Vector Clocks and Conflicting Data

Let’s learn how Dynamo uses vector clocks to keep track of data history and reconcile divergent histories at read time.

We'll cover the following

As described in the previous lesson (https://www.educative.io/collection/page/5668639101419520/5559029852536832/4935298205614080), sloppy quorum means multiple conflicting values against the same key can exist in the system and must be resolved somehow. Let’s understand how this can happen.

What is clock skew?#

On a single machine, all we need to know about is the absolute or **wall clock** time: suppose we perform a write to key `k` with timestamp `t1` and then perform another write to `k` with timestamp `t2`. Since `t2 > t1`, the second write must have been newer than the first write, and therefore the database can safely overwrite the original value.

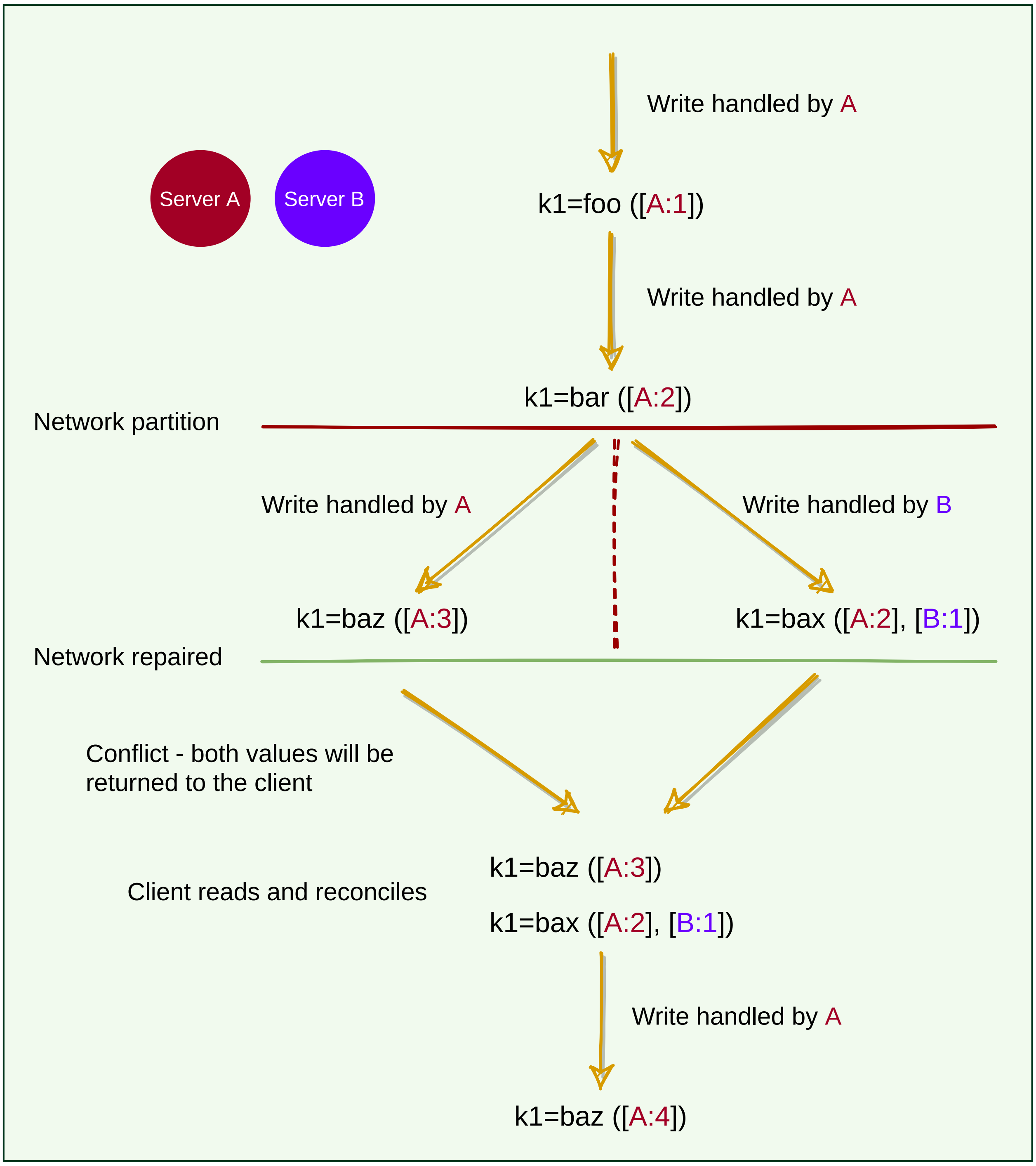
In a distributed system, this assumption does not hold. The problem is **clock skew**, i.e., different clocks tend to run at different rates, so we cannot assume that time `t` on node `a` happened before time `t + 1` on node `b`. The most practical techniques that help with synchronizing clocks, like NTP, still do not guarantee that every clock in a distributed system is synchronized at all times. So, without special hardware like GPS units and atomic clocks, just using wall clock timestamps is not enough.

What is a vector clock?#

Instead of employing tight synchronization mechanics, Dynamo uses something called **vector clock** in order to **capture causality between different versions of the same object**. A vector clock is effectively a **`(node, counter)`** pair. One vector clock is associated with every version of every object stored in Dynamo. One can determine whether two versions of an object are on parallel branches or have a causal ordering by examining their vector clocks. If the counters on the first object's clock are less-than-or-equal to all of the nodes in the second clock, then the first is an ancestor of the second and can be forgotten. Otherwise, the two changes are considered to be in conflict and require reconciliation. Dynamo resolves these conflicts at read-time. Let's understands this with an example:

1. Server `A` serves a write to key `k1`, with value `foo`. It assigns it a version of `[A:1]`. This write gets replicated to server `B`.
2. Server `A` serves a write to key `k1`, with value `bar`. It assigns it a version of `[A:2]`. This write also gets replicated to server `B`.
3. A network partition occurs. `A` and `B` cannot talk to each other.
4. Server `A` serves a write to key `k1`, with value `baz`. It assigns it a version of `[A:3]`. It cannot replicate it to server `B`, but it gets stored in a hinted handoff buffer on another server.
5. Server `B` sees a write to key `k1`, with value `bax`. It assigns it a version of `[B:1]`. It cannot replicate it to server `A`, but it gets stored in a hinted handoff buffer on another server.
6. The network heals. Server `A` and `B` can talk to each other again.

7. Either server gets a read request for key `k1`. It sees the same key with different versions `[A:3]` and `[A:2][B:1]`, but it does not know which one is newer. It returns both and tells the client to figure out the version and write the newer version back into the system.



Conflict resolution using vector clocks

As we saw in the above example, most of the time, new versions subsume the previous version(s), and the system itself can determine the correct version (e.g., `[A:2]` is newer than `[A:1]`). However, version branching may happen in the presence of failures combined with concurrent updates, resulting in conflicting versions of an object. In these cases, the system cannot reconcile the multiple versions of the same object, and the client must perform the reconciliation to collapse multiple branches of data evolution back into one (*this process is called semantic reconciliation*). A typical example of a collapse operation is "merging" different versions of a customer's shopping cart. Using this reconciliation mechanism, an add operation (i.e., adding an item to the cart) is never lost. However, deleted items can resurface.

Resolving conflicts is similar to how Git works. If Git can merge different versions into one, merging is done automatically. If not, the client (i.e., the developer) has to reconcile conflicts manually.

Dynamo truncates vector clocks (*oldest first*) when they grow too large. If Dynamo ends up deleting older vector clocks that are required to reconcile an object's state, Dynamo would not be able to achieve eventual consistency. Dynamo's authors note that this is a potential problem but do not specify how this may be addressed. They do mention that this problem has not yet surfaced in any of their production systems.

Conflict-free replicated data types (CRDTs)#

A more straightforward way to handle conflicts is through the use of **CRDTs**. To make use of CRDTs, we need to model our data in such a way that concurrent changes can be applied to the data in any order and will produce the same end result. This way, the system does not need to worry about any ordering guarantees. Amazon's shopping cart is an excellent example of CRDT. When a user adds two items (A & B) to the cart, these two operations of adding A & B can be done on any node and with any order, as the end result is the two items are added to the cart. (*Removing from the shopping cart is modeled as a negative add.*) The idea that any two nodes that have received the same set of updates will see the same end result is called **strong eventual consistency**. Riak has a few built-in CRDTs (https://docs.riak.com/riak/kv/2.2.0/developing/data-types/).

Last-write-wins (LWW)#

Unfortunately, it is not easy to model the data as CRDTs. In many cases, it involves too much effort. Therefore, vector clocks with client-side resolution are considered good enough.

Instead of vector clocks, Dynamo also offers ways to resolve the conflicts automatically on the server-side. Dynamo (*and Apache Cassandra*) often uses a simple conflict resolution policy: **last-write-wins (LWW)**, based on the wall-clock timestamp. LWW can easily end up losing data. For example, if two conflicting writes happen simultaneously, it is equivalent to flipping a coin on which write to throw away.

← Back
Replication

Next →
The Life of Dynamo's put() & get() Op...

Mark as Completed

Report an Issue