Swift (2) – 実装と運用tips

OSS基盤技術センター
OSS技術第二課

項目

- Swiftのデータ構造と処理フロー
- 障害時の動作と対策
- 高負荷対策
- バックアップ
- 実践編: Swift機能拡張例
  - 既存システムとの認証統合
  - クォータ機能の実装
- その他

今回説明を省略するもの

- インストールと初期設定
- コンフィグ詳細
- ツール群の使い方
- REST API
- swauthの認証モデルと実装詳細
- 小規模構成の話（ロードバランサ無し構成、SIAO構成）
- Amazon S3 （時間が足りないので）

Swiftの構造

- クラスタ構成
  - 多数のプロキシノード、アカウントノード、コンテナノード、オブジェクトノードが並列動作
  - システム構成を集中管理するノードが存在しない
  - 動的に拡張・構成変更可能
- データ冗長化
  - ゾーンという考え方
  - リング

Swiftの構成ノード

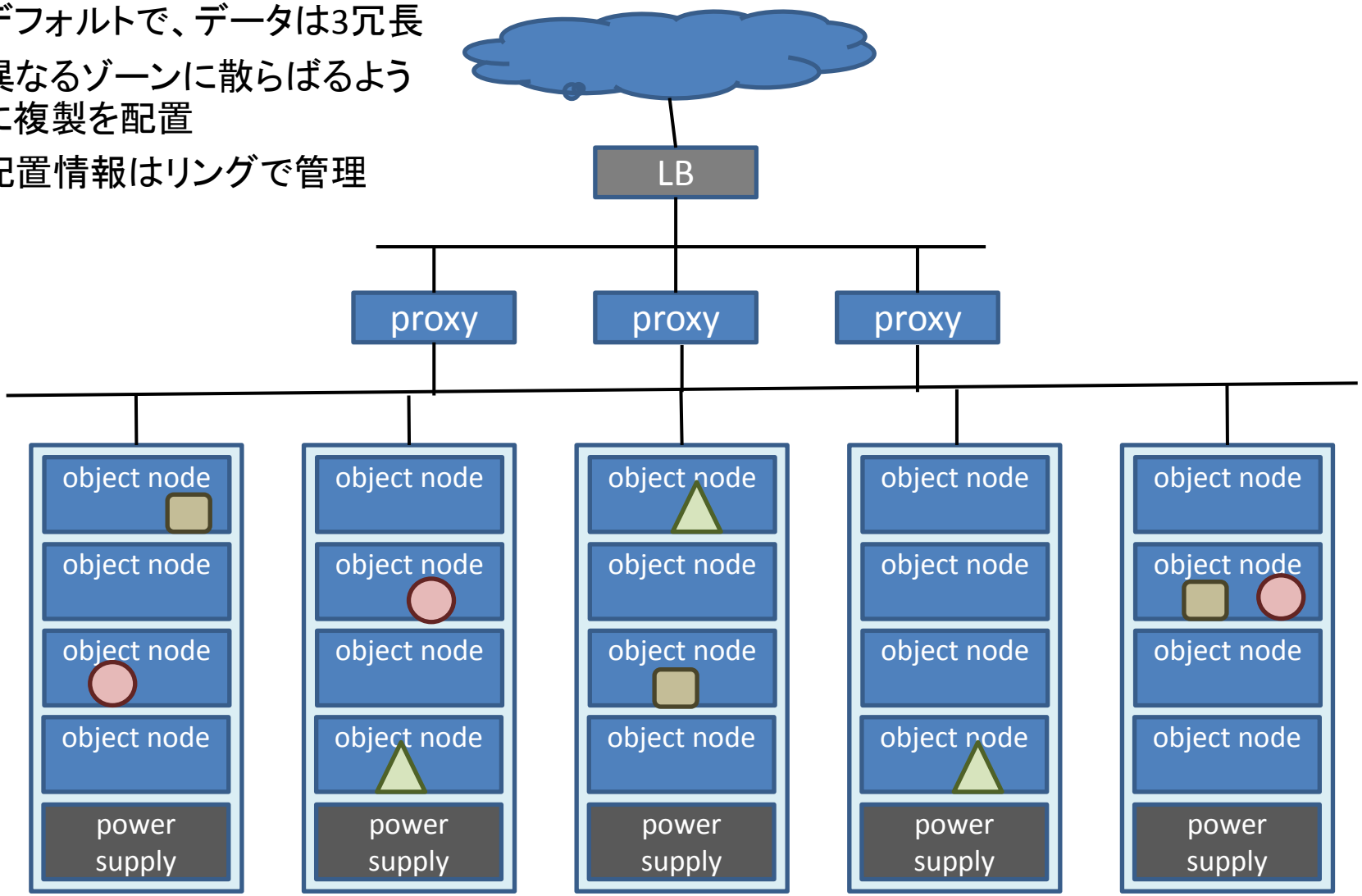
- プロキシノード
  - ReST APIを実現。クライアントからの要求を、データを管理する適切なノードに転送する
- アカウントノード
  - アカウント情報をSQLite3データベースを用いて管理する。コンテナ名の一覧、アカウント全体の統計情報などを管理
- コンテナノード
  - コンテナ情報をSQLite3データベースを用いて管理する。オブジェクト名の一覧、ACL情報などを管理
- オブジェクトノード
  - オブジェクトの実体を保存する
- 各ノードが、同一ハードウェアに同居することも可能

アカウント、コンテナ、オブジェクトノードで動作するデーモン

- サーバ　server
  - GET/PUT/DELETEなどのリクエストを処理する
- レプリケータ replicator
  - データ修復や、不要となったデータの削除を行う
- アップデータ updater
  - 上位ノードへの通知が失敗したとき、アップデータが再試行を行う
- オーディタ auditor
  - データの破損していないか周期的に確認する
- リーパ reaper
  - アカウントを削除したとき、配下のコンテナやオブジェクトを削除する。アカウントノードで動作する。

ゾーンとレプリカ(複製)

- デフォルトで、データは3冗長
- 異なるゾーンに散らばるように複製を配置
- 配置情報はリングで管理

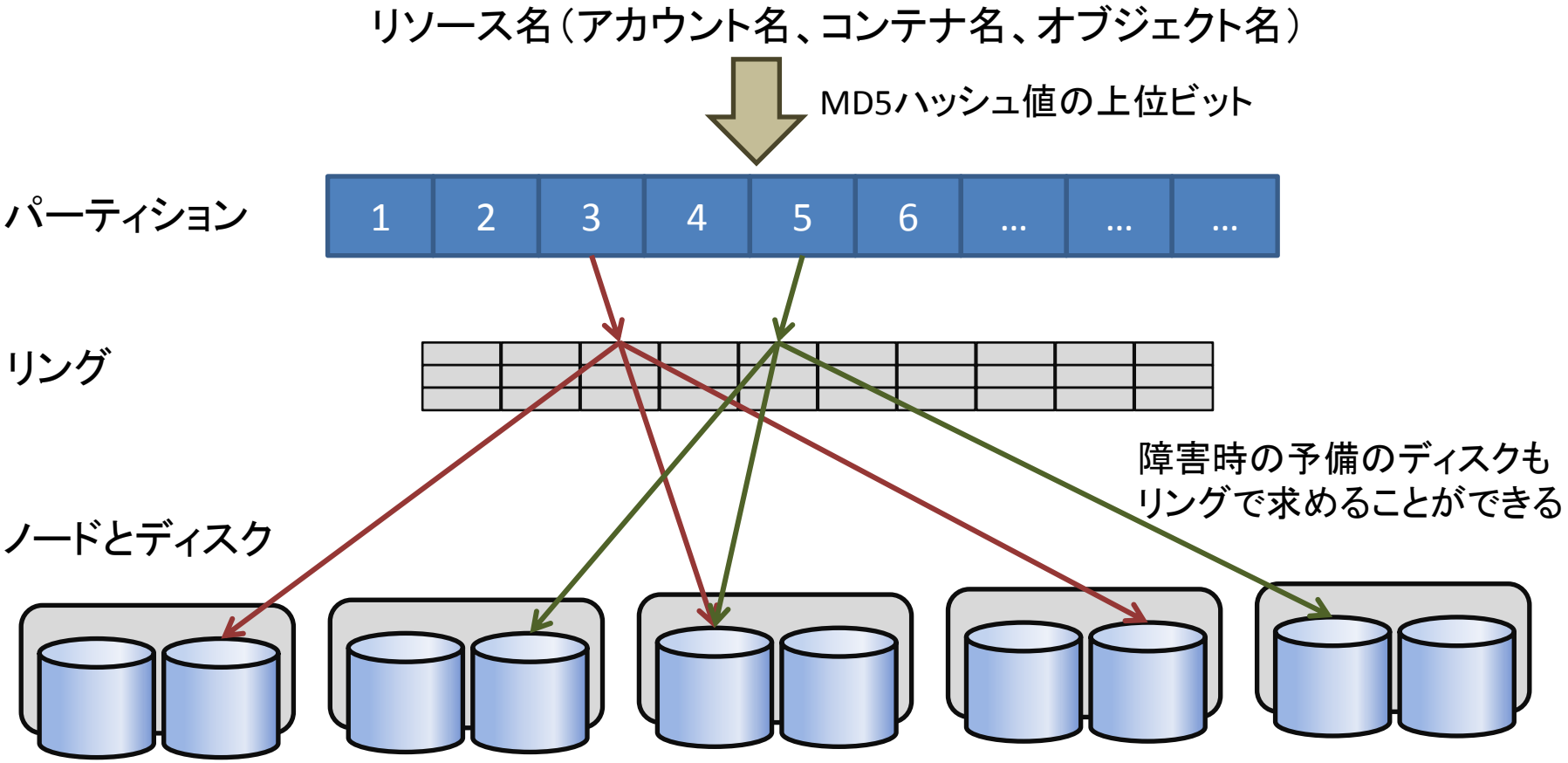


リングの考え方

- 狙う効果
  - スケーラビリティ。高度な分散処理の実現
  - 耐故障性
- 実現方法
  - データ配置を静的に決定してしまう
    - 動的な空き領域の確保、データ配置位置の記録が不要
    - リソース名(コンテナ名やオブジェクト名）が決まれば、データの配置位置が一意に決まる
  - アカウント、コンテナ、オブジェクト用それぞれに配置情報を管理するリングを用意
  - すべてのノードに、同じリングを配布

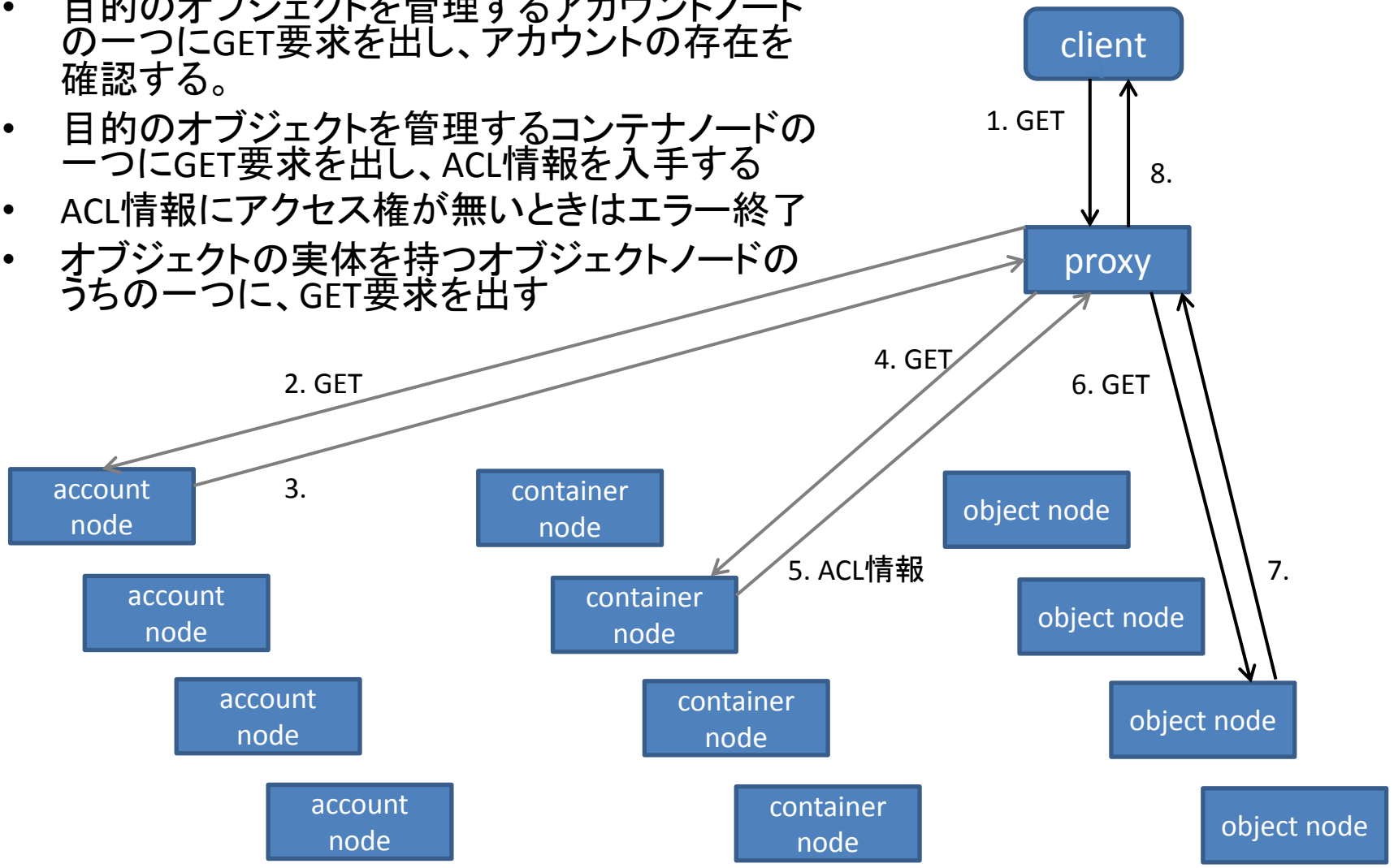
リソース名とデータ格納位置の対応

- リソースをグルーピング（パーティション）
- リングは、パーティションとディスクの対応表を持つ



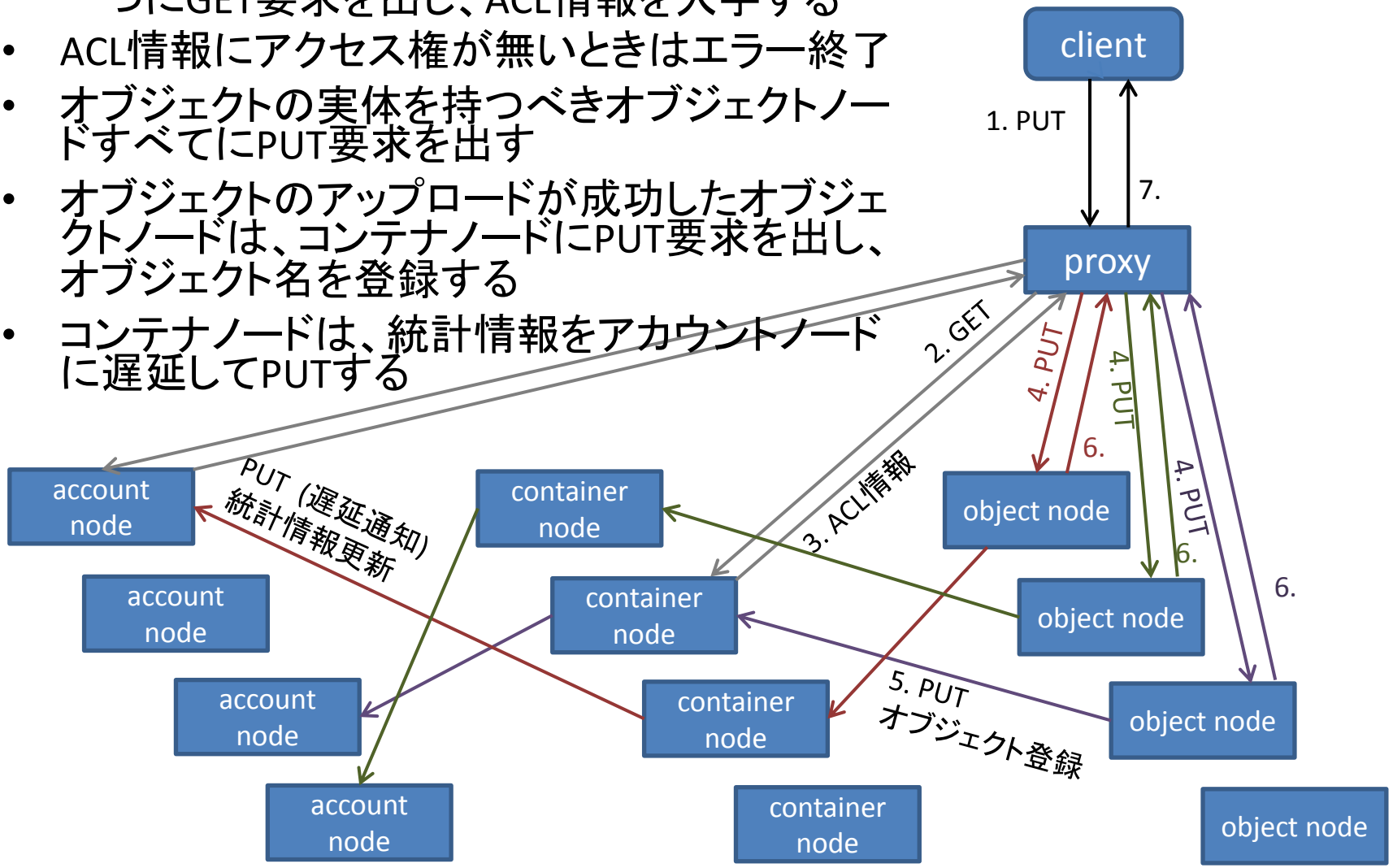
オブジェクトのGET処理フロー

- 目的のオブジェクトを管理するアカウントノードの一つにGET要求を出し、アカウントの存在を確認する。
- 目的のオブジェクトを管理するコンテナノードの一つにGET要求を出し、ACL情報を入手する
- ACL情報にアクセス権が無いときはエラー終了
- オブジェクトの実体を持つオブジェクトノードのうちの一つに、GET要求を出す

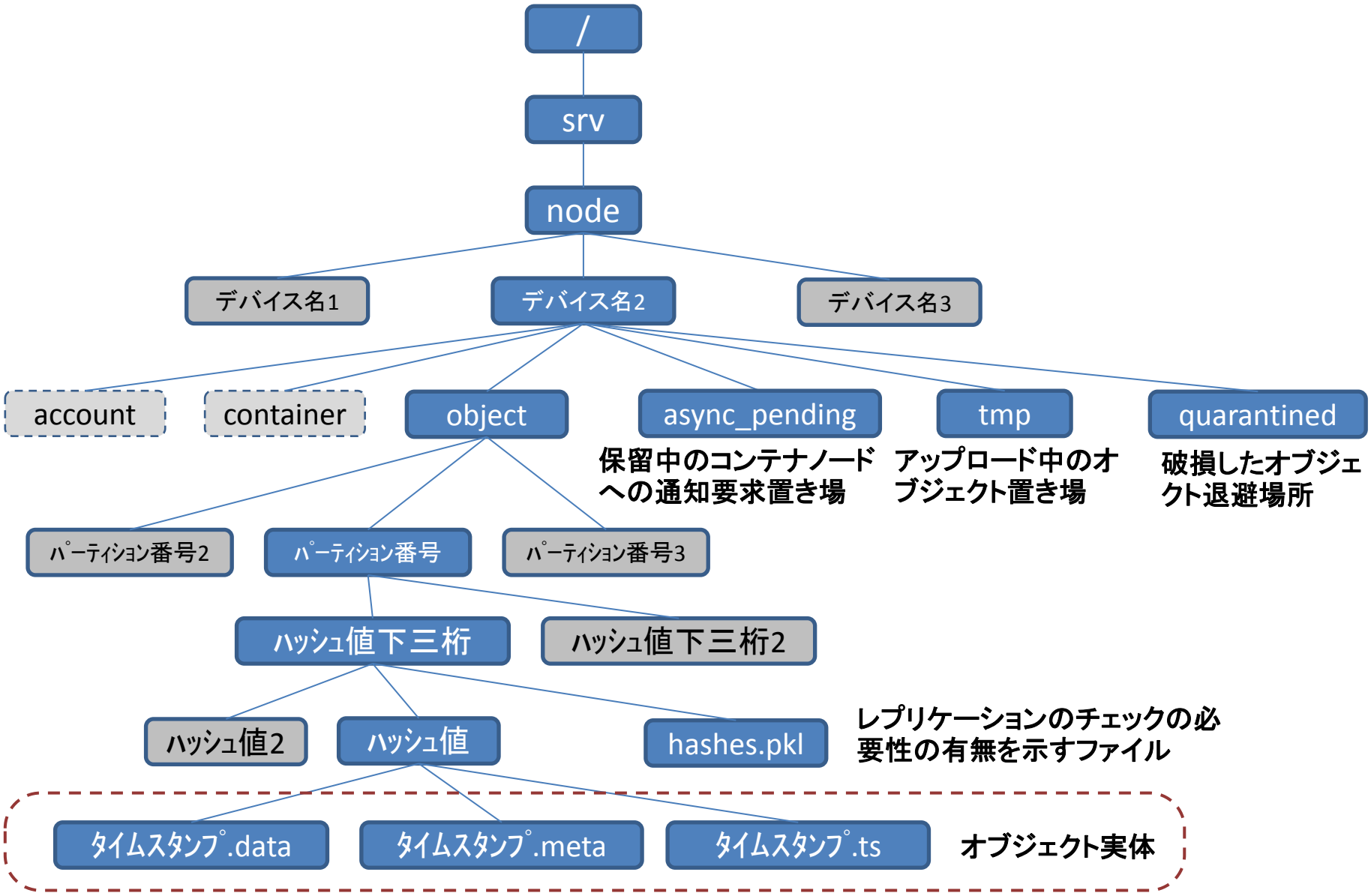


オブジェクトのPUT処理フロー

- 目的のオブジェクトを管理するコンテナノードの一つにGET要求を出し、ACL情報を入手する
- ACL情報にアクセス権が無いときはエラー終了
- オブジェクトの実体を持つべきオブジェクトノードすべてにPUT要求を出す
- オブジェクトのアップロードが成功したオブジェクトノードは、コンテナノードにPUT要求を出し、オブジェクト名を登録する
- コンテナノードは、統計情報をアカウントノードに遅延してPUTする

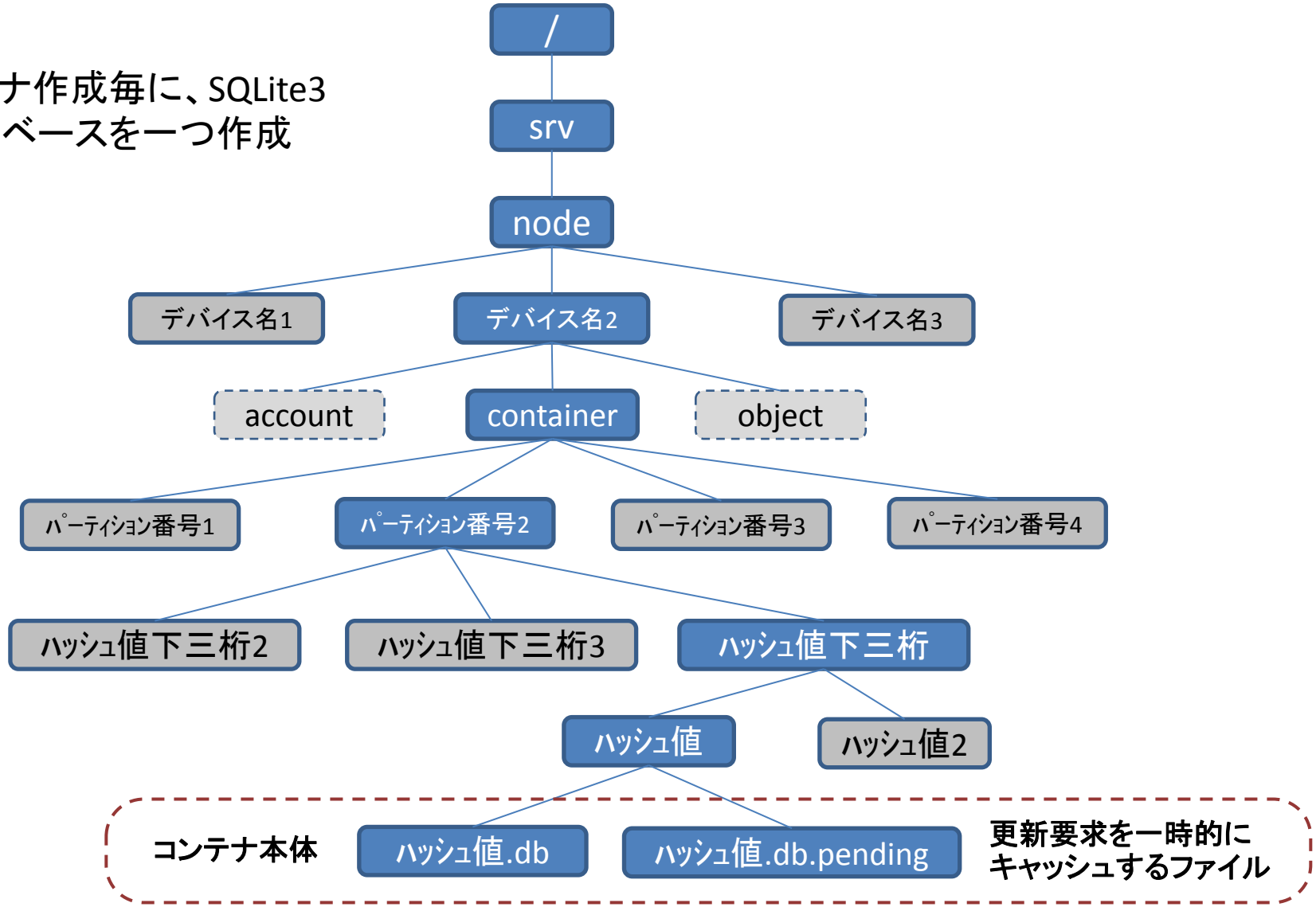


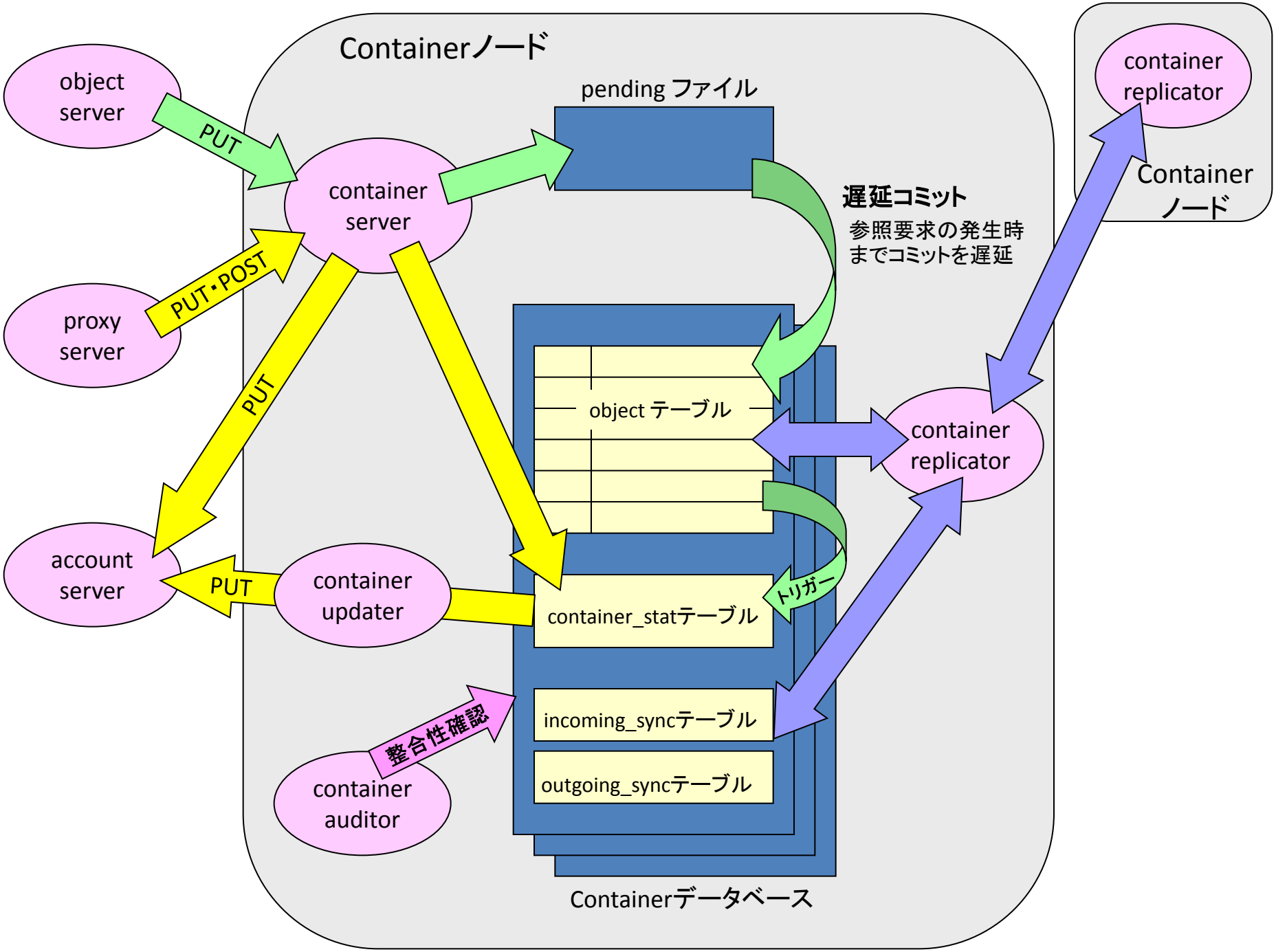
オブジェクトノードディレクトリ構成



コンテナノードディレクトリ構成

コンテナ作成毎に、SQLite3
データベースを一つ作成

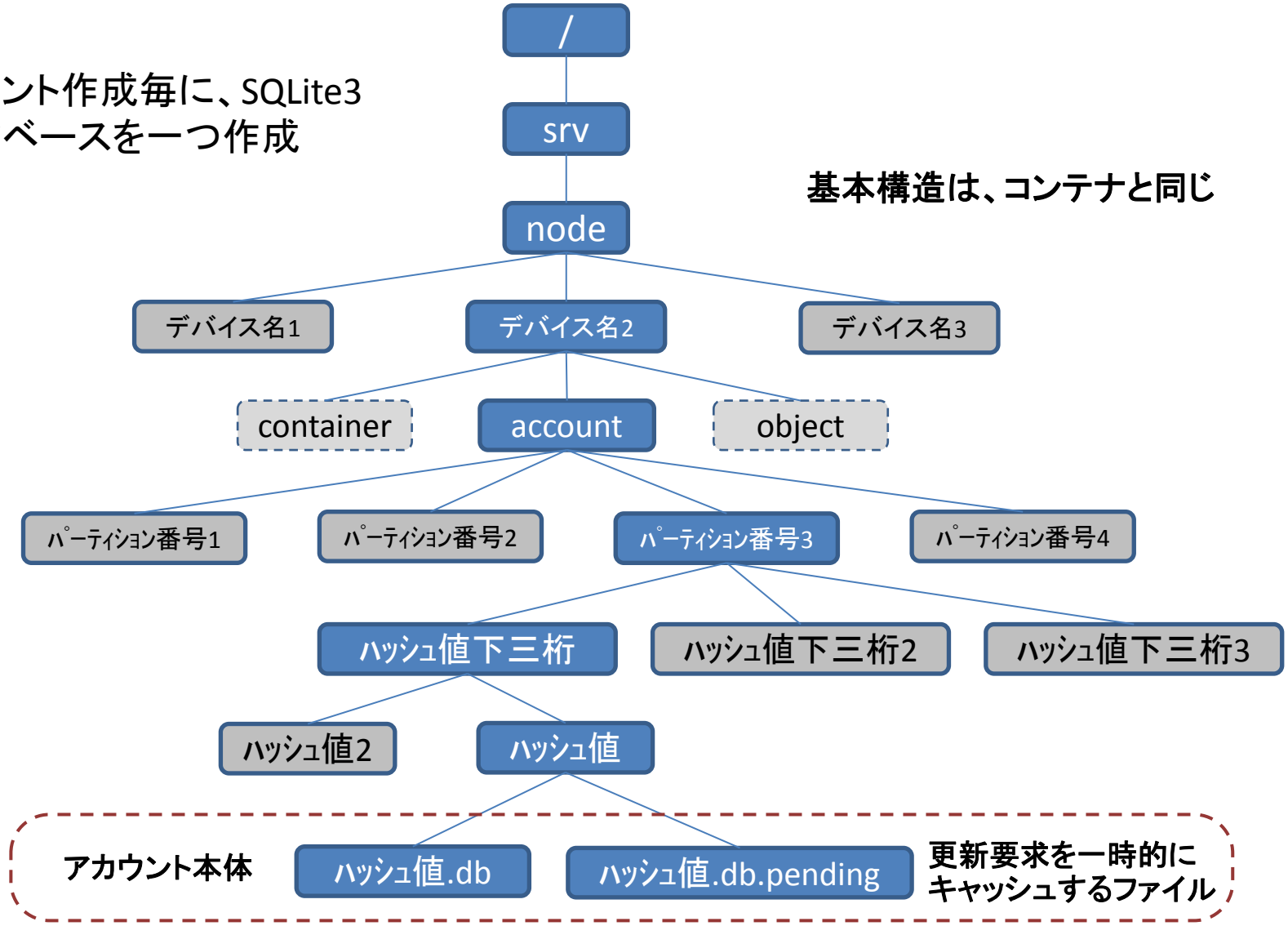


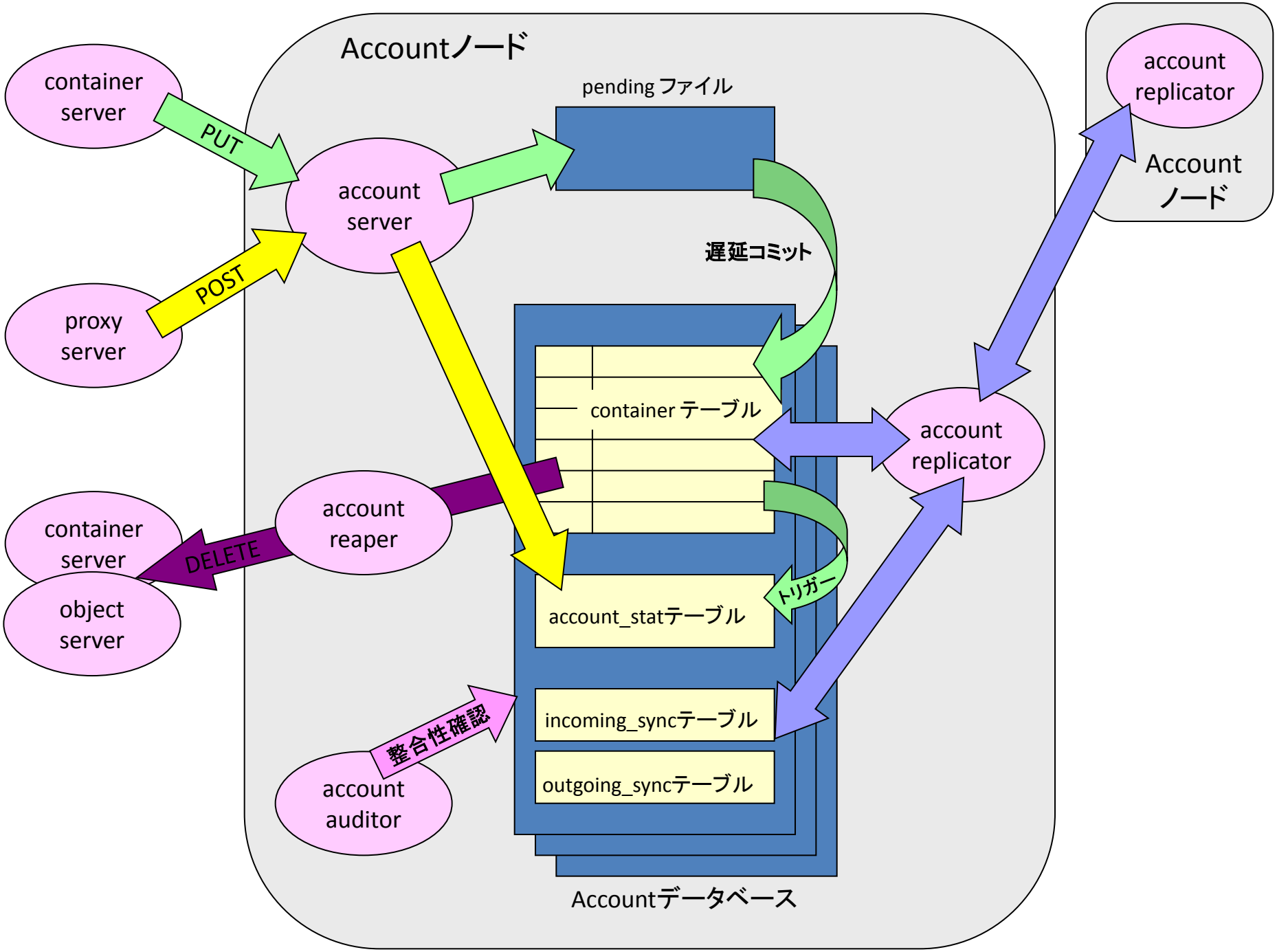

Containerノード
object server
PUT
container server
pending ファイル
遅延コミット
参照要求の発生時までコミットを遅延
container replicator
Container ノード
proxy server
PUT・POST
PUT
object テーブル
container replicator
account server
PUT
container updater
container_statテーブル
トリガー
incoming_syncテーブル
整合性確認
container auditor
outgoing_syncテーブル
Containerデータベース

アカウントノードディレクトリ構成

アカウント作成毎に、SQLite3
データベースを一つ作成

基本構造は、コンテナと同じ



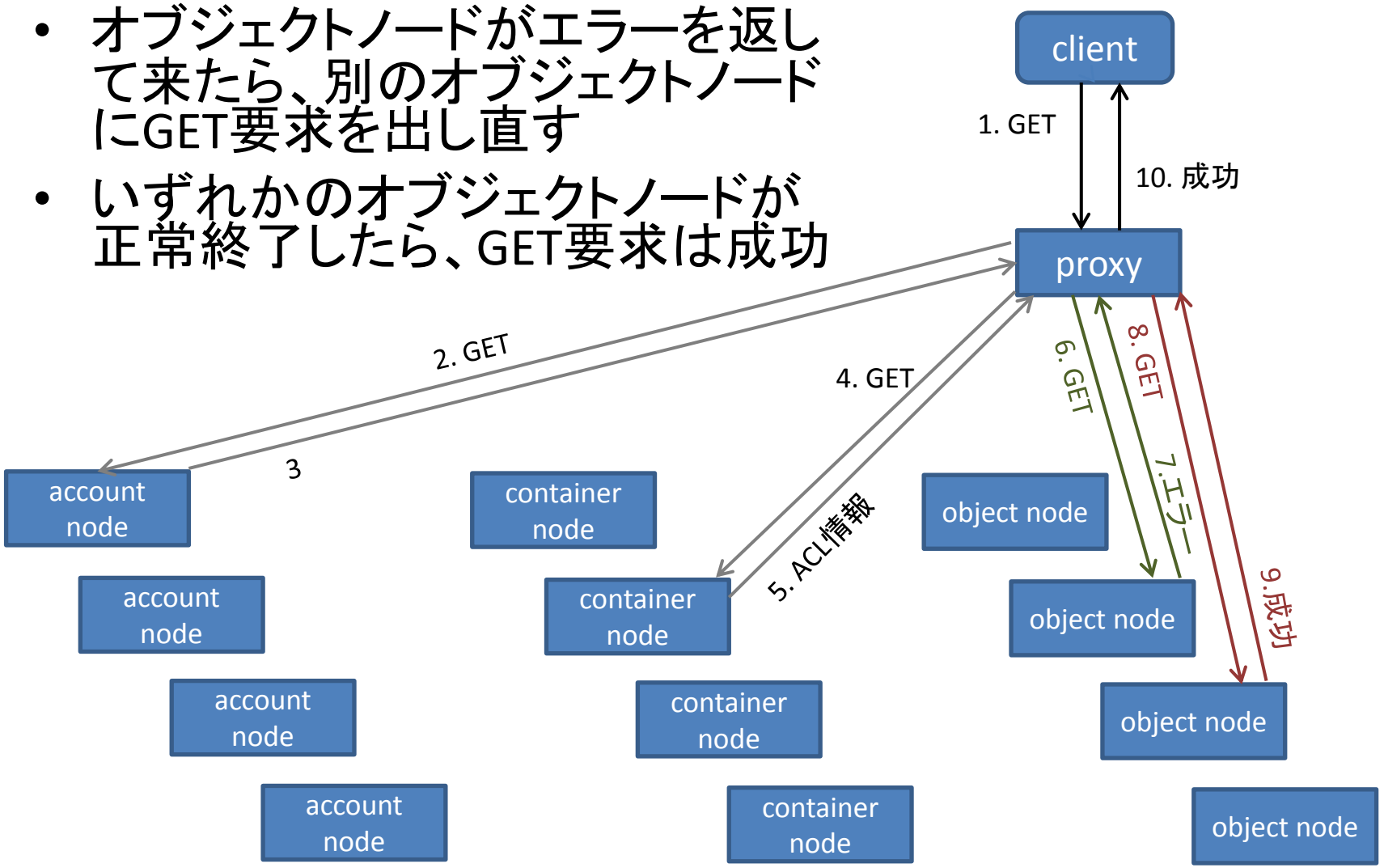

Accountノード
container server
PUT
account server
pending ファイル
遅延コミット
proxy server
POST
container テーブル
account replicator
account reaper
container server
DELETE
object server
トリガー
account_statテーブル
incoming_syncテーブル
整合性確認
account auditor
outgoing_syncテーブル
Accountデータベース
account replicator
Account ノード

ノード障害、ネットワーク障害

- 通信障害は容易に発生する
  - 応答性確保のための非常に短いタイムアウト設定
- クライアントからのリクエストは、簡単にはエラーにならない
- レプリケータによるデータの自動復元
- アップデータによる通信の再試行
- 代替ノードの利用

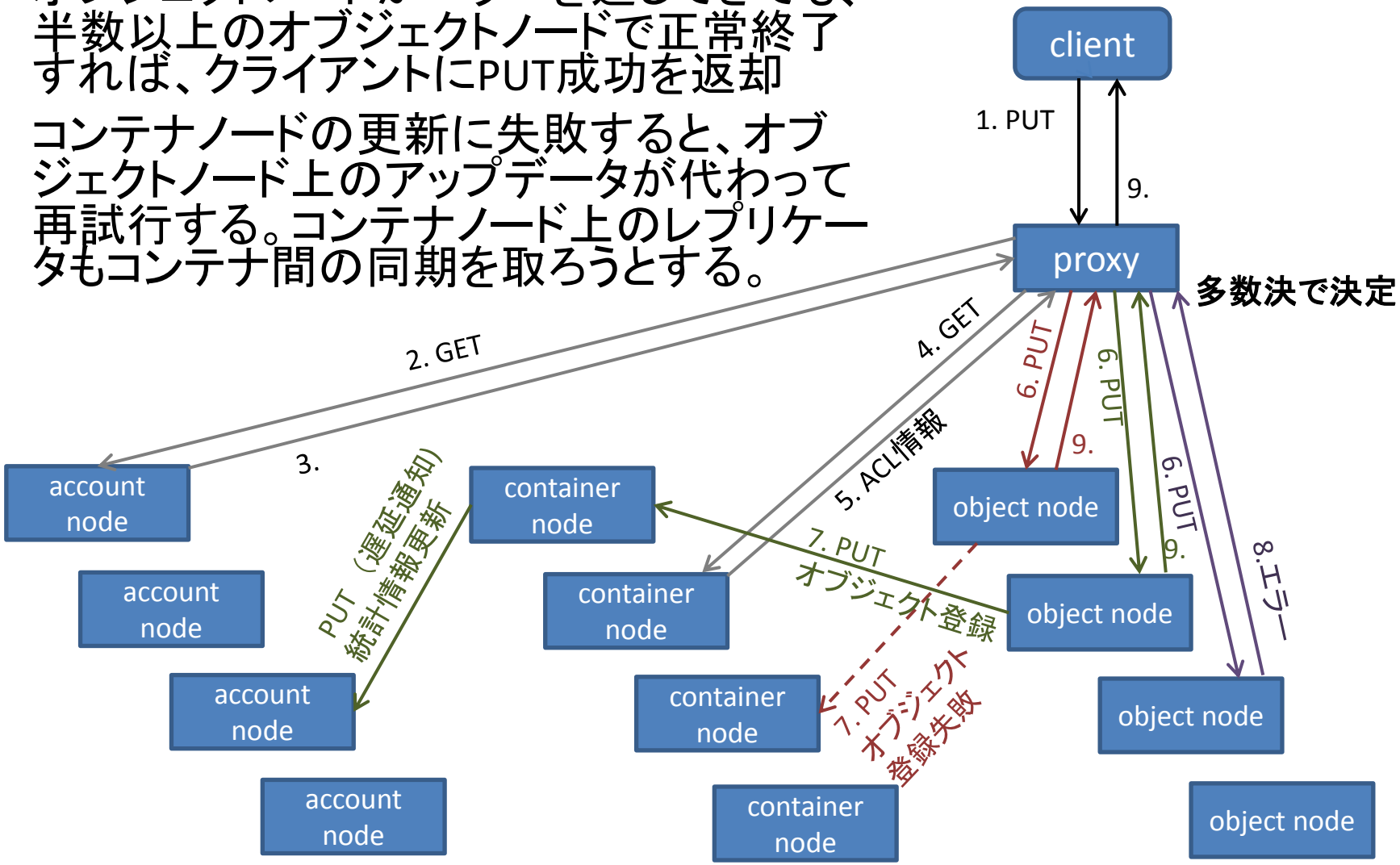
オブジェクトのGET処理フロー

- オブジェクトノードがエラーを返して来たら、別のオブジェクトノードにGET要求を出し直す
- いずれかのオブジェクトノードが正常終了したら、GET要求は成功



オブジェクトのPUT処理フロー

- オブジェクトノードがエラーを返してきても、半数以上のオブジェクトノードで正常終了すれば、クライアントにPUT成功を返却
- コンテナノードの更新に失敗すると、オブジェクトノード上のアップデータが代わって再試行する。コンテナノード上のレプリケータもコンテナ間の同期を取ろうとする。

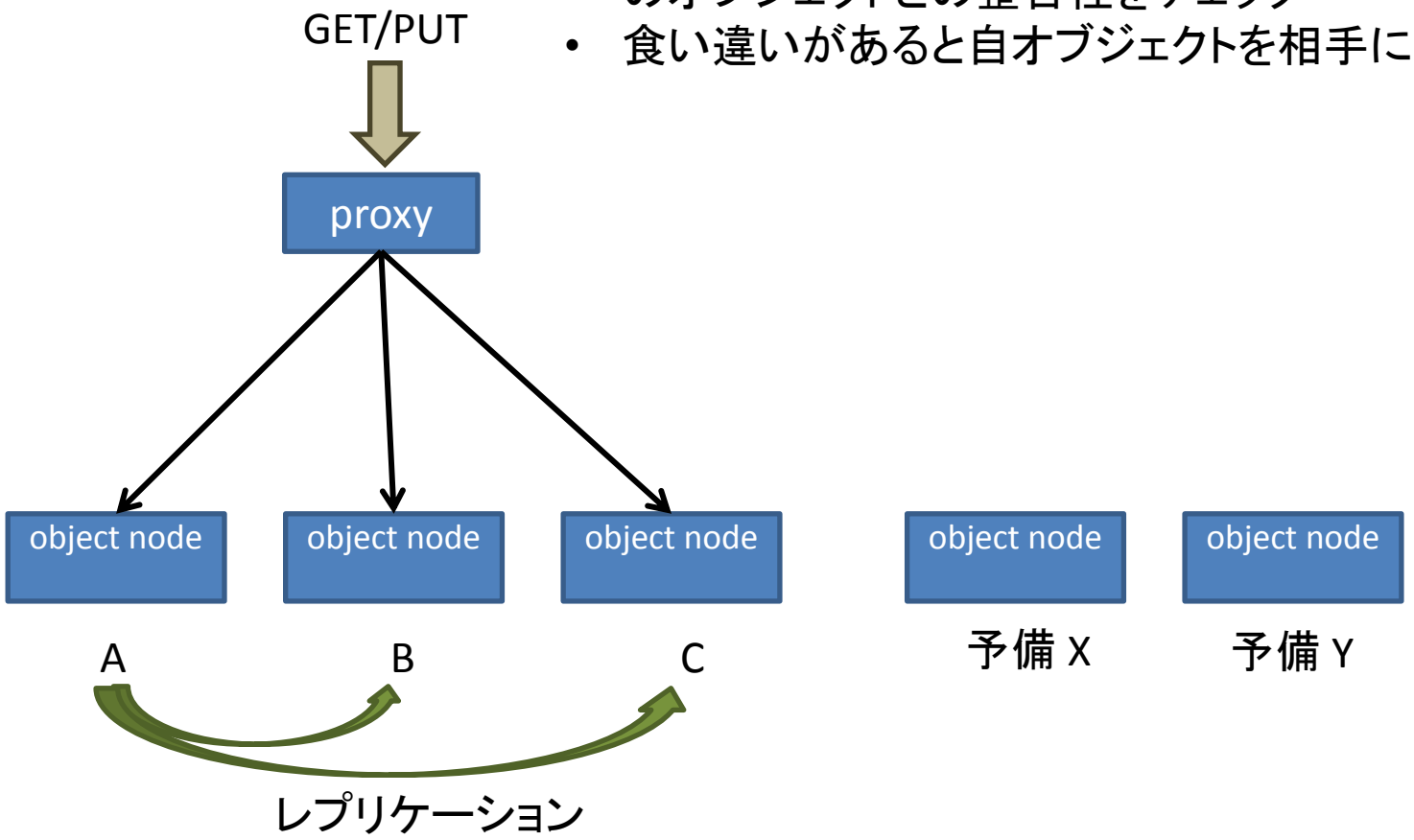


Eventual Consistency

- PUT処理が成功しても、その直後に更新後のデータが参照できるとは限らない
  - GET処理が選択したノードは、まだ更新前のデータしか持っていないかもしれない
- アップデータ、レプリケータにより、最終的には全てのノードが最新状態で同期する。
- スケーラビリティとのトレードオフで導入したモデル
- Swift 1.4.3(Diabloリリース)では、X-Newestヘッダを指定することにより、オブジェクトの最新データの取得を保証できる。ただし、すべてのオブジェクトノードにアクセスするため、オーバヘッドは大きい

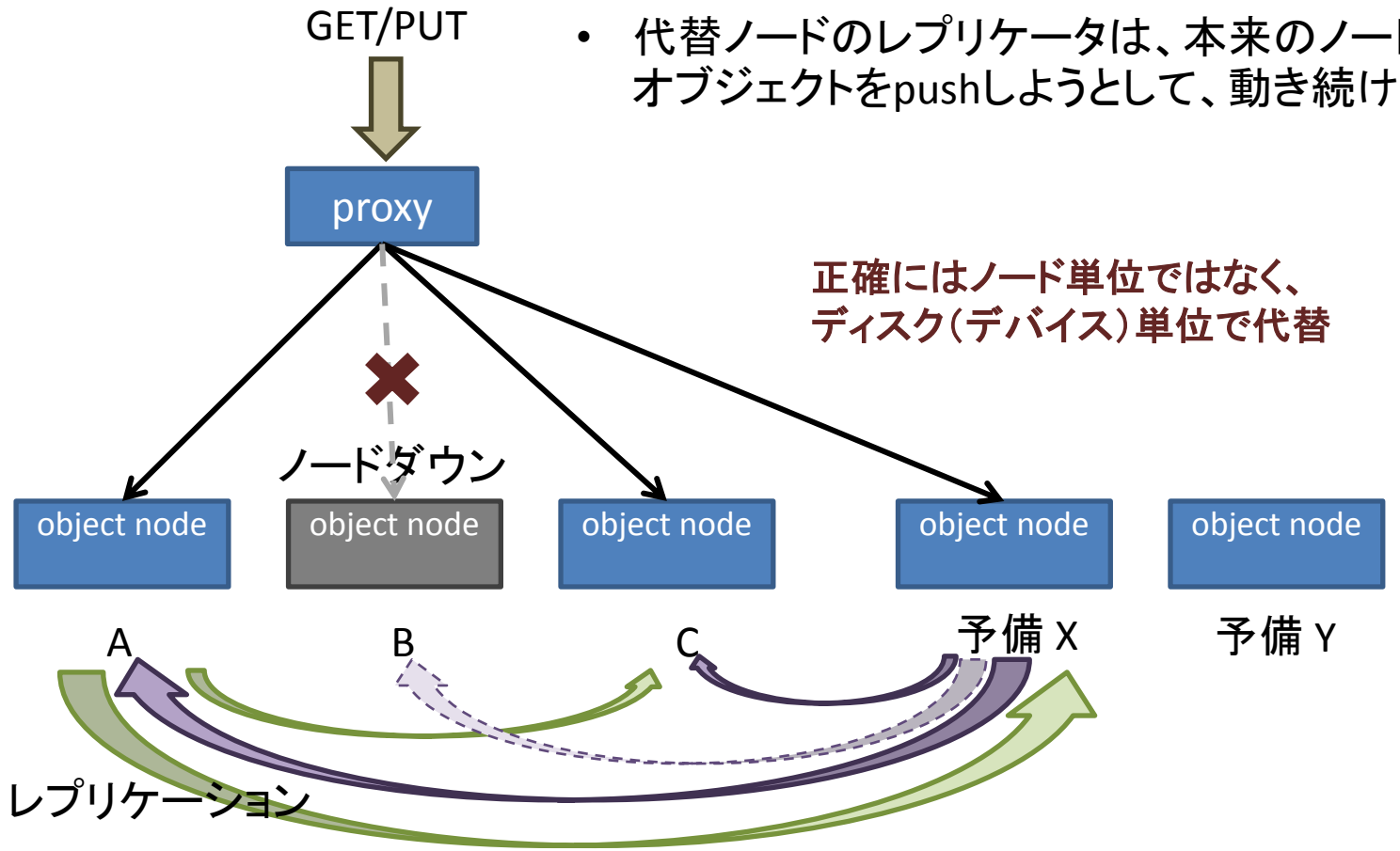
ハンドオフ：代替ノードの利用

- 正常時
  - 各ノードのレプリケータは、それぞれ他ノード上のオブジェクトとの整合性をチェック
  - 食い違いがあると自オブジェクトを相手にpush



ハンドオフ：代替ノードの利用

- ノード障害時
  - ノードがダウンしている場合、予備のノードを代替ノードとして扱う
  - 代替ノードのレプリケータは、本来のノード群にオブジェクトをpushしようとして、動き続ける。

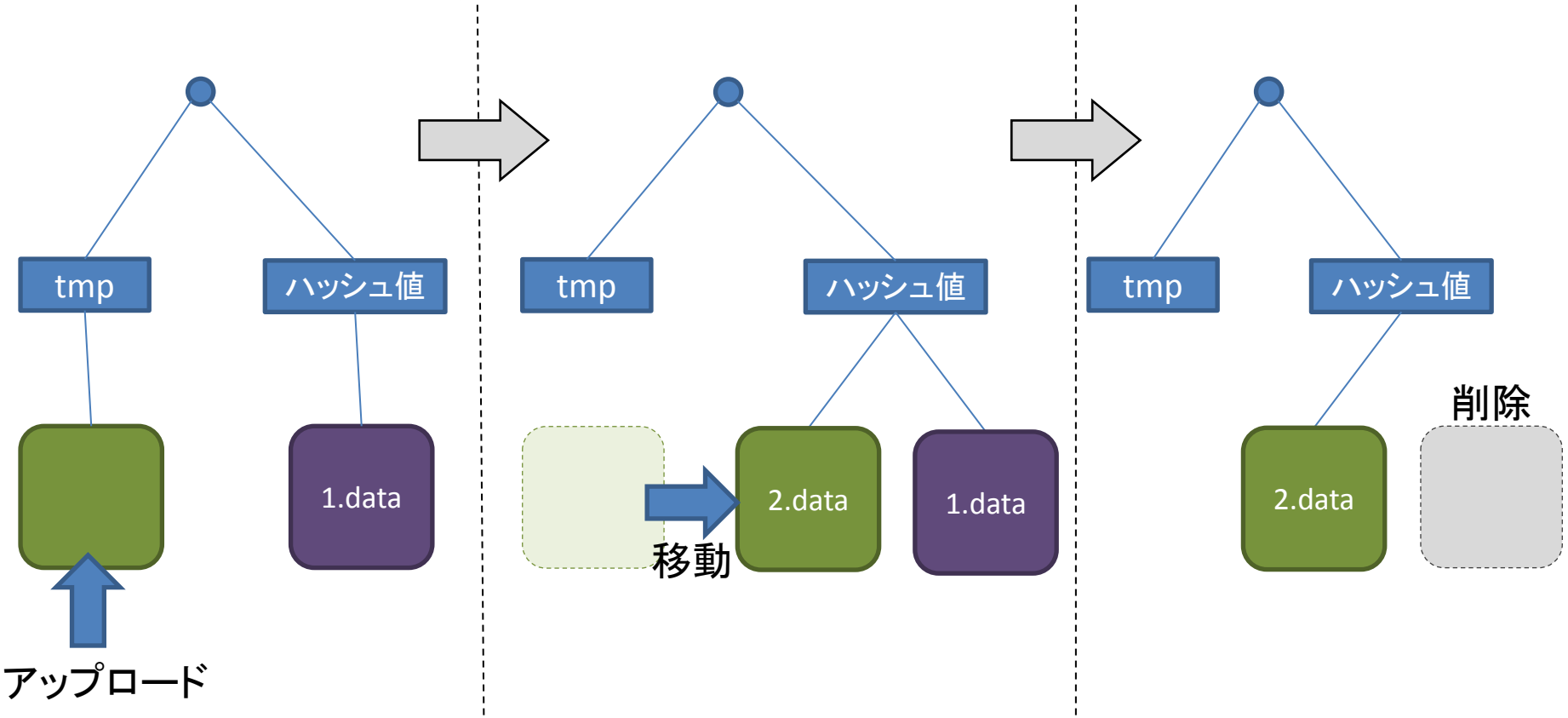


レプリケーションのための仕掛け

- ノードダウン中に、オブジェクトが上書きされたり、削除されることがある
- レプリケータは自ノードにあるデータを、本来あるべきノードにPUSHするのみ
  - データの新旧に関係なくで、単純にPUSHする
- タイムスタンプとtombstone
  - 古いデータを持つノードがシステムに組み込まれても矛盾しないための仕組み
  - タイムスタンプにより、新旧のデータを見分ける
  - tombstoneは、削除したオブジェクトが幽霊のように復活させないために必要

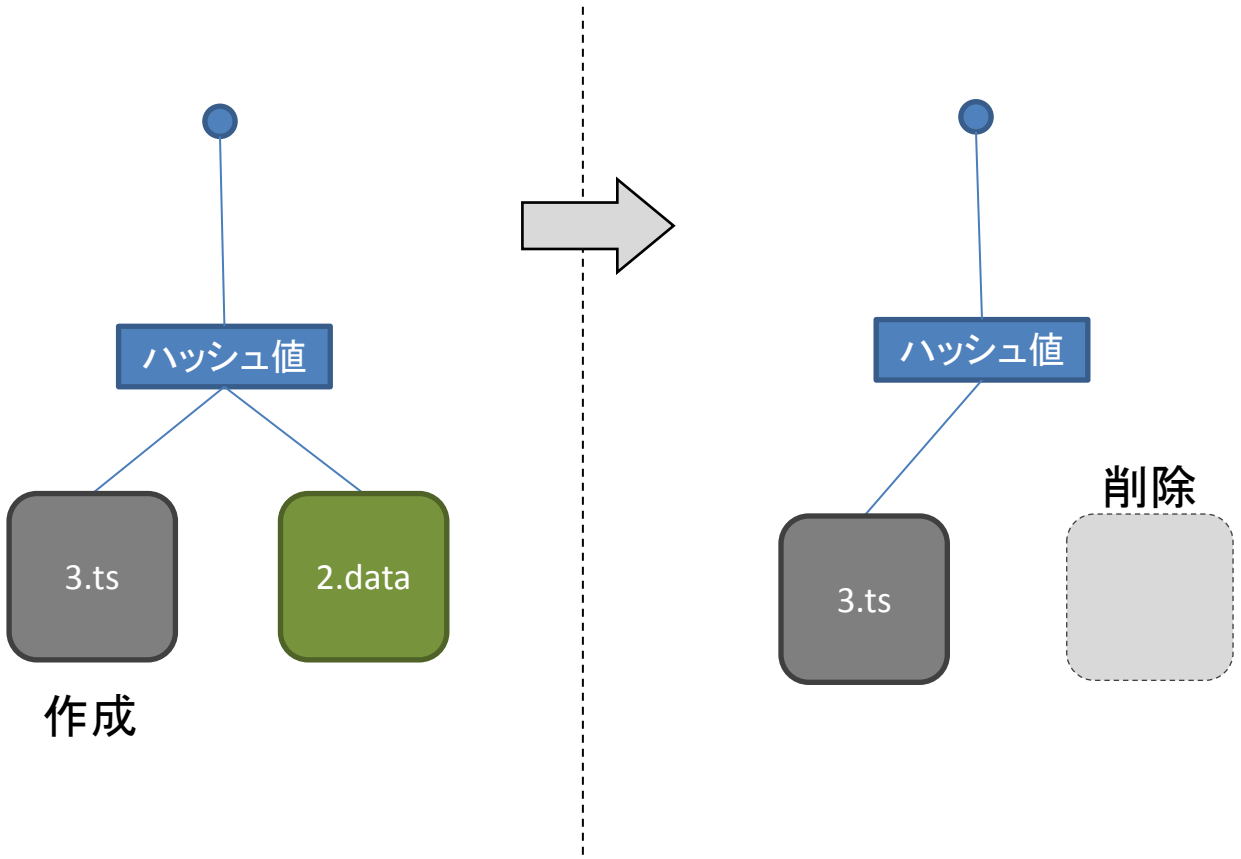
オブジェクトの上書き

- ハッシュ値を名前とするディレクトリの配下に、タイムスタンプ.data という名前でオブジェクトの実体を置く
- dataファイルが複数存在する場合は、もっとも新しいタイムスタンプを持つものが有効



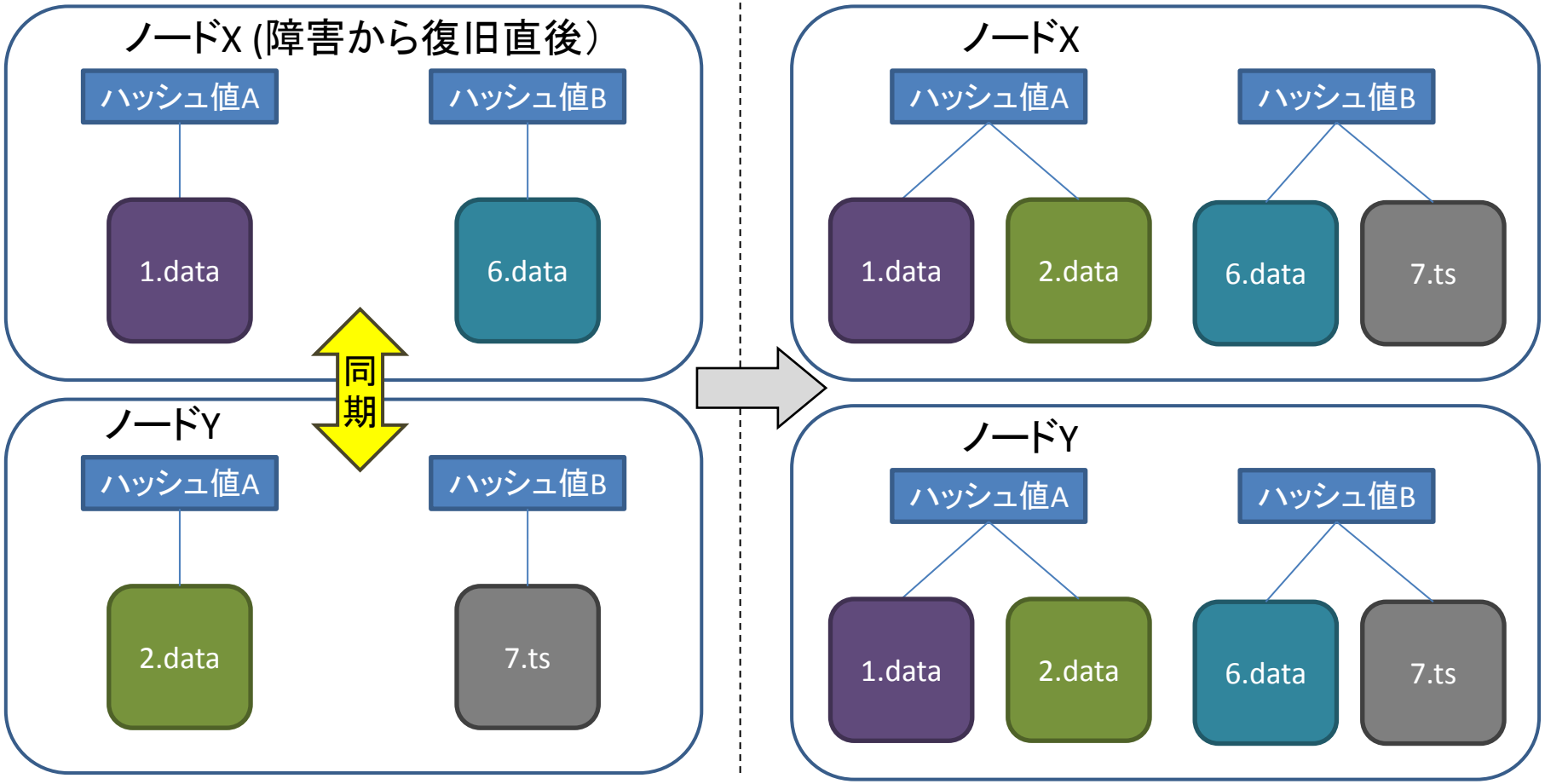
オブジェクトの削除

- オブジェクトを削除した場合、オブジェクトの代わりにtombstoneと呼ばれるファイルを置く。タイムスタンプ.ts という名前。



オブジェクトの同期

- 自分が持つデータ(ファイル)を、本来あるべきノードの上のものと比較し、違いがあれば送りつける(PUSH)。tombstoneを送ることにより、オブジェクトの削除状態も複製できる。
- オブジェクトデータとして新旧のデータがある場合、新しいタイムスタンプを持つものが有効。古いタイムスタンプのものは、レプリケータが遅延して片づける。



障害時の対応

- ディスク、ノードを切り離す
  - 障害ディスクのumount、ノードをネットワークから切り離す。
  - または復旧に時間が掛かるときは、障害ディスクのweightを0としたリングを全ノードに配布しても良い。
  - 自動的に代替ディスク、代替ノードに切り替わる
- ノード再起動
  - 再起動完了後、自動的にシステムに組み込まれる。レプリケータにより再同期が行われる
  - ディスクのweightを変更している場合は元の値に戻す。
- ノードの削除
  - 壊れたディスク・ノードを交換する時は、それらを削除したリングを作成し配布する

オブジェクトノードの起動

- オブジェクトノードをシステムに組み込む前に、オブジェクトのファイルサイズのチェックを行っておくとよい。オーディタ(auditor)を一度だけ実行する。

```
# swift-object-auditor /etc/swift/object-server.conf once -z 1000
```

障害監視

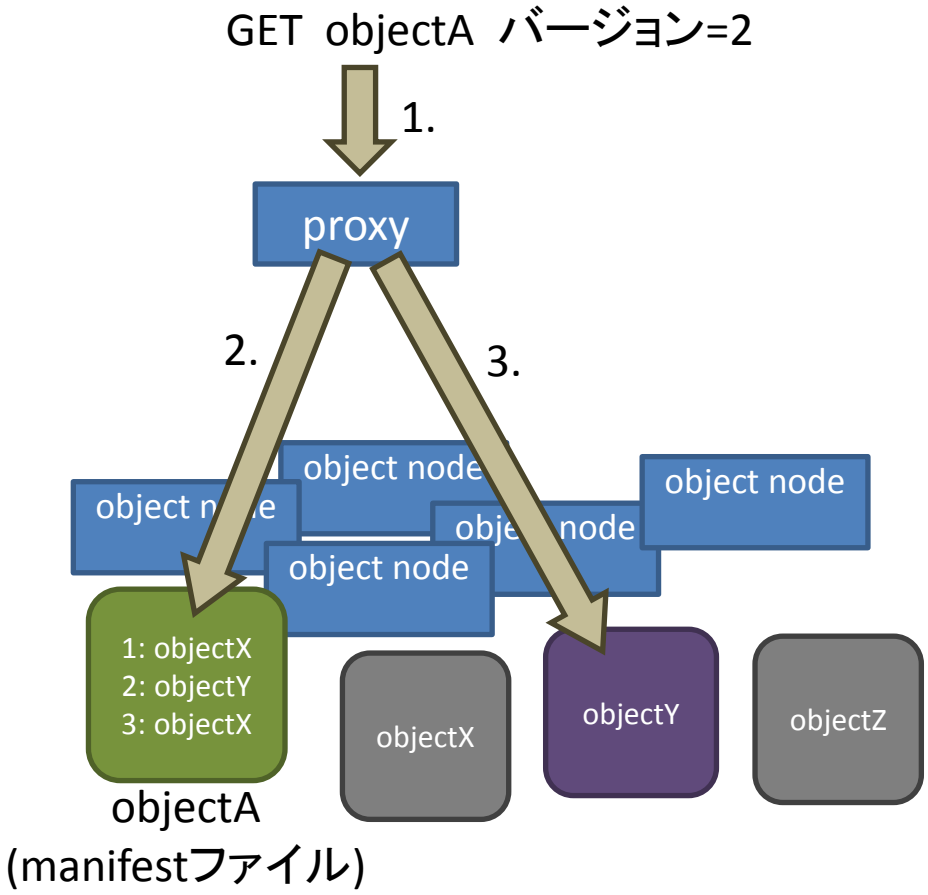
- ハードウェア監視
  - 監視ソフトウェア何でも構わない
  - syslogの監視。*swift-drive-audit*スクリプトも利用可能
- サービス監視
  - healthcheckミドルウェアをすべてのノードに組み込む。GET /healthcheck リクエストを送ると、200 OKレスポンスを返す。
- recon
  - 調子の悪いノード、負荷の高いノードが見つかる
  - Swift 1.4.3 (Diabloリリース)から利用可能

バックアップ

- リングと設定ファイルのバックアップ
  - 全ノードにコピー、Swiftのオブジェクトしてアップロード
- オブジェクトのバックアップ
  - 標準APIを利用して実装する（作りこむ）
    - 自由度は高い
  - 通常のバックアップソフトウェアの利用
    - 小規模構成の範囲であれば現実的
  - コンテナのsync機能
    - Swift 1.4.3 (Diabloリリース)から利用可能
  - リージョンによる遠隔レプリカ
    - Swiftプロジェクトで開発中。ゾーンの上にリージョンの概念を持ち込む。リージョンに跨ってレプリカ（複製）を持ち合う。
  - オブジェクトバージョニング機能
    - Swiftプロジェクトで開発中。Essexリリースには間に合わなかった模様。manifestファイルを利用して実現する。

オブジェクトのバージョニング（Folsom）

- オブジェクト名をもつmanifestファイル(索引)に、各バージョンのオブジェクトの実体が登録されている。
- プロキシノードがmanifestファイルを参照し、アクセスすべきオブジェクトの実体を決定する



スナップショット

- ある時刻一瞬のイメージを残す、取り出す
  - **Swiftの改造が必要（下記は実装案のひとつ）**
  - オブジェクトの上書き時に古いオブジェクトファイルを削除しない
  - オブジェクトの削除時（tombstone作成時)、古いオブジェクトファイルを削除しない
  - レプリケータの古いオブジェクトファイルの削除処理を抑制
  - オブジェクト削除時、オブジェクト名をコンテナDBのobjectテーブルから削除しない。レプリケータのdeleted属性エントリの削除処理を抑制
  - GETリクエストのパラメータに時刻を指定できるようにする
- スナップショットとバックアップを組み合わせることも可能

致命的障害からの復旧(1)

- リングを失ったら
  - リング更新時の、ビルダファイル（リング作成用の中間ファイル）のバックアップ忘れなど
  - 少し古い世代のリングを配布。リングが示すノードの一つには、目的のデータが存在する可能性が高い。
  - 新規のリングを作って配布。しばらくはGETでデータは参照できないが、レプリケータによりいずれあるべき場所にデータが移動する
- 設定ファイルを失ったら
  - MD5ハッシュ計算時の種だけは失ってならない。失うとオブジェクトの手動復旧以外に道は無くなる。システム構築時に、確実に一度バックアップしておくこと。
  - リセラープレフィックス（アカウントIDに付与される文字列ーデフォルト”AUTH_”）は、アクセスログなどから容易に見つかる

致命的障害からの復旧(2)
オブジェクトの手動復旧

- オブジェクト名が分かり、リングファイルがある場合
  - **オブジェクトの物理的位置を求める**
  - **2世代前までのリングでも構わない。下記コマンドが示す場所のいずれかには、オブジェクトが存在する**

```
# swift-get-nodes /etc/swift/object.ring.gz AUTH_orion-cabinet con object_1
Account         AUTH_orion-cabinet
Container       con
Object          object_1

Partition       54999
Hash            35b5ebcd21107f41ae41e230bb367a3c

Server:Port Device              172.10.0.4:6000 swdisk1
Server:Port Device              172.10.0.2:6000 swdisk1
Server:Port Device              172.10.0.1:6000 swdisk1
Server:Port Device              172.10.0.3:6000 swdisk1       [Handoff]

curl -I -XHEAD "http://172.10.0.4:6000/swdisk1/54999/AUTH_orion-cabinet/con/object_1"
curl -I -XHEAD "http://172.10.0.2:6000/swdisk1/54999/AUTH_orion-cabinet/con/object_1"
curl -I -XHEAD "http://172.10.0.1:6000/swdisk1/54999/AUTH_orion-cabinet/con/object_1"
curl -I -XHEAD "http://172.10.0.3:6000/swdisk1/54999/AUTH_orion-cabinet/con/object_1" # [Handoff]

ssh 172.10.0.4 "ls -lah /srv/node/swdisk1/objects/54999/a3c/35b5ebcd21107f41ae41e230bb367a3c/"
ssh 172.10.0.2 "ls -lah /srv/node/swdisk1/objects/54999/a3c/35b5ebcd21107f41ae41e230bb367a3c/"
ssh 172.10.0.1 "ls -lah /srv/node/swdisk1/objects/54999/a3c/35b5ebcd21107f41ae41e230bb367a3c/"
ssh 172.10.0.3 "ls -lah /srv/node/swdisk1/objects/54999/a3c/35b5ebcd21107f41ae41e230bb367a3c/" # [Handoff]
```

致命的障害からの復旧(2)
オブジェクトの手動復旧

- 見つかったオブジェクトファイルの拡張属性(exattr)域にメタデータが入っている
- リソース名（アカウント名、コンテナ名、オブジェクト名）もメタデータの一つとして保存されている

```
# swift-object-info /tmp/swdisk1/objects/54999/a3c/35b5ebcd21107f41ae41e230bb367a3c/1330619410.99149.data
Path: /AUTH_orion-cabinet/con/object_1
  Account: AUTH_orion-cabinet
  Container: con
  Object: object_1
  Object hash: 35b5ebcd21107f41ae41e230bb367a3c
Ring locations:
  172.10.0.2:6000 - /srv/node/swdisk1/objects/54999/a3c/35b5ebcd21107f41ae41e230bb367a3c /1330619410.99149.data
  172.10.0.1:6000 - /srv/node/swdisk1/objects/54999/a3c/35b5ebcd21107f41ae41e230bb367a3c /1330619410.99149.data
  172.10.0.3:6000 - /srv/node/swdisk1/objects/54999/a3c/35b5ebcd21107f41ae41e230bb367a3c /1330619410.99149.data
Content-Type: application/octet-stream
Timestamp: 2012-02-21 01:30:10.991490 (1327822691.35174)
ETag: 8975583a03073e60b34af5d2d3528661 (valid)
Content-Length: 120 (valid)
User Metadata: {'X-Object-Meta-Favorite': 'Apple'}
```

致命的障害からの復旧(2)
オブジェクトの手動復旧

- オブジェクト名が分からない場合は、コンテナDBをダンプする
  - まず、コンテナDBの位置を求める

```
# swift-get-nodes /etc/swift/container.ring.gz AUTH_orion-cabinet con
Account         AUTH_orion-cabinet
Container       foo
Object          None

Partition       245698
Hash            eff0834d2393fbb23f45d8b62cc47dfe

Server:Port Device              172.10.0.2:6001 swdisk1
Server:Port Device              172.10.0.3:6001 swdisk1
Server:Port Device              172.10.0.4:6001 swdisk1
Server:Port Device              172.10.0.1:6001 swdisk1       [Handoff]

curl -I -XHEAD "http://172.0.0.2:6001/swdisk1/245698/AUTH_orion-cabinet/con"
curl -I -XHEAD "http://172.0.0.3:6001/swdisk1/245698/AUTH_orion-cabinet/con"
curl -I -XHEAD "http://172.0.0.4:6001/swdisk1/245698/AUTH_orion-cabinet/con"
curl -I -XHEAD "http://172.0.0.1:6001/swdisk1/245698/AUTH_orion-cabinet/con" # [Handoff]

ssh 172.0.0.2 "ls -lah /srv/node/swdisk1/containers/245698/dfe/eff0834d2393fbb23f45d8b62cc47dfe/"
ssh 172.0.0.3 "ls -lah /srv/node/swdisk1/containers/245698/dfe/eff0834d2393fbb23f45d8b62cc47dfe/"
ssh 172.0.0.4 "ls -lah /srv/node/swdisk1/containers/245698/dfe/eff0834d2393fbb23f45d8b62cc47dfe/"
ssh 172.0.0.1 "ls -lah /srv/node/swdisk1/containers/245698/dfe/eff0834d2393fbb23f45d8b62cc47dfe/" # [Handoff]
```

致命的障害からの復旧(2)
オブジェクトの手動復旧

- コンテナDBのobjectテーブルをダンプし、オブジェクト一覧を入手

（次ページ）

```
# sqlite3 -line /srv/node/swdisk1/containers/10504/7a3/0a4224369c35c3aadf07d924869167a3/0a4224369c35c3aadf07d924869167a3.db
'.dump object'
PRAGMA foreign_keys=OFF;
BEGIN TRANSACTION;
CREATE TABLE object (
            ROWID INTEGER PRIMARY KEY AUTOINCREMENT,
            name TEXT,
            created_at TEXT,
            size INTEGER,
            content_type TEXT,
            etag TEXT,
            deleted INTEGER DEFAULT 0
          );
INSERT INTO "object" VALUES(2,'object_1','1327822691.35174',120,'application/octet-stream','0e07221c3d284281c348eaf83e70ddce',0);
INSERT INTO "object" VALUES(4,'object_2','7822132691.18574',3456,'application/octet-stream','4281c348eaf83e70ddce0e07221c3d28',0);
INSERT INTO "object" VALUES(5,'object_3','7812691232.17392',7890,'application/octet-stream','b039efe731ad111bc1b0ef221c3849d0',0);
CREATE INDEX ix_object_deleted_name ON object (deleted, name);
CREATE TRIGGER object_insert AFTER INSERT ON object
          BEGIN
            UPDATE container_stat
            SET object_count = object_count + (1 - new.deleted),
                bytes_used = bytes_used + new.size,
                hash = chexor(hash, new.name, new.created_at);
          END;
CREATE TRIGGER object_update BEFORE UPDATE ON object
          BEGIN
            SELECT RAISE(FAIL, 'UPDATE not allowed; DELETE and INSERT');
          END;
CREATE TRIGGER object_delete AFTER DELETE ON object
          BEGIN
            UPDATE container_stat
            SET object_count = object_count - (1 - old.deleted),
                bytes_used = bytes_used - old.size,
                hash = chexor(hash, old.name, old.created_at);
          END;
COMMIT;
```

致命的障害からの復旧(2)
オブジェクトの手動復旧

- コンテナDBのcontainer_statテーブルをダンプし、コンテナに設定したACL、メタデータを入手

```
# sqlite3 -line /srv/node/swdisk1/containers/10504/7a3/0a4224369c35c3aadf07d924869167a3/0a4224369c35c3aadf07d924869167a3.db '.dump container_stat'
INSERT INTO "container_stat" VALUES('AUTH_orion-cabinet','con','1327822690.14660','1327822690.10314','0',1,17,'1327822690.10314','0',1,17,'5d0cc939171d8ac057872bc9d18ab858','8d7f6bf4-ab92-41fc-ae20-3eddd0774a2f','','0','{"X-Container-Read": [".r:www.valinux.co.jp,pleiades", "1329889234.29659"],"X-Container-Write": ["pleiades:larry", "1329889234.29659"],"X-Container-Meta-Type": ["binary", "1329889234.29659"]}');
COMMIT;
```

致命的障害からの復旧(3)
リングを失った場合の手動復旧

- リングを失っている場合でも、オブジェクトノードのオブジェクトファイルを全てダンプすることにより、データを復元できる
  - オブジェクトファイルには、データとメタデータが含まれている
  - オブジェクトのメタデータとして、アカウント名、コンテナ名、オブジェクト名が含まれている
- コンテナDBも全てダンプすれば、ACL情報も復元できる

高負荷対策

- 適切なプロセス数、スレッド数
- 突発的負荷対策
  - システム全体で負荷を絞る
  - ノード単位で負荷を絞る
- 一時的回避
- ハードウェア増強

適切なプロセス数、スレッド数

- pythonインタプリタの制約により、各デーモンは1つのCPUコアしか利用することができない。デーモンをCPUコア数以上起動する必要がある。
- 各デーモン内で動作させるスレッド数も明示的に指定する

/etc/swift/object-server.conf
サーバのプロセス数を8として指定、レプリケータのスレッド数は2

```
[DEFAULT]
workers = 8

[object-replicator]
concurrency = 2

[object-updater]
concurrency = 1

[object-auditor]
concurrency = 1
```

突発的負荷対策 – システム全体

- ratelimitミドルウェアをプロキシに組み込む
- 設定ファイルproxy-server.conf で挙動を指定する

```
[pipeline:main]
pipeline = healthcheck cache ratelimit swauth proxy-server

[filter:ratelimit]
use = egg:swift#ratellimit
account_ratelimit = 100
container_ratelimit_100 = 100
container_ratelimit_200 = 50
container_ratelimit_500 = 20
```

- account_ratelimitで、アカウントおよびコンテナに対する秒間リクエスト数の上限を指定
- container_ratelimitで、コンテナに対するリクエストとオブジェクトのPUT/DELETEリクエスト数の合計の上限を、コンテナのサイズ（登録オブジェクト数）毎に指定できる

突発的負荷対策 – ノード単体

- オブジェクト更新速度の抑制
  - /etc/swift/object-server.conf　object-serverセクションのslow項目

```
[object-server]
slow = 2
```

- 2秒間に1つのPUTまたはDELETEリクエストを受け付ける

突発的負荷対策 – ノード単体

- アップデータ（updater）のスローダウン
  - /etc/swift/object-server.conf、container-server.conf にて設定

```
[object-updater]
interval = 600
slowdown = 0.1
```

- interval : アップデータの実行周期（秒）
- slowdown : 上位ノードへの更新情報通知それぞれの間に空ける時間（秒）

一時的回避

- 通信のタイムアウト値を伸ばす
  - 応答性確保のために、標準設定では短め
  - 各種処理にタイムアウト値が設定されている

◆ conn_timeout
- デーモン間のコネクション確立処理のタイムアウト値。0.3～0.5秒程度に設定されている処理が多く、負荷が高くなると容易にタイムアウトする。

◆ node_timeout
- デーモン間の通信において、要求を出してからここで指定した値以上の時間応答がないとタイムアウトし、コネクションを切ってしまう。

一時的回避

- レプリケータ（Replicator)の動作をマイルドにする
  - 動作周期を伸ばす
- オーディタ（Auditor)の動作をマイルドにする
  - 動作周期を伸ばす
  - オブジェクトノードのオーディタの場合、オブジェクト破損チェック方法を軽いものにする
    - オブジェクトファイルのMD5ハッシュ値の確認でなく、オブジェクトファイル長の確認のみにする
- 負荷が集中してるディスク（デバイス）のweightを小さくする

ハードウェア増強

- アカウント・コンテナ・オブジェクトノード追加
  - 新しいノードのディスクを登録したリングを作成し、全ノードに配布する
  - 既存のパーティション割り当てからの変更量を抑えるように再割り当てが行われる
- プロキシノードの追加
  - 既存のリングを追加するノードにコピー
  - ハードウェアスペックの低いノードの切り離し
- ハードウェアスペックの低いノードの切り離し
  - 切り離すノードのディスクのweightを0としたリングを配布
  - ディスク上のパーティションが、別のノードに全て移動した後で、切り離すノードを削除したリングを再度配布

リングの再構成について

- 各データの複製は、必ず異なるゾーンに置く
- 一定時間内に、パーティションを2回移動させない
  - 削除されたデバイス、ノード上のパーティションは除く
- 移動しなければならないパーティション数をなるべく少なくする
- 各ゾーン、各ノードに配置されるパーティション数がなるべく均等になるようにする
- ノード上のリングの更新は、デーモンからのポーリングにより検出される
- 新旧のリングの情報が混在して動作する期間がある
  - 最終的には新しいリングの情報に合わせて、レプリケータが正しい位置にパーティションを配置する

既存システムとの認証統合

- Swiftは認証モジュールを交換することが可能
  - swauth: Swift 1.3.0(Cactusリリース)までは、Swiftに同梱。Swift 1.4.3からは、別プロジェクトに分離
  - tempauth: 移植用の認証モジュール。Swift 1.4.3(Diabloリリース）から、Swift同梱の標準モジュール。
  - OpenStack Identity (Keystone): Nova（Openstack Compute)と共通の認証モジュール。

認証モジュールの機能

Swiftの認証モジュールは二回呼び出される

1. 認証
   - Swiftシステムの登録ユーザであることを確認
2. アクセス権チェック
   - ACL情報や、ユーザ権限をもとに、対象とするリソースに指定された操作を行う権利があるか否かを判定する

認証モジュールvaauthを作る

- tempauthをひな形にして、vaauthを作る。認証機能のコードを修正する。

```
# cp swift/common/middleware/tempauth.py  vaauth.py
```

- コード修正個所は3箇所
  - クラス定義
  - サービス用のURL設定
  - ユーザの確認とトークンの発行
- アカウントの作成
- パッケージ化とSwiftへの組み込み
- アカウント登録、ユーザ登録機能はない。連携先の認証システム側で行う。

クラス定義とオブジェクト生成

- クラスTempAuthをVAAuthに書き換える

```
class VAAuth(object):
        .....
        .....
def auth_filter(app):
    return VAAuth(app, conf)
```

サービスURL設定

- **tempauthは、ロードバランサ用の設定ができないので拡張する**

/etc/swift/proxy-server.confの[filter:swauth]セクションにおいて、サービス用のURLを以下のように設定できるようにする。

```
[filter:swauth]
service_url = https://objstore.example.jp/
```

__init__メソッドで、設定ファイル/etc/swift/proxy-confからURLを読み込む。

```
def __init__(self, app, conf):
        .......
  url = 'http://127.0.0.1/'
   url = conf.get('service_url', url)
   url = url.rstrip('/ ')
   url += '/v1/' + self.reseller_prefix
   self.url_prefix = url
```

ユーザ確認

- ユーザ認証を行うhandle_get_tokenメソッドを編集
  - 以下のコードを、外部認証サーバに問い合わせて、認証するように変更する

```
def handle_get_token(self, req):
        ……
  # Authenticate user
  account_user = account + ':' + user
  if account_user not in self.users:
     return HTTPUnauthorized(request=req)
  if self.users[account_user]['key'] != key:
     return HTTPUnauthorized(request=req)
```

ユーザ権限設定、ストレージURL生成、トークンの発行

```
def handle_get_token(self, req):
        ……
# 設定ファイルに書いてあるURLとアカウントを組み合わせ、ストレージURLを生成する
url = self.url_prefix + account
if not token:
    # Generate new token
    token = '%stk%s' % (self.reseller_prefix, uuid4().hex)
    expires = time() + self.token_life
    groups = [account, account_user]
    if ユーザにアカウント管理者権限がある:
        groups.append('.admin')
    if ユーザにサービス提供者権限がある:
        groups.append('.reseller_admin')
    if '.admin' in groups:
        groups.remove('.admin')
        # URL中からアカウントID部を取り出す
        account_id = url.rsplit('/', 1)[-1]
        groups.append(account_id)
    groups = ','.join(groups)
    # Save token
    memcache_token_key = '%s/token/%s' % (self.reseller_prefix, token)
    memcache_client.set(memcache_token_key, (expires, groups), timeout=float(expires - time()))
    # Record the token with the user info for future use.
    memcache_user_key =  '%s/user/%s' % (self.reseller_prefix, account_user)
    memcache_client.set(memcache_user_key, token, timeout=float(expires - time()))
self.logger.debug('ooauth.handle_get_token groups=%s' % groups)
# 生成したストレージURLを、クライアントのレスポンスに含める
return Response(request=req, headers={'x-auth-token': token, 'x-storage-token': token, 'x-storage-url': url})
```

アカウントの作成

- アカウントノード上にアカウントDBを作る必要がある
  - Swift 1.4.3 (Diabloリリース)以降は、初回のアクセス時に自動生成する機能がある
  - Swift 1.3.0 (Cactusリリース）では、リセラー管理者が、アカウント名を指定したPUTリクエストを発行することにより作成する

認証モジュールの パッケージ化とインストール

- vaauthをeggモジュールという形式でパッケージ化し、インストールする
  - モジュール作成手順については、setuptoolsのマニュアルを参照のこと
  - ここでは、パッケージ名とモジュール名のどちらもvaauthとして作成するものとする
- 設定ファイル/etc/swift/proxy-server.conf
  - swauthの代わりにvaauthを認証モジュールとして組み込む

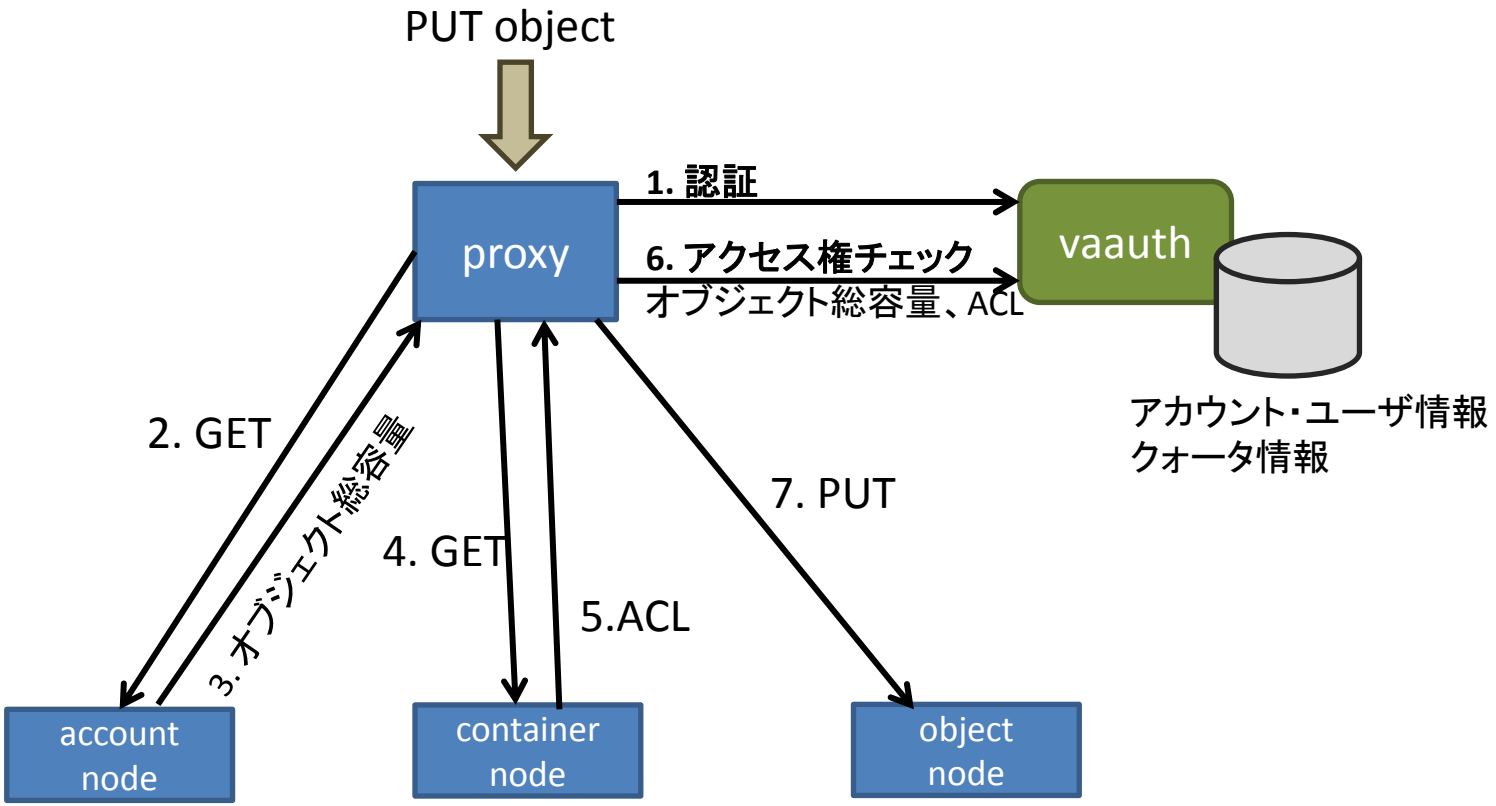
```
[pipeline:main]
pipeline = catch_errors cache vaauth proxy-server

[filter:vaauth]
use = egg:vaauth#vaauth
```

クォータ機能実装（案１）
アカウント単位のクォータ制御

- オブジェクトアクセス時に呼び出す認証モジュール(vaauthなど)のauthorizeメソッドでは、ACL情報や特権ユーザ情報に基づき、アクセスの許可・不許可を判断している
  - クォータ情報は、アカウントと結び付けてSwiftの外で管理
  - プロキシサーバは、オブジェクトのPUTリクエスト発生時、アカウントノードで管理されているオブジェクト総容量を読み出し、認証モジュールに渡す。（ACL情報の渡し方が参考になる）
  - 認証モジュールは、オブジェクトのPUTリクエストの場合、「オブジェクト総容量 + オブジェクトサイズ(リクエストのContent-Length) <= クォータ」であることもチェックする
  - 特権ユーザは、クォータチェックの例外とすることも可能

クォータ機能実装（案１）
アカウント単位のクォータ制御



vaauth修正個所

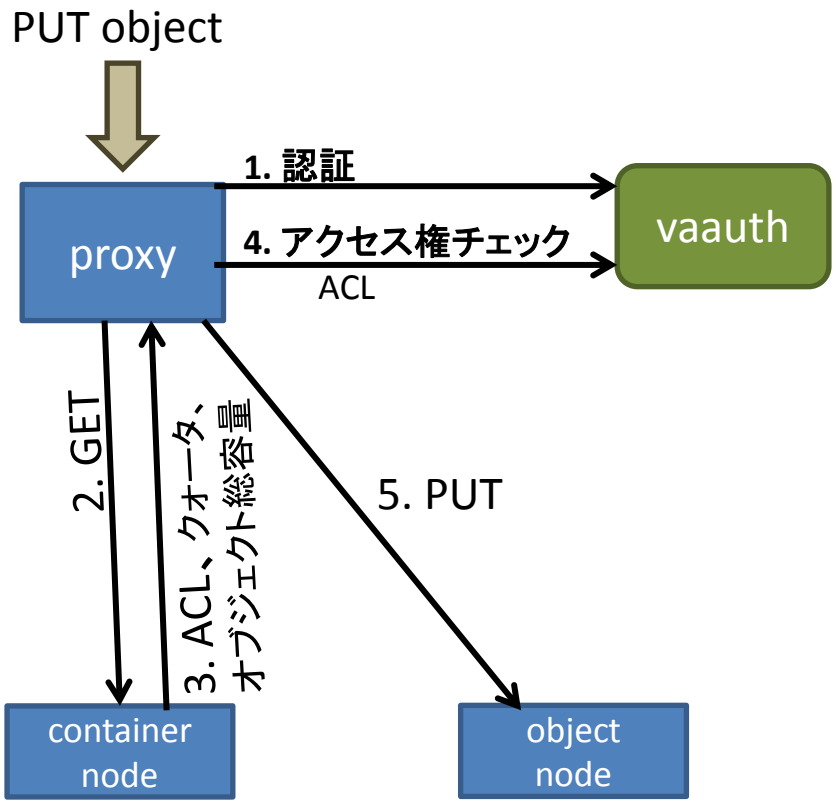
- アクセス権チェックを行うauthorizeメソッドで、クォータのチェックも行うようにする

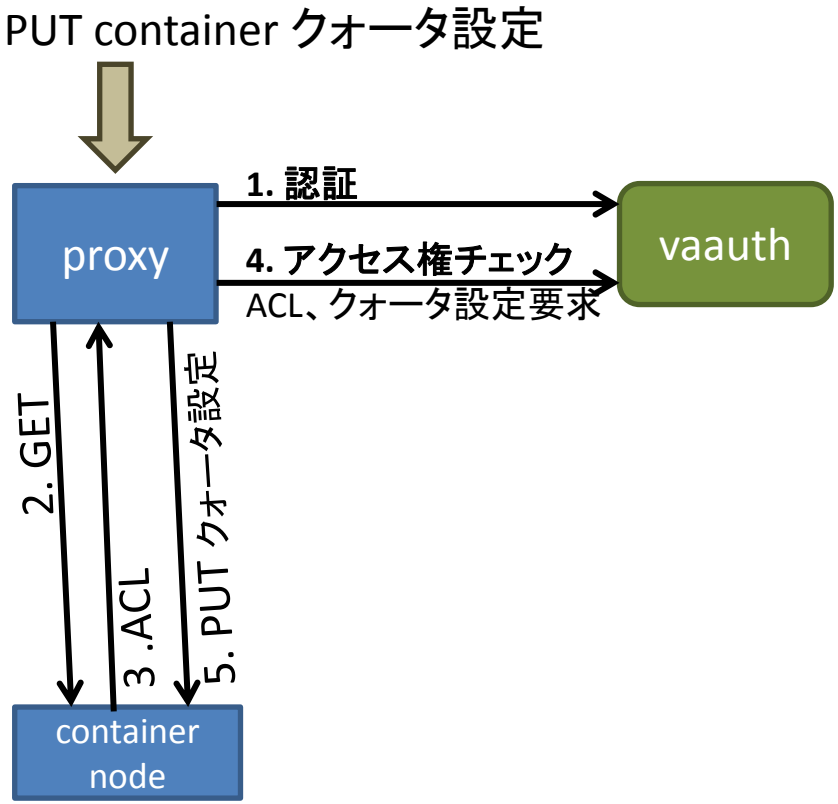
```
def authorize(self, req):
    """
    Returns None if the request is authorized to continue or a standard   WSGI response callable if not.
    """
                :
    user_groups = (req.remote_user or '').split(',')
    if '.reseller_admin' in user_groups and account != self.reseller_prefix and account[len(self.reseller_prefix)] != '.':
        return None
    if account in user_groups and (req.method not in ('DELETE', 'PUT') or container):
        # If the user is admin for the account and is not trying to do an account DELETE or PUT...
        return None
    このあたりで、クォータのチェックをすると楽
    referrers, groups = parse_acl(getattr(req, 'acl', None))
    if referrer_allowed(req.referer, referrers):
        if obj or '.rlistings' in groups:
            return None
        return self.denied_response(req)
    if not req.remote_user:
        return self.denied_response(req)
    for user_group in user_groups:
        if user_group in groups:
            return None
```

クォータ機能実装（案2）
コンテナ単位のクォータ制御

- コンテナDBのメタデータ格納域にクォータ情報も登録する
  - プロキシサーバは、オブジェクトのPUTリクエスト時、コンテナDBのクォータ情報・コンテナ配下のオブジェクトの総容量、およびPUT対象のオブジェクトのサイズから、書き込み許可・禁止を決める
- コンテナDBのメタデータにクォータ情報を登録できるのは、リセラー管理者（サービス提供者）のみとする
  - コンテナへのPUTリクエストに、クォータ設定ヘッダを新設
  - 認証モジュールvaauthは、リセラー管理者以外がこのヘッダを利用していたら、エラーとする

クォータ機能実装（案）
コンテナ単位のクォータ制御





クォータ機能の制限

- クォータを少し超えてオブジェクトが登録されることがある
  - 複数のリクエストに対する処理が同時に並列動作
  - オブジェクトアップロード前に予約できない（してはならない）
- クォータ情報を格納した分、自由に登録できるメタデータの最大量が減る

その他

- Ubuntuで動作させるのがお勧め
  - RHEL/CentOSでの運用は苦労する
    - RedHatが対応を始めたので、来年あたりには使えるかも
  - Swiftは、最新パッケージの最新機能を使う
- ソフトウェアのバージョンアップ
  - 最近のSwiftは、少し古いバージョンのSwiftのデータ構造（データベース）に対しても動作するように考慮されている
    - データベースのテーブル形式のバージョンを意識して動作。テーブルへのカラム動的追加なども行う
  - ゾーン単位で動作結果を見ながらゆっくりアップデートすると安全
  - proxyは一番最後に更新する

その他

- 定期的なノード再起動
  - ネットワーク障害などにより、解放漏れ資源が残ることがある。定期的なリブートを勧める
- データベースはSQLite3のまま使う
  - 別のデータベースへの交換はえらく大変
  - SQLite3固有の高機能を多用している
- ログ出力の理解には、Swift実装知識が必要
- Swiftに関する情報があるように見えて、ドキュメント化されていない情報が多い

その他

- 特定の契約者、特定の用途用のプロキシノードを立てることが可能（例: バックアップ専用)
  - プロキシ毎に、ratelimitミドルウェアに異なる値を設定可能
- ストレージURLにIPアドレスは入れない
- ノード間の時刻同期は正確に。特にプロキシ間で時刻が揃っていることは必須。

コミュニティへの参加

- OpenStack Design Summitとは別にSwiftだけのカンファレンスも開催されている
- 機能追加を行うには、ブループリントを提案し、上記会議で承認され、開発を行うというプロセスが必要
- 提供コードはRackSpace社に譲渡するという契約が必要（会社レベルと個人レベルの両方が必要）
- SwiftはOpenStackとしてのリリースの間に、細かくマイナーバージョンアップリリースを繰り返す
- 運営主体をRackspace社から、コミュニティ参加企業のボードメンバ制へ移行中
- Swiftのコードライセンスは Apache v2であるため、改造したものの使い方によらず、コード公開義務は生じない