BASHO BLOG

Why Vector Clocks are Easy

Posted January 29, 2010 | by Bryan Fink (http://basho.com/posts/author/bryan-fink/)
| Category: Technical Blog (http://basho.com/category/technical/)

January 29, 2010

Vector clocks (http://en.wikipedia.org/wiki/Vector_clock) are confusing the first time you're introduced to them. It's not clear what their benefits are, nor how it is you derive said benefits. Indeed, each Riak developer has had his own set of false starts in making them behave.

The truth, though, is that vector clocks are actually very simple, and a couple of quick rules will get you all the power you need to use them effectively.

The simple rule is: assign each of your actors an ID, then make sure you include that ID and the last vector clock you saw for a given value whenever to store a modification.

The rest of this post will explain why and how to follow that simple rule. First, I'll explain how vector clocks work with a **very** simple example, and then show how to use them easily in Riak.

Vector Clocks by Example

We've all had this problem:

> 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.

The story changes, but the end result is always the same: you ask two people for the latest version of a piece of information, and they reply with two different answers, and there's no way to tell which one is **really** the most recent.

Vector clocks to the rescue, but how? Simple: tag the date choice with a vector clock, and then have each party member update the clock whenever they alter the choice. Start with Alice's initial message:

```
date = Wednesday
vclock = Alice:1
```

Alice says, "Let's meet Wednesday," and tags that value as the first version of the message that she has seen. Now Dave and Ben start talking. Ben suggests Tuesday:

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

Ben left Alice's mark alone, but added a mark specifying that it was the first version of the message that he had seen. Dave replies, confirming Tuesday:

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

Just like Ben's modification, Dave just adds his own first-revision mark. Now Cathy gets into the act, suggesting Thursday:

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

But wait, what happened to Ben's and Dave's marks? Cathy didn't have a version of the object that had been modified by Ben or Dave, so their marks can't appear in her modification. This means that **Dave** has two conflicting objects:

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

and

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

Dave can tell that these versions are in conflict, because neither vclock “descends” from the other. In order for vclock B to be considered a descendant of vclock A, each marker in vclock A must have a corresponding marker in B that has a revision number greater than or equal to the marker in vclock A. Markers not contained in a vclock can be considered to have revision number zero. So, since the *Tuesday* value has a Cathy revision of zero while *Thursday* has a Cathy revision of one, *Tuesday* cannot descend from *Thursday*. But, since *Thursday* has Ben and Dave revisions of zero while *Tuesday* has Bend and Dave revisions of one, *Thursday* is also not descended from *Tuesday*. Neither succeeds the other, so Dave has a conflict to sort out.

Luckily, Dave’s a reasonable guy, and chooses Thursday:

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

Dave also created a vector clock that is successor to all previously-seen vector clocks: it has revision numbers for every actor equal to or greater than the last revision number he saw for that actor. He emails this value back to Cathy.

So now when Alice asks Ben and Cathy for the latest decision, the replies she receive are, from Ben:

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

and from Cathy:

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

From this, she can tell that Dave intended his correspondence with Cathy to override the decision he made with Ben. All Alice has to do is show Ben the vector clock from Cathy’s message, and Ben will know that he has been overruled. (Dave will, almost certainly, blame his broken email software for failing to inform Ben of the change.)

How to do this in Riak

Now that you understand vector clocks, using them with Riak is easy. I’ll use the raw HTTP interface to illustrate.

First, whenever you store a value, include an `X-Riak-ClientId` header to identify your actor. For Alice’s first message above, you’d say:

```
curl -X PUT -H "X-Riak-ClientId: Alice" -H "content-type: text/plain"
http://localhost:8098/raw/plans/dinner --data "Wednesday"
```

When Ben, Cathy, and Dave each GET Alice’s plans, they’ll get the same vector clock (I’ve removed some of the other headers for brevity):

```
curl -i http://localhost:8098/raw/plans/dinner
HTTP/1.1 200 OK
X-Riak-Vclock: a85hYGBgzGDKBVIsrLnh3BlMiYx5rAzLJpw7wpcFAA==
Content-Type: text/plain
Content-Length: 9

Wednesday
```

The `X-Riak-Vclock` header contains an encoded version of a vclock that is the same as out earlier example: Alice has modified this value once.

Now when Ben sends his change to Dave, he includes both the vector clock he pulled down (in the `X-Riak-Vclock` header), and his own `X-Riak-Client-Id:`

```
curl -X PUT -H "X-Riak-ClientId: Ben" -H "content-type: text/plain"
-H "X-Riak-Vclock: a85hYGBgzGDKBVIsrLnh3BlMiYx5rAzLJpw7wpcFAA=="
http://localhost:8098/raw/plans/dinner --data "Tuesday"
```

Dave pulls down a fresh copy, and then confirms Tuesday:

```
curl -i http://localhost:8098/raw/plans/dinner
...
X-Riak-Vclock: a85hYGBgymDKBVIsrLnh3BlMiYx5rAymfeeO8EGFWRLl30GF/00ACmcBAA==
...
curl -X PUT -H "X-Riak-ClientId: Dave" -H "content-type: text/plain"
-H "X-Riak-Vclock: a85hYGBgymDKBVIsrLnh3BlMiYx5rAymfeeO8EGFWRLl30GF/00ACmcBAA=="
http://localhost:8098/raw/plans/dinner --data "Tuesday"
```

Cathy, on the other hand, hasn't pulled down a new version, and instead merely updated the plans with her suggestion of Thursday:

```
curl -X PUT -H "X-Riak-ClientId: Cathy" -H "content-type: text/plain"
-H "X-Riak-Vclock: a85hYGBgzGDKBVIsrLnh3BlMiYx5rAzLJpw7wpcFAA=="
http://localhost:8098/raw/plans/dinner --data "Thursday"
```

(That's the same vector clock that Ben used, in that encoded gibberish is making your eyes cross.)

Now, when Dave goes to grab this new copy (after Cathy tells him she has posted it), he'll see one of two things. If the "plans" Riak bucket has the `allow_mult` property set to `false`, he'll see just Cathy's update. If `allow_mult` is `true` for the "plans" bucket, he'll see both his last update and Cathy's. I'm going to show the `allow_mult=true` version below, because I think it illustrates the flow better.

```
curl -i -H "Accept: multipart/mixed" http://localhost:8098/raw/plans/dinner
HTTP/1.1 300 Multiple Choices
X-Riak-Vclock: a85hYGBgzWDKBVIsrLnh3BlMiYx5rAymfeeO8EGFWRLl30GF1fsRwsypF59BhT0mIoTZ/1SYQIUrEcJszUksu9R6kCWyAA==
Content-Type: multipart/mixed; boundary=ZZ3eyjUlLBi7GXRRMJsUublFxjn
Content-Length: 368

--ZZ3eyjUlLBi7GXRRMJsUublFxjn
Content-Type: text/plain

Tuesday
--ZZ3eyjUlLBi7GXRRMJsUublFxjn
Content-Type: text/plain

Thursday
--ZZ3eyjUlLBi7GXRRMJsUublFxjn--
```

Dave sees two values because the vclock that Cathy generated wasn't a successor to the vclock that Dave had generated with his last modification. Riak couldn't choose between them, and therefore kept both values.

Dave picks Thursday, and updates the object, resolving the conflict. Riak has already computed a unified, descendant vector clock for Dave, so he uses the vector clock from the multi-value version he just pulled down, just like before:

```
curl -X PUT -H "X-Riak-ClientId: Dave" -H "content-type: text/plain"
-H "X-Riak-Vclock: a85hYGBgzWDKBVIsrLnh3BlMiYx5rAymfeeO8EGFWRLl30GF1fsRwsypF59BhT0mIoTZ/1SYQIUrEcJszUksu9R6kCWyAA=="
http://localhost:8098/raw/plans/dinner --data "Thursday"
```

Now when Alice check for the latest version, she just sees the final decision:

```
curl -i http://localhost:8098/raw/plans/dinner
HTTP/1.1 200 OK
X-Riak-Vclock: a85hYGBgzWDKBVIsrLnh3BlMiYx5rAymfeeO8EGFWRLl30GF1fvhwmzNSSy7lHqgEpUTEerZ/lSYYBFmTr34DCjMBBTOnQwUzgIA
Content-Type: text/plain
Content-Length: 7

Thursday
```

While Riak couldn't decide whether to choose Cathy's modification over Dave's earlier modification, it was easy to choose Dave's latest modification, because the vclock created was a successor to the vclock in place.

Review

So, vclocks are easy: assign each of your actors an ID ("Alice", "Ben", "Cathy", and "Dave" in these examples), then make sure you include that ID and the last vector clock you saw for a given value whenever to store a modification.

If two actors store changes with vector clocks that don't descend from each other, Riak will store and hand back both values. When descendancy can be calculated, values stored with vector clocks that have been succeeded will be removed.

-Bryan (http://twitter.com/hobbyist)

blog (http://basho.com/tag/blog/), Riak (http://basho.com/tag/riak/), vector clocks (http://basho.com/tag/vector-clocks/)

Share: (https://twitter.com/... text=http://basho.com/posts/... vector-clocks-are-easy/) vector-clocks-are-easy/) vector-clocks-are-easy/) vector-clocks-are-easy/)

RECENT ARTICLES

DC/OS 1.9 AND RIAK MESOS FRAMEWORK (HTTP://BASHO.COM/POSTS/TECHNICAL/DCOS-AND-RIAK-MESOS-FRAMEWORK/)

MARCH 13, 2017 | PAVEL HARDAK (HTTP://BASHO.COM/POSTS/AUTHOR/PAVELHARDAK/)

We are pleased today to announce with Mesosphere, the launch of DC/OS 1.9 and a major upgrade of the integration…

TRADITIONAL "DATA LAKE" APPROACH MAY NOT BE A GOOD CHOICE FOR IOT DATA (HTTP://BASHO.COM/POSTS/TECHNICAL/TRADITIONAL-DATA-LAKE-APPROACH-MAY-NOT-BE-A-GOOD-CHOICE-FOR-IOT-DATA/)

MARCH 1, 2017 | DOROTHY PULTS (HTTP://BASHO.COM/POSTS/AUTHOR/DOROTHY-PULTS/)

The momentum around Apache Spark continues. Spark Summit East was a big success and Basho's own Pavel Hardak was among…

BASHO ACADEMY: INSTALL, CODE AND GO (HTTP://BASHO.COM/POSTS/TECHNICAL/BASHO-ACADEMY-INSTALL-CODE-AND-GO/)

FEBRUARY 27, 2017 | DOROTHY PULTS (HTTP://BASHO.COM/POSTS/AUTHOR/DOROTHY-PULTS/)

CREATE YOUR FIRST RIAK TS TABLE (HTTP://BASHO.COM/POSTS/TECHNICAL/CREATE-YOUR-FIRST-RIAK-TS-TABLE/)

FEBRUARY 21, 2017 | DOROTHY PULTS (HTTP://BASHO.COM/POSTS/AUTHOR/DOROTHY-PULTS/)

Are you new to NoSQL and want to find out more about Riak? Basho Academy may be the right place for...

Customers like Intellicore are doing real-time analysis of IoT data using Riak TS. We want to make it easy for...

PRODUCTS (HTTP://BASHO.COM/PRODUCTS/)
INTEGRATIONS (HTTP://BASHO.COM/PRODUCTS/INTEGRATIONS/)
RESOURCES (HTTP://BASHO.COM/RESOURCES/)
NOSQL EXPLAINED (HTTP://BASHO.COM/RESOURCES/NOSQL-DATABASES/)
INDUSTRIES (HTTP://BASHO.COM/INDUSTRIES/)
USE CASES (HTTP://BASHO.COM/USE-CASES/)
ABOUT BASHO (HTTP://BASHO.COM/ABOUT/)
NEWSROOM (HTTP://BASHO.COM/NEWSROOM/)
BLOG (HTTP://BASHO.COM/BLOG/)
DEVELOPERS (HTTP://BASHO.COM/COMMUNITY)
EVENTS (HTTP://BASHO.COM/EVENTS/)
CAREERS (HTTP://BASHO.COM/CAREERS/)
CONTACT (HTTP://BASHO.COM/CONTACT/)
PARTNERS (HTTP://BASHO.COM/PARTNERS/)

(http://basho.com)