Cloud Storage – A look at Amazon's Dyanmo

A presentation that look's at Amazon's Dynamo service (based on a research paper published by Amazon.com) as well as related cloud storage implementations

The Traditional

- Cloud Data Services are traditionally oriented around Relational Database systems
  - Oracle, Microsoft SQL Server and even MySQL have traditionally powered enterprise and online data clouds
  - **Clustered -** Traditional Enterprise RDBMS provide the ability to cluster and replicate data over multiple servers – providing reliability
  - **Highly Available –** Provide Synchronization (“Always Consistent”), Load-Balancing and High-Availability features to provide nearly 100% Service Uptime
  - **Structured Querying –** Allow for complex data models and structured querying – It is possible to off-load much of data processing and manipulation to the back-end database

The Traditional

- However, Traditional RDBMS clouds are:

  EXPENSIVE!

  To maintain, license and store large amounts of data

- The service guarantees of traditional enterprise relational databases like Oracle, put high overheads on the cloud
- Complex data models make the cloud more expensive to maintain, update and keep synchronized
- Load distribution often requires expensive networking equipment
- To maintain the “elasticity” of the cloud, often requires expensive upgrades to the network

The Solution

- Downgrade some of the service guarantees of traditional RDBMS

  - Replace the highly complex data models Oracle and SQL Server offer, with a simpler one – This means classifying service data models based on the complexity of the data model they may required

  - Replace the “Always Consistent” guarantee synchronization model with an “**Eventually Consistent” model** – This means classifying services based on how “updated” its data set must be

Redesign or distinguish between services that require a simpler data model and lower expectations on consistency.
We could then offer something different from traditional RDBMS!

The Solution

- **Amazon's Dynamo** – Used by Amazon's EC2 Cloud Hosting Service. Powers their Elastic Storage Service called S2 as well as their E-commerce platform

  Offers a simple Primary-key based data model. Stores vast amounts of information on distributed, low-cost virtualized nodes

- **Google's BigTable –** Google's principle data cloud, for their services – Uses a more complex column-family data model compared to Dynamo, yet much simpler than traditional RMDBS

  Google's underlying file-system provides the distributed architecture on low-cost nodes

- **Facebook's Cassandra –** Facebook's principle data cloud, for their services.

  This project was recently open-sourced. Provides a data-model similar to Google's BigTable, but the distributed characteristics of Amazon's Dynamo

Dynamo - Motivation

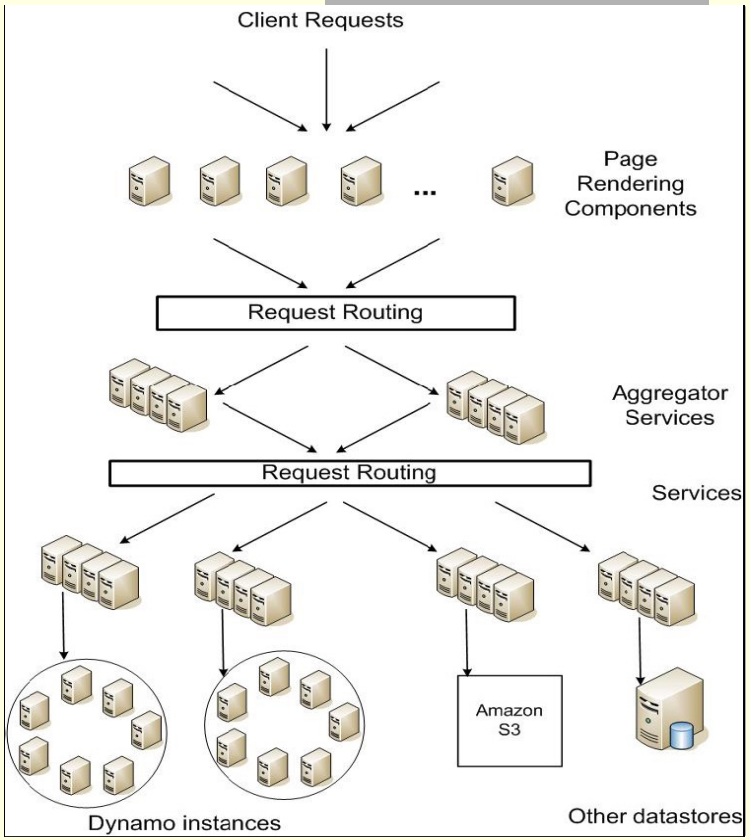
- Build a distributed storage system:
  - Scale
  - Simple: key-value
  - **Highly available**
  - **Guarantee Service Level Agreements (SLA)**

System Assumptions and Requirements

- *Query Model*: simple read and write operations to a data item that is uniquely identified by a key.
- *ACID Properties: Atomicity, Consistency, Isolation, Durability.*
- *Efficiency:* latency requirements which are in general measured at the 99.9th percentile of the distribution.
- Other Assumptions: operation environment is assumed to be non-hostile and there are no security related requirements such as authentication and authorization.

Service Level Agreements (SLA)

- Application can deliver its functionality in abounded time: Every dependency in the platform needs to deliver its functionality with even tighter bounds.
- Example: service guaranteeing that it will provide a response within 300ms for 99.9% of its requests for a peak client load of 500 requests per second.



Service-oriented architecture of Amazon's platform

Design Consideration

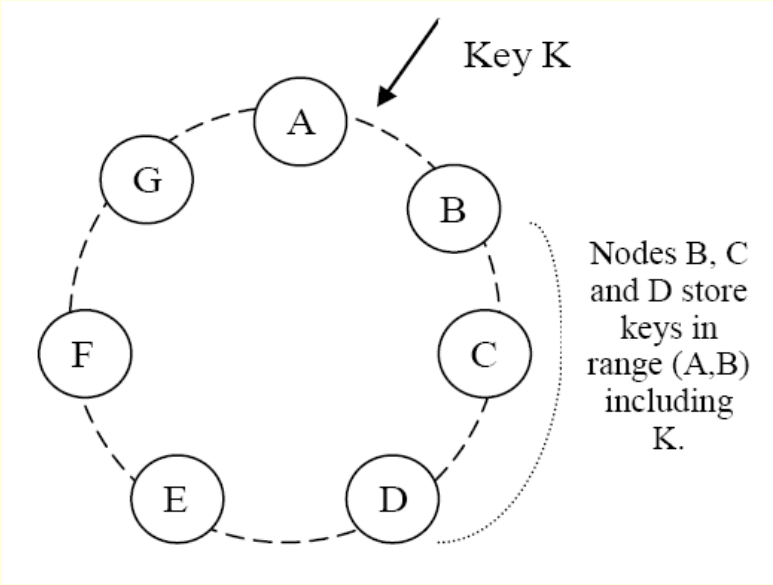
- Sacrifice strong consistency for availability
- Conflict resolution is executed during ***read*** instead of ***write***, i.e. “always writeable”.
- Other principles:
  - Incremental scalability.
  - Symmetry.
  - Decentralization.
  - Heterogeneity.

Summary of techniques used in *Dynamo* and their advantages

Problem	Technique	Advantage
Partitioning	Consistent Hashing	Incremental Scalability
High Availability for writes	Vector clocks with reconciliation during reads	Version size is decoupled from update rates.
Handling temporary failures	Sloppy Quorum and hinted handoff	Provides high availability and durability guarantee when some of the replicas are not available.
Recovering from permanent failures	Anti-entropy using Merkle trees	Synchronizes divergent replicas in the background.
Membership and failure detection	Gossip-based membership protocol and failure detection.	Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.

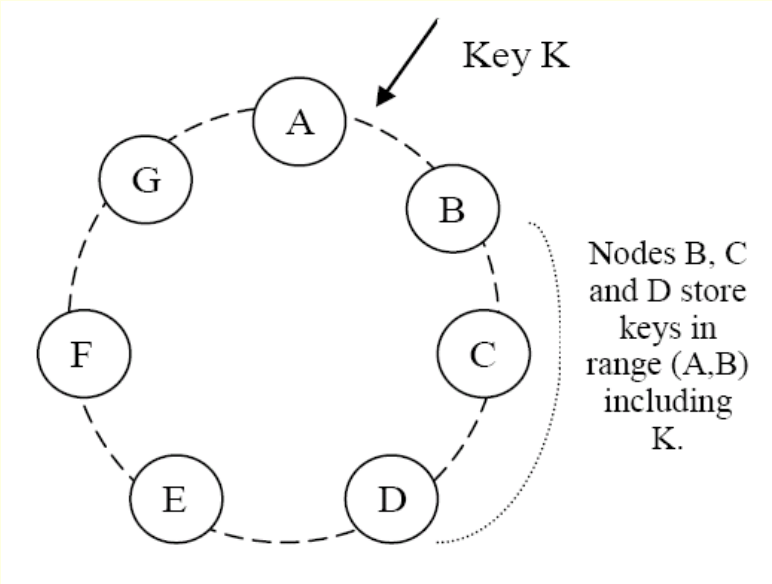
Partition Algorithm

- Consistent hashing: the output range of a hash function is treated as a fixed circular space or “ring”.
  - No finger tables
  - Each node knows complete assignment ➔ no network routing
- ”Virtual Nodes”: Each node can be responsible for more than one virtual node.



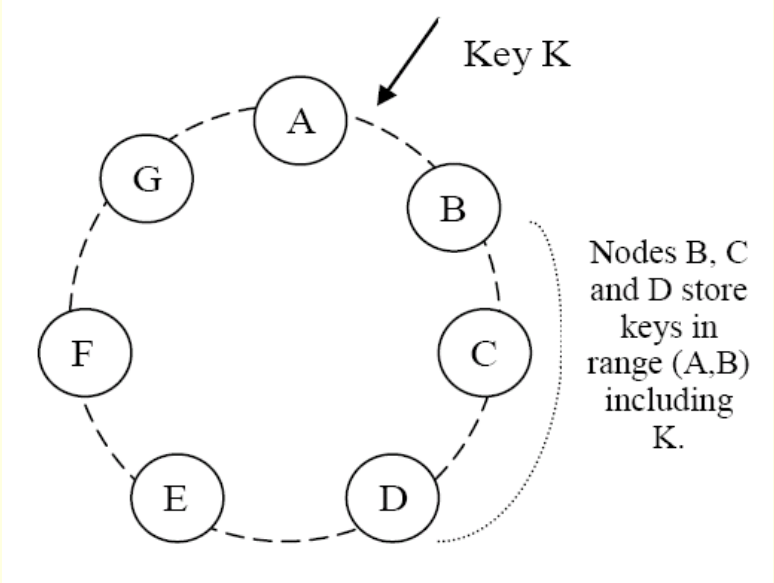
Advantages of using virtual nodes

- If a node becomes unavailable the load handled by this node is evenly dispersed across the remaining available nodes.
- When a node becomes available again, the newly available node accepts a roughly equivalent amount of load from each of the other available nodes.
- The number of virtual nodes that a node is responsible can decided based on its capacity, accounting for heterogeneity in the physical infrastructure.



Replication

- Each data item is replicated at N hosts.
- "*preference list*": The list of nodes that is responsible for storing a particular key.
- Reconciliation only on reads
  - Easy if causally ordered
  - Otherwise application handles



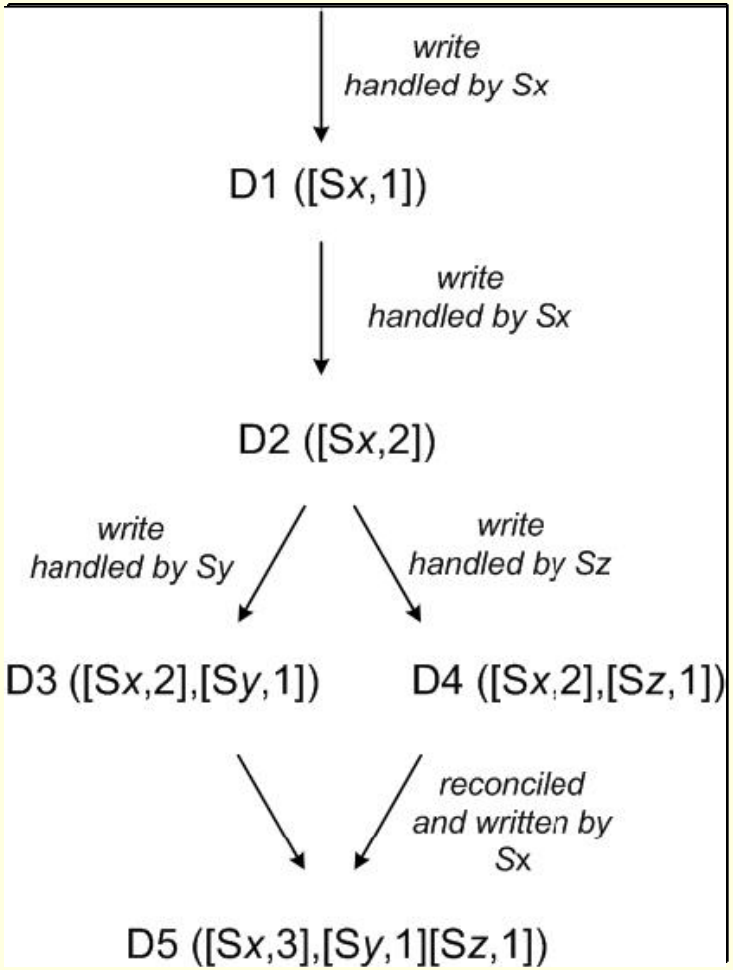
Data Versioning

- A put() call may return to its caller before the update has been applied at all the replicas
- A get() call may return many versions of the same object.
- Challenge: an object having distinct version sub-histories, which the system will need to reconcile in the future.
- Solution: uses vector clocks in order to capture causality between different versions of the same object.

Vector Clock

- A vector clock is a list of (node, counter) pairs.
- Every version of every object is associated with one vector clock.
- *If the counters on the first object's clock are less-than-or-equal to all of the nodes in the second clock, then the first is an ancestor of the second and can be forgotten.*

Vector clock example



Execution of get () and put () operations

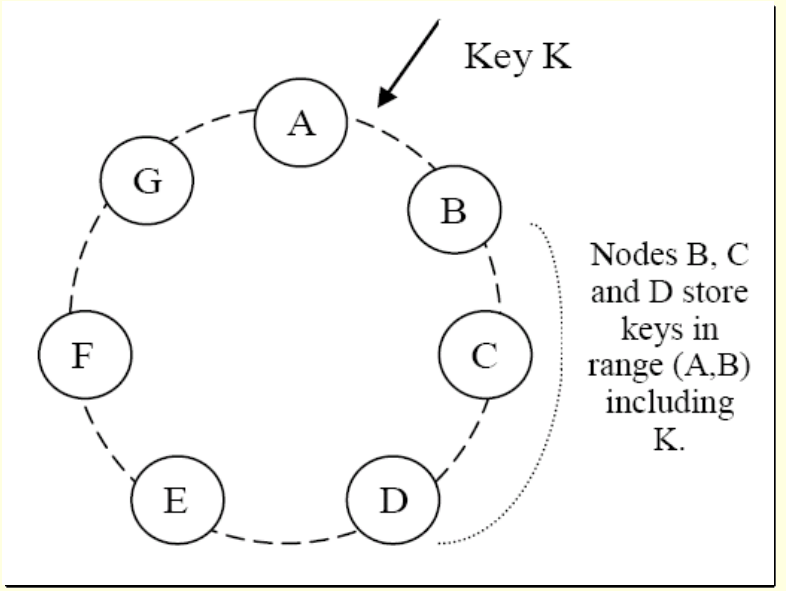
1. Route its request through a generic load balancer that will select a node based on load information.
2. Use a partition-aware client library that routes requests directly to the appropriate coordinator nodes.

Sloppy Quorum

- R/W is the minimum number of (healthy) nodes that must participate in a successful read/write operation.
- Setting R + W > N yields a quorum-like system.
  - Typically R=W=2 and N=3
- In this model, the latency of a get (or put) operation is dictated by the slowest of the R (or W) replicas. For this reason, R and W are usually configured to be less than N, to provide better latency.

Hinted handoff

- Assume N = 3. When A is temporarily down or unreachable during a write, send replica to D.
- D is hinted that the replica is belong to A and it will deliver to A when A is recovered.
- Again: “always writeable”



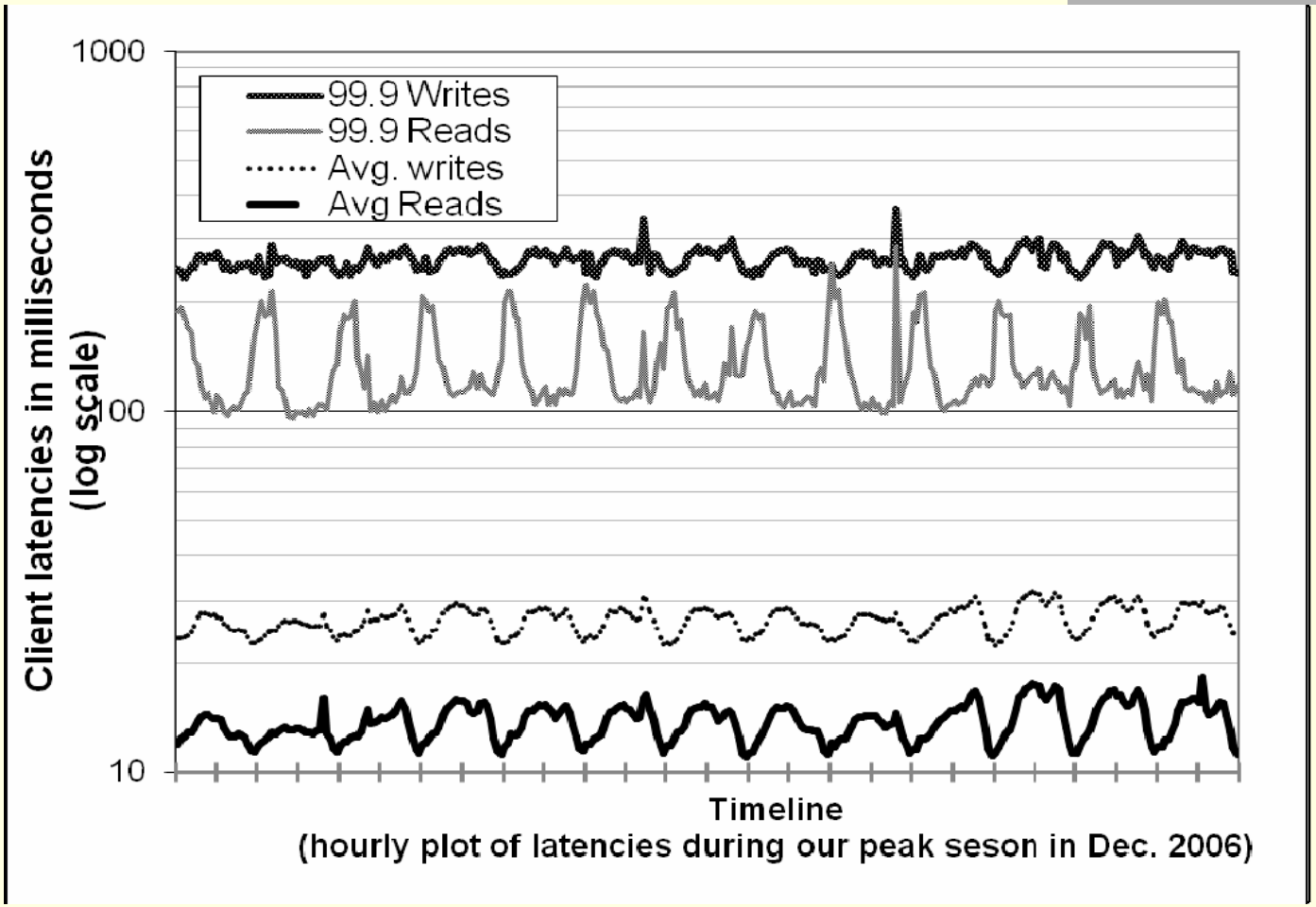
Other techniques

- Replica synchronization:
  - Merkle hash tree.
    - Leaves are hashes of contents
    - Parents are hashes of children
- Membership and Failure Detection:
  - Gossip

Implementation

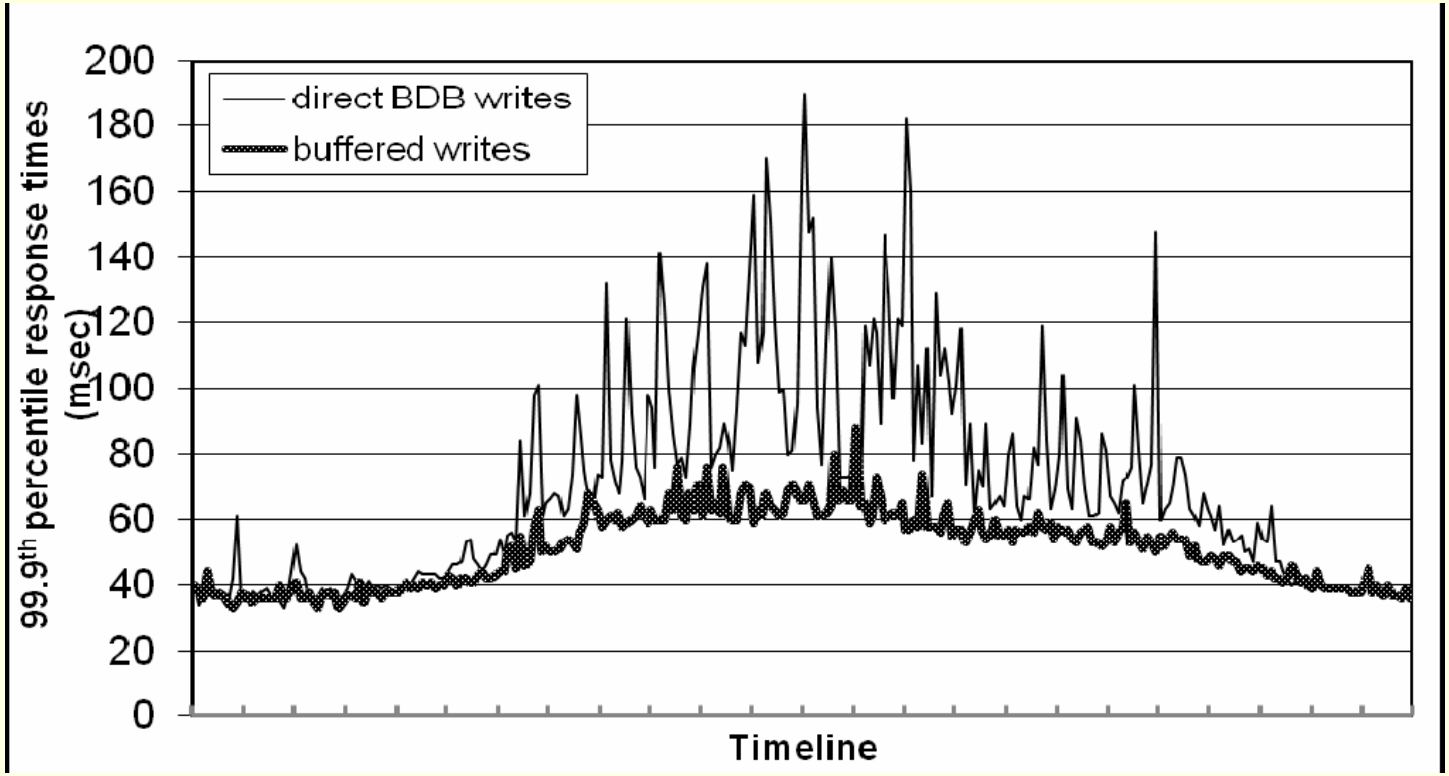
- Java
- Local persistence component allows for different storage engines to be plugged in:
  - Berkeley Database (BDB) Transactional Data Store: object of tens of kilobytes
  - MySQL: object of > tens of kilobytes
  - BDB Java Edition, etc.

Evaluation



X axis 12 hour ticks

Evaluation



Effect of buffering writes in memory
- writing to disk in batches; quicker read response
- loss of durability
- coordinator chooses one replica to write durable write synchronous, others async

Critique

- Would've liked more discussion on calibrating R,W,N values
- Testing partitioning with only S=30, N=3
- Table 2 lack of configuration details

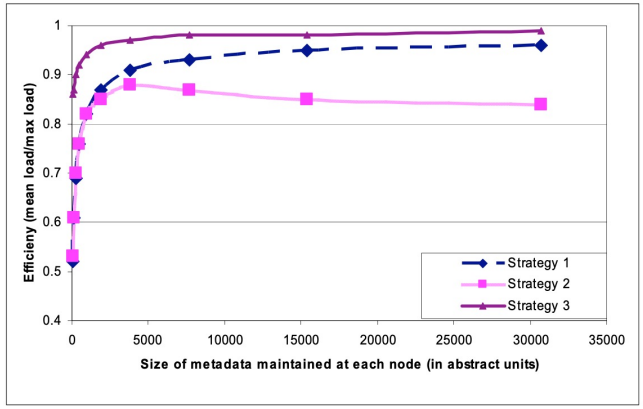


Figure 8: Comparison of the load distribution efficiency of different strategies for system with 30 nodes and N=3 with equal amount of metadata maintained at each node. The values of the system size and number of replicas are based on the typical configuration deployed for majority of our services.

Table 2: Performance of client-driven and server-driven coordination approaches.

	99.9th percentile read latency (ms)	99.9th percentile write latency (ms)	Average read latency (ms)	Average write latency (ms)
Server-driven	68.9	68.5	3.9	4.02
Client-driven	30.4	30.4	1.55	1.9

Contributions

- Highly available, sacrifice consistency
  - For small (<1MB files)
- Always writeable
- Reliability
- Service (customer) oriented architecture
- Sloppy quorum
- Combination of many techniques we previously discussed