Logical Ordering of Events in Distributed System

• Last lecture we looked at consensus algorithm(s) to agree on execution order of requests for State Machine Replication.

• Now: logical (partial) ordering of events

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

Ordering of Events

• Consider events of a specific process.
• We assume the events form a sequence, so it is known which one appears before another.
• If event a occurs before b in this sequence, then a “happens before” b.

• How does this translate to multiple processes?
The “Happens Before” Relation →

• The relation “→” is the smallest relation such that:
  – (1) If a and b are events in the same process, and a comes before b, then a→b
  – (2) If a is the sending of a message by one process and b is the receipt of the same message by another process, then a→b
  – (3) If a→b and b→c then a→c.

• Two distinct events are called concurrent if a↛b and b↛a
a → b if one can get from a to b by following processes or messages. For instance, $p_1 \rightarrow r_4$. 

*Image source: L. Lamport.*
Causally Affecting

• Another way of viewing the definition of $a \rightarrow b$ is to say that $a \rightarrow b$ means that it is possible for event $a$ to affect event $b$.

• Two events are concurrent if neither can causally affect the other. For instance, $p_3$ and $q_3$ are concurrent (previous Figure).

P cannot know what process Q did at $q_3$ until it receives the message at $p_4$.
Lamport Timestamps (Logical Clocks)

- 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

then \( C(x) < C(y) \), i.e., \( x \rightarrow y \Rightarrow C(x) < C(y) \)

This is called **Clock Condition** in Lamport’s paper.

• But **not**: \( C(x) < C(y) \Rightarrow x \rightarrow y \)
  (we will see later, that **vector clocks** can say this!)

• Why?
Total Order

• The relation $\rightarrow$ defines only a **partial order**.
• How can we **break ties of “concurrent” events**?
• Assume we have an (arbitrary) total order $\prec$ of the processes.
• Define **relation $\Rightarrow$** as follows:
  – If $a$ is an event in process $P_i$ and $b$ is event in $P_j$, then $a \Rightarrow b$ if and only if either
    – (i) $C_i(a) < C_j(b)$ or
    – (ii) $C_i(a) = C_j(b)$ and $P_i \prec P_j$
Replicated State Machines with Logical Clocks

- **Can build a replicated state machine using logical clocks.**
- See tutorial by F. Scheider* (Section 3)
- Idea: Use total order described before as order the state machines use
- Needs “stability”: Process next stable request with lowest identifier
- **Stable means:** No other request with lower timestamp can arrive after the request.

Recap: Concurrency Control in DBS

• Addresses the problem of dealing with **concurrent reads/writes** to same data
• **Transactions** encapsulate batch of commands
• [http://www.youtube.com/watch?v=G3xH2SoMO_F0](http://www.youtube.com/watch?v=G3xH2SoMO_F0)

• 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
  – ...
Recap: 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** (and variants)

• Or: **PAXOS** commit protocol (for higher reliability)
Distributed Execution Monitor

- Begin_transaction, Read, Write, Commit, Abort
- Results
- Scheduling/Descheduling Requests
- To Data Processors (DPs)
- With other TMs
- With other SCs
- With Data Processors (DPs)

Transaction Manager (TM)

Scheduler (SC)
Synchronization based on Timestamp Ordering (T/O)

- Transaction (TA) gets **timestamp** assigned
- Timestamps impose ordering of transactions.
- I.e. one specific order of TA is chosen (unlike with locking!)

- **Conflicts** are resolved by resetting TAs (which gets a new timestamp). There are no deadlocks.
- Creates conflict serializable schedules.

- Record **for each record** (item) the **largest timestamp** of any read or write operation.
(Pessimistic) Timestamp Ordering

- With each database item two timestamp (TS) values are associated

- **read_TS(X):** The read timestamp of item X; i.e., largest TS of all transactions that have successfully read item X

- **write_TS(X):** The write timestamp of item X; i.e., largest of all TS of transactions that have successfully written item X
Conflicts in (Pessimistic) T/O

- **R/W**: If TAT with timestamp TS(T) wants to **read** X
  - reject if TS(T) < write_TS(X)
  - otherwise: read and set read-timestamp of record x to max(TS, read-timestamp of x)

- **R/W**: If TAT with timestamp TS(T) wants to **write** X
  - reject if TS(T) < read_TS(X)
  - otherwise: write and set write-timestamp of x to max(TS, write-timestamp of x)

- **W/W**: If a TAT with timestamp TS(T) wants to **write** X
  - reject if TS < write_TS(X)
  - otherwise: write and set write-timestamp to TS
Thomas’s Write Rule

- Modification of basic T/O algorithm
- Rejects fewer writes. No conflict serializability.
- If read_TS(X) > TS(T) then abort / reject
- If write_TS(X) > TS(T) then do not execute operation, but continue.
- If neither the condition (1) or (2) occurs, execute write_item(X) and set write_TS(X) to TS(T).

- Also called: “Last Write Wins”
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**

Alice

Bob

$t_0$ $t_1$

$v_0$ $v_1$ $v_1$

Read $v_0$

Write $v_0 \rightarrow v_{1a}$

Write $v_0 \rightarrow v_{1b}$

Conflicts:
- $v_{\text{latest}} = v_0$
- $v_{\text{latest}} = v_{1a}$
- $v_{\text{latest}} \neq v_0$
Distributed Implementation

• Basic implementation

• 2PL can be handled in one central instance, called centralized 2PL (C2PL), or as distributed 2PL (D2PL) with lock managers on each site

• Requires distributed deadlock management

• Timestamp Ordering rather straightforward to realize

• 2PC etc. for agreeing on commit (before changes are made permanent)
Discussion

• **Strong consistency** as we know it from traditional database systems is achievable also in distributed fault-tolerant systems

• But it **comes with a price to pay**

• Synchronization is costly, thus, increases response times

*Literature for the aforementioned techniques:*
- standard DB books like the one by Ramakrishnan and Gehrke (Database Management Systems) or the one by Elmasri and Navathe (Fundamentals of Database Systems).
- distributed case specifically, book by Özsu and Valduriez (Principles of Distributed Database Systems)
Going Distributed: Some Thoughts

• What happens if network gets partitioned, e.g., falls into two disconnected sub networks?

• What about clients that want to read/write data?

• **Do we allow** that (they pick one partition) or **do we have to wait** (how long?) for the problem to be fixed?
High Availability Aim

• Availability/Response Time (Latency)

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

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

CAP Theorem (Brewer's Theorem)

- System **cannot provide all 3** properties at the same time:
  - **C**onsistency
  - **A**vailability
  - **P**artition Tolerance

- After **Eric Brewer**, as a conjecture presented at PODC 2000, later formally proved by Seth Gilbert and Nancy Lynch

CAP Theorem (Cont’d)

• **Consistency**: All nodes/clients see same data at same times.

• **Availability**: System is responding to requests

• **Partition Tolerance**: In a distributed system (scale-out) data is put on various nodes, network problems can create partitions of nodes. System has to be able to live with this.
In Different (Older) Words ....

• "Partitioning - When communication failures break all connections between two or more active segments of the network ... each isolated segment will continue ... processing updates, but there is no way for the separate pieces to coordinate their activities. Hence ... the database ... will become inconsistent. This divergence is unavoidable if the segments are permitted to continue general updating operations and in many situations it is essential that these updates proceed."

• [Rothnie & Goodman, VLDB 1977]

CAP Theorem: Proof Idea

• Consider a system with multiple partitions
• Failure prevents sync between node1 and node2.

• Now?
  – Prohibit reads until synced? (system not available)
  – Or let clients read (system not consistent)
Consistent + Available

- No support of multiple partitions.
- Strong (ACID) consistency enforced.

- **Example:** Single-site database
Partition Tolerant + Available

- **Example:** Domain Name Service (DNS) and Cloud services (Facebook msg. or Twitter)
Consistent + Partition Tolerant

- **Example:** Distributed Databases with distributed concurrency control or banking applications (ATM)
Can’t do without “P”

• Large data => Scale out architecture => Partition Tolerance is a strict requirement

• Leaves: **Trading off consistency and availability**
Is Weaker Consistency Acceptable?

• Weaker “isolation” produces errors. Why is this ok?

• Some guesses:
  – Anyway, DBs are inconsistent for many other reasons:
    • Bad data entry, bugs, duplicate requests, disk errors, ....
  – Maybe errors due to weaker isolation levels are infrequent
  – When DB consistency matters a lot, there are external controls:
    • People look closely at their paychecks
    • Financial information is audited
    • Retailers take inventory periodically

Best effort: BASE

- **Basically Available**
- **Soft State**
- **Eventual Consistency**

- W. Vogels. Eventually Consistent. ACM Queue vol. 6, no. 6, December 2008.
Anecdote: According to http://www.julianbrowne.com/article/viewer/brewers-cap-theorem Prof. Brewer said in personal communication:

"the term "BASE" was first presented in the 1997 SOSP article that you cite. I came up with acronym with my students in their office earlier that year. I agree it is contrived a bit, but so is "ACID" – much more than people realize, so we figured it was good enough. Jim Gray and I discussed these acronyms and he readily admitted that ACID was a stretch too – the A and D have high overlap and the C is ill-defined at best. But the pair connotes the idea of a spectrum, which is one of the points of the PODC lecture as you correctly point out."
Wide Consistency Spectrum

• Extreme points:
  – Strong Consistency
  – Eventual Consistency

• And various intermediate “levels” of consistency

• We will first have a look at eventual consistency
Idea

• **Trade off consistency and availability**

• **Read/Write Pattern:**
  – Guarantee write successful to subset of machines
  – Read data from subsets of machines

• Maintain multiple versions per data item

• Resolve conflicts based on version mismatch once it occurs (be optimistic)
(Logical) Anatomy of a Write

Client

write request with $W=2$

Coordinator

write requests to all

OK, if received 2 ACKs

node1

node2

node3
(Logical) Anatomy of a Write (Cont’d)

• 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 out of the N nodes**.

• Can lead to multiple conflicting versions of a data item, depending on configuration.
• Needs to detected and resolved
W < N

time it takes for write (or sync to arrive at node3)
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

![Diagram showing the process of write and read operations across multiple nodes.](image)
Consistency by Read From All

- Wait for one write to be ack’ed: W=1
- Read from all: R=N
- Guarantee to see the last version

```
node1
D1
```

```
node2
D0
```

```
node3
D0
```

```
node4
D0
```
WARS Model (of Amazon Dynamo)

Coordinator

send to N replicas

WRITE
(W)

ACK
(A)

wait for W responses

time

Replica(s)

send to N replicas

READ
(R)

RESPONSE
(S)

wait for R responses
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 ≤ N

• What if W < (N+1)/2? Write sets do not overlap. What does this mean?
## 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&lt;=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?

• Goal is to **quantify eventual consistency**

• **Probability that we miss reading the last write.**

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

• What does that mean for real systems? (Let’s put in some numbers .....)

---

*Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, Ion Stoica: Probabilistically Bounded Staleness for Practical Partial Quorums. PVLDB 5(8): 776-787 (2012)*
Putting in Some Numbers

<table>
<thead>
<tr>
<th>N</th>
<th>W</th>
<th>R</th>
<th>$p_s$</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0.667</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>0.883</td>
</tr>
<tr>
<td>30</td>
<td>5</td>
<td>5</td>
<td>0.400</td>
</tr>
<tr>
<td>30</td>
<td>10</td>
<td>10</td>
<td>0.006149306</td>
</tr>
<tr>
<td>50</td>
<td>20</td>
<td>20</td>
<td>6.37503E-07</td>
</tr>
<tr>
<td>100</td>
<td>30</td>
<td>30</td>
<td>1.884E-06</td>
</tr>
</tbody>
</table>

- Does this look promising? Well, for some cases at least.
- Asymptotically is appears to work out well.
More Probabilities

• How likely is it that we get a version that is not older than $k$ versions (form now)?

\[ p_{sk} = \left( \frac{\binom{N-W}{R}}{\binom{N}{R}} \right)^k \]

• When $N=3$, $R=W=1$, this means that the probability of returning a version within 2 versions is 0.556, within 3 versions, 0.703, 5 versions, 0.868, 10 versions, 0.98.

• **Note that this is assuming that the set of updated replicas is not growing over time.**

• There are more complex models in the paper to handle other cases.
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)
Vector Clocks

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

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

**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
Comparing Two Vector Clocks

- $\text{VC}_1 = \text{VC}_2,$
  
  iff $\text{VC}_1[i] = \text{VC}_2[i]$, for all $i = 1, \ldots, n$

- $\text{VC}_1 \leq \text{VC}_2,$
  
  iff $\text{VC}_1[i] \leq \text{VC}_2[i]$, for all $i = 1, \ldots, n$

- $\text{VC}_1 < \text{VC}_2,$
  
  iff $\text{VC}_1 \leq \text{VC}_2 \land$
  
  $\exists j (1 \leq j \leq n \land \text{VC}_1[j] < \text{VC}_2[j])$

- $\text{VC}_1$ is concurrent with $\text{VC}_2$
  
  iff (not $\text{VC}_1 \leq \text{VC}_2$ AND not $\text{VC}_2 \leq \text{VC}_1$)
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 also exchanges email with Ben, and they decide on Tuesday. Also, Dave discuss alternatives with Cathy, and they decide on Thursday instead. 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.”

“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 also exchanges email with Ben, and they decide on Tuesday. Also, Dave discusses alternatives with Cathy, and they decide on Thursday instead. 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 = Tuesday
vclock = Alice:1, Ben:1

Now Dave and Ben start talking. Ben suggests Tuesday:

“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 also exchanges email with Ben, and they decide on Tuesday. Also, Dave discusses alternatives with Cathy, and they decide on Thursday instead. 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.”

“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 also exchanges email with Ben, and they decide on Tuesday. Also, Dave discuss alternatives with Cathy, and they decide on Thursday instead. 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 = Thursday
vclock = Alice:1, Cathy:1

Now Cathy is suggesting Thursday

“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 also exchanges email with Ben, and they decide on Tuesday. Also, Dave discuss alternatives with Cathy, and they decide on Thursday instead. 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.”
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.
Dave Resolves Conflict …

• ... by choosing Thursday

  date = Thursday
  vclock = Alice:1, Ben:1, Cathy:1, Dave:2

• New clock tells that it is successor of the two previous clocks!
date = Thursday
vclock = Alice:1, Ben:1, Cathy:1, Dave:2
Ben informed Alice about Tuesday, Dave informs Cathy and Alice about Thursday. That means, Alice gets two different proposals! A problem?

Again Conflict?

• Alice gets from Ben

\[
\text{date} = \text{Tuesday} \\
\text{vclock} = \text{Alice:1, Ben:1, Dave:1}
\]

• and from Cathy

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

• **Conflict? No.** Cathy’s clock is successor of Ben’s
And we are 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 won’t miss any items=>don’t lose money)

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

• Key/Value store
• With namespaces (buckets)
• Queries:
  – CRUD (create, retrieve, update, delete)
  – 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/
Riak: Parameters Last Write Wins and Allow Multiple

• **multiple versions = false**
  – Last write wins = false: then use “time” annotations to objects and TAs for conflict resolution
  – Last write wins = true: then just consider last write.

• **multiple versions = true**
  – Last write wins = false: then retain even concurrent writes, client (application) has to resolve
  – Last write wins = true: Don’t do this, unpredictable behavior.....

See: [http://docs.basho.com/riak/2.0.1/dev/using/conflict-resolution/](http://docs.basho.com/riak/2.0.1/dev/using/conflict-resolution/)
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 …
Riak: Siblings – Different Versions

siblings:
16vic4eU9ny46o4KPiDz1f
4v5xOg4bVwUYZdMkqf0d6I
6nr5tDTmhxnwuAFJDd2s6G
6zRSZFUJJHXZ15o9CG0BYI

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=16vic4eU9ny46o4KP_tDzf1f
Get ALL Versions


--YinLMzyUR9feB17okMytgKsylvh  Content-Type: application/json  Link: </riak/test>; rel="up"  Etag: 6nr5tDTmhxnwuAFJDDd2s6G  Last-Modified: Wed, 10 Mar 2010 17:58:08 GMT  {"bar":"baz"}

--YinLMzyUR9feB17okMytgKsylvh  Content-Type: application/json  Link: </riak/test>; rel="up"  Etag: 6zRSZFUJIHXZ15o9CG0BYI  Last-Modified: Wed, 10 Mar 2010 17:55:03 GMT  {"foo":"bar"}

.....
Comments

• 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

• Next to vector clocks Riak supports also dotted version vectors

More details on conflict resolution in a more recent version of Riak:
http://docs.basho.com/riak/2.0.1/dev/using/conflict-resolution/