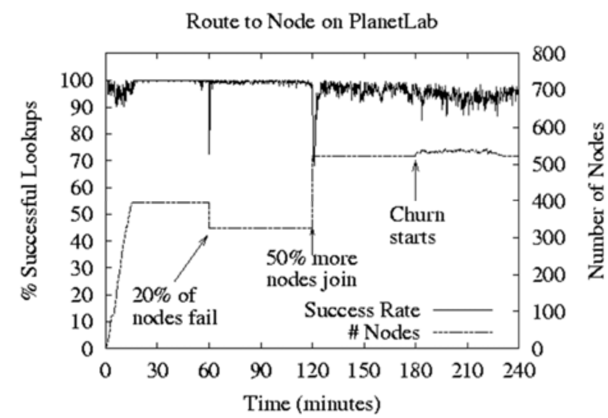
EECS 262a
Advanced Topics in Computer Systems
Lecture 22

P2P Storage: Dynamo
April 13th, 2016

John Kubiatowicz
Electrical Engineering and Computer Sciences
University of California, Berkeley

http://www.eecs.berkeley.edu/~kubitron/cs262

Reprise: Stability under churn (Tapestry)



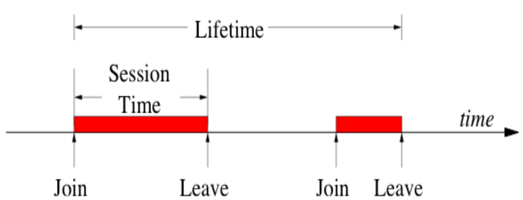
(May 2003: 1.5 TB over 4 hours)

DOLR Model generalizes to many simultaneous apps

Churn (Optional Bamboo paper last time)

Chord is a "scalable protocol for lookup in a dynamic peer-to-peer system with frequent node arrivals and departures"

-- Stoica et al., 2001



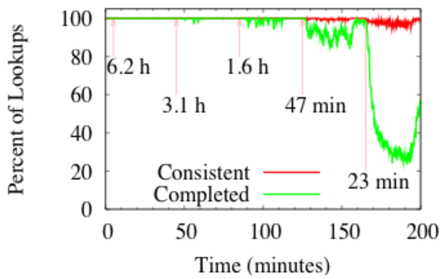
Authors	Systems Observed	Session Time
SGG02	Gnutella, Napster	50% < 60 minutes
CLL02	Gnutella, Napster	31% < 10 minutes
SW02	FastTrack	50% < 1 minute
BSV03	Overnet	50% < 60 minutes
GDS03	Kazaa	50% < 2.4 minutes

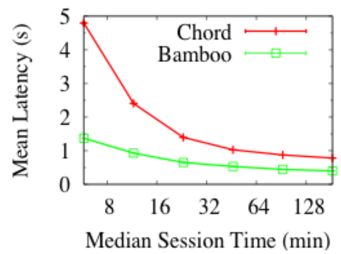
A Simple *lookup* Test

- Start up 1,000 DHT nodes on ModelNet network
  – Emulates a 10,000-node, AS-level topology
  – Unlike simulations, models cross traffic and packet loss
  – Unlike PlanetLab, gives reproducible results
- Churn nodes at some rate
  – Poisson arrival of new nodes
  – Random node departs on every new arrival
  – Exponentially distributed session times
- Each node does 1 lookup every 10 seconds
  – Log results, process them after test

Early Test Results

- Tapestry had trouble under this level of stress
  - Worked great in simulations, but not as well on more realistic network
  - Despite sharing almost all code between the two!
- Problem was not limited to Tapestry consider Chord:





Handling Churn in a DHT

- Forget about comparing different impls.
  - Too many differing factors
  - Hard to isolate effects of any one feature
- Implement all relevant features in one DHT
  - Using Bamboo (similar to Pastry)
- Isolate important issues in handling churn
  1. Recovering from failures
  2. Routing around suspected failures
  3. Proximity neighbor selection

Reactive Recovery: The obvious technique

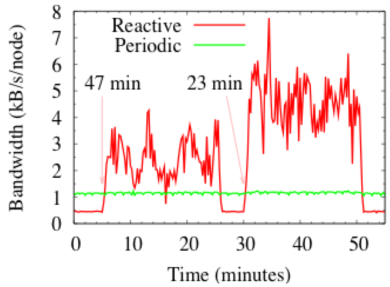
- For correctness, maintain leaf set during churn
  - Also routing table, but not needed for correctness
- The Basics
  - Ping new nodes before adding them
  - Periodically ping neighbors
  - Remove nodes that don’t respond
- Simple algorithm
  - After every change in leaf set, send to all neighbors
  - Called *reactive* recovery

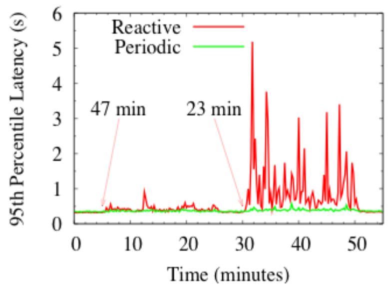
The Problem With Reactive Recovery

- Under churn, many pings and change messages
  - If bandwidth limited, interfere with each other
  - Lots of dropped pings looks like a failure
- Respond to failure by sending more messages
  - Probability of drop goes up
  - We have a positive feedback cycle (squelch)
- Can break cycle two ways
  1. Limit probability of “false suspicions of failure”
  2. Recovery periodically

Periodic Recovery

- Periodically send whole leaf set to a random member
  - Breaks feedback loop
  - Converges in O(log N)
- Back off period on message loss
  - Makes a negative feedback cycle (damping)





Conclusions/Recommendations

- Avoid positive feedback cycles in recovery
  - Beware of “false suspicions of failure”
  - Recover periodically rather than reactively
- Route around potential failures early
  - Don’t wait to conclude definite failure
  - TCP-style timeouts quickest for recursive routing
  - Virtual-coordinate-based timeouts not prohibitive
- PNS can be cheap and effective
  - Only need simple random sampling

Today’s Paper

- Dynamo: Amazon’s Highly Available Key-value Store, Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels. Appears in *Proceedings of the Symposium on Operating Systems Design and Implementation* (OSDI), 2007

- Thoughts?

The “Traditional” approaches to storage

- Relational Database systems
  - Clustered - Traditional Enterprise RDBMS provide the ability to cluster and replicate data over multiple servers – providing reliability
    - » Oracle, Microsoft SQL Server and even MySQL have traditionally powered enterprise and online data clouds
  - 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
- 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: Simplify

- Downgrade some of the service guarantees of traditional RDBMS
  - Replace the highly complex data models with a simpler one
    - » Classify services based on complexity of data model they require
  - Replace the “Always Consistent” guarantee synchronization model with an “Eventually Consistent” model
    - » Classify services based on how “updated” their data sets must be
- Redesign or distinguish between services that require a simpler data model and lower expectations on consistency

Many Systems in this space:

- **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

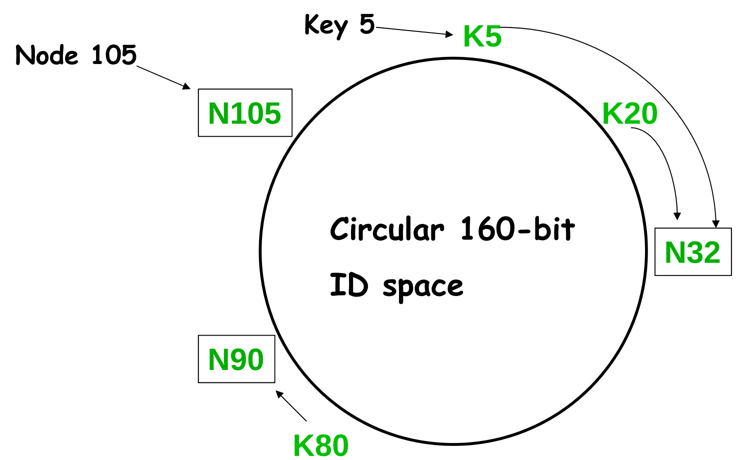
- **Facebook’s Cassandra –** Facebook’s principle data cloud, for their services.

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

Why Peer-to-Peer ideas for storage?

- Incremental Scalability
  - Add or remove nodes as necessary
    - » Systems stays online during changes
  - With many other systems:
    - » Must add large groups of nodes at once
    - » System downtime during change in active set of nodes
- Low Management Overhead (related to first property)
  - System automatically adapts as nodes die or are added
  - Data automatically migrated to avoid failure or take advantage of new nodes
- Self Load-Balance
  - Automatic partitioning of data among available nodes
  - Automatic rearrangement of information or query loads to avoid hot-spots
- Not bound by commercial notions of semantics
  - Can use weaker consistency when desired
  - Can provide flexibility to vary semantics on a per-application basis
  - Leads to higher efficiency or performance

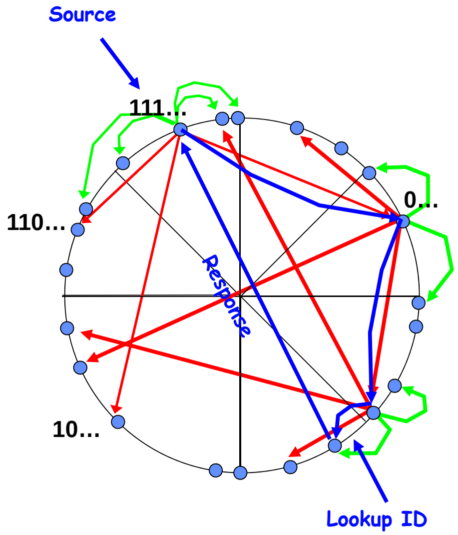
Recall: Consistent hashing [Karger 97]



A key is stored at its successor: node with next higher ID

Recall: Lookup with Leaf Set

- Assign IDs to nodes
  - Map hash values to node with closest ID
- Leaf set is successors and predecessors
  - All that's needed for correctness
- Routing table matches successively longer prefixes
  - Allows efficient lookups
- Data Replication:
  - On leaf set



Advantages/Disadvantages of Consistent Hashing

- Advantages:
  - Automatically adapts data partitioning as node membership changes
  - Node given random key value automatically "knows" how to participate in routing and data management
  - Random key assignment gives approximation to load balance
- Disadvantages
  - Uneven distribution of key storage natural consequence of random node names ⇒ Leads to uneven query load
  - Key management can be expensive when nodes transiently fail
    - » Assuming that we immediately respond to node failure, must transfer state to new node set
    - » Then when node returns, must transfer state back
    - » Can be a significant cost if transient failure common
- Disadvantages of "Scalable" routing algorithms
  - More than one hop to find data ⇒ O(log N) or worse
  - Number of hops unpredictable and almost always > 1
    - » Node failure, randomness, etc

Dynamo Goals

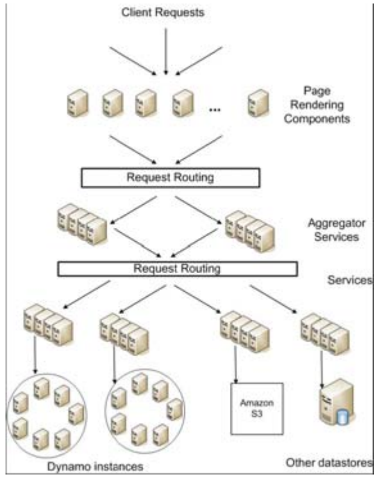
- Scale – adding systems to network causes minimal impact
- Symmetry – No special roles, all features in all nodes
- Decentralization – No Master node(s)
- Highly Available – Focus on end user experience
- SPEED – A system can only be as fast as the lowest level
- Service Level Agreements – System can be adapted to an application's specific needs, allows flexibility

Dynamo Assumptions

- Query Model – Simple interface exposed to application level
  - Get(), Put()
  - No Delete()
  - No transactions, no complex queries
- Atomicity, Consistency, Isolation, Durability
  - Operations either succeed or fail, no middle ground
  - System will be eventually consistent, no sacrifice of availability to assure consistency
  - Conflicts can occur while updates propagate through system
  - System can still function while entire sections of network are down
- Efficiency – Measure system by the 99.9th percentile
  - Important with millions of users, 0.1% can be in the 10,000s
- Non Hostile Environment
  - No need to authenticate query, no malicious queries
  - Behind web services, not in front of them

Service Level Agreements (SLA)

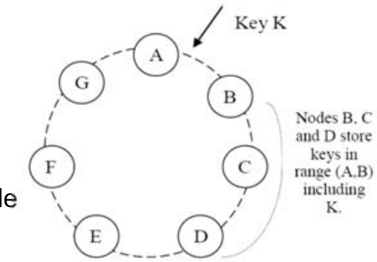
- Application can deliver its functionality in a bounded time:
  – Every dependency in the platform needs to deliver its functionality with even tighter bounds.
- Example: service guaranteeing that it will provide a response within 300ms for 99.9% of its requests for a peak client load of 500 requests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



Service-oriented architecture of Amazon's platform

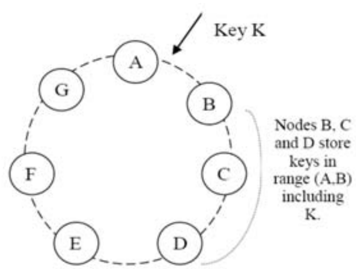
Partitioning and Routing Algorithm

- Consistent hashing:
  – the output range of a hash function is treated as a fixed circular space or "ring".
- Virtual Nodes:
  – Each physical node can be responsible for more than one virtual node
  – Used for load balancing
- Routing: "zero-hop"
  – Every node knows about every other node
  – Queries can be routed directly to the root node for given key
  – Also – every node has sufficient information to route query to all nodes that store information about that key



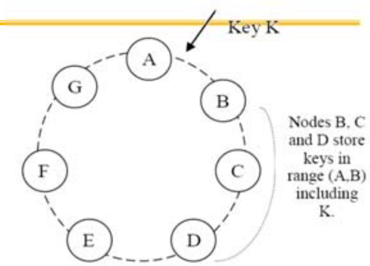
Advantages of using virtual nodes

- 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 responsible for storing a particular key
  – Successive nodes not guaranteed to be on different physical nodes
  – Thus preference list includes physically distinct nodes
- Replicas synchronized via anti-entropy protocol
  – Use of Merkle tree for each unique range
  – Nodes exchange root of trees for shared key range



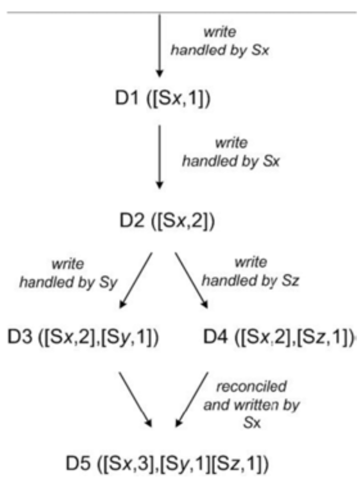
Data Versioning

- 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



Conflicts (multiversion data)

- Client must resolve conflicts
  - Only resolve conflicts on reads
  - Different resolution options:
    - » Use vector clocks to decide based on history
    - » Use timestamps to pick latest version
  - Examples given in paper:
    - » For shopping cart, simply merge different versions
    - » For customer's session information, use latest version
  - Stale versions returned on reads are updated ("read repair")
- Vary N, R, W to match requirements of applications
  - High performance reads: R=1, W=N
  - Fast writes with possible inconsistency: W=1
  - Common configuration: N=3, R=2, W=2
- When do branches occur?
  - Branches uncommon: 0.0006% of requests saw > 1 version over 24 hours
  - Divergence occurs because of high write rate (more coordinators), not necessarily because of failure

Execution of get () and put () operations

- Route its request through a generic load balancer that will select a node based on load information
  - Simple idea, keeps functionality within Dynamo
- Use a partition-aware client library that routes requests directly to the appropriate coordinator nodes
  - Requires client to participate in protocol
  - Much higher performance

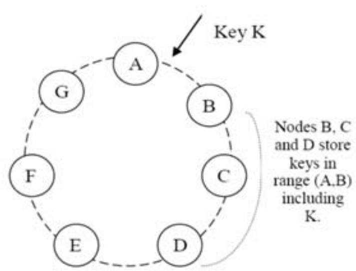
	99.9th percentile read latency (ms)	99.9th percentile write latency (ms)	Average read latency (ms)	Average write latency (ms)
Server-driven	68.9	68.5	3.9	4.02
Client-driven	30.4	30.4	1.55	1.9

Sloppy Quorum

- R/W is the minimum number of nodes that must participate in a successful read/write operation.
- Setting R + W > N yields a quorum-like system.
- 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 B is temporarily down or unreachable during a write, send replica to E
- E is hinted that the replica belongs to B and it will deliver to B when B is recovered.
- Again: “always writeable”



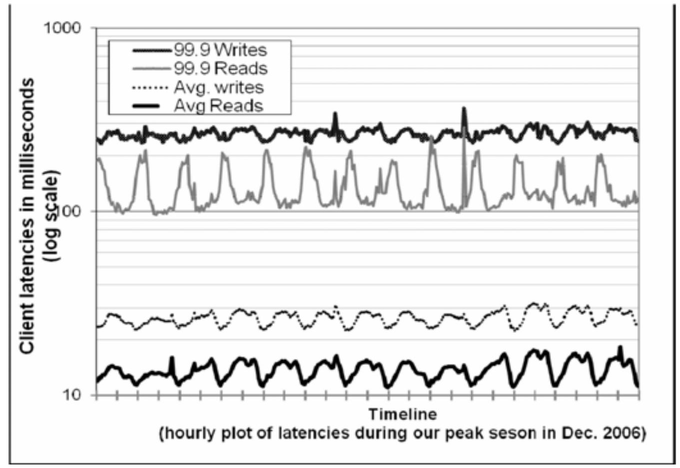
Implementation

- Java
  - Event-triggered framework similar to SEDA
- 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.

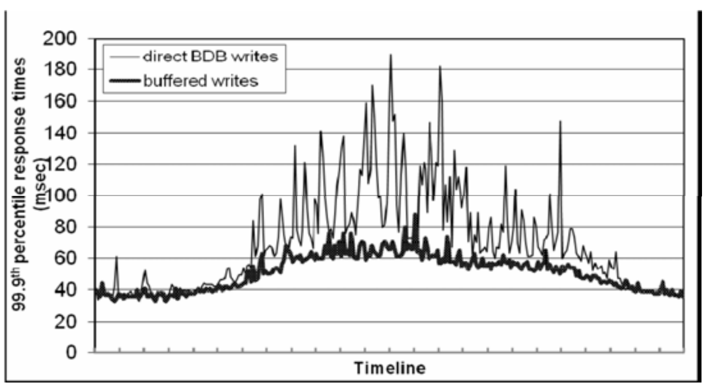
Summary of techniques used in *Dynamo* and their advantages

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.

Evaluation



Evaluation: Relaxed durability⇒performance



Is this a good paper?

- What were the authors' goals?
- What about the evaluation/metrics?
- Did they convince you that this was a good system/approach?
- Were there any red-flags?
- What mistakes did they make?
- Does the system/approach meet the "Test of Time" challenge?
- How would you review this paper today?