Eventually Consistent

W. Vogels

Communications of the ACM
Jan. 2009

Introduction

- Under computing conditions involving massive scale, requiring good performance and availability, use:
  - Consequences to using replication
    - Data consistency is affected
  - Trade off between high availability and consistency

Introduction

- In perfect world
  - Updates seen by all (consider real-time…)
- First became issue in late 1970’s with DBs
  - Distributed DBs – techniques proposed to achieve distribution transparency
    - Better to fail than to break transparency (oh no!!)
- In 1990s with Internet, availability more important

CAP

- CAP theorem – only 2 of the following at same time
  - Data **consistency**, system **availability**, tolerance to network **partitions**
    - System not tolerant to data partition can achieve consistency and availability (transactions)
      - Requires client and storage systems to be part of same environment
  - Partitions part of distributed systems so relax consistency for availability (or vice versa)
- Client developer must be aware of trade-off

Client-side consistency

- Range of applications can tolerate stale data
- This is not the ACID kind of consistency
- Client-side consistency:
  - How/when processes see updates to stored objects
  - Storage system – large, distributed, guarantee durability and availability

Client-side consistency

Assume process *A* R/W to data, processes *B*, *C* independent of *A*, R/W to data

- Strong consistency – subsequent accesses by *A*, *B*, or *C* will return updated value
- Weak consistency – no guarantee subsequent accesses return updated value
  - Inconsistency window – period between update and when guarateed will see update

Client-side – eventual consistency

- Eventual consistency – form of weak
  - If no new updates, eventually all accesses return last updated value
    - Size of inconsistency window determined by communication delays, system load, number of replicas
    - Implemented by domain name system (DNS)

Client-side – eventual consistency

- Variations on Eventual consistency
  - Causal consistency – If *A* tells *B* updated, access by *B* will see updated value, W guaranteed to supersede earlier W. Access by *C* subject of normal rules
  - R-your-W consistency – If *A* updates, *A* always sees updated value
  - Session consistency – Process accesses storage in sessions, R-your-W consistency guaranteed during session
  - Monotonic R consistency – once process sees a value, never sees previous value
  - Monotonic W consistency – serializes W by same process

Client-side – eventual consistency

- Can combine properties
  - Monotonic R and R-your-W, most desirable in eventually consistent system. Provide high availability
  - Many modern RDMSs providing primary-backup reliability implement replication in both synchronous and asynchronous modes
    - Synchronous part of transactions
    - Asynchronous – updates arrive delayed, through log shipping
    - For scalable performance, RDBMSs read from back-up – eventual consistency with inconsistency window period of log shipping

Server-side consistency

N - number of nodes storing replicas of data

W - number of replicas needed to acknowledge receipt of update before update completes

R - number of replicas contacted when data is read

- If W+R>N, then W and R set overlap so can guarantee strong consistency
  - In primary-backup RDBMS with synchronous replication, N=2, W=2 and R=1
    - No matter which replica, always consistent
  - For asynchronous replication, N=2, W=1, R=1
    - No guarantees

Server-side consistency

- Basic quorum protocols fail when cannot write to W nodes – unavailable
  - E.g. with N=3, W=3, if only 2 nodes available
- For high performance distributed systems, number of replicas > 2, e.g. N=3, W=2, R=2
- If high read loads, N = 10s or 100s nodes, with R=1
- If system concerned with consistency W=N
- Systems concerned with fault tolerance but not consistency W=1 (minimal durability on update, lazy update to other replicas)

Server-side consistency

- Configuration of N, W, R depends on which performance aspect needs to be optimized
  - R=1, N=W optimized for read case
  - W=1, R=N, optimized for fast write, but no durability if failure
  - If $W<(N+1)/2$, conflicting W when W sets do not overlap

Server-side consistency

- Weak/eventual consistency when W+R<=N as R,W will not overlap. Might as well set R=1 for
  - Read scaling
  - Data access more complicated
- In key-value model, easy to determine latest version, not true if return a set of objects
  - In such systems, lazy updates by inconsistency window
  - But can read from nodes not yet updated if (W+R<=N)
- Achieving R-your-W, session, monotonic consistency depends on "stickiness" of client to server
  - E.g. same server executes protocol each time
    - But more difficult for load balancing and fault tolerance
  - Sometimes client implements R-your-W and monotonic reads by discarding reads if previous versions

Server-side consistency

- If partitions occur within or between data centers
  - Can use classic majority quorum approach
    - Partition that has W nodes can make updates while other partitions unavailable
    - Same for R set
    - If these two sets overlap – minority set unavailable
  - Or, both sides of partition assign new set of storage nodes, merge operation when partition heals
    - Amazon uses such W-always systems in shopping cart

Example

- Amazon's dynamo
  - Key value storage system
  - Used in e-commerce, Web services
  - Allows application service owner who creates instance of Dynamo storage systems trades-off among
    - consistency, durability, availability and performance

Conclusion

- Data inconsistency in large scale reliable distributed systems must be tolerated for:
  - Improving R,W under highly concurrent conditions
  - Handling partition cases
- Inconsistency acceptability depends on client application
  - Must be aware of consistency guarantees provided by storage system
  - Example is web site with notion of user-perceived consistency
    - Inconsistent window smaller than time expected for customer to return for next page load
- Need to operate at global scale