Distributed Data Management
Summer Semester 2013
TU Kaiserslautern

Dr.-Ing. Sebastian Michel

smichel@mmci.uni-saarland.de
NOSQL: CONCURRENCY CONTROL, EVENTUAL CONSISTENCY, VECTOR CLOCKS, AND CONSISTENT HASHING
Recap: Consistency Tradeoff

• Recall from last lecture: CAP Theorem
• Says: C+A+P not possible together

• Required P=Partition Tolerance
Recap: Concurrency Control

• Addresses the problem of dealing with concurrent reads/writes to same data
• Transactions encapsulate batch of commands
• http://www.youtube.com/watch?v=G3xH2SoMOF0

• Traditional RDBMS: ACID
  – Isolation: transaction (TA) works with the DB as if it is the only TA
  – Atomicity: transaction is executed entirely or no changes are made at all
  – ...

Distributed Data Management, SoSe 2013, S. Michel
Read/Write and Write/Write Conflicts

• Database needs to handle conflicting operations on the same data record

• Two kinds of conflicts:
  – read/write
    • one TA wants to read record
    • second TA wants to write
  – write/write
    • two TAs want to write record
Conflict Prevention vs. Detection

- **Conflict Prevention:**
  - (Distributed) locking algorithms.
  - **Pessimistic** approach (assume things will go wrong, prevent that from happening)

- **Conflict Detection:**
  - Using “timestamps”
  - (+Keeping multiple versions)
  - **Optimistic** approach (resolve when actually happened)
Recap: Locking: Two Phase Locking

• Recall: Traditional way to ensure multi user concurrency control: Locking
  – *(Strict)* 2PL (Two Phase Locking)
  – Transaction claims required locks
  – Starts releasing locks at some point or all at once *(strict)*

• Distributed: Distributed 2PL, 2 Phase Commit
• Or: PAXOS commit protocol (for higher reliability)
Synchronization based on Timestamp Ordering (T/O)

- Transaction (TA) gets **timestamp** assigned
- Timestamps impose ordering of transactions.

- Conflicts are resolved by resetting TAs (which gets a new timestamp)

- Record **for each record** (item) the **largest timestamp** of any read or write operation.
Conflicts in T/O

• **R/W**: If TA with timestamp TS wants to read x 
  – reject if TS < write-timestamp of record x 
  – otherwise: read and set read-timestamp of record x to \( \max(\text{TS}, \text{read-timestamp of x}) \)

• **R/W**: If TA with timestamp TS wants to write x 
  – reject if TS < read-timestamp of x 
  – otherwise: write and set write-timestamp of x to \( \max(\text{TS}, \text{write-timestamp of x}) \)

• **W/W**: If a TA with timestamp TS wants to write x 
  – reject if TS < write-timestamp of record x 
  – otherwise: write and set write-timestamp to TS
Multiversion Concurrency Control

- No overwriting of old values: 
  
  Each write operation creates a new version

- Transactions have timestamp of begin

- Transaction that wants to read can immediately do that (no locking).
Multiversion Concurrency Control

• Read vs. Write Operations
• Read is never rejected; reads record version that is largest but < TS
Multiversion Concurrency Control (2)

- Conflict detection

$v_{\text{latest}} = v_0$

Alice

Bob

$t_0$

$t_1$

$v_0$

read $v_0$

write $v_0 \rightarrow v_{1a}$

$v_1$

read $v_0$

$v_1$

write $v_0 \rightarrow v_{1b}$

$t_2$

$t_3$

$v_{\text{latest}} = v_{1a}$

Conflict!

$v_{\text{latest}} \neq v_0$
Distributed Setup

• N copies per record/object, spread across servers
# Read Consistency Guarantees

<table>
<thead>
<tr>
<th>Guarantee</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strong Consistency</td>
<td>See all previous writes.</td>
</tr>
<tr>
<td>Eventual Consistency</td>
<td>See (any) subset of previous writes.</td>
</tr>
<tr>
<td>Consistent Prefix</td>
<td>See initial sequence of writes.</td>
</tr>
<tr>
<td>Bounded Staleness</td>
<td>See all “old” writes. E.g., everything older than 10 minutes.</td>
</tr>
<tr>
<td>Monotonic Reads</td>
<td>See increasing subset of writes.</td>
</tr>
<tr>
<td>Read My Writes</td>
<td>See all writes performed by reader.</td>
</tr>
</tbody>
</table>
High Availability Aim

• Availability/Response Time (Latency)

• 100ms additional latency in Amazon results in 1% of sales*

• 500ms additional latency in Google causes 20% decrease in traffic*

• In such settings: P and A are required. Hence, tradeoff C.

Consistency Tradeoff

• Client sends write request
• Assume there are $N$ copies (replicas) of data record/tuple/item
• Server does not wait until all (of the $N$) nodes (that keep the replicas) ack’ed the write but returns already after $W$ acks. So, we (as a client) know, there are for sure $W$ successful writes.
• A client’s read request is returned after reading from $R$ our of the $N$ nodes.

• Can lead to multiple conflicting versions of a data item, depending on configuration.

• Needs to detected and resolved
(Logical) Anatomy of a Write

Client

write request with $W=2$

Coordinator

write requests to all

response

OK, if received 2 ACKs

node1

node2

node3

Distributed Data Management, SoSe 2013, S. Michel
W < N

write

D1

node1

D1

node2

D0

node3

D1

node4

time to sync

4me to sync
Eventual Consistency

- After a write, data can be at some nodes/machines inconsistent
- But will eventually(!) become consistent again
- By (background) sync protocol

Client gets ACK after write
All nodes have same values for data

Time range of possibly inconsistent data

http://www.allthingsdistributed.com/2008/12/eventually_consistent.html
Consistency by Write to All

• Write to all: $W=N$ (i.e., wait until all writes are acknowledged)

• Read from one: $R=1$

• Guarantee to see at least one recent version
Consistency by Read From All

- Write to one: \( W=1 \)
- Read from all: \( R=N \)
- Guarantee to see the last version
Quorums and Configurations

• In general: Methodology gives space of solutions.
  – Number of servers/replicas: \( N \)
  – Simultaneous writes: \( W \)
  – Simultaneous reads: \( R \)

• Problematic if sets of written-to and read-from servers do not overlap:
  – Partial quorum if \( W+R \leq N \)

• What if \( W < (N+1)/2 \)? Write sets do not overlap.
## Configurations

<table>
<thead>
<tr>
<th>R/W Configuration</th>
<th>Kind of Consistency</th>
</tr>
</thead>
<tbody>
<tr>
<td>$W=N$ and $R=1$</td>
<td>Read optimized strong consistency.</td>
</tr>
<tr>
<td>$W=1$ and $R=N$</td>
<td>Write optimized strong consistency.</td>
</tr>
<tr>
<td>$W+R \leq N$</td>
<td>Eventual consistency. Read might miss recent writes.</td>
</tr>
<tr>
<td>$W+R &gt; N$</td>
<td>Strong consistency. Read will see at least one most recent write.</td>
</tr>
</tbody>
</table>
How Consistent is Eventual Consistency?

• Probability that we miss reading the last write.

• Put in some numbers

\[
p_s = \frac{\binom{N-W}{R}}{\binom{N}{R}}
\]

• Recent work* aims to quantify eventual consistency

Example Amazon’s Dynamo

- Key/Value store
- Simple operations: CRUD on a per key basis
- Aiming at **high availability**: “Never reject a write”
  - Leading to different versions of data (due to node failures but also concurrent writes)
- Conflicting versions are **reconciled** at read/write (using **vector clocks**)
- Data placement: **consistent hashing** (with virtual nodes)
Logical Ordering of Events in Distributed System

• So far, assumed way to detect conflicting reads/writes in distributed system based on version identifiers (clock)

• Want: logical (partial) ordering of events

• A kind of “timestamp” attached to records that indicate conflict or “descendant” relation
Lamport Timestamps

- Each node keeps integer count.
- Incremented at each atomic event.

Lamport Timestamps (2)

• At communication: receiver adapts clock to received value + 1 (if not larger than value already)
Lamport Timestamps: Properties

• If event x has happened before y
  – either x and y on same process
  – or x is sending and y receiving event on diff. process
    (written x->y)
  then C(x) < C(y), i.e.,
    x->y => C(y) < C(y)

• But not:  (Vector clocks can say this!)
  C(x) < C(y) => x -> y

• Why?
• Can we detect concurrent events?
Vector Clocks

- Idea: each node keeps separate counter
- By C. Fidge and F. Mattern in 1988 (independently)
- Vector clock: Vector of counters
  \[ [c_0, c_1, ..., c_n] \]
  \( c_i \) is counter for node \( i \)

**Initialization:** all \( c_i \) are zero: \([c_0, c_1, ..., c_n]\)

**Upon event** at local event at node \( i \): node increments \( c_i \) in its vector.

**Sending** clock: node \( i \) increments \( c_i \) and sends vector
Vector Clocks: Merging upon Receive

• When node $i$ receives clock of other node
  – node $i$ merges its vector clock $VC$ with the received one $VC_{\text{other}}$
  – as follows:

  increment own counter $c_i$, i.e., $VC[i]=VC[i]+1$

  for each $j$ do
    $VC[j] = \max(VC[j], VC_{\text{other}}[j])$
  end
Example Story

• “Alice, Ben, Cathy, and Dave are planning to meet next week for dinner. The planning starts with Alice suggesting they meet on Wednesday. Later, Dave discuss alternatives with Cathy, and they decide on Thursday instead. Dave also exchanges email with Ben, and they decide on Tuesday. When Alice pings everyone again to find out whether they still agree with her Wednesday suggestion, she gets mixed messages: Cathy claims to have settled on Thursday with Dave, and Ben claims to have settled on Tuesday with Dave. Dave can’t be reached, and so no one is able to determine the order in which these communications happened, and so none of Alice, Ben, and Cathy know whether Tuesday or Thursday is the correct choice.”

date = Wednesday
vclock = Alice:1

date = Tuesday
vclock = Alice:1, Ben:1

date = Tuesday
clock = Alice:1, Ben:1, Dave:1

date = Thursday
vclock = Alice:1, Cathy:1
Comparing Vector Clocks

• Dave has the following clocks:

  
  
  date = Tuesday  
  vclock = Alice:1, Ben:1, Dave:1

  
  date = Thursday  
  vclock = Alice:1, Cathy:1

• **Conflict** because neither clocks descends from the other.
Comparing Two Vector Clocks

- $VC_1 = VC_2$, 
  \[ iff \quad VC_1[i] = VC_2[i], \text{ for all } i = 1, \ldots, n \]
- $VC_1 \leq VC_2$, 
  \[ iff \quad VC_1[i] \leq VC_2[i], \text{ for all } i = 1, \ldots, n \]
- $VC_1 < VC_2$, 
  \[ iff \quad VC_1 \leq VC_2 \land \exists j \ (1 \leq j \leq n \land VC_1[j] < VC_2[j]) \]
- $VC_1$ is concurrent with $VC_2$ 
  \[ iff \quad (not \ VC_1 < VC_2 \land not \ VC_2 < VC_1) \]
Dave Resolves Conflict

• By choosing Thursday

\[ \text{date} = \text{Thursday} \]
\[ \text{vclock} = \text{Alice:1, Ben:1, Cathy:1, Dave:2} \]

• New clock tells that it is successor of the two previous clocks!
date = Thursday
clock = Alice:1, Cathy:1

Again Conflict?

- Alice gets from Ben

\[
\begin{align*}
\text{date} & = \text{Tuesday} \\
\text{vclock} & = \text{Alice:1, Ben:1, Dave:1}
\end{align*}
\]

- and from Cathy

\[
\begin{align*}
\text{date} & = \text{Thursday} \\
\text{vclock} & = \text{Alice:1, Ben:1, Cathy:1, Dave:2}
\end{align*}
\]

- Conflict? No. Cathy’s clock is successor of Ben’s
Done

Conflicts and Their Resolution

• Assume two or more conflicting versions of the same object/item.

• What can the database do?
  – Limited possibilities since application logic is not known. E.g., take most recent one

• What can client software do?
  – Full-fledged resolution, since app logic is known.
Conflict Resolution: Example

• Typical use case at Amazon
• Multiple versions of shopping cart
  – merged by a union of their contents
  – what can go wrong? might put back a deleted item (but you wont miss any items=>don’t loose money)

• Let’s see how one would work with multiple versions in a real system
Riak

• Key/Value store
• With namespaces (buckets)
• Queries:
  – CRUD
  – MapReduce
  – Riak Search (i.e., full text search engine)
  – Support of secondary indices

http://basho.com/riak/
Riak Architecture

- Set of equal nodes (no master)
- Placement of data: consistent hashing (will see later)
- Replication (default: 3 per object)
- Fault tolerant

- Various different setups (choices) for consistency: R, W, number of copies, etc.

http://docs.basho.com/riak/1.2.1/references/appendices/concepts/Eventual-Consistency/
API: GET

- curl -v http://127.0.0.1:8098/riak/test/doc

- Response: HTTP/1.1 200 OK
- Plus: Content of the document

- But could also end up with
- HTTP/1.1 300 Multiple Choices ...
- Plus: a number of versions ...

http://docs.basho.com/riak/1.2.1/references/apis/http/HTTP-Fetch-Object/
Riak: Siblings

Siblings:
16vic4eU9ny46o4KPiDz1f
4v5xOg4bVwUYZdMkqf0d6I
6nr5tDTmhxnwuAFJDd2s6G
6zRSZFUJIHXZ15o9CG0BYl

http://docs.basho.com/riak/1.2.1/references/apis/http/HTTP-Fetch-Object/
Riak: Get Specific Version

• `curl -v http://127.0.0.1:8098/riak/test/doc?vtag=16vic4eU9ny46o4KPiDz1f`
Get ALL Versions


--YinLMzyUR9feB17okMytgKsylvh Content-Type: application/json Link: </riak/test>; rel="up" Etag: 6nr5tDTmhxnwuAFJDd2s6G Last-Modified: Wed, 10 Mar 2010 17:58:08 GMT {"bar":"baz"}

--YinLMzyUR9feB17okMytgKsylvh Content-Type: application/json Link: </riak/test>; rel="up" Etag: 6zRSZFUFJlHXZ15o9CG0BYl Last-Modified: Wed, 10 Mar 2010 17:55:03 GMT {"foo":"bar"}

.....
Java Client for Riak

• Gives good overview of what it takes to work with Riak. Particularly in terms of managing conflicting (multiple) versions.

• https://github.com/basho/riak-java-client
DATA PLACEMENT: CONSISTENT HASHING
Hash Based Data Placement

• Use of hash function f to place data to machines
  – m machines, placement based on f(key)
  – e.g., f(key) := U*key + C mod m

• For instance in NoSQL key/value stores, but also in general for assigning data by key to machines
Problem: Moving Data Around when Adding/Removing Machines

- Assume data: [13, 34, 11, 9]
- Function: \( f(k) := 17 \cdot k \mod m \)
Wish List for Hashing Properties

- Only *local data movement* if machines are
  - added or
  - removed

- Load balancing: *Strong machines can get larger share* of data/work
Cyclic Identifier Space

64→0
Place Servers on Ring
Place Data to Servers
Added Server (id 20)
Removed Server (id 55)
Consistent Hashing: Formal Definition

- Given a set of items $I$ and a set of buckets $B$
- A view $V$ is any subset of $B$
- A hash function is given as $f: 2^B \times I \rightarrow B$
- $f(V, i)$ is bucket $I$ is mapped to given (available) buckets (view) $V$

Consistent Hashing: Formal Properties

- **Balance**: with high probability, each bucket gets \( O(|I|/|V|) \) items assigned

  *Means: buckets get roughly the same load in terms of number of items assigned*

- **Monotonicity**: Given views \( V_1, V_2 \) with \( V_1 \) subsetOf \( V_2 \)

  Then \( f(V_2, i) \) in \( V_1 \) implies \( f(V_1, i) = f(V_2, i) \)

  *Means: if a new bucket (node) is added, an item might move from an old bucket to a new one, but never from an old one to another old one.*
Consistent Hashing: Implementation

• Given two random hash functions:
  – $r_V$ maps $V$ to the unit interval
  – $r_B$ maps $B$ to the unit interval

• Then
  – $f(V,i)$ should map item $i$ to bucket $b$ that minimizes $|r_V(i)-r_B(b)|$
  – That means: it maps to bucket that is closest to $i$
Used In

- NoSQL systems like Amazon’s Dynamo, Riak
- Chord, a distributed hashtable (will see later again in Peer-to-Peer systems)
Literature