Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.Doi Number

Strict Serializable Multidatabase Certification with Out-of-Order Updates

Emil Koutanov

Distributed Systems Research Division, Obsidian Dynamics, NSW 2204 Australia

Corresponding author: Emil Koutanov (ek@obsidiandynamics.com)

ABSTRACT Multi-phase atomic commitment protocols require long-lived resource locks on the participants and introduce blocking behaviour at the coordinator. They are also pessimistic in nature, preventing reads from executing concurrently with writes. Despite their known shortfalls, multi-phase protocols are the mainstay of transactional integration between autonomous, federated systems. This paper presents a novel atomic commitment protocol, STRIDE (Speculative Transactions in Decentralised Environments), that offers strict serializable certification of distributed transactions across autonomous, replicated sites. The protocol follows the principles of optimistic concurrency control, operating on the premise that conflicting transactions are infrequent. When they do occur, conflicting transactions are identified through antidependency testing on the certifier, which may be replicated for performance and availability. The majority of transactions can be certified entirely in memory. Unlike its multi-phase counterparts, STRIDE is nonblocking, decentralised and does not mandate the use of long-lived resource locks on the participants. It also offers a flexible isolation model for read-only transactions, which can be served directly from the participant sites without undergoing certification. Also, update transactions are Φ -serializable, making the certifier immune to the recently disclosed logical timestamp skew anomaly.

INDEX TERMS Atomic Commitment, Concurrency control, Consistency, Databases, Distributed systems, Replication, Transaction processing

I. INTRODUCTION

Many complex systems comprise multiple autonomous applications that “own” some subset of the overall system state and encapsulate parts of the business logic. Often, these systems are labelled as microservices [31] and service-oriented; occasionally, they might also be event-driven. Their contemporary classification notwithstanding, it may be more instructive to think of them as a set of federated subsystems that balance between the ambition for autonomy and the overwhelming need to partake in the greater whole. Ideally, these applications adhere to the single responsibility principle and are fairly elemental on their own; their composition, however, can readily grow into a complex web of process communication that presents many problems to system designers. One such problem is maintaining the notion of global consistency without sacrificing autonomy, which is the main focus of this paper.

We elaborate on the problem before delving further. Consider a simple decentralised credit union, comprising multiple branches, each maintaining a subset of the member

accounts. Branches may run different versions of the banking software. Funds can be transferred between members internally within the bank’s secure network. A transfer must ensure that there are sufficient funds in the outgoing account and that the incoming account can receive those funds.

A naive way of approaching the problem is to engage in a series of discrete (local) transactions with each partaking branch, applying some sort of compensating action in the event of a processing error or a breach of some invariant. This is commonly referred to as the **saga** pattern [23]. While acceptable in some systems, sagas present a consistency nightmare for system designers. At any point during the orchestration of a saga, the system may appear to be inconsistent, i.e., violate some invariant. Even if the situation is remedied eventually, intermediate inconsistent states may lead to other undesirable states that cannot be easily detected and remedied by the initiating saga. For example, a report that is run during the saga’s execution will observe an inconsistency between some pairs of accounts; this could subsequently be misreported as a financial anomaly. Another

problem is failures: processes and network links may be unresponsive and may crash altogether. Especially problematic is the failure of the coordinator, which can leave sagas in a partially completed state.

Next, we turn to **choreography**—a distant cousin of the saga pattern [30]. In a choreographed system, participants closely observe each other's actions and initiate actions of their own, according to some scripted scheme. This pattern is often employed in event-driven systems. In our example, the branch housing the outgoing account might publish an event on a well-known message topic, signifying the start of the transfer. Every branch subscribes to the topic, and upon identifying a transaction that involves one of its accounts, enters the reciprocal transaction in its own ledger, completing the transfer. The main advantage of choreography is that there is no centralised coordinator; provided that the message broker is available, each participant will eventually complete its side of the transaction. There is a problem, however, if one of the participants is unable to complete the transaction; for example, if the incoming account is suspended due to a pending investigation. The affected branch might publish a failure message, which could be acted upon by the initiator, rolling back the transfer on the outgoing account. Even in such a simple example, choreography quickly becomes unwieldy: we effectively find ourselves spreading fragments of the overarching business logic across autonomous participants, making it difficult to reason about as a whole. The more impending problem, however, is that choreography, like sagas, is eventually consistent. Generally speaking, if some transaction observes and later comes to depend on an intermediate state resulting from another in-flight transaction (executed within a saga or by choreography), the former may complete despite the latter aborting; hence, the system will fail to converge on a consistent state. In concurrency control theory, this phenomenon is referred to as “dirty reads” and stems from the lack of *recoverability* [17]. One might say that an eventually consistent system is, in fact, *eventually inconsistent*.

The issue of transactions, atomicity and isolation has been studied for many decades in the context of databases and distributed systems, and the relevant theories, for example, *concurrency control*, are well understood; they have been heavily scrutinised in both the academic circles and in the broader industry and have stood the test of time. ACID works, to put it plainly. It may hence be prudent to consider these theories in the construction of distributed systems.

In yet another solution, one might partially dispense with the notion of autonomy in favour of a shared state, using ACID to govern transactions across branches. Depending on the expected volume of transactions, one may consider a distributed database that conforms to the necessary serializability guarantees; for example, Fauna [33], [39] and FoundationDB [32]. This addresses the problem from a functional standpoint, and a distributed database will ostensibly provide sufficient performance; however, the sites

are now strongly coupled by the sharing of their database schema definition and the underlying entity states.

We remark on a minor variation of the above: the use of replicated databases with reduced isolation guarantees to achieve global strict serializability, as demonstrated by Bornea *et al.* [11] in their work on **serializable generalized snapshot isolation** (SGSI). This is an extension of the distributed database model, wherein decentralized sites can perform serializable reads from their local replica databases and are required to send their update transactions to an external certifier for final verification and commitment. The replicas and the certifier share a schema definition; however, each replica may have a different version of the entities installed at any given time.

ACID has served as a largely indispensable tool for solving this very problem in monolithic systems; its projection to federated systems has been less successful, however. Using ACID at the database level invariably leads to a significant reduction of autonomy, as participants are bound by a schema contract.

An alternative approach is to group entity data on the basis of known transactions that intersect those entities. For example, transfers span account ledgers and possibly some status flags, therefore, those data elements should be grouped in a single logical database. The same transactions do not modify member details and some sundry account attributes: the latter could reside in a different logical database. In this manner, the system's overall data set is partitioned not by entity aggregates, as it is commonly done in microservices and service-oriented architectures, but by the nature of business processes that atomically read and update those entities. When any branch wishes to initiate a transfer, it forwards the transaction request to the authoritative system responsible for transfers (e.g., a transfer coordinator), which accepts or rejects transfers using ACID under the hood. Once the transfer completes, the initiator is notified of the outcome. Furthermore, each branch can maintain a local read-only projection of the global state that is asynchronously populated by learning the outcomes of all transfers in the system. This bears some similarities to the SGSI approach, in that the transfer coordinator acts as the certifier and the bank branches are de facto replicas; however, the replicas do not share a schema. Nonetheless, there are several notable problems with this approach. First, there is an *a priori* stability of transaction scopes—a premise that may not persist as the system evolves. If the transaction scope changes or a new transaction is introduced that intersects with entities coordinated by different systems, a costly re-engineering effort may be incurred. Furthermore, entities will invariably converge onto a single coordinator over time as more transactions are introduced. (The number of disjoint entity sets decreases over time, as more groupings are added.) Second, the authoritative systems represent a point of failure and may bottleneck transactions; if the transfer coordinator is faulty or backlogged, key aspects of the system will become inoperable.

This leaves us with **atomic commitment protocols**. The role of atomic commitment is to coordinate distributed transactions between multiple autonomous sites, such that a transaction either commits at all sites simultaneously or on none. Once the commitment decision is reached, it is durable, and no party may observe a state containing some effects of the transaction but not others. *Two-Phase Commit* (2PC) [18], [19] is the mainstay of atomic commitment protocols that has received much attention and numerous optimisations [20], [21], [22]. The protocol comprises separate voting and decision phases. A single encumbered *coordinator* is permitted to execute the protocol over a set of *resource managers*. 2PC has no mechanism for replacing coordinators, leading to the criticism of it blocking indefinitely if the coordinator is unavailable. *Three-Phase Commit* (3PC) [24] attempted to address 2PC's main deficiency by adding a coordinator election step. Keidar and Dolev have shown 3PC to block after carefully chosen network partition and merge steps [25] and devised E3PC, which uses view-based exclusive coordinator election.

Atomic commitment addresses the key issues of database-centric models; namely, site autonomy. On the flip side, the commitment protocols described above present problems of their own. First, resource locking on the participants is a pessimistic concurrency control measure that negatively impacts throughput [17], preventing reads from executing concurrently with writes. Second, it requires centralised coordination, creating a point of failure on one hand, and forcing business logic to be collocated with the coordinator on the other. Third, participants in a multi-phase commitment protocol must support long-lived resource locks that span multiple phases. Finally, the locking schedule of these protocols may lead to distributed deadlocks, which are more difficult to identify than the local variety. Nonetheless, for lack of viable alternatives, multi-phase atomic commitment is a canonical model in heterogeneous multidatabase systems and federated architectures where site autonomy is imperative [22].

A. CONTRIBUTIONS OF THIS PAPER

This paper presents a novel atomic commitment protocol named **STRIDE** (Speculative Transactions in Decentralised Environments). STRIDE combines foundational concepts from different disciplines: *strict serializable certification* from concurrency control theory, *heterogeneous autonomous sites* from atomic commitment and *atomic broadcast* from distributed consensus. The resulting protocol coordinates distributed transactions across autonomous sites in an optimistic fashion, on the presumption that most transactions do not conflict. Conflicting transactions are identified through antidependency testing on the certifier, which may be replicated for performance and availability. STRIDE is

nonblocking, decentralised, deterministic [16], schema-agnostic and does not mandate the use of long-lived resource locks on the participants. It also offers a flexible isolation model for read-only transactions, which can be served directly from the participant sites. Also, update transactions are Φ -serializable, making them immune to the recently disclosed *Logical Timestamp Skew* (LTS) anomaly [16].

The rest of this paper is structured as follows. Section II presents the necessary theoretical concepts and common definitions that are cited throughout this paper. Section III introduces the system model and outlines the model assumptions. Section IV describes the *Stateless Certification* algorithm that operates over a finite suffix of transactions. Section V offers a refinement of the algorithm that significantly reduces the false abort rate by introducing *Antecedent Set Reification*. Section VI recounts related work in the field of distributed databases. Section VII summarises this paper.

II. DEFINITIONS AND THEORETICAL FOUNDATIONS

This section summarises known concepts and results of concurrency control theory and introduces some useful definitions¹ as a basis for the following sections.

A. TRANSACTIONS AND HISTORIES

A **transaction**, informally, comprises a set of operations over shared resources. A transaction is **atomic**; i.e., it either completes successfully and all of its writes are made **durable**, or its writes are undone. In the former, the transaction is said to be **committed**; in the latter, it is **aborted**.

A **data item** is an elemental, named data resource within the set of shared resources. In practice, a data item may be a tuple in a relational database, identified by its primary key, or a field in a tuple, identified by a combination of the key and column name. Alternatively, it may be a document in a key-value database, identified by a document key, or some attribute of that document. The precise semantics of data access and the granularity of data elements, especially for the purpose of concurrency control, vary between database implementations and we ignore these distinctions in this paper. A transaction may **read** a data item, consuming its state, or **write** to a data item, substituting its state with another. A transaction is **deterministic** if repeat applications over the same set of read inputs produce the same set of writes; i.e., the writeset of a transaction is a deterministic function of its readset. This paper only deals with deterministic transactions.

Like elsewhere in the literature, we use the notation $r_i[x]$ and $w_i[x]$ to denote the execution of a read and write operation, respectively, by a transaction T_i on data item x . This notation can be expanded to $r_i[x, v]$, meaning that T_i reads an item x

¹ Large parts of this section were sourced from our earlier work on the Logical Timestamp Skew anomaly [16].

written by T_j with value v . Similarly, $w_i[x, v]$ means that T_i writes v into x .

The $\rightarrow_S$ precedence operator denotes a happened-before relationship, where $a \rightarrow_S b$ means that a 's occurrence clearly preceded b 's in some schedule S . If S is obvious from the discussion context, the subscript may be omitted. Formally, the precedence operator is a strict partial order binary relation that is irreflexive ($\neg(a \rightarrow a)$), transitive ($a \rightarrow b \wedge b \rightarrow c \Rightarrow a \rightarrow c$) and asymmetric ($a \rightarrow b \Rightarrow \neg(b \rightarrow a)$).

We use c_i and a_i to indicate the commit and abort operations, respectively, of T_i . When the terminal operation exists but is not specified to be one of commit or abort, it is denoted as e_i (ended). Once a transaction has ended, it cannot introduce any further operations. Formally, a transaction T_i has a single end event e_i ; $e_i = c$ if the transaction committed; $e_i = a$ if the transaction aborted; and for every operation $p_i[x] \in \{r_i[x], w_i[x]\}$, $p_i[x] \rightarrow_i e_i$.

A **complete history** H over a set of transactions $\mathcal{T}$ is a partial order over $\mathcal{T}$ with relation $\rightarrow_H$, defined according to the following axioms:

1. **Projection of transaction-local precedence:** If T_i is in $\mathcal{T}$ and $p_i \rightarrow_i q_i$ then $p_i \rightarrow_H q_i$, for all pairs of events p_i, q_i .
2. **Connectivity of conflicting operations:** If T_i and T_j are in $\mathcal{T}$ then for every pair of conflicting operations $p_i[x], q_j[x]$, either $p_i[x] \rightarrow_H q_j[x]$ or $q_j[x] \rightarrow_H p_i[x]$.
3. **Deterministic aborts:** Let T_i, T_j be transactions in $\mathcal{T}$, where $e_i = a$. If $w_i[x] \rightarrow_H r_j[x]$ then either $e_i \rightarrow_H r_j[x]$ or $r_j[x] \rightarrow_H e_i$. I.e., either T_j read the value after T_i aborted or before T_i aborted; otherwise, the value read by T_j cannot be accurately determined.

A prefix of a partial order P over a set S is a partial order P' over a set $S' \subseteq S$ such that:

- If $q \in S'$ and $p \rightarrow_P q$ then $p \in S'$; and
- If $p, q \in S'$ then $p \rightarrow_P q \Leftrightarrow p \rightarrow_{P'} q$.

A **history** is any prefix of a complete history. We consider prefixes to reason about instantaneous state, where only some of the queued transactions have been fully executed, or even partially executed, for instance, in the event of a failure.

A history is represented as a partial order over a set of operations; e.g., the partial order $\{(r_2[x], w_2[z]), (r_2[y], w_2[z]), (w_2[z], c_2)\}$ over the operations $\{r_2[x], r_2[y], c_2\}$. The set of operations may be omitted as it is trivially inferred from the partial order. A partial order is sufficient to produce a directed graph (digraph) like the one in Fig. 1, where operations are vertices and edges denote precedence.

FIGURE 1. A digraph showing a partial order over a set of operations.

Let $S(H)$ be the set of all topological sorts of H . For the graph in Fig. 1, $S(H) = \{(r_2[x], r_2[y], w_2[z], c_2), (r_2[y], r_2[x], w_2[z], c_2)\}$.

If the digraph is of degree one uniformly throughout, then $|S(H)| = 1$ and the history can be written more compactly using a sequence; e.g., $(w_2[x], w_2[y], w_2[z])$.

Returning to the earlier informal description of a transaction, we now present a formal definition. A transaction T_i is a partial order with the strict ordering relation (irreflexive, transitive, asymmetric) $\rightarrow_i$, defined according to the following axioms (assume T_i reads and writes at most once to a data item, otherwise take $r_i[x]$ and $w_i[x]$ to mean the first read and the last write of x):

1. **A transaction may comprise reads, writes, commits and aborts:** $T_i \subseteq \{r_i[x] : x \text{ is a read item}\} \cup \{w_i[x] : x \text{ is a written item}\} \cup \{a_i, c_i\}$.
2. **A transaction must either commit or abort:** $a_i \in T_i \vee c_i \in T_i$.
3. **All non-terminal operations precede a terminal operation:** for a terminal operation $e_i \in \{a_i, c_i\}$, every other operation $p_i \in T_i \setminus \{e_i\}$, $p_i \rightarrow_i e_i$.
4. **Read and write operations do not commute:** for $r_i[x], w_i[x] \in T_i$, $r_i[x] \rightarrow_i w_i[x] \vee w_i[x] \rightarrow_i r_i[x]$.

A **schedule** is a partial order over a set of events. A history is a schedule. This paper uses schedules to refer to partial orders (and by extension, total orders) outside of transactional operations, for instance, to depict sequences of logical timestamps.

Let S and S' be two sets such that $S' \subseteq S$ and let P be a partial order over a set S . A **projection** of P on a set S' is a partial order P' (a subset of P) consisting of all elements in P involving only elements of S' .

For a complete history H (a partial order $\rightarrow_H$ over a set of transactions $\mathcal{T}$), a projection of H on a set of all operations of some transaction $T_i \in \mathcal{T}$, produces a partial order $\rightarrow_i$ over T_i , complete with all operations of T_i and void of all other operations. A (prefix) history, however, may project a partial history for some transaction that does not conform to the transaction axioms above.

The **commit projection** $C(H)$ of a history H is its projection on its set of committed transactions. Commit projections are used to erase the operations of aborted transactions.

B. CONFLICTS AND EQUIVALENCE OF HISTORIES

Two operations are said to **conflict** if they both operate on the same data item and at least one of them is a write; i.e., $r_i[x]$ conflicts with $w_j[x]$, whereas $w_i[x]$ conflicts with both $r_j[x]$ and $w_j[x]$, for all pairs of transactions T_i, T_j .

Informally, two histories are **view equivalent** if they exhibit the same residual effects on item states after the effects of all aborted transactions have been erased. Formally, histories H and H' are view equivalent if:

1. They contain the same operations.

2. For every T_i, T_j such that $a_i, a_j \notin H$ (hence $a_i, a_j \notin H'$) and for every x , if T_i reads x from T_j in H , then T_i reads x from T_j in H' .
3. For every x , if $w_i[x]$ is the final write of x in H then it is also the final write of x in H' . The final write of x is the operation $w_i[x]$ such that $a_i \notin H$ and for all $w_j[x] \in H$ ($i \neq j$) either $w_j[x] \rightarrow w_i[x]$ or $a_j \in H$.

We also assume that, for all x , some initial value of x is written by a notional progenitor transaction T_{init} omitted from H (and hence H'). As per rule 2, if T_j reads the initial value of x (written by T_{init}) in H then it also reads the initial value of x in H' .

A more restrictive definition of equivalence is **conflict equivalence**, which states that two histories H and H' conflict are equivalent if:

1. H and H' are defined over the same set of operations.
2. For their commit projections $C(H)$ and $C(H')$ respectively, the order of conflicting operations is identical in both $C(H)$ and $C(H')$. I.e., for pairs of conflicting operations $p[x], q[x] \in C(H)$, $p[x] \rightarrow_{C(H)} q[x] \Leftrightarrow p[x] \rightarrow_{C(H')} q[x]$.

Alternatively, H and H' are **conflict equivalent** if, for some pair of topological sorts, $G \in S(H)$ and $G' \in S(H')$, it is possible to arrive from G to G' through a series of swaps of pairs of adjacent nonconflicting operations in G .

All pairs of histories that are conflict equivalent are also view equivalent.

C. SERIAL HISTORIES, ISOLATION PROPERTIES AND SCHEDULERS

A **serial history** is one in which, for every transaction, the operations of that transaction are contiguous (not interleaved with the operations from any other transaction). Formally, for a serial history H , for all $T_i, T_j \in H$ ($i \neq j$), either $\forall p \in T_i, q \in T_j : p \rightarrow_H q$ or $\forall p \in T_i, q \in T_j : q \rightarrow_H p$. A serial history may be compactly represented by listing the transactions in their serial order; e.g., (T_1, T_2, T_3) . (Assuming the partial order of those transactions' operations is recorded elsewhere.)

An **isolation property** (or **isolation level**, as the terms may be used interchangeably) is a predicate I applicable to an execution history H , such that $I(H)$ iff (if and only if) the property holds.

As it is common elsewhere in the literature, a property's abbreviation is used as a surrogate for the set of all histories permitted by that property. For example, SR is an abbreviation for serializable and is also a surrogate for the set of all serializable histories. Formally, for an isolation property I , its abbreviation $\mathcal{J}$ is the set comprehension $\{H \in \mathcal{H} : I(H)\}$, where $\mathcal{H}$ is the set of all histories.

As preamble for the subsequent definitions, let $Start(T)$ and $End(T)$ resolve the real-time start and end timestamps, respectively, of a transaction T . $Start(T)$ and $End(T)$ are

injective: $Start(T_i) = Start(T_j) \Rightarrow T_i = T_j$. Also, transactions take some time to complete, hence $Start(T_i) < End(T_i)$. $Start$ and End are monotonic increasing with their arguments: if T_i started before T_j , then $Start(T_i) < Start(T_j)$, similarly if T_i ended before T_j , then $End(T_i) < End(T_j)$. The images of $Start$ and End do not intersect; i.e., there is no pair of transactions T_i and T_j for which $Start(T_i) = End(T_j)$.

$Start$ and End can be used to enforce a real-time constrained precedence partial order $\rightarrow_{RT}$ over the set of transactions, such that $T_i \rightarrow_{RT} T_j \Rightarrow End(T_i) < Start(T_j)$. Note, there is no requirement that $Start$ and End faithfully mirror some reference global clock; only that they correctly impose a temporal order over events. In other words:

- Every timestamp is unique.
- No two timestamps are the same unless they represent the same event.
- A higher-valued timestamp occurs later² than a lower-valued timestamp.

We say that a transaction T_i **precedes** another transaction T_j if T_i ended before T_j started; i.e., $End(T_i) < Start(T_j) \Rightarrow T_i \rightarrow T_j$.

Snapshot isolation (SI) is both an isolation property and a concurrency control mechanism that satisfies this property. Informally, it guarantees that all reads made in a transaction will see the most recent consistent snapshot of the database [6], [25]. Formally, if T_i reads a data item x , then x was written by a committed transaction T_j that precedes T_i , and there is no committed T_k ($j \neq k$) that also precedes T_i that wrote to x after T_j , and T_i either observes T_j 's write to x or its own write to x , whichever is more recent. In other words, writes by concurrent transactions that are active after T_i starts are not visible to T_i [1], [27].

Snapshot isolation prevents the **read skew** anomaly, in which a transaction observes an inconsistent view of the database. For example, suppose T_1 reads x , then T_2 updates both x and y . If now T_1 reads y , it will see an inconsistent state. SI prevents T_1 from observing T_2 's concurrent write to y .

Generalized snapshot isolation (GSI) relaxes the timing constraint [12] of SI. In the generalised variant, a transaction T_i observes a snapshot that is consistent with some prefix of operations involving committed transactions that precede T_i . GSI allows pathological orderings. In the most extreme example, the returned snapshot may be consistent with an empty history; i.e., a view of the database that corresponds to its initial state.

Item cut isolation (I-CI) is an isolation property wherein each transaction reads from a non-changing cut, or snapshot, over the discrete data items [5].

Monotonic atomic view (MAV) is an isolation property wherein once some of the effects of a transaction T_i are observed by another transaction T_j , thereafter, all effects of T_i are observed by T_j [5]. Applying MAV in conjunction with I-CI (i.e., $MAV \cap I-CI$) prevents *read skew* anomalies [5].

² This is the only criterion in the entire formulation that relies on the passage of time. The easiest way to satisfy this is via a shared counter that is atomically incremented at each invocation of $Start$ and End .

Serializability, informally, is an isolation property that states that a concurrent execution of transactions is serializable if its outcome is equivalent to some outcome in which those transactions execute serially. It serves as the foundation for reasoning about the consistency of a system in the presence of concurrent operations. Serializability is often presented as a necessary correctness condition in the design of concurrent systems.

A history is **(conflict) serializable** (SR; is in SR) if its commit projection is conflict equivalent to some serial history. The qualifier ‘conflict’ is typically omitted; serializable histories are assumed to be conflict serializable. A serial history is serializable.

A history is **view serializable** if the commit projection of every prefix of that history is view equivalent to some serial history. Clearly, from the earlier definition of equivalence, a conflict serializable history is view serializable.

A **blind write** occurs when a transaction writes a value without reading it. A serializable history that is not conflict serializable must contain a blind write.

A **scheduler** controls the concurrent execution of transactions [17]. In practice, a scheduler is a program or a set of programs that form a core part of a database system. A scheduler restricts the order in which read, write, commit and abort operations from different transactions are applied, such that the resulting history conforms to some *a priori* isolation property. Formally, $\mathcal{J}$ schedulers are schedulers that generate histories in $\mathcal{J}$, where $\mathcal{J}$ is the set comprehension of an isolation property I . For example, “serializable schedulers generate histories in SR,” or stated otherwise, “serializable schedulers generate serializable histories.”

Serializable histories permit certain pathological orderings of transactions in the equivalent serial histories. A serializable scheduler can reorder all transactions comprising only reads towards the beginning of the history, and all transactions comprising only writes to the end of the history. Thereby, in an extreme (but plausible) scenario, all read transactions return the initial state for some value x and all writes to x are discarded. These, loosely speaking, “optimisations,” are not consciously employed in practice in database systems; however, in some databases (particularly in the distributed variety) these effects may inadvertently occur from time to time.

Consider a pair of transactions T_i and T_j that are processed serially (T_j is submitted only after T_i is decided) with no other interfering transactions. For some data item x written by T_i and subsequently read by T_j , $r_j[x]$ is not guaranteed to observe $w_i[x]$ in serializable histories. I.e., serializability does not offer any real-time, causal or even per-process guarantees.

Strict serializability is an isolation property that includes the serial-emulating property of serializability, in addition to imposing a real-time constraint on nonconcurrent transactions.

A history is **strictly serializable** (S-SR; is in S-SR) if its commit projection is conflict equivalent to a serial history H ,

and for all pairs of transactions T_i, T_j that are both in H , $End(T_i) < Start(T_j) \Rightarrow T_i \rightarrow_H T_j$.

Strict serializable histories do not permit the pathological orderings that serializable histories are prone to. Specifically, if transactions T_i and T_j are processed sequentially, then for some data item x written by T_i and subsequently read by T_j , $w_i[x] \rightarrow r_j[x]$.

D. CONCURRENCY CONTROL METHODS

When a scheduler receives an operation, it has the choice of scheduling it immediately, delaying it (inserting it into a queue to schedule later) or rejecting it (thereby aborting the transaction).

Schedulers can be classified based on their propensity to delay or reject operations. An **aggressive scheduler** tends to schedule operations immediately, but in doing so, it foregoes the opportunity to reorder the operations later on. Conversely, a **conservative scheduler** tends to delay operations, which may lead to unnecessarily postponing operations that could be completed immediately. A serial scheduler is an extreme case of a conservative scheduler, in which the operations of all transactions but one are delayed. A **certifier** is a special class of schedulers that never delays transactions.

Pessimistic concurrency control is a concurrency control method (a set of algorithms) that assumes that conflicts among transactions are frequent and thereby acts conservatively with respect to scheduling. Conversely, **optimistic concurrency control** assumes that conflicts are rare and schedules operations aggressively, looking for indications of conflicts and aborting transactions as appropriate.

Two-phase locking (2PL) is a pessimistic concurrency control method that guarantees serializability. Locks are applied and removed in two phases:

1. **Expanding phase:** locks are acquired, and no locks are released.
2. **Shrinking phase:** locks are released, and no locks are acquired.

2PL can be reduced to a single rule wherein a *lock is never acquired after it has been released*. 2PL is not strict; the basic protocol allows data items to be read from transactions that have not committed. 2PL (and its variants) are deadlock-prone, in that operations may be delayed to a point where none of the queued operations may proceed, pending other queued operations.

Strict two-phase locking (S2PL) is a strengthened variant of 2PL in which write locks held by a transaction are not released until it commits or aborts; thus, data items cannot be read from undecided transactions.

Strong strict two-phase locking (S-S2PL) is a further strengthening of S2PL in which both read and write locks are not released until the transaction terminates. The property gained by S-S2PL is **commitment ordering**, wherein for all conflicting operations $p_i[x], q_j[x]$, for all committed transactions T_i, T_j , $p_i[x] \rightarrow q_j[x] \Rightarrow c_i \rightarrow c_j$ [15].

Multiversion concurrency control (MVCC) is a family of optimistic concurrency control methods that permit several versions of a data item to coexist, rather than substituting data items for their most recent updates. MVCC avoids contention by presenting a snapshot view of the database to concurrent transactions; changes made by one transaction will not be disclosed to others until the former commits, at which point (or earlier) transactions may be tested for conflicts.

An execution of transactions can be represented with a direct serialization graph [3], with transactions as vertices, and edges indicating precedence in the apparent serial history. Three types of dependencies exist:

1. **write-read dependencies:** if T_i writes a version of a data item, and T_j reads that version, then $T_i \rightarrow T_j$. (We say T_j read-depends on T_i .) Also known as a *flow dependency* or a *true dependency*.
2. **write-write dependencies:** if T_i writes a version of a data item, and T_j replaces that version with the next version, then $T_i \rightarrow T_j$. (We say T_j write-depends on T_i .) Also known as an *output dependency*.
3. **read-write antidependencies:** if T_i writes a version of a data item, and T_j reads the previous version, then $T_j \rightarrow T_i$, because T_j apparently missed T_i 's update. (We say T_j antidepends on T_i .)

Snapshot isolation (SI), in addition to being an isolation property, is an MVCC method that allows a transaction T_i to commit if T_i 's writes have not been superseded by another transaction T_j that committed after T_i 's snapshot was taken—the *first-committer-wins* rule. SI is not serializable, being prone to *write skew* anomalies.

Serializable snapshot isolation (SSI) is a strengthened variant of SI that relies on an abridged form of serialization graph testing to identity potentially nonserializable executions [1], [28].

III. SYSTEM MODEL

The system comprises the following elements.

A. TRANSACTION

A transaction T describes a set of operations that must execute atomically across several autonomous cohorts. It comprises the following attributes.

Transaction ID: Immutable. Abbrev. **XID**. A unique identifier, e.g., a UUID, assigned by the cohort initiating the transaction.

Snapshot version: Immutable. The version of the cohort database used to validate the transaction on the cohort. Denoted by $snapshot(T)$.

Readset: Immutable. The set of items read by the transaction. Denoted by $readset(T)$.

Read versions: Immutable. The logical timestamps of transactions that last updated the items in T 's readset. Denoted by $readvers(T)$.

Writeset: Immutable. The set of items written by the transaction. Denoted by $writeset(T)$.

State mapping: Immutable. Describes how the transaction's updates should be installed, given the existing state. Formally, $statemap(T)$ is a function of the form $[x, S_x \mapsto S_x']$; i.e., given the written item x and its current state S_x , produce a new state S_x' .

A state mapping may be **idempotent**, wherein the result of $statemap(T)$ is independent of S_x ; i.e., repeat applications of the state mapping produce the same outcome independent of the initial state. In the absence of other updates for x , the cohort can safely install an idempotent update any number of times for S_x , consistently producing S_x' . $[x, S_x \mapsto 4]$ is an idempotent state mapping. A **nonidempotent** state mapping is variant on S_x and therefore lacks this property; repeat applications of $statemap(T)$ produce different outcomes. $[x, S_x \mapsto S_x + 1]$ is a nonidempotent state mapping.

Version: Once-mutable. Initially $\emptyset$; once set, it cannot be changed. Indicates the logical timestamp of the transaction within the encompassing total order. Logical timestamps are monotonically increasing sequence numbers assigned to transactions by the atomic broadcast primitive. Denoted by $ver(T)$.

Status: Once-mutable. Initially *pending*; terminal states are *committed* and *aborted*. Indicates the certifier's final decision to commit or abort the transaction.

Safepoint: Once-mutable. Unset on the candidate and assigned only when the transaction commits. Indicates the lower bound of the snapshot version to which the transaction's update may be applied. Denoted by $safepoint(T)$.

Note, since STRIDE is based on replication, we *limit our scope to deterministic transactions*. Otherwise, if transactions are nondeterministic, their installation in identical replicas may lead to divergent replica states.

B. GLOBAL DATABASE

A set of all objects under the certifier's purview. This is a purely virtual construct that need not manifest in reality; it is nonetheless useful for reasoning about the system's state. The global database can be materialised by suspending the certifier and applying all atomically broadcast updates from the first committed transaction to the last in their logical timestamp order.

C. COHORT DATABASE

A local view of the global database, possibly containing a subset of its items, and for those items, possibly depicting their earlier versions. A cohort database is attached to a set of cohort processes.

D. COHORT

Reads from and writes to its local cohort database and initiates transactions based on the replicated state of the cohort database. A cohort—

- Reads the state of its local database.

- Stages a transaction for subsequent certification (and submits it if the transaction is valid locally).
- Serially applies updates from atomic broadcast, upgrading the database's version in the process.
- May apply updates to the local database out of order if it is safe to do so.

A cohort process may be replicated for performance and fault tolerance.

E. AGENT

Acts as a stateless proxy between the initiator process and the certifier. The role of the agent is to accept candidate transactions from cohorts, submit them for certification, and respond to the initiator when the outcome of the transaction has been decided. An agent may be replicated for performance and fault tolerance. Each agent has a unique ID in the system. Agents may be colocated with cohorts for convenience and compactness. An agent process may be replicated for performance and fault tolerance.

F. ATOMIC BROADCAST CHANNEL

Abbrev. **abcast**. A persistent, total order over the messages used by the certifier, where a message represents some action recorded against a transaction. Abcast ensures that all participating processes receive the same set of messages in the same order, and messages are not lost.

The contents of abcast are available to processes outside the certifier, e.g., to cohorts and agents. Delivered messages are assigned monotonically increasing sequence numbers—their logical timestamps; the timestamps are uniformly observed by all recipients.

We specify the behaviour of abcast without constraining its implementation. In practice, abcast may be constructed by employing distributed consensus algorithms such as Paxos [34] (and its derivatives) or Spire [35].

G. TRANSACTION DATABASE

Abbrev. **XDB**. Maintains the hard state of every decided transaction, i.e., those transactions for which a commit/abort decision has been established. There is a single logical XDB instance for the entire system, which is attached to a set of certifier processes. Writes to the XDB are durable and atomic at the item level; an item in the XDB being a transaction decision record written by a certifier. There is no requirement for cross-item transactions; hence, the XDB may be sharded for performance.

H. CERTIFIER

Inspects candidate transactions for conflicts, determines an outcome (commit/abort), broadcasts the outcome and saves the transaction state to the XDB. A certifier may be replicated for performance and fault tolerance.

I. CLIENT

An external process that initiates a transaction with a cohort. Its role is not critical to the algorithm's specification; it is included here for completeness.

J. NETWORK AND PROCESS FAILURES

All processes in the system are prone to failures and communicate asynchronously by message passing in a non-Byzantine environment. While processes may crash, they eventually recover (for example, by restarting).

Message communication over abcast satisfies the usual abcast safety and liveness properties; i.e., messages may take arbitrarily long to arrive, but they are not lost, duplicated or delivered out of order, and they are either delivered on all nonfaulty processes or on none. Message communication outside of abcast is subject to losses, duplications and out-of-order delivery; however, messages cannot be undetectably corrupted.

All databases, unless otherwise specified, have access to nonvolatile storage, and persisted data survive the failure of the attached processes.

Fig. 2 illustrates the basic system model. It is later expanded upon in subsequent refinements of the algorithm.

FIGURE 2. Basic system model.

K. GUARANTEES

Certification is subject to several “ground rules” that underpin its safety:

- S1. Decision stability.** Once a transaction has been decided, the decision may not be altered.
- S2. Abort side effect freedom.** A transaction can always be safely aborted, unless it has already been committed. In the trivial case, every transaction is aborted by the certifier.
- S3. Serializability.** A transaction can only be committed if doing so results in a history that is equivalent to some history in which all transactions executed serially. In the trivial case, if all transactions are submitted strictly serially, they are all committed, yielding a serial history.

Certification is also subject to the usual safety guarantees expected of agreement protocols:

- S4. Nontriviality.** If a transaction is committed, then it was submitted by some cohort.
- S5. Agreement.** If a transaction is decided, then all cohorts observe the same decision value for that transaction.

To satisfy liveness, we assume that sufficient processes and links are nonfaulty for some cohort to eventually submit a transaction, for it to eventually be certified, and for the decision to eventually be learned by the initiating cohort.

IV. ALGORITHM 1 – STATELESS CERTIFICATION

We start with a simple algorithm that satisfies the invariants *S1–S5*, is reasonably performant, but may abort more transactions than necessary in certain cases.

A. PRELIMINARIES

Transactions are denoted with the literal T_i , where the subscript is the transaction's logical timestamp, a synonym for its version. I.e., $ver(T_i) \triangleq i$. Transactions are arranged in a total order, captured by abcast. Transactions are described by a *Candidate* message in the abcast sequence, signalling an intent that may eventuate. The certifier's decision is communicated as a *Decision* message over abcast.

Logical timestamps are not gap-free; i.e., there may be a "hole" in the numbering where some abcast sequence numbers are not assigned to transactions. A *Candidate* message may be published in duplicate, in which case the first abcast sequence number associated with a transaction is its logical timestamp; the 2nd and subsequent depictions of that transaction with higher sequence numbers are ignored. (Duplicates are possible owing to failures and retries.) The logical timestamp of the decision message is strictly greater than that of its corresponding candidate message³.

A transaction starts when it reads its local cohort database and completes when it is decided by the certifier. Within that period, the transaction is said to be **active**. The active period is demarcated by two versions: $snapshot(T)$ for the start and $ver(T)$ for the end. I.e., for a transaction T_i , the active period is $[snapshot(T_i), i]$. Also, every transaction has a nonzero duration; i.e., $snapshot(T_i) < i$.

We now present the **conventional definition of concurrency**. If two transactions are **concurrent** then there is a period in which they overlap; i.e., one transaction starts while the other is active; neither starts after the other completes. Formally, if a pair of transactions T_i, T_j , are concurrent, then $snapshot(T_i) < j \wedge snapshot(T_j) < i$; that is, they overlap. Conversely, if $snapshot(T_i) \geq j \vee snapshot(T_j) \geq i$, then the pair are **serial** (or **nonconcurrent**). Note the similarities to the classical definition quoted for snapshot

isolation systems [2], where a snapshot represents a point-in-time cut of the items in the database. STRIDE snapshots are somewhat more amorphous, containing updates from transactions eagerly applied out of (logical timestamp) order. Therefore, we do not use *iff* in the conventional definition, as there is a more elaborate, **extended definition of concurrency** that will be presented later. That is, if two transactions overlap, it does not imply that they are concurrent. Similarly, if transactions are serial, it does not imply that they do not overlap. Avoiding bidirectional implication here ensures that the conventional definition is still correct, albeit not complete. Specifically, some transactions are not classifiable under the conventional definition.

A transaction is initiated when a cohort (possibly acting on its client's behalf) decides that it must update a set of data items atomically while maintaining global strict serializability. A cohort only has its local database state to go on. We define an **update transaction** as one that writes some items (its **writeset**) and optionally read some items (its **readset**). If the declared readset is nonempty, we assume that the transaction relied on the items in the readset, such that those items may be considered a part of the transaction's invariants.

For example, a transaction might read integer items x and y , and write their sum to z , but only if they sum to 10; otherwise, the transaction aborts. Hence, the readset is $\{x, y\}$, the writeset is $\{z\}$, and it is assumed that the transaction is dependent on the values of its readset. I.e., should those values change, its invariants may be breached. In every serial history containing the transaction in the committed state, its inputs sum to 10.

A **read-only** transaction has an empty writeset. Note, either the readset or the writeset of a transaction must be nonempty for it to be considered legal.

The originating cohort of transaction T_i is C_i , which may be used interchangeably to refer to the cohort's local database. (The reference is clear from the context.) Also, C_i has a baseline version, denoted by $ver(C_i)$, being the version of the most recent transaction, T_i , serially installed in C_i , such that all committed transactions prior to T_i have also been installed. I.e., for every cohort C , $(\forall T_i : prefix_installed(C, T_i) \wedge \nexists T_j : j > i \wedge prefix_installed(C, T_j)) \Rightarrow ver(C) = i$, where $prefix_installed(C, T) \triangleq \forall h \in 1..ver(T) : committed(T_h) \Rightarrow installed(C, T_h)$ and $installed(C, T)$ denotes that the state mapping of T was installed on C . We assume that some initial progenitor transaction T_0 is installed prior to all other transactions; thus, $ver(C) \geq 0$.

B. ALGORITHM OVERVIEW

At an outline level, the certification of T_k involves the following steps.

Step 1: Local validation. C_k determines that a candidate transaction is valid locally. I.e., it can be applied to the local database while satisfying all invariants of the transaction.

³ Owing to the decision message being sent over abcast only after the delivery of the corresponding candidate message.

Failing local validation, the cohort may either abort the transaction or retry validation with a more recent set of values.

In the course of validation, the cohort captures the baseline version of its local database, $ver(C_k)$, at a time that is no later than the time of its reads, storing the result in $cpt_snapshot(T_k)$. I.e., $cpt_snapshot(T_k) \triangleq picklower(ver(C_k))$, where $picklower(n) \triangleq \epsilon \ m \in 0..n : TRUE$. (ϵ is Hilbert's epsilon operator.)

It also captures the versions of all items in its readset, storing the results in $cpt_readvers(T_k)$; i.e., $cpt_readvers(T_k) \triangleq \{ver(x) : x \in readset(T_k)\}$, where $ver(x)$ is the logical timestamp of the most recent transaction that wrote to x on the cohort in question. The capture of read versions and the reading of the values are performed in an atomic step.

Ideally, acquiring the readset and the capture of $ver(C_k)$ are also atomic, such that the captured version is consistent with the time of T_k 's reads. The algorithm, however, permits some earlier version of C_k to be captured for a readset containing updates installed after $cpt_snapshot(T_k)$. This is suboptimal in that it may produce unnecessary aborts; however, it does not compromise safety.

Step 2: Candidate submission. The cohort creates a *CertifyRequest*, containing just the immutable parts of the transaction, and invokes the agent. The request contains—

- The transaction's unique XID, assigned by the cohort.
- The snapshot version, $snapshot(T_k) \triangleq \max(cpt_snapshot(T_k), \min(cpt_readvers(T_k)))$ if $cpt_readvers(T_k) \neq \emptyset$ else $cpt_snapshot(T_k)$.
- The readset, $readset(T_k)$, comprising the items read by the transaction.
- The read versions, $readvers(T_k)$, containing the logical timestamps of the most recent transactions that have written to items in $readset(T_k)$, excluding versions that are equal to or lower than the snapshot version. I.e., $readvers(T_k) \triangleq \{v \in cpt_readvers(T_k) : v > snapshot(T_k)\}$.
- The writeset, $writeset(T_k)$, containing the items written by the transaction.
- The state mapping function, $statemap(T_k)$, describing how the transaction should be applied to an existing state to produce the new state, for each item updated by the transaction.

The *CertifyRequest* call is a blocking remote procedure call on the agent, returning when the transaction completes or times out. Steps 1 and 2 are collectively described by Alg. 1a.

Step 3: Candidate ordering. The agent creates a *Candidate* message containing the elements of the *CertifyRequest*, appends its agent ID and abcasts the resulting message. Abcasting a candidate message has the effect of making it persistent and assigning a sequence number to it. (All abcast messages are totally ordered, persistent and uniquely numbered.)

Step 4: Certification. The certifier inspects the candidate transaction, comparing it to some suffix of transactions that precede it in the total order, to identify potential

antidependencies. The certification algorithm is described in Section IV.C and Alg. 1b.

The certifier decides the outcome of the transaction. In either case (commit/abort), the certifier installs the decision state of the transaction into the XDB, honouring a prior decision state if one exists. The certifier then abcasts a *Decision* message, containing the XID, the logical timestamp of the transaction (i.e., its version), the commit/abort outcome and the initiating agent's ID. In the case of a commit, the *Decision* message also includes a safepoint version, whose calculation is described in Alg. 1c. *Decision* messages may safely be abcast in duplicate, as their payloads cannot differ.

Step 5: Certification response. The agent receives the *Decision* message for a transaction ID that has a pending request as of Step 2. It replies to the initiating cohort with a *CertifyResponse*, containing the decision. The agent may time out the request if it fails to receive a corresponding *Decision* within a set period.

Step 6: Completion. The cohort learns the outcome of the transaction. In the event of commitment, it may apply the transaction's updates directly to its local database if it is safe to do so, as indicated by the transaction's safepoint. The criteria for installing the update of T_k at C_k and the installation process are described in Alg. 1d. In the event of an abort, the cohort may retry the transaction with updated values or cease further activity on the transaction.

Background applicator. The cohort process maintains a separate background applicator thread that receives abcast *Decision* messages and installs the state mapping of the committed transactions, either concurrently, in accordance with their safepoint instructions (Alg. 1d), or serially, upgrading $ver(C_k)$ in the process (Alg. 1e).

Fig. 3 outlines the certification process.

FIGURE 3. Certification overview.

C. CERTIFIER PROCESS

The certifier process maintains a suffix $\mathcal{S}$ of candidate messages received from abcast. $\mathcal{S}$ is held in memory and sized such that it encompasses all undecided transactions, as well as some decided ones. The size of the suffix is tuneable; typically, it is in the order of several seconds' worth of transactions at peak throughput. The certification algorithm is formally presented in Alg. 1b.

Let the earliest transaction in the suffix be T_i . For some candidate transaction $T_k \in \mathcal{S}$, $k \geq i$:

- R1.** Commit write-only transactions. I.e., if $readset(T_k) = \emptyset$, then commit T_k .
- R2.** Otherwise, if T_k 's snapshot version falls short of the suffix boundary by more than one transaction, then try to abort it. I.e., if $snapshot(T_k) < i - 1$, then try to abort T_k .
- R3.** Otherwise, if there exists an earlier non-aborted transaction in the suffix that overlaps with T_k , is not among T_k 's read versions, and T_k has an antidependency directed upon it, then try to abort T_k .

I.e., if $\exists T_j : snapshot(T_k) < j < k \wedge \neg aborted(T_j) \wedge j \notin readvers(T_k) \wedge writeset(T_j) \cap readset(T_k) \neq \emptyset$, then try to abort T_k .

R4. Otherwise, try to commit T_k .

Aborting and committing of transactions must satisfy *S1*: For *R2*, *R3* and *R4*, if some prior decision state exists, the proposed decision is discarded in favour of the existing decision. For *R1*, a contradiction in decisions between successive applications of Alg. 1b is not possible by the invariants of the algorithm. (See Lemma 7.)

A **safe**point is an optimisation that allows cohorts to eagerly install T_k 's update out of order, concurrently with other such updates. Upon commitment of T_k , the certifier assigns a safe point to it by identifying the last non-aborted transaction in the suffix before T_k that is a direct dependency of T_k . That is, it either has a read-write antidependency directed upon T_k or read-conflicts with T_k or write-conflicts with T_k . If such a predecessor transaction exists, then its logical timestamp is assigned to T_k 's safe point. Otherwise, there is nothing in the suffix that is superseded by T_k ; hence, T_k 's safe point is conservatively assigned the logical timestamp of the transaction that immediately precedes the earliest transaction in the suffix. Formally, for a transaction T_k , let the **conflict set** $\mathcal{C} \triangleq \{T_p \in \mathcal{S} : \neg aborted(T_p) \wedge conflicts(T_p, T_k)\}$, where $conflicts(T_p, T_q) \triangleq readset(T_p) \cap writeset(T_q) \neq \emptyset \vee writeset(T_p) \cap readset(T_q) \neq \emptyset \vee writeset(T_p) \cap writeset(T_q) \neq \emptyset$. (The only nonconflicting intersection is that of the readsets.) $safe point(T_k) \triangleq \max(vers(\mathcal{C}))$ if $\mathcal{C} \neq \emptyset$ else $i - 1$.

The operator $vers(\mathcal{T})$ is the set of the logical timestamps of all transactions in $\mathcal{T}$ defined by $vers(\mathcal{T}) \triangleq \{ver(T) : T \in \mathcal{T}\}$.

The operator $\max(N)$ is the maximum of a finite, nonempty set of zero-inclusive natural numbers N , defined as $\max(N \in \wp(\mathbb{N}_0)) \triangleq \varepsilon n \in N : \forall m \in N : m \leq n$.

D. EXAMPLES

We now provide several examples of certification, covering the rules *R1*–*R4* of Alg. 1b as well as the safe point calculation in Alg. 1c for committed transactions. Refer to Fig. 4, illustrating a suffix $\mathcal{S}$ of *Candidate* messages received from abcast. Candidate transactions are depicted as an ordered sequence; the bold outline indicates the transaction under consideration (i.e., T_k in Alg. 1b). For simplicity, we assume that none of the antidependencies located by *R3* was aborted.

Example 1—acceptance of unconditional writes: T_6 specifies an empty readset in Fig. 4 (a). It is committed immediately by *R1*. In committing T_6 , the certifier invokes Alg. 1c to calculate $safe point(T_6)$. This involves finding the transaction with the most recent transaction that precedes T_6 and conflicts with it. This transaction is T_5 , as $readset(T_5) \cap writeset(T_6) \neq \emptyset$. Hence, $safe point(T_6) = 5$.

We know this to be safe because T_6 does not depend on anything outside of its readset and we only consider deterministic transactions; since $readset(T_6) = \emptyset$, there is no possibility of T_6 having its invariants breached upon installation.

ver	...	4	5	6	7			
readset	...	{x, y}	{x, y}	{}	{v, w}			
writeset	...	{x, y}	{}	{x, y}	{}			
readvers	...	{}	{4}	{}	{}			
snapshot	...	0	0	4	6			

(a)

ver	...	12	13	14	15			
readset	...	{x, y}	{x, y}	{}	{v, w}			
writeset	...	{x, y}	{}	{x, y}	{z}			
readvers	...	{}	{}	{}	{}			
snapshot	...	11	12	5	10			

(b)

ver	...	24	25	26	27	28	29	
readset	...	{x, y}	{x, y}	{}	{v, w}	{x, z}	{x, z}	
writeset	...	{}	{x, y}	{y, z}	{}	{z}	{x, z}	
readvers	...	{}	{}	{}	{}	{25}	{}	
snapshot	...	19	22	25	26	23	22	

(c)

ver	...	30	31	32	33	34	35	
readset	...	{x, y}	{x, y}	{}	{v, z}	{}	{x, z}	
writeset	...	{}	{w, x}	{y}	{y}	{w}	{z}	
readvers	...	{}	{}	{}	{}	{}	{}	
snapshot	...	23	24	25	26	31	31	

(d)

FIGURE 4. Example certification scenarios.

Example 2—straying outside the suffix: $snapshot(T_{15}) = 10$ in Fig. 4 (b), whereas the snapshot boundary is demarcated by T_{12} . $10 < 12 - 1$; thus, T_{15} is conditionally aborted by R_2 .

The behaviour of Alg. 1b is conservative for the assessment of transactions outside the suffix boundary. Perhaps transaction T_{11} antidepends on T_{15} ; there is no way of knowing, so we assume the worst case. Put simply, T_{15} is too stale to be certified; it must be retried or abandoned. Section V offers ways of dealing with this contingency. If $snapshot(T_{15})$ was 11 instead of 10, it would be safe, despite the suffix not starting until 12: R_3 considers transactions ahead of the snapshot, which excludes the transaction to which the snapshot points.

Example 3—antidependency: $snapshot(T_{28}) = 23$ in Fig. 4 (c), which qualifies for assessment by R_3 . The algorithm considers all candidates in the range 24..27. $writeset(T_{25})$ intersects with $readset(T_{28})$; however, $25 \in readvers(T_{28})$, meaning that T_{28} 's cohort installed T_{25} out of order and therefore observed its writes. Next, we consider T_{26} , whose writeset also intersects with $readset(T_{28})$. This time R_3 finds a bona fide antidependency, as T_{28} does not appear to have observed T_{26} 's writes. T_{28} is hence conditionally aborted. T_{28} reads z as of version 25 while T_{26} concurrently installs the successor version of z and commits prior to T_{28} . Intuitively, for this history to be serializable, either 1) T_{28} must observe z at

version 26 or later, or 2) T_{28} must precede T_{26} , or 3) T_{26} must abort, or 4) T_{28} must abort. Only option 4 is viable. (Option 2 is excluded due to a serialization order guarantee discussed in Section IV.L.)

Example 4—conditional commit: $snapshot(T_{35}) = 31$ in Fig. 4 (d), and there are no transactions in the range 32..34 that antidepend on T_{35} . R_4 attempts to commit T_{35} , honouring a previous decision if one was assigned. Assuming T_{35} was previously undecided, its safepoint will be the highest logical timestamp of the transactions in the conflict set, being 33, seeing that $readset(T_{33}) \cap writeset(T_{35}) \neq \emptyset$.

The commitment of T_{35} in R_4 is conditional because there is a chance that T_{35} may have aborted in some previous run of Alg. 1b. This might appear counterintuitive at first, so we elaborate. There is in fact a subtle recursive step concealed in Alg. 1b. Each decision is tied to the decision states of preceding transactions. Suppose we altered the example such that there is some undecided transaction in the range $safepoint(T_{35})..34$ that antidepends on T_{35} , hence the certifier aborts the latter by R_3 . At some later point, the conflicting transaction itself is aborted for whatever reason. A repeat run of Alg. 1b will disregard this antidependency in R_3 and fall through to R_4 , attempting to commit T_{35} . To ensure that decision stability is not compromised, the commitment in R_4 must be conditional. Equivalently, the aborting of transactions in R_2 and R_3 is also conditional.

E. ALGORITHM SPECIFICATION

We now present more formal definitions of the algorithms introduced earlier. There are five subordinate algorithms involved, labelled Alg. 1a through to 1e.

Algorithm 1a Candidate submission

```

1: let  $cpt\_snapshot(T_k) \triangleq picklower(ver(C_k))$ 
2: let  $readset(T_k) \triangleq$  (determine  $T_k$ 's readset)
3: let  $readvals(T_k) \triangleq \perp$  /* initialised later */
4: let  $cpt\_readvers(T_k) \triangleq \perp$  /* initialised later */
   /* atomically capture read values and their versions */
5: begin atomic
6:    $readvals(T_k) \triangleq \{read(x) : x \in readset(T_k)\}$ 
7:    $cpt\_readvers(T_k) \triangleq \{ver(x) : x \in readset(T_k)\}$ 
8: end atomic
   /* locally validate the transaction before submitting for certification */
9:  $validate(readvals(T_k)) \vee \mathbf{exit}$ 
   /* stage the candidate message */
10: let  $xid(T_k) \triangleq$  (assign a globally unique ID to  $T_k$ )
11: let  $writeset(T_k) \triangleq$  determine  $T_k$ 's writeset
12: let  $snapshot(T_k) \triangleq \max(cpt\_snapshot(T_k), \min(cpt\_readvers(T_k)))$ 
   if  $cpt\_readvers(T_k) \neq \emptyset$  else  $cpt\_snapshot(T_k)$ 
13: let  $readvers(T_k) \triangleq \{v \in cpt\_readvers(T_k) : v > snapshot(T_k)\}$ 
14: let  $statemap(T_k) \triangleq$  (some function of the form  $[x, S_x \mapsto S_x']$ )
   /* submit candidate message via abcast */
15:  $abcast((Candidate\ xid(T_k), readset(T_k), readvers(T_k),$ 
    $writeset(T_k), snapshot(T_k), statemap(T_k)))$ 

```

The operator $picklower(n)$ returns some natural number inclusive of zero that is lower than n :

$$picklower(n \in \mathbb{N}_0) \triangleq \varepsilon m \in 0..n : TRUE \quad /* \varepsilon \text{ is Hilbert's epsilon } */$$

The operators $\min(N)$ and $\max(N)$ return the minimum and maximum values, respectively, in the given finite set of zero-inclusive natural numbers N :

$$\min(N \in \wp(\mathbb{N}_0)) \triangleq \varepsilon n \in N : \forall m \in N : m \geq n$$

$$\max(N \in \wp(\mathbb{N}_0)) \triangleq \varepsilon n \in N : \forall m \in N : m \leq n$$

For brevity, we assume that the cohort directly submits candidates over abcast; that is, the agent is collocated with the cohort. In most cases, the role of the agent can be collapsed into the cohort for Alg. 1a. (For example, the agent may be deployed as a library embedded in the cohort process.) We later discuss conditions under which the separation of agents from cohorts is worthwhile.

Algorithm 1b Certification

```

1: let  $\mathcal{S} \triangleq$  (the ordered suffix of transactions  $T_1..T_z$ ;  $T_z$  being the last
   candidate transaction in abcast)
2: upon considering  $T_k \in \mathcal{S}$ :
3:   if
     /* R1: commit a write-only transaction */
4:      $readset(T_k) = \emptyset$ 
5:     then
6:        $commit(T_k)$ 
7:     else if
     /* R2: try to abort  $T_k$  if it falls short of the suffix
       boundary by more than one transaction */
8:        $snapshot(T_k) < i - 1$ 
9:       then
10:         $try\_abort(T_k)$ 
11:      else if
     /* R3: abort  $T_k$  if there exists an earlier transaction  $T_j$  in the suffix
       that overlaps with  $T_k$  that has not aborted, is not among  $T_k$ 's
       read versions, and  $T_j$  antidepends on  $T_k$  */
12:         $\exists T_j \in \mathcal{S} : \wedge snapshot(T_k) < j < k$ 
            $\wedge \neg aborted(T_j)$ 
            $\wedge j \notin readvers(T_k)$ 
            $\wedge writeset(T_j) \cap readset(T_k) \neq \emptyset$ 
13:        then
14:           $try\_abort(T_k)$ 
15:        else
     /* R4: try to commit a transaction if it has made it this far */
16:           $try\_commit(T_k)$ 
17:        end if

```

The operators $commit(T)$ and $abort(T)$ are *fail-fast*: they do not attempt to overwrite the decision if one has already been assigned in the XDB for T , but if the assigned decision contradicts with the proposed decision, they return an error. In practice, these operators may be used to assert system invariants; a contradiction for $commit(T)$ or $abort(T)$ implies an implementation defect or state corruption. Also, $commit(T)$ will skip safepoint calculation for an existing decision, honouring the existing safepoint.

The operators $try_commit(T)$ and $try_abort(T)$ are *fail-safe*. They will not attempt to overwrite the decision, nor will they raise an error if they encounter a contradiction.

Algorithm 1c Safepoint calculation

```

1: let  $\mathcal{S} \triangleq$  (the suffix of transactions from Alg. 1b)
2: upon committing  $T_k$ :
3:   let  $safepoint(T_k) \triangleq$ 
4:     let  $\mathcal{C} \triangleq \{T_p \in \mathcal{S} : \wedge p < k \wedge \neg aborted(T_p) \wedge conflicts(T_p, T_k)\}$ 
5:     if  $\mathcal{C} \neq \emptyset$  then
6:        $\max(vers(\mathcal{C}))$ 
7:     else
8:        $i - 1$ 
9:     end if

```

The operator $\text{conflicts}(T_p, T_q)$ returns *TRUE* iff the given pair of transactions T_p, T_q , have a nonempty intersection between either their writesets or one's readset and the other's writeset:

$$\begin{aligned} \text{conflicts}(T_p, T_q) \triangleq & \\ & \vee \text{readset}(T_p) \cap \text{writeset}(T_q) \neq \emptyset \\ & \vee \text{writeset}(T_p) \cap \text{readset}(T_q) \neq \emptyset \\ & \vee \text{writeset}(T_p) \cap \text{writeset}(T_q) \neq \emptyset \end{aligned}$$

The operator $\text{vers}(\mathcal{T})$ is the set of the logical timestamps of all transactions in $\mathcal{T}$:

$$\text{vers}(\mathcal{T}) \triangleq \{\text{ver}(T) : T \in \mathcal{T}\}$$

Algorithm 1d Out-of-order installation

```

1: upon receiving a decision for  $T_k$  on  $C$ :
2: begin atomic
3:   if  $\text{ver}(C) \geq \text{safepoint}(T_k) \wedge k > \text{ver}(C)$  then
4:      $\text{install}(T_k)$ 
5:   end if
6: end atomic

```

The operator $\text{install}(T_k)$ conditionally installs updates in T_k 's state mapping for those items whose installed versions have not been superseded by a successor of T_k :

$$\begin{aligned} \text{install}(T) \triangleq & \\ & \textbf{for each } w \in \text{writeset}(T) \\ & \quad \textbf{if } \text{ver}(w) < \text{ver}(T) \textbf{ then} \\ & \quad \quad S_w' \leftarrow \text{statemap}(T)[w, S_w] \\ & \quad \quad \text{ver}(w)' \leftarrow \text{ver}(T) \\ & \quad \textbf{end if} \\ & \textbf{end for} \end{aligned}$$

The notation g' denotes the successor state of g .

Algorithm 1e Serial installation

```

1: upon receiving a decision for  $T_k$  on  $C$ :
2: begin atomic
3:   if  $k > \text{ver}(C) \wedge (\nexists T_h : \text{ver}(C) < h < k \wedge \text{committed}(T_h))$  then
4:      $\text{install}(T_k)$ 
5:      $\text{ver}(C)' \leftarrow k$  /* upgrade DB version */
6:   end if
7: end atomic

```

F. PROOF SKETCH

We now prove that Alg. 1 is correct; i.e., it satisfies the criteria *SI*–*S5*. However, instead of directly proving Alg. 1, we propose a weaker algorithm, Alg. 0. This hypothetical algorithm permits all histories that are also permitted by Alg. 1, there are no histories in Alg. 1 proscribed by Alg. 0, and Alg. 1 rejects some histories that Alg. 0 admits; i.e., $\text{Alg. 1} \subset$

Alg. 0. We subsequently prove its correctness, *a fortiori* proving Alg. 1.

The two algorithms are structured identically, with the same number of subordinate algorithms performing the same roles. Alg. 0 is mostly identical to Alg. 1, with one notable difference: we relax the safepoint calculation, ignoring dependencies between transactions.

Algorithm 0c Safepoint calculation

```

1: let  $S \triangleq$  (the suffix of transactions from Alg. 1b)
2: upon committing  $T_k$ :
3:   let  $\text{safepoint}(T_k) \triangleq 0$ 

```

As a result, one of the predicates in Alg. 0d, $\text{ver}(C) \geq \text{safepoint}(T_k)$, reduces to a no-op. Clearly, if $\text{ver}(C) \geq 0$ for all C and $\text{safepoint}(T) = 0$ for all T , then $\text{ver}(C) \geq \text{safepoint}(T)$ for all C and T .

To simplify the proof, we also assume that all transactions' state mappings are idempotent in Alg. 0. Later, we separately show that Alg. 1 supports nonidempotent transactions.

Lemma 1. *Every committed transaction T_i is installed in every cohort C at most once, atomically, and in accordance with its state mapping.*

Proof. We consider the case where T_i is installed in C in some earlier run of Alg. 0; otherwise, the proposition trivially holds. Transactions are installed in Alg. 0d and 0e. In both cases, T_i 's writes are installed by $\text{install}(T_i)$, which applies the result of T_i 's state mapping function. In both cases, the evaluation of the conditional predicate and the subsequent installation are in an atomic step. Hence, once the writeset is installed for some T_i , and the item versions are advanced to i , no transaction T_h , $h \leq i$, can subsequently install the same or older version of any of the items in $\text{writeset}(T_i)$. $\square$

Let $\text{wr}(T_p, T_q)$ mean that T_q reads an item written by T_p , although not necessarily the same version (i.e., no assumption is made as to the relative order of T_p and T_q):

$$\text{wr}(T_p, T_q) \triangleq \text{writeset}(T_p) \cap \text{readset}(T_q) \neq \emptyset$$

From the above definition, $T_p \rightarrow_{\text{wr}} T_q \Rightarrow \text{wr}(T_p, T_q)$.

For a pair of transactions T_i, T_j , we say T_j **observes** T_i (denoted $T_i \triangleright T_j$) if T_i committed before T_j , T_j read some item written by T_i , and for every data item x written by T_i and read by T_j , T_j observes T_i 's write of x or the write of some later transaction T_k , where $k > i$. Formally: $T_i \triangleright T_j \Rightarrow \text{committed}(T_i) \wedge \text{wr}(T_i, T_j) \wedge \forall x \in \text{readset}(T_j) : \exists v \in \mathbb{N}_0 : r_j[x_v] \wedge v \geq i$.

The definition of the *observes* relation trivially implies transitivity when combined with the *wr* operator: $T_i \triangleright T_j \wedge \text{wr}(T_i, T_k) \wedge T_j \triangleright T_k \Rightarrow T_i \triangleright T_k$.

Lemma 2. Reads in Alg. 0 satisfy item cut isolation and monotonic atomic view.

Proof. Reads occur only in Alg. 0a. For I-CI, reads occur at most once in Alg. 0a, therefore a staged transaction cannot observe two versions of the same item. For MAV, if T_j read an item written by T_i , it is assumed that T_i 's state update was completely installed, that is, the writes of T_i were installed exactly once, atomically, and in accordance with its state mapping. This is satisfied by Lemma 1. Additionally, T_j 's reads are atomic over the entire readset, by the atomic assignments of $readvals(T_j)$ and $cpt_readvers(T_j)$ in Alg. 0a. Then for every item read by T_j and written by T_i , T_j observes T_i 's write or that of a more recent transaction. $\square$

Lemma 3. For a pair of transactions T_i, T_j , if $i \in cpt_readvers(T_j) \vee i \leq cpt_snapshot(T_j)$, then for every $x \in readset(T_j) \cap writeset(T_i)$, there is no earlier transaction $T_h, h < i$, that T_j read x from.

Proof. It suffices to prove the lemma for cases $\langle 1 \rangle i \in cpt_readvers(T_j)$ and $\langle 2 \rangle i \leq cpt_snapshot(T_j)$ separately.

For $\langle 1 \rangle$, we pick some x written by T_i and assume there exists some $T_h, h < i$, that T_j read x from, then prove T_h 's absence by contradiction. We consider Alg. 0d and 0e, being the only algorithms that change the state of C_j . In both algorithms, writes are installed monotonically, such that if x is written by some transaction T_v , it is never subsequently updated by T_u , where $u \leq v$. (This is owed to the conditional evaluation of the update, subject to the predicate $ver(w) < ver(T)$ in the definition of $install(T)$.) We know that at least one of the algorithms was run with T_i as input and that T_j read some item from T_i , otherwise i would not be in $cpt_readvers(T_j)$. Therefore, $ver(x) \geq i$. If T_h 's written value was observed by T_j for some x written by T_i , then either $\langle 1.1 \rangle h > i$ or $\langle 1.2 \rangle T_j$ read x from T_h despite $ver(x) \geq i$.

For $\langle 1.1 \rangle$, this is a contradiction, as $h < i$ by the initial assumption.

For $\langle 1.2 \rangle$, this is a contradiction of Lemma 2, stating that T_j 's reads satisfy MAV $\cap$ I-CI: if T_i 's write to x overwrites T_h 's, and T_j read an item from T_i , then T_j could not have read x from T_h in its item cut.

For $\langle 2 \rangle$, again assume T_h exists and consider Alg. 0e, being the only algorithm that both updates values in C_j and upgrades $ver(C_j)$. Recall that $cpt_snapshot(T_j) \leq ver(C_j)$, therefore $i \leq cpt_snapshot(T_j) \Rightarrow i \leq ver(C_j)$. Alg. 0e mandates contiguous applications of transaction updates; thus, we know that if some T_k was installed in C_j , where $k > i$, then T_i was also installed in C_j . Because $i \leq ver(C_j)$ we know that Alg. 0e was run with T_i as input. If T_h 's written value was observed by T_j for some x written by T_i , then either $\langle 2.1 \rangle h > i$ or $\langle 2.2 \rangle T_j$ read T_h 's earlier write.

For $\langle 2.1 \rangle$, this is a contradiction, as $h < i$ by the initial

assumption.

For $\langle 2.2 \rangle$, $ver(C)$ and $ver(x)$ are set atomically with the installation of transactions; thus, if $r_j[ver(C_j), w] \wedge w \geq i$ and T_i wrote to x , then $r_j[ver(x), v] \wedge v \geq i$ by MAV $\cap$ I-CI of Lemma 2. A contradiction if $r_j[ver(x), h]$. $\square$

Lemma 4. For a pair of transactions $T_i, T_j, i \in readvers(T_j) \vee i \leq snapshot(T_j) \wedge wr(T_i, T_j) \Rightarrow T_i \triangleright T_j$.

Proof. It suffices to prove the lemma for cases $\langle 1 \rangle i \in readvers(T_j)$ and $\langle 2 \rangle i \leq snapshot(T_j) \wedge wr(T_i, T_j)$ separately.

For $\langle 1 \rangle$, $readvers(T_j) \subseteq cpt_readvers(T_j)$; thus, $i \in readvers(T_j) \Rightarrow i \in cpt_readvers(T_j)$. This implies that T_j read a data item in $readset(T_j)$ from T_i , which implies that T_i committed before the read occurred. Also, $i \in readvers(T_j)$ implies that, for all x written by T_i and read by T_j , there is no earlier transaction $T_h, h < i$, that T_j read x from by Lemma 3.

For $\langle 2 \rangle$, $snapshot(T_j)$ is either $\langle 2.1 \rangle cpt_snapshot(T_j)$ or $\langle 2.2 \rangle \min(cpt_readvers(T_j))$.

For $\langle 2.1 \rangle, i \leq cpt_snapshot(T_j)$ implies that $ver(C_j) \geq i$ at the time of the read, by the definition of $cpt_snapshot(T)$. C_j has thus installed T_i 's update, which requires T_i to have committed before T_j started. Their writeset-readset intersection implies that T_j read at least one item that T_i has written. Also, $i \leq cpt_snapshot(T_j)$ implies that for all x written by T_i and read by T_j , there is no earlier transaction $T_h, h < i$, that T_j read x from by Lemma 3.

For $\langle 2.2 \rangle, i \leq \min(cpt_readvers(T_j))$ implies that $\forall v \in cpt_readvers(T_j) : v \geq i$; thus, T_i committed before any of T_j 's reads. It also implies that $i \in cpt_readvers(T_j)$; thus, T_j read an item written by T_i , and for all x written by T_i and read by T_j , there is no earlier transaction $T_h, h < i$, that T_j read x from by Lemma 3. $\square$

In the derivation of $readvers(T)$, given by the set comprehension $\{v \in cpt_readvers(T) : v > snapshot(T)\}$, the filter predicate $v > snapshot(T)$ is not used in the proof of Lemma 4. This is because $v > snapshot(T)$ can only restrict the set $readvers(T)$; if the claim $i \in readvers(T_j) \Rightarrow T_i \triangleright T_j$ in Lemma 4 was satisfied without the filter predicate, it is a *fortiori* satisfied with the predicate applied.

The assignment of $readvers(T_k)$ in Alg. 0a and its subsequent read in Alg. 0b are needed to support out-of-order updates. Ordinarily, with serial installation of updates, transactions are concurrent iff they overlap. For Alg. 0, as it turns out, this is a necessary but insufficient criterion for concurrency. Consider a pair of overlapping transactions T_i, T_j , such that $snapshot(T_j) < i \wedge snapshot(T_i) < j$. Without loss of generality, assume that $i < j$, C_j has installed T_i 's updates by Alg. 0d, and for all items written by T_i and read by T_j , the version read by T_j is at least as recent as that installed by T_i . It follows that either T_j has been submitted serially after T_i , or T_j reads no items written by T_i .

This is the archetypal scenario for out-of-order updates. In the absence of $readvers(T_j)$, the certifier will erroneously assume an antidependency $T_j \rightarrow_{rw} T_i$ and summarily abort T_j . Worse still, this will happen in every such case, obviating all benefits of out-of-order updates. By disclosing $readvers(T_j)$, the cohort signals to the certifier that certain transactions preceding T_j in commit order, while overlapping with T_j , are in fact serial with T_j .

We now reveal the **extended definition of concurrency** that accounts for out-of-order updates. Two transactions are concurrent iff if there is a period in which they overlap and neither transaction observes the other's updates. Formally, a pair of transactions T_i, T_j , are concurrent iff $snapshot(T_i) < j \wedge snapshot(T_j) < i \wedge \neg(T_i \triangleright T_j) \wedge \neg(T_j \triangleright T_i)$. Conversely, they are serial iff $snapshot(T_i) \geq j \vee snapshot(T_j) \geq i \vee T_i \triangleright T_j \vee T_j \triangleright T_i$.

Next, we introduce Fig. 5 to guide several of the forthcoming proofs. We represent a transaction T_i as a horizontal segment demarcated by a pair of vertices r_i (occurring at $snapshot(T_i)$) and w_i (occurring at $ver(T_i)$, or simply, i). The horizontal axis depicts logical time; i.e., points to the left relate to logical timestamps that occur before those to the right. All reads of T_i occur at an instant represented by its r_i vertex; all writes of T_i occur at an instant represented by its w_i vertex, sometime after the reads. Transactions have nonzero duration; thus, r_i is always to the left of w_i . Dependencies are shown as directed edges between transaction vertices. This is similar in spirit to a special case of Adya's **unfolded serialization graph** (USG) [2], where transactions are restricted to reading and writing exactly once, at the beginning and end of their lifespan, respectively. It is also similar to the SC-graph technique of Shasha *et al.* [8], adopted later by Fekete *et al.* [1].

We depict an antidependency $T_i \rightarrow_{rw} T_j$ as a dashed edge from the r_i vertex to w_j . $T_i \rightarrow_{rw} T_j$ is said to be **backward-facing** if $i > j$; otherwise, $T_i \rightarrow_{rw} T_j$ is said to be **forward-facing** if $i < j$. $i \neq j$ owing to the uniqueness of version numbers. Backward-facing antidependencies only occur among concurrent transactions. Note, it is the transactions' relative commit order that determines the direction of the antidependency, not the spatial orientation of the edge (i.e., the direction it might appear to be facing on the Cartesian plane). In Fig. 5 (a), the antidependency $T_k \rightarrow_{rw} T_v$ is backward-facing, while $T_k \rightarrow_{rw} T_v$ is forward-facing in Fig. 5 (b).

FIGURE 5. Dependency graph showing antidependencies.

Dependencies of type *write-write* and *write-read* are depicted with a solid edge (annotated with ww and wr , respectively) from the appropriate vertices of the respective transactions, i.e., w_i to w_j for $T_i \rightarrow_{ww} T_j$ and w_i to r_j for $T_i \rightarrow_{wr} T_j$. Note, ww and wr dependencies always point in a direction such that T_i occurs before T_j for both $T_i \rightarrow_{ww} T_j$ and $T_i \rightarrow_{wr} T_j$. For the $T_i \rightarrow_{wr} T_j$ case, the pair must execute serially, whereas for $T_i \rightarrow_{ww} T_j$, the pair may execute concurrently. Also, a ww dependency cannot face backward in STRIDE because write versions follow logical timestamp order: if T_j overwrites an item written by T_i , then T_j 's version is greater than T_i 's.

As a corollary of the aforementioned definitions, given that ww and wr dependencies may only point forward, for a directed cycle to occur among a set of transactions, there must exist at least one backward-facing antidependency in the directed graph [3]. In other words, dependencies must “go back in time” somehow for a cycle to materialise [1]; such a cycle is depicted in Fig. 5 (c).

Lemma 5. *Alg. 0b proscribes a backward-facing antidependency between all pairs of committed transactions. I.e., $\nexists T_k, T_v : T_k \rightarrow_{rw} T_v \wedge v < k$; T_k and T_v are depicted in Fig. 5 (a).*

Proof. We begin by assuming that Alg. 0b commits a transaction T_k that partakes in such an antidependency and prove its absence by contradiction. In addition to the source transaction, T_k , an antidependency requires a target, T_v , upon which the antidependency is incident, i.e., $T_k \rightarrow_{rw} T_v$, and $v < k$ by the assumption of the lemma. There are two separate cases under which Alg. 0b may commit a transaction: (1) by the action of rule $R1$, and (2) by that of $R4$.

For (1), T_k is committed iff $readset(T_k) = \emptyset$. Yet, $writeset(T_v) \cap readset(T_k) \neq \emptyset$ by the definition of an antidependency. No set can produce a nonempty intersection with an empty set, therefore $writeset(T_v) \cap readset(T_k) = \emptyset$: a contradiction.

For (2), T_v must be concurrent with T_k by the definition of a backward-facing antidependency; hence, $snapshot(T_k) < v$, by the definition of concurrency. Due to out-of-order updates on C_k , it may be that $v \in readvers(T_k)$ even if T_k overlaps with T_v . (T_k 's snapshot version may lag v , yet T_k may observe T_v 's writes.) In that case, $T_v \triangleright T_k$ by Lemma 4. A contradiction if $T_k \rightarrow_{rw} T_v$.

In the remaining case, $v \notin readvers(T_k)$. According to $\neg R2$, if T_v exists, it must be in the suffix, because $snapshot(T_k) \geq i - 1$ and $v > snapshot(T_k)$, therefore $v \geq i$. Thus, we have for T_v : $v < k \wedge v > snapshot(T_k) \wedge \neg aborted(T_v) \wedge v \notin readvers(T_k) \wedge writeset(T_v) \cap readset(T_k) \neq \emptyset$. This is a direct contradiction of $\neg R3$, necessary for T_k 's commitment by $R4$. $\square$

Lemma 6. *Alg. 0b proscribes a directed cycle in every subset of committed transactions in all admissible execution*

histories.

Proof. A directed cycle involves at least two transactions and at least one antidependency by the result of Adya *et al.* [3]. It has been shown in *Lemma 5* that a backward-facing antidependency is proscribed by the algorithm; hence, for a cycle to exist, a dependency of some type must project forward, starting from some transaction T_k , incident upon T_v (i.e., $T_k \rightarrow? T_v$, $v > k$, where $\rightarrow?$ denotes any dependency type), transit through zero or more intermediate transactions (involving any dependency type), then close upon T_k from some T_u , such that $T_u \rightarrow? T_k$ and $u < k$. (*Lemma 5* proscribes the $u > k$ case.) This cycle is exemplified in Fig. 5 (c). Assume T_u exists, with the requisite cyclic dependency path originating from T_k . Since $u < k$ and $v > k$, then $u < v$. The only dependency type that may face backward is an antidependency, which is proscribed by *Lemma 5*. This leads to a contradiction since T_v must (directly or indirectly) project a backward-facing antidependency upon T_u for a cycle to materialize. $\square$

We use the general definition of a cycle for *Lemma 6* of Adya *et al.* [3], rather than the specific result of Fekete *et al.* [1] for snapshot isolation, that requires a pivot transaction, T_{pivot} , and a pair of (possibly identical) transactions T_u, T_v , such that $T_u \rightarrow_{rw} T_{pivot} \rightarrow_{rw} T_v$, $T_u \neq T_{pivot}$, $T_v \neq T_{pivot}$. The strongest cohort isolation model required by Alg. 0 is MAV $\cap$ I-CI per *Lemma 2*, which is weaker than generalized snapshot isolation (and by extension, snapshot isolation) [2], owing to out-of-order updates; therefore, theorems that hold under SI will not hold here.

Lemma 7. *No decision made by Alg. 0b can be contradicted by a subsequent run of Alg. 0b, involving all transactions from the previous run.*

Proof. The only definitive decisions are made by the actions of rule *R1*; *R2*, *R3* and *R4* merely attempt to assign a decision, honouring the previous decision if one exists. *R1* can only commit a transaction; it cannot abort. It suffices to show that if a transaction was aborted, then no subsequent run of Alg. 0b will result in a commit by *R1*. Assume some transaction T_k was aborted, implying *R2* $\vee$ *R3*. Therefore, $\neg R1$; that is, $readset(T_k) \neq \emptyset$. The readset of T_k is a constant; therefore, if $readset(T_k) \neq \emptyset$ in some run, then $readset(T_k) \neq \emptyset$ in every run. *R1* never commits T_k . $\square$

The result of *Lemma 7* is important for two reasons. First, the system model assumes fail-recovery; therefore, a process must safely resume any operation when recovering from failure. A certifier cannot skip over a pending transaction or issue a contradictory decision. Second, multiple certifier instances may be operating simultaneously over the same set of transactions. It may be, for example, due to one certifier

appearing to have failed as a result of a false suspicion by an eventually perfect failure detector, and another certifier instance taking over, not sensing that the former is in fact operational. Even in the absence of external coordination, and without the guarantee of mutual exclusion, parallel certifiers cannot lead to divergent outcomes. In another example, multiple certifier instances may be intentionally operating in parallel for load balancing, for example, where in a group of n certifiers $F_1, F_2, \dots, F_n$, a certifier F_i decides transaction T_k , such that $k \bmod n = i$. If the process group is subject to dynamic membership, for example, autoscaling, where the group population may expand and contract automatically depending on the load, changes to n may lead to conflicting work assignments; i.e., two certifiers may attempt to assess the same candidate. *Lemma 7* guarantees decision stability in all scenarios.

Lemma 8. *Alg. 0 is correct for idempotent state mappings.*

Proof. Alg. 0b is the algorithm responsible for certification, hence it is the cynosure of this proof. It relies on lemmas that involve key parts of Alg. 0b and other algorithms.

For the *stability of decisions* criterion, *S1*, this is discharged by *Lemma 7*.

For the *safe abort* criterion, *S2*, an aborted transaction is not installed in any cohort; hence, it has no bearing on the system state, other than to assign an abort decision in the XDB. The status of a transaction is only consulted in *R3*, in the predicate $\neg aborted(T_i)$ inside an existential quantifier. The correctness of this case is proven separately by *Lemma 7*.

For the *serializability* criterion, *S3*, it suffices to show that there is no subset of committed transactions S in the resulting history, whose constituents form a directed cycle in the serialization graph [3]. This is discharged by *Lemma 6*.

For the *nontriviality* criterion, *S4*, only a transaction received from abcast is subject to certification, which must have been submitted by Alg. 0a according to the invariants of abcast; in turn, the transaction must have been initiated by some cohort.

For the *agreement* criterion, *S5*, if a transaction is decided for some outcome, then it cannot subsequently have its state altered by *S1* and all cohorts see the same decision value for that transaction by the invariants of abcast and *Lemma 7*. $\square$

Theorem 1. *Alg. 1 is correct for idempotent state mappings.*

Proof. By the relation Alg. 1 $\subset$ Alg. 0 for idempotent writes and the result of *Lemma 8*. $\square$

G. CONSISTENCY MODELS

In Alg. 0, we permit the out-of-order installation of updates, provided the transaction's version is greater than the baseline version of the cohort. Recall, the $ver(C) \geq safepoint(T_k)$ predicate in Alg. 0d was effectively short-circuited by the invariant $safepoint(T_k) = 0$. In Alg. 1, what is the purpose of limiting out-of-order assignments, thereby restricting the set of admissible histories, and reducing concurrency in the process⁴, if Alg. 0 is already safe as it is?

Consider the guarantees provided by Alg. 0, vis-à-vis the safepoint calculation and out-of-order updates. For simplicity, we assume that all state mappings are idempotent for now. For every transaction T_i , writes may be installed at any point, provided $i > ver(C)$. Once an update for T_i is installed, all updates for $T_h, h \leq i$, are thereafter prohibited by the predicate in $install(T)$. Thus, updates under Alg. 0 are both atomic and monotonic: all items in T_i 's state mapping are installed or none, and once any transaction $T_j, j > i$, reads the updates of T_i , then for all items written by T_i , a read by T_j cannot return a value installed by some transaction $T_g, g < i$. This equates *Monotonic Atomic View* (MAV) by Bailis *et al.* [5]. Specifically, MAV only prescribes monotonicity from the perspective of any given transaction performing the read, but not across transactions [5]. For example, under MAV, a newer transaction T_k , may read an older version of the same item read by T_j , where $j < k$.

Next, consider the atomicity of reads in Alg. 0a. For every staged transaction T_i , we require the capture of $readvals(T_i)$ and $cpt_readvers(T_i)$ to proceed in an atomic step, reading from a uniform snapshot of the database. This property is labelled as *Item Cut Isolation* (I-CI) by Bailis *et al.* [5].

Combining MAV with I-CI, reads in Alg. 0a cannot observe the *read skew* anomaly (A5A in Berenson *et al.* [6]). MAV $\cap$ I-CI is equivalent to the *read atomic* (RA) isolation level from Bailis *et al.* [7]. According to the authors, "a system provides RA isolation if it prevents fractured reads phenomena and also proscribes [Adya's] phenomena G0 [write cycles], G1a [aborted reads], G1b [intermediate reads], G1c [circular information flow]," where *fractured reads* are those that are not sourced from a cut.

The main problem with RA is that it is susceptible to the **G-update** phenomenon of Adya [2]; i.e., single antidependency cycles with update transactions. We briefly recount the phenomenon here; however, Adya provides the most comprehensive treatment. A history H and transaction T_i show phenomenon *G-update* if a DSG containing all update transactions of H and T_i contains a cycle with at least one antidependency edge. Adya also introduces the **no-update-conflict-misses** isolation level and states its equivalence with *G-update* [2]. In summary, if T_i depends on T_j , it must not miss the effects of T_j and all update transactions that T_j depends or antidepends on.

To explore the problem, consider the following scenario: A leaderboard system is tracking the ranks of three riders in a speedway race, $\{A, B, C\}$, with ranks assigned from the set 1..3. Upon each successive lap, the leaderboard is updated by assigning new ranks as appropriate. The updates are recorded in a primary database and asynchronously installed in a replica database, where they are later read and displayed on a spectator screen overlooking the grandstand. Updates are applied atomically at the replica but in arbitrary order. All read-only transactions on the replica satisfy RA isolation.

Riders are initially ranked $\{A:1, B:2, C:3\}$ on the starting grid, installed by transaction T_0 . After the first lap, B overtakes A , issuing T_1 . After the second lap, C overtakes A , issuing T_2 . The transaction history on the primary is:

$$H_{primary} = w_0[A, 1], w_0[B, 2], w_0[C, 3], c_0, \\ w_1[A, 2], w_1[B, 1], c_1, w_2[A, 3], w_2[C, 2], c_2$$

A query transaction T_q executes on the replica, seeing the effects of transactions T_0 and T_2 ; T_1 is yet to be installed:

$$H_{replica} = w_0[A, 1], w_0[B, 2], w_0[C, 3], c_0, \\ w_2[A, 3], w_2[C, 2], c_2, \\ r_q[A, 3], r_q[B, 2], r_q[C, 2], c_q$$

As a result, T_q observes the state $\{A:3, B:2, C:2\}$ —a clear breach of invariants, wherein two riders are simultaneously ranked 2nd. Inspecting the transaction dependencies in $H_{replica}$, we observe the following:

$$\begin{array}{ll} r_q[B] \rightarrow_{rw} w_1[B] & w_2[A] \rightarrow_{wr} r_q[A] \\ w_2[C] \rightarrow_{wr} r_q[C] & w_1[A] \rightarrow_{ww} w_2[A] \\ w_0[C] \rightarrow_{ww} w_2[C] & w_0[A] \rightarrow_{ww} w_1[A] \\ w_0[B] \rightarrow_{ww} w_1[B] \end{array}$$

Fig. 6 depicts the resulting DSG, showing the dependency cycle $T_1 \rightarrow_{ww} T_2 \rightarrow_{wr} T_q \rightarrow_{rw} T_1$. The cycle contains one antidependency, as per the *G-update* phenomenon. Had T_q observed T_1 's update, it would read-depend on T_1 , avoiding the cycle.

FIGURE 6. Missed update of T_1 causing a cycle in $H_{replica}$.

The natural question might be: what would occur in Alg. 0, given our cohorts are no more consistent than RA? Namely, would the serializability of read-only transactions be somehow compromised? The answer: *no, provided they are certified*. In the example above, $readset(T_q) = \{A, B, C\}$, $snapshot(T_q) = 0$, $readvers(T_q) = \{2\}$, $writeset(T_q) = \emptyset$. Assume $ver(T_q) = 3$, as it appears after all other transactions

⁴ This is an obvious and invariable consequence of rejecting histories [4].

considered here. Because T_1 precedes T_q in the abcast sequence, Alg. 0b will reject T_q either by R_2 —if T_q falls short of the suffix boundary, or by R_3 —as it locates T_1 that antidepends on T_q , occurs after $\text{snapshot}(T_q)$ and is not among $\text{readvers}(T_q)$.

In Alg. 0, any query, even one void of writes, must be certified before its results may be used safely elsewhere in the system. (One might make exceptions for analytics-style queries that can make do with an approximation of the global state.) Although Alg. 0 is safe with certification, it would be ideal if read-only transactions could be handled more efficiently.

As a minor segue, the selective allowance for missing updates has been successfully employed in large scale geo-replicated systems, for example, in *Parallel Snapshot Isolation* (PSI) of Sovran *et al.* [8], where the penalty of synchronization is paid in terms of increased latency, hence some pragmatic performance-consistency trade-off is sought. However, PSI guarantees causality of some operations and employs a combination of *preferred sites* and *conflict-resistant data types* (CRDTs) to either localise conflicts or avoid them altogether. Unfortunately, this is not a property of RA (or $\text{MAV} \cap \text{I-CI}$, the two being equivalent).

H. FALSE ANTIDEPENDENCY IDENTIFICATION

Alg. 0b employs a conservative tactic for identifying prospective antidependencies. The predicate in R_3 is a sufficient, but not a necessary condition for T_j to antidepend on T_k . Consider the following history, comprising committed transactions T_1 – T_3 and an undecided transaction T_4 . T_1 installs the states of x and y . T_2 and T_3 both update x . T_4 subsequently reads both x and y .

$$H_{\text{false-rw}} = w_1[x, 10], w_1[y, 10], c_1, w_2[x, 20], c_2, \\ w_3[x, 30], c_3, r_4[x_3, 30], r_4[y_1, 10]$$

Assume that T_1 was the last transaction to have been serially installed on T_4 's cohort. T_3 was installed ahead of T_2 , such that $\text{snapshot}(T_4) = 1$, $\text{readvers}(T_4) = \{3\}$ and $\text{readset}(T_4) = \{x, y\}$. $H_{\text{false-rw}}$ is nonetheless serializable because T_3 masks T_2 's writes.

Consider the behaviour of Alg. 0b with $\mathcal{S}$ in the range $T_1..T_4$. $\neg R_1 \wedge \neg R_2$ for T_4 , therefore Alg. 0b evaluates R_3 . There are two non-aborted transactions in $\mathcal{S}$ located between $\text{snapshot}(T_4)$ and $\text{ver}(T_4)$ that satisfy the set intersection criterion: T_2 and T_3 . $T_3 \triangleright T_4$ because $3 \in \text{readvers}(T_4)$; hence, T_3 is disregarded by R_3 . However, $2 \notin \text{readvers}(T_4)$, tripping R_3 for T_2 , despite T_4 reading from a newer version of x . R_3 is not “trained” to recognise the transitive relationship between T_2 and T_4 via T_3 , hence T_4 is aborted unnecessarily. Clearly, this behaviour is suboptimal.

I. OPTIMISING READ-ONLY TRANSACTIONS

Read-only transactions are common among many workloads [2] and finding efficient ways for ensuring their consistency is important. Ideally, we would like a strong consistency model that closely approximates serializability for most read-only workloads, with only the most stringent of read-only transactions requiring explicit certification. Such a model exists: **update serializability**, communicated by Hansdah and Patnaik [4]. Roughly speaking, update serializability ensures that read-only transactions observe a state that is consistent; i.e., preserving all invariants. Adya also presents an account of update serializability [2], labelling it **PL-3U**. This is a slight relaxation of regular serializability, labelled **PL-3**.

In PL-3U, transactions T_i and T_j may observe a serializable database state, but unlike PL-3, the serial ordering observed by both transactions could vary. The difference between PL-3U and PL-3 is exemplified in the following: consider three transactions, T_0 , T_1 , T_2 , updating the set of items $\{x, y\}$, such that T_1 and T_2 do not conflict. The history also contains a pair of read-only transactions, T_v and T_u .

$$H_{3U} = w_0[x, 1], w_0[y, 2], c_0, w_1[x, 10], c_1, w_2[y, 20], c_2, \\ r_u[x_0, 1], r_u[y_1, 20], c_u, r_v[x_1, 10], r_v[y_0, 2], c_v$$

The dependencies of H_{3U} are depicted in Fig. 7.

FIGURE 7. DSG of H_{3U} .

Note the cycle $T_1 \rightarrow_{wr} T_v \rightarrow_{rw} T_2 \rightarrow_{wr} T_u \rightarrow_{rw} T_1$. Each of T_u and T_v individually observes a state that passes for serializable in the absence of the other. However, they do not agree on the order of the update transactions. According to T_u , the serial order is (T_0, T_2, T_1) . According to T_v , the serial order is (T_0, T_1, T_2) . Thus, read-only transactions may induce the reversal of the order of update transactions. Nonetheless, both transactions observe a consistent state, and provided the two do not directly communicate in a way that makes these discrepancies apparent, update serializability is as good as serializability [2].

Unsurprisingly, H_{3U} can be made serializable by deleting any one of the two read-only transactions. Formally, a history H is **update serializable** (USR; is in USR) if every history H' obtained by deleting all but one read-only transaction from H is serializable. In the absence of read-only transactions, H is in USR iff H is in SR [4]. Additionally, $\text{SR} \subset \text{USR}$.

Resuming the discussion on Alg. 1 and its comparison to Alg. 0, we can now state the cardinal reason for the demarcation of safe-points: *update serializability*. The careful choice of safe-point placement in Alg. 1c and its subsequent enforcement in Alg. 1d ensures that histories on the cohorts satisfy the *no-update-conflict-misses* invariant. That is, the G -

update phenomenon is proscribed, and uncertified reads are thereby consistent.

For the forthcoming propositions, let $installed(C, T)$ denote that transaction T is installed on cohort C .

Lemma 9. *For committed transactions T_i, T_j , if $i < safepoint(T_j)$ and T_j is installed on some cohort, then T_i is also installed on that cohort. I.e., $i < safepoint(T_j) \wedge installed(C, T_j) \Rightarrow installed(C, T_i)$ for all C .*

Proof. By Alg. 1d, $installed(C, T_j) \Rightarrow ver(C) \geq safepoint(T_j)$, for all C . Also, $ver(C)$ is only assigned in Alg. 1e, wherein $\forall i, C : ver(C) \geq i \Rightarrow installed(C, T_i) \vee aborted(T_i)$. Combining the two: $installed(C, T_j) \Rightarrow (\forall i : i \leq safepoint(T_j) \Rightarrow installed(C, T_i) \vee aborted(T_i))$. $\square$

Theorem 2. *Alg. 1 satisfies update serializability for uncertified read-only transactions.*

Proof. By the definition of *no-update-conflict-misses*, if T_i depends on T_j , it must not miss the effects of T_j and all update transactions that T_j depends or antidepends on [2]. It suffices to show that, if $installed(C, T_k)$, then $\forall T_p : T_p \rightarrow_r T_k \Rightarrow installed(C, T_p)$. We consider two separate cases: $\langle 1 \rangle T_p \rightarrow_{wr} T_k$ or $T_p \rightarrow_{ww} T_k$, and $\langle 2 \rangle T_p \rightarrow_{rw} T_k$. We also assume that every transaction T_i staged on C_i , having read $ver(C_i) = n$ in Alg. 1a, also observes the effects of transactions $T_1..T_n$ installed on C_i , by the definition of Alg. 1a and the fundamental guarantees of the underlying database.

For $\langle 1 \rangle$, $T_p \rightarrow_{wr} T_k$ implies that $p < k$ and $writeset(T_p) \cap readset(T_k) \neq \emptyset$. Similarly, $T_p \rightarrow_{ww} T_k$ implies that $p < k$ and $writeset(T_p) \cap writeset(T_k) \neq \emptyset$. (The difference is in the choice of the intersecting sets.) Let $\mathcal{S}$ be the suffix of transactions $T_i..T_z$, T_z being the last submitted transaction. $\mathcal{S}$ is used in Alg. 1b and 1c. Examining Alg. 1c, we consider two sub-cases: $\langle 1.1 \rangle T_p \notin \mathcal{S}$ and $\langle 1.2 \rangle T_p \in \mathcal{S}$.

For $\langle 1.1 \rangle$, if $T_p \notin \mathcal{S}$, it means that $p < i$ and $safepoint(T_k) = i - 1$. Therefore, $p \leq safepoint(T_k)$, implying that $installed(C, T_p)$ by Lemma 9.

For $\langle 1.2 \rangle$, if $T_p \in \mathcal{S}$, it means that $p \geq i$ and $T_p \in \mathcal{C}$ and $safepoint(T_k) = \max(vers(\mathcal{C}))$. Therefore, $safepoint(T_k) \geq p$ by the definition of $\max(N)$, implying that $installed(C, T_p)$ by Lemma 9.

For $\langle 2 \rangle$, $T_p \rightarrow_{rw} T_k$, $readset(T_p) \cap writeset(T_k) \neq \emptyset$. We consider two cases: $\langle 2.1 \rangle p < k$ and $\langle 2.2 \rangle p > k$.

For $\langle 2.1 \rangle$, if $p < k$, the proof of is identical to that of $\langle 1 \rangle$ and is omitted for brevity.

For $\langle 2.2 \rangle$, if $p > k$, it implies a backward-facing antidependency, which is prohibited by Lemma 5. $\square$

Earlier, we remarked on the propensity of Alg. 0 to abort transactions due to false antidependency identification. We could modify R3 to cater for the $H_{false-rw}$ scenario (and others like it), extending the predicate to ignore apparent conflicts

that have been superseded by a later transaction. There is no need, as it turns out: Alg. 1 de facto caters to this scenario by its safepoint calculation. Consider the reason behind the false abort. The underlying problem is that Alg. 0 permits the *G-update* phenomenon: T_2 is absent from T_4 's snapshot despite being a write-write dependency of T_3 which, in turn, is a write-read dependency of T_4 . Alg. 1, on the other hand, proscribes *G-update*, meaning that T_2 must be in T_4 's snapshot for it to have observed T_3 's writes. Under Alg. 1, $snapshot(T_4) = 2$ and $readvers(T_4) = \{3\}$; hence, $\neg R3$ in Alg. 1b, and T_4 is eventually committed by R4.

Update serializability is a sufficient isolation property for many read-only workloads [2]; however, there may be occasional queries where PL-3U is insufficient. Is certification the only way of achieving serializability for read-only transactions? *Not in the least.*

Recall that the opportunistic out-of-order update mechanism of Alg. 1c and 1d is an optimisation that fast-tracks the state of replicated items, so that update transactions can be seeded from a more up-to-date cohort state, reducing the temporal overlap between transactions. This is unequivocally useful for update transactions because it lowers the rate of conflicts. Suppose, however, one or more cohorts were dedicated solely for handling read-only transactions, while others could process a mixture of read-only and update transactions. Then for the dedicated read-only cohorts, one could disable Alg. 1d altogether and apply all updates serially, using batching to maximise throughput. By simply splitting the workloads, we obtain:

- PL-3 for uncertified read-only transactions;
- PL-3U for uncertified read-only transactions that benefit from observing a more recent state; and
- Strict serializability for all certified transactions.

On idempotence: During our proof of Alg. 0 we assumed that all state mappings were idempotent. This tightening of the model is necessary because Alg. 0 disregards dependencies in its safepoint calculation. A nonidempotent state mapping produces a value that depends on the prior state, which may be arbitrary according to Alg. 0. To satisfy nonidempotent state mappings, it is necessary to show that transactions are installed deterministically on every cohort, and if a transaction reads some value for an item x on some cohort, then it reads the same value for x on every cohort at which it is installed. This, in turn, requires us to show that the immediate predecessor of x is installed on every cohort before the nonidempotent state mapping is exercised.

Theorem 3. *Alg. 1 is correct for nonidempotent state mappings.*

Proof. Alg. 1 is correct for idempotent state mappings by Theorem 1. The proof is extended by reducing every nonidempotent state mapping to an idempotent one. It suffices to show that $\langle 1 \rangle$ every update is applied at most once and, $\langle 2 \rangle$ for any updated item, the most recent state of the item is installed on the cohort before the update is

applied.

Obligation (1) is discharged by *Lemma 1*.

Obligation (2) is discharged by *Theorem 2*. Specifically, if the installation of every transaction T_j on every cohort C implies the installation of all of T_j 's dependencies on C , and every x in $writeset(T_j)$ had been updated by one of T_j 's direct dependencies T_i , then T_i is installed on C . $\square$

PL-3U offers another useful property in relation to Alg. 1; namely, the mixing of a direct $wr/wr/(rw)$ (anti)dependency and an *observes* relation creates transitivity: $T_i \rightarrow T_j \wedge wr(T_i, T_k) \wedge T_j \triangleright T_k \Rightarrow T_i \triangleright T_k$ because T_j is installed after T_i . We exploit this property in the subsequent proof of Alg. 2 in Section V.

J. SUPPORT FOR SERIALIZABLE BLIND WRITES

Compare the antidependency identification approach of STRIDE with other prominent optimistic and multiversion concurrency control systems. Conventional SI relies exclusively on write conflict identification [6], indiscriminately rejecting concurrent transactions whose writesets intersect. *Serializable Snapshot Isolation* (SSI) expands upon the write-write conflict identification of SI by identifying a *dangerous structure*—a condition wherein a triplet of transactions exhibit an adjacent pair of read-write antidependencies. The condition in SSI is sufficient but not necessary for cycle identification [1]. *Precisely Serializable Snapshot Isolation* (PSSI) elaborates upon SSI with complete serializable graph testing [10]. PSSI is more permissive than SSI, in that it admits a larger set of histories while still rejecting all nonserializable histories. *Serializable Generalized Snapshot Isolation* (SGSI) extends GSI in a replicated environment, providing additional performance and availability while guaranteeing one-copy serializability.

While SI is not serializable, it is nonetheless a useful property [6] for many systems and offers high levels of concurrency, as reads are not blocked by writes, and vice versa. SI prevents a broad range of anomalies: dirty reads, read skew, nonrepeatable reads and lost writes [6]. It also respects the real-time precedence order of nonconflicting transactions—a favourable trait it shares with strict serializability and linearizability [14] while being neither of those things. Importantly, it is the archetypal isolation model upon which its descendants—SSI, PSSI and SGSI—are built. The latter offer serializability while rejecting the same histories that SI would ordinarily reject due to write conflicts.

We suggest that write conflict elimination is a “hangover” from SI that was designed as a crude but effective mechanism for avoiding histories that might contain lost writes. Due to its known limitations, namely, the lack of antidependency testing [1], SI is unnecessarily conservative, eliminating histories that are otherwise serializable [2]. Specifically, SI's requirement that the writesets of concurrent transactions be disjoint is a

sufficient condition for identifying lost updates, but not a necessary one. Consider a history containing a blind write:

$$H_{blind} = w_0[x], c_0, r_1[x_0], r_2[x_0], w_1[z], w_2[z], c_1, c_2$$

The serialization graph of H_{blind} is depicted in Fig. 8, clearly indicating that the history is serializable with the conflict equivalent serial order (T_0, T_1, T_2) ; however, it is proscribed by SI [2]. Furthermore, H_{blind} is also permitted by strict serializability because T_1 and T_2 are concurrent. Finally, H_{blind} is also permitted by *Commitment Ordering* [15] and Φ -serializability [16], both of which are stronger still.

FIGURE 8. Serialization graph of H_{blind} .

Adya suggests that blind writes are rare, without submitting empirical evidence in support of this [2]. While this viewpoint may have been plausible at some point, it is not unequivocally true today. There are several classes of systems where reads are routinely disjoint with writesets; for example, gossiping protocols, content caching, sensor networks, wagering systems⁵ and financial markets. There is a frequent requirement among such systems to record the most recent value of some data item, such that the update is unconditional on prior values; for example, to record the most recent temperature reading (it does not matter what the previous reading was); or replace a value in a cache with a newer value, possibly concurrently from multiple processes; or update the odds of a betting proposition or the market price of a financial instrument. In such systems, unconditional concurrent updates to the same item often result in either the same value or in values that converge with time. Another set of use cases concerns atomic in-place operations, where items are updated without requiring a prior read; for example, in atomic counters. This is a typical requirement for clickstream analytics, complex event processing, etc., where tallies are transactionally incremented when specific conditions are observed, and the previous value of the tallied item need not be a condition of the transactions in question.

The lack of proper antidependency analysis in SI means that transactions containing blind writes are indiscriminately aborted, leading to unnecessary retries that eventually succeed, often after several attempts (depending on the level of concurrency).

⁵ The corresponding author's principal area of industry expertise.

As a side note, because both SI and S-SR each admit histories proscribed by the other, we can now state that *snapshot isolation is incomparable to strict serializability* (and by extension, to serializability); i.e., $SI \not\subseteq S\text{-}SR \wedge S\text{-}SR \not\subseteq SI$. Bailis *et al.* mischaracterise SI as being a proper superset of SR [13]. Adya does not compare SI with SR; instead, it is mislabelled as a proper superset of S-SR [2]. Although the models are incomparable from a set-theoretic perspective, in that one is not a subset of the other, we accept that from a pragmatic viewpoint, SR is a “stronger” model than SI because the latter proscribes histories that do not cause known anomalies yet permits histories that lead to serialization cycles.

The advantage of antidependency testing in STRIDE is that we can finally dispense with the archaic write conflict test—permitting benign blind writes while simultaneously rejecting lost updates.

Like most other schedulers, STRIDE is conservative, in that it rejects some serializable histories. Consider, for example, an acyclic history containing a backward-facing antidependency:

$$H_{back-ser} = w_0[x_0], c_0, r_1[x_0], r_2[x_0], w_1[x], w_2[y], c_1, c_2$$

The serialization graph of $H_{back-ser}$ is depicted in Fig. 9. It is serializable with the conflict equivalent serial order (T_0, T_2, T_1) . SI accepts this history; however, STRIDE rejects it due a backward-facing antidependency $T_2 \rightarrow_{rw} T_1$.

FIGURE 9. Serialization graph of $H_{back-ser}$.

K. THE ROLE OF AGENTS

The agent plays the least interesting part in the protocol for most update transactions and will not be traversed for most read-only transactions. Its role is to emulate a request-response style synchronous interaction with the certifier, which is otherwise entirely asynchronous. Database queries tend to be invoked synchronously, as are coordinators in atomic commitment protocols; likely, this style of interaction will also be preferred by the users of STRIDE. We suggest that in most systems the agent is part of a client library that is collocated with the cohort, rather than a separately deployed process.

One notable use case for a dedicated agent process is the certification of blind writes. A transaction performing a blind write may contain an empty readset; the cohort database serves no purpose therein. Therefore, we can allow a client to submit a write-only transaction directly to the agent, bypassing the cohort, assuming that the client is only concerned with the outcome of the transaction.

L. CONFORMANCE TO STRONGER ISOLATION PROPERTIES

Generally speaking, all known practical scheduler implementations are conservative in rejecting some histories that are otherwise admissible by their claimed isolation properties. This is especially apparent in pessimistic concurrency control systems, for example, S-S2PL, being the dominant pessimistic scheduler among commercial and open-source database systems. S-S2PL proscribes $H_{back-ser}$ by the shared lock of T_1 and T_2 on x that prevents the subsequent write of x by T_1 until T_2 commits. (Hence $c_2 \rightarrow c_1$.) Of the typical pessimistic schedulers, both 2PL and S2PL admit $H_{back-ser}$, but 2PL does not guarantee recoverability [17] whereas S2PL has the drawback of requiring prior knowledge of transactions' read patterns⁶.

Conservatism in a scheduler is invariably suboptimal unless it offers some stronger isolation property that is also genuinely useful. While the rejection of $H_{back-ser}$ does make STRIDE conservative, this approach carries distinct benefits. Namely, STRIDE conforms to both **commitment ordering** (CO) proposed by Raz [15] and **Φ -serializability** (Φ -SR) proposed by Koutanov [16]. We briefly recount these properties here.

For a history to be commitment ordered, the order of every pair of conflicting operations in every pair of committed transactions must match the order of their respective commit events [15]. Formally, a history is commitment ordered (CO; is in CO) if for all conflicting operations $p_i[x], q_j[x]$, for all committed transactions $T_i, T_j, p_i[x] \rightarrow q_j[x] \Rightarrow c_i \rightarrow c_j$. CO was devised to ensure global serializability in multidatabase systems [15].

Let $Start(T)$ denote the submission time of transaction T to a scheduler. A history H is Φ -serializable (Φ -SR; is in Φ -SR) if it is strict serializable and for every pair of conflicting transactions T_i, T_j , that are committed in H , $Start(T_i) < Start(T_j) \Leftrightarrow T_i \rightarrow_H T_j$ [16]. In other words, a Φ -SR scheduler ensures that conflicting transactions are committed in their logical timestamp order. Φ -SR was designed to avoid the **logical timestamp skew** (LTS) anomaly, wherein records streamed from a serializable primary database to a replica and serially installed on the latter in their logical timestamp order can compromise the consistency of the replica. In STRIDE, logical timestamp order is equivalent to commitment ordering.

Theorem 4. *Alg. 1 is at least as strong as both Φ -SR and CO for certified transactions. I.e., $Alg. 1 \subseteq \Phi\text{-SR} \cap CO$.*

Proof. Under abcast, Φ -SR and CO are equivalent, as commitment order follows logical timestamp order. It suffices to show that either $Alg. 1 \subseteq \Phi\text{-SR}$ or $Alg. 1 \subseteq CO$.

We assume there exists a pair of committed transactions, T_i, T_j , in some history H admissible by Alg. 1, that are serialized in the reverse order of their logical timestamps,

⁶ Once a read lock is released by a transaction, it cannot be reacquired, being the main reason that S2PL is not used in practice.

and show their absence by contradiction. Without loss of generality, assume $Start(T_i) < Start(T_j)$, hence $i < j$. Then, $T_j \rightarrow^? T_i$ for their serialization order to be reversed. Of the three dependency types, only read-write antidependencies may be projected backwards. This is a contradiction: backward-facing antidependencies are proscribed by Lemma 5. $\square$

M. LIMITATIONS

In its stateless incarnation, STRIDE is limited by the size of the suffix $\mathcal{S}$ in Alg. 1b. Increasing the suffix allows the certifier to look further “back in time,” so to speak, when certifying transactions. A transaction that is staged over a severely lagging cohort snapshot does not need to be unnecessarily aborted if no conflicting transactions emerged since its snapshot was taken. The larger the suffix, the further the certifier can look back in time, and the lower the likelihood of aborting nonconflicting transactions. Increasing the suffix, however, brings about several problems:

1. It takes up additional memory on the certifier.
2. It requires scanning through a larger set of records to identify outbound antidependencies in Alg. 1b.
3. It requires scanning through a larger set of records to identify inbound dependencies in Alg. 1c.
4. During recovery, the certifier has to scan from the head of the suffix to ensure that all pending transactions have been decided.

Points 2 and 3 can be addressed by employing either hashing or probabilistic data structures, such as counting Bloom filters, that support both the addition and removal of elements. For point 2, the use of such structures enables R3 of Alg. 1b to expediently determine, for some candidate transaction T_k , that there is no non-aborted transaction T_j whose writeset intersects with the T_k 's readset. If T_j 's absence can be ascertained, Alg. 1b can safely commit T_k by R4; otherwise, if the presence of T_j cannot be ruled out, R3 can initiate a full scan of $\mathcal{S}$ and either abort T_k or proceed to R4 if the scan comes up empty.

A similar technique may be employed for point 3 in Alg. 1c. We also have the option of truncating the quantified range of the set comprehension assigned to $\mathcal{C}$, making safepoint identification conservative. This restricts the range of versions on the cohorts to which T_k may be concurrently applied, implicitly preserving the safety property of the algorithm⁷.

For point 4, the certifier can periodically checkpoint its progress in the suffix to a database, to expedite recovery. This is the logical timestamp of the first undecided transaction in the suffix. If all transactions in the suffix have been decided, the checkpoint is the successor timestamp of the last decided transaction. Upon recovery (or if a new certifier joins the group), the certifier loads a suffix $\mathcal{S}$ that starts no later than the checkpointed timestamp; possibly earlier, depending on the preferred suffix length. (Recall, $\mathcal{S}$ must contain all undecided transactions and possibly some decided ones.) However, the

certifier only needs to start processing transactions from its checkpoint timestamp.

Lastly, point 1 cannot be easily addressed. Alg. 1 represents a fundamental trade-off between false aborts and resource utilisation on certifiers. Some prefix of $\mathcal{S}$ may be periodically relocated to stable storage to conserve memory; however, this will impact certification performance for transactions whose snapshot falls into this prefix. The following section builds on this basic premise.

V. ALGORITHM 2 — ANTECEDENT SET REIFICATION

The rejection rate of Alg. 1 is constrained by the chosen suffix size, as previously noted. We now consider an optimisation of Alg. 1 that introduces the notion of a semi-permanent state, allowing the certifier to look further back in time without affecting the suffix size. The design objective is to achieve in-memory certification of an overwhelming majority of transactions while allowing for off-process certification of a relatively small subset of transactions, rather than conservatively aborting the latter.

A. PRELIMINARIES

Assume a notional **antecedent set** of $\mathcal{S}$, $\mathcal{A}(\mathcal{S})$, being the set of synthetic transactions encompassing the writesets of all committed transactions in the prefix $\mathcal{P} = T_1..T_{i-1}$, where T_i is the first transaction in $\mathcal{S}$. Transactions in $\mathcal{A}$ are derived by taking transactions in $\mathcal{P}$ and applying the following rules, for each corresponding (A_g, T_g) pair. (For brevity, we refer to the antecedent set simply as $\mathcal{A}$ if the suffix $\mathcal{S}$ is discernible from the context.)

- A1. Eliminate all reads. I.e., $readset(A_g) \triangleq \emptyset$.
- A2. Eliminate writes that have been overwritten by a later transaction in $\mathcal{P}$. I.e., $writeset(A_g) \triangleq \{w \in writeset(T_g) : \nexists T_h \in \mathcal{P} : h > i \wedge w \in writeset(T_h)\}$.
- A3. Only keep A_g if it has a nonempty writeset. I.e., $A_g \in \mathcal{A} \Leftrightarrow writeset(A_g) \neq \emptyset$.

Simply put, $\mathcal{A}$ is the coalescing of all writes in $\mathcal{P}$ such that only the most recent writes are preserved and assigned the same version numbers that they had in $\mathcal{P}$. Serially applying transactions in $\mathcal{A}$ and in $\mathcal{P}$ produces two potentially different histories that are nonetheless state equivalent, which is to say, they produce the same end-result.

Consider again R2 of Alg. 1b. What compels R2 to abort T_k if $snapshot(T_k) < i - 1$? Suppose R2 was rewritten to the provisional variation, as per Alg. 2b-pr.

⁷ I.e., if Alg. 1 is safe to begin with, then safety is preserved if some transaction is assigned a higher safepoint than that permitted by Alg. 1c. In

the extreme case, $safepoint(T) = \infty$ for any T , which effectively disables Alg. 1d.

Algorithm 2b-pr Certification

```

1: let  $\mathcal{S} \triangleq$  (the suffix of transactions  $T_i..T_z$  from Alg. 1b)
2: upon receiving  $T_k \in \mathcal{S}$ :
3:   if
4:     ...
5:   else if
6:     /* R2: inspect the antecedent set for antidependencies */
7:      $\wedge \text{snapshot}(T_k) < i - 1$ 
8:      $\wedge \exists A_j \in \mathcal{A} : \wedge \text{snapshot}(T_k) < j$ 
9:        $\wedge j \notin \text{readvers}(T_k)$ 
10:       $\wedge \text{writeset}(A_j) \cap \text{readset}(T_k) \neq \emptyset$ 
11:     then
12:        $\text{abort}(T_k)$ 
13:   else
14:     /* R3 and R4 from Alg. 1b */
15:     ...
16:   end if

```

R2 in Alg. 2b-pr is derived by copying R3 with subsequent elimination of the obvious tautologies, such as $\neg \text{aborted}(T_j)$ and $j < k$. Clearly, $\neg \text{aborted}(T_j)$, because only committed transactions are admissible in $\mathcal{P}$ and, by extension, in $\mathcal{A}$. Then $j < k$; otherwise, T_k would have already been committed.

Compare Alg. 2b-pr to Alg. 1b and assume the algorithm only knows $\mathcal{S}$. Also, assume that $\text{readset}(T_k) \neq \emptyset$, hence $\neg R1$. $\mathcal{A}$ is obviously unknown at this point because the algorithm has no knowledge of $\mathcal{P}$. At this stage, we must conservatively assume that $\mathcal{A}$ contains a transaction that antidepends on T_k . I.e., $\exists x \in \text{readset}(T_k) : \exists A_j \in \mathcal{A} : \text{snapshot}(T_k) < j \wedge x \in \text{writeset}(A_j) \wedge j \notin \text{readvers}(T_k)$. Under this assumption, Alg. 2b-pr is equivalent to Alg. 1b; they behave identically for all inputs. In other words, the certification algorithm proscribes a backward-facing antidependency from T_k to any transaction $A_j \in \mathcal{A}$ on the assumption that such a transaction might exist.

So far, all we have done is delineate the intuition behind Alg. 1b by codifying it in Alg. 2b-pr using our notional antecedent set. As we know, this algorithm may lead to a false abort of T_k ; its likelihood can be approximated by some linear function of $|\mathcal{A}| / |\mathcal{A} \cup \mathcal{S}|$. Hence the pressure of increasing $\mathcal{S}$, especially in high-throughput certifiers. Ideally, we would like to significantly reduce the false abort rate by reifying $\mathcal{A}$ and thus making its contents available to the algorithm. We now remark on several useful properties of $\mathcal{A}$.

Compactness: The nice thing about the antecedent set is that it can be represented compactly. Let there be n distinct items processed by $\mathcal{P}$. Clearly, $|\mathcal{A}| \leq n$, because there can be at most one transaction for any written item in $\mathcal{P}$ by the combination of A2 and A3. Note, A1 merely drops the readset; it does not affect the cardinality of $\mathcal{A}$.

Transitivity: Another convenient property of $\mathcal{A}$ stems from the transitivity of the *observes* relation. If some transaction $A_i \in \mathcal{A}$ later has its write superseded by $T_j \in \mathcal{S}$, $j > i$, we can say that $A_i \rightarrow_{ww} T_j$. Furthermore, if $T_j \triangleright T_k$ and $wr(T_j)$,

T_k), then $A_i \triangleright T_k$. This property is essential: it allows us to safely apply A2 (and subsequently A3) without inadvertently discarding a critical antidependency. (This property is later used in the proof of Lemma 11.)

Commutativity: A final noteworthy property is that all transactions in $\mathcal{A}$ commute. By A1 and A2, there is no pair of transactions A_i, A_j , such that $(\text{readset}(A_i) \cup \text{writeset}(A_i)) \cap (\text{readset}(A_j) \cup \text{writeset}(A_j))$. This is convenient because we can arbitrarily reorder transactions in $\mathcal{A}$ after its formation.

The addition of a new transaction, A_n , by applying rules A1–A3 leads to the set $\mathcal{A}'$ that also satisfies all of the aforementioned properties.

The commutativity property implies that the ordering of transactions in $\mathcal{A}$ is unimportant, and rather than representing $\mathcal{A}$ as a sequence, it can instead be represented as a mapping, α , of data items to versions of committed transactions:

$$\alpha \triangleq [x \mapsto \text{let } \mathcal{M} \triangleq \{A \in \mathcal{A} : x \in \text{writeset}(A)\} \\ \text{in } \text{ver}(\varepsilon \in \mathcal{M} : \text{TRUE}) \text{ if } \mathcal{M} \neq \emptyset \text{ else } 0]$$

If the requested item does not exist among the transactions in $\mathcal{A}$, we assume that a notional placeholder for that item has been written by the initial transaction T_0 , which cannot antidepend on candidate transactions. (Hence α returns 0 if $\mathcal{M} = \emptyset$.) A cohort can exploit this property to insert a new item into the global state while satisfying the uniqueness constraint—by declaring the item in its readset. The first transaction that attempts to install a new item commits; the subsequent transactions will be aborted by the certifier.

B. EXPANDED SYSTEM MODEL

In the original system model, depicted in Section III, the certifier process only has access to the abcast primitive and the XDB for recording decision states. Before we introduce the revised algorithm, it is necessary to accommodate the antecedent set and the derived α mapping function in the system model; we revise the system model to encompass the following additional elements. The expanded system model is depicted in Fig. 10.

1) ANTECEDENT STORE

Abbrev. **α -store**. A soft-state database of item versions that is lazily populated by installing transaction updates from abcast in logical timestamp order. An α -store contains a version number, $\text{ver}(\alpha)$, which is the logical timestamp of the most recent transaction installed in that α -store. There is no requirement for the durability of writes; the contents of the store may be lost, provided $\text{ver}(\alpha)$ always accurately corresponds to the state of the store. An α -store is effectively a cache.

2) ANTECEDENT REIFIER

Consumes updates from some prefix in abcast, serially installing them in the associated α -store and updating $\text{ver}(\alpha)$ in the process. The reifier may be replicated for fault tolerance.

FIGURE 10. Expanded system model.

C. ALGORITHM SPECIFICATION

There is now a sufficient foundation for presenting the next algorithm. Alg. 2 is equivalent to Alg. 1 in all but one of its subordinate algorithms: Alg. 2b, for $R2$. (Rules $R1$, $R3$ and $R4$ of Alg. 2b are carried over unchanged from Alg. 1b.) The function $\alpha[x]$ produces the version of data item x , taken from some α -store, being the logical timestamp of the most recent transaction that updated x in $\mathcal{A}$.

Algorithm 2b Certification

```

1: let  $\mathcal{S} \triangleq$  (the suffix of transactions  $T_1..T_z$ ,
    $T_z$  being the last submitted transaction)
2: upon receiving  $T_k \in \mathcal{S}$ :
3:   if
4:     /* R1: commit a write-only transaction */
5:      $readset(T_k) = \emptyset$ 
6:     then
7:        $commit(T_k)$ 
8:   else if
9:     /* R2: abort  $T_k$  if there exists an earlier transaction
      in the antecedent set that antidepends on  $T_k$  */
10:     $\wedge snapshot(T_k) < i - 1$ 
11:     $\wedge \{v \in \{\alpha[x] : x \in readset(T_k)\} : v > snapshot(T_k)\} \setminus readvers(T_k) \neq \emptyset$ 
12:     $\exists T_j \in \mathcal{S} : \wedge snapshot(T_k) < j < k$ 
13:     $\wedge \neg aborted(T_j)$ 
14:     $\wedge j \notin readvers(T_k)$ 
15:     $\wedge writeset(T_j) \cap readset(T_k) \neq \emptyset$ 
16:    then
17:       $try\_abort(T_k)$ 
18:   else
19:     /* R3: abort  $T_k$  if there exists an earlier transaction  $T_j$  in the suffix
      that overlaps with  $T_k$  that has not aborted, is not among  $T_k$ 's
      read versions, and  $T_j$  antidepends on  $T_k$  */
20:      $\exists T_j \in \mathcal{S} : \wedge snapshot(T_k) < j < k$ 
21:      $\wedge \neg aborted(T_j)$ 
22:      $\wedge j \notin readvers(T_k)$ 
23:      $\wedge writeset(T_j) \cap readset(T_k) \neq \emptyset$ 
24:     then
25:        $try\_abort(T_k)$ 
26:   else
27:     /* R4: try to commit a transaction if it has made it this far */
28:      $try\_commit(T_k)$ 
29:   end if

```

Intuitively, Alg. 2b works because $R2$ and $R3$ collectively operate over the complete abcast sequence $T_1..T_z$, with transactions $T_1..T_{i-1}$ handled by $R2$ and $T_i..T_z$ handled by $R3$. $\mathcal{S}$ must include all undecided transactions, so for T_i to move from $\mathcal{S}$ to $\mathcal{A}$, T_i must be committed. If T_i antidepends on T_k , then its subsequent move to $\mathcal{A}$ will preserve this antidependency.

D. PRACTICAL CONSIDERATIONS

1) OVERLAPPING SUBSETS AND GAPS

Alg. 2b assumes that $ver(\alpha) \geq i - 1$; i.e., it allows for the contents of $\mathcal{A}$ to overlap with the suffix $\mathcal{S}$. In the case $ver(\alpha) = i - 1$, $\mathcal{A}$ and $\mathcal{S}$ are complementary—every transaction is captured in either one set or the other, but not in both. In the case $ver(\alpha) > i - 1$, $R3$ can be further optimised to certify over a truncated portion of $\mathcal{S}$ that starts from $ver(\alpha) + 1$. Finally, in the undesirable case that $ver(\alpha) < i - 1$ and $snapshot(T_k) < i - 1$, the certifier has the choice of either downgrading to Alg. 1b, that is, conservatively aborting T_k on the assumption that an antidependency exists in $\mathcal{A}$, or loading the missing transactions from abcast into memory, such that $ver(\alpha) \geq i - 1$.

Compacting the prefix of transactions into an antecedent cache alleviates the strain on the certifier in relation to the

suffix size while significantly reducing the number of false aborts due to stale cohort snapshots. Crucially, α is an optimisation; the complete or partial loss of the α -store does not impact the correctness properties of the algorithm, provided the data loss is detectable and that the α -store is always prefix-complete, be it a zero-length prefix in the extreme case. In other words, if the α -store loses some transaction T_n , then it must also forfeit all subsequent transactions in the corresponding prefix $\mathcal{P}$, that is, $\{T_s \in \mathcal{P} : s > n\}$, and $\text{ver}(\alpha)$ must report the immediate committed predecessor of T_n .

2) PERFORMANCE OF THE ANTECEDENT STORE

It is possible to operate the system with multiple independent certifiers, each reading from a dedicated α -store or a pool of α -stores. In this manner, the antecedent set can be load-balanced for read throughput.

An α -store instance is also write-intensive, as it must ingest all transaction updates. To maintain prefix completeness, the installation of updates must occur serially. However, there is no need for it to be synchronous with the commitment of transactions: the corresponding reifier process can apply updates asynchronously, at its discretion, akin to the installation of updates on the cohorts in Alg. 2e. The ability to install updates asynchronously accords another optimisation: Ordinarily, serial updates are slow because an update cannot commence until the previous update is completed, leaving the system in an underutilised state. So, rather than updating its α -store one transaction at a time, the reifier process can batch updates together, installing them with fewer round trips. This has virtually no bearing on the latency of certification, as most transactions will be certified directly from $\mathcal{S}$, which remains in memory.

E. PROOF SKETCH

For the forthcoming proof, we use the primed notation $\mathcal{G}'$ to depict the next state of $\mathcal{G}$. Similarly, P' depicts the truth value of some predicate P in its next state.

Lemma 10. *No decision made by Alg. 2b can be contradicted by a subsequent run of Alg. 2b, involving all transactions from the previous run.*

Proof. The proof is by induction over naturals. We consider a base case in which $i = 1$; i.e., $\mathcal{A} = \emptyset$ and the suffix $\mathcal{S}$ encompasses all transactions up to T_k (and possibly T_k 's successors). When $i = 1$, Alg. 2b and Alg. 1b are equivalent for all inputs because $R2$ is never triggered. (There is no T_k such that $\text{snapshot}(T_k) < i - 1$.) Therefore, Alg. 2 is correct in the base case, owing to the established correctness of Alg. 1. In the inductive step, $i = g$ and it suffices to show that the invariant holds for $i = g + 1$. It further suffices to show that (1) if T_k was aborted in some previous run, it will not be committed by $R1$ in a subsequent run, and (2) if T_k was committed in a previous run, it will not be aborted $R2$ in a subsequent run.

For (1), if T_k was aborted in some previous run, then $\text{readset}(T_k) \neq \emptyset$; therefore, T_k will never be committed by $R1$ in any run. This is analogous to the corresponding proof in Lemma 7.

For (2), if T_k was committed in some previous run, then $R1 \vee \neg R2 \wedge \neg R3$. For $R1$, if T_k was committed, then $\text{readset}(T_k) = \emptyset$ and it will be committed in every run. For $\neg R2 \wedge \neg R3$, it implies that neither $\mathcal{A}$ nor $\mathcal{S}$ contains a transaction T_j that antidepends on T_k and $j < k$. Therefore, T_i does not antidepend on T_k : either (2.1) $i \leq \text{snapshot}(T_k)$, or (2.2) $i \geq k$, or (2.3) $\text{aborted}(T_i)$, or (2.4) $i \in \text{readvers}(T_k)$, or (2.5) $\text{writeset}(T_i) \cap \text{readset}(T_k) = \emptyset$. We consider each case separately.

For (2.1), if T_i writes an item x that T_k reads, then $\alpha[x]' \leq \text{snapshot}(T_k)$; therefore, $\neg R2'$.

For (2.2), $i \geq k$ implies that $T_k \notin \mathcal{S}'$; hence, none of the rules will be evaluated for T_k in the next state.

For (2.3), $\text{aborted}(T_i)$ implies $\neg \text{committed}(T_i)'$ and $T_i \notin \mathcal{A}'$; hence, $\mathcal{A} = \mathcal{A}'$. Therefore, $\neg R2'$.

For (2.4), $i \in \text{readvers}(T_k)$ implies that $\forall x \in \text{writeset}(T_i) : \alpha[x]' = i$. This means $\forall x \in \text{readset}(T_k) : (x \in \text{writeset}(T_i) \Rightarrow \alpha[x]' = i) \wedge (x \notin \text{writeset}(T_i) \Rightarrow \alpha[x]' = \alpha[x])$. It follows that $\forall x \in \text{readset}(T_k) : \alpha[x]' = i \vee \alpha[x]' = \alpha[x]$. That is, for all x read by T_k , either $\alpha[x]'$ is assigned a version in $\text{readvers}(T_k)$ or remains unchanged. At the same time, $\{v \in \{\alpha[x] : x \in \text{readset}(T_k)\} : v > \text{snapshot}(T_k)\} \subseteq \text{readvers}(T_k)$ by $\neg R2$. Then in the next state, $\{v \in \{\alpha[x]' : x \in \text{readset}(T_k)\} : v > \text{snapshot}(T_k)\} \subseteq \text{readvers}(T_k)$; thus, $\neg R2'$.

For (2.5), $\text{writeset}(T_i) \cap \text{readset}(T_k) = \emptyset$ implies that $\nexists x \in \text{readset}(T_k) : \alpha[x]' \neq \alpha[x]$; thus, $\neg R2'$. $\square$

Lemma 11. *Alg. 2b proscribes a backward-facing antidependency between all pairs of committed transactions.*

Proof. The proof is by induction over naturals. We consider a base case in which $i = 1$; i.e., $\mathcal{A} = \emptyset$ and the suffix $\mathcal{S}$ encompasses all transactions up to T_k (and possibly T_k 's successors). When $i = 1$, Alg. 2b and Alg. 1b are equivalent for all inputs because $R2$ is never triggered. Therefore, Alg. 2 is correct in the base case, owing to the established correctness of Alg. 1. In the inductive step, $i = g$ and it suffices to show that the invariant holds for $i = g + 1$. We consider two separate cases: (1) a backward-facing antidependency is incident on a transaction in $\mathcal{A}$, and (2) a backward-facing antidependency is incident on a transaction in $\mathcal{S}$.

For (1), let A_c be the conflicting transaction that antidepends on T_k , and pick some $x \in \text{writeset}(A_c) \cap \text{readset}(T_k)$. Consider two cases: (1.1) where there exists a non-aborted T_j in $\mathcal{S}$ such that $x \in \text{writeset}(T_j) \wedge j < k$ and (1.2) there is no such T_j . In both cases, T_k is not committed by the assumption of the inductive step; it must be shown that T_k does not commit in the next state.

For (1.1), assume that $T_j = T_i$, and T_i is committed, otherwise $a[x]' = a[x]$ and $R2'$. $A_c \rightarrow_{ww} T_i$ because their writesets intersect and both transactions are committed.

Initially, we do not know whether $T_i \triangleright T_k$. However, the *G-update* phenomenon is proscribed by *Theorem 2*; therefore, if $A_c \rightarrow_{ww} T_i$ and $T_i \triangleright T_k$ and $wr(A_c, T_k)$, then $A_c \triangleright T_k$ by the transitivity of *ww-depends* and *observes* relations via the *wr* operator. However, since $\neg(A_c \triangleright T_k)$ by the assumption of (1.1), then $\neg(T_i \triangleright T_k)$. $\mathcal{A}' = \mathcal{A} \cup \{T_i\}$, thus $a[x]' = i$ and $R2'$.

For (1.2), $a[x]' = a[x]$ and $R2'$.

For (2), let T_j antidepend on T_k in $\mathcal{S}$. Consider two cases: (2.1) $T_i = T_j$ and (2.2) where $T_i \neq T_j$.

For (2.1), $T_i = T_j$, thus $\mathcal{A}' = \mathcal{A} \cup \{T_i\}$ and $R2'$.

For (2.2), $T_i \neq T_j$, thus $T_j \in \mathcal{S}'$ and $R3'$. $\square$

Theorem 5. *Alg. 2 is correct.*

Proof. Except for *Lemma 5* and *Lemma 7*, the previous lemmas are unaffected by changes to $R2$. The safety properties $S2$, $S4$ and $S5$ are also trivially preserved; we omit their proofs for brevity. To prove $S1$ and $S3$ it suffices to show that (1) no decision made by Alg. 2b can be contradicted by a subsequent run of Alg. 2b, involving all transactions from the previous run, and (2) Alg. 2b proscribes a backward-facing antidependency between all pairs of committed transactions. Obligations (1) and (2) are discharged by *Lemma 10* and *Lemma 11*, respectively. $\square$

F. CONSIDERATIONS ON GENERALITY

Alg. 2 significantly reduces the false abort rate for short suffixes but does not eliminate it. Interestingly, neither does a configuration in which $\mathcal{S}$ encompasses the complete set of transactions. The reason behind the false aborts lies in the $\neg aborted(T_j)$ predicate in $R3$. If there is some non-aborted T_j in $\mathcal{S}$, such that $j < k$ and T_j antidepends on T_k , then the transaction scheduler must adhere to one of three strategies: either 1) wait until T_j is decided, 2) attempt to decide T_j first, or 3) abort T_k on the worst-case assumption that T_j will be committed. As it stands, the algorithm applies strategy 3.

For (1), it would not be difficult to adapt $R3$ of Alg. 2b to halt T_k 's evaluation until T_j is decided; however, this introduces blocking in the scheduler⁸ and runs contrary to the spirit of optimistic concurrency control. However, even in this case, it cannot induce a deadlock, as blocking only applies to backward-facing antidependencies, and T_j clearly cannot directly block on T_k since $j < k$. Nor can T_j block on T_k via some transitive dependency, because for a deadlock cycle to

materialise, the antidependency direction must be reversed at some point, which would immediately render it nonblocking.

For (2), adapting $R3$ to recursively decide T_j before T_k has the advantage of forming a perfectly accurate certifier that never falsely aborts. It also means that transactions' decision states can be deterministically reconstructed from abcast, relieving the need for durable writes on the XDB. On the flip side, if transactions are certified concurrently, then in the absence of process coordination, multiple attempts at certifying a transaction may take place concurrently. This will not compromise safety, as per *Lemma 7*; however, it may result in duplicate work. Also, checking transaction states requires a round-trip to the database; alternatively, transaction states can be cached in memory. This strategy is roughly equivalent to (1), but instead of waiting for some other certifier to commit or abort the antidependency, the present certifier does so itself. Still, it may be more effective than conservatively aborting T_k , and is worth considering. Other serializable certifiers, for example, *SLOG* [36] and *Tashkent* [37] also proactively decide antidependency states by certifying transactions in the order of arrival.

The algorithms in this paper are described in terms of (3) primarily because doing so generalises their specifications. That is, an algorithm that does not mandate decisions on antidependency states contains more behaviours than an alternative based on (1) or (2) for the same model size. Namely, for antidependency scenarios, the behaviour of (1) and (2) always results in either a commit or abort state, whereas (3) permits both behaviours while ensuring decision stability ($S1$). Although (2) has the hallmarks of a compelling optimisation, we rest the final decision with the implementer.

VI. RELATED WORK

This section recounts related work in the field of distributed transaction processing and concurrency control, relating the results of this paper to other contributions that preceded it.

A. SGSI

The prior work that most closely resembles STRIDE is *Serializable Generalized Snapshot Isolation* (SGSI) by Bornea *et al.* [11]. The two algorithms share much in common, including—

- Many elements of the overall system model;
- Lazily replicated databases⁹ with weakened local consistency properties;
- Support for uncertified read-only transactions¹⁰;
- Commitment ordering and Φ -serializability¹¹ at the system level; and
- Certification over a totally ordered sequence of candidate transactions.

⁸ Technically, it may no longer be called a certifier if it queues or otherwise delays operations.

⁹ SGSI refers to them as replicas; STRIDE refers to them as cohorts to highlight their autonomy.

¹⁰ SGSI supports serializability for read-only transactions, while STRIDE offers the choice of serializability or update serializability.

¹¹ SGSI histories are in Φ -SR, although it was devised prior to the work on Φ -serializability, and externalizability in general.

Roughly speaking, SGSI works by staging the updates of in-flight transactions in an in-memory, disk-backed SQL database attached to the *certifier* process, called *CertDB*. For each candidate transaction T_k , the certifier executes a *certification query* against *CertDB* to determine whether any of the predicates used by the transaction have been read-impacted or write-impacted by the set of committed transactions that overlap with T_k ; i.e., $\exists T_j : \text{snapshot}(T_k) < j < k \wedge \text{writeset}(T_j) \cap (\text{readset}(T_k) \cup \text{writeset}(T_k)) \neq \emptyset$. The result of the certification query is called the *conflict set*. If the conflict set is nonempty, T_k is aborted; otherwise, T_k is committed, and all replicas serially install T_k 's writeset according to its commitment order.

The differences between STRIDE and SGSI are primarily in the following:

1. SGSI requires generalised snapshot isolation on the replicas, while in STRIDE the equivalent requirement is RA (MAV $\cap$ I-CI) in the stripped-down variant (Alg. 0) and PL-3U in the stateless and antecedent-reified variants (Alg. 1 and Alg. 2, respectively). For Alg. 0, RA $\subset$ GSI; thus, bare-bones STRIDE materially relaxes its expectations of replicas while maintaining strict serializability at the system level. For Alg. 1 and 2, PL-3U is still more permissive than GSI¹² as it allows nonconflicting updates to be applied out of order.
2. SGSI requires prefix consistency, in turn, requiring the serial installation of writesets on the replicas. As such, SGSI does not maximise resource utilisation on replicas for writes. (A write cannot begin until its predecessor completes.) This is a major performance bottleneck of SGSI, resulting in replicas that can significantly lag behind the global state at high throughput levels. The likelihood of aborts increases with replica lag, meaning that transaction latencies are expected to increase before replicas reach the point of saturation. The redeeming benefit: serializability (PL-3) for read-only transactions. STRIDE, on the other hand, supports the concurrent installation of nonconflicting updates on the replicas while maintaining update serializability (PL-3U) for read-only transactions in the worst case. In effect, STRIDE achieves a similar practical level of consistency on the replicas, without constraining replication throughput. Furthermore, STRIDE can support serializable read-only transactions without certification by dedicating a subset of replicas to apply updates serially. Thus, read-only transactions are accorded the flexibility of either executing at concurrently updated cohorts for PL-3U, or at serially updated cohorts for full PL-3.
3. SGSI relies on a certification database, which is both read and write-intensive. This is the second major
4. Being a descendant of snapshot isolation, SGSI rejects concurrent write conflicts on the assumption that they might conceal a lost update anomaly. In doing so, SGSI also rejects serializable blind writes. STRIDE permits serializable blind writes, using antidependency testing to identify lost updates. A STRIDE transaction must therefore explicitly declare its read dependencies; there is no presumption that if a transaction writes to an item, then it must have also read from it.
5. SGSI mandates the use of identical schemas among replicas and *CertDB*; i.e., SGSI is intrinsically schema-aware. (Without this, transactions cannot be certified using SQL queries on *CertDB*.) STRIDE is schema-agnostic; neither the certifier nor the antecedent store cares about the internal structure of the data items; moreover, the cohort databases can be structured differently from each other. The only uniformity required by STRIDE is the identity of the data items; i.e., for every data item x , the mapping of x to the underlying entity is equivalent on every cohort. The antecedent store does not consider the entity—only its version number. Furthermore, there is no requirement that each cohort maintains a full set of data items processed during the lifetime of the system; it may safely limit its view to the subset of data items that it cares about. (Although in doing so, it limits the scope of queries it can support.)
6. SGSI supports SQL queries with predicates. STRIDE only supports named data items; predicate support is absent from STRIDE and is unlikely to be offered without significant changes to the algorithm.

If we limit our consideration to the attributes of the two algorithms, disregarding their usage context, STRIDE is an objective improvement over SGSI in all areas except for point 6—predicate support. The bottlenecks described in points 2 and 3 call the performance of SGSI into question: on one hand, SGSI attempts to scale SQL databases by separating

¹² If we disregard the conservative write conflict avoidance of GSI, being appropriate in this case because updates are installed serially and are thus conflict-free.

certification from reads, using replication for read throughput and availability; on the other hand, the throughput of the certifier is limited by that of CertDB and writes are applied serially on the replicas and synchronously on CertDB. Bornea *et al.* did not discuss these issues in [11]. Instead, the authors demonstrated a combined throughput of 88,000 TPM in the TPC-W benchmark under some workloads, showing a 7x improvement over single-replica performance (admittedly, in 2011). At first glance, SGSI appears to live up to its promise; upon closer inspection, however, the result is unsurprising for its time, given that replicas were equipped with 7,200 RPM mechanical disks with 2 GB of RAM, whereas the certifier used an in-memory SQLite.NET database for conflict detection, buffering updates and committing to disk in batches of 32 certifications [11]. Under these conditions, the bottleneck is presumably in the IO subsystem of the replicas, masking the bottlenecks of the overarching algorithm; the buffered certifier will outperform a conventional DBMS with little effort. We speculate that using solid-state disks on the replicas, with sufficient memory to support table pinning, will noticeably shift the balance of performance and make SGSI's bottlenecks more apparent; SGSI will still scale to some degree, but the saturation point will occur at a lower multiple.

Returning to point 6, the lack of predicate certification in STRIDE is a significant functional reduction over SGSI. The flip side, however, is that STRIDE is schema-agnostic in its entirety, permitting site autonomy. It is difficult to imagine a schema-agnostic predicate certifier; thus, the two design criteria are inherently contradictory. We suggest that the contrast in points 5 and 6 naturally leads to a distinct classification of the two algorithms: SGSI is a *distributed SQL database*, whereas STRIDE is an *atomic commitment protocol* over a heterogeneous *multidatabase* system. To that point, SGSI is more likely to be used as a common database for one or more closely coupled applications. Conversely, STRIDE is better suited to atomically coordinating distributed transactions across a set of federated, autonomous applications (or systems) with separate databases.

B. Calvin

Calvin [38] is a transactional scheduling and replication layer that uses the deterministic ordering guarantee to achieve strict serializability without relying on a multi-phase atomic commitment protocol for coordination.

Calvin works by assigning a total order to candidate transactions using an abcast primitive, which also doubles as a persistence layer. Transactions encompass data items residing in multiple partitions and replicas. A scheduler at each partitioned replica serially computes a locking plan for the inbound transactions, which are subsequently executed concurrently according to the established plan. Owing to the combination of total order and the upfront declaration of

readsets and writesets, the locking schedules agree at each replica. Partitions may synchronously exchange information to facilitate multi-partition transactions, wherein a transaction executing on one partition reads from or writes to an item that resides at another partition.

Calvin and STRIDE share many design elements and resulting properties:

- Avoiding multi-phase synchronous coordination by way of deterministic scheduling. Transactions behave identically at each replica; therefore, once their relative order is stably agreed upon, the coordination step becomes unnecessary.
- Use of abcast to fulfil both ordering and durability. Whereas convention databases rely on a durable REDO log, Calvin and STRIDE exploit the durability of the abcast primitive.
- Disclosing the transactions' readsets and writesets prior to scheduling. A transaction may not prospect for new items during execution.
- Asynchronous replication of transactions.
- Concurrent installation of updates. In Calvin's case, this is accomplished by concurrent execution. In STRIDE, certified updates may be installed concurrently on the replicas.
- Support for serializable blind writes.
- Commitment ordering and Φ -serializability¹³ at the system level.

They differ by the following characteristics:

1. Transactions are modelled as stored procedures on Calvin, leading to code sharing across sites. The sites are therefore not autonomous. STRIDE embeds the state mapping in the transaction record, supporting site autonomy. On the flip side, the use of stored procedures may result in a more compact transaction representation because the state mapping is derived locally.
2. Calvin employs locking at each site, meaning it may queue certain transactions behind others, but the scheduler never aborts (unless the transaction aborts of its own will). The commitment model is reversed in STRIDE: a transaction may voluntarily abort during preliminary validation; however, it has no influence over its outcome during or after certification.
3. Calvin combines the scheduling and execution of transactions. In STRIDE, transactions are scheduled on the certifier independently of their subsequent execution on the replicas.
4. While both algorithms are concurrent in their processing of transactions, their concurrency exploits differ markedly. Calvin executes transactions concurrently, subject to a predetermined locking plan. There is no splitting of transactions' read and write

¹³ Histories produced by Calvin are in Φ -SR, although it was devised prior to the work on Φ -serializability and externalizability.

phases: they execute collectively at the relevant partitions. As a result, slow reads and writes block subsequent transactions. In STRIDE, read, certification and write phases support concurrent execution and are subject to pipelining; in other words, transactions may be reading concurrently from local snapshots, while candidates are being certified, and committed writes are being installed concurrently for overlapping items. Slow updates have no bearing on certification performance.

5. While both algorithms use asynchronous replication, they differ vastly. Calvin replicates just the candidate transactions; the decision is implicit in the execution phase. The initiator of a Calvin transaction does not know of its outcome until it is executed on the relevant active partitions. By comparison, STRIDE replicates both candidates and decisions; the initiator of a transaction learns of its outcome after certification.
6. Read-only transactions are treated identically to update transactions in Calvin—they require scheduling and entail locking. Calvin cannot serve lock-free, read-only queries unless it is bonded with a multiversion storage engine [38]. By comparison, STRIDE supports consistent uncertified reads at the cohorts.
7. Following from the above, STRIDE offers multiple consistency models for read-only transactions: USR, SR and S-SR, depending on the readers' requirements. This flexibility is absent from Calvin for as long as multiple partitions are used. Reducing Calvin to a single partition supports fast reads at the expense of write throughput.
8. Calvin partitions its dataset mainly for update performance, trading read performance. STRIDE does not employ partitioning; instead, replicating data to support site autonomy and local reads. Calvin's partitioned execution model is susceptible to *progress skew*—a condition wherein execution runs out of phase across partitions, stalling the progress of multi-partition transactions [38]. STRIDE avoids this by design.

Finally, we note that in the same vein as SGSI, Calvin's design is optimised for use in a distributed database, where the use of stored procedures is commonplace and site autonomy is a non-requirement. Calvin has recently been adopted in Fauna—a commercial, SQL-compatible distributed database—with minor variations [39].

VII. SUMMARY AND CONCLUSIONS

This paper presents a novel protocol for optimistically certifying distributed transactions in multidatabase systems. In STRIDE, sites called *cohorts* maintain local replica databases containing data items that form a subset of the notional *global state*, at versions that may lag behind their most recent committed state. Transactions are initiated by cohorts after

querying their local state to ascertain provisional validity, before forwarding the *candidate* transactions on to a *certifier* via an *agent* proxy for categorical validation and atomic commitment. (The agent may be collocated with the cohort.) The agent assigns a *logical timestamp order* to the candidate transactions by sending them over an *atomic broadcast* primitive, being the order in which the transactions are eventually certified. The cohort databases are asynchronously populated by installing the writesets of certified transactions; a cohort eventually catches up to the global state for the subset of data items that it cares about.

The installation of writesets on the cohort may proceed concurrently, and out of order with respect to the transactions' commitment order; nonetheless, the algorithm guarantees that no transaction is installed before any of its direct or transient dependencies, and that all writesets are installed atomically. In doing so, the cohorts conform to *update serializability* (USR, PL-3U) for read-only transactions, relieving the need for their certification. Additionally, the isolation property for uncertified read-only transactions can be elevated to full serializability (SR, PL-3) on those cohorts that apply updates serially, in strict commitment order.

For transactions that complete certification, the resulting isolation property is the intersection of *commitment ordering* and Φ -*serializability*; the latter essential to avoiding the *logical timestamp skew* anomaly that was recently shown to plague event-replicated systems [16].

The overarching design of STRIDE bears many similarities to SGSI and was designed from the outset with the limitations of SGSI in mind, ensuring that neither the certifier nor the cohorts are bottlenecked for nonconflicting transactions, and offering a choice of consistency models for uncertified read-only transactions. In addition to concurrent, out-of-order updates and support for serializable blind writes, STRIDE gains schema autonomy, but forfeits predicate support in doing so. Evidently, the algorithms are different in their practical application: SGSI is a single distributed database system, while STRIDE is a transactional multidatabase system.

By paring STRIDE down to its minimally correct form (viz. Alg. 0), it was shown that the isolation levels on the replicas need not be stronger than *read atomic* (RA), that is, the intersection of *monotonic atomic view* (MAV) and *item cut isolation* (I-CI), to produce strict serializable post-certification histories. This result is an improvement over the previous result of [11] that alluded to a minimum of *generalised snapshot isolation* (GSI). In its more conventional form (Alg. 1 and 2), STRIDE delivers USR on the replicas, which is still weaker than GSI (if we disregard write conflicts, which cannot occur during serial installation). We can now categorically state that local GSI is not a requirement for global SR.

We also remarked on the inherent design trade-offs in STRIDE. In particular, the *stateless certification* variant (Alg. 1) is limited by the extent of the in-memory suffix of candidate transactions: increasing the suffix results in a lower false abort

rate for lapsed nonconflicting transactions but increases memory utilisation on the certifier. We presented an optimised variant (Alg. 2) that coalesces the prefix of committed transactions into an antecedent store, which is subsequently queried by the certifier when assessing a lapsed transaction. With *antecedent set reification*, certifiers can operate over a much smaller in-memory suffix, acquiring immunity to most false aborts while assessing the overwhelming majority of transactions entirely in memory.

REFERENCES

- [1] A. Fekete, D. Liarokapis, E. O’Neil, P. O’Neil, and D. Shasha, “Making snapshot isolation serializable,” *ACM Transactions on Database Systems*, vol. 30:2, 2005, pp. 492–528.
- [2] A. Adya, “Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions,” PhD dissertation, Massachusetts Institute of Technology, 1999.
- [3] A. Adya, B. Liskov, and P. O’Neil, “Generalized isolation level definitions,” *Proc. of the 16th International Conference on Data Engineering*, Feb. 2000, pp. 67–80.
- [4] R.C. Hansdah and L.M. Patnaik, “Update Serializability in Locking,” *ICDT ’86: Proceedings of the International Conference on Database Theory*, Sep. 1986, pp. 171–185.
- [5] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J.M. Hellerstein, and I. Stoica, “Highly Available Transactions: Virtues and Limitations,” *Proceedings of the VLDB Endowment*, vol. 7, issue 3, Nov. 2013, pp. 181–192.
- [6] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O’Neil, and P. O’Neil, “A critique of ANSI SQL isolation levels,” *Proc. of the ACM SIGMOD International Conference on Management of Data*, ACM, New York, Jun. 1995, pp. 1–10.
- [7] P. Bailis, A. Fekete, A. Ghodsi, J.M. Hellerstein, and I. Stoica, “Scalable Atomic Visibility with RAMP Transactions,” *ACM Transactions on Database Systems*, vol. 41:3, article 15, Jul. 2016.
- [8] D. Shasha, F. Llirbat, E. Simon, and P. Valduriez, “Transaction chopping: Algorithms and performance studies,” *ACM Trans. Datab. Syst.*, vol. 20:3, Sep. 1995, pp. 325–363.
- [9] Y. Sovran, R. Power, M.K. Aguilera, and J. Li, “Transactional storage for geo-replicated systems,” *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, Oct. 2011, pp. 385–400.
- [10] S. Revilak, P. O’Neil and E. O’Neil, “Precisely serializable snapshot isolation (PSSI),” *Proc. of the 27th Int. Conf. on Data Eng.*, Apr. 2011, pp. 482–493.
- [11] M.A. Bornea, O. Hodson, S. Elnikety, and A. Fekete, “One-copy serializability with snapshot isolation under the hood,” *IEEE 27th International Conference on Data Engineering*, DOI: 10.1109/ICDE.2011.5767897, 2011, pp. 625–636.
- [12] S. Elnikety, F. Pedone, and W. Zwaenepoel, “Generalized Snapshot Isolation and a Prefix-Consistent Implementation,” *Proc. of the 24th IEEE Symposium on Reliable Distributed Systems*, 2004.
- [13] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J.M. Hellerstein and I. Stoica, “Highly Available Transactions: Virtues and Limitations,” *Proceedings of the VLDB Endowment*, vol. 7, issue 3, Nov. 2013, pp. 181–192.
- [14] M. Herlihy and J.M. Wing, “Linearizability: A Correctness Condition for Concurrent Objects,” *ACM Transactions on Programming Languages and Systems*, vol. 12(3), Jul. 1990, pp. 463–492.
- [15] Y. Raz, “The Principle of Commitment Ordering, or Guaranteeing Serializability in a Heterogeneous Environment of Multiple Autonomous Resource Managers Using Atomic Commitment,” *VLDB ’92: Proc. of the 18th International Conference on Very Large Databases*, Aug. 1992, pp. 292–312.
- [16] E. Koutanov, “The Logical Timestamp Skew Anomaly in Event-Replicated Transaction Schedulers,” in *IEEE Access*, vol. 9, Aug. 2021, doi: 10.1109/ACCESS.2021.3107506, pp. 123375–123397.
- [17] P.A. Bernstein, V. Hadzilacos, and N. Goodman, “Concurrency Control and Recovery in Database Systems,” *Addison-Wesley*, 1987.
- [18] B. Lampson and H. Sturgis, “Crash recovery in a distributed data storage system. Technical report,” *Computer Science Laboratory, Xerox Palo Alto Research Centre*, CA, 1976.
- [19] J.N. Gray, “Notes on database operating systems,” *Operating Systems – An Advanced Course*, M.J. Flynn (eds.), LNCS, vol. 60, Springer, London, 1978, pp. 393–481.
- [20] N. Nouali, H. Drias, and A. Doucet, “A mobility-aware two-phase commit protocol,” *Int. Arab J. Inf. Technol.*, 3(1):2006.
- [21] J.N. Gray and A. Reuter, “Transaction Processing: Concepts and Techniques,” Morgan Kaufman, USA, 1993.
- [22] L. Liu, D. Agrawal, and A. El Abbadi, “The Performance of Two-Phase Commit Protocols in the Presence of Site Failures,” *Technical Report TRCS94-09*, Santa Barbara, Department of Computer Science, University of California, Apr. 1994.
- [23] H. Garcia-Molina and K. Salem, “Sagas,” *Proceedings of the 1987 ACM SIGMOD Int. Conf. on Management of Data*, Dec. 1987, pp. 249–259.
- [24] D. Skeen, “A quorum-based commit protocol,” *Cornell Univ.*, New York, NY, USA, Tech. Rep. 82-483, 1982.
- [25] I. Keidar and D. Dolev, “Increasing the Resilience of Distributed and Replicated Database Systems,” *Journal of Computer and System Sciences*, vol. 57, issue 3, Dec. 1998, pp. 309–324.
- [26] A. Cerone, G. Bernardi, and A. Gotsman, “A Framework for Transactional Consistency Models with Atomic Visibility,” *Proc. of the Concur Conference*, Madrid, Spain, Sep. 2015.
- [27] A. Fekete, E. O’Neil, and P. O’Neil, “A read-only transaction anomaly under snapshot isolation,” *ACM SIGMOD Record*, vol. 33:3, 2004, pp. 12–14.
- [28] M.J. Cahill, “Serializable Isolation for Snapshot Databases,” *ACM Trans. on Database Systems*, vol. 34:4, Dec. 2009, pp. 1–42.
- [29] D.R.K. Ports and K. Grittnr, “Serializable Snapshot Isolation in PostgreSQL,” *Proc. of the VLDB Endowment*, vol. 5:12, Aug. 2012, pp. 1850–1861.
- [30] C.K. Rudrabhatla, “Comparison of Event Choreography and Orchestration Techniques in Microservice Architecture,” *International Journal of Advanced Computer Science and Applications*, vol. 9, 2018.
- [31] A. Messina, R. Rizzo, P. Stormiolo, M. Tripiciano, and A. Urso, “The Database-is-the-Service Pattern for Microservice Architectures,” *Intl. Conf. on Inf. Tech. in Bio- and Medical Informatics*, Sep. 2016, pp. 223–233.
- [32] FoundationDB Features, “FoundationDB Atomic Operations,” [Online]. Available: <https://apple.github.io/foundationdb/features.html#atomic-operations>, accessed 27 Jun. 2021.
- [33] E. Weaver, “A Comparison of Scalable Database Isolation Levels,” *Fauna website*. [Online]. Available: <https://fauna.com/blog/a-comparison-of-scalable-database-isolation-levels>, accessed 27 Jun. 2021.
- [34] L. Lamport, “The part-time parliament,” *ACM Trans. Comput. Syst.*, vol. 16, no. 2, 1998, pp. 133–169.
- [35] E. Koutanov, “Spire: A Cooperative, Phase-Symmetric Solution to Distributed Consensus,” in *IEEE Access*, vol. 9, Jul. 2021 DOI: 10.1109/ACCESS.2021.3096326, pp. 101702–101717.
- [36] K. Ren, D. Li, and D.J. Abadi, “SLOG: serializable, low-latency, geo-replicated transactions,” *Proc. VLDB Endow.*, vol. 12, issue 11, Jul. 2019, DOI: 10.14778/3342263.3342647, pp. 1747–1761.

- [37] S. Elnikety, S. Dropsho, and F. Pedone, "Tashkent: Uniting Durability with Transaction Ordering for High-Performance Scalable Database Replication," *SIGOPS Oper. Syst. Rev.*, vol. 40, issue 4, DOI: 10.1145/1218063.1217947, Oct. 2006, pp. 117–130.
- [38] A. Thomson, T. Diamond, S.C. Weng, K. Ren, P. Shao, and D.J. Abadi, "Calvin: Fast distributed transactions for partitioned database systems," *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, May 2012, pp. 1–12. DOI: 10.1145/2213836.2213838.
- [39] D.J. Abadi and M. Freels, "Consistency without Clocks: The Fauna Distributed Transaction Protocol," Fauna Engineering Features, Oct. 2018. [Online]. Available: <https://fauna.com/blog/consistency-without-clocks-faunadb-transaction-protocol>

EMIL KOUTANOV received his B.E. degree *summa cum laude* from the University of Technology Sydney (Australia) in 2007. He is a consulting software architect and part-time researcher.

His professional career has spanned several diverse industries including wagering, telecommunications, construction and finance, focusing on the design and implementation of reliable and scalable distributed systems. He has authored several articles on open-source middleware, application design and architecture, as well as a reference book on the subject of event streaming. His research interests include the analysis and design of distributed algorithms and models of computation, fault-tolerance and continuous availability, formal methods, concurrent and parallel computing, consistency models and consensus.