SNOW Revisited: Understanding When Ideal READ Transactions Are Possible

Kishori M. Konwar
RLE, MIT
Cambridge, MA

Wyatt Lloyd
Computer Science
Princeton University
Princeton, NJ, USA

Haonan Lu
Computer Science
Princeton University
Princeton, NJ, USA

Nancy Lynch
CSAIL, MIT
Cambridge, MA, USA

***Abstract*—READ transactions that read data distributed across servers dominate the workloads of real-world distributed storage systems. The SNOW Theorem [13] stated that ideal READ transactions that have optimal latency and the strongest guarantees—i.e., "SNOW" READ transactions—are impossible in one specific setting that requires three or more clients: at least two readers and one writer. However, it left many open questions.**

We close all of these open questions with new impossibility results and new algorithms. First, we prove rigorously the result from [13] saying that it is impossible to have a READ transactions system that satisfies SNOW properties with three or more clients. The insight we gained from this proof led to teasing out the implicit assumptions that are required to state the results and also, resolving the open question regarding the possibility of SNOW with two clients. We show that it is possible to design an algorithm, where SNOW is possible in a multi-writer, single-reader (MWSR) setting when a client can send messages to other clients; on the other hand, we prove it is impossible to implement SNOW in a multi-writer, single-reader (MWSR) setting–which is more general than the two-client setting–when client-to-client communication is disallowed. We also correct the previous claim in [13] that incorrectly identified one existing system, Eiger [12], as supporting the strongest guarantees (SW) and whose read-only transactions had bounded latency. Thus, there were no previous algorithms that provided the strongest guarantees and had bounded latency. Finally, we introduce the first two algorithms to provide the strongest guarantees with bounded latency.

***Keywords*-distributed transactions; strict-serializability;**

I. Introduction

Today's web services are built on *distributed storage systems* that provide fault tolerant and scalable access to data. Distributed storage systems scale their capacity and throughput by *sharding* (i.e., partitioning) data across many machines within a datacenter, i.e., each machine stores a subset of the data. They also *geo-replicate* the data across several geographically dispersed datacenters to tolerate failures and to increase their proximity to users.

Distributed storage systems abstract away the complexities of sharding and replication from application code by providing guarantees for accesses to data. These *guarantees* include consistency and transactions. *Consistency* controls the values of data that accesses may observe and *transactions* dictate what accesses may be grouped together. Stronger guarantees provide an abstraction closer to a single-threaded environment, greatly simplifying application code. Ensuring the guarantees hold, however, often comes with worse performance. Therefore, the tradeoff between performance and guarantees lies at the heart of designing such systems.

The performance-guarantee tradeoffs that result from replication have been well-studied with several well-known impossibility results [1], [6], [7], [11]. For instance, the CAP Theorem [7] proves that system designers must choose either availability during network partitions (performance) or strong consistency across replicas (guarantee). However, little prior work exists on what performance-guarantee tradeoffs result from sharding.

Understanding the performance-guarantee tradeoff due to sharding is important because user requests are typically handled across many shards but within a single nearby datacenter (replica). This is particularly true for the reads needed to handle a user request, which are what dominate real-world workloads: Facebook reported 500 reads for every write in their TAO system [3] and Google reported three orders of magnitude more reads than general transactions for their F1 database that runs on their Spanner system [5]. In this work, we focus on clarifying the performance-guarantee tradeoff for reads that results from sharding. Distributed storage systems group read requests (that each individually accesses a separate shard) into *READ transactions* that together return a consistent, cross-shard view of the system. Whether a view is consistent is determined by the consistency model a system provides. The ideal READ transactions would have the strongest guarantees: They would provide strict serializability [15], the strongest consistency model, and they could be used in a system that also includes *WRITE transactions* that group write requests (each to a separate shard) together. The alternative to the latter property are READ transactions that can only be used in systems that have non-transactional, *simple writes*.

The ideal READ transactions would also provide the best performance. In particular, they would provide the lowest possible latency because the prevalence of reads makes them dominate the user response times that are aggressively optimized by web services [4], [10], [16]. The *optimal latency* for a READ transaction is to match the latency of non-transactional, *simple reads*: complete in a single round trip of non-blocking parallel requests to the shards that return only the requested data [13].

1530-2075/21/$31.00 ©2021 IEEE
DOI 10.1109/IPDPS49936.2021.00101

A. Previous Results and Open Questions

The SNOW Theorem was the first result in the sharding dimension that is relevant to READ transactions [13]. The SNOW Theorem is an impossibility result that proves no READ transaction can provide **S**trict serializability with **N**on-blocking client-server communication that completes with **O**ne response, with only one version of the data, per read in a system with concurrent **W**RITE transactions (§II-A). It shows there is a fundamental tradeoff between the latency and guarantees of READ transactions that system designers must grapple with, they must pick either the strongest guarantees (S and W) or optimal latency (N and O).

SNOW is trivially possible in systems with a single client or a single server because the single entity naturally serializes all transactions. The SNOW Theorem shows SNOW is impossible in systems with at least three clients and at least two servers. It explicitly leaves open the question of the possibility of SNOW in a system with two clients. In addition, the model used in the prior work implicitly leaves open several questions. It assumed the three clients were a single writer and multiple readers (SWMR). This leaves open the possibility of SNOW with multiple writers and a single reader (MWSR). The SNOW Theorem also implicitly assumed that clients do not directly exchange messages and that write operations in such a system must eventually complete. This also leaves open the question of whether allowing or disallowing client-to-client (C2C) communication has any impact on the feasibility of READ transactions with SNOW properties.

In this work, the new impossibility results are philosophically similar to other impossibility results—such as FLP [6] and CAP [2], [7]—in that they help system designers avoid wasting effort in trying to achieve the impossible. That is, the SNOW Theorem identifies a boundary in the design space of READ transactions, beyond which no algorithms can possibly exist. By revisiting SNOW, our work makes this boundary more precise.

B. Our Contributions

Our work builds on the SNOW Theorem to clarify this fundamental tradeoff by providing a thorough proof for the SNOW Theorem [13] and also, answer the open questions mentioned above. First, we formally state the SNOW properties of executions using the I/O automata framework [14] with the additional requirement, for the W property, that any WRITE must eventually complete (§II). Next, we identify and prove some basic results, required in the proofs, for transforming one valid execution to another possible and safe execution in a READ transaction system (§III). Then we present a new, rigorous proof of the impossibility of SNOW with three or more clients even when client-to-client (C2C) communication is allowed (§IV). Next, we show that the feasibility of implementing an algorithm with SNOW in MWSR (which also includes the two-client system model) depends on whether C2C communication is allowed: when it is not allowed, SNOW is impossible (§V-A); and when it is allowed, SNOW is possible (§V-B) for any MWSR setting.

	Client-to-Client?	
Setting	**Yes**	**No**
2 clients	✓	×
MWSR	✓	×
≥ 3 clients	×	(×)

(a) Is SNOW possible?

	Rounds		
Versions	**1**	**2**	**∞**
1	(×)	✓	(✓)
$\|W\|$	✓		

(b) Bounded SNW algorithms.

Fig. 1: A summary of our new results. Previous results are marked in parentheses. × indicates we have proved that such READ transactions are impossible. ✓ indicates we have described such a new READ transaction algorithm. $|W|$ is the number of concurrent WRITE transactions.

Prior to this work, the Eiger [12] algorithm was previously believed to be the only algorithm that provided a bounded number of non-blocking rounds [12] and guaranteed strict serializability. Next, we show this claim is not true by showing that not all execution of Eiger is strictly serializable (§VI).

Next, after realizing the limits posed by the SNOW Theorem, we ask ourselves whether it is possible to construct READ transaction algorithms with no C2C communication, as in most practical systems, where one of the SNOW properties is relaxed. One obvious candidate property is the "O" property, where one of the two restrictions (i.e., "one-round" of communication and "one-version" of data) can be relaxed. We provide two algorithms for the *multiple-writers multi-reader* (MWMR) setting: the first algorithm B guarantees SNW and the "one-version" property and completes READ transactions in two rounds (§VII); the second algorithm, C, guarantees SNW and the "one-round" property but returns up to as many versions of the data as there are concurrent WRITE transactions (§VIII). Thereby, making these READ transactions algorithms with a bounded number of non-blocking rounds and guarantees strict serializability. Due to space limitations, we omit most of the proofs and present them in an extended version in arXiv [9].

II. Transactions Processing System

Web services typically have two tiers of machines within a datacenter: a stateless frontend tier and a stateful storage tier. The frontends handle user requests by executing application logic that generates sub-requests to read/write data in the storage tier that shards (or splits) data across many machines. We refer to the front-ends as the *clients*, the storage machines as the *servers* and the stored data items as *objects*, to match common terminology. While web services are typically geo-replicated, we focus on sharding within a datacenter because the reads that dominate their workloads are handled within a single datacenter.

We consider a transaction processing system that comprises a set of read/write objects $\mathcal{O}$, where each object $o \in \mathcal{O}$ is maintained by a separate server process, and also another set of processes, we refer to as *clients*, that can initiate transactions, after the previous ones, if any, have completed. The system allows two types of transaction: READ transaction, a group of read requests for the values stored in some subset of

objects in $\mathcal{O}$; and WRITE transaction, a group of write requests intending to update the values stored in some subset of objects $\mathcal{O}$. A read-client executes only READ transactions, while write-client executes only WRITE transactions; no client executes both types of transaction.

A typical READ transaction, we denote as $R(o_{i_1}, o_{i_2}, \ldots, o_{i_q})$ or in short by R, consists a set of individual read requests $read(o_{i_1})$, $read(o_{i_2})$ and $read(o_{i_q})$ to read values in objects $o_{i_1}, o_{i_2}, \ldots, o_{i_q}$, respectively. $read(o)$ denotes a read that intends to read the value of object o. A typical WRITE, denoted as $W((o_{i_1}, v_{i_1}), (o_{i_2}, v_{i_2}), \ldots, (o_{i_p}, v_{i_p}))$ or in short as W, consists of a set of which requests to update the values of objects $o_{i_1}, o_{i_2}, \ldots, o_{i_p}$ with $v_{i_1}, v_{i_2}, \ldots, v_{i_p}$, that are values from the domans $V_{i_1}, V_{i_2}, \ldots, V_{i_p}$, respectively.

A read (or write) client initiates a READ (or WRITE) transaction with an invocation step *INV*(R) (or *INV*(W)), then it carries out the read or write operations in the transaction; and eventually completes the transaction with a *RESP*(R) (or *RESP*(W)). After the completion of the reads or writes in a transaction the client responds, in the case of R, with the values of objects; and, in the case of W, an *ok* status, to the external client.

We assume that the network channels are reliable but asynchronous, i.e., any message sent by a process will eventually arrive at its destination uncorrupted. We assume local computations are asynchronous, i.e., local computations at various processes proceed at arbitrary and unpredictable speeds. When a client receives a transaction request, usually from an external client, such as an user's device, it executes the transaction, denote by R or W, and finally, responds to the external client with the results.

We model a distributed algorithm using the I/O automata modeling framework (see [14] for a detailed account). In the rest of this paper, for any execution of an automaton $\mathcal{A}$, $\sigma_0, a_1, \ldots, a_k, \sigma_k \ldots$, where σ's and a's are states and actions, we use the notation $a_1, \ldots, a_k \ldots$ which shows only the actions to simplify notation. We use the notation *prefix*(α, a) to refer to the finite prefix of any execution α ending with action a such that a occurs within α. In our model, an individual read, such as $read(o)$, in some read transaction R initiated by some read client r consists of the following sequence of actions: after *INV*(R) at r, r sends a message m (requesting the value stored in o) to a server s via action *send*$(m)_{r,s}$. When s receives m via action *recv*$(m)_{r,s}$ it sends the value v (stored in o) to r via action *send*$(v)_{s,r}$. Then $read(o)$ completes as soon as r receives v_j via action *recv*$(v)_{s,r}$; R completes with *RESP*(R) after all the reads in it are complete.

A. SNOW Properties

In this subsection, we define the SNOW properties for a transaction processing system. Namely, we require that any fair execution of the system satisfies the following four properties: (i) *Strict serializability* (S), which means there is a total ordering of the transactions such that all transactions in the resulting execution appear to be processed by a single machine one at a time; (ii) *Non-blocking operations* (N), which means that the servers respond immediately to the read requests of a READ transaction without waiting for any input from other processes; (iii) *One response per read* (O), which requires that any read operation consists of one round trip of communication with a server, and also, that the server responds with a message that contains exactly one version of the object value; and (iv) *WRITE transactions that conflict* (W) implies the existence of concurrent WRITE transactions that update the data objects while READ transactions are in progress reading the same objects. Below we describe the individual properties of the SNOW properties in more detail.

Strict serializability (S). By *strict serializability* (for a formal definition please see [8]), we mean each WRITE or READ transaction appears to the clients to be executed atomically, at some point in an execution between the invocation and response events.

Next, we describe the non-blocking and one-response properties. Both are defined as properties of read operations to an individual object. For the purpose of elucidation, we consider an execution α of a transaction processing system T that has a set of objects $\mathcal{O}$, where there is a READ transaction $R(o_{i_1}, o_{i_2}, \ldots, o_{i_q})$, in short R, invoked at some reader r, such that R contains a read $read(o_j)$ for some $o_j \in \mathcal{O}$ maintained at server s_j.

Non-blocking reads (N). The *non-blocking* property means that if r, a reader-client, sends any message to any s_i (s_i manages object o_i) during the transaction then s_i can respond to r without waiting for any external input event, such as the arrival of messages, any mutex operations, time, etc. This property ensures that READ transactions are delayed only due to delay in message delivery between r and s_i. We define this property formally as follows.

Definition II.1 (Non-blocking read (N))**.** *Suppose in α, following the action INV(R), the actions recv$(m_j^r)_{r,s_j}$ and send$(v_j)_{s_j,r}$ corresponding to $read(o_j)$, occurs at s_j. Then there exists an execution α' of T such that*

- (i) *The execution fragments prefix$(\alpha, \text{recv}(m_j^r)_{r,s_j})$ and prefix$(\alpha', \text{recv}(m_j^r)_{r,s_j})$ are identical, where prefix(α, a) is the prefix of α ending with a.*
- (ii) *In α' the action send$(v_j)_{s_j,r}$ at s_j occurs after recv$(m_j^r)_{r,s_j}$ without any input action in between.*

One-response per read (O). The *one-response* property requires that each read operation, $read(o_i)$, during any READ transaction, completes successfully in one round of client-to-server communication and the *one-version* states that exactly one version of the value is sent by server s_i, that manages o_i, to r. *One-round* consists of a read request from the client initiating the read operation to the server and the response containing value sent by the server.

Definition II.2 (One response per read (O))**.** *Suppose in α, the action INV(R) occur at r then in α there exists exactly a pair of actions recv$(m_j^r)_{r,s_j}$ and send$(v_j)_{s_j,r}$, corresponding to R, occur at s_j, such that v_j is the object value of o_j.*

If the reads, of some READ transaction, of a transaction processing system respect the *non-blocking* and *one-response* properties then each read includes one-round trip from client to server, where the server returns only the requested value as soon as it receives the request. It is worth noting that the READ transaction can complete only after all the $read(\cdot)$s in it complete.

Definition II.3 (Conflicting writes (W))**.** *Suppose in α, the action INV(W), the actions occur at a write client w then there is an action RESP(W) in α that appears after INV(W).*

WRITE transactions that conflict (W). The *conflicting writes* property states that READ transactions complete even in the presence of concurrent WRITE transactions, where the write operations might update some objects that are also being read by read operations in READ transaction. This shows that READ transactions can be invoked at any point, even in the presence of ongoing WRITE transactions. Note that the liveness of any WRITE transaction is not implied by any of the SNOW properties; however, for useful practical systems the WRITE transactions must eventually complete. Therefore, we assume that every WRITE transaction eventually completes via the $RESP$ event, and think of this constraint as a part of the W property.

The SNOW Theorem. Consider a transaction processing system with an asynchronous network where a set $\mathcal{O}$ of objects are maintained by individual server processes, with at least one write client and at least two read clients. Then the SNOW Theorem [13] can be stated as follows.

"For any transaction processing system in an asynchronous setting, with at least one writer and two reader clients, and at least two sharded objects, it is impossible to have an algorithm such that all of its executions guarantee the SNOW properties."

III. Technical Preliminaries

In this section, we present some useful preliminary results and ideas that we will later use to prove the impossibility results. We assume a simple system with two servers, s_x and s_y, denote the stored values as x and y, respectively, and either two or three clients. One of the clients is a writer w, which initiates only WRITE transactions. One or two of the clients are readers, r_1 and r_2, which initiate only READ transactions.

Client-to-client (C2C) communication. We consider two types of settings pertinent to communication among the clients: (i) allow C2C communication, where a client can send a message to any other client, and (ii) disallow C2C communications, where a client cannot send any message directly to another client in the system.

Servers s_x and s_y store values for objects o_x and o_y, respectively. the initial values of o_x and o_y are x_0 and y_0, respectively. Because there is one object on each server the server and object identifiers are often used interchangeably to remove redundancy. For instance, we simply say that s_x returns x_0 to the client that initiated the transaction, which means that s_x returns the value x_0 of object o_x at the end of the READ transaction.

Our proofs often use a special type of execution fragment, named non-blocking fragments, that represent the READ transaction algorithm is non-blocking and returns one version of each object. The one-round property is captured by allowing only one non-blocking fragment on each server for a READ transaction. Our proof strategy plays non-blocking fragments against the requirements of strict serializability and write isolation under the freedom of network asynchrony. We explain non-blocking fragments and helper notations in the context of execution α, of the system described as above, as follows:

1) *Non-blocking fragments.* For a READ transaction R_i by reader r_i, $i \in \{1, 2\}$, suppose there is a execution fragment that starts with $recv(m_j^r)_{r_i,s_j}$ and ends with $send(v_j)_{s_j,r_i}$, both of which occur at s_j. Moreover, suppose there is no other input action at s_j in this fragment. Then we call this execution fragment a *non-blocking fragment* for R_i at s_j and denote by $F_{i,j}(\alpha)^{(v_j)}$, $j \in \{x, y\}$(Fig. 2). When the context is clear, we omit the first subscript of F. For instance, for a READ transaction R, $F_x(\alpha)^{(x_0)}$ denotes the non-blocking fragment of R on s_x.
2) Suppose READ transaction R_i completes in α. Consider the execution fragment in α between the event *INV*(R_i) and whichever of the events *send*$(m_y^{r_i})_{r_i,s_y}$ and *send*$(m_x^{r_i})_{r_i,s_x}$ that occurs later. If all the actions in this fragment occur at r_i, then we denote this fragment as $I_i(\alpha)$ (Fig. 2).
3) Suppose READ transaction R_i completes in α. Consider the execution fragments in α that occurs between the later of the events *recv*$(x)_{s_x,r_i}$ or *recv*$(y)_{s_y,r_i}$, i.e., at the point in α when r_i receives responses from both servers, and the event *RESP*(R_i). If all the actions in this fragment occur at r_i, then we denote this fragment by $E_i(\alpha)^{(x,y)}$, where R_i returns the values (x, y) (Fig. 2) to the external client.
4) We use $R(\alpha)$ and $W(\alpha)$ to denote the READ and WRITE transactions in the context of α. When the context is clear, we simply use R and W.
5) We use the subscript of a returned value to denote the version identifier, which uniquely identifies a version from a totally ordered set. For instance, x_0 is the 0^{th} version of x (the initial value of object o_x) on server s_x.

In our proofs, we frequently use arguments that rely on the existence of non-blocking fragments and the constraints of strict serializability and write isolation. Below we state a few useful lemmas regarding the executions, of some algorithm $\mathcal{A}$, where all READ transactions are assumed to have all SNOW properties; we will use these lemmas in later sections. Due to space constraints, we explain these lemmas at a high level.

The following lemma states that a READ transaction has to return the same version from both servers in order to satisfy strict serializability and write isolation.

Lemma 1. *Suppose α is any execution of $\mathcal{A}$ such that READ transaction R is in α. Suppose the execution fragment $I(\alpha) \circ F_x(\alpha)^{(x_t)} \circ F_y(\alpha)^{(y_s)} \circ E(\alpha)^{(x_{t'}, y_{s'})}$ in α, corresponds to R, where $x_t, x_{t'} \in V_1$ and $y_s, y_{s'} \in V_2$, then $s = s' = t = t'$.*

The following lemma states that we can create a new execution α' that is indistinguishable to α by swapping two

adjoining fragments, which happen on two distinct automata in α if either (a) both fragments have no input actions or (b) one of the fragments have no external (input or output) actions. Our proofs leverage this lemma to create new executions by swapping such fragments and finally derive an execution that violates strict serializability.

Lemma 2 (Commuting fragments)**.** *Let α be an execution of $\mathcal{A}$. Suppose $G_1(\alpha)$ and $G_2(\alpha)$ are any execution fragments in α such that all actions in each fragment occur only at one automaton and either (a) none of the fragments contain input actions, or (b) at least one of the fragments have no external actions. Suppose $G_1(\alpha)$ and $G_2(\alpha)$ occur at two distinct automata and the execution fragment $G_1(\alpha) \circ G_2(\alpha)$ occurs in α. Then there exists an execution α' of $\mathcal{A}$, where the execution fragment $G_2(\alpha) \circ G_1(\alpha)$ appears in α', such that (i) $G_1(\alpha) \sim G_1(\alpha')$ and $G_2(\alpha) \sim G_2(\alpha')$ (ii) the prefix in α before $G_1(\alpha) \circ G_2(\alpha)$ is identical to the prefix in α' before $G_1(\alpha') \circ G_2(\alpha')$; and (ii) the suffix in α after $G_1(\alpha) \circ G_2(\alpha)$ is identical to the suffix in α' after the execution fragment $G_2(\alpha') \circ G_1(\alpha')$.*

The following lemma states that if there are two fair executions of $\mathcal{A}$ with READ transaction R in each of them, and suppose at any server the non-blocking fragments of R are identical (in terms of the sequence of states and actions), then R returns the similar values in both executions.

Lemma 3 (Indistinguishability)**.** *Let α and β be executions of $\mathcal{A}$ and let R be any READ transaction. Then (i) if $F_x(\alpha) \overset{s_x}{\sim} F_x(\beta)$ then both $R(\alpha)$ and $R(\beta)$ respond with the same value x at s_x; and (ii) if $F_y(\alpha) \overset{s_y}{\sim} F_y(\beta)$ then both $R(\alpha)$ and $R(\beta)$ respond with the same value y at s_y.*

The following lemma shows that for any finite execution of $\mathcal{A}$ that ends with the invocation of READ transaction R_1, it is always possible to have an extended execution of $\mathcal{A}$ where the fragments I, F_x, F_y and E appear consecutively due to the asynchronous network.

Lemma 4. *If any finite execution of $\mathcal{A}$ ends with INV(R), for a READ transaction R_1 then there exists an extension α which is a fair execution of $\mathcal{A}$ and is of the form $P(\alpha) \circ I(\alpha) \circ F_{1,x}(\alpha)^{(x)} \circ F_{1,y}(\alpha)^{(y)} \circ E(\alpha)^{(x,y)} \circ S(\alpha)$, where $P(\alpha)$ is the prefix and $S(\alpha)$ denotes the rest of the execution.*

IV. No SNOW with Three Clients and C2C

This section provides the sketch of a formal proof of the SNOW Theorem with 3 clients, i.e., SNOW is impossible in a system with 3 or more clients even when client-to-client communication is allowed. The main result of this section is captured by the following theorem.

Theorem 1. *The SNOW properties cannot be implemented in a system with two readers and one writer, for two servers even in the presence of client-to-client communication.*

Our proof strategy is to assume the existence of an algorithm $\mathcal{A}$ that satisfies all SNOW properties and create an execution

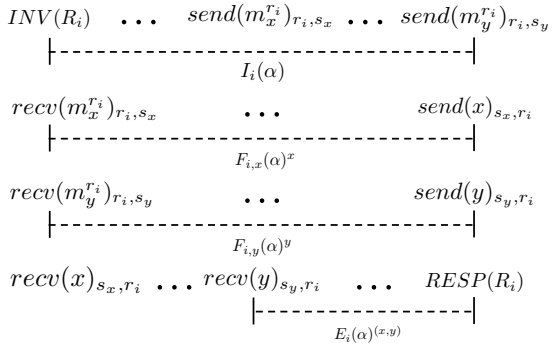

Fig. 2: The relevant actions in the execution fragments $I_i(\alpha)$, $F_{i,x}(\alpha)^x$, $F_{i,y}(\alpha)^y$ and $E_i(\alpha)^{(x,y)}$ for any READ transaction R_i, $i \in \{1, 2\}$ of a fair execution α of $\mathcal{A}$.

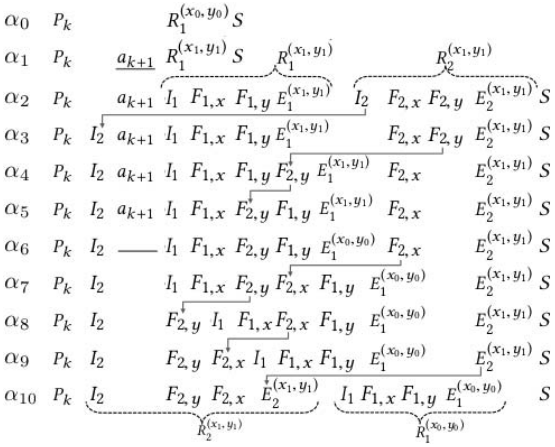

Fig. 3: Executions of $\mathcal{A}$ with three clients and operations W, R_1 and R_2 leading to the contradiction of S in α_{10}. Arrows show the transposition of execution fragments from the previous execution.

α of $\mathcal{A}$ that contradicts the S property. We begin with an execution of $\mathcal{A}$ that contains READ transactions R_1 and R_2, which both read s_x and s_y, and WRITE transaction W that writes (x_1, y_1) to s_x and s_y respectively (both servers have initial values x_0, y_0). R_1 begins after W completes, and R_2 begins after R_1 completes. By the S property both R_1 and R_2 should return (x_1, y_1). Then we create a sequence of executions of $\mathcal{A}$ (Fig. 3), where we interchange the fragments until we finally reach an execution in which R_2 completes before R_1 begins, but R_2 returns (x_1, y_1) and R_1 returns (x_0, y_0) which contradicts the S property.

The following lemma shows that in an execution of $\mathcal{A}$ with a WRITE transaction W and a READ transaction R_1, there exists a point in the execution such that if R_1 is invoked before that point then R_1 returns (x_0, y_0) and if R_1 invoked after that point then R_1 returns (x_1, y_1).

Lemma 5 (Existence of α_0 and α_1)**.** *There exist executions α_0 and α_1 of $\mathcal{A}$ that contain transactions W and R_1 that satisfy the following properties where k is some positive integer and $a_1, \ldots, a_k$ is a prefix of $a_1, \ldots, a_{k+1}$: (i) α_0 can be written*

as $a_1, \ldots, a_k \circ R_1(\alpha_0)^{(x_0,y_0)} \circ S(\alpha_0)$ *; (ii)* α_1 *can be written as* $a_1, \ldots, a_{k+1} \circ R_1(\alpha_1)^{(x_1,y_1)} \circ S(\alpha_1)$*; and (iii)* a_{k+1} *in* α_1 *occurs at* r_1.

Proof: Now we describe the construction of a sequence of finite executions of $\mathcal{A}$, $\{\gamma_k\}_{k=0}^{\infty}$ such that each γ_k contains W and R_1. Consider an execution α of $\mathcal{A}$ that contains W. Suppose R_1 is invoked at r_1 after the execution fragment $a_1, \ldots, a_{k+1}$, a prefix of α. Allowed by network asynchrony, let *INV*(R) be followed by only internal and external actions at r_1 until both *send*$(m_x^{r_1})_{r_1,s_x}$ and *send*$(m_y^{r_1})_{r_1,s_y}$ occur, thereby creating an execution fragment of the form $a_1, \ldots, a_{k+1} \circ I_1(\alpha)$. We denote $a_1, \ldots, a_{k+1}$ by P_{k+1}.

Next, consider the network delivers the message $m_x^{r_1}$ at s_x, and delays all actions at other automata and also any input action at s_x until s_x sends x to r_1. Therefore, we achieve the execution fragment $P_{k+1} \circ I_{1,x}(\alpha) \circ F_{1,x}(\alpha)$ of $\mathcal{A}$. Next, the network delivers $m_y^{r_1}$ at s_y and delays all actions at other automata and input actions at s_y until s_y sends y to r_1. Then the network delivers x and y at r_1 but it delays actions at other automata and any other input action at r_1 until *RESP*(R_1) occurs. Now we have an execution fragment of $\mathcal{A}$, which can be written as $P_{k+1} \circ I_1(\alpha) \circ F_{1,x}(\alpha)^{(x)} \circ F_{1,y}(\alpha)^{(y)} \circ E_1(\alpha)^{(x,y)}$, where R_1 responds with (x, y) such that $(x, y) \in \{(x_0, y_0), (x_1, y_1)\}$. We denote this finite execution prefix as γ_k. Therefore, there exists a sequence of such finite executions $\{\gamma_k\}_{k=0}^{\infty}$.

Because R_1 precedes W, by the S property R_1 must respond with (x_0, y_0) in γ_0. If k is large enough such that a_k occurs in α after the completion of W then by the S property, R_1 must return (x_1, y_1) in γ_{k+1}. Therefore, there exists a minimum k where in γ_k READ transaction R_1 returns (x_0, y_0) and in γ_{k+1}, R_1 returns (x_1, y_1). We denote this minimum by k^*. Note that γ_{k^*} corresponds to α_0 and γ_{k^*+1} corresponds to α_1 in (i) and (ii) respectively.

Now, we prove case (iii) by eliminating the possibility of a_{k^*+1} occurring at s_x, s_y, w or r_2. The S property requires that R_1 must retrieve the same version from both s_x and s_y, which implies that s_x and s_y must send values of the same version. Observe that R_1 returns the 0^{th} version in α_0 and the 1^{st} version in α_1, while the prefixes P_{k^*} and P_{k^*+1} differ by a single action a_{k^*+1}. Importantly, just one action at any of s_x, s_y, r_2 or w is not enough for s_x and s_y to coordinate the same version to send. Therefore, a_{k^*+1} must occur at r_1, which can possibly help coordinate by sending some information via m_x and m_y sent to s_x and s_y respectively.

Case a_{k^*+1} *occurs at* s_x*:* Consider the prefix of execution α_0 up to a_{k^*}. Suppose the network invokes R_1 immediately after action a_{k^*} via *INV*(R_1). By Lemma 4 there exists an execution α' that contains an execution fragment of the form $P_{k^*} \circ I_1(\alpha') \circ F_{1,x}(\alpha')^{(x)} \circ F_{1,y}(\alpha')^{(y)} \circ E(\alpha')^{(x,y)}$. Then, $I_1(\alpha_1) \overset{r_1}{\sim} I_1(\alpha')$ and $F_{1,y}(\alpha_1) \overset{s_y}{\sim} F_{1,y}(\alpha')$ because in both executions the actions of I_1 occur entirely at r_1 and those of $F_{1,y}$ occur entirely at s_y, and thus they are unaffected by the addition of the single action a_{k^*+1} at s_x. As a result, $F_{1,y}(\alpha')$ must send the same value y_1 to r_1 as in $F_{1,y}(\alpha_1)$. Then in α', $R_1(\alpha')$ returns y_1 by Lemma 3, and thus $R_1(\alpha')$ returns (x_1, y_1) by the S property. However, this contradicts the fact that in γ_{k^*} R_1 responds with (x_0, y_0).

Case a_{k^*+1} *occurs at* s_y*:* A contradiction can be shown by following a line of reasoning similar to the preceding case.

Case a_{k^*+1} *occurs at* w*:* This can be argued in a similar manner as the previous case with the trivial fact that $F_{1,x}(\alpha_1) \overset{s_x}{\sim} F_{1,x}(\alpha')$ and $F_{1,y}(\alpha_1) \overset{s_y}{\sim} F_{1,y}(\alpha')$.

Case a_{k^*+1} *occurs at* r_2*:* A contradiction can be derived using a line of reasoning as in the previous case.

So we conclude that a_{k^*+1} must occur at r_1 in α_1. ∎

In the remainder of the section, we suppress the explicit reference to the execution. For instance, we use I_i, $F_{i,x}^{(x)}$, $F_{i,y}^{(y)}$, $E_i^{(x,y)}$ and S, where we drop α, instead of $I_i(\alpha)$, $F_{i,x}(\alpha)^{(x)}$, $F_{i,y}(\alpha)^{(y)}$, $E_i(\alpha)^{(x,y)}$ and $S(\alpha)$. If a READ transaction R_i has an execution fragment of the form $I_i \circ F_{i,x}^{(x)} \circ F_{i,y}^{(y)} \circ E_i^{(x,y)}$ we denote it as $R_i^{(x,y)}$. In the rest of the section, α_0, α_1, and the value of k are the same as in the discussion above. We denote the execution fragments $a_1, \ldots, a_k$ and $a_1, \ldots, a_{k+1}$ as P_k and P_{k+1} respectively. Our proof proceeds by stating a sequence set of lemmas (Fig. 3). The first lemma states there exists an execution in which two consecutive READ transactions follow a WRITE transaction, and both READ transactions return the new values by the WRITE transaction.

Lemma 6 (Existence of α_2)**.** *There exists an execution* α_2 *of* $\mathcal{A}$ *that contains* W*,* R_1*, and* R_2*, and can be written in the form* $P_{k+1} \circ R_1^{(x_1,y_1)} \circ R_2^{(x_1,y_1)} \circ S$*, where both* R_1 *and* R_2 *return* (x_1, y_1).

Based on the previous execution, the following lemma proves that there is an execution of $\mathcal{A}$ where I_2 occurs earlier than the action a_{k+1} and the invocation of R_1.

Lemma 7 (Existence of α_3)**.** *There exists execution* α_3 *of* $\mathcal{A}$ *that contains transactions* W*,* R_1 *and* R_2*, and can be written in the form* $P_k \circ I_2 \circ a_{k+1} \circ R_1^{(x_1,y_1)} \circ F_{2,x} \circ F_{2,y} \circ E_2 \circ S$*, where both* R_1 *and* R_2 *return* (x_1, y_1).

Proof: Consider the execution α_2 as in Lemma 6. In the execution fragment $I_1 \circ F_{1,x}^{(x_1)} \circ F_{1,y}^{(y_1)} \circ E_1^{(x_1,y_1)}$ in α_2, none of the actions occur at r_2 and by Lemma 5, a_{k+1} occurs at r_1, also the actions in I_2 occur only at r_2. Starting with α_2, and by repeatedly using Lemma 2, we create a sequence of four executions of $\mathcal{A}$ by repeatedly swapping I_2 with the execution fragments $E_1^{(x_1,y_1)}$, $F_{1,y}^{(y_1)}$, $F_{1,x}^{(x_1)}$ and I_1, which appears in $I_1 \circ F_{1,x}^{(x_1)} \circ F_{1,y}^{(y_1)} \circ E_1^{(x_1,y_1)} \circ I_2$, where the following sequence of execution fragments $I_1 \circ F_{1,x}^{(x_1)} \circ F_{1,y}^{(y_1)} \circ I_2 \circ E_1^{(x_1,y_1)}$ (by commuting I_2 and $E_1^{(x_1,y_1)}$); $I_1 \circ F_{1,x}^{(x_1)} \circ I_2 \circ F_{1,y}^{(y_1)} \circ E_1^{(x_1,y_1)}$ (by commuting I_2 and $F_{1,y}^{(y_1)}$); $I_1 \circ I_2 \circ F_{1,x}^{(x_1)} \circ F_{1,y}^{(y_1)} \circ E_1^{(x_1,y_1)}$ (by commuting I_2 and $F_{1,x}^{(x_1)}$) appear. Finally, we have an execution α' of the form $P_{k+1} \circ I_2 \circ R_1^{(x_1,y_1)} \circ F_{2,x}^{(x_1)} \circ F_{2,y}^{(y_1)} \circ E_2^{(x_1,y_1)} \circ S$ (by commuting I_2 and I_1) Next, from α', by using Lemma 2 and swapping a_{k+1} with I_2 we have shown the existence of an execution α_3. ∎

Now by constructing a sequence of executions, α_3 through α_{10} (Fig. 3, lemmas and proofs omitted due to lack of space) realize the existence of an execution α_{10} of $\mathcal{A}$ where the execution fragments corresponding to R_2 appears before R_1, where R_1 returns (x_0, y_0) and R_2 completes by returning (x_1, y_1) but R_1 is in real time after R_2, therefore, violates the S property.

Lemma 8 (Existence of α_{10})**.** *There exists an execution α_{10} of $\mathcal{A}$ that contains transactions W, R_1 and R_2 and can be written in the form $P_k \circ R_2^{(x_1,y_1)} \circ R_1^{(x_0,y_0)} \circ S$. where R_1 returns (x_0, y_0) and R_2 returns (x_1, y_1).*

Proof: Now, by applying Lemma 2 to α_9, we can swap $F_{2,x}$ and I_1 to create an execution α_{10} (Fig. 3) of $\mathcal{A}$, which is of the form $P_k \circ I_2 \circ F_{2,y}^{(y_1)} \circ F_{2,x}^{(x_1)} \circ R_1^{(x_0,y_0)} \circ E_2^{(x_1,y_1)} \circ S$, where the returned values are determined by Lemma 1.

Note that none of the actions in $I_1 \circ F_{1,x}^{(x_0)} \circ F_{1,y}^{(y_0)} \circ E_1^{(x_0,y_0)}$ occur at r_2 and all actions in $E_2^{(x_1,y_1)}$ occur at r_2. Therefore, by applying Lemma 2, we can consecutively swap E_2 with E_1, $F_{1,y}$, I_1, and $F_{1,x}$. Therefore, we create a sequence of four executions of $\mathcal{A}$ to arrive at execution α_{10} (Fig. 3) of the form $P_k \circ R_2^{(x_1,y_1)} \circ R_1^{(x_0,y_0)} \circ S$. ■

V. Two Client Open Question

This section closes the open question of whether SNOW properties can be implemented in the MWSR setting. The MWSR setting also generalizes the two-clients setting left open in [13]. We first prove that SNOW remains impossible in a MWSR and 2-server system if C2C communication is disallowed. Next, we present an algorithm that implements SNOW properties in an MWSR setting with at least two servers. Hence we resolve a more general version of the open question raised in [13]: the feasibility of SNOW in this setting depends on whether C2C communication is allowed.

A. No SNOW Without C2C Messages

We use the same system model as in Section IV: two servers s_x and s_y with two clients, a reader r_1 that issues only READ transactions and a writer w that issues only WRITE transactions. A WRITE transaction W writes (x_1, y_1) to s_x and s_y, and a READ transaction R reads both servers. We assume that there is a bi-directional communication channel between any pair of client and server and any pair of servers but there is no direct communication channels between clients. We assume that each transaction can be identified by a unique number, e.g., transaction identifier.

Our strategy is still proof by contradiction: We assume there exists some algorithm $\mathcal{A}$ that satisfies all SNOW properties, and then we show the existence of a sequence of executions of $\mathcal{A}$, finally leading to an execution contradicting the S property. The following theorem (proof in arXiv [9]) states when client-to-client communication is not allowed it is impossible to have the SNOW properties even with two clients.

Theorem 2. *The SNOW properties cannot be implemented in a system with two clients and two servers, where the clients are not allowed to communicate with each other.*

B. SNOW with C2C Communication

In this section, we show that SNOW is possible in the *multiple-writers single-reader* (MWSR) setting when client-to-client communication is allowed. In particular, we present an algorithm A, which has all SNOW properties in such setting. We consider a system that has $\ell \geq 1$ writers with ids $w_1, w_2 \cdots w_\ell \in \mathcal{W}$, one reader r, and $k \geq 1$ servers with ids $s_1, s_2 \cdots s_k \in \mathcal{S}$. Client-to-client communication is allowed. The pseudocode for algorithm A is presented in Pseudocode 4. We use keys to uniquely identify a WRITE transaction. A key $\kappa \in \mathcal{K}$ is defined as a pair (z, w), where $z \in \mathbb{N}$, and $w \in \mathcal{W}$ is the id of a writer. $\mathcal{K}$ denotes the set of all possible keys. Also, with each transaction we associate a tag $t \in \mathbb{N}$.

State variables: (i) Each *writer* w stores a counter z corresponding to the number of WRITE transactions it has invoked so far, initially 0. (ii) The *reader* r has an ordered list of elements, $List$, as $(\kappa, (b_1, \cdots, b_k))$, where $\kappa \in \mathcal{K}$ and $(b_1, \cdots b_k) \in \{0,1\}^k$. Initially, $List = [(\kappa^0, (1, \cdots 1)]$, where $\kappa^0 \equiv (0, w_0)$, and w_0 is any place holder identifier for writer id. (iii) Each *server* $s_i \in \mathcal{S}$ stores a set variable $Vals$ with elements of key-value pairs $(\kappa, v_i) \in \mathcal{K} \times \mathcal{V}_i$. Initially, $Vals = \{(\kappa^0, v_i^0)\}$.

Writer steps: Any writer client, $w \in \mathcal{W}$, may invoke a WRITE transaction $W((o_{i_1}, v_{i_1}), (o_{i_2}, v_{i_2}), \cdots, (o_{i_p}, v_{i_p}))$, comprising a set of write operations, where $I = \{i_1, i_2, \cdots, i_p\}$ is some subset of p indices of $[k]$. We define the set $S_I \triangleq \{s_{i_1}, s_{i_2}, \cdots, s_{i_p}\}$. This procedure consists of two consecutive phases: *write-value* and *info-reader*. In the *write-value* phase, w creates a key κ as $\kappa \equiv (z+1, w)$; and also increments the local counter z by one. Then it sends (WRITE-VAL, (κ, v_i)) to each server s_i in S_I, and awaits ACKs from each server in S_I. After receiving all ACKs, w initiates the *info-reader* phase during which it sends (INFO-READER, $(\kappa, (b_1, \cdots b_k))$ to r, where for any $i \in [k]$, b_i is a boolean variable, such that $b_i = 1$ if $s_i \in S_I$, otherwise $b_i = 0$. Essentially, such a $(k+1)$-tuple identifies the set of objects that are updated during that WRITE transaction, i.e., if $b_i = 1$ then object o_i was updated during the execution of the WRITE transaction, otherwise $b_i = 0$. After w receives ACK from r it completes the WRITE transaction.

Reader steps: We use the same notations for I and S_I as above for the set of indices and corresponding servers, possibly different across transactions. The procedure *R*$(o_{i_1}, o_{i_2}, \cdots, o_{i_p})$, for any READ transaction, is initiated at reader r, where $o_{i_1}, o_{i_2}, \cdots, o_{i_p}$ denotes the subset of objects r intends to read. This procedure consists of only one phase, *read-value*, of communication between the reader and the servers in S_I. Here r sends the message (READ-VAL, κ_i) to each server $s_i \in S_I$, where the κ_i is the key in the tuple $(\kappa_i, (b_1, \cdots, b_k))$ in $List$ located at index j^* such that $b_i = 1$ such that $i \in I$. After receiving the values $v_{i_1}, v_{i_2}, \cdots v_{i_p}$ from all servers in $\mathcal{S}_\mathcal{I}$, where $S_I \triangleq \{s_{i_1}, s_{i_2}, \cdots, s_{i_p}\}$, the transaction completes by returning $(v_{i_1}, \cdots v_{i_p})$.

On receiving a message (INFO-READER, $(\kappa, (b_1, \cdots, b_k))$ from any writer w, r appends $(\kappa, (b_1, \cdots, b_k)$ to its $List$,

and responds to w with ACK and $t_w = |List|$, i.e., number of elements in $List$. The order of the elements in $List$ corresponds to the order the WRITE transactions, the order of the incoming INFO-READER updates, as seen by the reader.

Server steps: The server responds to messages containing the tags WRITE-VAL and READ-VAL. The first procedure is used if a server s_i receives a message $(\text{WRITE-VAL}, (\kappa, v_i))$ from a writer w, it adds (κ, v_i) to its set variable $Vals$ and sends ACK back to w. The second procedure is used if s_i receives a message, i.e., $(\text{READ-VAL}, \kappa_i)$, from r, then it responds with v_i such that (κ_i, v_i) is in its $Vals$.

Pseudocode. 4 Steps at writer w, reader r and server s_i in A.

```
At writer w
State Variables at w:
z ∈ ℕ, initially 0
W((o_{i_1}, v_{i_1}), ··· , (o_{o_p}, v_{i_p}))
write-value:
  κ ← (z + 1, w)
  z ← z + 1
  I ≜ {i_1, i_2, ··· , i_p}
  for i ∈ I do
    Send (WRITE-VAL, (κ, v_{s_i})) to s_i
  Await ACK from s_i ∀ i ∈ I.
info-reader:
  for i ∈ [k] do
    if i ∈ I then
      b_i ← 1
    else
      b_i ← 0
  Send (INFO-READER, (κ, (b_1, ··· , b_k))) to r
  Receive (ACK, t_w) from r
-----------------------------------------------
At reader r
State Variables at r:
List, a list of elements in K × {0,1}^k,
  initially [(κ^0, 1, ··· 1)]
  Await responses v_i from s_i ∀ i ∈ I
  Return (v_{i_1}, v_{i_2}, ··· , v_{i_p})
R(o_{i_1}, o_{i_2}, ··· , o_{i_p})
read-value:
  I ≜ {i_1, i_2, ··· , i_p}
  for i ∈ I do
    j* ← max_{1≤j≤|List|} {j : List[j].b_i = 1}
    κ_i ← List[j*].κ
    Send (READ-VAL, κ_i) to s_i
Response routines
On recv (INFO-READER, (κ, (b_1, ··· b_k))) from w:
  List ← List ⊕ (κ, (b_1, ··· b_k))
  /* ⊕ for append */
  tag ← |List| /* | · | list size */
  Send (ACK, tag) to w
-----------------------------------------------
At server s_i for any i ∈ [k]
State Variables:
Vals ⊂ K × V_i, initially {(t^0_key, v^0_i)}
On recv (WRITE-VAL, (κ, v)) from w:
  Vals ← Vals ∪ {(κ, v)}
  Send ACK to w.
On recv (READ-VAL, κ) from r :
  Send v s.t. (κ, v) ∈ Vals to r
```

A respects the SNOW properties as stated below.

Theorem 3. *Any well-formed and fair execution of A guarantees all of the SNOW properties.*

VI. No Prior Bounded Latency for SW

The SNOW work [13] claimed, after examining existing, work there existed only one system, Eiger [12], whose READ transactions had bounded latency—i.e., non-blocking and finish in three rounds—while providing the strongest guarantees– i.e., having properties W and S—because Eiger claimed that its READ transactions provide strict serializability within a datacenter. In this section, we correct this claim and show there were no existing algorithms that had bounded latency

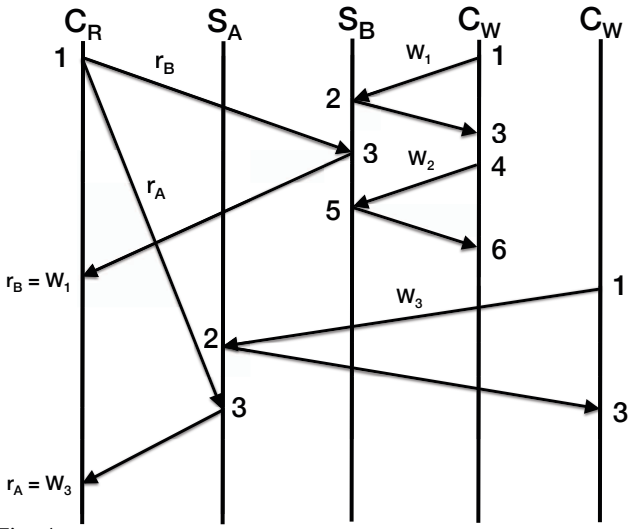

Fig. 4: An example execution that shows Eiger's READ transaction is not strictly serializable. w_1, w_2, and w_3 are three writes, where w_3 is issued after w_2 finishes. r_A and r_B are read operations from the same READ transaction R, which is concurrent with all three writes. Each number is a value of the logical clock on the machine (process) as a result of message exchange.

while providing the strongest guarantees by proving Eiger's READ transactions are not strictly serializable.

The insight on why Eiger's READ transactions are not strictly serializable is that Eiger uses logical timestamps, i.e., Lamport clocks, to track the ordering of operations, and logical clocks are not able to identify the real-time ordering between operations that do not have causal relationship, i.e., operations from different processes. Strict serializability, however, requires the real-time ordering to be respected.

Figure 4 is an example execution, which is allowed by Eiger but violates strict serializability. Real time goes downwards in the diagram. Each number is a value of the Lamport clock on the machine (process) as a result of message exchange. Initially all processes have Lamport clock value 0, and no messages happened before the execution in Figure 4. A READ transaction $R = \{r_A, r_B\}$ reads values on S_A and S_B respectively. Due to the asynchronous nature of the network, r_B arrives on S_B before w_2 and r_A arrives after w_3. Following the READ transaction algorithm of Eiger, r_A returns the value of w_3 and its valid logical duration, i.e., $[2, 3]$. Similarly, r_B returns the value of w_1 and its valid logical duration, i.e., $[2, 3]$. Because the two logical durations overlap, Eiger claims the combined values of r_A and r_B are consistent and accept them. However, because w_3 starts after w_2 finishes, w_3 is in real time after w_2. By strict serializability, if a READ transaction sees the value of w_3, then it must observe the effect of w_2. Hence, R, which returns the values of w_1 and w_3, violates strict serializability.

VII. SNW + One Version, MWMR setting

Here present algorithm B, which satisfies SNW and "one-version" properties, in MWMR setting where a READ transaction must consist of one version of the data but, possibly, mul-

tiple communication trips between the reader and the servers. In B, the steps for the writers are shown in Pseudocode 5 and for readers and the servers are presented in Pseudocode 6. We assume a set of writers $\mathcal{W}$, a set of readers $\mathcal{R}$ and a set of $k \geq 1$ servers, $\mathcal{S}$, with ids $s_1, s_2 \cdots s_k$ that stores the objects $o_1, o_2, \cdots, o_k$, respectively. A key κ is defined as a pair (z, w), where $z \in \mathbb{N}$ and $w \in \mathcal{W}$ the id of a writer. We use $\mathcal{K}$ to denote the set of all possible keys. In B, a key uniquely identifies some transaction. Also, with each transaction we associate a tag $t \in \mathbb{N}$.

In B, we designate one of the servers as coordinator, denote as s^*, for the transactions. The s^* maintains the order of the WRITE transactions and the objects that are updated during the WRITE transaction in the variable $List$.

State variables: Each of the writers and servers maintain a set of state variables as follows: (i) At any *writer* w, there is a counter z to keep track of the number of WRITE transaction the writer has invoked, initially 0. (ii) At any *server*, s_i, for $i \in [k]$, there is a set variable $Vals$ with elements that are key-value pairs $(\kappa, v_i) \in \mathcal{K} \times \mathcal{V}_i$. Initially, $Vals = \{(\kappa^0, v_i^0)\}$. A server also contains an ordered list variable $List$ of elements as $(\kappa, (b_1, \cdots, b_k))$, where $\kappa \in \mathcal{K}$ and $(b_1, \cdots b_k) \in \{0,1\}^k$. Initially, $List = [(\kappa^0, (1, \cdots 1)]$, where $\kappa^0 \equiv (0, w_0)$, where w_0 is any place holder identifier string for writer id. The elements in $List$ can be identified with an index, e.g., $List[0] = (\kappa^0, (1, \cdots, 1))$. Essentially, a $(k+1)$-tuple $(\kappa, (b_1, \cdots, b_k))$ in $List$ corresponds to a WRITE transaction and identifies the set of objects that are updated during the WRITE transaction, i.e., if $b_i = 1$ then object o_i was updated during the WRITE transaction, otherwise $b_i = 0$.

Writer steps: A WRITE transaction updates a list of p objects $o_{i_1}, o_{i_2}, \cdots o_{i_p}$ with values $v_{i_1}, v_{i_2}, \cdots v_{i_p}$, respectively, is invoked at w via the procedure $W((o_{i_1}, v_{i_1}), \cdots, (o_{i_p}, v_{i_p}))$. We use the notations: $I \triangleq \{i_1, i_2, \cdots, i_p\}$ and $S_I \triangleq \{s_{i_1}, s_{i_2}, \cdots, s_{i_p}\}$. This procedure consists of two phases: *write-value* and *update-coor*. During the *write-value* phase, w creates a new key κ as $\kappa \equiv (z+1, w)$, where w identifies the writer; and also increments the local counter z by one. Then w sends (WRITE-VAL, (κ, v_i)) to each server in S_I, and awaits ACK from all servers in S_I. After receiving ACK from all servers in S_I, w initiates the *update-coor* phase where it sends (UPDATE-COOR, $(\kappa, (b_1, \cdots b_k))$ to s^*, where for any $i \in [k]$, $b_i = 1$ if $s_i \in S_I$, otherwise $b_i = 0$, and completes then WRITE transaction after it receive a (ACK, t_w) from s^*.

Reader steps: We use the same notations for I and S_I as above but the indices can vary across transactions. The procedure $R(o_{i_1}, o_{i_2}, \cdots, o_{i_p})$ can be initiated by some reader r, as a READ transaction, intending to read the values of subset $o_{i_1}, o_{i_2}, \cdots, o_{i_p}$ of the objects. The procedure consists of two consecutively executed phases of communication rounds between the r and the servers, viz., *get-tag-array* and *read-value*. During the phase *get-tag-array*, r sends s^* the message GET-TAG-ARR requesting the list of the latest added keys for each object. Once r receives a list of tags, such as, $(t_r, (\kappa_1, \kappa_2, \cdots, \kappa_k))$ from s^* the phase completes. In the subsequence phase, *read-value*, r requests each server s_i in S_I by sending the message (READ-VAL, κ_i). After receiving the values $v_{i_1}, v_{i_2}, \cdots v_{i_p}$ from the servers in $\mathcal{S}_\mathcal{I}$, r completes the transaction by returning the tuple of values $(v_{i_1}, \cdots v_{i_p})$.

Server steps: When a server s_i receives a message of type (WRITE-VAL, (κ, v_i)) from a writer w then it adds (κ, v_i) to its set variable $Vals$ and sends ACK back to w.

If the coordinator s^* receives (UPDATE-COOR, $(\kappa, (b_1, \cdots, b_k))$ from writer w, then it appends $(\kappa, (b_1, \cdots, b_k))$ to its $List$, and responds with ACK and t_w (set to be the number of elements in the local list $List$) to w. The order of the elements in $List$ corresponds to the order the WRITE transactions, the order of the incoming UPDATE-COOR updates, as seen by s^*.

When s^* receives the message GET-TAG-ARR from r it responds with $(\kappa_1, \cdots, \kappa_k)$ such that for each $i \in [k]$, κ_i is the key part of the $(k+1)$-tuple that was modified last, i.e., $\kappa_i = List[j^*].\kappa$ such that $j^* \triangleq \max\{j : List[j].b_i = 1\}$, and t_r, $t_r \triangleq \max_{1 \leq j \leq |List|}\{j : List[j].b_i = 1 \wedge i \in I\}$. If any server s_i receives a message (READ-VAL, κ) from a reader r then it responds to r with the value v_i corresponding to key with value κ in $Vals$.

Theorem 4. *Any well-formed and fair execution of algorithm B satisfies the SNW and "one-version" properties.*

Pseudocode. 5 Protocol for writer w in algorithms B and C.

```
    At writer w
    State Variables:
    z ∈ ℕ, initially 0
    W((i_1, v_{i_1}), ··· , (i_p, v_{i_p}))
    I ≜ {i_1, i_2, ··· , i_p}
3:  write-value:
      κ ← (z + 1, w)
      z ← z + 1
6:  for i ∈ I do
      Send (WRITE-VAL, (κ, v_{s_i}))
    to s_i
9:  Await ACK from servers in S_I.
    update-coor:
      for i ∈ [k] do
12:     if i ∈ I then
          b_i ← 1
        else
15:       b_i ← 0
      Send (UPDATE-COOR, (κ,
    (b_1, ··· , b_k))) to s*
18:   Receive (ACK,
    t_w) from s*
```

VIII. SNW + ONE ROUND, MWMR SETTING

Here, we present algorithm C which satisfies SNW and "one-round" properties in the MWMR setting, where a *READ* consists one round of communications between the reader and the servers but servers may respond with multiple versions of the data. The notation for the writers, servers and tag are similar to algorithm B. Pseudocodes 5 and 7 show the steps for the writers, and the readers and the servers, respectively. We designate a server as the coordinator, denote as s^*.

State variables: The state variables are similar to B.

Writer steps: *WRITE* transaction is similar to algorithm B..

Reader steps: The step $R(o_{i_1}, o_{i_2}, \cdots, o_{i_p})$ can be initiated by some reader r intending to read the values of subset $o_{i_1}, o_{i_2}, \cdots, o_{i_p}$ of the objects. Denote $I \triangleq \{i_1, i_2, \cdots, i_p\}$ and $S_I \triangleq \{s_{i_1}, s_{i_2}, \cdots, s_{i_p}\}$. The procedure consists of only one phase of communication round between the r and the servers, called *read-values-and-tags*. During *read-values-and-tags*, r sends s^* the message GET-TAG-ARR requesting the list of the latest added keys for each object, and also sends requests

Pseudocode. 6 Protocols reader r and server s_i in alg. B.

At reader r
R$(o_{i_1}, o_{i_2}, \cdots, o_{i_p})$
$I \triangleq \{i_1, i_2, \cdots, i_p\}$
get-tag-array:
Send (GET-TAG-ARR) to s^*
Receive $(t_r, (\kappa_1, \kappa_2, \cdots, \kappa_k))$ from s^*
read-value:
for $i \in I$ **do**
Send (READ-VAL, κ_i) to s_i
Await responses as $v_i \; \forall \; s_i \in S$
Return $(v_{i_1}, v_{i_2}, \cdots, v_{i_p})$

At server s_i **for any** $i \in [k]$
State Variables:
$Vals \subset \mathcal{K} \times \mathcal{V}_i$, initially $\{(\kappa^0, v_i^0)\}$
$List$, list of $\mathcal{K} \times \{0,1\}^k$, initially $[(\kappa^0, (1, \cdots 1))]$
On recv (WRITE-VAL, (κ, v)) **from** w:
$Vals \leftarrow Vals \cup \{(\kappa, v)\}$
Send ACK to w.
On recv (UPDATE-COOR, $(\kappa, (b_1, \cdots b_k))$) **from** w :
$List \leftarrow List \bigoplus (\kappa, (b_1, \cdots b_k))$
//$\bigoplus$ for append
$tag \leftarrow |List|$ // $|\cdot|$ list size
Send (ACK, tag) to w
On recv (READ-VAL, κ) **from** r :
Send v_i s.t. $(\kappa, v) \in Vals$ to r
/* used only by s^* */
On recv GET-TAG-ARR **from** r :
for $i \in [k]$ **do**
$j^* \leftarrow \max\{j : List[j].b_i = 1\}$
$\kappa_i \leftarrow List[j^*].\kappa$
$t_r \triangleq \max_{1 \le j \le |List|}\{j : List[j].b_i = 1 \wedge i \in I\}$
Send $(t_r, (\kappa_1, \kappa_2, \cdots, \kappa_k))$ to r

Pseudocode. 7 Protocols for reader r & server s_i in alg. C.

At reader r
R$(o_{i_1}, o_{i_2}, \cdots, o_{i_p})$
$I \triangleq \{i_1, i_2, \cdots, i_p\}$
read-values-and-tags:
Send (GET-TAG-ARR) to s^*
for $i \in I$ **do**
Send (READ-VALS) to s_i
Recv $(t_r, (\kappa_1, \kappa_2, \cdots, \kappa_k))$ from s^*
Recv. $Vals_i$ from $\forall \; s_i \in S_I$
Return $(v_{i_1}, v_{i_2}, \cdots, v_{i_p})$
s.t. $(\kappa_j, v_j) \in Vals_j$, $j \in I$

At server s_i **for any** $i \in [k]$
State Variables:
$Vals \subset \mathcal{K} \times \mathcal{V}_i$, initially $\{(\kappa^0, v_i^0)\}$
$List$, a list of $\mathcal{K} \times \{0,1\}^k$, initially $[(\kappa^0, (1, \cdots 1))]$
On recv (WRITE-VAL, (κ, v)) **from** w:
$Vals \leftarrow Vals \cup \{(\kappa, v)\}$
Send ACK to writer w.
On recv (UPDATE-COOR, $(\kappa, (b_1, \cdots b_k))$) **from** w:
$List \leftarrow List \bigoplus (\kappa, (b_1, \cdots b_k))$
//$\bigoplus$ for append
$tag \leftarrow |List|$ /* $|\cdot|$ list size */
Send (ACK, tag) to w
On recv (READ-VALS) **from** r :
Send $Vals$ to r
/* used only by s^* */
On recv GET-TAG-ARR **from** r :
for $i \in [k]$ **do**
$j^* \leftarrow \max\{j : List[j].b_i = 1\}$
$\kappa_i \leftarrow List[j^*].\kappa$
$t_r \triangleq \max_{1 \le j \le |List|}\{j : List[j].b_i = 1 \wedge i \in I\}$
Send $(t_r, (\kappa_1, \kappa_2, \cdots, \kappa_k))$ to r

(READ-VALS) each server s_i in S_I. Note that if s^* is also one of the servers in S_I then the GET-TAG-ARR and READ-VALS messages to s^* can be combined to create one message; however, we keep them separate for clarity of presentation. Once r receives a list of tags, such as, $(t_r, (\kappa_1, \kappa_2, \cdots, \kappa_k))$ from s^* and the set of $Vals_i$ from each $s_i \in S_I$ then r returns the values $v_{i_1}, v_{i_2}, \cdots v_{i_p}$ such that $(\kappa_j, v_j) \in Vals_j$, $j \in \{1, \cdots p\}$, and completes the *READ*.

Server steps: When a server s_i receives a message (WRITE-VAL, (κ, v_i)) from a writer w or s^*, receives (UPDATE-COOR, $(\kappa, (b_1, \cdots, b_k))$ from writer w or receives a message as GET-TAG-ARR r the steps are similar to those in B. On the other hand, if any server s_i receives a message (READ-VALS) from a reader r then it responds to r with $Vals$. The following result states that C respects SNW and "one-round" properties.

Theorem 5. *Any well-formed and fair execution of C, in the MWMR setting satisfies the SNW and "one-round" properties.*

IX. Conclusion

We revisited the SNOW Theorem and when it is possible for READ transactions to have the same latency as simple reads. We provided a new and more rigorous proof of the original result. We also closed several open questions that were either explicitly posed by the original work or that emerged from our careful analysis. We found that READ transactions can match the latency of simple reads when client-to-client communication is allowed in MWSR setting. We found that they cannot and must have higher worst-case latency when client-to-client communication is disallowed or there are at least two readers. We also presented the first algorithms that provide bounded worst-case latency for read-only transactions in strictly serializable systems with WRITE transactions.

Acknowledgment

This work was supported by NSF awards CNS-1824130, CCF-2003830, CCF-0939370 and NSF CCF-1461559.

References

[1] Hagit Attiya and Jennifer L. Welch. Sequential consistency versus linearizability. *ACM Trans. Comput. Syst.*, 12(2):91–122, 1994.
[2] Eric A. Brewer. Towards robust distributed systems. In *Proc. Principles of Distributed Computing*, Jul 2000.
[3] Nathan Bronson and et. al. TAO: Facebook's distributed data store for the social graph. In *USENIX Annual Technical Conference (USENIX ATC 13)*., pages 49–60, 2013.
[4] Phil Dixon. Shopzilla site redesign: We get what we measure. Velocity Conf. Talk, 2009.
[5] James C. Corbett et. al. Spanner: Google's globally-distributed database. In *Proc. OSDI*, Oct 2012.
[6] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. In *Proc. Prin. of Database Sys*, 1983.
[7] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. In *ACM SIGACT News*, Jun 2002.
[8] M. P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.
[9] Kishori M Konwar, Wyatt Lloyd, Haonan Lu, and Nancy Lynch. The snow theorem revisited, 2018.
[10] Greg Linden. Make data useful. Stanford CS345 Talk, 2006.
[11] Richard J. Lipton and Jonathan S. Sandberg. PRAM: A scalable shared memory. Technical Report TR-180-88, Princeton Univ., Dept. Comp. Sci., 1988.
[12] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Anderson. Stronger Semantics for Low-Latency Geo-Replicated Storage. In *Proc. NSDI*, Apr 2013.
[13] H. Lu, C. Hodsdon, K. Ngo, S Mu, and W. Lloyd. The SNOW theorem and latency-optimal read-only transactions. In *12th USENIX Symp. on Operating Sys. Design and Implementation (OSDI 16)*, pages 135–150, Savannah, GA, 2016.
[14] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Pub., 1996.
[15] Christos H. Papadimitriou. The serializability of concurrent database updates. *Journal of ACM*, 26(4):631–653, 1979.
[16] E. Schurman and J. Brutlag. The user and business impact of server delays, additional bytes, and HTTP chunking in web search. Velocity Conf. Talk, 2009.