Swift(1) - OpenStack Object Storage 全体像とシステム構築

OSS基盤技術センター
OSS技術第二課

目次

- Swift全体像
- ReST API
- Amazon S3 と　Swift
- インストールと初期設定
- 障害対応、高負荷対策
- 運用のTips
- Swiftプロジェクト開発状況

Swift全体像

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
  - レプリカを異なるゾーン上に配置
  - リング : データの配置を決めるテーブル
  - データ破損の検出と自動修復
- Eventual Consistencyモデル
  - データ更新後、直後の参照では最新データが見えるとは限らない

Swiftの構成ノード

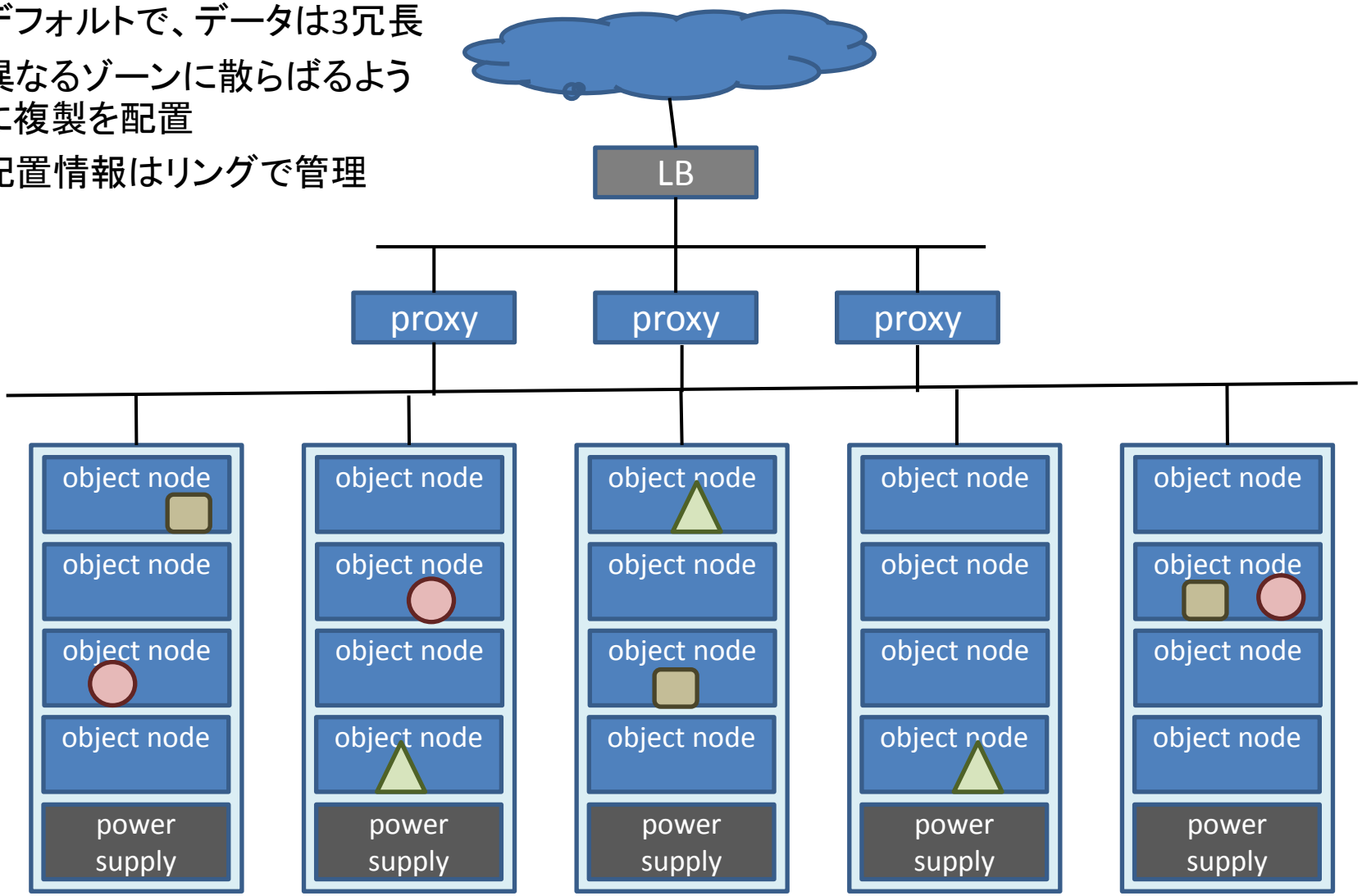
- プロキシノード
  - ReST APIを実現。クライアントからの要求を、データを管理する適切なノードに転送する
  - 認証モジュールはここに組み込まれている
- アカウントノード
  - アカウント情報をSQLite3データベースを用いて管理する。コンテナ名の一覧、アカウント全体の統計情報などを管理
- コンテナノード
  - コンテナ情報をSQLite3データベースを用いて管理する。オブジェクト名の一覧、ACL情報などを管理
- オブジェクトノード
  - オブジェクトの実体を保存する
- 各ノードが、同一ハードウェアに同居することも可能

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。

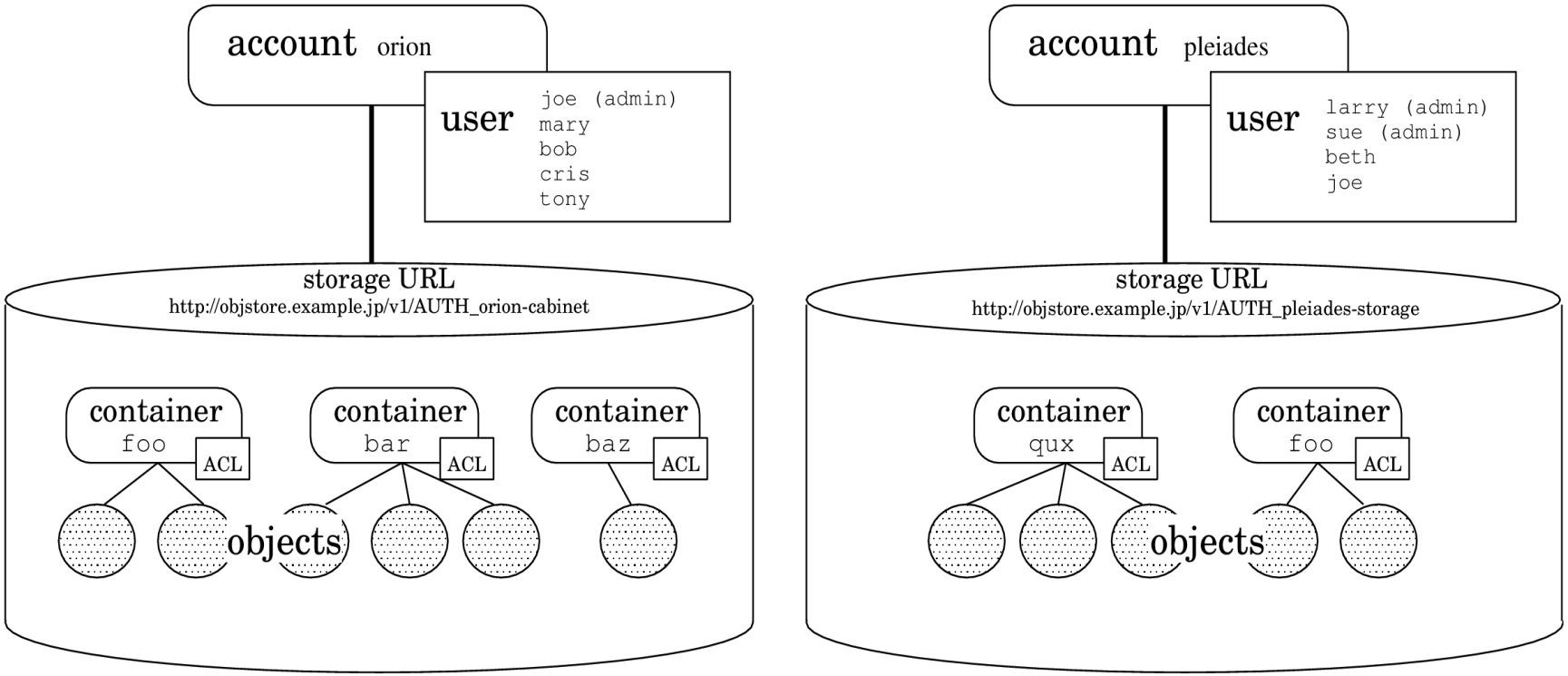
ゾーンとレプリカ(複製)

- 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



論理構造

- アカウントに対してユニークなストレージURLを割り当て
- アカウント単位、ユーザとコンテナを管理
- コンテナ階層は一段。オブジェクトは、コンテナ配下にのみ置ける
- アクセス権設定は、コンテナ単位



アカウントとユーザと権限

- Swauthのユーザ管理モデル
- ３種類の特権ユーザ
  - システム管理者(super admin)
  - サービス提供者(reseller admin)
  - アカウント管理者(admin)
- アクセス制御はコンテナ単位
  - 他アカウントのユーザにもアクセスを許可できる

API

Restインターフェイス

- GET/PUT/POST/DELETEリクエストによる操作
- 認証操作
  - アカウント名、ユーザ名、パスワードをGETリクエストで送り、トークンを取得する
- リソース制御操作
  - 認証操作で得られたトークンを、HTTPヘッダに記述したうえで、対象リソースに対してGET/PUTなどのリクエストを送る

認証（swauth)

- 認証用URLに対して、GETリクエストを発行する
  - アカウント、ユーザ、パスワード情報を渡す
  - 成功すると、トークンとストレージURLが返却される
- 認証時に得られたトークンとストレージURLは、オブジェクトやコンテナなどへのGET/PUTリクエスト発行時に利用

認証

- GETリクエスト

```
GET /auth/v1.0 HTTP/1.1
User-Agent: curl/7.21.0  libcurl/7.21.0     OpenSSL/0.9.8o zlib/1.2.3.4 libidn/1.18
Host: objstore.example.jp
Accept: */*
X-Storage-User: orion:joe
X-Storage-Pass: testpassword
```

- レスポンス

```
HTTP/1.1 200 OK
X-Storage-Url: https://objstore.example.jp/v1/AUTH_orion-cabinet
X-Storage-Token: AUTH_tkfdd103dc022f49308bf9c96568fafbb2
X-Auth-Token: AUTH_tkfdd103dc022f49308bf9c96568fafbb2
Content-Length: 96
Date: Fri, 15 Jul 2011 09:08:14 GMT

{"storage": {"default": "local",
"local": "https://objstore.example.jp/v1/AUTH_orion-cabinet"}}
```

認証

- 続くリソースアクセス用リクエスト時
  - リソースのパスにストレージURLを含める
  - HTTPヘッダにトークンを指定する

```
GET /v1/AUTH_orion-cabinet/foo/baa HTTP/1.1
User-Agent: curl/7.21.0  libcurl/7.21.0  OpenSSL/0.9.8ozlib/1.2.3.4 libidn/1.18
Host: objstore.example.jp
Accept: */*
X-Auth-Token: AUTH_tkfdd103dc022f49308bf9c96568fafbb2
```

アカウント

- GETリクエストを送ると、アカウント配下のコンテナ一覧を取得できる
  - formatパラメータを指定することにより、コンテナ数、オブジェクト数、使用ディスク容量の情報も取得できる
  - limit, markerパラメータにより、指定した範囲のコンテナ名のみを取得可能

```
GET /v1/AUTH_orion-cabinet?format=json HTTP/1.1
User-Agent: curl/7.21.0 libcurl/7.21.0  OpenSSL/0.9.8o zlib/1.2.3.4 libidn/1.18
Host: objstore.example.jp
Accept: */*
X-Auth-Token: AUTH_tkcbd0b108401245a3acf1b550aa4862bb
```

```
HTTP/1.1 200 OK
X-Account-Object-Count: 2
X-Account-Bytes-Used: 715653825
X-Account-Container-Count: 4
Content-Length: 154
Content-Type: application/json; charset=utf8
Date: Thu, 07 Jul 2011 02:35:56 GMT

[{"name":"bin","count":1,"bytes":8897},
 {"name":"doc","count":0,"bytes":0},
 {"name":"image","count":1,"bytes":715644928},
 {"name":"src","count":0,"bytes":0}]
```

コンテナ

- GETリクエストにより、コンテナ配下のオブジェクト一覧取得
  - path, delimiterパラメータを利用し、階層構造を持ったディレクトリ構造のように操作することも可能
  - formatパラメータにより、各オブジェクトの属性情報（ハッシュ値、最終更新時刻、オブジェクトサイズ）も取得可能
  - limit, markerパラメータにより、指定した範囲のオブジェクト名のみを取得可能
- PUTリクエストで、コンテナの作成
  - メタデータ、ACLも設定可能

```
GET /v1/AUTH_orion-cabinet/src?prefix=swift-1.3.0/swift/&delimiter=/ HTTP/1.1
User-Agent: curl/7.19.7  libcurl/7.19.7  OpenSSL/0.9.8k zlib/1.2.3.3 libidn/1.15
Host: objstore.example.jp
Accept: */*
X-Auth-Token: AUTH_tk734c83ab579a4f938a1b82e5841d8d22
```

```
HTTP/1.1 200 OK
X-Container-Object-Count: 217
X-Container-Bytes-Used: 2387310
Content-Length: 174
Content-Type: text/plain; charset=utf8
Date: Wed, 13 Jul 2011 08:27:12 GMT

swift-1.3.0/swift/account/
swift-1.3.0/swift/common/
swift-1.3.0/swift/container/
swift-1.3.0/swift/__init__.py
swift-1.3.0/swift/obj/
swift-1.3.0/swift/proxy/
swift-1.3.0/swift/stats/
```

オブジェクト

- PUTでオブジェクトのアップロード、GETでオブジェクトのダウンロード
- POSTで、メタデータのみの更新が可能
- 最大サイズ5GB。ただし、マルチパートオブジェクト機能を利用することにより、疑似的に5GBを超えるオブジェクトも扱える。
- Etag(MD5ハッシュ）によるオブジェクトデータの破損検出
- オブジェクトコピー機能。オブジェクトのGETとPUTを連続して自動的に実行させることができる。

オブジェクト

- オブジェクトの一部データのみを取得可能
- 指定した時刻後の更新の有無でオブジェクト取得するか否かを指定可能
- 最新のオブジェクトデータ読み出しを保証する指定ができる（Diabloリリース）。Eventual Consistencyモデルの問題の回避可能
  - ただし、オーバヘッドが大きい
- 有効期限付きのオブジェクト生成可能(Essexリリース）
- バージョニング機能は、Folsomリリースにて

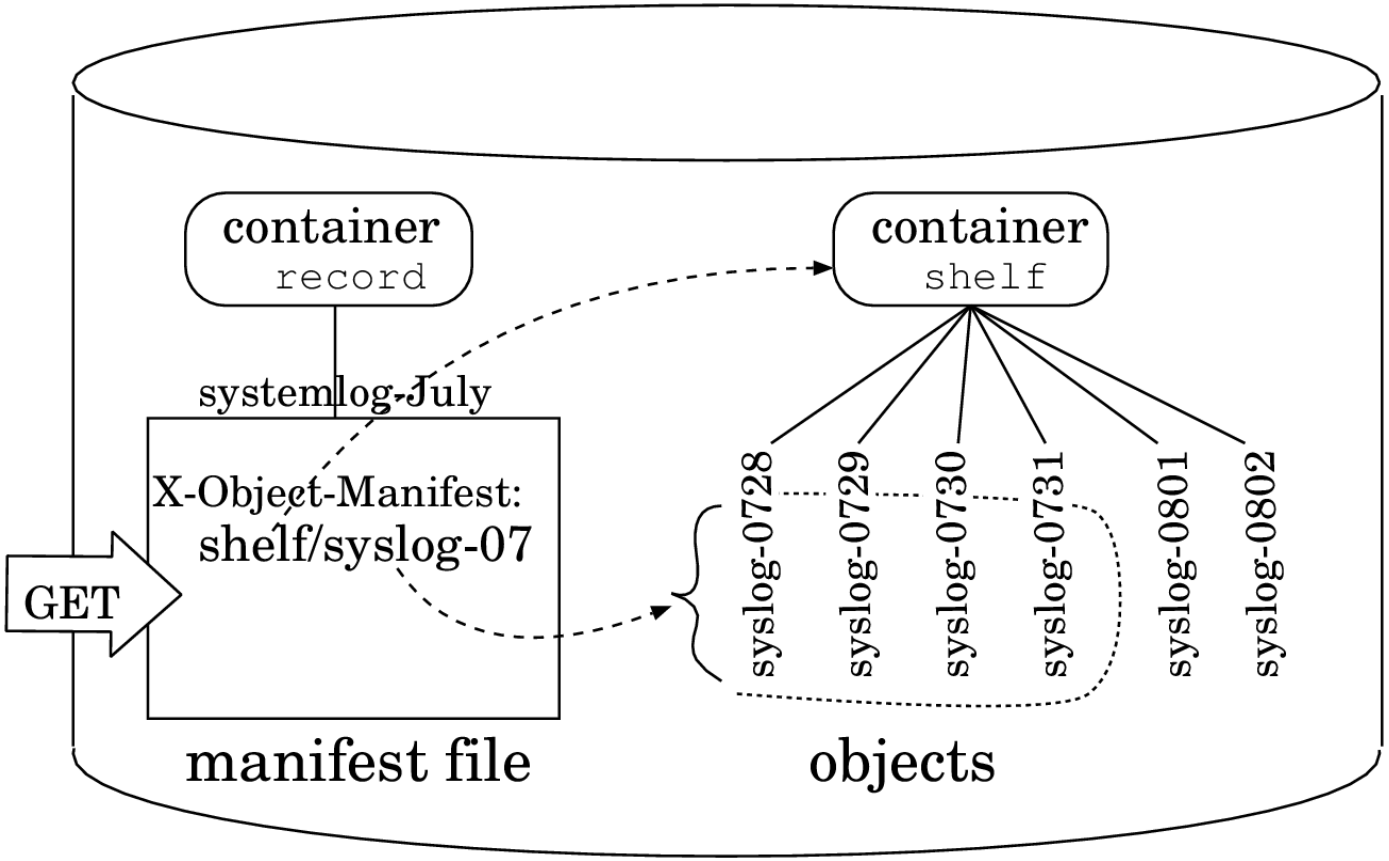
```
PUT /v1/AUTH_orion-cabinet/src/swift-1.3.0.taz HTTP/1.1
User-Agent: curl/7.21.0 libcurl/7.21.0 OpenSSL/0.9.8o zlib/1.2.3.4 libidn/1.18
Host: objstore.example.jp
Accept: */*
X-Auth-Token: AUTH_tk3b0094c49a90435b9f86f580cb39e363
Content-Length: 397030
```

```
HTTP/1.1 201 Created
Content-Length: 118
Content-Type: text/html; charset=UTF-8
Etag: 97e0b18bb44fbd8b2ea38b06e166a0fd
Last-Modified: Thu, 07 Jul 2011 07:47:44 GMT
Date: Thu, 07 Jul 2011 07:47:44 GMT

<html>
 <head>
  <title>201 Created</title>
 </head>
 <body>
  <h1>201 Created</h1>
  <br /><br />
 </body>
</html>
```

マルチパートオブジェクト

- Manifestファイル経由の間接参照



アクセス制御

- コンテナ単位で制御
- ユーザ毎にRead権限、Write権限を指定する。他のアカウントのユーザも指定できる
- Referer指定で、Read権限を与えることが可能
- コンテナに対するPUT/POST時の、X-Container-Read、X-Container-Writeヘッダにて、ACLを設定

アクセス制御

- アカウント名のみを指定すると、そのアカウント配下の全ユーザを指定したことになる
- リファラによるアクセス許可はRead権限のみ

```
POST /v1/AUTH_orion-cabinet/foo HTTP/1.1
User-Agent: curl/7.19.7 libcurl/7.19.7  OpenSSL/0.9.8k zlib/1.2.3.3 libidn/1.15
Host: objstore.example.jp
Accept: */*
X-Auth-Token: AUTH_tk4967d312608749e39c49956dd889589d
X-Container-Read: orion,pleiades:beth,pleiades:joe, r:.valinux.co.jp
X-Container-Write: orion:mary,orion:cris
```

```
HTTP/1.1 204 No Content
Content-Length: 0
Content-Type: text/html; charset=UTF-8
Date: Fri, 22 Jul 2011 12:56:52 GMT
```

Amazon S3とSwift

S3とSwiftの差異

ツールの作成や移植を行う場合、両オブジェクトストレージに共通のサービスを用意する場合など、双方の仕様の違いの把握が重要になる

- S3は、APIとしてReSTとSOAPをサポート
  - SwiftはReSTのみ
- S3のサービス(service)、バケット(bucket)が、それぞれSwiftのアカウント(account)、コンテナ(container)に対応する

リソースへのパス

- S3のバケットはシステム全体でユニークであるが、Swiftのコンテナはアカウント内でのみユニークである
  - S3は、リソースを仮想ホスト(virtual)ホスト形式のURLで指定が可能であるが、Swiftではその機能を実現できない

例）

従来のパス形式指定 : http://s3.amazonaws.com/bucketname/objectname
仮想ホスト形式指定: http://bucketname.s3.amazonaws.com/objectname

バケットとコンテナ

- バケットの属性にバケット作成時刻があるが、コンテナには無い
- コンテナの属性には、コンテナ内のオブジェクト数と総容量の情報があるが、バケットには無い
- バケットには作成上限数（１００）がある

認証

- S3はアカウントは契約単位でもあり、ユーザの単位でもある。一方Swiftでは、1アカウントの中に複数ユーザを定義できる階層構造になっている。
- S3では、全リクエストに認証に必要な情報を載せる必要がある。Swiftでは認証は最初の一度のみ。

アクセス制御

- Swiftはコンテナ単位でアクセス制御をおこなうが、S3ではオブジェクト単位で制御可能。
- S3は、複雑な制御を行うポリシーという機構も提供

オブジェクト

- S3はオブジェクトのバージョニングができる。Swiftで同様の機能がサポートされるのは、Folsomリリースから
- オブジェクトのコピー機能において、Swiftはアカウントを跨るコピーができない。S3では可能。
- オブジェクトのメタデータの更新(変更)ができるのは、Swiftのみ

SwiftのS3エミュレーション機能

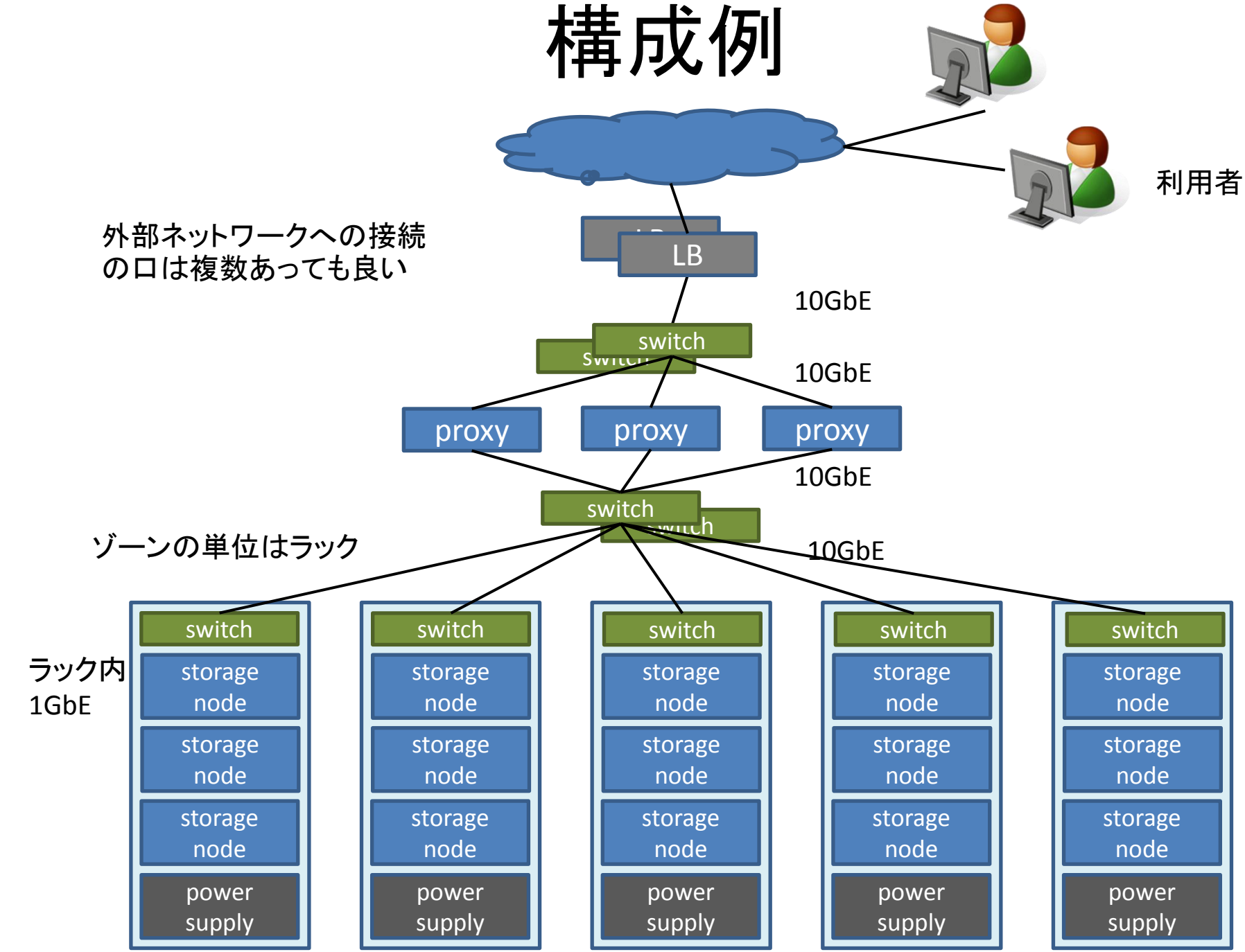
- Swifは、Amazon S3 APIをエミュレートするswift3ミドルウェアを用意している
  - 基本的な機能のみをサポート
  - SwiftネイティブAPIと併存させて利用することが可能

インストールと初期設定

推奨ハードウェア構成

- プロキシノード
  - 2台以上（冗長化と負荷分散）
  - CPU、ネットワークがボトルネック
- ストレージノード
  - アカウントノード、コンテナノード、オブジェクトノード同居
  - 多数のストレージを接続、ディスクI/Oがボトルネック
  - ゾーン（ノードのグルーピング）は5つ以上（耐障害性）
- ネットワーク
  - Proxy周りは10GbE
  - ラック内は1GbE

構成例


利用者
外部ネットワークへの接続
の口は複数あっても良い
LB
10GbE
switch
10GbE
proxy
proxy
proxy
10GbE
switch
ゾーンの単位はラック
10GbE
ラック内
1GbE
switch
storage node
storage node
storage node
power supply

ディストリビューション

- Ubuntu
  - 推奨。OpenStackプロジェクトが開発基盤としている。RackSpace社でも利用、利用実績が大きい。
  - Swiftは、Ubuntuに収録されているバージョンのパッケージに強く依存。最新機能をどんどん利用している。
- RHEL/CentOS
  - Swiftを動作させるためのパッケージ導入に苦労
    - Redhat社が提供していないパッケージ、Swiftが要求するバージョンと合わないパッケージが多数
  - Redhat社が、OpenStackプロジェクトへのコントリビュートを大々的に始めた。RHEL上では容易に動かせない問題は、近く解消されることが期待される

プロキシノードの設定

- NTP設定
  - 全プロキシノードの時刻の同期が必須
- /etc/swift/swift.confの設定
- memcachedの起動
- /etc/swift/proxy-server.confの設定
- リングの作成

/etc/swift/swift.conf

- リソース名（コンテナ名やオブジェクト名）のMD5ハッシュを計算するときに利用する種をswift_hash_path_suffixに設定
  - 後からの変更は不可能

```
[swift-hash]
# random unique (preferably alphanumeric) string that can never change (DO NOT LOSE)
swift_hash_path_suffix = 5c747739000bb7c3
```

/etc/swift/proxy-server.conf

- super_admin_keyにシステム管理者（super admin)のパスワード設定
- default_swift_clusterには、ロードバランサを指すURLを指定

```
[DEFAULT]
bind_port = 8080
workers = 8
user = swift

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

[app:proxy-server]
use = egg:swift#proxy
allow_account_management = true
log_facility = LOG_LOCAL1

[filter:swauth]
use = egg:swift#swauth
default_swift_cluster = local#https://swift.example.jp/v1
super_admin_key = passwordfoobar

[filter:healthcheck]
use = egg:swift#healthcheck

[filter:cache]
use = egg:swift#memcache
memcache_servers = 192.168.0.21:11211
```

ストレージノードの設定

- ntp設定
- データ格納用デバイス（ディスク）の初期化
- rsync設定
- /etc/swift/account-server.conf、container-server.conf、object-server.confの設定
- リングの用意

データ格納用デバイスの初期化

- ファイルシステムはXFS推奨(xattrが利用できることが必須）
  - iノードの大きさは1024バイトに拡張した方が処理効率が良い
- /srv/node配下に、デバイス名のディレクトリを作り、それぞれmountする

```
# mount /dev/sdb1 /srv/node/sdb1
```

object-server.conf

- 基本設定のままで動作するが、workersはCPUコア数に合わせて変更した方が良い

```
[DEFAULT]
workers = 2

[pipeline:main]
pipeline = object-server

[app:object-server]
use = egg:swift#object

[object-replicator]

[object-updater]

[object-auditor]
```

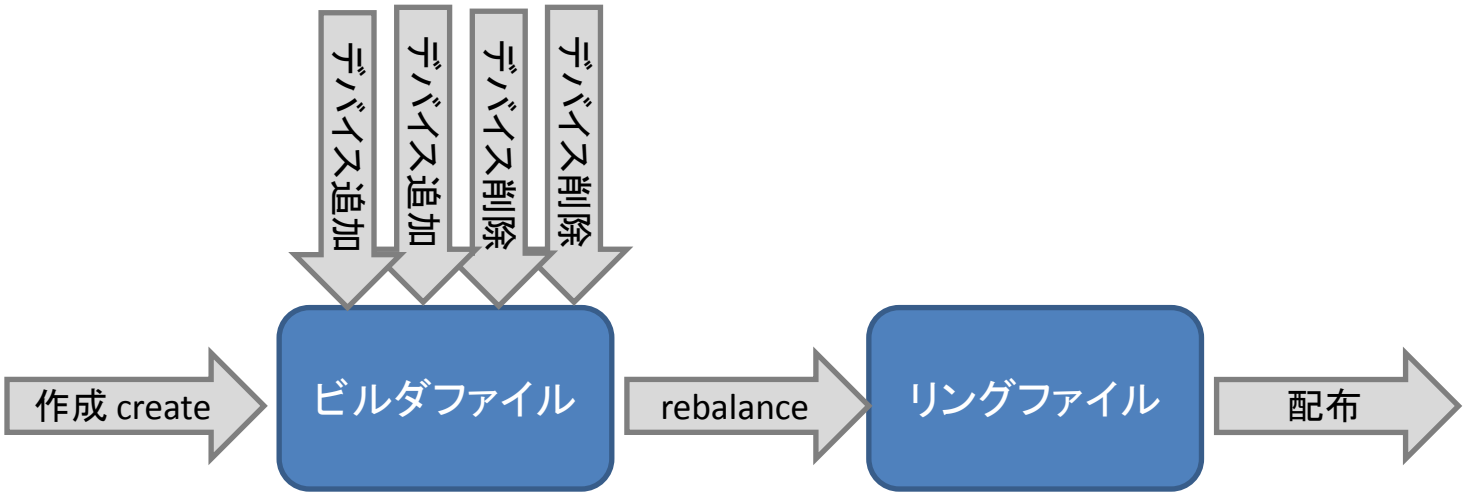
リングの作成

- アカウント、コンテナ、オブジェクト管理のためのリングをそれぞれ作成
  - ストレージノードに同居させる構成では、同じ設定で作成すればよい
- すべてのノードに同じリングを配布する

リングの作成手順

1. ビルダファイルの作成
2. ディスク(デバイス)の登録・削除
3. ビルダファイルをリングファイルに変換
4. リングを全ノードに配布

✓ リングを作成し直すとき、つまりハードウェア構成を変更するときは、手順2から行う



リングの作成

- ビルダファイル（中間ファイル）を経由して作成する。
  - リングを再構成するときにも必要になる
- レプリカ数（複製数）は、3が推奨
- パーティション数（リソースのグループ数のこと）は総ディスク本数の100倍が推奨
  - この値は後から変更不可。将来の拡張を見越した値に設定する
- パーティションの連続移動用抑制時間は24Hが推奨

```
# cd /etc/swift
# swift-ring-builder object.builder create 18 3 24
```

リングの作成

- ビルダファイルにデバイス（ディスク）を登録
  - ディスクが存在するノードとゾーンを指定
  - ディスク容量に偏りがあるときは、weight値で容量の相対値を指定

```
# swift-ring-builder object.builder add z1-172.17.222.31:6000/sdb1 1
# swift-ring-builder object.builder add z1-172.17.222.31:6000/sdc1 1
# swift-ring-builder object.builder add z2-172.17.222.32:6000/sdb1 1
# swift-ring-builder object.builder add z2-172.17.222.32:6000/sdc1 1
# swift-ring-builder object.builder add z3-172.17.222.33:6000/sdb1 1
# swift-ring-builder object.builder add z3-172.17.222.33:6000/sdc1 1
# swift-ring-builder object.builder add z4-172.17.222.34:6000/sdb1 1
# swift-ring-builder object.builder add z4-172.17.222.34:6000/sdc1 1
# swift-ring-builder object.builder add z5-172.17.222.35:6000/sdb1 1
# swift-ring-builder object.builder add z5-172.17.222.35:6000/sdc1 1
```

- 設定が終わったらリングファイルを生成する

```
# swift-ring-builder object.builder rebalance
```

アカウント、ユーザの作成

- アカウントの作成には、サービス提供者(reseller admin)、またはシステム管理者(super admin)権限が必要

```
# swauth-add-account -A https://objstore.example.jp/auth/ -U reseller:rgroup -K rpass123 -s orion-cabinet orion
```

- ユーザの作成は、アカウント管理者(admin)が行うことができる

```
# swauth-add-user -A https://objstore.example.jp/auth/ -U orion:joe -K passjoe -a orion mary passmary
```

- サービス提供者(reseller admin)、またはシステム管理者(super admin)であれば、アカウント管理者(admin)権限を持ったユーザを作ることができる

```
# swauth-add-user -A https://objstore.example.jp/auth/ -U reseller:rgroup -K rpass123 -a orion joe passjoe
```

動作確認

- Swift上のオブジェクトを簡単に操作するためのコマンドst（Diablo以降はswiftと名称を変更）が用意されている

アカウントの情報を取得する

```
# st -A https://objstore.example.jp/auth/v1.0 -U orion:joe -K passjoe stat
```

mydataディレクトリにあるファイルを、コンテナconにアップロードする。

```
# st -A https://objstore.example.jp/auth/v1.0 -U orion:joe -K passjoe upload -c con mydata
```

コンテナcon配下のオブジェクト一覧表示

```
# st -A https://objstore.example.jp/auth/v1.0 -U orion:joe -K passjoe list con
```

コンテナcon配下をダウンロード

```
# st -A https://objstore.example.jp/auth/v1.0 -U orion:joe -K passjoe download con
```

各種ミドルウェア

- 必要な機能をノードに組み込むことができる
  - healthcheck : ノードの生死監視用
  - ratelimit : リクエストの受付け制限
  - domain_remap : ドメイン名の変換
  - catch_errors : 詳細エラーログ出力
  - cname_lookup : ストレージURLの変換
  - staticweb : オブジェクトをWEBコンテンツとして公開
  - swift3 : Amazon S3エミュレーション
  - name_check : リソース名の有効性チェック(Essex)
  - tempurl : モバイル環境からのアクセス用(Essex)

障害対応

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。リング作成時の中間ファイル（ビルダファイル）もバックアップする
- オブジェクトのバックアップ
  - 標準APIを利用して独自に実装する
  - 通常のバックアップソフトウェアの利用（小規模構成時のみ）
  - コンテナのsync機能
    - Swift 1.4.3 (Diabloリリース)から利用可能
  - 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。コードはほぼ完成している。

高負荷対策

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

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
  - 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

Swift開発動向

開発動向 – Diablo (Swift 1.4.3)

- 移植用認証ミドルウェア tempauth
- slogging – statsの改良版
- recon – 各ノードの動作状況取得
- 最新オブジェクトデータの読み出し
  - Eventual Consistency問題の回避
- コンテナのレプリケーション機能
- アカウントデータベース自動生成
- swiftコマンド （旧stコマンドの名称変更）

開発動向 – Essex (Swift 1.4.8)

- 有効期限付きのオブジェクト
- TempURLミドルウェア
- 認証機能IF v2.0 (Identity – keystone対応)
- マニュアルの整備
- GUI – DashBoard(Horizon)の提供

開発動向 – Folsomに向けて

- Folsom目標
  - オブジェクトのバージョニング
  - リアルタイムモニタリング
  - CDMI (Cloud Data Management Interface)対応
  - クライアントライブラリの充実
  - DashBoard(Horizon)強化 （GUI）
- Folsom以降？
  - 5GB超の大きなオブジェクトのPUT
  - 遠隔地のSwiftシステム上へのレプリカ（進んでない）

Tips

Tips

- Ubuntuで動作させるのがお勧め
  - RHEL/CentOSでの運用は苦労する
  - 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

Tips

- プロキシ間で時刻が揃っていることは必須
- 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
- 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
  - Essexリリースで、ようやくマニュアル類が整備された

予定: Swift (2)
実装知識とその応答

- 実装
  - データ構造と処理フロー
  - Eventual Consistency
  - 障害発生とレプリケーション
- 致命的障害からの手動復旧
- Swiftの改造
  - 既存システムとの認証統合
  - クォータ機能の実装
  - スナップショットの実装
- Tips