Department: Head
Editor: Name, xxxx@email

Enabling Cost-Benefit Analysis of Data Sync Protocols

Novak Boškov, Ari Trachtenberg and David Starobinski
Boston University

***Abstract*—The problem of data synchronization arises in networked applications that require some measure of consistency. Indeed data synchronization approaches have demonstrated a significant potential for improving performance in various applications ranging from distributed ledgers to fog-enabled storage offloading for IoT. Although several protocols for data sets synchronization have been proposed over the years, there is currently no widespread utility implementing them, unlike the popular Rsync utility available for file synchronization. To that end, we describe a new middleware called *GenSync* that abstracts the subtleties of the state-of-the-art data synchronization protocols, allows users to choose protocols based on a comparative evaluation under realistic system conditions, and seamlessly integrate protocols in existing applications through a public API. We showcase *GenSync* through a case study, in which we integrate it into one of the world's largest wireless emulators and compare the performance of its included protocols.**

INTRODUCTION

Replication is a common thread among disparate distributed systems, typically arising when there is a need for *fault tolerance* or *availability*. Multiple replicas enable an administrator to reroute requests from failed to healthy replicas, while a repair is completed. Repaired replicas can then be returned to a consistent state from existing healthy replicas.

Replication can also be used as a tool to improve the *performance* of distributed systems with many concurrent accesses. For instance, globally distributed databases and content delivery networks use replication to speed up accesses from geographically dispersed locations.

Beyond availability, fault tolerance and performance benefits, replicated systems can also facilitate *decentralization*. For example, in distributed ledger technologies (*e.g.,* blockchains), each participant holds a replica of an entire data set (*i.e.,* transactions), which allows any participant to independently verify the state of system. As long as enough participants maintain a substantial level of consistency among their replicas, no one needs to trust a central authority.

At the core of replication is *data synchronization (sync)* or *reconciliation*, the process that brings multiple replicas into a consistent state. Assuming that there are two parties aiming at syncing their data sets, a naive solution would simply exchange data sets. Having both sets,

 Published by the IEEE Computer Society

the parties can easily identify the differing elements and include them in their own replicas. However, this is prohibitively expensive, because communicating huge data sets can be slow for bandwidth-constrained networks or resource constrained compute platforms (*e.g.,* augmented reality and wearables). Worse yet, huge sets may differ in only a few elements, making most of the communication redundant. The waste is trebled when multiple participants exchange their sets in a peer-to-peer fashion, as is the case for distributed ledgers.

Better Ways to Synchronize Data

Many protocols have been developed over time to address the weaknesses of the naive sync approach. Perhaps the most popular is Tridgell and Mackerras's Rsync [1], which is included in most Linux distributions and used as a default network-enabled synchronization utility. Originally designed for files (or bit arrays), Rsync operates by dividing the files into chunks, exchanging the hashes of chunks, and generating *instructions* on how to make two files equal. These instructions are transferred over the network and applied to synchronize the files.

Rsync does not readily generalize from files to data sets, however, there have been other approaches proposed for that purpose. One such approach is somewhat similar to Rsync and uses a specialized hash known as *Bloom filter* to compactly represent the data set. Rather than exchanging hashes of chunks like Rsync, this approach exchanges Bloom filters. The main drawback of this approach, however, is non-optimal usage of bandwidth — it exchanges some traffic even for the set elements evident on both sides of the communication channel. A *communication-efficient* protocol, on the other hand, should only exchange the information related to differences between the sets [2].

Choosing the Right Sync Protocol

Several communication-efficient protocols have been proposed in the literature. Some rely on coding theory, while other make use of probabilistic data structures. These subtle but significant differences in design make it important to be able to compare them under practical conditions. Yet prior to our work [3], there were no publicly available general tools that afforded such a systematic comparison. Our work has shown that a protocol may dominate under certain network conditions, but grossly underperform when the network conditions change. Worse yet, the best protocol choice is a function not only of network conditions, but also of the immediate compute capacities of the nodes.

Contributions

There are two significant obstacles for developers in choosing the right protocol for their applications: (1) the lack of a utility for comparative analysis of the sync protocols under practical system conditions, and (2) the complexity of integrating sync protocols into existing implementations. Therefore, the main objective of this work is to overcome these obstacles and enable the integration of the state-of-the art sync protocols in future applications, such as 6G-enabled Enhanced Reality (ER), Internet-of-Things (IoT) and Internet-of-Vehicles (IoV), where sync protocol customization is needed at the application level. We summarize our contributions as follows:

- We describe our middleware *GenSync*, to the best of our knowledge, the first utility that enables a systematic cost-benefit analysis of utilizing different sync protocols.
- We demonstrate the use of this middleware in an independent, large-scale wireless emulator, *Colosseum*.
- We show that the choice of sync protocols can significantly impair or improve performance.

Applications of Data Sync

Data sync serves as a building block for a diverse collection of applications and computing paradigms, though it is sometimes embedded deeply within the architecture of these applications. Next, we describe example applications that can benefit from *GenSync*, focusing on distributed ledgers, cloud storage services, and IoT storage offloading.

Distributed Ledgers

Distributed ledgers record the decentralized transactions of massive amounts of participants. The most popular among distributed ledgers,

blockchains, implement their transaction inventory as a list of transaction blocks that are logically chained together with cryptographic hashes.

Recent advances in blockchain and distribute ledger technologies have opened a range of new possibilities for tackling long-standing problems in related areas such as IoT access management [4], security of federated learning in fog computing [5], accident forensics in vehicular networks (VANET) of self-driving cars [6], or information poisoning prevention in mission-critical unmanned aerial vehicle networks (UAANET) [7].

Applying blockchain technology over disparate layers of Distributed Computing Continuum Systems (DCCS) [8], which are characterized by heterogeneous compute and network resources, pose new challenges to performance and reliability. To cope with these challenges, several improvements to the blockchain's networking layer have recently been proposed, based on set reconciliation protocols, including MempoolSync [9], Graphene [10], and Erlay [11]. The performance of these protocols has been shown to vary significantly depending on the network conditions and compute capabilities of nodes [3], complicating the analysis and choice of optimum reconciliation protocol and parameters.

Cloud Storage Services

Cloud storage services such as Apple iCloud, Dropbox, Google Cloud, and Microsoft OneDrive have also become commonplace for modern internet users and are known to generate tremendous amounts of data traffic for the cloud providers [12]. To reduce the amount of data transfer, these applications employ protocols that are commonly referred to as *delta synchronization*. The main objective of these protocols is to determine and transmit only those portions of data that have been updated locally. In that vein, the research community has defined a metric called TUE (Traffic Usage Efficiency) that is the ratio between total sync data traffic and the update size [12]. The state-of-the art delta synchronization protocols currently in use rely on improved versions of Rsync.

IoT Storage Offloading

One of the significant problems in IoT applications is storage offloading. A standard IoT setup deals with a relatively large number of IoT devices that each produce a substantial amount of data but lack storage and compute capacity to maintain it. Traditionally, this problem has been addressed by moving data to the cloud for processing. However, there are two significant drawbacks of such approaches: (1) the data is physically transferred to another entity, which raises various security concerns, and (2) data access latency may be unacceptable for real-time applications as each data access goes through a wide-area network. To tackle these drawbacks, Wang *at al.* [13] have come up with a fog-based architecture, based on Rsync, that allows for data storage within the boundaries of the entity that operates the IoT devices. The proposed architecture has three layers. The lowest layer consists of IoT devices that synchronize their data with the fog layer above them. The fog layer accumulates this data and synchronizes it with the cloud in batches.

Other Applications

There are many other distributed systems where data sync is used as a building block. For instance, researchers proposed a dissemination protocol for wireless sensor networks that use a variant of Bloom filter-based data sync to reduce energy consumption and propagation delay [14]. Similarly, a physiological value-based key agreement scheme for body area networks called E2PKA [15] uses data sync to reduce memory footprint and energy consumption. Data sync protocols have also been proposed as a solution to network partitioning in information-centric networking (ICN) [16] as well as re-establishing consistency among replicas of distributed databases [17]. As Distributed Computing Continuum Systems [8], [18] and applications that operate across the layers of this continuum emerge, we expect that many more applications could take advantage of an efficient data sync middleware.

GenSync Middleware

We implement *GenSync* framework as a C++ middleware library that can be used through

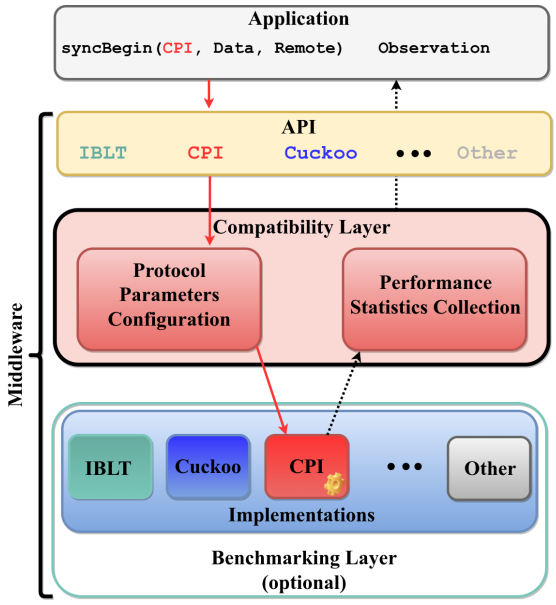

Figure 1: Middleware design.

an API, as shown in Fig. 1. The main challenges that we tackled in designing *GenSync* were (1) unifying and simplifying the parameterization of disparate state-of-the-art data sync protocols; (2) designing generic protocol implementations that enable *GenSync* utilization in various applications; and (3) constructing a lightweight yet versatile benchmarking layer. From the *GenSync* users' standpoint, the interaction between the application and the middleware library is carried out through three abstractions, named *GenSync*, *Observation*, and *Builder*.

GenSync is a generic representation of a data sync protocol. Its interface consists of two main methods, *addElement*, which adds elements to the associated data set, and *syncBegin*, which initiates synchronization with an external party using the desired protocol. In a typical application, set elements can be easily mapped to a set of unique identifiers using a *hash function* and passed to *addElement*. We design *GenSync* as an abstract interface to allow for middleware extensions, which is particularly useful to researchers that design novel set reconciliation protocols and want to benchmark against the state-of-the art, and the practitioners that design platform-specific implementations of existing protocols. A custom sync implementation needs only to implement the two *GenSync* methods. By implementing *addElement*, users can control how data set is being maintained, while *syncBegin* can be used to provide the core implementation of the custom protocol.

Observation is the summarized result of the sync and will be generated by *syncBegin*. It represents a collection of execution statistics including the measure of success, exact protocol parameters that have been used and monitoring information, such as the communication cost and time expended on data transfer and computation.

Builder is an auxiliary abstraction that facilitates the creation and connection of *GenSyncs* to each other. For example, an application may want to instantiate multiple *GenSync* objects (say, one for each of several neighbors in a peer-to-peer system). The builder allows the application to attach each *GenSync* object to remote peers through *Communicants*, which abstract out a communication channel (*e.g.,* a TCP, UDP, or local Unix socket). The *Builder* abstraction is particularly useful in heterogeneous environments, where peers may connect using different underlying transport or even physical-layer protocols, or may want to use individually optimized sync protocols for different neighbors. Code listing 1 shows how *Builder* and *GenSync* can be chained together to conduct the sync and produce an *Observation*.

```
#include <GenSync.h>

// Point to a remote and pick a protocol
auto builder = GenSync::Builder();
builder.setProtocol(GenSync::CPI);
builder.setCommunicant(GenSync::socket);
builder.setHost("the.peer.remote.addr");

GenSync gs = builder.build();

// Add data
for (auto data_point : data_set)
  gs.addElement(hash(data_point));

// Perform sync
if ( gs.syncBegin() ) {
  // Get execution statistics
  Observation ob = gs.getObservation();
  ob.communicationTime;
  ob.computationTime;
  ob.bytesTransmitted;
} else {
  // Sync failed
}
```

Listing 1: Illustrative usage of *GenSync* from an application.

Benchmarking Layer

To allow for performance evaluation under realistic system conditions, *GenSync* is equipped with a *benchmarking* layer. The benchmarking layer creates an execution environment, based on the `cgroups` feature of the Linux kernel, that simulates the target system with a given set of system parameters. Developers can readily extract the *Observations* from this simulated environment for further analysis, as we expose *GenSync*'s benchmarking layer through a script wherein developers can configure the desired system conditions and the sync protocol to evaluate (see listing 2).

```
# Protocol identifier
protocol=CPI
# Latency in milliseconds
latency=20
# Bandwidth in Mbps (in two directions)
bandwidth="10/25"
# Packet loss (percentage)
packet_loss=0.01
# Percentage of CPU cycles used for sync
cpu_server=100
cpu_client=20
# Repeat each experiment
repeat=100
```

Listing 2: *GenSync*'s benchmarking layer configuration script.

Included Protocols

GenSync includes a number of sync protocols that are based on compact auxiliary data structures (*sketches*) through a similar high-level structure. Roughly speaking, the structure of these protocols consists of four phases: ① compute sketches of local data, ② exchange the sketches between syncing hosts, ③ compute local differences, and ④ exchange differences. The protocols themselves are distinguished through their choice of sketches and how the sketches are utilized to infer the differences between the sets. Specifically, users can currently select from the following protocols.

CPI (Characteristic Polynomial Interpolation) sync is based on a representing data sets as characteristic polynomials. In phase ①, both parties encode their elements as zeros of a characteristic polynomial; they exchange evaluations of these polynomials in ②. In ③, one party extrapolates the rational function resulting from dividing these

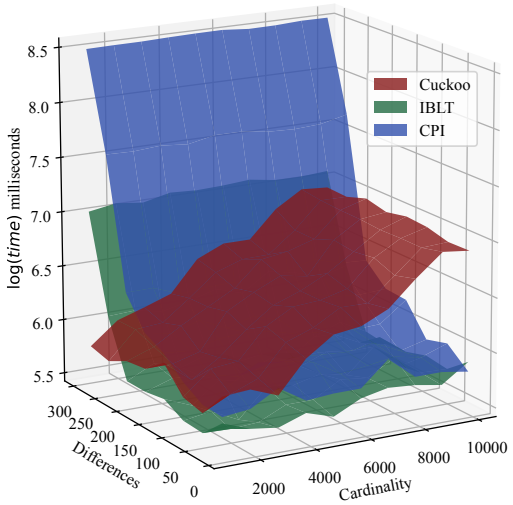

Figure 2: Logarithm of sync time as a function of set cardinality (size) and the number of differences.

polynomials, and extracts the roots of this function to determine the set differences, which are then exchanged in ④.

Cuckoo sync uses Cuckoo filters as sketches. In ①, parties insert all their elements into a Cuckoo filter, and exchange them in stage ②. In ③, each party queries its own data set elements against the Cuckoo filter of the other party. Any element for which the Cuckoo filter returns a negative answer is certainly only locally available and should thus be sent over in ④.

IBLT (Invertible Bloom Lookup Table) sync uses IBLTs as sketches, which makes it somewhat similar to Cuckoo. In ①, each party constructs their own IBLT and exchange them in ②. In ③, one party can subtract the other party's IBLT from their own to learn the elements that it needs to exchange in ④.

Navigating Trade-offs

Typical performance metrics for data sync protocols are *transferred data size* (sum of all traffic until the sets are in sync) and *total sync time*. The transferred data size depends on *data set parameters* (*i.e.,* size of the sets and the number of their mutual differences) and the protocols' theoretical bounds on communication complexity. The total sync time, however, depends on *system parameters*, which includes network bandwidth and latency (jointly referred to as *network conditions*), and the compute capabilities of the

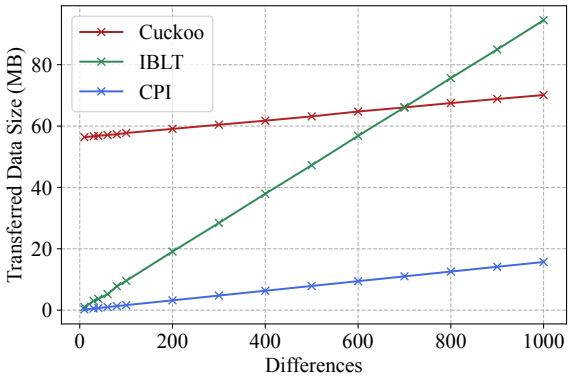

Figure 3: Protocol transfer size as a function of the number of mutual differences. Data set size is constant at 10 000.

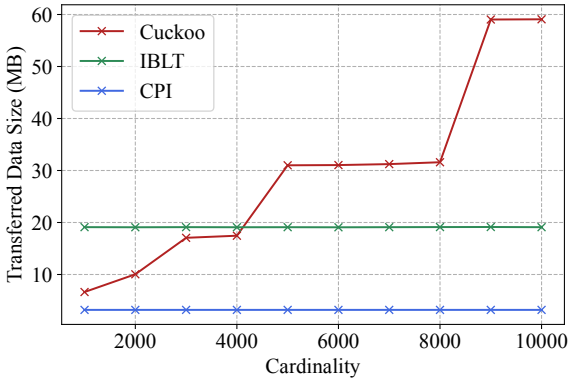

Figure 4: Protocol transfer size as a function of the data set size. The number of mutual differences is constant at 100.

nodes (*compute conditions*). Given the systems' complexity, the total sync time cannot easily be estimated using only the theoretical bounds. Worse yet, a bad protocol choice for the given system parameters can cause a 5x loss in the total sync time performance [3].

Using the *GenSync*'s benchmarking layer, we modeled a bandwidth-constrained system to explore the effects of data set parameters on total sync time. We varied set size from ten thousand to one hundred thousand, and the number of differences between zero and 300. We ran *GenSync*'s benchmarking layer on an Intel Core i7-7700 experimental server with 5.18.10 version of the Linux kernel. The benchmarking layer parameters appear in listing 2. The resulting surfaces are plotted in Fig. 2, where we can observe the following trends.

In other words, Cuckoo sync performs relatively well for very small sets, but worsens as the set size increases (regardless of the differences count). This can be explained with two observations: (1) we are dealing with a bandwidth–constrained network, and (2) the transfer size for Cuckoo increases (in steps) with the set size (Fig. 4) but stays almost constant as a function of differences count (Fig. 3). The slight increase in transfer size that we observe in Fig. 3 is mostly due to the final transfer of the differences themselves.

Trend 1

The sync time of Cuckoo is largely *invariant* to the number of differences.

Trend 2

The sync times of IBLT and CPI are largely *invariant* to the size of the data sets being synced.

That is, IBLT and CPI perform well relative to Cuckoo for very similar sets (*i.e.,* differing in only a few elements). IBLT and CPI generally do *not* transfer data for the elements that are common. Moreover, CPI transfers a nearly optimal amount of data per difference [2], whereas IBLT transfers more (Fig. 4). Whether this discrepancy in transferred data size will result in CPI's dominance (with respect to total sync time) is the matter of system parameters, and can be evaluated through *GenSync*'s benchmarking layer.

Sync on the Edge

To showcase the *GenSync* middleware's power in estimating the actual sync performance in practical systems, we apply it to Colosseum, one of the world's largest wireless network emulators [19], capable of emulating various real world radio frequency *scenarios* using software-defined radio (SDR) technology. We use Colosseum's "Boston" cellular network scenario to emulate the cellular network in the vicinity of Boston Common in Boston, Massachusetts. The scenario has *stationary* and *pedestrian* regimes, where the latter captures moderate user movement relative to the base station (see Table 1).

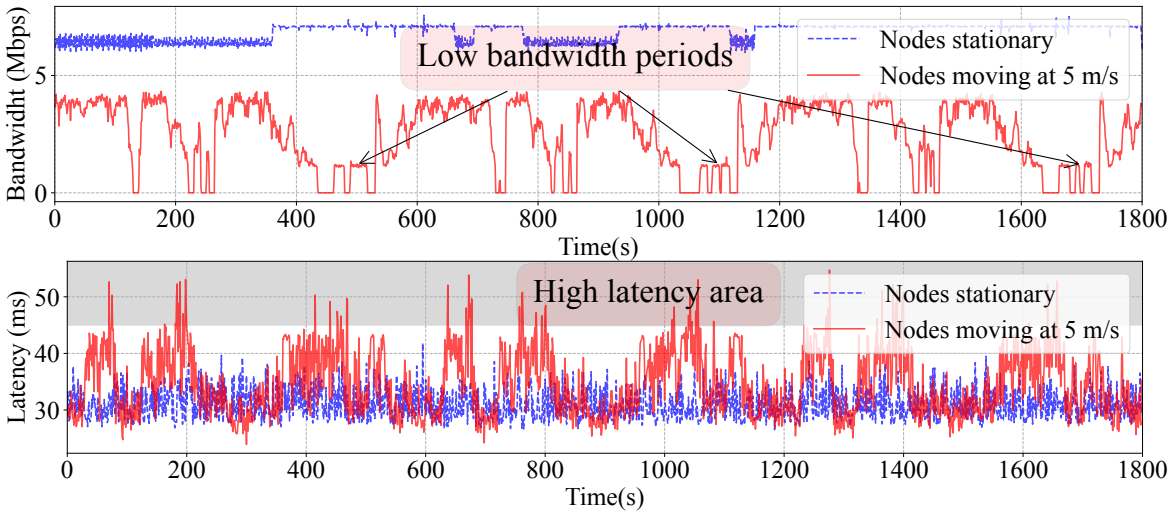

Figure 5: Bandwidth (up) and latency (down) traces from Colosseum [19], one of the world's largest wireless network emulators.

Scenario Regime	User Speed	Base Station Distance	Scenario Duration
Stationary	**0 m/s**	20 m	600 s
Pedestrian	**5 m/s**	20 m	600 s

Table 1: Emulation parameters for the Colosseum's Boston scenario [19].

The resulting network traces (the available bandwidth and latency) are plotted in Fig. 5, with their extremes annotated. These traces show that user movement results in wider oscillations of available bandwidth and latency. Using the average values of bandwidth and latency during the extreme periods, we define two sets of network conditions against which to evaluate our sync protocols:

(1) *bad* (bandwidth 1 Mbps, latency 50 ms), and
(2) *good* (bandwidth 7 Mbps, latency 30 ms).

The resulting total sync time for the two sets of network conditions and the three *GenSync* protocols is plotted in Fig. 6. IBLT performs the best in both bad and good network conditions. The reason for this effect lies in IBLT's balance of low computational complexity (linear in the number of differences) and communication cost that, although higher than CPI [2], beats Cuckoo for large data sets. As the average bandwidth in both good and bad network conditions does not drop below a critical point, IBLT also beats CPI.

However, an application-specific protocol that

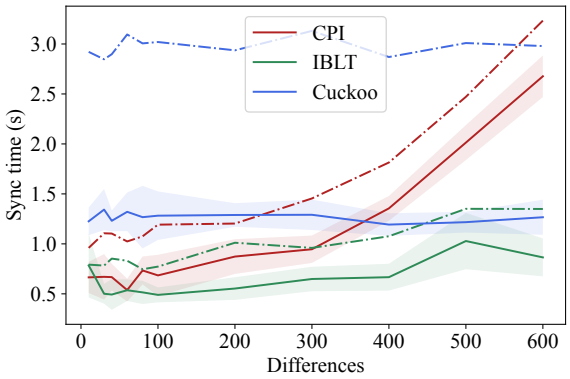

Figure 6: Sync time for the three main sync protocols in *good* (solid line) and *bad* (dashed line) network conditions. Data cardinality is 10^8. Confidence intervals are shaded.

is being constructed using *GenSync* may have additional performance objectives. For instance, it may want to be conservative about the amount of *consumed bandwidth*, while still keeping a reasonable sync time (even under bad network conditions). Since the synchronizing device may run several applications concurrently, this kind of bandwidth budgeting could be an important dimension to consider when designing an application-specific sync protocol. Bandwidth savings in the sync protocol could generate substantial performance gains for Quality-of-Experience-critical processes, such as streaming point clouds in augmented reality (AR) appli-

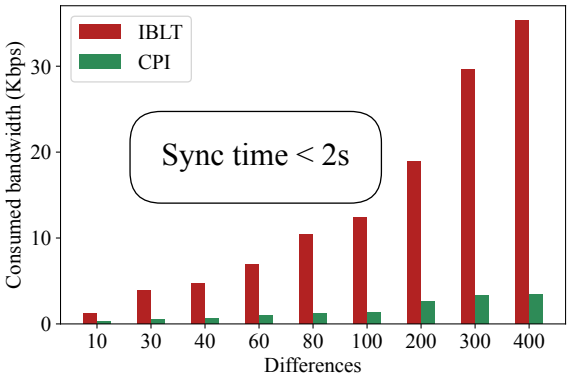

Figure 7: Bandwidth consumed for the two most bandwidth-efficient sync protocols that still complete under 2 seconds in *bad* network conditions.

cations [20]. In Fig. 7 we plot the amounts of bandwidth that IBLT and CPI consume when sync time is constrained to two seconds. In this case, CPI achieves almost nine times better performance across all difference counts. The reason for CPI's dominance over IBLT in this scenario is CPI's nearly optional communication cost, while IBLT adds a multiplicative constant to that cost.

Conclusions and Future Goals

The *GenSync* middleware library is the first open and general framework that (1) enables comparative evaluation of state-of-the-art data synchronization protocols in practical environments through a versatile benchmarking layer, and (2) allows developers to seamlessly integrate the protocol of their choice into their applications.

A current limitation of *GenSync* is that is non-adaptive, in the sense that it cannot intelligently detect the sways in system conditions (*e.g.,* available bandwidth) and automatically replace the current protocol with better one for the new conditions. We leave this promising direction for future work.

We also suggest exploring the extension of *GenSync*'s protocols to file (bit array) sync, with the hope of improving the venerable Rsync-based techniques.

Acknowledgments

The authors would like to thank Sevval Simsek for discussions regarding the Colosseum framework. The authors would also like to thank Red Hat, the Boston University Red Hat Collaboratory (award # 2022-01-RH03), and the US National Science Foundation (award # CNS-2210029) for their support.

REFERENCES

1. A. Tridgell, "Efficient Algorithms for Sorting and Synchronization," Ph.D. dissertation, The Australian National University, 1999.
2. Y. Minsky, A. Trachtenberg, and R. Zippel, "Set Reconciliation with Nearly Optimal Communication Complexity," *IEEE Transactions on Information Theory*, vol. 49, no. 9, pp. 2213–2218, 2003.
3. N. Boškov, A. Trachtenberg, and D. Starobinski, "GenSync: A New Framework for Benchmarking and Optimizing Reconciliation of Data," *IEEE Transactions on Network and Service Management*, pp. 1–1, 2022.
4. O. Novo, "Blockchain Meets IoT: An Architecture for Scalable Access Management in IoT," *IEEE Internet of Things Journal*, vol. 5, no. 2, pp. 1184–1195, 2018.
5. W. Issa, N. Moustafa, B. Turnbull, N. Sohrabi, and Z. Tari, "Blockchain-Based Federated Learning for Securing Internet of Things: A Comprehensive Survey," *ACM Computing Surveys*, August 2022.
6. M. Cebe, E. Erdin, K. Akkaya, H. Aksu, and S. Uluagac, "Block4Forensic: An Integrated Lightweight Blockchain Framework for Forensics Applications of Connected Vehicles," *IEEE Communications Magazine*, vol. 56, no. 10, pp. 50–57, 2018.
7. K. Lei, Q. Zhang, J. Lou, B. Bai, and K. Xu, "Securing ICN-Based UAV Ad Hoc Networks with Blockchain," *IEEE Communications Magazine*, vol. 57, no. 6, pp. 26–32, 2019.
8. S. Dustdar, V. Casamajor Pujol, and P. K. Donta, "On distributed computing continuum systems," *IEEE Transactions on Knowledge and Data Engineering*, pp. 1–1, 2022.
9. M. A. Imtiaz, D. Starobinski, A. Trachtenberg, and N. Younis, "Churn in the Bitcoin Network," *IEEE Transactions on Network and Service Management*, pp. 1–1, 2021.
10. A. P. Ozisik, G. Andresen, B. N. Levine, D. Tapp, G. Bissias, and S. Katkuri, "Graphene: efficient interactive set reconciliation applied to blockchain propagation," in *Proceedings of the ACM Special Interest Group on Data Communication*, 2019, pp. 303–317.
11. G. Naumenko, G. Maxwell, P. Wuille, A. Fedorova, and I. Beschastnikh, "Erlay: Efficient transaction relay for bitcoin," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 817–831.

12. Z. Li, C. Jin, T. Xu, C. Wilson, Y. Liu, L. Cheng, Y. Liu, Y. Dai, and Z.-L. Zhang, "Towards Network-Level Efficiency for Cloud Storage Services," in *Proceedings of the Internet Measurement Conference (IMC)*. New York, NY, USA: Association for Computing Machinery, 2014, p. 115–128.
13. T. Wang, J. Zhou, A. Liu, M. Z. A. Bhuiyan, G. Wang, and W. Jia, "Fog-based computing and storage offloading for data synchronization in IoT," *IEEE Internet of Things Journal*, vol. 6, no. 3, pp. 4272–4282, 2018.
14. T. Chen, D. Guo, X. Liu, H. Chen, X. Luo, and J. Liu, "Bdp: A bloom filters based dissemination protocol in wireless sensor networks," in *2009 IEEE 6th International Conference on Mobile Adhoc and Sensor Systems*, 2009, pp. 593–602.
15. W. Choi, I. S. Kim, and D. H. Lee, "E2PKA: An Energy-Efficient and PV-Based Key Agreement Scheme for Body Area Networks," *Wireless Personal Communications*, vol. 97, no. 1, pp. 977–998, Nov 2017.
16. Z. Zhu and A. Afanasyev, "Let's ChronoSync: Decentralized dataset state synchronization in Named Data Networking," in *2013 21st IEEE International Conference on Network Protocols (ICNP)*, 2013, pp. 1–10.
17. D. Chen, C. Konrad, K. Yi, W. Yu, and Q. Zhang, "Robust Set Reconciliation," in *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2014, pp. 135–146.
18. P. K. Donta and S. Dustdar, "The Promising Role of Representation Learning for Distributed Computing Continuum Systems," in *2022 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, 2022, pp. 126–132.
19. L. Bonati, P. Johari, M. Polese, S. D'Oro, S. Mohanti, M. Tehrani-Moayyed, D. Villa, S. Shrivastava, C. Tassie, K. Yoder, A. Bagga, P. Patel, V. Petkov, M. Seltser, F. Restuccia, A. Gosain, K. R. Chowdhury, S. Basagni, and T. Melodia, "Colosseum: Large-Scale Wireless Experimentation Through Hardware-in-the-Loop Network Emulation," in *Proc. of IEEE International Symposium on Dynamic Spectrum Access Networks (DySPAN)*, December 2021.
20. L. Wang, C. Li, W. Dai, J. Zou, and H. Xiong, "QoE-Driven and Tile-Based Adaptive Streaming for Point Clouds," in *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2021, pp. 1930–1934.

Novak Boškov is a Ph.D. candidate at Boston University's Department of Electrical and Computer Engineering. His research interests include decentralized networks, cloud computing, and cybersecurity. Contact him at boskov@bu.edu.

Ari Trachtenberg is a professor at Boston University, with appointments in Electrical and Computer Engineering, Computer Science, and Systems Engineering. His research interests include cybersecurity, distributed systems, and information theory. Contact him at trachten@bu.edu.

David Starobinski is a professor at Boston University, with appointments in Electrical and Computer Engineering, Computer Science, and Systems Engineering. His research interests include cybersecurity, wireless networking, and network economics. Contact him at staro@bu.edu.