DISTRIBUTED CHECKPOINTING FOR GLOBALLY CONSISTENT STATES OF DATABASES

Sang Hyuk Son

Computer Science Report No. TR-86-23
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[4, 7]. 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, a new algorithm for checkpointing in distributed database systems is proposed and its correctness is proved. The practicality of the algorithm is discussed by analyzing the extra workload and the robustness of it with respect to site failures.

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 the checkpointing more complicated and 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 work-load incurred by the checkpointing mechanism.

In this paper, we propose a new checkpointing algorithm which is non-interfering and which efficiently generates globally consistent checkpoints. The correctness of the algorithm is shown, and the practicality of the algorithm is discussed. This paper is organized as follows. Section 2 introduces a model of computation used in this paper. Section 3 discusses the design issues for checkpointing algorithms and reviews previous work which has appeared in the literature. Section 4 describes the checkpointing algorithm. Section 5 presents an informal proof of the correctness of the algorithm. Sections 6 and 7 discuss the practicality of the algorithm by analyzing the workload and the robustness of the algorithm, and describe the recovery methods associated with the algorithm. Section 8 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, we assume that 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. 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.

We assume that the database system runs a correct transaction control mechanism (e.g., atomic commit algorithm[19] and 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. Related Work

In order to achieve the goal of efficient database system recoverability, it is necessary to consider the following issues when a checkpointing mechanism is designed for a distributed database system.

(1) it should generate globally consistent checkpoints,

(2) it should be non-interfering in that it does not affect the ongoing processing of transactions,

(3) the storage and the communication overhead should be small.

The need and the desirability of these properties is self evident. For example, even though an inconsistent checkpoint may be quick and inexpensive to obtain, it may require a lot of additional work to recover a consistent state of the database. Some of the schemes appearing in the literature (e.g. [1, 5]) do not meet this criteria.

Checkpointing can be classified into three categories according to the coordination necessary among the autonomous sites. These are (1) fully synchronized[10], (2) loosely synchronized[17], and (3) nonsynchronized[5]. 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 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 global transactions run in the local checkpoint intervals bounded by checkpoints with the same sequence numbers. In nonsynchronized checkpointing, global coordination with respect to the recording of checkpoints does not take place at all. Each site is independent from all others with respect to the frequency of checkpointing and the time instants when local checkpoints are recorded. A logically consistent database state is not constructed until a global reconstruction of the database is required.

One of the drawbacks common to the checkpointing schemes above is that the processing of transactions must be stopped for checkpointing. Maintaining transaction inactivity for the duration of the checkpointing operation is undesirable, or even not feasible, depending on the availability constraints imposed by the system.

In [13], checkpointing is always performed exclusively as part of the commitment of transactions. This scheme has the advantage of not having a separate checkpointing mechanism, but may have problems if the number of transactions allowed is too many or if it is necessary to keep checkpoints for a long time. A similar checkpointing mechanism is suggested in [11]. The synchronization of checkpointing in [11] is achieved through the time-stamp ordering, making the global reconstruction easier than in the scheme of [13]. The storage requirements of these transaction-based checkpointing mechanisms depend upon the amount of information saved for each transaction, and are difficult to compare with the checkpointing mechanisms which save only the values of data objects.

In [1], 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 drawback of this scheme is that the backup generation does interfere with update transactions.

In [7], a different approach 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 [7] describes the behavior of an abstract system and does not provide a practical procedure for obtaining a checkpoint.

Our new 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 the messages are delivered in the order they are sent. The notion of diverged computation in [7] is captured in the "committed temporary versions" of data objects in our algorithm.

4. An Algorithm for Non-Interfering Checkpoints

4.1. 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.

4.2. 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 checkpointing algorithm 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:

$$CONVERT_{CC} := false$$

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:

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

(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.

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

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 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 5 and time-stamp(T) $\leqslant$ GCPN, then a TIM-NACK message is returned. 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.

4.3. 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 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 [2, 20] 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 [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 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.

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.

6. Performance Characteristics

In order to discuss the practicality of the proposed algorithm, we consider two performance measures: extra workload and extra storage required. We assume that for each transaction during its execution, there exists a private buffer. All updates made by a transaction are performed tentatively on copies of data objects in the private buffer. When a transaction commits, the updates are propagated from the buffer space either to the database (for

BCPT) or to the committed temporary versions file (for ACPT), and the buffer space is cleared. If a transaction aborts, the buffer space is simply cleared without any data propagation. The updates in the CTV file are propagated to the database by the *reflect* operation when the current checkpointing is terminated or when an ACPT is converted to a BCPT. Figure 1 shows the different execution sequences of BCPT and ACPT.

The extra workload imposed by the algorithm mainly consists of the workload for (1) determining the GCPN, (2) committing ACPT (move data objects from the buffer space 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.

It takes three message exchanges to determine GCPN at each site. Since the time for processing the messages of LCPN and GCPN is negligible when compared to the I/O time for performing the commit and the reflect operations, we neglect the portion of extra workload for determining the GCPN. We also neglect the portion of extra workload for making the CTV file clear.

The commit operation of an ACPT consists of the following two steps: (1) transferring the data objects from the buffer space to the CTV file, (2) inserting these data objects into the CTV file. We assume that these two steps are performed independently, that is, while a data object is being inserted into the CTV file, other data objects can be transferred to the CTV file. The time required to commit an ACPT, T_{CA}, is a function of the number of data objects updated by the transaction, and the maximum time to perform these two steps:

$$T_{CA} = \max(T_{tc}(n), T_{ic}(n))$$

where n is the number of data objects updated by the transaction, $T_{tc}(n)$ is the time required to transfer n data objects to the CTV file, and $T_{ic}(n)$ is the time required to insert n data objects into the CTV file.

Let T_{CB} be the time required to commit a BCPT. When a BCPT commits, all the updates are inserted from the buffer space to the database. This is the minimum time required to commit a transaction, and thus it must be subtracted from the extra workload required by the algorithm. As in the commit operation of an ACPT, the commit operation of a BCPT consists of two steps: (1) transferring the data objects from the buffer space to the database, (2) inserting these data objects into the database. T_{CB} is a function of the number of data objects updated by the transaction, and the maximum time to perform these two steps:

$$T_{CB} = \max(T_{td}(n), T_{id}(n))$$

where $T_{td}(n)$ is the time required to transfer n data objects to the database, and $T_{id}(n)$ is the time required to insert n data objects into the database.

Let T_R be the time required to reflect the data objects updated by an ACPT into the database. The reflect operation also consists of two steps: (1) transferring the data objects from the CTV file to the database, (2) inserting these data objects into the database. T_R is a function of the number of data objects updated by the transaction, and the maximum time to perform these two steps:

$$T_R = \max(T_{tcd}(n), T_{id}(n))$$

where $T_{tcd}(n)$ is the time required to transfer n data objects from the CTV file to the database.

For each ACPT, the extra time required to process it is formulated by summing the commit time and reflect time, and subtracting the commit time of a BCPT. Let N_A be the number of ACPT at the site. Then, the overall extra time required to execute the algorithm is

$$T_{OH}=N_A(T_{CA}+T_R-T_{CB})$$

In the following subsections, we formulate the expected number of ACPT, and analyze the effects of different parameters on the overall extra time.

6.1. Expected Number of ACPT

We model the database system as a queuing system with Poisson process of transaction arrivals. For the simplicity in the analysis, we assume that transactions are processed by the simple first-come, first-served principle. In real database systems, more than one transaction can be processed concurrently by the virtue of concurrency control mechanisms.

For BCPT, the database system is a single server which performs necessary processing and the commit (or abort) operation for the transaction. The database system is a two server system for ACPT; the first server performs the same operation as the single server for BCPT, while the second server performs the reflect operation. We assume an exponential service time distribution for the first server, and general service time distribution for the second server. Figure 2 shows the queueing model of the database system used in our workload consideration. Using this model, the expected number of ACPT at a single site will be formulated.

Note that we are not interested in estimating the exact number of ACPT. We are interested in determining the number of transactions that cause any extra workload to the system during a single checkpoint. Thus, the expected number of ACPT includes the number of temporary ACPT which are eventually converted to BCPT, because they incur overhead to the database system.

There is a single stream of transactions that flows into the database system. The user is not aware of checkpointing process when he submits a transaction. From the user's viewpoint, there is no distinction between a BCPT and an ACPT. It is the database system that must mark the transaction either as a BCPT or an ACPT. If a transaction must be an ACPT, then extra processing is performed by the database system. Therefore, the input process of ACPT into the database system is the same as that of BCPT.

Let the arrival rate of transactions to each site of the database system be λ and the mean service time of BCPT be exponentially distributed with mean value $\frac{1}{\mu}$. The mean number of arrivals during the service time of a BCPT is

$$\rho = \int_0^{\infty} \lambda t \mu e^{-\mu t} dt = \frac{\lambda}{\mu}$$

Since a single stream of BCPT changes to a single stream of ACPT when the local checkpoint number is determined, there must be a transaction which arrives as the last transaction of the BCPT stream. It has the largest time-stamp not greater than the current local checkpoint number. We call it the *last* BCPT. We now observe the system immediately after the arrival of the last BCPT at the site. As far as the BCPT processing is concerned, the database system is a M/M/1 queue. Therefore, the time to finish the work present when the last BCPT arrives is the same as the waiting time of a transaction in the system, which is $\frac{\lambda}{\mu(\mu-\lambda)}$.

Let T_B be the time required to finish processing of all BCPT. T_B is the sum of the waiting time and the service time of the last BCPT, thus we have

$$\begin{aligned} T_B &= \frac{\lambda}{\mu(\mu-\lambda)} + \frac{1}{\mu} \\ &= \frac{1}{\mu-\lambda}. \end{aligned}$$

Let T_D be the time required to save the state of the data objects to generate a local checkpoint. T_D is proportional to the number of data objects stored at the site, and is fixed for a given database system. The time interval during which the system needs to process ACPT instead of BCPT is the sum of T_B and T_D. Thus from the Little's Theorem, the expected number of ACPT is

$$N_A = \left(\frac{1}{\mu-\lambda} + T_D\right)\lambda$$

6.2. Analysis of Extra Workload

As shown in figure 1, the difference in committing BCPT and ACPT is whether to use the database or the CTV file in the commit operation. For a BCPT, the updated data objects in the buffer space of the transaction have to be transferred to the database; for an ACPT, they have to be transferred to the CTV file. In most implementations the same hardware is most likely to be used for both the database and the CTV file. Hence we assume that the commit operation of a BCPT requires the same service time as the commit operation of an ACPT: $T_{CB} = T_{CA}$. Now the extra workload becomes a function of the expected number of ACPT and the service time for the reflect operation:

$$T_{OH} = N_A \times \max(T_{tcd}(n), T_{id}(n))$$

Let us assume that K_0 unit time is required to insert one data object into the database. Therefore,

$$T_{id}(n) = K_0 \times n$$

Before any insertions are made to the database for the reflect operation, the data objects have to be transferred from the CTV file to the buffer of the database. In the best case it takes the minimum required time regardless of the number of data objects updated by the transaction. The time required in the worst case is proportional to the number of data objects in the CTV file.

We calculate the extra workload of the algorithm as the following, considering different service requirements for the transfer operation. Case I is the best case where the minimum time unit C_0 is required for each ACPT. In this case, $T_{OH}=N_A \times \max(C_0, K_0 \times n)$. In case II, the worst situation is considered where the time required for the reflect operation is a linear function of the number of data objects updated. We have $T_{OH}=N_A \times \max(C_1 \times n, K_0 \times n)$. Case III might be closer to the real system performance where the time required for the transfer operation is proportional to the square root of the number of data objects updated: $T_{OH}=N_A \times (C_2 \times \sqrt{n}, K_0 \times n)$.

We present below some typical combinations of parameters that illustrate the performance range of the algorithm. In general, if the number of data objects involved in the transaction is small, the performance of the best case is closer to the real system performance. As the number of data objects involved in the transaction becomes large, the performance of the worst case becomes closer to the real system performance.

Among several parameters we can change, ρ (the mean number of arrivals) and n (the average number of updates) are key parameters to characterize the extra workload. The extra workload will be increased when the number of ACPT is increased. It will be also increased when the number of updates are increased.

As shown in figure 3, the extra workload increases very rapidly as ρ approaches to 1. In other words, when the database system is saturated by the incoming transactions ($\lambda \geqslant 0.9\mu$), the extra workload by the checkpointing algorithm becomes unacceptable.

Figure 4 shows the extra workload as a function of the average number of data objects updated by an ACPT. As we can see, the extra workload is not as sensitive to n as λ in its higher range ($\lambda \geqslant 0.8\mu$). However, case I is not practical for a large number of data objects because the capability of reading and transferring the data objects in a constant time is limited by hardware. Therefore, the actual extra workload is closer to case I only when the number of data objects involved is very small. When the number of data objects involved in the execution of a transaction becomes large, the extra workload emerges toward case III or even toward case II. It implies that if the average number of data objects updated by an ACPT is quite large, the extra workload of the scheme may become unacceptable.

The extra workload of the algorithm is determined by the dynamic parameters of the transaction characteristics (e.g., ρ, n). Naturally, this number is an important criteria for deciding whether or not to perform checkpointing non-intrusively. For the circumstances in which ρ and/or n is so high that the execution of the algorithm would severely degrade the performance of the system, a conventional intrusive checkpointing might be preferred. In most cases, however, the system may require non-intrusiveness and global consistency of checkpoints. In order to make a proper decision, the system may provide a mechanism to determine the dynamic parameters at regular intervals during normal operation and update them at some safe storage. When the next checkpointing needs to be done, the checkpoint coordinator decides whether or not to execute the algorithm, based on the expected extra workload calculated using the values of the parameters saved. If the extra workload is less than certain threshold value, the non-intrusive checkpointing will be executed.

6.3. Storage Requirement

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. In terms of the system parameters, we have

$$\begin{aligned} \text{CTV file size} &= N_A \times (\text{number of updates}) \times (\text{size of a data object}) \\ &= (\frac{1}{\mu - \lambda} + T_D)\lambda \times n \times m \end{aligned}$$

where m is the size of a data object fixed by the system.

As in the extra workload estimation, the parameters ρ and n play a critical role in estimating the required CTV file size. The size of the CTV file may become unacceptably large if ρ approaches to 1 or n 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 m, the granularity of a data object. The size of the CTV file can be minimized if we minimize m. 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, 5, 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.

7. Discussion

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.

7.1. Site Failures

The algorithm is insensitive to failures of subordinates before or during its execution. 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.

The algorithm is, however, sensitive to failures of the coordinator. In particular, if the coordinator crashes during the first phase (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 $p_1, p_2, \ldots, p_n$. 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.

The role of the checkpointing coordinator in the algorithm is simply that of getting a uniformly agreed GCPN. Apart from this function the coordinator is not critical to the operation of the proposed algorithm. If a uniformly agreeable GCPN can be made known to the individual sites, then the centralized nature of the coordinator can be eliminated. 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 the preassigned GCPN will not work as well. In this case, a process will have to assume the role of the checkpointing coordinator to initiate the checkpointing. A unique process has to be identified as the coordinator. This may be achieved by using the solutions to the mutual exclusion problem [15] and making the selection of the coordinator a critical section activity.

7.2. 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 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 log file 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. 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 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, 22].

8. 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, a new checkpointing algorithm for distributed database systems is proposed and discussed. The correctness of the algorithm is proved, and the performance characteristics of it are discussed by formulating the extra workload and the storage requirement.

A non-interfering checkpointing is desirable in many applications, and it has been shown that it can be a viable solution if the extra workload and storage requirement remain in an acceptable range. Two important parameters in making a non-interfering checkpointing practical are the mean number of transaction arrivals and the average number of updates of a transaction. As far as they remain below certain threshold values, the overhead of non-interfering checkpointing can be justified.

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] 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.

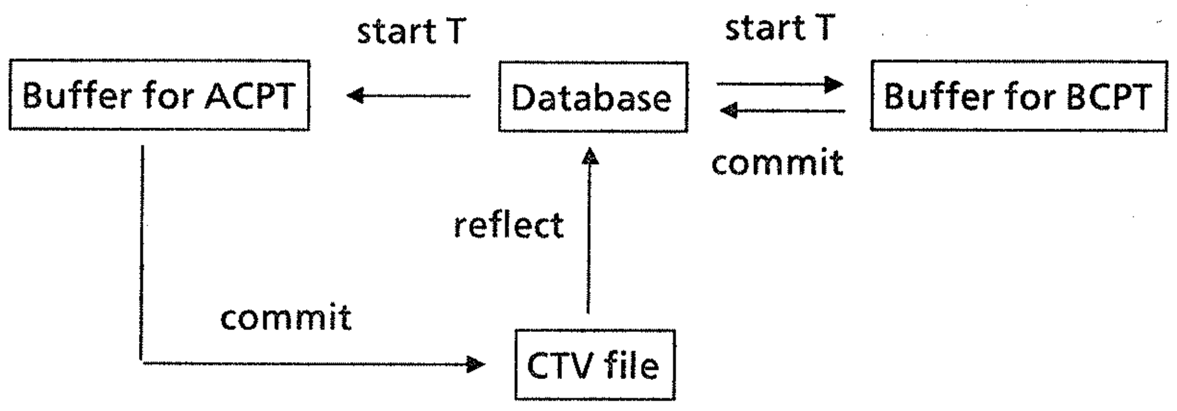

Fig. 1 Execution sequence of ACPT and BCPT.

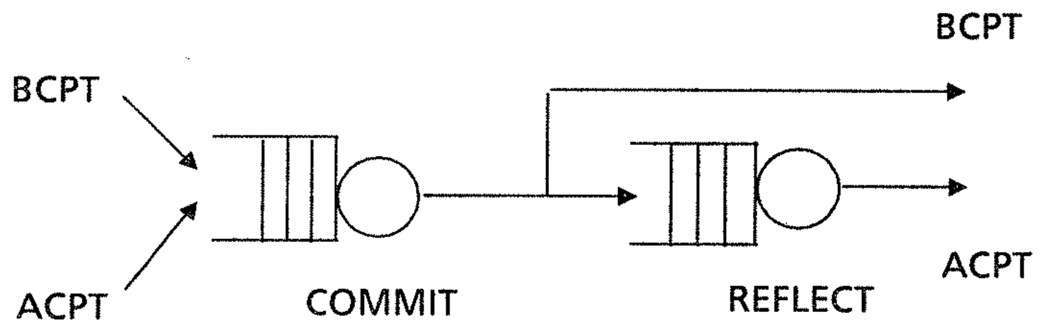

Fig. 2 Queueing model of the system.

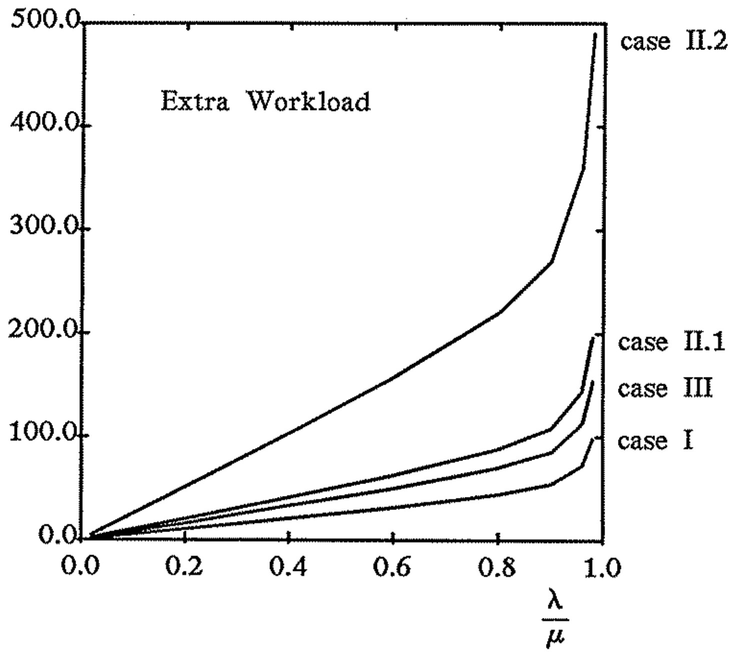

Fig. 3 Extra workload of the database system.
Parameters: n=4(Case II.1), n=10(Case II.2 & III), T_D=10, K_0=0.1, C_0=1, C_1=C_2=0.5.

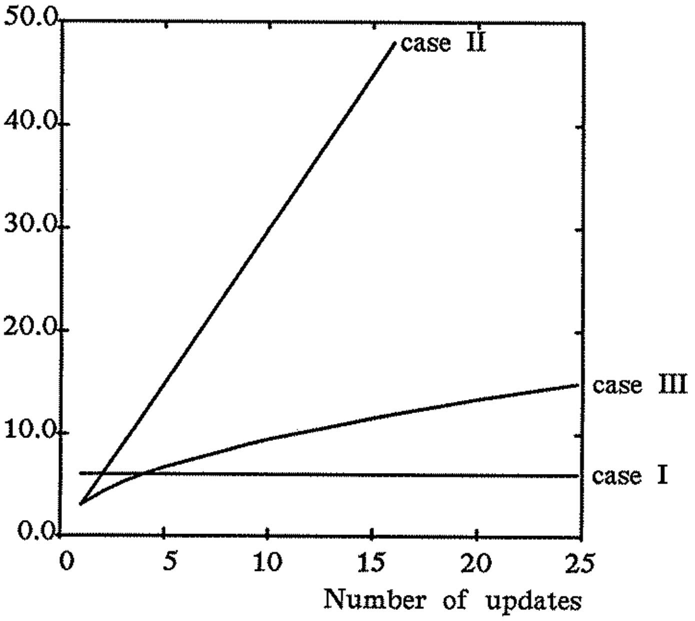

Fig. 4 Extra workload of the database system.
Parameters: T_D=10, K_0=0.1, C_0=1, C_1=C_2=0.5, $\frac{\lambda}{\mu}$=0.5.