САНКТ – ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Математико-механический факультет

Кафедра системного программирования

Разработка и оптимизация алгоритма репликации для облака Cirrostratus

Дипломная работа студента 461 группы

Кузнецова Кирилла Олеговича

Научный руководитель		С. В. Богатырев
	……	
Рецензент		А. Н. Косякин
	……	
“Допустить к защите”		д.ф.-м.н., проф. А.Н. Терехов
заведующий кафедрой	……	

Санкт-Петербург

2011

St. Petersburg State University

Faculty of Mathematics and Mechanics

Chair of Software Engineering

Developement and optimization of replication algorithm for Cirrostratus cloud

Bachelor's graduation paper by

Kirill Kuznetsov

Supervisor		S. V. Bogatyrev
	……	
Reviewer		A. N. Kosyakin
	……	
"Approved by"		Professor A. N. Terekhov
Head of Chair	……	

St. Petersburg, 2011

Оглавление

1 Введение

Концепция облачных вычислений (cloud computing) основывается на предоставлении пользователю вычислительных ресурсов по сети «по требованию». Вычислительные облака делятся на два основных вида: публичные и частные. Публичные облака предоставляют доступ к своим ресурсам по Интернет и в общем случае клиентом такого облака может быть любой пользователь или компания, частные облака используются одной компанией, корпоративными офисами, подразделениями, её партнерами и т. д. Доступ к таким облакам обычно осуществляется из внутренней сети и защищен брэндмауэром. Услуги облачных вычислений оказываются на нескольких уровнях: ПО как услуга (Software as a service – SaaS), платформа как услуга (Platform as a service – PaaS), инфраструктура как услуга (Infrastructure as a service – IaaS). Внутренняя архитектура облака состоит из двух основных частей: сервера с виртуальными машинами и распределенного хранилища данных, где располагаются диски виртуальных машин. Такое хранилище состоит из множества узлов, каждый из которых имеет собственную операционную систему и набор жестких дисков.

В настоящее время представлено множество решений в области распределённого хранения данных (например, линейка Sun Storage, продукты EMC – Clarion, Centera, системы хранения от IBM и т. д.), но на небольших предприятиях они являются редкостью из-за слишком высокой стоимости. Целью проекта Cirrostratus (https://github.com/realloc/cirrostratus) является создание надежной сети хранения данных для частного облака, работающей на недорогом, доступном оборудовании.

При организации распределенных хранилищ данных приходится решать следующие проблемы:

- **Производительность.**

  Объем информации в таких хранилищах достигает сотен и даже тысяч терабайт, расположенных на множестве устройств. Равномерное распределение данных устройствам хранения позволяет эффективно балансировать нагрузку и, таким образом, играет важную роль в обеспечении производительности. Для определения местоположения хранимой информации используются метаданные – информация о расположении данных, в связи с чем возникает ряд проблем. Во-первых, метаданные занимают дополнительное место, во-вторых, с ростом хранилища затраты на обработку метаданных могут расти катастрофически — появляются метаметаданные и т. д. Ситуация усугубляется тем, что иногда, для того чтобы добраться до совсем небольшого фрагмента данных, приходится хранить и обрабатывать значительно больший объем метаданных.

- **Надежность.**

  Для обеспечения требования надежности используется репликация данных (data replication) — при записи информации в хранилище помещается несколько копий, которые могут хранится на разных носителях, в разных узлах, а в некоторых случаях даже в разных центрах хранения и обработки данных (data ceneters). Таким образом при сбоях конкретных устройств пользовательская информация будет в полной сохранности.

- **Масштабируемость.**

  Добавление и удаление устройств хранения должно происходить в режиме реального времени, не препятствуя работе пользователей с облаком, при этом максимально сохраняя равномерность распределения

данных. В контексте масштабируемости также возникает проблема метаданных при изменении кластера — добавлении или удалении устройств. При каждом таком изменении в сбалансированно загруженных хранилищах должно происходить перемещение части данных, соответственно приходится перестраивать все связанные метаданные, что неблагоприятно сказывается на производительности хранилища.

В распределенном хранилище алгоритм репликации отвечает за распределение реплик блоков данных по кластеру в соответствии с определенными требованиями. В случае когда уровень репликации равен 1, алгоритм просто осуществляет распределение записанных данных по хранилищу. Помимо N-репликации — т. е. хранения точных копий данных, в хранилищах также используются методы избыточного кодирования. В частности эти методы реализованы для Cirrostrarus. И под репликами может пониматься не только точные копии блоков, но и фрагменты закодированных данных. Для каждого запроса на чтение или запись возвращается множество устройств, на которые нужно записать копии или откуда их можно прочитать. Проект находится на стадии разработки поэтому алгоритм репликации разрабатывается «с нуля». Разрабатываемый алгоритм должен эффективно решать проблемы производительности, надежности и масштабируемости. Также для алгоритма важно уметь определять оптимальные параметры для работы с определенными вариантами кластера.

2 Постановка задачи

Цель данной работы заключается в том, чтобы предложить алгоритм для распределения и репликации данных в облаке Cirrostratus. В рамках диплома должны быть решены следующие задачи:

1. Сформулировать требования к разрабатываемому алгоритму.
2. Проанализировать существующие механизмы репликации данных в распределенных хранилищах и выбрать наиболее подходящий.
3. Адаптировать выбранный алгоритм в соответствии с требованиями Cirrostratus.
4. Разработать и реализовать механизм для определения оптимальных параметров алгоритма для конкретных вариантов структуры кластера.
5. Осуществить интеграцию алгоритма в Cirrostratus.

3 Cirrostratus

Cirrostratus – это распределенное блочное хранилище для обслуживания дисков виртуальных машин в вычислительном облаке. Предназначается для использования в малых и средних частных облаках. Обеспечить доступность для массового использования предполагается за счёт использования недорогого и доступного оборудования.

В 2010 году Cirrostratus начал свое существование как Co-op проект компании EMC Corporation в котором принимали участи студенты СПбГУ и СПбГУ ИТМО. Начиная с 2011 года проект существует независимо от EMC. В настоящий момент Cirrostratus находится на стадии разработки. Проект создается под ОС Linux и реализуется на C.

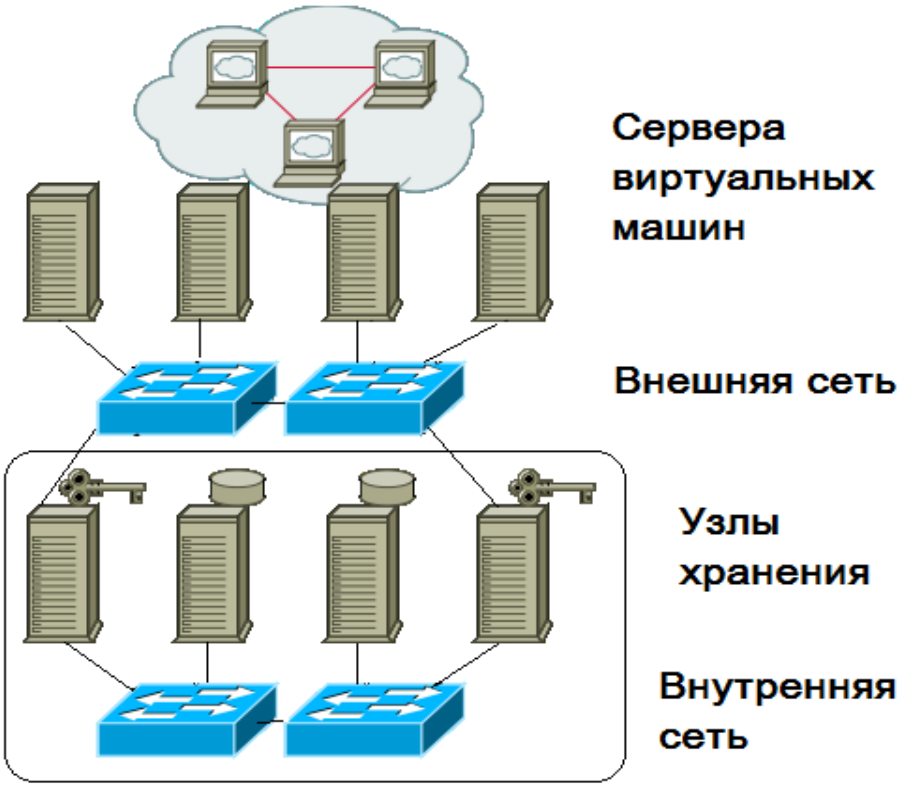

Рисунок 1: Архитектура Cirrostratus.

На рис. 1 изображена архитектура Cirrostratus. Виртуальные машины, используемые клиентами облака, работают на серверах под управлением

гипервизора. Распределенное хранилище обслуживает диски этих виртуальных машин и подключено к серверам виртуальных машин через внешнюю сеть посредством Gigabit Ethernet. Пользователю предоставляются виртуальные диски с определенной заявленной вместимостью, однако физически занимают лишь необходимый объем, расширяясь в случае надобности. Само хранилище состоит из набора узлов хранения, на каждом узле работает несколько дисков.

Запросы на чтение и запись блоков от виртуальных машин обрабатываются на выделенных узлах доступа (access node). Все блоки виртуальных дисков однозначно определяются идентификатором диска и сдвигом относительно начала. При записи или чтении, к блоку пришедшему на запись применяется выбранный алгоритм избыточного кодирования, если это определено политикой хранения. Далее записываемый фрагмент данных сериализуется на блоки, кратные 512 байт. Затем описываемый в этой работе алгоритм репликации, в соответствии с определенными правилами, возвращает набор устройств, на которых будут размещены реплики или откуда будет осуществляться чтение. На конкретном узле хранения реплики так же адресуются идентификатором виртуального диска и смещением относительно начала. При этом в Cirrostratus используется менеджер логических томов (LVM - Logical Volume Manager), и вместо того, что бы использовать физические диски, чтение и запись реплик осуществляется на расширяемые логические тома. Для записи и чтения блоков как во внутренней так и во внешней сети используется модифицированный в соответствии с требованиями Cirrostratus протокол AoE (ATA over Ethernet) [9]. Для поддержания сети могут использоваться самые обычные коммутаторы.

4 Разработка и реализация

4.1 Требования к алгоритму

Прежде чем осуществлять разработку алгоритма необходимо сформулировать требования, которым он должен соответствовать. Для того чтобы эффективно использовать ресурсы распределенного хранилища и обеспечить высокую производительность необходимо распределять нагрузку на узлы кластера сбалансировано в соответствии с возможностями устройств хранения. Следовательно, нужно распределять реплики записываемых блоков равномерно с учетом таких характеристик дисков хранения как вместимость и производительность. Как было сказано во введении, использование метаданных для адресации блоков влечет за собой проблемы производительности и дополнительные затраты памяти, поэтому для алгоритма репликации важно минимизировать их использование. Кластер Cirrostratus расширяем и не статичен, поэтому алгоритм должен учитывать такие изменения, как добавление и удаление дисков и узлов хранения, а также ситуации сбоев. В описанных ситуациях для поддержания сбалансированного заполнения устройств хранения приходится осуществлять перераспределение блоков, что создает дополнительную нагрузку на хранилище. Исходя из этого, алгоритм должен обеспечивать как можно меньшую миграцию данных. Так как Cirrostratus разрабатывается для массового использования на кластерах с различными структурами, необходимо обеспечить гибкость и настраиваемость алгоритма.

Таким образом, исходя из вышесказанного алгоритм репликации в Cirrostratus должен соответствовать следующим требованиям:

1. Равномерное заполнение устройств хранения с учетом характеристик дисков.
2. Минимальное использование метаданных
3. Минимальная миграция данных в случае изменения кластера
4. Гибкость настройки для работы в различных конфигурациях кластера

4.2 Существующие подходы

4.2.1 Сравнительный анализ

Проблема размещения данных в хранилищах широко изучалась в контексте распределенных файловых систем[7][8], в этих случаях используется псевдослучайная аллокация, однако отсутствует поддержка сбалансированной загрузки устройств, более того, для адресации данных используются директории метаданных.

Одним из подходов к проблеме репликации в распределенных хранилищах без использования метаданных является устойчивое хеширование [3]. Каждому устройству хранения сопоставляется определенный интервал (или несколько интервалов) из множества значений соответствующей хеш-функции. И при записи в хранилище для каждого блока данных вычисляется хеш а затем он помещается на соответствующее устройство. При этом при добавлении или удалении секторов, необходимо переместить минимально число блоков. Это достаточно общий подход, изначально разработанный для запросов на изменяющемся множестве серверов, сейчас он используется в системах хранения[6] и в распределенных базах данных[13]. Однако метод устойчивого хеширования не учитывает разные характеристики производительности и вместимости устройств хранения и не гарантирует хорошо сбалансированного распределения.

Бринкманном[1] был предложен алгоритм для размещения реплик в неоднородном по характеристикам устройств кластере за O(n), однако без поддержки добавления и удаления дисков или узлов хранения. Другой подход - SCADDAR (SCAling Disks for Data Arranged Randomly [12]) адаптируется к изменениям кластера, однако обеспечивает ограниченный набор стратегий репликации.

Наиболее близко поставленным требованиям соответствует семейство

алгоритмов RUSH[2][5] и их обобщение — алгоритм CRUSH[4]. RUSH/CRUSH вместо метаданных используют отображающую функцию, учитывают различные возможности производительности и вместимости для дисков, используя веса. А также поддерживают эффективное удаление и добавление элементов кластера. При этом CRUSH, используя полезные элементы RUSH, добавляет гибкий механизм правил репликации. Так как эти алгоритмы в целом соответствуют требованиям, предъявляемым к алгоритму репликации в Cirrostratus, было решено взять их за основу. Ниже семейство алгоритмов RUSH и CRUSH будут рассмотрены подробнее.

4.2.2 Семейство алгоритмов RUSH

RUSH (Replication Under Scalable Hashing) это семейство алгоритмов, каждый из которых на основе хеша объекта и степени репликации возвращает набор дисков, где будут размещены реплики. При этом все реплики размещаются псевдослучайно. При размещении реплик алгоритмы RUSH не используют директории метаданных. Кластер рассматривается как набор из субкластеров, каждый из которых имеет свой вес. Веса субкластеров могут отличаться, веса дисков внутри одного субкластера одинаковы. Все алгоритмы имеют различные показатели скорости выбора диском под реплики и различными затратами на ребалансировку в хранилище после добавления и удаления дисков.

Ниже приведено описание общих идеи алгоритмов из семейства RUSH, детальное изложение алгоритмов представлено в работах авторов [2][5].

4.2.2.1 $RUSH_P$

Для выбора реплики используется список субкластеров, упорядоченный по давности добавления. Для выбора используется хеш блока, для которого размещаются реплики, который лежит в промежутке [0, 1). Хеш сравнивается с отношением веса текущего субкластера к суммарному весу ранее добавленных.

Если значение хеша меньше тогда реплика помещается в это субкластера. Для выбора устройства внутри субкластера используется функция $f(x, r) = z + rp \pmod m$, где x – идентификатор блока, r – номер реплики, z – значение хеша блока, p – случайное простое число, а m - число дисков в субкластере.

```
RUSHP (x, r)
for all subclusters j
        z = hash(x,j,r);
        z *= w(j);
        if (z < W(j))
                return  ( z + r * p ) mod m
```

Псевдокод для $RUSH_P$: w(j) – вес субкластера,W(j) – суммарный вес всех ранее добавленных субкластеров

Этот алгоритм работает за O(n) и обладает оптимальными показателями миграции при добавлении новых субкластеров, однако абсолютно неэффективен при удалении субкластеров и дисков.

4.2.2.2 $RUSH_R$

$RUSH_R$ для выбора устройств для размещения реплик также использует упорядоченный по давности добавления список субкластеров. А устройства выбирает на основе хеша блока и того же соотношения весов, что и $RUSH_P$. Однако эти величины передаются как параметры взвешенного гипергеометрического распределения, а выбор всех реплик происходит одновременно. Для каждого субкластера определяется число реплик k, которые будут там размещены, а затем происходит случайный выбор k дисков из n, где n - общее число дисков в субкластере.

Алгоритм работает за линейное время и дает оптимальные результаты при добавлении субкластера. При удалении субкластеров или дисков перемещается близкое к оптимальному число блоков.

```
RUSHR(x, R)
result := {}
for all subclusters j
        u =  H (R, W(i), w(j), ws(j), x )
        if ( u > 0)
                y : = choose( u, n(j))
                R := R – u
                append y to result
                if R = 0
                        return result
```

Псевдокод для RUSHr : ws(j) - вес одного диска, n(j) - число дисков в субкластере

4.2.2.3 $RUSH_T$

$RUSH_T$ использует в качестве структуры данных бинарное дерево. В листьях дерева находятся устройства. Для всех узлов определен вес, который является суммой весов детей. Все узлы помечены уникальным идентификатором. Алгоритм начинает работу в корне, и на каждом шаге выбирает одно поддерево, и повторяет шаги до тех пор, пока не выбран лист. Выбор поддерева осуществляется на основе хеша объекта, который параметризован идентификатором узла дерева. Значение хеша сравнивается с отношением весов левого поддерева и суммарного веса левого и правого поддеревьев. Если оно меньше, выбирается левой поддерево, иначе правое. Устройство в субкластере выбирается аналогично $RUSH_P$.

Алгоритм работает за O(log n). Показывает близкие к оптимальным результаты как при добавлении дисков.

```
RUSHT(x, r):
while node is not subcluster
        if hash(x,node.index, r)  * node. totalWeight < node.left.totalWeight:
                node := node.left
        else:
                node : =node.right
return ( z + r * p ) mod m
```

Псевдокод для RUSHt

4.2.3 Алгоритм CRUSH

CRUSH (Controlled Replication Under Scalable Hashing) был разработан как обобщение семейства алгоритмов RUSH, для использования в объектной файловой системе CEPH [10]. Используя полезные достижения RUSH, CRUSH предлагает большую гибкость, несколько улучшенную производительность, решая также некоторые проблемы надежности.

4.2.3.1 Иерархическая карта сети

В основе работы алгоритма лежит иерархическая карта сети, которая представляет из себя дерево, разделенное на несколько уровней. В узлах дерева находятся контейнеры (buckets), которые содержат в себе контейнеры нижнего уровня или диски, которые располагаются в листьях дерева. Каждый из контейнеров определяет алгоритм, по которому псевдослучайно выбирается нужное число его детей. Этот выбор задается правилами, которые определяет, сколько контейнеров или дисков нужно выбрать на каждом уровне. Все диски имеют вес, который отражает вместимость диска или его характеристики производительности. Вес контейнера складывается из весов элементов, которые он содержит. Распределение реплик по карте сети осуществляется с учетом этих весов. Все элементы карты имеют свои идентификаторы.

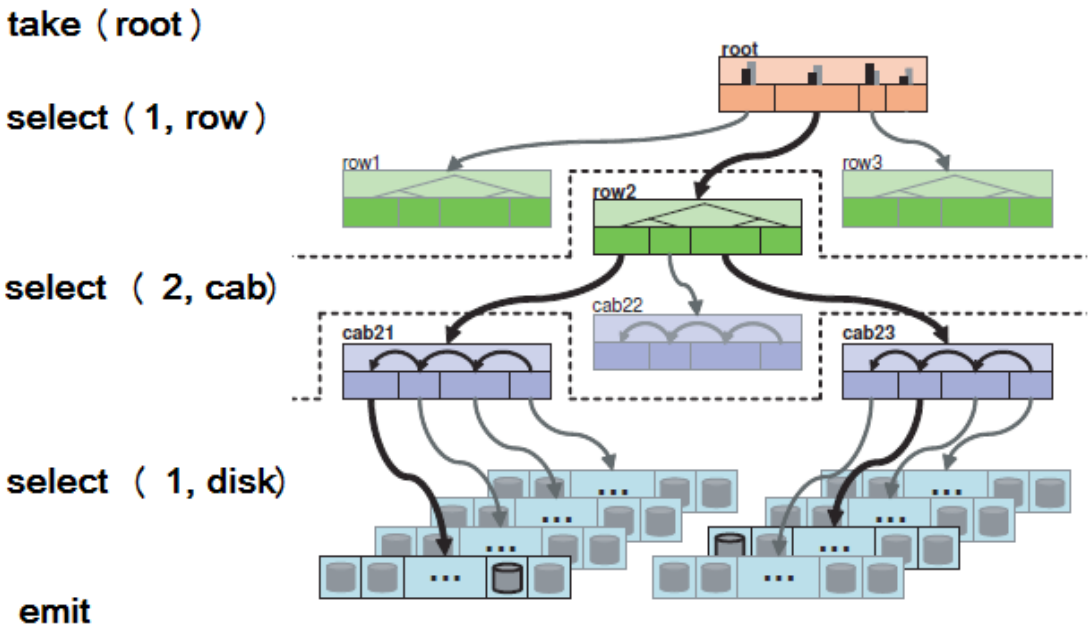

Рисунок 2: Правила и карта сети

На рис. 2 приведен пример применения правил к карте сети. В данном случае карта состоит из трех уровней контейнеров: корень, ряд, стойка. Итог применения правил — набор из двух дисков, которые находятся на двух разных стойках в одном ряду.

4.2.3.2 Контейнеры

Существует четыре вида контейнеров: List, Tree, Uniform и Straw. Контейнеры List и Tree используют алгоритмы выбора субкластеров из $RUSH_P$ и $RUSH_T$ соответственно. Uniform реализует уровень выбора устройства в субкластере из этих же алгоритмов. Uniform состоит только из элементов одного веса. Алгоритм работает за O(1), однако не поддерживает добавление или удаление элементов, что влечет полное перераспределение блоков. Так для Cirrostratus требуется поддержка изменений кластера на всех уровнях, этот контейнер не планируется использовать.

Straw-контейнер был разработан непосредственно автором CRUSH. Алгоритм Straw имитирует процесс вытягивания соломинок. При таком подходе все устройства «соревнуются» за право хранить реплику. Алгоритм

проходит по всем элементам контейнера, для каждого элемента считается хеш от идентификатора блока *x*, номера реплики *r* и позиции устройства *i* - «соломинка». Затем это значение умножается на модификатор f(w(i)), который учитывает вероятность выбрать элемент с меньшим модификатором. Реплика размещается на устройство с наибольшей «соломинкой». Модификаторы вычисляются при инициализации контейнера. Алгоритм их вычисления изложен в [5]. Выбор в контейнере происходит за O(n), при изменении карты сети происходит перераспределение минимального числа блоков.

```
Straw(x, r)
high_draw := 0
result
for all items i:
        draw := hash(x, r, i) * f(w(i))
        if ( draw > high_draw):
                high_draw := draw
                result := I
return result
```

Псевдокод для Straw

4.2.3.3 Обработка сбоев, коллизий и перегрузок.

При выборе элемента контейнера в CRUSH существуют три ситуации ситуации, при которых необходимо выбрать элемент заново: когда выбранный элемент уже используется для размещения другой реплики, когда он помечен как сбойный или перегруженный. Сбойные и перегруженные элементы остаются на карте, чтобы избежать лишних перемещений данных.

При работе CRUSH в качестве «реплик» могут рассматриваться как непосредственно копии записываемого блока, так и части схемы избыточного кодирования. Для поддержки обоих случаев в CRUSH используются две стратегии выбора: «First N» - когда нам не важен порядок выбора элементов, и

стратегия, при которой каждой конкретной реплике соответствует конкретный набор устройств.

На рис.3 изображен пример применения двух этих стратегий. Слева — номера реплик для выбранных элементов просто «сдвигаются», во втором случае для каждой реплики существует вероятностно независимая последовательность элементов. В случае сбоя, устройство просто заменяется следующим из этой последовательности.

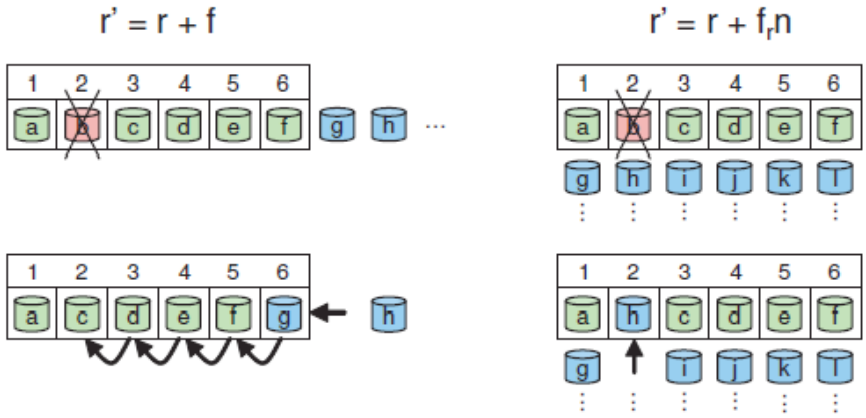

Рисунок 3: Стратегии обработки сбойных устройств.

4.3 Контейнер RUSH_R

Как было сказано выше, CRUSH использует идеи $RUSH_P$ и $RUSH_T$. При этом не используется $RUSH_R$. Автор CRUSH объясняет это тем, что алгоритм $RUSH_R$ не позволят сопоставлять определенным репликам определенные устройства, тем самым не поддерживая избыточное кодирование. Однако, как видно на рис. 4, алгоритм показывает в целом хорошие результаты при миграции данных: минимальное перемещение блоков при добавлении устройств и , а скорость размещения реплик выше чем в $RUSH_P$ за счёт того, что все реплики размещаются одновременно, а не последовательно. Так как одним из требований к алгоритму репликации в Cirrostratus была гибкость алгоритма, было принято решение реализовать контейнер на основе $RUSH_R$.

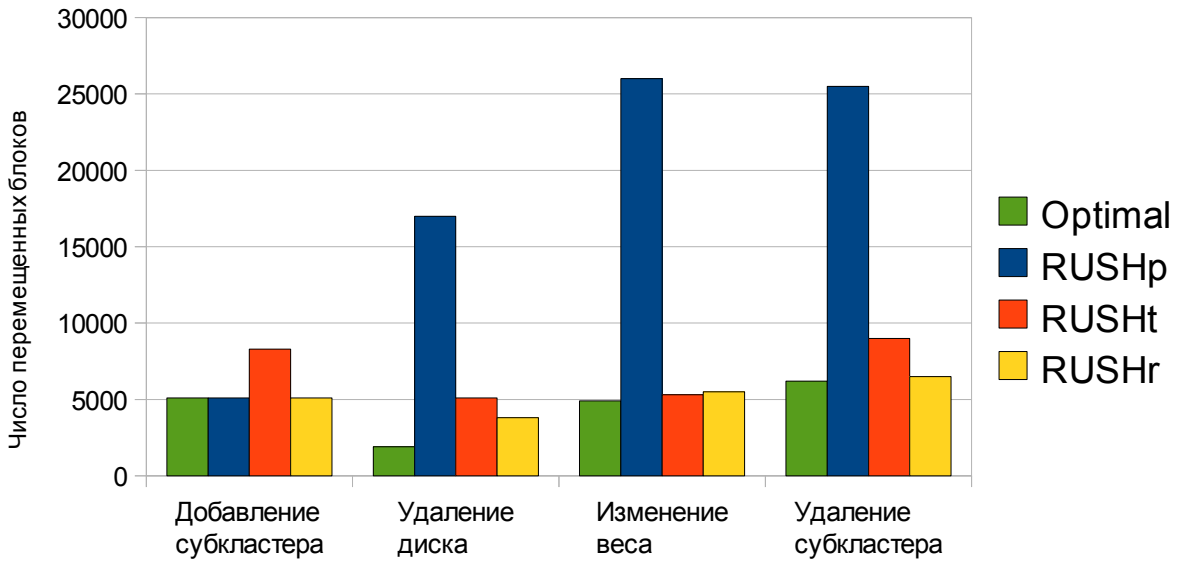

Рисунок 4: Миграция данных при изменениях в кластере для 100000 блоков

4.3.1 Особенности реализации

Реализация CRUSH для CEPH спроектирована таким образом, что в случае необходимости можно добавлять реализации новых контейнеров, использующих другие алгоритмы. Для этого необходимо определить структуру

в которой буду хранится данные, специфичные для реализуемого контейнера а ссылка на структуру *struct crush_bucket*, которая хранит общие для контейнеров параметры, такие как размер контейнера, вес, используемая хеш функция, и др. По умолчанию в CRUSH для всех контейнеров используется хеш-функция Дженкинса[11]. В случае $RUSH_R$ для каждого элемента хранится его вес и суммарный вес всех ранее добавленных элементов, т. е. элементов с меньшим индексом. Также для контейнера необходимо определить функции для создания, удаления (реализация осуществляется на C, поэтому нельзя забывать об освобождении памяти) и, наконец, функцию, реализующую сам алгоритм выбора элементов.

Для оригинальных контейнеров реплики размещаются последовательно при этом обработка коллизий осуществляется вне алгоритма выбора. В реализуемом алгоритме все реплики размещаются одновременно, т. е. за один проход, и поэтому обработка коллизий должна происходить внутри самого контейнера. Ситуация, когда для реплики выбирается уже занятый другой репликой элемент, для $RUSH_R$ может возникнуть только в том случае, когда алгоритм не смог разместить все реплики за один проход, а вероятность это близка к нуля. Однако такие ситуации все равно нужно обрабатывать.

В оригинальном алгоритме $RUSH_R$ для сбалансированного размещения реплик используется имитация выборки из взвешенного гипергеометрического распределения[5], которое парметризовано числом реплик, которые осталось разместить, весом текущего субкластера и суммарным весом всех субкластеров добавленных ранее . Результат выборки – число дисков из текущего субкластера, на которых будут размещены реплики. Затем осуществляется выбор нужного числа дисков из субкластера. Для контейнера $RUSH_R$ последний шаг алгоритма не реализуется, так как за это должны отвечать контейнеры нижестоящего уровня. И распределение реплик в них задается правилами для соответствующего уровня. Таким образом проходя список всех элементов

алгоритм контейнера определят, использовать этот элемент или нет, вместо того чтобы определять сколько реплик туда нужно разместить.

4.3.2 Сравнение с другими контейнерами

Для реализованного контейнера были проведены тесты по измерению скорости выбора устройств для расположения реплик а так же по миграции при изменении кластера.

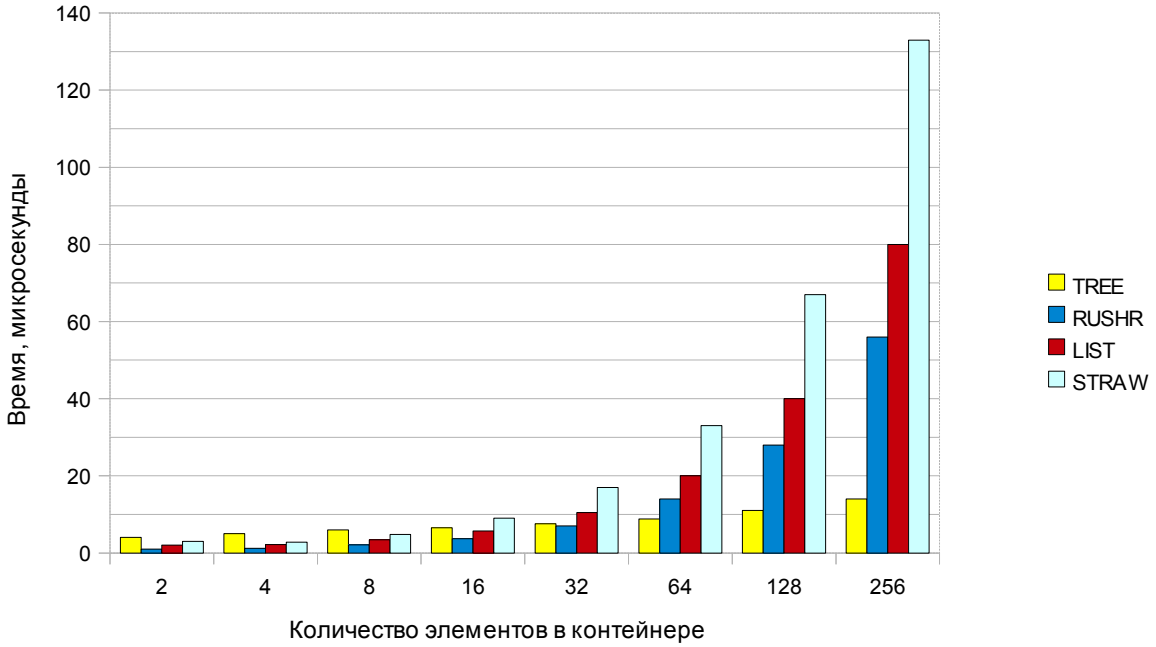

Рисунок 5: Временные затраты на выбор элементов в контейнерах

Данные по измерению скорости изображены на рис.5. Среднее время размещения двух реплик проводились для 1 000 000 блоков на контейнерах различных размеров. Для контейнеров достаточно малых размеров, $RUSH_R$ показывает лучшее время, т. к. с вероятностью близкой к 1 все реплики размещаются за один проход, для других алгоритмов существует высокая вероятность выбрать элемент повторно, что влечет за собой повторное размещение. С ростом числа элементов, алгоритм начинает уступать по скорости Tree-контейнеру. Что объясняется тем, что сложность Tree - O(log n), а

для остальных контейнеров она линейна. При этом временные затраты в целом соответствуют результатам алгоритма $RUSH_R$ в сравнении с $RUSH_P$ и $RUSH_T$.

На рис.6 отражена относительная эффективность реорганизации контейнеров. Для это используется фактор миграции $f = m_{actual}/m_{optimal}$, где 1 соответствует оптимальному числу перемещаемых блоков. Реализованный контейнер показывает оптимальные результаты перемещения блоков при добавлении элементов и близкие к оптимальным — при удалении. При этом наиболее близкий по своей организации контейнер List и самый быстрый контейнер Tree, показывают рост фактора миграции для удаления старых устройств (т. е. элементов из середины или конца контейнера) с возрастанием размера кластера.

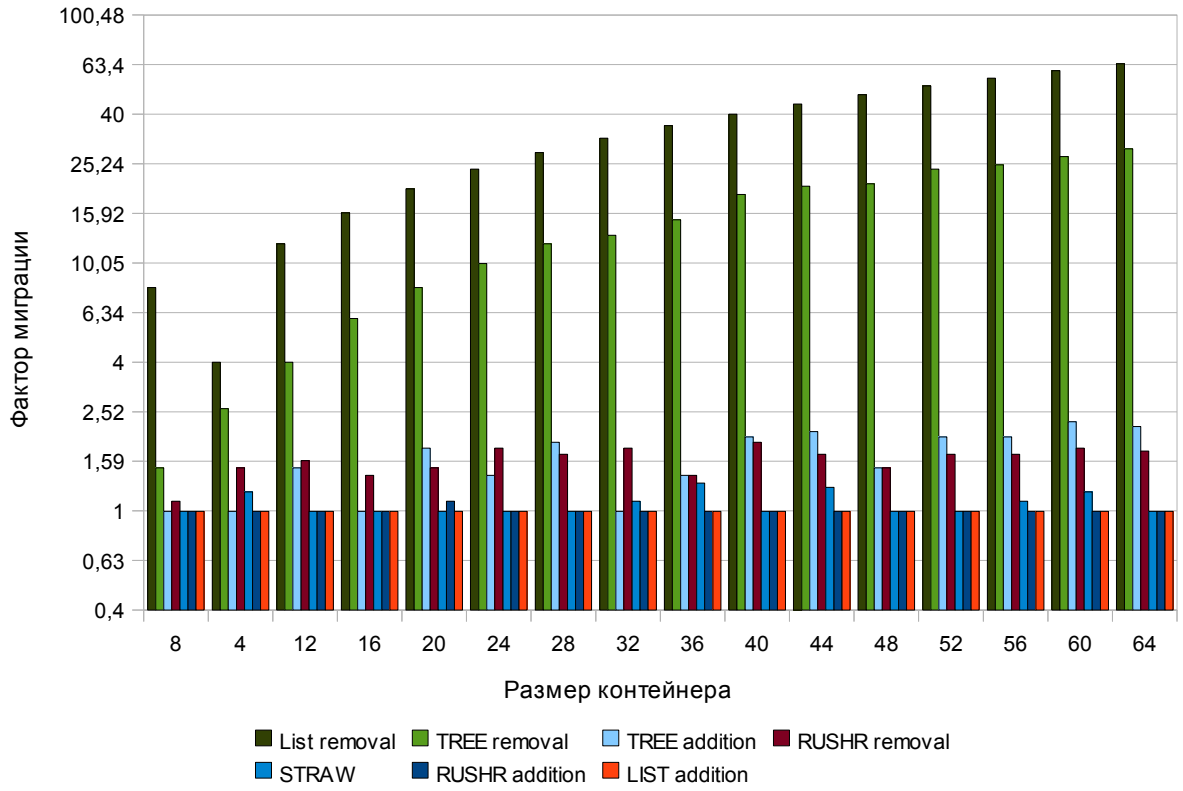

Рисунок 6: Миграция блоков

Таким образом, можно подвести итог, по скорости размещения реплик контейнер $RUSH_R$ уступает только Tree-контейнеру по скорости, превосходя

при этом его по показателям миграции. Использование данного контейнера будет целесообразным для тех уровней кластера, где происходит частые изменения в составе устройств и не используется политика избыточного кодирования. Непосредственно в Cirrostratus этот контейнер может использоваться на уровне узлов хранения в растущем кластере, где происходит частое удаление и добавление дисков.

5 Оптимальные параметры алгоритма

Cirrostratus разрабатывается таким образом, чтобы его можно было использовать на самых различных конфигурациях кластера. Поэтому гибкость была одной из причин выбора CRUSH качестве базового алгоритма репликации. Для разных вариантов хранилища могут быть использованы различные карты сети и правила репликации. Следовательно, важно определять какое возможное сочетание параметров алгоритма будет оптимальным. Для этого был разработан механизм для тестирования.

5.1 Описание теста

На входе тестирующая программа получает данные о кластере: количество узлов хранения, количество дисков на них, а так же веса дисков, отражающие их вместительность и производительность. Устройствам, обладающим лучшими характеристиками производительности присваевается вес 1, чем меньше вес — тем хуже характеристики. Перегруженные устройства помечаются весом больше единицы, сбойные нулем. На основе данных кластера генерируются различные варианты карт. Конкретный вариант карты представляет собой определенное сочетание контейнеров для каждого уровня и набор правил, определяющие число выбранных реплик на каждом уровне и политку обработки коллизий, сбоев и переполнений устройств. Для каждой карты программа замеряет среднюю скорость размещения реплик для заданного числа блоков, также количество перемещаемых блоков в случае добавления и удаления узлов, а также оценивает равномерность размещения блоков.

5.2 Оценка миграции

Для оценки миграции блоков для тестируемой карты генерируются дополнительные карты — с удаленным диском, удаленным узлом,добавленным диском и добавленным узлом. Затем для каждого блока происходит размещение

реплик на всех этих картах, после чего сравниваются массивы, хранящие результаты. Для контейнеров List и $RUSH_R$ число перемещаемых блоков различается в зависимости от места элемента в вышестоящем контейнере. Поэтому рассматриваются варианты с удалением из начала, середины и конца контейнера. Перемещение блоков внутри одного узла в меньшей степени влияет на работу всего кластера чем миграция между различными узлами, поэтому при оценке миграции два этих случая рассматриваются отдельно.

6 Заключение

В рамках этой работы были достигнуты следующие результаты:

1. Сформированы требования к алгоритму репликации.
2. Проанализированы существующие подходы и на основе требований выбран алгоритм CRUSH. Алгоритм эффективно решает проблемы производительности, надежности и масштабируемости в условиях хранилища Cirrostratus.
3. Реализована модификация алгоритма CRUSH, которая позволяет получить оптимальные результаты для распределения реплик в узлах хранения в растущем кластере, в том случае когда для хранения данных не используется политика избыточного кодирования.
4. Разработан и реализован механизм тестирования карт сети для определения оптимальных конфигураций CRUSH для различных вариантов кластера.
5. Реализованный алгоритм интегрирован с остальными модулями Cirrostratus.

7 Литература

[1] *Brinkmann, A., Salzwedel, K., Scheideler, C.* 2000. Efficient, distributed data placement strategiesfor storage area networks. In Proceedings of the 12th ACM Symposium on Parallel Algorithms and Architectures (SPAA), ACM Press, 119–128. Extended Abstract.

[2] *Honicky, R. J., Miller, E. L.* 2004. Replication under scalable hashing: A family of algorithms for scalabledecentralized data distribution. In Proceedings of the 18thInternational Parallel & Distributed Processing Symposium (IPDPS 2004), IEEE

[3] *Karger, D., Lehman, E., Leighton, T., Levine, M.,Lewin, D., Panigrahy, R.* 1997. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In ACM Symposium on Theory of Computing, 654–663

[4] *Weil, S. A., Brandt, S. A., Miller E. L., Maltzahn C.,* 2006 CRUSH: Controlled, Scalable, Decentralized Placement of Replicated Data, Proceedings of SC ‘06

[5] *Honicky, R. J., Miller, E. L.* RUSH: Balanced, Decentralized Distribution for Replicated Data in Scalable Storage Clusters

[6] *Tang, H., Gulbeden, A., Zhou, J., Strathearn, W.,Yang, T., Chu, L*. 2004. A self-organizing storage cluster for parallel data-intensive applications. In Proceedings of the 2004 ACM/IEEE Conference on Supercomputing (SC ’04)

[7] *Schmuck, F., Haskin, R.* 2002. GPFS: A shareddisk file system for large computing clusters. In Proceedings of the 2002 Conference on File and Storage Technologies (FAST), USENIX, 231–244

[8] *Saito, Y., Frølund, S., Veitch, A., Merchant, A., Spence, S.* 2004. FAB: Building distributed enterprise disk arrays from commodity components. In Proceedings of the 11th International Conference on Architectural Support for Programming

Languages and Operating Systems (ASPLOS), 48–58.

[9] *Coile, B., Hopkins, S.,* The ATA over Ethernet Protocol, Coraid, Inc. http://www.ofek.biz/pdfs/documentation/AoEDescription.pdf

[10] *Weil, S. A.* Ceph: Reliable, Scalable, and High-Performance Distributed Storage. Ph.D. thesis, University of California, Santa Cruz, December, 2007. 82-115

[11] *Jenkins, R. J.,* 1997. Hash functions for hash table lookup. http://burtleburtle.net/bob/hash/evahash.html.

[12] *Goel, A., Shahabi, C., Yao, D. S.-Y., Zimmerman, R*. 2002. SCADDAR: An efficient randomized technique to reorganize continuous media blocks. In Proceedings of the 18th International Conference on Data Engineering (ICDE '02), 473–482

[13] Project Voldemort
http://project-voldemort.com/design.php