Swift Object Storage: Adding Erasure Codes

Paul Luse, Sr. Staff Engineer – Intel Corporation
Kevin Greenan, Staff Software Engineer – Box
Sep 2014

SNIA Legal Notice

Abstract

- **Swift Object Storage: Adding Erasure Codes**
  - This session will provide insight into this extremely successful community effort of adding an Erasure Code capability to the OpenStack Swift Object Storage System by walking the audience through the design and development experience through the eyes of the developers from key contributors. An overview of Swift Architecture and basic Erasure Codes will be followed by design/implementation details.

Agenda

- Swift
  - A Community Project
  - Swift Overview
  - Storage Policies
- Erasure Codes
  - History
  - Variations
  - Matrix encode/decode
  - PyECLib & liberasurecode
- Erasure Code Implementation for Swift
  - Design considerations
  - Architecture overview

Swift: A Community Project

- Core OpenStack* Service
  - One of the original 2 projects
  - 100% Python
  - ~ 35K LOC
  - > 2x that in unit, functional, error injection code
- Vibrant community,
  - top contributing companies for Juno include: SwiftStack*, Intel, Redhat*, IBM*, HP*, Rackspace*, Box*
- The path to EC…

Early EC Discussion

Policies Development

EC Library Development

Policies Released

Swift EC Development

EC Ready!

Havana
Fall '13

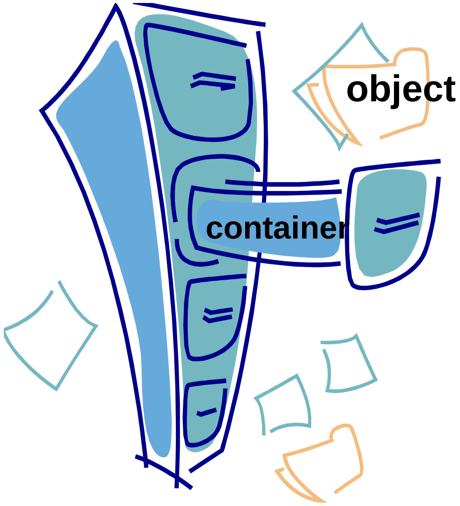
IceHouse
Spring '14

Juno
Fall '14

Kilo
Spring '15

Swift Overview

- Uses container model for grouping objects with like characteristics
  - Objects are identified by their paths and have user-defined metadata associated with them
- Accessed via RESTful interface
  - GET, PUT, DELETE
- Built upon standard hardware and highly scalable
  - Cost effective, efficient

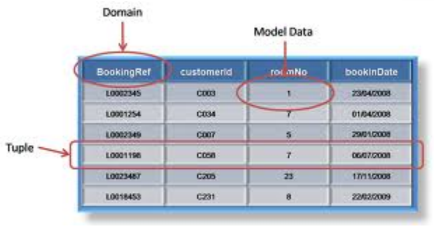


Objects are organized with containers

What Swift is Not

- Distributed File System
  - Does not provide POSIX file system API support

- Relational Database
  - Does not support ACID semantics



BookingRef	customerId	roomNo	bookinDate
L0002345	C003	1	23/04/2008
L0001254	C034	7	01/04/2008
L0002349	C007	5	29/01/2008
L0001198	C058	7	06/07/2008
L0023487	C205	23	17/11/2008
L0018453	C231	8	22/02/2009

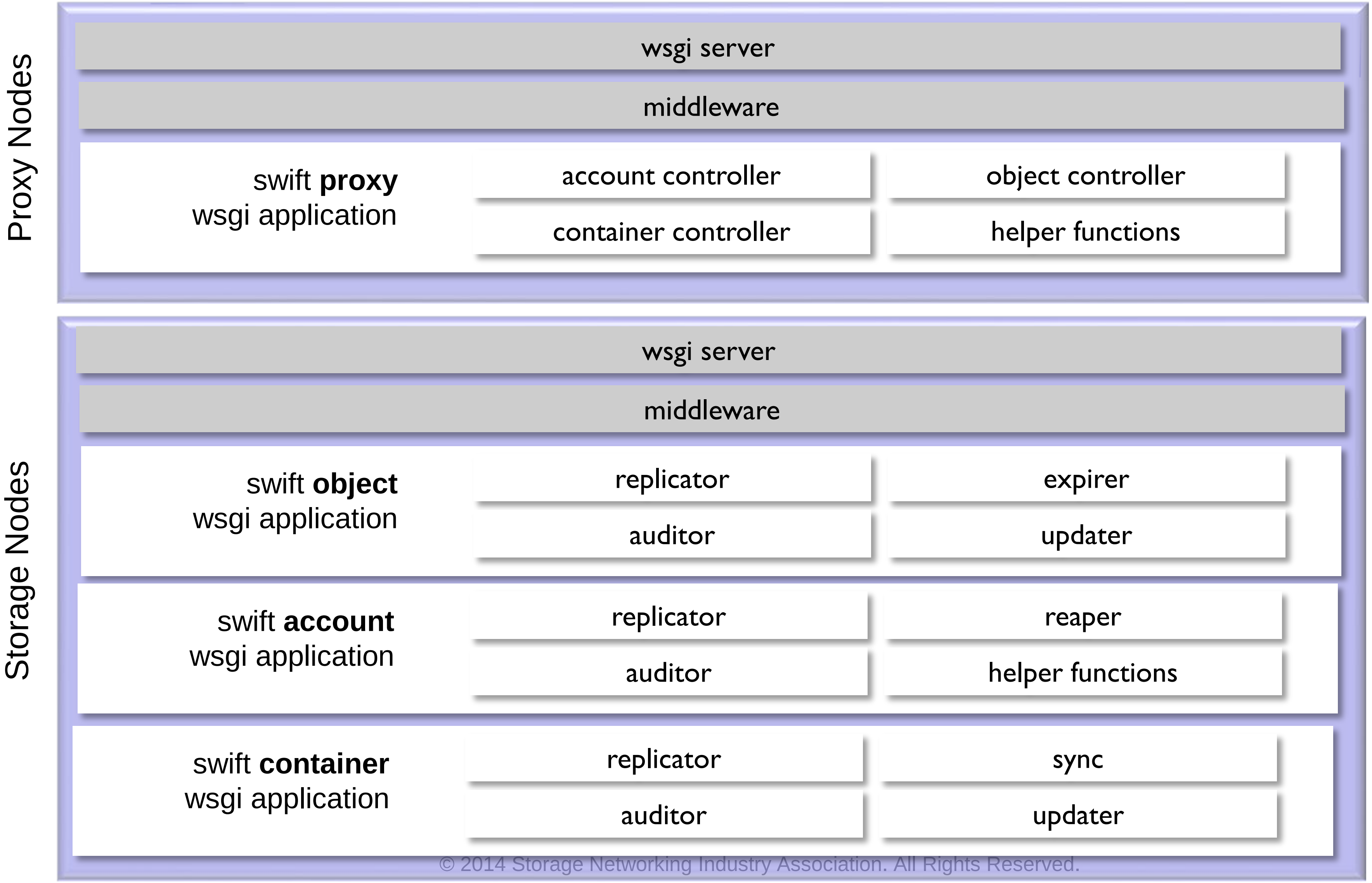
- NoSQL Data Store
  - Not built on the Key-Value/Document/Column-Family model

- Block Storage System
  - Does not provide block-level storage service

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19

Not a "One Size Fits All" Storage Solution

Swift Software Architecture



Proxy Nodes

wsgi server

middleware

swift **proxy**
wsgi application

account controller

object controller

container controller

helper functions

Storage Nodes

wsgi server

middleware

swift **object**
wsgi application

replicator

expirer

auditor

updater

swift **account**
wsgi application

replicator

reaper

auditor

helper functions

swift **container**
wsgi application

replicator

sync

auditor

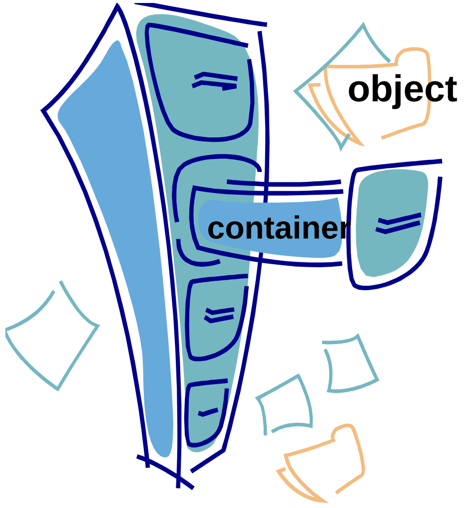
updater

Swift 2.0: Why Storage Policies?

- Prior to, durability scheme applies to entire cluster
  - Can do replication of 2x, 3x, etc., however the entire cluster must use that setting

- There were no core capabilities to expose or make use of differentiated hardware within the cluster
  - If several nodes of a cluster have newer/faster characteristics, they can’t be fully realized (the administrator/users are at the mercy of the dispersion algorithm alone for data placement).

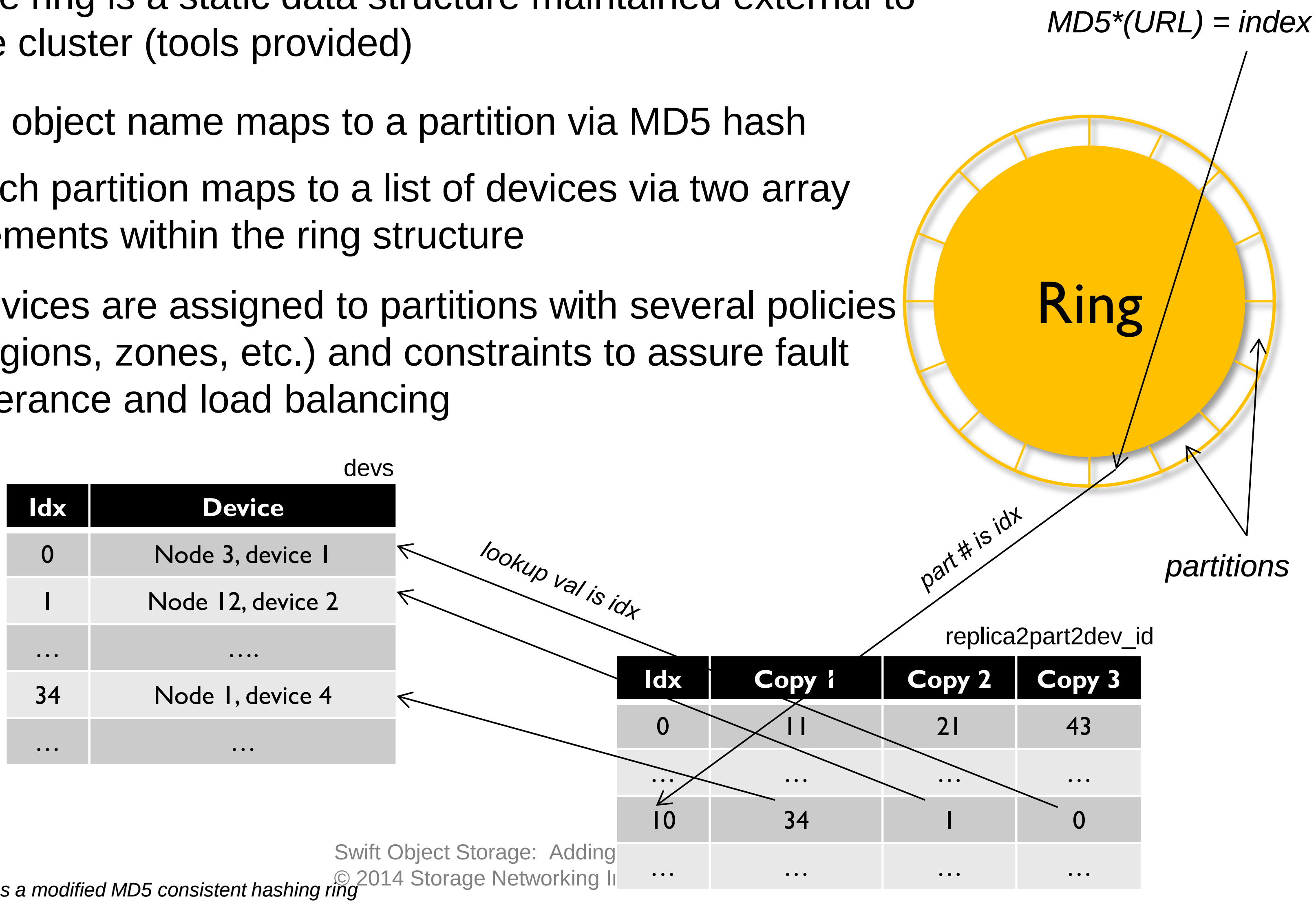
- There’s was no extensibility for additional durability schemes
  - Use of erasure codes (EC)
  - Mixed use of schemes (some nodes do 2x, some do 3x, some do EC)



You get N-replication in every container

The Swift Ring

- The ring is a static data structure maintained external to the cluster (tools provided)
- An object name maps to a partition via MD5 hash
- Each partition maps to a list of devices via two array elements within the ring structure
- Devices are assigned to partitions with several policies (regions, zones, etc.) and constraints to assure fault tolerance and load balancing



devs

Idx	Device
0	Node 3, device 1
1	Node 12, device 2
…	….
34	Node 1, device 4
…	…

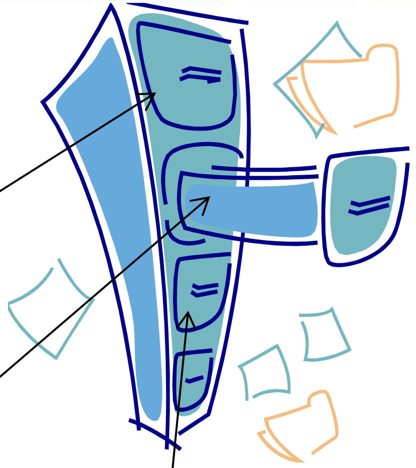
replica2part2dev_id

Idx	Copy 1	Copy 2	Copy 3
0	11	21	43
…	…	…	…
10	34	1	0
…	…	…	…

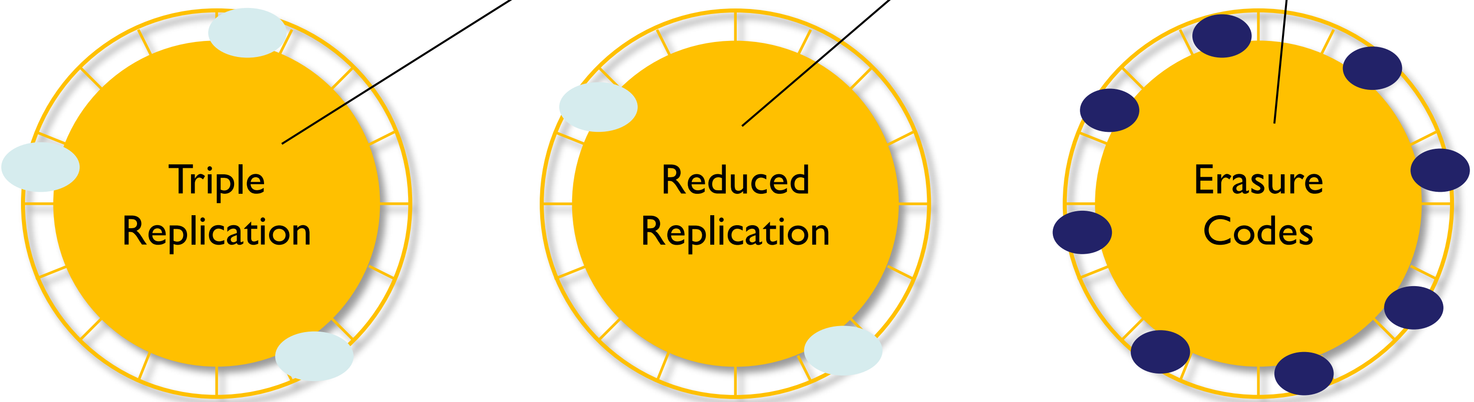
Note: *Swift uses a modified MD5 consistent hashing ring*

What are Policies?

- Introduction of multiple object rings
  - Swift supports multiple rings already, but only one for object – the others are for account and container DB.
- Introduction of container tag: X-Storage-Policy
  - New immutable container metadata
  - Policy change accomplished via data movement
  - Each container is associated with a potentially different ring



3 Different Policies:
3 Different Rings

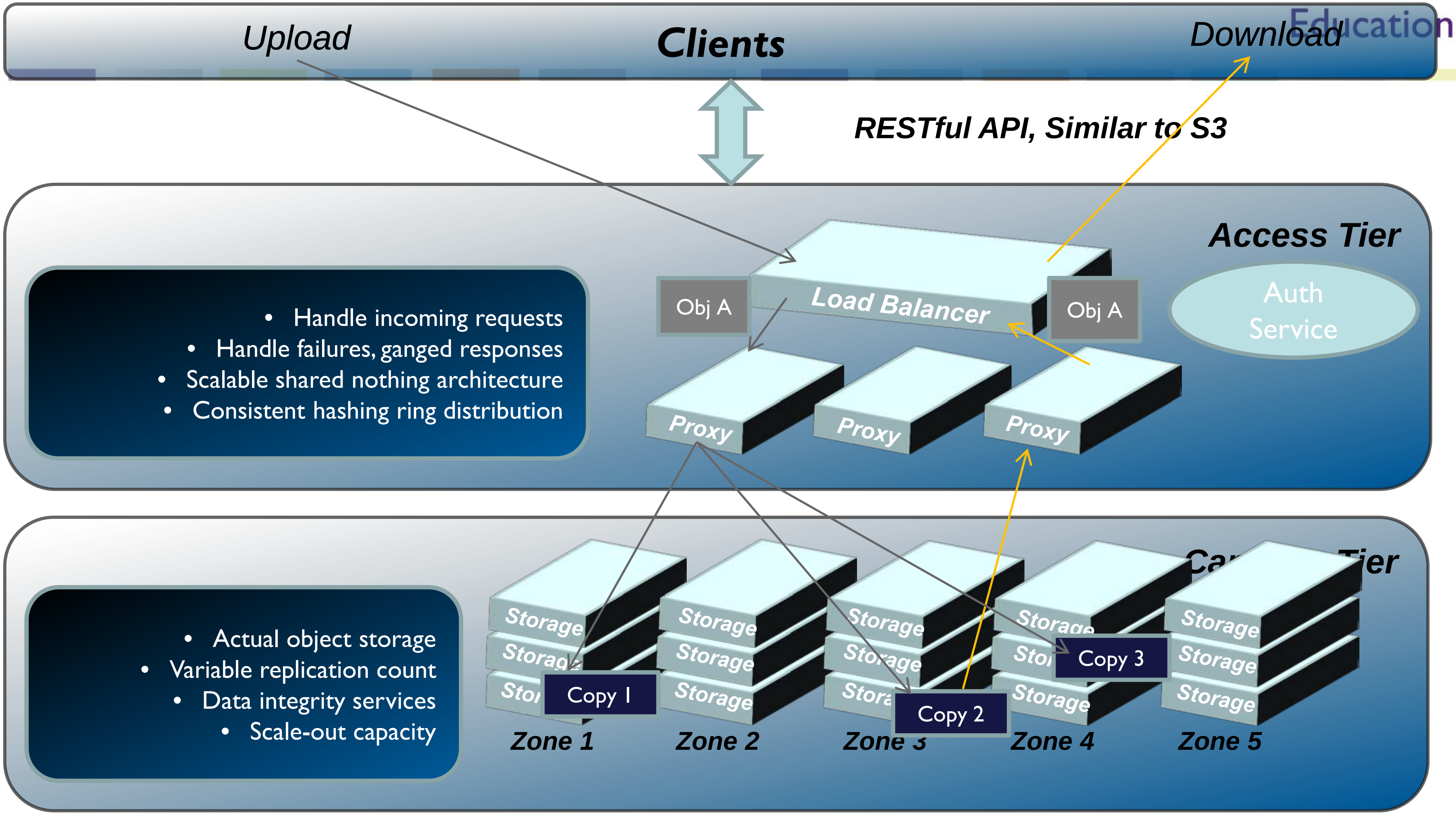


3 locations, same object

2 locations, same object

n locations, object fragments

Putting it All Together



- Handle incoming requests
- Handle failures, ganged responses
- Scalable shared nothing architecture
- Consistent hashing ring distribution

- Actual object storage
- Variable replication count
- Data integrity services
- Scale-out capacity

Scalable for concurrency and/or capacity independently

Agenda

- Swift
  - Swift Overview
  - Storage Policies
- Erasure Codes
  - Background
  - Example encode/decode using Reed-Solomon
  - Minimizing reconstruction cost
  - PyECLib & liberasurecode
- Erasure Code Implementation for Swift
  - Design considerations
  - Architecture overview

History of Erasure Coding

1960’s

- Coding Theory
  - Reed Solomon, Berlekamp–Massey algorithm

1990’s

- Storage
  - RAID-6: EVENODD, RDP, X-Code
- Graph Theory
  - LDPC Codes (Tornado, Raptor, LT)

2000’s

- Coding Theory
  - Network / Regenerating Codes

2010’s

- Storage
  - Non-MDS codes for cloud and recovery

Source: Tutorial on Erasure Coding for Storage Applications, Part 1. James Plank FAST ‘13

Terminology

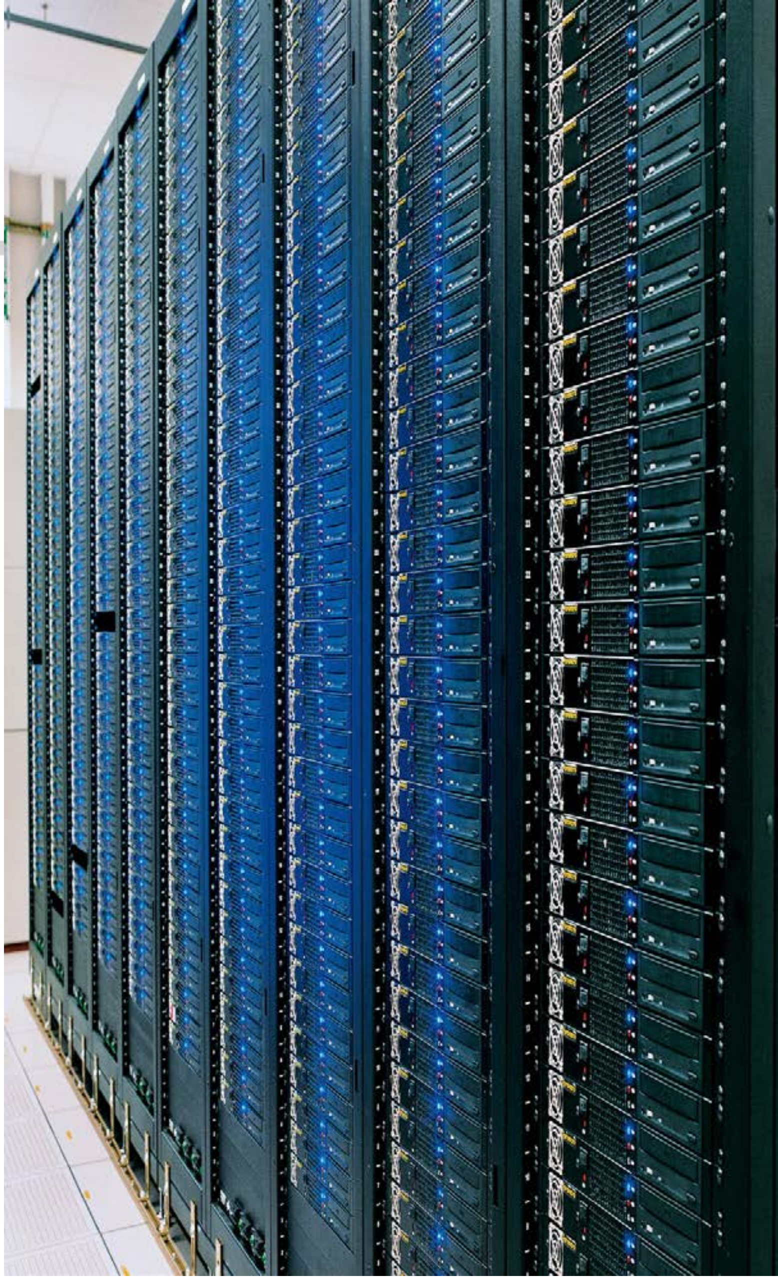
- Split a file into ***k*** chunks and encode into ***n*** chunks, where ***n-k=m***
- Systematic vs. Non-systematic
  - Systematic: encoded output contains input symbols
  - Non-systematic: encoded output ***does not*** contain input symbols
- Code word
  - A set of data and parity related via a set of parity equations
  - Systematic: f(data) = code word = (data, parity)
- Layout
  - Flat horizontal: each coded symbol is mapped to one device
  - Array codes have multiple symbols per device: horizontal and vertical
- MDS vs. non-MDS
  - MDS: any ***k*** chunks can be used to recover the original file
  - Non-MDS: ***k*** chunks may not be sufficient to recover the file

Traditionally, storage systems use systematic, MDS codes

Variations

- Reed-Solomon Codes
- Fountain Codes
- RAID-6 EVENODD
- RAID-6 X-Code
- Generalized XOR
- Pyramid Codes
- Local Repairable Codes (LRC)
- Partial MDS Codes (PMDS)
- Simple Regenerating Codes

and the list goes on…

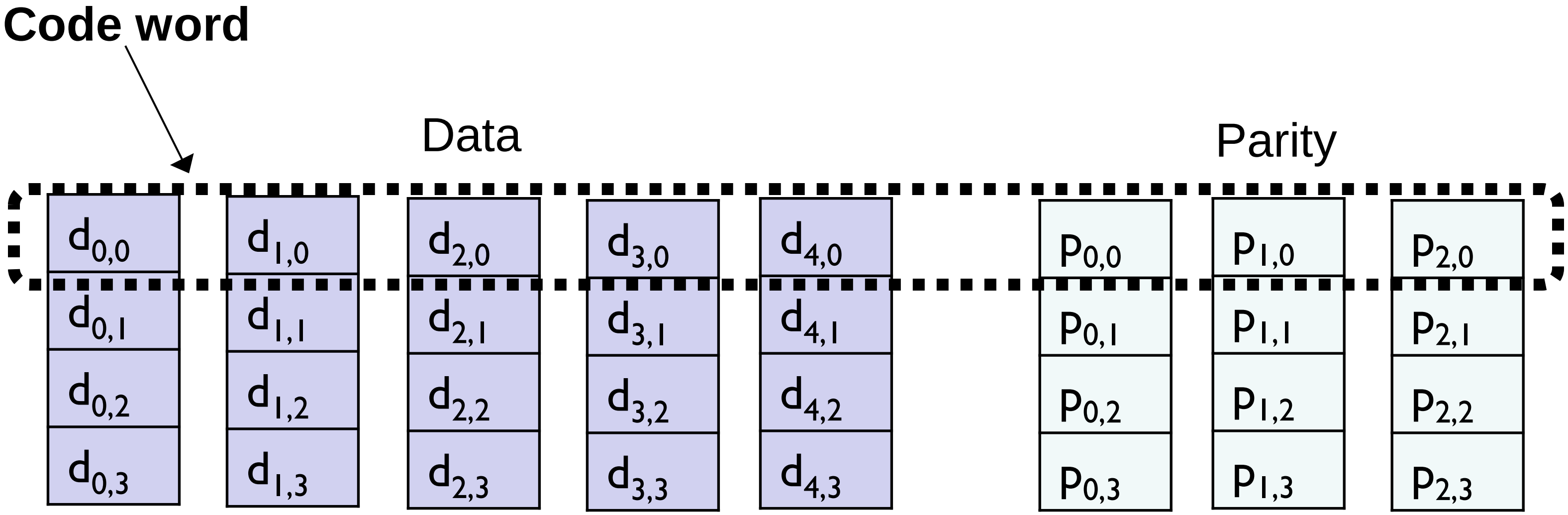


Reed Solomon Systematic Horizontal Code Layout

Example RS(8,5)

total disks = n = 8
data disks = k = 5
parity disks = m = 3

- ***Can Tolerate (n-k) Failures***
- ***Overhead of just (n/k)***



Code word

Data

Parity

$d_{0,0}$	$d_{1,0}$	$d_{2,0}$	$d_{3,0}$	$d_{4,0}$	$p_{0,0}$	$p_{1,0}$	$p_{2,0}$
$d_{0,1}$	$d_{1,1}$	$d_{2,1}$	$d_{3,1}$	$d_{4,1}$	$p_{0,1}$	$p_{1,1}$	$p_{2,1}$
$d_{0,2}$	$d_{1,2}$	$d_{2,2}$	$d_{3,2}$	$d_{4,2}$	$p_{0,2}$	$p_{1,2}$	$p_{2,2}$
$d_{0,3}$	$d_{1,3}$	$d_{2,3}$	$d_{3,3}$	$d_{4,3}$	$p_{0,3}$	$p_{1,3}$	$p_{2,3}$

Reed Solomon Systematic Generator Matrix

$$\begin{aligned} f(\alpha_0) &= y_0 \\ f(\alpha_1) &= y_1 \\ &\dots \\ f(\alpha_{n-1}) &= y_{n-1} \end{aligned}$$

Reed-Solomon is ***encoded*** by oversampling a polynomial

$$f(x) = c_0 + c_1x^1 + c_2x^2 + \dots + c_{k-1}x^{k-1}$$

Coefficients are the data

$f(\alpha_0)$

$$\begin{bmatrix} 1 & \alpha_0^1 & \alpha_0^2 & \alpha_0^3 & \alpha_0^4 \\ 1 & \alpha_1^1 & \alpha_1^2 & \alpha_1^3 & \alpha_1^4 \\ 1 & \alpha_2^1 & \alpha_2^2 & \alpha_2^3 & \alpha_2^4 \\ 1 & \alpha_3^1 & \alpha_3^2 & \alpha_3^3 & \alpha_3^4 \\ 1 & \alpha_4^1 & \alpha_4^2 & \alpha_4^3 & \alpha_4^4 \\ 1 & \alpha_5^1 & \alpha_5^2 & \alpha_5^3 & \alpha_5^4 \\ 1 & \alpha_6^1 & \alpha_6^2 & \alpha_6^3 & \alpha_6^4 \\ 1 & \alpha_7^1 & \alpha_7^2 & \alpha_7^3 & \alpha_7^4 \end{bmatrix} \xrightarrow[\text{Result has same rank}]{\text{Elementary Ops}} \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ g_0 & g_1 & g_2 & g_3 & g_4 \\ g_5 & g_6 & g_7 & g_8 & g_9 \end{bmatrix}$$

Reed Solomon Systematic Matrix Encoding Process

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ g_0 & g_1 & g_2 & g_3 & g_4 \\ g_5 & g_6 & g_7 & g_8 & g_9 \end{bmatrix} \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \\ d_4 \end{bmatrix} = \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \\ d_4 \\ p_1 \\ p_2 \\ p_3 \end{bmatrix}$$

Generator Matrix data code word

All operations are done in a Galois field

Any (k x k) sub-matrix is invertible

Code word is the vector-matrix product of the generator matrix and source data

$p_i = d_0 + g_i d_1 + g_i^2 d_2 + g_i^3 d_3 + \ldots + g_i^{k-1} d_{k-1}$

Reed Solomon Systematic Matrix Decoding Process

$$\begin{bmatrix} \cancel{1} & \cancel{0} & \cancel{0} & \cancel{0} & \cancel{0} \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ \cancel{0} & \cancel{0} & \cancel{0} & \cancel{1} & \cancel{0} \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ g_0 & g_1 & g_2 & g_3 & g_4 \\ \cancel{g_5} & \cancel{g_6} & \cancel{g_7} & \cancel{g_8} & \cancel{g_9} \end{bmatrix} \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \\ d_4 \end{bmatrix} = \begin{bmatrix} \xcancel{d_0} \\ d_1 \\ d_2 \\ \xcancel{d_3} \\ d_4 \\ p_1 \\ p_2 \\ \xcancel{p_3} \end{bmatrix}$$

These disks just died

Generator Matrix × data = parity

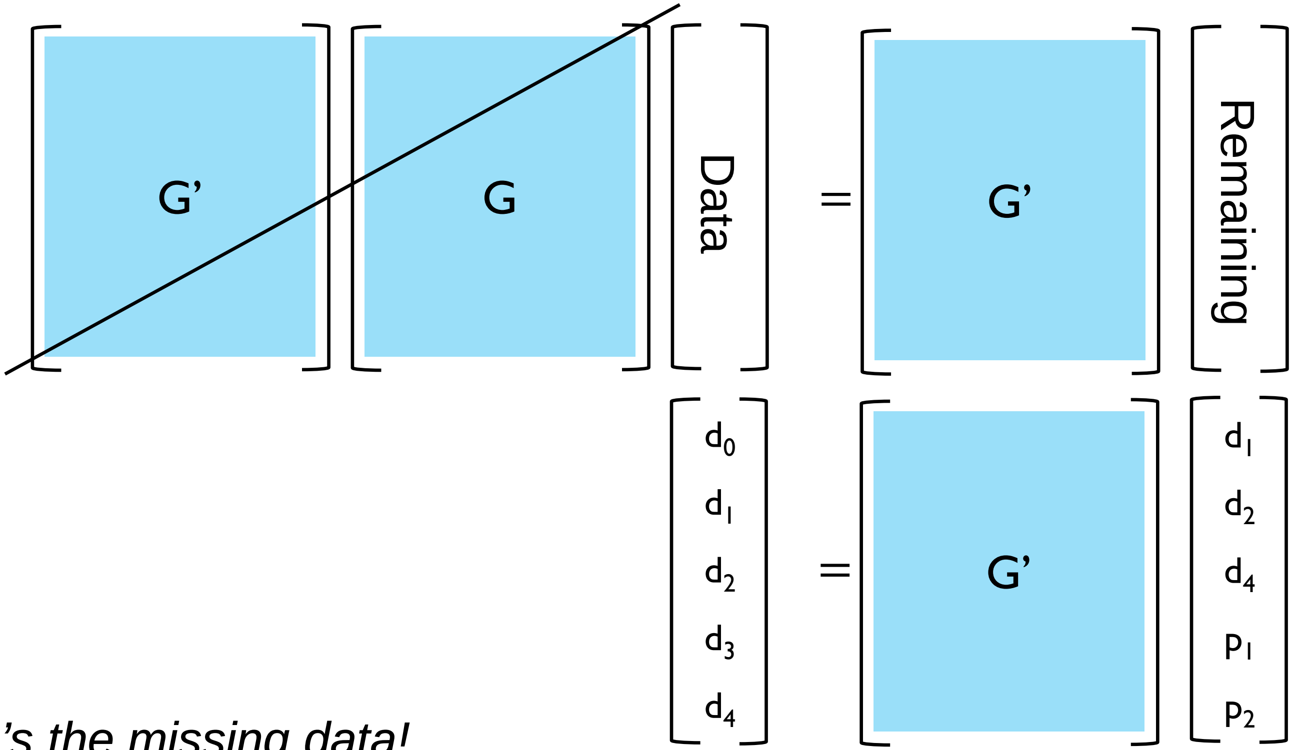
Step 1: Eliminate all but k available rows in the generator matrix

Reed Solomon Systematic Matrix Decoding Process

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ g_0 & g_1 & g_2 & g_3 & g_4 \end{bmatrix} \xrightarrow{\text{Invert}} \begin{bmatrix} G' \end{bmatrix}$$

Step 2:Invert the resulting matrix

Reed Solomon Systematic Matrix Decoding Process



Here's the missing data!

Step 3: "Solve" by multiplying k element vector of available symbols by corresponding rows of G'

$$d_0 = g'_{00}d_1 + g'_{01}d_2 + g'_{02}d_4 + g'_{03}p_1 + g'_{04}p_2$$

$$d_3 = g'_{30}d_1 + g'_{31}d_2 + g'_{32}d_4 + g'_{33}p_1 + g'_{34}p_2$$

Minimizing Reconstruction Cost

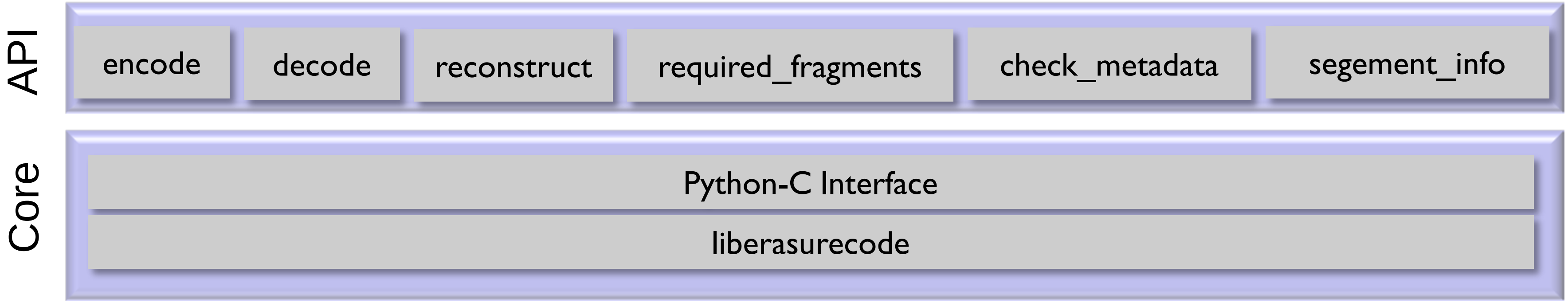
- Reed-Solomon requires ***k*** available elements to reconstruct any missing element
- This has given rise to many codes that minimize repair costs
  - Regenerating codes, locally repairable codes, flat-XOR codes, etc.
  - Trade space efficiency for more efficient reconstruction
- Replication repair-optimal, RS is space-optimal, these are somewhere in the middle
- Simple XOR-only example with k = 6, m = 4:

$$\begin{aligned} P_0 &= \cancel{D_0} + D_1 + D_3 \\ P_1 &= D_1 + D_2 + D_5 \\ P_2 &= \cancel{D_0} + D_2 + D_4 \\ P_3 &= D_3 + D_4 + D_5 \end{aligned} \quad \longrightarrow \quad \begin{aligned} D_0 &= P_0 + D_1 + D_3 \\ D_0 &= P_2 + D_2 + D_4 \end{aligned}$$

Only requires 3 devices to reconstruct one failed device

PyECLib

- ***Goal***: provide a pluggable, easy-to-use EC library for Python
- Swift is the main use-case
- Originally had ***all logic*** in PyECLib, but have offloaded "smarts" to liberasurecode
  - Separation of concerns: one converts between Python and C, and the other does erasure coding
  - API of PyECLib is same as liberasurecode



API

encode

decode

reconstruct

required_fragments

check_metadata

segement_info

Core

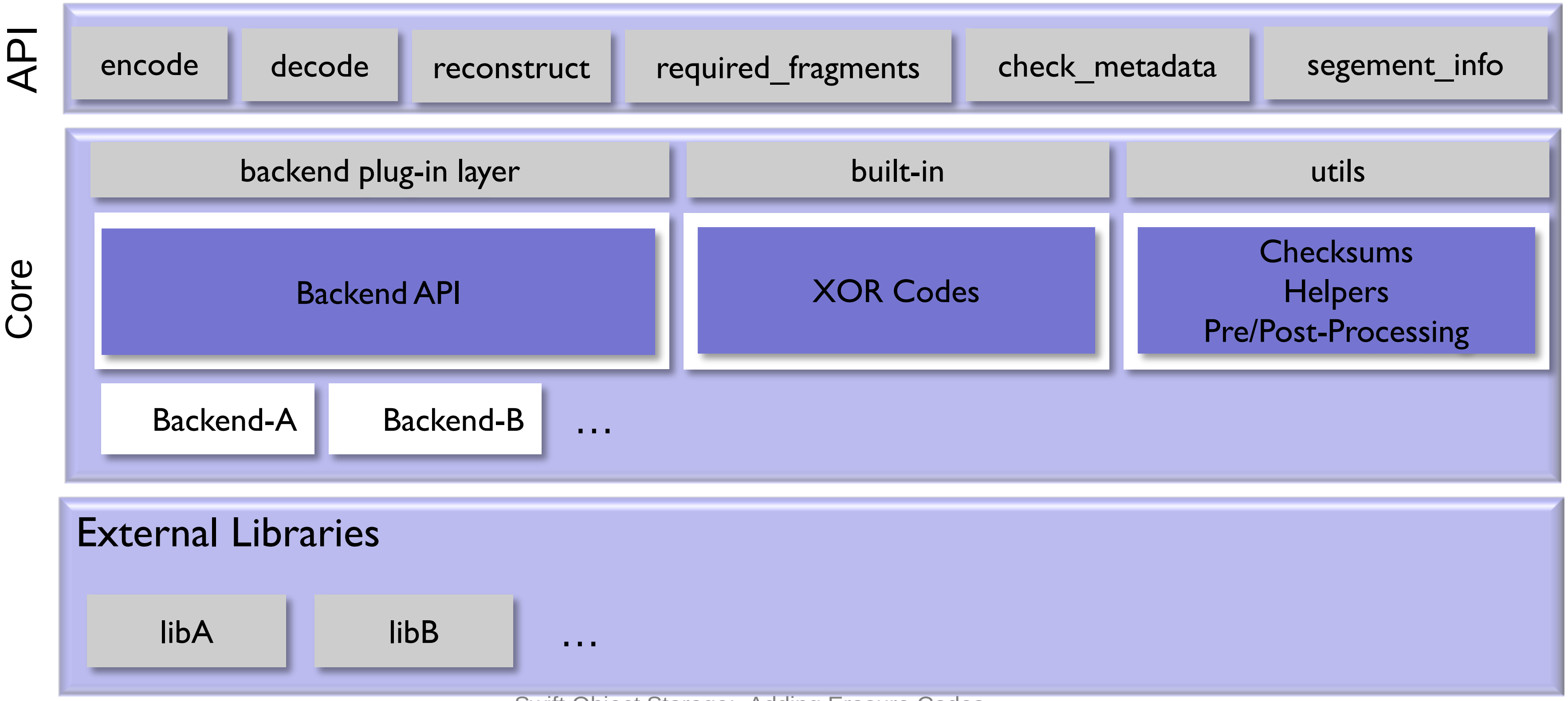
Python-C Interface

liberasurecode

†Relative tests conducted with home grown test code using ISA-L libr

liberasurecode

- ***Goal***: Separate EC-specific logic from language-specific translation
- Embedded metadata: original file size, checksum, version info, etc.
- Provides ability to plug-in and use new erasure code schemes/libraries
  - In addition to XOR codes, we currently provide Jerasure and ISA-L



API

encode

decode

reconstruct

required_fragments

check_metadata

segement_info

Core

backend plug-in layer

built-in

utils

Backend API

XOR Codes

Checksums
Helpers
Pre/Post-Processing

Backend-A

Backend-B

…

External Libraries

libA

libB

…

†Relative tests conducted with home grown test code using ISA-L libr

Agenda

- Swift
  - Swift Overview
  - Storage Policies
- Erasure Codes
  - History
  - Variations
  - Matrix encode/decode
  - PyECLib & liberasurecode
- Erasure Code Implementation for Swift
  - Design considerations
  - Architecture overview

Design Considerations

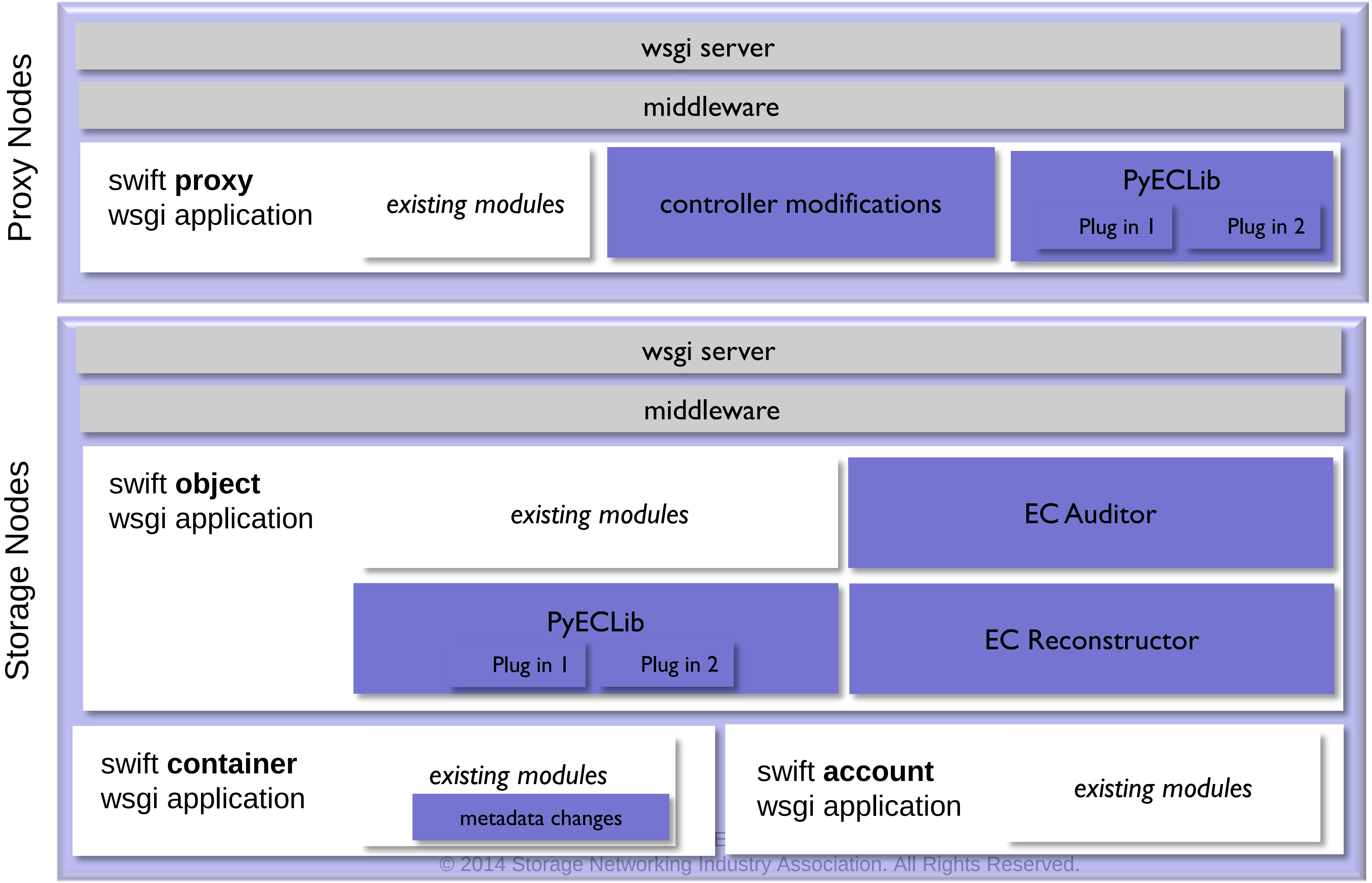
- GET/PUT Erasure Code encode/decode done at proxy server
  - Aligned with current Swift architecture to focus hardware demanding services in the access tier
  - Enable in-line Erasure Code directed by client as well as off-line Erasure Code directed by sideband application / management tier

- Build Upon Storage Policies
  - New container metadata will identify whether objects within it are erasure coded

- Keep it simple and leverage current architecture
  - Multiple new storage node services required to assure Erasure Code chunk integrity as well as Erasure Code stripe integrity; modeled after replica services
  - Storage nodes participate in Erasure Code encode/decode for reconstruction analogous to replication services synchronizing objects

Swift With EC Architecture High Level

SNIA Education



Proxy Nodes

wsgi server

middleware

swift **proxy**
wsgi application

existing modules

controller modifications

PyECLib

Plug in 1

Plug in 2

Storage Nodes

wsgi server

middleware

swift **object**
wsgi application

existing modules

EC Auditor

PyECLib

Plug in 1

Plug in 2

EC Reconstructor

swift **container**
wsgi application

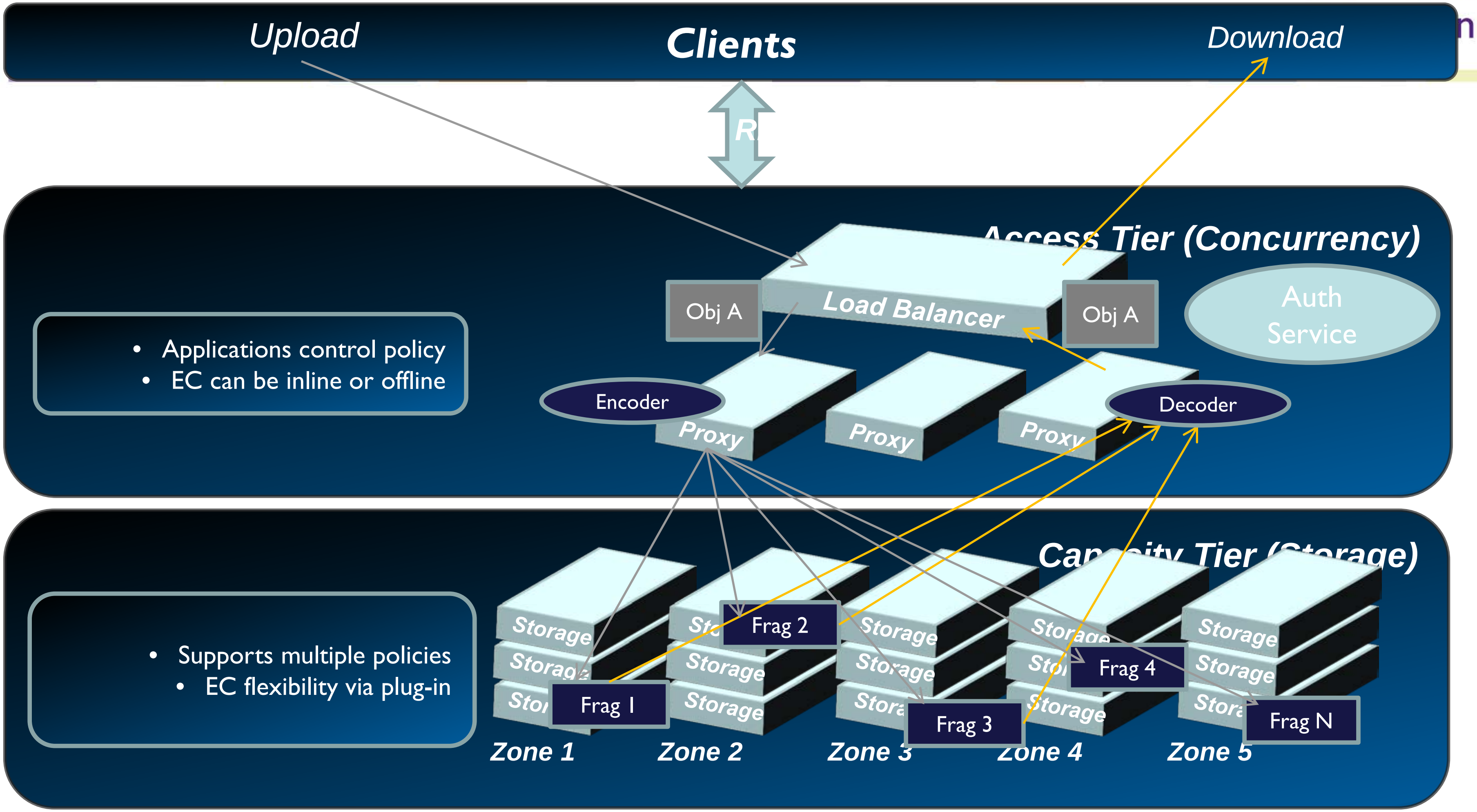
existing modules

metadata changes

swift **account**
wsgi application

existing modules

Swift With Erasure Code



- Applications control policy
- EC can be inline or offline

- Supports multiple policies
- EC flexibility via plug-in

Erasure Code Technology Lowering TCO for Swift

For More Information…

- Trello discussion board:
  https://trello.com/b/LlvIFIQs/swift-erasure-codes
- Launchpad blueprints:
  https://blueprints.launchpad.net/swift
- Swift Code (see feature/EC branch):
  https://code.launchpad.net/swift
- PyECLib:
  https://bitbucket.org/kmgreen2/pyeclib
- Liberasurecode:
  https://bitbucket.org/tsg-/liberasurecode

Attribution & Feedback

The SNIA Education Committee thanks the following individuals for their contributions to this Tutorial.

Authorship History

Name/Date of Original Author here:
Paul Luse, Kevin Greenan. 8/2014

Updates:

None

Additional Contributors

None

Please send any questions or comments regarding this SNIA Tutorial to ***tracktutorials@snia.org***

Backup

Block, File & Object

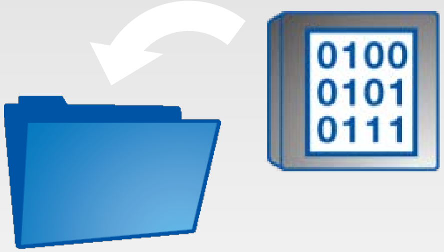
Block

Specific location on disks / memory

Tracks

Sectors

File



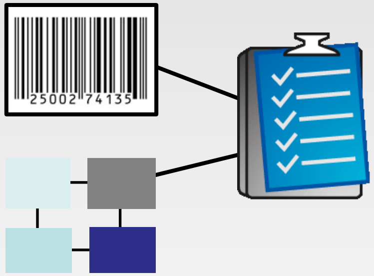
Specific folder in fixed logical order

File path

File name

Date

Object



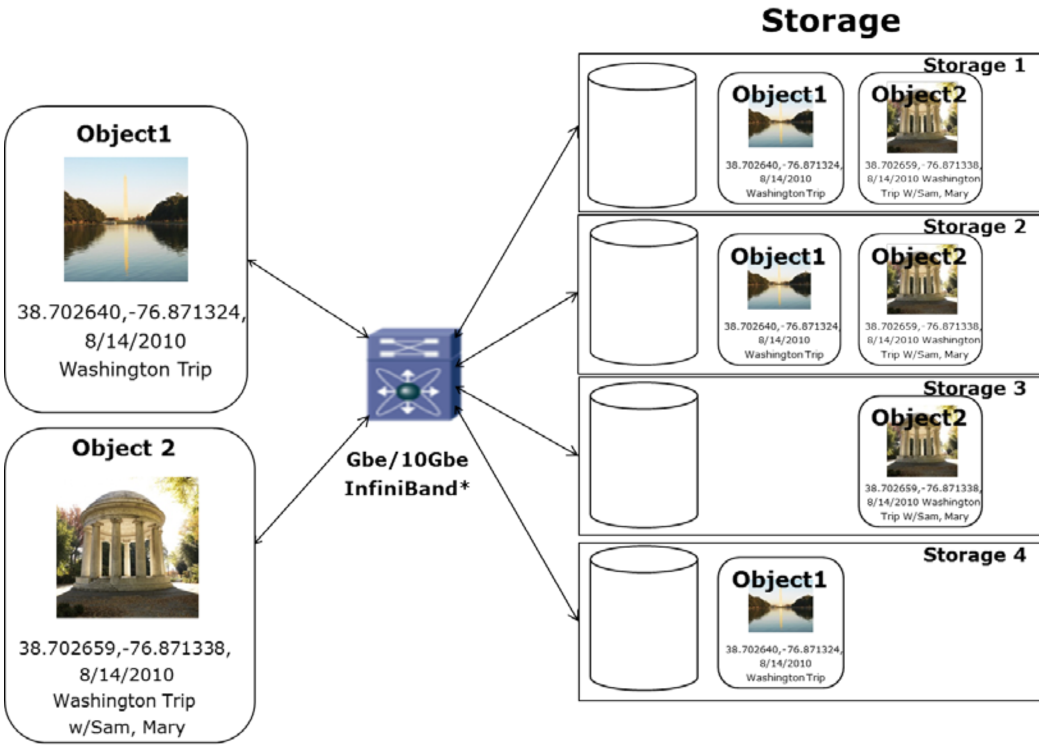
Flexible container size

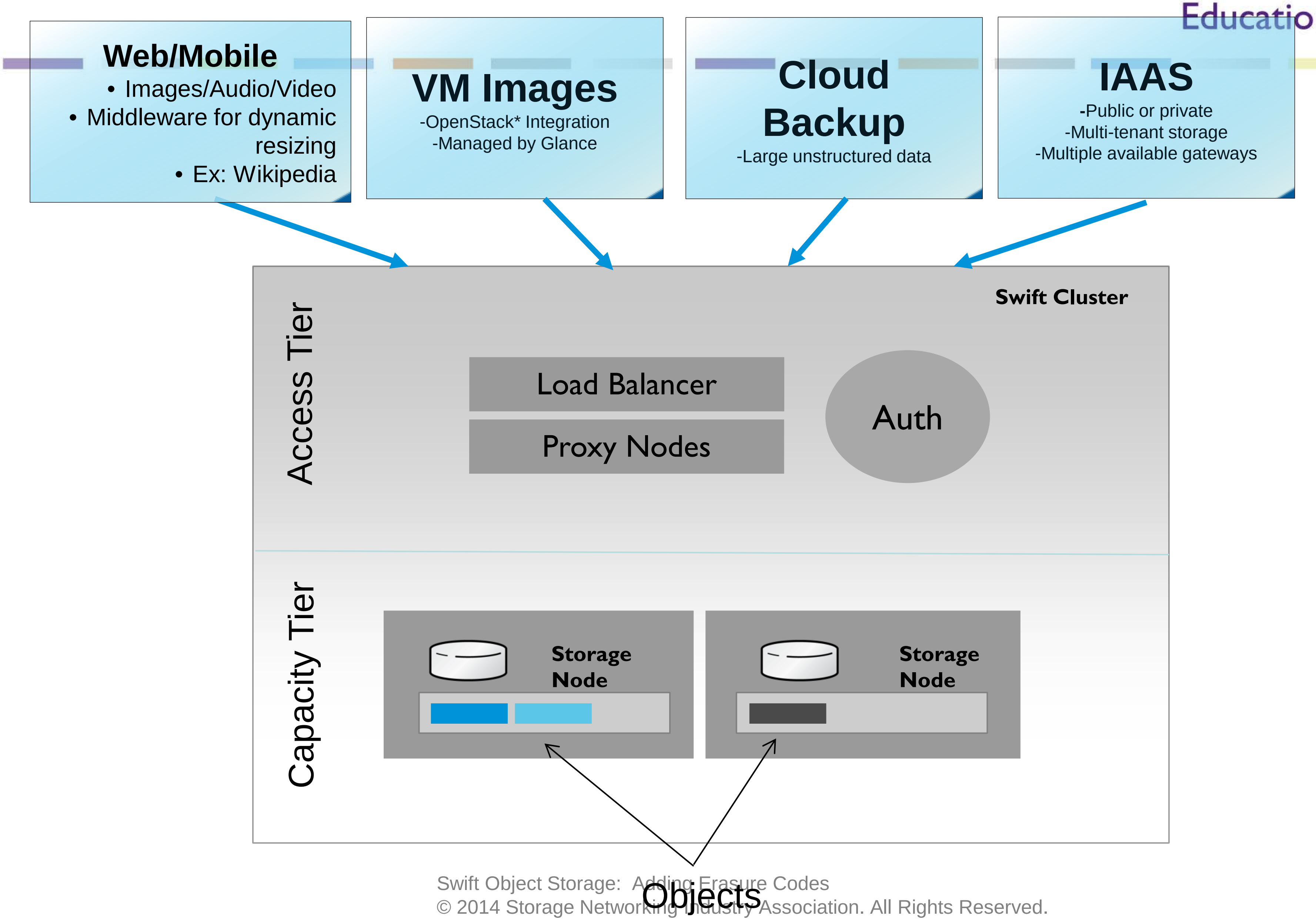
Data and Metadata

Unique ID

Object Store

- Scalability
  - Flat namespace
  - No volume semantics
  - No Locking/Attributes
  - Contains metadata
- Durability
  - Replication or Erasure code
- Manageability
  - REST API
  - Low overhead
- Consistency
  - Eventually consistent





Web/Mobile
- Images/Audio/Video
- Middleware for dynamic resizing
- Ex: Wikipedia

VM Images

-OpenStack* Integration
-Managed by Glance

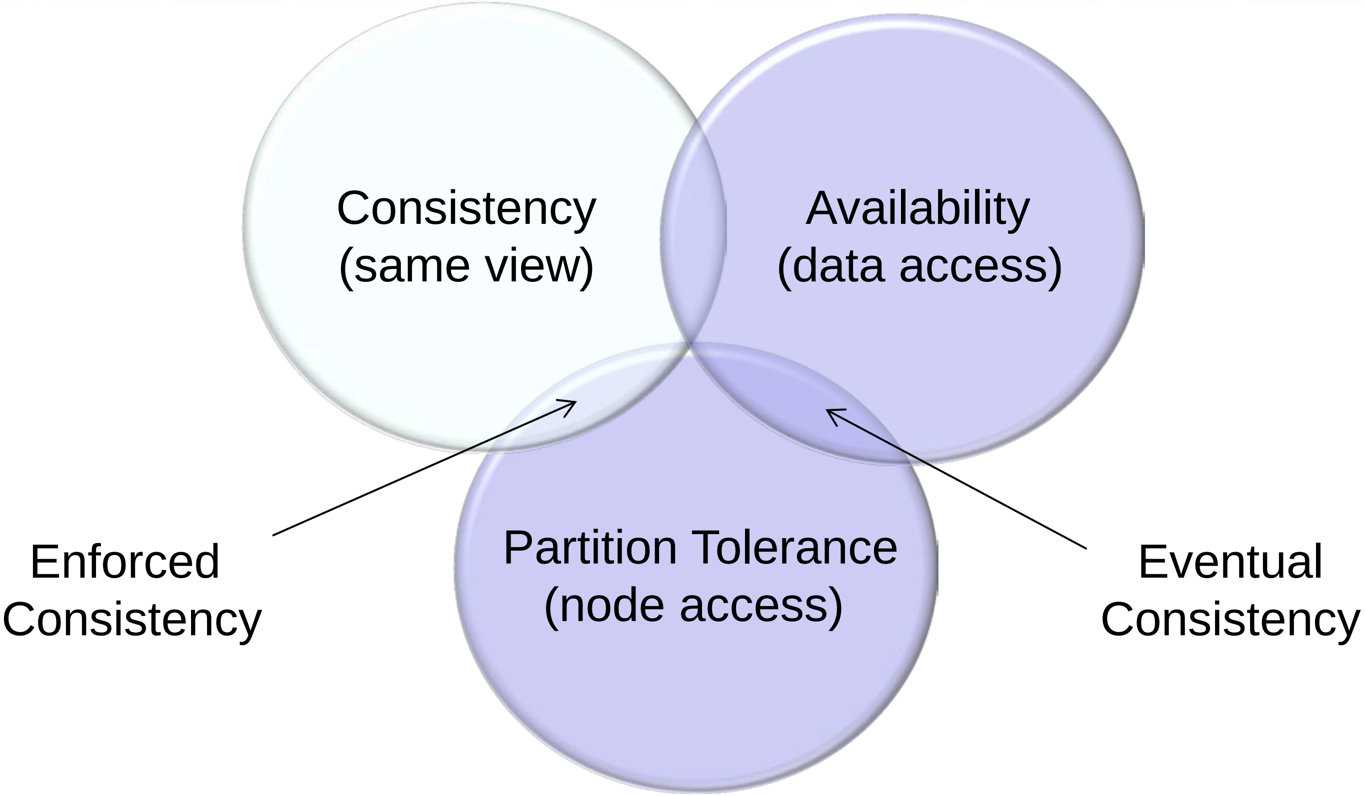
Cloud Backup

-Large unstructured data

IAAS

-Public or private
-Multi-tenant storage
-Multiple available gateways

The CAP Theorem: Pick 2



Consistency
(same view)

Availability
(data access)

Partition Tolerance
(node access)

Enforced
Consistency

Eventual
Consistency

Swift chooses **Availability** and **Partition Tolerance** *over* **Consistency**