All Things Clock, Time and Order in Distributed Systems: Logical Clocks in Real Life

Kousik Nath · Follow

Published in Geek Culture · 45 min read · Mar 10, 2021

Photo by Brooke Lark on Unsplash

Setting The Context

In the previous article, we talked about physical clocks in detail, we discussed that in most of the use cases, for a high scale system, using physical clock time is not the right choice unless the time difference across the nodes in a cluster is bounded by an extreme tight limit which is not a feasible option for many companies.

We discuss an alternative option here — logical clocks, **monotonic counter-based clocks** in short. **This article is going to be very comprehensive and practical system design heavy.**

Don't be in hurry, take enough time to digest the content, understanding is more important as it's very hardcore.

The Rise of Logical Clocks

NoSQL systems have got huge traction in recent few years mostly due to inbuilt horizontal scaling capability. With the explosion of mobile and IoT apps, digital business, e-commerce, and payment solutions in emerging markets, the inflow of data is huge for many companies, it's hard for them to scale up traditional relational systems to tolerate such scale (barring few tech giants like Google, YouTube, Facebook, Twitter, etc who still use many SQL based systems, materials for another post :P).

These systems are designed in a **decentralized** and **clustered** fashion for an extremely high scale. They prefer **availability over consistency. Most of the NoSQL stores are always writable (AP systems in terms of CAP theorem) — in the presence of partition, they are capable to accept writes.** They are not ACID compliant (some NoSQL stores are now offering ACID capability but in a very restricted way and incomparable to Relational Systems) rather **BASE** systems — **Basically Available, Soft State, Eventually Consistent.**

For high availability purposes, these stores maintain several replicas of data across different physical nodes meaning **multiple nodes are allowed to take up concurrent write requests for the same data and they eventually converge to the same version of data at a later point in time.**

The Cost Of Eventual Consistency

There are several issues that could happen in such a system:

- If a network partition happens (a node failure also has similar effect) between two replicas storing the same data, they go out of sync. In a very high-scale environment, the replica nodes would end up with different values for the same key after some time.
- If different concurrent clients update the same key in different replicas at the same time, potentially their values would diverge.
- If a node loses some data or somehow data corruption happens, it goes out of sync with other replicas.

All these above scenarios **create conflicts and anomalies:**

- A **single node** might end up with different versions of value objects for the same key due to concurrent client updates.
- Same key could exist across **different nodes** with different versions of value objects.

For high availability purpose, KV data stores allow such divergence and hence conflict is inevitable.

The scene is akin to git branch merging with conflicts that git might not be able to resolve automatically but a developer can. Allowing conflicts in a system is fine as long as there is some mechanism in place to resolve them otherwise it could cause data loss and **customer would move away to competitors.**

Logical clocks pitch in this scenario and help us to manage ordering of such concurrent updates happening across nodes. **Logical clocks sit at the core of versioned data management in distributed clustered systems.** As we keep on exploring, we'll see how NoSQL key value stores like Riak, Voldemort,

Amazon Dynamo DB etc use variety of logical clocks to resolve such issues at scale in real life.

Logical Clocks

Logical clocks do not give importance to when exactly things happened — they don't understand whether an event happened at March 2, 2021, 10:52 PM or Dec 31, 1967, 04:00 AM, rather in what order the events happened is the main area of focus. In a distributed system, it helps to maintain a consistent order of events across nodes. At the same time, using logical time causes many edge cases and difficult to manage data versioning issues.

Before we dive deep further, let's familiarize ourselves with few important concepts below:

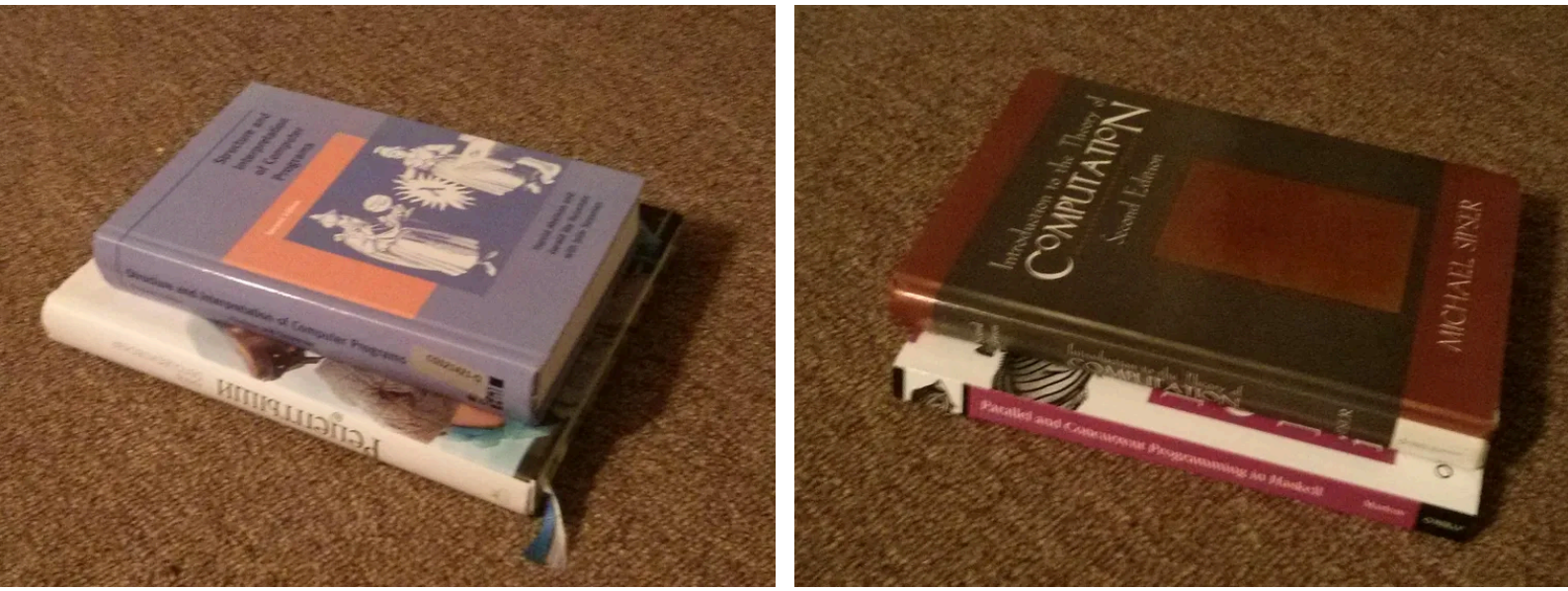


Figure: 1, Left: Total Order, Right: Partial Order, Courtesy: Quora

Causality: Causality in distributed systems means **dependency — relationship between two events.** Two events are **causally related** means one's existence is caused by the other one (*cause-effect*). An event `b` is causally dependent on another event `a` if `b` happens just because `a` has already happened.

Total Order: If two elements can be **deterministically compared**, that's called total ordering. Like in the above left side figure, a smaller book is

placed on top of a larger book. How do we know the top book is smaller? It depends on our definition — we don't care about thickness but as long as a book is smaller in length and width than the other one, we call it smaller. The top book satisfies this condition.

Similarly numbers can be **deterministically compared** (two numbers can be either equal or one is smaller than the other) with the same result every time like: `1 ≤ 2`, `5 ≤ 8` is always true. So numbers represent complete order.

Partial Order: When two elements **can't be compared deterministically**, it's called partial order. Example: in the above right side figure, the top book is equal in length but smaller in width than the book below it. So, we don't have any clear answer whether the top one is smaller than than the bottom one.

Actions: Any node can execute three type of actions: **send message, receive message and execute local action.**

Logical clocks mostly define partial order.

Lamport's Logical Clock

The history of logical clock starts in 1978 when Leslie Lamport published a paper on called *Time, Clocks, and the Ordering of Events in Distributed Systems*. Lamport primarily defined **happens before** relationship "→" which essentially defines **partial ordering** in distributed systems as below:

1. If in a process `P`, an event `a` occurs before another event `b`, then a → b.
2. If a process `P` sends an event to another process `Q`, `a` represents sending of the event by `P` whereas `b` represents receipt of the same event by `Q`, then a → b.
3. If a → b and b → c then a → c i.e; **happens before is a transitive relationship.**

Algorithm

Consider `n` processes `P1`, `P2`, `P3`, ..., `Pn` running in different machines. All of them manage their own monotonic counters — there is no single global counter. All the counters are initialized to `0`. The processes can sync with each other by sending message **m** as shown in figure 2.

The processes can increment the counters at their own pace: `P1` increments its counter by `6`, `P2` increments by `8` and `P3` increments by `10`. In the end of the day, irrespective of how they increment, **they should be monotonic in nature.**

Every process increments its own counter between any two successive events. So **before sending an event over the network, the process increments its counter, similarly after receiving a message, the process increments its counter and deliver the message to the concerned application.**

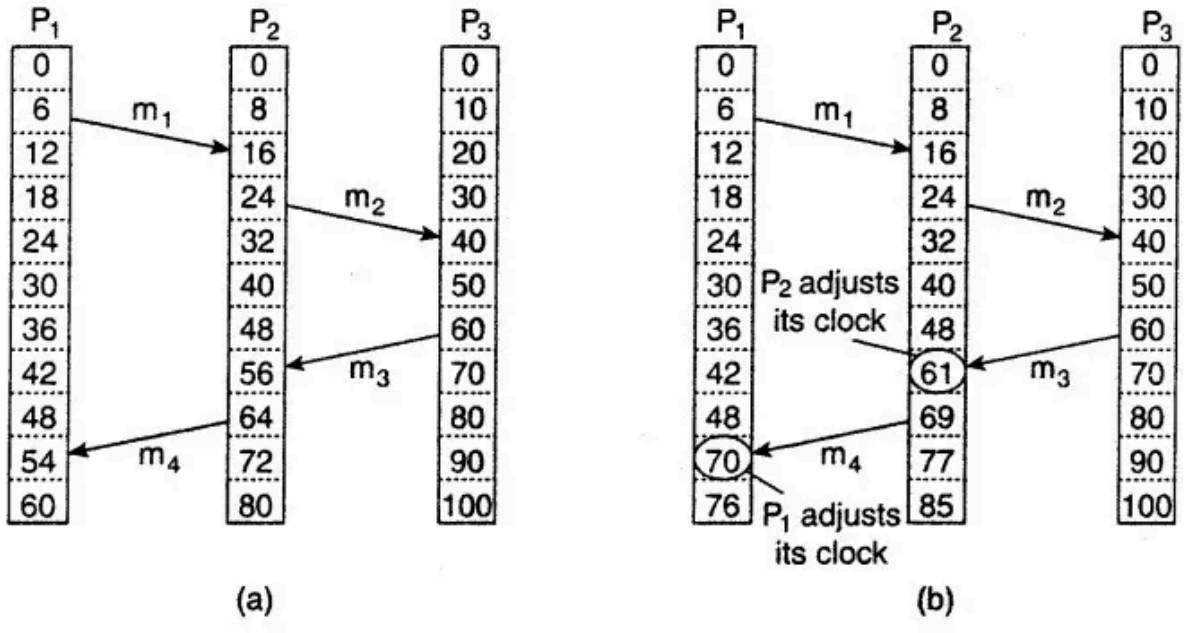


Figure: 2, Lamport Clock

Let,

`C(a)` = the counter value (logical time) when `a` happened.

`C(b)` = the counter value (logical time) when `b` happened.

So,

1. If `a` happens before `b` (a → b) **in the same process,** `C(a) < C(b)` holds true.
2. If `a` is the event of sending a message from a process `Pi` and `b` is the event of receiving the same in another process `Pj`, then clearly `C(a) < C(b)`.
3. If `a` and `b` occurs in two different processes and there is no synchronization (no message passing) between them, neither a → b is true nor b→ a is true. So `C(a)` and `C(b)` can not be correlated, `a` and `b` are considered **concurrent**. So the exact order of events can't be determined in this case, hence this ordering is **partial order**.

How does the algorithm work then? Let's see the steps below:

1. Before executing an event (e.g; sending an event to another process), `Pi` increments its counter `C(i)` i.e; `C(i) = C(i) + x (any non-zero positive value)`. In Figure 2(b), `P3` increments its counter from `50` to `60` before sending the message `m3` to `P2`.
2. `Pi` then sets message `m`'s timestamp `t(m)` to the incremented counter value. In Figure 2(b), `m3` carries a timestamp of `60` now.
3. When the message `m` is received by another process `Pj`, `Pj` first sets its own counter value to `max(Cj, t(m))` where `Cj` represents its current counter value. After that, `Cj` executes step 1 and delivers the message to the application. In the context of Figure 2(b), when `P2` receives the message `m3` from `P3`, its counter is `48`, but `m3` has a timestamp of `60`, so `P2` sets its counter to `max(48, 60) = 60` and increments it to `61` before delivering it to the concerned application.

Q. Why to increment before delivering the message to the application?
A. Since process `P3` sends `m3` with timestamp `60` to `P2`, `P2` should receive it logically at a later timestamp. If we just set the `C(P2)` to `60` and don't increment it, it would look like both the sending and receiving events are concurrent. But we know `P3` sends the message before `P2` receives it, so event at `P3` happened before event at `P2`, hence incrementing `C(P2)` further

by `1` makes the events correlated and satisfy the **happens before** (→) relationship.

Problem With Lamport Clock

We defined if an event `a` happens before another event `b` (a → b) then `C(a) < C(b)`. But the reverse is not true i.e; `C(a) < C(b)` does not mean `a` happened before `b`. **So given Lamport timestamp C(i) of events, it's not possible to derive the causal order among them.**

In Figure 2(b), when `C(P2)` is `8` and `C(P3)` is `10`, there is no way to derive whether the event at `P2` happened before `P3` or vice versa or the events are concurrent since the process counters are not yet synced to each other. It could be very well possible that event at `P3` happened before `P2`.

Vector Clock

Vector clocks are more generalized Lamport clocks. Unlike Lamport clocks, each process in vector clock knows about the counter values in other processes as well. If there are `n` processes, each process maintains **a vector or an array of `n` counters.** When the processes sync up with each other, the counters get updated accordingly.

Note: Although we have mentioned that a **process** maintains a counter, to be generic, we may call it an **actor. An actor is an entity which makes change to an object.** So **an actor could mean a process, a server, a virtual node, a client or a thread etc.** Multiple actors (e.g; multiple processes or threads etc) can run in the same node as well.
actor id represents the id of the associated actor.

Note: *We may use process, server, virtual node and actor interchangeably going forward.*

If an event `a` happened before another event `b` (a → b), vector clock for `b`, `Vb` will be greater than `Va`. Similarly, reverse is also true — unlike Lamport clock, comparing `Va` and `Vb`, we can conclude that `a` happened before `b`.

A vector clock `Vi` is said to be less than another vector clock `Vj` if all the elements of `Vi` are less than or equal to that of `Vj`.

```
Vector Clock Format: [C(Ai), C(Ai+1), C(Ai+2), ..., C(Aj)] where,
C(Ai) = counter value of actor Ai.

Note: Vector clock does not necessarily need to be encoded in an
array , you could store [actor_id, counter] pair in a map also. Array
is used across examples just for better understanding.

Let's say there are 3 actors, Vi and Vj represents the vector clock
of any two of them:

If,
    Vi = [1, 2, 1]
    Vj = [2, 3, 2]
    In this case, since for any k, Vi[k] <= Vj[k], Vi is less than Vj
thus Vi happened before Vj.

Similarly if,
    Vi = [2, 3, 1]
    Vj = [2, 3, 2]
    Vi is still less than Vj thus Vi happened before Vj.
```

Similarly, `Vi` is greater than `Vj` if all the elements in `Vi` is greater than or equal to `Vj`.

```
If,
    Vi = [2, 3, 4]
    Vj = [1, 2, 1]
    Since each of the elements of Vi is greater than that of Vj, Vi
is greater than Vj thus Vj happened before Vi.

If,
    Vi = [2, 3, 4]
    Vj = [2, 3, 1]
    Still Vi is greater than Vj or thus Vj happened before Vi since
the last element of Vi greater than that of Vj and others are same.
```

Algorithm

Consider the following representation where three processes (actors) `P1` , `P2` , `P3` have copy of the same object with different versions. For any two processes `Pi` and `Pj` , `VCi` is the vector clock of Pi, `VCj` represents the vector clock of `Pj` .

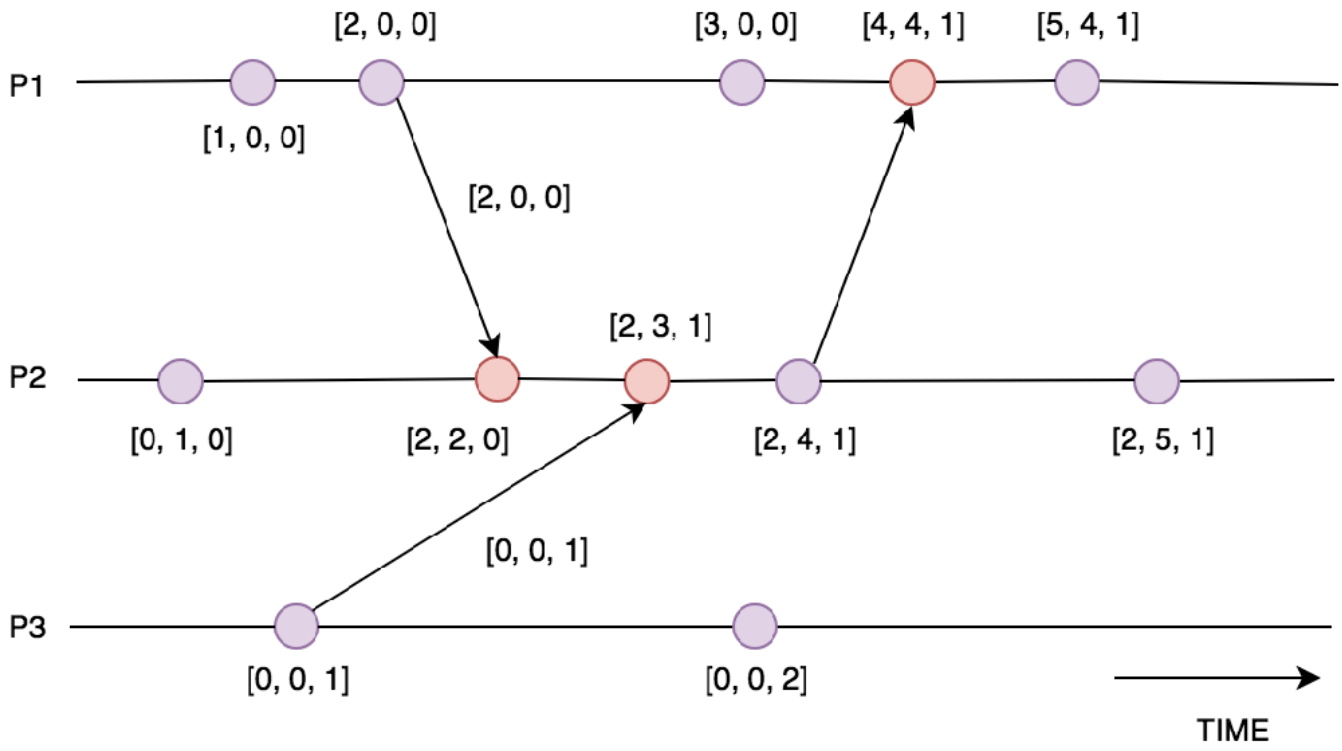


Figure 3: Vector Clock for a single object

1. Before executing any event (excluding the event of receiving a message from another process), `Pi` increases its own counter by executing `VCi[i] = VCi[i] + 1` . In the above figure, when `P1` is at `[1, 0, 0]` , it executes some event and its vector clock becomes `[2, 0, 0]` .

2. When `Pi` sends a message `m` to `Pj` , the timestamp `t(m)` of the message is set to `VCi` after step 1 executes i.e; the message timestamp is nothing but the vector clock of `Pi` . In the above figure, `P1` updates the message timestamp and sends the message to `P2` .

3. After `m` is received, `Pj` updates each element of its own vector clock to the maximum of current value and received value i.e;
   `VCj[k] = max { VCj[k], t(m)[k] }` .
   Then the first steps is executed and the message is delivered to the concerned application. In the above figure, just before receiving the message sent by `P1` , the vector clock at `P2` was `[0, 1, 0]` , after receiving

the message with timestamp `[2, 0, 0]`, the vector clock at `P2` becomes `[max(0, 2), max(1, 0), max(0, 0)] = [2, 1, 0]`, while delivering the message, `P2` increases its own counter, so the final vector clock becomes `[2, 2, 0]`.

Q. Does a process manage a global counter for all the objects it's processing or counters are local to objects?
A. Usually they are local to the objects. Every object or key has some associated logical clock information stored along with it. So the clock information is local to the objects.

Concurrent Vector Clocks

We have already seen how to compare two vector clocks. If some elements `Vi[k]` in a vector clock `Vi` are greater than the corresponding elements in another clock `Vj` and rest of the elements in `Vi` are lesser than that of `Vj`, then `Vi` and `Vj` are called **concurrent vector clocks** because the exact order of the clocks can not be determined. An Example below:

```
If,
    Vi = [2, 3, 2]
    Vj = [1, 2, 4]

Since the first two elements of Vi are greater than Vj, but the
last one is lesser than that of Vj, neither Vi is lesser than Vj nor
Vj is lesser than Vi. They are concurrent.

Similarly if,
    Vi is exactly same as Vj, they are concurrent as well. Some
people prefer to call them identical vector clocks also.
```

Q. What does concurrency mean here, do the events happen at the exact same time?
A. No, here concurrent does not always mean events happening at the same moment, rather it's a time window. The window can represent the same moment or even a period of hours or more. Over a period, **if two clocks don't sync with each other (probably due to network partition, power failure,**

abrupt termination etc) , they lose track of each other which results into concurrency.

Examining two vector clocks, we can derive their causal order — whether one happened before the other or they are concurrent, but as we saw earlier, we could not do the same with Lamport clock. Hence vector clock is more generalized Lamport clock.

Version Vector

Version vector is not vector clock.

Often version vector is used to refer to vector clock, thus creating some confusion. They are not exactly same though, let's see the plausible differences:

Let's say in a two nodes distributed database system, node `2` receives couple of updates for a particular piece of data from node `1` at some point in time and the events in that node look like this in form of vector: `[[1, 2], [2, 1]] -> [[1, 3], [2, 2]] -> [[1, 4], [2, 5]] -> [[1, 8], [2, 10]]` etc.

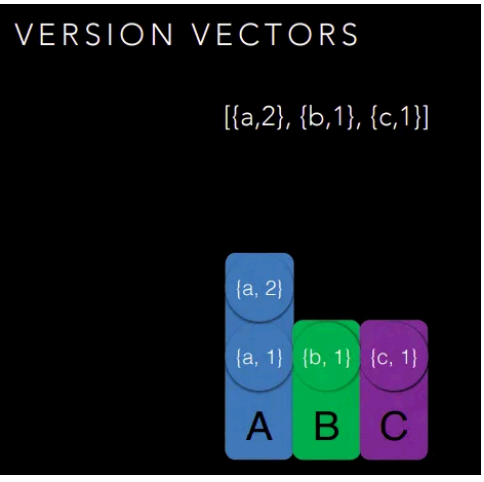
- If you have to derive **causality among all the events** that you have, **vector clock** could be used. So, if you have potentially ever growing events occurring across processes in a distributed system, **the vector clocks will also grow accordingly in size and quantity**. In the above example, `[[1, 2], [2, 1]] -> [[1, 3], [2, 2]] -> [[1, 4], [2, 5]] -> [[1, 8], [2, 10]]` , we can easily derive how all the update events from node `1` to node `2` relate to each other by comparing the vectors and caused the current state of the replicated data in node `2` .
- **Version vector** is used to derive the final state of a piece of data in a node without concerning about the sequence of events which caused the final state.

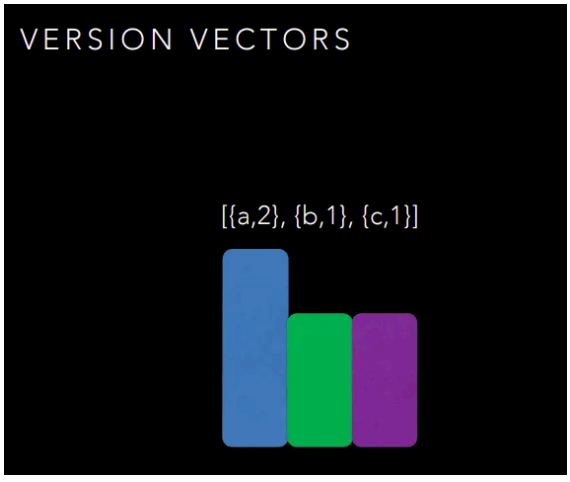
In the above example, four events in sequence: `[[1, 2], [2, 1]] -> [[1, 3], [2, 2]] -> [[1, 4], [2, 5]] -> [[1, 8], [2, 10]]` are recognized by node `2` and the final state of the data in node `2` is determined by the final event in the chain. In the evolution of `[2, 1] -> [2, 2] -> [2, 5] -> [2, 10]` in node `2`, we know that `[2, 2] > [2, 1]`, `[2, 5] > [2, 2]` and `[2, 10]` is greater than all of the predecessors. Knowing `[2, 10]` as the final event is enough to determine the final state of the data in node `2`. Hence if node `2` now wants to pass the concerned data to another node `x`, it can only attach `[2, 10]` to the version info rather than passing the whole chain of predecessors.

So, in short, we can say **a version vector is nothing but a summary of vector clocks, they have similar structure but different semantics.** When a machine `X` comes across a version vector like `[[1, 3], [2, 4]]`, it understands that node `1` has executed three events: `1, 2, 3` whereas node `2` has executed another set of four events: `1, 2, 3, 4` on the given piece of data. So the **counters indeed represent a great summary of events that happened.**

Essentially vector clock and version vector are calculated in the same way, but **version vector is used mostly in a network of replicated systems which just concerns about the final state of the data at some point in time whereas vector clock is useful to derive the partial order among an ever growing pool of events across any node in a distributed system.**

Version Vector Pictorially





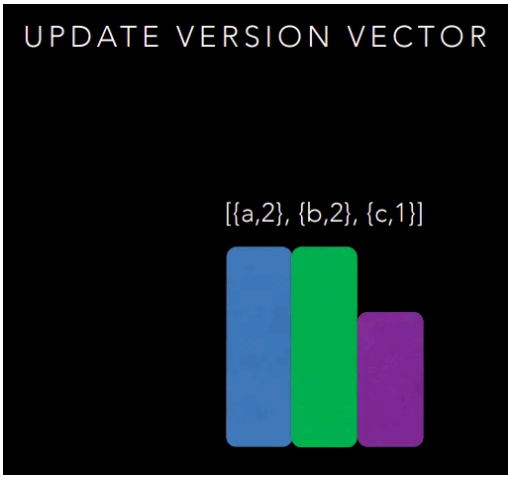


Figure 4: Left: We have a version vector where the initial vector is V1 = [{a, 1}, {b, 1}, {c, 1}], then 'a' gets incremented in the same process, hence the vector becomes: V2 = [{a, 2}, {b, 1}, {c, 1}]. As discussed, we don't need to preserve {a, 1} part any more because V2 descends from V1, the centre figure shows this appropriately. In the right side, the version vector is updated where 'b' gets incremented — note that update operation works as same as vector clock. Courtesy: A Brief History of Time in Riak

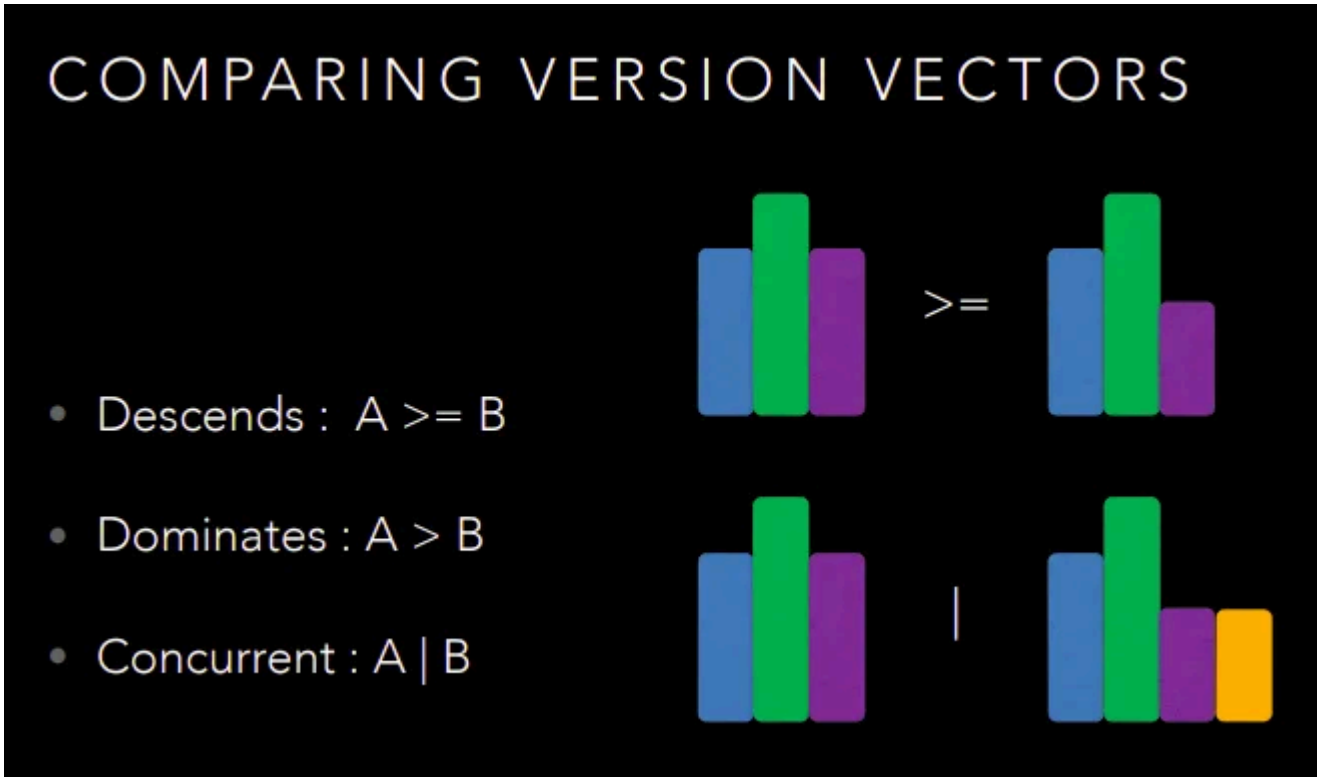


Figure 5: Version Vector comparison, Courtesy: A Brief History of Time in Riak

Version vector also follow the same comparison principle as vector clock. The following definitions apply to vector clock as well.

For two version vectors `A` and `B`, **element by element comparison happens.**

Descending Version Vector: For two version vectors `A` and `B`, if zero or more elements in `A` are **same (equal)** as `B` whereas at least one element is **strictly greater** than `B` or do not exist in `B`, but all elements in `B` exist in `A`, then `B` happened before `A` and `A` **descends** `B`.

Example:

```
If A = [2, 3, 4],    B = [1, 2, 4], then A Descends B since
A[0] > B[0], A[1] > B[1], A[2] = B[2] and all elements in B exist in
A.
```

```
If A = [2, 3, 4, 5], B = [1, 2, 4] then A Descends B since
A[0] > B[0], A[1] > B[1], A[2] = B[2], A[3] does not exist in B and
all elements in B exist in A.
```

Dominating Version Vector: For two version vectors `A` and `B`, if **all the elements** in `A` are **strictly greater** than `B` or some elements in `A` do not exist in `B`, but all the elements in `B` exist in `A`, then `B` happened before `A` and `A` **dominates** `B`.

Example:

```
If A = [2, 3, 4],    B = [1, 1, 2], then A Dominates B since
A[0] > B[0], A[1] > B[1], A[2] > B[2] and all elements in B exist in
A.

If A = [2, 3, 4, 5], B = [1, 2, 1] then A Dominates B since
A[0] > B[0], A[1] > B[1], A[2] > B[2], A[3] does not exist in B and
all the elements in B exist in A.
```

`A Dominates B` is a special use case of `A Descends B`.

Concurrent Version Vector: We have already seen definition of concurrent vector clock, this is same as that.

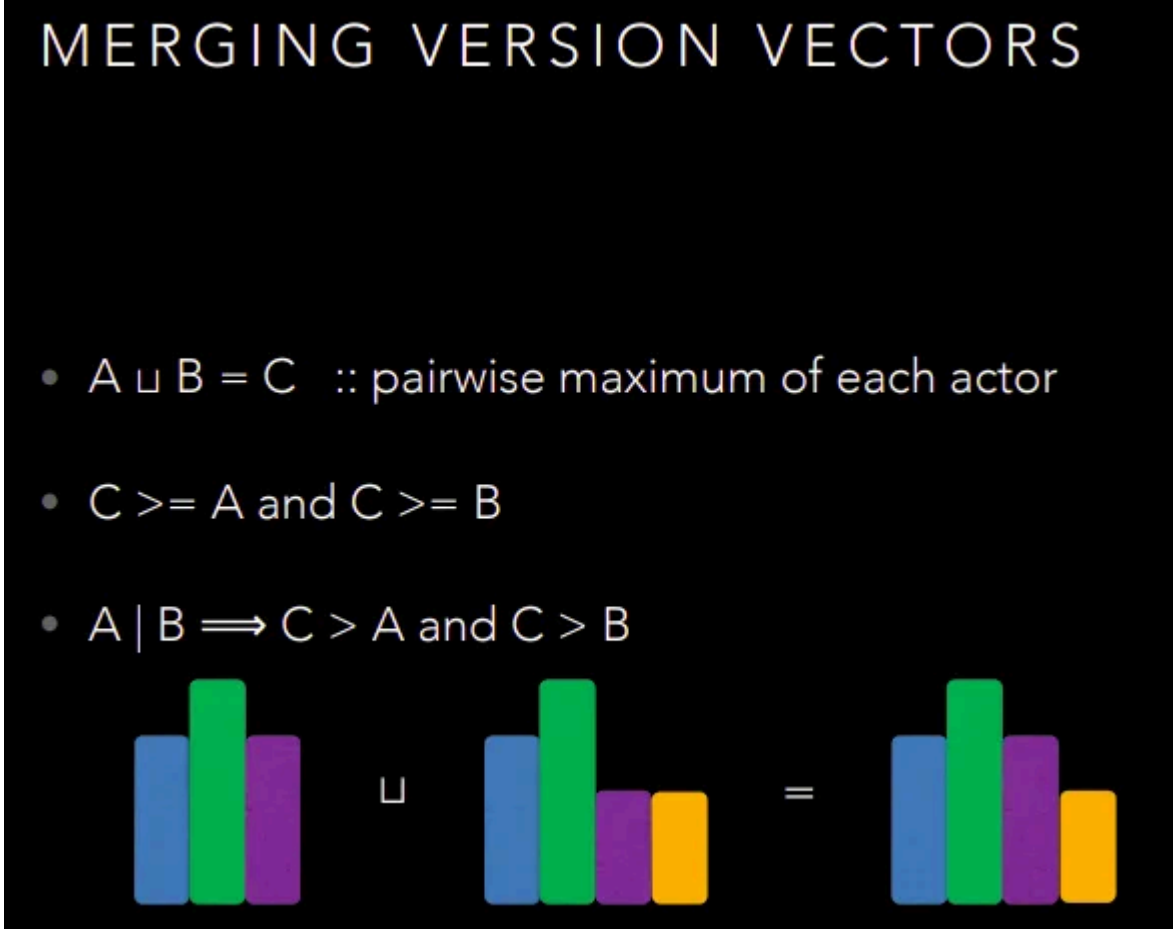


Figure 6: In merging, we take pairwise maximum of elements in the version vectors. In case, some elements are uncommon between two version vectors, they also become part of the union. Hence the union is always greater than or equal to the participating version vectors. In the figure, the yellow part is available only in B, however after union, it becomes part of the union C. Also if two version vectors are concurrent, their union is greater than individual version vectors. Vector clock merging process is same as version vector. Courtesy: A Brief History of Time in Riak

Note: *In our further discussions, we mostly talk about version vectors, however, the operations are more or less same as vector clocks.*

Choosing actor id in Version Vectors

We now know that a vector clock essentially represents a list of `[actor_id, counter]` pair. But where should you generate the id? Is it done in the client side or server side? This is a very important question to analyze with **trade-offs.**

Riak has a very good history of evolving through different variations of version vector. Let's see how it worked out for them, even though a good amount of our discussions ahead are inspired by Riak, the learning, different approaches taken and their trade-offs would be applicable in other real life systems as well.

Any KV storage systems exposes two main functionalities: `PUT` to create, update or even delete key-value pairs and `GET` to fetch them. In order to track updates by clients, KV stores associate every key with a vector clock or version vector or something similar. As we have seen, **each replica of a key could have different version of value objects across nodes and above APIs are stateless in nature meaning clients can talk to any node at any moment**, hence there should be some way to derive which version of data the client has modified. That correlation is called **causal context** and it has to be included in the request and response between the client and the server in every API call.

Causal Context: It's an **immutable** metadata that carries version information for the concerned key in a non-human readable encoded string format. The version could be represented as a vector clock or version vector or in some other suitable form. Causal context looks like this:

```
a85hYGBgzGDKBVIcR4M2cgczH7HPYEpkzGNlsP/VfYYvCwA=
```

While making a `PUT` request, a client has to supply the latest context it has seen in the request header. When the KV store receives the request, it extracts the version information from the client context and **compares the version with the one currently stored in the storage.** This comparison helps discern the store whether the client provided data is more recent, not relevant or concurrent to the existing data.

Client Side Id

Riak used to use client id as actor id in `0.X` versions (The initial Riak versions basically). In this case, each client **uniquely** chooses an id for itself.

- **Get API:** When a client gets the data, Riak passes the associated version information encoded in a context in the response header called **`X-Riak-Vclock`.**

```
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
```

- **Put API:** `PUT` is used to both insert and update data. Let's see both of the use cases.

Insert Request: Clients use `PUT` API to insert new data and passes its unique id through a header called `X-Riak-ClientId`. Clients don't need to pass any context information here since the data is new to the system.

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

In the above request, a client sends some data `"Wednesday"` and passes its client id `"Alice"` in the header identifying itself. Riak identifies the client and accordingly creates the vector clock for the data.

Update Request: Riak, in fact all data store APIs are mostly stateless, clients can talk to any node. Hence if a client wants to modify the data, it has to **pass back the context information that it got in the most recent `GET` API call along with its own unique client id** so Riak can internally compute the version based on the information provided.

```
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"
```

Here, another client Ben wants to modify the data with a new value `Tuesday`, the vector clock received in earlier GET call was:

`a85hYGBgzGDKBVIsrLnh3BlMiYx5rAzLJpw7wpcFAA==` , so Ben passes this vector clock in **`X-Riak-Vclock`** request header, also passes its own actor id in **`X-Riak-ClientId`** request header**.** With these headers, Riak identifies which version of data has been modified by Ben and it **internally creates another version (vector clock) descending from the passed vector clock** which reflects Ben's update.

Now, if another client calls GET API for the same data, the new vector clock would be transmitted by Riak.

Illustration

Riak explains the concept very well in its blogs. Let's use the same example here.

There are four actors in the example:
A => Alice, B => Bob, C => Cathy, D => Dave, they are planning a meeting after long time, they are suggesting suitable time to each other in this example.
All the actors communicate with the system through the APIs discussed above.

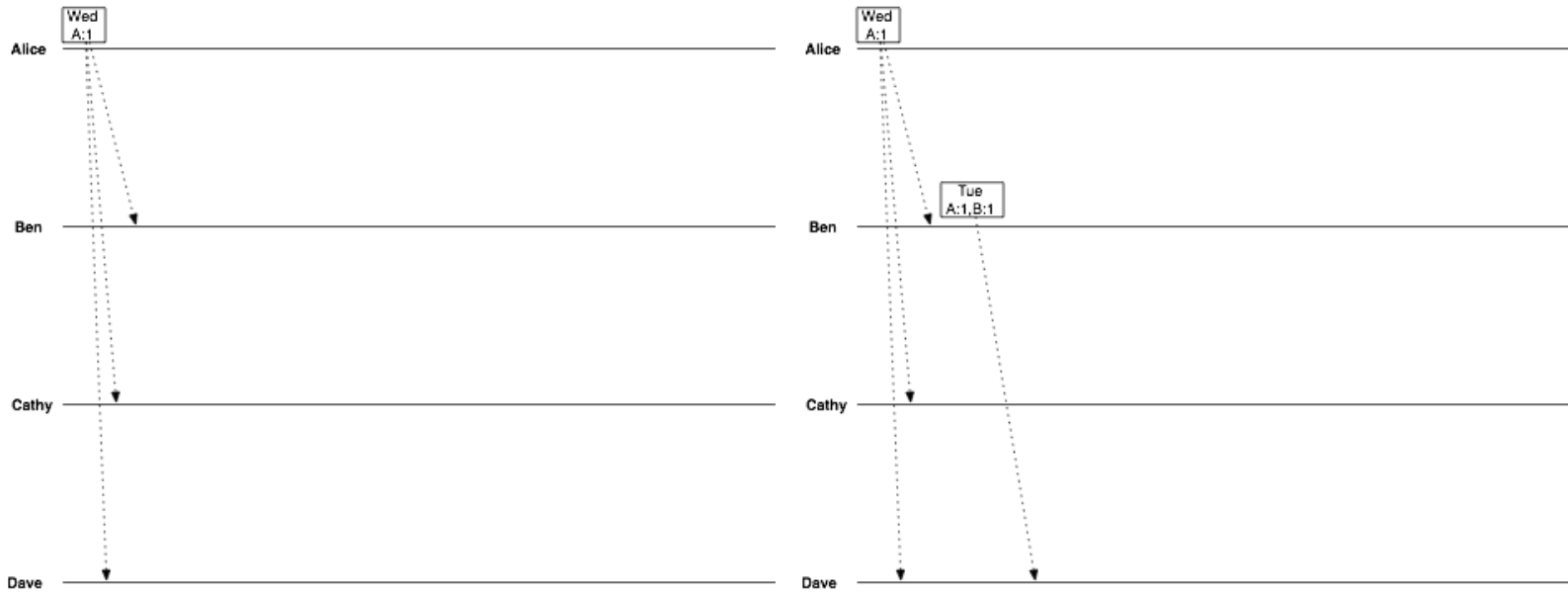


Figure 7

Alice starts the process by suggesting `"Wednesday"`, hence the version of the data is: `[{A, 1}]`. The message is sent to other actors as well.

Ben gets Alice's message and suggests `"Tuesday"`, sends the message across to other actors. Since Ben overrides Alice's message, the new version of the data is `[{A, 1}, {B, 1}]`. Note that Only Dave received the message from Ben, somehow the message did not reach Cathy.

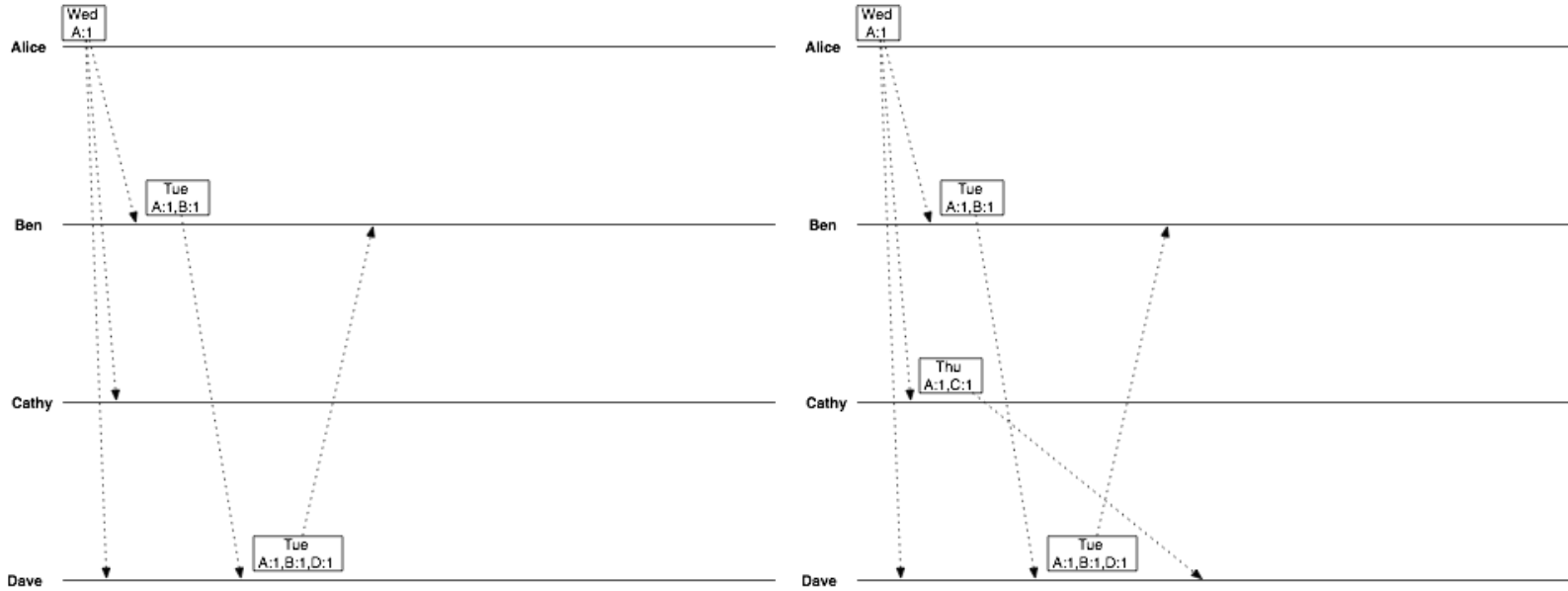


Figure 8

On receiving Ben's message, Dave confirms `"Tuesday"` as well with a new version of the data: `[{A, 1}, {B, 1}, {D, 1}]`, this version descends from Ben's version.

Cathy has only seen Alice's message, her version `[{A, 1}, {C, 1}]` descends from Alice's version. She suggests `"Thursday"` and sends the message across.

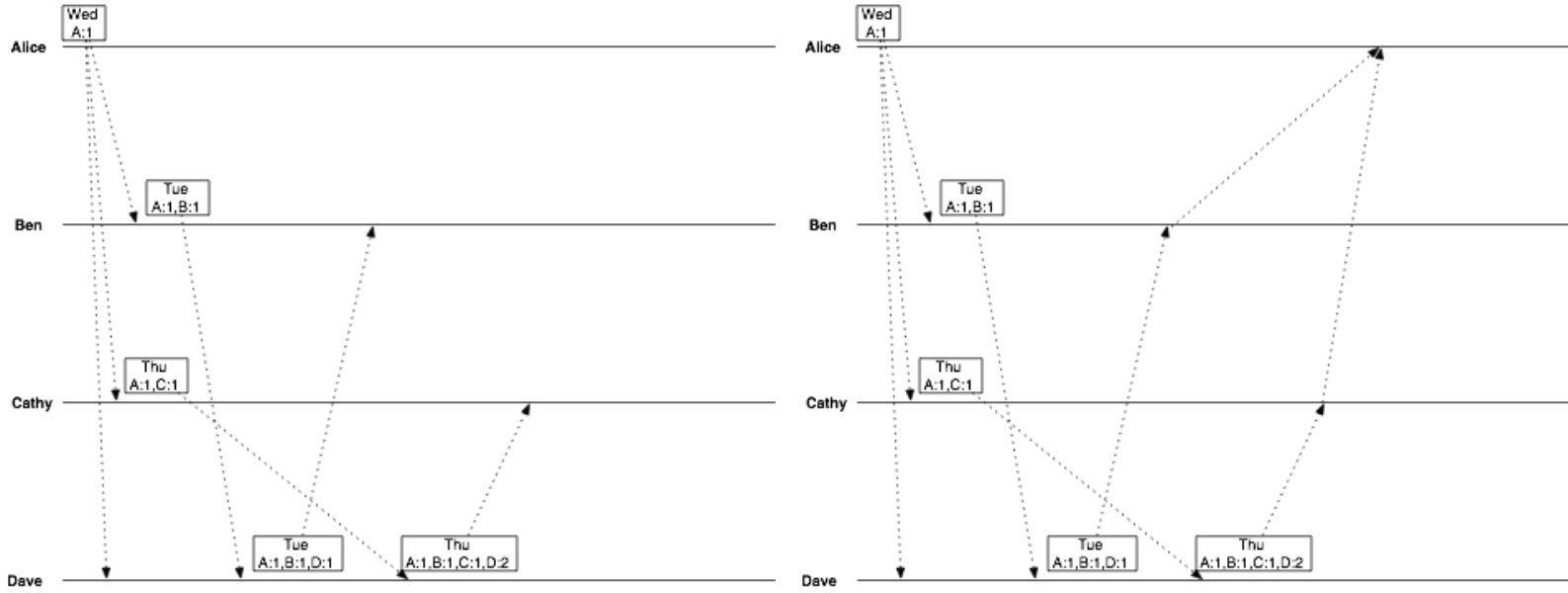


Figure 9

Dave receives Cathy's message, now Dave has a conflict (concurrent vector clock since Cathy and Ben don't know about each others version): he has received a message for the same request earlier from Ben and stored the message with version `[{A, 1}, {B, 1}, {D, 1}]`, now he receives another message for the same request from Cathy, so Dave has to ideally version the new message as `[{A, 1}, {C, 1}, {D, 1}]` . But Dave being a smart guy identifies the conflict and self resolves the conflict, accommodates both Ben and Cathy in his version and sends back the latest seen data `"Thursday"`. Hence the version emitted by Dave is: `[{A, 1}, {B, 1}, {C, 1}, {D, 2}]`.

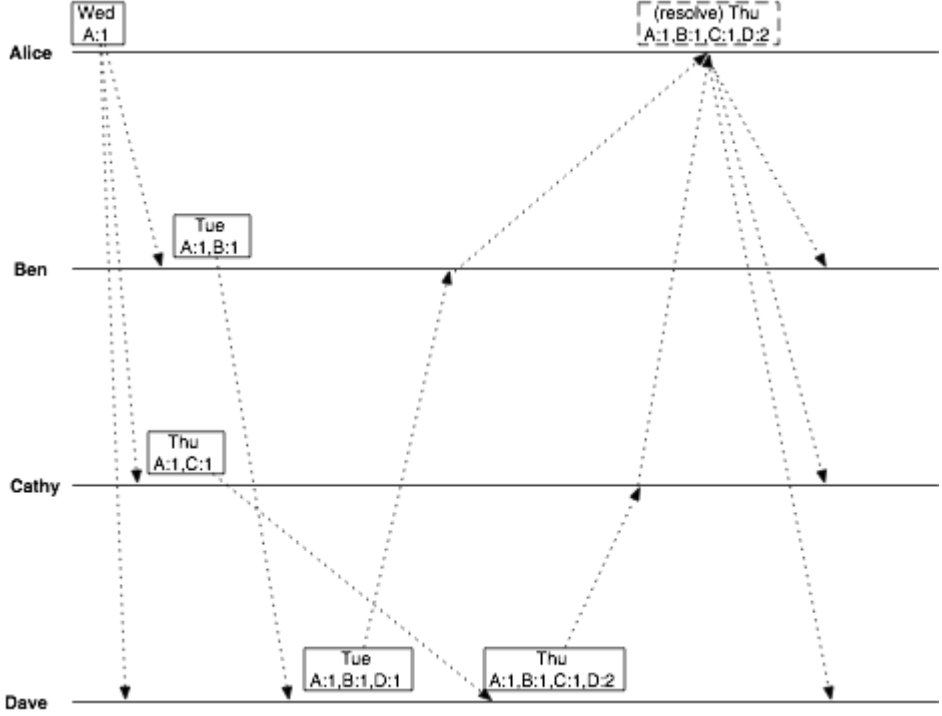


Figure 10

After some time, Alice receives all the messages being transmitted by all other actors, so Alice gets to decide the final meeting date. Since Dave's final version `[{A, 1}, {B, 1}, {C, 1}, {D, 2}]` is greater than all other versions, it dominates all of them, hence final meeting day is confirmed as `"Thursday"`.

Q. What if Dave does not receive Cathy's message with version `[{A, 1}, {C, 1}]`?
A. Then Dave would not have the conflict, rather Alice would have to resolve the conflict when she receives message from all other actors.

Q. How does Riak handle vector clock conflicts?
A. If two different clients end up modifying the same version of a data, conflict happens and two vector clocks which are descending from the same parent but are siblings to each other are created. Riak hands over both the clocks in encoded format through `X-Riak-Vclock` response header and expects the client to resolve the conflict.
If Riak is able to identify one version / clock descends another version / clock, then the older one is automatically deleted.

In Short, the Algorithm with Client Id

- Client Gets Value+Version Vector for Key
- Client Sends Put Request to Riak with Value and ClientID and Version Vector
- Riak increments the entry for ClientID in Version Vector
- At each of N Vnodes:
    - Read local value from disk
    - If local Version Vector descends incoming Version Vector ignore write (you've seen it!)
    - If Incoming Version Vector descends local Version Vector overwrite local value with new one
    - If Incoming Version Vector is concurrent with local Version Vector, add new value as `sibling`

Figure 11

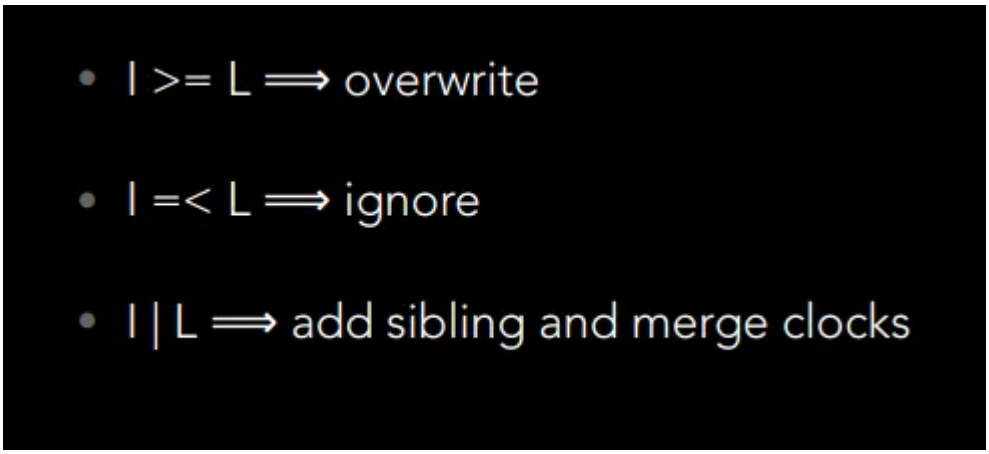


Figure 12: Client Id based easy to remember algorithm. I = Incoming Version Vector, L = vNode's Local Version Vector

A **Sibling** is nothing but a concurrent value which is stored in the system for conflict resolution purpose later since the system might not be good at auto conflict resolution. More on conflict resolution later.

Q. So, how are conflicted writes taken care of according to the above algorithm?

A. Let's consider the following example to understand the conflict:

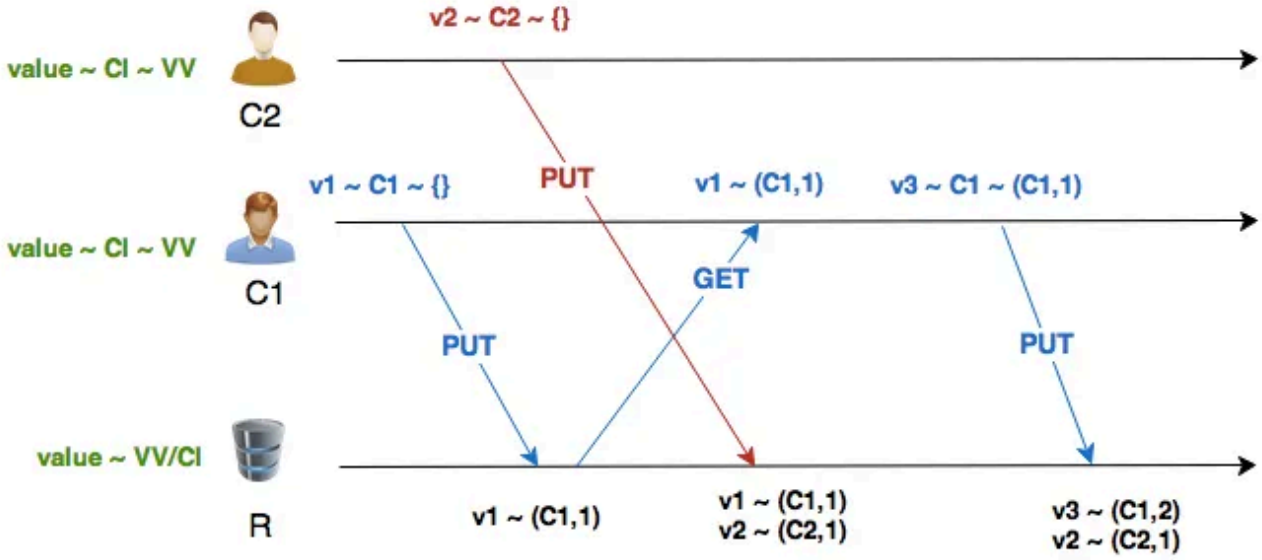


Figure 13, Courtesy: https://github.com/ricardobcl

There are two clients `C1` and `C2` talking to the system `R` (let's say Riak), they are operating on the same key, however the key remains invisible in the above figure as we are just concerned about the values being updated by the clients here.

1. `C1` first puts a value `v1` against the key with empty context `{}` to the system `R`.
2. R has no entry associated with the key, so the local context for the key is also empty `{}`. According to the step 3 in the algorithm in figure 11, `R` first increments the client id counter in the incoming version vector. `{C1, 1}` is logically the incoming version vector which **descends from the local empty one**, `{}`. Hence the new value v1 gets stored against the key and `R` updates its local context to `{C1, 1}`.
3. `C2` being unaware of `C1`'s update, it puts a different value `v2` for the same key with an empty context `{}`.
4. `R` first increments the client id counter for `C2` in the incoming version vector, thus the incoming context is logically `{C2, 1}`. `R`'s **local context `{C1, 1}` neither descends from the incoming context `{C2, 1}` nor the incoming context descends from the local one.** They are **concurrent.** So according to the algorithm in figure 11, `v2` is added as sibling to the

existing value `v1` , `["v1", "v2"]` is stored in `R` and the local context is updated to `{{C1, 1}, {C2, 1}}`.

5. Now, `C1` wants to put a new value `v3`. `C1` got a confirmation of its first update in step 2 with context `{C1, 1}` in response from `R`. Hence in order to update its previous value, `C1` passes back the same context to `R` along with the new value. So `C1` is thinking it's updating the older value `v1` to `v3`.
6. `R` increments the client counter in the incoming context, thus the incoming context becomes `{C1, 2}`, **it does not descend from the local context `{{C1, 1}, {C2, 1}}` neither the local one descends from the incoming one.** However, `R` identifies that the update came from `C1` and while merging, it can replace `C1`'s old value `v1` with the new value `v3`. Thus the sibling value now becomes `["v2", "v3"]` and the system's context is updated to `{{C1, 2}, {C2, 1}}`.

The goodness of client id: In step 6, even though `v3` is added as sibling, the system is able to identify that the update came from the client `C1` and hence it's able to replace the older value `v1` which was earlier placed by `C1`. **They are causally related updates and the identification is possible because of presence of the client ids in the version vectors.** Thus there is no chance that `v3` coexists with `v1` as a false concurrent value.

What we gain with this approach:

- Since client ids are proper unit of concurrency, this approach **appropriately capture causality among the version vectors i.e; no issues in capturing concurrent updates.**

What we lose with this approach:

- **This approach does not scale well.** Since clients are actors here, **actor explosion** (let's say you have `100` nodes in a data centre and each of the nodes are serving `10_000` user requests at some point in time, hence total

request at that moment is `1_000_000` , this is an actor explosion where actors are client or applications using the system, this could become even worse) could cause the width of vector clocks to grow unbounded — if a lot of actors have modified the same data, each of them would have an entry in the version vector. **Also each key stored in each replica would have a version vector associated with it.** It requires more space to store such information, also more CPU cycles to compare the vectors.

- Client application manages id, a buggy code can end up generating new id for each `PUT` call, or generate duplicate id across clients which could potentially cause data loss. So the invariant of maintaining unique id by the clients looks very risky.
- A client has to **read its own write. The correctness of the vector clock algorithm depends on the latest vector clock value that a client fetches from Riak**. Look at the algorithm in figure 11. A client first fetches the latest vector clock information in the `GET` API call, then passes the same clock back to Riak while updating the concerned data. The problem is Riak and many other key value based storage systems prefer eventual consistency for better availability. Hence there is no guarantee that the `GET` API would fetch the latest written data from Riak storage layer depending on which node the request hits. In such cases, if the client fetches older value, the new update might get lost or become a conflicting one.

Server Side (virtual node or vNode) Id

Actor explosion causing more time and space consumption looks like the biggest con of client side id approach.

Can we do better? Can we have bounded width of our vectors somehow?

What options do we have at our hand?
Option 1: Client id is out of question, due to actor explosion it's not a scalable solution.
Option 2: Server id: We can have thousands of servers geographically

distributed or even hundred of servers just in the same data centre. This option is better than using client id however, if our system allows the same data to be stored across all these servers, that solution would not scale much since adding more servers potentially increases our vector clock / version vector size.
Option 3: Replica id: Replicas are servers only (in other words, a bunch of replicated data is hosted by a server). A piece of data (key value pair) resides only in few designated replicas. Hence the data can be modified only in those replicas thus restricting the entry size in the version vector only to maximum number of replicas configured. A default typical replica size is `3` in many cloud computing systems however usually replicas are kept very small in number.

So, it's better to use replica id in our vector clock since it's bounded in size.

Note:
- Since replicas are servers only, to make things generic, we call them server in our further discussions, but remember we just mean replica.
- Many key value storage systems use a consistent hash ring of nodes called **virtual node (vNode)** which are internally mapped to physical servers. A set of vNode works as a replica for a given set of data. vNode id could also be used instead of replica id depending on implementation.

Riak used vNode vector clocks on `1.x` releases. Let's consider the exact same example that we just saw with client id but this time with server (replica) id.

There are two servers `x` and `y`. Alice and Dave are connected to `x`, whereas Ben and Cathy are connected to `y`. For brevity, assume there are two dedicated threads running in `x` serving Alice and Dave, same happens in server `y` for Ben and Cathy.

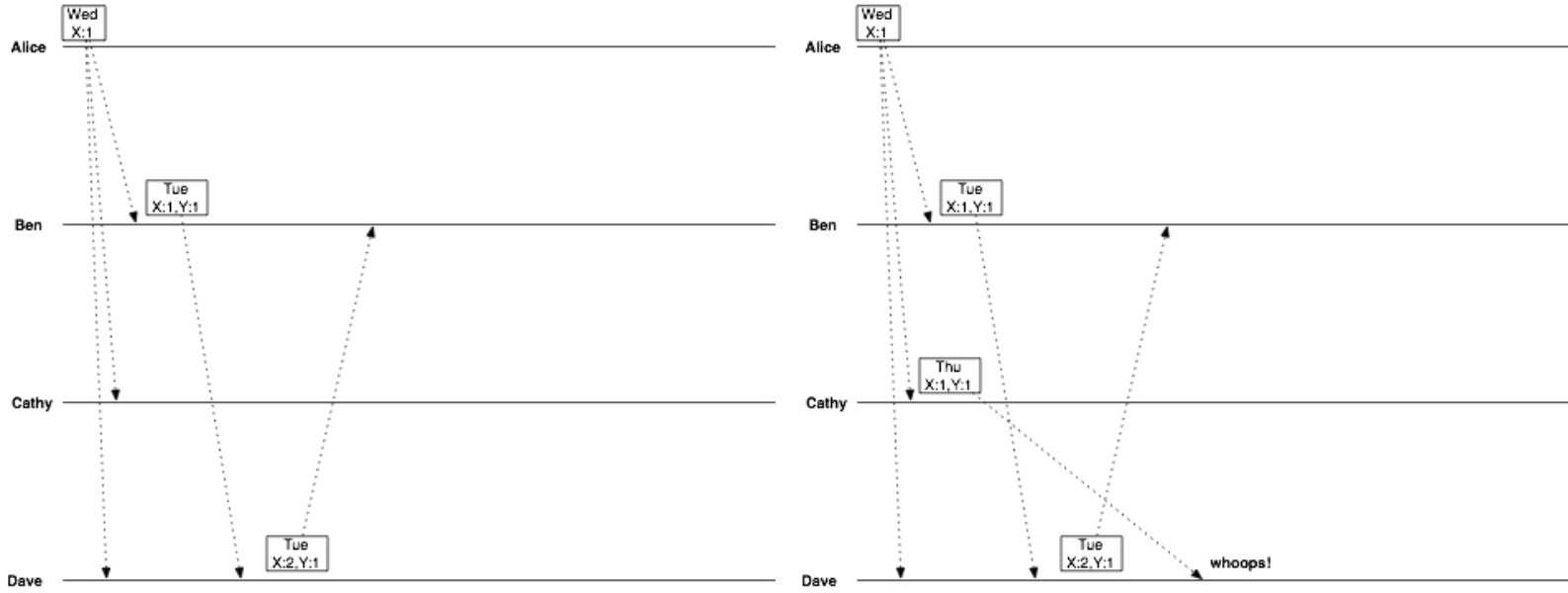


Figure 14

Alice suggests `"Wednesday"`, server `X` gets the request, hence the version is `[{X, 1}]`. The message gets replicated to server `Y` internally.

Ben suggests `"Tuesday"`, server `Y` gets the request, since `Y` is aware of Alice's message, the new version becomes `[{X, 1}, {Y, 1}]`.

Dave is now aware of Ben's message and suggests "Tuesday". Dave's message is served by `X`, hence the new version of the data is `[{X, 2}, {Y, 1}]`.

Till now all is well.

What if Cathy suggests something?

Cathy is **concurrent** with Ben, while Ben is in the process of suggesting suitable date, Cathy has already read Alice's message in server `Y`. Hence Cathy works on Alice's message, suggests "Thursday" with a version `[{X, 1}, {Y, 1}]`. The message is replicated to `X` however there is some lag in replication and Cathy's message reaches `X` after Ben's message reach there.

When Dave receives Cathy's update, Dave does not perform any action on the data since Cathy's data version is `[{X, 1}, {Y, 1}]` whereas Dave has already

seen version `[{X, 2}, {Y, 1}]` from Ben for the same request earlier which is higher than Cathy's version. Hence Dave ignores Cathy's update making the **update silently lost.**

Let's consider a classic example below where **multiple clients update the same key concurrently in the same server:**

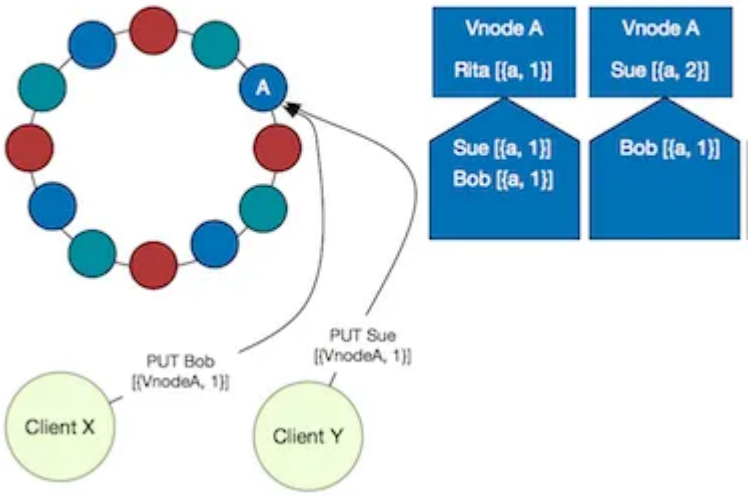


Figure 15, Courtesy: Riak docs

- There are two client `X` and `Y` who are talking to the same virtual node `A`. Both have read the same data "Rita" with version `[{a, 1}]` from `A`.
- Both of them intend to update the data however they don't know they are **concurrent.**
- Client `X` makes a `PUT` request with data "Bob" and passes around the latest version vector seen: `[{a, 1}]`.
- Similarly `Y` makes another concurrent `PUT` request with data "Sue" and latest seen version vector: `[{a, 1}]`.
- Client `Y`'s request is handled by `A` first. `A` observes that the client's version vector ***descends*** it local version vector associated with the data (both incoming and local contexts are same). Hence it increments the local version vector to `[{a, 2}]` and accepts the write request.

- While serving client `X`'s request, `A` observes that `X`'s version vector (the incoming version vector) `[{a, 1}]` ***does not descend*** its local version vector `[{a, 2}]` i.e; the local version vector is already higher than what `X` has provided. Hence `A` ignores the write request and `X`'s update is **lost silently**.

Q. No system is supposed to lose data deliberately, how to work around this problem of silently losing updates?
A. Riak solves this problem by adding the new data as **sibling (concurrent)** when the **local version vector descends the incoming one.** In this way, Riak does not lose the data silently. By storing siblings, we create an opportunity for **sibling explosion.** This is nothing but creating **false concurrency** and repair the data later.

Q. What happens if some client passes older version vector in `PUT` request?
A. With the above trick, the data would be added as a sibling with the existing data.

Q. What if a system does not support storing siblings?
A. If any system does not store siblings, in such a scenario, it makes peace with losing data silently by either ignoring the new data or serializing the concurrent updates thus overwriting the previous data with the new one — this policy is called ***Last Write Win (LWW)***. More on `LWW` later.

Algorithm with server side id

1. If Incoming *descends* Local
    1. Increment Vnode's entry in *incoming* Version Vector
    2. Write new value to disk
2. Otherwise
    1. *Merge* Incoming and Local Version Vectors
    2. Increment Vnode's entry in *merged* Version Vector
    3. Store Incoming value as a *sibling*

Figure 16, Courtesy: Riak docs

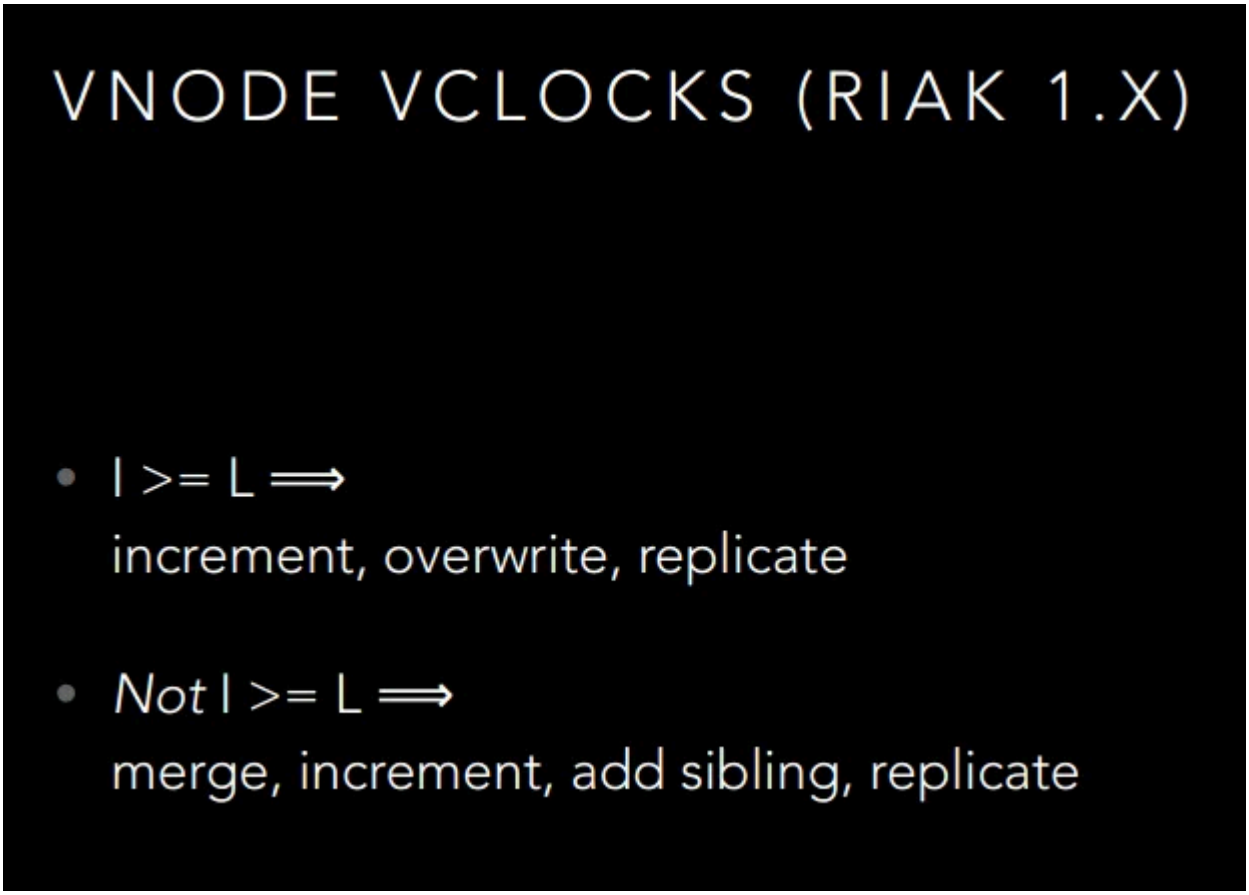


Figure 17: Server Id based easy to remember algorithm. I = Incoming Version Vector, L = vNode's Local Version Vector, Courtesy: A Brief History of Time in Riak

Let's consider the same example as in figure 17 to understand how we can afford not losing data using siblings:

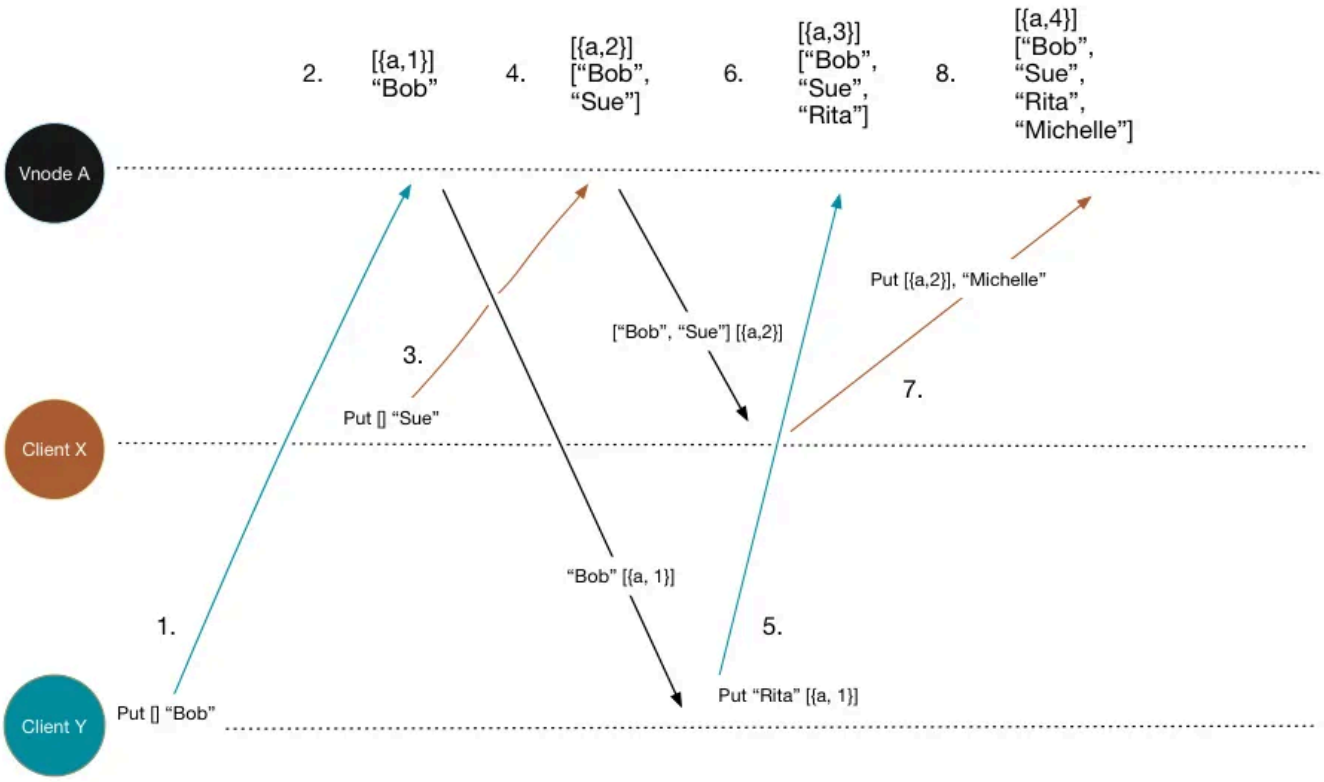


Figure 18, Courtesy: Riak docs

Client `X` and `Y` are talking to a Riak vNode `A` and they want to concurrently update the same key with some values.

1. Client `Y` makes a request to update the value to `"Bob"`. Since it's the first request by `Y`, it passes an empty context `[]` in the `PUT` API call.
2. The system does not have the key stored yet, hence its local context is also empty (`[]`). According to the algorithm in figure 18, **the incoming context (`[]`) descends the local context (`[]`)**, so the counter at the vNode `A` increments for the incoming request and the data `["Bob"]` is stored with causal context (version vector) `[{a, 1}]`. The updated data `["Bob"]` and the context `[{a, 1}]` is passed back in response to `Y`.
3. Client `X` now wants to put `"Sue"` for the same key. `X` is **unaware** of `Y`'s update. `X` thinks it's the first client to put the key value pair. Hence it passes empty context `[]` along with value `"Sue"` to the system.
4. At this point the system knows that it already has context `[{a, 1}]` stored for the same key. According to the algorithm in figure 18, the system identifies that **incoming request context `[]` does not descend local**

context `[{a, 1}]`, hence it treats the incoming request as **concurrent.** It increments the local version vector to `[{a, 2}]`, stores the incoming value as **sibling** to the existing value. The new value `["Bob", "Sue"]` along with context `[{a, 2}]` is passed back in response to `X`.

5. Client `Y` is **unaware** of `X`'s update, `Y` is only aware of its own update that it made in step 1. It now wants to update the value to `"Rita"`. So, from `Y`'s perspective, its updating the previous value `"Bob"` to `"Rita"`. It passes the context `[{a, 1}]` along with the new value in the `PUT` request to the system.

6. The system is aware of all updates happening for the key at vNode `A`. It identifies that the **local context `[{a, 2}]` does not descend the incoming context [{a, 1}], the request is concurrent.** Hence it increments the local version vector to `[{a, 3}]`, stores the value `"Rita"` as a sibling, passes the context `[{a, 3}]` and the new value `["Bob", "Sue", "Rita"]` to `Y` in the API response.

7. The cycle goes on, `X` wants to update the previous value `"Sue"` to `"Michelle"`, it's aware of the last update that it made in step 3. Hence `X` passes the context `[{a, 2}]` along with the new value to the system.

8. The system identifies it's concurrent, increments the local version vector to `[{a, 4}]`, adds the new value as sibling to the existing value, passes back the new context `[{a, 4}]` and the updated value `["Bob", "Sue", "Rita", "Michelle"]` to `X` in response.

As you can observe, in the approach we are not losing concurrent data as they are stored as siblings. However there is some problem here. Can you identify?

The worse of server id based approach: In step 1, when `Y` makes a `PUT` request, the system stores `"Bob"` with context `[{a, 1}]`. In step 3, when `X` makes a `PUT` request with `"Sue"`, the system assumes they are **concurrent updates** as described above and it **creates a new causal context `[{a, 2}]`**, adds `"Sue"` as sibling to `"Bob"`. At this point, **the system completely forgets that `"Bob"` came from an older causal context `[{a, 1}]`, there is no**

metadata tracking that information when the new context `[{a, 2}]` is created. The system can only see that the context `[{a, 2}]` holds value `["Bob", "Sue"]`.
At step 5, when the client `Y` actually wants to **update the previous value** `"Bob"` (sent in step 1) to `"Rita"`, the system fails to identify `Y`'s intention, it has now no clue that `"Bob"` was earlier stored in the system with context `[{a, 1}]` and the logical right thing is to update `"Bob"` to `"Rita"`. Instead, it observes the current context is `[{a, 2}]` and adds the new value as sibling to `["Bob", "Sue"]` thus making both `"Bob"` and `"Rita"` **false concurrent updates.**

So **this approach is not able to appropriately track causality across updates.**

What we gain with this approach:

- No actor explosion since number of servers (replicas) is limited. Hence smaller vectors and this approach **scales better then client id based one.**
- No more mandatory read your own write by a client since passing around encoded version vector is optional, hence simpler client implementation. For better behaviour and lesser conflicts, it's good to still provide the latest version vector that the client has seen.

What we lose with this approach:

- As we just saw, there is an edge case of potentially losing data if siblings are not stored since **server or replica id works as a proxy for multiple clients and is not the real unit of concurrency.**
- If siblings are stored, **sibling explosion** could happen. At peak scale, **a node could even explode** with high number of siblings.
- Increasing siblings could cause **performance issue** in a node since a lot of siblings have to be read or written or reconciled for correctness.
- This approach uses a **single version vector to represent the merged siblings,** hence it **does not offer proper causality tracking** and no good

support for identification of precise conflicting values.

Q. Using client side id causes much larger vectors and using server side id causes sibling explosion — both of them increase space consumption. How do we tackle that?

Vector Clock Pruning

We can prune version vectors / vector clocks based on some heuristics:
- **Discard based on time:** Prune version vectors which were created or modified before some threshold ex: `150 ms` (just an example) from now. How to decide this threshold depends on scale and probably business use cases as well.
- **Discard based on maximum number of entries:** Start pruning version vector when the number of entries is between `20` to `50`. A version vector with `size ≥ 50` must be pruned.

These are just some examples.

In order to implement these ideas, we need to store physical timestamp along with version vectors when they are created or modified. **This timestamp is not going to be used for vector comparison**, the sole purpose is to delete older than threshold vectors.

Pruning Issues: Version vector pruning could be **unsafe** as you might end up deleting legitimate entries which would produce different result for version vector comparison for the same data before and after pruning, thus potentially changing the value for a key or even losing valid data. You may end up in **false concurrency**.

Is there any solution which is bounded in size thus scales well, appropriately tracks causality without losing data and doesn't need aggressive pruning?

Dotted Version Vector (DVV)

Riak `2.X` releases use more modern version vectors called DVV.

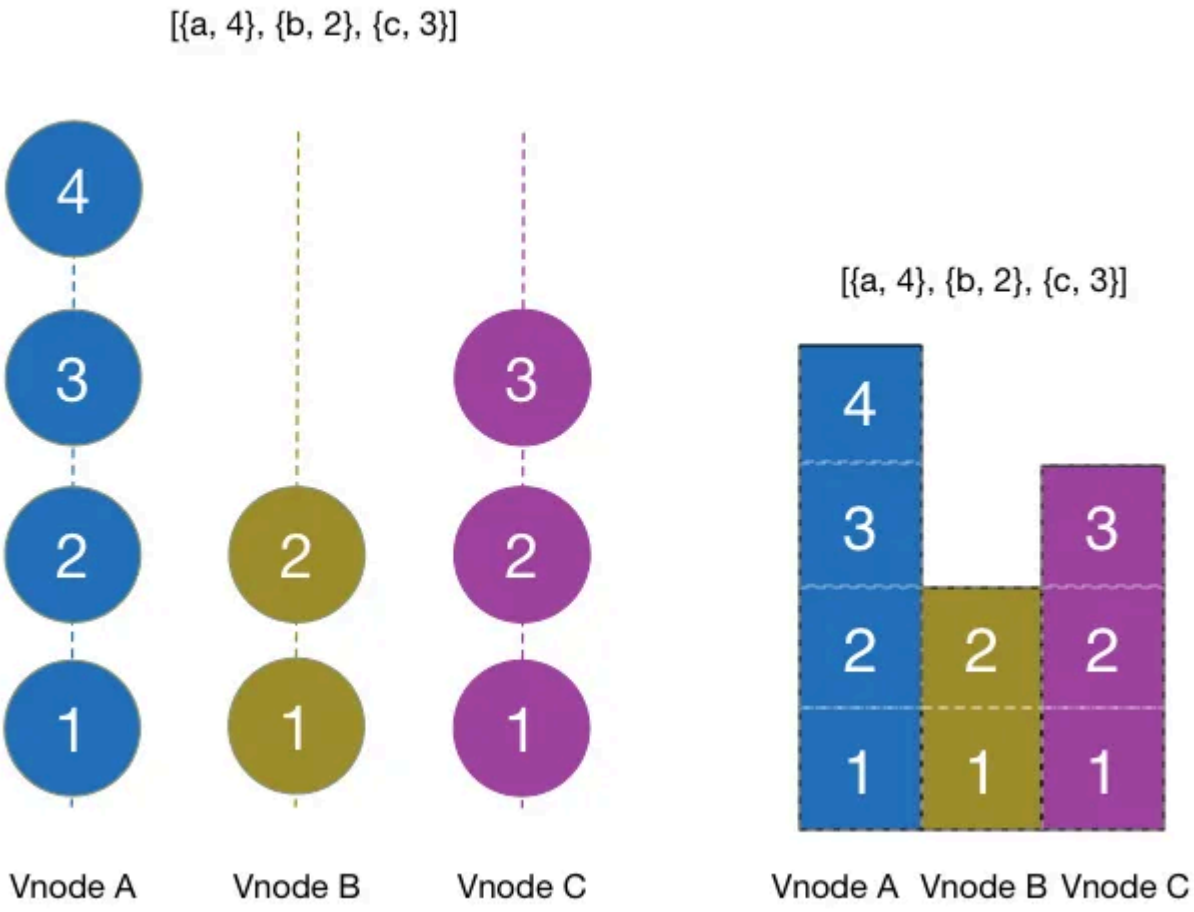


Figure 19, Courtesy: Riak docs

Consider the above figure. It shows a version vector `[{a, 4}, {b, 2}, {c, 3}]` associated with some data. It means the vNode A executed `4` events, vNode B executed `2` events and vNode C executed `3` events on the data. **Version vector is nothing but a set of these discrete events.** With vNode or server id based version vector, the context represents the full set of siblings. Like `[{a, 4}, {b, 2}, {c, 3}]` may be mapped to a set of siblings like `["Bob", "Sue", "Rita"]`, however, we don't know the origin version of any of the siblings i.e; observing this context, we don't get any idea whether `"Bob"` originated from `{a, 1}` or `{c, 3}` and so on. That's why the sibling list keeps on growing.

But, what if we instead of storing a set of siblings, we store the context also along with them? I mean what if we store `(id, counter)` mapping along with the sibling something like: `{a, 1} => "Bob"` or `{b, 4} => "Sue"` etc so that next time when a request arrives with causal context `{a, 1}` in the header, we easily identify that requests wants to update `"Bob"` and take appropriate action. This `(id, counter)` pair like `{a, 3}` indicates the event that happened when a's counter is `3`, it does not associate to any other previous counter like `{a, 1}` or `{a, 2}`. This `(id, counter)` pair is called a **dot**, it represents an event that happened at that exact moment.

So, **we not only store the overall version vector for a key, but also the most recent siblings along with appropriate contexts.**

When we receive a request, we can compare each entry of the incoming version vector with each of these dots to identify whether the incoming version vector descends the dots, if yes, then the client is already aware of the earlier events and we can update / replace that old events. This way, we don't need to store unnecessary faulty concurrent siblings.

Consider the same example as figure 18 below but with DVV this time:

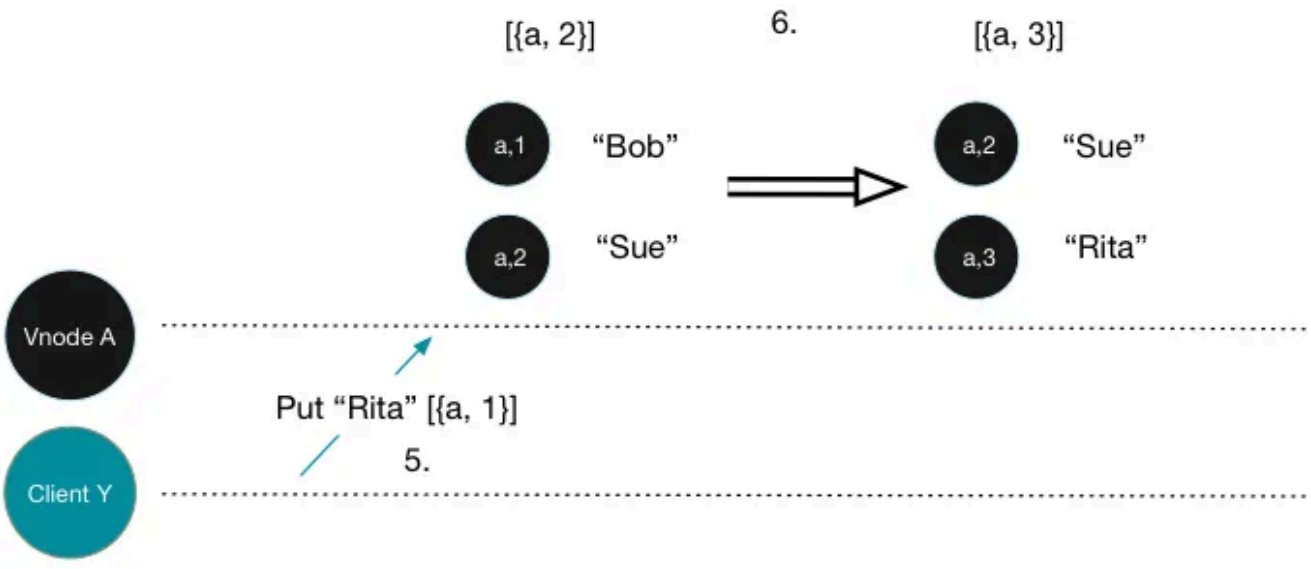


Figure 20, Courtesy: Riak docs

- When client `Y` updates `"Bob"`, in DVV, we store the dot along with the value e.g; `[{a, 1}] => "Bob"` (this notation does not necessarily mean a

mapping, it's just for better understanding) so that we have a track of the source version of the data.

- Similarly, when client `X` updates `"Sue"`, we store the dot `[{a, 2}] => "Sue"`. Client `Y` did not mean to update `X`'s update since anyway the request came in with an empty context `[]`, thus **they are valid concurrent updates (siblings).**
- In figure 18, in step 5, when `Y` sent an update request with context `[{a, 1}]` and value `"Rita"` to update its previous value `"Bob"`, the system could not understand `Y`'s intention and ended up storing `"Rita"` as a false sibling to `["Bob", "Sue"]`. However, this time, we know that the system has a dot `[{a, 1}]` already stored with `"Bob"`. Hence when the system receives `Y`'s request, reading the incoming context, it observes that the incoming context `[{a, 1}]` descends the dot saved with `"Bob"`. Hence it removes the older dot `[{a, 1}]`, replaces `"Bob"` with `"Rita"`. The current dot `[{a, 3}] => "Rita"` is stored as well. While saving `"Rita"`, `"Sue"` is untouched since it's associated with a different dot and it continues to be a legitimate sibling.
- In a similar way, other updates continue.

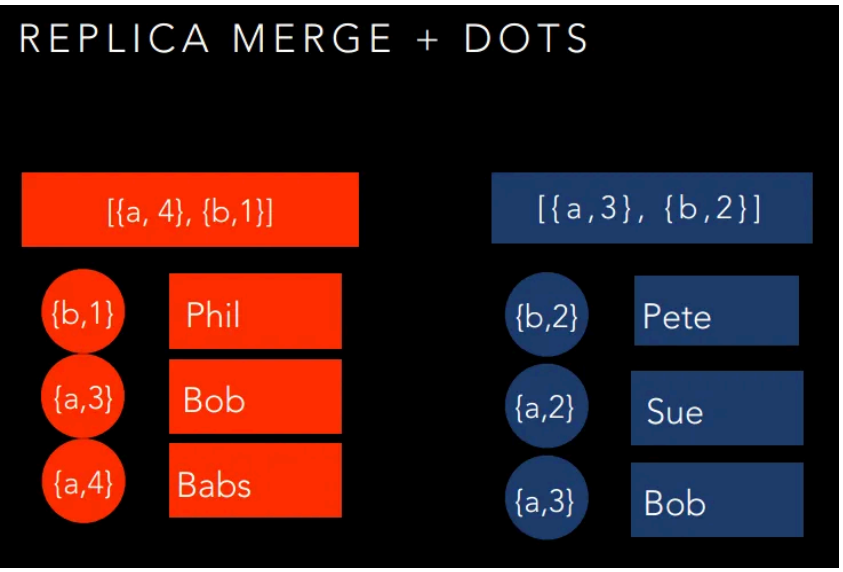
In Short, the DVV algorithm

Figure 21, Courtesy: A Brief History of Time in Riak

Replica Merging With DVV

When two replicas have divergent values for the same key, how does the conflict resolution take place there?

In this case, both the replicas have their own dots stored. While resolving the conflict, we can take the pairwise maximum of the dots, if any of the dots exists in only one replica, the one currently without the dot can store that.



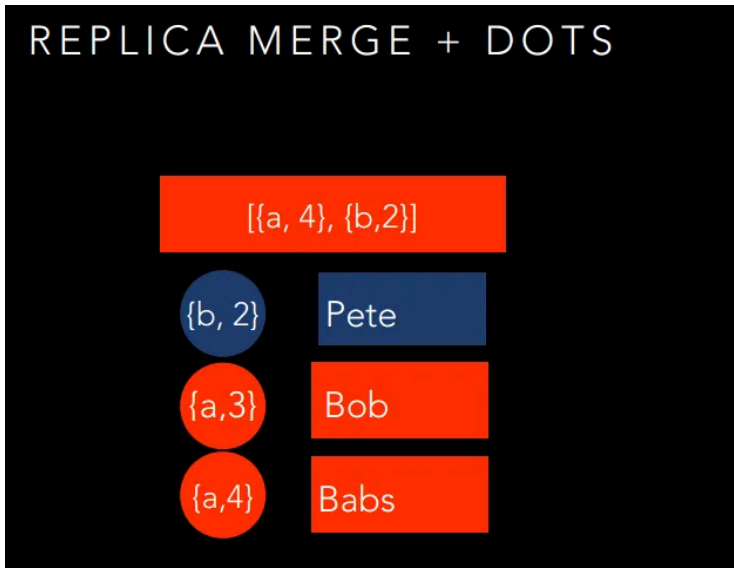


Figure 22, Courtesy: A Brief History of Time in Riak

In the above example, we are trying to merge two contexts: `[{a, 4}, {b, 1}]` and `[{a, 4}, {b, 2}]` from two different replicas. In the pairwise dots comparison, we observe that:

- `{a, 4}` descends `{a, 3}`, hence `"Babs"` gets precedence than `"Bob"`.
- `{a, 3}` descends `{a, 2}` hence `"Bob"` gets precedence than `"Sue"`.
- Similarly `{b, 2}` descends `{b, 1}`, hence `"Pete"` gets precedence than `"Phil"`.

Thus the final context becomes `[{a, 4}, {b, 2}]` with dots: `{a, 4}, {a, 3}, {b, 2}`.

The above is just an easy example, possibly it get more difficult with more edge cases when a real system is developed.

Advantages of DVV

- DVV helps to identify valid conflict among versions. The biggest gain is **no sibling explosion with appropriate causality tracking** which helps to quickly resolve conflicts at peak hours.
- The most recent and accurate data can be retrieved efficiently. Thus it improves customer experience and potentially provides better user experience.
- No pruning is required since the maximum **number of dots at some point is bounded by the maximum number of replica nodes** configured in the system.

Disadvantages of DVV

- In the replication process, when a node locally executes the updates, stores the dots and then replicates to other replicas, the overall data

transfer size could be high since dots also need to be replicated.

DVV Set

There is one more implementation of Dotted Version Vector which is more space efficient and achieves the same goals. For now, we are not going to discuss that solution here as it's more complex. You can find it here.

Conflict Resolution Techniques in Version Vector

Whatever variation of version vector we use, **conflict at scale is unavoidable**. Hence such systems offer us conflict resolution mechanisms which can be done both at the server and client side.

Just to recap: whether a conflict for a key happens in a single node causing siblings or conflicts happens across different nodes, **all the conflicting versions are kept in the system until they are resolved**.

Server Side Resolution

Last Write Wins (LWW) Strategy: Every data object can be associated with a timestamp indicating the last modification time. If the system is configured to support `LWW`, depending on the timestamp, concurrent updates (siblings) are deleted barring the latest update. This could lead to dropping arbitrary data, create **inconsistent and unexpected behaviour** across replicas. In the presence of partition, if two clients write two different versions of the same data whose parent versions are same to different replicas, the system discards the older write even though it's a valid competitor creating data race condition.

Data inaccuracy becomes a big problem with this approach. Hence many real life applications might not find `LWW` strategy useful.

In short, **LWW is an unsafe strategy for immutable data where a client can update the data after reading it from the system**.

Q. Are there any suitable use cases which can tolerate `LWW` strategy?

A. There are certain use cases which are good fit for `LWW`:

- If the system scale is very low and you don't expect much concurrent writes to happen.
- Your business use cases are fine with the implications of inconsistent data removal from replicas.
- If your data is **immutable**, then anyway you are not allowed to update it meaning all writes are stored as different versions of the same data. Hence `LWW` is a safe strategy there.

Q. Huh! everything has a good side as well. What are the merits of LWW?
A. The biggest merit: `LWW` is a simple to understand strategy for customers since they don't need to care about conflict resolution or siblings etc. The server automatically take care of it.

Read Repair: It's a mechanism where **version vector conflict resolution kicks in at the time of reading data**. The node coordinating the read request fetches possibly different versions of the data from suitable replicas which contain the data, then it identifies the conflict, tries to merge the version vectors. In case there are conflicting versions, they are presented to the client.

Since read repair happens only for the data currently being read, it's a **CPU friendly process**, however, it can repair only those data which are read by the clients. Any data which have not been read for long time, remain unaffected and **cold**.

Proactive Repairing / Active Anti Entropy (AAE): It's also a read repair kind of mechanism however instead of coming into action only at the read time, it runs in the background continuously and tries to resolve conflicts across all replicas. **It's more suitable for cold data.**

Client Side Resolution

Callback Mechanism: Generally the systems relying on logical time give the liberty of conflict resolution to the client applications since at times it's very difficult for the system to decide the most recent value and resolving on `LWW` or some other automated strategy might result into erroneous result.

These systems provide client libraries where callbacks can be implemented. Applications can provide their own **use case specific conflict resolution strategy in those callbacks** so that when the updated data is sent to the system, appropriate version information is sent along.

Minimize Conflicts and Siblings

When a client provides a **stale** causal context in `PUT` request, the system gets confused and stores that data as a sibling. To mitigate such issues, one approach is to update data in a `read-modify-write` cycle.

- Call `GET` API for the key. The client receives the latest value and version vector associated.
- Modify the value locally.
- Call `PUT` API to update the key and pass back the latest version vector as the context.

The above steps ensures that the client has done its best to minimize conflict and possible siblings.

How Amazon Dynamo Paper describes usage of Vector Clock

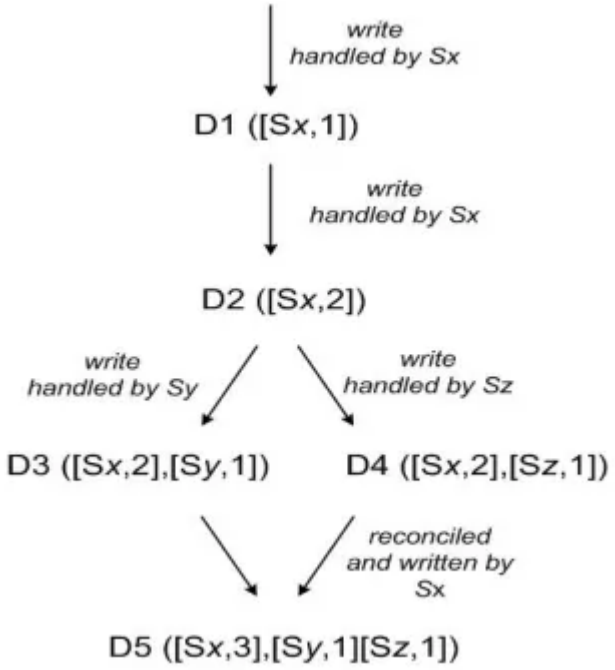


Figure 23: Amazon Dynamo vector clock usage, Courtesy: Dynamo paper

Dynamo paper describes version management using vector clocks with server side id which is susceptible to already mentioned issues earlier. The paper is pretty old, it's not clear whether AWS Dynamo DB uses vector clock or something else now.

Let's analyze figure 23 to understand the scenario better:

With every interaction (request and response) a context encoding version information is shared between the client and the server.

- A client writes some data (a key value pair) to Dynamo cluster and one of the nodes, *Sx* handles the request. *Sx* increases its counter and the version (vector clock) becomes *D1* = `([Sx, 1])`.
- The node performs another update on the data and the same node *Sx* handles the request. Let's call this Hence the version is incremented at the node to *D2* = `([Sx, 2])`. Here we say *D2* descends from *D1* and *D1* gets overridden at *Sx*.
- The client updates the same data again, but this time, the write request is being handled by another node *Sy*. *Sy* is seeing the data for the first time,

so it increments the counter to `1`. Hence the version of the data at *Sy* becomes *D3* = `([Sx, 2], [Sy, 1])`. *D3* descends *D2*.

- Now another client reads D2 from Sx, updates the data and the write request is handled by another node called *Sz*. Since *Sz* is also seeing the data for the first time, it increments its counter to `1`. The version of the data at *Sz* becomes *D4* = `([Sx, 2], [Sz, 1])`. *D4* descends *D2* as well.
- Note that though *D3* at *Sy* and *D4* at *Sz* branch out of *D2* at *Sx*, *D3* and *D4* are unaware of each other i.e; *Sy* and *Sz* nodes don't even know that for the same key, the values being held by them could be potentially different. So *D3* and *D4* are not causally consistent from each other — they just can't be related to each other since they have mismatch in their version information. These events are concurrent in nature.
- A nodes which is aware of *D1, D2* and *D3* or *D1*, *D2*, and *D4* can derive the causal relation that *D3* or *D4* is derived from *D2*. So *D1* and *D2* can be purged or garbage collected.
- If any node which is aware of *D3*, receives *D4* from a client, the node can not resolve the conflict since it identifies that *D3* and *D4* are unrelated. So the client has to solve the conflict and update the data. Similarly, if a client reads both *D3* and *D4* together, the context will reflect presence of both the version, hence the conflict has to be resolved by the client and resolved data has to be written back to Dynamo. Let's say this write request is again handled by *Sx*, so *Sx* increments its own counter to `3` and a new version *D5* = `([Sx, 3], [Sy, 1], [Sz, 1])` is created.

For the Curious: Implementation Of Vector Clock in Voldemort

Following implementation is taken from Voldemort, is an open source key value distributed storage system much like Amazon Dynamo DB for a better understanding of Vector clock.

```
class VectorClockTest {
    @Test
    public void testDeserializationBackwardCompatibility() {
        assertEquals("The empty clock serializes incorrectly.",
                     getClock(),
                     new VectorClock(getClock().toBytes()));
        VectorClock clock = getClock(1, 1, 2, 3, 4, 4, 6);
        // Old Vector Clock would serialize to this:
        // 0 5; 1; 0 1, 2; 0 2, 1; 0 3, 1; 0 4, 2; 0 6, 1; [timestamp=random]
        byte[] knownSerialized = { 0, 5, 1, 0, 1, 2, 0, 2, 1, 0, 3, 1, 0, 4, 2, 0, 6, 1, 0, 0, 1
                0x3e, 0x7b, (byte) 0x8c, (byte) 0x9d, 0x19 };
        assertEquals("vector clock does not deserialize correctly on given byte array",
                     clock,
                     new VectorClock(knownSerialized));

        DataInputStream ds = new DataInputStream(new ByteArrayInputStream(knownSerialized));
        VectorClock clock2 = VectorClock.createVectorClock(ds);
        assertEquals("vector clock does not deserialize correctly on given input stream",
                     clock,
                     clock2);
    }
}

class VectorClock {
    /**
     * Takes the bytes of a VectorClock and creates a java object from them. For
     * efficiency reasons the extra bytes can be attached to the end of the byte
     * array that are not related to the VectorClock
     *
     * @param bytes The serialized bytes of the VectorClock
     */
    public VectorClock(byte[] bytes) {
        this(bytes, 0);
    }

    /**
     * Read the vector clock from the given bytes starting from a particular
     * offset
     *
     * @param bytes The bytes to read from
     * @param offset The offset to start reading from
     */
    public VectorClock(byte[] bytes, int offset) {
        if(bytes == null || bytes.length <= offset)
            throw new IllegalArgumentException("Invalid byte array for serialization--no bytes t
        int numEntries = ByteUtils.readShort(bytes, offset); // First 2 bytes from the given off
        // how many entries are there in the vector clock hash map
        int versionSize = bytes[offset + 2]; // Next 1 byte represents the size of each version
        // being represented in the given stream.
        int entrySize = ByteUtils.SIZE_OF_SHORT + versionSize; // size of each node id + size of
        int minimumBytes = offset + ByteUtils.SIZE_OF_SHORT + 1 + numEntries * entrySize
```

```
52                              + ByteUtils.SIZE_OF_LONG;
53
54          if(bytes.length < minimumBytes)
55              throw new IllegalArgumentException("Too few bytes: expected at least " + minimumByte
56                                                 + " but found only " + bytes.length + ".");
57
58          this.versionMap = new TreeMap<Short, Long>();
59          int index = 3 + offset;
60          for(int i = 0; i < numEntries; i++) {
61              short nodeId = ByteUtils.readShort(bytes, index);
62              long version = ByteUtils.readBytes(bytes, index + ByteUtils.SIZE_OF_SHORT, versionSi
63              this.versionMap.put(nodeId, version);
64              index += entrySize;
65          }
66          this.timestamp = ByteUtils.readLong(bytes, index);
67      }
68
69      public static VectorClock createVectorClock(DataInputStream inputStream) {
70          try {
71              final int HEADER_LENGTH = ByteUtils.SIZE_OF_SHORT + ByteUtils.SIZE_OF_BYTE; // 3
72              byte[] header = new byte[HEADER_LENGTH]; // length = 3
73              inputStream.readFully(header); // read 3 bytes starting from the last read position
74              int numEntries = ByteUtils.readShort(header, 0); // read first 2 bytes from the heac
75              // total number of entries in the vector clock hash map
76
77              byte versionSize = header[ByteUtils.SIZE_OF_SHORT]; // read the last byte of the hea
78              // required to represent each version. Volemort uses Long (8 bytes) to represent a v
79              // of version can be from 1 to Long.MAX_VALUE. Hence maximum value of `versionSize`
80              // So `versionSize` is represented by a byte type here.
81
82              int entrySize = ByteUtils.SIZE_OF_SHORT + versionSize; // node id size ( maximum num
83              // cluster is Short.MAX_VALUE) + version size per node
84              int totalEntrySize = numEntries * entrySize; // total entry size
85
86              byte[] vectorClockBytes = new byte[HEADER_LENGTH + totalEntrySize
87                                                 + ByteUtils.SIZE_OF_LONG];
88              // no of entries (2 bytes) + version size (1 byte) + all entries
89              // ( pair of node id and version size numEntries times ) + The most recent timestamp
90              // when the operation happened (8 bytes)
91              System.arraycopy(header, 0, vectorClockBytes, 0, header.length);
92
93              inputStream.readFully(vectorClockBytes, HEADER_LENGTH, vectorClockBytes.length
94                                                                     - HEADER_LENGTH);
95
96              return new VectorClock(vectorClockBytes);
97          } catch(IOException e) {
98              throw new IllegalArgumentException("Can't deserialize vectorclock from stream", e);
99          }
100     }
101
102     public VectorClock merge(VectorClock clock) {
```

```
103             VectorClock newClock = new VectorClock();
104             for(Map.Entry<Short, Long> entry: this.versionMap.entrySet()) {
105                 newClock.versionMap.put(entry.getKey(), entry.getValue());
106             }
107             for(Map.Entry<Short, Long> entry: clock.versionMap.entrySet()) {
108                 Long version = newClock.versionMap.get(entry.getKey());
109                 if(version == null) {
110                     newClock.versionMap.put(entry.getKey(), entry.getValue());
111                 } else {
112                     newClock.versionMap.put(entry.getKey(), Math.max(version, entry.getValue()));
113                 }
114             }
115 
116             return newClock;
117         }
118 
119         @Override
120         public Occurred compare(Version v) {
121             if(!(v instanceof VectorClock))
122                 throw new IllegalArgumentException("Cannot compare Versions of different types.");
123 
124             return VectorClockUtils.compare(this, (VectorClock) v);
125         }
126 
127     }
```

VectorClock.java hosted with ❤ by GitHub view raw

Voldemort uses a hash map internally to store vector clock in memory. Voldemort server receives client inputs like `put` or `get` request along with version information as a byte stream. Hence vector clock is encoded to or decoded from a byte array representation at times in the code base.

```
Voldemort Vector Clock Encoding
================================

Internal vector clock hash map stores one on one mapping of Node Id -
> Version. That hash map data is encoded in a byte array in the
following format:

-------------------------------------------------------------------
| 2 bytes | 1 byte | consecutive [node_id,version] pairs | 8 bytes |
-------------------------------------------------------------------
```

Basically the byte array needs to store some consecutive `[node_id, version]` pairs. In order to enable that, we need to know how many such nodes are

there, how we can represent the version information. Hence some extra metadata we need to store in the byte array.

Line 3–21: It's an unit test which tests the Voldemort vector clock implementation in some scenario. We don't much need to concern about the test, however, take at look at line 10–11 to understand how such a byte array looks like.

Line 46: **numEntries:** The first `2` bytes representing how many entries are there in the given vector clock. Maximum value for **numEntries** is nothing but the number of nodes in the cluster. Java's **Short** data type is used to represent this field limiting maximum number of nodes in the cluster to **Short.MAX_VALUE**.

Line 48: **versionSize:** The next `1` byte representing the size of data required to represent version information in the byte array encoding. Voldemort stores version number in the internal hash map as **Long** type data (`8` bytes). Hence we need maximum `8` bytes of space to represent a version. The minimum space required is `1` byte. Hence the range of the variable **versionSize** is from `1` to `8` and a single byte is enough to represent such information.

Line 50: **entrySize:** An entry is a pair of node id and version number. We need `2` bytes to represent a node id and **versionSize** bytes of data to represent the associated version. Hence `entrySize = 2 + versionSize` bytes. Clearly, lesser the **versionSize**, more efficient the memory footprint of the byte array is.

Line 51–52:
numEntries * entrySize: Total size of all entries together.
Last `8` bytes in the byte array is reserved for the timestamp at which the associated `get` or `put` operation happened.
So total size of the byte array is: `2` bytes to store **numEntries** + `1` byte to store version information + **numEntries * entrySize** bytes for the actual

vector clock pairs + `8` bytes for timestamp. Also an extra **offset** is added because the byte array may be prefixed with extra data information which a client passes to the server, but we can ignore this part for now.

Line 60–65: Consecutive entries are read one by one and put into the vector clock hash map.
Line 61: Read node id of size `2` bytes.
Line 62: Read the next **versionSize** amount of bytes into `version` corresponding to the above node id.
Line 63: Put the node id and version pair to the vector clock hash map.

Line 69–100: This method takes some input stream of data and converts it to a byte array encoding of vector clock, then it sends the data to the vector clock constructor just what we discussed above.

Line 102–117: Vector clock merging happens here. Merging means creating a new vector clock by taking union of entries from two given vector clocks. If there is an element which is common across them, then take the maximum value. If an entry exists in one clock, but does not exist in another, simply put it to the merged vector clock.

Following is the code for vector clock comparison and conflict resolution:

```java
/*
 * Copyright 2008-2013 LinkedIn, Inc
 *
 * Licensed under the Apache License, Version 2.0 (the "License"); you may not
 * use this file except in compliance with the License. You may obtain a copy of
 * the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS, WITHOUT
 * WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
 * License for the specific language governing permissions and limitations under
 * the License.
 */
package voldemort.versioning;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.Set;
import java.util.SortedSet;

import com.google.common.collect.Sets;

public class VectorClockUtils {

    /**
     * Compare two VectorClocks, the outcomes will be one of the following: <br>
     * -- Clock 1 is BEFORE clock 2, if there exists an nodeId such that
     * c1(nodeId) <= c2(nodeId) and there does not exist another nodeId such
     * that c1(nodeId) > c2(nodeId). <br>
     * -- Clock 1 is CONCURRENT to clock 2 if there exists an nodeId, nodeId2
     * such that c1(nodeId) < c2(nodeId) and c1(nodeId2) > c2(nodeId2)<br>
     * -- Clock 1 is AFTER clock 2 otherwise
     *
     * @param v1 The first VectorClock
     * @param v2 The second VectorClock
     */
    public static Occurred compare(VectorClock v1, VectorClock v2) {
        if(v1 == null || v2 == null)
            throw new IllegalArgumentException("Can't compare null vector clocks!");
        // We do two checks: v1 <= v2 and v2 <= v1 if both are true then
        boolean v1Bigger = false;
        boolean v2Bigger = false;

        SortedSet<Short> v1Nodes = v1.getVersionMap().navigableKeySet();
        SortedSet<Short> v2Nodes = v2.getVersionMap().navigableKeySet();
        // get clocks(nodeIds) that both v1 and v2 has
        SortedSet<Short> commonNodes = Sets.newTreeSet(v1Nodes);
        commonNodes.retainAll(v2Nodes);
```

```
52          // if v1 has more nodes than common nodes
53          // v1 has clocks that v2 does not
54          if(v1Nodes.size() > commonNodes.size()) {
55              v1Bigger = true;
56          }
57          // if v2 has more nodes than common nodes
58          // v2 has clocks that v1 does not
59          if(v2Nodes.size() > commonNodes.size()) {
60              v2Bigger = true;
61          }
62          // compare the common parts
63          for(Short nodeId: commonNodes) {
64              // no need to compare more
65              if(v1Bigger && v2Bigger) {
66                  break;
67              }
68              long v1Version = v1.getVersionMap().get(nodeId);
69              long v2Version = v2.getVersionMap().get(nodeId);
70              if(v1Version > v2Version) {
71                  v1Bigger = true;
72              } else if(v1Version < v2Version) {
73                  v2Bigger = true;
74              }
75          }
76
77          /*
78           * This is the case where they are equal. Consciously return BEFORE, so
79           * that the we would throw back an ObsoleteVersionException for online
80           * writes with the same clock.
81           */
82          if(!v1Bigger && !v2Bigger)
83              return Occurred.BEFORE;
84          /* This is the case where v1 is a successor clock to v2 */
85          else if(v1Bigger && !v2Bigger)
86              return Occurred.AFTER;
87          /* This is the case where v2 is a successor clock to v1 */
88          else if(!v1Bigger && v2Bigger)
89              return Occurred.BEFORE;
90          /* This is the case where both clocks are parallel to one another */
91          else
92              return Occurred.CONCURRENTLY;
93      }
94
95      /**
96       * Given a set of versions, constructs a resolved list of versions based on
97       * the compare function above
98       *
99       * @param values
100      * @return list of values after resolution
101      */
102     public static List<Versioned<byte[]>> resolveVersions(List<Versioned<byte[]>> values) {
```

```
103         List<Versioned<byte[]>> resolvedVersions = new ArrayList<Versioned<byte[]>>(values.size(
104         // Go over all the values and determine whether the version is
105         // acceptable
106         for(Versioned<byte[]> value: values) {
107             Iterator<Versioned<byte[]>> iter = resolvedVersions.iterator();
108             boolean obsolete = false;
109             // Compare the current version with a set of accepted versions
110             while(iter.hasNext()) {
111                 Versioned<byte[]> curr = iter.next();
112                 Occurred occurred = value.getVersion().compare(curr.getVersion());
113                 if(occurred == Occurred.BEFORE) {
114                     obsolete = true;
115                     break;
116                 } else if(occurred == Occurred.AFTER) {
117                     iter.remove();
118                 }
119             }
120             if(!obsolete) {
121                 // else update the set of accepted versions
122                 resolvedVersions.add(value);
123             }
124         }
125 
126         return resolvedVersions;
127     }
128 
129     /**
130      * Generates a vector clock with the provided values
131      *
132      * @param serverIds servers in the clock
133      * @param clockValue value of the clock for each server entry
134      * @param timestamp ts value to be set for the clock
135      * @return
136      */
137     public static VectorClock makeClock(Set<Integer> serverIds, long clockValue, long timestamp)
138         List<ClockEntry> clockEntries = new ArrayList<ClockEntry>(serverIds.size());
139         for(Integer serverId: serverIds) {
140             clockEntries.add(new ClockEntry(serverId.shortValue(), clockValue));
141         }
142         return new VectorClock(clockEntries, timestamp);
143     }
144 
145     /**
146      * Generates a vector clock with the provided nodes and current time stamp
147      * This clock can be used to overwrite the existing value avoiding obsolete
148      * version exceptions in most cases, except If the existing Vector Clock was
149      * generated in custom way. (i.e. existing vector clock does not use
150      * milliseconds)
151      *
152      * @param serverIds servers in the clock
153      */
```

```
154     public static VectorClock makeClockWithCurrentTime(Set<Integer> serverIds) {
155         return makeClock(serverIds, System.currentTimeMillis(), System.currentTimeMillis());
156     }
157
158 }
```

VectorClockUtils.java hosted with ❤ by GitHub view raw

The above code self explanatory and it's well documented as well. As we already discussed some examples earlier, a vector clock can be a predecessor, a successor or parallel to another vector clock. Voldemort defines an enum called `Occurred` and represents these relationships as `Occurred.BEFORE`, `Occurred.AFTER` and `Occurred.CONCURRENT` respectively.

The gist of the above code is:

- Two vector clocks can have zero or more common nodes. If a vector clock has extra nodes than the common nodes, the clock diverges, it's said to be bigger than the other one since the other clock does not contain these extra nodes. Line 44–61 in the above snippet determines whether any clock is bigger. Note that both the vector clocks can have extra nodes which are not common to each other — in that case, the clock diverges from each other. The vector clocks are **concurrent** in this scenario.
- If two vector clocks have exactly same nodes, then versions of corresponding nodes are compared to determine whether any clock **descends** from another one or both of them are divergent thus **concurrent**. Line 63–75 does this job.
- Line 82–92 compares the divergence and determines the final result. Line 82 looks like a special case, we can ignore that part.

Conclusion

I hope, with all the detailed approaches, pros and cons described in this article, it'll help us to get a better understanding of what happens behind eventually consistent systems, how are data versions managed, how their resolution happens. **The same techniques can be used in distributed caching and messaging systems, distributed file storage systems as well.** We

also analyzed code snippet from Voldemort's vector clock which gives us an idea about how things are implemented in real systems.

All the learning from this article are very crucial for designing real distributed, highly scalable and concurrent systems. Hope you liked it and you'll get a chance to work on such kind of systems at some point in time in your career.

Please give **multiple claps** and share it on social media like Twitter, LinkedIn with broader audience to help them.

Reference

1. http://people.cs.aau.dk/~bnielsen/DS-E08/material/clock.pdf
2. https://www.quora.com/How-can-you-explain-partial-order-and-total-order-in-simple-terms
3. https://8thlight.com/blog/rylan-dirksen/2013/10/04/synchronization-in-a-distributed-system.html
4. https://lamport.azurewebsites.net/pubs/time-clocks.pdf
5. https://levelup.gitconnected.com/distributed-systems-physical-logical-and-vector-clocks-7ca989f5f780
6. https://haslab.wordpress.com/2011/07/08/version-vectors-are-not-vector-clocks/
7. https://stackoverflow.com/questions/58544442/what-are-the-use-cases-for-a-vector-clock-versus-a-version-vector
8. Version vector related images: https://speakerdeck.com/seancribbs/a-brief-history-of-time-in-riak
9. https://riak.com/why-vector-clocks-are-easy/
10. https://riak.com/posts/technical/why-vector-clocks-are-hard/
11. https://riak.com/posts/technical/vector-clocks-revisited/index.html?p=9545.html