CubeFS Deep Dive

Возможности единого хранилища для ML, аналитики и контейнерных платформ

Иван Архипов

SmartData 2026

Кто говорит?

Иван Архипов

email: me@endevir.ru
tg: @endevir

Чем занимаюсь

Kubernetes

Пишу код в ядро k8s и разрабатываю k8s-операторы для stateful сервисов

DBMS

В 2023 запустил DBaaS для PostgreSQL в production на десятки тысяч БД в Kubernetes

Networking

Разрабатываю оверлейную сеть в Яндекс.Облаке

Storage

В свободное время собираю и тестирую распределенные хранилки в Kubernetes

Работа с CubeFS – личный проект, поэтому в докладе не будет чисел крупного продакшена, зато будет честный рассказ о том, что болело и как дорабатывали, без NDA :)

Развитие CubeFS

Развитие CubeFS

2017

ChubaoFS 1.0.0

Появился в JD.com как дешевая распределенная ФС для задач ML, распределенного кэша, логов и бэкапов

2019

v1.5.0
CNCF Sandbox

Проект переименован в CubeFS и передан CNCF

Новый fuse-клиент, кэширование и чтение с реплик, лимиты по iops

2020

v2.0.0

S3 API, POSIX атрибуты, мультизональность и размещение волюмов с привязкой по зонам

2022

v3.0.0
CNCF Incubating

Появление BlobStore-движка с поддержкой erasure coding

2024

CNCF Graduated

Безопасность и мультитенантность: аутентификация, авторизация, beta снепшотов, QoS по IOPS/throughput

2026

v3.6.0

Адаптация под AI-нагрузки: cluster-wide кэш, оптимизация read path, клиентский read-ahead

multi-region метаданные и улучшения автоматического self-healing

OPPO
JD.com
Meizu
Xiaomi
NetEase

200+
организаций-пользователей

Clickhouse
Kafka/AutoMQ

100+ PB
кластеры крупных клиентов

Elasticsearch
Iceberg

Устройство CubeFS: общая картина

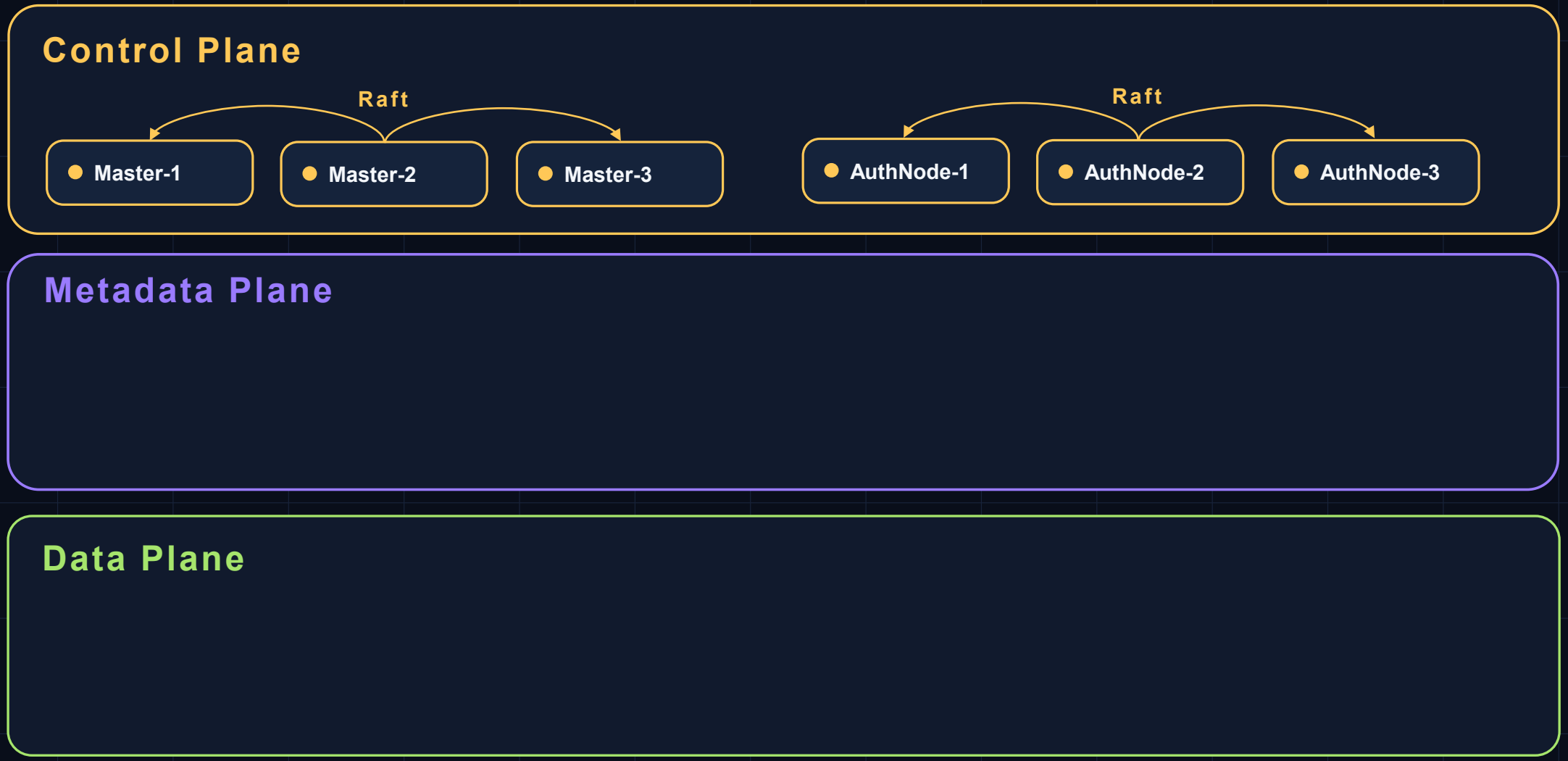
Control Plane

Metadata Plane

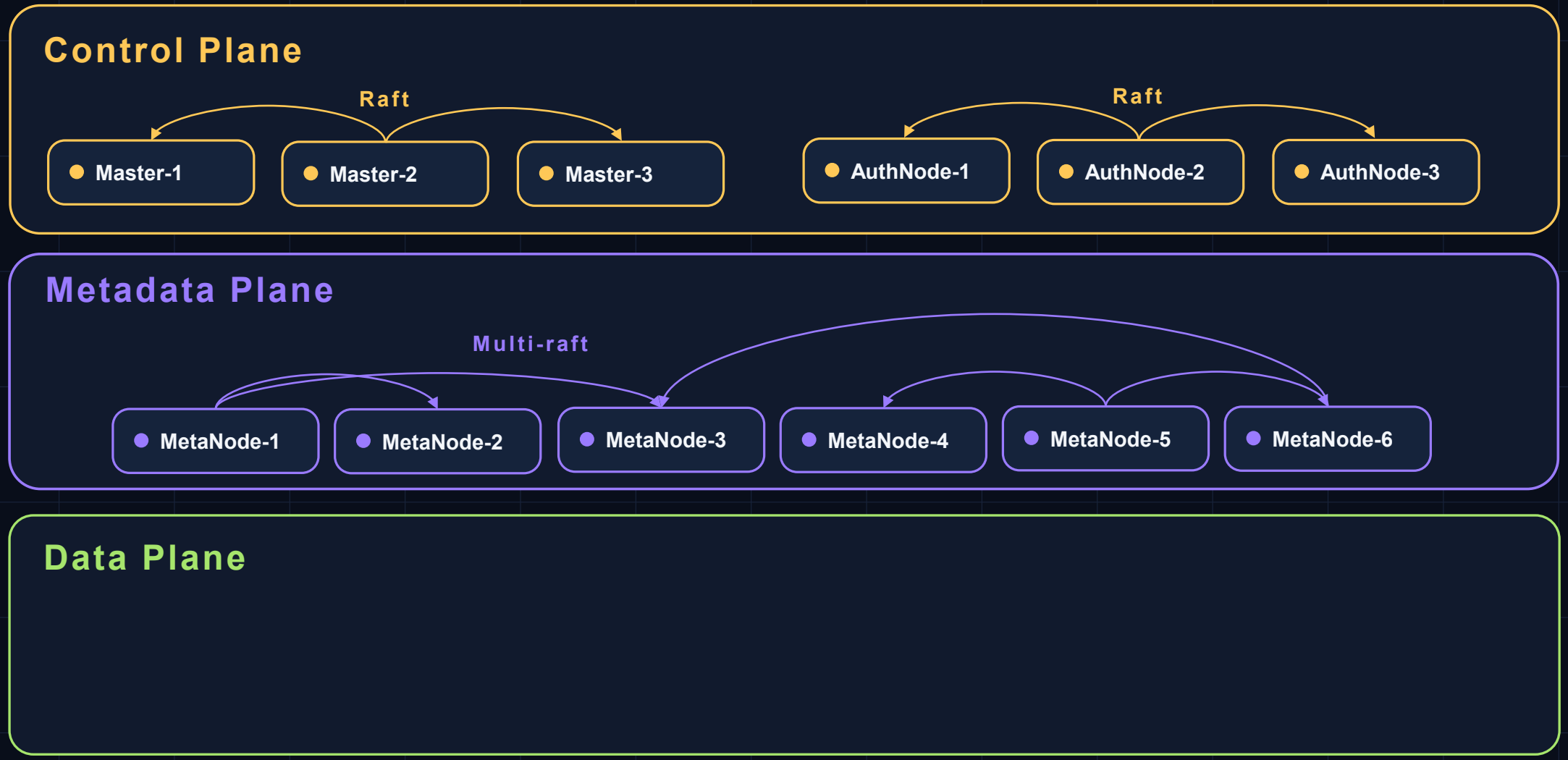
Data Plane

Устройство CubeFS: общая картина

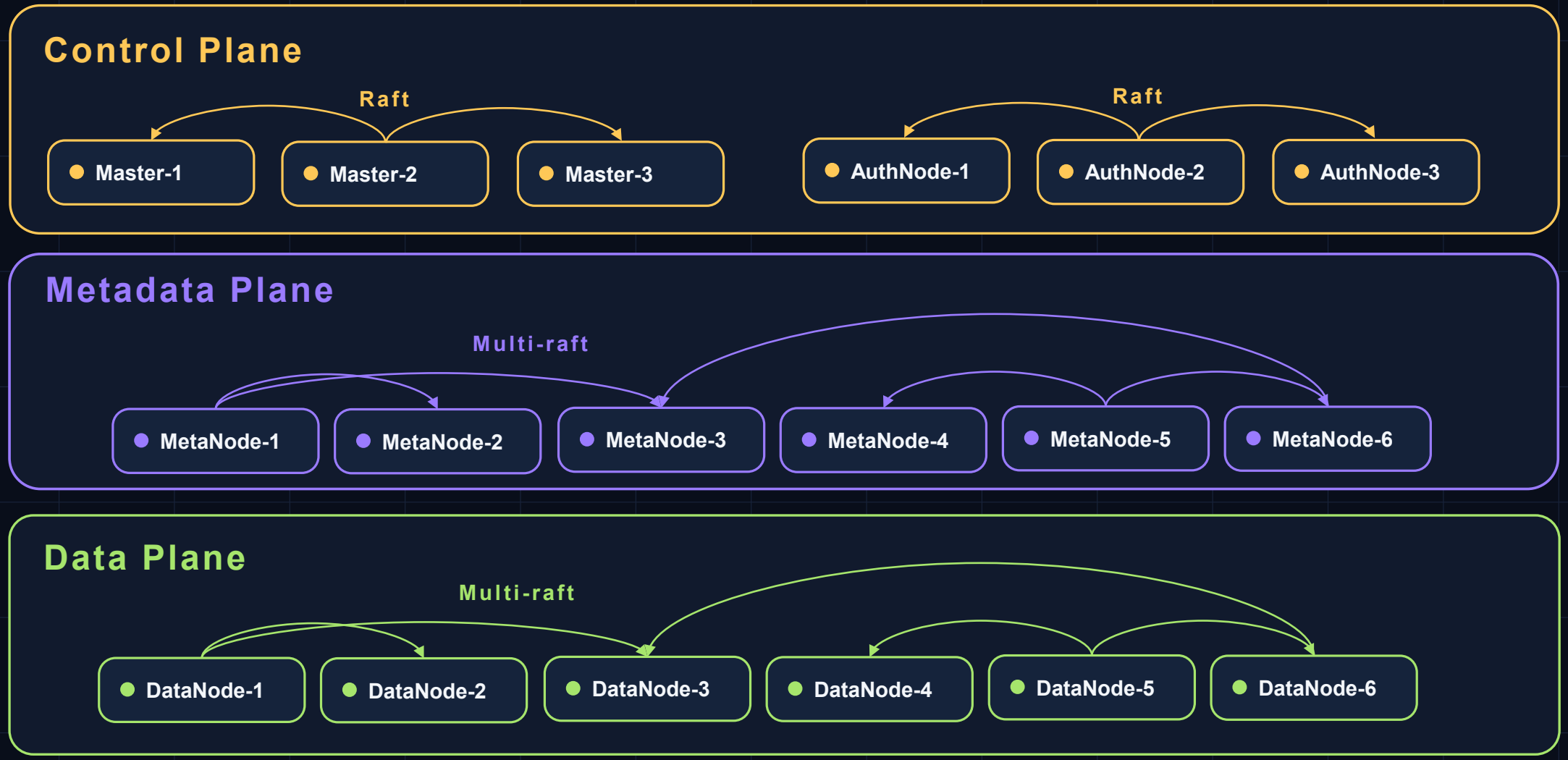
Устройство CubeFS: общая картина



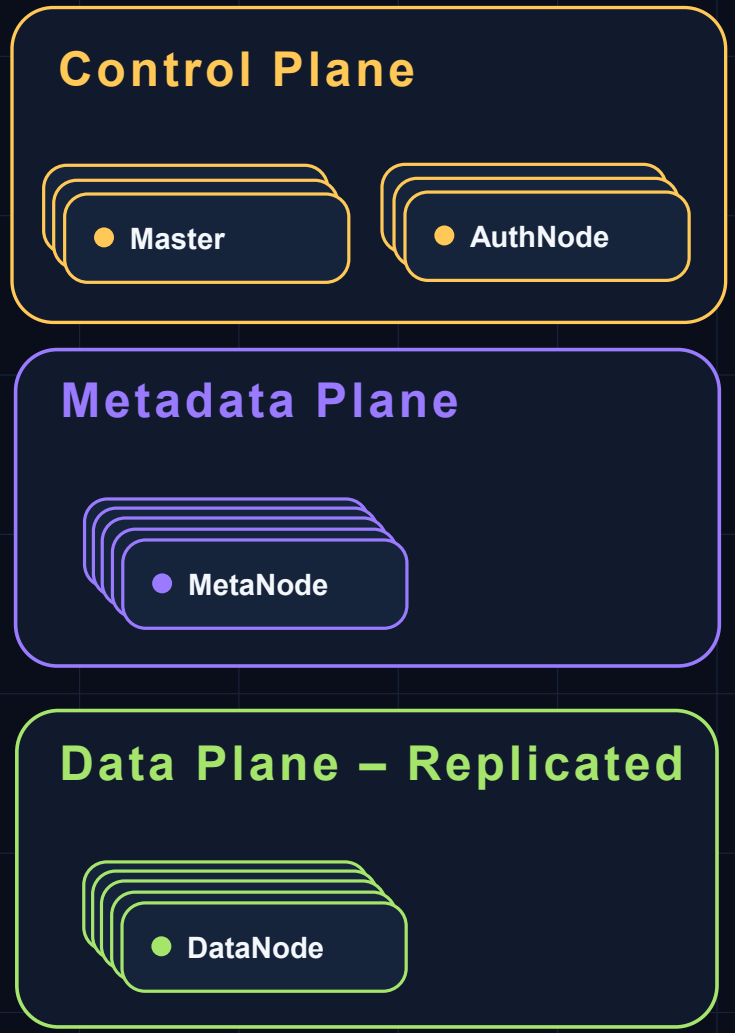
Устройство CubeFS: общая картина



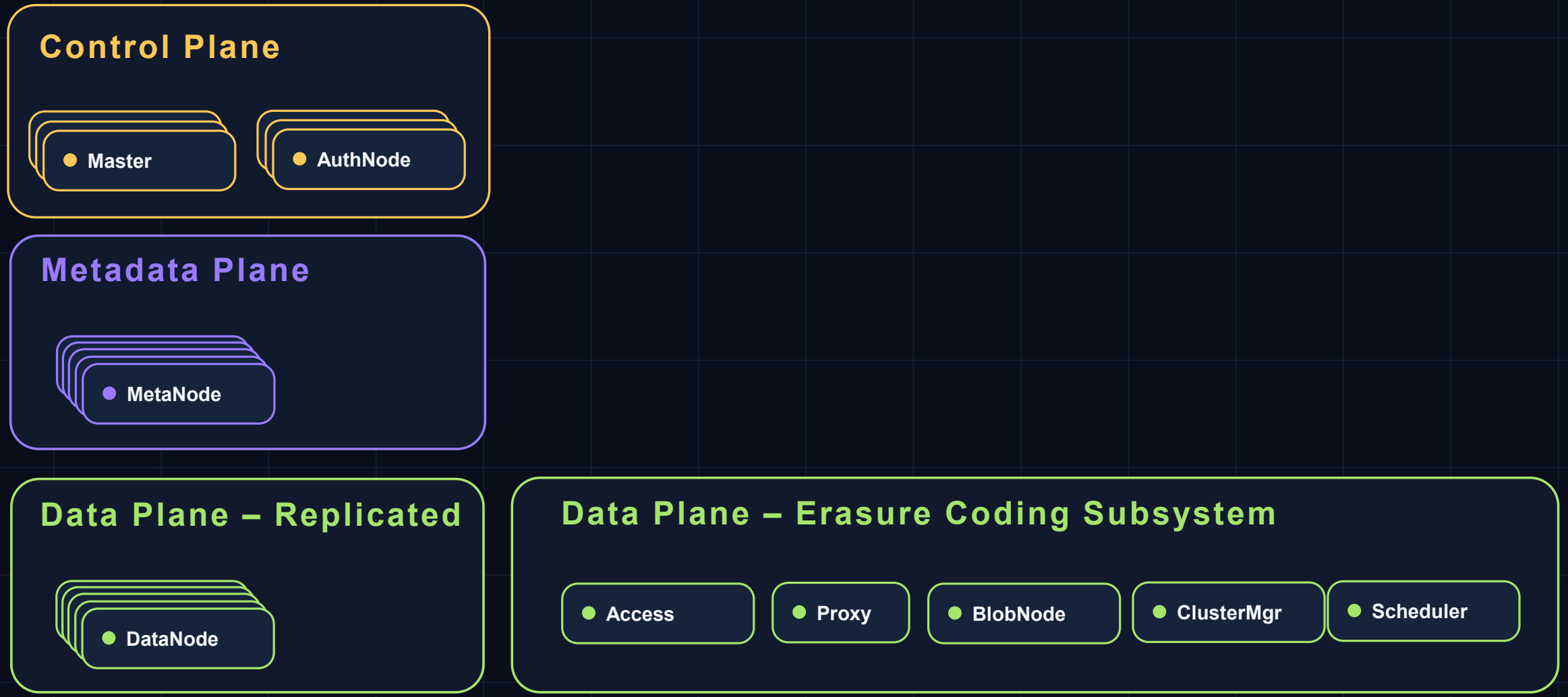
Устройство CubeFS: общая картина



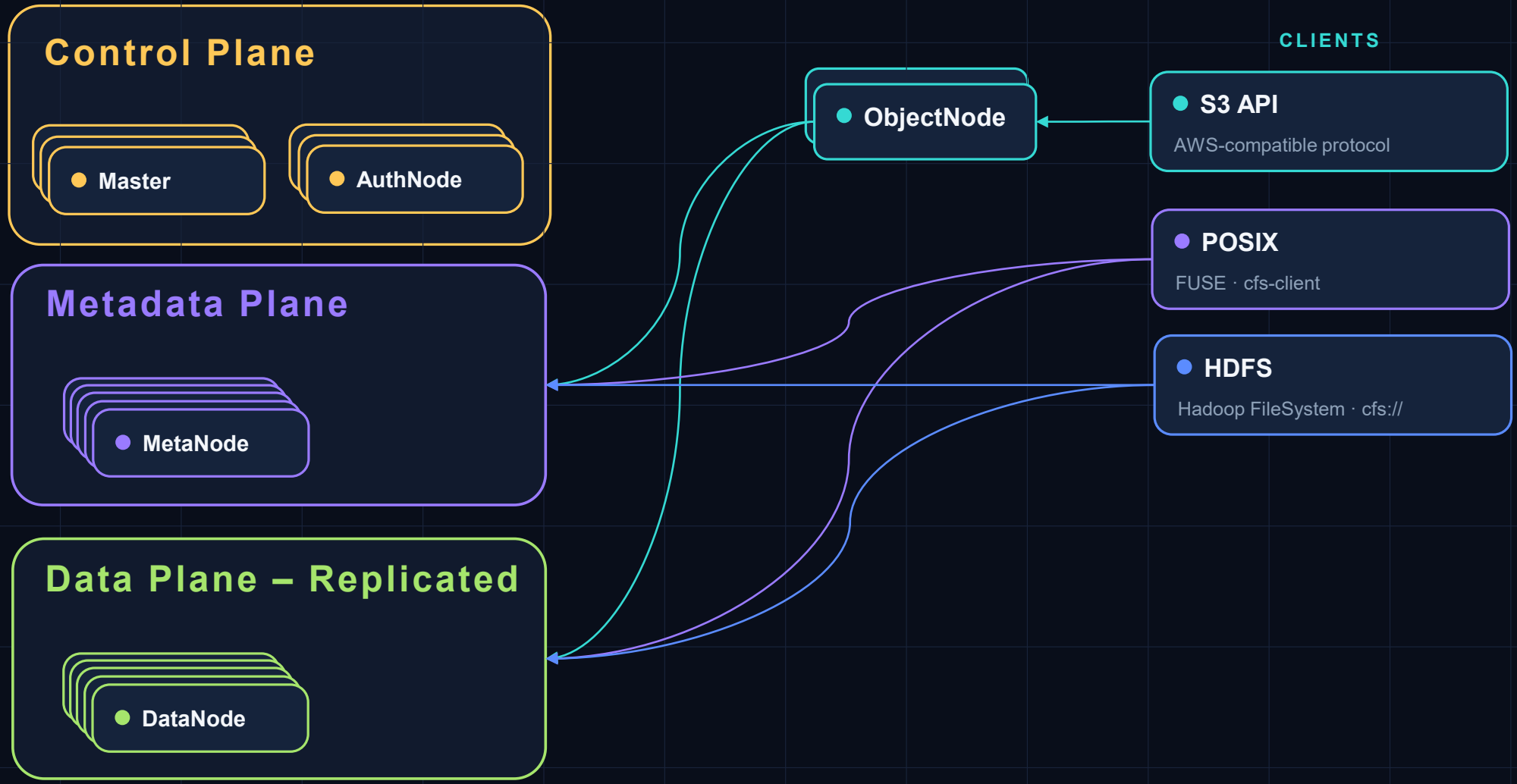
Устройство CubeFS: общая картина



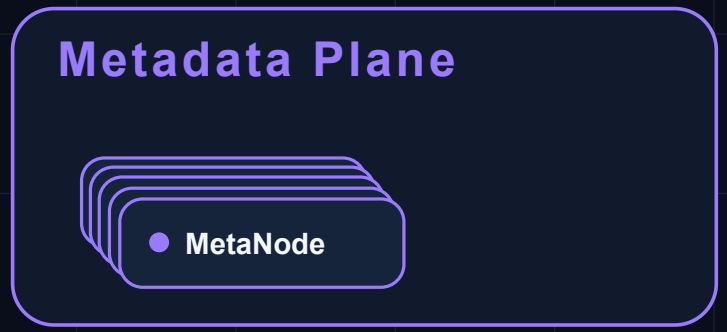
Устройство CubeFS: общая картина



Устройство CubeFS: общая картина



MetaNode: MetaPartition, Raft и две B-Tree



MetaNode: MetaPartition, Raft и две B-Tree

Одна MetaNode содержит много MetaPartition;

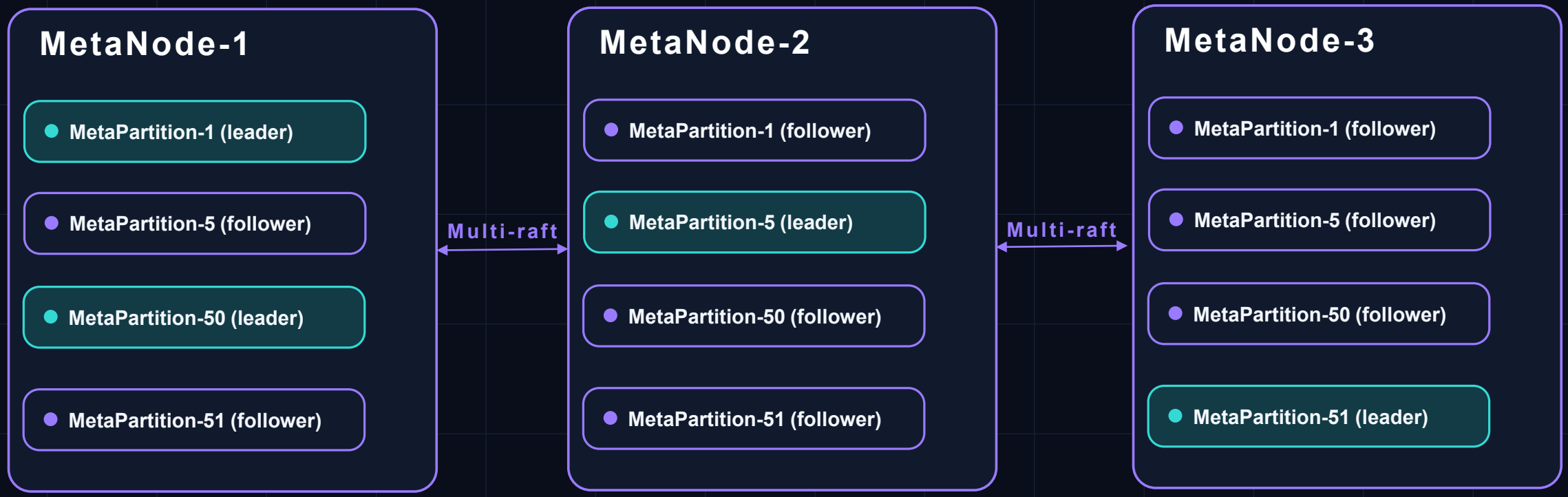
Каждая MetaPartition хранит данные только одного Volume.

При этом у каждого Volume может быть много MetaPartition, распределенных между MetaNode

MetaNode: MetaPartition, Raft и две B-Tree

MetaPartition реплицируется между разными MetaNode с помощью Raft

У каждой MetaNode могут быть одновременно и партиции-лидеры, и партиции-фолловеры



MetaNode: MetaPartition, Raft и две B-Tree

Каждый MetaPartition содержит в себе два B-дерева:

- Inode-tree – метаинформация об объекте (файле/папке) и размещение его данных
- Dentry-tree – маппинг (parent inode + child name) => child inode, задающий иерархию файловой системы

Inode отвечает на вопросы «что это за объект?» и «где лежат его байты?»

Dentry отвечает за положение объекта в дереве файловой системы.

Dentry · место в дереве

key: (parentId, name)

```
// Dentry defines the dentry struct.
type Dentry struct {
    ParentId  uint64 // inode of the parent directory
     Name      string // The name of the current directory entry
     inode     uint64 // inode of the current file
     Type      uint32 // File Types and Permissions
}
```

Inode – объект ФС

key: inode ID

```
type inodeInfo struct {
    sync.RWMutex
    inode      uint64 // inode id
    Type       uint32 // file type and permissions
    Uid        uint32 // user id
    Gid        uint32 // group id
    Size       uint64 // File size
    Generation uint64 // The version number, incremented for each revision
    CreateTime int64  // creation time
    AccessTime int64  // interview time
    ModifyTime int64  // Change the time
    LinkTarget []byte // Save the path name of the source file corresponding to the soft link
    NLink      uint32 // number of hard links
    Flag       int32  // Whether the mark can be deleted
    Extents    *SortedExtents // The location information of the file in the three copies
    ObjExtents *SortedObjExtents // The location information of the file erasure code
}
```

MetaNode: MetaPartition, Raft и две B-Tree

Каждый MetaPartition содержит в себе два B-дерева:

- Inode-tree – метаинформация об объекте (файле/папке) и размещение его данных
- Dentry-tree – маппинг (parent inode + child name) => child inode, задающий иерархию файловой системы

Inode отвечает на вопросы «что это за объект?» и «где лежат его байты?»

Dentry отвечает за положение объекта в дереве файловой системы.

Чтобы разрешить путь warehouse/events/day=28/part.parquet, клиент последовательно ищет dentry каждого элемента дерева, а затем читает inode конечного файла.

Dentry · место в дереве

key: (parentId, name)

```
// Dentry defines the dentry struct.
type Dentry struct {
    ParentId  uint64 // inode of the parent directory
     Name      string // The name of the current directory entry
     inode     uint64 // inode of the current file
     Type      uint32 // File Types and Permissions
}
```

Inode – объект ФС

key: inode ID

```
type inodeInfo struct {
    sync.RWMutex
    inode      uint64 // inode id
    Type       uint32 // file type and permissions
    Uid        uint32 // user id
    Gid        uint32 // group id
    Size       uint64 // File size
    Generation uint64 // The version number, incremented for each revision
    CreateTime int64  // creation time
    AccessTime int64  // interview time
    ModifyTime int64  // Change the time
    LinkTarget []byte // Save the path name of the source file corresponding to the soft link
    NLink      uint32 // number of hard links
    Flag       int32  // Whether the mark can be deleted
    Extents    *SortedExtents // The location information of the file in the three copies
    ObjExtents *SortedObjExtents // The location information of the file erasure code
}
```

MetaNode: in-memory и сайзинг

Inode- и Dentry-деревья целиком находятся в оперативной памяти, а Raft log и снапшоты позволяют восстановить их после перезапуска.

Ёмкость всего кластера ограничена объёмом оперативной памяти, которую мы резервируем под слой метаданных.

In-memory B-trees

Dentry · место в дереве

key: (parentId, name)

```
// Dentry defines the dentry struct.
type Dentry struct {
    ParentId  uint64 // inode of the parent directory
     Name      string // The name of the current directory entry
     inode     uint64 // inode of the current file
     Type      uint32 // File Types and Permissions
}
```

Inode – объект ФС

key: inode ID

```
type inodeInfo struct {
    sync.RWMutex
    inode      uint64 // inode id
    Type       uint32 // file type and permissions
    Uid        uint32 // user id
    Gid        uint32 // group id
    Size       uint64 // File size
    Generation uint64 // The version number, incremented for each revision
    CreateTime int64  // creation time
    AccessTime int64  // interview time
    ModifyTime int64  // Change the time
    LinkTarget []byte // Save the path name of the source file corresponding to the soft link
    NLink      uint32 // number of hard links
    Flag       int32  // Whether the mark can be deleted
    Extents    *SortedExtents // The location information of the file in the three copies
    ObjExtents *SortedObjExtents // The location information of the file erasure code
}
```

MetaNode: in-memory и сайзинг

Inode- и Dentry-деревья целиком находятся в оперативной памяти, а Raft log и снапшоты позволяют восстановить их после перезапуска.

Ёмкость всего кластера ограничена объёмом оперативной памяти, которую мы резервируем под слой метаданных.

Оценка для сайзинга - 1ГБ RAM под metanode на 1 млн объектов.

In-memory B-trees

Dentry · место в дереве

key: (parentId, name)

```
// Dentry defines the dentry struct.
type Dentry struct {
    ParentId  uint64 // inode of the parent directory
     Name      string // The name of the current directory entry
     inode     uint64 // inode of the current file
     Type      uint32 // File Types and Permissions
}
```

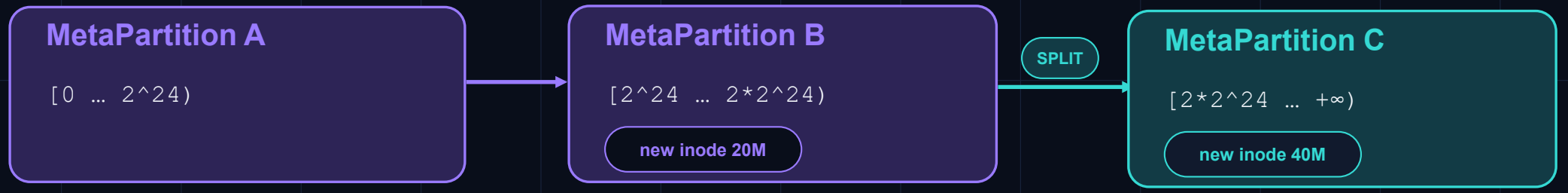
Inode – объект ФС

key: inode ID

```
type inodeInfo struct {
    sync.RWMutex
    inode      uint64 // inode id
    Type       uint32 // file type and permissions
    Uid        uint32 // user id
    Gid        uint32 // group id
    Size       uint64 // File size
    Generation uint64 // The version number, incremented for each revision
    CreateTime int64  // creation time
    AccessTime int64  // interview time
    ModifyTime int64  // Change the time
    LinkTarget []byte // Save the path name of the source file corresponding to the soft link
    NLink      uint32 // number of hard links
    Flag       int32  // Whether the mark can be deleted
    Extents    *SortedExtents // The location information of the file in the three copies
    ObjExtents *SortedObjExtents // The location information of the file erasure code
}
```

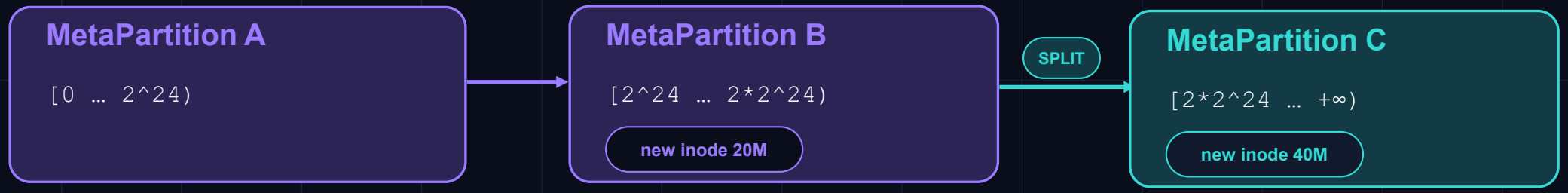
MetaNode: шардирование MetaPartition

MetaPartition делят метаданные по диапазонам inode ID автоматически по достижении порога по RAM



MetaNode: шардирование MetaPartition

MetaPartition делят метаданные по диапазонам inode ID автоматически по достижении порога по RAM



Что шардируется

Dentry records

Запись индексируется парой parentId + name и хранится с parent.

Inode records

Новые inode попадают в следующий диапазон MetaPartition.

Что не шардируется

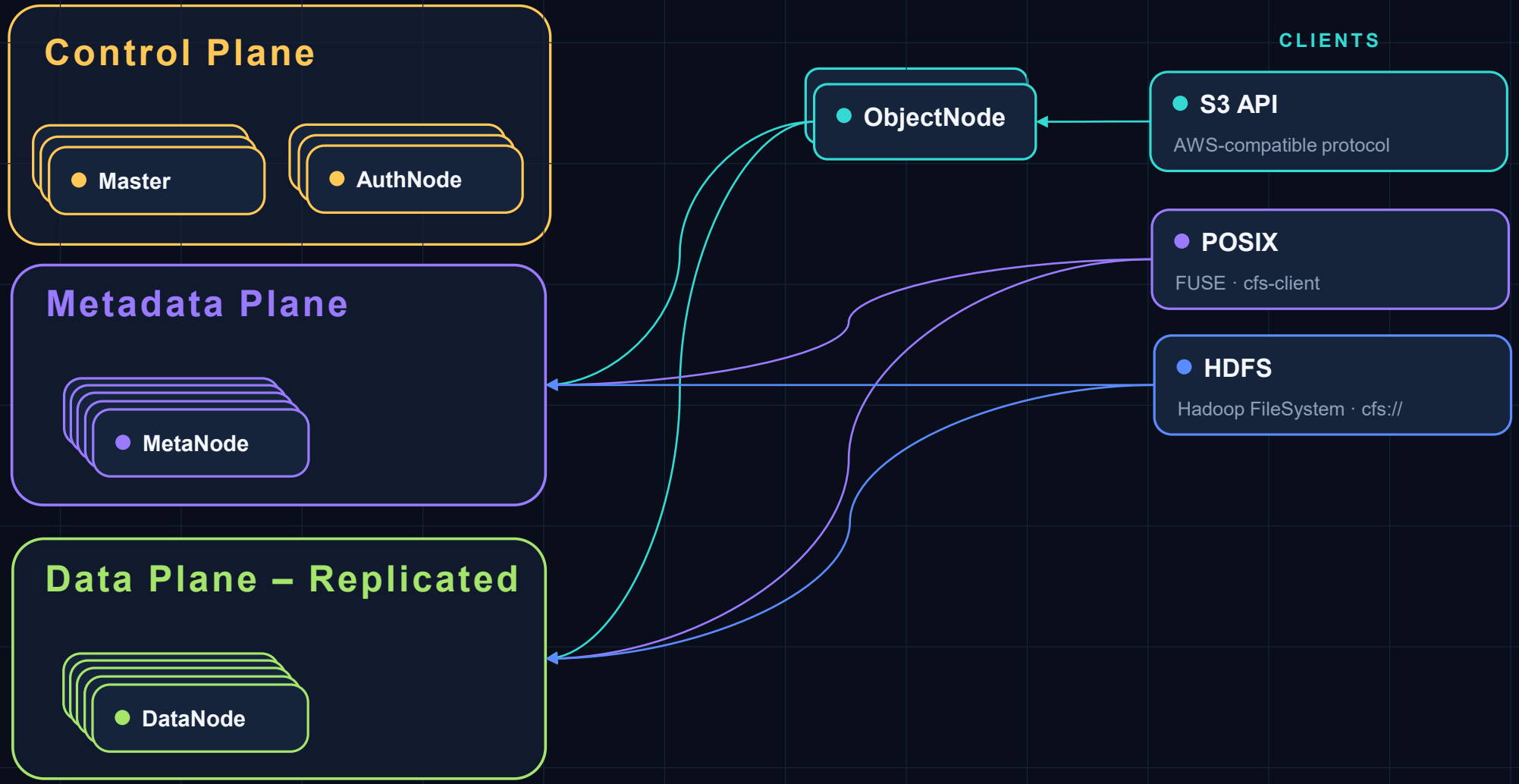
Одна Inode - неделима

Следствие: файлы сверхбольших размеров будут дополнительно нагружать MetaNode за счёт раздутой Inode

Одна Dentry - неделима

Следствие: нельзя создавать папки на >20млн. файлов

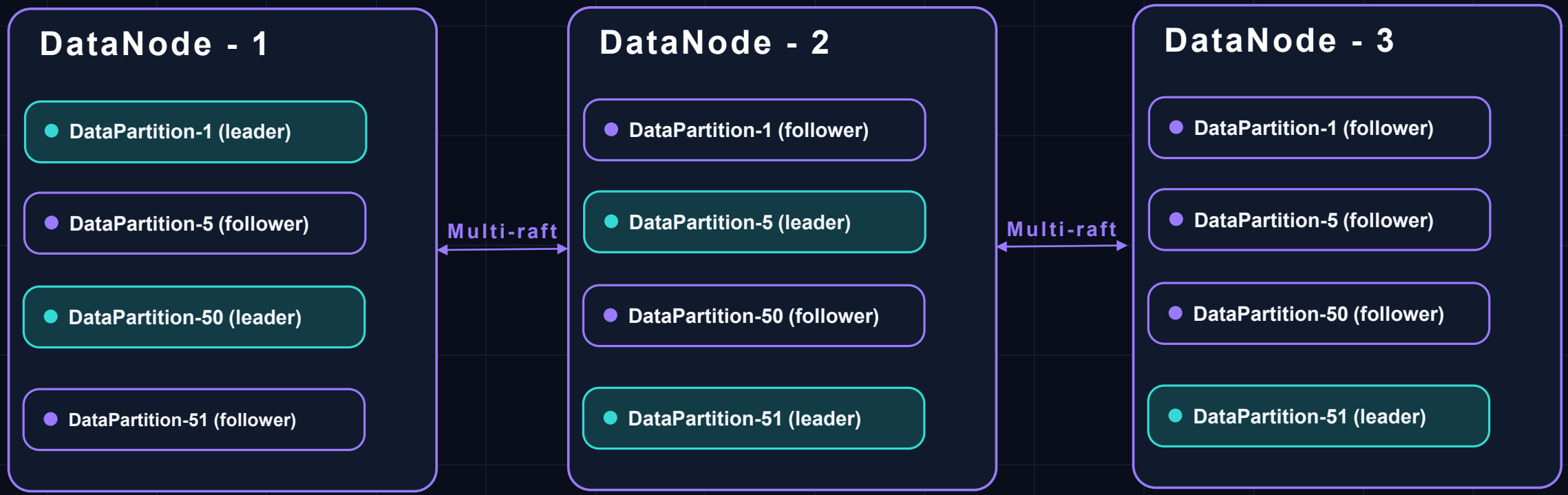
Устройство CubeFS: общая картина



Устройство CubeFS: общая картина

Data plane: Replica subsystem

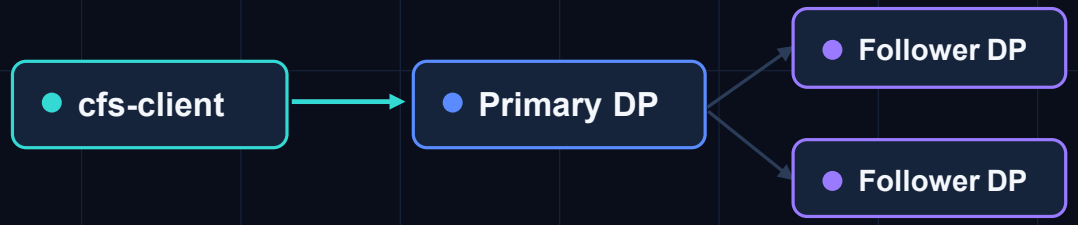
DataPartition реплицируется между разными DataNode с помощью Raft
У каждой DataNode могут быть одновременно и партиции-лидеры, и партиции-фолловеры



~120 GB типовой DataPartition

Data plane: Replica subsystem

Клиент сам выбирает DataPartition для записи, DataPartition рассылает обновления на фолловеров

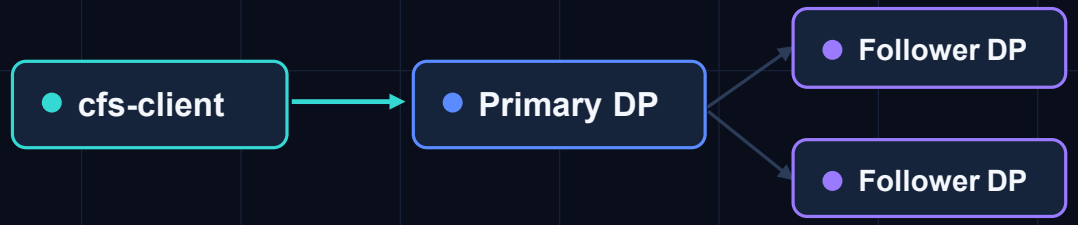


При последовательной записи новых данных Primary DP синхронно рассылает изменения во все follower DP. Если это не удалось – клиент повторяет попытку уже в другую партицию.

При перезаписи существующих данных используется ослабленное требование – достаточно кворума из DP, т.к. возможности ретраиться в другие DP у клиента нет.

Data plane: Replica subsystem

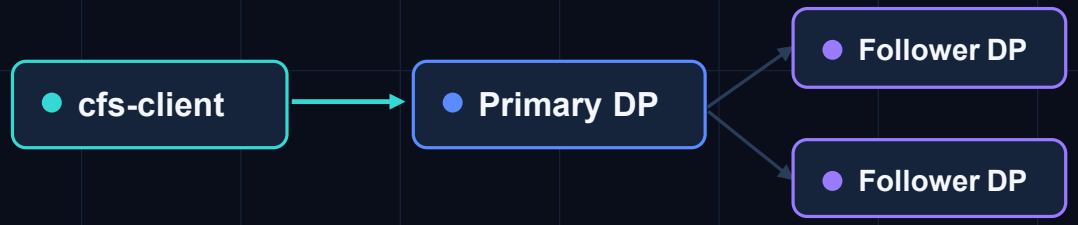
Клиент сам выбирает DataPartition для записи, DataPartition рассылает обновления на фолловеров



Обычные и крупные объекты записываются в файлы NormalExtent блоками по 128КБ. Файл NormalExtent ограничен 1024 блоками.

Data plane: Replica subsystem

Клиент сам выбирает DataPartition для записи, DataPartition рассылает обновления на фолловеров



NormalExtent <=128 MB

Блоки по 128 KB

Обычные и крупные объекты записываются в файлы NormalExtent блоками по 128КБ. Файл NormalExtent ограничен 1024 блоками.

TinyExtent 64 файла на partition, размер файла – до 4 TiB

Блоки по 4 KB

Для небольших объектов есть специализированный путь — файлы TinyExtent.

CubeFS агрегирует много маленьких объектов в общие append-файлы.

Data plane: Replica subsystem – особенности

Простота и надёжность

Нижележащее хранилище – привычная файловая система (ext4, xfs, btrfs, etc.)

Поддержка in-place перезаписи объектов

Вместо copy-on-write на весь блок или файл

Прозрачное multi-tier хранилище

Данные автоматически перемещаются в более дешевые хранилища в соответствии с политиками жизненного цикла

Управляемое горизонтальное масшабирование чтения-записи

Для каждого Volume можем задавать, сколько активных Meta- и Data-партиций держать

Data plane: Replica subsystem – особенности

Простота и надёжность

Нижележащее хранилище – привычная файловая система (ext4, xfs, btrfs, etc.)

Поддержка in-place перезаписи объектов

Вместо copy-on-write на весь блок или файл

Прозрачное multi-tier хранилище

Данные автоматически перемещаются в более дешевые хранилища в соответствии с политиками жизненного цикла

Управляемое горизонтальное масшабирование чтения-записи

Для каждого Volume можем задавать, сколько активных Meta- и Data-партиций держать

Нет полноценной поддержки снепшотов

В cubefs реализован механизм снепшотов, но на данный момент апстрим не поддерживает multi-tier volume’ы и функциональность отключена по-умолчанию

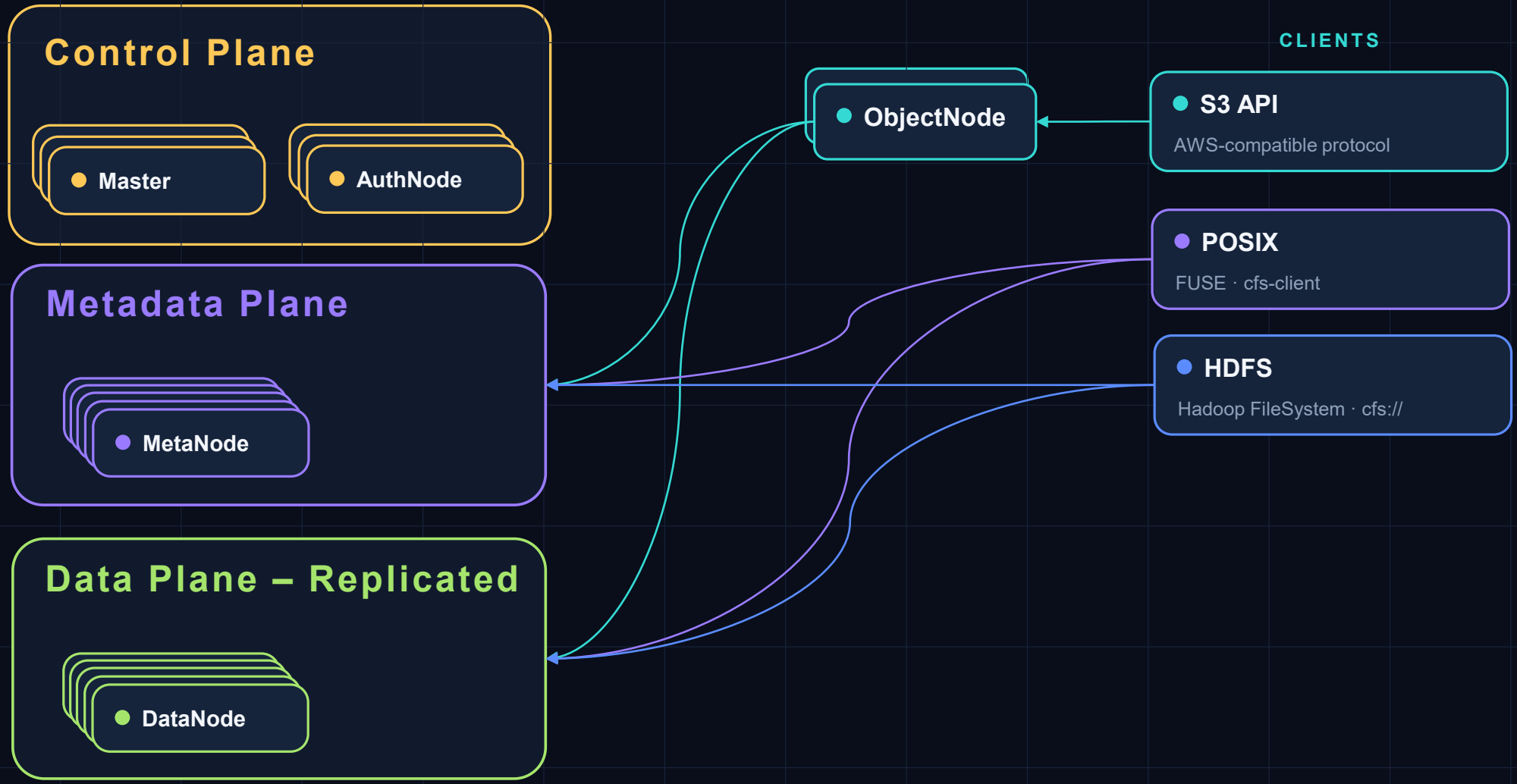
Асинхронная проверка размера партиций

Заполненные партиции не сразу переходят в RO по достижении порога в 120 ГБ. Рекомендуется добавлять 5-10% запаса при расчетах сайзинга

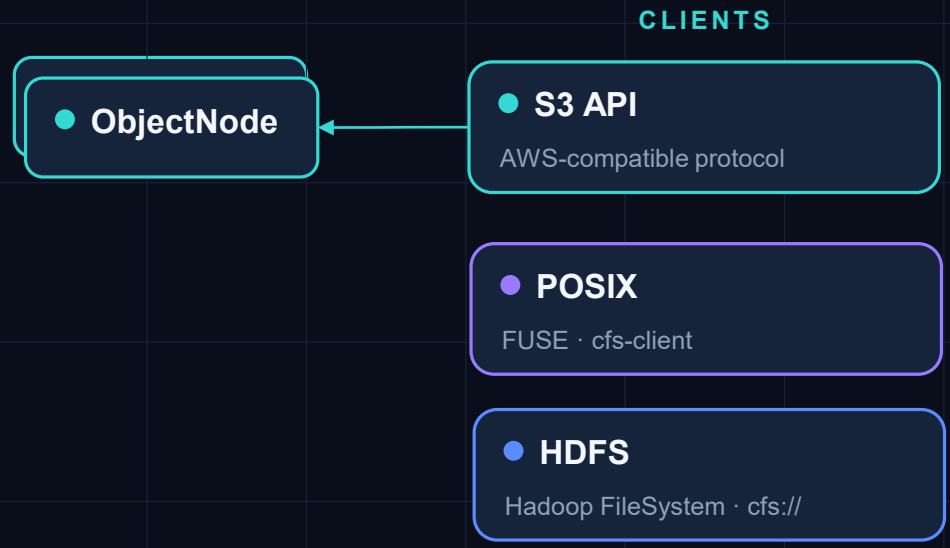
Отсутствует авто-детекция bitrot

Для блоков в Data-партициях сохраняются CRC-чексуммы, но нет аналогов Deep Scrub ceph’a. Автоматика проверяет на I/O ошибки диска и расхождения в размерах extent’ов (неполную запись)

Устройство CubeFS: общая картина

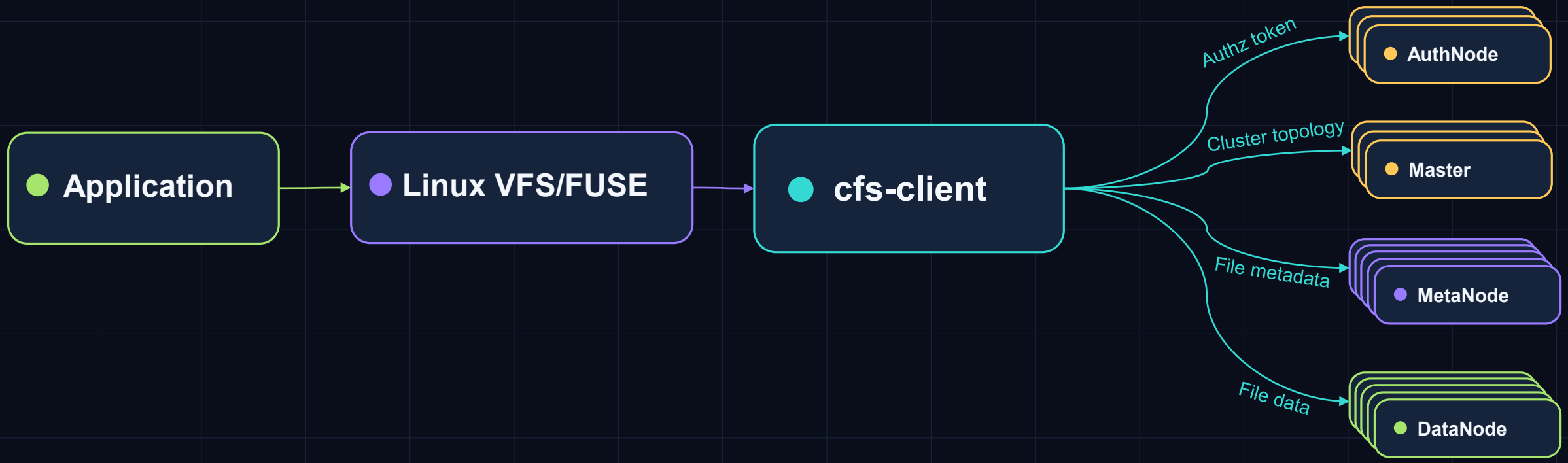


Устройство CubeFS: общая картина

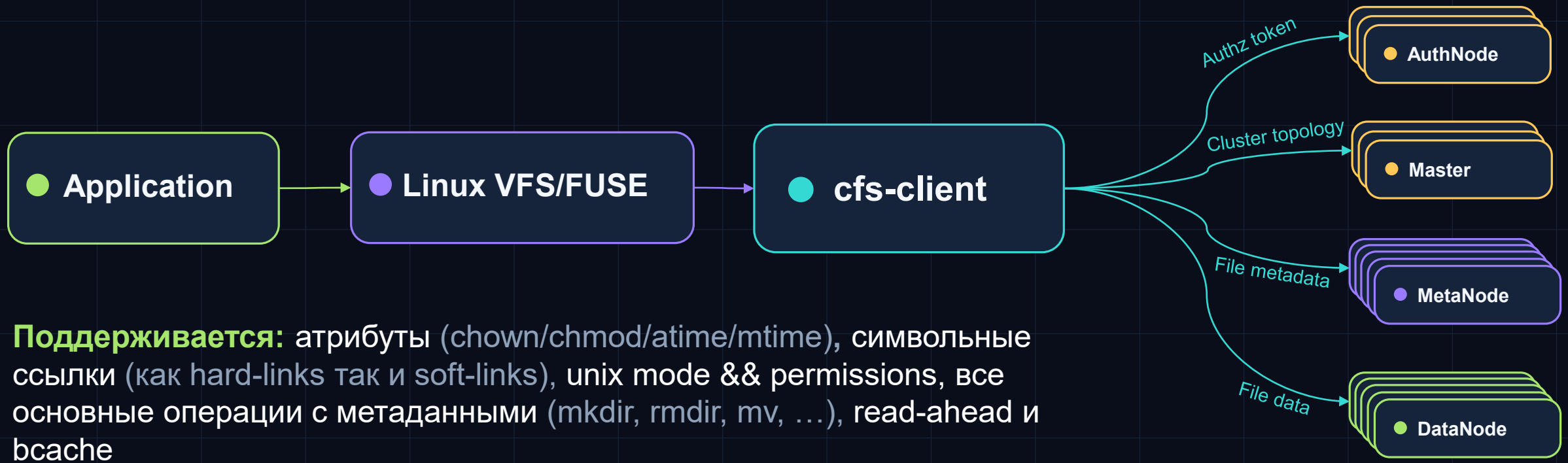


POSIX: возможности и ограничения файловой системы

POSIX: возможности и ограничения файловой системы



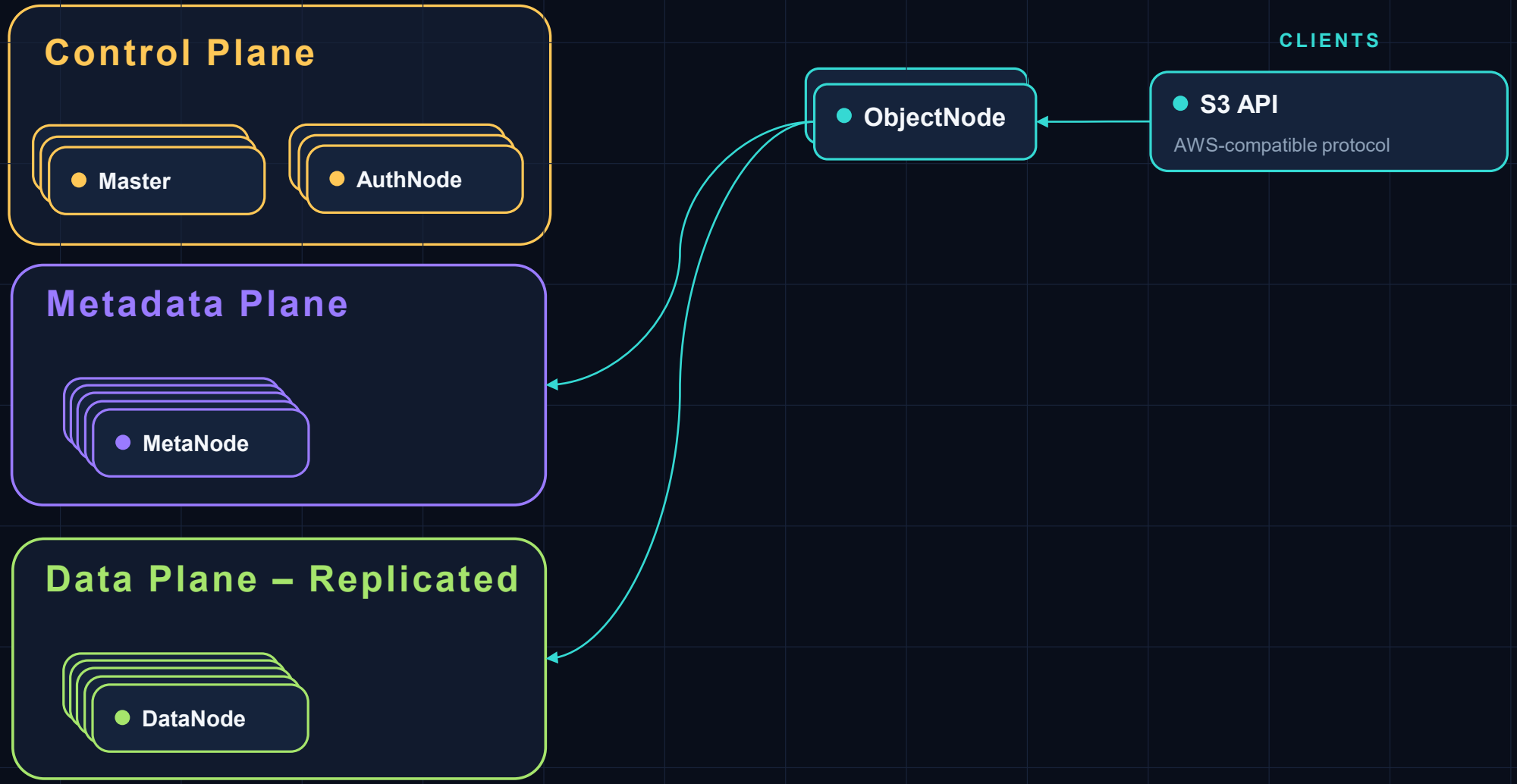
POSIX: возможности и ограничения файловой системы



Поддерживается: атрибуты (chown/chmod/atime/mtime), символьные ссылки (как hard-links так и soft-links), unix mode && permissions, все основные операции с метаданными (mkdir, rmdir, mv, …), read-ahead и bcache

Не поддерживается: flock и распределенные блокировки на запись, fsnotify

S3 API: возможности и ограничения ObjectNode



S3 API: возможности и ограничения ObjectNode

PUT events/2026/08/part.parquet

Ключи S3 объектов становятся реальными папками и файлами во внутреннем представлении

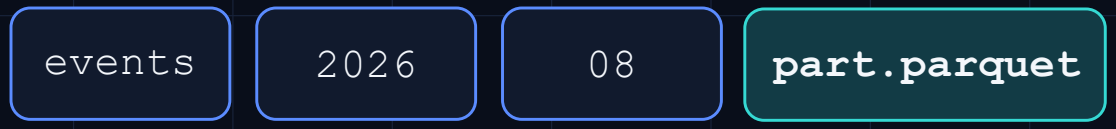
При создании объекта рекурсивно создаются все недостающие папки в дереве

При удалении объекта в штатном CubeFS удаляется только непосредственно файл. Папки остаются на месте.

У себя реализовали опциональный механизм рекурсивной очистки при удалениях объектов через S3 API, получилось дёшево.

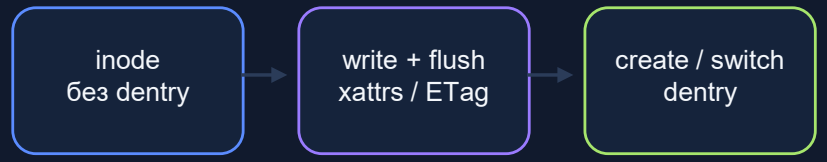
S3 API: возможности и ограничения ObjectNode

PUT events/2026/08/part.parquet



Ключи S3 объектов становятся реальными папками и файлами во внутреннем представлении

PUT большого объекта – атомарный



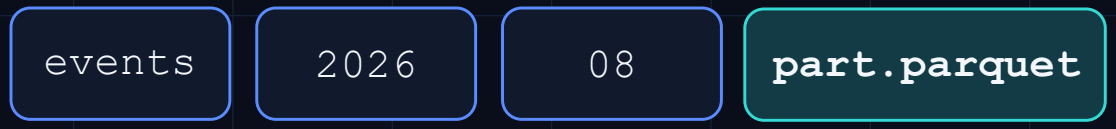
Читатель не увидит частично загруженный объект

Multipart – атомарный и параллельный поверх одной inode

При завершении ObjectNode проставляет правильные офсеты загруженных данных и атомарно публикует inode аналогично с единичным PUT

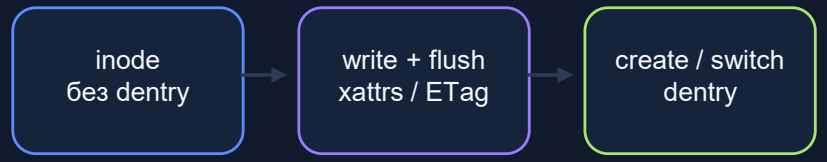
S3 API: возможности и ограничения ObjectNode

`PUT events/2026/08/part.parquet`



Ключи S3 объектов становятся реальными папками и файлами во внутреннем представлении

PUT большого объекта – атомарный



Читатель не увидит частично загруженный объект

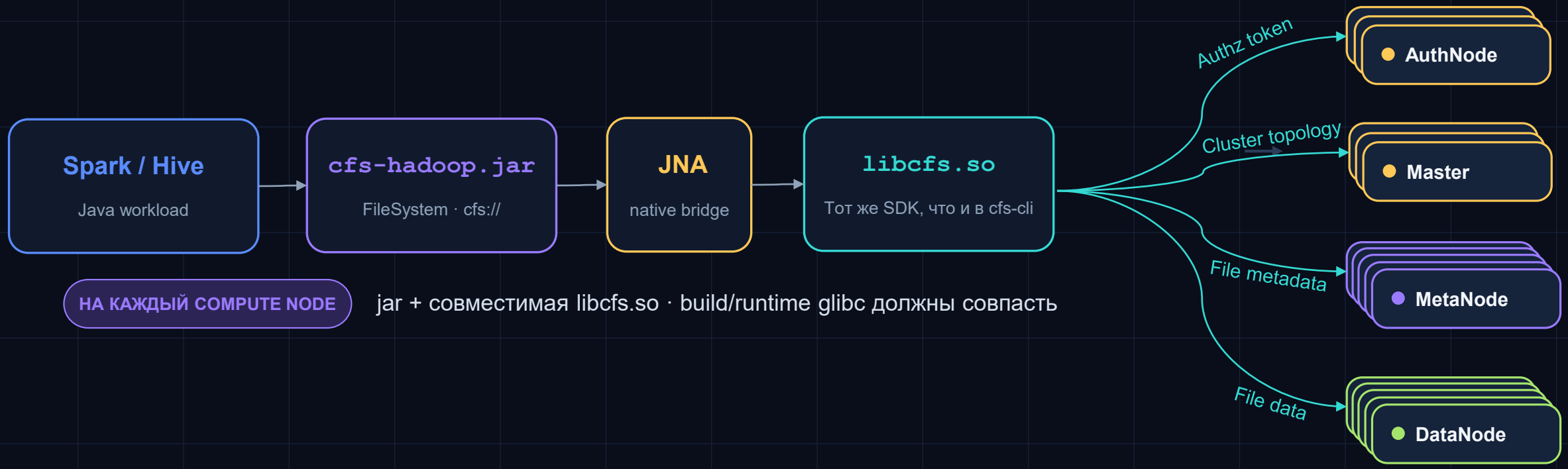
Multipart – атомарный и параллельный поверх одной inode

При завершении ObjectNode проставляет правильные офсеты загруженных данных и атомарно публикует inode аналогично с единичным PUT

Ограничения, которые слишком сложно преодолеть:

- Версионирование объектов
- Невозможность создать объект a/b и затем создать объект a/b/c
- Тяжёлый ListObjects, который требует запроса по множеству MetaNode

HDFS adapter: возможности и ограничения



НА КАЖДЫЙ COMPUTE NODE

jar + совместимая libcfs.so · build/runtime glibc должны совпасть

HDFS adapter: возможности и ограничения

НА КАЖДЫЙ COMPUTE NODE

jar + совместимая libcfs.so · build/runtime glibc должны совпасть

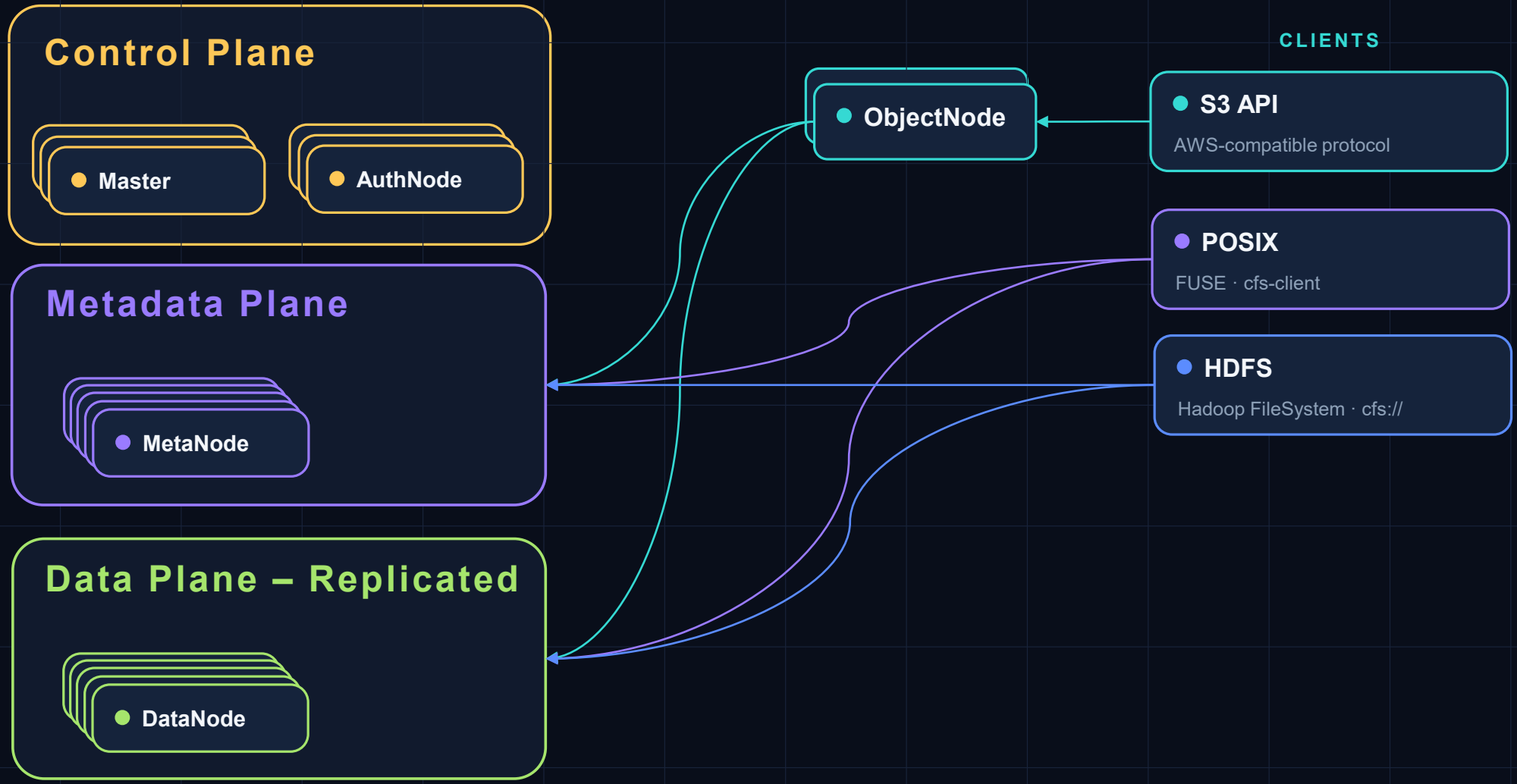
Базовая совместимость

Hadoop FileSystem API и привычная схема cfs:// для legacy jobs.

Без поддержки data locality

Block locations синтетические: scheduler не видит физическое размещение blocks.

Устройство CubeFS: общая картина



Опыт и нюансы эксплуатации

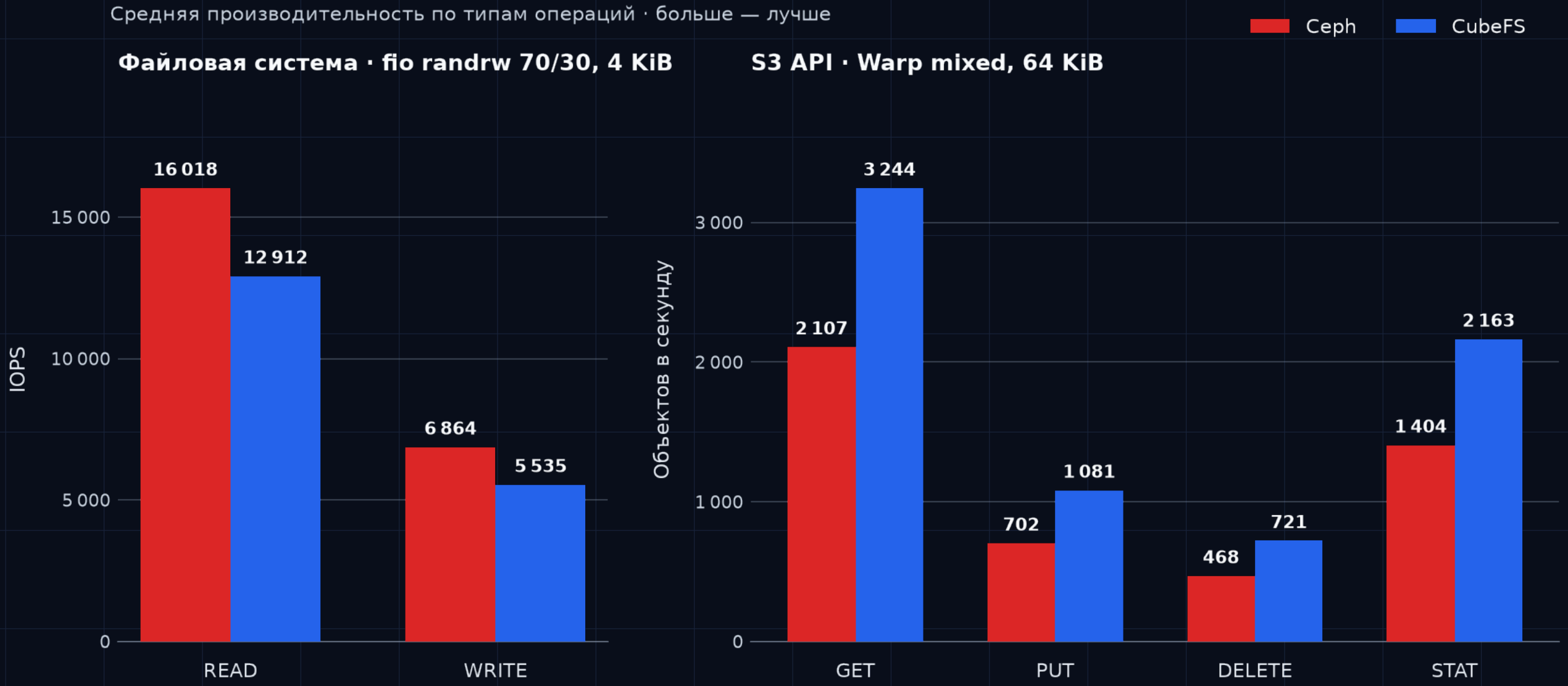
- Нет Backfill на добавлении новых нод, кластер может оставаться неравномерно заполненным
- UI де-факто заброшен, необходимо писать свой
- Документация и логи достаточно сложные для восприятия
- Меньше удобства в управлении топологией, чем у Ceph
- Развертывание в k8s helm-чартом == невозможность управлять топологией кластера в рантайме)
- Мало lifecycle-политик (не до конца реализован Expiration по времени)

Опыт и нюансы эксплуатации

- Нет Backfill на добавлении новых нод, кластер может оставаться неравномерно заполненным
- UI де-факто заброшен, необходимо писать свой
- Документация и логи достаточно сложные для восприятия
- Меньше удобства в управлении топологией, чем у Ceph
- Развертывание в k8s helm-чартом == невозможность управлять топологией кластера в рантайме)
- Мало lifecycle-политик (не до конца реализован Expiration по времени)
- Написан на Go и легко дорабатывается
- Экономичный по ресурсам, легко разворачивается в Kind
- Нет Backfill на добавлении новых нод => нет больших перемещений данных при вводе :)

CubeFS и Ceph: сравнительная производительность

Средняя производительность по типам операций · больше — лучше



Тесты проводились на dev-кластере из 4x машин с 16 vCPU, 32 GB RAM, 1000 QLC SSD, 10 Gb/s ethernet
Перепроверяйте результаты на своих мощностях и конфигурациях!

CubeFS сейчас

- Быстрая работа с метаданными на миллиардах объектов
- Одновременная поддержка S3, HDFS и POSIX API над одними и теми же данными
- Поддержка частичной перезаписи объектов
- Многоуровневое кэширование
- Прозрачное гибридное хранилище

Но нужен ресурс разработки/SRE на доработку для удобной эксплуатации!