Технологии облачных вычислений

Электронное учебное пособие

Оглавление

Предисловие

Учебное пособие предназначено для оказания методической помощи по дисциплине ***Технологии облачных вычислений*** студентам, обучающимся по направлению 09.03.02 Информационные системы и технологии. Пособие может быть полезно обучающимся по другим направлениям, связанным с разработкой и применением современных информационных систем и технологий для получения основных сведений об облачных технологиях, средах формирования виртуальной среды разработки.

Пособие направлено на формирование у учащихся компетенции «осуществлять выбор платформ и инструментальных программно-аппаратных средств для реализации информационных систем». Учебное пособие ставит своей целью формирования: *знаний* об основных технологиях, реализуемых в концепции облачных вычислений (системы виртуализации, гипервизоры, системы управления и балансировка облачных ресурсов, обеспечения гарантированного качества обслуживания); *умений* сформировать виртуальную облачную среду разработки программного обеспечения с использованием Vagrant; *овладений* программными инструментами настройки конфигурирования виртуальных облачных ресурсов и способами экспериментальной оценки эффективности облачных инфраструктур.

1. Основные понятия и классификация облачных систем

Облачные вычисления являются сегодня наиболее популярной концепцией информационных систем. Данная концепция представляет собой результат развития целой цепи концепций построения информационных систем (рис. 1).

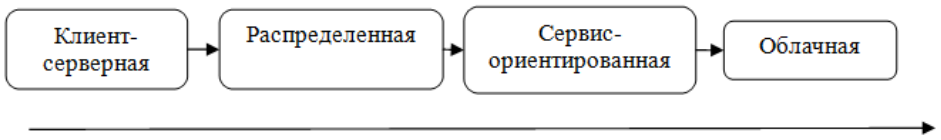

Рис. 1. Последовательность развития концепций

Клиент-серверная архитектура — вычислительная архитектура, в которой вычислительные задачи и сетевая нагрузка распределены между поставщиками услуг, называемыми серверами, и заказчиками услуг, называемыми клиентами. По сути, это первая архитектура, позволяющая перенаправить вычислительную нагрузку на сервера, а не выполнять операции на клиентах. Клиент-серверная архитектура может включать в себя множество уровней серверов (трехзвенная архитектура, многозвенная архитектура). Данная архитектура и ее разновидности имеют недостатки надежности, так как при выходе сервера из строя, система становится неработоспособной. Повышенные качественные требования к серверам привели к появлению распределенных систем.

Распределенная архитектура — это набор независимых вычислительных систем, представляющихся их пользователю единой объединенной системой. Основная задача распределенных систем — облегчить доступ к удаленным ресурсам и контроль совместного использования этих ресурсов. Для решения этих задач система должна удовлетворять следующим требованиям:

- прозрачность — сокрытие разницы в способах представления данных и способы доступа к ресурсам;
- использование открытых протоколов — применение синтаксических и семантических правил, основанных на известных или введенных и опубликованных стандартах и формализованных протоколах;

– масштабируемость — возможность изменение по отношению к размеру, к географическому размещению и к управлению.

Распределенная архитектура, вообще говоря, является разновидностью клиент-серверной концепции, так как при выполнении любых операций в системе хост выполняет функции клиента или сервера. Данная концепция позволила решить проблемы с надежностью и распределить нагрузку, однако распределенные системы сложны в реализации. Также в них имеют место проблемы с синхронизацией данных.

Сервис-ориентированная архитектура (SOA) — это специфическая распределенная архитектура, состоящая из сервисов. Под вычислительными сервисами понимаются небольшие обособленные программные элементы, которые решают одну задачу и могут быть использованы во многих приложениях и другими сервисами. SOA основывается на принципе слабой связанности, что означает, что каждый сервис — это изолированная сущность с ограниченными зависимостями от других общих ресурсов, таких как базы данных, приложения или разные API. Такая архитектура системы позволяет создать уровень абстракции между потребителями и разработчиками. Это позволяет менять реализацию и обновлять без ущерба для потребителей сервиса.

SOA архитектура легко масштабируема. На каждом вычислительном узле комплекса может работать любое количество сервисов, а они, в свою очередь, могут использовать другие сервисы. В результате вычислительные сервисы могут быть объединены, обеспечивая функциональность приложения. Принципы проектирования SOA широко используются при разработке и интеграции информационных систем.

W3C (Web Services Architecture Working Group Consortium — рабочая группа консорциума веб-сервисной архитектуры) рассматривает SOA как форму распределенных систем и характеризует следующими свойствами:

– Логическое представление: сервис — это представление программ, баз данных, бизнес-процессов и т. д.
– Ориентация на сообщения: сервис формально определен в терминах сообщений, которыми обмениваются поставщик агентов и заказчик агентов. Потребителю сервиса не нужно знание внутренней структуры агента: язык реализации, структуры процесса, структуры

базы данных. Основное преимущество: нет необходимости знать внутреннюю структуру агента. Это позволяет включить любой компонент программного обеспечения или приложения, которое «завернуто» в код, и представить в виде услуги клиенту.

- Ориентация на описание: сервисы описываются программно обрабатываемыми метаданными. Описываются только внешние и важные детали сервиса. Семантика службы должна быть документирована прямо либо косвенно.
- Степень детализации: сервисы, как правило, имеют небольшое количество операций для работы с большими и сложными сообщениями.
- Ориентация на сетевое взаимодействие: сервисы, как правило, ориентированы на использование по сети, хотя это не абсолютное требование.
- Независимость от платформы: сообщения отправляются независимо от платформы в стандартизированном формате. XML является наиболее подходящим форматом, который соответствует этому ограничению.

Облачные вычисления — информационно-технологическая концепция, подразумевающая обеспечение удаленного доступа к вычислительным ресурсам: сетям передачи данных, серверам, устройствам хранения данных, приложениям и сервисам.

Основная задача *облачных технологий* — это создание виртуальной вычислительной *облачной* инфраструктуры, состоящей из виртуальных распределенных ресурсов, обеспечивающих удаленное предоставление услуги доступа к инфраструктуре с гарантируемым требуемым уровнем обслуживания пользователя.

Облачные технологии включают решение следующих задач:

- выполнение приложений в облаке;
- виртуализация оборудования и вычислительных ресурсов;
- обеспечение одновременной работы большого количества клиентов, количество которых может меняться.

Когда речь идет об услуге (особенно платной), актуальным является обеспечение высокого уровня обслуживания клиентов, в случае информационных технологий — это значит, что системы должны обеспечивать быстрый отклик и гарантированный обмен данными («не тормозить»), доступ должен быть удобным и понятным. Т. е. система должна обеспечивать *требуемый уровень обслуживания пользователя*. Важным фактором также являются экономические аспекты, связанные с затратами на вычислительные ресурсы. Поэтому в облачных системах часто решается задача поиска оптимального соотношения между требуемым уровнем обслуживания пользователем и экономическими затратами на вычислительные ресурсы. Например, если приобрести дорогой сервер, а пользователей будет мало, то вычислительные ресурсы будут простаивать, что, очевидно, экономически невыгодно. И напротив, дешевый сервер не сможет обеспечить требуемый уровень скорости вычислений и обмена данными, то есть не будет гарантировать заданный уровень качества обслуживания запросов пользователей.

Существенной задачей облачных технологий является *балансировка нагрузки* вычислительных ресурсов. Решение этих задач основано на виртуализации оборудования, что позволяет гибко изменять объемы реальных ресурсов в облачной вычислительной системе.

Исторические факты развития облачных технологий

В 1999 году Salesforce.com предоставила доступ к своему приложению через сайт, по сути, предоставив свое программное обеспечение по принципу — программное обеспечение как сервис (SaaS).

В 2002 году Amazon разработал облачный веб-сервис, позволяющий хранить, информацию и производить вычисления, а уже в 2006 году запустила сервис Elastic Compute cloud (EC2), как веб-сервис, который позволял его пользователям запускать свои собственные приложения. Сервисы Amazon EC2 и Amazon S3 стали первыми доступными сервисами облачных вычислений.

В 2009 году Google создает платформу Google Apps для веб-приложений в бизнес-секторе.

В 2009–2011 годы были определены важные постулаты об облачных вычислениях: модель частных облачных вычислений и модели обслуживания.

Национальный институт стандартов и технологий в 2011 г. сформировал определение, которое объединило и зафиксировало все возникшие к этому времени вариации и трактовки относительно облачных вычислений.

Классификация облачных систем

Сегодня облачные системы классифицируют по моделям развертывания, определяющим размещение и доступ к данным, и по моделям обслуживания, определяющим тип и объемы предоставляемых услуг.

По модели развертывания различают следующие виды:

- *Частное облако* — это инфраструктура, ограниченная группой пользователей, зачастую в пределах сотрудников одной компании. В этом случае никто кроме них (т. к. доступ может быть ограничен на физическом уровне) не может получить доступ к данным, тем самым обеспечивая безопасность данных.
- *Публичное облако* — это инфраструктура для свободного использования широким спектром пользователей. Разрешение доступа большого количества различных пользователей ставит под большие риски утечку и потерю данных.
- Общественное облако — вид инфраструктуры для группы или нескольких групп имеющие общие задачи и цели.

Публичное облако предоставляет удобный и свободный доступ к данным, но из-за этого является уязвимым. Частное облако, в свою очередь, предоставляет доступ к данным ограниченному числу пользователей, например, внутри одной компании, тем самым обеспечивая изолированность данных, а значит их безопасность. Предоставить удобный доступ и обезопасить данные позволяет объединение данных моделей в гибридное облако.

Гибридное облако — это комбинация, сочетающая две и более различные облачные инфраструктуры. Разделение облака на публичную и приватную части, позволяет обеспечить безопасность данных в приватной части и свободный доступ до оставшихся данных в публичной части, как показано на рис. 2.

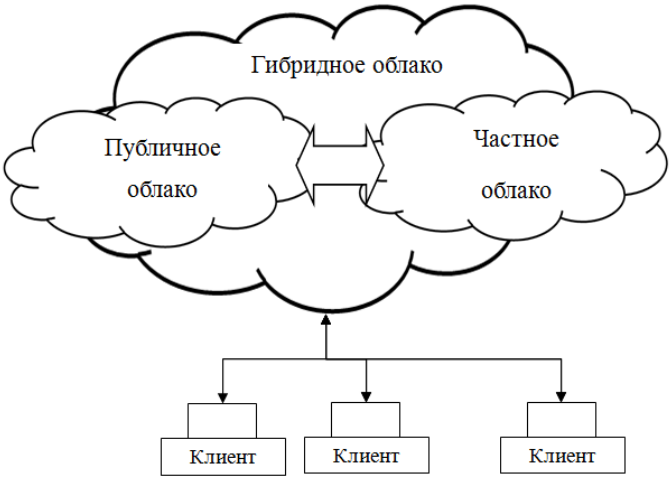

Рис. 2. Гибридная облачная инфраструктура

Однако данная модель имеет и недостатки по производительности, так как данные разделены и требуются дополнительные вычисления для их получения и объединения. Эксперименты по оценки производительности двух реляционных БД (одна — в частной части, а другая — в публичной), результаты которых показаны на рис. 3.

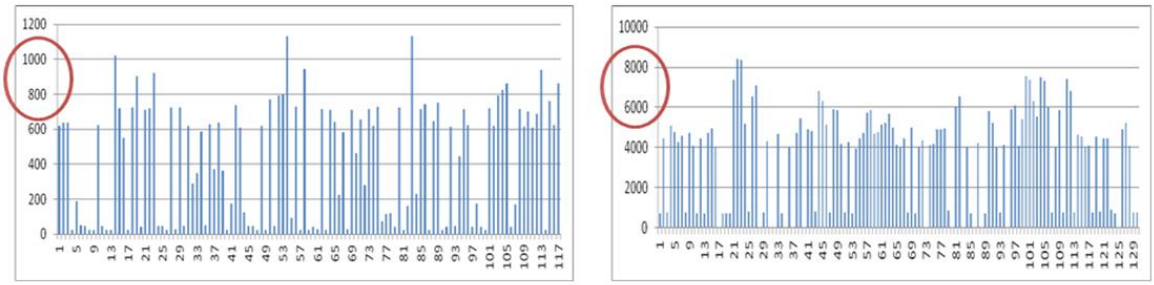

Рис. 3. Результаты профилирования БД без разделения
и с разделением на публичную и частную части

Как видно из результатов, при нагрузке СУБД, разница достигает до 7.5 раз, тем самым показывая, что хоть и выполнена защита данных, выделив эти данные на отдельный вычислительный узел и организовав до него специальное разграничение доступа, однако при этом имеет место потеря в производительности.

По модели развертывания различают следующие виды облачных систем:

– Программное обеспечение как услуга (англ. Software-as-a-Service SaaS) — это модель, в которой пользователь использует в качестве

услуги прикладное программное обеспечение, доступного с тонкого клиента. Управление виртуальной средой (вычислительными ресурсами) осуществляется провайдером, предоставляющим облачную услугу.

- *Платформа как услуга* (англ. Platform-as-a-Service PaaS) — это модель, в которой пользователю предоставляется в качестве услуги платформа, на которой он может размещать прикладное программное обеспечение. Управление виртуальной средой осуществляется провайдером, предоставляющим облачную услугу.
- *Инфраструктура как услуга* (англ. Infrastructure-as-a-Service IaaS) — это модель, в которой пользователю предоставляется вся облачная инфраструктура, распределением вычислительных ресурсов которой он имеет возможность управлять.

Распределенные вычисления (grid computing)

Коллективные, или распределенные вычисления (grid computing) представляют собой распределение большой ресурсоемкой вычислительной задачи для выполнения между множеством компьютеров, объединенных в мощный вычислительный кластер сетью в общем случае или сетью Интернет в частности. GRID-вычисления представляют собой совокупность вычислительных ресурсов в различных местах, которые работают для достижения общей цели. GRID является географически распределенной инфраструктурой, объединяющей множество вычислительных ресурсов разных типов (процессоры, долговременная и оперативная память, хранилища и базы данных, сети), доступ к которым можно получить из любой точки мира, независимо от их месторасположения.

Иэн Фостер и Карл Кесселмен в начале 90-х гг. ввели понятие GRID-вычислений, проводя аналогию с электрической сетью, к которой могли подключаться пользователи. GRID-вычисления основываются на методах кластерных вычислительных моделей, где многократные независимые группы действуют как сеть. В частности, развитие GRID-технологий позволило создать GRID-сети, в которых участники могли объединить вычислительные ресурсы для решения сложных задач.

Есть два критерия, позволяющих выделить GRID-системы среди других систем, которые обеспечивают разделяемый доступ к ресурсам:

- GRID-система координирует разрозненные ресурсы. В GRID-системе ресурсы не имеют общего центра управления, а сама GRID-система занимается координацией их использования, например, решая задачу балансировки нагрузки. Это отличает GRID-систему от простой системы управления ресурсами кластера, так как в ней осуществляется централизованное управление всеми узлами кластера, обеспечивая полный доступ к ним. В свою очередь, GRID-системы предоставляют лишь ограниченный доступ к ресурсам, в зависимости от политики административного домена (организации–владельца), в котором этот ресурс находится.
- GRID-система строится на базе стандартных и открытых протоколов, сервисов и интерфейсов. Открытость позволяет легко и быстро подключать новые ресурсы в GRID-систему, разрабатывать новые виды сервисов и так далее.

В своем современном виде GRID-технологии ориентированы на создание универсальных распределенных информационно-вычислительных сред (GRID-сред), обеспечивающих гибкий и унифицированный доступ к самому широкому спектру распределенных информационных ресурсов.

На практике граница между типами вычислений Cloud и GRID достаточно размыта, поэтому сегодня можно встретить облачные системы на базе модели распределенных вычислений, и наоборот. Однако развитие облачных вычислений перспективней распределенных систем, так как не каждый облачный сервис требует больших вычислительных мощностей с единой управляющей инфраструктурой или централизованным пунктом обработки платежей.

2. Обзор существующих облачных систем

Облачные хранилища — это модель хранилища, в котором данные хранятся на множестве распределенных в сети серверах, предоставляемых в пользование клиентам. В отличие от модели хранения данных на собственных выделенных серверах, количество или какая-либо внутренняя структура серверов клиенту не доступна, потому что данные хранятся и обрабатываются в облаке, представляющем собой один большой виртуальный сервер, а физически такие серверы могут располагаться удаленно друг от друга географически.

Среди известных хранилищ можно выделить: Яндекс.Диск, microsoft OneDrive, iCloud, Google Drive, Dropbox. Работа пользователя с данными осуществляется через web-интерфейс или через синхронизацию, таким образом, что папка в файловой системе устройства пользователя имеет одинаковое содержимое в облаке независимо от того, какое устройство используется для просмотра или редактирования данных.

Amazon's Elastic Compute Cloud (Amazon EC2)

Вычислительное облако *Amazon Elastic Compute Cloud* (*Amazon EC2*) — это веб-сервис, предоставляющий безопасные масштабируемые вычислительные ресурсы в облаке. Он помогает разработчикам, облегчая проведение крупномасштабных вычислений в облаке.

Amazon EC2 имеет веб-интерфейс, который позволяет получить доступ к вычислительным ресурсам и настроить их с минимальными трудозатратами. Пользователи получают полный контроль над ресурсами, которые они могут использовать по своему усмотрению в вычислительной среде Amazon. Сервис Amazon EC2 облегчает процесс настройки и запуска новых экземпляров и позволяет быстро масштабировать вычислительные ресурсы с учетом изменяющихся требований. Amazon EC2, за счет предоставления возможности платить только за используемые ресурсы, изменил экономическую составляющую процесса вычислений. Также разработчики получают возможность избегать распространенных ошибочных сценариев и создавать отказоустойчивые приложения.

Работа с Amazon EC2 начинается с создания образа Amazon Machine Image (AMI). Этот образ содержит приложения, библиотеки, данные и связанные параметры конфигурации, используемые в виртуальной вычислительной

среде. Amazon EC2 содержит предварительно сконфигурированные образы с шаблонами, необходимыми для быстрого начала работы. Образы загружаются в Amazon S3, и он обеспечивает безопасный, надежный и быстрый доступ к клиенту AMI. Перед использованием необходимо через веб-интерфейс настроить безопасность и сетевой доступ.

При конфигурировании выбирается, какой тип категории (Стандартный процессор или High-CPU процессор) и какая операционная система будет использоваться.

Дополнительные Веб-службы Amazon

Amazon EC2 предоставляет набор других служб, например, *Amazon Simple Storage Service (Amazon S3), Amazon SimpleDB, Amazon Simple Queue Service (Amazon SQS)* и *Amazon CloudFront.* Все они интегрированы, чтобы обеспечить полное решение для вычислений, обработки запросов и хранение между широким диапазоном приложений.

Amazon S3 обеспечивает хранение данных Amazon, чтобы пользователи могли хранить и восстанавливать любой объем данных. Разработчикам предпочтительно использовать данное хранилище, потому что оно функционирует на инфраструктуре Amazon в результате хорошо масштабируемо и надежно.

Amazon SimpleDB — предоставляет возможность СУБД в облачной инфраструктуре Amazon, позволяя выполнять запросы на структурированных данных *Amazon Simple Storage Service (Amazon S3)* в режиме реального времени. Этот сервис работает в соединении с *Amazon EC2,* чтобы предоставить пользователям возможность хранения, обработки и запросов наборов данных в пределах окружающей среды облака.

Amazon SimpleDB обеспечивает основную функциональность баз данных: поиск в реальном времени и запросы структурированных данных. *Amazon SimpleDB* не требует схемы, данные индексируются автоматически, обеспечивает простой интерфейс API для хранения и доступа к данным.

Amazon Simple Queue Service (Amazon SQS) — обеспечивает работу с очередями сообщений, обеспечивая обработку запросов большого числа пользователей. При использовании Amazon SQS разработчики могут просто переместить данные, распределенные между компонентами своих приложений, которые выполняют различные задачи, не теряя при этом сообщения.

Amazon CloudFront — осуществляет доставку контента (содержания). Разработчики получают простой способ распространять контент для конечных пользователей с низкой задержкой, высокой скоростью передачи данных.

Microsoft Azure

Платформа корпорации Майкрософт Windows Azure (ранее Azure Services Platform) — это группа «облачных» технологий, каждая из которых предоставляет определенный набор служб для разработчиков приложений. Платформа Windows Azure может быть использована как приложениями, выполняющимися в «облаке», так приложениями, работающими на локальных компьютерах, как показано на рис. 4.

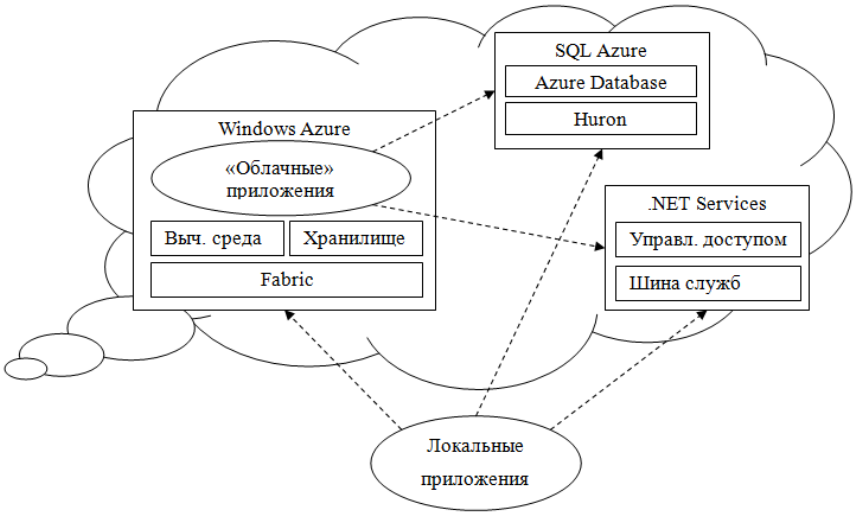

Рис. 4. Компоненты платформы Windows Azure

Платформа Windows Azure включает в себя следующие компоненты:

- Windows Azure — обеспечивает среду на базе Windows для выполнения приложений и хранения данных на серверах в центрах обработки данных корпорации Майкрософт.
- SQL Azure — обеспечивает работу службы данных в «облаке» на базе SQL Server.
- .NET Services — обеспечивает распределенную инфраструктуру для «облачных» приложений и локальных приложений.

Windows Azure функционирует на серверах, расположенных в центрах обработки данных корпорации Майкрософт, которые доступны через Интернет. Fabric Windows Azure позволяет соединить множество вычислительных мощностей в единое целое.

Windows Azure позволяет выполнять приложения, предпочтительнее разработанные на платформе .NET Framework, однако, Windows Azure также поддерживает неуправляемый код, позволяя разработчикам выполнять приложения, которые разработаны не на базе .NET Framework. Разработчики могут создавать веб-приложения с помощью таких технологий, как ASP.NET и Windows Communication Foundation (WCF), приложения, которые выполняются как независимые фоновые процессы, или приложения, сочетающие и то и другое.

Как приложения Windows Azure, так и локальные приложения могут получать доступ к службе хранилища Windows Azure, делая это одним и тем же способом: с помощью подхода RESTful. Однако Microsoft SQL Server не является базовым хранилищем данных. Фактически хранилище Windows Azure не относится к реляционным системам, и язык его запросов не SQL. Поскольку оно изначально предназначено для поддержки приложений на базе Windows Azure, то обеспечивает более простые и масштабируемые способы хранения. Следовательно, оно позволяет хранить большие двоичные объекты (binary large object — blob), обеспечивает создание очередей для взаимодействия между компонентами приложений и даже что-то вроде таблиц с простым языком запросов. (Для тех приложений Windows Azure, которым требуется обычное реляционное хранилище, платформа Windows Azure предоставляет базу данных SQL Azure, описанную далее.)

Выполнение приложений в облаке — один из самых важных аспектов облачных технологий. С помощью Windows Azure корпорация Microsoft обеспечивает как платформу для выполнения приложений, так и способ хранения данных. По мере того, как растет интерес к облачным вычислениям, ожидается создание еще большего количества приложений Windows для этой новой области.

Один из наиболее привлекательных способов использования серверов, доступных через Интернет, — это обработка данных. Цель SQL Azure — решить эту проблему, предлагая набор веб-служб для хранения самой разной ин-

формации и работы с ней. В то же время представители Microsoft заявляют, что постепенно SQL Azure будет содержать целый ряд возможностей, ориентированных на данные, включая создание отчетов, анализ данных и многое другое. Первыми компонентами SQL Azure станут база данных SQL Azure Database и средство синхронизации данных Huron.

База данных SQL Azure Database (ранее известная под названием SQL Data Services) обеспечивает систему управления базами данных (СУБД) в Интернете. Эта технология позволяет локальным и веб-приложениям хранить реляционные и другие типы данных на серверах Microsoft в центрах обработки данных Microsoft. Так же как при работе с другими веб-технологиями, компания платит только за то, что использует, увеличивая и уменьшая объем использования (и затраты) по мере возникновения необходимости в изменениях. Использование базы данных в «облаке» также меняет характер капитальных затрат: на место инвестиций в жесткие диски и ПО для СУБД приходят эксплуатационные затраты.

3. Разработка облачных служб

Службы (демоны) — это приложения, не требующие взаимодействия с пользователем, работающие в фоновом режиме и наиболее эффективно обрабатываются процессором. В среде windows данные приложения принято называть службами, в среде Linux — демонами.

Также от консольных приложений их отличает наличие функций: запуска, остановки. Помимо этих операций могут присутствовать и другие, такие как пауза или статус.

Создание служб с помощью библиотеки Qt

Для создания служб с помощью библиотеки Qt создан отдельный проект QtService. Для его использования необходимо создать класс, наследующийся от QtService<T>, и переопределить методы void start() и void stop(), выполняющие действия запуска и остановки служб.

```
class ChatListenService : public QtService<QCoreApplication>
{
public:
 ChatListenService(int argc, ///< Количество аргументов
  char **argv ///< Параметры программы
  );
 ~ChatListenService();
 void start(); void stop();
};
int main(int argc, char *argv[])
{
 ChatListenService service(argc, argv);
 return service.exec();
}
```

После этого вашему приложению службы будут доступны следующие функции, при передачи соответствующих параметров.

-i — install — установка службы;

-u — uninstall — дезинсталляция службы;

-e — exec — запуск службы как стандартное приложение;

-s — start — запустить службу;

-t — terminate — остановить службу;

-p — pause — поставить службу на паузу;

-r — resume — возобновить выполнение службы после паузы.

Создание служб с помощью .Net Framework

В среде Visual Studio имеется шаблон проекта Windows Service, который создает следующий класс, являющийся наследником ServiceBase.

```
namespace ExampleSrv
{
 public partial class MyService : ServiceBase
 {
 public MyService()
 {
 InitializeComponent();
 }

 protected override void OnStart(string[] args)
 {
 }

 protected override void OnStop()
 {
 }
 }
}
```

OnStart и **OnStop** — события для реализации запуска и остановки службы.

Чтобы установить сервис, нужно вызвать утилиту установки и передать параметром путь к своему сервису, следующей командой

```
C:\Windows\Microsoft.NET\Framework\v4.0.30319\installutil.exe
D:\...\ExampleSrv\bin\Debug\ExampleSrv.exe
```

Управление службами в ОС семейства Linux

Чтобы приложение запускалось как служба, необходим скрипт, который обычно располагается в директории /etc/init.d. Далее пользователь командами *service название_сервиса start*, *service название_сервиса stop* и пр. может управлять сервисом (название_сервиса — это название скрипта в /etc/init.d).

Журналирование служб

Для журналирования действий службы используется системное логирование.

В ОС семейства Windows для просмотра журнала используется «журнал событий» (Панель управления — администрирование — журнал событий). Для добавления туда записей в приложении используется сервис **EventLog, добавление в который выполняется методом** AddLog()

```
 protected override void OnStart(string[] args)
 {
 AddLog("start");
 }
```

В ОС семейства Linux для журналирования используется служба syslog, Механизма Syslog заключается в формировании источниками простых текстовых сообщений о происходящих в них событиях и передачи их на обработку серверу Syslog, который использует сетевые протоколы UDP или TCP (по

умолчанию порт 514). За сервер Syslog отвечают службы «syslogd», «syslog daemon», «rsyslog», в зависимости от ОС.

Основным конфигурационным файлом данных служб является /etc/syslog.conf (/etc/rsyslog.conf), который структурирован по следующему принципу:

- Модули (Modules),
- Конфигурационные директивы (Configuration Directives),
- Шаблоны (template),
- Правила сортировки (Rule line).

С помощью данного конфигурационного файла настраиваются также места сохранения журналов. Это могут быть как локальные файлы, которые обычно храниться в директории /var/log, так и удаленные узлы. Для добавления сообщения необходимо подключить #include <syslog.h> и вызвать метод void syslog(int, const char *, ...);, где первый параметр — приоритет, а второй — само сообщение.

Каждое сообщение содержит источник и уровень серьезности. Источник сообщения кодируется числом от 0 до 23, а уровень серьезности кодируется числом от 0 до 7.

Протоколы сервисов

Так как к облачным сервисам предоставляется доступ через Интернет, то основным протоколом взаимодействия является HTTP.

Изначально HTTP использовался для передачи гипертекстовых документов в формате HTML, но сегодня он используется для передачи разнородных данных и используется как базовый транспортный протокол для других протоколов прикладного уровня, таких как SOAP, XML-RPC, WebDAV.

HTTP сообщение состоит из трех частей:

1. Стартовая строка. Она состоит из метода (get,post,head,put) и URL.
2. Заголовок, который соответствует стандарту RFC 822. Основными параметрами заголовка являются Content-Type, Content-Language, Content-Length.
3. Тело сообщения

Для взаимодействия сервисов широко используется сообщения в формате XML, особенно протоколы WSDL (*Web Services Description Language* — язык

описания веб-служб) и SOAP (*Simple Object Access Protocol* — простой протокол доступа к объектам). Если служба представляет собой простой интерфейс, который абстрагирует ее всю сложность, то пользователи могут получить доступ к независимым услугам без знания реализации платформы сервиса.

Каждый документ WSDL состоит из:

1. Определения типов данных (вид отправляемых и получаемых сервисом сообщений);
2. Элементов данных (сообщения);
3. Абстрактные операции (список операций);
4. Связывание сервисов (способ доставки сообщений).

Пример WSDL

```
<message name="getServerRequest">
 <part name="term" type="xs:string"/>
</message>

<message name="getServerResponse">
 <part name="value" type="xs:string"/>
</message>

<portType name="server_communication">
 <operation name="getServerData">
 <input message="getServerRequest"/>
 <output message="getServerResponse"/>
 </operation>
</portType>
```

SOAP представляет собой протокол обмена структурированными сообщениями, на котором базируются стандарты web служб. SOAP — это набор правил, формализующих и управляющих форматом и правилами обработки информации, которой обмениваются отправитель SOAP и получатель SOAP. Узлы SOAP — это физические / логические машины с процессорами, которые используются для передачи / пересылки, приема и обработки сообщений SOAP. Каждый узел принимает на себя определенную роль. Роль узла определяет действие, которое узел выполняет над полученным сообщением. Например, роль «нет» означает, что ни один узел не будет каким-либо образом обрабатывать заголовок SOAP и просто передавать сообщение по своему пути. Сообщение SOAP должно работать в сочетании с другими протоколами, передаваемыми по сети. Например, сообщение SOAP может использовать TCP в качестве протокола нижнего уровня для передачи сообщений. SOAP предоставляет только структуру обмена сообщениями, однако его можно расширить, добавив такие

функции, как надежность, безопасность и прочие. SOAP-сообщение представляет собой информацию, которой обмениваются 2 SOAP-узла, и состоит из:

- Конверта SOAP. По своему названию это включающий элемент сообщения XML, идентифицирующий его как сообщение SOAP.
- Блока заголовка SOAP. Заголовок SOAP может содержать более одного из этих блоков, каждый из которых представляет собой дискретный вычислительный блок в заголовке. Как правило, информация о роли SOAP используется для назначения узлов на пути. Считается, что блок заголовка нацелен на узел SOAP, если роль SOAP для блока заголовка — это имя роли, в которой работает узел SOAP.
- Заголовка SOAP. Набор из одного или нескольких блоков заголовков, предназначенных для каждого получателя SOAP.
- Тела SOAP. Содержит тело сообщения, предназначенного для получателя SOAP. Интерпретация и обработка тела SOAP определяется блоками заголовка.
- Ошибки SOAP. Если узлу SOAP не удается обработать сообщение SOAP, он добавляет информацию об ошибке в элемент ошибки SOAP. Этот элемент содержится в теле SOAP как дочерний элемент.

Пример SOAP сообщения:

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
xmlns:m="http://www.example.org">
 <soap:Header>
 </soap:Header>
 <soap:Body>
 <m:GetTaxes>
 <m:Tax>20</m:Tax >
 </m:GetTaxes>
 </soap:Body>
</soap:Envelope>
```

4. Системы управления облачной инфраструктурой

Для обеспечения требуемого уровня обслуживания клиентов облачная система должна быть масштабируемой и гибкой. Поэтому в облачных системах нашли широкое применение системы виртуализации оборудования, позволяющие выделять виртуальные вычислительные ресурсы для решения задач от реального оборудования. Используя виртуализацию, можно консолидировать ресурсы, такие как оперативная память, вычислительные ресурсы процессоров, объемы дисковых пространств и пропускную способность сети. *Виртуализация* — это создание гибкой замены реальных ресурсов с теми же функциями и внешними интерфейсами, что и у физических прототипов, но с разными атрибутами, такими как размер, производительность и стоимость.

Виртуальные системы чаще всего осуществляются с помощью гипервизоров. *Гипервизор* — это программное или микропрограммное обеспечение, позволяющее виртуализировать системные ресурсы.

В реализации технологий виртуальных машин выделяются три основных подхода (рис. 5):

- Гипервизор первого типа (автономный, тонкий, исполняемый на «голом железе» — Type 1, native, bare-metal) — программа, исполняемая непосредственно на аппаратном уровне компьютера и выполняющая функции эмуляции физического аппаратного обеспечения и управления аппаратными средствами и гостевыми ОС. То есть такой гипервизор сам по себе в некотором роде является минимальной операционной системой. ·
- Гипервизор второго типа (хостовый, монитор виртуальных машин — hosted, Type-2, V) — специальный дополнительный программный слой, расположенный поверх основной хостовой ОС, который в основном выполняет функции управления гостевыми ОС, а эмуляцию и управление аппаратурой берет на себя хостовая ОС. ·
- Гипервизор гибридный (Hybrid, Type-1+) — объединенный вариант первых двух, в котором функции управления аппаратными средствами выполняются тонким гипервизором и специальной депривилегированной сервисной ОС, работающей под управлением тонкого гипервизора. Обычно гипервизор управляет напрямую

процессором и памятью компьютера, а через сервисную ОС гостевые ОС работают с остальными аппаратными компонентами.

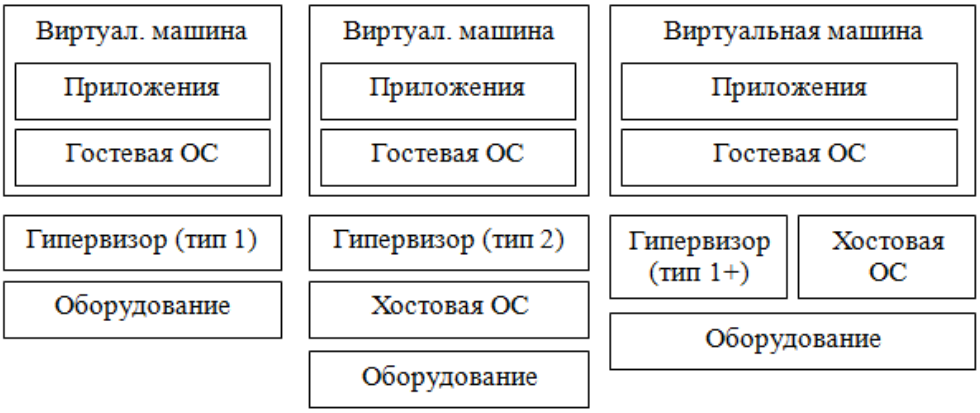

Рис. 5. Виды гипервизоров

На сегодняшний день существует ряд проектов, реализующих гипервизоры, ориентированные под различные аппаратные и программные платформы:

- **Xen:** монитор виртуальных машин для процессорных архитектур IA-32, x86-64, Itanium и ARM, Xen позволяет выполнять несколько гостевых операционных систем на одном и том же оборудовании одновременно. Xen-системы имеют структуру, в которой гипервизор Xen занимает самый низкий и привилегированный уровень.
- **KVM:** инфраструктура виртуализации для ядра Linux, KVM поддерживает платформенно-зависимую виртуализацию на процессорах с аппаратными расширениями для виртуализации. Первоначально он поддерживал процессоры x86, но в настоящее время к ним добавился широкий спектр процессоров и гостевых операционных систем, в том числе множество вариаций Linux, BSD, Solaris, Windows®, Haiku, ReactOS и AROS Research Operating System (есть даже модифицированная версия QEMU, способная использовать KVM для работы с Mac OS X).
- **PowerVM:** принадлежность серверов на базе IBM POWER5, POWER6 и POWER7, этот гипервизор поддерживается операционными системами IBM i, AIX® и Linux®; PowerVM поддерживается в среде IBM SmartCloud Enterprise.

- **VMware ESXi** встроенный гипервизор VMware ESX работает непосредственно на аппаратуре серверов, не требуя дополнительной операционной системы. Он поддерживается в среде IBM SmartCloud Enterprise.
- **Hyper-V** гипервизор, разработанный компании Microsoft, который относится ко второму типу гипервизоров, которые могут работать только под управлением ОС Microsoft Server 2012 и Microsoft 8.1 Professional и выше.

Как видно, гипервизоры различаются между собой и ориентированы под различные платформы. Поэтому необходимо выбирать гипервизор под решаемые задачи, а также руководствуясь критериями: производительностью гипервизора, возможностями управления ресурсами и стоимостью.

Управление ресурсами виртуальных систем

Гипервизоры имитируют работу практически всего оборудования, создавая их виртуальные копии. Среди них, влияющих на производительность, можно выделить только процессор, оперативную память, жесткий диск и сетевой адаптер. Именно управляя этим оборудованием можно изменять производительность виртуальной машины.

Хотя теоретически виртуальные ресурсы бесконечны, однако эта бесконечность довольно условная, потому что физические ресурсы, на которых работают гипервизоры, ограничены. Поэтому важной задачей является оптимальное распределение физических вычислительных ресурсов между виртуальными машинами.

Для каждой виртуальной машины выделяются требуемое количество вычислительных ресурсов: объем оперативной памяти, количество и частота процессоров. Это так называемые конфигурационные (configurated) настройки виртуальной машины. При этом виртуальные машины не всегда занимают все выделенные для них вычислительные ресурсы, иногда они простаивают, когда к ним нет запросов, поэтому в это время данными ресурсами могут пользоваться другие виртуальные машины. Для настройки этого в гипервизоре VMWare ESXi предусмотрен ряд параметров как Limit, Reservation и Shares для пулов ресурсов (Resource Pool) в пределах кластеров VMware DRS и отельных хостов ESX. Именно этими тремя параметрами определяется потребление виртуаль-

ными машинами оперативной памяти и процессорных ресурсов хоста VMware ESX [3].

Limit — определяет ограничение потребления физических ресурсов виртуальной машиной или пулом. Виртуальная машина ни при каких условиях не сможет использовать больше ресурсов чем установлено в данном параметре. Таким образом, если данный параметр задан в пределах физических ресурсов, то никаких конфликтов между виртуальными машинами не возникает, но если данный параметр пересекается между виртуальными машинами, то возникают конфликты, как показано на рис. 6 и 7.

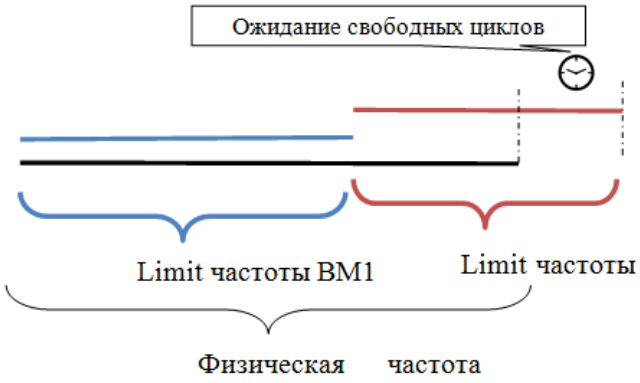

Рис. 6. Ситуация нехватки вычислительных ресурсов процессора

Если оба процессора в совокупности не получают количество циклов процессора, выделенных параметром Limit, то они начинают просто ожидать их, когда процессор освободится. В этом случае возникают задержки.

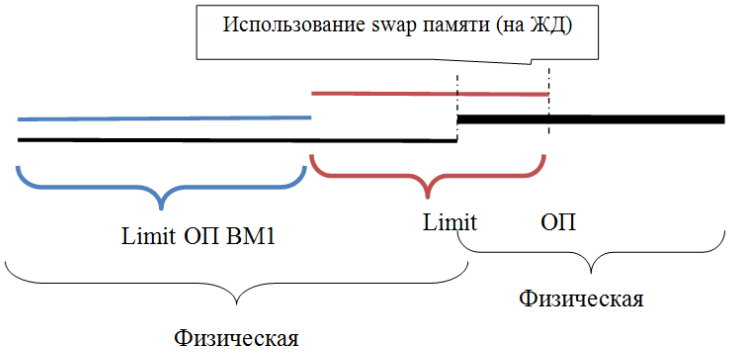

Рис. 7. Ситуация нехватки оперативной памяти

Если потребление оперативной памяти доходит до параметра Limit, то она уходит в swap, которая работает медленнее, тем самым вызывая задержки.

Shares определяет приоретизацию потребления виртуальных машин между собой в пределах хоста ESX или пула ресурсов. Это свойство имеет значение, когда возникает нехватка вычислительных ресурсов, как в случаях, показанных на рис. 1 и 2. Тремя стандартными настройками High, Medium и Low обозначаются соотношение приоритетов, определяющие какая из виртуальных машин будет ожидать вычислительных ресурсов (та, у которой приоритет ниже), и у какой не будет возникать никаких проблем с вычислительными ресурсами (у которой приоритет выше).

Параметр Reservation для виртуальной машины или пула ресурсов определяет, сколько физической памяти или ресурсов процессора будет гарантировано виртуальной машине при работе. То есть, если виртуальная машина еще не достигла параметра Reservation, то неиспользуемые ресурсы могут быть выданы другим виртуальным машинам (в Shares), иначе, если уже достигла своего уровня, то у нее уже не будут отниматься ее вычислительные ресурсы. Описанные параметры распределения ресурсов между виртуальными машинами изображены на рис. 8.

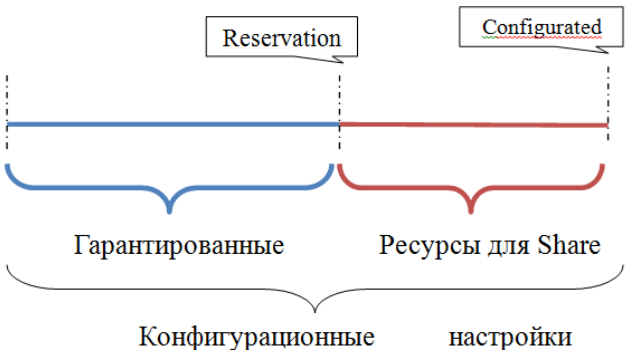

Рис. 8. Параметры распределения ресурсов виртуальных машин

Оптимально настроив и управляя этими параметрами распределения вычислительных ресурсов между виртуальными машинами, можно по максимуму использовать физические ресурсы серверов, уменьшив их простои без задач.

VMware ESXi с настройками по умолчанию самостоятельно распределяет нагрузку виртуальных машин, но также позволяют сделать это вручную разра-

ботчикам или администраторам. Для управления гипервизором VMWare ESXi предоставляется ряд программного обеспечения администрирования: VMware vSphere, VMware vSphere Web Client, VMware vSphere PowerCLI.

Разработка программных средств управления гипервизором

Для автоматизации управления и разработки собственных методов балансировки нагрузки можно использовать функционал библиотек VMware или консоль PowerCLI.

Наиболее простой способ для автоматизации управления — это разработка скриптов с помощью **PowerCLI**, которые можно вызывать отдельно или из приложения. PowerCLI — это расширение для Windows Powershell, которое добавляет более 400 новых команд для управления виртуальной инфраструктурой, в том числе и Cloud. Результаты выполнения команд PowerCLI возвращают результаты в формате объектов .NET, что делает удобной разработку средств на языке C# с библиотекой .NET Framework.

Для обеспечения управления системой виртуализации VMware ESXi достаточно следующего набора команд (подробное описание всех команд PowerCLI можно найти в справке):

- Connect-VIServer — выполняет подключение к системе виртуализации, после которого можно выполнять все остальные команды;
- Get-VMHost — получает информацию о хосте, включая его вычислительные характеристики;
- Get-VM — получает список виртуальных машин и их вычислительные характеристики;
- Get-Stat — получает журнал загрузки вычислительных ресурсов;
- Set-VMResourceConfiguration — изменяет параметры пределов и приоритетов распределения вычислительных ресурсов хоста.

Помимо консоли VMWare предоставляет ряд библиотек и средств разработки под различные языки программирования, что позволяет разрабатывать практически любые приложения для управления гипервизором и отдельно гостевыми операционными системами.

5. Алгоритмы и методы балансировки нагрузки

Мониторинг и распределение загрузки серверов в облачных системах — это актуальная сфера исследований, только начавшая набирать популярность. Многие открытые облачные системы (например, OpenStack) сейчас используют простейшие планировщики нагрузки на физические сервера.

Вопрос о балансировке нагрузки нужно решать на ранних стадиях развития проекта, так как при «падениях» сервера у клиентов отпадает желания пользоваться продуктом, что чревато материальными последствиями. Если в начале создания проекта можно наращивать мощности с помощью новых серверов или как-то оптимизировать сам код, то настает момент, когда и эти меры оказываются недостаточными.

Распределенные ИАС включают, как правило — виртуализацию, сети, программное обеспечение и веб-сервисы, и состоит из нескольких элементов такие как:

- клиент,
- центр обработки данных,
- распределенные серверы.

При этом это все включает:

- отказоустойчивость,
- высокую доступность,
- масштабируемость,
- гибкость,
- уменьшенные накладных расходов для пользователя,
- уменьшения стоимости владения, на сопровождения и т. д.

Из этого следует, что нагрузка на сервера в системах колоссальная и встает вопрос — как эффективно распределять пользователей по ним. Нагрузка может быть, на CPU, на память, на сети и т. д.

Чтобы сбалансировать запросы ресурсов, важно распознать несколько главных целей методов выравнивания нагрузки:

- Рентабельность: основная цель состоит в том, чтобы достигнуть полного улучшения системы производительность по разумной стоимости.

- Масштабируемость и гибкость: распределенная система в котором реализован алгоритм может измениться в размере или топологии. Таким образом, алгоритм должен сохранять работоспособность при увеличении нагрузки.
- Сокращение времени отклика и времени выполнения запроса: минимизировать время ответа на запрос и обеспечить минимальное время между началом обработки и его концом.
- Обеспечение эффективности: желательно что бы все серверы были загружены одинаково и не было такого что один работает на полную мощность, а другие простаивают.
- Понимание алгоритма: нужно четко знать минусы и плюсы алгоритма и понимать, как он работает для эффективного его использования.

Балансировка нагрузки — это процесс распределения нагрузки среди различных узлов распределенной системы, для улучшения использования ресурсов и времени отклика их, также этот процесс предотвращает нагрузку на одну систему и простой в другой системе, качественно распределяя на них нагрузку. Балансировка нагрузки гарантирует, что весь процессор в системе или каждый узел в сети делает приблизительно равный объем работы в любой момент из времени.

Цели балансировки нагрузки:

- существенно улучшить работу системы;
- иметь резервный план в случае, если система будет в критическом состоянии;
- поддержать стабильность системы;
- приспосабливать будущие модификации в системе.

Можно выделить несколько уровней балансировки.

Кластеризация — этот метод позволяет управлять несколькими независимыми серверами как одной системой, тем самым упрощает управление системой, позволяет добиться большой масштабируемостью и отказоустойчивостью (рис. 9). В этом методе балансировки система распределяет поток IP-данных между узлами, чаще всего несколько аппаратных платформ разделяют единый IP-адрес.

Рис. 9. Балансировка нагрузки методом кластеризации.

В данном методе есть несколько вариантов распределения нагрузки, например, либо администратор может определить, какую нагрузку должен иметь тот или иной узел, либо по умолчанию все узлы будут иметь одинаковую нагрузку. Все запросы клиентов статически распределяются между узлами, тем самым распределяя нагрузку при включении и удалении узла кластера. Если изменяются параметры нагрузки на аппаратные платформы серверов, то распределенная нагрузка не меняется. В всевозможных веб-сервисах, с большим количеством клиентов и множественными короткими запросами от них такой механизм распределения нагрузки очень эффективен и обеспечивает быструю реакцию на изменение состава узлов кластера.

Все сервера кластера, которые управляются сервисом балансировки нагрузки с течением какого-то времени рассылают широковещательные или групповые сообщения всем остальным узлам и принимают тоже самое от других частей кластера. Если вдруг сервер «упал», то оставшиеся узлы распределяют нагрузку и тем самым гарантируется бесперебойное обслуживания и сервисы остаются доступны. В основном при таких сбоях, например, при балансировке нагрузки в веб-сервисах, клиентская часть (браузер) автоматически повторяет подключения, и сам сбой отражается как небольшая задержка отклика на запрос.

Примеры такой работы:

- Oracle Solaris Cluster;
- Linux Virtual Server;
- Open HA Cluster и др.

Недостатки данного метода:

- расположение серверных платформ должны находиться в одном сегменте сети;
- на каждом узле должно быть специально программное обеспечение что бы кластер функционировал (программное обеспечение контролирует состояния узлов и выполняет оперативный контроль за поступающими запросами);
- невозможно обеспечить мульти платформенный кластер.

Метод NAT (Network Address Translation)

В данной модели, когда пользователи обращаются к облаку, запрос проходит через выделенное устройство, которое, на основании заранее заданных статических правил, распределяет всю нагрузку, либо ориентируется на время обработки запроса серверами, производит изменения параметров установленных сессий с изменением отдельных полей в заголовках пересылаемых пакетов (рис. 10).

Рис. 10. Балансировка нагрузки через одно устройство

Таким образом, от клиента поступает пакет на балансировщик, тот в свою очередь заменяет в пакете адрес получателя IP-адресом сервера (которого

назначит), заменяет IP-адрес отправителя (клиента) на свой и тем самым скрывает от клиента адрес веб-сервера. Это позволяет убрать проблему использования серверов в одном сегменте сети, достаточно, чтобы они могли работать с использованием протокола IP.

Этот метод применяется только если нет необходимости создания сессии между клиентом и сервером на длительное время, поскольку каждому из пользователей нужно дать оригинальный IP-адрес на внутреннем интерфейсе системы балансировки на всю сессию.

Метод NAPT (Network Address Port Translation).

В данном подходе система балансировки меняет не только IP-адрес в заголовке, как в методе выше, но и номер портов транспортного уровня. Данный метод убирает проблему потребности в присвоении новым клиентам уникального IP-адреса, данный метод балансировки нагрузки выделяет уникальный порт транспортного уровня. Примеры реализации данного метода:

- универсальные маршрутизаторы Cisco 1900;
- XTM Firewall Appliance компании WatchGuard;
- межсетевой экран D-Link DFL-860 и другие.

Как они работают:

- поступает запрос на внешний адрес;
- балансировщик изменяет заголовок;
- перенаправляют запрос на один из внутренних адресов (сервер к которому адрес присвоен).

Перед тем как установлено соединение, на устройстве хранится соответствие:

- локального порта;
- локального IP-адреса;
- внешнего IP-адреса;
- внешнего порта транспортного уровня.

При этом устройство не проводит анализ времени отклика, возможности сервера, доступности серверов и др.

Данный метод использует балансировку, которая за основу берет количество открытых сессий TCP на каждом из серверов или алгоритм последова-

тельного перебора. Обычно данный функционал служит дополнительной функцией к основным в маршрутизаторах или межсетевых экранах.

Проксирование.

Для балансировки нагрузки с помощью изменения заголовков на уровне выше транспортного используются прокси-сервера (рис. 11).

Как правило, когда клиент делает запрос, запрос отправляется на веб-сервер. Веб-сервер обрабатывает запрос и отправляет ответное сообщение запрашивающему клиенту. Для того чтобы улучшить производительность, создается прокси-сервер между клиентом и веб-сервером. Теперь, как сообщение запроса, отправленного клиентом, так и ответного сообщения, поставленного на веб-сервере, проходят через прокси-сервер. Другими словами, клиент запрашивает объекты через прокси-сервер (прокси-сервер при этом меняет поля заголовков уровня приложения и транспортного). Прокси-сервер пересылает запрос клиента к веб-серверу. Веб-сервер генерирует ответное сообщение и доставляет его на прокси-сервер, который, в свою очередь кэширует ответ от сервера и отдает сохраненную информацию в ответ на все остальные запросы клиента.

Рис. 11. Балансировка нагрузки с помощью прокси-сервера.

Основные функциональные возможности кэширования происходит следующим образом:

1. Когда прокси-сервер получает запрос, он проверяет, кэшируется ли запрашиваемый объект. Если ответ положительный, то объект возвращается из кэша, без соединения с сервером.

2. Если объект не кэшируется, прокси-сервер извлекает объект с сервера, возвращает его клиенту, и кэширует копию для будущих запросов. На практике прокси-сервер должен проверить, что кэшированные ответы являются действительными и что ответ являются правильным на запрос клиента (рис. 12).

Также прокси-сервер может распределять нагрузку между веб серверами, которые работают с разными областями приложения, но при этом ему нужно преобразовать имена ресурсов, которые доступны снаружи, в имена внутренней сети. Те прокси-сервера, которые оставляют запросы и ответы в первоначальном виде, обычно называют туннелирующим прокси или шлюзом.

Рис. 12. Работа прокси-сервера

В параметрах этого метода можно задать:

- допустимое время отклика от сервера;
- допустимое количество неуспешных запросов;
- веса, сформированные серверами.

Более продвинутыми функциями программного решения является HAProxy, в ее функции еще входит балансировка приложений. Это решение используется на платформах Solaris, Linux, Free BSD.

Аппаратно-программные решения, которые играют роль балансировки с использованием изменения заголовков сетевого, транспортного и уровня прилодений:

- xBalancer компании Net Optics;
- Brocade Server Iron ADX;
- Cisco ACE 4700 Application Control Engine и ряд других продуктов.

Эти системы анализируют количество запросов, которые были им отправлены, смотрят за временем ответа сервера, за нагрузкой на которые они распределяют. Эти данные анализируются, и нагрузка на серверы балансирует-

ся. Также возможно сделать защищенные соединения SSL на эти устройствах и возможность сжатия данных, что, в конечном счете, снижает нагрузку на серверные приложения. Данный метод обычно используется в облачных системах, которые предоставляют сервисы SaaS, но совместно с балансировкой нагрузки между ЦОД (данный метод используют для балансировки нагрузки внутри ЦОД). Brocade Server Iron ADX используется в облачных системах Yahoo, Apple, Google, в корпоративных продуктах HP, IBM, Microsoft и других.

Основные недостатки данной модели балансировки нагрузки:

- если использовать системы балансировки на базе универсальных аппаратных платформ, то это ограничения возможности по функциональности и масштабировании производительности;
- так как в данной модели единая точка пропуска трафика, то и точка отказа — одна;
- высокая цена на аппаратно-программные комплексы, которые решают проблему балансировки нагрузки,
- узкая сфера применений при больших требованиях к техническим характеристикам по надежности и производительности.

Балансировка — запросил, узнал, забыл

Этот метод состоит из 2-х этапов.

1. Пользователь отправляет запрос на устройство балансировки.
2. Балансировщик получает запрос, указывает с каким сервером приложений продолжать работу пользователю, либо возвращает IP-адрес этого сервера, либо перенаправляя его средствами HTTP Redirect.

Дальнейшая работа пользователя с системой происходит без взаимодействия с балансировщиком.

Как пример можно привести круговой DNS. В DNS-серверах есть возможность внесения нескольких записей одинаковых доменных имен, но указания при этом разных IP-адресов, так как в данных серверах (в их базах) хранятся записи которые определяют соответствия имени хоста и его IP-адреса. Тем самым для одного приложения можно указать множество серверных платформ и при обращении первого клиента он будет отправлен на первый компьютер, второй пользователь — на второй и так до конца, когда все записи кончатся, он

начнет все сначала. Данный метод сейчас присутствует как опция в большинстве современных DNS-серверов.

Балансировкой средствами HTTP Redirect пользуются модули Mod_backhand для серверов Apache, а также Citrix NetScaler, оптимизирующий доставку веб-приложений.

Но этот метод используют реже. Данный подход позволяет менять в настройках сервера группу IP-адресов серверов приложений и для каждого адреса вес, который определяет долю запросов, который отвечает за балансировку нагрузки.

В данном методе есть очень ощутимые плюсы:

- Данная система очень простая, из-за того, что не нужно что-то постоянно покупать или настраивать, или устанавливать, достаточно только один раз внести в базу несколько записей и все, без всяких специальных балансировщиков или программного обеспечения.
- Масштабирование тоже на высоком уровне, из-за того, что весь трафик пользователя не проходит через балансировщик, а проходит только 1 запрос 1 раз
- Если посмотреть с точки зрения администрирования, то все предсказуемо, прозрачно и просто.

При использовании резервирования отказоустойчивость данных систем можно держать на нормальном уровне, но все же если вдруг откажут один и более серверов на которых были копии приложений, то все запросы пропадут. Так как в данной модели отсутствует обратная связь, то и исключить сервер приложений из списка невозможно, тем самым пере направив клиента.

Все методы, которые не используют метод пропуска трафика через одно устройство, всего лишь делят равномерно или пропорционально запросы между всеми платформами, которые входят в состав распределенного сервера. Проблема в том, что если вдруг все платформы имеют различные вычислительные мощности, различные установленные приложения, то такое статическое распределение запросов может привести к перегрузке одних и простаивании других платформ. Но даже в случае если все вычислительные ресурсы одинаковы, это не значит, что каждая машина будет выполнять запрос одинаково, ведь од-

ни пользователи могут запросить одну информацию, а другие запустить множество скриптов которые потребуют много системных ресурсов. При этом банальное зависание системы (увеличение времени отклика от системы) пользователем воспринимается отрицательно, и он старается постоянными нажатиями кнопки загрузить на своем браузере восстановление работы с системой, тем самым сразу перегружает машину.

6. Обеспечение качества обслуживания

В облачной среде на процесс распределения ресурсов могут влиять следующие факторы.

Загруженность системы неравномерная. Так как одним из плюсов облачных систем является то, что не нужно сопровождать систему или среда предполагает минимальное вмешательство администратора, то при распределении ресурсов нужно учитывать много факторов, такие как отклик дисковых ресурсов, сетевые задержки, процессорное время, оперативную память и многое другое.

Отсутствие ручного назначения ресурсов пользователем. В облачных системах нет методов, которые бы позволяли пользователю делать привязки на конкретные сервера или системы хранения данных, так как это противоречит идее облачных систем.

Потребляемые ресурсы не соответствуют запрошенным. При использовании облачных систем всегда запрашивают вычислительные ресурсы с запасом, зачастую с большим. И в ходе эксплуатации системы получается так что расходуется небольшая часть ресурсов, а остальное простаивает.

Отсутствия сведений о реальных потреблениях приложения в ресурсах. При использовании тех или иных приложений у администратора облачной среды нет информации какие приложения и с какими потребностями работают.

Сочетания ресурсов и классы оборудования различные. Так как идея при создании концепции облачных технологий состояла в экономии средств при создании и масштабируемости системы, из этого следует, что и классы оборудования в облачной системе могут быть разные, обладающие разными характеристиками.

Приложения потребляют разные ресурсы. Допустим на двух разных экземплярах могут находиться два приложения, одно приложение потребляющее 10% ресурсов, а второе 80%, тем самым приводя к неэффективному использованию ресурсов облачной системы.

При решении этих факторов в облачных вычислительных средах появилось несколько подходов к первоначальному выделению и последующему распределению ресурсов, из них можно выделить три типа:

– планирование ресурсов диспетчером облачной вычислительной

среды (прохождения трафика через одно устройства);

- планирование ресурсов средой визуализации (кластеризация);
- ручное назначения ресурсов (балансировка — запросил, узнал, забыл).

Ниже приведена таблица в которой укажем сравнительные характеристики методов балансировки (табл. 1).

Таблица 1. Сравнительные характеристики методов балансировки.

	Кластер	**Прохождения трафика через одно устройства**	**Без пропуска трафика через одно устройство**
Масштабируемость	Удовлетворительная	Плохая	Отличная
Оптимальность распределения нагрузки	Отличная	Удовлетворительная	Плохая
Надежность распределенного приложения	Удовлетворительная	Удовлетворительная	Плохая
Сложность администрирования	Плохая	Удовлетворительная	Отличная
Суммарная стоимость решения балансировки нагрузки	Средняя	Высокая	Низкая

В табл. 1 видно, что универсальной системы балансировки нагрузки нету, и для каждой из моделей обслуживания нужен свой метод или применение группы методов.

Кроме того, ни один из рассмотренных методов распределения вычислительных ресурсов не учитывает такие важные составляющие, как дисковая подсистема и сеть — фактически, приложения с повышенными требованиями к производительности диска (базы данных, аналитические системы) и повышенные требования к производительности сети (видеоконференции, интернет-шлюзы) выпадают из логики распределения ресурсов. Для контроля и управления облачной инфраструктурой нужен качественный мониторинг системы, который бы не мешал работе всей системы, производил ее полный мониторинг. Облачные системы контроля облаком должны быть усовершенствованы и иметь возможность полного мониторинга системы, чтобы удовлетворить по-

требности всех, кто работает с облаком, начиная от пользователя и заканчивая разработчиками.

Можно выделить несколько проблем при мониторинге системы:

- разнородность систем мониторинга (встроенный, внутренний, внешний и промежуточный);
- нет единого пула информации со всех систем мониторинга;
- системы мониторинга имеют очень сложные архитектуры;
- часто система мониторинга может являться причиной возникновения проблемы (например, при некачественном мониторинге может производиться миграция виртуальных машин слишком часто, тормозя работу системы).

7. Облачная виртуальная среда разработки программ

Создание рабочего окружения является основной моделью проектируемой системы, позволяющей проводить ее разработку, экспериментальное тестирование и отладку.

Для организации рабочего процесса и совместной разработки целесообразным является использование системы управления проектами и системы отслеживания ошибок.

Система управления проектами предоставляет из себя набор инструментов и методов, позволяющих управлять задачами в организации и помогать повысить эффективность их выполнения. Системы управления проектами позволяют решать ряд задач: повышение эффективности работы, улучшение качества управления, организация планирования, коммуникация между командой, контролирование хода выполнения задач и многое другое. Система отслеживания позволяет разработчикам ПО учитывать и контролировать ошибки и следить за процессом их устранения.

Большинство систем ориентированы, в первую очередь, на простое управление проектами. В текущем случае необходимо было решение, подходящее прежде всего для разработки ПО и включающее в себя систему управления проектами и отслеживание ошибок.

В качестве системы управления проектами, задачами и ошибками широко используется веб-приложение Redmine с открытым исходным кодом. Redmine не является веб-сервисом, что позволяет установить его на свой собственный сервер без каких-либо ограничений и сконфигурировать под необходимые задачи.

Аналогичные системы, такие как JetBrainsYouTrack, AtlassianJira и Asana, являются коммерческими, и предоставляются по подписке, где платить приходится за дополнительных пользователей, что достаточно дорого. В связи с этим применение подобных решений не является целесообразным в рамках ограниченного бюджета и большого количества сотрудников, задействованных в проекте. Среди остальных решений с открытым исходным кодом подходящих решений не было найдено, кроме того, Redmine является одним из самых попу-

лярных решений серди разработчиков ПО. На рис. 13 показан пример страницы задачи в Redmine.

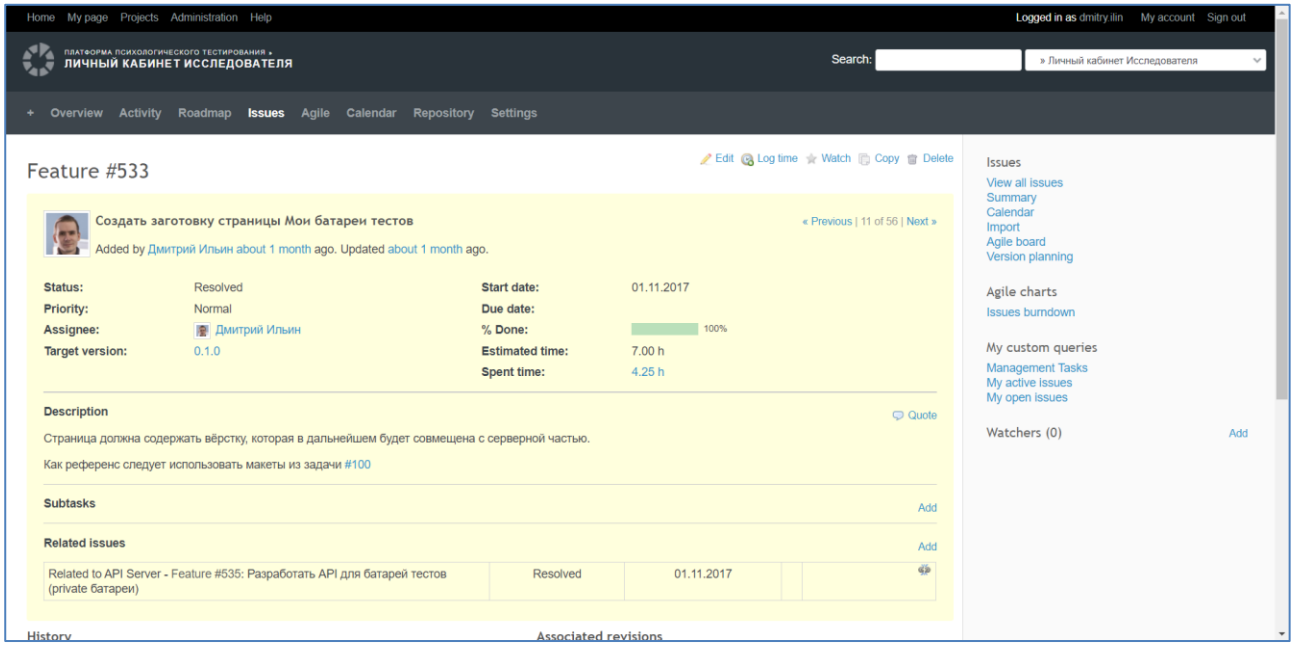

Рис. 13. Пример страницы задачи в Redmine

Одной из удобных функций является возможность интеграции репозитория Git с проектом в Redmine. На рис. 14 показан пример интеграции с данными репозитория.

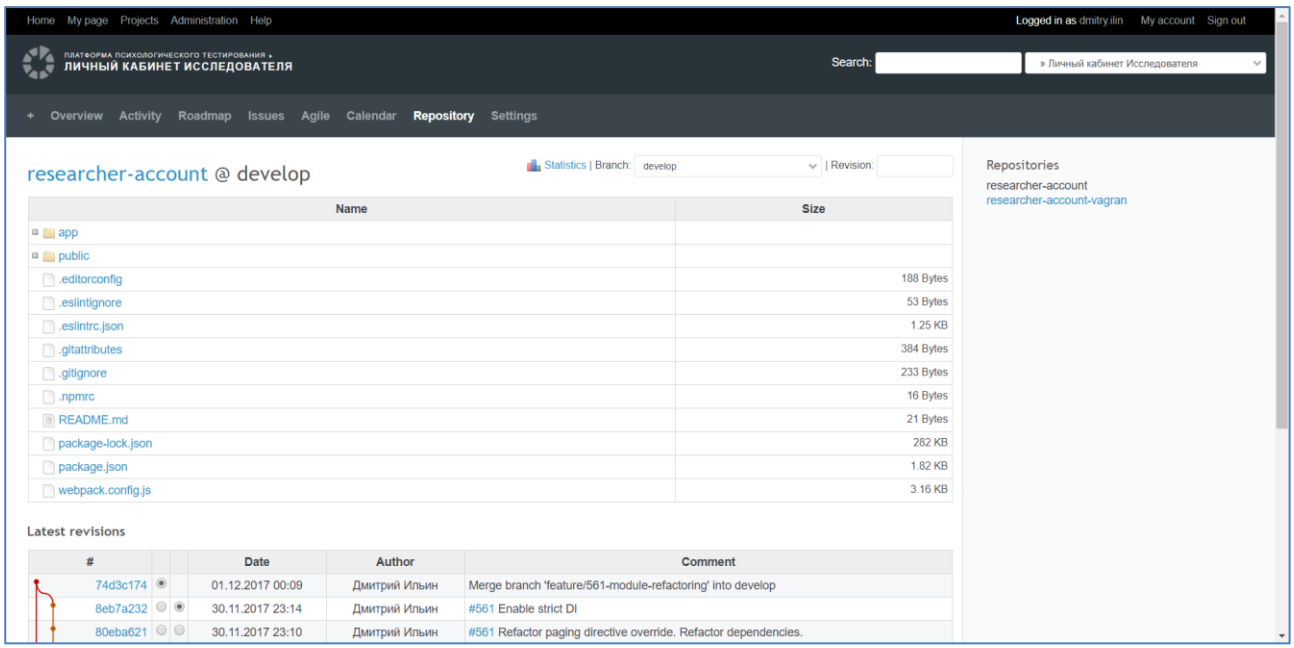

Рис. 14. Интеграция с данными репозитория в Redmine

В системе управления проектами Redmine реализовано отслеживание прогресса изменения задачи, включая интеграцию с данными репозитория (рис. 15). Это очень важно в процессе работы над задачами, так как всегда можно отследить процесс выполнения и понять кем были внесены изменения.

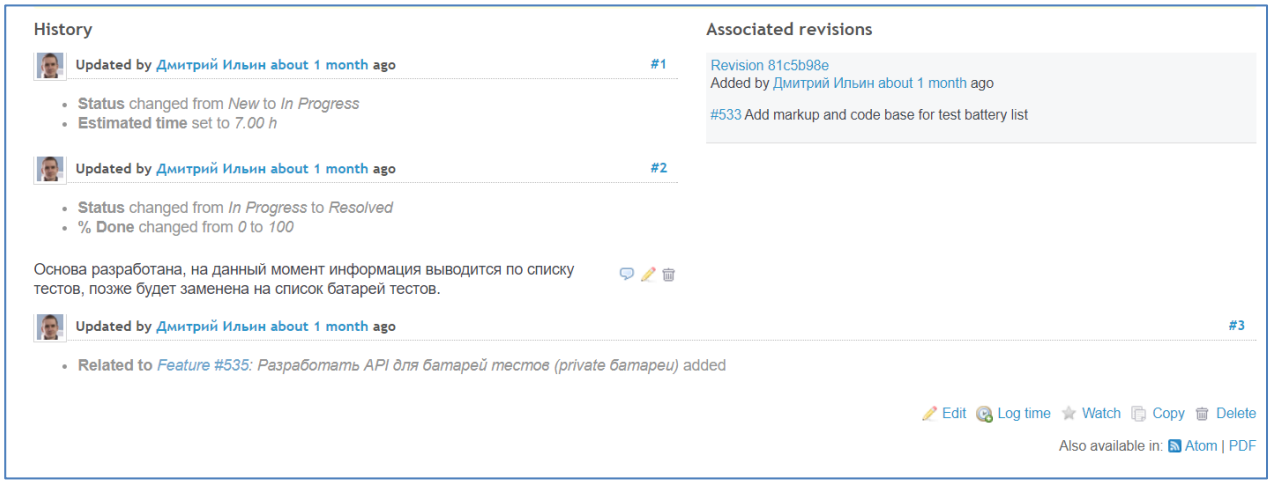

Рис. 15. Отслеживание прогресса задачи в Redmine

Redmine поддерживает плагины и темы, тем самым, можно сильно расширить функциональность. Используются следующие плагины (рис. 16):

- Redmine Agile plugin (Light version) — плагин для поддержки Scrum и Agile методологий, а также добавляет Kanban доску (рис. 17);
- Redmine Checklists plugin (Light version) — добавляет список подзадач;
- Redmine Lightbox 2 — просмотр изображений во всплывающем окне;
- Revision Branches — добавляет информацию о ветках на страницу с ревизией.

Рис. 16. Список используемых плагинов в Redmine

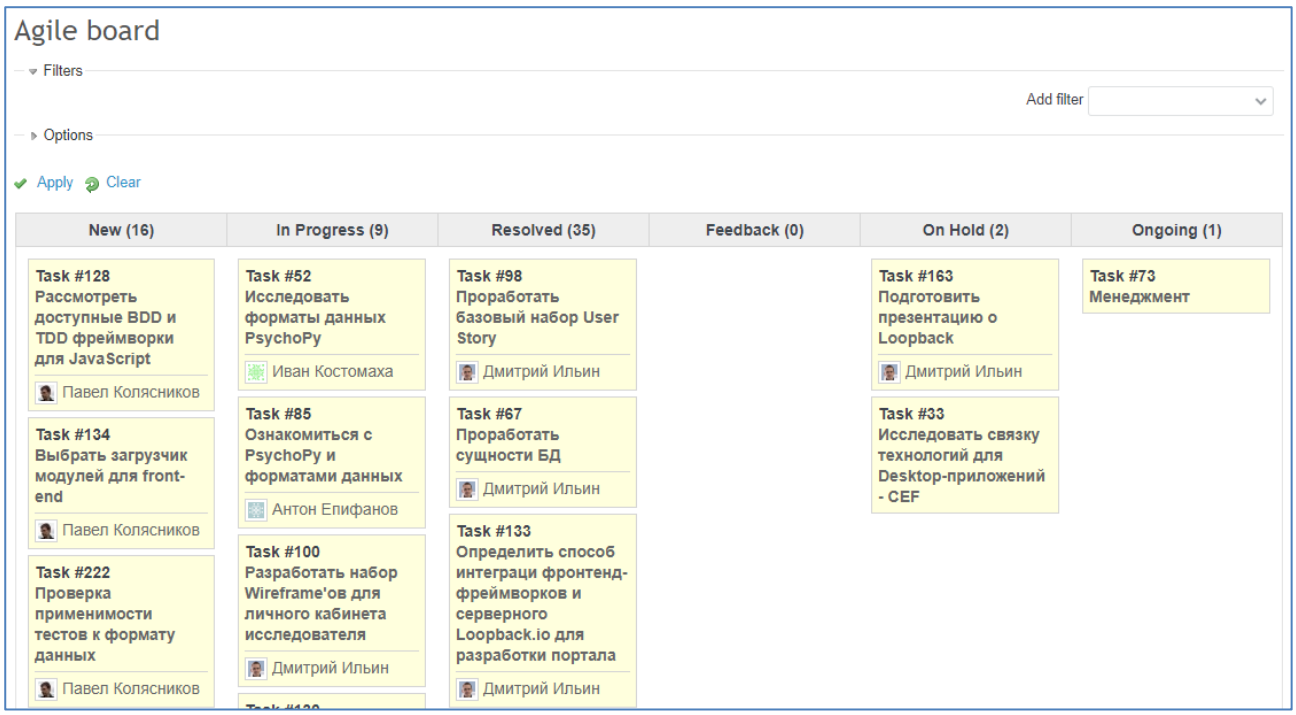

Рис. 17. Kanban-доска плагина Agile проекта в Redmine

Redmine поддерживает различные статусы задач, которые помогают понять в каком состоянии находится та или иная задача. Кроме того, можно добавлять свои собственные статусы и изменять существующие. Используемые статусы задач:

1. **New** — задача, которая только что была создана, но за нее никто не брался. Обычно менеджер распределяет задачи между сотрудниками.

2. **In Progress** — задача, которая на текущий момент находится в процессе выполнения. Такую задачу менеджер просто так переводить на другого исполнителя не будет. Обязательно необходимо в этот статус ставить задачу, если над ней началась работа.
3. **Feedback** — задача, которая требует ответа от другого участника, например, код ревью, тестирование или ответы на вопросы. Главное, при смене статуса не забыть назначить задачу тому, кому этот вопрос адресован.
4. **Resolved** — задача, которая является выполненной (решена, реализована, исправлена), однако заказчик (менеджер) не подтвердил, что его все устраивает.
5. **Closed** — задача, которая является закрытой. Это означает, что задача принята заказчиком (менеджером) и работы по ней больше никогда не ведутся, а все доработки делаются в новых задачах.
6. **Rejected** — задача, которая отменена и необходимости в ее выполнении больше нет.
7. **On Hold** — задача, которая временно приостановлена до изменения статуса менеджером.
8. **Ongoing** — постоянная задача, которая никогда не заканчивается, например, по менеджменту.
9. **Published** — задача, которая применяется для задач по написанию статей и означает, что она опубликована (аналогично статусу closed).

На рис. 18 показан пример схемы жизненного цикла задача. Был разработан собственный жизненный цикл задач, который описывает правила того, в каких состояниях может находиться задача, а также из какого статуса в какой она переходит.

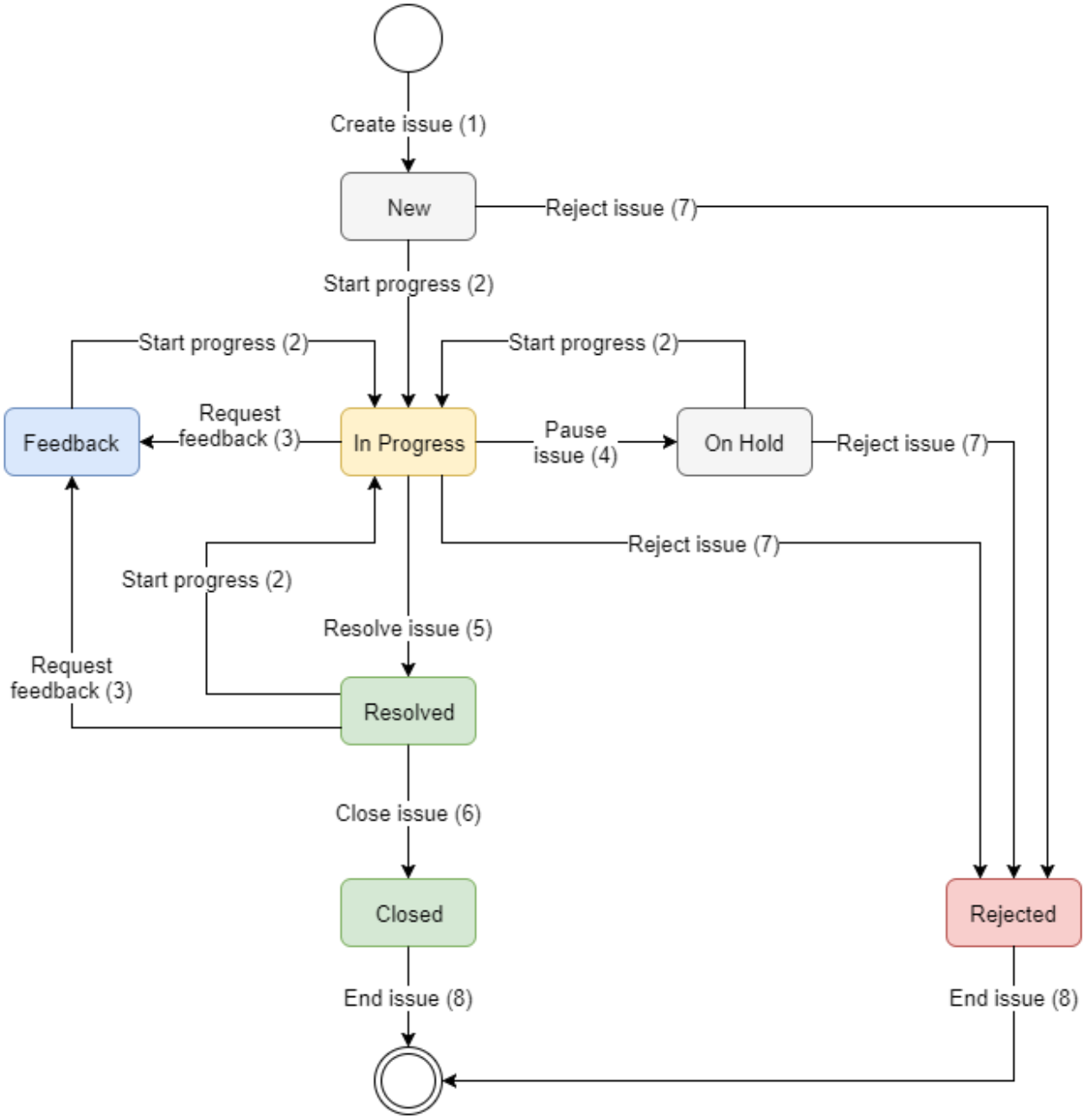

Рис. 18. Разработанный жизненный цикл задач в Redmine

Описание переходов между статусами в жизненном цикле задач:

1. **Create issue (Создание задачи)** — после создания задачи она получает статус «New» и ожидает начала работы после указания исполнителя.
2. **Start progress (Старт работы)** — начало или возобновление ранее приостановленной работы.

3. **Request feedback (Запрос обратной связи)** — задача получает статус «Feedback» если необходимо получить обратную связь, например, ответ на вопрос или проверку результата выполнения задачи.
4. **Pause issue (Приостановка задачи)** — задача получает статус «On Hold», если она по какой-то причине временно приостановлена и не возобновится до смены статуса.
5. **Resolve issue (Задача решена)** — после решения задачи она получает статус «Resolved», где будет ожидать проверки заказчика (менеджера).
6. **Close issue (Задача закрыта)** — после одобрения заказчиком (менеджером) задача переходит в статус закрытия «Closed».
7. **Reject issue (Отмена задачи)** — бывают случаи, когда задачу необходимо отменить по каким-либо причинам. Задачу можно отменить при любом статусе, однако не рекомендуется делать этого из статуса «Resolved», так как задача уже почти решена и ожидает перехода в «Closed».
8. **End issue (Завершение задачи)** — когда задача получает статус «Closed» или «Rejected», то к ней больше не возвращаются.

Система управления версиями Git и методология GitFlow

При современной разработке программного обеспечения для хранения исходных кодов программного обеспечения применяются системы управления версиями (система контроля версий, СКВ; Version Control System, VCS). Данные системы позволяют хранить несколько версий файла и регистрировать изменения в нем, возвращаться к более ранним его версиям, а также отслеживать кем и когда были внесены эти изменения. Среди наиболее популярных систем контроля версий можно отметить Git, Subversion (SVN) и Mercurial.

Система управления версиями Git является распределенной и позволяет работать с репозиториями локально, благодаря чему для работы не требуется установка сервера и инициализировать репозиторий можно прямо из директории с исходным кодом проекта, после чего сразу начинать работать с ним. Для удобства и совместной работы над проектом используется удаленный репози-

торий (origin), установленный на сервере и на который выгружаются изменения из локальных копий разработчиков. Subversion является централизованной системой контроля версий, что менее удобно и накладывает дополнительные ограничения, такие как невозможность работы без сервера. Mercurial также является распределенной системой, однако от нее было решено отказаться из-за того, что Git является более популярным решением, его использует и знает большее количество разработчиков.

Для удобства разработки с использованием Git используется методология GitFlow, которая предоставляет из себя модель ветвления и очень хорошо зарекомендовала себя на проектах различной сложности. Схема методологии GitFlow изображена на рис. 19.

Для удобного использования и администрирования репозиториев на основе системы контроля версиями Git применяются системы управления репозиториями (хостинг репозиториев). Наиболее известными системами являются GitHub, Atlassian Bitbucket и GitLab.

В качестве системы управления репозиториями был выбран GitLab. Данную систему можно установить бесплатно на свой сервис и использовать без каких-либо ограничений в отличие от аналогов. GitHub предоставляется исключительно как SaaS решение и не предоставляет возможности бесплатного создания закрытых репозиториев. Bitbucket также является веб-сервисом, но при этом и позволяет бесплатно создавать репозитории, но с сильно ограниченном количеством человек. Кроме того, Bitbucket предлагает возможность установки на собственный сервер, но за очень большие деньги и предназначено это для очень крупных команд.

В связи с большим количеством людей, имеющих доступ к репозиторию, постоянное изменение их количества и необходимость установки решения на свой сервер, чтобы не зависеть от внешних сервисов, то выбор в пользу GitLab является очевидным.

GitLab поддерживает комментирования кода, что очень удобно и расширяет возможности разработчиков.

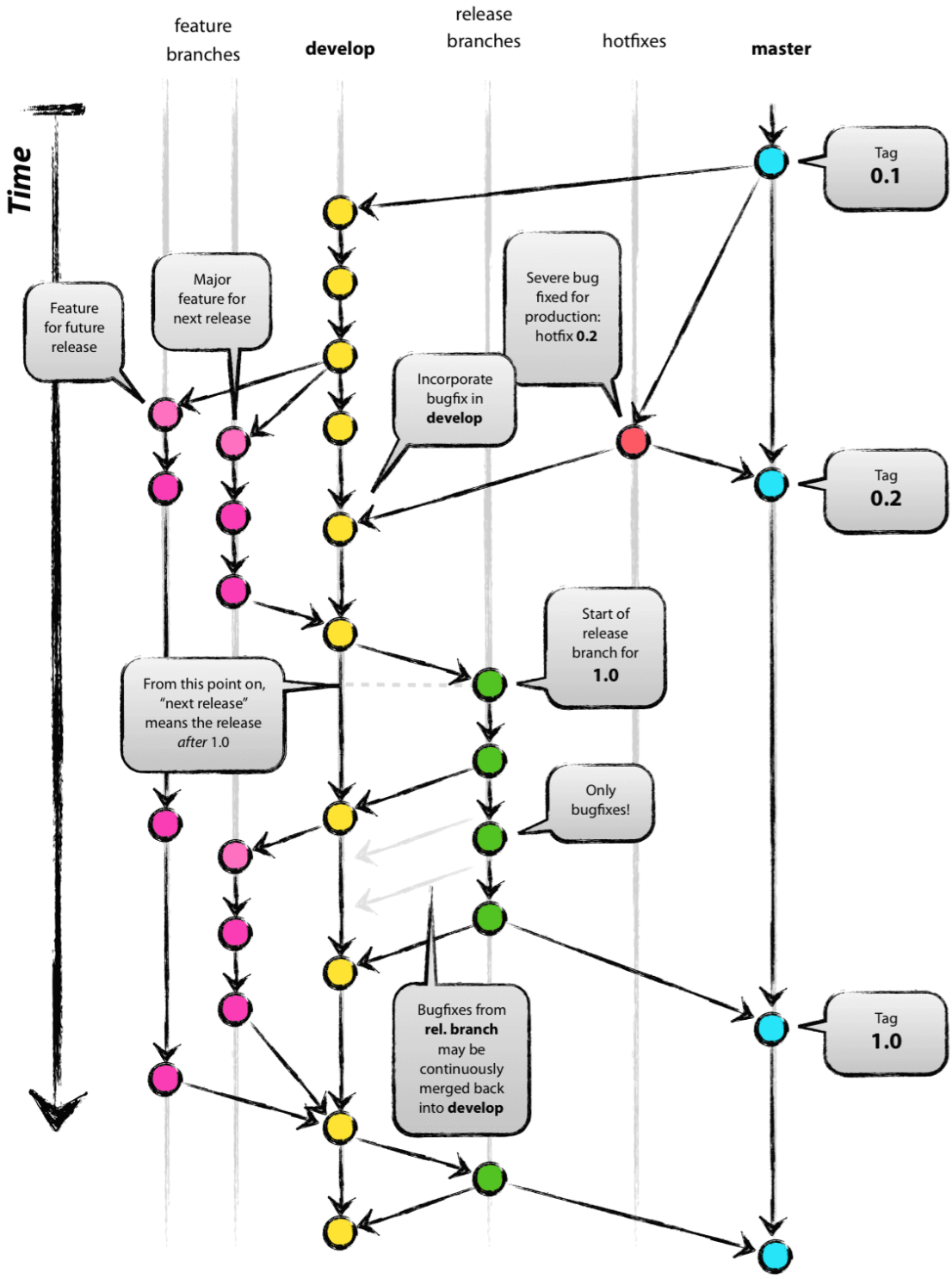

Рис. 19. Используемая модель ветвления GitFlow в Git

8. Экспериментальные оценки эффективности виртуального облачного рабочего окружения распределенной разработки программ

При командной распределенной разработке больших проектов по созданию программного обеспечения в настоящее время широко применяются *виртуальные облачные рабочие окружения*. Данная технология использует виртуализацию и средства управления конфигурациями виртуальных машин для применения необходимых параметров и установки требуемых компонентов, автоматизируя процесс синхронизации, настройки и запуска рабочего окружения.

Использование виртуальных рабочих окружений позволяет избежать различий между локальными рабочими окружениями разработчиков, а также конечной платформой, и значительно упростить установку и настройку окружений на новых машинах.

Рассматривается система для подготовки виртуальных рабочих окружений Vagrant, позволяющая создавать воспроизводимые виртуальные рабочие окружения, снижая ряд сложностей, возникающих по причине несовместимости программно-аппаратных средств, используемых разработчиками.

Использование системы управления разработкой облегчает одновременную распределенную работу над несколькими компонентами разрабатываемого программного обеспечения, а также автоматизирует процесс установки и настройки всех необходимых компонентов среды разработки. Процессы обновления и модификации используемых компонентов также упрощаются, так как достаточно только внести изменения в конфигурационные файлы.

Данные раздел содержит результаты исследования по повышению эффективности виртуальной разработки программного обеспечения цифровой платформы психологических исследований. При использовании для разработки структуры виртуальных машин, максимально приближенных к структуре реальных серверов, были обнаружены характерные проблемы производительности: низкая скорость обмена данными, а также нестабильность работы.

Рассмотрим вопросы исследования среды на примере проекта создания веб-платформы.

В качестве исходного рабочего окружения использована структура виртуальных машин, максимально приближенная к структуре серверов, задействованных в проекте (рис. 20). При данном подходе каждому компоненту платформы соответствует отдельная конфигурация виртуальной машины.

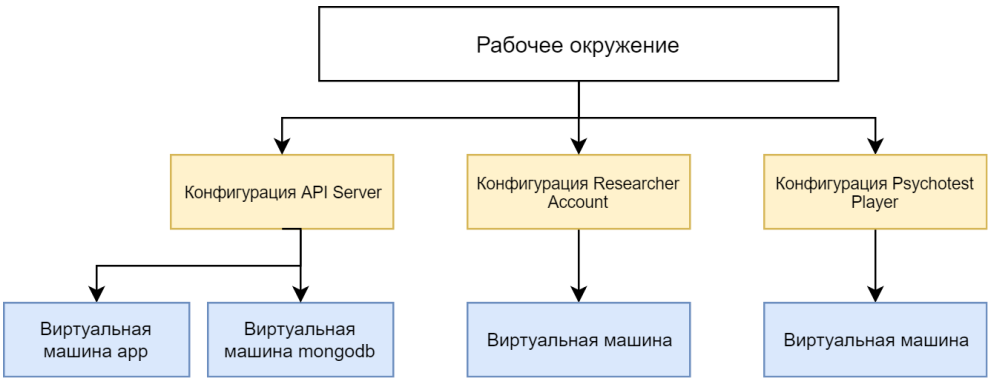

Рис. 20. Схема исходного рабочего окружения

При разработке исходного рабочего окружения были выбраны следующие параметры виртуальных машин для компонентов:

- **API Server**: ОС ubuntu/xenial64, версия 20180424.0.0, 1 CPU, 1024 MB RAM;
- **Researcher Account**: ОС ubuntu/xenial64, версия 20180424.0.0, 2 CPU, 1024 MB RAM;
- **Psychotest Player**: ОС ubuntu/xenial64, версия 20180424.0.0, 2 CPU, 512 MB RAM.

При разработке с использованием подобной структуры рабочего окружения возникает необходимость в одновременном запуске нескольких виртуальных машин для одновременной работы нескольких взаимосвязанных компонентов. Во время практического использования разработанной структуры виртуального рабочего окружения на компьютерах разработчиков было замечено значительное падение производительности. В ходе дальнейших наблюдений были выявлены следующие проблемы использованного рабочего окружения:

- высокая нагрузка на рабочих машинах разработчиков;
- низкая скорость обмена данными виртуальных машин с основной системой;

– высокое время сборки компонентов;
– нестабильность работы из-за возрастающих нагрузок;
– необходимость ручного выполнения большого количества операций при запуске окружения.

В ходе разработки крупных проектов количество разрабатываемых компонентов возрастает, что при использовании подобной структуры может привести к еще большему понижению производительности компьютеров и понижению эффективности разработчиков. Помимо этого, при разработке также необходимо использовать дополнительное программное обеспечение (среда разработки, веб-браузер, сетевые драйверы ввода данных), что также приводит к значительному росту нагрузки на рабочие машины разработчиков, понижению стабильности работы системы и повышению времени выполнения программного обеспечения, задействованного в разработке.

Низкая скорость обмена данными основной системы и виртуальной машины, наблюдаемая при использовании разработанной структуры виртуального окружения, может быть следствием двух факторов: значительного роста нагрузки на машину разработчика, но также используемого виртуальной машиной драйвера для доступа к родительской файловой системе.

Можно выделить задачу необходимости выполнения большого количества повторяющихся действий вручную при запуске рабочего окружения, что также требует затрат времени на ожидание окончания запуска каждой из виртуальных машин для перехода к запуску остальных компонентов.

Описанные выше проблемы значительно влияют на скорость разработки программ в связи с высокими потерями производительности и временными затратами. При расширении общей структуры проекта параллельная разработка нескольких компонентов становится практически невозможной из-за еще большего роста требований к ресурсам.

Таким образом, можно сделать вывод о необходимости исследования подходов к созданию рабочих окружений разработчиков и разработке усовершенствованной структуры. Исходя из выявленных в ходе наблюдений проблем выделены основные требования к искомому решению для рабочих окружений разработчиков:

– быстрое развертывание;

– повышение быстродействия и понижение используемых ресурсов;
– повышение скорости обмена данными между хост-машиной и виртуальной машиной.

Необходимо провести оценку альтернативных технологий и разработать решение, соответствующее описанным требованиям. Также необходимо произвести экспериментальную оценку используемому и альтернативному решениям, и оценить целесообразность перехода на альтернативную структуру рабочего окружения.

Методика исследования

Для проведения экспериментов использовались замеры следующих показателей:

– задействованные ресурсы процессора (процент);
– объем задействованной оперативной памяти (ГБ);
– время запуска виртуальных машин (секунды);
– время выполнения полной сборки компонента Build (секунды);
– время выполнения повторной сборки компонента Watch (секунды).

Все эксперименты проводились на одной рабочей машине со следующей конфигурацией:

– материнская плата: Dell 00TMJ3;
– процессор: Intel Core i5-5250U;
– оперативная память: DDR3 12 ГБ, частота 800 МГц;
– внутренний диск: Samsung SSD 860 EVO 500 ГБ;
– операционная система: Windows 10.

Для каждого из компонентов виртуального рабочего окружения проводилось 10 экспериментов для замера каждого из показателей. Для оценки отклонения полученных значений использован коэффициент Стьюдента при доверительной вероятности P=0.95.

Для измерения задействованных ресурсов процессора и оперативной памяти был разработан программный сценарий на языке bash, который сопоставлял показатели до запуска виртуального рабочего окружения с показателями после полного завершения запуска виртуальной машины. Для получения дан-

ных о необходимых показателях в коде сценария была задействована системная команда «WMIC», предоставляющая возможность получения необходимых данных посредством командного интерфейса.

Все замеры показателей проводились в состоянии ожидания виртуальной машины — после завершения ее запуска и при работе ключевых (веб-сервер, сервер баз данных и т. д.) компонентов.

Для оценки времени запуска виртуальных машин использовалась системная утилита «time», запускаемая с Vagrant в качестве префикса командой «time vagrant up» и предоставляющая после завершения информацию о времени, затраченном на выполнение. При необходимости запуска нескольких компонентов рабочего окружения запуск необходимых контейнеров Vagrant осуществлялся параллельно.

Для оценки времени выполнения задач сборки (полной сборки «build» и повторной сборки в режиме отслеживания «watch») использовались данные, указываемые после выполнения задачи инструментом для сборки веб-приложений Webpack.

Оценка исходного рабочего окружения

С целью оценки нагрузки и скорости работы используемого рабочего окружения был проведен эксперимент, включающий замеры времени, затрачиваемого на запуск, а также оценку задействованных ресурсов центрального процессора и оперативной памяти. Результаты эксперимента представлены в табл. 2.

Таблица 2. Результаты оценки потребления ресурсов рабочими окружениями

Конфигурация	**Кол-во ВМ**	**Время запуска, сек**	**Нагрузка на ЦП, %**	**Занято ОЗУ, ГБ**
Исходная конфигурация (API Server)	2	289,4 ± 2,8	24,3% ± 6,7%	1,5 ± 0,1
Исходная конфигурация (Researcher Account)	1	118,5 ± 13	21,8% ± 6%	0,5 ± 0,08
Исходная конфигурация (Psychotest Player)	1	155,3 ± 19,1	16,1% ± 3%	0,7 ± 0,04
Исходная конфигурация (API Server, Researcher Account)	3	347,4 ± 4,4	35,4% ± 9,1%	2 ± 0,2
Исходная конфигурация (API Server, Psychotest Player)	3	290,1 ± 4,5	29% ± 5,3%	2 ± 0,06
Исходная конфигурация (3 компонента)	4	404,8 ± 4,1	49% ± 5%	2,2 ± 0,18

На основе данных, полученных в ходе эксперимента, можно сделать вывод об объективности проблем, описанных в ходе наблюдений, что подкрепляет необходимость рассмотрения альтернативных решений.

Разработка механизмов повышения эффективности окружения

В связи с высокой нагрузкой, возникающей из-за использования виртуальных машин, используемых в основе Vagrant, целесообразно рассмотреть альтернативные технологии для организации рабочих окружений. В качестве альтернативы была рассмотрена технология Docker, которая также может применяться в целях организации синхронизируемых рабочих окружений.

Система Docker основана на использовании абстрагирования при помощи встроенных в ядро Linux возможностей виртуализации для изоляции различных используемых компонентов в рамках автономных контейнеров с обособленным окружением.

Изначально подобный подход был ориентирован на доставку разработанных программных решений до сервера, однако позднее платформа также стала применяться для замены виртуальных рабочих окружений. При этом Docker может выступать как самостоятельным решением, так и работать в качестве основы решения на основе Vagrant, тем самым заменяя собой использование виртуальных машин.

Для оценки Docker в качестве альтернативы Vagrant был выделен ряд критериев, по которым было проведено сравнение технологий, результаты которого приведены в табл. 3.

Можно сделать вывод, что обе технологии предоставляют сравнимые преимущества при разных подходах к выполнению схожих задач. Использование Docker для синхронизации и быстрой настройки рабочих окружений разработчиков потенциально может являться допустимым решением благодаря преимуществам контейнеризации и решениям, используемым в рамках реализации Docker. Тем не менее, подобный подход также обязывает разработать новую структуру компонентов с использованием Docker контейнеров.

В рамках разрабатываемой платформы важной задачей выделялось сохранение максимальной приближенности к окружению рабочих систем, выполняемых на удаленных серверах. В подобном случае изменение структуры компонентов для использования в качестве контейнеров Docker неизбежно стано-

вится причиной расхождения окружения разработчика и системного окружения сервера, а также значительно осложняет поддержку существующего решения в рамках разрабатываемой платформы. В связи с этим использование технологии контейнеризации становится нецелесообразным.

Таблица 3. Сравнение Vagrant и Docker

Критерий	**Vagrant**	**Docker**
Лицензия	MIT	Apache 2.0
Разработчик	Hashi Corp.	Docker, Inc.
Поддерживаемые платформы	Linux, Windows, macOS	Linux, Windows, macOS
Назначение	Виртуальные рабочие окружения	Контейнеризация приложений и автоматизация задач, доставка компонентов до сервера
Простота эксплуатации	Запуск окружения одной командой	Необходимо понимание системы контейнеров и зависимостей; запуск окружения одной командой в комбинации с Vagrant
Простота конфигурирования	Конфигурационный файл на языке Ruby	Собственный формат конфигурационных файлов
Структура контейнеров	Контейнер включает в себя все зависимости, указанные в конфигурации, является лишь средой выполнения	Каждый компонент и его зависимости являются отдельными контейнерами

Другим возможным решением в данном случае может выступить сохранение Vagrant в качестве основы выбранного решения, но с использованием единой виртуальной машины для выполнения всех компонентов.

В качестве альтернативного решения стоит рассмотреть конфигурацию на основе единого виртуального окружения Vagrant. В этом случае каждый компонент выполняется в рамках одной виртуальной машины, используя также единую модульную конфигурацию Vagrant (рис. 21).

Использование подобной структуры должно понизить нагрузку на компьютер за счет использования лишь одной виртуальной машины, при этом использование модульной конфигурации, основанной на исходном решении, должно упростить поддержку и минимизировать расхождения с окружением сервера.

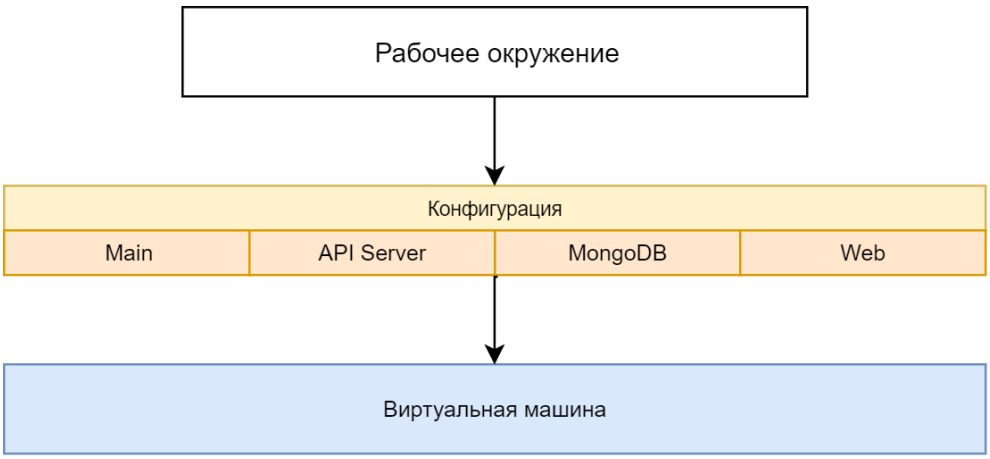

Рис. 21. Альтернативная структура рабочего окружения

Для сравнения исходной и альтернативной структур рабочего окружения была проведена экспериментальная оценка потребляемых ресурсов (табл. 4). В качестве конфигурации виртуальной машины в основе альтернативного решения использовалась ОС ubuntu/xenial64, версия 20180424.0.0 с 2 CPU, 2048 MB RAM.

Таблица 4. Результаты замеров выполнения рабочих окружений

Конфигурация	**Кол-во ВМ**	**Время запуска, сек**	**Нагрузка на ЦП, %**	**Занято ОЗУ, ГБ**
Исходная конфигурация (3 компонента)	4	404,8 ± 4,1	49% ± 5%	2,2 ± 0,18
Альтернативная конфигурация (3 компонента)	1	210,3 ± 4,6	23,8% ± 7,7%	1,3 ± 0,13

В целях дополнительного повышения эффективности также стоит рассмотреть альтернативный драйвер обмена данными с родительской системой, основанный на протоколе NFS, так как подобное решение может значительно повысить скорость работы с файловой системой и выполнение задач.

Для использования NFS на машинах под управлением ОС Windows требуется использование дополнительного драйвера. Кроме того, для корректной работы протокола NFS для обмена данными между родительской системой и виртуальной машиной необходимо подключение расширения bindfs, позволяющего перенести права доступа родительской системы.

В связи с архитектурной особенностью протокола NFS, не предоставляющего реализацию системных сигналов при отслеживании изменения файлов,

также возникла необходимость модифицировать конфигурацию ПО Webpack, используемого для сборки веб компонентов. Были внесены следующие изменения:

- для обеспечения совместимости с NFS была установлена опция «watchOptions.poll», реализующая обход отслеживаемых файлов по заданному временному интервалу;
- для исключения неизменяемых файлов библиотек из опроса была использована опция «watchOptions.ignore».

Были проведены дополнительные замеры времени, затрачиваемого на сборку компонента при внесенных изменениях конфигурации (табл. 5).

Таблица 5. Результаты замеров времени сборки компонентов (в секундах)

Конфигурация	**Researcher Account**		**Psychotest Player**	
	Build	**Watch**	**Build**	**Watch**
Исходная конфигурация (3 компонента)	120,4 ± 2,7	5 ± 0,4	131 ± 3,1	4,4 ± 0,1
Альтернативная конфигурация (3 компонента)	100,7 ± 2,5	3,6 ± 0,3	82,4 ± 2,4	3,3 ± 0,1
Улучшенная альтернативная конфигурация (3 компонента)	90,1 ± 2,4	1,2 ± 0,3	79 ± 1,3	1,1 ± 0,2

Подобные изменения конфигурации также повлекли за собой рост времени запуска (с 210,3 до 227,7 секунд в среднем), связанный с добавлением времени ожидания запуска драйвера NFS и монтирования директорий в виртуальной машине, а также рост используемых ресурсов процессора (с 24% до 29% в среднем. Тем не менее, показатели потребляемых ресурсов все еще остаются значительно более низкими в сравнении с полным запуском всех компонентов исходного рабочего окружения. При этом было замечено резкое понижение временных затрат на операции сборки компонентов при использовании предложенных улучшений.

Результаты экспериментальных оценок

Итоговые результаты проведенных экспериментов по оценке времени запуска и нагрузки на компьютер различных конфигураций виртуального рабочего окружения разработчиков приведены в табл. 6.

Результаты экспериментов по оценке времени выполнения полной сборки компонента Build и повторной сборки компонента Watch для компонентов платформы Researcher Account и Psychotest Player — представлены в табл. 7.

Таблица 6. Результаты оценки нагрузки и времени запуска рабочих окружений

Конфигурация	**Кол-во ВМ**	**Время запуска, сек**	**Нагрузка на ЦП, %**	**Занято ОЗУ, ГБ**
Исходная конфигурация (API Server)	2	289,4 ± 2,8	24,3% ± 6,7%	1,5 ± 0,1
Исходная конфигурация (Researcher Account)	1	118,5 ± 13	21,8% ± 6%	0,5 ± 0,08
Исходная конфигурация (Psychotest Player)	1	155,3 ± 19,1	16,1% ± 3%	0,7 ± 0,04
Исходная конфигурация (API Server, Researcher Account)	3	347,4 ± 4,4	35,4% ± 9,1%	2 ± 0,2
Исходная конфигурация (API Server, Psychotest Player)	3	290,1 ± 4,5	29% ± 5,3%	2 ± 0,06
Исходная конфигурация (3 компонента)	4	404,8 ± 4,1	49% ± 5%	2,2 ± 0,18
Альтернативная конфигурация (3 компонента)	1	210,3 ± 4,6	23,8% ± 7,7%	1,3 ± 0,13
Улучшенная альтернативная конфигурация (3 компонента)	1	227,7 ± 3,4	28,7% ± 5,4%	1,2 ± 0,09

Таблица 7. Результаты оценки времени выполнения сборки компонентов

Конфигурация	**Researcher Account**		**Psychotest Player**	
	Build	**Watch**	**Build**	**Watch**
Исходная конфигурация (API Server)	101,9 ± 2,9	3 ± 0,4	86,1 ± 2,1	3,1 ± 0,1
Исходная конфигурация (Researcher Account)	100,6 ± 2,5	2,7 ± 0,4	91,8 ± 2,4	3,7 ± 0,1
Исходная конфигурация (Psychotest Player)	99 ± 2,4	3,6 ± 0,4	89,3 ± 3	3,7 ± 0,1
Исходная конфигурация (API Server, Researcher Account)	98,7 ± 2,2	3,7 ± 0,5	119,2 ± 2,4	4 ± 0,2
Исходная конфигурация (API Server, Psychotest Player)	94,6 ± 2,7	4,1 ± 0,4	137,7 ± 3,6	4 ± 0,2
Исходная конфигурация (3 компонента)	120,4 ± 2,7	5 ± 0,4	131 ± 3,1	4,4 ± 0,1
Альтернативная конфигурация (3 компонента)	100,7 ± 2,5	3,6 ± 0,3	82,4 ± 2,4	3,3 ± 0,1
Улучшенная альтернативная конфигурация (3 компонента)	90,1 ± 2,4	1,2 ± 0,3	79 ± 1,3	1,1 ± 0,2

Из результатов проведенных экспериментов видно, что предложенная альтернативная конфигурация оказывает значительно меньшую нагрузку на

центральный процессор (рис. 22). Улучшенная альтернативная конфигурация использует несколько больше ресурсов процессора, что связано с использованием дополнительного драйвера, однако даже в этом случае нагрузка оказывается значительно ниже, чем при полном запуске исходного рабочего окружения.

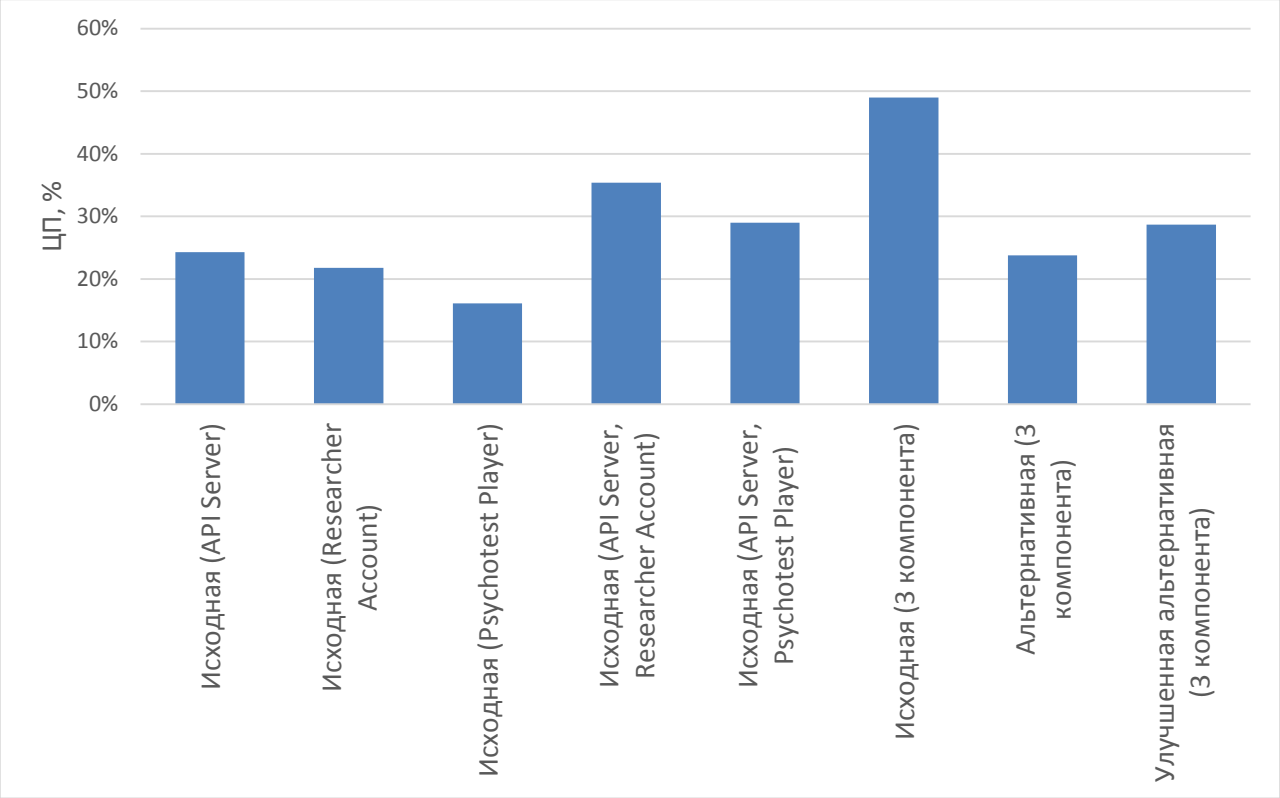

Рис. 22. Средняя нагрузка на центральный процессор, в процентах

Подобные изменения заметны также при сравнении используемой оперативной памяти (рис. 23).

Понижение нагрузки на процессор, а также понижение количества используемых виртуальных машин, повлекло за собой значительное понижение среднего времени, необходимого для запуска виртуального рабочего окружения (рис. 24). Улучшенная альтернативная конфигурация, как и в случае с нагрузкой на процессор, уступает по этому показателю альтернативной конфигурации без дополнительных модификаций. Это также может быть следствием использования дополнительного драйвера и необходимости ожидания его инициализации.

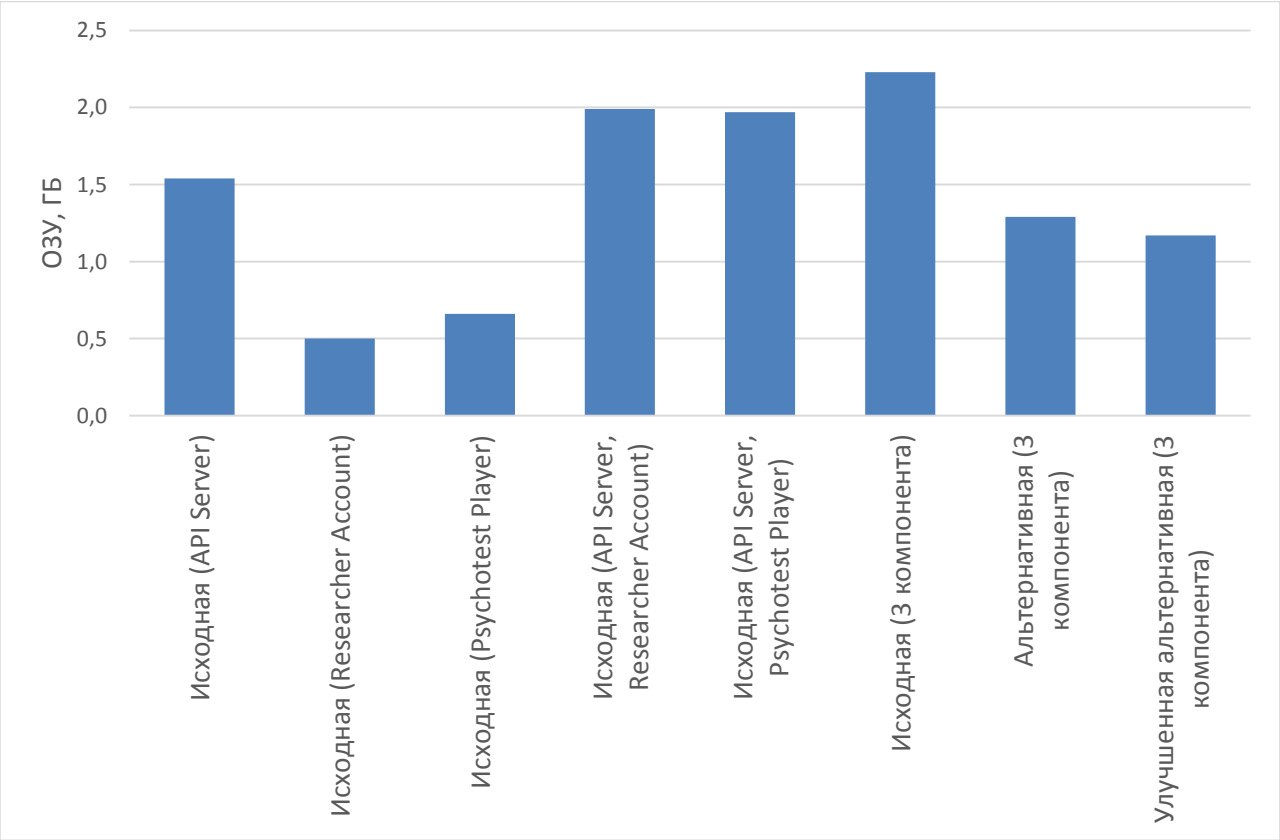

Рис. 23. Занято ОЗУ в среднем, ГБ

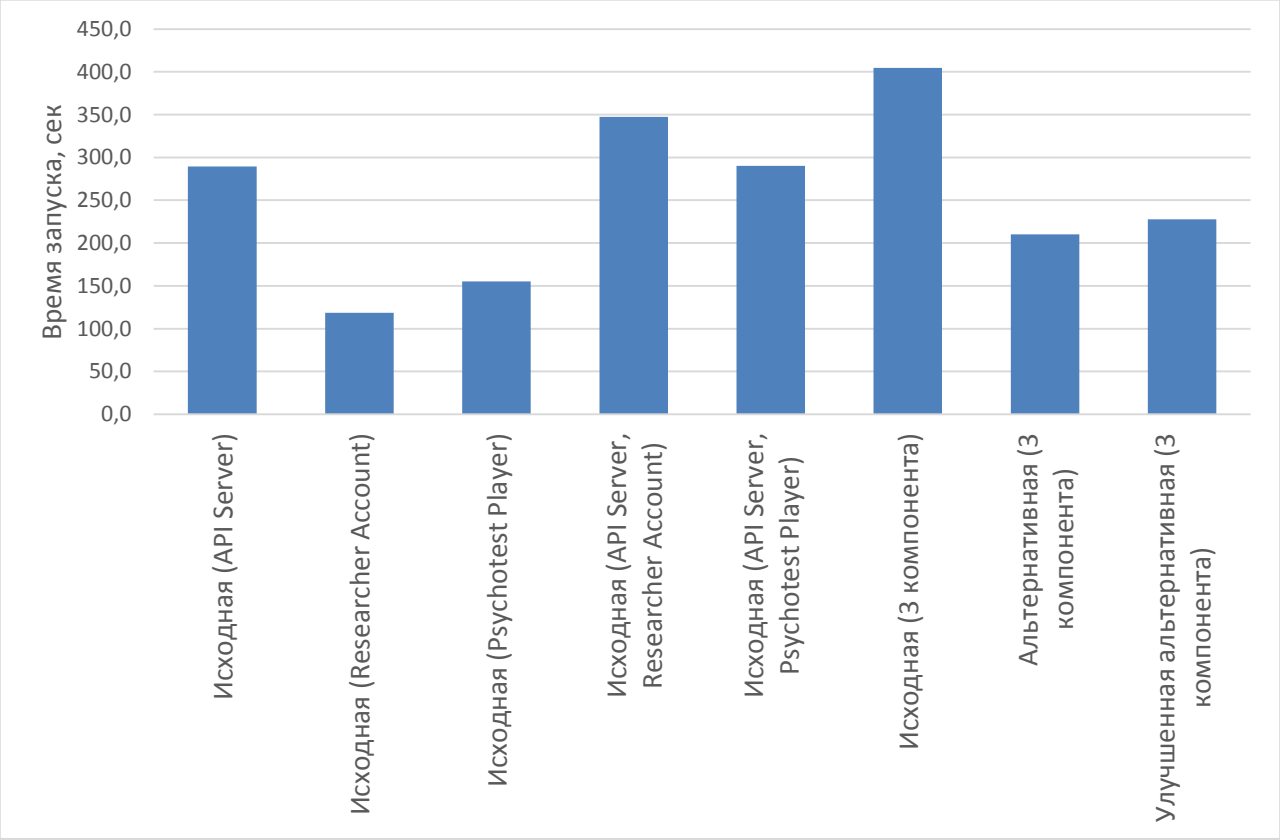

Рис. 24. Среднее время запуска рабочего окружения, сек

Тем не менее, улучшенная альтернативная конфигурация лидирует по показателям при выполнении полной сборки компонентов Researcher Account и Psychotest Player (рис. 25), а также демонстрирует значительно лучшие показатели при выполнении их повторной сборки (рис. 26). Подобные улучшения показателей являются прямым следствием использования драйвера NFS, повышающего скорость обмена данными с хост-машиной, что позволяет значительно сократить временные затраты при сборке компонентов.

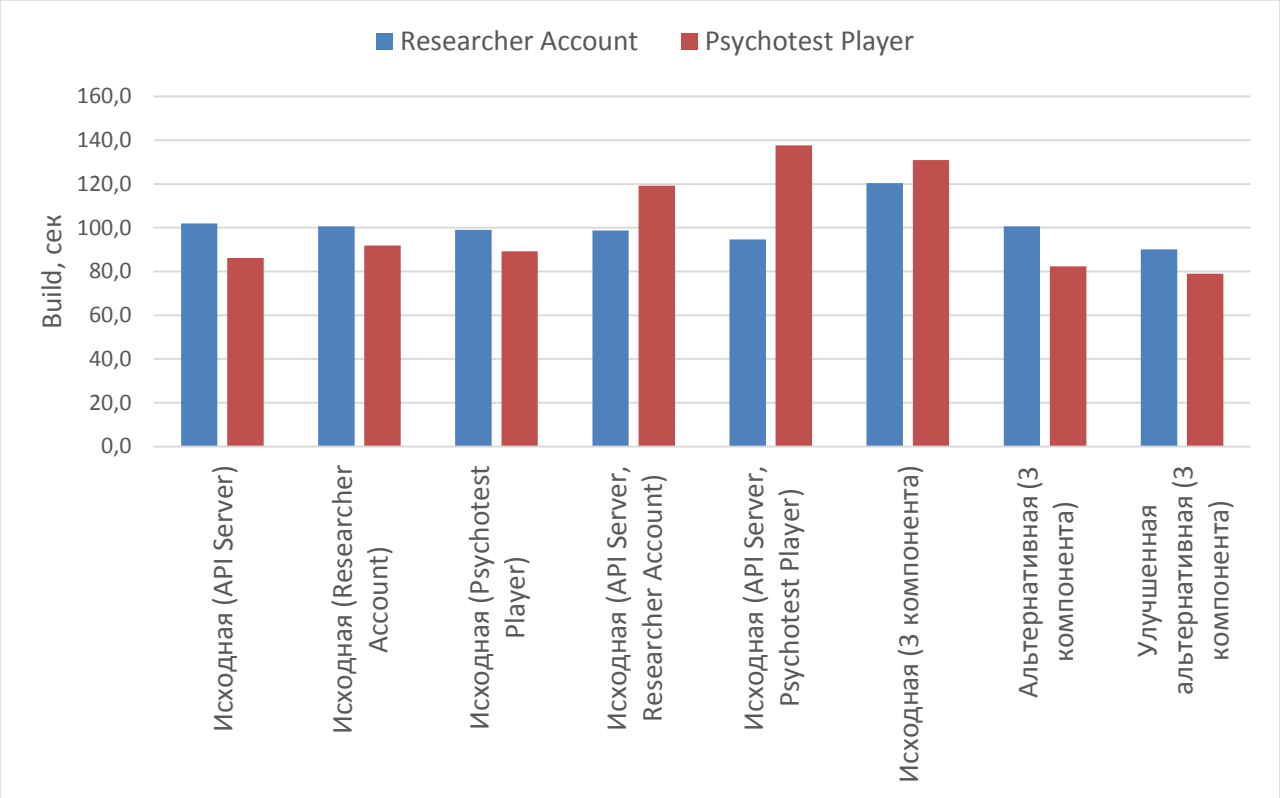

Рис. 25. Среднее время выполнения полной сборки компонентов Researcher Account и Psychotest Player (Build), сек

Таким образом, полученная улучшенная альтернативная конфигурация рабочего окружения для Vagrant демонстрирует значительное понижение нагрузки на процессор и загрузки оперативной памяти в сравнении с полным запуском исходного рабочего окружения. При этом также демонстрируется значительное ускорение процесса сборки компонентов, что понижает время ожидания со стороны разработчиков, тем самым повышая их эффективность.

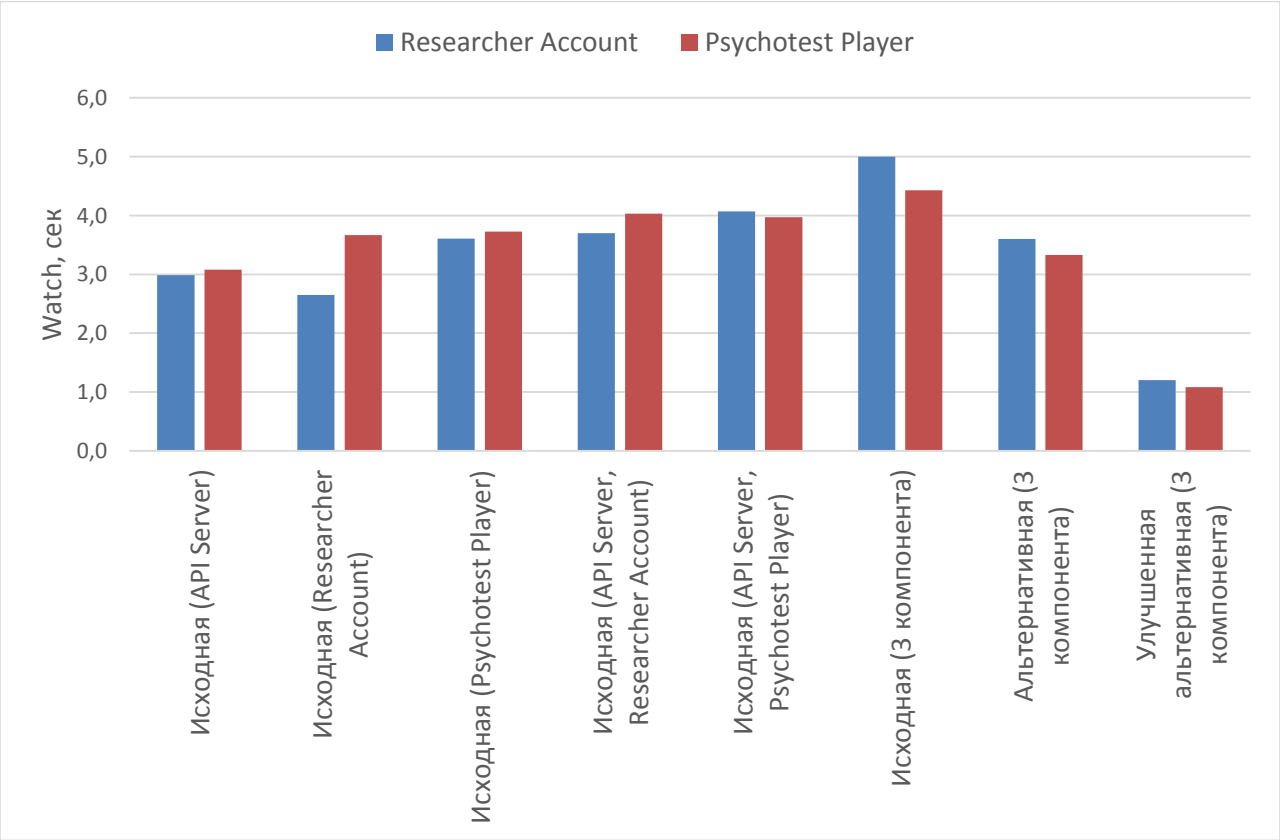

Рис. 26. Среднее время выполнения повторной сборки компонентов Researcher Account и Psychotest Player (Watch), сек

При использовании виртуального рабочего окружения, в структуре которого каждый компонент использовал автономную виртуальную машину, были замечены проблемы с высоким потреблением ресурсов и падением производительности рабочих машин разработчиков, в связи с чем возникла необходимость в рассмотрении альтернативных решений.

Таким образом, проведен анализ причин снижения производительности при использовании рабочих окружений на основе виртуализации. Рассмотрены основные технологии, применяемые для разработки виртуальных рабочих окружений, а также предложена улучшенная структура. Кроме того, была произведена экспериментальная оценка исходного и альтернативного решений.

Реализованное альтернативное решение на основе единого рабочего окружения показало значительно меньшее потребление ресурсов, а также понижение времени выполнения задач сборки, благодаря драйвера для доступа к файловой системе на основе NFS.

Практическая работа №1. Система виртуализации VirtualBox

Цель работы: установить дистрибутив GNU/Linux на виртуальной машине.

Для выполнения работы предлагается использовать установочный образ дистрибутива GNU/Linux Ubuntu Server, который можно скачать по следующей ссылке: https://www.ubuntu.com/download/server

Допустимо использовать любой другой GNU/Linux дистрибутив.

Также убедитесь, что на Вашем ПК установлено следующее ПО:

1. VirtualBox

Настройка виртуальной машины

В начале работы необходимо создать и настроить новую виртуальную машину, используя менеджер VirtualBox. Нажмите кнопку «Создать» в интерфейсе главного окна VirtualBox (см. рис. П.1).

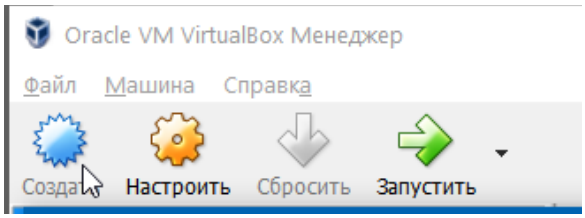

Рис. П.1 Панель управления виртуальными машинами

Для перехода к режиму детальной настройки необходимо нажать кнопку «Экспертный режим». В появившемся окне (рис. П.2) выберите тип «Linux» и версию «Ubuntu (64-bit)». Также необходимо указать объем оперативной памяти и выбрать виртуальный жесткий диск (возможно использование параметров по умолчанию).

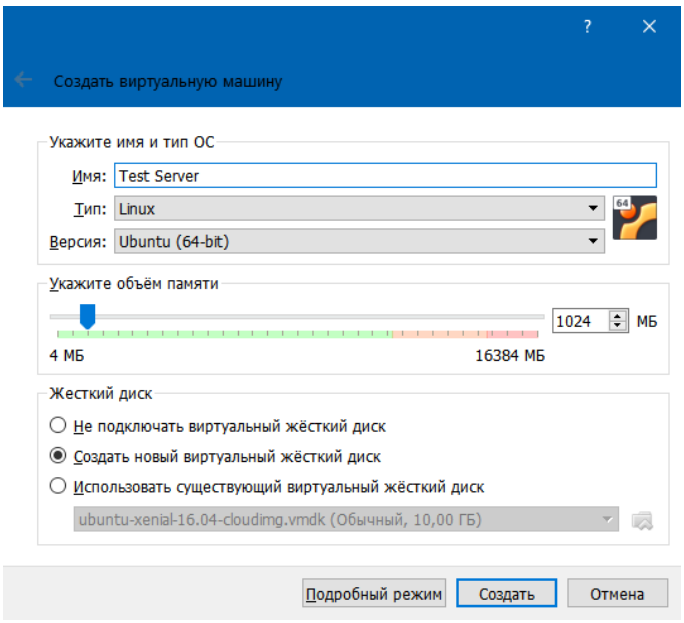

Рис. П.2 Конфигурация новой виртуальной машины

В открывшемся окне конфигурации нового виртуального жесткого диска необходимо указать его тип и объем, как показано на рис. П.3 (возможно использование параметров по умолчанию), после чего подтвердить создание виртуального диска.

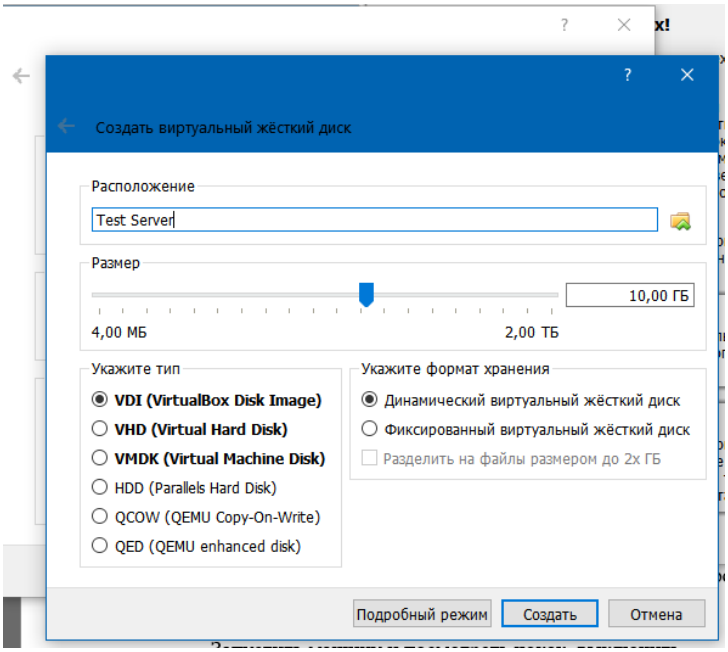

Рис. П.3. Интерфейс настройки виртуального диска

После завершения процесса создания виртуальной машины, запустите ее. После завершения процесса первичной настройки ознакомьтесь с открывшимся интерфейсом плеера виртуальных машин. После отображения ошибки BIOS из-

за отсутствия устройства для загрузки системы, выключите запущенную виртуальную машину, выключив захват ввода (нажатием хост-клавиши, по умолчанию правый Ctrl) и выбрав пункт «Выключить машину» при закрытии окна.

Установка дистрибутива GNU/Linux

После завершения работы виртуальной машины необходимо открыть меню конфигурации, нажав правой кнопкой на виртуальную машину в списке и выбрав пункт «Настроить…» (рис. П.4).

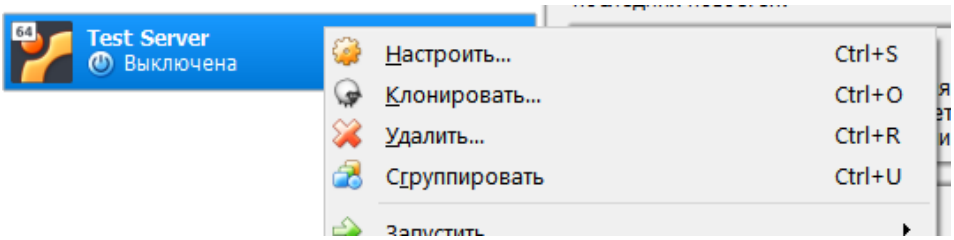

Рис. П.4. Параметры виртуальной машины

В меню настроек перейдите в раздел «Носители» и выберите контроллер IDE, отображенный в списке пиктограммой оптического диска. В меню атрибутов нажмите на пиктограмму диска и в появившемся меню нажмите «Выбрать образ оптического диска…». Укажите путь до образа ISO установочного диска, как показано на рис. П.5. Отметьте пункт «Живой CD/DVD».

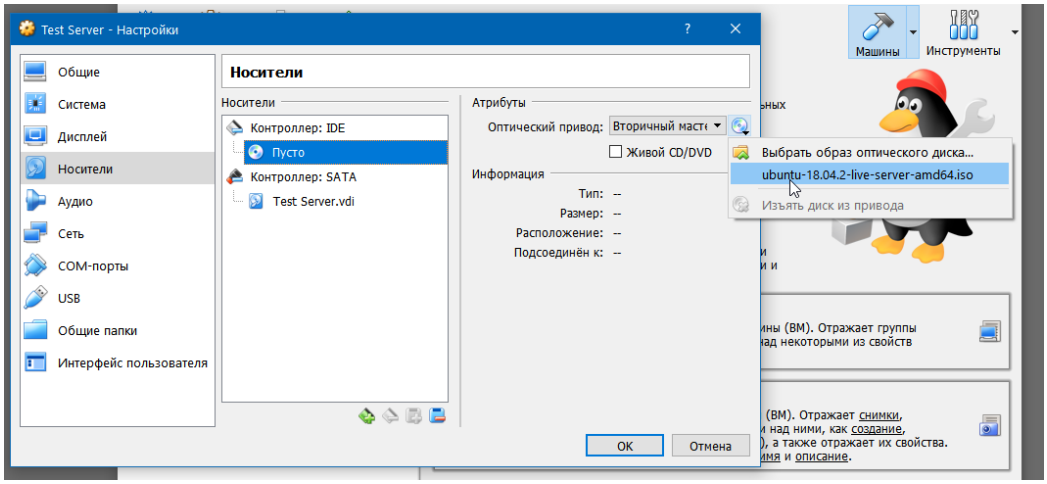

Рис. П.5. Выбор образа оптического диска

Повторно запустите виртуальную машину для начала загрузки с использованием образа установочного диска.

Продолжите процесс установки системы. Пройдите процесс конфигурации инсталлятора (дополнительные изменения параметров не требуются) и дождитесь завершения процесса установки.

После этого авторизуйтесь в установленной операционной системе и ознакомьтесь с ней, используя навыки работы с операционными системами GNU/Linux. По завершению ознакомления выключите виртуальную машину средствами VirtualBox.

Практическая работа №2. Система создания и конфигурирования виртуальной среды разработки

Цель работы: подготовить конфигурацию виртуальной машины с использованием Vagrant.

Vagrant — свободное и открытое программное обеспечение для создания и конфигурирования виртуальной среды разработки. Vagrant позволяет конфигурировать и автоматизировать задачи, выполняемые в средах виртуализации, вроде VirtualBox, с помощью средств управления конфигурациями, к примеру, Puppet или Ansible.

Общие директории (также называемые shared directories) — это механизм, использующийся для получения доступа к директории хост-системы в виртуальной машине. Для создания доступа к общим директориям используются виртуальный сетевой драйвер и соответствующая конфигурация виртуальной машины

Перед началом работы следует убедиться, что на ПК установлено следующее ПО:

1. VirtualBox;
2. Vagrant;
3. Редактор кода с подсветкой синтаксиса Ruby, и YML (например, Atom или VSCode);
4. Git, с наличием инструмента Git Bash (для ОС Windows).

Для работы предлагается создать папку **lab-2**. В дальнейшем подразумевается, что все файлы будут располагаться относительно этой папки.

Установка базового образа операционной системы

Создайте пустую папку для будущего проекта. В этой папке создайте пустой текстовый файл и дайте ему название **Vagrantfile**. Убедитесь в корректности используемого регистра.

Откройте файл для редактирования и разместите в нем следующий код:

```
Vagrant.configure("2") do |config|
 config.vm.box = "ubuntu/xenial64"
 config.vm.box_version = "20180424.0.0"
 # Network Settings
 config.vm.hostname = "vm.localhost"
 config.vm.network "forwarded_port", guest: 80, host: 8080
```

```
 # VM Settings
 config.vm.provider "virtualbox" do |v|
 v.memory = 1024
 v.cpus = 2
 end
end
```

Обратите внимание на содержательную часть. Внутри конфигурационного блока заданы параметры виртуальной машины, отвечающие за версию используемой операционной системы. Далее, в разделе настроек сети, заданы имя виртуальной машины и переадресация порта 8080 с хост-машины на виртуальную машину. Ниже заданы настройки объема оперативной памяти и количества процессорных ядер, доступных виртуальной машине.

С помощью Git Bash или иного терминала перейдите в директорию проекта и выполните следующую команду: **vagrant up**. Дождитесь, пока установка завершится полностью.

Подключение к виртуальной машине

После завершения установки, подключитесь к виртуальной машине с помощью команды: **vagrant ssh.**

Для того, чтобы убедиться, что консольный доступ к виртуальному окружению получен, выведите полное имя машины. В GNU/Linux это возможно с помощью команды: **hostname -f.**

Проверьте, что пользователь имеет права на исполнение команд от имени пользователя **root**, попробуйте вывести содержимое файла, недоступного обычным пользователям для чтения. Например: **sudo less /var/log/kern.log**

Настройка характеристик виртуальной машины

Отредактируйте **Vagrantfile**, добавив в секцию VM Settings новую настройку, ограничивающую максимальную утилизацию процессорных ресурсов. Теперь она должна выглядеть следующим образом:

```
# VM Settings
config.vm.provider "virtualbox" do |v|
 v.memory = 1024
 v.cpus = 2
 v.customize ["modifyvm", :id, "--cpuexecutioncap", "30"]
end
```

Изменения вступают в силу не сразу, а только после перезапуска виртуальной машины. Для этого выполните команду: **vagrant reload**. Сравните время запуска виртуальной машины с указанными изменениями и без них.

Настройка общих папок

Подключитесь к виртуальной машине используя команду **vagrant ssh**. Перейдите в каталог **/vagrant**. Это каталог, к которому по умолчанию подключаются общие директории. Для того, чтобы посмотреть перечень файлов и директорий введите следующую команду: **ls -lah**.

Как можно увидеть из выведенного на экран результата, к этому каталогу подключена папка с самой конфигурацией виртуальной машины, т. е. папка, в которой размещен **Vagrantfile**.

Для того, чтобы изменить это, требуется указать подключаемую папку в явном виде. Добавьте в **Vagrantfile** следующий фрагмент кода:

```
config.vm.synced_folder "../", "/vagrant", id: "vagrant-root",
 owner: "vagrant",
 group: "www-data",
 mount_options: ["dmode=775,fmode=664"]
```

Новая общая директория будет подключена после перезагрузки виртуальной машины. Чтобы проверить результат изменений, повторите операции, описанные в начале раздела. Таким образом с помощью одного виртуального рабочего окружения можно обеспечивать работу нескольких проектов, что особенно важно, когда они взаимосвязаны.

Завершение работы виртуальной машины

Для завершения работы выполните команду: **vagrant halt**. Виртуальная машина будет выключена, тем самым высвободив занятые ресурсы хост-машины. Чтобы удостовериться в результате, введите команду: **vagrant status**. Она покажет текущее состояние виртуальной машины, если в каталоге присутствует **Vagrantfile**. Для отслеживания статуса всех виртуальных машин под управлением Vagrant следует использовать команду: **vagrant global-status**.

Практическая работа №3. Конфигурирование виртуальной среды

Цель работы: сконфигурировать веб-сервер для работы с файлами, находящимися в общей директории.

Убедитесь, что успешно прошли предыдущую работу. Эта Практическая работа основывается на результате предыдущей, за исключением ограничения по утилизации CPU, которое можно отключить.

В терминологии Vagrant «провизия» — это процесс конфигурирования программных компонентов виртуальной машины и установки в нее программного обеспечения.

Для работы предлагается создать папку **lab-3** и перенести в нее результаты предыдущей работы. В дальнейшем подразумевается, что все файлы будут располагаться относительно этой папки.

Добавление базовой настройки «провизии»

Откройте Vagrant-файл для редактирования и разместите в нем следующий код:

```
config.vm.provision "shell", path: "./provision/script.sh"
```

В нем указывается то, что на этапе «провизии» Vagrant должен обратиться к файлу **script.sh** и выполнить его как скрипт на языке Bash. Так как его в указанном каталоге нет (как и самого каталога), создайте файл со следующим содержимым:

```
touch /vagrant/touchfile.txt
date >> /vagrant/touchfile.txt
```

Запустите процесс «провизии» с помощью команды: **vagrant provision**. Повторите эту операцию и обратите внимание на содержимое файла touchfile.txt. Вносимые изменения не заменяют, а дополняют существующий файл. Эти изменения не идемпотентны, что важно учитывать при настройке каких-либо программных систем (веб-серверов, систем отправки писем и др.). Необходимо разрабатывать такие скрипты «провизии», которые не будут подвергать изменениям виртуальную машину, если конфигурация не менялась.

Установка программ

Для установки программы с помощью Bash-скрипта достаточно выполнить стандартные для целевой операционной системы команды. Предлагается установить следующие программы: **git**, **htop**, **nginx**.

Измените файл **script.sh**, пусть теперь он будет следующего вида:

```
apt update
apt install git htop nginx -y
```

Проведите повторный запуск «провизии», после чего повторно подключитесь к виртуальной машине и проверьте результаты. При выполнении команды **htop** должен запускаться монитор процессов в операционной системе.

Так как утилита **apt** отслеживает, какие именно пакеты были установлены, то в общем случае этого достаточно. Однако, следует учитывать, что в приведенном примере нет указания конкретных версий программ. Это стоит учесть, так как при обновлении программы могут быть внесены изменения, несовместимые с текущей конфигурацией виртуальной машины.

Настройка веб-сервера

Изменить корневую директорию веб-сервера можно с помощью следующего набора команд:

```
sed -i '\/root \/var\/www\/html;/c root /vagrant/lab-3/html;'
/etc/nginx/sites-available/default
service nginx restart
```

Добавьте их в файл **script.sh**, а затем, повторите запуск процесса «провизии». После этого зайдите в браузере по адресу http://localhost:8080. Пока в каталоге **lab-3** нет директории **html** и файла **index.html**, поэтому страница ошибки — ожидаемый результат.

Создайте их и разместите следующий код в файл **index.html**:

```
<!DOCTYPE html>
<html lang="ru">
<head>
 <meta charset="UTF-8">
 <meta name="viewport" content="width=device-width, initial-
scale=1, minimum-scale=1, maximum-scale=1">
 <title>Веб-страница</title>
 <base href="/">
```

```
</head>
<body>
 <h1>Hello World</h1>
</body>
</html>
```

Зайдите повторно в браузере по адресу http://localhost:8080. Теперь, вместо ошибки, должна отображаться созданная станица.

Поскольку конфигурация сервера по умолчанию может меняться, данный способ служит скорее для демонстрации и не является указанием к действию. Для проектов рекомендуется использовать системы управления конфигурациями, такие как **Ansible**.

Список литературы

Основная

1. Карр Н. Великий переход: что готовит революция облачных технологий Электрон. дан. — М. : Манн, Иванов и Фербер, 2014.
2. Андреевский И.Л. Технологии облачных вычислений — СПб. : Санкт-Петербургский государственный экономический университет, 2018.
3. Савельев А.О. Введение в облачные решения Microsoft. Курс лекций. 2-е издание, исправленное. — М. : НОУ Интуит, 2016.
4. Бернс Б. Распределенные системы. Паттерны проектирования — СПб. : Питер, 2019.
5. Атчисон Ли. Масштабирование приложений. Выращивание сложных систем СПб.: Питер, 2018.
6. Ильин Д.Ю., Никульчев Е.В., Колясников П.В. Выбор технологических решений для разработки программного обеспечения распределенных информационных систем // Современные информационные технологии и ИТ-образование. 2018. Т. 14. № 2. С. 344–354
7. Ильин Д.Ю., Никульчев Е.В., Бубнов Г.Г., Матешук Е.О. Информационно-аналитический сервис формирования актуальных профессиональных компетенций на основе патентного анализа технологий и выделения профессиональных навыков в вакансиях работодателей // Прикаспийский журнал: управление и высокие технологии. 2017. № 2 (38). С. 71–88.

Дополнительная

1. Caballer M., Blanquer I., Moltó G., de Alfonso C. Dynamic management of virtual infrastructures // Journal of Grid Computing. 2015. V. 13. No. 1. P. 53–70. Doi: 10.1007/s10723-014-9296-5
2. Giannakopoulos I., Konstantinou I., Tsoumakos D., Koziris N. Cloud application deployment with transient failure recovery // Journal of

Cloud Computing. 2018. V. 7. No. 1. Art. no. 11. Doi: 10.1186/s13677-018-0112-9

3. Spanaki P., Sklavos N. Cloud Computing: Security Issues and Establishing Virtual Cloud Environment via Vagrant to Secure Cloud Hosts // Computer and Network Security Essentials. — Springer, 2018. P. 539–553. Doi: 10.1007/978-3-319-58424-9_31
4. Hashimoto M. Vagrant: Up and Running: Create and Manage Virtualized Development Environments – O'Reilly Media Inc, 2013.
5. Mouat A. Using Docker: Developing and Deploying Software with Containers – O'Reilly Media Inc, 2016.
6. Sammons G. Learning Vagrant: Fast programming guide – CreateSpace Independent Publishing Platform, 2016.
7. Peacock, M. Creating Development Environments with Vagrant – Packt Publishing Ltd, 2015.
8. Iuhasz G., Pop D., Dragan I. Architecture of a scalable platform for monitoring multiple big data frameworks // Scalable Computing: Practice and Experience. 2016. V. 17. No. 4. P. 313-321. Doi: 10.12694/scpe.v17i4.1203
9. Nikulchev E., Ilin D., Kolyasnikov P., Belov V., Zakharov I., Malykh S. Programming Technologies for the Development of Web-Based Platform for Digital Psychological Tools // International journal of advanced computer science and applications. 2018. V. 9. No. 8. P. 34-45. Doi: 10.14569/IJACSA.2018.090806
10. Kashyap S., Min C., Kim T. Opportunistic spinlocks: Achieving virtual machine scalability in the clouds // ACM SIGOPS Operating Systems Review. 2016. V. 50. No. 1. P. 9-16. Doi: 10.1145/2903267.2903271
11. Saikrishna P. S., Pasumarthy R., Bhatt N. P. Identification and multivariable gain-scheduling control for cloud computing systems // IEEE

Transactions on Control Systems Technology. 2017. V. 25. No. 3. P. 792–807. Doi: 10.1109/TCST.2016.2580659

12. Li J., Xue S., Zhang W., Qi Z. When i/o interrupt becomes system bottleneck: Efficiency and scalability enhancement for sr-iov network virtualization // IEEE Transactions on Cloud Computing. 2017. Early Access. P. 1 Doi: 10.1109/TCC.2017.2712686
13. Peinl R., Holzschuher F., Pfitzer F. Docker cluster management for the cloud-survey results and own solution // Journal of Grid Computing. 2016. V. 14. No. 2. P. 265–282. Doi: 10.1007/s10723-016-9366-y
14. Krieger, M. T., Torreno, O., Trelles, O., & Kranzlmüller, D. Krieger M. T. et al. Building an open source cloud environment with auto-scaling resources for executing bioinformatics and biomedical workflows //Future Generation Computer Systems. – 2017. – Т. 67. – С. 329–340. 10.1016/j.future.2016.02.008

Электронные ресурсы

1. Vagrant, 2019. [Электронный ресурс]. Режим доступа: https://www.vagrantup.com/ [Дата обращения 27.02.2019].
2. Docker, 2019. [Электронный ресурс]. Режим доступа: https://www.docker.com/ [Дата обращения 27.02.2019].
3. Vagrant WinNFSd — GitHub, 2019. [Электронный ресурс]. Режим доступа: https://github.com/winnfsd/vagrant-winnfsd [Дата обращения 29.02.2019].
4. Vagrant bindfs — GitHub, 2019. [Электронный ресурс]. Режим доступа: https://github.com/gael-ian/vagrant-bindfs [Дата обращения 29.02.2019].
5. Webpack, 2019. [Электронный ресурс]. Режим доступа: https://webpack.js.org/ [Дата обращения 27.02.2019].