Efficient Synchronization of Recursively Partitionable Data Structures

Technische Universität Berlin

Aljoscha Meyer

May 18, 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 has been fully reconciled and the protocol terminates.
- $\mathcal{B}$ 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 described in this thesis work by computing fingerprints of sets. This chapter defines and motivates a specific fingerprinting scheme that admits fast computation with small overhead for the storage and maintenance of auxiliary data structures. section 2.1 outlines the solution space and theoretic bounds. We examine randomized solutions that with high probability compute fingerprints in logarithmic time in section 2.2. Section 2.3 characterizes a family of functions that admit efficient incremental computation, and Section 2.4 proposes members of this family that can be used for fingerprinting.

2.1 Initial Considerations

The protocols explored in this thesis 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 $\frac{1}{|D|}$. $\mathrm{h}(u)$ 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 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 $\mathit{height}(T) \leq \lceil \varepsilon \cdot \mathit{log}_2(|V|) \rceil$. 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 $\mathcal{O}(\mathit{log}(n))$. 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 [NN98], 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: utilizing randomization to define a unique tree layout which allows logarithmic operations with high probability, which we examine in section 2.2, or letting the fingerprint function abstract over the tree shape, downgrading it to an implementation detail, which we examine in section 2.3 and beyond.

2.2 Pseudorandom Data Structures

TODO: hash tries, possibly skip lists

2.3 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 $\mathrm{lift}_{\mathrm{f}}^{\mathcal{M}} : \mathcal{P}(U) \to M$ with:

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

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* $\mathrm{label}_{\mathrm{f}}^{\mathcal{M}} : S \to M$:

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

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 $\mathrm{label}_{\mathrm{f}}^{\mathcal{M}}(r) = \mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}(S)$.

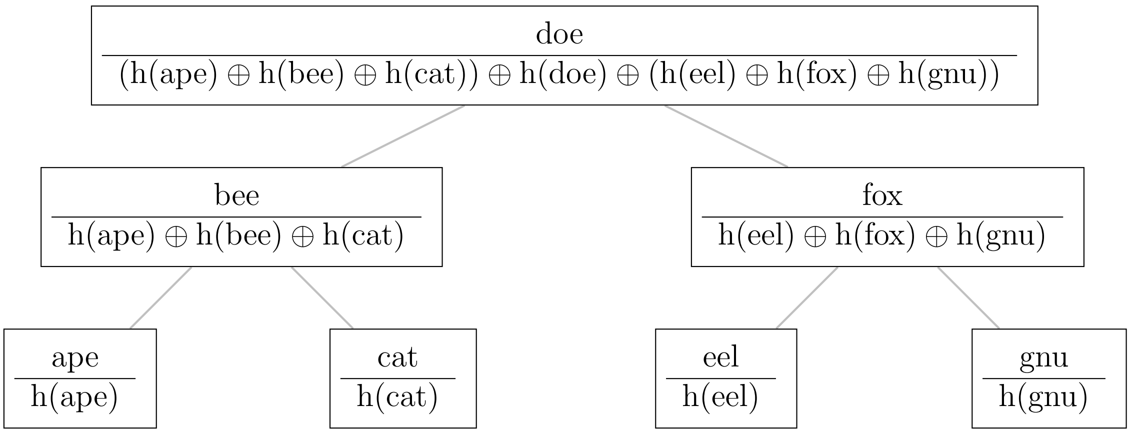


Figure 2.1: A balanced search tree labeled by $\text{label}_{\text{h}}^{(\text{M},\oplus,\mathbb{0})}$. For fingerprinting, h could be a hash function and $\oplus$ the xor operation on fixed-width bitstrings.

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

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

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:

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

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 $\mathbb{0}$.

2.3.1 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 $\{s \in S | x \preceq s \prec y\}$. We call x the *lower boundary* and y the *upper boundary* of the interval.

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

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.

Listing 1 gives a precise definition of how to compute $\mathrm{lift}_\mathrm{f}^\mathcal{M}([x, y)_S)$, for clarity of presentation only complete binary trees are considered. Arbitrary binary trees can also have inner nodes with exactly one child, almost the same algorithm can be used, acting as if these nodes also had a second child labeled with $\mathbb{0}$.

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. If there is no such v, then $[x, y)_S = \emptyset$ and thus $\mathrm{lift}_\mathrm{f}^\mathcal{M}([x, y)_S) = \mathbb{0}$. All vertices of T that are not vertices 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. Thus, $[x, y)_S = [x, \max(\mathrm{V}(T_{c_<})))_{\mathrm{V}(T_{c_<})} \dot\cup \{v\} \dot\cup [\min(\mathrm{V}(T_{c_>})), y)_{\mathrm{V}(T_{c_>})}$ and $\mathrm{lift}_\mathrm{f}^\mathcal{M}([x, y)_S) = \mathrm{lift}_\mathrm{f}^\mathcal{M}([x, \max(\mathrm{V}(T_{c_<})))_{\mathrm{V}(T_{c_<})}) \oplus \mathrm{f}(v) \oplus \mathrm{lift}_\mathrm{f}^\mathcal{M}([\min(\mathrm{V}(T_{c_>})), y)_{\mathrm{V}(T_{c_>})})$.

Proving that `sumGeq` from listing 1 does indeed sum over all $\mathrm{f}(v)$ in the given tree with $x \preceq v$, i.e. computes $\mathrm{lift}_\mathrm{f}^\mathcal{M}([x, \max(\mathrm{V}(T_{c_<})))_{\mathrm{V}(T_{c_<})})$ can be done by a rather technical but straightforward induction which we omit, same for `sumLt` computing $\mathrm{lift}_\mathrm{f}^\mathcal{M}([\min(\mathrm{V}(T_{c_>})), y)_{\mathrm{V}(T_{c_>})})$. From those facts, correctness of `intervalFingerprintGroup` follows from the previous arguments.

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` do a traversal to a leaf. 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 $\mathcal{O}(\log(|S|))$.

A slightly simpler approach 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 $fp[x, y)_S = -\mathrm{fp}([\min(S), x)_S) \oplus \mathrm{fp}([\min(S), y)_S)$. Reusing the definitions from listing 1, we get:

```
intervalFingerprintGroup :: Node -> U -> U -> D
intervalFingerprintGroup v x y = -(sumLt v x) + (sumLt v y)
```

Alternatively, a slightly more efficient version that still does not require `sumGeq`:

```
intervalFingerprint :: Node -> U -> U -> D
intervalFingerprint v x y = case findInitial v x y of
    Nothing               -> 0
    Just (Leaf _ fp)      -> fp
    Just (Inner l v' r _) -> -(sumLt l x) + (f v') + (sumLt r y)
```

2.4 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 of 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.

TODO read more papers on this, then continue writing with proper references

```haskell
-- U is the type of items, D the type of fingerprints.
-- A node is either a leaf or an inner vertex.
-- Both store a fingerprint as a label.
data Node = Leaf U D | Inner Node U Node D

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

-- Compute the fingerprint over all items stored in v
-- within the interval [x, y)_S.
intervalFingerprint :: Node -> U -> U -> D
intervalFingerprint v x y = case findInitial v x y of
    Nothing               -> 0
    Just (Leaf _ fp)      -> fp
    Just (Inner l v' r _) -> (sumGeq l x) + (f v') + (sumLt r y)

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

-- Sum up the fingerprints of all items in the given tree
-- which are greater than or equal to x.
sumGeq :: Node -> U -> D
sumGeq (Leaf v fp) x
    | v < x     = 0
    | otherwise = fp
sumGeq (Inner l v r _) x
    | v < x     = sumGeq r x
    | otherwise = (sumGeq l x) + (f v) + (label r)

-- Sum up the fingerprints of all items in the given tree
-- which are strictly less than y.
sumLt :: Node -> U -> D
sumLt (Leaf v fp) y
    | v >= y    = 0
    | otherwise = fp
sumLt (Inner l v r _) y
    | v >= y    = sumLt l y
    | otherwise = (label l) + (f v) + (sumLt r y)
```

Listing 1: Computing $[x, y)_S$ from the complete labeled search tree of S.

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 $X_1 \subseteq U$. $\mathcal{X}_0$ and $\mathcal{X}_1$ exchange messages, a message consists of an arbitrary number of *interval fingerprints* and *interval item set*. An interval fingerprint is a triple $(x, y, \mathrm{fp}([x, y)_{X_i}))$ for $x, y \in U$, an interval item set a four-tuple $(x, y, S)_b$ for $x, y \in U, S \subseteq [x, y)_{X_i}, b \in \{0, 1\}$. b indicates whether the interval item set is a response to a previous interval item set.

Recall that $\mathrm{fp}(A)$ denotes the fingerprint for $A \subseteq U$, and that $[x, y)_A := \{a \in A | x \preceq a \prec y\}$.

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

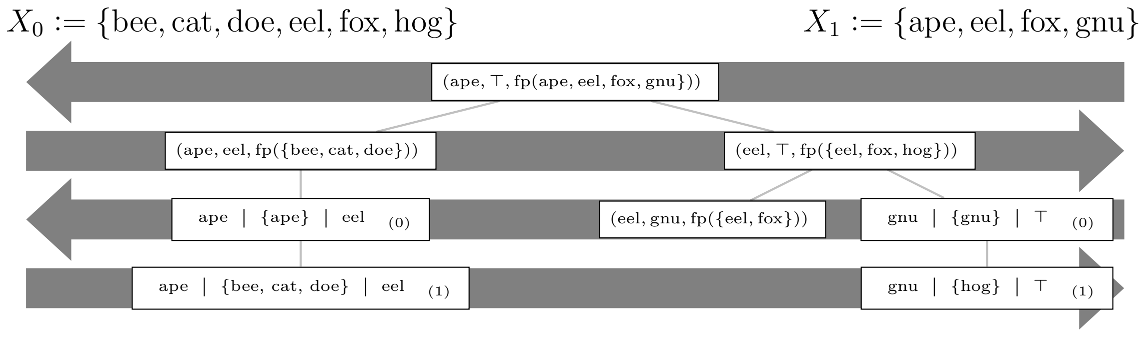
- For every interval item set $(x, y, S)_b$ in the message, all items in S are added to the locally stored set X_i. If $b = 0$, the node then adds the interval item set $(x, y, [x, y)_{X_i} \setminus S)_1$ to the response, unless $[x, y)_{X_i} \setminus S = \emptyset$.

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

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

  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



X0 := {bee, cat, doe, eel, fox, hog}
X1 := {ape, eel, fox, gnu}
(ape, ⊤, fp(ape, eel, fox, gnu}))
(ape, eel, fp({bee, cat, doe}))
(eel, ⊤, fp({eel, fox, hog}))
ape | {ape} | eel (0)
(eel, gnu, fp({eel, fox}))
gnu | {gnu} | ⊤ (0)
ape | {bee, cat, doe} | eel (1)
gnu | {hog} | ⊤ (1)

Figure 3.1: An example run of the protocol. $\mathcal{X}_1$ initiates reconciliation for all items between ape and $\top$(ordered alphabetically) by sending its fingerprint for the whole interval. Upon receiving this interval fingerprint, $\mathcal{X}_0$ locally computes $\text{fp}([\text{ape}, \top)_{X_0})$. 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. $(\text{ape}, \text{eel}, \text{ape})_0$. $|[\text{eel}, \top)_{X_1}| > 1$, 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

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 finite, the greatest element of the universe cannot be exchanged, since all ranges have an exclusive upper boundary. We will thus assume that for a universe U of interest, nodes are actually using $\tilde{U} := U \dot{\cup} \top$ with $u \preceq \top$ for all $u \in U$.

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.

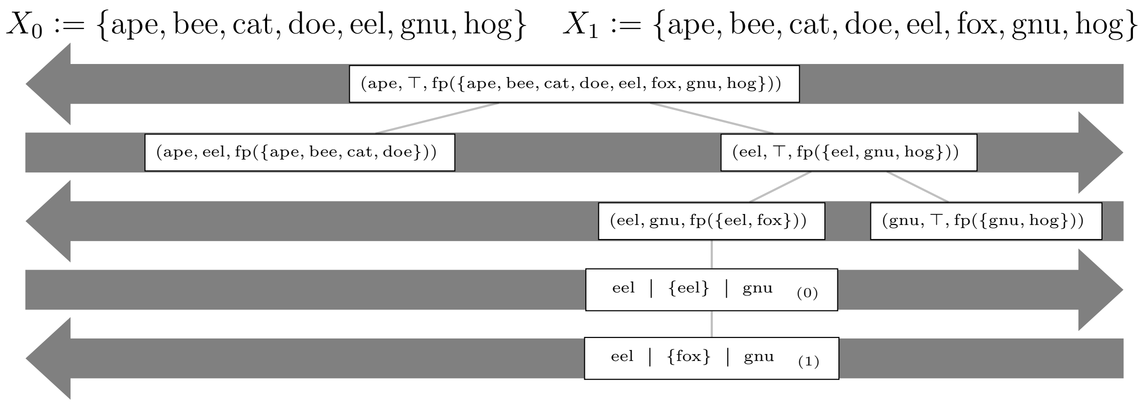

$X_0 := \{\text{ape}, \text{bee}, \text{cat}, \text{doe}, \text{eel}, \text{gnu}, \text{hog}\}$
$X_1 := \{\text{ape}, \text{bee}, \text{cat}, \text{doe}, \text{eel}, \text{fox}, \text{gnu}, \text{hog}\}$
(ape, ⊤, fp({ape, bee, cat, doe, eel, fox, gnu, hog}))
(ape, eel, fp({ape, bee, cat, doe}))
(eel, ⊤, fp({eel, gnu, hog}))
(eel, gnu, fp({eel, fox}))
(gnu, ⊤, fp({gnu, hog}))
eel | {eel} | gnu (0)
eel | {fox} | gnu (1)

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

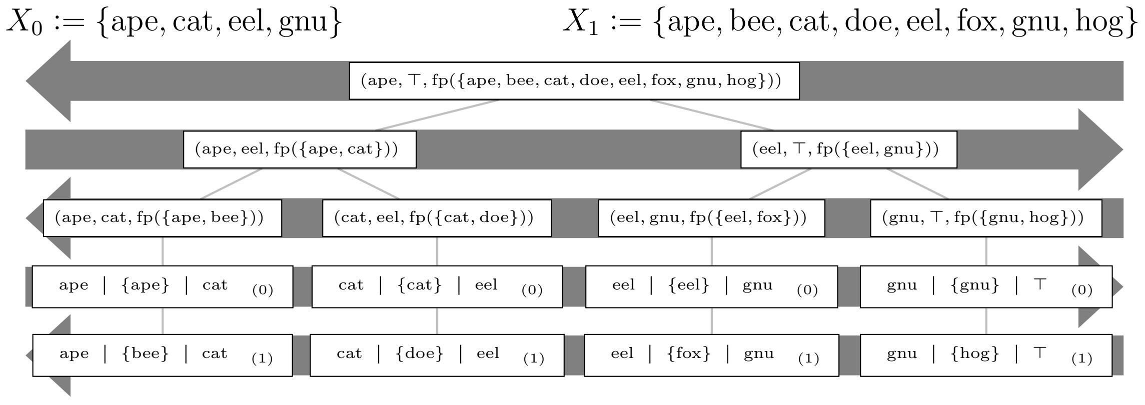

$X_0 := \{\text{ape}, \text{cat}, \text{eel}, \text{gnu}\}$
$X_1 := \{\text{ape}, \text{bee}, \text{cat}, \text{doe}, \text{eel}, \text{fox}, \text{gnu}, \text{hog}\}$
(ape, ⊤, fp({ape, bee, cat, doe, eel, fox, gnu, hog}))
(ape, eel, fp({ape, cat}))
(eel, ⊤, fp({eel, gnu}))
(ape, cat, fp({ape, bee}))
(cat, eel, fp({cat, doe}))
(eel, gnu, fp({eel, fox}))
(gnu, ⊤, fp({gnu, hog}))
ape | {ape} | cat (0)
cat | {cat} | eel (0)
eel | {eel} | gnu (0)
gnu | {gnu} | ⊤ (0)
ape | {bee} | cat (1)
cat | {doe} | eel (1)
eel | {fox} | gnu (1)
gnu | {hog} | ⊤ (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 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 $\mathrm{fp}([x,y)_{X_i})$ for every received interval fingerprint $(x, y, \mathrm{fp}([x,y)_{X_j}))$, 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 $(x, y, \mathrm{fp}([x,y)_{X_j}))$, 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 $\mathrm{fp}([x, y)_{X_j})$ 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 $\mathcal{O}(n^3)$ 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 TODO: this is O(1), not polynomial?

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.

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 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 intervals 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 and with the greatest element $\top$, which never occurs as the lower boundary of an interval. For $x \in U \setminus \{\top\}$ we can thus denote by successor(x) the unique $z \in U$ such that $x \prec z$ and there exists no $y \in U$ such that $x \prec y \prec z$. 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 from $(0, 0)$ to $\top$, the node can reconcile the interval from $(0, 0)$ to $(d + 1, 0)$ as well as sending an empty interval item set from $(d + 1, 0)$ to $\top$. 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

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

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

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

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

[NN98] Kobbi Nissim and Moni Naor. Certificate revocation and certificate update. In *USENIX Security Symposium.* Citeseer, 1998.

[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`.

[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.