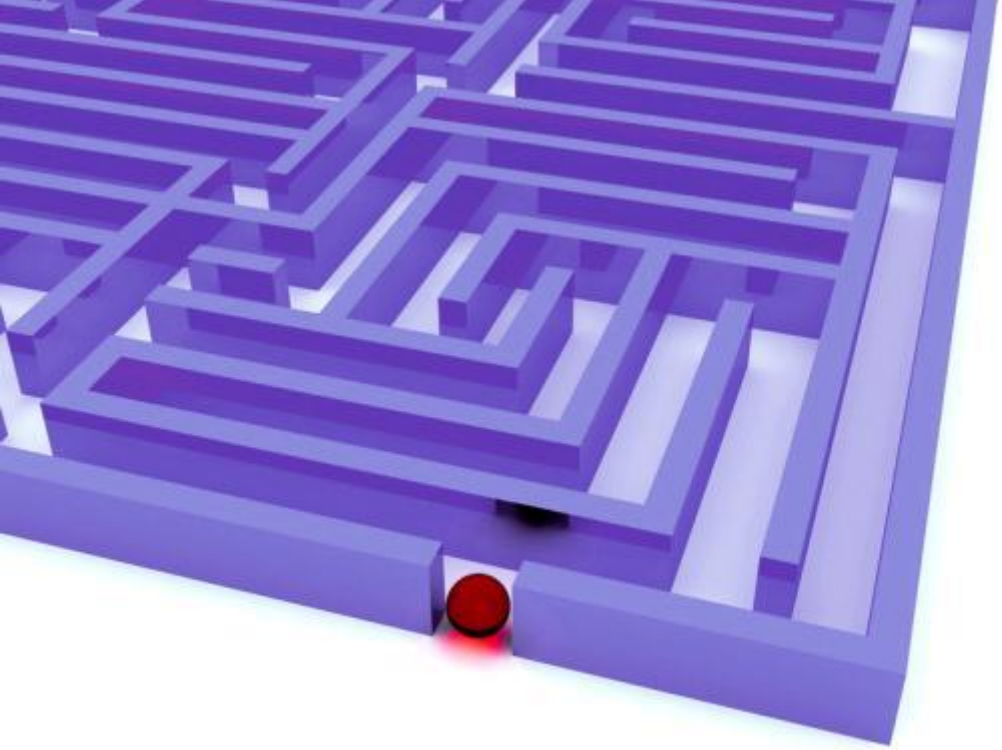


calsoft

Object Striping in Swift

OpenStack Summit 2013, Hong Kong

Author/Presenter: Shriram Pore, Sr. Architect
Contributors: Bipin Kunal, Kashish Bhatia

openstack™

Agenda

- Problem statement
- Proposed solution
- Solution prerequisites
- Approach - I : client unaware striping - striping and collation @ proxy
- Approach - II : client aware striping - striping and collation @ client
- Use-cases
- Enhancements and future scope
- Appendix
- See also

Problem statement

- Traditionally object stores are viewed as stores for smaller size objects
- Swift support for large objects (dynamic and static)
  - By segmentation
  - Via manifest file support
- Limitations
  - GET and PUT are costly in terms of time, network and storage utilization
  - Manifest file is limited by number of segments that can be supported
  - **Varying-size objects handled differently**

Proposed Solution – **Object Striping in Swift**

Object striping + (Sparse Object + Parallel Read/Write + Vectored IO)

- Object Striping
  - Object Segmentation / Chunking
  - Not confined to large objects
- Sparse Object
  - Not all the stripes of the object need to be consumed
  - Object size and on-disk object size may differ vastly (e.g. 1 GB object could have 1 MB on-disk size)
- Parallel Read/Write from/to Multiple Object servers
  - Stripes of an object span across multiple partitions and hence object servers
  - Optimally involve maximum possible object servers in the read/write operations
- Vectored IO
  - Multiple stripes stored in the sub-object are read/written via vectored IO to optimize the IO performance

Solution Prerequisites

- Following slides define the prerequisites changes to implement the proposed solution

Glossary

Term	Description
Object ID	Hash of URL /account/container/object
Stripe ID	An unique identifier for stripe within an object
Stripe Size	Size of the stripe
Sub object	Portion of an object, consisting of one or more stripes stored sparsely in a partition
Metadata cache	Cache of striping information at swift clients
Fingerprint	MD5 of stripe data

Prerequisite - Container Database Schema

- Container DB – stores all the information about the objects associated with the container
- Schema **changes**
  - object ID = object identifier
  - **stripe size** = size of stripe(for the specific object)
  - **object size** = size of object(to determine number of stripes it has)
  - **object on-disk size** = total size of stripes in sparse object
  - **object version delta size** = total size of only changed stripes in sparse object
- Different objects may have different stripe size but for a given object, stripe size remains same
- Stripe size may vary between 1M-10M. Default stripe size is 1M

object ID	stripe size	object size	object on-disk size	object version delta size

Prerequisite - Stripe ID and Stripe Offset Formulae

- Stripe ID generated by object striping service

- Stripe ID is a function f

  f : Stripe ID -> (stripe offset,
  stripe size,
  [object path],
  [partition size])

- Stripe Offset is function g

  g : Stripe Offset -> (stripe id,
  stripe size,
  [object path],
  [partition size]) ([] - optional)

  where

  $g = f^{-1}$

Prerequisite - Role of Stripe ID Hash

- object-ID = hash of "/Account/Container/object"
- stripe-ID-hash = hash of "/Account/Container/object**/stripe-ID**"
- Salient points
  - object-ID does not play any role in determining the partition
  - stripe-ID-hash is used to decide the partition
  - Partition is now a set of stripe-id hashes
- Advantage
  - Evenly distribute stripes across partitions
  - Optimize multiple object server participation in PUT/GET request handling
  - Ring re-balancing in case of addition or removal of object server nodes

Prerequisites- Extended Attributes

- Metadata as extended attributes
- Per-stripe extended attributes is used by Replicator, Auditor and Updater

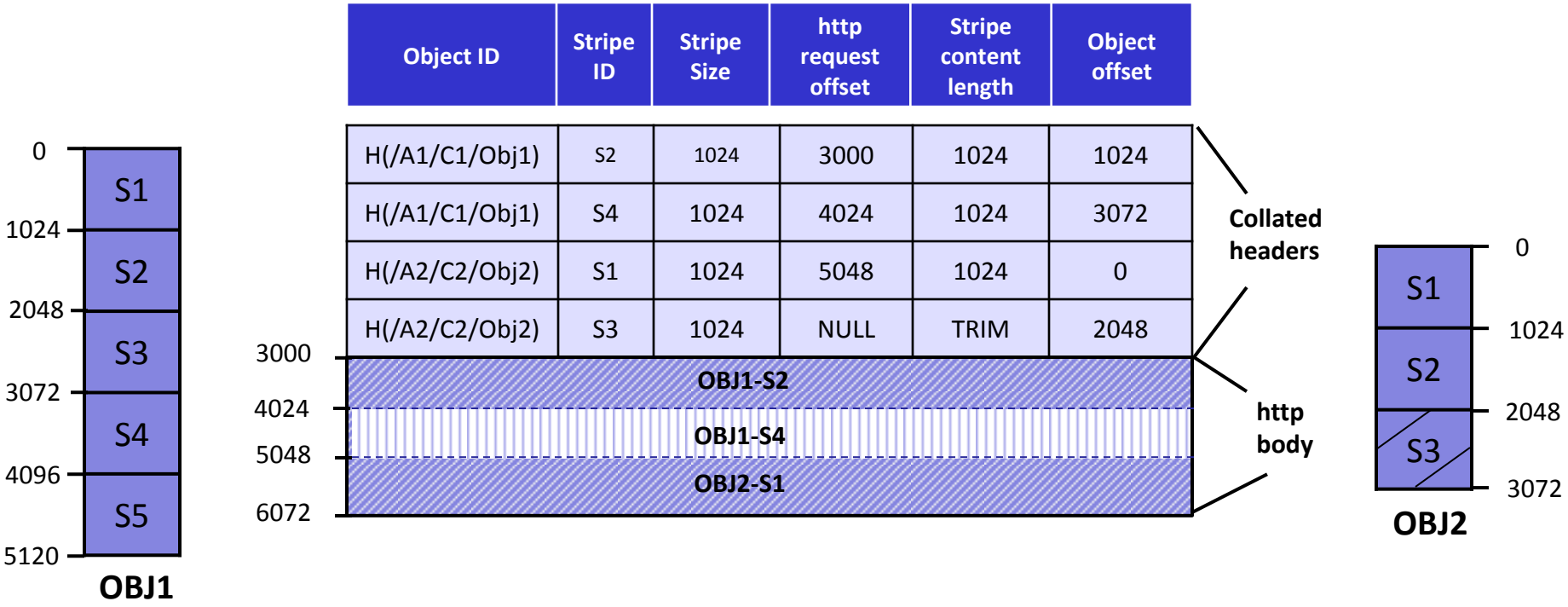
Now - Per-object extended attributes

Content-Type	Content-Length	Etag (fingerprint)	Creation time	Path (eg: /A/C/obj)

Proposed - Per-stripe extended attributes

Stripe-ID	Content-Type	Content - Length	Etag per stripe (fingerprint)	Creation time	Path (eg: /A/C/obj)

Prerequisite - HTTP PUT Request



- PUT request from proxy to the object server
- HTTP Header represents list of stripes to store in the sub-object
- HTTP body contains the corresponding data of the stripes
- Stripe content length <= stripe size

Prerequisite - HTTP PUT Response

Object ID	Stripe ID	Stripe delta size	Stripe Max offset	Status
H(/A1/C1/Obj1)	S2	1024	2047	"New Write"
H(/A1/C1/Obj1)	S4	0	4095	"Updated"
H(/A2/C2/Obj2)	S1	1024	1023	"New Write"
H(/A2/C2/Obj1)	S3	-1024	2047	"Trimmed"

Collated http response

- PUT response from object server to proxy
- HTTP Header represents list of stripes added, updated or trimmed in the sub-object

Description of header fields and derived values

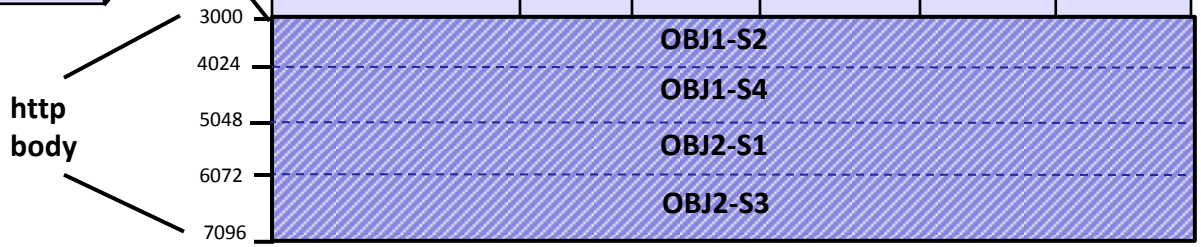
- **Status**: "new write", "updated", "trimmed", "replicated"
- **Stripe Max Offset:** is the max offset of stripe within the sub-object
- **Object size** = MAX(existing object-size, MAX(stripe-max-offset))
- **Stripe delta size:** effective data written to the disk
- **Object on-disk size** = SUM(existing object on-disk size, SUM(Stripe delta size))

Approach 1: Read/GET Request and Response Illustration

Object ID	Stripe ID	Stripe size	Stripe content length	Object offset
H(/A1/C1/Obj1)	S2	1024	1024	1024
H(/A1/C1/Obj1)	S4	1024	1024	3072
H(/A2/C2/Obj2)	S1	1024	1024	0
H(/A2/C2/Obj2)	S3	1024	1024	2048

Collated headers

Object ID	Stripe ID	Stripe Size	http response offset	Stripe content length	Object offset
H(/A1/C1/Obj1)	S2	1024	3000	1024	1024
H(/A1/C1/Obj1)	S4	1024	4024	1024	3072
H(/A2/C2/Obj2)	S1	1024	5048	1024	0
H(/A2/C2/Obj2)	S3	1024	6072	1024	2048



- GET request from proxy to the object server
- HTTP Header represents list of stripes to fetch from the sub-object

- GET response from object server to proxy
- HTTP Header represents list of stripes fetched from the sub-object
- HTTP Body contains the data of corresponding stripes

Miscellaneous Changes

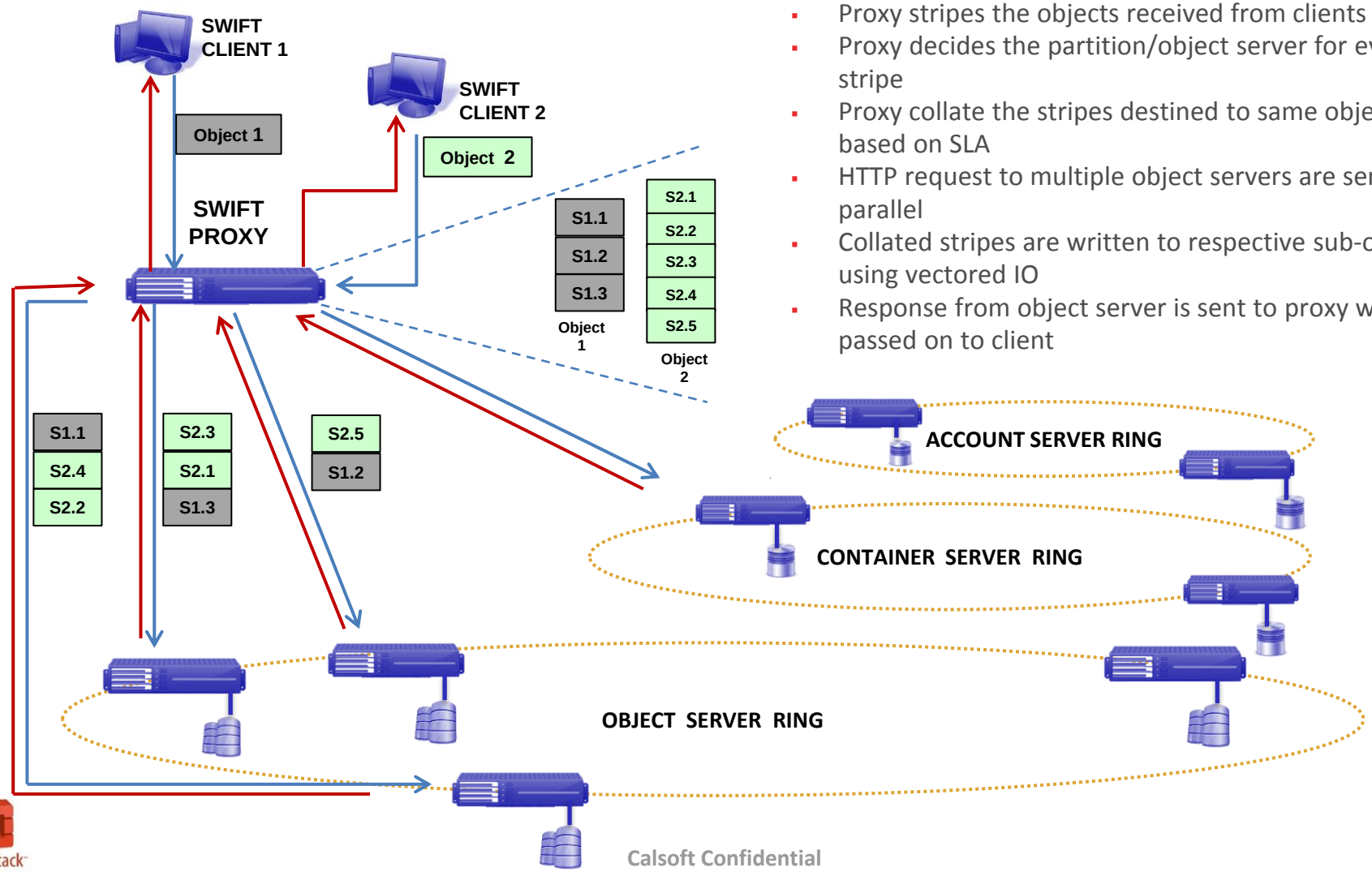
- Avoiding extra write I/O by using fingerprint(MD5 Sum)
  - Object server calculates the fingerprint for every stripe and compares it with fingerprint stored as extended attribute of the sub-object
  - If fingerprint matches, discard the stripe write
- Services like replicator, auditor, updater are also impacted to perform action upon stripes instead of objects
  - Along with objects, partition also stores a hash table
  - Hash table stores fingerprints for each object in the partition
  - Replicator/auditor/updater works by comparing entries in hash table
  - Introducing object striping, requires per stripe fingerprint to be stored in the hash table
  - Consequently the services would take decisions based on changes at stripe granularity

Approach – I: Client Unaware Striping Striping and Collation @ Proxy

- Object striping and its collation
  - Transparent to the client
  - Performed at proxy server
  - Proxy collates multiple stripes of multiple objects destined to same object server in single GET/PUT request
  - The stripes being collated can be sourced from different clients
- Collation criteria
  - Collation is based on service level agreement (SLA) as follows:
    - Timeout based
    - Size based
- Proxy decides partition and hence object server based on stripe-ID-hash

Note: The stripe size for all the objects is same and is configured at installation

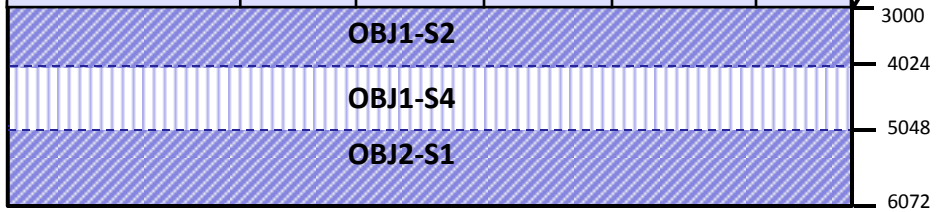
Approach – I
PUT Operation



- Proxy stripes the objects received from clients
- Proxy decides the partition/object server for every stripe
- Proxy collate the stripes destined to same object server based on SLA
- HTTP request to multiple object servers are sent in parallel
- Collated stripes are written to respective sub-objects using vectored IO
- Response from object server is sent to proxy which is passed on to client

Approach 1: Write/PUT Request and Response Illustration

Object path	Stripe ID	Stripe Size	http request offset	Stripe content length	Object offset
H(/A1/C1/Obj1)	S2	1024	3000	1024	1024
H(/A1/C1/Obj1)	S4	1024	4024	1024	3072
H(/A2/C2/Obj2)	S1	1024	5048	1024	0
H(/A2/C2/Obj2)	S3	1024	NULL	TRIM	2048



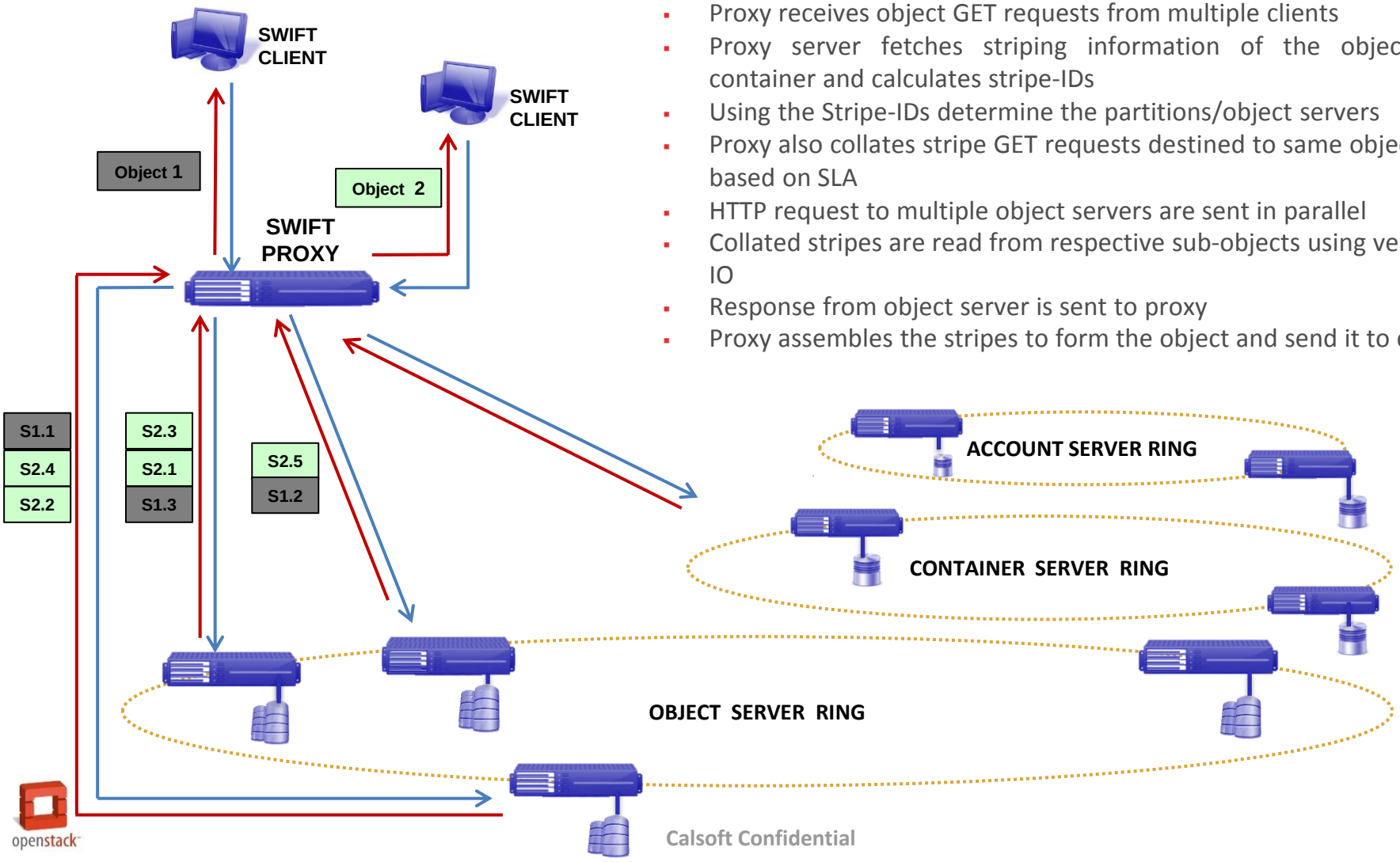
Collated headers

Object ID	Stripe ID	Stripe delta size	Stripe Max offset	Status
H(/A1/C1/Obj1)	S2	1024	2047	"New Write"
H(/A1/C1/Obj1)	S4	0	4095	"Updated"
H(/A2/C2/Obj2)	S1	1024	1023	"New Write"
H(/A2/C2/Obj1)	S3	-1024	2047	"Trimmed"

- **Scenario:** when client1 is writing obj1 and client 2 is writing obj2
- At proxy, the obj1 and obj2 are striped based on configured stripe size
- The stripes destined for same object server are collated by proxy in single HTTP PUT request
- Figure above shows S2, S4 of obj1 and S1, S3 of obj2 are clubbed together

- HTTP PUT response returns the status as "New Write", "Updated" or "Trimmed" for stripes to be added, modified or de-allocated respectively
- Also Stripe delta size for each stripe is sent, using which on-disk size of object is calculated by proxy

Approach – I
GET Operation



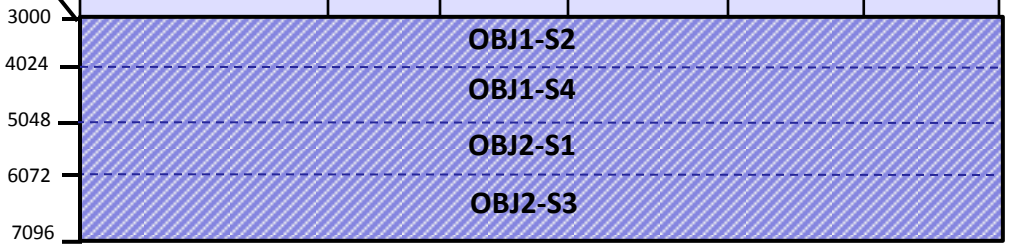
- Proxy receives object GET requests from multiple clients
- Proxy server fetches striping information of the objects from container and calculates stripe-IDs
- Using the Stripe-IDs determine the partitions/object servers
- Proxy also collates stripe GET requests destined to same object server based on SLA
- HTTP request to multiple object servers are sent in parallel
- Collated stripes are read from respective sub-objects using vectored IO
- Response from object server is sent to proxy
- Proxy assembles the stripes to form the object and send it to client

Approach 1: Read/GET Request and Response Illustration

Object ID	Stripe ID	Stripe size	Stripe content length	Object offset
H(/A1/C1/Obj1)	S2	1024	1024	1024
H(/A1/C1/Obj1)	S4	1024	1024	3072
H(/A2/C2/Obj2)	S1	1024	1024	0
H(/A2/C2/Obj2)	S3	1024	1024	2048

Collated headers

Object ID	Stripe ID	Stripe Size	http response offset	Stripe content length	Object offset
H(/A1/C1/Obj1)	S2	1024	3000	1024	1024
H(/A1/C1/Obj1)	S4	1024	4024	1024	3072
H(/A2/C2/Obj2)	S1	1024	5048	1024	0
H(/A2/C2/Obj2)	S3	1024	6072	1024	2048



- **Scenario:** The client1 sends read request for obj1 and client2 for obj2, to proxy
- Proxy fetches object-size from container
- Proxy generate stripe-ids and form new URLs to locate object servers and partitions
- Proxy collate read request for stripes which lie on same object server
- As shown in Figure above, stripe read request for S2, S4 of obj1 and S1, S3 of obj2 are clubbed together

- Object server sends the stripes requested
- Proxy again collates stripes of an obj1 came from different object servers and send the obj1 requested by the client1
- Similarly obj2 is sent to client2 after collating stripes by proxy

Approach – I : Pros and Cons and Miscellaneous Changes

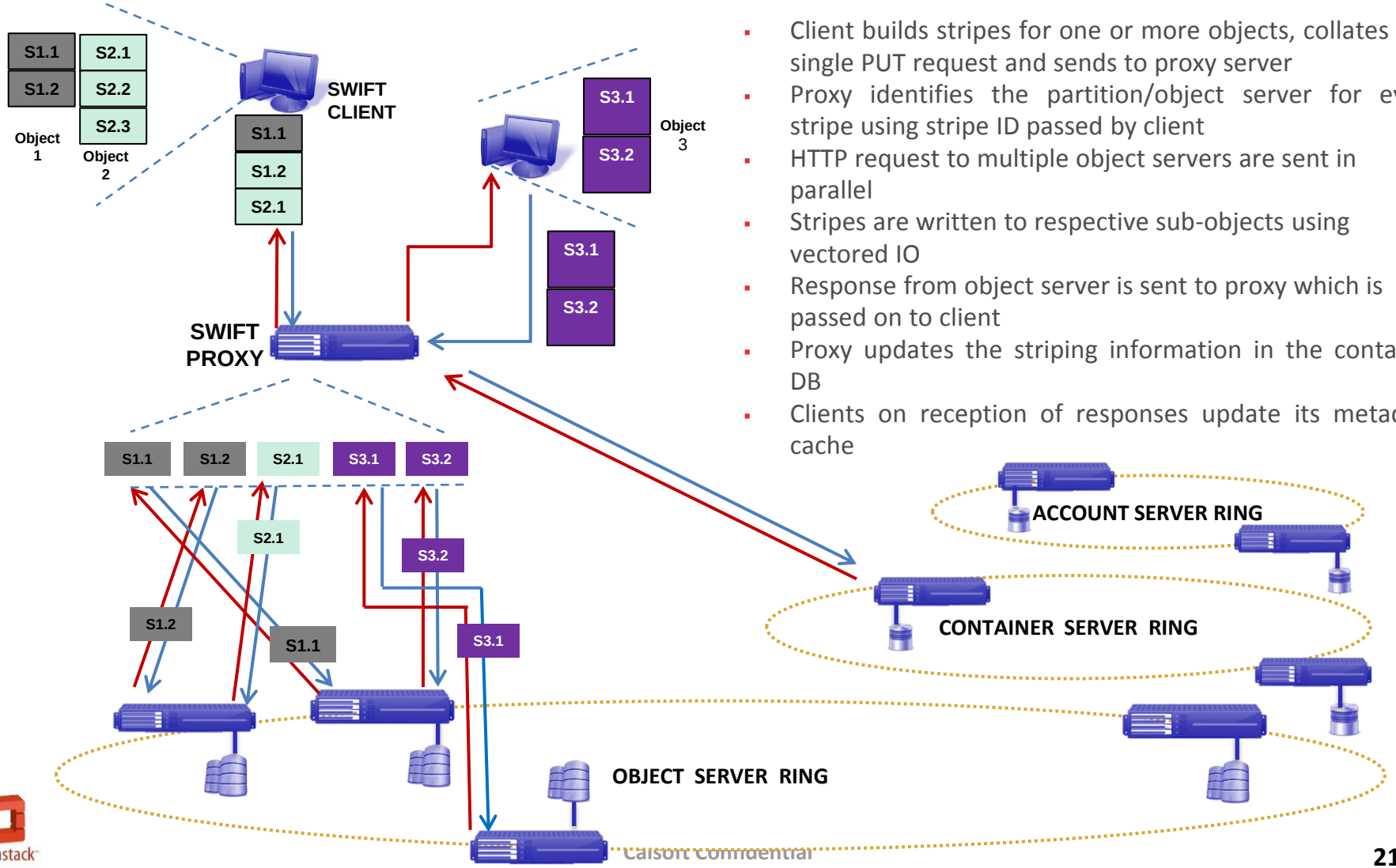
- Pros
  - Modifications are confined to swift server
  - Increase in effectiveness of Vectored IO at object server as proxy collates requests from multiple clients
- Cons
  - Proxy can be bottleneck
  - Collation of stripes of objects from multiple clients, leads to synchronization issue

- Miscellaneous changes
  - This approach needs modification in services namely swift-container-server, swift-object-server and swift-proxy-server
  - Replicator, Auditor and Updater services also requires change to operate over stripes instead of objects

Approach – II : Client Aware Striping - Striping and Collation @ Client

- Client is aware of stripes
  - Striping and collation is done at client side
  - Proxy acts more as pass-through
  - Metadata cache on the client holds the striping information same as in container DB
  - Listing operation on container yields striping information to client
  - Striping information is piggy-backed in GET Response
- Metadata cache of client contains following attributes:
  - object name, stripe size, object size, object-on-disk-size, object-version-delta-size
- Stripe ID is generated by client for both GET/PUT requests
- Stripe size of an object
  - May vary across different objects but is same within the object, so it is stored in container DB per object
  - Decided by client during first PUT (based on parameters like network bandwidth, object size, etc.)
  - Immutable there onwards

Approach – II
PUT Operation



- Client builds stripes for one or more objects, collates into single PUT request and sends to proxy server
- Proxy identifies the partition/object server for every stripe using stripe ID passed by client
- HTTP request to multiple object servers are sent in parallel
- Stripes are written to respective sub-objects using vectored IO
- Response from object server is sent to proxy which is passed on to client
- Proxy updates the striping information in the container DB
- Clients on reception of responses update its metadata cache

Approach – II : Write/PUT Request Illustration

Object path	Stripe ID	Stripe Size	http request offset	Stripe content length	Object offset
/A1/C2/ Obj1	S2	1024	3000	1024	1024
/A1/C2/ Obj1	S1	1024	4024	1024	0
/A1/C2/ Obj2	S1	1536	5048	1536	0
/A1/C2/ Obj2	S3	1536	6584	1536	3072

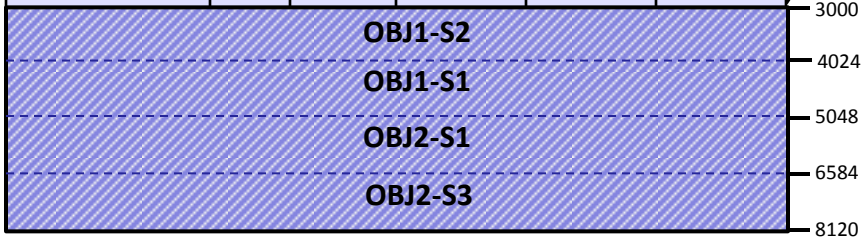


Fig 1. http PUT request from client to proxy

Collated headers

Object ID	Stripe ID	Stripe Size	http request offset	Stripe content length	Object offset
H(/A1/C2/ Obj1)	S2	1024	3000	1024	1024
H(/A1/C2/ Obj2)	S1	1536	4024	1536	0
H(/A1/C2/ Obj2)	S3	1536	5560	1536	3072

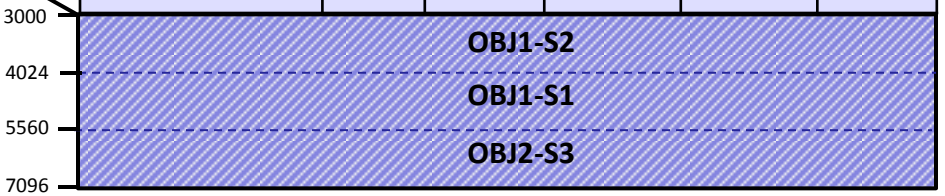


Fig 2. http PUT request from proxy to one of the object servers

- **Scenario:** when Swift Client wants to write/modify some stripes of obj1 and obj2
- stripe size for obj1 is 1024 bytes and obj2 is 1536 bytes
- Swift client forms a single request and clubs information of modified data of stripes in it and send the write/PUT request to proxy
- Proxy based on stripe-id identifies the object servers and forms a new write/PUT request for stripes. In Fig 2, S2 of obj1 and S1,S3 of obj2 reside on same object server

Approach – II : Write/PUT Response Illustration

Object ID	Stripe ID	Stripe delta size	Stripe Max offset	Status
H(/A1/C2//Obj1)	S2	1024	2047	“New Write”
H(/A1/C2/Obj2)	S1	1536	1535	“New Write”
H(/A1/C2/Obj2)	S3	1536	4607	“New Write”

Fig 1. http PUT response from one of the object servers to proxy

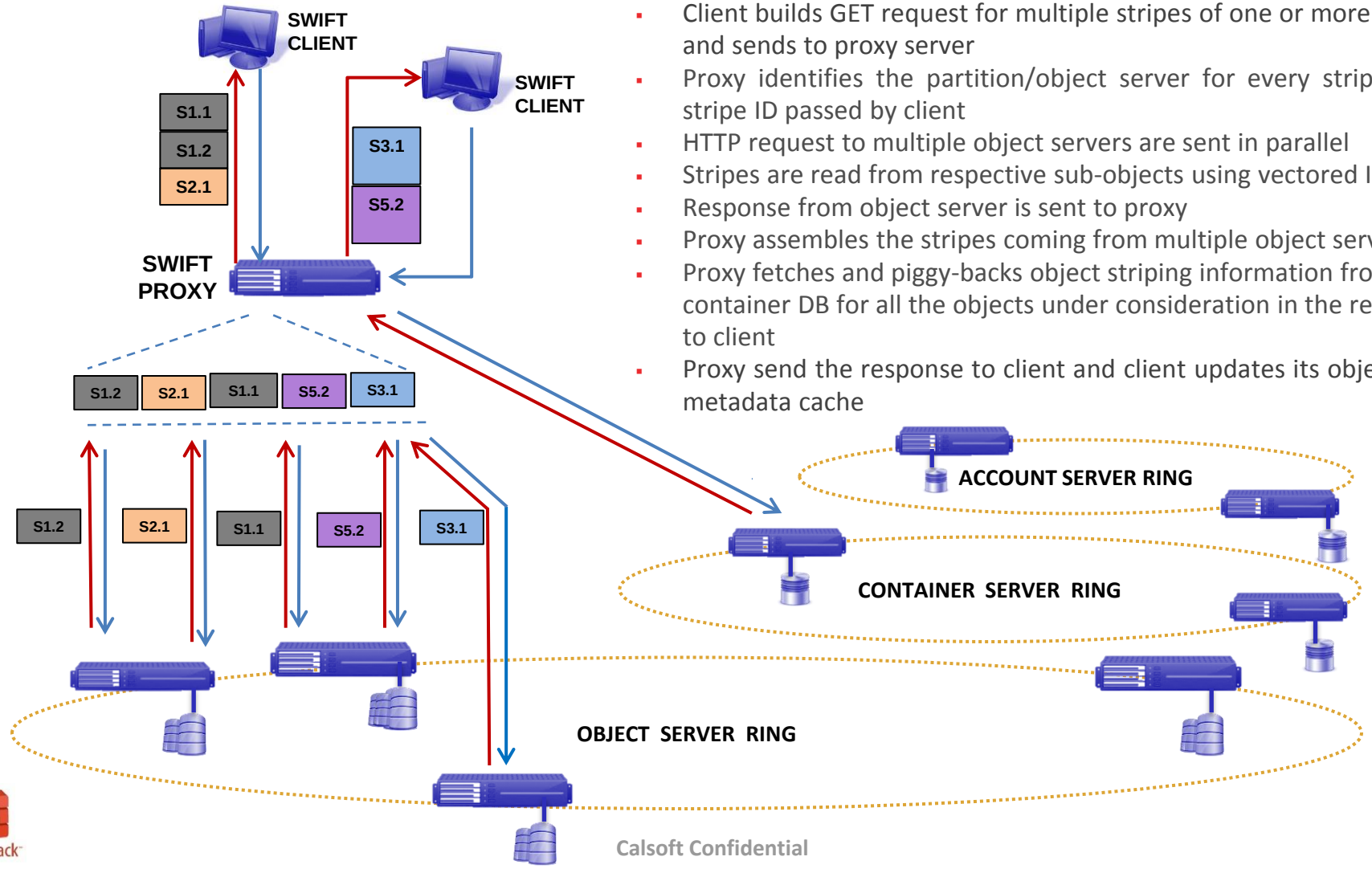
Collated headers

Object path	Stripe ID	Stripe Size	Stripe delta size	Stripe Max Offset	Status
/A1/C2/Obj1	S2	1024	1024	2047	“New Write”
/A1/C2/Obj2	S1	1536	1536	1535	“New Write”
/A1/C2/Obj2	S3	1536	1536	4607	“New Write”
/A1/C2/ Obj1	S1	1024	1024	1024	“New Write”

Fig 2. http PUT response from proxy to client

- **Scenario:** when Swift Client wants to write/modify some stripes of obj1 and obj2
- Object server write/modify the stripes in sparse object and sends the response to proxy with status=”new write”. Refer Fig 1.
- Proxy in turn sends the response to client with same status. Refer Fig 2.

Approach – II
GET Operation



- Client builds GET request for multiple stripes of one or more objects and sends to proxy server
- Proxy identifies the partition/object server for every stripe using stripe ID passed by client
- HTTP request to multiple object servers are sent in parallel
- Stripes are read from respective sub-objects using vectored IO
- Response from object server is sent to proxy
- Proxy assembles the stripes coming from multiple object servers
- Proxy fetches and piggy-backs object striping information from container DB for all the objects under consideration in the response to client
- Proxy send the response to client and client updates its objects and metadata cache

Approach – II : READ/GET Request Illustration

Object Path	Stripe ID	Stripe Size	Stripe content length	Object offset
/A1/C2/Obj1	S2	1024	1024	1024
/A1/C2/Obj1	S1	1024	1024	0
/A1/C2/Obj2	S1	1536	1536	0
/A1/C2/Obj2	S3	1536	1536	3072

Fig 1. http GET request from client to proxy

Collated headers

Object ID	Stripe ID	Stripe Size	Stripe content length	Object offset
H(/A1/C2/Obj1)	S2	1024	1024	1024
H(/A1/C2/Obj2)	S1	1536	1536	0
H(/A1/C2/Obj2)	S3	1536	1536	3072

Fig 2. http GET request from proxy to one of the object servers

- **Scenario:** when swift client wants to read stripes belonging to different objects
- Fig 1 shows http GET request to read S2,S1 of obj1 and S1,S3 of obj2
- The client collates multiple stripe read requests and send it to proxy
- Proxy based on stripe-id identifies the location of object servers and forms a new GET request for stripes which reside on same object server. In Fig 2. S2 of obj1 and S1,S3 of obj2 reside on same object server

Approach – II : READ/GET Response Illustration

Object ID	Stripe ID	Stripe Size	http response offset	Stripe content length	Object offset
H(/A1/C2/Obj1)	S2	1024	3000	1024	1024
H(/A1/C2/Obj2)	S1	1536	4024	1536	0
H(/A1/C2/Obj2)	S3	1536	5048	1536	3072

Collated headers

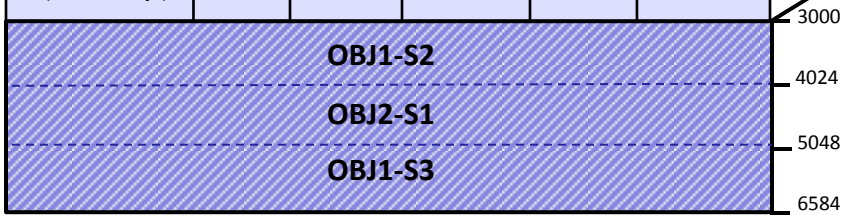


Fig 1. http GET response from one of the object servers to proxy

- **Scenario:** when swift client wants to read stripes belonging to different objects
- Object server reads the stripes from sparse objects and sends the response with data read to proxy. Refer Fig 1.
- Proxy in turn sends the data read back to client. It also piggy-backs object striping information. Refer Fig 2.

Piggy-Back headers

Object Path	Stripe Size	Object size	Object on-disk size	Object version delta size
/A1/C2/Obj1	1024	4096	3072	NO_SIZE
/A1/C2/Obj2	1536	4608	3072	NO_SIZE

Collated headers

Object Path	Stripe ID	Stripe Size	http response offset	Stripe content length	Object offset
/A1/C2/Obj1	S2	1024	3000	1024	1024
/A1/C2/Obj2	S1	1536	4024	1536	0
/A1/C2/Obj2	S3	1536	5048	1536	3072
/A1/C2/Obj1	S1	1024	6584	1024	0

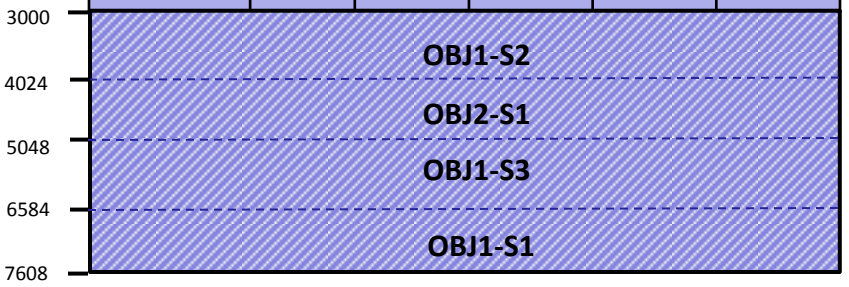


Fig 2. http GET response from proxy to client

Approach – II : Pros and Cons and Miscellaneous Changes

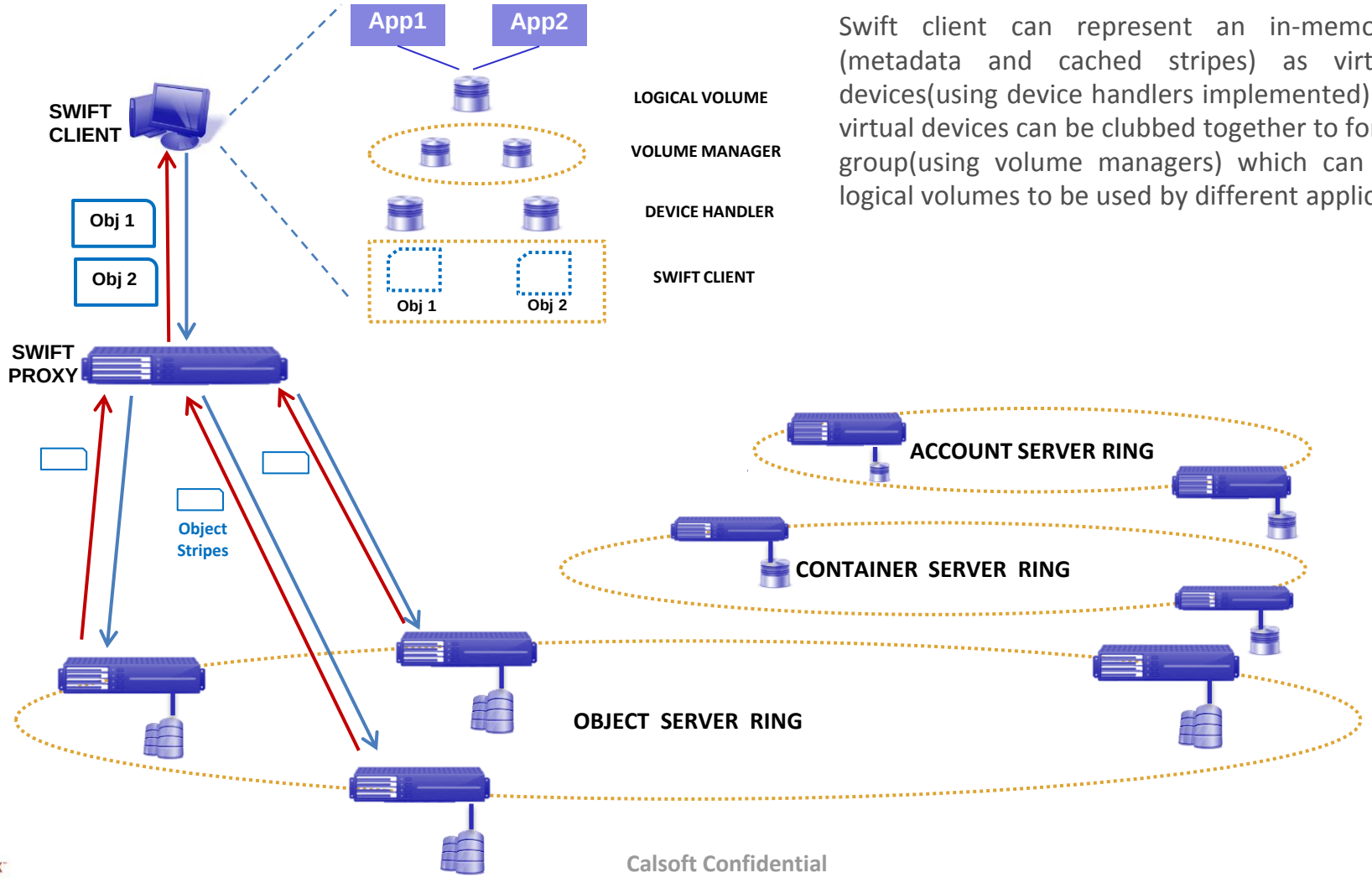
- Pros
  - Proxy is offloaded from the striping and collation task
  - Utilizing client resources more
  - Reducing network bandwidth usage from client end to object server end
- Cons
  - Proxy has to assemble all the stripes coming from multiple object servers to build response for client

- Miscellaneous changes
  - This approach needs modification in services namely swift-container-server, swift-object-server, swift-proxy-server and swift-client
  - Replicator, Auditor and Updater services also requires change to operate over stripes instead of objects

Use Case – I : Exporting Object Storage as Volume

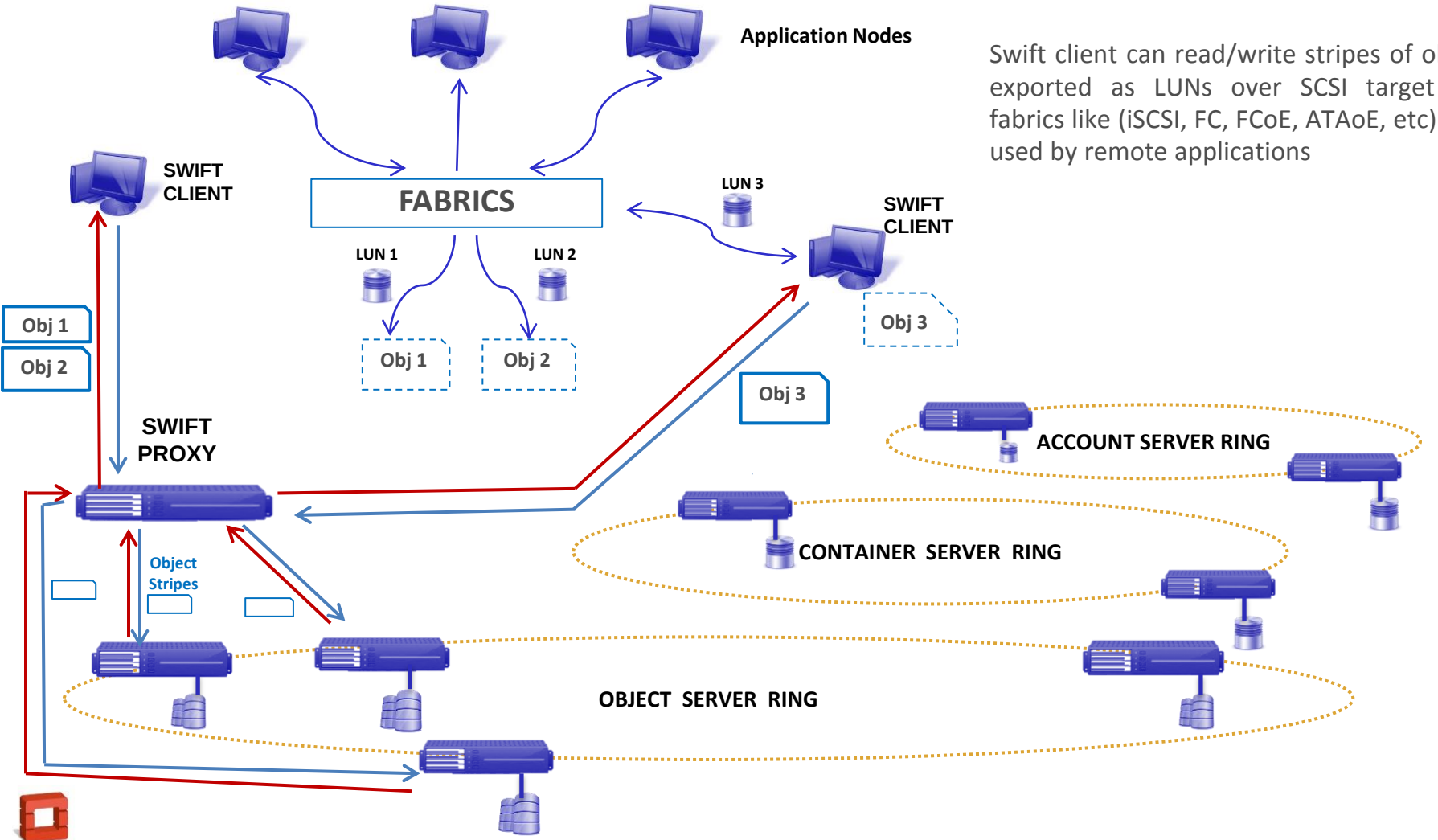
- Overview
  - Use swift to export object storage to be utilized by applications as block storage
  - In-memory representation of set of large objects can be exported by swift client as logical units/ virtual block devices to applications
  - The IO performance has no severe impact as the large objects are striped transparently by the swift client. Stripes of any object that needs to read or written, are transmitted over network which reduces consumption of network bandwidth
  - By building volume layer over objects, one can implement various other features and functionalities like de-duplication, compression, snapshots

Use Case – I : Volume Stack Over Object Storage



Swift client can represent an in-memory object (metadata and cached stripes) as virtual block devices(using device handlers implemented) and those virtual devices can be clubbed together to form volume group(using volume managers) which can carve out logical volumes to be used by different applications

Use Case – I : Exporting Object Storage over Fabrics



Swift client can read/write stripes of objects exported as LUNs over SCSI target with fabrics like (iSCSI, FC, FCoE, ATAoE, etc) to be used by remote applications

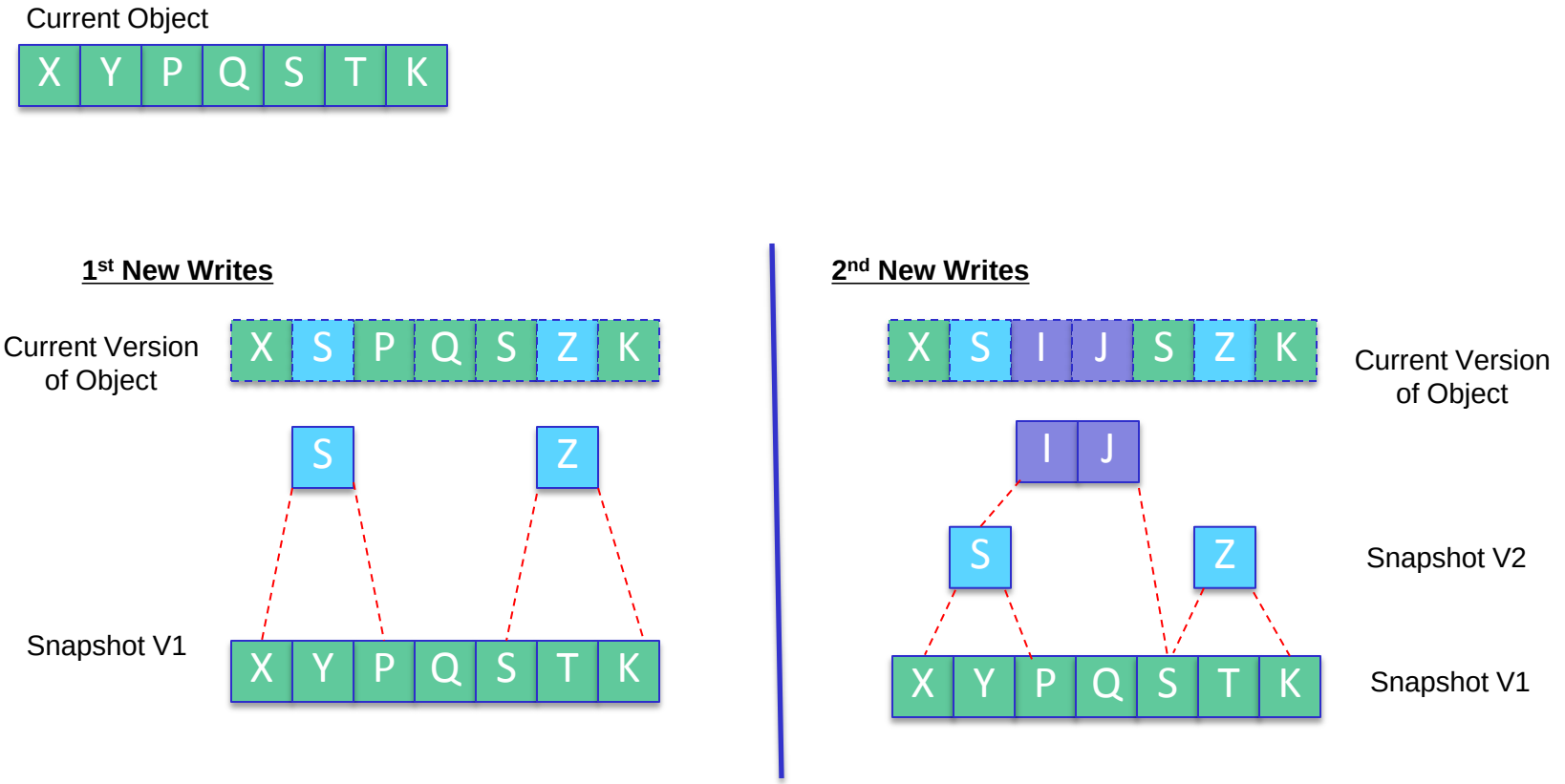
Use Case – II : Redirect-On-Write Snapshot

- Overview
  - On demand snapshots creation request can be initiated from swift client or by applications on swift client by sending requests to container server to take a snapshot
  - With each snapshot, a new version of object is created
  - As container DB stores striping information of objects, it would also store size of delta i.e. total size of stripes which got updated, added and/or trimmed

Extending the view:

- If both the Use-case I and Use-case II are seen together, snapshot providers (e.g. VSS providers) can be implemented at SCSI target end, resulting into single call to swift to take a snapshot

Use Case – II : Redirect-On-Write Snapshot



Use Case – II : Operations

- Swift client can request, on demand, snapshot creation
- Snapshot is created as a new object version in container DB and respective files would get created on the object servers as PUT requests for various stripes to be added, updated or trimmed as received following the snapshot creation
- Current/active object version is piggy backed in client response
- The client has to explicitly do listing operation over containers to list out all the snapshot. During list request, the response would contain all the metadata of objects for all the snapshots
- The client can also request for stripes of specific snapshot in the GET request

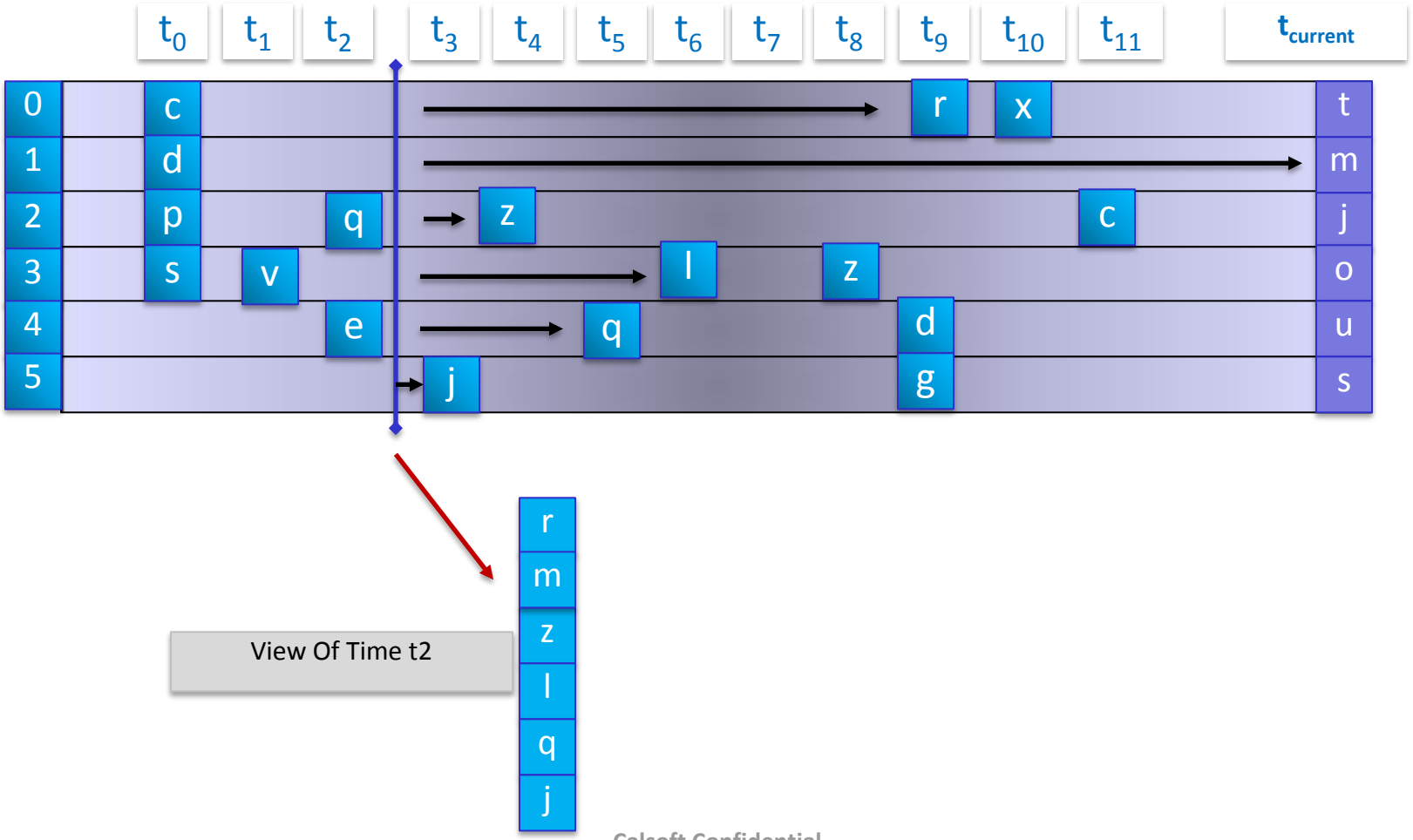
Use Case – III : Continuous Data Protection (CDP)

- Overview
  - Swift supports object versioning
  - When versioning is enabled for any container, for each new write on objects residing in object server under the container, a new version of that object is created. Since today object is nothing but the entire file in swift, storage is consumed significantly to store version of file(object)
  - With object striping in swift, object version would store only changed stripes of objects resulting in less storage consumption. Thus we can have better solution of CDP even for storing versions of large objects
  - As container DB stores information of objects, it would also store size of delta i.e. total size of stripes which got changed

Extending the view:

- If both the Use-case I and Use-case III are seen together, near CDP can be achieved using object versioning at stripe granularity

Use Case – III : Continuous Data Protection (CDP)



Use Case – III : Operations

- As per current implementation before object versioning, the client would send a request to create a new container to store object versions and link it with main container
- Now on every new write, newer version of object is created and would be part of linked container
- In GET request, the client has to mention the version of the object. Based on version number, proxy collaboratively working with container, determines the location of the set of version stripes and requests in parallel to get the stripes from object servers
- If no object version is mentioned, the latest state of the object is fetched

Use Case – IV : TRIM/UNMAP

- Overview
  - Due to sparse object support, TRIM/UNMAP command can be used to erase stripes, thereby reducing used storage space
  - The extended attributes for trimmed stripes are as follows:

stripe-ID	Content-Type= **"trimmed"**	Content-Length	Etag per stripe (md5sum)	Creation time	path (eg: /A/C/obj)

  - When swift client wants to read such sparse object, swift server sends information of which stripes are trimmed, in http header of response. Similar status can be sent even if stripe does not exist
  - Swift client then can allocate zeroed pages for those trimmed stripes(file holes) and then send to application. So no data is transferred over network

Use Case – IV : Write/PUT Operations

- TRIM as a write operation introduces holes into the object
- Trim operation is initiated by swift client to de-allocate the stripes
- TRIM request is passed to respective object servers. The object server would de-allocate range of blocks representing the stripes. And the file/sub-object hole is created
- The corresponding entries in the extended attributes are updated as “trimmed”
- In response, along with ACK the object servers send the number of bytes removed(stripe de-allocated information) of stripe in “Stripe delta size” and the “Stripe Max offset”. In this case the “Stripe delta size” is negative, to represent the stripe has been trimmed
- The on-disk size of object is updated in container by subtracting those many bytes

Use Case – IV : Write/PUT Request illustration

Object path	Stripe ID	Stripe Size	http request offset	Stripe content length	Object offset
/A1/C1/Obj1	S2	1024	NULL	TRIM	1024
/A1/C1/obj1	S1	1024	NULL	TRIM	0
/A1/C2/obj2	S1	1536	NULL	TRIM	0
/A1/C2/obj2	S3	1536	NULL	TRIM	3072

Fig 1. http PUT request from client to proxy

Collated headers

Object path	Stripe ID	Stripe Size	http request offset	Stripe content length	Object offset
H(/A1/C1/ Obj1)	S2	1024	NULL	TRIM	1024
H(/A1/C1/Obj1)	S1	1024	NULL	TRIM	0
H(/A1/C2/ Obj2)	S1	1536	NULL	TRIM	0

Fig 2. http PUT request from proxy to one of the object servers

- **Scenario:** Client wants to trim stripes S2 and S1 of obj1 and stripes S1 and S3 of obj2
- Client forms a single http request to club multiple trim requests on stripes and send it to proxy. In the request, it sends the stripe content length=”TRIM” to notify trim operation. Refer Fig 1.
- Based on stripe-id, proxy identifies the object servers and club requests destined for same object server. Fig 2. shows that trim for S2 and S1 of obj1; and S1 of obj2 is sent in single request

Use Case – IV : Write/PUT Response illustration

Object ID	Stripe ID	Stripe delta size	Stripe Max offset	Status
H(/A1/C1/Obj1)	S2	-1024	1023	"TRIMMED"
H(/A1/C1Obj1)	S1	-1024	0	"TRIMMED"
H(/A1/C2/Obj2)	S1	-1536	0	"TRIMMED"

Fig 1. http PUT response from object servers to proxy

Collated headers

Object path	Stripe ID	Stripe Size	Stripe delta size	Stripe Max offset	Status
/A1/C1/ Obj1	S2	1024	-1024	1023	"TRIMMED"
/A1/C1/ Obj1	S1	1024	-1024	0	"TRIMMED"
/A1/C2/ Obj2	S1	1536	-1536	0	"TRIMMED"
/A1/C2/ Obj2	S3	1536	-1536	3071	"TRIMMED"

Fig 2. http PUT response from proxy to client

- **Scenario:** Client wants to trim stripes S2 and S1 of obj1 and stripes S1 and S3 of obj2
- Object server de-allocate the stripes, update the extended attributes and sends the response to proxy with status="trimmed" and Stripe delta size=<bytes de-allocated>. Refer Fig 1.
- Proxy sends the response back to swift client with same status. Refer Fig 2.

Use Case – IV : Read Operations

- Since a client is initially unaware of the striping information or in case of the cached striping information of a client is stale, it may request for a non-existing/trimmed stripe
- The request would be forwarded to object server by proxy
- Object servers would be able to identify either the stripe was never written or got trimmed (marked as trimmed) from the extended attributed pertaining to the requested stripe
- It would send this information to swift client via proxy. No zeroed stripes are passed over network, instead only relevant attributes in the http response would able to identify that the stripe got trimmed
- Swift client would prepare zeroed pages for trimmed stripes and send it to applications sitting over swift client

Use Case – IV : READ/GET Request illustration

Object Path	Stripe ID	Stripe Size	Stripe content length	Object offset
/A1/C2/Obj1	S2	1024	1024	1024
/A1/C2/Obj1	S1	1024	1024	0
/A1/C2/Obj2	S1	1536	1536	0
/A1/C2/Obj2	S3	1536	1536	4608

Fig 1. http GET request from client to proxy

Collated headers

Object path	Stripe ID	Stripe Size	Stripe content length	Object offset
H(/A1/C2/obj1)	S2	1024	1024	1024
H(/A1/C2/obj2)	S1	1536	1536	0
H(/A1/C2/obj2)	S3	1536	1536	4608

Fig 2. http GET request from proxy to one of the object servers

- **Scenario**: when client want to read stripes on behalf of applications
- The client collates multiple stripe read requests and send it to proxy. Refer Fig1.
- Proxy based on stripe-id identifies the location of object servers and forms a new GET request for stripes which reside on same object server. In Fig2. S2 of obj1; and S3 and S1 of obj2 reside on same object server
- For GET response, refer the next slide

Use Case – IV : READ/GET Response illustration

Object ID	Stripe ID	Stripe Size	http response offset	Stripe content length	Object offset
H(/A1/C2/Obj1)	S2	1024	0	TRIM	1024
H(/A1/C2/Obj2)	S1	1536	0	TRIM	0
H(/A1/C2/Obj2)	S3	1536	0	TRIM	4608

Fig 1. http GET response from one of the object servers to proxy

Collated headers

Object Path	Stripe Size	Object size	Object on-disk size	Object version delta size
/A1/C2/Obj1	1024	4096	2048	NO_SIZE
/A1/C2/Obj2	1536	4608	1536	NO_SIZE

Piggy-Back headers

Object Path	Stripe ID	Stripe Size	http response offset	Stripe content length	Object offset
/A1/C2/Obj1	S2	1024	0	TRIM	1024
/A1/C2/Obj2	S1	1536	0	TRIM	0
/A1/C2/Obj2	S3	1536	0	TRIM	4608
/A1/C2/Obj1	S1	1024	0	TRIM	0

Fig 2. http GET response from proxy to client

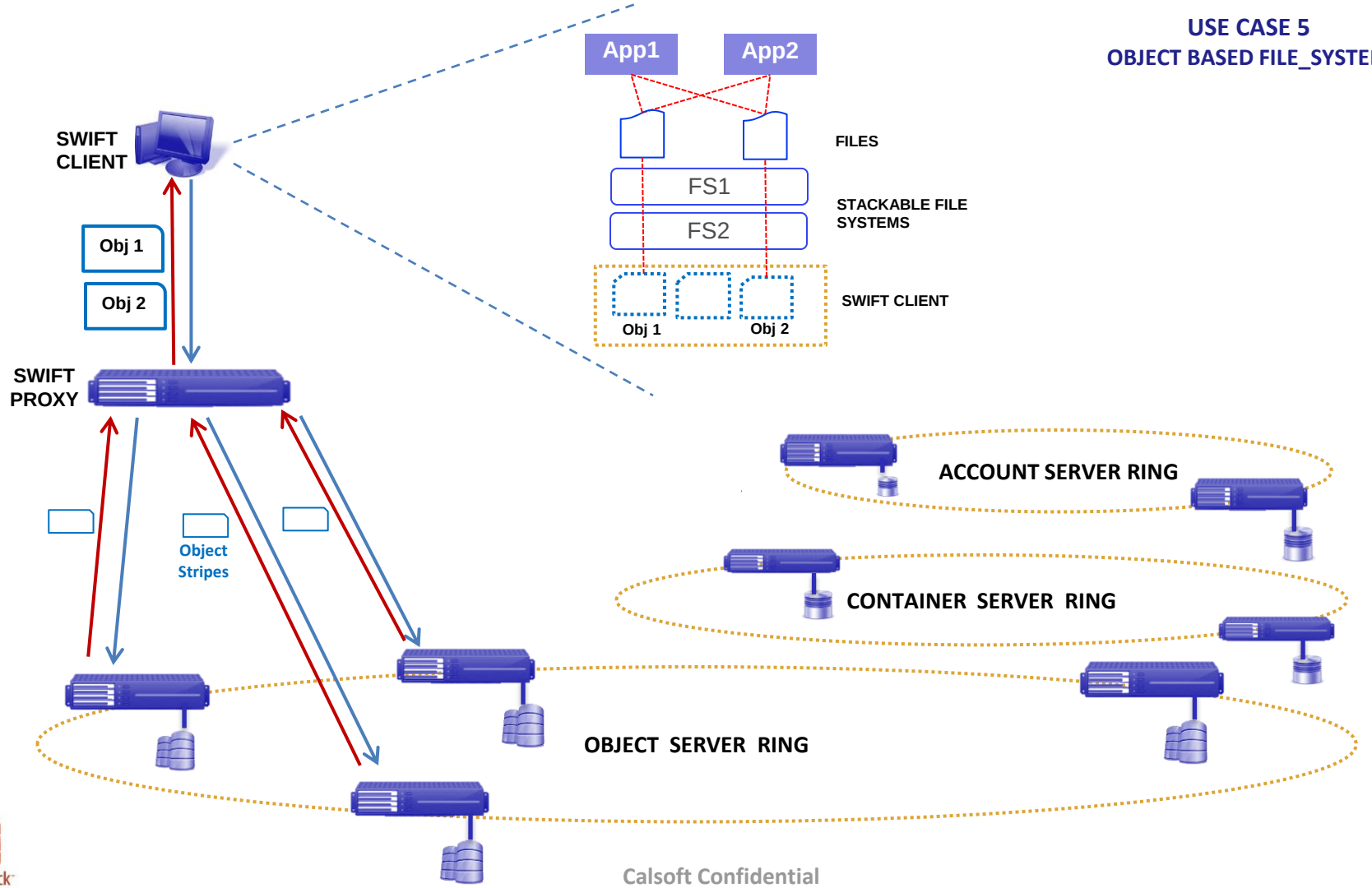
- **Scenario**: when client want to read stripes on behalf of applications
- Object server learns that the stripes requested are trimmed, by checking extended attributes of sparse object. Object server forms a response with stripe content length=TRIM and sends it to proxy. Refer Fig 1.
- Proxy in turn sends the response to client with same status. It also piggy-backs the object striping information. Refer Fig 2.
- By reading the status, swift client comes to know that the stripes were trimmed. It creates zeroed pages and sends to application which initiated the read request

Use Case – V : Objects based file-system

- Overview
  - One can have file-system layer over objects. End users will perform IO on files which are internally mapped to one or more swift objects depending upon the size of the file at the swift client end

- Operations
  - A stackable file-system can be built over the objects which maps the files to objects
  - Known frameworks like FiST or FUSE(file-system in user-space) can be used to implement stackable file-systems
  - The file system could be pass through, just handling the mapping of files and objects Or could provide extended capabilities like encryption, compression, etc.
  - Now when IOs are encountered over files directly mapped to objects, only the stripe that is changed are written back by clients which reduces consumption of network bandwidth
  - Similarly only stripes of the objects corresponding to the file blocks being read are transmitted over network

Use Case – V : Objects based file-system



Enhancements

- Enhancements of approach 2:
  - The stripe size for client can be made different across different sessions depending on the client and network capabilities
  - Stripe size on the server could be constant but for clients, it can vary
  - Client based on network bandwidth available to access server nodes, can decide stripe size and can share stripe size with server nodes
  - Communication between server nodes(proxy and object server) can happen with some different stripe size
  - But requests and responses to/for clients will be addressed based on their stripe size
  - With this stripes in the sub-objects could be partially overwritten which can be handled by using the “stripe content length” header field
- Use erasure coding instead of multi-copy mirroring approach as of today

Future Scope

- Future Scope:
  - Proxy could be a bottleneck as it handles all the control and data requests from the swift clients
  - To overcome this bottleneck, a complete new design, where swift clients can communicate with object servers bypassing the proxy server can be thought over
  - In such a design, swift clients would only go to proxy once to get the access to object servers and the location of object within object server
  - Once client is aware, it can directly send http requests to object servers
  - Using such approach, there could be security issues which would need to be tackled by object servers but thought through well

Appendix

- HTTP headers in JSON format

Approach 1:

1)PUT request from proxy to object server

{"r0" : {"object_id" : "H(/A1/C1/Obj1)", "stripe_id" : "S2", "stripe_size" : 1024, "http_request_offset" : 3000, "stripe_content_length" : 1024, "object_offset" : 1024},
"r1" : {"object_id" : "H(/A1/C1/Obj1)", "stripe_id" : "S4", "stripe_size" : 1024, "http_request_offset" : 4024, "stripe_content_length" : 1024, "object_offset" : 3072},
"r2" : {"object_id" : "H(/A2/C2/Obj2)", "stripe_id" : "S1", "stripe_size" : 1024, "http_request_offset" : 5048, "stripe_content_length" : 1024, "object_offset" : 0},
"r3" : {"object_id" : "H(/A2/C2/Obj2)", "stripe_id" : "S3", "stripe_size" : 1024, "http_request_offset" : NULL, "stripe_content_length" : TRIM, "object_offset" : 2048}
}

2)PUT request response from object server to proxy

{"r0" : {"object_id" : "H(/A1/C1/Obj1)", "stripe_id" : "S2", "stripe_delta_size" : 1024, "stripe_max_offset" : 2047, "status" : "New Write"},
"r1" : {"object_id" : "H(/A1/C1/Obj1)", "stripe_id" : "S4", "stripe_delta_size" : 0, "stripe_max_offset" : 4095, "status" : "Updated"},
"r2" : {"object_id" : "H(/A2/C2/Obj2)", "stripe_id" : "S1", "stripe_delta_size" : 1024, "stripe_max_offset" : 1023, "status" : "New Write"},
"r3" : {"object_id" : "H(/A2/C2/Obj2)", "stripe_id" : "S3", "stripe_delta_size" : -1024, "stripe_max_offset" : 2047, "status" : "Trimmed"}
}

Appendix … Contd

- HTTP headers in JSON format

Approach 1:

3)HTTP GET request from proxy to object server -

{"r0" : {"object_id" : "H(/A1/C1/Obj1)", "stripe_id" : "S2", "stripe_size" : 1024, "stripe_content_length" : 1024, "object_offset" : 1024},
"r1" : {"object_id" : "H(/A1/C1/Obj1)", "stripe_id" : "S4", "stripe_size" : 1024, "stripe_content_length" : 1024, "object_offset" : 3072},
"r2" : {"object_id" : "H(/A2/C2/Obj2)", "stripe_id" : "S1", "stripe_size" : 1024, "stripe_content_length" : 1024, "object_offset" : 0},
"r3" : {"object_id" : "H(/A2/C2/Obj2)", "stripe_id" : "S3", "stripe_size" : 1024, "stripe_content_length" : 1024, "object_offset" : 2048}
}

4)HTTP GET response from object server to proxy -

{"r0" : {"object_id" : "H(/A1/C1/Obj1)", "stripe_id" : "S2", "stripe_size" : 1024, "http_response_offset" : 3000, "stripe_content_length" : 1024, "object_offset" : 1024},
"r1" : {"object_id" : "H(/A1/C1/Obj1)", "stripe_id" : "S4", "stripe_size" : 1024, "http_response_offset" : 4024, "stripe_content_length" : 1024, "object_offset" : 3072},
"r2" : {"object_id" : "H(/A2/C2/Obj2)", "stripe_id" : "S1", "stripe_size" : 1024, "http_response_offset" : 5048, "stripe_content_length" : 1024, "object_offset" : 0},
"r3" : {"object_id" : "H(/A2/C2/Obj2)", "stripe_id" : "S3", "stripe_size" : 1024, "http_response_offset" : 6072, "stripe_content_length" : 1024, "object_offset" : 2048}
}

Appendix … Contd

- HTTP headers in JSON format

Approach 2:

1)PUT request from client to proxy

{"r0" : {"object_path" : "H(/A1/C2/Obj1)", "stripe_id" : "S2", "stripe_size" : 1024, "http_request_offset" : 3000,
"stripe_content_length" : 1024, "object_offset" : 1024},
"r1" : {"object_path" : "H(/A1/C2/Obj1)", "stripe_id" : "S1", "stripe_size" : 1024, "http_request_offset" : 4024,
"stripe_content_length" : 1024, "object_offset" : 0},
"r2" : {"object_path" : "H(/A1/C2/Obj2)", "stripe_id" : "S1", "stripe_size" : 1536, "http_request_offset" : 5048,
"stripe_content_length" : 1536, "object_offset" : 0},
"r3" : {"object_path" : "H(/A1/C2/Obj2)", "stripe_id" : "S3", "stripe_size" : 1536, "http_request_offset" : 6584,
"stripe_content_length" : 1536, "object_offset" : 3072}
}

2)PUT request from proxy to object server

{"r0" : {"object_id" : "H(/A1/C2/Obj1)", "stripe_id" : "S2", "stripe_size" : 1024, "http_request_offset" : 3000,
"stripe_content_length" : 1024, "object_offset" : 1024},
"r1" : {"object_id" : "H(/A1/C2/Obj2)", "stripe_id" : "S3", "stripe_size" : 1536, "http_request_offset" : 4024,
"stripe_content_length" : 1536, "object_offset" : 0},
"r2" : {"object_id" : "H(/A1/C2/Obj2)", "stripe_id" : "S1", "stripe_size" : 1536, "http_request_offset" : 5560,
"stripe_content_length" : 1536, "object_offset" : 3072}
}

Appendix … Contd

- HTTP headers in JSON format

Approach 2:

3)PUT response from object server to proxy

{"r0" : {"object_id" : "H(/A1/C2/Obj1)", "stripe_id" : "S2", "stripe_delta_size" : 1024, "stripe_max_offset" : 2047, "status" : "New Write"},
"r1" : {"object_id" : "H(/A1/C2/Obj1)", "stripe_id" : "S1", "stripe_delta_size" : 1536, "stripe_max_offset" : 1535, "status" : "New Write"},
"r2" : {"object_id" : "H(/A1/C2/Obj2)", "stripe_id" : "S3", "stripe_delta_size" : 1536, "stripe_max_offset" : 4607, "status" : "New Write"}
}

4) PUT response from proxy to client

{"r0" : {"object_path" : "/A1/C2/Obj1", "stripe_id" : "S2", "stripe_size" : 1024, "stripe_delta_size" : 1024, "stripe_max_offset" : 2047, "status" : "New Write"},
"r1" : {"object_path" : "/A1/C2/Obj2", "stripe_id" : "S1", "stripe_size" : 1536, "stripe_delta_size" : 1536, "stripe_max_offset" : 1535, "status" : "New Write"},
"r2" : {"object_path" : "/A1/C2/Obj2", "stripe_id" : "S3", "stripe_size" : 1536, "stripe_delta_size" : 1536, "stripe_max_offset" : 4607, "status" : "New Write"},
"r3" : {"object_path" : "/A1/C2/Obj1", "stripe_id" : "S1", "stripe_size" : 1024, "stripe_delta_size" : 1024, "stripe_max_offset" : 1024, "status" : "New Write"}
}

Appendix … Contd

- HTTP headers in JSON format

Approach 2:

5)GET Request from client to proxy

{"r0" : {"object_path" : "/A1/C2/Obj1", "stripe_id" : "S2", "stripe_size" : 1024, "stripe_content_length" : 1024, "object_offset" : 1024},
"r1" : {"object_path" : "/A1/C2/Obj1", "stripe_id" : "S1", "stripe_size" : 1024, "stripe_content_length" : 1024, "object_offset" : 0},
"r2" : {"object_path" : "/A1/C2/Obj2", "stripe_id" : "S1", "stripe_size" : 1536, "stripe_content_length" : 1536, "object_offset" : 0}
"r3" : {"object_path" : "/A2/C2/Obj2", "stripe_id" : "S3", "stripe_size" : 1536, "stripe_content_length" : 1536, "object_offset" : 4608}
}

6)GET Request from proxy to object server

{"r0" : {"object_id" : "H(/A1/C2/Obj1)", "stripe_id" : "S2", "stripe_size" : 1024, "stripe_content_length" : 1024, "object_offset" : 1024},
"r1" : {"object_id" : "H(/A1/C2/Obj2)", "stripe_id" : "S1", "stripe_size" : 1536, "stripe_content_length" : 1536, "object_offset" : 0},
"r2" : {"object_id" : "H(/A1/C2/Obj2)", "stripe_id" : "S3", "stripe_size" : 1536, "stripe_content_length" : 1536, "object_offset" : 4608}
}

Appendix … Contd

- HTTP headers in JSON format

Approach 2:

7)GET Response from object server to proxy

{"r0" : {"object_id" : "H(/A1/C2/Obj1)", "stripe_id" : "S2", "stripe_size" : 1024, "http_response_offset" : 3000,
"stripe_content_length" : 1024, "object_offset" : 1024},
"r1" : {"object_id" : "H(/A1/C2/Obj2)", "stripe_id" : "S1", "stripe_size" : 1536, "http_response_offset" : 4024,
"stripe_content_length" : 1536, "object_offset" : 0},
"r2" : {"object_id" : "H(/A1/C2/Obj2)", "stripe_id" : "S3", "stripe_size" : 1536, "http_response_offset" : 5048,
"stripe_content_length" : 1536, "object_offset" : 4608}
}

8)GET Response from proxy to client

OBJECT_HEADER:

{"r0" : {"object_path" : "/A1/C2/Obj1)", "stripe_size" : 1024, “object_size" : 4096, “object_on_disk_size" : 2048,
"object_version_delta_size" : NO_SIZE},
"r1" : {"object_path" : "/A1/C2/Obj2)", "stripe_size" : 1536, “object_size" : 4608, “object_on_disk_size" : 1536,
"object_version_delta_size" : NO_SIZE}
}

STRIP_HEADER:

{"r0" : {"object_path" : "/A1/C2/Obj1)", "stripe_id" : "S2", "stripe_size" : 1024, "http_response_offset" : 3000,
"stripe_content_length" : 1024, "object_offset" : 1024},
"r1" : {"object_path" : "/A1/C2/Obj2)", "stripe_id" : "S1", "stripe_size" : 1536, "http_response_offset" : 4024,
"stripe_content_length" : 1536, "object_offset" : 0},
"r2" : {"object_path" : "/A1/C2/Obj2)", "stripe_id" : "S3", "stripe_size" : 1536, "http_response_offset" : 5048,
"stripe_content_length" : 1536, "object_offset" : 4608},
"r3" : {"object_path" : "/A1/C2/Obj1)", "stripe_id" : "S1", "stripe_size" : 1024, "http_response_offset" : 6584,
"stripe_content_length" : 1024, "object_offset" : 0}
}

References

- Lustre file-system architecture
- Swift Architecture
- Swift Large Objects Dynamic and Static
- Vectored IO or Scatter/Gather IO
- Sparse file and Thin provisioning
- SCSI target and Logical Volume Manager
- Redirect-On-Write Snapshot
- Continuous Data Protection
- TRIM/UNMAP
- FiST and FUSE

See also

Enterprise aware/embracing OpenStack

OpenStack enabler

Intelligent Load Balancer for Compute Resources

OpenStack Health Monitoring & Management Framework

Tempest extension for Horizon testing support

Enterprise Aware/Embracing OpenStack

OpenStack components/services do not leverage on the high end capabilities of appliances (software & hardware) built by enterprise vendors in compute, storage and networking space.
The appliances are efficient, greener, low at maintenance, lesser form factors suited for datacenters. Embracing these well into the OpenStack framework along with their enhanced features adds to OpenStack adoptions across the industry.
For instance, enterprise storage vendors (NAS, SAN, SSD based arrays) have features like replication, snapshotting, de-duplication, also at times provide object store support
Enterprise Network vendors have features like Deep Packet Inspections & Analysis, Firewall, etc.
Leveraging on the features can help reduce processing, power consumption, and increase efficiency in respective areas of cloud infrastructures.
Go along with the enterprise vendors for OpenStack to be omnipresent. Increasing the relevance and longevity of OpenStack for the industry and community.
This is a seed of thought of making OpenStack more and more relevant in emerging cloud world from strategic and business perspective.

Contributors: Tejas Sumant, Shriram Pore

OpenStack Enabler

The idea behind the OpenStack enabler is to have a tool or application which helps OpenStack based cloud providers to deploy their cloud infrastructure. The enabler graphically assists a provider to estimate the hardware requirements for meeting the cloud service needs. It also provides a deployment view of various OpenStack services to orchestrate the infrastructure. This can give possible blueprints of the cloud deployment which the cloud service provider or private cloud administrator can choose from.

Consider following use cases -

Case 1:

If the available hardware information is provided, then the OpenStack enabler will predict all possible ways to deploy the cloud and also predict the virtualization capacity generated by the given set of hardware.

Case 2:

If the virtualization needs are provided, which are tunable parameters like, number of VMs, storage capacity per VM, processing power per VM, RAM consumption per VM, objects serving throughput, etc. then OpenStack enabler predicts the required hardware to satisfy the need. It can even estimate the CapEx requirements.

Contributors: Sachin Patil, Shrirang Phadke, Shriram Pore

Intelligent Load Balancer for Compute Resources

The compute resource monitoring enables the administrators to get the QoS, health & resource utilization parameters. This helps achieve the migration of compute resources (VMs) as and when required. By providing support for auto-scaling of **Heat** in mind, the idea is to create a framework that will enable effective usage of 'Nova compute resources' utilizing the monitoring capabilities on the compute resources **(Host)** and perform live migration between hosts as required.

The load balancer will create an automated system with capable decision making to address the impact of monitoring. It will also enable more green resources by enabling and disabling the compute resources on demand.

Provisioning, Live-migration and Auto-scaling (provided the resources support, remote management capabilities).

The framework will in turn utilize the supported APIs provided by underlying virtualization platforms. This framework can also be extended to include:

Automated and seamless cross-platform migration using OVF (Open Virtualization Format) without the user intervention. E.g. migration between the Open stack implementations hosted by VMware, Citrix, KVM, Hyper-V and so on. HA (High Availability), FT (Fault Tolerance), SRA (Scattered Resource Administration) can also be achieved.

Contributors: Swapnil Kulkarni, Shriram Pore

OpenStack Health Monitoring & Management Framework

With OpenStack deployments and usage in mind, the idea is to provide a standardized, integrated yet independent health monitoring management framework for OpenStack services and infrastructure (Hypervisor, Storage and Network). The framework can be based on the Common Information Model (CIM). CIM is an open standard that defines how managed elements in an IT environment are represented as a common set of objects and relationships between them. This is intended to allow consistent management of the elements, independent of their manufacturer or provider.

For OpenStack services, CIM based management framework can be provided with defined available interfaces (preferably REST & CLI) for monitoring & standardized service responses. The REST interfaces could be used by OpenStack dashboard or user client directly, whereas the standard service responses can be used by standard monitoring tools. This will help standardize OpenStack monitoring capabilities for existing as well as future services.

For OpenStack infrastructure, the same set of interfaces can be implemented for underlying hardware to get the health statistics. It can be SNMP based CIM implementation or can vary depending on the provider.

Above implementation of OpenStack health monitoring will help in:

1. Standardized service status responses
2. Standardized API for health monitoring for services and infrastructure

Contributors: Swapnil Kulkarni, Arun Sharma, Shriram Pore

Tempest Extension for Horizon Testing

User Interface is one of the major reasons for cloud administrator's acceptance. The more user friendly and bug free it is, higher is the acceptance and adoption. Horizon being user interface for OpenStack, needs to stand the testimony and get more acceptances. The interface needs to be standardized, consistent and bug free, and hence requires testing. But testing of user interface is not similar to testing other services/components of OpenStack; hence we suggest an extension to tempest for Horizon testing.

As OpenStack is growing, here is an integration test which will ensure that the horizon runs smoothly. It tests basic functionality of three central dashboards "User Dashboard", "System Dashboard", and "Settings". Further extend the scope to more complex and complicated scenarios.

Currently in tempest project, there is no test coverage for horizon dashboard. In this paper we aim to introduce testing of OpenStack horizon as a part of tempest framework using selenium python web driver and unittest(2) python standard library. This will execute the tests of horizon with continues integration. We suggest the use of xvfb (virtual frame-buffer X server for X Version 11).

This will enable current directory structure of tempest to extend with 'horizon' support making no changes in existing framework.

Contributors: Shrikant Chaudhari, Shriram Pore

Tempest Extension for Horizon Testing

User Interface is one of the major reasons for cloud administrator's acceptance. The more user friendly and bug free it is, higher is the acceptance and adoption. Horizon being user interface for OpenStack, needs to stand the testimony and get more acceptances. The interface needs to be standardized, consistent and bug free, and hence requires testing. But testing of user interface is not similar to testing other services/components of OpenStack; hence we suggest an extension to tempest for Horizon testing.

As OpenStack is growing, here is an integration test which will ensure that the horizon runs smoothly. It tests basic functionality of three central dashboards "User Dashboard", "System Dashboard", and "Settings". Further extend the scope to more complex and complicated scenarios.

Currently in tempest project, there is no test coverage for horizon dashboard. In this paper we aim to introduce testing of OpenStack horizon as a part of tempest framework using selenium python web driver and unittest(2) python standard library. This will execute the tests of horizon with continues integration. We suggest the use of xvfb (virtual frame-buffer X server for X Version 11).

This will enable current directory structure of tempest to extend with 'horizon' support making no changes in existing framework.

Contributors: Shrikant Chaudhari, Shriram Pore

Authors Biography

Shriram Pore - Senior Architect, Calsoft Inc.

- A veteran of storage industry
- More than 12+ years of experience in architecting and developing products
- Key strength lies in quickly understanding product requirements and translating them into architectural and engineering specs for implementation.
- Working on Virtualization and cloud platforms more towards OpenStack
- Building in-house expertise and executing projects using or extending OpenStack
- Architected, designed and implemented solutions for NAS, Virtualization integrated Backup-Recovery and Clone of VM
- Led FS and replication teams on the file-server and also led the CORE component team from system management perspective (configuration, provision, and manage various facilities of file-server).
- Executed ideation projects related to Lustre file-system.
- Master Of Computer Science from India, Pune University
- Bachelor of Computer Science from India, Pune University
- Publication - http://goo.gl/3eVwAm - Pragmatic about software product performance
- Publication - http://goo.gl/BbABCO - How can Hypervisors leverage Advanced Storage features?
- To know more about Shriram connect on http://lnkd.in/iWFJz2

Contributors Biography

Kashish Bhatia - Senior Development Engineer, Calsoft Inc.

- Technology enthusiast, with most of the experience in storage domain
- Worked on different Storage technologies (SAN and NAS)
- Has exposure to other Linux subsystems like file systems, block layer, SELinux, etc.
- Experience of working on distributed file system(Lustre) and storage virtualization.
- Deep understanding of OpenStack Swift Architecture
- Interests: contribution in designing and implementation of challenging softwares
- Bachelor of Engineering, Computer Science, Pune University
- To know more about Kashish connect on LinkedIn

Contributors Biography

Bipin Kunal - Senior Development Engineer, Calsoft Inc.

- Zealous for technology, with most of the experience in storage domain
- Worked on different Storage technologies (SAN and NAS)
- Experience of working on SELinux, distributed file system(Lustre) and Storage virtualization
- Deep understanding of OpenStack Swift Architecture
- Interests: Cloud computing, File systems
- Bachelor of Engineering, Computer Science, Pune University
- To know more about Bipin connect on LinkedIn
- Checkout blogs on KGDB Tutorials, Facebook's Flashcache, LSM and SElinux overview and internals

Thank You.

Calsoft Inc.


www.calsoftinc.com


smm@calsoftinc.com

❖ USA

2953 Bunker Hill Lane, Suite 400,

Santa Clara, CA 95054.

Phone: +1 (408) 834 7086

❖ INDIA

SR Iriz, 4th Floor, Plot A, S.No. 134/2/1

Pashan - Baner Link Road, Pune 411008

Phone: +91 (20) 4079 2900