# NCC: Natural Concurrency Control for Strictly Serializable Datastores by Avoiding the Timestamp-Inversion Pitfall

Haonan Lu, *University at Buffalo;* Shuai Mu, *Stony Brook University;* Siddhartha Sen, *Microsoft Research;* Wyatt Lloyd, *Princeton University*

https://www.usenix.org/conference/osdi23/presentation/lu

This paper is included in the Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-34-2

Open access to the Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation is sponsored by

جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology

# NCC: Natural Concurrency Control for Strictly Serializable Datastores by Avoiding the Timestamp-Inversion Pitfall

Haonan Lu[⋆], Shuai Mu[†], Siddhartha Sen[‡], Wyatt Lloyd[⋄]
[⋆]*University at Buffalo*, [†]*Stony Brook University*,
[‡]*Microsoft Research*, [⋄]*Princeton University*

## Abstract

Strictly serializable datastores greatly simplify application development. However, existing techniques pay unnecessary costs for naturally consistent transactions, which arrive at servers in an order that is already strictly serializable. We exploit this natural arrival order by executing transactions with minimal costs while optimistically assuming they are naturally consistent, and then leverage a timestamp-based technique to efficiently verify if the execution is indeed consistent. In the process of this design, we identify a fundamental pitfall in relying on timestamps to provide strict serializability and name it the timestamp-inversion pitfall. We show that timestamp inversion has affected several existing systems.

We present Natural Concurrency Control (NCC), a new concurrency control technique that guarantees strict serializability and ensures minimal costs—i.e., one-round latency, lock-free, and non-blocking execution—in the common case by leveraging natural consistency. NCC is enabled by three components: non-blocking execution, decoupled response management, and timestamp-based consistency checking. NCC avoids the timestamp-inversion pitfall with response timing control and proposes two optimization techniques, asynchrony-aware timestamps and smart retry, to reduce false aborts. Moreover, NCC designs a specialized protocol for read-only transactions, which is the first to achieve optimal best-case performance while guaranteeing strict serializability without relying on synchronized clocks. Our evaluation shows NCC outperforms state-of-the-art strictly serializable solutions by an order of magnitude on many workloads.

## 1 Introduction

Strictly serializable datastores have been advocated by much recent work [12, 18, 19, 33, 52, 58, 68] because they provide the powerful abstraction of programming in a single-threaded, transactionally isolated environment, which greatly simplifies application development and prevents consistency anomalies [8]. However, only a few concurrency control techniques provide strict serializability and they are expensive.

Common techniques include distributed optimistic concurrency control (dOCC), distributed two-phase locking (d2PL), and transaction reordering (TR). They incur high overheads which manifest in extra rounds of messages, distributed lock management, blocking, and excessive aborts. The validation round in dOCC, required lock management in d2PL, blocking during the exchange of ordering information in TR, and aborts due to conflicts in dOCC and d2PL are examples of these four overheads, respectively. These costs are paid to enforce the two requirements of strict serializability: (1) ensuring there is a total order by avoiding interleaving transactions, and (2) ensuring the *real-time ordering* i.e., later-issued transactions take effect after previously-finished ones. However, we find these costs are unnecessary for many *datacenter workloads* where transactions are executed within a datacenter and then replicated within or across datacenters.

Many datacenter transactions do not interleave: e.g., many of them are dominated by reads [12], and the interleaving of reads returning the same value does not affect correctness. Many of them are short [24, 27, 40, 52, 64, 71], and short lifetimes reduce the likelihood of interleaving. Advances in datacenter networking also reduce variance in delivery times of concurrent requests [5, 14, 22], resulting in less interleaving.

In addition, many datacenter transactions arrive at servers in an order that trivially satisfies their real-time order requirement. That is, a transaction arrives at all participant servers after all previously committed transactions.

Because many transactions do not interleave and their arrival order satisfies the real-time order constraints, intuitively, simply executing their requests in the order servers receive them (i.e., treating them as if they were non-transactional simple operations) will naturally satisfy strict serializability. We call these transactions *naturally consistent*.

Ideally, naturally consistent transactions can be safely executed without any concurrency control, incurring zero costs. However, existing techniques pay unnecessary overheads. For instance, dOCC still requires extra rounds of messages for validation, d2PL still acquires locks, and TR still blocks transactions to exchange ordering information, even if validation always succeeds, locks are always available, and nothing needs to be reordered. Therefore, this paper strives to make naturally consistent transactions as cheap as possible.

In this paper, we present Natural Concurrency Control (NCC), a new concurrency control technique that guarantees strict serializability and ensures minimal costs—i.e., one-round latency, lock-free, and non-blocking execution—in the common case. NCC's design insight is to execute naturally consistent transactions in the order they arrive, as if they were non-transactional operations, while guaranteeing correctness without interfering with transaction execution.

NCC is enabled by three components. *Non-blocking execution* ensures that servers execute transactions in a way that is similar to executing non-transactional operations. *Decoupled response management* separates the execution of requests from the sending of their responses, ensuring that only correct results are returned. *Timestamp-based consistency checking* uses timestamps to verify transactions' results, without interfering with execution.

While designing the consistency-checking component, we identified a correctness pitfall in timestamp-based, strictly serializable techniques. Specifically, these techniques sometimes fail to guard against an execution order that is total but incorrectly inverts the real-time ordering between transactions, thus violating strict serializability. We call this the *timestamp-inversion* pitfall. Timestamp inversion is subtle because it can happen only if a transaction interleaves with a set of *non*-conflicting transactions that have real-time order relationships. The pitfall is fundamental as we find it affects multiple prior systems (TAPIR [71] and DrTM [66]), which, as a result, do not provide strict serializability as claimed.

NCC handles timestamp inversion through response timing control (RTC), an integral part of decoupled response management, without interfering with non-blocking execution or relying on synchronized clocks. NCC proposes two timestamp optimization techniques, asynchrony-aware timestamps and smart retry, to reduce false aborts. Moreover, NCC designs a specialized protocol for read-only transactions, which, to the best of our knowledge, is the first to achieve optimal performance [40] in the best case while ensuring strict serializability, without relying on synchronized clocks.

We compare NCC with common strictly serializable techniques: dOCC, d2PL, and TR, and two serializable protocols, TAPIR [71] and MVTO [55]. We use three workloads: Google-F1, Facebook-TAO, and TPC-C (§6). The Google-F1 and Facebook-TAO workloads synthesize production-like workloads for Google's Spanner [12, 59] and Facebook's TAO [10], respectively. Both workloads are read-dominated. TPC-C [63] consists of few-shot transactions that are write-intensive. We further explore the workload space by varying the write fractions in Google-F1. NCC significantly outperforms dOCC, d2PL, and TR with 2–10× lower latency and 2–20× higher throughput. NCC outperforms TAPIR with 2× higher throughput and 2× lower latency, and closely matches the performance of MVTO.

In summary, this work makes the following contributions:

- Identifies timestamp inversion, a fundamental correctness pitfall in timestamp-based, strictly serializable concurrency control techniques.
- Proposes NCC, a new concurrency control technique that provides strict serializability and achieves minimal overhead in the common case by exploiting natural consistency in datacenter workloads.
- A strictly serializable read-only protocol with optimal best-case performance that does not rely on synchronized clocks.
- An implementation and evaluation that shows NCC outperforms existing strictly serializable systems by an order of magnitude and closely matches the performance of systems that provide weaker consistency.

## 2 Background

This section provides the necessary background on transactional datastores, strict serializability, and general techniques for providing strict serializability.

### 2.1 Transactional Datastores

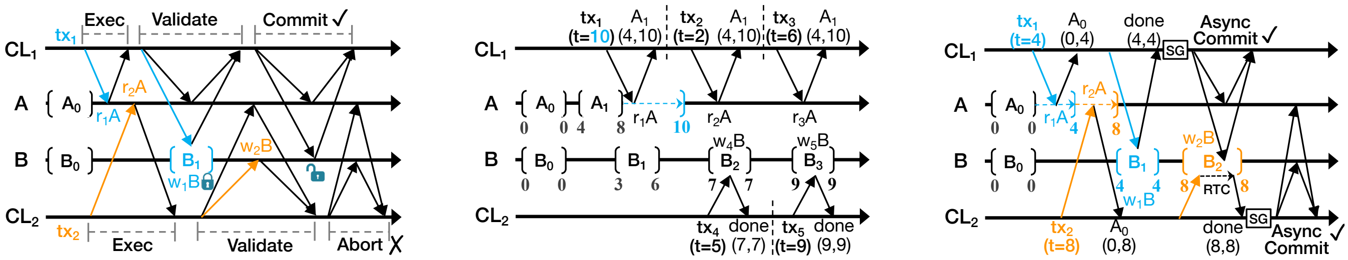
Transactional datastores are the back-end workhorse of many web applications. They typically consist of two types of machines. Front-end *client* machines receive users' requests, e.g., managing a web page, and execute these requests on behalf of users by issuing transactions to the storage *servers* that store the data. Servers are fault-tolerant, e.g., the system state is made persistent on disks and replicated via replicated state machines (RSM), like Paxos [30].

Transactions are managed by coordinators, which can be co-located either with a server or the client. This paper adopts the latter approach to avoid the delays caused by shipping the transaction from the client to a server, while explicitly handling client failures. The coordinator issues read/write operations to relevant servers, called *participants*, following the transaction's logic, which can be *one-shot*, i.e., it knows a priori which data to read/write and can send all requests in one step, or *multi-shot*, i.e., it takes multiple steps as the data read in one step determines which data to read/write in later steps. The system executes transactions following a concurrency control protocol, which ensures that transactions appear to take effect in an order that satisfies the system's consistency requirements. The stronger the consistency provided by the system, the easier it is to develop correct applications.

### 2.2 Strict Serializability

*Strict serializability* [23, 53], also known as external consistency [21], is often considered the strongest consistency model. It requires that (1) there exists a *total order* of transactions, and (2) the total order must respect the *real-time order*, which means if transaction $tx_1$ ends before $tx_2$ starts, then $tx_1$ must appear before $tx_2$ in the total order. As a result, transactions appear to take effect one at a time in the order the system receives them.

**Formal definition.** We use Real-time Serialization Graphs (RSG) [1] to formalize the total order and real-time order requirements. An RSG is a directed graph that captures the order in which transactions take effect. Specifically, two requests from different transactions have an *execution edge* $req_1 \xrightarrow{exe} req_2$ if any of the following happens: $req_1$ creates some data version $v_i$ and $req_2$ reads $v_i$; $req_1$ reads some data version $v_j$ and $req_2$ creates $v$'s next version that is after $v_j$; or




**(a) dOCC falsely aborts $tx_2$** **(b) Timestamp refinement examples** **(c) NCC commits both $tx_1$ and $tx_2$**

**Figure 1: $tx_1$ and $tx_2$ are naturally consistent. dOCC incurs unnecessary validation costs, and $tx_2$ could be falsely aborted due to lock unavailability. NCC can commit both transactions with timestamp pre-assignment, refinement, and the safeguard check (denoted by SG). These techniques are detailed in Section 5.1. Each version in NCC has a $(t_w, t_r)$ pair which is included in server responses. RTC means response timing control, detailed in Section 5.2.**

$req_1$ creates some data version $v_k$ and $req_2$ creates $v$'s next version that is after $v_k$. Two transactions have an execution edge $tx_1 \xrightarrow{\text{exe}} tx_2$ if there exist $req_1$ and $req_2$ from $tx_1$ and $tx_2$, respectively, such that $req_1 \xrightarrow{\text{exe}} req_2$. A chain of execution edges constructs a directed path between two transactions (requests), denoted by $tx_1 \xmapsto{\textbf{exe}} tx_2$ ($req_1 \xmapsto{\textbf{exe}} req_2$), meaning that $tx_1$ ($req_1$) "transitively" affects $tx_2$ ($req_2$) through some intermediary transactions (requests). Two transactions have a *real-time edge* $tx_1 \xrightarrow{\text{rto}} tx_2$ if there is a real-time ordering between $tx_1$ and $tx_2$, meaning that $tx_1$ commits before $tx_2$'s client issues $tx_2$'s first request. In an RSG, vertices are committed transactions, connected by execution and real-time edges.

There exists a total order if and only if transactions do not circularly affect each other. That is, the subgraph that comprises all vertices and only execution edges is acyclic, meaning that the following invariant holds:

**Invariant 1:** $\forall tx_1, tx_2\, (tx_1 \xmapsto{\textbf{exe}} tx_2 \implies \neg(tx_2 \xmapsto{\textbf{exe}} tx_1))$

The (total) execution order respects the real-time order if and only if the execution edges (paths) do not *invert* the real-time edges, meaning that the following invariant holds:

**Invariant 2:** $\forall tx_1, tx_2\, (tx_1 \xrightarrow{\text{rto}} tx_2 \implies \neg(tx_2 \xmapsto{\textbf{exe}} tx_1))$

These invariants correspond to the total order and real-time order requirements, respectively. Therefore, a system is strictly serializable if and only if for any execution it allows, both invariants hold.

By enforcing a total order and the real-time order, strictly serializable systems provide application programmers with the powerful abstraction of programming in a single-threaded, transactionally isolated environment, and thus they greatly simplify application development and eliminate consistency anomalies. For example, if an admin removes Alice from a shared album and then notifies Bob of the change (via a channel external to the system, e.g., a phone call), who then uploads a photo he does not want Alice to see, then Alice must not see Bob's photo, since *remove_Alice* $\xrightarrow{\text{rto}}$ *new_photo*. Such guarantees cannot be enforced by weaker consistency models, e.g., serializability, because they do not enforce the real-time order that is external to the system.

## 2.3 dOCC, d2PL, & Transaction Reordering

Only a few techniques provide strict serializability. The common ones are dOCC, d2PL, and transaction reordering (TR). dOCC and d2PL typically require three round trips, one for each phase: execute, prepare, and commit. In the execute phase, the coordinator reads the data from the servers while writes are buffered locally. d2PL acquires read locks in this phase while dOCC does not. In the prepare phase, the coordinator sends prepare messages and the buffered writes to the participant servers. d2PL locks all participants while dOCC only locks the written data. dOCC must also validate that values read in the execute phase have not changed. If all requests are successfully prepared, i.e., locks are available and/or validation succeeds, the coordinator notifies the participants to commit the transaction and apply the writes; otherwise, the transaction is aborted and retried.

Transaction reordering typically requires two steps. In the first step, the coordinator sends the requests to the servers, which make requests wait while recording their arrival order relative to those of concurrent transactions. This ordering information usually increases linearly in size with respect to the number of concurrent transactions. In the second step, the coordinator collects the ordering information from participants, sorts the requests to eliminate interleavings, and servers execute the transactions in the sorted order.

These techniques are expensive, e.g., they require multiple rounds of messages, locking, waiting, and aborts. We find that these overheads are wasteful for most of the transactions in many datacenter workloads, and this observation has inspired our protocol design.

# 3 Design Insight & Overview

This section explains natural consistency, which inspires our design, and overviews the key design components.

## 3.1 Exploiting Natural Consistency

For many datacenter transactions, simply executing their requests in the order servers receive them, as if they were non-transactional read/write operations, would naturally satisfy

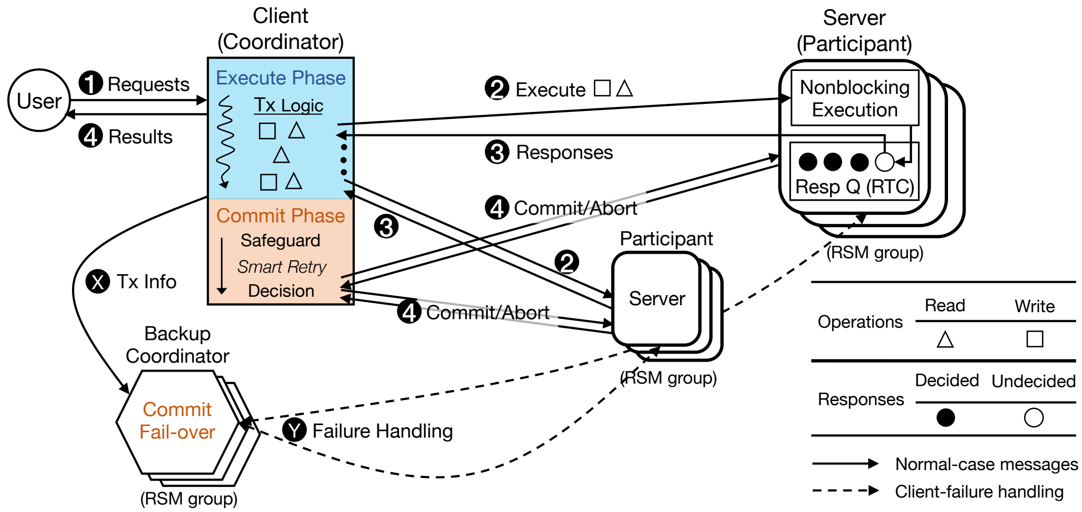



**Figure 2: An overview of system architecture and transaction execution. NCC follows two-phase commit and has three design pillars: non-blocking execution, decoupled response management, and timestamp-based consistency checking.**

strict serializability. In other words, they arrive at servers in an order that is already strictly serializable. We call these transactions *naturally consistent*. Key to natural consistency is the arrival order of transaction requests.

Many requests in datacenter workloads arrive in an order that is total, i.e., transactions do not circularly affect each other, due to the following reasons. First, many requests in real-world workloads are reads [10, 12], and reads do not affect other reads. For instance, reads that return the same value can be executed in any order, and thus servers can safely execute them in their arrival order. Second, many transactions are short, e.g., they are one-shot [24, 27, 40, 52, 64, 71] or can be made one-shot using stored procedures [20, 34, 51, 60, 67], and thus their requests are less likely to interleave with others' requests. Third, advances in datacenter networks reduce the variance of message delivery times [49,50,54], and thus further reduces the likelihood of request interleaving.

In most cases, the (total) arrival order satisfies the real-time order between transactions because a transaction that happens later in real-time, i.e., it starts after another transaction has been committed, must arrive at servers after the committed transaction has arrived.

Ideally, the system would treat naturally consistent transactions as non-transactional operations and execute them in the order they arrive without any concurrency control, while still guaranteeing strict serializability. This insight suggests room for improvement in existing techniques. For instance, dOCC still requires validation messages which are unnecessary when transactions are naturally consistent. Further, during validation between prepare and commit, dOCC has a *contention window* where it can cause other concurrent transactions to abort. As shown in Figure 1a, such contention windows lead to *false aborts*, where a transaction is aborted despite being consistent. Our design aims to minimize costs for as many naturally consistent transactions as possible.

## 3.2 Three Pillars of Design

Our design executes naturally consistent transactions in a manner that closely resembles non-transactional operations. This is made possible through three components.

**Non-blocking execution.** Assuming transactions are naturally consistent, servers execute requests in the order they arrive. Requests are executed "urgently" to completion without acquiring locks, and their results are immediately made visible to prevent blocking subsequent requests. As a result, transactions are executed as cheaply as non-transactional operations, without incurring contention windows.

**Decoupled response management.** Because not all transactions are naturally consistent, servers must prevent returning inconsistent results to clients and ensure there are no cascading aborts. This is achieved by decoupling requests' responses from their execution, with a response sent asynchronously only once it is verified consistent. Inconsistent results are discarded, and their requests are re-executed.

**Timestamp-based consistency checking.** We must check consistency as efficiently as possible, without interfering with server-side execution. We leverage timestamps to capture the arrival order (thus the execution order) of requests and design a client-side checker that verifies if requests were executed in a total order, without incurring overheads such as messages (as in dOCC and TR) or locks (as in dOCC and d2PL).

Figure 2 shows at a high level how these three pillars support our design, and depicts the life cycle of transactions:

❶ The user submits application requests to a client, which translates the requests into transactions.

❷ The (client) coordinator sends operations to the participant servers, following the transaction's logic. The servers execute requests in their arrival order. Their responses are inserted into a queue and sent asynchronously. The responses include timestamps that capture requests' execution order.

❸ Responses are sent to the client when it is safe, determined by response timing control (RTC).

❹ The safeguard checks if transactions were executed in a total order by examining the timestamps in responses. The coordinator sends commit/abort messages to the servers and returns the results of committed transactions to the user in parallel, without waiting for servers' acknowledgments. Ⓧ and Ⓨ explicitly handle client failures by leveraging a server as a backup coordinator.

**Limitations.** First, our design leverages natural consistency, which is observed in short (e.g., one or few shots) datacenter transactions; while our design supports arbitrary-shot transactions, many-shot long-lasting transactions that are more likely to interleave might not benefit from our design. Second, the timestamps associated with each request, including both reads and writes, must be made persistent (e.g., written to disks) and replicated for correctly handling failures, which could lead to replication overhead, which we detail in Section 5.6.

**An observation.** Key to the correctness of our design is leveraging timestamps to verify a total order that respects the real-time order. Yet, we identify a correctness pitfall in relying on timestamps to ensure strict serializability.

## 4 Timestamp-Inversion Pitfall

We discover that timestamp-based techniques sometimes fail to guard against a total order that violates the real-time order in subtle cases. As a result, executing transactions in such a total order inverts the real-time relationship between transactions, which leads to a violation of strict serializability. We call such violations the *timestamp-inversion* pitfall. Figure 3 shows a minimal construction of timestamp inversion using three transactions. $tx_1$ and $tx_2$ are single-machine transactions issued by different clients, and $tx_2$ starts after $tx_1$ finishes, so there exists a real-time order $tx_1 \xrightarrow{\text{rto}} tx_2$ that strict serializability must enforce. $tx_3$ is a multi-shard transaction by a third client that interleaves with $tx_1$ and $tx_2$. $tx_1$, $tx_2$, and $tx_3$ have timestamps 10, 5, and 7, respectively.[1] By following these timestamps, the transactions are executed in a total order denoted as $tx_2 \xrightarrow{\text{exe}} tx_3 \xrightarrow{\text{exe}} tx_1$, which inverts the real-time order $tx_1 \xrightarrow{\text{rto}} tx_2$ and thus violates strict serializability. Specifically, the execution of these transactions violates Invariant 2, subjecting them to consistency anomalies discussed in §2.2.

The timestamp-inversion pitfall is subtle because it happens only if a transaction interleaves with a set of *non*-conflicting transactions that have real-time ordering constraints. We find

[1] A timestamp is generated by either a loosely synchronized physical clock [48] or a causal counter, e.g., a Lamport clock [28].

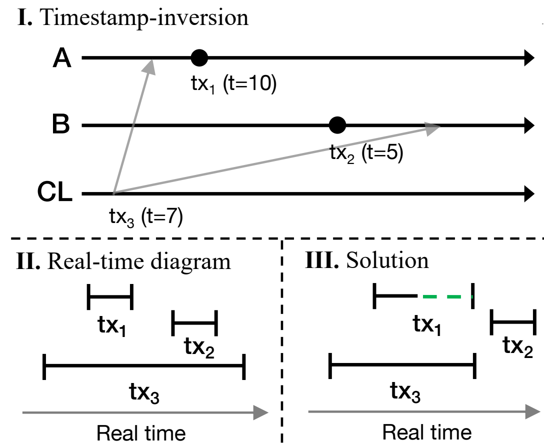



**Figure 3: A minimal example of timestamp inversion, a real-time diagram shows the ordering of transactions, and how NCC tackles the timestamp-inversion pitfall.**

timestamp inversion to be fundamental as it has affected multiple different systems; we discuss two such systems below. In addition, we find that there are several existing systems that do not explicitly define their consistency model, but give a strong indication of providing strict serializability—e.g., they claim invariants that are equivalent to strict serializability, or are built on or evaluated against strictly serializable protocols. We find that these systems also fall into the pitfall.

**Timestamp inversion affects several prior systems.** The minimal example in Figure 3 can be extended to variants of timestamp inversion that affect different types of transactions in real system designs, suggesting that this pitfall is general and fundamental. For instance, we find two systems from recent SOSPs fall into different variants of the pitfall, and thus are not strictly serializable as claimed. We elaborate below to help future work avoid timestamp inversion, and provide the full counterexamples in a technical report [41].

TAPIR [71, 72] is an integrated protocol that co-designs concurrency control and replication. Its concurrency control is a variant of dOCC which validates writes using timestamps without acquiring locks, while reads are validated in the traditional way. Because reads and writes are executed in timestamp order but validated with separate mechanisms, TAPIR's read-write transactions may cause an inversion of concurrent writes. For instance, if $tx_1$, $tx_2$, and $tx_3$ in Figure 3 are read-write transactions, then all three transactions would pass TAPIR's validation, which results in the inversion of $tx_1 \xrightarrow{\text{rto}} tx_2$. The effect of this inversion is perceivable to the client via future reads. This variant of timestamp inversion requires a detailed analysis of the possible executions, showing that none of them are admissible by strict serializability [41].

DrTM [11, 66] is a specialized design for modern datastores equipped with hardware transactional memory and remote direct memory access. DrTM uses timestamps to validate read leases which are acquired before reading the data, a technique equivalent to executing read requests in the timestamp order.

**Algorithm 5.1:** Client (transaction coordinator) logic

```
1  Function ExecuteRWTransaction(tx) :
2      results ← {}; t_pairs ← {} // server responses
3      t.clk ← AsynchronyAwareTS(tx); t.cid ← clientID
4      for req in tx do
           // send requests shot by shot,
           // following tx's logic
5          res, t_pair ← NonblockingExecute(req, t)
6          results ← results ∪ res
7          t_pairs ← t_pairs ∪ t_pair
       // all shots done, tx's logic complete
8      ok, t' ← SafeguardCheck(t_pairs)
9      if not ok then
10         ok ← SmartRetry(tx, t') // §5.4
11     if ok then
12         AsyncCommitOrAbort(tx, "committed")
13         return results
14     else
15         AsyncCommitOrAbort(tx, "aborted")
16         go to 2 // abort, and retry from scratch
17
18 Function SafeguardCheck(t_pairs) :
19     t_w_set ← {}; t_r_set ← {}
20     for t_pair in t_pairs do
21         t_w_set ← t_w_set ∪ t_pair.left
22         t_r_set ← t_r_set ∪ t_pair.right
23     t_w_max ← max{t_w_set}; t_r_min ← min{t_r_set}
24     if t_w_max ≤ t_r_min then
           // t_pairs overlap, ∃ a snapshot
25         return true, t_w_max
26     else
27         return false, t_w_max
```

This makes DrTM's read-only transactions subject to inversion, e.g., when $tx_1$, $tx_2$, and $tx_3$ in Figure 3 are read-write, read-write, and read-only transactions, respectively.

The main contributions of TAPIR and DrTM still stand, just with weaker consistency than claimed. Both teams conjecture that they can fix the systems by using synchronized clocks (e.g., TrueTime [12]) and adapting their designs to use these clocks. Thus, it is likely that their contributions still stand with strict serializability when synchronized clocks are used. However, synchronized clocks require specialized infrastructure and are not generally available (§7). Therefore, NCC is designed to avoid timestamp-inversion without relying on synchronized clocks.

# 5 Natural Concurrency Control

This section presents the basic components of NCC, explains how NCC avoids the timestamp-inversion pitfall, introduces two timestamp optimization techniques and a specialized algorithm for read-only transactions, and concludes with discussions of failure handling and correctness.

## 5.1 Protocol Basics

We build NCC on the three design pillars (§3.2) to minimize the costs for naturally consistent transactions.

**Pre-timestamping transactions.** NCC processes a transaction in two phases: execute and commit. Algorithm 5.1 shows the client (coordinator)'s logic. The coordinator starts a transaction *tx* by pre-assigning it a timestamp *t* that consists of two fields: *clk* which is the client's physical time (Section 5.3 details how it is computed), and *cid* which is the client identifier. *t* uniquely identifies *tx* (line 3). When two timestamps have the same *clk*, NCC breaks the tie by comparing their *cid*. *t* is included in all of *tx*'s requests that are sent to servers shot by shot, following *tx*'s application logic (lines 4 and 5). These timestamps accompany *tx* throughout its life cycle and will be used to verify if the results are consistent.

**Refining timestamps to match execution order.** Algorithm 5.2 details the server-side logic for request execution and commitment. Each key stores a list of versions in the order of the server creating them. A version has three fields: *value*, a pair of *timestamps* $(t_w, t_r)$, and *status*. *value* stores the data; $t_w$ is the timestamp of the transaction that created the version; $t_r$ is the highest timestamp of transactions that read the version; and *status* indicates the state of the transaction that created the version: either (initially) *undecided*, or *committed*. An aborted version is removed from the datastore.

The server always executes a request against the most recent version *curr_ver*, which is either undecided or committed (line 35). Specifically, the server executes a write by creating a new undecided version *new_ver*, which is now the most recent version of the key, ordered after *curr_ver* (lines 39 and 40), and executes a read by reading the *value* of *curr_ver* (line 44). NCC's basic protocol can work with a single-versioned data store while multi-versioning is required only for smart retry, a timestamp optimization technique (§5.4). The server refines the most recent version's timestamp pair to match the order in which requests are executed. Specifically, a write request computes *new_ver*'s $t_w$ as follows: its physical time field is no less than that of the write's timestamp *t* and that of *curr_ver*'s $t_r$, and its client identifier is the same as *t*'s (line 37); *new_ver*'s $t_r$ is initialized to $t_w$ (line 38). Similarly, a read request updates *curr_ver*'s $t_r$ if needed (line 43). Figure 1b shows examples of how timestamps are refined. A version is associated with a $t_w$ and a $t_r$, e.g., $A_1$ initially has a timestamp pair (4, 8). $tx_1$–$tx_3$ are single-key read transactions with pre-assigned timestamps 10, 2, and 6, respectively. They return the most recent version of $A$, i.e., $A_1$, update its $t_r$ if needed, and return $A_1$'s timestamp pair. $tx_4$ and $tx_5$ show how writes manage timestamps.

These (refined) timestamps match requests' arrival order and thus also match the execution order: on each key, a read must have a timestamp greater than that of the write it sees, i.e., a read is ordered after the most recent write, and a write must have a timestamp greater than that of the most recent read, i.e., a write is ordered after the most recent read (and thus all previous writes).

**Non-blocking execution and response queues.** The server executes requests in a non-blocking manner and decouples

**Algorithm 5.2:** Server execution and commitment

```
28 Multi-versioned data store:
29 DS[key][ver]            // indexed by key, vers sorted by t_w
                           // ver is either committed or undecided
30 Response queue:
31 resp_qs[key][resp_q]            // resp queues for each key
32
33 Function NonblockingExecute(req, t) :
34     resp ← [ ] // response message
35     curr_ver ← DS[req.key].most_recent
36     if req is write then
37         t_w.clk ← max{t.clk, curr_ver.t_r.clk + 1}; t_w.cid ← t.cid
38         t_r ← t_w
39         new_ver ← [req.value, (t_w, t_r), "undecided"]
40         DS[req.key] ← DS[req.key] + new_ver
41         resp ← ["done", (t_w, t_r)]
42     else
43         curr_ver.t_r ← max{t, curr_ver.t_r}
44         resp ← [curr_ver.value, (curr_ver.t_w, curr_ver.t_r)]
45     resp_qs[req.key].enqueue(resp, req, t, "undecided")
46     RespTimingControl(resp_qs[req.key]) // §5.2
47
48 Function AsyncCommitOrAbort(tx, decision) :
49     foreach ver created by tx do
50         if decision = "committed" then
51             ver.status ← decision
52         else
53             DS.remove(ver)
54     foreach resp_q in resp_qs do
55         foreach resp in resp_q do
56             if resp.request ∈ tx then
57                 resp.q_status ← decision
58         RespTimingControl(resp_q) // §5.2
```

their execution from responses. Specifically, a write creates a version and immediately makes it visible to subsequent transactions; a read fetches the value of the most recent version whose *status* could be undecided, without waiting for it to commit; the server prepares the response (lines 34, 41, and 44), inserts it into a *response queue* (lines 45 and 46), which asynchronously sends the responses to clients when it is safe. (Section 5.2 details response timing control, which determines when sending a response is safe so timestamp inversion and cascading aborts are prevented.) Unlike d2PL and dOCC, which lock data for at least one round-trip time in the execute and prepare phases (i.e., the contention window), non-blocking execution ensures that a transaction never exclusively owns the data without performing useful work. As a result, the server never stalls, and CPUs are fully utilized to execute requests. Moreover, non-blocking execution eliminates the contention window and thus reduces false aborts.

**Client-side safeguard.** A server response includes the timestamp pair $(t_w, t_r)$ of the most recent version, e.g., *new_ver* for a write and *curr_ver* for a read. The returned $(t_w, t_r)$ represents the time range in which the request is valid. That is, a read must take effect after $t_w$, which is the time when the most recent write on the same key took effect, and no later writes can take effect between $t_w$ and $t_r$ on the same key. A write must have $t_w = t_r$, meaning that it takes effect exactly at $t_w$. When a transaction has completed its logic (i.e., all shots are executed) and the client has received responses to all its requests, the safeguard looks for a consistent snapshot that intersects all $(t_w, t_r)$ pairs in server responses by checking if the $(t_w, t_r)$ pairs overlap (lines 8, 18–27). This intersecting snapshot identifies the transaction's synchronization point, i.e., all requests are valid at the intersecting timestamp.

Figure 1c shows an example where NCC executes the same transactions in Figure 1a. The default versions $A_0$ and $B_0$ both have a timestamp pair $(0, 0)$. $tx_1$ and $tx_2$ are pre-assigned 4 and 8, respectively, and their requests arrive in the same order as they were in Figure 1a. The safeguard enables NCC to commit both transactions, i.e., $tx_1$'s responses intersect at 4 while $tx_2$'s intersect at 8, without unnecessary overhead such as dOCC's validation cost and false aborts.

When the client has decided to commit or abort the transaction, the protocol enters the commit phase by sending the commit/abort messages to the servers. If the transaction is committed, the server updates the *status* of the created versions from undecided to committed; otherwise, the versions are deleted (lines 48–53). The client retries the aborted transaction. The client sends the results of the committed transaction to the user in parallel with the commit messages, i.e., asynchronous commit, without waiting for servers' acknowledgments (lines 11–16).

**Supporting complex transaction logic.** NCC supports transactions accessing a key multiple times, e.g., read-modify-writes and repeated reads/writes, by treating its requests to the same key as a single logical request. For instance, if a read-modify-write has its read and write requests executed consecutively (i.e., they are not intersected by other writes), then only the write response is checked by the safeguard, treating read-modify-write as one logical request; otherwise, it is aborted if there are intersecting writes, e.g., when the most recent version has a $t_w$ greater than that returned by the read of this read-modify-write. The responses of these requests are grouped together in the response queue, e.g., the write response of a read-modify-write is inserted right after the read response of the same read-modify-write. We explain the details of handling complex logic in the technical report [41].

NCC achieves minimal costs by urgently executing transactions in a non-blocking manner and by ensuring a total order with the light-weight timestamp-based safeguard. Yet, in order to provide strict serializability, NCC must enforce the real-time order between transactions by handling the timestamp-inversion pitfall, as we discuss next.

## 5.2 Response Timing Control

NCC avoids the timestamp-inversion pitfall by disentangling the subtle interleaving between a set of non-conflicting transactions that have real-time order dependencies (e.g., Figure 3), without relying on synchronized clocks. Specifically, NCC introduces *response timing control* (RTC), which controls the sending time of responses. It is safe to send the response of a request $req_1$ when the following dependencies are satisfied:

- $D_1$ If $req_1$ reads a version created by $req_0$ of another transaction, then $req_1$'s response is not returned until $req_0$ is committed or it is discarded if $req_0$ is aborted (then $req_1$ will be re-executed).
- $D_2$ If $req_1$ is a write and there are reads that read the version which immediately precedes the one created by $req_1$, then $req_1$'s response is not returned until the reads are committed/aborted.
- $D_3$ If $req_1$ creates a version immediately after the version created by $req_0$ of another transaction, then $req_1$'s response is not returned until $req_0$ is committed/aborted.

By enforcing these dependencies, NCC controls the sending of responses so that the transactions which form the subtle interleaving are forced to take effect in their real-time order. For instance, in Figure 3, server *A* cannot send the response of $tx_1$ until $tx_3$ has been committed (assuming at least one of them writes to *A*). As a result, any transaction $tx_2$ that begins after $tx_1$ receives its response, i.e., $tx_1 \xrightarrow{\text{rto}} tx_2$, must be executed after $tx_1$, and thus after $tx_3$ as well: $tx_2$'s execution on each server is after it begins, which is after $tx_1$ ends, which is after $tx_1$'s response is sent, which is after $tx_3$ commits, which is after $tx_3$ executes on each server. This results in a total order $tx_3 \xrightarrow{\text{exe}} tx_1 \xrightarrow{\text{exe}} tx_2$, which respects the real-time order, enforcing Invariant 2, as shown in Part III of Figure 3.

NCC implements RTC by managing response queues, independently from request execution. NCC maintains one queue per key. A queue item consists of four fields: *response* that stores the response message of a request, the *request* itself, *ts* which is the pre-assigned timestamp of the request, and *q_status* that indicates the state of the request, which is initially *undecided*, and updated to either *committed* or *aborted* when the server receives the commit/abort message for this request (lines 54–57, Algorithm 5.2).

**Managing response queues.** Algorithm 5.3 details how NCC manages the response queue of each key. This logic is invoked every time the server finishes executing a request (line 46) and receives a commit/response message (line 58). NCC iterates over the queue items from the head (i.e., the oldest response) until it finds the first response whose *q_status* is undecided, which means all earlier requests on the same key have been committed or aborted, i.e., this response has satisfied the three dependencies (lines 60–62 and 71). The server sends this response message to the client if it has not done so (lines 72, 74–77). If this is a read response, then the server sends all consecutive read responses that follow it (lines 73 and 78–81), because all these read responses satisfy the three dependencies. In other words, reads returning the same value do not have dependencies between them. RTC is *effectively* similar to locking the response queues, e.g., the queue is "locked" when a response is sent and other responses must wait, and is "unlocked" when the commit/abort message for the request to which the sent response belongs is received. However, RTC differs from lock-based mechanisms in that it is decoupled from execution and does not introduce contention windows, i.e., data objects are not locked.

**Algorithm 5.3:** Response timing control

```
59 Function RespTimingControl(resp_q) :
60     head ← resp_q.head() // the oldest response
61     while head.q_status ≠ "undecided" do
           // find the first response we can send
62         resp_q.dequeue()
63         new_head ← resp_q.head()
64         new_req ← new_head.request; t ← new_head.ts
65         while head.q_status = "aborted"
66         and head.request is write and new_req is read do
               // handle reads seeing aborted writes
67             resp_q.dequeue() // discard read response
               // re-execute the read locally
68             NonblockingExecute(new_req, t)
69             new_head ← resp_q.head()
70             new_req ← new_head.request; t ← new_head.ts
71         head ← resp_q.head()
72     curr_item ← head
73     repeated loop
           // send dependency-satisfied responses
74         resp ← curr_item.response
75         if resp.is_sent ≠ true then
76             sys_call.send(resp) // send to client
77             resp.is_sent ← true
           // send consecutive read responses
78         next_item ← curr_item.next()
79         if curr_item.request is not read
80         or next_item.request is not read then
81             break repeated loop
82         curr_item ← next_item
```

**Fixing reads locally.** When the server receives an abort message for a write request, it must invalidate the responses of any reads that have fetched the value of the aborted write. This is necessary to avoid returning invalid results to the client and to prevent cascading aborts. Specifically, the server removes the response of such a read from the response queue and re-executes the read request, e.g., it fetches the current most recent version, prepares a new response, and inserts the new response to the tail of the queue (lines 65–68).

**Avoiding indefinite waits.** To avoid responses from circularly waiting on dependencies across different keys, NCC early aborts a request (thereby aborting the transaction to which it belongs) if its pre-assigned timestamp is not the highest the server has seen *and* if its response cannot be sent immediately, i.e., it is not the head of the queue. Specifically, a write (read) is aborted if there is an undecided request (write request) with a higher timestamp. Then, the server sends a special response to the client without executing the request. The special response includes a field *early_abort* which allows the client to bypass the safeguard and abort the transaction. We omit the details from the pseudocode for clarity.

RTC is a general solution to timestamp inversion, without the need for synchronized clocks. It does not incur more aborts even when responses are not sent immediately, because response management is decoupled from request execution. That is, whether a transaction is committed or aborted is solely based on timestamps, and RTC does not affect either pre-assignment or refinement of timestamps. Yet, NCC's performance also depends on how well timestamps capture the arrival order of (naturally consistent) transactions. That is, timestamps that do not match transactions' arrival order could cause transactions to falsely abort even if they are naturally consistent. Next, we will discuss optimization techniques that enable timestamps to better match the arrival order.

## 5.3 Asynchrony-Aware Timestamps

NCC proposes two optimizations: a proactive approach that controls how timestamps are generated before transactions start, and a reactive approach that updates timestamps to match the naturally consistent arrival order after requests are executed. This subsection discusses the proactive approach.

The client pre-assigns the same timestamp to all requests of a transaction; however, these requests may arrive at their participant servers at different physical times, which could result in a mismatch between timestamp and arrival order, as shown in Figure 4a. Transactions $tx_1$ and $tx_2$ start around the same time and thus are assigned close timestamps, e.g., $t_1 = 1004$ and $t_2 = 1005$, respectively (client IDs are omitted). Because the latency between $B$ and $CL_1$ is greater than that between $B$ and $CL_2$, $tx_1$ may arrive at $B$ later than $tx_2$, but $tx_1$ has a smaller timestamp. As a result, the safeguard may falsely reject $tx_1$, e.g., server $B$ responds with a refined timestamp pair (1006, 1006) which does not overlap with (1004, 1004), the timestamp pair returned by server $A$. However, aborting $tx_1$ is unnecessary because $tx_1$ and $tx_2$ are naturally consistent.

To tackle this challenge, NCC generates timestamps while accounting for the time difference, $t_\Delta$, between when a request is sent by the client and when the server starts executing the request. Specifically, the client records the physical time $t_c$ before sending the request to the server; the server records the physical time $t_s$ before executing the request and piggybacks $t_s$ onto the response sent back to the client; and the client calculates $t_\Delta$ by finding the difference between $t_c$ and $t_s$, i.e., $t_\Delta = t_s - t_c$. By measuring the end-to-end time difference, $t_\Delta$ effectively masks the impact of queuing delays and clock skew. The client maintains a $t_\Delta$ for each server it has contacted. An asynchrony-aware timestamp is generated by adding the client's current physical time and the greatest $t_\Delta$ among the servers this transaction will access. For instance, given the values of $t_\Delta$ shown in Figure 4a, $CL_1$ assigns $tx_1$ timestamp 1014 (i.e., $1004+10$) and $CL_2$ assigns $tx_2$ 1010 (i.e., $1005+5$), and both transactions may successfully pass their safeguard check, capturing natural consistency.

**Algorithm 5.4:** Smart retry

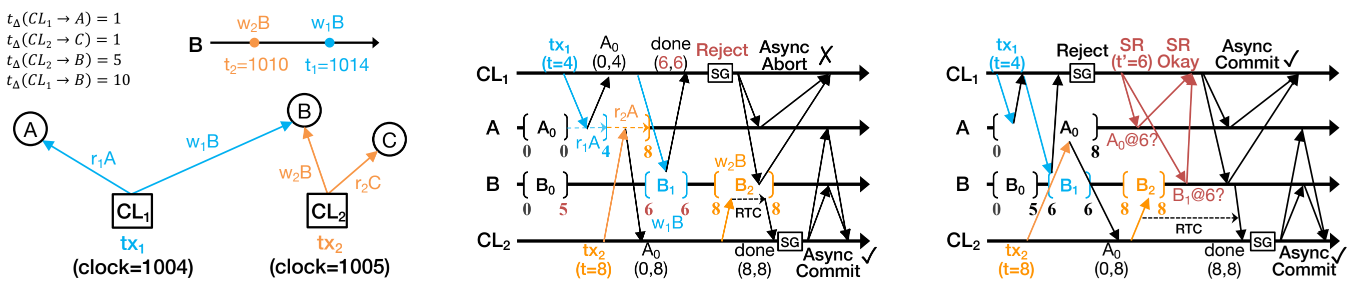
83 **Function** SMARTRETRY(*tx*, $t'$) **:**
84   **foreach** *ver* **accessed by** *tx* **do**
    `// next version of the same key`
85     $next\_ver \leftarrow ver$.next()
86     **if** $next\_ver.t_w \leq t'$ **then**
87       **return** *false*
88     **if** *ver* **created by** *tx* ***and*** $ver.t_w \neq ver.t_r$ **then**
89       **return** *false*
90     **if** *ver* **created by** *tx* **then**
91       $ver.t_w \leftarrow t'$; $ver.t_r \leftarrow t'$
92     **else**
93       $ver.t_r \leftarrow max\{ver.t_r, t'\}$
94   **return** *true*

## 5.4 Smart Retry

NCC proposes a reactive approach to minimizing the performance impact of the safeguard's false rejects, which happen when timestamps fail to identify the naturally consistent arrival order, as shown in Figure 4b. Initially, version $A_0$ has a timestamp pair $(0, 0)$, and $B_0$ has $(0, 5)$. The same transactions $tx_1$ and $tx_2$ as those in Figure 1c access both keys. Following NCC's protocol, $tx_1$'s responses contain the timestamp pairs $(0, 4)$ and $(6, 6)$ from $A$ and $B$, respectively, which will be rejected by the safeguard because they do not overlap. However, aborting $tx_1$ is unnecessary because $tx_1$ and $tx_2$ are naturally consistent.

Instead, NCC tries to "reposition" a rejected transaction with respect to the transactions before and after it to construct a total order, instead of aborting and re-executing the rejected transaction from scratch, which would waste all the work the server has done for executing it. Specifically, NCC chooses a timestamp that is nearest "in the future" and hopes the rejected transaction can be re-committed at that time. This is possible if the chosen time has not been taken by other transactions.

Algorithm 5.4 shows the pseudocode for smart retry. When the transaction fails the safeguard check, NCC suggests a new timestamp $t'$, which is the maximum $t_w$ in the server responses. The client then sends smart retry messages that include $t'$ to the participant servers, which then attempt to reposition the transaction's requests at $t'$. The server can reposition a request if there has not been a newer version that was created before $t'$ (lines 85–87) and, if the request is a write, the version it created has not been read by any transactions (lines 88 and 89). The server updates the timestamps of relevant versions if smart retry succeeds, e.g., the created version has a new timestamp pair $(t', t')$, and $t_r$ of the read version is updated to $t'$ if $t'$ is greater (lines 90–93). (Our implementation does not smart-retry the request that returned the maximum $t_w$, i.e., $t_w = t'$, because its smart retry always succeeds.) The client commits the safeguard-rejected transaction if all its smart retry requests succeed, and aborts and retries it from scratch otherwise (lines 9 and 10, Algorithm 5.1).




**(a) Asynchrony-aware timestamps** **(b) Safeguard false rejects** **(c) Smart retry reduces false rejects**

**Figure 4: Optimizations that match the timestamps with transactions' arrival order. Asynchrony-aware timestamps proactively controls the pre-assigned timestamps before execution. Smart retry reactively fixes the safeguard's false rejects after execution thus avoids aborting and re-executing transactions.**

Not only does smart retry avoid false aborts, it also unleashes a higher degree of concurrency, as shown in Figure 4c. The servers have executed a newer transaction $tx_2$ when $tx_1$'s smart retry (SR) messages arrive, and both transactions can be committed even if the messages interleave, e.g., $tx_1$'s smart retry succeeds and $tx_2$ passes its safeguard check, because $tx_2$'s pre-assigned timestamps have left enough room for repositioning $tx_1$'s requests. In contrast, validation-based techniques would unnecessarily abort $tx_1$ (considering SR as dOCC's validation messages) due to the presence of the conflicting transaction $tx_2$.

**Garbage collection.** Old versions are temporarily stored and garbage collected as soon as they are no longer needed by undecided transactions for smart retry. Only the most recent versions are used to serve new transactions.

## 5.5 Read-Only Transactions

NCC designs a specialized read-only transaction protocol for read-dominated workloads [10, 12, 26, 40, 44]. Similar to existing works, NCC optimizes read-only transactions by eliminating their commit phase because they do not modify the system state and have nothing to commit. By eliminating commit messages, read-only transactions achieve *optimal performance* in the best case, i.e., one round of non-blocking messages with constant metadata [40, 42, 43].

Eliminating commit messages brings a new challenge to response timing control: write responses can no longer track their dependencies on preceding read-only transactions, as they do not know if and when those reads are committed/aborted. To tackle this challenge, NCC aborts a read-only transaction if it could possibly cause the subtle interleaving that leads to timestamp inversion. In other words, NCC commits a read-only transaction if its requests arrive in a naturally consistent order and no intervening writes have been executed since the last time the client accessed these servers.

Specifically, each client tracks $t_{ro}$ which is the $t_w$ of the version created by the most recent write on a server, and the client maintains a map of $t_{ro}$ for each server this client has contacted. A read-only transaction is identified by a Boolean field *IS_READ_ONLY*. The client sends each of its requests to the participant server together with the pre-assigned timestamp (as in the basic protocol) and the $t_{ro}$ of the server. To execute a read request, the server checks the version at $t_{ro}$. If the version is still the most recent, the server continues to execute the read following the basic protocol, e.g., it fetches the most recent version, refines its $t_r$ if needed, and returns its timestamp pair; otherwise, the server sends a special response that contains a field *ro_abort* immediately without executing the request. If any of the responses contain *ro_abort*, the client aborts this read-only transaction; otherwise, the client continues with the safeguard check and, if needed, smart retry, after which the client does not send any commit/abort messages.

This protocol pays more aborts in the worst case in exchange for reduced message overhead in the normal case, a trade-off that is worthwhile for read-dominated workloads where writes are few so aborts are rare, and read-only transactions are many so the savings in message cost are significant. This protocol also expedites the sending of responses for read-write transactions because read-only transactions do not insert responses into the response queue, i.e., a write response depends only on the reads of preceding read-write transactions in Dependency $D_2$, not those of read-only transactions.

## 5.6 Failure Handling

**Tolerating server failures.** NCC assumes servers never fail as their state is typically made persistent on disks and replicated via state machine replication such as Paxos [29]. All state changes incurred by a transaction in the execute phase (e.g., $t_w$ and $t_r$ of each request) must be written to the disk and replicated for correctness. For instance, after a request is executed, the server inserts its response into the response queue and, in parallel, writes the state changes to the disk and replicates the request to other replicas. Its response is sent back to the client when it is allowed by response timing control and when its replication is finished. Commit/abort and smart retry messages are also made persistent and replicated. This simple scheme ensures correctness but incurs high overhead. We plan to investigate possible optimizations in future work, e.g., NCC could defer disk writes and replication to the last shot of a transaction where all state changes are made persistent and replicated once and for all, without having to replicate each request separately. Server replication inevitably increases latency but does not introduce more aborts, because whether a transaction is committed or aborted is solely based on its timestamps, which are decided during request execution and before replication starts.

**Tolerating client failures.** NCC must handle client failures explicitly because clients are not replicated in most systems and NCC co-locates coordinators with clients. NCC adopts an approach similar to that in Sinfonia [4] and RIFL [31]. We briefly explain it as follows. For a transaction *tx*, one of the storage servers *tx* accesses is selected as the backup coordinator, and the other servers are cohorts. In the last shot of the transaction logic, which is identified by a field *IS_LAST_SHOT* in the requests, the client notifies the backup coordinator of the identities of the complete set of cohorts. Cohorts always know which server is the backup coordinator. When the client crashes, e.g., is unresponsive for a certain amount of time, the backup coordinator reconstructs the final state of *tx* by querying the cohorts for how they executed *tx*, and commits/aborts *tx* following the same safeguard and smart retry logic. Because computation is deterministic, the backup coordinator always makes the same commit/abort decision as the client would if the client did not fail. To tolerate one client failure, NCC needs one backup coordinator which is a storage server replicated in a usual way.

## 5.7 Correctness

This section provides proof intuition for why NCC is safe and live. At a high level, NCC guarantees a total order, the real-time order, and liveness, with the mechanisms ($M_1$) the safeguard, ($M_2$) non-blocking execution with response timing control, and ($M_3$) early aborts, respectively. We provide a formal proof of correctness in a technical report [41].

**NCC is safe.** We prove that NCC guarantees strict serializability by demonstrating that both Invariants 1 and 2 are upheld. These two invariants correspond to the total order and real-time order requirements, respectively.

Intuitively, NCC commits all requests of a transaction at the same synchronization point, which is the intersection of all $(t_w, t_r)$ pairs in responses, and the synchronization points of all committed transactions construct a total order. Specifically, we prove that the safeguard enforces Invariant 1, by contradiction. Assume both $tx_1$ and $tx_n$ are committed, and $tx_1 \overset{\textbf{exe}}{\longmapsto} tx_n \overset{\textbf{exe}}{\longmapsto} tx_1$. Without loss of generality, there must exist a chain of transactions such that $tx_1 \xrightarrow{\text{exe}} tx_2 \xrightarrow{\text{exe}} \ldots \xrightarrow{\text{exe}} tx_n \xrightarrow{\text{exe}} tx_1$. Then, each transaction may have two requests, *req* and *req′*, such that $req'_1 \xrightarrow{\text{exe}} req_2$, $req'_2 \xrightarrow{\text{exe}} req_3, \ldots, req'_{n-1} \xrightarrow{\text{exe}} req_n$, $req'_n \xrightarrow{\text{exe}} req_1$. Consider their returned timestamps, we can derive the following:

① $t'_{r1} \leq t_{w2}, t'_{r2} \leq t_{w3}, \ldots, t'_{rn} \leq t_{w1}$, by NCC's protocol.

② $t_{w1} \leq t'_{r1}, t_{w2} \leq t'_{r2}, \ldots, t_{wn} \leq t'_{rn}$, because all transactions are committed and by the safeguard logic.

③ $t_{w1} \leq t'_{r1} \leq t_{w2} \leq t'_{r2} \leq \ldots \leq t_{wn} \leq t'_{rn} \leq t_{w1}$, by ① and ②.

④ $t'_{r1} = t_{w2} = t'_{r2} = t_{w3} = \ldots = t_{wn} = t'_{rn} = t_{w1}$, by ③.

⑤ $req'_i$ is a write and $req_i$ is a read, $i \in [1, n]$, by ④, NCC's protocol, and $tx_1 \xrightarrow{\text{exe}} tx_2 \xrightarrow{\text{exe}} \ldots \xrightarrow{\text{exe}} tx_n \xrightarrow{\text{exe}} tx_1$.

⑥ $t_{w2} = t'_{w1}$ and $t_{w1} = t'_{wn}$, by ⑤ and NCC's protocol.

⑦ $t'_{w1} = t'_{wn}$, by ④ and ⑥, which contradicts that writes from different transactions must have distinct $t_w$ because timestamps are unique. Therefore, Invariant 1 holds.

We prove that NCC enforces Invariant 2 by considering two cases while assuming $tx_1 \xrightarrow{\text{rto}} tx_2$. In case 1, $tx_1$ and $tx_2$ access some common data items. Then, we must have $tx_1 \overset{\textbf{exe}}{\longmapsto} tx_2$, because NCC executes requests in their arrival order. Then, it must be true that $\neg(tx_2 \overset{\textbf{exe}}{\longmapsto} tx_1)$, by Invariant 1. In case 2, $tx_1$ and $tx_2$ access disjoint data sets, and we prove the claim by contradiction. Assume $tx_2 \overset{\textbf{exe}}{\longmapsto} tx_1$, then there must exist $req_2$ and $req_1$ in $tx_2$ and $tx_1$, respectively, such that $req_2 \overset{\textbf{exe}}{\longmapsto} req_1$. $req_1$'s response is not returned until $req_2$ is committed or aborted, by applying response timing control transitively (§5.2). Then, $req_2$ is issued before $req_1$'s client receives $req_1$'s response because a request, e.g., $req_2$, can be committed or aborted only after it is issued and executed. Thus, we can derive $\neg(tx_1 \xrightarrow{\text{rto}} tx_2)$ because $tx_2$ has at least one request, e.g., $req_2$, which starts before $tx_1$ receives all its responses. This means $tx_2$ starts before $tx_1$ is committed, which contradicts our assumption $tx_1 \xrightarrow{\text{rto}} tx_2$. Therefore, Invariant 2 must hold.

**NCC is live.** NCC's non-blocking execution guarantees that requests always run to completion, i.e., execution never stalls (§5.1). Blocking can happen only to the sending of responses due to response timing control, and NCC avoids circular waiting with early aborts (§5.2). Thus, NCC guarantees that transactions finish eventually.

NCC's specialized read-only transaction protocol and optimization techniques such as asynchrony-aware timestamps and smart retry do not affect correctness, because transactions are protected by the three mechanisms (i.e., $M_1$, $M_2$, and $M_3$ summarized at the beginning of this subsection) regardless of whether optimizations or the specialized protocol are used.

# 6 Evaluation

This section answers the following questions:

1. How well does NCC perform, compared to common strictly serializable techniques dOCC, d2PL, and TR?
2. How well does NCC perform, compared to state-of-the-art serializable (weaker consistency) techniques?
3. How well does NCC recover from client failures?

**Implementation.** We developed NCC on Janus's framework [52]. We improved the framework by making it support multi-shot transactions, optimizing its baselines, and adding more benchmarks. NCC's core protocols have ∼3 K lines of C++ code. We also show the results of NCC-RW, a version

| Workload | Write fraction | Assoc-to-obj | # keys/RO | # keys/RW | Value size | # cols/key | Zipfian |
|---|---|---|---|---|---|---|---|
| Google-F1 | 0.3% [0.3%–30%] | — | 1–10 | 1–10 | 1.6 KB ± 119 B | 10 | 0.8 |
| Facebook-TAO | 0.2% | 9.5 : 1 | 1–1 K | 1 | 1–4 KB | 1–1 K | 0.8 |
| TPC-C | New-Order | Payment | Delivery | Order-Status | Stock-Level | Dist/WH | WH/svr |
| | 44% | 44% | 4% | 4% | 4% | 10 | 8 |

**Figure 5: Workload parameters. RO and RW mean read-only and read-write transactions, respectively. TPC-C has a scaling factor of 10 districts per warehouse and 8 warehouses per server.**

| Workload | Contention | # shots | Characteristics | NCC takeaway |
|---|---|---|---|---|
| Facebook-TAO | Low | 1 | Read-dominated | Performance-optimal reads by the RO protocol |
| Google-F1 | Low | 1 | Read-dominated | Performance-optimal reads by the RO protocol |
| TPC-C | Medium→High | Multi-shot | Write-intensive | Leverages the natural arrival order, minimizes false aborts |
| Google-WF | Low→High | 1 | Write-intensive | Leverages the natural arrival order, minimizes false aborts |

**Figure 6: Facebook-TAO and Google-F1 have low contention. TPC-C and Google-WF (varying write fractions) are write-intensive. TPC-C Payment and Order-Status are multi-shot.**

without the read-only transaction protocol, i.e., all transactions are executed as read-write transactions.

**Baselines.** The evaluation includes three strict serializable baselines (dOCC, d2PL, and Janus) and two serializable baselines (MVTO and TAPIR). We chose d2PL and dOCC because they are the most common strictly serializable techniques. We chose Janus because it is the only open-source TR-based strictly serializable system we could find. We chose MVTO because it has the highest best-case performance among all (weaker) serializable techniques, presenting a performance upper bound. We chose TAPIR because it utilizes timestamp-based concurrency control.

Our evaluation focuses on concurrency control and assumes servers never fail. Janus and TAPIR are unified designs of the concurrency control and replication layers, so we disabled their replication and only compare with their concurrency control protocols, shown as Janus-CC and TAPIR-CC, to make the comparisons fair. We compare with two variants of d2PL. d2PL-no-wait aborts a transaction if the lock is not available. d2PL-wound-wait makes the transaction wait if it has a larger timestamp and aborts the lock-holding transaction otherwise. All baselines are fully optimized: we co-locate coordinators with clients (even if baselines cannot handle client failures), combine the execute and prepare phases for d2PL-no-wait and TAPIR-CC, and enable asynchronous commitment, i.e., the client replies to the user without waiting for the acknowledgments of commit messages.

## 6.1 Workloads and Experimental Setup

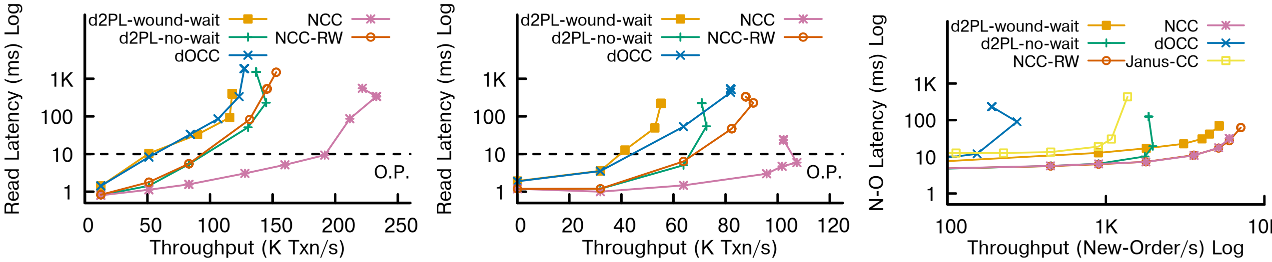
We evaluate NCC under three workloads that cover both read-dominated "simpler" transactions and many-write more "complex" transactions. Google-F1 and Facebook-TAO synthesize real-world applications and capture the former: they are one-shot and read-heavy. TPC-C has multi-shot transactions and is write-intensive, capturing the latter. We also vary write fractions in Google-F1 to further explore the latter. Table 5 shows the workload parameters.

Google-F1 parameters were published in F1 [59] and Spanner [12]. Facebook-TAO parameters were published in TAO [10]. TPC-C's New-Order, Payment, and Delivery are read-write transactions. Its Order-Status and Stock-Level are read-only. Janus's original implementation of TPC-C is one-shot, so we modified it to make Payment and Order-Status multi-shot, to demonstrate NCC is compatible with multi-shot transactions and evaluate its performance beyond one-shot transactions (though they are still relatively short).

**Experimental setting.** We use Microsoft Azure [47]. Each machine has 4 CPUs (8 cores), 16 GB memory, and a 1 Gbps network interface. We use 8 machines as servers and 16–32 machines as clients that generate open-loop requests to saturate the servers. (The open-loop clients back off when the system is overloaded to mitigate queuing delays.) Google-F1 and Facebook-TAO have 1 M keys, with the popular keys randomly distributed to balance load. We run 3 trials for each test and 60 seconds for each trial. Experiments are CPU-bound (i.e., handling network interrupts).

## 6.2 Result Overview

NCC outperforms strictly serializable protocols dOCC, d2PL, and TR (Janus-CC) by 80%–20× higher throughput and 2–10× lower latency under various workloads (Figure 7) and write fractions (Figure 8a). NCC outperforms and closely matches serializable systems, TAPIR-CC and MVTO, respectively (Figure 8b). NCC recovers from client failures with minimal performance impact (Figure 8c). Please note that Figure 7 and Figure 8b have log-scale axes. Figure 6 summarizes the takeaway of performance improvements.




**(a) Google-F1 Workload** **(b) Facebook-TAO Workload** **(c) TPC-C Workload**

**Figure 7: NCC achieves much lower latency under read-dominated workloads with its specialized read-only transaction algorithm, 50% lower latency under write-intensive workload, and at least 80% higher throughput across workloads.**

## 6.3 Latency vs. Throughput Experiments

Figure 7 shows NCC's overall performance is strictly better than the baselines, i.e., higher throughput with the same latency and lower latency with the same throughput.

**Google-F1 and Facebook-TAO.** Figure 7a shows the results under Google-F1. X-axis is the system throughput, and y-axis shows the median read latency in log scale. A horizontal line (O.P.) marks the operating point with reasonably low latency ($< 10$ ms). At the operating point, NCC has a 2–4× higher throughput than dOCC and d2PL. We omit the results for Janus-CC to make the graph clearer as we found that Janus-CC's performance is incomparable (consistently worse) with other baselines, because Janus-CC is designed for highly contended workloads by relying on heavy dependency tracking, which is more costly under low contention.

NCC has better performance because Google-F1 and Facebook-TAO have many naturally consistent transactions due to the prevalence of reads. NCC enables low overhead by leveraging natural consistency. In particular, its read-only transaction protocol executes the dominating reads with the minimum costs (Figure 6). For instance, at the operating point, NCC has about 99% of the transactions that passed their safeguard check and finished in one round trip. 99.1% of the transactions did not delay their responses, i.e., the real-time order dependencies were already satisfied when they arrived. That is, 99% of the transactions were finished by NCC within a single RTT without any delays. For the 1% of the transactions that did not pass the safeguard check initially, 70% of them passed the smart retry. Only 0.2% of the transactions were aborted and retried from scratch. All of them were committed eventually.

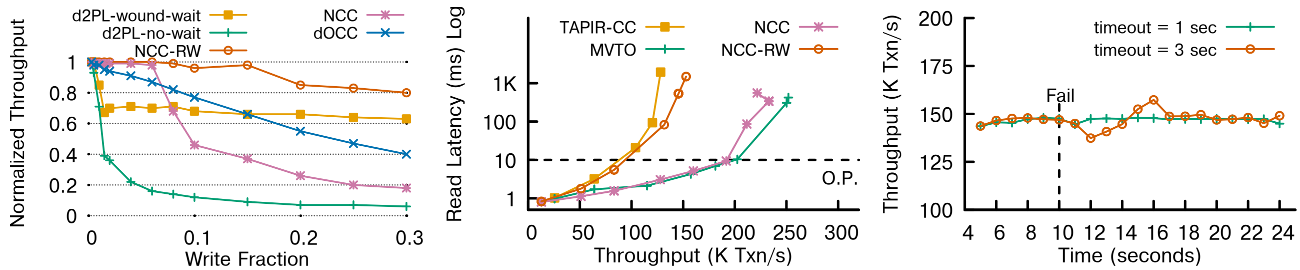
As a result, NCC can finish most transactions with one round of messages (for the read-only ones) and a latency of one RTT (for both read-only and read-write) while dOCC and d2PL-wound-wait require three rounds of messages and a latency of two RTTs (asynchronous commitment saves one RTT). NCC has much higher throughput than d2PL-no-wait due to its novel read-only protocol which requires one round of messages, while d2PL-no-wait requires two. The fewer messages of NCC translate to lower latency under medium and high load due to lower queuing delay. d2PL-no-wait performs similar to NCC-RW because NCC-RW executes read-only transactions by following its read-write protocol. However, NCC-RW outperforms d2PL-no-wait under higher load because conflicts cause d2PL-no-wait to abort more frequently, while NCC-RW has fewer false aborts by leveraging the natural arrival order. This is more obvious in the Facebook-TAO results shown in Figure 7b, because Facebook-TAO has larger read transactions that are more likely to conflict with writes. The results of Facebook-TAO show similar takeaways.

**TPC-C.** Each experiment ran all five types of TPC-C transactions, and Figure 7c shows the latency and throughput (both in log scale) of New-Order while the throughput of the other four types is proportional. NCC and NCC-RW have ∼20× higher peak throughput with ∼10× lower latency compared to dOCC. dOCC and d2PL-no-wait have many false aborts when load increases due to conflicting writes. NCC and NCC-RW can execute most naturally consistent transactions with low costs, even if they conflict. For instance, NCC-RW has more than 80% of the transactions passing the safeguard check and fewer than 10% of the transactions being aborted and retried from scratch. NCC-RW has a 50% higher peak throughput than d2PL-wound-wait because NCC-RW requires only two rounds of messages, while d2PL-wound-wait requires three. NCC-RW has higher peak throughput than NCC because TPC-C has very few read-only transactions, which are also more likely to abort in NCC due to conflicting writes. Janus-CC's performance benefits mostly come from unifying the transaction and replication layers and are less significant in a single-datacenter setting, especially after we made some TPC-C transactions multi-shot.

## 6.4 Additional Experiments

We show more experiments with Google-F1. We chose Google-F1 because it has both read-write and read-only transactions, while Facebook-TAO only has read-only transactions and non-transactional writes.

**Varying write fractions.** Figure 8a shows the throughput while increasing the write fraction. Each system is run at ∼75% load according to Figure 7a. The y-axis is the through-




**(a) Varying Write Fractions** **(b) Weaker Serializability** **(c) Failure Recovery**

**Figure 8: NCC's performance with different write fractions (Google-WF), compared to serializable protocols (TAPIR-CC and MVTO), and under failures for the Google-F1 workload.**

put normalized to the maximum throughput of each system during the experiment. The higher the write fraction, the more conflicts in the system. The results show that NCC-RW is most resilient to conflicts because NCC-RW can exploit more concurrency in those conflicting but naturally consistent transactions, i.e., NCC has fewer aborts. In contrast, other protocols may falsely abort transactions due to failed validation (dOCC) or lock unavailability (d2PL variants). NCC's read-only transactions are more likely to abort when writes increase because frequent writes cause the client to have stale knowledge of the most recently executed writes on each server; as a result, NCC must abort the reads to avoid timestamp inversion.

**Comparing with serializable systems.** Figure 8b compares NCC with MVTO and TAPIR-CC, which provide serializability, under Google-F1. NCC outperforms TAPIR-CC because NCC has fewer messages with its read-only transaction protocol. MVTO and NCC have similar performance under low and medium load because they have the same number of messages and RTTs. Under high load, MVTO outperforms NCC when many read-only transactions in NCC are aborted: MVTO never aborts reads because it is allowed to read stale versions, whereas NCC must read the most recent version and handle timestamp inversion. In this sense, MVTO presents a performance upper bound for strictly serializable systems, and NCC closely matches the upper bound.

**Failure recovery.** Figure 8c shows how well NCC-RW handles client failures under Google-F1. We inject failures 10 seconds into the experiment by forcing *all* clients to stop sending the commit messages of ongoing transactions while they continue issuing new transactions. Undelivered commit messages cause servers to delay the responses of later transactions due to response timing control, until the recovery mechanism is triggered after a timeout. We show two timeout values, 1 and 3 seconds. NCC-RW recovers quickly after failures are detected, thus client failures have a limited impact on throughput. In realistic settings, failures on one or a few clients would have a negligible impact because uncommitted reads do not block other reads. Similarly, NCC is minimally impacted by client failures because its read-only transactions do not send commit messages and thus never delay later writes.

## 7 Related Work

NCC proposes a new strictly serializable distributed protocol. This section places it in the context of existing strictly serializable techniques, single-machine concurrency control, and techniques that provide weaker consistency. At a high-level, NCC provides better performance, addresses a different problem setting, and provides stronger guarantees, compared to these categories of work, respectively.

**General strictly serializable protocols.** As discussed in Section 2.3, existing general strictly serializable protocols are d2PL, dOCC, TR, or their variants, suffering extra costs when transactions are naturally consistent. For instance, Spanner's read-write transactions [12], Sinfonia [4], and Carousel [68] are variants of d2PL that must acquire locks. FaRM [15], FaRMv2 [58], RIFL [31] are variants of dOCC that suffer extra validation costs, even if they use timestamp-based techniques to reduce validation aborts. AOCC [2] is a variant of dOCC and operates in a data-shipping environment, e.g., data can move from servers to client caches, which is different from NCC which works in a function-shipping environment, i.e., data resides only on servers. Rococo [51] and its descendant Janus [52] reorder transactions to minimize aborts. Granola [13] requires an all-to-all exchange of timestamps between servers, incurring extra messages and RTTs. Our evaluation shows that NCC outperforms these techniques for real-world workloads where natural consistency is prevalent. When transactions are not naturally consistent, however, these techniques could outperform NCC. Figure 9 summarizes performance and consistency properties of NCC and some representative distributed systems.

**Special strictly serializable techniques.** In addition to the general techniques discussed above, there are several interesting research directions that use specialized techniques to provide strict serializability. Some work utilizes a centralized sequencer to enforce strict serializability [6, 19, 33, 36, 45, 56, 62, 73]. Because all transactions must contact the sequencer before execution (e.g., Eris [33]), in addition to the extra latency, the sequencer can be a single point of failure and scalability bottleneck. Scaling out sequencers incurs extra costs, e.g., Calvin [62] requires all-to-all messages among

| System | Consistency | Technique | Latency (RTT) | Lock-free | Non-blocking | False aborts |
|---|---|---|---|---|---|---|
| NCC | Strict Ser. | NC+TS | 1 | Yes | Yes | Low |
| Spanner [12] | Strict Ser. | d2PL+TrueTime | RO: 1, RW: 2 | RO: Yes, RW: No | No | RO: None, RW: Med |
| d2PL-NoWait | Strict Ser. | d2PL | 1 | No | No | High |
| AOCC [2] | Strict Ser. | dOCC | 2 | Yes | No | High |
| Janus [52] | Strict Ser. | TR | 2 | Yes | No | None |
| dOCC | Strict Ser. | dOCC | 2 | No | No | High |
| d2PL-WoundWait | Strict Ser. | d2PL | 2 | No | No | Med |
| FaRMv2 [58] | Strict Ser. | dOCC | 2 | No | No | Med |
| TAPIR [72] | Ser. | dOCC+TS | 1 | Yes | No | Med |
| DrTM [66] | Ser. | RO: TS, RW: d2PL | RO: 2, RW: 3 | RO: Yes, RW: No | No | Med |
| TO [7] | Ser. | TS | 1 | Yes | No | Med |
| MVTO [55] | Ser. | TS | 1 | Yes | No | Low |

**Figure 9: The consistency and best-case performance of representative distributed protocols for naturally consistent workloads, processing one-shot transactions with possible optimizations considered. NC means natural consistency, and TS means timestamp-based technique. NCC has the lowest performance costs while providing strict serializability.**

sequencers for each transaction (epoch). Some ensure strict serializability by moving all data a transaction accesses to the same machine, e.g., LEAP [35]. Some rely on program analysis and are application-dependent, e.g., the homeostasis protocol [57]. Some rely on extensive gossip messages for liveness, which lower throughput and increase latency, e.g., Ocean Vista [18] whose latency of a transaction cannot be lower than the gossiping delay of the slowest server even if this server is not accessed by the transaction. General techniques such as NCC do not have the above limitations.

**Strictly serializable read-only transaction protocols.** To the best of our knowledge, the only existing strictly serializable read-only transaction protocol that has optimal *best-case* performance is Spanner [12]. Spanner ensures strict serializability by using d2PL for read-write transactions and by using synchronized clocks (TrueTime) for read-only transactions. TrueTime must be accurately bounded for correctness and those bounds need to be small to achieve good performance, which are achieved by Google's infrastructure using special hardware, e.g., GPS and atomic clocks [9] that are not generally available. For instance, CockroachDB [61], which began as an external Spanner clone, chose not to support strict serializability because it does not have access to such infrastructure [25]. In contrast, NCC's read-only transactions achieve optimal best-case performance and provide strict serializability, without requiring synchronized clocks.

**Single-machine concurrency control.** Concurrency control for single-machine databases is different from the distributed setting on which this paper focuses. First, some techniques are not feasible in a distributed setting. For instance, Silo [64] relies on atomic instructions, and MVTL [3] relies on shared lock state, which are challenging across machines. Second, most techniques, e.g., Silo [64] and TicToc [69], follow a multi-phase design and would be expensive if made distributed, e.g., they need distributed lock management and one round of inter-machine messages for each phase, which would be unnecessary costs for naturally consistent transactions. Their designs, however, are feasible and highly performant for the single-machine setting they target.

**Protocols for weaker consistency.** Many systems trade strong consistency for better performance. For instance, some settle for restricted transaction APIs, e.g., read-only and/or write-only transactions [16, 37, 38]. Some choose to support weaker consistency models, e.g., causal consistency and serializability [17, 32, 38, 39, 46, 61, 65, 70]. In contrast, NCC provides stronger consistency and supports general transactions, greatly simplifying application development.

## 8 Conclusion

Strictly serializable datastores are advocated by recent work because they greatly simplify application development. This paper presents NCC, a new design that provides strict serializability with minimal overhead by leveraging natural consistency in datacenter workloads. NCC identifies and overcomes timestamp inversion, a fundamental correctness pitfall in timestamp-based concurrency control techniques. NCC significantly outperforms existing strictly serializable techniques and closely matches the performance of serializable systems.

**Acknowledgments.** We gratefully thank our shepherd, Atul Adya, for his invaluable feedback that significantly improved this work. We thank the anonymous reviewers for their careful reading of our paper and their many insightful comments and suggestions. This work was supported by the National Science Foundation under grant CNS 1824130, 2130590, 2238768 as well as a gift from Microsoft Research. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

**Availability.** Code and experimental scripts are available at https://github.com/nyu-news/janus/tree/ncc. More details on NCC are in the technical report [41].

## References


[1] Atul Adya. *Weak consistency: a generalized theory and optimistic implementations for distributed transactions*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 1999.

[2] Atul Adya, Robert Gruber, Barbara Liskov, and Umesh Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. *ACM SIGMOD Record*, 24(2):23–34, 1995.

[3] Marcos K Aguilera, Tudor David, Rachid Guerraoui, and Junxiong Wang. Locking timestamps versus locking objects. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 2018.

[4] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. In *ACM Symposium on Operating System Principles (SOSP)*, 2007.

[5] InfiniBand Trade Association. Infiniband architecture specification, release 1.0, october 2000. `http://www.infinibandta.org/`, 2000.

[6] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobbler, Michael Wei, and John D Davis. CORFU: A shared log design for flash clusters. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[7] Philip A Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR)*, 13(2):185–221, 1981.

[8] Google Cloud Blog. Why you should pick strong consistency, whenever possible. `https://cloud.google.com/blog/products/databases/why-you-should-pick-strong-consistency-whenever-possible`, 2018.

[9] Eric Brewer. Spanner, TrueTime and the CAP theorem. Technical report, Google Research, 2017.

[10] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. TAO: Facebook's distributed data store for the social graph. In *USENIX Annual Technical Conference (ATC)*, Jun 2013.

[11] Haibo Chen, Rong Chen, Xingda Wei, Jiaxin Shi, Yanzhe Chen, Zhaoguo Wang, Binyu Zang, and Haibing Guan. Fast in-memory transaction processing using RDMA and HTM. *ACM Transactions on Computer Systems (TOCS)*, 35(1):1–37, 2017.

[12] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman andSanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[13] James Cowling and Barbara Liskov. Granola: Low-overhead distributed transaction coordination. In *USENIX Annual Technical Conference (ATC)*, Jun 2012.

[14] DPDK. DPDK. `http://dpdk.org/`, 2020.

[15] Aleksandar Dragojevic, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: distributed transactions with consistency, availability, and performance. In *ACM Symposium on Operating System Principles (SOSP)*, Oct 2015.

[16] Jiaqing Du, Sameh Elnikety, Amitabha Roy, and Willy Zwaenepoel. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *ACM Symposium on Cloud Computing (SoCC)*, 2013.

[17] Amazon DynamoDB. Amazon DynamoDB :: Fast and flexible NoSQL database service for any scale. `http://aws.amazon.com/dynamodb/`, 2021.

[18] Hua Fan and Wojciech Golab. Ocean Vista: Gossip-based visibility control for speedy geo-distributed transactions. *Proceedings of the VLDB Endowment (PVLDB)*, 12(11):1471–1484, 2019.

[19] FaunaDB. FaunaDB :: The data API for your client-serverless applications. `https://fauna.com/`, 2021.

[20] Hector Garcia-Molina and Kenneth Salem. Main memory database systems: An overview. *IEEE Transactions on knowledge and data engineering*, 4(6):509–516, 1992.

[21] David K. Gifford. *Information storage in a decentralized computer system*. PhD thesis, Stanford University, Department of Electrical Engineering, 1981.

[22] Matthew P Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert NM Watson, Andrew W Moore, Steven Hand, and Jon Crowcroft. Queues don't matter when you can

JUMP them! In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[23] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[24] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan PC Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-store: A high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment (PVLDB)*, 1(2):1496–1499, 2008.

[25] Spencer Kimball and Irfan Sharif. Living without atomic clocks. https://www.cockroachlabs.com/blog/living-without-atomic-clocks/, 2021.

[26] Kishori M Konwar, Wyatt Lloyd, Haonan Lu, and Nancy Lynch. SNOW revisited: Understanding when ideal read transactions are possible. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2021.

[27] Tim Kraska, Gene Pang, Michael J Franklin, Samuel Madden, and Alan Fekete. MDCC: Multi-data center consistency. In *ACM SIGOPS European Conference on Computer Systems (EuroSys)*, 2013.

[28] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM (CACM)*, 21(7), 1978.

[29] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[30] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, 2001.

[31] Collin Lee, Seo Jin Park, Ankita Kejriwal, Satoshi Matsushitay, and John Ousterhout. Implementing linearizability at large scale and low latency. In *ACM Symposium on Operating System Principles (SOSP)*, 2015.

[32] Justin Levandoski, David Lomet, Sudipta Sengupta, Ryan Stutsman, and Rui Wang. High performance transactions in deuteronomy. In *Conference on Innovative Data Systems Research (CIDR)*, 2015.

[33] Jialin Li, Ellis Michael, and Dan RK Ports. Eris: Coordination-free consistent transactions using in-network concurrency control. In *ACM Symposium on Operating System Principles (SOSP)*, 2017.

[34] Kai Li and Jeffrey F Naughton. Multiprocessor main memory transaction processing. In *Proceedings International Symposium on Databases in Parallel and Distributed Systems*, pages 177–178. IEEE Computer Society, 1988.

[35] Qian Lin, Pengfei Chang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Zhengkui Wang. Towards a non-2PC transaction management in distributed database systems. In *ACM International Conference on Management of Data (SIGMOD)*, 2016.

[36] Yu-Shan Lin, Shao-Kan Pi, Meng-Kai Liao, Ching Tsai, Aaron Elmore, and Shan-Hung Wu. MgCrab: Transaction crabbing for live migration in deterministic database systems. *Proceedings of the VLDB Endowment (PVLDB)*, 12(5):597–610, 2019.

[37] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *ACM Symposium on Operating System Principles (SOSP)*, 2011.

[38] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[39] David Lomet, Alan Fekete, Rui Wang, and Peter Ward. Multi-version concurrency via timestamp range conflict management. In *IEEE International Conference on Data Engineering (ICDE)*, 2012.

[40] Haonan Lu, Christopher Hodsdon, Khiem Ngo, Shuai Mu, and Wyatt Lloyd. The SNOW theorem and latency-optimal read-only transactions. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[41] Haonan Lu, Shuai Mu, Siddhartha Sen, and Wyatt Lloyd. NCC: Natural concurrency control for strictly serializable datastores by avoiding the timestamp-inversion pitfall (extended version). https://arxiv.org/abs/2305.14270, 2023.

[42] Haonan Lu, Siddhartha Sen, and Wyatt Lloyd. Performance-optimal read-only transactions. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[43] Haonan Lu, Siddhartha Sen, and Wyatt Lloyd. Performance-optimal read-only transactions (extended version). Technical report, TR-005-20 v1, Princeton University, Department of Computer Science, 2020.

[44] Haonan Lu, Kaushik Veeraraghavan, Philippe Ajoux, Jim Hunt, Yee Jiun Song, Wendy Tobagus, Sanjeev Kumar, and Wyatt Lloyd. Existential consistency: Measuring and understanding consistency at Facebook. In *ACM Symposium on Operating System Principles (SOSP)*, Oct 2015.

[45] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. Aria: A fast and practical deterministic OLTP database. *Proceedings of the VLDB Endowment (PVLDB)*, 13(12):2047–2060, 2020.

[46] Hatem A Mahmoud, Vaibhav Arora, Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. MaaT: Effective and scalable coordination of distributed transactions in the cloud. *Proceedings of the VLDB Endowment (PVLDB)*, 7(5):329–340, 2014.

[47] Microsoft. Microsoft Azure :: New challenges need agile solutions. Invent with purpose. https://azure.microsoft.com/en-us/, 2020.

[48] David Mills. *RFC1305: Network Time Protocol (Version 3) Specification, Implementation*. RFC Editor, 1992.

[49] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. Timely: RTT-based congestion control for the datacenter. *ACM SIGCOMM Computer Communication Review*, 45(4):537–550, 2015.

[50] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *ACM Special Interest Group on Data Communication (SIGCOMM)*, 2018.

[51] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. Extracting more concurrency from distributed transactions. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[52] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. Consolidating concurrency control and consensus for commits under conflicts. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[53] Christos H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4), 1979.

[54] Costin Raiciu and Gianni Antichi. NDP: Rethinking datacenter networks and stacks two years after. *ACM SIGCOMM Computer Communication Review*, 49(5):112–114, 2019.

[55] David P Reed. Implementing atomic actions on decentralized data. *ACM Transactions on Computer Systems (TOCS)*, 1(1):3–23, 1983.

[56] Kun Ren, Dennis Li, and Daniel J Abadi. SLOG: Serializable, low-latency, geo-replicated transactions. *Proceedings of the VLDB Endowment (PVLDB)*, 12(11):1747–1761, 2019.

[57] Sudip Roy, Lucja Kot, Gabriel Bender, Bailu Ding, Hossein Hojjat, Christoph Koch, Nate Foster, and Johannes Gehrke. The homeostasis protocol: Avoiding transaction coordination through program analysis. In *ACM International Conference on Management of Data (SIGMOD)*, 2015.

[58] Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojević, Dushyanth Narayanan, and Miguel Castro. Fast general distributed transactions with opacity. In *ACM International Conference on Management of Data (SIGMOD)*, 2019.

[59] 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. F1: A distributed SQL database that scales. *Proceedings of the VLDB Endowment (PVLDB)*, 2013.

[60] Michael Stonebraker, Samuel Madden, Daniel J Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: It's time for a complete rewrite. In *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*, pages 463–489. Association for Computing Machinery and Morgan & Claypool, 2018.

[61] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. CockroachDB: The resilient geo-distributed SQL database. In *ACM International Conference on Management of Data (SIGMOD)*, 2020.

[62] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *ACM International Conference on Management of Data (SIGMOD)*, 2012.

[63] TPC. TPC-C: An on-line transaction processing benchmark. http://www.tpc.org/tpcc/, 2020.

[64] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *ACM Symposium on Operating System Principles (SOSP)*, 2013.

[65] Xingda Wei, Rong Chen, Haibo Chen, Zhaoguo Wang, Zhenhan Gong, and Binyu Zang. Unifying timestamp with transaction ordering for MVCC with decentralized scalar timestamp. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2021.

[66] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using RDMA and HTM. In *ACM Symposium on Operating System Principles (SOSP)*, 2015.

[67] Arthur Whitney, Dennis Shasha, and Stevan Apter. High volume transaction processing without concurrency control, two phase commit, SQL or C++, 1997.

[68] Xinan Yan, Linguan Yang, Hongbo Zhang, Xiayue Charles Lin, Bernard Wong, Kenneth Salem, and Tim Brecht. Carousel: Low-latency transaction processing for globally-distributed data. In *ACM International Conference on Management of Data (SIGMOD)*, 2018.

[69] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. TicToc: Time traveling optimistic concurrency control. In *ACM International Conference on Management of Data (SIGMOD)*, 2016.

[70] Xiangyao Yu, Yu Xia, Andrew Pavlo, Daniel Sanchez, Larry Rudolph, and Srinivas Devadas. Sundial: Harmonizing concurrency control and caching in a distributed OLTP database management system. *Proceedings of the VLDB Endowment (PVLDB)*, 11(10):1289–1302, 2018.

[71] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Building consistent transactions with inconsistent replication. In *ACM Symposium on Operating System Principles (SOSP)*, 2015.

[72] Irene Zhang, Naveen Kr Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan RK Ports. Building consistent transactions with inconsistent replication. *ACM Transactions on Computer Systems (TOCS)*, 35(4):1–37, 2018.

[73] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J Beamon, Rusty Sears, John Leach, Dave Rosenthal, Xin Dong, Will Wilson, Ben Collins, David Scherer, Alec Grieser, Young Liu, Alvin Moore, Bhaskar Muppana, Xiaoge Su, and Vishesh Yadav. FoundationDB: A distributed unbundled transactional key value store. In *ACM International Conference on Management of Data (SIGMOD)*, 2021.