Why Vector Clocks are Easy

January 29, 2010 at 11:15 AM | categories: Riak

Vector clocks 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/1SYQIUrEcJszU
ksu9R6kCWyAA==
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/1SYQIUrEcJszU
ksu9R6kCWyAA==" \
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:
a85hYGBgzWDKBVIsrLnh3BlMiYx5rAymfeeO8EGFWRLl30GF1fvhwmzNSSy71HqgEpUTEerZ/1SYYBFmTr
34DCjMBBTOnQwUzgIA
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

18 Comments

Add New Comment

You are commenting as a Guest. Optional: Login below.

- DISQUS
-

Share this comment on...
Configure options...

Showing 18 comments

Sort by Popular nowBest RatingNewest firstOldest first Subscribe by email Subscribe by RSS

cpb 3 months ago

3 people liked this.

- My question is rather more civil. Besides planning dinners, how else have you used vector clocks Bryan?
- Flag
-
-

Bryan Fink 3 months ago in reply to cpb

2 people liked this.

- Oh, nothing too out of the ordinary. I've used them to improve my mental well-being, to store data sanely in a distributed system, to track evil-doers, to surive weeks stranded in the jungle, to win at rock-paper-scissors … the usual.
- Flag
-
-
-  edlich 1 month ago
- Good example. However I think the sentence "Dave's a reasonable guy" should be explained more. E.g. where does his decision come from to overrule poor Ben?! (after Cathy has spoken for the first time).
- Flag
-
-
-  *Bryan Fink* 1 month ago in reply to edlich
- Indeed, poor Ben. Dave is "reasonable" only in that he's able to choose on his own among the options presented to him. Without writing a novel about it, suppose that Dave was prepared to commit to either Tuesday or Thursday when he sent his reply to Ben, but decided he preferred Thursday when he saw Cathy's choice. The point is that Dave gets to choose, and his intention is obvious to later readers.
- Flag
-
-
-  bewest 2 months ago
- Why not integrate vclock and the agent ID into the 'edit' URI type and provide all objects with edit URI's each time?
- Flag
-
-
-  *Bryan Fink* 2 months ago in reply to bewest
- Interesting idea. Basically just adding another item to the Link header like "</raw/plans/dinner?vclock=VVVV>;rel=edit", right? That could work quite easily. Agent ID is a little different, just

because a read doesn't require and agent ID. But, if one were given at read time, it could be added to the URI easily. The current implementation fit our needs, but I could see this idea working well for some apps.

- Flag
-
-

bewest 1 month ago in reply to Bryan Fink

-

Bryan, right. I found this essential for some applications I've deployed. I got the idea after looking at google gdata documentation http://code.google.com/apis/gdata/docs/1.0/refe.... Apparently v2 uses etags with session support.

Anyway, in my case I just wanted what v1 describes: reject any write request to a stale version. This seems a bit simpler than the scenario you described. Is it easy to do? This made the business clients easy to write. The reason I mention vclock along with the id is because in the scenario I outlined, a GET must always be performed to obtain a workable edit URI. This allowed the client to process the interesting bits and send the result to one of the URIs given. No need to mangle different pieces of data because it's all in the URI. In this kind of system, your canonical URIs may be well known or queryable, but you must perform a GET to obtain the information necessary to edit the object, which is similar to what you've outlined.

Doing this effectively puts resolving version conflicts, currently in headers, into URIs. Gdata v2 uses headers like you guys. I like edit uris because it's one less thing to worry about and I'm lazy.

- Flag
-
-

Bryan Fink 1 month ago in reply to bewest

-

Rejecting write requests to stale versions can be done, but it's done in the HTTP-standard way, using the If-Match header. Just take the VTag header that the GET returned and stick it in the If-Match header of your PUT. You'll end up at a 412 if the object has changed since your GET.

- Flag
-
-

lilgovi 3 months ago

-

"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 higher than the marker in vclock

A."

I think you mean "greater than or equal to the marker in vclock A". If i'm not wrong, this is how the system is intended to work and the riak source confirms:

From vclock.erl:69 -- "remember, a vclock is its own descendant!"

- Flag
-
-

Bryan Fink 3 months ago in reply to tilgovi

-

Correct! Thanks for the catch - I've fixed the post.

(And, you're welcome. I wrote this post for myself as much as anyone. I decided it was time to write down the essentials for quick-reference in answering questions. I'm glad others are getting use out of it as well.)

- Flag
-
-

tilgovi 3 months ago in reply to tilgovi

-

Also, thank you for your excellent post.

- Flag
-
-

Mr. Bean 3 months ago

-

I think you meant to say in your RIAK example that Dave confirms Tuesday as Cathy has not suggested Thursday yet. (see below)

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 Thursday: <<< Should be 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" <<< Isn't this confirming Tuesday?

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

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"

- Flag
-
-


Bryan Fink 3 months ago in reply to Mr. Bean

-

Doh! Good catch. I've updated the post. Thanks!

- Flag
-
-


jmiechiposting 3 months ago

-

Vector clocks are easy from a conceptual standpoint. Implementing a data storage engine that manages vector clocks and allows for complex (e.g., not last-writer wins) reconciliations is not easy. Throw into the mix a focus on availability and scale (in the form of elasticity) and you are really opening Pandora's box. There are several dimensions of elasticity - most people tend to think only about two of them: 1. how far you can stretch and how 2. quickly you can stretch. In your internet data store engine this will translate into how much total data you can manage and how quickly you can add new capacity. Given all of the decisions that you are making you will eventually find that certain optimizations will favor one elastic dimension and negatively impact the other.

Avoid vector clocks within your data storage engine enables you to avoid having to make this trade off.

The short of it is this:

1. vector clocks will complicate your implementation,
2. there really aren't that many applications that actually need more flexible reconciliation capabilities (everyone thinks that they would like this but there really are very few cases in

which it is actually necessary),
3. for those that do need vector clocks and more complex reconciliation push the complexity to them.

Accepting the complexity in your general purpose storage engine is the wrong decision.

- Flag
-
-

Bryan Fink 3 months ago in reply to jmtechpostings

1 person liked this.

-

Hi, jmtechpostings. Thanks for your comments. I'm going to both agree and disagree with you.

First, agreement: implementing a data storage engine that manages vector clocks and allows for complex reconciliations is not easy. Luckily, we've already done this part for you. We called it Riak. You can download it from http://riak.basho.com/

Now on to that second matter. Ever notice how apps get bogged down in transaction locking? How those same apps fail to be able to grow in size and throughput? How they get very sensitive to network connectivity? The reason is that these apps chose storage systems that hack around inherent difficulties in distributed systems by introducing single points of failure and complex protocols that require all members to be connected all the time.

We baked vector clocks into Riak after reading, studying, and learning that they were one way to *actually* solve a problem in distributed storage systems: that of reconciliation of versions of data across a real-world network (one that has unpredictable delays and failures). Clients can perform reads and writes through any node in a Riak cluster. Riak nodes can drop in and out of connection. The data stays intact because vector clocks allow the system to resolve internally which data is correct.

For our work, you get all the benefits of vector clocks with none of the hassle, as long as you do that one simple thing I mentioned in the article: hand back the vector clock you saw on your last read along with your client (actor) ID whenever you store a value. That's it. No complicated hacking necessary. The detail in this article is purely for people, like me, who feel more comfortable using a system that they understand than they do playing the cargo-cult game.

- Flag
-

-  *jmtechpostings* 3 months ago in reply to Bryan Fink

  1 person liked this.

- There are many mechanisms to ensure application level consistency when updating data in a distributed system. The most lightweight and common approach is to apply optimistic concurrency and pass along the specific version of data that you are modifying. This ensures, in a distributed system with all of the failure modes and manifestations, that the data that you are updating has not changed out from underneath you. Other than the "I don't care what has happen and last writer wins" model this is the most common.

  Vector clocks do nothing above and beyond simpler techniques (and by simpler I mean simpler for you the implementer of the data storage engine) other than let you apply application specific conflict reconciliations.

  My comments to you were all about whether or not you should have introduced this complexity into Riak. Your product is still young (based on the little information that I have found on it) so imagine carrying forward and evolving it over the course of the next 5-10 years.

  Speaking from experience, specifically from having been involved in this very same mistake, I would never do it again.

  Anyway, I don't mean to be a downer. Just wanted to share my perspective.

- Flag
-
-  *Bryan Fink* 3 months ago in reply to jmtechpostings

  6 people liked this.

- The optimistic concurrency system you propose fails to provide write availability in the face of unreachable nodes. If a machine is, for any reason, unreachable, the system you suggest *must* reject the write, because it can not know whether or not the data on the unreachable node "has changed out from underneath" the writer.

  A goal that we strived for (and attained!) in building Riak is constant write-availability. It doesn't matter if nodes are unreachable; Riak can perform writes anyway. We gained this ability by using vector clocks. When nodes reconnect to each other, they can compare the vector clocks of their data, and figure out for themselves which is newest.

  Many uses of vector clocks are entirely internal. Riak automatically

heals data on nodes that were down, for example (because the vector clocks on the down node would all be equal to or succeeded by the vector clocks in the live system). It is also possible to tell Riak to just make a last-write-wins-like decision if there ever would be a case where application interaction would be needed to resolve a vector clock conflict -- simplicity where possible!

Coding in vector clocks is about providing functionality that simply is not available in other systems. Speaking from experience, having hacked on Riak and having written applications to run on top of it for two years now, I'm totally happy trading off some simple vclock rules for constant write availability.

- Flag
-
-

jmiechyposting 3 months ago in reply to Bryan Fink

-

You are correct, the concurrency that I described favors consistency over availability. In retrospect it wasn't the best example for me to give since I didn't intend to muddy the waters with CAP tradeoffs. :)

Anyway, I don't think we are actually disagreeing on anything. All of the advantages that you list are valid. My intent was to share my experiences and thoughts on this. Specifically, my opinion is that the advantages of implementing vector clocks in a storage engine do not outweigh the disadvantages. Especially as you move to scale and your system matures.

One minor clarification for you. You wrote: "Coding in vector clocks is about providing functionality that simply is not available in other systems." This is not accurate. LinkedIn's project Voldemort (they have lots of blog postings and one of their lead engineers presented at QCon this past year) and Amazon's Dynamo (see the SSOP paper from '07 here http://s3.amazonaws.com/AllThingsDistributed/so...) both use vector clocks. Both are also systems that are being used at scale.

- Flag
-

Why Vector Clocks are Hard

April 05, 2010 at 03:16 PM | categories: Riak

A couple of months ago, Bryan wrote about vector clocks on this blog. The title of the post was "Why Vector Clocks are Easy"; anyone who read the post would realize that he meant that they're easy for a client to use when talking to a system that implements them. For that reason, there is no reason to fear

or avoid using a service that exposes the existence of vector clocks in its API.

Of course, actually implementing such a system is not easy. Two of the hardest things are deciding what an actor is (i.e. where the incrementing and resolution is, and what parties get their own field in the vector) and how to keep vclocks from growing without bound over time.

In Bryan's example the parties that actually proposed changes ("clients") were the actors in the vector clocks. This is the model that vector clocks are designed for and work well with, but it has a drawback. The width of the vectors will grow proportionally with the number of clients. In a group of friends deciding when to have dinner this isn't a problem, but in a distributed storage system the number of clients over time can be large. Once the vector clocks get that large, they not only take up more space in disk and RAM but also take longer to compute comparisons over.

Let's run through that same example again, but this time visualize the vector clocks throughout the scenario. If you don't recall the whole story in the example, you should read Bryan's post again as I am just going to show the data flow aspect of it here.

Vector Clocks by Example, in detail

Start with Alice's initial message where she suggests Wednesday. (In the diagrams I abbreviate names, so that "Alice" will be "A" in the vclocks and so on for Ben, Cathy, and Dave.)

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

Ben suggests Tuesday:

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

Dave replies, confirming Tuesday:

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

Now Cathy gets into the act, suggesting Thursday:

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

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. Luckily, Dave's a reasonable guy, and chooses Thursday. Dave also created a vector clock that is a successor to all previously-seen vector clocks. He emails this value back to Cathy.

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

So now when Alice asks Ben and Cathy for the latest decision, the replies she receives 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.

That worked out pretty well.

Making it Easier Makes it Harder

Notice that even in this short and simple example the vector clock grew from nothing up to a 4-pairs mapping? In a real world scenario with long-lived data, each data element would end up with a vector clock with a length proportional to the number of clients that had ever modified it. That's a (potentially unbounded) large growth in storage volume and computation, so it's a good idea to think about how to prevent it.

One straightforward idea is to make the servers handling client requests be the "actors", instead of representing the clients directly. Since any given system usually has a known bounded number of servers over time and also usually has less servers than clients, this serves to reduce and cap the size of the vclocks. I know of at least two real systems that have tried this. In addition to keeping growth under control, this approach attracts people because it means you don't expose "hard" things like vector clocks to clients at all.

Let's think through the same example, but with that difference, to see how it goes. We'll assume that a 2-server distributed system is coordinating the communication, with clients evenly distributed among them. We'll be easy on ourselves and allow for client affinity, so for the duration of the session each

client will use only one server. Alice and Dave happen to get server X, and Ben and Cathy get server Y. To avoid getting too complicated here I am not going to draw the server communication; instead I'll just abstract over it by changing the vector clocks accordingly.

We're fine through the first few steps:

The only real difference so far is that each update increments a vector clock field named for the client's chosen server instead of the client itself. This will mean that the number of fields needed won't grow without bound; it will be the same as the number of servers. This is the desired effect of the change.

We run into trouble, though, when Cathy sends her update:

In the original example, this is where a conflict was created. Dave sorted out the conflict, and everything was fine. With our new strategy, though, something else happened. Ben and Cathy were both modifying from the same original object. Since we used their server id instead of their own name to identify the change, Cathy's message has the same vector clock as Ben's! This means that Dave's message (responding to Ben) appears to be a simple successor to Cathy's... and we lose her data silently!

Clearly, this approach won't work. Remember the two systems I mentioned that tried this approach? Neither of them stuck with it once they discovered that it can be expected to silently lose updates.

For vector clocks to have their desired effect without causing accidents such as this, the elements represented by the fields in the vclock must be the real units of concurrency. In a case like this little example or a distributed storage system, that means client identifiers, not server-based ones.

Just Lose a Little Information and Everything Will Be Fine

If we use client identifiers, we're back in the situation where vector clocks will grow and grow as more clients use a system over time. The solution most people end up with is to "prune" their vector clocks as they grow.

This is done by adding a timestamp to each field, and updating it to the current local time whenever that field is incremented. This timestamp is never used for vclock comparison -- that is purely a matter of logical time -- but is only for pruning purposes.

This way, when a given vclock gets too big, you can remove fields, starting at the one that was updated longest ago, until you hit a size/age threshold that makes sense for your application.

But, you ask, doesn't this lose information?

Yes, it does -- but it won't make you lose your data. The only case where this kind of pruning will matter at all is when a client holds a very old copy of the unpruned vclock and submits data descended from that. This will create a sibling (conflict) even though you might have been able to resolve it automatically if you had the complete unpruned vclock at the server. That is the tradeoff with pruning: in exchange for keeping growth under control, you run the chance of occasionally having to do a "false merge"... but you never lose data quietly, which makes this approach unequivocally better than moving the field identifiers off of the real client and onto the server.

Review

So, vclocks are hard: even with perfect implementation you can't have perfect information about causality in an open system without unbounded information growth. Realize this and design accordingly.

Of course, that is just advice for people building brand new distributed systems or trying to improve existing ones. Using a system that exposes vclocks is still easy.

-Justin

6 Comments

---

because in fact Ben and Cathy had created a split by responding independently. If Dave wants to supersede his earlier reply, all he needs to do is tag the new one with X:3 Y:2 so that either Alice or Ben can clearly recognize it as a successor to both and thus as an intentional merge of the two temporarily separate branches (much as was done in the client-clock example).

- Flag
-
-

 *Justin Sheehy* 1 month ago

-

Jeff,

Even in that variant, you will generally create another version of the same problem. If Cathy's message to Dave was X:1,Y:2 then anyone receiving it would think that it descended from Ben's message to Dave (X:1,Y:1) and thus anyone seeing both would silently lose Ben's message due to (falsely) believing that it is an ancestor of Cathy's.

-Justin

- Flag
-
-

 *Jeff Darcy* 1 month ago in reply to Justin Sheehy

-

It might be a problem, but it's certainly not the same problem. In the article text, there's a fairly specific statement about the original problem.

"This means that Dave's message (responding to Ben) appears to be a simple successor to Cathy's... and we lose her data silently!"

In the scenario I described, Ben does not believe that Dave's X:2 Y:1 was a successor of Cathy's X:1 Y:2, and it is Ben's update - not Cathy's - that ends up being discarded. I say discarded, not lost, because that is exactly what should have happened according to Dave's expressed intent. Had Dave intended to let Ben's update stand, discarding Cathy's, he would have sent "Tuesday" instead of "Thursday" to Alice with the same successor-to-both version number. Either way, Dave sorts out the conflict the same way he would have with client-based clocks, and his intended result is preserved. An intentional overwrite is not loss, corruption, or inconsistency.

In a more general sense, you say that anyone receiving Cathy's message would perceive it as a successor to Ben's. In fact, it might be. It's possible that she was completely aware of Ben's update when she issued her own, that it was an intentional overwrite before it even got to Dave, and everything is again as it should be. On the other hand, she might have been unaware of Ben's update, creating a true conflict which still has to be resolved. How? As it turns out, Riak has a feature that can help: X-Riak-Vclock (as described in http://riak.basho.com/edoc/raw-http-

howto.txt). I suppose that it's intended to work with client versions, but it can work with server versions just as well - or just as poorly. In this scenario, Cathy must have read before Ben's update but written after it - and that write should fail because the read had version X:1 and the write requires at least X:1 Y:1. If she re-reads she'll get both Ben's update and the later version number, and can then re-write (or not) using that version number. This is exactly the kind of conditional-write scheme I was referring to in my own post on this subject (http://pl.atyp.us/wordpress/?p=2601) and the downside is the partition problem I mention there.

Considering all these issues, I'm sure many people might wonder if the complexity is worth it. If the information about causality is still imperfect for all of the reasons we've discussed (including clock truncation), is "last write wins" really any worse? How many use that option as described in http://riak.basho.com/edoc/basic-client.txt? How many do the same, or worse, in simplistic client code? How many still use locking (e.g. via Zookeeper) to avoid such conflicts? How many just can't deal with these issues, and cling to their ACID RDBMSes even where they're inappropriate? I suspect that all of these groups are much larger than the set of people who can use *any* flavor of vector clocks effectively. Reducing the potential for data loss within the system might not be a net win if the additional exposed complexity increases the potential for data loss in applications written by people who were already struggling with eventual consistency and CAP tradeoffs and such.

- Flag
-
-

Justin Sheehy 1 month ago in reply to Jeff Darcy

-

Jeff,

Your whole message really boils down to your statement "It's possible that she was completely aware of Ben's update."

If she's using a client-identified vclock, then you know for sure if she was aware of Ben's update or not. In the scenario you described, then you "might" know.

Your addition about how her "write should fail" assumes that concurrent actors are disallowed. That kind of serialized conditional behavior doesn't work in a write-available distributed system.

Regarding your last sentence, I refer the reader back to Bryan's post. The amount of exposed complexity is very small.

-Justin

- Flag
-
-

 *Jeff Darcy* 1 month ago in reply to Justin Sheehy

- "Your whole message really boils down to your statement "It's possible that she was completely aware of Ben's update.""

  That's a bit of an oversimplification. What I was trying to say was that if she was aware then the observed behavior was correct, and if she was not then there are mechanisms available to deal with it.

  "Your addition about how her "write should fail" assumes that concurrent actors are disallowed."

  Only if her write was intended to be conditional, as when she has provided the vclock for the version she intends to supersede. If she wants to do a blind overwrite, she certainly should be able to, though it's not clear whether such a per-request decision can be made when conflict resolution is based on a per-bucket setting.

  "That kind of serialized conditional behavior doesn't work in a write-available distributed system."

  Exactly the point I made on March 21, and referenced again yesterday. That's why I don't think conditional writes solve the problem, and that's true for either client-based or server-based vclocks.

  "Regarding your last sentence, I refer the reader back to Bryan's post. The amount of exposed complexity is very small."

  I refer you in turn to my March 21 post, specifically the "last straw" part. Also, I think the amount of added value is pretty closely proportional to the added complexity. Exposing limited complexity solves only limited problems, where the limits are related to issues such as partitions and clock truncation which we have discussed. By exposing more complexity one can help to solve more complex problems, many of which I would consider the clients' own problems rather than the storage system's, but the point of my last paragraph is that users might well prefer different solutions and find no value in the offered solution. The question, always, is how *users* can benefit from one approach vs. another, and not just be lulled into a false belief that hard problems are easy or solved.

- Flag
-

 Justin Sheehy 1 month ago in reply to Jeff Darcy

- It's now clear that there was a fundamental misunderstanding that led to this long side trip of a conversation.

Vector clocks are not used in Riak to enable conditional writes, for two reasons. The first is that Riak's core does not support conditional writes as that is in opposition to our core write-available approach. (the Webmachine-powered HTTP interface supports a limited variety of conditional writes) The second is that vector clocks make a poor fit (or at least vast overkill) for flagging a conditional write; a fixed-size simple hash is much better, and is what (e.g.) most HTTP servers supporting conditional writes use for their ETag headers.

Of course, we are very much in agreement that developers should be more conscious of the tradeoffs they make with different storage systems. This post was about tools that help them to apply that consciousness.

- Flag
-