Non-Monotonic Snapshot Isolation

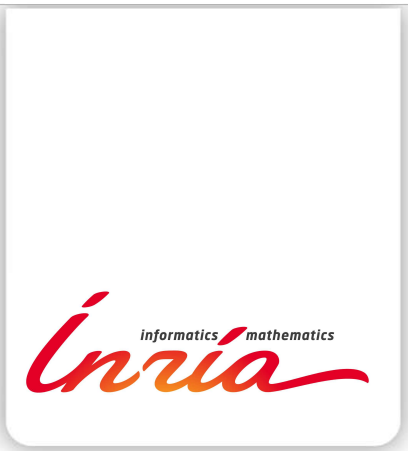
Masoud Saeida Ardekani, Pierre Sutra, Nuno Preguiça, Marc Shapiro

Non-Monotonic Snapshot Isolation

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

RESEARCH REPORT
N° 7805
November 2011
Project-Teams Regal

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



Non-Monotonic Snapshot Isolation*

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

Project-Teams Regal

Research Report n° 7805 — November 2011 — 34 pages

Abstract: We study two important properties for the scalability of a replicated system: genuine partial replication (GPR) and snapshot isolation (SI). We prove that these properties are incompatible. To side step this impossibility result, we propose a novel consistency criterion called Non-Monotonic Snapshot Isolation (NMSI). NMSI retains the most important properties of SI: read-only transactions always commit, and two concurrent write-conflicting updates never both commit. We also introduce a GPR protocol that ensures NMSI, and commits transactions faster and/or contacts fewer replicas than previous systems.

Key-words: oncurrency control, database machines, protocols, replicated databases, transaction processing

* The work presented in this paper has been partially funded by ANR projects Prose and Concordant.

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 ces deux propriétés sont incompatibles. Nous proposons ensuite un nouveau critère de cohérence nommé Non-Monotonic Snapshot Isolation (NMSI). NMSI est proche de SI et compatible avec GPR. Pour justifier ce propos, nous présentons un protocole authentique implémentant de manière efficace NMSI. Au regard des travaux précédents sur le contrôle de concurrence dans les systèmes répartis transactionnelles, notre protocole est plus performant en latence et/ou en nombre de messages échangés.

Mots-clés : contrôle de concurrence, base de données répliquées, transaction, protocoles

1 Introduction

Modern data stores replicate data for both performance and availability. Full replication does not scale well as every replica must replay every update. Partial replication addresses this problem by replicating only a subset of available data at each process. This allows the system to handle independent parts of the workload in parallel, and has the potential to scale better.

Many existing partial replication protocols however, still rely on a global consensus for both read (taking snapshots) and update operations. This means that they do not exploit the available parallelism well. In contrast, under *genuine* partial replication [1], only replicas of objects accessed by the transaction make steps to execute the transaction. Therefore, independent transactions do not interfere with each other. Genuine partial replication (GPR) leverages the intrinsic parallelism of workloads.

Another factor that limits scalability is the chosen consistency criteria. Some Internet-based applications sacrifice strong consistency criteria to circumvent the CAP impossibility [2] and increase scalability. However, it is necessary for modern interactive-based applications (e.g. social networks) to ensure a certain level of consistency, and to provide some balance between consistency and scalability.

This paper addresses the problem of defining a scalable multi-object consistency criterion which preserves strong object semantics. Snapshot isolation (SI) is a natural solution to this problem. Indeed, under SI read-only transactions do not interfere with update transactions, and with a few programming efforts, SI systems can produce serializable histories.

The first contribution of this paper is to show that no genuine partial replication system can ensure SI. Our impossibility result relies on the fact that the following four properties are necessary to achieve SI:

- No cascading aborts,
- Every transaction observes a strictly consistent snapshot, i.e., a snapshot that coincides with some point in time,
- Two concurrent write-conflicting update transactions never commit both, and
- Snapshots taken by transactions are totally ordered.

We show that an asynchronous message-passing system that supports GPR, cannot compute neither totally ordered snapshots, nor strictly consistent snapshots.

To overcome this impossibility, we propose a novel consistency criterion called *Non-Monotonic Snapshot Isolation* (NMSI). NMSI provides guarantees very similar to SI. In particular, transactions are wait-free, they always observe consistent snapshots, and two write-conflicting updates never both

commit. Snapshots taken under NMSI are however not strictly consistent, nor totally ordered.

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

Outline: We introduce our system model in Section 2. We deconstruct SI in Section 3. Section 4 shows that no GPR system supports SI. We introduce Non-Monotonic Snapshot Isolation in Section 5. Section 6 describes Jessy, our NMSI protocol. We survey related work in Section 7, then close in Section 8.

2 Model

In this section, we formally introduce the model used to represent histories, transactions, and the replication system. We also formalism the SI safety and progress conditions. Finally, genuine partial replication is defined.

Objects & transactions Let *Objects* be a set of objects, and $\mathcal{T}$ be a set of transaction identifiers. Given an object x, and transaction identifier i, x_i denotes version i of x. A *transaction* $T_{i\in\mathcal{T}}$ is a finite sequence of read and write operations followed by a terminating operation which can be either a commit (c_i) or an abort (a_i) operation. In a history (defined shortly), $w_i(x_i)$ denotes that transaction T_i writes version i of object x, and $r_i(x_j)$ means that T_i reads version j of object x. In a transaction, every object is read or write at most once, and every write is preceded by a read on the same object.[1]

We note $WS(T_i)$ the write set of T, i.e., the set of objects written by transaction T_i. Similarly, $RS(T_i)$ denotes the read set of transaction T_i. Two transactions *conflict* when they access the same object x and one of them executes a write to x. Two transactions *write-conflict* when they both write the same object.

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

[1] In practice, a transaction that reads the same object multiple times will read a transaction-cached version. For writes, only the last write affects the committed state.

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. 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 the different versions of an object, i.e., $\ll_h = \{(x_i, x_j) : \exists x, w_i(x_i) <_h w_j(x_j)\}$. Following Bernstein et al. [3], we depict a history as a graph. We illustrate this with history h_1 below in which transaction T_a reads the initial versions of objects x and y, while transaction T_1 (respectively T_2) updates x (resp. y).

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

When order $<_h$ is total, or it does not harm reasoning, we shall write an history as a permutation of operations; e.g., $h'_1 = r_1(x_0).r_2(y_0).w_2(y_2).c_1.c_2$.

Snapshot Isolation Snapshot isolation (SI) is a consistency criterion previously introduced by Berenson et al. [4], then later generalized by Elnikety et al. [5]. In this paper, we make no distinction between SI and GSI. Our definition is derived from the definition given by Adya [6] and Elnikety et al. [5]. To define SI, we introduce a function $\mathcal{S}$ over histories. Function $\mathcal{S}$ takes as input a history h, and returns an extended history h_s by adding a *snapshot point* to h for each transaction in h. Given a transaction T_i, the snapshot point of T_i in h_s, denoted s_i, precedes every operation of transaction T_i in h_s. A history h is in SI if and only if there exists a function $\mathcal{S}$ such that $h_s = \mathcal{S}(h)$ and h_s satisfies the following safety rules:

- **D1 (Read Rules)**
  $\forall r_i(x_{j \neq i}) \in h_s$:
  $D1.1.\ c_j \in h_s$
  $D1.2.\ c_j <_{h_s} s_i$
  $D1.3.\ \forall w_{k \neq j}(x_k), c_k \in h_s : c_k <_{h_s} c_j \vee s_i <_{h_s} c_k$

- **D2 (Write Rule)**
  $\forall c_i, c_j \in h_s$:
  $WS(T_i) \cap WS(T_j) \neq \{\} \Rightarrow (c_i <_{h_s} s_j \vee c_j <_{h_s} s_i)$

System A database $\mathcal{D}$ 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. We consider a message-passing distributed system of n processes $\Pi = \{p_1, \ldots, p_n\}$. Each process holds a copy of a subset of $\mathcal{D}$ as its local database. Initially every object x has

version x_0. For an object x, *Replicas*(x) denotes the set of processes (or replicas) that hold a copy of x. We make no assumption about how objects are replicated. For some transaction $T_{i\neq 0}$, we note *Replicas*(T_i) the set of processes replicating an object accessed by T_i. The coordinator of T_i, denoted *coord*(T_i) $\in$ *Replicas*(T_i), is in charge of executing T_i on behalf of some client (not modeled). The coordinator does not know in advance the read set or the write set of T_i. To model this, we consider that every prefix of a transaction (followed by a terminating operation) is a transaction with the same id.

We consider the global time model of Fischer et al. [7]. A *configuration* of the system consists of the internal state of each process, together with the content of the message buffer modeling inter-process communication. An *initial configuration* is one in which each process starts at an initial state and the message buffer is empty. 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*. For every object replicated in the system, we assume that during every execution at least one process is correct.

We note $\mathfrak{F}$ the refinement mapping [8] from executions to histories, i.e., if ρ is an execution of the system, then $\mathfrak{F}(\rho)$ is the history produced by ρ. A history h is *admissible* if there exists an execution ρ such that $h = \mathfrak{F}(\rho)$. We consider that given two sequences of steps U and V, if U precedes V in some execution ρ, then the operations implemented by U precedes (in the sense of $<_h$) the operations implemented by V in the history $\mathfrak{F}(\rho)$.

We consider that the system provides the following progress guarantees :

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

- **Wait-free Queries.** If *coord*(T_i) is correct and T_i is a read-only transaction then transaction T_i eventually commits.

SI read rules do not define what is the snapshot to be read. This implies, as stated by Adya [6], that "transaction T_i's start point need not be chosen after the most recent commit when T_i started, but can be selected to be some (convenient) earlier point" . To preclude read-only transactions to always see outdated data, we require in addition that the following progress property holds:

- **Non-trivial SI.** Consider an admissible history h such that a transaction T_i is pending in h, and the next operation of T_i is a read on

some object x. Note x_j the latest committed version of x in h. Let ρ be an execution with $\mathfrak{F}(\rho) = h$. If there is no concurrent conflicting transaction to T_i in h, and history $h' = h.r_i(x_j)$ is in SI, then there exists an admissible execution ρ' extending ρ such that $\mathfrak{F}(\rho') = h'$.

This paper addresses the question of constructing scalable partially replicated systems that support SI. Genuine partial replication [1] captures the goal that the throughput of the system scales linearly with the number of nodes when the workload is parallel:

- **Genuine Partial Replication.** A system is genuine if in every execution of the system, for every transaction T_i, only processes that hold objects accessed by T_i make steps to execute T_i.

3 Deconstructing SI

In this section, we introduce four disjoint properties that are necessary to attain SI. Our characterization *solely* relies on real transaction operations, i.e., read, write, commit, and abort, and does not employ abstract snapshot points. This helps us precisely understand the cost of implementing SI, and as we will see in Section 7, ease comparing different consistency criteria with almost similar isolation levels.

Roughly speaking, the following four properties are necessary to achieve SI: (1) there are no cascading aborts, (2) every transaction observes a strictly consistent snapshot, i.e., a snapshot that coincides with some point in time, (3) concurrent transactions both commit only if they do not write-conflict, and (4) snapshots taken by transactions are monotonic, i.e., they can be ordered. In the remaining of this section, we formally define each of these properties, then we prove that each property is necessary. Section 4 evaluates the cost of each of these properties.

3.1 Cascading Aborts

When a transaction T_i aborts, the system must wipe out its effects. As a consequence, if a transaction T_j depends on T_i, i.e., T_j observes the effect of T_i, the system must also abort T_j. This is called *cascading aborts* [9]. To avoid this costly effect, transactions should only observe committed states:

Definition (Avoiding cascading aborts)**.** *History h avoids cascading aborts, if for every read $r_i(x_{j\neq i})$ in h, c_j precedes $r_i(x_j)$ in h. We write $h \in ACA$ when h satisfies this property.*

3.2 Strictly Consistent Snapshots

Under SI a transaction T_i observes the state of the database as it appears at some point in the past. This implies that if we consider the time interval during which T_i can observe a version x_j of some object x, (SCONSa) the intervals for every object $x \in RS(T_i)$ intersect, and (SCONSb) there is no write over some object in $RS(T_i)$ prior to this intersection. We say that a history is *strictly consistent* when both SCONSa and SCONSb hold.
In $h = r_1(x_0).w_1(x_1).c_1.r_3(x_1).r_2(y_0).w_2(y_2).c_2.r_3(y_2).c_3$, transaction T_3 does not observe a strictly consistent snapshot because $r_3(x_1) <_h c_2$. Thus, y_2 cannot be observed when T_3 takes its snapshot. This problem is captured by SCONSa. Now consider history $h = r_1(x_0).w_1(x_1).c_1.r_1(y_0).w_2(y_2).$
$c_2.r_3(x_0).r_3(y_2).c_3$. Since $c_1 <_h c_2$, transaction T_3 cannot take a snapshot of the system such that it only observes the effect of T_2 and not T_1. SCONSb captures this issue. Formally, strictly consistent snapshots are defined in the following way.

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

$$\forall r_i(x_j), r_i(y_l) \in h,\ r_i(x_j) \not<_h c_l \qquad (SCONSa)$$
$$\forall r_i(x_j), r_i(y_l), w_k(x_k) \in h, c_k <_h c_l \Rightarrow x_k \ll_h x_j \qquad (SCONSb)$$

3.3 Snapshot Monotonicity

SCONS does not capture completely how transactions behave under SI. For instance, history h_2 below is in SCONS, but does not belong to SI (transaction T_a violates rule D1.3). The problem stems from the fact that T_a and T_b do not take monotonic snapshots, i.e., T_a reads $\{x_0, y_1\}$, while T_b reads $\{x_1, y_0\}$. We call this novel phenomenon *non-monotonic snapshots*.

$$h_2 : \begin{array}{ll} r_a(x_0) \longrightarrow & r_1(x_0).w_1(x_1).c_1 \longrightarrow\ r_b(x_1).c_b \\ r_b(y_0) \longrightarrow & r_2(y_0).w_1(y_2).c_2 \longrightarrow\ r_a(y_2).c_a \end{array}$$

Rule D1 implicitly disallows non-monotonic snapshots. Indeed, SI ensures that snapshots taken by all transactions should be monotonic with respect to the version order on objects induced by the history. We formalize below the monotonicity of snapshots.

Definition (Snapshot precedence)**.** *When in a history h, a transaction T_i reads a version x_k and transaction T_j reads a version x_l, and $x_k \ll_h x_l$ holds, we say that the snapshot read by T_i* precedes *the snapshot read by T_j, written $T_i \rightarrow T_j$.*

Definition (Snapshot monotonicity)**.** *Given some history h, if the relation $\rightarrow^*$ induced by h is a partial order, the snapshots in h are* monotonic.[2] *We write $h \in MON$ when h satisfies this property.*

3.4 Write-conflict Free

The last property in our deconstruction of SI constraints the behavior of concurrent writes. While under serializability (SER) concurrent conflicting updates can be reordered to produce a sequential history, under SI, they cannot both commit. This property is captured by rule D2.

In our model every update transaction that writes x must read it previously. This implies that every update transaction depends on a previous update transaction, or the initial transaction T_0. Therefore, two concurrent transactions are always independent. We use this characterization to define the behavior of concurrent writes.

Definition (Dependency)**.** *Consider a history h, and two transactions T_i and T_j in h. We note $T_i \rhd T_j$ when $r_i(x_j)$ is in h. By extension, transaction T_i* depends *on transaction T_j when $T_i \rhd^* T_j$ holds.*

Definition (Write-conflict free)**.** *A history h is write-conflict free if two independent transactions do not write the same object. We denote by WCF the histories that satisfy this property.*

3.5 The deconstruction

This section proves that a history h is in SI only if (1) h avoids cascading aborts, (2) every transaction in h observes a strictly consistent snapshot, (3) relation $\rightarrow^*$ is a partial order, and (4) h is write-conflict free. We first prove two technical lemmata, then we establish this result.

Lemma 1. *Consider a history $h \in SI$, and two versions x_i and x_j of some object x. Relation $x_i \ll_h x_j$ holds only if $T_j \rhd^* T_i$ holds.*

Proof. History h is in SI and $x_i \ll_h x_j$ holds. 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 over x between $w_i(x_i)$ and $r_j(x_k)$. Since h is in SI, it follows by rules D1 and D2 that T_j reads version x_i, i.e., $k = i$. Hence, T_j depends on T_i. Otherwise, we conclude by applying inductively the previous reasoning. □

[2]For some binary relation $\mathcal{R}$, we note $\mathcal{R}^*$ its transitive closure.

Lemma 2. *Let h be a history in SI. For every two transactions T_i and T_j, and every relation $\mathcal{S}$ such that $\mathcal{S}(h)$ satisfies D1 and D2, if $T_i \rightarrow T_j$ holds, then we have: $s_i <_{h_s} s_j$.*

Proof. Consider two transactions T_i and T_j such that $T_i \rightarrow T_j$ holds. By definition of the precedence relation, there exist two versions x_k and x_l of some object x such that in h, T_i reads version x_k, T_j reads version x_l, and $x_k \ll_h x_l$ holds. Consider some relation $\mathcal{S}$ that satisfies rules D1 and D2 for h, and note $h_s = \mathcal{S}(h)$.

Since h_s satisfies D2, and $x_k \ll_h x_l$ holds, we obtain that (F1) $c_k <_{h_s} s_l$. Applying D13 to transactions T_i and T_l, we know that either (C1) $c_l <_{h_s} c_k$, or (C2) $s_i <_{h_s} c_l$ is true. Case C1 cannot occurs, since it contradicts fact F1. From property D12, we know that (F2) $c_l <_{h_s} s_j$ holds. Facts F2 and C2 imply that $s_i <_{h_s} s_j$ hold. □

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

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

- $h \in$ ACA : Follow from rules D11 and D12.
- $h \in$ WCF : Consider two concurrent transactions T_i and T_j writing both some object x. Assume $x_i \ll_h x_j$. Lemma 1 tells us that T_j depends on T_i. If now $x_j \ll_h x_i$, applying again Lemma 1, we conclude that T_i depends on T_j.
- $h \in$ SCONSa: By contradiction. Assume four transactions T_i, T_j, $T_{k \neq j}$ and T_l such that: $r_i(x_j), r_i(y_l) \in h \wedge r_i(x_j) <_h c_l$. 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 D12.
- $h \in$ SCONSb: Assume for the sake of contradiction that there exist four transactions T_i, T_j, $T_{k \neq j}$ and T_l such that $r_i(x_j), r_i(y_l), w_k(x_k) \in h \wedge c_k <_h c_l \wedge x_j \ll_h x_k$ holds. Since transaction T_j and T_k both write x, by rule D2, $s_k <_{h_s} c_j \vee c_k <_{h_s} s_j$ holds. In conjunction with the fact that $x_j \ll_h x_k$ holds, we know that c_j precedes c_k in h_s. Thus,s $c_j <_h c_k <_h c_l$ holds. Because transaction T_i reads version y_l, snapshot point s_i must be placed after c_l. But according to the definition of $\mathcal{S}$, s_i must precede every operation of T_i. This leads us to the desired contradiction.
- $h \in MON$: Relation $\rightarrow^*$ is by definition transitive. Hence, if $\rightarrow^*$ is not a partial order, there exists a set of transactions $T_1, \ldots, T_n$ such that $T_1 \rightarrow \ldots \rightarrow T_{n \geq 1} \rightarrow T_1$ holds. Applying Lemma 2 to $h \in$ SI implies that relation $s_1 <_{h_s} s_1$ holds. A contradiction.

□

At the light of the previous theorem, the next section establishes several impossibility results regarding SI.

4 The inherent limitations of SI

In this section, we show that no GPR system implements MON. This implies that GPR and SI are two incompatible properties. Further, we turn our attention to the cost of implementing strictly consistent snapshot. We show that without synchrony assumptions SCONSb is not attainable, then that no GPR system provides SCONSa.

4.1 On the cost of computing strictly consistent snapshots

In what follows, we consider some GPR system Π satisfying the progress and safety requirements we described in Section 2. We first state an extensibility property over histories admissible by Π, then we establish our results.

Lemma 3. *Let $h = \mathfrak{F}(\rho)$ be an admissible history such that a transaction T_i is pending in h. Note X the objects accessed by T_i in h. Only processes in $Replicas(X)$ make steps to execute T_i in ρ.*

Proof. Suppose for the sake of contradiction 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 terminates without executing new operations in ρ', and $coord(T_i)$ is correct in ρ'. The progress requirements satisfied by the system implies that T_i terminates in ρ'. However, process $p \notin Replicas(X)$ makes steps to execute T_i in ρ'. Contradiction. □

Theorem 2. *No GPR system implements MON.*

Proof. By contradiction. Consider (1) some objects x,y,z and u, (2) four read-only transactions T_a, T_b, T_c and T_d accessing respectively $\{x, y\}$, $\{y, z\}$, $\{z, u\}$ and $\{u, x\}$, and (3) four update transactions T_1, T_2, T_3 and T_4 accessing respectively x, y, z and u.

History $r_b(y_0)$ is admissible because Π implements non-trivial SI. Then, since updates are obstruction-free, history $r_b(y_0).r_2(y_0).w_2(y_2).c_2$ is admissible. Applying again that Π satisfies non-trivial SI, we obtain that $r_b(y_0).r_2(y_0)$ $.w_2(y_2).c_2.r_a(y_2).r_a(x_0)$ is admissible. Since transaction T_a is wait-free, then history $h = r_b(y_0).r_2(y_0).w_2(y_2).c_2.r_a(y_2).r_a(x_0).c_a$ is admissible. Using a

symmetrical reasoning, we conclude that $h' = r_d(u_0).r_4(u_0).w_4(u_4).c_4.r_c(u_4).r_c(z_0).c_c$ is admissible.

Because Π satisfies GPR, Lemma 3 implies that only processes in $\mathit{Replicas}(x) \cup \mathit{Replicas}(y)$ make steps in ρ (with $\mathfrak{F}(\rho) = h$). Similarly, only processes in $\mathit{Replicas}(u) \cup \mathit{Replicas}(z)$ make steps in ρ' (with $\mathfrak{F}(\rho') = h'$). Thus, $\rho.\rho'$ and $\rho.\rho'$ are undistinguishable. This implies that $h.h' = h'.h$. We note hereafter ρ_0 the execution $\rho.\rho'$.

Because updates are obstruction-free, history $h.h'.w_3(z_3).c_3$ is admissible. Then, since (1) T_b is pending in $h.h'.w_3(z_3).c_3$, (2) concurrent transactions to T_b neither accesses z, nor y in $h.h'.w_3(z_3).c_3$, and (3) Π satisfies non-trivial SI, history $h_1 = h.h'.r_3(z_0).w_3(z_3).c_3.r_b(z_3).c_b$ is admissible. Note U_1 the sequence of steps satisfying $\mathfrak{F}(\rho_0.U_1) = h_1$. Applying 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 admissible, Note U_2 the sequence of steps such that $\mathfrak{F}(\rho_0.U_2) = h_2$ holds.

Because Π is GPR, only processes in $Q_1 = \mathit{Replicas}(y) \cup \mathit{Replicas}(z)$ execute steps in U_1. Similarly in U_2, only processes in $Q_2 = \mathit{Replicas}(x) \cup \mathit{Replicas}(u)$ make steps. The system Π makes no assumption about how objects are replicated. In particular, we can consider that Q_1 and Q_2 are disjoint. It follows that execution $\rho.U_1.U_2$ is an execution of Π. Therefore, $h'.h.r_1(x_0).w_1(x_1).c_1.r_d(x_1).c_d.r_3(z_0).w_3(z_3).c_3.r_b(z_3).c_b$ is admissible. This execution violates MON. Contradiction. □

As a corollary of the theorem above and the decomposition result we gave in Section 3, we conclude that SI and GPR are incompatible.

Theorem 3. *SI and GPR are incompatible.*

Proof. Follows from the conjunction of Theorem 1 and Theorem 2. □

4.2 On the cost of computing strictly consistent snapshots

In what follows, we show that in an asynchronous message-passing system, a choice must be made between SCONSb and GPR. Our impossibility result follows a schema similar to the proof of Theorem 4 in [10]. To establish this result, we construct an infinite execution of a read-only transaction T_a that never terminates. During the execution, we interleave between any two consecutive steps to execute T_a, a transaction updating one of the objects read by T_a. We first define a special (finite) execution of this form, called flippable [10], and show that during such an execution transaction T_a does not terminate successfully (Lemma 4). Next, we prove that if no synchrony

assumptions holds, we can extend the flippable execution *ad eternam*, contradicting the fact that read-only transaction are wait-free (Theorem 4).

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

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

is called flippable.

Lemma 4. *In a flippable execution ρ satisfying $\mathfrak{F}(\rho) \in$ SCONSb, query T_a does not terminate*

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

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

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

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

(Case $p_k \in Q_k$.) By a similar reasoning, we obtain that ρ is undistinguishable from $\rho' = \dots U_j \dots U_k U_l s_k \dots$.

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

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

Theorem 4. *No asynchronous GPR system implements SCONSb.*

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

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

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

By construction, execution ρ is flippable. Since every process in $\mathcal{P}$ eventually make a step in ρ, ρ is fair. Because there is no synchrony assumptions, this execution is admissible. In ρ, transaction T_a never commits. Contradiction. □

Our last impossibility result establishes that GPR and SCONSa are two incompatible properties.

Theorem 5. *No GPR system implements SCONSa.*

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

Since updates are non-blocking, history $r_1(x_0).w_1(x_1).c_1$ is admissible. The system Π satisfies non-trivial SI, hence history $r_1(x_0).w_1(x_1).c_1.r_a(x_1)$ is also admissible. Again, because the system supports non-blocking updates, history $h_1 = r_1(x_0).w_1(x_1).c_1.r_a(x_1).r_2(y_0).w_2(y_2)$ is admissible. Note ρ_1 an execution such that $\mathfrak{F}(\rho_1) = h_1$.

In execution ρ_1, note U_1, U_2 and U_a the steps made to execute respectively T_1, T_2 and T_a. Execution ρ_1 is of the form $\rho = U_1.U_a.U_2$. Applying Lemma 3, only processes in $Replicas(x)$ make steps to execute T_a in ρ. It follows that ρ_1 is undistinguishable from $\rho_2 = U_1.U_2.U_a$. Note h_2 the history $\mathfrak{F}(\rho_2) = r_1(x_0).w_1(x_1).c_1.r_2(y_0).w_2(y_2).c_2.r_a(x_1)$.

Applying the fact that Π satisfies non-triviality, and that read-only transactions are wait-free. there exists a run ρ_3 extending ρ_2 such that $\mathfrak{F}(\rho_3) = h_2.r_a(y_2).c_2$. Note V_a the steps made to execute $r_a(y_2).c_2$. Execution ρ_3 equals $U_1.U_2.U_a.V_a$. This execution is undistinguishable from $\rho_4 = U_1.U_a.U_2.V_a$. Hence, history $h_4 = r_1(x_0).w_1(x_1).c_1.r_a(x_1).r_2(y_0).w_2(y_2).r_a(y_2).c_2$ is admissible. This history is not in SCONSa. Contradiction. □

The above theorems assess the cost of implementing SI in message-passing distributed systems. Theorem 2 establishes that we cannot construct monotonic snapshots under GPR. Theorem 3 and Theorem 4 prove that no GPR system can provide strictly consistent snapshot without relying on some synchrony assumptions, even if no concurrent writes occurs. All these results hinder the scalability of SI.

5 Non-Monotonic Snapshot Isolation

To overcome the above restrictions, we propose a novel consistency criterion called Non-Monotonic Snapshot Isolation (NMSI). NMSI retains the most important properties of SI, namely that read-only transactions always observe consistent snapshots, and that two conflicting concurrent updates never both commit. Snapshots taken under NMSI are not strictly consistent, nor monotonic. NMSI relax these two requirements to improve scalability. Indeed, as we shall see in Section 6, this allows read-only transactions to commit locally without synchronization.

5.1 Definition

Roughly speaking, a transaction that misses an effect of another transaction upon which it depends, observes a non-consistent snapshot [11]. We formalize this concept below.

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

Consistency	G-1	G-Single	G-SIa	G-SIb
Consistent View (PL-2+)	X	X	-	-
Forward Consistent View (PL-FCV)	X	X	-	X
NMSI	X	X	X	-
SI	X	X	X	X

Table 1: Phenomena disallowed (X)

One can show that under SI, transactions always observe consistent snapshots (SI $\subseteq$ SCONS $\subseteq$ CONS). Moreover, as we have seen in Section 3, these snapshots are monotonic (SI $\subseteq$ MON). However, neither MON, nor SCONS scales because of the impossibility results. NMSI allows transactions to observe non-monotonic snapshots of concurrent but non-conflicting transactions:

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

For instance, history h_2 in Section 3.3 is allowed by NMSI. NMSI generalizes the correctness property of atomic snapshot protocols [12] that permits processes to observe non-monotonic consistent snapshots. In addition, NMSI captures the transactional link between different events, and it forbids two concurrent conflicting updates to both commit.

5.2 NMSI Anomalies

To better understand NMSI, we give across this section a phenomena-based characterization of our consistency criteria. We use the classification introduced by Adya [6], i.e., phenomena G-1, G-SIa, G-SIb, and G-Single. Each of these undesirable phenomena are informally described as follows:

(G-1) Dirty reads and dirty writes are happened.
(G-SIa) Concurrent transactions observe each others effects or modify the same objects.
(G-SIb) A transaction does not observe the effects of some other transaction that commit before it.
(G-Single) A transaction does not observe the effects of some other transaction that it depends on.

SI precludes all the above phenomena [6]. Propositions 1 to 3 below establish which of the mentioned phenomena are disallowed by ACA, WCF and CONS, hence by NMSI.

Proposition 1. *History h does not observe phenomenon G-single if h belongs to CONS ∩ WCF*

Proof. By contradiction, assume a history h such that $h \in \text{CONS} \cap \text{WCF}$ and phenomenon G-single is observed.

Therefore, there exists a directed cycle with exactly one anti-dependency edge in DSG(h).

Case a) There is a ww edge in the cycle between T_i and T_j while there is not a wr edge between them. Thus, $T_i \parallel T_j \wedge ws(T_i) \cap ws(T_j) \neq \varnothing \Rightarrow h \notin$ WCF. A contradiction to the assumption.

Case b) There is not any ww edge in the cycle. Therefore, the cycle comprises of several wr edges and one rw edge from T_i to T_j. Since there is a path with only wr edges from T_j to T_i, we have $T_i \rhd^* T_j$. Moreover, rw edge enforce that $\exists x_k : r_i(x_k) \in h \wedge w_j(x_j) \in h : x_j \gg x_k$. Thus, $h \notin$ CONS. A contradiction to the assumption. □

Proposition 2. *History h does not observe phenomenon G-1 if h is in ACA∩ WCF*

Proof. In order to prove the proposition, we prove that non of G-1 cases (G-1a, G-1b, G-1c) will be observed if $h \in \text{ACA} \cap \text{WCF}$.
G-1a (Aborted Reads) cannot be observed in h because $h \in$ ACA ensures that every read operation in history h reads a committed value.
G-1b (Intermediate Reads) cannot be observed since it is considered that a transaction can modify an object only once.
By contradiction assume that $h \in \text{ACA} \cap \text{WCF}$ and G-1c (Circular Information Flow) is observed. Thus, there should be a directed cycle in DSG(h) containing only ww and wr dependency edges.
Case a) There is a ww edge in the cycle between T_i and T_j while there is not a wr edge between them. Thus, $T_i \parallel T_j \wedge ws(T_i) \cap ws(T_j) \neq \varnothing \Rightarrow h \notin$ WCF. A contradiction to the assumption.
Case b) All the edges in the cycle are labeled with wr-dependency. It is straightforward to show that this phenomena violates the assumption that $h \in$ ACA. □

Proposition 3. *If history h belongs to ACA∩WCF, then phenomenon G-SIa does not occur.*

Proof. An augmented history $S(h)$ exhibits phenomenon G-SIa if $SSG(S(h))$ contains a rw-dependency or ww-dependency edge from T_i to T_j without there also being a start-dependency edge from T_i to T_j [13].

Consider some history h such that h belongs to ACA ∩ WCF. Case a) assume by contradiction that there is only ww-dependency edge between T_i

and T_j without start or wr dependency edge from T_j to T_i. Thus, $T_i \parallel T_j \wedge ws(T_i) \cap ws(T_j) \neq \varnothing \Rightarrow h \notin$ WCF. A contradiction to the assumption that $h \in$ WCF. Case b) assume by contradiction that there is only wr-dependency edge between T_i and T_j without start dependency edge from T_j to T_i. Therefore, the snapshot point of T_i precedes the commit point of T_j. Therefore, $r_i(x_j) <_h c_j$. A contradiction to the assumption that $h \in$ ACA. □

Table 1 depicts a phenomena-based comparison of NMSI to similar consistency criteria [6].

5.3 Mitigating Anomalies

While observing non-monotonicity can be considered as a small cost to boost scalability, it is still undesirable. This section investigates specific solutions to mitigate this phenomenon in the system.

5.3.1 Ensuring Causality

Non-monotonicity is particularly undesirable if it occurs at some client,i.e., when some client observes a version x_i of object x, then a version x_j with $x_j \ll x_i$. Such a situation can be remedied as follows.

- If a client is stateless, it can always sends its transactions to the same coordinator for execution. In this case, the coordinator maintains a list of the most recent versions of objects read by some client. Upon executing a new transaction T for the client, if $Coordinator(T)$ receives a object with a version older than the one in its list, it executes the operation again.
- If a client is stateful, it can send its transactions to different coordinators for execution. However, it should maintains a list of most recent versions of object read by itself before. Upon sending a transaction to a coordinator for execution, it also sends recent versions of object to the coordinator as well. As in the previous case, the coordinator ensures that versions of objects returned to the client are up-to-date.

5.3.2 Grouping Related Data

While ensuring snapshot monotonicity is desirable in transactional systems, observing non-monotonic snapshots in some cases is tolerable. In particular, when there is no correlation (either implicit, or explicit) between objects. Consider for instance history h_2 in Section 3.3. Assume that objects x and y store inventories of two different departments. If the two departments are

unrelated, having transactions T_a and T_b that take non-monotonic snapshots of the system does not harm correctness. In general, partitioning objects such that related objects are replicated together, removed any bad phenomenon due to non-monotonicity.

6 Protocol

This section describes Jessy, a genuine partial replication protocol that ensures NMSI. Jessy is divided to two parts: an execution protocol and a termination protocol. The former is in charge of executing transactions, and the latter guarantees that every execution produces a NMSI history. In Jessy, each read operation on an object x executes asynchronously at some replica of x. To ensure that a transaction accesses a consistent snapshot, Jessy makes use of novel data type called *dependence vectors*. We first give an overview of Jessy, and we introduce dependence vectors. Then, we describe the execution and termination protocol.

6.1 Overview

A transaction T_i executed by Jessy can be in one of four states at some process: *Executing*, *Submitted*, *Committed* or *Aborted* as follows:

- *Executing*: Each non-terminal operation $o_i(x)$ in T_i is optimistically executed at $coord(T_i)$. (We shall see shortly what happens when x is not replicated locally.) If $o_i(x)$ is a write, $coord(T_i)$ stores the corresponding update value locally. Storing updates serves two purposes: (1) subsequent operations see modifications made by $o_i(x)$, and (2) if transaction T_i commits, remote replicas apply the new values without re-executing all the operations. If $o_i(x)$ is a read, $coord(T_i)$ returns the response value.

- *Submitted*: Once all read and write operations of T_i have been executed, the transaction terminates, and $coord(T_i)$ submits to the termination protocol. The termination protocol executes 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 database. If T_i aborts, T_i enters the *Aborted* state. In both cases, a response value is returned.

6.2 Dependence vectors

When a transaction T_i is *global*, i.e., one of the objects T_i accesses is not replicated at $coord(T_i)$, the system must ensure that T_i observes a consistent snapshot. To this goal, Jessy assigns a special version vector, called dependence vector, to each version of each object. The dependence vector of some version x_i reflects all the versions read by T_i, or read by transactions that are causally related to T_i, as well as all the writes performed by T_i. In more detail, we define a dependence vector as follows:

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

$$\begin{aligned} &DV(r_i(x_0)) = 0^{|Objects|} \\ &DV(r_i(x_j)) = DV(w_j(x_j)) \\ &DV(w_i(x_i)) = Max_{y_j \in RS(T_i)} DV(r_i(y_j)) + \Sigma_{z \in WS(T_i)} 1_z \end{aligned}$$

where 1_z is the vector that equals 1 on component z, and 0 elsewhere.

To illustrate how dependence vectors work, let us consider history h_3 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.

$$h_3 : \begin{array}{l} r_1(x_0).w_1(x_1).c_1 \searrow \\ \qquad\qquad\qquad\quad r_3(x_1).r_3(y_2).w_3(y_3).c_3 \\ r_2(y_0).w_2(y_2).c_2 \nearrow \end{array}$$

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 $\langle 1, 0\rangle + \langle 0, 1\rangle + \langle 0, 1\rangle$.

Using dependence vectors, Jessy can compute a consistent snapshot for transaction T_i by ensuring that every pair of versions x_l and y_j read by T_i satisfy the following rule: $DV(r_i(x_l))[x] \geq DV(r_i(y_j))[x]$ (Proposition 6).

To prove Proposition 6, we first prove in Proposition 4 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. Further, in Proposition 5, we show that if $DV(w_j(y_j))[x] > DV(w_i(x_i))[x]$ holds, then transaction T_j depends on transaction T_i.

Proposition 4. *Consider two transactions T_i and T_j in h, then:*

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

Proof. ($\Rightarrow$): We first prove that : $T_i \rhd T_j \Rightarrow \forall x, y \in Objects, \forall w(x), w(y) \in T_i \times T_j, DV(w_i(x_i)) > DV(w_j(y_j))$.

$T_i \rhd T_j \Rightarrow \exists r_i(z_j) \in T_i \wedge \exists w_j(z_j) \in T_j$. According to dependence vector definition, we have $DV(r_i(z_j)) < DV(w_i(x_i))$, and $DV(r_i(z_j)) = DV(w_j(z_j)) = DV(w_j(y_j))$. Thus, $DV(w_j(y_j)) < DV(w_i(x_i))$. By induction, it is straightforward to show that $DV(w_i(x_i)) > DV(w_j(y_j))$ holds for every write operation in T_i and T_j if $T_i \rhd^* T_j$.
($\Leftarrow$): Since $DV(w_i(x_i)) > DV(w_j(y_j))$, and from the definition of dependence vector, there should exist $r_i(y'_{j'}) \in T_i$ such that:
Case 1) $DV(w_{j'}(y'_{j'})) = DV(w_j(y_j))$. Thus, $j' = j$, and $T_i \rhd T_j$
Case 2) $DV(w_{j'}(y'_{j'})) > DV(w_j(y_j))$ and $T_i \rhd T_{j'}$. By applying inductively the previous reasoning, eventually, case 1 will be the only case that hold. Thus, $T_i \rhd T_{j'} \rhd T_{j''} \ldots \rhd T_j$. Therefore, $T_i \rhd^* T_j$.

□

Proposition 5. *Consider a write conflict history free h and two transactions T_i and T_j in h. If $DV(w_j(y_j))[x] > DV(w_i(x_i))[x]$ then $T_j \rhd^* T_i$.*

Proof. Case $x \in WS(T_j)$: Therefore, $DV(w_j(x_j))[x] > DV(w_i(x_i))[x]$. Since history is write conflict free, and according to dependence vector definition, we have $DV(w_j(x_j)) > DV(w_i(x_i))$. From Proposition 4, we have $T_j \rhd^* T_i$.

Case $x \notin WS(T_j)$: Thus, $\exists r_j(z_{j'}) \in T_j : DV(r_j(z_{j'}))[x] > DV(w_i(x_i))$. According to dependence vector definition, $DV(w_{j'}(z_{j'}))[x] > DV(w_i(x_i))$ holds. Therefore, by applying inductively the same reasoning, we will have $T_j \rhd^* \ldots \rhd^* T_{j'} \rhd^* T_i \Rightarrow T_j \rhd^* T_i$.

□

Proposition 6. *Consider a history h in WCF, and a transaction T_i in h. Transaction T_i sees a consistent snapshot during h, if an only if for every pair of versions x_l and y_j read by T_i , $DV(r_i(x_l))[x] \geq DV(r_i(y_j))[x]$ holds.*

Proof. ($\Rightarrow$): By contradiction, assume that $\exists x_k, y_{k'} \in RS(T_i) : DV(r_i(x_k))[x] < DV(r_i(y_{k'}))[x]$
$\Rightarrow DV(w_k(x_k))[x] < DV(w_{k'}(y_{k'}))[x]$

Case 1) $DV(w_k(x_k)) < DV(w_{k'}(y_{k'}))$:
From Proposition 4, we have $T_i \rhd^* T_{k'} \rhd^* T_k$. If transaction k' writes on x, we have $x_k \ll x_{k'}$. A contradiction to the consistent snapshot assumption. If transaction k' does not write on x, then from dependence vector definition, and inductively, it is easy to show that there must exist a transaction T_j that writes on x such that $T_{k'} \rhd^* T_j \rhd T_k$. Therefore, we have $T_i \rhd^* T_j \rhd^* T_k$. Since both T_j and T_k writes on x, and $T_j \rhd T_k$, $x_j \gg x_k$. A contradiction to

the consistent snapshot assumption.
Case 2) $DV(w_k(x_k))[x] < DV(w_{k'}(y_{k'}))[x] \wedge DV(w_k(x_k))[z] > DV(w_{k'}(y_{k'}))[z]$: From Proposition 5, we have that $T_{k'} \rhd^* T_k$, and from Proposition 4, we have $DV(w_k(x_k)) < DV(w_{k'}(y_{k'}))$. A contradiction.

($\Leftarrow$): By contradiction, assume that $\exists T_j \in h, \exists x_k \in RS(T_i) : T_i \rhd^* T_j \wedge x_k \ll x_j$. From Proposition 4, we have $T_j \rhd^* T_k$. Thus, $T_i \rhd^* T_j \rhd^* T_k$.

Case 1) $T_i \rhd T_j \rhd^* T_k \Rightarrow \exists r_i(x_j), r_i(x_k) \in T_i$. This case is impossible since a transaction cannot read two different versions of the same object.

Case 2) $T_i \rhd T_{j'} \rhd T_j \rhd^* T_k \Rightarrow \exists r_i(z_{j'}), r_i(x_k) \in T_i$. From Proposition 4, we have that: $DV(w_{j'}(z_{j'})) > DV(w_j(x_j)) > DV(w_k(x_k)) \Rightarrow DV(w_{j'}(z_{j'}))[x] > DV(w_k(x_k))[x]$.
Therefore, $\exists r_i(z_{j'}), r_i(x_k) \in T_i : DV(w_{j'}(z_{j'}))[x] > DV(w_k(x_k))[x]$, a contradiction to the assumption.

Case 3) $T_i \rhd^* T_{j'} \rhd T_j \rhd^* T_k$. By applying inductively the reasoning presented in case 1 and 2, it is straightforward to prove the contradiction to the assumption for this case as well.

□

6.3 Execution Protocol

Algorithm 1 depicts the pseudo code of execution protocol executed at every replica in the system. The execution protocol of Jessy is in charge of executing transaction T_i at *coord*(T_i) (actions *execute*()), and providing access to local objects for remote transactions (action *readResolve*()). This protocol uses five variables: *db* contains the local database, *submitted* contains locally-submitted transactions, *committed* (respectively *aborted*) stores committed (respectively aborted) transactions, and *updates* contains the updates of each transaction.

Upon receiving a read request for some object x, *coord*(T_i) checks *updates* to see if x has been previously updated. If so, it returns the corresponding value from *updates* (line 15). Otherwise, *coord*(T_i) sends a READ_RESOLVE request to replicas that holds x (lines 18 to 19). When the execution protocol receives a READ_RESOLVE request for some object x, it returns a version of x which complies with Proposition 6 if possible (lines 5 to 9).

When the execution protocol executes a write request, it stores updated values in *updates* variable (line 12). This variable will be sent to all replicas

Algorithm 1 Execution Protocol of Jessy

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

holding an object that is modified inside T_i during commitment.

When transaction T_i terminates, T_i is submitted to the termination protocol (line 22) The execution protocol then waits until T_i is committed or aborted (lines 23 to 25), and returns the corresponding value. In Algorithm 1, *decided* is a shorthand for *committed* $\cup$ *aborted*.

6.4 Termination Protocol

Algorithm 2 depicts the code of the termination protocol of Jessy. It accesses the same five variables *db*, *submitted*, *committed*, and *updates*, along with a FIFO queue named *queue*. The termination protocol makes use of an atomic multicast service. We first describe this service, then we explain how the protocol handles the termination of a transaction.

Genuine Multicast Atomic multicast [1] sends messages atomically to a subset of processes in the system. It is defined by primitives AM-Cast and AM-Deliver. Operation AM-Cast takes as input a message m, and sends it to some set $m.dst$ of processes. Operation AM-Deliver delivers a message. Given two messages m and m', relation $m <_{amcast} m'$ holds iff there exists a process p such that p delivers m then m'. Atomic multicast satisfies the following properties: (*Uniform Integrity*) For any process p and any message m, p AM-Delivers m at most once, and only if p belongs to $m.dst$, and m was previously AM-Cast. (*Validity*) If a correct process p AM-Casts a message m, then eventually every correct process $q \in m.dst$ AM-Delivers m. (*Uniform Agreement*) If a process p AM-Delivers a message m, then eventually every correct process $q \in m.dst$ AM-Delivers m. (*Uniform Prefix Order*) For any two messages m and m' and any two processes p and q, if p AM-Delivers m, q AM-Delivers m' and $\{p, q\} \subseteq m.dst \cap m'.dst$, then either p AM-Delivers m' before m, or q AM-Delivers m before m'. (*Uniform Acyclic Order*) The transitive closure of relation $<_{amcast}$ is a strict partial order over the set of messages.

To guarantee genuine partial replication, Jessy uses a *genuine* atomic multicast service [14]: during a run, if a process p sends or receives a message m, then either p has multicast m, or p belongs to $m.dst$.

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

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

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

A replica q can safely decide on the outcome of T_i when q received votes from a *voting quorum* Q for T_i. A voting quorum Q for T_i is a set of replicas such that for every object x written by T_i, the set Q contains at least one of the process replicating x. Formally, a set of processes is a voting quorum for

T_i iff it belongs to $vquorum(T_i)$, defined as follows:

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

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

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

To commit transaction T_i, process p first applies T_i's updates to its local database, then p adds T_i to variable *committed* (lines 20 to 23). If instead T_i aborts, p adds T_i to *aborted* (lines 26 to 27).

6.5 Correctness of Jessy

In this section, we sketch a correctness proof of Jessy. First, we establish that Jessy generates NMSI histories in Proposition 7. Then, Proposition 8 shows that read-only transactions are wait-free, Proposition 9 proves that updates are obstruction-free. Finaly, we show that Jessy satisfies non-trivial SI in Proposition 10.

6.5.1 Safety

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

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

Consider now that a transaction T_i read versions x_l and y_j of objects x and y during some execution of Jessy. The two objects where read sequentially; let us say x before y. According to the code of the execution module, there exists a process p replicating y such p executes lines 6 to 9 in Algorithm 1. It follows that $DV(w_l(x_l))[x] \geq DV(r_i(y_j))[x]$ holds. Since $DV(r_i(x_l))$ equals $DV(w_l(x_l))$, we know that $DV(r_i(x_l))[x] \geq DV(r_i(y_j))[x]$ holds. Similarly,

Algorithm 2 Termination Protocol of Jessy

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

we deduce that $DV(r_i(y_j))[y] \geq DV(r_i(x_l))[y]$ holds. Proposition 6 tells us that in such a case T_i has read a consistent snapshot.

It remains to show that that the histories generated by Jessy are write-conflict free. To this goal, we consider two independent write-conflicting transactions T_i and T_j, and we assume for the sake of contradiction that they both commit. We note p_i (resp. p_j) the coordinator of T_i (resp. T_j). Since T_i and T_j write-conflict, there exists some object x in $WS(T_i) \cap WS(T_j)$. One can show, using the preconditions of *doCertify*(), the monotonicity of variable *queue*, and the properties of atomic multicast, that (F1) for any two replicas p and q of x, denoting $committed_p$ (resp. $committed_q$) the set $\{T_j \in committed : x \in WS(T_j)\}$, at the time p (resp. q) executes $outcome(T_i)$, it is true that $committed_p$ equals $committed_q$. According to line 20 of Algorithm 2 and the

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

6.5.2 Liveness

Lemma 5. *For every transaction T_i, if T_i is submitted at coord(T_i) and coord(T_i) is correct, T_i eventually terminates at every correct process in WReplicas*(T_i).*

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

Because *queue* 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 ensures that no two process in the system wait for a vote from each other and none of them makes step. It follows that every correct replicas in *WReplicas**(T_i) eventually dequeue T_i, and send the outcome of function *certify*(T_i) to other replicas in *WReplicas**(T_i) (lines 16 to 17).

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

Lemma 6. *For every transaction T_i, if coord(T_i) executes T_i and coord(T_i) is correct, then eventually T_i is submitted to the termination protocol at coord(T_i).*

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

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

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

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

We consider two cases:

1. $DV(w_l(x_l))[x] < DV(r_i(y_j))[x]$ forever holds.
   According to the definition of function DV there exists a version $x_{k\neq 0}$ of object x written by some transaction T_k upon which transaction T_i depends, and such that $DV(w_k(x_k))[x] = DV(r_i(y_j))[x]$. Because transaction T_k committed at some site, Lemma 5 and Proposition 7 tell us that eventually T_k commits at process p. Contradiction.

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

□

Proposition 8. *Read-only transactions are wait-free.*

Proof. Consider some read-only transaction T_i and assume that $coord(T_i)$ is correct, Lemma 6 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 20 in Algorithm 2 is always true, whereas precondition at line 26 is always false. It follows that T_i eventually commits. □

Proposition 9. *Updates transactions are obstruction-free.*

Proof. Consider some update transaction T_i such that $coord(T_i)$ is correct. From the cunjonction of Lemmata 5 and 6, transaction T_i eventually terminates.

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

6.5.3 Non-trivial SI

To prove that Jessy implements non-trivial SI, we consider that when a process resolves a remote read request over some object x at lines 7 to 8, it always returns the greatest version of x (in the sense of function DV stored in variable db.

Proposition 10. *Consider an admissible history h containing a pending transaction T_i such that the next operation of T_i is a read over some object x. Note x_j the latest committed version of x in h. If history $h' = h.r_i(x_j)$ belongs to NMSI then history h' is admissible.*

Proof. Consider a replica p of x storing version x_j, and assume from now that process p always answers first to a remote read request from $coord(T_i)$ over x. Since history $h.r_i(x_j)$ is in NMSI, it belongs to CONS. As a consequence, T_i reads a consistent snapshot in $h.r_j(x_i)$. According to Proposition 6, it follows that both $DV(r_i(x_j))[x] \geq DV(r_i(y_k))[x]$ and $DV(r_i(x_j))[y] \leq DV(r_i(y_k))[y]$ hold. According to the preconditions of operation $readResolve(x, T_i)$, and the assumptions above, process p returns version x_j to $coord(T_i)$. □

7 Related Work

Schiper et al. [15] define quasi-genuine partial replication, i.e., processes that do not replicate objects accessed by a transaction may at most store transaction identifier. They propose a quasi-genuine replication algorithm based on atomic broadcast and a voting phase. The algorithm of Sousa et al. [16] is close to this approach. Both protocols assume that there is at least one replica that holds all the data accessed by a transaction. This assumption is unreasonable in many scenarios. The epidemic algorithm of Holliday et al. [17] supports serializability (SER) under genuine partial replication. However, it always aborts concurrent conflicting transactions, and blocks forever if a single process fails. G-Store [18] proposes transactional access to a distributed key value store, and relies on a dynamic key grouping protocol to partition data. At any time, a given object belongs to at most one group. A transaction may only access objects belonging to a single group. The ownership of all the keys inside a group is given to the leader of the group. This algorithm necessitates an *a priori* knowledge of the workload. All the mentioned protocols have some strong assumptions either on transactions, or on partitioned data. Hereafter, we review in details protocols not having such assumptions. Table 2 and 3 summarize our study.

We compare partial replication protocols in terms of time and message complexity when executing a transaction T_i with r_r remote reads, and w_r

Algorithm	Cons.	Genuine?	Multi-Master?
Scalaris [19]	SER	no	yes
P-Store [1]	SER	yes	yes
SIPRe[21]	SI	no	yes
Serrano[22]	SI	no	yes
Walter [23]	PSI	no	no
Jessy	NMSI	yes	yes

Table 2: Comparison of partial replication protocols

remote writes. Transaction T_i can be of the following three types: a read-only transaction, a local update transaction, i.e., when the coordinator replicates all the objects accessed by the transaction, and a global update transaction. We note Δ the message delay between replicas, N is the number of replicas in the system, and n denotes the number of replicas that replicate some object accessed by T_i. We note k the number of acceptors for Paxos Commit [19]. We consider that the latency of atomic broadcast (resp. atomic multicast) is 3Δ [20] (resp 4Δ [14]) during solo step execution. All latencies are computed from the coordinator's perspective.

Scalaris [19] is a transactional system on top of a DHT. Operations are performed on the majority of replicas, and paxos commit ensures both atomicity and SER. P-Store [1] is a genuine partial replication algorithm for WANs based on scalable genuine atomic multicast primitives. Read operations are optimistically performed at some replicas, while writes are executed locally. P-Store ensures SER.

A few algorithms [21, 22, 24] address partial replication with SI semantics. Sipre [21] atomically broadcasts a message to all replicas at the start of a transaction. The transaction uses this start message to observe a strictly consistent snapshot. SIPRe commits read-only transactions without further communication. To terminate an update transaction, SIPRe broadcasts the write set to all processes, then each process independently certify the transaction.

The system of Serrano et al. [22] supports partial replication and SI. In this solution, the first replica to receive some transaction T_i assigns a timestamp to T_i. This timestamp stores the latest committed values of objects read by T_i. Commit operations for transactions are broadcast atomically to all replicas. After a transaction commits, each replica starts some dummy transactions that are associated with database snapshots. Theorem 3 implies that none of these protocols is genuine.

More recently, Sovran et al. [23] proposed Walter, a transactional key-value store that supports a consistency criterion called parallel snapshot

Algorithm	Message complexity	Time complexity		
		Read-only	Global Update	Local Update
Scalaris [19]	$O(n \times k)$	$(r_r \times 2\Delta) + 4\Delta$	$(r_r + w_r) \times 2\Delta + 4\Delta$	$(r_r + w_r) \times 2\Delta + 4\Delta$
P-Store [1]	$O(n^2)$	$(r_r \times 2\Delta) + 3\Delta$	$(r_r \times 2\Delta) + 5\Delta$	4Δ
SIPRe[21]	$O(N^2)$	$(r_r \times 2\Delta) + 3\Delta$	$(r_r + w_r) \times 2\Delta + 6\Delta$	6Δ
Serrano[22]	$O(N^2)$	$r_r \times 2\Delta$	$(r_r + w_r) \times 2\Delta + 3\Delta$	3Δ
Walter [23]	$O(N \times w_r)$	$r_r \times 2\Delta$	$(r_r \times 2\Delta) + 2\Delta$	$2\Delta \mid 0$
Jessy	$O(n^2)$	$r_r \times 2\Delta$	$(r_r \times 2\Delta) + 5\Delta$	4Δ

Table 3: Complexity Comparison of partial replication protocols
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; k: number of acceptors in Paxos Commit.

isolation (PSI). PSI and NMSI were discovered independently, but have similarities. In particular, snapshots under PSI are non-monotonic (named *long fork* in [23]). Because it enforces SCONSa, PSI is however stronger than NMSI: a read operation over x in transaction T_i reads the most recent value of x at the time T_i starts at its site. To ensure PSI, Walter relies on a single master replication schema per object. If objects modified by a transaction have the same preferred site, the transaction commits locally, otherwise preferred sites start a two-phase commit protocol to commit the transaction. After the transaction commits, it is propagated to all sites in the system in the background before it becomes visible.

The latency of executing local update transactions in Walter [23] is the lowest among the surveyed protocols. When an update transaction is local, and the coordinator is the preferred site for all objects modified inside that transaction, the latency of update transaction is If the coordinator is not the preferred site for all objects, two phase commit protocol is employed among preferred sites, thus the latency is 2Δ. The latency of local updates in Serrano [22] is 3Δ because it uses one atomic broadcast for each local update transaction. This latency is 4Δ in case of Jessy or P-Store [1] since they employ genuine atomic multicast. SIPRe [21] and Scalaris [19] have the worst local update latency. SIPRe uses two atomic broadcast, and Scalaris cannot be optimized for performing local update transactions since it should read/write from/to a majority of replicas.

8 Conclusion

Partial replication and genuineness are two key factors of scalability in replicated systems. This paper shows that ensuring snapshot isolation (SI) in a genuine partial replication system is impossible. To state this impossibility

result, we introduce four necessary properties to attain SI. We show that two of them, namely Snapshot Monotonicity and Strictly Consistent Snapshots cannot be ensured.

To side step this impossibility result, we propose a novel consistency criterion named NMSI. NMSI prunes most anomalies disallowed by SI, while providing guarantees close to SI: transactions under NMSI are wait-free, always observe consistent snapshots, and two write-conflicting 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 commits transactions faster and/or contacts fewer replicas.

References

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

[2] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News*, vol. 33, pp. 51–59, June 2002.

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

[4] H. Berenson, P. Bernstein, J. Gray *et al.*, "A critique of ANSI SQL isolation levels," in *Management of Data - SIGMOD*, New York, New York, USA, 1995, pp. 1–10.

[5] 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.

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

[7] 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.

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

[9] P. A. Bernstein and N. Goodman, "An algorithm for concurrency control and recovery in replicated distributed databases," *ACM Transactions on Database Systems*, vol. 9, pp. 596–615, December 1984.

[10] H. Attiya, E. Hillel, and A. Milani, "Inherent limitations on disjoint-access parallel implementations of transactional memory," in *Symposium on Parallelism in Algorithms and Architectures*. New York, New York, USA: ACM Press, Aug. 2009, p. 69.

[11] 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.

[12] K. M. Chandy and L. Lamport, "Distributed snapshots: determining global states of distributed systems," *ACM Transactions on Computer Systems*, vol. 3, pp. 63–75, February 1985.

[13] A. Adya, B. Liskov, and P. O'Neil, "Generalized isolation level definitions," in *International Conference on Data Engineering*, Washington, DC, USA, 2000, pp. 67–78.

[14] 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.

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

[16] A. Sousa, R. Oliveira, F. Moura *et al.*, "Partial replication in the database state machine," in *Symposium on Network Computing and Applications*, Washington, DC, USA, 2001, pp. 298–309.

[17] 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.

[18] S. Das, D. Agrawal, and A. El Abbadi, "G-store: a scalable data store for transactional multi key access in the cloud," in *Symposium on Cloud Computing*, New York, NY, USA, 2010, pp. 163–174.

[19] F. Schintke, A. Reinefeld, S. Haridi *et al.*, "Enhanced Paxos Commit for Transactions on DHTs," in *Conference on Cluster, Cloud and Grid Computing*, Washington, DC, USA, May 2010, pp. 448–454.

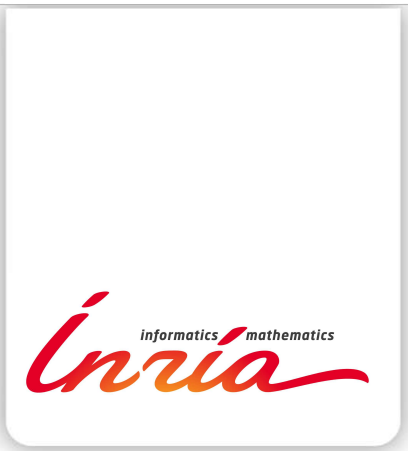
[20] A. Schiper, "Early consensus in an asynchronous system with a weak failure detector," *Distributed Computing*, vol. 10, pp. 149–157, April 1997.

[21] J. E. Armendáriz-Iñigo, A. Mauch-Goya, J. R. G. de Mendívil *et al.*, "SIPRe: a partial database replication protocol with SI replicas," in *Symposium on Applied computing*, New York, New York, USA, 2008, p. 2181.

[22] D. Serrano, M. Patino-Martinez, R. Jimenez-Peris *et al.*, "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.

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

[24] D. Serrano, M. Patiño-Martínez, R. Jiménez-Peris *et al.*, "An autonomic approach for replication of internet-based services," in *Symposium on Reliable Distributed Systems*, Washington, DC, USA, 2008, pp. 127–136.



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