Efficient Synchronization of Recursively Partitionable Data Structures

Technische Universität Berlin

Aljoscha Meyer

April 28, 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. Chapter 4 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 5 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 6 gives an overview of related work and justifies the chosen approach. We conclude in chapter 7.

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.

- Merkle trees and why they don't cut it
- hashing into a group and adding things, using search trees for efficient computation
- why not monoids instead of groups
- fingerprint collisions (with and without help from malicious peers)
- miscellaneous (also nice for putting data structures into hash tables, intervals as primitive queries, progress over unreliable links)

TODO: move the following definitions to where they are needed

Definition 1. 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 2. Let U be a set, $\preceq$ a linear order on U, and $A \subseteq U$. A *binary search tree on A* is a rooted tree T with vertex set A such that for any inner vertex p with left child a and right child b: $a \prec p \prec b$.

Definition 3. 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.

Definition 4. Let U be a set, $\oplus : U \times U \to U$, and $\mathbb{0} \in U$. We call $(U, \oplus, \mathbb{0})$ a *monoid* if it satisfies two properties:

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

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

Definition 5. Let $(U, \oplus, \mathbb{0})$ be a monoid. We call it a *transitive monoid* if for all $x, z \in U$ there exists $y \in U$ such that $x \oplus y = z$.

Definition 6. Let $(U, \oplus, \mathbb{0})$ be a (transitive) monoid. We call it a *(transitive) group* if for all $x \in U$ there exists $y \in U$ such that $x \oplus y = \mathbb{0}$. This y is necessarily unique and denoted by $-x$. For $x, y \in U$ we write $x \ominus y$ as a shorthand for $x \oplus -y$.

Chapter 3

Set Reconciliation

In this chapter, we consider the set reconciliation protocol sketched in the introduction in greater detail. We define an unoptimized but simple version of the protocol in section 3.1, and we prove its correctness in section 3.2. Section 3.3 lists optimizations which eliminate unnecessary work from the protocol. We then define the proper set reconciliation protocol in section 3.4 and do a complexity analysis in section 3.5. We conclude the chapter with an example application in section 3.6, 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 Simple 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 *items*. An interval fingerprint is a triple $(x, y, \mathit{fp}([x, y)_{X_i}))$ for $x, y \in U$, an item is simply some $x \in U$.

Recall that $\mathit{fp}(A)$ 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 item in the message, the item is added to the locally stored set X_i.
- For every interval fingerprint $(x, y, \mathit{fp}([x, y)_{X_j}))$ in the message, it does one of following:

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

  Case 2 (Receiving Empty). If $\mathit{fp}([x, y)_{X_j}) = \mathbb{0}$, it adds all items in $[x, y)_{X_i}$ to the response.

  Case 3 (Sending Empty). If $\mathit{fp}([x, y)_{X_j}) \neq \mathbb{0} = \mathit{fp}([x, y)_{X_i})$, it adds the interval fingerprint $(x, y, \mathit{fp}([x, y)_{X_i})) = (x, y, \mathbb{0})$ to the response.

  Case 4 (Recursive). Otherwise, it finds some middle item $m \in X_i$ that equally partitions the items of X_i within the interval, i.e. m is chosen such that $-1 \leq |\{a \in X_i | x \preceq a \prec m\}| - |\{b \in X_i | m \preceq b \prec y\}| \leq 0$. It then adds the interval fingerprints $(x, m, \mathit{fp}([x, m)_{X_i}))$ and $(m, y, \mathit{fp}([m, y)_{X_i}))$ to the response.

- If the accumulated response is nonempty, it is sent to the other node.

To initiate reconciliation of an interval, a node sends a message consisting solely of its interval fingerprint of the interval to reconcile.

Figure 3.1 gives an example run of the protocol.

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 $[x, y)_{X_0} \cup [x, y)_{X_1}$ after one node $\mathcal{X}_i$ has sent an interval fingerprint $(x, y, \mathit{fp}([x, y)_{X_i}))$.

The proof is necessarily rather technical, but conceptually correctness follows rather straightforwardly by induction from the fact that two sets can be reconciled by individually reconciling their partitions:

Proposition 1. Let $S = \biguplus_{i \in \mathcal{I}} S_i, T = \biguplus_{i \in \mathcal{I}} T_i$, then $S \cup T = \biguplus_{i \in \mathcal{I}} (S_i \cup T_i)$.

Without loss of generality we consider the case where $\mathcal{X}_0$ has sent the interval fingerprint $(x, y, \mathit{fp}([x, y)_{X_0}))$. Let $\mathit{count}_0 := |[x, y)_{X_0}|$ and $\mathit{count}_1 := |[x, y)_{X_1}|$. We prove the statement by induction on $\mathit{count}_0 + \mathit{count}_1$. There are four base cases:

Case 1 (Equal Fingerprints). If $\mathit{fp}([x, y)_{X_0}) = \mathit{fp}([x, y)_{X_1})$, then the protocol terminates immediately and no changes are performed by any node. Assuming no fingerprint collision occurred, $[x, y)_{X_0} = [x, y)_{X_1} = [x, y)_{X_0} \cup [x, y)_{X_1}$ as desired.

Case 2 (Receiving Empty). If $[x, y)_{X_0} = \emptyset$, its fingerprint is $\mathbb{0}$, and $\mathcal{X}_1$ sends all items in $[x, y)_{X_1}$. $\mathcal{X}_1$ does not modify the set it holds, so it ends up with $[x, y)_{X_1} = [x, y)_{X_1} \cup \emptyset = [x, y)_{X_1} \cup [x, y)_{X_0} = [x, y)_{X_0} \cup [x, y)_{X_1}$ as desired. $\mathcal{X}_0$ does not receive any interval fingerprint, so it does not send a response and the protocol terminates. It adds the received items to its local copy, so it ends up with $\emptyset \cup [x, y)_{X_1} = [x, y)_{X_0} \cup [x, y)_{X_1}$ as desired.

Case 3 (Sending Empty). If $[x, y)_{X_0} \neq \emptyset$ but $[x, y)_{X_1} = \emptyset$, then $\mathcal{X}_1$ responds to the interval fingerprint sent by $\mathcal{X}_0$ by sending the interval fingerprint for the same interval, which is necessarily $(x, y, \mathbb{0})$. Correctness then follows from case 2 with the roles reversed.

Case 4 (Two Singletons). If $\mathit{count}_0 = 1 = \mathit{count}_1$ but $\mathit{fp}([x, y)_{X_0}) \neq \mathit{fp}([x, y)_{X_1})$, let u_i be the one item held by $\mathcal{X}_i$ in the interval. $\mathcal{X}_1$ responds with the two interval fingerprints $(x, u_1, \mathit{fp}([x, u_1)_{X_1})) = (x, u_1, \mathbb{0})$ and $(u_1, y, \mathit{fp}([u_1, y)_{X_1})) = (u_1, y, \mathit{fp}(u_1))$. By proposition 1 we only need to show that these two intervals are being reconciled correctly. Correct reconciliation of $(x, u_1, \mathbb{0})$ is covered by case 2 with the roles reversed, it remains to show that $(u_1, y, \mathit{fp}(u_1))$ is reconciled correctly. Assuming no fingerprint collision occurred, $u_0 \neq u_1$.

Case 4.1. If $u_0 < u_1$, then $\mathit{fp}([u_1, y)_{X_0}) = \mathbb{0}$, so $\mathcal{X}_0$ receiving $(u_1, y, \mathit{fp}(u_1))$ is covered by case 3 with the roles reversed.

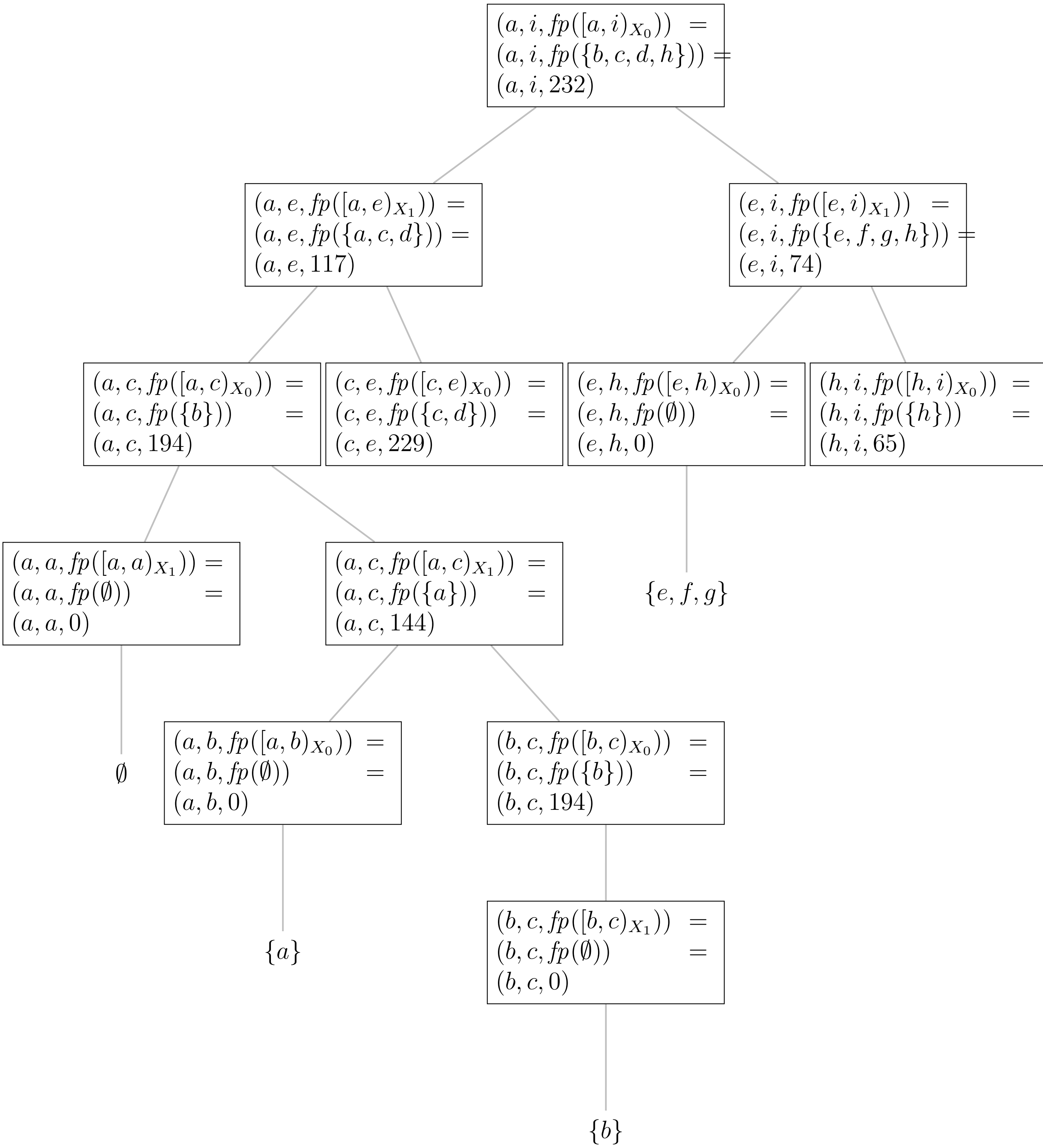
$(a, i, fp([a, i)_{X_0})) = (a, i, fp(\{b, c, d, h\})) = (a, i, 232)$
$(a, e, fp([a, e)_{X_1})) = (a, e, fp(\{a, c, d\})) = (a, e, 117)$
$(e, i, fp([e, i)_{X_1})) = (e, i, fp(\{e, f, g, h\})) = (e, i, 74)$
$(a, c, fp([a, c)_{X_0})) = (a, c, fp(\{b\})) = (a, c, 194)$
$(c, e, fp([c, e)_{X_0})) = (c, e, fp(\{c, d\})) = (c, e, 229)$
$(e, h, fp([e, h)_{X_0})) = (e, h, fp(\emptyset)) = (e, h, 0)$
$(h, i, fp([h, i)_{X_0})) = (h, i, fp(\{h\})) = (h, i, 65)$
$(a, a, fp([a, a)_{X_1})) = (a, a, fp(\emptyset)) = (a, a, 0)$
$(a, c, fp([a, c)_{X_1})) = (a, c, fp(\{a\})) = (a, c, 144)$
$\{e, f, g\}$
$\emptyset$
$(a, b, fp([a, b)_{X_0})) = (a, b, fp(\emptyset)) = (a, b, 0)$
$(b, c, fp([b, c)_{X_0})) = (b, c, fp(\{b\})) = (b, c, 194)$
$\{a\}$
$(b, c, fp([b, c)_{X_1})) = (b, c, fp(\emptyset)) = (b, c, 0)$
$\{b\}$

Figure 3.1: TODO WIP, ignore for now

Case 4.2. So assume that $u_0 > u_1$. Then $\mathcal{X}_0$ receiving $(u_1, y, fp(u_1))$ is again case 4, with the roles reversed. This time however, the initiating node holds the lesser item, so case 4.1 applies and correctness follows.

These base cases cover all configurations where $|[x, y)_{X_0}| + |[x, y)_{X_1}| \leq 2$. So let $n := count_0 + count_1 > 2$, and assume that for all $x', y' \in U$ such that $|[x', y')_{X_0}| + |[x', y')_{X_1}| < n$ the protocol correctly reconciles $[x', y')_{X_0}$ and $[x', y')_{X_1}$.

$\mathcal{X}_1$ responds with two interval fingerprints $(x, m, fp([x, m)_{X_1}))$ and $(m, y, fp([m, y)_{X_1}))$ for some $m \in X_1, x \preceq m \preceq y$. By proposition 1 we only need to show that these two intervals are being reconciled correctly. Note that $|[x, m)_{X_1}| + |[m, y)_{X_1}| = count_1$ and $|[x, m)_{X_0}| + |[m, y)_{X_0}| = count_0$. Since $n > 2$, at least one $count_i$ is greater than 1, so that $\mathcal{X}_i$ will split the interval into two smaller ones, to which we can apply the induction hypothesis.

Case 1. If $count_1 > 1$, then $|[x, m)_{X_1}| < count_1$, and $|[x, m)_{X_0}| \leq count_0$, thus $|[x, m)_{X_1}| + |[x, m)_{X_0}| < n$ and the interval is reconciled correctly by induction hypothesis. For the other interval, $|[m, y)_{X_1}| + |[m, y)_{X_0}| < n$ follows analogously.

Case 2. Otherwise, if $count_1 = 1$, then $count_0 > 1$, thus $|[x, m)_{X_0}| < count_0$, and $|[x, m)_{X_1}| \leq count_1$, thus $|[x, m)_{X_0}| + |[x, m)_{X_1}| < n$ and the interval is reconciled correctly by induction hypothesis. For the other interval, $|[m, y)_{X_0}| + |[m, y)_{X_1}| < n$ follows analogously.

This concludes the proof. Note that when $\mathcal{X}_i$ splits an interval $[x, y)_{X_i}$ into $[x, m)_{X_i}$ and $[m, y)_{X_i}$, then m has to be from X_i (as opposed to $U \setminus X_i$) so that case 4.1 can be reduced to case 3, and case 4.2 to case 4.1. The induction step however still works if m is chosen from U, so the restriction can be lifted for intervals that contain at least two items. This can become relevant if some items from U can be encoded more efficiently than others.

3.3 Optimizations

We now give a couple of optimizations over the simple protocol.

3.3.1 Encodings

A first, simple optimization is to encode empty intervals not via $\mathbb{0}$, but as a dedicated message part. This allows a more compact representation, which is appropriate since $\mathbb{0}$ should by far be the most frequently occurring fingerprint. We will designate an *empty interval fingerprint* as (x, y).

A second optimization that merely changes the encoding of messages consists of a more compact representation of "adjacent" intervals: Interval boundaries $[x_0, x_1), [x_1, x_2), \ldots, [x_{k-1}, x_k)$ can be encoded as a simple list $x_0, x_1, \ldots, x_{k-1}, x_k$. As long as the protocol only talks about intervals which partition the original interval, expressing k interval boundaries only consumes $1 + k$ space as opposed to the naïve $2k$.

We can in fact do even better: when a node receives some interval $[x, y)$ and splits it into $[x, m)$ and $[m, y)$, it merely needs to send m to the other node. If the other node can reconstruct which interval is being split at point m, then all the necessary information has been conveyed. Since any m falls into exactly one previously used interval, there can be no ambiguity. It thus suffices to send triples

$(\mathit{fingerprint}_{\prec m}, m, \mathit{fingerprint}_{\succeq m})$. As a consequence, any particular $u \in U$ is transmitted at most one time during a protocol run. This optimization does however come at the cost of nodes needing to store interval boundaries across communication rounds.

3.3.2 Utilizing Interval Boundaries

The next two optimizations stem from the fact that roughly half of all intervals include their lower boundary. When a node receives an interval fingerprint starting at some $x \in U$ and knows that the other node holds that x, it can simply add x to its own set and x can be excluded from any further reconciliation effort.

An optimization that can be applied independent of U and $\preceq$ is to introduce *lower singleton intervals*, ranging from x to y and containing no item but x, denoted as $(x, y, \emptyset)$. When $\mathcal{X}_i$ receives a lower singleton interval $(x, y, \emptyset)$, it can reply with all items u such that $x \prec u \prec y$. If $x \notin X_i$, it adds it to the set, otherwise no further action is necessary. This optimization shaves off one communication round in case 4 because $\mathcal{X}_1$ responds with $(u1, y, \emptyset)$ rather than $(u_1, y, \mathit{fp}(u_1))$, thus finishing reconciliation in at most one more communication round.

In order to efficiently leverage the knowledge that some interval of size at least two contains its lower boundary, $\preceq$ must be antidense:

Definition 7. Let U be a set and $\preceq$ be a linear order on u. We call $\preceq$ *antidense* if for all $x \in U$, if there exists $y \in U$ such that $x \prec y$, then there exists a least such element, denoted as $\mathit{successor}(x)$.

If $\preceq$ is antidense, we can introduce *least-containing intervals*, ranging from x to y and guaranteed to contain x and some other items whose fingerprint is f, denoted $(x, y, \mathit{fp}(x) \oplus f)$ (this is just a notation, $\mathit{fp}(x)$ does not have to be transmitted). A node receiving a least-containing interval $(x, y, \mathit{fp}(x) \oplus f)$ reacts as if it received the item x and the regular interval fingerprint $(\mathit{successor}(x), y, f)$. Furthermore, whenever $\mathcal{X}_i$ sends a regular interval fingerprint $(x, y, \mathit{fp}([x, y)_{X_i}))$ and then receives a regular interval fingerprint $(x, m, \mathit{fp}([x, m)_{X_j}))$, neither node has item x, so $\mathcal{X}_i$ can act as if it received $(\mathit{successor}(x), y, \mathit{fp}([\mathit{successor}(x), y)_{X_j}))$ instead. In both of these cases, if $\mathit{successor}(x)$ is not defined, i.e. x is the maximal element of the order, the notes merely act as if they just reconciled x and do not perform any further work.

This approach explicitly adds a single bit of information to each interval, namely whether the lower boundary is contained or not. A different approach is to implicitly distinguish between intervals that definitely contain the lower boundary, i.e. the greater of the two intervals obtained after a split, and intervals that may or may not contain their lower boundary, i.e. all other intervals. In this setting, sending and receiving a regular interval with the same lower boundary does not imply that the other node does not hold that item.

Utilizing this information can even be done for universes with a non-antidense order, by introducing intervals which exclude both their boundaries: $(x, y)_A := \{a \in A | x \prec a \prec y\}$. When a node receives an interval $[m, y)_{X_i}$ which is a greater interval obtained from a split, it reacts as if it received the item x and the fingerprint over the interval $(m, y, (m, y)_{X_i})$.

3.3.3 Branching Degree

For the correctness of the protocol, large, populated intervals need to be partitioned and then recursively reconciled. It does however neither matter to partition the intervals into exactly two subintervals, nor that the intervals are evenly partitioned in the middle. Increasing the number of subintervals per recursion step reduces the total number of roundtrips at the cost of less efficient bandwidth usage, as will be seen in the complexity analysis in section 3.5.

In a similar vein, there is no inherent reason to initiate reconciliation by only sending a single fingerprint for the whole interval of interest, the initiating node and just as well partition the interval of interest and send fingerprints for each subinterval.

Splitting intervals into partitions of unequal sizes can be beneficial if missing items are not expected to be distributed uniformly at random across the whole order. Items might for example be ordered by the time at which they were created, a long-running node would then expect a continuous stream of new items, but would rarely receive an unknown item from the far past. Interval selection could reflect this by e.g. partitioning the $\frac{1}{2}$ oldest items into the first subinterval, then the next $\frac{1}{4}$ into the second subinterval, the next $\frac{1}{8}$ into the third, and so on. If unknown old items are rare enough, the average size of intervals which require recursion is less than under a split into intervals of equal sizes.

3.3.4 Bounded Boundaries

The boundaries of intervals come from the universe U and need to be transmitted. If U contains arbitrarily large elements, e.g. $U := \{0, 1\}^*$, the protocol cannot uphold any reasonable complexity guarantees. The solution is to employ indirection, for example by hashing the items from U and then comparing the hashes lexicographically rather than using any order on U. After the protocol has terminated, each node holds the hashes of the items it is missing.

As a last phase, the nodes can then exchange those lists of hashes and answer with the items thus requested by the other node. Alternatively, the items could be retrieved from some content-addressable storage substrate. Both of these solutions add additional roundtrips to the reconciliation. A more efficient solution is to make the protocol aware of the difference between the hashes being used for interval delimiting and the actual items of interest. Whenever the protocol determines that an item needs to be transmitted, the node transmits the actual item rather than the hash.

If the protocol is made aware of the difference and inlines items, most of the optimizations from section 3.3.2 become inapplicable, since knowing that the other node holds an item hashing to some lower boundary of an interval is not the same as obtaining that item. The only optimization in the same spirit that can be applied as the following: if a node $\mathcal{X}_i$ receives an interval fingerprint $(m, y, fp(m))$, it can act as if it received an interval containing only m and it additionally sends all items in $(m, y)_{X_i}$.

The interval boundaries do not necessarily have to be only the hashes of the items, additional information can be added so that the order on the boundaries exhibit similarity with some order on the items. Hashing has to be involved though in order to map the infinite universe to a finite set of boundaries with a low probability of collisions.

3.4 Recursive Set Reconciliation

3.5 Complexity Analysis

TODO: I am looking forward to writing this section about as much as you are probably looking forward to reading it.

3.6 Reconciling Hash Graphs

Chapter 4

Bounded Memory Set Reconciliation

- bounded memory - the need for backpressure
- credit-based backpressure
- bounded memory set reconciliation - conceptual
- bounded memory set reconciliation - protocol

Chapter 5

Other Data Structures

5.1 Higher-Dimensional Intervals

k-d-trees

5.2 Maps

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

5.3 Tries

optimizing lexicographically ordered items

5.4 Sequences

Why we need content-based slicing, even recursively

sequences as maps from rational numbers to items

Chapter 6

Related Work

- set reconciliation literature
- hash graph synchronization
- filesystem synchronization
- history-based synchronization

Chapter 7

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.

[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.