The SNOW Theorem Revisited

Kishori M Konwar, Wyatt Lloyd, Haonan Lu, Nancy Lynch

November 27, 2018

Abstract

In highly-scalable storage systems for Web services, data is sharded into separate *objects*, also called *shards*, across several servers. Transaction isolation, while reading the objects, is at the heart of consistent data access when concurrent updates are present. In practice, systems experience a much higher number of READ transactions, consisting only of read operations, compared to WRITE transactions; consequently, lowering latency of READ transactions boosts service performance. In [9], the authors proposed four desirable properties in transaction processing systems for achieving low-latency of READ transactions, with asynchronous and reliable communications, and referred to them collectively as the *SNOW properties*: The underlying properties, in the context of an execution, are (*i*) *strict serializability* (S) property where read and WRITE transactions seem to occur atomically; (*ii*) *non-blocking* (N) property implies that for every read operation on any object, during a READ transaction, the response at the corresponding server is non-blocking; (*iii*) *one version and one round* (O) property implies every read operation, during a read transaction, completes in *one-round* of client-server communication and the respective server responds with only one version of the object value; and (*iv*) *concurrent* WRITE *transactions* (W) property states that READ transactions can have concurrent WRITE transactions. Then they argued that it is impossible to implement all the four properties, in the same system, even with at least three clients. They referred to their result as the *SNOW* theorem, and they posed the two-client setting as an open question.

Here we revisit the results of the SNOW theorem and present several new results. In our first result, we resolve the two-client scenario: We prove that even with two clients, without client-to-client messaging, it is impossible to design an transaction processing system which satisfies the SNOW properties. Second, we provide a rigorous proof of the SNOW theorem for systems with at least three clients, i.e., we show that it is impossible to implement a transaction processing system, consisting of at least three clients, even with client-to-client messaging, that satisfies the SNOW properties. In our third result, we derive a useful property for executions of algorithms that implement objects of data types considered in our work that helps us show the *strict serializability* property (S property) of algorithms presented in the paper.

In our fourth result, we present an algorithm with *multiple writers, single reader* (MWSR) which satisfies the SNOW properties, with client-to-client messaging. In our fifth, we present an algorithm, for *multiple-writer, multiple-reader* (MWMR) setting in the absence of client-to-client messaging, which satisfies the "S", "N", "W" properties and a weaker version the O property "o", where we use "o" to refer to the one-version requirement of a read operation, but it can take multiple rounds of communication to complete a read operation. Collectively, we refer to the "S", "N", "o" and "W" properties as the "SNoW" property. In our sixth result, we present an algorithm in the MWMR setting which satisfies the "S", "N", "W" properties and a property "$\tilde{o}$", which refers to the one-round requirement of a read operation, but a server can response with multiple versions of a shard. We refer to these properties as the "SN$\tilde{o}$W" property.

1 Introduction

Most highly accessed websites rely on large volumes of data in the back end. Scalability of these websites is dependent on the ability to serve thousands of web-requests per second. Although it is desirable to store all of this data at a single server, due to limited resources available on a single machine it is impossible to do so. Therefore, such data is partitioned and stored across multiple machines connected through high-speed networks. Subsequently, serving one webpage request may involve retrieving various data objects of the page from multiple servers. These individual data objects are often referred to as *shards* or *objects*, where each object is managed by a separate server. For example, a dynamically generated webpage may contain various objects, such as texts, images, and videos: the text elements may be stored in one server, the images and thumbnails in another, and, similarly, the video clips may be managed by a separate server. During the webpage synthesis and loading, the proxy reader reads all of these separate objects via a READ transaction. Some of these objects may be updated while READ transactions are carried out. A *transaction* consists of a set of read or write operations on the underlying objects which are to be executed atomically.

In our work, we consider two types of transactions: WRITE *transactions* and READ *transactions*. A READ transaction consists of a set of read operations, one per each object, and a WRITE transaction comprises of a set of individual write operations on a subset of the objects. Consistent data access during a transactional read requires *transactional isolation*, where reads to different objects must retrieve either all updates for a given update or none. Similar transactional isolation of WRITE transactions is also desirable, where the collection of all the write operations to different objects is carried out atomically. Handling transactional isolation with concurrent read and WRITE transactions is complex and expensive which may often lead to poor performance and lower throughput, which in turn leads to substantial revenue loss and poor user experience. In practice, it is observed that most of the transactions are READ transactions, and therefore, one approach to boost performance is to improve the performance of the READ transactions [3].

In [9], the authors study the question whether READ transactions can be fast enough even in the presence of concurrent WRITE transactions. They introduce four desirable properties of a system to achieve low-latency READ transactions, which they refer to collectively as the *SNOW property*: **S**trict serializability among transactions, **N**on-blocking operations and **O**ne-response from each shard for the READ transactions, and compatibility with conflicting **W**rite transactions. The "strict serializability" property requires that the set of read or writer operations of any read and WRITE transactions is executed atomically as if executed by a single server. "Non-blocking" requires the read requests corresponding to different objects to be served by the managing servers immediately, without blocking for any other external input or output messages. The "one-response" property requires that for any read operation of an object, the server managing the object responds with only one version, and the operation completes in one round of client to server communication. The final property "conflicting and concurrent WRITE transactions" allows WRITE transactions to execute concurrently with READ transactions on the same set of objects.

In distributed systems, the availability of the systems is often stated in terms of liveness properties of the transactions or operations [5, 6, 12]. Such a requirement on responsiveness of a system is also referred to as availability. One of the strongest, and the most desirable of these properties is *wait-freedom*, which requires that as long as the client is taking a sufficient number of steps, it completes the operations irrespective of the speed of other clients. In [9], the wait-free property of the read operations, corresponding to a READ transaction in a system that satisfies SNOW properties, is inherent in the N and O properties; this also implies the wait-freedom property for

READ transactions. However, in [9], no liveness property for WRITE transactions is stated explicitly. In their work, the authors prove that with at least three clients, in an asynchronous message-passing environment, even in the absence of any failures, it is impossible to have a protocol that satisfies all of the four properties of SNOW, which they refer to as the *SNOW theorem*. In their proof, they consider a system with two read clients, which issues only READ transactions, and a write client invoking WRITE transactions. From their proof, it is not clear whether they consider the possibility of exchanging messages between clients. Additionally, there is no explicitly stated requirement on the liveness of the WRITE transactions. They pose the case for two clients as an unresolved case, i.e., it is not known whether all of SNOW properties can be implemented for any two client system.

Our contribution In this work, we revisit the results of the SNOW theorem and present four new results. First, we show that even with two clients, in the absence of client-to-client messaging, it is impossible to design a transaction processing system which satisfies the SNOW properties. Second, we provide a thorough proof of the SNOW theorem for systems with at least three clients, i.e., we show that it is impossible to implement a transaction processing system, which can implement the SNOW property, consisting of at least three clients, even with client-to-client messaging. In our proof, we explicitly make assumptions regarding the liveness of WRITE transactions and client-to-client messaging. Our proof technique involves assuming an execution of an arbitrary transaction processing system, assumed to be specified as an I/O automaton, that satisfies the SNOW property, and then, argue by constructing a sequence of executions of the automata, finally, leading to an execution which contradicts the S property. The above two results indicate that client-to-client messaging ability plays a key role in the achievability of the SNOW property. This motivates our third result, where we present an algorithm with *multiple writers, single reader* (MWSR) which satisfies the SNOW properties when client-to-client messaging is allowed. Finally, we present an algorithm, for *multiple-writer, multiple-reader* (MWMR) setting in the absence of client-to-client messaging, which satisfies the "S", "N", "W" properties and and a weaker version the O property, which we denote by "o", where "o" refers to the one-version property of a read operation, but can take multiple rounds of communication to complete the read operation. Collectively, we refer to the "S", "N", "o" and "W" properties as the "SNoW" property.

Related work This will be filled in depending what type of conference we send the paper to and whether we add some experimental results of the two algorithms presented in the paper.

Document Structure The remainder of the document is organized as follows. Section 2 presents the models and definitions. Section 3, provides the proof for the impossibility result for the two-client setting when clients do not communicate among each other. In section 3, we present the impossibility result for the three-client setting. Next, in section 6 we present the algorithm with multi-writer, single-reader (MWSR) which satisfies the SNOW properties, when client-to-client communication is allowed. In section 7, we present an algorithm for multi-writer, multi-reader scenario which satisfies the SNoW properties.

2 System model and architecture

2.1 I/O Automata

We consider a distributed system consisting of *asynchronous* processes of three types: a set of *readers* and *writers*, called clients, and a set of *servers*. Each of these processes is associated with an unique identifier. Processes can communicate via asynchronous reliable communication links. This means that any message sent on the link is guaranteed to eventually reach the destination process. The model assumes that client or server processes never fail. We do not make any assumption regarding the order of message delivery in the same channel.

We assume that the formal specification of an algorithm in our context is specified using the I/O Automata. An algorithm is specified from the composition A of a set of *automata* where each automaton A_i corresponds to a process i in the system. An automaton A_i consists of a set of states $states(A_i)$, including a special subset $start(A_i) \subseteq states(A)$ called the *start states* ; a *signature*, denoted $sig(A_i)$; and a set of *transitions* $trans(A_i)$. The set of states $states(A_i)$ are essentially defined by a set of state variables in A_i. The signature $sig(A_i)$ consists of three disjoint sets in $in(A_i)$, $out(A_i)$ and $int(A_i)$, where $in(A_i)$ referred to as the *input actions*, $out(A_i)$ the *output actions* and $int(A_i)$ are the *internal actions*. The set $trans(A_i)$ consists of the set of *transitions* or *steps*, where each step consists of a 3-tuple $(\sigma_i, a_i, \sigma_{i+1})$, where $\sigma_i, \sigma_{i+1} \in states(A_i)$ and $a_i \in trans(A_i)$. The set $in(A_i) \cup out(A_i)$, often denoted as $ext(A_i)$, are called the *external signature* of automaton A_i. Note that A is a composition of the set of automata A_i, $i \in \mathcal{I}$ and the following analogous components for A are defined as $states(A) \triangleq \prod_{i \in \mathcal{I}} states(A_i)$; $sig(A) \triangleq \prod_{i \in \mathcal{I}} sig(A_i)$; $start(A) \triangleq \prod_{i \in \mathcal{I}} start(A_i)$ and $trans(A) \triangleq \{(\sigma, a, \sigma') :$ if $a \in trans(A_i)$ for some $i \in \mathcal{I}$ then $\sigma, \sigma' \in states(A_i)$, else $\sigma = \sigma'\}$.

Informally, the execution of the algorithm corresponds to the concept of the execution of A. First, an *execution fragment* α of A is a finite or infinite sequence $\sigma_0, a_1, \sigma_1, a_2, \sigma_2, \ldots$, where each σ_i belong to $states(A)$ and a_i belongs to $trans(A)$. An *execution* of A is an execution fragment such that $\sigma_0 \in start(A)$. We use the notation $trace(\alpha)$ to denote the sequence of external actions in α; and by $trace(\alpha)|A_i$ to denote the sequence of external actions in $trace(\alpha)$ that belongs to automaton A_i. If α is a finite execution and β is an execution fragment, such that, β starts with the final state of α then we use the symbol $\alpha \circ \beta$ to denote execution fragment obtained as a result of concatenation of execution fragments α and β. The set of locally controller actions in an automaton can be partitioned into tasks. Moreover, in the case of an execution, if the execution is infinite and events from each class occurs infinitely often, and if the execution is finite and if none of the events from any of the classes is enabled then the execution is called *fair*. If ϵ and ϵ' are two execution fragments such that at some automaton A they are the same, i.e., $\epsilon|A = \epsilon'|A$, then we use the notation $\epsilon \overset{A}{\sim} \epsilon'$. When the automaton is clear from the context, for the sake of brevity we drop the symbol for the automaton over the $\sim$. Below we add some useful theorems are useful related to executions of a composed I/O Automata [10].

Theorem 2.1. *Let $\{A_i\}_{i \in I}$ be a compatible collection of automata and let $A = \Pi_{i \in I} A_i$. Suppose α_i is an execution of A_i for every $i \in I$, and suppose β is a sequence of actions in $ext(A)$ such that $\beta|A_i = trace(\alpha_i)$ for every $i \in I$. Then there is an execution α of A such that $\beta = trace(\alpha)$ and $\alpha_i = \alpha|A$, for every $i \in I$.*

Theorem 2.2. *Let A be any I/O automaton.*

1. *If α is a finite execution of A, then there is a fair execution of A that starts with α.*

2. *If β is a finite trace of A, then there is a fair trace of A that starts with β.*

3. *If α is a finite execution of A and β is any finite or infinite sequence of input actions of A, then there is a fair execution $\alpha \circ \alpha'$ of A such that the sequence of input actions in α' is exactly β.*

4. *If β is a finite trace of A and β' is any finite or infinite sequence of input actions of A, then there is a fair execution $\alpha \circ \alpha'$ of A such that $trace(\alpha) = \beta$ and such that the sequence of input actions in α' is exactly β'.*

The following useful claim is adopted from Chapter 16 of [10].

Claim 1. *Suppose we have an automaton $A = \Pi_{i=1}^{k} A_i$ where A is composed of the compatible collection of automata A_i, where $i \in \{1, \cdots, k\}$. Let β be a fair trace of A then we define an irreflexive partial order $\rightarrow_\beta$ on the actions of β as follows. If π and ϕ are events in β, with π preceding ϕ, then we say ϕ depends on π, which we denote as $\pi \rightarrow_\beta \phi$, if one of the following holds:*

1. *π and ϕ are actions at the same automaton;*

2. *π is some $send(\cdot)_{j,i}$ at some A_j and ϕ is some $recv(\cdot)_{j,i}$ at A_i; and*

3. *π and ϕ are related by a chain of the relations of items 1. and 2.*

Then if γ is a sequence obtained by reordering the events in β while preserving the $\rightarrow_\beta$, then γ is also a fair trace of A.

Theorem 2.3. *Let $\{A_i\}_{i \in I}$ be a compatible collection of automata and let $A = \Pi_{i=1}^{k} A_i$. Suppose α_i is a fair execution of A_i for every $i \in I$, and suppose β is a sequence of actions in $ext(A)$ such that $\beta | A_i = trace(\alpha_i)$ for every $i \in I$. Then there is a fair execution α of A such that $\beta = trace(\alpha)$ and $\alpha_i = \alpha | A$ for every $i \in I$.*

2.2 Data type

In this section, we formally describe the data type, which we denote as $\mathcal{O}_T$, for the transaction processing systems considered in this paper. We assume there is a set of k objects, where k is some positive integer, and o_k denotes the k^{th} object. Object o_k stores a value from some non-empty domain V_k and supports two types of operations: *read* and *write* operations. A read operation on object o_k, denoted by $read(o_k)$, on completion returns the value stored in o_k. A write operation on o_k with some value v_k from V_k, denoted as $write(o_k, v_k)$, on completion, updates the value of o_k.

A WRITE transaction consists of a subset of p distinct write operations for a subset of p distinct objects, where $1 \leq p \leq k$. For example, a WRITE transaction with the set of operations $\{write(o_{i_1}, v_{i_1}), write(o_{i_2}, v_{i_2}) \cdots write(o_{i_p}, v_{i_p})\}$, means value v_{i_1} is to be written to object o_{i_1}, value v_{i_2} to object o_{i_2}, and so on. We denote such a WRITE transaction as $WRITE((o_{i_1}, v_{i_1}), (o_{i_2}, v_{i_2}), \cdots, (o_{i_p}, v_{i_p}))$.

A READ transaction consisting of a set of read operations is denoted as $READ(o_{i_1}, o_{i_2}, \cdots, o_{i_q})$, where $o_{i_1}, o_{i_2}, \cdots, o_{i_q}$ is a set of distinct objects, q is any positive integer, $1 \leq q \leq k$, which upon completion returns the values $(v_{i_1}, v_{i_2}, \cdots, v_{i_q})$, where $v_{i_j} \in V_{i_j}$ is the value returned by $read(o_{i_j})$, for any $i_j \in \{i_1, i_2, \cdots i_q\}$ and $1 \leq i_1 < i_2 < \cdots < i_q \leq k$.

Formally, we define the value type used in this paper for a k object data-type as follows:

(*i*) a tuple $(v_1, v_2, \cdots, v_k) \in \Pi_{i=1}^{k} V_i$, where V_i is a domain of values, for each $i \in \{1, \cdots, k\}$;

(*ii*) an initial value v_i^0, $v_i^0 \in V_i$ for each object o_i;

(*iii*) *invocations*: $READ(o_{i_1}, o_{i_2}, \cdots, o_{i_p})$ and $WRITE((o_{i_1}, v_{i_1}), (o_{i_2}, v_{i_2}), \cdots, (o_{i_p}, v_{i_p}))$, such that $v_{i_j} \in V_{i_j}$ for any $i_j \in \{i_1, i_2, \cdots i_p\}$ where $1 \leq i_1 < i_2 < \cdots < i_p \leq k$ and p is some integer with $1 \leq p \leq k$;

(*iv*) *responses*: a tuple $(v_1, v_2, \cdots, v_k) \in \Pi_{i=1}^{k} V_i$ and ok; and

(*v*) for any subset of p objects we define $f : invocations \times V \rightarrow responses \times V$, such that:

(a) $f(READ(o_{i_1}, o_{i_2}, \cdots, o_{i_p}), (v_1, v_2, \cdots, v_k)) = ((v_{i_1}, v_{i_2}, \cdots, v_{i_p}), (v_1, v_2, \cdots, v_k))$; and

(b) $f(WRITE((o_{i_1}, u_{i_1}), (o_{i_2}, u_{i_2}), \cdots, (o_{i_p}, u_{i_p})), (v_1, v_2, \cdots, v_k)) = (ack, (w_1, w_2, \cdots, w_k))$, where for any i, $w_i = u_i$ if $i \in \{i_1, i_2, \cdots, i_p\}$, and for all other values of i, $w_i = v_i$.

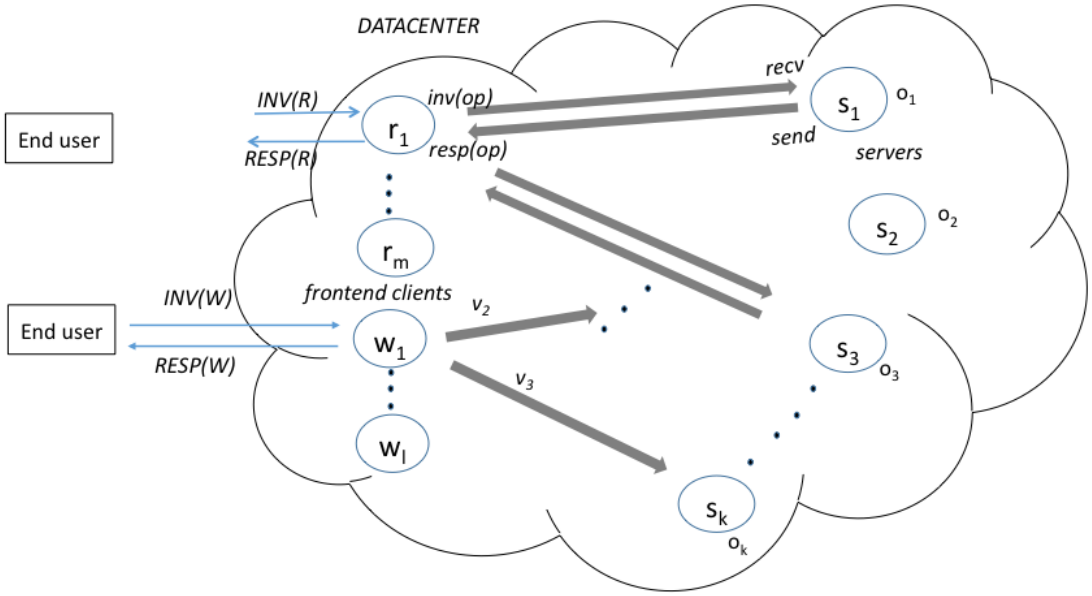

Figure 1: The architecture of a typical web service with clients and servers inside a datacenter.

2.3 System Model

In our system, we assume there are ℓ writers, for some $\ell \geq 1$; m readers, for some $m \geq 1$, and k servers for some $k \geq 1$. We denote the set of writers as $\mathcal{W}$, which essentially consists of the writer identifiers $w_1, w_2 \cdots w_\ell$. The set of readers, with ids $r_1, r_2 \cdots r_m$, is denoted as $\mathcal{R}$. The set of servers $\mathcal{S}$ consists of the server identifiers $s_1, s_2 \cdots s_k$. The servers $s_1, s_2 \cdots s_k$ manage the objects $o_1, o_2, \cdots, o_k$, respectively, in particular, server s_i is responsible for storing object o_i, for any i, $1 \leq i \leq k$.

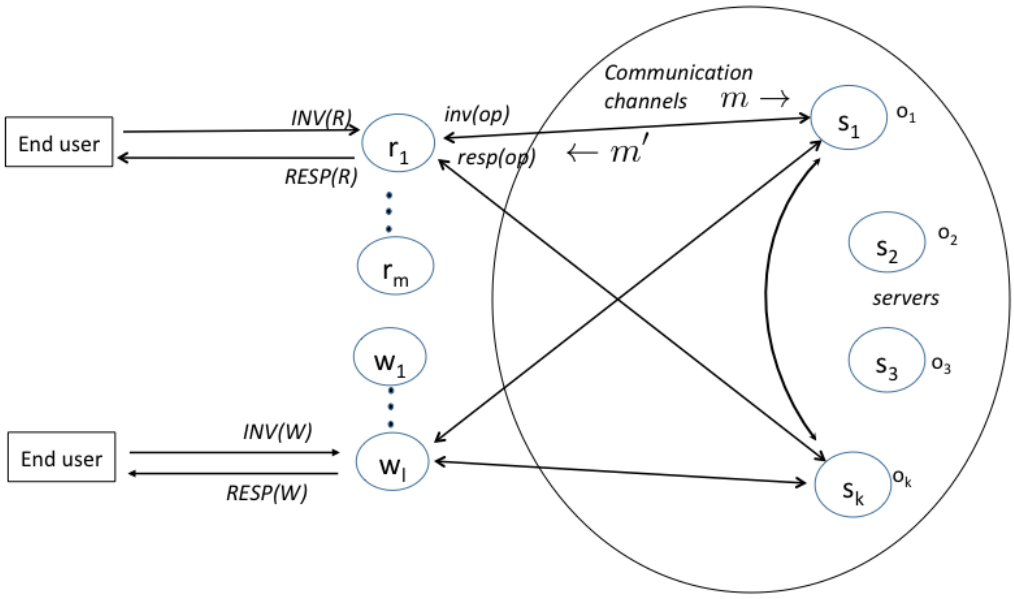

Figure 2: The architecture of a typical web service with clients, servers, and the communication channels inside a datacenter is modeled as a collection of I/O automata.

Communications between processes during the executions are point-to-point and are assumed to be reliable and asynchronous. The communications are modeled with *Channel* I/O automata and are carried out via *send* and *recv* actions at the source and destination processes. For example, in Figure 2, the reader r_1 automaton sends, via the action $send(m)$, some message m, from some alphabet $\mathcal{M}$, to server s_1 through the channel automaton $Channel_{r_1,s_1}$, and s_1 receives it though the action $recv(m)$. Similarly, the message m', $m' \in \mathcal{M}$, is transmitted from s_1 to r_1, through the channel automaton $Channel_{s_1,r_1}$, via the actions $send(m')$ at s_1, and finally received at r_1 via the action $recv(m')$.

The schematic flow of a typical, but simplified, webpage request from a service hosted in a datacenter is shown in Figure 1. An end user sends a webpage request to a front-end client r_1. Upon receiving it r_1 generates a READ transaction R, which consists of a set of read operations and is invoked via a $INV(R)$ action at the client, which is essentially input to the client from the end user. The goal of these operations is to read a subset of the data objects by contacting the respective managing servers. The operations in a transaction are carried out between the invocation and response actions of the transaction without any particular order of execution among the operations. Any operation *op* begins with an invocation action, $inv(op)$, at the relevant client, and ends with a response action $resp(op)$, also at the client. Once the front-end client completes the read operations, then r_1 synthesizes the webpage with the data retrieved by the read operations, and responds to the end user request through the response action $RESP(R)$ at the client r_1.[1] Similarly, the update request from the end user can be thought of a WRITE transaction W, consisting of a set of write

[1]In a typical production system the depth of the transactions (due to nested transactions) is more than one [1], however, we assume only transactions of depth one as in [9].

operations for individual objects, to update the set of object values, initiated via an invocation action $INV(W)$ at the client. Then the writes operations are carried out by the client, and once they complete, the transaction completes with the response action $RESP(W)$, at the client.

A transaction π is *incomplete* in an execution α when the action $INV(\pi)$ does not have the $RESP(\pi)$ in α; otherwise, we say that π is *complete* in α. In an execution, we say that a transaction (read or write) π_1 *precedes* another transaction π_2 (denote as $\pi_1 \rightarrow \pi_2$), if the response action $RESP(\pi_1)$ precedes the invocation action $INV(\pi_2)$. Two transactions are *concurrent* if neither precedes the other.

If in any execution of the system every client initiates a new transaction only after the previous transactions initiated at the same client have completed, then we say the execution is *well-formed*.

2.4 SNOW Properties

The SNOW properties for a transaction processing system can be described by requiring 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 the resulting execution seems to occur at a single machine atomically; (*ii*) *Non-Blocking Operations* (N) property, which means that the servers respond immediately, without waiting for any input from other processes, to read requests for the read operations, which belong to the READ transactions; (*iii*) *One Response Per Read* (O) property requires that any read operation consists of one-round trip of communication with a server, and also, 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. Below we describe the individual properties of the SNOW properties in more detail.

Strict serializability By *strict serializability* [7], we mean each WRITE or READ transaction is executed atomically, at some point in an execution between the invocation and response events. In other words, for each transaction, we can associate a serialization point between the invocation and response actions, such that all the read operations return values that are consistent with the ordering of the serialization points.

Non-blocking reads Consider a READ transaction initiated by a read client r and a read operation in it to read the value of an object o_i managed by some server s_i. The read operation involves r communicating a *send value* message to s_i requesting the stored value of o_i. The *non-blocking* property means that the server responds to the read request 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 of messages between r and s_i. Below we define the non-blocking nature of any server's response to a read request from a reader as follows.

Definition 1. *If α is a fair execution of some server automaton for a server s and action $recv(m^r)_{r,s}$ appears in α, where r is a reader automaton and m^r is a read value request to server s, and no other input actions follow $recv(m^r)_{r,s}$, then eventually, the action $send(v_i)_{s,r}$ occurs in α.*

One-response read operations The *one-response* property requires that each read operation, during any READ transaction, completes in one round of client to server communication and receives exactly one version of the object value from the corresponding server. A round consists of a read

request from the client initiating the read operation to the server, and subsequently, the server sending only one version of the object value in its response. The *one-round* property is imposed to reduce the number of messages between the client and the server during a read operation, and hence reducing the latency of the READ transaction. The *one-version* property is added to reduce the amount of data transmitted during the read operation and hence reduce latency of read operations, which also means lowering latency for the READ transaction.

Concurrent and conflicting writes The *conflicting*, or *concurrent 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. 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 WRITEs 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 [9] The theorem states that *no read-only transaction algorithm provides all of the SNOW properties.*

2.5 One-version property

Motivated by the impossibility result of the SNOW theorem, we propose in Section 7, an algorithm for transaction processing systems under constraints weaker than the SNOW properties. With this in mind, we introduce a weaker version of the O property, denote it as "o", where during a read operation the server returns exactly one object value but there exists a finite bound on the number of rounds of communication between the client and the server. We use the acronym SNoW to denote the S, N, o and W properties. Moreover, it is easy to realize that if an algorithm satisfies the SNoW properties then READ transactions always complete, and hence, READ's are always live. This is because the N and o properties imply that the server must immediately respond to a read request due to a read operation with exactly one object value without waiting in expectation of incoming messages, and there a fixed upper bound on the number of such rounds for a READ.

3 Impossibility of SNOW properties with two clients with restricted communication

In this section, we prove that for a two-client transaction processing system with at least two objects it is impossible to design an algorithm that satisfies the SNOW properties.We prove our result by showing a contradiction. For the model consider in this section, we assume that between every pair of client and server, or every pair of servers there are two communication channels, one in each direction; however, there are no communication channels between any pair of clients Fig 2.

Consider a system consisting of two servers, s_1 and s_2, and two clients, a reader r, which initiates only READ transactions, and a writer w, which initiates only WRITE transactions. Servers s_1 and s_2 store values for objects o_1 and o_2, respectively. The values stored in o_1 and o_2 belong to the domains V_1 and V_2, respectively; the initial values of o_1 and o_2 are v_1^0 and v_2^0, respectively. We denote the automata for servers s_1 and s_2 by s_1^A and s_2^A, respectively, and the automata for the

clients r and w by r^A and w^A, respectively. We denote by S_1 the subsystem consisting of s_1, s_2 and w, and by S_1^A the automata composed of s_1^A, s_2^A and w^A, i.e., $s_1^A \times s_2^A \times w^A$. Also, we denote by $\mathcal{A}$, which can also be interpreted as the algorithm, the automaton representing the entire system consisting of S_1^A and r^A (i.e., $s_1^A \times s_2^A \times w^A \times r^A$). We assume that between any client c, $c \in \{r, w\}$ and any server s, $s \in \{s_1, s_2\}$ there are channel automata $Channel_{c,s}$ and $Channel_{s,c}$. Here, we consider only fair executions of $\mathcal{A}$. For contradiction, we also assume that any fair execution of $\mathcal{A}$ respects the SNOW properties. Also, we assume that in an execution ϵ of $\mathcal{A}$ we can identify each transaction with an unique identifier.

Consider an execution of $\mathcal{A}$ with two transactions in it as: a WRITE transaction $W \equiv WRITE((o_1, v_1^1), (o_2, v_2^1))$ initiated by w, where $v_1^1 \neq v_1^0$ and $v_2^1 \neq v_2^0$; and a READ transaction $R \equiv READ(o_1, o_2)$, initiated by r. Let us denote by op_1^r and op_2^r the read operations $read(o_1)$ and $read(o_2)$, respectively. We assume that the patterns of sequences of invocations of transaction, the times of these invocations, and delays in local computation and message deliveries, are controlled by an omniscient entity. which we refer to as the *adversary*. The assumption of such an adversary is in accordance with the asynchronous nature of the model. This is also important in real systems, because such executions are possible to occur and thus important for making the system safe.

Notations and Definitions: We introduce the following notations and definitions in the context of an execution ϵ, of $\mathcal{A}$, with transactions R and W in it, where $j \in \{1, 2\}$:

1. $send(m_j^r)_{r,s_j}$: an output action at r^A, which sends a message m_j^r from reader r to server s_j, requesting the value for o_j;

2. $recv(m_j^r)_{r,s_j}$: an input action at s_j^A, that receives the message m_j^r, sent from r;

3. $send(v_j)_{s_j,r}$: an output action at s_j^A, that sends value v_j, for o_j, to r.

4. $recv(v_j)_{s_j,r}$: an input action at r^A, to receive a message v_j from s_j at r^A.

5. *Non-blocking fragments* $F_i(\epsilon)$, $i \in \{1, 2\}$. Suppose there is a fragment of execution in ϵ where the first action is $recv(m_i^r)_{r,s_i}$ and the last action is $send(v_i)_{s_i,r}$, both of which occur at s_i^A. Moreover, suppose there is no other input action at s_i^A in this fragment. Then we call this execution fragment a *non-blocking response fragment* for op_i^r at s_i^A. We use the notation $F_i(\epsilon)$ to denote this fragment of execution of ϵ.

6. We use the notations $R(\epsilon)$ and $W(\epsilon)$ to denote the transactions R and W, in ϵ. When the underlying execution is clear from the context we simply use R and W.

7. If the non-blocking fragment $F_1(\epsilon)$ appears in ϵ such that $recv(m_2^r)_{r,s_2}$, at s_2^A, does not occur before $F_1(\epsilon)$ completes and ϵ is of the form $\sigma_0, a_1, \cdots, a_\ell, \sigma_\ell \circ F_1(\epsilon) \circ S(\epsilon)$, where $S(\epsilon)$ is any continuation of the execution, then we denote $\sigma_0, a_1, \cdots, a_\ell, \sigma_\ell$ by $P(\epsilon)$.

8. If ϵ is of the form $\sigma_0, a_1, \cdots, a_\ell, \sigma_\ell \circ F_1(\epsilon) \circ \kappa \circ F_2(\epsilon), a_p, \sigma_p, \ldots$ or $\sigma_0, a_1, \cdots, a_\ell, \sigma_\ell \circ F_2(\epsilon) \circ \kappa \circ F_1(\epsilon) \circ S(\epsilon)$, where ℓ is a positive integer, κ is a segment of ϵ, possibly even of length zero and $S(\epsilon)$ is any suffix part of the execution then we denote $\sigma_0, a_1, \cdots, a_\ell, \sigma_\ell$ by $P(\epsilon)$. Clearly, we can write ϵ as $P(\epsilon) \circ F_1(\epsilon) \circ \kappa \circ F_2(\epsilon) \circ S(\epsilon)$ or $P(\epsilon) \circ F_2(\epsilon) \circ \kappa \circ F_1(\epsilon) \circ S(\epsilon)$.

Now we describe the set of actions corresponding to the READ transactions in a fair execution ϵ of $\mathcal{A}$. Clearly, in ϵ, the actions $inv(op_i^r)$ and $resp(op_i^r)$ appear between the actions $INV(R)$ and $RESP(R)$. The action $inv(op_i^r)$ is followed by action $send(m_i^r)_{r,s_i}$, which is for sending the read object value request to server s_i. This request m_i^r is communicated to s_i, via the channel automaton $Channel_{r,s_i}$. Automaton s_i^A eventually receives the request via the action $recv(m_i^r)_{r,s_i}$, and subsequently, responds back to r, with object value v_i, through the action $send(v_i)_{s_i,r}$. Next, value v_i, $v_i \in V_i$, communicated by the automaton $Channel_{s_i,r}$, is received at r via the action $recv(v_i)_{s_i,r}$, and finally, op_i^r completes with response action $resp(op_i^r)$ and returns v_i to r^A.

Here, we give a high-level idea of our proof, which is based on the existence of a sequence of fair executions of $\mathcal{A}$, eventually leading to a execution of $\mathcal{A}$ that contradicts the S property. First, we show the existence of an execution α of $\mathcal{A}$ where R is invoked after W completes, where the send actions $send(m_1^r)_{r,s_1}$ and $send(m_2^r)_{r,s_2}$ at the r^A, occur consecutively, in $P(\alpha)$. Then we show α can be written in the form $P(\alpha) \circ F_1(\alpha)$ (Figure 3 (a), Lemma 3.1). Based on α, we show the existence of another execution β, of $\mathcal{A}$, which can be written in the form $P(\beta) \circ F_1(\beta) \circ F_2(\beta)$, by extending α with an execution fragment $F_2(\beta)$ such that, $F_1(\beta) \overset{s_1}{\sim} F_1(\alpha)$ (Figure 3 (b); Lemma 3.2). Note that in any arbitrary extension (as a fair execution) of β, eventually R returns (v_1^1, v_2^1). Next, from β, we show the existence of an execution γ, of $\mathcal{A}$, of the form $P(\gamma) \circ F_1(\gamma) \circ F_2(\gamma)$, where the send actions $send(m_1^r)_{r,s_1}$ and $send(m_2^r)_{r,s_2}$ at r^A, occur before W is invoked (Figure 3 (c); Lemma 3.3), but the fragments $F_1(\gamma)$ and $F_2(\gamma)$ occur after $RESP(W)$, as in β. From γ we show the existence of a fair execution δ, of $\mathcal{A}$, of the form $P(\eta) \circ F_1(\eta) \circ F_2(\eta) \circ S(\eta)$, where R responds with (v_1^1, v_2^1). Finally, starting with η we create a sequence of fair executions $\delta(\equiv \eta)$, $\delta^{(1)}$, $\cdots$ $\delta^{(f)}$, of $\mathcal{A}$, where in each of them R responds with (v_1^1, v_2^1) (Figure 4 (e) and (g); Lemma 3.6). Additionally, for any $\delta^{(i)}$, the fragments $F_1(\delta^{(i)})$ and $F_2(\delta^{(i)})$ appear in the execution before $\delta^{(i-1)}$. From $\delta^{(f)}$ we show the existence of a fair execution ϕ (see Figure 4 (h)), of $\mathcal{A}$, where R completes by returning (v_1^1, v_2^1) even before W begins, which is violation of the property S.

Now, we state and prove the the relevant lemmas. The following lemma states that there is a finite execution of $\mathcal{A}$ where R begins after W completes where the two send actions $send(m_1^r)_{r,s_1}$ and $send(m_2^r)_{r,s_2}$ occur before either of the servers s_1 and s_2 receives the messages m_1^r or m_2^r from r; also, server s_1 responds to r in a non-blocking manner (execution α in Figure 3 (a)).

Lemma 3.1. *There exists a finite execution α of $\mathcal{A}$ that contains transactions $R(\alpha)$ and $W(\alpha)$ where $INV(R)$ appears after $RESP(W)$ and the following conditions hold:*

(i) The actions $send(m_1^r)_{r,s_1}$ and $send(m_2^r)_{r,s_2}$ appear consecutively in $trace(\alpha)|r^A$; and

(ii) α contains the execution fragment $F_1(\alpha)$.

Proof. Consider a finite execution fragment of $\mathcal{A}$ with a completed transaction W, where after W completes the adversary invokes R, i.e., $INV(R)$ occurs. Note that each of the read operations op_1^r and op_2^r, in R, can be invoked by the adversary at any point in the execution. Following $INV(R)$, the adversary introduces the invocation action $inv(op_1^r)$; by the O property of the read operations of $\mathcal{A}$ the action $send(m_1^r)_{r,s_1}$ eventually occurs. Next, the adversary introduces $inv(op_2^r)$ and also, delays the arrival of m_1^r until action $send(m_2^r)_{r,s_2}$ eventually occurs, which must occur in accordance with the property O of read operations. Let us call this finite execution α^0.

Next, suppose at the end of α^0 the adversary delivers the message m_1^r, which has been delayed so far, via the action $recv(m_1^r)_{r,s_1}$, at s_1^A, but it delays any other input actions at s_1^A. Note that by the N property of read operations s_1^A eventually responds with $send(v_1)_{s_1,r}$, with one value v_1 by

O property, where $v_1 = v_1^1$ by the S property, since R begins after W completes. Let us call this execution α. Note that α satisfies conditions (i) and (ii) by the design of the execution. □

The following lemma states that there is an execution β of $\mathcal{A}$ where R begins after W completes where the two send events at r^A occurs before $F_1(\beta)$, which in turn, occurs before $F_2(\beta)$ (execution β in Figure 3 (b)).

Lemma 3.2. *There exists an execution β of $\mathcal{A}$ that contains transactions R and W where $INV(R)$ appears after $RESP(W)$ and the following conditions hold:*

(i) *The actions $send(m_1^r)_{r,s_1}$ and $send(m_2^r)_{r,s_2}$ appear consecutively in $trace(\beta)|r^A$; and*

(ii) *β contains the execution fragment $F_1(\beta) \circ F_2(\beta)$.*

Proof. Consider the execution α of $\mathcal{A}$ as constructed in Lemma 3.1. At the end of the execution fragment α, the adversary delivers the previously delayed message m_2^r, which is sent via the action $send(m_2^r)_{r,s_2}$, by introducing the action $recv(m_2^r)_{r,s_2}$. The adversary then delays any other input action in $\mathcal{A}$. By the N property, server s_2^A must respond to r^A, with some value v_2, and hence the output action $send(v_2)_{s_2,r}$ must eventually occur at s_2^A. Let us call this finite execution as β. Note that β satisfies the properties (i) and (ii) in the statement of the lemma. □

The following result shows that starting with β there is an execution γ of $\mathcal{A}$ where R is initiated before W is initiated , also, the send events $send(m_1^r)_{r,s_1}$ and $send(m_2^r)_{r,s_2}$ occur before $INV(W)$ (execution Figure 3 (c)) and the messages m_1^r and m_2^r from r^A reach the servers s_1 and s_2, respectively, after the action $RESP(W)$.

Lemma 3.3. *There exists a fair execution γ of $\mathcal{A}$ with transactions R and W where the action $INV(R)$ appears before $INV(W)$ and $RESP(R)$ appears after $RESP(W)$, and the following conditions hold for γ:*

(i) *The actions $send(m_1^r)_{r,s_1}$ and $send(m_2^r)_{r,s_2}$ appear before $INV(W)$ and they appear consecutively in $trace(\gamma)|r^A$;*

(ii) *γ contains the execution fragment $F_1(\gamma) \circ F_2(\gamma)$; and*

(iii) *action $RESP(W)$ occurs before $F_1(\gamma)$.*

Proof. Consider the execution β of $\mathcal{A}$ as in Lemma 3.2. Note that β is an execution of the composed automaton $\mathcal{A}$ ($\equiv S_1^A \times r^A$). In β, the actions $send(m_1^r)_{r,s_1}$ and $send(m_2^r)_{r,s_2}$ occur at r^A; and following that, the actions $recv(m_1^r)_{r,s_1}$, $recv(m_2^r)_{r,s_2}$, $send(v_1)_{s_1,r}$ and $send(v_2)_{s_2,r}$ occur at S_1^A. Consider the executions $\alpha_r \equiv \beta|r^A$ and $\alpha_{S_1} \equiv \beta|S_1^A$. Let s_β denote $trace(\beta)$.

In s_β, $send(m_1^r)_{r,s_1}$, $send(m_2^r)_{r,s_2}$ appear after $RESP(W)$, as in $trace(\beta)$. Let s'_β be the sequence of external actions of S_2^A which we construct from s_β by moving $send(m_1^r)_{r,s_1}$, $send(m_2^r)_{r,s_2}$ before $INV(W)$, which is also an external action of $\mathcal{A}$, and leaving the rest of the actions in s_β as it is.

In β, $INV(R)$, $recv(v_1)_{s_1,r}$ and $recv(v_2)_{s_2,r}$ are the only input actions at r^A, therefore, $s'_\beta|r^A = trace(\alpha_r)$. On the other hand, $recv(m_1^r)_{r,s_1}$, $recv(m_2^r)_{r,s_2}$ are the only input actions at s_1^A , therefore, $s'_\beta|S_1^A = trace(\alpha_{S_1})$. Now, by Theorem 2.1, there exists an execution γ of $\mathcal{A}$ such that, $s'_\beta = trace(\gamma)$ and $\alpha_r = \gamma|r^A$ and $\alpha_{S_1} = \gamma|S_1^A$. Therefore, in γ, $send(m_1^r)_{r,s_1}$, $send(m_2^r)_{r,s_2}$ appear before $INV(W)$ (condition (i)) and since $s'_\beta = trace(\gamma)$ condition (ii) holds. Conditions (iii) holds trivially. □

In the following lemma we show that in any fair execution, of $\mathcal{A}$, that is an extension of either execution β or execution γ, as in the preceding lemmas, R eventually returns (v_1^1, v_2^1).

Lemma 3.4. *Let ξ be a fair execution of $\mathcal{A}$ that is an extension of the either execution β from Lemma 3.2 or execution γ from Lemma 3.3, then $R(\xi)$ responds with (v_1^1, v_2^1).*

Proof. Note that in executions β and γ, the traces $trace(\beta)|s_1^A$ is a prefix of $trace(\gamma)|s_1^A$, since β ends with $F_1(\beta)$ and γ ends with $F_2(\gamma)$ Therefore, in both β and γ, the respective $send(v_1)_{s_1,r}$ actions have the same value for their v_1's. Now, in any extended fair execution η of A, which starts with β or γ, by the properties N and O the transaction R completes; and by the property S, R returns (v_1^1, v_2^1). Therefore, $R(\xi)$ returns (v_1^1, v_2^1). □

In the following lemma, we show there exists an execution η of $\mathcal{A}$ of the form $P(\eta) \circ F_1(\eta) \circ F_2(\eta) \circ S(\eta)$ where $RESP(R)$ appears in $S(\eta)$ (Figure 3 (d)) and $R(\eta)$ returns (v_1^1, v_2^1).

Lemma 3.5. *There exists a fair execution η of $\mathcal{A}$ that contains transactions R and W where $INV(R)$ appears before $INV(W)$; $RESP(R)$ appears after $RESP(W)$ and the following conditions hold for η:*

- (i) *η can be written in the form $P(\eta) \circ F_1(\eta) \circ F_2(\eta) \circ S(\eta)$, for some $P(\eta)$ and $S(\eta)$;*
- (ii) *The actions $send(m_1^r)_{r,s_1}$ and $send(m_2^r)_{r,s_2}$ appear before $INV(W)$ and they appear consecutively in $trace(\eta)|r^A$;*
- (iii) *action $RESP(W)$ occurs before $F_1(\eta)$; and*
- (iv) *$R(\eta)$ returns (v_1^1, v_2^1).*

Proof. Let γ be a fair execution of $\mathcal{A}$, as described in Lemma 3.3. Let γ^0 be the execution fragment of γ up to the action $send(v_2)_{s_2,r}$. Now, by Theorem 2.2 (1), there exists a fair execution $\gamma^0 \circ \mu$, of $\mathcal{A}$, where μ denotes the extended portion of the execution.

Clearly, by the N and O properties, the actions $resp(op_1^r)$ and $resp(op_2^r)$ must eventually occur in $\gamma^0 \circ \mu$. Now, identify η as $\gamma^0 \circ \mu$, where $P(\eta) \circ F_1(\eta) \circ F_2(\eta)$ is γ^0, and μ is $S(\eta)$, thereby, proving condition (i).

Note the condition (ii) is satisfied by η because $RESP(W)$ appears in $P(\eta)$, therefore, the fair execution γ is equivalent to the execution fragment of $P(\eta)$ up to the event $INV(W)$, and also, γ satisfies condition (ii) as stated in Lemma 3.3.

Condition (iii) is true because $F_1(\eta)$ begins with action $recv(m_1^r)_{r,s_1}$, which occurs after $RESP(W)$. Condition (iv) is satisfied by η because η is an extension of γ and due to the result of Lemma 3.4. □

In the following theorem we prove the impossibility result for achieving the SNOW properties for the two-client system by starting with a fair execution η and creating a sequence of fair executions of $\mathcal{A}$, where each one is of the form $P(\cdot) \circ F_1(\cdot) \circ F_2(\cdot) \circ S(\cdot)$, with progressively shorter $P(\cdot)$ until one of them contradicts the property S.

Theorem 3.6. *The SNOW properties cannot be implemented in a system with two clients and two servers, where the clients do not communicate with other clients.*

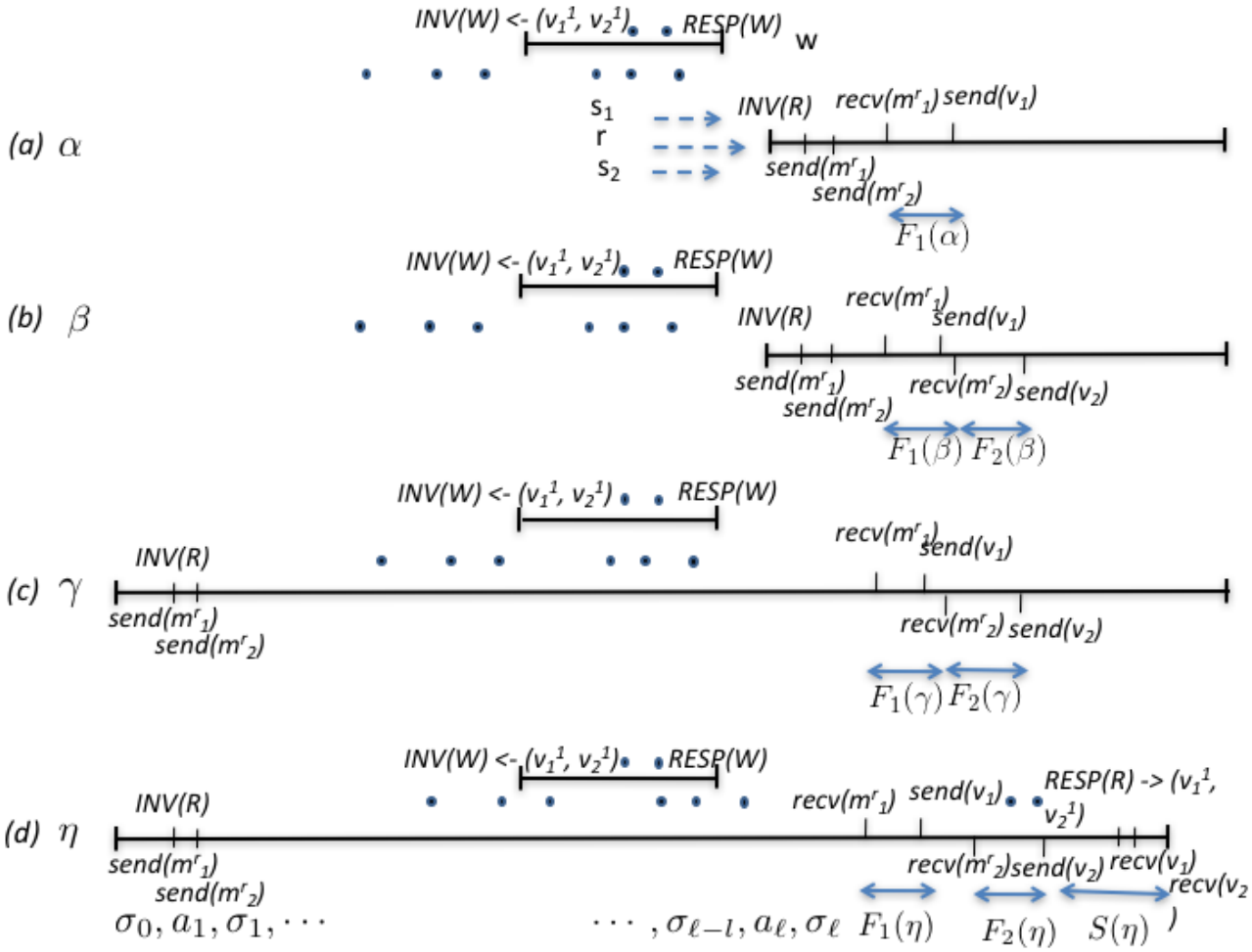

Figure 3: Schematic representation of executions α, β, γ and η, of $\mathcal{A}$ with transactions R and W. The executions evolve from left to right. The vertical marks denote external events at the reader and the server automata. The marks above the horizontal line denote external actions at s_1^A, and the marks below the line for external actions at s_2^A, and for actions r^A the mark cuts through the line. The dots stand for other external actions at the individual automata.

Proof. Consider a fair execution $\delta^{(\ell)}$ of $\mathcal{A}$ as in Lemma 3.5, and let $P(\delta^{(\ell)})$ be the execution fragment $\sigma_0, a_1, \cdots, a_\ell, \sigma_\ell$, where ℓ is some positive integer, and the σ_i's and a_i's denote states and actions, respectively. By Lemma 3.5, $RESP(R(\delta^{(\ell)}))$ returns (v_1^1, v_2^1) and $\delta^{(\ell)}$ is also of the form $\sigma_0, a_1, \cdots, a_\ell, \sigma_\ell \circ F_1(\delta^{(\ell)}) \circ F_2(\delta^{(\ell)}) \circ S(\delta^{(\ell)})$.

Now, inductively we prove the existence of a finite sequence of fair executions of $\mathcal{A}$–by proving the existence of one from the previous one–as $\delta^{(\ell)}, \delta^{(\ell-1)}, \cdots \delta^{(i)}, \delta^{(i-1)}, \cdots \delta^{(f)}$, for some positive integer f, with the following properties: (a) Each of the execution in the sequence can be written in the form $\sigma_0, a_1, \cdots, a_i, \sigma_i \circ F_1(\delta^{(i)}) \circ F_2(\delta^{(i)}) \circ S(\delta^{(i)})$ (or $P(\cdot) \circ F_1(\cdot) \circ F_2(\cdot) \circ S(\cdot)$); (b) for each i, $f \leq i < \ell$ we have $P(\delta^{(i)})$ is a prefix of $P(\delta^{(i+1)})$; and (c) $R(\delta^{(f)})$ returns (v_1^0, v_2^0) and for any i, $f < i \leq \ell$ we have $R(\delta^{(i)})$ returns (v_1^1, v_2^1). Note that there is a final execution of the form $\delta^{(f)}$ because of the initial values of v_1^0 and v_2^0, and the WRITE W.

Clearly, there exists an integer k, $f \leq k < \ell$ such that $R(\delta^{(k)})$ returns (v_1^0, v_2^0) and $R(\delta^{(k+1)})$

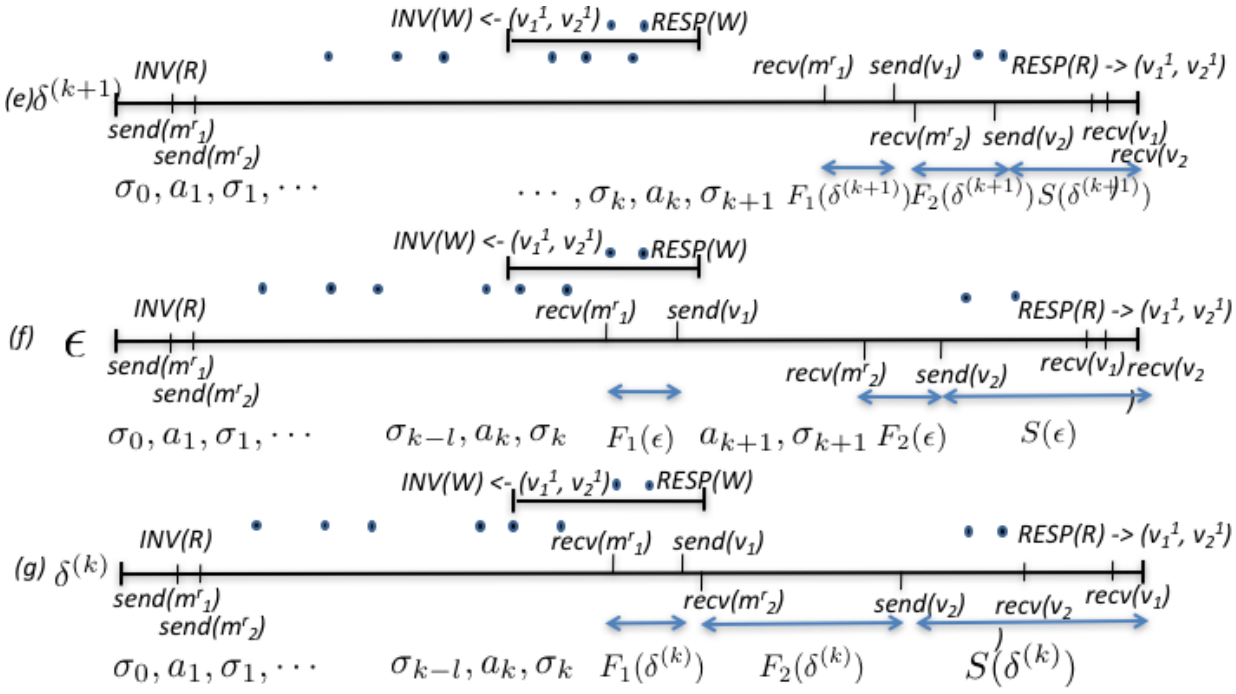

Figure 4: Figure shows, starting with execution η of $\mathcal{A}$, the construction of the progressive sequence of executions $\delta, \delta', \ldots$, of $\mathcal{A}$, of the form $P(\cdot) \circ F_1(\cdot) \circ F_2(\cdot) \circ S(\cdot)$, where we finally construct the execution ϕ of $\mathcal{A}$ to contradict the S property.

returns (v_1^1, v_2^1). Now we start with execution $\delta^{(k+1)}$ and construct an execution $\delta^{(k)}$ as described in the rest of the proof. The following argument will show that $R(\delta^{(k)})$ must also return (v_1^1, v_2^1) thereby contradicting the above.

Consider the fair execution $\delta^{(k+1)}$ of $\mathcal{A}$ which is of the form $\sigma_0, a_1, \cdots, a_{k+1}, \sigma_{k+1} \circ F_1(\delta^{(k+1)}) \circ F_2(\delta^{(k+1)}) \circ S(\delta^{(k+1)})$. The action a_{k+1} can occur at any of the automata r^A, w^A, s_1^A or s_2^A, therefore, we prove our claim by considering the following four possible cases.

Case (i) a_{k+1} occurs at w^A: The execution fragments $F_1(\delta^{(k+1)})$ and $F_2(\delta^{(k+1)})$ do not contain any input action at s_1^A and s_2^A, respectively, and also, a_{k+1} does not occur at s_1^A or s_2^A. Hence, the adversary can delay the occurrence of action a_{k+1}, at w^A, to create the finite execution $\sigma_0, a_1, \cdots, a_k, \sigma_k \circ F_1(\delta^{(k+1)}) \circ F_2(\delta^{(k+1)})$, of $\mathcal{A}$. Note by Theorem 2.2 (1), there exists a fair execution $\delta^{(k)}$, of $\mathcal{A}$, that is an extension of the above finite execution where, by the liveness property of a READ transaction, R completes in $\delta^{(k)}$. Clearly, $\delta^{(k)}$ can be written as $\sigma_0, a_1, \cdots, a_k, \sigma_k \circ F_1(\delta^{(k)}) \circ F_2(\delta^{(k)}) \circ S(\delta^{(k)})$, where $S(\delta^{(k)})$ is the tail part of the execution resulting from the extension. At s_1^A, $F_1(\delta^{(k+1)})$ is indistinguishable from $F_1(\delta^{(k)})$ i.e., $F_1(\delta^{(k+1)}) \overset{s_1}{\sim} F_1(\delta^{(k)})$. [2] Therefore, in both of the above fragments, the $send(v_1)_{s_1,r}$ action has the same object value v_1. But this means R returns v_1^1 as the value for o_1 and hence, by the property S, $R(\delta^{(k)})$ must respond with (v_1^1, v_2^1).

[2] We drop the superscript A, such as s_1^A to s_1, from the symbol above $\sim$ for formatting reasons, when necessary.

Case (ii) a_{k+1} occurs at r^A: Similar to Case (i).

Case (iii) a_{k+1} occurs at s_1^A: Observe that the execution fragments $a_{k+1}\sigma_{k+1} \circ F_1(\delta^{(k+1)})$ and $F_2(\delta^{(k+1)})$ occur at separate automata, i.e., at s_1^A and s_2^A, respectively. Observe that the execution fragments $a_{k+1}\sigma_{k+1} \circ F_1(\delta^{(k+1)})$ and $F_2(\delta^{(k+1)})$, of $\mathcal{A}$, do not contain any input actions at s_1^A and s_2^A, respectively. Therefore, by Claim 1, we can create a fair execution ϵ, of $\mathcal{A}$, which can be expressed as $\sigma_0, a_1, \cdots, a_k, \sigma_k \circ F_2(\epsilon) \circ a_{k+1}, \sigma_{k+1} \circ F_1(\epsilon) \circ S(\epsilon)$. Clearly, $F_1(\epsilon) \overset{s_1}{\sim} F_1(\delta^{(k+1)})$ and $F_2(\epsilon) \overset{s_2}{\sim} F_2(\delta^{(k+1)})$. Now, since action $send(v_1)_{s_1,r}$, in both $F_1(\epsilon)$ and $F_1(\delta^{(k+1)})$, sends the same object value v_1^1 to r. Therefore, R returns (v_1^1, v_2^1).

Now let us denote the execution fragment $\sigma_0, a_1, \cdots, a_k, \sigma_k \circ F_2(\epsilon)$ by ϵ', which is simply a finite prefix of ϵ. Now, suppose the adversary appends the $recv(m_1^r)_{r,s_1}$ to ϵ', i.e., delivers the value request message from r to s_1, and creates a finite execution ϵ'', of $\mathcal{A}$, as $\sigma_0, a_1, \cdots, a_k, \sigma_k \circ F_2(\epsilon''), recv(m_1^r)_{r,s_1}$, and delays any input action at s_2^A. Now, by Theorem 2.2 (1), there exists a fair execution ϵ''' of $\mathcal{A}$, which extends ϵ''. Clearly, $F_2(\epsilon''') \overset{s_2}{\sim} F_2(\epsilon'')$ and hence the $send(v_2)_{s_2,r}$ actions in ϵ'' and ϵ''', send the same value v_2^1 for o_2. Then by the N property action $send(v_1)_{s_1,r}$ eventually occurs and by O property v_1 is send to r^A, therefore, R completes in ϵ''', this would imply that $R(\epsilon''')$ must respond with (v_1^1, v_2^1).

Note that the execution fragment of ϵ''', between the actions $recv(m_1^r)_{r,s_1}$ and $send(v_1)_{s_1,r}$, has no input actions of s_1^A, which can be identified as $F_1(\epsilon''')$. Therefore, ϵ''' can be written as $\sigma_0, a_1, \cdots, a_k, \sigma_k \circ F_2(\epsilon''') \circ F_1(\epsilon''') \circ S(\epsilon''')$.

Next, since $F_1(\epsilon''')$ and $F_2(\epsilon''')$ contains actions of two separate automata, therefore, by Claim 1, we can create an execution prefix $\epsilon^{(iv)}$ as $\sigma_0, a_1, \cdots, a_k, \sigma_k \circ F_1(\epsilon^{(iv)}) \circ F_2(\epsilon^{(iv)})$, where $F_1(\epsilon^{(iv)})$ appears before $F_2(\epsilon^{(iv)})$. Next by using Theorem 2.2 (1) we create fair execution $\delta^{(k)}$ as an extension of $\epsilon^{(iv)}$. Clearly, it can be written as $\sigma_0, a_1, \cdots, a_k, \sigma_k \circ F_1(\delta^{(k)}) \circ F_2(\delta^{(k)}) \circ S(\delta^{(k)})$. Then by O and N properties R completes in $\delta^{(k)}$. Since $F_1(\epsilon^{(iv)}) \overset{s_1}{\sim} F_1(\epsilon''')$ this implies action $send(v_1)_{s_1,r}$ sends v_1^1 for object o_1 in $\epsilon^{(iv)}$ and since $F_1(\delta^{(k)}) \overset{s_1}{\sim} F_1(\epsilon^{(iv)})$, this implies action $send(v_1)_{s_1,r}$ sends v_1^1 in $\delta^{(k)}$, this would imply that $R(\delta^{(k)})$ returns (v_1^1, v_2^1).

Case (iv) a_{k+1} occur at s_2^A: Since a_{k+1} occurs at s_2^A and $F_1(\delta^{(k+1)})$ occurs at s_1^A, (i.e., separate automata), then by applying Claim 1 as in the previous cases we can create a new fair execution ϵ of $\mathcal{A}$ (Figure 4 (f)) as $\sigma_0, a_1, \cdots, a_k, \sigma_k \circ F_1(\epsilon) \circ a_{k+1}, \sigma_{k+1} \circ F_2(\epsilon) \circ S(\epsilon)$ such that, $F_1(\epsilon) \overset{s_1}{\sim} F_1(\delta^{(k+1)})$ where a_{k+1}, σ_{k+1} occurs after $F_1(\delta^{(k+1)})$ and $R(\epsilon)$ returns (v_1^1, v_2^1).

Now, consider the finite execution $\sigma_0, a_1, \cdots, a_k, \sigma_k \circ F_1(\epsilon)$ for $\mathcal{A}$ and suppose the adversary appends the $recv(m_2^r)_{r,s_2}$ to create a finite execution of $\mathcal{A}$ as $\sigma_0, a_1, \cdots, a_k, \sigma_k \circ F_1(\epsilon), recv(m_2^r)_{r,s_2}$. Now, by Theorem 2.2 (1), there exists a fair execution ϵ' of $\mathcal{A}$, where the adversary delays the input actions at s_2^A. By N and O properties $send(v_2)_{s_2,r}$ occurs in ϵ'. Clearly, since $F_1(\epsilon') \overset{s_1}{\sim} F_1(\epsilon)$ the $send(v_1)_{s_1,r}$ actions, in ϵ and ϵ', send the same value for o_1. Therefore, R responds with (v_1^1, v_2), where $v_2 \in V_2$.

Now, in ϵ', we identify the fragment which begins with $recv(m_2^r)_{r,s_2}$ and end with the action $send(v_2)_{s_2,r}$ as $F_2(\epsilon')$ to write ϵ' as $\sigma_0, a_1, \cdots, a_k, \sigma_k \circ F_1(\epsilon') \circ F_2(\epsilon')$. Now using Theorem 2.2 (1), we know there exists a fair execution $\delta^{(k)}$ of $\mathcal{A}$, which is an extension of ϵ'. Clearly, $\delta^{(k)}$ can be written as $\sigma_0, a_1, \cdots, a_k, \sigma_k \circ F_1(\delta^{(k)}) \circ F_2(\delta^{(k)}) \circ S(\delta^{(k)})$, where $S(\delta^{(k)})$ is the tail part of the extended fair execution. Clearly, $F_1(\delta^{(k)}) \overset{s_1}{\sim} F_1(\epsilon')$; therefore, $R(\delta^{(k)})$ returns v_1^1 for object o_1, in $\delta^{(k)}$ and this implies $R(\delta^{(k)})$ must return (v_1^1, v_2^1). □

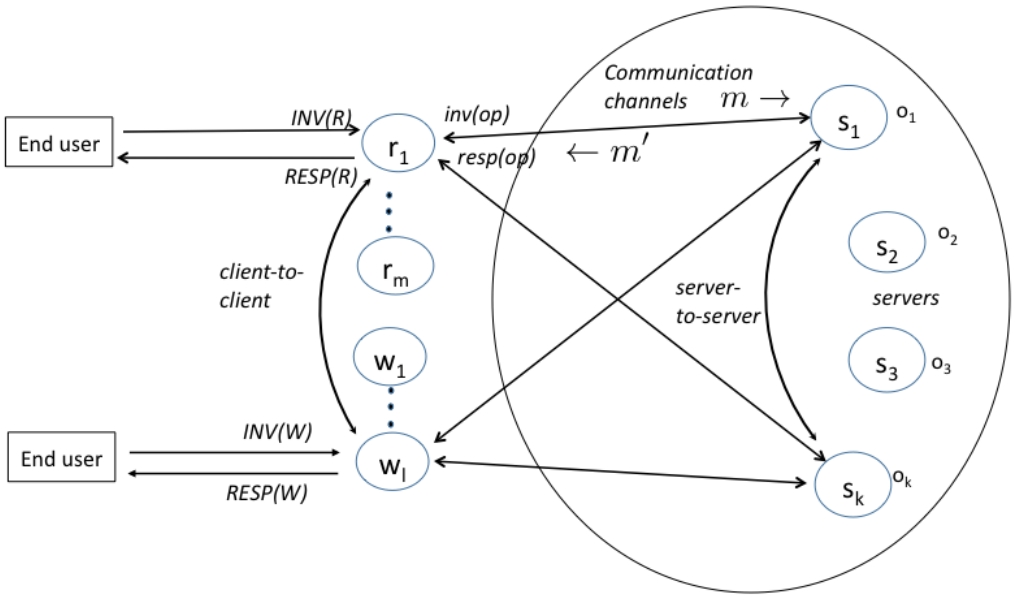

Figure 5: The architecture of a typical web service with clients, servers, and the communication channels, between every pair of processes, inside a datacenter is modeled as a collection of I/O automata. Note that, unlike the architecture in Fig. 2, in this setup there are communication channels between every pair of clients.

4 Impossibility of SNOW properties with three clients

In this section, we prove that for a three-client transaction processing system, with at least two objects, it is impossible to design an algorithm that satisfies the SNOW properties. From the proof of the SNOW Theorem [9], it is not clear whether a WRITE transaction is ever required to complete in the presence of ongoing READ transactions (Fig. 5). Moreover, the authors in [9] do not state explicitly whether the result of the SNOW Theorem holds if clients communicate with each other. In our proof, we state these points explicitly: each WRITE must complete even if there are concurrent READ transactions, and between every pair of processes, there are two communication channels, one in each direction (Fig 5). Furthermore, we assume that if a READ is invoked at a reader, then the reader can proceed to contact the servers without waiting for any incoming messages.

We prove our result by showing a contradiction. We consider a system consisting of two servers, s_1 and s_2, and three clients: two readers r_1 and r_2, which initiate only READ transactions, and a writer w, which initiates only WRITE transactions. Servers s_1 and s_2 store values for objects o_1 and o_2, respectively; the values in o_1 and o_2 belong to the domains V_1 and V_2, respectively; the initial values of o_1 and o_2 are v_1^0 and v_2^0, respectively. We denote the automata for servers s_1 and s_2 by s_1^A and s_2^A, respectively, and the automata for the clients r_1, r_2 and w by r_1^A, r_2^A and w^A, respectively. We assume that between any pair of processes p_1 and p_2, such that $p_1 \neq p_2$, there are channel automata $Channel_{p_1,p_2}$ and $Channel_{p_2,p_1}$. For the purpose of proving by contradiction, we also assume that any fair execution of $\mathcal{B}$ respects the SNOW properties. Also, we assume that in any execution of $\mathcal{B}$ we can identify each transaction with a unique identifier.

Consider an execution of $\mathcal{B}$ with three transactions: a WRITE transaction $W \equiv WRITE((o_1, v_1^1), (o_2, v_2^1))$ initiated by w, where $v_1^1 \neq v_1^0$ and $v_2^1 \neq v_2^0$; and READ transactions $R_1 \equiv READ(o_1, o_2)$ and $R_2 \equiv READ(o_1, o_2)$ initiated by r_1 and r_2, respectively. Let us denote by op_1^r and op_2^r the read operations of types $read(o_1)$ and $read(o_2)$, respectively. As in Section 3, we assume an omniscient adversary that can control the patterns of sequences of invocations of transactions, the times of these invocations, and delays in local computation and message deliveries. In the rest of the section, in order to reduce notational clutter, for any execution of $\mathcal{B}$, $\sigma_0, a_1, \cdots, a_k, \sigma_k \cdots$, where σ's and a's are states and actions, we use the notation $a_1, \cdots, a_k \cdots$ that shows only the actions while leaving out the states.

Notations and Definitions: We introduce the following notations and definitions in the context of a fair execution α, of $\mathcal{B}$, with transactions R and W in it, where $j \in \{1, 2\}$:

1. $send(m_j^r)_{r,s_j}$: an output action at r^A, which sends a message m_j^r from reader r to server s_j, requesting the value for o_j;

2. $recv(m_j^r)_{r,s_j}$: an input action at s_j^A, that receives the message m_j^r, sent from r;

3. $send(v_j)_{s_j,r}$: an output action at s_j^A, that sends value v_j, for o_j, to r.

4. $recv(v_j)_{s_j,r}$: an input action at r^A, to receive a message v_j from s_j at r^A.

5. *Non-blocking fragments* $F_j(\alpha)^{(v_j)}$, $j \in \{1, 2\}$. Suppose there is a fragment of execution in α where the first action is $recv(m_i^r)_{r,s_i}$ and the last action is $send(v_i)_{s_i,r}$, both of which occur at s_i^A. Moreover, suppose there is no other input action at s_i^A in this fragment. Then we call this execution fragment a *non-blocking response fragment* for op_i^r at s_i^A. We use the notation $F_i(\alpha)^{(v_j)}$ to denote this fragment of execution of α (Fig. 6). In the context of a READ R_i, for $i \in \{1, 2\}$, we use the notation $F_{i,j}(\alpha)^{(v_j)}$ to denote $F_j(\alpha)^{(v_j)}$.

6. Suppose READ R completes in α. Consider the execution fragment in α between the event $INV(R)$ and whichever of the events $send(m_2^r)_{r,s_2}$ and $send(m_1^r)_{r,s_1}$ that occurs later. If all the actions in this fragment correspond to r^A, then we denote this fragment as $I(\alpha)$ (Fig. 6). In case of a READ R_i, for $i \in \{1, 2\}$, we use the notation $I_i(\alpha)$ for $I(\alpha)$.

7. Suppose READs R completes in α. Consider the execution fragments in α that occurs between the later of the events $recv(v_1)_{s_1,r}$ or $recv(v_2)_{s_2,r}$, i.e., at the point in α when r receives responses from both the servers, and the event $RESP(R)$. If all the actions in this fragment occur at r^A, then we denote this fragment by $E(\alpha)^{(v_1,v_2)}$, where R returns the values (v_1, v_2) (Fig. 6). In case of a READ R_i, for $i \in \{1, 2\}$, we use the notation $E_i(\alpha)^{(v_1,v_2)}$ for $E(\alpha)^{(v_1,v_2)}$.

8. We use the notations $R(\alpha)$ and $W(\alpha)$ to denote the transactions R and W, in the context of α. When the underlying execution is clear from the context we simply use R and W.

9. For any $v_j^i \in V_j$, the superscript i corresponds to the version identifier, which uniquely identifies a version from a totally ordered set.

Any READ R initiated at a reader r, via the invocation action $INV(R)$ at r, after which the actions $send(m_1^r)_{r,s_1}$ and $send(m_2^r)_{r,s_2}$, at r, send message m_1^r to s_1 and m_2^r to s_2, respectively. Once s_1 receives $recv(m_1^r)_{r,s_1}$ then s_1 responds to r, in a non-blocking manner, with value v_1 via action

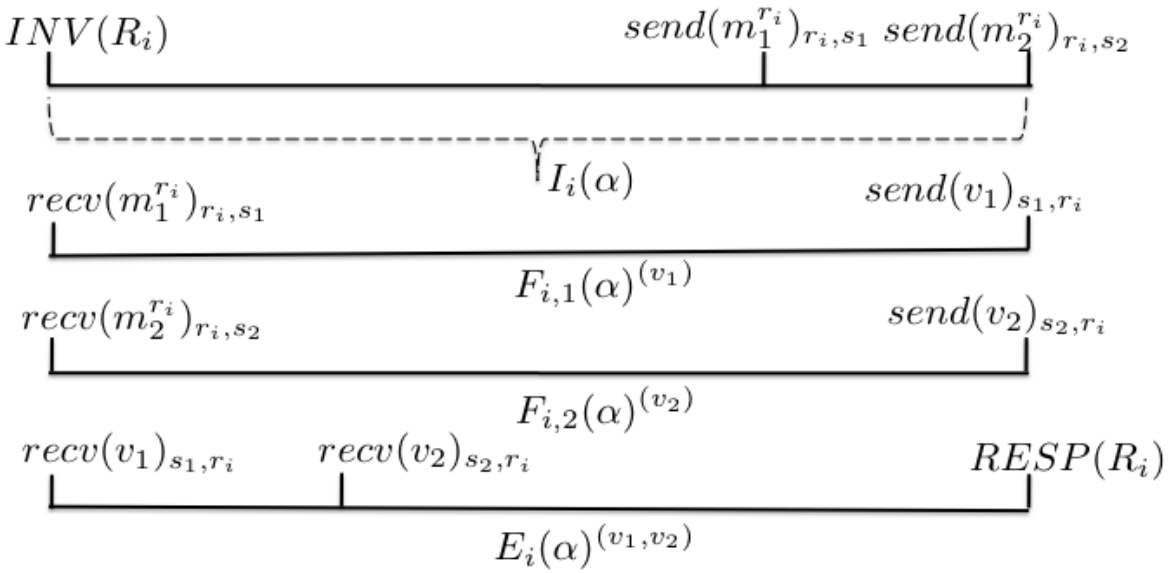

Figure 6: The figure depicts for a fair execution α of $\mathcal{B}$ the relevant actions in the execution fragments of $I_i(\alpha)$, $F_{i,1}(\alpha)^{(v_1)}$, $F_{i,2}(\alpha)^{(v_2)}$ and $E_i(\alpha)^{(v_1,v_2)}$ for any READ R_i, $i \in \{1,2\}$.

$send(v_1)_{s_1,r}$. Similarly, after s_2 receives m_2^r it responds with v_2 to r in a non-blocking manner via the action $send(v_2)_{s_2,r}$. After r receives v_1 and v_2 via actions $recv(v_1)_{s_1,r}$ and $recv(v_2)_{s_2,r}$, respectively, R completes with the response action $RESP(R)$ and returns (v_1, v_2).

From the above discussion we can state the following useful lemma, which states that in any execution of $\mathcal{B}$, if a server s_i responds with a value v_i during a READ then the READ returns v_i for object o_i and also, the pair of values (v_1^t, v_2^t) are from some version t. The result follows from the reliable channel model, where messages reach at their destinations unaltered; and by the S property the object values, for objects o_1 and o_2, returned by R are of the same version.

Lemma 4.1. *Suppose α is any execution of $\mathcal{B}$ such that a* READ *R is in α. Suppose the execution fragment $I(\alpha) \circ F_1(\alpha)^{(v_1^t)} \circ F_2(\alpha)^{(v_2^s)} \circ E(\alpha)^{(v_1^{t'}, v_2^{s'})}$ in α, corresponds to R, where $v_1^t, v_1^{t'} \in V_1$ and $v_2^s, v_2^{s'} \in V_2$, and s, s', t, t' are version identifiers then (i) $s = s'$ and $t = t'$ and (ii) $s' = t'$.*

Proof. Suppose R is invoked at reader r. Then, via the action $send(v_1^t)_{s_1,r}$, in execution fragment $F_1(\alpha)^{(v_1^t)}$, server s_1 sends the value v_1^t to r, which is received at r through the action $recv(v_1^{t'})_{s_1,r}$ in $E(\alpha)^{(v_1^{t'}, v_2^{s'})}$. By the assumptions of the reliable channel automata in our model, we have $v_1^t = v_1^{t'}$, i.e., $t = t'$. Similar argument for $F_2(\alpha)^{(v_2^s)}$ and $E(\alpha)^{(v_1^{t'}, v_2^{s'})}$ leads us to conclude $s = s'$. Next, R responds with $(v_1^{t'}, v_2^{s'})$, which implies by the S property for executions of $\mathcal{B}$ that $v_1^{t'}$ and $v_2^{s'}$ must correspond to the same version, i.e., $s' = t'$. □

Note that the above results hold even if there are any other execution fragments, that do not contain any actions at r, s_1 or s_2, in-between the I, F_i and E execution fragments.

Corollary 1. *Suppose α is any execution of $\mathcal{B}$ such that a* READ *R is in α. Suppose the execution fragment $I(\alpha) \circ X_1 \circ F_1(\alpha)^{(v_1^t)} \circ X_2 \circ F_2(\alpha)^{(v_2^s)} \circ X_3 \circ E(\alpha)^{(v_1^{t'}, v_2^{s'})}$ in α, corresponds to R, where*

$v_1^t, v_1^{t'} \in V_1$ and $v_2^s, v_2^{s'} \in V_2$, X_1, X_2, X_3 are some execution fragments that do not contain any action at r, s_1 or s_2, and s, s', t, t' are version identifiers then (i) $s = s'$ and $t = t'$ and (ii) $s' = t'$.

The following lemma states that new fair executions of $\mathcal{B}$ can be created by swapping execution fragments that have no input actions, and in which each fragment contains actions that occur only at one automaton and the automata for the two fragments are different.

Lemma 4.2 (Commuting fragments)**.** *Let α be a fair execution of $\mathcal{B}$. 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 also, none of them are input 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 a fair execution α' of $\mathcal{B}$, where the execution fragment $G_2(\alpha) \circ G_1(\alpha)$ appears in α', such that (i) 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 $G_2(\alpha') \circ G_1(\alpha')$.*

Proof. This is clear because the adversary can move the actions in G_2 to occur before G_1 at their respective automata, and these fragments do not contain any input actions, and hence the actions in one of these fragments cannot affect the actions in the other fragment. □

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

Lemma 4.3 (Indistinguishability)**.** *Let α and β be executions of $\mathcal{B}$ and let R be any* READ*. Then (i) if $F_1(\alpha) \overset{s_1}{\sim} F_1(\beta)$ then both $R(\alpha)$ and $R(\beta)$ respond with the same value v_1 for o_1; and (ii) if $F_2(\alpha) \overset{s_2}{\sim} F_2(\beta)$ then both $R(\alpha)$ and $R(\beta)$ respond with the same value v_2 for o_2;*

Proof. Suppose R is invoked at some reader r. Let $j \in \{1, 2\}$ and suppose the fragments $F_j(\alpha)$ and $F_j(\beta)$ appears in α and β respectively, where in $F_j(\alpha)$ server s_j sends $v_j \in V_j$ to r. Then $R(\alpha)$ must return v_j for object o_j by the O property of $\mathcal{B}$. Then since $F_j(\alpha) \overset{s_j}{\sim} F_j(\beta)$ then in $F_j(\beta)$ the server s_j must also send v_j to r, therefore, both $R(\alpha)$ and $R(\beta)$ must return value v_j for o_j. □

Below we show that in any finite execution of $\mathcal{B}$ where the final action is an invocation of READ R at a reader r, the adversary can always induce a fair execution of $\mathcal{B}$ where the fragments I, F_1, F_2 and E appear consecutively in that order.

Lemma 4.4. *If any finite execution of $\mathcal{B}$ ends with $INV(R)$, for a* READ *R then there exists an extension α which is a fair execution of $\mathcal{B}$ and is of the form $P(\alpha) \circ I(\alpha) \circ F_1(\alpha)^{(v_1)} \circ F_2(\alpha)^{(v_2)} \circ E(\alpha)^{(v_1, v_2)} \circ S(\alpha)$, where $P(\alpha)$ is the prefix and $S(\alpha)$ denotes the rest of the execution.*

Proof. Consider a finite execution of $\mathcal{B}$ that end with $INV(R)$, which occurs at some reader r, then the adversary induces the execution fragment $I(\alpha)$ by delaying all actions, except the internal and output actions at r^A, between the actions $INV(R)$ and the later of the actions $send(m_1^r)_{r,s_1}$ and $send(m_2^r)_{r,s_2}$. Next, the adversary delivers m_1^r at s_1 (via the action $recv(m_1^r)_{r,s_1}$) and delays all actions, other than internal and output actions at s_1^A, until s_1 responds with v_1, via $send(v_1)_{s_1,r}$; we identify this execution fragment as $F_1(\alpha)^{(v_1)}$. Subsequently, in a similar manner, the adversary delivers the message m_2^r and delays appropriate actions to induce the execution fragment $F_2(\alpha)^{(v_2)}$. Finally, the adversary delivers the values v_1 and v_2 to r (via the events

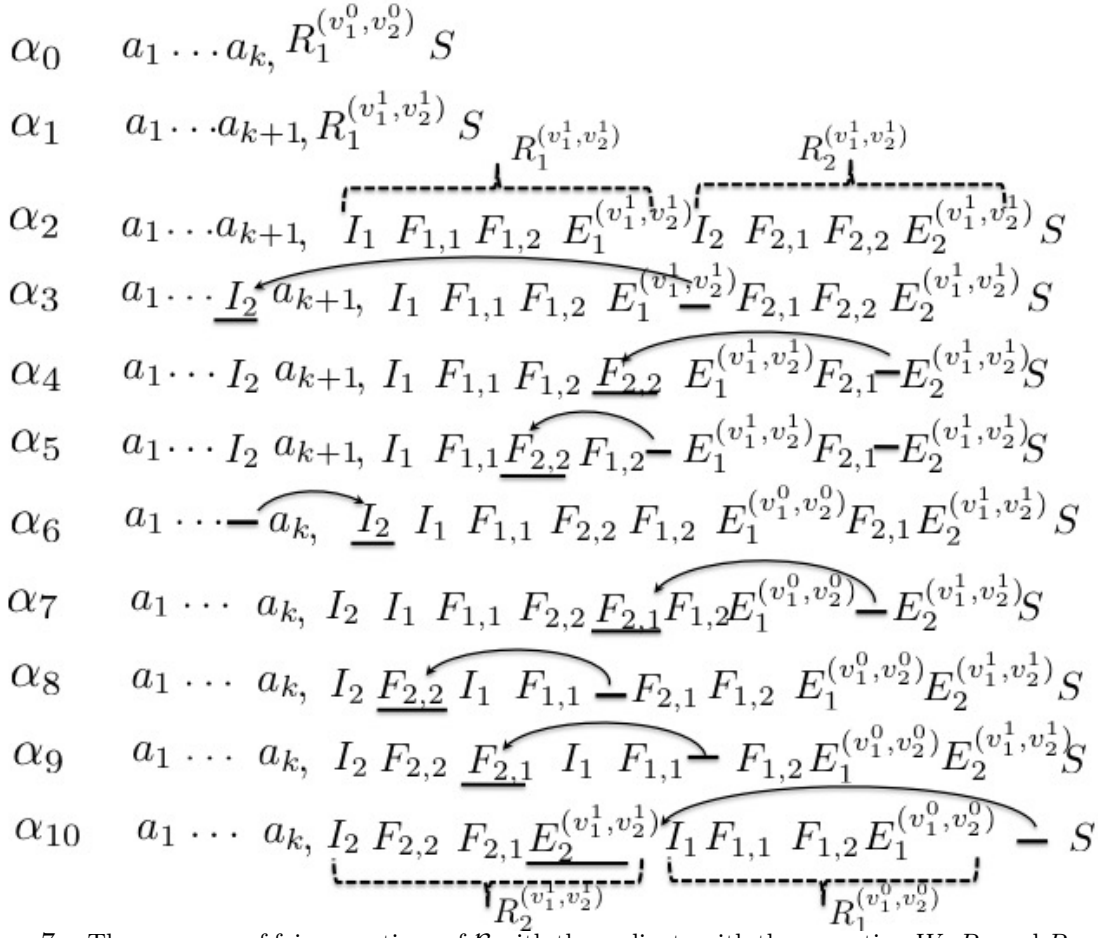

Figure 7: The sequence of fair executions of $\mathcal{B}$ with three clients with the operation W, R_1 and R_2 eventually leading to the execution $\alpha_1 0$ that contradicts the S property. The directed arcs depicts the transposition of execution fragments from the previous execution in the sequence.

$recv(v_1)_{s_1,r}$ and $recv(v_2)_{s_2,r}$), and delays all actions at other automata until R completes with action $RESP(R)$ by returning (v_1, v_2). As a result, we arrive at a fair execution of $\mathcal{B}$ of the form $I(\alpha) \circ F_1(\alpha)^{(v_1)} \circ F_2(\alpha)^{(v_2)} \circ E(\alpha)^{(v_1,v_2)} \circ S(\alpha)$. □

The high-level view of our proof strategy is to create a fair execution α of $\mathcal{B}$ that contradicts the S property. We begin with a fair execution of $\mathcal{B}$ that contains READs R_1 and R_2, and WRITE W, where R_1 begins after W completes, and R_2 begins after R_1 completes. Clearly, by the S property both R_1 and R_2 return (v_1^1, v_2^1). Then we successively create a sequence of fair executions of $\mathcal{B}$ (Fig. 7), where we interchange the fragments until we finally reach an execution of where R_2 completes before R_1 begins, but R_2 returns (v_1^1, v_2^1) and R_1 returns (v_1^0, v_2^0) which contradicts the S property.

The following lemma show that in an execution of $\mathcal{B}$ with a WRITE W and a READ R_1, there exists a point in the execution such that if R_1 is invoked before that point then R_1 returns (v_1^0, v_2^0) and if invoked after that point then R_1 returns (v_1^1, v_2^1).

Lemma 4.5 (Existence of α_0 and α_1)**.** *There exist fair executions α_0 and α_1 of $\mathcal{B}$ that contain transactions W and R_1 with the following properties:*

(*i*) α_0 *can be written as* $a_1, \cdots, a_k \circ R_1(\alpha_0)^{(v_1^0, v_2^0)} \circ S(\alpha_0)$ *;*

(*ii*) α_1 *can be written as* $a_1, \cdots, a_{k+1} \circ R_1(\alpha_1)^{(v_1^1, v_2^1)} \circ S(\alpha_1)$*; and*

(*iii*) a_{k+1} *in* α_1 *occurs at* r_1^A*,*

where k *is some positive integer and* $a_1, \cdots, a_k$ *is a prefix of* $a_1, \cdots, a_{k+1}$*.*

Proof. Now we describe the construction of a sequence $\{\gamma_k\}_{k=0}^{\infty}$ of finite executions of $\mathcal{B}$ such that each γ_k contains W and R_1. Let us consider a fair execution α of $\mathcal{B}$ that contains W. Suppose k is any positive integer and suppose R_1 is invoked at r_1^A after the execution fragment $a_1, \cdots, a_{k+1}$, a prefix of α. After the action $INV(R)$, the adversary schedules only internal and external actions at r_1^A until both the events $send(m_1^{r_1})_{r_1,s_1}$ and $send(m_2^{r_1})_{r_1,s_2}$ occur, thereby creating an execution fragment of the form $a_1, \cdots, a_{k+1} \circ I_1(\alpha)$. Let us denote $a_1, \cdots, a_{k+1}$ by P_{k+1}.

Following this, the adversary delivers the messages $m_1^{r_1}$ at s_1, and delays all actions at other automata and also any input action at s_1, until s_1 sends v_1 to r_1, therefore, inducing the execution fragment $P_{k+1} \circ I_{1,1}(\alpha) \circ F_{1,1}(\alpha)$ of $\mathcal{B}$. Next, the adversary delivers $m_2^{r_1}$ at s_2 and delays all actions at other automata and input actions at s_2, until s_2 sends v_2 to r_1. Then the adversary delivers v_1 and v_2 at r_1 but it delays actions at other automata and any other input action at r_1, until $RESP(R_1)$ occurs. Up to this point this is an execution fragment of $\mathcal{B}$, which can be written as $P_{k+1} \circ I_1(\alpha) \circ F_{1,1}(\alpha)^{(v_1)} \circ F_{1,2}(\alpha)^{(v_2)} \circ E_1(\alpha)^{(v_1,v_2)}$, where R_1 responds with (v_1, v_2) such that $(v_1, v_2) \in \{(v_1^0, v_2^0), (v_1^1, v_2^1)\}$. We denote this finite execution prefix as γ_k. Therefore, there exists the sequence of such finite executions $\{\gamma_k\}_{k=0}^{\infty}$.

In γ_0, R_1 precedes W and therefore, by the S property, in γ_0, R_1 must respond with (v_1^0, v_2^0). On the other hand, if k is large enough such that a_k occurs in α after the completion of W then by the S property, in γ_{k+1}, R_1 must return (v_1^1, v_2^1). Therefore, there exists a minimum k where in γ_k READ R_1 returns (v_1^0, v_2^0) and in γ_{k+1}, R_1 returns (v_1^1, v_2^1). We identify γ_k as α_0 and γ_{k+1} as α_1 as in claims (*i*) and (*ii*), respectively.

Now, we show (*iii*) by eliminating the possibility of a_{k+1} occurring at s_1^A, s_2^A, w^A or r_2^A by showing contradictions. Our proof is based on the following argument. The S property implies that R_1 must respond with values for o_1 and o_2 corresponding to the same version, which implies that s_1 and s_2 must send values of the same version. Observe that R_1, in α_0 and α_1, returns values for versions 0 (i.e., (v_1^0, v_2^0)) and 1 (i.e., (v_1^1, v_2^1)), respectively, but the prefixes P_k and P_{k+1} differ from one another only by action a_{k+1}. But just one action at any of s_1, s_2, r_2 or w is not enough for s_1 and s_2 to coordinate to return values of the same version. Therefore, a_{k+1} must occur in r_1, which can possibly synchronize by sending some information in m_1 and m_2 to s_1 and s_2, respectively.

Case a_{k+1} *occurs at* s_1^A*:* Consider the prefix of execution α_0 up to the action a_k. Suppose the adversary invokes R_1 immediately after action a_k, i.e., via $INV(R_1)$. By Lemma 4.4 there exists fair execution α' of $\mathcal{B}$ that contains an execution fragment of the form $P_k \circ I_1(\alpha') \circ F_{1,1}(\alpha')^{(v_1)} \circ F_{1,2}(\alpha')^{(v_2)} \circ E(\alpha')^{(v_1,v_2)}$. Note that in both α' and α_1 we can have $I_1(\alpha_1) \overset{r_1}{\sim} I_1(\alpha')$ and $F_{1,2}(\alpha_1) \overset{s_2}{\sim} F_{1,2}(\alpha')$, this is because in both executions the actions of I_1 occur entirely at r_1 and those of $F_{1,2}$ entirely at s_2. As a result, $F_{1,2}(\alpha')$ must send the same value v_2^1 for o_2 to r_1 as in $F_{1,2}(\alpha_1)$. Then in α', by Lemma 4.3, $R_1(\alpha')$ returns v_2^1 for object o_2, and by the S property, $R_1(\alpha')$ returns (v_1^1, v_2^1). But this contradicts the definition of k, as the minimum value of k such that R_1 responds with (v_1^0, v_2^0).

Case a_{k+1} *occurs at* s_2^A*:* A contradiction can be shown by following a line of reasoning similar to the preceding case.

Case a_{k+1} occurs at w^A: This can be argued in a similar manner as the previous case and a bit easier because we have $F_{1,1}(\alpha_1) \overset{s_1}{\sim} F_{1,1}(\alpha')$, and also, $F_{1,2}(\alpha_1) \overset{s_2}{\sim} F_{1,2}(\alpha')$.

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

From the above, we conclude that in α_1, action a_{k+1} must occur at r_1^A. □

Additional notation: In the remainder of the section, for the execution fragments $I_i(\alpha)$, $F_{i,1}(\alpha)^{(v_1)}$, $F_{i,2}(\alpha)^{(v_2)}$, $E_i(\alpha)^{(v_1,v_2)}$ and $S(\alpha)$, for $i \in \{1,2\}$ we use the notations I_i, $F_{i,1}^{(v_1)}$, $F_{i,2}^{(v_2)}$, $E_i^{(v_1,v_2)}$ and S, respectively, suppressing the explicit reference to the execution. With regard to any READ R_i if it has an execution fragment of the form $I_i(\alpha) \circ F_{i,1}(\alpha)^{(v_1)} \circ F_{i,2}(\alpha)^{(v_2)} \circ E_i(\alpha)^{(v_1,v_2)}$ we denote it as $R_i^{(v_1,v_2)}$. Also, wherever the returned values in the fragments are not known, clear from the context or irrelevant we omit them. In the rest of the section, we fix α_0 and α_1, and the value of k; we denote the execution fragments $a_1, \cdots, a_k$ and $a_1, \cdots, a_{k+1}$ as P_k and P_{k+1}, respectively, which are the same irrespective of the execution they appear in.

The following lemma states the existence of an execution of $\mathcal{B}$ where, following a WRITE, there are two consecutive READs and both return the object values updated by the WRITE.

Lemma 4.6 (Existence of α_2)**.** *There exists fair execution α_2 of $\mathcal{B}$ that contains transactions W, R_1 and R_2 and can be written in the form $P_{k+1} \circ R_1^{(v_1^1,v_2^1)} \circ R_2^{(v_1^1,v_2^1)} \circ S$, where both R_1 and R_2 return (v_1^1, v_2^1).*

Proof. We can construct a fair execution α_2 of $\mathcal{B}$ as follows. Consider the prefix $a_1, \cdots, a_{k+1} \circ R_1(\alpha_1)^{(v_1^1,v_2^1)}$ of the execution α_1, from Lemma 4.5. At the end of this prefix, the adversary invokes R_2. Now, by Lemma 4.4, due to $INV(R_2)$ there is an extension of the prefix of the form $a_1, \cdots, a_{k+1} \circ R_1(\alpha_1)^{(v_1^1,v_2^1)} \circ I(\alpha) \circ F_1(\alpha)^{(v_1)} \circ F_2(\alpha)^{(v_2)} \circ E(\alpha)^{(v_1,v_2)}$. By the S property, we have $v_1 = v_1^1$ and $v_2 = v_2^1$. Therefore, α_2 (Fig. 7) can be written in the form $P_{k+1} \circ R_1^{(v_1^1,v_2^1)} \circ R_2^{(v_1^1,v_2^1)} \circ S$, where S is the rest of the execution. □

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

Lemma 4.7 (Existence of α_3)**.** *There exists fair execution α_3 of $\mathcal{B}$ that contains transactions W, R_1 and R_2, and can be written in the form $P_k \circ I_2 \circ a_{k+1} \circ R_1^{(v_1^1,v_2^1)} \circ F_{2,1} \circ F_{2,2} \circ E_2 \circ S$, where both R_1 and R_2 return (v_1^1, v_2^1).*

Proof. Consider the execution α_2 as in Lemma 4.6. In the execution fragment $I_1 \circ F_{1,1}^{(v_1^1)} \circ F_{1,2}^{(v_2^1)} \circ E_1^{(v_1^1,v_2^1)}$ in α_2, none of the actions occur at r_2 and by Lemma 4.5, a_{k+1} occurs at r_1^A, also the actions in I_2 occur only at r_2. Starting with α_2, and by repeatedly using Lemma 4.2, we create a sequence of four fair executions of $\mathcal{B}$ by repeatedly swapping I_2 with the execution fragments $E_1^{(v_1^1,v_2^1)}$, $F_{1,2}^{(v_2^1)}$, $F_{1,1}^{(v_1^1)}$ and I_1, which appears in $I_1 \circ F_{1,1}^{(v_1^1)} \circ F_{1,2}^{(v_2^1)} \circ E_1^{(v_1^1,v_2^1)} \circ I_2$, where the following sequence of execution fragments $I_1 \circ F_{1,1}^{(v_1^1)} \circ F_{1,2}^{(v_2^1)} \circ I_2 \circ E_1^{(v_1^1,v_2^1)}$ (by commuting I_2 and $E_1^{(v_1^1,v_2^1)}$); $I_1 \circ F_{1,1}^{(v_1^1)} \circ I_2 \circ F_{1,2}^{(v_2^1)} \circ E_1^{(v_1^1,v_2^1)}$ (by commuting I_2 and $F_{1,2}^{(v_2^1)}$); $I_1 \circ I_2 \circ F_{1,1}^{(v_1^1)} \circ F_{1,2}^{(v_2^1)} \circ E_1^{(v_1^1,v_2^1)}$ (by commuting I_2 and $F_{1,1}^{(v_1^1)}$) appear. Finally, we have an execution α' of the form $P_{k+1} \circ I_2 \circ R_1^{(v_1^1,v_2^1)} \circ F_{2,1}^{(v_1^1)} \circ F_{2,2}^{(v_2^1)} \circ E_2^{(v_1^1,v_2^1)} \circ S$ (by commuting I_2 and I_1) Next, from α', by using Lemma 4.2 and swapping a_{k+1} with I_2 we show the existence of a fair execution α_3 as stated in the lemma. □

In the following lemma, we show that we can create a fair execution α_4, of $\mathcal{B}$, where $F_{2,2}$ occurs immediately before $E_1^{(v_1^1,v_2^1)}$, while R_1 and R_2 both return (v_1^1, v_2^1).

Lemma 4.8 (Existence of α_4)**.** *There exists fair execution α_4 of $\mathcal{B}$ that contains transactions W, R_1 and R_2 and can be written in the form $P_k \circ I_2 \circ a_{k+1} \circ I_1 \circ F_{1,1} \circ F_{1,2} \circ F_{2,2} \circ E_1 \circ F_{2,1} \circ E_2 \circ S$, where both R_1 and R_2 return (v_1^1, v_2^1).*

Proof. We start with an execution α_3, as in Lemma 4.7, and apply Lemma 4.2 twice.

First, by Lemma 4.2, we know there exists a fair execution α' of $\mathcal{B}$ where $F_{2,1}$ (identify as G_1) and $F_{2,2}$ (identify as G_2) are interchanged since actions of $F_{2,1}$ occurs solely at s_1 and those of $F_{2,2}$ at s_2, and $F_{2,1}$ and $F_{2,2}$ return v_1^1 and v_2^1, respectively, to r_2.

Next, by Lemma 4.2 there is fair execution of $\mathcal{B}$, say α_4 where the fragments E_1 (identify as G_1) and $F_{2,2}$ (identify as G_2) are interchanged, with respect to α', because the actions in E_1 occur at r_1 and those of $F_{2,2}$ at s_2. Furthermore, α_4 can be written in the form $P_k \circ I_2 \circ a_{k+1} \circ I_1 \circ F_{1,1}^{(v_1^1)} \circ F_{1,2}^{(v_2^1)} \circ F_{2,2}^{(v_2^1)} \circ E_1^{(v_1^1,v_2^1)} \circ F_{2,1}^{(v_1^1)} \circ E_2^{(v_1^1,v_2^1)} \circ S$. □

Next, we create a new fair execution α_5 of $\mathcal{B}$ where $F_{2,2}$ occurs before $F_{1,2}$. However, because the actions in both of these execution fragments occur at the same automaton care has to be taken to swap these fragments compared to α_4.

Lemma 4.9 (Existence of α_5)**.** *There exists fair execution α_4 of $\mathcal{B}$ that contains transactions W, R_1 and R_2 and can be written in the form $P_k \circ I_2 \circ a_{k+1} \circ I_1 \circ F_{1,1} \circ F_{2,2} \circ F_{1,2} \circ E_1 \circ F_{2,1} \circ E_2 \circ S$, where both R_1 and R_2 return (v_1^1, v_2^1).*

Proof. In α_4, all actions in $F_{1,2}$ and $F_{2,2}$ occur at s_2. Consider the prefix of α_4 that ends with $F_{1,1}$. We extend this prefix as follows. In this prefix, the actions $send(m_2^{r_2})_{r_2,s_2}$ and $send(m_2^{r_1})_{r_1,s_2}$ do not have their corresponding *recv* actions. Suppose the adversary delivers $m_2^{r_2}$ at s_2 (via the action $recv(m_2^{r_2})_{r_2,s_2}$) and delays all actions, other than internal and output actions at s_2^A, until s_2 responds with v_2, via action $send(v_2)_{s_2,r_2}$. This extended execution fragment is of the form $F_{2,2}$. Similarly, the adversary further extends the execution by placing the action $recv(m_2^{r_1})_{r_1,s_2}$ at s_2 and create the execution fragment of form $F_{1,2}$. Note that, so far, the actions due to the above extensions are entirely at s_2. Suppose the adversary makes the execution fragments E_1 next, by delivering values sent during $F_{1,1}$ and $F_{1,2}$ via the actions $recv(v_1)_{s_1,r_1}$ and $recv(v_2)_{s_2,r_1}$, respectively, at r_1. Then $F_{2,1}$ appear next, such that this fragment contains exactly the same sequence of actions as in the corresponding execution fragment in α_4. This is possible because they are not influenced by any output action in $F_{2,2}$ or $F_{1,2}$. Suppose the adversary places the execution fragment E_2 next. Let us denote the fair execution that is a extension of this finite execution so far as α_5, which is of the form $P_k \circ I_2 \circ a_{k+1} \circ I_1 \circ F_{1,1} \circ F_{2,2} \circ F_{1,2} \circ E_1 \circ F_{2,1} \circ E_2 \circ S$. Now we need to argue about the values returned by the reads.

Note that the execution fragment $F_{1,1}(\alpha_4)$ in both α_4 and α_5 is the same, therefore, $F_{1,1}(\alpha_4) \overset{s_1}{\sim} F_{1,1}(\alpha_5)$. Hence as in α_4, s_1 returns v_1^1 in the execution fragment $F_{1,1}$ in α_5. Next by Lemma 4.3 for R_1, s_2 returns v_2^1 in $F_{1,2}$ and hence by the S property, $R_1(\alpha_5)$ returns (v_1^1, v_2^1), i.e., that r_1 returns the new version of object values. Therefore, $F_{1,1}(\alpha_4)$, $F_{1,2}$ and E_1 are of the form $F_{1,1}^{(v_1^1)}$, $F_{1,2}^{(v_2^1)}$ and $E_1^{(v_1^1,v_2^1)}$, respectively.

Note that by construction of α_5 above, the execution fragment $F_{2,1}$ in both α_4 and α_5 is the same, therefore, $F_{2,1}(\alpha_4) \overset{s_1}{\sim} F_{2,1}(\alpha_5)$. Hence as in α_4, s_1 returns v_1^1 in the execution fragment

$F_{2,1}(\alpha_5)$ in α_5, i.e., of the form $F_{2,1}(\alpha_5)^{(v_1^1)}$. Since s_1 returns v_1^1 in $F_{2,1}$ in α_5, by Lemma 4.3 and the S property, R_2 returns (v_1^1, v_2^1) and hence E_2 is of the form $E_2^{(v_1^1, v_2^1)}$.

From the above argument we know that α_5 is of the form $P_k \circ I_2 \circ a_{k+1} \circ I_1 \circ F_{1,1}^{(v_1^1)} \circ F_{2,2}^{(v_2^1)} \circ F_{1,2}^{(v_2^1)} \circ E_1^{(v_1^1, v_2^1)} \circ F_{2,1}^{(v_1^1)} \circ E_2^{(v_1^1, v_2^1)} \circ S$. □

In the next lemma, we show the existence of a fair execution of $\mathcal{B}$ where R_1 returns (v_1^0, v_2^0) and I_2 occurs immediately after a_k and R_2 responds with (v_1^1, v_2^1). In fact, this is the most important of the sequence of fair executions of $\mathcal{B}$ (Fig. 7) because later with this we prove in Theorem 4.15 the existence of a fair execution of $\mathcal{B}$ where a READ returns object-values of an earlier version compared to a previous READ that is not concurrent with it.

Lemma 4.10 (Existence of α_6)**.** *There exists fair execution α_6 of $\mathcal{B}$ that contains transactions W, R_1 and R_2 and can be written in the form $P_k \circ I_2 \circ I_1 \circ F_{1,1} \circ F_{2,2} \circ F_{1,2} \circ E_1 \circ F_{2,1} \circ E_2 \circ S$, where R_1 returns (v_1^0, v_2^0) and R_2 returns (v_1^1, v_2^1).*

Proof. The crucial part of this proof is to carefully use the result of Lemma 4.5 so that R_1 returns (v_1^0, v_2^0), instead of (v_1^1, v_2^1). Note that by the construction of α_5, as in Lemma 4.9, the same prefix P_k appears in the fair executions α_5, and the executions α_0 and α_1 as in Lemma 4.5, where k is defined as in Lemma 4.5.

Note that by Lemma 4.5, action a_{k+1} occurs at r_1^A. In α_5, in the execution fragment $a_{k+1} \circ I_1 \circ F_{1,1}^{(v_1^1)} \circ F_{2,2}^{(v_2^1)}$, the actions in execution fragment $a_{k+1} \circ I_1$ occur at r_1^A; actions in $F_{1,1}^{(v_1^1)}$ occur at s_1^A; and actions in $F_{2,2}^{(v_2^1)}$ occur at s_2^A. Now consider the prefix of execution α_4 ending with I_2 and suppose the adversary invokes R_1 immediately after I_2 (instead of after a_{k+1}) and extends by the execution fragment $I_1 \circ F_{1,1} \circ F_{2,2}$ to create a new finite execution ϵ, which is of the form $P_k \circ I_2 \circ I_1 \circ F_{1,1} \circ F_{2,2}$. Note that as a result, a_{k+1} may not be in ϵ because we are introducing changes before a_{k+1} can occur.

Note that if in the prefix $P_k \circ I_2(\epsilon) \circ I_1(\epsilon) \circ F_{1,1}(\epsilon) \circ F_{2,2}(\epsilon)$ of ϵ we ignore the actions in $I_2(\epsilon)$ then the remaining execution is the same as the prefix $P_k \circ I_1(\alpha_0) \circ F_{1,1}(\alpha_0) \circ F_{2,2}(\alpha_0)$ of α_0 in Lemma 4.5. Here we explicitly use the notations ϵ and α_0 to avoid confusion. Since the actions in $I_2(\epsilon)$ have no influence in the actions on $I_1(\epsilon) \circ F_{1,1}(\epsilon) \circ F_{2,2}(\epsilon)$, therefore, we have $F_{1,1}(\epsilon) \overset{s_1}{\sim} F_{1,1}(\alpha_0)$, and hence by Lemma 4.1 $F_{1,1}(\epsilon)$ returns v_1^0 as in $F_{1,1}(\alpha)$, i.e., in $F_{1,1}$, s_1 returns v_1^0. Now by Lemma 4.3 we conclude that for any extension of ϵ, say γ, READ $R_1(\gamma)$ returns v_1^0 for object o_1 and by the S property $R_1(\gamma)$ returns (v_1^0, v_2^0). Also, since $F_{2,2}(\alpha_5) \overset{s_2}{\sim} F_{2,2}(\epsilon) \overset{s_2}{\sim} F_{2,2}(\gamma)$ by Lemma 4.3 and the S property, $R_2(\gamma)$ must return (v_1^1, v_2^1). Therefore, γ has an extension to a fair execution α_6 (Fig. 7) which is of the form $P_k \circ I_2 \circ I_1 \circ F_{1,1}^{(v_1^0)} \circ F_{2,2}^{(v_2^1)} \circ F_{1,2}^{(v_2^0)} \circ E_1^{(v_1^0, v_2^0)} \circ F_{2,1}^{(v_1^1)} \circ E_2^{(v_1^1, v_2^1)} \circ S$ as in the statement of the lemma. □

In the following lemma, starting from α_6 in Lemma 4.10 we create a fair execution α_7 for $\mathcal{B}$ where $F_{2,1}$ appears before $F_{1,2} \circ E_1$, where R_1 returns (v_1^0, v_2^0) and R_2 returns (v_1^1, v_2^1). At high level, we will be working on moving the execution fragments of R_2 forward, a little at a time, until finally we have R_2 finishing before R_1 starts. This simply uses commutativity since the actions in the swapped execution fragments occurs at different automata.

Lemma 4.11 (Existence of α_7)**.** *There exists fair execution α_7 of $\mathcal{B}$ that contains transactions W, R_1 and R_2, and can be written in the form $P_k \circ I_2 \circ I_1 \circ F_{1,1} \circ F_{2,2} \circ F_{2,1} \circ F_{1,2} \circ E_1 \circ E_2 \circ S$ where R_1 returns (v_1^0, v_2^0) and R_2 returns (v_1^1, v_2^1).*

Proof. This result is proved by applying the result of Lemma 4.2 to the fair execution created in Lemma 4.10. Suppose, α_6 (Fig. 7) is a fair execution as in Lemma 4.10, where in the execution fragment $E_1^{(v_1^0,v_2^0)} \circ F_{2,1}$ we identify $E_1^{(v_1^0,v_2^0)}$ as G_1 and $F_{1,2}^{(v_2^0)}$ as G_2. The actions of G_1 and G_2 occur at two distinct automata, therefore, we can use the result of Lemma 4.2, to argue that there exists a fair execution α' of $\mathcal{B}$ that contains the execution fragment $F_{2,1} \circ E_1^{(v_1^0,v_2^0)}$, and α_6 and α' are identical in the prefixes and suffixes corresponding to G_1 and G_2.

Now, α' contains $F_{1,2} \circ F_{2,1}$, where the actions in $F_{1,2}$ (identified as G_1) and $F_{2,1}$ (identify as G_2) occur at distinct automata. Hence, by Lemma 4.2 there exists an execution α_7 of the form $P_k \circ I_2 \circ I_1 \circ F_{1,1}^{(v_1^0)} \circ F_{2,2}^{(v_2^1)} \circ F_{2,1}^{(v_1^1)} \circ F_{1,2}^{(v_2^0)} \circ E_1^{(v_1^0,v_2^0)} \circ E_2^{(v_1^1,v_2^1)} \circ S$. □

In the following lemma by using simple commuting arguments of Lemma 4.2, we show the existence of a fair execution α_7 of $\mathcal{B}$ where $F_{2,2}$ appears before $I_1 \circ F_{1,1}$, where R_1 returns (v_1^0, v_2^0) and R_2 returns (v_1^1, v_2^1).

Lemma 4.12 (Existence of α_8)**.** *There exists fair execution α_8 of $\mathcal{B}$ that contains transactions W, R_1 and R_2 and can be written in the form $P_k \circ I_2 \circ F_{2,2} \circ I_1 \circ F_{1,1} \circ F_{2,1} \circ F_{1,2} \circ E_1 \circ E_2 \circ S$, where R_1 returns (v_1^0, v_2^0) and R_2 returns (v_1^1, v_2^1).*

Proof. Consider the fair execution α_7 of $\mathcal{B}$ as in Lemma 4.11. In the context of of Lemma 4.2, in α_7 (Fig. 7) the actions in $F_{1,1}$ (identify as G_1) occur at s_1^A and those in $F_{2,2}$ (identify as G_2) at s_2^A. Then by Lemma 4.2 there exists a fair execution α' of $\mathcal{B}$, of the form $P_k \circ I_2 \circ I_1 \circ F_{2,2} \circ F_{1,1} \circ F_{2,1} \circ F_{1,2} \circ E_1 \circ E_2 \circ S$, where $F_{2,2}$ and $F_{1,1}$ are interchanged.

Since actions in $F_{2,2}$ (identify as G_1) occur at s_1^A and those in I_1 (identify as G_1) occur at s_1^A then by Lemma 4.2 there is a fair execution of $\mathcal{B}$, α_8 where $F_{2,2}$ appear before I_1, i.e., of the form $P_k \circ I_2 \circ F_{2,2} \circ I_1 \circ F_{1,1} \circ F_{2,1} \circ F_{1,2} \circ E_1 \circ E_2 \circ S$, where $F_{2,2}$ and I_1 are interchanged.

By (ii) of Lemma 4.2 we have $F_{2,1}(\alpha') \overset{s_1}{\sim} F_{2,1}(\alpha_8)$ hence $F_{2,1}$ sends v_1^1 and $F_{1,1}$ and $F_{1,2}$ sends v_1^0 and v_2^0, respectively. So considering these returned values we have α_8 (Fig. 7) in the form as stated in the lemma. □

The following lemma shows the existence of a fair execution of $\mathcal{B}$, α_9, where we move $F_{2,1}$ forward past $F_{1,1}$.

Lemma 4.13 (Existence of α_9)**.** *There exists fair execution α_9 of $\mathcal{B}$ that contains transactions W, R_1 and R_2 and can be written in the form $P_k \circ I_2 \circ F_{2,2}^{(v_2^1)} \circ I_1 \circ F_{2,1}^{(v_1^1)} \circ F_{1,1}^{(v_1^0)} \circ F_{1,2}^{(v_2^0)} \circ E_1^{(v_1^0,v_2^0)} \circ E_2^{(v_1^1,v_2^1)} \circ S$ where R_1 returns (v_1^0, v_2^0) and R_2 returns (v_1^1, v_2^1).*

Proof. In α_8 from Lemma 4.12, all the actions in I_1 occur at r_1; those in $F_{1,1}$ occur at s_1; and the actions in $F_{2,1}$ occur only at s_1. Note that actions of both execution fragments $F_{2,1}$ and $F_{1,1}$ occur at r_1. Consider the prefix of α_8 that ends with I_1 then suppose the adversary extends this prefix by adding an execution fragment of the form $F_{2,1} \circ F_{1,1}$ as follows. First note that the actions $send(m_1^{r_2})_{r_2,s_1}$ and $send(m_1^{r_1})_{r_1,s_1}$ appears in the prefix but do not have corresponding $recv$ actions. The adversary places action $recv(m_1^{r_2})_{r_2,s_1}$, and allows an execution fragment of the form $F_{2,1}$ to appear. Now, immediately after this the adversary further extends it with an execution fragment of the form $F_{1,1}$ by placing action $recv(m_1^{r_1})_{r_1,s_1}$. Next the fragment $F_{1,2}$ is added and is the same as $F_{1,2}(\alpha_8)$. This last step can be argued by the fact that none of the actions in $F_{1,2}$ can be affected by any of the output actions at $F_{2,1}$ and $F_{1,1}$. Note that the actions in $F_{1,2}$ are taking place at s_2, which is not affected by the above fragments, and therefore, at the end of this fragment s_2 returns

v_1^0, as in α_8. But by S property $F_{1,1}$ has to return v_1^0 as well. Note that a careful argument can be done by using Theorem 2.3 to conclude the same. Following this the adversary allows the rest of the execution by adding an execution fragment of the form $E_1 \circ E_2 \circ S$. The resulting fair execution is of the form $P_k \circ I_2 \circ F_{2,2}^{(v_2^1)} \circ I_1 \circ F_{2,1} \circ F_{1,1} \circ F_{1,2}^{(v_2^0)} \circ E_1 \circ E_2 \circ S$, where we retained the values wherever it is known, and we denote this fair execution by α_9.

Now, we argue about the return values in α_9. Applying Lemma 4.3 to R_2 and $F_{2,2}$ implies that R_2 returns (v_1^1, v_2^1). Similarly, applying Lemma 4.3 to R_1 and $F_{1,2}$ implies that R_1 must return (v_1^0, v_2^0) in α_9. □

Now we show the existence of a fair execution of $\mathcal{B}$ where the execution fragments corresponding to R_1 appears before R_2, where R_1 returns (v_1^0, v_2^0) and R_2 completes by returning (v_1^1, v_2^1).

Lemma 4.14 (Existence of α_{10})**.** *There exists fair execution α_{10} of $\mathcal{B}$ that contains transactions W, R_1 and R_2 and can be written in the form $P_k \circ R_2^{(v_1^1, v_2^1)} \circ R_1^{(v_1^0, v_2^0)} \circ S$. where R_1 returns (v_1^0, v_2^0) and R_2 returns (v_1^1, v_2^1).*

Proof. Now, by using Lemma 4.2 to α_9, we can interchange $F_{2,1}$ and I_1 to create a fair execution α_{10} (Fig. 7) of $\mathcal{B}$, which is of the form $P_k \circ I_2 \circ F_{2,2}^{(v_2^1)} \circ F_{2,1}^{(v_1^1)} \circ R_1^{(v_1^0, v_2^0)} \circ E_2^{(v_1^1, v_2^1)} \circ S$, where the returned values are determined by Lemma 4.1.

Note that in the execution fragment none of the actions in the execution fragment $I_1 \circ F_{1,1}^{(v_1^0)} \circ F_{1,2}^{(v_2^0)} \circ E_1^{(v_1^0, v_2^0)}$ occur at r_2^A and in $E_2^{(v_1^1, v_2^1)}$ each of the action occur at r_2^A. Therefore, by applying Lemma 4.2, we can consecutively swap E_2 with E_1, $F_{1,2}$, I_1 and $F_{1,1}$ we create a sequence of four fair executions of $\mathcal{B}$ to arrive at fair execution α_{10} (Fig. 7) of the form $P_k \circ R_2^{(v_1^1, v_2^1)} \circ R_1^{(v_1^0, v_2^0)} \circ S$. □

The following statement proves the statement of the SNOW Theorem by showing the existence of fair execution α_{10}, of $\mathcal{B}$, where R_2 completes before R_1 is invoked and R_2 completes by returning (v_1^1, v_2^1) whereas R_1 returns (v_1^0, v_2^0), which violates the S property in α_{10}.

Theorem 4.15. *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.*

Proof. Note that in α_{10}, R_2 completes by returning (v_1^1, v_2^1) and R_1, although invoked after R_2 returns (v_1^0, v_2^0), is initiated and as a result α_{10} violates the S property. □

5 Condition for proving strict serializability

In this section, we derive a useful property for executions of algorithms that implement objects of data type $\mathcal{O}_T$ that will later help us show the *strict serializability* property (S property) of algorithms presented in later sections.

Although the strict serializability property in transaction-processing systems is a well-studied topic, the specific setting considered in this paper is much simpler. Therefore, this allows us to derive simpler conditions to prove the safety of these algorithms. A wide range of transaction types and transaction processing systems are considered in the literature. For example, in [13], Papadimitriou defined the strict serializability conditions as a part of developing a theory for analyzing transaction processing systems. In this work, each transaction T consists of a set of write operations W, at individual objects, and a set of read operations R from individual objects, where the operations

in W must complete before the operations in R execute. Other types of transaction processing systems allow nested transactions [4, 8, 11], where the transactions may contain sub-transactions [2] which may further contain a mix of read or write operations, or even child-transactions. In most transaction processing systems considered in the literature, transactions can be *aborted* so as to handle failed transactions. As a result, the serializability theories are developed while considering the presence of aborts. However, in our system, we do not consider any abort, nor any client or server failures. A transaction in our system is either a set of independent writes or a set of reads (see Section 2.2) with all the reads or writes in a transaction operating on different objects. Such simplifications allow us to formulate an equivalent condition for the execution of an algorithm to prove the S property of such algorithms while implementing an object of data type $\mathcal{O}_T$.

We note that an execution of a variable of type $\mathcal{O}_T$ is a finite sequence $\mathbf{v}_0, INV_1, RESP_1, \mathbf{v}_1, INV_2, RESP_2, \mathbf{v}_2, \cdots, \mathbf{v}_r$ or an infinite sequence $\mathbf{v}_0, INV_1, RESP_1, \mathbf{v}_1, INV_2, RESP_2, \mathbf{v}_2, \ldots$, where INV's and $RESP$'s are invocations and responses, respectively. The $\mathbf{v}_i$s are tuples of the the form $(v_1, v_2, \cdots, v_k) \in \Pi_{i=1}^{q} V_i$, that corresponds to the latest values stored across the objects $o_1, o_2 \cdots o_k$, and the values in $\mathbf{v}_0$ are the initial values of the objects. Any adjacent quadruple such as $\mathbf{v}_i, INV_{i+1}, RESP_{i+1}, \mathbf{v}_{i+1}$ is consistent with the f function for an object of type $\mathcal{O}_T$ (see Section 5) . Now, the safety property of such an object is a trace that describes the correct response to a sequence of INVs when all the transactions are executed sequentially. The *strict serializability* of $\mathcal{O}_T$ says that each trace produced by an execution of $\mathcal{O}_T$ with concurrent transactions appears as some trace of $\mathcal{O}_T$. We describe this below in more detail.

Definition 2 (Strict-serializability)**.** *Let us consider an execution β of an object of type $\mathcal{O}_T$, such that the invocations of any transaction at any client respects the well-formedness property. Let Π denote the set of complete transactions in β then we say β satisfies the strict-serializability property for $\mathcal{O}_T$ if the following are possible:*

(*i*) *For every complete* READ *or* WRITE *transaction π we insert a point (serialization point) π_* between the actions $INV(\pi)$ and $RESP(\pi)$.*

(*ii*) *We select a set Φ of incomplete transactions in β such that for each $\pi \in \Phi$ we select a response $RESP(\pi)$.*

(*iii*) *For each $\pi \in \Phi$ we insert π_* somewhere after $INV(\pi)$ in β, and remove the INV for the rest of the incomplete transactions in β.*

(*iv*) *If we assume for each $\pi \in \Pi \cup \Phi$ both $INV(\pi)$ and $RESP(\pi)$ to occur consecutively at π_*, with the interval of the transaction shrunk to π_*, then the sequence of transactions in this new trace is a trace of an object of data type $\mathcal{O}_T$.*

Now, we consider any automaton $\mathcal{B}$ that implements an object of type $\mathcal{O}_T$, and prove a result that serves us an equivalent condition for proving the strict serializability property of $\mathcal{B}$. Any trace property P of an automaton is a *safety property* if the set of executions in P is non-empty; *prefix-closed*, meaning any prefix of an execution in P is also in P; and *limit-closed*, i.e., if β_1, $\beta_2, \cdots$ is any infinite sequence of executions in P is such that β_i is prefix of β_{i+1} for any i, then the limit β of the sequence of executions $\{\beta_i\}_{i=0}^{\infty}$ is also in P. From Theorem 13.1 in [10], we know that the trace property, which we denote by P_{SC}, of any well-formed execution of $\mathcal{B}$ that satisfies the strict-serializability property is a safety property. Moreover, from Lemma 13.10 in [10] we can deduce that if every execution of $\mathcal{B}$ that is well-formed and failure-free, and also contains

no incomplete transactions, satisfies P_{SC}, then any well-formed execution of $\mathcal{B}$ that can possibly have incomplete transactions is also in P_{SC}. Therefore, in the following lemma, which gives us an equivalent condition for the strict serializability property of an execution β, we consider only executions without any incomplete transactions. The lemma is proved in a manner similar to Lemma 13.16 in [10], for atomicity guarantee of a single multi-reader multi-writer object.

Lemma 5.1. *Let β be an execution (finite or infinite) of an automaton $\mathcal{B}$ that implements an object of type $\mathcal{O}_T$, which consists of a set of k sub-objects. Suppose all clients in β behave in an well-formed manner. Suppose β contains no incomplete transactions and let Π be the set of transactions in β. Suppose there exists an irreflexive partial ordering ($\prec$) among the transactions in Π, such that,*

- $P1$ *For any transaction $\pi \in \Pi$ there are only a finite number of transactions $\phi \in \Pi$ such that $\phi \prec \pi$;*
- $P2$ *If the response event for π precedes the invocation event for ϕ in β, then it cannot be that $\phi \prec \pi$;*
- $P3$ *If π is a* WRITE *transaction in π and ϕ is any transaction in Π, then either $\pi \prec \phi$ or $\phi \prec \pi$; and*
- $P4$ *A tuple $\mathbf{v} \equiv (v_{i_1}, v_{i_2}, \cdots, v_{i_q})$ returned by a $READ(o_{i_1}, o_{i_2}, \cdots, o_{i_q})$, where q is any positive integer, $1 \leq q \leq k$, is such that v_{i_j} $j \in \{1, \cdots, q\}$ is written in β by the last preceding (w.r.t. $\prec$)* WRITE *transaction that contains a write$(o_{i_j}, *)$, or the initial value $v_{i_j}^0$ if no such* WRITE *exists in β.*

Then execution β is strictly serializable.

Proof. We discuss how to insert a serialization point $*_\pi$ in β for every transaction $\pi \in \Pi$. First, we add $*_\pi$ immediately after the latest of the invocations of π or $\phi \in \Pi$ such that $\phi \prec \pi$. Note that according to condition $P1$ for π there are only finite number of such invocations in β, therefore, π_* is well-defined. Now, since the order of the invocation events of the transactions in Π are already defined the order of the corresponding set of serialization points are well-defined except for the case when more than one serialization points are placed immediately after an invocation. In the case such multiple serialization points corresponding to an invocation we order order these serialization points in accordance with the $\prec$ relation of the underlying transactions.

Next, we show that for any pair of transactions $\phi, \pi \in \Pi$ if $\phi \prec \pi$ then $*_\phi$ precedes $*_\pi$. Suppose $\phi \prec \pi$. By construction, each of π_* and ϕ_* appear immediately after some invocation of some transaction in Π. If both π_* and ϕ_* appear immediately after the same invocation, then since $\phi \prec \pi$, by construction of π_*, π_* is ordered after ϕ_*. Also, if the invocations after which π_* and ϕ_* appear are distinct, then by construction of π_*, π_* appear after ϕ_* since $\phi \prec \pi$.

Next we argue that each $*_\pi$ serialization point for any $\pi \in \Pi$ is placed between the invocation $INV(\pi)$ and responses $RESP(\pi)$. By construction, $*_\pi$ is after $INV(\pi)$. To show that $*_\pi$ is before $RESP(\pi)$ for the sake of contradiction assume that $*_\pi$ appears after $RESP(\pi)$. By construction, $*_\pi$ must be after $INV(\phi)$ for some $\phi \in \Pi$ and $\phi \neq \pi$, they by the condition of construction of π_* we have $\phi \prec \pi$. But from above $INV(\phi)$ occurs after $RESP(\pi)$, i.e., π completes before ϕ in invoked which means, by property $P2$, we cannot have $\phi \prec \pi$, a contradiction.

Next, we show that if we were to shrink the transactions intervals to their corresponding serialization points, the resulting trace would be a trace of the underlying data type $\mathcal{O}_T$. In other

words, we show any READ $READ(o_{i_1}, o_{i_2}, \cdots, o_{i_q})$ returns the values $(v_{i_1}, v_{i_2}, \cdots, v_{i_q})$, such that each value v_{i_j}, $j \in [q]$, was written by the immediately preceding (w.r.t. the serialization points) WRITE that contained $write(o_{i_j}, v_{i_j})$ or the initial values if no such previous WRITE exists. Let us denote the set of WRITEs that precedes (w.r.t. $\prec$) π by $\Pi_W^{\prec\pi}$, i.e., $\phi \in \Pi_W^{\prec\pi}$ ϕ is a write and $\phi \prec \pi$. By property $P3$, all transactions in $\Pi_W^{\prec\pi}$ are totally-ordered. By property $P4$, v_{i_j} must be the value updated by the most recent WRITE in $\Pi_W^{\prec\pi}$. Since the total order of serialization points are consistent with $\prec$ and hence the v_{i_j} corresponds to the write operation of a WRITE transaction with the most recent serialization point and contains a operation of type $write(o_{i_j}, *)$. □

6 SNOW on MWSR with client-to-client messages

In this section, we present algorithm A for transaction processing in the *multiple-writers single-reader* (MWSR) setting, and prove that any fair and well-formed execution of A satisfies the SNOW properties. In practice, a system with a single reader may not be very useful but this algorithm serves as counter example algorithm to exhibit the point that if client-to-client communication is allowed it is still possible to implement the SNOW properties. Algorithm A shows that if client-to-client messaging is allowed, it is possible to have algorithms for transaction processing with two clients that satisfies the SNOW properties. We consider a system where there are $\ell \geq 1$ writers with ids $w_1, w_2 \cdots w_\ell$ (we denoted this set by $\mathcal{W}$), one reader with id r, and $k \geq 1$ servers with ids $s_1, s_2 \cdots s_k$ (denote as $\mathcal{S}$) that maintains the objects $o_1, \cdots, o_k$, respectively. We assume writers can send messages to the reader, and the reader can respond back to the writers, i.e., we allow client-to-client messages. Note that for a two-client system, when both clients are of the same type, i.e., two writers or two reads, the SNOW properties are trivially satisfied.

The steps of algorithm A are presented in Fig.4. We assume that each of the processes are run in a single-threaded manner, and therefore, each of the servers or the clients executes the algorithmic steps sequentially. For uniquely identifying a WRITE transaction we use keys in algorithm A. 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. Also, with each transaction we associate a tag $t \in \mathbb{N}$, which will help us define an order among the transactions.

State Variables: The state variables in writer, reader and server processes are as follows. (i) Any *writer* w has a counter z to keep track of the number of WRITE transactions it has invoked so far, initially 0. (ii) The *reader* r has an ordered list $List$ 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 $List$ can be though of as an array, with 0 as the starting index. (iii) Each *server* $s_i \in \mathcal{S}$ there is a set variable $Vals$ with elements that are key-value pairs $(\kappa, v_i) \in \mathcal{K} \times \mathcal{V}_i$. Initially, $Vals = \{(\kappa^0, v_i^0)\}$.

Writer steps: The procedure $WRITE((o_{i_1}, v_{i_1}), (o_{i_2}, v_{i_2}), \cdots, (o_{i_p}, v_{i_p}))$, for a WRITE transaction, can be invoked at any writer w, where $I = \{i_1, i_2, \cdots, i_p\}$ is any 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 *inform-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-VALUE, (κ, v_i)) to each server s_i in S_I, and awaits ACKNOWLEDGEs from each server in S_I. After receiving ACKNOWLEDGEs from each server in S_I, w initiates the *inform-reader* phase during which it sends (INFORM-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 ACKNOWLEDGE from r it completes the WRITE.

Reader steps: Note that 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 READ$(o_{i_1}, o_{i_2}, \cdots, o_{i_p})$, 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-VALUE, 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 $(v_{i_1}, \cdots v_{i_p})$.

Next, if reader r receives a message (INFORM-READER, $(\kappa, (b_1, \cdots, b_k))$ from any writer w, then r appends $(\kappa, (b_1, \cdots, b_k)$ to its $List$, and responds to w with ACKNOWLEDGE and $t_w = |List|$, i.e., number of elements in list $List$. The order of the elements in $List$ corresponds to the order the WRITE transactions, the order of the incoming INFORM-READER updates, as seen by the reader.

Server steps: The server protocol consists of two procedures corresponding to the messages containing the tags WRITE-VALUE and READ-VALUE. The first procedure is used if a server s_i receives a message (WRITE-VALUE, (κ, v_i)) from a writer w, it adds (κ, v_i) to its set variable $Vals$ and sends ACKNOWLEDGE back to w. The second procedure is used if s_i receives a message such as (READ-VALUE, κ_i) from r then it responds back with v_i such that the pair (κ_i, v_i) is in its variable $Vals$.

The following result states that algorithm A respects SNOW properties. Note that the liveness property of READ and WRITE transactions are a part of the SNOW properties. Consider any failure-free execution of algorithm A. In the steps for the reader assume the quantity $t_r \triangleq \max_{1 \le j \le |List|}\{j : List[j].b_i = 1 \wedge i \in I\}$, which is presented as a comment in the pseudo-code for A. We associate with any transaction ϕ a tag $tag(\phi)$ such that if ϕ is a WRITE $tag(\phi) = t_w$, i.e., the value of t_w before the completion of the operation, and $tag(\phi) = t_r$ when ϕ is a READ.

Theorem 6.1. *Any well-formed and fair execution of algorithm A is an wait-free implementation of transaction processing in the MWSR setting with for objects of type $\mathcal{O}_T$, consisting of objects $o_1, o_2, \cdots o_k$ maintained by the servers $s_1, s_2, \cdots, s_k$, respectively; and it respects the SNOW properties and* WRITE*s transactions are live.*

Proof. Below we show that A satisfies the SNOW properties.
S property: Let β be any fair execution of A and suppose all clients in β behave in an well-formed manner. Suppose β contains no incomplete transactions and let Π be the set of transactions in β. We define an irreflexive partial ordering $(\prec)$ among the transactions in Π as follows: if ϕ and π are any two distinct transactions in Π then we say $\phi \prec \pi$ if either (i) $tag(\phi) < tag(\pi)$ or (ii) $tag(\phi) = tag(\pi)$ and ϕ is a WRITE and π is a READ. We will prove the S (strict-serializability) property of A by proving that the properties $P1$, $P2$, $P3$ and $P4$ of Lemma 5.1 hold for β.

P1: If π is a READ then since all READs are invoked by a single reader r and in a well-formed manner, therefore, there cannot be an infinite number of READs such that they all precede π (w.r.t $\prec$). Now, suppose π is a WRITE. Clearly, from an inspection of the algorithm, $tag(\pi) \in \mathbb{N}$. From inspection of the algorithm, each WRITE increases the size of $List$, and the value of the tags are defined by the size of $List$. Therefore, there can be at most a finite number of WRITEs such that can precede π (w.r.t. $\prec$) in β.

P2: Suppose ϕ and π are any two transactions in Π, such that, π begins after ϕ completes.

Fig. 4 The protocol for a writer w, reader r and server s_i for algorithm A.

At writer w

State Variables at w**:**

$z \in \mathbb{N}$, initially 0

WRITE$((o_{i_1}, v_{i_1}), (o_{i_2}, v_{i_2}), \cdots, (o_{o_p}, v_{i_p}))$

write-value:

$\kappa \leftarrow (z+1, w)$; $z \leftarrow z+1$

$I \triangleq \{i_1, i_2, \cdots, i_p\}$

for $i \in I$ **do**

Send WRITE-VALUE, (κ, v_{s_i}) to s_i

Await ACKNOWLEDGE from s_i for every $i \in I$.

inform-reader:

for $i \in [k]$ **do**

if $i \in I$ **then**

$b_i \leftarrow 1$

else

$b_i \leftarrow 0$

Send (INFORM-READER, $(\kappa, (b_1, \cdots, b_k))$) to r

Receive (ACKNOWLEDGE, t_w) from r

At reader r

State Variables at r**:**

List, a list of elements in $\mathcal{K} \times \{0,1\}^k$,

initially $[(\kappa^0, 1, \cdots 1)]$

READ$(o_{i_1}, o_{i_2}, \cdots, o_{i_p})$

read-value:

$I \triangleq \{i_1, i_2, \cdots, i_p\}$

for $i \in I$ **do**

$j^* \leftarrow \max_{1 \le j \le |List|}\{j : List[j].b_i = 1\}$

$\kappa_i = List[j^*].\kappa$

Send (READ-VALUE, κ_i) to s_i

Await responses v_i from s_i for each $i \in I$

/* $t_r \triangleq \max_{1 \le j \le |List|}\{j : List[j].b_i = 1 \wedge i \in I\}$ */

Return $(v_{i_1}, v_{i_2}, \cdots, v_{i_p})$

Response routines

On recv (INFORM-READER, $(\kappa, (b_1, \cdots b_k))$) **from** w**:**

$List \leftarrow List \bigoplus (\kappa, (b_1, \cdots b_k))$ //$\bigoplus$ for append

$tag \leftarrow |List|$ // $|\cdot|$ size of the list

Send (ACKNOWLEDGE, tag) to w

At server s_i **for any** $i \in [k]$

State Variables:

$Vals \subset \mathcal{K} \times \mathcal{V}_i$, initially $\{(t^0_{key}, v^0_i)\}$

On recv (WRITE-VALUE, (κ, v)) **from writer** w**:**

$Vals \leftarrow Vals \cup \{(\kappa, v)\}$

Send ACKNOWLEDGE to writer w.

On recv (READ-VALUE, κ) **from reader** r **:**

Send v s.t. $(\kappa, v) \in Vals$ to reader r

Then we show that we cannot have $\pi \prec \phi$. Now, we consider four cases, depending on whether ϕ and π are READs or WRITEs.

(a) ϕ and π are WRITEs invoked by writers w_ϕ and w_π, respectively. Since the size of *List*, in r, grows monotonically with each WRITE hence w_π receives the tag at least as high as $tag(\phi)$, so $\pi \not\prec \phi$.

(b) ϕ is a WRITE, π is a READ transactions invoked by writer w_ϕ and r, respectively. Since the size of *List*, in r, grows monotonically, and because w_π invokes π after ϕ completes hence $tag(\pi)$ is at least as high as $tag(\phi)$, so $\pi \not\prec \phi$.

(c) ϕ and π are READs invoked by reader r. Since the size of *List*, in r, grows monotonically, hence w_π invoked π after ϕ completes hence $tag(\pi)$ is at least as high as $tag(\phi)$, so $\pi \not\prec \phi$.

(d) ϕ is a READ, π is a WRITE invoked by reader r and w_π, respectively. This case is simple because new values are added to *List* only by writers, and $tag(\pi)$ is at least as large as the tag of ϕ and hence $\pi \not\prec \phi$.

P3: This is clear by the fact that any WRITE transaction always creates a unique tag and all tags are totally ordered since they all belong to $\mathbb{N}$

P4: Consider a READ ρ as $READ(o_{i_1}, o_{i_2}, \cdots, o_{i_q})$, in β. Let the returned value from ρ be $\mathbf{v} \equiv (v_{i_1}, v_{i_2}, \cdots, v_{i_q})$ such that $1 \leq i_1 < i_2 < \cdots < i_q \leq k$, where value v_{i_j} corresponds to o_{i_j}. Suppose $tag(\rho) \in \mathbb{N}$ was created during some WRITE transaction, say ϕ, i.e., ϕ is the WRITE that added the elements in index $(tag(\rho) - 1)$ of *List*. Note that element in index 0 contains the initial value. Now we consider two cases:

Case $tag(\rho) = 1$. We know that it corresponds the initial default value v_i^0 at each sub-object o_i, and this equates to ρ returning the default initial value for each sub-object.

Case $tag(\rho) > 1$. Then we argue that there exists no WRITE transaction, say π, that updated object o_{i_j}, in β, such that, $\pi \neq \phi$ and ρ returns values written by π and $\phi \prec \pi \prec \rho$. Suppose we assume the contrary, which means $tag(\phi) < tag(\pi) < tag(\rho)$. The latter implies $tag(\phi) = tag(\pi)$ which is not possible because this contradicts the fact that for any two distinct WRITEs $tag(\phi) \neq tag(\pi)$ in any execution of A.

N property: By inspection of algorithm A for the response steps of the servers to the reader.

O property: By inspection of the *read-value* phase: it consists of one round of communication between the reader and the servers, where the servers send only one version of the value of the object it maintains.

W property: By inspection of the WRITE transaction steps, and and that writers always get to complete the transactions they invoke. □

Note that the above theorem holds in the presence of any writer crashes.

7 SNoW for MWMR setting

In this section, we present algorithm B for transaction processing in the *multiple-writers multi-reader* (MWMR) setting and show that its execution satisfies SNoW properties, where "o" means a READ may consist of more than one round trip of communications between the reader and the servers. We denote the type of object that satisfies the SNoW properties by $\tilde{\mathcal{O}}_T$. The steps of the algorithm for the writers, readers and the servers are presented in Fig.5. We assume there is 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. We define 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. Like in algorithm A from Section 6, also in B, the keys are used to uniquely identify each transaction. Also, with each transaction we associate a tag $t \in \mathbb{N}$.

In algorithm B, we designate one of the servers as the coordinator, we denote as s^*, for the transactions. Essentially, the coordinator s^* is used to maintain the order of the WRITEs and the objects that are updated during the WRITE in the variable *List*. Note that in a system, where there are many objects different objects may be use different servers as coordinators based on some load-balancing rule.

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 $(b_1, \cdots b_k) \in \{0, 1\}^k$.

Fig. 5 The protocol for any writer w, reader r or server s_i for algorithm B.

At writer w

State Variables:

$z \in \mathbb{N}$, initially 0

WRITE$((i_1, v_{i_1}), (i_2, v_{i_2}), \cdots, (i_p, v_{i_p}))$

$I \triangleq \{i_1, i_2, \cdots, i_p\}$

write-value:

$\kappa \leftarrow (z+1, w)$; $z \leftarrow z+1$

for $i \in I$ **do**

  Send WRITE-VALUE, (κ, v_{s_i}) to server s_i

Await ACKNOWLEDGE from servers in S_I.

update-coord:

for $i \in [k]$ **do**

  if $i \in I$ **then**

    $b_i \leftarrow 1$

  else

    $b_i \leftarrow 0$

Send (UPDATE-COORD, $(\kappa, (b_1, \cdots, b_k))$) to s^*

Receive (ACKNOWLEDGE, t_w) from coordinator s^*

At reader r

READ$(o_{i_1}, o_{i_2}, \cdots, o_{i_p})$

$I \triangleq \{i_1, i_2, \cdots, i_p\}$

get-tag-array:

Send (GET-TAG-ARRAY) to server s^*

Receive response $(t_r, (\kappa_1, \kappa_2, \cdots, \kappa_k))$ from s^*

read-value:

for $i \in I$ **do**

  Send (READ-VALUE, κ_i) to s_i

Wait responses as v_i for each $s_i \in S$

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 $\{(\kappa^0, v_i^0)\}$

$List$, a list of $\mathcal{K} \times \{0,1\}^k$, initially $[(\kappa^0, (1, \cdots 1))]$

On recv (WRITE-VALUE, (κ, v)) **from** w:

$Vals \leftarrow Vals \cup \{(\kappa, v)\}$

Send ACKNOWLEDGE to writer w.

On recv (UPDATE-COORD, $(\kappa, (b_1, \cdots b_k))$) **from** w:

$List \leftarrow List \bigoplus (\kappa, (b_1, \cdots b_k))$ //$\bigoplus$ for append

$tag \leftarrow |List|$ // $|\cdot|$ size of the list

Send (ACKNOWLEDGE, tag) to w

On recv (READ-VALUE, κ) **from** r :

Send v_i s.t. $(\kappa, v) \in Vals$ to r

/* used only by s^* */

On recv GET-TAG-ARRAY **from** r :

for $i \in [k]$ **do**

  $j^* \leftarrow \max\{j : List[j].b_i = 1\}$

  $\kappa_i = List[j^*].\kappa$

$t_r \triangleq \max_{1 \le j \le |List|}\{j : List[j].b_i = 1 \wedge i \in I\}$

Send $(t_r, (\kappa_1, \kappa_2, \cdots, \kappa_k))$ to r

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 execution of the WRITE transaction, otherwise $b_i = 0$.

Writer steps: A WRITE transaction that is meant to update 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 $WRITE((o_{i_1}, v_{i_1}), (o_{i_2}, v_{i_2}), \cdots, (o_{i_p}, v_{i_p}))$. 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-coord*. During the *write-value* phase, w creates a new key κ as $\kappa \equiv (z+1, w)$, where w is the identity of the writer; and also increments the local counter z by one. Then w sends (WRITE-VALUE, (κ, v_i)) to each server in S_I, and awaits ACKNOWLEDGE from all servers in S_I. After receiving ACKNOWLEDGE from all servers in S_I, w initiates the *update-coord* phase where it sends (UPDATE-COORD, $(\kappa, (b_1, \cdots b_k))$ to s^*, where for any $i \in [k]$, $b_i = 1$ if $s_i \in S_I$, otherwise $b_i = 0$, and completes the procedure after it receive a ACKNOWLEDGE message from s^*. After receiving message as (ACKNOWLEDGE, t_w), w completes the WRITE.

Reader steps: Note that we use the same notations for I and S_I as above by the set of indices are not necessarily similar across different transactions. The procedure READ$(o_{i_1}, o_{i_2}, \cdots, o_{i_p})$ 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-ARRAY 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-VALUE, 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-VALUE, (κ, v_i)) from a writer w then it adds (κ, v_i) to its set variable $Vals$ and sends ACKNOWLEDGE back to w.

If the coordinator s^*, receives (UPDATE-COORD, $(\kappa, (b_1, \cdots, b_k))$ from writer w, then it appends $(\kappa, (b_1, \cdots, b_k))$ to its $List$, and responds with ACKNOWLEDGE 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-COORD updates, as seen by s^*.

Again, when s^* receives the message GET-TAG-ARRAY from r it responds with a message $(\kappa_1, \cdots, \kappa_k)$ 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, $t_r \triangleq \max_{1 \le j \le |List|}\{j : List[j].b_i = 1 \wedge i \in I\}$.

If any server s_i receives a message (READ-VALUE, κ) from a reader r then it responds to r with the value v_i corresponding to key with value κ in $Vals$.

Note the the following result states that algorithm B respects SNoW property. Consider any failure-free and fair execution of algorithm B. For the purpose of proving the S property, for every transaction transaction ϕ in an execution of B we associate a tag $tag(\phi)$ as described below. If ϕ is a WRITE (READ) then $tag(\phi)$ is the value of the variable t_w (t_r) immediately before the operation completes.

Theorem 7.1. *Any well-formed and fair execution of algorithm B is an implementation of an object of type $\tilde{\mathcal{O}}_T$ in the MWMR setting, with no client-to-client communication, comprising of objects $o_1, o_2, \cdots o_k$ stored in servers $s_1, s_2, \cdots, s_k$, respectively; and it satisfies the SNoW properties.*

Proof. Below we show that algorithm B satisfies the SNoW properties.
S property: Let β be any fair execution of B and suppose all clients in β behave in an well-formed manner. Suppose β contains no incomplete transactions and let Π be the set of transactions in β. We define an irreflexive partial ordering $(\prec)$ in Π as follows: if ϕ and π are any two distinct transactions in Π then we say $\phi \prec \pi$ if either (i) $tag(\phi) < tag(\pi)$ or (ii) $tag(\phi) = tag(\pi)$ and ϕ is a WRITE and π is a READ. Below we prove the S property of B by showing that properties $P1$, $P2$, $P3$ and $P4$ of Lemma 5.1 hold for β.

P1: Clearly, from an inspection of the algorithm, $tag(\pi) \in \mathbb{N}$. From inspection of the algorithm, each WRITE increases the size of $List$, and the value of the tags are defined by the size of $List$. Therefore, there can be at most a finite number of WRITEs such that can precede π (w.r.t. $\prec$) in β. On the other hand, if π is a READ then since all READs are invoked by readers in a well-formed manner, and there are only finite number of readers therefore, there cannot be an infinite number of READs such that they all precede π (w.r.t $\prec$).

P2: Suppose ϕ and π are any two transactions in Π, such that, π begins after ϕ completes.

Then we show that we cannot have $\pi \prec \phi$. Now, we consider four cases, depending on whether ϕ and π are READs or WRITEs.

(*a*) π and ϕ are WRITEs invoked by writers w_ϕ and w_π, respectively. Since the size of $List$, in s^* grows monotonically due to each WRITE hence w_π receives the tag from s^* at least as high as $tag(\phi)$, so $\pi \not\prec \phi$.

(*b*) π is a READ, ϕ is a WRITE invoked by reader r_π and writer w_ϕ, respectively. Since the size of $List$, in s^*, grows monotonically, because r_π invokes π after ϕ completes hence $tag(\pi)$ is at least as high as $tag(\phi)$, so $\pi \not\prec \phi$.

(*c*) π and ϕ are both READs invoked by readers r_π and r_ϕ, respectively. Since the size of $List$, in s^*, grows monotonically, because w_π invokes π after ϕ completes hence $tag(\pi)$ is at least as high as $tag(\phi)$, so $\pi \not\prec \phi$.

(*d*) π is a WRITE, ϕ is a READ invoked by writer w_π and reader r_ϕ, respectively. This case is simple because new values are added to $List$, in s^*, only by writers, and $tag(\pi)$ is at least as large as the tag of ϕ and hence $\pi \not\prec \phi$.

P3: This is from the fact that any WRITE transaction always creates a unique tag and all tags are totally ordered since they all belong to $\mathbb{N}$

P4: Consider a READ ρ as $READ(o_{i_1}, o_{i_2}, \cdots, o_{i_q})$, in β. Let the returned value from ρ be $\mathbf{v} \equiv (v_{i_1}, v_{i_2}, \cdots, v_{i_q})$ such that $1 \leq i_1 < i_2 < \cdots < i_q \leq k$, where value v_{i_j} corresponds to o_{i_j}. Suppose $tag(\rho) \in \mathbb{N}$ was created during some WRITE transaction, say ϕ, i.e., ϕ is the WRITE that added the elements in index $(tag(\rho) - 1)$ of $List$ at the coordinator s^*. Note that element in index 0 contains the initial value. Now we consider two cases:

Case $tag(\rho) = 1$. We know that it corresponds the initial default value v_i^0 at each sub-object o_i, and this equates to ρ returning the default initial value for each sub-object.

Case $tag(\rho) > 1$. Then we argue that there exists no WRITE transaction, say π, that updated object o_{i_j}, in β, such that, $\pi \neq \phi$ and ρ returns values written by π and $\phi \prec \pi \prec \rho$. Suppose we assume the contrary, which means $tag(\phi) < tag(\pi) < tag(\rho)$. The latter implies $tag(\phi) = tag(\pi)$ which is not possible because this contradicts the fact that for any two distinct WRITEs $tag(\phi) \neq tag(\pi)$ in any execution of B.

N, o and W properties: Evident from an inspection of the algorithm. □

8 SN$\bar{o}$W for MWMR setting

In this section, we present algorithm C for transaction processing in the *multiple-writers multi-reader* (MWMR) setting and show that its execution satisfies SN$\bar{o}$W properties, where "$\bar{o}$" means a READ consists of only one round trip of communications between the reader and the servers but the servers may respond with multiple versions of the data. We denote the type of object that satisfies the SN$\bar{o}$W properties by $\bar{\mathcal{O}}_T$. The steps of the algorithm for the writers, readers and the servers are presented in Fig.6. We assume there is 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. We define 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. As in the algorithms presented in Section 6 and 7, the keys are used to uniquely identify each transaction. Also, with each transaction we associate a tag $t \in \mathbb{N}$.

Fig. 6 The protocol for any writer w, reader r or server s_i for algorithm B.

At writer w
State Variables:
$z \in \mathbb{N}$, initially 0
WRITE$((i_1, v_{i_1}), (i_2, v_{i_2}), \cdots, (i_p, v_{i_p}))$
$I \triangleq \{i_1, i_2, \cdots, i_p\}$
write-value:
  $\kappa \leftarrow (z+1, w)$; $z \leftarrow z+1$
  for $i \in I$ **do**
    Send WRITE-VALUE, (κ, v_{s_i}) to server s_i
  Await ACKNOWLEDGE from servers in S_I.

update-coord:
  for $i \in [k]$ **do**
    if $i \in I$ **then**
      $b_i \leftarrow 1$
    else
      $b_i \leftarrow 0$
  Send (UPDATE-COORD, $(\kappa, (b_1, \cdots, b_k))$) to s^*
  Receive (ACKNOWLEDGE, t_w) from coordinator s^*

At reader r
READ$(o_{i_1}, o_{i_2}, \cdots, o_{i_p})$
$I \triangleq \{i_1, i_2, \cdots, i_p\}$
read-values-and-tags:
  Send (GET-TAG-ARRAY) to server s^*
  for $i \in I$ **do**
    Send (READ-VALUES) to s_i
  Receive response $(t_r, (\kappa_1, \kappa_2, \cdots, \kappa_k))$ from s^*
  Wait responses as $Vals_i$ from each $s_i \in S_I$
  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-VALUE, (κ, v)) **from** w:
  $Vals \leftarrow Vals \cup \{(\kappa, v)\}$
  Send ACKNOWLEDGE to writer w.

On recv (UPDATE-COORD, $(\kappa, (b_1, \cdots b_k))$) **from** w:
  $List \leftarrow List \bigoplus (\kappa, (b_1, \cdots b_k))$ //$\bigoplus$ for append
  $tag \leftarrow |List|$ // $|\cdot|$ size of the list
  Send (ACKNOWLEDGE, tag) to w

On recv (READ-VALUES) **from** r :
  Send $Vals$ to r

/* used only by s^* */
On recv GET-TAG-ARRAY **from** r :
  for $i \in [k]$ **do**
    $j^* \leftarrow \max\{j : List[j].b_i = 1\}$
    $\kappa_i = List[j^*].\kappa$
  $t_r \triangleq \max_{1 \le j \le |List|}\{j : List[j].b_i = 1 \wedge i \in I\}$
  Send $(t_r, (\kappa_1, \kappa_2, \cdots, \kappa_k))$ to r

In algorithm C, we designate one of the servers as the coordinator, we denote as s^*, for the transactions. In addition to managing some object, the server s^* is used to maintain the order of the WRITEs and the objects that are updated during the WRITE in the variable $List$. In any system, where there are many objects different objects may be use different servers as coordinators based on some load-balancing rule.

State Variables: The state variables in algorithm C is similar to those of algorithm B, therefore, we omit here. *Writer steps:* The steps of a WRITE transaction is similarly to algorithm B, therefore, we omit here.

Reader steps: The procedure READ$(o_{i_1}, o_{i_2}, \cdots, o_{i_p})$ 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. We use the notations: $I \triangleq \{i_1, i_2, \cdots, i_p\}$ and $S_I \triangleq \{s_{i_1}, s_{i_2}, \cdots, s_{i_p}\}$. The procedure consists of only one phase of communication round between the r and the servers, called *read-values-and-tags*. During the phase *read-values-and-tags*, r sends s^* the message GET-TAG-ARRAY requesting the list of the latest added keys for each object, and also sends requests (READ-VALUES) each server s_i in S_I. Note that if s^* is also one of the servers in S_I then the GET-TAG-ARRAY and READ-VALUES 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 of type (WRITE-VALUE, (κ, v_i)) from a writer w; or if the coordinator s^*, receives (UPDATE-COORD, $(\kappa, (b_1, \cdots, b_k))$ from writer w or receives a message as GET-TAG-ARRAY from r the steps are similar to their counterparts in algorithm B. Therefore, we omit them. On the other hand, if any server s_i receives a message (READ-VALUES) from a reader r then it responds to r with the entire set of values, i.e., $Vals$.

Note the the following result states that algorithm C respects SN$\bar{o}$W property. Consider any failure-free and fair execution of algorithm C. For the purpose of proving the S property, for every transaction transaction ϕ in an execution of C we associate a tag $tag(\phi)$ as described below. If ϕ is a WRITE (READ) then $tag(\phi)$ is the value of the variable t_w (t_r) immediately before the operation completes.

Theorem 8.1. *Any well-formed and fair execution of algorithm C is an implementation of an object of type $\bar{\mathcal{O}}_T$ in the MWMR setting, with no client-to-client communication, comprising of objects $o_1, o_2, \cdots o_k$ stored in servers $s_1, s_2, \cdots, s_k$, respectively; and it satisfies the SN$\bar{o}$W properties.*

Proof. Below we show that algorithm C satisfies the SN$\bar{o}$W properties.
S property: Let β be any fair execution of B and suppose all clients in β behave in an well-formed manner. Suppose β contains no incomplete transactions and let Π be the set of transactions in β. We define an irreflexive partial ordering ($\prec$) in Π as follows: if ϕ and π are any two distinct transactions in Π then we say $\phi \prec \pi$ if either (i) $tag(\phi) < tag(\pi)$ or (ii) $tag(\phi) = tag(\pi)$ and ϕ is a WRITE and π is a READ. Below we prove the S property of B by showing that properties $P1$, $P2$, $P3$ and $P4$ of Lemma 5.1 hold for β. The properties $P1$-$P4$ can be proved to hold in a manner very similar to algorithm B (Section 7). Therefore, we avoid repeating them.
N, $\bar{o}$ and W properties: Evident from an inspection of the algorithm. □

References

[1] Phillipe Ajoux, Nathan Bronson, Sanjeev Kumar, Wyatt Lloyd, and Kaushik Veeraraghavan. Challenges to adopting stronger consistency at scale. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, 2015. USENIX Association.

[2] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[3] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. TAO: Facebook's distributed data store for the social graph. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 49–60. USENIX, 2013.

[4] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, San Mateo, CA, 1993.

[5] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[6] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, ICDCS '03, pages 522–, Washington, DC, USA, 2003.

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

[8] B. Liskov. Distributed Programming in Argus. *Communications for the ACM*, 31(3):300–312, 1988.

[9] Haonan Lu, Christopher Hodsdon, Khiem Ngo, Shuai Mu, and Wyatt Lloyd. The SNOW theorem and latency-optimal read-only transactions. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 135–150, Savannah, GA, 2016.

[10] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[11] N. A. Lynch, M. Merritt, W. William, and A. Fekete. *Atomic Transactions*. Morgan Kaufmann Publishers, 1994.

[12] Henry Massalin and Calton Pu. A lock-free multiprocessor os kernel. *SIGOPS Oper. Syst. Rev.*, 26(2):8–, April 1992.

[13] Christos H. Papadimitriou. The serializability of concurrent database updates. *Journal of ACM*, 26(4):631–653, 1979.