P2P: Storage

Overall outline

- (Relatively) chronological overview of P2P areas:
  - What is P2P?
  - Filesharing → structured networks → storage → the cloud
- Dynamo
  - Design considerations
  - Challenges and design techniques
  - Evaluation, takeaways, and discussion
- Cassandra
  - Vs Dynamo
  - Notable design choices

Background: P2P

- Formal definition?
- Symmetric division of responsibility and functionality
- Client-server: Nodes both request and provide service
- Each node enjoys conglomerate service provided by peers

- Can offer better load distribution, fault-tolerance, scalability...

- On a fast rise in the early 2000’s

Background:P2P filesharing & unstructured networks

- Napster (1999)
- Gnutella (2000)
- FreeNet (2000)

- Key challenges:
    - Decentralize content search and routing

Background: P2P structured networks

- CAN (2001)
- Chord (2001)
- Pastry (2001)
- Tapestry (2001)

- More systematic+formal
- Key challenges:
    - Routing latency
    - Churn-resistance
    - Scalability

Background: P2P Storage

- CAN (2001)
- Chord (2001) → DHash++ (2004)
- Pastry (2001) → PAST (2001)
- Tapestry (2001) → Pond (2003)
- Chord/Pastry → Bamboo (2004)

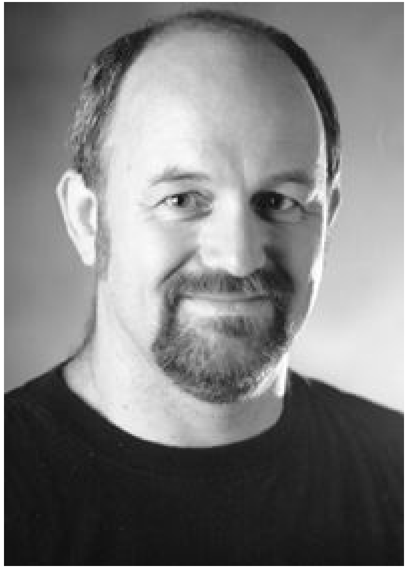
- Key challenges:
  - Distrusting peers
  - High churn rate
  - Low bandwidth connections

Background: P2P on the Cloud

- In contrast:
    - Single administrative domain
    - Low churn (only due to permanent failure)
    - High bandwidth connections

Dynamo: Amazon's Highly Available Key-value Store

SOSP 2007: Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels



Best seller lists, shopping carts, etc.
Also proprietary service@AWS

Werner Vogels: Cornell → Amazon

Interface

Put (key, context, object) → Success/Fail

Success of (set of values, context)/Fail ← Get (key)

Dynamo’s design considerations

- Strict performance requirements, tailored closely to the cloud environment
- Very high write availability
    - C**AP**
    - No isolation, single-key updates
- 99.9th percentile SLA system
- Regional power outages are tolerable → Symmetry of function
- Incremental scalability
    - Explicit node joins
    - Low churn rate assumed

List of challenges:

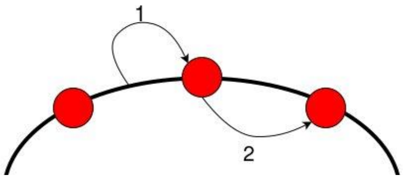
1. Incremental scalability and load balance
2. Flexible durability
3. High write availability
4. Handling temporary failure
5. Handling permanent failure
6. Membership protocol and failure detection

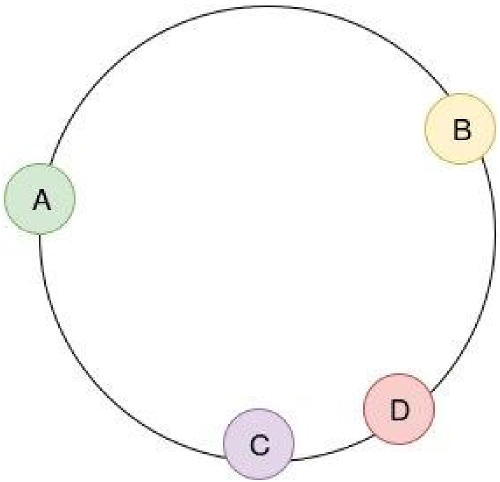
List of challenges:

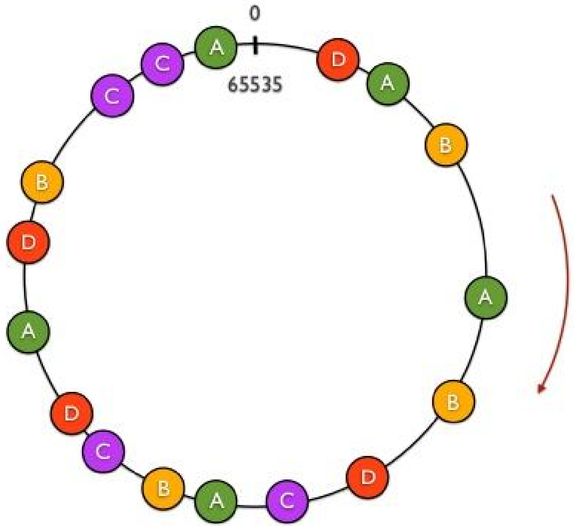
1. **Incremental scalability and load balance**
   - **Adding one node at a time**
   - **Uniform node-key distribution**
   - **Node heterogeneity**
2. Flexible durability
3. High write availability
4. Handling temporary failure
5. Handling permanent failure
6. Membership protocol and failure detection

Incremental scalability and load balance

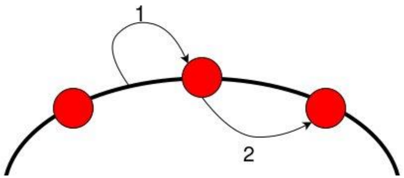
- Consistent Hashing
- Virtual nodes (as seen in Chord):
  Node gets several, smaller key ranges instead of one big one







Incremental scalability and load balance



- Consistent Hashing
- Virtual nodes (as seen in Chord):
  Node gets several, smaller key ranges instead of one big one

- Benefits:
  - More uniform key-node distribution
  - Node join and leaves requires only neighbor nodes
  - Variable number of virtual nodes per physical node

List of challenges:

1. Incremental scalability and load balance
2. **Flexible durability**
    - **Latency vs durability**
3. High write availability
4. Handling temporary failure
5. Handling permanent failure
6. Membership protocol and failure detection

Flexible Durability

- Key preference list
- N - # of healthy nodes coordinator references
- W - min # of responses for put
- R - min # of responses for get
- R, W, N tradeoffs
    - W↑ ⇒ Consistency↑, latency↑
    - R↑ ⇒ Consistency↑, latency↑
    - N↑ ⇒ Durability↑, load on coord↑
    - R + W > N : Read-your-writes

Flexible Durability

- Key preference list
- N - # of healthy nodes coordinator references
- W - min # of responses for put
- R - min # of responses for get
- R, W, N tradeoffs

- Benefits:
    - Tunable consistency, latency, and fault-tolerance
    - Fastest possible latency out of the N healthy replicas every time
    - Allows hinted handoff

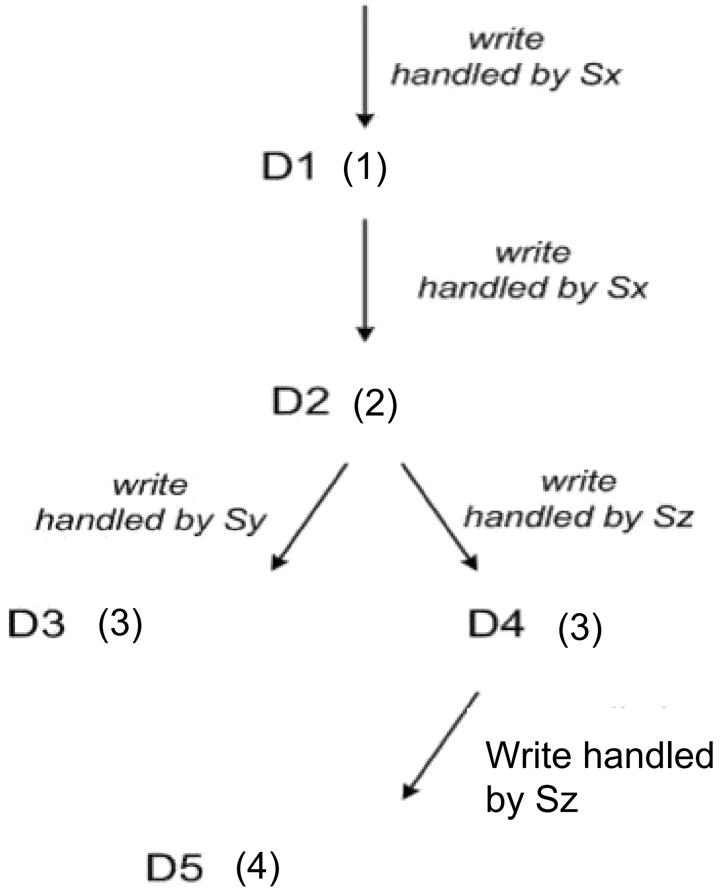
List of challenges:

1. Incremental scalability and load balance
2. Flexible durability
3. **High write availability**
    - **Writes cannot fail or delay because of consistency management**
4. Handling temporary failure
5. Handling permanent failure
6. Membership protocol and failure detection

Achieving High Write Availability

- Weak consistency
  - Small W → outdated objects lying around
  - Small R → outdated objects reads
- Update by itself is meaningful and should preserve
- Accept all updates, even on outdated copies
- Updates on outdated copies ⇒ DAG object was-before relation
- Given two copies, should be able to tell:
  - Was-before relation → Subsume
  - Independent → preserve both
- But single version number forces total ordering (Lamport clock)

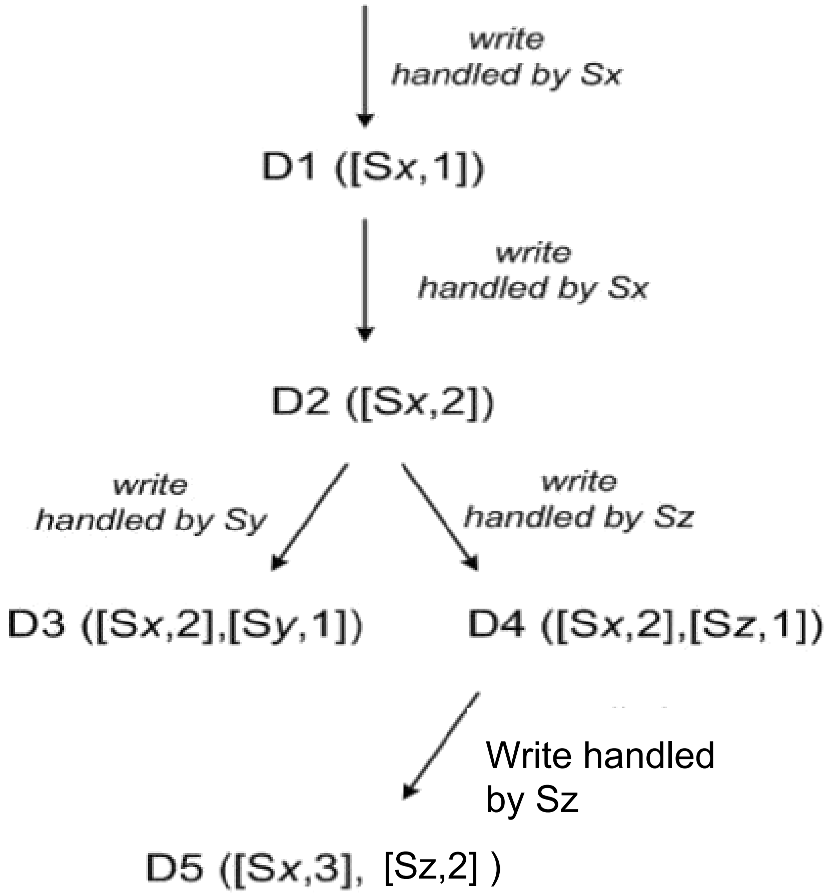
Hiding Concurrency



Achieving High Write Availability

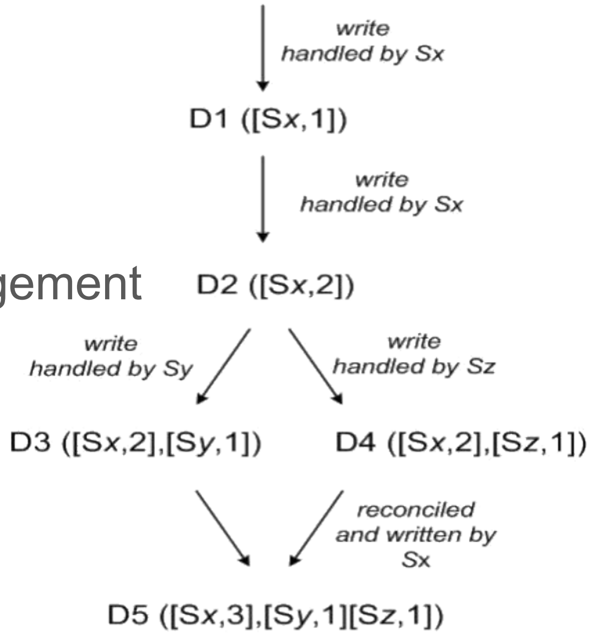
- Weak consistency
  - Small W → outdated objects lying around
  - Small R → outdated objects reads
- Update by itself is meaningful and should preserve
- Accept all updates, even on outdated copies
- Updates on outdated copies ⇒ DAG object was-before relation
- Given two copies, should be able to tell:
  - Was-before relation → Subsume
  - Independent → preserve both
- But single version number forces total ordering (Lamport clock)
- *Vector clock: version number per key per machine, preserves concurrence*

Showing Concurrency

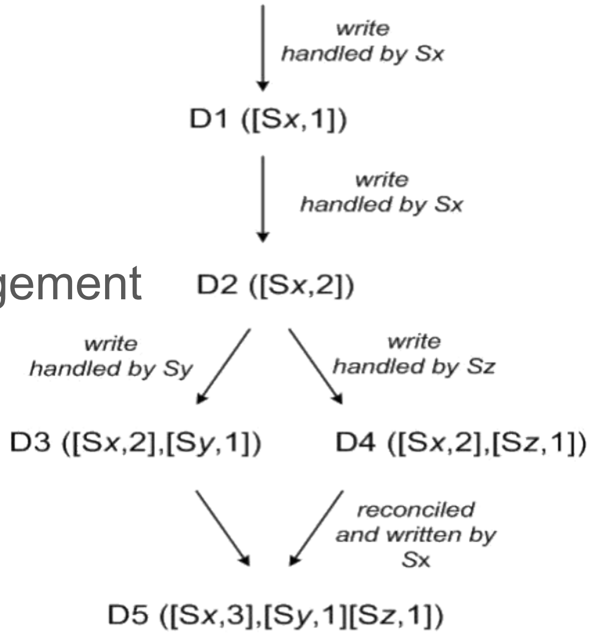


Achieving High Write Availability

- No write fail or delay because of consistency management
- Immutable objects + vector clock as version
- Automatic subsumption reconciliation
- Client resolves unknown relation through *context*



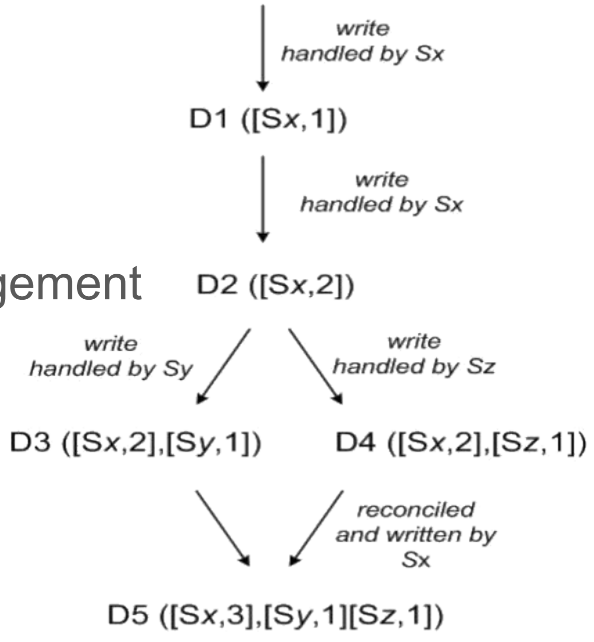
Achieving High Write Availability



- No write fail or delay because of consistency management
- Immutable objects + vector clock as version
- Automatic subsumption reconciliation
- Client resolves unknown relation through *context*

- `Read (k) = {D3, D4}, Opaque_context(D3(vector), D4(vector))`
- `/* Client reconciles D3 and D4 into D5 */`
- `Write (k, Opaque_context(D3(vector), D4(vector), D5)`
- `Dynamo creates a vector clock that subsumes clocks in context`

Achieving High Write Availability



- No write fail or delay because of consistency management
- Immutable objects + vector clock as version
- Automatic subsumption reconciliation
- Client resolves unknown relation through *context*

- Benefits:
  - Aggressively accept all updates
- Problem:
  - Client-side reconciliation
  - Reconciliation not always possible
  - Must read after each write to chain a sequence of updates

List of challenges:

1. Incremental scalability and load balance
2. Flexible durability
3. High write availability
4. **Handling temporary failure**
    - **Writes cannot fail or delay because of temporary inaccessibility**
5. Handling permanent failure
6. Membership protocol and failure detection

Handling Temporary Failures

- No write fail or delay because of temporary inaccessibility
- Assume node will be accessible again soon
- coordinator walks off the N-preference list
- References node N+a on list to reach W responses
- N+a keeps passes object back to the hinted node at first opportunity

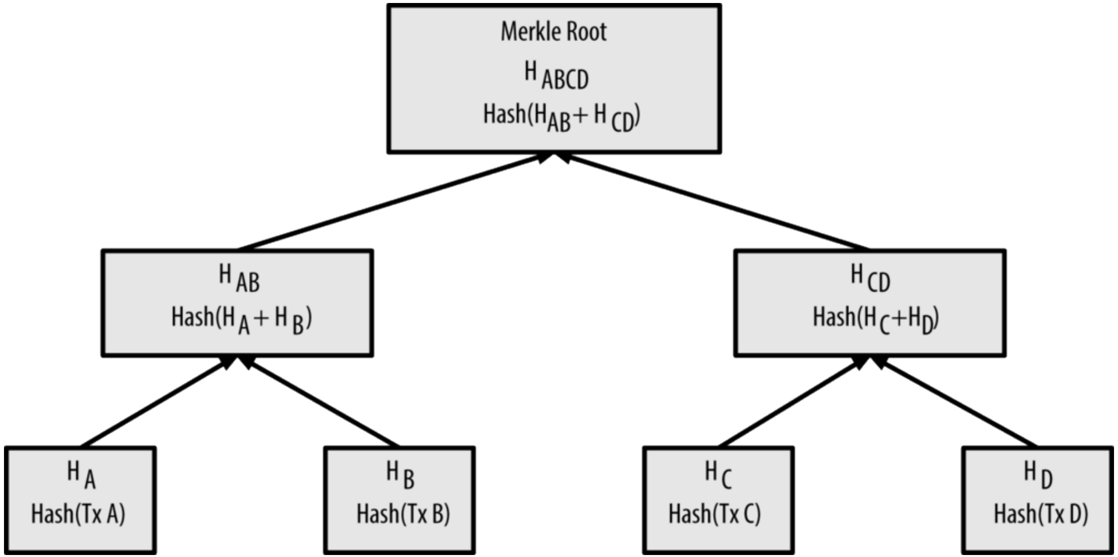
- Benefits:
  - Aggressively accept all updates

List of challenges:

1. Incremental scalability and load balance
2. Flexible durability
3. High write availability
4. Handling temporary failure
5. **Handling permanent failure**
    - **Maintain eventual consistency with permanent failure**
6. Membership protocol and failure detection

Permanent failures in Dynamo

- Use anti-entropy between replicas
- Merkle Trees
- Speeds up subsumption



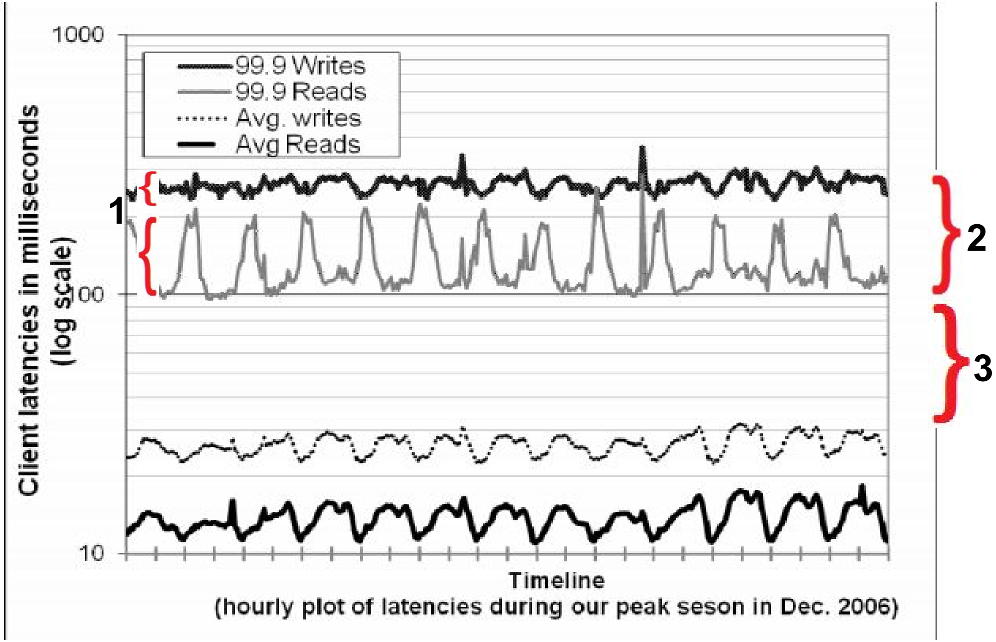
List of challenges:

1. Incremental scalability and load balance
2. Flexible durability
3. High write availability
4. Handling temporary failure
5. Handling permanent failure
6. **Membership protocol and failure detection**

Membership and failure detection in Dynamo

- Anti-entropy to reconcile membership (eventually consistent view)
- Constant time lookup
- Explicit node join and removal
- Seed nodes to avoid logical network partitions
- Temporary inaccessibility detected through timeouts and handled locally

Evaluation



1 low variance in read and write latencies
2 Writes directly to memory, cache reads
3 Shows skewed distribution of latency

Evaluation

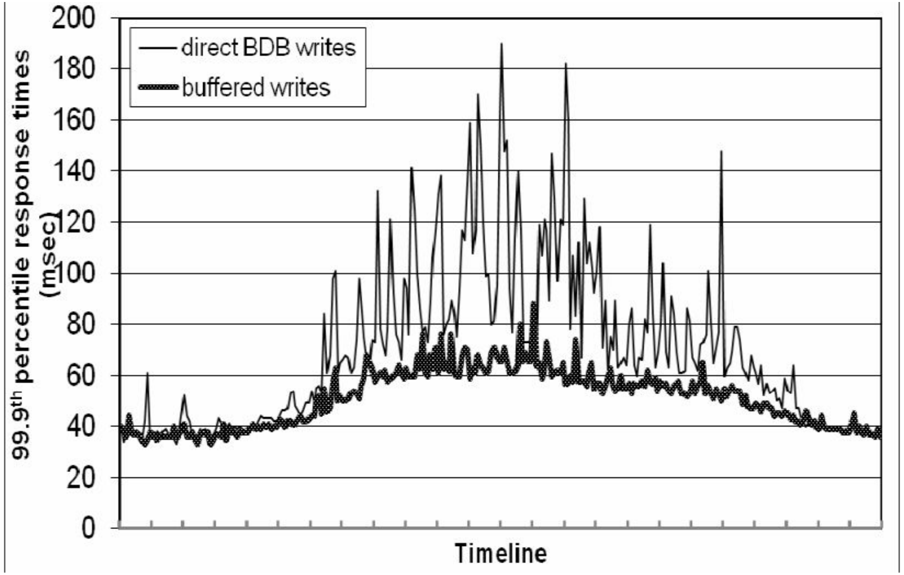
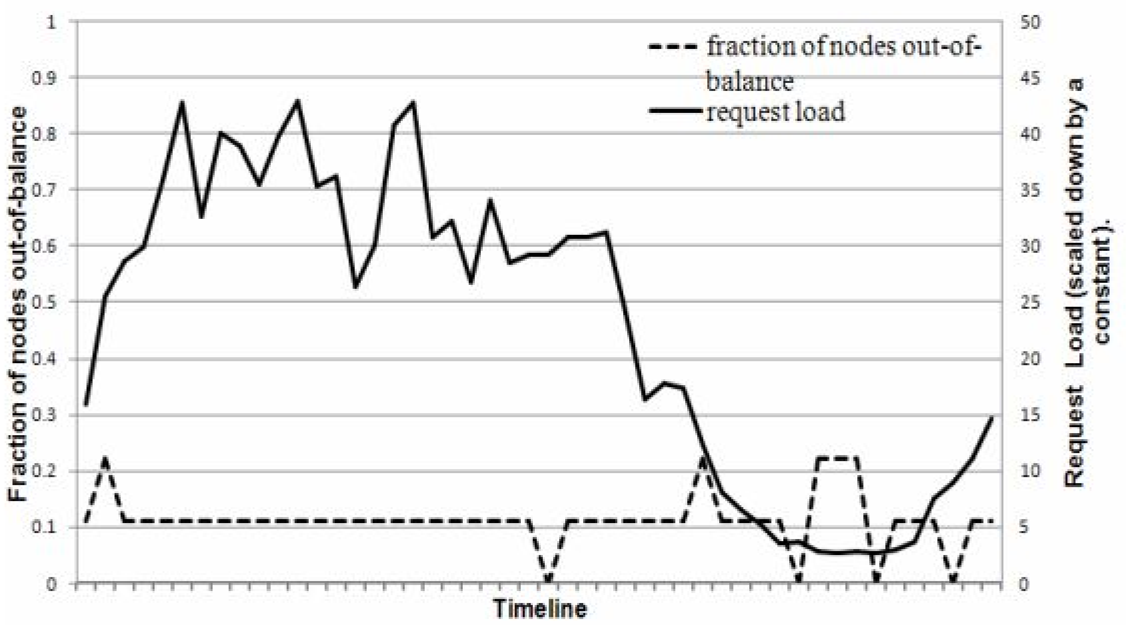


Figure 5: Comparison of performance of 99.9th percentile latencies for buffered vs. non-buffered writes over a period of 24 hours. The intervals between consecutive ticks in the x-axis

- Lowers write latency
- smoothes 99.9th percentile extremes
- At a durability cost

Evaluation



- lower loads:
Fewer popular keys

- In higher loads:
Many popular keys roughly equally among the nodes, most node don’t deviate more than 15%

Imbalance = 15% away from average node load

Takeaways

- User gets knobs to balance durability, latency, and consistency
- P2P techniques can be used in the cloud environment to produce highly-available services
- Instead resolving consistency for all clients at a universally higher latency, let each client resolve their own consistency individually
- ∃ Industry services that require the update to always preserve

Cassandra - A Decentralized Structured Storage System

Avinash Lakshman, Prashant Malik

Avinash from Dynamo team

Used in multiple internals in FB, including inbox search

Interface / Data Model

- Borrows from BigTable
- Rows are keys
- Columns are common key attributes
- Column families and super columns

In Relation to Dynamo

- Implement very similar systems
- “*A write operation in Dynamo also requires a read to be performed for managing the vector timestamps … limiting [when] handling a very high write throughput.*”
- Instead of virtual nodes, moves tokens
  - “*Makes the design and implementation very tractable … deterministic choices about load balancing*”
- Consistency option between quorum or single-machine and anti-entropy
- Automate bootstraping through ZooKeeper

Results

Latency Stat	Search Interactions	Term Search
Min	7.69ms	7.78ms
Median	15.69ms	18.27ms
Max	26.13ms	44.41ms

Thank you for listening!