Consistent Checkpointing for Transaction Systems*

Roberto Baldoni, Francesco Quaglia
DIS, Università di Roma "La Sapienza"
Via Salaria 113, Roma, Italy
baldoni,quaglia@dis.uniroma1.it

Michel Raynal
IRISA, Campus de Beaulieu
35042 Rennes-Cedex, France.
raynal@irisa.fr

Abstract

Whether it is for audit or for recovery purposes, data checkpointing is an important problem of transaction systems. Actually, transactions establish dependence relations on data checkpoints taken by data object managers. So, given an arbitrary set of data checkpoints (including at least a single data checkpoint from a data manager, and at most a data checkpoint from each data manager), an important question is the following one: "Can these data checkpoints be members of a same consistent global checkpoint?". This paper answers this question by providing a necessary and sufficient condition suited to transaction systems. Moreover, to show its usefulness, two *non-intrusive* data checkpointing protocols are designed from this condition.

Keywords: Causal/Hidden Dependence, Checkpointing Protocol, Consistent Data Checkpoints, Transaction Systems, Fault-Tolerance.

1 Introduction

A transaction system is formed by a set of transactions that manipulate a set of data objects. Checkpointing the state of a such a system is important for audit or recovery purposes as this system abstracts the behavior of a database. During their execution, transactions, create dependencies among data objects. These dependencies are due to: (i) the serialization order of the transactions and (ii) messages exchanged to make a consistent access to the data. When one has to checkpoint

*This article appeared in The Computer Journal, Vol.44, No.2.

the state of a transaction system, it is always possible to design a special transaction, that reads all data objects and saves their current values. The underlying concurrency control mechanism ensures that this transaction gets a consistent state of the data objects. But this strategy is inefficient, intrusive (from the concurrency control point of view [21]) and not practical since the special read only transaction may take a very long time to execute and may cause intolerable delays for other transactions [16]. Moreover, as pointed out in [20], this strategy may drastically increase the cost of rerunning aborted transactions. So, it is preferable to base global checkpointing (1) on local checkpoints of data objects taken by their managers, and (2) on a mechanism ensuring mutual consistency of local checkpoints (this will ensure that it will always be possible to get consistent global checkpoints by piecing together local checkpoints).

In this paper we are interested in exploiting such an approach. We consider a transaction system in which each data object can be individually checkpointed (note that a data object could include, practically, a set of physical data items). If these checkpoints are taken in an independent way, there is the risk that no consistent global checkpoint can ever be formed (in the context of rollback recovery, this leads to the well known *domino effect* [19]). So, some kind of coordination is necessary when local checkpoints are taken in order to guarantee mutual consistency. In this paper we are interested in characterizing mutual consistency of local checkpoints. More precisely, we are interested in the following two issues.

- First, we address the following question $\mathcal{Q}(S)$: "Given an arbitrary set S of checkpoints, can this set be extended to get a global checkpoint (i.e., a set including exactly one checkpoint from each data object) that is consistent?". The answer to this question is well known when the set S includes exactly one checkpoint per data object [16][1]. It becomes far from being trivial, when the set S is incomplete, i.e., when it includes checkpoints only from a subset of data objects. When S includes a single data checkpoint, the previous question is equivalent to "Can this local checkpoint belong to a consistent global checkpoint?". This problem has been previously addressed and solved in a general setting [1] which includes message-passing and shared-memory models. We actually answer previous question by showing that also the transaction model presented in this paper is an instance of that general setting.

- Then, we focus on data checkpointing protocols. Let us consider the property $\mathcal{P}(C)$: "Local

[1] [6] provides a similar result for message-passing systems.

checkpoint C belongs to a consistent global checkpoint". We design two non-intrusive protocols. The first one ensures the previous property $\mathcal{P}$ when C is any local checkpoint. The second one ensures $\mathcal{P}$ when C belongs to a predefined set of local checkpoints.

The remainder of the paper consists of 5 main sections and an Appendix. Section 2 introduces the model of the transaction system we consider. Section 3 defines consistency of global checkpoints. Section 4 answers the previous question $\mathcal{Q}$. To provide such an answer, we study the kind of dependencies both the transactions and their serialization order create among checkpoints of distinct data objects. Specifically, it is shown that, while some data checkpoint dependencies are causal, and consequently can be captured on the fly, some others are "hidden", in the sense that, they cannot be revealed by causality. It is the existence of those hidden dependencies that actually makes non-trivial the answer to the previous question. Then, Section 5 shows how the necessary and sufficient condition stated in Section 4, can be used to design "transaction-induced" data checkpointing protocols ensuring the property $\mathcal{P}$ (namely, "Local checkpoint C belongs to a consistent global checkpoint"). These protocols allow managers of data objects to take checkpoints independently of each other[2], and use transactions as a means to diffuse information, among data managers, encoding dependencies on the states of data objects. When a transaction that accessed a data object is committed, the data manager of this object may be directed to take a checkpoint to guarantee that previously taken checkpoints belong to consistent global checkpoints. Such a checkpoint is called *forced* checkpoint. This is done by the data manager which exploits both its local control data and the information exchanged at the transaction commit point. Section 6 compares our protocols with other checkpointing protocols presented in the literature in the context of distributed database systems [16, 18, 21]. The comparison shows that, as [18] and [21], our protocols are *non-intrusive*, in the sense that, they do not interfere with normal system activities (for example, no transaction is ever aborted to determine a consistent global checkpoint). As opposed to other protocols, the ones we propose (i) determine consistent global checkpoints by means of an implicit (i.e., lazy) coordination among data managers, and (ii) do not need additional special purpose transactions. Finally Appendix shows how the proof of the necessary and sufficient condition stated in Section 4 can be seen as a particular instantiation of the proof of Theorem 3.2 given in [1].

[2]These checkpoints are called *basic*. They can be taken, for example, during CPU idle time.

2 Transaction Model

We consider a classical model of a transaction system consisting of a finite set of data objects, a set of transactions and a concurrency control mechanism (see [3, 8]).

Data Objects and Data Managers. A data object constitutes the smallest unit of data accessible to users. A non-empty set of data objects is managed by a data manager DM. From a logical viewpoint, we assume each data object x has a data manager DM_x. In Section 5 we discuss how this assumption can be implemented. We assume each data objects x has an initial value $init_x$.

Transactions. A transaction is defined as a partial order on *read* and *write* operations on data objects and terminates with a *commit* or an *abort* operation. $R_i(x)$ (resp. $W_i(x)$) denotes a read (resp. write) operation issued by transaction T_i on data object x. Each transaction is managed by an instance of the transaction manager (TM) that forwards its operations to the scheduler which runs a specific concurrency control protocol. The read/write set of a transaction is the set of all the data objects it reads and/or writes.

Concurrency control. A concurrency control mechanism schedules read and write operations issued by transactions in such a way that any execution of transactions is *strict* and *serializable*. This is not a restriction as concurrency control mechanisms used in practice (e.g., two-phase locking 2PL and timestamp ordering) generate schedules ensuring both properties [4]. The *strictness* property states that no data object may be read or written until the transaction that currently writes it either commits or aborts. So, a transaction actually writes a data object at its commit point. Hence, at some abstract level, which is the one considered in our analysis, transactions execute atomically at their commit points. If a transaction is aborted, strictness ensures no cascading aborts and the possibility to use *before images* for implementing abort operations which restore the value of an object before the transaction access. For example, 2PL generates such a behavior since it requires transactions to keep their write locks until they commit (or abort) [4].

Underlying system. TMs and DMs communicate by message exchange. No common memory is available. Message transmission times are unpredictable but finite and communication is reliable (each message will eventually be delivered).

Execution. Let $T = \{T_1, T_2, \ldots, T_n\}$ be a set of transactions accessing a set $O = \{o_1, o_2, \ldots, o_m\}$ of data objects (to simplify notations, data object o_i is identified by its index i). An execution E over T is a partial order on all read and write operations of the transactions belonging to T; this partial order respects the order defined in each transaction. Moreover, let $<_x$ be the partial order defined on all operations accessing a data object x, i.e., $<_x$ orders all pairs of conflicting operations (two operations are conflicting if they access the same object and one of them is a write). Given an execution E defined over T, T is structured as a *partial order* $\widehat{T} = (T, <_T)$ (see [3]) where $<_T$ is the following (classical) relation defined on T:

$$T_i <_T T_j \iff (i \neq j) \wedge (\exists x \ : \ (R_i(x) <_x W_j(x)) \vee (W_i(x) <_x W_j(x)) \vee (W_i(x) <_x R_j(x)))$$

3 Consistent Global Checkpoints

3.1 Local States and Their Relations

Each write on a data object x issued by a transaction defines a new version of x. Let σ_x^i denote the i-th version of x; σ_x^i is called a *local state* (σ_x^0 is the initial local state of x). Transactions establish dependencies between local states. This can be formalized in the following way. When T_k issues a write operation $W_k(x)$, it changes the state of x from σ_x^i to σ_x^{i+1}. More precisely, σ_x^i and σ_x^{i+1} are the local states of x, just before and just after the execution[3] of T_k, respectively. This can be expressed in the following way by extending the relation $<_T$ to include local states:

$$T_k \ changes \ x \ from \ \sigma_x^i \ to \ \sigma_x^{i+1} \iff (\sigma_x^i <_T T_k) \wedge (T_k <_T \sigma_x^{i+1})$$

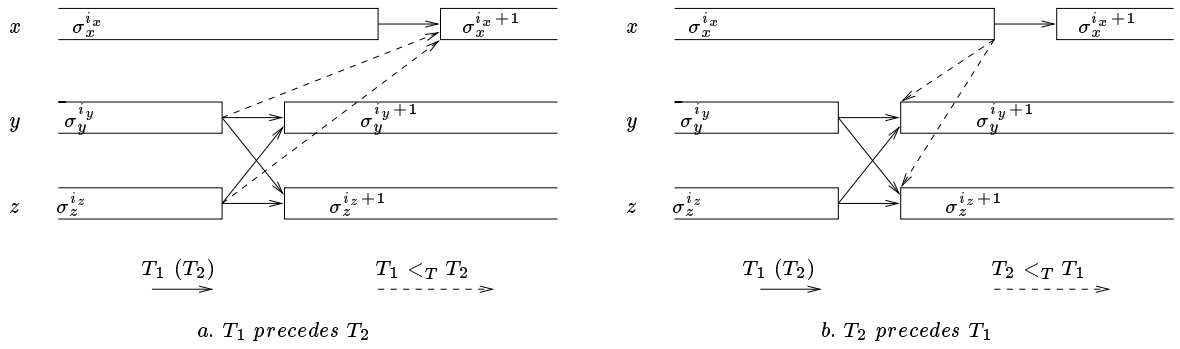
Let $<_T^+$ be the transitive closure of the extended relation $<_T$. When we consider only local states, we get the following *happened-before* relation denoted $<_{LS}$ (which is similar to Lamport's happened-before relation defined on events [13] in the process/message-passing model):

Definition 3.1 *(Precedence on local states, denoted $<_{LS}$)*

$$\sigma_x^i \ <_{LS} \sigma_y^j \iff \sigma_x^i <_T^+ \sigma_y^j$$

As the relation $<_T$ defined on transactions is a partial order, it is easy to see that the relation $<_{LS}$ defined on local states is also a partial order. Figure 1 shows examples of the relation $<_{LS}$. It

[3] Remind that, as we consider strict and serializable executions, "Just before and just after the execution of T_k" means "Just before and just after T_k is committed".



a. T_1 *precedes* T_2 *b.* T_2 *precedes* T_1

Figure 1: Partial Order on Local States

considers three data objects x, y, and z, and two transactions T_1 and T_2. Transactions are defined in the following way:

$$T_1 : \ R_1(x);\ W_1(y);\ W_1(z);\ commit_1$$
$$T_2 : \ R_2(y);\ W_2(x);\ commit_2$$

As there is a read-write conflict on x, two serialization orders are possible. Figure 1.a displays the case $T_1 <_T T_2$ while Figure 1.b displays the case $T_2 <_T T_1$. Each horizontal axis depicts the evolution of the state of a data object. For example, the second axis is devoted to the evolution of y: $\sigma_y^{i_y}$ and $\sigma_y^{i_y+1}$ are the states of y before and after T_1, respectively.

Let us consider Figure 1.a. It shows that $W_1(y)$ and $W_1(z)$ add four pairs of local states to the relation $<_{LS}$, namely:

$$\sigma_y^{i_y} <_{LS} \sigma_y^{i_y+1};\ \sigma_z^{i_z} <_{LS} \sigma_z^{i_z+1};\ \sigma_y^{i_y} <_{LS} \sigma_z^{i_z+1};\ \sigma_z^{i_z} <_{LS} \sigma_y^{i_y+1}$$

The relation $<_T$ adds two pairs of local states to $<_{LS}$:

$$\sigma_y^{i_y} <_{LS} \sigma_x^{i_x+1};\ \sigma_z^{i_z} <_{LS} \sigma_x^{i_x+1}$$

The latter two dependencies are due to the serialization order.

Precedence on local states, due to write operations of transactions T_1 and T_2, are indicated with continuous arrows, while the ones due to the serialization order are indicated with dashed arrows. Figure 1.b shows precedences on local states when the serialization order is reversed.

3.2 Consistent Global States

Let T be any set of transactions. Informally, the state reached by a transaction system which includes all the updates made by all the transactions in T is a consistent one.

A *global state* of the transaction system is a set of local states, one from each data object. A global state $G = \{\sigma_1^{i_1}, \sigma_2^{i_2}, \ldots, \sigma_m^{i_m}\}$ is *consistent* if it does not contain two dependent local states, i.e., if:

$$\forall x, y \in [1, \ldots, m] \Rightarrow \neg(\sigma_x^{i_x} <_{LS} \sigma_y^{i_y})$$

Let us consider again Figure 1.a. The three global states $(\sigma_x^{i_x}, \sigma_y^{i_y}, \sigma_z^{i_z})$, $(\sigma_x^{i_x}, \sigma_y^{i_y+1}, \sigma_z^{i_z+1})$ and $(\sigma_x^{i_x+1}, \sigma_y^{i_y+1}, \sigma_z^{i_z+1})$ are consistent. The global state $(\sigma_x^{i_x+1}, \sigma_y^{i_y}, \sigma_z^{i_z+1})$ is not consistent either because $\sigma_y^{i_y} <_{LS} \sigma_x^{i_x+1}$ (due to the fact that $T_1 <_T T_2$) or because $\sigma_y^{i_y} <_{LS} \sigma_z^{i_z+1}$ (due to the fact that T_1 writes both y and z). Intuitively, a non-consistent global state is a global state that could not be seen by any omniscient observer of the system. It is possible to show that, as in the process/message-passing model, the set of all the consistent global states is a lattice [7].

3.3 Consistent Global Checkpoints

A *local checkpoint* (or equivalently a *data checkpoint*) of a data object x is a local state of x that has been saved in a safe place[4] by the data manager of x. So, all the local checkpoints are local states, but only a subset of local states are defined as local checkpoints. Let C_x^i $(i \geq 0)$ denote the i-th local checkpoint of x; i is called the *index* of C_x^i. Note that C_x^i corresponds to some σ_x^j with $i \leq j$. A *global checkpoint* is a set of local checkpoints one for each data object. It is *consistent* if it is a consistent global state.

We assume that all the initial local states are checkpointed. Moreover, we also assume that, when we consider any point of an execution E, each data object will eventually be checkpointed.

4 Dependence on Data Checkpoints

4.1 Introductory Example

As indicated in the previous section, transactions create dependencies among local states of data objects due to write operations of each transaction, or due to the serialization order. Let us consider the following 7 transactions accessing data objects x, y, z and u:

[4]For example, if x is stored on a disk, a copy is saved on another disk.

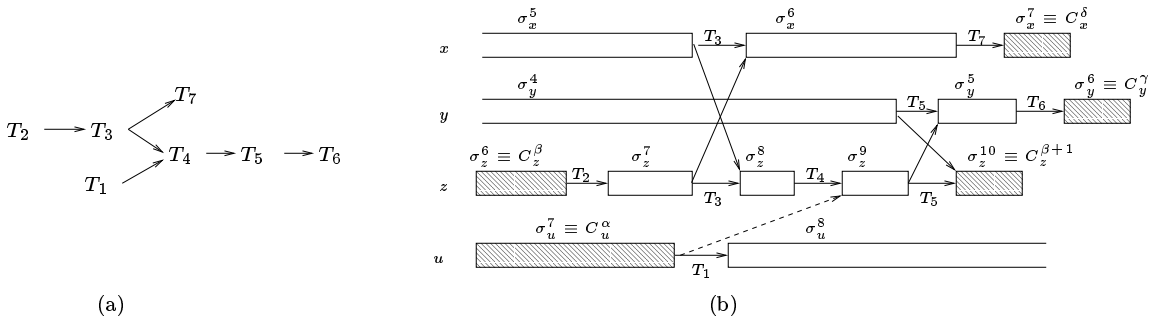


Figure 2: (a) A Serialization Order and (b) Data Checkpoint Dependencies

$$T_1 : \ R_1(u);\ W_1(u);\ commit_1$$
$$T_2 : \ R_2(z);\ W_2(z);\ commit_2$$
$$T_3 : \ R_3(z);\ W_3(z);\ W_3(x);\ commit_3$$
$$T_4 : \ R_4(z);\ R_4(u);\ W_4(z);\ commit_4$$
$$T_5 : \ R_5(z);\ W_5(y);\ W_5(z);\ commit_5$$
$$T_6 : \ R_6(y);\ W_6(y);\ commit_6$$
$$T_7 : \ R_7(x);\ W_7(x);\ commit_7$$

Figure 2.a depicts a possible serialization imposed by the concurrency control mechanism. Figure 2.b describes dependencies between local states generated by this execution. Five local states are defined as data checkpoints (they are indicated by dark rectangles). We study dependencies between those data checkpoints. Let us first consider C_u^α and C_y^γ. C_u^α is the (checkpointed) state of u before T_1 wrote it, while C_y^γ is the (checkpointed) state of y after T_6 wrote it (i.e., just after T_6 is committed). The serialization order (see Figure 2.a) shows that $T_1 <_T T_6$, and consequently $C_u^\alpha <_{LS} C_y^\gamma$, i.e., the data checkpoint C_y^γ is causally dependent on [13] the data checkpoint C_u^α (Figure 2.b shows that there is a directed path from C_u^α to C_y^γ). Now let us consider the pair of data checkpoints consisting of C_u^α and C_x^δ. Figure 2.b shows that C_u^α precedes T_1, and that C_x^δ follows T_7. Figure 2.a indicates that T_1 and T_7 are not connected in the serialization graph. So, there is no causal dependence between C_u^α and C_x^δ (Figure 2.b shows that there is no directed path from C_u^α to C_x^δ). But, as the reader can check, there is no consistent global checkpoint including both C_u^α and C_x^δ ([5]). So there is a *hidden* dependence between C_u^α and C_x^δ which prevents them to belong to the same consistent global checkpoint.

[5] Adding C_y^γ and C_z^β to C_u^α and C_x^δ cannot produce a consistent global state as $C_z^\beta <_{LS} C_x^\delta$. Adding $C_z^{\beta+1}$ instead of C_z^β has the same effect as $C_u^\alpha <_{LS} C_z^{\beta+1}$.

4.2 Necessary and Sufficient Condition

In this section we provide the necessary and sufficient condition stated in the Introduction. This condition is based on an unified definition of dependence that takes into account both causal and hidden dependencies. We provide this definition below:

Definition 4.1 *(Dependence Path)*[6]

There is a dependence path (DP) from a data checkpoint C_x^i to C_y^j (denoted $C_x^i \overset{DP}{\rightarrow} C_y^j$) if there exists a sequence $C_{z_1}^{i_1}, \ldots, C_{z_r}^{i_r}$ ($r \geq 0$; $r = 0$ corresponding to the empty sequence) such that:

$$(C_x^i <_{LS} C_{z_1}^{i_1}) \ \wedge \ (C_{z_1}^{i_1-1} <_{LS} C_{z_2}^{i_2}) \wedge \ \ldots \ \wedge \ (C_{z_{r-1}}^{i_{r-1}-1} <_{LS} C_{z_r}^{i_r}) \ \wedge \ (C_{z_r}^{i_r-1} <_{LS} C_y^j)$$

In the example depicted in Figure 2, the hidden dependence between C_u^α and C_x^δ can be now denoted $C_u^\alpha \overset{DP}{\rightarrow} C_x^\delta$ as $C_u^\alpha = \sigma_u^7 <_{LS} C_z^{\beta+1}$ and $C_z^\beta <_{LS} C_x^\delta$.

We are now in the position to state the necessary and sufficient condition whose proof is shown in Appendix. This condition will be used in the following to derive "transaction-induced" checkpointing protocols.

Theorem 4.1 *Let $\mathcal{I} \subseteq \{1, \ldots, m\}$ and $\mathcal{S} = \{C_x^{i_x}\}_{x \in \mathcal{I}}$ be a set of data checkpoints. Then $\mathcal{S}$ is a part of a consistent global checkpoint if and only if $\forall x, y \in \mathcal{I} \ \Rightarrow \ \neg(C_x^{i_x} \overset{DP}{\rightarrow} C_y^{i_y})$.*

4.3 Globally Transaction-Consistent Checkpointing

Let us call *Recovery Line*[7] a line joining all the data checkpoints of a consistent global checkpoint. Let us remind black and dashed arrows introduced in the example of Section 3.1: a black arrow denotes a local checkpoint precedence created by a transaction, while a dashed arrow denotes a local checkpoint precedence created by the serialization order. When considering such black and dashed arrows (see Figure 2), it is possible to show that a recovery line $\mathcal{L}$ is consistent iff:

- No black arrow crosses $\mathcal{L}$.
- No dashed arrow crosses $\mathcal{L}$ from the right of $\mathcal{L}$ to the left of $\mathcal{L}$.

[6] This definition generalizes the Z-path notion which has been introduced by Netzer and Xu [15] in the context of asynchronous message-passing systems. This notion has been investigated in [11, 15].

[7] Also called *cut*, when adopting the distributed computing terminology.

In a recovery line of a message-passing system, no message crosses it from the right to the left [6]. It is interesting to remark that the notion of recovery line in the process/message-passing model is actually equivalent to the notion of *globally transaction-consistent checkpoint* used by many authors in the data object/transaction model (e.g., [14], [16], [18],[21]).

A global checkpoint $\mathcal{C}$ is transaction-consistent if (i) for each transaction T_i all the T_i's updates are either included in $\mathcal{C}$ or not at all, and, (ii) in the case T_i is included in $\mathcal{C}$, all the transactions T_j such that $T_j <_T T_i$ are also included in $\mathcal{C}$.

The part (i) of the previous definition is equivalent to "*no black arrow crosses the recovery line* $\mathcal{L}$". If a black arrow would cross $\mathcal{L}$, this indicates that part of the updates of a transaction have been performed on the left side of $\mathcal{L}$ and the remaining transaction's updates on the $\mathcal{L}$'s right side. Then $\mathcal{L}$ would not be transaction-consistent. The part (ii) is equivalent to "*no dashed arrow crosses* $\mathcal{L}$ *from the right of* $\mathcal{L}$ *to the left of* $\mathcal{L}$". In fact, if a dashed arrow would cross $\mathcal{L}$ from the right to the left, this means there exists a pair of transactions T_j and T_i with $T_j <_T T_i$ such that T_j's updates are on the right side of $\mathcal{L}$ and T_i's updates are on the left side. Then $\mathcal{L}$ would not be transaction-consistent.

5 Deriving "Transaction-Induced" Checkpointing Protocols

Required Properties. If we suppose that the set S includes only a checkpoint C, Theorem 4.1 leads to an interesting corollary:

Corollary 5.1 *C belongs to a consistent global checkpoint if and only if* $\neg(C \overset{DP}{\rightarrow} C)$.

Hence, providing checkpointing protocols ensuring that $\neg(C \overset{DP}{\rightarrow} C)$ for each checkpoint C, guarantees the property $\mathcal{P}(C)$ defined in the introduction for each checkpoint. These protocols are interesting for two reasons:

1. They avoid wasting time in taking a data checkpoint that will never be used in any consistent global checkpoint, and
2. No domino effect can ever take place as any data checkpoint belongs to a consistent global checkpoint.

Let the *timestamp* of a local checkpoint be a non-negative integer such that:

$$\forall x, i, j : ((i < j) \Rightarrow (timestamp(C_x^i) < timestamp(C_x^j)))$$

A *timestamping function* associates a timestamp with each local checkpoint. (The indexes of local checkpoints define a particular timestamping function).

Moreover, let us consider the following property $\mathcal{TS}$: "Let $\mathcal{S}_n$ be the set formed by data checkpoints timestamped n. If $\mathcal{S}_n$ includes a checkpoint per object, then it constitutes a consistent global checkpoint". In the following we provide two checkpointing protocols:

- The first protocol ($\mathcal{A}$) guarantees $\mathcal{P}$ for all local checkpoints, and guarantees $\mathcal{TS}$ for any value of n.
- The second protocol ($\mathcal{B}$) ensures $\mathcal{P}$ only for a subset of local checkpoints, and $\mathcal{TS}$ for some particular values of n.

Actually, those protocols can be seen as adaptations (to the data object/transaction model) of protocols developed for the process/message-passing model. More precisely, protocol $\mathcal{A}$ adapts Briatico *et al.*'s protocol [5], while protocol $\mathcal{B}$ modifies the Wang-Fuchs's protocol [22]. Note that these two particular protocols can be derived from a more general timestamp based protocol proposed in [9].

Local Control Variables. In both protocols we assume each data manager DM_x has a variable ts_x, which stores the timestamp of the last checkpoint of x (it is initialized to zero); i_x denotes the index value of the last checkpoint of x. Moreover, data managers can take checkpoints independently of each other (*basic checkpoints*), for example, by using a periodic algorithm which could be implemented by associating a timer with each data manager (a local timer is set whenever a checkpoint is taken; and a basic checkpoint is taken by the data manager when its timer expires). Data managers are directed to take additional data checkpoints (*forced checkpoints*) in order to ensure $\mathcal{P}$ or $\mathcal{TS}$. The decision to take forced checkpoints is based on the control information piggybacked by commit messages of transactions.

A protocol consists of two interacting parts. The first part, shared by both protocols, specifies the checkpointing-related actions of transaction managers. The second part defines the rules data managers have to follow to take data checkpoints.

Protocols $\mathcal{A}$ and $\mathcal{B}$: Behavior of a Transaction Manager. Let RW_{T_i} be the read/write set of a transaction T_i managed by a transaction manager TM_i. We assume each time an operation of T_i is issued by TM_i to a data manager DM_x, it returns the pair <value of x, value of its current

timestamp ts_x>. TM_i stores in $MAX\_TS_{T_i}$ the maximum value among the timestamps of the data objects read and/or written by T_i. When the transaction T_i is committed, the transaction manager TM_i sends a COMMIT message to each data manager DM_x involved in RW_{T_i}. These COMMIT messages piggyback $MAX\_TS_{T_i}$.

Protocol $\mathcal{A}$: Behavior of a Data Manager. As far as checkpointing is concerned, the behavior of a data manager DM_x is defined by the two following procedures namely `take-basic-ckpt` and `take-forced-ckpt`. They define the rules associated with checkpointing.

`take-basic-ckpt(`$\mathcal{A}$`)` :

When the timer expires:

(AB1) $i_x \leftarrow i_x + 1$; $ts_x \leftarrow ts_x + 1$;

(AB2) Take checkpoint $C_x^{i_x}$; $timestamp(C_x^{i_x}) \leftarrow ts_x$;

(AB3) Reset the local timer.

`take-forced-ckpt(`$\mathcal{A}$`)` :

When DM_x **receives** COMMIT$(MAX_TS_{T_i})$ **from** TM_i:

if $ts_x < MAX\_TS_{T_i}$ **then**

(A1) $i_x \leftarrow i_x + 1$; $ts_x \leftarrow MAX\_TS_{T_i}$;

(A2) Take a (forced) checkpoint $C_x^{i_x}$; $timestamp(C_x^{i_x}) \leftarrow ts_x$;

(A3) Reset the local timer.

endif;

(A4) process the COMMIT message.

From the increase of the timestamp variable ts_x of a data object x, and from the rule associated with the taking of forced checkpoints (`take-forced-ckpt(`$\mathcal{A}$`)` which forces a data checkpoint whenever $ts_x < MAX\_TS_{T_i}$), the condition $\neg(C_x^{i_x} \stackrel{DP}{\rightarrow} C_x^{i_x})$ follows for any data checkpoint $C_x^{i_x}$. Actually, this simple protocol ensures that, if $C_x^{i_x} \stackrel{DP}{\rightarrow} C_y^{i_y}$, then $timestamp(C_x^{i_x}) < timestamp(C_y^{i_y})$ [10].

It follows from the previous observation that if two data checkpoints have the same timestamp value, then they cannot be related by $\stackrel{DP}{\rightarrow}$. So, all the sets $\mathcal{S}_n$ that exist are consistent. Note that the `take-forced-ckpt(`$\mathcal{A}$`)` rule may produce gaps in the sequence of timestamps assigned to data checkpoints of a data object x. So, from a practical point of view, the following remark is interesting: when no data checkpoint of a data object x is timestamped by a given value n, then the

first data checkpoint of x whose timestamp is greater than n, can be included in a set containing data checkpoints timestamped by n, to form a consistent global checkpoint.

Protocol $\mathcal{B}$: Behavior of a Data Manager. This protocol introduces a system parameter $Z \geq 1$ known by all the data managers [22]. When considering a data object x, this protocol ensures $\neg(C_x \overset{DP}{\rightarrow} C_x)$ only for a subset of data checkpoints, namely, those whose timestamps are equal to $a \times Z$ (where $a \geq 0$ is an integer). Moreover, if $\forall x$ there exists a data checkpoint timestamped $a \times Z$, then the global checkpoint $\mathcal{S}_{a \times Z}$ exists and is consistent.

The rule `take-basic-ckpt(`$\mathcal{B}$`)` is the same as the one of the protocol $\mathcal{A}$. In addition to the previous control variables, each data manager DM_x has an additional variable V_x, which is incremented by Z each time a data checkpoint timestamped $a \times Z$ is taken. The rule `take-forced-ckpt(`$\mathcal{B}$`)` is the following.

`take-forced-ckpt(`$\mathcal{B}$`)` :

When DM_x **receives** COMMIT$(MAX_TS_{T_i})$ **from** TM_i:

if $V_x < MAX\_TS_{T_i}$ **then**

(B1) $i_x \leftarrow i_x + 1$; $ts_x \leftarrow \lfloor MAX\_TS_{T_i}/Z \rfloor \times Z$;

(B2) Take a (forced) checkpoint $C_x^{i_x}$; $timestamp(C_x^{i_x}) \leftarrow ts_x$;

(B3) Reset the local timer;

(B4) $V_x \leftarrow V_x + Z$.

endif;

(B5) Process the COMMIT message.

About coordination. The proposed protocols determine consistent global checkpoints by using a *lazy* coordination (i.e., neither control messages nor checkpoint coordinator process) which is propagated among data managers by transactions (with COMMIT messages).

Protocol $\mathcal{B}$ seems to be particularly interesting as it shows a classical tradeoff, mastered by the parameter Z, between the number of forced checkpoints and the extent of rollback during a recovery phase. The greater Z, the larger the rollback distance.

Moreover, basic checkpoints whose indices are not a multiple of Z can be used, in case of a failure, to attempt a local recovery which do not force the other data objects to rollback. If a local recovery is not possible due to the inter-data checkpoint dependencies, then all the data objects rollback to the previous consistent global checkpoint.

Implementation issues. In Section 2 we have assumed a logical data manager for each data object. This simplifies the theoretical framework presented in Section 4 and the presentation of the transaction-induced protocols. However, in a real system, data objects can be grouped into sites and, in such a case, there is one data manager for each site which is responsible for accessing all local data objects. This data manager could be implemented as a sequential server or a concurrent one managing a finite number of threads. Each time a commit request is received, the data manager server executes the "take-forced-checkpoint" rule atomically. Timers associated with each data object could be managed by a background process at each site. When a timer expires, the background process sends a message to the data manager that will execute the "take-basic-checkpoint" rule. Note that this background process could also run some more refined basic checkpoint strategies in order, for example, to keep the timestamp values of the data object as close as possible (taking more basic checkpoints in less accessed data objects). This would allow, first, a reduction of the number of forced checkpoints and, secondly, a reduction of the rollback extent during recovery [2].

In a database, which is an istance of the transaction system presented in this paper, the checkpointing technique is mainly used for recovery purposes. A global checkpoint is usually seen as a consistent back-up copy [14]. So when a data object is checkpointed, the value of the object and its timestamp are saved in a safe area (e.g., another disk). When a site fails (e.g., loss of volatile storage, media failure etc.), a recovery procedure is started. This procedure seeks, in the safe area of each site, the minimum among timestamps associated with each data objects (say n). Then the transaction system can be restarted from the consistent global checkpoint including all the checkpoint timestamped n one for each data object. Note that the same procedure, when running in parallel with the normal activities, may act as a garbage-collection process. All checkpoints whose timestamp is less than n can be discarded from the safe area of a site as they will no longer be used by a recovery procedure.

6 Related Work on Distributed Databases

This section presents three checkpointing protocols proposed in the context of distributed database systems, namely Son-Agrawal protocol (SA) [21], Pilarski-Kameda protocol [16] (PK) and Pu et al. protocol [17, 18] (PU). In this context, we show main differences among these protocols and our solutions.

SA determines consistent global checkpoints by means of a two phase protocol using a checkpoint coordinator process that exchanges messages with its checkpoint subordinates processes, one for each site. Each site maintains an independent local timestamp (like a Lamport scalar clock [13]) and a timestamp is associated with each transaction[8]. The first phase is used to agree on a common timestamp value among all sites. This value, say n, actually slits transactions into two groups: transactions that have a timestamp less than, or equal to, n and transactions with timestamp greater than n. In the second phase, the checkpoint process in each site is delayed till all transactions whose timestamp is less than or equal to n are committed. Once the checkpoint process dumped the system state in a safe place, transactions whose timestamp is greater than n are executed. Note that during the first phase, transactions are not stopped, however their updates are stored in a private area that can be read by the checkpointing process to execute a transaction-consistent dump. A similar approach using control messages to split transactions in two groups in order to get globally transaction consistent checkpoints has been proposed by Kim and Park in [12].

PU assumes each data object has a colour either black or white. Before the checkpointing process starts all data objects are white. The black colour indicates that the data object has been read by the checkpointing process. The checkpointing process continues till all data objects are black. Transactions takes a colour from the data objects they access. A transaction is white (resp. black) if all data object it accessed were white (resp. black). A transaction is grey if it accessed at least one black and one white data object. In a first version ([17]), PU aborted each grey transaction in order to ensure serializability and to determine transaction-consistent global checkpoints. A second version of PU, namely *save some* [18], avoids to abort grey transactions by saving the *before values* of the data objects updated by each grey or white transaction in a private memory area accessible to the checkpointing process. This allows the execution of a transaction consistent dump. Compared to SA, PU splits transaction into two groups in a lazy way (by means of an "infection" from data objects) without exchanging control messages. However this approach increases the transaction response time and requires a large and unbounded memory capacity (the private memory required for saving before values could be larger than the size of the database itself) as it is expected that grey transactions will be a wider majority of all transactions.

PK modifies PU in order to bound the size of the required private memory. The checkpoint process is implemented as a set of read-only transactions one for each data object. Each data

[8] As it uses timestamps, this protocol is well suited to concurrency control based on timestamp.

object has a colour white, grey or black. Initially each data object is white. Transactions can be black or white. Initially checkpointing transactions are black and normal transactions can be either black or white. A normal transaction turns black after either overwriting a gray data object or accessing a black data object. A data object changes from white to gray (resp. black) when a finally black transaction (i.e., a transaction which is black at the commit time) reads (resp. overwrites) it. A data object changes from grey to black when written by any transaction. A consistent global checkpoint of the database is formed by the final non-black state of each data object. The introduction of the grey colour actually delays the time of reading of the data object by the checkpoint transaction, this reduces the size of the required private memory compared to PU. On the other hand, transaction response time increases due to previous delay and to the fact that the concurrency control has to manage the checkpointing transactions.

Compared to SA, our checkpointing protocols employ a *lazy* coordination among data managers (neither control messages nor a checkpoint coordinator is required). As opposed to PU and PK, our protocols do not overload the concurrency control with special purpose checkpointing transactions and do not need to manage colours. Moreover, they do not need private memory to be read by the checkpointing process to store partial transaction updates. On the negative side, the safe memory area where storing a copy of the checkpointed data objects can be larger than the size of the entire system (many checkpoints with distinct timestamps of a data object can be stored in the safe area at the same time). However, this size can be kept as small as possible by running frequently the garbage collection procedure described in Section 5.

7 Conclusion

This paper has presented a formal approach for consistent data checkpoints in transaction systems. Given an arbitrary set of data checkpoints (including at least a single data checkpoint from a data manager, and at most a data checkpoint from each data manager), we answered the following important question "Can these data checkpoints be members of a same consistent global checkpoint?" by providing a necessary and sufficient condition. We have also designed two *non-intrusive* data checkpointing protocols from this condition; these checkpointing protocols use a lazy coordination by means of transactions that diffuse information among data managers and thus do not interfere with normal activities.

In the context of distributed databases, when ensuring each data checkpoint belongs to a consis-

tent global checkpoint, the previous two protocols are expected to exhibit good performance with respect to previous checkpointing protocols appeared in the literature such as [16], [18] and [21]. The latter protocol relies on an intrusive global coordination among data managers using control messages like the on proposed by Chandy-Lamport in [6] in the context of message-passing applications. As opposed to [16] and [18], our protocols do not special purpose checkpointing transactions, do not need to manage colours and do not need private memory to be read by the checkpointing process to store partial transaction updates.

Acknowledgments

The authors would like to thank the anonymous referees who made very helpful suggestions that improved the content and the presentation of the paper. More specifically, the authors thank one of the anonymous referees for his/her help in the construction of the proof shown in Appendix. This construction actually points out the generality and the powerfulness of Theorem 3.2 presented in [1]. The authors want also to thank Jean-Michel Hélary, Rob Netzer and Yi-Min Wang for interesting discussions on the checkpointing problem in message-passing distributed systems.

References

[1] Baldoni, R., Hélary, J.M. and Raynal, M., Consistent Records in Asynchronous Computations, *Acta Informatica*, 35:441-455, 1998.

[2] Baldoni, R., Quaglia, F., and Fornara, P., An Index-Based Checkpointing Algorithm for Autonomous Distributed Systems, *IEEE Transactions on Parallel and Distributed Systems*, 10(2):181-192, 1999.

[3] Bernstein, P.A., Hadzilacos, V. and Goodman, N., Concurrency Control and Recovery in Database systems, *Addison Wesley Publishing Co.*, Reading, MA, 1987.

[4] Breitbart, Y., Georgakopoulos, D., Rusinkiewicz, M. and Silberschatz, A., On Rigorous Transaction Scheduling, *IEEE Transactions on Software Engineering*, 17(9):954-960, 1991.

[5] Briatico, D., Ciuffoletti, A. and Simoncini, L., A Distributed Domino-Effect Free Recovery Algorithm, *in Proc. IEEE Int. Symposium on Reliability Distributed Software and Database*, pp. 207-215, 1984.

[6] Chandy, K.M. and Lamport, L., Distributed Snapshots: Determining Global States of Distributed Systems, *ACM Transactions on Computer Systems*, 3(1):63-75, 1985.

[7] Fromentin, E. and Raynal, M., Shared Global States in Distributed Computations. *Journal of Computer and System Sciences* 55(3): 522-528, 1997.

[8] Gray J.N. and Reuter A., Transaction Processing: Concepts and Techniques, *Morgan Kaufmann*, 1070 pages, 1993.

[9] Hélary J.-M., Mostéfaoui A. and Raynal M., Virtual Precedence in Asynchronous Distributed Systems: Concept and Applications. *Proc. 11th Int. Workshop on Distributed Algorithms*, Springer-Verlag LNCS 13220, September 1997, pp. 170-184.

[10] Hélary J.-M., Mostéfaoui A., Netzer R.H.B. and Raynal M., Communication-Based Prevention of Useless Checkpoints in Distributed Computations. *Distributed Computing*, 13(1):29-43, 2000.

[11] Hélary J.-M., Netzer R.H.B. and Raynal M., Consistency Issues in Distributed Checkpoints. *IEEE Transactions on Software Engineering*, 25(2):274-281, 1999.

[12] Kim J.L., and Park, T., An Efficient Recovery Scheme for Locking-Based Distributed Database Systems, *Proc. 13th IEEE Symposium on Reliable Distributed Systems*, Dana Point, CA, USA, 1997, pp. 183-190.

[13] Lamport, L., Time, Clocks and The Ordering of Events in a Distributed System, *Communications of the ACM*, 21(7):558-565, 1978.

[14] Lin, J., and Dunham, M. H., A Survey of Distributed Database Checkpointing, *Distributed and Parallel Databases*,5(3): 289-319, 1997.

[15] Netzer, R.H.B. and Xu, J., Necessary and Sufficient Conditions for Consistent Global Snapshots, *IEEE Transactions on Parallel and Distributed Systems*, 6(2):165-169, 1995.

[16] Pilarski, S. and Kameda, T., Checkpointing for Distributed Databases: Starting from the Basics, *IEEE Transactions on Parallel and Distributed Systems*, 3(5):602-610, 1992.

[17] Pu, C., On-the-fly, Incremental, Consistent Reading of Entire Databases, *Algorithmica*, 1(3):271-287, 1986.

[18] Pu, C., Hong, H., and J.M. Wha, Performance Evaluation of Global Reading of Entire Databases, *Proc. International Symposium on Databases in Parallel and Distributed Systems*, Austin, TX, USA, 1988, pp. 167-176.

[19] Randel, B., System Structure for Software Fault Tolerance, *IEEE Transactions on Software Engineering*, SE1(2):220-232, 1975.

[20] Salem, K. and Garcia-Molina, H., Checkpointing Memory Resident Databases, *Tech. Rep. CS-TR-126-87*, Department of Computer Science, Princeton University, December 1987.

[21] Son, S.H. and Agrawala, A.K., Distributed Checkpointing for Globally Consistent States of Databases, *IEEE Transactions on Software Engineering*, 15(10):1157-1166, 1989.

[22] Wang, Y.M. and Fuchs, W.K., Lazy Checkpoint Coordination for Bounding Rollback Propagation, *in Proc. 12th IEEE Int. Symposium on Reliable Distributed Systems*, pp. 78-85, 1993.

[23] Wang, Y.M. Consistent Global Checkpoints That Contain a Given Set of Local Checkpoints. *IEEE Transactions on Computers*, 46(4):456-468, 1997.

Appendix

This appendix provides the proof of Theorem 4.1 in Section 4.2. The proof consists in showing that Theorem 4.1 is an instantiation of Theorem 3.2 in [1].

At this aim let us introduce the notion of intervals of data object states and a dependency relation between intervals:

Definition 7.1 *(Interval)*

An interval I_x^i consists of all the local states σ_x^k such that:

$$(\sigma_x^k = C_x^i) \vee (C_x^i <_{LS} \sigma_x^k <_{LS} C_x^{i+1})$$

In other words, an interval is defined based on successive checkpoints of the state of a data object.

Definition 7.2 *(Relation $\prec$)*

I_x^i precedes I_y^j (denoted $I_x^i \prec I_y^j$), if and only if:

1. *$x = y$ and $j = i + 1$; or*
2. *$\sigma_x^s <_{LS} \sigma_y^t$ and $\sigma_x^s \in I_x^i$ and $\sigma_y^t \in I_y^j$; or*
3. *$\exists I_z^u$: $I_x^i \prec I_z^u \wedge I_z^u \prec I_y^j$.*

Using Definition 7.1 and Definition 7.2, the statement of Theorem 4.1 can be replaced with:

Theorem

Let $\mathcal{I} \subseteq \{1, \ldots, m\}$ and $\mathcal{S} = \{C_x^{i_x}\}_{x \in \mathcal{I}}$ be a set of data checkpoints. Then $\mathcal{S}$ is a part of a consistent global checkpoint if and only if $\forall x, y \in \mathcal{I} \Rightarrow \neg(I_x^{i_x} \prec I_y^{i_y - 1})$.

This statement is actually a particular instantiation of the statement of Theorem 3.2 in [1]. Therefore, the proof of this theorem is a particular instantiation of proof of Theorem 3.2 in [1]. That theorem provides a necessary and sufficient condition for the question $\mathcal{Q}(S)$ in a general setting which includes shared-memory and message passing models. This general setting abstracts a computation of processes, with interdependencies between local states, as a partial order based on intervals and on a precedence relation among these intervals. Intervals are defined based on successive *records* on a process and the precedence relation defines a partial order over those intervals.

This is exactly the same abstraction level we can associate with the transaction model through Definition 7.1 and Definition 7.2. Specifically, in our transaction model, data objects (with interdependences between local states) correspond to processes of the general setting. Intervals of Definition 7.1 are analogous to intervals considered in the general setting, and the relation in Definition 7.2 originates a partial order between intervals analogous to the partial order considered in the general setting.