Lecture 19 Consistency, a lack thereof, CAP, NoSQL, and Dynamo 11/17/2014
Aaron Elmore Lecturing

Replication refresher: we have many choices with replication protocols including primary copy vs multi-master and async vs sync.

Replication is often used to address fault-tolerance and performance.
How does it address performance? Fast Reads and Locality. What about writes?

When designing a system for scale a common design principle to follow:
Keep it simple and things will fail. With reasonable mean time to failure (MttF) we can expect regular failures.

Netflix lessons: Rambo architecture and chaos monkey

Replication clearly helps with read availability, but how about write-availability?
To get there may have to make some trade-offs.

Consistency -- or how we reason about replicated state.
Many notions of consistency:

Eventual Consistency	Strong Consistency
(replicas will eventually	act if not replicated
converge if updates/reads stop)	1-copy serializability

Many models of consistency (admissibility criteria). Examples:
- read your writes
- monotonicity

No free lunch: decision between availability and consistency.

CAP Theorem

Eric Brewer at PODC 02 stated system can have 2 of 3 properties
-Consistency
-Availability
-Partition Tolerance

(CAP) proof on systems with async communication

With DBs that scale we will need to tolerate partitions (they happen - some say they don't)
Why is the decision between A & C?

3 nodes n1, n2, and n3 storing value of x as 4. Update x=5 arrives at n1 while n3 is partitioned. A read arrives at n3… What do we do?
Does it come down to ACID vs BASE (Basically Available Soft-State Eventual Consistency)?

We know partitioning data makes updates hard.

Enter NoSQL

Web scale companies had issue scaling databases (transactions). Homebrew new DBMS to address scale-out issues.

Common Attributes:
Partition data on key
Single-key atomicity
Drop expensive ops (txns, joins, secondary indexes)
Avoid single points of failure

Dynamo

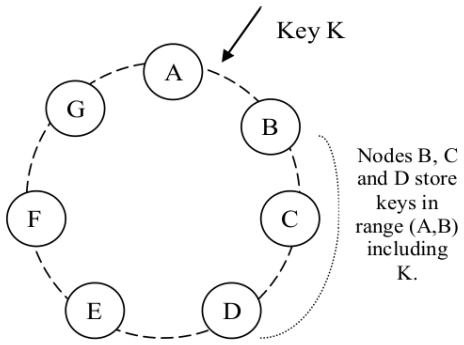
Amazon wanted always available DB (i.e. add to shopping cart should never fail)
A page render can use up to 150 services (diagram), so stringent SLA requirements.
Partitions or temporary failures happen.
Simple query language/data model:
- key:value assume both are byte array, md5 on key to generate ID
- get(key)
- put(key,context,value)
- single-key atomicity

Other key design principles:
Incremental scalability (add nodes)
Symmetry/ decentralizations (each node does same thing, no central control)
Heterogeneity in nodes (servers will change)

Challenges faced:
partitioning
highly available for writes
handle temporary failures
recovery from permanent failures
membership / failure detection

Partitioning

Consistent hashing goes to point in ring (wraps for larger values). One hop routing
Node(s) that move clockwise along ring are responsible



Uses a variation with *virtual nodes*, where multiple smaller vnodes are mapped to a single server. Has the following benefits:

- Node unavailability distributes load requirements
- New nodes or node re-availability accepts equivalent amount of from each of other available noes
- Allows for heterogeneity in nodes with different vnode allocations

Replication

N replicas

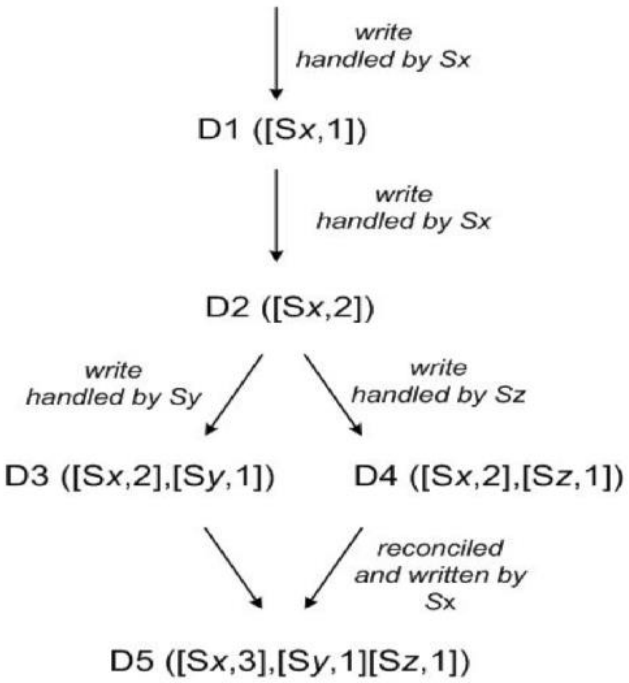
Discover N nodes through preference list

Reads and writes

Client contacts coordinator which uses N top healthy nodes
Reads and writes are driven by N, R, W
R+W > N for consistency *
(for load balancing can contact other node)

Data versioning

Allow divergent writes & cannot forget a write
How to detect conflicts? Dynamo uses vector clocks to capture causality between versions of object.
A vector clock is effectively a list of (node,counter) pairs. This is why write needs context



Need a way to reconcile divergent versions, can be application driven or last write wins

Hinted handoff

Sloppy Quorum (healthy N)
Node can get request intended for another node, hints include the expected list
In ring example D gets request intended for A if A is unavailable, D will check A later to send update.

Replica synchronization

Hinted handoff works with low churn and transient failure
We need to o synchronize replicas and minimize data transfer: merkle trees are used
Merkle Tree is a hash tree where leaves are hashes of value and parents are hashes of children

Each node maintains separate merkle tree for each key range it hosts

Membership

Gossip based protocol propagates membership changes (each node contacts random peer and reconcile membership view)

When node is added is takes a set of tokens (virtual nodes) and mapping is persisted and reconciled with other nodes. This creates the mapping of keys to nodes, which allows one hop lookup for reads/writes.

Seed nodes are used to bootstrap nodes and prevent logical partitions. The seed nodes are well known nodes and can be used to connect or add membership

Failure detection is detected through failed get / put requests.

Other details

Pluggable storage engine (mysql, bdb, in-mem)
Read repairs (after read quorum R is answered, wait for responses to reconcile any divergent views)

Problem	Technique	Advantage
partitioning	consistent hashing	incremental scalability
highly available for writes	vector clocks with read repair	version size decoupled from update rate
handle temporary failures	sloppy quorum and hinted handoff	HA with some durability
recovery from permanent failures	anti-entropy with merkle trees	sync replicas
membership / failure detection	gossip based membership	symmetry and no centralized repo

Thoughts on this design?

BIG TABLE Next time

Internal database for google
Built on distributed file system: GFS
Centralized control