An Adaptive Scheme For Checkpointing And Recovery In Distributed Databases With Mixed Types Of Transactions

Sang Hyuk Son

Computer Science Report No. TR-87-04
February 24, 1987

An Adaptive Scheme for Checkpointing and Recovery in Distributed Databases with Mixed Types of Transactions

Sang Hyuk Son

Department of Computer Science
Thornton Hall
University of Virginia
Charlottesville, Virginia 22903

ABSTRACT

The problem of ensuring the global consistency after system failures is considered in the context of distributed database systems. The problem is decomposed into two components: checkpointing and recovery. Recent study shows the possibility of having a checkpointing mechanism that does not interfere with the transaction processing, and yet achieves the global consistency of the checkpoints. The motivation of non-interfering checkpointing is to improve the system availability. Non-interfering checkpointing mechanisms, however, may suffer from the fact that the diverged computation needs to be maintained by the system until all of the transactions that are in progress when the checkpoint begins, come to completion. For database systems with many long-lived transactions that need long execution time, this requirement of maintaining diverged computation may make non-interfering checkpointing not practical. In this paper, we present a checkpointing algorithm that is non-interfering with transaction processing, and efficiently generates globally consistent checkpoints. It also prevents the well-known "domino effect", and saves intermediate results of the transaction, in an adaptive manner. In contrast to previous checkpointing algorithms, our checkpointing algorithm can handle effectively both short and long-lived transactions in the system.

Index Terms - distributed database, recovery, consistency, checkpoint, transaction, non-interference, availability

1. Introduction

The need for having recovery mechanisms in distributed database systems is well acknowledged. In spite of powerful database integrity checking mechanisms which detect errors and undesirable data, it is possible that some erroneous data may be included in the database. Furthermore, even with a perfect integrity checking mechanism, failures of hardware and/or software at the processing sites may destroy the consistency of the database. In order to cope with those errors and failures, distributed database systems provide recovery mechanisms, and checkpointing is a technique frequently used in such recovery mechanisms.

The goal of checkpointing in database management systems is to save a consistent state of the database on a separate secure device. In case of a failure, the stored data can be used to restore the database. Checkpointing must be performed so as to minimize both the costs of performing checkpoints and the costs of recovering the database. If the checkpoint intervals are very small, too much time and resources are spent in checkpointing; if these intervals are large, too much time is spent in recovery. Since checkpointing is an effective method for maintaining consistency of database systems, it has been widely used and studied by many researchers[1, 4, 5, 7, 8, 10, 11, 13, 17, 18].

When checkpointing is performed during normal operation of the system, the interference with transaction processing must be kept to a minimum. It is highly desirable that users are allowed to submit transactions while the checkpointing is in progress, and the transactions are executed in the system concurrently with the checkpointing process. A quick recovery from failures is also desirable to many applications of distributed databases. For achieving quick recovery, each checkpoint needs to be globally consistent so that a simple restoration of the latest checkpoint can bring the database to a consistent state. In distributed database systems these desirable properties of non-interference and global consistency make checkpointing more complicated and increase the workload of the system.

Recently, the possibility of having a checkpointing mechanism that does not interfere with the transaction processing, and yet achieves the global consistency of the checkpoints, has been studied [4, 7, 21]. The motivation of non-interfering checkpointing is to improve the system availability, that is, the system must be able to execute user transactions concurrently with the checkpointing process. The basic principle behind non-interfering checkpointing mechanisms is to create a diverged computation of the system such that the checkpointing process can view a consistent state that could result by running to completion all of the transactions that are in progress when the checkpoint begins, instead of viewing a consistent state that actually occurs by suspending further transaction execution.

Non-interfering checkpointing mechanisms, however, may suffer from the fact that the diverged computation needs to be maintained by the system until all of the transactions that are in progress when the checkpoint begins, come to completion. This may not be a major concern for a database system in which all the transactions are relatively short, and hence can be executed in a short time period. However, for database systems with many long-lived transactions that need long execution time, a non-interfering checkpointing may not be practical because of the following reasons:

(1) It takes a long time to complete one non-interfering checkpoint, resulting a high storage and processing overhead.

(2) If a crash occurs before reflecting the result of a long-lived transaction in the checkpoint, the system must re-execute the transaction from the beginning, wasting all the resources used for the initial execution of the transaction.

In this paper, we present a checkpointing algorithm which can handle effectively both short and long-lived transactions in the database system. Our checkpointing algorithm operates in two different modes: *global mode* and *local mode*. In the global mode of operation, the algorithm is non-interfering with transaction processing, and efficiently generates globally consistent checkpoints. In the local mode of operation, it prevents the well-known "domino effect", and saves intermediate results of the transaction. Furthermore, only a minimal number of processes are involved in the checkpointing. This paper is organized as follows. Section 2 introduces a model of computation used in this paper. Section 3 describes the algorithm for non-interfering checkpoint creation. Section 4 raises problems associated with non-interfering checkpoint creation, and presents an adaptive checkpointing algorithm as a possible solution. Section 5 presents an informal proof of the correctness of the algorithm. Section 6 discusses the practicality and the robustness of the algorithm, and describes the recovery methods associated with the algorithm. Section 7 concludes the paper.

2. A Model of Computation

This section introduces the model of computation used in this paper. We describe the notion of transactions and the assumptions about the effects of failures.

2.1. Data Objects and Transactions

We consider a distributed database system implemented on a computing system where several autonomous computers (called *sites)* are connected via a communication network. A database consists of a set of data objects. A data object contains a data value and represents the smallest unit of the database accessible to the user. Data objects are an abstraction; in a particular system, they may be files, pages, records, items, etc. The set of data objects in a distributed database system is partitioned among its sites.

The basic units of user activity in database systems are *transactions*. Each transaction represents a complete and correct computation, i.e., if a transaction is executed alone on an initially consistent database, it would terminate in a finite time and produce correct results, leaving the database consistent. The *read set* of a transaction T is defined as the set of data objects that T reads. Similarly, the set of data objects that T writes is called the *write set* of T. A transaction is said to be *committed* when it is executed to completion, and it is said to be *aborted* when it is not executed at all. When a transaction is committed, the output values are finalized and made available to all subsequent transactions.

We assume that each transaction has a time-stamp associated with it [12]. A time-stamp is a number that is assigned to a transaction when initiated and is kept by the transaction. Two important properties of time-stamps are (1) no two transactions have the same time-stamp, and (2) only a finite number of transactions can have a time-stamp less than that of a given transaction.

The transaction managers that have been involved in the execution of a transaction are called the *participants* of the transaction. The *coordinator* is one of the participants which initiates and terminates the transaction by controlling all other participants. In our transaction processing model, the coordinator decides on the participants using suitable decision algorithms, based on the data objects in the read set and write set of the transaction. We assume that messages are exchanged only between the coordinator and the participants: no message exchange between participants of a transaction.

We assume that the database system runs a correct transaction control mechanism (e.g., an atomic commit algorithm[19] and a concurrency control algorithm[2]), and hence assures the atomicity and serializability of transactions.

2.2. Failure Assumptions

A distributed database system can fail in many different ways, and it is almost impossible to make an algorithm which can tolerate all possible failures. In general, failures in distributed database systems can be classified as failures of *omission* or *commission* depending on whether some action required by the system specification was not taken or some action not specified was taken[14]. The simplest failures of omission are *simple crashes* in which a site simply stops running when it fails. The hardest failures are *malicious runs* in which a site continues to run, but performs incorrect actions. Most real failures lie between these two extremes.

In this paper, we do not consider failures of commission such as the "malicious runs" type of failure. When a site fails, it simply stops running (fail-stop). When the failed site recovers, the fact that it has failed is recognized, and a recovery procedure is initiated. We assume that site failures are detectable by other sites. This can be achieved either by network protocols or by high-level time-out mechanisms in the application layer[3]. We also assume that network partitioning never occurs. This assumption is reasonable for most local area networks and some long-haul networks.

3. Non-Interfering Checkpoint Creation

3.1. Motivation of Non-interference

The motivation of having a checkpointing scheme which does not interfere with transaction processing is well explained in [4] by using the analogy of migrating birds and a group of photographers. Suppose a group of photographers observe a sky filled with migrating birds. Because the scene is so vast that it cannot be captured by a single photograph, the photographers must take several snapshots and piece the snapshots together to form a picture of the overall scene. Furthermore, it is desirable that the photographers do not disturb the process that is being photographed. The snapshots cannot all be taken at precisely the same instance because of synchronization problems, and yet they should generate a ''meaningful'' composite picture.

In a distributed database system, each site saves the state of the data objects stored at it to generate a local checkpoint. We cannot ensure that the local checkpoints are saved at the same instance, unless a global clock can be accessed by all the checkpointing processes. Moreover, we cannot guarantee that the global checkpoint, consisting of local checkpoints saved, is consistent. Non-interfering checkpointing algorithms are very useful for the situations in which a quick recovery as well as no blocking of transactions is desirable. Instead of waiting for a consistent state to occur, the non-interfering checkpointing approach constructs a state that would result by completing the transactions that are in progress when the global checkpoint begins.

In order to make each checkpoint globally consistent, updates of a transaction must be either included in the checkpoint completely or not at all. To achieve this, transactions are divided into two groups according to their relations to the current checkpoint: *after-checkpoint-transactions* (ACPT) and *before-checkpoint-transactions* (BCPT). Updates belonging to BCPT are included in the current checkpoint while those belonging to ACPT are not included. In a centralized database system, it is an easy task to separate transactions for this purpose. However, it is not easy in a distributed environment. For the separation of transactions in a distributed environment, a special time-stamp which is globally agreed upon by the participating sites is used. This special time-stamp is called the *Global Checkpoint Number* (GCPN), and it is determined as the maximum of the *Local Checkpoint Numbers* (LCPN) through the coordination of all the participating sites.

An ACPT can be reclassified as a BCPT if it turns out that the transaction must be executed before the current checkpoint. This is called the *conversion* of transactions. The updates of a converted transaction are included in the current checkpoint.

3.2. The Algorithm

There are two types of processes involved in the execution of the algorithm: *checkpoint coordinator* (CC) and *checkpoint subordinate* (CS). The checkpoint coordinator starts and terminates the global checkpointing process. Once a checkpoint has started, the coordinator does not issue the next checkpoint request until the first one has terminated.

The variables used in the algorithm are as follows:

(1) *Local Clock* (LC): a clock maintained at each site which is manipulated by the clock rules of Lamport[12].

(2) *Local Checkpoint Number* (LCPN): a number determined locally for the current checkpoint.

(3) *Global Checkpoint Number* (GCPN): a globally unique number for the current checkpoint.

(4) CONVERT: a Boolean variable showing the completion of the conversion of all the eligible transactions at the site.

Our basic checkpointing algorithm, called CP1, works as follows:

(1) The checkpoint coordinator broadcasts a Checkpoint Request Message with a time-stamp LC_{CC}. The local checkpoint number of the coordinator is set to LC_{CC}. The coordinator sets the Boolean variable CONVERT to false:

$$CONVERT_{CC} := \text{false}$$

and marks all the transactions at the coordinator site with the time-stamps not greater than $LCPN_{CC}$ as BCPT.

(2) On receiving a Checkpoint Request Message, the local clock of site m is updated and $LCPN_m$ is determined by the checkpoint subordinate as follows:

$$LC_m := \max(LC_{CC} + 1, LC_m)$$

$$LCPN_m := LC_m$$

The checkpoint subordinate of site m replies to the coordinator with $LCPN_m$, and sets the Boolean variable CONVERT to false:

$$CONVERT_m := FALSE$$

and marks all the transactions at the site m with the time-stamps not greater than $LCPN_m$ as BCPT.

(3) The coordinator broadcasts the GCPN which is decided by:

$$GCPN := \max(LCPN_n) \qquad n = 1,..,N$$

(4) For all sites, after LCPN is fixed, all the transactions with the time-stamps greater than LCPN are marked as temporary ACPT. If a temporary ACPT wants to update any data objects, those data objects are copied from the database to the buffer space of the transaction. When a temporary ACPT commits, updated data objects are not stored in the database as usual, but are maintained as *committed temporary versions* (CTV) of data objects. The data manager of each site maintains the permanent and temporary versions of data objects. When a read request is made for a data object which has committed temporary versions, the value of the latest committed temporary version is returned. When a write request is made for a data object which has

committed temporary versions, another committed temporary version is created for it rather than overwriting the previous committed temporary version.

(5) When the GCPN is known, each checkpointing process compares the time-stamps of the temporary ACPT with the GCPN. Transactions that satisfy the following condition become BCPT; their updates are reflected into the database, and are included in the current checkpoint.

$$\text{LCPN} < \text{time-stamp}(T) \leq \text{GCPN}$$

The remaining temporary ACPT are treated as actual ACPT; their updates are not included in the current checkpoint. These updates are included in the database after the current checkpointing has been completed. After the conversion of all the eligible BCPT, the checkpointing process sets the Boolean variable CONVERT to true:

$$\text{CONVERT} := \text{true}$$

(6) Local checkpointing is executed by saving the state of data objects when there is no active BCPT and the variable CONVERT is true.

(7) After the execution of local checkpointing, the values of the latest committed temporary versions are used to replace the values of data objects in the actual database. Then, all committed temporary versions are deleted.

The above checkpointing algorithm essentially consists of two phases. The function of the first phase (steps 1 through 3) is the assignment of GCPN that is determined from the local clocks of the system. The second phase begins by fixing the LCPN at each site. This is necessary because each LCPN sent to the checkpoint coordinator is a candidate of the GCPN of the current checkpoint, and the committed temporary versions must be created for the data objects updated by ACPT. The notions of committed temporary versions and conversion from ACPT to BCPT are introduced to assure that each checkpoint contains all the updates made by transactions with earlier time-stamps than the GCPN of the checkpoint.

When a participant receives a Transaction Initiation Message from the coordinator, it checks whether or not the transaction can be executed at this time. If the checkpointing process has already executed step 5 and time-stamp(T) ≤ GCPN, then a reject message is returned, and the transaction is aborted. Therefore in order to execute step 6, each checkpointing process only needs to check active BCPT at its own site, and yet the consistency of the checkpoint can be achieved.

3.3. Termination of the Algorithm

The algorithm described so far has no restriction on the method of arranging the execution order of transactions. With no restriction, however, it is possible that the algorithm may never terminate. In order to ensure that the algorithm terminates in a finite time, we must ensure that all BCPT terminate in a finite time, because local checkpointing in step 6 can occur only when there is no active BCPT at the site.

Termination of transactions in a finite time is ensured if the concurrency control mechanism gives priority to older transactions over younger transactions. With such a time-based priority, it is guaranteed that once a transaction T_i is initiated, then T_i is never blocked by subsequent transactions that are younger than T_i. The number of transactions that may block the execution of T_i is finite because only a finite number of transactions can be older than T_i. Among older transactions which may block T_i, there must be the oldest transaction which will terminate in a finite time, since no other transaction can block it. When it terminates, the second oldest transaction can be executed, and then the third, and so on. Therefore, T_i will be executed in a finite time. Since we have a finite number of BCPT when the checkpointing is initiated, all of them will terminate in a finite time, and hence the checkpointing itself will terminate in a finite time. Concurrency control mechanisms based on time-stamp ordering as in [2, 20] can ensure the termination of transactions in a finite time.

4. Adaptive Checkpoint Creation

In the previous section, we have shown that the algorithm will terminate in a finite time by selecting appropriate concurrency control mechanisms. However, the amount of time necessary to complete one checkpoint cannot be bound in advance; it depends on the execution time of the longest transaction classified as a BCPT. It implies that the storage and processing cost of the checkpointing algorithm may become unacceptably high if a long-lived transaction is included in the set of BCPT. We discuss the practicality of the non-interfering checkpoints in Section 6. In addition to that, all the resources used for the execution of such a long-lived transaction would be wasted if the transaction must be re-executed from the beginning due to system failures.

In this section, we extend our checkpointing algorithm CP1 to solve these problems. We assume that each transaction must carry the mark with it, when initiated, which tells whether it is a normal transaction or a long-lived transaction. The threshold to separate two types of transactions is application-dependent. In general, transactions that need hours of execution can be considered as long-lived transactions.

The new checkpointing algorithm, called CP2, operates in two different modes: *global mode* and *local mode*. The global mode operation of CP2 is basically the same as CP1, and it will efficiently generate consistent checkpoints in a non-interfering manner. In the local mode of operation, CP2 provides a mechanism to save consistent states of a transaction so that the transaction can resume execution from its most recent checkpoint.

As in the algorithm CP1, the checkpoint coordinator begins the algorithm CP2 by sending out Checkpoint Request Messages. Upon receiving this request message, each site checks whether any long-lived transaction is being executed at the site. If yes, the site reports it to the coordinator, instead of sending LCPN. Otherwise (i.e., no long-lived transaction in the system), CP2 continues the same procedure as CP1. If any site reports the existence of long-lived transaction, the coordinator switches to the local mode of operation, and informs each site to operate in the local mode. The checkpoint coordinator sends Checkpoint Request Messages to each site at an appropriate time interval to initiate the next checkpoint in the global mode. This attempt will succeed if there is no active long-lived transactions in the system.

In the local mode of operation, each long-lived transaction is checkpointed separately from other long-lived transactions. The coordinator of the long-lived transaction initiates the checkpoint by sending Checkpoint Request Messages to its participants. A checkpoint at each site saves a local state of a long-lived transaction. For satisfying the correctness requirement, a set of checkpoints, one per each participating site of a global long-lived transaction, should reflect the consistent state of the transaction. Inconsistent set of checkpoints may result by non-synchronized execution of associated checkpoint. For example, consider a long-lived transaction T being executed at sites P and Q, and a checkpoint taken at site P at time X, and at site Q at time Y. If a message M is sent from P after X, and received at Q before Y, then the checkpoints would save the reception of M but not the sending of M, resulting in a checkpoint representing an inconsistent state of T.

We use message numbers for achieving consistency in a set of local checkpoints of a long-lived transaction. Messages that are exchanged by the participating transaction managers of a long-lived transaction contains a message number tag. Transaction managers of a long-lived transaction use monotonically increasing numbers in the tag of its outgoing messages, and it maintains the tag numbers of the last messages it received from other participants. On receiving a checkpoint request, a participant compares the message number attached to the request message with the tag number it received last from the coordinator. The participant replies OK to the coordinator and executes local checkpointing only if the request tag number is not less than the number it has maintained. Otherwise, it reports to the coordinator that the checkpointing cannot be executed with that request message.

If all the replies from the participants arrive and are all OK, the coordinator decides to make all the local checkpoints permanent. Otherwise, the decision is to discard the current checkpoint, and to initiate a new checkpoint. This decision is delivered to all participants. After a new permanent checkpoint is taken, any previous checkpoints will be discarded at each site.

5. Consistency of Global Checkpoints

In this section we give an informal proof of the correctness of the algorithm. We show that each mode of operation satisfies the requirement of correctness. Although the consistency is our correctness criteria for the checkpointing algorithm, the unit for consistency is different for different mode of operation; a transaction is the unit of consistency in the global mode, while an event of a transaction is the unit of consistency in the local mode. We first show the consistency in the global mode.

5.1. Consistency in Global Mode

In addition to proving the consistency of the checkpoints generated by the algorithm in the global mode, we show that the algorithm has another desirable property that each checkpoint contains all the updates of transactions with earlier time-stamps than its GCPN. This property reduces the work required in the actual recovery, which is discussed in Section 6. A longer and more thorough discussion on the correctness of the algorithm is given in [21].

The properties of the algorithm we want to show are

(1) a set of all local checkpoints with the same GCPN represents a consistent database state, and

(2) all the updates of the committed transactions with earlier time-stamps than the GCPN are reflected in the current checkpoint.

Note that only one checkpointing process can be active at a time because the checkpointing coordinator is not allowed to issue another checkpointing request before the termination of the previous one.

A database state is consistent if the set of data objects satisfies the consistency constraints[6]. Since a transaction is the unit of consistency, a database state S is consistent if the following holds:

(1) For each transaction T, S contains all subtransactions of T or it contains none of them.

(2) If T is contained in S, then each predecessor T' of T is also contained in S. (T' is a predecessor of T if it modified the data object which T accessed at some later point in time.)

For a set of local checkpoints to be globally consistent, all the local checkpoints with the same GCPN must be consistent with each other concerning the updates of transactions that are executed before and after the checkpoint. Therefore, to prove that the algorithm satisfies both properties, it is sufficient to show that the updates of a global transaction T are included in CP_i at each participating site of T, if and only if $\text{time-stamp}(T) \leq \text{GCPN}(CP_i)$. This is enforced by the mechanism to determine the value of the GCPN, and by the conversion of the temporary ACPT into BCPT.

A transaction is said to be *reflected* in data objects if the values of data objects represent the updates made by the transaction. We assume that the database system provides a reliable mechanism for writing into the secondary storage such that a writing operation of a transaction is atomic and always successful when the transaction commits. Because updates of a transaction are reflected in the database only after the transaction has been successfully executed and committed, partial results of transactions cannot be included in checkpoints.

The checkpointing algorithm assures that the sequence of actions are executed in some specific order. At each site, conversion of eligible transactions occurs after the GCPN is known, and local checkpointing cannot start before the Boolean variable CONVERT becomes true. CONVERT is set to false at each site after it determines the LCPN, and it becomes true only after the conversion of all the eligible transactions. Thus, it is not possible for a local checkpoint to save the state of the database in which some of the eligible transactions are not reflected because they remain unconverted.

We can show that a transaction becomes BCPT if and only if its time-stamp is not greater than the current GCPN. This implies that all the eligible BCPT will become BCPT before local checkpointing begins in step 6. Therefore, updates of all BCPT are reflected in the current checkpoint.

From the atomic property of transactions provided by the transaction control mechanism (e.g. commit protocol in [19]), it can be assured that if a transaction is committed at a participating site then it is committed at all other participating sites. Therefore if a transaction is committed at one site, and if it satisfies the time-stamp condition above, its updates are reflected in the database and also in the current checkpoint at all the participating sites.

5.2. Consistency in Local Mode

In order to prove that the algorithm CP2 is correct in the local mode of operation, we need to show that a set of local checkpoints always represents a consistent state of the transaction that is checkpointed. In other words, t is sufficient to show that if the set of local checkpoints is consistent before the execution of CP2, the set of checkpoints is also consistent after the completion of CP2.

Since the initiation point of a transaction is consistent, the system have at least one set of consistent checkpoints of a transaction (i.e., the initiation point). Therefore, if CP2 does not generate a new set of checkpoints upon its termination, the system has the previous checkpoint which is consistent.

Without loss of generality, assume a new set of checkpoints is taken by CP2. We show by contradiction that the set of checkpoints after the termination of CP2 is consistent. Suppose it is not consistent. Then there are two transaction managers P and Q such that P sent Q a message M after making its checkpoint, and Q received M before making its checkpoint. Consider the case that P is the coordinator. Upon receiving a request message from the coordinator, Q must have sent OK because Q could not have made its checkpoint permanent otherwise. It implies that the tag number of the request message is greater than those of messages Q has received, a contradiction. If Q is the coordinator, P cannot start local checkpointing before receiving a request message from Q. Since Q sent the request message after receiving M, P must have received it after it sent M, a contradiction.

6. Discussion

The desirable properties of non-interference and global consistency not only make the checkpointing more complicated in distributed database systems, but also increase the workload of the system. It may turn out that the overhead of the checkpointing mechanism is unacceptably high, in which case the mechanism should be abandoned in spite of its desirable properties. The practicality of non-interfering checkpointing, therefore, depends partially on the amount of extra workload incurred by the checkpointing mechanism. In this section we consider practicality of non-interfering checkpointing algorithms, and discuss the robustness and recovery methods associated with the algorithm CP2.

6.1. Practicality of Non-interfering Checkpoints

There are two performance measures that can be used in discussing the practicality of non-interfering checkpointing: extra storage and extra workload required. The extra storage requirement of the algorithm is simply the CTV file size, which is a function of the expected number of ACPT of the site, the number of data objects updated by a typical transaction, and the size of the basic unit of information:

$$\text{CTV file size} = N_A \times (\text{number of updates}) \times (\text{size of the data object})$$

where N_A is the expected number of ACPT of the site.

The size of the CTV file may become unacceptably large if N_A or the number of updates becomes very large. Unfortunately, they are determined dynamically from the characteristics of transactions submitted to the database system, and hence cannot be controlled. Since N_A is proportional to the execution time of the longest BCPT at the site, it would become unacceptably large if a long-lived transaction is being executed when a checkpoint begins at the site. The only parameter we can change in order to reduce the CTV file size is the granularity of a data object. The size of the CTV file can be minimized if we minimize the size of the data object. By doing so, however, the overhead of normal transaction processing (e.g., locking and unlocking, deadlock detection, etc) will be increased. Also, there is a trade-off between the degree of concurrency and the lock granularity[16]. Therefore the granularity of a data object should be determined carefully by considering all such trade-offs, and we cannot minimize the size of the CTV file by simply minimizing the data object granularity.

There is no extra storage requirement in intrusive checkpointing mechanisms[1, 10, 17]. However this property is balanced by the cases in which the system must block ACPT or abort half-way done global transactions because of the checkpointing process.

The extra workload imposed by the algorithm mainly consists of the workload for (1) determining the GCPN, (2) committing ACPT (move data objects to the CTV file), (3) reflecting the CTV file (move committed temporary versions from the CTV file to the database), and (4) making the CTV file clear when the reflect operation is finished. Among these, workload for (2) and (3) dominates others. As in extra storage estimation, they are determined by the number of ACPT and the number of updates. Therefore, as far as the values of these variables can be maintained within a certain threshold level, non-interfering checkpointing would not severely degrade the performance of the system. A detailed discussion on the practicality of non-interfering checkpointing is given in [21].

6.2. Site Failures

So far, we assumed that no failure occurs during checkpointing. This assumption can be justified if the probability of failures during a single checkpoint is extremely small. However, it is not always the case, and we now consider the method to make the algorithm resilient to failures.

During the global mode of operation, the algorithm CP2 is insensitive to failures of subordinates. If a subordinate fails before the broadcast of a Checkpoint Request Message, it is excluded from the next checkpoint. If a subordinate does not send its LCPN to the coordinator, it is excluded from the current checkpoint. When the site recovers from the failure, the recovery manager of the site must find out the GCPN of the latest checkpoint. After receiving information of transactions which must be executed for recovery, the recovery manager brings the database up to date by executing all the transactions whose time-stamps are not greater than the latest GCPN. Other transactions are executed after the state of the data objects at the site is saved by the checkpointing process.

An atomic commit protocol guarantees that a transaction is aborted if any participant fails before it sends a Precommit message to the coordinator. Therefore, site failures during the execution of the algorithm cannot affect the consistency of checkpoints because each checkpoint reflects only the updates of committed BCPT.

In the local mode of operation, a failure of a participant prevents the coordinator from receiving OK from all the participants, or prevents the participants from receiving the decision message from the coordinator. However, because a transaction is aborted by an atomic commit protocol, it is not necessary to make checkpointing robust to failures of participants.

The algorithm is, however, sensitive to failures of the coordinator. In particular, if the coordinator crashes during the first phase of the global mode of operation (i.e., before the GCPN message is sent to subordinates), every transactions become ACPT, requiring too much storage for committed temporary versions.

One possible solution to this involves the use of a number of *backup* processes; these are processes that can assume responsibility for completing the coordinator's activity in the event of its failure. These backup processes are in fact checkpointing subordinates. If the coordinator fails before it broadcasts the GCPN message, one of the backups takes the control. A similar mechanism is used in SDD-1 [9] for reliable commitment of transactions. Proper coordination among the backup processes is crucial here. In the event of the failure of the coordinator, one, and only one backup process has to assume the control. The algorithm for accomplishing this assumes an ordering among the backup processes, designated in order as $p_1, p_2, ..., p_n$. Process p_{k-1} is referred to as the *predecessor* of process p_k (for $k > 0$), and the coordinator is taken as the predecessor of process p_1.

We assume that the network service enables processes to be informed when a given site achieves a specified status (simply UP or DOWN in this case). Initially, each of the backup processes checks the failure of its predecessor. Then the following rules are used.

(1) If the predecessor is found to be down, then the process begins to check the predecessor of the failed process.

(2) If the coordinator is found to be down, the first backup process assumes the control of checkpointing.

(3) If a backup process recovers, it ceases to be a part of the current checkpointing.

(4) After each checkpoint, the list of backup processes is adjusted by including all the UP sites.

These rules guarantee that at most one process, either the coordinator or one of the backup processes, will be in control at any given time. Thus a checkpointing will terminate in a finite time once it begins.

6.3. Recovery

The recovery from site crashes is called the *site recovery*. The complexity of the site recovery varies in distributed database systems according to the failure situation[17]. If the crashed site has no replicated data objects and if all the recovery information is available at the crashed site, local recovery is enough. Global recovery is necessary because of failures which require the global database to be restored to some earlier consistent state. For instance, if the transaction log is partially destroyed at the crashed site, local recovery cannot be executed to completion.

When a global recovery is required, the database system has two alternatives: a *fast* recovery and a *complete* recovery. A fast recovery is a simple restoration of the latest global checkpoint. Since each checkpoint generated by the algorithm is globally consistent, the restored state of the database is assured to be consistent. However, all the transactions committed during the time interval from the latest checkpoint until the time of crash would be lost. A complete recovery is performed to restore as many transactions that can be redone as possible. The trade-offs between the two recovery methods are the recovery time and the number of transactions saved by the recovery.

Quick recovery from failures is critical for some applications of distributed database systems which require high availability (e.g., ballistic missile defense or air traffic control). For those applications, the fate of the mission, or even the lives of human beings, may depend on the correct values of the data and the accessibility to it. Availability of a consistent state is of primary concern for them, not the most up-to-date consistent state. If a simple restoration of the latest checkpoint could bring the database to a consistent state, it may not be worthwhile to spend time in recovery by executing a complete recovery to recover some of the transactions.

For the applications in which each committed transaction is so important that the most up-to-date consistent state of the database is highly desirable, or if the checkpoint intervals are large such that a lot of transactions may be lost by the fast recovery, a complete recovery is appropriate to use. The cost of a complete recovery is the increased recovery time which reduces the availability of the database. Searching through the transaction log is necessary for a complete recovery. The second property of the algorithm (i.e., each checkpoint reflects all the updates of transactions with earlier time-stamps than its GCPN) is useful in reducing the amount of searching because the set of transactions whose updates must be redone can be determined by the simple comparison of the time-stamps of transactions with the GCPN of the checkpoint. Complete recovery mechanisms based on the special time-stamp of checkpoints (e.g., GCPN) have been proposed in [11, 22].

After the site recovery is completed using either a fast recovery procedure or a complete recovery procedure, the recovering site checks whether it has completed a local mode checkpointing for any long-lived transactions. If any local mode checkpoints are found, those transactions can be restarted from the saved checkpoints. In this case, the coordinator of the transaction requests all the participants to restart from their checkpoints if and only if they all are able to restart from that checkpoint. The coordinator makes a decision whether to restart the transaction from the checkpoint or from the beginning based on the responses from the participants, and sends the decision message to all the participants. We provide such a two-phase recovery protocol in order to maintain the consistency of the database in case of damaged checkpoints at the failure site. A transaction will be restarted from the beginning if any participant is not able to restore the checkpointed state of the transaction for any reason.

7. Concluding Remarks

During normal operation of the database system, checkpointing is performed to save information necessary for recovery from a failure. For better recoverability and availability of distributed database systems, checkpointing must be able to generate a globally consistent database state, without interfering with transaction processing. Site autonomy in distributed database systems makes the checkpointing more complicated than in centralized database systems. Also, long-lived transactions may substantially increase the overhead associated with non-interfering checkpointing, and make it unacceptable in many applications of the distributed systems. In this paper, a new checkpointing algorithm for distributed database systems is presented and discussed. The correctness of the algorithm is shown, and the robustness of the algorithm and recovery procedures associated with it are discussed. For the applications in which the system must execute a mixture of short and long-lived transactions, and the ability of continuous processing of transactions is so critical that the blocking of transaction activity for checkpointing is not feasible, we believe that the algorithm presented in this paper provides a practical solution to the problem of checkpointing and recovery in distributed database systems.

REFERENCES

[1] Attar, R., Bernstein, P. A. and Goodman, N., Site Initialization, Recovery, and Backup in a Distributed Database System, IEEE Trans. on Software Engineering, November 1984, pp 645-650.

[2] Bernstein, P., Goodman N., Concurrency Control in Distributed Database Systems, ACM Computing Surveys, June 1981, pp 185-222.

[3] Bernstein, P., Goodman, N., An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases, ACM Trans. on Database Systems, Dec. 1984, pp 596-615.

[4] Chandy, K. M., Lamport, L., Distributed Snapshots: Determining Global States of Distributed Systems, ACM Trans. on Computer Systems, February 1985, pp 63-75.

[5] Dadam, P. and Schlageter, G., Recovery in Distributed Databases Based on Non-synchronized Local Checkpoints, Information Processing 80, North-Holland Publishing Company, Amsterdam, 1980, pp 457-462.

[6] Eswaran, K. P. et al, The Notion of Consistency and Predicate Locks in a Database System, Commun. of ACM, Nov. 1976, pp 624-633.

[7] Fischer, M. J., Griffeth, N. D. and Lynch, N. A., Global States of a Distributed System, IEEE Trans. on Software Engineering, May 1982, pp 198-202.

[8] Gelenbe, E., On the Optimum Checkpoint Interval, JACM, April 1979, pp 259-270.

[9] Hammer, M. and Shipman, D., Reliability Mechanisms for SDD-1: A System for Distributed Databases, ACM Trans. on Database Systems, December 1980, pp 431-466.

[10] Jouve, M., Reliability Aspects in a Distributed Database Management System, Proc. of AICA, 1977, pp 199-209.

[11] Kuss, H., On Totally Ordering Checkpoints in Distributed Databases, Proc. ACM SIGMOD, 1982, pp 293-302.

[12] Lamport, L., Time, Clocks and Ordering of Events in Distributed Systems, Commun. ACM, July 1978, pp 558-565.

[13] McDermid, J., Checkpointing and Error Recovery in Distributed Systems, Proc. 2nd International Conference on Distributed Computing Systems, April 1981, pp 271-282.

[14] Mohan, C., Strong, R., and Finkelstein, S., Method for Distributed Transaction Commit and Recovery Using Byzantine Agreement Within Clusters of Processors, Proc. 2nd ACM SIGACT/SIGOPS Symposium on Principles of Distributed Computing, August 1983.

[15] Ricart, G. and Agrawala, A. K., An Optimal Algorithm for Mutual Exclusion in Computer Networks, Commun. of ACM, Jan. 1981, pp 9-17.

[16] Ries, D., The Effect of Concurrency Control on The Performance of A Distributed Data Management System, 4th Berkeley Conference on Distributed Data Management and Computer Networks, Aug. 1979, pp 221-234.

[17] Schlageter, G. and Dadam, P., Reconstruction of Consistent Global States in Distributed Databases, International Symposium on Distributed Databases, North-Holland Publishing Company, INRIA, 1980, pp 191-200.

[18] Shin, K. G., Lin, T.-H., Lee, Y.-H., Optimal Checkpointing of Real-Time Tasks, 5th Symposium on Reliability in Distributed Software and Database Systems, January 1986, pp 151-158.

[19] Skeen, D., Nonblocking Commit Protocols, Proc. ACM SIGMOD International Conference on Management of Data, 1981, pp 133-142.

[20] Son, S. H. and Agrawala, A. K., A Token-Based Resiliency Control Scheme in Replicated Database Systems, 5th Symposium on Reliability in Distributed Software and Database Systems, January 1986, pp 199-206.

[21] Son, S. H., On Reliability Mechanisms in Distributed Database Systems, (Ph.D. Dissertation), Technical Report TR-1614, Dept. of Computer Science, University of Maryland, College Park, January 1986

[22] Son, S. H. and Agrawala, A. K., An Algorithm for Database Reconstruction in Distributed Environments, 6th International Conference on Distributed Computing Systems, Cambridge, Massachusetts, May 1986, pp 532-539.