

**МИНОБРНАУКИ РОССИИ**

**федеральное государственное бюджетное образовательное учреждение**

**высшего образования**

**«Московский государственный технологический университет «СТАНКИН»**

**(ФГБОУ ВО «МГТУ «СТАНКИН»)**

|  |  |
| --- | --- |
| **Институт**  **информационных систем**  **и технологий** | **Кафедра**  **информационных систем** |

**Уланович Илья Дмитриевич**

**Тема**: **«Визуализция серверной инфраструктуры телекоммуникационной компании на основе докер-контейнеров »**

**Выпускная квалификационная работа на присвоение квалификации «бакалавр» по направлению 09.03.02 «Информационные системы и технологии»**

|  |  |  |
| --- | --- | --- |
| Заведующий кафедрой  д.т.н., проф. | \_\_\_\_\_\_\_\_\_\_\_\_\_\_ **Позднеев Б.М.**  подпись |  |
| Научный руководитель  к.т.н., доц. | \_\_\_\_\_\_\_\_\_\_\_\_\_\_ **Тясто С.А.**  подпись |  |
| Студент  группы ИДБ-15-13 | \_\_\_\_\_\_\_\_\_\_\_\_\_\_ **Уланович.И.**  подпись |  |
|  |  |  |
|  |  |  |

Регистрационный № \_\_\_\_\_\_\_\_

[ВВЕДЕНИЕ 2](#_Toc34492254)

ВВЕДЕНИЕ

Стремительное развитие информационных технологий вывело организационную структуру предприятий на новый уровень. На сегодняшний день каждая даже самая малая компания обязана иметь свои сервера и какую то либо бы базу данных и это не говоря о том, что многие компании не могут существовать без такой структуры.

Серверная инфраструктура — специализированное оборудование, предназначенное для сбора, обработки и хранения большого количества информации. На устройства устанавливается специализированный софт, который обеспечит функционирование предприятия или компании. Сбои в работе данных систем парализуют работу фирмы или предприятия. Так же в зависимости от важности хранения данных необходимо обеспечить должный уровень защиты. А если на этих серверах должны храниться персональные данные пользователей, то нужно не просто обеспечить информационную безопасность, а получить разрешения у фсб на хранение в данном месте данные такого типа.

Нормальной практикой для компаний считается иметь свой штат системных администраторов, которые вводят в строй новые сервера, устанавливают обновления, решают проблемы с серверами и т.д. Чем шире перечень серверных операционных систем, тем больше системных администраторов необходимо иметь в штате.

Для решения задач поддержки серверной инфраструктуры, как правило, необходимо обеспечить круглосуточный мониторинг и обслуживание серверов, а также организовать группу специалистов для восстановления работоспособности оборудования. Обычно эти мероприятия требуют выделения большого объема ресурсов, что дорого и не всегда эффективно.

При рассмотрении вопросов поддержки серверной инфраструктуры ключевым становится разработка и реализация серверной архитектуре компании - чем проще архитектура (однотипные устройства, стандартизированные системы, описанные настройки и т.д.), тем меньше нужно ресурсов на ее поддержку, а значит – тем меньше затраты на поддержку.

Особое место в модернизации существующей серверной архитектуры и поддержке серверной инфраструктуры занимает виртуализация. В данной дипломной работе будут использоваться сторонние открытые ресурсы виртуализации draw.io в котором будет о отрисована вся структура на прикладном уровне и Portainer. Portainer — это удобное управленческое решение для докеров, позволяющее визуализировать и управлять ими с помощью удобного веб интерфейса.

Таким образом, была поставлена цель автоматизации серверной инфраструктуры телекоммуникационной компании на основе докер-контейнеров

Для достижения поставленной цели, необходимо решить следующие задачи:

1. Проанализировать современное open source программное обеспечение для маршрутизации. Выбрать из имеющихся сборок одну на dockehub;
2. Визуализировать структуру которую необходимо будет построить;
3. Запустить вручную один контейнер;
4. Автоматизировать контроль работы контейнеров в yml файле;
5. Произвести тестовое подключение к внешнему оператору;

Глава1. Анализ современных средств программной маршрутизации на основе докер-контейнеров.

1.1. Основные характеристики средств программной маршрутизации.

Программная маршрутизация выполняется либо специализированным ПО маршрутизаторов (в случае, когда аппаратные методы не могут быть использованы, например, в случае организации туннелей), либо программным обеспечением на компьютере. В общем случае, любой компьютер осуществляет маршрутизацию своих собственных исходящих пакетов (как минимум, для разделения пакетов, отправляемых на шлюз по умолчанию и пакетов, предназначенных узлам в локальном сегменте сети). Для маршрутизации чужих IP-пакетов, а также построения таблиц маршрутизации используется различное ПО:

Сервис RRAS ([анг.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) routing and remote access service) в Windows Server

Домены routed, gated, quagga в Unix-подобных операционных системах (Linux, FreeBSD и т. д.)

Маршрутизация в сети Интернет основана на протоколах TCP/IP.

Передача информации осуществляется с помощью IP-пакетов, заголовок каждого IP-пакета содержит IP-адрес получателя и отправителя пакета. Каждый пакет обрабатывается маршрутизатором в соответствии с его таблицей маршрутизации. Таблица, в свою очередь, содержит информацию, компьютеру с каким адресом направлять пакеты с тем или иным диапазоном адресов. Например, все пакеты определённого диапазона могут направляться другому маршрутизатору, который «отвечает» за этот сегмент.

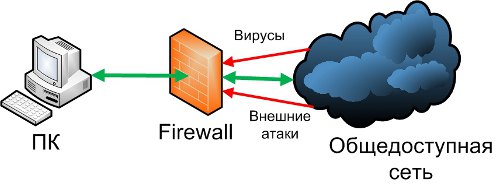
В ряде случаев маршрутизатор может преобразовывать заголовок пакета, заменяя адреса отправителя и/или получателя пакета. В частности, это происходит при взаимодействии локальной сети (имеющей свои адреса) с глобальной сетью Интернет. В этом случае локальная сеть может быть видна извне по одному глобальному IP-адресу. Для того, чтобы маршрутизатор мог направлять пакеты с одним глобальным адресом тем или иным получателям в локальной сети, используется таблица NAT, где помимо IP-адресов указываются порты, идентифицирующие приложения, устанавливающие соединение. При этом номера портов указаны не в заголовке IP-пакета, а в заголовке сегмента TCP либо UDP (сегменты инкапсулируются в поле данных IP-пакетов). Это позволяет осуществлять взаимно-однозначную идентификацию получателя и отправителя в тех случаях, когда за одним глобальным адресом находится множество компьютеров локальных сетей.

Пример таблицы NAT:

|  |  |
| --- | --- |
| 209.165.200.226:1444 | 192.168.1.15:1444 |
| 209.165.200.226:1445 | 192.168.1.16:1444 |

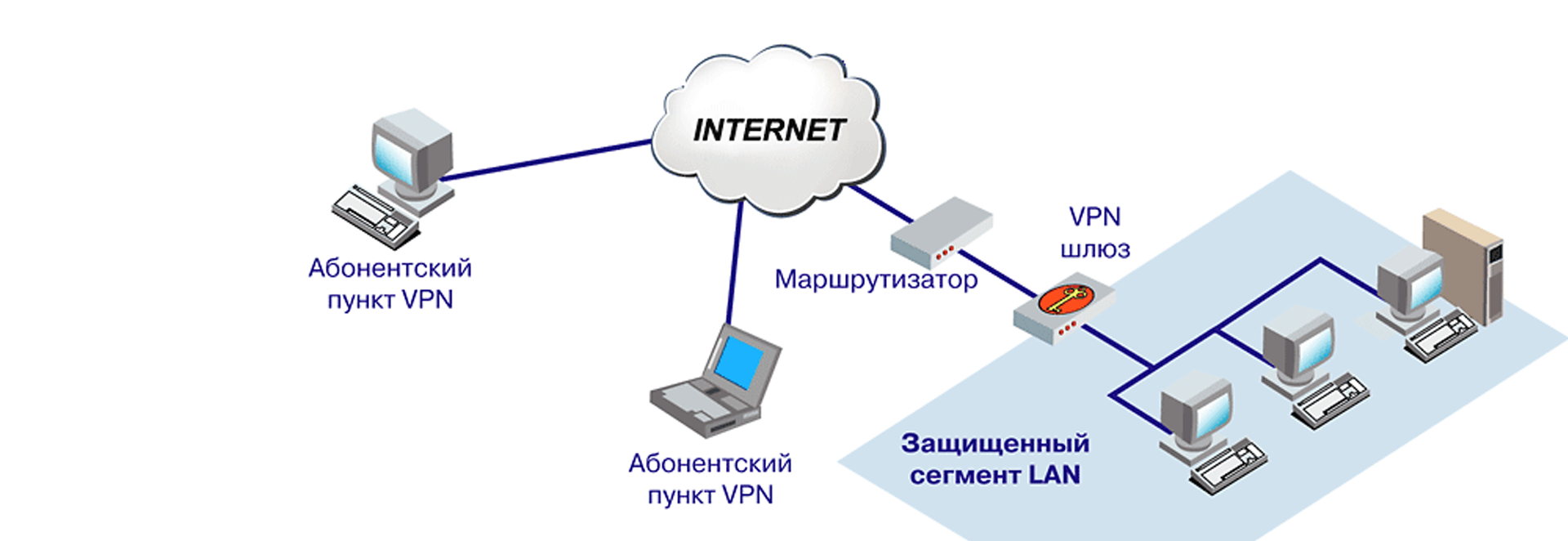
Межсетево́й экра́н, сетево́й экра́н — программный или программно-аппаратный элемент компьютерной сети, осуществляющий контроль и фильтрацию проходящего через него сетевого трафика в соответствии с заданными правилами.

Среди задач, которые решают межсетевые экраны, основной является защита сегментов сети или отдельных хостов от несанкционированного доступа с использованием уязвимых мест в протоколах сетевой модели OSI или в программном обеспечении, установленном на компьютерах сети. Межсетевые экраны пропускают или запрещают трафик, сравнивая его характеристики с заданными шаблонами.



Наиболее распространённое место для установки межсетевых экранов — граница периметра локальной сети для защиты внутренних хостов от атак извне. Однако атаки могут начинаться и с внутренних узлов — в этом случае, если атакуемый хост расположен в той же сети, трафик не пересечёт границу сетевого периметра, и межсетевой экран не будет задействован. Поэтому в настоящее время межсетевые экраны размещают не только на границе, но и между различными сегментами сети, что обеспечивает дополнительный уровень безопасности.

VPN ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) Virtual Private Network «виртуальная частная сеть») — обобщённое название технологий, позволяющих обеспечить одно или несколько сетевых соединений (логическую сеть) поверх другой сети (например [Интернет](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D0%BD%D0%B5%D1%82)). Несмотря на то, что коммуникации осуществляются по сетям с меньшим или неизвестным уровнем доверия (например по публичным сетям) уровень доверия к построенной логической сети не зависит от уровня доверия к базовым сетям благодаря использованию средств криптографии (шифрования, аутентификации, [инфраструктуры открытых ключей](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%84%D1%80%D0%B0%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%BE%D1%82%D0%BA%D1%80%D1%8B%D1%82%D1%8B%D1%85_%D0%BA%D0%BB%D1%8E%D1%87%D0%B5%D0%B9), средств для защиты от повторов и изменений передаваемых по логической сети сообщений).



При должном уровне реализации и использовании специального программного обеспечения сеть VPN может обеспечить высокий уровень шифрования передаваемой информации. При правильной настройке всех компонентов технология VPN обеспечивает [анонимность](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%BE%D0%BD%D0%B8%D0%BC%D0%BD%D0%BE%D1%81%D1%82%D1%8C) в Сети.

В данном проекте ПО маршрутизатор должен обладать следующим функционалом.

* межсетевой экран для IPv4 и IPv6, включая фильтрацию p2p-трафика;  
  трансляция сетевых адресов (NAT);
* система обнаружения вторжений;
* балансировка нагрузки и резервирование канала;
* резервирование маршрутизаторов с синхронизацией таблицы состояний соединений;
* виртуальные частные сети (IPsec, L2TP/IPsec, PPTP, OpenVPN);
* учёт трафика (Netflow и sFlow);
* Система единой конфигурации.
* Бесплатен.

1.2. Существующие системы.

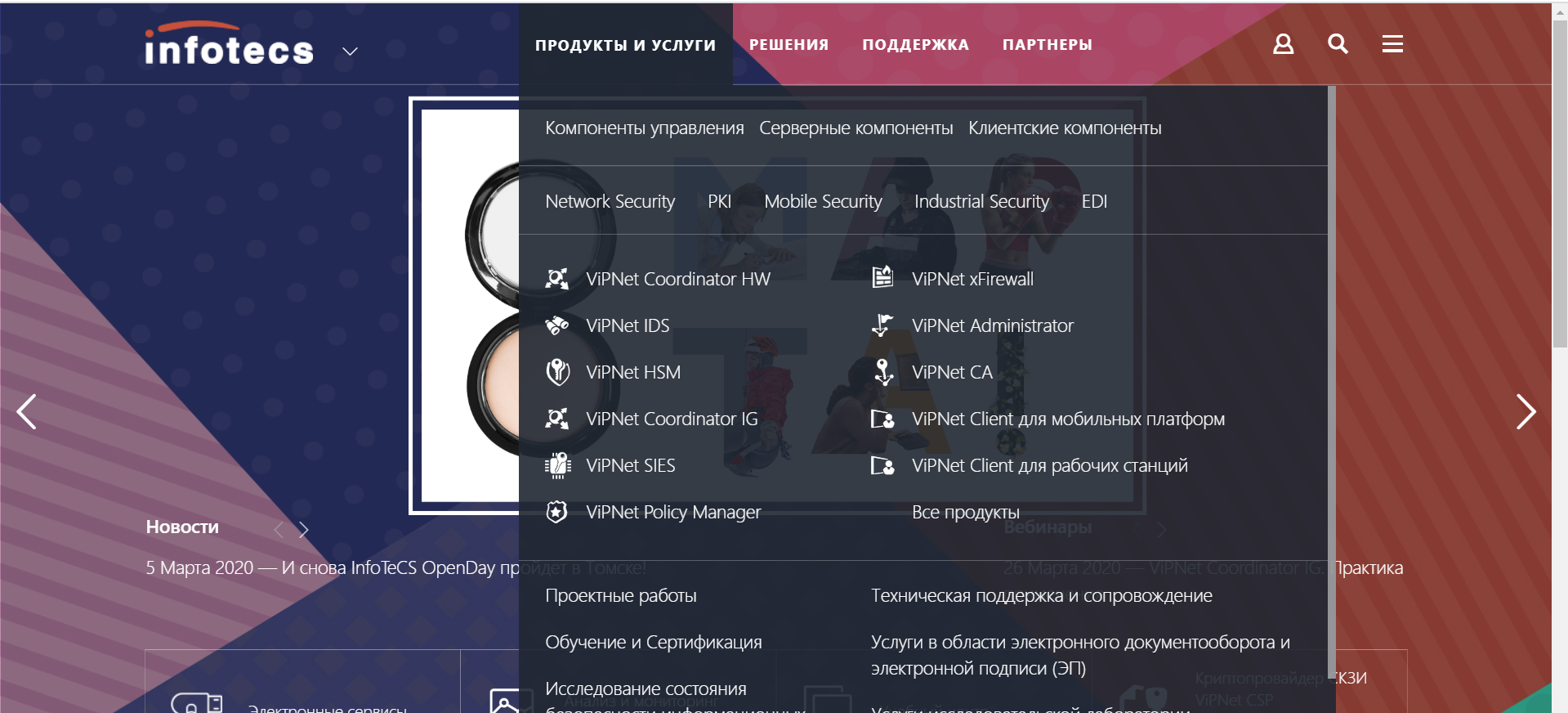
На данный момент существует множество основных решений программных межсетевых экранов: ViPNet Coordinator, Strongswan, Vyos.

Начнем с ViPNet CoordinatoR- семейство шлюзов безопасности, входящих в состав продуктовой линейки ViPNet Network Security.

Это сертифицированный продукт позволяет производить все необходимые нам настройки и поддерживает высокую безопасность сети при должном контроле. Благодаря тому что он имеет такие сертификаты как:

* [Сертификат соответствия ФСБ России № СФ/124-3657 от 20.03.2019](https://infotecs.ru/include/pages/withpdf.detail.php?ELEMENT_ID=12082&IBLOCK_ID=14)
* [Сертификат соответствия ФСБ России №СФ/118-3510](https://infotecs.ru/include/pages/withpdf.detail.php?ELEMENT_ID=11876&IBLOCK_ID=14)
* [Сертификат соответствия ФСБ России № СФ/124-3430 ПК ViPNet Client 4](https://infotecs.ru/include/pages/withpdf.detail.php?ELEMENT_ID=11771&IBLOCK_ID=14)
* [Сертификат соответствия ФСБ России № СФ/124-3431 на ПК ViPNet Coordinator 4](https://infotecs.ru/include/pages/withpdf.detail.php?ELEMENT_ID=11907&IBLOCK_ID=14)
* [Сертификат соответствия ФСБ России № СФ/124-3563 на программный комплекс ViPNet Coordinator 4 for Linux](https://infotecs.ru/include/pages/withpdf.detail.php?ELEMENT_ID=11936&IBLOCK_ID=14)
* [Сертификат соответствия ФСБ России № СФ/124-3674 на ПАК ViPNet Coordinator HW 4](https://infotecs.ru/include/pages/withpdf.detail.php?ELEMENT_ID=12140&IBLOCK_ID=14)

Компания INFOTECS может просить за этот продукт большие деньги для подключение к государственным структурам. Поэтому по факторы цены данный кандидат не подходит.



Следующий кандидат это Strongswan. StrongSwan является демоном IPSEC, который поддерживает IKEv1 и IKEv2. На данный момент это развивающий продукт. Установка StrongSwan может быть выполнена из исходников или репазитория.



Файлы конфигурирования по умолчанию хранятся в директории /etc/ и имеют следующие названия:

ipsec.conf – определяет параметры IPSEC-соединений и параметры подключений в целом;

ipsec.secrets – служит для хранения ссылок на сертификаты и ключи аутентификации;

strongswan.conf – для подключения криптографических алгоритмов и дополнительных функций.  
Помимо этого во время установки программного обеспечения для хранения сертификатов и CRL-файлов используемых демонами pluto и charon создается директория /etc/ipsec.d, в которой находятся следующие каталоги:

* private – содержит закрытые ключи RSA и ECDSA;
* certs – содержит сертификаты X.509 и PGP;
* crls – хранит список отозванных сертиифкатов;
* cacerts – хранит доверенные сертификаты CA;
* ocspcerts – содержит подписанные OCSP сертификаты;
* reqs – содержит запросы на сертификаты в формате PKCS#10.

Файл /etc/ipsec.secrets содержит неограниченное количество следующих типов ключей (паролей):RSA для определения пароля к сертификату открытого ключа;ECDS для определения пароля к сертификату открытого ключа;

PSK для определения Pre-shared ключа;

EAP для учетных записей EAP;

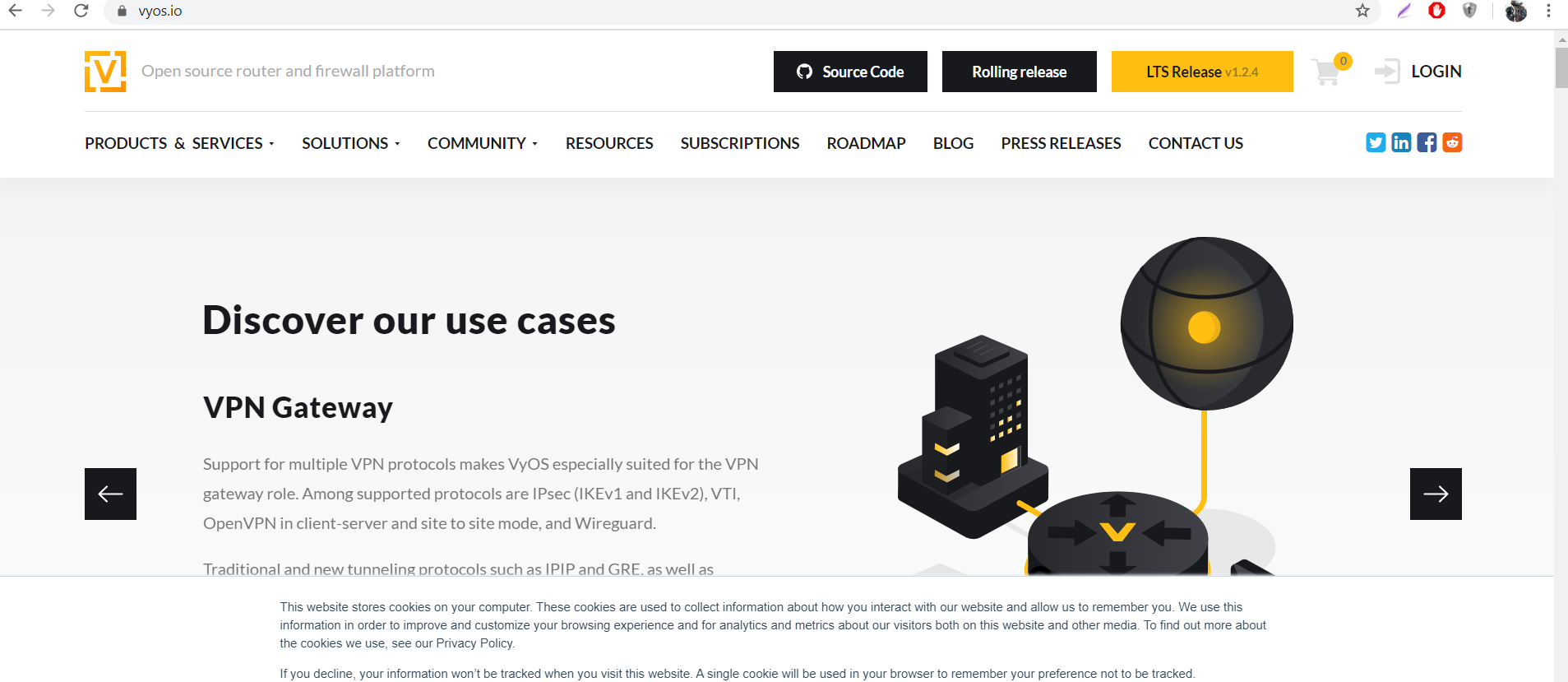
NTLM для учетных записей NTLM;

XAUTH для учетных записей XAUTH;

PIN для пин-кода смарт-карт.

Соответственно поддерживаются все типы аутентификации.  
Основные параметры команды ipsec, которая управляет подключениями StrongSwan:  
Логи хранятся в /var/log/auth.log и /var/log/daemon.log.  
 Данный продукт позволяет на базе linux систем сделать то что нам нужно, но все файл конфигурации будут раскиданы по репозиториям из за чего он нам не подходит.

Следующий кандидат VyoOS: VyOS представляет собой форк известной сетевой операционной системы [Vyatta](http://www.brocade.com/launch/vyatta/). Его первый релиз под кодовым названием Hydrogen был представлен в декабре 2013 года.  
Последний на сегодняшний день релиз — Helium — вышел в свет в сентябре 2014 года. Интерфейс командной строки (CLI) в VyOS похож на СLI устройств от Juniper Networks. Как и Vyatta, VyOS основан на Debian. Это позволяет расширить функциональность путём установки дополнительных deb-пакетов.



Возможности VyOS действительно широки. Вот далеко не полный список:

* межсетевой экран для IPv4 и IPv6, включая фильтрацию p2p-трафика;  
  трансляция сетевых адресов (NAT);
* DHCP-сервер для IPv4 и IPv6;
* система обнаружения вторжений;
* балансировка нагрузки и резервирование канала;
* резервирование маршрутизаторов с синхронизацией таблицы состояний соединений;
* виртуальные частные сети (IPsec, L2TP/IPsec, PPTP, OpenVPN);
* учёт трафика (Netflow и sFlow);
* веб-прокси и фильтрация URL.

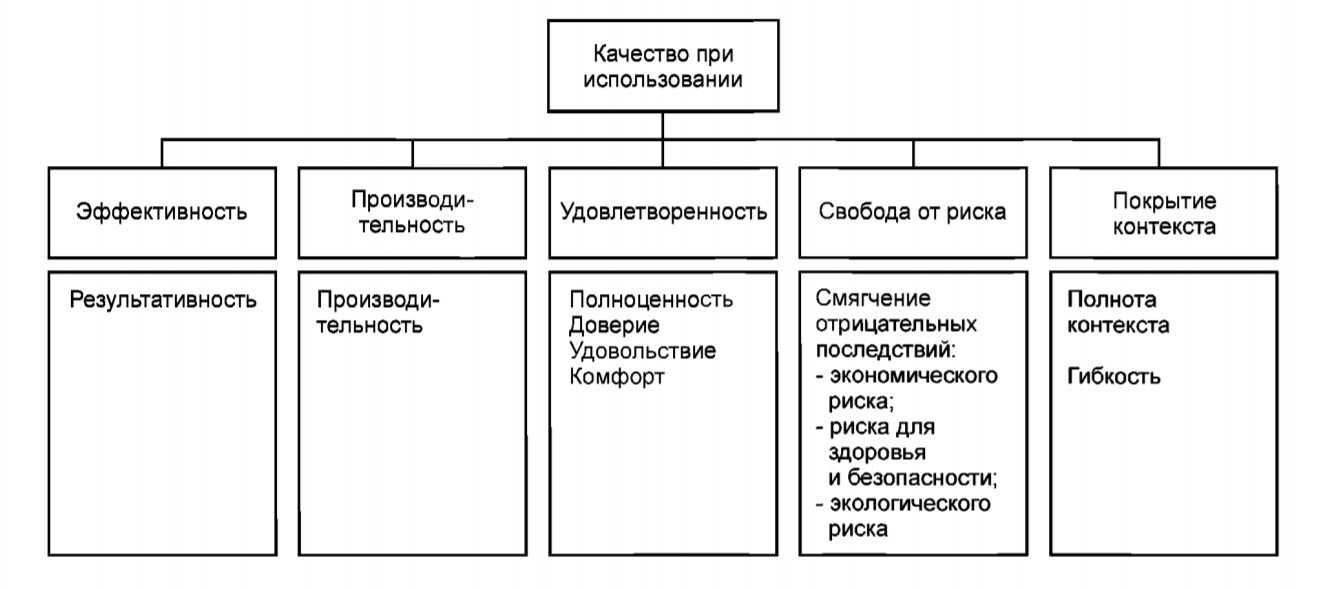
Тем самым данное программное обеспечение нам полностью подходит и у него имеется готовая официальная сборка для разворачивания в среде контейнера.

1.3. Требования автоматизации и отказоустойчивости структуры.

Прежде чем проектировать серверную структуру, которая будет располагаться в облачных сервисах необходимо определить требования и ограничения для нее. В данном проекте планируется, что эта структура будет использоваться для подключений к smsc и ussd серверов, то есть будет ходить большой поток информации. И каждый простой или отказ будет стоить больших убытков. Потому что отсутствующая магистраль до серверов означает, что абоненты не могут получить услуги связи.

Если представить проектирующуюся структуру, как продукт то к нему можно наложить требования ГОСТ Р ИСО/МЭК 25010— 2015 «Требования и оценка качества систем и программного обеспечения». Данный стандарт был выбран по причине отсутствия по данному виду архитектуры других стандартов, ведь проектирование будет осуществляться на уровне выше второго модели OSI. Это связано с тем что коммутацию на уровне 2 и 1 модели OSI осуществляется средствами дата-центра который предоставляет физическое и логическое пространство. Так же данный выбор позволяет не беспокоится о безопасности на уровне 2 и 1 модели OSI, потому что перед выбором дата-центра можно проверить и убедиться в наличии сертификатов безопасности которые будут гарантировать указанный на них уровень безопасности.

Если рассмотреть пункт 3.2 Модель качества при использовании.



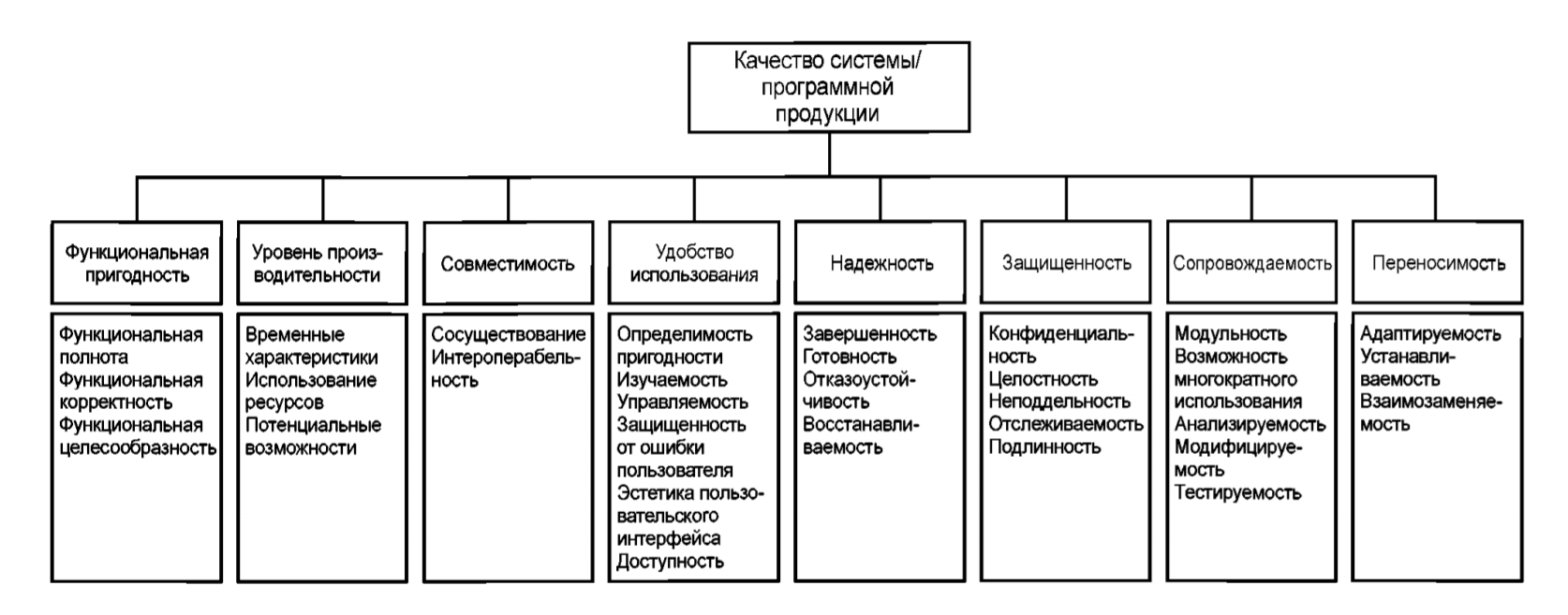
Первые два пункта говорят о том, что система должна в полной мере отвечать поставленным ей задачам. В данном случае это полноценная передача без потерь если рассматривать только эту сторону задачи.

Параметр удовлетворенность размыт так как удовлетворенность будет зависть только от отсутствия с проблем с сетью.

А вот параметр свобода от риска один из самых сложных в данной задаче. Проектируемая Сетевая структура должна обеспечивать резервный маршрут для бесперебойной работы. И в случае выхода из строя по техническим причинам одного контейнера должна использовать другой и делать попытки восстановить первый. Так же должна быть настроена система мониторинга изменений конфигураций, то есть резервное копирование. То есть нужно будет указать в конфигурационном файле количество изменений после которых должна происходить отправка конфигурационного файла на внутренний сервер.

Полноту контекста можно охарактеризовать как: Степень, в которой структура применима при использовании малх вычислительных ресурсов с высокой сетевой пропускной способностью, квалифицированными пользователями и в отказоустойчивом режиме.

В пункте модели качество продукта данного Госта указаны характеристики и под характеристики как показано ниже.



Глава 2. Обзор технологий и их вариаций для использования в разработке.

2.1. Принцип работы докера.

Докер — это открытая платформа для разработки, доставки и эксплуатации приложений. Docker разработан для более быстрого выкладывания ваших приложений. С помощью docker можно отделить приложение от инфраструктуры и обращаться с инфраструктурой как управляемым приложением. Docker помогает выкладывать ваш код быстрее, быстрее тестировать, быстрее выкладывать приложения и уменьшить время между написанием кода и запуска кода. Docker делает это с помощью легковесной платформы контейнерной виртуализации, используя процессы и утилиты, которые помогают управлять и выкладывать ваши приложения.  
В своем ядре docker позволяет запускать практически любое приложение, безопасно изолированное в контейнере. Безопасная изоляция позволяет запускать на одном хосте много контейнеров одновременно. Легковесная природа контейнера, который запускается без дополнительной нагрузки гипервизора, позволяет вам добиваться больше от вашего железа.  
Платформа и средства контейнерной виртуализации могут быть полезны в следующих случаях:

- упаковывание приложения (и так же используемых компонент) в docker контейнеры;

- раздача и доставка этих контейнеров командам для разработки и тестирования;

- выкладывания этих контейнеров на продакшены, как в дата центры так и в облака.

Основанная на контейнерах docker платформа позволят легко портировать вашу полезную нагрузку. Docker контейнеры могут работать на вашей локальной машине, как реальной так и на виртуальной машине в дата центре, так и в облаке.  
Портируемость и легковесная природа docker позволяет легко динамически управлять вашей нагрузкой. Вы можете использовать docker, чтобы развернуть или погасить приложение или сервисы. Скорость docker позволяет делать это почти в режиме реального времени.

Docker легковесен и быстр. Он предоставляет устойчивую, рентабельную альтернативу виртуальным машинам на основе гипервизора. Он особенно полезен в условиях высоких нагрузок, например, при создания собственного облака или платформа-как-сервис (platform-as-service). Но он так же полезен для маленьких и средних приложений, когда хочется получать больше из имеющихся ресурсов.

Docker состоит из двух главных компонент:

Docker: платформа виртуализации с открытым кодом;

Docker Hub: наша платформа-как-сервис для распространения и управления docker контейнерами.

Docker использует архитектуру клиент-сервер. Docker клиент общается с демоном Docker, который берет на себя тяжесть создания, запуска, распределения ваших контейнеров. Оба, клиент и сервер могут работать на одной системе, вы можете подключить клиент к удаленному демону docker. Клиент и сервер общаются через сокет или через RESTful API.

- Docker-демон: Демон за пускается на хост-машине. Пользователь не взаимодействует с сервером на прямую, а использует для этого клиент.  
- Docker-клиент: Docker-клиент, программа docker — главный интерфейс к Docker. Она получает команды от пользователя и взаимодействует с docker-демоном.

- Внутри docker-а: Чтобы понимать, из чего состоит docker, вам нужно знать о трех компонентах:

* образы (images)
* реестр (registries)
* контейнеры

- Образы: Docker-образ — это read-only шаблон. Например, образ может содержать операционку Ubuntu c Apache и приложением на ней. Образы используются для создания контейнеров. Docker позволяет легко создавать новые образы, обновлять существующие, или вы можете скачать образы созданные другими людьми. Образы — это компонента сборки docker-а.

- Реестр: Docker-реестр хранит образы. Есть публичные и приватные реестры, из которых можно скачать либо загрузить образы. Публичный Docker-реестр — это [Docker Hub](http://hub.docker.com/). Там хранится огромная коллекция образов. Как вы знаете, образы могут быть созданы вами или вы можете использовать образы созданные другими. Реестры — это компонента распространения.  
- Контейнеры: Контейнеры похожи на директории. В контейнерах содержится все, что нужно для работы приложения. Каждый контейнер создается из образа. Контейнеры могут быть созданы, запущены, остановлены, перенесены или удалены. Каждый контейнер изолирован и является безопасной платформой для приложения. Контейнеры — это компонента работы.

- Так как же работает Docker?

Пока мы знаем, что:  
можем создавать образы, в которых находятся наши приложения;

можем создавать контейнеры из образов, для запуска приложений;

можем распространять образы через Docker Hub или другой реестр образов.  
Давайте посмотрим, как эти компоненты сочетаются.  
 - Как работает образ?

Мы уже знаем, что образ — это read-only шаблон, из которого создается контейнер. Каждый образ состоит из набора уровней. Docker использует [union file system](http://en.wikipedia.org/wiki/UnionFS) для сочетания этих уровней в один образ. Union file system позволяет файлам и директориями из разных файловых систем (разным ветвям) прозрачно накладываться, создавая когерентную файловую систему.  
Одна из причин, по которой docker легковесен — это использование таких уровней. Когда вы изменяете образ, например, обновляете приложение, создается новый уровень. Так, без замены всего образа или его пересборки, как вам возможно придётся сделать с виртуальной машиной, только уровень добавляется или обновляется. И вам не нужно раздавать весь новый образ, раздается только обновление, что позволяет распространять образы проще и быстрее.  
В основе каждого образа находится базовый образ. Например, ubuntu, базовый образ Ubuntu, или fedora, базовый образ дистрибутива Fedora. Так же вы можете использовать образы как базу для создания новых образов. Например, если у вас есть образ apache, вы можете использовать его как базовый образ для ваших веб-приложений.

Docker образы могут создаться из этих базовых образов, шаги описания для создания этих образов мы называем инструкциями. Каждая инструкция создает новый образ или уровень. Инструкциями будут следующие действия:

* запуск команды
* добавление файла или директории
* создание переменной окружения
* указания что запускать когда запускается контейнер этого образа

Эти инструкции хранятся в файле Dockerfile. Docker считывает это Dockerfile, когда вы собираете образ, выполняет эти инструкции, и возвращает конечный образ.

Как работает docker реестр?  
Реестр — это хранилище docker образов. После создания образа вы можете опубликовать его на публичном реестре Docker Hub или на вашем личном реестре.  
С помощью docker клиента вы можете искать уже опубликованные образы и скачивать их на вашу машину с docker для создания контейнеров.  
Docker Hub предоставляет публичные и приватные хранилища образов. Поиск и скачивание образов из публичных хранилищ доступно для всех. Содержимое приватных хранилищ не попадает в результат поиска. И только вы и ваши пользователи могут получать эти образы и создавать из них контейнеры.

Как работает контейнер?

Контейнер состоит из операционной системы, пользовательских файлов и метаданных. Как мы знаем, каждый контейнер создается из образа. Этот образ говорит docker-у, что находится в контейнере, какой процесс запустить, когда запускается контейнер и другие конфигурационные данные. Docker образ доступен только для чтения. Когда docker запускает контейнер, он создает уровень для чтения/записи сверху образа (используя union file system, как было указано раньше), в котором может быть запущено приложение.  
Что происходит, когда запускается контейнер?

Или с помощью программы docker, или с помощью RESTful API, docker клиент говорит docker демону запустить контейнер.  
$ sudo docker run -i -t ubuntu /bin/bash  
Давайте разберемся с этой командой. Клиент запускается с помощью команды docker, с опцией run, которая говорит, что будет запущен новый контейнер. Минимальными требованиями для запуска контейнера являются следующие атрибуты: какой образ использовать для создания контейнера. В нашем случае ubuntu команду которую вы хотите запустить когда контейнер будет запущен. В нашем случае /bin/bash  
Что же происходит под капотом, когда мы запускаем эту команду?  
Docker, по порядку, делает следующее: скачивает образ ubuntu: docker проверяет наличие образа ubuntu на локальной машине, и если его нет — то скачивает его с [Docker Hub](https://hub.docker.com/). Если же образ есть, то использует его для создания контейнера; создает контейнер: когда образ получен, docker использует его для создания контейнера; инициализирует файловую систему и монтирует read-only уровень: контейнер создан в файловой системе и read-only уровень добавлен образ; инициализирует сеть/мост: создает сетевой интерфейс, который позволяет docker-у общаться хост машиной;

* Установка IP адреса: находит и задает адрес;
* Запускает указанный процесс: запускает ваше приложение;

Обрабатывает и выдает вывод вашего приложения: подключается и логирует стандартный вход, вывод и поток ошибок вашего приложения, что бы вы могли отслеживать как работает ваше приложение.  
Теперь у вас есть рабочий контейнер. Вы можете управлять своим контейнером, взаимодействовать с вашим приложением. Когда решите остановить приложение, удалите контейнер.

Используемые технологии

Докер написан на Go и использует некоторые возможности ядра Linux, чтобы реализовать приведенный выше функционал.

Пространство имен(namespaces)

Docker использует технологию namespaces для организации изолированных рабочих пространств, которые мы называем контейнерами. Когда мы запускаем контейнер, docker создает набор пространств имен для данного контейнера.  
Это создает изолированный уровень, каждый аспект контейнера запущен в своем простанстве имен, и не имеет доступ к внешней системе.  
Список некоторых пространств имен, которые использует docker:

* pid: для изоляции процесса;
* net: для управления сетевыми интерфейсами;
* ipc: для управления IPC ресурсами. (ICP: InterProccess Communication);
* mnt: для управления точками монтирования;
* utc: для изолирования ядра и контроля генерации версий(UTC: Unix timesharing system).

Control groups (контрольные группы)  
Docker также использует технологию cgroups или контрольные группы. Ключ к работе приложения в изоляции, предоставление приложению только тех ресурсов, которые вы хотите предоставить. Это гарантирует, что контейнеры будут хорошими соседями. Контрольные группы позволяют разделять доступные ресурсы железа и если необходимо, устанавливать пределы и ограничения. Например, ограничить возможное количество памяти контейнеру.

Union File System  
Union File Sysem или UnionFS — это файловая система, которая работает создавая уровни, делая ее очень легковесной и быстрой. Docker использует UnionFS для создания блоков, из которых строится контейнер. Docker может использовать несколько вариантов UnionFS включая: AUFS, btrfs, vfs и DeviceMapper.

Форматы контейнеров

Docker сочетает эти компоненты в обертку, которую мы называем форматом контейнера. Формат, используемый по умолчанию, называется libcontainer. Так же docker поддерживает традиционный формат контейнеров в Linux c помощью [LXC](http://linuxcontainers.org/). В будущем Docker возможно будет поддерживать другие форматы контейнеров. Например, интегрируясь с BSD Jails или Solaris Zones.

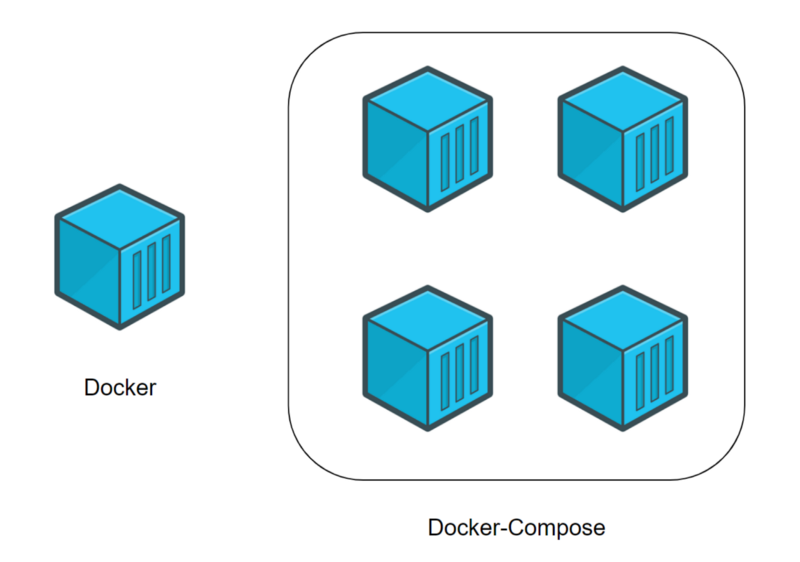
2.2. Основные методы создания файла конфигураций.

2.2.1 Docker-compose

Docker Compose — это инструментальное средство, входящее в состав Docker. Оно предназначено для решения задач, связанных с развёртыванием проектов.

Изучая основы Docker, можно столкнуться с созданием простейших приложений, работающих автономно, не зависящих, например, от внешних источников данных или от неких сервисов. На практике же подобные приложения — редкость. Реальные проекты обычно включают в себя целый набор совместно работающих приложений.  
Как узнать, нужно ли вам, при развёртывании некоего проекта, воспользоваться Docker Compose? На самом деле — очень просто. Если для обеспечения функционирования этого проекта используется несколько сервисов, то Docker Compose может вам пригодиться. Например, в ситуации, когда создают веб-сайт, которому, для выполнения аутентификации пользователей, нужно подключиться к базе данных. Подобный проект может состоять из двух сервисов — того, что обеспечивает работу сайта, и того, который отвечает за поддержку базы данных.  
Технология Docker Compose, если описывать её упрощённо, позволяет, с помощью одной команды, запускать множество сервисов.

Docker применяется для управления отдельными контейнерами (сервисами), из которых состоит приложение.  
Docker Compose используется для одновременного управления несколькими контейнерами, входящими в состав приложения. Этот инструмент предлагает те же возможности, что и Docker, но позволяет работать с более сложными приложениями.



Docker Compose — это, в умелых руках, весьма мощный инструмент, позволяющий очень быстро развёртывать приложения, отличающиеся сложной архитектурой. Сейчас мы рассмотрим пример практического использования Docker Compose, разбор которого позволит вам оценить те преимущества, которые даст вам использование Docker Compose.  
Представьте себе, что вы являетесь разработчиком некоего веб-проекта. В этот проект входит два веб-сайта. Первый позволяет людям, занимающимся бизнесом, создавать, всего в несколько щелчков мышью, интернет-магазины. Второй нацелен на поддержку клиентов. Эти два сайта взаимодействуют с одной и той же базой данных.  
Ваш проект становится всё популярнее, и оказывается, что мощности сервера, на котором он работает, уже недостаточно. В результате вы решаете перевести весь проект на другую машину.  
К сожалению, нечто вроде Docker Compose вы не использовали. Поэтому вам придётся переносить и перенастраивать сервисы по одному, надеясь на то, что вы, в процессе этой работы, ничего не забудете.  
Если же вы используете Docker Compose, то перенос вашего проекта на новый сервер — это вопрос, который решается выполнением нескольких команд. Для того чтобы завершить перенос проекта на новое место, вам нужно лишь выполнить кое-какие настройки и загрузить на новый сервер резервную копию базы данных.

Теперь, когда вы знаете о том, для чего мы собираемся использовать Docker Compose, пришло время создать ваше первое клиент-серверное приложение с использованием этого инструмента. А именно, речь идёт о разработке небольшого веб-сайта (сервера) на Python, который умеет выдавать файл с фрагментом текста. Этот файл у сервера запрашивает программа (клиент), тоже написанная на Python. После получения файла с сервера программа выводит текст, хранящийся в нём, на экран.  
Обратите внимание на то, что мы рассчитываем на то, что вы владеете основами Docker, и на то, что у вас уже установлена платформа Docker.  
Приступим к работе над проектом.

1. Создание проекта

Для того чтобы построить ваше первое клиент-серверное приложение, предлагаю начать с создания папки проекта. Она должна содержать следующие файлы и папки:

Файл docker-compose.yml. Это файл Docker Compose, который будет содержать инструкции, необходимые для запуска и настройки сервисов.

Папка server. Она будет содержать файлы, необходимые для обеспечения работы сервера.

Папка client. Здесь будут находиться файлы клиентского приложения.  
В результате содержимое главной папки вашего проекта должно выглядеть так:

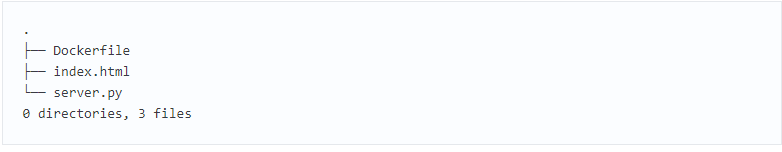


Тут мы, в процессе создания сервера, затронем некоторые базовые вещи, касающиеся Docker.

2a. Создание файлов

Перейдите в папку server и создайте в ней следующие файлы:

* Файл server.py. В нём будет находиться код сервера.
* Файл index.html. В этом файле будет находиться фрагмент текста, который должно вывести клиентское приложение.
* Файл Dockerfile. Это — файл Docker, который будет содержать инструкции, необходимые для создания окружения сервера.  
  Вот как должно выглядеть содержимое вашей папки server/:



2b. Редактирование Python-файла.

Добавим в файл server.py следующий код:



Этот код позволяет создать простой веб-сервер. Он будет отдавать клиентам файл index.html, содержимое которого позже будет выводиться на веб-странице.

2c. Редактирование HTML-файла

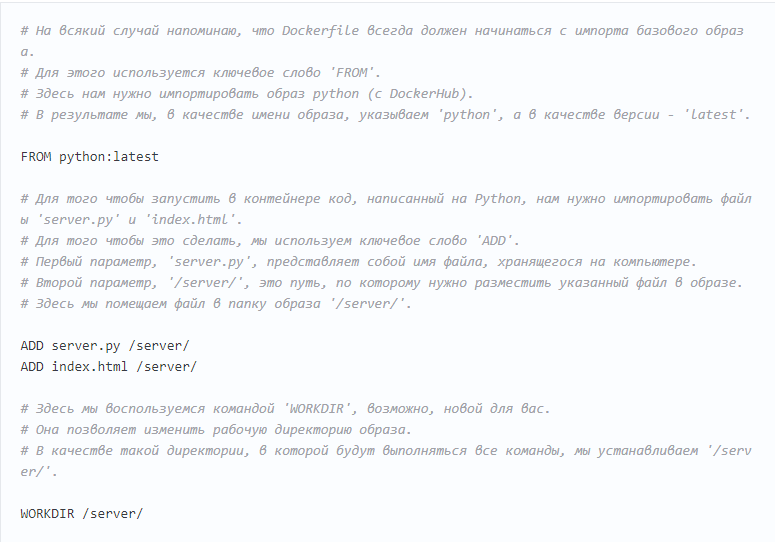
В файл index.html добавим следующий текст:



Этот текст будет передаваться клиенту.

2d. Редактирование файла Dockerfile

Сейчас мы создадим простой файл Dockerfile, который будет отвечать за организацию среды выполнения для Python-сервера. В качестве основы создаваемого образа воспользуемся [официальным образом](https://hub.docker.com/_/python), предназначенным для выполнения программ, написанных на Python. Вот содержимое Dockerfile:



теперь займёмся работой над клиентом.

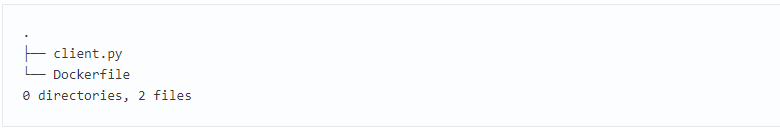
3. Создание клиента

Создавая клиентскую часть нашего проекта, мы попутно вспомним некоторые основы Docker.

3a. Создание файлов

Перейдите в папку вашего проекта client и создайте в ней следующие файлы:

* Файл client.py. Тут будет находиться код клиента.
* Файл Dockerfile. Этот файл играет ту же роль, что и аналогичный файл в папке сервера. А именно, он содержит инструкцию, описывающую создание среды для выполнения клиентского кода.  
  В результате ваша папка client/ на данном этапе работы должна выглядеть так:



3b. Редактирование Python-файла

Добавим в файл client.py следующий код:



Благодаря этому коду клиентское приложение может загрузить данные с сервера и вывести их на экран.

3c. Редактирование файла Dockerfile

Как и в случае с сервером, мы создаём для клиента простой Dockerfile, ответственный за формирование среды, в которой будет работать клиентское Python-приложение. Вот код клиентского Dockerfile:



4. Docker Compose

Как вы могли заметить, мы создали два разных проекта: сервер и клиент. У каждого из них имеется собственный файл Dockerfile. До сих пор всё происходящее не выходит за рамки основ работы с Docker. Теперь же мы приступаем к работе с Docker Compose. Для этого обратимся к файлу docker-compose.yml, расположенному в корневой папке проекта.  
Обратите внимание на то, что тут мы не стремимся рассмотреть абсолютно все команды, которые можно использовать в docker-compose.yml. Наша главная цель — разобрать практический пример, дающий вам базовые знания по Docker Compose.  
Вот код, который нужно поместить в файл docker-compose.yml:





5. Сборка проекта

После того, как в docker-compose.yml внесены все необходимые инструкции, проект нужно собрать. Этот шаг нашей работы напоминает использование команды docker build, но соответствующая команда имеет отношение к нескольким сервисам:



6. Запуск проекта

Теперь, когда проект собран, пришло время его запустить. Этот шаг нашей работы соответствует шагу, на котором, при работе с отдельными контейнерами, выполняется команда docker run:



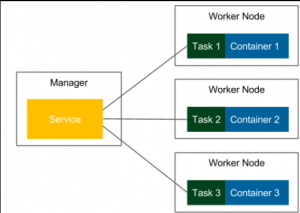
После выполнения этой команды в терминале должен появиться текст, загруженный клиентом с сервера: Docker-Compose is magic!.  
Как уже было сказано, сервер использует порт компьютера 1234 для обслуживания запросов клиента. Поэтому, если перейти в браузере по адресу <http://localhost:1234/>, в нём будет отображена страница с текстом Docker-Compose is magic!.

2.2.2  Docker Swarm и Docker Stack.

Docker Swarm — это родная система кластеризации для Docker, которая превращает набор хостов Docker в один последовательный кластер, называемый Swarm.  
Каждая/ый нода/хост, в составе такого кластера выступает в качестве либо управляющей(manager) либо рабочей ноды(worker).  
В кластере должна быть, как минимум, одна управляющая нода(manager).  
Технически физическое расположение машин не имеет значения, однако, желательно иметь все Docker-ноды внутри одной локальной сети, в противном случае — управление операциями (или  
поиск консенсуса между несколькими управляющими нодами) может занять значительное количество времени.  
Начиная с Docker 1.12, Docker Swarm уже интегрирован в Docker Engine как Swarm-режим  
В более старых версиях необходимо было запускать swarm-контейнер на каждом из хостов для обеспечения функционала кластеризации.

**Термины в Docker Swarm:**  
**Service** – запущенный образ ( в отличии от Docker Engine, где запущенный на Docker-хосте образ называется контейнером).  
Одна service запускает определенное количество задач.  
**Задача** — это атомная запланированная единица Docker Swarm, которая содержит информацию о контейнере и команду, которая должна запущена внутри контейнера.  
**Реплика** – каждый контейнер, который запускается на ноде  
**Количество реплик** — это ожидаемое число всех контейнеров для данной service

Запуск Docker Swarm происходит с указанием имени service, образа Docker, который нужно использовать для контейнеров и желаемого количества реплик.  
Управляющая нода автоматически назначает задачи рабочим нодам.  
Каждый реплицируемый контейнер запускается с одного и того же Docker-образа.  
Docker Swarm может быть рассмотрен как слой поверх механизма Docker Engine, который отвечает за оркестрацию контейнеров



В данном изображения у нас есть три задачи, и каждая из них запускается на отдельной Docker-ноде.  
Тем не менее, также может случиться так, что все контейнеры будут запущены на одной и той же Docker-ноде.  
Все зависит от управляющей ноды, которая распределяет задачи для рабочих нод, используя стратегию планирования, которая будет описана в разделе  
Стратегия планирования – распределения задач по рабочим нодам

**Интересные возможности Docker Swarm:**

**Балансировка нагрузки:** Docker Swarm отвечает за балансировку нагрузки и назначение  
уникальных DNS-имен, чтобы приложение, развернутое в кластере, можно было использовать  
так же, как и, если приложение было бы развернуто на одном Docker-хосте.  
Другими словами, Docker Swarm может публиковать порты так же, как контейнер в Docker Engine, а затем управляющая нода распределяет запросы между service-ами в кластере.

**Динамическое управление ролями:** Docker-хосты могут быть добавлены Swarm-кластеру без необходимости перезапуска кластера. Более того, роль узла (управляющий или рабочий) также может динамически меняться.

**Динамическое масштабирование сервисов:** Каждая service может динамически масштабироваться как в сторону увеличения, так и в сторону уменьшения с клиентом Docker. Управляющая нода заботится о добавлении или удалении контейнеров на узлах.

**Восстановление отказа:** ноды постоянно контролируются управляющей нодой и, если  
какая-либо нода сбоит, то новые задачи запускаются на других рабочих нодах с целью, чтобы обеспечивалось заявленное/желаемое количество реплик.  
Docker Swarm также позволяет создавать несколько управлющих нод для предотвращения поломки кластера в случае выхода со строя единственной управляющей ноды

**Rolling-обновления:** обновление сервисов может применяться постепенно.  
Например, если у нас есть 10 реплик, и мы хотим внести изменения, мы можем определить задержку между развертыванием для каждой реплики.  
В таком случае, когда что-то пойдет не так, процесс обновления автоматически прерывается, тем самым защищая нас от ситуации, когда в кластере не останется рабочих реплик.

**Режимы работы:** Есть два режима, в которых можно запускать Docker Swarm:

Реплицированные service: указанное количество реплицируемых контейнеров  
распределяются между узлами на основе стратегии планированния

Глобальные service: один контейнер запускается на каждом доступном узле в  
кластере

**Безопасность:** Как и все в Docker, Docker Swarm обеспечивает TLS  
аутентификации и шифрования связи. Также возможно использовать CA (или  
самоподписанные) сертификаты.

**Инициализация manager-ноды**

# docker swarm init

Если хост имеет несколько интерфейсов,то можно определить IP-адрес, на котором нужно запускать Swarm-cluster

# docker swarm init --advertise-addr IP-address

Проверяем состояние управляющей ноды

# docker node ls



На всех нодах кластера в файрволле открываем следующие порты и протоколы  
2377-tcp  
7946-tcp/udp  
4889-udp

# iptables -N swarm

# iptables -A INPUT -p tcp --dport 2377 -j swarm

# iptables -A INPUT -p tcp --dport 7946 -j swarm

# iptables -A INPUT -p udp --dport 7946 -j swarm

# iptables -A INPUT -p udp --dport 4889 -j swarm

# iptables -A swarm -s IP-address-manager-node -j ACCEPT

# iptables -A swarm -s IP-address-worker-node1 -j ACCEPT

# iptables -A swarm -s IP-address-worker-node2 -j ACCEPT

|  |  |
| --- | --- |
| 1 | # iptables -A swarm -j DROP |

**Добавление рабочих нод**  
В выводе команды, выполненной на предыдущем этапе

# docker swarm init

Есть подсказка, как добавить рабочие ноды

docker swarm join --token SWMTKN-1-5cbvu9zv8u8o2qkddjlfctrd3lfbk2l1gek0hec1dsgexuxw6h-1tls8aogc4443pb7zcq7c5gkf 163.172.XXX.YYY:2377

Также для получения необходимой команды для добавления ноды в качестве worker-ноды в swarm-кластер можно воспользоваться командой

# docker swarm join-token worker

Добавим ноду

docker swarm join --token SWMTKN-1-5cbvu9zv8u8o2qkddjlfctrd3lfbk2l1gek0hec1dsgexuxw6h-1tls8aogc4443pb7zcq7c5gkf 163.172.XXX.YYY:2377

Проверим список нод в Swarm-кластере



Как видно, в даном случае 3 ноды в кластере, одна управляющая нода(lxc) и две рабочих ноды(ubuntu161 и ubuntu162)  
По умолчанию в Docker Swarm контейнеры/службы/задачи могут запускаться,как на рабочих нодах(worker), так и на управляющих нодах(manager)  
Для отключения запуска задач на управляющей ноде,чтобы она занималась только орекстрацией задач/контейнеров, а не их выполнением, установим для управляющей ноды дотупность в drain

|  |  |
| --- | --- |
| 1 | # docker node update --availability drain lxc |

Проверим,что статус управляющей ноды изменился с Active на Drain

# docker node ls | grep lxc

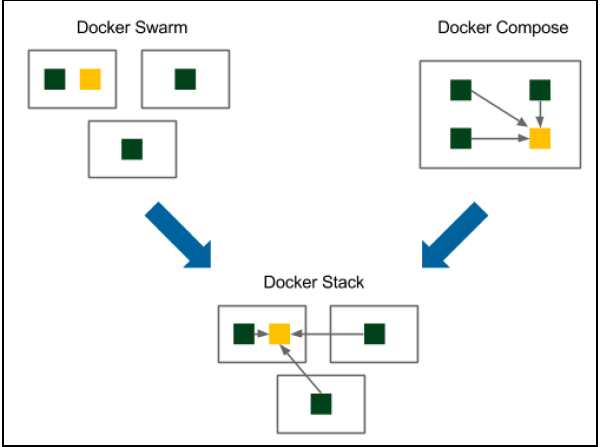
Альтернативным вариантом для НЕ запуска задач на управляющей ноде используется параметр –constraint с указанным ниже значением

--constraint node.role==worker

docker service create --replicas 1 --name tomcat tomcat

**Интеграция Docker Swarm и Docker Compose в Docker Stack**  
Docker Swarm запускает контейнеры на разных физических хостах не предоставляя связи между такими запущенными контейнерами(т.е. для того, чтобы контейнеры могли коммуницировать между собой необходимо вручную создавать такую сетевую связь)  
Docker-compose, наоборот, запускает несколько взаимосвязанных между собой контейнеров, но в пределах одного физического сервера(т.е. горизонтальное масштабирование ограничено ресурсами одной физической ноды)  
Docker Stack объединяет в себе обе технологи и позволяет запускать несколько связанных контейнеров на различных физических нодах(хостах Docker-кластера)  
Возможность запуска нескольких контейнеров, описанных в Docker-compose на нескольких физических нодах кластера Dcocker-swarm реализуется с помощью Docke

r Stack



Создадим docker-compose.yaml файл, который будет запускать 3 контейнера с приложеним calculator и 1 контейнер с Redis размещая контейнеры на рабочих нодах ubuntu16



При этом docker-образ mydocker.repo.servername/calculator:latest уже должен присутствовать на всех рабочих нодах Swarm-кластера(иначе на ноде, на которой нет этого образа, не запустится приложение calculator).  Альтернативным вариантом для получения рабочими нодами docker-образа mydocker.repo.servername/calculator:latest является использование параметра

|  |  |
| --- | --- |
| 1 | --with-registry-auth |

при выполнении команды docker stack deploy(это позволит рабочим нодам аутентифицироваться в Docker-репозитарии и скачать этот образ)

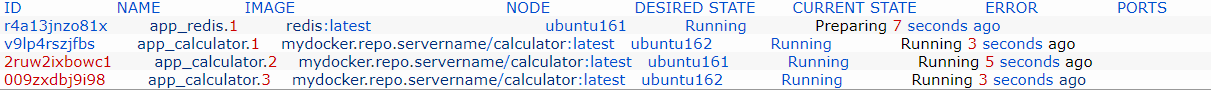
Запуск такого стека приложений

# docker stack deploy --compose-file docker-compose.yaml app



Просмотр списка задач в стеке app

# docker stack ps app



Как видно, 2 контейнера с приложением запустились на рабочей ноде ubuntu162 и один – на ubuntu161.  
Один контейнер с Redis запустился на рабочей ноде ubuntu161