On the Synchronization Bottleneck of OpenStack Swift-Like Cloud Storage Systems

IEEE Transctions on Parallel and Distributed Systems, Vol.28, No.9, 2018

Ruan, Titcheu, Zhai, Li, Liu, E, Cui, Xu

Presented by Andrew Vaillancourt
Grid and Cloud Computing – Winter 2019

Overview

- **Introduction & Motivation**
- **Preliminaries/Background**
- **Problem Statement**
- **Proposed Solution**
- **Results**
- **Conclusions**

Introduction

- ***OpenStack Swift-like*** **systems are an** ***object storage*** **method that replicates each object across multiple nodes.**
- **These systems rely on certain** ***object-synchronization protocols*** **to achieve high reliabilty and** ***eventual consistency***
- **This paper shows that the performance of these protocols relies heavily on the number of replicas** ***per object*** **and the number of objects, hosted on** ***each node*****.**

Methodology

- **Building of a small Swift cluster to measure performance in varying data intensive environments.**
- **Determine under which conditions performance degrades (Synch Bottleneck).**
- **Review source code of OpenStack Swift to determine root cause of Synch Bottleneck.**
- **Design and implement improvements to OpenStack Swift (called LightSynch).**
- **Measure performance of LightSynch on lab scale and large scale Swift environments.**

Preliminaries

- **CAP Theorem**
- **Eventual Consistency**
- **Object Storage**
- **OpenStack Swift design and discussion of the synch protocols used**

CAP Theorem

The CAP Theorem(Brewer) **states that in distributed data storage systems, you can only provide 2 out of the following 3 attributes simultaneously:**

1. **Consistency**
2. **Availability**
3. **Partition Tolerance**

CAP Theorem - Consistency

Consistency:

- A guarantee that every node in a distributed cluster returns the same, most recent, successful write.
- Every client has the same view of the data.
- There are various types of consistency models.
- Consistency in CAP (used to prove the theorem) refers to sequential consistency, a very strong form of consistency.

CAP Theorem - Availability

Availability:

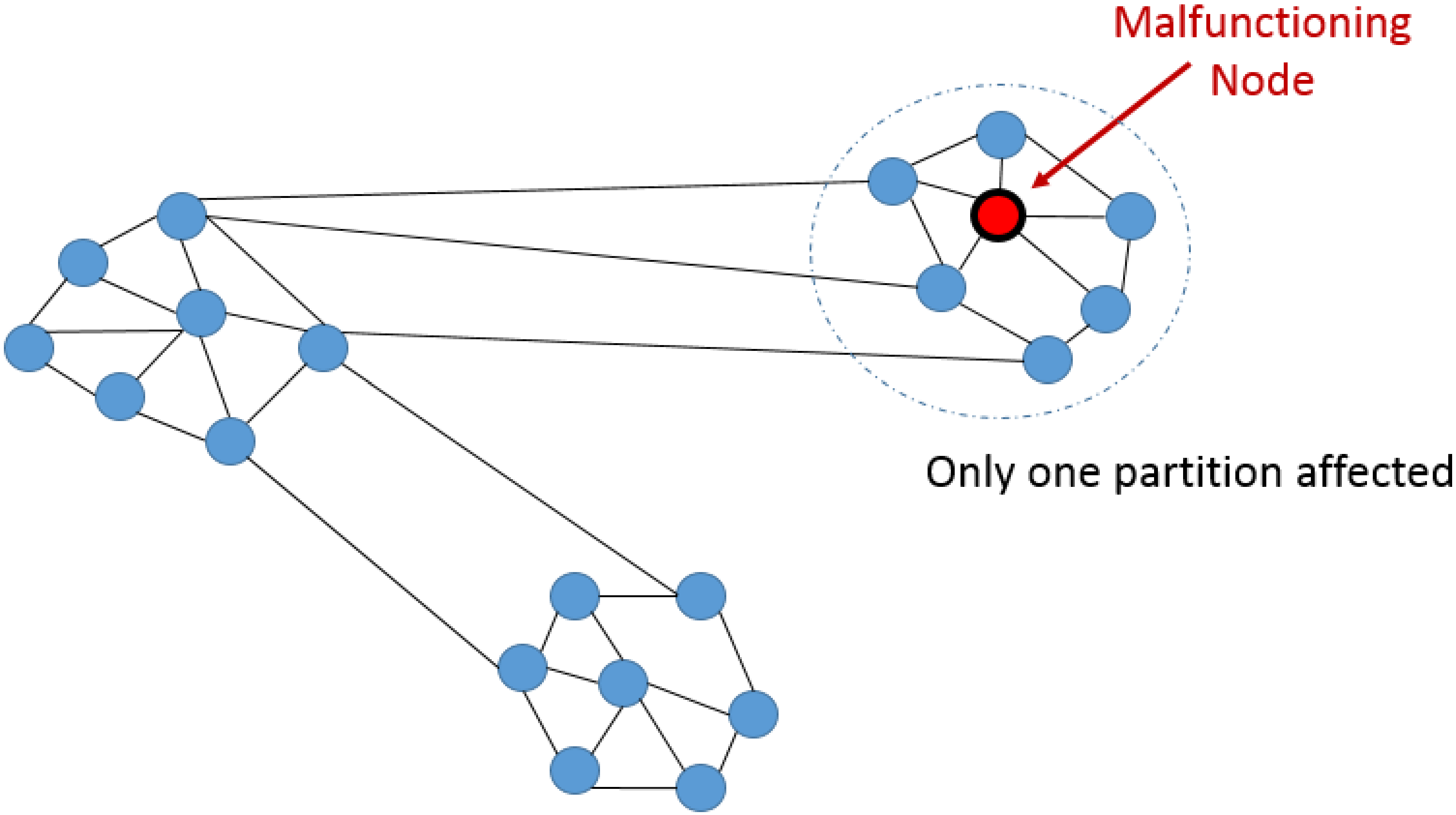
- Every *non-failing node* returns a response for all read and write requests in a *reasonable* amount of time.
- Guarantees that every request receives a response about whether it succeeded or failed.

CAP Theorem – Partion Tolerance

Partition Tolerance:

- The system continues to operate even if any one part of the system is lost or fails
- A system that is partition-tolerant can sustain any amount of network failure that doesn't result in a failure of the entire network.

CAP Theorem – Partion Tolerance



https://towardsdatascience.com/cap-theorem-and-distributed-database-management-systems-5c2be977950e

CAP Theorem – Availability vs Consistency

- In modern day distributed systems, partition tolerance is a requirement, *not* an option.
- Therefore, the trade-offs to be considered when designing a distributed data store are almost always between availability and consistency
- *Swift* compromises on consistency, opting for a model known as *eventual consistency*

Eventual Consistency

- In order to maintain high-availabilty with reasonable response times, Swift uses the ***eventual consistency*** model.
- Given enough time, the replica values distributed across all nodes will be consistent eventually
- This implies that in *some* cases a client will read an old copy of the data object
- We will refer to the time period between an update and *convergence* (*all connected nodes observe one another's updates*) as the **synch delay**

OpenStack Swift Architecture

There are 2 types of nodes in a Swift cluster:

- **Storage Node**:
  - responsible for storing objects
- **Proxy Node**:
  - acts as a bridge between client and storage nodes
  - communicate with clients and retreive or allocate requested objects to/from storage nodes
  - Uses the hash of an object to find which partition it’s in, and which disks/nodes have a replica of that partition

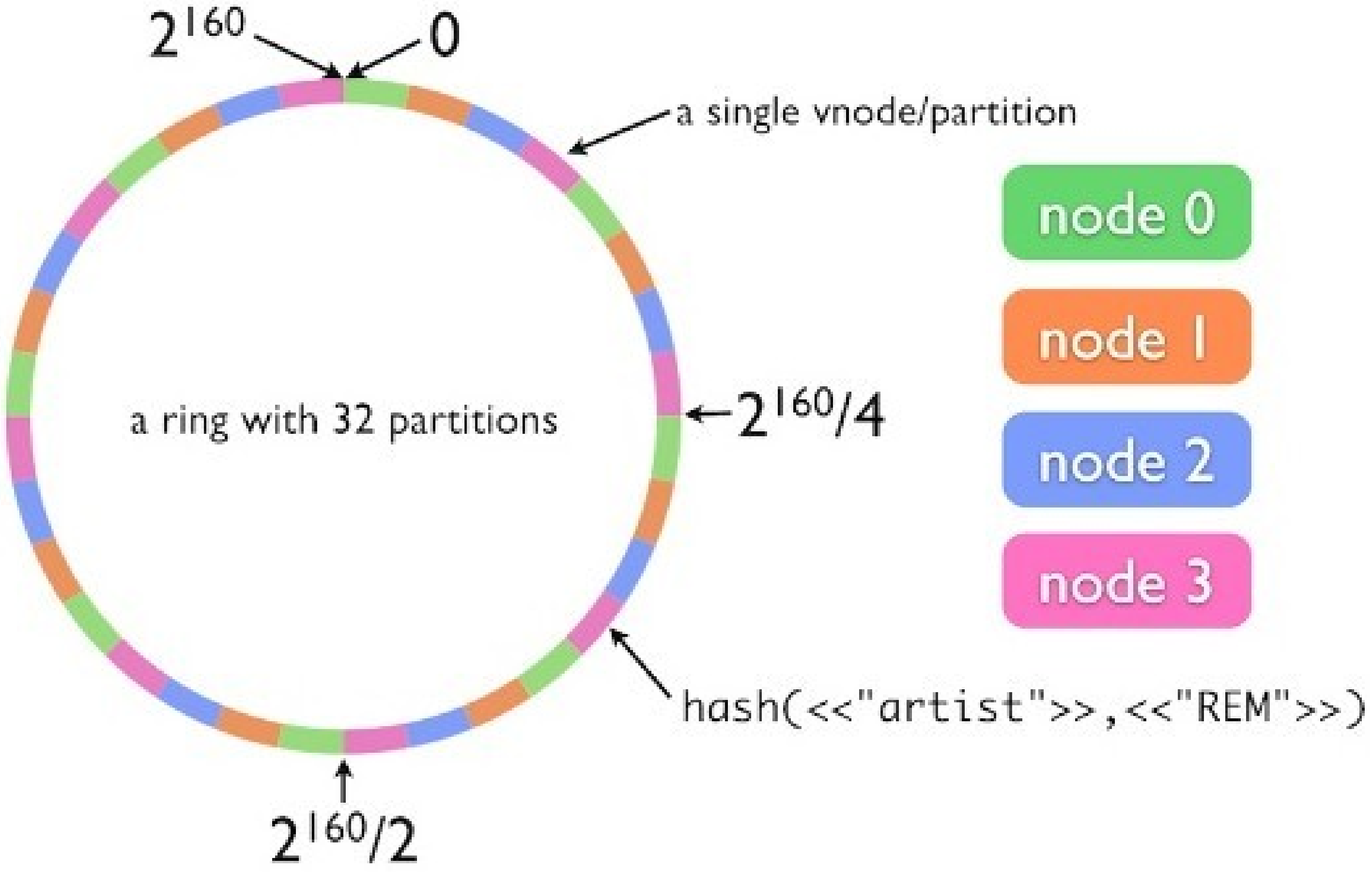
OpenStack Swift Architecture

- **Proxy Nodes –** handle incoming API requests
- **Storage Nodes –** store partitions on disks
- **Partition** – container of objects and lookup tables
  - Replication and data movement among nodes is done at the *partition-level*
- **Rings (DHT) –** map logical names of data to locations on particular disks

OpenStack Swift Architecture

- **Consistent Hashing: data is distributed using a hashing algorithm to determine its location.**
- Using only the hash of the ID of the data you can determine exactly where that data should be
  - *This mapping of hashes to locations is usually organized in a logical ring*

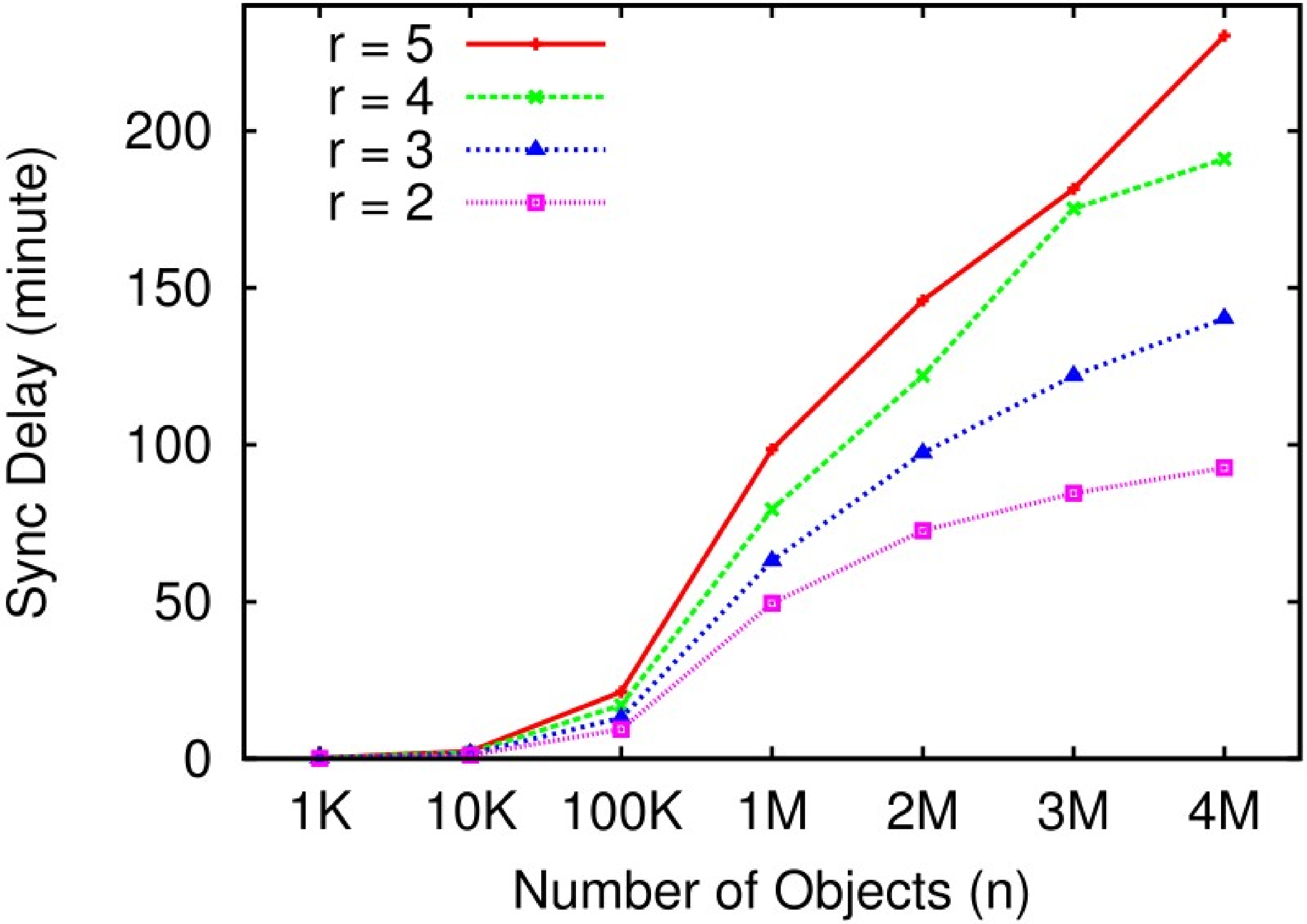
OpenStack Swift Architecture



Experimental Setup

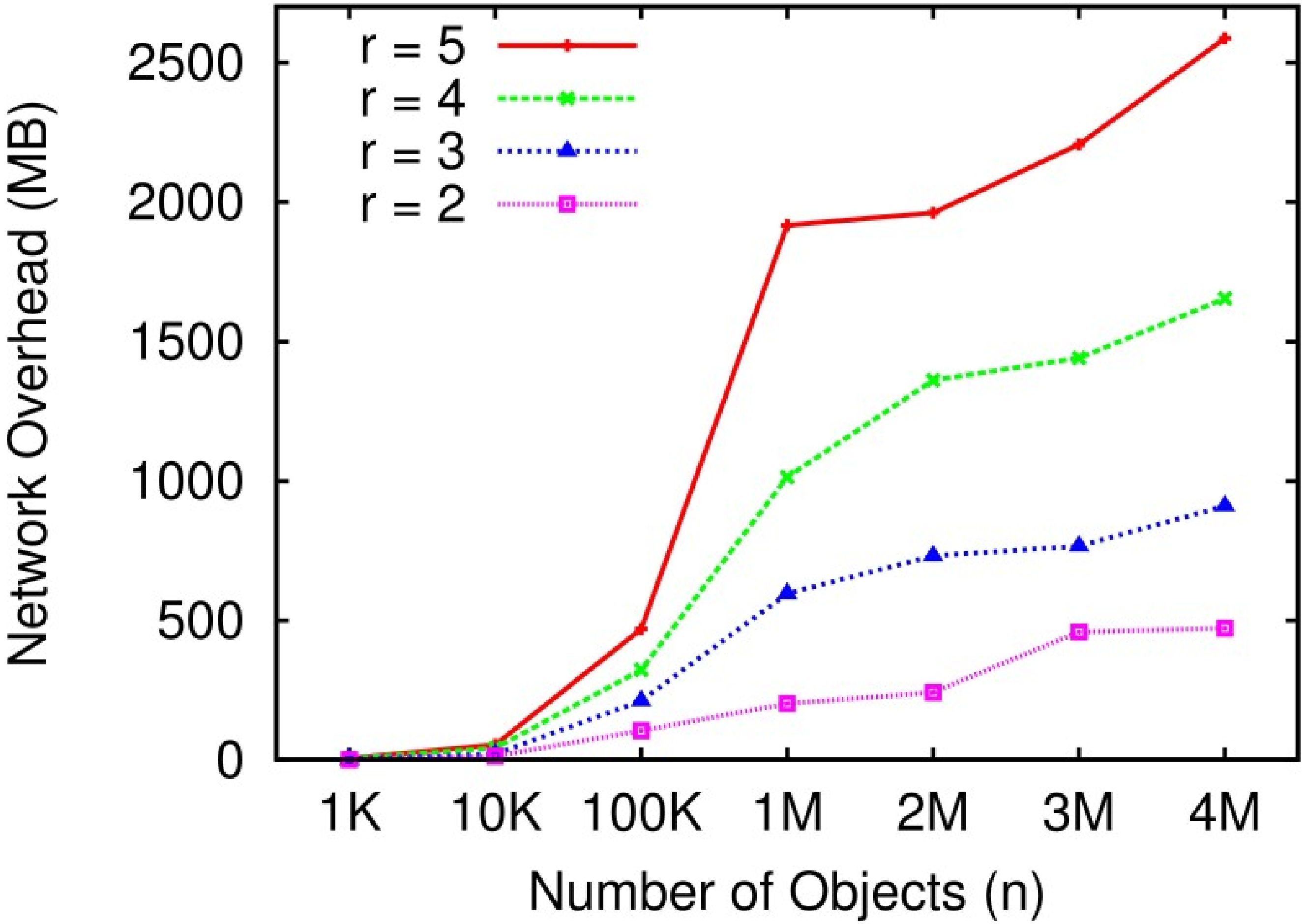
- 5 Nodes connected via Ethernet switch each with:
  - 8 cores, 32gb RAM, 8 x 600 gb SAS disk drives
  - 1 node runs the OpenStack authentication and networking services, and also acts as *both* the proxy node and storage node
  - Other 4 nodes are *only* used for storage
- Multiple laptops attached to the switch to act as clients
  - They will send object storage requests via SwiftStack Benchmark Suite as 6-10kb sized objects
    - They state in their paper that object size/type has *negligible* impact on object synch performance

Swift Results - Synch Delay



* *in stable state*

Swift Results - Network Overhead



* *in stable state*

Problem Statement

- **The performance of the object-sync protocols relies heavily on 2 parameters:**
  - ***r*** –> number of *replicas* for each object
  - ***n*** –> number of *objects* hosted on each node
- **It was found that in data-intensive scenarios (** *when* ***r*** **>** ***3*****,** ***n*** **>>** ***1000*****), the synch process is significantly delayed and produces massive network overhead.**
- **Referred to as the *Synch Bottleneck Problem*.**

Synchronization Bottleneck

- **These synch delay results occured in a stable state** *(few object updates)*
- **Synch delay increases by an additional 34% and 40.2%, respectively, in the presence of data creations and deletions**
- **It appears that the *root cause* of the synch bottleneck is the large network overhead**

Swift Results - Network Overhead

- This massive network overhead results because the *per-node*, *per-synch-round* number of messages sent is Θ(n x r)
- *all* hashes of the objects, for *each* partition replica, must be sent to *each* node containing a replica
  - *Note:* this is an all-to-all communication
  - There is also the added overhead of having to push object updates to inconsistent nodes

Root Cause

- There seems to be 2 main causes to the synch bottleneck problem:
    1. Large message size
        - *Hashes of each object in the partition is sent*
    2. High message count per synch round
        - *All-to-all communication*

- Can this be improved?

Proposed Solution - LightSynch

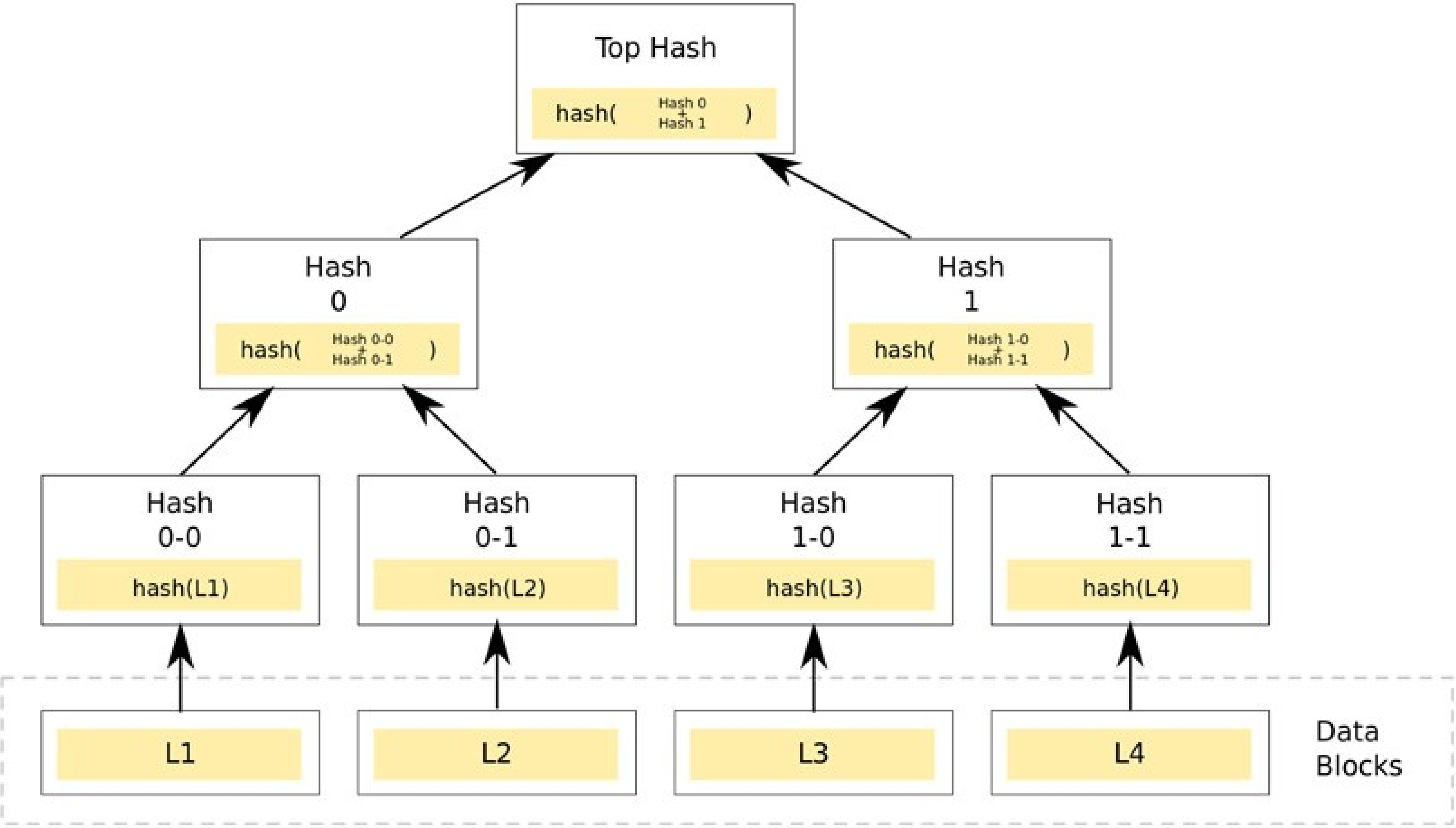
3 Main Components:

- **Hashing of Hashes (HoH)**
- **Circular Hash Checking (CHC)**
- **Failed Neighbour Handling (FNH)**

Preliminary – Merkle Tree

- **HoH** aggregates *all* hashes of a partition into a Merkel Tree structure
- *Merkle Tree*: A hash tree where the parent nodes contain a hash of its child nodes
- **Merkle trees** are used so that data integrity can be compared quickly with one hash value, and if inconsistency is discovered, the offending leaf node can be be found in ***O(logn)*** time.
  - This data structure underpins many distributed technologies like BitTorrent and Blockchain.

Merkle Trees



LightSynch - HoH

- *Problem*: This just trades one large synch message for *log(n)* smaller messages
- Not really an improvement because of round-trip network latency
- *Solution*: LightSynch only maintains the root hash, and the leaf nodes.
  - If the root hashes of 2 partitions do *not match*, LightSynch will directly compare the hash values in *all* the leaves of the Merkle tree.

LightSynch - HoH

- each *initial* synch message will only contain the root hash value of the partition
- the *larger* synch message containing the leaves will *only* be sent in the case when *inconsistency is encountered*
  - *Inconsistency is encountered far less than consistency, even in bursty update conditions and node failures*

LightSynch – Circular Hash Checking

- HoH cuts down on message size..

  ...but what about the number of messages?

- **Swift Architecture**
  - In Swift, when a node receives an update, it is responsible for pushing those updates to all other remote nodes
  - Since the remote nodes have now been updated, they will also send synch messages back to all other nodes checking for consistency
  - This is an **all-to-all** operation

LightSynch – Circular Hash Checking

- **Circular Hash Checking**
  - Lightsynch instead organizes replicas in a logical ring, and only pushes updates to it's clockwise neighbour
  - This was not too challenging to implement because Swift already organizes its partitions in a ring
  - **This reduces the number of synch messages from $r(r-1)$ to r**

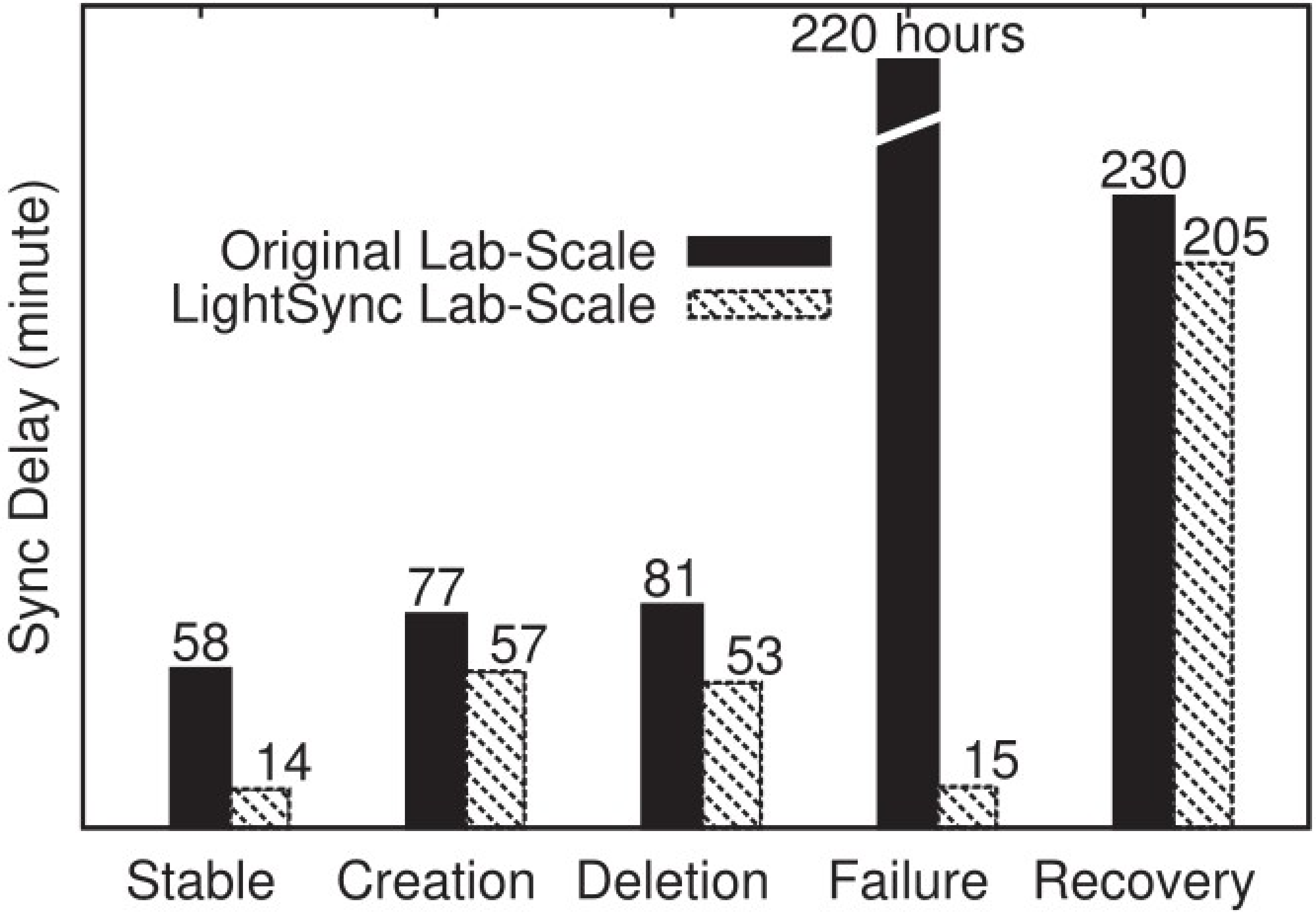
LightSynch – Failed Node Handling

- **Node failures significantly *degrade* the synch performance of Swift**
- **HoH and CHC do not alleviate these issues**
- **In fact, a node failure would likely impair the Circular Hash Checking protocol in LightSynch**
- **Thus, LightSynch needs to improve node failure detection and handling**

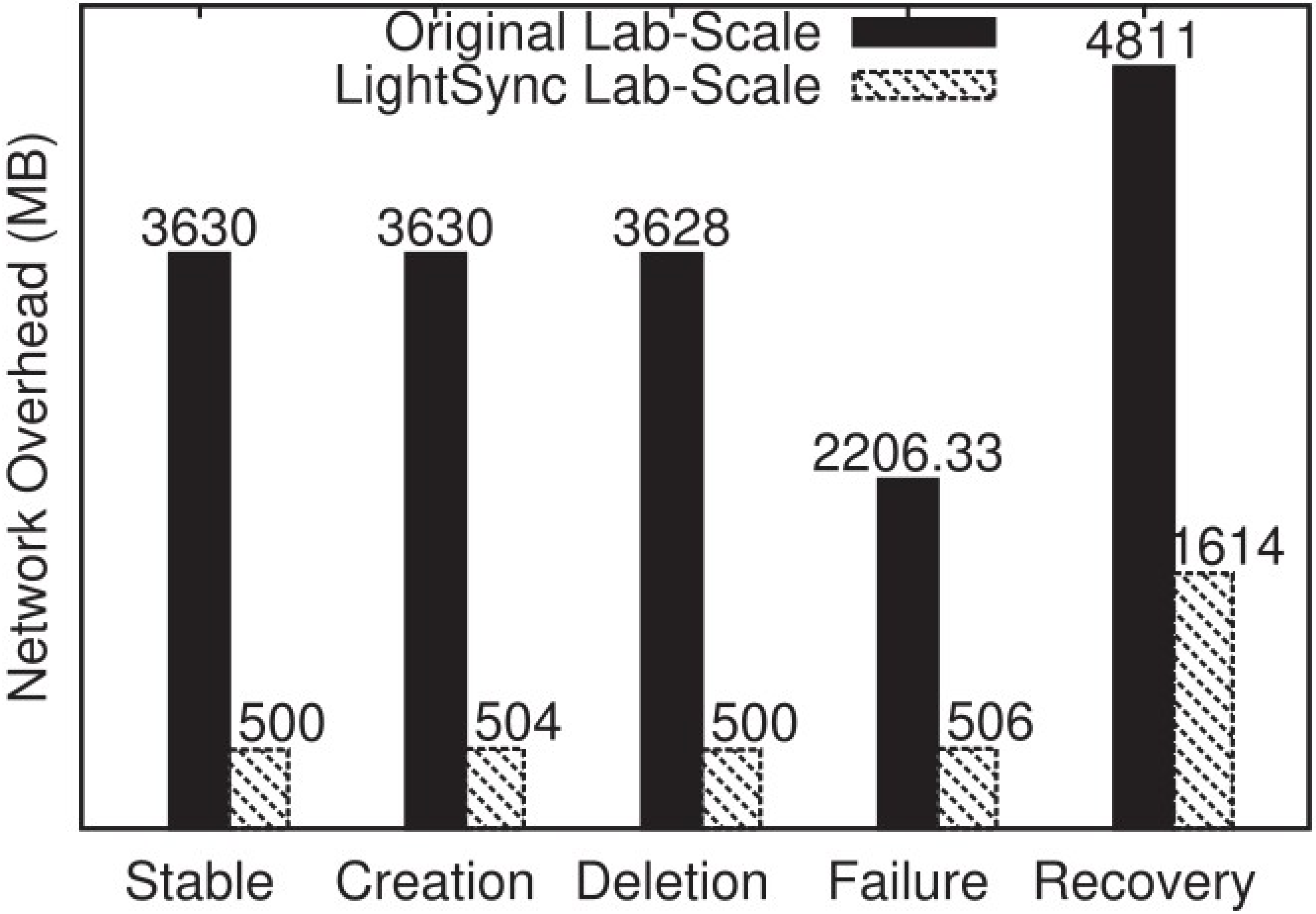
LightSynch – Failed Node Handling

- **Each CHC ring maintains a table of *heartbeat responses* from member nodes.**
- **If a threshold is *passed* without a response, the node is considered as failed and removed from the CHC ring.**
- **Also, when a node rejoins the ring, it's neighbours' partitions are moved to head of OpenStack's synch queue so that the ring can be rebuilt quicker.**

LightSynch Results – Synch Delay



LightSynch Results – Network Overhead

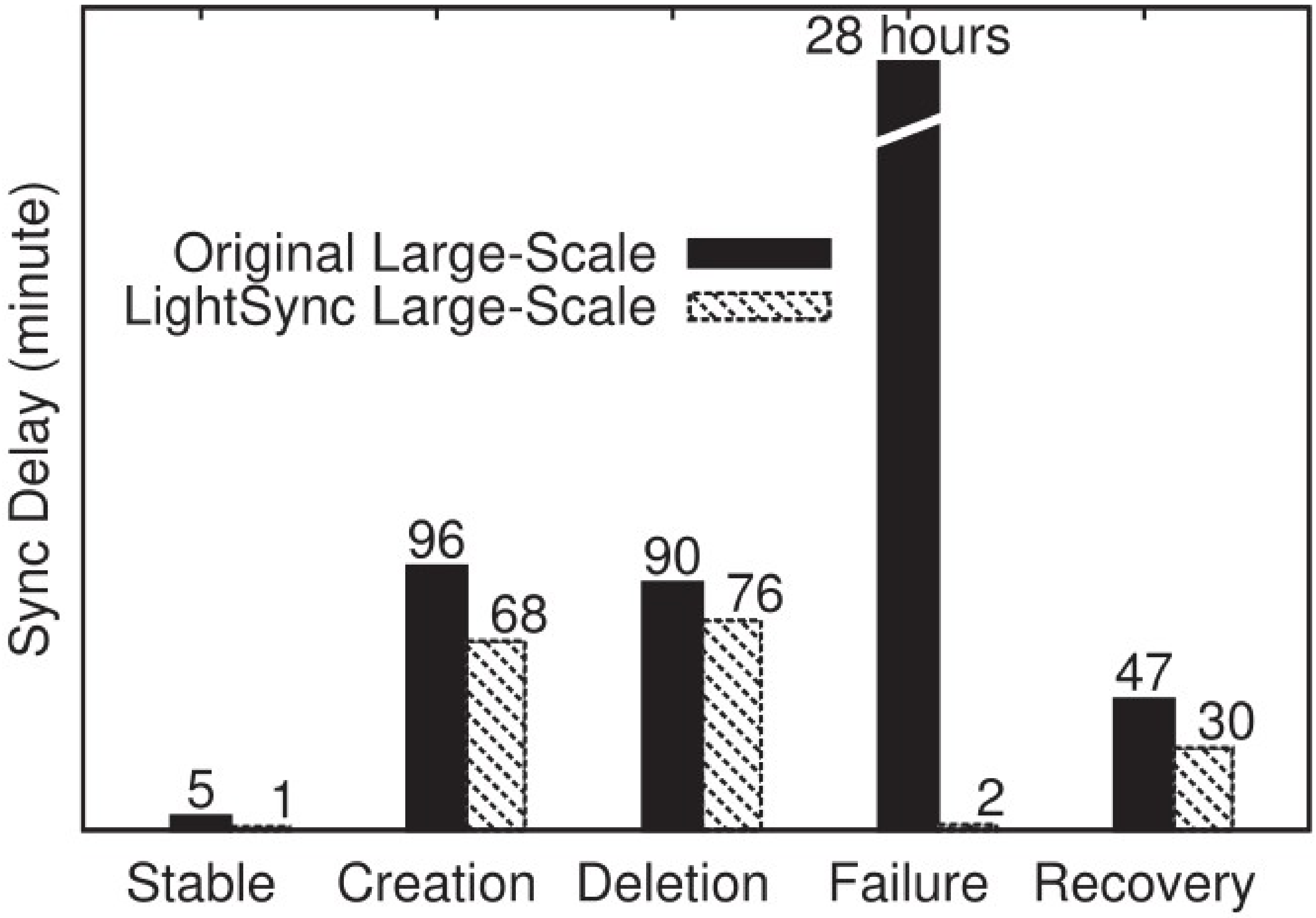


Large Scale Experiement

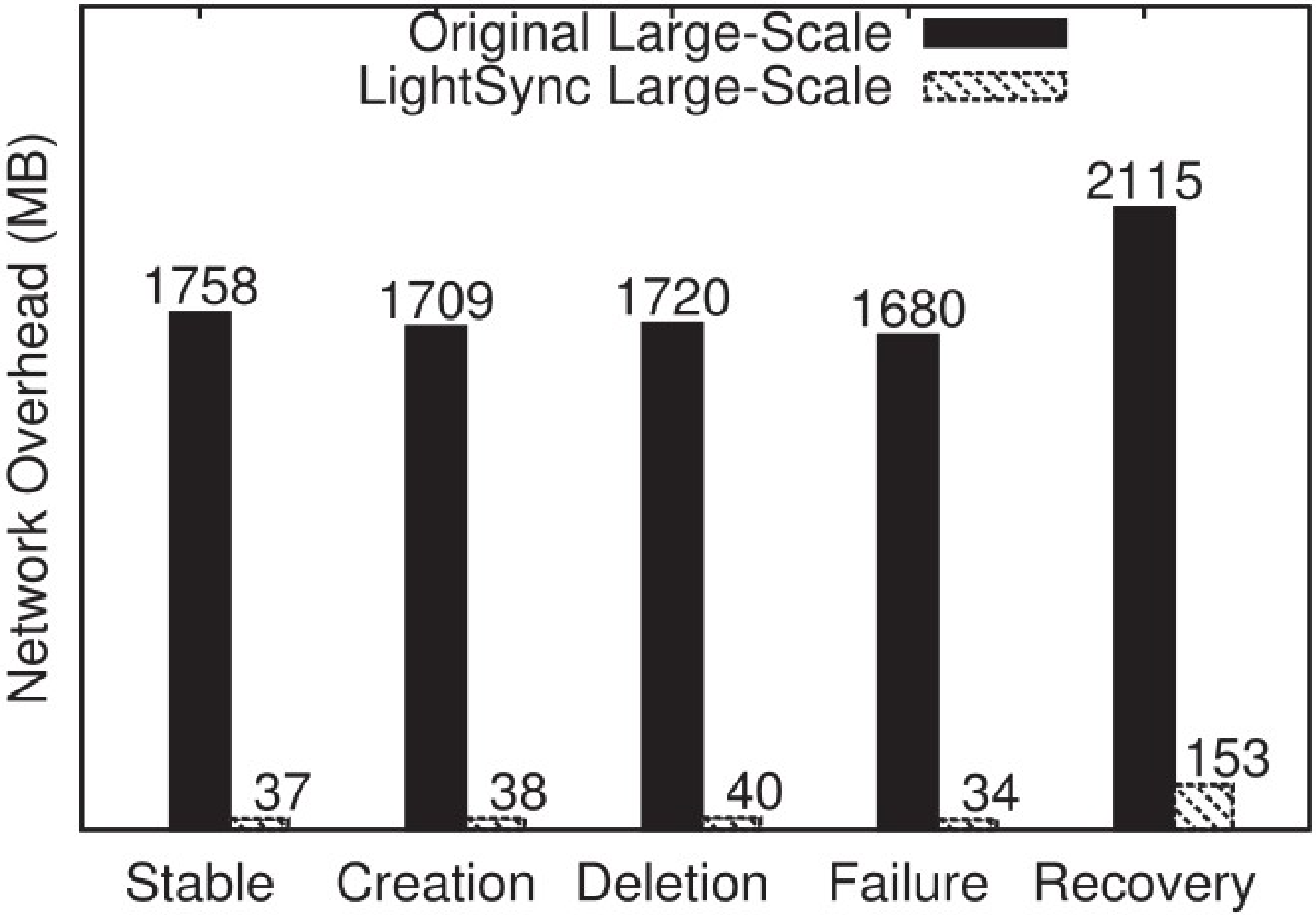
64 VMs on Alibaba Cloud:

- Dual Intel Xeon 2.3GHz
- 4 GB RAM
- 600GB disk storage
- Ubuntu 16.04
- Connected by LAN

Large Scale Results – Synch Delay



Large Scale Results – Network Overhead



Conclusion

- **The OpenStack Swift object synchronization protocols are not well suited to data-intensive scenarios.**
  - *This is mainly due to the large network overhead.*
- **This problem is significantly aggravated in the case of data updates and node failures.**
- **The design of LightSynch provides a provable guarantee on the reduction of network traffic with comparable CPU and memory usage.**
- **LightSynch works directly as an OpenStack Swift patch and can reduce the synch delay by up to *879*X and the network overhead by up to *47.5*X.**

My Conclusions

- **Well-written paper**
- **Interesting work**
- **Quite well presented**

Personal Conclusions/Criticism

- **In lab-scale experiments, Node-0 was used to run the user authentication *and* networking services as well as function as *both* the proxy node *and* a storage node...**

Personal Conclusions/Criticism

- Would this tripling of responsibilties have a negative effect on the throughput of the synch messages being sent?
  - What about it’s own local object synchronization?
    - some of its resources are diverted to authenticate requests, as well as act as a proxy to the other nodes.
  - They did mention that CPU and memory usage was affordable, but this seems like more of a networking issue.
- Large scale experiments, however, show similar results to the lab experiements, so perhaps this would not have a dramatic effect on performance.

Personal Conclusions/Criticism

- They seem to conflate replication at the *object-level* and *partition-level* throughout their paper.
  - *This may be confusing for a reader unfamiliar with Swift's internal architecture.*
- Partition count and size is static after cluster configuration, so I would've liked to have seen these varied in their experiments.
- They didn't mention at what capacity the drives were at in their experiments, and if this would affect the synch delay.

Personal Conclusions/Criticism

- Could’ve used a little more description about Swift’s internal architecture and design of it’s synch protocols throughout to better appreciate their improvements.
  - I had to do *a lot* of research on my own.

Personal Conclusions/Criticism

Contradictions?

- Brief mention of synch threads optimization:
  - Swift’s way of “parallelzing” some portion of the synch functions.
  - They show that using 8 threads can reduce the synch delay from 58 to 21 minutes..

    .. 2 sentences later say increasing parallelism contributes little to reduce synch delay??

Personal Conclusions/Criticism

Contradictions?

- State that the *size* of object reads/updates have negligible effect on the synch delay, but this was not proved to my satisfaction.
  - At one point they state that the synch process is I/O bound.
  - This would imply that object size ***is*** a factor, especially during object creation.