A Connectionless Grow-Only Set CRDT

Christian Tschudin, University of Basel, Switzerland
christian.tschudin@unibas.ch
Sep 7, 2022

ABSTRACT

Using a single message type, we show how to obtain an efficient convergence protocol for a grow-only set CRDT (Conflict-Free Replicated Data Type) although the communication channel can drop and reorder messages at will. This permits to remove the middleware layer that is usually required in CRDT protocols for providing reliable or ordered message delivery. We use the obtained grow-only set for synchronizing a *compression dictionary* among peers, without the need of addresses, membership protocols or connections. In this paper we describe our CRDT protocol (which is a variant of a delta-CRDT that does not have to repeatedly send the whole state), how it can be used to bootstrap other convergent data structures and report on first simulations as well as implementations for three different platforms.

KEYWORDS

Conflict-free replicated data types, delta-CRDT, grow-only set, constraint networks, middleware, header compression, set difference protocols

1 INTRODUCTION

In peer-to-peer networks and distributed computing, Conflict-Free Replicated Data Types (CRDT) have been singled out as the key ingredient that permits off-line first operations and does not require consensus (which comes with scaling limits and introduces a dependency on some central truth): with CRDTs, replicas will naturally converge to the same state as they guarantee eventual consistency, also implying that the synchronization sequence or communication pattern among peers does not matter.

In practice, CRDT protocols often make use of *connections* in order to reliably exchange large amounts of state or to run some set difference protocol, which has undesirable consequences: connections require contact times of multiple round-trips; information transfer via connections cannot exploit a medium's broadcast capabilities (as they only synchronize node pairs); connections impose the management of auxiliary state ("connection closed") while conceptually all CRDT replicas should be "open" for updates at all time. Our interest is in fully connectionless protocols for implementing CRDTs.

From Delta Mutators with Large State Transfers ...

The two classic protocols for CRDTs are either state-based or operations-based. In the first case, which is suitable for gossip-style communications, the whole state is shipped that aggregates all changes a replica has collected so far. In the operations-based approach, only the *actions* that change a CRDT's state need to be communicated but this requires some middleware that provides the reliable in-order delivery of events. A third type, delta-CRDTs, was introduced in 2015 that is also state-based but with the twist of exchanging *state changes* (generated by delta mutators), except for a periodic retransmission of partial or full state in order to compensate for dropped gossip messages.

The need for periodic retransmission of potentially large state in delta-CRDTs has the undesirable property of requiring long contact times. For example, in a vehicular network setting, cars have only limited message exchange opportunities while they drive by wireless road-side units: there may even not be enough time to set up a WiFi session and having the TCP 3-way handshake succeed when the car already starts to move outside the reception range. Similarly, some low-bandwidth networks like LoRa make it infeasible to use them for large (delta-) CRDT state because that state has to be retransmitted over and over again, potentially using up all available transport capacity.

One way to mitigate full state exchange is to run a "set difference protocol" where two peers first identify their differences in the stored state and then run a reconciliation protocol. Since the early peer-to-peer networks of 2000, many such protocol have been proposed and studied e.g., [5]. While these approaches look promising for reducing contact time, the above paper for example requires a reliable TCP connection between peers. This point-to-point pattern excludes any gain that would be possible in a wireless broadcast media, forcing each pair of peers into redundant dialogues.

... to fully Connectionless and Incremental Updates

Our goal is to have a CRDT protocol that does *not* require any kind of connection, nor peer addresses (e.g. needed to run TCP), nor middleware that would have to do bookkeeping for delivering updates in the right order. Instead, we wish that changes to the CRDT propagate in a wave-like fashion whenever peers get into contact, incrementally updating the nodes that the wave encounters and using as few data packets as possible. We designed such a connectionless

protocol for the grow-only set CRDT that reduces the minimum contact time to roughly one round trip i.e., well below a "connection threshold": it maps the three tasks of reliability (using ARQ), the set difference computation and the set reconciliation into a single message type where no other signaling or ACK messages are needed.

Use Case and Implementation: Compression

Our use case for such a grow-only set CRDT with minimal communications requirements is the *compression* of messages of a second-level set difference protocol. Specifically, we apply it to sets of 32 Byte identifiers and use the sets as compression dictionaries. Such a dictionary allows a peer to refer to any of the long identifiers by using an index value of a few bytes at most, resulting in a ten-fold compression gain. Moreover, we can *name* the current state of the grow-only set and use this name as a prefix for all compressed packets such that these packets always are decompressed with the right dictionary. We applied this technique to the decentralized *Secure Scuttlebutt* gossip replication protocol [6] (SSB) where the append-only logs are identified by long ED25519 public keys. We have implemented our compressed replication protocol for three different platforms (ESP32, Kotlin, Python) and can now replicate SSB logs over a wireless and addressless LoRa mesh network with data packets as small as 128 Bytes.

1.1 Contributions

Drawing an arc from CRDTs to data packets and back, one can say that the delta-CRDT approach compresses the state transfer by shipping mutator-generated deltas and our grow-only construction replaces the periodic partial or full state transfer by an on-demand "set difference and reconciliation exchange". Together they can be used as an incremental, connection-less convergence protocol for replicated compression dictionaries that permit to compress replication protocols for higher-layer CRDTs. The contributions of this paper are:

- the description of a connectionless convergence protocol for grow-only sets
- that requires only minimal contact time of roughly one round-trip
- that is incremental (ideally, only novel state is transmitted, not already known state)
- yet works with lossy and reordering channels
- that does not require peer identifiers (peers are anonymous)
- that can be used for compressing other convergence protocols
- and that has been implemented for low-bandwidth wireless mesh networks.

2 RELATED WORK

State-based (also called convergent) and operations-based (also called commutative) replicated data types have been introduced around 2010, see for example the comprehensive study by Shapiro et al from 2011 [8]. State-based CRDTs are well suited for gossip style communication as long as messages cannot be lost. Beside having to send with each update the full state of the CRDT, another drawback of convergent CRDTs is that it may be necessary to keep track of the replica instances e.g., reserve a dimension for each replica in the vector that encodes a convergent integer register.

δ-CRDTs have been introduced in 2015 by Almeida et al [1] as a way to avoid sending the full state: delta mutators compute state changes that are collected and sent as change sets. However, mechanisms must be in place to recover from message loss. In [1] this is achieved by periodically sending the full state. An improvement is presented in [4] that features two strategies: avoid back-propagation/BP, and remove redundant state/RR. The BP strategy requires additional bookkeeping regarding which replica introduced a change, while RR suppresses the ingestion (and thus popagation) of those part of an update that have already been forwarded. Their system still relies on connections ("for simplicity of presentation") or requires to keep track of per-peer acknowledgment numbers. In our system, replicas are anonymous and no origin nor neighborhood state has to be maintained.

Optimizing state transfer in peer-to-peer networks (but also earlier, for example in the times of the Palmtop handheld device), is closely related to set difference protocols. The challenge is to mutually compute the number of differences that two sets have and to identify the specific elements that have to be exchanged in order to synchronize the sets. An early paper is from Byers et all from 2002 [3]. A "synopsis structure" called "difference digest" is used in the paper "What's the difference?" by Eppstein et al in 2011 [5] for extracting the set of keys that differ between machines. Although the computation of the set difference using Bloom filters can happen in a single communication round, later on a reliable connection (and thus long contact times) is necessary to transfer the differences.

Suitable for large data sets, set differences can also be computed through homomorphic hashing, the hash values being digests with special properties. The idea has been expressed already in 1994 by Bellare et al [2] where the task is to compute, out of the hash of a large data set and some small input, the hash of the combined set without having to recompute the hash of the large set from scratch. A property of homomorphic hashing is that given the hash of two sets one can directly compute the hash of the difference set without knowing its elements. In 2019, Lewi et al [7] from Facebook published their library for homomorphic hashing as open source code and uses the technique to securely propagate updates instead of replicating the full data of large data sets. Although we share the spirit of incremental operations, our interest is in a more lightweight mechanism that trades cryptographic complexity (and hash values of multiple KBytes) for an iterative approach and small hash values.

Scuttlebutt is the name of a general algorithm for gossip-based set reconciliation [9] while *Secure Scuttlebutt* (SSB [6]) is the name of a specific system for disseminating cryptographically protected append-only logs. SSB can be seen as a special case of *Scuttlebutt* in that each participant works with only one key for which they produce new versions of a single value, which is their append-only log. Common to both is the use of anti-entropy gossip protocols for replication and the use of version numbers (= sequence numbers in case of the append-only logs). In this paper we aim at speeding up these replication protocols i.e., compressing the representation of the keys that are included in the exchanged digests. While it seems that we work below these two replication layers, synchronizing the directory of abbreviations is a replication task by its own right. What differs in our way of exchanging digests is that we do not make use of version numbers. Instead, our digests (called "claims") are a direct fingerprints of a replica's content, or part of it; moreover,

unlike both *Scuttlebutt* instances, there is no identification of replica holders.

3 THE SCUTTLESYNC PROTOCOL

We assume a set of keys of equal length and of sufficient orthogonality such that one key cannot be the result of XOR-ing other keys. Typically, cryptographic public keys satisfy this requirement; alternatively, arbitrary values can be turned to the desired keys using hashing, given that the values are varied enough; finally, pre-allocated keys can be envisaged where only one distinguishing bit is set, at a different position for each key.

The set of such keys is to be replicated over a network of trusted nodes with arbitrary and potentially changing topology. New set members can be added at any time to any of the anonymous replicas. Nodes can enter and leave data exchange proximity at any time where we assume that they remain in contact long enough such that at least one request/reply exchange can take place. Our goal is that when two nodes come into proximity, or several nodes become reachable in a broadcast domain, such short encounters will be sufficient to let the replicas converge. This section describes the ScuttleSync Protocol that implements the desired convergent grow-only set data structure.

3.1 Local Data Structure and "Claims"

Nodes store their set replica as a sorted list of keys. Without loss of generality we assume that there are at least two keys in the local set (see further down how to handle the case of zero or one key). Figure 1 shows the sorted list of keys as well as a "claim" for a span of this list. Claims are assertions about a node's local replica: they expose a property of a span of the local set. Specifically, a claim for a given span is a tuple that contains four fields:

- the bits of the smallest key in the span (L for 'low')
- the bits of the biggest key in the span (H for 'high')
- the bit-wise XOR of all keys between L and H, theses keys included
- the size of the span

The ScuttleSync protocol uses only one message type which represents one claim. The following section describes the protocol logic governing when such claims shall be sent and how a node reacts to a received claim. Afterwards, the protocol is shown in algorithmic form followed by a discussion.

3.2 Informal Description of ScuttleSync

Claims are either sent opportunistically or as a reaction to a received claim.

Opportunistic means that nodes periodically send a claim that covers their whole set, which is especially useful in a broadcast setup where that claim can be picked up by any node that happens to be part of that domain. Also when two peers meet (e.g., coupling via some nearfield communication or a cable), opportunistic claims can and should be sent although this can be delayed for any reasons (energy conservation, bandwidth considerations).

Reactive claims are used to narrow down and reconcile set differences. They typically cover a smaller span than full sets. There are only a few scenarios to consider that are based on a comparison of (a) the received claim with (b) the claim that results from plugging in the received claim's values and apply it to the local array of keys.

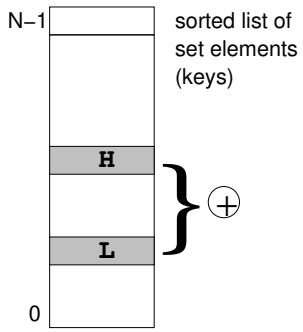

Figure 1: A claim about (a subset of) the ordered list of keys.

First, we locate the received claim's L-field in our set. If our set does not contain L, we add it. Same goes for H. Afterwards, the set of keys is sorted again. Next we retrieve the indices both for L and H which gives us the size of the span. We then compute the XOR of all keys in that local span and compare it to the received claim as follows:

- If the received claim matches our own set, no reaction is required. In all other cases, the XOR sums will differ. We continue by examining the sizes of the two spans (ours and the received one).
- If the received claim covers a bigger span, we send our freshly computed "counter claim", exposing that we lack one or more keys.
- If the received claim has a smaller span, we can help out: first, we shrink the received span (because the received boundary keys are known to the sender) and split this reduced span in two roughly equally sized spans, compute the claims for them and send these two counter claims back-to-back. This will help the other side to react on the subspan(s) where one or more keys are missing.
- When the span of the received claim has the same size as in our set (and the XOR sums differ), there must be a mutual difference where each side has some keys which the other side does not have. As in the previous case we narrow the subsequent reconciliation discussion by replying with two claims, each covering one half of the shrunk span.

The effect of these rules is that in each exchange the other side always learns something, leading to a termination of the chain of reactive claims: when for example the two back-to-back counter claims return (and the set difference is just one key), then one of them will match and therefore *not* trigger a re-reaction, while the other counter claim indicates a narrower span with a discrepancy, which triggers a counter-counter claim of reduced span size. Roughly speaking this means that a one key difference is found in $\log n$ exchanges where n is the set size.

Algorithm 1 gives a more precise formulation of the protocol logic for reactive and opportunistic claims and also contains hooks to prevent "NACK storms". Empty sets are handled by the node

being completely silent, waiting to hear a claim from others and by that way learning about the first key(s). A set of one key leads to a claim that contains that key twice, an XOR sum of 0, and a span size of 1.

Algorithm 1 The ScuttleSync Anti-Entropy Protocol

```
1:  eLst ← ϵ                                          ▷ sorted list of elements
2:  cSet ← ϵ                                          ▷ pending contradicting claims
3:  startRepetitiveTimer()

4:  proc xorSum(i, j) :                               ▷ sum up given span
5:    return reduce(xor, eLst.sublist(i to j))

6:  proc mkClaim(i, j) :                              ▷ create 4-tuple claim
7:    return ⟨eLst[i], eLst[j], xorSum(i, j), j − i + 1⟩
8:                                  ▷ field names are: .lo, .hi, .sum, .cnt

9:  proc ix(e) :                                      ▷ get index of element e
10:   return eLst.index(e)

11: onReceivedClaim (⟨lo, hi, xsum, cnt⟩) do
12:   if lo ∉ eLst then eLst.add(lo)
13:   if hi ∉ eLst then eLst.add(hi)
14:   sort eLst
15:   cSet.add(⟨lo, hi, xsum, cnt⟩)

16: onTimer () do
17:   sendClaim(mkClaim(0, |eLst| − 1))               ▷ once per period

18:   sort cSet by the count field .cnt
19:   r ← ϵ                                           ▷ set of claims to be retained
20:   foreach c ∈ cSet do                             ▷ smallest span first
21:     if c == mkClaim(ix(c.lo), ix(c.hi)) then
22:       continue                                    ▷ we are synced on this portion
23:     if c.cnt > ix(c.hi) − ix(c.lo) + 1 then
24:       r.add(c)                                    ▷ remember it
25:       c ← mkClaim(ix(c.lo), ix(c.hi))
26:       sendClaim(c)                    ▷ ask for help, if not rate limited
27:       continue

28:     ▷ we have larger or equal span size, must help
29:     lo ← ix(c.low) + 1; hi ← ix(c.hi) − 1         ▷ shrink
30:     if hi − lo < 3 then
31:       sendClaim(mkClaim(lo, hi))                  ▷ send two new keys
32:     else                                          ▷ split interval in two
33:       sz ← (hi + 1 − lo) div 2
34:       sendClaim(mkClaim(lo, lo + sz))
35:       sendClaim(mkClaim(lo + sz + 1, hi))
36:   cSet ← r
```

3.3 Operations

In the steady state of fully replicated sets, the periodic beacons emitted by each node, or sent at peering time, will not trigger any reactions (line 17 for the periodic sending and lines 21–22 for the ignoring of matching claims).

Adding a new element to one replica will trigger a push wave through the graph of reachable replicas. If a new key ends up as the new lowest or highest key, they are automatically added to the set and the same claim will be forwarded when the timer triggers the next beacon message. One could also think about immediately forwarding that new claim.

In case the new element lands in the middle of the sorted list of keys, and was not pushed in some other explicit novelty message type for optimization, the set reconciliation game will start.

Eventually, the node possessing a new key will receive claims for smaller sets than its own. It will react by sending more narrow claims (lines 28 and subsequent). If the shrunk span is small enough, only one claim must be sent that will include the new key as one of its boundaries (line 30). Otherwise the shrunk span is split in two and two claims are sent. Note that one could also randomly choose one of two subspans, limiting the chatter but increasing convergence time.

The more subtle case is now a node that realizes that it lacks one or more keys. This happens when observing that the locations of the received claims' border keys leads to a smaller span (line 23). It is important that the inferior node records the claim from the superior node and will repeatedly ask for help by sending its (knowingly wrong) claim about the too short span. These (knowingly wrong) claims act like ARQ messages for lossy channels wherefore it is important to remember the claim that triggers them: Otherwise, the chain of claims that narrow down the set difference will be broken too often and convergence would take arbitrary amounts of time. We therefore keep a list of pending "true" claims (lines 2 and 15). Over time when we learn about the missing keys, they will become matching claims (lines 21-22) and not be retained anymore.

Note that pending claims should be reacted to in the order of increasing span size (line 18). This leads to always favoring a "race to the bottom" with increasingly smaller span sizes. As we then learn about new keys, several pending claims with larger span size will automatically become matching claims and be removed from the *cSet* of pending claims. For performance reasons in a broadcast medium it is also advised to limit reactions to only a few claims per round in order to avoid NACK implosion.

Another subtle case is when two peers have the same amount of keys in their sets but each node has one key that the other does not have. In this case both nodes will help each other (line 28 and subsequent) and send claims for their split spans. Our protocol has no problem in having several "reconciliation dialogues" running in parallel and in opposite directions.

4 IMPLEMENTATION AND FIRST SIMULATION RESULTS

We have implemented the ScuttleSync protocol for Python, ESP32 (C++) and Android/Kotlin in order to synchronize Secure Scuttlebutt logs among Smartphones, Laptops and small LoRa-capable embedded devices. In all three cases the core event processing and message generation logic shown in algorithm 1 fits in 50 to 150 lines of code, depending in the chosen language. The protocol turned out to be extremely robust, often permitting set synchronization and higher-level log replication even when bugs were present or communication

channels only worked halfway. Speaking from practical experience we are very satisfied with the protocol's robust behavior.

In order to understand the scaling behavior we have started to explore the runtime behavior also with simulations. This is work in progress and has not reached enough maturity yet to be directly compared to e.g., the extensive simulations in [9] and [4]. We nevertheless report on first observations.

The original delta-CRDT algorithm ([1]) serves as a baseline and comparison point. As mentioned in the related work section, this protocol assumes that connections are used for the state transfer (which means that this transfer could take arbitrarily long to finish). We modeled the necessary retransmission logic by enqueuing all set elements at regular intervals: this strategy will recover packets that were lost in a previous interval while permitting us to model the reliability service without introducing node identifiers and their connection state. Our ScuttleSync protocol is subject to the same packet loss rate. The comparison is thus between delta-CRDT updates plus an aggressive retransmission strategy of repeatedly queuing the full set vs our exchange of selected claims.

The simulation was implemented in Python and assumes a ring topology of eight nodes and set sizes from 32 to 1024. Four scenarios were considered:

(1) "priming": one node has the full set, all others need a copy
(2) "onboarding" one node lacks all keys, the others are in sync
(3) "fixing": one node lacks one key
(4) "spreading": keys are randomly assigned to one node

We report here on first findings regarding convergence time measured in *number of communication rounds*. All experiments were run 200 times and the diagrams show the average values together with a vertical bar that shows the range of values, for a specific configuration. We chose the number of set elements as variable, ranging from 32 to 1024 and doubling in each step. We show here the series of experiments where packets were lost with a probability of 10%.

The performance of the delta-CRDT baseline algorithm is shown as a thin blue line, the algorithm is configured to send out the full grow-only set CRDT every 15 rounds. ScuttleSync runs once per round according to algorithm 1 and its performance is shown as a thick red line.

Starting with the easy "fixing" case, we see in Fig. 2 that delta-CRDT always needs at least 15 rounds to find the single missing key, sometimes even 30 rounds due to packet loss. ScuttleSync's performance is dependent on the set size, as it needs several rounds to iteratively narrow down the span where in the sets that key is missing.

In the "onboarding" scenario where one node starts with an empty set, ScuttleSync is quite effective in filling up that, due to always sending at least two new keys with each narrowed claim, although packet loss can force ScuttleSync to go into finding single missing keys, as the vertical bars show. Delta-CRDT needs either two or three of its 15-round terms in order to compensate for the lost packets (sending packets three times with 10% packet loss is probably a suitable strategy).

The two surprising scenarios are the "priming" (one node has the full set, all others are initially empty) and "spreading" (each key is randomly assigned to one node, initially). In both cases it seems that the informed set reconciliation of ScuttleSync really pays off.

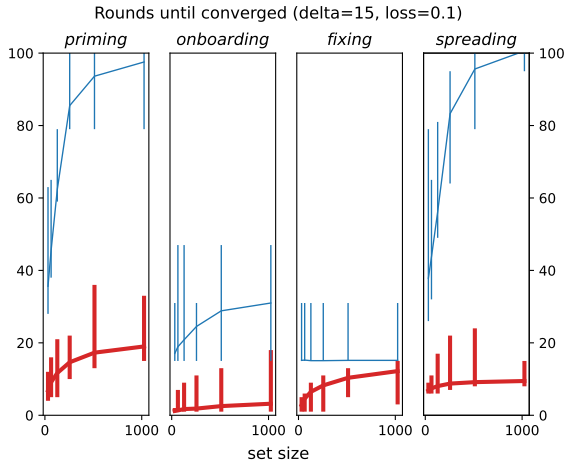

Figure 2: Convergence time in rounds, assuming 10% packet loss. Blue/thin: delta-CRDT, red/thick: ScuttleSync

These are first, preliminary findings that so far have mostly served to tune some of ScuttleSync's strategy knobs, such as how many help requests to serve per round. More explorations are needed, especially on the message complexity side (not shown here) where taming the NACK implosion is a key concern for dense wireless networks.

5 DISCUSSION

The convergence protocol as presented above has no security measures: keys can be added by everybody and without control about content (e.g., random bits instead of valid ED25519 identifiers).

In order to secure the protocol, either the protocol messages would have to be encrypted and the trustworthy nodes share a key, or claims would have to be signed and nodes would only accept them if corresponding certificates can be validated. The later would have the undesirable consequence that replicas would not be anonymous anymore.

Regarding the protection against set pollution with random bits, the set *elements* could be made larger than for example an ED25519 public key by mandating that the element contains the ED25519 key plus a self-signed certificate (as a proof that the key indeed is cryptographically functional). In order to avoid that *claims* now become very large, we suggest to use the hash of these extended set elements in our reconciliation protocol, and to send the extended elements only on request, in a new message type.

5.1 Out of Sync Compression Dictionaries

In our use case of using the grow-only set to represent a compression dictionary, one concern was that different versions of the set will be found and nodes would send index values that refer to different keys, depending at which dictionary state the receivers are.

We solved this problem by introducing dynamic "port values" that depend on a node's grow-only set: the XOR sum of all keys is a fingerprint that we use to derive a 7 Bytes wide port number. The higher-level replication protocol sends its replication commands to such a port number, using the *sender's* state: only when the receiver

has the same state, then the receiving ports will have been activated and the message is received, while any message with another port value will be simple ignored. With this mechanism, cliques of nodes sharing the same CRDT progress will be able to talk to each other, in parallel – there is no need to force a network-wide consensus for compression to work. As a future optimization for reducing the downtime due to divergent state one can envisage nodes that remember past states, hence can continue to serve straggler nodes while at the same time talking to nodes with more advanced states.

5.2 Optimizations

We already mentioned the introduction of a new "novelty" message type that would be used to fasttrack the forwarding of new keys. Only when the novelty push wave suffers from a packet loss would the nodes have to trigger the set reconciliation functionality.

The claim message type could be complemented by a smaller beacon message that only contains the XOR sum and the size of the set replica, saving precious bandwidth because of the constant repetition of these beacon messages. Only when a beacon message is received that does not match the current set state, would full claim messages be used.

In case of a claim with a span size of three, and the middle key being the one that the recipient is lacking, this key can be directly computed from the claim.

6 CONCLUSIONS

We showed a first fully connectionless CRDT protocol for wireless settings where contact time can be too short for establishing connection state and where replicas do not need an identifier. The later property permits to add new replicas on the fly, without having to inform other nodes. Having only a single message type, the combined dissemination and set reconciliation protocol has been shown to be simple, compact and efficient. We use the grow-only set as a compression dictionary for the Secure Scuttlebutt replication protocol where the 32-Bytes ED25519 identifiers can now be replaced by indices fitting in a few bytes. We implemented a tiny version of the SSB protocol for the ESP32 processor, using 128 Bytes LoRa frames, and also have implementations in Kotlin and Python.

While many efforts for decentralized software and data structures strive for feature-richness and generality, we think that there is a benefit in investigating bare-bones solutions, helping to sharpen our understanding of decentralized communication and computation principles with opportunities to collapse responsibilities currently spread across multiple layers into simpler and more efficient protocols.

ACKNOWLEDGMENTS

We would like to thank Erick Lavoie and the anonymous reviewers for their useful comments on an earlier draft of this paper.

REFERENCES

[1] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. 2015. Efficient State-Based CRDTs by Delta-Mutation. In *International Conference on Networked Systems (NETYS 2015)*. Springer LNCS volume 9466, 62–76. https://doi.org/10.1007/978-3-319-26850-7_5 arXiv:1410.2803

[2] Mihir Bellare, Oded Goldreich, and Shafi Goldwasser. 1994. Incremental Cryptography: The Case of Hashing and Signing. In *Advances in Cryptology — CRYPTO '94*, Yvo G. Desmedt (Ed.). Springer Berlin Heidelberg, 216–233.

[3] John Byers, Jeffrey Considine, Michael Mitzenmacher, and Stanislav Rost. 2002. Informed Content Delivery across Adaptive Overlay Networks. In *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (Pittsburgh, Pennsylvania, USA) *(SIGCOMM '02)*. New York, NY, USA, 47–60. https://doi.org/10.1145/633025.633031

[4] Vitor Enes, Paulo Sérgio Almeida, Carlos Baquero, and João Leitão. 2019. Efficient Synchronization of State-Based CRDTs. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 148–159. https://doi.org/10.1109/ICDE.2019.00022

[5] David Eppstein, Michael T. Goodrich, Frank Uyeda, and George Varghese. 2011. What's the Difference? Efficient Set Reconciliation without Prior Context. *SIGCOMM Comput. Commun. Rev.* 41, 4 (aug 2011), 218–229. https://doi.org/10.1145/2043164.2018462

[6] Anne-Marie Kermarrec, Erick Lavoie, and Christian Tschudin. 2020. Gossiping with Append-Only Logs in Secure-Scuttlebutt. In *Proceedings of the 1st International Workshop on Distributed Infrastructure for Common Good*. 19–24.

[7] Kevin Lewi, Wonho Kim, Ilya Maykov, and Stephen Weis. 2019. Securing Update Propagation with Homomorphic Hashing. Cryptology ePrint Archive, Paper 2019/227. https://eprint.iacr.org/2019/227 https://eprint.iacr.org/2019/227.

[8] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. *A comprehensive study of Convergent and Commutative Replicated Data Types*. Research Report 7506. INRIA. http://hal.inria.fr/inria-00555588/

[9] Robbert van Renesse, Dan Dumitriu, Valient Gough, and Chris Thomas. 2008. Efficient Reconciliation and Flow Control for Anti-Entropy Protocols. In *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware* (Yorktown Heights, New York, USA) *(LADIS '08)*. New York, NY, USA, Article 6. https://doi.org/10.1145/1529974.1529983