Dynamo: Amazon's Highly Available Key-value Store

Distributed Storage Systems
CS 6464
2-12-09

presented by: Hussam Abu-Libdeh

Motivation

- In Modern Data Centers:
  - Hundreds of services
  - Thousands of commodity machines
  - Millions of customers at peak times
  - Performance + Reliability + Efficiency = $$^{10}
  - Outages are bad
    - Customers lose confidence , Business loses money
  - Accidents happen

Motivation

- Data center services must address
  - Availability
    - Service must be accessible at all times
  - Scalability
    - Service must scale well to handle customer growth & machine growth
  - Failure Tolerance
    - With thousands of machines, failure is the default case
  - Manageability
    - Must not cost a fortune to maintain

Today's Topic

- Discuss Dynamo
  - A highly available key-value storage system at Amazon
- Compare design decisions with other systems such as Porcupine

Agenda

- Overview
- Design Decisions/Trade-offs
- Dynamo's Architecture
- Evaluation

Insight

- Brewer's conjecture
  - Consistency, Availability, and Partition-tolerance
  - Pick 2/3
- Availability of online services == customer trust
  - Can not sacrifice that
- In data centers failures happen all the time
  - We must tolerate partitions

Eventual Consistency

- Many services do tolerate small inconsistencies
  - loose consistency ==> Eventual Consistency

- Agreement point:
  - Both Dynamo & Porcupine make this design decision

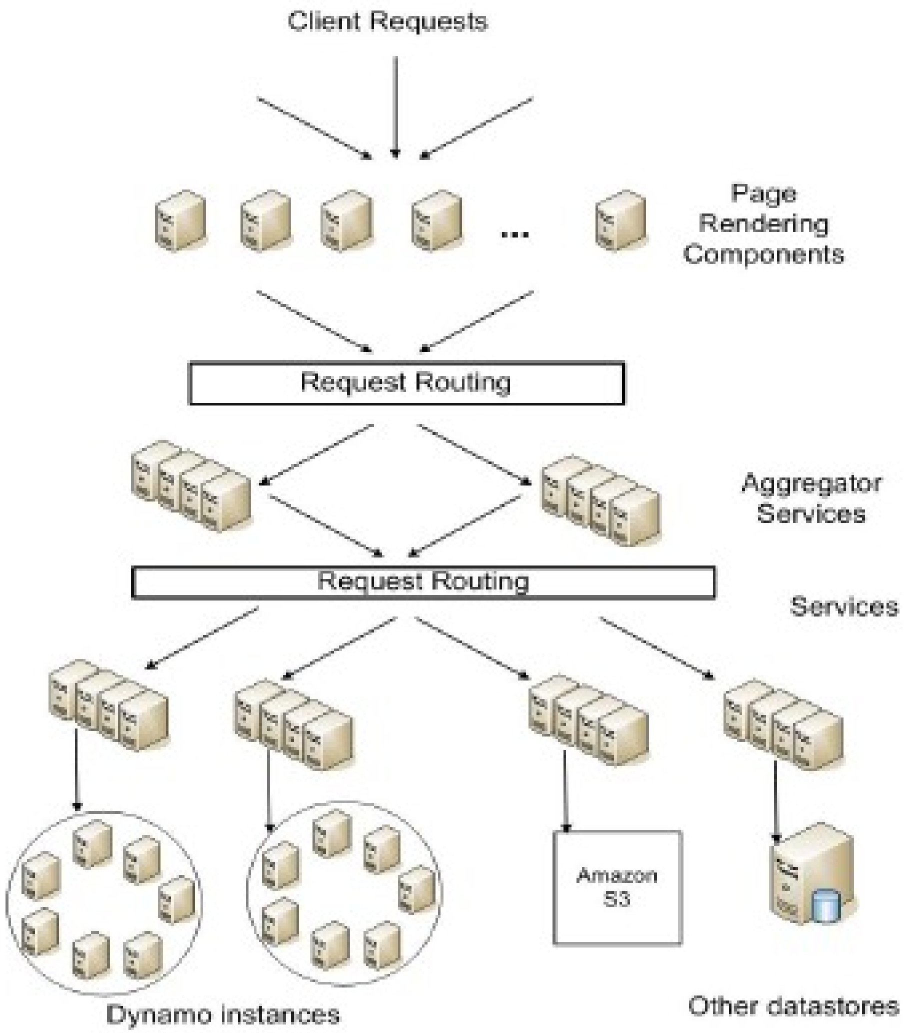
Dynamo's Assumptions

- Query Model:
  - Simple R/W ops to data with unique IDs
  - No ops span multiple records
  - Data stored as binary objects of small size
- ACID Properties:
  - Weaker (eventual) consistency
- Efficiency:
  - Optimize for the 99.9th percentile

Service Level Agreements (SLAs)

- Cloud-computing and virtual hosting contracts include SLAs
- Most are described in terms of mean, median, and variance of response times
  - Suffers from outliers
- Amazon targets optimization for 99.9% of the queries
  - Example: 300ms response-time for 99.9% of requests w/ peak load of 500 rpc

Service-oriented Architecture (SoA)



Design Decisions

- Incremental Scalability
  - Must be able to add nodes on-demand with minimal impact
  - In Dynamo: a chord-like scheme is used
  - In Porcupine: nodes are discovered and new groups are formed.
- Load Balancing & Exploiting Heterogeneity
  - In Dynamo a chord-like scheme is used
  - In Porcupine nodes track CPU/disk stats

Design Decisions

- Replication
  - Must do conflict-resolution
  - Porcupine is a little vague on conflict resolution
  - Two questions:
    - When ?
      - Solve on write to reduce read complexity
      - Solve on read and reduce write complexity
        - Dynamo is an “always writeable” data store
        - Fine for shopping carts and such services
    - Who ?
      - Data store
      - User application

Design Decisions

- Symmetry
  - All nodes are peers in responsibility
- Decentralization
  - Avoid single points of failure

- Both Dynamo & Porcupine agree on this

Dynamo Design Decisions

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.

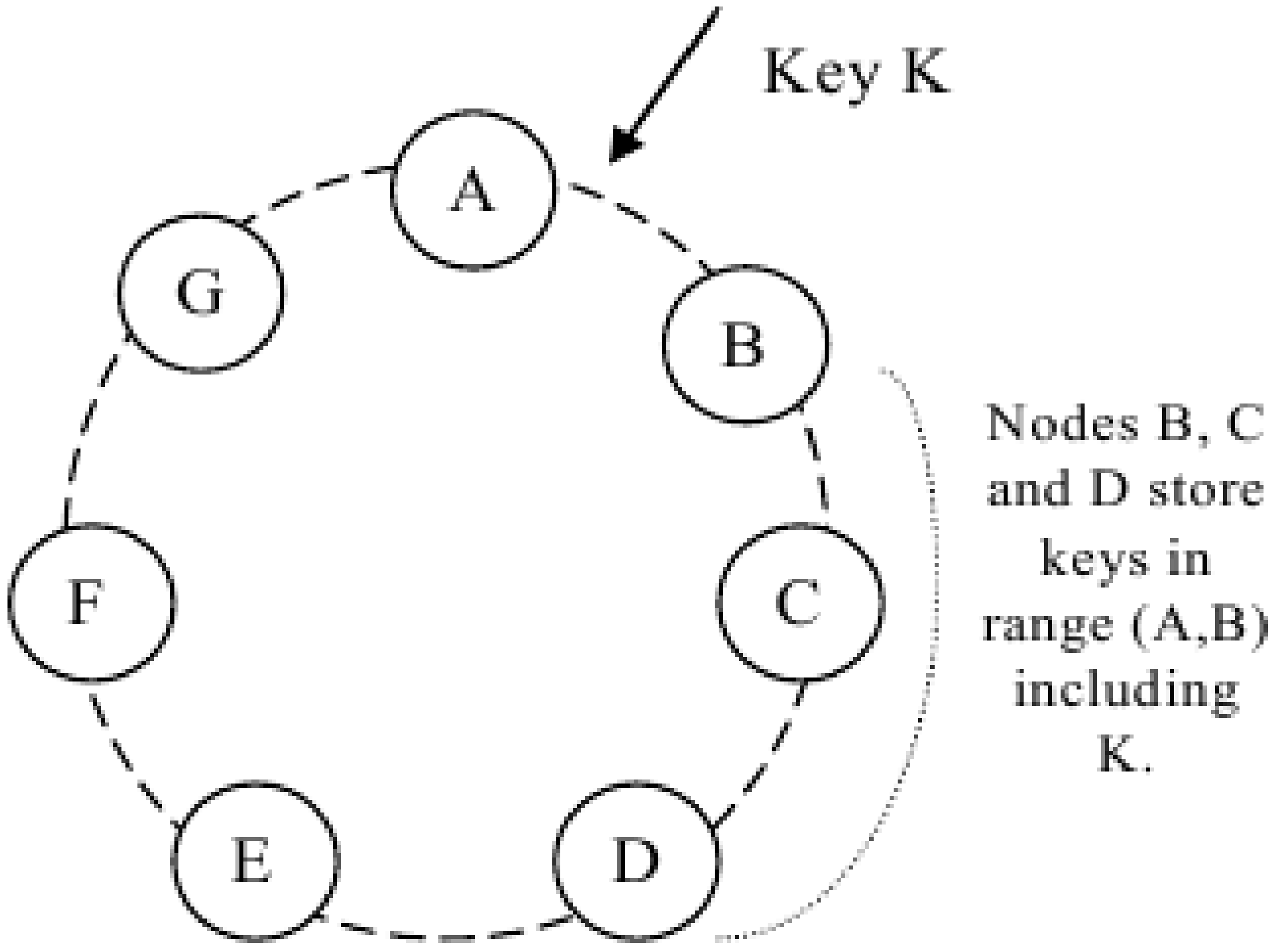
Dynamo's System Interface

- Only two operations
- put (key, context, object)
  - key: primary key associated with data object
  - context: vector clocks and history (needed for merging)
  - object: data to store
- get (key)

Data Partitioning & Replication

- Use consistent hashing
- Similar to Chord
  - Each node gets an ID from the space of keys
  - Nodes are arranged in a ring
  - Data stored on the first node clockwise of the current placement of the data key
- Replication
  - Preference lists of N nodes following the associated node

The Chord Ring



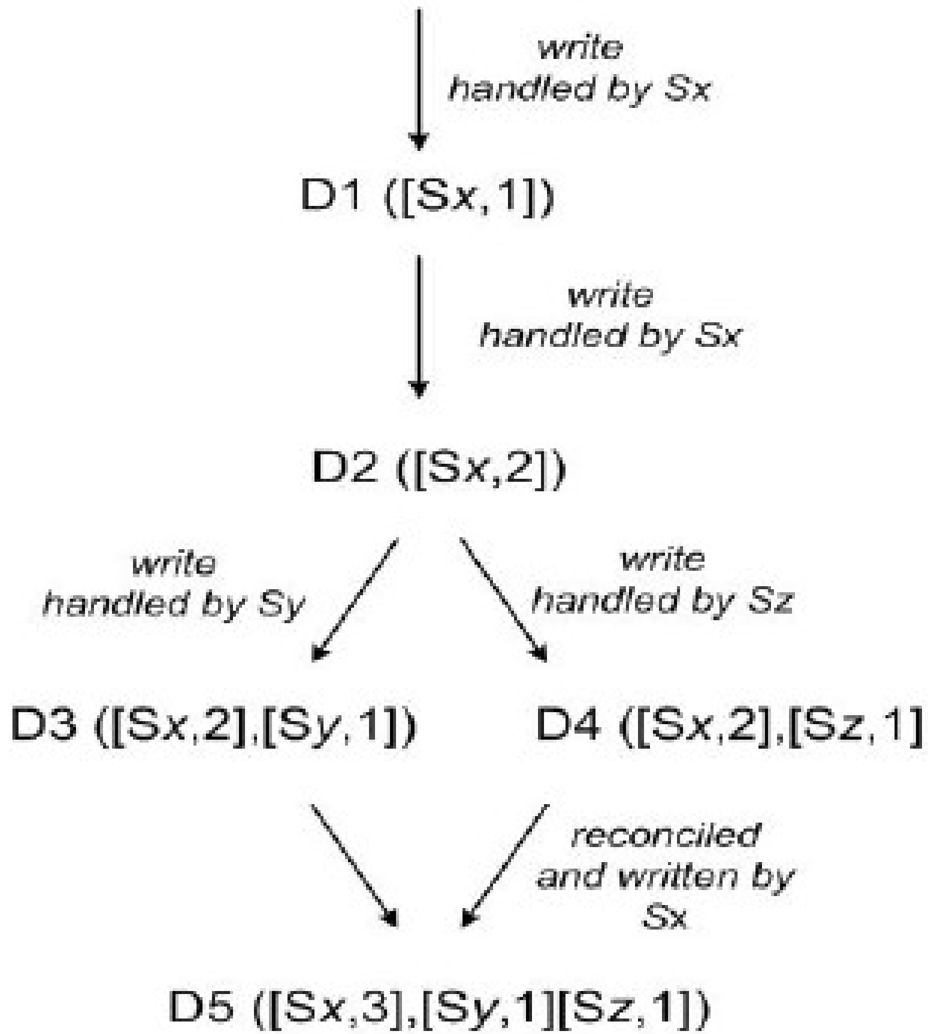
Virtual Nodes on the Chord Ring

- A problem with the Chord scheme
  - Nodes placed randomly on ring
  - Leads to uneven data & load distribution
- In Dynamo
  - Use "virtual nodes"
  - Each physical node has multiple virtual nodes
    - More powerful machines have more virtual nodes
  - Distribute virtual nodes across the ring

Data Versioning

- Updates generate a new timestamp
- Eventual consistency
  - Multiple versions of the same object might co-exist
- Syntactic Reconciliation
  - System might be able to resolve conflicts automatically
- Semantic Reconciliation
  - Conflict resolution pushed to application

Data Versioning



Execution of get() & put()

- Coordinator node is among the top N in the preference list
- Coordinator runs a R W quorum system
  - Identical to Weighted Voting System by Gifford ('79)
- R = read quorum
- W = write quorum
- R + W > N

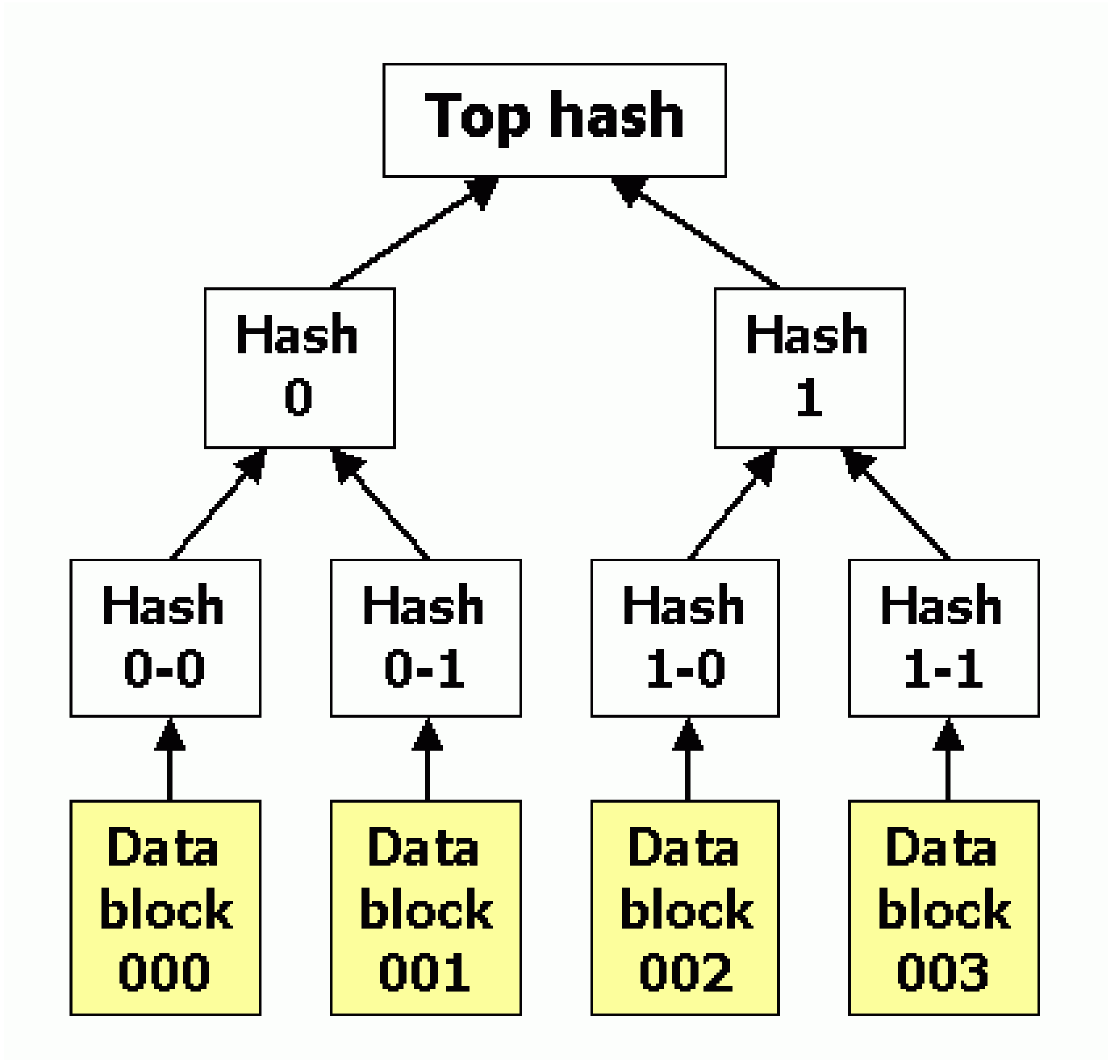
Handling Failures

- Temporary failures: Hinted Handoff
  - Offload your dataset to a node that follows the last of your preference list on the ring
  - Hint that this is temporary
  - Responsibility sent back when node recovers

Handling Failures

- Permanent failures: Replica Synchronization
  - Synchronize with another node
  - Use Merkle Trees

Merkle Tree



Top hash
Hash 0
Hash 1
Hash 0-0
Hash 0-1
Hash 1-0
Hash 1-1
Data block 000
Data block 001
Data block 002
Data block 003

Membership & Failure Detection

- Ring Membership
  - Use background gossip to build 1-hop DHT
  - Use external entity to bootstrap the system to avoid partitioned rings
- Failure Detection
  - Use standard gossip, heartbeats, and timeouts to implement failure detection

Evaluation

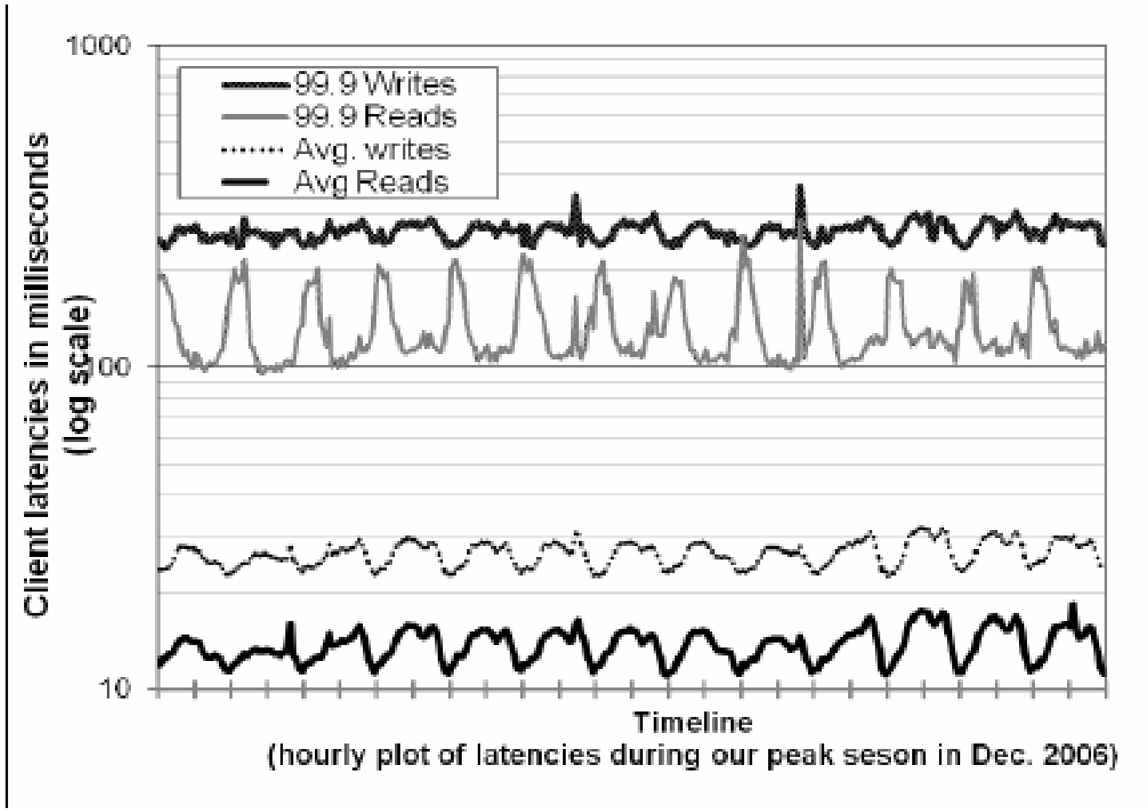


Figure 4: Average and 99.9 percentiles of latencies for read and write requests during our peak request season of December 2006. The intervals between consecutive ticks in the x-axis correspond to 12 hours. Latencies follow a diurnal pattern similar to the request rate and 99.9 percentile latencies are an order of magnitude higher than averages

Evaluation

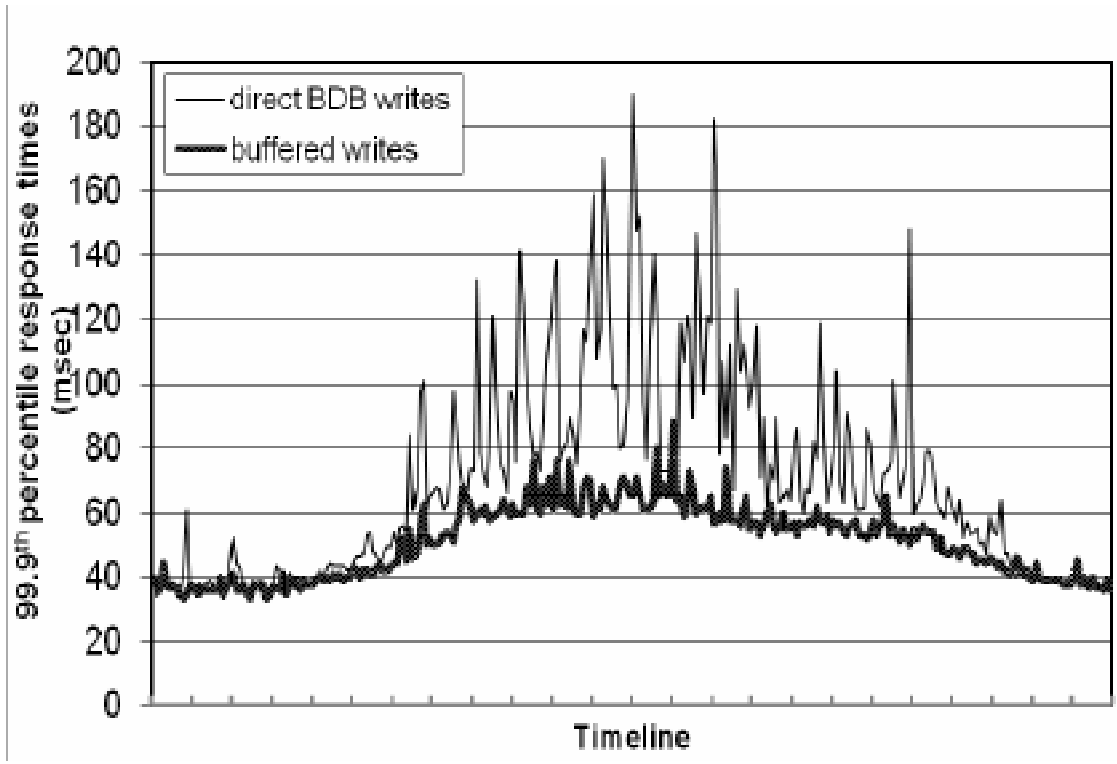


Figure 5: Comparison of performance of 99.9th percentile latencies for buffered vs. non-buffered writes over a period of 24 hours. The intervals between consecutive ticks in the x-axis correspond to one hour.

Evaluation

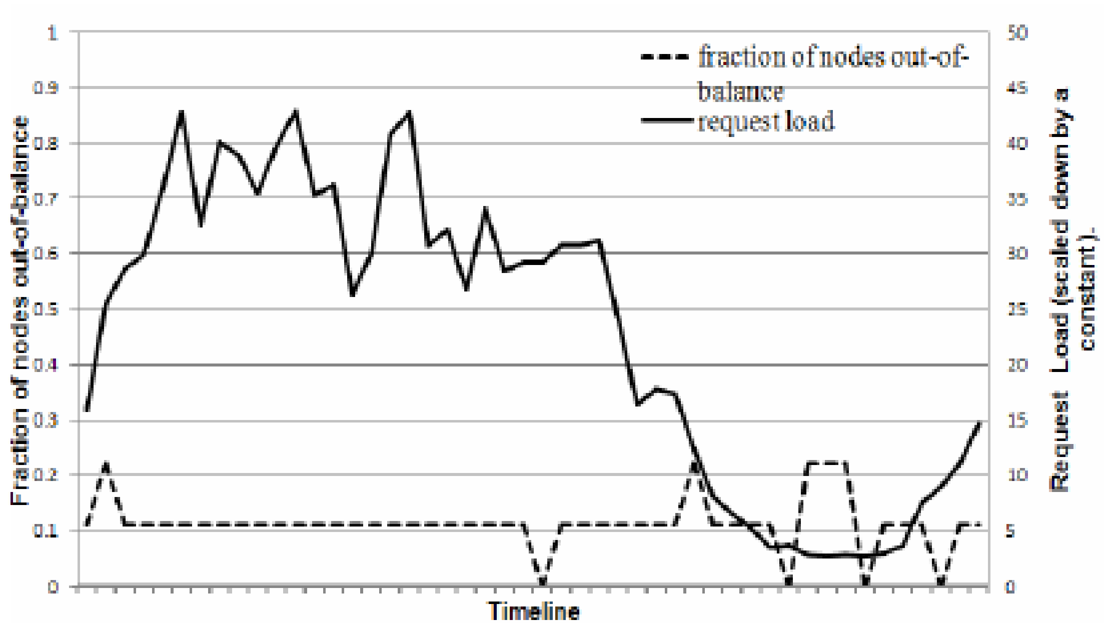


Figure 6: Fraction of nodes that are out-of-balance (i.e., nodes whose request load is above a certain threshold from the average system load) and their corresponding request load. The interval between ticks in x-axis corresponds to a time period of 30 minutes.

Thank You