EFFICIENT DECENTRALIZED CHECKPOINTING IN DISTRIBUTED DATABASE SYSTEMS

Sang Hyuk Son

Computer Science Report No. TR-86-25
October 17, 1986

This work was partially supported by the Office of Naval Research under contract no. N00014-86-K-0245 to the Department of Computer Science, University of Virginia, Charlottesville, VA.

ABSTRACT

The goal of checkpointing in database management systems is to save database states on a separate secure device so that the database can be recovered when errors and failures occur. Recently, the possibility of having a checkpointing mechanism which does not interfere with the transaction processing has been studied[5, 8, 23]. Users are allowed to submit transactions while the checkpointing is in progress, and the transactions are performed in the system concurrently with the checkpointing process. This property of non-interference is highly desirable to real-time applications, where restricting transaction activity during the checkpointing operation is in many cases not feasible. In this paper, we present a decentralized algorithm for non-interfering checkpointing in distributed database systems, and prove its correctness.

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[2, 5, 6, 8, 10, 11, 20, 21, 23].

In order to achieve the goal of efficient database system recoverability, it is necessary to consider issues such as global consistency, non-interference, and communication overhead, when a checkpointing mechanism is designed for a distributed database system. The need and the desirability of these properties is self evident. For example, even though an inconsistent checkpoint may be easy and inexpensive to obtain, it may require a lot of additional work to recover a consistent state of the database. Some of the checkpointing schemes appearing in the literature (e.g. [6]) do not meet this criteria.

A quick recovery from failures is desirable in many applications of distributed databases that require high availability. 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, this property of global con-

sistency makes the checkpointing more complicated. In order to make each checkpoint globally consistent, updates of a transaction must be either included completely in one checkpoint, or not included at all. A straightforward method of assuring this is to suspend further processing of transactions so that all sites can reach a state of inactivity in which no transaction is active, before writing the local checkpoint. After the checkpointing has been completed, normal processing of transactions can be resumed. However, restricting transaction activity during the checkpointing operation is undesirable, and in many cases not feasible, depending on the availability constraints imposed on the system.

Traditional checkpointing in distributed database systems can be classified into three categories according to the coordination necessary among the autonomous sites. These are (1) fully synchronized[10], (2) loosely synchronized[20], and (3) nonsynchronized[6]. Fully synchronized checkpointing is done only when there is no active transaction in the database system. In this scheme, before writing a local checkpoint, all sites must have reached a state of inactivity. In a loosely synchronized system, each site is not compelled to write its local checkpoint in the same global interval of time. Instead, each site can choose the point of time to stop processing and take the checkpoint. A distinguished site locally manages a checkpoint sequence number and broadcasts it for the creation of a checkpoint. Each site takes the local checkpoint as soon as it is possible, and then resumes normal transaction processing. It is then the responsibility of the local transaction managers to guarantee that all transactions run in the local checkpoint intervals bounded by checkpoints with the same sequence numbers. In nonsynchronized checkpointing, global coordination with respect to the checkpointing does not take place at all. It is a decentralized approach in that each site is independent from all others with respect to the frequency of checkpointing and the time instants when local checkpoints are saved. However, a logically consistent database state is not constructed until a global reconstruction of the database is required.

In [2], a backup database is created by pretending that the backup database is a new site being added to the system. An initialization algorithm is executed to bring the new site up-to-date.

One of the drawbacks common to the checkpointing schemes above is that transaction processing must be stopped for checkpointing. Maintaining transaction inactivity for the duration of the checkpointing operation is not feasible for many applications of distributed database systems.

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. In [8], an approach for checkpointing, based on a formal model of asynchronous parallel processes and an abstract distributed transaction system, is proposed. It is called *non-intrusive* in the sense that no operations of the underlying system need be halted while the global checkpoint is being executed. The non-intrusive checkpointing approach as suggested in [8] describes the behavior of an abstract system and does not provide a practical procedure for obtaining a checkpoint.

One way of achieving both properties of non-interference and global consistency is that, for each checkpoint, a centralized control process makes a decision on whether or not to include updates of a transaction in it. The algorithm proposed in [23] follows this approach.

Algorithms with fully distributed control have been claimed to be more reliable than those with centralized control, because even if one control site fails, it might be possible to continue the tasks in progress by using other control sites. One of the problems associated with the distributed control is the number of control messages that must be exchanged among the participating sites; $O(N^2)$ messages are to be sent in one round of message exchange, where N is the number of participants.

In this paper, we propose a decentralized checkpointing algorithm which is non-interfering and which efficiently generates globally consistent checkpoints. The algorithm provides a practical procedure for non-interfering checkpointing in distributed environments, through efficient implementation of the abstract idea of non-intrusiveness. The algorithm

constructs globally consistent checkpoints, and yet the interference of it with the transaction processing is greatly reduced. Perfect non-interference can be achieved by the algorithm if messages are delivered in the order they are sent. The notion of diverged computation in [8] is captured in the "committed temporary versions" of data objects in our algorithm. The algorithm needs only $O(N\sqrt{N})$ messages in one round of message exchange, by using the communication structure based on finite projective planes[1]. This paper is organized as follows. Section 2 introduces a model of computation used in this paper. Section 3 presents the communication structure used by the algorithm. Section 4 describes the checkpointing algorithm. Section 5 presents an informal proof of the correctness of the algorithm. Section 6 discusses 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

A *database* consists of a set of data objects. Each data object has a *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. All user requests for access to the database are handled by the *database system*. We consider a distributed database system implemented on a computing system where several autonomous computers (called *sites*) are connected via a communication network. The set of data objects in a distributed database system is partitioned among its sites. A database is said to be *consistent* if the values of data objects satisfy a set of assertions. The assertions that characterize the consistent states of the database are called the *consistency constraints* [7].

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. A transaction is the unit of consistency and hence, it must be *atomic*. By atomic, we mean that intermediate states of the database must not be visible outside the transaction, and every updates of a transaction must be executed in an all-or-nothing fashion. 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 the database system runs a correct transaction control mechanism (e.g., atomic commit algorithm[22] and concurrency control algorithm[3]), and hence assures the atomicity and the serializability of transactions.

Each transaction has a time-stamp associated with it [14]. 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, we assume that the coordinator decides on the participants using suitable decision algorithms, based on the data objects the transaction reads and writes. The coordinator creates and sends a Transaction Initiating Message (TIM) to each participants. A TIM contains the definition of the transaction, including the list of participants, the objects to be accessed, and the time-stamp.

All participants that receive a TIM and are able to execute it reply with a TIM-ACK message to the coordinator. The other sites send a TIM-NACK message indicating that the transaction cannot be executed at this time. The coordinator waits for a response from all of the participants. If they are all TIM-ACKs then it sends a Start Transaction Message

(STM). The transaction is started at a participating site only after it has received the STM. One TIM-NACK message is enough to reject the transaction. In that case, the coordinator sends a Reject message to each participants, and the transaction is rejected.

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[16]. 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[4]. We also assume that network partitioning never occurs. This assumption is reasonable for most local area networks and some long-haul networks.

3. Communication Structure

In a decentralized algorithm, each site is associated with it a set of other sites with which it communicates. In many cases, this set consists of all other sites in the system. The cardinality, membership, and intersection properties of these sets affect the message complexity and the number of rounds of message exchange of the algorithm[13]. The communication structure discussed in this section aims to reduce the number of messages by requiring each site to communicate only with a subset of the set of sites in the system.

For correctness of the algorithm, each site must be capable of obtaining the necessary information from any other sites. Therefore, the intersection of the sets associated with the different sites must be non-null. One possible method of constructing sets with this property is to use a finite projective plane, which consists of a finite collection of points and lines that satisfy the following postulates:

P1: Two distinct points lie on one and only one common line.

P2: Two distinct lines pass through one and only one common point.

P3: There are four distinct points, no three of which lie on the same line.

Postulate P3 is necessary to eliminate certain degenerate finite projective planes such as a set of points and a single line. Construction of finite projective planes is discussed in [19], and a family of decentralized commit protocols that uses finite projective planes is developed in [12, 13]. In [15], sets satisfying similar properties are used in the context of mutual exclusion.

The key to our communication structure is the construction of N subsets of the set of sites in the system. The subsets must satisfy constraints on the cardinality, membership, and intersections among them. Let N be the number of sites in the system, and S be a set of subsets of them. It has been shown in [13] that a set of subsets can be constructed which satisfies the following constraints:

(1) $|S| = N$.

(2) $|S_i| = m$, where $S_i \in S$, $1 \leqslant i \leqslant N$.

(3) Each site is a member of exactly m subsets.

(4) The intersection graph of S is connected.

Consider a site i. It belongs to S_i and m-1 other subsets. At each round of message exchange, a site i sends messages only to other sites in S_i and to the m-1 sites of other subsets to which i belongs. We call them the *communication group* of the site. Messages are exchanged only within a communication group, and hence at most $2\times(m-1)$ messages are

sent by a site. Since $m = O(\sqrt{N})$ by the properties of the projective planes[1], only $O(N\sqrt{N})$, instead of $O(N^2)$, messages are sent at each round of message exchange.

A projective plane may not exist for the given N. It happens when m is not the power of a prime (i.e., $m \neq P^k$ for P prime and k a positive integer). In this case, several virtual sites have to be added to the system for the communication structure to be used. These virtual sites execute the algorithm identical to actual sites except that they should have initial value zero to exchange, and no data objects are stored at those sites. It can be easily shown that, despite the addition of virtual sites, only $O(N\sqrt{N})$ messages are sent at each round of message exchange[12].

4. An Algorithm for Non-Interfering Checkpoints

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.

For each checkpoint to be 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.

4.1. The Algorithm

The checkpointing procedure begins when certain conditions for generating the next checkpoint are satisfied. We assume that a special site (called *checkpoint initiator*, or CI) has the responsibility for checking the conditions for the next checkpoint generation (e.g., the number of transactions executed), and initiates the global checkpointing procedure when the conditions are satisfied. Once a checkpoint has started, the initiator 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[14].

(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 checkpointing algorithm works as follows:

(1) The initiator sends a Checkpoint Request Message with a time-stamp LC_{CI} to its communication group. The local checkpoint number of the initiator is set to LC_{CI}. The initiator sets the Boolean variable CONVERT to false:

$CONVERT_{CI} := false$

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

(2) On receiving a Checkpoint Request Message, the local clock of site n is updated and $LCPN_n$ is determined as follows:

$LC_n := max(LC_{CI} + 1, LC_n)$

$LCPN_n := LC_n$

The checkpoint process of site n sends $LCPN_n$ to its communication group, and sets the Boolean variable CONVERT to false:

$CONVERT_n := false$

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

(3) When $LCPN_i$ is received from each of its communication group, $LCPN_n$ is updated as follows:

$LCPN_n := max(LCPN_i, LCPN_n)$

(4) After the reception of the $LCPN_i$ from all members of its communication group, the checkpoint process sends $LCPN_n^f$ to its communication group, notifying that this is the final value of $LCPN_n$.

(5) When all the $LCPN_i^f$ have been received from each of its communication group, the GCPN is determined as follows:

$GCPN := max(LCPN_i^f)$

(6) 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.

(7) 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.

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

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:

$$CONVERT := true$$

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

(9) 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 5) 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 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 site receives a Transaction Initiation Message, the transaction manager checks whether or not the transaction can be executed at this time. If the checkpointing process has already executed step 7 and time-stamp(T) $\leqslant$ GCPN, then a TIM-NACK message is returned. Therefore in order to execute step 8, each checkpointing process only needs to check active BCPT at its own site, and yet the consistency of the checkpoint can be achieved.

4.2. 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 8 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 by sending Start Transaction Messages, 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 [3, 24] can ensure the termination of transactions in a finite time.

5. Consistency of Global Checkpoints

In this section we give an informal proof of the correctness of the algorithm. In addition to proving the consistency of the checkpoints generated by the algorithm, we show that the algorithm has another nice 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 7. A longer and more thorough discussion on the correctness of the algorithm is given in [25].

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[7]. 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 time-stamp(T) $\leqslant$ 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.

Consider the GCPN first. It can be shown that when the GCPN is determined at step 5, it is the maximum of all the LCPN in the system by the following arguments:

(1) Let the maximum of the LCPN in the system be the one at site j ($LCPN_j$). Let site i have the GCPN different from $LCPN_j$ by the end of step 5.

(2) If j is a member of the subset S_i, i must have received $LCPN_j$ in step 3, and therefore $LCPN_i^f$ must be $LCPN_j$. The maximum determined in step 5 must again be $LCPN_j$, which contradicts the assumption.

(3) If j is not a member of S_i, then j must belong to S_k for some k which is a member of the communication group of i. In step 3, j must send its LCPN to k, and hence $LCPN_k^f$ at step 4 must be $LCPN_j$. In step 4, k sends this value to i. Therefore, GCPN determined at i in step 5 must be $LCPN_j$, which contradicts the assumption.

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 8. 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 [22]), 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.

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 and the robustness of the proposed algorithm, and present recovery methods associated with the algorithm.

6.1. Practicality of the Algorithm

There are two performance measures that can be used in discussing the practicality of the proposed algorithm: 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:

CTV file size = N_A×(number of updates)×(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 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. 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[18]. 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[2, 10, 20]. 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, the proposed algorithm would not severely degrade the performance of the system. A more detailed discussion on the practicality of non-interfering checkpointing is given in [27].

6.2. Site Failures

So far, we assumed that no failure occurs during a checkpoint. 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.

The algorithm is insensitive to the failure of site i once it has sent $LCPN_i^f$ to its communication group. If the site fails before sending out $LCPN_i^f$, each member of its communication group must communicate with the members of S_i to find out the values of their LCPN. Because the algorithm is fully distributed, each site can take its local checkpoint without further coordination once GCPN of the current checkpoint is known. 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.

The algorithm is, however, sensitive to failures of the initiator. In particular, if the initiator crashes before the broadcast of a Checkpoint Request Message, none will initiate the next checkpoint. Too many transactions should be redone, or may even be lost if the transaction log is damaged in the failure, if the intercheckpoint interval becomes very long due to the crash of the initiator. One possible solution to this involves the use of a number of

backup processes; these are processss that can assume responsibility for completing the initiator's activity in the event of its failure. These backup processes are in fact checkpointing processes at other sites. If the initiator fails before it broadcasts the Checkpoint Request Message, one of the backups takes the initiative. 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 initiator, one, and only one backup process has to assume the initiative. 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 initiator 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 initiator is found to be down, the first backup process assumes the initiative 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 initiator or one of the backup processes, will be active at any given time. Thus a checkpointing will begin in a finite time.

The role of the initiator in the algorithm is simply that of starting the next checkpoint. Apart from this function the initiator is not critical to the operation of the proposed algorithm. If a uniformly agreeable point in time can be made known to the individual sites, then the initiator can be eliminated from the algorithm. One way to achieve this is to

preassign the clock values at which the checkpoints will be taken. For example, we may take checkpoints at the clock values in the multiple of 1000. Whenever the local clock of a site crosses the multiple of this value, checkpointing can begin.

If the frequency of checkpointing is related to the load conditions and not necessarily to the clock values, then this method of preassignment will not work as well. In this case, a process will have to assume the role of the checkpointing initiator to initiate the checkpointing. A unique process has to be identified as the initiator. This may be achieved by using the solutions to the mutual exclusion problem [17] and making the selection of the initiator a critical section activity.

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[20]. If the crashed site has no replicated data objects and if 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 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. Te 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 save 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, 26].

7. Concluding Remarks

During normal operation of the database system, checkpointing is performed to prepare information necessary for a recovery from failures. 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.

In this paper, we have proposed a decentralized checkpointing algorithm for distributed database systems. The properties of global consistency and non-interference of checkpointing result in some overhead on the one hand, and increase the system availability on the other

hand. For the applications where the ability of continuous processing of transactions is so critical that the blocking of transaction processing for checkpointing is not feasible, we believe that the checkpointing algorithm presented in this paper provides a practical solution to the problem of constructing globally consistent states in distributed database systems.

REFERENCES

[1] Albert, A. and Sandler, R., An Introduction to Finite Projective Planes, Holt, Rinehart and Winston, New York, 1968.

[2] 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.

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

[4] 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.

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

[6] 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.

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

[8] 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.

[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] Lakshman, T. V. and Agrawala, A. K., Efficient Decentralized Consensus Protocols, IEEE Trans. on Software Engineering, May 1986, pp 600-607.

[13] Lakshman, T. V. and Agrawala, A. K., Communication Structure of Decentralized Commit Protocols, 6th International Conference on Distributed Computing Systems, Cambridge, Massachusetts, May 1986, pp 100-107.

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

[15] Maekawa, M., A SQRT(N) algorithm for mutual exclusion in Decentralized Systems, ACM Trans. on Computer Systems, May 1985, pp 145-159.

[16] 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.

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

[18] 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.

[19] Roberts, F. S., Applied Combinatorics, Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

[20] 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.

[21] 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.

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

[23] Son, S. H. and Agrawala, A. K., A Non-Intrusive Checkpointing Scheme in Distributed Database Systems, 15th International Symposium on Fault-Tolerant Computing, June 1985, pp 99-104.

[24] 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.

[25] 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

[26] 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.

[27] Son, S. H. and Agrawala, A. K., Practicality of Non-Interfering Checkpoints in Distributed Database Systems, Real-Time Systems Symposium, New Orleans, Louisiana, December 1986.