

Московский государственный университет имени М.В. Ломоносова

Факультет Вычислительной математики и кибернетики

Кафедра Автоматизации Систем Вычислительных Комплексов

Василенко Анатолий Эдуардович

**Требования, предъявляемые сетевыми приложениями к легковесным контейнерам, при их использовании для моделирования сетей**

РЕФЕРАТ

Москва, 2015

1. Содержание

[1 Содержание 2](#_Toc435441640)

[2 Введение 3](#_Toc435441641)

[3 Введение в предметную область. Определения 4](#_Toc435441642)

[4 Виртуализация 5](#_Toc435441643)

[5 Симуляция и эмуляция 7](#_Toc435441644)

[6 Linux Контейнеры 9](#_Toc435441645)

[6.1 Общая информация о Linux контейнерах 9](#_Toc435441646)

[6.2 Внутреннее устройство Linux контейнеров 9](#_Toc435441647)

[6.2.1 Namespaces 10](#_Toc435441648)

[6.2.2 Cgroup 13](#_Toc435441649)

[6.2.3 Технологии Namespaces и Cgroups в совокупности 15](#_Toc435441650)

[6.3 Linux Containers (LXC) как подход к виртуализации 17](#_Toc435441651)

[6.4 Docker - система менеджмента легковесной виртуализации 19](#_Toc435441652)

[6.5 Сравнение Linux контейнеров и виртуальных машин. 22](#_Toc435441653)

[6.6 Место контейнеров в реальном мире 24](#_Toc435441654)

[7 Особенности моделирования на основе контейнеров 26](#_Toc435441655)

[7.1 Примеры существующих систем моделирования сети 26](#_Toc435441656)

[7.2 Основные требования к системам моделирования на основе легковесных контейнеров 27](#_Toc435441657)

[7.3 Возможности легковесных контейнеров по контролю выделяемых ресурсов 29](#_Toc435441658)

[8 Заключение 30](#_Toc435441659)

[9 Список литературы 31](#_Toc435441660)

1. Введение

Современный мир IT продолжает активно развиваться, создаются новые концепции и происходят новые витки развития. В частности, это происходит и в мире сетевых технологий, не смотря на уже достаточно большой возраст этой отрасли. Самыми яркими примерами тенденций современного сетевого мира является концепция *SDN* и *NFV*, которые связаны с другой тенденцией мира IT в целом – это виртуализация, основанная на понятиях виртуальных машин и контейнеров.

Для сетей технология виртуализации, основанная на контейнерах, может использоваться двумя принципиальными путями: для работы абстрактного сетевого приложения в рамках контейнера, генерирующего трафик, который необходимо маршрутизировать, и для работы виртуализированного сетевого сервиса в рамках контейнера, который должен обрабатывать проходящий через него трафик.

Упомянутые тенденции находятся в активном развитии и технологии продолжают улучшаться, набирая новый функционал.

Цель реферата:

1. Изучить возможности, предоставляемые контейнерами и сопоставить их с требованиями приложений, по возможности выявив недостатки в использовании легковесных контейнеров.

Задачи, которые необходимо решить для достижения цели:

1. Изучить возможности и недостатки современных технологий легковесной виртуализации и сравнить их с возможностями полновесной виртуализации.
2. Обозначить возможные варианты использования легковесной виртуализации в современном мире.
3. Изучить роль легковесной виртуализации при моделировании сетей, и какие возможности от неё при этом требуются.

Структура работы основана на последовательном выполнении задач для достижения цели реферата после предварительно знакомства с предметной областью. На каждом этапе, по возможности, будут выявляться недостатки современных технологий виртуализации, которые важны для запуска приложений различного характера, в частности при моделировании сети. В завершение будут сформулированы необходимые требования при моделировании, которые касаются легковесной виртуализации.

1. Введение в предметную область. Определения

**Виртуализация** - предоставление набора вычислительных ресурсов или их логического объединения, абстрагированное от аппаратной реализации, и обеспечивающее при этом логическую изоляцию вычислительных процессов, выполняемых на одном физическом ресурсе.

**Виртуальная машина** - программная и/или аппаратная система, эмулирующая аппаратное обеспечение некоторой платформы. Существуют технологии аппаратной поддержки виртуализации: Intel-VT, AMD-V.

По мере развития концепции виртуализации, технологии росли в сторону минимизации её накладных расходов и увеличения преимуществ. Была создана аппаратная поддержка и были разработаны различные гипервизоры (KVM, XenServer, VMWare ESXi, …), разработаны системы, занимающиеся менеджментом виртуальных машин, для развёртывания и поддержки облаков (OpenStack, VMware vSphere, …)

Появилась идея виртуализации на уровне операционной системы – контейнеры (LXC (Linux Containers), OpenVZ, FreeBSD jails, Solaris zones). На их основе были созданы системы менеджмента Docker (на базе LXC, libcontainer), Virtuozza (на базе OpenVZ).

**Контейнеры** - система виртуализации на уровне операционной системы для создания видимости нескольких изолированных экземпляров операционной системы Linux на одном узле.

**Docker** - программное обеспечение для автоматизации развёртывания и управления приложениями в среде виртуализации на уровне операционной системы (управления контейнерами). Позволяет «упаковать» приложение со всем его окружением и зависимостями в контейнер, который может быть перенесён на любую Linux-систему с поддержкой “cgroups” и “namespaces” в ядре, а также предоставляет среду по управлению контейнерами.

Технология **NFV** (*Network Function Virtualization – Виртуализация Сетевых Функций*) основывается на принципе написания программного обеспечения, выполняющего ту или иную сетевую функцию и работающую с использованием технологии виртуализации.

В данном реферате мы будем касаться вопросов симуляции и эмуляции сетевых узлов, поэтому определим необходимые термины. **Симуляция** - имитация какого-либо физического процесса при помощи искусственной системы. **Эмуляция** - комплекс программно-аппаратных средств, предназначенный для копирования функций одной вычислительной системы (*гостевая система*) на другой, отличной от первой вычислительной системе, (*хост-система*), таким образом, чтобы эмулированное поведение как можно ближе соответствовало поведению оригинальной системы. Таким образом, цель симуляции в том, чтобы произвести имитацию какого-либо объекта для внешнего наблюдателя, а цель эмуляции в максимально точном воспроизведении внутреннего поведения. Сущностное различие приводит к различию в их назначении: основная цель симуляции – это симуляция **API** (*Application Programming Interface*) объекта, чтобы для него можно было разрабатывать другие системы, или чтобы проверять, на сколько правильно подходит созданное API для другого объекта. В то время как для эмуляции, как правило, в качестве основной задачи стоит запуск уже существующего объекта, но на основе другого окружения[[1]](#footnote-1). Эмуляция так же решает и задачу симуляции, однако, как правило, работает медленнее.

У эмулятора в отличие от симулятора есть свои внутренние часы, отсчитывающие такты, в соответствии с которым определяется скорость работы. Это может позволить изменять производительность эмулируемой системы на некоторый коэффициент k, при этом если производительность внешних компонент не меняется, то с точки зрения эмулируемой системы, в случае если всё совместимо, ей будет казаться, что производительность окружения соответственно ниже или выше, с коэффициентом 1/k.

*Модель сети* состоит из следующих основных частей: описание топологии и ресурсов, описание потоков данных и загрузка каналов, описание обработки трафика ([3]).

1. Виртуализация

Основным преимуществом виртуализации является возможность запускать на одной вычислительной системе несколько независимых виртуальных.

Спустя годы развития, технологии виртуализации добились следующих преимуществ:

1. Выполнение на одной вычислительной системе нескольких различных виртуальных систем. (позволяет разделять ресурсы между виртуальными системами и одновременно позволяет приложениям работать в своём окружении)
2. Изолированное выполнение приложений без возможности повлиять на другие системы. (безопасность)
3. Эмуляция различных архитектур для вычислительных систем.
4. Удобство создания, развёртывания и даже миграции систем в зависимости от ситуации. (гибкость)
5. Автоматизация при управлении множествами виртуальных систем[[2]](#footnote-2). (гибкость)
6. Используя подходящие драйверы и аппаратные техники ускорения, накладные расходы использования гипервизора в системах общего назначения сократились до десятка процентов[[3]](#footnote-3). (производительность)

При этом должны гарантироваться следующие возможности:

1. Моделирование необходимых логических ресурсов и возможность передачи той или иной виртуальной системе непосредственного управления некоторыми физическими ресурсами.
2. Управление разделением физических ресурсов (процессорный ресурс, сетевые адаптеры, оперативная память, блочный ввод/вывод) между виртуальными машини, контроль и ограничение использования ресурсов.

Рассматривая виртуализацию в рамках контейнеров, для них есть ограничение, вытекающие из дизайна: так как контейнеры работают на уровне операционной системы, то гостевая система не может иметь своей операционной системы.

Заметим, что гипервизоры были созданы в первую очередь с целью изолирования нагрузки (контроль ресурсов) и создания виртуального окружения, однако строго говоря это зона ответственности операционной системы, и поэтому виртуализация на её уровне более целесообразна.

На практике, виртуализация часто используется в следующих целях: организация центров обработки данных (в частности для предоставления услуг типа IaaS, PaaS, SaaS), организация тестовых стендов, использование разработчиками и использование в учебных целях.

1. Симуляция и эмуляция

Симуляция или эмуляция является одним из ярких примеров использования легковесных контейнеров в сетевой отрасли. На их основе строятся распределённые модели сетевых топологий, включающих в себя различные системы и сервисы. Цели, для которых это может использоваться, можно разбить на следующие категории (в различных экспериментах цели могут смешиваться):

1. Для предоставления модели сети некоторому программному или аппаратному устройству с целью его тестирования.
2. Для использования модели в тестировании алгоритмов, непосредственно взаимодействующих с ней и использующих её как объект своего влияния (например, изучение сходимости протокола или изучение свойств топологии).
3. Для использования модели в тестировании алгоритмов, которые используют сеть лишь как среду передачи данных, не являющуюся непосредственным объектом влияния (например, изучение распространения червя по сети).

Система симуляции по мнению авторов статьи [3] должна иметь следующие компоненты:

1. Run-time окружение, необходимое для функционирования модели.
2. Подсистема описания модели, её создания и менеджмента.
3. Моделирование аналитических утилит (например, утилит мониторинга) и моделирование входных экспериментальных данных.

Так же дополнительно будут полезными следующие компоненты:

1. Подсистема управления моделируемым временем (данное свойство относится к системам эмуляции, а не симуляции)
2. Подсистема визуализации экспериментальных данных

Корректное управление временем важно по нескольким причинам ([3]):

1. Синхронизация между различными узлами (например, для корректного ведения общего лога, или для корректного моделирования экспериментальной нагрузки)
2. Имея возможность изменять время, можно компенсировать некоторые задержки, также можно замедляя время (понижая производительность) конкретного узла добиться эффекта, когда данному узлу будет казаться, что окружающие системы работают быстрее. (это в первую очередь особенность эмуляции, а не симуляции)

На мой взгляд следует уточнить, где используется симуляция, а где эмуляция: как правило узел топологии эмулируется, т.е. создаётся виртуальная система (например, контейнер), которая выполняется некоторые операции (например, генерирует сетевой трафик, или обрабатывает запросы в базу данных, …), при этом когда мы рассматриваем сеть в целом, то корректно говорить о симуляции, так как решается другой класс задач, например, выравнивание задержек между всеми соединениями (настоящими и эмулируемыми, которые соединяют две виртуальные системы в рамках одного хоста). Таким образом, происходит смесь эмуляции и симуляции, которую всегда нужно иметь ввиду.

В результате, например, в статье [6], в которой ведётся рассказ о MaxiNet, основанном на Mininet (программном обеспечении, позволяющем эмулировать узлы на основе контейнеров в рамках одного хоста), предпочитают называть свою систему эмулятором. С другой стороны, например, в статье [5], в которой представляется PRIME (*расширяемая система симуляции для производительных серверов*), говорят о своём продукте, как о симуляторе. При этом в контексте всей топологии в целом, они считают, что симуляция применима лучше эмуляции для исследования альтернатив дизайна и реализации оптимизаций, затрагивающих большую часть системы в целом.

Симуляция и эмуляция всегда имеют ограничения, необходимо следить за корректностью проводимого эксперимента, чтобы построенная модель соответствовала тому, какие выводы мы хотим сделать, так как модель не должна быть слишком общей и пренебрегать критическими для эксперимента параметрами. Так, например, может быть достаточно проблематично провести нагрузочное тестирование для совокупности реального оборудования, или же моделирование не позволяет найти все ошибки в приложениях и их конфигурациях.

Симуляция сети, это компромисс между детализацией описания сети, сложности описания сети, точностью предсказания поведения сети и сложностью настройки модели сети ([3]).

1. Linux Контейнеры
   1. Общая информация о Linux контейнерах

Эмуляция на основе контейнеров называется “*Container Based Emulation*”. Ожидается, что Linux контейнеры выгоднее, чем виртуальные машины, потому что при их функционировании не работает дополнительное ядро операционной системы, что позволяет экономить ресурсы.

В рамках OpenVZ разрабатывается возможность по переносу контейнера на другое оборудование на лету, без его остановки (проект “*CRIO*”). Однако целесообразнее, писать приложения, которые будут работать в контейнере так, чтобы их можно было в любой момент удалить и создать заново, т.е. писать “stateless” приложения, либо держать состояние сохранённым в каком-либо хранилище[[4]](#footnote-4).

Одной из самых удобных возможностей контейнера является его мобильность в смысле быстрого развёртывания по необходимости. Так же это даёт возможность для удобного повторения любого эксперимента, если он был оформлен в виде контейнера, что хорошо подходит для исследовательских работ ([4]).

В работе [7] тестировались возможности по мониторингу системы и количеству контейнеров, которые можно было запустить. Они использовали Dell PowerEdge 1950 сервер с двумя Quad-Core Intel Xeon E5430 (2.66GHz) процессорами, 8GB RAM и Gigabit Ethernet NIC. Для тестирования они использовали контейнеры OpenVZ, на которых был запущен apache, в качестве клиента использовался wget. Такой выбор был сделан потому, что хотелось загрузить в первую очередь не процессор и оперативную память, а память жёсткого диска и сетевые интерфейсы. В результате оборудование позволило запустить 600 контейнеров на сервере.

* 1. Внутреннее устройство Linux контейнеров

Основные типы ресурсов, которые есть в системе это процессорный ресурс, оперативная память, сеть и память жёстких дисков. Однако для исполняемого процесса помимо этого существует в первую очередь окружение: среда передачи сигналов, id пользователя, файловая система, интерфейсы взаимодействия, … Задачи контейнера объединяют главную задачу гипервизора и операционных систем: управление физическими и логическими ресурсами, поддержка переменных окружения и реализация межпроцессорных коммуникаций.

* + 1. Namespaces

Выделяются так называемые “**namespaces**” (“**области видимости**”), которые представляют тот или иной ресурс или элемент окружения. При реализации контейнера, т.е. изолированной среды для выполнения процессов, необходимо, чтобы в каждой из виртуальных сущностей была имитирована своя область видимости:

1. Файловая система: может меняться структура файловой системы и права на отдельные её части (может эмулироваться разными способами[[5]](#footnote-5), позволяет реализовать нестандартные решения, например, распределённую файловую систему UnionFS)

При создании новой области видимости скопируются все параметры текущей области видимости, в которой работает процесс. При этом в областях видимости не видны монтирования файловой системы, которые происходят в других областях видимости, однако монтирования, происходящие в области видимости хост машины, будут видны всем другим. Также файловую систему можно монтировать как приватную, или как общедоступную.

1. “Network namespace” (*Сетевая область видимости*): виртуальные устройства, ip адреса, правила и таблицы маршрутизации, таблицы фильтров, открытые сокеты, …

Сетевой интерфейс и сокет могут принадлежать одновременно лишь одной сетевой области видимости, однако могут быть перенесены в другую. На практике, как правило, создаётся в каждом контейнере (в каждой области видимости сети) свой виртуальный интерфейс, который с помощью механизма Linux-моста соединяется вместе с другими с сетевым интерфейсом хост машины.

1. “UTS namespace” (*unix timesharing*) – изменение имени хоста (*hostname*) и NIS доменного имени в структуре данных, которую возвращает команда “uname”, для процессов в пределах соответствующей области видимости.
2. “IPC namespace” – область видимости, находясь в которой процесс может использовать средства IPC межпроцессорной коммуникации (разделяемая память, семафоры, очереди сообщений) только с процессами из этой области видимости.
3. “PID namespace” – изменение номера процесса в рамках конкретной области видимости. Как правило процессы имеют два идентификатора, один в рамках namespace, а другой в рамках всей системы, однако если процесс находится сразу в нескольких областях видимости, то ему может быть присвоено ещё большее количество идентификаторов.

В зависимости от того, в какой области видимости находится процесс, папка “/proc” будет симулироваться для него системой индивидуально.

1. “User namespaces” (UID/GID) – позволяет процессу одновременно работать от имени разных пользователей в зависимости от области видимости. (например, процесс может иметь id равный нулю в пределах контейнера, однако быть обычным не привилегированном пользователем за пределами namespace)

В мире разработки дополнительно обсуждались следующие типы областей видимости:

1. “Security namespace” (область видимости для изоляции политик безопасности и их проверок) и “Security keys namespace” (целью является предоставление места для хранения ключей безопасности изолированно друг от друга), однако они до сих пор активно не разрабатываются, и сейчас механизм безопасности, раздельный для каждого контейнера представлен только лишь в виде “user namespace”, дополнительная безопасность должна быть реализована другими средствами (например, SELinux).
2. “Device namespace” ([15]) – уже существующие возможности позволяют изолировать в своей области видимости сетевые устройства (интерфейсы) и файловые системы, однако не хватает возможностей по изоляции и разделению всех остальных устройств в Linux.

Например, такая возможность могла бы пригодиться в Android устройствах, для организации двух телефонов в рамках одного, чтобы у человека рабочий и личные телефоны не перемешивались и были изолированы друг от друга, что повысило бы уровень безопасности. Это так же позволило бы ввести более жёсткие политики со стороны компании на корпоративную часть устройства.

Заметим, что в данном случае необходимо решить задачу работы с интерактивными сценариями, когда нужно передавать доступ к устройствам то одному контейнеру, то другому, например, при касании экрана пользователем. (При этом предоставление одновременного доступа для обоих контейнеров было бы не самым безопасным решением в силу того, что один контейнер мог бы вмешиваться в работу с данным устройством другого контейнера). Проблему необходимо не только решить, но и автоматизировать, для корректной работы приложений на устройстве в целом.

Однако подобная концепция так же задевает ещё одну тематику в контексте областей видимости, а именно – сохранение состояния. Т.е. подразумевается, что разделение между корпоративными и личными сущностями телефона является постоянным, в то время как области видимости не сохраняются от запуска к запуску, т.е. должна быть предусмотрена и создана система, которая бы справлялась с подобной задачей. Рассматривать Docker как одну из возможных альтернатив на мой взгляд не удачная идея, потому что несмотря на то, что он может справиться с этой задачей, замораживая контейнеры и создавая из них образы, а потом разворачивая их при перезагрузке системы, он всё же является системой более широкого назначения и не оптимален для решения поставленной задачи в рамках обсуждаемого контекста.

В 2013 году начались обсуждения создания области видимости “device namespace”, были созданы патчи к ядру[[6]](#footnote-6), однако так как полное введение в эксплуатацию этой технологии подразумевает переписывание огромного множества драйверов для различного типа устройств, работа до сих пор не была продолжена.

1. “Time namespace” – данный тип области видимости подразумевает, что в каждом namespace процессы видят своё время. Данная идея не получила распространения, возможно из-за не востребованности разработчиками, потому что внести необходимые поправки во времени можно непосредственно в самом приложении.

Области видимости для процессов отражаются, как виртуальные файлы в файловой системе по пути “/proc/<pid>/ns”

Средства связи родственных процессов (pipes) – остаются не изменёнными, т.е. если родственные процессы находятся в разных областях видимости, то они всё равно могут использовать эти методы для взаимодействия.

* + 1. Cgroup

Для контроля за ресурсами используются **cgroup (Control group – контрольная группа)** – набор процессов, которые объединены по некоторому общему признаку, и ассоциированы с множеством параметров и ограничений. Группы процессов могут быть организованны иерархически в виде дерева, параметры контрольных групп при этом наследуются. Cgroup контролируют сеть, память, процессорное время и блочный ввод/вывод, т.е. является инструментом для группового управления ресурсами, и даёт преимущества:

1. Ограничение ресурсов: например, могут быть настроены для контроля лимита памяти, или контроля доступа к блочным устройствам.
2. Приоретизация: некоторые группы могут получать больше процессорного времени или дискового ввода/вывода.
3. Учёт ресурсов: измерение количества ресурсов, которые используются определёнными процессами.
4. Контроль: заморозка групп процессов, сохранение промежуточных состояний и их перезапуск

Для реализации этих возможностей контрольных групп были созданы следующие подсистемы:

1. *blkio* – подсистема ограничивающая ввод\вывод блочных устройств (диск, usb, …).

Можно задать относительные веса для каждой группы, по каждому устройству (можно задавать как для чтения, так и для записи, как в операциях в секунду, так и в байтах в секунду)

1. *cpu* – подсистема, использующая планировщик задач для распределения доступа к процессорному времени.

Позволяет задавать веса, однако не может задать чёткие ограничения на распределение процессорного времени. Таким образом, мы можем задать в процентном соотношении сколько процессорного времени необходимо отвести данной контрольной группе, однако эти соотношения могут не выполняться в случае, если машина перегружена и процессорного времени на всех не хватает.

1. *сpuacct* – подсистема, генерирующая автоматические отчёты по использованию процессора задачами из контрольных групп.
2. *cpuset* – подсистема способная закрепить выбранную контрольную группу за конкретным ядром процессора и ячейками памяти[[7]](#footnote-7).
3. *devices* – подсистема контролирующая разрешения для контрольных групп на чтение/запись из устройств.
4. *freezer* – подсистема, для остановки и продолжения задач из контрольной группы.
5. *memory* – подсистема, устанавливающая лимит на количество используемой памяти контрольной группой, и генерирующая автоматические отчёты об использовании памяти.

Ограничения контрольной группы, могут задаваться в двух видах одновременно: мягкие ограничения и жёсткие ограничения. Мягкие ограничения не являются принуждающими, однако при достижении жёстких ограничений будут произведены немедленные меры, например, может произойти заморозка всех процессов в контрольной группе, или удаление процесса, поднятие лимитов, извещение userspace, …

1. *net\_cls* – подсистема, помечающая трафик с помощью меток “classid”, позволяющих контроллеру трафика в Linux определить источник пакета, это необходимо для работы tc/iptables.
2. *net\_prio* – подсистема, позволяющая динамически устанавливать приоритеты сетевого трафика на разных сетевых интерфейсах, которые будут учитываться в сетевых очередях.

Обрабатывает только исходящий трафик.

1. *hugetlb* – позволяет ограничивать поддержку больших страниц памяти в ядре Linux в рамках контрольной группы. (данная подсистема мало распространена и почти не используется)
2. *perf\_event* – позволяет отслеживать использование ресурсов контрольной группой в целом, как единого процесса при помощи утилиты “perf”.

Управление контрольными группами происходит через работу с файловой системой, т.е. удаление/создание каталогов, монтирование каталогов в “/cgroup” или для systemd в “/sys/fs/cgroup”. Файловое поддерево является виртуальным, и все изменения будут потеряны после перезапуска. Названия папок не имеют значения, но имеют значения флаги, с которыми происходит монтирование. В файловой системе можно выстраивать иерархию контрольных групп. При создании контрольной группы, так же создаются файлы, в которых указываются необходимые параметры, например, PID процессов, TGID группы процессов. Системы, основанные на systemd, имеют некоторый набор отличий, однако допустимый функционал идентичен.

* + 1. Технологии Namespaces и Cgroups в совокупности

Заметим, что технология namespaces, позволяющая создавать группы процессов, которые по-разному видят доступные ресурсы в рамках одной операционной системы и технология cgroups, позволяющая контролировать группы процессов, друг с другом не связаны и могут использоваться как отдельно, так и совместно. Причём группы процессов для областей видимости и для контрольных групп могут быть совершенно разными.[[8]](#footnote-8)

Таким образом, в совокупности технологии namespaces и cgroups могут использоваться лишь с той степенью, с которой необходимо в конкретном проекте, они очень гибки. И в то же время они обеспечивают главную задачу виртуализации: контроль ресурсов и изоляция.

Однако, у вышеописанных развивающихся технологий есть следующие недостатки[[9]](#footnote-9):

1. Иногда бывает затруднён переход из одной области видимости (или контрольной группы) в другую. В основном это связано с неочевидностью API или какими-либо зависимостями, или правами доступа.
2. Недостатки при жёстком распределении ресурсов. Это касается тех ресурсов, в которых ограничения задаются не жёстким образом, а в виде приоритетов. А именно ограничения на использование процессорного времени и операций ввода/вывода.

В целом расстановка приоритетов исправно работает для распределения ресурсов до тех пор, пока ресурса хватает, однако в случае если перегружать систему, то в результате ресурсы между процессорами начинает разделяться непредсказуемым образом.

1. Проблема безопасности технологий контрольных областей и областей видимости. RedHat отключил возможности namespaces по умолчанию в Linux 7.1, потому что считает, что технология ещё недостаточно развилась и необходимо осознать последствия этой технологии для безопасности системы, и снизить риски и устранить любые векторы атак связанные с этой технологией.[[10]](#footnote-10)

Самая распространённая проблема – это проблема несоответствия областей видимости и областей доступа. Это, например, может приводить к следующим последствиям:

* 1. Некоторые системные вызовы, или отдельные части виртуальной файловой системы могут иметь доступ к информации, к которой контейнер доступ иметь не должен.
  2. Процессы из контейнера, не смотря на то, сколько им доступно того или иного ресурса в соответствии с установленными ограничениями (например, cpu или памяти), всё равно видят наличие всех ресурсов: как доступных, так и не доступных.

Например, примонтированная точка файловой системы в соседней области видимости будет видна всем при использовании утилиты mount, потому что она берёт информацию из “/etc/mtab”, в то время как реально действующие примонтированные системы необходимо смотреть в /proc/mounts. Таким образом, несмотря на то, что доступ к подмонтированной файловой системе отсутствует, мы знаем о существовании точки монтирования. ([11])

На текущий момент системы **LSM** (*Linux Security Modules*), например, SELinux, AppArmor, GRSEC, - совершенно не продуманы с точки зрения namespaces и cgroups. Совместная работа этих технологий находится на стадии обсуждения и фактически не дошла до вопроса реализации.

Таким образом, единственные способы разделения привилегий это userspace namespace, позволяющая изменять UID/GID, изоляция супер-пользователя root (супер-пользователь в рамках контейнера не является им в рамках хоста и не может влиять на всю систему в целом, а только лишь в рамках своей области видимости), а также продолжают работать механизмы CAP\_SYS\_ADMIN, позволяющие разделять привилегии супер-пользователя между другими пользователями.

* 1. Linux Containers (LXC) как подход к виртуализации

Linux Containers (LXC) можно рассматривать как в общем, как технологию, так и в частном, как конкретную реализацию. Linux Container (LXC) – это технология, объединяющая в себе cgroups, namespaces, chroot, Linux Security Modules (LSM) и Mandatory Access Control (MAC) и предоставляющая более абстрактные интерфейсы управления пользователю Linux системы, для организации виртуализации на уровне операционной системы. На практике для пользователя это выражается в виде набора возможностей в пользовательском пространстве для использования преимущественно cgroups и namespaces на более абстрактном уровне (Рисунок 1 отображает на каком уровне работает LXC).

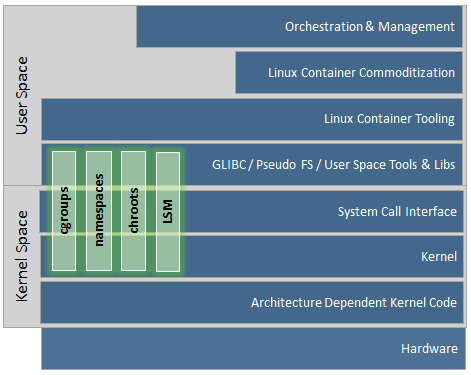


Рисунок 1. Linux Container Technology Stack

Реализация LXC позволяет работать с контейнерами, управляя необходимыми параметрами, на основе которых создаются namespaces и cgroups: hostname, первый процесс и его UID/GID, сетевые параметры, файл для журналирования, точки монтирования файловой системы и доступ к ним, а также корневой файловой системы, настройка любых параметров cgroup, контекста SELinux, перехватчиков тех или иных событий, происходящих с контейнером, также можно передать дополнительно переменные окружения.

LXC поддерживают совместимость с функционалом “Copy on Write”[[11]](#footnote-11), основанную на файловой системе, позволяющей образовывать каскадно-объединённое монтирование (например, AUFS). Это позволяет запускать множество одинаковых контейнеров на базе единого экземпляра файловой системы, а изменения, производимые каждым контейнером записывать отдельно по мере необходимости, и по запросу пользователя скомпоновать образ новой файловой системы с учётом всех изменений. Такой подход позволяет запустить в очень сжатые сроки огромное количество контейнеров, и экономить на лишних экземплярах одной и той же файловой системы.

При сравнении возможностей OpenVZ и LXC ([21]), единственное, в чём LXC уступает своему конкуренту, это в отсутствии возможности контролировать качество предоставляемого сервиса по блочному вводу/выводу. Однако неформально считается, что контейнеры в рамках OpenVZ являются более безопасными (изолированными), по сравнению с LXC.

Из возможных недостатков LXC следует выделить отсутствие автоматического создания резервных копий контейнеров, отсутствие возможности восстановления в случае проблем с хост-системой и отсутствие централизованного средства для управления сетью. Однако первые 2 недостатка являются, в первую очередь, областью ответственности самих приложений, а не контейнеров, и они сами или сторонние специализированные утилиты должны заботиться о наличии резервных копий, защиты от ошибок и различных сбоев. А последний из недостатков является областью ответственности средств, занимающихся оркестрацией. История показала, что самый главный недостаток реализации LXC заключается в том, что предоставляемый интерфейс взаимодействия был не оптимален для проектов по менеджменту контейнеров (например, Docker), из-за чего он перешёл на собственную реализацию концепции LXC – libcontainer. Весомые недостатки заключаются в недоказанной безопасности Linux контейнеров[[12]](#footnote-12), поэтому при использовании LXC рекомендуется включать систему разграничения прав (например, GrSecurity или SELinux), и отсутствие возможности миграции произвольного контейнера без его закрытия и перезапуска. (Дальше всего в реализации этой возможности продвинулся проект CRIU)

* 1. Docker - система менеджмента легковесной виртуализации

Docker предоставляет возможности по развёртыванию приложений, оформленных в виде контейнеров, настройке предоставляемых ресурсов и установлении ограничений. По своей сути Docker приблизился к тому, чтобы расценивать инфраструктуру, как приложение, из чего следует одна из важных концепций: Docker образы контейнеров не имеют зависимостей, они самодостаточны.

Docker имеет клиент-серверную архитектуру: демон управляет образами и контейнерами, в то время как клиент, посылающий команды, может находиться на любой другой машине. Работа происходит с тремя основными сущностями: *Docker образ* - шаблон, доступный только на чтение, является основной единицей дистрибуции (распространяются в формате Dockerfile, содержащего образ, сопутствующую информацию и инструкции для правильной подготовки необходимых ресурсов для запуска контейнера), *Docker реестры* - хранилища образов[[13]](#footnote-13), и *Docker контейнеры* - исполняемая единица, которая может быть запущена, приостановлена, уничтожена, …

Docker, используя концепцию Linux Container[[14]](#footnote-14), предоставляет следующие возможности:

1. Развёртывание контейнеров между машинами, при этом образ контейнера может быть запущен на любой машине, поддерживающей Docker технологию, выступающую при этом в качестве “Open Virtualization Appliance”
2. Автоматическая сборка: создание контейнеров с учётом систем управления конфигурациями (например, Chef, Puppet, Maven, Ansible, Salt, …)
3. Поддержка версий Docker контейнеров, работающая, аналогично системе контроля версий git, запоминая изменения внутри контейнеров.

Так же поддерживаются возможности по использованию публичных и частных репозиториев, таких как “Docker Hub”.

1. Переиспользование компонент: каждый контейнер может быть использован в качестве основы для создания другого контейнера.
2. Инструментарий: CLI (command line interface) и REST API для управления Docker.

Docker следует воспринимать именно как средство для распространения контейнеров и автоматизации их запуска. Однако несмотря на свои возможности он не может выступать в качестве полноценной платформы виртуализации, потому что в нём отсутствуют, или не имеют должной степени развития компоненты для сбора логов, аутентификации, сбора статистик, оркестрации. Таким образом, Docker является средством виртуализации, на основе которого можно создавать более полную систему, позволяющую управлять большими виртуальными вычислительными системами. Однако это не значит, что он не может использоваться и в одиночку в случае если создаваемая система не велика и не требует традиционных для облачных структур сервисов.

В открытых источниках часто говорится о том, что Docker очень лёгок в использовании, что приводит к некоторым заблуждениям в широкой публике. Следует понимать, что он является ещё одним средством управления, призванным для упрощения развёртывания Linux контейнеров. Если проект не использовал контейнеры, то внедрение Docker лишь добавит сложность в системе, которая будет располагаться в области ответственности администраторов. С точки зрения администрирования при использовании Docker будет необходимо контролировать ([20]):

1. Контейнер, занимающийся оркестрацией развёртывается мгновенно, без каких-либо задержек.
2. Контейнер, занимающийся оркестрацией должен уметь производить откаты других контейнеров и настроек.
3. Необходимо поддерживать сеть между контейнерами на нескольких хостах.
4. Поддержка системы сбора телеметрии с различных контейнеров.
5. Централизованный сбор журнала сообщений со всех контейнеров.
6. Обработка данных контейнеров (баз данных, объектных хранилищ, файловых систем, …). Так же необходимо, чтобы некоторая система задавала правила, в соответствии с которыми будут сохраняться или удаляться данные, которые записал контейнер в предоставленную ему файловую систему.

Эти возможности должны поддерживаться для масштабируемой системы, и для этого придётся либо использовать системы подобные OpenStack, переделав его для работы с Docker как с гипервизором[[15]](#footnote-15), позволяющим запустить несколько виртуальных машин, либо тем или иным образом адаптировать “Configuration Management Tools” (например, Ansible, Chef, Puppet). Так же эти системы будут нести ответственность за развёртывание машин, на которых запущены Docker-серверы ([20]). Однако все эти подходы являются лишь способами адаптации уже существующих систем управления, имеющих другие назначения. Так же их адаптация усложняется тем фактом, что API Docker и его внутреннее устройство претерпевают большие изменения от версии к версии, что затрудняет использование некоторых возможностей и тонкую настройку для Docker контейнеров.

Docker не упрощает разработку приложений самих по себе, и не ставит подобной задачи, всё равно должен реализовываться безопасный доступ с принципом минимальных привилегий, необходимо делать резервное копирование, отслеживать поведение критических сервисов, иметь документацию, … ([20])

Уровень безопасности Docker по сравнению с виртуальными машинами ниже, это связано с тем, что контейнер гораздо больше взаимодействует с ядром операционной системы.

Docker позволяет настроить все основные namespaces при запуске контейнера (PID, UTS, IPC, network, user security configuration) и все основные cgroups (Memory, CPU, blockIO), при этом в параметрах памяти можно указать мягкие и жёсткие, ограничения на файл подкачки, на общее количество памяти и на номера доступных блоков памяти в случае использования NUMA систем. Для процессора можно указать номера ядер, на которых должен исполняться контейнер и приоритет, в соответствии с которым, планировщик задач будет распределять процессорное время между различными контейнерами. Однако отсутствует возможность жёстко задать количество процессорного времени, которое будет доступно контейнеру, чтобы в случае превышения этого лимита контейнеру больше не выделялось ресурса. Аналогичный недостаток присутствует и для блочного ввода/вывода. Есть возможность указать приоритеты, но не задать жёсткий лимит на запись/чтение.

Этот недостаток является важным как для реальных систем (например, при продаже услуг хостинга), так и для систем моделирования, которые могут быть нацелены на возможность моделирования поведения сети в случае, если один из узлов перегружен и не справляется.

Если рассматривать конкурирующую систему, называемую virtuozza[[16]](#footnote-16), основанную на OpenVZ, то там поддерживаются дополнительные возможности по ограничению ресурсов. В ней можно задать жёсткие лимиты для контейнера на использование процессорного времени, можно задавать квоту или указывать приоритеты на блочный ввод/вывод для каждого контейнера или для каждого отдельного пользователя.

* 1. Сравнение Linux контейнеров и виртуальных машин.

Виртуальные машины являются более безопасными, по сравнению с контейнерами, так как меньше взаимодействуют с ядром хост-системы, и, следовательно, в них потенциально меньше уязвимостей. Также гипервизоры могут запускать разнотипные операционные системы. С другой стороны, у контейнеров есть ряд преимуществ:

1. Так как в случае использования контейнеров отсутствует дополнительная операционная система на каждой гостевой системе, то это ускоряет работу, по сравнению с гипервизорами, все из которых используют аппаратную поддержку (например, VT-x, VT-d, AMD-V) и уже вряд ли сильно ускорят свою производительность в будущем.

В статье [22] проводилось исследование, где сравнивались контейнеры на основе LXC, Docker, гипервизор KVM и производительность не виртуальной системы. При этом раздельно изучались характеристики производительности процессора, памяти, блочного ввода/вывода и сети. В статье [8] исследовался аналогичный вопрос, однако более детально изучался вопрос производительности сети в различных режимах. По итогам, можно сделать следующие выводы:

* 1. Производительность вычислений на процессоре и доступа к оперативной памяти (как случайный доступ, так и последовательный) почти не зависит от условий виртуализации и замедляется не больше, чем на 2-3%.
  2. Производительность блочного ввода/вывода для контейнеров не отличается от производительности без виртуализации. Однако для KVM, скорость последовательного доступа почти не отличается, в то время как скорость случайного доступа к данным в 2 раза ниже, из-за того, что на одну операцию чтения приходится больше циклов работы процессора.
  3. В случае анализа производительности сети, результаты сильно варьировались. Так, например, в случае использования NAT для Docker и KVM – RTT (*round trip time*) вырастал почти вдвое, как для UDP, так и для TCP (стоит отметить, что KVM был несколько быстрее, чем Docker).

Для гипервизоров существуют техники ускорения[[17]](#footnote-17), которые приводят к тому, что накладные расходы на процессор при использовании KVM будут отсутствовать на этапе отправки, однако при принятии трафика накладные расходы в два раза больше, по сравнению с использованием контейнеров. Docker имел накладные расходы примерно на 20% больше, чем не виртуальная система, как при посылке, так и при отправке TCP трафика.

Измерять каждую характеристику отдельно не является правильным решением для сравнения производительности, так как в данном случае не учитываются накладные расходы, основанные на использовании ядра операционной системы. Так же мало учитываются переключения контекста. Для этого в статье [8] была исследована производительность базы данных MySQL и Redis (хранилище данных типа ключ-значение). В случае MySQL, накладные расходы для контейнера составляли 2%, в то время как для KVM – 40%. В случае Redis, проблемы с производительностью проявлялись в первую очередь для сетевого обмена данными: для Docker в режиме работы с NAT дополнительно загружался процессор, и для KVM, для которого увеличивалась задержка. В результате количество запросов, обрабатываемых на KVM было на 10% меньше, а для Docker на 20% меньше, по сравнению с не виртуальной системой.

1. В рамках виртуализации на основе контейнеров очень легко реализовать совместное использование ресурсов, таких как память и блочный ввод/вывод (например, “shared memory” или “AUFS”), в то время как гипервизорам это почти недоступно. (Технология “shared memory” в контейнерах OpenVZ распространяется даже на случай, когда в различных контейнерах запускают одинаковые программы, и тогда в память загружается лишь одна копия бинарного файла)

Это также позволяет запускать множество контейнеров в более сжатые сроки, так как они могут использовать единую копию файловой системы.

1. Контейнеры более гибки при перераспределении ресурсов на ходу.
   1. Место контейнеров в реальном мире

Контейнеры в сетевом мире можно рассматривать с точки зрения двух разных концепций: концепция NFV (Network Function Virtualization) (т.е. на сколько контейнеры подходят для реализации возможности виртуализации сетевых функций) и концепция моделирования сетей (для проведения различного рода тестирований). Так же контейнеры можно рассматривать с точки зрения возможностей, которые они предоставляют обычным приложениям, использующих сетевые интерфейсы.

Примерами использования контейнеров в концепции NFV могут служить:

1. Перенос оказываемых сервисов оператором для компаний и их филиалов с выходного маршрутизатора в центр обработки данных оператора, а между выходным маршрутизатором и ЦОД-ом установка канала (примерами сервисов могут быть firewall, NAT, DPI, …)
2. Перенос сервисов, оказываемых частным лицам в центр обработки данных операторов (например, фильтрация “контента для взрослых”).

Контейнеры при моделировании могут использоваться, чтобы проводить исследование по сети в целом (например, скорость распространения информации в той или иной топологии (будь это червь или какая-то служебная информация о сети) или проверка сходимости протокола) или чтобы моделировать сеть для тестируемого объекта (например, SDN контроллера или для тестирования CDN ([5])).

При моделировании крупной сети, как правило запускается большое множество однотипных контейнеров, поэтому возможность использования таких файловых систем, как AUFS, очень важна, так как это экономит место на файловой системе и сильно ускоряет массовый запуск контейнеров. Так же очень важно уметь контролировать нагрузку на серверах при создании модели, чтобы можно было определить наличие перегрузок, которые могут искажать эксперимент.

При использовании контейнеров для реализации NFV очень важно иметь возможность зафиксировать контейнер на конкретном ядре процессора, чтобы минимизировать потери от переключения контекста, а также, чтобы распределить ядра процессора между контейнерами, тем самым распределив ресурс между ними. Помимо этого, необходимо иметь возможность давать доступ к драйверу сетевой карты, чтобы можно было реализовывать сетевые приложения с использованием таких особенностей сетевых карт, как DPDK, дающих существенное ускорение.

В статье [3] говорится, что система моделирования NPS может быть использована для изучения таких задач, как распространения вредоносного ПО в сети, сходимости протоколов маршрутизации, проверка производительности сети (максимальная пропускная способность через определённый узел, общий уровень задержек и потерь) и разработка сетевых приложений.

Для обычных приложений, запускаемых в контейнерах требуется в первую очередь возможность работы без дополнительных изменений в самом приложении, его изоляция и правильное распределение ресурсов.

1. Особенности моделирования на основе контейнеров
   1. Примеры существующих систем моделирования сети

Существует множество эмуляторов сети: Mininet, Mininet-Hifi, vEmulab, PRIME, NetKit, Trellis, CORE, Estinet (платный пример), OMNeT++. Так же существует ряд надстроек, которые дополняют некоторые системы, позволяя создавать распределённые сетевые модели: MaxiNet, NPS. Существуют специализированные системы моделирования, например, RINSE ([1]), целью которой является предоставление некоторой модели для обучения сетевой безопасности.

Mininet имеет открытый исходный код, быстрее своих аналогов, таких как vEmulab ([4]), и имеет необходимые возможности, поэтому является достаточно популярной системой.

Для каждого эмулируемого сетевого хоста Mininet создаёт отдельный контейнер, в котором настраивается network namespace, чтобы у контейнера был свой сетевой интерфейс, который соединяется с некоторым программным маршрутизатором (например, Open vSwitch (их может быть несколько в рамках одного хоста)). Mininet не даёт никаких гарантий о производительности, в отличие от vEmulab, который может ограничивать скорость полосы пропускания. Таким образом, Mininet использует одну единственную возможность контейнера - сетевую область видимости.

PRIME специализируется на моделировании в рамках многопроцессорных машин с общей или распределённой памятью ([5]). Данная система была успешно использована для моделирования сети с общим количеством в 45 000 узлов для тестирования CoralCDN.

Mininet-Hifi (high level of confidence) – является надстройкой над Mininet, дополнительно предоставляющей ограничение производительности, подготовку ресурсов и мониторинг производительности, позволяя обеспечить такие характеристики, как уровень пропускной способности, производительность очереди и время ожидания ([4]).

Maxinet и NPS (*Network Prototyping Simulator*) представляют из себя системы, которые на базе Mininet создают распределённую сеть, между несколькими произвольными хостами. Это позволяет увеличить число симулируемых узлов до тысяч. Maxinet в качестве маршрутизатора не используют OpenVSwitch, так как он плохо работает в случаях, когда на одном хосте много интерфейсов ([6]).

Система NPS позволяет, используя Supervisor Console, подключаться и конфигурировать каждый экземпляр Mininet, запускаемый на каждом кластере (Cluster Node). Для управления трафиком в рамках кластеров и между ними используются OpenFlow контроллеры ([3]).

Предпринимались успешные попытки строились распределённые системы на базе vEmulab с использованием в качестве контейнеров FreeBSD jails. Однако не все эмуляторы сети подходят для построения распределённых моделей, например, Estinet, устроен так, что его трудно масштабировать ([6]).

* 1. Основные требования к системам моделирования на основе легковесных контейнеров[[18]](#footnote-18)

Такие системы моделирования, как vEmulab, Mininet, NPS позволяют виртуализовать хост, маршрутизатор и сеть, при этом поддерживая прозрачность для приложения, точность производительности, возможность интерактивной работы и воспроизводимости эксперимента, эффективное использование ресурсов ([3], [9]).

При моделировании необходимы следующие основные возможности:

1. Скорость работы модели должна иметь возможность соответствовать производительности современных сетей. Это в первую очередь необходимо для сетевых приложений, учитывающих в своей работе сетевые задержки.
2. Гибкость системы моделирования (как создание самых разнообразных топологий, так и использование разнообразных приложений и моделируемых подсистем)
3. Возможность масштабирования системы моделирования на несколько вычислительных систем. При этом увеличение моделируемой сети должно линейно зависеть от количества используемых вычислительных систем. Должна быть решена проблема неравномерности задержек между виртуальными узлами на общей хост-системе и при коммуникации узлов через среду передачи данных между хост-системами. (либо различие задержек не значительно, либо необходимо принимать меры по выравниванию времени между виртуальными узлами ([6]))
4. Низкая стоимость модели, что сопряжено с её максимальной производительностью, т.е. с минимизацией накладных расходов.
5. Воспроизводимость симуляции (хорошо коррелирует с концепцией контейнеров по упаковыванию окружения вместе с приложением)
6. Прозрачность для приложений, как с внешней точки зрения, так и из внутренней, т.е. с одной стороны, модель должна быть представлена для внешнего приложения, как множество хостов, а с другой стороны, каждое приложение, которое запускается в рамках хоста должно видеть модель, как сетевую инфраструктуру.

Не должно возникать проблем в случае необходимости подключить модель к интернету.

Следует также учитывать, что такие приложения, как tcpdump, могут быть особенно чувствительными к способу эмулирования инфраструктуры, так как они работают в первую очередь не с ip-адресом, а с сетевым интерфейсом.

1. Необходимы методы для измерения нагрузки при работе модели, чтобы можно было говорить о корректности эксперимента, а также необходимы методы по ограничению ресурсов, с целью изучения поведения модели при перегрузках в тех или иных местах и для фиксации нагрузки в необходимых узлах и соединениях.

Как правило средства для контроля перегрузок на хост-системах создаются отдельно от самой модели, однако модель должна быть настраиваемой, чтобы можно было контролировать распределение виртуальных узлов и их коммуникационных каналов поверх вычислительной системы.

1. Возможность запуска большого количества экземпляров частей модели (важные ограничения, например, на максимальное количество виртуальных интерфейсов, на максимальное количество процессов, …).
2. Возможность интерактивного использования, т.е. возможность вмешиваться в работу модели.

Требования, реализация которых в контейнерах отсутствует или вызывает затруднения:

1. Отсутствует способ ограничения пропускной способности виртуального интерфейса, выделенного для контейнера. Также в случае, когда в рамках одного хоста запускается множество контейнеров, и нету возможности выделить каждому из них отдельный процессор, то приходится расставлять приоритеты для распределения процессорного времени, таким образом нету возможности жёстко фиксировать производительность каждого контейнера.
2. В реальном мире уже подходят к возможным ограничениям в Linux системах на количество допустимых ресурсов, таких, например, как виртуальных сетевых интерфейсов, количество процессов.
3. Наблюдаются некоторые проблемы по сбору статистики использования памяти, процессорного ресурса, использованию блочного ввода/вывода. И если нагрузка создаётся одним контейнером, то вычленить его из множества контейнеров является нетривиальной задачей, таким образом, в случае если один из контейнеров начнёт по тем или иным причинам переиспользовать ресурс, то даже не смотря на его, возможно, низкий приоритет, он, фактически, может провести DoS (*Denial of Service* – отказ в обслуживании на основе недостаточности ресурсов вычислительной системы) атаке как минимум в рамках своей хост-системы.
   1. Возможности легковесных контейнеров по контролю выделяемых ресурсов

Есть два принципиальных способа контроля ресурсов с точки зрения моделирования: первый способ заключается в том, чтобы ограничить ресурсы каждого контейнера, с помощью встроенных в него механизмов (это реализуется с помощью cgroups, однако этот подход не самодостаточен), другой способ заключается в том, чтобы следить за степенью нагрузки на вычислительную систему, и запускать в рамках одного хоста столько контейнеров, сколько он может обработать без исчерпывания лимита по тем или иным ресурсам. В случае реализации второго подхода, необходимо создавать дополнительное средство для мониторинга и автоматического изменения модели в зависимости от происходящей ситуации. Так же следует помнить о “принципе неопределённости Гейзенберга”, который в данном случае подразумевает, что чем чаще снимаются показатели с вычислительной системы, тем менее производительной она становится[[19]](#footnote-19). В зависимости от модели, наличие перегрузки на вычислительной системе может означать с одной стороны, неудавшийся эксперимент, из чего будет следовать, что нужно перезапускать модель, но с меньшим количеством контейнеров в рамках одного хоста, а с другой стороны, если эксперимент не считается сразу неудачным, то система может оперативно перенести контейнер на другой вычислительный узел, сохранив топологию.

В статье [4] эмпирически было выявлено, что использование времени простоя процессора является достаточно качественным критерием для оценки загруженности вычислительного узла. И по мнению авторов [3], моделирование сети на основе контейнеров LXC может быть высокоточным, если нигде не создавались незапланированные перегрузки элементов модели.

1. Заключение

Легковесные контейнеры в последние несколько лет стали очень активно использоваться в сфере виртуализации (например, NFV) и в сфере моделирования. Их возможности (гибкость, быстродействие, самодостаточность контейнеров, низкие сопутствующие затраты виртуализации) оказались очень полезны в сетевом моделировании. При этом моделирование требует не только виртуализации, т.е. управление и разделение, сетевой части операционной системы, но и многих других ресурсов: процессора, памяти, блочного ввода/вывода. Не все возможности реализованы в контейнерах необходимым образом, так, например, при моделировании не хватает возможности жёсткого контроля процессорного ресурса, наблюдаются проблемы с безопасностью, не хватает возможностей по разделению устройств между областями видимости.

Однако есть и сильные стороны, заключающиеся в возможности одновременного использования ресурсов несколькими контейнерами, что совершенно недоступно в случае использования виртуализации на основе гипервизоров (например, файловой системы или виртуальной памяти), это позволяет снижать затраты при создании большого количества однотипных контейнеров, что часто случается при моделировании сети и в реальном мире, так же контейнеры работают не медленнее, чем гипервизоры, а в некоторых случаях существенно быстрее.

Для легковесной виртуализации создаются системы менеджмента (например, Docker), однако в них есть недостатки (например, проблемы с оркестрацией), которые в будущем могут быть устранены.

Таким образом, возможности легковесной виртуализации очень хорошо подходят для моделирования сетей, в которых необходимо создавать огромное множество однотипных узлов.

Цель реферата достигнута, в силу того, что были изучены возможности и недостатки легковесной виртуализации и были сопоставлены с функциональными запросами систем моделирования.

1. Список литературы
2. Liljenstam M. et al. Rinse: The real-time immersive network simulation environment for network security exercises //Principles of Advanced and Distributed Simulation, 2005. PADS 2005. Workshop on. – IEEE, 2005. – С. 119-128.
3. Im E. G. et al. Hybrid modeling for large-scale worm propagation simulations //Intelligence and Security Informatics. – Springer Berlin Heidelberg, 2006. – С. 572-577.
4. Antonenko V., Smelyanskiy R., Nikolaev A. Large scale network simulation based on hi-fi approach //Proceedings of the 2014 Summer Simulation Multiconference. – Society for Computer Simulation International, 2014. – С. 4.
5. Handigol N. et al. Reproducible network experiments using container-based emulation //Proceedings of the 8th international conference on Emerging networking experiments and technologies. – ACM, 2012. – С. 253-264.
6. Liu J., Li Y., He Y. A large-scale real-time network simulation study using prime //Winter Simulation Conference. – Winter Simulation Conference, 2009. – С. 797-806.
7. Wette P. et al. Maxinet: Distributed emulation of software-defined networks //Networking Conference, 2014 IFIP. – IEEE, 2014. – С. 1-9.
8. Jia Q., Wang Z., Stavrou A. The Heisenberg Measuring Uncertainty in Lightweight Virtualization Testbeds //CSET. – 2009.
9. Felter W. et al. An updated performance comparison of virtual machines and linux containers //technology. – 2014. – Т. 28. – С. 32.
10. Hibler M. et al. Large-scale Virtualization in the Emulab Network Testbed //USENIX Annual Technical Conference. – 2008. – С. 113-128.
11. “Cgroups” [Электронный ресурс]. / Paul Menage. – Режим доступа: <https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt>
12. Rosen R. Resource management: Linux kernel Namespaces and cgroups //Haifux, May. – 2013.
13. “Resource Management Guide” [Электронный ресурс] / Red Hat Inc. – Режим доступа: <https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Resource_Management_Guide/>
14. Jerome Petazzoni. Container's Anatomy. Namespaces, Cgroups and some filesystem magic // 19 Aug. – 2015
15. Boden Russell. Realizing Linux Containers (LXC). Building blocks, Underpinnings & Motivations // 11 Mar. – 2014
16. “Namespaces in operation, part 1: namespaces overview” [Электронный ресурс] / Michael Kerrisk. – Режим доступа: <https://lwn.net/Articles/531114/>
17. “Device namespaces” [Электронный ресурс] / Jake Edge. – Режим доступа: <https://lwn.net/Articles/564854/>
18. Reshetova E. et al. Security of OS-level virtualization technologies //Secure IT Systems. – Springer International Publishing, 2014. – С. 77-93.
19. Kumar A. S. Virtualizing Intelligent River R: A Comparative Study of Alternative Virtualization Technologies : дис. – Clemson University, 2013.
20. “Docker Docs” [Электронный ресурс] / Docker Inc. – Режим доступа: <https://docs.docker.com/>
21. “Docker misconceptions” [Электронный ресурс] / Matt Jaynes. – Режим доступа: <https://valdhaus.co/writings/docker-misconceptions/>
22. “Feature comparison of different virtualization solutions” [Электронный ресурс] / OpenVZ Inc. – Режим доступа: <https://openvz.org/Comparison>
23. Morabito R., Kjällman J., Komu M. Hypervisors vs. Lightweight Virtualization: a Performance Comparison.
24. Xavier M. G. et al. Performance evaluation of container-based virtualization for high performance computing environments //Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on. – IEEE, 2013. – С. 233-240.

1. Примерами эмуляторов могут служить: гипервизор QEMU, FPGA (*программируемая пользователем вентильная матрица*). [↑](#footnote-ref-1)
2. Существует множество проектов подобных систем. Одними из самых успешных являются OpenStack, Docker, Amazon Web Services EC2, Virtuozzo. [↑](#footnote-ref-2)
3. Говоря о разумной степени производительности, подразумеваются гипервизоры, которые не эмулируют платформу, отличную от платформы хост-системы (примерами таких гипервизоров могут служить VMware ESXI, KVM, но не QEMU) [↑](#footnote-ref-3)
4. Задача целостности приложения и защиты от внезапного падения должна в первую очередь решаться самим приложением, а не средой (контейнером) [↑](#footnote-ref-4)
5. Считается расширением возможностей, предоставляемых “chroot” [↑](#footnote-ref-5)
6. <https://github.com/Cellrox/devns-patches/wiki/DeviceNamespace%3APatches> [↑](#footnote-ref-6)
7. Управление распределением памяти между процессами может быть важно для мультипроцессорных систем (например, NUMA) [↑](#footnote-ref-7)
8. Раньше каждой cgroup соответствовал namespace, но потом от этого отказались. [↑](#footnote-ref-8)
9. Из-за того, что технологии всё ещё развиваются, некоторые возможности ещё не были реализованы, а некоторые недостатки до сих пор не осознанны. [↑](#footnote-ref-9)
10. Стоит отметить, что в сентябре 2015 года namespaces были включены в дистрибутив RedHat в качестве “Technology Preview”. До тех пор одним из ключевых сообществ, которые дорабатывали данные возможности были разработчики дистрибутива Fedora, в котором namespaces присутствовали. (<http://rhelblog.redhat.com/2015/07/07/whats-next-for-containers-user-namespaces/>) [↑](#footnote-ref-10)
11. Совместная работа контейнеров с разделяемыми файловыми системами преимущественно поддерживается проектом Docker. [↑](#footnote-ref-11)
12. Данная проблема имеет место быть в первую очередь из-за незакрытого вопроса безопасности технологий namespace и cgroup. [↑](#footnote-ref-12)
13. Публично доступный Docker Hub является примером реестра докер образов. Реестры могут быть как публичные, так и приватные. [↑](#footnote-ref-13)
14. Изначально проект Docker базировался на технологии LXC, однако перешёл на собственную реализацию подхода Linux Containers под названием “libcontainer” (считается, что LXC предоставлял слишком высокоуровневый интерфейс, который не удовлетворял требования Docker) [↑](#footnote-ref-14)
15. Данное решение не будет являться правильным, в силу того, что не следует строить одну систему менеджмента поверх другой. Гораздо более продуктивно будет использовать LXC напрямую из OpenStack. [↑](#footnote-ref-15)
16. Проект является платным, и работает только для специально создаваемого дистрибутива Linux. [↑](#footnote-ref-16)
17. Обход гипервизора и специальные драйверы [↑](#footnote-ref-17)
18. Параграф использует информацию, основанную на статьях [4], [6], [9] [↑](#footnote-ref-18)
19. В статье [7] были приведены характерные числа, так при измерениях каждые 0,001 секунду у системы заметно снижалась производительность, а при измерениях каждые 0,01 измерения были почти незаметны [↑](#footnote-ref-19)