

АННОТАЦИЯ

Цель выпускной квалификационной работы – исследование и создание кластерной инфраструктуры, которая позволит разрабатывать и развертывать программные проекты множеством команд из нескольких человек.

В работе составлен обзор по архитектурам приложений и инфраструктур. Произведен анализ инструментов для создания инфраструктуры.

В ходе работы в рамках проекта был развернут кластер Kubernetes, его вспомогательные компоненты, настроено их взаимодействие.

Созданная инфраструктура внедрена в МИЭМ НИУ ВШЭ и происходит поэтапный процесс переноса проекта в нее.

В структуру работы входит введение, ??? глав, заключение, список использованных источников , 1 рис. и 5 источн.

Общий объем проекта составляет 23 стр.

ABSTRACT

blabla

СОДЕРЖАНИЕ

Введение	6
СТАРАЯ Терминология	7
1 Исследовательская часть работы	9
1.1 Общие сведения о архитектуре инфраструктуры и приложений в ней	9
1.2 Область применения	19
1.3 Обзор литературы	19
1.4 Разработка архитектуры и выбор инструментария	19
1.5 Выстроение этапов реализации инфраструктуры	19
2 Практическая часть работы	20
2.1 GitLab	20
2.2 Kubernetes	20
2.3 NFS	20
2.4 Traefik	20
2.5 CI/CD	20
2.6 Certs	20
3 Документация	21
3.1 Инструкция для пользователя	21
3.2 Руководство администратора	21
3.3 Документация разработчика	21
Заключение	22
Список использованных источников	23

ОПРЕДЕЛЕНИЯ

Block input and output — обращение на считывание и запись с хранителей данных.

Central processing unit — процессор.

Control group — управление (контроль) групп.

DLL Hell — сбой, возникающий, когда одна часть программного обеспечения ведет себя не так, как ожидалось второй частью программного обеспечения, что в некотором роде зависит от действия первого [3].

Docker — программное обеспечение для автоматизации развертывания и управления приложениями в среде виртуализации на уровне операционной системы [4].

Image или образ контейнера — архив файловой системы, который используется для запуска контейнера из него.

Memory — память.

Mount — монтирование, например дисков как директорий. Способ подключения различных хранителей данных в OS GNU/Linux.

Namespaces — пространства имен.

Networking или network — сеть.

Process ID — идентификатор процесса.

Resource management — управление и ограничение ресурсов машины.

User — пользователь.

Виртуальная машина — программная и/или аппаратная система, эмулирующая аппаратное обеспечение некоторой платформы (target — целевая, или гостевая платформа) и исполняющая программы для target-платформы на host-платформе (host —

хост-платформа, платформа-хозяин) или виртуализирующая некоторую платформу и создающая на ней среды, изолирующие друг от друга программы и даже операционные системы [1].

Гипервизор — программа или аппаратная схема, обеспечивающая или позволяющая одновременное, параллельное выполнение нескольких операционных систем на одном и том же хост-компьютере [2].

Инстанс — экземпляр объекта.

ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

VM — virtual machine (виртуальная машина).

OS или ОС — операционная система.

ПО — программное обеспечение.

PID — Process ID.

Cgroup — control group.

CPU — central processing unit.

I/O — input and output.

ВВЕДЕНИЕ

В настоящее время существует спрос на реализацию решений по автоматизации процессов разработки и развертывания программных продуктов. Для этого используются такие средства, как:

- Система контроля версий (Version Control System, далее – VCS).
- Средства непрерывной интеграции и доставки артефактов (Continuous Integration, Continuous Delivery или Deployment, далее – CI/CD).
- Система контейнеризации.
- Оркестрация контейнеров.
- Вспомогательные инструменты.

Цель работы – исследование и создание кластерной инфраструктуры на основе контейнеров.

СТАРАЯ ТЕРМИНОЛОГИЯ

Immutable - неизменный.

Self-healing - самолечение.

Declarative - декларативность.

SLA - доступность.

Daemon или **Демон** - процесс, запущенный в фоне.

CLI (Command line interface) - консольный интерфейс приложения.

FS (File system) - файловая система раздела.

Hash или **Хэш** - результат выполнения некоторой математической хэш-функции.

Tag или **Тэг** - некоторая отметка на чем-либо для более удобного поиска объектов.

Yaml (YAML Ain't Markup Language) - "дружественный" формат сериализации данных, концептуально близкий к языкам разметки.

SSD (Solid State Drive) или *Твердотельный диск* - вид накопителя, хранящего данные.

REST API - архитектурный стиль взаимодействия компонентов распределенного приложения в сети.

Garbage Collector - "сборщик мусора".

QoS (Quality of Service) - ограничения на что-либо, разграничивающие по ролям.

Requested resources - запрошенные ресурсы.

Netwrok Policies - сетевые политики.

Iptables - обертка над netfilter.

Ipv6 - обертка над netfilter.

OSI - сетевая модель стека сетевых протоколов.

VRRP (Virtual Router Redundancy Protocol) - сетевой протокол, предназначенный для увеличения доступности маршрутизаторов.

SSL/TLS (Secure Sockets Layer) (Transport Layer Security) - криптографические протоколы, обеспечивающие защищенную передачу данных между узлами в сети Интернет.

Hook - перехват чего-либо.

Git - одна из систем контроля версий.

HTTP (HyperText Transfer Protocol) - протокол прикладного уровня передачи данных.

Tar - архиватор.

GET Requests - метод *HTTP*.

1 Исследовательская часть работы

1.1 Общие сведения о архитектуре инфраструктуры и приложений в ней

На текущий момент существует множество решений со стороны архитектуры инфраструктуры и, как следствие, приложений в ней. Для выбора конкретного решения под вышеуказанные **Указать цели выше** цели необходимо рассмотреть основные из них. Далее они будут описаны по хронологии появления.

а) Монолитная эра. Ей свойственны следующие аспекты:

- Приложения монолитные.
- Куча зависимостей.
- Долгая разработка до релиза.
- Все инстансы известны по именам.
- Использование виртуализации. Это означает:
 - Один сервер – несколько VM [1].
 - Resource Management.
 - Изоляция окружений.

Соответственно использовались VM:

- VMWare.
- Microsoft Hyper-V.
- VirtualBox.
- Qemu.

Подход был следующий: один большой сервер делили на несколько виртуальных машин. Это давало полную изоляцию, но недостатками были:

- Hypervisor [2].
- Большие образы.
- Как следствие больших образов с разным ПО – медленное управление VM.

б) На смену им пришла виртуализация на уровне ядра с помощью следующих инструментов:

- OpenVZ.
- Systemd-nspawn.
- LXC.

Но остались прежние проблемы:

- Большие образы с OS с большим количеством ПО.
- Нет стандарта упаковки и доставки.
- DLL Hell [3].

в) Но далее пришли контейнеры. Разница между VM и контейнером:

- Виртуальная машина подразумевает виртуализацию железа для запуска гостевой ОС.
- Контейнер использует ядро хостовой ОС.
- В VM может работать любая ОС.
- В контейнере может работать только GNU/Linux (с недавних пор и Windows).
- VM хороша для изоляции.
- Контейнеры не подходят для изоляции.

В итоге мы приходим к ситуации, изображенной на (Рис. 1.1):

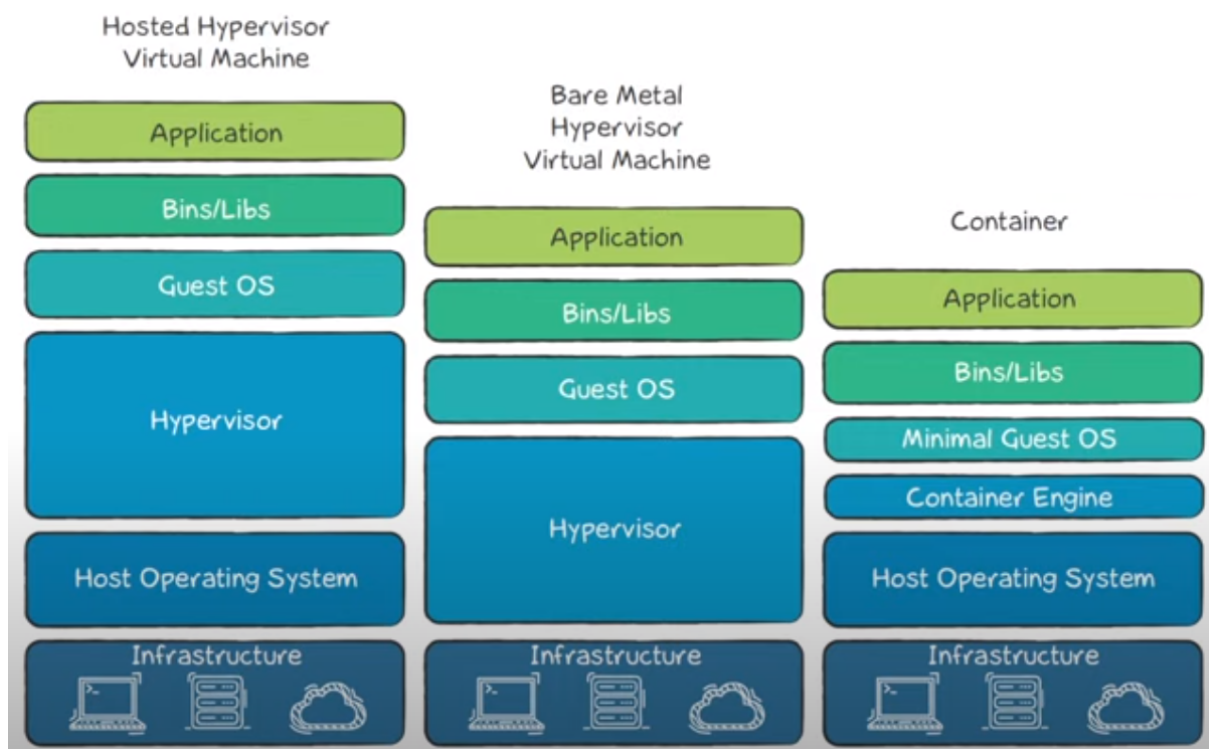


Рисунок 1.1 — Сравнение VM и контейнеров

Способ реализации контейнеризации:

- Namespaces:
 - PID.
 - Networking.
 - Mount.
 - User;
- Control Groups:
 - Memory. .
 - CPU
 - Block I/O;
 - Network;

В итоге получается следующая логика:

- Один процесс – один контейнер.
- Все зависимости в контейнере.
- Чем меньше образ контейнера – тем лучше.
- Инстансы становятся эфемерными.

И в 2014-2015 годах пришел расцвет Docker [4], который:

- Меняет философию подхода к разработке.
- Стандартизует упаковку приложения.
- Решает вопрос зависимостей.
- Гарантирует воспроизводимость.
- Обеспечивает минимум дополнительных средств для использования.

По своей сути Docker – это обертка или надстройка над Namespaces и Cgroup, которые существуют как родные инструменты

изоляции в OS GNU/Linux, которые упрощают запуск процесса в изолированном пространстве.

!!! Переписать

Выбранный стек технологий является одним из основных на текущий момент для разработки и развертывания приложений во всем мире. Но отдельного упоминания заслуживают причины, которые привели к этой ситуации:

1.1.0.1 Docker

Docker - это программное обеспечения для автоматизации развертывания и управления приложениями в средах с поддержкой контейнеризации. *Но что это значит?* Для понимания этого инструмента необходимо посмотреть, что было до него:

- Монолитная эра:
 - Приложения монолитные;
 - Куча зависимостей;
 - Долгая разработка до релиза;
 - Все инстансы известны по именам;
 - Использование виртуализации:
 - Один сервер - несколько *VM*;
 - *Resource Management*;
 - Изоляция окружений.

Соответственно использовались *VM*:

- VMWare;
- Microsoft Hyper-V;
- VirtualBox;
- Qemu.

Подход был такой: один большой сервер делили на несколько виртуальных машин. Это давало полную изоляцию, но недостатками были:

- Hypervisor;
- Большие образы;
- И как следствие больших образов с разным ПО - медленное управление *VM*.
- На смену им пришла виртуализация на уровне ядра:
 - OpenVZ;
 - Systemd-nspawn;
 - LXC.

Но остались прежние проблемы:

- Большие образы с *OS* с большим количеством ПО.
- Нет стандарта упаковки и доставки.
- *DLL Hell*.
- Но далее пришли контейнеры. Разница между *VM* и контейнером:
 - Виртуальная машина подразумевает виртуализацию железа для запуска гостевой ОС.
 - Контейнер использует ядро хостовой ОС.
 - В *VM* может работать любая ОС.
 - в контейнере может работать только GNU/Linux (с недавних пор и Windows).
 - *VM* хороша для изоляции.
 - Контейнеры не подходят для изоляции.

Способ реализации контейнеризации:

- *Namespaces*:
 - *PID*;
 - *Networking*;
 - *Mount*;
 - *User*;
 - *etc.*
- Control Groups:
 - Memory;
 - CPU;
 - Block *I/O*;
 - *Network*;
 - *etc.*

В итоге получается следующая логика:

- Один процесс - один контейнер.
- Все зависимости в контейнере.
- Чем меньше образ контейнера - тем лучше.
- Инстансы становятся эфемерными.

И в 2014-2015 годах пришел расцвет *Docker*, который:

- Меняет философию подхода к разработке.
- Стандартизует упаковку приложения.
- Решает вопрос зависимостей.
- Гарантирует воспроизводимость.

— Обеспечивает минимум дополнительных средств для использования.

По своей сути *Docker* - это обертка или надстройка над *Namespaces* и *Cgroup*, которые существуют как родные инструменты изоляции в *OS GNU/Linux*, которые упрощают запуск процесса в изолированном пространстве.

1.1.0.2 Kubernetes

Kubernetes - оркестратор контейнеров. *Зачем он нужен?* У нас уже есть *Docker*, который удобен для разработки и развертки приложений, особенно в формате микросервисов. Его можно сравнить с фурой, на которую грузят один контейнер и которая его везет. Но этого зачастую недостаточно. Есть *Docker Compose*, который может запускать несколько контейнеров, взаимодействующих друг с другом и внешним миром. Его можно сравнить с паровозом, который может везти несколько контейнеров. В свое время *Kubernetes* правильно сравнивать с морским портом, где находится огромное количество контейнеров, которые распределены по разным складам, перемещаются между складами при необходимости, чинит их при поломке. То есть, вне зависимости от масштаба и количества контейнеров, *Kubernetes* позволяет их гибко разварачивать, управлять, перемещать и масштабировать. Его основные преимущества:

— *Immutable* - неизменяемая структура. Эта идея уже была реализована в *Docker* касательно *Docker Image* - он не изменяется в процессе работы. В *Kubernetes* же неизменна вся структура.

— *Self-healing* - каждый компонент отвечает за свою часть инфраструктуры и постоянно поддерживает ее в актуальном состоянии. Например, при падении одного из нескольких серверов те контейнеры, что были запущены на упавшем сервере, будут запущены на одном из оставшихся в живых сервере.

— *Declarative* - описание не того, что надо сделать (императивный подход), а как должен выглядеть итоговый результат. В случае *Kubernetes* все управление происходит посредством *Yaml* манифестов.

— Каждый компонент инфраструктуры независим от других и полагается на их *SLA*.

По своей сути *Kubernetes* - это надстройка над *Docker*, которая позволяет управлять огромным количеством контейнеров, распределенных на разных серверах.

1.1.0.3 Helm

Helm - инструмент, который помогает управлять приложениями в *Kubernetes*. Но зачем нам еще помощь, когда уже есть *Docker* и *Kubernetes*, которые сильно упрощают жизнь? Для этого надо посмотреть на логику работы с приложением. Допустим есть приложение, которое состоит из нескольких *YAML* манифестов для *Kubernetes*, которые уже настроены, и их можно применить с помощью *kubectl apply -f path/to/dir*. Вроде все хорошо. Но ведь приложение растет, оно может начать занимать сотни *YAML* манифестов, а помнить все переменные окружения и прочие вещи станет достаточно сложно. Самый простой пример: замена *labels* и *selector* во всех манифеста одного микросервиса. Есть варианты использовать *sed* или *ansible*, и они решат проблему запуска чего-то нового. Но что делать, если что-то пошло не так? То есть как откатить на предыдущую версию или как контролировать процесс релиза? И тут на помощь приходит *Helm*. Он при применении создает артефакт с указанной версии и сохраняет его где-то. И в случае, если что-то пошло не так, возможно взять сохраненный предыдущий артефакт и запустить из него стабильную версию. При этом создатели *Helm* позиционируют его как "пакетный менеджер с помощью которого можно скачать, установить, удалить

приложение вместе с его зависимостями. Он, как и сам *Kubernetes*, декларативный. Также, он умеет не просто применить *YAML* манифесты, но и отследить процесс запуска всего необходимого, а в случае ошибки, откатиться назад автоматически.

1.2 Область применения

Для большого количества проектов с множеством команд

1.3 Обзор литературы

Обзор на статьи Фланта, SouthBridge etc

1.4 Разработка архитектуры и выбор инструментария

Картинки с разными уровнями погружений и т.д. в новую архитектуру с пояснениями

1.5 Выстроение этапов реализации инфраструктуры

Что-то типа roadmap

2 Практическая часть работы

2.1 GitLab

2.2 Kubernetes

2.3 NFS

2.4 Traefik

2.5 CI/CD

2.6 Certs

3 Документация

3.1 Инструкция для пользователя

3.2 Руководство администратора

3.3 Документация разработчика

ЗАКЛЮЧЕНИЕ

[5] test

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Wikipedia. Виртуальная машина — Википедия. — 2021. — Режим доступа: https://ru.wikipedia.org/w/index.php?title=%D0%92%D0%B8%D1%80%D1%82%D1%83%D0%B0%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D0%BC%D0%B0%D1%88%D0%B8%D0%BD%D0%B0&stable=1 (дата обращения: 07.05.2021).
2. Wikipedia. Гипервизор — Википедия. — 2021. — Режим доступа: <https://ru.wikipedia.org/wiki/%D0%93%D0%B8%D0%BF%D0%B5%D1%80%D0%B2%D0%B8%D0%B7%D0%BE%D1%80> (дата обращения: 07.05.2021).
3. Dick Stephanie, Volmar Daniel. DLL hell: software dependencies, failure, and the maintenance of microsoft windows // IEEE Annals of the History of Computing. — 2018. — Vol. 40, no. 4. — P. 28–51.
4. Демидова ТС, Соболев АА. Использование Docker для развёртывания вычислительного программного комплекса // Информационно-телекоммуникационные технологии и математическое моделирование высокотехнологичных систем. — 2019. — P. 432–435.
5. Пупкин Василий, Эйнштейн А. ЛАТЭХ для «чайников». — М., 2009. — 299 с.