Non-Monotonic Snapshot Isolation

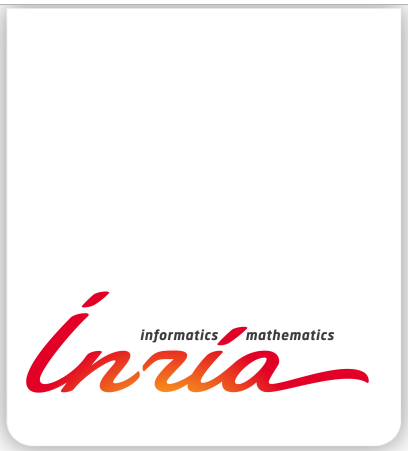
Masoud Saeida Ardekani, Pierre Sutra, Nuno Preguiça, Marc Shapiro

Non-Monotonic Snapshot Isolation

Masoud Saeida Ardekani UPMC-LIP6
Pierre Sutra INRIA & UPMC-LIP6
Nuno Preguiça Universidade Nova de Lisboa
Marc Shapiro INRIA & UPMC-LIP6

RESEARCH REPORT
N° 7805
October 2012
Project-Teams Regal

ISSN 0249-6399 ISRN INRIA/RR--7805--FR+ENG



Non-Monotonic Snapshot Isolation*

Masoud Saeida Ardekani UPMC-LIP6
Pierre Sutra INRIA & UPMC-LIP6
Nuno Preguiça Universidade Nova de Lisboa
Marc Shapiro INRIA & UPMC-LIP6

Project-Teams Regal

Research Report n° 7805 — version 2 — initial version
October 2012 — revised version Octobre 2012 — 38 pages

Abstract: Many distributed applications require transactions. However, transactional protocols that require strong synchronization are costly in large scale environments. Two properties help with scalability of a transactional system: genuine partial replication (GPR), which leverages the intrinsic parallelism of a workload, and snapshot isolation (SI), which decreases the need for synchronization. We show that, under standard assumptions (data store accesses are not known in advance, and transactions may access arbitrary objects in the data store), it is impossible to have both SI and GPR. To circumvent this impossibility, we propose a weaker consistency criterion, called Non-Monotonic Snapshot Isolation (NMSI). NMSI retains the most important properties of SI, i.e., read-only transactions always commit, and two write-conflicting updates do not both commit. We present a GPR protocol that ensures NMSI, and has lower message cost (i.e., it contacts fewer replicas and/or commits faster) than previous approaches.

Key-words: distributed systems; transcational systems; replication; concurrency control; transactions; database

* The work presented in this paper has been partially funded by ANR projects Prose (ANR-09-VERS-007-02) and Concordant (ANR-10-BLAN 0208).

RESEARCH CENTRE
PARIS – ROCQUENCOURT

Domaine de Voluceau, - Rocquencourt
B.P. 105 - 78153 Le Chesnay Cedex

Non-Monotonic Snapshot Isolation

Résumé : Cet article étudie deux propriétés favorisant le passage à l'échelle des systèmes répartis transactionnels: la réplication partielle authentique (GPR), et le critère de cohérence Snapshot Isolation (SI). GPR spécifie que pour valider une transaction T, seules les répliques des données accédées par T effectuent des pas de calcul. SI définit que toute transaction doit lire une vue cohérente du système, et que deux transactions concurrentes ne peuvent écrire la même donnée. Nous montrons que SI et GPR sont deux propriétés incompatibles. Afin de contourner cette limitation, nous proposons un nouveau critère de cohérence: Non-Monotonic Snapshot Isolation (NMSI). NMSI est proche de SI et néanmoins compatible avec GPR. Afin de justifier ce dernier point, nous présentons un protocole authentique implémentant de manière efficace NMSI. Au regard des travaux précédents sur le contrôle de concurrence dans les systèmes répartis transactionnels, notre protocole est le plus performant en latence et/ou en nombre de messages échangés.

Mots-clés : systèmes répartis, systèmes transactionnels, contrôle de concurrence, transaction, base de données

1 Introduction

Large scale transactional systems have conflicting requirements. On the one hand, strong transactional guarantees are fundamental to many applications. On the other, remote communication and synchronization is costly and should be avoided.[1]

To maintain strong consistency guarantees while alleviating the high cost of synchronization, Snapshot Isolation (SI) is a popular approach in both distributed database replications [1–3], and software transactional memories [4, 5]. Under SI, a transaction accesses its own *consistent snapshot* of the data, which is unaffected by concurrent updates. A read-only transaction always commits unilaterally and without synchronization. An update transaction synchronizes on commit to ensure that no concurrent conflicting transaction has committed before it.

Our first contribution is to prove that SI is equivalent to the conjunction of the following properties: *(i)* no cascading aborts, *(ii)* strictly consistent snapshots, i.e., a transaction observes a snapshot that coincides with some point in (linear) time, *(iii)* two concurrent write-conflicting update transactions never both commit, and *(iv)* snapshots observed by transactions are monotonically ordered. Previous definitions [6, 7] of SI extend histories with abstract snapshot points. Our decomposition shows that SI can be expressed on plain histories like serializability [8].

Modern data stores replicate data for both performance and availability. Full replication does not scale, as every process must perform all updates. *Partial replication* (PR) aims to address this problem, by replicating only a subset of the data at each process. Thus, if transactions would communicate only over the minimal number of replicas, synchronisation and computation overhead would be reduced. However, in the general case, the overlap of transactions cannot be predicted; therefore, many PR protocols perform system-wide global consensus [1, 2] or communication [9]. This negates the potential advantages of PR; hence, we require *genuine* partial replication [10] (GPR), in which a transaction communicates only with those processes that replicate some object accessed in the transaction. With GPR, independent transactions do not interfere with each other, and the intrinsic parallelism of a workload can be exploited. Our second contribution is to show that SI and GPR are incompatible. More precisely, we prove that an asynchronous message-passing system supporting GPR cannot compute monotonically ordered snapshots, nor strictly consistent ones, even if it is failure-free.

[1] We address general-purpose transactions, i.e., we assume that a transaction may access any object in the system, and that its read- and write-sets are not known in advance.

The good news is our third contribution: a consistency criterion, called *Non-Monotonic Snapshot Isolation* (NMSI) that overcomes this impossibility. NMSI is very similar to SI, as every transaction observes a consistent snapshot, and two concurrent write-conflicting updates never both commit. However, under NMSI, snapshots are neither strictly consistent nor monotonically ordered.

Our final contribution is a GPR protocol ensuring NMSI, called Jessy. Jessy uses a novel variant of version vectors, called *dependence vectors*, to compute consistent partial snapshots asynchronously. To commit an update transaction, Jessy uses a single atomic multicast. Compared to previous protocols, Jessy commits transactions faster and/or contacts fewer replicas.

This paper proceeds as follows. We introduce our system model in Section 2. Section 3 presents our decomposition of SI. Section 4 shows that GPR and SI are mutually incompatible. We introduce NMSI in Section 5. Section 6 describes Jessy, our NMSI protocol. We compare with related work in Section 7, and conclude in Section 8.

2 Model

This section defines the elements in our model and formalizes SI and GPR .

2.1 Objects & transactions

Let *Objects* be a set of objects, and $\mathcal{T}$ be a set of transaction identifiers. Given an object x and an identifier i, x_i denotes *version* i of x. A *transaction* $T_{i\in\mathcal{T}}$ is a finite permutation of read and write operations followed by a *terminating* operation, commit (c_i) or abort (a_i). We write $w_i(x_i)$ to denote transaction T_i writing version i of object x, and $r_i(x_j)$ to mean that T_i reads version j of object x. In a transaction, every write is preceded by a read on the same object, and every object is read or written at most once.[2] We note $ws(T_i)$ the write set of T, i.e., the set of objects written by transaction T_i. Similarly, $rs(T_i)$ denotes the read set of transaction T_i. The *snapshot* of T_i is the set of versions read by T_i. Two transactions *conflict* when they access the same object x and one of them modifies x. Two transactions *write-conflict* when they both write to the same object.

2.2 Histories

A *complete history* h is a partially ordered set of operations such that (1) for every operation o_i appearing in h, transaction T_i terminates in h, (2) for every two operations o_i and o'_i appearing in h, if o_i precedes o'_i in T_i, then $o_i <_h o'_i$, (3) for every read $r_i(x_j)$ in h, there exists a write operation $w_j(x_j)$ such that $w_j(x_j) <_h r_i(x_j)$, and (4) any two write operations over the same objects are ordered by $<_h$. A *history* is a prefix of a complete history. For some history h, order $<_h$ is the *real-time order* induced by h. Transaction T_i is *pending* in history h if T_i does not commit, nor abort in h. We note $\ll_h$ the version order induced by h between different versions of an object, i.e., for every object x, and every pair of transactions (T_i, T_j), $x_i \ll_h x_j \Leftrightarrow w_i(x_i) <_h w_j(x_j)$. Following Bernstein et al. [11], we depict a history as a graph. We illustrate this with history h_1 below in which transaction T_a reads the initial versions of objects x and y, while transaction T_1 (respectively T_2) updates x (resp. y).

$$h_1 = r_a(x_0) \longrightarrow r_1(x_0).w_1(x_1).c_1$$
$$r_a(y_0).c_a \longrightarrow r_2(y_0).w_2(y_2).c_2$$

[2] This restriction can be easily overcome by reading shared object in privates variables and computing with them.

When order $<_h$ is total, we shall write a history as a permutation of operations, e.g., $h_2 = r_1(x_0).r_2(y_0).w_2(y_2).c_1.c_2$.

2.3 Snapshot Isolation

Snapshot isolation (SI) was introduced by Berenson et al. [8], then later generalized under the name GSI by Elnikety et al. [7]. In this paper, we make no distinction between SI and GSI. Our definition is derived from Adya [6] and Elnikety et al. [7]. We introduce a function $\mathcal{S}$ over histories, which takes as input a history h, and returns an extended history h_s by adding a *snapshot point* to h for each transaction in h. Given a transaction T_i, the snapshot point of T_i in h_s, denoted s_i, precedes every operation of transaction T_i in h_s. A history h is in SI if and only if there exists a function $\mathcal{S}$ such that $h_s = \mathcal{S}(h)$ and h_s satisfies the following safety rules:

D1 (Read Rule)

$\forall r_i(x_{j\neq i}) \in h_s :$

$$c_j \in h_s \qquad (D1.1)$$
$$\wedge\ c_j <_{h_s} s_i \qquad (D1.2)$$
$$\wedge\ \forall w_{k\neq j}(x_k), c_k \in h_s : c_k <_{h_s} c_j \vee s_i <_{h_s} c_k \qquad (D1.3)$$

D2 (Write Rule)

$\forall c_i, c_j \in h_s :$

$$ws(T_i) \cap ws(T_j) \neq \{\} \Rightarrow (c_i <_{h_s} s_j \vee c_j <_{h_s} s_i)$$

2.4 System

We consider a message-passing distributed system Π of n processes. We shall define our synchrony assumptions later. Following Fischer et al. [12], an execution is a sequence of steps made by one or more processes. During an execution, processes may fail by crashing. A process that does not crash is said *correct*; otherwise it is *faulty*. We note $\mathfrak{F}$ the refinement mapping [13] from executions to histories, i.e., if ρ is an execution of the system, then $\mathfrak{F}(\rho)$ is the history produced by ρ. A history h is *admissible* if there exists an execution ρ such that $h = \mathfrak{F}(\rho)$. We consider that given two sequences of steps U and V, if U precedes V in some execution ρ, then the operations implemented by U precedes (in the sense of $<_h$) the operations implemented by V in the history $\mathfrak{F}(\rho)$.

2.5 Partial Replication

A data store $\mathcal{D}$ is a finite set of tuples (x, v, i) where x is an object (data item), v a value, and $i \in \mathcal{T}$ a version. Each process in Π holds a data store

such that initially every object x has version x_0. For an object x, $\mathit{Replicas}(x)$ denotes the set of processes, or *replicas*, that hold a copy of x. We assume the existence of at least one correct replica per object in the system. For some transaction T_i, we note $\mathit{Replicas}(T_i)$ the set of processes replicating an object accessed by T_i. We make no assumption about how objects are replicated. The coordinator of T_i, denoted $\mathit{coord}(T_i) \in \mathit{Replicas}(T_i)$, is in charge of executing T_i on behalf of some client (not modeled). The coordinator does not know in advance the read set or the write set of T_i. To model this, we consider that every prefix of a transaction (followed by a terminating operation) is a transaction with the same id.

Genuine Partial Replication (GPR) aims to ensure that, when the workload is parallel, throughput scales linearly with the number of nodes [10]:

- **GPR.** For any transaction T_i, only processes that replicate objects accessed by T_i make steps to execute T_i.

2.6 Progress

The read rule of SI does not define what is the snapshot to be read. According to Adya [6], "transaction T_i's snapshot point needs not be chosen after the most recent commit when T_i started, but can be selected to be some (convenient) earlier point." To avoid that read-only transactions always observe outdated data, we add the following rule:

- **Non-trivial SI.** Consider an admissible history h such that a transaction T_i is pending in h and the next operation of T_i is a read on some object x. Note x_j the latest (according to $<_h$) committed version of x in h.[3] Let ρ be an execution with $\mathfrak{F}(\rho) = h$. If there is no concurrent conflicting transaction to T_i in h, and history $h' = h.r_i(x_j)$ is in SI, then *there exists* an execution ρ' extending ρ such that $\mathfrak{F}(\rho') = h'$.

In addition, we consider that the system provides the following guarantees:

- **Obstruction-free Updates.** If T_i is an update transaction and $\mathit{coord}(T_i)$ is correct then T_i eventually terminates. Moreover, if T_i does not write-conflict with some concurrent transaction, then T_i eventually commits.

- **Wait-free Queries.** If $\mathit{coord}(T_i)$ is correct and T_i is a read-only transaction, then transaction T_i eventually commits.

[3] According to rule (4) in our history definition, there is single latest version in any history.

3 Decomposing SI

This section defines four properties, whose conjunction is necessary and sufficient to attain SI. We later use these properties in Section 4 to prove our impossibility result.

The main difference between our characterization and the characterizations given by Elnikety et al. [7] or Adya [6] is that ours does not refer to abstract snapshot points. Instead, it *solely* relies on real transaction operations (i.e., read, write, commit and abort). As a consequence, SI is expressible in plain histories, like serializability [8], update serializability [14] or strict serializability [15]. Moreover, as we shall see in Section 4, our decomposition helps to reason about SI in a distributed system.

3.1 Cascading Aborts

Intuitively, a read-only transaction must abort if it observes the effects of some uncommitted transaction that later aborts. By guaranteeing that every version read by a transaction is committed, rules D1.1 and D1.2 of SI prevent such a situation to occur. In other words, these rules *avoid cascading aborts*. We formalize this property below:

Definition (Avoiding Cascading aborts)**.** *History h avoids cascading aborts, if for every read $r_i(x_j)$ in h, c_j precedes $r_i(x_j)$ in h. ACA denotes the set of histories that avoid cascading aborts.*

3.2 Consistent and Strictly Consistent Snapshots

Consistent and strictly consistent snapshots are defined by refining causality into a dependency relation as follows:

Definition (Dependency)**.** *Consider a history h and two transactions T_i and T_j. We note $T_i \rhd T_j$ when $r_i(x_j)$ is in h. Transaction T_i depends on transaction T_j when $T_i \rhd^* T_j$ holds.*[4] *Transaction T_i and T_j are independent if neither $T_i \rhd^* T_j$, nor $T_j \rhd^* T_i$ hold.*

This means that a transaction T_i depends on a transaction T_j if T_i reads an object modified by T_j, or such a relation holds by transitive closure. To illustrate this definition, consider history $h_3 = r_1(x_0).w_1(x_1).c_1.r_a(x_1).c_a.r_b(y_0).c_b$. In h_3, transaction T_a depends on T_1. Notice that however, even if T_1 causally precedes T_b, T_b does not depend on T_1 in h_3.

[4] We note $\mathcal{R}^*$ the transitive closure of some binary relation $\mathcal{R}$.

We now define consistent snapshots with the above dependency relation. A transaction sees a consistent snapshot iff it observes the effects of all transactions it depends on [16]. For example, consider the history $h_4 = r_1(x_0).w_1(x_1).c_1.r_2(x_1).r_2(y_0).w_2(y_2).c_2.r_a(y_2).r_a(x_0).c_a$ In this history, transaction T_a does not see a consistent snapshot: T_a depends on T_2, and T_2 also depends on T_1, but T_a does not observe the effect of T_1 (i.e., x_1). Formally, consistent snapshots are defined as follows:

Definition (Consistent snapshot)**.** *A transaction T_i in a history h observes a consistent snapshot iff, for every object x, if (i) T_i reads version x_j, (ii) T_k writes version x_k, and (iii) T_i depends on T_k, then version x_k is followed by version x_j in the version order induced by h ($x_k \ll_h x_j$). We write $h \in CONS$ when all transactions in h observe a consistent snapshot.*

SI requires that a transaction observes the committed state of the data store at some *point* in the past. This requirement is stronger than consistent snapshot. For some transaction T_i, it implies that (SCONSa) there exists a snapshot point for T_i, and (SCONSb) if transaction T_i observes the effects of transaction T_j, it must also observe the effects of all transactions that precede T_j in time. A history is called strictly consistent if both SCONSa and SCONSb hold.

For instance, consider the following history: $h_5 = r_1(x_0).w_1(x_1).c_1.r_a(x_1).r_2(y_0).w_2(y_2).c_2.r_a(y_2).c_a$. Because $r_a(x_1)$ precedes c_2 in h_5, y_2 cannot be observed when T_a takes its snapshot. As a consequence, the snapshot of transaction T_a is not strictly consistent. This issue is disallowed by SCONSa. Now, consider history $h_6 = r_1(x_0).w_1(x_1).c_1.r_2(y_0).w_2(y_2).c_2.r_a(x_0).r_a(y_2).c_a$. Since c_1 precedes c_2 in h_6, transaction T_a cannot observe T_2 but not T_1. SCONSb prevents such a situation to occur.

Definition (Strictly consistent snapshot)**.** *Snapshots in history h are strictly consistent, when for any committed transactions T_i, T_j, $T_{k \neq j}$ and T_l, the following two properties hold:*

- $\forall r_i(x_j), r_i(y_l) \in h : r_i(x_j) \not<_h c_l$ *(SCONSa)*

- $\forall r_i(x_j), r_i(y_l), w_k(x_k) \in h : c_k <_h c_l \Rightarrow c_k <_h c_j$ *(SCONSb)*

We note SCONS the set of strictly consistent histories.

Note that, as expected, strictly consistent snapshots are stronger than consistent snapshots (SCONS $\subseteq$ CONS).

3.3 Snapshot Monotonicity

In addition, SI requires what we call monotonic snapshots. For instance, although history h_7 below satisfies SCONS, this history does not belong to SI: since T_a reads $\{x_0, y_2\}$, and T_b reads $\{x_1, y_0\}$, there is no extended history that would guarantee the read rule of SI.

$$h_7 = r_a(x_0) \longrightarrow r_1(x_0).w_1(x_1).c_1 \longrightarrow r_b(x_1).c_b$$
$$r_b(y_0) \longrightarrow r_2(y_0).w_2(y_2).c_2 \longrightarrow r_a(y_2).c_a$$

SI requires monotonic snapshots. However, the underlying reason is intricate enough that some previous works [4, for instance] do not ensure this property, while claiming to be SI. In the following, we first introduce an ordering relation between snapshots, then we formalize snapshot monotonicity.

Definition (Snapshot precedence)**.** *When in a history h there exist two distinct transactions T_i and T_j such that $r_i(x_k), r_j(y_l)$ belong to h and either (i) $r_i(x_k) <_h c_l$ holds, or (ii) transaction T_l writes x and $c_k <_h c_l$ holds, then the snapshot read by T_i precedes the snapshot read by T_j, written $T_i \rightarrow T_j$.*

For more illustration, consider history $h_8 = r_1(x_0).w_1(x_1).c_1.r_2(y_0).w_2(y_2)$ $.r_a(x_1).c_2.r_b(y_2).c_a.c_b$ and history $h_9 = r_1(x_0).w_1(x_1).c_1.r_a(x_1).c_a.r_2(x_1).r_2(y_0)$ $.w_2(x_2).w_2(y_2).c_2.r_b(y_2).c_b$. In history h_8, $T_a \rightarrow T_b$ holds because $r_a(x_1)$ precedes c_2. In h_9, c_1 precedes c_2 and both T_1 and T_2 modify object x. Thus $T_a \rightarrow T_b$ also holds.

We define snapshot monotonicity using snapshot precedence as follows:

Definition (Snapshot monotonicity)**.** *Given some history h, if the relation $\rightarrow^*$ induced by h is a partial order, the snapshots in h are* monotonic*. We note MON the set of histories that satisfy this property.*

According to this definition, since both $T_a \rightarrow T_b$ and $T_b \rightarrow T_a$ hold in history h_7, this history does not belong to MON.

Non-monotonic snapshots can be observed for instance under update serializability [14], that is when queries observe consistent state, but only updates are serializable.

3.4 Write-Conflict Freedom

In contrast to serializability, which allows concurrent conflicting updates to be reordered into a sequential history, rule D2 of SI forbids two concurrent conflicting transactions from both committing. Since we assume that a transaction that writes x must have read it previously, every update transaction depends on a previous update transaction (or on the initial transaction T_0).

Therefore, under SI, concurrent conflicting transactions must be independent.

Definition (Write-Conflict Freedom)**.** *A history h is write-conflict free if two independent transactions never write to the same object. We denote by WCF the histories that satisfy this property.*

3.5 The decomposition

We now state that a history h is in SI if and only if (1) every transaction in h sees a committed state, (2) every transaction in h observes a strictly consistent snapshot, (3) relation $\rightarrow^*$ is a partial order, and (4) h is write-conflict free. We start by the three technical lemmata bellow.

Lemma 1. *Consider a history $h \in SI$ and two versions x_i and x_j of some object x. If $x_i \ll_h x_j$ holds then $T_j \rhd^* T_i$ is true.*

Proof. Assume some history $h \in$ SI such that $x_i \ll_h x_j$ holds. Let h_s be an extended history for h that satisfies rules D1 and D2. According to the model, transaction T_j first reads some version x_k, then writes version x_j.

First, assume that there is no write to x between $w_i(x_i)$ and $w_j(x_j)$. Since x belongs to $ws(T_i) \cap ws(T_j)$, rule D2 tells us that either $c_i <_{h_s} s_j$, or $c_j <_{h_s} s_i$ holds. We observe that because $x_i \ll_h x_j$ holds, it must be true that $c_i <_{h_s} s_j$. Since there is no write to x between $w_i(x_i)$ and $w_j(x_j)$, $x_k \ll x_i$ holds, or $k = i$. Observe that in the former case rule D1.3 is violated. Thus, transaction T_j reads version x_i. To obtain the general case, we apply inductively the previous reasoning. □

Lemma 2. *Let $h \in SI$ be a history, and $\mathcal{S}$ be a function such that $h_s = \mathcal{S}(h)$ satisfies D1 and D2. Consider $T_i, T_j \in h$. If $T_i \rightarrow T_j$ holds then $s_i <_{h_s} s_j$.*

Proof. Consider two transactions T_i and T_j such that the snapshot of T_i precedes the snapshot of T_j. By definition of the snapshot precedence relation, there exist $T_k, T_l \in h$ such that $r_i(x_k), r_j(y_l) \in h$ and either *(i)* $r_i(x_k) <_h c_l$, or *(ii)* $w_l(x_l) \in h$ and $c_k <_h c_l$. Let us distinguish each case:

(Case $r_i(x_k) <_h c_l$) By definition of function $\mathcal{S}$, s_i precedes $r_i(x_k)$ in h_s. From $r_j(y_l) \in h$ and rule D1.2, $c_l <_{h_s} s_j$ holds. Hence, $s_i <_{h_s} s_j$ holds.

(Case $c_k <_h c_l$) From *(i)* $r_i(x_k), w_l(x_l) \in h$, *(ii)* $c_k <_h c_l$ and *(iii)* rule D1.3, we obtain $s_i <_{h_s} c_l$. From $r_j(y_l) \in h$ and rule D1.2, $c_l <_{h_s} s_j$ holds. It follows that $s_i <_{h_s} s_j$ holds.

□

Lemma 3. *Consider a history $h \in ACA \cap CONS \cap WCF$, and two versions x_i and x_j of some object x. If $x_i \ll_h x_j$ holds then $c_i <_h c_j$.*

Proof. Since both T_i and T_j write to x and h belongs to WCF either $T_j \rhd^* T_i$ or $T_i \rhd^* T_j$ holds. We distinguish the two cases below:

(Case $T_j \rhd^* T_i$) First, assume that $T_j \rhd T_i$ holds. Note y an object such that $r_j(y_i)$ is in h. Since h belongs to ACA, $c_i <_h r_j(y_i)$ holds. Because h is an history, $r_j(y_i) <_h c_j$ must hold. Hence we obtain $c_i <_h c_j$. By a short induction, we obtain the general case.

(Case $T_i \rhd^* T_j$) Let us note x_k the version of x read by transaction T_i. From the definition of an history and since h belongs to to ACA, we know that $w_k(x_k) <_h c_k <_h r_i(x_k) <_h w_i(x_i)$ holds. As a consequence, $x_k \ll_h x_i$ is true. Since *(i)* h belongs to CONS, *(ii)* $T_i \rhd^* T_j$, and *(iii)* T_j writes to x, it must be the case that $x_j \ll_h x_k$. We deduce that $x_j \ll_h x_i$ holds; a contradiction.

□

Using these lemmata, we successively prove each inclusion.

Proposition 1. *$SI \subseteq ACA \cap SCONS \cap WCF \cap MON$*

Proof. Choose h in SI. Note $\mathcal{S}$ a function such that history $h_s = \mathcal{S}(h)$ satisfies rules D1 and D2.

($h \in$ ACA) It is immediate from rules D1.1 and D1.2.

($h \in$ WCF) Consider two independent transactions T_i and T_j modifying the same object x. By the definition of a history, $x_i \ll_h x_j$, or $x_j \ll_h x_i$ holds. Applying Lemma 1, we conclude that in the former case T_j depends on T_i, and that the converse holds in the later.

($h \in$ SCONSa) By contradiction. Assume three transactions T_i, T_j and T_l such that $r_i(x_j), r_i(y_l) \in h$ and $r_i(x_j) <_h c_l$ are true. In h_s, the snapshot point s_i of transaction T_i is placed prior to every operation of T_i in h_s. Hence, s_i precedes $r_i(x_j)$ in h_s. This implies that $s_i <_{h_s} c_l \wedge r_i(y_l) \in h_s$ holds. A contradiction to rule D1.2.

($h \in$ SCONSb) Assume for the sake of contradiction four transactions T_i, T_j, $T_{k \neq j}$ and T_l such that: $r_i(x_j), r_i(y_l)$, $w_k(x_k) \in h$, $c_k <_h c_l$ and $c_k \not<_h c_j$ are all true. Since transaction T_j and T_k both write x, by rule D2, we know that $c_j <_{h_s} c_k$ holds. Thus, $c_j <_{h_s} c_k <_{h_s} c_l$ holds. According to rule D1.2, since $r_i(y_l)$ is in h, $c_l <_{h_s} s_i$ is true. We consequently obtain that $c_j <_{h_s} c_k < s_i$ holds. A contradiction to rule D1.3.

($h \in MON$) If $\rightarrow^*$ is not a partial order, there exist transactions $T_1, \ldots, T_{n \geq 1}$ such that: $T_1 \rightarrow \ldots \rightarrow T_n \rightarrow T_1$. Applying Lemma 2, we obtain that the relation $s_1 <_{h_s} s_1$ is true. A contradiction.

□

Proposition 2. *$ACA \cap SCONS \cap WCF \cap MON \subseteq SI$*

Proof. Consider some history h in ACA∩SCONS∩WCF∩MON. If history h belongs to SI then there must exist a function $\mathcal{S}$ such that $h' = \mathcal{S}(h)$ satisfies rules D1 and D2. In what follows, we build such an extended history h', then we prove its correctness.

[Construction] Initially h' equals h. For every transaction T_i in h' we add a snapshot point s_i in h', and for every operation o_i in h', we execute the following steps:

S1. We add the order (s_i, o_i) to h'.

S2. If o_i equals $r_i(x_j)$ for some object x then

S2a. we add the order (c_j, s_i) to h',

S2b. and, for every committed transaction T_k such that $w_k(x_k)$ is in h, if $c_k <_h c_j$ does not hold then we add the order (s_i, c_k) to h'.

[Correctness] We now prove that h' is an extended history that satisfies rules D1 and D2.

- h' is an extended history.

  Observe that for every transaction T_i in h', there exists a snapshot point s_i, and that according to step S1, s_i is before all operations of transaction T_i. It remains to show that order $<_{h'}$ is acyclic. We proceed by contradiction.

  Since h is a history, it follows that any cycle formed by relation $<_{h'}$ contains a snapshot point s_i. Furthermore, according to steps S1 and S2 above, we know that for some operation $c_{j\neq i}$, relation $c_j <_{h'} s_i <^*_{h'} c_j$ holds.

  By developing relation $s_i <^*_{h'} c_j$, we obtain the following three relations. The first two relations are terminal, while the last is recursive.

  - Relation $s_i <_{h'} c_j$ holds. This relation has to be produced by step S2b. Hence, there exist operations $r_i(x_k), w_j(x_j)$ in h' such that $c_j <_h c_k$ does not hold. Observe that since h belongs to ACA ∩ CONS ∩ WCF, by Lemma 3, it must be the case that $c_k <_h c_j$ holds.
  - Relation $s_i <_{h'} o_i <^*_h c_j$ holds for some read operation o_i in T_i. (If $o_i <^*_h c_j$ with o_i a write or a terminating operation, we may consider a preceding read that satisfies the same relation.)
  - Relation $s_i <_{h'} o_i <^*_{h'} c_j$ holds for some read operation o_i in T_i, and $o_i <^*_{h'} c_j$ does not imply $o_i <^*_h c_j$. (Again if o_i is a write or a terminating operation, we may consider a preceding read that satisfies this relation.) Relation $o_i <^*_{h'} c_j$ cannot be produced by steps S1 and S2. Hence, there must exist a commit operation c_k

and a snapshot point s_l such that $s_i <_{h'} o_i <_h c_k <_{h'} s_l <^*_{h'} c_j$ holds.

From the result above, we deduce that there exist snapshot points $s_1, \ldots, s_{n\geq 1}$ and commit points $c_{k_1} \ldots c_{k_n}$ such that:

$$s_1 \prec c_{k_1} <_{h'} s_2 \prec c_{k_2} \ldots s_n \prec c_{k_n} <_{h'} s_1 \quad (1)$$

where $s_i \prec c_{k_i}$ is a shorthand for either *(i)* $s_i <_{h'} c_{k_i}$ with $r_i(x_j), w_{k_i}(x_{k_i}) \in h$ and $c_j <_h c_{k_i}$, or *(ii)* $s_i <_{h'} o_i <_h c_{k_i}$ with o_i is some read operation.

We now prove that for every i, $T_i \to T_{i+1}$ holds. Consider some i. First of all, observe that a relation $c_{k_{i-1}} < s_i$ is always produced by step S2a. Then, since relation $s_i \prec c_{k_i} <_{h'} s_{i+1}$ holds we may consider the two following cases:

- Relation $s_i <_{h'} c_{k_i} <_{h'} s_{i+1}$ holds with $r_i(x_j), w_{k_i}(x_{k_i}) \in h$ and $c_j <_h c_{k_i}$. From $c_{k_i} <_{h'} s_{i+1}$ and step S2a, there exists an object y such that $r_{i+1}(y_{k_i})$. Thus, by definition of the snapshot precedence relation, $T_i \to T_{i+1}$ holds.
- Relation $s_i \prec c_{k_i}$ equals $s_i <_{h'} o_i <_h c_{k_i}$ where o_i is some read operation of T_i, Since $c_{k_i} <_{h'} s_{i+1}$ is produced by step S2a, we know that for some object y, $r_{i+1}(y_{k_i})$ belongs to h. According to the definition of the snapshot precedence, $T_i \to T_{i+1}$ holds.

Applying the result above to Equation 1, we obtain: $T_1 \to T_2 \ldots \to T_n \to T_1$. History h violates MON, a contradiction.

- h' satisfies rules D1 and D2.

  (h' satisfies D1.1) Follows from $h \in$ ACA,
  (h' satisfies D1.2) Immediate from step S1.
  (h' satisfies D1.3) Consider three transactions T_i, T_j and T_k such that operations $r_i(x_j)$, $w_j(x_j)$ and $w_k(x_k)$ are in h. The definition of a history tells us that either $x_k \ll_h x_j$ or the converse holds. We consider the following two cases:
    (Case $x_k \ll_h x_j$) Since h belongs to ACA$\cap$CONS$\cap$WCF, Lemma 3 tells us that $c_k <_h c_j$ holds. Hence, $c_k <_{h'} c_j$ holds.
    (Case $x_j \ll_h x_k$) Applying again Lemma 3, we obtain that $c_j <_h c_k$ holds. Since $<_h$ is a partial order, then $c_j <_h c_k$ does not hold. By step S2b, the order (s_i, c_k) is in h'.
  (h' satisfies D2) Consider two conflicting transaction (T_i, T_j) in h'. Since h belongs to WCF, one of the following two cases occurs:
    (Case $T_i \rhd^* T_j$) At first glance, assume that $T_i \rhd^* T_j$ holds. By step S2a, s_i is in h' after every operation c_j such that $r_i(x_j)$ is

in h', and by step S1, s_i precedes the first operation of T_i. Thus $c_j <_{h'} s_i$ holds, and h' satisfies D2 in this case. To obtain the general case, we applying inductively the previous reasoning.

(Case $T_j \rhd^* T_i$) The proof is symmetrical to the case above, and thus omitted.

□

From the conjunction of Proposition 1 and Proposition 2, we deduce our decomposition theorem.

Theorem 1. *$SI = ACA \cap SCONS \cap MON \cap WCF$*

Notice that this decomposition is well-formed in the sense that the four properties SCONS, MON, WCF and ACA are all distinct and that no strict subset of $\{\text{SCONS}, \text{MON}, \text{WCF}, \text{ACA}\}$ attains SI.

Proposition 3. *For every $S \subsetneq \{SCONS, MON, WCF, ACA\}$, it is true that $\cap_{X \in S} X \neq SI$.*

Proof. For every set $S \subsetneq \{\text{SCONS}, \text{MON}, \text{WCF}, \text{ACA}\}$ containing three of the four properties, we exhibit below a history in $\cap_{X \in S} X \setminus \text{SI}$. Trivially, the result then holds for every S.
(SCONS∩ACA∩WCF) History h_7 in Section 3.2. (MON∩ACA∩WCF) History h_6 in Section 3.2. (SCONS∩MON∩WCF) History $r_1(x_0).w_1(x_1).r_a(x_0).c_1.c_a$.
(SCONS ∩ MON ∩ ACA) History $r_1(x_0).r_2(x_0).w_1(x_1).w_2(x_2).c_1.c_2$. □

4 The impossibility of SI with GPR

In this section, we show *three* different reasons why SI is not attainable in an asynchronous failure-free GPR system. We first state a technical lemma that characterizes histories admissible by such a system, then we prove our impossibility results.

In what follows, Π denotes some asynchronous failure-free GPR system.

Lemma 4. *Let $h = \mathfrak{F}(\rho)$ be an admissible history by Π such that a transaction T_i is pending in h. Note X the objects accessed by T_i in h. Only processes in Replicas(X) make steps to execute T_i in ρ.*

Proof. By contradiction, assume that a process $p \notin Replicas(X)$ makes steps to execute T_i in ρ. Since the prefix of a transaction is a transaction with the same id, we can consider an extension ρ' of ρ such that T_i does not execute additional operations in ρ' and $coord(T_i)$ is correct in ρ'. The progress requirements satisfied by Π imply that T_i terminates in ρ'. However, process $p \notin Replicas(X)$ makes steps to execute T_i in ρ'. A contradiction to the fact that Π is GPR. □

Our first theorem states that monotonic snapshots are not constructable in an asynchronous failure-free GPR system. The proof of this theorem relies on an indistinguishably argument that holds because objects accessed by a transaction are not known in advance.

Theorem 2. *No asynchronous failure-free GPR system implements MON.*

Proof. We proceed by contradiction. Let us consider,

- Four objects x,y,z and u such that for any two objects in $\{x, y, z, u\}$, their replica sets do not intersect;
- Four queries T_a, T_b, T_c and T_d accessing respectively $\{x, y\}$, $\{y, z\}$, $\{z, u\}$ and $\{u, x\}$; and
- Four update transactions T_1, T_2, T_3 and T_4 modifying respectively x, y, z and u.

History $r_b(y_0)$ is admissible because Π implements non-trivial SI. Since updates are obstruction-free, history $r_b(y_0).r_2(y_0).w_2(y_2).c_2$ is also admissible. Applying again that Π satisfies non-trivial SI, we obtain that $r_b(y_0).r_2(y_0).w_2(y_2).c_2.r_a(y_2).r_a(x_0)$ is admissible. Since transaction T_a is wait-free, history $h = r_b(y_0).r_2(y_0).w_2(y_2).c_2.r_a(x_0).r_a(y_2).c_a$ is admissible as well. Using a symmetrical reasoning, we conclude that $h' = r_d(u_0).r_4(u_0).w_4(u_4).c_4.r_c(z_0).r_c(u_4).c_c$ is also admissible. Let ρ and ρ' denote two sequences of steps such that $\mathfrak{F}(\rho) = h$ and $\mathfrak{F}(\rho') = h'$.

Because updates are obstruction-free, history $h_1 = h.h'.r_3(z_0).w_3(z_3).c_3$ is admissible. Since *(i)* T_b is pending in h_1, *(ii)* concurrent transactions to T_b in h_1, namely T_a and T_d, do not write-accesses z or y, and *(iii)* Π satisfies non-trivial SI, there exists a sequence of steps U_1 such that *(i)* $\rho.\rho'.U_1$ extends $\rho.\rho'$, *(ii)* $\mathfrak{F}(U_1)$ equals $r_b(z_3).c_b$ and *(iii)* history $h_1.r_b(z_3).c_b$ is admissible.

Applying a reasoning symmetrical to the one above, we note U_2 a sequence of steps such that *(i)* $\rho'.\rho.U_2$ extends $\rho'.\rho$, *(ii)* $\mathfrak{F}(U_2)$ equals $r_d(x_1).c_d$, and *(iii)* history $h'.h.r_1(x_0).w_1(x_1).c_1.r_d(x_1).c_d$ is admissible.

Because Π satisfies GPR, Lemma 4 implies that only processes in $Replicas(x) \cup Replicas(y)$ make steps in ρ. Similarly, only processes in $Replicas(u) \cup Replicas(z)$ make steps in ρ'. By hypothesis, the set $Replicas(x) \cup Replicas(y)$ and $Replicas(u) \cup Replicas(z)$ are disjoint. As a consequence, Lemma 1 in [12] tells us that $\rho.\rho'$ and $\rho.\rho'$ are undistinguishable. Since $\rho'.\rho.U_2$ is admissible, it follows that $\rho.\rho'.U_2$ is also admissible.

Executions $\rho.\rho'.U_1$ and $\rho.\rho'.U_2$ are both admissible. Because Π is GPR, only processes in $Replicas(y) \cup Replicas(z)$ execute steps in U_1. Similarly in U_2, only processes in $Replicas(x) \cup Replicas(u)$ make steps. By hypothesis, these two replica sets are disjoint. It follows that execution $\rho.\rho'.U_1.U_2$ is undistinguishable from $\rho.\rho'.U_1$ (resp. $\rho.\rho'.U_2$) for the processes in $Replicas(y) \cup Replicas(z)$ (resp. $Replicas(x) \cup Replicas(u)$). As a consequence, $\rho.\rho'.U_1.U_2$ is an execution of Π. Therefore, $\hat{h} = h'.h.r_1(x_0).w_1(x_1).c_1.r_d(x_1).c_d.r_3(z_0).w_3(z_3).c_3.r_b(z_3).c_b$ is admissible.

According to the precedence relation between snapshots, we observe that in history $\hat{h}$ we have the relation: $T_a \to T_b \to T_c \to T_d \to T_a$. Hence, this execution violates MON. Contradiction. □

Our next theorem states that SCONSb is not possible in an asynchronous failure-free GPR system. Similarly to Attiya et al. [17], our proof builds an infinite execution in which a query T_a on two objects never terminates. We first define a finite execution during which we interleave between any two consecutive steps to execute T_a, a transaction updating one of the objects read by T_a. We show that during such an execution, transaction T_a does not terminate successfully. Then, we prove that asynchrony allows us to continuously extend such an execution, contradicting the fact that queries are wait-free.

Definition (Flippable execution)**.** *Consider two distinct objects x and y, a read-only transaction T_a over both objects, and a set of updates $T_{j \in [\![1,m]\!]}$ accessing x if j is odd, and y otherwise. An execution $\rho = U_1 s_2 U_2 s_2 \ldots s_m U_m$ where,*

- *transaction T_a reads in history $h = \mathfrak{F}(\rho)$ at least (in the sense of $\ll_h$) version x_1 of x,*
- *$s_{j\in[\![1,m]\!]}$ is a single step to execute T_a by some process p_j,*
- *$U_{j\in[\![1,m]\!]}$ is the execution of transaction T_j by some set of processes Q_j, and*
- *for any j in $[\![1,m[\![$, $Q_j \cap Q_{j+1} = \{\}$ holds,*

is called flippable.

Lemma 5. *In a flippable execution ρ satisfying $\mathfrak{F}(\rho) \in$ SCONSb, query T_a does not terminate*

Proof. Let h be the history $\mathfrak{F}(\rho)$. In history h transaction T_j precedes transaction T_{j+1}, it follows that h is of the form $h = w_1(x_1).c_1.*.w_2(y_2).c_2.*\ldots$, where each symbol $*$ correspond to either no operation, or to some read operation by T_a on either object x or y.

Because ρ is flippable, transaction T_a reads at least version x_1 of object x in h. For some odd natural $j \geq 1$, let x_j denote the version of object x read by T_a. Similarly, for some even natural l, let y_l be the version of y read by T_a. Assume that $k < l$ holds. Therefore, h is of the form $h = \ldots w_j(x_j) \ldots w_l(y_l) \ldots$.

Note k the value $l+1$, and consider the step s_k made by p_k right after U_l to execute T_a. According to the definition of a flippable execution, we know that: (F1) $p_k \in Q_l \oplus p_k \in Q_k$, and (F2) $Q_l \cap Q_k = \{\}$. Consider the following cases:

(Case $p_k \in Q_k$.) Applying fact F1, execution ρ is indistinguishable from $\rho'' = \ldots U_j \ldots s_k U_l U_k \ldots$. Then applying fact F2, ρ is indistinguishable from execution $\rho' = \ldots U_j \ldots s_k U_k U_l \ldots$.

(Case $p_k \in Q_l$.) With a similar reasoning, we obtain that ρ is indistinguishable from $\rho' = \ldots U_j \ldots U_k U_l s_k \ldots$.

(Case $p_k \notin Q_l \cup Q_k$.) This case reduces to any of the two above cases.

Note h' the history $\mathfrak{F}(\rho')$. In history ρ', both $w_k(x_k) <_{h'} w_l(y_l)$ and $x_j \ll_{h'} x_k$ hold. Besides, operations $r_i(x_j)$, $r_i(y_l)$ and $w_k(x_k)$ all belong to h'. Thus, history h' does not belong to SCONSb, or transaction T_a does not commit in h'. Since ρ' is indistinguishable from ρ, history h' is admissible. It follows that T_a does not commit in h'. (The case $k > l$ follows a symmetrical reasoning to the case $l > k$ we considered previously.) □

Theorem 3. *No asynchronous failure-free GPR system implements SCONSb.*

Proof. Consider some read-only transaction T_a, two distinct objects x and y read by T_a, and assume that *Replicas*(x) and *Replicas*(y) are disjoint.

We reason by contradiction, exhibiting an admissible execution during which transaction T_a never terminates. This execution is constructed as follows:

Let $\mathcal{P}$ be an initially empty FIFO list, and consider an initially empty execution ρ. Start executing T_a by *coord*(T_a). Repeat for all $i \geq 1$. Add to $\mathcal{P}$ (in some arbitrary order) the processes that have to execute a step for T_a. Pop from $\mathcal{P}$ the next process p to execute a step for T_a. Extend ρ with step s_i, the next step of p. Let T_i be an update of x, if i is even, and y otherwise. Start the execution of transaction T_i. Since no transaction are concurrent, updates are obstruction-free and the system is genuine, there exists an extension $\rho' = \rho \circ U_i$ during which T_i commits and such that in U_i, only processes in *Replicas*(x), if i is odd, or in *Replicas*(y) otherwise, execute steps. Assign to ρ the value of ρ'.

By construction, execution ρ is flippable. Hence, Lemma 5 tells us that transaction T_a does not terminate in this run. Since every process in $\mathcal{P}$ eventually make a step in ρ, ρ is fair. Because there is no synchrony assumptions, this execution is admissible. In ρ, transaction T_a never commits. Contradiction. □

SCONSa disallows some real time orderings between operations accessing different objects. Our last theorem shows that this property cannot be maintained under GPR.

Theorem 4. *No asynchronous failure-free GPR system implements SCONSa.*

Proof. Consider two distinct objects x and y such that *Replicas*(x) and *Replicas*(y) are disjoint. Let T_1 be an update transaction accessing x, T_2 be an update on y, and T_a be a read-only transaction accessing both objects.

Since updates are obstruction-free, history $h_1 = r_1(x_0).w_1(x_1).c_1$ is admissible. Let U_1 be a sequence of steps such that $\mathfrak{F}(U_1) = r_1(x_0).w_1(x_1).c_1$.

The system Π satisfies non-trivial SI. Consequently, there exists an extension $U_1.U_a$ of U_1 such that $\mathfrak{F}(U_a) = r_a(x_1)$, and history $h_2 = h_1.\mathfrak{F}(U_a)$ is admissible.

Then, since the system supports obstruction-free updates, there exists an extension $U_1.U_a.U_2$ such that $\mathfrak{F}(U_2) = r_2(y_0).w_2(y_2)$, and history $h_3 = h_2.\mathfrak{F}(U_2)$ is admissible.

Finally, we may extend $U_1.U_a.U_2$ by a sequence of steps V_a such that $\mathfrak{F}(V_a) = r_a(y_2).c_2$, and $h_3.\mathfrak{F}(V_a)$ is admissible. Such an extension is possible since Π satisfies both non-trivial SI and wait-free queries.

Consider the execution $U_1.U_a.U_2$. Applying Lemma 4, only processes in $Replicas(x)$ make steps to execute T_a in this execution. Since $Replicas(x)$ and $Replicas(y)$ are disjoint, it follows that $U_1.U_a.U_2$ is indistinguishable from $U_1.U_2.U_a$. As a consequence, the execution $U_1.U_2.U_a.V_a$ is admissible. Notice that $\mathfrak{F}(U_1.U_2.U_a.V_a)$ equals $r_1(x_0).w_1(x_1).c_1.r_a(x_1).r_2(y_0).w_2(y_2).r_a(y_2).c_2$. This history is not in SCONSa. Contradiction. □

As a result of the above, no asynchronous system, even if it is failure-free, can support both GPR and SI. This fact strongly hinders the usage of SI at large scale. In particular, this consistency criterion cannot be implemented under GPR even if the system is augmented with failure detectors [18], a common approach to model partial synchrony.

5 Non-Monotonic Snapshot Isolation

We just showed that the SI requirements of strictly consistent (SCONS) and monotonic (MON) snapshots hurt scalability, as they are impossible with GPR. To overcome the impossibility, this section presents a slightly weaker criterion, called Non-Monotonic Snapshot Isolation (NMSI).

NMSI retains the most important properties of SI, namely snapshots are consistent, a read-only transaction can commit locally without synchronization, and two concurrent conflicting updates do not both commit. However, NMSI allows non-strict, non-monotonic snapshots. For instance, history h_7 in Section 3.3, which is not in SI, is allowed by NMSI. Formally, we define NMSI as follows:

Definition (Non-Monotonic Snapshot Isolation)**.** *A history h is in NMSI iff h belongs to $ACA \cap CONS \cap WCF$.*

To clarify our understanding of NMSI, Table 1 compares it to well-known approaches, based on the anomalies an application might observe. In addition to the classical anomalies [6, 8] (dirty reads, non-repeatable reads, read skew, dirty writes, lost updates, and write skew), we also consider the following: (Non-Monotonic Snapshots) snapshots taken by transactions are not monotonically ordered, and (Real-Time Causality Violation) a transaction T_2 observes the effect of some transaction T_1, but does not observe the effect of all the transactions that precede (in real time) T_1.

	Strict Serializ-ablity [15]	Serializablity [8]	Update Serializ-ablity [14]	Snapshot Isola-tion	NMSI
Dirty Reads	x	x	x	x	x
Non-repeatable Reads	x	x	x	x	x
Read Skew	x	x	x	x	x
Dirty Writes	x	x	x	x	x
Lost Updates	x	x	x	x	x
Write Skew	x	x	x	-	-
Non-Monotonic Snapshots	x	x	-	x	-
Real-time Causality Violation	x	-	-	x	-

Table 1: Comparing consistency criterion by their anomalies (x: disallowed)

Write Skew, the classical anomaly of SI, is observable under NMSI. (Cahill et al. [19] show how an application can easily avoid it.) Because NMSI does not ensure SCONSb, it suffers the Real-Time Causality Violation anomaly.

Note that it is not new, as it occurs with serializability as well; this argues that it is not considered a problem in practice. Non-Monotonic Snapshots occur both under NMSI and update serializability. Following Garcia-Molina and Wiederhold [14], we believe that this is a small price to pay for improved performance.

6 Protocol

We now describe Jessy, a scalable transactional system that implements NMSI and ensures GPR. Our description is divided in two parts: an execution protocol and a termination protocol. The former is in charge of executing transactions, and the latter guarantees that every execution produces an NMSI history.

Reading some object x fetches it (using asynchronous messages) from some replica. To ensure a consistent snapshot, Jessy makes use of novel data type called *dependence vectors*. We first give an overview of Jessy, then we introduce dependence vectors and detail the two protocols.

6.1 Overview

A transaction T_i executed by Jessy can be in one of the following four states at some process:

- *Executing*: Each non-terminating operation $o_i(x)$ in T_i is executed optimistically (i.e., without synchronisation with other replicas) at the transaction coordinator $coord(T_i)$. If $o_i(x)$ is a read, $coord(T_i)$ returns the corresponding value, fetched either from the local or a remote replica. If $o_i(x)$ is a write, $coord(T_i)$ stores the corresponding update value in a local buffer, enabling *(i)* subsequent reads to observe the modification, and *(ii)* a subsequent commit to send the write-set to remote replicas.
- *Submitted*: Once all the read and write operations of T_i have executed, the transaction terminates, and the coordinator submits it to the termination protocol. The protocol applies a certification test on T_i to enforce NMSI. This test ensures that if two concurrent conflicting update transactions terminate, one of them aborts.
- *Committed*/*Aborted*: When T_i enters the *Committed* state at $r \in Replicas(T_i)$, its updates (if any) are applied to the local data store. If T_i aborts, T_i enters the *Aborted* state.

6.2 Dependence vectors

To compute consistent snapshots, Jessy makes use of a novel data type called *dependence vectors*. Each version of each object is assigned its own dependence vector. The dependence vector of some version x_i reflects all the versions read by T_i, or read by transactions that precede T_i, as well as the writes of T_i itself:

Definition (Dependence Vector)**.** *A dependence vector is a function DV that maps every read (or write) operation $o(x)$ in a history h to a vector $DV(o(x)) \in \mathbb{N}^{|Objects|}$ such that:*

$$\begin{aligned} DV(r_i(x_0)) &= 0^{|Objects|} \\ DV(r_i(x_j)) &= DV(w_j(x_j)) \\ DV(w_i(x_i)) &= max\ \{DV(r_i(y_j)) : y_j \in rs(T_i)\} \\ &\quad +\ \Sigma_{z_i \in ws(T_i)}\ 1_z \end{aligned}$$

where max $\mathcal{V}$ is the vector containing for each dimension z, the maximal z component in the set of vectors $\mathcal{V}$, and 1_z is the vector that equals 1 on dimension z and 0 elsewhere.

To illustrate this definition, consider history h_{10} below. In this history, transactions T_1 and T_2 update objects x and y respectively, while transaction T_3 reads x, then updates y. The dependence vector of x_1 equals $\langle 1, 0\rangle$, and

$$\begin{aligned} h_{10} = & r_1(x_0).w_1(x_1).c_1 \searrow \\ & \qquad\qquad\qquad\quad r_3(x_1).r_3(y_2).w_3(y_3).c_3 \\ & r_2(y_0).w_2(y_2).c_2 \nearrow \end{aligned}$$

of y_1 equals $\langle 0, 1\rangle$. Since transaction T_3 reads x then updates y, this implies that dependence vector of y_3 equals $\langle 1, 2\rangle$.

Using dependence vectors, Jessy can compute a consistent snapshot for transaction T_i by ensuring that every pair of versions x_l and y_j read by T_i satisfy the following rule: $DV(r_i(x_l))[x] \geq DV(r_i(y_j))[x]$. We assess below that such a claim holds.

First of all, we show in Lemma 6 that if transaction T_i depends on transaction T_j then the dependence vector of any object written by T_i is greater than the dependence vector of any object written by T_j.

Lemma 6. *Consider a history h in WCF, and two transactions T_i and T_j in h. Then,*

$$T_i \rhd^* T_j \Leftrightarrow \forall x, y \in Objects : \forall w(x), w(y) \in T_i \times T_j : DV(w_i(x_i)) > DV(w_j(y_j))$$

Proof. The proof goes as follows:

- ($\Rightarrow$) First consider that $T_i \rhd T_j$ holds. By definition of relation $\rhd$, we know that for some object z, operations $r_i(z_j)$ and $w_j(z_j)$ are in h. According to definition of function DV we have: $DV(w_i(x_i)) \geq DV(r_i(z_j)) + 1_x$. Besides, always according to the definition of DV, it is true that the following equalities hold: $DV(r_i(z_j)) = DV(w_j(z_j)) = DV(w_j(y_j))$. Thus, we have: $DV(w_i(x_i)) > DV(w_j(y_j))$. The general case $T_i \rhd^* T_j$ is obtained by applying inductively the previous reasoning.

- ($\Leftarrow$) From the definition of function DV, it must be the case that $r_i(y'_{j'})$ is in h with $j' \neq 0$. We then consider the following two cases: (Case $j' = j$) By definition of relation $\rhd$, $T_i \rhd T_j$ holds. (Case $j' \neq j$) By construction, we have that: $T_i \rhd T_{j'}$. By definition of function DV, we have that $DV(r_{j'}(y_{j'})) = DV(w_{j'}(y_{j'}))$. Since $DV(w_i(x_i)) > DV(w_j(y_j))$ holds, $DV(w_{j'}(y_{j'}))[y] \geq DV(w_j(x_j))[y]$ is true. Both transactions T_j and $T_{j'}$ write y. Since h belongs to WCF, it must be the case that either $T_j \rhd^* T_{j'}$ or that $T_{j'} \rhd^* T_j$ holds. If $T_j \rhd^* T_{j'}$ holds, then we just proved that $DV(w_j(y_j)) > DV(w_{j'}(y_{j'}))$ is true. A contradiction. Hence necessarily $T_{j'} \rhd^* T_j$ holds. From which we conclude that $T_i \rhd^* T_j$ is true.

□

The following theorem shows that dependence vectors enable taking consistent snapshots.

Theorem 5. *Consider a history h in WCF, and a transaction T_i in h. Transaction T_i sees a consistent snapshot during h if an only if, for every pair of versions x_l and y_j read by T_i, $DV(r_i(x_l))[x] \geq DV(r_i(y_j))[x]$ holds.*

Proof. The proof goes as follows:

- ($\Rightarrow$) By contradiction. Assume the existence of two versions x_l and y_j in the snapshot of T_i such that $DV(r_i(x_l))[x] < DV(r_i(y_j))[x]$ holds. By definition of function DV, we have $DV(r_i(x_l)) = DV(w_l(x_l))$ and $DV(r_i(y_j)) = DV(w_j(y_j))$. Hence, $DV(w_l(x_l))[x] < DV(w_j(y_j))[x]$ holds. Again from the definition of function DV, there exists a transaction $T_{k\neq 0}$ writing on x such that (i) $DV(w_j(y_j)) \geq DV(w_k(x_k))$ and (ii) $DV(w_j(y_j))[x] = DV(w_k(x_k))[x]$. Applying Lemma 6 to (i), we obtain $T_j \rhd^* T_k$. From which we deduce that $T_i \rhd^* T_k$. Now since both transactions T_l and T_k write x and h belongs to WCF, $T_l \rhd^* T_k$ or $T_k \rhd^* T_l$ holds. From (ii) and $DV(w_l(x_l))[x] < DV(w_j(y_j))[x]$, we deduce that $DV(w_l(x_l))[x] < DV(w_k(x_k))[x]$. As a consequence of Lemma 6, $T_k \rhd^* T_l$ holds. Hence $x_l \ll_h x_k$. But $T_i \rhd^* T_k$ and $r_i(x_l)$ is in h. It follows that T_i does not read a consistent snapshot. Contradiction.
- ($\Leftarrow$) By contradiction. Assume that there exists an object x and a transaction T_k on which T_i depends such that T_i reads version x_j, T_k writes version x_k, and $x_j \ll_h x_k$. First of all, since h is in WCF, one can easily show that $T_k \rhd^* T_j$. Since $T_k \rhd^* T_j$, Lemma 6 tells us that $DV(w_k(x_k)) > DV(w_j(x_j))$ holds. Since $T_i \rhd^* T_k$ holds, a short induction on the definition of function DV tells us that $DV(r_i(x_j))[x] \geq$

$DV(w_k(x_k))[x]$ is true. Hence, $DV(r_i(x_j))[x] \geq DV(w_k(x_k))[x] > DV(w_j(x_j))[x] = DV(r_i(x_j))[x]$. Contradiction.

□

Although dependence vectors may be large, in the common case they are sparse, and thus can be implemented efficiently. Furthermore, the size can be reduced, and dependence approximated safely, by coarsening the granularity, grouping objects into disjoint partitions and serialising updates to a group as if it was a single larger object [20].

6.3 Execution Protocol

Algorithm 1 describes the execution protocol in pseudocode. Logically, it can be divided into two parts: action *readResolve*(), executed by some process, reads an object replicated at that process, in a consistent snapshot; and the transaction coordinator $coord(T_i)$ performs actions *execute*() to execute the transaction and to buffer the updates in $up(T_i)$.

The variables of the execution protocol are: *store*, the local data store; *submitted* contains locally-submitted transactions; and *committed* (respectively *aborted*) stores committed (respectively aborted) transactions. We use the shorthand *decided* for *committed* ∪ *aborted*.

Upon a read request for x, $coord(T_i)$ checks against $up(T_i)$ if x has been previously updated by the same transaction; if so, it returns the corresponding value (Line 15). Otherwise, $coord(T_i)$ sends an (asynchronous) READ_RESOLVE request to the processes that replicate x (Lines 18–19). When a process receives a READ_RESOLVE request for object x that it replicates, it returns a version of x which complies with Theorem 5 (Lines 5–9).

Upon a write request of T_i, the process buffers the update value in $up(T_i)$ (Line 12). The updates of T_i will be sent to all replicas holding an object that is modified inside T_i during commitment.

When transaction T_i terminates, T_i is submitted to the termination protocol (Line 22) The execution protocol then waits until T_i is committed or aborted, and returns the outcome.

6.4 Termination Protocol

Algorithm 2 depicts the code of the termination protocol of Jessy. It accesses the same four variables *store*, *submitted* and *committed*, along with a FIFO queue named *queue*. The termination protocol makes use of an atomic multicast primitive. In order to satisfy GPR, this primitive is genuine. This requires that either one can partition replicas into a set of non-intersecting

Algorithm 1 Execution Protocol of Jessy

```
 1: Variables:
 2:     store, submitted, committed, aborted
 3:
 4: readResolve(x, T_i)
 5:     pre:  received ⟨READ_RESOLVE, T_i, x⟩ from q
 6:           ∃(x, v, l) ∈ store : ∀y ∈ rs(T_i) :
 7:               DV(w_l(x_l))[x] ≥ DV(r_i(y_j))[x]
 8:               ∧ DV(w_l(x_l))[y] ≤ DV(r_i(y_j))[y]
 9:      eff:  send ⟨READ_RESOLVE, T_i, x, v⟩ to q
10:
11: execute(WRITE, x, v, T_i)
12:      eff:  up(T_i) ← up(T_i) ∪ {(x, v, i)}
13:
14: execute(READ, x, T_i)
15:      eff:  if ∃(x, v, i) ∈ up(T_i) then return v
16:            else
17:              send ⟨READ_RESOLVE, T_i, x⟩ to Replicas(x)
18:              wait until received ⟨READ_RESOLVE, T_i, x, v⟩
19:              return v
20:
21: execute(TERMINATE, T_i)
22:      eff:  submitted ← submitted ∪ {T_i}
23:            wait until T_i ∈ decided
24:            if T_i ∈ committed then return COMMIT
25:            return ABORT
26:
```

groups and an eventual leader oracle is available in each group, or that a system-wide *reliable* failure detector is available [21].

To terminate an update transaction T_i, $coord(T_i)$ atomic multicasts T_i to every process that holds an object written by T_i. Every process p in $WReplicas(T_i)$ that atomic delivers T_i certifies it by calling function $certify(T_i)$ (Line 16). This function returns *true* at process p, iff for every transaction T_j committed prior to T_i at p, if T_j write-conflicts with T_i, then T_i depends on T_j. Formally:

$$certify(T_i) \triangleq \forall T_j \in committed : ws(T_i) \cap ws(T_j) \neq \{\} \Rightarrow T_i \rhd^* T_j$$

Under partial replication, a process p might store only a subset of the objects written by T_i, in which case p does not have enough information to decide on the outcome of T_i. Therefore, we introduce a voting phase where replicas of the objects written by T_i send the result of their certification test in a VOTE message to every process in $WReplicas(T_i) \cup \{coord(T_i)\}$ (Lines 17–18).

A replica q can safely decide on the outcome of T_i when it receives votes from a *voting quorum* Q for T_i. A voting quorum Q for T_i is a set of replicas such that for every object x written by T_i, the set Q contains at least one of the processes replicating x. Formally, a set of processes is a voting quorum for T_i iff it belongs to $vquorum(T_i)$, defined as follows:

$$vquorum(T_i) \triangleq \{Q \subseteq \Pi | \forall x \in ws(T_i) : \exists j \in Q \ \cap \ Replicas(x)\}$$

A process p makes use of the following (three-values) predicate $outcome(T_i)$ to determine whether some transaction T_i commits, or not:

$$\begin{array}{rl} outcome(T_i) \triangleq & \textbf{if } T_i \in submitted \wedge ws(T_i) = \{\} \\ & \quad \textbf{then } true \\ & \textbf{else if } \forall Q \in vquorum(T_i), \exists q \in Q, \\ & \qquad \neg received\ \langle \text{VOTE}, T, \_ \rangle\ from\ q \\ & \quad \textbf{then } \bot \\ & \textbf{if } \exists Q \in vquorum(T_i), \forall q \in Q, \\ & \quad received\ \langle \text{VOTE}, T, true \rangle\ from\ q \\ & \quad \textbf{then } true \\ & \textbf{else } false \end{array}$$

To commit transaction T_i, process p first applies T_i's updates to its local data store, then p adds T_i to variable *committed* (Lines 21–24). If instead T_i aborts, p adds T_i to *aborted* (Lines 27–28).

6.5 Correctness of Jessy

We now sketch a correctness proof of Jessy. First, we establish that Jessy generates NMSI histories in Proposition 4. Then, Proposition 5 shows that read-only transactions are wait-free, Proposition 6 proves that updates are obstruction-free. Finally, we show that Jessy satisfies a non-trivial progress condition in Proposition 7.

6.5.1 Safety

Proposition 4. *Every history admissible by Jessy belongs to NMSI.*

Proof. First of all, we observe that in Jessy transactions always reads from committed transactions (Line 19 in Algorithm 1). As a consequence, Jessy ensures ACA.

Consider now that a transaction T_i read versions x_l and y_j of objects x and y during some execution of Jessy. The two objects where read sequentially;

Algorithm 2 Termination Protocol of Jessy

```
 1: Variables:
 2:     store, submitted, committed, aborted, queue
 3:
 4: doSubmit(T_i)
 5:     pre:  T_i ∈ submitted
 6:           ws(T_i) ≠ {}
 7:     eff:  AM-Cast(T_i) to WReplicas(T_i)
 8:
 9: getSubmission(T_i)
10:     pre:  T_i = AM-Deliver()
11:     eff:  queue ← queue ∘ ⟨T_i⟩
12:
13: doCertify(T_i)
14:     pre:  T_i ∈ queue \ decided
15:           ∀T_j ∈ queue, T_j <_queue T_i ⇒ T_j ∈ decided
16:     eff:  v ← certify(T_i)
17:           send ⟨VOTE, T_i, v⟩ to WReplicas(T_i)
18:                               ∪ {coord(T_i)}
19:
20: commit(T_i)
21:     pre:  outcome(T_i)
22:     eff:  foreach (x, v, i) in up(T_i) do
23:             if x ∈ store then store ← store ∪ {(x, v, i)}
24:           committed ← committed ∪ {T_i}
25:
26: abort(T_i)
27:     pre:  ¬outcome(T_i)
28:     eff:  aborted ← aborted ∪ {T_i}
29:
```

let us say x before y. According to the code of the execution module, there exists a process p replicating y such p executes Lines 6–9 in Algorithm 1. It follows that $DV(w_l(x_l))[x] \geq DV(r_i(y_j))[x]$ holds. Since $DV(r_i(x_l))$ equals $DV(w_l(x_l))$, we know that $DV(r_i(x_l))[x] \geq DV(r_i(y_j))[x]$ holds. Similarly, we deduce that $DV(r_i(y_j))[y] \geq DV(r_i(x_l))[y]$ holds. Theorem 5 tells us that in such a case T_i has read a consistent snapshot.

It remains to show that that the histories generated by Jessy are write-conflict free. To this goal, we consider two independent write-conflicting transactions T_i and T_j, and we assume for the sake of contradiction that they both commit. We note p_i (resp. p_j) the coordinator of T_i (resp. T_j). Since T_i and T_j write-conflict, there exists some object x in $ws(T_i) \cap ws(T_j)$. One can show, using the preconditions of *doCertify*(), the monotonicity of variable *queue*, and the properties of atomic multicast, that (F1) for any two replicas p and q of x, denoting $committed_p$ (resp. $committed_q$) the set $\{T_j \in committed$:

$x \in ws(T_j)\}$, at the time p (resp. q) executes $outcome(T_i)$, it is true that $committed_p$ equals $committed_q$. According to Line 21 of Algorithm 2 and the definition of function $outcome()$, p_i (respectively p_j) received a positive VOTE message from some process q_i (resp. q_j) replicating x. Observe that T_i (resp. T_j) is in variable *queue* at process q_i (resp. q_j) before this process sends its VOTE message. It follows that either (1) at the time q_i sends its VOTE message, $T_j <_{queue} T_i$ holds, or (2) at the time q_j sends its VOTE message, $T_i <_{queue} T_j$ holds. Assume that case (1) holds (the reasoning for case (2) is symmetrical). From the precondition at Line 15 in Algorithm 2 we know that process q_i must wait that T_j is decided before casting a vote for T_i. From fact F1 above, we easily deduce that T_j is committed at process q_i. Hence, $certify(T_i)$ returns *false* at process q_i; a contradiction. □

6.5.2 Liveness

Lemma 7. *For every transaction T_i, if T_i is submitted at $coord(T_i)$ and $coord(T_i)$ is correct, T_i eventually terminates at every correct process in $Replicas(T_i) \cup coord(T_i)$.*

Proof. According to the termination, validity and uniform agreement properties of atomic multicast, transaction T_i is delivered at every correct process in $WReplicas(T_i)$. It is then enqueued in variable *queue* (Lines 10–11 in Algorithm 2).

Because *queue* is FIFO, processes dequeue transactions in the order they deliver them (Lines 14–15). The uniform prefix order and acyclicity properties of genuine atomic multicast ensure that no two processes in the system wait for a vote from each other. It follows that every correct replicas in $WReplicas(T_i)$ eventually dequeue T_i, and send the outcome of function $certify(T_i)$ to other replicas in $WReplicas(T_i) \cup coord(T_i)$ (Lines 16–18).

Since there exists at least one correct replica for each object modified by T_i eventually every correct process in $WReplicas(T_i) \cup coord(T_i)$ collects enough votes to decide upon the outcome of T_i (definition of predicate $outcome(T_i)$). □

Lemma 8. *For every transaction T_i, if $coord(T_i)$ executes T_i and $coord(T_i)$ is correct, then eventually T_i is submitted to the termination protocol at $coord(T_i)$.*

Proof. Transaction T_i executes all its write operation locally at $coord(T_i)$. Upon receiving a read request for an object x, if x was modified previously by T_i, the corresponding value is returned. Otherwise, the transaction send a READ_RESOLVE request to $Replicas(x)$. Thus to prove the lemma, we have

to show that eventually one of the replicas returns a value of x back to the coordinator.

According to the model, there exists one correct process that replicates object x. In what follows, we name it p. Upon receiving the READ_RESOLVE message from $coord(T_i)$, process p tries to return a value of x such that preconditions at Lines 6–8 hold.

By contradiction, assume that replica p never finds such a version. This means that the following predicate is always true:

$$\begin{aligned}\forall y \in rs(T_i), \forall (x, v, l) \in store : \\ DV(w_l(x_l))[x] < DV(r_i(y_j))[x] \\ \vee\ DV(w_l(x_l))[y] > DV(r_i(y_j))[y]\end{aligned}$$

We consider two cases:

1. $DV(w_l(x_l))[x] < DV(r_i(y_j))[x]$ forever holds.
   According to the definition of function DV, there exists a version $x_{k\neq 0}$ of object x written by some transaction T_k upon which transaction T_i depends, and such that $DV(w_k(x_k))[x] = DV(r_i(y_j))[x]$. Because transaction T_k committed at some site, Lemma 7 and Proposition 4 tell us that eventually T_k commits at process p. Contradiction.

2. $DV(w_l(x_l))[y] > DV(r_i(y_j))[y]$ forever holds.
   This case is symmetric to the case above, and thus omitted.

□

Proposition 5. *Read-only transactions are wait-free.*

Proof. Consider some read-only transaction T_i and assume that $coord(T_i)$ is correct, Lemma 8 tells us that T_i is eventually submitted at $coord(T_i)$

According to the definition of predicate *outcome*, $outcome(T_i)$ always equals true. Hence, the precondition at Line 21 in Algorithm 2 is always true, whereas precondition at Line 27 is always false. It follows that T_i eventually commits. □

Proposition 6. *Updates transactions are obstruction-free.*

Proof. Consider some update transaction T_i such that $coord(T_i)$ is correct. From the conjunction of Lemmata 7 and 8, transaction T_i eventually terminates.

Then, assume that at the time T_i starts its execution, every transaction write-conflicting with T_i has terminated. This implies that T_i depends on every write-conflicting transactions. Thus, the outcome of $certify(T_i)$ always equals true. Hence, transaction T_i eventually commits. □

6.5.3 Non-triviality

In this section, we show that Jessy implements the following progress condition:

- **Non-trivial NMSI.** Consider an admissible history h such that a transaction T_i is pending in h, and the next operation of T_i is a read on some object x. Note x_j the latest committed version of x in h. Let ρ be an execution with $\mathfrak{F}(\rho) = h$. If there is no concurrent conflicting transaction to T_i in h, and history $h' = h.r_i(x_j)$ is in NMSI, then there exists an execution ρ' extending ρ such that $\mathfrak{F}(\rho') = h'$.

To this goal, we consider that (P1) when a process resolves a remote read request over some object x at Lines 7–8, it always returns the greatest version of x (in the sense of function DV) stored in variable *store*. Since Jessy produces histories satisfying WCF, there a single such version.

Proposition 7. *Consider an admissible history h containing a pending transaction T_i such that the next operation of T_i is a read over some object x. Note x_j the latest committed version of x in h. If history $h' = h.r_i(x_j)$ belongs to NMSI then history h' is admissible.*

Proof. Consider a replica p of x storing version x_j, and assume from now that process p always answers first to a remote read request from $coord(T_i)$ over x. Since history $h.r_i(x_j)$ is in NMSI, it belongs to CONS. As a consequence, T_i reads a consistent snapshot in $h.r_j(x_i)$. According to Theorem 5, it follows that both $DV(r_i(x_j))[x] \geq DV(r_i(y_k))[x]$ and $DV(r_i(x_j))[y] \leq DV(r_i(y_k))[y]$ hold. According to the preconditions of operation $readResolve(x, T_i)$ and the property P1 above, process p returns version x_j to $coord(T_i)$. □

7 Related Work

Table 2 compares different partial replication protocols, in terms of time and message complexity (from the coordinator's perspective), when executing a transaction with r_r remote reads and w_r remote writes. A transaction can be of the following three types: a read-only transaction, a local update transaction (the coordinator replicates all the objects accessed by the transaction), or a global update transaction (some object is not available at the coordinator).

Several protocols solve particular instances of the partial replication problem. Some assume that a correct replica holds all the data accessed by a transaction [22, 23] . Others consider that data can be partitioned into conflict sets [24], or that always aborting concurrent conflicting transactions [25] is reasonable. Hereafter, we review in details algorithms that do not make such an assumption.

P-Store [10] is a genuine partial replication algorithm that ensures SER by leveraging genuine atomic multicast. Like in Jessy, read operations are performed optimistically at some replicas and update operations are applied at commit time. However, unlike Jessy, P-Store certifies read-only transactions as well.

A few algorithms [1, 2] offer partial replication with SI semantics. At the start of a transaction T_i, the algorithm of Armendáriz-Iñigo et al. [2] atomically broadcasts T_i to all processes. This message defines the consistent snapshot of T_i. If T_i is an update transaction, T_i's write set is atomic broadcast to all processes at commit time and each process independently certifies it. The algorithm of Serrano et al. [1] executes a dummy transaction after each commit. As the commit of a transaction is known by all processes, a dummy transaction identifies a snapshot point. This avoids the cost of the start message. As a consequence of the impossibility result depicted in Section 4, none of these algorithms is genuine.

Walter is a transactional key-value store proposed by Sovran et al. [9] that supports Parallel Snapshot Isolation (PSI). PSI is somewhat similar to NMSI; in particular, PSI snapshots are non-monotonic. However, PSI is stronger than NMSI, as it enforces SCONSa: NMSI allows reading versions of objects that have committed after the start of the transaction, as long as it is consistent. On the contrary in PSI, an operation has to read the most recent versions at the time the transaction starts. Enforcing SCONSa does not preclude any anomaly, and it increases the probability that a write skew, or a conflict between concurrent writes occurs. To ensure PSI, Walter relies on a single master replication schema per object and 2PC. After the transaction commits, it is propagated to all processes in the system in the

Algorithm	Cons.	Genuine?	Multi-Master?	Message Complexity	Time complexity: Read-only	Time complexity: Global Update	Time complexity: Local Update
P-Store [10]	SER	yes	yes	$O(n^2)$	$(r_r \times 2\Delta) + 4\Delta$	$(r_r \times 2\Delta) + 5\Delta$	4Δ
GMU [26]	US	yes	yes	$O(n^2)$	$r_r \times 2\Delta$	$(r_r \times 2\Delta) + 2\Delta$	2Δ
SIPRe[2]	SI	no	yes	$O(N^2)$	$(r_r \times 2\Delta) + 3\Delta$	$(r_r + w_r) \times 2\Delta + 6\Delta$	6Δ
Serrano[1]	SI	no	yes	$O(N^2)$	$r_r \times 2\Delta$	$(r_r + w_r) \times 2\Delta + 3\Delta$	3Δ
Walter [9]	PSI	no	no	$O(N)$	$r_r \times 2\Delta$	$(r_r \times 2\Delta) + 2\Delta$	$2\Delta \mid 0$
Jessy	NMSI	yes	yes	$O({w_r}^2)$	$r_r \times 2\Delta$	$(r_r \times 2\Delta) + 5\Delta$	4Δ

Message complexity: number of messages sent on behalf of transaction. Time complexity: delay for executing a transaction. N: number of replicas; n: number of replicas involved in transaction; Δ: message latency between replicas; r_r: number of remote reads; w_r: number of remote writes. The latency of atomic broadcast (resp. atomic multicast) is considered 3Δ (resp 4Δ) during solo step execution [21].

Table 2: Comparison of partial replication protocols

background before it becomes visible.

More recently, Peluso et al. [26] proposed GMU, an algorithm that supports an extended form of update serializability. GMU relies on vector clocks to read consistent snapshots. At commit time, both GMU and Walter use locks to commit transactions. Because locks are not ordered before voting (contrary to P-Store and Jessy), these algorithms are subjected to the occurrence of distributed deadlocks, and scalability problems leading to poor performance for global update transactions [27, 28].

8 Conclusion

Partial replication and genuineness are two key factors of scalability in replicated systems. This paper shows that ensuring snapshot isolation (SI) in a genuine partial replication (GPR) system is impossible. To state this impossibility result, we introduce four properties whose conjunction is equivalent to SI. We show that two of them, namely snapshot monotonicity and strictly consistent snapshots cannot be ensured. This implies that (provided wait-free queries and obstruction-free updates) consistency criterion stronger than SI, e.g., strict serializability or opacity [29], are not attainable neither.. In the case of opacity, this answers a recent question raised by Peluso et al. [30].

To side step the incompatibility of SI with GPR, we propose a novel consistency criterion named NMSI. NMSI prunes most anomalies disallowed by SI, while providing guarantees close to SI: transactions under NMSI always observe consistent snapshots and two write-conflicting concurrent updates never both commit.

The last contribution of this paper is Jessy, a genuine partial replication protocol that supports NMSI. To read consistent partial snapshots of the system, Jessy uses a novel variation of version vectors called dependence vectors. An analytical comparison between Jessy and previous partial replication protocol shows that Jessy contacts fewer replicas, and that, in addition, it may commit faster.

References

[1] D. Serrano, M. Patino-Martinez, R. Jimenez-Peris, and B. Kemme, "Boosting Database Replication Scalability through Partial Replication and 1-Copy-Snapshot-Isolation," in *Pacific Rim International Symposium on Dependable Computing*, Washington, DC, USA, Dec. 2007, pp. 290–297.

[2] J. E. Armendáriz-Iñigo, A. Mauch-Goya, J. R. G. de Mendívil, and F. D. Muñoz Escoí, "SIPRe: a partial database replication protocol with SI replicas," in *Sym. on Applied computing*, ser. SAC '08, New York, USA, 2008, p. 2181.

[3] K. Daudjee and K. Salem, "Lazy database replication with snapshot isolation," in *International Conference on Very Large Data Bases*, ser. VLDB '06. VLDB Endowment, 2006, pp. 715–726.

[4] A. Bieniusa and T. Fuhrmann, "Consistency in hindsight: A fully decentralized STM algorithm," in *International Symposium on Parallel & Distributed Processing*, 2010, pp. 1–12.

[5] T. Riegel, C. Fetzer, and P. Felber, "Snapshot isolation for software transactional memory," in *1st Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2006.

[6] A. Adya, "Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions," Ph.D., MIT, Cambridge, MA, USA, Mar. 1999.

[7] S. Elnikety, W. Zwaenepoel, and F. Pedone, "Database Replication Using Generalized Snapshot Isolation," in *Symposium on Reliable Distributed Systems*, Washington, DC, USA, Oct. 2005, pp. 73–84.

[8] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A critique of ANSI SQL isolation levels," in *Conference on Management of Data*, New York, NY, USA, 1995, pp. 1–10.

[9] Y. Sovran, R. Power, M. K. Aguilera, and J. Li, "Transactional storage for geo-replicated systems," in *Symposium on Operating Systems Principles*, New York, NY, USA, 2011, pp. 385–400.

[10] N. Schiper, P. Sutra, and F. Pedone, "P-store: Genuine partial replication in wide area networks," in *Symposium on Reliable Distributed Systems*, ser. SRDS '10, Washington, DC, USA, 2010, pp. 214–224.

[11] P. Bernstein, V. Radzilacos, and V. Hadzilacos, *Concurrency Control and Recovery in Database Systems.* Addison Wesley Publishing Company, 1987.

[12] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM*, vol. 32, no. 2, pp. 374–382, 1985.

[13] M. Abadi and L. Lamport, "The existence of refinement mappings," *Theory Computer Science*, vol. 82, pp. 253–284, May 1991.

[14] H. Garcia-Molina and G. Wiederhold, "Read-only transactions in a distributed database," *ACM Trans. Database Syst.*, vol. 7, no. 2, pp. 209–234, Jun. 1982.

[15] C. H. Papadimitriou, "The serializability of concurrent database updates," *Journal of the ACM*, vol. 26, no. 4, pp. 631–653, Oct. 1979.

[16] A. Chan and R. Gray, "Implementing Distributed Read-Only Transactions," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 2, pp. 205–212, Feb. 1985.

[17] H. Attiya, E. Hillel, and A. Milani, "Inherent limitations on disjoint-access parallel implementations of transactional memory," in *symposium on Parallelism in algorithms and architectures*, ser. SPAA '09. New York, NY, USA: ACM, 2009, pp. 69–78.

[18] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM*, vol. 43, no. 2, pp. 225–267, 1996.

[19] M. J. Cahill, U. Röhm, and A. D. Fekete, "Serializable isolation for snapshot databases," in *Conference on Management of Data.* New York, New York, USA: ACM Press, Jun. 2008, p. 729.

[20] M. Saeida Ardekani, M. Zawirski, P. Sutra, and M. Shapiro, "The space complexity of transactional interactive reads," in *International Workshop on Hot Topics in Cloud Data Processing*, Bern, Switzerland, Apr. 2012.

[21] N. Schiper, "On Multicast Primitives in Large Networks and Partial Replication Protocols," Ph.D. dissertation, Faculty of Informatics of the University of Lugano, October 2009.

[22] C. Coulon, E. Pacitti, and P. Valduriez, "Consistency management for partial replication in a high performance database cluster," in *International Conference on Parallel and Distributed Systems*, vol. 1, Los Alamitos, CA, USA, 2005, pp. 809–815.

[23] N. Schiper, R. Schmidt, and F. Pedone, "Brief announcement: Optimistic algorithms for partial database replication," in *Symposium on Distributed Computing*, 2006, pp. 557–559.

[24] R. Jiménez-Peris, M. Patiño-Martínez, B. Kemme, and G. Alonso, "Improving the scalability of fault-tolerant database clusters," in *International Conference on Distributed Computing Systems*, ser. ICDCS '02, Washington, DC, USA, 2002, pp. 477–484.

[25] J. Holliday, D. Agrawal, and A. E. Abbadi, "Partial database replication using epidemic communication," in *International Conference on Distributed Computing Systems*, Washington, DC, USA, 2002, pp. 485–493.

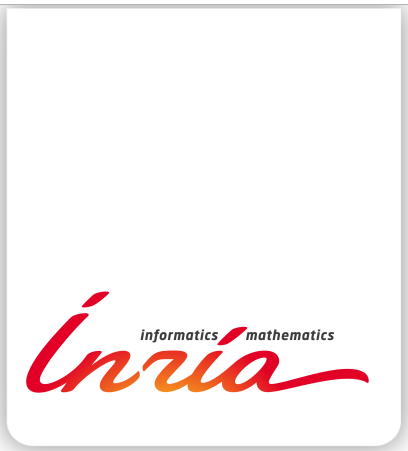
[26] S. Peluso, P. Ruivo, P. Romano, F. Quaglia, and L. Rodrigues, "When scalability meets consistency: Genuine multiversion update-serializable partial data replication," in *International Conference on Distributed Computing Systems*, Los Alamitos, CA, USA, Jun. 2012, pp. 455–465.

[27] J. Gray, P. Helland, P. O'Neil, and D. Shasha, "The dangers of replication and a solution," *ACM SIGMOD Record*, vol. 25, no. 2, pp. 173–182, 1996.

[28] M. Wiesmann and A. Schiper, "Comparison of database replication techniques based on total order broadcast," *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, no. 4, pp. 551–566, 2005.

[29] R. Guerraoui and M. Kapalka, "On the correctness of transactional memory," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, ser. PPoPP '08. New York, NY, USA: ACM, 2008, pp. 175–184.

[30] S. Peluso, P. Romano, and F. Quaglia, "Genuine replication, opacity and wait-free read transactions: can a STM get them all?" in *Workshop on the Theory of Transactional Memory (WTTM)*, Madeira, Portugal, Jul. 2012.



RESEARCH CENTRE
PARIS – ROCQUENCOURT

Domaine de Voluceau, - Rocquencourt

B.P. 105 - 78153 Le Chesnay Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399