INFO5011 – Advanced Topics in IT: Cloud Computing

Week 5: Distributed Data Management: From 2PC to Dynamo

Dr. Uwe Röhm
School of Information Technologies
Outline

- Distributed Data Processing
  - Data Partitioning
  - Data Replication
  - Distributed Transactions
  - 2-Phase Commit Protocol

- The CAP Principle

- NoSQL Data Stores
  - Example: Amazon’s Dynamo

Goals for a Shared Data System?

- **Strong Consistency**
  - We would like to have ‘all-or-nothing’ semantics and ideally have all copies of the same data always consistent as if there is only 1 copy.

- **High Availability**
  - A data system should always be up.
    - E.g. Werner Vogels keynote: Amazon will always take your order.
    - The challenge: the larger the system, the higher the prob of failures.

- **Partition Tolerance**
  - If there is a network failure that splits the processing nodes into two groups that cannot talk to each other, then the goal would be to allow processing to continue in both subgroups.
How to achieve this?

Are distributed transactions with ACID guarantees the solution?

Firstly: Appropriate data placement
  - Replication
  - Data Partitioning

Secondly: Concept of a distributed transaction: a transaction that invokes local transactions on different sites
  - Eg. Bank transfer: invoke withdraw at one site and deposit at another
Distributed Data

- **Partitioning**
  - Horizontal vs Vertical
  - Round robin vs. range partitioning vs. hash partitioning

- **Replication**
  - when the value of some data item is stored in more than one place
    - Typically in different databases at different physical locations
    - Similar issues arise with caches
  - Gives increased availability
  - Faster query evaluation
How to Keep Replicas Consistent?

- “Always” consistent
  - At least, apps shouldn’t observe difference from using single dbms
    - “transparent replication”
  - Formal definition for “1 copy serializable (abbreviated as 1-SR)”
  - Some systems propose “1-copy SI”

- Eventually consistent
  - “convergent”
  - If updates cease for long enough, all copies will reach a common value

- Intermediate approach: limited divergence
Distributed Transactions

- Incorporate transactions at multiple servers into a single (distributed) transaction
  - Not all distributed applications are legacy systems; some are built from scratch as distributed systems

```
tx_begin;
order_part;
withdraw_part;
payment;
tx_commit;
```

```
Inventory Application

order_part;
withdraw_part;
payment;
```

```
Billing Application

Site A
```

```
DBMS 1

Site B
```

```
DBMS 2

Site C
```
Distributed Database Systems

- Each local DBMS might export
  - stored procedures, or
  - an SQL interface.

- In either case, operations at each site are grouped together as a subtransaction and the site is referred to as a participant of the distributed transaction
  - Each subtransaction is treated as a transaction at its site

- Coordinator module (part of TP monitor) supports ACID properties of distributed transaction
  - Transaction manager acts as coordinator
Correctness of Distributed Transactions

**Goal:** distributed transaction should be ACID
- Each *local* DBMS
  - supports ACID properties locally for each subtransaction
    - Just like any other transaction that executes there
  - eliminates local deadlocks

- In addition the transaction should be *globally* ACID
  - **A:** Either *all* subtransactions commit or all abort
    - two-phase commit protocol
  - **C:** *Global* integrity constraints are maintained
  - **I:** Concurrently executing distributed transactions are *globally* serializable
  - Even if local sites are serializable, subtransactions of two distributed transactions might be serialized in different orders at different sites:
    - At site A, $T_{1A}$ is serialized before $T_{2A}$
    - At site B, $T_{2B}$ is serialized before $T_{1B}$
  - **D:** Each subtransaction is durable
Example: Placing an e-Biz Order

tx_begin;
    reserve_stock(item); ... 
    place_new_order(item); ... 
    tx_commit;

- **Atomicity:** Either both subtransactions commit or neither does
- **Consistency:** \( \text{Sum of all reserved items} = \text{total ordered items} \)
- **Global isolation** - local serializability at each site does not guarantee global serializability
  - *Shipping* subtransaction is serialized after *audit* subtransaction in DBMS at warehouse 1 and before *audit* in DBMS at warehouse 2 (local isolation), *but*
  - there is no global order

\[
\begin{array}{ccc}
\text{post\_interest} & \text{audit} \\
\text{time} & \sum \text{items at warehouse 1;}
\end{array}
\]

\[
\downarrow \\
\begin{array}{c}
\text{ship item1 from warehouse1;} \\
\text{ship item2 from warehouse2;} \\
\sum \text{items at warehouse 2;}
\end{array}
\]
Global Atomicity

- All subtransactions of a distributed transaction release locks only at commit; and they must commit or all must abort.

- An *atomic commit protocol*, initiated by a *coordinator* (e.g., the transaction manager), ensures this.
  - Coordinator polls *cohorts* to determine if they are all willing to commit.

Why is this hard?

- Imagine sending to each DBMS to say “commit this txn T now”

- Even though this message is on its way, any DBMS might abort T spontaneously.
  - e.g. due to a system crash.
Basic Idea

- Two round-trips of messages
  - Request to prepare/ prepared or aborted
  - Either Commit/committed or Abort/aborted

Only if all participating sites are already prepared!
2-Phase Commit Protocol

- Implemented as an exchange of messages between the coordinator and the subordinates
  - The subordinates are the individual subtransactions that participated in the transaction
  - The coordinator polls the subordinates to see if they want to commit
- When a distributed transaction wants to commit:
  1. Coordinator sends *prepare message* to each subordinate
  2. Each subordinate ‘prepares’ (flushes abort or prepare log entry) and then sends a *vote message* (*yes* or *no*) to the coordinator
  3. Coordinator decides on global outcome, logs it, and then sends a *commit/abort message* to each subordinate
  4. Subordinate write a commit or abort log entry and then send an *acknowledge message* back to the coordinator
The Two-Phase Commit Protocol

Global Serializability

- **Theorem**: If all sites use a two-phase locking protocol and a two-phase commit protocol is used, transactions are globally serializable
  - Transactions are serialized in the same order at every site – the order in which the transactions committed

- **Problems**: Global deadlock can be another result of implementing two-phase locking and two-phase commit protocols
  - At site A, $T_{1A}$ is waiting for a lock held by $T_{2A}$
  - At site B, $T_{2B}$ is waiting for a lock held by $T_{1B}$

- Systems use deadlock detection algorithms or timeout to deal with this

- 2PC protocol is blocking -> does not scale
Is this enough for large-scale data sharing?

- Problem of 2-Phase Commit is its blocking nature
  - Timeout helps some way, but it still means waiting
  - In general, a participant cannot complete the protocol until some failure is repaired, it is said to be blocked
  - Blocking can impact performance at the cohort site since locks cannot be released

- This is not good enough for large-scale web applications
The CAP Theorem

Theorem:
You can have at most two of these properties for any shared-data system.

- Consistency
- Availability
- Partitioning Tolerance

[Brewer, PODC2000]
Forfeit Partitioning Tolerance

Examples:
- Single-site DBMS
- Cluster of databases

Techniques:
- 2-Phase Commit
- Cache Validation Protocol
- 1cp Replication Protocols

[Forfeit Partitioning Tolerance] (Brewer, PODC2000)
Forfeit Availability

- **Examples:**
  - Distributed databases
  - Majority Protocols

- **Techniques:**
  - Pessimistic Locking
  - Make minority partitions available

[INFO5011 "Cloud Computing" - 2011 (U. Röhm and Y. Zhou)]
Forfeit Consistency

- **Examples:**
  - Web Caching
  - Amazon’s Dynamo
  - Google …

- **Techniques:**
  - Eventual consistency protocols
  - Access to stale data
  - Conflict resolution in Apps

[Source: Brewer, PODC2000]
Open Research Area

- The jury is out: CAP or ACID?

- E.g. Blog by Stonebraker:

- Sparked new interest in replication techniques
  - Eventually consistency protocols
  - Better scalable SI-based replication protocols

  - Still problems with WAN setting though
NoSQL Data Storage
NoSQL or NoRel?

- Original idea: a lightweight DBMS that does not expose a full-fledged SQL interface

- Originally, MySQL was started with this in goal
  - But in the meanwhile, clearly moves towards a full-fledged DBMS including complex SQL, triggers, stored procedures etc.
  - Single outstanding feature though: supports different storage engines that can be used per table

- Today:
  Better name would be ‘Non-Relational’ as the most common interpretation of "NoSQL" is nowadays "non-relational"
  - or as some say: “Not-Only-SQL”
Criticisms about ‘SQL’ databases

- Heavy-weight and hard to understand
  - Yes, Oracle comes in CDROMs
  - But then, sqlite is ~300KB and available in most Unix’s by default

- “Slow”
  - What is the definition of ‘slow’? (better: think scalability)
  - Yes, an SQL interface introduces some overhead,
    - but can help if used wisely – e.g. by pushing complex queries to DBMS
  - But most overhead is less about SQL, but how it is used + Transacts

- Expensive
  - Need qualified DBA personal; license costs of commercial DBMS…

- Schema must be known first
  - This is a valid point…

- We don’t need transactions or strong consistency
  - That depends – in general, it’s simply the price people are Ok to pay atm.
The Different Flavors of NoSQL

- **In the blue corner:** Relational DBMS
  - Set of fixed-structured tuples; 1NF; joins; SQL; transactions

- **In the red corner:** ‘NoSQL’
  - ‘document-oriented’ databases
    - Nested, hierarchical lists of key-value pairs
    - e.g. MongoDB or XML databases
  - Key-map-of-values stores
    - e.g. Amazon’s SimpleDB
  - Pure Key-Value stores
    - e.g. Amazon’s Dynamo
  - Graph databases
    - E.g. Neo4J
  - ...

- **Note:** So far, no standard data model or API for ‘NoSQL’
Case Study: Amazon’s Dynamo

Highly scalable (key-value) store

Core Assumptions

- Simple read/write operations to data with unique IDs
- No operation spans multiple entries
- Data stored of small size

Trade-off:

- **Scalability** and **Availability** is everything (assume that errors happen)
  - Amazon really cares for ‘write-is-always-possible’
- Guaranteed SLAs
  - Goes back to Scalability: guaranteed response time for 99.9% requests
- Consistency, Declarative Querying and general Transactions not as important in some scenarios
Case-Study: SLA Idea at Amazon

- Application can deliver its functionality in abounded 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.
## Summary of Techniques in Dynamo

<table>
<thead>
<tr>
<th>Problem</th>
<th>Technique</th>
<th>Expected Advantage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partitioning</td>
<td>Consistent Hashing</td>
<td>Incremental Scalability</td>
</tr>
<tr>
<td>High Availability for writes</td>
<td>Vector clocks with reconciliation during reads</td>
<td>Version size is decoupled from update rates.</td>
</tr>
<tr>
<td>Handling temporary failures</td>
<td>Sloppy Quorum and hinted handoff</td>
<td>Provides high availability and durability guarantee when some of the replicas are not available.</td>
</tr>
<tr>
<td>Recovering from permanent failures</td>
<td>Anti-entropy using Merkle trees</td>
<td>Synchronizes divergent replicas in the background.</td>
</tr>
<tr>
<td>Membership and failure detection</td>
<td>Gossip-based membership protocol and failure detection.</td>
<td>Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.</td>
</tr>
</tbody>
</table>

[SOSP2007, Table1]
Dynamo = DHT

- Basically:
  A huge distributed hash table (DHT) with replication

- Just two operations:
  - `Put(key, context, object)`
  - `object Get(key)`

  *context*: vector clocks and history (needed for merging)

DHT: cf. also Chord system
Partition Algorithm

- **Consistent hashing:**
  - the output range of a hash function is treated as a fixed circular space or “ring”.
  - Each node gets an ID from the space of keys
  - Data stored on the first node clockwise of the current placement of the data key

- **Replication:** Each data item is replicated at $N$ nodes following the responsible node (so-called ‘preference list’)

[SOSP2007, Figure2]
Load Balancing => Virtual Nodes

- A problem with the basic DHT scheme (Chord-like)
  - Nodes placed randomly on ring
  - Leads to uneven data & load distribution

- In Dynamo
  - Use “virtual nodes”
  - Each physical node (or host) can be responsible for multiple virtual nodes
    - More powerful machines have more virtual nodes
  - Distribute virtual nodes across the ring

- Load Balancing Effect:
  - If a node becomes unavailable the load handled by this node is evenly dispersed across the remaining available nodes.
CAP for Dynamo: Choose 2 of 3

- **Priority 1: High Availability**
  - Rationale: Availability of online services == customer trust
  - ‘We always can take your order’

- **But: In data centers failures happen all the time**
  - Hence Dynamo must tolerate partitions

- **According to the CAP Principle, if we choose A and P, Consistency (C) has to be sacrificed**
  - Rationale: Many services do tolerate small inconsistencies
  - loose consistency ==> Eventual Consistency
Data Versioning

Get() and Put() execution:

1. Route each request through a generic load balancer that will select a node based on load information.
2. Use a partition-aware client library that routes requests directly to the appropriate coordinator nodes.

Effect:

- 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.*
Data Versioning

- Updates generate a new timestamp
- Eventual consistency
  - Multiple versions of the same object might co-exist
- Syntactic Reconciliation
  - System might be able to resolve conflicts automatically
- Semantic Reconciliation
  - Conflict resolution pushed to application

[SOSP2007, Figure3]
Quorum Protocol for Get&Put

- Challenge: Is a Put() /Get() on just one node enough? No!

- Solution: Quorum-like Protocol
  - Weighted Voting System by Gifford ('79)
  - \( R \) and \( W \) is the minimum number of nodes that must participate in a successful read/write operation.

- Put() /Get() is routed to a coordinator node among the top \( N \) in the preference list

- Coordinator then runs a \( R \ W \) quorum protocol
  - \( R = \) read quorum
  - \( W = \) write quorum
  - \( R + W > N \)
# NoSQL Classification Attempt

<table>
<thead>
<tr>
<th></th>
<th>SQL</th>
<th>Document Stores</th>
<th>Key-Value Stores</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data Model</strong></td>
<td>Set of tuples</td>
<td>Set of (key, value) pairs</td>
<td>(key, value) pairs</td>
</tr>
<tr>
<td><strong>Declarative Querying</strong></td>
<td>SQL</td>
<td>Selections on keys</td>
<td>Select key</td>
</tr>
<tr>
<td><strong>Updates</strong></td>
<td>in-place update of single or tuple set</td>
<td>in-place update of single value or doc</td>
<td>update(key: value)</td>
</tr>
<tr>
<td><strong>Transactions</strong></td>
<td>ACID</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Assumes single (k,v) access</td>
</tr>
<tr>
<td><strong>Physical Design</strong></td>
<td>Indexes, Partition, Mat. Views, ...</td>
<td>Indexes</td>
<td>Index on key</td>
</tr>
<tr>
<td><strong>Distribution</strong></td>
<td>horiz. or vertical partitioning</td>
<td>Horizontal partitioning</td>
<td>Partitioning by key</td>
</tr>
<tr>
<td><strong>Replication</strong></td>
<td>All kinds…</td>
<td>eventual consistency</td>
<td>eventual consistency</td>
</tr>
<tr>
<td><strong>Examples</strong></td>
<td>Postgres, MySQL, Oracle, DB2, SQL Server, Sybase</td>
<td>MongoDB, CouchDB ?</td>
<td>Amazon Dynamo, memcachedb ?</td>
</tr>
</tbody>
</table>
Summary

- **Distributed Data Management**
  - Two Main Physical Design Alternatives
    - Data Replication
    - Data Partitioning
  - **Strong Consistency: 2-Phase-Commit Protocol**
    - Bad scalability for very large system due to blocking nature

- **CAP Theorem**
  - In large distributed systems, at most 2/3 DAP properties achievable

- **NoSQL Storage Systems**
  - Non-SQL data model + CAP principles
  - Example: Amazon’s Dynamo
References

- **2-Phase Commit**
  - Kifer/Bernstein/Lewis (2nd edition)

- **NoSQL:**
  - *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. SOSP 2007.