Министерство науки и высшего образования Российской Федерации Федеральное государственное бюджетное образовательное учреждение высшего образования «Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)» (МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ Информатика и системы управления

КАФЕДРА Теоретическая информатика и компьютерные технологии

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА К КУРСОВОЙ РАБОТЕ ПО КУРСУ АЛГОРИТМЫ КОМПЬЮТЕРНОЙ ГРАФИКИ НА ТЕМУ:

Визуализация работы объектного хранилища Ceph

Студент _____ _____

подпись, дата *фамилия, и.о.*

Научный руководитель _____ _____

подпись, дата *фамилия, и.о.*

2024 г

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Компаниям необходимо хранить пользовательские данные для предоставления им своих услуг. Небольшие и средние компании способны справляться с нагрузкой уместив всё в рамках одного или нескольких серверов.

Для крупных компании ситуация сложнее. Они оперируют во многих регионах. Для этого требуется уметь размещать и обрабатывать данные большого потока пользователей в нескольких гео-распределенных датацентрах. Кроме того многие распространенные виды данных, такие как видео- или аудио-записи, нельзя хранить в табличном виде, что ещё больше усложняет дело. Такие ограничения накладывают определенные требования на систему хранения данных.

Руководствуюсь такими ограничениями был разработан специальный тип хранилищ данных — распределенные объектные хранилища. В условиях стремительного роста объемов данных и их разнообразия, объектные хранилища становятся неотъемлемой частью современной информационной инфраструктуры. Они представляют собой эффективное решение для хранения, управления и обработки больших массивов данных, которые характерны для бизнеса, науки и технологий. Объектные хранилища обеспечивают высокую масштабируемость, доступность и надежность, что делает их особенно важными для организаций, стремящихся оптимизировать свои процессы и ресурсы.

Объектные хранилища могут быть полезны не только крупным компаниям — исследовательские учреждения также находят им применение. В научной среде объектные хранилища позволяют эффективно обрабатывать и анализировать большие объемы данных, получаемых в ходе экспериментов и исследований. Так, например, хранилище Ceph используется в CERN — ведущем институте по исследованию физики частиц [1].

Многие компании готовы представить собственные разработки в этом направлении как сервис: Google Cloud Storage [2], Amazon S3 [3], Yandex Object Storage [4]. Существуют и решения, которые можно развернуть

собственноручно. К такому типу относится Ceph [5]. Работа внутреннего устройства этого хранилища будет смоделирована в настоящей работе.

1. Основные понятия

1.1. Устройство объектных хранилищ

Объектные хранилища — это системы хранения данных, которые предназначены для управления большими объемами неструктурированных данных, таких как изображения, видео, документы и резервные копии. Они представляют собой альтернативу традиционным файловым системам и блочным хранилищам.

Единицей хранимой информации в объектном хранилище, неудивительно, является объект. Обычно, такие хранилища ничего не знают о структуре хранимой информации в объектах, и обходятся с ними как с набором байтов. Этот подход схож с тем что используется в распространенных файловых системах в пользовательских операционных системах. Обязанность понимания структуры объекта лежит на клиенте. Каждому объекту присуждается его уникальный идентификатор для дальнейшей его адресации клиентом и дополнительная информация об объекте, необходимая для поддержания хранилища в рабочем состоянии. Обычно она представляется в формате ключ-значение.

В объектном хранилище отсутствует иерархическая структура, привычная для обычных файловых систем с их иерархией папок. Поддержка такой иерархии существенно увеличило бы накладные расходы на хранение данных. Эта особенность объектных хранилищ существенно упрощает горизонтальную масштабируемость кластера на несколько узлов, а также значительно упрощает репликацию данных на соседние узлы, делая систему более устойчивой к потере узлов, которая является обыденностью на больших масштабах — по словам почетного сотрудника Google (англ. Google fellow) Джефа Дина [6], в первый год работы датацентра число отказов жестких дисков может измеряется в тысячах.

1.2. Устройство Ceph

1.2.1. Особенности хранения в Ceph

Пусть задана такая иерархия кластера: в данном датацентре имеется в распоряжении два сервера, у каждого из которых по два жестких диска. Она представлена на рисунке 1.

рис. 1: Иерархия кластера

Любую такую иерархию можно интерпретировать как дерево, где корень — это датацентр, а листья — устройства памяти. Определим *область сбоев* (англ. failure domain) кластера как поддерево такой иерархии. При описании топологии кластера, Ceph умеет брать во внимание области сбоев чтобы обеспечить большую отказоустойчивость. Это используется в репликации: например, можно указать Ceph чтобы копируемые объекты находились на разных серверах.

Все объекты в Ceph хранятся в пулах (англ. pool). Пул — это способ логического разделения объектов. Каждый пул поделен на заранее известное количество групп размещений (англ. placement group). Им нельзя дать конкретное имя — они проиндексированы от 0 до N-1, где N — общее количество таких групп. За каждой такой группой размещения закрепляется Ceph OSD (Object Storage Daemon) — процесс операционной системы, работающий на узле кластера, управляющий чтением и записью на подконтрольное устройство хранения, а так же следящий за целостностью хранимой на нем информацией. Распределение OSD по группам размещения определяется основываясь на областях сбоев с помощью алгоритма CRUSH [7].

По сути, алгоритм CRUSH — это семейство функций, принимающие на вход номер группы размещения и топологию кластера. На выходе она производит список OSD, которые будут закреплены за данной группой размещения. Пул прямо влияет на работу CRUSH, задавая правила, диктующие порядок выбора OSD основываясь на областях сбоев. Подробнее про правила будет рассказано позднее. При изменении топологии распределение также меняется, так как меняется и входной параметр CRUSH. Стоит отметить, что отдельный OSD может быть закреплен за несколькими группами размещения. На рисунке 2 представлена диаграмма пример устройства пула.

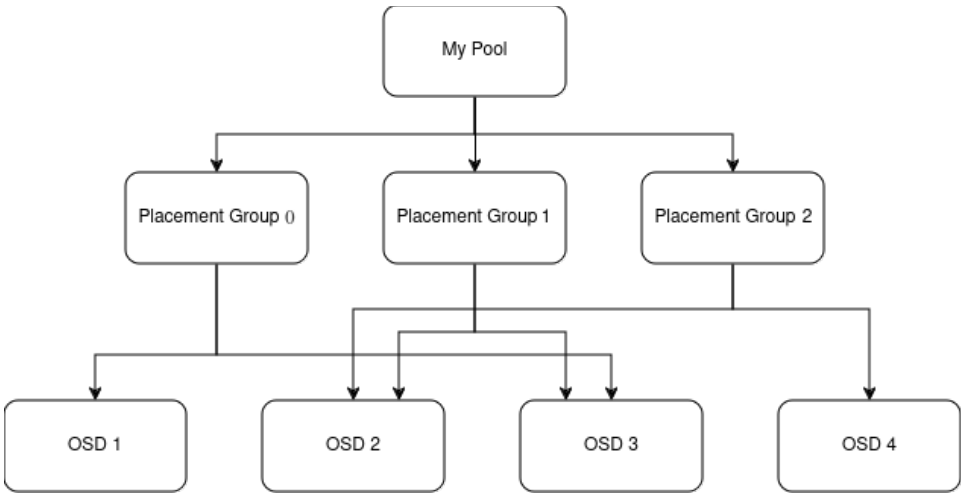

рис. 2: Пул

Все объекты конкретного пула хранилища распределяются именно по группам размещения, а не по отдельным OSD. Рассчитать на какой группе размещения находится данный объект очень просто — берется хэш от идентификатора объекта по модулю количества групп размещений пула [8]. Таким образом, группы размещения являются промежуточным уровнем между клиентом и кластером. Это позволяет не запоминать клиенту на каком именно устройстве находится каждый конкретный объект, и просто обращаться к нужной группе размещения. Это также позволяет клиенту найти объект даже после перераспределения OSD между группами размещений, происходящая после изменения топологии кластера.

Распределение объектов по группам размещения так же упрощает восстановление данных при отказе OSD, так как при таком подходе на больших

объемах данных находить расположение реплик каждого из миллионов объектов слишком вычислительно затратно.

Хранилище Ceph имеют одну структурную ограничение — фиксированный размер хранимого объекта [9]. Эта особенность позволяет хранилищу распределять объекты равномерно по кластеру (учитывая вычислительные характеристики узлов), что также равномерно распределяет нагрузку между его узлами. На практике эта ограничение обходится на одном из уровней абстракции над объектным хранилищем — он нарезает загружаемые объекты на части равного размера и хранит уже эти части [10].

1.2.2. Архитектура Ceph

В традиционной архитектуре распределенной системы обычно существует центральный узел, с которым клиенту необходимо общаться для совершения каждой его операции. Это создает единую точку отказа (англ. single point of failure) системы, то есть если откажет этот узел, то весь кластер так же отказывает в обслуживании. Центральный узел так же является узким горлышком для операций, особенно в контексте высокой нагрузки на сервис, ведь все операции должны проходить через него.

Ceph решает эту проблему позволяя клиентам общаться с OSD напрямую. OSD далее сами копируют данные на соседние OSD для обеспечения дополнительной отказоустойчивости. Всё это достигается с помощью алгоритму CRUSH.

И клиент Ceph, и OSD используют CRUSH для определения местонахождения объектов, вместо того чтобы зависеть на централизованный метод просмотра (такой как, например, таблица). Для определения текущих состояний OSD и их положении, и клиенты, и OSD обращаются к специальному кластеру мониторов, каждый из которых представляет собой высокодоступное консистентное хранилище инфорамации о кластере. Преимущества такой системы:

- Так как у любой сетевой карты есть ограничение на количество

одновременных соединений, централизованная система будет очень ограниченной в контексте высокой нагрузки. Разрешая клиентам и OSD подключаться напрямую, Ceph решает эту проблему, одновременно с этим решая проблему единой точки отказа.

- OSD сами отслеживают работоспособность соседей и докладывают об этом мониторам для обновления карты кластера. Это достигается благодаря heartbeat протоколу [11]. Перевод «сердцебиение» отражает суть такого протокола: рассылка такого сообщения происходит с некоторым интервалом, и если ответа не последовало, то узел будет считаться отказавшим.
- OSD используют CRUSH алгоритм чтобы рассчитать где хранить реплики объектов. Во время записи, клиент кладет объект в пул, где он после распределяется на группу размещения. Затем с помощью CRUSH алгоритма вычисляется присужденный группе размещения OSD. Назовем OSD *главным*, если он стоит первым в списке OSD, закрепленных за группой размещения, и *подчиненными* всех остальных OSD списка. Клиент производит запись только на главный OSD. После чего сам главный OSD с его собственной копией карты кластера находит подчиненные OSD и реплицирует данные на них. Только после подтверждения записи от всех подчиненных OSD главный OSD подтверждает запись клиенту. Это так же увеличивает производительность хранилища, так как, обычно, пропускная способность сети датацентра рассчитана на гораздо больше чем сеть между клиентом и кластером. На рисунке 3 продемонстрирована схема записи данных.

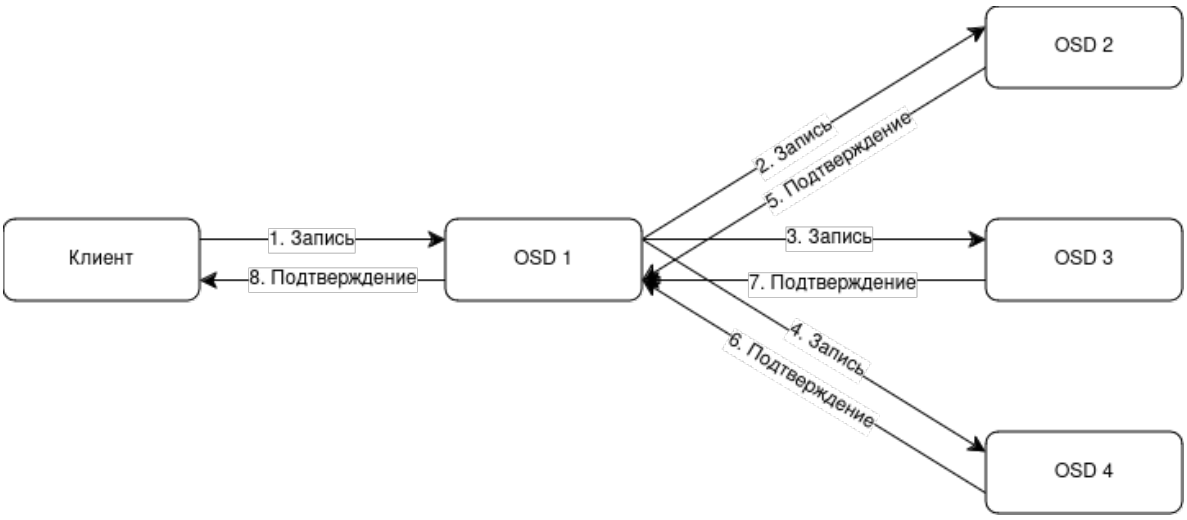

рис. 3: процедура записи

1.2.3. Алгоритм CRUSH

Перед тем как перейти к описанию работы алгоритма, стоит ввести несколько терминов. Ранее было написано, что иерархия областей сбоев моделируется в виде дерева. Узел такого дерева в контексте CRUSH называется *бакетом* (англ. bucket). Учет областей сбоев производится через операции над этими бакетами. Каждый бакет относится к определенному *типу*. По умолчанию в Ceph определены следующие типы бакетов: device — устройство (памяти), host — хост, chassis — серверное шасси, rack — стойка, power distribution unit (pdu) — распределитель питания, pod — под, row — ряд, room — комната, и datacenter — датацентр. Тип бакета задаёт ограничение на вложенность в другие бакеты. Например, нельзя включить бакет типа row в бакет типа host.

У каждого бакета также имеется *вес*, определенный рекурсивно следующим образом:

- Если тип бакета — device, то есть если это устройство памяти, то вес определяется через его объем памяти в терабайтах.
- Для всех остальных типов бакетов вес равен сумме весов его дочерних бакетов.

Для примера возьмем иерархию, изображенную на рис. 1. Жесткие диски задаются типом device. Пусть «Жесткий диск 1» имеет размер 1 терабайт,

«Жесткий диск 2» — 2 терабайта и так далее. Тогда веса жестких дисков будут 1, 2, 3 и 4 соответственно. Сервера задаются типом host. «Сервер-1» будет иметь вес равный 3, рассчитанный как сумма весов дочернего жесткого диска 1 и жесткого диска 2. «Сервер-2» будет иметь вес 7, рассчитывается аналогично. Аналогично считается вес бакета «датацентр» — он будет равен 10.

Алгоритм CRUSH используется при распределении OSD на группы размещения, обращая внимание на области сбоев. Как именно CRUSH берет во внимание области сбоев диктует пул, которому принадлежат группы размещения, через определение *правила*. Правило — это набор шагов, выполняемых CRUSH при выбора OSD. Всего существует 3 типа шагов:

1. *step take <bucket-name>*: находит в иерархии бакет с именем <bucket-name> и кладет его в список обработки. Изначально список обработки пуст.
2. *step [choose|chooseleaf] firstn <N> <bucket-type>*:
    1. *step choose firstn <N> <bucket-type>*: нужно обойти дочерние бакеты каждого элемента списка обработки пока не найдется N бакетов типа <bucket-type>, после чего исходный бакет удаляется из списка обработки. Найденные бакеты кладутся в список обработки.
    2. *step chooseleaf firstn <N> <bucket-type>*: нужно обойти дочерние бакеты каждого элемента списка обработки пока не найдется N бакетов типа *<bucket-type>*, после чего исходный бакет удаляется из списка обработки. Затем нужно спускаться до первого найденного листового бакета (то есть пока не найдет бакет типа device). Найденные бакеты добавляются в список обработки.
3. *step emit*: добавить список обработки в результат. Опустошить список обработки.

Если текущий бакет не удовлетворяет критериям выбора (не подошли тип или имя бакета), то он продолжает поиск у своего потомка о основываясь на *алгоритме выбора бакета*. Всего в Ceph их 4, но в рамках данной курсовой

работы было реализовано 2 наиболее чаще используемые из них:

- *uniform*
  - Псевдослучайно выбирает между потомками с равной вероятностью. Используется в бакете, где потомки имеют равные веса. Существование этого алгоритма обусловлено тем, что обычно в кластерах вводят и выводят оборудование не поштучно, а несколько за раз. Более того, новое оборудование закупается в нескольких экземплярах. Самый эффективный алгоритм. Работает за O(1).
- *straw2*:
  - Псевдослучайно производит выбор потомка, присуждая больший вес потомку с бóльшим весом. Работает за O(n), где n — число потомков. Скорость работы алгоритма компенсирует его честность. Этот алгоритм выбирается для бакетов по умолчанию.

2. Метод визуализации работы Ceph

В рамках данной работы реализована визуализация упрощенной модели работы объектного хранилища. Это проявляется в упрощении работы последующего после отказа одного из OSD и установления новой карты кластера процесса пиринга — установки консистентного состояния между OSD новой и старой карт. Визуализация также не обрабатывает переполнение OSD.

Для начала визуализации пользователь должен предоставить приложению топологию кластера, заданную на предметно-ориентированном языке, основанном языке, использующемся в Ceph. Сам кластер визуализирован как перевернутое дерево, где в основании стоит пользователь — User, от которого идет связь к корневому бакету. Бакеты всех типов изображены в виде прямоугольников. От родительских бакетов проведены связи к дочерним бакетам из середины нижнего основания в середину верхнего основания соответственно.

Отрисовка бакетов типа device — особенная. В визуализации они поставлены в ряд друг за другом под своими родительскими бакетами в виде столбца. Связи от родительского бакета к ним образуются несколько иначе ранее описанной процедуры следующим образом:

1. OSD присваиваются группы размещения. Группа размещения визуализируется прямоугольником внутри OSD.
2. От родительского бакета главного OSD из середины нижней грани проводится связь в середину левой грани группы размещения.
3. Из правой грани группы размещения главного OSD проводятся связи в середины левых граней групп размещений на подчиненных OSD.

Объект на визуализации изображен в виде круга. Уникальный идентификатор объекта изображен внутри этого круга. Во время вставки объекта круг проходит по связям от Пользователя до целевых OSD.

У пользователя есть возможность включить симуляцию работы кластера с отказом OSD по нажатию кнопки. Тогда на каждом шаге визуализации с

некоторой вероятностью OSD может отключиться. Это отображается через покраску OSD в красный цвет. Если этот OSD был главным для какой-то группы размещения на нем, то карта кластера начнет перестраиваться, и для этих групп размещения выдвинутся в кандидаты новые OSD, после чего начинается процесс пиринга. Чтобы новая карта была принята, пиринг должен завершиться успехом. В противном случае будет повторена попытка перестройки карты. Во время пиринга экземпляры группы размещения, участвующие в нем, окрашиваются в оранжевый цвет.

У пользователя есть возможность править топологию кластера во время работы визуализации по нажатию отдельной кнопки. Так можно добавить или удалить OSD из кластера, а так же исправить вес существующего OSD. Изменения в топологии отразятся без перезапуска визуализации.

3. Разработка

Проект спроектирован как клиент-серверное приложение. Клиент не выполняет никаких вычислений и занимается только визуализацией данных, полученных с сервера. Так же с клиента отправляются команды серверу для его управления через кнопки интерфейса.

Было принято решение разбить разработку на несколько шагов:

1. Разработка серверной части приложения:
    1. разработка парсера предметно-ориентированного языка;
    2. реализация алгоритма CRUSH;
    3. разработка симуляции работы кластера;
    4. реализация WebSocket сервера для прослушивания команд от клиента.
2. Разработка клиентской части приложения:
    1. реализации WebSocket клиента;
    2. реализация визуализации топологии кластера.

3.1. Разработка серверной части приложения

Языком реализации серверной части был выбран Python в виду хорошей системы типов. На Python просто и быстро строить прототипы, что ускоряет разработку. Скорость серверной части приложения не виделось проблемой, что также разрешает выбрать Python как язык реализации.

3.1.1. Разработка парсера предметно-ориентированного языка

Грамматика предметно-ориентированного языка, заданная в форме Бэкуса — Наура:

```
<program> ::= <device>+ <bucket>+ <rule>+
<device> ::= "device" "osd." INT "\n"
<bucket> ::= <bucket_type> <bucket_name> "{" "\n"
  "id" "-" INT "\n"
  "alg" ("straw2" | "uniform") "\n"
  "item" <item_name> "weight" FLOAT "\n"
```

```
"}" "\n"
<bucket_type> ::= "host" | "chassis" | "rack" | "row"
                | "pdu" | "pod" | "room" | "datacenter"
                | "region" | "root"
<bucket_name> ::= STR
<item_name> ::= STR

<rule> ::= "rule" <rule_name> "{" "\n"
  "id" INT "\n"
  <rule_step>* "\n"
  "step" "emit" "\n"
"}" "\n"
<rule_name> ::= STR
<rule_step> ::= "step"
  ("take" <bucket_name>
  | ("choose" | "chooseleaf") "firstn" INT <bucket_type>)
```

Разбором этой грамматики будет заниматься отдельный класс — `Parser`. В конструкторе он принимает исходный текст описания топологии кластер, заданная языком, описанным выше. У него есть метод `parse()` отдающий результат исходного теста, в дополнении к чему ещё выполняющий ряд дополнительных проверок, такие как, например:

- все бакеты должны бить использованы ровно один раз;
- имеется единственный бакет типа «root»;
- веса можно указывать только для бакетов типа device;
- имена бакетов — уникальны.

При несоблюдении условий вернется ошибка разбора с местом совершения ошибки.

Выходом парсера является описанная топология кластера в виде дерева и правила с шагами выбора бакетов. Узел дерева имеет следующую сигнатуру:

```
@dataclass
class Bucket:
    name: str
    type: BucketT
    id: BucketID_T
    alg: AlgType # straw2 or uniform
    weight: WeightT = OutOfClusterWeight
    children: list[Self | Device] =
field(default_factory=list)
```

```
    _parent: Self | None = field(init=False,
default=None, repr=False)
    …
    def choose(self, pg_id: int, fails_count: int) ->
Self | Device:
        …
```

3.1.2. Реализация алгоритма CRUSH

Алгоритм CRUSH оперирует над деревом и правилом, полученными после разбора исходного теста описания топологии кластера. Для этого реализована функция `apply()`, принимающая идентификатор группы размещения, дерево и правило с шагами выбора.

Во время работы CRUSH каждое правило выполняется последовательно друг за другом. Разберу каждый тип шагов:

- *step take*: выполняет классический поиск в ширину по дереву для нахождения бакета с нужным именем.
- *step chooseleaf* | *choosen*: выполняет поиск используя типы алгоритма бакета. Если текущий бакет не удовлетворяет условию поиска, то выбор потомка, в котором будет продолжаться поиск, производится основываясь на числе, полученном в результате кеширования идентификатора группы размещения, идентификаторе бакета, и количестве совершенных попыток поиска. Так гарантируется равномерность распределения нагрузки между OSD. Код выбора потомка приведем ниже.
- *step emit*: сохраняет результат предыдущего шага в возвраз.

```
class Bucket:
    ...
    def _choose_uniform(self, pg_id: int, fails_count:
int) -> Self | Device:
        s = int(sha256(str((pg_id, abs(self.id),
failed_attempts)).encode()).hexdigest(), 16)
        return self.children[s % len(self.children)]

    def _choose_straw2(self, pg_id: int, fails_count:
int) -> Self | Device:
```

```
        ws = [c.weight for c in self.children]
        if sum(ws) == 0:
            ws[0] = UnitWeight

        h = int(sha256(str((pg_id, abs(self.id),
failed_attempts)).encode()).hexdigest(), 16)
        random.seed(h)
        res = random.choices(self.children, ws)
        return res[0]
```

3.1.3. Разработка симуляции работы кластера

Необходимо чтобы элементы кластера работали параллельно со всеми остальными. Для этого можно спроектировать симуляцию несколькими способами: Использовать модуль `multiprocessing` и моделировать каждую сущность в отдельном процессе. Но это будет чересчур накладно по ресурсам. Также при таком подходе будет необходимо реализовывать схемы коммуникаций между процессами и их синхронизации, что также значительно усложнит разработку.

Есть ещё один подход: событийно-ориентированная архитектура. При таком подходе не придется размещать сущности по процессам — за всё будет отвечать единый *цикл событий* (англ. event loop), который представляет собой очередь с приоритетом. События будут обрабатываться в порядке возрастания времени. События могут порождаться как со стороны пользователя, так и другими событиями. Этот подход и был взят. За один шаг симуляции будут обрабатываться все события с одной временно́й меткой.

Для того чтобы события не конфликтовали друг с другом необходимо чтобы был способ узнавать работают ли конкретные OSD в конкретный момент времени в будущем или нет. Для этого реализован класс: `AliveIntervals`. В конструкторе он принимает идентификатор OSD и вероятность его сбоя. Он помогает узнать работает ли конкретный OSD следующим образом: берется совместный хэш от идентификатор OSD и временной метки. Затем берутся его последние 16 бит с помощью битовой операции конъюнкции с маской `0xFFFF`. Назовём это *хвостом хэша*. Хвост далее сравниваются с `0xFFFF`,

домноженным на вероятность смерти. Если хвост хэша больше этого порога, то OSD остается работающим. Иначе — OSD отключается. Такой подход обеспечивает необходимую случайность, при этом оставаясь детерминированным.

В начальный момент времени в цикле событий находится единственное событие, отвечающее за жизненный цикл OSD. При обработке оно выполняет следующие действия:

- выключает или включает OSD, основываясь на результатах о их статусе, данном с помощью `AliveIntervals`;
- если карта кластера обновилась, то в цикл обработки кладется события, соответствующие началу и концу пиринга; успешность пиринга можно просчитать заранее с помощью `AliveIntervals`: если после нескольких шагов симуляции OSD, участвующие в пиринге, остаются работающими, то пиринг считается успешным;
- кладет свою копию обратно в цикл, но с большей временной меткой, чтобы обработчик обрабатывал это событие на каждом шаге симуляции.

Вставка объекта в хранилище также происходит на основе событий. Алгоритм обработки вставки следующий:

1. От пользователя поступает сигнал о начале вставки. Объект распределяется в группу размещения. С главным OSD этой группы устанавливается связь.
2. Если после нескольких шагов OSD остается работающим, то процесс продолжается и в цикл обработки кладется событие, соответствующее репликации объекта на подчиненные OSD. Иначе, в цикл обработки кладется событие, соответствующее провалу попытке записи на главный OSD.
3. При обработке события репликации производится проверка работоспособности подчиненных OSD. Если после нескольких шагов симуляции подчиненный OSD всё ещё работает, тогда от него кладется в

цикл событий событие, подтверждающее запись. Если подчиненный OSD не работает, тогда кладется событие о провале записи.

4. Если все подчиненные OSD ответили успехом, то главный OSD порождает событие о подтверждении записи пользователю. Иначе — о провале.

Замечу, что все события генерируются вместе, а не рассчитывая новые события основываясь на старых, ведь возможно просчитать будут ли работать OSD или нет заранее благодаря `AliveIntervals`.

3.1.4. Реализация WebSocket сервера

В реализации асинхронного сервера была использована библиотека websockets [12].

Все команды передаются в формате JSON. Всего сервер ожидает 5 типов команд, поступающих от пользователя. Типы команд записаны в поле *type*.

1. «*rule*»: разбирает присланный исходный текст программы, сохраняя у себя топологию кластера и правила для дальнейшей работы над ними. Пользователю возвращается разобранная топология в формате JSON для дальнейшей её отрисовки.
2. «*adjust_rule*»: разбирает присланный исходный текст программы, обновляя у себя топологию кластера и правила. Пользователю возвращается разобранная топология в формате JSON для дальнейшей её перерисовки.
3. «*step*»: производит шаг симуляции. Отправляет пользователю информацию об обработанных событиях в формате JSON. Описывается тип обработанного события с дополнительной информацией.
4. «*insert*»: событие, описывающее вставку объекта, кладется в очередь с текущей временной меткой.
5. «*mode*»: отключает или включает нестабильную работу OSD, выставляя вероятность их отключения в 0.

3.2. Разработка клиентской части приложения

Визуализации топологии кластера будет проводится на web-странице в браузере с использованием JavaScript. Отрисовка объектов будет проводится на элементе Canvas.

Для упрощения создания web-страницы были использованы следующие инструменты разработки:

- Библиотека FabricJS [13]: она значительно упрощает работу с Canvas давая дополнительный слой абстракции, благодаря которому можно отрисовывать и анимировать фигуры. Функционал библиотеки намного богаче, но в работе использовалось только это.
- Tailwind CSS [14]: фреймворк, упрощающий стилизацию web-страницы. Основная идея фреймворка заключается в том, что вместо определения стиля для каждого элемента страницы в CSS файле, это делается напрямую на HTML странице, задавая элементам набор классов, которые предоставляются фреймворком. Каждый класс Tailwind CSS задает какое-либо свойство элемента — цвет, шрифт, размер, положение, эффекты и т. п.
- Vite [15]: современное средство сборки JavaScript проекта. Даёт возможность мгновенного отражения изменений кода на web-странице.
- JSDoc [16]: даёт возможность аннотировать типы переменных и функций.

3.2.1. Описание интерфейса

Страница разбита на два раздела:

1. Редактор описания топологии кластера. Расположен в левой части страницы. В нем редактируется описание топологии кластера. Редактор имеет две функциональные кнопки
    1. Кнопка «Adjust» отвечает за отправку описания топологии на сервер с целью правки текущей топологии. Не прерывает текущую

визуализацию.

2. Кнопка «Submit» отвечает за отправку описания топологии на сервер с целью переписывания текущий топологии новой. Перезапускает визуализацию сначала.

2. Панель визуализации топологии кластера. Занимает бóльшую часть площади страницы. На ней расположены:
   1. Панель управления, на которой расположено 3 кнопки:
      1. Кнопка «Mode» отвечает за включение/выключение режима нестабильной работы кластера. Изначально выставлена в режим нестабильной работы.
      2. Кнопка «Insert» уведомление сервера о начале вставки объекта в хранилище.
      3. Кнопка «Step» переходит к следующему шагу симуляции, обращаясь к серверу за новыми событиями.
   2. Элемент Canvas. На нем визуализируется работа объектного хранилища. В дополнении, у него реализованы возможности увеличения/уменьшения приближения по прокручиванию колеса мыши, а также возможность перемещения по нему по зажатию левой кнопки мыши и перемещению её в нужную сторону. В левом верхнем углу Canvas имеется таймер, показывающий текущее время симуляции.

На рисунке 4 представлен вид страницы из браузера

рис. 4: страница

3.2.2. Реализации WebSocket клиента

Реализация WebSocket клиента производилась с помощью встроенной поддержки браузера этого протокола. Общение с сервером происходит по инициативе клиента после нажатия им одной из кнопок. Клиент и сервер обмениваются сообщениями формата JSON. О типах сообщений было рассказано в главе 3.1.4.

3.2.3. Реализация визуализации работы кластера

Для визуализации было создано несколько абстракций:

- `Bucket`: занимается отрисовкой бакета на Canvas, а также связями между бакетами. При отрисовке использует класс `fabric.Rect`.
- `PG`: занимается отрисовками экземпляра группы размещения на Canvas, путей к другим экземплярам группы размещения, а также пути к родительскому бакету. При отрисовке использует класс `fabric.Rect`.
- `OSD`: занимается отрисовкой OSD на Canvas. Выступает в виде слоя над `PG`, позволяющий перефразировать отрисовку связи между `PG` в терминах связи между `OSD`, содержащих экземпляры групп размещения. При отрисовке использует класс `fabric.Rect`.

- `RCPath`: абстракция над связью. Абстрагирует детали реализации перерисовки и удаления связи с Canvas. В основании использует класс `fabric.Polyline`.
- `ConnectorAllocator`: занимается выделением места для связей на Canvas. Благодаря этому классу связи не путаются между собой.

3.2.3.1. Отрисовка бакетов

При отрисовке топологии кластера, полученную после загрузки серверу её описания, алгоритм просчитывает оптимальное местоположение объектов на Canvas. Это реализовано следующим образом:

- Каждому бакету рекурсивно просчитывается ширина, необходимая чтобы уместить его потомков, следующим образом: $requiredWidth = gap \cdot (childrentCount - 1) + \sum_{i=0}^{childrentCount} calculateWidth(child_i)$, если потомки не являются бакетами типа device, и $requiredWidth = osdWidth$, если потомки являются бакетами типа device.
- Чтобы отрисовать бакеты на экране нужно воспользоваться рассчитанным параметром *requiredWidth*, ставя бакет в по середине *requiredWidh*.
- Бакеты типа device отрисовываюстя под своими родительскими бакетами.

На рисунке 5 изображена визуализация топологии кластера, отрисованная сейчас описанной процедурой.

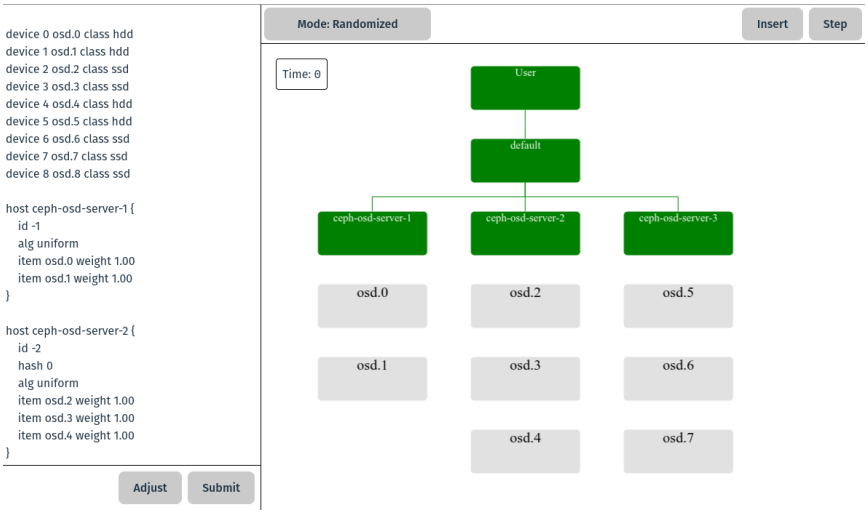

рис. 5: топология кластера

3.2.3.2. Отрисовка связей

После начала симуляции от сервера начнут поступать сообщения об обновлении карты кластера. В сообщении от сервера указывается где располагаются экземпляры групп размещений. После этого группы размещения, расположенные на главных OSD начинают связываться с группами на подчиненными OSD. На рисунке 6 можно увидеть состояние кластера после установки всех соединений.

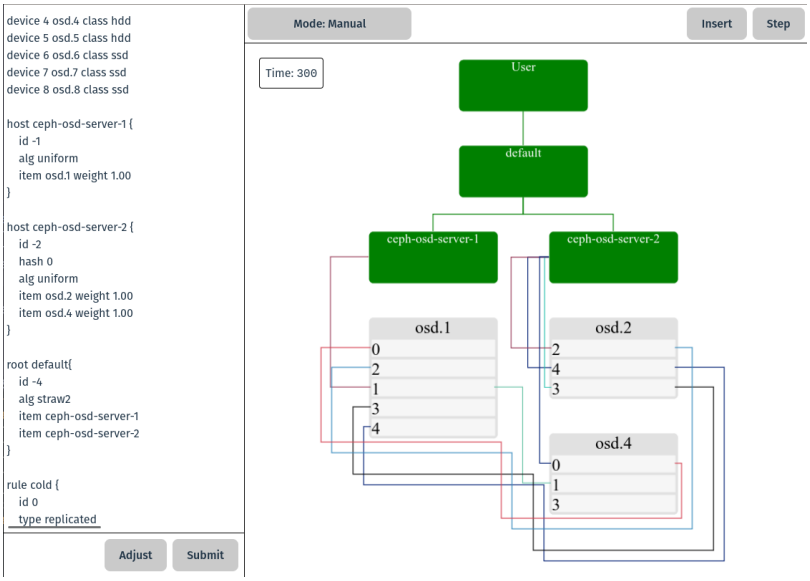

рис. 6: кластер со связями

На рисунке показан кластер со следующим расположением групп размещения:

- 0 → [osd.4, osd.1]
- 1 → [osd.1, osd.4]
- 2 → [osd.2, osd.1]
- 3 → [osd.2, osd.1]
- 4 → [osd.2, osd.1]

Правило CRUSH алгоритма:

```
rule cold {
    id 0
    step take default
    step chooseleaf firstn 0 type host
    step emit
}
```

Каждый экземпляр класса PG разделяет между другими экземплярами единственный `ConnectorAllocator`. Он нужен для выделения места под линии связей между ними. Общий аллокатор нужен для того чтобы линии связей пересекались минимальное количество раз.

3.2.3.3. Отрисовка вставки

Процесс вставки визуализирован следующим образом:

1. Вставляемый объект спускается из корня дерева по связям вниз к листьям в свою группу размещения в главном OSD.
2. Попав на главный OSD в свою группу размещения, объект загорается зеленым цветом, обозначая успешность записи; затем, начинается реплицирование объекта на подчиненные OSD. Процесс репликации изображен через движение объекта от главного OSD к подчиненным.
3. Если репликация на подчиненный OSD прошла успешно, то объект отрисовывается на нём внутри группы размещения и загорается зеленым цветом, обозначая успешность записи. Иначе, объект загорается красным цветом.
4. Через некоторое время после успешной записи объект ещё раз загорается зеленым цветом, сигнализируя подтверждение успешной записи главному OSD.
5. После подтверждения записей на подчиненных OSD, запись, наконец, подтверждается на главном OSD, что показывается в виде отрисовки объекта на главном OSD внутри своей группы размещения с окраской его в зеленый цвет. Если же хоть один OSD не подтвердил успешность записи, объект также отрисовывается на главном OSD внутри своей группы размещения, но окрашивается уже в красный.

Процессы вставки можно увидеть на рисунках 7 и 8.

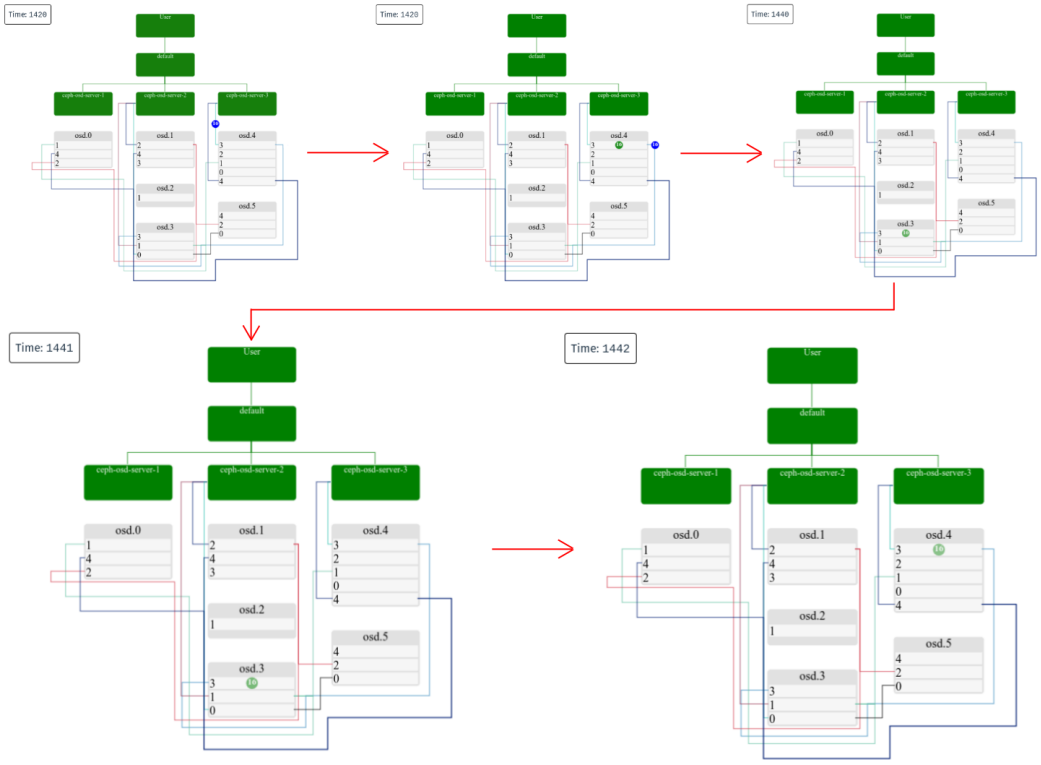

рис. 7: успешная вставка объекта

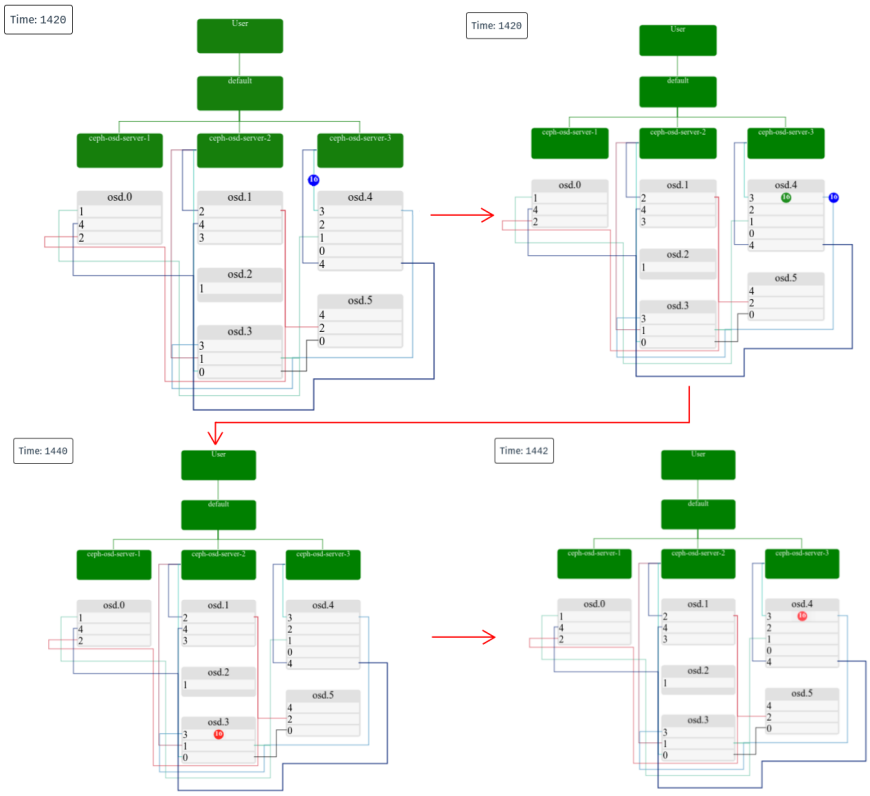

рис. 8: вставка объекта провалилась

3.2.3.4. Отрисовка неработающих OSD

Неработающие OSD окрашиваются в красный цвет. На рисунке 9 представлен кластер, у которого отказал OSD с именем «osd.0».

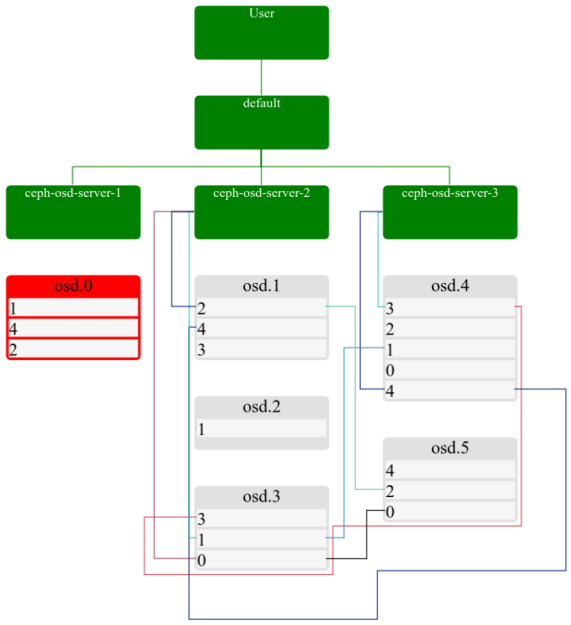

рис. 9: визуализация неработающего OSD

3.2.3.5. Отрисовка пиринга

Пиринг групп размещения визуализируется окраской их в оранжевый цвет. Кластер с группами размещения в состоянии пиринга расположен на рисунке 10.

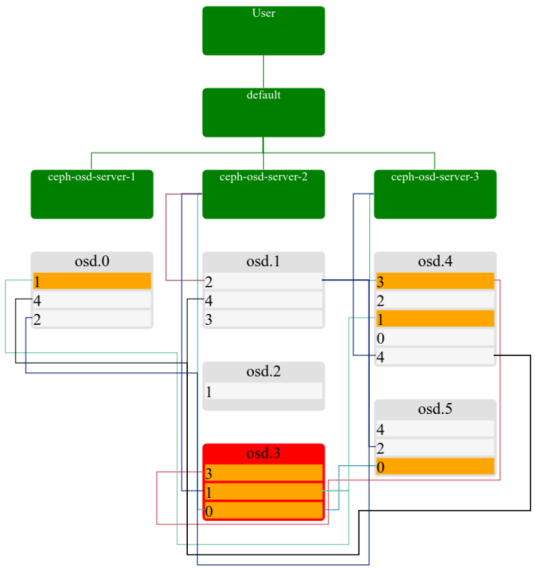

рис. 10: пиринг

4. Тестирование

4.1. Тестирование работы функции корректировки топологии кластера

Тестируемый функционал: кластер должен продолжить работу после внешнего вмешательства.

Протестируем кластер на следующей топологии: 3 хоста, у первого из которых 1 OSD на 1 ТБ, у второго — 2 OSD на 1 ТБ, у третьего — 3 OSD на 1 ТБ. Кластер изображен на рисунке 11.

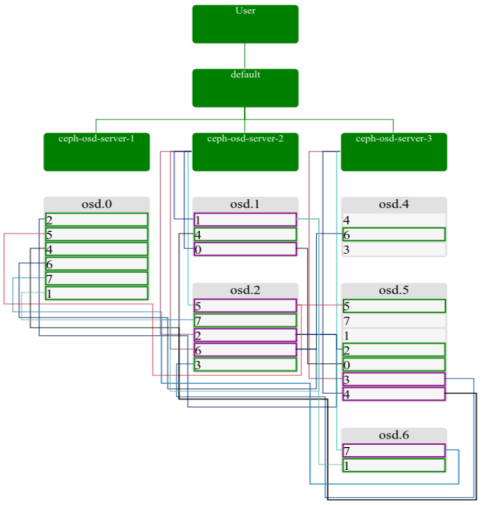

рис. 11: исходный кластер

Выведем из строя osd.2, ожидая, что группы размещения 5, 7, 2, 6, 3 войдут в состояние пиринга. Действительно так и вышло. Переход к новому кластеру изображен на рисунке 12.

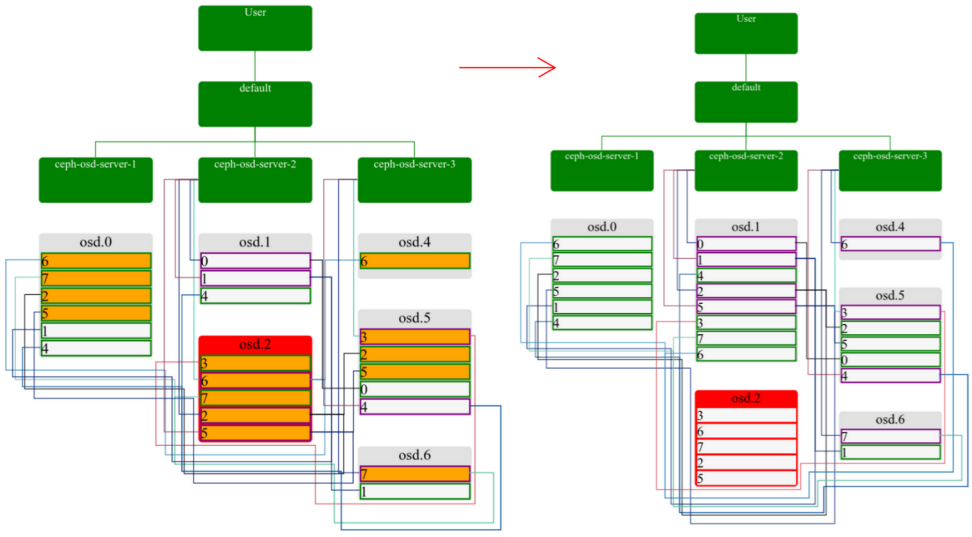

рис. 12: перестраивание карты кластера после вывода osd.2

Попробуем добавить новые OSD в кластер. Ожидается, что часть групп размещения перейден на новые OSD. Введем в кластер новый хост — ceph-osd-server-4, на который расположим 2 OSD, вес которых равен 1 ТБ. После введения новых OSD произошла миграция групп размещения. Результирующий кластер можно увидеть на рисунке 13.

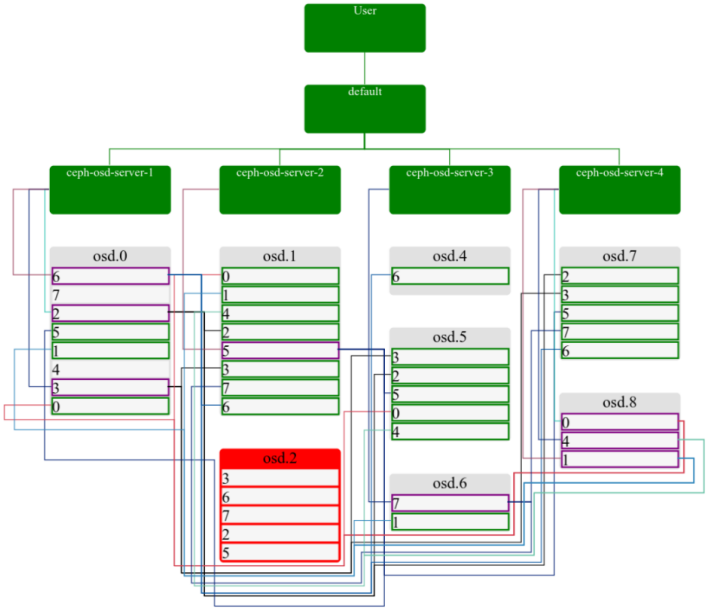

рис. 13: кластер после добавления OSD

Попробуем ввести osd.2 обратно в кластер, ожидая что на него также мигрируют несколько групп размещения. Так и вышло. Результат миграции виден на рисунке .

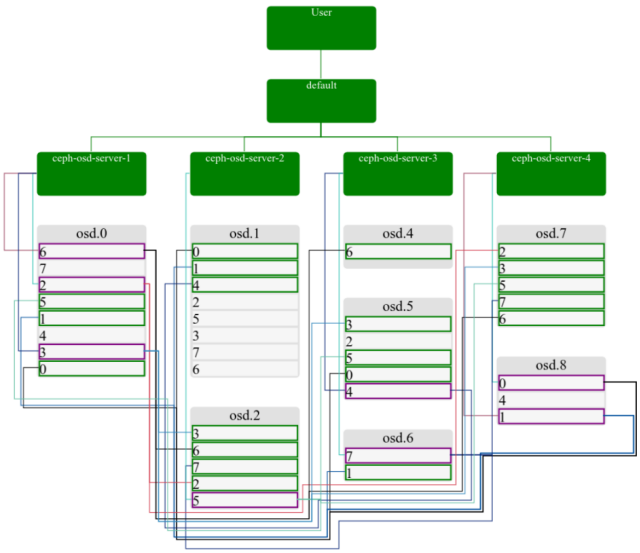

рис. 14: кластер после возврата osd.2

Результат тестирования: функционал ведет себя как нужно.

4.2. Тестирование влияния веса на распределение групп размещения

Тестируемый функционал: OSD с большим весом должны выбираться как главные чаще при условии использования алгоритма *straw2*.

Проверим это следующим образом: Создадим кластер из трех OSD с равными весами. У всех бакетов выставим значения алгоритма в straw2. Кластер изображен на рисунке 15:

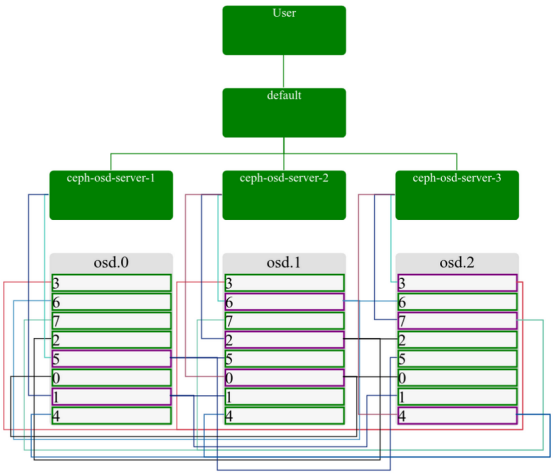

рис. 15: исходный кластер

Теперь увеличим значение веса для osd.2 в 10 раз, ожидая, что он будет выступать в роли главного OSD чаще. Так и вышло. Кластер после изменения веса изображен на рисунке 16.

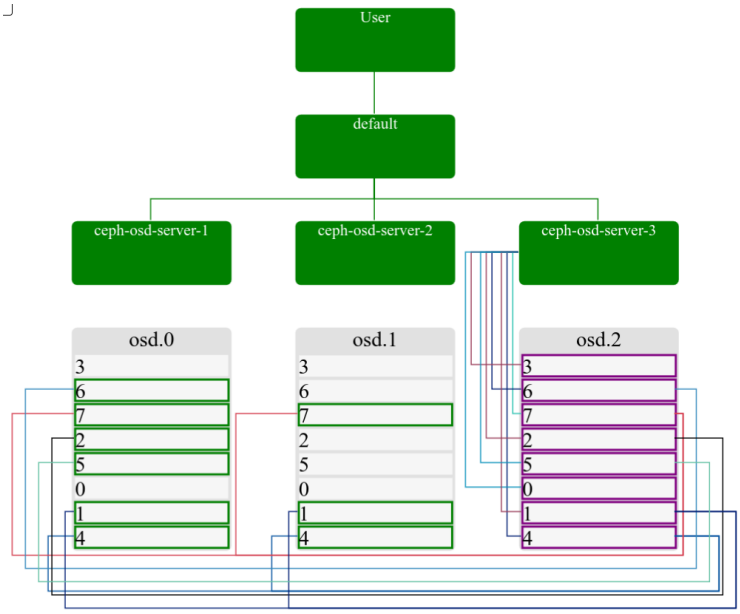

рис. 16: измененный кластер

Результат тестирования: функционал ведет себя как нужно.

СПИСОК ЛИТЕРАТУРЫ

1. Dan van der Ster. Ceph at CERN: A Ten-Year Retrospective // Ceph Days NYC, 2023
2. Google Cloud Storage Documentation [Электронный ресурс]. — URL: https://cloud.google.com/storage?hl=en
3. Amazon S3 Documentation [Электронный ресурс]. — URL: https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html
4. Yandex Object Storage [Электронный ресурс]. — URL: https://yandex.cloud/en-ru/docs/storage/quickstart/
5. Ceph [Электронный ресурс]. - URL: https://ceph.io/en/discover/
6. Google spotlights data center inner workings [Электронный ресурс]. — URL: https://www.cnet.com/culture/google-spotlights-data-center-inner-workings/
7. Sage A. Weil Scott A. Brandt Ethan L. Miller Carlos Maltzahn. CRUSH: Controlled, Scalable, Decentralized Placement of Replicated Data — URL: https://ceph.com/assets/pdfs/weil-crush-sc06.pdf
8. Ceph Documentation. Calculating PG IDs [Электронный ресурс]. — URL: https://docs.ceph.com/en/reef/architecture/#calculating-pg-ids
9. Sage A. Weil Andrew W. Leung Scott A. Brandt Carlos Maltzahn. RADOS: A Scalable, Reliable Storage Service for Petabyte-scale Storage Clusters. — URL: https://ceph.io/assets/pdfs/weil-rados-pdsw07.pdf
10. Ceph Documentation. Ceph Object Gateway [Электронный ресурс]. — URL: https://docs.ceph.com/en/reef/radosgw/
11. Ceph Documentation. Configuring Monitor/OSD Interaction [Электронный ресурс]. — URL: https://docs.ceph.com/en/reef/rados/configuration/mon-osd-interaction/
12. Websockets library [Электронный ресурс]. — URL: https://websockets.readthedocs.io/en/stable/
13. FabricJS Documentation [Электронный ресурс]. — URL: https://fabricjs.com
14. Tailwind CSS Documentation [Электронный ресурс]. — URL: https://tailwindcss.com/

15. Vite build tool [Электронный ресурс]. — URL: https://vite.dev/
16. JSDoc Documentation [Электронный ресурс]. — https://jsdoc.app/