Comparing Optimistic Database Replication Techniques *

Pierre Sutra Marc Shapiro
Université Paris VI and INRIA Rocquencourt, France
LIP6, 104, ave. du Président Kennedy, 75016 Paris, France
E-mail: pierre.sutra@lip6.fr, marc.shapiro@acm.org

Keywords: database replication, transaction processing, optimistic concurrency control, synchronous replication and atomic broadcast

Abstract

Replication is attractive for scaling databases up, as it does not require costly equipment and it enables fault tolerance. However, as the latency gap between local and remote accesses continues to widen, maintaining consistency between replicas remains a performance and complexity bottleneck. Optimistic replication (OR) addresses these problems. In OR, a database tentatively executes transactions against its local cache; databases reconcile *a posteriori* to agree on a common schedule of committed transactions. We present three OR protocols based on the deferred update scheme. The first two are representative of the state the art. The third is new; we describe it in detail. As all three protocols are expressed within a common formal framework, we are able to compare them, to identify similarities and differences, and to introduce common variants. We show that our protocol behaves better than the other two, with respect to latency, message cost and abort rate.

*This research is funded in part by the European project Grid4All and by the French project Respire.

1 Introduction

In order to scale up a database system, several approaches are possible: buying a bigger machine, dividing the work, or replicating the load across several remote machines. Replication does not require costly equipment and enables fault tolerance. However, remote access has a high latency, and the latency gap only keeps increasing. Furthermore, remote access is subject to disconnections. Therefore, maintaining consistency between replicas is difficult.

Optimistic replication (OR) is an attempt to address this problem. An OR system caches data. A database executes transactions against its local cache *tentatively*. Remote databases reconcile after the fact to agree on a common schedule of committed transactions.

Among OR techniques, the *deferred update* scheme has recently raised an increasing interest of the community [PGS03, HSAA03, KA00, SSP06]. In the deferred update scheme a database executes a new incoming transaction against its local cache. If the transaction is a query it is immediately committed; in the other case the database computes a logical clock, and the read set, write set and update values of the transaction. This information is then sent to distant sites to globally commit the transaction. The deferred update scheme has proven to

be efficient, outperforming existing pessimistic approaches, while maintaining consistency [WS05].

This paper compares three OR commitment protocols based on deferred updates. The first two are representative of the state of the art: the epidemic protocol of Agrawal et al. [AES97], and the Database State Machine (DBSM) approach of Pedone et al. [PGS03]. The third one is new. We describe it in detail.

Our contributions are the following:

- We describe all three protocols in the same formal framework. This clarifies the comparison.
- Using the framework, we can explore common variants. For instance, we propose a new variant to the Agrawal et al. protocol, ensuring snapshot isolation.
- We propose a new OR protocol that batches transactions. This allows it to optimise the abort rate. Furthermore, batching amortizes communication and computation costs.
- We show that our protocol improves over the other two, in terms of latency, message cost and abort rate.
- We propose two variants of our protocol: one that is more optimistic, and one that ensures snapshot isolation.

The rest of this paper is organized as follows. We present our model in Section 2. Section 3 studies the Agrawal et al. protocol, and proposes a variant for snapshot isolation. Section 4 studies the protocol of Pedone et al. Section 5 presents our proposal and its variants. We compare the different protocols in Section 6. Section 7 surveys related work. We close in Section 8 with a discussion.

2 System Model

We consider a distributed system in which any client can submit an operation on shared data, at any site at any time. In the general case, maintaining consistency requires a complex concurrency control mechanism. However, providing the system with some semantic knowledge can simplify consistency. For instance, suppose that all updates commute; in this case, maintaining consistency reduces to propagating the update operations to all sites, and executing them in any order.

Building upon this insight, our model (which is a refinement of the Action-Constraint Framework [SBK04]) maintains an explicit graph, where the nodes are the actions that access shared data, and the edges represent semantic links between actions. A consistency protocol is a particular solution to a graph problem. The complexity of the problem is related to the shape of the graph. In our experience, our model clarifies the understanding of consistency, makes it easier to compare protocols, and helps with the design of new solutions.

2.1 The Action-Constraint Framework

2.1.1 Actions, constraints and multilogs

We postulate a universal set of *actions* $\mathbb{A}$.

Actions are linked each others by *constraints*, which are relations over $\mathbb{A}$. Five constraints are of particular interest in our framework: $+$, $-$, $\rightarrow$, $\lhd$ and $\between$; respectively pronounced "commit," "abort," "not after," "enables" and "non-commuting." The constraints $+$ and $-$ are unary relations over $\mathbb{A}$, whereas $\rightarrow$, $\lhd$ and $\between$ are binary relations over $\mathbb{A}$. Their semantics will be explained shortly, in Section 2.1.2.

Our central structure in the Action-Constraint Framework (ACF) is the *multilog*. A multilog is a sextuple $(K,+,-,\rightarrow,\lhd,\between)$ where K is a set of actions, and $+$, $-$, $\rightarrow$, $\lhd$, $\between$ are some sets of constraints over $\mathbb{A}$. We note $\mathbb{M}$ the universal set of multilogs over $\mathbb{A}$.

We define union, intersection, difference, etc., between multilogs as component-wise operations. For instance, let $M=(K,+,-,\rightarrow,\lhd,\between)$ and $M'=(K',+',-',\rightarrow',\lhd',\between')$ be two multilogs. Then, $M\cup M'=(K\cup K',+\cup+',-\cup-',\rightarrow\cup\rightarrow',\lhd\cup\lhd',\between\cup\between')$. By abuse of notation, we also use the union operator to add an element to a single component, which should be clear from the context. For instance, $M\cup\{\alpha\}$ adds α to the K component, i.e., $M\cup\{\alpha\}\stackrel{\triangle}{=}(K\cup\{\alpha\},+,-,\rightarrow,\lhd,\between)$. Similarly, $M\cup\{\alpha^+\}$ adds (α,α) to the $+$ component, and $M\cup\{\alpha\rightarrow\beta\}$ adds the pair (α,β) to the $\rightarrow$ component.

The notation $\alpha^-\in M$, or just (when clear from the context) α^-, are used as a shorthand for "(α,α) is in the $+$ component of M." Similarly, either $\{\alpha\rightarrow\beta\}\in M$ or just $\alpha\rightarrow\beta$ are shorthands for "The pair (α,β) is in the $\rightarrow$ component of M."

We also use the following shorthand notations:

$\alpha\leftrightarrows\beta\stackrel{\triangle}{=}\alpha\rightarrow\beta\wedge\beta\rightarrow\alpha$

$\alpha\stackrel{\lhd}{\rightarrow}\beta\stackrel{\triangle}{=}\alpha\rightarrow\beta\wedge\alpha\lhd\beta$

$\alpha\stackrel{\lhd}{\triangleright}\beta\stackrel{\triangle}{=}\alpha\lhd\beta\wedge\beta\lhd\alpha$

2.1.2 Schedules of multilogs and classes of schedules

Let E be a subset of $\mathbb{A}$. We call schedule a couple $S=(E,<_S)$ where $<_S$ is a strict total order over E. We note $\mathbb{S}$ the universal set of schedules over $\mathbb{A}$.

Given a multilog $M=(K,+,-,\rightarrow,\lhd,\between)$, we say that $S=(E,<_S)$ is a schedule of M, iff :

$$\begin{array}{l}\forall\alpha,\beta\in K\\ \alpha^-\in M\Rightarrow\alpha\notin E\\ \alpha^+\in M\Rightarrow\alpha\in E\\ \alpha\lhd\beta\in M\Rightarrow(\beta\in E\Rightarrow\alpha\in E)\\ \alpha\rightarrow\beta\in M\Rightarrow(\alpha,\beta\in E\Rightarrow\alpha<_s\beta)\end{array}$$

For some action α and a schedule $S=(E,<_S)$, we say that α is *scheduled* in S (noted $\alpha\in S$) iff $\alpha\in E$. We note $\Sigma(M)$ the set of schedules of M.

Thus the $-$, $+$, $\rightarrow$ and $\lhd$ constraints restrict which schedules may appear in $\Sigma(M)$. This defines their semantics.

In contrast, $\between$ divides $\Sigma(M)$ into equivalence classes of schedules. Let $M=(K,+,-,\rightarrow,\lhd,\between)$ be a multilog. Two schedules S and S' of $\Sigma(M)$ are said *equivalent* according to $\between$, noted $S\sim S'$, iff:

$$\forall\alpha,\beta\in K,\ \begin{cases}\alpha\in S\Leftrightarrow\alpha\in S'\\ (\alpha,\beta)\in S^2\wedge\alpha\between\beta\in M\Rightarrow(\alpha<_S\beta\Leftrightarrow\alpha<_{S'}\beta)\end{cases}$$

We note $\Sigma(M)/\!\sim$ the quotient set of $\Sigma(M)$ by $\sim$, and $|\Sigma(M)/\!\sim|$ the number of equivalence classes of schedules induced by $\sim$.

The following constraints or combinations of constraints are particularly useful for defining application semantics. Let $M=(K,+,-,\rightarrow,\lhd,\between)$ be a multilog and α,β two actions of K. A $\rightarrow$-cycle in M (e.g., $\alpha\leftrightarrows\beta$) represents *antagonism*, i.e for any schedule S of $\Sigma(M)$, either α is in S, or β is in S, or neither of them; the conjunction $\beta\stackrel{\lhd}{\rightarrow}\alpha$ means α *depends causally* on β; and an $\lhd$-cycle such as $\alpha\stackrel{\lhd}{\triangleright}\beta$ expresses an atomic grouping. Finally $\alpha\between\beta$ means that α and β do not commute; if α and β are transactions, this models the isolation constraint (the I of ACID).

2.1.3 Particular subsets of multilogs and concept of soundness

Let $M = (K, +, -, \rightarrow, \lhd, \nprec)$ be a multilog. The following subsets of K are of particular interest for the study of consistency.

- *Committed* actions appear in every schedule of M. This set is the greatest subset of K satisfying:
  $Committed(M) \stackrel{\triangle}{=}$
  $\{\alpha \,|\, \alpha^+ \vee \exists \beta \in Committed(M) : \alpha \lhd \beta\}$
- *Aborted* actions never appear in a schedule of M. $Aborted(M)$ is the greatest subset of K that satisfies:
  $Aborted(M) \stackrel{\triangle}{=}$
  $$\{\alpha |\ (\exists \beta_1, \ldots, \beta_{m \geq 0} \in Committed(M), \alpha \rightarrow \beta_1 \rightarrow \ldots \rightarrow \beta_m \rightarrow \alpha) \vee \exists \beta \in Aborted(M) : \beta \lhd \alpha \vee \alpha^-\}$$
- *Serialized* actions are either aborted, or are ordered with respect to all non-commuting constraints against non-aborted actions:
  $Serialized(M) \stackrel{\triangle}{=}$
  $$\{\alpha |\ \forall \alpha, \beta \in K, \alpha \nprec \beta \Rightarrow (\alpha \rightarrow \beta \vee \beta \rightarrow \alpha \vee \alpha \in Aborted(M) \vee \beta \in Aborted(M))\}$$
- *Decided* actions are either aborted, or both committed and serialized:
  $Decided(M) \stackrel{\triangle}{=}$
  $Aborted(M) \cup (Committed(M) \cap Serialized(M))$
- A *Durable* action is decided, and, if committed, all actions that precede it, either by $\rightarrow$ or by $\lhd$, are themselves durable. This is the greatest subset of K satisfying :
  $$Durable(M) \stackrel{\triangle}{=} Aborted(M) \cup \{\alpha \in Committed(M) \mid \forall \beta \in K : (\beta \rightarrow \alpha \vee \beta \lhd \alpha) \Rightarrow \beta \in Durable(M)\}$$

A multilog M is said *sound* iff $Committed(M) \cap Aborted(M) = \emptyset$. Observe that $\Sigma(M) \neq \emptyset$ implies M sound.

A multilog M is said *decided* iff $Decided(M) = K$.

A multilog M is said *durable* iff $Durable(M) = K$, or equivalently iff M is sound and $|\Sigma(M)/\sim| = 1$.

2.2 Formalizing consistency in replicated systems

We consider an asynchronous distributed system of n sites $i, j, \ldots$, connected through fair-lossy links [BCBT96]. The failure model is fail-stop. A global clock $t \in \mathcal{T}$ ticks at every step of any process, but processes do not have access to it.

We assume that some shared data is replicated at every site. Initially, at $t = 0$, the data is in the same state at every site. We make no further assumption about the data; indeed data does not appear explicitly in the model, which considers only the actions that access the data.

A site contains two processes: an application process called the *client*, and a single *consistency agent* (or just *agent* hereafter).

Clients receive and execute user actions accessing shared data. Agents ensure the consistency of the system by executing a protocol.

ACF constraints capture both the schedule semantics of actions, and the decisions taken by the protocol.

2.2.1 Site-multilogs and site-schedules

At any point in time t, each site i is entirely defined by its *site-multilog* $M_i(t) = (K_i(t), +_i(t), -_i(t), \rightarrow_i (t)), \lhd_i (t), \nVdash_i (t)$ and its *site-schedule* $S_i(t)$.

- $M_i(t)$ is the local knowledge that i has at time t of the set of actions and of the semantics linking them.

  Initially, every site-multilog is equal to $(\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$.

  Site-multilogs grow monotonically over time, as clients add actions and constraints, and agents add constraints. The following rule captures this monotonic growth:

$$\forall i \in [\![1,n]\!], \forall t \in \mathcal{T}, \exists M \in \mathbb{M}, \\ M_i(t+1) = M_i(t) \cup M$$

  We abstract the computation of constraints into a routine noted *addConstraints*() that takes as input a multilog $M = (K, +, -, \rightarrow, \lhd, \nVdash)$, and returns a multilog $M' = (K', +', -', \rightarrow', \lhd', \nVdash')$ such that: $M \subseteq M'$ and $K' = K$. Different concurrency control differ, in particular, in how they compute *addConstraints*().

- $S_i(t) \in \Sigma(M_i(t))$ represents the state of shared data on i at time t. The choice of $S_i(t)$ is arbitrary when $|\Sigma(M_i(t))/\sim| > 1$. If $S_i(t-1)$ is not a prefix of $S_i(t)$, it represents a roll-back.

Agents and clients both have access to the site-schedule and the site-multilog, but our clock is assumed sufficiently fine-grain that between t and $t+1$, only one or the other may access it. We formalise this using transitions: $(M_i(t), S_i(t)) \rightsquigarrow_A (M_i(t+1), S_i(t+1))$ for the agent, and $(M_i(t), S_i(t)) \rightsquigarrow_C (M_i(t+1), S_i(t+1))$ for the client.

2.2.2 Definition of System and of Commitment Protocol

We note a system of n sites as $\mathcal{S}_n = ((M_1, S_1), \ldots, (M_n, S_n))$.

We call *protocol* a family of algorithms $\mathcal{P} = \{\mathcal{P}_1, \mathcal{P}_2 \ldots\}$ where each $\mathcal{P}_i$ is defined by a set of couples $(S, T) \in (\mathbb{M}x\mathbb{S})^2$, where S is a state and T a transition.

In our framework both clients and agents execute protocols. Given a system $\mathcal{S}_n$, we note $C = \{C_1, \ldots, C_n\}$, (resp. $A = \{A_1, \ldots, A_n\}$) the protocol of clients (resp. agents) executing at sites $1 \ldots n$.

The client protocol is left mostly unspecified, as clients are free to do anything, as long as they do not put the system into an error state. The agent protocol aims to bring the system to consistency; we refer to A as a *commitment* protocol. Hereafter, we study three different commitment protocols, and variants of each.

2.2.3 Runs

A run r of $\mathcal{S}_n$ according to C and A is an array of n rows, each row i representing the evolution over time of (M_i, S_i) starting at $t = 0$, and such that :

$$\forall i \in [\![1,n]\!], \forall t \in \mathcal{T}, \exists M \in \mathbb{M}, \\ M_i(t+1) = M_i(t) \cup M \\ \wedge \begin{cases} (M_i(t), S_i(t)) \rightsquigarrow_C (M_i(t+1), S_i(t+1)) \\ \Rightarrow ((M_i(t), S_i(t)), (M, S_i(t+1))) \in C_i \end{cases} \\ \wedge \begin{cases} (M_i(t), S_i(t)) \rightsquigarrow_A (M_i(t+1), S_i(t+1)) \\ \Rightarrow ((M_i(t), S_i(t)), (M, S_i(t+1))) \in A_i \end{cases}$$

As usual, considering a run r, we say that a site i is *correct* in r iff $r[i]$ is infinite (noted $i \in correct(r)$); otherwise we say that i is *crashed* in r ($i \in crash(r)$).

A column of r at time t represents the state of the system at time t. We note it $\mathcal{S}_n(t)$.

We note $R(\mathcal{S}_n, C, A)$ *the set of runs of A and C in* $\mathcal{S}_n$.

2.3 Consistency

Serialization theory [BHG87] considers only finite sets of transactions; accordingly, hereafter we consider only quiescent systems. Given a system $\mathcal{S}_n$, a set of clients C, and a set of agents A, we say that a system $\mathcal{S}_n$ is *quiescent* iff in any run of $\mathcal{S}_n$, both agents and clients eventually stop submitting new actions:

$$\begin{array}{l}\forall r \in R(\mathcal{S}_n, C, A),\\ \exists T \in \mathcal{T}, \forall t \geq T \in \mathcal{T}, \forall i \in [\![1,n]\!],\\ M_i(t+1).K = M_i(t).K\end{array}$$

where $M.K$ denotes the K component of multilog M.
Definition (Eventual Consistency)**.** *A system* $\mathcal{S}_n$ *is eventually consistent (EC) in a run r iff it satisfies the following correctness conditions:*

- Eventual Decision:

$$\begin{array}{r}\forall i \in \mathit{correct}(r), \forall t \in \mathcal{T}, \forall \alpha \in K_i(t),\\ \exists t' \in \mathcal{T}, \alpha \in \mathit{Decided}(M_i(t'))\end{array}$$

- Mergeability:

$$\Sigma(\bigcup_{\substack{i \in [\![1,n]\!]\\ t \in \mathcal{T}}} M_i(t)) \neq \emptyset$$

- Eventual Agreement:

$$\begin{array}{r}\exists t \in \mathcal{T}, \forall t' \geq t, \forall i, j \in \mathit{correct}(r),\\ S_i(t') \sim S_j(t')\end{array}$$

Roughly speaking, eventual decision ensures that the system makes progress. Eventual agreement ensures that all sites eventually agree on the decisions. Mergeability ensures that the system is globally sound, i.e., no decision ever puts it in an error state.

If in any run of $R(\mathcal{S}_n, C, A)$, with at most f crashes, eventual consistency is attained by every correct process, we say that A is f-resilient.

Given a system $\mathcal{S}_n$ a client C, and a fault-resilience degree f we call the problem of finding such a protocol A, the *consistency problem*.

2.4 Modeling database replication

This section refines the previous model to the specific case of a fully replicated database accessed through ACID transactions.

We model ACID transactions in our framework at a coarse-grained level, where a single action represents a whole transaction. Given a transaction T, we note $RS(T)$ its read set, $WS(T)$ its write set and $UV(T)$ the corresponding update values.

Two transactions may be related by constraints derived from their respective read and write sets, and from whether they are concurrent or not. Commitment protocols differ on how they compute these constraints, as will become apparent later.

We model a set of fully replicated databases as a system of sites. A processes that issues transactions is a client, and agents execute the protocol.

The client, Algorithm 1, models the application processes. A client submits a new transaction at a time to its local replica, by adding it to the site-multilog.

We divide an agent into three modules that execute in parallel:

- The *execution* module schedules and executes transactions.

Algorithm 1 Client C_i at site i

1: M_i {the site-multilog of i}
2: S_i {the site-schedule of i}
3: **loop**
4: choose some transaction T
5: $M_i := M_i \cup \{T\}$
6: **end loop**

- The *certification* module decides which transactions to abort or commit.
- The *propagation* module sends and receives messages to co-ordinate replicas.

All the commitment protocols considered in this paper are based on a scheme known as *deferred execution*. A transaction first executes at the local site, under local serializability. The system records the transaction's read set, write set and update values. At this point, no remote locks are taken. After the transaction terminates the system contacts remote sites, attempts to apply the update values to the write set remotely, and to certify the transaction. The transaction may commit only if the certification succeeds.

More formally, let i be a site, and T a transaction submitted at i. The deferred execution algorithm is as follows.

1. i executes T under two phase-locking (2PL) [BHG87].[1]
2. When T terminates without aborting, it keeps its write locks and releases its read locks.
3. Site i computes $RS(T)$, $WS(T)$ and $UV(T)$, and assigns a vector clock value to T (see hereafter).
4. If T is a read-only transaction, it commits.
5. Otherwise, $WS(T)$, $UV(T)$ and T's vector clock are sent to sites $j \neq i$.
6. When site j receives T, it examines $WS(T)$, $UV(T)$ and T's vector clock, and either aborts or commits T according to a specific certification algorithm. If it commits, it applies $UV(T)$ to $WS(T)$.

In the rest of this paper, we discuss the differences between commitment protocols, in particular different certification algorithms.

We model bullets 1–4 with the execution module. Bullet 5 constitutes the propagation module. Bullet 6 constitutes the certification module.

2.4.1 The execution module

Algorithm 2 shows in more detail how, given the current site-schedule S_i, the execution module computes a new schedule S.

Algorithm 2 Execution module in the deferred scheme

1: M_i {the site-multilog of i}
2: S_i {the site-schedule of i}
3: **loop** {execution}
4: choose $S \in \Sigma(M_i)$ such that $\forall T, T' \in K$,
5: $T \in S_i \Rightarrow T \in S \vee T \in Aborted(M_i)$
6: $(T, T' \in S \wedge T >_{S_i} T') \Rightarrow T >_S T'$
7: $\left.\begin{array}{l}(T \in S_i \wedge\ T' \notin S_i \wedge T' >_S T) \\ \wedge WS(T) \cap (RS(T') \cup WS(T')) = \emptyset\end{array}\right\} \Rightarrow T \in Committed(M_i)$
8: **for all** $T, WS(T) = \emptyset$ **do**
9: $M_i := M_i \cup T^+$
10: **end for**
11: $S_i := S$
12: **end loop**

Two-phase locking ensures that any new schedule extends the current schedule. Consequently a transaction never rolls back unless it is aborted (Line 5), and transactions execute in the same order (Line 6). When a transaction terminates, it releases its read locks but keeps its write locks; therefore any new transaction can execute only if all the

[1] With no loss of generality, we can ignore local deadlocks.

transactions with which it conflicts are already committed (Line 7). Finally read-only transactions commit when they terminate (Line 9).

2.4.2 The propagation module

The propagation module differs between commitment protocols. In particular they are based on different communication primitives. Consider some message m.

Epidemic Propagation consists of two primitives: $EPsend(m)$ and $EPreceive(m)$. With epidemic propagation, processes have the following guarantees:

- Integrity: if j performs $EPreceive(m)$, then a process i performed $EPsend(m)$ previously.
- If a correct process i performs $EPsend(m)$ infinitely often, and j is correct and performs $EPreceive()$ infinitely often, then j eventually performs $EPreceive(m)$.

Atomic broadcast consists of the primitives $ABcast(m)$ and $ABdeliver(m)$, with the following properties:

- Uniform Integrity: for every message m every process performs $ABdeliver(m)$ at most once and only if a process i performed $ABcast(m)$ previously.
- Validity: if a correct process i performs $ABcast(m)$, then it eventually performs $ABdeliver(m)$.
- Agreement : if a correct process i performs $ABdeliver(m)$, then every other correct processes eventually perform $ABdeliver(m)$.
- Uniform Total order: if a process performs $ABdeliver(m)$ and $ABdeliver(m')$ in this order, then every process that performs $ABdeliver(m')$ has previously performed $ABdeliver(m)$.

	$T \prec T'$	$T \parallel T'$	$T' \prec T$
$RS(T) \cap WS(T') \neq \emptyset$	$T \rightarrow T'$	$T \rightarrow T'$	$T' \stackrel{\lhd}{\rightarrow} T$
$WS(T) \cap WS(T') \neq \emptyset$	$T \rightarrow T'$	$T \nvdash T'$	$T' \rightarrow T$

Table 1: An example of constraints computation

2.4.3 The certification module

The certification modules differs from one commitment protocol to another, but they all base their certification on static constraints computed using:

1. Read-set and write-set intersection. Two transactions T and T' are said to *conflict* iff $(RS(T) \cap WS(T') \neq \emptyset) \vee (RS(T') \cap WS(T) \neq \emptyset) \vee (WS(T) \cap WS(T') \neq \emptyset)$.
2. The *happens-before* relation [Lam78]. Transaction T happens-before T', noted $T \prec T'$, iff T' is submitted at some site i after T has terminated at site i, or if there exists a transaction T'' such that $T \prec T'' \wedge T'' \prec T'$. If neither $T \prec T'$ nor $T' \prec T$ the two transactions are said concurrent, noted $T \parallel T'$.

As mentioned previously, commitment protocols enforce constraints computed by *addConstraints*(). These depend on the consistency criterion that needs to be ensured. Table 1 provides an example of such a computation.

For instance given two transactions T and T' such that $T \parallel T'$, $RS(T) = \{x\}$,$WS(T) = \{y\}$,$RS(T) = \emptyset$, and $WS(T') = \{x,y\}$, Table 1 defines the constraints between T and T' as: $T \rightarrow T'$ and $T \nvdash T'$.

If we consider an empty multilog M, then the result of $addConstraints(M \cup \{T,T'\})$ is the multilog M' such that: $M' = (\{T,T'\},\emptyset,\emptyset,\{(T,T')\},\emptyset,\{(T,T')\})$.

	$T' \in snapshot(T)$	$Max_{X \in snapshot(T)}\{V(X)\} < Max_{Y \in snapshot(T')}\{V(Y)\}$
$RS(T) \cap WS(T') \neq \emptyset$	$T' \overset{\lhd}{\rightarrow} T$	$\emptyset$
$WS(T) \cap WS(T') \neq \emptyset$	$\emptyset$	$T \overset{\leftarrow}{\rightarrow} T'$

Table 2: Constraint computation for GSI

	$T' \in workflow(T)$
$RS(T) \cap WS(T') \neq \emptyset$	$T' \rightarrow T$

Table 3: Constraint computation for PCSI

2.4.4 Serializability and Snapshot Isolation

This paper considers two consistency criteria, Serializability and Snapshot Isolation. Serializability (SER) means that the multiversion serialization graph of committed transactions is acyclic [BHG87]. Snapshot Isolation (SI) is weaker, ensuring that read-only transactions never block and do not cause update transactions to abort.

We also consider Generalized Snapshot Isolation (GSI), whereby a transaction always observes a consistent state of the database, but not necessarily the latest one [EZP05], and Prefix Consistent Snapshot Isolation (PCSI), in which a transactions observes at least the effects of transactions that precede it in the same "workflow."

SI is used in many commercial databases, such as Oracle [Ora97], PostGres [Glo04] and SQLServer [Mic05]. In practice, most computations are serializable under SI [FLO$^+$05].

We introduce SI and friends into our framework as follows. Let T be a transaction. We note $snapshot(T)$ the set of transactions that T reads from its snapshot ($snapshot(T)$ is any subset of $\{T'|T' \prec T\}$). Similarly the workflow of a transaction t is some subset of $\{T'|T' \prec T\}$, noted $workflow(T)$.

SER, GSI and PCSI are mapped to Eventual Consistency: if during a run, the system is eventually consistent, and the constraints linking transactions are *sufficient*, then the system reaches the consistency criterion.

For instance, if the system is eventually consistent in a run r and constraints are computed according to Table 1, then the execution r is serializable; if constraints are computed according to Table 2, r is GSI; and if constraints are computed according to Table 2 in addition to Table 3, r is PCSI.

3 Database replication with epidemic propagation

Agrawal, El Abbadi and Steinke propose a family of commitment protocols based on an epidemic communication between sites [AES97]. We first model their pessimistic scheme (AES), then consider their optimistic variant (AESO).

3.1 Overview

AES uses a deferred scheme in which sites exchange epidemically their local logs. AES ensures serializability with a certification test, which ensures that any two concurrent transactions that conflict will both abort. When a site i receives a log containing a transaction T, if T is not aborted, its write locks are taken on site i, and update values of T are applied. When i learns that T was successfully executed at all sites, T is committed at i.

	$T \prec T'$	$T \parallel T'$	$T' \prec T$
$RS(T) \cap WS(T') \neq \emptyset$	$T \rightarrow T'$	$T \leftrightarrows T'$	$T' \triangleleft T$
$WS(T) \cap WS(T') \neq \emptyset$	$T \rightarrow T'$	$T \leftrightarrows T'$	$T' \rightarrow T$
true	$T \rightarrow T'$	$\emptyset$	$T' \rightarrow T$

Table 4: Constraints computation for AES

3.2 Computing constraints in AES

Table 4 summarises the constraints used in AES. In AES transactions are executed according in the order they appear in logs, whether they commute or not, hence the $\rightarrow$ relation in the bottom row of the table. In AES concurrent conflicting transactions cannot be both executed; we translate this with an antagonism (first two rows). Then, as transactions are executed under 2PL:

- If a committed write happens-before a read with which it conflicts, the write is causally before the read:

$$\left.\begin{array}{l} T' \prec T \\ \wedge\ RS(T) \cap WS(T') \neq \emptyset \end{array}\right\} \Rightarrow T' \triangleleft T$$

- If a committed read or write happens-before a write, the former is ordered before the write.

$$\left.\begin{array}{l} T' \prec T \\ \wedge\ WS(T) \cap (RS(T') \cup WS(T')) \neq \emptyset \end{array}\right\} \Rightarrow T' \rightarrow T$$

3.3 AES

Algorithm 3 expresses the AES algorithm. (Each loop iteration is atomic.)

Algorithm 3 AES, code for site i

```
 1: M_i {the site-multilog of i}
 2: S_i {the site-schedule of i}
 3: Logs[n] {an array of n multilogs. Logs[i] = M_i}
 4: loop {execution}
 5:     same as Algorithm 2
 6: end loop
 7: ||
 8: loop {propagation}
 9:     let L = {T ∈ S_i | WS(T) ≠ ∅}
10:     choose k ∈ [[1, n]]
11:     L := {T ∈ L | ∀T' ∈ Logs[k], T → T' ∉ Logs[k]}
12:     EPsend(L) to j
13: end loop
14: ||
15: loop {commitment}
16:     EPreceive(L) from some process j
17:     Logs[j] := addConstraints(Logs[j] ∪ L)
18:     M_i := addConstraints(M_i ∪ L)
19:     for all T ∈ L do
20:         if ∃T' ∈ M_i : T ⇆ T' ∈ M_i then
21:             M_i := M_i ∪ {T⁻, T'⁻}
22:         else
23:             if T ∉ Aborted(M_i) ∧ (∀k ∈ [[1, n]], ∃T' ∈
                Logs[k], T → T') then
24:                 M_i := M_i ∪ {T⁺}
25:             end if
26:         end if
27:     end for
28: end loop
```

3.4 Correctness of our translation and observations

Concurrency control in AES is based on the predicate $HasRecvd(i,T,k)$. This predicate captures the fact that site i knows that k has received transaction T. We capture this information with an array of n multilogs, $Logs$. $Logs[k]$ contains the knowledge that i has of site k: $HasRecvd(i,T,k) \stackrel{\triangle}{=} \exists T' \in Logs[k], \{T \rightarrow T'\} \in Logs[k]$.

A non-query transactions that has executed lo-

cally, and has not yet been received by some remote site, is sent to that site. Formally, transaction T is sent to remote site k if $\neg HasRecvd(i,T,k) \Leftrightarrow (\forall T' \in Logs[k], T \rightarrow T' \notin Logs[k])$ (Line 11). Observe that this propagation scheme might block when clients stop submitting new transactions to the system.

Using *HasRecvd*, AES aborts T and T' if both of them have executed on at least one site, and if they are conflicting and concurrent [AES97]. See Lines 20 to 21.

AES defines the predicate $Commit(T,i)$ such that i commits T if i knows that T has been received by every site and no concurrent conflicting transactions exist (Lines 23 and 24).

3.5 The optimistic variant

In AES, an optimistic variant of AES, transactions release their write locks at the end of execution. With this modification, cascading aborts may occur, and read-only transactions (queries) may read uncommitted values.

Our model for AESO is almost identical to AES. Indeed $\lhd$ captures the existing abort dependencies between transactions: if $T \lhd T'$, then $T' \in Committed(M_i) \Rightarrow T \in Committed(M_i)$ and conversely (ii) $T \in Aborted(M_i(t)) \Rightarrow T' \in Aborted(M_i(t))$. Releasing write locks the end of execution translates to removing Line 7 from Algorithm 2. The rest is unchanged, see Algorithm 4.

3.6 The Snapshot Isolation variant

As an illustration of our framework, we propose a variant of AES that ensures GSI or PCSI.

For GSI, the change is very simple: in either AES or AESO, replace Table 4 with Table 2 . To obtain PCSI, add Table 3 to Table 2.

Algorithm 4 AESO, code for site i

```
 1: M_i {the site-multilog of i}
 2: S_i {the site-schedule of i}
 3: Logs[n] {an array of n multilogs. Logs[i] = M_i}
 4: loop {execution}
 5:     choose S ∈ Σ(M_i) such that ∀T,T' ∈ K,
 6:     T ∈ S_i ⇒ T ∈ S ∨ T ∈ Aborted(M_i)
 7:     (T,T' ∈ S ∧ T >_{S_i} T') ⇒ T >_S T'
 8:     S_i := S
 9: end loop
10: ||
11: loop {propagation}
12:     same as Algorithm 3
13: end loop
14: ||
15: loop {commitment}
16:     same as Algorithm 3
17: end loop
```

4 The Database State Machine Approach

The database state machine approach [PGS03] uses a deferred scheme where the certification test is based on atomic broadcast. Two approaches exist: (1) A classical approach (DBSM) in which an update transaction commits or aborts as soon as it is delivered to sites; and (2) a reordering technique (DBSMR) in which a delivered transaction is re-ordered with relation to the set of already committed transactions.

4.1 Static constraints in DBSM

In DBSM, all update transactions are ordered. Consequently any pair of transactions with a non-null write-set is considered non-commuting.

Now let us refine these constraints according to the certification test. Let $Committed(j)$ be the set of committed transactions at site j. The certification

	$T \prec T'$	$T \parallel T'$	$T' \prec T$
$RS(T) \cap WS(T') \neq \emptyset$	$T \rightarrow T'$	$T \rightarrow T'$	$T' \overset{\lhd}{\rightarrow} T$
$WS(T) \neq \emptyset \wedge WS(T') \neq \emptyset$	$T \rightarrow T'$	$T \nparallel T'$	$T' \rightarrow T$

Table 5: Constraints in DBSM

test commit transaction T at j after it has been delivered, iff:

$$\forall T' \in Committed(j), \quad T' \prec T \vee WS(T') \cap RS(T) = \emptyset$$

Consequently T is aborted iff:

$$\exists T' \in Committed(j), \quad T' \parallel T \vee T \prec T' \wedge WS(T') \cap RS(T) \neq \emptyset$$

This test ensures that if both T and T' are committed, then $T \overset{\leftarrow}{\rightarrow} T'$ cannot occur. Consequently since T is executed after T', this certification test checks that $T \rightarrow T'$ is not an existing constraint between T and T'. It follows that:

$$\left.\begin{array}{c} T' \parallel T \wedge WS(T') \cap RS(T) \neq \emptyset \\ \vee\ (T \prec T' \wedge WS(T') \cap RS(T) \neq \emptyset) \end{array}\right\} \Rightarrow T \rightarrow T'$$

Moreover, since transactions execute in DBSM with 2PL, observations appearing in Section 3.2 hold. Table 5 sums up the constraints.

For the DBSMR reordering technique [PGS03, page 11] a similar reasoning leads to the same table.

4.2 DBSM

Algorithm 5 presents the translation of the classical database state machine approach in our framework.

Algorithm 5 DBSM, code for site i

```
 1: M_i {the site-multilog of i}
 2: S_i {the site-schedule of i}
 3: loop {execution}
 4:     same as Algorithm 2
 5: end loop
 6: ||
 7: loop {propagation}
 8:     choose a T ∈ M_i s.t. T ∈ S_i ∧ T ∉ Committed(M_i)
 9:     AB-cast(T)
10: end loop
11: ||
12: loop {Commitment}
13:     AB-deliver(T)
14:     M_i := addConstraints(M_i ∪ T)
15:     for all T' ∈ Committed(M_i) do
16:         M_i := M_i ∪ {T' → T}
17:     end for
18:     if T ∉ Aborted(M_i) then
19:         M_i := M_i ∪ {T+}
20:     end if
21: end loop
```

4.3 DBSMR

In its classical form DBSM leads to a high abort rate due to the unnecessary order appearing at line 16. To solve this problem Pedone et al. propose a reordering technique based on the deterministic construction of a partial order over certified transactions: Algorithm 6.

However this approach has a drawback : when the system becomes quiescent, transactions block in buffer B. To preserve liveness, "null" transactions have to be sent to sites.

4.4 Snapshot Isolation

Elnikety et al. depict a variant of DBSM to guarantee Generalized Snapshot Isolation [EZP05].

Algorithm 6 DBSMR, code for site i

1: M_i {the site-multilog of i}
2: S_i {the site-schedule of i}
3: B {a buffer of size l}
4: **loop** {execution}
5: same as Algorithm 2
6: **end loop**
7: ||
8: **loop** {propagation}
9: same as Algorithm 5.
10: **end loop**
11: ||
12: **loop** {commitment}
13: AB-deliver(T)
14: $M_i := addConstraints(M_i \cup T)$
15: **if**

$$\forall T' \in Committed(M_i), \{T \to T'\} \notin M_i \\ \wedge \left\{ \begin{array}{l} \exists j \in [\![0, l-1]\!], \\ \forall k \in [\![0, j-1]\!], \{B[k] \to T\} \notin M_i \\ \wedge\ (\forall k \in [\![j, l-1[\![, \{T \to B[k]\} \notin M_i) \end{array} \right\}$$

then
16: **for all** $T' \in Committed(M_i)$ **do**
17: $M_i := M_i \cup \{T' \to B[l-1]\}$
18: **end for**
19: $M_i := M_i \cup \{B[l-1]^+\}$
20: **for all** $k \in [\![j, l-1]\!]$ **do**
21: $B[k+1] := B[k]$
22: **end for**
23: $B[j] := T$
24: **else**
25: $M_i := M_i \cup \{T^-\}$
26: **end if**
27: **end loop**

We translate this algorithm in our framework similarily to what we did with AES. We use Algorithm 5, and switch from Table 5 to Table 2.

5 Optimization-Based Replication

The protocols depicted in the previous sections suffer a problematic abort rate as they kill transactions more than necessary (AES, Table 3.2), or do not serialize concurrent updates in a good order (DBSM, Algorithm 5, line 16). We also pointed out that they may experience liveness issues when the system is under low load (DBSMR, see Section 4.3 and AES, see Section 3.4). Futhermore, they propagate one transaction at a time over the network (DBSM and DBSMR), whereas batch-processing transactions is possible.

We propose a new commitment protocol to remedy these issues.

Our idea is triple. We batch-process transactions in the same atomic broadcast. We compute the weakest static constraints to preserve serializability and causality. And we commit transactions trying to minimize the number of transactions aborted.

This last computation step is ensured with an heuristic as the problem is an NP-hard optimization problem (see futher).

In this section we first present our new protocol (OBR). Then we expose a more optimistic variant (OBRO) where we release write locks at the end of execution. We conclude with a variant for snapshot isolation.

5.1 OBR: overview

Our protocol works as follows:

	$T \prec T'$	$T \parallel T'$	$T' \prec T$
$RS(T) \cap WS(T') \neq \emptyset$	$T \rightarrow T'$	$T \rightarrow T'$	$T' \overset{\lhd}{\rightarrow} T$
$WS(T) \cap WS(T') \neq \emptyset$	$T \rightarrow T'$	$T \curlywedge T'$	$T' \rightarrow T$

Table 6: Static constraints in OBR

1. Transactions are executed against local cache using 2PL. Read locks are released at the end of execution, read sets, write sets and logical timestamps are then computed. Queries are locally committed.
2. sites batch transactions in the same atomic broadcast.
3. When a site *i* delivers such a set of transactions, it computes constraints linking transactions according to Table 6.
4. Then *i* takes a decision upon these transactions with an heuristic: *Decide*(). We specify this decision such that:
   - Non-commuting transactions are serialized.
   - All transactions are either committed or aborted.

   The decision process is strictly monotonic, i.e. each new decision is sound with relation to previous decisions.

5.2 Computing a decision

Formally *Decide*() is an algorithm whose input is a multilog $M = (K, +, -, \rightarrow, \lhd, \curlywedge)$, and whose output is a multilog $M' = (K', +', -', \rightarrow', \lhd', \curlywedge')$ such that :

1. *Decide*() adds only *decisions*, namely:
   - *Decide*() does not add new transactions: $K' = K$.
   - $+' \supseteq +$
   - $-' \supseteq -$
   - $\alpha \rightarrow' \beta \Rightarrow \alpha \rightarrow \beta \vee \alpha \curlywedge \beta$
   - $\curlywedge' = \curlywedge$
2. Multilog M' is decided.
3. If M is sound, then M' is sound.

According to this definition certification loops appearing in previous sections are all instances of *Decide*(). However we offer an improved *Decide*() algorithm (Algorithm 7) that aims at minimizing the abort rate.

We follow the general guidelines proposed by Shapiro and Krishna [SK05]. We decompose decision into three parts: serialization, conflict-breaking and validation: serialization orders any non-commuting pairs of transactions, conflict-breaking aborts at least one transaction in any $\rightarrow$-cycle, and validation commits the remaining set of non-aborted transactions.

Given T a blindwrite transaction ($RS(T) = \emptyset$), we serialize $T \curlywedge T'$ in $T' \rightarrow T$: lines 3 to 8. Indeed, for any transaction T", $T \rightarrow T''$ is not possible according to Table 6. Consequently no $\rightarrow$-cycle may exist serializing T and T' in $T' \rightarrow T$.

We serialize the remaining pairs of non-commuting transactions computing $\rightarrow'$: line 9 to 10. This relation extends the previous relation $\rightarrow$, minimizing the number of $\rightarrow$-cycles newly introduced.

Breaking $\rightarrow$-cycles minimizing the number of transactions aborted is stated as follows:

Definition. *Consider a graph $G = (V, E)$ where (i) each node in V is a transaction T of $K \setminus Committed(M)$ weighted by k, with k equals to one plus the number of distinct predecessors by $\lhd$ that*

Algorithm 7 *Decide*(M)

1: {Serialization}
2: let $SER := K \setminus Serialized(M)$
3: **for all** $T \in SER : RS(T) = \emptyset$ **do**
4: **for all** $T' \in SER : T \not\rightarrow T'$ **do**
5: $M := M \cup \{T' \rightarrow T\}$
6: **end for**
7: $SER := SER \setminus \{T\}$
8: **end for**
9: choose $\rightarrow'$ such that:
$\rightarrow \subseteq \rightarrow'$
$\wedge \forall T, T' \in SER,\ ((T,T') \in \rightarrow') \vee ((T',T) \in \rightarrow')$
10: $\rightarrow := \rightarrow'$
11: {Cycle breaking}
12: $M := breakCycles(M)$
13: {Validation}
14: **for all** $T \notin Decided(M)$ **do**
15: $M := M \cup \{T^+\}$
16: **end for**

T has in M, and (ii) for $(v,v') \in V$, a directed edge going from v to v′ exists in E, iff $v \rightarrow v'$ is in M. Conflict breaking is the problem of finding the minimum feedback vertex set of G.

This problem is an NP-complete optimization problem, and the literature upon this subject is important [GJ90]. Consequently we postulate the existence of an heuristic: *breakCycles*() (line 12).

At the end of the serialization process and the conflict breaking, remaining non-aborted transactions are committed: lines 14-16.

Algorithm 7 minimizes the number of $\rightarrow$-cycles created when serializing two writes , and reduces the number of transactions aborted when breaking conflicts. The abort rate of Algorithm 7 is consequently lower than AES and DBSM. We futher detail this result in Section 6.

5.3 OBR

OBR is depicted in Algorithm 8. Each loop is atomic.

Algorithm 8 OBR, code for site i

1: M_i {the site-multilog of i}
2: S_i {the site-schedule of i}
3: $D = (\emptyset,\emptyset,\emptyset,\emptyset,\emptyset,\emptyset)$ {a multilog containing previous decisions}
4: **loop** {execution}
5: same as Algorithm 2
6: **end loop**
7: ||
8: **loop** {propagation}
9: let $L := K \setminus Decided(M)$
10: AB-cast(L)
11: **end loop**
12: ||
13: **loop** {commitment}
14: AB-deliver(L)
15: $D := addConstraints(D \cup L)$
16: $D := Decide(D)$
17: $M_i := M_i \cup D$
18: **end loop**

The second idea in OBR is to broadcast batches of transactions with atomic broadcast: line 10. We ensure the growing monotonicity of our decisions with a local variable containing previous decisions D: lines 15 and 16. OBR ensures serializability and preserves causality.

5.4 Increasing the optimism

If we release write locks when transactions finish to execute, we increase the transactions throughput of the system. This result comes from the fact that we batch process transactions in a single atomic broadcast.

On the other hand, this technique also augments the probability that a $\rightarrow$-cycle may exist. But OBR

is designed to reduce the abort rate. Consequenlty we may expect that the number of transactions committed increase. Algorithm 9 depicts our proposal, OBRO.

Algorithm 9 OBRO, code for site i

```
1:  M_i {the site-multilog of i}
2:  S_i {the site-schedule of i}
3:  D {a multilog containing previous decisions}
4:  loop {execution}
5:     same as Algorithm 4
6:  end loop
7:  loop {propagation}
8:     let L := K \ Decided(M)
9:     L := L \ {T ∈ L | WS(T) = Ø}
10:    AB-cast(L)
11: end loop
12: ||
13: loop {commitment}
14:    same as Algorithm 8
15: end loop
```

Similarly to AESO, we release write locks at the end of execution: line 5. Doing so, queries have to wait before being committed, since they may see an inconstant state: line 9.

Interestingly Algorithm 9 may also serialize transactions in a better way than OBR. This result is detailed in Section 6.

5.5 The impact of Snapshot Isolation

Similarly to what we did in Section 3.6, we ensures SI in both of our algorithms by switching from Table 6 to Table 3 and Table 2 in our algorithms.

6 Comparison between AES, DBSM and OBR

AES, DBSM and OBR all ensure serializability. In this section we compare them according to fault tolerance and liveness, time and message complexity, abort rate, and implementation considerations.

6.1 Fault tolerance and Liveness

DBSM and OBR are based on an atomic broadcast primitive. Atomic broadcast is not solvable in asynchronous systems with crash-fail processes [FLP85]. However in a partially synchronous system with failure detectors, atomic broadcast becomes solvable even in the presence of faulty processes [CT96].

AES was not designed to be fault-resilient. It blocks if a site crashes, for instance (in a real-case deployment) during maintenance or if a site disconnects.

In Sections 3.4 and 4.3 we pointed out a liveness problem that could occur with AES and DBSMR. The impact of this issue is not negligible as during quiescent periods the commitment may block. Conversely, our protocol is designed to not suffer this liveness issue as transactions received by atomic broadcast are immediately decided.

6.2 Time performance

We measure the time performance of a distributed protocol with the latency degree: the smallest number of non-parallel communication steps required to solve a problem. The latency degree measures for a commitment protocol, the number of communication steps to commit or abort a transaction.

In AES a transaction T is sent epidemically to every distant sites, and decided once $HasRecvd(i,T,k)$ holds for every k. In the best case, it requires 2 communication steps.

DBSM and OBR are based on atomic broadcast. This communication primitive has a latency degree of 3 [CT96]. If broadcast over IP is possible, this value is reduced to 1 ([PSUC02].

We observed previously that AES and OBR batch-process transactions when they communicate. This idea improves substantially time-performance when the system becomes under medium to high charge.

6.3 Message complexity

Message complexity is measured as the total number of messages required to commit or abort a transaction. AES has a message complexity of $2n$, DBSM and OBR $3n$; $3n$ decreases to n if broadcast over IP is available.

Once again, batch-processing transactions in AES and OBR, decreases the message cost to commit a transaction since we send them many at a time.

6.4 Abort rate

AES aborts all concurrent conflicting transactions, DBSM and OBR try to minimize them; and DBSMR reorders transactions whereas DBSM not. It follows that AES aborts more transactions than DBSM which aborts more transactions than DBSMR. We now compare DBSMR with OBR.

First of all observe that Table 5 is a strict augmentation of Table 6. Consequently DBSMR computes stronger constraints than OBR to obtain the same result: serializability. But as DBSMR computes more constraints, it may also abort more transactions.

Table 7 illustrates this matter. We use the notation of [BHG87]: $r1[x]$ models a read from transaction $T1$ on data item x and $w2[z]$ models a write by transaction $T2$ on data item z.

Table 7 depicts a run during which two transactions $T1$ and $T2$ are concurrent to $T3$. The order of delivery is the following: $deliver(T1) < deliver(T2) < deliver(T3)$.

Recall now that two transactions with a non-empty write set, do not commute in DBSMR. Consequently when DBSMR receives $T2$, $T2$ is ordered after $T1$. The resulting schedule is $T1.T2$. Now when $T3$ is received; DBSMR aborts it as the schedules $T1.T2.T3$, $T1.T3.T2$ and $T3.T1.T2$ are not possible.

On the contrary our protocol does not order $T1$ and $T2$, and $T3$ is committed when received. The resulting schedule is $T2.T3.T1$.

$T1 = \{w1[x]\}$	$T2 \prec T1$
$T2 = \{r2[y], w2[z]\}$	$T1 \parallel T3$
$T3 = \{r3[x], w3[y]\}$	$T2 \parallel T3$
$deliver(T1) < deliver(T2) < deliver(T3)$	

Table 7: Unnecessary ordering of transactions with DBSMR

DBSMR serializes concurrent writes according to the order they are received with atomic broadcast. In particular it does not serialize blindwrite transactions properly.

Table 8 illustrates such a situation. Three concurrent transactions $T1$, $T2$ and $T3$ are delivered in the following order: $deliver(T2) < deliver(T1) < deliver(T3)$. DBSMR computes $T2$, then $T2.T1$, and finally aborts $T3$.

On the contrary OBR schedules $T1$ and $T2$ in $T1.T2$ when it receives the blindwrite transaction

$T2$. Then it computes $T1.T3.T2$ when $T3$ is delivered..

$T1 = \{w1[x], r1[y]\}$	$T1 \parallel T2$
$T2 = \{w2[x], w2[z]\}$	$T1 \parallel T3$
$T3 = \{w3[y], r3[z]\}$	$T2 \parallel T3$
$deliver(T2) < deliver(T1) < deliver(T3)$	

Table 8: Serialization of blindwrite transactions

Batch-processing transactions induces a lower abortion rate. Indeed we can compute a greater number of schedules when transactions are received set by set, than one by one.

To illustrate this claim, we consider a run depicted Table 9. In this run $T1$ has already been received, and $T2, T3$ are received within a set. Since $T2$ and $T3$ are batch-processed, we obtain the resulting schedule $T1.T2.T3$ where all the transactions are committed.

On the contrary, suppose that we deliver in two distinct messages $T3$ then $T2$ (it is possible since $T2$ release its read locks, and the two atomic broadcasts are independent). The serialization of $T1$ and $T2$ can lead to $T2.T1$, as both orders $T1.T2$ and $T2.T1$ do not abort any transaction. But when we deliver $T3$, we must abort it.

$T1 = \{w1[y], r2[z]\}$	$T1 \parallel T2$
$T2 = \{r2[x], w2[z]\}$	$T1 \parallel T3$
$T3 = \{w3[x], w3[y]\}$	$T2 \prec T3$
$deliver(T1) < deliver(T2, T3)$	

Table 9: Batch-processing transactions reduce abort rate

We said in Section 5.4 that OBRO may serialize transactions in a better way than OBR, Table 10 illustrates this.

Let $T1$, $T2$ and $T3$ be three transactions such that $T2 \stackrel{\lhd}{\rightarrow} T1$, $T2 \nparallel T3$ and $T3 \rightarrow T1$. OBR keeps lock at the end of execution, consequently $T2$ and $T3$ are

$T1 = \{r1[x], w1[y]\}$	$T2 \prec T1$
$T2 = \{w2[x], r2[z]\}$	$T1 \parallel T3$
$T3 = \{w3[x], r3[y]\}$	$T2 \parallel T3$

Table 10: Serialization in OBRO

ordered before $T1$ is received. Now since $T2 \rightarrow T3$ does not create more $\rightarrow$-cycles than $T3 \rightarrow T2$, $T2$ and $T3$ may be serialized in $T2 \rightarrow T3$; and when $T1$ is received, the constraints $T3 \rightarrow T1$, $T1 \rightarrow T3$, and $T3^+$ induce that $T1$ is aborted.

If we consider the execution with OBRO, and that $T1$ and $T2$ are sent in the same atomic broadcast, $T2$ and $T3$ are serialized in $T3 \rightarrow T2$ as it minimizes the number of $\rightarrow$-cycles: all transactions are committed.

6.5 Implementation considerations

Garbage collecting transactions in logs is encompassed in our concept of durable actions: an action is durable if it is decided and its predecessors by $\rightarrow$ and $\lhd$ are durable. Consequently given a run r, a certain point t of r, and a transactions $T \in M_i(t)$, T is durable in r, if $T \in Durable(M_i(t))$ holds, and in the remaining of the run, no new predecessors of T appear in M_i.

In our framework according to Table 4, a transaction T is durable in a run of AES as soon as it is executed on every sites (and hence committed). This is what Agrawal et al. do in [AES97]; they garbage-collect transactions as soon as they are committed.

In DBSM and OBR the durability of a transaction is achieved similarly. Indeed according to Tables 5 and 6, if every site execute T, T is durable.

7 Related work

Holliday et al. propose a quorum-based variant of AES to lower the abort rate of concurrent and conflicting transactions [HSAA03] . This variant does not ensure serializability, but only external consistency. Moreover the drawbacks of the approach remain: concurrent and conflicting transactions are antagonist, and the protocol still suffers a liveness issue.

Pedone et al. propose initially the database state machine approach as a reordering technique for distributed databases.[PGS97]. Oliveira et al. revisit the 1-copy equivalence of DBSM and point out that session guarantees such as read-yours-writes are not ensured [OPAaCA06]. To solve the problem they introduce a semantic link between reads and writes causally preceding them; this solution is very similar to what we depicted in Section 2.4.4 to introduce Prefix Consistent Snapshot Isolation.

The idea of considering optimistic replication as an optimization problem was firstly proposed in IceCube [PSM03a]. The IceCube approach was applied to databases in mobile environments [PSM03b], and in P2P environments [MP06]. However the reconciliation process was always centralized to a primary site.

IceCube is based on coarse-grained constraints. Shapiro et al. refine these constraints and introduce the Action-Constraint Framework to ease the understanding of replication [SBK04].

Kemme et al. propose a novel approach to implement eager replication [KA00]. This commitment protocol is based on the deferred update technique and atomic broadcast, but a single site decides if a transaction commits or aborts, and only one transaction at a time is sent in a single atomic broadcast.

Wiesmann and Schiper performed a quantitative comparison between the protocol of Kemme et al., DBSM and existing pessimistic approaches [WS05]. Their work show that the deferred update technique outperforms pessimistic approaches.

8 Conclusion

This paper depicts a detailed comparison between two existing optimistic database replication techniques: AES [AES97] and DBSM [PGS03], and a new solution: OBR, that we describe in detail. These techniques all implement the deferred update scheme, a database replication technique managing anywhere-anytime-anyway updates.

Our comparison emphasizes the basic building blocks of the deferred update scheme viz. the execution module, the propagation module, the certification module, and the static constraints computation. It furthermore alleviates the design of new variants: a snapshot isolated variant for AES and OBR, and an optimistic variant for OBR.

In our new commitment protocol OBR, we refine the consistency problem: serializability or snapshot isolation, as a graph problem, and solve it with an heuristic: $Decide()$. We also batch process transactions in a single communication primitive whereas previous approaches only send transactions one by one. We finally show qualitatively that our solution outperforms DBSM and AES with respect to latency, message cost and abort rate.

In a shorter term, we plan to corroborate these results with an implementation into our ACF middleware Telex [Tel]. In particular we intend to analyze the tradeoff between releasing write locks (OBRO) and keeping them at the end of execution (OBR), according to different workloads.

References

[AES97] D. Agrawal, A. El Abbadi, and R. C. Steinke. Epidemic algorithms in replicated databases (extended abstract). In *PODS '97: Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 161–172, New York, NY, USA, 1997. ACM Press.

[BCBT96] Anindya Basu, Bernadette Charron-Bost, and Sam Toueg. Solving problems in the presence of process crashes and lossy links. Technical Report TR96-1609, Cornell University, Computer Science Department, 1996.

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[EZP05] Sameh Elnikety, Willy Zwaenepoel, and Fernando Pedone. Database replication using generalized snapshot isolation. In *SRDS '05: Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*, pages 73–84, Washington, DC, USA, 2005. IEEE Computer Society.

[FLO+05] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, 2005.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Patterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[GJ90] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1990.

[Glo04] Global Development Group. PostGreSQL 7.4 Documentation, 2004.

[HSAA03] JoAnne Holliday, Robert Steinke, Divyakant Agrawal, and Amr El Abbadi. Epidemic algorithms for replicated databases. *IEEE Transactions on Knowledge and Data Engineering*, 15(5):1218–1238, 2003.

[KA00] Bettina Kemme and Gustavo Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Transactions on Database Systems*, 25(3):333–379, 2000.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[Mic05] Microsoft Corporation. Yukon Release Microsoft SQL Server, 2005.

[MP06] Vidal Martins and Esther Pacitti. Dynamic and distributed reconciliation in p2p-dht networks. In *European Conf. on Parallel Computing (Euro-Par)*, Dresden, Germany, 2006. Springer.

[OPAaCA06] Rui Oliveira, José Pereira, Jr Afrãnio Correia, and Edward Archibald. Revisiting 1-copy equivalence in clustered databases. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 728–732, New York, NY, USA, 2006. ACM Press.

[Ora97] Oracle corporation. Data concurrency and Consistency, Oracle8 Concepts, 1997.

[PGS97] F. Pedone, R. Guerraoui, and A. Schiper. Transaction reordering in replicated databases. In *Proceedings of the 16th*

Symposium on Reliable Distributed Systems (SRDS-16), Durham, North Carolina, USA, 1997.

[PGS03] F Pedone, R Guerraoui, and A Schiper. The database state machine approach. *Distrib. Parallel Databases*, 14(1):71–98, July 2003.

[PSM03a] Nuno Preguiça, Marc Shapiro, and J. Legatheaux Martins. SqlIceCube: Automatic semantics-based reconciliation for mobile databases. Technical Report TR-02-2003 DI-FCT-UNL, Universidade Nova de Lisboa, Dep. Informática, FCT, 2003.

[PSM03b] N. Preguiça, Marc Shapiro, and J. Legatheaux Martins. Sqlicecube: Automatic sematics-based reconciliation for mobile databases. Technical Report 2, Departamento de Informática FCT/UNL, 2003. URL=http://asc.di.fct.unl.pt/ nmp/papers/sqlice3-rep.pdf.

[PSUC02] Fernando Pedone, André Schiper, Péter Urbán, and David Cavin. Solving agreement problems with weak ordering oracles. In *EDCC-4: Proceedings of the 4th European Dependable Computing Conference on Dependable Computing*, pages 44–61, London, UK, 2002. Springer-Verlag.

[SBK04] Marc Shapiro, Karthikeyan Bhargavan, and Nishith Krishna. A constraint-based formalism for consistency in replicated systems. In *Proc. 8th Int. Conf. on Principles of Dist. Sys. (OPODIS)*, number 3544 in Springer-Verlag, pages 331–345, Grenoble, France, December 2004.

[SK05] Marc Shapiro and Nishith Krishna. The three dimensions of data consistency. In *Journées Francophones sur la Cohérence des Données en Univers Réparti (CDUR)*, pages 54–58, CNAM, Paris, France, November 2005.

[SSP06] Nicolas Schiper, Rodrigo Schmidt, and Fernando Pedone. Optimistic Algorithms for Partial Database Replication. In *10th International Conference on Principles of Distributed Systems (OPODIS'2006)*, pages 81–93, 2006. Also published as a Brief Announcement in the Proceedings of the 20th International Symposium on Distributed Computing (DISC'2006).

[Tel] Telex , http://gforge.inria.fr/projects/telex2/.

[WS05] Matthias Wiesmann and André Schiper. Comparison of database replication techniques based on total order broadcast. *IEEE Transactions on Knowledge and Data Engineering*, 17(4):551–566, 2005.