МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное бюджетное образовательное учреждение высшего образования
«Тольяттинский государственный университет»

Институт математики, физики и информационных технологий
(наименование института полностью)

Кафедра «Прикладная математика и информатика»
(наименование)

09.04.03 Прикладная информатика
(код и наименование направления подготовки)

Управление корпоративными информационными процессами
(направленность (профиль))

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА (МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ)

на тему «Оптимизация инфраструктуры информационных систем на базе микросервисной архитектуры»

Обучающийся Л. О. Герасимов
(Инициалы Фамилия) (личная подпись)

Научный руководитель доцент, О. В. Аникина
(ученая степень (при наличии), ученое звание (при наличии), Инициалы Фамилия)

Тольятти 2023

Содержание

Введение.. 4
1 Анализ существующих работ по теме размещения микросервисов.............. 7
1.1 Подходы к планированию ресурсов кластера 7
1.2 Постановка проблемы 10
1.3 Обзор существующих решений задачи размещения 13
микросервисов.. 13
1.3.1 Tetris.. 14
1.3.2 Firmament 16
1.3.3 Kubernetes... 18
1.3.4 OpenShift... 21
1.4 Сравнение существующих решений.. 22
1.5 Предполагаемый способ решения проблемы 24
2 Планировщик ресурсов на основе алгоритма поиска k-разреза................... 29
2.1 Определение модели 30
2.2 Проблема наименьшего k-разреза.. 33
2.3 Алгоритм размещения.. 36
2.3.1 Разделение приложения.. 36
2.3.2 Бинарное разделение... 38
2.3.3 K-разделение.. 40
2.4 Эвристическая упаковка 42
2.5 Поиск размещения.. 44
3 Тестирование полученного планировщика 47
3.1 Проведение эксперимента 47
3.2 Сравнение показателей 49

3.3 Влияние значения порога α 53
3.4 Оценка издержек 55
4 Внедрение полученного планировщика на производство 58
4.1 Производство 58
4.2 Внедрение 59
4.3 Результаты внедрения планировщика 64
Заключение 66
Список используемых источников 68

Введение

Архитектура микросервисов – это новая тенденция, которая быстро набирает обороты при разработке приложений, поскольку она повышает гибкость за счет включения различных технологий, снижает сложность за счет использования легких и модульных служб, а также улучшает общую масштабируемость и отказоустойчивость системы.

Однако это также создает проблемы. При развертывании приложения на основе служб в облаках планировщик должен тщательно планировать каждую службу, которая может иметь различные потребности в ресурсах, на распределенных вычислительных кластерах. Обеспечение желаемой производительности сервисных приложений, особенно производительности сети между задействованными сервисами, становится все более важным.

При развертывании сервисного приложения в облаке необходимо учитывать несколько важных аспектов. Во-первых, сервисы, задействованные в приложении, часто имеют различные потребности в ресурсах, таких как ЦП, память и диск. Базовая машина должна обеспечивать достаточные ресурсы для запуска каждой службы и в то же время обеспечивать согласованные функции. Во-вторых, сервисы, задействованные в приложении, часто имеют потребности в трафике из-за передачи данных, что требует тщательной обработки. Плохая обработка запросов трафика может привести к серьезному снижению производительности, поскольку время отклика службы напрямую зависит от ситуации с трафиком. Учитывая потребности в трафике, интуитивно понятное решение состоит в том, чтобы разместить сервисы, которые требуют большого трафика, на одном компьютере, что может обеспечить внутримашинную связь и уменьшить межмашинный трафик. Чтобы достичь желаемой производительности приложения на основе микросервисов, планировщики кластеров должны тщательно размещать каждый сервис приложения с учетом требований к ресурсам и требованиям трафика. Для этого разработчикам ИС необходимо составлять конфигурацию ИС таким образом, чтобы планировщик выполнял поставленные условия.

Таким образом, научная проблема заключается в том, что современные алгоритмы распределения ресурсов не учитывают сетевое взаимодействие между сервисами, что влияет на скорость работы приложений с большим объемом внутреннего трафика и на скорость разработки и развертывания приложения.

Целью исследования является разработка алгоритма распределения ресурсов, в котором будет учитываться связь между сервисами по сети, и который позволит сократить время развертывания приложения.

Объектом исследования является процесс развертывания приложения на основе микросервисной архитектуры.

Предметом исследования является методы и алгоритмы, повышающие эффективность работы инфраструктуры микросервисных приложений.

Гипотеза исследования состоит в том, что процесс развертывания приложения будет более эффективным, если при распределении ресурсов учитывать трафик между сервисами.

Для достижения поставленной цели были сформулированы и решены следующие задачи:

- анализ современной научной литературы, связанной с подходами и алгоритмами размещения, технологиями их реализации;
- выявление достоинств и недостатков существующих подходов;
- спроектировать новый алгоритм, подход или методику, в основе которого лежат методы, позволяющие повысить эффективность инфраструктуры;
- провести оценку эффективности внедренного решения.

Научная новизна состоит в разработке нового алгоритма для размещения сервисов.

В процессе исследования были использованы следующие методы: методы системного анализа, экспертной оценки, методы объектно-ориентированного анализа и проектирования.

На защиту выносятся:

– результаты анализа способов размещения сервисов;
– спроектированный алгоритм, который позволяет улучшить использование ресурсов;
– результаты оценки эффективности спроектированного алгоритма.

Диссертация состоит из введения, четырех разделов, заключения и списка литературы. Объем диссертации составляет 70 страниц и содержит 22 рисунка, 4 таблицы, список литературы включает 29 наименований источников зарубежных и отечественных авторов.

Во введении обосновывается актуальность выбранной темы исследования, определяется объект, предмет и цель исследования, выдвигается гипотеза и формулируются задачи работы, рассматриваются научная новизна исследования.

В первом разделе диссертации описываются решения задачи размещения сервисов, достоинства и недостатки, обнаруженные различными авторами научных работ, а также описывается найденная проблема, которая не рассматривалась авторами.

Во втором разделе выполнен анализ существующих решений для размещения сервисов.

В третьем разделе описывается новый подход к решению рассматриваемой задачи, формируется математическая модель и алгоритм решения задачи. Предложенная модель реализовываться с помощью выбранных технологий, приводится сравнение результатов с другими системами.

В четвертом разделе рассматривается внедрение нового алгоритма в существующую систему управления контейнерами и проводится оценка эффективности алгоритма.

В заключении представлены основные результаты поставленных задач и полученные выводы.

1 Анализ существующих работ по теме размещения микросервисов

1.1 Подходы к планированию ресурсов кластера

Поскольку микросервисная архитектура становится все более популярной, чем когда-либо, разработчики намерены преобразовать традиционные монолитные приложения в приложения на основе сервисов (составленные из ряда сервисов). Для развертывания сервисного приложения в облаке, помимо требований к ресурсам каждого сервиса, требования трафика между сервисами для совместной работы имеют решающее значение для общей производительности. Плохая обработка требований трафика может привести к серьезному снижению производительности, например к большому времени отклика и нестабильному соединению. Однако текущие планировщики кластеров не могут разместить сервисы на наилучшем возможном компьютере, поскольку они учитывают только ограничения ресурсов, но игнорируют требования трафика между службами. Для решения этой проблемы предлагается новый подход к оптимизации размещения сервисных приложений в облаках. Этот подход сначала разделяет приложение на несколько частей, сводя к минимуму общий трафик между различными частями, а затем тщательно упаковывает различные части в машины с учетом их требований к ресурсам и требованиям трафика. В будущем планируется реализовать прототип планировщика и оцениваем его с помощью обширных экспериментов на тестовых кластерах.

Архитектура микросервисов – это новая тенденция, которая быстро набирает обороты при разработке приложений, поскольку она повышает гибкость за счет включения различных технологий, снижает сложность за счет использования легких и модульных служб, а также улучшает общую масштабируемость и отказоустойчивость системы. В определении Мартина Фаулера [1] архитектурный стиль микросервисов – это подход к разработке одного приложения как набора небольших сервисов, каждый из которых

работает в собственном процессе и взаимодействует с легковесными механизмы (например, API ресурсов HTTP). Таким образом, приложение состоит из ряда служб (приложение на основе служб), которые работают согласованно для обеспечения сложных функций. Благодаря преимуществам архитектуры микросервисов многие разработчики намерены превратить традиционные монолитные приложения в приложения на основе служб. Например, приложение для онлайн-покупок можно в основном разделить на сервис продукта, сервис корзины и сервис заказа, что может значительно повысить производительность, маневренность и отказоустойчивость приложения. Однако это также создает проблемы. При развертывании приложения на основе служб в облаках планировщик должен тщательно планировать каждую службу, которая может иметь различные потребности в ресурсах, на распределенных вычислительных кластерах. Кроме того, сетевая связь между различными службами должна обрабатываться хорошо, так как условия связи значительно влияют на качество обслуживания (например, время отклика услуги). Обеспечение желаемой производительности сервисных приложений, особенно производительности сети между задействованными сервисами, становится все более важным.

Как правило, сервисные приложения включают в себя множество распределенных и сложных сервисов, которые обычно требуют больше вычислительных ресурсов, чем возможности отдельной машины. Следовательно, кластер сетевых машин или платформ облачных вычислений (например, Amazon EC2, Microsoft Azure или Google Cloud Platform) обычно используется для запуска сервисных приложений. Что еще более важно, контейнеры становятся прорывной технологией для эффективной инкапсуляции контекстов времени выполнения программных компонентов и служб, что значительно улучшает переносимость и эффективность развертывания приложений в облаках. При развертывании сервисного приложения в облаке необходимо учитывать несколько важных аспектов. Во-первых, сервисы, задействованные в приложении, часто имеют различные

потребности в ресурсах, таких как ЦП, память и диск. Базовая машина должна обеспечивать достаточные ресурсы для запуска каждой службы и в то же время обеспечивать согласованные функции. Эффективное распределение ресурсов для каждой службы затруднено, в то время как это становится еще более сложной задачей, когда кластер состоит из разнородных машин. Во-вторых, сервисы, задействованные в приложении, часто имеют потребности в трафике из-за передачи данных, что требует тщательной обработки. Плохая обработка запросов трафика может привести к серьезному снижению производительности, поскольку время отклика службы напрямую зависит от ситуации с трафиком. Учитывая потребности в трафике, интуитивно понятное решение состоит в том, чтобы разместить сервисы, которые требуют большого трафика, на одном компьютере, что может обеспечить внутримашинную связь и уменьшить межмашинный трафик. Однако все такие службы не могут быть размещены на одной машине из-за ограниченных ресурсов. Следовательно, размещение сервисных приложений в облаках довольно сложно. Чтобы достичь желаемой производительности приложения на основе служб, планировщики кластеров должны тщательно размещать каждую службу приложения с учетом требований к ресурсам и требованиям трафика. Для этого разработчикам ИС необходимо составлять конфигурацию ИС таким образом, чтобы планировщик выполнял поставленные условия.

Последние методы планирования кластера в основном сосредоточены на эффективности использования ресурсов кластера или времени завершения задания пакетных рабочих нагрузок [1,2,3]. Например, Tetris [4], планировщик кластера с несколькими ресурсами, адаптирует эвристику для задачи многомерной упаковки в контейнеры, чтобы эффективно упаковать задачи в кластере с несколькими ресурсами. Firmament [5], централизованный планировщик кластеров, может принимать высококачественные решения о размещении в крупномасштабных кластерах с помощью оптимизации с минимальной стоимостью и максимальным потоком. К сожалению, эти решения столкнутся с трудностями при работе с сервисными приложениями,

поскольку они игнорируют требования трафика при принятии решений о размещении. Некоторые другие исследовательские работы [6,7] сосредоточены на проблеме размещения составного программного обеспечения как услуги (SaaS – Software as a Service), которые пытаются минимизировать общее время выполнения составного SaaS. Однако они сосредоточены только на наборе предопределенных компонентов службы для размещения приложения. Для планирования с учетом трафика предлагаются соответствующие исследовательские решения [8,9] для решения проблемы размещения виртуальных машин (ВМ), которые направлены на оптимизацию использования сетевых ресурсов в кластере. Однако эти решения полагаются на определенную топологию сети, в то время как большинство существующих планировщиков кластеров не зависят от топологии сети. В частности, трудно получить информацию о топологии сети при развертывании сервисного приложения в виртуальной инфраструктуре.

1.2 Постановка проблемы

Для достижения желаемой производительности приложений на основе сервисов планировщик должен учитывать не только потребность сервисов в нескольких ресурсах, но и ситуацию с трафиком между сервисами. Поскольку сервисы, в особенности сервисы с интенсивным использованием данных, часто нуждаются в частой передаче данных, производительность сети напрямую влияет на общую производительность. Учитывая динамику сети, размещение различных сервисов приложения имеет решающее значение для поддержания общей производительности, особенно когда в кластере возникает непредвиденная задержка или перегрузка сети. Учитывая ситуацию с трафиком, наиболее интуитивным решением является размещение сервисов с высокой скоростью трафика на одном компьютере, чтобы совместно расположенные сервисы могли использовать петлевой интерфейс для получения высокой производительности сети, не потребляя фактические

сетевые ресурсы компьютерного кластера. Однако такие сервисы не могут быть размещены на одном компьютере из-за ограниченных ресурсов. Таким образом, с ограничениями ресурсов мы пытаемся найти решение для размещения, чтобы минимизировать общий трафик между сервисами, размещенными на разных машинах (межмашинный трафик), при этом удовлетворяя потребности сервисов в нескольких ресурсах. Чтобы цель этой работы могла быть достигнута, её можно сформулировать следующим образом:

$$min \sum_{i=1}^{N} \sum_{j=1}^{N} \sum_{p=1}^{M} \sum_{\substack{q=1 \\ q \neq p}}^{M} t_{ij} * x_{ip} * x_{jq} \quad (1)$$

при условии, что

$$\sum_{j=1}^{M} x_{ij} = 1 \ (\forall i \in \{1,2,\dots,N\}), \quad (2)$$

$$\sum_{j=1}^{M} x_{ij} * d_i^k \leq v_j^k \ (\forall j \in \{1,2,\dots,M\}, \forall k \in \{1,2,\dots,R\}), \quad (3)$$

$$x_{ij} \in \{0,1\} \ (\forall i \in \{1,2,\dots,N\}, \forall j \in \{1,2,\dots,M\}) . \quad (4)$$

Уравнение (2) гарантирует, что каждый сервис будет размещен на машине. Уравнение (3) гарантирует, что потребности машины в ресурсах не превышают ее ресурсных возможностей. Уравнение (1) выражает цель данной работы.

Также стоит отметить, что точкой оптимизации может являться процесс развертывания приложения. На данный момент его можно представить в виде диаграммы, представленной на рисунке 1.

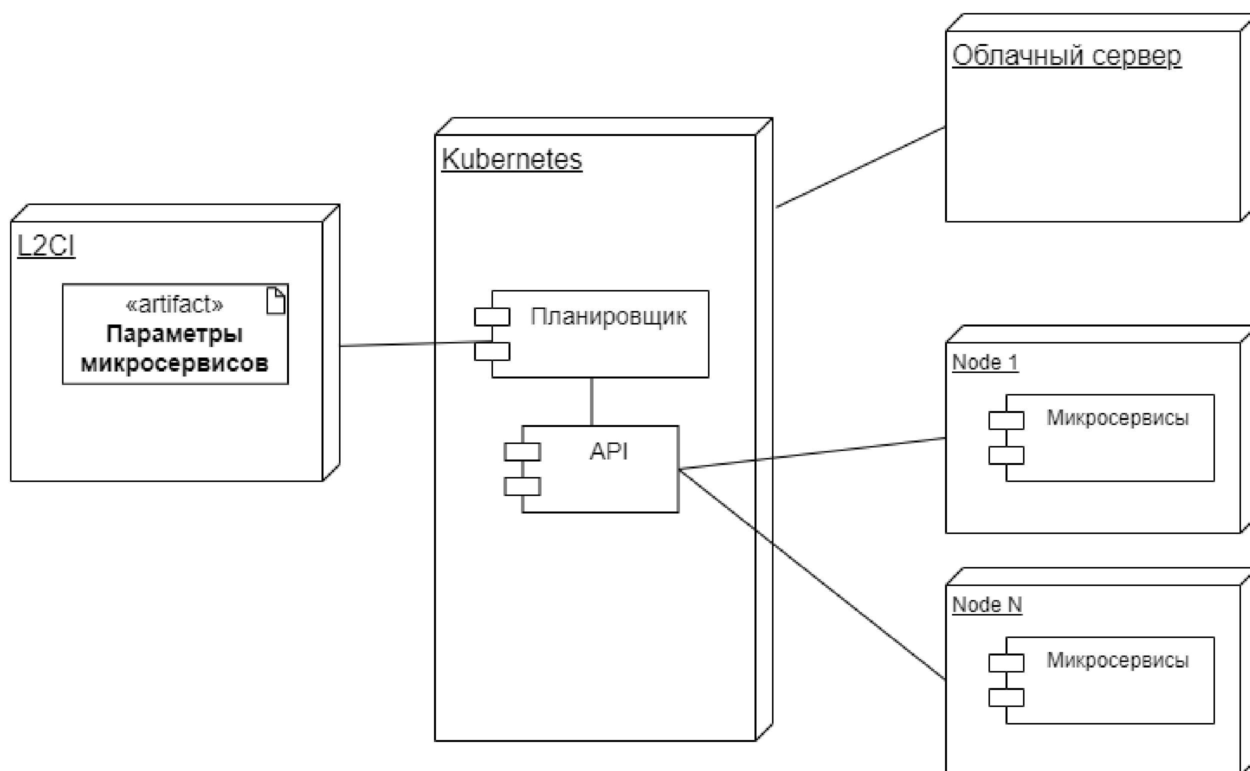

Рисунок 1 – диаграмма развертывания микросервисного приложения

На диаграмме видно, что планировщик ресурсов зависит от параметров микросервисов. В этих параметрах указываются как служебные параметры, необходимые для настройки бизнес-логики сервисов, так и параметры, определяющие потребности в ресурсах, такие как необходимый объем пространства на диске, объем используемой оперативной памяти и количество используемых ядер процессора. Зачастую эти параметры определяются разработчиками вручную, на что они тратят около двух человеко-часов на каждый сервис.

Далее эта конфигурация отправляется планировщику ресурсов, а тот, в свою очередь, используя интерфейс платформы распределяет микросервисы по узлам кластера.

При таком подходе очевидно, что разработчики тратят много времени на составление конфигурации приложения и на сборку даже несмотря на использование дополнительных инструментов сборки.

1.3 Обзор существующих решений задачи размещения микросервисов

Сегодня многие приложения работают на больших кластерах центров обработки данных. Эти кластеры совместно используются приложениями множества организаций и пользователей. Пользователи выполняют задачи, каждая из которых состоит из одной или нескольких параллельных задач.

Планировщик кластера решает, как разместить эти задачи на машинах кластера, где они создаются как процессы, контейнеры или виртуальные машины.

Лучшее размещение задач ведущим планировщиком кластера – это более высокая загрузка машины, более короткое время выполнения пакетных заданий, улучшенная балансировка нагрузки, более предсказуемая производительность приложений и повышенная отказоустойчивость. Добиться высокого качества размещения задач сложно: это требует алгоритмически сложной оптимизации в нескольких измерениях. Эта цель противоречит потребности в низкой задержке размещения, времени, которое требуется планировщику для запуска новой задачи. Требуется низкая задержка размещения как для удовлетворения ожиданий пользователей, так и для предотвращения простаивания кластера ресурсов, пока есть ожидающие задачи. Более короткая партия время выполнения задач и увеличивающийся масштаб кластера затрудняют достижение обеих противоречивых целей.

Таким образом, текущие планировщики выбирают один из этих критериев для определения приоритета.

На сегодняшний день существуют три различные архитектуры кластерного планировщика. Во-первых, это централизованные планировщики, которые используют сложные алгоритмы для поиска качественных мест размещения, но имеют латентность в секунды или минуты, что является недопустимым, когда требование к отказоустойчивости системы является основным критерием. Второй тип – распределенные планировщики, которые

используют простые алгоритмы, которые позволяют добиться высокой пропускной способности, параллельного размещения задач с малой задержкой в масштабе. Однако их несогласованные решения, основанные на частичном, устаревшем состоянии, могут привести к неудовлетворительному результату места размещения, что может негативно сказаться на стабильности системы в целом. В-третьих, гибридные планировщики, разделяющие рабочую нагрузку через централизованный и распределенный компонент. Они используют сложные алгоритмы для длительных задач, но полагаются на распределенное размещение для коротких задач.

В данном разделе будут рассмотрены планировщики перечисленных типов, а также будет проведен сравнительный анализ их характеристик.

1.3.1 Tetris

Tetris – это планировщик кластера с несколькими ресурсами, который упаковывает задачи на машины в зависимости от их требований ко всем типам ресурсов. Это позволяет избежать фрагментации ресурсов, а также чрезмерного выделения ресурсов, которые не выделены явно, что является недостатками текущих планировщиков.

Достижение хорошей эффективности упаковки увеличивает срок службы, но не обязательно ускоряет выполнение отдельных заданий. Предпочтительное выделение ресурсов для задания с наименьшим оставшимся временем (SRTF – Shortest Remaining Time First) сведет к минимуму среднее время выполнения задания. Чем меньше задач остается в задании, тем меньше его оставшаяся продолжительность. Однако задачи могут иметь разную продолжительность. Хуже того, задачи могут иметь разные требования к разным ресурсам. Основная работа улучшает среднее время выполнения с наименьшей нехваткой времени. Тем не менее, работа с наименьшим количеством оставшейся работы может не иметь задач, которые хорошо упаковываются, и, следовательно, строгий порядок рабочего времени может замедлить работу всех. Tetris использует многоресурсную версию SRTF для заданий, которые представляют собой DAG зависимых задач, и сочетает в

себе обе эвристики – наилучшую упаковку и кратчайшее оставшееся время выполнения задания – для сокращения среднего времени выполнения задания. Данный планировщик объединяет две метрики – оставшееся использование ресурсов и показатель выравнивания – используя взвешенную сумму.

Использование Tetris для выбора лучшей для упаковки задачи из этого ограниченного подмножества повысит производительность без ущерба для справедливости размещения. Tetris предлагает естественное обобщение: используйте эвристику упаковки и времени выполнения задачи, чтобы выбрать лучшую задачу из (1 − f) доли запущенных задач (или групп), которые наиболее далеки от своей справедливой доли. Выбор f = 0 приводит к лучшему сроку изготовления и времени завершения, тогда как f → 1 обеспечивает строгую справедливость. Для f ∈ [0.25, 0.5] Tetris обеспечивает почти лучшую производительность, а влияние на рабочие места из-за возникающей несправедливости пренебрежимо мало.

Среди особенностей Tetris можно выделить следующие достижения.

- Tetris определяет важность планирования и упаковки всех соответствующих ресурсов. В противном случае ресурсы фрагментируются и могут быть перераспределены, что значительно влияет на производительность.
- Tetris представляет эвристическое решение сложной для APX проблемы упаковки задач по нескольким ресурсам в планировщиках кластера.
- Tetris объединяет эвристики, повышающие эффективность упаковки с теми, которые снижают среднее время выполнения работы.
- Парето-эффективное справедливое распределение не дает наилучших результатов. В то время как производительность и справедливость зачастую недостижимы вместе, Tetris показывает, что в контексте кластерных планировщиков можно добиться гораздо большей производительности с помощью небольшой несправедливости, и раскрывает компромисс с помощью ручки.

Tetris был создан как первый кластерный планировщик с параллельными данными, в котором явным образом учитывается упаковка нескольких ресурсов, как модификация планировщика в YARN. Tetris оценивает потребности в задачах по предыдущим выполнениям той же работы и по завершенным задачам текущей работы. Tetris использует средство отслеживания ресурсов для независимого мониторинга использования машин, что позволяет ему учитывать любые ошибки в оценке спроса и корректировать планирование с учетом непредвиденных горячих точек и неправильно работающих узлов.

Основные моменты оценки разработчиков Tetris заключаются в следующем.

Tetris улучшает время создания кластера и среднее время выполнения задач примерно на 30 % в развертывании и до 40 % при воспроизведении трассировок Facebook в моделировании. По оценкам авторов, эти выигрыши составляют 80% от простой верхней границы. Выигрыш достигается за счет предотвращения фрагментации и избыточного распределения ресурсов, а также за счет использования дополнительных требований для лучшей упаковки.

Tetris дополняет вышеописанный выигрыш в эффективности лишь с небольшой потерей честности. Эти выгоды используют ручку справедливости f = 0,25, и менее 6% заданий задерживаются не более чем на 10% по сравнению со справедливым распределением. Установка f = 0, наиболее эффективный и несправедливый график, увеличивает срок изготовления до 50%, но вытекающая из этого несправедливость приводит к задержке большего количества заданий. Удивительно, но f = 1, наиболее справедливый график, также обеспечивает значительное повышение эффективности.

1.3.2 Firmament

Firmament обобщает Quincy, который представляет задачу планирования как оптимизацию min-cost max-flow (MCMF) по графу и непрерывно перепланирует всю рабочую нагрузку. Оригинальный алгоритм Quincy MCMF

приводит к задержке размещения задач в несколько минут в большом кластере. Однако Firmament в общем случае обеспечивает задержку размещения в сотни миллисекунд и достигает того же качества размещения, что и Quincy.

Авторами данного планировщика были изучены несколько алгоритмов оптимизации MCMF и их производительность. Они обнаружили, что релаксация, кажущийся неэффективным алгоритм MCMF, превосходит другие алгоритмы на графах, генерируемых задачей планирования. Однако релаксация может быть медленной в критических крайних случаях, поэтому мы исследовали три метода уменьшения задержки размещения Firmament в разных алгоритмах:

- Раннее прекращение работы алгоритмов MCMF для поиска приближенных решений приводит к недопустимо плохим и нестабильным размещениям, и мы отвергаем эту идею.
- Инкрементная повторная оптимизация улучшает время выполнения исходного алгоритма MCMF Quincy (масштабирование стоимости) и делает его приемлемым запасным вариантом.
- Эвристики для конкретных задач помогают некоторым алгоритмам MCMF работать быстрее на графах определенной структуры.

Разработчики объединили эти алгоритмические идеи с несколькими методами уровня реализации, чтобы еще больше уменьшить задержку размещения Firmament. Firmament запускает два алгоритма MCMF одновременно, чтобы избежать замедления в крайних случаях; он реализует эффективный алгоритм обновления графа для обработки изменений состояния кластера; и он быстро извлекает места размещения задач из рассчитанного оптимального потока.

Эксперименты с трассировкой рабочей нагрузки Google для кластера из 12 500 машин показывают, что Firmament уменьшает задержку размещения в 20 раз по сравнению с Quincy, предшествующим централизованным планировщиком, использующим ту же оптимизацию MCMF. Более того,

несмотря на то, что Firmament является централизованным, он соответствует задержке размещения распределенных планировщиков для рабочих нагрузок коротких задач. Наконец, Firmament превосходит качество размещения четырех широко используемых централизованных и распределенных планировщиков в реальном кластере и, следовательно, улучшает время отклика на пакетные задачи в 6 раз.

Также они решили разработать Firmament как планировщик на основе потоков по трем причинам. Во-первых, планирование на основе потока учитывает всю рабочую нагрузку, что позволяет нам поддерживать повторное планирование и вытеснение приоритетов. Во-вторых, планирование на основе потока обеспечивает высокое качество размещения и, следовательно, малое время отклика на задачу. В-третьих, в качестве пакетного подхода планирование на основе потока хорошо амортизирует работу по многим задачам и решениям о размещении и, следовательно, обеспечивает высокую пропускную способность задач, хотя и с высокой задержкой размещения, которую стремятся улучшить.

1.3.3 Kubernetes

Kubernetes автоматически предоставляет общий доступ к ресурсам физического центра обработки данных и управляет ими. У каждого есть планировщик, который отвечает за размещение задач на машинах. На рисунке 4 показан жизненный цикл задачи в диспетчере кластера: после того, как пользователь отправляет задачу, он ждет, пока планировщик поместит ее на машину, где она впоследствии запустится.

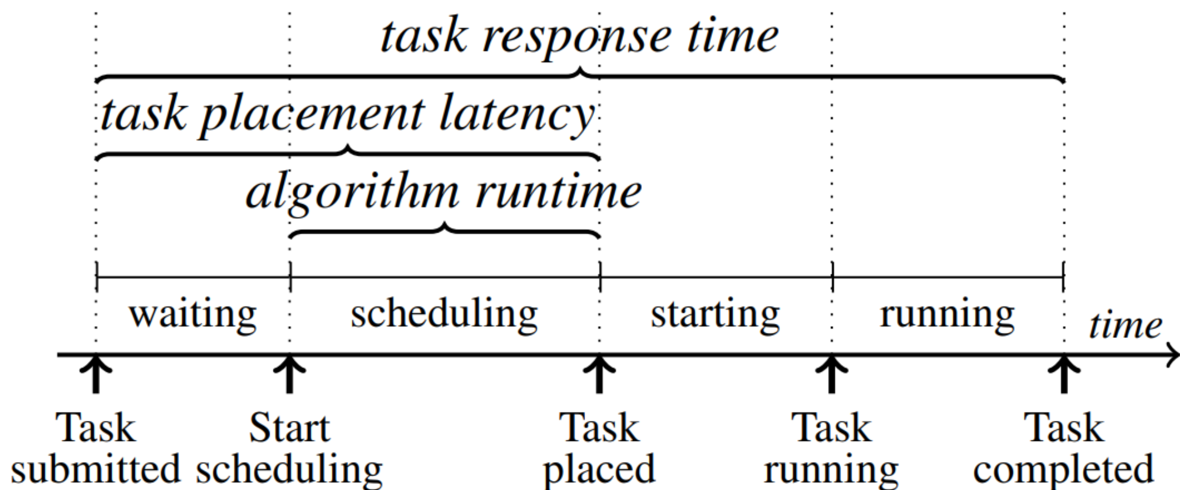

Рисунок 4 – Жизненный цикл задач в кластере

Время между отправкой и размещением задачи – это задержка размещения задачи, а общее время между отправкой задачи и ее завершением – время ответа задачи. Время, в течение которого задача находится в активном планировании, является временем выполнения алгоритма планировщика. Для каждой задачи алгоритм планирования обычно сначала выполняет проверку выполнимости, чтобы определить подходящие машины, затем оценивает их в соответствии с порядком предпочтения и, наконец, помещает задачу на машину с лучшими оценками. Оценка, то есть оценка различных вариантов размещения для задачи, может быть дорогостоящей. Google Borg, например, полагается на несколько оптимизаций пакетной обработки, кэширования и аппроксимации, чтобы сохранить управляемость оценки. Высокое качество размещения увеличивает загрузку кластера и позволяет избежать снижения производительности из-за чрезмерной фиксации. Напротив, низкое качество размещения увеличивает время отклика задачи (для пакетных задач) или снижает производительность на уровне приложения (для долго работающих служб).

Kube-scheduler выбирает оптимальный узел для запуска только что созданных или еще не запланированных (незапланированных) подов.

Поскольку контейнеры в модулях – и сами модули – могут иметь разные требования, планировщик отфильтровывает любые узлы, которые не соответствуют конкретным потребностям модуля в планировании. Кроме того, API позволяет указать узел для пода при его создании, но это необычно и делается только в особых случаях .

В кластере узлы, отвечающие требованиям планирования для пода, называются возможными узлами. Если ни один из узлов не подходит, модуль остается незапланированным до тех пор, пока планировщик не сможет его разместить.

Планировщик находит подходящие узлы для пода, а затем запускает набор функций для оценки подходящих узлов и выбирает узел с наибольшим количеством очков среди подходящих для запуска пода. Затем планировщик уведомляет сервер API об этом решении в процессе, называемом привязкой.

Факторы, которые необходимо учитывать при планировании, включают индивидуальные и коллективные требования к ресурсам, аппаратные/программные/политические ограничения, спецификации привязки и антипривязки, локализацию данных, интерференцию между рабочими нагрузками и т.д.

На высоком уровне детализации используемый здесь алгоритм можно описать так.

Шаг фильтрации находит набор узлов, на которых возможно запланировать под. Например, фильтр PodFitsResources проверяет, имеет ли узел-кандидат достаточно доступных ресурсов для удовлетворения конкретных запросов ресурсов пода. После этого шага список узлов содержит любые подходящие узлы; часто их будет больше одного. Если список пуст, этот Pod (пока) не может быть запланирован.

На этапе оценки планировщик ранжирует оставшиеся узлы, чтобы выбрать наиболее подходящее размещение модуля. Планировщик присваивает оценку каждому узлу, пережившему фильтрацию, на основе активных правил оценки.

Наконец, kube-scheduler назначает Pod узлу с наивысшим рейтингом. Если имеется более одного узла с одинаковыми оценками, kube-scheduler выбирает один из них случайным образом.

1.3.4 OpenShift

Стандартный планировщик OpenShift имеет следующие особенности. Конфигурация по умолчанию, поставляемая с OpenShift Container Platform, поддерживает общие концепции центров обработки данных зон и регионов с использованием меток узлов, правил сходства и правил анти-сходства. Он предназначен для гибкой настройки и адаптации к различным кластерам.

Алгоритм планировщика модулей OpenShift в основном следует трехэтапному процессу.

Первым этапом идет фильтрация узлов. Планировщик фильтрует список работающих узлов, оценивая каждый узел по набору предикатов, таких как доступность хост-портов или возможность планирования модуля для узла, испытывающего нехватку диска или памяти. Модуль также может запрашивать вычислительные ресурсы, такие как ЦП, память и хранилище. Узлы, у которых недостаточно свободных компьютерных ресурсов, не подходят. Кроме того, модуль может определить селектор узлов, соответствующий меткам в узлах кластера. Узлы, метки которых не совпадают, не подходят. Другая проверка фильтрации оценивает, есть ли в списке узлов какие-либо пороки, и если да, то есть ли у пода связанный допуск, который может принять пороки. Если модуль не может принять заражение узла, то этот узел не подходит. По умолчанию главные узлы включают taint node-role.kubernetes.io/master:NoSchedule. Модуль, который не имеет соответствующего допуска для этого заражения, не будет назначен на главный узел. Конечным результатом этапа фильтрации обычно является более короткий список узлов-кандидатов, которые имеют право запускать модуль. В некоторых случаях ни один из узлов не отфильтровывается, что означает, что модуль может работать на любом из узлов. В других случаях все узлы отфильтровываются, что означает, что модуль не может быть запланирован до

тех пор, пока не станет доступным узел с требуемыми предварительными условиями.

Далее следует приоритизация отфильтрованного списка узлов. Список узлов-кандидатов оценивается с использованием нескольких критериев приоритета, которые в сумме составляют взвешенную оценку. Узлы с более высокими значениями лучше подходят для запуска модуля. Среди критериев – правила сходства и анти-сходства. Узлы с более высоким сродством к поду имеют более высокий балл, а узлы с более высоким анти-сходством имеют более низкий балл. Обычно правила сходства используются для планирования связанных модулей так, чтобы они находились близко друг к другу по соображениям производительности. Примером может служить использование одной и той же сетевой магистрали для модулей, которые синхронизируются друг с другом. Обычное использование правил анти-сходства – планирование связанных модулей, которые не находятся слишком близко друг к другу, для обеспечения высокой доступности. Например, чтобы избежать планирования всех модулей из одного приложения на один и тот же узел.

Последний этап – выбор наиболее подходящего узла. Список узлов сортируется на основе приведенных выше показателей матрицы, и узел с наивысшим значением выбирается для размещения модуля. Если несколько узлов имеют одинаковый высокий балл, то один из них выбирается в циклическом режиме.

Планировщик является гибким и может быть настроен для сложных ситуаций планирования. Поды могут быть размещены с использованием правил сходства и анти-сходства как для подов, так и для узлов.

1.4 Сравнение существующих решений

Чтобы лучше понять различия между рассмотренными планировщиками, ниже приведена таблица (Таблица 1), тезисно описывающая преимущества и недостатки каждого планировщика.

Таблица 1 – Сравнение характеристик существующих планировщиков кластера

Планировщик	Достоинства	Недостатки
Tetris	Скорость размещения новых задач	Показывает плохие результаты при фокусировании на «справедливом» размещении при частичном или полном игнорировании задач ввода/вывода
Firmament	Скорость размещения в больших кластерах	Может показывать плохие результаты при пограничных случаях задачи размещения
Kubernetes	Гибкая конфигурация за счет использования правил сходства	Уступает по скорости другим планировщикам
OpenShift	Гибкая конфигурация за счет использования правил сходства	Уступает по скорости другим планировщикам

Из таблицы видно, что все эти планировщики представляют из себя различные подходы к распределению ресурсов, используя разные алгоритмы и архитектуры. Поэтому при выборе планировщика нужно опираться на свои потребности.

При этом ни один из рассмотренных выше планировщиков не учитывает сетевое взаимодействие сервисов при распределении ресурсов.

Также разработчики Firmament проводили собственное исследование и сравнение с другими планировщиками. Результаты их исследования представлены на рисунке 5.

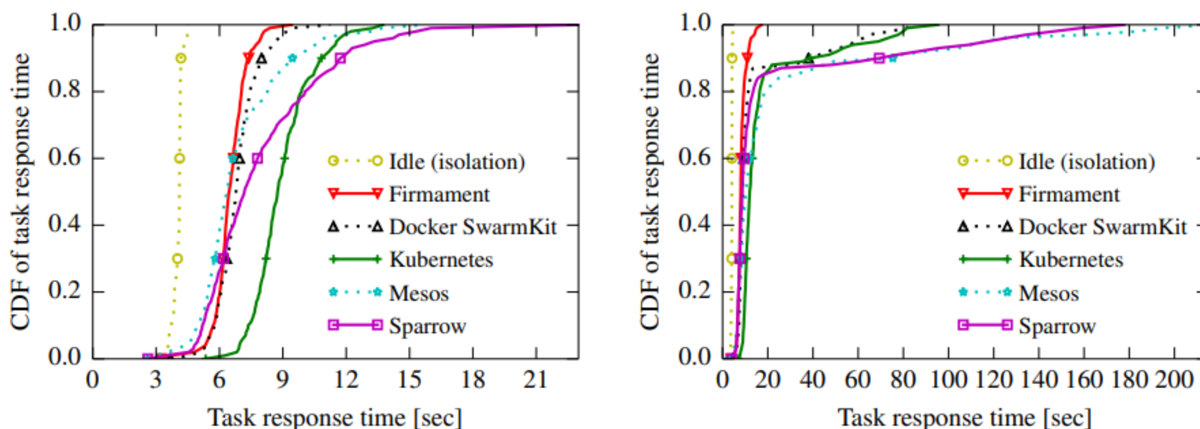

Рисунок 5 – Сравнение скорости работы различных планировщиков кластеров

По графику видно, что один из лучших результатов принадлежит Firmament, но в то же время интсрумент Docker Swarm, поставляемый в стандартной сборке Docker, уступает ему лишь немного. Из этого можно сделать вывод, что выбирать планировщик ресурсов кластера нужно исходя из своих потребностей и принимая во внимание предполагаемый размер устанавливаемого приложения, количество новых установок во время работы и иных показателей.

1.5 Предполагаемый способ решения проблемы

Поскольку приложение на основе служб обычно не может быть размещено на одном компьютере, во время развертывания необходимо эффективное разделение набора служб, задействованных в приложении. После разделения каждое подмножество сервисов должно быть упаковано в машину, что означает, что машина имеет достаточно ресурсов для запуска всех сервисов в подмножестве. Учитывая скорость трафика между различными сервисами, качество раздела имеет решающее значение для

производительности приложения. Чтобы решить эту проблему, сначала рассмотрим задачу о минимальном k-разрезе, чтобы понять ее сложность.

Минимальный разрез – это просто минимальный разрез k при k = 2. На рисунке 1 показан пример минимального разреза графа. На рисунке показаны 2 разреза, а пунктирная линия – это минимальный разрез графа, так как общий вес ребер, разрезанных штриховой линией, является минимумом всех разрезов. Учитывая приложение на основе служб, мы можем представить его в виде графа, где узлы представляют службы, а веса ребер представляют скорость трафика. В частности, скорость трафика от сервиса s_i к сервису s_j и скорость от сервиса s_j к сервису s_i представлены в виде двух ребер соответственно на графике. Следовательно, нахождение минимального k-разреза графа эквивалентно разделению приложения на k частей при сведении к минимуму общего трафика между различными частями. Однако для произвольного k задача о минимальном k-разрезе является NP-полной [10,11].

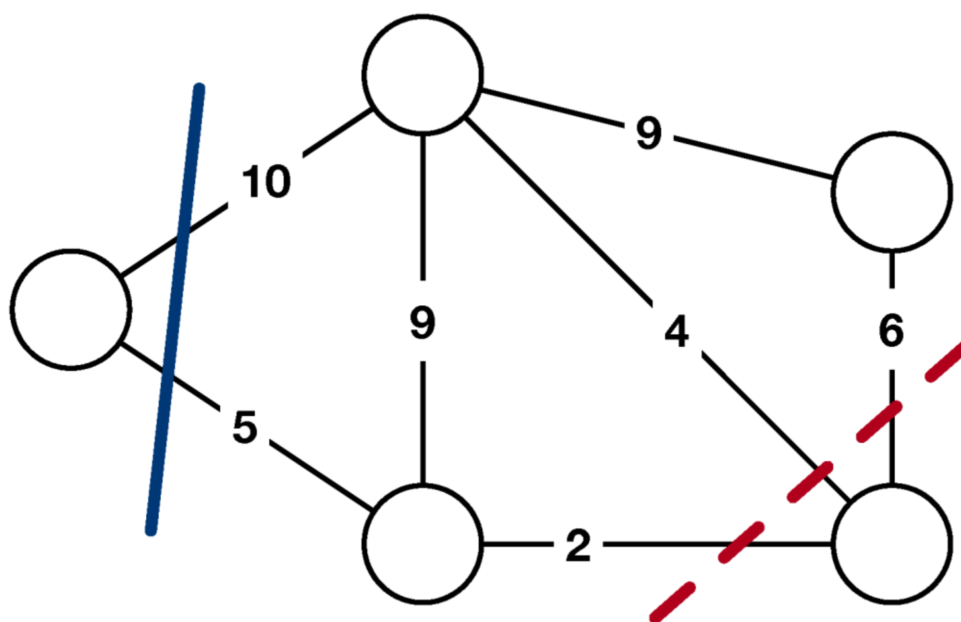

Рисунок 1 – Пример минимального разреза графа

Таким образом, задачу оптимизации можно свести к нахождению минимального k-разреза.

В отличие от разработки детерминированного алгоритма, алгоритм Каргера [12] обеспечивает эффективный рандомизированный подход для поиска минимального разреза графа. Основная идея алгоритма Каргера состоит в том, чтобы случайным образом выбрать ребро eu, v из графа с вероятностью, пропорциональной весу ребра eu, v, и объединить узел u и узел v в одно (так называемое сжатие ребер). Чтобы найти минимальный разрез, алгоритм итеративно сжимает края, которые выбираются случайным образом, пока не останется два узла. Затем алгоритм выводит оставшиеся ребра. Псевдокод показан на рисунке 2.

Input: $G = (V, E)$
Output: A cut of G
1 **while** $|V| > 2$ **do**
2 | choose an edge $e_{u,v}$ with probability proportional to its weight;
3 | $G \leftarrow G/e_{u,v}$; // contract edge $e_{u,v}$
4 **end**
5 **return** the cut in G;

Рисунок 2 – Псевдокод алгоритма сжатия

На рисунке 3 показан пример процесса алгоритма сжатия (k = 2). Алгоритм итеративно объединяет два узла выбранного ребра, а все остальные ребра повторно подключаются к объединенному узлу. Для графа G = (V, E) с n = | V | узлов и m = | E | рёбер, Каргер [12] утверждает, что алгоритм сжатия возвращает минимальный разрез графа с вероятностью Ω (1 / n2).

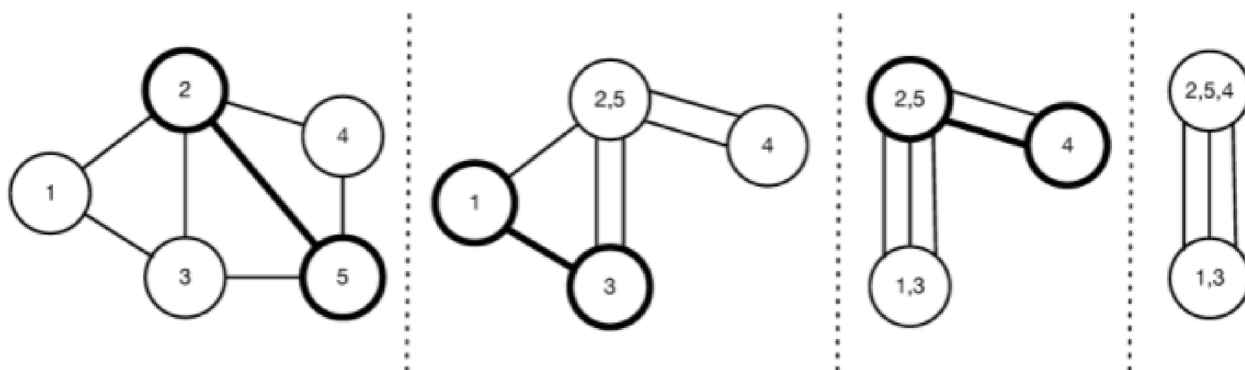

Рисунок 3 – Пример процесса алгоритма сжатия

Следовательно, если мы выполним алгоритм сокращения независимо n2logn раз, мы сможем найти минимальный разрез с высокой вероятностью; если мы не получим минимального разреза, вероятность будет меньше Ω (1 / n). Для минимального k-сокращения алгоритм сжатия в основном тот же, за исключением того, что он завершается, когда остаются k узлов (измените | V |> 2 на | V |> k в алгоритме 1) и возвращает все ребра, оставшиеся в графе G. Точно так же алгоритм сжатия возвращает минимальный k-разрез графа с вероятностью Ω (1 / n2k − 2). Если мы выполним алгоритм независимо n2k − 2logn раз, мы сможем получить минимальный k-разрез с большой вероятностью. Что касается временной сложности, алгоритм сжатия может быть реализован для работы за строго полиномиальное время O (mlog2n) [12].

2.3 Выводы по разделу

В результате выполненной работы были изучены существующие подходы к оптимизации микросервисных приложений, был проверен анализ этих решений, показавший, что при планировании ресурсов почти ни один планировщик не учитывает трафик между микросервисами. Это позволяет сделать вывод о том, что рассматриваемая тема исследования является актуальной.

Для дальнейшего исследования проблемы предлагается рассмотреть существующие алгоритмы нахождения минимального k-среза в качестве решения задачи. Один из таких алгоритмов – алгоритм Каргера, основанный на подходе с рандомизированным выбором ребер в графе.

Далее предполагается выбрать наиболее подходящий алгоритм и на его основе построить алгоритм размещения микросервисов с учетом их сетевого взаимодействия между собой. Затем рассмотреть возможные варианты конфигурации микросервисов, которые позволят применить полученный алгоритм при развертывании приложения в облачной платформе.

Были изучены существующие планировщики кластеров, был проведен сравнительный анализ этих решений, показавший, что при планировании ресурсов почти ни один планировщик не учитывает трафик между микросервисами.

Для дальнейшего исследования проблемы предлагается рассмотреть существующие алгоритмы нахождения минимального k-среза в качестве решения задачи. Один из таких алгоритмов – алгоритм Каргера, основанный на подходе с рандомизированным выбором ребер в графе.

Далее предполагается выбрать наиболее подходящий алгоритм и на его основе построить алгоритм размещения микросервисов с учетом их сетевого взаимодействия между собой. Затем рассмотреть возможные варианты конфигурации микросервисов, которые позволят применить полученный алгоритм при развертывании приложения в облачной платформе.

2 Планировщик ресурсов на основе алгоритма поиска k-разреза

Сегодня многие приложения работают на больших кластерах центров обработки данных. Эти кластеры совместно используются приложениями множества организаций и пользователей. Пользователи выполняют задачи, каждая из которых состоит из одной или нескольких параллельных задач.

Планировщик кластера решает, как разместить эти задачи на машинах кластера, где они создаются как процессы, контейнеры или виртуальные машины.

Лучшее размещение задач ведущим планировщиком кластера – это более высокая загрузка машины, более короткое время выполнения пакетных заданий, улучшенная балансировка нагрузки, более предсказуемая производительность приложений и повышенная отказоустойчивость. Добиться высокого качества размещения задач сложно: это требует алгоритмически сложной оптимизации в нескольких измерениях. Эта цель противоречит потребности в низкой задержке размещения, времени, которое требуется планировщику для запуска новой задачи. Требуется низкая задержка размещения как для удовлетворения ожиданий пользователей, так и для предотвращения простаивания кластера ресурсов, пока есть ожидающие задачи. Более короткая партия время выполнения задач и увеличивающийся масштаб кластера затрудняют достижение обеих противоречивых целей.

Таким образом, текущие планировщики выбирают один из этих критериев для определения приоритета.

На сегодняшний день существуют три различные архитектуры кластерного планировщика. Во-первых, это централизованные планировщики используют сложные алгоритмы для поиска качественных мест размещения, но имеют латентность в секунды или минуты. Второй тип – распределенные планировщики, которые используют простые алгоритмы, которые позволяют добиться высокой пропускной способности, параллельного размещения задач с малой задержкой в масштабе. Однако их несогласованные решения,

основанные на частичном, устаревшем состоянии, могут привести к неудовлетворительному результату места размещения. В-третьих, гибридные планировщики, разделяющие рабочую нагрузку через централизованный и распределенный компонент. Они используют сложные алгоритмы для длительных задач, но полагаются на распределенное размещение для коротких задач.

В данном разделе будут рассмотрены планировщики перечисленных типов, а также будет проведен сравнительный анализ их характеристик.

2.1 Определение модели

Для того, чтобы выполнить реализацию алгоритма нахождения минимального k-среза для распределения сервисов на кластере, нужно определить, какими входными данными должен располагать этот процесс.

Во-первых, должно быть известно о машинах в кластере и их ресурсах, так как каждая из них обладает своим набором ресурсов, таких как CPU, оперативная память, дисковое пространство и т.д. Также должно быть известно количество машин и количество доступных ресурсов на каждой из них.

Во-вторых, при использовании моделей IaaS (Infrastructure as a Service – инфраструктура как сервис) и CaaS (Container as a Service – контейнер как сервис) пользователь может установить потребность в ресурсах (комбинацию CPU, оперативной памяти, дискового пространства) для виртуальной машины, либо контейнера при составлении конфигурации развертывания. Поэтому потребности в ресурсах известны при получении запроса на развертывание. Приложение, построенное на базе микросервисной архитектуры, зачастую состоит из набора сервисов со своими уникальными наборами требований к ресурсам. Поэтому следующие параметры необходимы для работы: набор сервисов, количество сервисов, набор требований к ресурсам для каждого сервиса.

В-третьих, так как в данной работе рассматривается размещение сервисов с учетом траффика между ними, требуется знать, как сервисы взаимодействуют друг с другом по сети – то есть связи между сервисами и частота такого взаимодействия.

Таким образом все параметры можно свести в одну таблицу:

Таблица 2 – Параметры, необходимые для нахождения размещения сервисов

Параметр	**Описание**
$\mathcal{M}$	Набор машин в кластере: $\mathcal{M}= \{m_1, m_2, \dots, m_M\}$
M	Количество машин в кластере: M = | $\mathcal{M}$|
$\mathcal{R}$	Набор типов ресурсов: $\mathcal{R} = \{r_1, r_2, \dots, r_R\}$
R	Количество типов ресурсов: R = | $\mathcal{R}$ |
V_i	Вектор, описывающий доступные ресурсы машины m_i: $V_i = (v_i^1, v_i^2, \dots, v_i^R)$
v_i^j	Количество ресурса r_j, доступного на машине m_i
S	Приложение, состоящее из набора сервисов: $S = \{s_1, s_2, \dots, s_N\}$
N	Количество сервисов в приложении: N = |S|
D_i	Вектор, описывающий ресурсные требования сервиса s_i: $D_i = (d_i^1, d_i^2, \dots, d_i^R)$
d_i^i	Количество ресурса r_j, которое необходимо сервису s_i
T	Матрица траффика между сервисами: $T = [t_{ij}]_{N \times N}$
t_{ij}	Частота взаимодействия от сервиса s_i к сервису s_j
X	Построенное размещение: $X = [x_{ij}]_{N \times M}$, где x_{ij} = 1, если сервис s_i нужно разместить на машине m_j, и x_{ij} = 0 в остальных случаях

Чтобы достичь желаемой производительности сервисных приложений, планировщик должен учитывать не только потребности сервисов в нескольких ресурсах, но и ситуацию с трафиком между сервисами. Поскольку службы, особенно службы с интенсивным использованием данных, часто нуждаются в

частой передаче данных, производительность сети напрямую влияет на общую производительность. Учитывая динамику сети, размещение различных служб приложения имеет решающее значение для поддержания общей производительности, особенно когда в кластере возникает непредвиденная задержка или перегрузка сети. Учитывая ситуацию с трафиком, наиболее интуитивным решением является размещение сервисов с высокой скоростью трафика на одном компьютере, чтобы совместно расположенные сервисы могли использовать петлевой интерфейс для получения высокой производительности сети, не потребляя фактические сетевые ресурсы компьютера. кластер. Однако такие службы не могут быть размещены на одном компьютере из-за ограниченных ресурсов. Таким образом, с ограничениями ресурсов нужно найти решение для размещения, чтобы минимизировать общий трафик между сервисами, размещенными на разных машинах (межмашинный трафик), при этом удовлетворяя потребности сервисов в нескольких ресурсах, чтобы цель этой работы могла быть достигнута. быть сформулировано как:

$$min \sum_{i=1}^{N} \sum_{j=1}^{N} \sum_{p=1}^{M} \sum_{\substack{q=1 \\ q \neq p}}^{M} t_{ij} * x_{ip} * x_{jq} \quad (5)$$

при условии, что

$$\sum_{j=1}^{M} x_{ij} = 1 \ (\forall i \in \{1,2,\dots,N\}), \quad (6)$$

$$\sum_{j=1}^{M} x_{ij} * d_i^k \leq v_j^k \ (\forall j \in \{1,2,\dots,M\}, \forall k \in \{1,2,\dots,R\}), \quad (7)$$

$$x_{ij} \in \{0,1\} \ (\forall i \in \{1,2,\dots,N\}, \forall j \in \{1,2,\dots,M\}) \ . \quad (8)$$

Уравнение (2) гарантирует, что каждый сервис будет размещен на машине. Уравнение (3) гарантирует, что потребности машины в ресурсах не

превышают ее ресурсных возможностей. Уравнение (1) выражает цель данной работы.

2.2 Проблема наименьшего k-разреза

Поскольку приложение, основанное на микросервисах, как правило, не может быть размещено на одном компьютере, во время развертывания необходимо эффективное разделение набора сервисов, задействованных в приложении. После разделения каждое подмножество сервисов должно иметь возможность упаковываться на машину, что означает, что у машины достаточно ресурсов для запуска всех сервисов в подмножестве. Учитывая скорость трафика между различными сервисами, качество раздела имеет решающее значение для производительности приложения. Чтобы решить эту проблему, сначала рассмотрим задачу о минимальном разрезе k, чтобы понять ее сложность.

Пусть G = (V, E) – неориентированный граф, где V – множество узлов, а E – множество ребер. В графе каждое ребро $e_{u,v} \in$ E имеет неотрицательный вес $w_{u,v}$. k-разрезом в графе G называется множество ребер, удаление которых разбивает граф на k непересекающихся непустых компонент $G' = \{G_1, G_2, \dots, G_k\}$. Задача о минимальном k-разрезе состоит в том, чтобы найти k-разрез минимального общего веса ребер, два конца которого находятся в разных компонентах, что можно вычислить как:

$$\sum_{i=1}^{k-1} \sum_{j=i+1}^{k-1} \sum_{\substack{u \in G_i \\ v \in G_j}} w_{u,v} \tag{9}$$

Минимальный разрез – это просто минимальный k-разрез при k = 2. На рис. 1 показан пример минимального разреза графа. На рисунке показаны 6 разреза, и штриховая линия – это минимальный разрез графа, так как

суммарный вес ребер, разрезаемых штриховой линией, является минимальным из всех разрезов. Учитывая сервисное приложение, мы можем представить его в виде графа, где узлы представляют сервисы, а веса ребер представляют скорость трафика. В частности, скорость трафика от службы s_i к службе s_j и скорость от службы s_j к службе s_i представлены на графике в виде двух ребер соответственно. Следовательно, нахождение минимального k-разреза графа эквивалентно разбиению приложения на k частей при сохранении минимального общего трафика между различными частями. Однако для произвольного k задача о минимальном k-разрезе является NP-трудной.

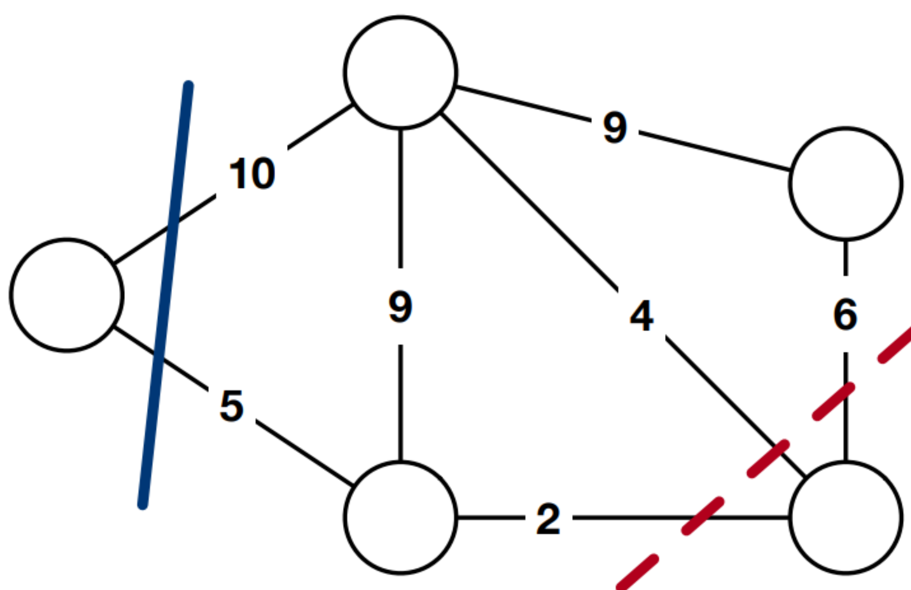

Рисунок 6 – Пример минимального разреза (красный пунктир)

В отличие от разработки детерминированного алгоритма, алгоритм Каргера [12] обеспечивает эффективный рандомизированный подход к поиску минимального сечения графа. Основная идея алгоритма Каргера состоит в том, чтобы случайным образом выбрать ребро $e_{u,v}$ из графа с вероятностью, пропорциональной весу ребра $e_{u,v}$, и объединить вершину u и вершину v в одну

(так называемое сжатие ребер). Чтобы найти минимальное сечение, алгоритм итеративно сжимает случайно выбранное ребро, пока не останется два узла. Оставшиеся ребра затем выводятся алгоритмом. Псевдокод показан на рисунке 7.

Algorithm 1: Contraction Algorithm ($k = 2$)

Input: $G = (V, E)$
Output: A cut of G
1 **while** $|V| > 2$ **do**
2 choose an edge $e_{u,v}$ with probability proportional to its weight;
3 $G \leftarrow G/e_{u,v}$; // contract edge $e_{u,v}$
4 **end**
5 **return** the cut in G;

Рисунок 7 – Алгоритм Каргера по сжатию графа

На рисунке 8 показан пример процесса алгоритма сжатия (k = 2). Алгоритм итеративно объединяет два узла выбранного ребра, а все остальные ребра повторно соединяются с объединенным узлом.

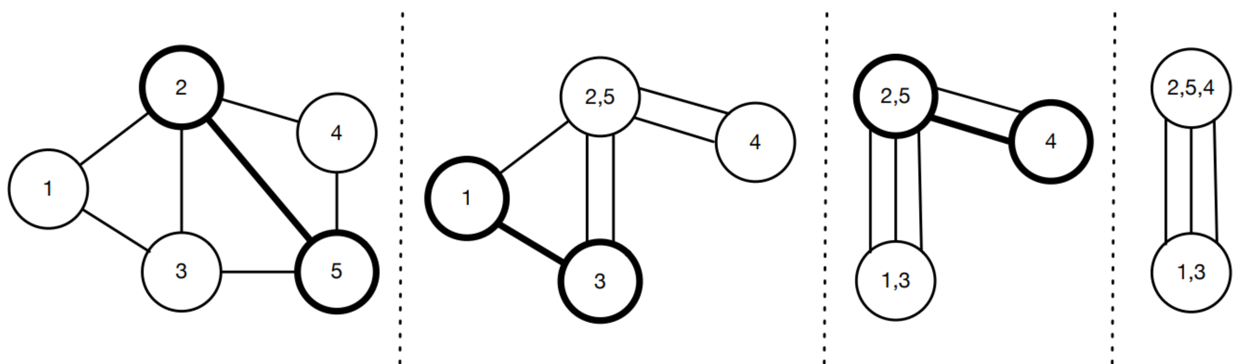

Рисунок 8 – Пример работы алгоритма сжатия

Для графа G = (V, E) с n = |V| узлы и m = |E| ребер, Каргер [12] утверждает, что алгоритм сжатия возвращает минимальный разрез графа с вероятностью Ω(1/n2). Следовательно, если мы выполним алгоритм сжатия

независимо n2log n раз, мы сможем найти минимальное сечение с высокой вероятностью; если мы не получим минимальный разрез, вероятность меньше, чем Ω(1/n). Для минимального k-разреза алгоритм сжатия в основном такой же, за исключением того, что он завершается, когда остается k узлов (замените |V| > 2 на |V| > k в алгоритме 1), и возвращает все ребра, оставшиеся в графе G. Точно так же алгоритм сжатия возвращает минимальный k-разрез графа с вероятностью Ω(1/n2k−2). Если мы выполним алгоритм независимо n2k−2log n раз, мы можем получить минимальный k-разрез с высокой вероятностью. Что касается временной сложности, алгоритм сжатия может быть реализован для работы за строго полиномиальное время O(mlog2n) [12].

2.3 Алгоритм размещения

Цель алгоритмов – найти решение для размещения, которое сведет к минимуму межмашинный трафик при одновременном удовлетворении потребностей в нескольких ресурсах. Ключевой дизайн предлагаемого подхода включает в себя:

- разделение приложения на основе алгоритмов сжатия;
- эвристическое уплотнение с учетом трафика;
- определение места размещения с настройкой порога.

2.3.1 Разделение приложения

Чтобы сделать значения различных ресурсов сопоставимыми друг с другом и простыми в обращении, сначала нормализуем количество доступных ресурсов на машинах и ресурсы, которые требуются службам, чтобы они были долей от максимальных. Определим термин $v_{max\text{-}j}$ как максимальное количество доступных ресурсов r_j на машине.

$$v_{max-j} = \max_{i\in\{1,2,\dots,M\}}\left(v_j^i\right) \tag{10}$$

Тогда вектор V_i доступных ресурсов на машине m_i и вектор D_i потребности в ресурсах обслуживания si нормализуется следующим образом:

$$V_i = \left(\frac{v_i^1}{v_{max-1}}, \frac{v_i^2}{v_{max-2}}, \dots, \frac{v_i^R}{v_{max-R}}\right) \quad (11)$$

$$D_i = \left(\frac{d_i^1}{v_{max-1}}, \frac{d_i^2}{v_{max-2}}, \dots, \frac{d_i^R}{v_{max-R}}\right) \quad (12)$$

После нормализации мы начинаем разбивать сервисное приложение. Ключевой вопрос, который нужно задать в первую очередь, заключается в том, на сколько частей разбито приложение. Принимая во внимание потребность в нескольких ресурсах для различных сервисов, введём пороговое значение α для определения количества частей при выполнении алгоритмов разделения. Порог α обозначает верхнюю границу потребностей в ресурсах разделенных частей, что означает, что мы непрерывно выполняем алгоритмы разделения до тех пор, пока общие потребности в ресурсах для каждой части не превысят α или ни одна часть не будет содержать более одного сервиса. С порогом $\alpha \in [0, 1]$ (поскольку требования к ресурсам были нормализованы) он гарантирует, что каждая часть после раздела может быть упакована в машину. На рисунке 9 показан пример раздела приложения с порогом $\alpha = 0.5$.

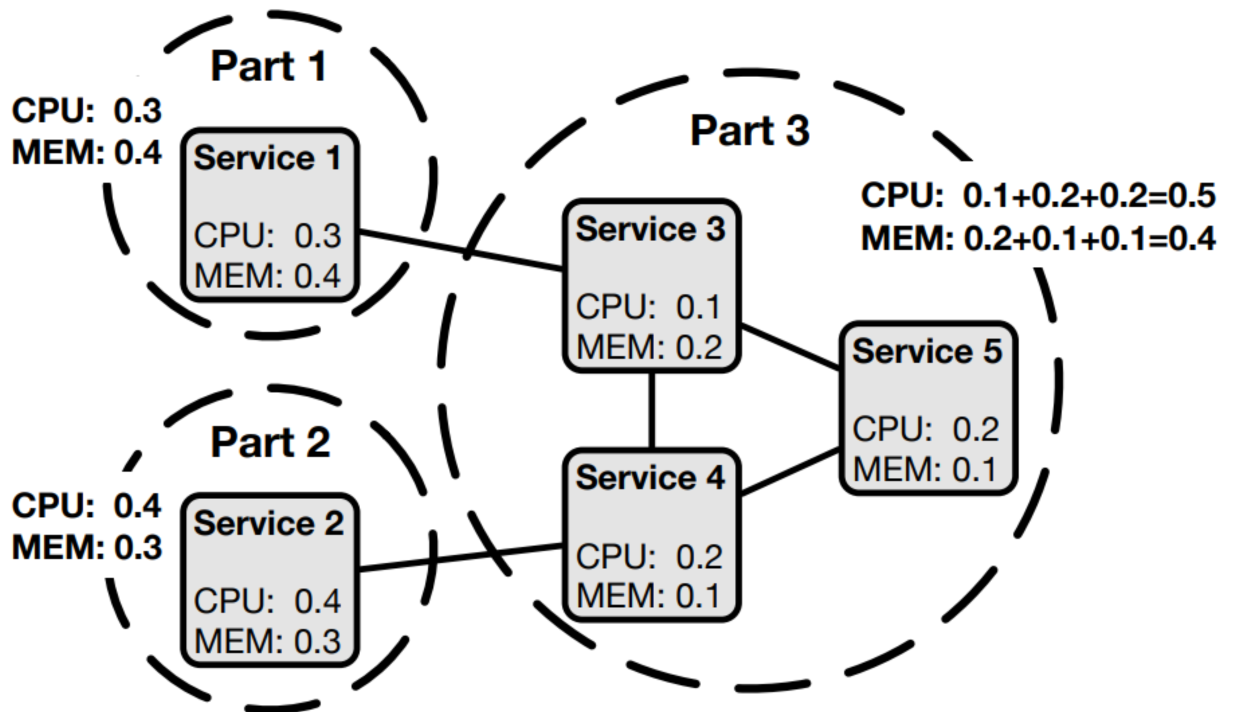

Рисунок 9 – Пример деления приложения с порогом α = 0.5

На рисунке суммарные требования CPU и требования к памяти от каждой части не превышают 0.5. Для заданного порога α предлагается два алгоритма разбиения: бинарное разбиение и k-разбиение, основанные на алгоритме сжатия.

2.3.2 Бинарное разделение

Идея алгоритма двоичного разбиения состоит в том, чтобы непрерывно выполнять двоичное разбиение приложения до тех пор, пока потребности в ресурсах каждой части не превысят α или ни одна часть не будет содержать более одного сервиса. Псевдокод показан на рисунке 10.

```
Algorithm 2: Binary Partition
   Input: service-based application S, threshold α
   Output: a partition of the application P = {S_1, S_2, ..., S_N'}, N' is number of parts after
            partition
 1 P ← {S};
 2 while exists part S_i in P that the total resource demands exceed α, and part S_i contains more than one
    service do
 3 |  P ← P − {S_i};
 4 |  Construct a graph G = (V, E) based on S_i;
 5 |  n ← |V|;
 6 |  G_min ← G;
 7 |  t ← 0;
 8 |  repeat
 9 |  |  Perform the contraction algorithm (k = 2) to get a cut G';
10 |  |  G_min ← min(G_min, G') ;                          // Store the smaller cut in G_min
11 |  |  t ← t + 1;
12 |  until t > n;
13 |  Get a partition {S_x, S_y} of part S_i according to G_min;
14 |  P ← P ∪ {S_x, S_y};
15 end
16 return P;
```

Рисунок 10 – описание алгоритма бинарного разделения

Основной процесс можно описать следующим образом. Алгоритм непрерывно проверяет потребность в ресурсах каждой части в текущем разделе приложения P. Начальный раздел P = {S}, где все приложение рассматривается как одна часть. Если общие потребности в ресурсах части S_i в P превышают пороговое значение α и часть S_i содержит более одного сервиса, часть выбирается для разделения на 2 части (двоичное разделение). Сначала он строит граф G = (V, E) на основе S_i, где узлы представляют сервисы, а веса ребер представляют скорость трафика. Как упоминалось в предыдущем разделе, если мы многократно выполняем алгоритм сжатия, мы можем получить минимальное сечение с высокой вероятностью. Принимая во внимание как качество разбиения, так и скорость разбиения, было решено выполнить алгоритм сжатия n раз в нашем алгоритме (в автономном режиме его можно настроить на запуск $n^2 \log n$ раз, чтобы получить минимальный разрез с высокой вероятностью). Затем, в соответствии с минимальным

разрезом G_{min}, который получается из алгоритма сжатия, он разбивает S_i на две части $\{S_x, S_y\}$. Этот процесс будет выполняться повторно до тех пор, пока потребности в ресурсах от каждой части не превысят порог α или ни одна часть не будет содержать более одного сервиса.

2.3.3 K-разделение

Идея алгоритма k-разбиения состоит в том, чтобы напрямую разбить приложение на k частей. Итеративно увеличивая k, он завершается, когда потребности в ресурсах от каждой части не превышают α или ни одна часть не содержит более одного сервиса. Псевдокод показан на рисунке 11.

```
Algorithm 3: K Partition
   Input: service-based application S, threshold α
   Output: a partition of the application P = {S1, S2, ..., SN'}, N' is number of parts after
           partition
 1 P ← {S};
 2 Construct a graph G = (V, E) based on S;
 3 n ← |V|;
 4 k ← 1;
 5 while exists part Si in P that the total resource demands exceed α and part Si contains more than one
    service do
 6    Gmin ← G;
 7    k ← k + 1;
 8    t ← 0;
 9    repeat
10       Perform the contraction algorithm until k nodes remain to get a k-cut G';
11       Gmin ← min(Gmin, G') ;                  // Store the smaller k-cut in Gmin
12       t ← t + 1;
13    until t > n;
14    Get a partition {S1, S2, ..., Sk} of the application S according to Gmin;
15    P ← {S1, S2, ..., Sk};
16 end
17 return P;
```

Рисунок 11 – описание алгоритма k-разделения

Основной процесс можно описать следующим образом. Алгоритм сначала строит граф G = (V, E) на основе приложения S, а затем непрерывно проверяет потребности в ресурсах каждой части в текущем разделе

приложения P, где изначально P = {S}. Если общие потребности в ресурсах части S_i в P превышают пороговое значение α и часть S_i содержит более одной услуги, это увеличивает k, которое является количеством разделенных частей. Как упоминалось в первом разделе, для получения минимального k-разреза с высокой вероятностью мы должны выполнить алгоритм сжатия независимо $n^{2k-2}\log n$ раз. Однако временная сложность возрастает экспоненциально с ростом k, что является чрезмерно высоким. Таким образом, мы делаем временную сложность совместимой с алгоритмом двоичного разбиения, жертвуя некоторой вероятностью нахождения минимального k-разреза. Он также выполняет алгоритм сжатия n раз. Затем, в соответствии с минимальным k-разрезом G_{min}, который мы получаем из алгоритма сжатия, приложение разбивается на k частей P = {S_1, S_2, ..., S_k}. Точно так же этот процесс будет выполняться повторно до тех пор, пока потребности в ресурсах от каждой части не превысят порог α или ни одна часть не будет содержать более одного сервиса.

Данный алгоритм можно представить в виде блок-схемы, изображенной на рисунке 12.

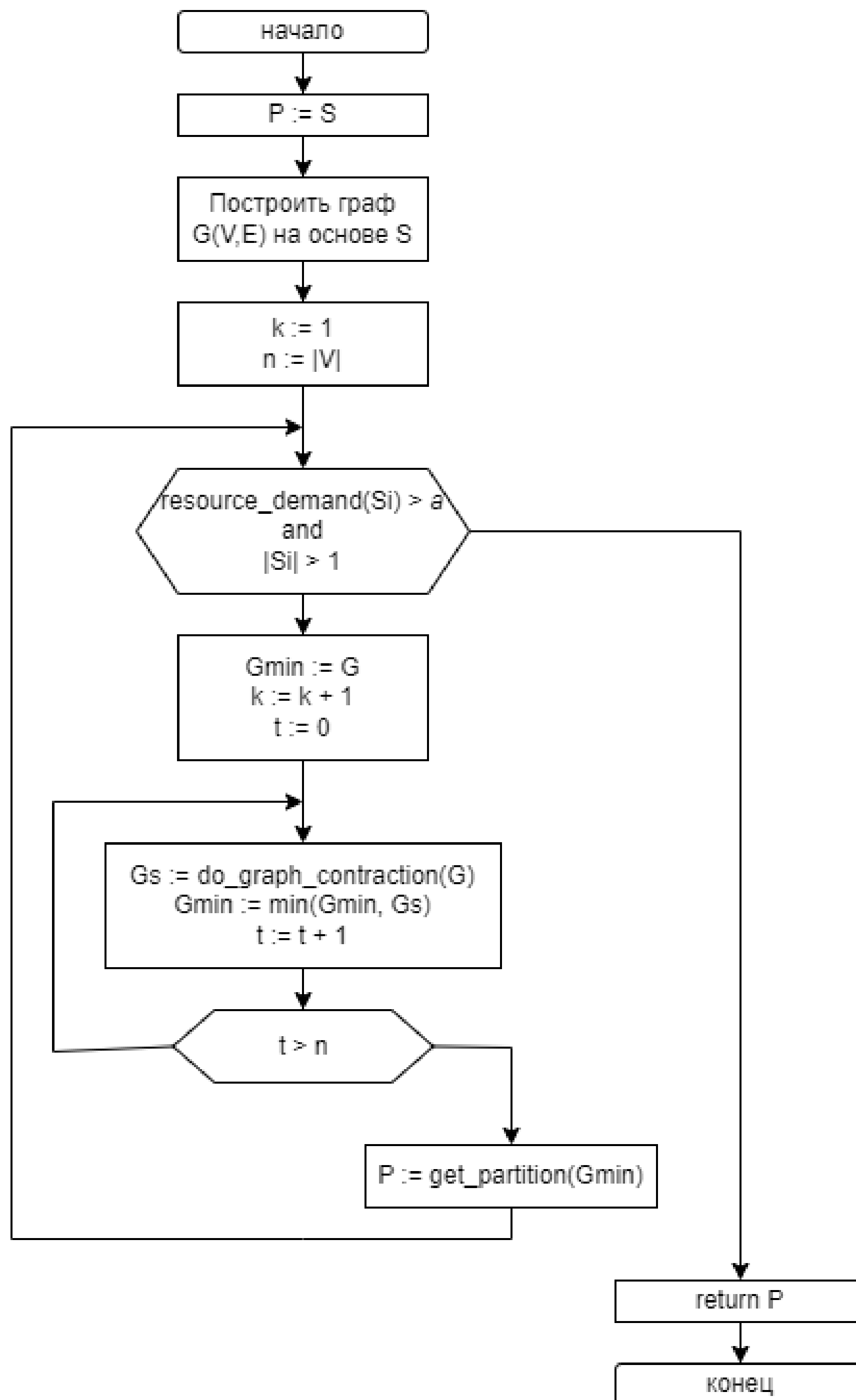

Рисунок 12 – алгоритм k-разделения

2.4 Эвристическая упаковка

Учитывая раздел приложения, алгоритм здесь состоит в том, чтобы упаковать каждую часть в разнородные машины. Без учета скорости трафика задачу можно сформулировать как классическую многомерную задачу

упаковки контейнеров, которая, как известно, является NP-полной [23]. Когда в приложении задействовано большое количество сервисов, невозможно найти оптимальное решение за полиномиальное время. Учитывая временную сложность и качество упаковки, мы используем две жадные эвристики в нашем алгоритме упаковки: осведомленность о трафике и эвристику наибольшей загрузки. Алгоритм представлен в виде псевдокода на рисунке 13.

Algorithm 4: Heuristic Packing

Input: partition of the application $P = \{\mathcal{S}_1, \mathcal{S}_2, ..., \mathcal{S}_{N'}\}$, vectors of available resources on each machine $\{V_1, V_2, ..., V_M\}$

Output: a placement solution X

1 Calculate vectors of resource demands of each part as: $\{D'_1, D'_2, ..., D'_{N'}\}$;
2 $X \leftarrow [x_{ij} = 0]_{N' \times M}$;
3 **for** $i \leftarrow 1$; $i \leq N'$; $i \leftarrow i + 1$ **do**
4 $tf \leftarrow 0$; $ml \leftarrow 0$; $y \leftarrow 0$;
5 **for** $j \leftarrow 1$; $j \leq M$; $j \leftarrow j + 1$ **do**
6 **if** *part* $\mathcal{S}_i$ *can be packed into machine* m_j **then**
7 $tf_j \leftarrow \sum t_{uv}$;
/* Calculate the total traffic rates between part $\mathcal{S}_i$ and machine m_j, for any service s_u in $\mathcal{S}_i$ and any service s_v packed into machine m_j before */
8 $ml_j \leftarrow \sum_{k=1}^{R} \frac{d'^k_i}{v^k_j}$;
/* Calculate the load situation between the vector of resource demands from part $\mathcal{S}_i$ and the vector of available resources on machine m_j */
9 **if** $tf_j > tf$ **then**
10 $tf \leftarrow tf_j$; $ml \leftarrow ml_j$; $y \leftarrow j$;
11 **end**
12 **else if** $tf_j == tf$ *and* $ml_j > ml$ **then**
13 $tf \leftarrow tf_j$; $ml \leftarrow ml_j$; $y \leftarrow j$;
14 **end**
15 **end**
16 **end**
17 **if** $y == 0$ **then**
18 **return** *null*;
19 **end**
20 **else**
21 $V_y \leftarrow V_y - D'_i$;
22 $x_{iy} \leftarrow 1$;
23 **end**
24 **end**
25 **return** X;

Рисунок 13 – описание алгоритма эвристической упаковки

Чтобы найти наилучшую возможную машину для детали S_i, алгоритм вычисляет два коэффициента соответствия: tf и ml. Для машины m_j коэффициент *tf* представляет собой сумму скорости трафика между службами в части S_i, и ранее было определено, что службы упакованы в машину m_j. Коэффициент *ml* представляет собой скалярное значение ситуации загрузки между вектором потребностей в ресурсах со стороны части S_i и вектором доступных ресурсов на машине m_j. Предполагая, что D_i' — это вектор потребности в ресурсах части S_i, а $d_i'^k$ – это количество ресурсов r_k, требуемых частью S_i, оно равно $ml = \sum_{k=1}^{R} \frac{d\prime_j^k}{v_j^k}$. Чем больше *ml*, тем больше загружена машина [24,25]. Идея этой эвристики состоит в том, чтобы повысить эффективность использования ресурсов за счет размещения детали на наиболее загруженной машине. Поскольку основной целью является минимизация межмашинного трафика, алгоритм разработан таким образом, чтобы в первую очередь расставлять приоритеты для машин на основе коэффициентов *tf*. Если коэффициенты *tf* одинаковы, он отдает приоритет машинам на основе коэффициентов ml. Следовательно, если все части в разделе могут быть упакованы в машины, алгоритм возвращает решение о размещении. В противном случае возвращается ноль.

2.5 Поиск размещения

Как было сказано ранее, для разделения приложения алгоритму требуется пороговое значение α. Однако задать соответствующий детерминированный порог α сложно, так как он не может гарантировать, что алгоритм сможет найти решение для размещения посредством рандомизированного разбиения и эвристической упаковки при определенном пороге α. Интуитивно более высокий порог α приводит к меньшему количеству частей в разделе, что приводит к меньшей скорости трафика между

различными частями. Таким образом, мы вводим простой алгоритм для поиска лучшего порога α путем перечисления от большего к меньшему. Алгоритм показан на рисунке 14.

```
Algorithm 5: Placement Finding
   Input: service-based application S, vectors of available resources on each machine
          {V_1, V_2, ..., V_M}
   Output: a placement solution X
 1 X ← [x_ij = 0]_{N×M};
 2 α ← 1.0;
 3 Δ ← 0.1;
 4 while α ≥ 0.0 do
 5 |  P ← Binary_Partition(S, α);
   |  /* Or P ← K_Partition(S, α);                                          */
 6 |  X' ← Heuristic_Packing(P, {V_1, V_2, ..., V_M});
 7 |  if X' ≠ null then
 8 |  |  Calculate X according to X' and P;
 9 |  |  return X;
10 |  end
11 |  α ← α − Δ;
12 end
13 return null;
```

Рисунок 14 – описание алгоритма для нахождения оптимального размещения

В начале значение α равно 1.0. Чтобы настроить пороги, мы устанавливаем значение шага Δ, и значение по умолчанию равно 0.1, которое может быть настроено пользователями. На каждой итерации с порогом α алгоритм сначала разбивает данное приложение S на основе алгоритма двоичного разбиения или алгоритма разбиения k. Алгоритм записывает последние результаты разделения, чтобы избежать многократного повторения разделения. Затем он пытается упаковать все части раздела в машины на основе эвристического алгоритма упаковки, чтобы найти решение для размещения приложения.

Далее, рассмотрим временную сложность предложенного алгоритма. Предположим, что количество сервисов равно n; количество ребер в сервисном графе равно m (т. е. количество скоростей трафика $t_{ij} > 0$); количество машин равно M. Приложение на основе микросервисов может быть разделено на n частей. Для каждого раздела мы выполняем алгоритм сжатия n раз, а временная сложность алгоритма сжатия составляет $O(m\log^2 n)$. Поскольку мы записываем последние результаты разделения, чтобы избежать многократного повторения разделения, временная сложность всего раздела составляет $O(n^2 m\log^2 n)$. Для эвристической упаковки временная сложность равна $O(nM + n^2)$, так как общая временная сложность вычисления фактора *tf* равна $O(n^2)$. Пусть $C = 1/\Delta$ обозначает количество итераций. Общая временная сложность предложенного алгоритма составляет $(n^2 m\log^2 n + CnM + Cn^2)$.

2.6 Выводы по разделу

В данном разделе была составлена математическая модель процесса размещения микросервисов в кластере. Исследование показало, что за счет использования алгоритма Каргера NP-полную задачу поиска минимального сечения графа можно выполнить за приемлемое время O(mlog2n).

Также были спроектированы алгоритмы разделения приложения на части, которые впоследствии будут устанавливаться на узлы кластера.

3 Тестирование полученного планировщика

3.1 Проведение эксперимента

Прототип планировщика был реализован с использованием python, основанного на предложенных в работе алгоритмах, для развертывания сервисных приложений в контейнерных кластерах. В экспериментах проводится оценка планировщика на тестовых кластерах экспериментальной среды ExoGENI [13].

Было создано два разных тестовых кластера для проведения экспериментов. Для первого кластера использовалось 30 однородных виртуальных машин с 2 ядрами ЦП и 6 ГБ ОЗУ. Учитывая неоднородность, мы используем 10 виртуальных машин (ВМ) с 2 ядрами ЦП и 6 ГБ ОЗУ и 10 ВМ с 4 ядрами ЦП и 12 ГБ ОЗУ для второго кластера. В однородном кластере 30 ВМ, а в гетерогенном кластере 20 ВМ, но общая емкость ресурсов одинакова.

Для того чтобы оценить предлагаемые алгоритмы в различных сценариях, в экспериментах используются синтетические приложения. Учитывая масштаб тестового кластера, мы получаем сервисные приложения, состоящие из 64, 96 и 128 сервисов. Для размера 64 потребность в ЦП для каждой службы равномерно выбирается случайным образом из [30, 100], где 100 представляет 1 ядро ЦП, а потребность в памяти выбирается случайным образом из [100,300], где 100 представляет 1 ГБ ОЗУ. Для размера 96 потребность в ЦП выбирается случайным образом из [20,67], а потребность в памяти выбирается случайным образом из [67,200]. Для размера 128 требования к ЦП выбираются случайным образом из [15,50], а требования к памяти выбираются случайным образом из [50,150]. В соответствии с этими диапазонами общие потребности в ресурсах приложений разного размера примерно одинаковы. Для каждого размера приложения создавалось 10000 экземпляров для тестирования. Поскольку работа [14] показывает, что

логарифмически нормальное распределение обеспечивает наилучшее соответствие трафику центра обработки данных, было решено генерировать потребности в трафике между сервисами с вероятностью 0,05 (чтобы убедиться, что граф приложений подключен), а скорость трафика соответствует логарифмически нормальное распределение (среднее = 5 Мбит/с, стандартное отклонение = 1 Мбит/с).

Все предложенные алгоритмы были реализованы в прототипе планировщика, где алгоритм сжатия основан на параллельной реализации [12]. Поскольку было предложено два алгоритма для разделения приложений, существует два типа конфигурации. BP-HP основан на двоичном разделе (BP) и эвристической упаковке (HP). KP-HP основан на k-разбиении (KP) и эвристической упаковке.

Как было упомянуто выше, многие исследовательские усилия были посвящены проблеме размещения сложного программного обеспечения как услуги (SaaS) [6,15]. Однако они нацелены на размещение определенного набора предопределенных сервисных компонентов. Что еще более важно, эти метаэвристические подходы часто занимают минуты или даже часы, особенно для крупномасштабных кластеров, для создания решения по размещению, которое может столкнуться с трудностями при онлайн-ответе. Другая исследовательская работа посвящена проблеме размещения виртуальных машин с учетом трафика [16,17]. Однако эти решения основаны на определенной топологии сети, в то время как предлагаемый в данной работе подход не зависит от топологии сети. Таким образом, для сравнения с созданным планировщиком были вырбраны следующие схемы:

– Планировщик Kubernetes (KS): планировщик по умолчанию в кластере контейнеров Kubernetes [18] имеет тенденцию равномерно распределять контейнеры по кластеру, чтобы сбалансировать общее использование ресурсов кластера. В частности, была добавлена мягкая привязка (то есть привязка к модулям в Kubernetes) для сервисов, между которыми

есть трафик, поскольку планировщик попытается разместить сервисы, между которыми есть сходство, на одной машине.

– First Fit Decreasing (FFD): это простой и широко используемый алгоритм для многомерной задачи упаковки контейнеров [19]. FFD сначала сортирует сервисы в порядке убывания в соответствии с определенным спросом на ресурсы, а затем упаковывает каждый сервис на первую машину с достаточными ресурсами.
– Best-Fit Decreasing (BFD): сервис размещается на самом загруженном компьютере, на котором еще достаточно мощности. BFD сначала сортирует машины в порядке убывания в соответствии с определенной емкостью ресурсов, а затем упаковывает каждую службу на первую машину с достаточными ресурсами.
– Multi-resource Packer (PACK): идея этой эвристики [4] заключается в том, что она планирует сервисы в возрастающем порядке согласования между потребностями в ресурсах сервисов и доступностью ресурсов машин (т. е. скалярное произведение между вектором потребностей в ресурсах и вектором имеющихся ресурсов).
– Случайный (RAND): он случайным образом выбирает сервис в приложении, а затем упаковывает ее на первую машину с достаточными ресурсами.

3.2 Сравнение показателей

На рисунке 15 показан коэффициент успешного размещения различных схем в двух кластерах.

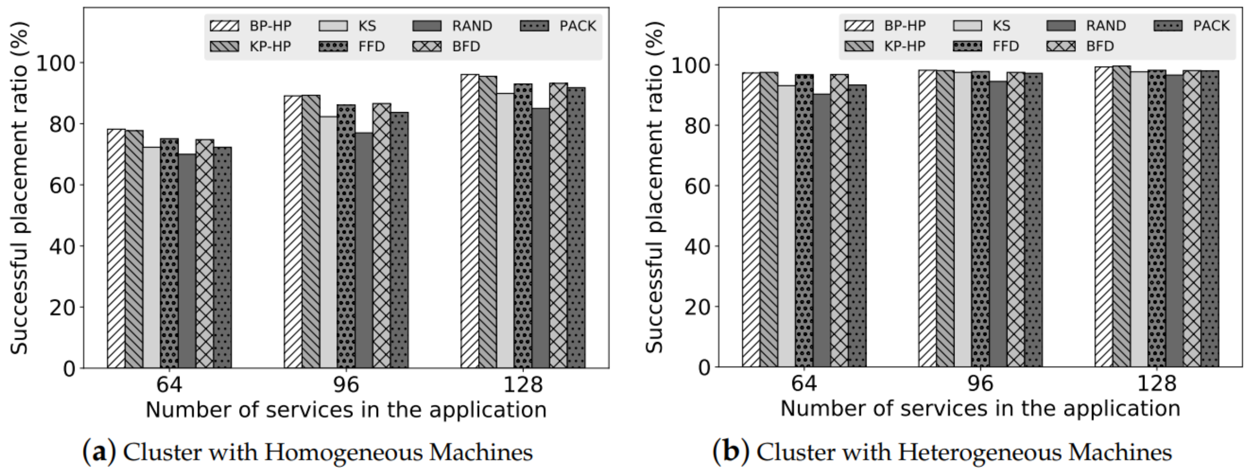

Рисунок 15 – Сравнение показателей различных алгоритмов размещения

Успешное размещение приложения заключается в том, что алгоритм может найти решение для размещения всех задействованных сервисов, поэтому отношение количества успешно размещенных приложений к количеству всех запрошенных приложений [26]. Мы видим, что RAND работает хуже всего, так как у него нет эвристики для упаковки сервисов. FFD и BFD работают лучше, чем KS и PACK, потому что KS в основном фокусируется на балансировке использования ресурсов в кластере, а PACK фокусируется на согласовании между потребностями в ресурсах и доступностью ресурсов. FFD и BFD были продемонстрированы как эффективные алгоритмы для многомерных задач упаковки контейнеров [20]. BP-HP работает сравнимо с KP-HP, и оба они немного превосходят другие схемы в этой оценке. В основном это связано с тем, что итеративное разделение и упаковка с различными пороговыми значениями повышают вероятность нахождения решения по размещению. Более того, алгоритм упаковки может плотно упаковывать сервисы за счет наиболее загруженной эвристики. Результаты однородного кластера также показывают, что коэффициент успешного размещения увеличивается с увеличением количества услуг. Поскольку общие потребности в ресурсах приложений

разного размера (разное количество сервисов) примерно одинаковы, меньшее количество сервисов приводит к большей потребности в ресурсах для каждого отдельного сервиса, что легко вызывает проблему фрагментации ресурсов при размещении. По сравнению с однородным кластером коэффициент успешного размещения намного выше в гетерогенном кластере. Поскольку машины имеют большую емкость ресурсов в гетерогенном кластере, проще упаковать сервисы, ограниченные несколькими ресурсами [21,22].

Далее рассмотрим ситуацию с траффиком на разных схемах. При оценке сравним только приложения, все сервисы которых размещены в кластере по разным алгоритмам.

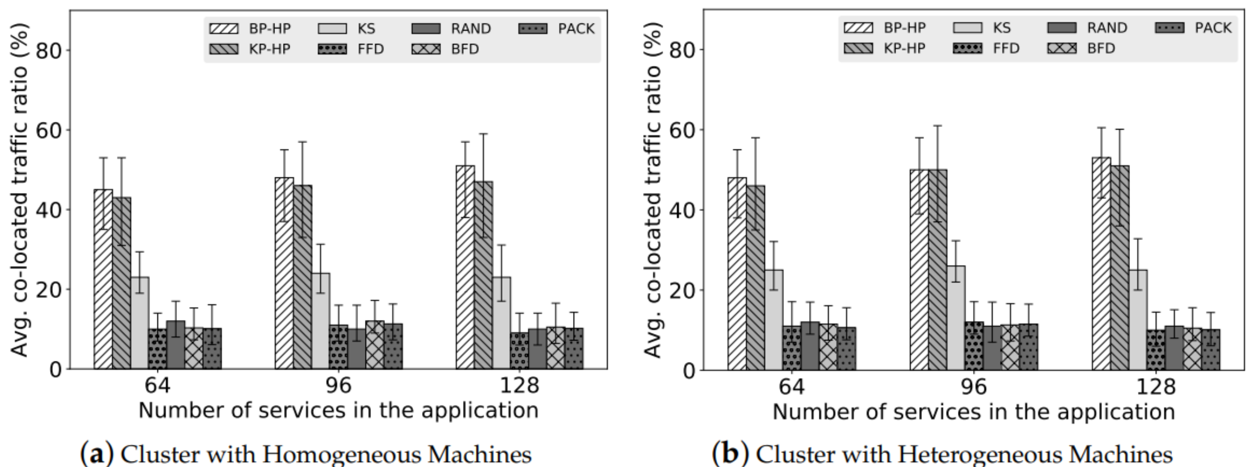

Рисунок 16 – Сравнение среднее соотношение совмещенного трафика для различных схем размещения

На рисунке 16 показано среднее соотношение совмещенного трафика для различных схем, а планки погрешностей представляют максимальное и минимальное соотношение. Совместно размещенный трафик – это трафик между сервисами, размещенными на одном компьютере, поэтому отношение – это объем совместно размещенного трафика к объему всего трафика. Для минимизации межмашинного трафика чем выше коэффициент совместного

размещения, тем лучше решение для размещения. Более точные значения коэффициентов совместного трафика приведены в таблице 3.

Таблица 3 – Среднее соотношение (%) совмещенного трафика для различных схем размещения

Схема	Однородный кластер			Неоднородный кластер		
	Средн.	Мин.	Макс.	Средн.	Мин.	Макс.
BP-HP	48.1	35.2	56.9	50.3	37.9	60.5
KP-HP	45.3	30.8	59.1	49.0	35.2	61.1
KS	23.3	17.1	31.4	25.6	20.4	32.8
FFD	10.0	6.5	15.8	10.9	6.2	17.1
BFD	10.9	6.4	17.2	11.1	7.3	16.6
PACK	10.5	6.1	16.3	10.7	6.1	16.5
RAND	10.6	6.2	16.7	11.0	6.8	16.8

Мы видим, что BP-HP и KP-HP значительно превосходят базовые показатели. Для кластера с однородными машинами BP-HP улучшает средний коэффициент совместного трафика с 24,8% до 38,1%; KP-HP улучшает соотношение на 22% до 35,3%. Для кластера с разнородными машинами BP-HP улучшает средний коэффициент совместного трафика с 24,7% до 39,6%; KP-HP улучшает соотношение на 23,4% до 38,3%. FFD, BFD, PACK и RAND работают плохо, поскольку они сосредоточены только на упаковке сервисов без учета скорости трафика. Поскольку в данной работе устанавливается привязка к сервисам, между которыми есть трафик в KS, KS пытается разместить связанные сервисы на той же машине. Однако KS игнорирует конкретную скорость трафика при принятии решений о размещении. Что касается BP-HP и KP-HP, можно обнаружить, что BP-HP работает немного лучше и стабильнее, чем KP-HP, но в некоторых случаях KP-HP может найти лучшее решение (согласно планкам погрешностей). Напротив, KP-HP также легко возвращает худшее решение. В основном это связано с тем, что BP-HP

выполняет алгоритм сжатия для нахождения минимального разреза с вероятностью $\Omega(1/n^2)$; KP-HP выполняет алгоритм сжатия, чтобы найти минимальный k-разрез с вероятностью $\Omega(1/n^{2k-2})$, которая намного меньше, чем BP-HP. Таким образом, производительность KP-HP сильно различается в экспериментах. Тем не менее, благодаря разделению, которое стремится совместить большие потребности в трафике, и упаковке с учетом трафика, BP-HP и KP-HP могут эффективно сократить межмашинный трафик для развертывания приложений на основе микросервисов в компьютерных кластерах.

3.3 Влияние значения порога α

В этом разделе рассмотрим влияние порога α на размещение приложений на основе услуг. Для иллюстрации мы фиксируем порог α, используя BP-HP в кластере с однородными машинами. На рисунке 11 показан коэффициент успешного размещения при различных значениях порога α. Например, BP-HP может найти решение для размещения 77% приложений с 64 сервисами при $\alpha = 0.5$. Мы наблюдаем, что коэффициент успешного размещения уменьшается, когда значение порога α в целом увеличивается, и лишь немногие заявки могут быть успешно размещены, когда $\alpha > 0.7$. Более высокий порог α приводит к меньшему количеству частей и более высоким средним требованиям к ресурсам частей в разделе, поэтому становится сложнее упаковать их в машины с ограничениями по нескольким ресурсам. Чтобы понять влияние на сетевой трафик, на рисунке 12 показаны результаты среднего коэффициента совместного трафика для каждого значения порога α, а планки погрешностей представляют максимальное и минимальное соотношение. Это явно демонстрирует, что коэффициент совместного трафика увеличивается больше, когда α больше. Однако больший порог α увеличивает сложность упаковки приложений. Таким образом, мы пытаемся

найти подходящий порог α путем перебора от большего к меньшему в предлагаемых алгоритмах.

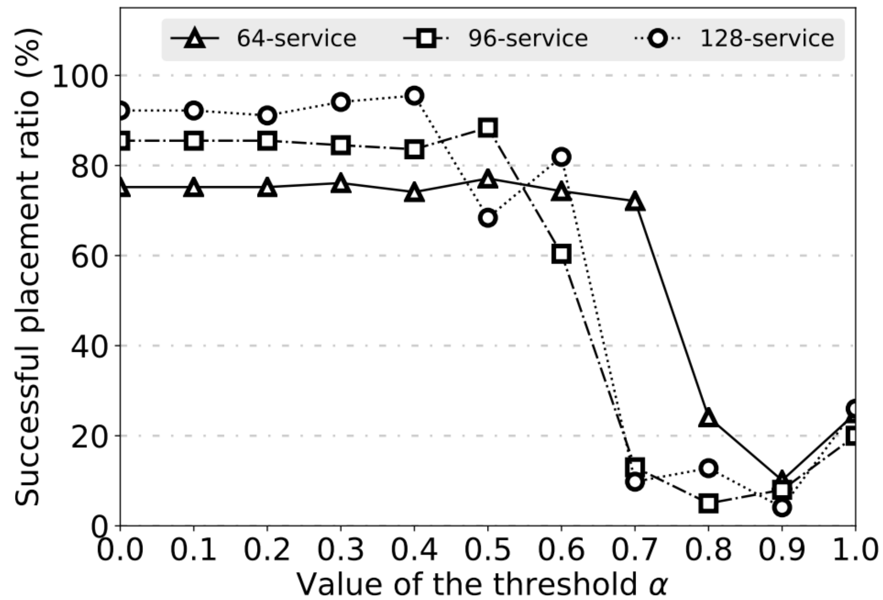

Рисунок 17 – Коэффициент успешного размещения на однородном кластере при использовании BP-HP с разными значениями порога α

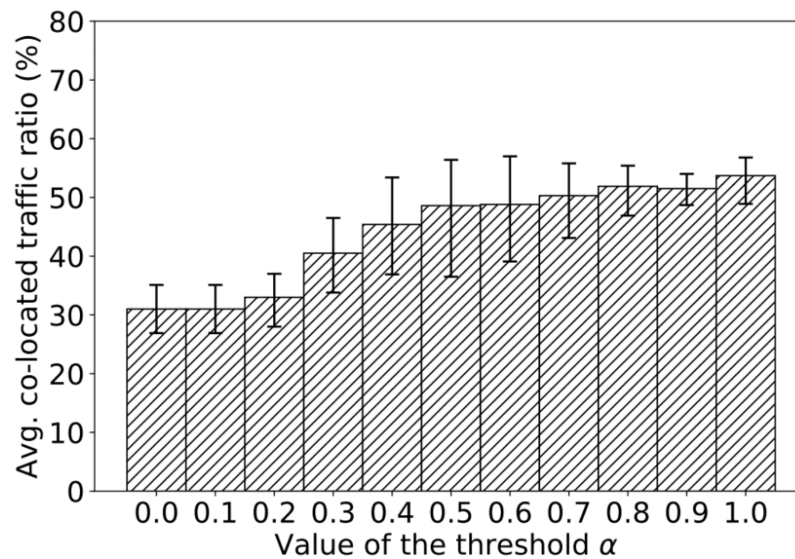

Рисунок 18 – Средний коэффициент совместного трафика в однородном кластере при использовании BP-HP с разными значениями порога α

3.4 Оценка издержек

В этом разделе рассмотрим накладные расходы, измеряя время выполнения алгоритма и сравнивая его с KS и RAND. Чтобы справедливо сравнить время выполнения алгоритма, алгоритм планирования KS был также реализован в Python, который аналогичен другим схемам. Данный эксперимент проводился на выделенном сервере с процессором Intel Xeon E5-2630 2,4 ГГц и памятью 64 ГБ.

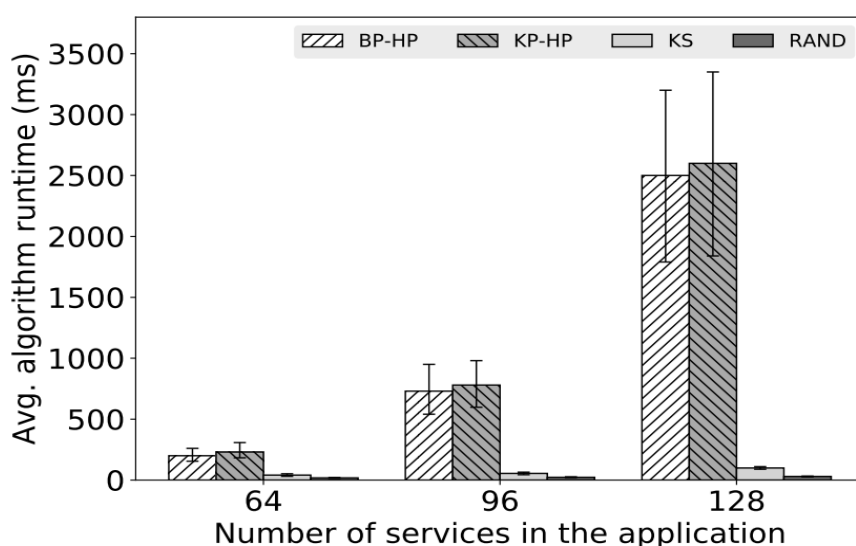

Рисунок 19 – Среднее время работы алгоритмов разных схем для гетерогенного кластера

На рисунке 19 показаны результаты среднего времени работы алгоритма различных схем для гетерогенного кластера (однородный кластер аналогичен), а планки погрешностей представляют максимальное и минимальное время работы алгоритма. RAND влечет за собой небольшие накладные расходы, так как это очень простой алгоритм. По сравнению с RAND, KS немного сложнее, поскольку KS имеет несколько предикативных политик и политик приоритетов для фильтрации и оценки машин, таких как обработка сходства между сервисами. BP-HP и KP-HP более сложны, чем

базовые варианты, и имеют явно более высокие накладные расходы. Также видно, что разница между максимальным и минимальным временем выполнения алгоритма довольно велика, так как время выполнения алгоритма сильно зависит от значения порога α. В алгоритме более высокий порог α приводит к меньшему количеству итераций, а более низкий порог α вызывает большее количество итераций. Тем не менее, BP-HP и KP-HP могут реагировать за секунды для приложений разного размера. В частности, для приложения с менее чем 100 сервисами BP HP и KP-HP могут отвечать менее чем за секунду, что приемлемо для оперативного планирования. Более того, наиболее трудоемкой частью предлагаемых алгоритмов является разбиение приложений, а значит, не будет большой разницы во времени выполнения алгоритма для крупных кластеров с одинаковым количеством сервисов. Предполагается, что предложенные алгоритмы также могут эффективно решать проблему размещения на крупномасштабных кластерах.

3.5 Выводы по разделу

В этом разделе исследовалась проблема размещения сервисов для микросервисной архитектуры в облаках. Чтобы найти высококачественное разделение сервисных приложений, было предложено два алгоритма разделения: Binary Partition и K Partition, которые основаны на хорошо разработанном алгоритме рандомизированного сжатия. Для эффективной упаковки приложения используется наиболее загруженная эвристика и учет трафика в алгоритме упаковки. Путем корректировки порога α, обозначающего верхнюю границу потребности в ресурсах, можно найти лучшее решение для размещения сервисных приложений. Был реализован прототип планировщика на основе предложенных алгоритмов и оценен на тестовых кластерах. В ходе оценки было показано, что эти алгоритмы могут улучшить коэффициент успешного размещения приложений в кластере, при этом значительно увеличив долю совместного трафика (т. е. уменьшив

межмашинный трафик). При оценке накладных расходов результаты показывают, что предложенные алгоритмы несут некоторые накладные расходы, но в приемлемое время. Можно считать, что предложенные алгоритмы практичны для реальных случаев использования.

В результате выполненной работы были изучены существующие планировщики кластеров, был проведен сравнительный анализ этих решений, показавший, что при планировании ресурсов почти ни один планировщик не учитывает трафик между микросервисами.

4 Внедрение полученного планировщика на производство

4.1 Производство

В качестве производства, на котором выполнялось внедрение полученного планировщик ресурсов, была выбрана компания Netcracker. Она занимается разработкой программного обеспечения для операторов связи и интернет-провайдеров, в том числе с применением облачных технологий. За долгое время работы компания разработала множество решений, которые размещаются в облаках, а поэтому они идеально подходят для интеграции с новым планировщиком ресурсов.

Для внедрения нового планировщика был выбран один из продуктов Netcracker, который включает в себя более ста микросервисов, логически разделенных на независимые приложения, состоящие из нескольких микросервисов, которые могут обмениваться сообщениями друг с другом как внутри приложения, так и с микросервисами, входящими в состав других приложений. Учитывая общий размер решения, количество связей по сети между сервисами довольно высоко, поэтому влияние нового алгоритма планировщика на скорость развертывания будет хорошо заметно. Также данный продукт содержит три сервиса, к которым обращаются 90% остальных сервисов.

Данный продукт имеет возможность установки в разные облачные системы, такие как Kubernetes, OpenShift, Amazon AWS. Командой разработки был предоставлен сервер с установленным Kubernetes, поэтому далее в работе будут сравниваться результаты работы стандартного планировщика Kubernetes и планировщика, разработанного в рамках данного исследования.

Стоит отметить, что развертывание настолько крупного приложения занимает достаточно много времени – порядка 8 часов при полной установке, включающей все части приложения. Это также должно облегчить анализ

результатов, так как на длительном времени будет лучше заметно влияние внесенных изменений.

4.2 Внедрение

Как говорилось ранее, для того, чтобы алгоритм нахождения минимального k-среза, на котором основан планировщик, работал для распределения сервисов на кластере, нужно определить, какими входными данными обладает процесс развертывания предоставленного продукта.

Во-первых, должно быть известно о машинах в кластере и их ресурсах, так как каждая из них обладает своим набором ресурсов, таких как CPU, оперативная память, дисковое пространство и т.д. Также должно быть известно количество машин и количество доступных ресурсов на каждой из них.

Во-вторых, при использовании моделей IaaS (Infrastructure as a Service – инфраструктура как сервис) и CaaS (Container as a Service – контейнер как сервис) пользователь может установить потребность в ресурсах (комбинацию CPU, оперативной памяти, дискового пространства) для виртуальной машины, либо контейнера при составлении конфигурации развертывания [27,28]. Поэтому потребности в ресурсах известны при получении запроса на развертывание. Приложение, построенное на базе микросервисной архитектуры, зачастую состоит из набора сервисов со своими уникальными наборами требований к ресурсам [29]. Поэтому следующие параметры необходимы для работы: набор сервисов, количество сервисов, набор требований к ресурсам для каждого сервиса.

В-третьих, так как в данной работе рассматривается размещение сервисов с учетом траффика между ними, требуется знать, как сервисы взаимодействуют друг с другом по сети – то есть связи между сервисами и частота такого взаимодействия.

Для тестового стенда значения параметров, перечисленных в таблице 1, представлены в таблице 4.

Таблица 4 – Параметры тестового стенда

Параметр	**Значение**
$\mathcal{M}$(Набор машин в кластере)	$\mathcal{M}= \{m_1, m_2, m_3, m_4\}$
M (Количество машин в кластере)	4
$\mathcal{R}$ (Набор типов ресурсов)	$\mathcal{R} = \{ОЗУ, CPU, MEM\}$
R (Количество типов ресурсов)	3
V_i (Вектор, описывающий доступные ресурсы машины)	$V_1 = (v_1^1, v_1^2, v_1^3)$ $V_2 = (v_2^1, v_2^2, v_2^3)$ $V_3 = (v_3^1, v_3^2, v_3^3)$ $V_4 = (v_4^1, v_4^2, v_4^3)$
v_i^j (Количество ресурса r_j, доступного на машине m_i)	Вычисляется в процессе выполнения размещения
S (Приложение, состоящее из набора сервисов)	$S = \{s_1, s_2, \dots, s_{125}\}$
N (Количество сервисов в приложении)	125
D_i (Вектор, описывающий ресурсные требования сервиса s_i)	Для большинства сервисов $D_i = (256Mb\ \ , 64mc\ \ , 2Gb\ \ )$

В таблице не приводятся ресурсные требования сервисов и матрица траффика между сервисами, а также частота взаимодействия сервисов, так как эти параметры занимают много места в таблице и их тяжело воспринимать.

Чтобы достичь желаемой производительности сервисных приложений, планировщик должен учитывать не только потребности сервисов в нескольких ресурсах, но и ситуацию с трафиком между сервисами. Поэтому для работы планировщика необходимо было произвести сбор данных о сетевых взаимодействиях между микросервисами и их частоте, чтобы построить

матрицу взаимодействия. Полученная информация также была внесена в конфигурацию сервисов наряду с потребляемой оперативной памятью, количеством используемых ядер процессора и требуемого места на диске.

```yaml
name: small_flavor
parameters:
  # CPU limit should be specified in millicores
  - AIS_CPU_LIMITS: 1000m
  - AIS_CPU_REQUEST: $(( ${AIS_CPU_LIMITS::-1} / 10 ))m
  - AIS_MEMORY_REQUEST: 500Mi
  - AIS_MEMORY_LIMITS: 1024Mi
  - AIS_HEAP_MIN_SIZE: 512m
  - AIS_HEAP_MAX_SIZE: 700m
  - AIS_INTEGRATIONS: ["streaming-platform", "api-gateway", "identity-provider"]
  # Requests per hour
  - AIS_TRAFFIC_RATE: [7200, 3600, 3600]
```

Рисунок 20 – Пример конфигурации одного из сервисов

Далее необходимо было подключить созданный планировщик к Kubernetes. Это можно сделать при помощи API самого Kubernetes и файла конфигурации, в котором используются соответственные абстракции.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    component: scheduler
    tier: control-plane
  name: k-cut-scheduler
  namespace: kube-system
spec:
  selector:
    matchLabels:
      component: scheduler
      tier: control-plane
  replicas: 1
  template:
    metadata:
      labels:
        component: scheduler
        tier: control-plane
        version: second
    spec:
      serviceAccountName: k-cut-scheduler
```

Рисунок 21 – Фрагмент конфигурации Kubernetes

Также было необходимо добавить в конфигурацию всего приложения новые параметры, чтобы система Continuous Integration смогла передать их в Kubernetes для развертывания приложения.

Таким образом процесс развертывания системы можно представить в виде диаграммы, представленной на рисунке 22.

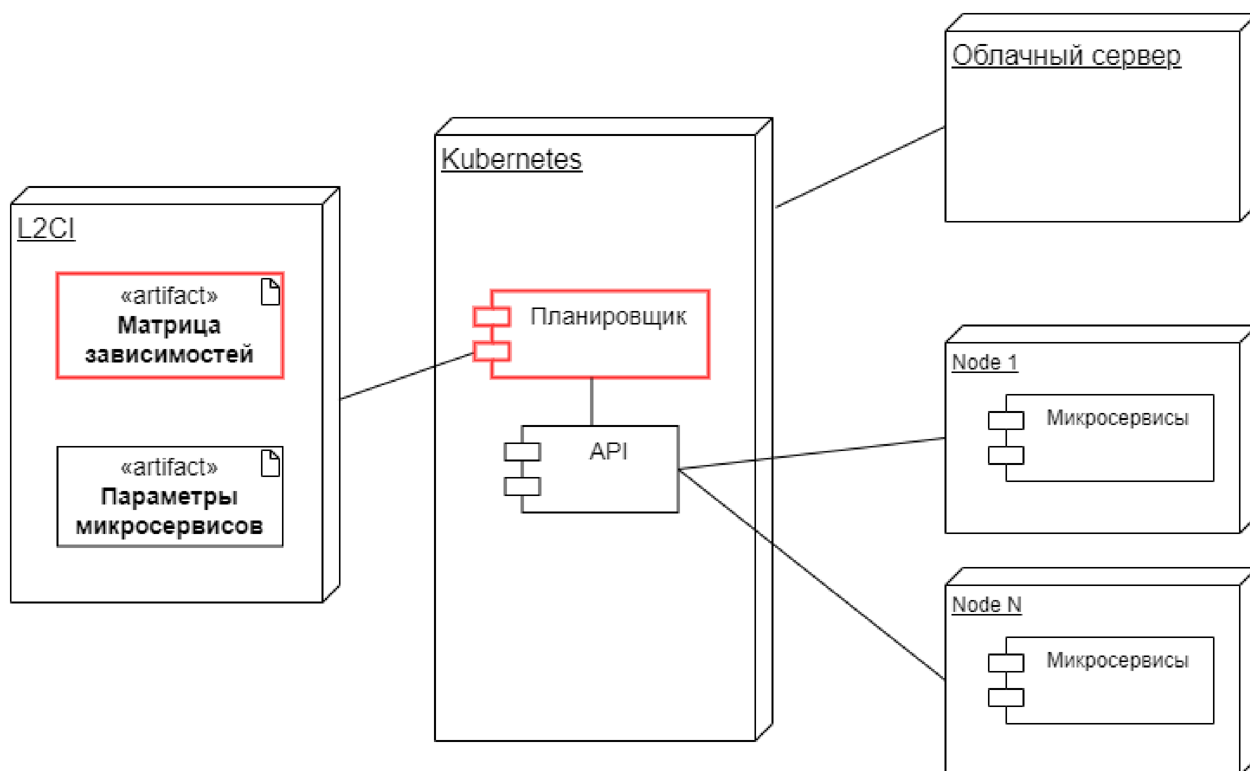

Рисунок 22 – диаграмма развертывания приложения

На этапе создания конфигураций в компоненте L2CI [30] выполняется сбор конфигурационных файлов каждого микросервиса, из которых строится матрица зависимостей между сервисами, а также вычисляются их параметры и подставляются значения в итоговую конфигурацию. В этом узле развертывания добавился новый артефакт «Матрица зависимостей», являющийся текстовым файлом, который содержит матрицу связности графа зависимостей микросервисов приложения и объем траффика, проходящего по этим связям.

Далее эти артефакты попадают в Kubernetes. Стандартный планировщик Kubernetes заменяется созданным в рамках данной работы планировщиком. Он, в свою очередь, распределяет сервисы по узлам (нодам) в соответствие с необходимыми ресурсами и межсервисным траффиком, полученным из узла L2CI, путем использования внутреннего API Kubernetes'а.

Красным цветом на диаграмме выделены новые компоненты, либо подвергшиеся изменению при выполнении работы.

4.3 Результаты внедрения планировщика

В результате внедрения нового планировщика изменению подверглась конфигурация микросервисов: для каждого микросервиса необходимо добавить 2 новых параметра, отвечающих за зависимости сервиса и объем траффика. По результатам апробации на определение значений этих параметров в среднем тратится около 0.25 человеко-часа при учете того, что каждый сервис сопровождается 1-2 разработчиками (один разработчик может отвечать за несколько сервисов), то есть всегда найдется разработчик, погруженный в специфику сервиса и не испытывающий необходимости в изучении документации для определения данных параметров. Таким образом, при разработке приложения, состоящего из сотни микросервисов, нужно затратить приблизительно 25 человеко-часов на модификацию конфигурации. Это около 20% от времени, которое обычно затрачивается на такую работу.

При этом за счет оптимизированного размещения микросервисов снизится нагрузка на инженеров, занимающихся развертыванием ИС, так как устраняется необходимость писать вручную правила размещения сервисов.

Исходя из этого, в процесс развертывания должен встроиться промежуточный этап создания дополнительной конфигурации, позволяющей планировщику учитывать траффик, что незначительно увеличит время разработки приложения относительно суммарно затраченных ресурсов. При этом вычисление значений переменных, содержащих объем траффика, можно автоматизировать и параметризировать, чтобы улучшить масштабируемость решения. Также матрицу зависимостей можно использовать повторно при условии, что в приложение не добавлялись новые микросервисы или не изменялись связи между существующими сервисами. Это позволяет поставлять приложение с заранее вычисленной матрицей, благодаря чему время развертывание приложения не только не изменится, но и сократится за счет использования нового алгоритма размещения.

В общей сложности внедрение нового алгоритма незначительно влияет на время разработки итогового продукта, но дает прирост по эффективности и гибкости развертывания.

По длительности развертывания новый алгоритм дает небольшое преимущество: в среднем общее время развертывания приложения целиком снизилось с 488 минут до 472 минут, то есть на 3%. При частичном развертывании приложения – от 10 до 30 процентов от общего количества сервисов – новый планировщик ускорил выполнение процесса на 1%, что можно считать незначительным приростом в производительности. На установке одиночных микросервисов значимых отличий от стандартного планировщика выявлено не было.

4.4 Выводы по разделу

Апробация полученного решения показала, что изменение процесса разворачивания приложения с учетом использования нового планировщика может улучшить производительность системы в целом, снижая время обработки запросов за счет размещения связанных микросервисов на одном узле кластера. Несмотря на увеличивающуюся разработку, такое решение может помочь проектам, состоящим из множества микросервисов.

Кроме того, такое решение может потребовать более квалифицированную команду разработчиков, т.к. для достижения наибольшей производительности требуется автоматизация создания конфигурации для микросервисов.

Заключение

В данном исследовании были рассмотрены проблемы размещения сервисов для микросервисной архитектуры в облаках и утилизация человеческих ресурсов во время разработки таких приложений. Чтобы найти высококачественный раздел сервисных приложений, было предложено два алгоритма разделения: бинарное разделение и K-разделение, которые основаны на хорошо разработанном алгоритме рандомизированного сжатия. Для эффективной упаковки приложения используется эвристика наиболее загруженной и учитывающей трафик в алгоритме упаковки. Путем корректировки порога α, обозначающего верхнюю границу потребности в ресурсах, можно найти лучшее решение для размещения сервисных приложений. В ходе оценки было показано, что представленные алгоритмы могут улучшить коэффициент успешного размещения приложений в кластере, при этом значительно увеличив долю совместного трафика (т. е. уменьшив межмашинный трафик). При оценке накладных расходов результаты показывают, что эти алгоритмы несут некоторые накладные расходы, но в приемлемое время. Также было выяснено, что предложенные алгоритмы практичны для реальных случаев использования.

Магистерская диссертация посвящена поиску путей оптимизации инфраструктуры приложений, созданных на базе микросервисной архитектуры, за счет изменения алгоритма размещения в кластере облачного сервера.

Выполненные в работе научные исследования представлены следующими основными результатами:

– проанализирован процесс развертывания приложения, а также алгоритмы, используемые для размещения микросервисов. Как показал анализ, главным недостатком известных алгоритмов является отсутствие учета межсервисного трафика при размещении, в связи с чем увеличивается общее время обработки запросов системой, когда

микросервисы вынуждены обмениваться сообщениями, будучи расположенными на разных узлах кластера. На основании проведенного анализа, в качестве решения проблемы был разработан алгоритм, позволяющий учитывать трафик при размещении сервисов таким образом, чтобы снизить сетевое взаимодействие между узлами кластера;

- усовершенствован метод разработки программного обеспечения путем изменения конфигурации микросервисов, что позволяет гибче настраивать систему и ускорить ее выдачу заказчику за счет заранее подготовленной конфигурации (матрицы зависимостей) для развертывания. Такая модификация позволяет повысить степень автоматизации в разработке, что позволяет снизить стоимость разработки продукта;
- проведена экспериментальная апробация усовершенствованного механизма разворачивания приложения с учетом межсервисного трафика, которая показала эффективность использования предложенного алгоритма несмотря на увеличившееся время развертывания, так как снижается время разработки приложения и повышается его гибкость.

Таким образом, в работе решена научно-практическая проблема оптимизации инфраструктуры микросервисного приложения.

Гипотеза исследования подтверждена.

Проведенная работа имеет большую значимость, так как благодаря использованию вышеописанных решений возможно повышение эффективности инфраструктуры микросервисных приложений.

Список используемых источников

1. Артамонов И.В. Разработка распределенных сервисноориентированных программных средств: учеб. пособие / И.В. Артамонов. — Иркутск: Изд-во БГУЭП, 2016. — 129 с.

2. Генкин Б.М. Анализ производительности труда / Б.М. Генкин. М.: НОРМАИНФРАМ, 2016. – 506 с.

3. Истигечева Е.В. Моделирование бизнес-процессов на примере модели «Сбыт» / Е.В. Истигечева, Т.Е. Григорьева, С.А. Панов // Текст: непосредственный // журнал «Таврический научный обозреватель», том 2, № 3, с. 55-59

4. Истигечева Е.В. Моделирование логических схем бизнеспроцессов / Е.В. Истигечева, Т.Е. Григорьева // Текст: непосредственный // журнал «Информатика и системы управления», том 2, № 48, с. 36-47

5. Липатова С.Е. Подходы к оптимизации размещения микросервисов в Kubernetes. Алгоритм NSGA-II. / С.Е. Липатова, Ю.С. Белов // Текст: непосредственный // сборник статей «Фундаментальные и прикладные исследования. актуальные проблемы и достижения», 2022, с. 22-24

6. Hu, Y.; Wang, J.; Zhou, H.; Martin, P.; Taal, A.; de Laat, C.; Zhao, Z. Deadline-Aware Deployment for Time Critical Applications in Clouds. // In Proceedings of the European Conference on Parallel Processing, Santiago de Compostela, Spain, 28 August–1 September 2017; Springer: New York, NY, USA, 2017; с. 345–357.

7. Koulouzis, S.; Martin, P.; Zhou, H.; Hu, Y.; Wang, J.; Carval, T.; Grenier, B.; Heikkinen, J.; de Laat, C.; Zhao, Z. Time-critical data management in clouds: Challenges and a Dynamic Real-Time Infrastructure Planner (DRIP) solution. // Concurr. Comput. Pract. Exp. 2019, e5269.

8. Hu, Y.; Zhou, H.; de Laat, C.; Zhao, Z. Ecsched: Efficient container scheduling on heterogeneous clusters. // In Proceedings of the European Conference

on Parallel Processing, Turin, Italy, 27–31 August 2018; Springer: New York, NY, USA, 2018; c. 365–377.

9. Grandl, R.; Ananthanarayanan, G.; Kandula, S.; Rao, S.; Akella, A. Multi-resource packing for cluster schedulers. //ACM SIGCOMM Comput. Commun. Rev.2015, 44, c. 455–466.

10. Gog, I.; Schwarzkopf, M.; Gleave, A.; Watson, R.N.; Hand, S. Firmament: Fast, centralized cluster scheduling at scale. // In Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), Savannah, GA, USA, 2–4 November 2016; c. 99–115.

11. Yusoh, Z.I.M.; Tang, M. A penalty-based genetic algorithm for the composite SaaS placement problem in the cloud. // In Proceedings of the IEEE Congress on Evolutionary Computation, Barcelona, Spain, 18–23 July 2010; c. 1–8.

12. Hajji, M.A.; Mezni, H. A composite particle swarm optimization approach for the composite saas placement in cloud environment. // Soft Comput. 2018, 22, c. 4025–4045.

13. Meng, X.; Pappas, V.; Zhang, L. Improving the scalability of data center networks with traffic-aware virtual machine placement. // In Proceedings of the 2010 IEEE INFOCOM, San Diego, CA, USA, 15–19 March 2010; pp. 1–9.

14. Wang, M.; Meng, X.; Zhang, L. Consolidating virtual machines with dynamic bandwidth demand in data centers. // Infocom 2011, 201, c. 71–75.

15. Goldschmidt, O.; Hochbaum, D.S. Polynomial algorithm for the k-cut problem. // In Proceedings of the 1988 29th Annual Symposium on Foundations of Computer Science, White Plains, NY, USA, 24–26 October 1988; c. 444–451.

16. Woeginger, G.J. There is no asymptotic PTAS for two-dimensional vector packing. // Inf. Process. Lett. 1997, 64, c. 293–297.

17. Karger, D.R. Global Min-cuts in RNC, and Other Ramifications of a Simple Min-Cut Algorithm. // SODA 1993, 93, c. 21–30.

18. Baldin, I.; Chase, J.; Xin, Y.; Mandal, A.; Ruth, P.; Castillo, C.; Orlikowski, V.; Heermann, C.; Mills, J. Exogeni: A multi-domain infrastructure-as-a-service testbed. // In The GENI Book; Springer: New York, NY, USA, 2016; c. 279–315.

19. Benson, T.; Anand, A.; Akella, A.; Zhang, M. Understanding data center traffic characteristics. // In Proceedings of the 1st ACM Workshop on Research on Enterprise Networking, Barcelona, Spain, 21 August 2009; ACM: New York, NY, USA, 2009; c. 65–72.

20. Huang, K.C.; Shen, B.J. Service deployment strategies for efficient execution of composite SaaS applications on cloud platform. // J. Syst. Softw. 2015, c. 107, 127–141.

21. Alicherry, M.; Lakshman, T. Network aware resource allocation in distributed clouds. // In Proceedings of the 2012 IEEE INFOCOM, Orlando, FL, USA, 25–30 March 2012, c. 963–971.

22. Kliazovich, D.; Bouvry, P.; Khan, S.U. DENS: Data center energy-efficient network-aware scheduling. // Clust. Comput. 2013, 16, 65–75.

23. Hightower, K.; Burns, B.; Beda, J. Kubernetes: Up and Running: Dive into the Future of Infrastructure // O'Reilly Media, Inc.: Champaign, IL, USA, 2017.

24. Ajiro, Y.; Tanaka, A. Improving packing algorithms for server consolidation. In Proceedings of the International CMG Conference, San Diego, CA, USA, 2–7 December 2007; Volume 253.

25. Stillwell, M.; Schanzenbach, D.; Vivien, F.; Casanova, H. Resource allocation algorithms for virtualized service hosting platforms. J. Parallel Distrib. Comput. 2010, 70, c. 962–974.

26. Thönes, J. Microservices. IEEE Softw. 2015, 32, 116.

27. Cerny, T.; Donahoo, M.J.; Trnka, M. Contextual understanding of microservice architecture: Current and future directions. ACM SIGAPP Appl. Comput. Rev. 2018, 17, 29–45.

28. Hasselbring, W.; Steinacker, G. Microservice architectures for scalability, agility and reliability in e-commerce. In Proceedings of the 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), Gothenburg, Sweden, 5–7 April 2017; pp. 243–246.

29. Dragoni, N.; Giallorenzo, S.; Lafuente, A.L.; Mazzara, M.; Montesi, F.; Mustafin, R.; Safina, L. Microservices: Yesterday, today, and tomorrow. // In

Present and Ulterior Software Engineering, Springer: New York, NY, USA, 2017, c. 195–216.

30. Leitner, P.; Cito, J.; Stöckli, E. Modelling and managing deployment costs of microservice-based cloud applications. // In Proceedings of the 9th International Conference on Utility and Cloud Computing; ACM: New York, NY, USA, 2016, c. 165-174.