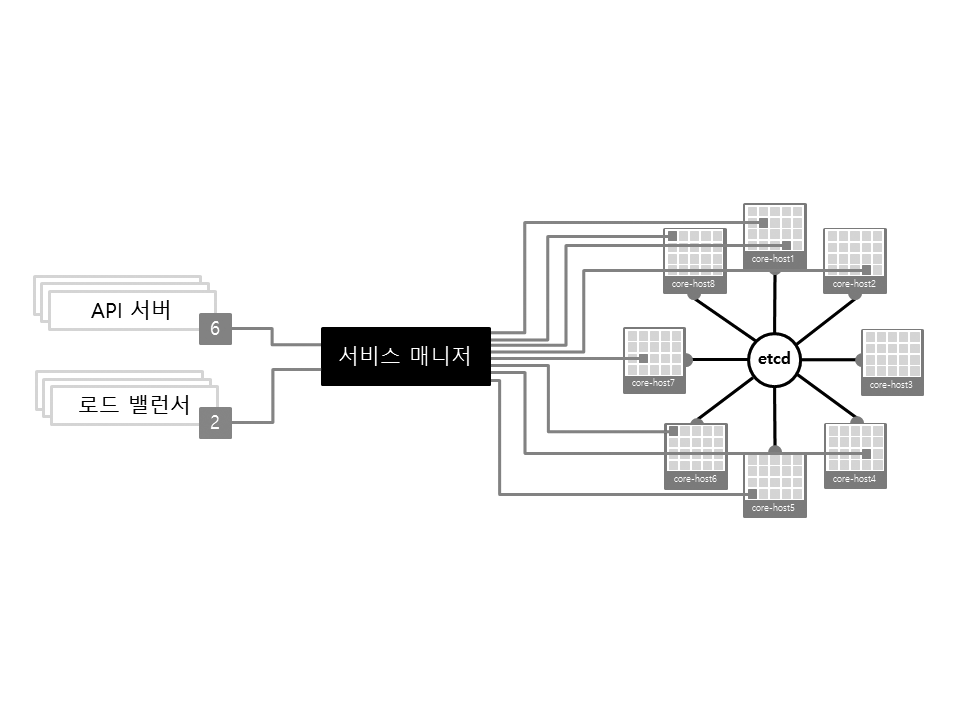
CoreOS fleet은 D-Bus 인터페이스(정확히는 IPC)를 통해 systemd에 Docker 이미지의 기동이나 중단을 명령합니다. 개개의 애플리케이션을 systemd에서는 unit이라고 표현하는데, fleet은 unit의 상태도 이 인터페이스를 통해 주기적으로 받아옵니다.

**2.4 fleet**

먼저도 언급했지만 fleet은 클러스터 레벨의 init 시스템이라 할 수 있습니다. CoreOS는 개별 시스템 레벨에서는 프로세스 관리를 위해 systemd를 사용하지만, 클러스터 레벨에서는 fleet을 사용합니다.

fleet은 etcd와 systemd를 이용한 분산 서비스 실행(init) 시스템 입니다. systemd는 로컬의 서비스를 관리하는 시스템이지만 fleet은 원격에서 여러 서버에 서비스를 실행할 수 있습니다.

그림 15-2는 fleet를 이용하여 API 서버 컨테이너 6개와 로드 밸런서 컨테이너 2개를 CoreOS 클러스터에 배포한 모습입니다. 클러스터 상황에 맞게 fleet가 알아서 호스트를 선택한 뒤 Docker 컨테이너를 배포합니다. 또는, 특정 호스트를 설정하여 컨테이너를 배포할 수도 있습니다.

  
**그림 15-2** fleet로 여러 서버에 Docker 컨테이너 배포(출처: https://coreos.com/using-coreos/clustering/)

다음은 fleet의 특징입니다.

* Docker 컨테이너를 임의의 호스트에 배포합니다.
* 특정 호스트에 서비스가 몰리지 않게 적절히 분산해줍니다.
* 특정 호스트에 장애가 발생해도 정해진 서비스 개수를 유지해줍니다. 즉 특정 호스트가 정지하면 해당 호스트에서 실행되던 서비스를 다른 호스트에서 실행해줍니다.
* 클러스터에 속한 호스트를 자동으로 발견(discover)합니다

개별 장비에 설치된 fleet 에이전트는 D-Bus를 통해 systemd에 접근, 현재 로컬 시스템에서 구동중인 서비스를 알 수 있습니다. 그리고 systemd를 통해, (그리고 다시 Docker를 통해) 애플리케이션을 구동할 수 있습니다.

fleet을 통해 애플리케이션을 구동하는 과정을 간단히 살펴보면 다음과 같습니다.

* systemd unit 파일을 명세: 실행하고자하는 동작을 명세 (e.g., docker run application)
* fleetctl 커맨드라인 툴로 etcd 디렉터리 서비스에 해당 unit 파일을 등재
* fleetctl 커맨드라인 틀로 클러스터에 애플리케이션(unit) 구동

보다 자세한 내용은 다음과 같은 링크에서 살펴볼 수 있습니다.  
: <https://www.digitalocean.com/community/tutorials/how-to-create-and-run-a-service-on-a-coreos-cluster>

unit(즉, 애플리케이션 or 서비스)를 클러스터 레벨에서 구동하려면, 클러스터를 이루는 각 노드들의 정보나 상태 등은 중앙 집중적인 저장소에 관리되고 있어야 합니다. 그리고, 그 중앙 집중 저장소는 단일 장애지점(single failure point)가 되므로 굉장한 고가용성(highly availability)가 요구됩니다.

fleet과 비슷한 플랫폼으로는 Apache Mesos(http://mesos.apache.org)가 있는데요. Mesos에서는 이 저장소로 zookeeper를 사용합니다. CoreOS에서는 다음 절에서 이야기 할 etcd를 사용합니다.

**2.5 etcd**

etcd는 go언어와 [Raft프레임워크](https://speakerdeck.com/benbjohnson/raft-the-understandable-distributed-consensus-protocol) 이용해 작성된 오픈소스 key-value 저장소로 대규모 Docker 클러스터링에 있어서 컨테이너들을 유기적으로 연동시키고 억세스하기 위한 세련된 아키텍처를 제공한다.  
etcd는 분산 키-값(Distributed Key-Value) 저장소이며 클러스터의 설정 값과 노드 정보를 저장하고 공유하는 시스템입니다. Apache Zookeeper, doozer와 비슷합니다

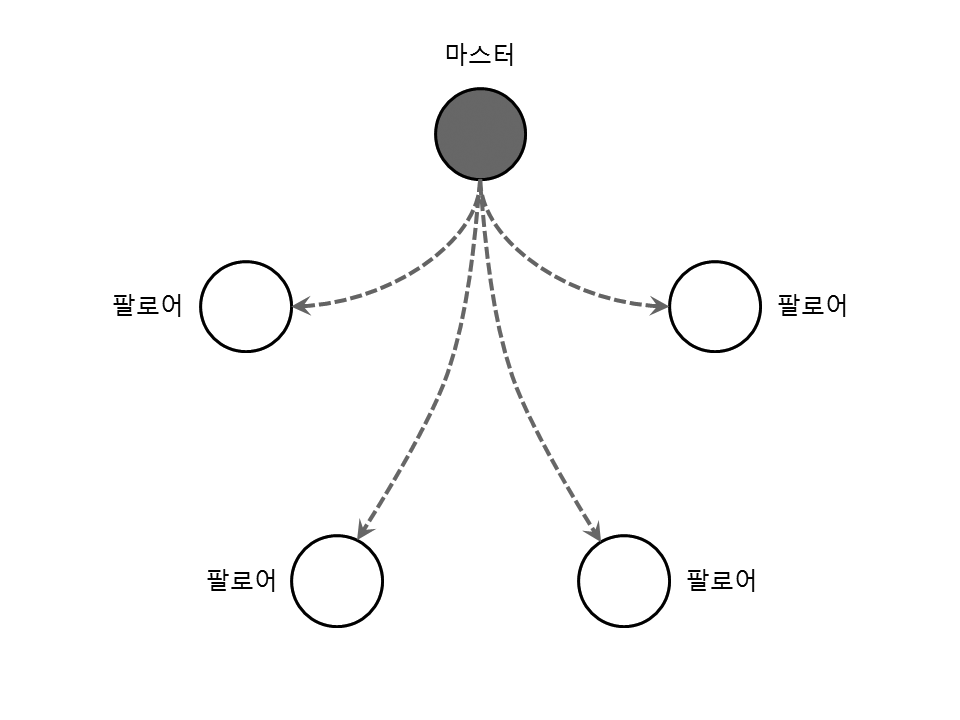
CoreOS는 클러스터 정보 관리를 위해 etcd라는 고가용성의 중앙 집중 저장소를 사용합니다.  
일반적으로 이러한 용도의 저장소에는 zookeeper가 사용되는데요. CoreOS에서는 etcd라는 별도의 솔루션을 사용합니다.

etcd는 Raft라는 Paxos 대안 알고리즘을 사용합니다. Raft는 2013년에 스탠퍼드의 Diego Ongaro가 고안한 컨센서스 알고리즘으로, Paxos와 동일한 능력을 가지되 사람에게 보다 쉽게 이해할 수 있도록 작성되어 있습니다.

etcd는 다음과 같은 특징을 가지고 있습니다.

* HTTP 프로토콜에 JSON 형식 API를 제공합니다.
* 초당 1000회 쓰기 성능을 제공합니다.
* Raft 컨센서스 알고리즘을 이용하여 여러 서버들 중에서 마스터를 선출합니다.
* 키 자동 삭제 기능(TTL, Time to live)을 제공합니다.
* 언제나 데이터의 일관성을 보장하는 Atomic 읽기/쓰기를 제공합니다.
* HTTP 롱 폴링(long-polling)을 통해 키 변경 사항을 감시할 수 있습니다.

etcd는 모든 키 변경 사항을 로그로 저장합니다. 마스터는 로그를 각 팔로어에 복제하여 데이터를 공유합니다. 마스터뿐만 아니라 각 노드에서도 키를 추가하거나 값을 변경하면 모든 노드에 반영됩니다.

  
**그림 15-1** 클러스터의 각 팔로어에 로그(데이터) 복제(출처: https://coreos.com/using-coreos/etcd/)

**2.6 Go language**

CoreOS는 fleet, etcd 등의 구현을 모두 Go 언어로 통합하고 있습니다. (저도 참 좋아하는데요.) Go 언어는 그 유명한 Ken Thompson, Rob Pike가 2007년에 구글에서 만든 언어입니다.

컴파일시 타입 추론(compile-time type inference), 빌트인 동시성 프리미티브(e.g., channel, light-weight thread;goroutine) 등을 제공하는 최신 세대의 언어입니다.

요새들어 devops들에게 각광받고 있는데요. 파이썬만큼 간단한 문법, 이미 대부분 기능이 빌트인된 현대적인 표준 라이브러리, 파이썬보다 훨씬 좋은 성능(컴파일 언어이므로), 정적 컴파일(동적 라이브러리 같이 배포할 필요 없고, 바이너리 한개만 배포하면 됨) 등이 그들에게 큰 점수를 받은 것 같습니다.

이 언어의 창시자인 Rob Pike는 점점 복잡해지는 C++ 개발자를 위해 만들었다지만, 시장에서는 파이썬 개발자에게 큰 어필을 하고 있습니다.

CoreOS는 특히나 minimal OS를 지향했고, Go 언어는 특히나 minimal 배포를 그 강점으로 삼고 있으므로 궁합이 잘 맞습니다.