Merkle Search Trees: Efficient State-Based CRDTs in Open Networks

Alex Auvolat, François Taïani

HAL Id: hal-02303490
https://inria.hal.science/hal-02303490

Submitted on 3 Oct 2019

Merkle Search Trees:
Efficient State-Based CRDTs in Open Networks

Alex Auvolat
École Normale Supérieure
Paris, France
alex.auvolat@inria.fr

François Taïani
Univ. Rennes, Inria, CNRS, IRISA
Rennes, France
francois.taiani@irisa.fr

Abstract—**Most recent CRDT techniques rely on a causal broadcast primitive to provide guarantees on the delivery of operation deltas. Such a primitive is unfortunately hard to implement efficiently in large open networks, whose membership is often difficult to track. As an alternative, we argue in this paper that pure state-based CRDTs can be efficiently implemented by encoding states as specialized Merkle trees, and that this approach is well suited to open networks where many nodes may join and leave. At the core of our contribution lies a new kind of Merkle tree, called Merkle Search Tree (MST), that implements a balanced search tree while maintaining key ordering. This latter property makes it particularly efficient in the case of updates on sets of sequential keys, a common occurrence in many applications. We use this new data structure to implement a distributed event store, and show its efficiency in very large systems with low rates of updates. In particular, we show that in some scenarios our approach is able to achieve both a 66% reduction of bandwidth cost over a vector-clock approach, as well as a 34% improvement in consistency level. We finally suggest other uses of our construction for distributed databases in open networks.**

Index Terms—**Merkle trees, search trees, CRDT, IoT, georeplicated systems, peer-to-peer, anti-entropy**

I. INTRODUCTION

The advent of massive geo-replicated systems [1], [2] has prompted a growing interest in data replication techniques that are both consistent and scalable. Among these, *Conflict-free Replicated Data Types*, or CRDTs [3], stand out by their ability to provide eventual consistency while offering a natural and modular programming paradigm to developers.

CRDTs allow replicas to perform operations concurrently and without any synchronization by providing resolution rules that allow concurrent operations to be combined a-posteriori. They can thus be used to build weakly-consistent systems with eventual consistency.

Recent CRDT techniques have focused primarily on leveraging causal broadcast primitives and vector clocks in order to ensure reliable causal delivery of operation deltas [4], [5], meaning that they never need to compare two full states directly. However these approaches show their limit in open networks where many nodes may join and leave, as causal broadcast primitives and vector clocks are both unsuited in this context: causal broadcast has only recently been extended in a scalable fashion to networks with dynamic structures [6] and the practicality of this approach has not yet been demonstrated,

whereas vector clocks require metadata that grows linearly with the number of nodes, past and present, that have participated in the network, which is an important scalability barrier even in the case of extensive optimisations [7].

In order to target open networks while avoiding these limitations, we focus in this paper on pure *state-based* CRDTs that do not require either causal broadcast or vector clocks, and to their fundamental issue: how to efficiently implement remote state merge for large state between nodes that might have had no previous interaction and might know nothing of one another's state. This problem is similar to *anti-entropy*, which was studied before the formalization of CRDTs. In the following, we will use the terms anti-entropy, state merge and state reconciliation as synonyms. We target more specifically CRDTs implementing sets or maps, two fundamental building blocks that can be combined with other CRDT semantics, and seek to resolve differences between two very large sets or maps with low latency and minimum bandwidth usage.

A common strategy for the reconciliation of sets and maps exploits Merkle trees [8], a hash-based data structure that can be used to rapidly identify set-differences [9]. Unfortunately, usual anti-entropy algorithms that use Merkle trees do not preserve the order of elements, making them particularly inefficient when updates are applied on sequences of close-by items. This is a very common scenario, a basic example being the case where elements are events ordered by timestamps.

In this paper we propose to overcome this fundamental challenge thanks to a novel data structure, which we call a *Merkle Search Tree*, that deterministically builds a balanced search tree from a set of items and encodes it as a Merkle tree. We demonstrate how Merkle Search Trees can be used to build a causally consistent event store that ensures eventual delivery of all past events to all connected nodes. We compare this approach to a vector clock-based approach [10] as well as to a Merkle tree construction that does not preserve order, and show that, in large networks and under low or moderate update rates, Merkle Search Trees provide the best trade-off: a 66% reduction of bandwidth usage was achieved in our simulation when compared to the vector-clock approach, as well as a 34% improvement in our consistency measure and a 32% improvement in 99th percentile delivery delay. When compared to Merkle trees without order, Merkle Search Trees were also better on all three metrics.

II. Context and Related Work

Our work draws from several well-studied domains in distributed systems as well as databases and file systems, which we present in this section.

A. CRDTs and the CAP Theorem

CRDTs [3] are a generic framework in which eventually consistent algorithms can be formulated. The CAP theorem [11], [12] states that a distributed system may only achieve two of the following three properties: strong consistency, availability and partition tolerance. CRDT-based systems typically forgo strong consistency in favor of availability, by allowing replicas to diverge temporarily. The particular CRDT used by a system defines a way in which replicas in divergent states can reconcile automatically to a unique shared state as soon as they are able to communicate reliably.

In this paper we use the formulation of CRDTs as a join-semilattice on replica states, that is a CRDT is a set of possible states $\mathbb{V}$ with a symmetric join operation $\sqcup$. Operations on a state $x \in \mathbb{V}$ consist in changing x into a state x' that is strictly superior according to $\sqcup$, i.e. such that there is an item $o \in \mathbb{V}$ such that $x' = x \sqcup o$. For instance if the data type we want to implement is a grow-only set, $\mathbb{V}$ is the space of possible sets and $\sqcup$ is the set union operator. Adding an element e to a set x consists in changing x into the state $x' = x \cup \{e\}$.

The operator $\sqcup$ is also used to combine concurrent operations done at different nodes and resolve them deterministically: if a node is in state x_1 and another is in state x_2, both will resolve to state $x_1 \sqcup x_2$ (which is the same as $x_2 \sqcup x_1$). For instance, for a grow-only set CRDT, the merge operation is defined as set union, such that the resulting state is defined as the set containing all items that have been added at all nodes. More complex CRDTs can be defined to implement additional operations such as deletion or different data types such as maps or counters, the only requirements being to implement a commutative, associative and idempotent join operator $\sqcup$.

In this work we focus on set and map CRDTs and how to implement efficient computation of $x_1 \sqcup x_2$ when x_1 and x_2 are located on two different nodes, where efficiency is measured in terms of network round-trips and bandwidth usage.

B. Gossip Algorithms

Anti-entropy protocols typically adopt an epidemic reconciliation strategy in which the nodes of a distributed system repeatedly reconcile their differences with randomly selected other nodes in order to converge to an agreed-upon state [13]. We adopt this strategy to implement CRDTs, and combine a gossip algorithm with the CRDT state merge operator to ensure eventual delivery of updates at all nodes. Gossip algorithms [13] are efficient schemes for data dissemination in large scale distributed systems, and are well adapted in open networks when used in association with random peer sampling [14]. Gossip algorithms have well studied dissemination properties [15] and are widely used as a basis for distributed computing [16].

C. Hash Functions and Merkle Trees

Hash functions and the checksums they produce are commonly used to identify data and check its integrity and/or authenticity. They therefore might appear as a natural choice to compare data residing on different computers, and help implement an efficient distributed reconciliation mechanism. Unfortunately, to check a very large piece of data, the hash function must be calculated on the whole data piece, a particularly costly proposition when dealing with large distributed objects, such as a typical key-value data store. To reduce this cost, the data can be chunked and each chunk hashed independently, but then a single hash is replaced by a list of hashes that grows linearly with the data size, which remains problematic.

To overcome this linear cost, Merkle introduced [8] a method that is able to hash large pieces of data by chunking the data and computing a tree of hashes, where leaves are hashes of data chunks and nodes are hashes of the concatenations of their children's values. The single root hash is sufficient to identify all the data, and the validity of a single block can be checked by walking through the path to the corresponding leaf, thus verifying only a logarithmic number of hashes provided that the tree is well balanced. This verification can also occur without having access to the whole data.

The original Merkle trees are binary trees, however the core principle of Merkle trees can be generalized and used in other contexts. Most notably, projects such as ZFS [17] and IPFS [18] make use of recursive hashing in the fashion of Merkle trees in order to guarantee authenticity and integrity of whole file systems. Such recursive hashing of data structures also allows for data de-duplication [17], [18], in which case the Merkle tree becomes a DAG where nodes may have several parents.

In distributed systems, Merkle trees have shown to be particularly useful: given a deterministic way to encode data sets in Merkle trees, two nodes can determine whether they have the same version of the data simply by exchanging and comparing their root hashes. If their versions differ, they are able to identify which branches of the tree contain changes, as branches that represent the same contents have the same hash on both nodes, therefore allowing them to exchange only the differing parts of the data set.

Merkle trees come in different shapes. The standard binary Merkle trees can be used to identify and exchange numbered sequences of data blocks, but they are not able to detect differences efficiently on arbitrary key spaces when the sets of keys present at two nodes might not be the same. Database systems such as Dynamo [9] use Merkle trees to exchange and repair states that consist of arbitrary key-value mappings, however to ensure that a tree is well balanced they must destroy key ordering and project the keys into a uniform distribution by hashing them, which makes this method sub-optimal in scenarios where sequences of close-by keys are updated. A Merkle trees construction that preserve key order while staying balanced on arbitrary key ranges was introduced in [19],

however this method is not deterministic: a same dataset may have several Merkle tree representations, which makes it unfit for anti-entropy. A first deterministic balanced Merkle tree construction that preserves key order on arbitrary key ranges was introduced using binary treaps [20]. The construction we propose achieves the same properties with wider tree nodes, thus making for shallower trees and reducing the number of round-trips required for remote difference computation. This also reduces the number of hashes that need to be computed and stored compared to binary Merkle trees.

D. B-trees

The Merkle Search Tree construction we propose uses a structure similar to B-trees [21], albeit with some key differences. B-trees are indexing data structures used in relational database management systems (RDBMS). They implement shallow search trees by having nodes with large out-degrees, allowing faster traversal than standard binary search trees. Nodes typically have out-degrees such that the machine representation of a tree node is the size of a memory page. In a B-tree, a node is composed of a set of values that define how the space of keys is split, and of a set of pointers to the sub-trees containing the values for each interval in the split. The Merkle Search Trees we introduce also contain intermediate nodes whose values serve to split the key space for their children. Their nodes, however, do not have a constant size and their size is bounded only probabilistically.

E. Other State-of-the-art Methods and Their Limitations

Other methods for remote dataset comparison, which are not based on Merkle trees, have been proposed. In particular, mathematical methods for calculating set differences have been well studied, however they are either limited to approximate difference detection using Bloom filters [22], [23] or have crippling requirements such as requiring expensive computations to be done on the whole dataset [24]. In practice, typical anti-entropy protocols use vector clocks [25] to identify missing updates between two nodes [10], however even with extensive optimisations such as those used by DottedDB [7], vector clocks are not able to identify divergences between two versions of the dataset without requiring metadata linear in the total number of participants, past and present.

III. THE MERKLE SEARCH TREE DATA STRUCTURE

To implement an efficient reconciliation procedure for CRDTs implementing large maps and sets, we propose a tree construction, which we call a *Merkle Search Tree* (abbreviated MST), that is to our knowledge the first to combine the following three features:

- a given set of items has a unique deterministic representation as a tree;
- key order is preserved;
- trees are always balanced (probabilistically).

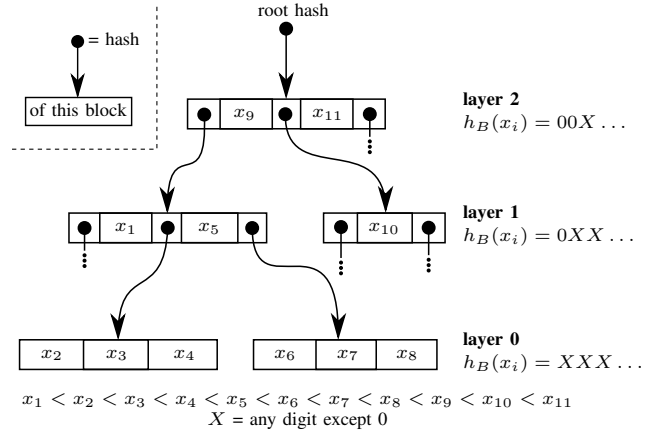The combination of these properties makes remote tree comparison for anti-entropy extremely efficient.

$x_1 < x_2 < x_3 < x_4 < x_5 < x_6 < x_7 < x_8 < x_9 < x_{10} < x_{11}$
X = any digit except 0

Fig. 1: Structure of a Merkle Search Tree

A. MST Construction

Merkle Search Trees implement an ordered set with elements taken in a totally ordered space $\mathbb{K}$. In order to implement maps, elements can be accompanied by a tag from a set $\mathbb{V}$ (the *values*, in which case $\mathbb{K}$ becomes the set of *keys*). $\mathbb{V}$ contains a default element $\bot$ ('bottom'), used to indicate that a key is absent from the map $f : \mathbb{K} \rightarrow \mathbb{V}$.

The MST construction is based on a hash function over arbitrary byte strings that is assumed to be collision-resistant: we assume the probability of finding two strings that have the same hash to be negligible. This is a common assumption which is essential to the Merkle tree construction. We also assume that the hash functions projects uniformly to values of a certain interval whose size is a power of B. This additional property is required in our construction as we also use the hash function for the secondary purpose of providing a deterministic source of randomness over the space $\mathbb{K}$. Both of these properties are achieved in practice by modern hash functions such as SHA-512.

An MST is a search tree, similar to a B-trees in the sense that internal tree nodes contain several values that define a partition of $\mathbb{K}$ in which the children values are separated. The tree is divided in layers which are numbered starting at layer 0 which corresponds to the layer of leaf nodes (Figure 1). The tree nodes in a layer l are blocks of consecutive items whose boundaries corresponds to items of layers $l' > l$. For instance, in Figure 1, the first nodes shown for layer 0 stored the values $\{x_2, x_3, x_4\}$ which are contained between the boundaries x_1 and x_5, with x_1 and x_5 stored in the layer 1. Similarly to a previous construction [20], deterministic randomness obtained by hashing the values is used to determine tree shape. Values stored in the MST are assigned a layer by computing their hash and writing that value in base B, which we will refer to as $h_B(x)$. The layer $l(x)$ to which an item x is assigned is the layer whose number is the length of the longest prefix of $h_B(x)$ constituted of only zeros.

B. MST Unicity and Operations

The construction we have just described is deterministic: it produces a single possible representation as an MST for a given set of items, an essential property to implement a reconciliation procedure. The implementation of standard operations follows from this definition.

When using MSTs as maps, we can efficiently implement read operations $get(f, k)$ for single items and $getrange(f, [k_0, k_1))$ for ranges of consecutive items in a tree f, as well as write operations $put(f, k, v)$ and $delete(f, k)$. In the case where a put operation inserts an item x at a non-leaf layer $l > 0$, some of the tree nodes at lower levels $l' < l$ might need to be split in two at boundary x. Similarly when values are deleted at non-leaf layers, some nodes at lower layers might need to be merged. At most one split or merge can happen at each layer $l' < l$ for a given put or delete operation, therefore the complexity of these operations is proportional to the depth of the tree.

As a consequence of structural unicity, inserting the same elements in different orders will always converge to the same representation, and the Merkle hash of the root will be the same. Structural unicity is also valid at intermediary nodes: two intermediary nodes have the same Merkle hash if and only if the sub-trees starting at these nodes contain exactly the same items. Therefore when comparing two trees in order to find differences, we can skip whole sub-trees when they are found to have the same Merkle hash as we know that there are no differences between them.

C. Structural Properties: Balance and Depth

Since we use a hashing function that projects uniformly to strings of constant length that write out numbers in base B, the probability of being in layer l for an item is $p_l = \left(\frac{1}{B}\right)^l \frac{B-1}{B}$. It is easy to see that the average number of values at layer l is B times the average number of values at layer $l + 1$. Since values of layer l are split at boundaries that are all values at layers $l' > l$, nodes of a layer l have on average $B - 1$ values stored and, at non-leaf layers, B children. The probability of a node having many more values and children than these average values decreases exponentially, therefore we can bound the node size with arbitrarily high probability.

The depth of the tree is the maximum level that an item may be assigned to. The probability of having an item at layer l is bounded by np_l (where n is the number of elements in the tree), which decreases exponentially with l. Therefore we can bound the tree depth as $log_B n$ plus a constant c with a probability that decreases exponentially in c.

Since the tree depth is about $log_B n$ and nodes have a constant number of items with high probability, we conclude that the tree has a well balanced structure and therefore reads, puts and deletes can be implemented in $O(log_B n)$ complexity.

D. Efficiency for Dataset Comparison

The comparison of two Merkle Search Trees stored at two different nodes is efficient: if we are comparing f which is on node n_1 with g which is on node n_2, then nodes n_1 and n_2 need exchange only $O(d \log_B n)$ messages, with $d = |\{x | f(x) \neq g(x)\}|$, and $n = \max(|f|, |g|)$. Supposing elements of $\mathbb{K}$ and $\mathbb{V}$ are of constant size, exchanged message are of average size proportional to B.

The average out-degree of inner tree nodes can be controlled by changing the value of B. Using larger values of B makes for nodes that contain more data, thus requiring more bandwidth expenditure when exchanged between peers, but makes for a shallower tree and therefore requires less network round trips to obtain the full set of nodes leading to a leaf. In our experiments, we fix $B = 16$.

E. Merkle Search Trees as CRDTs

If $\mathbb{V}$ is a CRDT with a merge operation $\sqcup_\mathbb{V}$ such that $\forall x, \bot \sqcup_\mathbb{V} x = x \sqcup_\mathbb{V} \bot = x$, then Merkle Search Trees implement a CRDT on $\mathbb{K} \rightarrow \mathbb{V}$ defined by the point-to-point application of $\sqcup_\mathbb{V}$: $(f \sqcup g)(x) = f(x) \sqcup_\mathbb{V} g(x)$.

The sequence of states a CRDT takes is supposed to be monotonic with respect to $\sqcup$, as explained in Section II-A: a transition from a state x to a state x' must be of the form $x' = x \sqcup o$. In the case of Merkle Search Trees, this means restricting the set of allowed operations to those that lead to monotonic sequences for each individual key. Therefore operations put and $delete$ must not be used directly, but rather updates must be of the form $update(f, k, v) = put(f, k, get(f, k) \sqcup v)$.

By selecting $\mathbb{V}$ appropriately, we can obtain various CRDTs. For instance if $\mathbb{V} = \{\bot, \top\}$ is a boolean indicating if an item is present or not, we obtain a grow-only set defined on $\mathbb{K}$. If $\mathbb{V}$ is a last-writer-wins register with a version number then we get a key-value store with last-writer-wins reconciliation. Any existing CRDT type can be used as the value type $\mathbb{V}$, yielding a map CRDT construction with efficient detection of differing items.

IV. CRDTs IN LARGE-SCALE OPEN NETWORKS

The Merkle Search Trees we have presented can be used to implement a particularly efficient pair-wise distributed reconciliation procedure for set or maps CRDTs. This procedure allows two nodes to converge to a unique state on the basis of a voluntary exchange that does not require any node to have prior knowledge of the other node's state. We combine this procedure with a simple gossip-based algorithm to allow for efficient dissemination of updates in open networks with high churn rates.

A. Gossip-based Reconciliation for State-based CRDTs

Our approach builds on a simple distributed state-update method for state-based CRDTs shown in Algorithm 1 in its basic form, without the use of MSTs. Algorithm 1 adopts a reactive gossip strategy: when a node does an operation that results in a state transition from x_0 to x_1, where $x_1 = x_0 \sqcup o$, it gossips x_1 to some random peers. When a node receives an incoming state x_1 when it was in state x'_0, it does the transition to $x'_0 \sqcup x_1 = x'_1$ and in the case where $x'_1 \neq x'_0$ it in turn gossips x'_1 to some random peers. The selection of random peers assumes a peer sampling service [14] that is

---

ALGORITHM 1: Basic State-Based CRDT

```
1  initialization
2  |    state ← ⊥
3  |    fanout ← algorithm parameter
4  function do(op)
5  |    state' ← state ⊔ op
6  |    if state' ≠ state then
7  |    |    Gossip state' to fanout random peers
8  |    |    state ← state'
9  on receive Gossip new_state
10 |    state' ← state ⊔ new_state
11 |    if state' ≠ state then
12 |    |    Gossip state' to fanout random peers
13 |    |    state ← state'
```

ALGORITHM 2: State CRDT w/ Merkle Search Trees

```
1  initialization
2  |    fanout ← algorithm parameter
3  |    max_merges ← algorithm parameter
4  |    state ← ⊥; merges ← ∅
5  function do(op)
6  |    handle_update(op)
7  on receive Gossip root_hash from peer
8  |    if root_hash ≠ h(state) ∧ |merges| ≤ max_merges then
9  |    |    handle_merge(root_hash, peer)
10 function handle_merge(root_hash, peer)
11 |    Q ← missing_blocks(root_hash, local_store)
12 |    if Q ≠ ∅ then
13 |    |    Request Q to peer
14 |    |    merges ← merges ∪ {(root_hash, peer)}
15 |    else
16 |    |    update ← load(root_hash, local_store)
17 |    |    handle_update(update)
18 |    |    merges ← merges \ {(root_hash, peer)}
19 on receive Request Q from peer
20 |    R ← {block ∈ local_store|h(block) ∈ Q}
21 |    Reply R to peer
22 on receive Reply R from peer
23 |    if ∃(root_hash, p) ∈ merges|p = peer then
24 |    |    save(R, local_store)
25 |    |    handle_merge(root_hash, peer)
26 function missing_blocks(root_hash, local_store)
27 |    (omitted for brievty; returns the hashes of the blocks of the tree
        identified by root_hash that are missing from local_store)
28 function handle_update(update)
29 |    state' ← state ⊔ update
30 |    if state' ≠ state then
31 |    |    Gossip h(state') to fanout random peers
32 |    |    state ← state'
```

---

able to give the identity of some nodes of the network taken at random.

Algorithm 1 implements reactive push-only exchanges. Other methods with pull or push/pull strategies are also possible, and can also be extended with a periodic component to realize a traditional anti-entropy protocol.

B. Using Merkle Search Trees to implement Large CRDTs

When using Algorithm 1 to implement large map-like CRDT types such as sets or key-value stores, a naive implementation that exchanges whole states rapidly becomes intractable. In such cases, MSTs can be used to reduce

dramatically network bandwidth usage at the cost of several round-trips between nodes. Algorithm 2 shows how this can easily be achieved in the background, without blocking local operations: the current state is not changed until all the remote information necessary for the merge is obtained (line 12), and get or update operations can continue to be executed on this state. In this algorithm, a set of remote MSTs that we wish to merge is kept in a buffer (lines 4, 14, 18) and all missing blocks are requested to the remote peer (line 13). Once all the blocks are locally available, the merge operation is done without network communication (line 16) and the local MST is updated.

C. Causal Consistency

Algorithms 1 and 2 implement causal consistency in the following sense: if a node does an operation that results in a transition from x to $x' = x \sqcup o$ after having read from the state x, then the causal past of the put operation can be described as being included in x, and every other node that observes the put will observe its causal past x (since x' contains x), or a successor of x according to $\sqcup$. This algorithm and resulting property is applicable to all CRDTs that are defined by a merge $\sqcup$ and Merkle Search Trees extend this naturally to maps of CRDTs. This is done without any conditions on the network topology as is required by causal broadcast [6], [26]: it results from the fact of doing a full state merge at every gossip event.

If causal consistency is not required and eventual consistency is sufficient, Algorithm 2 can be optimized by gossiping individual single operations and applying them as soon as they are received, while executing periodic merges of the full data structure in parallel. In that case, periodic merges acts as a termination mechanism for ensuring eventual consistency. The use of explicit merges allows us to reduce the gossip fanout or time-to-live for single operations in order to save bandwidth, to values that would result in e.g. only 99% network coverage, and count on the full merge operation as an anti-entropy procedure for the small proportion of nodes that will miss a gossip event.

D. A Note on Crashes

Algorithm 2 uses a limit on the number of simultaneous merge operations that can be happening in order to preserve bandwidth (line 8). Therefore it may get stuck if it is waiting for a reply from a node that has crashed. Removing the limit on simultaneous merges is not desirable in practice as it might spread bandwidth usage too thinly across many concurrent pairwise merges and lead to high tail latencies. Therefore a fault detector or an approximate fault detector must be employed to cancel merges when a node has crashed. Canceling a merge when a node has not crashed but is just slow is not a problem as no safety property is violated. However in order for termination (eventual consistency) to be achieved, nodes must be able to complete at least some merge operations.

E. Extension to Distributed MSTs

Merkle Search Trees are a persistent data structure in the sense of functional programming. This means that when doing

a $\text{put}(k, v)$, the obtained MST needs not replace the previous MST but both can remain available simultaneously at a low storage cost by sharing memory for common blocks. Hashes in this case play a role similar to that of traditional pointers, with the key differences that hash values can now be used for automatic de-duplication of subtrees that contain the same values, and remain meaningful outside of the nodes on which they have been computed. Although in this paper we use MSTs as a local data-structure that is stored on a single node, the above features could be exploited to implement *Distributed MSTs*.

1) Storage Location: A distributed MST (or D-MST) is an MST where the data structure is split over many nodes in a network, for instance using a DHT. This is easy to do with any Merkle DAG data structure, as introduced by IPFS [18], and has already been done for binary search trees [19]. In our case, tree nodes correspond to blocks that are stored independently in the DHT and identified by their hash. If no caching is used, a D-MST requires $O(\log_B n \times \log m)$ network round trips for all operations, where m is the number of nodes in the network. Caching can reduce this to $O(\log_B n)$ for get operations in favorable conditions (no churn) [19].

2) Garbage Collection: Garbage collection in the D-MST scenario is harder than in the local MST case. An option is to expire all stored blocks after a certain time, and to have a distributed keep-alive mechanism that traverses the data structure resetting the expiry counter. Details of such a system are outside the scope of this paper.

V. APPLICATION: A CAUSALLY-CONSISTENT DISTRIBUTED EVENT STORE

We illustrate the benefits of CRDTs built with Algorithm 2 on the example of a *Causally-consistent Distributed Event Store*, a data-structure that encapsulates the notion of causal broadcast. Causal broadcast is a fundamental primitive of distributed algorithms that is usually directly implemented using *send/receive* network primitives and vector clocks [5] which are the main alternative to our method and which we show do not scale efficiently. For completeness, let us note that alternative approaches to Causal Broadcast exist for static overlays [26], which have been recently extended to a dynamic setting [6], but the practicality of this latest development has not yet been demonstrated.

Causal Broadcast is often one of the building blocks of operation-based CRDTs, such as δ-CRDTs [4]. In this section we reverse this usual perspective: instead of deploying a message-passing algorithm to ensure the delivery properties of causal broadcast, to then implement a distributed object, we provide causal broadcast by building a storage of all the events produced by the system, which we call an *event store*. We leverage Merkle Search Trees to provide an efficient way to propagate missing updates in an ad-hoc manner between two peers that have no a-priori knowledge of each other's states. A direct application of such a system would allow for example to build a peer-to-peer chat room or a log system but it may also be used as a primitive for more sophisticated algorithms.

In the following we study the theoretical complexity of such an approach, and compare it to *Scuttlebutt*, an anti-entropy algorithm introduced in [10] which uses vector clocks. We confirm this analysis experimentally in the next section. Both methods employ a push-pull gossip strategy to ensure update propagation on a large network. Our theoretical analysis is synthesized in Table II.

A. Scuttlebutt anti-entropy

1) Algorithm: An event in the log is identified by pair constituted of a producer node identifier and a sequence number generated locally by the producer. Scuttlebutt uses a reactive pull-push design: new events propagate as nodes exchange vectors of the last sequence number they have for each producer node. This exchange allows nodes to compute an exact set of missing events to send to the other node.

2) Diffusion time: An anti-entropy round is completed in one round-trip-time. The gossip protocol requires on average $\log m$ steps to reach the whole network, where m is the number of nodes currently present in the network, therefore the total diffusion time is $2\lambda \log m$ with 2λ the round-trip-time.

3) Bandwidth use: The bandwidth use for the first step is $O(p)$, where p is the number of producer nodes for which we have to send a sequence number. p is therefore equal to the count of all the nodes that have ever participated in the network, because we have no way of knowing which nodes are still present or not. For the second step, only the d missing events need to be sent so the bandwidth used is $O(d)$. Therefore the total bandwidth use is $O(p + d)$.

B. Merkle Search Tree anti-entropy

1) Algorithm: Merkle search trees are used as a simple grow-only set, with $\mathbb{V} = \{\bot, \top\}$ a boolean that is set to $\top$ for present items, and $\mathbb{K}$ is the space of events, ordered by their timestamp of production (either a logical clock or an approximation of real time). As a consequence, the events produced the most recently will be located at one end of the tree, increasing access locality. The gossip algorithm is the one described in Algorithm 2.

2) Diffusion time: An anti-entropy round is completed in one round-trip-time if no changes are present. Otherwise, for each new item the algorithm may need to visit a leaf of the tree. The tree is of depth $\log_B n$ (plus a small constant) with high probability, where n is the number of events stored in the tree, and the algorithm requires one round-trip for each tree level, therefore an anti-entropy round is completed in $\log_B n$ round trip time. The gossip protocol requires on average $\log m$ steps to reach the whole network, therefore the total diffusion time is $2\lambda \log m \log_B n$.

3) Bandwidth use: The total number of Merkle tree nodes that are requested by the first node and sent back by the second node is bounded by $d \log_B n$, therefore the total bandwidth use of one anti-entropy round is $O(d \log_B n)$.

VI. EXPERIMENTAL EVALUATION

In this section, we describe our experiment in simulating a large distributed event store using Merkle Search Trees

TABLE I: Notation

n: number of past events
d: number of new events in anti-entropy round
p: number of nodes, past and present
m: number of nodes currently connected
λ: network latency

TABLE II: Theoretical Comparison of Methods

Anti-entropy Algorithm	Dissemination Time	Traffic per anti-entropy round
Scuttlebutt	$2\lambda \log m$	$O(p + d)$
Merkle Search Trees (ours)	$2\lambda \log m \log_B n$	$O(d \log_B n)$

and compare it to other existing approaches. Our simulator, available online [1], uses an actor-based design and consists in about 2000 lines of Elixir code including all Merkle Search Tree algorithms.

A. Methodology

We simulate a network of 1000 nodes with synchronous communication rounds and no crashes. Events are generated with increasing timestamps on nodes chosen at random in the network. We consider two scenarios: in the first scenario, events are generated with an average rate of 0.1 events per round in the whole network, while they are generated at an average rate of 1 event per round in the second.

We compare our algorithm (Section V-B) with two competitors: the Scuttlebutt reconciliation algorithm described in Section V-A, and a second construction based on Merkle trees, a Merkle-encoded prefix tree on the hashes of the items (called *Merkle Prefix Tree*, or MPT for short).

The MPT competitor finds missing elements in the same manner as with Merkle Search Trees, and uses the same gossip-based algorithm. An MPT is built by hashing the key values and then building a prefix tree, thus the main difference with Merkle Search Trees is that this construction does not preserve key ordering. However this construction does build a tree that is on average well balanced, since the probability of hash prefix collisions decrease exponentially with the prefix length.

The experiments are run several times with different algorithm parameters, as shown in Table III.

Our aim is to show that when the modified values have similar keys, the ordering property of Merkle Search Trees improves the performance of the anti-entropy algorithm because fewer intermediate tree nodes will need to be transmitted: in the case of the distributed event store, the new events are those with the highest timestamps and thus form a compact subtree at the right of the tree, whereas if a prefix tree on the hashes is used, the events are randomly disseminated and the paths to the newly added leaves share only few intermediate nodes.

B. A Metric for Event Dissemination

In order to evaluate the inconsistencies in message spreading in the network, we define a metric based on the definition of binary entropy: at a certain simulation time step, the entropy of the network is defined as the sum for each message of the binary entropy of the fraction of nodes that have received the message. More formally, if M is the set of messages and

[1] https://gitlab.inria.fr/aauvolat/mst_exp/

TABLE III: Simulation Configuration

Common Parameters	
Number of nodes	1000
Events generated per round[a]	0.1 (Table IV, Figures 2, 4)
	1 (Table V, Figure 3)
Scuttlebutt Anti-Entropy	
Fanout	$1^\dagger$, 2^*, 4
Gossip interval	1^*, 2, $4^\dagger$, 8 rounds
MST and MPT Anti-Entropy	
Fanout	6
Max simultaneous merges	1, 2, 3, $4^{*\dagger}$, 6, 8

[a]in the whole network

TABLE IV: Simulation results on scenario with light load, for representative configurations of the simulated algorithms (indicated by * in Table III). Lower entropy values indicate more uniform diffusion of events and therefore better consistency. Lower bandwidth usage is better. Trade-off for different configurations is presented in Figure 2.

Method	Bandwidth use[a]	Entropy[b]	99% delivery delay
Scuttlebutt	1.3 Mo	1.61	64 rounds
MPT	0.51 Mo	1.44	56 rounds
MST (ours)	**0.44 Mo**	**1.06**	**44 rounds**
Gain vs. SB	-66%	-34%	-31%
Gain vs. MPT	-13%	-26%	-21%

[a]per round, on average
[b]at each round, on average

p_m the proportion of nodes in the network to have received a message $m \in M$, the entropy measure we use is defined by:

$$
\begin{aligned}
H &= \sum_{m \in M} H_b(p_m) \\
&= \sum_{m \in M} -p_m \log_2 p_m - (1 - p_m) \log_2 (1 - p_m)
\end{aligned}
\tag{1}
$$

With this measure, lower entropy values indicate more uniform diffusion of events and therefore better consistency of the overall network state.

C. Results

1) Entropy-Bandwidth Trade-Off: The first scenario, with low rate of events, is the most favorable scenario for the Merkle Search Tree approach. Figure 2 shows the curve obtained by plotting the bandwidth usage and entropy for different parameters of the algorithms (see Table III) on this scenario. Numerical results corresponding to the most successful configuration of each method are reported in Table IV. The lowest bandwidth usage is achieved by the two Merkle tree reconciliation methods, both of which are much more efficient than Scuttlebutt reconciliation without providing worse entropy measures or sacrificing message delivery delays. Figure 4 plots the bandwidth usage over time of the different methods. Merkle Search Trees also show a slight advantage compared to Merkle Prefix Trees on all evaluated metrics.
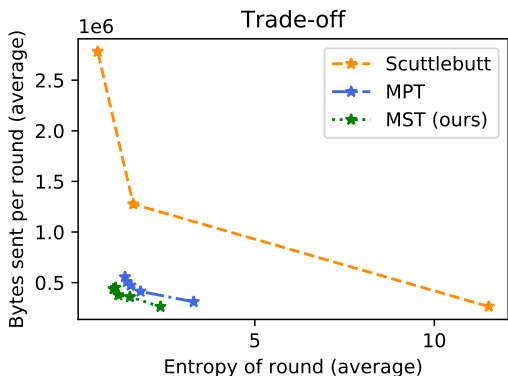
Fig. 2: Consistency vs Bandwidth usage compromise. x-axis: average entropy measure of the diffusion of events in the network (lower is better). y-axis: bandwidth usage (lower is better). MST: Merkle Search tree. MPT: Merkle Prefix Trees on hashes of items. SB: Scuttlebutt anti-entropy.
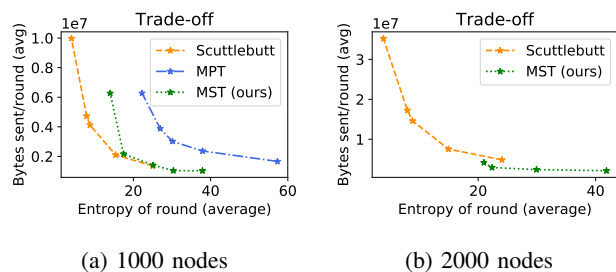

(a) 1000 nodes

(b) 2000 nodes

Fig. 3: Results with a more intense workload. Merkle Search Trees are still competitive and allow for a high-entropy low-bandwidth configuration. The limits of Merkle Prefix Trees on hashes of items become apparent in this configuration. By comparing a 1000 node scenario with a 2000 node scenario we also show that Merkle Search Trees scale better than Scuttlebutt with the number of nodes.
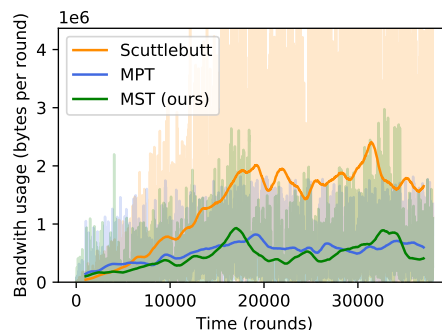
TABLE V: Simulation results on scenario with heavy load with 1000 and 2000 nodes, for representative configurations of the simulated algorithms (indicated by † in Table III). Trade-off for different configurations is presented in Figure 3.

1000 nodes			
Method	**Bandwidth use**	**Entropy**	**99% delivery delay**
Scuttlebutt	**2.1 Mo**	**15.4**	**50 rounds**
MPT	3.9 Mo	26.9	100 rounds
MST (ours)	2.2 Mo	17.5	74 rounds

2000 nodes			
Scuttlebutt	7.6 Mo	**14.9**	**54 rounds**
MPT	-[a]	-	-
MST (ours)	**4.2 Mo**	21.0	88 rounds

[a]Experiment did not terminate

Fig. 4: Bandwidth usage over time for experiments shown in Table IV.

The second scenario, with ten times higher event rate, is more ambivalent: both approaches perform equally well with 1000 nodes (Figure 3a). We increase the number of nodes to 2000 and show that the Merkle Search Tree approach scales better than the Scuttlebutt approach (Figure 3b and Table V). This scenario also clearly demonstrates the advantage of Merkle Search Trees over the Merkle Prefix Tree order: the latter show worst result than both methods for 1000 nodes, and the experiment on 2000 nodes did not terminate for Merkle Prefix Trees due to an explosion of the number of messages in the network, thus we were not able to complete the simulation.

2) Message Delivery Delay: We measure the delivery delay in rounds of events in the network. We study the 99th percentile worst case scenario. We show that in the light load case, Merkle Search Trees provide an advantage over Scuttlebutt (Table IV), and in the higher load scenario our method provides an acceptable degradation of about 50% (Table V).

3) When Are We Better?: We plot in Figure 5 the theoretical boundary that separates conditions for which Merkle Search Trees are better suited than the Scuttlebutt method. This boundary was used by writing an equivalence at the boundary

based on the theoretical results of Table II and replacing the number of differences d in an anti-entropy round by the rate of events in the network r:

$$\alpha p + \beta r = r \log_B n$$

where p is the number of processes in the simulation. We consider $\log_B n$ to be a constant (equal to $\log_B N$ where N is the total number of events produced in our simulation). After reordering terms, we obtain an affine boundary:

$$r = ap + b$$

which we calibrate using two simulations where both methods seemed to perform as well, one of which is shown in Figure 3(a) with $p = 1000, r = 1$ and the other is not shown and yields $p = 500, r = .33$. The plot of Figure 5 also shows the experimental configurations for which we ran a complete set of simulations, and the zones in which each method have an advantage are clearly identified.

4) Overall Evaluation: As expected, Merkle Search Trees are efficient when the rate of events is moderate to low: in this situation, the overhead of the Scuttlebutt method becomes prohibitive.
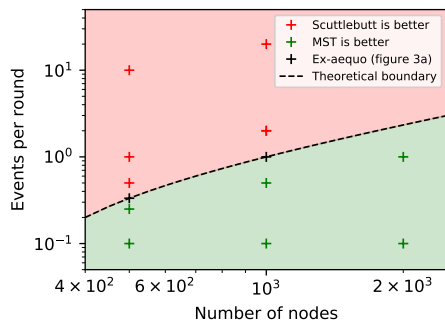
Fig. 5: Compromise between number of nodes and event rate in the network. Our method is better in regards to bandwidth consumption in the green area, Scuttlebutt anti-entropy is better in the red area. Crosses correspond to experiments we ran. In red, Scuttlebutt showed a clear advantage. In green, Merkle Search Trees showed a clear advantage.

In the light load scenario, Merkle Search Trees outperform all the other approaches in the three studied metrics: entropy, bandwidth usage and 99th percentile delivery delay. In the higher load scenario, Merkle search trees are not able to provide as low entropy as the Scuttlebutt protocol but they are able to extend the trade-off started with the different Scuttlebutt configurations in the direction of lower bandwidth usage and higher entropy. Merkle Search Trees scale better with the number of nodes present in the network, as the communication costs only depends of the rate of messages and not of the number of participating nodes.

We also show that Merkle Search Trees clearly outperform standard prefix trees in all possible scenarios and on both entropy and bandwidth usage, confirming our prediction that randomized prefix trees are not optimal for this application.

While we have only studied configuration where no nodes join or leave, we conjecture that the overhead of the Scuttlebutt method becomes even worse in open networks as the vector clocks will accumulate values for inactive nodes, leading to useless metadata being exchanged at every anti-entropy round.

VII. CONCLUSION AND OUTLOOK

In this paper, we have introduced a new data structure based on Merkle trees that allows for efficient remote comparison of sets of values. We have shown that this data structure can be used to implement the CRDT merge operator between two full states in an efficient manner even for very large states. This allows to replicate CRDTs in open networks without requiring primitives such as causal broadcast which are a necessity for common delta-based approaches, giving our approach a unique edge in the world of open networks with churn. We have shown in a simulation that Merkle Search Trees are more efficient than vector clocks for detecting changes without any a-priori knowledge in particular in situations of low update rates in very large networks.

Our experimental evaluation of Merkle Search Tree was restricted to a very narrow problem, building an event store, which is equivalent to a grow-only set, one of the simplest known CRDTs. We have also focused on a very simple gossip-based algorithm to drive the distributed reconciliation process. Many other applications and more complex algorithms can be envisioned. This section discusses some of these perspectives.

A. Adaptive Algorithms

In our experimental evaluation, we studied two different methods: Scuttlebutt anti-entropy which is based on vector clocks and Merkle Search Trees. Our experimental evaluation has shown that Merkle Search Trees clearly outperform Scuttlebutt when networks are large and events are produced below a threshold frequency (Figure 5). One could build an adaptive algorithm that automatically selects the best anti-entropy method depending on observed system metrics. Such an algorithm could dynamically switch algorithms when the workload conditions vary in order to obtain the best performance at a lowest cost in bandwidth usage.

B. Merkle DAGs as Persistent Data Structures

Many data structure can be encoded in a Merkle tree or Merkle DAG [18], simply by changing standard pointers into the hashes of the blocks they reference. This enables manipulation of the whole data structure by manipulating the root hash in the manner of persistent data structures in functional programming languages [27]. Contrarily to standard pointers that are local to a machine, Merkle hashes allow us to reference data that is present on another node, making it practical to build very large distributed data structures. CRDTs are a specific case where the data structure is determinstic and operation order-agnostic, which allows for efficient implementations of comparison and reconciliation.

Seeing root Merkle hashes as easily swappable pointers to persistent data structures leads the way to a functional programming approach to distributed programming and opens up we believe interesting avenues to reason about and implement system-wide consistency. This implies using the Merkle data structure as the storage itself, and not only as a way to achieve reconciliation or reparation after node failure.

C. Other Applications of Merkle Search Trees

1) As a primitive for optimizing other CRDTs: Tree-shaped CRDTs have already been proposed [28] and are used in sequence CRDTs such as Treedoc [29] and LSEQ [30]. These approach could benefit from a construction similar to the Merkle Search Tree to avoid balancing issues, which were already identified as an important limitation [28]. Previous solutions suggested to synchronize the whole network so that the trees are re-balanced simultaneously at all nodes. Using a construction that is always balanced by default removes this requirement, which is impractical in large or open networks.

2) Composed with other CRDTs: By using various CRDTs as the value type $\mathbb{V}$, the Merkle Search Tree data structure can be used to create many composed CRDT types. An ordered key-value store could be naturally implemented by using last-writer-wins registers or multi-value registers as the value type $\mathbb{V}$. For CRDT types that need to track causality, such as sets that support deletion or multi-value maps, this simple construction would imply storing the information required for causality tracking alongside each item, thus generating potentially large quantities of redundant data. This issue could be addressed by storing the causality metadata separately.

Transactions, in a sense similar but weaker to snapshot isolation, could be implemented easily in such a system, with no need for distributed locking as in other approaches [31]: the first guarantee of snapshot isolation, that the reads of a transaction are all done on a consistent snapshot of the database, can be obtained simply by not integrating updates from other peers during the transaction. Once a transaction has completed we propagate the set of updates produced and also integrate updates that have been produced on other nodes, all using the CRDT reconciliation rule to resolve conflicts. The use of a Merkle data structure allows such snapshots to be kept easily even when the data is distributed over many nodes, as all data is content-addressed and consists of immutable blocks identified by their hashes. In this model, write operations consist only of reading previously existing blocks and creating new blocks that do not impact the presence and usability of other blocks currently referenced by other nodes.

3) As a secondary indexing data structure: A Merkle Search Tree could also be used as a secondary data structure analogous to B-trees used for indexes in RDBMSs: a large data set that is shared over nodes and stored in a DHT could be indexed in a Merkle Search Tree in order to implement efficient search and SQL-like primitives in large scale peer-to-peer networks. Allowing such uses to scale to large datasets depends on the ability to distribute the Merkle Search Tree data blocks over nodes using a DHT. The feasibility of such an approach when many updates occur at many nodes has not yet been evaluated, and we would like to explore it in a future work, for instance by extending the work of Tamassia and Triandopoulos [19].

4) To hold vector clock information: We presented Scuttlebutt and Merkle Search Trees as two competing methods. However they can be combined: Scuttlebutt could for instance be extended to use a Merkle Search Tree to store the vector clock information, which would enable finding of nodes that have produced new events without having to exchange all of the clock data.

References

[1] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani, "TAO: Facebook's Distributed Data Store for the Social Graph," 2013.

[2] W. Vogels, "Eventually Consistent," *Queue*, vol. 6, Oct. 2008.

[3] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-Free Replicated Data Types," in *Stabilization, Safety, and Security of Distributed Systems*, Springer Berlin Heidelberg, 2011.

[4] P. S. Almeida, A. Shoker, and C. Baquero, "Efficient state-based crdts by delta-mutation," *arXiv:1410.2803 [cs]*, 2014.

[5] A. van der Linde, J. Leitão, and N. Preguiça, "δ-crdts: Making δ-crdts delta-based," PaPoC '16, p. 12:1–12:4, ACM, 2016.

[6] B. Nédelec, P. Molli, and A. Mostéfaoui, "Breaking the Scalability Barrier of Causal Broadcast for Large and Dynamic Systems," *arXiv:1805.05201 [cs]*, May 2018.

[7] R. J. T. Goncalves, P. S. Almeida, C. Baquero, and V. Fonte, "DottedDB: Anti-Entropy without Merkle Trees, Deletes without Tombstones," in *SRDS '17*, 2017.

[8] R. C. Merkle, "A digital signature based on a conventional encryption function," 1987.

[9] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's Highly Available Key-value Store," p. 16, 2007.

[10] R. van Renesse, D. Dumitriu, V. Gough, and C. Thomas, "Efficient Reconciliation and Flow Control for Anti-entropy Protocols," LADIS '08, ACM, 2008.

[11] A. Fox and E. A. Brewer, "Harvest, yield, and scalable tolerant systems," pp. 174–178, 1999.

[12] S. Gilbert and N. Lynch, "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services," *SIGACT News*, 2002.

[13] A. Demers, D. Greene, C. Houser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, "Epidemic algorithms for replicated database maintenance," 1987.

[14] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. van Steen, "Gossip-based peer sampling," *ACM Trans. Comput. Syst.*, 2007.

[15] A. Kermarrec, L. Massoulie, and A. J. Ganesh, "Probabilistic reliable dissemination in large-scale systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, pp. 248–258, Mar. 2003.

[16] P. Euster, R. Guerraoui, A.-M. Kermarrec, and L. Maussoulie, "From epidemics to distributed computing," *IEEE Computer*, 2004.

[17] Y. Zhang, A. Rajimwale, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "End-to-end data integrity for file systems: A zfs case study.," in *FAST*, pp. 29–42, 2010.

[18] J. Benet, "Ipfs-content addressed, versioned, p2p file system," *arXiv preprint arXiv:1407.3561*, 2014.

[19] R. Tamassia and N. Triandopoulos, "Efficient Content Authentication in Peer-to-Peer Networks," in *Applied Cryptography and Network Security*, Springer Berlin Heidelberg, 2007.

[20] S. A. Crosby and D. S. Wallach, "Super-Efficient Aggregating History-Independent Persistent Authenticated Dictionaries," in *ESORICS*, 2009.

[21] R. Bayer and E. M. McCreight, "Organization and maintenance of large ordered indexes," ACM, 1970.

[22] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[23] J. Byers, J. Considine, and M. Mitzenmacher, "Fast approximate reconciliation of set differences," p. 17, 2002.

[24] Y. Minsky, A. Trachtenberg, and R. Zippel, "Set reconciliation with nearly optimal communication complexity," *IEEE Transactions on Information Theory*, vol. 49, p. 2213–2218, Sep 2003.

[25] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Commun. ACM*, vol. 21, pp. 558–565, July 1978.

[26] R. Friedman and S. Manor, "Causal Ordering in Deterministic Overlay Networks," Tech. Rep. CS Technion report CS-2004-04, 2004.

[27] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan, "Making Data Structures Persistent," STOC '86, (New York, NY, USA), ACM, 1986.

[28] M. Shapiro, *A comprehensive study of Convergent and Commutative Replicated Data Types*, p. 1–5. Springer New York, 2011.

[29] N. Preguica, J. M. Marques, M. Shapiro, and M. Letia, "A Commutative Replicated Data Type for Cooperative Editing," in *29th IEEE International Conference on Distributed Computing Systems*, 2009.

[30] B. Nédelec, P. Molli, A. Mostefaoui, and E. Desmontils, "LSEQ: An Adaptive Structure for Sequences in Distributed Collaborative Editing," DocEng '13, ACM, 2013.

[31] T. Schütt, F. Schintke, and A. Reinefeld, "Scalaris: reliable transactional p2p key/value store," in *ERLANG '08*, p. 41, ACM Press, 2008.