Efficient Synchronization of Recursively Partitionable Data Structures

Technische Universität Berlin

Aljoscha Meyer

April 22, 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 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, in particular 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.

Distributed version control systems can be viewn 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 in common with other nodes encountered on an 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 the efficiency 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 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].

1.1 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 peers immediately sending all their data to the other peer.

Let n be the number of items held by peer A, and m the number of items held by peer 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 peer 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 $\mathcal{O}(1)$.

In addition to the space required for per-round-trip computations, 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.

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.

1.2 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 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 A, B each hold a set of natural numbers. They can reconcile all numbers within an interval as follows: A computes a fingerprint over all the numbers it holds within the interval and then sends this fingerprint to B, together with the interval boundaries. B then computes the fingerprint of all numbers it holds within that same interval. There are three possible cases:

- B computed the same fingerprint it received, then the interval has been fully reconciled and the protocol terminates.
- B has no numbers within the interval, B then notifies A, A then transmits all its numbers from the interval, and the interval has been fully reconciled.
- Otherwise, B splits the interval into two sub-intervals, such that B has a roughly equal number of numbers within each interval. B then initiates reconciliation for both of these intervals, the roles A and 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.

The remainder of this thesis fleshes out details and applies the same idea to some other data structures. 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 X 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 X shows how to apply the same basic ideas to k-d-trees, maps, tries and radix trees, and briefly discusses why it does not make sense to apply it to arrays. Chapter X gives an overview of related work and justifies the chosen approach. We conclude in chapter X.

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.

2.1 Preliminary Definitions

We first state some general mathematical definitions that will be used in this chapter.

Definition 1. Let U be a set and $\preceq$ a binary relation on U. We call $\preceq$ a **linear order** 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

Basic Set Reconciliation

In this chapter, we consider the set reconciliation protocol sketched in the introduction. We define an unoptimized but simple version of the protocol in section 3.1, we prove its correctness in section 3.2, and do a complexity analysis in section 3.3. Section 3.4 lists some optimizations which eliminate some unnecessary work from the protocol, although not effecting the asymptotic complexity. We give an example application in section 3.5, briefly describing synchronization of hash graphs as used in e.g. git.

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 **items**. An interval fingerprint is a triple $(x, y, \mathrm{f}([x, y)_{X_i}))$ for $x, y \in U$, an item is simply some $x \in U$.

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 $(x, y, \mathrm{f}([x, y)_{X_j}))$ in the message, it does one of three things:
  - If $\mathrm{f}([x, y)_{X_j}) = \mathrm{f}([x, y)_{X_i})$, nothing happens.
  - If $\mathrm{f}([x, y)_{X_j}) = \mathbb{0}$, it adds all items in $[x, y)_{X_i}$ to the response.
  - If $\mathrm{f}([x, y)_{X_j}) \neq \mathbb{0} = \mathrm{f}([x, y)_{X_i})$, it adds the interval fingerprint $(x, y, \mathrm{f}([x, y)_{X_i})) = (x, y, \mathbb{0})$ to the response.
  - 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 $(x, m, \mathrm{f}([x, m)_{X_i}))$ and $(m, y, \mathrm{f}([m, y)_{X_i}))$ 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. TODO example figure

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 sent an interval fingerprint $(x, y, \mathrm{f}([x, y)_{X_i}))$. W.l.o.g we consider the case where $i = 0$.

We prove the statement by induction on $|[x, y)_{X_0}| + |[x, y)_{X_1}|$. There are four base cases:

If $\mathrm{f}([x, y)_{X_0}) = \mathrm{f}([x, y)_{X_1})$, then the protocol terminates and no changes are performed by any node. Assuming no fingerprint collision occurred, $[x, y)_{X_0} = [x, y)_{X_1} = [x, y)_{X_0} \cup [x, y)_{X_0}$ as desired.

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.

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 the previous case.

If $|[x, y)_{X_0}| = 1 = |[x, y)_{X_1}|$ but $\mathrm{f}([x, y)_{X_0}) \neq \mathrm{f}([x, y)_{X_1})$, let u_i be the one item held by $\mathcal{X}_i$ in the interval. $\mathcal{X}_1$ responds with the two interval fingerprints $(x, u_1, \mathrm{f}([x, u_1)_{X_1})) = (x, u_1, \mathbb{0})$ and $(u_1, y, \mathrm{f}([u_1, y)_{X_1})) = (u_1, y, \mathrm{f}(u_1))$. Since the two intervals partition the original interval, if the two into votes are reconciled correctly then so is the original interval. Correct reconciliation of $(x, u_1, \mathbb{0})$ is covered by a previous case, it remains to show that $(u_1, y, \mathrm{f}(u_1))$ is reconciled correctly. Assuming no fingerprint collision occurred, $u_0 \neq u_1$. If $u_0 < u_1$, then $\mathrm{f}([u_1, y)_{X_0}) = \mathbb{0}$, this case is already covered. So assume that $u_0 > u_1$. Then $\mathcal{X}_0$ receiving $(u_1, y, \mathrm{f}(u_1))$ is again the case where both nodes hold exactly one but different items, with the roles reversed. This time however, the initiating node holds the lesser item, so reconsiliation finishes correctly.

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

partitioning the interval at some item $x \preceq m \preceq y$. Since $[x, y)_{X_1} = \emptyset$, so are also $[x, m)_{X_1} = \emptyset$ and $[m, y)_{X_1} = \emptyset$. As shown in the previous case, these subintervals are reconciled correctly, and as they partition the original interval, so is it.

3.3 Complexity Analysis

3.4 Optimizations

3.5 Reconciling Hash Graphs

Chapter 4

Related Work

(I'm giving a very brief, very opinionated summary for the proposal, this is not meant as an actual draft for the section).

The literature for set reconciliation is fixated on minimizing roundtrips, at the cost of high computation times. [MTZ03] gives theoretical limits and a very clever protocol approaching them, but which requires $O(n^3)$ computation time per roundtrip. The authors acknowledge the practical infeasibility and offer [MT02], which is also the only paper that cares whether auxiliary data structures can be efficiently synchronized with the set to reconcile as it changes. They use a simpler approach highly related to error-correcting codes: Both peers send a digest, if the digests are similar enough, the union can be computed from holding both digests and one of the sets. If they are not similar enough, the set has been too large, so they recursively reconcile partitions of the set. Unfortunately, their auxiliary data structure is not self-balancing, so their complexity guarantees degrade as the set to reconcile is being modified.

More recent work such as [EGUV11] or [OAL+19] focuses on invertible bloom filters, and fully embraces taking $O(n)$ computation time per synchronization session. The probabilistic guarantees also involve enough math with actual numbers to require some healthy suspicion.

All of the previous work assumes that the items to be synchronized all have equal and relatively small size, which our approach does not require. None of the literature approaches utilize any structure of the data, whereas we can easily reconcile certain subsets (e.g. all data from within a timeframe if the total order being used sorts by timestamp).

I am not aware of any literature at all that acknowledges the fact that the participating peers only have finite memory available, particularly a server which synchronizes with many peers concurrently has to enforce low memory consumption per session. Any implementation of a protocol not acknowledging this will simply crash at some point.

Filesystem synchronization literature usually focuses on variants of rsync to optimize the single-file case, efficiently determining which files need updating is rarely discussed. Practical implementations usually sort all filenames, concatenate them, and then run their particular rsync the variant on that string. Using map synchronization should be more efficient.

The basic idea of adding up hashes in a tree is not particularly original, e.g. the CCNx 0.8 Sync protocol [SYW+17] does the same to solve a specific goal in a specific context. This thesis highlights the concept as a standalone algorithmic solution to a general problem, examines the concept more closely (discussing choice of the group operation, security issues, etc.), adapts it to deal with bounded memory, and applies it to more data structures than just sets.

Chapter 5

Work Plan

- 1 - 2 weeks: fingerprint chapter
- 1 week: set reconciliation chapter
- 2 weeks: bounded-memory set reconciliation chapter
- 2 weeks: synchronizing other data structures chapter
- 2 weeks: introduction, conclusion, abstract, coherence, polishing
- 4 - 8 weeks: slack, no matter how accurate those estimates, nobody keeps self-inflicted deadlines

Possibly a chapter discussing more specifics that would occur when plugging set reconciliation into a scuttlebutt-like architecture. Would highlight a context in which the complexity trade-offs are better suited than those of the related literature.

Actual implementation work, benchmarking? Probably too time intensive, but who knows.

Bibliography

[EGUV11] David Eppstein, Michael T Goodrich, Frank Uyeda, and George Varghese. What's the difference? efficient set reconciliation without prior context. *ACM SIGCOMM Computer Communication Review*, 41(4):218–229, 2011.

[MT02] Yaron Minsky and Ari Trachtenberg. Practical set reconciliation. In *40th Annual Allerton Conference on Communication, Control, and Computing*, volume 248. Citeseer, 2002.

[MTZ03] Yaron Minsky, Ari Trachtenberg, and Richard Zippel. Set reconciliation with nearly optimal communication complexity. *IEEE Transactions on Information Theory*, 49(9):2213–2218, 2003.

[OAL$^+$19] A Pinar Ozisik, Gavin Andresen, Brian N Levine, Darren Tapp, George Bissias, and Sunny Katkuri. Graphene: efficient interactive set reconciliation applied to blockchain propagation. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 303–317. 2019.

[SYW$^+$17] Wentao Shang, Yingdi Yu, Lijing Wang, Alexander Afanasyev, and Lixia Zhang. A survey of distributed dataset synchronization in named data networking. *NDN, Technical Report NDN-0053*, 2017.

[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.