NOC-NOC: Towards Performance-optimal Distributed Transactions

Author(s):
Liu, Si; Multazzu, Luca; Wei, Hengfeng; Basin, David

Publication date:
2024-02

Permanent link:
https://doi.org/10.3929/ethz-b-000665216

Originally published in:
Proceedings of the ACM on Management of Data 2(1), https://doi.org/10.1145/3639264

NOC-NOC: Towards Performance-optimal Distributed Transactions

SI LIU, ETH Zurich, Switzerland
LUCA MULTAZZU, ETH Zurich, Switzerland
HENGFENG WEI, Nanjing University, China
DAVID A. BASIN, ETH Zurich, Switzerland

Substantial research efforts have been devoted to studying the performance optimality problem for distributed database transactions. However, they focus just on optimizing transactional reads, and thus overlook crucial factors, such as the efficiency of writes, which also impact the overall system performance. Motivated by a recent study on Twitter's workloads showing the prominence of write-heavy workloads in practice, we make a substantial step towards performance-optimal distributed transactions by also aiming to optimize writes, a fundamentally new dimension to this problem. We propose a new design objective and establish impossibility results with respect to the achievable isolation levels. Guided by these results, we present two new transaction algorithms with different isolation guarantees that fulfill this design objective. Our evaluation demonstrates that these algorithms outperform the state of the art.

CCS Concepts: • **Information systems → Distributed database transactions**; **Database performance evaluation**.

Additional Key Words and Phrases: Isolation levels, concurrency control, impossibility results

ACM Reference Format:
Si Liu, Luca Multazzu, Hengfeng Wei, and David A. Basin. 2024. NOC-NOC: Towards Performance-optimal Distributed Transactions . *Proc. ACM Manag. Data* 2, 1 (SIGMOD), Article 9 (February 2024), 25 pages. https://doi.org/10.1145/3639264

1 INTRODUCTION

Modern web services are layered atop high-performance database systems running in partitioned, geo-distributed environments for system scalability and data availability. Distributed transactions, encapsulating user requests, are an important building block of such database systems. To balance the inherent trade-off between data consistency and system performance [8, 20], there is, therefore, a plethora of isolation levels (or guarantees) for distributed databases. These include not only the gold-standard *serializability* but also weaker guarantees such as *read atomicity* [5] and *transactional causal consistency* [2, 31], catering for various web applications.

Substantial research efforts [3, 14, 15, 24, 32, 34, 49] have been devoted to studying performance optimality for distributed transactions with respect to isolation levels. Two representative results are the recent SNOW [32] and NOCS [34] theorems. Both of these are impossibility results that capture conflicts among desirable properties. SNOW claims that it is impossible to design a **S**trictly

Authors' addresses: Si Liu, si.liu@inf.ethz.ch, ETH Zurich, Department of Computer Science, Zurich, Switzerland; Luca Multazzu, lmultazzu@student.ethz.ch, ETH Zurich, Department of Computer Science, Zurich, Switzerland; Hengfeng Wei, hfwei@nju.edu.cn, Nanjing University, State Key Laboratory for Novel Software Technology, Nanjing, China; David A. Basin, basin@inf.ethz.ch, ETH Zurich, Department of Computer Science, Zurich, Switzerland.

ACM 2836-6573/2024/2-ART9
https://doi.org/10.1145/3639264

serializable system with **N**on-blocking read-only transactions that finish in **O**ne round-trip when transactional **W**rites are present. NOCS proves that a read-only transaction cannot terminate with **O**ne round of **N**on-blocking communication with **C**onstant-size metadata, while achieving **S**trict serializability. In both cases, three of the four properties can be achieved at best, i.e., SNOW-optimality or NOCS-optimality; in particular, under a *weaker* isolation level than strict serializability, NOW and NOC are achievable, respectively.

Although these theorems state that achieving all four desirable performance criteria is impossible, one should aim to achieve three of the four. For example, achieving N, O, and C, would boost the performance of read-only transactions. These theorems are then considered as design objectives and used to assess existing systems for potential performance improvements. For example, the performance of Eiger [31], a causally consistent transaction system, has been significantly improved by optimizing its transactional reads to meet the NOC criteria [34].

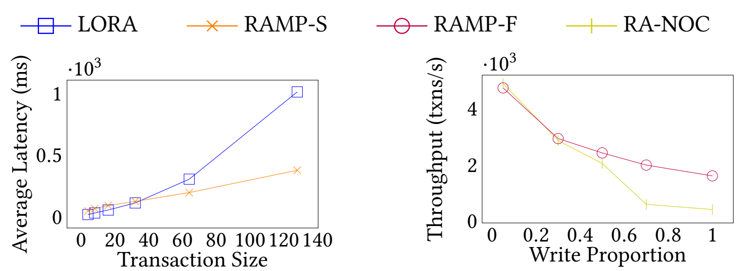


Fig. 1. Two examples suggesting that there is room for improvement beyond the optimality criteria of SNOW (left) and NOCS (right). LORA and RA-NOC are SNOW-optimal and NOCS-optimal transaction algorithms that satisfy read atomicity (RA), respectively. RAMP-F and RAMP-S are the original RAMP-family algorithms designed for RA.

Unfortunately, these performance criteria miss crucial factors that also impact on overall system performance. As shown in Figure 1, we have found two examples that suggest room for improvement beyond the optimality criteria of SNOW and NOCS, respectively. First, LORA [27], a SNOW-optimal algorithm satisfying *read atomicity* (RA), is expected to deliver optimal latency compared to other RA algorithms that do not meet SNOW such as RAMP-S [5]. However, LORA incurs significantly more latency under large-sized transaction workloads. As the metadata size is not considered by SNOW, LORA is designed with the metadata, carried by both its reads and writes, being linear in the transaction size.

Second, RA-NOC, a NOCS-optimal transaction algorithm, exhibits much lower throughput than another original RAMP-family algorithm RAMP-F [5] when writes become heavy. This is because NOCS only optimizes reads, while a NOCS-optimal algorithm like RA-NOC may use expensive locks for processing writes.

Given that such important factors are missed by the state-of-the-art design criteria, a natural question then to ask is “What would an ideal design objective be for distributed transactions?” SNOW and NOCS have already provided a promising baseline, i.e., improving system performance by optimizing reads. While reads still dominate the workloads in many applications (e.g., Google’s AdWords [44]), according to a recent study on Twitter’s workloads [52], write-heavy workloads, with 30% or more writes, are significantly more common in practice than previously thought and expected to rise in prominence.

The NOC-NOC Design Objective. In this paper, we make a substantial step towards performance-optimal distributed transactions by aiming to additionally optimize writes. This adds a fundamentally new dimension to the performance optimality problem.

Optimizing writes can improve overall system performance, especially given that system components are often co-designed. In particular, less write latency leads to less average latency for the entire system; moreover, higher throughput for writes gives rise to higher overall throughput. As we will see in our case studies, even under read-heavy workloads, making writes efficient significantly improve the overall system performance over the state of the art.

We propose the NOC-NOC design objective that aims at improving *both latency and throughput* for *both reads and writes*. Along with the aforementioned NOC for reads, we define fine-grained criteria, also called NOC, for writes as: an optimal write transaction shall proceed under **N**on-blocking concurrency control, safely commit in **O**ne round-trip,[1] and carry only **C**onstant-size metadata, while fulfilling the promised isolation guarantee, together with the accompanying transactional reads.

Blocking for a write, due to, e.g., the use of locks or validation in optimistic concurrency control, would increase the latency of the write transaction and decrease the system throughput due to, e.g., CPU underutilization [54]. One-phase commit would incur less write latency and thus overall system latency. Extra metadata (or message payload), e.g., increasing with the number of database partitions, would burden the transmission or processing, thus negatively affecting both latency and throughput.

Moreover, to save system developers' effort trying to achieve impossible objectives, we also identify the upper bound of achievable isolation guarantees for NOC-NOC. We prove that no transaction system that provides parallel snapshot isolation [45] (a slightly weaker variant of snapshot isolation) or any stronger isolation guarantees can achieve all the NOC-NOC criteria. This suggests room for potential improvements to existing transaction algorithms that offer weaker isolation guarantees, such as read atomicity and transactional causal consistency.

There is an inherent trade-off between data freshness and one round-trip reads with atomic visibility [49]. We prove that, with additional one-phase writes, no transaction system that supports read committed or any higher isolation level can make its writes visible to readers from a different session immediately after the prepare phase completes. This, however, suggests that the desired read-your-write session guarantee [48] is still potentially achievable under NOC-NOC.

Through the lens of NOC-NOC, along with its impossibility results, we examine the state-of-the-art transaction algorithms and identify a gap in the design space. We focus in particular on two recent isolation guarantees, namely read atomicity and transactional causal consistency, which have attracted the attention of both academia and industry (see Section 2 and Section 3.4).

We present a new transaction algorithm for each isolation guarantee that fulfills the NOC-NOC design objective. The key to achieving NOC-NOC common to both algorithms is the incorporation of two novel ideas: dual views and version vectors. The dual view extracts global safe snapshots of the database with respect to a certain isolation level and local safe snapshots for reading one's own writes without breaking the isolation guarantee, even when they are only prepared. We also leverage a version vector to encode the dual view, with one element per database partition. In both algorithms, a read request always carries two timestamps, one for each of the dual view, and a server always returns one single timestamp to update these two views.

Contributions. Overall, we make the following contributions:

(1) At the conceptual level, we address the performance optimality problem by proposing the NOC-NOC design objective that requires optimizing both reads and writes for both latency

[1] This one-phase commit is from a client's perspective: when running a two-phase commit protocol, a client can return after the prepare phase and then execute the commit phase asynchronously. This is different from the definition of one round-trip reads, which disallows off-path messages (see Section 3.2.1).

and throughput. We also establish impossibility results with respect to its achievable isolation guarantees.

(2) At the technical level, we present two new transaction algorithms, each for a different isolation guarantee, that fulfill all the NOC-NOC design criteria. We also establish their correctness. Along with both designs, we propose a novel combination of two techniques, dual views and version vectors, which can be leveraged to optimize transaction algorithms developed for other isolation guarantees.
(3) At the practical level, we implement and extensively evaluate both algorithms. Our experimental results show that both algorithms significantly outperform the state of the art and achieve satisfactory data freshness. This demonstrates NOC-NOC's effectiveness.

2 DISTRIBUTED TRANSACTIONS AND THEIR ISOLATION LEVELS

Modern web applications are built on distributed databases. With data partitioning, very large amounts of data are divided into smaller parts stored across multiple servers (or database partitions) for scalability. User requests are submitted as transactions to the database, typically represented by front ends called clients. Each client then executes the transactions in its own session.

Distributed databases provide various isolation guarantees (or levels), depending on the desired data consistency and availability. Figure 2 shows a spectrum of prevalent isolation guarantees, ranging from weak levels such as *read committed*, through various forms of *snapshot isolation*, to strong guarantees like *strict serializability*. We briefly explain two isolation levels and their variants, which we focus our case studies on.

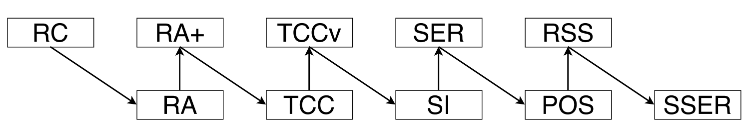


Fig. 2. A spectrum of isolation levels. $A \rightarrow B$ means that A is strictly weaker than B. RC: read committed [6]; RA: read atomicity [5]; RA+: RA with read-your-writes [5, 48]; TCC: transactional causal consistency [34]; TCCv: TCC with convergence [2, 31]; SI: snapshot isolation [6], including its weaker variant parallel SI [45]; SER: serializability [42]; POS: process-ordered serializability [13, 32]; RSS: regular sequential serializability [22]; SSER: strict serializability [42].

Read Atomicity (RA). This ensures that either all or none of a transaction's updates are observed by another transaction [5]. It prohibits *fractured reads* anomalies, e.g., in a social network Cara only observes that Ann is a friend of Bob, but Bob is not a friend of Ann. Many industrial databases have integrated read-atomic transactions as an important building block. For example, RAMP-TAO [12] has recently layered RA on Facebook's TAO data store [9] to provide atomically visible and highly available transactions.

RA with Read-your-writes (RA+). This isolation level is stronger than read atomicity. On a social networking website, all of a user's requests, submitted as transactions between login and logout, typically form a session. Read-your-writes [48] is commonly provided by many production databases such as Facebook's TAO [9], MongoDB [39], and Cosmos DB [38]. It guarantees that the effects of all writes performed in a client session are visible to its subsequent reads. For example, after Ann tweets and refreshes, she should be able to see her own tweet. All read-atomic systems we are aware of actually provide RA+, even including the RAMP-family algorithms [5] for which read atomicity is originally proposed.

Transactional Causal Consistency (TCC). This combines two properties, namely read atomicity and causal consistency (CC) [1, 43], and implies RA+. CC guarantees that two transactions that are causally related must appear to all client sessions in the same causal order. It prevents causality violations, such as Ann observing Bob's comment on Cara's post without seeing the post itself.

TCC with Convergence (TCCv). With TCC, transactions that are *not* causally related may be observed in different orders by different sessions. This may result in permanent divergence of these sessions' views under conflicting concurrent updates. TCCv's *convergence* property prohibits such phenomena by requiring these views to converge to the same state [2, 30]. For example, without convergence, Ann's and Bob's updates on the meeting place in a trip planner may end up differently to Cara and Dan, while, with TCCv, they would see the same final place. To the best of our knowledge, all production database systems claiming to support causal consistency actually delivery its stronger variant with convergence.

3 THE NOC-NOC DESIGN OBJECTIVE

In this section we present the NOC-NOC design objective and establish impossibility results with respect to its achievable isolation guarantees. Examining the state-of-the-art RA(+) and TCC(v) transaction algorithms in the light of NOC-NOC, we identify a large gap in the design space.

3.1 System Model and Assumptions

System Model. We follow the system model as in SNOW [32] and NOCS [34]. A distributed (transaction) system consists of a set of client processes and server processes that communicate by sending and receiving messages. Processes behave deterministically: in each atomic step, they receive messages (if any), perform deterministic local computations, and send messages (if any) to other processes.

The network is asynchronous with no global clock. Processes run at arbitrary speeds and messages can be arbitrarily delayed.

Definitions. A transaction starts when the client sends requests to the associated servers and ends when the client receives all necessary responses. A transaction T_1 *happens before* another transaction T_2 if T_1 ends before T_2 starts. Two transactions are *concurrent* if neither happens before the other, i.e., their lifetimes overlap. Two transactions *conflict* if they access the same key and at least one of them writes to this key. Two transactions *write-conflict* if they both write to the same object.

Given its wide adoption in practice, we employ the two-phase commit (2PC) protocol as the atomic commitment protocol for committing transactions. Each 2PC instance runs the prepare phase first, followed by the commit phase. We call a 2PC variant *one-phase commit* (or *one-phase writes*) if the 2PC coordinator (often the client) returns after completing the first phase, where all writes are fully prepared on the associated servers. The coordinator then executes the second phase asynchronously, which races with its subsequent transactions.

A transaction system satisfies the *one-phase global visibility* property if every transaction is visible to all transactions that start after this transaction completes the first phase of 2PC.

Assumptions. We assume that the system, the processes, and the network are failure-free. We also assume that every message will be delivered eventually. Our impossibility results also apply to systems with faults.

We assume one-shot transactions [23], which are common in practice. A one-shot transaction knows the database partitions that store the keys accessed by its reads/writes a priori. For one-shot transactions, we can send read/write requests to database partitions in parallel, as there are no key dependencies. Our impossibility results for one-shot transactions also apply to multi-shot transactions.

Moreover, we assume that clients issue one-shot read-only and write-only transactions to the database. Our transaction algorithms presented in this paper, along with the competing algorithms, can be naturally extended to support general read-write transactions [5]. Our impossibility results for read-only and write-only transactions also apply to general read-write transactions.

Finally, we focus on the single-datacenter setting that supports data partitioning. Our impossibility results also apply to multi-datacenters with data replication.

3.2 Defining NOC-NOC

3.2.1 Performance Criteria for Reads. NOC-NOC adopts the performance criteria for read-only transactions defined in NOCS [34].

Non-blocking Reads (N_R). Read-only transactions are considered to be non-blocking if transactional reads are processed by servers without waiting for any external events, e.g., a lock to release, a message to receive, or a timer to expire. Recall that we assume failure-free servers; otherwise, read requests to faulty servers would be stalled. N_R is desirable as blocking reads naturally increase the latency of read-only transactions. Furthermore, as blocking delays may transitively cascade, overall latency could scale with the size of the system. In addition, system throughput can also be reduced due to, e.g., CPU underutilization [54].

One Round-trip Reads (O_R). This property requires a read-only transaction to finish in one round-trip. Specifically, a client sends a parallel round of transactional reads to all the database partitions involved, and each partition sends at most one response back. O_R excludes read-only transaction algorithms that may abort, since retries are essentially extra rounds of communication. Read-only transactions that rely on "off-path" messages to ensure their correctness are also disallowed. For example, COPS-SNOW [32] uses extra messages during writes to help transactional reads find consistent snapshots, which are, however, not a necessity for processing writes. The O_R criterion is desirable as otherwise system latency and throughput would be negatively affected; in particular, extra off-path messages would burden servers when processing writes.

Constant-size Metadata for Reads (C_R). Metadata, such as transaction identifiers and timestamps, are typically required by servers to find the correct versions of data for read requests. The metadata associated with each transactional read are of constant size if they do not increase with the transaction size, the number of database partitions, or the number of conflicting operations. Transmitting/processing extra metadata naturally increases latency and decreases system throughput.

3.2.2 Performance Criteria for Writes. While missing C_R, the SNOW design objective defines a "write transactions" property for reads (in addition to non-blocking and one round-trip reads), requiring a read-only transaction algorithm to be able to coexist with conflicting transactional writes (W). NOC-NOC further turns this compatibility criterion into fine-grained performance criteria for transactional writes. The key insight is that reducing the communication complexity for writes can improve overall system performance, especially given that system components, e.g., read and write transaction algorithms, are often co-designed.

Non-blocking Writes (N_W). This criterion requires a non-blocking concurrency control mechanism, which excludes the use of locks and any optimistic concurrency control mechanisms that may block during validation. An example of a negative effect of using locks was shown in Figure 1, where RA-NOC incurs significant overhead due to blocked writes. Hence, similar to N_R, N_W is desirable not only for less write latency and for preventing overall network and system latency from stacking, but also for overall throughput improvement.

One-phase Writes (O_W). After finishing the prepare phase of two-phase commit, the client should commit the transaction. Subsequently, it issues the next transaction, if any, and runs the commit phase asynchronously. Achieving O_W decreases write latency, as a transaction can start immediately after the prepare phase of the previous write transaction, rather than waiting for the commit phase to complete. Moreover, committing a transaction in one phase enables earlier visibility of the versions written. This improves data freshness that may be sacrificed for one round-trip reads [49].

Constant-size Metadata for Writes (C_W). Similar to C_R, this criterion requires each request or response of a write transaction to carry constant-size metadata. Otherwise, extra metadata would downgrade overall system performance. As we have observed in Figure 1, LORA incurs significantly higher average latency under large-sized transaction workloads as the metadata carried by its writes (and reads) are linear in the transaction size.

3.3 Impossibility Results for NOC-NOC

We present two impossibility results for NOC-NOC.

THEOREM 3.1. *No transaction algorithms that support parallel snapshot isolation or snapshot isolation can achieve all six NOC-NOC criteria.*

Ardekani et al. [4] have proved that both parallel snapshot isolation (PSI) and snapshot isolation (SI) are inherently non-scalable. The key to our proof of Theorem 3.1 is to show that NOC-NOC implies scalability. Hence, both PSI and SI would be incompatible with NOC-NOC.

PROOF. It has been proved that no transaction algorithms that support PSI or SI can simultaneously achieve the GPR (genuine partial replication), OFU (obstruction-free updates), and WFQ (wait-free queries) properties [4]. We now show that the NOC-NOC criteria imply these three properties.

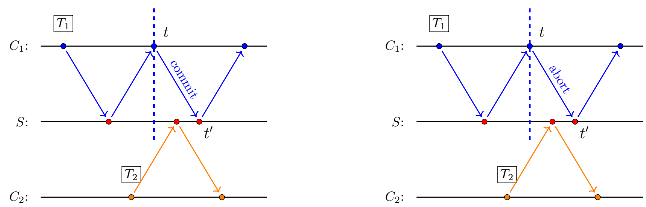
First, GPR requires that each transaction contacts only the servers that store the keys accessed by its reads/writes. Communication with other servers is a special kind of external event, which is disallowed by both N_R and N_W. Second, OFU requires every write transaction to eventually terminate in a failure-free system and commit if it does not write-conflict with another concurrent transaction. This is implied by O_W. Third, WFQ requires a read-only transaction to never wait for another transaction and eventually commit. This is implied by N_R. □

Theorem 3.1 suggests that isolation levels weaker than PSI/SI are possibly achievable under NOC-NOC. This paper focuses on two such weaker isolation levels, RA+ and TCCv, and provides a proof-by-construction for these possibilities.

THEOREM 3.2. *No transaction algorithms that support read committed can achieve all six NOC-NOC criteria with global visibility.*

This impossibility theorem holds as a read-only transaction, occurring after a write-only transaction T completes its prepare phase but before T finishes the commit phase, is unable to conclusively determine whether it is safe to read the value written by T.

PROOF. We prove the theorem using an indistinguishability argument. Consider a scenario where the client C_1 issues a write-only transaction T_1. Suppose that one of the writes in T_1 accesses the key K stored on the server S. The client C_1 first performs the prepare phase, where it sends the prepare message to the server S and receives the vote at time t. Then, C_1 proceeds with the commit phase, where S receives the notification from C_1 at time t'. Figure 3 depicts this scenario, where, for simplicity, we omit the other servers involved in the write-only transaction.



(a) T_1 commits (b) T_1 aborts

Fig. 3. Two indistinguishable scenarios for the client C_2.

Now consider another client C_2 that issues a read-only transaction T_2. Without loss of generality, suppose that T_2 only reads from the key K. The client C_2 issues a read request to S, which arrives in between t and t', i.e., when S is waiting for C_1's notification.

Suppose, by contradiction, that the system satisfies O_W with global visibility. Hence, T_1 is globally visible after t, which implies that C_2 can then safely read the value written by T_1.

Figure 3 shows two scenarios, where T_1 commits and aborts (because, e.g., another server failed to prepare the write) after t', respectively. These two scenarios are indistinguishable to C_2. Since C_2's computation is deterministic, it should behave the same in both scenarios. However, if T_2 reads the value written by T_1, then the execution in Figure 3b violates read committed. Otherwise, the execution in Figure 3a violates O_R. □

Despite being incompatible with one-phase global visibility under read committed (and beyond), NOC-NOC still allows a client's writes to be visible to its own subsequent reads. As we will see in our case studies, with the dual-view design and co-location of client sessions in practice, sufficiently fresh data are returned for reads.

3.4 Examining Existing Transaction Algorithms

The aforementioned two impossibility results suggest that it is possible to design a NOC-NOC-optimal transaction algorithm with "local visibility", which provides an isolation guarantee weaker than PSI/SI. By examining the state-of-the-art transaction algorithms, we identify a significant gap in the design space.

We focus on two isolation levels and their variants, namely RA(+) and TCC(v). The reason is fourfold. First, all these guarantees are weaker than PSI/SI, which are potentially achievable under NOC-NOC; in particular, TCCv is, to the best of our knowledge, the strongest isolation level weaker than SI and its variants.

Second, they are prevalent and have attracted the attention of both academia and industry. TCC(v) successfully combines ideas from the distributed computing and database communities and is a relatively new isolation level compared to those defined by the SQL-92 standard. We have seen a recent torrent of academic advances [2, 15, 16, 30–32, 34, 46, 47]. TCCv and its variants have also been adopted by many production database systems such as Neo4j [41] (via "bookmarked" transactions), ElectricSQL [18] (a successful transition from Cure [2] to production), and CosmosDB [38] (session guarantees and prefix consistency for its transactional batch). Moreover, Facebook also advocates the combination of causal consistency and transactions [35]. Read atomicity is an even newer isolation guarantee. In addition to its already wide range of applications in practice, including secondary indexing, foreign key enforcement, and materialized view maintenance [5], read-atomic transactions have recently been deployed atop Facebook's TAO [12].

Third, they present various underlying properties: atomic visibility for RA, additionally read-your-writes for RA+, causal consistency for TCC, and additionally convergence for TCCv. Fourth,

Table 1. Comparison of the state-of-the-art RAMP-family and Eiger-family transaction algorithms. RAMP-OPW does not satisfy the read-your-writes session guarantee. Eiger-PORT provides a weaker isolation guarantee without convergence.

System	Isolation	N_R	O_R	C_R	N_W	O_W	C_W	Optimal
RAMP-F	RA+	✓	≤ 2	×	✓	×	×	None
RAMP-S	RA+	✓	2	×	✓	×	✓	None
RAMP-H	RA+	✓	≤ 2	×	✓	×	×	None
RAMP-OPW	RA	✓	≤ 2	×	✓	✓	×	None
LORA	RA+	✓	✓	×	✓	✓	×	SNOW
RA-NOC	RA+	✓	✓	✓	×	×	✓	SNOW, NOCS
RA-NOC2	RA+	✓	✓	✓	✓	✓	✓	All
Eiger	TCCv	✓	≤ 3	✓	✓	×	✓	None
Eiger-PORT	TCC	✓	✓	✓	✓	×	✓	SNOW, NOCS
Eiger-NOC2	TCCv	✓	✓	✓	✓	✓	✓	All

the upper bound of achievable isolation levels for NOCS-optimal read-only transactions in the presence of transactional writes remains an open research question, and TCC (without convergence) is conjectured as the upper bound [34].

Table 1 shows the comparison results, where we focus on two state-of-the-art families of transaction systems, namely the RAMP-family [5, 27, 29] and the Eiger-family [31, 34] algorithms. See Section 9 for a discussion on other algorithms.

RA(+) Algorithms. None of the existing RAMP-family algorithms achieves NOC-NOC. In particular, all the original RAMP-family algorithms (i.e., RAMP-F/S/H), as well as the conjectured optimization RAMP-OPW [5], fail to provide O_R and C_R, and they do not even satisfy SNOW and NOCS. Moreover, only RAMP-OPW achieves one-phase writes at the cost of loosing the read-your-writes session guarantee. Only RAMP-S attempts to achieve constant-size metadata, but only for writes. LORA [27] is SNOW-optimal but not NOC-optimal as its metadata in both reads and writes are linear in the transaction size. RA-NOC is NOCS-optimal; however, its writes are blocking and do not commit in one phase. Our new design, RA-NOC2 fully meets all six NOC-NOC performance criteria and provides RA+.

TCC(v) Algorithms. Eiger [31] supports TCCv but fails to achieve one round-trip reads and one-phase writes. Eiger-PORT [34] optimizes Eiger with one round-trip reads by sacrificing the convergence property. We present in Section 6 a new NOC-NOC-optimal transaction algorithm, called Eiger-NOC2, which guarantees TCCv.

4 THE RA-NOC2 ALGORITHM

Following NOC-NOC's performance criteria, we now present our new design, RA-NOC2, that provides NOC-NOC-optimal distributed transactions with the RA+ isolation guarantee.

4.1 Overview

The main challenge for achieving NOC-NOC-optimal RA+ transactions is to satisfy *all eight properties together*, including the six criteria of NOC-NOC and the two consistency properties, namely read atomicity and read-your-writes (RYW). Compared to the state of the art, the most challenging part is to achieve O and C for both reads and writes (each criterion is missed by at least four of the six existing algorithms) without trading off the RYW session guarantee (RAMP-OPW sacrifices this property for optimizing writes).

To provide both one round-trip reads and atomic visibility (which RA+ subsumes), RA-NOC2 may incur data staleness [49], i.e., it cannot be guaranteed to return the most recently committed data. Given that users prefer recent data [21, 49], another challenge is, despite this theoretical result, for RA-NOC2 to achieve satisfactory data freshness in practice.

RA-NOC2 addresses these challenges by incorporating two key ideas: *version vectors* and *dual views.*

Version Vectors. Inspired by [34], we leverage version vectors to encode client views of database states. Each client maintains a version vector, with one element per database partition. Each element stores the *latest safe time* (LST) for the corresponding partition, which is defined as the minimum of the timestamps of prepared-only and committed transactions, i.e., the most recent snapshot of the database from the partition's perspective where all writes are committed and safe to return.

Unlike the RAMP-family algorithms and LORA, which rely on message metadata of linear size to compute the correct version to return, RA-NOC2 has only two timestamps for its dual view (see below) in a read request (C_R), and reads from either of these two snapshots. Moreover, the client-side version vector is updated upon receiving a response from the server, which always carries the single timestamp LST (C_W).

Dual Views. Each client maintains a dual view and reads from one of them (O_R): (i) a *global safe view* (GSV) for extracting RA-safe snapshots of the database, where each snapshot is the most recent in the version order with the incorporated versions all committed on the servers (i.e., the minimum of latest safe times across the associated partitions); and (ii) a *local safe view* (LSV) for computing RYW-consistent snapshots, when the client must fetch any potential prepared-only versions (due to one-phase writes in RA-NOC2) to read its own writes. To incorporate these versions into the LSV, the client keeps track of its most recently prepared, possibly uncommitted, version of each key locally.

Note that an LSV is often beyond a GSV, e.g., when the commit phase of a write transaction races with subsequent transactional reads from the same client. Nonetheless, the associated prepared-only writes are guaranteed to be present on the server as, otherwise, the client would not have "found" them from its LSV, *and* safe as, otherwise, the client would have read within its GSV.

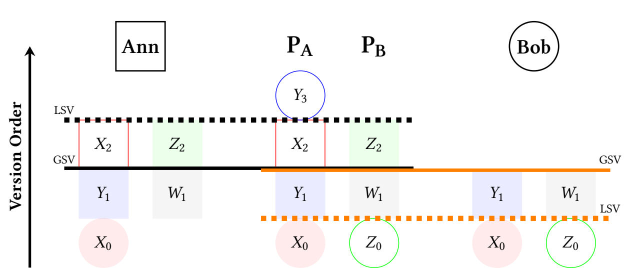


Fig. 4. An illustrative example of dual views. Colors represent different keys, and shapes different clients (squares for Ann and circles for Bob). Prepared-only and committed versions are transparent and filled, respectively. Black and orange lines refer to Ann's and Bob's dual views, respectively.

Example 4.1. Figure 4 illustrates dual views with an example of two clients, Ann and Bob, and two database partitions, each storing two keys. We assume that the keys X and Z, resp. Y and W, are always siblings in a write transaction.[2] First, as a GSV is the minimum of the LSTs across all

[2]For a key (or version), its sibling keys (or versions) are those keys (or versions) written in the same transaction.

partitions, Ann's GSV is at 1, even though her actual view of partition B is at 2 by the version Z_2. Note that her GSV cannot be advanced by X_2 on partition A since the version is only prepared and thus not safe from A's perspective.

Moreover, a client's LSV is advanced by any write of its own as long as the associated write transaction is at least fully prepared. For instance, Bob's LSV is at 0 as the version Z_0 has been prepared and its sibling version X_0 is already committed. Similarly, Ann's LSV is advanced by the write transaction that writes X_2 (prepared only) and Z_2 (committed). However, since Y_3's sibling version W_3 has not been prepared on partition B, Bob cannot advance his LSV. Note that an LSV can only include one's own prepared-only versions. Hence, Ann cannot see Z_0 prepared by Bob. Note also that an LSV can be either ahead of a GSV like in Ann's case, behind a GSV as for Bob, or even the same as a GSV.

To reduce data staleness, RA-NOC2 tightly couples dual views and version vectors by design. Instead of naively returning stale versions that satisfy RA+, a client constantly updates (i) its LSV, upon completing a write transaction's prepare phase, to include the latest writes of its own, and (ii) its GSV of latest committed versions across partitions, during the processing of both reads and writes, to always fetch the most recent versions up to the safe frontier of RA+. Moreover, via co-location of sessions in practice, clients can keep each other's dual view fresh by sharing.

4.2 Algorithm

We leverage dual views and version vectors to design RA-NOC2's read-only and write-only transaction algorithms, together with a multi-versioned database. Its pseudocode is given in Algorithm 1.

Client Dual Views. A client stores a version vector *last* (line 2), associating one timestamp to each database partition. This allows the client to then compute its GSV, i.e., the latest database snapshot that it knows to be read-atomic. Moreover, the client also maintains the *prep* data structure (line 3), containing all fully prepared, possibly uncommitted, versions' timestamps, along with their sibling keys. These versions are used to compute its LSV, i.e., the latest safe version to date for a given key from its perspective.

Server-side Data Structures. Each partition stores part of a multi-versioned database, where a version maps each key to its value and timestamp (line 26). It also holds the highest committed timestamp, called *latest* (line 27), and a set *pending* of prepared, yet uncommitted versions (line 28). The latest safe time is computed by taking the minimum of *pending*, if it is nonempty, or otherwise *latest* (lines 45–48).

Transactional Reads. A client constructs two views when reading from a key. The GSV is encapsulated by a timestamp ts_c, the minimum of *last* across all partitions involved in the read-only transaction (line 17). The LSV is unique for each key k and corresponds to the highest timestamp in *prep* where the associated version has k as its sibling key (line 19).

If the LSV is *beyond* the GSV, the client reads from the LSV of its own write by requesting from the partition the version matched by ts_p (lines 20–21). Figure 5a shows an example where Ann is reading from key X. Since her LSV is beyond GSV, her own write X_3 is returned, although it is prepared only.

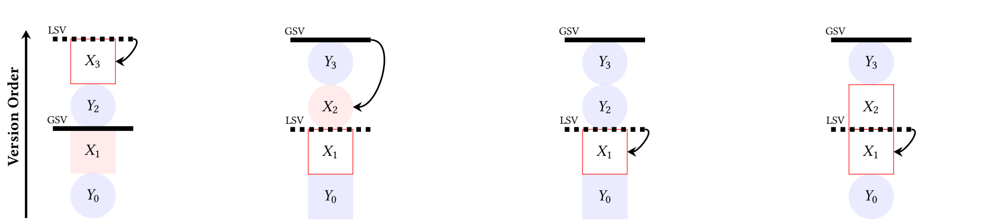
If the LSV is *within* the GSV, the client reads from its GSV by sending both ts_c and ts_p (lines 22–23). The partition then returns the most recent committed version in between these two timestamps (lines 41–43). This is the case in Figure 5b where the committed version X_2 between LSV and GSV is returned to Ann.

If no version can be found (as the client's GSV was advanced by other keys on this partition), the exact version at ts_p is sent back instead (line 44). For example, no version of X exists between the two views in Figure 5c; the version X_1 situated at the LSV is therefore returned.

Algorithm 1 The RA-NOC2 Algorithm

```
—— Client-side Data Structures & Methods ——
last[svr]: last committed timestamp on server svr
prep[ts]: write set of prepared version ts

procedure PREPARE_ALL(W : set of ⟨key, value⟩)
    ts ← generate new timestamp
    parallel-for ⟨k, v⟩ ∈ W do
        ts_svr ← PREPARE(⟨k, v, ts⟩)
        last[svr] ← max(ts_svr, last[svr])
    prep.add(ts, {w.key | w ∈ W})
    return

procedure COMMIT_ALL(V : set of versions)
    parallel-for ver ∈ V do
        ts_svr ← COMMIT(ver.ts)
        last[svr] ← max(ts_svr, ver.ts, last[svr])
    return

procedure GET_ALL(K : set of keys)
    ts_c ← min({last[svr] | svr storing k ∧ k ∈ K})
    parallel-for k ∈ K do
        ts_p ← max({ts | k ∈ prep[ts]})
        if ts_c < ts_p then
            rs[k], last[svr] ← GET(k, ts_p, null)
        if ts_c ≥ ts_p then
            rs[k], last[svr] ← GET(k, ts_c, ts_p)
    return rs

—— Server-side Data Structures & Methods ——
vers: multi-versions ⟨key, value, timestamp⟩
latest: highest committed timestamp
pending: timestamps for uncommitted write txns

procedure PREPARE(ver : version)
    vers.add(ver)
    pending.add(ver.ts)
    return LST(pending,latest)

procedure COMMIT(ts_c : timestamp)
    latest ← max(ts_c, latest)
    pending.remove(ts_c)
    return LST(pending,latest)

procedure GET(k, ts_req, ts_p)
    lst ← LST(pending,latest)
    if ts_p = null then
        return vers[k].at(ts_req), lst
    for ver ∈ vers[k].between(ts_p, ts_req) do
        if ver.ts ∉ pending then
            return ver.value, lst
    return vers[k].at(ts_p), lst

procedure LST(pending, latest)
    if pending is empty then
        return latest
    return min(pending)
```

Note that we exclude any prepared version by the same client in between the aforementioned two timestamps, as the associated write transaction has not been fully prepared yet (otherwise, the client would have sent this version's prepared timestamp, which is higher than ts_p). Figure 5d depicts such a case: Ann's own write X_2 sits above the LSV, but it is unsafe to return; instead, the read skips over it and fetches X_1. In both cases, the partition returns the latest safe time as well, which is then used by the client to advance its GSV.



(a) LSV beyond GSV: return at LSV **(b)** LSV within GSV: return in between **(c)** LSV within GSV: return at LSV **(d)** LSV within GSV: skipping own write

Fig. 5. Reading from different views (by Ann). We use the same setup as in Figure 4. For simplicity, we use only one partition to illustrate different scenarios.

Transactional Writes. The traditional 2PC protocol [25] underlies RA-NOC2's write-only transactions. A client first generates a new timestamp for the versions that it intends to prepare on the associated partitions (in parallel). Upon receiving the prepared version, the partition adds it to the local database (line 30), as well as the *pending* list (line 31). After completing the prepare phase, the client incorporates the new prepared versions into its LSV so that any subsequent reads are guaranteed to see its own writes (line 9).

Upon finishing the prepare phase, the client asynchronously commits the transaction (in parallel) with the version's (commit) timestamp. The partition then updates *latest* and *pending* accordingly. Upon receiving the ack, the client advances its GSV (for freshness) by incorporating the timestamp of the version just committed and the partition's latest safe time (line 14).

4.3 Correctness

RA-NOC2 meets all the NOC-NOC criteria by design. In particular, by encoding a client dual view into two timestamps in a read request (C_R), this enables a server to extract the precise version to return without any intervention (N_R), thus completing the read in one round-trip (O_R). RA-NOC2 meets O_W by incorporating a client's own writes into its LSV for future reads, thus enabling safe asynchronous commits. It also achieves C_W for both write request (the client's identification) and response (the latest safe timestamp).

We prove that RA-NOC2 satisfies both RA and RYW. The complete proof is given in [28]. Intuitively, as a client's GSV captures the latest safe snapshot across all partitions, which includes committed versions only, no fractured reads would ever happen, thus guaranteeing RA. Even though we may jump beyond the GSV for the client's own writes (RYW), the LSV then ensures that any versions fetched are at least fully prepared, thus no fractured reads happen either. Note that client *A*'s reads would never depend on client *B*'s prepared writes as they are only visible to client *B*.

Note also that, from a single client's perspective, one partition's LST could be far behind those of the other partitions in extreme cases, e.g., the client has not accessed the partition for a long time. This may result in stale reads, but the correctness of RA-NOC2, as well as Eiger-NOC2, is not affected. As we will see in our experiments, both algorithms can actually achieve satisfactory data freshness in practice.

5 RA-NOC2 EVALUATION

We extensively compare RA-NOC2 to the state-of-the-art algorithms, demonstrating its throughput and latency improvement. We also show that RA-NOC2's data freshness is competitive.

5.1 Competitors

We consider five *strong* competitors (see also Table 1):

- RAMP-F and RAMP-S [5], which are the two original, yet state-of-the-art, read-atomic algorithms;
- the RAMP-OPW design [5], which optimizes RAMP with one-phase writes while sacrificing the read-your-writes session property (thus providing a weaker isolation guarantee than RA-NOC2);
- LORA [27], which is a SNOW-optimal read-atomic algorithm, missing only C_R and C_W; and
- the NOCS-optimal algorithm RA-NOC, which we have designed following the NOCS design objective.[3]

We do not consider RAMP-H since its performance lies between that of RAMP-F and RAMP-S [5].

[3] The pseudocode of RA-NOC is given in [28].

5.2 Implementation, Setup, and Workloads

Implementation. For a fair comparison, we implement our algorithm RA-NOC2 (around 1000 LOC in Java), along with LORA, RAMP-OPW, and RA-NOC, atop a multi-versioned database in the original RAMP codebase [5]. Our implementation also incorporates the *cooperative termination protocol* [7] used by the RAMP-family algorithms to handle the inherent blocking issue in two-phase commit; see Section 8 for details. For RAMP-OPW, we modify the RAMP-F implementation to commit write-only transactions after the prepare phase and complete the commit phase asynchronously. Keys are assigned to a database partition by a distributed hash table. Each client contacts the front end of a partition that executes the requested transactions, i.e., the front end plays the role of the client in Algorithm 1.

Experimental Setup. As RAMP is a concurrency control protocol, its original codebase does not provide data replication. We follow the same primary-backup replication setup as in [34] where two logical data centers are co-located in a CloudLab cluster [17]. Our extension for the replication leads to additional 250 LOC. By default, each data center has five servers to partition the entire database and five client machines to load the servers. We do not consider dynamic resharding in our experiments. Each machine has two 10-Core, 3.4 GHz, Xeon E5-2640 v4 CPUs (xl170 node from the Utah cluster) and a 10Gbps network interface. We run five 60-second trials for each data point and plot the average.

Workloads. For a fair comparison, we employ the same YCSB benchmark as used by RAMP [5] to generate transactional workloads where multiple operations are grouped into read-only or write-only transactions.[4] By default, we choose the transaction size of 16 operations and the value size of 1 byte, in order to fully expose the impact of metadata size on system performance. We match the key-access distribution (Zipfian with the skewing factor of 0.99), database size (1 million keys), and read/write ratio (95% read-heavy workloads) in RAMP. We also run 5000 YCSB client threads distributed over 5 client machines.

5.3 Evaluation

Summary of Results. RA-NOC2 shows significant performance improvement over the competitors under various workloads, which do not fulfill all the NOC-NOC criteria. In particular, under large-sized transaction workloads, RA-NOC2 achieves 194%–650% improvement in throughput, along with 55%–87% reduction in latency; it outperforms the competitors under full-spectrum workloads with varying read/write ratios, exhibiting up to 82% higher throughput and incurring 46% less latency. RA-NOC2 also demonstrates its scalability with increasing numbers of servers and clients. All these performance achievements are attributed to RA-NOC2's adherence to NOC-NOC for both reads and writes. Given its one round-trip reads, RA-NOC2 is expected theoretically to trade off data freshness [49]. Nonetheless, its data freshness remains competitive with that of the existing algorithms, with over 99% up-to-date reads.

We defer the plots for the comparison between RA-NOC2 and RA-NOC to our technical report [28], where the conclusions we draw from this section (e.g., on throughput and latency improvements) also apply. Note that the design of RA-NOC is exactly the same with that of RA-NOC2 except for N_W and O_W. Hence, RA-NOC can serve as a strong baseline to showcase RA-NOC2's performance improvement by adhering to the NOC criteria for writes.

Latency Improvement. RA-NOC2 performs well with large-sized workloads. Figure 6a and Figure 6b show that, with an increase in the transaction size, RA-NOC2 significantly outperforms

[4]The RAMP codebase currently only supports the key-value API.

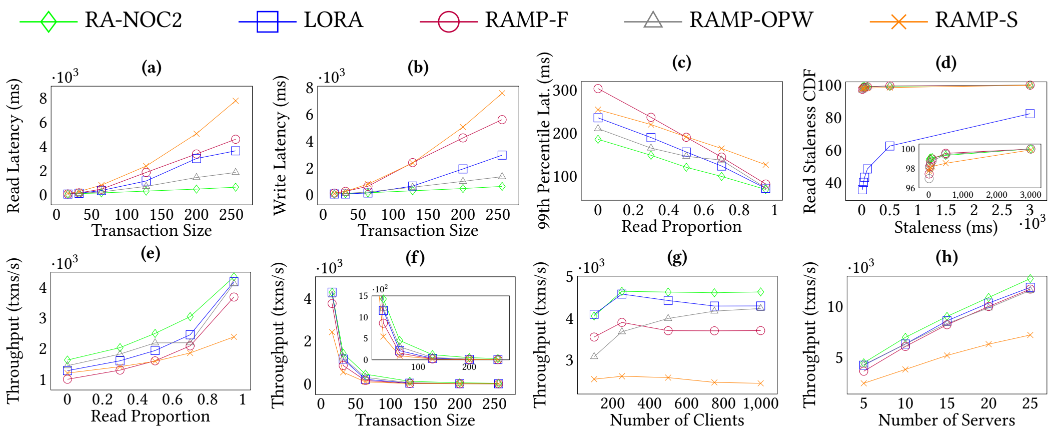


Fig. 6. Performance and data freshness comparisons. RAMP-OPW provides RA, which is strictly weaker than RA+ provided by the other algorithms including RA-NOC2.

the competitors in both read and write latency. In particular, even compared to LORA (resp. RAMP-OPW), which also has one round-trip reads (resp. writes), it achieves up to 83% (resp. 54%) reduction in read (resp. write) latency. Overall, RA-NOC2's latency improvement owes to O_R, which the RAMP-family algorithms do not satisfy, and to O_W, which all RA+ algorithms except LORA fail to provide. Two additional contributors are C_R and C_W, which naturally improve RA-NOC2's latency when transferring and handling larger metadata. We also measure 99th percentile latency with varying read/write proportions. As shown in Figure 6c, one-phase writes, along with their asynchronous commits, do not lead to more overhead, and RA-NOC2 consistently surpasses the competing algorithms under full-spectrum workloads, with 22%–46% latency reduction.

Throughput Improvement. Overall, RA-NOC2 shows substantially higher throughput than the state-of-the-art algorithms and owes this improvement to its adherence to C_R and C_W. Figure 6e shows that RA-NOC2 consistently outperforms the competitors as we vary the read/write ratio: it achieves up to 27% throughput improvement over the SNOW-optimal algorithm LORA; it also surpasses RAMP-S by 82%, even under 95% read-heavy workloads for which the RAMP-family algorithms are specifically designed. Moreover, RA-NOC2 exhibits significantly higher throughput under large-sized workloads, as shown in Figure 6f. In particular, with 128 operations per transaction, it achieves 563% improvement in throughput over LORA and at least 194% improvement over the optimized RAMP algorithm OPW.

Finally, we explore the scalability of RA-NOC2. Figure 6g shows that RA-NOC2 scales well as the number of client sessions increases (while we keep the number of client machines constant), with noticeably higher throughput. Additionally, along with an increasing number of partitions, we scale up the number of clients while keeping sessions per client machine constant, in order to fully saturate the system. Figure 6h shows that, compared to the competitors, RA-NOC2 exhibits superior scalability, consistently achieving higher throughput.

Data Freshness. We measure data freshness in terms of staleness defined as the time difference (in milliseconds) between the version read and the latest commit of the associated key. Overall, RA-NOC2 achieves competitive data freshness compared with the state-of-the-art algorithms; see Figure 6d. In particular, despite the inherent loss of freshness due to O_R [49], RA-NOC2 still reports 99% up-to-date reads, while LORA, also with one round-trip reads, only manages 35%. This owes to our design choice of RA-NOC2 whereby a client's dual view is always advanced up to the most recent, yet safe frontier of RA+, and our implementation choice of sharing views among co-located clients.

6 THE EIGER-NOC2 ALGORITHM

Guided by the NOC-NOC design objective, we improve an existing NOCS-optimal algorithm called Eiger-PORT [34], that is, to the best of our knowledge, the most performant TCC algorithm to date.

The reason for choosing Eiger-PORT as a base algorithm to demonstrate NOC-NOC is threefold. First, Eiger-PORT already provides optimal read-only transactions (NOC for reads), plus highly efficient transactional writes (N_W and C_W). Moreover, it already satisfies a sufficiently strong isolation guarantee, namely TCC. Finally, the upper bound of achievable isolation levels for NOCS-optimal read-only transactions in the presence of transactional writes remains an open research question—TCC *without convergence* is conjectured as the upper bound [34]. All together, this renders Eiger-PORT a strong baseline and any improvement to it non-trivial and challenging.

6.1 Eiger-PORT in a Nutshell

The Eiger-PORT design is guided by NOCS, thus satisfying NOC for reads. The core idea is to capture a TCC-consistent snapshot of the database per client request (O_R), which is computed over the client-side version stamps (similar to version vectors; see Section 4.1). The value of a version stamp represents the safe time (similar to the latest safe time in RA-NOC2) on a partition, and the minimum of such values across partitions like a global safe view is selected as the snapshot embedded in a read (C_R). Moreover, version stamps are extracted from the Lamport clocks used by the partitions to guarantee a causal ordering. Upon receiving a requested snapshot, the partition checks for the existence of a committed version by the client (to satisfy read-your-writes) that is strictly beyond the snapshot; given such a version, the client reorders it before the snapshot in the version order. Altervatively, if such a version does not exist, the partition performs a recursive backward search through the versions within the snapshot, finding a version that ensures read atomicity (RA). As a result, clients may observe versions in different orders; this is allowed by TCC, yet breaks convergence.

Write transactions proceed using a variant of the traditional 2PC that always commits [31]: a client sends, in addition to the coordinator (as in the traditional 2PC), to each cohort (or partition) a prepare message directly; upon receiving a request, a cohort proactively confirms with the coordinator the commitment of a transaction if it has voted "yes" but not yet received the commit. Between the two phases, the coordinator ensures that each cohort commits with the same version timestamp by synchronizing the Lamport clocks across the cohorts up to the maximum of all the proposed timestamps. This ensures a consistent snapshot of the database by the write transaction. The coordinator and the cohorts return their local safe times for updating the client's version stamp.

6.2 Overview of Eiger-NOC2

Eiger-PORT does not provide O_W and satisfies only TCC without the convergence guarantee. O_W is valuable given that write-heavy workloads (with 30% or even more writes) are significantly more common in practice than previously thought [52]. Moreover, system components are often co-designed, indicating that optimizing writes, even just their latency, would improve overall system performance (as we have observed in RA-NOC2 and will see in Eiger-NOC2). Finally, convergence is the *de facto* guarantee by causally consistent systems in practice [18, 38, 41].

Eiger-NOC2 leverages *dual views* to improve Eiger-PORT with both O_W and convergence. A dual view computes two separate snapshots of a database, i.e., the local safe view (LSV) and global safe view (GSV), which underlies the fulfillment of O_W without sacrificing RYW (as we have seen in RA-NOC2). However, its integration into Eiger-PORT is challenging. We must guarantee (i) the causal ordering, a strictly stronger consistency requirement than RYW, for both views, especially local safe views that may include prepared-only versions, and (ii) convergence without losing read atomicity (RA); the authors [34] conjecture that convergence would be incompatible with RA in the presence of causally consistent reads that satisfy NOC.

Regarding challenge (i), like in Eiger-PORT, GSVs constructed over Lamport clocks across partitions can precisely capture the causality among committed versions. However, to further ensure a causal order of LSVs, we collect on the client side during its write transaction all the prepare timestamps proposed by the partitions (also based on their Lamport clocks), and advance the client's LSV by mimicking how the coordinator would commit the transaction. Therefore, we can correctly order the prepared versions for the client's subsequent reads in the same way as they would be causally ordered across partitions upon commitment. This guarantees causality whilst achieving one-phase writes.

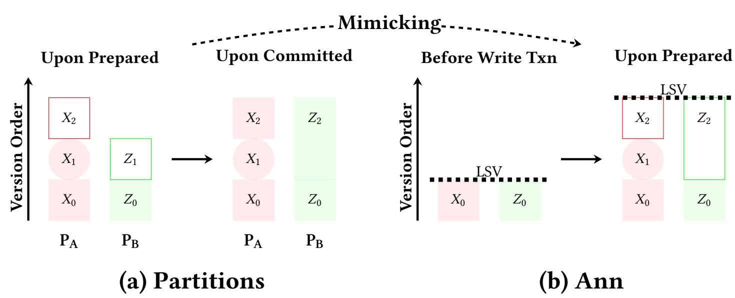


Fig. 7. Advancing a client's LSV by mimicking the partition-side 2PC. Left: partition-side dynamics of 2PC. Right: advancing the client Ann's LSV upon finishing the prepare phase. Squares refer to Ann's writes.

Example 6.1. In Figure 7, Ann is writing to the keys X and Z via a write-only transaction, and her original local safe view is at X_0 and Z_0 (left in Figure 7b). Upon finishing the prepare phase, the partitions have prepared X_2 and Z_1, respectively. Both timestamps are assigned by advancing the partitions' local Lamport clocks, respectively, which are not synchronized (left in Figure 7a). The write transaction is eventually committed at version 2 (i.e., the maximum of the prepared timestamps), with Z_1 promoted to Z_2 in particular (right in Figure 7a).[5] This commit phase is mimicked on the client side even before it happens. Specifically, the prepared messages sent back to the 2PC coordinator are also received by Ann to advance her local safe view according to the proposed prepare timestamps. Consequently, Ann can correctly order the prepared-only writes even before they are committed (right in Figure 7b).

Regarding challenge (ii), we leverage GSVs to achieve convergence while keeping read-atomic reads. Similar to Eiger-PORT, an Eiger-NOC2 client also encodes a database snapshot using a version vector and takes the minimum as its GSV. Eiger-PORT has a conservative view of RA in the sense that it tends to return the exact versions written by a write transaction, while RA in fact allows part of the reads to fetch higher versions as long as no fractured reads are exhibited. This forces Eiger-PORT to search for a conservative snapshot from the database within a client's GSV, which depends on whether a version was written by itself or another client. Consequently, this would result in different orderings of versions among clients. In contrast, we recognize that it is possible to return the highest committed version within a GSV without losing RA, and a convergent ordering can be agreed upon by all readers.

[5]Promotion in the PORT design [34] cannot ensure that all writes in the same transaction are promoted at the same time. In contrast, our promotion respects the transaction boundary, even if a transaction is only prepared.

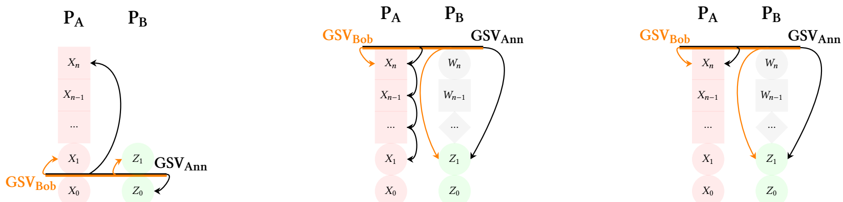


Fig. 8. Illustrating convergence in Eiger-NOC2 and read atomicity in both Eiger-PORT and Eiger-NOC2. Squares, circles, and diamonds refer to Ann's, Bob's, and a third client's writes, respectively.

Example 6.2. In Figure 8, Ann and Bob are reading from X and Z using read-only transactions and both share the same GSVs. For simplicity, we assume no prepared versions, which suffices to distinguish the two algorithms. Figure 8a shows the initial scenario, where both algorithms behave the same. In particular, with the GSV at 0, Bob returns his own committed writes X_1 and Z_1 by jumping over the GSV; Ann reads X_n written by herself for the same reason, but Bob's Z_0 as it is the only version included in the GSV.

Both GSVs are then advance to n. Bob still behaves the same in both algorithms, as shown in Figure 8b for Eiger-PORT and Figure 8c for Eiger-NOC2, and returns the highest committed versions within the GSV that satisfy RA, i.e., X_n and Z_1.

These two algorithms differ when it comes to Ann. Eiger-PORT goes through an expensive backward search until it hits a write by a different client, i.e., X_1 (Figure 8b), while Eiger-NOC2 allows Ann to return her most recent write X_n (Figure 8c). Note that both returns exhibit no fractured reads, thus satisfying read atomicity [5]. Nonetheless, in Eiger-PORT, Ann orders X_n before X_1 in the version order, while Bob sees the opposite order. This is allowed by TCC, which, however, does not satisfy convergence. In contrast, a convergent order, i.e., X_1 before X_n, is established on both Ann and Bob in Eiger-NOC2. Moreover, returning X_n to Ann reduces the partition's overhead of backward search.

6.3 Algorithm

Eiger-NOC2 leverages dual views to improve Eiger-PORT's read-only and write-only transactions. Both algorithms utilize version vectors to encode database snapshots and Lamport clocks to capture causal relations among transactions. We present its pseudocode in Algorithm 2, where we defer to our technical report [28] the partition-side procedures for transactional writes that are largely shared by both algorithms.

Transactional Reads. A client leverages its dual view when performing transactional reads. The global safe view (GSV) is taken as the minimum of *last* (line 20), a version vector encoding the most recent database snapshot that includes committed versions only (line 17). The local safe view (LSV) per key is represented by a pair $(txnid, t_{own})$, with t_{own} the commit timestamp of the latest write to the key by the client and *txnid* the associated write transaction's identifier. Upon receiving a read request, the server returns the highest committed version *ver* within the client's GSV (line 9), unless it is aware of a later, at least fully prepared, version of its own. Specifically, if the LSV t_{own} is larger than *ver*'s commit timestamp (line 10), meaning that there indeed exists a safe version by the client that is more advanced than *ver*, then the server returns the version at t_{own} (line 14). Note that, if the version has not been committed yet, the server finds it from *pending* (line 12).

Transactional Writes. Both Eiger-NOC2 and Eiger-PORT adopt the variant of two-phase commit in [31]. Eiger-NOC2 further adapts it for one-phase writes and convergence mainly in two ways.

Algorithm 2 The Eiger-NOC2 Algorithm

1: */* Eiger-NOC2 adopts Eiger-PORT's 2PC variant, except*
2: *coord/cohorts (i) return prepared timestamp in addition,*
3: *and (ii) perform commits asynchronously. See [28]. */*

4: **_Partition-side Data Structures & Method_**
5: *vers*: multi-versioned DB $\langle key, value, t_{prep}, t_{com} \rangle$
6: ts_{svr}: latest safe time
7: *pending*: uncommitted write txns $txnid \rightarrow t_{pend}$

8: **procedure** GET(k, gsv, t_{own}, $txnid$)
9: $ver \leftarrow vers[k].at(gsv)$
10: **if** $t_{own} \geq ver.t_{com}$ **then**
11: **if** $txnid \in pending$ **then**
12: **return** $vers[k].at(pending[txnid])$, ts_{svr}
13: **else**
14: **return** $vers[k].at(t_{own})$, ts_{svr}
15: **return** ver, ts_{svr}

16: **_Client-side Data Structures & Methods_**
17: $last[svr]$: last committed timestamp on server *svr*
18: $latestWrite[key]$: $key \rightarrow txnid, t_{own}$

19: **procedure** GET_ALL(K : set of keys)
20: $gsv \leftarrow \min(last)$ *// global safe view*
21: **parallel-for** $k \in K$ **do**
22: $txnid, t_{own} \leftarrow latestWrite[k]$
23: $rs[k], last[svr] \leftarrow$ GET(k, gsv, t_{own}, $txnid$)
24: **return** *rs*

25: **procedure** PUT_ALL(W : set of $\langle key, value \rangle$)
26: $txnid \leftarrow$ generate new transaction ID
27: **parallel-for** $\langle k, v \rangle \in W$ **do**
28: **if** k.server is coordinator **then**
29: $ts_{svr}, t_{prep} \leftarrow$ WRITE_COORD(...) *// see [28]*
30: **else**
31: $ts_{svr}, t_{prep} \leftarrow$ WRITE_COHORT(...) *// [28]*
32: $last[k.server] \leftarrow \max(ts_{svr}, last[svr])$
33: $t_{own} \leftarrow \max(t_{prep}, t_{own})$ *// t_{own} initialized as -1*
34: **for** $k \in W.keySet$ **do**
35: **if** $latestWrite[k].t_{own} < t_{own}$ **then**
36: $latestWrite[k] \leftarrow (txnid, t_{own})$
37: **return**

First, Eiger-NOC2 completes the commit phase asynchronously, allowing any subsequent transactions to race with it. Second, the way a write transaction's commit timestamp is decided by the coordinator is mimicked at the client side, so that clients can safely read prepared-only writes of their own without breaking TCCv.

More specifically, when processing a write transaction, each cohort prepares a version with a timestamp extracted from its Lamport clock. The coordinator chooses the highest timestamp among all the received timestamps as the commit timestamp for the transaction. The cohorts then proceed with the commits asynchronously. Each of the coordinator and cohorts returns its latest safe time ts_{svr}, which the client uses to advance its GSV (line 32), alongside its proposed prepare timestamp t_{prep}. This is in turn used to construct the client's LSV with respect to the transaction (line 36), mimicking how the commit timestamp would be decided on the server side.

6.4 Correctness

Eiger-NOC2 adheres to NOC-NOC's performance criteria by design and improves Eiger-PORT by additionally providing O_W. The reasoning for RA-NOC2 applies to Eiger-NOC2 in general. In particular, to guarantee safe asynchronous commits for O_W, Eiger-NOC2 leverages the LSV that keeps track of a client's own writes *in a causally consistent order* with respect to other writes across the database.

We also prove that Eiger-NOC2 satisfies TCCv. Intuitively, Eiger-NOC2 establishes the causal relations among transactions using Lamport clocks. Moreover, it leverages GSVs to represent safe snapshots of the database, where returning the most recent versions within a GSV guarantees no fractured reads. When a client jumps over the GSV to fetch its own writes, the LSV ensures that the jump is aligned along transactional boundaries and thus satisfies read atomicity. Finally, convergence is achieved, since, by updating their dual views, all clients always share the same total order of versions per key, which is established on the partition via monotonically advancing Lamport clocks. We provide the proof in our technical report [28].

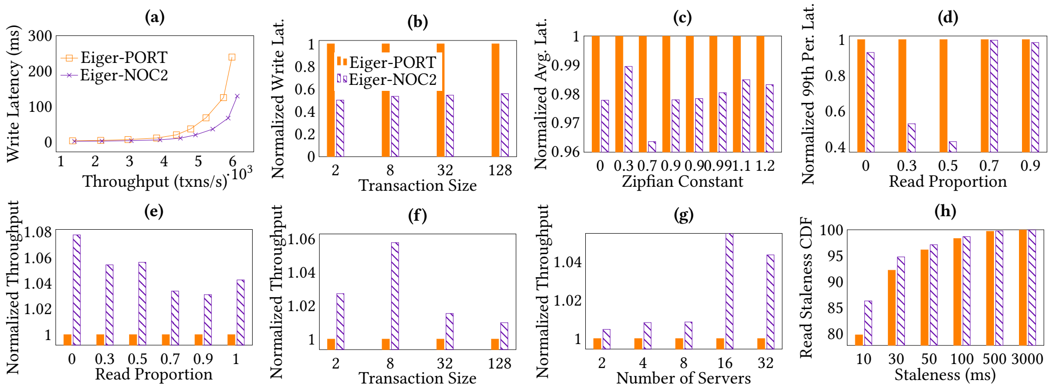


Fig. 9. Performance and data freshness comparisons between Eiger-NOC2 and Eiger-PORT.

7 EIGER-NOC2 EVALUATION

We assess Eiger-NOC2, showing its superior performance and data freshness[6] over the NOCS-optimal Eiger-PORT algorithm as a strong baseline. Note that Eiger-PORT provides a weaker isolation guarantee than Eiger-NOC2.

7.1 Implementation, Setup, and Workloads

Implementation. We build Eiger-NOC2, along with Eiger-PORT, on the RAMP codebase [5]. This consists of around 1250 LOC in Java for each algorithm including data replication, where we reuse RAMP's facilities such as distributed hash table and serializer. To prevent blocking in the traditional 2PC, Eiger-NOC2 utilizes the same 2PC variant [31] as in Eiger-PORT, which ensures that writes always commit (see Section 6.1). Similar to RA-NOC2, the front end of a partition executes client-requested transactions.

Experimental Setup. We run our experiments on an Emulab [51] cluster of machines, each with 2.4GHz Quad-Core Xeon CPU, 12GB RAM, and a 1Gbps network interface. We use the same primary-backup replication setting as in Eiger-PORT [34], with two logical data centers co-located in the cluster. By default, each data center employs eight servers for partitioning the database and eight client machines to load the servers. The database partitioning is fixed once set up. For each data point, we report the average over five trials, each lasting 60 seconds.

Workloads. For a fair comparison, we employ the same YCSB-like dynamic workload generator and match Eiger-PORT's default parameters [34]: 32 threads per client, a Zipfian distribution with a skew factor of 0.8, 1 million keys, 128-byte values, 5 keys per transaction, and 90% reads.

7.2 Evaluation

Summary of Results. Our measurements show that Eiger-NOC2, even with a stronger isolation guarantee, still surpasses Eiger-PORT in all experiments conducted. This demonstrates the *effectiveness of optimizing writes*, even only with O_W, in improving the overall system performance. In particular, Eiger-NOC2 exhibits significantly lower write latency and noticeably higher throughput. Moreover, the 99th percentile latency in Eiger-NOC2 is consistently lower or on par with Eiger-PORT, which demonstrates that the O_W optimization does not introduce extra overhead.

[6]We have also observed that Eiger-NOC2 significantly outperforms Eiger [31]; both algorithms provide the same isolation guarantee. See our technical report [28] for the experimental results.

Eiger-NOC2 also scales well with an increasing number of partitions and is resilient to larger-sized transaction workloads. Finally, it achieves slightly better data freshness results than Eiger-PORT.

Latency Improvement. Figure 9a depicts the write latency as a function of system throughput. Compared to Eiger-PORT, Eiger-NOC2 achieves higher throughput (up to 7%) with the same write latency and lower write latency (up to 48%) with the same throughput. The write latency improvement is significant, around 47%, independent of the transaction size; see Figure 9b. Despite varying skews (Figure 9c) and read/write ratios (Figure 9d), the latency in Eiger-NOC2 is overall lower than Eiger-PORT, and the improvement tends to increase under highly skewed and write-heavy workloads. Eiger-NOC2 owes all these improvements to the O_W optimization.

Throughput Improvement. Overall, Eiger-NOC2 exhibits higher throughput than Eiger-PORT under various workloads. Figure 9e shows that, when writes dominate the workload, the improvement becomes more pronounced. Figure 9f depicts that Eiger-NOC2 consistently outperforms Eiger-PORT regardless of transaction sizes, with up to 7% improvement. As shown in Figure 9g, Eiger-NOC2 scales better when we increase the number of database partitions. All these throughput improvements can be attributed mainly to two factors: (i) the O_W optimization that boosts the overall system performance and (ii) the precise capturing of TCCv snapshots, which reduces the server-side overhead of backward search for safe versions.

Data Freshness. From Figure 9h we can observe that over 86% of the reads in Eiger-NOC2 (slightly more than those in Eiger-PORT) are 10ms staler than up-to-date values, and almost all reads experience less than 500 ms staleness. This is mainly because Eiger-NOC2 (like RA-NOC2) always pushes a client's dual view to the most recent, safe snapshot of the database by synchronizing with the partitions and co-located clients.

8 DISCUSSION

Non-blocking Writes. N_W focuses on non-blocking concurrency control mechanisms. When coupling them with an atomic commitment protocol (ACP) for committing write transactions (NOC-NOC assumes two-phase commit given its wide adoption in practice), a transaction system may not make progress during failures as ACPs are inherently blocking [7] when, e.g., network partitions occur. Many solutions exist in the literature for mitigating this blocking issue. Both Eiger-NOC2 and Eiger-PORT employ the 2PC variant that always commits [31]; see Section 6.1.

RA-NOC2 runs the *cooperative termination protocol* (CTP) [7], which is both lightweight and effective in practice [5]. CTP can always complete a transaction when failures occur during the commit phase and a server has prepared the transaction but times out when waiting for the commit message. Note that, by further leveraging LSVs, the writes of a transaction that are already fully prepared can be safely returned, even before CTP recovers the blocked server.

Overhead of Local Computations. NOC-NOC, like many other design objectives or impossibility results such as SNOW and NOCS, concentrates on communication complexity. However, there may be other factors that negatively affect system performance, such as the overhead of local computations. In particular, even though this is usually negligible compared to network latency, especially in a geo-distributed setting, poor design choices or inefficient implementations could still accumulate system-wide computation overhead, impairing overall system performance. To return read-atomic versions in one round-trip, Eiger-PORT may perform expensive recursive scans of the database. In both of our algorithms, to achieve one round-trip reads and one-phase writes, maintaining dual views across clients may incur extra overhead under extremely skewed workloads, although we have not observed this in practice. Investigating the trade-off between communication complexity and the overhead of local computations is interesting future work.

RA-NOC2 vs Eiger-NOC2. Despite both being NOC-NOC-optimal, the actual implementations of RA-NOC2 and Eiger-NOC2 rely on the aforementioned different mechanisms to mitigate the 2PC blocking issue (for fair comparisons with their respective competitors). For example, the 2PC variant used by Eiger-NOC2 issues more prepare messages during the first phase, which are linear in the size of cohorts. Upon timeout waiting for a commit message, a cohort in this 2PC variant checks the commitment of a transaction T only on the coordinator, while, with CTP, a cohort confirms T's status with any other sibling cohorts involved in T. Although we have observed RA-NOC2's superior performance over Eiger-NOC2,[7] a fair comparison would require re-implementing, e.g., Eiger-NOC2 with CTP, which we leave for future work. Nonetheless, Eiger-NOC2 is computationally more expensive due to the supported *stronger* isolation guarantee. For example, clients need to mimic the computation of commit timestamps on the server side.

Benchmarking with Realistic Workloads. Our evaluations only consider synthetic YCSB-like benchmarks that are widely used by the database community [10, 22, 26, 36, 37, 53]. Although we have experimented with a variety of workload parameters, standard benchmarks such as TPC-C would provide more insights into our proposed optimizations. However, a recent study reveals that TPC-C, essentially as an I/O benchmark, may not be ideal for benchmarking concurrency control algorithms [50]. We could therefore consider benchmarks with realistic workloads (e.g., highly-skewed transactions [11]) which are also suitable for concurrency control, or even design new ones, e.g., by creating hot warehouses in TPC-C with YCSB's Zipfian or hotspot distribution [50]. Ultimately, we could use realistic transaction workloads collected from production systems.

9 RELATED WORK

Improving Existing Algorithms. Through the lens of NOC-NOC, we have examined a collection of read-atomic and causally consistent distributed transaction algorithms, with the focus on the RAMP-family and Eiger-family algorithms. There are many other algorithms in the literature which do not fulfill all the NOC-NOC performance criteria. These include MySQL Cluster [40] for read committed, RAMP with faster commit [5, 29] for read atomicity, COPS [30] and COPS-SNOW [32] for causally consistent read-only transactions (with single-key writes), and a large number of TCCv systems, e.g., GentleRain [16], Cure [2], Contrarian [15], PaRiS [46], and OCC [47]. Note that, as NOC-NOC subsumes both SNOW and NOCS, any transaction algorithms that are suboptimal, or even optimal (e.g., MySQL Cluster is NOCS-optimal; COPS-SNOW is SNOW-optimal), with respect to these two design objectives, can be potentially optimized to achieve better system performance. We showcase the possibilities by two novel algorithms. RA-NOC2 optimizes the SNOW-optimal read-atomic algorithm LORA [27]. Eiger-NOC2 optimizes the Eiger-PORT design, which provides TCC, and our prior design [19], which guarantees TCCv (see [28] for the performance comparison); both designs are NOCS-optimal.

Note also that as TCCv is compatible with NOC-NOC, it is also achievable under NOCS. This resolves the conjecture that TCC is the upper bound of achievable isolation levels for NOCS-optimal read-only transactions in the presence of transactional writes [34].

Algorithms for Stronger Isolation Levels. NOC-NOC is incompatible with parallel snapshot isolation and beyond (Theorem 3.1), while many distributed transaction systems offering stronger isolation guarantees partially meet its criteria. Below we discuss some representative systems. See also Figure 2 for the relationship among their supporting isolation levels. Walter [45], which provides parallel snapshot isolation, can complete its transactions in one round-trip in the best case. However, it has non-constant metadata for reads and fails to meet NOC for writes in general.

[7] The performance comparison result is given in our technical report [28].

Scylla-PORT [34] is NOCS-optimal (satisfying NOC for reads) with process-ordered serializability, but it does not support transactional writes. Spanner-RSS [22], providing regular sequential serializability, improves latency for Spanner's reads by reducing their blocking chances in the presence of conflicting writes. The strictly serializable system NCC [33] exhibits optimal performance for its read-only transactions (i.e., NOC for reads) in the best case; however, it still aborts/retries transactions in general, thereby not satisfying O_R. These three recently-proposed systems focus on optimizing reads. An open research question is whether they can be optimized to achieve NOC-NOC-optimal *best-case* performance.

Design Objectives. Many performance criteria have been proposed for designing highly efficient distributed transactions [3, 14, 15, 24, 32, 34, 49]. These criteria focus on optimizing reads. However, some of them, such as SNOW [24, 32], miss crucial factors such as the metadata size, which also impact system latency and throughput. Moreover, all these criteria, including NOCS, overlook how optimizing writes can potentially improve overall system performance, even under read-heavy workloads. In contrast, our NOC-NOC design objective aims at optimizing both reads and writes.

Some design objectives have stronger data freshness requirements (e.g., minimal progress [14]), thus restricting the achievable combinations of performance criteria [3, 14, 15, 49]. NOC-NOC assumes a weaker freshness criterion, as for SNOW and NOCS, which allows returning stale snapshots. Nonetheless, satisfactory data freshness results can still be achieved in practice, as shown by our evaluation.

10 CONCLUSION

We have proposed the NOC-NOC design objective and established related impossibility results. Examining existing transaction algorithms in the light of NOC-NOC, we have identified a significant gap in the design space. We have therefore designed two algorithms that fulfill all six NOC-NOC criteria. Our evaluation shows their superior system performance and competitive data freshness.

Along with these two case studies, we have presented dual views which, when coupled with version vectors, can be leveraged to design NOC-NOC-optimal transaction algorithms that provide other isolation guarantees, in addition to RA+ and TCCv.

We expect NOC-NOC to help transaction system developers rethink their designs and implementations by accounting for the optimization of writes, avoiding efforts on achieving the impossible, and guiding them to focus on what is actually possible.

DATA AVAILABILITY

Our technical report, prototypes, and experimental data are available at [28].

ACKNOWLEDGMENTS

We thank the reviewers for their valuable feedback. We express our gratitude to Kaile Huang for his assistance in establishing the impossibility result for global visibility. We also extend our gratitude to Haonan Lu for his assistance in facilitating our understanding of Eiger-PORT.

REFERENCES

[1] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. 1995. Causal Memory: Definitions, Implementation, and Programming. *Distributed Comput.* 9, 1 (1995), 37–49.

[2] Deepthi Devaki Akkoorath, Alejandro Z. Tomsic, Manuel Bravo, Zhongmiao Li, Tyler Crain, Annette Bieniusa, Nuno M. Preguiça, and Marc Shapiro. 2016. Cure: Strong Semantics Meets High Availability and Low Latency. In *ICDCS 2016*. IEEE Computer Society, 405–414.

[3] Karolos Antoniadis, Diego Didona, Rachid Guerraoui, and Willy Zwaenepoel. 2020. The Impossibility of Fast Transactions. In *IPDPS'20*. IEEE, 1143–1154.

[4] Masoud Saeida Ardekani, Pierre Sutra, Nuno Preguiça, and Marc Shapiro. 2013. Non-Monotonic Snapshot Isolation. arXiv:1306.3906 [cs.DC]

[5] Peter Bailis, Alan D. Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2016. Scalable Atomic Visibility with RAMP Transactions. *ACM Trans. Database Syst.* 41, 3 (2016), 15:1–15:45.

[6] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. 1995. A Critique of ANSI SQL Isolation Levels. In *SIGMOD'95*. ACM Press, 1–10.

[7] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley.

[8] Eric A. Brewer. 2000. Towards robust distributed systems (abstract). In *PODC*. 7.

[9] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry C. Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkateshwaran Venkataramani. 2013. TAO: Facebook's Distributed Data Store for the Social Graph. In *ATC'13*. USENIX Association, 49–60.

[10] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. 2018. Efficient Distributed Memory Management with RDMA and Caching. *Proc. VLDB Endow.* 11, 11 (2018), 1604–1617.

[11] Audrey Cheng, Xiao Shi, Aaron Kabcenell, Shilpa Lawande, Hamza Qadeer, Jason Chan, Harrison Tin, Ryan Zhao, Peter Bailis, Mahesh Balakrishnan, Nathan Bronson, Natacha Crooks, and Ion Stoica. 2022. TAOBench: An End-to-End Benchmark for Social Network Workloads. *Proc. VLDB Endow.* 15, 9 (2022), 1965–1977.

[12] Audrey Cheng, Xiao Shi, Lu Pan, Anthony Simpson, Neil Wheaton, Shilpa Lawande, Nathan Bronson, Peter Bailis, Natacha Crooks, and Ion Stoica. 2021. RAMP-TAO: Layering Atomic Transactions on Facebook's Online TAO Data Store. *Proc. VLDB Endow.* 14, 12 (2021), 3014–3027.

[13] Khuzaima Daudjee and Kenneth Salem. 2004. Lazy Database Replication with Ordering Guarantees. In *ICDE 2004*. IEEE Computer Society, 424–435.

[14] Diego Didona, Panagiota Fatourou, Rachid Guerraoui, Jingjing Wang, and Willy Zwaenepoel. 2019. Distributed Transactional Systems Cannot Be Fast. In *SPAA'19*. ACM, 369–380.

[15] Diego Didona, Rachid Guerraoui, Jingjing Wang, and Willy Zwaenepoel. 2018. Causal Consistency and Latency Optimality: Friend or Foe? *Proc. VLDB Endow.* 11, 11 (2018), 1618–1632.

[16] Jiaqing Du, Calin Iorgulescu, Amitabha Roy, and Willy Zwaenepoel. 2014. GentleRain: Cheap and Scalable Causal Consistency with Physical Clocks. In *SoCC 2014*. ACM, 4:1–4:13.

[17] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *ATC'19*. USENIX Association, 1–14.

[18] ElectricSQL. Accessed in July, 2023. https://electric-sql.com/.

[19] Shabnam Ghasemirad. 2022. *Mechanized Data Consistency Models for Distributed Database Transactions.* Master's thesis. ETH Zurich. https://doi.org/10.3929/ethz-b-000581334.

[20] Seth Gilbert and Nancy A. Lynch. 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News* 33, 2 (2002), 51–59.

[21] Wojciech Golab, Xiaozhou Li, and Mehul A. Shah. 2011. Analyzing consistency properties for fun and profit. In *PODC'11*. ACM, 197–206.

[22] Jeffrey Helt, Matthew Burke, Amit Levy, and Wyatt Lloyd. 2021. Regular Sequential Serializability and Regular Sequential Consistency. In *SOSP'21*. ACM, 163–179.

[23] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. 2008. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. *Proc. VLDB Endow.* 1, 2 (2008), 1496–1499.

[24] Kishori M. Konwar, Wyatt Lloyd, Haonan Lu, and Nancy A. Lynch. 2021. SNOW Revisited: Understanding When Ideal READ Transactions Are Possible. In *IPDPS'21*. IEEE, 922–931.

[25] Butler Lampson and Howard E. Sturgis. 1979. Crash recovery in a distributed storage system. Xerox Palo Alto Research Center.

[26] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. 2017. Cicada: Dependably Fast Multi-Core In-Memory Transactions. In *SIGMOD '17*. ACM, 21–35.

[27] Si Liu. 2022. All in One: Design, Verification, and Implementation of SNOW-Optimal Read Atomic Transactions. *ACM Trans. Softw. Eng. Methodol.* 31, 3 (2022).

[28] Si Liu, Luca Multazzu, Hengfeng Wei, and David Basin. 2023. Artifact and technical report for "NOC-NOC: Towards Performance-optimal Distributed Transactions". https://github.com/siliunobi/NOC-NOC.

[29] Si Liu, Peter Csaba Ölveczky, Muntasir Raihan Rahman, Jatin Ganhotra, Indranil Gupta, and José Meseguer. 2016. Formal modeling and analysis of RAMP transaction systems. In *SAC'16*. ACM, 1700–1707.

[30] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2011. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *SOSP 2011*. ACM, 401–416.

[31] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2013. Stronger Semantics for Low-Latency Geo-Replicated Storage. In *NSDI'13*. USENIX Association, 313–328.

[32] Haonan Lu, Christopher Hodsdon, Khiem Ngo, Shuai Mu, and Wyatt Lloyd. 2016. The SNOW Theorem and Latency-Optimal Read-Only Transactions. In *OSDI'16*. USENIX Association, 135–150.

[33] Haonan Lu, Shuai Mu, Siddhartha Sen, and Wyatt Lloyd. 2023. NCC: Natural Concurrency Control for Strictly Serializable Datastores by Avoiding the Timestamp-Inversion Pitfall. In *OSDI'23*. USENIX Association, 305–323.

[34] Haonan Lu, Siddhartha Sen, and Wyatt Lloyd. 2020. Performance-Optimal Read-Only Transactions. In *OSDI'20*. USENIX Association, 333–349.

[35] Haonan Lu, Kaushik Veeraraghavan, Philippe Ajoux, Jim Hunt, Yee Jiun Song, Wendy Tobagus, Sanjeev Kumar, and Wyatt Lloyd. 2015. Existential consistency: measuring and understanding consistency at Facebook. In *SOSP 2015*. ACM, 295–310.

[36] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2020. Aria: A Fast and Practical Deterministic OLTP Database. *Proc. VLDB Endow.* 13, 12 (2020), 2047–2060.

[37] Yi Lu, Xiangyao Yu, and Samuel Madden. 2019. STAR: Scaling Transactions through Asymmetric Replication. *Proc. VLDB Endow.* 12, 11 (2019), 1316–1329.

[38] Microsoft. Accessed in July, 2023. Azure Cosmos DB. https://learn.microsoft.com/en-us/azure/cosmos-db/consistency-levels.

[39] MongoDB. Accessed in July, 2023. Read Isolation, Consistency, and Recency. https://docs.mongodb.com/manual/core/read-isolation-consistency-recency/.

[40] MySQL. Accessed in July, 2023. MySQL Cluster CGE. https://www.mysql.com/products/cluster/.

[41] Neo4j. Accessed in July, 2023. https://neo4j.com/docs/operations-manual/current/clustering/introduction/.

[42] Christos H. Papadimitriou. 1979. The Serializability of Concurrent Database Updates. *J. ACM* 26, 4 (1979), 631–653.

[43] Matthieu Perrin, Achour Mostefaoui, and Claude Jard. 2016. Causal Consistency: Beyond Memory. *SIGPLAN Not.* 51, 8, Article 26 (2016), 26:1–26:12 pages.

[44] Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea, Kyle Littlefield, David Menestrina, Stephan Ellner, John Cieslewicz, Ian Rae, Traian Stancescu, and Himani Apte. 2013. F1: A Distributed SQL Database That Scales. *Proc. VLDB Endow.* 6, 11 (2013), 1068–1079.

[45] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. 2011. Transactional storage for geo-replicated systems. In *SOSP'11*. ACM, 385–400.

[46] Kristina Spirovska, Diego Didona, and Willy Zwaenepoel. 2019. PaRiS: Causally Consistent Transactions with Non-blocking Reads and Partial Replication. In *ICDCS 2019*. IEEE, 304–316.

[47] Kristina Spirovska, Diego Didona, and Willy Zwaenepoel. 2021. Optimistic Causal Consistency for Geo-Replicated Key-Value Stores. *IEEE Trans. Parallel Distributed Syst.* 32, 3 (2021), 527–542.

[48] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent B. Welch. 1994. Session Guarantees for Weakly Consistent Replicated Data. In *PDIS*. IEEE Computer Society, 140–149.

[49] Alejandro Z. Tomsic, Manuel Bravo, and Marc Shapiro. 2018. Distributed transactional reads: the strong, the quick, the fresh & the impossible. In *Middleware'18*. ACM, 120–133.

[50] Yang Wang, Miao Yu, Yujie Hui, Fang Zhou, Yuyang Huang, Rui Zhu, Xueyuan Ren, Tianxi Li, and Xiaoyi Lu. 2022. A Study of Database Performance Sensitivity to Experiment Settings. *Proc. VLDB Endow.* 15, 7 (2022), 1439–1452.

[51] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. 2002. An Integrated Experimental Environment for Distributed Systems and Networks. In *OSDI*. USENIX Association, 255–270.

[52] Juncheng Yang, Yao Yue, and K. V. Rashmi. 2021. A Large-scale Analysis of Hundreds of In-memory Key-value Cache Clusters at Twitter. *ACM Trans. Storage* 17, 3 (2021), 17:1–17:35.

[53] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. 2015. Building consistent transactions with inconsistent replication. In *SOSP 2015*. ACM, 263–278.

[54] Fang Zhou, Yifan Gan, Sixiang Ma, and Yang Wang. 2018. wPerf: Generic Off-CPU Analysis to Identify Bottleneck Waiting Events. In *OSDI'18*. USENIX Association, 527–543.

Received 15 July 2023; revised 20 October 2023; accepted 20 November 2023