Non-Monotonic Snapshot Isolation

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

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

RESEARCH REPORT
N° 7805
Juin 2013
Project-Teams Regal

Non-Monotonic Snapshot Isolation*

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

Project-Teams Regal

Research Report n° 7805 — version 5 — initial version
November 2011 — revised version Juin 2013 — 45 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 6. Section 7 describes Jessy, our NMSI protocol. We compare with related work in Section 8, and conclude in Section 9.

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 use $w_i(x_i)$ 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, 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 and one of them modifies it; they *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).[3]

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] These restrictions ease the exposition of our results but do not change their validity.

[3] Throughout the paper, read-only transactions are specified with an alphabet subscript, and update transactions are shown with numeric subscript.

$$\begin{tikzcd} h_1 = r_a(x_0) \arrow[r] \arrow[dr, dash] & r_1(x_0).w_1(x_1).c_1 & \\ & r_a(y_0).c_a \arrow[r] & r_2(y_0).w_2(y_2).c_2 \end{tikzcd}$$

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.

Let us consider a function $\mathcal{S}$ 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 rules:

D1 (Read Rule)

$$\begin{array}{lll} \multicolumn{3}{l}{\forall r_i(x_{j\neq i}), w_{k\neq j}(x_k), c_k \in h_s :} \\ & c_j \in h_s & (D1.1) \\ \wedge & c_j <_{h_s} s_i & (D1.2) \\ \wedge & (c_k <_{h_s} c_j \vee s_i <_{h_s} c_k) & (D1.3) \end{array}$$

D2 (Write Rule)

$$\begin{array}{l} \forall c_i, c_j \in h_s : \\ \quad ws(T_i) \cap ws(T_j) \neq \varnothing \\ \quad \Rightarrow (c_i <_{h_s} s_j \vee c_j <_{h_s} s_i) \end{array}$$

2.4 System

We consider a message-passing system of n processes $\Pi = \{p_1, \ldots, p_n\}$. Links are quasi-reliable. 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 *acceptable* 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 $\mathfrak{F}(\rho)$.[4]

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

[4] Notice that since steps to implement operations may interleave, $<_h$ is not necessarily a total order.

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. By extension for some set of objects X, $\mathit{replicas}(X)$ denotes the replicas of X; given a transaction T_i, $\mathit{replicas}(T_i)$ equals $\mathit{replicas}(\mathit{rs}(T_i) \cup \mathit{ws}(T_i))$.

We make no assumption about how objects are replicated. The coordinator of T_i, denoted $\mathit{coord}(T_i)$, 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." As a consequence, SI does not preclude a transaction to always observe outdated data. This implies that an update transaction may always abort even if it runs alone. To ensure that a transactional system remains practical, Herlihy et al. [14], as well as Guerraoui and Kapalka [15], consider that an update transaction should abort only if a conflict occurs. In the case of SI, we require that this property holds for write-conflict, i.e., a query never forces an update to abort.

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

Most workloads exhibit a high proportion of read-only transactions, or *queries*. The wait-free queries property (see below) ensures that such accesses are fast. SI was designed at core to offer this property.

- **Wait-free Queries (WFQ).** A read-only transaction T_i never waits for another transaction and eventually commits.

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 derive our impossibility result.

3.1 Cascading Aborts

Intuitively, a read-only transaction must abort if it observes the effects of an 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 1 (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 2 (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.*[5] *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.Ho Notice that, even if T_1 causally precedes T_b, T_b does not depend on T_1 in h_3.

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:

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

Definition 3 (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 at some *point* in the past. This requirement is stronger than consistent snapshot. For some transaction T_i, it implies that *(i)* there exists a snapshot point for T_i (SCONSa), and *(ii)* 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 (SCONSb). 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 and transaction T_a observes the effect of T_2 (i.e., y_2), it should also observe the effect of T_1 (i.e., x_1). SCONSb prevents history h_6 to occur.

Definition 4 (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:*

- $\forall r_i(x_j), r_i(y_l) \in h : r_i(x_j) \not<_h c_l$ $\quad (SCONSa)$
- $\forall r_i(x_j), r_i(y_l), w_k(x_k) \in h :$
  $c_k <_h c_l \Rightarrow c_k <_h c_j$ $\quad (SCONSb)$

We note SCONS the set of strictly consistent histories.

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.

$$\begin{tikzcd}
h_7 = r_a(x_0) \arrow[r] \arrow[drr] & r_1(x_0).w_1(x_1).c_1 \arrow[r] & r_b(x_1).c_b \\
r_b(y_0) \arrow[r] \arrow[urr] & r_2(y_0).w_2(y_2).c_2 \arrow[r] & r_a(y_2).c_a
\end{tikzcd}$$

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. Below, we introduce an ordering relation between snapshots to formalize snapshot monotonicity.

Definition 5 (Snapshot precedence)**.** *Consider a history h and two distinct transactions T_i and T_j. The snapshot read by T_i precedes the snapshot read by T_j in history h, written $T_i \rightarrow T_j$, when $r_i(x_k)$ and $r_j(y_l)$ 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.*

For more illustration, consider $h_8 = r_1(x_0).w_1(x_1).c_1.r_2(y_0).w_2(y_2).r_a(x_1).c_2$ $.r_b(y_2).c_a.c_b$ and $h_9 = r_1(x_0).w_1(x_1).c_1.r_a(x_1).c_a.r_2(x_1).r_2(y_0).w_2(x_2).w_2(y_2).c_2$ $.r_b(y_2).c_b$. In history h_8, $T_a \rightarrow T_b$ holds because $r_a(x_1)$ precedes c_2 and T_b reads y_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 6 (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 are observed under update serializability [17], that is when queries observe consistent state, but only updates are serializable.

3.4 Write-Conflict Freedom

Rule D2 of SI forbids two concurrent write-conflicting transactions from both committing. Since in our model we assume that every write is preceeded by a corresponding read on the same object, 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 7 (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

Theorem 1 below establishes that a history h is in SI iff (1) every transaction in h sees a committed state, (2) every transaction in h observes a strictly consistent snapshot, (3) snapshots are monotonic, and (4) h is write-conflict free.

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 $h \in \mathrm{SI}$ 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 $T_i \to T_j$ 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$\cap$SCONS$\cap$WCF$\cap$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 $\cap$ CONS $\cap$ 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:

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

  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, $T_i \rightarrow T_{i+1}$ 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, $T_i \rightarrow T_{i+1}$ 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, $T_i \rightarrow T_{i+1}$ holds.

Applying the result above to Equation 1, we obtain: $T_1 \rightarrow T_2 \ldots \rightarrow T_n \rightarrow T_1$. 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 {SCONS, MON, WCF, 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 $r_1(x_0).r_2(x_0).w_1(x_1).w_2(x_2).c_1.c_2$.

□

To the best of our knowledge, this result is the first to prove that SI can be split into simpler properties. Theorem 1 also establishes that SI is definable on plain histories. This has two interesting consequences: (i) a transactional system does not have to explicitly implement snapshots to support SI, and (ii) one can compare SI to other consistency criterion without relying on a phenomena based characterization (contrary to, e.g., the work of Adya [6]).

4 The impossibility of SI with GPR

This section leverages our previous decomposition result to show that SI is inherently non-scalable. In more details, we show that none of MON, SCONSa or SCONSb is attainable in some asynchronous failure-free GPR system Π when updates are obstruction-free and queries are wait-free. To prove these results, we first characterize in Lemmata 4 and 5 histories acceptable by Π.

Lemma 4 (Positive-freshness Acceptance)**.** *Consider an acceptable history h and a transaction T_i pending in h such that the next operation invoked by T_i is a read on some object x. Note x_j the latest committed version of x prior to the first operation of T_i in h. Let ρ be an execution satisfying $\mathfrak{F}(\rho) = h$. If $h.r_i(x_j)$ belongs to SI and there is no concurrent write-conflicting transaction with T_i, then there exists an execution ρ' extending ρ such that in history $\mathfrak{F}(\rho')$, transaction T_i reads at least (in the sense of $\ll_h$) version x_j of x.*

Proof. By contradiction. Assume that in every execution extending ρ, transaction T_i reads a version $x_k \ll_h x_j$. Let ρ' be such an extension in which (i) no other transaction than T_i makes steps, (ii) we extend T_i after its read upon x by a write on x, then (iii) *coord*(T_i) tries committing T_i. Since T_i reads version x_k in $\mathfrak{F}(\rho')$, transaction T_i should abort. However in history $\mathfrak{F}(\rho')$ there is no concurrent write-conflicting transaction with T_i. Hence, this execution contradicts that updates are obstruction-free. □

Lemma 5 (Genuine Acceptance)**.** *Let $h = \mathfrak{F}(\rho)$ be an acceptable history by* Π *such that a transaction T_i is pending in h. Note X the set of objects accessed by T_i in h. Only processes in replicas(X) make steps to execute T_i in ρ.*

Proof. (By contradiction.) Consider 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 any additional operation 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. □

We now state that monotonic snapshots are not constructable by Π. Our proof holds because objects accessed by a transaction are not known in advance.

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

Proof. (By contradiction.) Let us consider (i) four objects x, y, z and u such that for any two objects in $\{x, y, z, u\}$, their replica sets do not intersect; (ii) four queries T_a, T_b, T_c and T_d accessing respectively $\{x, y\}$, $\{y, z\}$, $\{z, u\}$ and $\{u, x\}$; and (iii) four updates T_1, T_2, T_3 and T_4 modifying respectively x, y, z and u.

Obviously, history $r_b(y_0)$ is acceptable, and since updates are obstruction-free, $r_b(y_0).r_2(y_0).w_2(y_2).c_2$ is also acceptable. Applying that Lemma 4, we obtain that history $r_b(y_0).r_2(y_0).w_2(y_2).c_2.r_a(x_0).r_a(y_2)$ is acceptable. Since T_a is wait-free, $h = r_b(y_0).r_2(y_0).w_2(y_2).c_2.r_a(x_0).r_a(y_2).c_a$ is acceptable as well. Using a similar reasoning, $h' = r_d(u_0).r_4(u_0).w_4(u_4).c_4.r_c(z_0).r_c(u_4).c_c$ is also acceptable. We note ρ and ρ' respectively two sequences of steps such that $\mathfrak{F}(\rho) = h$ and $\mathfrak{F}(\rho') = h'$.

The system Π is GPR. As a consequence, Lemma 5 tells us that only processes in *replicas*(x, y) make steps in ρ. Similarly, only processes in *replicas*(u, z) make steps in ρ'. By hypothesis, *replicas*(x, y) and *replicas*(u, z) are disjoint. Applying a classical indistinguishably argument [12, Lemma 1], both $\rho'.\rho$ and $\rho.\rho'$ are admissible by Π. Thus, histories $h'.h = \mathfrak{F}(\rho'.\rho)$ and $h.h' = \mathfrak{F}(\rho.\rho')$ are acceptable.

Since updates are obstruction-free, history $h'.h.r_3(z_0).w_3(z_3).c_3$ is acceptable. Note U the sequence of steps following $\rho'.\rho$ with $\mathfrak{F}(U) = r_3(z_0).w_3(z_3).c_3$. Observe that by Lemma 5 $\rho'.\rho.U$ is indistinguishable from $\rho'.U.\rho$. Then consider history $\mathfrak{F}(\rho'.U.\rho)$. In this history, T_b is pending and the latest version of object z is z_3, As a consequence, by applying Lemma 4, there exists an extension of $\rho'.U.\rho$ in which transaction T_b reads z_3. From the fact that queries are wait-free and since $\rho'.\rho.U$ is indistinguishable from $\rho'.U.\rho$, we obtain that history $h_1 = h'.h.r_3(z_0).w_3(z_3).c_3.r_b(z_3).c_b$ is acceptable. We note U_1 the sequence of steps following $\rho'.\rho$ such that $\mathfrak{F}(U_1)$ equals $r_3(z_0).w_3(z_3).c_3.r_b(z_3).c_b$.

With a similar reasoning, history $h_2 = h'.h.r_1(x_0).w_1(x_1).c_1.r_d(x_1).c_d$ is acceptable. Note U_2 the sequence satisfying $\mathfrak{F}(U_2) = r_1(x_0).w_1(x_1).c_1.r_d(x_1).c_d$.

Executions $\rho'.\rho.U_1$ and $\rho'.\rho.U_2$ are both admissible. Because Π is GPR, only processes in *replicas*(y, z) (resp. *replicas*(x, u)) make steps in U_1 (resp. U_2). By hypothesis, these two replica sets are disjoint. Applying again an indistinguishably argument, $\rho'.\rho.U_1.U_2$ is an execution of Π. Therefore, the history $\hat{h} = \mathfrak{F}(\rho'.\rho.U_1.U_2)$ is acceptable. In this history, relation $T_a \rightarrow T_b \rightarrow T_c \rightarrow T_d \rightarrow T_a$ holds. Thus, $\hat{h}$ does not belong to MON. Contradiction. □

Our next theorem states that SCONSb is not attainable. Similarly to Attiya et al. [18], 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 8 (Flippable execution)**.** *Consider two distinct objects x and y, a query 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 $\rho = U_1 V_2 U_2 \ldots V_m U_m$ where,*

- *transaction T_a reads in history $h = \mathfrak{F}(\rho)$ at least version x_1 of x,*
- *for any j in $[\![1,m]\!]$, U_j is the execution of transaction T_j by processes Q_j,*
- *for any j in $[\![2,m]\!]$, V_j are steps to execute T_a by processes P_j, and*
- *both $(Q_j \cap P_j = \varnothing) \oplus (P_j \cap Q_{j+1} = \varnothing)$ and $Q_j \cap Q_{j+1} = \varnothing$ hold,*

is called flippable.

Lemma 6. *Let ρ be an execution admissible by Π. If ρ is flippable and histories accepted by Π satisfy 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 $*$ corresponds to either no operation, or to some read operation by T_a on 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 $j < l$ 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 sequence of steps V_k made by P_k right after U_l to execute T_a. Applying the definition of a flippable execution, we know that (F1) $(Q_l \cap P_k = \varnothing) \oplus (P_k \cap Q_k = \varnothing)$, and (F2) $Q_l \cap Q_k = \varnothing$. Consider now the following cases:

(Case $Q_l \cap P_k = \varnothing$.) It follows that ρ is indistinguishable from the execution $\rho'' = \ldots U_j \ldots V_k U_l U_k \ldots$. Then from fact F2, ρ is indistinguishable from execution $\rho' = \ldots U_j \ldots V_k U_k U_l \ldots$.

(Case $P_k \cap Q_k = \varnothing$) With a similar reasoning, we obtain that ρ is indistinguishable from $\rho' = \ldots U_j \ldots U_k U_l V_k \ldots$.

(Case $P_k \cap (Q_l \cup Q_k) = \varnothing$.) This case reduces to any of the two above cases.

Note h' the history $\mathfrak{F}(\rho')$. Observe that since ρ' is indistinguishable from ρ, history h' is acceptable. In history h', $c_k <_{h'} c_l$ holds. Moreover, $c_j <_{h'} c_k$

holds by the assumption $j < l$ and the fact that k equals $l+1$. Besides, operations $r_i(x_j)$, $r_i(y_l)$ and $w_k(x_k)$ all belong to h'. According to the definition of SCONSb, transaction T_a does not commit in h'. (The case $j > l$ follows a symmetrical reasoning to the case $l > j$ we considered previously.) □

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

Proof. (By contradiction.) Consider two objects x and y such that $replicas(x)$ and $replicas(y)$ are disjoint. Assume a read-only transaction T_a that reads successively x then y. Below, we exhibit an execution admissible by Π during which transaction T_a never terminates. We build this execution as follows:

[Construction.] Consider some empty execution ρ. Repeat for all $i >= 1$: Let T_i be an update of x, if i is odd, and y otherwise. Start the execution of transaction T_i. Since no concurrent transaction is write-conflicting with T_i in ρ and updates are obstruction-free, there must exist an extension $\rho.U_i$ of ρ during which T_i commits. Assign to ρ the value of $\rho.U_i$. Execution ρ is flippable. Hence, Lemma 6 tells us that transaction T_a does not terminate in this execution. Consider the two following cases: (Case $i = 1$) Because Π satisfies non-trivial SI, there exists an extension ρ' of ρ in which transaction T_a reads at least version x_1 of object x. Notice that execution ρ' is of the form $U_1.V_2.s.\ldots$ where *(i)* all steps in V_2 are made by processes in $replicas(x)$, and *(ii)* s is the first step such that $\mathfrak{F}(U_1.V_2.s.) = r_1(x_0).w_1(x_1).c_1.r_a(x_1)$. Assign $U_1.V_2$ to ρ . (Case $i > 2$) Consider any step V_{i+1} to terminate T_a and append it to ρ.

Execution ρ is admissible by Π. Hence $\mathfrak{F}(\rho)$ is acceptable. However, in this history transaction T_a does not terminate. This contradicts the fact that queries are wait-free. □

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. (By contradiction.) Consider two distinct objects x and y such that $replicas(x)$ and $replicas(y)$ are disjoint. Let T_1 be an update accessing y, and T_a be a query reading both objects.

Obviously, history $h = r_a(x_0)$ is acceptable. Note U_a a sequence of steps satisfying $U_a = \mathfrak{F}(r_a(x_0))$. Because Π supports obstruction-free updates, we know the existence of an extension $U_a.U_1$ of U_a such that $\mathfrak{F}(U_1) = r_1(y_0).w_1(y_1).c_1$. By Lemma 5, we observe that $U_a.U_1$ is indistinguishable from $U_1.U_a$. Then by Lemma 4, there must exist an extension $U_1.U_a.V_a$ of $U_1.U_a$ admissible by Π and such that $\mathfrak{F}(V_a) = r_a(y_1).c_a$. Finally, since $U_a.U_1$

is indistinguishable from $U_1.U_a$ and $U_1.U_a.V_a$ is admissible, $U_a.U_1.V_a$ is admissible too. The history $\mathfrak{F}(U_a.U_1.V_a)$ is not in SCONSa. Contradiction. □

As a consequence of the above, no asynchronous system, even if it is failure-free, can support both GPR and SI. In particular, even if the system is augmented with failure detectors [19], a common approach to model partial synchrony, SI cannot be implemented under GPR. This fact strongly hinders the usage of SI at large scale. In the following sections, we further discuss implications of this impossibility result then we introduce a novel consistency criterion to overcome it.

5 Discussion

In this section, we discuss the consequences of our impossibility results, with an emphasis on other consistency criteria than SI.

5.1 Declaring the Read-set in Advance

When a transaction declares objects it accesses *in advance*, a GPR system can install a strictly consistent and monotonic snapshot just after the start of the transaction. As a consequence, such an assumption sidesteps our impossibility result. This is the approach employed in the SI protocol of Armendáriz-Iñigo et al. [2]. Still, this protocol makes use of atomic broadcast to install a snapshot. We obtain a GPR system that supports SI by replacing this group communication primitive by a genuine atomic multicast.

5.2 Strict Serializability and Opacity

We observe that Theorem 4 also holds if we consider the following (classical) definition of obstruction-free updates in which both read-write and write-write conflicts are taken into account:

- **Obstruction-free Updates (OFU-a).** For every update transaction T_i, if $coord(T_i)$ is correct then T_i eventually terminates. Moreover, if T_i does not *conflict* with some concurrent transaction then T_i eventually commits.

As a consequence, neither strict serializability [20], nor opacity [21] is attainable under GPR. In the case of opacity, this answers negatively to a problem recently posed by Peluso et al. [22].

5.3 Serializability (SER)

5.3.1 Permissiveness

A transactional system Π is *permissive* with respect to a consistency criterion $\mathcal{C}$ when every history $h \in \mathcal{C}$ is acceptable by Π. Permissiveness [23] measures the optimal amount of concurrency a system allows. If we consider again histories h_1 and h_2 in the proof of Theorem 2, we observe that both histories are serializable. Hence, every system permissive with respect to SER accepts both histories. By relying on the very same argument as the one we exhibit to close the proof of Theorem 2, we conclude that no transactional system is both GPR and permissive with respect to SER. For instance, P-Store [10], a GPR protocol that ensures SER, does not accept history $h_{10} = r_1(x_0).w_1(x_1).c_1.r_2(x_0).r_2(y_0).w_2(y_2).c_2$.

5.3.2 Wait-free Queries.

Under SI, a query never forces an update to abort. This key feature of SI greatly improves performance. Most recent transactional systems that support SER (e.g., [10, 24–34]) offer such a progress property as well as positive-freshness acceptance:[6]

- **Obstruction-free Updates (OFU-b).** For every update transaction T_i, if $coord(T_i)$ is correct then T_i eventually terminates. Moreover, if T_i does not conflict with some concurrent *update* transaction then T_i eventually commits.
- **Positive Freshness Acceptance.** Consider an acceptable history h and a transaction T_i pending in h such that the next operation invoked by T_i is a read on some object x. Note x_j the latest committed version of x prior to the first operation of T_i in h. Let ρ be an execution satisfying $\mathfrak{F}(\rho) = h$. If $h.r_i(x_j)$ belongs to SER and there is no concurrent write-conflicting update transaction with T_i, then there exists an execution ρ' extending ρ such that in history $\mathfrak{F}(\rho')$, transaction T_i reads at least (in the sense of $\ll_h$) version x_j of x.

When the two above progress properties holds, Theorem 2 applies to SER transactional systems, implying a choice between WFQ and GPR. The P-Store transactional system of Schiper et al. [10] favors GPR over WFQ. On the contrary, the protocol of Sciascia et al. [33] ensures WFQ but is not GPR. Recently, Peluso et al. [34] have proposed a GPR algorithm that supports both SER and WFQ in the failure-free case. This protocol sidesteps the impossibility result by dropping obstruction-freedom for updates in certain scenarios.[7]

5.4 Parallel Snapshot Isolation (PSI)

Recently, Sovran et al. [9] have introduced a weaker consistency criterion than SI named parallel snapshot isolation (PSI). PSI allow snapshots to

[6] Lemma 4 proves positive-freshness acceptance for SI under standard assumptions (OFU and WFQ). In the case of SER, this property is a feature of the input acceptance of the protocol.

[7] In more details, this algorithm numbers every version with a scalar. If a transaction T_i first reads an object x then updates an object y, in case the version of x is smaller than the latest version of y, say y_k, T_i will not be able to read y_k , and it will thus abort.

be non-monotonic, but still require them to ensure SCONSa. Sovran et al. justify the use of PSI in Walter by the fact that SI is too expensive in a geographically distributed environment [9, page 4]. Our impossibility result establishes that, in order to scale, a transactional system needs supporting both non-monotonic *and* non-strictly consistent snapshots. Thus, while being more scalable than SI, PSI yet cannot be implemented in a GPR system.

6 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 Serializablity [20]	Serializablity [8]	Update Serializablity [17]	Snapshot Isolation	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. [35] 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 [17], we believe that this is a small price to pay for improved performance.

7 Protocol

We now describe Jessy, a scalable transactional system that implements NMSI with GPR. Because distributed locking policies do not scale [36, 37], Jessy employs deferred update replication: transactions are executed optimistically, then certified by a termination protocol. Jessy uses a novel clock mechanism to ensure that snapshots are both fresh and consistent, while preserving wait-freedom of queries and genuineness. We describe it in the next section.

7.1 Building Consistent Snapshots

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 V that maps every read (or write) operation $o(x)$ in a history h to a vector $V(o(x)) \in \mathbb{N}^{|Objects|}$ such that:*

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

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 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 $\langle 1, 2 \rangle$.

$$\begin{tikzcd}[column sep=small, row sep=small]
h_{10} = r_1(x_0).w_1(x_1).c_1 \arrow[dr] & \\
& r_3(x_1).r_3(y_2).w_3(y_3).c_3 \\
r_2(y_0).w_2(y_2).c_2 \arrow[ur] &
\end{tikzcd}$$

Definition (Compatibility Relation)**.** *Consider a transaction T_i and two versions x_j and y_l read by T_i. We shall say that x_j and y_l are* compatible *for T_i, written $compat(T_i, x_j, y_l)$, when both $V(r_i(x_j))[x] \geq V(r_i(y_l))[x]$ and $V(r_i(y_l))[y] \geq V(r_i(x_j))[y]$ hold.*

Using the compatibility relation, we can prove that dependence vectors fully characterize consistent snapshots. First of all, we show in Lemma 7 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 7. *Consider a history h in NMSI, and two transactions T_i and T_j in h. Then,*

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

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 V we have: $V(w_i(x_i)) \geq V(r_i(z_j)) + 1_x$. Besides, always according to the definition of V, it is true that the following equalities hold: $V(r_i(z_j)) = V(w_j(z_j)) = V(w_j(y_j))$. Thus, we have: $V(w_i(x_i)) > V(w_j(y_j))$. The general case $T_i \rhd^* T_j$ is obtained by applying inductively the previous reasoning.
- ($\Leftarrow$) From the definition of function V, 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 V, we have that $V(r_{j'}(y_{j'})) = V(w_{j'}(y_{j'}))$. Since $V(w_i(x_i)) > V(w_j(y_j))$ holds, $V(w_{j'}(y_{j'}))[y] \geq V(w_j(x_j))[y]$ is true. Both transactions T_j and $T_{j'}$ write y. Since h belongs to NMSI, 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 $V(w_j(y_j)) > V(w_{j'}(y_{j'}))$ 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 NMSI and a transaction T_i in h. Transaction T_i sees a consistent snapshot in h* if, an only if, *every pair of versions x_j and y_l read by T_i is compatible.*

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 $V(r_i(x_l))[x] < V(r_i(y_j))[x]$ holds. By definition of function V, we have $V(r_i(x_l)) = V(w_l(x_l))$ and $V(r_i(y_j)) =$

$V(w_j(y_j))$. Hence, $V(w_l(x_l))[x] < V(w_j(y_j))[x]$ holds. Again from the definition of function V, there exists a transaction $T_{k\neq 0}$ writing on x such that (i) $V(w_j(y_j)) \geq V(w_k(x_k))$ and (ii) $V(w_j(y_j))[x] = V(w_k(x_k))[x]$. Applying Lemma 7 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 NMSI, $T_l \rhd^* T_k$ or $T_k \rhd^* T_l$ holds. From (ii) and $V(w_l(x_l))[x] < V(w_j(y_j))[x]$, we deduce that $V(w_l(x_l))[x] < V(w_k(x_k))[x]$. As a consequence of Lemma 7, $T_k \rhd^* T_l$ 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 NMSI, one can easily show that $T_k \rhd^* T_j$. Since $T_k \rhd^* T_j$, Lemma 7 tells us that $V(w_k(x_k)) > V(w_j(x_j))$ holds. Since $T_i \rhd^* T_k$ holds, a short induction on the definition of function V tells us that $V(r_i(x_j))[x] \geq V(w_k(x_k))|x]$ is true. From which we obtain that: $V(r_i(x_j))[x] \geq V(w_k(x_k))[x] > V(w_j(x_j))[x] = V(r_i(x_j))[x]$. Contradiction.

□

Despite that in the common case dependence vectors are sparse, they might be large for certain workloads. For instance, if transactions execute random accesses, the size of each vector tends asymptotically to the number of objects in the system. To address the above problem, Jessy employs a mechanism to approximate dependencies safely, by coarsening the granularity, grouping objects into disjoint partitions and serializing updates in a group as if it was a single larger object. We cover this mechanism in what follows.

7.2 Partitioned Dependence Vector

Consider some partition $\mathcal{P}$ of *Objects*. For some object x, note $P(x)$ the partition x belongs to, and by extension, for some $S \subseteq$ *Objects*, note $P(S)$ the set $\{\mathcal{P}(x) \mid x \in S\}$. A partition is *proper* when updates inside the same partition are serialized, that is, for every $X \in \mathcal{P}$ and every two writes $w_i(x_i)$, $w_j(y_j)$ with $\mathcal{P}(x) = \mathcal{P}(y)$, either $w_i(x_i) <_h w_j(y_j)$ or $w_j(y_j) <_h w_i(x_i)$ holds.

Now, consider some history h, and for every object x replace every operation $o_i(x)$ in h by $o_i(\mathcal{P}(x))$. We obtain a history that we note $h^{\mathcal{P}}$. The following result linked the consistency of h to the consistency of $h^{\mathcal{P}}$:

Proposition 4. *Consider some history h. If $\mathcal{P}$ is a proper partition of Objects for h and history $h^{\mathcal{P}}$ belongs to CONS, then h is in CONS.*

Proof. First of all we observe that for any two transactions T_i and T_j:

- If $T_i \rhd^* T_j$ holds in h then $T_i \rhd^* T_j$ holds in $h^{\mathcal{P}}$.
  Proof. If $T_i \rhd T_j$ holds in h, then $r_i(x_j)$ is in h. Thus $r_i(\mathcal{P}(x_j))$ is in $h^{\mathcal{P}}$. It follows that $T_i \rhd T_j$ holds in $h^{\mathcal{P}}$. If $T_i \rhd^* T_j$ in h then there exist a set of transactions $\{T_1, \ldots, T_m\}$ such that: $T_i \rhd T_1 \ldots \rhd T_m \rhd T_j$ hold in h. From the result above, we deduce that $T_i \rhd T_1 \ldots \rhd T_m \rhd T_j$ hold in $h^{\mathcal{P}}$. Hence, $T_i \rhd^* T_j$ holds in $h^{\mathcal{P}}$. □

- If $x_i \ll x_j$ holds in h then $\mathcal{P}(x_i) \ll \mathcal{P}(x_j)$ holds in $h^{\mathcal{P}(x)}$.
  Proof. If $x_i \ll x_j$ holds in h then $\mathcal{P}(x_i) \ll \mathcal{P}(x_j)$ holds in h. □

For the sake of contradiction, assume that $h^{\mathcal{P}}$ is in CONS while h is not in CONS. It follows that there exist a transaction T_i, some object x and a transaction T_k on which T_i depends such that in h, T_i reads version x_j, T_k writes version x_k, and $x_j \ll_h x_k$. From the two observations above, we obtain that $T_i \rhd T_j$, $T_i \rhd^* T_k$ and $\mathcal{P}(x_j) \ll_h \mathcal{P}(x_k)$ hold in $h^{\mathcal{P}}$. Hence, $h^{\mathcal{P}}$ is not consistent. Contradiction.

□

Given two operations $o_i(x_j)$ and $o_k(y_l)$, let us introduce relation $o_i(x_j) \leq_h^{\mathcal{P}} o_k(y_l)$ when $o_i(x_j) = o_k(y_l)$, or $o_i(x_j) <_h o_k(y_l) \wedge \mathcal{P}(x) = \mathcal{P}(y)$ is true. Based on Proposition 4, we define below a function that approximates dependencies safely:

Definition 9 (Partitioned Dependence Vector)**.** *A partitioned dependence vector is a function PV that maps every read (or write) operation $o(x)$ in a history h to a vector $PV(o(x)) \in \mathbb{N}^{|\mathcal{P}|}$ such that:*

$$
\begin{aligned}
&PV(r_i(x_0)) = 0^{|\mathcal{P}|}\\
&PV(r_i(x_j)) = max\ \{PV(w_l(y_l)) \mid w_l(y_l) \leq_h^{\mathcal{P}} r_i(x_j)\\
&\qquad\qquad\qquad\qquad \wedge \big(\forall k : x_j \ll_h x_k \Rightarrow w_l(y_l) \leq_h^{\mathcal{P}} w_k(x_k)\big)\}\\
&PV(w_i(x_i)) = max\ \{PV(r_i(y_j)) \mid y_j \in rs(T_i)\} \cup\\
&\qquad\qquad\qquad \{PV(w_k(z_k)) : w_k(z_k) \leq_h^{\mathcal{P}} w_i(x_i)\}\\
&\qquad\qquad + \ \Sigma_{X \in \mathcal{P}(ws(T_i))}\ 1_X
\end{aligned}
$$

The first two rules of function PV are identical to the ones that would give us function V on history $h^{\mathcal{P}}$. The second part of the third rule serializes objects in the same partition

We now prove that partitioned dependence vectors properly capture consistent snapshots. Consider the following definition of $compat(T_i, x_j, y_l)$ for a proper partition $\mathcal{P}$:

Case $\mathcal{P}(x) \neq \mathcal{P}(y)$. This case is identical to the definition we gave for function V. In other words, both $PV(r_i(x_j))[\mathcal{P}(x)] \geq PV(r_i(y_l))[\mathcal{P}(x)]$ and $PV(r_i(y_l))[\mathcal{P}(y)] \geq PV(r_i(x_j))[\mathcal{P}(y)]$ must hold.

Case $\mathcal{P}(x) = \mathcal{P}(y)$. This case deals with the fact that inside a partition writes are serialized. We have (i) if $PV(r_i(x_j))[\mathcal{P}(y)] > PV(r_i(y_l))[\mathcal{P}(y)]$ holds then $y_l = max\ \{y_k \mid w_k(y_k) \leq_h^{\mathcal{P}} w_j(x_j)\}$, or symmetrically (ii) if $PV(r_i(y_l))[\mathcal{P}(x)] > PV(r_i(x_j))[\mathcal{P}(x)]$ holds then $x_j = max\ \{x_k \mid w_k(x_k) \leq_h^{\mathcal{P}} w_l(y_l)\}$, or otherwise (iii) the predicate equals *true*.

We prove next that the "if" part of Theorem 5 holds for the above definition of compatibility:

Proposition 5. *Consider a history h in NMSI and a transaction T_i in h. If every pair of versions x_j and y_l read by T_i is compatible, then transaction T_i sees a consistent snapshot in h*

Proof. Using a reasoning identical to the one we depicted in the proof of Theorem 5, we can prove that $h^{\mathcal{P}}$ belongs to CONS. Then, from Proposition 4, we know that if $h^{\mathcal{P}}$ belongs to CONS, then h belong to CONS. □

As discussed in [38], we notice here the existence of a trade-off between the size of the vectors and the freshness of the snapshots. For instance, if x and y belong to the same partition and transaction T_i reads a version x_j, T_i cannot read a version y_l that committed after a version x_k posterior to x_j.

7.3 Transaction Lifetime in Jessy

Jessy is a distributed system of processes which communicate by message passing. When a client (not modeled) executes a transaction T_i with Jessy, T_i is handled by a coordinator. The coordinator of a transaction can be any process in the system. A transaction T_i can be in one of the following four states at some process:

- *Executing*: Each non-termination operation $o_i(x)$ in T_i is executed optimistically (i.e., without synchronization 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 replica or a remote one. 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.

Algorithm 1 Execution Protocol of Jessy

```
Variables:
    $db$, $submitted$, $committed$, $aborted$

$remoteRead(x, T_i)$
    pre:  $received\ \langle \text{REQUEST}, T_i, x\rangle\ from\ q$
          $\exists(x, v, j) \in db : \forall y_l \in rs(T_i) : compat(T_i, x_j, y_l)$
    eff:  $send\ \langle \text{REPLY}, T_i, x, v\rangle\ to\ q$

$execute(\text{WRITE}, x, v, T_i)$
    eff:  $up(T_i) \leftarrow up(T_i) \cup \{(x, v, i)\}$

$execute(\text{READ}, x, T_i)$
    eff:  if $\exists(x, v, i) \in up(T_i)$ then return $v$
          else
             $send\ \langle \text{REQUEST}, T_i, x\rangle\ to\ replicas(x)$
             wait until $received\ \langle \text{REPLY}, T_i, x, v\rangle$
             return $v$

$execute(\text{TERM}, T_i)$
    eff:  $submitted \leftarrow submitted \cup \{T_i\}$
          wait until $T_i \in decided$
          if $T_i \in committed$ then return COMMIT
          return ABORT

```

- *Submitted*: Once all the read and write operations of T_i have executed, T_i 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.

7.4 Execution Protocol

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

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

Upon a read request for x, $\mathit{coord}(T_i)$ checks against $\mathit{up}(T_i)$ if x has been previously updated by the same transaction; if so, it returns the corresponding value (line 13). Otherwise, $\mathit{coord}(T_i)$ sends an (asynchronous) read request to the processes that replicate x (lines 15 to 17). When a process receives a read request for object x that it replicates, it returns a version of x which complies with Theorem 5 (lines 5 to 7).

Upon a write request of T_i, the process buffers the update value in $\mathit{up}(T_i)$ (line 10). During commitment, the updates of T_i will be sent to all replicas holding an object that is modified by T_i .

When transaction T_i terminates, it is submitted to the termination protocol (line 20). The execution protocol then waits until T_i either commits or aborts, and returns the outcome.

7.5 Termination Protocol

Algorithm 2 depicts the termination protocol of Jessy. It accesses the same four variables *db*, *submitted* and *committed*, along with a FIFO queue named $\mathcal{Q}$.

In order to satisfy GPR, the termination protocol uses a genuine atomic multicast primitive [39]. In our model, this requires that either (i) we form non-intersecting groups of replicas, and an eventual leader oracle is available in each group, or (ii) that a system-wide *reliable* failure detector is available. The latter setting allows Jessy to tolerate a disaster [40].

To terminate an update transaction T_i, $\mathit{coord}(T_i)$ atomic-multicasts it to every process that holds an object written by T_i. Every such process p certifies T_i by calling function $\mathit{certify}(T_i)$ (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:

$$\mathit{certify}(T_i) \overset{\triangle}{=} \forall T_j \in \mathit{committed} : \mathit{ws}(T_i) \cap \mathit{ws}(T_j) \neq \varnothing \Rightarrow T_i \rhd^* T_j$$

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 $\mathit{replicas}(\mathit{ws}(T_i)) \cup \{\mathit{coord}(T_i)\}$ (lines 17 to 18).

A process can safely decide on the outcome of T_i when it has received votes from a *voting quorum* for T_i. A voting quorum Q for T_i is a set of replicas such that for every object $x \in \mathit{cert}(T_i)$, 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 $\mathit{vquorum}(T_i)$, defined as follows:

$$\mathit{vquorum}(T_i) \triangleq \{Q \subseteq \Pi \mid \forall x \in \mathit{cert}(T_i) : \exists j \in Q \cap \mathit{replicas}(x)\}$$

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

```
$\mathit{outcome}(T_i) \triangleq$
    if $\mathit{cert}(T_i) = \varnothing$
      then $\mathit{true}$
    else if $\forall Q \in \mathit{vquorum}(T_i), \exists q \in Q,$
          $\neg\mathit{received}\ \langle \text{VOTE}, T, - \rangle\ \mathit{from}\ q$
        then $\bot$
    else if $\exists Q \in \mathit{vquorum}(T_i), \forall q \in Q,$
          $\mathit{received}\ \langle \text{VOTE}, T, \mathit{true} \rangle\ \mathit{from}\ q$
        then $\mathit{true}$
    else $\mathit{false}$
```

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 to 24). If instead T_i aborts, p adds T_i to *aborted* (lines 27 to 28).

7.6 Correctness of Jessy

We now sketch a correctness proof of Jessy: Proposition 7 establishes that Jessy generates histories in NMSI. Proposition 8 shows that read-only transactions are wait-free. Propositions 9 and 10, respectively, prove that Jessy satisfies obstruction-freedom for updates and non-triviality for NMSI.

7.6.1 Safety

Proposition 6. *If a transaction T_i commits (respectively aborts) at some process in $\mathit{replicas}(\mathit{ws}(T_i)) \cup \mathit{coord}(T_i)$, it commits (resp. aborts) at every correct process in $\mathit{replicas}(\mathit{ws}(T_i)) \cup \mathit{coord}(T_i)$.*

Proof. This proposition follows from the properties of atomic multicast, the fact that the queue $\mathcal{Q}$ is FIFO, the preconditions at lines 14 to 15 in Algorithm 2, and the definitions of *vote*() and *outcome*(). □

Proposition 7. *Every history admissible by Jessy belongs to NMSI.*

Algorithm 2 Termination Protocol of Jessy

```
Variables:
    $db$, $submitted$, $committed$, $aborted$, $\mathcal{Q}$

$submit(T_i)$
    pre: $T_i \in submitted$
         $ws(T_i) \neq \varnothing$
    eff: AM-Cast($T_i$) to $replicas(ws(T_i))$

$deliver(T_i)$
    pre: $T_i =$ AM-Deliver()
    eff: $\mathcal{Q} \leftarrow \mathcal{Q} \circ \langle T_i \rangle$

$vote(T_i)$
    pre: $T_i \in \mathcal{Q} \setminus decided$
         $\forall T_j \in \mathcal{Q},\ T_j <_{\mathcal{Q}} T_i \Rightarrow T_j \in decided$
    eff: $v \leftarrow certify(T_i)$
         $send\ \langle \text{VOTE}, T_i, v \rangle\ to\ replicas(ws(T_i))$
                                        $\cup \{coord(T_i)\}$

$commit(T_i)$
    pre: $outcome(T_i)$
    eff: foreach $(x, v, i)$ in $up(T_i)$ do
             if $x \in db$ then $db \leftarrow db \cup \{(x, v, i)\}$
         $committed \leftarrow committed \cup \{T_i\}$

$abort(T_i)$
    pre: $\neg outcome(T_i)$
    eff: $aborted \leftarrow aborted \cup \{T_i\}$

```

Proof. We first observe that transactions in Jessy always read committed versions of the objects (line 6 in Algorithm 1). Moreover, we know by Theorem 5 that reads are consistent when Jessy uses dependence vectors, and that this property also holds in case Jessy employs partitioned dependence vectors (Proposition 5). It thus remains to show that histories generated by Jessy are write-conflict free (WCF).

To prove that WCF holds, 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 that the following claim holds:

(C1) 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) decides 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 $\mathcal{Q}$ at process q_i (resp. q_j) before this process sends its VOTE message. It follows from claim C1 that either (1) at the time q_i sends its VOTE message, $T_j <_{\mathcal{Q}} T_i$ holds, or (2) at the time q_j sends its VOTE message, $T_i <_{\mathcal{Q}} T_j$ 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 Proposition 6, we deduce that T_j is committed at process q_i. Hence, *certify*(T_i) returns *false* at process q_i; a contradiction. □

7.6.2 Progress

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

Proof. Transaction T_i executes all its write operations locally at its coordinator. Now, upon executing a read request on some object x, if x was modified previously by T_i, the corresponding value is returned. Otherwise, *coord*(T_i) sends a read request to *replicas*(x). To prove this lemma, we have to show that eventually one of the replica replies to the coordinator.

According to our model, there exists one correct process replica of x. In what follows, we name it p. Observe that since links are quasi-reliable, p eventually receives the read request from *coord*(T_i). Upon receiving this request, process p tries returning a version of x compatible with all versions previously read by T_i.

Consider that Jessy uses dependence vectors (the reasoning for partitioned dependence vectors is similar), and assume, by contradiction, that p never finds such a compatible version. From the definition of *compat*(T_i, x_j, y_l), this means that the following predicate is always true:

$$\begin{aligned} \forall (x, v, l) \in db : \; & V(w_l(x_l))[x] < V(r_i(y_j))[x] \\ & \vee \; V(w_l(x_l))[y] > V(r_i(y_j))[y] \end{aligned}$$

This means that there exists a version x_k upon which transaction T_i depends, and such that $V(w_k(x_k))[x] = V(r_i(y_j))[x]$. Transaction T_k committed at some site. As a consequence, Proposition 6 tells us that eventually T_k commits at process p. We conclude by observing that since Jessy satisfies both CONS and WCF, $V(w_k(x_k))[y] > V(r_i(y_j))[y]$ cannot hold.

□

Lemma 9. *For every transaction T_i, if T_i is submitted at coord(T_i) and coord(T_i) is correct, every correct process in replicas$(ws(T_i)) \cup$ coord(T_i) eventually decides T_i.*

Proof. According to Lemma 8 and the properties of atomic multicast, transaction T_i is delivered at every correct process in *replicas*$(ws(T_i)) \cup$ *coord*(T_i). It is then enqueued in variable $\mathcal{Q}$ (lines 10 to 11 in Algorithm 2).

Because $\mathcal{Q}$ is FIFO, processes dequeue transactions in the order they deliver them (lines 14 to 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 replica in *replicas*$(ws(T_i))$ eventually dequeues T_i, and sends the outcome of function *certify*(T_i) to *replicas*$(ws(T_i)) \cup$ *coord*(T_i) (lines 16 to 18).

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

Proposition 8. *Jessy satisfies WFQ.*

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. □

We now prove that Jessy satisfies obstruction-freedom for updates (OFU) and non-triviality for NMSI. These results are both stated in the case where Jessy employs non-partitioned dependence vectors. The question of ensuring any of these properties with a smaller space-complexity than $O(m)$ where m is the number of objects in the system remains open.

Proposition 9. *Jessy ensures non-trivial NMSI.*

Proof. Consider a replica p of x storing version x_j, and assume an extension of the execution in which p 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. Because Jessy use dependence vectors, Theorem 5 tells us that: $V(r_i(x_j))[x] \geq V(r_i(y_k))[x]$ and $V(r_i(x_j))[y] \leq V(r_i(y_k))[y]$ hold. According to the preconditions of operation *remoteRead*(x, T_i) and modification M1, process p returns version x_j to *coord*(T_i). □

Proposition 10. *Jessy satisfies OFU.*

Proof. Consider an execution ρ of Jessy and note $h = \mathfrak{F}(\rho)$ the history produced by ρ. Let T_i be an update transaction not executed in ρ. First of all, we observe that in any continuation of ρ during which $\mathit{coord}(T_i)$ is correct, from Lemma 9, $\mathit{coord}(T_i)$ eventually decides transaction T_i. Then, assume that T_i is not conflicting in some continuation $h' = \mathfrak{F}(\rho \sqsubseteq \rho')$ with any concurrent transaction in h'. This means that for every transaction T_j, if T_j conflicts with T_i, then T_i depends upon T_j. Accordingly to Theorem 5, the code at line 16 in Algorithm 2, and the definition of function $\mathit{certify}()$, transaction T_i commits in h'.

□

8 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 [41, 42] . Others consider that data can be partitioned into conflict sets [43], or that always aborting concurrent conflicting transactions [44] 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

					Time complexity		
Algorithm	Cons.	Genuine?	Multi-Master?	Message Complexity	Read-only	Global Update	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 [45]	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 [40].

Table 2: Comparison of partial replication protocols

background before it becomes visible.

More recently, Peluso et al. [45] 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 [46, 47].

9 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.

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.

Acknowledgments

We thank Sameh Elnikety and Vivien Quéma for insightful discussions and feedbacks.

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] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III, "Software transactional memory for dynamic-sized data structures," in *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, ser. PODC '03. New York, NY, USA: ACM, 2003, pp. 92–101.

[15] R. Guerraoui and M. Kapalka, "The semantics of progress in lock-based transactional memory," in *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '09. New York, NY, USA: ACM, 2009, pp. 404–415.

[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. 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.

[18] H. Attiya, E. Hillel, and A. Milani, "Inherent limitations on disjoint-access parallel implementations of transactional memory," in *SPAA*, ser. SPAA '09, 2009, pp. 69–78.

[19] 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.

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

[21] R. Guerraoui and M. Kapalka, "On the correctness of transactional memory," in *PPoPP*, ser. PPoPP '08, 2008, pp. 175–184.

[22] S. Peluso, P. Romano, and F. Quaglia, "Genuine replication, opacity and wait-free read transactions: can a stm get them all?" in *WTTM*, Madeira, Portugal, Jul. 2012.

[23] R. Guerraoui, T. A. Henzinger, and V. Singh, "Permissiveness in transactional memories," in *DISC*, Sep. 2008, pp. 305–319.

[24] D. Agrawal, G. Alonso, A. E. Abbadi, and I. Stanoi, "Exploiting atomic broadcast in replicated databases (extended abstract)." in *Proceedings of Euro-Par'97*. Springer-Verlag, 1997, pp. 496–503.

[25] I. Stanoi, D. Agrawal, and A. E. Abbadi, "Using broadcast primitives in replicated databases," in *Proceedings of ICDCS'98*. IEEE Computer Society, 1998, pp. 148–155.

[26] B. Kemme and G. Alonso, "Don't be lazy, be consistent: Postgres-r, a new way to implement database replication," in *The VLDB Journal*, 2000, pp. 134–143.

[27] U. Fritzke and P. Ingels, "Transactions on partially replicated data based on reliable and atomic multicasts." in *Proceedings of ICDCS'01*. IEEE Computer Society, 2001, pp. 284–291.

[28] F. Pedone, R. Guerraoui, and A. Schiper, "The database state machine approach." *Journal of Distributed and Parallel Databases and Technology*, vol. 14, no. 1, pp. 71–98, 2003.

[29] M. Patino-Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso, "Middle-r: Consistent database replication at the middleware level," *ACM Transactions on Computer Systems*, vol. 23, no. 4, pp. 375–423, 2005.

[30] Y. Lin, B. Kemme, M. Patiño Martínez, and R. Jiménez-Peris, "Middleware based data replication providing snapshot isolation," in *Proceedings of SIGMOD '05*. New York, NY, USA: ACM, 2005, pp. 419–430.

[31] L. Camargos, F. Pedone, and M. Wieloch, "Sprint: a middleware for high-performance transaction processing," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 3, pp. 385–398, 2007.

[32] F. Pedone and S. Frølund, "Pronto: High availability for standard off-the-shelf databases," *Journal of Parallel and Distributed Computing*, vol. 68, no. 2, pp. 150–164, 2008.

[33] D. Sciascia, F. Pedone, and F. Junqueira, "Scalable deferred update replication," in *DSN*, Jun. 2012, pp. 1–12.

[34] S. Peluso, P. Romano, and F. Quaglia, "SCORe: a scalable one-copy serializable partial replication protocol," in *Middleware*. Springer Berlin Heidelberg, Dec. 2012, pp. 456–475.

[35] 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.

[36] J. Gray, P. Helland, P. O'Neil, and D. Shasha, "The dangers of replication and a solution," in *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM Press, 1996, pp. 173–182.

[37] 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.

[38] 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.

[39] R. Guerraoui and A. Schiper, "Genuine atomic multicast in asynchronous distributed systems," *Theoretical Computer Science*, vol. 254, no. 1-2, pp. 297–316, Mar. 2001.

[40] 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.

[41] 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.

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

[43] 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.

[44] 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.

[45] S. Peluso, P. Ruivo, P. Romano, F. Quaglia, and L. Rodrigues, "When scalability meets consistency: Genuine multiversion update-serializable partial data replication," in *ICDCS*, Jun. 2012, pp. 455–465.

[46] 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.

[47] 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.

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