Imperial College London

LSDS
Large-Scale Data & Systems Group

Dynamo: Amazon's Highly Available Key-Value Store

Peter Pietzuch
prp@doc.ic.ac.uk

Department of Computing
Imperial College London
http://lsds.doc.ic.ac.uk

Based on slides by Amazon, H. Abu-Libdeh, P. Krzyzanowski and R. Morris

Autumn 2020

Motivation: Services in Modern Data Centres

Hundreds of services

Thousands of commodity machines

Millions of customers at peak times

Performance + Reliability + Efficiency = $$$

Outages are bad

- Customers lose confidence, business loses money

Accidents happen

Service Requirements in DCs

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 default case

Manageability

- Must not cost a fortune to maintain

Dynamo's Design 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

- Optimise for 99.9th percentile

Dynamo's API

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)

CAP Theorem

Brewer's conjecture: CAP Theorem

- Consistency, Availability, and Partition-tolerance
- Pick 2 out of 3!

Availability of online services = customer trust

- Cannot sacrifice that

In data centres, failures happen all the time

- We must tolerate partitions

Eventual Consistency

Eventual consistency model

- Many services do tolerate small inconsistencies
- Lose consistency → eventual consistency

Dynamo: Sacrifice strong consistency for availability

Conflict resolution executed during ***read*** instead of ***write***,
ie “always writeable”

Service Level Agreements (SLAs)

Cloud computing and virtual hosting contracts include SLAs

Most described in terms of mean, median, and variance of response times

- Suffers from outliers

Amazon targets optimisation for 99.9% of queries

Apps should be able to configure Dynamo for desired latency & throughput

At least 99.9% of read/write operations must be performed within few hundred milliseconds

Design Considerations

Other principles:

Incremental scalability

- System should be able to grow by adding storage host (node) at a time

Symmetry

- Every node has same set of responsibilities

Decentralisation

- Favor decentralised techniques over central coordinators

Heterogeneity

- Workload partitioning should be proportional to capabilities of servers

Summary of Techniques in Dynamo

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.

Consistency & Availability I

Strong consistency & high availability cannot be achieved simultaneously

Optimistic replication techniques – eventually consistent model

- Propagate changes to replicas in the background
- Can lead to conflicting changes that have to be detected & resolved

When do you resolve conflicts?

- During writes: Traditional approach
  - Reject write if cannot reach all (or majority) of replicas
- During reads: Dynamo approach
  - Design for "always writable" data store - highly available
  - Read/write operations can continue even during network partitions
  - Rejecting customer updates won't be good experience
    – eg customer should always be able to add/remove items in shopping cart

Consistency & Availability II

Who resolves conflicts?

Choices: data store or application

Data store

- Application-unaware, so choices limited
- Eg simple policy, such as "last write wins”

Application

- Application aware of meaning of data
- Can do application-aware conflict resolution
- Eg merge shopping cart versions to get unified shopping cart

Fall back on "last write wins" if app doesn't want to bother

Data Partitioning & Replication

Use consistent hashing

- Regular hashing: change in # slots requires all keys to be remapped
- Consistent hashing
  - K/n keys need to be remapped, K = # keys, n = # slots

Similar to Chord -- Distributed Hash Table (DHT)

- Each node gets ID from space of keys
- Nodes arranged in ring
- Data stored on first node clockwise of current placement of data key

Replication

- Preference lists of N nodes following associated node

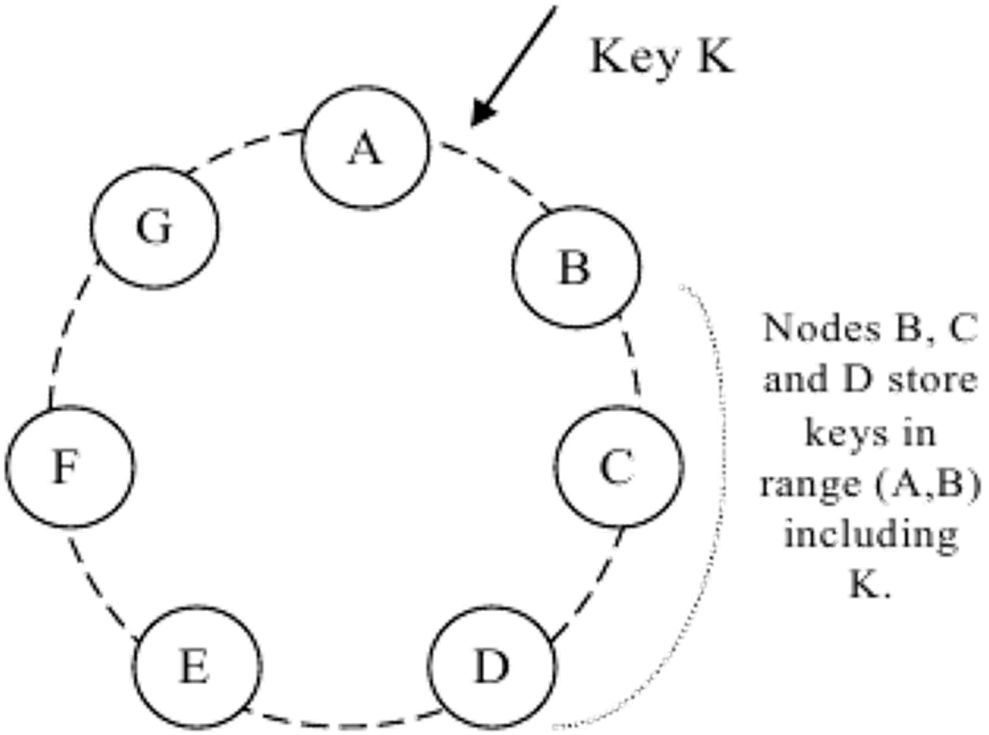
DHT: Overview

Provides peer-to-peer hash lookup:

- Lookup(key) $\rightarrow$ IP address

How does a DHT route lookups?

How does a DHT maintain routing tables?



DHT IDs

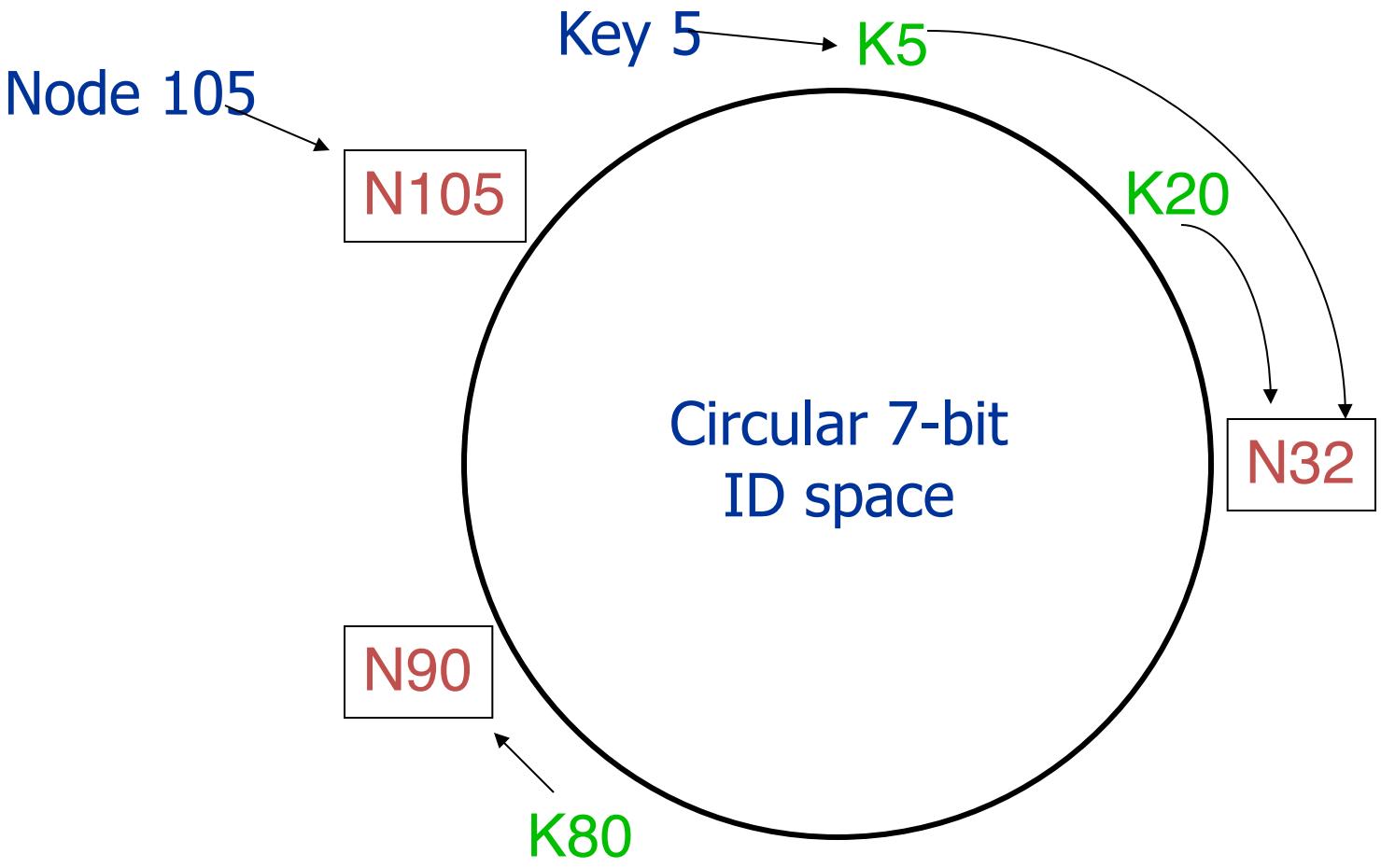
Key identifier = SHA-1(key)

Node identifier = SHA-1(IP address)

- Both are uniformly distributed
- Both exist in the same ID space

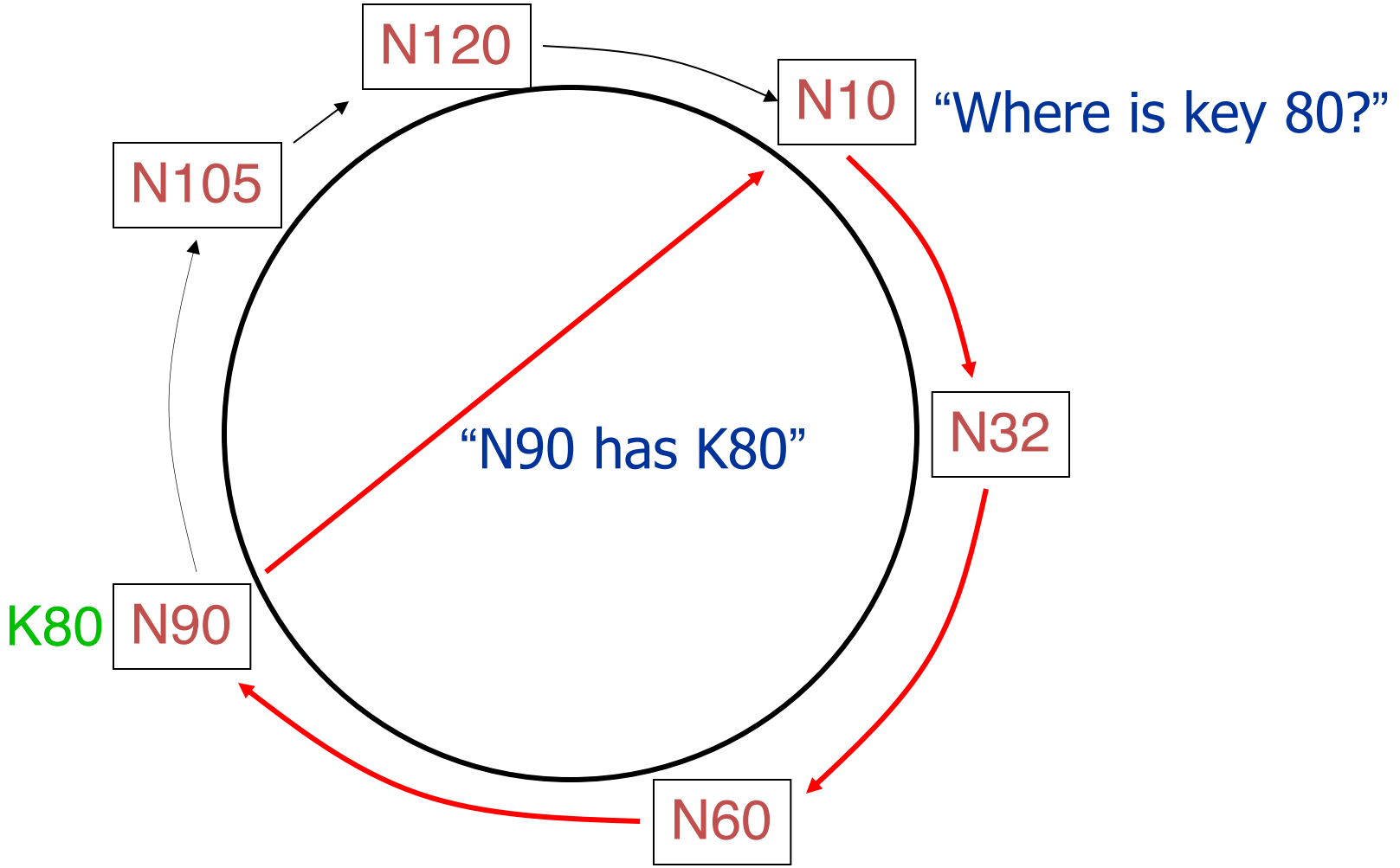
How to map key IDs to node IDs?

DHT: Consistent hashing [Karger 97]



Key stored at its successor: node with next higher ID

DHT: Basic lookup



DHT: Simple Lookup Algorithm

```
Lookup(my-id, key-id)
     n = my successor
     if my-id < n < key-id
          call Lookup(id) on node n    // next hop
     else
          return my successor          // done
```

Virtual Nodes on DHT Ring

Problem with DHT 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 ring

Advantages: balanced load distribution

- If node becomes unavailable, load evenly dispersed among available nodes
- If node added, it accepts equivalent amount of load from other nodes
- # of virtual nodes per system can be based on capacity of that node

Data Replication

Data replicated on N hosts (N is configurable)

- Key assigned coordinator node (via hashing)
- Coordinator in charge of replication

Coordinator replicates keys at N-1 clockwise successor nodes in ring

Data Versioning

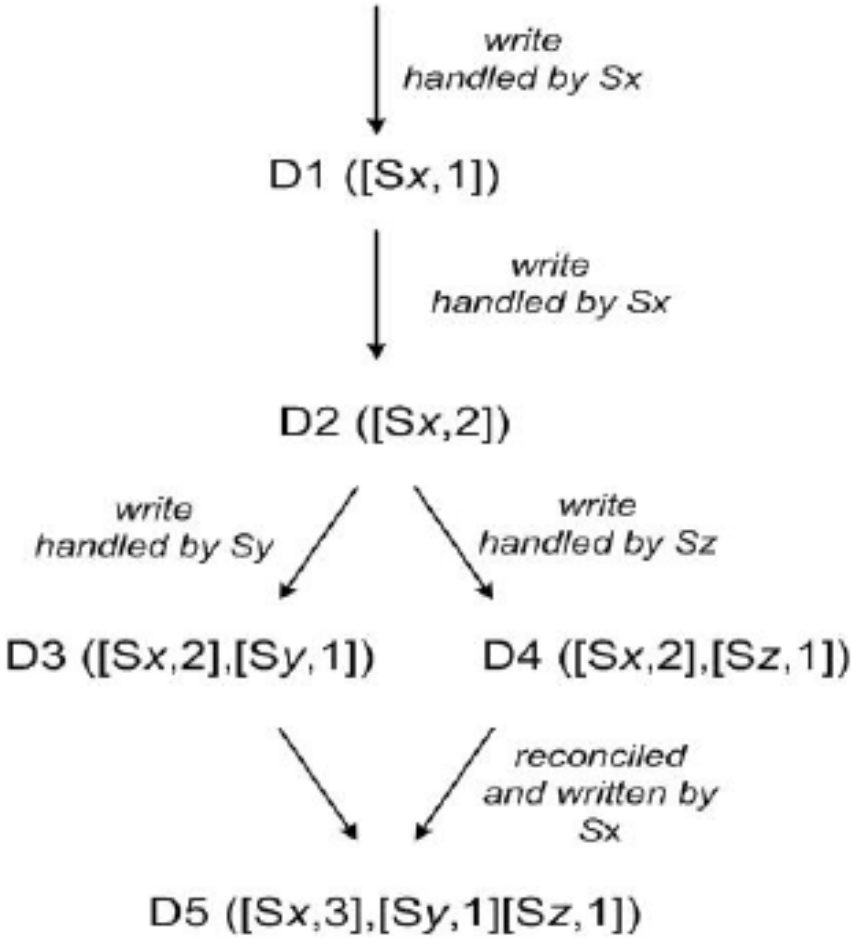
Not all updates may arrive at all replicas

Application-based reconciliation

- Each modification of data is treated as new version

Vector clocks used for versioning

- Capture causality between different versions of same object
- Vector clock is set of (node, counter) pairs
- Returned as context from a get() operation



Execution of get() & put()

Coordinator node is among top N in preference list

Coordinator runs R W quorum system

- Identical to Weighted Voting System by Gifford ('79)

R = read quorum

W = write quorum

R + W > N

Storage Nodes

Each node has three components:

1. Request coordination

- Coordinator executes read/write requests on behalf of requesting clients
- State machine contains all logic for identifying nodes responsible for key, sending requests, waiting for responses, retries, processing retries, packaging response
- Each state machine instance handles one request

2. Membership and failure detection

3. Local persistent storage

- Different storage engines may be used depending on needs:
  - Berkeley Database (BDB) Transactional Data Store (most popular)
  - BDB Java Edition
  - MySQL (for large objects)
  - In-memory buffer with persistent backing store

Handling Failures

Temporary failures: Hinted Handoff

- Offloads data to node that follows the last node in preference list on ring
- Hint that this is temporary
- Responsibility to send back when node recovers

Permanent failures: Replica Synchronisation

- Synchronise with another node
- Use Merkle Trees

Merkle Trees

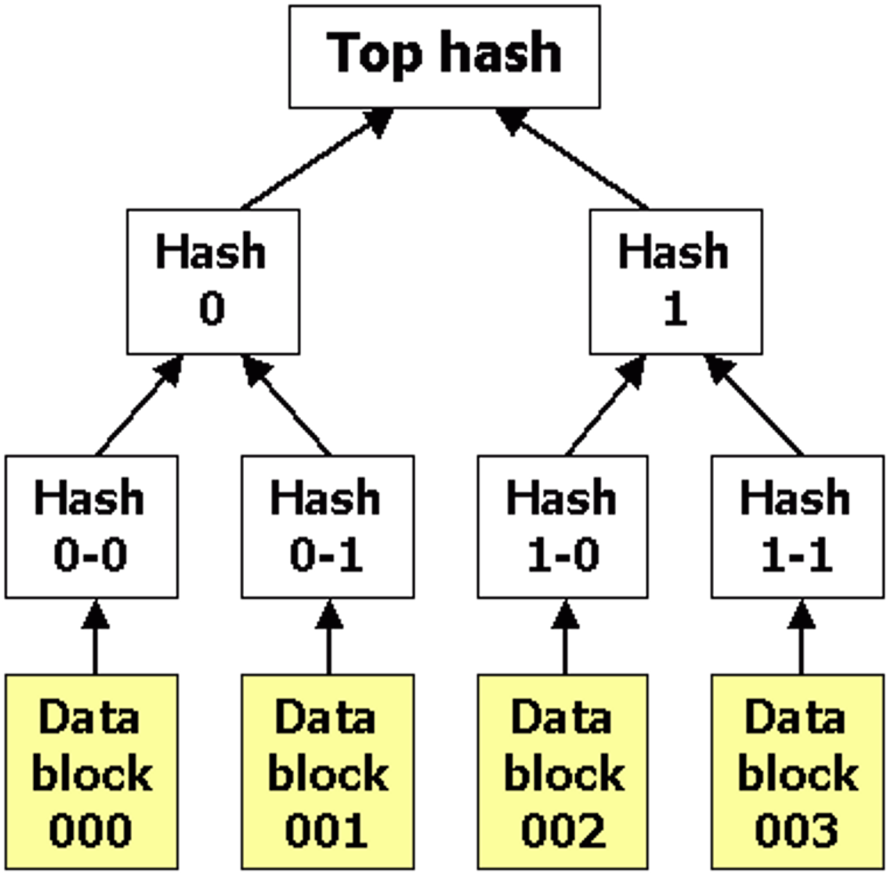
Tree consisting of hashes

- Leaves: hashes of individual keys
- Parents: hashes of their children

Advantage: parts can be checked without need to compare whole tree

Example:

- two root hashes equal → trees equal



Membership & Failure Detection

Ring Membership

- Use background gossip to build 1-hop DHT
- Use external entity to bootstrap the system to avoid partitioned rings

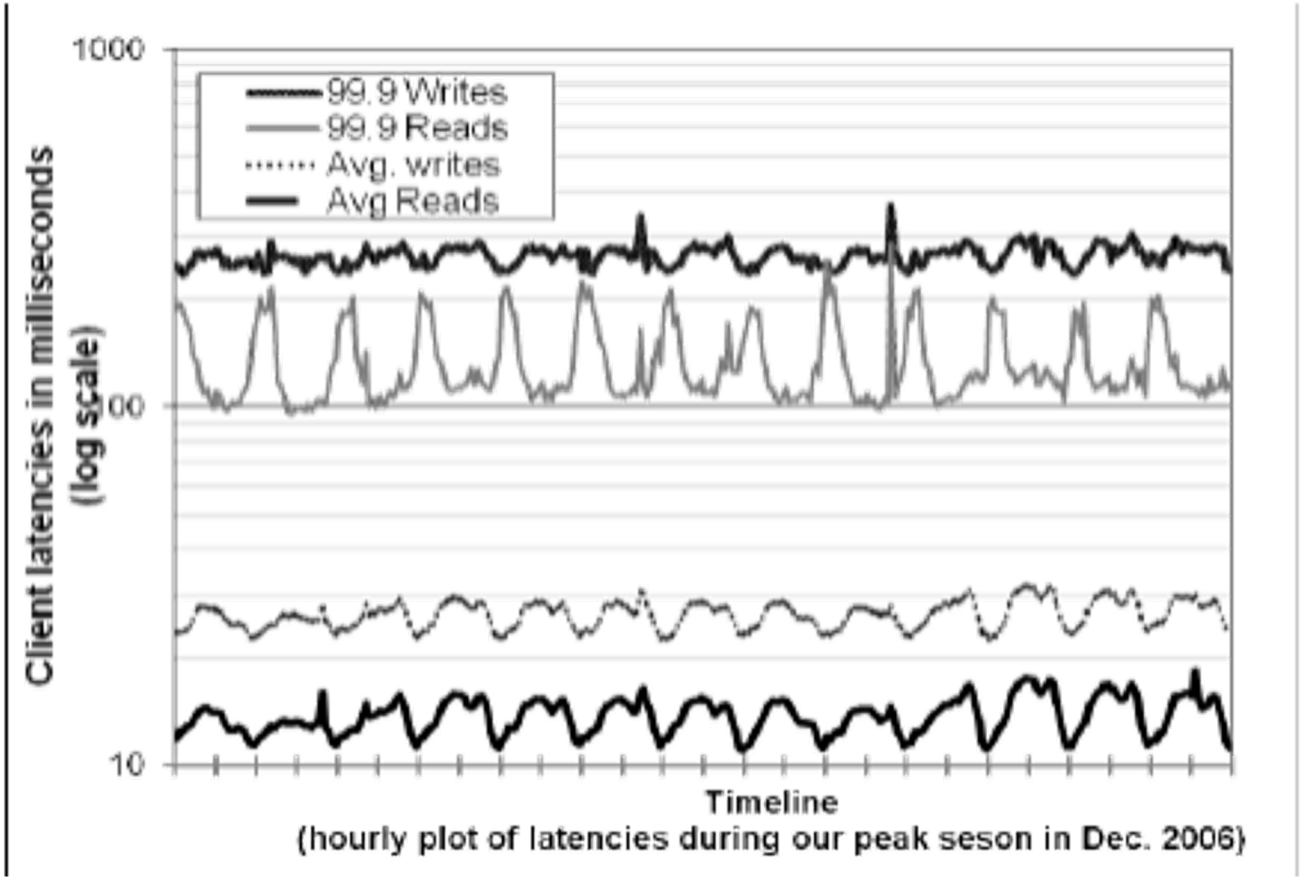
Failure Detection

- Use standard gossip, heartbeats, and timeouts to implement failure detection

Evaluation: Request Latency

All experiments run in real production environment

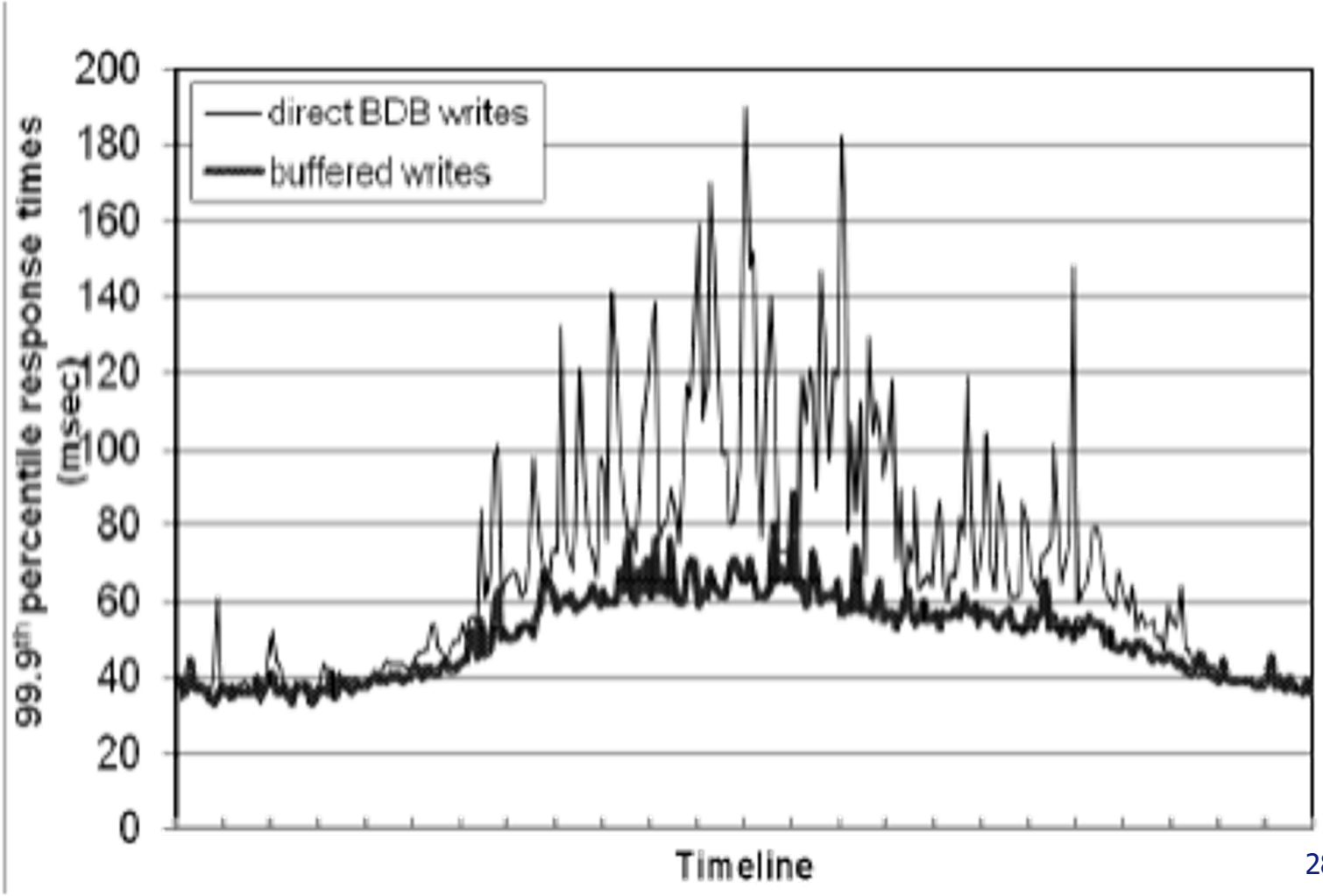
- Graph: Average and 99.9 percentile latencies
- As seen: upper bound is good, average is really good



Evaluation: Buffered Writes

Graph: Comparison between buffered and un-buffered writes

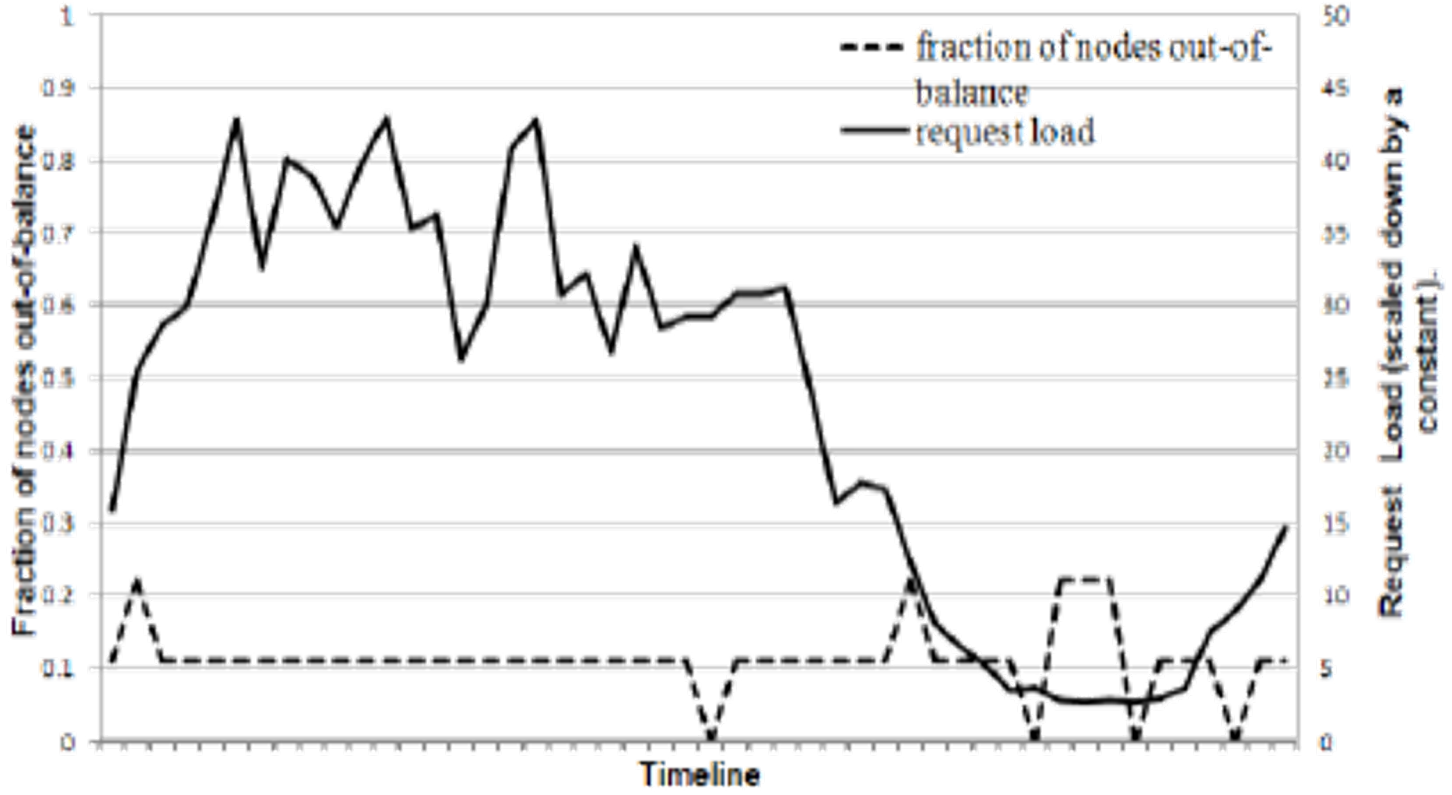
- As seen: Buffered writes boost performance



Evaluation: Load Balancing

Graph: Node imbalance and request load

- As seen: the higher the request, the better the balance



Comparison with Google Bigtable

Dynamo targets apps that only need key/value access with primary focus on high availability:

Key-value store versus column-store (column families and columns within them)

Bigtable: distributed DB built on GFS

Dynamo: distributed hash table

Updates are not rejected even during network partitions or server failures