ScuttleSort: Incremental and Convergent Linearization of Tangles

2022-05-09 christian.tschudin@unibas.ch

Abstract: We present an incremental topological sort algorithm that permits to linearize tangled append-only feeds in a convergent way with modest effort. Moreover, our architecture permits to synchronize the linearization with a local replica (database, GUI) such that on average one to three edit operations per log entry are sufficient, for reasonably sized tangles. We discuss the use and performance of "ScuttleSort" for append-only systems like Secure Scuttlebutt (SSB). In terms of conflict-free replicated data types (CRDT), our update algorithm implements a partial order-preserving replicated add-only sequence ("timeline") with strong eventual consistency.

1) Introduction

Merging several tangled append-only logs into one linearization is a core task in SSB: starting from the causality relationship between entries of different logs (which overall only yields a *partial* ordering) we wish to derive a *totally ordered* sequence of events. The resulting linearization (aka "linear extension of the partial order") makes chat conversations understandable to a human reader and is essential for getting the moves in a chess game right. This sorted order has to be independent from the order in which log entries from the various participants were received, as otherwise inconsistent shared state would emerge.

To the best of our knowledge, there is no code template (in SSB nor any other decentralized project working with tangled event streams) that would show how to do this *incrementally*, yielding a convergent linearization despite concurrent append events. In this writeup we describe such an algorithm where we also discuss the derivation of edit commands for a database and a GUI, all while adhering to the constraint that linearization should be incremental and as lightweight as possible. The name "ScuttleSort" seems appropriate not only because of the SSB context but also the algorithm's approach of scuttling in swift runs towards the grand goal of total order.

1.1) Stream Processing of Log Entries

Conceptually, we wish to build a processing pipeline that looks like this:

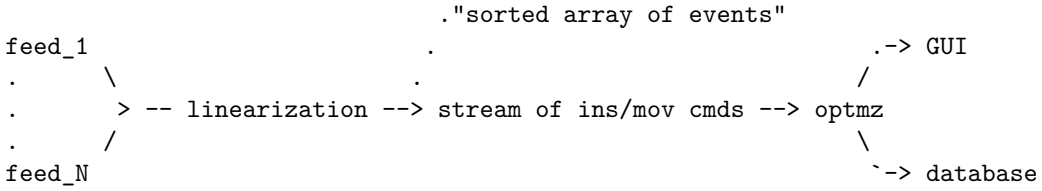

Figure 1: Turning 'tangled append events' from multiple sources into a single stream of update messages for the GUI.

To the left, multiple feeds generate `append` events that can carry additional tangling information: The genesis event of a feed has no predecessor but all other events will at least reference the previous event of their append-only log. Optionally, an event can also reference entries in other feeds, leading to a tangling of the feeds. Together, these events form a directed acyclic graph (DAG) where each append event points to events that predate it. Our goal is to merge several input feeds into one output stream that permits to continuously update for example a GUI. The output stream cannot be a stream of events in the sense of append-only logs, as the arrival order will lead to reordering of previous events over time. However, the output stream can be a *stream of update instructions* that carry the necessary information to convert an existing linearization into a next version of it.

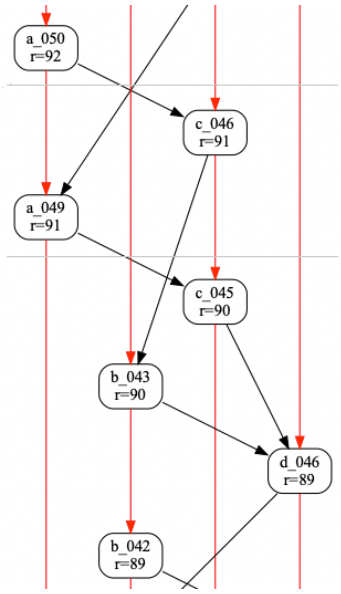

Figure 2: Tangled SSB events in four hash chains (red). Time is pointing upwards: event c_045 is more recent than d_046.

As an example of a tangle, in Figure 2 we see four hash chains, running vertically, where pointers represent references from one entry to another (the pointer is materialized as a hash value of the target entry). The figure also shows a node's computed rank (see Section 2) where for example event `a_049` (in chain `a`) happened concurrently with event `c_046` in chain `c` as both have rank 91. The rank is an internal value that can change over time. The linearization is externally visible as the vertical position, or height, of a node which stands for the logical time that is assigned to a node in the DAG after having linearized it. Also this value can change over time as new events or feeds are added to the graph. Our goal is to generate instructions such that when event `c_046` is received, we learn that this new event has to be inserted between `a_049` and `a_050`.

For concurrent (aka "causally unrelated") events there is no objective total order of the DAG and several linearizations are valid. This is a problem because different users could compute different valid linearizations and potentially draw different conclusions although they ingested the same feeds and all their events (but in different order). Fortunately, we can *declare* a convention that will lead to a consistent ordering. One trick is to use the hash of an event (called messageID in SSB) and to lexicographically sort events that have the same rank, thus are concurrent. In the example tangle from the figure above this trick has been applied: events from log `a` have always less height than equally ranked events from "lexicographically higher logs". In a real system one would use the event's hash value, not the log identity (used here for visualization purposes only). But as we will see later does lexicographic order by itself not lead to a unique linearization and needs additional "shape constraints". Nevertheless and in anticipation of a solution with shape constraints, let's assume already here that a single (convergent) linearization can be computed by all parties if they have the same input, and let's look at the consequences this has.

1.2) Consistent Total Order Induces a Shared Array

The outcome of a convergent linearization of several tangled feeds can best be thought of as a **shared array**: all observer will compute the same consistent array. The array grows with each new event (that was originally appended to one of the input feeds), resulting in the insertion of that event somewhere in the shared array and potentially also the rearrangement of other array elements. Because elements are

never removed from the array it suffices to have an `insert` and a `move` command to convey updates. The task, as shown in Figure 1, is thus to transform the multiple incoming event streams into a single stream of *instructions* that update the shared array.

In our architecture we will feed that stream of update commands first to an optimizer module `optmz` and then to a database and potentially to some interactive GUI where a user can scroll through the linearized events. The database is important for persisting the accumulated array's state, used foremost when restarting the GUI: The GUI will then first load (portions of) the array from the DB for an initial rendering of the content and then, in parallel with the DB, start to consume the command stream.

Given such an architecture it becomes easy to write a GUI that incrementally updates the screen for log entries "as they are received". If a copy of the array is linked to the GUI using some model-view-controller framework, there is literally no additional logic needed to render for example a chat room:

```
msg_list = []   # list of chat msgs, linked to the GUI via some model-view-ctrl

PROC on_insert(pos, content):   # "ins" update
    msg_list.insert(pos, content)

PROC on_move(from, to):         # "mov" update
    e = msg_list(from)
    del msg_list[from]
    msg_list.insert(to, e)
```

2) Ranking (Topological Sort)

Sorting elements that obey partial order can be done with a topological sort algorithm that assigns to each element a rank. We will make use of the rank, which is an integer value, to reflect logical time: elements that are concurrent are ranked identically while a happened-immediately-before relationship results in a rank difference of exactly one. In Figure 3, arrows show this "happened-immediately-before" relationship. The result can be represented as a sequence of sets `{D,E}{B,C}{A}{F}` where each set contains concurrent events.

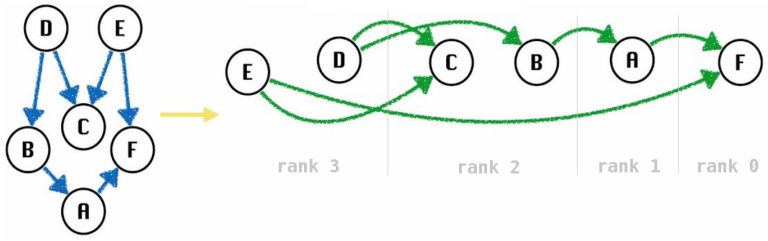

Figure 3: Topologically sorting a DAG and grouping nodes by rank. Note that the arrows represent $\leq$ ("D before C"), not SSB's hash pointers as in Figure 2.

Typically, the API for topological sort algorithms requires the whole causality graph to be available as an input. Because for such algorithms the sorting time is linear in the number of nodes plus the number of edges [ref to Wikipedia], this means that the longer our shared array gets, the longer it takes to insert a new element. Intuitively we expect a different behavior for SSB where events are only added "at the end" of the append-only logs: Our hope is that a new event can be ingested in constant time (in most cases), i.e. the linearization has a bounded number of graph exploration steps and a bounded number of edit commands for the corresponding shared array, which would permit even low-powered devices with minimal database support to untangle a set of incoming feeds in real time.

2.1) Incremental Topological Sort

An *incremental* topological sort algorithms allows to add elements one by one. Substantial acceleration can be obtained by keeping a linearization of the current graph, as it allows to limit the depth of search when traversing the reachable part of the graph. This is also the approach e.g., taken in [PEA2006].

The following skeleton shows a simple incremental topological sort algorithm that makes use of already computed ranks (we copy the presentation from [SIG2016]):

```
AddEdge(v, w) {     // "v less-or-equal w"
    nodesVisited = [];
    w.Cycle = true ;
    hasCycle = Visit(v, w, ref nodesVisited);
    w.Cycle = false;
    foreach(node in nodesVisited)
        node.Visited = false;
    return hasCycle;
}

Visit(parent, child, ref nodesVisited) {
    parent.Visited = true ;
    nodesVisited.Add(parent);
    if (parent.Cycle)
        return true;             // stop and report cycle
    if (parent.Rank <= child.Rank) {
        parent.Rank = child.Rank + 1;
        foreach (inEdge in parent.incoming) {
            if (!inEdge.Visited or inEdge.rank == parent.rank)
                if (Visit(inEdge, parent, ref nodesVisited)) // inEdge becomes parent
                    return true; // stop and report cycle
    }
    return false;
}
```

The following observations and additional details apply:

- We use the term parent and child for two nodes linked through an $\leq$ edge (the parent "comes before" the child node i.e., it is older).
- When adding an edge $v \leq w$, we first check whether a parent node v exists, and create one if not. A new node is initialized with rank 0 which stands for "now".
- Similarly, a new child node w is created if necessary. If it was not existing, it also receives value 0 as its starting rank.
- A discovery wave is then started that propagates edge-by-edge "towards the past", pushing the child's rank value along and incrementing this value for each edge that is traversed.
- The newest nodes ("tangle tips") will have a rank of 0 and the oldest node will be in the rank with the highest rank value.
- An essential data structure is a child node's `incoming` array which contains all parent nodes that have a $\leq$ relationship with the given child. In other words, in this array a child points to all its parents. Note that these pointers are in opposite direction of the "happened-before" arrows.

This latter artifact, the *incoming* array, is exactly what SSB implements in form of the `previous` field (the hash chain implementing the append-only log abstraction) as well as cross references (found inside a log entry) that point to other log entries. It is thus tempting to directly map the above topological sort algorithm to SSB. However, this does not work out well ...

The main problem is that SSB mainly appends new *children* nodes, which happens to the right of above Figure 3, and only seldomly parent nodes (this happens if delivery of log entries is delayed). The algorithm requires that a new node starts with rank 0, which forces the child's parent nodes to have at least rank 1 ... while typically the parent node got its rank 0 in the round before, thus has to change its rank. This creates a wave that more or less renumbers the whole graph down to the genesis entries to the left, following a cone that starts from the new child node. From a runtime perspective this is unfortunate (having to spend more and more time as the graph becomes bigger) and also counter-intuitive: we should only have to visit a few recent parent nodes when appending a new node, not the whole graph's history. Operationally we also point out that the recursive nature of the `Visit()` procedure would have to be changed to an iterative form, as some languages have an internal recursion limit (in Python this limit is set to 1000 by default), as otherwise tangles would be limited to some maximum height.

Based on this insight we now turn the problem upside down and design a topological sort algorithm where rank 0 stands for genesis time and the rank values increase as time passes by.

2.2) "Upside Down" (forward) Incremental Topological Sort

As we explained in the previous section, assigning rank 0 to recent SSB append events results in a renumbering of a potentially large portion of the graph. In this section we want the reachability wave to work in the opposite direction: starting from the cause (and its rank) we will visit all effects (and add 1 to the rank as we go), assuming that most of the time the cause has already been registered and ranked while it is an effect that is added to the graph.

The following code skeleton of this "forward strategy" has the same structure as the classic "backward strategy" and differs mostly in the recursion step, but also in the assignment of rank values: rank 0 now stands for the past and values increase for each parent-child edge. We discuss the two strategies immediately after the code.

```
AddEdge_forward(v, w) {     // "v less-or-equal w"
    nodesVisited = [];
    w.Cycle = true ;
    hasCycle = Visit(w, v.rank, ref nodesVisited);
    w.Cycle = false;
    foreach(node in nodesVisited)
        node.Visited = false;
    return hasCycle;
}

Visit_forward(child, parent_rank, ref nodesVisited) {
    child.Visited = true ;
    nodesVisited.Add(child);
    if (child.Cycle)
        return true;             // stop and report cycle
    if (child.Rank <= parent_rank) {
        child.Rank = parent_rank + 1;
        foreach (grandchild in child.Outgoing)
            if (Visit(grandchild, child.rank, ref nodesVisited))
                return true; // stop and report cycle
    }
    return false;
}
```

Figure 4 depicts the two graph exploration strategies. In the classical incremental topological sort algorithm, a new child node obtains rank 0 and this new assignment (incremented by one) is pushed to all parent nodes and recursively to all reachable nodes. In the other case, shown in the subfigure to the right, the recursion starts with the parent node, whose rank is assumed to be known, and rank values increase as the recursion wave explores all nodes affected by this parent.

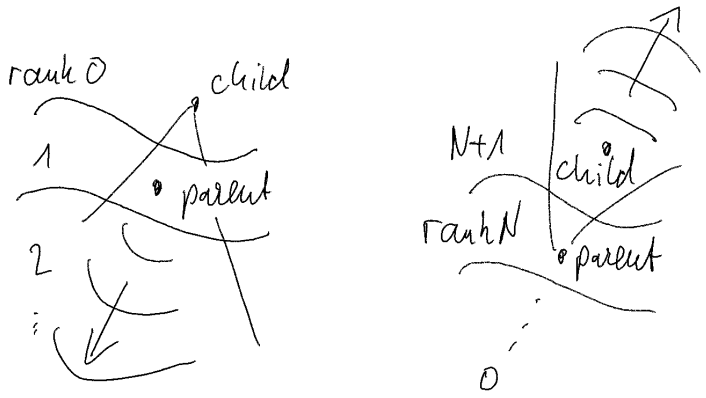

Figure 4: Two rank assignment strategies: backward (left) and forward (right).

2.2.1) Implementing the "Outgoing" Array (List of Children) While it seems obvious why the forward strategy works better for SSB, we point out that this forward strategy requires an additional data structure "Outgoing" where a (parent) node has information about all its successors (children) nodes. This information is not existing in SSB's append-only log and must be gathered and stored on top of it.

Implementation-wise, the "Outgoing" array is inconvenient to have as its size (=number of children) cannot be predicted: any number of feeds could decide to reference the parent node, and can continue to cross-reference a parent node in all future. However, if one restricts each node to have **at most two cross-references**, one can implement this open list of children nodes with constant memory per node, as follows:

From SSB's event encoding, each node has a predecessor reference (`prev`) for the hash chain, and now is allowed to have at most one "tip reference" for tangling, like Iota does. In the receiver's graph representation of a SSB event we implement the `Outgoing` array as three additional slots:

- the first slot a) points to the successor in the append-only log, where SSB guarantees that there is only one such successor
- a second slot b) is used to store the first element of a linked list of all successors which referenced this node via a tip cross-reference
- the third slot c) is used to store the linked list i.e., all siblings which pointed to the same tip.

The use of the three slots for the `Outgoing` array is shown in Figure 5 where log entry *k* has successor entry *k+1* (slot a) and a list of all successors which cross-reference the entry *k*, this list being stored in the c) slots *of these referencer entries*.

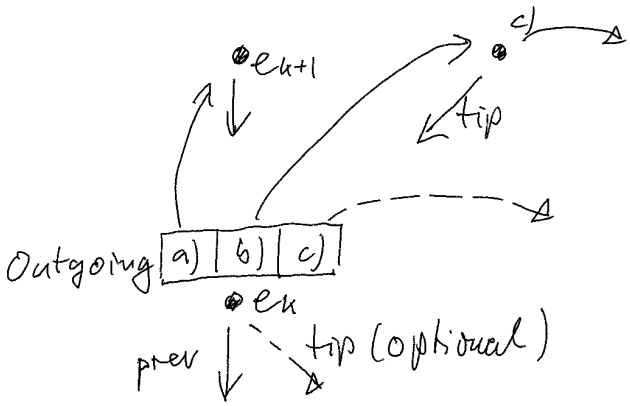

Figure 5: Limiting tangling to one cross-reference per log entry (beside the hash chaining via the prev field) permits to allocate a fixed-size Outgoing array.

2.3) Sorting "With Style" (Shaping the Rendering of Tangles)

The result of topological sorting is not unique, in general. For example, the two linearizations in Figure 6 differ but both are valid, just that the `a_022` event got a different rank assigned. Consider also the causality chain `d_020, b_020, d_021` which is tight, but `c_019` could be pushed into the future, for example have rank 40 and consequently appear in a different position of the linearization.

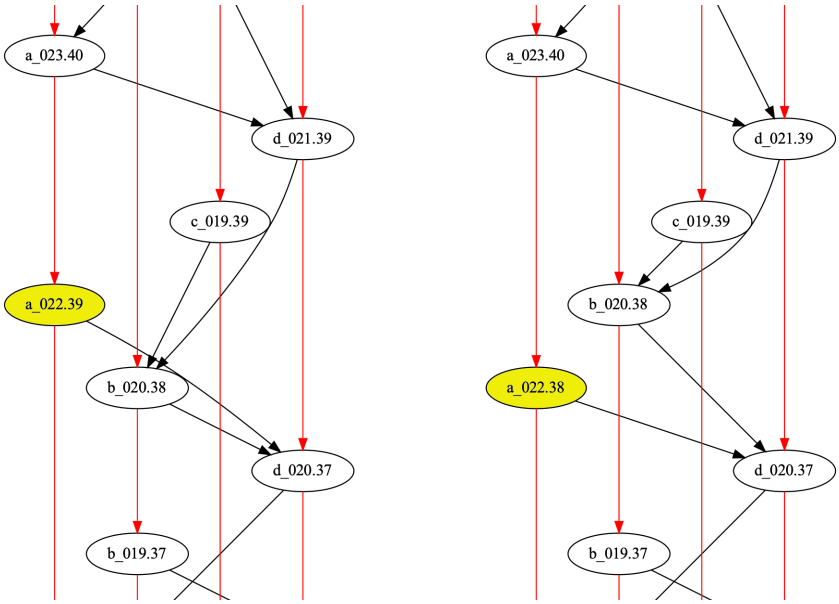

Figure 6: Example for choice in linearization that involves different ranks and that cannot be solved by lexicographic sorting.

The problem with both the classic "rank towards the past" algorithm and the "rank towards the future" algorithm is that the final rank depends on the order in which events are added, which is bad news: while append events for the same feed are strictly ordered, the newest events from different feeds will be delivered in non-predictable order. This results in each SSB receiver potentially computing a different linearization, although that overall they all received the same log contents.

However, using a “style”, there is a chance to fix this behavior. The challenge consists in finding an appropriate positioning rule and in identifying the right moments in the sorting process *when* stylistic rules should be applied, as they can have long-ranging effects of reshuffling node ranks throughout the graph. The result hopefully is a linearization that for all participants “has the same shape”.

2.4) ScuttleSort

With `ScuttleSort` we found an algorithm that produces convergent sort results regardless the order in which new nodes are added to the graph, as long as they respect the append-only principle of SSB. The ScuttleSort algorithm is shown in the Appendix - it is an extension of the “forward algorithm” of section 2.2.

The ScuttleSort algorithm works by adopting a “gravitational style” that drags new events to the lowest level possible (minimal rank). This is implemented by starting with the lowest possible rank and let an event “bubble up” to a next lowest energy position, if necessary.

Instead of having to consider the whole graph we can restrict our work area to the nodes that we recursively visited during the rank determination. These nodes, together with the new node, are sorted by rank and their new position determined via a bubble-up phase as implemented in the `_rise()` method.

As can be seen in Figure 7 in Section 3, can a new event lead to major shifts of many other events' positions. This happens if the new event was already referenced in the past but it is only now that we receive it and can determine its rank (on which the rank of the other events depend), leading to a cascade of bubbling up (re-ranking).

An event that has no predecessor, hence no “stepping stone” from which it could bubble up, is added at the bottom of the linearization, subject to lexicographic sorting among all nodes with rank 0.

In the appendix we show the full ScuttleSort algorithm for Python. We ran extensive simulations where the same tangled events were delivered in different orders, and the same linearization was obtained all the times.

Several aspects regarding our implementation are worth highlighting:

Our node data structure keeps an `indx` field which records the event's position in the linearization. We do this (extra effort to keep this information up to date) for performance reasons, instead of using e.g., Python's built-in `index()` function. Without this `indx` field, ScuttleSort would need to call the `index()` function on the (linearization) list, which has linear search time. That is, knowing the *position* of a node, on top of its rank, is an important run-time optimization, especially for long tangles.

There are cases where elements are inserted at the beginning of the linearization, which now has the consequence that we have to change the `indx` field of all following list elements, hence a linear run-time penalty that grows with the tangle length. While this is unfortunate, these insertions only happen rarely (namely when a new feed joins the tangle). We made sure that insertion at the beginning of the linearization only happens when absolutely necessary.

Deviating from the code skeletons shown in 2.1 and 2.2, we converted the recursive tangle exploration (`visit`) into an iterative version as otherwise we could run out of stack space at run-time.

The optimization module `optmz` currently detects and converts two patterns in the instruction stream:

- `ins(X,eventID); mov(X,Y);` → `ins(Y,eventID);`
- `mov(U,V);mov(V,X); ... mov(Y,Z);` → `mov(U,Z);`

Optimization is done for single events only (no cross-event optimization).

3) Synchronizing the Linearization with an External Replica

As described in Section 1, is the linearization of the DAG subject to change, potentially anywhere in the sequence. In fact, the arrival of a new element can lead to a reshufffeling of major parts of the linearization.

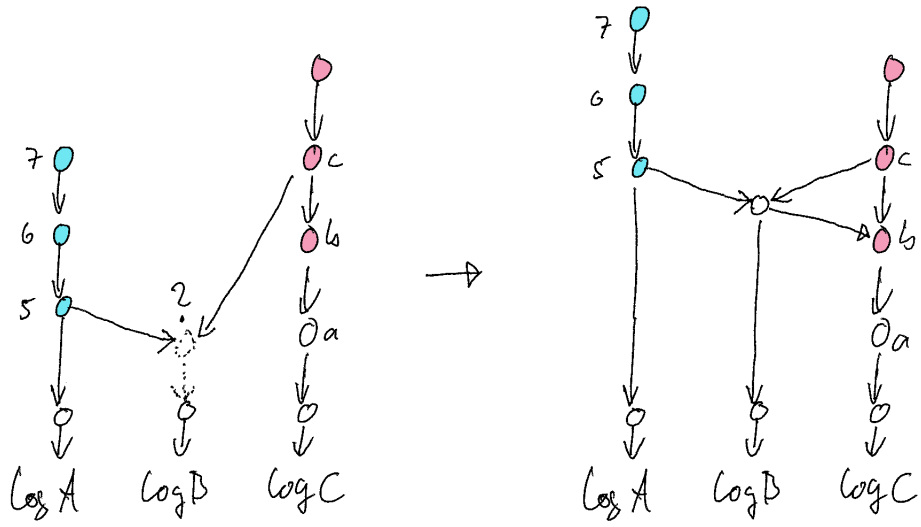

Figure 7: Example of far-reaching consequences of having to rise a single entry in the linearization.

As an example, consider the configuration on the left of Figure 7 where the two logs A and C contain elements that refer to an entry of log B that has not been received yet. A provisional linearization could let the sequence of entries 5/6/7 start earlier than the sequence of entries a/b/c, with some interleaving. When the outstanding event is finally received (right side of Figure 7), it turns out that it references an element of the a/b/c chain, hence forces the 5/6/7 chain to be placed higher, which in turn can force yet another chain in some forth log to rise etc.

ScuttleSort translates these internal position changes into “mov” commands such that replicas of the linearization can easily be built and kept in sync. Based on the above example it seems hard to predict, in general, how much change a single new event can trigger, and what are the dominant parameters that drive this number. However, first insights from simulations show that the single most relevant parameter is the number of involved feeds, while the impact of the length of the linearization is marginal, as we show in the following section.

4) Evaluation

We were interested in the scaling properties of ScuttleSort both for the height and the width of a tangle.

Regarding height, 32K to 512K log events were generated; regarding width, the number of feeds hosting these events was varied from 16 to 1024. For example, 521K events and 16 feeds means that each feed had *approximately* 32K log entries (because event appending is randomized). At each generation step, two feeds were randomly picked to extend their feed. Each event had two backlinks: one for the hash chain, and one to the currently longest feed (itself excepted). Figure 8 shows that ScuttleSort correctly reconstructed that there are always two events per rank.

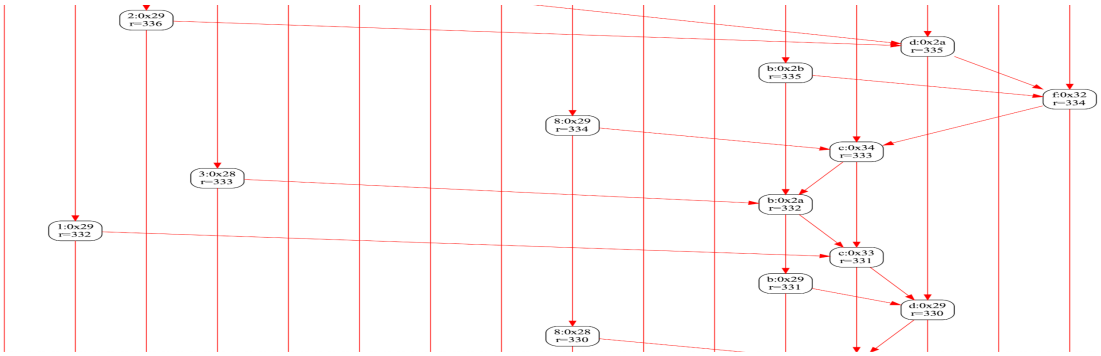

Figure 8: Clip from a 16 feed scenario (each rank has two events).

As second required input to the evaluation is the delivery strategy. Clearly, playing back the appended events in the sequence they were generated results in a trivial reconstruction task. If, however, the network will delay some feed updates, ScuttleSort will be forced to provisionally place events that reference not-yet-delivered ones, and later correct these placements. We chose an aggressive "random-feed" model i.e., at each round, an event from an arbitrary feed was picked which most of the time is an event that references another event that has not yet been delivered. In reality we expect a much more benign behavior because chatty feeds are delivered quickly (and all others remain silent and as a consequence do not inject any premature cross-references).

The following tables summarizes our measurements that we also present in chart form farther below:

tangle width:	4	8	16	32	64	128	256	512	1024
32K	2.5	3.0	3.6	4.7	7.1	10.4	13.1	16.8	21.5
64K	3.8	2.9	5.7	4.4	7.0	10.1	13.2	17.0	22.0
128K	1.7	2.7	3.7	5.3	6.6	9.2	13.9	18.1	24.8
256K	2.6	2.4	3.5	5.8	6.9	10.6	14.6	19.7	24.5
512K	1.6	3.3	2.9	6.2	7.3	10.7	15.0	19.3	26.0

Table showing the average number of edit instructions to the linearization, per added log entry. Columns indicate the width of the tangle (number of feeds), rows share the same total number of log entries (height of the tangle).

Figure 9 shows the findings where we report the average number of edit commands for each evaluated configuration. (We need to do more runs and average these numbers). For example, the 512K events/16 feed scenario required around 3 to 4 edit instructions per received log entry to incrementally update the linearization, while in the 64 feed case, this number increases to approximately 7.

From these numbers we see that ScuttleSort is almost insensitive to the height of the tangle (number of events): in all choices of number of feeds, the number of edit commands per log entry increases only slightly as we double the number of events (2% per doubling of the event count).

Scaling the number of involved feeds however leads to a growth of approx 30% for each doubling of the feed number (while keeping the number of events constant). As observed above, this is due to the delivery model: real-life data for real-life delivery behavior is needed in order to know the typical number of edit commands per log entry.

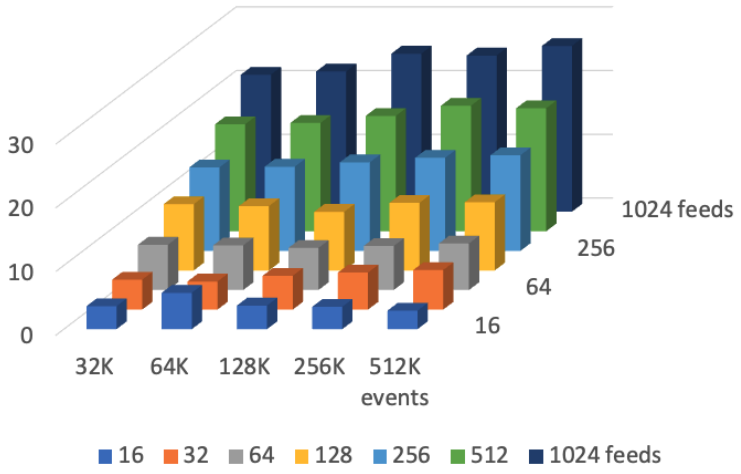

Figure 9: Average number of edit commands per ingested log entry, in dependency of total number of ingested entries (tangle height) and number of feeds which contain them (tangle width).

Our evaluations sample the configuration space for many scenarios which we would not recommend. For example, in the future, all communications should be private, wherefore private groups with explicit membership management will become the norm. Private groups beyond 32 or 64 members are difficult to really keep private, wherefore we conjecture that because of bounded tangle width, in the worst (delayed delivery) case we have to expect an average of 4 to 6 edit instructions per log entry. But even for uncommon tangles of width of 64 or beyond, it seems that the load will be predictable and bareable.

The height of a tangle is not an issue with ScuttleSort, and by including configurations with up to 512K log events we are confident to have covered very long lasting tangles.

Overall, the really good news is that for small-scale coordination (2 to 4 feeds, for games, joint editing etc), incremental topological sorting has almost zero cost and for all practical purposes can be done in low constant time. Also, from a topological sort point of view, there is no pressure to trim logs, except pragmatic operational reasons outside of ScuttleSort.

Todo:

- additional synthetic mixed behavior case, with the typical Internet asymmetry: 32 active users, 512 lurchers, 256000 messages
- should repeat each run 10 times, then average (and mark percentiles)
- should validate with more than 1 cross-reference (although we are confident that ScuttleSort will still be correct)

5) Towards a Formal Correctness Proof for ScuttleSort

In terms of Conflict-Free Replicated Data Types (CRDTs), ScuttleSort produces a partial order-preserving add-only sequence (useful for a “timeline”) with strong eventual consistency - at least this is our hypothesis. Note that add-only means that events cannot be removed and that the position of an added element can be anywhere and can change over time, which differs from an append-only log.

Currently we do not have a proof that ScuttleSort is correct. The easy part of such a proof is the “for all delivery schedules” side where SSB’s reliable pub/sub service using append-only logs eliminates duplicates and retains partial order. The more challenging part are the “shape rules” in form of the `_rise()` procedure for which it must be shown that early decisions regarding the placement of events, which influence later decisions, are not able to diverge the outcome from the single possible result (=

strong eventual consistency requirement and the underlying Least Upper Bound property of the partial order). Intuitively, the "lowest energy level"-strategy in the `_rise()` function seems suitable for such an assessment. The internal machinery auf Automerge [KLE2017] could be a good starting point as well as the DSON paper [RIN2022] where "dots" correspond quite literally to log entries.

6) Conclusions and Future Work

We presented an incremental topological sort algorithm that converts the streams of incoming append-events into a single stream of update commands for an eventually consistent shared sequence.

An important asset will be the formal proof for the claimed convergence property of ScuttleSort (i.e., strong eventual consistency of the linearization). Time complexity is another domain where more insight is needed, especially as ScuttleSort uses sorting internally.

Space complexity is also of concern, this time from a systems perspective, because the linearization is maintained as an in-memory data structure which is not sustainable for open-ended stream processing. A next version of ScuttleSort should be designed that uses a database backend in order to cap the amount of required main memory (that otherwise scales linearly with the number of processed events).

Another memory bound concern is the "outgoing" array which keeps track of all (future) events that reference a given event. As this number can grow arbitrarily large (also as part of a denial-of-service attack), we suggest to limit the number of predecessors an event can have to two which leads to a fixed-memory-size overhead per event, as we showed in Section 2.2 . However, this restriction and feature has not been implemented yet.

Finally, as the number of generated update instructions per received event depends on the delivery properties of the pub/sub substrate and on the tangling style of the applications, it would be interesting to apply ScuttleSort to the body of existing messages in the Secure Scuttlebutt system in order to learn real-world numbers for update instructions per new event.

Bibliography

[KLE2017] Martin Kleppmann and Alastair R. Beresford: A Conflict-Free Replicated JSON Datatype, in IEEE Transactions on Parallel and Distributed Systems, vol. 28, no. 10, pp. 2733-2746, Oct 2017, doi: 10.1109/TPDS.2017.2697382 and https://arxiv.org/pdf/1608.03960.pdf

[PEA2006] David J. Pearce and Paul H. J. Kelly: A dynamic topological sort algorithm for directed acyclic graphs. ACM Journal of Experimental Algorithmics, Vol 11,2006, https://doi.org/10.1145/1187436.1210590

[RIN2022] Arik Rinberg, Tomer Solomon, Roee Shlomo, Guy Khazma, Gal Lushi, Idit Keidar, and Paula Ta-Shma. DSON: JSON CRDT Using Delta-Mutations For Document Stores. PVLDB 2022, https://www.vldb.org/pvldb/vol15/p1053-rinberg.pdf

[SIG2016] Ragnar Lárus Sigurðsson: Practical performance of incremental topological sorting and cycle detection algorithms, 2016 (MSc thesis Department of Computer Science and Engineering CHALMERS UNIVERSITY OF TECHNOLOGY UNIVERSITY OF GOTHENBURG Gothenburg, Sweden)

Appendix A: Python Source Code for ScuttleSort (0.0.6)

```
class Timeline:

    def __init__(self, update_callback=None):
        self.linear = []   # linearized DAG
        self.name2p = {}   # name ~ point_in_time
        self.pending = {}  # cause_name ~ [effect_name]
        self.notify = update_callback
        self.cmds = []

    def _insert(self, pos, h):
        self.linear.insert(pos, h)
        if self.notify:
            self.cmds.append( ('ins', h.name, pos) )

    def _move(self, old, new):
        h = self.linear[old]
        del self.linear[old]
        self.linear.insert(new, h)
        if self.notify:
            self.cmds.append( ('mov', old, new) )

    def add(self, nm, after=[]):
        self.cmds = []   # this is not reentrant: add a lock if necessary
        SCUTTLESORT_NODE(nm, self, after)
        # optimizer: compress the stream of update commands
        #            ins(X,nm), mov X Y, mov Y Z etc --> ins(Z,nm)
        #            mov X Y, mov Y Z etc            --> mov X Z
        if self.notify:
            base = None
            for c in self.cmds:
                if base != None:
                    if c[0] == 'mov' and base[2] == c[1]:
                        base[2] = c[2]
                        continue
                    self.notify( *base )
                base = list(c)
            if base != None:
                self.notify( *base )
```

```
class SCUTTLESORT_NODE: # push updates towards the future, "genesis" has rank 0

    def __init__(self, name, timeline, after=[]):
        if name in timeline.name2p: # can add a name only once, must be unique
            raise KeyError
        self.name = name
        self.prev = [x for x in after] # copy of the causes we depend on
                    # hack alert: these are str/bytes, will be replaced by nodes
        # internal fields of sorting algorithm:
        self.cycl = False   # cycle detection (could be removed for SSB)
```

```
        self.succ = []      # my future successors (="outgoing")
        self.vstd = False   # visited
        self.rank = 0       # 0 for a start, we will soon know better

        timeline.name2p[name] = self

        for i in range(len(self.prev)):
            c = self.prev[i]
            if not c in timeline.name2p:
                if not c in timeline.pending:
                    timeline.pending[c] = []
                if not self in timeline.pending[c]:
                    timeline.pending[c].append(self)
            else:
                p = timeline.name2p[c]
                p.succ.append(self)
                self.prev[i] = p # replace str/bytes by respective node

        pos = 0
        for p in self.prev:
            if type(p) != str and p.indx > pos:
                pos = p.indx
        for i in range(pos, len(timeline.linear)):
            timeline.linear[i].indx += 1
        self.indx = pos
        timeline._insert(pos, self)

        anchors = [x for x in self.prev if type(x) != str]
        if len(anchors) > 0:
            for p in anchors:
                self.add_edge_to_the_past(timeline, p)
        elif len(timeline.linear) > 1: # there was already at least one feed
            self._rise(timeline)       # insert us lexicographically at time:

        if self.name in timeline.pending:  # our node was pending
            for e in timeline.pending[self.name]:
                for c in [x for x in e.prev if x == self.name]:
                    e.add_edge_to_the_past(timeline, self)
                    self.succ.append(e)
                    e.prev[e.prev.index(c)] = self # replace str/bytes
            del timeline.pending[self.name]

        # FIXME: we should undo the changes in case of a cycle ...

    def add_edge_to_the_past(self, timeline, cause):
        # insert causality edge (self-to-cause) into topologically sorted graph
        visited = set()
        cause.cycl = True
        self._visit(cause.rank, visited)
        cause.cycl = False

        si = self.indx
```

```
        ci = cause.indx
        if si < ci:
            self._jump(timeline, ci)
        else:
            self._rise(timeline)

        for v in sorted(visited, key=lambda x: -x.indx):
            v._rise(timeline) # bubble up towards the future
            v.vstd = False

    def _visit(self, rnk, visited): # "affected" wave towards the future
        out = [[self]]
        while len(out) > 0:
            o = out[-1]
            if len(o) == 0:
                out.pop()
                continue
            c = o.pop()
            c.vstd = True
            visited.add(c)
            if c.cycl:
                raise Exception('cycle')
            if c.rank <= rnk + len(out) - 1:
                c.rank = rnk + len(out)
                out.append([x for x in c.succ])

    def _jump(self, timeline, pos):
        #              self.indx   pos
        #              v           v
        #  before .. | e | f | g | h | ... -> future
        #
        #  after  .. | f | g | h | e | ... -> future
        si = self.indx
        for i in range(si+1, pos+1):
            timeline.linear[i].indx -= 1
        timeline._move(si, pos)
        self.indx = pos

    def _rise(self, timeline):
        len1 = len(timeline.linear) - 1
        si = self.indx
        pos = si
        while pos < len1 and self.rank > timeline.linear[pos+1].rank:
            pos += 1
        while pos < len1 and self.rank == timeline.linear[pos+1].rank and \
                                          timeline.linear[pos+1].name < self.name:
            pos += 1
        if si < pos:
            self._jump(timeline, pos)

# end of ScuttleSort algorithm
```

Appendix B: JavaScript Source Code for ScuttleSort (0.0.7)

```
"use strict"

class Timeline {

    constructor(update_cb) {
        this.linear  = [];
        this.name2p  = {}; // name ~ point_in_time
        this.pending = {}; // cause_name ~ [effect_name]
        this.notify  = update_cb;
        this.cmds    = [];
    }

    _insert(pos, h) {
        this.linear.splice(pos, 0, h);
        if (this.notify)
            this.cmds.push( ['ins', h.name, pos] );
    }

    _move(from, to) {
        let h = this.linear[from];
        this.linear.splice(from, 1);
        this.linear.splice(to, 0, h);
        if (this.notify)
            this.cmds.push( ['mov', from, to] );
    }

    add(nm, after) {
        this.cmds = []; // this is not reentrant: add a lock if necessary
        let n = new ScuttleSortNode(nm, this, after);
        // optimizer: compress the stream of update commands
        //            ins(X,nm), mov X Y, mov Y Z etc --> ins(Z,nm)
        //            mov X Y, mov Y Z etc            --> mov X Z
        if (this.notify) {
            var base = null;
            for (let c of this.cmds) {
                if (base) {
                    if (c[0] == 'mov' && base[2] == c[1]) {
                        base[2] = c[2];
                        continue;
                    }
                    this.notify( base );
                }
                base = c;
            }
            if (base)
                this.notify( base );
        }
    }

}
```

```
class ScuttleSortNode {

    constructor(name, timeline, after) {
        if (name in timeline.name2p) // can add a name only once, must be unique
            throw new Error("KeyError");
        this.name = name;
        this.prev = after.map( x => { return x; } ); // copy of the causes we depend on
                  // hack alert: these are str/bytes, will be replaced by nodes
        // --- internal fields for insertion algorithm:
        this.cycl = false;  // cycle detection, could be removed for SSB
        this.succ = [];     // my future successors (="outgoing")
        this.vstd = false;  // visited
        this.rank = 0;      // 0 for a start, we will soon know better

        timeline.name2p[name] = this
        for (let i = 0; i < this.prev.length; i++) {
            let c = this.prev[i];
            let p = timeline.name2p[c]
            if (p) {
                p.succ.push(this);
                this.prev[i] = p; // replace str/bytes by respective node
            } else {
                if (!timeline.pending[c])
                    timeline.pending[c] = [];
                let a = timeline.pending[c];
                if (!a.includes(this))
                    a.splice(a.length, 0, this);
            }
        }

        var pos = 0;
        for (let i = 0; i < this.prev.length; i++) {
            let p = this.prev[i];
            if (typeof(p) != "string" && p.indx > pos)
                pos = p.indx;
        }
        for (let i = pos; i < timeline.linear.length; i++)
            timeline.linear[i].indx += 1;
        this.indx = pos;
        timeline._insert(pos, this);

        var no_anchor = true;
        for (let p of this.prev) {
            if (typeof(p) != "string") {
                this.add_edge_to_the_past(timeline, p);
                no_anchor = false;
            }
        }
        if (no_anchor && timeline.linear.length > 1) {
            // there was already at least one feed, hence
            // insert us lexicographically at time t=0
            this._rise(timeline);
```

```
    }

    let s = timeline.pending[this.name];
    if (s) {
        for (let e of s) {
            for (let i = 0; i < e.prev.length; i++) {
                if (e.prev[i] != this.name)
                    continue;
                e.add_edge_to_the_past(timeline, this);
                this.succ.push(e);
                e.prev[i] = this;
            }
        }
        delete timeline.pending[this.name];
    }

    // FIXME: should undo the changes in case of a cycle exception ...
}

add_edge_to_the_past(timeline, cause) {
    // insert causality edge (self-to-cause) into topologically sorted graph
    let visited = new Set();
    cause.cycl = true;
    this._visit(cause.rank, visited)
    cause.cycl = false;

    let si = this.indx;
    let ci = cause.indx;
    if (si < ci)
        this._jump(timeline, ci);
    else
        this._rise(timeline)

    let a = Array.from(visited);
    a.sort( (x,y) => { return y.indx - x.indx; });
    for (let v of a) {
        v._rise(timeline); // bubble up towards the future
        v.vstd = false;
    }
}

_visit(rnk, visited) { // "affected" wave towards the future
    let out = [[this]];
    while (out.length > 0) {
        let o = out[out.length - 1];
        if (o.length == 0) {
            out.pop();
            continue
        }
        let c = o.pop();
        c.vstd = true;
        visited.add(c);
```

```
            if (c.cycl)
                throw new Error('cycle');
            if (c.rank <= (rnk + out.length - 1)) {
                c.rank = rnk + out.length;
                out.push(Array.from(c.succ));
            }
        }
    }

    _jump(timeline, pos) {
        //              this.indx   pos
        //              v           v
        //  before .. | e | f | g | h | ... -> future
        //
        //  after  .. | f | g | h | e | ... -> future
        let si = this.indx
        for (let i = si+1; i < pos+1; i++)
            timeline.linear[i].indx -= 1;
        timeline._move(si, pos);
        this.indx = pos
    }

    _rise(timeline) {
        let len1 = timeline.linear.length - 1;
        let si = this.indx;
        var pos = si
        while (pos < len1 && this.rank > timeline.linear[pos+1].rank)
            pos += 1;
        while (pos < len1 && this.rank == timeline.linear[pos+1].rank
                          && timeline.linear[pos+1].name < this.name)
            pos += 1;
        if (si < pos)
            this._jump(timeline, pos);
    }
}

module.exports = Timeline
```
