Table of Contents

An introduction to state-based CRDTs

Here we want to cover a topic of Conflict-free Replicated Data Types: what problems they aim to solve and provide some basic implementations (in F#) to help you understand how to build them.

A motivation

Conflict-free Replicated Data Types are answer for a common problem of synchronizing data in distributed environments. While issues on that field where well-known and there were numerous attempts to solve it in the past, usually they were a variants of decentralized 2-phase commit transactions. However they all suffer for similar problems:

- We are assuming, that all participants are available for the time of the transaction, which often (i.e. in case of edge computing and mobile apps) is not true.
- Multi-phase and quorum-based commits require numerous round trips between communicating parties. That comes with a cost in terms of latency and throughput. Moreover this approach has tendency to degrade at scale, when number of machines or distance between them increases. In many cases it's not feasible to use distributed transaction across boundaries of a single data center.

- They often determine a single master node used for exclusive write access or as a single source of truth. In some cases - as the ones mentioned above - it's not a feasible solution.

This doesn't mean, that existing approaches are bad. However what we are striving for is to give a wider area of possible solutions, that may better fit the problem under certain conditions. One of the examples, that I like to use to visualize the problem is that:

> Imagine that you need to build a planet-scale video streaming service. Whenever a user uploads a video, we are replicating it across different data centers located on different continents to maintain good throughput and latency, and in result a better user experience. Additionally we want to show users a view count for that video.

Video uploading is a good example of **master-slave** replication. However things may complicate for such a small feature as view count. With many concurrent users over the entire planet and write-heavy characteristics, using the same approach for counter increments is not a great idea, as this may end up with congestion for more popular videos. We don't need transactions, as the nature of the problem allows us to loose constrains of strong consistency in favor of higher availability of our application. However as described earlier most of the existing solutions are based on exclusive write access to a replicated resource. This is where CRDTs and **multi-master** scenarios come to play.

Use cases and implementations

While this was quite simple example, there are many others that we can look up for in current industry:

- Amazon uses CRDTs to keep their order cart in sync. They've also published their database known as Dynamo, which allows AWS audience to make use of CRDTs.
- Riak is one of the most popular solutions in this area. One of their well-known customers are Riot games (company behind League of Legend), which uses Riak to implement their in-game chat.
- Rovio (company behind Angry Birds game series) uses conflict-free counters for their advertisement platform to make their impression counters work freely even in offline scenarios.
- SoundClound has their own implementation of Last-Write-Wins Set build in Go on top of Redis, known as Roshi, which they use for their observers management.
- TomTom makes use of CRDTs to manage their navigation data.
- CREAustralia uses them for their click stream analytics.

There are also other solutions around there:

- AntidoteDB is another, pretty innovative approach to eventually consistent databases. One of its unique feature is transaction support in eventually consistent environment.
- Akka.DistributedData is a plugin for Akka (and Akka.NET) distributed actor programming model, which exposes several CRDT types on top of Akka cluster.
- Redislabs offers CRDB as part of their enterprise Redis database solution.

- Cassandra and ScyllaDB allow their users to make use of eventually consistent counters in their databases.

What does it mean to be conflict-free?

Conflict-free is a vague description, but it comes to a simple statement: **we are operating on a data structures, that don't require exclusive write access and are able to detect concurrent updates and perform deterministic, automatic conflict resolution**. This doesn't mean that conflict doesn't ever occur, but we are able to always determine the output up front, based on a metadata contained within the structure itself. The core structures here are counters, registers and sets, but from them we can compose more advanced ones like maps, graphs or even JSON.

Types of CRDTs

We can discriminate CRDTs using two core categories: state-based (convergent) and operation-based (commutative) data types. No matter which one we talk about, they all consists of two parts: *replication protocol* and *state application* algorithms.

In practice both versions differ heavily on implementation and their "center of gravity" is focused in different place. Since we need some additional metadata to provide automatic conflict resolution: state-based CRDTs encapsulate it as part of the data structure, while operation-based tend to put more of it onto replication protocol itself. Here, I'm going to cover one of the simpler state-based approaches.

Convergent replicated data types

The single most important operation of state-based CRDTs is `Merge` method. What it does is essentially to take a two corresponding replicas of the same logical entity, and produce an updated state as an output. If any conflicts occur, it's up to merge operation to resolve them.

Moreover merge operation must conform to three properties, which give us great perks for using them:

- **Commutativity** `(x • y = y • x)` and **Asociativity** `((x • y) • z = x • (y • z))` which means that we can perform out of order merge operations and still end up with correct state.
- **Idempotency** `(x • x = x)`, so we don't need to care about potential duplicates send from replication layer.

Those properties are not easy to guarantee, but you're going to see how far we can go only by using two basic operations, which meet those criteria:

- union of two sets
- maximum of two values

With them in our hands, the only requirement on our replication layer is to eventually dispatch all state changes to all replicas. There is however a problem with this: to perform those merge operations we need to carry whole data structure on every change. Imagine that we need to send the entire collection of 1000 elements over the wire, only because someone added one extra element. This problem can be solved by the approach known as delta-state CRDTs, which I'll discuss another time.

Now, lets cover some basic data structures and see how we could compose them into more advanced ones. Please keep in mind, that those examples are mind to be simple and present you the approach to the problem.

Counters

Counters are the first type of CRDTs, we'll cover here. Essentially they allow to read, increment and (optionally) decrement a counter value concurrently on many different machines, without worrying about locking.

Growing-only Counter

Also known as **GCounter**. It's a counter which value can only be ever increasing. One of it's use cases could be a *page view counter* I've referred to in the motivation section. Simply speaking it's a map of replica-id/partial-counter values.

In order to calculate total counter's value, we need to sum the values of all known counters specific to particular replicas.

If we want to increment a counter's value, we simply increment a partial counter in the context of the replica, we're working with at the moment. It's crucial, that a single replica should never increment the value of another replica.

```
module GCounter =
    type GCounter = Map<ReplicaId, int64>
    let zero: GCounter = Map.empty
    let value (c: GCounter) =
        c |> Map.fold (fun acc _ v -> acc + v) 0L
    let inc r (c: GCounter) =
        match Map.tryFind r c with
        | Some x -> Map.add r (x + 1) c
        | None   -> Map.add r 1 c
    let merge (a: GCounter) (b: GCounter): GCounter =
        a |> Map.fold (fun acc ka va ->
            match Map.tryFind ka acc with
            | Some vb -> Map.add ka (max va vb) acc
            | None    -> Map.add ka va acc) b
```

When it comes to a `merge` operation, we simply concatenate key/value entries of both counters. When we detect that both counters have different values for the same replica, we simply take a `max`

of both values. This is a correct behavior, since we know that counter values could only be incremented. For the same reason, the decrement operation is not supported by this kind of CRDT.

Increment/decrement Counter

Now, lets talk about so called **PNCounter**, which is able to provide both increment and decrement operations. I think, it's particularly useful, as it's a simple example to show, how we can compose simple CRDTs to build more advanced ones.

The crucial trick here, is that `PNCounter` consists of two `GCounter`s - one of them used to count increments and other used for decrements - so decrement operation is simply incrementing `GCounter` part responsible for counting decrements. Our output value is basically a difference between the two.

```
module PNCounter =
    type PNCounter = GCounter.GCounter * GCounter.GCounter
    let zero: PNCounter = (GCounter.zero, GCounter.zero)
    let value (inc, dec) = GCounter.value inc - GCounter.value dec
    let inc replica (inc, dec) = (GCounter.inc replica inc, dec)
    let dec replica (inc, dec) = (inc, GCounter.inc replica dec)
    let merge (inc1, dec1) (inc2, dec2) =
        (GCounter.merge inc1 inc2, GCounter.merge dec1 dec2)
```

As you can see merge operation is again pretty trivial: a simple merge of corresponding `GCounter` parts from both `PNCounter`s.

There are also other types of counters, which I won't cover here. One of particularly interesting cases are Bounded Counters, which allow you to provide an arbitrary upper/lower bound on the counter to determine if a target threshold has been reached.

Note about vector clocks

We already mentioned counters implementation. Before we go forward, I think it's a good point to talk about the vector clocks and notion of time.

In many systems, a standard way of defining causality (a happened-before relationship) is by using time stamps. There is a problem however with using them in terms of high-frequency distributed systems:

- Operating systems, especially executing on machines in different data centers, can be subject of clock skews, which can kick your butt in write-heavy scenarios. Also another anomalies can occur: leap second bugs or even invalid time values happening between two threads.
- While timestamps can potentially give us information necessary to determine the most recent update (we'll use that in a minute), they won't tell us anything about the "state of the world" at the moment, when update has happened. This means, we cannot detect if one update knew about another, or if they happened concurrently.

This is where vector clocks come to work. They are form of logical clocks, represented by monotonically incremented values, specific for each replica. Sounds much like `GCounter` we've seen above ;)

Why do we talk about them here? The internal implementation of vector clock is very close to what `GCounter` looks like. The major difference here is an ability to partially compare two vector clocks.

> Unlike standard comparison, partial comparison allows us to determine fourth possible result - an indecisive one, when we are no longer able to determine if two values have lesser, greater or equal relationship. We can use this to recognize concurrent updates, which happened between two clocks.

Here, we'll define vector clocks as:

```
type Ord =
    | Lt = -1  // lower
    | Eq = 0   // equal
    | Gt = 1   // greater
    | Cc = 2   // concurrent

type VTime = GCounter.GCounter
module VClock =
    let zero = GCounter.zero
    let inc = GCounter.inc
    let merge = GCounter.merge
    let compare (a: VTime) (b: VTime): Ord =
        let valOrDefault k map =
            match Map.tryFind k map with
            | Some v -> v
            | None   -> 0L
        let akeys = a |> Map.toSeq |> Seq.map fst |> Set.ofSeq
        let bkeys = b |> Map.toSeq |> Seq.map fst |> Set.ofSeq
        (akeys + bkeys)
        |> Seq.fold (fun prev k ->
            let va = valOrDefault k a
            let vb = valOrDefault k b
            match prev with
            | Ord.Eq when va > vb -> Ord.Gt
            | Ord.Eq when va < vb -> Ord.Lt
            | Ord.Lt when va > vb -> Ord.Cc
            | Ord.Gt when va < vb -> Ord.Cc
            | _ -> prev ) Ord.Eq
```

The comparison function, even thou long, is pretty simple - we'll compare pairwise entries of both `VTime` maps (if an entry didn't exist on the opposite side, we count its value as 0):

- If all values of corresponding replicas are equal, clocks are equal:
- If all values on the left side are lower than or equal to their counterparts on the right side, left side is lesser than the right one.

- If all values on the left side are greater than or equal to their counterparts on the right side, left side is greater than the right one.
- Any mix of lesser/greater entries comparison means, that we detected a concurrent update.

If you're more interested about the topic of time in distributed systems, I can recommend you a great talk about this subject: Keeping Time in Real Systems by Kavya Joshi.

Registers

The next type of CRDTs are registers. You can think of them as value cells, that are able to provide CRDT semantic over any defined type. Remember, that we're still constrained by commutativity/associativity/idempotency rules. For this reason we must apply additional metadata, which will allow us to provide arbitrary conflict resolution in case of conflict detection.

Last Write Wins Register

The most obvious way to solve conflicts, we already have talked about earlier, is to use timestamps. This is exactly what our implementation of `LWWReg` uses.

```
module LWWReg =
    type LWWReg<'a> = 'a * DateTime
    let zero: LWWReg<'a> = (Unchecked.defaultof<'a>, DateTime.MinValue)
    let value (v, _) = v
    let set c2 v2 (v1, c1) = if c1 < c2 then (v2, c2) else (v1, c1)
    let merge (v1, c1) (v2, c2) = if c1 < c2 then (v2, c2) else (v1, c1)
```

It's quite obvious. Our `set` and `merge` operations simply compare two registers and pick register's value with a higher timestamp.

Just like in previous cases, we'll be able to compose LWW registers with other CRDTs to provide more advanced operations.

Sets

Once we've covered counters and registers, it's time to talk about collections. The most natural candidate there are different variations of sets - simply because set union conforms to associativity/commutativity/idempotency properties mentioned before. Later on, we could use them to define structures like maps, graphs or even indexed linear sequences (useful i.e. in collaborative text editing).

Growing-only Set

Just like in case of counters, here the most basic example is a growing-only set, also known as `GSet`. One of its cases could be i.e. a *voting system, where we'd like to tell if a person has participated in voting, while still making his/her vote anonymous (in this case a total voting result could be `GCounter` itself)*.

```
module GSet =
    type GSet<'a when 'a: comparison> = Set<'a>
    let zero: GSet<'a> = Set.empty
    let value (s: GSet<'a>) = s
    let add v (s: GSet<'a>) = Set.add v s
    let merge (a: GSet<'a>) (b: GSet<'a>) = a + b
```

No, it's not trolling. It's just a standard set! :) The only difference here is that we constrain ourselves not to perform any removals on the set. The reason for that is merge operator: since our merge is just a standard union, if we'd remove any element from any of replicas, after merging it with another replica (where that removal hasn't happened yet), removed element with auto-magically reappear in the result set.

There's also a lesson here: because we removed an element from the set, we lost some data. This is something, we often cannot afford in case of CRDTs and it's the reason, why we often must attach some additional metadata, even thou it may seem not to be explicitly needed by the result value.

2-Phase Set

The next step is two phase set. Like in case of `PNCounter`, we could simply combine two `GSet`s - one for added elements, and one for removed ones (often referred to as **tombstones**). Add/remove element and merge also works pretty much like in case of `PNCounter` / `GCounter`.

```
module PSet =
    type PSet<'a when 'a: comparison> = GSet.GSet<'a> * GSet.GSet<'a>
    let zero: PSet<'a> = (GSet.zero, GSet.zero)
    // (add, rem) is a single PSet instance
    let value (add, rem) = add - rem
    let add v (add, rem) = (GSet.add v add, rem)
    let rem v (add, rem) = (add, GSet.add v rem)
    let merge (add1, rem1) (add2, rem2) =
        (GSet.merge add1 add2, GSet.merge rem1 rem2)
```

There are several problems with following implementation:

- Common case of tombstone-based sets is the fact that removed set can grow infinitely, so that final value set will take only fraction of size of actual metadata necessary to keep the sets consistent. There are extra algorithms - known as *tombstone pruning* - to mitigate that problem.
- While we are able to remove added element, the problem appears when we'll try to add removed element again. Since we're don't have any semantic to remove elements from any of the underlying `GSet`s, once removed, element will stay in tombstone forever. This will cause removing it from the final value set. So no re-adding the value for you my friend. Again, we need some extra metadata that will allow us to track causality to determine when add/remove have happened.

Observed Remove Set

If you kept up to this point, congratulations! We're actually going to make a first semi-advanced case here: an observed remove set (known as **ORSet**), which will allow us to freely add/remove elements and still converge when merging replicas from different locations.

How does it work? We'll represent our **ORSet** as add/remove collections, but this time instead of sets, we'll use maps. The keys in those maps will be our elements, while values will be a (partially) comparable timestamps used to mark, when the latest add/remove has happened.

The actual specialization of `ORSet` depends on the timestamp and conflict resolution algorithm used:

- You could use `DateTime` for timestamps and prefer the latest value on conflict resolution. This will essentially give us Last Write Wins semantics (just like in `LWWReg`) over particular elements of the set. This would greatly simplify things, but we're going to do better than that :)
- Other approach is to use vector clocks, we defined earlier in this post. This will allow us to detect, when two replicas have added/removed the same element without knowing about other parties trying to do the same. When such case is detected, we need to arbitrary tell, what the outcome of our conflict resolution algorithm will be. The most common case is usually preferring additions over removals. This is known as **Add-Wins Observed Remove Set** (shortly **AWORSet**). This is what we'll implement here.

```
module ORSet =
    type ORSet<'a when 'a: comparison> = Map<'a, VTime> * Map<'a, VTime>
    let zero: ORSet<'a> = (Map.empty, Map.empty)
```

To get the result set, our `value` function will iterate over add map and remove from it all entries from removals map, where add timestamp is lower than remove timestamp (this means that if both updates were concurrent, we keep the result).

```
    let value (add, rem) =
        rem |> Map.fold(fun acc k vr ->
            match Map.tryFind k acc with
            | Some va when VClock.compare va vr = Ord.Lt -> Map.remove k acc
            | _ -> acc) add
```

Just like in case of `PNCounter` our add/remove operations need to work in context of a particular replica `r` - this is the result of using vector clocks as timestamps. Here, we'll simply add element to corresponding map and increase it's vector clock.

```
    let add r e (add, rem) =
        match Map.tryFind e add, Map.tryFind e rem with
        | Some v, _ -> (Map.add e (VClock.inc r v) add, Map.remove e rem)
        | _, Some v -> (Map.add e (VClock.inc r v) add, Map.remove e rem)
        | _, _ -> (Map.add e (VClock.inc r VClock.zero) add, rem)
```

```
let remove r e (add, rem) =
    match Map.tryFind e add, Map.tryFind e rem with
    | Some v, _ -> (Map.remove e add, Map.add e (VClock.inc r v) rem)
    | _, Some v -> (Map.remove e add, Map.add e (VClock.inc r v) rem)
    | _, _ -> (add, Map.add e (VClock.inc r VClock.zero) rem)
```

Additional thing, you may have noticed here is that we use `Map.remove`. It's safe to do so in this context, as we at the same time add value-timestamp pair to the opposite map, still keeping the information about element presence inside an object.

The most complex part is actual `merge` function. We start from simply squashing corresponding add/remove maps (in case of conflicting timestamps, we will simply merge them together). Then what we need is to converge merged add/remove maps by removing from add map all values with timestamps lower than corresponding entry timestamps in remove map (this already covers concurrent update case, as we decided to *favor additions over removals* at the beginning). For the remove set, we'll simply remove all elements with timestamps lower, equal or concurrent to the ones from add map. Just to keep things fairly compact.

```
let merge (add1, rem1) (add2, rem2) =
    let mergeKeys a b =
        b |> Map.fold (fun acc k vb ->
            match Map.tryFind k acc with
            | Some va -> Map.add k (VClock.merge va vb) acc
            | None -> Map.add k vb acc ) a
    let addk = mergeKeys add1 add2
    let remk = mergeKeys rem1 rem2
    let add = remk |> Map.fold (fun acc k vr ->
        match Map.tryFind k acc with
        | Some va when VClock.compare va vr = Ord.Lt -> Map.remove k acc
        | _ -> acc ) addk
    let rem = addk |> Map.fold (fun acc k va ->
        match Map.tryFind k acc with
        | Some vr when VClock.compare va vr = Ord.Lt -> acc
        | _ -> Map.remove k acc ) remk
    (add, rem)
```

You can see here, we're using a map of vector clocks (which are also maps), which is a sub-optimal solution for this implementation. There are different ways used to mitigate this problem:

1. The simplest way is to compress binary ORSet payload using for example L4Z or GZip.
2. More advanced approach is to modify the existing implementation using Dotted vector versions.

But I hope to write about them another time.

What next?

With this set of data structures in our arsenal, we could build more advanced ones:

- One example would be a Last-Write-Wins Map, which essentially looks somewhat like `type LWWMap<'k,'v> = ORSet<('k * LWWReg<'v>)>` (keep in mind, that comparison should depend only on key component).
- Another one would be a graph structure, which is composed of two sets: one for nodes and one for edges.

As you may see, this is quite big topic and IMHO a pretty interesting one. There are many more things, like delta-state based optimizations nad operation-based CRDTs, which I won't cover here - simply to not turn this post into a book - but I hope to continue the upcoming posts.

Optimizing state-based CRDTs

Last time we've talked about what are CRDTs and introduced the state-based variant of them. In this section we'll talk about the downsides of presented approaches and ways to optimize them. If you're not familiar with the previous post or CRDTs in general, I highly encourage you to read it, as I'll be referring to it a lot.

I've decided to split this topic in two. This is the first part, which aims to give you the better overview of delta-based state CRDT optimization. The second part will target more specific cases - causal contexts and observed-remove sets.

Updating state with deltas

As I mentioned in previous post, a state-based CRDT replication depends on the ability to serialize and disseminate the entire data structure over the network. While this approach has some advantages - i.e. very simple and straightforward replication protocol - it also comes with a cost.

Imagine that you have a set of 1000 elements. Now, when we'll try to add another item to it, the only way to synchronize this replica with others is to send an entire payload over the wire. It means serializing and sending 1001 elements, even though most of them have remain unchanged. This is even more painful given the fact that state-based CRDTs often carry additional metadata with them.

To face this problem, an alternative approach has been proposed: a delta-state CRDT. As the name suggests, **instead of sending an entire state, we're going to accumulate the updates since the last synchronization as part of change set (delta)**. Once sync message was send, we simply reset delta and start from scratch. This way we'll send only the unseen parts.

This design doesn't stop us from performing full-state merge from time to time: delta-aware CRDTs still maintain the semantics of a state-based ones. Keep in mind that in terms of persistence, the delta itself doesn't have to be stored on disk.

If you want to play with the example data structures, I'm presenting here, feel free to take a look at this repository.

Delta-state growing-only counter

Let's start from defining some helper functions, that we'll use later on:

```
[<AutoOpen>]
module Helpers =

    /// Insert or update the value of the map.
    let upsert k v fn map =
        match Map.tryFind k map with
        | None -> Map.add k v map
        | Some v -> Map.add k (fn v) map

    /// Option-aware merge operation.
    let mergeOption merge a b =
        match a, b with
        | Some x, Some y -> Some (merge x y)
        | Some x, None   -> Some x
        | None, Some y   -> Some y
        | None, None     -> None
```

Now, let's write the G-Counter implementation and the explanation will follow:

```
type GCounter = GCounter of values:Map<ReplicaId, int64> * delta:GCounter option

[<RequireQualifiedAccess>]
module GCounter =
    /// Empty G-counter
    let zero = GCounter(Map.empty, None)
    /// Compute value of the G-counter
    let value (GCounter(v, _)) =
        v |> Map.toSeq |> Seq.map snd |> Seq.fold (+) 0L
    /// Increment G-counter value for a given replica.
    let inc r (GCounter(v, d)) =
        let add1 map = upsert r 1L ((+) 1L) map
        let (GCounter(dmap,None)) = defaultArg d zero
        GCounter(add1 v, Some(GCounter(add1 dmap, None)))
    /// Merge two G-counters.
    let rec merge (GCounter(a, da)) (GCounter(b, db)) =
        let values = a |> Map.fold (fun acc k va -> upsert k va (max va) acc) b
        let delta = mergeOption merge da db
        GCounter(values, delta)
    /// Merge full-state G-counter with G-counter delta.
    let mergeDelta delta counter = merge counter delta
    /// Split G-counter into full-state G-counter with empty delta, and a delta itself.
    let split (GCounter(v, d)) = GCounter(v, None), d
```

While this example is more complex than the original GCounter implementation, we again try to keep things fairly easy. Here, we're gonna represent `GCounter` values as map of partial counters for each replica (just like the last time)... while the detla is optionally a `GCounter` itself! Why? Because of composition of course :) This way we can merge counters with deltas and deltas themselves using the same `merge` operation we defined before. In this design the delta of a delta counter will always be `None` - we don't use it, so there's no risk of creating possibly infinite recursive data structure.

Delta-state increment/decrement counter

Previously, we've build a `PNCounter` using two `GCounter`s (one for incremented and one for decremented values). While you may ask, how will this work now? This is pretty simple: we'll build a delta of a `PNCounter` dynamically from deltas of its two components.

```
type PNCounter = PNCounter of inc:GCounter * dec:GCounter

[<RequireQualifiedAccess>]
module PNCounter =
    let zero = PNCounter(GCounter.zero, GCounter.zero)
    let value (PNCounter(inc, dec)) = GCounter.value inc - GCounter.value dec
    let inc r (PNCounter(inc, dec)) = PNCounter(GCounter.inc r inc, dec)
    let dec r (PNCounter(inc, dec)) = PNCounter(inc, GCounter.inc r dec)
    let rec merge (PNCounter(inc1, dec1)) (PNCounter(inc2, dec2)) =
        PNCounter(GCounter.merge inc1 inc2, GCounter.merge dec1 dec2)
    let mergeDelta delta counter = merge counter delta
    let split (PNCounter(GCounter(inc, a), GCounter(dec, b))) =
        let delta =
            match a, b with
            | None, None -> None
            | _, _       ->
                let inc = defaultArg a GCounter.zero
                let dec = defaultArg b GCounter.zero
                Some <| PNCounter(inc, dec)
        PNCounter(GCounter(inc, None), GCounter(dec, None)), delta
```

I think, the only tricky part here is delta construction. We want it to be a (optionally) `PNCounter` itself - because of this way we still can preserve **commutativity**, **associativity** and **idempotency** rules, all of our CRDT conform to, but also to reuse implementations of existing functions. Thankfully we can simply build `PNCounter` from two `GCounter` and since **`GCounter.delta` itself is a `GCounter` instance**, we can leverage that fact. If one of the deltas is not provided, we simply use a zero element.

Delta-state growing-only set

For the `GSet`, its implementation is pretty analogous to `GCounter`:

```
type GSet<'a when 'a: comparison> = GSet of values:Set<'a> * delta:GSet<'a> option

module GSet =
```

```
let zero = GSet(Set.empty, None)
let value (GSet(s, _)) = s
let add elem (GSet(v, d)) =
    let (GSet(delta, None)) = defaultArg d zero
    GSet(Set.add elem v, Some(GSet(Set.add elem delta, None)))
let rec merge (GSet(a, da)) (GSet(b, db)) =
    let values = a + b
    let delta = Helpers.mergeOption merge da db
    GSet(values, delta)
let mergeDelta delta gset = merge gset delta
let split (GSet(v, d)) = GSet(v, None), d
```

Just like in case of original state-based CRDTs, we can take advantage of composability.

The technical difference here is that for G-Sets we should acknowledge either deltas or full state updates being broadcasted to all replicas. It was not so necessary for the counter implementations, as if we sent two following updates (*D1*, *D2*), if a first one (*D1*) didn't reach the target, but the later one (*D2*) did, we'll still end up in correct state - simply because *D2* already override the partial counter value of *D1*.

Final notes about replication protocol

Up to this point our replication protocol is still pretty simple - the only requirement here is that we have to eventually be able to reach every single replica. However this doesn't have to occur immediately and doesn't have to occur in order (read: you're not constrained to use TCP for delta replication).

In the code presented above I'm merging deltas as part of a merge operation. This however is not a requirement, i.e. if you disseminate delta right after the operation being performed, you may as well clear deltas on each merge.

In case if you're lost with code snippets described here, you can always play with them using this Github repository.

In the previous section we discussed problems with state-based approach to CRDTs. Major issue was, that in order to achieve convergence between replicas living on a different machines, we had to serialize and push the entire data structure over the network. With small number of changes and big collections this approach quickly becomes not optimal.

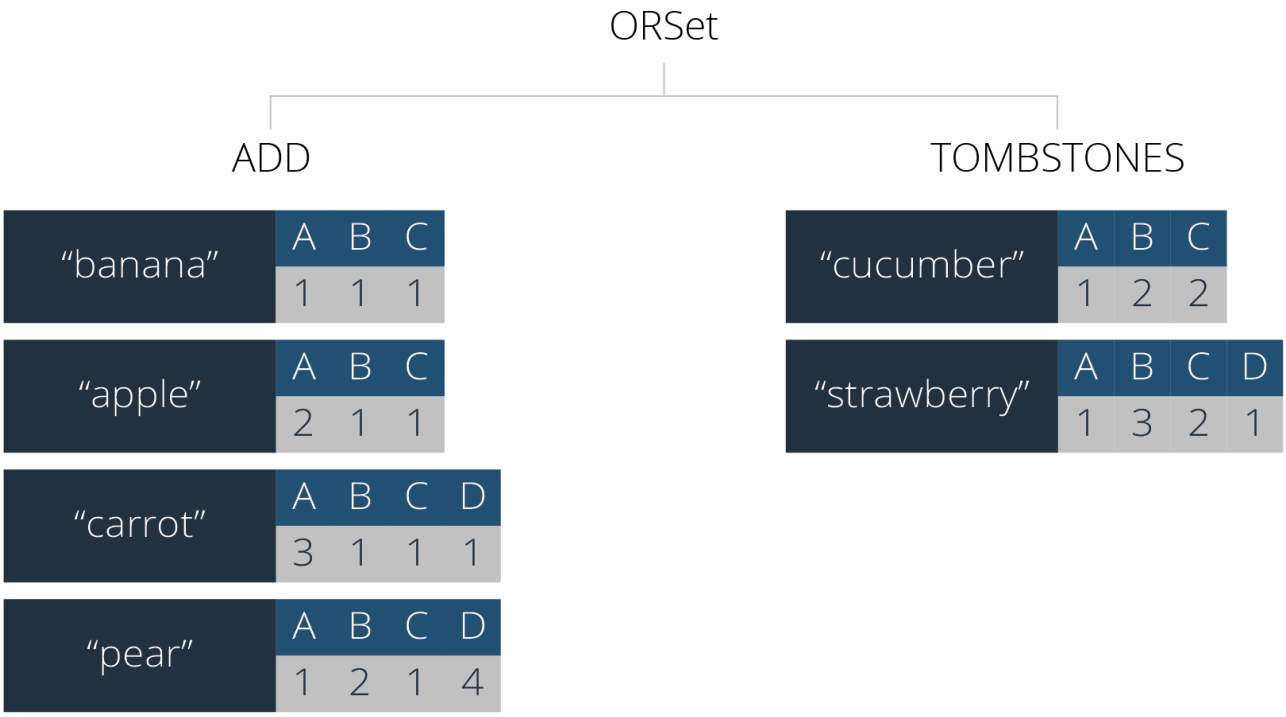
> **Example**: we created a Twitter-like service and used CRDT replicated ORSet to represent a list of user's followers. In order to keep replicas on different machines in sync, we need to copy entire set over the network - ideally on every update. Imagine replicating all followers of i.e. Elon Musk (22mln people at the moment) every time a new person follows/unfollows him.

One of the solutions we talked about, was introduction of deltas, used to contain only the latest changeset. Finally we presented, how to implement them using counters as an example. Here we're

take more rough ride - we'll do the same with sets and registers.

Inefficiency of Observed-Remove Sets

In the first part of the CRDT series we've seen a very naive implementation Observed-Remove Sets - a CRDTs collections which are able to track elements added and removed on different replicas without any need of synchronization. We could visualize this data structure using following image:



However this design comes with huge cost:

1. We're tracking not only active elements of the set, but all elements that have ever appeared in it - removed elements are stored as tombstones.
2. We attached a vector clock to keep "time" of element addition/removal. Since vector clocks are map-like structures, the size of metadata could even overgrow the actual payload we're interested in.

Can we do something with this? Of course, yes ;) But to do that, first we need to revisit our approach to how do we understand the notion of time.

Dots

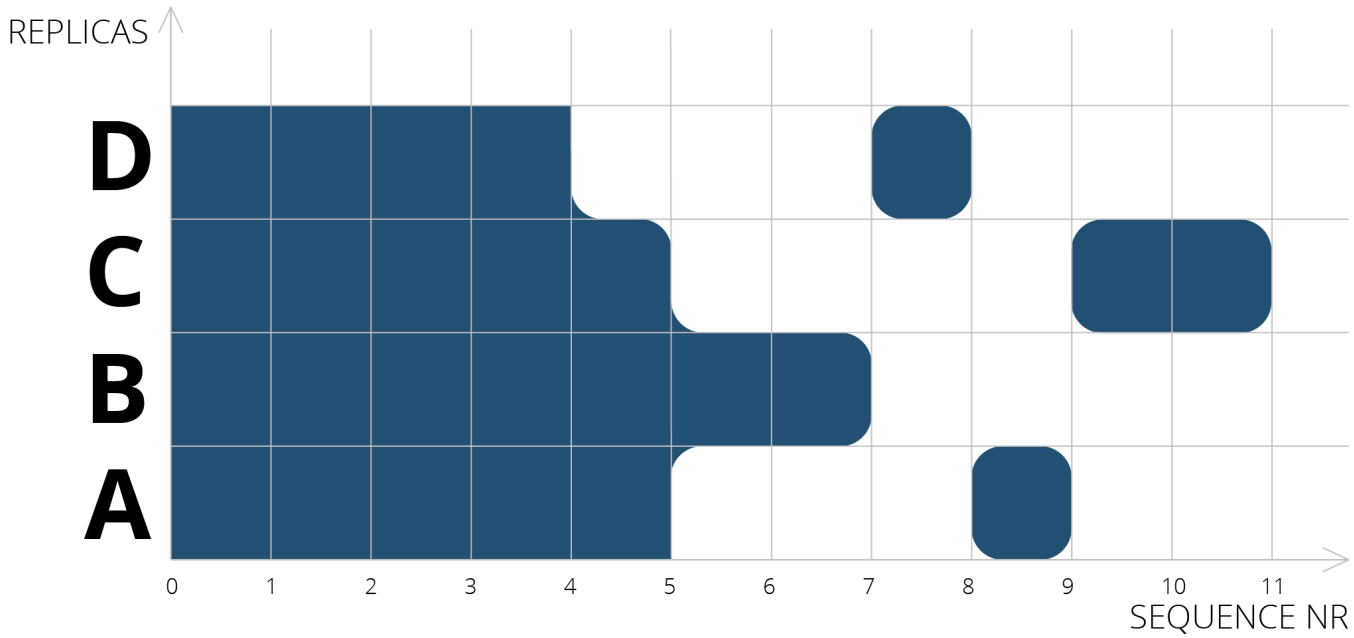
Previously, we already learned how to represent time (or rater happened-before relations between events) using vector clocks. But let's think, what a minimal representation of an operation (addition or removal) could look like in eventually consistent systems?

We can use monotonically auto-incremented sequence number for each operation - just like when we are using traditional SQL databases. However since we cannot use a single global sequence number - as this would require replicas to coordinate a next number with each other - we'll use a separate sequencer for every replica and use pair of `(replica-id, sequence-nr)` (also known as **dot**) to represent a timestamp of a particular operation made on that replica. A dot has several properties:

1. It allows us to uniquely identify every single operation made on any replica.
2. It allows us to track causal relations between events - so while we may not be able to set a total order of events (fortunately we don't need it), we still know how to order operations made by a single replica.
3. Unlike standard time, for every operation we increase sequence number by one. This way we can keep track of "holes" - a discontinuous events we haven't yet observed for some reason.

When using standard wall clock time, we're used to represent observed events using markers on some sort of a timeline, like this:

However, with **dots** we can represent them using 2-dimensional space instead:



Each filled cell represents a discrete event we observed, described as a dot. As you can see, most of the events form a continuous surface, only few ones are detached. This reflects a real life situation, since as we replicate our updates among replicas, each replica will try to keep up with incoming

changes, filling the gaps it was not aware of before, thus over time filling the space from left to right. This is the place, we can optimize.

We could split area of our observed dots in two:

1. A "flatten" version represented by vector clock, which contains only the oldest dot from continuous space. I.e. given dots `A:1`, `A:2`, `A:3`, `A:5`, `A:6` it would be `A:3` as this is as far as we can go without reaching the gap (`A:4` is missing in this case).
2. A second set of all dots that are could not fit inside vector clock. So for the example above this set would consist of `A:5` and `A:6`, as those dots are detached from the rest. We'll call it a dot cloud.

Together those two form something, we'll call a `DotContext` - a logical representation of all observed events from a perspective of given replica (we've seen a visual representation of it in on a diagram above):

```
type Dot = ReplicaId * int64
type VectorClock = Map<ReplicaId, int64>;
type DotContext = { Clock: VectorClock; DotCloud: Set<Dot> }
```

Now, we need to know which events have been already acknowledged by our replica, as we want to know which of the incoming events are important for us and which one are outdated or duplicated. With dot context, it's easy - dot has been observed if either its sequence number is less or equal a sequence number for corresponding replica in a vector clock or if that dot was found in a cloud of detached dots:

```
let contains dot ctx =
    let (r, n) = dot
    match Map.tryFind r ctx.Clock with
    | Some found when found >= n -> true
    | _ -> Set.contains dot ctx.DotCloud
```

We can also use `DotContext` assigning a new dots for the events, that we want to produce:

```
let nextDot r (ctx): Dot * DotContext =
    let newCtx = { ctx with Clock = Helpers.upsert r 1L ((+)1L) ctx.Clock }
    ((r, newCtx.Clock.[r]), newCtx)
```

As you can see here, in order to generate next dot value, we're using only vector clocks - we can do so simply because current replica is *always* up-to-date with its own updates, so it cannot see any detached events produced by itself.

Now, since we're using dot context to keep time of the updates, we'll need to pass it as a metadata along with the update itself, either in form of a full CRDT merge or using slimmer version - a delta

containing only dot context with the latest updates.

If you've read previous sections, you probably know what we'll need now - a `merge` function. Since we already talked how to merge sets and vector clocks/counters, we can compose both approaches to merge dot contexts themselves:

```
let merge a b =
    let mergeMap x y =
        y |> Map.fold (fun acc k v -> Helpers.upsert k v (max v) acc) x
    let clock = mergeMap a.Clock b.Clock
    let cloud = a.DotCloud + b.DotCloud
    { Clock = clock; DotCloud = cloud } |> compact
```

But wait: what is this mysterious `compact` function at the end? Well, this is when the fun begins...

When cloud grows

As you can imagine, when we merge two different contexts, some of the dots representing detached events can potentially no longer be detached - if we leave them that way we'll eventually end up with small vector clock and a big pile of garbage living in a dot cloud.

This is why we need compaction. The compaction process itself can be represented in following steps:

- Traverse for each dot in a dot cloud:
  i. Check if a dot is no longer detached - if its sequence number is exactly one more than its replica counterpart in vector clock, it means that this event is actually continuous, so it can be joined to vector clock.
  ii. Check if a dot's sequence number is less than or equal than its counterpart in vector clock - if it's so, it has been already represented inside vector clock itself, so we no longer need it.
  iii. If dot doesn't match cases 1. or 2., it remains detached, so it should stay in a dot cloud.

```
let compact ctx =
    let (clock, dotsToRemove) =
        ctx.DotCloud
        |> Set.fold (fun (cc, rem) (r,n) ->
            let n2 = defaultArg (Map.tryFind r cc) 0L
            // check if dot happens right after the latest one
            // observed in vector clock
            if n = n2 + 1L then (Map.add r n cc, Set.add (r,n) rem)
            // check if dot is represented "inside" vector clock
            elif n <= n2   then (cc, Set.add (r,n) rem)
            // go here if dot remains detached
            else (cc, rem)
        ) (ctx.Clock, Set.empty)
    { Clock = clock; DotCloud = ctx.DotCloud - dotsToRemove}
```

What is important here is that in F# `Set.fold` always traverses elements using lexicographic order. This means that we'll always process `A:2` before `A:3`, which is important for this implementation if we want to merge all continuous dots into a vector clock.

Kernel

Now, once we are able to manage time space of our observations via `DotContext`, we could use it to construct our first instance of delta-aware `AWORSet` (Add-Wins Observed Remove Set). However as you'll see later, we can implement several different CRDTs using more or less the same mechanics. Therefore we'll introduce here a common abstraction over them, we'll refer to as `DotKernel`.

A dot kernel can be imagined as a Map of dots to values with a single dot context used for time keeping.

```
type DotKernel<'a when 'a: equality> =
    { Context: DotContext
      Entries: Map<Dot, 'a> }
```

What's important here, the `Entries` map contains only information about active (not-removed) elements. You could ask now: *but in original implementation of `ORSet<>` we needed removed elements as well in order to resolve merge conflicts - how are we going to resolve them now?*

There's a simple solution here - we're going to use our dot context to track information about remove operations. As you remember each operation and its occurrence can be represented by a single **dot**. Now, when we merge two kernels, all we need to do is to check if element identified by its dot can be found inside entries and dot context of the second entries.

If element's dot was found in a dot context, it means that other kernel already knew about its addition. In that case we lookup for that element in kernel's entries - if dot was not there, it means that by now it must have been removed!

```
// get active values from the kernel
let values k = k.Entries |> Map.toSeq |> Seq.map snd

// merge two kernels together
let merge a b =
    let active =
        b.Entries
        |> Map.fold (fun acc dot v ->
            // add unseen elements
            if not <| (Map.containsKey dot a.Entries || DotContext.contains dot a.Context)
            then Map.add dot v acc else acc) a.Entries
    let final =
        a.Entries
        |> Map.fold (fun acc dot _ ->
            // remove elements visible in dot context but not among Entries
            if DotContext.contains dot b.Context && not <| Map.containsKey dot b.Entries
```

```
            then Map.remove dot acc else acc) active
    { Entries = final; Context = DotContext.merge a.Context b.Context }
```

All that's remaining now, is to implement addition/removal operations.

For element addition, we start from reserving a new dot for our addition operation, and we'll store it with element inside entries and in kernel's context.

```
let add replica value (k, d) =
    let (dot, ctx) = DotContext.nextDot replica k.Context
    let kernel = { k with
        Entries = Map.add dot value k.Entries;
        Context = ctx }
    let delta = { d with
        Entries = Map.add dot value d.Entries;
        Context = DotContext.add dot d.Context
                  |> DotContext.compact }
    (kernel, delta)
```

For element removal operation is even simpler - we don't even need a new dot! We can simply pick the dot which so far was used to mark additions and just drop element from kernel's entries - after all **we recognize removal when it was acknowledged by kernel's dot context but not found among its entries**, remember?

```
let remove value (k, d) =
    let (entries, deltaCtx) =
        k.Entries
        |> Map.fold (fun (e, dc) dot v2 ->
            if v2 = value
            then (Map.remove dot e, DotContext.add dot dc)
            else (e, dc)
        ) (k.Entries, d.Context)
    let kernel = { k with Entries = entries }
    let delta = { d with Context = deltaCtx |> DotContext.compact }
    (kernel, delta)
```

We'll return to `DotKernel` once more in a minute, but right now we have all, that we need to implement our ORSet.

Delta-aware Add-Wins Observed Remove Set

Now, finally! We've got everything in place to create our implementation of efficient `AWORSet` with delta updates. We'll can simply represent entire set with delta in terms of dot kernels:

```
type AWORSet<'a when 'a: comparison> =
```

```
    AWORSet of core:DotKernel<'a> * delta:DotKernel<'a> option
```

With dot kernel, we can simply recreate all core delta-CRDT operations, we've defined for our counters in the previous section:

```
module AWORSet

// zero or empty element
let zero = AWORSet(DotKernel.zero, None)

// An active set of entries
let value (AWORSet(k, _)) =
    k
    |> DotKernel.values
    |> Set.ofSeq

let add r v (AWORSet(k, d)) =
    let (k2, d2) = DotKernel.remove r v (k, defaultArg d DotKernel.zero)
    let (k3, d3) = DotKernel.add r v (k2, d2)
    AWORSet(k3, Some d3)

let rem r v (AWORSet(k, d)) =
    let (k2, d2) = DotKernel.remove r v (k, defaultArg d DotKernel.zero)
    AWORSet(k2, Some d2)

let merge (AWORSet(ka, da)) (AWORSet(kb, db)) =
    let dc = Helpers.mergeOption DotKernel.merge da db
    let kc = DotKernel.merge ka kb
    AWORSet(kc, dc)

let mergeDelta (AWORSet(ka, da)) (delta) =
    let dc = Helpers.mergeOption DotKernel.merge da (Some delta)
    let kc = DotKernel.merge ka delta
    AWORSet(kc, dc)

let split (AWORSet(k, d)) = AWORSet(k, None), d
```

As you can see, this implementation is pretty straightforward - we've already done all of the heavy lifting, when we defined `DotKernel` and `DotContext`. Now we're ready to reuse them in other fields.

Multi Value Registers

Remember, when I told you that `DotKernel` can be used as a generalization over many different kinds of CRDTs? Let's implement one more - a Multi-Value Register.

An idea of registers was mentioned previously - back then we used a Last-Write-Wins register, which allowed us to encapsulate any value into CRDT. As name suggests LWW register was using wall clock time to determine which value should stay in case of conflict.

Multi-Value Register (or `MVReg`) works differently - once a conflicting update has been detected, it doesn't try to guess the right answer. Instead it returns concurrent values and lets the programmer to decide which one should be used.

```
type MVReg<'a> when 'a: equality =
    MVReg of core:DotKernel<'a> * delta:DotKernel<'a> option

module MVReg =
    let zero: MVReg<_> = MVReg(DotKernel.zero, None)
    let value (MVReg(k, _)) =
        DotKernel.values k |> Seq.distinct |> Seq.toList

    let set r v (MVReg(k, d)) =
        let (k2, d2) = DotKernel.removeAll (k, defaultArg d DotKernel.zero)
        let (k3, d3) = DotKernel.add r v (k2, d2)
        MVReg(k3, Some d3)

    let merge (MVReg(ka, da)) (MVReg(kb, db)) =
        let dc = Helpers.mergeOption DotKernel.merge da db
        let kc = DotKernel.merge ka kb
        MVReg(kc, dc)

    let mergeDelta (MVReg(ka, da)) (delta) =
        let dc = Helpers.mergeOption DotKernel.merge da (Some delta)
        let kc = DotKernel.merge ka delta
        MVReg(kc, dc)

    let split (MVReg(k, d)) = MVReg(k, None), d
```

As you can see the actual implementation is almost identical to `AWORSet` - that's why we wanted to have dot kernel on the first place. The major difference where is `DotKernel.removeAll` function. As we haven't introduce it before, lets do it now:

```
module DotKernel =
    let removeAll (k, d) =
        let deltaCtx =
            k.Entries
            |> Map.fold (fun acc dot _ -> DotContext.add dot acc) d.Context
        let kernel = { k with Entries = Map.empty }
        let delta = { d with Context = deltaCtx |> DotContext.compact }
        (kernel, delta)
```

What we do here is to simply copy all dots from the kernel's entries to its context - so that we can keep them in tracked history for the merging purpose - and clear the entries themselves.

Summary

As you can see idea of dots is quite powerful, as it allows us to encode different kinds of CRDTs in efficient manner. We could elevate structures defined here to compose things like maps, graphs etc. However that's another story.

State-based CRDTs: Bounded Counter

Originally, I didn't want to make a separate section about design behind bounded counters, but since beside original paper and a very few implementation living in the wild, this CRDT is widely unknown, I've decided to give it a try.

Motivation

We already introduced some of the CRDT counters in the past, namely G-Counter (grow-only counter) and PN-Counter (which gave us both increment and decrement operations). While they have proven to be great in many eventually consistent scenarios working on a massive scale - some proposed ones where web page visit counters, likes/dislikes or click counters. Now let's see how we could extend their design further.

A **Bounded Counter** variant, we're describing here, was build to address a specific problem: *is it possible to build a highly available, eventually consistent counter with an upper/lower limit on its value?* This is a useful property, which we could make use of in few scenarios, eg.:

1. *An primary scenario shown in the research paper*: we have an advertisement company, showing ads over mobile devices. When a company buys X impressions (ad displays) on our platform, we want to track them even when devices are periodically offline, and eventually stop showing them when a total number of impressions displayed to our end users has been reached.
2. We want to distribute selling event tickets. Once all tickets are sold, we want to block platform from overselling them. All of that with tolerance to network splits, in globally-distributed manner.
3. A standard (maybe a bit far-fetched) example of money transfer, in which we want to minimize a risk of customer running into debit.

All of those rely on notion of some kind of boundary value, beyond which we should not allow operations to happen. However, it's not trivial in case of eventually consistent counters, as they don't provide us any "global" counter value, to which we can relate in order to check if our constraint was violated. We only know a "local" value, as observed from the point of view of specific replica.

This means, that we need to shift our approach a little to realize this goal.

Implementation

The core behavior of our bounded counter (here called `BCounter`) is very similar to `PNCounter` described in previous posts. However here we'll add one extra constraint: **our counter cannot be**

decremented to reach the value below zero. This concerns only decrements (increments have no bounds). With that:

- We can build a counter which has an upper bound simply by incrementing it ons start, and then reverting the operation (increment → decrement) to create the upper bound.
- We can easily build a counter with a value floating within a provided range of values by combining two bounded counters together and incrementing/decrementing them both.

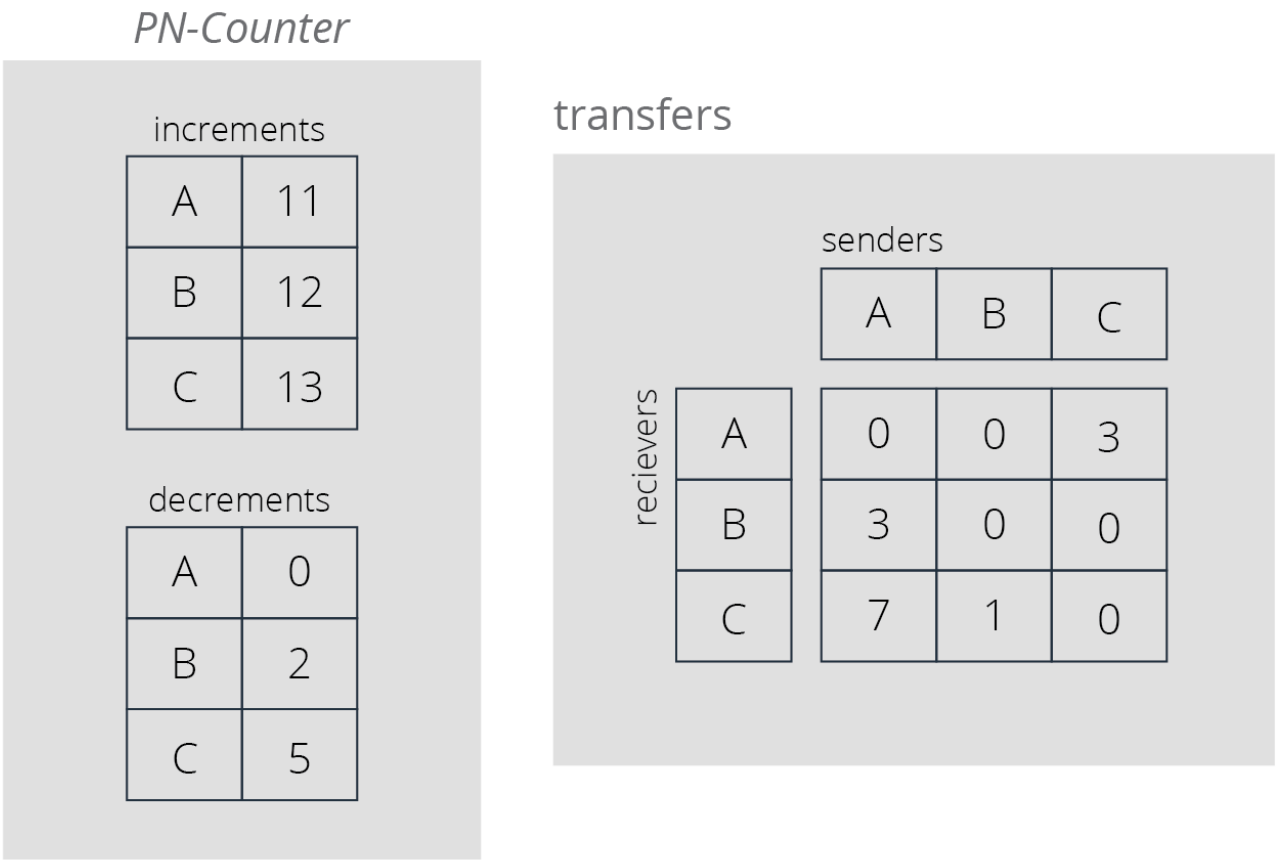
The core concept behind `BCounter` could be represented as purring water between different buckets (replicas): we treat our number as some amount of finite resource - water in this example - which we can pour out, refill or spill from one to another. For this implementation, I'll describe that amount as **quota**.

There's no magic in there. The rules are simple:

- Each replica has its own part of the total quota to consume.
- Replica can increment (`inc` operation) its own quota as it sees fit, therefore increasing a total quota available. This is safe operation, as we don't risk our counter to run below 0 this way.
- Replica can decrement its counter, but only up to it's local quota limit. This mean that our `dec` may fail, due to insufficient number of resources to spend. In that case we may need to try again on another replica.
- Replica can transfer part of its own quota (`transfer` operation) to another replica. Again it cannot share more that it has, so this operation can also potentially fail.

In this post, we won't describe how replicas should negotiate how to split their quota between each other: IMHO this is very dependent on the particular traits of your system. We'll focus on building a foundation, that will make those three operations (`inc` / `dec` / `transfer`) possible.

We'll use `PNCounter` as a base for our `BCounter` to keep track of counter's value. Additionally we'll need a matrix, which we'll use to remember the amounts of quota send between each pair of replicas. Conceptually we can present this data structure in a following way:



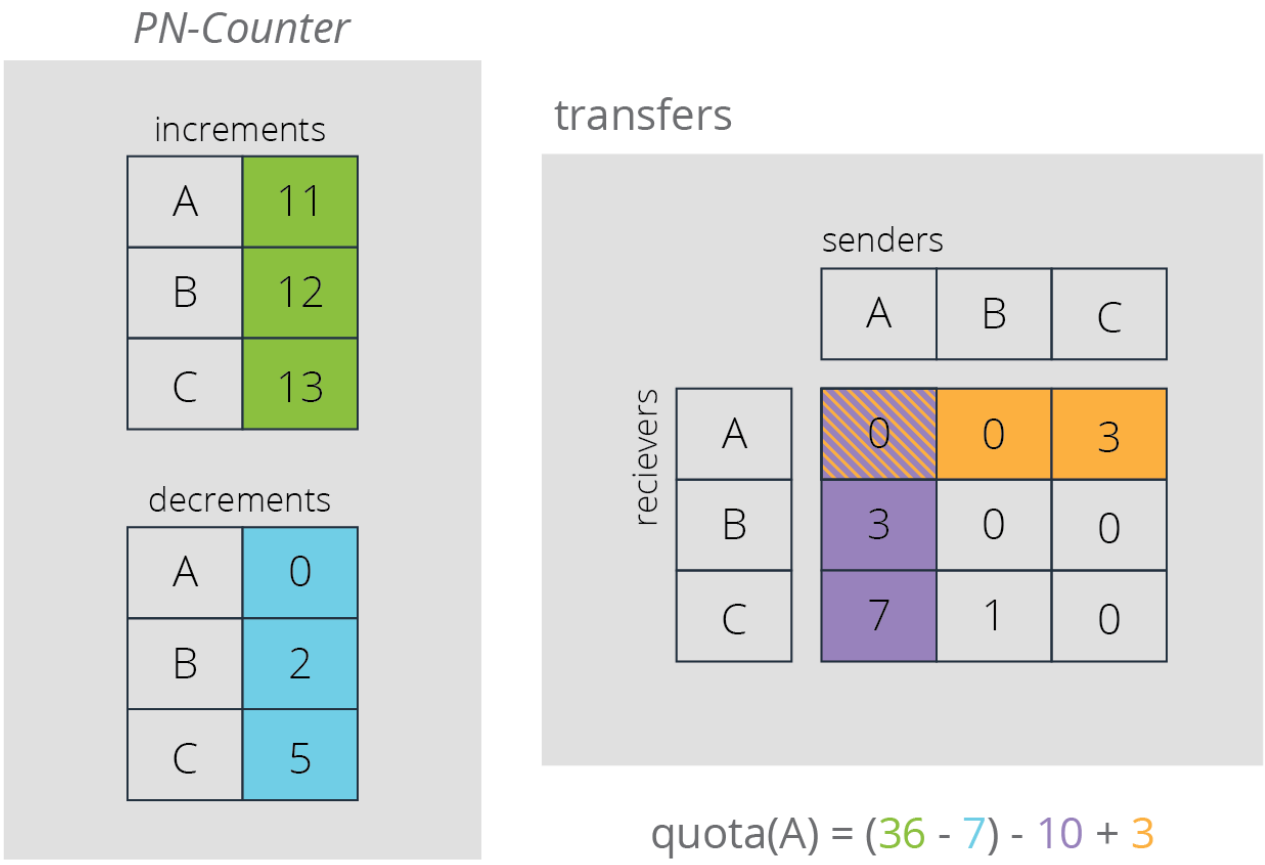
In this implementation, I'll assume that transfer matrix is a sparse one and I'll represent it as a map of keys (sender → receiver) and values (all quota transferred so far between that pair).

```
type BCounter = BCounter of PNCounter * transfers:Map<(ReplicaId * ReplicaId), int64>
```

`PNCounter` tracks the current known value of the counter, which as we mentioned, should never drop below 0:

```
let zero = BCounter(PNCounter.zero, Map.empty)

let value (BCounter(pn, _)) = PNCounter.value pn
```

Now, the most important thing there is to calculate the quota. Ultimately it's pretty simple: we pick current counter's value, add all quota transferred *to it* and subtract of of the quota transferred *from it*. We could illustrate this with our diagram as:



Or simply, in code:

```
let quota replica (BCounter(pn, transfers)) =
    transfers
    |> Map.fold (fun value (sender, receiver) transferred ->
        if   sender = replica   then value - transferred
        elif receiver = replica then value + transferred
        else value
    ) (PNCounter.value pn)
```

Just like in case of previous counters, remember that those operations are context-sensitive: it means that each replica should only "speak" by itself and never try to act on behalf of another replica. With this helper function, we're ready to implement our three operations. Let's start from increment, as it's the easiest one:

```
let inc replica value (BCounter(pn, transfers)) =
    BCounter(PNCounter.inc replica value, transfers)
```

As we said, we are free to increment the total value as we want to. Decrements are a little more tricky:

```
let dec replica value bcounter =
    let q = quota replica bcounter
    if q < value then Error q  // cannot consume more that what we have
```

```
    else
        let (BCounter(pn, transfers)) = bcounter
        Ok (BCounter(PNCounter.dec replica value pn, transfers))
```

Here, a returned value is actually `Result<BCounter, int64>` - on successful decrement we return updated `BCounter`. However as mentioned, decrement may fail if local quota limit doesn't allow us to freely perform that operation. In that case we return an error result with the amount of quota available on local replica.

*In failure scenario we may need to ask other remote replicas to share their quotas, which of course means, that we need at least a periodic connectivity to them and that our replica cannot perform operations **safely** fully on its own, even when a total quota allows to do that.*

This is a known limitation, but as I've said at the beginning, there's no magic here. If two things that cannot communicate, they cannot communicate. It's science.

The last operation we have left, is our quota transfer:

```
let transfer sender receiver value bcounter =
    let q = quota sender bcounter
    if q < value then Error q   // cannot send more than what we have
    else
        let (BCounter(pn, transfers)) = bcounter
        let pair = (sender,receiver)
        let updatedTransfers =
            // if entry for `pair` key doesn't exists insert `value` there
            // if it exists add `value` to a current entry
            Helpers.upsert pair value ((+) value) transfers
        Ok (BCounter (pn, updatedTransfers))
```

Of course, in real distributed environment, this means, that transfer request-response is performed asynchronously, and that remote replicas need to wait for a local state to propagate, before the receiver will be able to take advantage of newly assigned quota.

Since we're talking about state-based CRDTs, there's one thing missing - our glorified `merge` function, which allows us to automatically resolve all potential conflicts between two replicas. Fortunately, we already know how to merge `PNCounter`s from previous sections (right?), so we only need to figure out, how to merge transfers matrix.

At the very beginning of this series about CRDTs, I've talked about two core operations that maintain idempotence, associativity and commutativity rules, required by `merge` function: union of sets and max value of two numbers (and by extension composition of those operations).

Since we can treat keys of the map as set, we only need to know, how to merge values of the map. We already did that when implementing `GCounter.merge` operation. We only need to replace key type (from `replica` to pair of `(sender,receiver)`) since each entry describes transferred quota between that pair, we know that this value is only incremented - we cannot "untransfer" quota, we

can only transfer it back, by incrementing value of reversed `(receiver, sender)` entry. Because of that, we can safely apply our `max` value in this case (as the greater value is always the one most up-to-date):

```
let merge (BCounter(c1, t1)) (BCounter(c2, t2)) =
    let c3 = PNCounter.merge c1 c2
    let t3 = t2 |> Map.fold (fun acc pair v2 -> Helpers.upsert pair v2 (max v2) acc) t1
    BCounter(c3, t3)
```

With that in place, we have a `BCounter` with a complete set of operations necessary to perform its role.

Summary

This time, we covered how to build a basic bounded counters. We've also mentioned how to maintain upper and both upper-lower boundary (via `BCounter` composition).

While we haven't mentioned how to implement delta-state on top of these, it's really not that hard: we covered composition of delta-state of G-Counters and PN-Counters previously, and exactly the same rules also apply here.

State-based CRDTs: Maps

We'll cover the idea of CRDT maps, and how we could create them and utilize them in common scenarios. A prerequisite for this talk is some general knowledge of CRDTs, especially observed-remove sets, which were already covered on this blog in the past.

The code presented here is available at this repository on Github. Feel free to play with it.

Our map will work in a very similar manner to the Observed-Remove Set, that we implemented earlier:

- We're free to add and remove entries to the map any number of times we wish to.
- In case of concurrent conflicting updates (one replica adds the entry, while other removes it), we'll prefer Add-Wins approach.

One extra requirement for us is a need of addressing another type of conflict - when two replicas add two different values under the same key without knowing about others update. Here we're come after the most open approach: **we'll require, that the values of our map must be CRDTs themselves**. This way we can achieve variety of behaviors by further composition of different CRDTs.

For now let's define a basic frame of our map type, playing along with shape of other data types defined in previous posts:

```
type AWORMap<'k, 'v, 'm when 'k: comparison and 'm: struct and 'm :> IConvergent<'v>> =
    AWORMap of keys:AWORSet<'k> * entries:Map<'k,'v>

[<RequireQualifiedAccess>]
module AWORMap =

    let zero () = AWORMAP(AWORSet.zero, Map.empty)

    let value (AWORMap(_,map)): Map<'k,'v> = map

    let add replica key value (map) = ??? // to be defined

    let rem replica key map = ??? // to be defined

    let merge m1 m2 = ??? // to be defined
```

As you may already expect, the implementation here is pretty simple - we won't try to implement everything from scratch, because we already did it. We'll make use of AWORSet to keep CRDT properties over our keys, and we'll use a separate map to attach values to them.

This way our `AWORMap.value` function doesn't require any extra computations - simply return map component of OR-Map - while making our add and remove functions very straightforward:

```
let add r (key: 'k) (value: 'v) (m: AWORMap<'k,'v,'m>): AWORMap<'k,'v,'m> =
    let (AWORMap(keys, map)) = m
    let keys' = keys |> AWORSet.add r key
    let map' = map |> Map.add key value
    AWORMap(keys', map')

let rem r key (m: AWORMap<'k,'v,'m>): AWORMap<'k,'v,'m> =
    let (AWORMap(keys, map)) = m
    let keys' = keys |> AWORSet.rem r key
    let map' = map |> Map.remove key
    AWORMap(keys', map')
```

All we need is simply to add/remove elements to both OR-set and map building our OR-Map. I guess the confusing part for you might be the `m` type parameter. This is a longer story, quite specific to .NET, so if you're interested, read the appendix below, or just skip it otherwise.

The Trait pattern

Unlike some programming languages (eg. Scala, Rust or Haskell), there's no way to easy extend any arbitrary type with any **abstract** feature in F#. The keyword here is abstract - sure you can "add" new methods to existing types (even the ones defined in other assemblies) using extension methods, however they don't let you abstract over the generic features provided by these methods.

> C# and F# compilers understand extension methods alligned with some features (like LINQ, awaiter pattern or computation expressions), but only if they were defined by compiler creators.

Example of such generic feature could be a `merge` function working over any type implementing our `IConvergent<T>` interface. For exercise imagine that other CRDTs we defined in the past didn't implement that interface (because we defined it just a while ago), and there's no way to modify them now (maybe they were defined in assembly of someone's else library).

In object oriented paradigm we'd probably use an adapter pattern, but functional languages have solved this problem long ago by introduction of type classes a.k.a. traits. We could define such trait together with function using it as follows:

```
type IConvergent<'v> =
  interface
    abstract merge: 'v * 'v -> 'v
  end

module Helpers =
  let inline merge<'m, 'v when 'm: struct and 'm :> IConvergent<'v>> a b =
    Unchecked.defaultof<'m>.merge a b
```

Now, what's so special about it? It turns out, that .NET JIT does pretty good job on specializing the code execution over struct types - in that case we simply produce a zero-size structure implementing given generic functionality, which then is going to be devirtualized:

```
module LWWReg =
  /// Merger for Last-Write-Wins register defined previously.
  [<Struct;NoEquality;NoComparison>]
  type Merge<'a> =
    interface IConvergent<LWWReg<'a>> with
      member __.merge a b = merge a b

// use the API
let a = LWWReg.update DateTime.UtcNow "hello" LWWReg.zero
let b = LWWReg.update DateTime.UtcNow "world" LWWReg.zero
let c = Helpers.merge<LWWReg.Merge<_>, _> a b
```

Now, if you'd take a look into assembly code produced this way, you'd find that our generic parse function is equivalent to calling the underlying implementation directly - our custom generic parser evades all cost related to doing virtual method dispatch and boxing, that would be necessary in other case when interfaces are in use. This way we could use this pattern to create a code, that doesn't impose runtime cost usually related to using abstractions.

In case of our `AWORMap` implementation there's also another improvement from using generic merge resolution, i.e. we could use different struct implementing our `IConvergent<LWWReg<'a>>` interface to revert its behavior (as example using first-write-wins instead of last-write-wins) without any additional

cost but also without risking that two of our data types would behave differently. Counter-example of such case would be union of two C# sets using different comparers, eg:

```
var a = ImmutableHashSet.Create(StringComparer.CurrentCulture, new [] { "A", "a", "b"});
var b = ImmutableHashSet.Create(StringComparer.InvariantCultureIgnoreCase, new [] { "A", "a",

// let's compare the contents
var ab = a.Union(b);
var ba = b.Union(a);
ab.SetEquals(ba); // false - turns out set union is not always commutative
```

By promoting this "implementation detail" into generic type parameter using trait pattern we not only open our data type for further extensions, but also make it safer from weird unexpected behaviors that sometimes could happen at runtime.

Currently the biggest issue of that approach is that neither C# nor even F# compilers are able to infer our generic "trait" struct on their own - something that Rust, Scala or Haskell have no problems with. For this reason we need to specify these type info by hand. Fortunately, there's a hope that this issue will be solved in the future - see Classes for the Masses.

Merge

Now we need to implement our merge function. We could describe it with following steps:

1. From each OR-Map take their OR-Sets (used for keys) and merge them together. This will handle key addition/removal conflicts for us.
2. Now for each key in newly created OR-Set, try to find a corresponding value in each of the map-components of our OR-Maps:
   i. If both maps have a value related to that key, we merge them (using our generic `m` type).
   ii. In other case (when only one map has a value), we'll take it as it is.

> Step 2 defined here is pretty much a definition of recursive merge function for an Option<'a> type. Once you'll start to see how nice these properties fit with each other, there's no way to unsee it.

```
let merge (a: AWORMap<'k,'v,'m>) (b: AWORMap<'k,'v,'m>): AWORMap<'k,'v,'m> =
    let (AWORMap(keys1, m1)) = a
    let (AWORMap(keys2, m2)) = b
    let keys3 = AWORSet.merge keys1 keys2
    let m3 =
        keys3
        |> AWORSet.value
        |> Set.fold (fun acc key ->
            match Map.tryFind key m1, Map.tryFind key m2 with
            // this could be defines as Option.merge
```

```
            | Some v1, Some v2 ->
                let v3 = Helpers.merge<'m,'v> v1 v2
                Map.add key v3 acc
            | Some v, None -> Map.add key v acc
            | None, Some v -> Map.add key v acc
            | None, None -> acc
        ) Map.empty
    AWORMap(keys3, m3)
```

We could end here, as right now, our OR-Map is already done. However in practice, we tend to use it for further composition - while already pretty powerful, it restricts us to use other CRDT-compliant type instances as only possible values.

CRDT map with last write wins updates

In practice we often would like to use our CRDTs over ordinary types - usually it's not possible right away, as we need some sort of metadata to hint us through reliable conflict resolution. However we could take some compromises i.e. let us to operate on any value type using last-write-wins semantics.

In this context, LWW applies only to a situation when two different replicas updated the same key with different values without knowing of each other updates. Other conflicts - like add/remove entry - are still solved via add-wins semantics.

Given the types we already had in our disposition (we already defined Last-Write-Wins register here), definition of such map should be simple enough:

```
type LWWMap<'k,'v> = AWORMap<'k, 'v, LWWReg.Merge<'v>>

module LWWMap =

    let zero: LWWMap<'k,'v> = AWORMap (AWORSet.zero, Map.empty)

    let value (map: LWWMap<'k,'v>): Map<'k,'v> =
        AWORMap.value map
        |> Map.map (fun k reg -> LWWReg.value reg)

    let add replica key value (map: LWWMap<'k,'v>) =
        let reg = LWWReg.zero |> LWWReg.update DateTime.UtcNow value
        AWORMap.add replica key reg map

    let rem replica key (map: LWWMap<'k,'v>) = AWORMap.rem replica key reg map

    let merge (a: LWWMap<'k,'v>) (b: LWWMap<'k,'v>) = AWORMap.merge a b
```

Operation based CRDTs: protocol

Today we'll continue a series about CRDTs, this time however we'll stray from the path of state-based CRDTs and start talking about their operation-based relatives. The major difference that we need to cover, is the center of gravity of this approach: the replication protocol.

While we've rarely even mentioned it in case of Convergent Replicated Data Types. Thanks to associativity, commutativity and idempotency powering the `merge` operator, replication was not really an issue. Here it's the main difficulty. It has its merits though - once we're done with it, our path will be paved into much simpler and more straightforward data type implementations.

We'll cover necessary properties to be satisfied by such protocol, discuss technical challenges of its practical implementation and at the end provide a first most basic op-based CRDT.

Prerequisites

Since CRDTs have been discussed on this blog many times, we're not going to cover everything from scratch. You should understand motivation behind them - what they are, where and why we're using them for. If you're not familiar with it, please read this introduction.

Another piece, we're not going to discuss in detail (because we already did) are vector clocks, which will be used in further implementation.

While it's not strictly necessary knowledge, it will be useful for you to know the basic concepts of eventsourcing, as we're going to refer to them by analogy later on.

For the implementation part, we're going to use F# and Akka.NET. We do so, because we need a thread-safe state modification in face of concurrent updates, and actor programming model is pretty good way to achieve that.

Protocol

The core replication protocol used for operation-based CRDTs can be described as Reliable Causal Broadcast (RCB), which is not really a name of some algorithm but rather a set of requirements that this protocol must satisfy. Let's go and define what does it mean.

Reliability

In this context reliable means: all updates visible locally must be eventually seen by others, **always**. Sounds trivial, right? ~~Just put TCP pipe between two nodes and we're done~~ Nope. Sometimes, we can get periodic disconnects, while the system should keep on going. Even more: we may have updated state locally, our system was shutdown and came back up. If we committed them locally, we still need to send these updates to other peers. Otherwise we end up in desynchronized state.

More importantly, CRDTs are known by their instantaneous response time for updates. We shouldn't have to wait for others to confirm receiving the update before confirming it locally. CRDTs are meant to work independently (including offline mode) and we want to maintain that property.

Now, there's a known way of persisting operations known as eventsourcing, which as you'll see, plays nicely with the rest of our requirements. We'll persist our updates as events and replay them on demand. Now we only need to figure out how to replicate them to others...

Causality

We could ask ourselves: maybe we just let Kafka/Pulsar/event-log-of-the-week carry this problem for us? But we (indirectly) already answered it: we need our nodes to operate independently. Most of the existing distributed event logs replicate events by establishing consensus about the total order of events they all share within the scope of a given topic. This means that they need to coordinate that order with each other. Therefore when working within a group (cluster), they cannot establish the order of events on their own. Neither can they work when in offline mode.

This is where causality aka. happens-before relation comes in. As it turns out **it's not possible to have one total order of replicated events AND let nodes to write events independently at the same time**. We cannot eat cake and have it too. The max we can offer is partial order in form of causal ordering. What does it mean?:

1. We CAN guarantee the order of events within the scope of a single node/actor. Eg. If *Alice* wrote letters *A* then *B*, everyone reading her messages will always see *A* happening before *B*.
2. We CANNOT guarantee order of concurrent updates. Eg: if *Alice* wrote *A* and *Ben* wrote *B* at the same time (before they replicated each other's events), after replicating their changes to others, some peers may see events in order $A \rightarrow B$, while others $B \rightarrow A$. It's a necessary tradeoff of working independently.
3. We CAN guarantee order of events, when one of them could cause (hence name causal) another one to happen, even if they happened on different nodes. Eg:
    i. *Alice* wrote *A*, which was replicated to *Ben* and *Sam*.
    ii. After receiving *A*, *Ben* replied with *B*. *Sam* responded to *A* with *C* "in the same time" (meaning *C* and *B* were not yet replicated between the nodes).

    Now, after replicating all events, what's will be a possible order? We could see $A \rightarrow B \rightarrow C$ or $A \rightarrow C \rightarrow B$ (because of point 2), but we can promise that *B* and *C* will NEVER be visible before *A* (like in point 1). This means that we will never see an answer before question it refers to.

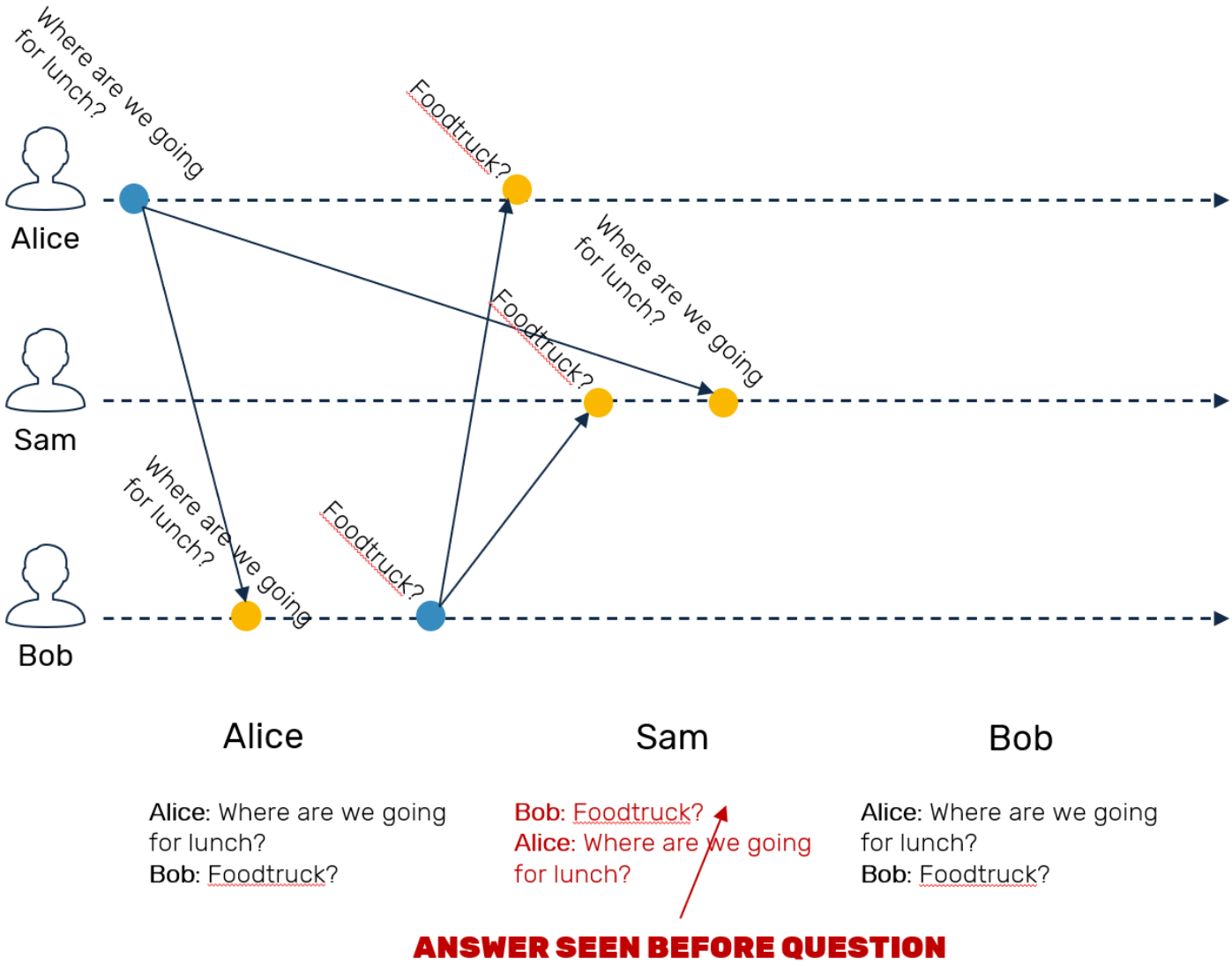
If we were thinking about usual order of events as a single linear timeline, we could visualize partial order as something that resembles more a tree-like structure - think for example about git repo, where every concurrent update is a new branch and replication leads to branch merges.

To establish total order, a popular technique is to use timestamps (in form of date or some sequence numbers). But it's not enough for partial order, as such timestamp alone doesn't cover enough information to recognize events that happened concurrently. We'll use vector clocks for that.

Now, let's reimagine scenario from point 3:

- Alice is sending event A.
- A was replicated to Ben, but not to the Sam eg. because they got disconnected.
- After receiving A, Ben produced event B.
- Sam still can talk with Ben, so the replication works between these two. Now we need to ensure, that Sam won't observe B before A (which was produced by Alice).

We can imagine this case in form of a chat talk like one seen below:



Replication

As you can see, naive push notifications won't work here. If Ben would only send his own events to Sam, he would receive incomplete history of known preceding events. Now, there are two popular ways of dealing with this issue.

Hiding non-continuous segments of the event stream

Each peer replicates by pushing its events, but the receiver keep tracks of the holes (eg. if we received events with timestamps `{A:1,B:1}` and `{A:2,B:3}` we can infer that there was some event with `B:2` in between, which we missed). Now we mark all events that happened after missing event as invisible until it arrives.

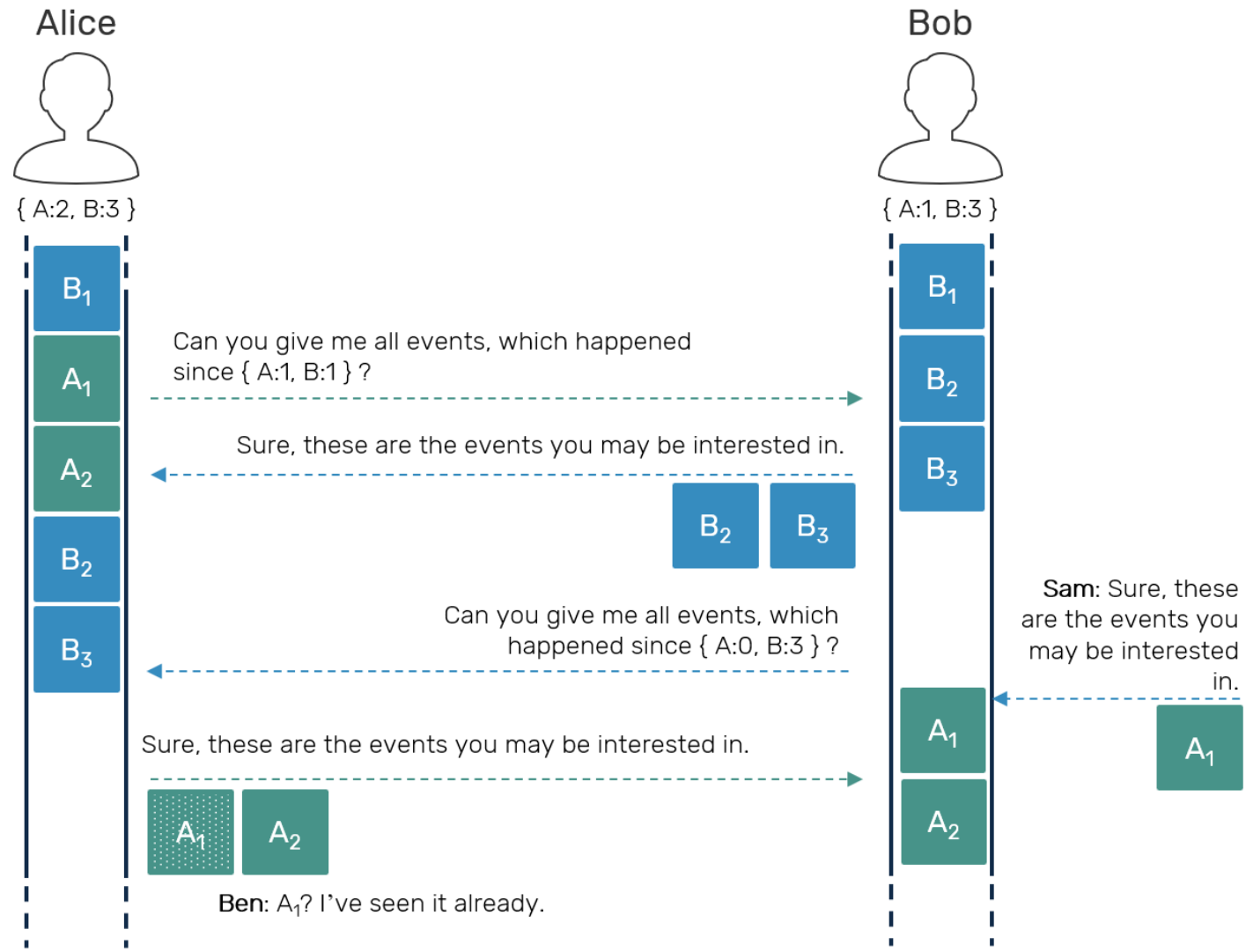
This can be problematic eg. when one peer went offline for long time before sending event to everyone else - in that case peers which didn't receive it, will have to keep invisible (and growing) backlog of events, possibly forever.

Allowing replication of other peer's events

Another way is to let peers replicate not only events produced by themselves, but also events emitted by other peers. Using example from above, while Sam is unable to receive event A from Alice (cause they are disconnected) he still can receive it together with B from Ben. This way be can prevent "holes" in event stream timeline from happening.

Now this means, that we need to deal with duplicates - a lot of duplicates. Given `N` peers, each one sending events of all others, we get `(N-1)^2` messages flying over the network. How can we improve that? Let's try to reverse the direction: first, instead of sender pushing the notifications to all receivers, we'll make each receiver pull events from the sender.

Pull-based model alone is not enough to fix the issue with sending duplicates though. This is why we'll add one more thing: when sending pull request, we'll send a timestamp describing all events, which pulling peer observed so far. This way we'll give sender a hint about which events on its stream could be skipped (because we've already seen them). Since this alone is still not guarantee for avoiding duplicates - as replication with multiple peers usually happens concurrently - receiver still needs to detect and skip potential duplicates on its own side as well.



So, while we didn't fully removed the issue, we amortized it to be much less of a problem.

Implementation

We're going into deep dive o a sample implementation, however we're not going to cover all required code here. **Please use this snippet if you're get lost**.

As usual, when it comes to an actual implementation, things are getting complicated. Example: *We were talking about replicating events by having a destination pull first with its vector clock to let the receiver filter events out on its side*. Sounds easy? So how are we actually going to do so? Are we going to go through entire event stream from the very beginning, read each one and determine if its should be replicated or not? This request will be repeated frequently, while the stream itself could have possibly millions of events.

Most of the eventsourcing systems order their events using sequence numbers. It's simple - we keep a persisted counter, that we're going to increment each time, we're about to store a new event. While we can indeed use this idea here, there's one catch: we cannot have one counter shared by all replicas. This would require coordination between replicas, which was one of the things we wanted to avoid, for the reasons mentioned above.

Because of that we're going to:

1. Let each replica keep it's own unique counter to producing sequence numbers.
2. Store a separate map, that will keep track of the sequence numbers we've replicated from others so far.

This way, when sending a replication request, we could simply say: *please give me all events that starts from your sequence number X that are greater or concurrent with a my vector clock Y*. With this at hand we can represent our events as:

```
type Event<'e> =
    { Origin: ReplicaId
      OriginSeqNr: uint64
      LocalSeqNr: uint64
      Version: VTime
      Data: 'e }
```

while our replica state could look like this:

```
type ReplicationState<'s> =
    { Id: ReplicaId // unique ID of replica
      SeqNr: uint64 // counter used to assign local sequence numbers
      Version: VTime // the highest timestamp observed so far
      Observed: Map<ReplicaId, uint64> // remote sequence numbers per replica
      Crdt: 's } // actual state of our CRDT
```

As you may remember we wanted to know when another replica has seen the state. We can do so by comparing last sequence number of observed replica we got so far and the currently observed vector clock of both incoming event and our own replication state:

```
[<RequireQualifiedAccess>]
module ReplicationState =

    /// Checks if current event has NOT been observed by a replica identified by state
    let unseen (state: ReplicationState<'s>) (e: Event<'e>) =
        match Map.tryFind e.Origin state.Observed with
        | Some ver when e.OriginSeqNr > ver -> true // event seqNr is higher than observed one
        | _ ->
          let cmp = Version.compare e.Version state.Version
          cmp = Ord.Gt || cmp = Ord.Cc
```

CRDT API

In order to make this design work with CRDT in the generic manner, we're also going to need some abstraction to unify the necessary capabilities of our CRDTs. In literature these are quite well defined.

Moreover, once you read through you'll see that operation-based CRDT's share many traits with eventsourced aggregate roots:

- We're going to need some way to obtain a **default**/empty/identity value of CRDT. It's quite understandable - as many peers may potentially start updating the same state without knowing or communicating with each other, there's no single point in time when we could "initialize" our object. For this reason we need to have some well-known default.
- We need to be able to **query** the state of a CRDT. Sometimes the CRDT itself consist not only of the actual state, user is interested about, but also some metadata used for conflict resolution. This was extremely common in state-based CRDTs discussed in the past on this blog. Maybe we'll see how to split the metadata from the state completely in the future with so called pure operation-based CRDTs, but for this protocol we also have user-state and metadata combined from time to time.
- Another definition is a **prepare** method, which is almost identical to a command handler from eventsourcing methodology. We basically take some operation send by the user, and change it into event (potentially enriched with CRDT metadata).
- The last method is an **effect** (this time analogous to event handler), which will be applied upon our CRDTs whenever event - either created locally or replicated from remote replicas - arrives.

All of these would be great example to create beautiful hierarchy of higher kinded types, however since we don't have these in F#, an ordinary interfaces must suffice:

```
[<Interface>]
type Crdt<'crdt,'state,'cmd,'event> =
    abstract Default: 'crdt
    abstract Query: 'crdt -> 'state
    abstract Prepare: state:'crdt * command:'cmd -> 'event
    abstract Effect: state:'crdt * event:Event<'event> -> 'crdt
```

Store

We're also going to need a store itself. Here, our requirements are not really very constraining:

```
[<Interface>]
type Db =
    /// Persist provided state snapshot.
    abstract SaveSnapshot: state:'s -> Async<unit>
    /// Load the latest known state snapshot if there's any.
    abstract LoadSnapshot: unit -> Async<'s option>
    /// Load ordered events with sequence number,
    /// that's equal or greater than provided one.
    abstract LoadEvents: startSeqNr:uint64 -> AsyncSeq<Event<'e>>
    /// Persist events, maintaining their sequence number ordering.
    abstract SaveEvents: events:Event<'e> seq -> Async<unit>
```

As you may imply from this declaration, we won't cover data deletion nor serialization, as these are outside the scope of this section.

You may notice, as we originally mentioned, that we're going to filter replicated events by vector clock, yet none is provided here. The reason for that is simple: so far I didn't found a database engine, that's able to natively understand the concept of version clocks and take advantage of them. We need to do that on application level. For this reason we also track ordering by sequence number, as this concept is well understood by most databases.

Local state recovery

Here, a replicator is our actor responsible for managing the state of our CRDT, its event stream and process of replicating event log with other replicators.

Before replicator will be able to serve any incoming requests, first it must recover its state from a persistent store. This can work pretty much like a standard eventsourced state recovery: we first try to read the latest snapshot and then replay all of the events that have happened since the snapshot was made. Local sequence number can be used to order snapshots and events in linear sequence.

We start by trying to load a last saved snapshot of the state, if such existed - otherwise we'll use the default instance of our CRDT. Next we're going to iterate over all locally persisted events that happened since that snapshot - this implies that state itself contains a corresponding sequence number at the time it was made - and apply them to our state.

```
async {
    // load state from DB snapshot or create a new empty one
    let! snapshot = db.LoadSnapshot()
    let mutable state =
        snapshot |> Option.defaultValue (ReplicationState.create id crdt.Default)
    // apply all events that happened since snapshot has been made
    for event in db.LoadEvents (state.SeqNr + 1UL) do
        state <- { state with
                       Crdt = crdt.Effect(state.Crdt, event)
                       SeqNr = max state.SeqNr event.LocalSeqNr
                       Version = Version.merge event.Version state.Version
                       Observed = Map.add event.Origin event.OriginSeqNr state.Observed }

    ctx.Self <! Loaded state
} |> Async.Start

recovering db ctx
```

All of that is pretty standard way of recovering state of aggregates in eventsourced systems. However as you may see in the snippet above our state consists of more than just simple user-defined data:

- `Crdt` is the actual replicated data type with user info, that we care about, sometimes enriched with CRDT-specific metadata. This piece is specialized depending on what CRDT our replicator is

responsible for.

- `LocalSeqNr` is the sequence number in a current replica. It's going to be incremented with every **persisted** event - no matter if it was replicated from remote replica or created locally.
- `Version` is a vector version that describes the most recent timestamp. We're also going to increment its segment (responsible for current replica), but only for events created locally.
- `Observed` is a map that describes pairs of `replicaId → originSeqNr` of all known replicas. We use it, so that when we're about to start/resume replication with other remote replicas, as continuation point in their event log.

You might ask: *if both `Version` and `Observed` fields can be represented as `Map<ReplicaId, uint64>`, so how do they differ*? `Version` counters are incremented ONLY when related replica is an origin (original creator) of an event, and only when a new event was created, not replicated. `Observed` map tracks sequence numbers. Each event log increments it's own sequence number every time it persist an event - it doesn't matter if that event was created locally or replicated from remote node. `Observed` map keeps track, up to which sequence number (relative to each replica) is current node synchronized.

Changing request into events

In eventsourcing, we have two types of messages. We already covered events - persisted, immutable and undeniable facts describing what already happened. Another kind are commands - these are user-made requests, that should result in a production of event: user-defined state should only change in result of events, not commands.

Of course, since our `state` parameter contains also some metadata - like `SeqNr` and `Version` which should be incremented prior to generation of new event - we need to update at least system-specific part of that state.

```
let replicator crdt db id (ctx: Actor<Protocol<_,_,_>>) =
  let rec active db state replicatingNodes = actor {
    match! ctx.Receive() with
    | Command(cmd) ->
      let requester = ctx.Sender()
      let seqNr = state.SeqNr + 1UL
      let version = Version.inc state.Id state.Version
      // generate user-event as result of a command
      let data = crdt.Prepare(state.Crdt, cmd)
      let event = { Origin = state.Id; OriginSeqNr = seqNr; LocalSeqNr = seqNr; Version = vers
      do! db.SaveEvents [event]
      // apply event to a state
      let ncrdt = crdt.Effect(state.Crdt, event)
      let nstate = { state with Version = version; SeqNr = seqNr; Crdt = ncrdt }
      requester <! crdt.Query ncrdt
      return! active db { nstate with IsDirty = true } replicatingNodes ctx
```

```
    // ... other cases
  }
```

After whole operation succeeded - just for convenience - we're going to send to requester a new, modified state view.

Now, we should discuss how to replay/sync events coming from a remote event log.

Remote replica synchronization

While we replayed local state via `AsyncSeq` - which C# programmers may know better as `IAsyncEnumerable` and others as streams - for remote event replication we'll use different approach. Replicating events one by one - either be it from local file storage or over the network - is not the best option, especially when events themselves are quite small. We need to pay extra toll for every network request made. Much better approach is to do it in batches.

We start by connecting one replica to another:

```
let replicator crdt db id (ctx: Actor<Protocol<_,_,_>>) =
  let rec active db state replicatingNodes = actor {
    match! ctx.Receive() with
    | Connect(nodeId, endpoint) ->
      let seqNr = Map.tryFind nodeId state.Observed |> Option.defaultValue 0UL
      endpoint <! Replicate(seqNr+1UL, 100, state.Version, ctx.Self)
      let timeout = ctx.Schedule recoverTimeout ctx.Self (RecoverTimeout nodeId)
      return! active db state (Map.add nodeId { Endpoint = endpoint; Timeout = timeout } repli
    // ... other cases
  }
```

Given the replica node identifier and its access endpoint, we can check what was the most recent sequence number of that replica's event log, we've seen so far: this is why `Observed` property exists in our replica state. Then we can use it - together with vector version describing our observed timeline. We do so to avoid problem of replicating N^2 events, as mentioned above. Finally we want to keep track of the nodes we called - for this reason we use `replicatingNodes` parameter. The reason why we have separate `state.Observed` and `replicatedNodes` is that first one is persisted, while later represents transient state (like timers).

Now, what happens when remote replica receives `Replicate` request?:

```
let replay nodeId (filter: VTime) target seqNr count = async {
  use cursor = db.LoadEvents(seqNr).GetEnumerator()
  let buf = ResizeArray()
  let mutable cont = count > 0
  let mutable i = 0
  let mutable lastSeqNr = 0UL
```

```
    while cont do
      match! cursor.MoveNext() with
      | Some e ->
        if Version.compare e.Version filter > Ord.Eq then
          buf.Add(e)
          i <- i + 1
        cont <- i < count
        lastSeqNr <- Math.Max(lastSeqNr, e.LocalSeqNr)
      | _ -> cont <- false
    let events = buf.ToArray()
    target <! Replicated(nodeId, lastSeqNr, events)
  }
```

Just like in case of local state recovery, we're going to iterate through events, starting from a given sequence number. However, we're going to batch these elements up to specified capacity. We're also not including events, which `Version` was less or equal to the version vector of a remote replica.

Additionally we also want to remember and return the last visited sequence number. Why don't just infer it straight from passed events? It's possible that every event present in current event log was already seen by our caller (its vector version doesn't pass the filter), therefore making batch empty. In that case we want to give it last visited sequence number anyway, simply so that when the next `Replicate` request arrives in the near future, we won't iterate over all of these already observed events again.

```
  let replicator crdt db id (ctx: Actor<Protocol<_,_,_>>) =
    let rec active db state replicatingNodes = actor {
      match! ctx.Receive() with
      | Replicated(nodeId, lastSeqNr, events) ->
        let mutable nstate = state
        let mutable remoteSeqNr = Map.tryFind nodeId nstate.Observed |> Option.defaultValue 0UL
        let toSave = ResizeArray()
        for e in events |> Array.filter (ReplicationState.unseen nodeId state) do
          let seqNr = nstate.SeqNr + 1UL
          let version = Version.merge nstate.Version e.Version
          remoteSeqNr <- max remoteSeqNr e.LocalSeqNr
          let nevent = { e with LocalSeqNr = seqNr }
          nstate <- { nstate with
                        Crdt = crdt.Effect(nstate.Crdt, nevent)
                        SeqNr = seqNr
                        Version = version
                        Observed = Map.add nodeId remoteSeqNr nstate.Observed }
          toSave.Add nevent
        do! db.SaveEvents toSave
        let target = Map.find nodeId replicatingNodes
        target.Endpoint <! Replicate(lastSeqNr+1UL, 100, nstate.Version, ctx.Self)
        let prog = refreshTimeouts nodeId replicatingNodes ctx
        return! active db { nstate with IsDirty = true } prog ctx
```

```
    // ... other cases
}
```

At this point we should already have some intuition about handling the `Replicated` event itself. Key points are:

- `LocalSeqNr` is relative to local replica sequence number. Therefore it must be replaced with incremented local sequencer value before persisting.
- Don't blindly persist all of the incoming events, make sure they were not seen yet on the receiver side. Sure, we already filtered them once on the sender side, but since we can be replicating with many senders in parallel over unpredictable network, some duplicates are still possible.

Wrapping it all together

Again, feel free to use this gist with entire code, as not all of it is present here. We covered only (I hope) the most confusing parts. If you managed to keep up, congratulations! Our reliable causal broadcast replication protocol is ready. We're now prepared to start using it to build our first operation-based CRDTs :D

Since this article is already quite long, at the moment we'll only cover the most basic one - increment/decrement counter. We're going to follow to more advanced structures in the future.

```
[<RequireQualifiedAccess>]
module Counter =

    let private crdt =
        { new Crdt<int64,int64,int64,int64> with
            member _.Default = 0L
            member _.Query crdt = crdt
            member _.Prepare(_, op) = op
            member _.Effect(counter, e) = counter + e.Data }

    /// Create replication endpoint handling operation-based Counter protocol.
    let props db replica ctx = replicator crdt db replica ctx
    /// Add value (positive or negative) to a counter replica.
    let add (value: int64) (replica) : Async<int64> = replica <? Command by
    /// Get the current state of the counter from its replica.
    let query (replica) : Async<int64> = replica <? Query
```

As you may see, the `crdt` instance here is extremely trivial. If you're disappointed you shouldn't. The reason why we've build so complex protocol was exactly to prepare a convenient platform for further work.

As you may see, unlike the state-based CRDT counter - which was backed by an underlying map(s) - this one is represented by one simple numeric value. Additionally command and event in this case are

indistinguishable: we're going to keep them however for the future reference, as more advanced CRDTs may differentiate between the two.

References

- Eventuate is a JVM Akka plugin that offers replicated event logs and operation-based CRDTs, and it was main inspiration for this section.
- Slides from the presentation about leaderless peer-to-peer evensourcing I did a while ago. They brings more visual step-by-step explanations for some of the stuff we talked about here and hints for our future direction.
- AntidoteDB is another (Erlang) implementation of operation-based CRDTs, wrapped in a context of a geo-replicated database.

Operation-based CRDTs: registers and sets

Last time we started our operation-based CRDTs sub-series, as we moved away from state-based CRDTs. We talked mostly about core requirements and sample implementation of RCB (Reliable Causal Broadcast) protocol, which was necessary to provide guarantees required by Commutative Replicated Data Types.

This time we're going to implement most common data types - registers and sets - in their operation-based variant.

Before we begin...

We already defined the core abstractions of our data types to be implemented. You can get a quick recap of hows & whys under this link. Since this is a continuation, I highly encourage you to get familiar with previous chapter, as otherwise you may feel lost. Here we're only remind the crucial part - the `Crdt` interface definition, which gives a shared structure for our data types:

```
type Crdt<'crdt,'state,'cmd,'event> =
    /// Initial CRDT instance.
    abstract Default: 'crdt
    /// Read CRDT value as user-friendly data type.
    abstract Query: 'crdt -> 'state
    /// Handle user request and produce a serializable event out of it.
    abstract Prepare: state:'crdt * command:'cmd -> 'event
    /// Handle event (either produced locally or replicated
    /// from remote source) to modify the CRDT state.
    abstract Effect: state:'crdt * event:Event<'event> -> 'crdt
```

You already could have seen an example Counter implementation in action. Now, let's go to more advanced structures.

Registers

Registers are one the most wide spread ways of working with CRDTs. The reason is simple: they allow us to wrap any sort of ordinary data types in order to give them CRDT-compliant semantics.

However, this comes at the price - we cannot simply change any non-CRDT value into CRDT without some compromises: if that would be possible, we wouldn't need CRDTs in the first place. For these, two popular approaches are known as **last-write-wins** and **multi-value** registers.

Last Write Wins Register

Last write wins is a popular way of dealing with conflicts being the result of concurrent updates in many systems (Cassandra is good example of a database that's quite known from this approach).

The algorithm - already mentioned in the past - is simple: on each register value update, we timestamp it. If our current timestamp is higher than the most recent one remembered by the register itself, we change the value, otherwise leave existing one (more recent) untouched.

What is our timestamp? The intuition says, that we could simply use current system clock (eg. `DateTime.UtcNow` in .NET API). However:

- We mentioned in the past that relying on a system clock to provide monotonically incrementing value can be naive assumption. While we're going to use it here for simplicity of implementation, I encourage you to look for other solutions (eg. Hybrid Logical Clocks).
- You never can be sure that reading a clock value concurrently on two different machines will always yield different results. While the risk is low, it's still possible to run into a situation when we have two different values sharing the same timestamp, exposing the integrity of our data in the result. For this reason we'll extend our timestamp to be a composite value: system clock + ID of the current replica. Since replica identifier is unique, it will ensure the same for our timestamp.

Given that we already prepared an API for building our CRDTs, this implementation is quite straightforward:

```
[<RequireQualifiedAccess>]
module LWWRegister

// this type is structuraly comparable: first compare dates,
// and if they're equal use replica ids
type Timestamp = (DateTime * ReplicaId)

type LWWRegister<'a> =
  { Timestamp: Timestamp
    Value: 'a voption }

type Operation<'a> = DateTime * 'a voption

type Endpoint<'a> = Endpoint<LWWRegister<'a>, 'a voption, Operation<'a>>
```

```
let private crdt : Crdt<LWWRegister<'a>, 'a voption, 'a voption, Operation<'a>> =
    { new Crdt<_,_,_,_> with
        // default register value - use lowest possible timestamp
        // to make sure it will always loose with incoming update
        member _.Default = { Timestamp = (DateTime.MinValue, ""); Value = ValueNone }
        // return register value alone
        member _.Query crdt = crdt.Value
        // produce an event data to be persisted/replicated,
        // timestamp it with clock only, as we get replica id
        // from the wrapping Event type
        member _.Prepare(_, value) = (DateTime.UtcNow, value)
        member _.Effect(existing, e) =
          // decompose our event data we created in Prepare method
          let (at, value) = e.Data
          let timestamp = (at, e.Origin) // create timestamp
          if existing.Timestamp < timestamp then
            // if given timestamp is more recent, return updated register
            { Timestamp = timestamp; Value = value }
          else existing }

let props db replica ctx = replicator crdt db replica ctx
let updte (value: 'a voption) (ref: Endpoint<'a>) : Async<'a voption> = ref <? Command value
let query (ref: Endpoint<'a>) : Async<'a voption> = ref <? Query
```

One of the issues of last-write-wins approach is its inherent risk of data loss - we took an easy to use API at price of automatically throwing out potentially useful data.

Multi Value Register

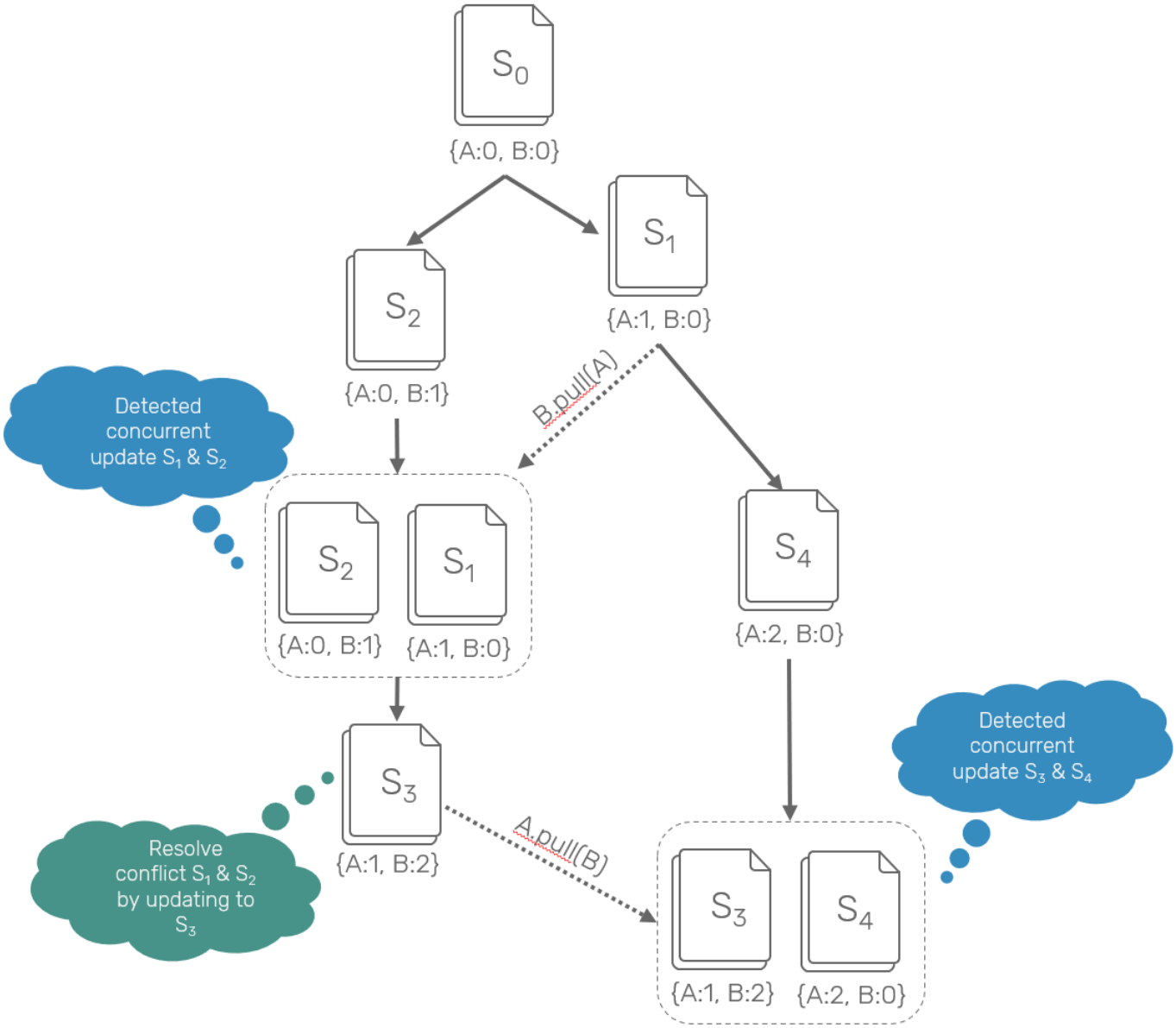
Another approach is to build a register that doesn't drop any data, instead it keeps track of all causal (happened-before) relationships between updates and in case of conflicts returns all conflicting cases. It's known as Multi Value Register.

The common analogy we could use here to better understand this register's behavior would be a Git interactive merge resolution: when two people edit the same block of text and they *git pull* changes from each other, what we often see is information about unresolved merge conflicts. What can we expect then?:

1. We're going to see not one, but both conflicting block updates, even when they relate to the same line of text.
2. Once we resolve this merge conflict manually, we expect that all other people, who pulled our changes onward will be able to read value from our merge commit, so that they don't need to resolve the same conflicts again by themselves.

The same intuition applies to multi-value registers. We track causality of register updates, and in case of conflicts - as when two independent updates are happening concurrently - we'll provide all

conflicting values. Programmer is then free to choose any value and override concurrent conflicts with more recent update. We can imagine that in the following scenario:



Now, let's analyse what's happening over here:

*1. We consider two actors: Alice and Bob. They both start with empty register at state `S0`.

2. Independently and without knowing about each others intentions, Alice modifies register to state `S1`, while Bob does the same to state `S2`.
3. Bob pulls changes from Alice and discovers that she also has changed the document: he sees both changes (`S1` and `S2`). Now, he decided to resolve these to a single value of `S3`.
4. While Bob tried to reconcile the conflict, Alice has overriden her state again into `S4`.
5. Now Alice is pulling Bob's changes, and notices that there's another conflict (`S3` vs `S4`). What's important here is that she is not aware of the previous one (`S1` vs `S2`) ever happening as it has been already solved by Bob.*

All of that is possible thanks to version vectors (eg. `{A:1,B:2}`), which inform us about the relationships between the corresponding changes: whether they happened one after another or have

they happened concurrently with each other.

How to represent that in code? We'll simply keep list of all concurrent values, each of them tagged with a version vector describing their causal relationships.

```
[<RequireQualifiedAccess>]
module MVRegister

// in order to make our register "clearable" we use an option type
// as attached vector version must be still kept to ensure that
// no previous value will override our 'clear' update
type MVRegister<'a> = (VTime * 'a voption) list

type Endpoint<'a> = Endpoint<MVRegister<'a>, 'a voption, 'a voption>

let private crdt : Crdt<MVRegister<'a>, 'a list, 'a voption, 'a voption> =
    { new Crdt<_,_,_,_> with
        // at the begining our register is empty
        member _.Default = []
        // return list of (possibly conflicting) values alone
        member _.Query crdt =
          crdt
          |> List.choose (function (_, ValueSome v) -> Some v | _ -> None)
        // we don't need to do anything here, as version vector
        // will be attached inside of Event type
        member _.Prepare(_, value) = value
        // prune all values, retaining only those which were updated
        // concurrently to a current event
        member _.Effect(existing, e) =
          let concurrent =
            existing
            |> List.filter (fun (vt, _) -> Version.compare vt e.Version = Ord.Cc)
          (e.Version, e.Data)::concurrent }

let props db replica ctx = replicator crdt db replica ctx
let updte (value: 'a voption) (ref: Endpoint<'a>) : Async<'a voption> = ref <? Command value
let query (ref: Endpoint<'a>) : Async<'a voption> = ref <? Query
```

You may see, that in `Effect` method we're only going to keep around values with concurrent version vectors. However comparison of these can yield one of 4 different outputs:

- `Ord.Lt` means that we can discard the value as it has been overridden (merge resolved in Git terms) by current update.
- `Ord.Eq` is not possible, as version vectors are globally unique and no previously existing value could have the same one as current one - we don't worry about duplicates as our Reliable Causal Broadcast protocol used for event replication keeps them in check.
- `Ord.Gt` is also not possible - again this is an advantage of using RCB replication protocol, which makes sure that events which are strictly greater will never be processed first.

- `Ord.Cc` (marking a concurrent update) is the only remaining option in this situation. It represents unresolved conflicts.

With these new registers in our arsenal, let's move to defining our first collection type.

Sets

Now, we're going to cover how to build the most basic operation-based CRDT collection: a set. In the past we already covered a huge amount of theory - spanned over 2 blog posts (1, 2) - needed to implement state-based sets efficiently. Thankfully, operation-based variant is much simpler.

First the representation of the set itself consists only of pairs of values and their corresponding timestamps (here represented as version vector type `VTime`), which tell us when given value was added to set.

```
type ORSet<'a> when 'a: comparison = Set<'a * VTime>
```

Keep in mind that the same value could have been added multiple times - here for simplicity reasons (preferrably this should be a `Map<'a, Set<VTime>>`), we'll just keep them separately and deduplicate them when user requests for the state. Luckily inherent set property (uniqueness of elements) will take care of it for us:

```
let private crdt : Crdt<ORSet<'a>, Set<'a>, Command<'a>, Operation<'a>> =
  { new Crdt<_,_,_,_> with
      member _.Query(orset) = orset |> Set.map fst
      // other methods ...
  }
```

Next, we need to support two most common operations, without which our set wouldn't be very useful: addition and removal. We can represent those as following commands (user requests) and events (actual data to be stored and replicated):

```
type Command<'a> =
  | Add of 'a
  | Remove of 'a

type Operation<'a> =
  | Added of 'a
  | Removed of Set<VTime>
```

You may notice that these corresponding types are not the same - `Removed` event doesn't contain information about removed element, instead it uses a collection of vector versions. Why?

1. As events are often serialized and pushed through various I/O boundaries (either on disk or through the network), we don't want to put potentially heavy user data together with them when not necessary.
2. In order to safely remove element we need to establish the causality relations between `Removed` event and potential additions that may have happened concurrently. For this reason we need to keep version vectors around - each `VTime` allows us to uniquely identify corresponding `Added` operation so that later on, when we're about to remove an element, we don't by accident delete the same elements that were added again concurrently to our removal.

Now let's split handling of additions and removals. Let's go with addition first:

```
let private crdt : Crdt<ORSet<'a>, Set<'a>, Command<'a>, Operation<'a>> =
  { new Crdt<_,_,_,_> with
      member _.Prepare(orset, cmd) =
        match cmd with
        | Add item -> Added item
        | Remove item -> // ...
      member _.Effect(orset, e) =
        match e.Data with
        | Added(item) -> Set.add (item, e.Version) orset
        | Removed(versions) -> // ...
      // other methods ...
  }
```

Command handler is simply reassigning an item from command into event data type. Applying the event itself attaches timestamp, which was generated by underlying replication protocol to uniquely identify that event. *PS: ability to elevate timestamps generation into underlying infrastructure simplifies a lot of things in operation-based CRDTs.*

Now, onto removals. Here the code is a bit more complicated. Preparing an event requires us to gather all vector versions of the element to be removed. Once this is done, we're ready to emit our event:

```
let private crdt : Crdt<ORSet<'a>, Set<'a>, Command<'a>, Operation<'a>> =
  { new Crdt<_,_,_,_> with
      member _.Prepare(orset, cmd) =
        match cmd with
        | Add item -> //...
        | Remove item ->
          let timestamps =
            orset
            |> Set.filter (fun (i, _) -> i = item)
            |> Set.map snd
          Removed timestamps
      // other methods ...
  }
```

When we're about to apply our events, we again go over our set looking for all entries, which have timestamps matching the one from passed element itself, and get rid of them:

```
let private crdt : Crdt<ORSet<'a>, Set<'a>, Command<'a>, Operation<'a>> =
  { new Crdt<_,_,_,_> with
      member _.Effect(orset, e) =
        match e.Data with
        | Added(item) -> // ...
        | Removed(versions) ->
          orset |> Set.filter (fun (_, ts) -> not (Set.contains ts versions))
      // other methods ...
  }
```

This way we're only truly remove element, if it wasn't added concurrently on other nodes. This means that **our set maintains Add Wins semantics** (in case when the same element was both inserted and removed on two nodes at the same time, it will remain in the set).

Operation-based CRDTs: arrays

In this post, we'll continue onto topic of Commutative Replicated Data Types. We already mentioned how to prepare first, the most basic types of collections: sets. This time we'll go take a look at indexed sequences with add/remove operations.

Now, why I said that indexed sequences are more complex than sets? It doesn't reflect our intuition about ordinary data types, where usually set implementations are more complex than eg. arrays. Here it's all about guarantees: unlike sets, indexed sequences come with a promise of maintaining order of inserted items, which is much harder to maintain when we take into account that multiple actors may choose to insert/remove elements in such collection in disjoined, concurrent fashion.

Maintaining order is very useful in many places - like building queues/stacks being one of the more obvious - however there's one domain, which brags especially loud about it. That is text operations & collaborative text editors.

> We're talking about fully decentralized collaboration, without any need of collaborators being online in order to perform updates or having a centralized server to resolve conflicting updates (see: Google Docs).

This area has been researched heavily over many years - many of the data structures created there predates term Conflict-free Replicated Data Types by years. Here we'll only mention two:

- **LSeq** - Linear Sequences
- **RGA** - Replicated Growable Arrays

PS: *there's another (older and perhaps even more popular) approach to text collaboration popularized by Google Docs, Etherpad, xi etc. called operational transformation. Nowadays it's often a preferred*

choice because of maturity and a speed of its implementations. But the times are changing...

Problem description

Our scenarios here will work in context of inserting/deleting characters from some kind of text document, as this way seems to be the most well recognized and easy to visualize.

A key observation of most (if not all) CRDT approaches to ordered sequences is simple: **we need to be able to track the individual elements as other elements in collection come and go**. For this reason we mark them with unique identifiers, which - unlike traditional array index, which can refer to different elements over time - will always stick to the same element. While in literature they don't have a single name, here I'll call them **virtual pointers**.

LSeq

In case of LSeq these indentifiers are represented as sequences of bytes, which can be ordered in lexical order .eg:

```
// example of lexically ordered sequence
a   => v1
ab  => v2
abc => v3
b   => v4
ba  => v5
```

Since no generator will ever guarantee that byte sequences produced on disconnected machines are unique 100% of the time, the good idea is to make virtual pointer a composite key of byte sequence together with unique replica identifier (using replica ID to guarantee uniqueness is quite common approach). Such structure could look like this:

```
[<Struct;CustomComparison;CustomEquality>]
type VPtr =
  { Sequence: byte[]; Id: ReplicaId }
  override this.ToString() =
    String.Join('.', this.Sequence) + ":" + string this.Id
  member this.CompareTo(other) =
    let len = min this.Sequence.Length other.Sequence.Length
    let mutable i = 0
    let mutable cmp = 0
    while cmp = 0 && i < len do
      cmp <- this.Sequence.[i].CompareTo other.Sequence.[i]
      i <- i + 1
    if cmp = 0 then
      // one of the sequences is subsequence of another one, compare their
      // lengths (cause maybe they're the same) then compare replica ids
      cmp <- this.Sequence.Length - other.Sequence.Length
```

```
        if cmp = 0 then this.Id.CompareTo other.Id else cmp
      else cmp
```

In order to insert an element at given index, in LSeq we first need to find virtual pointers of elements in adjacent indexes, then generate new pointer, which is lexically higher than its predecessor and lower than its successor (`lo.ptr < inserted.ptr < hi.ptr`).

There are many possible ways to implement such generator - the trivial one we're going to use is to simply compare both bounds byte by byte and insert the byte that's 1 higher than a corresponding byte of the lower bound, BUT only if it's smaller than corresponding byte of the higher bound:

```
let private generateSeq (lo: byte[]) (hi: byte[]) =
  let rec loop (acc: ResizeArray<byte>) i (lo: byte[]) (hi: byte[])  =
    let min = if i >= lo.Length then 0uy else lo.[i]
    let max = if i >= hi.Length then 255uy else hi.[i]
    if min + 1uy < max then
      acc.Add (min + 1uy)
      acc.ToArray()
    else
      acc.Add min
      loop acc (i+1) lo hi
  loop (ResizeArray (min lo.Length hi.Length)) 0 lo hi
```

If virtual pointers on both bounds are of different length we can replace missing byte by 0 and 255 respectivelly. If there's no space to put a free byte between both bounds, we simply extend the key by one byte, eg. we need to generate key between `[1]` and `[2]` → the resulting key will be `[1,1]` .

These boundary keys must only exists at the moment of creating a new event (so in prepare/command-handler phase), but are not necessary when we're about to apply it. This makes element deletion an incredibly simple procedure - all we need to do, is to just remove a corresponding entry from our collection.

```
let private crdt (replicaId: ReplicaId) : Crdt<LSeq<'a>, 'a[], Command<'a>, Operation<'a>> =
  { new Crdt<_,_,_,_> with
    member _.Effect(lseq, e) =
      match e.Data with
      | Inserted(ptr, value) ->
        let idx = binarySearch ptr lseq
        Array.insert idx (ptr, value) lseq
      | Removed(ptr) ->
        let idx = binarySearch ptr lseq
        Array.removeAt idx lseq
    // other methods...
  }
```

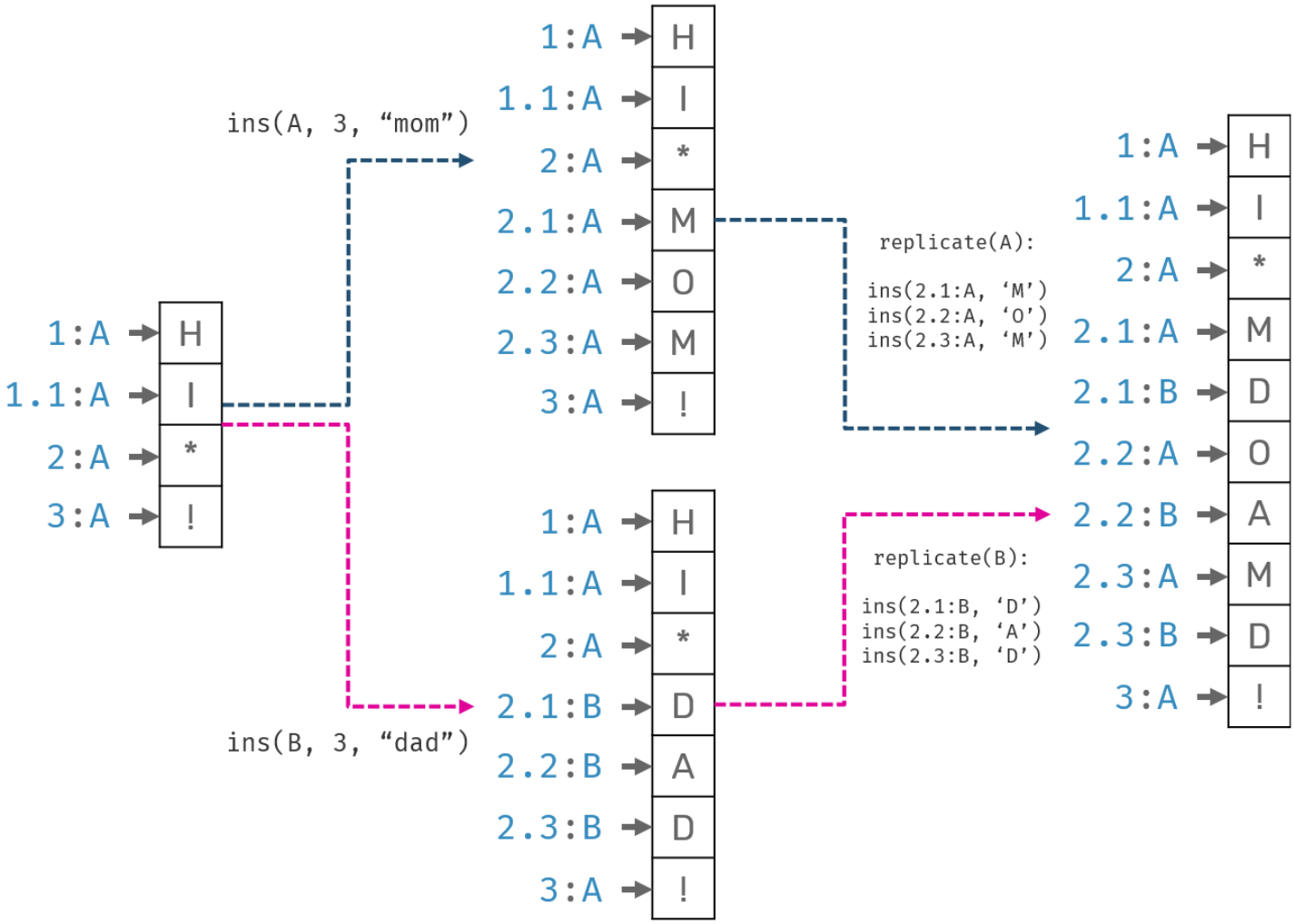
If you're interested to see it in details, the code we described in this section can be found here.

Interleaving

Every CRDT comes with its tradeoffs, and LSeq is no different. One issue of LSeq data type is how it deals with concurrent updates. Imagine following scenario:

Two people, Alice and Bob, want to collaborate over a text document. Initial document content was "hi !". While offline, Alice edited it to "hi mom!", while Bob to "hi dad!". Once they got online, their changes were synchronized. What's an acceptable state of the document after sync?

While original LSeq insertion algorithm didn't specify insertion of multiple elements we can replace it with series of single character inserts.



Now, as you see here, after replication document on both sides contains "hi mdoamd!". While technically correct, it's doubtfully a desired outcome: we may guess that mixed output would satisfy neither Alice nor Bob's expectations. We missed the intent: a correlation between inserted elements. This problem can be solved (or at least amortized) in theory: we just need to generate byte sequences in a "smart way", so that elements pushed by the same actor one after another will land next to each other after sync. Problem? So far no one presented such "smart" algorithm.

PS: *Based on the byte sequence generation algorithm you may even trace the order in which particular characters where inserted using only virtual pointers.*

RGA

A second data structure we're going to cover now is known as Replicated Growable Array (RGA for short). I'll be refering to code, which full snippet you can find here. It ofers slightly different approach: where LSeq virtual pointers where byte sequences, in RGA it's just a combination of a single monotonically increasing number and replica identifier:

```
type VPtr = uint64 * ReplicaId
```

Now, we'll use this structure to (permanently) tag our elements. These together will form a vertex:

```
type Vertex<'a> = VPtr * 'a option
```

As you may see, here is the catch - our element is of `option` type. Why? In case of LSeq, virtual pointers where generated to reflect a global lexical order that matches the insertion order. They were not related ot each other in any other way. In case of RGA, pointers are smaller and of fixed size, but they're not enough to describe insertion order alone. Instead insert event refers to a preceding virtual pointer. For this reason we cannot simply remove them, since we don't known if another user is using them as reference point for their own inserts at the moment. Instead of deleting permanently, **RGA puts tombstones in place of removed elements**.

```
type Command<'a> =
  | Insert of index:int * value:'a
  | RemoveAt of index:int

type Operation<'a> =
  | Inserted of predecessor:VPtr * ptr:VPtr * value:'a
  | Removed of ptr:Position
```

Advantage here is that - because RGA insert relates to previous element - the sequences of our own inserts will be linked together, preventing from interleaving with someone else's inserts.

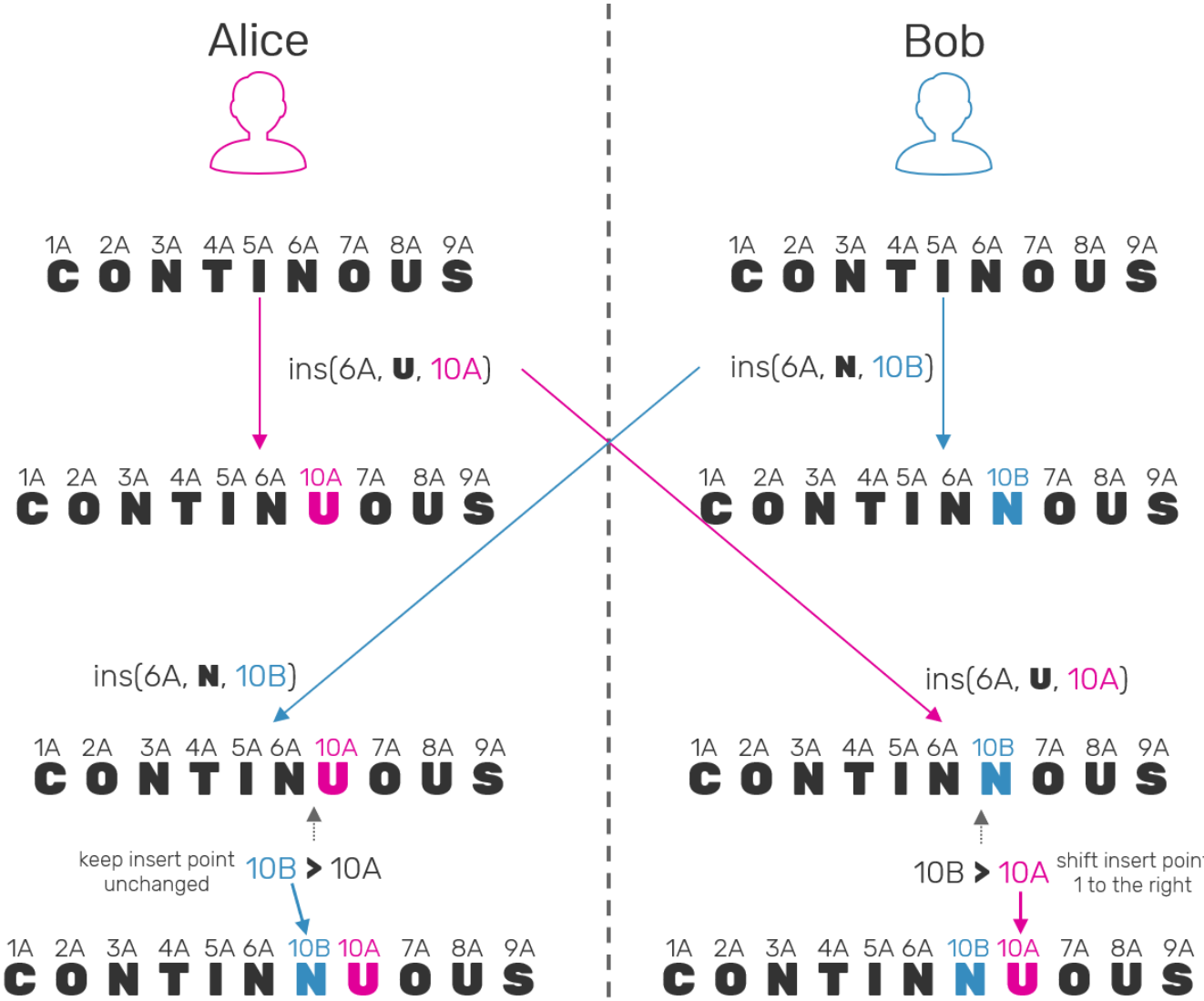
Simple question: how are we going to put first element, since there are no predecessors? When initiating an empty RGA, we can put an invisible header at its head as a starting point:

```
let private crdt (replicaId: ReplicaId) : Crdt<Rga<'a>, 'a[], Command<'a>, Operation<'a>> =
  { new Crdt<_,_,_,_> with
      member _.Default =
        let head = ((0,""), None)
        { Sequencer = (0,replicaId); Vertices = [| head |] }
    // other methods...
  }
```

PS: *in this implementation, we're using an ordinary array of vertices, but in practice it's possible to split metadata and contents into two different structures and/or use specialized tree-based data structures (like ropes) to make insert/remove operations more efficient*.

Now, how the insertion itself should look like? First we need to find an actual index, where a *predecessor* of inserted element can be found. Then just insert our element under next index. This sounds simple, but it misses one crucial edge case: *what if two actors will concurrently insert different elements after the same predecessor?*. The resulting collection after replication should present all elements in the exact same order. How could we achieve this?

Again we'll use our virtual pointers, and extend them with one important property - we'll make them ordered (first by sequence number, then replica ID), just like in the case of LSeq. Now the rule is: recursivelly, we shift our insertion index by one to the right every time when virtual pointer of the element living under that index already had a higher value than virtual pointer of inserted element.



Why does it even has any sense? Well, remember when generating new virtual pointer, we take a sequence number as the highest possible value? Also since our RGA is operation-based, we can count on the events being applied in partial order. Using example: when inserting element `C` after `A` inside of collection [A,B], we can be sure that `B` has virtual pointer that falls into one of two categories:

1. `B`s pointer has sequence number **lesser** than `C`'s which means, we can be sure that `B` was inserted before `C`, not concurrently. So the user must have knew about existence of `B` and still decided to insert `C` after `A`.
2. `B` pointer's sequence number is **equal or greater** than `C`s - that means that `B` has been inserted concurrently on another replica. In this case we also compare replica ID to ensure that virtual pointers can be either lesser or greater to each other (but never equal). This way even when having partially ordered log of events we can still guarantee, that elements inside of RGA itself will maintain a total order - the same order of elements on every replica.

> Another indirect advantage of having CRDT backed by a log of events is that we can travel over entire history of edited document and see its state at any point in time!

This check is performed recursively, and can be written as follows:

```
let rec private shift offset ptr (vertices: Vertex<'a>[]) =
  if offset >= vertices.Length then offset // append at the end
  else
    let (successor, _) = vertices.[offset]
    // F# creates structural comparison for tuples automatically
    if successor < ptr then offset
    else shift (offset+1) ptr vertices // move insertion point to the right
```

With this we can implement our insert operation as:

```
let private applyInserted (predecessor: VPtr) (ptr: VPtr) value rga =
  // find index where predecessor vertex can be found
  let predecessorIdx = indexOfVPtr predecessor rga.Vertices
  // adjust index where new vertex is to be inserted
  let insertIdx = shift (predecessorIdx+1) ptr rga.Vertices
  let newVertices = Array.insert insertIdx (ptr, Some value) rga.Vertices
  // update RGA to store the highest observed sequence number
  // (a.k.a. Lamport timestamp)
  let (seqNr, replicaId) = rga.Sequencer
  let nextSeqNr = (max (fst ptr) seqNr, replicaId)
  { Sequencer = nextSeqNr; Vertices = newVertices }
```

With inserts done, the hard part is now over. We can get to **removing elements at provided index** (which we already described above). It's extremelly simple in comparison:

```
let private applyRemoved ptr rga =
  // find index where removed vertex can be found and tombstone it
  let index = indexOfVPtr ptr rga.Vertices
  let (at, _) = rga.Vertices.[index]
  { rga with Vertices = Array.replace index (at, None) rga.Vertices }
```

All we really need, is to mark a particular vertex as tombstone (here done by clearing its content to `None`) and skip them over when materializing our CRDT to a user-facing data structure:

```
let private crdt replicaId : Crdt<Rga<'a>, 'a[], Command<'a>, Operation<'a>> =
  { new Crdt<_,_,_,_> with
      member _.Query rga = rga.Vertices |> Array.choose snd // skip empty vertices
      // other methods ...
  }
```

What's next?

We only scratched the surface here. The number of possible extra operations and optimizations that have been found here is trully astounding. We (probably) won't cover all but few of them on this blog.

1. As we mentioned earlier, in RGA approach we don't physically delete removed vertices. Instead we tombstone them. That could be wasteful in case of frequently used, delete heavy, long living collections. However **pruning of tombstones** is actually possible. We'll cover that up another time.
2. For specific cases like text editors, we'd like to insert entire range of elements (eg. characters) at once. This implementation doesn't allow us to do so. Imagine copy/pasting entire page of text, which would result in producing possibly thousand instantaneous insert operations. Again, it's possible to modify that implementation to make it work (it's called string-wise or **block-wise RGA**) and it's also in the plans for the future chapters.
3. RGA implementation adds a small metadata overhead to every inserted element (needed because of virtual pointer references). For small elements (eg. again, single characters) this may result in metadata constituting a majority of the entire payload. I can recommend you a presentation made by Martin Kleppmann about how this has been solved by using a custom formatting in the Automerge - a JavaScript library that allows us to use JSON-like datastructures with CRDT semantics. PS: *another known library in this space is Y.js, which also uses optimizations in that area*.

Block-wise Replicated Growable Array

Now it's the time to describe some possible modifications to original RGA algorithm - known as Block-wise Replicated Growable Array - that will let it fit better into scenarios, where we insert and delete entire ranges of elements.

A prerequisite here is for you to understand how RGA works: how to insert elements after predecessor, resolve conflict of concurrent inserts at the same position or tombstone removed elements.

As usually, here we'll support ourselves with most crucial snippets of the code, which full version can be found here.

Why do we even care?

One of the common use cases for indexed CRDT sequences is collaborative text editing. We already mentioned it - we'd like to be able to operate over a piece of document in peer-to-peer fashion (no central server), also offline if necessary. Algorithms presented in this blog so far would fit great to this purpose, but... there's a problem of scale.

Editing piece of text by multiple users means sending very small pieces of data (characters) with high frequency and volume. In traditional CRDTs this comes with two issues: a fairly high metadata overhead applied on each character and the cost of insertion of multiple adjacent elements. Metadata size can be amortized when stored on disk. What about in-memory size and computational cost?

Let's use an example: imagine that we're about to copy-paste huge piece of text into our collaborative text editor eg. few pages consisting of possibly thousands of characters. Using the approach we discussed so far this would be converted into thousands of separate insert operations, each with its own metadata.

Question is: *couldn't we just insert a block of text as a single element?* In theory, yes. In practice this would mean that **with existing approach our block of text would become one immutable piece of data**, that cannot be split (forget about inserting text in between pasted characters).

Below we're going to describe algorithm for insertion and deletion of multiple elements at once.

Block-wise Replicated Growable Arrays

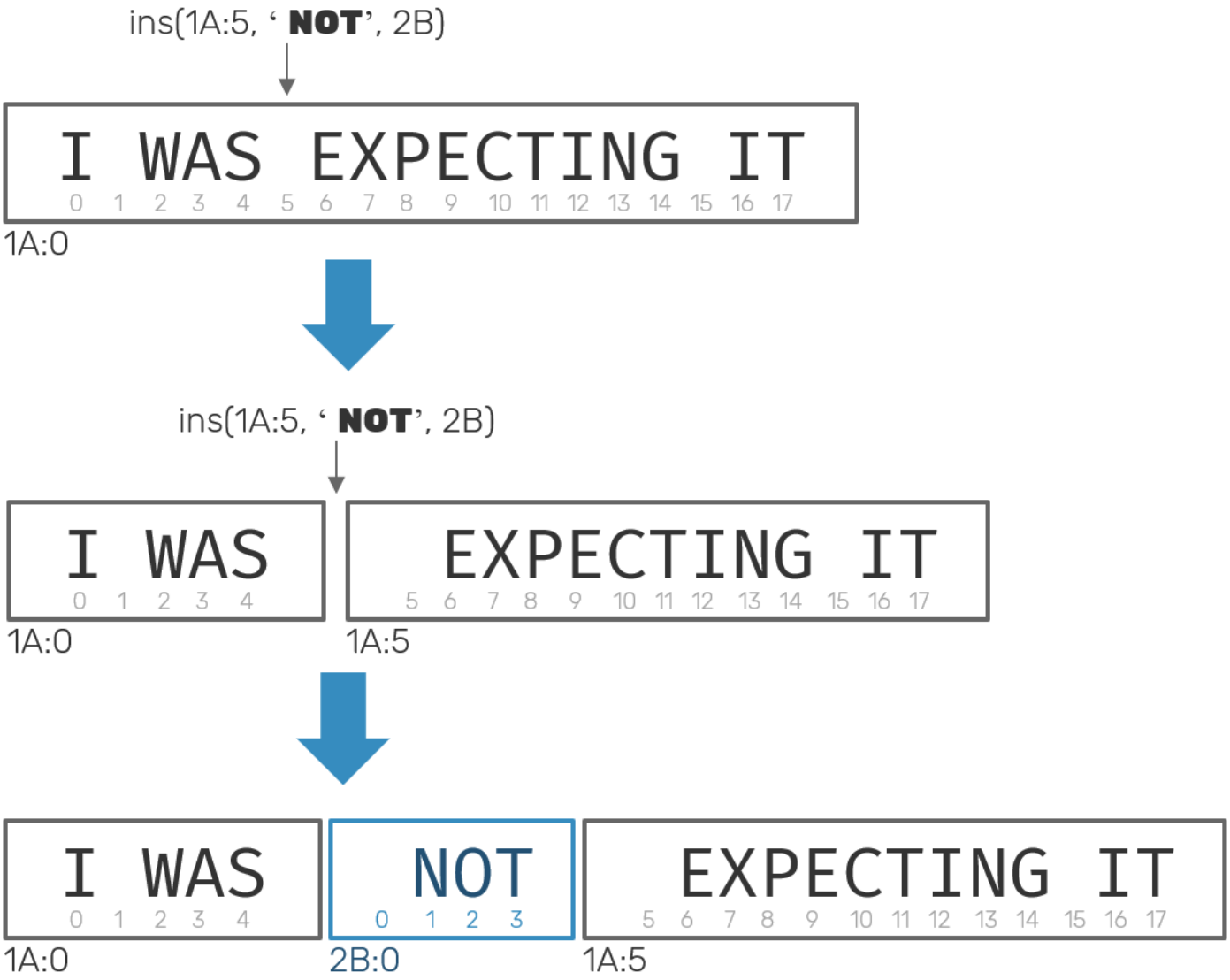
One of the core principles for indexed CRDTs was sticking a unique ID to every inserted element, that will can be used to track it for the end of times. This seems to cause an impossible to solve problem: **each block ID must be unique and persistent to that block**. Otherwise when a concurrent update arrives from remote peer we may no longer be able to tell to which position in current, locally-modified RGA it was referring to. For that we need to refer to absolute unique IDs. On the other side when we want to insert another block in the middle of existing one, it needs to be split in two - and **each of these needs unique ID**.

In the solution below, we'll identify each block by the virtual pointer (unique identifier assigned when block was inserted) together with initial offset from the beginning of inserted block.

```
type VPtr = (int * ReplicaId)
type VPtrOff = { Ptr: VPtr; Offset: int }

type Content<'a> =
  | Content of 'a[]
```

```
    | Tombstone of int // number of consecutive tombstoned items

type Block<'a> = { Ptr: VPtrOff; Data: Content<'a> }
type Rga<'a> = { Sequencer: VPtr; Blocks: Block<'a>[] }
```

When a new block is inserted, its offset is always 0. When we need to split it, we produce two blocks (left and right side of a split), both of which have the same virtual pointer, but with different offsets, where right one's is shifted by the length of the left one. This process can be imagined as:



Knowing that, we can represent splitting in following form:

```
module Block

let split index (block: Block<'a>) =
  // check if it's possible to slice the block
  if index = block.Length then (block, None)
  else
    let ptr = block.PtrOffset
    let (a, b) = block.Data.Slice index // slice the data in two
    let left = { block with Data = a }  // re-assign left slice to left block
```

```
    let rightPtr = { ptr with Offset = ptr.Offset + index }
    let right = { PtrOffset = rightPtr; Data = b }
    (left, Some right)
```

The next piece of our insertion algorithm is an ability to determine where the split index should be located. We already discussed how to map user-facing index into actual RGA position with regards to tombstones, but now we have to go deeper - we'll need not only position of the block but also index position within the block itself:

```
let findByPositionOffset (p: VPtrOff) blocks =
  let rec loop idx p (blocks: Block<'a>[]) =
    let block = blocks.[idx]
    if block.PtrOffset.Ptr = p.Ptr then
      if block |> Block.containsOffset p.Offset
      then
        // we found the correct block, now we need an index within that block
        let blockIdx = p.Offset - block.PtrOffset.Offset
        (idx, blockIdx)
      else loop (idx+1) p blocks
    else loop (idx+1) p blocks
  loop 0 p blocks
```

With all of these at hand insertion is similar to plain old RGA algorithm:

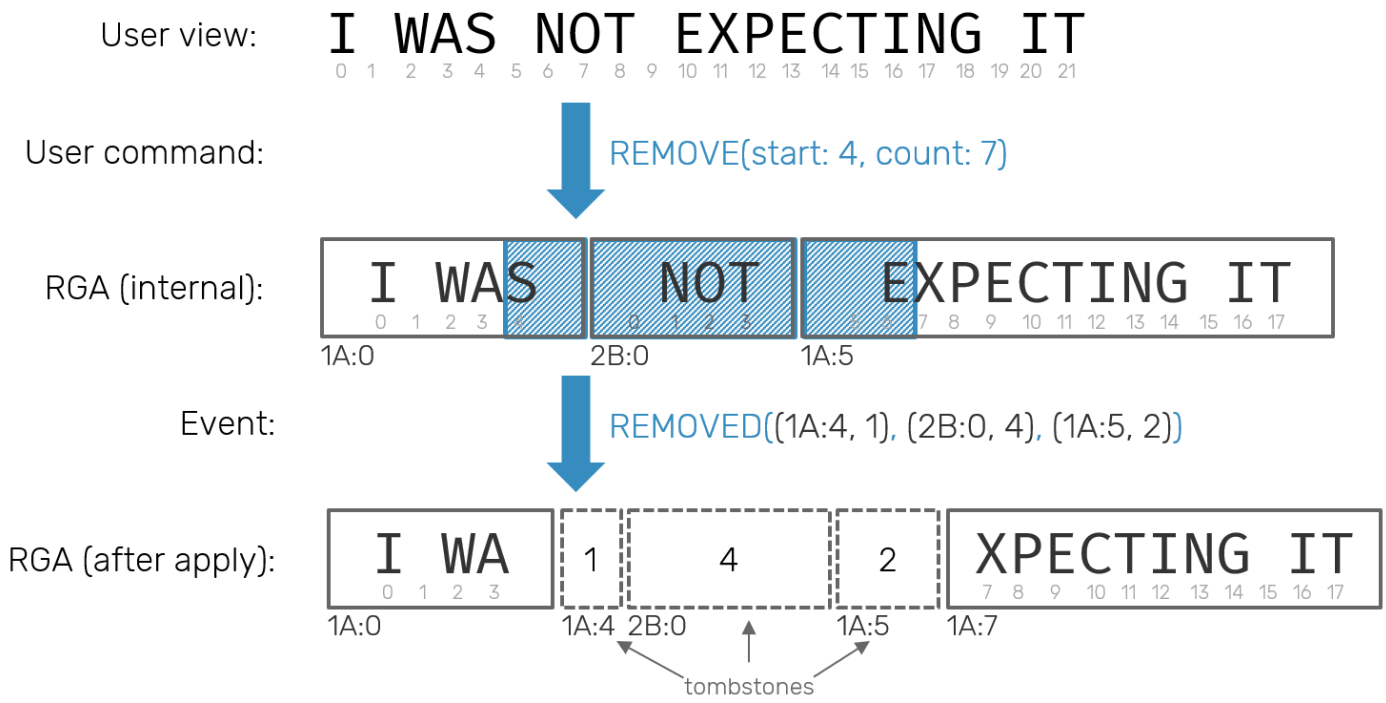
1. Find predecessor after which we need to insert our element.
2. If insert should be done inside of the predecessor block, split it in two.
3. If another block was inserted concurrently resolve the insertion index of the new block based on virtual pointer ordering (as we covered in previous chapter).
4. If split occurred in pt.2, replace predecessor block with its left half and insert the right one just after newly inserted block.

```
let applyInserted predecessor ptr items rga =
  let (index, blockIndex) = findByPositionOffset predecessor rga.Blocks
  // in case of concurrent insert of two or more blocks we need to check which of them
  // should be shifted to the right and which should stay, and adjust insertion index
  let indexAdjusted = shift (index+1) ptr rga.Blocks
  let block = rga.Blocks.[index]
  // since we're about to insert a new block, its offset will always start at 0
  // (it can only be changed, when the block is subject to slicing)
  let newBlock = { PtrOffset = { Ptr = ptr; Offset = 0}; Data = Content items }
  let (left, right) = Block.split blockIndex block
  let blocks =
      rga.Blocks
      |> Array.replace index left  // replace predecessor block if split occurred
      |> Array.insert indexAdjusted newBlock // insert new block
  // if split occurred we also need to insert right block piece back into an array
  let blocks = right |> Option.fold (fun blocks b -> Array.insert (indexAdjusted+1) b blocks)
```

```
    // update sequencer
    let (seqNr, replicaId) = rga.Sequencer
    let nextSeqNr = (max (fst ptr) seqNr, replicaId)
    { Sequencer = nextSeqNr; Blocks = blocks }
```

Range deletion

Deleting a range of elements is much more mind-boggling exercise. From the user perspective it should look simple: we pick start index and remove a continuous number of elements out of it. The problem is that this sequence of removed elements may be spanned over multiple RGA blocks, possibly starting and/or ending in the middle of one.



The distinction between command (user request) and event (replicated operation) is that events must not relate to user-facing indexes - as they are mutable and therefore not trustworthy when doing concurrent updates - but use block's virtual pointers with offsets + number of tombstoned elements - that number only really matters for the first and last block, as we can trust that all blocks in between will be deleted entirely (as they can only be splitted, but never joined).

```
type Command<'a> =
  | Insert of index:int * 'a[]
  | RemoveAt of index:int * count:int

type Operation<'a> =
  | Inserted of predecessor:VPtrOffset * ptr:VPtr * value:'a[]
  | Removed of slices:(VPtrOffset*int) list
```

Block splitting operation may come handy once again, when a removal range starts or ends in the middle of the block - as you've seen, the only two valid states for each block are either tombstoned

or alive (having active elements). Nothing in between. In that case we can always split that block and tombstone a part, that needs to be removed.

Things we also need to think about are:

1. What if blocks that are subject to tombstoning, were split in the meantime by another, concurrent operation? We still need to find and tombstone them given their old shape at the time when event was originally produced.
2. What if concurrent operation inserted new block in the middle of deleted range? Since it was not part of deleted range it should be displayed, we cannot tombstone it by accident or we'd end up with corrupted state.

These and many more questions are not necessarily very easy to express in code. For our implementation we simply assume that list of slices provided with `Removed` event is ordered adequately to their order in RGA itself: it has sense, as blocks can be inserted or split, but never reordered meaning that we **never run into situation** where RGA with blocks `[A,B,C]` will be transformed into `[B,A,C]`.

The core skeleton of removal algorithm looks like this:

```
let applyRemoved slices blocks =
  let rec loop (acc: ResizeArray<Block<'a>>) idx slices (blocks: Block<'a>[]) =
    match slices with
    | [] ->
      // once we tombstoned all expected blocks, just copy over remaining ones
      for i=idx to blocks.Length-1 do
        acc.Add blocks.[i]
      acc.ToArray()
    | (ptr, length)::tail ->
      let block = blocks.[idx]
      if block.PtrOffset.Ptr = ptr.Ptr then // we found a valid block

        // block tombstoning algorithm body...

      else
        // this is not a block we're looking for, just copy it over as is
        acc.Add block
        loop acc (idx+1) slices blocks
  loop (ResizeArray()) 1 slices blocks // start from 1 as 0 is RGA head element
```

We omitted a body of tombstoning algorithm for a moment to think about possible scenarios. We could put them into following categories:

1. Removal start position is inside of a given block.
2. Removal start position is outside of a given block - which is possible eg. because another concurrent operation had already fragmented it into more blocks.

We can represent these as:

```
// block tombstoning algorithm body...
if block |> Block.containsOffset ptr.Offset then

  // the tombstoned slice starts inside of a current block...

else
  // position ID is correct but offset doesn't fit, we need to move on
  acc.Add block
  loop acc (idx+1) slices blocks
```

Again we left one branch untouched - we know that we found the block we want to tombstone, but we first need to determine if the removed range starts at with the beginning of that block or somewhere in the middle:

```
let splitIndex = ptr.Offset - block.PtrOffset.Offset
let tombstone : Block<'a> =
  if splitIndex = 0
  then block // beginning of tombstoned block is exactly at the beginning of current one
  else
    // beginning of tombstoned block is in the middle of current one
    // split it in two and keep the left part alive
    let (left, Some right) = Block.split splitIndex block
    acc.Add left
    right

// tombstone...
```

So, this way we handled the left boundary of our removal range. Now to the right one - we need to check if number of removed elements matches the number of elements of the block itself:

1. Usually a number of removed elements will fit within the bounds of the current block. We may again run into situation, when we need to split that block in two, tombstone the left part and keep around the right one.
2. If concurrent update caused tombstoned block to be split if may turn out that the number of elements we're about to remove, is longer than the current block. In that case we need to tombstone the entire block and reinsert the updated slice (shortened by the number of elements we just tombstoned) back to the list of slices used to idenitify blocks to be removed.

```
// tombstone...
if length <= tombstone.Length then
  // split the block if number of elements to be removed is not equal its length
  let (left, right) = Block.split length tombstone
  acc.Add (Block.tombstone left) // tombstone left part
  right |> Option.iter acc.Add // if right part exists, add it to the result RGA
  loop acc (idx+1) tail blocks
```

```
  else
    // tombstone length is longer than size of a block
    acc.Add (Block.tombstone tombstone)
    // update slice to contain remaining length to be tombstoned
    let remaining =
      /// take a slice and shorten it by moving its pointer offset
      /// and reducing to-be-removed length
      let ptr = { ptr with Offset = ptr.Offset + tombstone.Length }
      let remainingLength = length - tombstone.Length
      (ptr, remainingLength)
    // put remaining part back onto processing list
    loop acc (idx+1) (remaining::tail) blocks
```

Uff, that's a lot of code for a single function, but we got a lot of cases to cover. If you're confused, feel free to help yourself with the source.

Summary

We described how to build a variant of Replicated Growable Array from the previous post, optimized for insertion and deletion of entire ranges of elements at the same time.

In this implementation we used an ordinary arrays, which were copied to maintain immutable characteristics. You probably already know, that this is not the most efficient approach. In practice structures like ropes are much better suited for this sort of workload.

You may have noticed that block slicing may eventually lead to suboptimal situation, where our entire sequence is internally fragmented into single character blocks, causing it to be more expensive than original RGA structure. For some situations this may be a valid concern. However for cases like collaborative text editing, which was the major motivation for this algorithm the cons shouldn't overwhelm the pros in most of the practical scenarios.

Operation-based CRDTs: JSON document

Today we're going to cover how to build a complex, JSON-like document CRDT. In the past, we focused on homogeneous data types like registers, sets, arrays or maps. This time we're going to combine them all and tackle some of the challenges that this approach presents.

Prerequisites

While this post reuses many concepts we learned previously in this series, it vaguely covers their properties in scope required to implement document CRDT. However if you want to get better intuition or find it not sufficient, you may want to read (at least fragments of) other chapters concerning vector clocks, reliable causal broadcast protocol, multi-value registers and linear sequences.

We'll use snippets from **source code** that can be found here. Feel free to use it to follow along.

Document representation

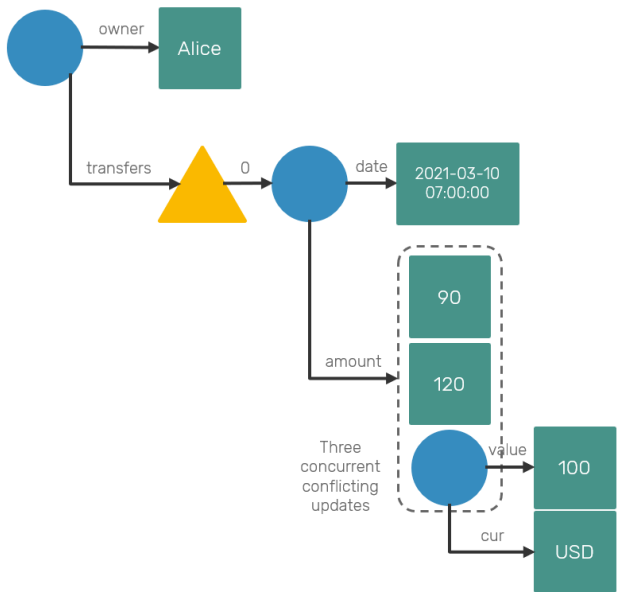
A conflict-free replicated JSON document can be basically looked at as a tree of nodes. We can differentiate two different types of them:

- **Leafs** which are used to store primitive types. In JSON we have several of them. We'll use: `int`, `float`, `string`, `bool` and `null`, all wrapped as a cases of discriminated union, which we're going to call `Primitive`.
- **Branches** represent more complex containers, like maps and arrays.

Since we're talking about JSON-like structure, our data type won't have a fixed, static schema. In previous chapters we were constraining ourselves to develop conflict resolution procedures that would work over objects of the same type. Here we can run into situation, where the same node is concurrently updated with values of different types eg. `account.transfers[0].amount = 120` on one replica and `account.transfers[0].amount = { value: 100, currency: "usd" }` on another.

```
{
  owner: “Alice”
  transfers: [
    {
      date: “2021-03-10T07:00:00”,
      amount: (90|120|{value: 100, cur: “USD”})
    }
  ]
}
```

Three concurrent conflicting updates



This is an expected problem to have. In reality our systems evolve over time and so does the schema we're using. Since we operate in decentralized peer-to-peer fashion, we cannot really stop others from submitting outdated updates. This means that while one peer may be in version 2.0 using new schema, it still might talk with a peer in v1.0 who was offline for long time, building up its own state according to old schema. We cannot just ignore it or otherwise we can end up with corrupted state. We need to be ready to embrace that.

In the past we talked about two methods of dealing with updating primitive values in conflict-free way: Last-Write-Wins and Multi-Value registers. Here, we're going with the latter approach and will extend it over different types living under the same node - so in case of concurrent updates, we'll show all conflicting values, expecting user to override them manually. Some people may find last

write wins might to be more convenient approach, but it also comes with a risk of data lost (as old values are automatically discarded).

For arrays, we'll use LSeq defined in the past - and reuse its `VPtr` and `generateSeq` definitions, as they stay exactly the same. LSeq is easy to construct and its behavior maps well enough for a JSON array. Finally for object/map we're going with something inspired by an Observed-Remove Map. We'll try to maintain Add-Wins semantics over the entire document - so in case of entry being updated on one replica and deleted on another one, we always will resolve such conflict in favor of updates.

With these semantics and tradeoffs defined up-front, let's start with core types definitions:

```
type Vertex<'a> = (VPtr * 'a) // a single LSeq element

type Node = Entry list
and Entry =
  | Leaf of VTime * Primitive // Int, Float, String, Bool or Null
  | Array of VTime list * Vertex<Node>[]
  | Object of VTime list * Map<string,Node>
```

As you can see in a snippet above, our node is in fact a list of concurrently updated values. It can consists of multiple `Leaf` s (as they may represent concurrent conflicts of multi-value register), while our branch entries (`Array` and `Object` cases) will always exist only once within the node. It's because they represent a complex containers that define their own conflict resolution strategies.

We want to keep information when a particular branch was updated by many events concurrently - so that we can detect concurrent conflicts - therefore each entry type has a vector clock (`VTime`) associated with it. For `Leaf` its a single value, representing when corresponding value was updated for the last time. Branch types (`Array` and `Object`) may contain multiple inner entries updated concurrently by different events. Hence for them we store concurrent (and only concurrent) updates as a `VTime list` .

In terms of allowed operations we'll stick to the basics: adding, updating and removing a primitive value under a path that can consist of object fields and/or array indexes. We don't consider specialized operations like incrementing counter value or text operations: if you want to have support for them, keep in mind that in context of CRDTs they require specialized node types and operation handling - intent is a king here and for concurrent/distributed operations there's a huge difference between `a = a + 1` and `increment(&a)` . We discussed these algorithms and data structures already (see links at the beginning of a post), but here we'll skip them for sake of simplicity.

```
type Command =
  | Assign   of Primitive
  | Remove
  | Update   of key:string * Command
  | UpdateAt of index:int * Command
```

```
    // append at the end by `InsertAt(list.Length, Assign (String "value"))`
    | InsertAt of index:int * Command
```

One thing you may notice here is that while there's no semantic difference between inserting and updating an object field, it's not the case for inserts/updates on array indexes - inserting a new item at index 0 and updating an item at index 0 produce very different results - so we need separate commands for these.

Now once we talked about user-facing commands, let's consider events that actually will be stored and replicated between machines. These events must convey enough metadata to make our operation commutative, so that they can be applied in causal order. Part of that (like operation's vector clock) is already carried as part of our generic reliable causal broadcast protocol. However document also requires some custom information.

```
type Event =
  | AtKey     of string * Operation
  | AtIndex   of VPtr * Operation
  | Assigned  of Primitive
  | Removed
```

Our event is a composite of navigation cases (like `AtKey` and `AtIndex`) and actual operation type (`Assigned` and `Removed`). Interesting thing to notice is that while on the command side we distinguished between `InsertAt` / `UpdateAt`, on the event side we have just a single `AtIndex`. That's because once we supply our event with virtual pointer (`VPtr` type we borrowed from the LSeq article), we no longer need to worry about mistaking insert with update, as virtual pointers - unlike standard array indexes - are immutable and unique position descriptors for stored elements.

We described the theory behind command→event conversion. Now let's look at the possible implementation:

```
let rec handle (replicaId: ReplicaId) (node: Node) (cmd: Command) =
  match cmd with
  | Assign value -> Assigned value

  | Remove -> Removed

  | Update(key, nested) ->
    // get Object component of the node or create it
    let map = Node.getObject node |> Option.defaultValue Map.empty
    let inner = Map.tryFind key map |> Option.defaultValue Node.empty
    AtKey(key, handle replicaId inner nested)

  | InsertAt(_, Remove) ->
    failwith "cannot insert and remove element at the same time"

  | InsertAt(i, nested) ->
    // get Array component of the node or create it
```

```
      let array = Node.getArray node |> Option.defaultValue [||]
      // LSeq generates VPtr predecesor/successor virtual pointers
      let left = if i = 0 then [||] else (fst array.[i-1]).Sequence
      let right = if i = array.Length then [||] else (fst array.[i]).Sequence
      let ptr = { Sequence = generateSeq left right; Id = replicaId }
      AtIndex(ptr, handle replicaId Node.empty nested)

    | UpdateAt(i, nested) ->
      // we cannot update index at array which doesn't exist
      let array = Node.getArray node |> Option.get
      let (ptr, inner) = array.[i]
      AtIndex(ptr, handle replicaId inner nested)
```

You may notice that, unlike in the case of JavaScript, when we want to create a value deep in the path that didn't previously exist eg. `{}.account.balance = 100`, we'll dynamically create all subsequent objects/arrays required to realize this request. Otherwise we couldn't serve concurrent set/removal operations i.e. when one replica is deleting entire branch upper in the tree, while another one is changing a leaf node value.

I think, the most confusing part of the code above is `InsertAt` case. It's because it exposes behavior characteristic to `LSeq` data type. A little reminder: *`LSeq` can be imagined as an array of elements sorted by their virtual pointers. We can realize inserting new element at index `n` by generating virtual pointer, that's logically greater than virtual pointer of element at index `n-1` and smaller that virtual pointer at index `n+1`. For array edges we can replace them with `VPtr.MIN` and `VPtr.MAX` substitutes.*

As you may have seen, our command/event allows to only update a single path element at the time. In practice nothing stands on your way to change discriminate union definition to store multiple nested operations instead of one.

Value assignment

Now it's time to talk about commutative event application to our document state. Let's start with the simplest one: assignment of a primitive at given node. Using multi-value register semantics we just add a new leaf entry to our node while keeping all concurrent entries around and discard others.

```
module Node

let assign (timestamp: VTime) (value: Primitive) (node: Node) : Node =
  let concurrent =
    node |> List.filter (fun e -> not (Entry.isBefore timestamp e))
  Leaf(timestamp, value)::concurrent
```

We haven't defined `Entry.isBefore` yet - its construction is fairly simple: entry happens before given time, if ALL of entry's timestamps have happened before submitted time.

```
module Entry

let isBefore (time: VTime) (e: Entry) =
  match e with
  | Leaf(ts, _) -> Version.compare ts time < Ord.Eq
  | Array(ts, _)
  | Object(ts, _) ->
    ts |> List.forall (fun ts -> Version.compare ts time < Ord.Eq)
```

You may have say "you've mentioned that we're concerned about concurrent operations, but here we're looking only at happen-before relations". Yes, but keep in mind, that our event application function is backed by properties of Causal Reliable Broadcast protocol we covered in the past. This protocol handles deduplication (case when events have equal timestamps) and will never apply event that would violate partial order (so applying successor event before its strict predecessor). So all remaining options are entries that are either concurrent to or have been updated strictly before current event.

Updating fields

Updating fields is very similar to a value assignment. We take an `Object` entry and apply a nested event to it. Then we need to check if there are concurrent events happening in scope of current object entry (the `VTime list`) and keep them. Do the same again, but this time with entries in scope of the node itself. Finally return a new updated `Object` entry with updated timestamp and remaining (concurrent) entries.

```
let rec apply replicaId (timestamp: VTime) (node: Node) (op: Event) : Node =
  match op with
  // ... other cases
  | AtKey(key, nested) ->
    // get Object component from the node or create it
    let timestamps, map =
      match List.tryFind (function Object _ -> true | _ -> false) node with
      | Some(Object(timestamps, map)) -> (timestamps, map)
      | _ -> ([], Map.empty)
    let innerNode = Map.tryFind key map |> Option.defaultValue Node.empty

    // apply inner event and update current object
    let map = Map.add key (apply replicaId timestamp innerNode nested) map
    // keep concurrent timestamps of a current object entry
    let timestamps =
      timestamp::(List.filter (fun ts -> Version.compare ts timestamp = Ord.Cc) timestamps)

    // keep concurrent entries of a current node
    let concurrent =
      node |> List.choose (fun e ->
        match e with
        | Object _ -> None // we cover object update separately
```

```
        | outdated when Entry.isBefore timestamp outdated -> None
        | other -> Some other)
    // attach modified object entry to result node
    (Object(timestamps, map))::concurrent
```

Array insert/update

Since we handled array insert/update distinction at command handler, within event handler we can treat them almost in the same way. We need to simply use binary search by virtual pointer to find the index of a node to insert update - that's how `LSeq` handles any potential concurrency conflicts. If virtual pointer under that index is equal to the one passed in event - it's an update. Otherwise it's an insert.

```
let rec apply replicaId (timestamp: VTime) (node: Node) (op: Event) : Node =
  match op with
  // ... other cases
  | AtIndex(ptr, nested) ->
    // get Array component from the node or create it
    let mutable timestamps, array =
      match List.tryFind (function Array _ -> true | _ -> false) node with
      | Some(Array(timestamps, array)) -> (timestamps, array)
      | _ -> ([], [||])

    // find index of a given VPtr or index where it should be inserted
    let i = array |> Array.binarySearch (fun (x, _) -> ptr >= x)
    if i < Array.length array && fst array.[i] = ptr then
      // we're updating existing node
      array <- Array.copy array // defensive copy for sake of update
      let (_, entry) = array.[i]
      array.[i] <- (ptr, apply replicaId timestamp entry nested)
    else
      // we're inserting new node
      let n = (ptr, apply replicaId timestamp Node.empty nested)
      array <- Array.insert i n array

    // keep concurrent timestamps of a current array entry
    let timestamps =
      timestamp::(List.filter (fun ts -> Version.compare ts timestamp = Ord.Cc) timestamps)
    // keep concurrent entries of a current node
    let concurrent =
      node |> List.choose (fun e ->
        match e with
        | Array _ -> None // we cover array update separately
        | outdated when Entry.isBefore timestamp outdated -> None
        | other -> Some other)
    // attach modified array entry to result node
    (Array(timestamps, array))::concurrent
```

Updating `LSeq` array elements shares many similarities with updating an object. In both cases we override other non-concurrent entries within the same node and non-concurrent timestamps within the same entry.

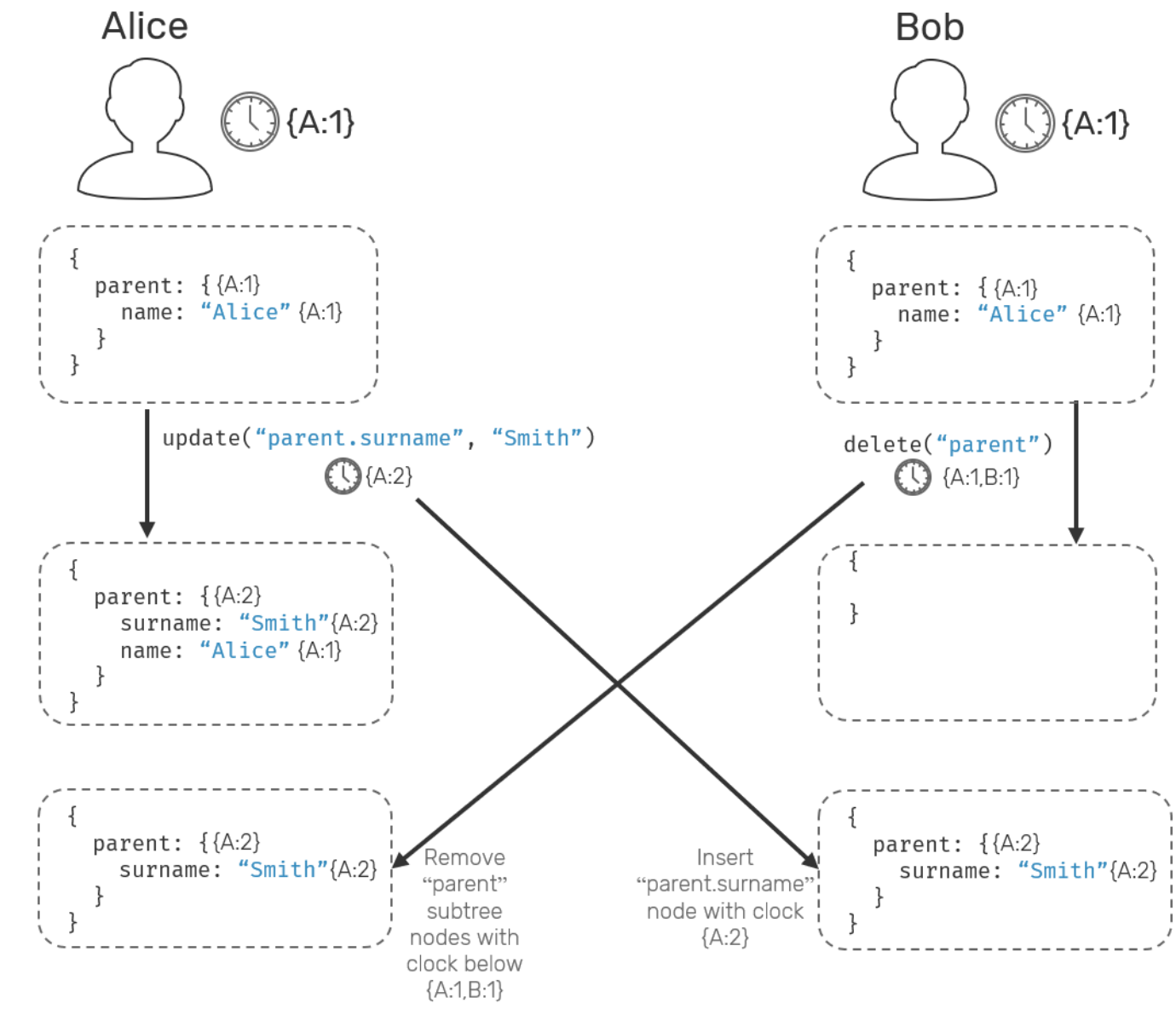
Removing nodes

The last remaining operation is node removal. In principle, we can take advantage of timestamps generated by our broadcast protocol and traverse over the tree to remove entries, which have all of their timestamps happening before removal event.

For entries having concurrent updates, we either preserve them (in case of `Leaf` s, which maps directly to add-wins semantics) or have to propagate removal procedure recursively (for branch node entries).

```
let rec removeNode (tombstone: VTime) (node: Node) : Node option =
  let node = node |> List.choose (fun e ->
    match e with
    | Leaf(timestamp, _) ->
      // remove node with timestamp lower than tombstone
      if Version.compare timestamp tombstone <= Ord.Eq then None else Some e

    | Array(timestamps, vertices) ->
      // check if all of array's timestamps are behind tombstone
      if timestamps |> List.forall (fun ts -> Version.compare ts tombstone < Ord.Eq) then
        None // remove node with lower timestamp
      else
        // recursivelly check if other array elements need removal
        let vertices =
          vertices
          |> Array.choose (fun (ptr, node) -> removeNode tombstone node |> Option.map (fun n -
        Some(Array(timestamps, vertices))

    | Object(timestamps, fields) ->
      // check if all of object's timestamps are behind tombstone
      if timestamps |> List.forall (fun ts -> Version.compare ts tombstone < Ord.Eq) then
        None // remove node with lower timestamp
      else
        // recursivelly check if other object fields needs removal
        let fields = fields |> Map.fold (fun acc key value ->
          match removeNode tombstone value with
          | None -> acc
          | Some v -> Map.add key v acc) Map.empty
        Some(Object(timestamps, fields))
  )
  // if all node entries were removed remove node itself
  if List.isEmpty node then None else Some node
```

Now, one thing we need to keep in mind is that concurrent removal and updates of nodes which are at the different depth of the document tree may result in state that may be surprising for some. Imagine following scenario: *we start with some initial document state* `{ parent: { name: "Alice"}}` *that's in sync between nodes A and B. Both nodes made a concurrent update: A did* `parent.surname = "Smith"` *while B called for* `delete("parent")` *. What should be the state of the document after both replicas synchronize again?.*



As we mentioned, we prefer add-wins semantics, so in our case the resulting object will be `{ parent: { surname: "Smith" } }` . But wait, what has happened to `parent.name` ? Well, since it was assigned prior to delete request, it had lower timestamp and therefore it has been removed. As far as our document CRDT is concerned, `parent.name` and `parent.surname` are independent entries, and their updates or removals are happening independently. If you want to prevent that, you'd need to refresh assignment of `parent = { name: "Alice", surname: "Smith" }` , so that a `parent.name` field timestamp will be reassigned.

Summary

We presented how to implement a strict core of what could be considered a minimal version of a JSON document using CRDT semantics.

We didn't cover more advanced operations like multi-entry updates, rich data types like counters or edit-capable text fields. Other interesting area would be compressing the metadata footprint or changing our data type to work natively with database storage engines (like LSM trees) so that we could support documents that are larger than RAM. For now let's save these topics for another time.

Pure operation-based CRDTs

In this chapter we'll continue on topic of operation-based CRDTs and focus on the optimizations and approach known as pure operation-based CRDTs: how can we use them to address some of the problems related to partially ordered event logs and optimize size of our data structures.

If you tracked down this series (or were just interested by the topic on your own), you may notice that CRDTs are usually exposed as 1. data types with metadata combined with 2. replication protocol. State- and operation-based implementations shuffle complexity between these two. The more guarantees and metadata we can attach at protocol level the less we need them at data type side. Pure operation-based CRDTs could be considered to stay at the very end of protocol-complexity (in logical sense, not necessarily implementation-wise) of this spectrum: here we'll define a replication protocol that will allow us to establish a great number of guarantees about the state of the distributed system. In return it will allow us to write our data type definitions in maximally succinct manner.

Rationale behind pure operation-based CRDTs

An original paper that introduced notion of pure operations, proposed solution to several problems of traditional operation-based CRDTs:

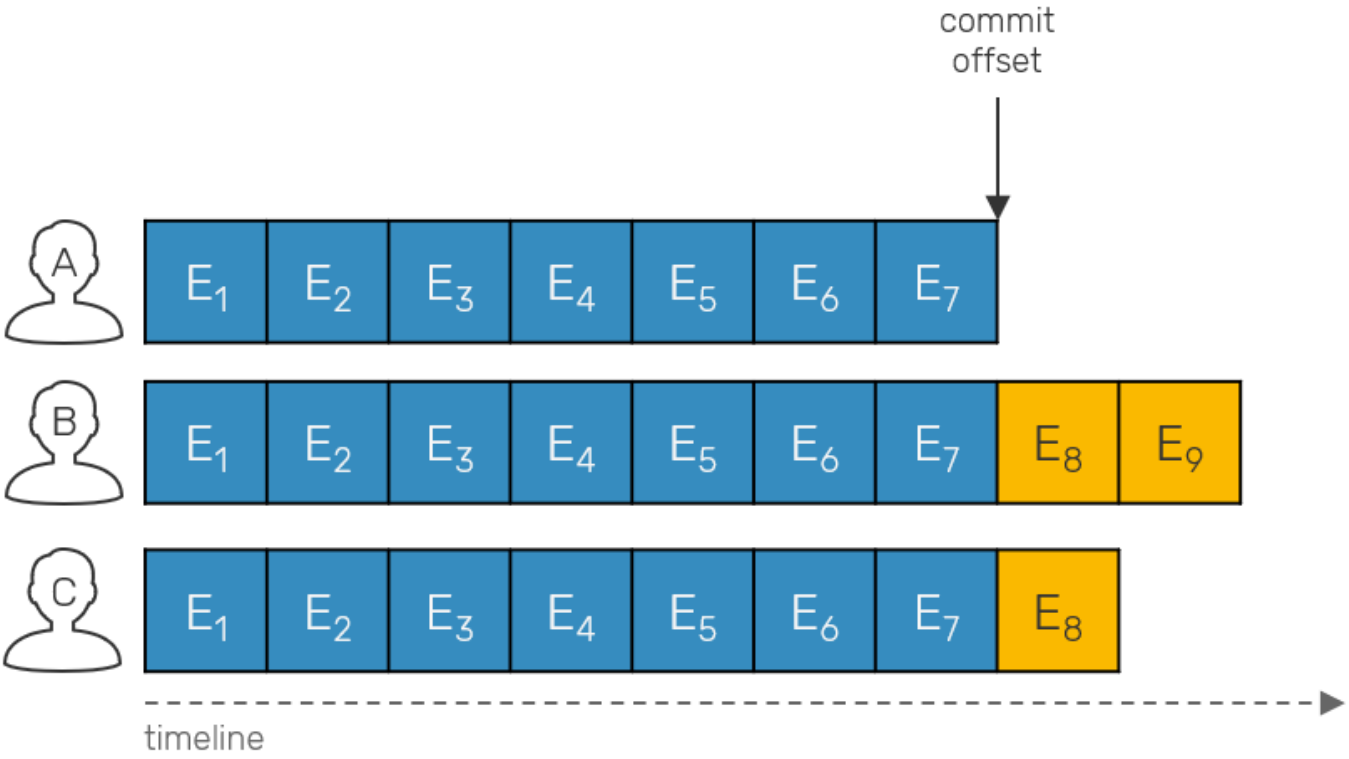
- **Reduce an amount of metadata kept together with the state**. Being able to independently resolve concurrent conflicts may require additional information and CRDTs are infamous from keeping a lot of metadata, which can potentially outweigh the actual payload.
- **Events pruning**. Once an event (eg. removing value from the set) has been applied on all replicas in the system we want to be able to eventually delete it. Determining when it's safe to do so without any consensus algorithm is not an easy task.

The idiom "pure" comes from the fact that in this implementation, there's no more distinction between commands and events, only operations. If you remember, we had this distinction in operation-based CRDTs because while our user-facing commands were stripped of CRDT-specific metadata (which was not exposed to the client), different data structures had custom requirements that could not be generalized. In pure variant, we'll get rid of this - we won't need commands, events or command handlers.

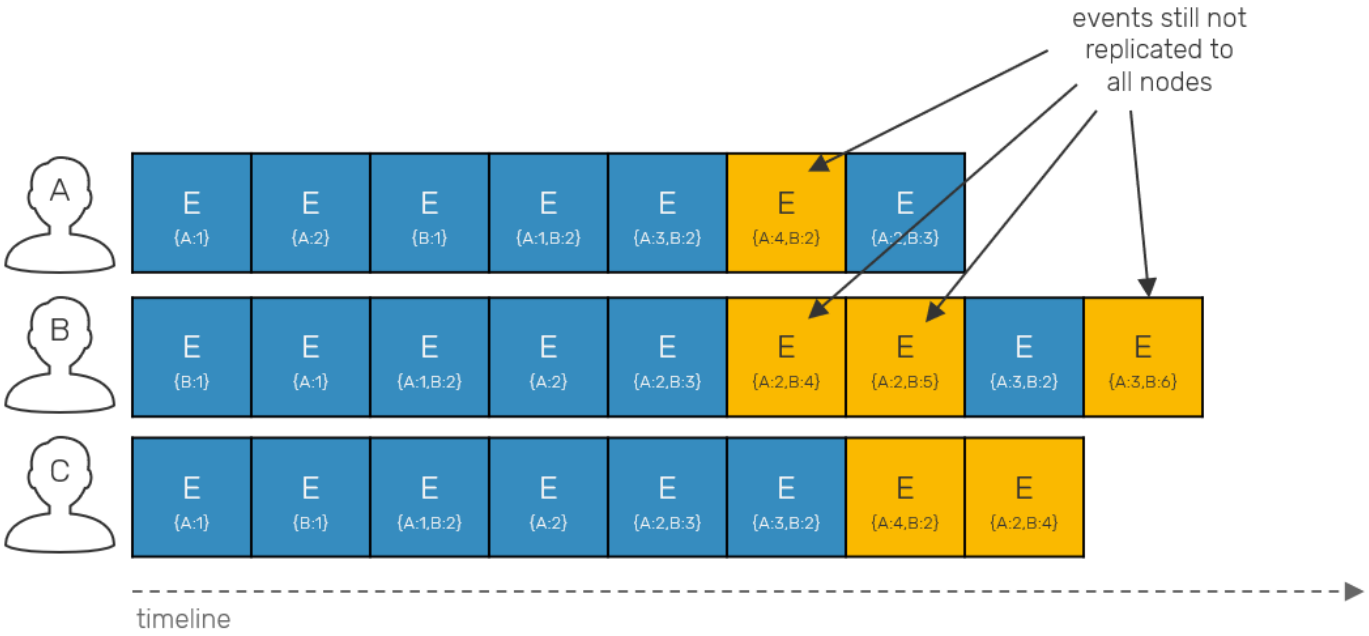
For the reasons described above we no longer stick to eventsourcing terminology used before - it's still possible to implement our solution on top of event log, this time we'll go with a simpler approach. But before we do so, first we need to understand concept of causal stability.

Causal stability

When we're building a partially ordered log of operations, at some point we need to answer the question: *when is it safe to prune events?* When we talk about logs with a total order (all replicas store events in exactly the same order eg. Kafka) we can use so called *commit offset* that allows us to determine, up to which position in a log all replicas are in sync. This commit offset tells us when it's safe to snapshot our state and then delete events.



Once we drop total order, things start to get more complicated. Partially-ordered logs can have the same events under different offsets in a their local log. We track individual events with vector clocks, as they allow us to track causality aka. happened-before relationships between events required by this approach.



We talked about vector clocks a lot over this series, and you really should have solid grasp on them before we move forward. The initial problem which we need to solve, is **how to tell when event was replicated on all sides in partially ordered log?**

Keep in mind that we cannot require any form of voting or arbitrary leader to solve this problem - our system is leaderless, which means that every replica needs to work in read-write manner on its own, even when disconnected from the rest of the system. Since machines cannot synchronize with each other, they must infer the safe point on their own, based only on incoming metadata.

If you remember our previous operation-based CRDT protocol implementation, we timestamped each event with a vector clock. It's used for ordering, but it also conveys more information. It tells us what was the observed *state of the world* from perspective of its emitter at the time it was produced. Now, we know that an event is safe to be deleted once it has been received by all replicas. How can we tell that event has been observed by everyone without explicitly asking them? Look at the incoming timestamps and let the mystery unravel.

Vector clocks define a merge method, which lets us generate the latest version of two given vectors by picking the maximum sequence numbers of their corresponding entries. This has sense, as these values are only incremented over time, so the highest sequence number in vector clock will always be more up-to-date. However, what will happen when instead of picking maximum, we **pick minimum values of the corresponding vector clock entries**? The resulting vector clock will describe a point in (logical) time seen by both emitters. If we'll fold this minimum over all most recent timestamps received from all replicas, what we get is the timestamp that describes the point in time seen by everyone - so called **stable timestamp**.

This timestamp serves us as a check point - since we know that all events prior to it have been acknowledged by all replicas. This also means, that **there are no operations in flight that could possibly happen concurrently to a stable timestamp**. It's important property, which we'll use later.

In order to keep track of vector clocks arriving from different replicas, we can use so called *Matrix Clock* - it's just a map of all replicas and their corresponding timestamps:

```
/// Matrix clock.
type MTime = Map<ReplicaId, VTime>

[<RequireQualifiedAccess>]
module MTime =

  let min (m: MTime): VTime =
    Map.fold (fun acc _ v -> Version.min acc v) Version.zero m

  let max (m: MTime): VTime =
    Map.fold (fun acc _ v -> Version.max acc v) Version.zero m

  let merge (m1: MTime) (m2: MTime) : MTime =
    (m1, m2) ||> Map.fold (fun acc k v ->
      match Map.tryFind k acc with
      | None -> Map.add k v acc
      | Some v2 -> Map.add k (Version.merge v v2) acc)

  let update (nodeId: ReplicaId) (version: VTime) (m: MTime) =
    match Map.tryFind nodeId m with
    | None -> Map.add nodeId version m
    | Some v -> Map.add nodeId (Version.merge version v) m
```

Tagged Stable Causal Broadcast protocol

The basic protocol behind pure operation-based CRDTs is called Tagged Reliable Causal Broadcast or Tagged Stable Causal Broadcast. The name comes from the fact that we expose part of machinery responsible for tagging events (vector timestamps) in the user API. If you recall, we already did that in the previous implementation, for practical reasons. Moreover we could just modify the same protocol a little, but this would mean that we'd carry all cruft, that's unnecessary here. Instead let's build something simpler.

As always, we're only presenting crucial snippets here. If you want to see a full **source code**, use this link.

The core object, we're going to use is an `Event`:

```
[<CustomComparison;CustomEquality>]
type Event<'a> =
  { Value: 'a
    Version: VTime
    Timestamp: DateTime
    Origin: ReplicaId }
  member this.CompareTo(other: Event<'a>) =
    match Version.compare this.Version other.Version with
```

```
    | Ord.Cc -> // if versions are concurrent, compare other fields
      match this.Timestamp.CompareTo other.Timestamp with
      | 0   -> this.Origin.CompareTo other.Origin
      | cmp -> cmp
    | cmp -> int cmp
```

While the original paper only discusses values and vector versions, here we'll also add system clock timestamps together with ID of the event emitter. Thanks to them we're able to properly order even concurrent events. It's also required to implement some data types like Last Write Wins Registers or Indexed Sequences.

You can also notice that we no longer attach any sequence numbers to events - we did so in the past to be able to efficiently scan events in our persistent store. Since now we'll be able to prune events efficiently, our event log will be much shorter though. Also in this implementation usually will need to scan all non-pruned events anyway.

One of the core concepts behind pure CRDTs is the ability to split their state in two:

1. A stable state represents compacted information from events that happened up to a stable timestamp. What's important here is that, because we know that there are no events concurrent to a stable vector version, we don't need any metadata associated with them, as we won't have any conflicts to resolve. This helps us to drastically reduce memory overhead.
2. An unstable state represents events, which happened after stable vector version and have not been confirmed to be received by all replicas.

As more and more replication events arrives from all sides, our stable version moves forward and when that happens we're able to prune events from unstable log and apply them to a stable state snapshot. Here for simplicity we'll keep both of them as in memory objects. For practical implementation, you most likely want to persist them.

```
type State<'state,'op> =
  { Id: ReplicaId             // id of current replica
    Stable: 'state            // stable state
    Unstable: Set<Event<'op>> // operations waiting to stabilize
    StableVersion: VTime      // last known stable timestamp
    LatestVersion: VTime      // most up-to-date vector timestamp
    Observed: MTime           // vector versions of observed universe
    Connections: Map<ReplicaId, (Endpoint<'state, 'op> * ICancelable)> }
```

We start by defining how to update our state locally. We'll do it by sending a single `Submit` message with user-defined operation, that's stripped of any specific metadata internal to the system.

```
let apply (crdt: PureCrdt<'state,'op>) (state: State<'state,'op>) =
  if Set.isEmpty state.Unstable
  then state.Stable
  else crdt.Apply(state.Stable, state.Unstable)
```

```
let rec active (crdt: PureCrdt<'state,'op>) (state: State<'state,'op>) (ctx: Actor<Protocol<'s
  match! ctx.Receive() with
  | Submit op ->
    let sender = ctx.Sender()
    let version = Version.inc id state.LatestVersion
    let observed = state.Observed |> MTime.update id version
    let event =
      { Version = version
        Value = op
        Timestamp = DateTime.UtcNow
        Origin = id }
    let pruned =
      state.Unstable
      |> Set.filter (fun o -> not (crdt.Obsoletes(event, o)))
    let state = { state with
                    Unstable = Set.add event pruned
                    LatestVersion = version
                    Observed = observed }

    sender <! (apply crdt state) // respond with new state view
    return! active crdt state ctx
```

What we're doing here is simply creating a new event with an incremented version and adding it to an unstable set - ignoring a single node systems, we can tell that this event was not yet received by all replicas (therefore cannot be applied into stable state) because we just created it. We also check if newly created event doesn't obsolete any other events we already have in our log.

What does *obsolete* even mean in this context? This is a tricky part as it depends on what actual CRDT implementation we talk about, but in principle keep in mind that unstable events can be in different causal relationships to each other - they can be concurrent or happen strictly one after another. In some cases it's useless to keep an event eg. if we have last-write-wins register, every event assigning a new value to the register obsoletes all older events. We can simply remove them as they're no longer needed.

Next, let's go into replication protocol. To make replication possible we'll need a set of serializable messages, like:

```
type Protocol<'state,'op> =
  | Replicate        of ReplicaId * VTime
  | Reset            of from:ReplicaId * Snapshot<'state>
  | Replicated       of from:ReplicaId * Set<Event<'op>>
  | ReplicateTimeout of ReplicaId
  // other messages not related to replication prodecure
```

The idea is simple:

1. Node *A* wants to check if node *B* has some new events to be replicated (pull-based model). It sends a `Replicate` request with its **latest known vector version**.
2. When node *B* receives replication request it can verify attached timestamp against its own **stable version**:
    i. If *A*'s latest version is equal or behind *B*'s stable version it means, that sending *B*'s unstable events is not enough as it doesn't send all of the information. We also need to send a snapshot of a stable state - it does so in `Reset` message. It's also necessary when fresh node joins the cluster of replicas.
3. *B* filters out all of its unstable events that happened after or concurrently to vector version received in `Replicate` request and sends them back in a `Replicated` message.
4. If response didn't arrive before expected timeout, a `ReplicateTimeout` signal will trigger and we'll repeat our request again back from point 1.

So, with this conceptual description in mind, let's see how this could look in code:

```
let rec active crdt state ctx = actor {
  match! ctx.Receive() with
  | Replicate(nodeId, filter) ->
    let (replyTo, _) = Map.find nodeId state.Connections
    if Version.compare filter state.StableVersion = Ord.Lt then
      replyTo <! Reset(id, toSnapshot state) // step 2.1 - send a snapshot

    // send events in batches no bigger than 100 at a time
    let batch = ResizeArray(100)
    for op in state.Unstable do
      if Version.compare op.Version filter > Ord.Eq then
        batch.Add op
      if batch.Count >= 100 then
        // step 3 - send events which need replication
        replyTo <! Replicated(id, Set.ofSeq batch)
        batch.Clear()

    if batch.Count > 0 then
      replyTo <! Replicated(id, Set.ofSeq batch)

    return! active crdt state ctx
  // handlers for other messages
```

We replicate by sending batches of operations, since we cannot guarantee their upper bound. These batches can be prepended by a `Reset` message with a snapshot if necessary. But what does this snapshot consists of?

```
type Snapshot<'state> =
  { Stable: 'state
    StableVersion: VTime
    LatestVersion: VTime
    Observed: MTime }
```

```
let rec active crdt state ctx = actor {
  match! ctx.Receive() with
  | Reset(nodeId, snapshot) when Version.compare snapshot.StableVersion state.LatestVersion =
        let state =
          { state with
              Stable = snapshot.Stable
              StableVersion = snapshot.StableVersion
              LatestVersion = Version.merge state.LatestVersion snapshot.StableVersion
              Observed = MTime.merge snapshot.Observed state.Observed }
        return! active crdt state ctx

  | Reset(nodeId, snapshot) -> return! active crdt state ctx // ignore
  // handlers for other messages
```

Since `Reset` can be used by a new replica just joining a system (it can be also used when combined with persistence to reinitialize current replica state from persistent store), we need to supply not the state alone, but also all of the corresponding context - most notably stable version describing the timestamp of state being sent and a matrix clock which describes a timespace of known replicas, so that our receiver will be able to make its own progress in gradually advancing what the upcoming stability point may be.

Also keep in mind that when a `Reset` is incoming, we should check if it's still actual on the receiver side - we use asynchronous message passing, so between request/response roundtrip we might have got another update from other replicas. The open question here is *what will happen if the `Reset` that we got has stable version which is concurrent to our own?* It's a sign of either bug or malicious behavior from one of the replica - one of the properties of stable timestamp is that we can be sure that there are no events in flight that might be concurrent to it.

Arguably, handling the incoming replicated events on the receiver side is the most complex piece here.

```
let stabilize (state: State<'state,'op>) =
  let stableTimestamp = MTime.min state.Observed
  let stable, unstable =
    state.Unstable
    |> Set.partition (fun op -> Version.compare op.Version stableTimestamp <= Ord.Eq)
  (stable, unstable, stableTimestamp)

let rec active (crdt: PureCrdt<'state,'op>) (state: State<'state,'op>) (ctx: Actor<Protocol<'s
  match! ctx.Receive() with
  | Replicated(nodeId, ops) ->
    let mutable state = state
    let actual = ops |> Set.filter (fun op -> Version.compare op.Version state.LatestVersion >

    for op in actual do
      let observed = state.Observed |> MTime.update nodeId op.Version
      let obsolete = state.Unstable |> Set.exists(fun o -> crdt.Obsoletes(o, op))
      let pruned = state.Unstable |> Set.filter (fun o -> not (crdt.Obsoletes(op, o)))
```

```
        state <- { state with
                    Observed = observed
                    Unstable = if obsolete then pruned else Set.add op pruned
                    LatestVersion = Version.merge op.Version state.LatestVersion }

      let stableOps, unstableOps, stableVersion = stabilize state
      let state =
        if not (Set.isEmpty stableOps) then
          // advance with stable state
          let stable = crdt.Apply(state.Stable, stableOps)
          { state with Stable = stable; Unstable = unstableOps; StableVersion = stableVersion }
        else
          state

      let state = refreshTimeout nodeId state ctx
      return! active crdt state ctx
```

So, again we need to start by filtering out events that might be behind local version by the time they arrived to the receiver. Then we update our observation (matrix clock) with event's own version and eventually we also need to check if incoming events are not obsoleted by the unstable events we already received and vice versa.

The final step of our `Replicated` handler is about stabilizing the state - once we got new events from another replica, we can improve our knowledge about what other events has this replica received/observed, thanks to the vector clocks attached to incoming events. With that knowledge we can set new stable timestamp and use it to filter out all events we considered unstable so far and promote them to a stable set. Finally we can apply them to our new state and forget about them. If you use persistent store here, this is a step when you want to store new system snapshot and delete (now stable) events.

Pure CRDTs implementations

With a replication protocol defined, now it's the time to talk about actual CRDT implementations. First, in order to nest them, we need some common API between our data types and replication protocol. As you've might already seen it, it can be condensed to few methods:

```
type PureCrdt<'state, 'op> =
  abstract Default: 'state
  abstract Obsoletes: Event<'op> * Event<'op> -> bool
  abstract Apply: state:'state * operations:Set<Event<'op>> -> 'state
```

Default (zero) state is quite obvious as there's no single initialization point in system and all nodes can execute updates independently from each other.

We also already mentioned that obsolete method is used to check if either replicated or unstable events are redundant in result of new event replication/emission. Some pure CRDT definitions never

obsolete their own events - don't conflate stable with obsolete events.

We also mentioned state application - here we're passing entire non-empty set of unstable events at once. In most cases it'll boil down to `Set.fold`, but in case of registers we need awareness of other unstable events during application. We're using in-memory sets here, but if you want to make it persistent, the event log could be also presented as an asynchronous sequence for purpose of this method.

As you'll see implementing actual CRDTs using this API leads to extremely concise code, all thanks to the guarantees we're able to provide at protocol level.

Counter

In case of our counter both our state and operation is just a simple number (positive or negative) and all we need to do is to sum all unstable operations on top of state to get a counter value.

```
let counter =
  { new PureCrdt<int64, int64> with
     member this.Default = 0L
     member this.Obsoletes(o, n) = false
     member this.Apply(state, ops) =
       ops
       |> Seq.map (fun o -> o.Value)
       |> Seq.fold (+) state }
```

Since counter updates are always independent, they never obsolete each other.

Last-Write-Wins register

Another structure is Last-Write-Wins register. Here our state is an option of contained type (as we start with uninitialized value), while operation is an instance of this type.

```
let inline comparer (o: Event<_>) = struct(o.Timestamp, o.Origin)

let lwwreg =
  { new PureCrdt<'t voption, 't> with
    member this.Default = ValueNone
    member this.Obsoletes(o, n) = comparer o > comparer n
    member this.Apply(state, ops) =
      let latest = ops |> Seq.maxBy comparer
      ValueSome latest.Value }
```

We take advantage of `Event<>` fields defined before, which carried over system timestamp. Now we can use it to determine which operation is "the most recent one". Since we cannot exclude risk, that that two different events will have the same timestamp, we additionally use origin field - ID of replica,

where operation was submitted - to distinguish them, effectively making our timestamp look like Lamport Clock.

Just as we mentioned previously, assignment event obsoletes older ones if its timestamp (see: `comparer` function above) is higher than any other.

Multi-Value register

Multi-Value Registers seems to be ideal candidates for our implementation here, as they are always returning all concurrently updated values - which in our case is simply an unstable set!

```
let mvreg =
  { new PureCrdt<'t list, 't> with
    member this.Default = []
    member this.Obsoletes(o, n): bool = Version.compare n.Version o.Version <= Ord.Eq
    member this.Apply(state, ops) =
      ops
      |> Seq.map (fun o -> o.Value)
      |> Seq.toList }
```

Here we can also obsolete older events, but unlike in the case of Last-Write-Wins register, here we define "older" in context of logical time (vector clocks) rather than physical one. Keep in mind that `Apply` method here doesn't take old state into account - since we made sure that unstable operations are never empty, they always represent elements assigned later than stable state and therefore override it.

Add-Wins Observed Remove set

For pure OR-Set, we can focus on two basic operations - add and remove element from the set. Application of unstable operation is again very simple: we just add an element to the set when we see `Add` operation, and remove it when `Remove` was detected.

```
type Operation<'t when 't:comparison> =
  | Add of 't
  | Remove of 't

let orset =
  { new PureCrdt<Set<'t>, Operation<'t>> with
    member this.Apply(state, ops) =
      ops
      |> Set.fold (fun acc op ->
        match op.Value with
        | Add value -> Set.add value acc
        | Remove value -> Set.remove value acc) state
    member this.Default = Set.empty
    member this.Obsoletes(o, n) =
      match n.Value, o.Value with
```

```
    | Add v2, Add v1 when Version.compare n.Version o.Version = Ord.Lt -> v1 = v2
    | Add v2, Remove v1 when Version.compare n.Version o.Version = Ord.Lt -> v1 = v2
    | Remove _, _ -> true
    | _ -> false }
```

Obsolete mechanism is about checking if two operations apply to the same logical element and (if so) if one of them is directly behind another. So `Add(t1, a)` will obsolete `Add(t0, a)` if `t0 < t1` because we cannot add the same element twice - so we'll just keep the more recent operation in our unstable set.

For removals we preserve add-wins semantics - if two add/remove operations are in concurrent update conflict, we'll prefer to `Add` an element. What's worth noticing here, our `Remove` operation will never really be present in unstable set. Since we obsoleted older operations by more recent ones, the only remaining ones in our unstable set could be concurrent updates - and as we said in case of any concurrent conflict we always prefer `Add` over `Remove`, hence we can get rid of `Remove` all together. The only exception from this rule is a unstable set with a single `Remove` element.

Indexed Sequence

We'll finish discussion about pure CRDTs with Indexed Sequence implementation. It's hard to say if it's really an LSeq, RGA or any other known data structure, as it doesn't really bear any characteristic attributes of original non-pure implementation nor was it defined in original paper.

Reminder: *what differs indexed sequences from sets is that they allow users to specify order in which elements are to be inserted rather than relying on natural order of comparable elements*. For this reason, we need to be able to resolve conflicts like different replicas inserting/deleting different values at the same index concurrently.

Thankfully, the pure-based variant is even simpler than OR-Set here. Just like in case of OR-Set, operation application is just straightforward insert/remove action done when a corresponding operation is applied.

```
type Operation<'t> =
  | Insert of index:int * value:'t
  | Remove of index:int

let iseq =
  { new PureCrdt<'t[], Operation<'t>> with
    member this.Apply(state, ops) =
      ops
      |> Set.fold (fun acc op ->
        match op.Value with
        | Insert(idx, value) -> Array.insert idx value acc
        | Remove idx -> Array.removeAt idx acc) state
    member this.Default = [||]
    member this.Obsoletes(o, n) = false }
```

Few important notes here:

1. Obsolete method is not necessary here, as we didn't defined any update method over existing elements. Since unstable event log grows or shrinks dynamically as replication is in progress we don't really want to obsolete removals, as they may be needed later on to undo inserts.
2. What's crucial here is the characteristic of an unstable set (`ops` parameter) - in our case it's a set of events ordered by vector version, then timestamp, then origin. This is the reason why we don't need virtual pointers, that we used in LSeq implementation in the past. Order of events in unstable set will eventually be the same on all replicas. When applying events over stable state, we keep that order. This allows us to guarantee that we won't mess the indexes of concurrent updates (represented by unstable set itself).

Keep in mind that above implementation is not free of interleaving issues, which may be problem is some use cases like collaborative text editing.

Eviction

We talked a lot about strong sides of pure operation-based CRDTs so far. But if it's so great, why are they so uncommon, so rarely used in real systems? There's a one hard problem, they introduce. Ability to determine stability point may not require explicit coordination, but it does require nodes to communicate with each other. **If at least one node in the cluster goes offline and never gets back up, we're no longer able to stabilize our state**.

Unfortunately this is quite common scenario for applications working at the edge: which is one of the most popular use cases for having CRDTs. Moreover problem of detecting dead nodes (like permanently dead ones) is hard to automatize without introducing risk of data loss in cases when our assumptions about dead node were invalid.

Here, we'll shortly cover the possible workaround that is node eviction (AFAIK it was not described in any paper). We're going to send a signal to inform that a given node no longer takes part in a replication process, and from now on, all of its concurrent updates will be ignored unless its willing to sync by reseting its state. This is something that could be done manually by an operator (eg. moderator of chat removing inactive users) or automatically (evict nodes that didn't respond within a given timeout). Keep in mind that trying to automate this process may be risky and not possible at all in some cases - while it seems doable for cross-datacenter systems, edge devices often cannot provide required properties to make this approach practical.

Here we're going to extend protocol we implemented with two extra messages: `Evict` and `Evicted`. Here we assume approach where evict is being send to a replica explicitly from outside. Actor, which received it is going to create an `Evicted` message with new vector timestamp and broadcast it to all other connected replicas. It will also remember which node was evicted at what timestamp.

```
let broadcast msg state =
  for e in state.Connections do
    (fst e.Value) <! msg
```

```
let rec active crdt state ctx = actor {
  match! ctx.Receive() with
  | Evict nodeId ->
    let version = Version.inc state.Id state.LatestVersion
    let msg = Evicted(nodeId, version)
    broadcast msg state
    let connections = terminateConnection nodeId state
    let observed =
      state.Observed
      |> Map.remove nodeId
      |> Map.add state.Id version
    let state = { state with
                  LatestVersion = version
                  Evicted = Map.add nodeId version state.Evicted
                  Observed = observed
                  Connections = connections }
    return! active crdt state ctx
  // other handlers
```

Why do we need to generate a new vector version? We're going to use it as a boundary. When it will be received on other nodes we have 3 cases to handle for incoming or unstable events (we can ignore equality as no event timestamp will be equal to it):

- Events that happened before this eviction timestamp will eventually be applied into stable state. We cannot really prevent that as different nodes stabilize at different rates.
- As a tradeoff we'll also accept and eventually apply events which happened directly after eviction version. It represents a scenario, when a node was evicted, resets its state and joined again. This comes with a risk - we'll discuss it in a minute - but it also lets us reuse replica identifiers. Otherwise we'd need a new replica ID every time node is evicted, and that could cause an explosion of vector clock size.
- The last case is when incoming events emitted by evicted node come with timestamps concurrent to evicted version itself. This means that we received an event from a node, which didn't yet know it was evicted. Here, we're going to drop these events. This case comes with inherent risk of data loss - we accepted an unstable event, but later revoke it.

First let's consider what would happen when node is informed about its own eviction:

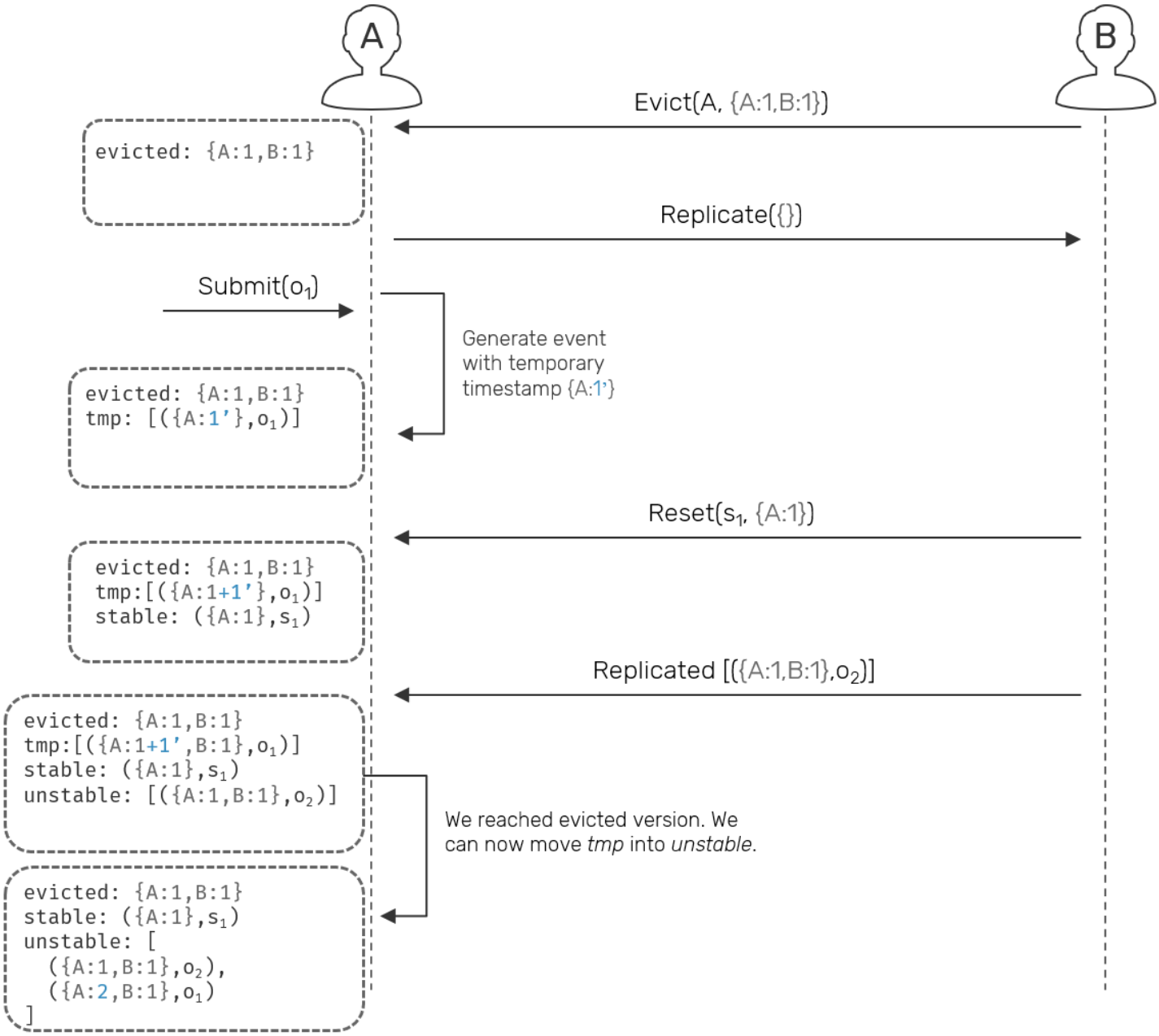
```
let rec active crdt state ctx = actor {
  match! ctx.Receive() with
    | Evicted(evicted, version) when evicted = state.Id ->
    // evacuate...
    return Stop // stop current actor
```

We didn't really described "evacuate" section, as it's very context specific. How do you want to handle eviction? Maybe redirect evicted events aside, so that later - after node will reset its state - they could

be reviewed and resubmitted again (as new operations). If you persisted a node state on disk, you may want to do some cleaning there as well.

What's important to remember is that - since we allowed node to reconnect if it agreed to reset its state - that there's a structural weakness here: if node recovers after eviction, its latest vector version must take into account sequence number of its ID included in evicted version - it cannot start from 0. Example: *node B evicts A at time* `{A:1,B:2}`. *Node A resets its state but before its synchronized, user submitted an operation. If A after reset starts with an empty version, a newly submitted operation will have timestamp* `{A:1}` *which potentially may corrupt the state of a system, as other nodes might already seen such timestamp in the past: it's no longer unique.*

Is it possible to keep previously evicted node writeable even before it updated its own state up to a given version? Yes, we could i.e. assign a temporary timestamps (eg. `{A:1'}`, `{A:2'}` *...) to locally submitted operations. These temporary events can be presented to local user, but are forbidden to replicate to remote replicas until local node synchronizes its own state up to evicted timestamp (eg.* `{A:1,B:2}` *). As new events arrive from remote nodes, we update the latest version and rewrite events with temporary timestamps on top of it (eg. latest will equal:* `{A:1,B:1}`*, with temporary version:* `{A:1'}` *→ observed temporary event version:* `{A:2,B:1}` *). Once the latest version reached evicted timestamp, local replica can be considered synchronized and its temporary events can now get actual version and be send outside.*



This is an idea for an eviction extension to an original tagged reliable causal broadcast protocol. We won't dive deeper than the diagram above, instead we'll focus on basic eviction process.

For other nodes, when they notice eviction, all we really need is to remember evicted node and remove unstable events originating from evicted node, which versions are concurrent to evicted timestamp.

```
let isEvicted version e =
  e.Origin = evicted && Version.compare e.Version version = Ord.Cc

let rec active crdt state ctx = actor {
  match! ctx.Receive() with
  | Evicted(evicted, version) ->
    let connections = terminateConnection evicted state
    let unstable =
      state.Unstable
      |> Set.filter (fun e -> not (isEvicted version e))
    let state = { state with
```

```
                    Evicted = Map.add evicted version state.Evicted
                    Observed = Map.remove evicted state.Observed
                    Connections = connections
                    Unstable = unstable }
    return! active crdt state ctx
```

We also need to modify our `Replicate` / `Replicated` cycle a little. If we received `Replicate` request from evicted node with timestamp concurrent to eviction version, it means that the request was send before a node knew about being evicted. It's possible that the message never arrived to it eg. due to network failure. So what we're going to do is to simply resend it an `Evicted` signal.

```
let rec active crdt state ctx = actor {
  match! ctx.Receive() with
  | Replicate(nodeId, filter) ->

    match Map.tryFind nodeId state.Evicted with
    | Some version when Version.compare filter version = Ord.Cc ->
      // inform node about it being evicted
      ctx.Sender() <! Evicted(nodeId, version)
      return! active crdt state ctx

    | _ -> // standard handle we defined before

  | Replicated(nodeId, ops) ->
    let mutable state = state
    let evictedVersion =
      Map.tryFind nodeId state.Evicted
      |> Option.defaultValue Version.zero
    let actual =
      ops
      |> Set.filter (fun op ->
        Version.compare op.Version state.LatestVersion > Ord.Eq &&
        not (isEvicted evictedVersion op)) // prune evicted events
    // rest of replication logic defined before
```

For `Replicated` response (which could be send concurrently to eviction), we'll simply remove all replicated events that should be evicted.

Take into account all warnings we talked about here. While far from ideal, this process offers an alternative for ever-growing unstable event log in face of disconnected or potentially dead nodes. Decision if this is a tradeoff worth taking depends on actual system at hand.

CRDT optimizations

This time let's discuss various optimizations that could be applied to CRDTs working at higher scale.

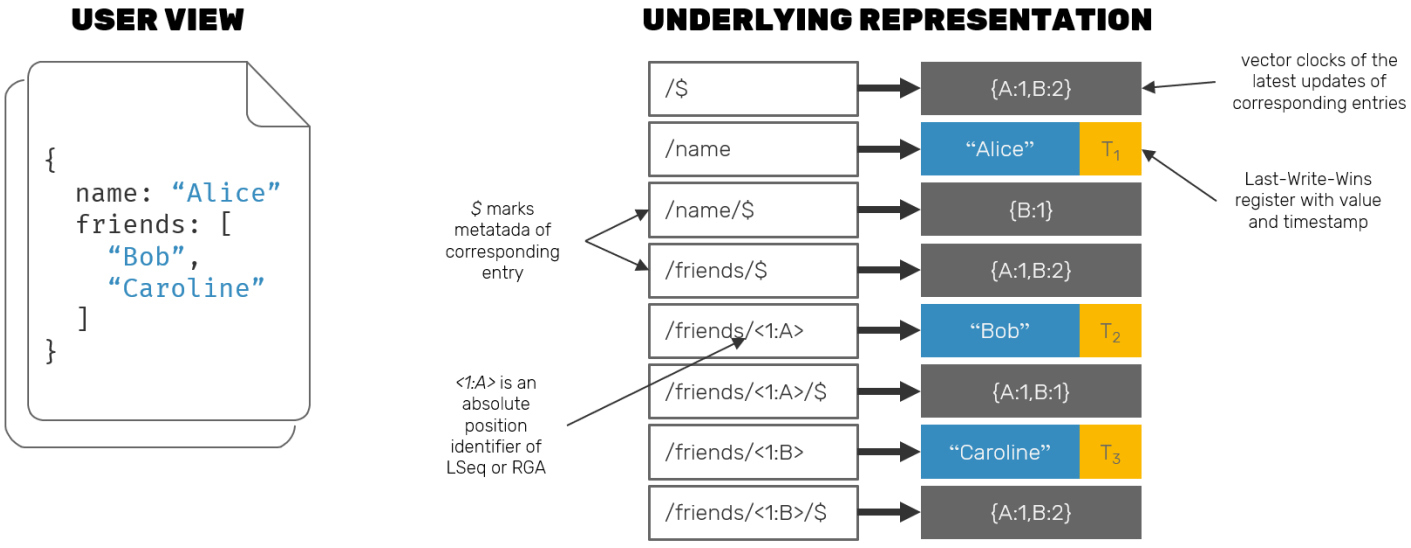
Since we're talking about optimizations for structures and protocols we already described in the past, a prerequisite here is to have an understanding of things like convergent maps/documents, principles of delta- and operation-based protocols and good understanding of vector clocks. Preferably you also want to have some general knowledge of durable data stores (like B+Trees and LSM trees) and database indexes.

Larger than memory CRDTs

Let's start with optimization that it's not really CRDT specific. All data structures we covered so far, had simple in-memory representations. Making them persistent sounds easy - just serialize/deserialize them on demand - but the problems starts when this process is expected to be efficient and working for high volumes of data.

When we're talking about a huge structures - the ones that could potentially not fit into memory - it's kind of obvious that we're need a way to store them in segments that are easy to navigate on disk. *Most of the on-disk structures, like B+Trees or LSM trees, are exposed to users in form of a simple transactional, ordered key-value store. These key-value pairs are atoms used for data navigation: you set cursor position by seeking for a given key (or prefix of a key) and then are free to move forward over all entries with keys lexically higher than current cursor pointer, usually at pretty low cost since they involve sequential disk reads. Usually these entries are read as whole - no matter what is the size of value stored in a given entry, since its most atomic location unit exposed by the data store, it will be read as whole.*

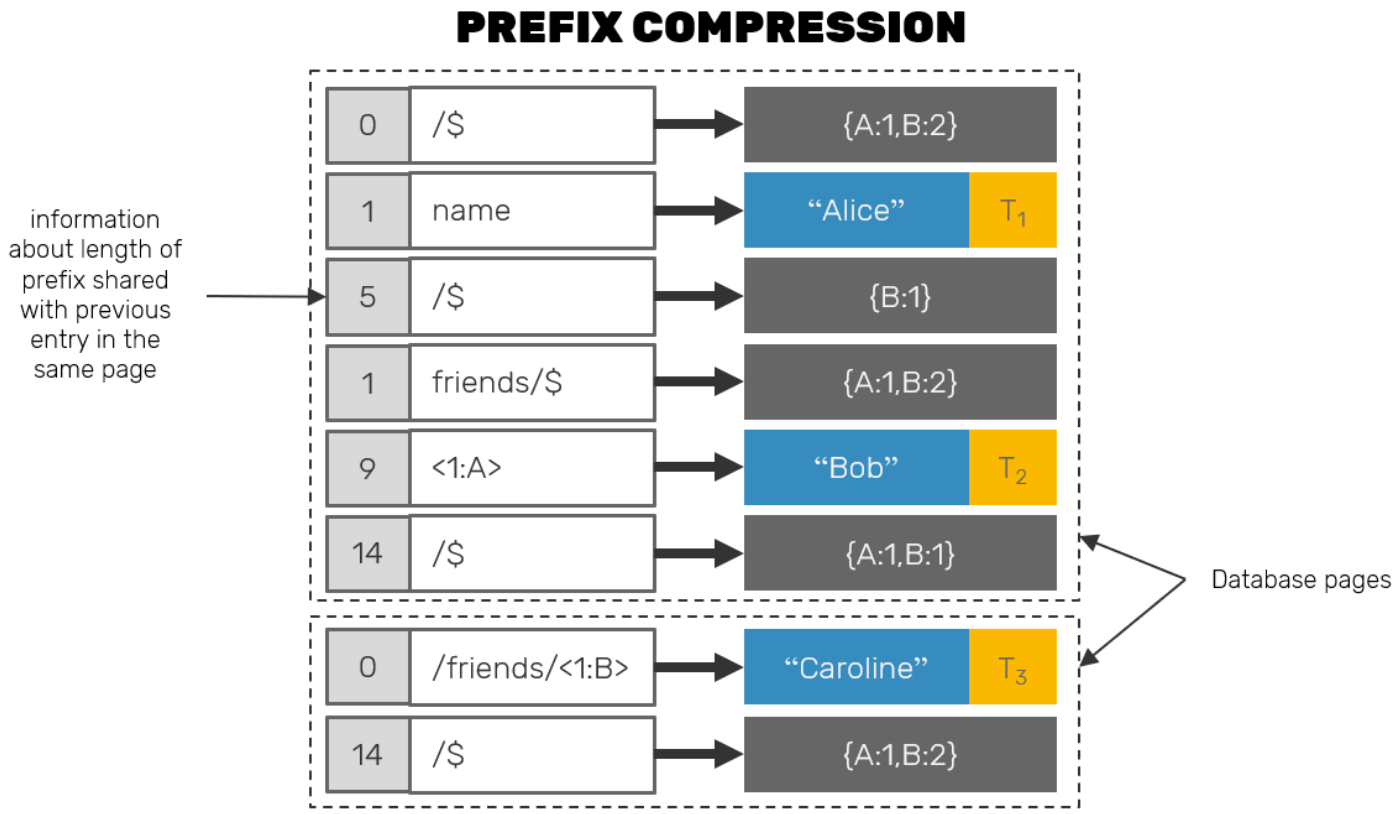
Issue of state growing beyond memory capacity was a common concern for many mature higher level database offerings using key-value data stores underneath. Solution for the problem is to split a logical data type (like SQL row, sets or even whole documents) onto multiple key-value pairs.



But won't this approach backfire in situation, when we'd like to return a document composed of many fields? Keep in mind that while random disk seeks may be moderately expensive, it's not the case here. As document's fields are laid out in order, reaching out for them is just a matter of doing sequential reads from disk, which are much faster. Additional advantage is that while at the API level,

we're iterating the cursor over key-value entries, at the "physical" layer neighboring entries are read together from disk in pages - which can vary in size, but usually match the allocation units (4-8kB is pretty much the norm) for an underlying hardware to make writes atomic - furthermore reducing the cost of this approach.

Another question to answer is: if each key consists of the entire path in the document, starting from the root, won't this mean, that eventually the disk representation of the document will consists mainly from the keys alone? It could be true, but in practice many of the disk storage engines are prepared for this use case and use a technique called **key prefix compression**:



As we mentioned, entries are written and read from the disk in blocks. Now, before an entry key is stored, the engine itself checks how many bytes are shared between that key and a key of preceding entry within the same block. Then it stores that number as a prefix of the key itself, while only writing down the differentiating part of the key. Example: if entry `(romulus => A)` is about to be stored in a block right after entry `(rome => B)`, we could as well write it as `([3]ulus => A)` where `[3]` is a number of shared bytes with key of its predecessor.

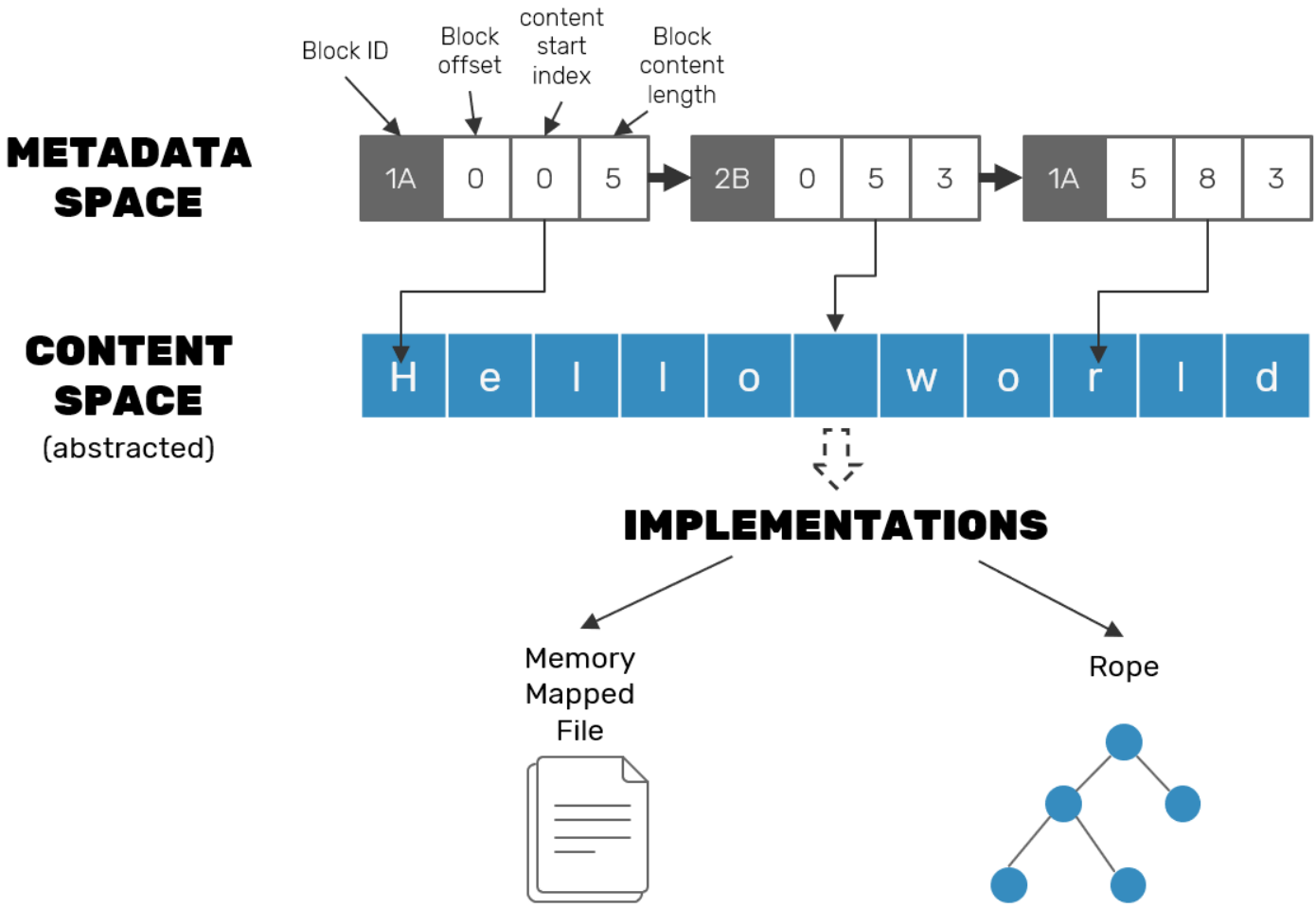
Split-wise Replicated Growable Array

Back in the past we talked about optimizing indexed sequences using block-wise (or split-wise) RGA algorithm. As collaborative text editing is one of the most common places where such structure could take a place, it's important to make it as fast and responsive as possible.

You've might noticed that our PoC was far from perfect in that sense - we simply stored blocks of inserted text as arrays, which we had to split when user wanted to insert new block in the middle of

old one, and we also reconstructed the whole buffer every time we wanted to present a view of our array to the user. All of this could become expensive. Can we mitigate that? It turns out we can.

If you take a closer look at the implementation details of block-wise RGA algorithms, you may notice that CRDT specifics usually depend on the metadata alone - we are usually only interested in the lenght of the content rather than content itself. In reality we could abstract it into a separate structure optimized for text manipulations (like Rope or string buffers of specific editors), while RGA-specific metadata only holds ranges that given block is responsible for.



On the other hand if we're about to make our content space aware of CRDT metadata, we could do optimizations like zero-copy split operations - splitting the block to insert a new one inside of it can be done solely on metadata nodes by shifting the indexes they point to, without any changes in the underlying string buffer. While this makes writes faster (they can be append-only) it comes at the price of making random-access on reads: we need to use block metadata info in order to know which parts of content store need to be read in what order.

The same idea can also work for deletions. In such case we first do a logical deletion, while a separate vacuum process can deal with physical removal and reordering of string fragments that are no longer in use.

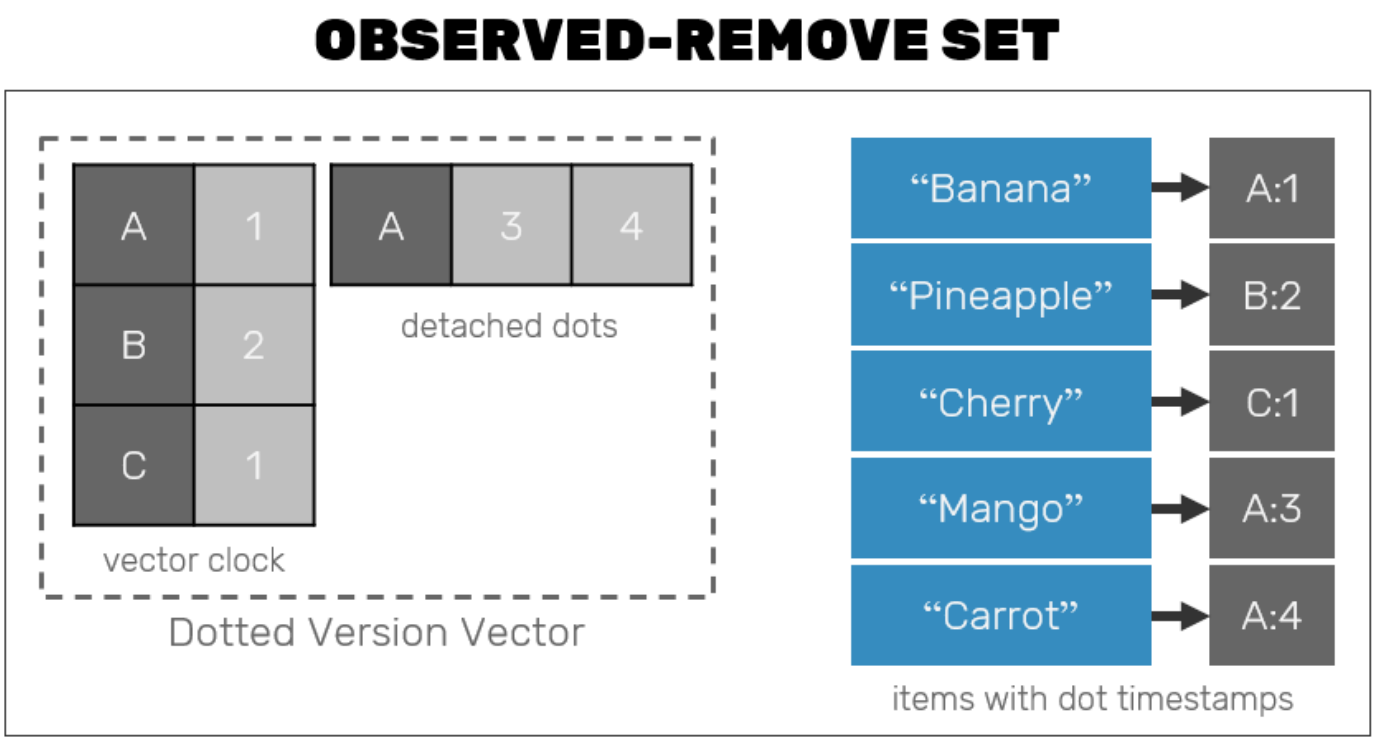
More efficient delta-state propagation

In the past, we covered the case of state-based CRDTs. We also talked about the issue with passing a huge state objects between the peers to provide necessary eventual consistency, and how to mitigate these issues with deltas. Now, let's talk about what deltas do and don't solve.

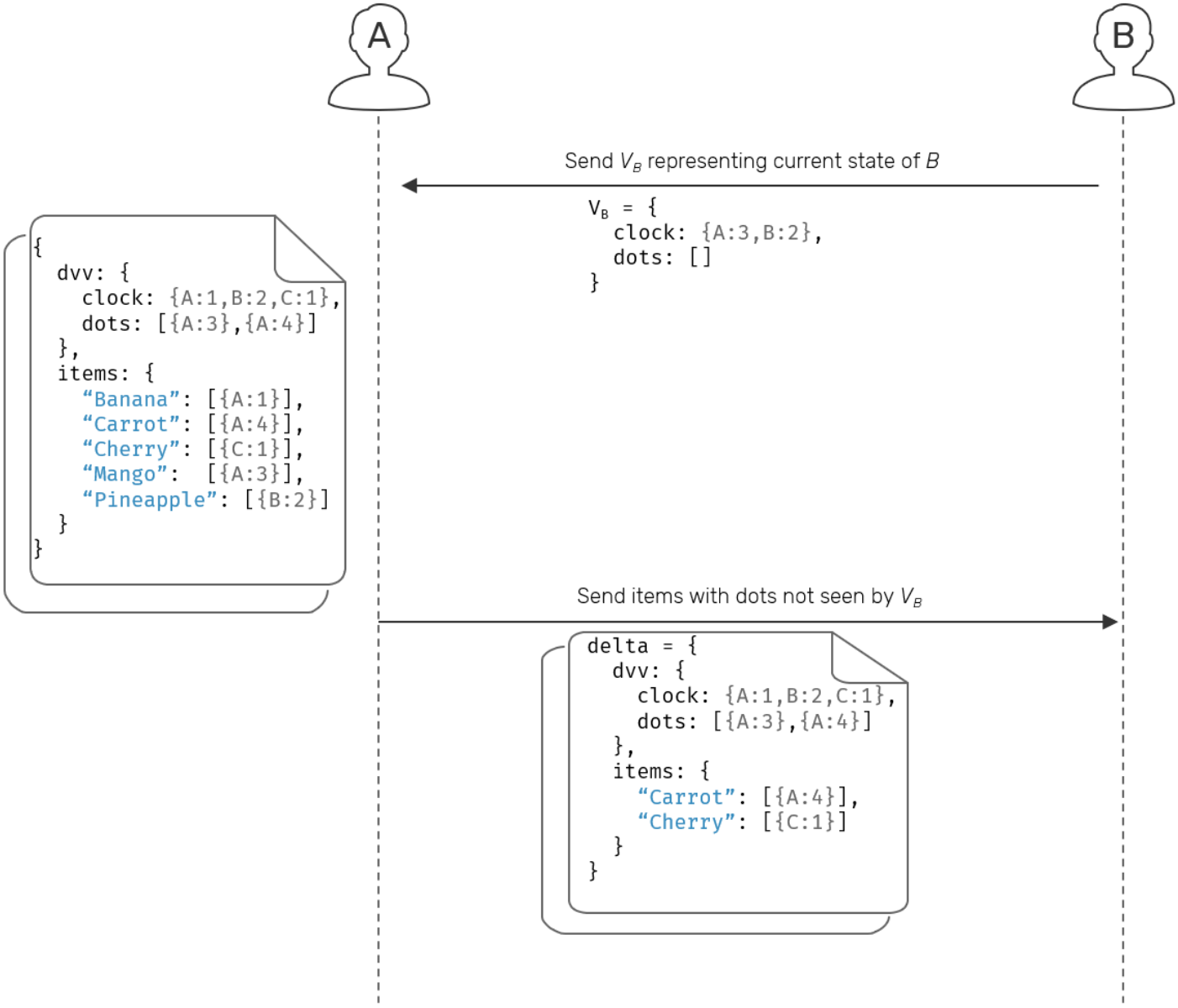
We didn't really mentioned the guarantees of a replication protocol in case of deltas, as it wasn't really that necessary. We only mentioned that in the case when the delta-state is lost, we can make a full CRDT synchronization from time to time. What if that's not possible? What if our "full" state is a multi-terabyte collection of data?

Let's approach this issue using our delta-state OR-Set as an example. When we developed it in the past, we constructed it from 2 parts:

1. A dotted vector version that described a "current time" as observed by our collection. This logical clock contained a compressed information about all dots (used to uniquely tag a single set insert operation) captured by the set.
2. A map of `(value, dots)` where dots where unique IDs of insert operations (the same element could be inserted multiple times) that caused a value to appear in the set.



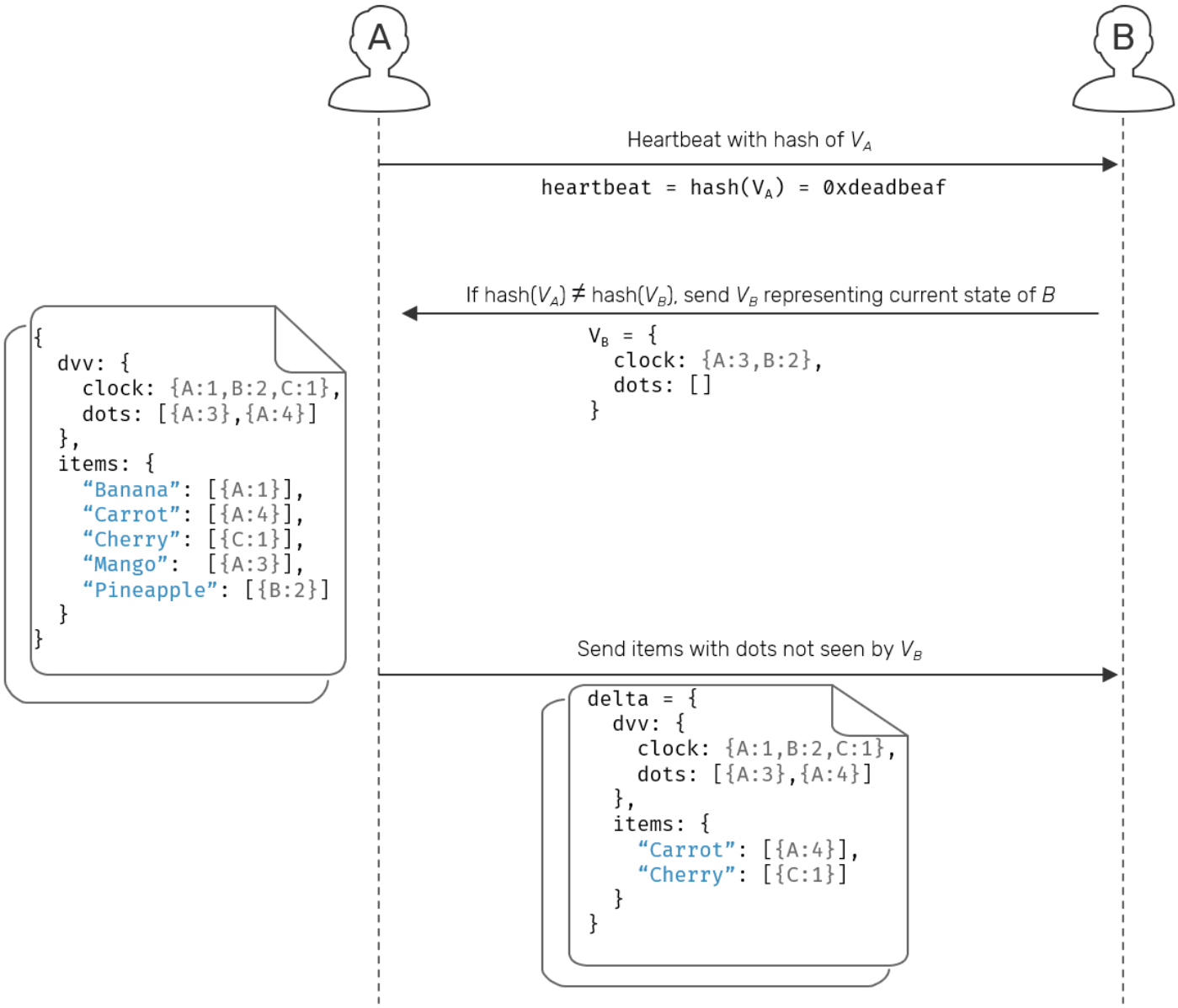
In our implementation we generated new dots only for inserts - we were able to do conflict resolution based on a fact, that if insert dot was present in dotted vector version (1) but not in a result map (2), then it must mean that this value was removed. This however is not a strict rule. We can generate dots for removals as well - in the past approach it was just wasteful, however now we're be able to utilize that extra information to decide if state of our set had changed based on the state of dotted vector version alone! But why? Since dotted version vector has compressed representation and its size is not dependent on the size of the elements stored in OR-Set, we can pass delta of version vector alone instead to signal to other replicas that our local state has changed!



Now, instead of one step replication, we can change it into request-response pattern. When looking for or notifying about updates, our replica is first going to pass a dotted version vector. Since it's usually smaller that the actual state of the set, the cost of that request is fairly low.

Another side can use this information to produce a diff between its own local version and incoming one, and use it to compute the delta-state that's exactly matching missing updates on the other side. This is possible, because version vectors contain identifiers of all update operations, while items are tagged with dots themselves. If we have the diff between local/remote DVV, all we need now is to gather all elements with dots found in that diff.

What's more interesting, we can go even further. We don't need to pass the versions all the time. All we really need to start the procedure is a simple digest (consistent hash) of the version vector itself.



While this adds an extra network call to our replication process, the initial payload is so small that it could be as well piggybacked on top of i.e. heartbeat messages used to identify alive and dead nodes, which makes easier to determine when other peers have new data to share.

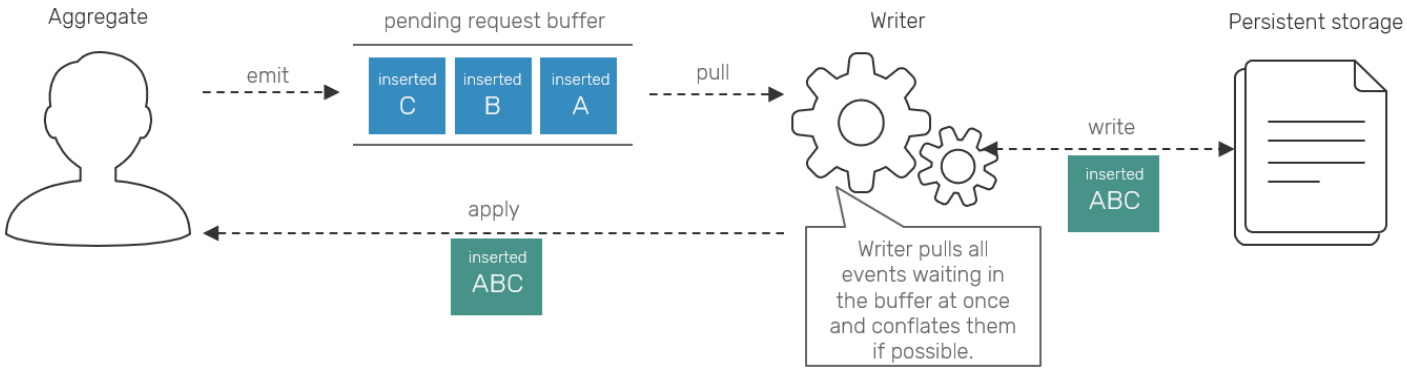
Associativity of events in operation-based CRDTs

Here we'll cover some ideas on how to optimize operation-based CRDTs. As we mentioned in the past, the core principle of op-based CRDTs is commutativity - since total ordering is not possible to maintain in system with multiple disconnected writers, we need this property to ensure that partially ordered events can be applied in different order.

While the associativity property was not required in operation-based variant (only in state-based one), we could also find improvements by applying this property to our events.

Associativity in this context means that we can conflate our events and apply their conflated representation to produce the same results, as if we'd apply them one by one. The obvious case for this is eg. counter incrementing. If we're not constrained with showing intermediate results - like

when we use counters as sequencers, which I strongly discourage you to use CRDTs for anyway - we can assume that series of operations like `[Inserted('A'), Inserted('B'), Inserted('C')]` emitted in a row could be as well represented as a single `[Inserted('ABC')]`.



The optimization here comes from the fact, that under Reliable Causal Broadcast, each separate event requires a bunch of metadata and persistence prior to replication/applying. Quite common technique of working with requests incoming at higher rate than they're stored with is to buffer them and perform any necessary I/O operations on buffered chunks of data rather than small pieces alone.

We apply a very similar principle here, but at application logic level. Since the upstream and downstream can work at different rates, we replaced our event buffering with conflation: if upstream is producing events at the higher rate, they are conflated together into single one and emitted when downstream is ready. This approach plays well together with workflow oriented approach and backpressure mechanisms.

Keep in mind that while in context of CRDTs this is usually an applicable solution it may not work in more general business contexts. Oftentimes we want to distinguish events from each other on the logical level eg. bank transfer transactions. This sort of associativity may cause loss of business context information, so as always terms and conditions apply.

Indexing vector clocks

One of the issues with vector clocks is that they are not supported very well by present day data stores. Even while some of databases are using them internally, they are rarely exposed to developers and it's even less common that they have first class support for operations like indexing.

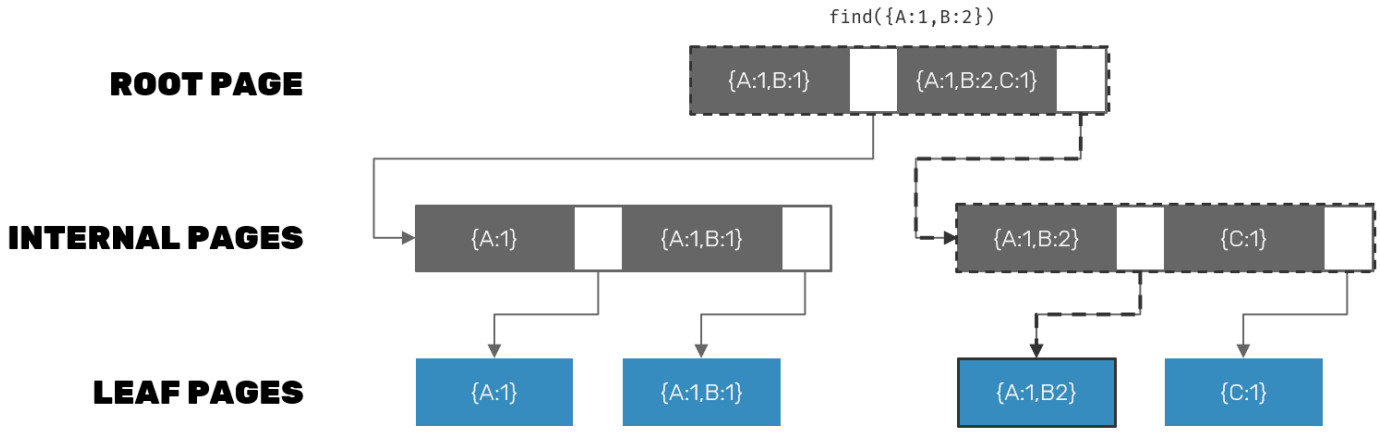
Some databases (like Postgres) have defined great extensibility points for the programmers to inject their own custom logic into DB runtime. One of the experiments, I'm working on is to add indexable vector clock support in PostgreSQL. If this succeeds, it will probably deserve it's own chapter, but here let's try to answer the question: since vector clocks don't comply to standard ordering rules (because they can be only partially ordered), can we even build indexes on top of them?

Behind Postgres GiST indexes

Generalized search trees (GiST) are one of the index types available in PostgreSQL. Their capabilities are implemented in many native PG types like timestamp ranges, spatial data or full text searching. Here we won't go really deep inside of how they work. If you're interested and want to read more, I think this blog post is great for start.

The basic principle we are interested in is that Postgres let's us to define our own GiST compatible data type utilizing R-tree indexes. The idea behind them is that we split an N-dimensional space objects into smaller slices. Then we recursively split them further into even smaller blocks, until all indexed values contained within a given block will fit into a single database page. Out of all of these elements we build a multi-level tree with with higher levels (containing bigger pieces) descending into lower levels with smaller pieces.

While spatial indexes are the most visual representation of such approach, it is really generalizable into any N-dimensional space. And how can we represent coordinates in N-dimensional space? Using vectors!



We can translate vector clock partial comparison states into operators supported by Postgres:

- If `t1` is less than `t2` we can say `t1` **is contained by** `t2` → `t1 <@ t2`.
- If `t1` is equal to `t2`, we say they are the **same** → `t1 ~= t2`
- Greater case is analogous to less, so `t1` **contains** `t2` → `t1 @> t2`.
- Finally, we can say that two concurrent vectors are concurrent when they **overlap** → `t1 && t2`

Additionally building GiST index requires defining some functions to be integrated into Postgres internals (you need a language that offers foreign function interface with C for that):

- `union` which composes higher-level key out of lower level ones. We can implement it simply in terms of merging vector clocks.
- `equal` to check if keys are equal. We already have that as part of partial comparison.
- `consistent` to check if given index key satisfied the query/operator: in this case partial comparison operators we defined above.
- `penalty` used to determine where it's least expensive to insert a new vector clock, given several possible pages that could fit this role.

- `picksplit` used when given index page is getting too big - in that case we need to split it in two and determine which vectors are about to go into which page.
- `compress` / `decompress` functions which allow us to use different data type to be used to represent index key - this may be useful to preserve space. For example: `{A:2,B:1}` may be good notion for a single vector clock but for many of them keeping IDs (`A`, `B`) is redundant. Instead we could have one global mapping for ID→index (`[A,B]` which tell us which replicas are mapped to which indexes), and represent each vector clock as an array with sequence numbers maching indexes of global mapping (eg. `[2,1]`), resulting in less disk space consumption.

The major challenge here is `picksplit` function implementation, as same-level vector placement will directly affect ability to narrow search space of an index. Fortunately many algorithms have been implemented for efficient space partitioning of R-Trees and some of them should be generalizable for vector clocks as well.

Faster vector clock comparison

The last optimization, I wanted to mention, is related to comparison operations over vector clocks. In many libraries vector clocks are implemented using maps. While this seems natural, it also comes with a cost - maps are not the most optimal things when it comes to efficient iteration over their elements, and this is exactly what we do when we want to compare them.

There's also other observation that we can take for our advantage - once a membership of our distributed system stabilizes, peer identifiers of our vector clocks stay the same, so do positions of their sequence numbers. We can optimize for that case by introducing SIMD operations.

The idea is simple - we keep our peer ids in one array, and their corresponding sequence numbers in another one. During comparison (but also potentially merging) we first check if peers of two vectors are equal - which in stable clusters should happen >99% of the time. If they are, it means that arrays with sequence numbers of both clocks have their elements under the same indexes. This means it's safe to load them into SIMD registers right away (given sequence number of 64-bits, a 512-bit SIMD instruction set can operate on 8 of them at once) and perform multiple comparisons / merges at once. Initial peer id check can also be done using SIMD registers.

How much faster is it? In example implementation, I've got 2 orders of magnitude speedup when compared to standard AVL-tree based implementation of vector clock. For smaller vector clocks it's less impressive (SIMD registers kick in for registers over certain size, but that can also be improved), but it's still faster, mostly because arrays are more hardware-friendly data structures than trees.

List of open source CRDT projects

There are many more optimizations in micro- and macro-scale, that are scattered throughout different projects. Let's make a few honorable mentions, where you can dive into to expand your knowledge further:

- RiakDB (Erlang) which is probably the most well known open-source key-value database implementing delta-state CRDTs.
- AntidoteDB (Erlang) which is an operation-based key-value database. One of its unique traits is support for interactive transactions - hopefully we'll cover an algorithm behind them in the future.
- Lasp (Erlang) which implements many of operation-based and delta-state based CRDTs.
- Y.js (JavaScript) which is an entire ecosystem - also one of the oldest and fastest in this domain - focused on adding collaborative support into rich text editors and applications. It uses delta-state CRDTs.
- Automerge (JavaScript) which is another library in the domain of collaborative apps. It represents an operations-based approach to CRDTs.
- Akka DistributedData (Scala and C#) which is a delta-state key-value store build on top of Akka framework.
- Akka Replicated Eventsourcing (Scala) which is another library from Akka ecosystem, that seems to originate from ideas of Eventuate and offers an ability to create operation-based CRDTs on top of evensourcing paradigm - very similar to what we talked about before.
- Pijul (Rust) which is a version control system (just like Git), that uses CRDT concepts to perform conflict resolution of commits (called *patches* in Pijul) submitted by different repository contributors.

While there are many many more, I decided to focus on the projects, that were released with open source and decent level of completeness.

Summary

We've mentioned some of the optimization techniques, that can be used to make our CRDTs better in real life workloads - some of these are already used in systems like ditto.live or y.js. Others have not yet been implemented at the moment of writing this post. While I'm going to post more about about conflict-free replicated data types in the future, I won't attach them to the common table of contents - I think this post serves as a good conclusion. If you're interested, keep an eye on #crdt tag on this blog.

Delta-state CRDTs: indexed sequences with YATA

In this chapter we're coming back to indexed sequence CRDTs - we already discussed some operation-based approaches in the past. This time we'll cover YATA (Yet Another Transformation Approach): a delta-state based variant, introduced and popularized by Yjs framework used to build collaborative documents.

If you're not familiar with topic of Conflict-free Replicated Data Types, you can start learning about them here.

Foundations of indexed collection CRDTs

A specific property of indexed CRDT collection is that - unlike sets or maps, which only define presence/absence of an element - they allow us to define insert/delete operations using arbitrary order: a property that's especially vital for use cases like collaborative text editing. We already discussed two different indexed sequence CRDTs on this blog: LSeq and RGA. Now it's a time for YATA.

A core concept behind all indexed CRDTs is notion of unique identifier assigned to the inserted element - I referred to it as a **virtual pointer** in the past chapters - which allows us to track the position of that element while its actual index (as observed by user) may change whenever other elements in a collection come and go.

A popular approach is to define that unique identifier as a composite of two values:

1. Sequence number used for comparison. It varies depending on the algorithm:
   - **LSeq** uses a variable-length byte sequence which is a function `id(left_neighbor, righ_neighbor)` that's expected to produce a sequence which is lexically greater than id of an item at previous index, but lower than id of an item at the next index (`lseq[n-1].id < lseq[n].id < lseq[n+1].id`).
   - **RGA** maintains a single globally incremented counter (which can be ordinary integer value), that's updated anytime we detect that remote insert has an id with sequence number higher that local counter. Therefore every time, we produce a new insert operation, we give it a highest counter value known at the time.
   - **YATA** also uses a single integer value, however unlike in case of RGA we don't use a single counter shared with other replicas, but rather let each peer keep its own, which is incremented monotonically only by that peer. Since increments are monotonic, we can also use them to detect missing operations eg. updates marked as `A:1` and `A:3` imply, that there must be another (potentially missing) update `A:2` .
2. Unique identifier of a given replica/peer. Since we cannot guarantee uniqueness of sequence numbers generated concurrently by different peers, we can use them together with peer's own ID to make it so.

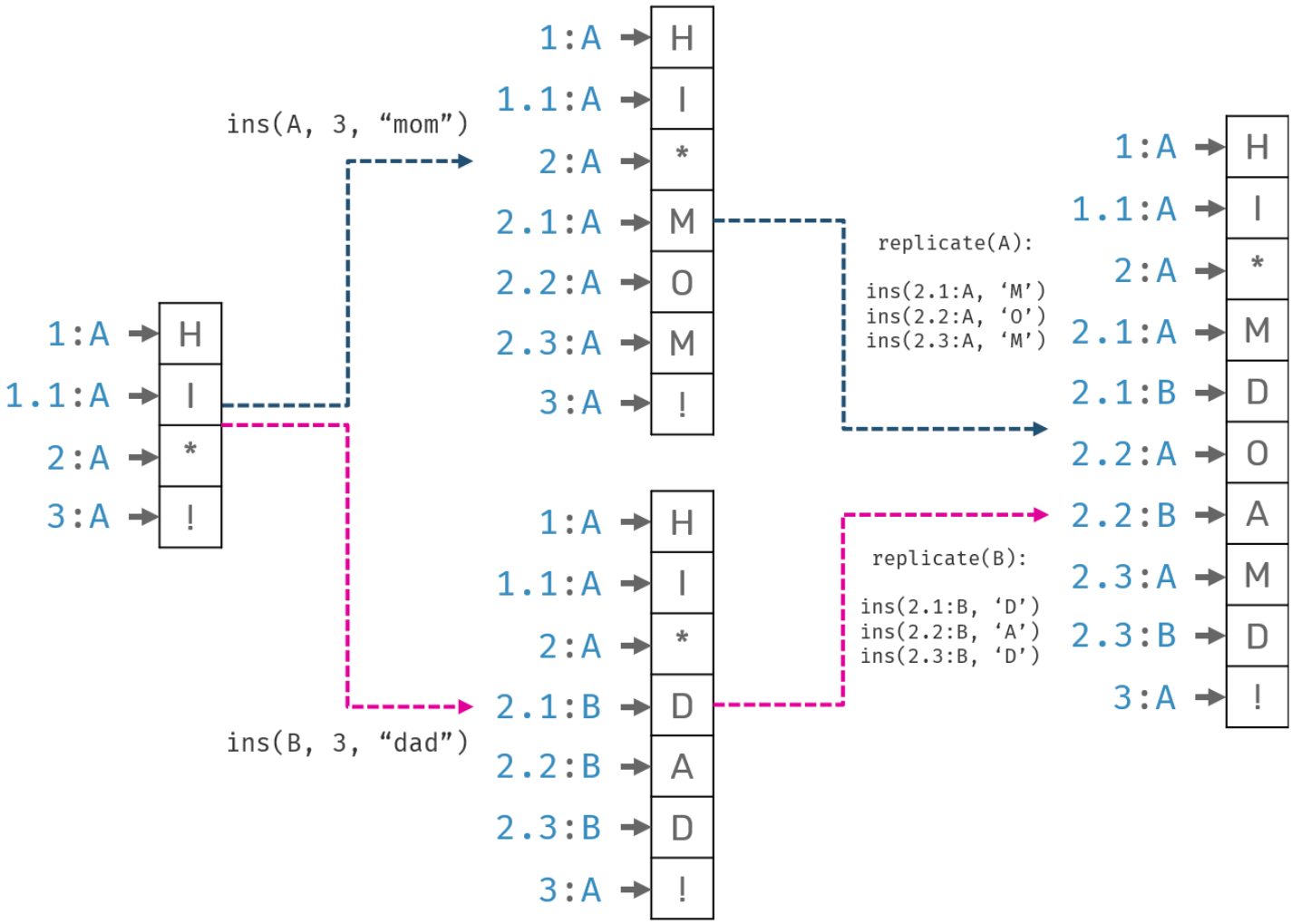
With these in hand we are not only able to identify each inserted element uniquely (no matter who and when assigned insert request), but also to compare them in a deterministic manner: which may be necessary when i.e. two peers will insert two different items at the same index without talking with each other.

Interleaving

We already discussed interleaving issue in the past, but lets quickly remind it here together with the ways on how to avoid it.

In short interleaving may happen, when two peers decide to insert entire ranges of elements at the same index. After synchronizing with each other it may turn out that their inserts are mixed (interleaved with each other). While this may not be an issue in some cases, there's a one famous scenario where this behavior wrecks havoc: collaborative text editing.

Two people, Alice and Bob, want to collaborate over a text document. Initial document content was "hi !". While offline, Alice edited it to "hi mom!", while Bob to "hi dad!". Once they got online, their changes were synchronized. What's a desired state of the document after sync?



While we could accept resolved document in form of "hi momdad!" or "hi dadmom!" - since they are easy to fix - the fact that characters from individual users could be mixed together would result in subpar user experience.

Mitigating interleaving issues

Interleaving was especially prevalent issue of a LSeq algorithm. But why? The reason for this is as always: **we didn't provide enough metadata to preserve the intent** in the face of concurrent updates - in this case placing series of characters one right after another.

If we define each **LSeq** insert operation as `ins(id, char)`, we can try to represent "*abc*" as series of inserts `[ins(1:A, 'a'), ins(2:A, 'b'), ins(3:A, 'c')]`. These however are not equivalent: we just

defined 3 operations that look like they are independent on each other. If our intent was to represent them one after another, we didn't have a way to specify it.

RGA solved problem of interleaving by attaching a predecessor as a part of operation eg. `ins(predecessor, id, char)`. This way we can define "*abc*" as `[ins(<root>, 1:A, 'a'), ins(1:A, 2:A, 'b'), ins(2:A, 3:A, 'c')]`, and preserve information about the dependency chain between these characters. Does that mean that RGA solved the interleaving issue? There are corner cases where interleaving can still happen - like inserting at the start of a list or in the case of prepends (since we only defined predecessor/left neighbor of an element).

YATA is similar to RGA in regard to keeping expected predecessor id as part of the operation. Additionally we also attach a successor id. In YATA naming convention, we call them left/right origins. Therefore insert operation can be defined as `ins(left, right, id, char)`. Why do we need two origins instead of one like in case of RGA? YATA is delta-state CRDT with a fairly low number of restrictions regarding its replication protocol - at least when we compare it to a reliable causal broadcast requirements of their operation-based counterparts. In order to correctly place inserts send out of order, sometimes we may need two pointers instead of just one.

Implementation

Here we won't touch all of the crazy optimizations that Yjs made - we'll discuss them other time - instead we'll focus on being succinct. We'll cover a code, which full working snippet can be found here.

In the introduction we've already discussed the basics of metadata carried over by insert operations, now it's the time to express them in code.

```
type ID = (ReplicaId * uint64)

type Block<'t> =
  { Id: ID                  // unique block identifier
    OriginLeft: Option<ID>  // left neighbor at moment of original insertion
    OriginRight: Option<ID> // right neighbor at moment of original insertion
    Value: Option<'t> }     // value stored in a block, None if tombstoned
  member this.IsDeleted = Option.isNone this.Value

type Yata<'t> = Block<'t>[]
```

Our core unit of composition is `Block` type. We can simply represent a `Yata` collection in terms of array, where blocks are laid out in their read order. This makes materialization to our user-readable view very simple:

```
let value (array: Yata<'t>) : 't[] =
    array |> Array.choose (fun block -> block.Value)
```

Next we're going to define two operations: insertion and deletion at a given index. We'll start with insert. It's very easy. First we need to map user defined index into actual index inside of our `Yata` collection - since we're keeping deleted blocks around, these two may not be equivalent. Then we obtain IDs of its neighbors (if they exist). Finally we're creating a new block and inserting it at mapped index.

```
let insert replicaId index value (array: Yata<'t>) : Yata<'t> =
  // find actual index in an array, without skipping deleted blocks
  let i = findPosition index array
  // try to get IDs of left and right neighbors
  let left = array |> getBlock (i-1) |> Option.map (fun b -> b.Id)
  let right = array |> getBlock i |> Option.map (fun b -> b.Id)
  // get the last known sequence number for a given replica
  // and increment it
  let seqNr = 1UL + lastSeqNr replicaId array
  let block =
    { Id = (replicaId, seqNr)
      OriginLeft = left
      OriginRight = right
      Value = Some value }
 Array.insert i block array
```

Deletions are even more straightforward - all we need to do is to map user index with regard to tombstones to find a correct block and remove its value:

```
let delete (index: int) (blocks: Yata<'t>) : Yata<'t> =
  let i = findPosition index blocks
  let tombstoned = { blocks.[i] with Value = None }
  Array.replace i tombstoned blocks
```

Merging

Merges are a little bit more complicated. Why? Left and right origins - the very same thing that protects us from interleaving issues - build a dependency chain that will complicate merging process. In order to safely resolve any potential conflicts, that may have appeared in result of concurrent block insertion, we use left and right origins as a reference points. That mean, they must have been there first.

When we're going to do merges we also need to remember about existing blocks that have been tombstoned - we only recognize them by absence of their value. Thankfully, once a block is deleted, it's not reintroduced again: inserting the same value at the same index is still a separate insert operation.

```
let merge (a: Yata<'t>) (b: Yata<'t>) : Yata<'t> =
  // IDs of the blocks that have been tombstoned
  let tombstones = Array.choose (fun b -> if b.IsDeleted then Some b.Id else None) b
```

```
    let mutable a =
      a // tombstone existing elements
      |> Array.map (fun block ->
        if not block.IsDeleted && Array.contains block.Id tombstones
        then { block with Value = None } // mark block as deleted
        else block
      )
    // IDs of blocks already existing in `a`
    let mutable seen = a |> Array.map (fun b -> b.Id) |> Set.ofArray
    let blocks =
      b
      // deduplicate blocks already existing in current array `a`
      |> Array.filter (fun block -> not (Set.contains block.Id seen))
    let mutable remaining = blocks.Length
    let inline isPresent seen id =
      id
      |> Option.map (fun id -> Set.contains id seen)
      |> Option.defaultValue true

    while remaining > 0 do
      for block in blocks do
        // make sure that block was not already inserted
        // but its dependencies are already present in `a`
        let canInsert =
          not (Set.contains block.Id seen) &&
          (isPresent seen block.OriginLeft) &&
          (isPresent seen block.OriginRight)
        if canInsert then
          a <- integrate a block
          seen <- Set.add block.Id seen
          remaining <- remaining - 1
    a
```

What's easy to miss here is a small `integrate` function call - it's in fact the heart of our conflict resolution algorithm:

```
  let private integrate (array: Yata<'t>) (block: Block<'t>) : Yata<'t> =
    let (id, seqNr) = block.Id
    let last = lastSeqNr id array
    if last <> seqNr - 1UL
    // since we operate of left/right origins we cannot allow for the gaps between blocks to hap
    then failwithf "missing operation: tried to insert after (%s,%i): %O" id last block
    else
      let left =
        block.OriginLeft
        |> Option.bind (indexOf array)
        |> Option.defaultValue -1
      let right =
        block.OriginRight
        |> Option.bind (indexOf array)
        |> Option.defaultValue (Array.length array)
```

```
    let i = findInsertIndex array block false left right (left+1) (left+1)
    Array.insert i block array
```

Based on the left and right origins of the block, we can establish a boundaries between which it's safe to insert a new block eg. if originally block was inserted at index 1, it's left and right origins prior to insertion were are positions 0 (left) and 1 (right). However as blocks can be inserted in the same place at different times on different replicas, the window between left and right origins may have grown. To resolve total order of blocks inserted within that window, we're use following function:

```
let rec private findInsertIndex (array: Yata<'t>) block scanning left right dst i =
  let dst = if scanning then dst else i
  if i = right then dst
  else
    let o = array.[i]
    let oleft = o.OriginLeft |> Option.bind (indexOf array) |> Option.defaultValue -1
    let oright = o.OriginRight |> Option.bind (indexOf array) |> Option.defaultValue array.Len
    let id1 = fst block.Id
    let id2 = fst o.Id
    if oleft < left || (oleft = left && oright = right && id1 <= id2)
    then dst
    else
      let scanning = if oleft = left then id1 <= id2 else scanning
      findInsertIndex array block scanning left right dst (i+1)
```

What this piece does is to check if there are other blocks that were potentially inserted concurrently at the same position. If so, we're going to compare the unique identifiers of each peer so that inserts made by peers with higher IDs will always be placed after peers with lower IDs.

This is in essence very similar to RGA conflict resolution algorithm (*in case of concurrent inserts at the same position shift element to the right if its contender had lower peer id*) with one difference:

- RGA block ID uses a single globally incremented counter. A right shift stop condition is: *until sequence number of a right-side neighbor's ID is lower than current block's*. This has sense, as globaly incremented sequence number means that only concurrently inserted blocks can have same or higher sequence numer.
- YATA block ID uses a counter per replica, however here a right shift stop condition is defined explicitly - it's a position of our right origin.

As you might have noticed this implementation doesn't allow for out-of-order deliveries - which puts some restrictions on our replication protocol. Some implementations (like Yjs) solve this by stashing blocks that have arrived before their predecessors and reintegrating them once block's dependencies have been received.

Deltas

Right now we know how to merge entire snapshot of two Yata replicas. However as we mentioned in the past, this approach is not flexible and can introduce heavy overhead - changing one element would require sending entire state to all peers. This is a motivation behind delta-state replication.

In this chapter we also discussed how to exchange deltas efficiently - so that they only carry information not observed by specific peers. But how can we tell, which updates are yet to be replicated? During the CRDTs series on this blog we already presented so called vector clocks and dotted vector versions (if we want to support out of order deliveries, which for simplicity reasons is not the case here) they let us efficiently describe an observed (logical) timeline of a given CRDT structure.

Fortunately there's a 1-1 relation between YATA block IDs and vector clocks. Here all we really need is to aggregate the highest IDs of all blocks in current YATA array:

```
let version (a: Yata<'t>) : VTime =
  a
  |> Array.fold (fun vtime block ->
    let (replicaId, seqNr) = block.Id
    match Map.tryFind replicaId vtime with
    | None -> Map.add replicaId seqNr vtime
    | Some seqNr' -> Map.add replicaId (max seqNr' seqNr) vtime
  ) Version.zero
```

Given such vector clock, we can also easily generate the deltas themselves. All we need to do is to take blocks which sequence numbers are higher that sequence numbers of corresponding vector clock entries. After this we're almost done: we also need to remember that there might be blocks known to remote peer, that have been deleted in the meantime. For this reason we take IDs of tombstoned blocks and pass them over as well.

```
type Delta<'t> = (Yata<'t> * ID[])
let delta (version: VTime) (a: Yata<'t>) : Delta<'t> =
  let deltaArray =
    a
    |> Array.filter (fun block ->
        let (replicaId, seqNr) = block.Id
        match Map.tryFind replicaId version with
        | None -> true
        | Some n -> seqNr > n)
  let tombstones =
    a
    |> Array.choose (fun block ->
        if block.IsDeleted then Some block.Id else None)
  (deltaArray, tombstones)
```

While it may look like pushing all tombstoned elements is not really matching what we'd like to expect from delta behavior, in practice they are minor issue:

1. Since we only pass identifiers, their size is pretty small.
2. Practical implementations may encode tombstoned ranges in very compact notation eg. `A:(1..100)` to encode **all** deleted blocks from `A:1` up to `A:100`.

After describing these details, implementation of delta merges becomes trivial:

```
let mergeDelta (delta: Delta<'t>) (a: Yata<'t>) : Yata<'t> =
  let (delta, tombstones) = delta
  merge a delta // merge newly inserted blocks
  |> Array.map (fun block ->
      if not block.IsDeleted && Array.contains block.Id tombstones
      then { block with Value = None } // tombstone block
      else block)
```

Summary

Here, we presented all the basics of YATA algorithm, used for building conflict-free, delta-state indexed sequences. In the future we're going into deep dive through Yjs/Yrs - a libraries which elevate these principles and improve over them to achieve high efficiency.

References

- Yjs - a production-grade implementation of YATA algorithm. It supports a block-wise CRDTs (similar in principle to what we already presented), different data types (like maps or XML elements).
- A reference implementation of several text-editing structures, written in TypeScript.
- Another implementation made by Seph Gentle, which explores variety of different optimizations that can be done to speedup YATA algorithm.

Deep dive into Yrs architecture

In this chapter we're going to cover the the internals and architecture behind Yrs (read: *wires*) - a Rust port of popular Yjs CRDT library used for building collaborative peer-to-peer applications, which now also have bindings for WebAssembly and native C interop.

But first: why would you even care? The reason is simple: Yjs is one of the most mature and so far the fastest production-grade Conflict-free Replicated Data Type project in its class. Here we're going to discuss the shared foundation and optimization techniques of both of these projects.

Document

Document is the most top-level organization unit in Yjs. It basically represents an independent key-value store, where keys are names defined by programmers, and values are so called **root types**: a Conflict-free Replicated Data Types (CRDTs) that serve particular purposes. We'll cover them in greater detail later on.

In Yjs every operation happens in a scope of its document: all information is stored in documents, all causality (represented by the vector clocks) is tracked on per document basis. If you want to compute or apply a delta update to synchronize two peers, this also must happen on an entire document.

Documents are using delta-state variant of CRDTs and offer a very simple 2-step protocol, which is similar to an optimization technique we discussed in the past:

1. Document *A* sends its vector version (in Yrs it's called `StateVector`) to Document *B*.
2. Based on *A*'s state vector, *B* can calculate missing delta which contains only the necessary data and then send it back to *A*.

This way we make deltas efficient and very lightweight. Even more, Yrs encodes deltas using customized binary format, optimized specifically to make payloads small and efficient.

Under the hood, document represents all user data together with an information used for tracking causality within a dedicated collection called the **block store**. But before we go there, lets first explain transactions.

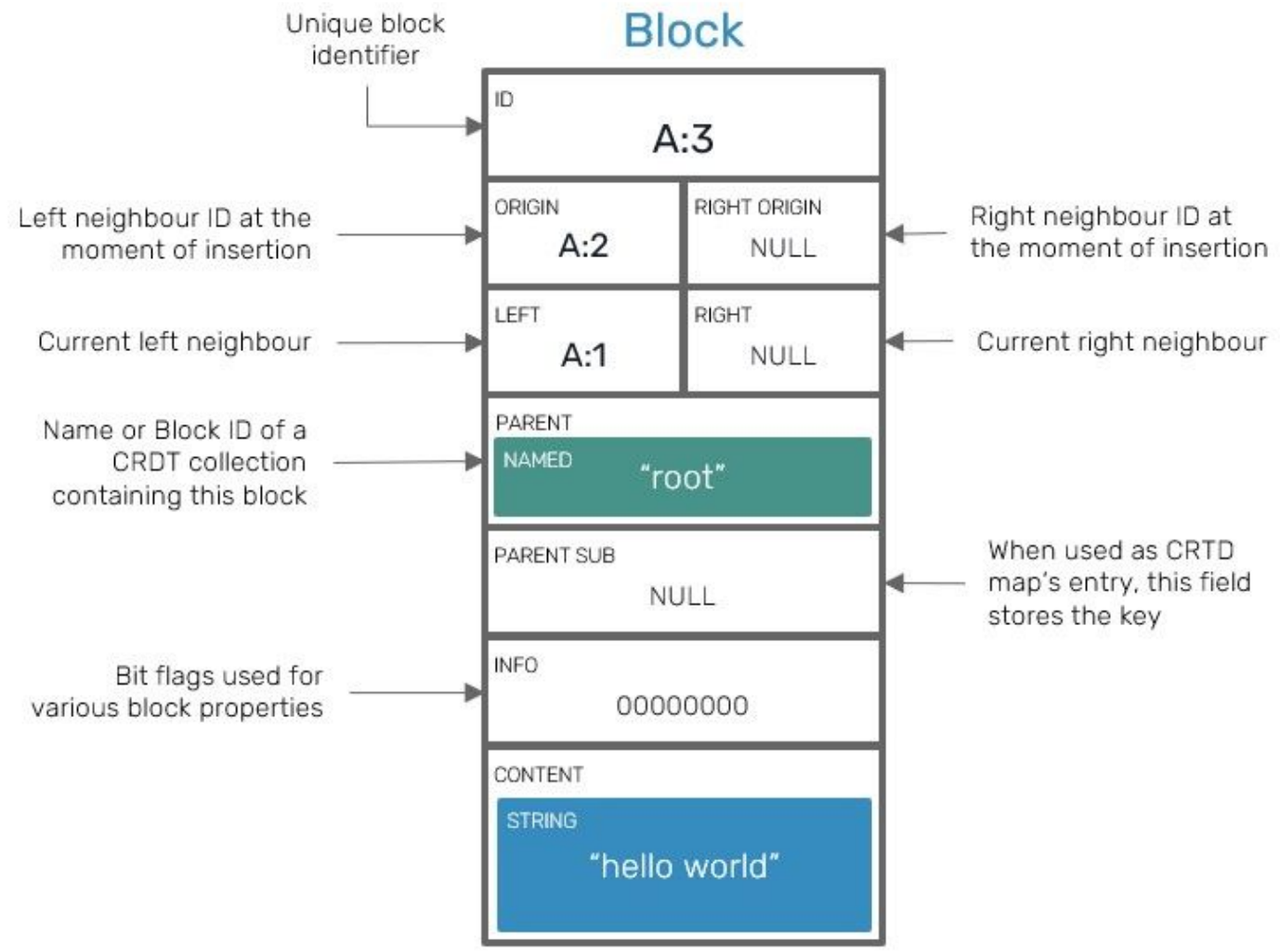
Transactions

A concept correlated to documents is one of a transaction - that name was inherited from Yjs and may be deceptive, as they don't have anything in common with ACID transactions known from standard databases. They are more like batches of updates reinforced with some optimization logic - like checking if updates that happen within a scope of transaction can be squashed into previously existing block store structure or triggering update events, programmers can subscribe to. These kick in upon transaction commit, which also happens automatically when the transaction is disposed.

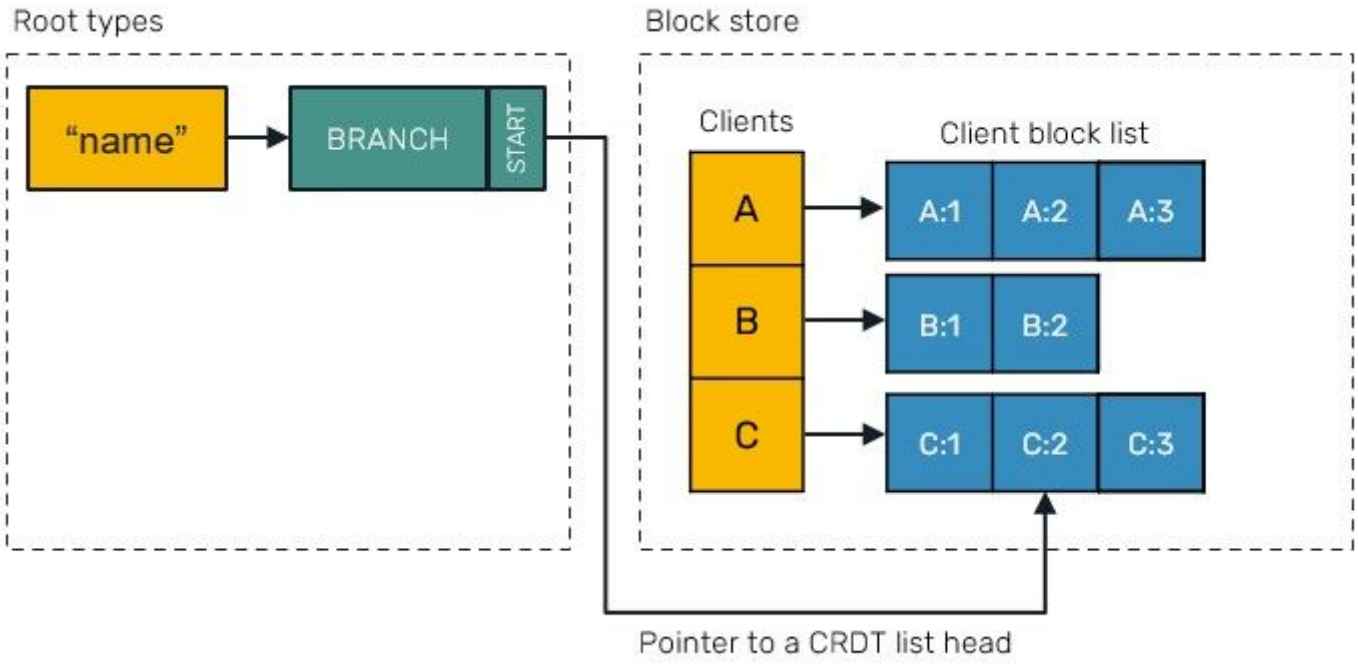
Block store

Store is the hearth of a document. It contains all of the necessary information about updates that have happened in the scope of that document, as well as the metadata about the shape of the data types created within. At the moment Yjs/Yrs are both schemaless and to some extent also weakly typed (we'll talk about it later).

All updates in Yrs are represented as blocks (a.k.a. structs in Yjs). Whenever you're inserting or removing a piece of data into your Y-data structure - be it a `YText`, `YArray`, `YMap` or XML types - you're producing a block. This block contains all metadata necessary to correctly resolve potential concurrency conflicts with blocks inserted concurrently by other users:



These blocks are organized in a block store on per client (peer) basis - all blocks produced by the same client are laid out next to each other in memory:



This layout is optimal to perform several operations like using state vectors and deltas, block squashing, efficient encoding format etc.

Yjs/Yrs use YATA algorithm as an universal conflict resolution strategy for all data types. We already discussed these with a sample implementation before, but only in context necessary to understand construction of indexed sequences. We'll cover how to extend this approach over to key-value maps when we'll discuss YMap design.

Each block is reinforced by a set of fields, most of which are optional:

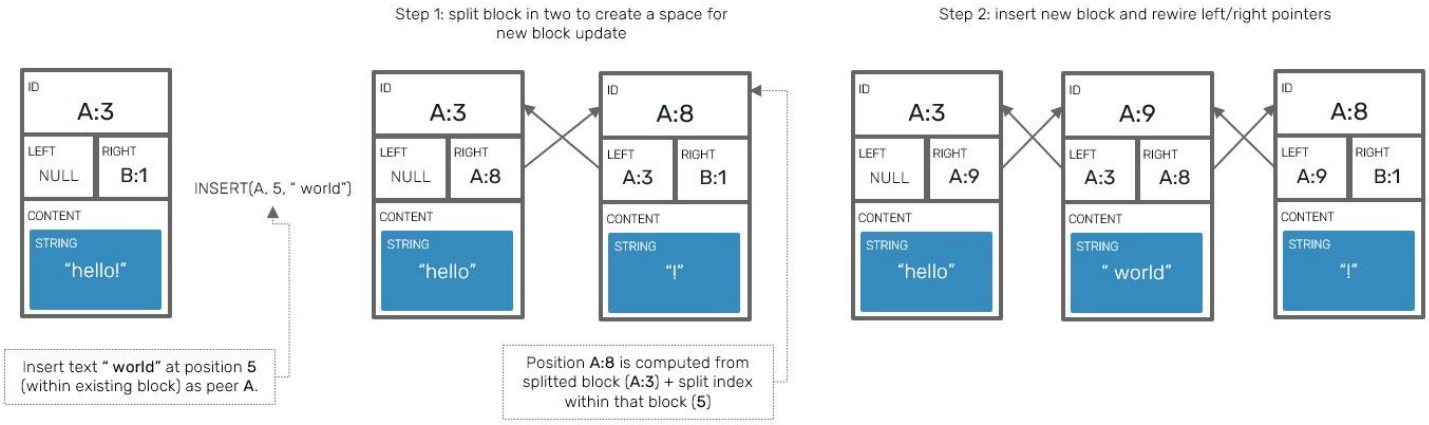
- Unique block identifier, which is logically similar to Lamport Timestamp.
- Left and right origins which point to its neighbours IDs at the moment of insertion and are used by YATA algorithm.
- Pointers to left and right blocks - in Yjs/Yrs block sequences are organized as double-linked lists. Keep in mind that, these fields don't have to be equal to origins: origins represent neighbors *at the moment of original block insertion*, while these pointers represent *current placement* of a block in relation to others.
- Parent, which points to a shared type containing this block. In case you haven't figured out: Yjs/Yrs collection types can be nested in each other, creating a recursively nested document trees.
- In cases of map-like types (`YMap` but also attributes of XML types), it can also contain an associated key, it's responsible for, effectively working as a key-value entry.
- Bit-flag field informing about various properties specific to that block data, such as information if such block has been tombstoned or what is its intended collection type.
- Content, which contains a user-defined data. It can be as simple as a JSON-like value - number, string, boolean or embedded map that doesn't expose CRDT properties, but also other nested CRDT collections.

You may think, this is a lot of metadata to hold for every single character inserted by the user. That would be right, however Yjs implements an optimization technique that allows us to squash blocks inserted or deleted one after another, effectively reducing their count while keeping all guarantees in check.

Block splitting

As we mentioned in order to optimize the metadata size of a particular insert operations, we can represent multiple consecutively inserted elements into one block. This however begs for question: *what happens when we want to insert a new value in between elements stored under the same block?* This is where a block splitting comes to work.

We already discussed in detail how to implement a block-wise variant or RGA CRDT on this chapter, and it's YATA equivalent is not that much different. The **tl;dr** explanation is: whenever we want to make a new insert in between elements grouped under the same block, we split that block in two parts and rewire their left and right neighbour pointers to a newly inserted block.

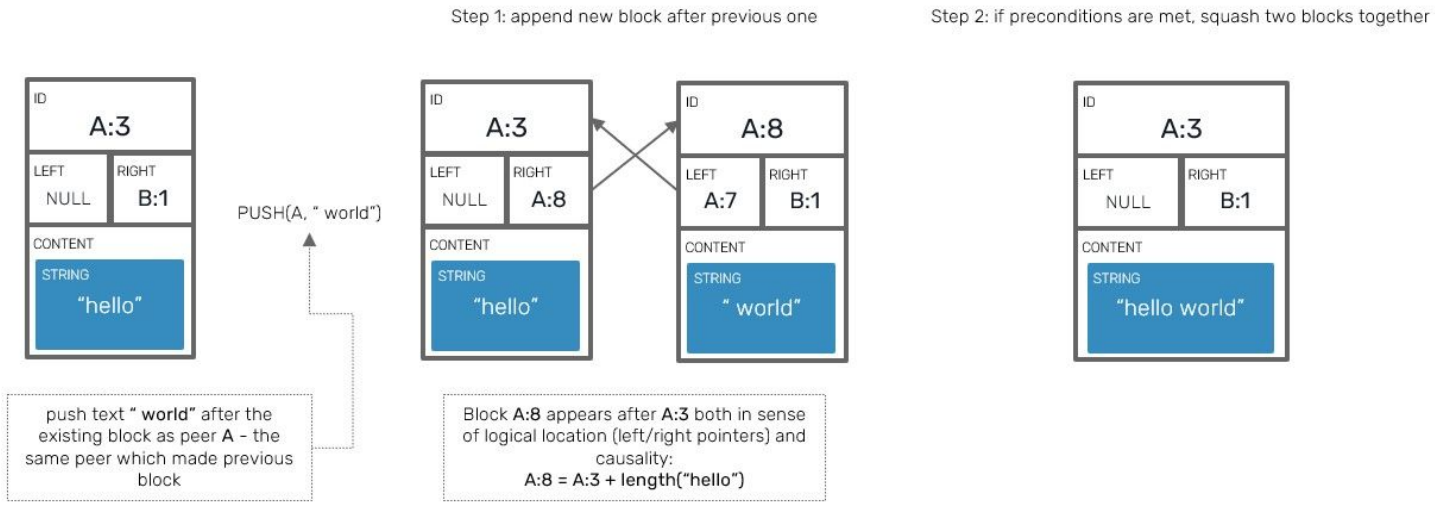


All of this is possible because consecutive block ID (client_id-clock pairs) are not assigned using clock numbers increased by one per block, but by one **per a block element** eg. *if we assigned an ID* `A:1` *to a 3-element block* `[a,b,c]` *and then insert a next element* `d` *on the same client, that inserted block's ID will be* `A:4` *(it's increased by the number of elements stored in the previous block).* By the same rule whenever we need to split block `A:1=[a,b,c]`, because we want to insert another element between `b` and `c`, we can create two blocks `A:1=[a,b]` and `A:3=[c]`. These blocks representations are logically equivalent within a document structure.

Block squashing

Given the example above we can easily explain process of **squash**ing (name is borrowed from a similar concept used by Git): it's a reverse of splitting, executed upon transaction commit. It's very useful in scenarios when the same peer inserts multiple elements one after another as a separate operations. The most obvious case of this is simply writing sentences character by character in your text editor.

Thanks to block squashing, we can represent separate consecutive blocks of data (eg. `A:1=[a]` followed by `A:2=[b]`) using a single block instead (`A:1=[a,b]`).



The major advantage is reduction of memory footprint (blocks contain extra metadata that can easily outweigh a small content footprint) and better memory locality (blocks are organized as double

linked list nodes, while contents of individual block form continuous array of elements placed next to each other in memory).

Keep in mind that squashing has some limits to it: squashed blocks must be next to each other both in terms of A) being each others neighbours B) emitted by the same client and C) their IDs must follow one another with no "holes between" (eg. we can potentially squash `A:1=[a,b]` with `A:3=[c]`, but not with `A:4=[c]` because we would miss some block with ID `A:3` in between). Both blocks must also contain data of the same, *splittable* type.

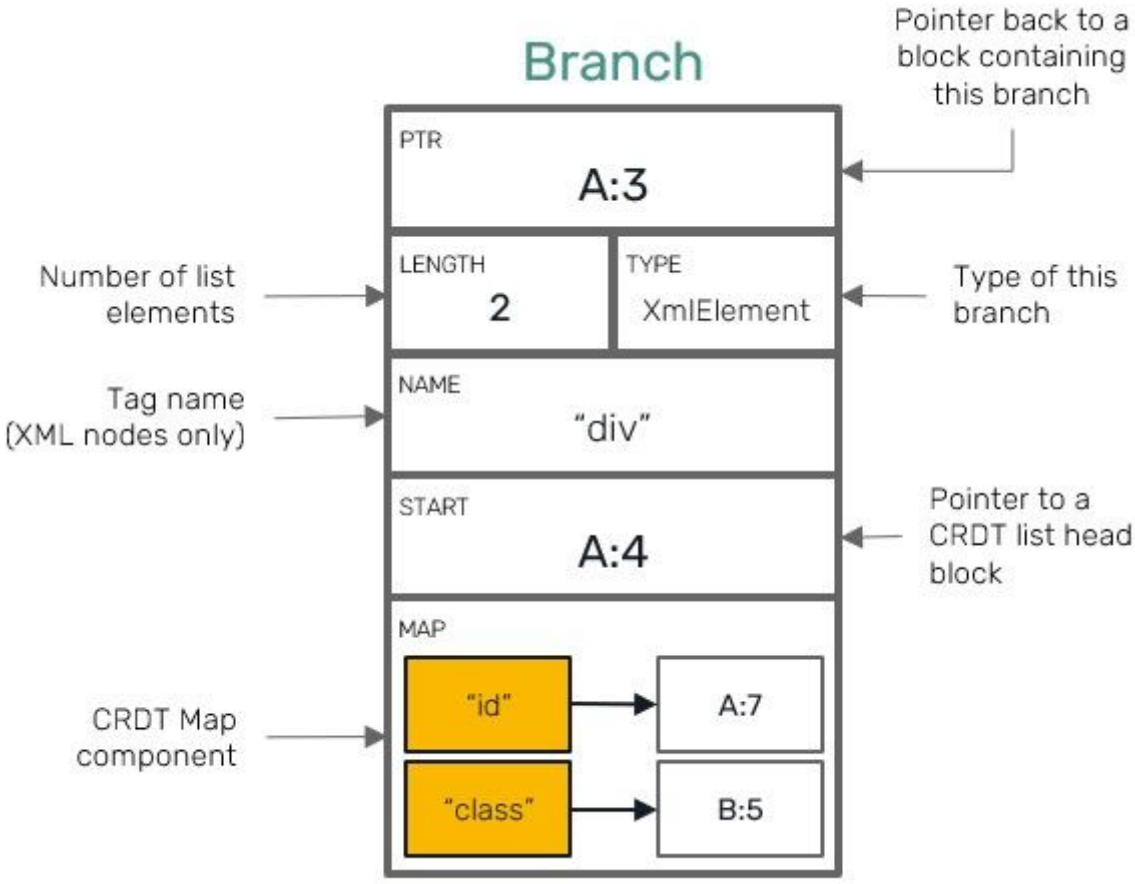
Shared types

Another group of types supported by Yrs are so called shared types, which represent collections capable of having CRDT semantics. Yrs uses delta-state approach to CRDTs with an extremely small and robust replication protocol. Currently there are several types of these:

- `Text` and `XmlText` specializing in supporting collaborative text editing.
- `Array` used as index-ordered sequence of elements.
- `Map` used as key-value hash map with last-write-wins semantics (for a quite specific definition of "last" which we'll cover later).
- `XmlElement` which works as an XML node capable of having key-value string attributes as well as other nodes in user-specified order: either other `XmlElement`s or `XmlText`.

These types can be nested in each other according to their limitations. Unlike primitive values they can also be assigned to the document itself (using their unique names for retrieval) and obtained right from it. Shared objects created straight at the document level are called a root level types and have a very interesting characteristic - their actual type is only a suggestion on how to present their content. What does it mean and how does it work?

Underneath all CRDT collections are represented by the same kind of block content known as a `Branch`. Branch node is capable of storing two specific types of collections at once: both key-value entries and index-ordered double-linked list containing batches of items.



These two capabilities define a common operations, which then can be used to represent more specific traits of each type, eg.:

- `Text` / `XmlText` use indexed sequence component of branch to build a list of text chunks. Calling `to_string` operation on these types simply glues these separate chunks of text together into a single coherent string object.
- `Array` uses the same chunks to store any kind of arbitrary data items (you could think about text chunks as batches of individual characters, each one being a separate item).
- `Map` uses only key-value entry component to store values of any type, be it a primitive or another CRDT collection.
- `XmlElement` uses key-value entries to store its attributes, and sequence component of a `Branch` node to contain nested XML nodes. Additionally `XmlText` is also capable of storing attributes - it's the main difference from standard `Text` field - which can be then used by specific text editors to store information used for eg. text formatting.

This structure also enables to reinterpret root-level types eg. you can read a `Text` as an `Array` (to change its materialized type to a list of characters) or `XmlElement` as a `Map` (and be able to read only its attributes). Just like with any other kind of weakly typed systems you should use these capabilities with caution.

YText and YArray

We already mentioned, how Yrs `Text` and `Array` types make use of the same indexed sequence component and their in-memory representation is basically a double linked list of continuous

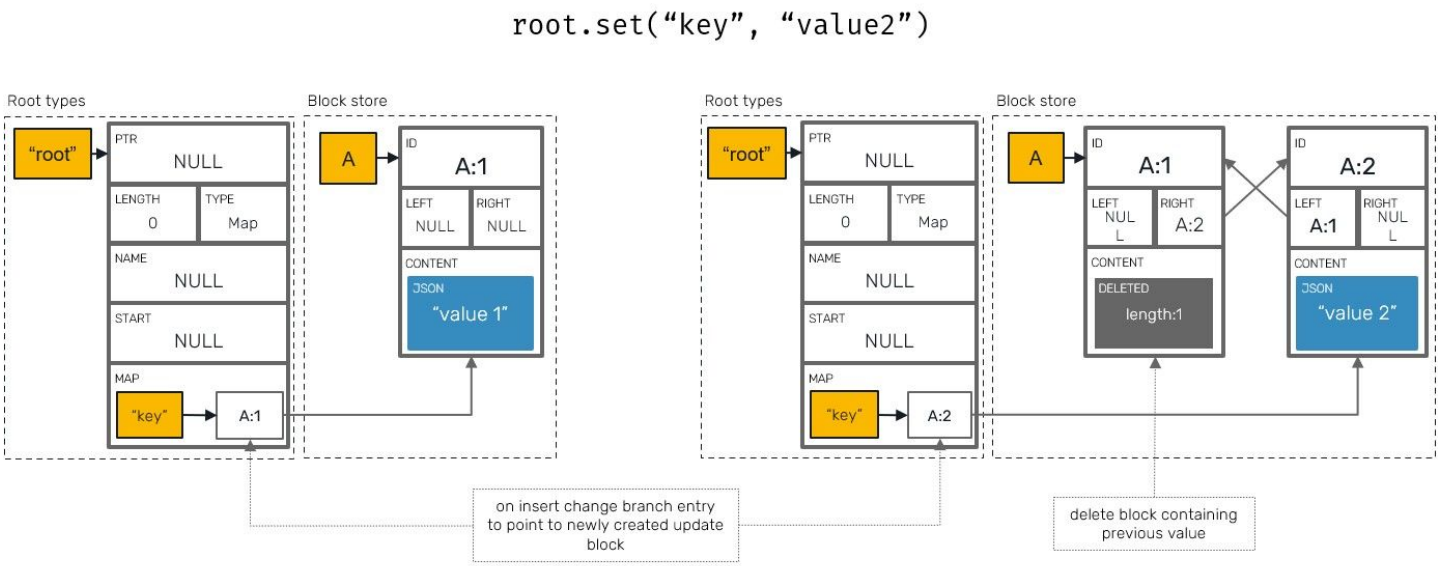
elements (such as individual string characters or ranges of JSON values).

Thanks to the use of YATA conflict resolution algorithm, these types are free from interleaving issues boggling other CRDT algorithms like LSeq. And since we're using split-block approach, it lets our insert operations computational complexity to become independent of the number of elements inserted (in most cases). In practice this means, that operations like text copy-paste always have complexity of `O(1)` no matter of the number of characters to insert. Sounds obvious, but atm. it's still not a standard in many other CRDT implementations you could find out there.

YMap

So far we only mentioned, that all branch nodes are by their nature capable of storing key-value data using Last Write Wins™ semantic. How does it work? Initial premise is simple - all branch nodes literally have special "map" field, which contains hashmap of entries, each of them being a string key and pointer to the last block (all Yrs values are stored in blocks) containing a corresponding value.

Now, whenever we override that value by another insert under the same key, we attach a new data as the right-most neighbour of the previously written block, point to it from now on, and tombstone all blocks on the left side of it, including a previous entry's value.

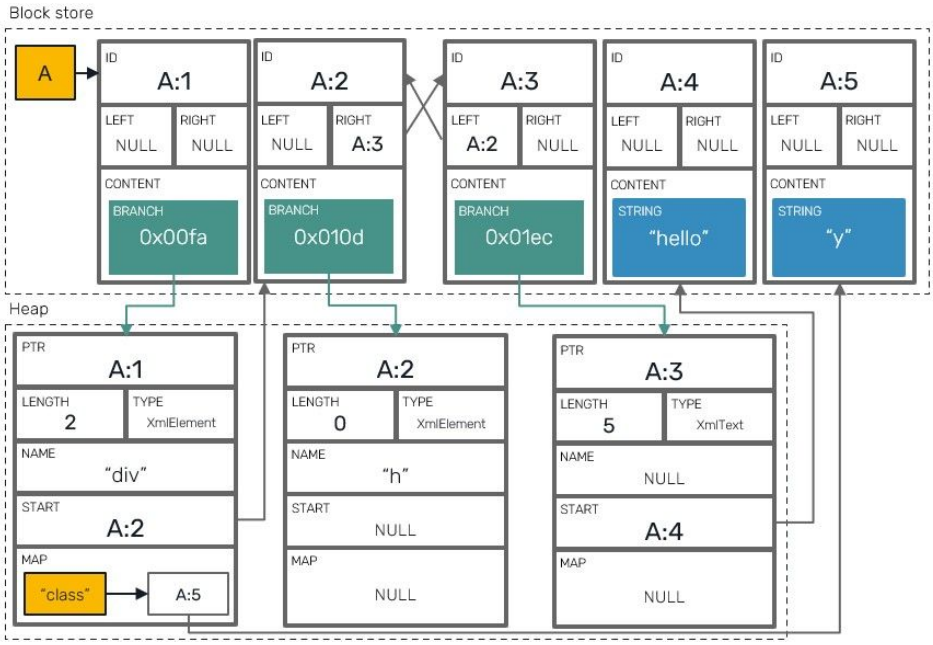


This is what we mean by "Last Write Wins": instead of using client system clock on the peer machines we use the same principles that YATTA algorithm offers to order text insertions. In this context *last* means "the block with the highest client ID and clock value".

Xml types

One of the things, that Yrs is also capable of, is a conflict-free support for XML representation. What we mean by that is a dynamic nested XML tree of nodes, which can contain attributes, text or other nodes. The way it works is actually pretty simple: a branch node uses its list head to point to a first XML child node block and map field to point to blocks used as attributes.

```
<div class="y">
  <h></h>
  hello
</div>
```



Serialization

Both Yjs and Yrs use the same very efficient data exchange format, which means that we can achieve backward compatibility with systems already using Yjs. It exists in two versions:

- **v1** encoding relies heavily on variable integer encoding and block store layout to omit fields which can be inferred using blocks mutual relationships.
- **v2** encoding was influenced by research made by Martin Kleppmann when he worked on automerge library, and adapted to Yjs. It extends the formatting to make use of run-length encoding, which can bring further savings, especially for deltas having many block updates.

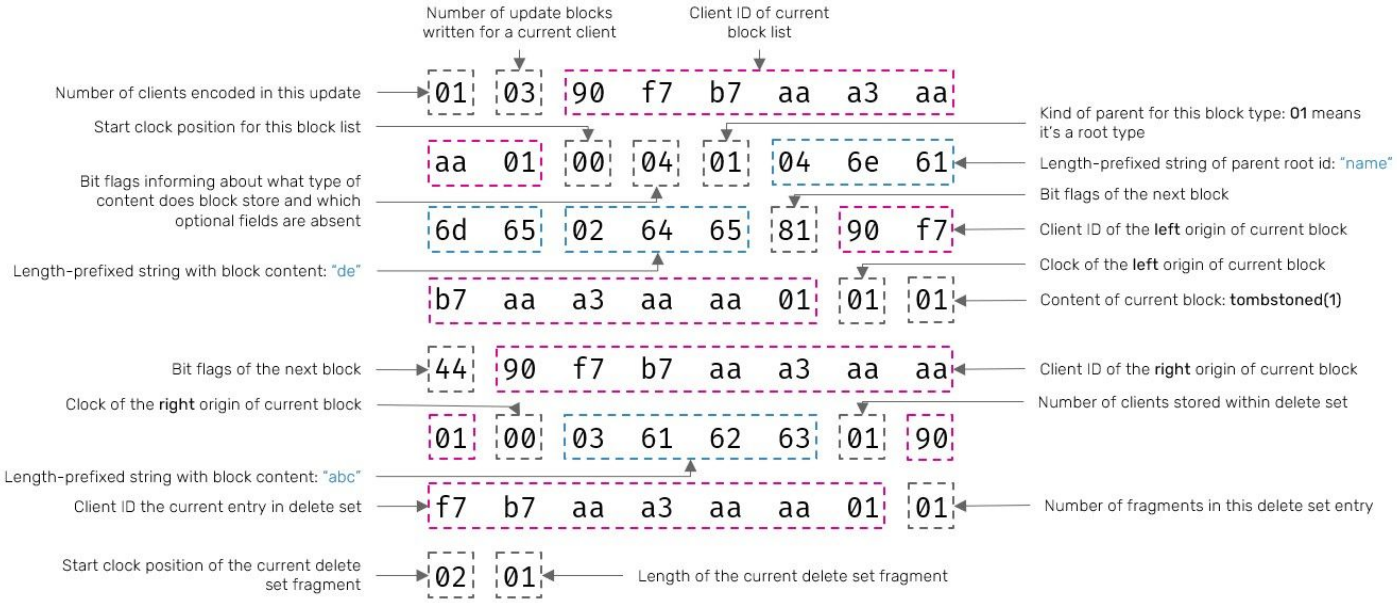
Note: *before moving on let's make quick recap of what varint encoding is all about - it allows us to save space when encoding integer values that usually take many bytes (like 4B and 8B integers) by getting rid of empty bytes. It's especially useful when stored integer values are small ie. string length or Lamport clock values.*

Here, we'll cover only v1 encoding. To do so, let's use some sample:

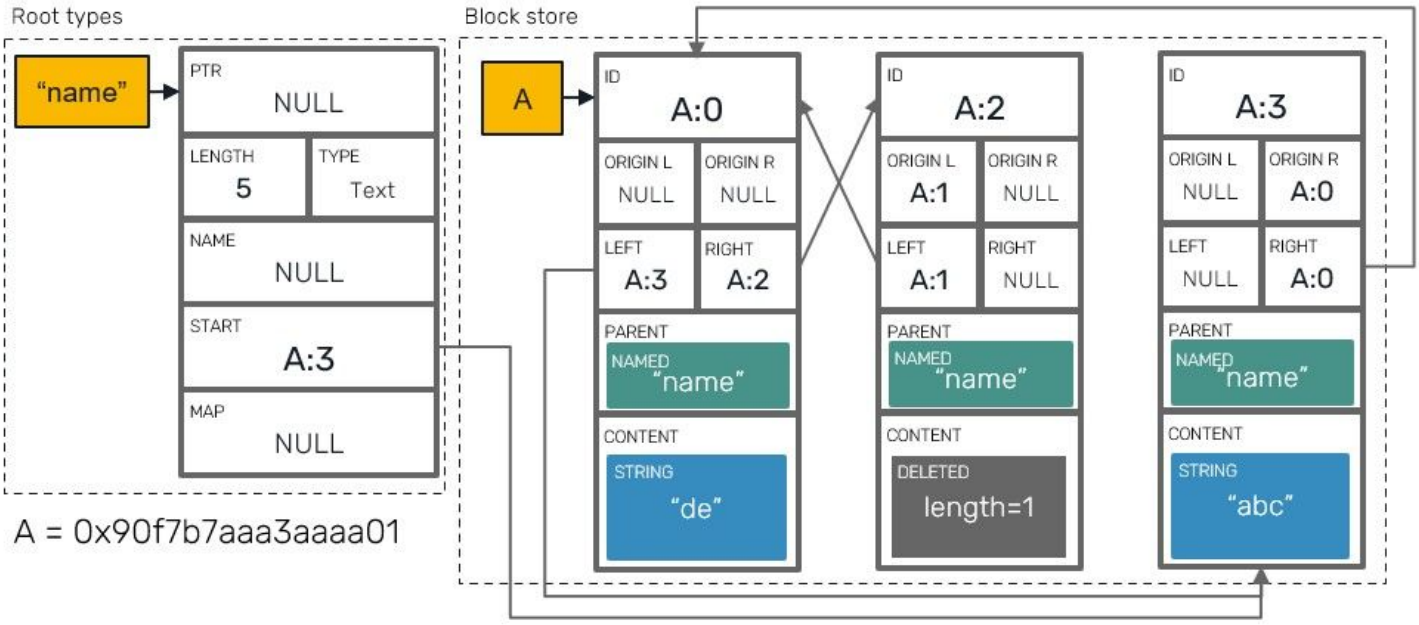
```
use yrs::Doc;

let mut doc = Doc::new();
let mut txn = doc.transact();
let ytext = txn.get_text("name");

ytext.push(&mut txn, "def");
ytext.insert(&mut txn, 0, "abc");
ytext.remove_range(&mut txn, 5, 1);

txn.commit();
```

```
let delta = txn.encode_update_v1();
```

This could produce following payload:



which further we could translate into something that resembles document store represented below:



We can make several observations here:

1. We try to deduplicate fields as much as possible:
   i. Block IDs are inferred based on start clock and length of previous block. Block ID's client field is also specified only once for all blocks sharing it.
   ii. Parent field is not repeated for every block, instead during decoding it will be inferred from linked neighbor blocks.
2. Some things like client IDs cannot be deduplicated by v1 encoding. In fact client IDs make up a big chunk of our delta payload seen above. It's because when using `Doc::new()` we're using a

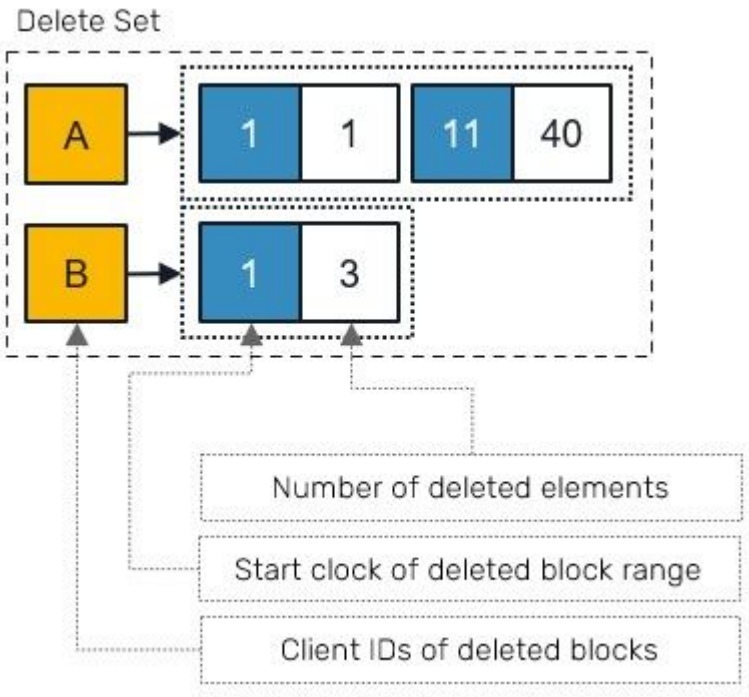
randomly generated numbers as client ids. If you're into byte-shaving business, you can assign small client ids manually by calling `Doc::with_client_id(id)` instead and let varint encoding compress bytes - this way we could even go down from **58B** → **30B** payload. However in such case you're responsible for keeping these IDs globally unique.

3. We use bit flag bytes to specify type of content stored in each block. We also utilize remaining bit positions to inform which fields are present (eg. left/right origins used above). This way we don't need to loose extra bytes to mark fields that were null.
4. As you may have noticed the delta payload doesn't even explicitly mention root types: their presence and identifiers are instead inferred directly from the block data.
5. In example above we keep information about tombstoned elements using both delete set and block with deleted content. It's because delete set propagation is mandatory, while blocks are encoded only if they were not observed by remote peers. It was the case here, but in reality it's not so frequent.

Using customized format we can make advantage of our knowledge about the domain of problem rather than relying on some generic serializer. This allows us to optimize payload size to a point, that wouldn't be possible otherwise.

Delete Set

We already mentioned Delete Sets, when we talked about YATA algorithm itself, but haven't covered it here. In short it's a collection of deleted ranges of block elements grouped by each block client ID:



Unlike in the case of Rust `Range<u32>` we use pair `(start, length)` rather than `(start,end)` since this gives us a chance for more compact representation when using varint encoding. Delete sets are mostly used as a serialization-time concepts. They are rarely present in the block store during normal updates and their state can be derived by traversing the document store itself.

With this approach we can inform about hundreds of removed elements at a cost of just few bytes.

Summary

With all the details described here I hope, you'll find this topic interesting. If you want to dive even more, feel free to take a look at Yrs repository. Here we only shared a top-level overview of the architecture, as it's the least likely to change but is still useful when trying to navigate over the code base.

We're still in the very early stages of the project and we're only just started our long way on optimizing the internals to take full advantage of the capabilities of Rust programming language - for these reasons keep a watchful eye on the version numbers in the benchmarks you might encounter ;)

Conflict-free reordering

In this chapter we'll define the basics for a move operation used by Conflict-free Replicated Data Types (CRDT), which allows us to reorder the items in the collection. We'll define the general approach and go through example implementation build on top of YATA sequences. What's nice about our approach is that it's not limited to a single algorithm - we can easy extend it to pretty much any indexable sequence CRDT.

We won't describe the details of YATA algorithm itself, as we did so in the past. We'll go straight to implementing our code on top of it. Therefore a prerequisite for the rest of this post would be to understand that algorithm.

Use case and motivation

Imagine that you need to build music service, that enables users to create their own playlists, listen them in offline mode and is available on different devices, like smartphones or desktop. The app should be responsive, allowing users to reorder tracks on their playlist at any time, and should synchronize with user account, making changes visible on other devices they are logged in, once they get online.

The problem might arise when a person at the same time reorders their playlist on both smartphone and laptop, while these devices are disconnected. We need to ensure that - once synchronized - both devices present the same order of tracks.

While we already have discussed Conflict-free Replicated Data Types a lot on this blog, defined foundations for building indexed sequences (at least using 3 different algorithms: LSeq, RGA and YATA) and described how to insert and remove elements in them, we cannot really just remove and re-insert item at a different position for the sake of feature described above. Let's see what could happen if we try:

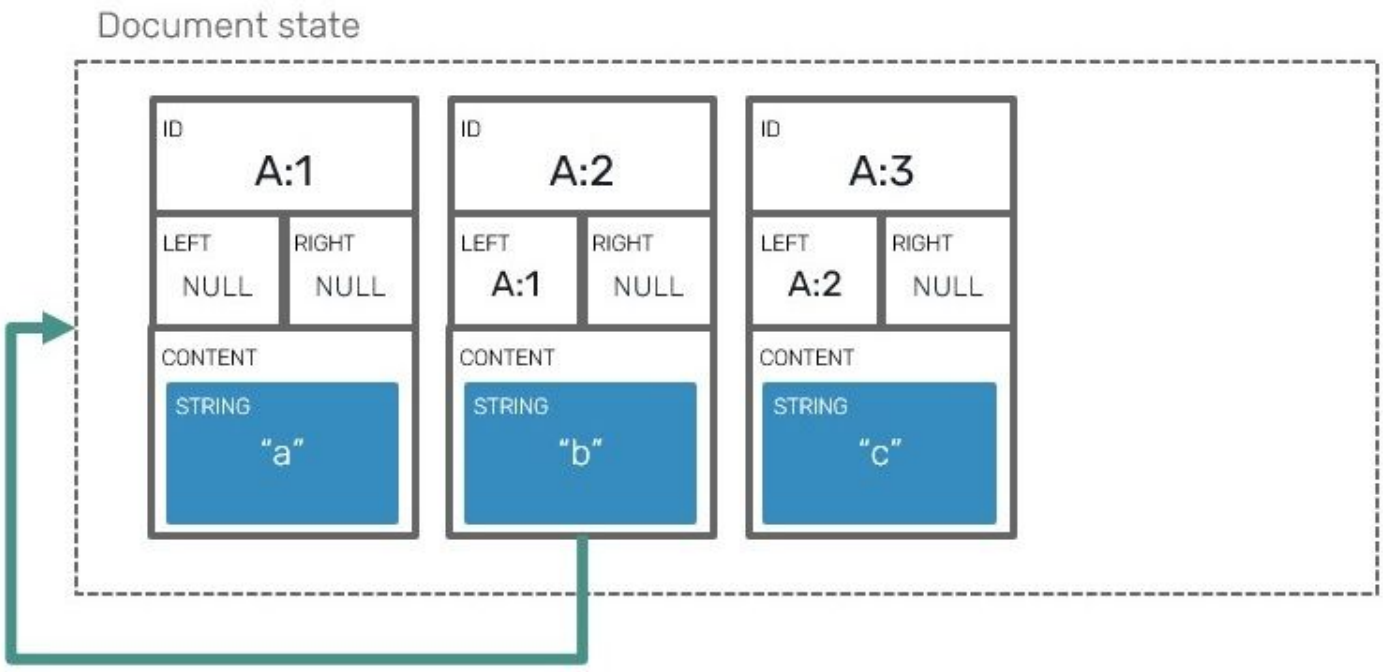
1. We start with an initial sequence of tracks: `[A,B,C]`

2. On our smartphone we **removed** `B` from its original position 1 and **inserted** it before `A` (index 0). Our expected order is: `[B,A,C]`.
3. On our laptop we **removed** `B` from position 1 and **inserted** it after `C` (index 3 prior removal). Our expected order is: `[A,C,B]`.

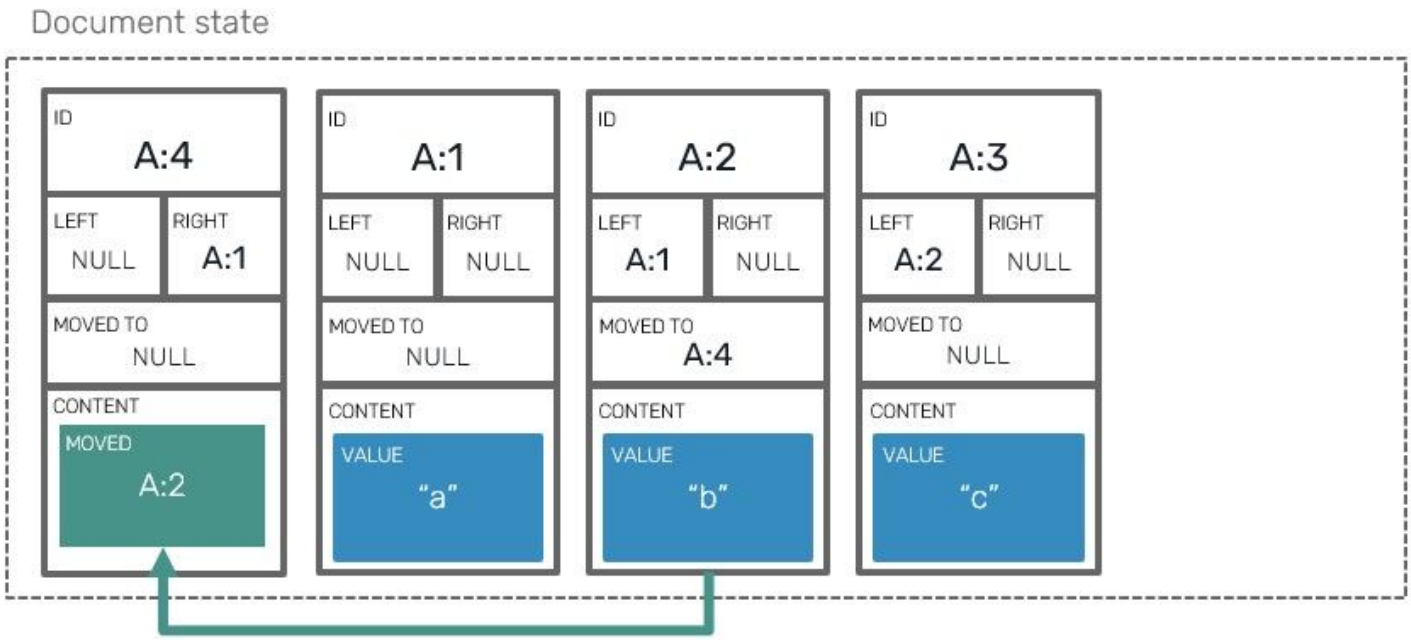
Now if we would synchronize these structures using any of the algorithms linked above, we could end up with `[B,A,C,B]`. This happens because each insertion counts as a new unique operation. We're not capable of establishing a correlation between previously removed element and newly inserted ones. **For purpose of resolving conflicts in concurrent operations, being able to correctly describe your intent is crucial**, and this is what we're missing here. We need a new method for that.

How to represent moved elements?

If we were to draw a diagram describing moving an element from one position to another, it could look like this:



In fact this is exactly how we could represent that operation:



In this chapter we're going to use snippets from the following gist. It's a modified source of the solution we described last time.

As it turns out, the enhancement we need focuses around two additions:

1. Add a field to a `Block` container informing, that it has been moved elsewhere.
2. Define new type of content informing that a containing `Block` works as a destination pointer for move operation of another block.

```
type Content<'t> =
  | Value of value:'t          // cell with a value inside
  | Tombstone                  // deleted value
  | Moved of ID * priority:int // move operation with an ID of moved block

type Block<'t> =
  { Id: ID
    OriginLeft: Option<ID>
    OriginRight: Option<ID>
    MovedTo: Option<ID> // where current block has been moved
    Value: Content<'t> }

module Block =
  let isDeleted b = match b.Value with Tombstone -> true | _ -> false
  let value b = match b.Value with Value v -> Some v | _ -> None
```

Before going further we should probably ask ourselves: do we even need that? Couldn't we just move the block physically into a new position? The answer is: NO. The core idea behind YATA (and many other CRDT sequences) conflict resolution relies on block defining its own position in relation to the neighbours at the moment when it was inserted. These origins must never change and algorithm expects to find block itself within a range bounds of its neighbours. Otherwise we'd run into either ignored or duplicated block updates during data merges, going back to square one. If we want to

establish a way to resolve conflict caused by an element moved concurrently we first need to provide information that it was moved in the first place.

So how do we actually move elements? It doesn't really differ so much from inserting a new element - we need to determine neighbours at a move destination index and use them to construct our `Moved` block. Additionally we need to reference the block we want to move (we use its unique `Id` for that).

```
let move replicaId (src: int) (dst: int) (array: Yata<'t>) : Yata<'t> =
  // get the left/right neighbours of the move destination
  let dst = findPosition dst array
  let seqNr = 1UL + lastSeqNr replicaId array
  let left = array |> getBlock (dst-1) |> Option.map (fun b -> b.Id)
  let right = array |> getBlock dst |> Option.map (fun b -> b.Id)
  // get reference to moved block
  let src = findPosition src array
  let moved = array |> getBlock src |> Option.get
  // if our moved block was already subject to move in the past,
  // we need to obtain latest move priority and increment it
  let priority =
    moved.MovedTo
    |> Option.bind (fun dst -> indexOf array dst)
    |> Option.bind (fun idx -> getBlock idx array)
    |> Option.map (fun block ->
      match block.Value with Moved(_,prio) -> prio + 1 | _ -> 0)
    |> Option.defaultValue 0
  let block =
    { Id = (replicaId, seqNr)
      OriginLeft = left
      OriginRight = right
      MovedTo = None
      Value = Moved(moved.Id, priority) }
  integrate block array |> integrateMoved block
```

For now let's forget about `integrateMoved` function (we'll come back to it in a second), and instead focus on `priority`. What do we need it for? The thing is that we might need to move the same block over the course of time more than once, while maintaining the idempotency, commutativity and associativity properties of conflict resolution algorithm. Priority field allows us to recognize that one move happened after another - this way we'll be able to correctly recognize outdated move operations.

Now let's discuss special case of move integration. The basic principle is that: whenever we put a move block into our Yata sequence, we need to check if its target block (the one that we've moved) was not already marked as moved by another operation. If so, then we have a conflict between the two moves. The way to resolve it is simple:

1. The move with higher priority always wins.

2. If priority of both move blocks was the same, it means that they happened concurrently. In this case our winner is the block with greater identifier.

```
let private integrateMoved block (array: Yata<'t>) =
  // since array parameter used here is always copied,
  // we can update it in-place
  match block.Value with
  | Moved(target, prio) ->
    // we need to check if target block was not already
    // moved by another operation, in such case the one with
    // higher priority and ID wins
    let targetIdx = indexOf array target |> Option.get
    let target = array.[targetIdx]
    match target.MovedTo with
    | None -> array.[targetIdx] <- { target with MovedTo = Some block.Id }
    | Some other ->
      // this block was already moved by a different operation
      let otherIdx = indexOf array other |> Option.get
      let other = array.[otherIdx]
      match other.Value with
      | Moved(_, prio2) ->
        // if we have conflict of two move operations over the same block:
        // 1. the one with higher priority field (most recent one) wins.
        // 2. in case of same priority (concurrent move operations),
        //    the one with greater ID wins.
        if prio > prio2 || (prio = prio2 && block.Id > other.Id) then
          array.[targetIdx] <- { target with MovedTo = Some block.Id }
      | _ -> failwith "moved-to field must point to a Moved block"
  | _ -> ()
  array
```

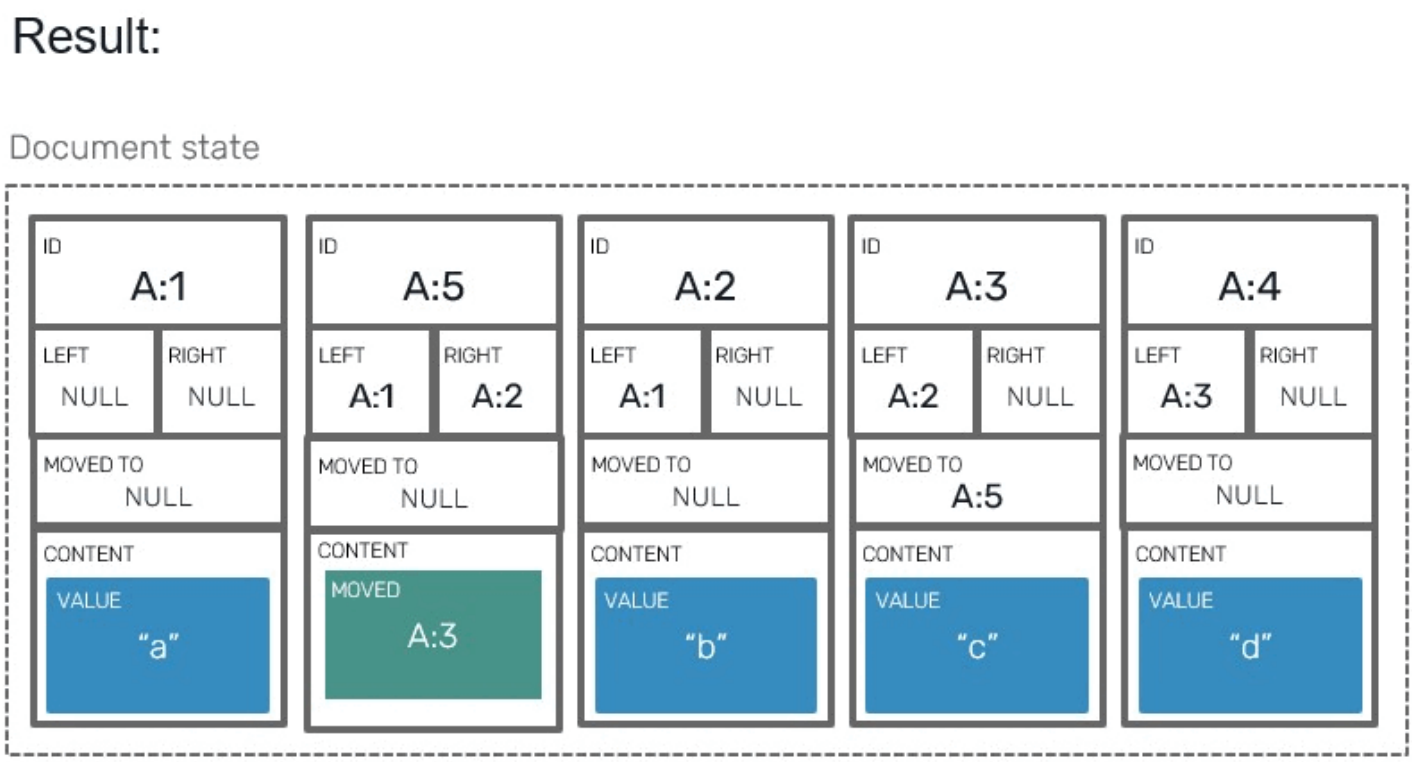
The thing about `integrateMoved` is that whenever we're integrating blocks as part of merge of two blocks, this function must be called after all of the blocks have already been "integrated" into returned Yata collection. It's because `Moved` blocks may refer to target blocks that weren't integrated yet:

```
let merge (a: Yata<'t>) (b: Yata<'t>) : Yata<'t> =
    // ... previous operations
    while remaining > 0 do
      for block in blocks do
        // make sure that block was not already inserted
        // but its dependencies are already present in `a`
        let canInsert =
          not (Set.contains block.Id seen) &&
          (isPresent seen block.OriginLeft) &&
          (isPresent seen block.OriginRight)
        if canInsert then
          a <- integrate block a
          seen <- Set.add block.Id seen
```

```
            remaining <- remaining - 1
        // once all blocks are integrated, resolve conflicts
        // between potential move operations
        for block in blocks do
          a <- integrateMoved block a
        a
```

Reading blocks with respect to moves

The last step in our implementation is all about reading the values. Previously we layed blocks in array in their direct read order, which made producing results easier. Since we have moved blocks in here, we need to prepare our code to make conditional jumps and skips when iterating over blocks:



For this we'll prepare a dedicated iterator, that will respect and order of moved blocks. This is were our newly added fields will be put into use:

1. If we encountered `Moved` block, we're going to make jump into its target. Since in our implementation `Moved` blocks will never be moved themselves, we can just do a simple single-level nested check. *Otherwise our jump target could turn out to be `Moved` block itself, turning jump into recursive process.*
2. If we encountered value block, before yielding it we need to check if it's a valid target:
   - Since we don't tombstone `Moved` blocks, as we iterate over them several ones could trigger a jump-yield procedure, causing false duplication. It's because only one move block is valid at a time (the one which Id is present in `taget.MovedTo` field).

- Additionally we may need to skip over the value block if it was moved elsewhere. Again we use `block.MovedTo` field to determine that.

```
// Returns iterator that produces pair of (blockIndexInYata,block).
// It skips tombstoned and move blocks.
let private blocks (array: Yata<'t>) = seq {
  let mutable i = 0
  while i < array.Length do
    let block = array.[i]
    match block.Value with
    | Moved(targetId,_) ->
      // jump into target block
      let j = indexOf array targetId |> Option.get
      let target = array.[j]
      match target.Value with
      // check if this block was not moved by another operation
      | Value _ when target.MovedTo = Some block.Id -> yield (j, target)
      | _ -> ()
    | Value _ when Option.isNone block.MovedTo -> yield (i, block)
    | Value _ -> ()  // this block was moved elsewhere
    | Tombstone -> ()
    i <- i + 1
}
```

Keep in mind that move operations now also affect the way, how to we count element offsets from client's perspective. Therefore, if we want to correctly map index provided by user onto correct block position, we need to reuse `blocks` iterator as well:

```
let private findPosition (index: int) (array: Yata<'t>) =
  blocks array |> Seq.skip index |> Seq.tryHead
```

Moving ranges of elements

While we won't go through implementation here, let's quickly discuss, how we could extend move operations further to work not only with a single item at the time, but the entire range of elements. It could be useful in some scenarios like moving the entire paragraphs or chapters of characters in a collaboratively edited text document, while others are still editing them.

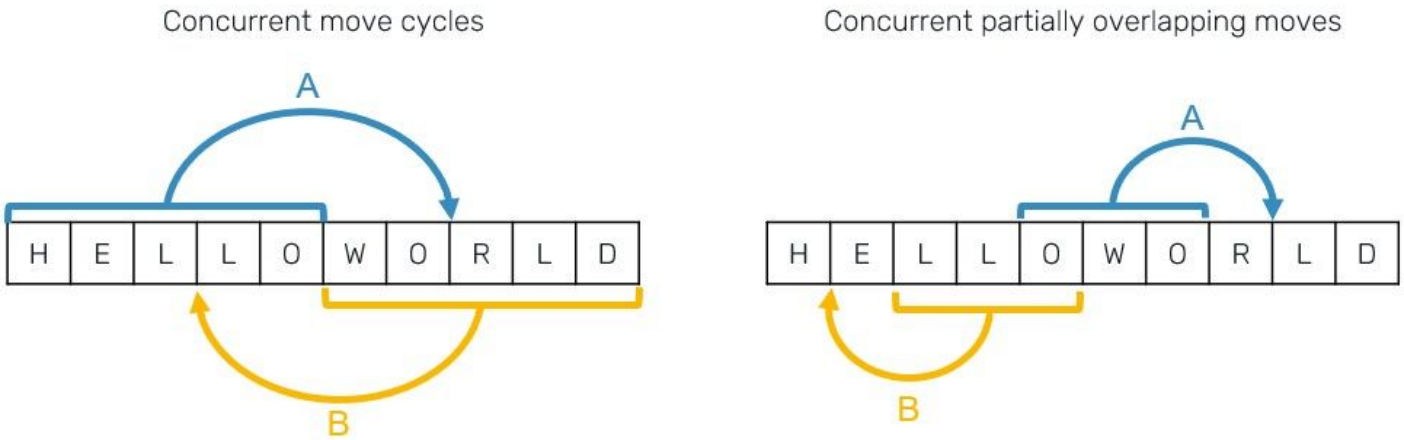
By this approach the basic structure change is that our `Moved` block instead of single `ID` target now contains a range of `(start: ID, end: ID)` marking the first and last block in a moved range. What's important to understand is that insertions in most CRDTs put an emphasis on a space in-between the characters, meaning that we need to decide about what happens with the blocks that are inserted concurrently at the edges of moved range - should we include them into range or not? (*Example: What to do if we move paragraph, while someone is appending text at the end of it? As it happens at the edge of moved range, should it be included or excluded into move operation?*)

Second change is related to reads - so far our `blocks` sequence function performed at most 1-step descending, when a `Moved` block was visited. It was easy, as by our design `Moved` block could never be a target of another move operation. This is no longer a case in move-range approach. As we're working towards full support to moving ranges in Yjs/Yrs, we've introduced a notion of stack:

1. Each time we jump with the iterator due to visited `Moved` block, we put so-called moved frame onto a (lazily allocated) stack structure. This frame contains bounds of currently visited move range and a return position of a `Moved` block itself, so that we know how to go back after we visited all moved blocks.
2. Each iterated block is checked against the latest move frame. Only blocks with `MovedTo` field equals to that frame's return ID are yielded this way. This way we can ensure correct order of visited blocks.
3. Once our iterator reached end of the boundary of moved frame range, we pop that frame from the stack and revert position iterator back to the moved block that caused this frame to appear using frame's return block ID.

This approach is quite similar to how we think about call stack and function activation frames. Some of the optimizations that we use in compilers (eg. tail call optimizations) can also be applied in this algorithm.

Probably the hardest changes are related to conflict resolution of move ranges. It's because merge of concurrently moved ranges may run into hard to reconcile conflicts. Just to show the few obvious ones:



The number of cases grows as we start to add more concurrent cases and introduce different combinations of sequential-concurrent operations. Not all of these could be possibly resolved by a simple agreement between move operations - in some cases the only sensible solution would be to cancel/revert the effect of one or both move operations. Such rollback however means that we need to reapply the previous move operation that was overridden by the one we're reverting. To do so we need to remember all of the previous move operations that happened within the range of the current `Moved` block.

Summary

We just covered the core concepts behind the idea of moving the elements within the CRDTs sequences. While we provided an examples using a YATA algorithm as our baseline, keep in mind that these can be generalized to other structures (eg. LSeq or RGA) as well. If you're interested in further reading, your can also take a look at the slides from my presentation about YATA, which can be found here.

Shelf: easy way for recursive CRDT documents

In the past we already discussed how to build JSON-like Conflict-free Replicated Data Type. We used operation-based approach, with wide support for many operations and characteristics of a dynamically typed recursive documents. However with that power came complexity. Today we'll discuss much simpler approach, which some may find more appealing, even thou it comes with much more severe limitations.

We'll be discussing a Shelf CRDT, which original JavaScript version could be found on github. Here, we'll discuss a pros and cons of this approach and show an implementation in statically typed F# code.

We'll start from defining our value containers:

```
[<Struct>]
type Shelf<'t when 't : comparison> =
  { value: Node<'t>
    version: int }   // used for conflict resolution

and Node<'t when 't : comparison> =
  | Value of 't
  | Object of Map<string, Shelf<'t>>
```

Shelf recognizes only two kinds of values: objects representing multiple entries and scalars treated by the algorithm as an atomic pieces of data (replaced in their entirety during conflict resolution).

If you got some intuition about conflict-free algorithms (if not, I encourage you to start reading here) you know that in order to figure out how to resolve potential conflicts between multiple agents updating the same value concurrently, we need some metadata to keep around to even discover that concurrent updates have happened. Here, we use `version` field for this - it's just a single number incremented every time a corresponding entry gets updated:

```
module Shelf

//NOTE: deeply nested shelf updates will increment version on all
```

```
// nodes on a path from the document root down to the leaf
let set v shelf = { value = v; version = shelf.version + 1 }
```

Ok, so now we have something to compare for concurrent updates. But how merging would work with it?

1. If we merge two `Object` nodes, we recursively merge their corresponding entries. Entry which didn't exist on merge side will just get added.
2. If we merge nodes of different types, we arbitrary allow for `Object`s to take precedence over `Value`s.
3. If we merge two `Value`s, we first compare their versions: greater version number wins. But what if versions were the same? You may have notices that our `type Shelf<'t when 't : comparison>` puts a comparison constraint over `'t`. That's right: since we don't use replica identifiers, we compare values themselves and pick the one that's logically "higher".

With these rules at hand, we can produce an extremely simple merge algorithm.

```
module Shelf

let rec merge a b =
  let cmp = a.version.CompareTo(b.version)
  if cmp > 0 then a
  elif cmp < 0 then b
  else
    match a.value, b.value with
    // objects takes precedence over scalar values
    | Object _, Value _ -> a
    | Value _, Object _ -> b
    // scalar values are compared against each other - greater one wins
    | Value ma, Value mb -> if mb > ma then b else a
    // recursively merge objects entries
    | Object ma, Object mb ->
      let result =
        mb
        |> Map.fold (fun acc kb vb ->
          match Map.tryFind kb ma with
          | Some va -> Map.add kb (merge va vb) acc
          | None -> Map.add kb vb acc
        ) ma
      { a with value = Object result }
```

That's basically all you need to work with this data type. But as you might have noticed, there are strings attached: this approach has multiple drawbacks and we're going to talk about them now.

Tradeoffs

Obvious observation is that our `version` used here is just a simple numeric value. What does it give?

Delta-based replication

Most (if not all) of the time we considered using vector clocks or at least pair `(id, seqNr)`. Shelf doesn't recognize concept of unique replica identifiers, which on positive side makes things easier and reduces the metadata size to only 4-8 bytes per entry. On negative one it makes impossible to simply calculate deltas, therefore forcing full state replication even when only one entry has been changed. What can we do with it?

1. Limit the use case of this approach only to a small documents.
2. Fall back to change version type to make use of vector clocks and dots. This would allow to compute deltas (as we would know which entry has been lately changed by which replica) and introduce branch cutting for potentially faster conflict resolution in deeply nested documents. Of course this comes at the price of higher memory footprint and overall complexity.
3. If you'd consider a push-based replication approach, it would be possible to store delta aside of main Shelf structure. That delta gets reset to zero state every time it's being replicated and it's filled only with copies of entries which have been modified locally by current peer since last replication round. This way we only replicate change sets, but at the flip side we're now responsible for ensuring that these changes were received by remote peers (*popular approach is to use push-based deltas for frequent and cheaper replication then from time to time send a full state to ensure that all changes have been synced*).

Accuracy

Another problem we could talk about is the accuracy of conflict resolution algorithm. We simply use combination of `version` + native comparison of value type stored inside of the shelf to determine which of the conflicting entries to store and which to drop. This means that ie. person who makes two updates of the same entry in a row will always be prioritized over the one who did it once. **It doesn't matter which update was the most recent, only which one is changed more often**. In order to fight that we could use ie. hybrid logical clocks instead of sequence numbers.

The property above, combined with the fact that native type comparison is rarely a desired way to do a conflict resolution, may lead to a situations where conflict resolution produces an unexpected outcomes. In defence of this approach what we're prioritizing here is eventual consistency - having a good accuracy that correctly reflects users intentions requires specialized algorithms and data structures. Here however we value simplicity and low metadata footprint.

Missing operations

You can notice, that Shelf doesn't have a concept of lists. We could mock them by using `Map<int, Shelf<'t>>`, but this approach doesn't cover all use cases ie. we're still unable to differentiate between updating and inserting elements.

Another thing is element deletion, which is also absent. Again we could mock it by having a dedicated tombstone type among the variants of `'t`, but we would need to correctly interpret it during value reading.

I would advise against using Shelf as universal one-size-fits-all CRDT document implementation, as this will inevitably result in raising the complexity over time as more and more uses cases will have to be covered. Instead, let's think about scenarios where the inherent limitations of Shelf could be ignored or they are not much of an issue.

Use cases

Lack of deletion support works strongly against using Shelfs for general-purpose CRDT map implementation. However, if all you need is to apply CRDT merge semantics over existing instances of **statically typed structures**, it's no longer a problem (for statically typed systems objects don't have their fields spontaneously removed at runtime).

Notice that concept of tombstones is also absent here. Since Shelf is also state-based approach, we don't have operations that we need to store and send over. This means, that this structure is good fit for **a frequent updates** that could otherwise produce a ton of quickly-outdated data.

Lack of replica identifiers means, that we're not affected by spontaneous vector clock size explosion (it could happen ie. for long living documents, **whenever a new user session creates a new replica ID**).

Lack of native support for delta-state replication suggests using it for **instances that are small in size or have all of their fields changed frequently**.

With these properties in mind, let's theorize about potential practical applications for this data structure:

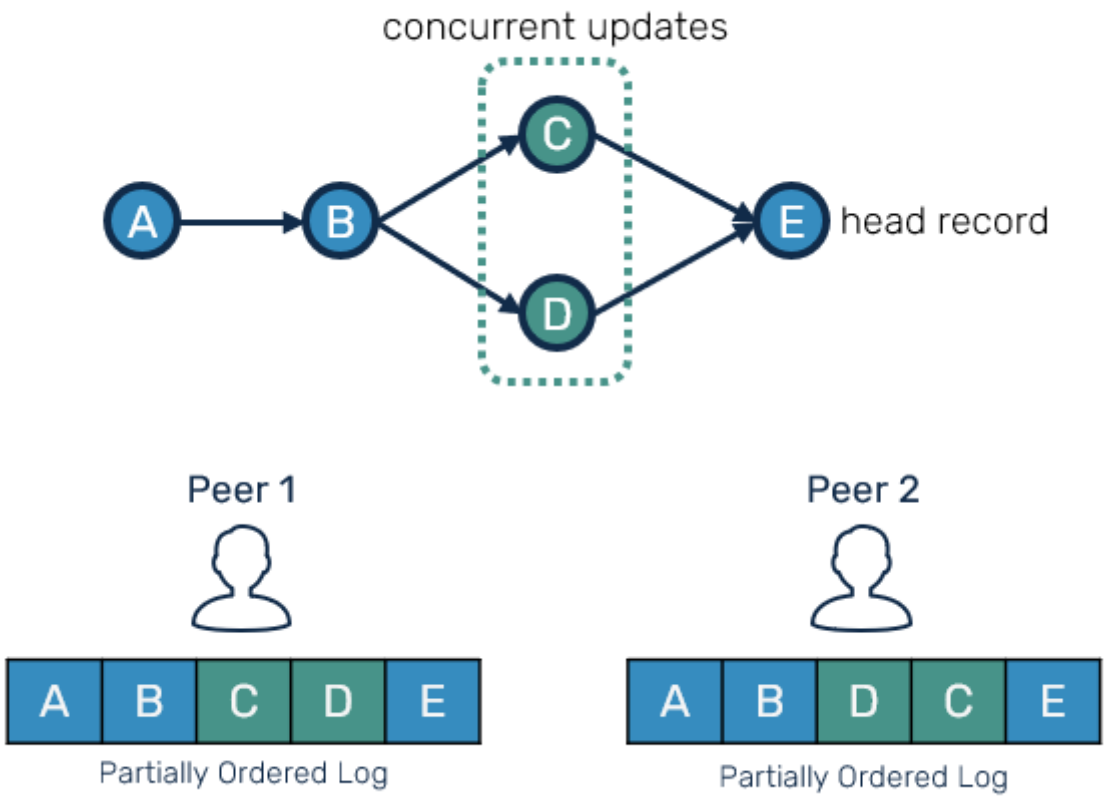
- When working on Yrs we figured out that this approach may be useful for keeping user metadata (cursor positions, name etc.). Since a user may be logged on multiple devices (each one would need its own replica ID), but usually uses only one at the time, inaccuracy of conflict resolution doesn't seem to be much of an issue. Our discussions there were also primary inspiration for this chapter.
- Another thing could be a fleet of vehicles, where we want to track their recent location.
- With potential overhead as low as 4-8 bytes per element, we could even create a **collaborative bitmap canvas** data type on top of Shelf merge algorithm! Imagine canvas which consists of pair: bitmap of RGBA pixels + bitmap of versions per each pixels. Each pixel change also increments a version of a corresponding pixel. This is also nicely aligned with usage of SIMD operations (including shaders!) for merging multiple entries (pixels) at once as well as potential of using well known compression algorithms to optimize replication payload. You can see proof of concept of this approach in this gist.

CRDTs & Security: Authentication

Today we're going to continue exploration of Conflict-free Replicated Data Types domain. This time we'll start designing protocols that focus on a security aspects as first class citizens.

Managing security and permissions in peer-to-peer systems may be quite cumbersome as often there's no single authority to rely on. For now let's concentrate on one thing and try to answer the question: *what problems do we want to solve when we talk about authentication*? We need a way to verify the identity of peer and prove that updates authored by that peer were in fact made by them.

The CRDT replication protocol we're about to implement could be put into operation-based category. We're going to use partially-ordered log of records, each representing some update authored by the peer. These records can have different order on each peer, however they will always preserve happened-before relationships between them and their predecessors.



If this illustration seems familiar to what you've seen in version control systems such as Git, it's because it is. In fact for people familiar with Git, it's a good mental model to begin with as we'll be moving forward.

For user identity we're gonna use a public/private key pair - it's pretty standard approach, used in some distributed version control systems such as Pijul for the same purpose. This way peer's public key will be used as their unique identifier.

For records themselves in the past we used either vector clocks or Lamport clocks: a combination of `PeerId` and some incremental sequence number. This time it won't be enough. If an impostor knew the `PeerId` - which is public - they could forge an update and use it to corrupt the state of the system. Instead we'll use content addressing - namely each record unique identifier is a secure hash

(here: SHA-256) of its content and records which happened directly before it. This way record impersonation won't be possible, as changing the content means changing the ID itself. If this triggers your geek senses and reminds you about Merkle trees, you're right - our log of records works the same way.

One of the downsides of replacing vector versions with content addressing is that we no longer can use it to track causality between one record and its predecessors. We cannot derive predecessors given the hash alone. We'll solve this by adding a `deps` field - a list of parent records IDs, which were the latest records in our log at the moment of committing current records. Usually this would be just a single parent (current head), however when we're having records representing concurrent updates being merged into our log, this number may change.

```
// Unique identifier of a Record.
type ID = [sha256.Size]byte

// Unique identifier and a public security key of a Peer.
type PeerId = [ed25519.PublicKeySize]byte

type Record struct {
    id     ID     // globally unique content addressed SHA256 hash of current Record
    author PeerId // creator of current Record
    sign   []byte // signature used by an author used for Record verification
    deps   []ID   // dependencies: IDs of parents of this Record
    data   []byte // user data
}

func NewRecord(pub ed25519.PublicKey, priv ed25519.PrivateKey, deps []ID, data []byte) *Record
    p := &Record{
        data:    data,
        parents: deps,
        author:  NewPeerId(pub),
    }
    p.id = p.hash()
    p.sign = ed25519.Sign(priv, p.data)
    return p
}

// Returns a content addressable hash of a given Record
func (r *Record) hash() ID {
    h := sha256.New()
    for _, d := range r.deps {
        h.Write(d[:])
    }
    h.Write(r.data)
    h.Write(r.author[:])
    return NewID(h.Sum(nil))
}
```

If you know Git etc. well, you may notice that the resemblance becomes even more uncanny.

Partially ordered log

Next step is to define a partially ordered log. Here we'll use an in memory version for simplicity, but we'll also discuss a basics for SQLite-oriented approach.

The thing about our current approach is that once we append a record to a log we cannot really remove it - pruning could be potentially possible with stabilization approach known from pure operation-based CRDTs and its an interesting topic to cover in the future. For sake of our snippet we'll use simple slice. Since log is append-only, we don't need to worry about index changes, therefore we'll use record indexes as a sort of primary key / clustered index. Additionally we'll add two secondary indexes:

1. Mapping from `Record` ID to its position in a log.
2. List of children for each record. Originally `Record` struct has embedded list of its parents, and this is a payload that we include during the serialization. However for the sake of convenience mapping the other way (parent→child) is also useful.

```
type POLog struct {
    records    []*Record  // append-only log of records
    index      map[ID]int // index of ID to its index position in the log
    childrenOf [][]int    // for each record at related records position,
                          // provide a list of indexes of its children
}
```

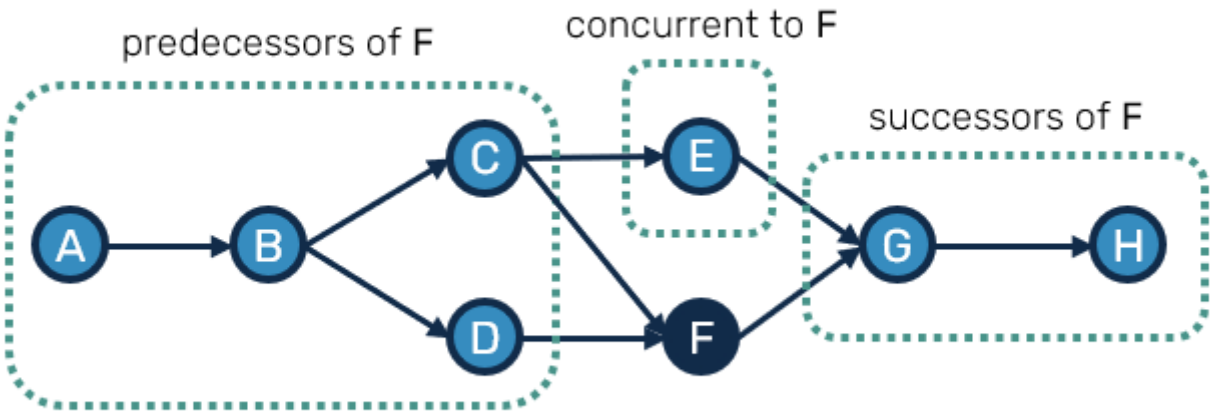
Now, appending process is simple - we need to verify if record has not been forged (verify its signature) and make sure that all of its children are present. We won't append a record until all of its parents were appended before - this would cause a causality teardown. In such case we'll stash them aside in a separate structure.

```
func (log *POLog) Append(p *Record) error {
    if err := p.Verify(); err != nil {
        return err // invalid patch trying to be committed
    }
    if _, found := log.index[p.id]; found {
        return AlreadyCommittedError // duplicate record
    }
    for _, dep := range p.parents {
        if _, found := log.index[dep]; !found {
            return DependencyNotFoundError // parent missing
        }
    }
    i := len(log.records)
    log.records = append(log.records, p)
    log.index[p.id] = i
```

```
    // update parent->children indexes
    log.childrenOf = append(log.childrenOf, nil)
    for _, dep := range p.parents {
        pi := log.index[dep]
        log.childrenOf[pi] = append(log.childrenOf[pi], i)
    }
    return nil
}
```

In order to navigate over the graph of records, we'll also need a way to identify parts of the graph in context of a given record(s):

- **predecessors** - records that must have happen before current record(s), basically their parents, grandparents etc.
- **successors** - records that happened after current ones, so their children, grandchildren etc.
- **concurrent** - neither predecessors nor successors, records that happened without prior knowledge about existence of current record(s).



These are pretty common concepts, that are crucial in order to understand and operate the partially-ordered log.

```
func (log *POLog) Predecessors(heads []ID) []*Record {
    var res []*Record
    log.predecessorsF(heads, func(i int, p *Record) {
        res = append(res, p)
    })
    return res
}

// Returns records that are successors or concurrent to given heads
func (log *POLog) Missing(heads []ID) []*Record {
    v := log.predecessorsF(heads, func(i int, r *Record) {
        /* do nothing */
```

```
        })
        var res []*Record
        for i, p := range log.records {
            if !v.Get(i) {
                res = append(res, p)
            }
        }
        return res
    }

    // Traverses over the head records and their predecessors, executing given
    // function f with patch index position in a log and record itself.
    // Returns a Bitmap which includes indexes describing all visited records
    func (log *POLog) predecessorsF(heads []ID, f func(int, *Record)) Bitmap {
        q := log.indexes(heads) // queue of records to visit
        visited := NewBitmap(len(log.records))
        for {
            i, ok := pop(&q) // get next record to visit in FIFO order
            if !ok {
                break // q is empty
            }
            // check if we haven't visited this patch already
            if !visited.Get(i) {
                visited.Set(i, true)
                p := log.records[i]
                // append parents in to-visit queue
                q = append(q, log.indexes(p.parents)...)
                f(i, p)
            }
        }
        return visited
    }
```

Persistent stores

Since the partially ordered log usually is not truncated, it could possibly outgrow the available memory. Moreover once the state is established, most of the records are rarely used. It's natural to build a log backed by some persistent store. It could be a key-value based one (like LMDB or Sanakirja used by Pijul), but it could also be a higher level database.

Below you can see a simple SQL schema and queries that can be used to store and navigate over the directed graph of records represented as a `nodes` and `edges` tables.

```
    -- records
    CREATE TABLE IF NOT EXISTS nodes(
        id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
        hash NCHAR(32) NOT NULL UNIQUE,
        author NCHAR(32) NOT NULL,
        sign NCHAR(64) NOT NULL,
        content BLOB NOT NULL
```

```
);
-- relationships between records
CREATE TABLE IF NOT EXISTS edges(
    parent_id INTEGER NOT NULL,
    child_id INTEGER NOT NULL,
    PRIMARY KEY(parent_id, child_id),
    FOREIGN KEY (parent_id) REFERENCES nodes(id),
    FOREIGN KEY (child_id) REFERENCES nodes(id)
);

-- get all predecessors of record 'xyz'
WITH RECURSIVE predecessors(id) AS (
    SELECT id as parent_id FROM nodes WHERE hash IN ('xyz')
    UNION
    SELECT parent_id
    FROM edges s
    JOIN predecessors p ON s.child_id = p.id
)
SELECT * FROM predecessors;

-- get all edges or nodes created after or concurrently to record 'xyz'
WITH RECURSIVE predecessors(id) AS (
    SELECT id as parent_id FROM nodes WHERE hash IN ('xyz')
    UNION
    SELECT parent_id
    FROM edges s
    JOIN predecessors p ON s.child_id = p.id
)
SELECT child_id FROM edges s WHERE child_id NOT IN predecessors;
```

Given the proximity between the concepts of distributed version control systems and topic we're describing here, you may find interesting that Fossil - a version control system developed and used by SQLite creators - is in fact backed by SQLite itself.

Replication

Basic replication protocol operates on two concepts: upon connection with remote peer try to pull missing records and once this is done then be ready to push and receive all new records onwards.

For operation-based approach we used in the past we were able to represent state of the world using vector clocks. Each peer at the beginning was sending its own vector clock to its remote counterpart, which then was able to compute what new records happened since given vector clock.

Since we don't rely on vector clocks here, we'll make each peer send a list of its latest record IDs instead. Unsurprisingly we'll call them heads (another analogy to Git). Unless there were several updates made concurrently by the different peers, `heads` list will usually consist of the single element.

But how do we figure out which records should be send back to requester? Fortunately, a `log.Missing` method we've shown before is capable of returning records which happened after or concurrently to provided heads. However, what if the requester had sent heads of records we've never seen so far? There is more than one way to figure this out, but we're going with the most brute force solution:

```
type Peer struct {
    pub         ed25519.PublicKey  // Peer's public key, equals to Author
    priv        ed25519.PrivateKey // Peer's private key, used for signing
    heads       []ID               // all newly created records on this peer will refer to hea
    store       *POLog             // record log

    stash       *Stash             // Stash used as a temporary container for records which ar
    missingDeps map[ID]struct{}    // "known" missing parents preventing records from stash to
}

func negotiate(src *Peer, dst *Peer) error {
    heads := src.Announce()        // A sends its own most recent PIDs
    missing := dst.NotFound(heads) // B picks IDs of the records it has not seen yet
    for len(missing) > 0 {         // until all missing records are found
        records := src.Request(missing) // send request to A asking for missing PIDs
        err := dst.Integrate(records)   // B tries to integrate A's records
        if err != nil {
            return err
        }
        missing = dst.MissingDeps() // B recalculates missing records
    }
    return nil
}

func (p *Peer) Integrate(rs []*Record) error {
    changed := false
    for _, r := range rs {
        if err := r.Verify(); err != nil {
            return err // remote patch was forged
        }
        if p.store.Contains(r.id) || p.stash.Contains(r.id) {
            continue // already seen in either log or stash
        }

        // check if there are missing predecessor records
        var missingDeps []ID
        for _, dep := range r.parents {
            if !p.store.Contains(dep) {
                if !p.stash.Contains(dep) {
                    missingDeps = append(missingDeps, dep)
                }
            }
        }

        if len(missingDeps) > 0 {
```

```
            // if there were missing predecessors, stash current record
            p.stash.Add(r)
            for _, dep := range missingDeps {
                p.missingDeps[dep] = struct{}{}
            }
        } else {
            // all dependencies were resolved, add record to log
            if err := p.store.Append(r); err != nil {
                return err
            }
            delete(p.missingDeps, r.id)
            changed = true
        }
    }
    if changed {
        // if any record was successfully integrated it means, we may
        // have stashed records that have their dependencies resolved
        // and are ready to be integrated as well
        p.heads = p.store.Heads()
        rs = p.stash.UnStash()
        if len(rs) != 0 {
            return p.Integrate(rs)
        }
    }
    return nil
}
```

Basically we traverse the graph of records from the latest to the oldest. Each traversal step requests for a batch of concurrently committed records and checks it their parents dependencies are satisfied. If some parents were missing from current log, we stash current records aside and then fetch the parents. Do this recursively until all dependencies have been fetched. Once this is done, reapply stashed records in LIFO order.

Performance considerations

While the ideas we presented may sound tempting, there are some serious performance issues that don't let us scale this approach over certain point:

- **Metadata overhead** which is pretty popular problem in all CRDT-related algorithms. For practical reasons the minimum is 2 * 32B = 64B per committed record (ID of the current record + ID of its parent dependency). While it doesn't sound bad, it can be quite a lot for some scenarios eg. text editing, where each key-stroke is separate operation: at the moment there are no proved ways to make this algorithm work with update splitting/squashing.
- Our algorithm itself is very naïve and comes with one serious downside - each loop iteration is basically a network RPC. If there's a long missing record dependency chain between two peers it means multiple roundtrips trying to fetch all missing parents. Thankfully due to wild popularity of Git there are many articles, papers and implementations of record negotiation/reconciliation

between peers. While some of them are pretty Git specific, others are capable targeting even closer to our use cases. We might try to cover them in the future.

Unfortunately, there are still unanswered questions in this problem domain, which at the moment may make this approach hard to scale for real-time collaboration. It's also the reason, why we can observe its wider adoption when we talk about manually committed changes (present ie. in distributed version control) instead.

References

- Github repository branch with full code, the snippets in this chapter come from.
- HyperHyperSpace which is a JavaScript library build around security and CRDTs in mind. It shares a lot with the ideas we had presented here.
- Pijul which is a CRDT-based distributed version control system we already mentioned at the beginning.
- Byzantine Eventual Consistency paper discussing possible improvements to replication protocol used by content-addressed partially-ordered log we presented here.