Efficient Synchronization of Recursively Partitionable Data Structures

Technische Universität Berlin

Aljoscha Meyer

June 9, 2021

Efficient Synchronization of Recursively Partitionable Data Structures

Abstract

Given two nodes in a distributed system, each of them holding a data structure, one or both of them might need to update their local replica based on the data available at the other node. An efficient solution should avoid redundantly sending data to a node which already holds it.

We give conceptually simple yet asymptotically efficient probabilistic solutions based on recursively exchanging fingerprints for data structures of exponentially decreasing size, obtained by recursively partitioning the data structures. We apply the technique to sets, maps and radix trees. For data structures containing n items, this leads to $\mathcal{O}(log(n))$ round-trips. We give a scheme by which the fingerprints can be computed in $\mathcal{O}(log(n))$ time, based on an auxiliary data structure which requires $\mathcal{O}(n)$ space and which can be updated to reflect changes to the underlying data structure in $\mathcal{O}(log(n))$ time.

To minimize the number of round-trips, the technique requires up to $\mathcal{O}(n)$ space per synchronization session. It can be adapted to require only a bounded amount of memory, which is essential for robust, scalable implementations. While this increases the worst-case number of round-trips, it guarantees continuous progress, even in adversarial environments.

Contents

Chapter 1

Introduction

One of the problems that needs to be solved when designing a distributed system is how to efficiently synchronize data between nodes. Two nodes may each hold a particular set of data, and may then wish to exchange the ideally minimum amount of information until they both reach the same state. Typical ways in which this can happen are one node taking on the state of the other, or both nodes ending up with the union of information available between the two of them.

1.1 Motivating Examples

Distributed version control systems can be seen as an example of the latter case: users independently create new objects which describe changes to a directory, and when connecting to each other, they both fetch all new updates from the other in order to obtain a (more) complete version history. Regarded more abstractly, the two nodes compute the union of the sets of objects they store. A version control system might attempt to leverage structured information about those objects, such as a happened-before relation, but this does not lead to good worst-case guarantees. The set reconciliation protocol we give guarantees the exchange to only take a number of rounds logarithmic in the number of objects.

A different example are peer-to-peer publish-subscribe systems such as Secure Scuttlebutt [TLMT19]. A node in the system can subscribe to any number of topics, and nodes continuously synchronize all topics they share with other nodes they encounter on a randomized overlay network. Scuttlebutt achieves efficient synchronization by enforcing a linear happened-before relation between messages published to the same topic, i.e. each message is assigned a unique sequence number that is one greater than the sequence number of the previous message. When two nodes share interest in a topic, they exchange the greatest sequence number they have for this topic, whichever node sent the greater one then knows which messages the other is missing.

The price to pay for this efficient protocol is that concurrent publishing of new messages to the same topic is forbidden, since it would lead to different messages with the same sequence number, breaking correctness of the synchronization procedure. An unordered pubsub mechanism based on set reconciliation would be able to support concurrent publishing, the other design aspects such as the overlay network could be left unchanged.

An example from a less decentralized setting are incremental software updates.

A server might host a new version of an operating system, users running an old version want to efficiently download the changes. An almost identical problem is that of efficiently creating a backup of a file system to a server already holding an older backup. Both of these examples can abstractly be regarded as updating a map from file paths to file contents. Our protocol for mirroring maps could be used for determining which files need to be updated. The protocol allows synchronizing the actual files via an arbitrary nested synchronization protocol, e.g. rsync [TM+96].

1.2 Efficiency Criteria

There are a variety of criteria by which to evaluate a synchronization protocol. We exemplify them by the trivial synchronization protocol, which consists of both node immediately sending all their data to the other node.

Let n be the number of items held by node $\mathcal{A}$, and m the number of items held by node $\mathcal{B}$. To simplify things we assume for now that all items have the same size in bytes.

The most obvious efficiency criteria are the *total bandwidth* ($\mathcal{O}(n + m)$ bytes for the trivial protocol) and the number of *round-trips* ($1 \in \mathcal{O}(1)$ for the trivial protocol).

Efficiently using the network is not everything, the computational *time complexity per round-trip* must be feasible so that computers can actually run the protocol. It is lower-bounded by the amount of bytes sent in a given round, for the trivial protocol it is $\mathcal{O}(n + m)$.

Similarly, the *space complexity per round-trip* plays a relevant role, since computers have only a limited amount of memory. In particular, if an adversarial node can make a node run out of memory, the protocol can only be run in trusted environments. Even then, when non-malicious nodes of vastly differing computational capabilities interact (e.g. a microcontroller connecting to a server farm), "accidental denial of service attacks" can easily occur. Since the trivial protocol does not perform any actual computation, its space complexity per round-trip is in $\mathcal{O}(1)$.

The *space complexity per session* measures the amount of state nodes need to store across an entire synchronization session, in particular while idly waiting for a response.

In addition to the space required for per-round-trip computations and per session, an implementation of a protocol might need to store auxiliary information that is kept in sync with the data to be synchronized, in order to achieve sufficiently efficient time and space complexity per round-trip. Of interest is not only the *space for the auxiliary information*, but also its *update complexity* for keeping it synchronized with the underlying data.

A protocol might be asymmetric, with different resource usage for different nodes. If there is client and a clear server role, traditionally protocol designs aim to keep the resource usage of the server as low as possible, motivated by the assumption that many clients might concurrently connect to a single server, but a single client rarely connects to a prohibitive amount of servers at the same time.

Any protocol design has to settle on certain trade-offs between these different criteria, which will make it suitable for certain use cases, but unsuitable for others. We do believe that our designs occupy a useful place in the design space that is applicable to many relevant problems, such as those mentioned in the introduction.

A final, "soft" criterium is that of simplicity. While ultimately time and space complexities should guide adoption decisions, complicated designs are often a good indicator that the protocol will never see any deployment. Our designs require merely comparisons of (sums of) hashes, and the auxiliary data structure that enables efficient implementation is a simple balanced tree.

1.3 Recursively Comparing Fingerprints

We conclude the introduction with a brief sketch of a set reconciliation protocol (i.e. a protocol for computing the union of two sets on different machines) that exemplifies the core ideas. The protocol leverages the fact that sets can be partitioned into a number of smaller subsets. The protocol assumes that the sets contain elements from a universe on which there is a total order based on which intervals can be defined, and that nodes can compute fingerprints for any subset of the universe.

Suppose for example two nodes $\mathcal{A}$, $\mathcal{B}$ each hold a set of natural numbers. They can reconcile all numbers within an interval as follows: $\mathcal{A}$ computes a fingerprint over all the numbers it holds within the interval and then sends this fingerprint to $\mathcal{B}$, together with the interval boundaries. $\mathcal{B}$ then computes the fingerprint over all numbers it holds within that same interval. There are three possible cases:

- $\mathcal{B}$ computed the same fingerprint it received, then the interval is considered to be fully reconciled and the protocol terminates. This makes the protocol probabilistic, it is only reliable if the probability of two different sets having the same fingerprint is negligible.
- $\mathcal{B}$ has no numbers within the interval, $\mathcal{B}$ then notifies $\mathcal{A}$, $\mathcal{A}$ transmits all its numbers from the interval, and the interval has been fully reconciled.
- Otherwise, $\mathcal{B}$ splits the interval into two sub-intervals, such that $\mathcal{B}$ has a roughly equal number of numbers within each interval. $\mathcal{B}$ then initiates reconciliation for both of these intervals, the roles $\mathcal{A}$ and $\mathcal{B}$ reverse.

Crucially, in the last case, the two recursive protocol invocations can be performed in parallel. The number of parallel sessions increases exponentially, so the original interval is being reconciled in a number of rounds logarithmic in the greater number of items held by any node within that interval.

1.4 Thesis Outline

The remainder of this thesis fleshes out details and applies the same idea to some data structures, all of which share the property that they can be partitioned into smaller instances of the same data structure.

The viability of this approach hinges on the efficient computation of fingerprints, which is discussed and solved in chapter 2. We then give a thorough definition of the set conciliation protocol in chapter 3, and prove its correctness and its complexity guarantees. **??** gives a more concrete protocol that allows nodes to enforce limits on the amount of computational resources they spend, at the cost of increasing the number of roundtrips if these resource limits are reached. Chapter 4 shows how to

apply the same basic ideas to k-d-trees, maps, tries and radix trees (TODO update as this cristallizes), and briefly discusses why it does not make sense to apply it to arrays. Chapter 5 gives an overview of related work and justifies the chosen approach. We conclude in chapter 6.

Chapter 2

Computing Fingerprints

The protocols we will discuss work by recursively computing and comparing fingerprints of sets. This chapter defines and motivates a specific fingerprinting scheme that admits efficient computation with small overhead for the storage and maintenance of auxiliary data structures. Section 2.1 outlines the solution space and theoretic bounds. Section 2.2 characterizes a family of functions that admit efficient incremental computation, and Section 2.3 proposes members of this family that can be used for fingerprinting, and section 2.4 examine security concerns in the face of malicious parties trying to find fingerprint collisions. We examine randomized solutions that with high probability compute fingerprints in logarithmic time in section 2.5.

2.1 Initial Considerations

Our protocols work by recursively testing fingerprints for equality. For our purposes, we can define a fingerprint or hash function as follows:

Definition 1. A *hash function* is a function $\mathrm{h} : U \to D$ with a finite codomain such that for randomly chosen $u \in U$ and $d \in D$ the probability that $\mathrm{h}(u) = d$ is roughly[1] $\frac{1}{|D|}$. $\mathrm{h}(u)$ is called the *hash of* u, *fingerprint of* u or *digest of* u.

Given a universe U of items, a function $\mathrm{enc} : U \to \{0,1\}^*$ for encoding items as binary strings, a linear order $\preceq$ on U, a hash function $\mathrm{h} : \{0,1\}^* \to \{0,1\}^k$ mapping binary strings to binary strings of length k and some finite $S \subseteq U$, a natural starting point for defining a fingerprint of the set S is to sort the items according to $\preceq$, concatenate the encodings, and hash the resulting string.

While this is straightforward to specify and implement, it does not suffice for our purposes. To allow for efficient set reconciliation, we need to be able to efficiently compute the new fingerprint after a small modification of the set such as insertion or deletion of a single item. Furthermore, we want to be able to efficiently compute the fingerprints of all subsets defined by an interval of the original set.

The fingerprint based on concatenating encodings does not allow for efficient incremental reevaluation. When an item is added to S that is less than any item

[1] To keep the focus on data structure synchronization rather than being sidetracked by cryptography, we will for the most part keep arguments about probabilities qualitative rather than quantitative.

previously in S, the hash function needs to be run over the whole string of length $\mathcal{O}(|S|+1)$ again. Furthermore, for any subinterval of the set, a full fingerprint computation needs to be performed as well. Precomputing the fingerprints of all subintervals requires a prohibitive amount of space. Every subinterval corresponds to a substring of the string consisting of all items in S in ascending order, so there are $\frac{|S|\cdot(|S|+1)}{2}+1 \in \mathcal{O}(n^2)$ many in total.

The go-to approach for efficiently handling small changes to a set of totally ordered items are (balanced) search trees, we briefly state some definitions.

Definition 2. Let U be a set and $\preceq$ a binary relation on U. We call $\preceq$ a *linear order on* U if it satisfies three properties:

anti-symmetry: for all $x, y \in U$: if $x \preceq y$ and $y \preceq x$ then $x = y$

transitivity: for all $x, y, z \in U$: if $x \preceq y$ and $y \preceq z$ then $x \preceq z$

linearity: for all $x, y \in U$: $x \preceq y$ or $y \preceq x$

If $\preceq$ is a linear order, we write $x \prec y$ to denote that $x \preceq y$ and $x \neq y$.

Definition 3. Let U be a set, $\preceq$ a linear order on U, and $V \subseteq U$. Let T be a rooted directed tree with vertex set V.

Let $v \in V$, then T_v denotes the subtree of T with root v.

T is a *binary search tree on* V if for all inner vertices v with left child a and right child b: $a' \prec v$ for all $a' \in T_a$ and $v \prec b'$ for all $b' \in T_b$.

Definition 4. Let $T = (V, E)$ be a binary search tree and $\varepsilon \in \mathbb{R}_{>0}$. We call T ε-*balanced* if $height(T) \leq \lceil \varepsilon \cdot log_2(|V|) \rceil$. Since the precise choice of ε will not matter for our complexity analyses, we will usually simply talk about *balanced* trees.

In the context of fingerprinting, balanced trees often take the form of Merkle trees [Mer89], binary trees storing items in their leaves, in which each leaf vertex is labeled with the hash of the associated item, and inner vertices are labeled with the hash of the concatenation of the child labels. The root label serves as a fingerprint for the set of items stored in the leaves.

When inserting or removing an item, the number of labels that need updating is proportional to the length of of the path from the root to the item, so in a balanced tree of n items $\mathcal{O}(log(n))$. The problem with this approach however is that fingerprints are no longer unique: there are different balanced search trees storing the same items set, and different tree arrangements result in different root labels.

Unfortunately it does not suffice to specify a particular balancing scheme, since different insertion orders of the same overall set of items can result in different trees, even when using the same balancing scheme. While this is sufficient for a setting in which only a single machine updates the set and all other machines apply updates in the same order, as assumed e.g. in [NN00], we aim for a less restrictive setting in which the evolution of the local set does not influence synchronization.

An alternative would be to define exactly one valid tree shape for any set of items, but this precludes logarithmic time complexity of updates, as [Sny77] shows that search, insertion and deletion in such trees take at least $\mathcal{O}(\sqrt{n})$ time in the worst case.

We will inspect two options for cheating this lower bound and to achieve logarithmic complexity: letting the fingerprint function abstract over the tree shape, downgrading it to an implementation detail, which we examine in section 2.2 to section 2.4, or utilizing randomization to define a unique tree layout which allows logarithmic operations with high probability, which we examine in section 2.5.

2.2 Incremental Computations

We now study a family of fingerprinting functions for sets that admit efficient incremental computation based on an auxiliary tree structure, but whose output does not depend on the exact shape of that tree. We first examine a general class of such functions, which reduce a finite set to a single value according to a monoid.

Definition 5. Let M be a set, $\oplus : M \times M \to U$, and $\mathbb{0} \in M$.
We call $(U, \oplus, \mathbb{0})$ a *monoid* if it satisfies two properties:

associativity: for all $x, y, z \in M$: $(x \oplus y) \oplus z = x \oplus y \oplus z$

neutral element: for all $x \in M$: $\mathbb{0} \oplus x = x = x \oplus \mathbb{0}$.

Definition 6. Let U be a set, $\preceq$ a linear order on U, $\mathcal{M} := (M, \oplus, \mathbb{0})$ a monoid, and $\mathrm{f} : U \to M$.
We *lift* f *to finite sets via* $\mathcal{M}$ to obtain $\mathrm{lift}_\mathrm{f}^\mathcal{M} : \mathcal{P}(U) \rightharpoonup M$ with:

$$\begin{aligned}\mathrm{lift}_\mathrm{f}^\mathcal{M}(\emptyset) &:= \mathbb{0}\\ \mathrm{lift}_\mathrm{f}^\mathcal{M}(S) &:= \mathrm{f}(\{\min(S)\}) \oplus \mathrm{lift}_\mathrm{f}^\mathcal{U}(S \setminus \{\min(S)\})\end{aligned}$$

In other words, if $S = \{s_0, s_1, \ldots, s_{|S|-1}\}$ with $s_0 \prec s_1 \prec \ldots \prec s_{|S|-1}$, then $\mathrm{lift}_\mathrm{f}^\mathcal{M}(S) = \mathrm{f}(s_0) \oplus \mathrm{f}(s_1) \oplus \ldots \oplus \mathrm{f}(s_{|S|-1})$.

Functions of the form $\mathrm{lift}_\mathrm{f}^\mathcal{M}$ can be incrementally computed by using labeled binary search trees:

Definition 7. Let U be a set, $S \subset U$ a finite set, $\preceq$ a linear order on U, $\mathcal{M} := (M, \oplus, \mathbb{0})$ a monoid, $\mathrm{f} : U \to M$, and let T be a binary search tree on S.
We define a *labeling function* $\mathrm{label}_\mathrm{f}^\mathcal{M} : S \to M$:

$$\mathrm{label}_\mathrm{f}^\mathcal{M}(v) := \begin{cases}\mathrm{f}(v), & \text{for leaf } v\\ \mathrm{label}_\mathrm{f}^\mathcal{M}(c_<) \oplus \mathrm{f}(v) & \text{v internal vertex with left child } c_< \text{ and no right child}\\ \mathrm{f}(v) \oplus \mathrm{label}_\mathrm{f}^\mathcal{M}(c_<) & \text{v internal vertex with right child } c_> \text{ and no left child}\\ \mathrm{label}_\mathrm{f}^\mathcal{M}(c_<) \oplus \mathrm{f}(v) \oplus \mathrm{label}_\mathrm{f}^\mathcal{M}(c_>) & \text{v internal vertex with left child } c_< \text{ and right child } c_>\end{cases}$$

TODO fix overflow See fig. 2.1 for an example.

Proposition 1. Let U be a set, $S \subset U$ a finite set, $\preceq$ a linear order on U, $\mathcal{M} := (M, \oplus, \mathbb{0})$ a monoid, $\mathrm{f} : U \to M$, and let T be a binary search tree on S with root $r \in S$.
Then $\mathrm{label}_\mathrm{f}^\mathcal{M}(r) = \mathrm{lift}_\mathrm{f}^\mathcal{M}(S)$.

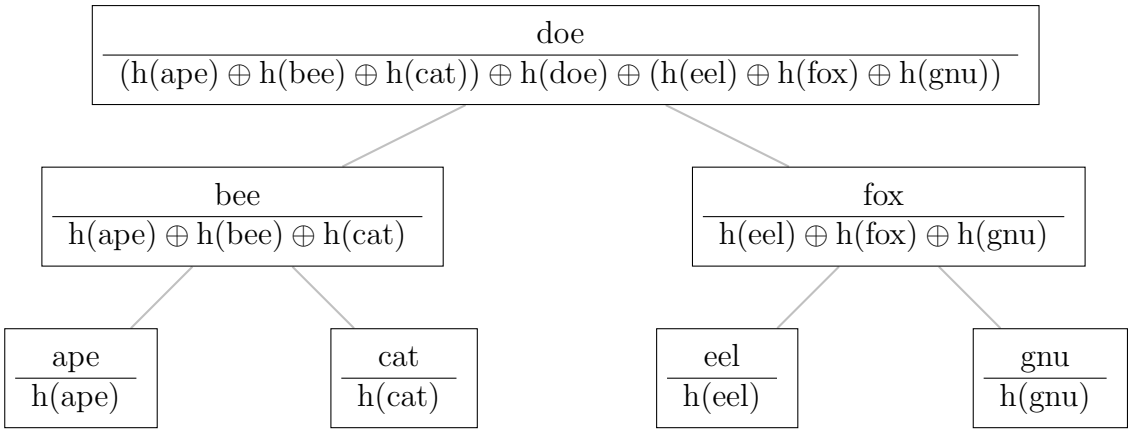

Figure 2.1: A balanced search tree labeled by $\text{label}_{\text{h}}^{(\text{M},\oplus,\mathbb{0})}$. For fingerprinting, h could be a hash function and $\oplus$ the xor operation on fixed-width bitstrings.

Proof. By induction on the number of vertices of T.

IB: If r is a leaf, then $|\text{V}(T)| = 1$ and thus $\text{label}_{\text{f}}^{\mathcal{M}}(r) \stackrel{\text{def}}{=} \text{f}(r) \stackrel{\text{def}}{=} \text{lift}_{\text{f}}^{\mathcal{M}}(\text{V}(T)) = \text{lift}_{\text{f}}^{\mathcal{M}}(S)$.

IH: Let $c_<$ and $c_>$ be vertices for which $\text{label}_{\text{f}}^{\mathcal{M}}(c_<) = \text{lift}_{\text{f}}^{\mathcal{M}}(\text{V}(T_{c_<}))$ and $\text{label}_{\text{f}}^{\mathcal{M}}(c_>) = \text{lift}_{\text{f}}^{\mathcal{M}}(\text{V}(T_{c_>}))$.

IS: If r is an internal vertex with left child $c_<$ and right child $c_>$, then:

$$\begin{aligned}
\text{label}_{\text{f}}^{\mathcal{M}}(r) &\stackrel{\text{def}}{=} \text{label}_{\text{f}}^{\mathcal{M}}(c_<) \oplus \text{f}(r) \oplus \text{label}_{\text{f}}^{\mathcal{M}}(c_>) \\
&\stackrel{\text{IH}}{=} \text{lift}_{\text{f}}^{\mathcal{M}}(\text{V}(T_{c_<})) \oplus \text{f}(p) \oplus \text{lift}_{\text{f}}^{\mathcal{M}}(\text{V}(T_{c_>})) \\
&\stackrel{\text{def}}{=} \text{lift}_{\text{f}}^{\mathcal{M}}(\text{V}(T)) \\
&= \text{lift}_{\text{f}}^{\mathcal{M}}(S)
\end{aligned}$$

The cases for internal vertices with exactly one child follow analogously. □

This correspondence can be used to incrementally compute $\text{lift}_{\text{f}}^{\mathcal{M}}(S)$: Initially, a labeled search tree storing the items in S is constructed. $\text{lift}_{\text{f}}^{\mathcal{M}}(S)$ is the root label. When an item is inserted or removed, only the labels on the path from the root to the point of modification require recomputation, so only a logarithmic number of operations is performed if a self-balancing tree is used.

Note that the exact shape of the tree determines the grouping of how to apply $\oplus$, but by associativity all groupings yield the same result. All trees storing the same set have the same root label.

If U is small enough that space usage of $\mathcal{O}(|U|)$ is acceptable, an implicit tree representation such as a binary indexed tree (Fenwick tree) [Fen94] can be used. Array positions that correspond to some $u \in U \setminus S$ are simply filled with a dummy value whose hash is defined to be $\mathbb{0}$.

2.2.1 Subsets

In addition to incremental computation of the fingerprint of a given set, the reconciliation protocol also requires the efficient computation of the fingerprints of arbitrary

intervals of the given set. We first fix some terminology and notation:

Definition 8. Let $S \subseteq U$, $\preceq$ a linear order over U, and $x, y \in U$.

The *interval from x to y in S*, denoted by $[x, y)_S$, is the set $\{s \in S \mid x \preceq s \prec y\}$ if $x \prec y$, and $S \setminus [y, x)_S$ if $y \preceq x$. We call x the *lower boundary* and y the *upper boundary* of the interval (even if $y \preceq x$).

Note that the upper boundary is excluded from the interval, so in particular $[x, x)_S = S$.

In the remainder of this section, we assume that for all intervals $[x, y)_S$ we have $x \prec y$. If that is not the case, all computations can be performed for the sets $\{s \in S \mid x \preceq s\}$ and $\{s \in S \mid s \prec y\}$ which partition $[x, y)_S$ if $y \preceq x$, and the resulting values can be combined via $\oplus$ to obtain the desired result.

Given a balanced search tree T with root r for a set S that is labeled by $\mathrm{label}_{\mathrm{f}}^{\mathcal{M}}$, we can compute $\mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}([x, y)_S)$ in logarithmic time. Intuitively, one traces paths in T to both x and y, and then the result is the sum over all vertices "in the area between" these paths. For every vertex on the traced paths, the label of the "inner" child vertex summarizes multiple vertices within the area. Summing over all these children yields the value corresponding to the whole inner area. Since the length of the delimiting paths is logarithmic, overall only a logarithmic number of labels needs to be added up.

We now give a precise definition of this algorithm, defining it via Haskell 98 [Jon03] code, not necessarily because it is executable, but mostly to have a typed mathematic notation. For clarity of presentation only complete binary trees are considered. Arbitrary binary trees can also have inner nodes with exactly one child. These can be handled with almost the same algorithm by acting as if these nodes had a second child which contributes `emptyFingerprint` to any application of `combine`.

We start by specifying the types involved in the computation, being slightly more general than currently necessary so that the algorithm can later be reused for certain non-monoidal fingerprinting schemes:

```
-- U is the type of items, D the codomain of the hash function.
hash :: U -> D

-- For now, the set of fingerprints is identical to the set of hashes.
type Fingerprint = D

-- The fingerprint of the empty set, for now the neutral element.
emptyFp :: Fingerprint
emptyFp = 0

-- Combine three fingerprints into a new fingerprint.
combine :: Fingerprint -> Fingerprint -> Fingerprint -> Fingerprint
combine a b c = a + b + c

-- Convert a hash to a fingerprint, for now the identity function.
liftFp :: D -> Fingerprint
liftFp d = d
```

```haskell
-- A node is either a leaf or an inner vertex.
data Node = Leaf U D | Inner Node U Node D

-- Extract the label of a node.
label :: Node -> D
label Leaf _ fp      = fp
label Inner _ _ _ fp = fp
```

The algorithm proceeds by first finding the vertex v with the smallest distance to the root such that $x \preceq v \prec y$. This might be r itself.

```haskell
-- Find the node within [x, y)_S that is closest to the root,
-- assuming x ≺ y.
findInitial :: Node -> U -> U -> Maybe Node
findInitial (Leaf v _) x y
    | x <= v && v < y = Just v
    | otherwise       = Nothing
findInitial (Inner l v r _) x y
    | v < x           = findInitial r x y
    | v >= y          = findInitial l x y
    | otherwise       = Just v
```

If there is no such v, then $[x, y)_S = \emptyset$ and thus $\mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}([x, y)_S) = \mathbb{0}$. If however there is such a v then all vertices of T that are not vertices in T_v are either greater than or equal to y if $v \prec x$, or strictly less than x if $x \prec v$, in either case they do not influence $\mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}([x, y)_S)$.

If v is a leaf, $[x, y)_S = \{v\}$ and thus $\mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}([x, y)_S) = \mathrm{f}(v)$. Otherwise, let $c_<$ be the left child of v and let $c_>$ be the right child. Since all vertices in $T_{c_>}$ are greater than v, they are in particular greater than x. Analogously all vertices in $T_{c_<}$ are less than y.

Keeping in mind that $[x, \max(\mathrm{V}(T_{c_<})))_{\mathrm{V}(T_{c_<})}$ simply denotes the set of vertices in $T_{c_<}$ that are strictly greater than x, and analogously $[\min(\mathrm{V}(T_{c_>})), y)_{\mathrm{V}(T_{c_>})}$ denotes the vertices of $T_{c_>}$ that are less than or equal to y, we have:

$$[x, y)_S = [x, \max(\mathrm{V}(T_{c_<})))_{\mathrm{V}(T_{c_<})} \dot\cup \{v\} \dot\cup [\min(\mathrm{V}(T_{c_>})), y)_{\mathrm{V}(T_{c_>})}$$

implying

$$\mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}([x, y)_S) = \mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}([x, \max(\mathrm{V}(T_{c_<})))_{\mathrm{V}(T_{c_<})}) \oplus \mathrm{f}(v) \oplus \mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}([\min(\mathrm{V}(T_{c_>})), y)_{\mathrm{V}(T_{c_>})})$$

```haskell
-- Compute the fingerprint over all items stored in v
-- within the interval [x, y)_S, assuming x ≺ y.
intervalFingerprint :: Node -> U -> U -> Fingerprint
intervalFingerprint v x y = case findInitial v x y of
    Nothing               -> emptyFp
    Just (Leaf _ fp)      -> liftFp 'sfp
    Just (Inner l v' r _) -> combine
                               (accGeq l x) -- lift_f^M(u ∈ V(T_l) | x ⪯ u)
                               (liftFp (hash v')) -- \f(v')
                               (accLt r y)  -- lift_f^M(u ∈ V(T_l) | u ≺ y)
```

$\mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}([x, \max(\mathrm{V}(T_{c_<}))])_{\mathrm{V}(T_{c_<})}$ can be computed by traversing from $c_<$ to x, summing over those vertices along the path that are greater or equal to x, as well as the right children of all vertices along the path. Correctness follows from a rather technical but straightforward induction we omit.

```
-- Sum up the fingerprints of all items in the given tree
-- which are greater than or equal to x.
accGeq :: Node -> U -> Fingerprint
accGeq (Leaf v fp) x
    | v < x     = emptyFp
    | otherwise = liftFp fp
accGeq (Inner l v r _) x
    | v < x     = accGeq r x
    | otherwise = combine (accGeq l x) (liftFp (hash v)) (label r)
```

Analogously $\mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}([\min(\mathrm{V}(T_{c_>})), y)_{\mathrm{V}(T_{c_>})})$ can be computed by traversing from $c_>$ to y, summing over those vertices along the path that are strictly less than y, as well as the left children of all vertices along the path.

```
-- Sum up the fingerprints of all items in the given tree
-- which are strictly less than y.
accLt :: Node -> U -> Fingerprint
accLt (Leaf v fp) y
    | v >= y    = emptyFp
    | otherwise = liftFp fp
accLt (Inner l v r _) y
    | v >= y    = accLt l y
    | otherwise = combine (label l) + (liftFp (hash v)) + (accLt r y)
```

A simplified approach to these computations can be taken if M has efficiently computable inverses with respect to $\oplus$, i.e. if $(M, \oplus, \mathbb{0})$ is a group.

Definition 9. Let $(M, \oplus, \mathbb{0})$ be a monoid. We call it a *group* if for all $x \in M$ there exists $y \in M$ such that $x \oplus y = \mathbb{0}$. This y is necessarily unique and denoted by $-x$. For $x, y \in M$ we write $x \ominus y$ as a shorthand for $x \oplus -y$.

Observe that $[x, y)_S = [\min(S), y)_S \setminus [\min(S), x)_S$, and thus also $\mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}([x, y)_S) = -\mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}([\min(S), x)_S) \oplus \mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}([\min(S), y)_S)$. We can thus simplify `intervalFingerprint`, getting rid of both `findInitial` and `accGeq`:

```
intervalFingerprintGroup :: Node -> U -> U -> Fingerprint
intervalFingerprintGroup v x y = -(accLt v x) + (accLt v y)
```

Complexity

The worst-case running time occurs when the vertex v with the smallest distance to r such that $x \preceq v \prec y$ is r itself, since then both `sumGeq` and `sumLt` perform a traversal to a leaf of maximal length. Since T is balanced, only a logarithmic number of recursive calls is executed. Assuming f can be computed in $\mathcal{O}(1)$, the resulting time complexity is in $\mathcal{O}(\log(|S|))$.

Neither `accLt` nor `accGeq` are tail-recursive, but they can be rewritten into a tail-recursive form via the standard technique of adding an accumulator argument:

```
accGeq :: Node -> U -> Fingerprint
accGeq v x = accGeq' emptyFp v x

accGeq' :: Fingerprint -> Node -> U -> Fingerprint
accGeq' acc (Leaf v fp) x
    | v < x     = acc
    | otherwise = combine emptyFp (liftFp fp) acc
accGeq' acc (Inner l v r _) x
    | v < x     = accGeq' acc r x
    | otherwise = accGeq' (combine (liftFp (hash v)) (label r) acc) l x

accLt :: Node -> U -> Fingerprint
accLt v y = accLt' emptyFp v y

accLt' :: Fingerprint -> Node -> U -> Fingerprint
accLt' acc (Leaf v fp) y
    | v >= y    = acc
    | otherwise = combine acc (liftFp fp) emptyFp
accLt' acc (Inner l v r _) y
    | v >= y    = accLt' acc l y
    | otherwise = accLt' (combine acc (label l) + (liftFp (hash v))) r y
```

The space complexity of computing the fingerprint for an interval is thus $\mathcal{O}(1)$. Note however that the tail-recursive forms require associativity to compute the same fingerprints as the original versions.

As a closing remark on these computations, we want to point out that associativity not only allows us to abstract over differently shaped binary search trees, but also results in trees of higher degree leading to the same fingerprints. While binary trees simplify the presentation and are theoretically optimal, on actual hardware traversing pointers is expensive and loading sequential memory is cheap, which becomes even more pronounced when the tree resides on secondary storage.

Optimized tree implementations thus have higher branching and store multiple items per vertex, a typical example being B-trees [BM02]. Our fingerprinting scheme admits such optimized implementations, in particular different nodes that want to synchronize data can choose their implementation techniques fully independently and yet maintain compatibility.

2.2.2 Differences to Homomorphic Hashing

The construction of lifting a function to sets resembles that of multiset homomorphic hash functions, introduced in [CDVD+03], with further instantiations given in [CNQ09] and [MSTA17]. We briefly define multiset homomorphic hashing in order to point out salient differences to our construction.

Definition 10. Let $\mathcal{U}_0 := (U_0, \oplus_0, \mathbb{0}_0)$ and $\mathcal{U}_1 := (U_1, \oplus_1, \mathbb{0}_1)$ be monoids and let $f : U_0 \to U_1$.

We call f a *monoid homomorphism from* $\mathcal{U}_0$ *to* $\mathcal{U}_1$ if it satisfies two properties:

preserves operation: for all $x, y \in U_0$: $\mathrm{f}(x \oplus_0 y) = \mathrm{f}(x) \oplus_1 \mathrm{f}(y)$

preserves neutral element: $f(\mathbb{0}_0) = \mathbb{0}_1$

The second property directly follows from the first one by considering the cases $x := \mathbb{0}_0$ and $y := \mathbb{0}_0$.

Definition 11. Let $\mathcal{S} := (\mathbb{N}^U, \cup, \emptyset)$ be the monoid of multisets over the universe U under union, $\mathcal{M} := (M, \oplus, \mathbb{0})$ a monoid, and $\mathrm{f} : \mathbb{N}^U \to M$.

We call f a *multiset homomorphic hash function* if f is a hash function and a monoid homomorphism from $\mathcal{S}$ to $\mathcal{M}$.

A multiset homomorphic hash function allows incrementally maintaining the hash of a multiset under insertions using only $\mathcal{O}(1)$ storage space, by storing the hash of the multiset and combining it with the hash of any inserted item. This can be generalized to support deletions as well: by allowing negative multiplicities for multiset items, multisets form a group under union. If the hash function is a group homomorphism, the hash of the multiset can be updated on deletion by subtracting the hash of the deleted item.

This is more efficient than our approach, which requires $\mathcal{O}(n)$ space for the tree structure. This efficiency comes however at the cost of loss of flexibility. As multiset union is commutative, so must be the homomorphic monoid/group of fingerprints. Since our protocols require the ability to compute fingerprints over arbitrary intervals and we need the tree of size $\mathcal{O}(n)$ to do so anyways, homomorphic hashing would not yield any space usage benefits, and we thus choose the approach that does not require commutativity.

We additionally want to explicitly work with regular sets, not multisets. The reason why homomorphic hashes operate on multisets is that they admit more natural homomorphisms under union. For example, taking the size of a multiset is a homomorphism into the additive monoid on the natural numbers, whereas this is not true for regular sets, as e.g. $|\{x, y\} \cup \{y, z\}| \neq |\{x, y\}| + |\{y, z\}|$. Set union is however well behaved with respect to taking the size of the sets if it is a disjoint union, i.e. no item occurs in both sets.

For an algebraic-flavored characterization of our fingerprint functions, we thus need to account for items occurring on both sides of the union. This alone still would not change that the resulting concept would necessarily require a commutative monoid, so we must also restrict possible inputs to be sorted. This leads to the following definition:

Definition 12. Let U be a set, $\preceq$ a linear order on U, $\mathcal{M} := (M, \oplus, \mathbb{0})$ a monoid, and $\mathrm{f} : \mathcal{P}(U) \rightharpoonup M$ a partial function mapping all finite subsets of U into M.

We call f a *set union somewhatmorphism* if for all finite sets $S_0, S_1 \in \mathcal{P}(U)$ such that $\max(S_0) \prec \min(S_1)$: $\mathrm{f}(S_0 \cup S_1) = \mathrm{f}(S_0) \oplus \mathrm{f}(S_1)$.

This definition captures exactly the functions of form $\mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}$, as shown in the following propositions:

Proposition 2. Let U be a set, $\preceq$ a linear order on U, $\mathcal{M} := (M, \oplus, \mathbb{0})$ a monoid, and $\mathrm{f} : U \to M$.

Then $\mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}$ is a set union somewhatmorphism.

Proof. Let $S_0, S_1 \in \mathcal{P}(U)$ be finite sets such that $\max(S_0) \prec \min(S_1)$. Then:

$$\begin{aligned}\mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}(S_0 \cup S_1) &= \bigoplus_{\substack{s_i \in S_0 \cup S_1, \\ \text{ascending}}} \mathrm{f}(s_i) \\ &= \bigoplus_{\substack{s_i \in S_0, \\ \text{ascending}}} \mathrm{f}(s_i) \oplus \bigoplus_{\substack{s_i \in S_1, \\ \text{ascending}}} \mathrm{f}(s_i) \\ &= \mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}(S_0) \oplus \mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}(S_1)\end{aligned}$$

Proposition 3. Let U be a set, $\preceq$ a linear order on U, $\mathcal{M} := (M, \oplus, \mathbb{0})$ a monoid, and $\mathrm{g} : \mathcal{P}(U) \rightharpoonup M$ a set union somewhatmorphism.

Then there exists $\mathrm{f} : U \to M$ such that $\mathrm{g} = \mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}$.

Proof. Define $\mathrm{f} : U \to M$ as $\mathrm{f}(u) := \mathrm{g}(\{u\})$. We show by induction on the size of $S \subseteq U$ that $\mathrm{g}(S) = \mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}(S)$.

IB: If $S = \emptyset$, then $\mathrm{g}(S) = \mathbb{0} = \mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}(S)$. Suppose that $\mathrm{g}(\emptyset) \neq \mathbb{0}$, this would contradict the fact that for all $x \in U$ we have $\mathrm{g}(\{x\}) = \mathrm{g}(\{x\}) \oplus \mathrm{g}(\emptyset) = \mathrm{g}(\emptyset) \oplus \mathrm{g}(\{x\})$, which holds because $\{x\} = \{x\} \cup \emptyset = \emptyset \cup \{x\}$ and g is a set union somewhatmorphism.

If $S = \{x\}$, then $\mathrm{g}(S) = \mathrm{f}(x) = \mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}(S)$.

IH: For all sets T with $|T| = n$ it holds that $\mathrm{g}(T) = \mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}(T)$.

IS: Let $S \subseteq U$ with $|S| = n + 1$, then:

$$\begin{aligned}\mathrm{g}(S) &= \mathrm{g}(\{\min(S)\}) \oplus \mathrm{g}(S \setminus \{\min(S)\}) \\ &\overset{\text{IH}}{=} \mathrm{g}(\{\min(S)\}) \oplus \mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}(S \setminus \{\min(S)\}) \\ &= \mathrm{f}(\min(S)) \oplus \mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}(S \setminus \{\min(S)\}) \\ &= \mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}(S)\end{aligned}$$

As g is only defined over finite inputs, we thus have $\mathrm{g} = \mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}$. □

2.3 Monoidal Fingerprints

Now that we have characterized a family of functions that admit efficient recomputation in response to changes to the underlying set as well as efficient computation for intervals within the set, the remaining task is to find such functions which are also suitable fingerprints. This consists of deciding on the monoid of fingerprints, and choosing the mapping from items to monoid elements.

As the fingerprint of a singleton set $\mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}(\{u\})$ is equal to $\mathrm{f}(u)$, f must itself already be a hash function. Typical hash functions map values to bit strings of a certain length, i.e. the codomain is $\{0,1\}^k$ for some $k \in \mathbb{N}$. We will thus consider monoids whose elements can be represented by such bit strings.

A natural choice of the monoid universe is then $[0, 2^k)_{\mathbb{N}}$, some simple monoidical operations on this universe include bitwise xor, addition modulo 2^k, and multiplication modulo 2^k. In the following, addition and multiplication will always be implicitly taken modulo 2^k. Note that xor and addition also admit inverses, so the slightly simplified computational fingerprints can be used.

Of these three options, multiplication is the least suitable, because multiplying 0 by any number again yields 0. Consequently for every set containing an item u with $\mathrm{f}(u) = 0$ the fingerprint of the set would also be 0, which clearly violates the criterion that all possible values for fingerprints occur with equal probability.

Addition and xor however are particularly well-behaved in that regard, as they form finite commutative groups:

Proposition 4. Let $\mathcal{G} := (G, \oplus, \mathbb{0})$ be a finite commutative group, i.e. a group with a finite universe such that for all $x, y \in G$: $x \oplus y = y \oplus x$. Let U be a set and let $f : U \to G$ be a hash function.

Then $\mathrm{lift}_{\mathrm{f}}^{\mathcal{G}}$ is a hash function as well.

Proof. We first show that for randomly chosen $x, y, z \in G$ the probability that $x \oplus y = z$ is $\frac{1}{|G|}$.

For $x, z \in G$ there is $y \in G$ such that $x \oplus y = z$, namely $y := z \ominus x$ (because $x \oplus z \ominus x \overset{\text{commutativity}}{=} x \oplus -x \oplus z = z$). As G is finite, this y has to be unique, since otherwise that would not be enough elements left that can be added to x to result in all of the $|G| - 1$ possible remaining z'. Thus for any fixed x, z the probability that a randomly chosen y satisfies $x \oplus y = z$ is $\frac{1}{|G|}$.

Computing $\mathrm{lift}_{\mathrm{f}}^{\mathcal{G}}(S)$ consists of repeatedly adding group elements which by themselves are distributed uniformly at random if S was chosen randomly and f is a high quality hash function. Thus the accumulated value after every step is any given $z \in G$ with probability $\frac{1}{|G|}$, as is in particular the probability for the final result being equal to z. □

A more formal proof for xor specifically is given in section 6.2 of [MEL+21].

By the same argument, knowing the fingerprint for some set does not provide any information about the fingerprints for sets that differ by even only a single value. In conclusion, high quality fingerprints can be achieved by choosing any transitive and commutative operation for the monoid, for example xor or addition modulo 2^k, as long as values are mapped into the monoid with a high quality hash function.

While multiplication does not form a group when performed on the numbers in $[0, 2^k)_{\mathbb{N}}$, there still are groups based on multiplication modulo some number, e.g. $\mathbb{Z}_n^*$, the group yielded by multiplication modulo n on the set $\{x \in [0, n)_{\mathbb{N}} \mid x \text{ is coprime to } n\}$. In the following, when talking about multiplication, we will assume that the universe is chosen such that multiplication forms a group.

2.4 Cryptographically Secure Fingerprints

In the protocols for synchronizing data structures, fingerprints of sets are used for probabilistic equality checking: sets with equal fingerprints are assumed to be equal. Synchronization can thus become faulty if it involves unequal sets with equal fingerprints. If the universe of possible fingerprints is chosen large enough, and the

distribution of fingerprints of randomly chosen sets is random within that universe, the probability for this occurring becomes negligible.

Random distribution of input sets is however a very strong assumption. What if a malicious party can influence the sets to be fingerprinted, with the goal of causing fingerprint collisions and consequently triggering faulty behavior of the system? Cryptographically secure fingerprints are an answer to this problem, being chosen such that it is computationally infeasible for an adversary to find inputs that lead to faulty synchronization.

2.4.1 General Considerations

A typical definition of cryptographically secure hash functions is the following [MVOV18]:

Definition 13. A *secure hash function* is a hash function $\mathrm{h} : U \to D$ that satisfies three additional properties:

pre-image resistance: Given $d \in D$, it is computationally infeasible to find a $u \in U$ such that $\mathrm{h}(u) = d$.

second pre-image resistance: Given $u \in U$ it is computationally infeasible to find a $u' \in U, u' \neq u$ such that $\mathrm{h}(u) = \mathrm{h}(u')$.

collision resistance: It is computationally infeasible to find $u, v \in U, u \ = v$ such that $\mathrm{h}(u) = \mathrm{h}(v)$.

Pre-image resistance has no influence on the vulnerability of the protocol to malicious actors, so all of the following discussion will focus on collision resistance only.

Since $\mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}(\{u\}) = \mathrm{f}(u)$, f must necessarily be collision resistant if $\mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}$ is to be collision resistant. This alone is unfortunately not sufficient, we will see a specific counterexamples in the following subsections. Choosing a secure hash function f always comes with a performance cost, insecure hash functions usually take less time and less space to compute. If the synchronization protocol is only being run in a trusted environment, an insecure hash function might be preferable.

Whether a hash function is secure is not a binary dichotomy, but depends on what is considered "feasible" for an adversary. Greater security can usually be obtained at the cost of longer digests and longer computation times. Before presenting options for secure hash functions, we thus examine the impact of hash collisions first.

We can generally distinguish between malicious actors in two different positions: those who can actively impact the contents of the data structure to be synchronized, and those who passively relay updates and need to search for a collision within the available data. As a set of size n has 2^n subsets, if fingerprints are bit strings of length k, then by the pigeonhole principle a fingerprint collision can be found within any set of size at least $k + 1$.

An attack against the fingerprinting scheme by an active adversary can involve computing many fingerprints and adding the required items to the set once a collision has been found. Such an attack is not usable by the passive adversary. We will primarily focus on discussing active adversaries, as they are strictly more powerful than passive ones. Yet it should be kept in mind that passive adversaries can be more common in certain settings, particularly in peer-to-peer systems: if a node

is interested in synchronizing a data structure, it probably trusts the source of the data, otherwise it would have little reason for expending resources on synchronization. The data may however be synchronized not with the original source but with completely untrusted nodes. Additionally, an active adversary that does not want to risk detection by adding suspicious items to the data structure is restricted to the operations of a passive adversary.

Fingerprint collisions result in parts of the data structure not being synchronized, so information is being withheld from one or both of the synchronizing nodes. When a malicious node synchronizes with an honest one, the malicious node can withhold arbitrary information by simply pretending not to have certain data, which does not require finding collisions at all.

So the cases in which a malicious node can do actual damage by finding a collision are those where it supplies data to two different, honest nodes such that these two nodes perform faulty synchronization amongst each other. Specifically: let $\mathcal{M}$ be a malicious node, $\mathcal{A}$ and $\mathcal{B}$ be honest nodes, then a successful attack consists of $\mathcal{M}$ crafting sets X_A, X_B and sending these to $\mathcal{A}$ and $\mathcal{B}$ respectively, so that when $\mathcal{A}$ and $\mathcal{B}$ then run the synchronization protocol, they end up with different data structures. A passive adversary does not craft X_A, X_B but must find them as subsets of some set X supplied by an honest node. As we assume the underlying hash function f to be secure, at least one non-singleton set has to be involved in a collision.

There are some qualitative arguments that even if an adversary finds a fingerprint collision, the impact is rather low. Let $S_A \subseteq X_A$ and $S_B \subseteq X_B$ be nonequal sets with the same fingerprint. To have any impact on the correctness of a particular protocol run, their two fingerprints need to actually be compared during that run. For that to happen, they need to be of the form $[x_A, y_A)_{X_A}$ and $[x_B, y_B)_{X_B}$ respectively. The fingerprints of these intervals can then be compared if one of the nodes sends the fingerprint for the interval $[\min(x_A, x_B), \max(y_A, y_B))_{X_i}$. That alone is still not sufficient, as any item within that interval that is not part of the sets would change the fingerprint. So the two intervals actually need to be of the form $[\min(x_A, x_B), \max(y_A, y_B))_{X_A}$ and $[\min(x_A, x_B), \max(y_A, y_B))_{X_B}$, or simplified: there have to be $x, y \in U$ such that $S_A = [x, y)_{X_A}$ and $S_B = [x, y)_{X_B}$.

If the adversary has found such sets, that is still no guarantee that the interval $[x, y)_{X_i}$ is being compared during the synchronization session of $\mathcal{A}$ and $\mathcal{B}$. For a set containing n items, there are $n^2 - (n-1) \in \mathcal{O}(n^2)$ distinct intervals, but only $\mathcal{O}(n)$ are compared in a given protocol run, since the worst-case message complexity is $\mathcal{O}(n)$ (see section 3.3.3). These numbers should only be considered as rough guidelines, they gloss over details such as the fact that there are more intervals containing roughly $\frac{n}{2}$ items then there are intervals containing almost all or almost no items. Yet they demonstrate that finding suitable colliding intervals still does not guarantee that a particular pair of nodes will be affected. In particular, there is no need for $\mathcal{A}$ and $\mathcal{B}$ to choose the interval boundaries that occur in a protocol run deterministically (see section 3.3.4). They can even perform multiple randomized protocol runs in parallel, while keeping track of item transmissions and sending every item at most once across all these protocol runs.

Another factor mitigating the impact of an adversary finding fingerprint collisions is the communication with other, non-colluding nodes. A fourth party could send some $u \in U, x \preceq u \prec y$ to $\mathcal{A}$ or $\mathcal{B}$ before $\mathcal{A}$ and $\mathcal{B}$ synchronize, disrupting the collision.

Finally, in systems where nodes repeatedly synchronize with different other nodes, a single fingerprint collision in a single synchronization session would merely delay propagation of information rather than stop it completely (note that this does not hold for collisions of two singleton sets). Since the attack model requires both $\mathcal{A}$ and $\mathcal{B}$ to synchronize with more than one node in total, this might apply to many affected settings. Peer-to-peer systems communicating on a random overlay network in particular fall into this category. A malicious actor with enough control over the communication of other nodes to guarantee a tangible benefit from fingerprint collisions can likely disrupt operation of the network more effectively by exercising that control than by sabotaging synchronization.

All of these arguments are however purely qualitative and should as such be taken into account with caution, they are not a substitute for quantitative cryptographic analysis. A strong attacker might be able to find many pairs of sets of colliding fingerprints, or many sets that all share the same fingerprint, and none of the above arguments consider these cases.

In a system with a consensus mechanism among all participating nodes, the choice of f can periodically be changed, with the frequency being some function of the time it takes to find a viable collision, the cost of rebuilding all auxiliary data structures, and the general level of paranoia among the participating nodes. The $\mathcal{O}(n)$ cost of rebuilding the data structures is then being amortized over the number of synchronization sessions occurring between rebuilds, which is still an improvement over protocols that need to perform $\mathcal{O}(n)$ computations per synchronization session.

2.4.2 Specific Monoids

Multiplication, addition and xor as a way of combining hashes have been studied in [BM97], in the context of hashing sequences, of which our ordered sets are a special case. Their setting requires not only associativity but also commutativity. They thoroughly break xor by reducing the problem of finding a collision to that of solving a system of linear equations. We will not restate the full attack, but we will describe a weaker but simpler mechanism based on similar ideas that allows finding subsets whose fingerprint is a specific target value. This can be used as an optimization in a trusted setting (see section 3.4.2).

The main observation is that xor is the additive operation in $\mathbb{F}_2$, the finite field on two elements $\{0, 1\}$. A fingerprint can be interpreted as a vector $\begin{bmatrix} b_1 & b_2 & \dots & b_k \end{bmatrix} \in \{0,1\}^{1\times k}$. The fingerprint of a set $S = \{s_1, s_2, \dots, s_n\}$ can thus be computed as the sum (within $\mathbb{F}_2$, i.e. as the xor) of the vectors corresponding to $s_1, s_2, \dots, s_n$. The fingerprint of some $S' \subseteq S$ can also be regarded as the sum over all vectors, but each vector being first multiplied with a coefficient of 1 if $s_i \in S'$ and coefficient 0 if $s_i \notin S'$. In other words, the fingerprint of S' is a linear combination of the hashes of the items in S.

This enables us to efficiently find subsets whose fingerprint are a particular vector. Let $S = \{s_1, s_2, \dots, s_n\}$ be a set of items, and let $b = \begin{bmatrix} b_1 & b_2 & \dots & b_k \end{bmatrix} \in \{0,1\}^{1\times k}$ be the target fingerprint. For $0 < i \leq n$ define $a_i = \begin{bmatrix} a_{i,1} & a_{i,2} & \dots & a_{i,k} \end{bmatrix} \in \{0,1\}^{1\times k}$ to be $\mathrm{f}(s_i)$ interpreted as a vector over $\mathbb{F}_2$. The coefficients for linear combinations of the a_i equal to b are the solutions to the following system of linear equations over $\mathbb{F}_2$:

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,k} \\ a_{2,1} & a_{2,2} & \dots & a_{2,k} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \dots & a_{n,k} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

Solutions can be found by gaussian elimination in $\mathcal{O}(n^3)$ time, an exponential improvement over brute forcing by computing the fingerprints of all subsets.

As xor admits polynomial-time attacks by solving linear equations, the authors of [BM97] next consider addition and multiplication. They unify parts of their discussion by relating the hardness of finding collisions to solving the balance problem: in a commutative group $(G, \oplus, \mathbb{0})$, given a set of group elements $S = \{s_1, s_2, \dots, s_n\}$, find disjoint, nonempty subsets $S_0 = \{s_{0,0}, s_{0,1}, \dots, s_{0,k}\} \subseteq S, S_1 = \{s_{1,0}, s_{1,1}, \dots, s_{1,l}\} \subseteq S$ such that $s_{0,0} \oplus s_{0,1} \oplus \dots \oplus s_{0,k} = s_{1,0} \oplus s_{1,1} \oplus \dots \oplus s_{1,l}$. They then reduce the hardness of the balance problem to other problems.

For addition, the balance problem is as hard as subset sum, which was at the time of publication conjectured to be sufficiently hard. Wagner showed however in [Wag02] how to solve the balance problem in subexponential time for addition. To give an impression for the impact of this attack, consider [MGS15] which uses addition for combining SHA-3 [Dwo15] digests, producing fingerprints of length between 2688 and 4160 or 6528 to 16512 bits to achieve security levels of 128 or 256 bit respectively against Wagner's attack. [Lyu05] gives an improvement over Wagner's attack finding collisions in $\mathcal{O}(2^{n^\varepsilon})$ for arbitrary $\varepsilon < 1$, further weakening addition as a choice of monoid operation.

For multiplication, the balance problem is as hard as the discrete logarithm problem in the group. This is a more "traditional" hardness assumption than subset sum, there are groups for which no efficient algorithm is known. The main drawback is that multiplication is less efficient to compute that addition. [SMBA10] includes a comparison between the performance of addition and multiplication for incremental hashing, the additive hash outperforms the multiplicative one by two orders of magnitude, even though it uses longer digests to account for Wagner's attack.

For our context in which fingerprints are frequently sent over the network, longer computation time might still be preferable over longer hashes. Fingerprints based on multiplication nevertheless need larger digests than traditional, non-incremental hash functions. [MSTA17] suggests fingerprints of 3200 bit to achieve 128 bit security, and motivated by this uses binary elliptic curves as the underlying group to achieve more compact fingerprints - fingerprints of length $2k$ bits give $\mathcal{O}(2^k)$ security.

[BM97] also proposes a fourth monoid based on lattices. [LKMW19] give a specific instantiation providing 200 bits of security with fingerprints of size $16 \cdot 1024 = 16384$ bit.

All of the preceeding monoid operations are commutative, which our approach to fingerprint computation does not require. A typical associative but not commutative operation is matrix multiplication. Study of a family of hash functions based on multiplication of invertible matrices was initiated in [Zém91], whose security is related to solving hard graph problems on the Caley graph of the matrix multiplication group. [PQ+11] gives a good overview about the general principle and the security aspects of Caley hash functions.

While [TZ94], an improvement over the originally proposed scheme, has been successfully attacked in [GIMS11] [PQ10], there are several modifications [Pet09] [BSV17] [Sos16] for which no attacks are currently known, and [MT16] showed random self-reducibility for Caley hash functions.

Whereas the schemes based on commutative groups operate on two bitstrings at a time, the Caley hash functions operate on two individual bits at a time. Attacks thus assume that the manipulated bits can be freely chosen, using this to e.g. craft palindromic inputs. This may mean that such attacks are hard to apply to our setting where the input bits are produced in non-reorderable batches by another, cryptographically secure hash function. After finding a bit sequence that yields a Caley hash collision, an attacker still needs to find items for which the concatenation of their hashes is the desired it sequence.

Alternatively, one can forgo the additional hash function and apply the Caley hash function directly to the encodings of items. This simplifies the overall scheme and removes reliance on a second hash function. On the other hand, if one expects the additional hash function to be harder to attack than the Caley hash, then making the attacks against the Caley hash function less flexible might overall make attacks more difficult. Furthermore, Caley hash functions are usually slower than typical choices for the hash function from items to bitstrings, so using both can produce significant speedups if items have long encodings.

Aside from Caley hashes, we are not aware of any non-commutative monoids used for hashing. We would further like to note that any hashes based on groups still have more structure than required for our designs, as we don't need existence of inverse elements. Strictly speaking our designs do not even require a neutral element: the fingerprint of the empty set never needs to be exchanged in a protocol run, so we could have just as well defined fingerprints for all nonempty sets without relying on the neutral element to do so. The presentation we chose is merely more elegant. And finally, we do not require pre-image resistance for our fingerprints. Suitable hash functions could thus be located in a more general design space than studied in any literature we know of.

2.5 Pseudorandom Data Structures

Monoidal fingerprints allow efficient deterministic computation of fingerprints, but at the price of having to rely on some non-standard cryptographic primitives. A more established cryptographic construction is that of Merkle trees [Mer89] and related data structures, all based on hashing the concatenation of child hashes. There is a significant body of work in the context of authenticated data structures based on these constructions, from rather simple balanced binary trees [NN00] to arbitrary acyclic directed graphs [MND+04], and we will be able to refer to such work for proofs of collision resistance.

The Merkle construction is however not associative when using a cryptographically secure hash function h, as $\mathrm{h}(\mathrm{h}(\mathrm{h}(a) \cdot \mathrm{h}(b)) \cdot \mathrm{h}(c)) = \mathrm{h}(\mathrm{h}(a) \cdot \mathrm{h}(\mathrm{h}(b) \cdot \mathrm{h}(c)))$ would constitute a violation of collision resistance. Different such tree representations of the same set thus lead to completely different fingerprints. So in order to use this construction, there must be a unique search tree representation defined for every set.

[Sny77] shows that deterministic search trees with unique shapes require $\mathcal{O}(\sqrt{n})$

time for insertions and deletions in the worst case, making them unsuitable for our use case. A natural extension is to look at randomized data structures that define a unique presentation which allows modification and search in $\mathcal{O}(\log(n))$ with high probability.

This has been thoroughly researched in the field of *history-independent data structures*, intuitively speaking data structures whose bitlevel representations do not leak information about their construction sequence. This requirement has been shown in [HHM+05] to be equivalent to having a unique representation. For more background, we refer to the excellent introduction of [BBJ+16].

In the following, we present two such data structures for representing sets - a randomized binary tree, and the skip list - and define Merkle-like fingerprinting schemes for them such that the fingerprints of arbitrary intervals can be computed in logarithmic time with high probability. As all nodes need to arrive at the same representation, no true randomness can be involved. Instead, all random decisions are based on pseudorandom bits derived from the data itself.

These schemes allow cryptographically secure fingerprints whose collision resistance does not depend on uncommon cryptographic building blocks. The price to pay is that computing fingerprints can take linear time in the worst case. Note however that the communication complexity of any synchronization protocol using these fingerprints is completely unaffected by the randomized computation time.

Unfortunately, adversaries can efficiently create data sets for which the randomized solutions degraded to linear performance, thus enabling denial-of-service attacks as the cost of computing the fingerprints during a protocol run can be made to dominate the communication cost. Passive adversaries however cannot effectively attack these fingerprints: even though they can technically only forward those parts of the data structure that have degraded performance relative to the number of items in those parts, they would still increase the absolute resource usage of their victims if they simply forwarded all data.

Proponents of randomized data structures also claim that they are significantly easier to implement than balanced search trees (see e.g. [SA96] or [Pug90]), making it worthwhile to consider them even in trusted environments, in which their expected logarithmic update complexities suffice.

2.5.1 Pseudorandom Treaps

A *treap* [SA96] is a search tree in which every vertex has an associated *priority* from a totally ordered set, and in which the priorities of the vertices form a heap:

Definition 14. Let U be a set, $\prec$ a linear order on U, P a set, $\leq$ a linear order on P, priority : $U \to P$, $V \subseteq U$, and T a binary search tree on V.

Then we call T a treap if for all $v \in V$ with parent p it holds that priority$(v) \leq$ priority(p).

From the properties of treaps shown in [SA96] we will rely on the following:

- If the priorities of all vertices are pairwise unequal, then there is exactly one treap on the vertex set, which is equal to of the search tree obtained by successively inserting items in decreasing priority without any balancing.

- If the priorities are distributed uniformly at random, the height of the treap is expected to be logarithmic in the number of vertices.
- Inserting or deleting an item while maintaining the treap properties can be done in time proportional to the height of the treap.

In the following, we will fix some cryptographically secure hash function $\mathrm{p} : U \to \{0,1\}^k$, and define $\mathrm{priority}(u) := \mathrm{p}(u)$, using the numeric order on $\{0,1\}^k$ for comparing priorities. Since p is collision resistant, we can assume the resulting treaps to be unique, and since the output of a secure hash function is indistinguishable from a random mapping, we can assume the resulting treaps to have expected logarithmic height.

Treaps store items in every vertex, whereas Merkle trees only store items in their leaves. [BLL02] proposes the following natural generalization and proves that the labels are collision free if the underlying hash function h is collision free:

$$\mathrm{treaplabel_h}(v) := \begin{cases} \mathrm{h}(v), & \text{for leaf } v \\ \mathrm{treaplabel_h}(\mathrm{treaplabel_h}(c_<) \cdot \mathrm{treaplabel_h}(v)) & \text{v internal vertex with l} \\ \mathrm{treaplabel_h}(\mathrm{treaplabel_h}(v) \cdot \mathrm{treaplabel_h}(c_>)) & \text{v internal vertex with r} \\ \mathrm{treaplabel_h}(\mathrm{treaplabel_h}(c_<) \cdot \mathrm{h}(v) \cdot \mathrm{treaplabel_h}(c_>)) & \text{v internal vertex with l} \end{cases}$$

TODO fix overflow

We will call a treap labeled according to this scheme a *Merkle treap*. Since Merkle treaps are unique, we can define the fingerprint of some set as the root label of the Merkle treap on that set if it is nonempty, or `Nothing` otherwise.

Subsets

Now that we have defined the fingerprint of a set, we need a way of efficiently computing the fingerprint of any interval $[x, y)_S$ given the Merkle treap of the set S. The algorithm can be expressed in exactly the same framework as that of section 2.2.1, changing merely some type definitions and the `combine` function:

```
-- A fingerprint is now either a hash or Nothing.
type Fingerprint = Maybe D

emptyFp :: Fingerprint
emptyFp = Nothing

liftFp :: D -> Fingerprint
liftFp d = Just d

-- Combine three fingerprints into a new fingerprint.
combine :: Fingerprint -> Fingerprint -> Fingerprint -> Fingerprint
combine Nothing Nothing Nothing = Nothing
combine (Just a) Nothing Nothing = Just a
combine Nothing (Just b) Nothing = Just b
combine Nothing Nothing (Just c) = Just c
combine (Just a) (Just b) Nothing = Just (hash (concat a b))
```

```
combine (Just a) Nothing (Just c) = Just (hash (concat a c))
combine Nothing (Just b) (Just c) = Just (hash (concat b c))
combine (Just a) (Just b) (Just c) = Just (hash (concat (concat a b) c))
```

While the tree traversal and accumulation of data otherwise stays the same, the nature of the correctness argument changes quite a bit, since we cannot rely on associativity of `combine` anymore. Instead, the core idea of the argument is that the computed fingerprint depends only on the relative positioning between the tree vertices within the interval, which stays identical no matter how many vertices outside the interval are added.

Recall that the shape of a treap is always that of the tree obtained by inserting items sorted by decreasing priority without rebalancing. Now consider the vertex v with the smallest distance to the root r such that $x \preceq v \prec y$, i.e. the vertex computed by `findInitial`. All items within $[x, y)_S$ are contained in $\mathrm{V}(T_v)$. The fingerprint computation for the interval is clearly local to T_v, and while insertion of items outside that tree might change the path from r to v, they do not change the shape of T_v.

It thus remains to show that `accGeq` and `accLt` compute the same result no matter how many items outside the interval are contained in the tree they process. We give the full argument for `accGeq`, the case for `accLt` follows analogously.

Proof. Let v be the root of a tree T_v and $x \in U$. We show by induction on $|\mathrm{V}(T_v)|$ that `accGeq` computes a value equal to the root label of the Merkle treap on $S_{\succeq} := \{u \in \mathrm{V}(T_v) \mid x \preceq u\}$ (or `Nothing` if the set is empty).

IB: If v is a leaf with $v \prec x$, then $(\texttt{accGeq v x}) = \texttt{Nothing}$ and indeed $S_t = \emptyset$. If v is a leaf with $x \preceq v$, then $(\texttt{accGeq v x}) = \texttt{Just}\,\mathrm{h}(\texttt{v}) = \texttt{Just}\,\mathrm{treaplabel_h}(\texttt{v})$.

IH: Let $n \in \mathbb{N}$ and T_r be a tree rooted at r with $|\mathrm{V}(T_r)| = n$. Then $(\texttt{accGeq v x}) = \mathrm{treaplabel_h}(v)$.

IS: Let T_v be a tree rooted at v with $|\mathrm{V}(T_v)| = n + 1$, let $c_<$ be the right child of v and let $c_>$ be the left child of v.

Case 1. If $v \prec x$, then:

$$\begin{aligned}(\texttt{accGeq v x}) &\overset{def}{=} (\texttt{accGeq c}_> \texttt{ x}) \\ &\overset{IH}{=} \mathrm{treaplabel_h}(c_>) \\ &\overset{v \prec x}{=} \mathit{tlabel}\mathrm{h}(v)\end{aligned}$$

Case 2. Otherwise, $x \preceq v$. Neither v nor any vertex in $T_{c_<}$ has any influence on the shape of $T_{c_>}$. We thus have:

$$\begin{aligned}(\texttt{accGeq v x}) &\overset{def}{=} (\texttt{combine } (\texttt{accGeq c}_< \texttt{ x})\ \mathrm{h}(\texttt{v})\,\mathrm{treaplabel_h}(\texttt{c}_>)) \\ &\overset{IH}{=} (\texttt{combine }\ \mathrm{treaplabel_h}(\texttt{c}_<)\ \mathrm{h}(\texttt{v})\,\mathrm{treaplabel_h}(\texttt{c}_>)) \\ &\overset{def}{=} \mathrm{treaplabel_h}(\mathrm{treaplabel_h}(c_<) \cdot \mathrm{h}(v) \cdot \mathrm{treaplabel_h}(c_>)) \\ &\overset{def}{=} \mathrm{treaplabel_h}(v)\end{aligned}$$

The computation time is proportional to the height of the treap, so expected $\mathcal{O}(\log(n))$. But as the hash function is not associative, the tail-recursive versions of `accGeq` and `accLt` do not compute the correct result. Intuitively, the problem is that the computation traverses the tree from top to bottom, but the fingerprints are defined through a computation from the bottom to the top. Accordingly, tail-recursive formulations can be achieved by adding parent-pointers to all vertices and performing the computation by first traversing to the interval boundary and then accumulating fingerprints while moving back up through the tree. The space complexity thus remains in $\mathcal{O}(1)$.

Just as with balanced search trees, a binary treap might be the most natural formulation, but not the most efficient one on real hardware. [Gol09] describes B-treaps, which are treaps of higher degree. If such an optimized realization is chosen, the need to map between the actual data layout and the shape of the treap which defines the fingerprints complicates the implementation, in particular compared to monoidal fingerprints which naturally abstract over tree implementation details.

Beyond treaps there are other randomized search trees. [PT89] suggests hash tries for fingerprinting sets, and our mechanism for computing fingerprints of intervals is compatible with their approach. The reason we opted for treaps is that the pseudorandom selection of the tree shape is completely decoupled from the ordering over the items, whereas in a hash trie the ordering follows from the tree shape (or vice versa, depending on the viewpoint). As our protocols can benefit from a non-arbitrary ordering (compare section 3.1.1), treaps are the more flexible solution.

There exists also some work on deterministic unique tree representations that stay efficient if the number of items is either a particularly small or a particularly large fraction of the size of the universe, see [ST94] for both a specific approach and more related work. These approaches could also be adapted for our purposes, but since they pose restrictions on the number of items that can be stored, they are less flexible than our other approaches.

Adversarial Treap Construction

The main motivation for looking at pseudorandom schemes was to be able to rely on well-known, conventional, non-associative hash functions, implying that resilience against malicious peers is desired. While Merkle-treaps are collision-resistant, they open up another angle of attack: a malicious party can try to construct unbalanced treaps to make the cost of computing fingerprints super-logarithmic in the number of items, leading to a denial-of-service attack as the cost of computing the fingerprints during a protocol around dominates the communication cost.

There is a fairly straightforward way of creating treaps on n vertices of height n in expected $\mathcal{O}(n^2)$ time and $\mathcal{O}(1)$ space, making treaps unsuitable for an adversarial environment. A sequence $(u_0, u_1, \ldots, u_{n-1})$ of items such that the treap on these items has height n can be computed by successively choosing arbitrary but always increasing with respect to $\preceq$ items p_j and letting $u_i := p_j$ if $\lfloor i \cdot \frac{2^k}{n} \rfloor \leq \text{priority}(p_j) < \lfloor (i+1) \cdot \frac{2^k}{n} \rfloor$ (recall that k is the number of bits of a hash digest). The probability for the priority of an arbitrary item to fall within the desired interval is $\frac{1}{n}$, so finding one takes expected $\mathcal{O}(n)$ time, finding a sequence of n of these accordingly takes expected $\mathcal{O}(n^2)$ time.

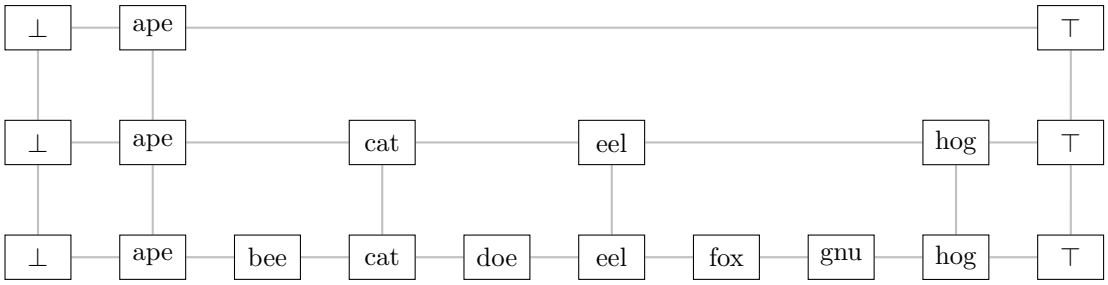

Figure 2.2: An example skip list, demonstrating the layers of sorted linked lists. The $\bot$ and $\top$ vertices are the start and end points for all lists.

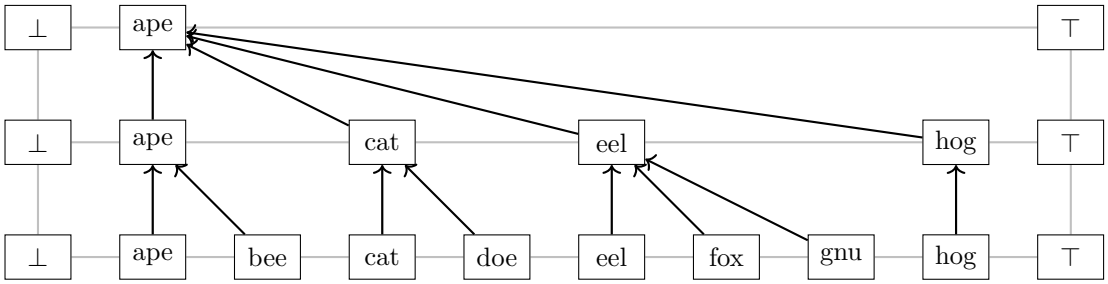

Figure 2.3: A visualization of which labels influence which other labels. One can see that these dependencies form a tree.

2.5.2 Pseudorandom Skip Lists

The skip list [Pug90] [Pug98] is a probabilistic data structure for storing sets that does not use a tree representation, but rather a hierarchy of progressively sparser linked lists containing set items in sorted order. The list at layer 0 contains every item, the list at layer $i+1$ contains every item from layer i with a probability of $\frac{1}{2}$. Intuitively, when inserting an item into the skip list, one performs coin flips until one loses, the item is then added at the correct position into as many layers as one performed coin flips. Figure 2.2 gives an example.

To define a pseudorandom skip list, we again fix some cryptographically secure hash function $\mathrm{p} : U \to \{0,1\}^k$, and determine the layers of an item u by counting the number of leading zero bits of $\mathrm{p}(u)$ and adding one.

Toward a fingerprinting scheme, we define a labeling over all items in all layers. Let $u_{l,i}$ denote the i-th item in layer l, recall that h is some cryptographically secure Hash function. Then $\mathrm{skiplabel_h}(u_{0,i}) := \mathrm{h}(u_{0,i})$ and $\mathrm{skiplabel_h}(u_{l+1,i}) := \mathrm{h}(\mathrm{skiplabel_h}(u_{l,j}) \cdot \mathrm{h}(\mathrm{skiplabel_h}(u_{l,j+1}) \cdot \mathrm{h}(\ldots \cdot \mathrm{skiplabel_h}(u_{l,j+m}))))$ where $(u_{l,j} \ldots u_{l,j+m})$ is the longest subsequence of layer l such that $u_{l,j} = u_{l+1,i}$ and $u_{l,j+m} \prec u_{l+1,i+1}$. Figure 2.3 visualizes an example.

Every label contributes to the computation of at most one other label, which is located on the next layer. Maintaining labels when inserting or deleting an item thus takes a number of hash computations bounded by the height of the skip list, which is expected $\mathcal{O}(\log(n))$. The number of labor concatenations that need to be performed at each layer is also randomized, but on average there are two, as the probability for an item to also occur at the next layer is $\frac{1}{2}$. The expected time thus stays in $\mathcal{O}(\log(n))$.

Denote by $\mathrm{domain}(u_{l,i})$ the set of items u such that $u_{l,i} \preceq u \prec u_{l,i+1}$, i.e. the

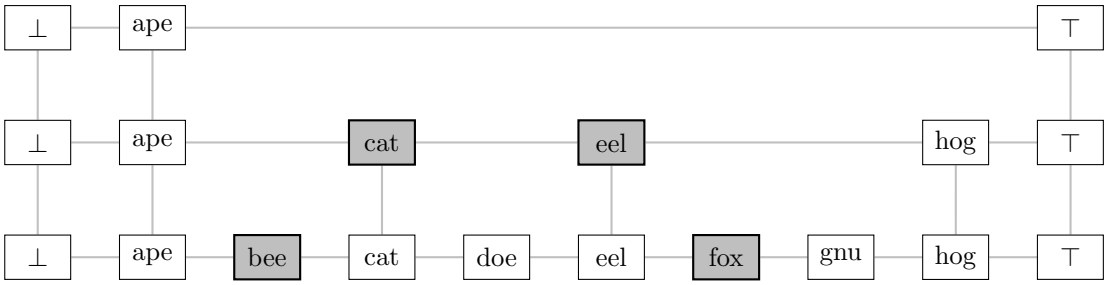

Figure 2.4: The skip list nodes contributing to the fingerprint of the interval $[\text{bee}, \text{gnu})_S$. If the interval had been $[\text{bee}, \text{fox})_S$, it would have again been the topmost eel vertex whose label would have been hashed.

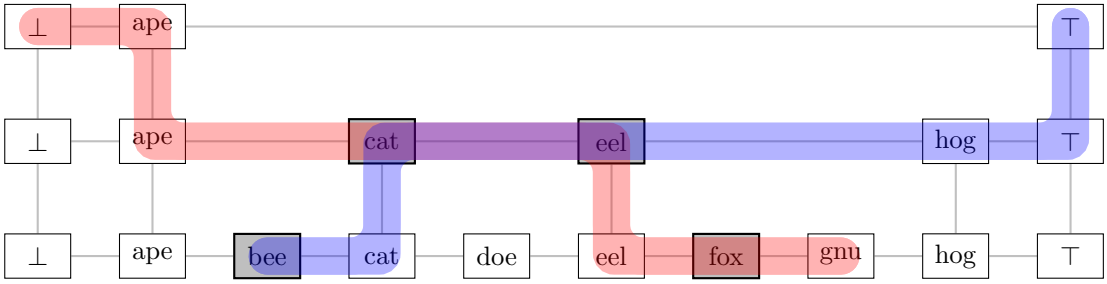

Figure 2.5: The forward search path to gnu (red) and the backward search path to bee (blue) allow to read off which vertices contribute to the fingerprint of $[\text{bee}, \text{gnu})_S$.

set of all items whose hash contributes to the label of $u_{l,i}$. Now let $S \subseteq U$ be a finite set. The fingerprint of S is defined as `Nothing` if $S = \emptyset$, and otherwise as $\mathrm{h}(u_{l_0,i_0} \cdot \mathrm{h}(u_{l_1,i_1} \cdot \mathrm{h}(\ldots \cdot u_{l_m,i_m})))$ where $(u_{l_0,i_0}, u_{l_1,i_1}, \ldots, u_{l_m,i_m})$ is the unique shortest sequence of vertices in the pseudorandom skip list on S such that its items are strictly increasing and their domains partition S, choosing the vertex of the highest possible layer if multiple vertices have equal domains. See fig. 2.4 for an example.

We next sketch an algorithmic way of determining this sequence for any interval $[x, y)_S$ given the deterministic skip list for S, but refrain from giving correctness proofs as they are rather verbose and convey no particular insight that is not better conveyed in the accompanying figures.

Let $\bar{u} = (u_0, u_1, \ldots, u_m)$ be the shortest path in the skip list from its beginning to the vertex y excluding y itself, i.e. the sequence of vertices traversed when looking up y in the skip list. This sequence has expected logarithmic length and can be computed in expected logarithmic time. Obtain a shorter sequence $\bar{u'} = (u'_0, u'_1, \ldots, u'_{m'})$ by keeping from all vertices in $\bar{u}$ that correspond to the same item the one in the greatest layer such that its domain does not contain any item greater than or equal to y. Figure 2.5 provides an example.

Now let $\bar{v} = (v_0, v_1, \ldots, v'_o)$ be the shortest path in the skip list obtained by inverting the order on items from its beginning to the vertex x. Intuitively, this corresponds to looking up x "from the back" in a doubly-linked skip list. Obtain from $\bar{v}$ the corresponding shorter sequence $\bar{v'} = (v'_0, v'_1, \ldots, v'_{o'})$ by keeping from all vertices that correspond to the same item the one of the greatest layer such that its domain does not contain any item strictly less than x.

We claim that $\bar{v'}$ and $\bar{u'}$ always intersect (if $[x, y)_S \neq \emptyset$) in at least one vertex, i.e. there are unique $v'_i \in \bar{v'}$ and $u'_j \in \bar{u'}$ with $v'_i = u'_j$. Then the sequence $(v'_0, v'_1, \ldots, v'_i =$

$u'_j, u'_{j+1}, \ldots, u'_m)$ is the unique shortest sequence of vertices of strictly increasing items whose domains partition $[x, y)_S$. Since $\bar{v'}$ and $\bar{u'}$ can be computed in expected logarithmic time, so can this sequence, and thus fingerprints can be computed in expected logarithmic time.

Adversarial Skip List Construction

Pseudorandom skip lists are even less suited for adversarial environments then pseudorandom treaps. Performance degrades if most items have the same maximum layer. A randomly chosen item has a hash beginning with a 1 bit with probability $\frac{1}{2}$, these items only reside in layer 0. An adversary can thus create a linear list of exclusively layer 0 items of length n in expected $2n \in \mathcal{O}(n)$ time.

Chapter 3

Set Reconciliation

In this chapter, we consider the set reconciliation protocol sketched in the introduction in greater detail. We define the protocol in section 3.1, prove its correctness in section 3.2, and do a complexity analysis in section 3.3. Section 3.4 lists some optimizations which do not change the asymptotic complexity but which avoid some unnecessary work. We conclude the chapter with an example application in section 3.5, briefly describing how the protocol can be applied to the synchronization of the hash graphs that arise e.g. in the context of distributed version control systems such as git [CS14].

3.1 Recursive Set Reconciliation

The set reconciliation protocol assumes that there is a set U, a linear order $\preceq$ on U, a node $\mathcal{X}_0$ locally holding some $X_0 \subseteq U$, and a node $\mathcal{X}_1$ locally holding $X_1 \subseteq U$. Both nodes are able to compute fingerprints for sets via some function $\text{fp} : \mathcal{P}(U) \to D$. This function could in principle be any hash function, but in the complexity analysis we will assume that it is instantiated as $\text{lift}_\text{f}^\mathcal{M}$ with a suitable underlying hash function f and a suitable monoid $\mathcal{M}$ over the universe D (see definition 6).

Recall that $[x, y)_S$ denotes the set $\{s \in S \mid x \preceq s \prec y\}$ if $x \prec y$, or $S \setminus [y, x)_S$ if $y \preceq x$ (definition 8).

$\mathcal{X}_0$ and $\mathcal{X}_1$ exchange messages over multiple rounds, a message consists of an arbitrary number of *interval fingerprints* and *interval item sets*. An interval fingerprint is a triple $(x, y, \text{fp}([x, y)_{X_i}))$ for $x, y \in U$, an interval item set a four-tuple (x, y, S, b) for $x, y \in U$, $S \subseteq [x, y)_{X_i}$, and $b \in \{0, 1\}$. b indicates whether the interval item set is a response to a previous interval item set.

When a node $\mathcal{X}_i$ receives a message, it performs the following actions:

- For every interval item set (x, y, S, b) in the message, all items in S are added to the locally stored set X_i. If $b = 0$, the node then adds the interval item set $(x, y, [x, y)_{X_i} \setminus S, 1)$ to the response, unless $[x, y)_{X_i} \setminus S = \emptyset$.

- For every interval fingerprint $(x, y, \text{fp}([x, y)_{X_j}))$ in the message, it does one of following:

  Case 1 (Equal Fingerprints). If $\text{fp}([x, y)_{X_j}) = \text{fp}([x, y)_{X_i})$, nothing happens.

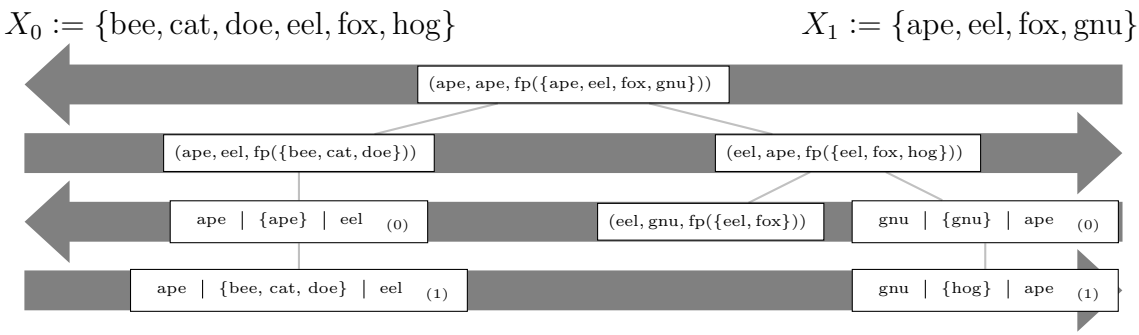

Figure 3.1: An example run of the protocol. $\mathcal{X}_1$ initiates reconciliation for all items between ape and ape(ordered alphabetically) by sending its fingerprint for the whole interval. Upon receiving this interval fingerprint, $\mathcal{X}_0$ locally computes $\text{fp}([\text{ape}, \text{ape})_{X_0})$. Since the result does not match the received interval, $\mathcal{X}_0$ splits X_0 into two parts of equal size and transmits interval fingerprints for these subintervals. In the third round, $\mathcal{X}_1$ locally computes fingerprints for the two received intervals, but neither matches. $|[\text{ape}, \text{eel})_{X_1}| \leq 1$, so $\mathcal{X}_1$ transmits the corresponding interval items set, i.e. $(\text{ape}, \text{eel}, \text{ape}, 0)$. $|[\text{eel}, \text{ape})_{X_1}| > 1$, so another recursion step is performed. After splitting the interval, the lower interval is large enough to send its fingerprint, the upper one however only contains one item and thus results in another interval item set. In the fourth and final communication round, $\mathcal{X}_0$ receives two interval item sets and answers with the items it holds within those intervals. When it receives the interval fingerprint $(\text{eel}, \text{gnu}, \text{fp}([\text{eel}, \text{gnu})_{X_1}))$, it computes an equal fingerprint for $(\text{eel}, \text{gnu}, \text{fp}([\text{eel}, \text{gnu})_{X_0}))$, so no further action is required for this particular interval. TODO prettify this caption

Case 2 (Recursion Anchor). The node may add the interval item set $(x, y, [x, y)_{X_i} 0)$ to the response. If $|[x, y)_{X_j}| \leq 1$, it must do so.

Case 3 (Recurse). Otherwise, the node selects $m_0 = x \prec m_1 \prec \ldots \prec m_k = y \in U$, $k \geq 2$ such that among all $[m_0, m_1)_{X_i}$ for $0 \leq l < k$ at least two intervals are non-empty. For all $0 \leq l < k$ it adds either the interval fingerprint $(m_l, m_{l+1}, \text{fp}([m_l, m_{l+1})_{X_i}))$ or the interval item set $(m_l, m_{l+1}, [m_l, m_{l+1})_{X_i} 0)$ to the response.

- If the accumulated response is nonempty, it is sent to the other node. Otherwise, the protocol has terminated successfully.

To initiate reconciliation of an interval $[x, y)$, a node $\mathcal{X}_i$ sends a message containing solely the interval fingerprint $(x, y, \text{fp}([x, y)_{X_i}))$.

Figure 3.1 gives an example run of the protocol.

3.1.1 Observations

Partitioning based on a total order allows the nodes to perform a limited form of queries, i.e. range queries. A node can ask for reconciliation within a certain interval, rather than over the whole universe.

If the universe U is not finite, then there are items that require an arbitrary amount of bytes to encode. Since the protocol needs to transmit items to denote interval boundaries, no reasonably complexity guarantees can be given for infinite

universes. We will thus assume U to be finite and small enough that items can be reasonably encoded. This assumption is not very restrictive in practice because nodes can always synchronize hashes of items rather than the items themselves. The protocol can then be either followed by a phase where hashes of interest are transferred and anwered by the actual items, or the protocol can be made aware of the distinction and use hashes as interval boundaries while transmitting actual items for interval item sets.

When reconciling hashes in place of actual items, any semantically interesting order on the items would be replaced by an arbitrary order on the hashes. But rather than using only's the hashes as interval boundaries, one can just as well add additional information. For example if the universe of interest consists of timestamped strings of arbitrary length, the interval boundaries can consist of timestamped hashes, ordered by timestamp first and using the numeric order on the hashes as a tiebreaker. Section 3.5 gives a more detailed example for utilizing this technique.

3.2 Proof of Correctness

We now prove the correctness of the protocol. The protocol is correct if for all $x, y \in U$ both nodes eventually hold $[x, y)_{X_i} \cup [x, y)_{X_j}$ after a node $\mathcal{X}_i$ has received a message pertaining to the interval $[x, y)$.

Case 1 (Interval Item Set). If the message contains the interval item set $(x, y, [x, y)_{X_j} 0)$, then $\mathcal{X}_i$ adds all items to its set, resulting in $[x, y)_{X_i} \cup [x, y)_{X_j}$ as desired. $\mathcal{X}_j$ then receives $(x, y, [x, y)_{X_i} \setminus [x, y)_{X_j}, 1)$, ending up with $[x, y)_{X_j} \cup ([x, y)_{X_i} \setminus [x, y)_{X_j}) = [x, y)_{X_i} \cup [x, y)_{X_j}$ as desired.

Case 2 (Interval Fingerprint). Otherwise, the message contains an interval fingerprint $(x, y, \mathrm{fp}([x, y)_{X_j}))$.

Case 2.1 (Equal Fingerprints). If $\mathrm{fp}([x, y)_{X_j}) = \mathrm{fp}([x, y)_{X_i})$, the protocol terminates immediately and no changes are performed by any node. Assuming no fingerprint collision occurred, $[x, y)_{X_i} = [x, y)_{X_j} = [x, y)_{X_i} \cup [x, y)_{X_j}$ as desired.

Case 2.2 (Recursion Anchor). If $\mathcal{X}_i$ adds the interval item set $(x, y, [x, y)_{X_i} 0)$, then case 1 applies when the other node receives the response, with the roles reversed.

Case 2.3 (Recurse). Let $count_i := |[x, y)_{X_i}|$ and $count_j := |[x, y)_{X_j}|$. $count_j \geq 2$, since otherwise $\mathcal{X}_j$ would have sent an item set for the interval. Similarly, $count_i \geq 2$, since we are not in case 2.2. Thus, $count_i + count_j \geq 4$, and the protocol has already been proven correct for all cases where $count_i + count_j < 4$.
We can thus finish the proof by induction on $count_i + count_j$, using the induction hypothesis that for all $x', y' \in U$ such that $|[x', y')_{X_i}| + |[x', y')_{X_j}| < count_i + count_j$ the protocol correctly reconciles $[x', y')_{X_i}$ and $[x', y')_{X_j}$.
$\mathcal{X}_i$ partitions the interval into $k \geq 2$ subintervals, of which at least two must be nonempty. Thus $|[m_l, m_{l+1})_{X_i}| < count_i$ for all $0 \leq l < k$. Furthermore, $[m_l, m_{l+1})_{X_j} \subseteq [x, y)_{X_j}$ and thus $|[m_l, m_{l+1})_{X_j}| \leq |[x, y)_{X_j}|$, so overall we have $|[m_l, m_{l+1})_{X_i}| + |[m_l, m_{l+1})_{X_j}| < count_i + count_j$ and can apply the induction hypothesis to conclude that every subinterval is correctly reconciled. Since the subintervals partition the original interval, the original interval is then correctly reconciled as well.

3.3 Complexity Analysis

The protocol gives nodes the freedom to respond to an interval fingerprint with an interval item set even if the interval fingerprint is arbitrarily large. For a meaningful complexity analysis we need to restrict the behavior of the node, a realistic modus operandi is for a node to send an interval item set whenever it holds a number of items less than or equal to some threshold $t \in \mathbb{N}, t \geq 1$ within the interval. Higher choices for t reduce the number of roundtrips, but increase the probability that a items is being sent even though the other node already holds it.

A node is similarly given freedom over the number of subintervals into which to split an interval when recursing. We will assume a node always splits into at most $b \in \mathbb{N}, b \geq 2$ subintervals. As with t, higher numbers reduce the number of roundtrips at the cost of potentially sending items or fingerprints that did not need sending.

Because we want to analyze not only the worst-case complexity but also the complexity depending on the similarity between the two sets held by the participating nodes, we define some rather fine-grained instance size parameters: n_0 and n_1 denote the number of items held by $\mathcal{X}_0$ and $\mathcal{X}_1$ respectively. We let $n := n_0+n_1$, $n_{min} := min(n_0, n_1)$, $n_{max} := max(n_0, n_1)$, $n_\cap$, $n_\cup$ and $n_\triangle := |([x, y)_{X_0} \cup [x, y)_{X_1}) \setminus ([x, y)_{X_0} \cap [x, y)_{X_1})|$. TODO remove those that are not needed

3.3.1 Preliminary Observations

A helpful observation for the following analysis is that the interval fingerprints that are being exchanged during a protocol run form a rooted tree where every vertex has at most b children. When a leaf of the tree is reached, an exchange of interval item sets follows. Equal fingerprints can also cut the tree short, but for the following worst-case analyses we will assume this does not occur.

Node $\mathcal{X}_i$ can branch at most $\lceil log_b(n_i) \rceil$ times, so the overall height of the tree is bounded by $2 \cdot \lceil log_b(n_{min}) \rceil$. The number of vertices of such a complete tree of height h is at most $\sum_{i=0}^{h} b^i = \frac{b^h-1}{b-1}$. For $h \leq 2 \cdot \lceil log_b(n_{min}) \rceil$, $\frac{b^h-1}{b-1} \leq 2 \cdot 2 \cdot n_{min} \leq 2n \in \mathcal{O}(n)$.

The parameter t determines when recursion is cut off, and thus influences the height of the tree. For $t = 1$, the protocol recurses as far as possible. For $t = b$, the last level of recursion is cut off, for $t = b^2$ the last two levels, and so on. Overall, the height of the tree is reduced by $\lfloor log_b(t) \rfloor$.

3.3.2 Communication Rounds

The number of communication rounds clearly corresponds to the height of the tree, plus 2 to account for the exchange of interval item sets, so the worst-case is $2 + 2 \cdot \lceil log_b(n_{min}) \rceil - \lfloor log_b(t) \rfloor \in \mathcal{O}(log_b(n))$. This number cannot be bounded by $n_\triangle$, as witnessed by problem instances where one node is missing exactly one item compared to the other node. In such an instance, $b - 1$ branches in each recursion step result in equal fingerprints, but the one branch that does continue reaches the recursion anchor only after the full number of rounds. See fig. 3.2 for an example.

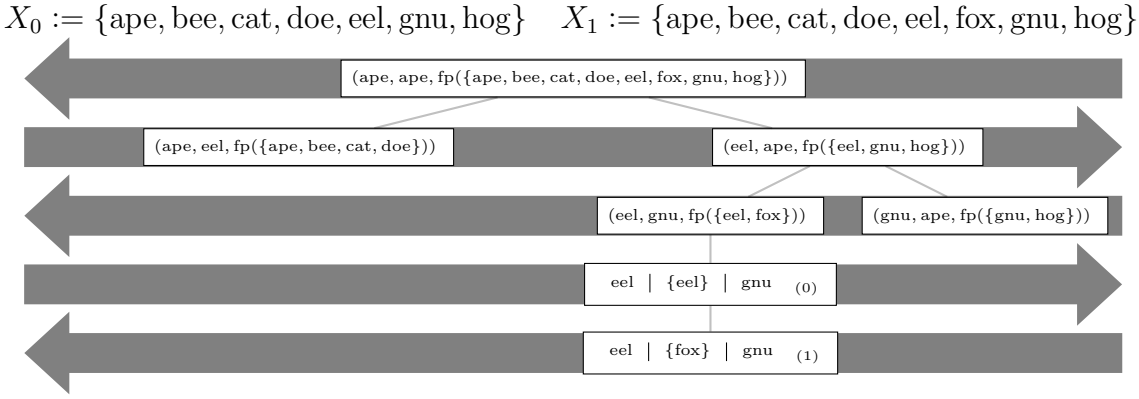

Figure 3.2: An example run of the protocol that takes the greatest possible number of rounds even though $n_\triangle = 1$. $b := 2, t := 1$.

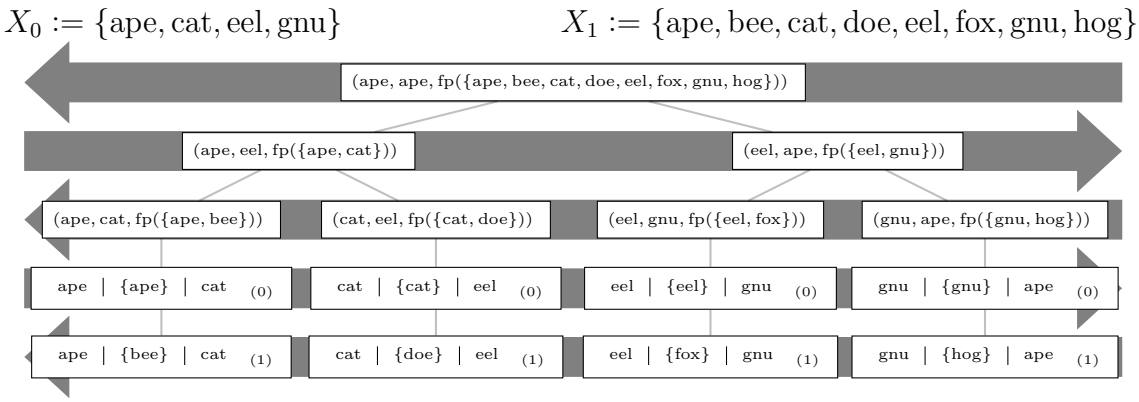

Figure 3.3: An example run of the protocol that requires transmitting the maximum amount of bytes. $b := 2, t := 1$.

3.3.3 Communication Complexity

The total number of bits that needs to be transmitted during a protocol run is proportional to the number of vertices in the tree. Every interval fingerprint consists of two items and one fingerprint, so assuming U is finite this lies in $\mathcal{O}(1)$. Since there are at most $2n$ vertices in the tree, the interval fingerprints require at most $\mathcal{O}(n)$ bits to be communicated.

The exchange of interval item sets consists in the worst case of exchanging every item using $\lceil \frac{n}{t} \rceil$ interval item sets. An interval item set needs to transmit two items to encode the boundaries, as well as the items themselves, which lies in $\mathcal{O}(1)$ per interval item set. All interval item sets together thus amount to another $\mathcal{O}(n)$, leading to a total of $\mathcal{O}(n)$ bits being transmitted in the worst case.

Figure 3.3 shows a worst-case example in which the tree of height $h := log_b(2 \cdot n_{min})$ has all $\frac{b^h-1}{b-1}$ vertices.

TODO bound complexity by difference, also average case

3.3.4 Impact of Randomized Intervals

TODO impact of boundary randomization

3.3.5 Computational Complexity

We now analyze the computational cost incurred by a single communication round, i.e. computing the response to a message. This includes both fingerprint comparisons as well as locating the items to transmit. We do however assume that an auxiliary data structure is available to help with this computation, e.g. the a fingerprint tree structure presented in **??**. We exclude both space usage and maintenance cost for this data structure from the per-round complexity.

We will assume that transferring an item as part of an interval item set requires $\mathcal{O}(1)$ time and space. The relevant computational overhead per communication round thus consists of computing $\text{fp}([x, y)_{X_i})$ for every received interval fingerprint $(x, y, \text{fp}([x, y)_{X_j}))$, as well as partitioning $[x, y)_{X_i}$ in case of a mismatch and computing the fingerprints over all subintervals. These computations can be performed independently for all received interval fingerprints, so in particular they can be performed sequentially, reusing space. The overall space complexity of the per-round computations is equal to that of the computations for a single interval fingerprint.

A naive approach is to query the auxiliary data structure for each received interval fingerprint individually. The maximum amount of queries is upper-bounded by n, it is certainly impossible to receive more than n interval fingerprints within a single communication round. Unfortunately, the greatest possible number of interval fingerprints within a single round corresponds to the number of leaves in the recursion tree, which is in $\mathcal{O}(n)$. The auxiliary data structure requires $\mathcal{O}(log(n))$ time, leading to an overall $\mathcal{O}(n \cdot log(n))$.

We can bring this down to $\mathcal{O}(log(n) + n_\triangle)$ by augmenting the auxiliary data structure with parent-pointers to allow efficient in-order traversal, and by memoizing some intermediate fingerprint computation results. TODO (write fingerprint chapter first)

3.4 Smaller Optimizations

We now give a list of optimizations which do not impact the overall complexity analysis, but which do improve on some constant factors.

3.4.1 Non-Uniform Partitions

When partitioning an interval into subintervals, the protocol does not specify where exactly to place the boundaries. Splitting into partitions of roughly equal sizes makes a lot of sense if new data could arise anywhere within the linear order with equal probability. Is however the items are likely to fall within certain ranges of the order, it can be more efficient to use more fine-grained partitions within those regions. If for example items are sorted by timestamp, and new items are expected to be propagated to every node in the system within a couple of seconds, then all items older than ten seconds can be comfortably lumped together in a large interval.

3.4.2 Subset Checks

When a node $\mathcal{X}_i$ receives an interval fingerprint $(x, y, \text{fp}([x, y)_{X_j}))$, it might have a different fingerprint for that interval, but one of the resulting subintervals could

match the received fingerprint. The node can then ignore that interval, transmitting only the remaining ones.

This scenario is a special case of receiving the fingerprint of a subset of the item one holds within an interval. In principle, a node can compute the fingerprints of arbitrary subsets of its interval, trying to find a match. If a match is found, the node knows that it holds a superset of the items the other node holds within the interval. The protocol could be extended with a message part that transmits items and does not warrant a response, this would be used to transmit $[x, y)_{X_i} \setminus [x, y)_{X_j}$.

Computing the fingerprint of all subsets of interval is infeasible since they are 2^n many subsets. As discussed in **??** however, if the group operation for fingerprint computation is xor, a node can check whether the fingerprint of a subset of items matches a given fingerprint in $\mathcal{O}(n^3)$. Assuming no fingerprint collisions occurred, checking whether a subset of $[x, y)_{X_i}$ matches $\text{fp}([x, y)_{X_j})$ is equivalent to checking whether $[x, y)_{X_i}$ is a superset of $[x, y)_{X_j}$.

This leads to a protocol with a per-round computational complexity of $\mathcal{O}(n^3)$ TODO n^4? and the same worst-case guarantees, but which can skip recursive steps more often than the basic protocol. In particular, whenever two nodes reconcile where one of them holds a subset of the items of the other node, reconciliation terminates after at most three communication rounds.

TODO precomputation, see comment

3.4.3 Efficiently Encoding Intervals

A naive encoding of interval fingerprints and interval item sets would transmit both boundaries of every interval, transmitting $2n$ items if there are n intervals in a message. This can be brought down to $1 + n$ by utilizing that the lower boundary of all but the first interval it is also the upper boundary of the preceding one. An efficiently encoded message consists of the number of intervals it contains, followed by the lower boundary of the first interval, followed by pairs of interval information and the upper boundary of the interval that the information pertains to, where each *interval information* is either a fingerprint, a set of items, or a dummy value signaling that this part of the overall interval is already fully synchronized.

The knowledge that interval boundaries are transmitted in ascending order can be used for compression. As a simple example, if interval boundaries are natural numbers, the upper boundaries can be transmitted as the difference to the previous boundary. This way, smaller numbers are transmitted, which admit shorter encodings. The same argument applies for transmitting sets of items in sorted order. TODO point to literature

Note additionally that two adjacent interval item sets can be merged into a single one, saving on the transmission of the boundary between them.

3.4.4 Utilizing Interval Boundaries

Whenever the lower boundary of an interval is transmitted, an actual item is transmitted. If the receiving node knew whether the other node held this item, it would automatically be reconciled and could be excluded from further recursion steps. One way for achieving this is to tag each interval with a bit to indicate whether the lower boundary is being held by the sender. A different approach is to require that inter-

vals are only split at items which the splitting node holds, this way no explicit bits needs to be transmitted, yet many boundaries can be identified as being held by the other node.

Recall that we assume U to be finite. For $x \in U$ we can thus denote by successor(x) the necessarily unique $z \in U$ such that $x \prec z$ and there exists no $y \in U$ such that $x \prec y \prec z$ if a z with $u \prec z$ exists, or $\min(U)$ otherwise. When sending or receiving an interval from x to y for which both nodes know that the sending node holds x, both nodes act as if the interval went from successor(x) to y, except that the receiving node adds x to its local set of items.

3.4.5 Multi-Level Fingerprints

Since the protocol interprets equal interval fingerprints as both nodes holding the same set of items within the interval, but never verifies this merely probabilistic assumption, fingerprints need to be long enough to guarantee low collision probability. The longer the fingerprints, the more bits need to be transmitted however.

One can use smaller fingerprints if in case of equal fingerprints the nodes then exchange an additional fingerprint rather than immediately stopping the recursion for this interval. Whenever a node receives a first-level interval fingerprint that is equal to its own, it answers with the second-level fingerprint for the same interval. When a node receives a second-level interval fingerprint that is equal to its own, it terminates the recursion. If it is not equal, it recurses as usual, in particular it sends first-level fingerprints for any large subintervals.

This scheme can of course be extended to an arbitrary number of fingerprint levels. Every level increases the worst-case number of roundtrips by one, but decreases the average message size. If cryptographically secure fingerprints are desired, low levels of the fingerprint hierarchy do not have to be secure, only the top level fingerprint needs to be so. Alternatively, each fingerprint level can consist of some substring of a long, cryptographically secure fingerprint, such that the concatenation of these substrings yields the original fingerprint.

3.5 Reconciling Hash Graphs

In this section, we demonstrate how to apply the set reconciliation protocol the problem of reconciling hash graphs. Hash graphs arise in contexts where pieces of data refer to other pieces of data by a secure hash computed over the data to be referred to. Distributed version control systems such as git [CS14] for example represent the evolving contents of a directory as a set of deltas which describe how the contents changed from an earlier version, this earlier version is referenced as the hash of another such object. Some objects describe how to merge conflicting concurrent changes, these objects can reference multiple other objects. Since addressing uses a secure hash function, an object can only reference objects which existed prior to it. All in all, objects can thus form arbitrary directed, acyclic graphs (dags).

When two nodes wish to synchronize, they update each other's histories to the union of all objects known to both of them. This is a natural setting for the set reconciliation protocol. Since objects can become arbitrarily large, one would reconcile sets of hashes of these objects, and then a second stage request the actual objects for all newly obtained hashes.

This approach completely ignores the edges of the dag. While at first glance it might seem inefficient to ignore available structural information, the fact that the dags can take arbitrary shapes makes it hard to utilize the edge structure. The graphs might be dense or sparse, could contain arbitrarily large independent sets, paths, tournaments etc.

Even though arbitrary dags are possible and thus the logarithmically bounded worst-case complexity is important, one can in practice make some reasonable assumptions about the hash graphs arising from an average, version-controlled repository. It seems likely for example that most concurrent new work is performed relative to a rather recent state of the repository, whereas work based off a very old state is rather unlikely. We could hope for better average reconciliation times if the reconciliation algorithm leveraged this expectation.

To that end, rather than reconciling merely hashes of objects, we can reconcile pairs $(\mathit{depth}, \mathit{hash})$, sorting by depth first and hash second. The *depth* of an object is the length of the longest path from that object to a root object, i.e. an object without predecessors. That is, the depth of an object is 1 greater than the greatest depth of any predecessor object. If most concurrent work is based off similarly new state, then it also falls into a similar interval of the linear order. A protocol run does not need to recurse into intervals of low depth then, since no concurrently working peer produces new objects of low depth. If this assumption turns out to be wrong, the protocol nevertheless upholds its worst-case guarantee of a logarithmic number of communication rounds.

Another common scenario is that a node has not produced any new local objects and merely wants to catch up with the current state. Let d be the maximal depth among all objects the node holds. Then rather than reconciling the interval $[(0,0), \top)$, the node can reconcile the interval $[(0,0), (d+1,0))$ as well as sending an empty interval item set for the interval $[(d+1,0), \top)$. All new changes of depth $d+1$ or greater are then fetched in only two communication rounds. Any concurrent work (from a depth perspective) is reconciled as usual, and if all new changes are based off a recent version, the whole reconciliation process only takes a single round trip.

Chapter 4

Other Data Structures

4.1 Higher-Dimensional Intervals

k-d-trees

4.2 Maps

two reconciliation sessions in parallel, one for the keys and one for the values, but both ordered by the keys

4.3 Tries

optimizing lexicographically ordered items

4.4 Sequences

Why we need content-based slicing, even recursively
    sequences as maps from rational numbers to items

Chapter 5

Related Work

- set reconciliation literature
- hash graph synchronization
- filesystem synchronization
- history-based synchronization
- authenticated data structures
- hashing sets (pugh, tarjan...)

Chapter 6

Conclusion

TODO: conclude things

Work Plan

- by 05.05: basic set reconciliation chapter
- by 26.05: fingerprint chapter
- by 16.06: bounded-memory set reconciliation chapter
- by 07.07: other data structures chapter
- by 28.07: conclusion, coherence, polishing
- 15.08: self-inflicted soft deadline, unless adding more content

Possibly a chapter discussing more specifics that would occur when using set reconciliation as the core of an unordered p2p pubsub mechanism.

Bibliography

[BBJ+16] Michael A Bender, Jonathan W Berry, Rob Johnson, Thomas M Kroeger, Samuel McCauley, Cynthia A Phillips, Bertrand Simon, Shikha Singh, and David Zage. Anti-persistence on persistent storage: History-independent sparse tables and dictionaries. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 289–302, 2016.

[BLL02] Ahto Buldas, Peeter Laud, and Helger Lipmaa. Eliminating counterevidence with applications to accountable certificate management 1. *Journal of Computer Security*, 10(3):273–296, 2002.

[BM97] Mihir Bellare and Daniele Micciancio. A new paradigm for collision-free hashing: Incrementality at reduced cost. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 163–192. Springer, 1997.

[BM02] Rudolf Bayer and Edward McCreight. Organization and maintenance of large ordered indexes. In *Software pioneers*, pages 245–262. Springer, 2002.

[BSV17] Lisa Bromberg, Vladimir Shpilrain, and Alina Vdovina. Navigating in the cayley graph of $sl_2(\mathbb{F}_p)$ and applications to hashing. In *Semigroup Forum*, volume 94, pages 314–324. Springer, 2017.

[CDVD+03] Dwaine Clarke, Srinivas Devadas, Marten Van Dijk, Blaise Gassend, and G Edward Suh. Incremental multiset hash functions and their application to memory integrity checking. In *International conference on the theory and application of cryptology and information security*, pages 188–207. Springer, 2003.

[CNQ09] Julien Cathalo, David Naccache, and Jean-Jacques Quisquater. Comparing with rsa. In *IMA International Conference on Cryptography and Coding*, pages 326–335. Springer, 2009.

[CS14] Scott Chacon and Ben Straub. *Pro git*. Springer Nature, 2014.

[Dwo15] Morris J Dworkin. Sha-3 standard: Permutation-based hash and extendable-output functions. 2015.

[Fen94] Peter M Fenwick. A new data structure for cumulative frequency tables. *Software: Practice and experience*, 24(3):327–336, 1994.

[GIMS11] Markus Grassl, Ivana Ilić, Spyros Magliveras, and Rainer Steinwandt. Cryptanalysis of the tillich–zúmor hash function. *Journal of cryptology*, 24(1):148–156, 2011.

[Gol09] Daniel Golovin. B-treaps: A uniquely represented alternative to b-trees. In *International Colloquium on Automata, Languages, and Programming*, pages 487–499. Springer, 2009.

[HHM+05] Jason D Hartline, Edwin S Hong, Alexander E Mohr, William R Pentney, and Emily C Rocke. Characterizing history independent data structures. *Algorithmica*, 42(1):57–74, 2005.

[Jon03] Simon Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[LKMW19] Kevin Lewi, Wonho Kim, Ilya Maykov, and Stephen Weis. Securing update propagation with homomorphic hashing. *IACR Cryptol. ePrint Arch.*, 2019:227, 2019.

[Lyu05] Vadim Lyubashevsky. The parity problem in the presence of noise, decoding random linear codes, and the subset sum problem. In *Approximation, randomization and combinatorial optimization. Algorithms and techniques*, pages 378–389. Springer, 2005.

[MEL+21] Krzysztof Maziarz, Tom Ellis, Alan Lawrence, Andrew Fitzgibbon, and Simon Peyton Jones. Hashing modulo alpha-equivalence. *arXiv preprint arXiv:2105.02856*, 2021.

[Mer89] Ralph C Merkle. A certified digital signature. In *Conference on the Theory and Application of Cryptology*, pages 218–238. Springer, 1989.

[MGS15] Hristina Mihajloska, Danilo Gligoroski, and Simona Samardjiska. Reviving the idea of incremental cryptography for the zettabyte era use case: Incremental hash functions based on sha-3. In *International Workshop on Open Problems in Network Security*, pages 97–111. Springer, 2015.

[MND+04] Charles Martel, Glen Nuckolls, Premkumar Devanbu, Michael Gertz, April Kwong, and Stuart G Stubblebine. A general model for authenticated data structures. *Algorithmica*, 39(1):21–41, 2004.

[MSTA17] Jeremy Maitin-Shepard, Mehdi Tibouchi, and Diego F Aranha. Elliptic curve multiset hash. *The Computer Journal*, 60(4):476–490, 2017.

[MT16] Ciaran Mullan and Boaz Tsaban. sl_2 homomorphic hash functions: worst case to average case reduction and short collision search. *Designs, Codes and Cryptography*, 81(1):83–107, 2016.

[MVOV18] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 2018.

[NN00] Moni Naor and Kobbi Nissim. Certificate revocation and certificate update. *IEEE Journal on selected areas in communications*, 18(4):561–570, 2000.

[Pet09] Christophe Petit. *On graph-based cryptographic hash functions.* PhD thesis, Catholic University of Louvain, Louvain-la-Neuve, Belgium, 2009.

[PQ10] Christophe Petit and Jean-Jacques Quisquater. Preimages for the tillich-zémor hash function. In *International Workshop on Selected Areas in Cryptography*, pages 282–301. Springer, 2010.

[PQ+11] Christophe Petit, Jean-Jacques Quisquater, et al. Rubik's for cryptographers. *IACR Cryptol. ePrint Arch.*, 2011:638, 2011.

[PT89] William Pugh and Tim Teitelbaum. Incremental computation via function caching. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 315–328, 1989.

[Pug90] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[Pug98] William Pugh. A skip list cookbook. Technical report, 1998.

[SA96] Raimund Seidel and Cecilia R Aragon. Randomized search trees. *Algorithmica*, 16(4):464–497, 1996.

[SMBA10] Paul T Stanton, Benjamin McKeown, Randal Burns, and Giuseppe Ateniese. Fastad: an authenticated directory for billions of objects. *ACM SIGOPS Operating Systems Review*, 44(1):45–49, 2010.

[Sny77] L. Snyder. On uniquely represented data strauctures. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, pages 142–146, Los Alamitos, CA, USA, oct 1977. IEEE Computer Society. URL: `https://doi.ieeecomputersociety.org/10.1109/SFCS.1977.22`, `doi:10.1109/SFCS.1977.22`.

[Sos16] Bianca Sosnovski. Cayley graphs of semigroups and applications to hashing. 2016.

[ST94] Rajamani Sundar and Robert E Tarjan. Unique binary-search-tree representations and equality testing of sets and sequences. *SIAM Journal on Computing*, 23(1):24–44, 1994.

[TLMT19] Dominic Tarr, Erick Lavoie, Aljoscha Meyer, and Christian Tschudin. Secure scuttlebutt: An identity-centric protocol for subjective and decentralized applications. In *Proceedings of the 6th ACM Conference on Information-Centric Networking*, pages 1–11, 2019.

[TM+96] Andrew Tridgell, Paul Mackerras, et al. The rsync algorithm. 1996.

[TZ94] Jean-Pierre Tillich and Gilles Zémor. Hashing with sl 2. In *Annual International Cryptology Conference*, pages 40–49. Springer, 1994.

[Wag02] David Wagner. A generalized birthday problem. In *Annual International Cryptology Conference*, pages 288–304. Springer, 2002.

[Zém91] Gilles Zémor. Hash functions and graphs with large girths. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 508–511. Springer, 1991.