Efficient Synchronization of Recursively Partitionable Data Structures

Aljoscha Meyer

April 19, 2021

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 O(log(n)) round-trips. We give a scheme by which the fingerprints can be computed in O(log(n)) time, based on an auxiliary data structure which requires O(n) space and which can be updated to reflect changes to the underlying data structure in O(log(n)) time.

To minimize round-trips, the technique requires up to 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.

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 regarded 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

(These are notes, I cut the prose version to shorten the proposal)

- number of round-trips
- total bandwidth usage
- computation time complexity per round-trip
- space complexity per round-trip
- space complexity for auxiliary data structures
- time complexity for keeping auxiliary data structure up to date as the main data structure is being modified

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 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, 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.

2 Computing Fingerprints

(Again only a minimal explanation to keep the proposal shorter): Let U be the set of items that can be placed inside a data structure. Let $\leq$ be a total order

on U. For $x, y \in U$, $x \leq y$ and $A \subseteq U$, we write $[x, y)_A$ to denote the set $\{a \in A | x \leq a < y\}$.

Let h be a hash function from U to a smaller set of hash digests H, e.g. the set of k-bit integers for some natural number k. Let $\oplus : H \times H \to H$ be a function such that $(H, \oplus, \mathbb{0})$ form a group with neutral element $\mathbb{0} \in H$, e.g. addition modulo k with neutral element 0.

We write $f(X)$ for the fingerprint of a set of items X, which is the same as the fingerprint for the ordered list obtained by sorting X according to $\leq$, denoted as $f(x_0, x_1, \ldots)$ and defined as the sum over all $h(x_1)$:

$$f() = \mathbb{0}$$
$$f(x_0, x_1, \ldots) = h(x_0) \oplus f(x_1, \ldots)$$

Observe that $f([x, y)_A) = f([min(A), y)_A) \ominus f([min(A), x)_A)$. For efficient computation of fingerprints for arbitrary intervals it thus suffices to be able to efficiently compute the sum of hashes over arbitrary prefixes of A sorted according to $\leq$.

To that end, store A in a balanced search tree that holds the sum over the hashes of all descendents in every internal vertex, and the elements of A in the leaves (this can be maintained as a self-balancing tree). $f([min(A), x)_A)$ can then be computed in $O(log(n))$ time by traversing from the root to x and summing over the hashes stored in the left children of all vertices encountered in the traversal.

3 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 round-trip. 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.

4 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 plucking 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.

References

[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.