MIT/LCS/TM-144

CONCURRENT AND RELIABLE UPDATES OF DISTRIBUTED DATABASES

Akihiro Takagi

November 1979

CONCURRENT AND RELIABLE UPDATES OF DISTRIBUTED DATABASES

by

Akihiro Takagi

ABSTRACT

A concurrent execution of transactions and various failures occurring during transaction processing in a distributed database system can lead to an inconsistent database state. In order to prevent such inconsistency from occurring, 1) the schedule of transactions must be equivalent to some serial schedule and 2) each transaction must be either completed or backed out. This paper develops a set of schemes that satisfy these requirements and still realize highly concurrent execution of transactions. This paper also shows how to incorporate these schemes into a multi-level distributed database system where there exists a hierarchy of transactions. Detailed algorithms for concurrent and reliable updates of distributed databases based on the proposed schemes are included in the appendix.

Key words and phrases: distributed database, concurrency control, recovery, recoverable objects, atomicicity, multiple versions.

ACKNOWLEDGMENT

This work was done while the auther was on a visit to the Laboratory for Computer Science, Massachusetts Institute of Technology, from 1977 to 1978. The auther wishes to thank Professor Liba Svobodova for her many helpful comments and suggestions on the early versions of this paper.

CONTENTS

	ABSTRACT	2
	ACKNOWLEDGMENT	3
	TABLE OF CONTENTS	4
1.	INTRODUCTION	5
2.	TRANSACTIONS --- A GENERAL DISCUSSION	7
2.1.	Consistent Schedule of Transactions	8
2.2.	All or Nothing Property of Transactions	11
2.2.1	Backing Out	12
2.2.2	Forcing Completion	20
3.	BASIC STRATEGIES	21
4.	DETAILED SCHEMES	23
4.1.	Assumptions	23
4.2.	Timestamps	24
4.3.	Transaction Scheduling based on Timestamps	25
4.4.	Management of Uncommitted Versions	27
4.4.1.	Commitment of Transactions	31
4.4.2.	Cascading of Backout	33
5.	CONCLUSION	34
	APPENDIX A: LANGUAGE CONSTRUCTS	37
	APPENDIX B: DETAILED ALGORITHMS	41
	REFERENCES	68

1. INTRODUCTION

Components of a database are related to each other in certain ways. Such relations are usually called consistency constraints. Since these consistency constraints cannot necessarily be enforced at each primitive action on components (usually called entities) such as read and write, sequences of actions are grouped to form transactions, which are then units of consistency. Each transaction must transform the database from a consistent state to a new consistent state [6][8][9]. Therefore, transactions are also units of recovery(1).

Although transactions, when executed one at a time, preserve consistency, concurrent execution of transactions and various failures occurring during transaction processing could cause such anomalies as lost updates, dirty read and unrepeatable read [9]. To prevent these anomalies from occurring, it is usually required that for given concurrent schedule of transactions there exists some serial schedule that is equivalent to it. Schedules that satisfy such a property are called consistent schedules. In addition, it is necessary to be able to restore the database to an earlier consistent state by backing out affected transactions when a failure occurs during transaction processing. This paper proposes some concepts and mechanisms that provide a highly concurrent and consistent schedule of transactions as well as facilities for recovering from various kind of failures.

---

(1) The concept of a transaction is similar to the concept of a sphere of control [3][5], the concept of conversation [21], or more generally, the concept of an atomic action [18][22]. Also it has close relation to the concept of monitor [11][12].

There are several related works that can be divided into three classes. The first class is represented by Gray et al [6][8][9] who proposed a lock protocol that guarantees any legal schedule to be consistent and transaction backout to be feasible. Their lock protocol requires each transaction to:

(a) set an exclusive lock on any entity it dirties

(b) set a share lock on any entity it reads

(c) hold all locks to the end of transaction

Apparently this lock protocol seriously restricts concurrent execution of transactions since it almost serializes any pair of transactions if there exists at least one entity that is needed in exclusive mode by both of them. In fact, the degree of concurrency in System R, which forces transactions to observe this lock protocol, is reported to be less than two [10]. In addition, it is subject to deadlock that would be very costly in a distributed environment.

The second class is represented by Bernstein et al [2] and Montgomery [20] who proposed methods to avoid locking wherever possible by utilizing the knowledge about the access patterns of transactions. These patterns are acquired from the pre-analysis of the transactions. A timestamp mechanism [2] or a hierarchical concurrency control mechanism [20] can be then used to provide a deadlock-free transaction serializer. Although these methods realize a highly concurrent execution of simple and predictable transactions, they do not seem to be of much help either when transactions need some degree of synchronization internally or when the access patterns of transactions are unpredictable. Moreover, recovery mechanisms do not seem to be fully developed in

these works[1].

The third class is represented by Reed [23] who developed the concept of a pseudo-temporal environment that is an extension of global timestamps and the concept of versions of an object. Reed's method that is based on these concepts provides a consistent schedule of transactions as well as facilities for recovering from various kinds of failures. However, concurrent transactions that access the same data must be executed almost serially. Moreover, they are subject to dynamic deadlock, where they cause each other to be mutually aborted [23].

This paper develops some new concepts and mechanisms that solve the above problems. In addition, this paper proposes a practical method to implement these mechanisms for concurrency control and recovery in a multi-level distributed system, which is based on the belief that different mechanisms should be used at different levels of the system although more than one level may use the same kind of mechanism[2].

2. TRANSACTIONS --- A GENERAL DISCUSSION

The concept of a transaction introduced in the previous section is generalized in this section such that it simple implies atomicity. The representation of a transaction generally consists of a set of underlying objects (e.g. database entities) and a sequence of actions

---

(1) However, some work to build recoverability into SDD-1, to which Bernstein's method was actually applied, is in progress.

(2) Randell [21][22], Verhofstad [28], Anderson et al [1] and Reed [23] treated recovery issues in a multi-level system from somewhat different viewpoints.

on them that may also be transactions of a lower level. Transactions are considered to be indivisible and instantaneous, as far as their users (callers) are concerned, such that their effect on the system is the same as if they were executed sequentially [18][22]. These characteristics must be preserved even if several transactions are invoked concurrently or if any kinds of failures occur during the processing of transactions. Therefore, our primary concerns are to ensure that:

(1) conflicts among transactions never occur (i.e. the schedule of transactions is equivalent to some serial schedule)

(2) temporary inconsistency never becomes permanent (i.e. each transaction is either completed or backed out).

2.1 Consistent Schedule of Transactions

A schedule S for a set of transactions T1, T2, ... , Tn defines the binary relation $<$ such that T1 $<$ T2 if transaction T1 performs action a1 on object e at some step in S and transaction T2 performs action a2 on e at a later step in S and if a1 is not permutable with a2. Let $<^*$ be the transitive closure of $<$. Then we can restate the condition that "the effects on the system be as if they were executed sequentially" more formally as the condition that "the schedule of transactions must be such that the relation $<^*$ is a partial order [6][8][9]". Such a schedule is called a consistent schedule.

There are three alternative schemes that can be used to implement a consistent schedule :

(1) serial schedule

(2) schedule based on a lock protocol

(3) schedule based on timestamps or similar mechanisms.

The first scheme completely serializes any pair of transactions between which the relation $<$ exists. This occurs in most conventional operating systems where (at least a part of) supervisor programs are executed serially. Clearly, this scheme is simplest and the associated overhead is smallest. However, the degree of concurrency achieved by this scheme is zero (of course, mutually independent transactions can be executed concurrently even in this scheme).

The second scheme was explored by Gray et al [6][8][9] and is widely used in many database systems. Although a transaction has to lock underlying objects in this scheme, it is possible to ensure that any legal schedule is consistent if each transaction observes a two-phase lock protocol [6] (i.e. transaction can not request new locks after releasing a lock). The degree of concurrency achieved by this scheme is in general better. But it is not the best since the two-phase constraint is only a sufficient condition [6]. It is also subject to deadlock.

The third scheme is based on the observation that a consistent schedule of transactions is merely a sequencing of actions performed on the underlying objects by these transactions such that the relation $<*$ be a partial order. This sequencing is directly controlled using a timestamp mechanism [2][23] or a hierarchical concurrency control mechanism [20] rather than a lock mechanism. When a timestamp mechanism is used, each transaction is assigned a globally unique timestamp, and thereby all transactions are totally ordered. The type manager of each object schedules actions on the object in the timestamp order of transactions

that requested these actions(1). This distributed (i.e. per-object-based) scheduling algorithm guarantees that, for any pair of transactions T1 and T2 both of which access the same objects e1, e2 ... en, T1 $<$ T2 if and only if the timestamp assigned to T1 is smaller than the timestamp asssigned to T2. Therefore, the relation $<^*$ defined by this scheduling algorithm is a partial order that can be extended to the timestamp order. When a hierarchical concurrency control is used, each of T1 and T2 broadcasts its requests for accesses to e1, e2, ..., en through a common serializer (thus there exists a hierarchy of serializers). Again, the relation $<^*$ is apparently a partial order. In any case, this type of scheme ensures the maximum degree of concurrency since it imposes no more restraints than necessary (i.e. the relation $<^*$ be a partial order). In addition, it is deadlock free since $<^*$ is an acyclic relatlion. However, it requires a non-trivial algorithm that ensures that actions are eventually executed in the timestamp order even if components of the system fail or sequence anomalies occur because of communication delays, processing delays etc.(2). Such an algorithm may induce a greater overhead than previous schemes do.

---

(1) The schedule based on timestamps is somewhat similar to the methods that were devised to solve the mutual exclusion problem by Lamport [14], Rivest et al [24] etc..

(2) For example, suppose that both of T1 and T2 perform actions on two objects e1 and e2. Then it may happen that e1 gets a request from T1 before that from T2, but e2 gets requests in the reverse order.

2.2 All or Nothing Property of Transactions

In order to prevent a temporary inconsistency from becoming permanent when a failure (1) is encountered, it must be always possible to decide whether or not to complete any outstanding transactions and perform the alternative thus selected. Unfortunately, there exists no finite length protocol which ensures that each transaction is either completed or backed out in a distributed system in which nodes or communication lines may fail at any time [20]. Therefore, the second best policy is to relax the requirement for finiteness of the protocol, but attempt to minimize the time window during which a failure causes unnecessary delay. This is the main aim of the two-phase commit protocol that emerged independently both in IBM and XEROX and was first mentioned publicly by Lampson et al [16]. In the two-phase commit protocol, a commit point is established after the first phase of commitment (2) is successfully completed. If something goes wrong before the commit point, the transaction is backed out. On the other hand, the transaction must be completed no matter what happens after the commit point (it may cause an infinite delay). The rest of this section examines in detail the schemes and mechanisms used to ensure the all or nothing property of transactions.

---

(1) This paper excludes media failures such as a head crash, dust on magnetic media etc. and serious failures of operating systems. To cope with such failures, extra recovery schemes such as incremental dump, long-term checkpoint, differential files etc. [4][9][19][29] are needed.

(2) Committing the change of an object's state means making this change decisive to the users of the object.

2.2.1 Backing Out

In order to back out a transaction,

1) the states of the underlying objects that were accessed by the transaction so far must be restored to the states they were in when the transaction was invoked

2) the information flow from the transaction must be undone and all other transactions affected by this information flow must also be backed out (this is called cascading of backouts [9] or domino effect [22])

3) these must be completely done even if a failure occurs at any time.

(1) Recovery of objects

Recovery of underlying objects accessed by a transaction consists of two phases as follows ;

1) deciding which objects the transaction accessed

2) restoring the states of these objects to ones they were in before the transaction was invoked

In this paper, a recovery scheme used to implement the first phase is called a transaction-oriented recovery scheme because the first phase associates objects that were processed together by a given transaction. On the other hand, a recovery scheme used to implement the second phase is called an object-oriented recovery scheme because the second phase deals with a history of actions performed on a given object by different transactions.

A transaction-oriented recovery scheme basically remembers the identifiers of the objects accessed by the transaction, and requests

object-oriented schemes associated with these objects to restore the states of the objects when something goes wrong. An audit trail (or a log) [4][9] and a recovery cache [13] can be used as transaction-oriented schemes.

Before discussing object-oriented schemes, it is necessary to give a definition of recoverability of objects. The representation of the object's state is managed solely by the appropriate type manager. A type manager is common to all objects of a particular type. Objects are classified into two categories : recoverable objects and non-recoverable objects. A recoverable object is one whose type manager provides recovery. Namely, the type manager saves recovery data and restores an earlier state when required to do so by the user of the object. On the other hand, a non-recoverable object is one whose type manager does not provide recovery. The type manager of a non-recoverable object does not save recovery data. (1) However, it is possible that the user of a non-recoverable object can undo the effect of the earlier action by invoking an inverse or a compensating action [3] (if such an action can be defined) ; this assumes that the user keeps sufficient recovery data. A careful replacement, multiple copies and differential files are examples of an object-oriented recovery that were proposed so far. However, no recovery scheme proposed so far permits sufficiently concurrent execution of mutually dependent transactions.

---

(1) Strictly speaking, an object whose type manager performs recovery using an audit trail saved by the user should be distinguished as another category. However, this kind of object is included in non-recoverable objects since such distinction is not essential as far as this paper is concerned.

In an object-oriented recovery scheme for a recoverable object, the type manager of the object remembers a history of the changes of object's state and identifiers of transactions [1] that depend on each state. The scope of the history that the type manager must remember depends on the kind of failures that must be tolerated and the kind of recovery requested. [2] This paper discusses a recovery scheme that backs out a transaction that has failed, that is, the transaction that cannot be completed because some error has been experienced during its execution; such an error may be a hardware failure, residual software bug, or a synchronization conflict. Media failures and serious failures of the operating system are not addressed here. The recovery scheme restores the states of the underlying objects to the consistent states they were in when the transaction was invoked. Therefor, the history of the state's changes that the type manager must remember includes only the most recently committed state plus all succeeding changes. The type manager of a recoverable object provides two special kinds of actions : commit and undo. When a transaction performs commit (undo) on some object, the type manager of this object commits (undoes) the state's change caused by the transaction.

In an object-oriented recovery scheme for a non-recoverable object, on the other hand, the user of the object has to save recovery data. However, it is difficult to implement a truely object-oriented and efficient recovery scheme for a non-recoverable object since recovery

---

(1) As will be discussed later, this information is necessary in order to control the cascading of backout of transactions.

(2) See Verhofstad [29] for further details.

data concerning the object are distributed among the users of the object. For example, suppose that several transactions are permitted to be executed concurrently. Then in order for a given transaction to undo the action on a non-recoverable object, it has to consult all other transactions that may have accessed this object [1] (this inefficiency will be intolerable in a distributed environment). Therefore, a non-recoverable object is useful only in a limited environment where the cascading of backout does not arise, namely where a transaction is not permitted to access objects until all previous transactions that accessed them are completed. Also, none of object-oriented recovery schemes for non-recoverable objects proposed so far are sufficiently general in the sense that they can be applied to all classes of objects. In particular, none of them can be applied to a non-recoverable object whose type manager does not provide an inverse or a compensating action for each action.

Unifying both of a transaction-oriented scheme and an object-oriented scheme into a single scheme is often usefull. An audit trail and a recovery cache are such examples. However, these schemes have a number of problems as discussed above.

(2) New recovery schemes

This paper proposes two new recovery schemes i.e. a backout/commit cache and multiple uncommitted versions of mutable objects to solve the above problems.

---

(1) This is necessary to control the cascading of backout.

A backout/commit cache is a transaction-oriented recovery scheme as well as an object-oriented recovery scheme for non-recoverable object. A backout/commit cache is associated with a transaction. It is created when the transaction is invoked, and is deleted after the transaction is completed or aborted. A backout cache contains a set of actions to be performed in the case of backout, on the other hand, a commit cache contains a set of actions to be performed after the commit points is passed [1]. This scheme requires that each transaction T accesses the underlying objects as follows. Let a be an action that T wants to perform on object e. Also let $\bar{a}$ be the inverse or compensating action of a if such an action can be defined. Then ;

1) if e is non-recoverable and $\bar{a}$ is not defined, T simply writes a into the commit cache associated with T[2],
2) if e is non-recoverable and $\bar{a}$ is defined, T writes $\bar{a}$ into the backout cache associated with T, and performs a,
3) if e is recoverable, T writes <u>undo</u> into the backout cache, writes <u>commit</u> into the commit cache, and performs a.

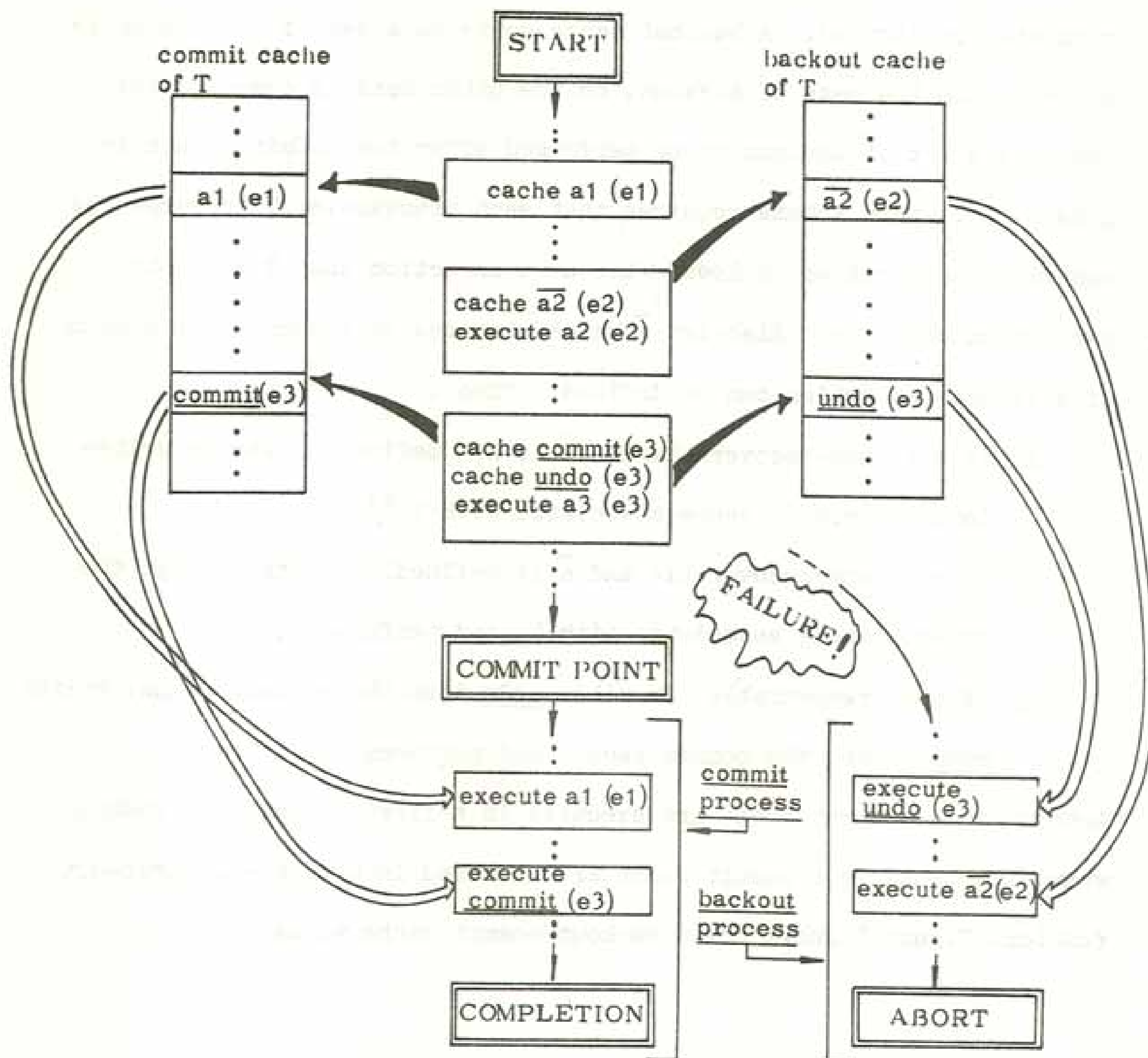
Actions in a backout cache are executed in a first-in last-out fashion, whereas actions in a commit cache are executed in a first-in first-out fashion. Figure 1 shows how a backout/commit cache works.

---

(1) Commit cache can be considered to be an extension of an intention list proposed by Lampson et al [16].

(2) It is necessary in this case that execution of a can be deferred until the end of the transaction without changing the logic of the transaction.

Execution History
of Transaction T



note : action a1 ; defined on non-recoverable object e1
(but $\overline{a1}$ is not defined)
action a2 ; defined on non-recoverable object e2
($\overline{a2}$ is also defined)
action a3 ; defined on recoverable object e3

Figure 1. Illustration of the behavior of a backout/commit cache

Under this scheme, backing out and committing a transaction simply means executing the actions saved in the backout and commit cache, respectively. Thus this scheme, unlike other ones proposed so far, reflects the all or nothing property of transactions explicitly. And this scheme is a sufficiently general one in the sense that it can be applied to almost any kind of objects.

This paper also proposes a new object-oriented recovery scheme for recoverable objects that permits highly concurrent execution of transactions accessing such objects. This scheme uses multiple uncommitted version of mutable objects ; it is an extension of careful replacement [27][29]. In this scheme, whenever a transaction tries to perform an action on an underlying object, a new version of the object is created and the actual action is performed on this new version. A transaction can create a new version before the immediately preceding transaction has committed the current version, although the commitment of the new version must be deferred until the current one is committed. Each version continues to exist until the immediately succeeding version is committed. Each version contains the additional information such as the identifier of the transaction that performed the action (i.e. created this version). Therefore, this new scheme records not only a complete history of the state's change of the object caused by uncommitted actions but also what transactions depend on each version. This makes it possible to control the cascading of backout caused by the backout of an uncompleted transaction, and therefore transactions can be executed highly concurrently.

Note that although multiple uncommitted versions are somewhat similar to Reed's tokens [23], there exists a crucial difference between two. Namely, the former provides a powerful concurrency control as its central facility, whereas the latter does not.

(3) Cascading of backout

The main problem with the cascading of backout is to keep track of the information flow that originates from a given transaction. There are two approaches to the problem.

1) Preventing interactions

This approach prevents each transaction from interacting with other transactions until the end of the transaction. All known database systems follow this approach usually by using locking schemes to avoid conflicts among transactions. Obviously the drawback of this approach is that it seriously restricts concurrent execution of transactions. For example, prevention of interactions is achieved by holding all locks to the end of transaction in database systems that use locking schemes.

2) Control the cascading

This approach is to devise a (new) scheme that keeps track of the information flow and backs out the affected transactions when a failure occurs. As it was discussed above, the multiple uncommitted versions, along with a backout/commit cache, also serve as such a scheme. In the later sections, it is shown how simply the backout can be done using this scheme. Since each transaction can perform actions on the underlying objects before

their current versions are committed, a highly concurrent schedule can be achieved.

(4) Guaranteeing complete backout

In order for a transaction to be completely backed out, actions performed during the backout process

1) must not be lost even if a failure occurs,
2) must be repeatable (idempotent [9][16])[(1)] since if a failure occurs during a backout process, this backout process may have to be repeated.

Stable storage [16] that holds objects safely across a failure plays an important role to satisfy the first requirement. Namely, implementing versions of objects and backout/commit caches in such stable storage satisfies this requirement. To satisfy the second requirement, different methods can be chosen. One method is to reduce the actions performed during the backout process to a sequence of write actions that are well known to be repeatable [16]. Another method is to prevent the actions from being performed more than once by using a mechanism that provides an unique identifier (such as a timestamp) for each invocation of an action.

2.2.2. Forcing Completion

As was stated before, a transaction must be completed no matter what happens after the commit point. In order to complete commitment by all

---

(1) Repeatability (or idempotency) of actions means that performing them several times produces the same result as performing them exactly once.

means, actions performed after the commit point

1) must not be lost even if a failure occurs,

2) must be repeatable since if a failure occurs during a completion process, the completion process may have to be repeated.

These requirements can be satisfied by using the same methods that were proposed to guarantee complete backout.

3. BASIC STRATEGIES

Database systems are considered to be composed of multiple levels, correspondingly there exist multiple levels of transactions. One of my beliefs is that it is most appropriate to apply different schemes for achieving atomicity to different levels although more than one level may use the same scheme. This is because different levels generally have different requirements.

Higher level transactions must be executed as concurrently as possible since their processing time tends to be very long (especially when transactions spread over more than one node in a distributed system). Also, scheduling of higher level transactions ought to be deadlock free since deadlock detection tends to be very expensive [9]. In addition, it is necessary to support recoverable objects at such a high level, partly because the users (i.e. the transactions) are distributed and executed concurrently, and partly because the users do not want to be involved in cumbersome details of recovery issues. And, of course, these three requirements must be compatible. The only feasible solution is to adopt a schedule based on timestamps as a consistent scheduling scheme and multiple uncommitted versions as an object-oriented recovery scheme.

On the other hand, simplicity and low overhead are more important than concurrency in the case of lower level transactions since these transactions are frequently invoked as primitive functions and their processing time is much shorter. Also, a non-recoverable object is feasible at a lower level where the transactions from diferent users are almost serially executed and are confined to a single node. Therefore, it would be appropriate at a lower level to adopt a serial schedule as a consistent scheduling scheme and a backout/commit cache as an object-oriented recovery scheme.

Since a backout/commit cache is so universal (combines object-oriented and transaction-oriented recovery schemes), it could be applied to most levels.

This paper assumes a simplified multi-level distributed system that consists of three levels. The top level provides database transactions that access multiple and distributed database entities (this kind of transaction is simply called "transaction" hereafter), the intermediate level provides logical actions on database entities and the bottom level, which is supported by the underlying operating system, provides physical actions on disk storages. The above discussion justifies the following basic strategies.

1) Assume that a disk storage is a non-recoverable object and physical actions on a disk storage are atomic. These are provided by the underlying operating system.
2) In order to make logical actions on a database entity atomic, use
   # a serial schedule as a consistent scheduling scheme

a backout/commit cache not only as a transaction-oriented recovery scheme but also as an object-oriented recovery scheme for a non-recoverable disk storage.

Also, in order to provide recoverable entities for transactions, use multiple uncommitted versions as an object-oriented recovery scheme.

3) In order to make transactions atomic, use

a schedule based on timestamps as a consistent scheduling scheme .

a backout/commit cache as a transaction-oriented recovery scheme.

4. DETAILED SCHEMES

This section mainly discusses a consistent schedule of transactions that uses a timestamp mechanism and multiple uncommitted versions of database entities since other schemes --- a serial schedule of logical actions on each entity and a backout/commit cache --- are rather straightforward. The details of the whole schemes are given in Appendix A and B.

4.1 Assumptions

This paper considers a distributed database system that consists of a set of nodes interconnected via communication lines. Each node consists of a set of subsystems ; data management subsystems and transaction management subsystems. A transaction management subsystem consists of transaction management processes that execute transactions, one at a time, by communicating with data management subsystems. Each data management subsystem maintains a portion of the database (i.e. a set of entities)

and controls accesses to them. It consists of type managers of entities and data management processes that access entities at the request of transaction management processes.

A transaction management process retrieves (updates) the content of an entity by sending a read (write) message to the data management subsystem that maintains the entity. The message is received by one of the idle data management processes of the subsystem. Then this process accesses the entity. For convenience, this paper classifies read messages into readr messages (i.e. read-only messages) and readu messages (i.e. read messages that are followed by write messages). An access to the database via a readr message is called a read-only access, and an access via a pair of a readu message and a write message is called an update access. A set of entities to be updated by a transaction is called its update set, and a set of entities to be read is called its read set. For the sake of simplicity, this paper assumes that :

(a) the update set of a given transaction T is a subset of its read set.

(b) T performs an (either read-only or update) access to each entity at most once.

4.2 Timestamps

Each node has a clock that generates globally unique timestamps. As was suggested by Thomas [26], it is possible to guarantee that every timestamp is globally unique by appending the transaction management subsystem number as the low order bits of each timestamp. The scheduling algorithm proposed here in itself requires only the uniqueness of each timestamp. However, in order to decrease possibilities that sequence

anomalies occur and to ensure each transaction an appropriate response time, it is desirable that clocks running in different nodes are reasonably synchronized. Lamport's method of synchronizing clocks in a distributed system [15] seems to be sufficient.

Each transaction is assigned a unique timestamp before accessing a set of entities. Each action (therefore each access) is assigned the same timetamp as was assigned to the transaction which performs the action. In addition, the timestamp assigned to each transaction is used as a version number in the versions of the set of entities accessed by that transaction. How such versions are created will be explained in the following sections.

4.3 Transaction Scheduling based on Timestamps

One of the key points of the concurrency control of transactions proposed here is that each type manager schedules accesses in the order of timestamps assigned to them. Therefore, the problem is how to ensure this ordering. There are three alternatives : a waiting approach, a no-wait-without-backup approach and a no-wait-with-backup approach.

(1) Waiting approach

In this approach, a type manager defers scheduling of an access to the entity it controls until it confirms that there exist no outstanding accesses to this entity that are assigned smaller timestamps. One of the drawbacks of this approach is that such a confirmation procedure takes a fairly long time [(1)], and is reduced to a pure overhead when requests for accesses arrive at each type manager in the timestamp order. Also, concurrent execution of transactions may be seriously restricted. SDD-1,

---

(1) If one takes into consideration possible failures of nodes or communication lines, this time may become unbounded.

adopting this approach, tries to remedy such a drawback by inventing null writes and limiting the number of transaction management subsystems that have to be polled [2]. This limitation of polling range is based on the information acquired from the preanalysis of transactions.

(2) No-wait-without-backup approach

In this approach, a type manager schedules each access immediately as far as it is assigned a greater timestamp than all accesses to the same entitiy that were already performed. On the other hand, if there exist some accesses that are assigned greater timestamps and already performed this outdated access is rejected. Reed [23] proposed this kind of approach. However, it is inappropriate especially in the environment where multiple uncommitted versions are permitted to exist since if requests for accesses by two different transactions are received in the reverse order at two different nodes, then both transactions are eventually rejected [(1)].

(3) No-wait-with-backup approach

This paper proposes another approach, that is, no-wait-with-backup approach. In this approach, a type manager schedules each access immediately as far as it is assigned a greater timestamp than all accesses to the same entity that were already performed. If there exist some accesses that are assigned greater timestamps and already performed but not yet committed, then transactions that performed these accesses are backed out, and after that the temporally outdated access is performed. On the other hand, if there exist some accesses that are

---

(1) This is likely to occur even if multiple uncommitted versions are not permitted. Reed [23] calls it dynamic deadlock.

assigned greater timestamps and already committed, then the outdated access is rejected. This approach not only realizes highly concurrent execution of transactions but greatly decreases the possibility of dynamic deadlock. In addition, introduction of multiple uncommitted versions makes this approach more attractive because, under this recovery scheme,

1) the backout algorithm is very simple and clean, and

2) if an outdated access is read-only, no transaction has to be backed out.

These points will be fully discussed in the next section.

4.4 Management of Uncommitted Versions

Whenever a transaction tries to access a given entity, a new version of that entity is created and the access is performed to this version [(1)].
Each version is preserved until its immediate successor is committed.

It is assumed that each entity is represented in the storage in the way shown in Figure 2. An entry of directory is associated with each entity and contains the address of the descriptor of the entity. Each descriptor entry consists of the following fields : v#, acc, s, addr.
The v# field contains the version number, which is equal to the timestamp of the transaction that created this version (by an access request).
The acc field indicates whether the access was read-only or update. The s field indicates the current state of this version, which may take on one of the following values :

1) dirty : already read, but not yet written (meaningless in the

---

(1) Note that a new version is not necessarily created at every action. Also note that a new version is (at least virtually) created even at a read-only access in order to prevent dirty read.

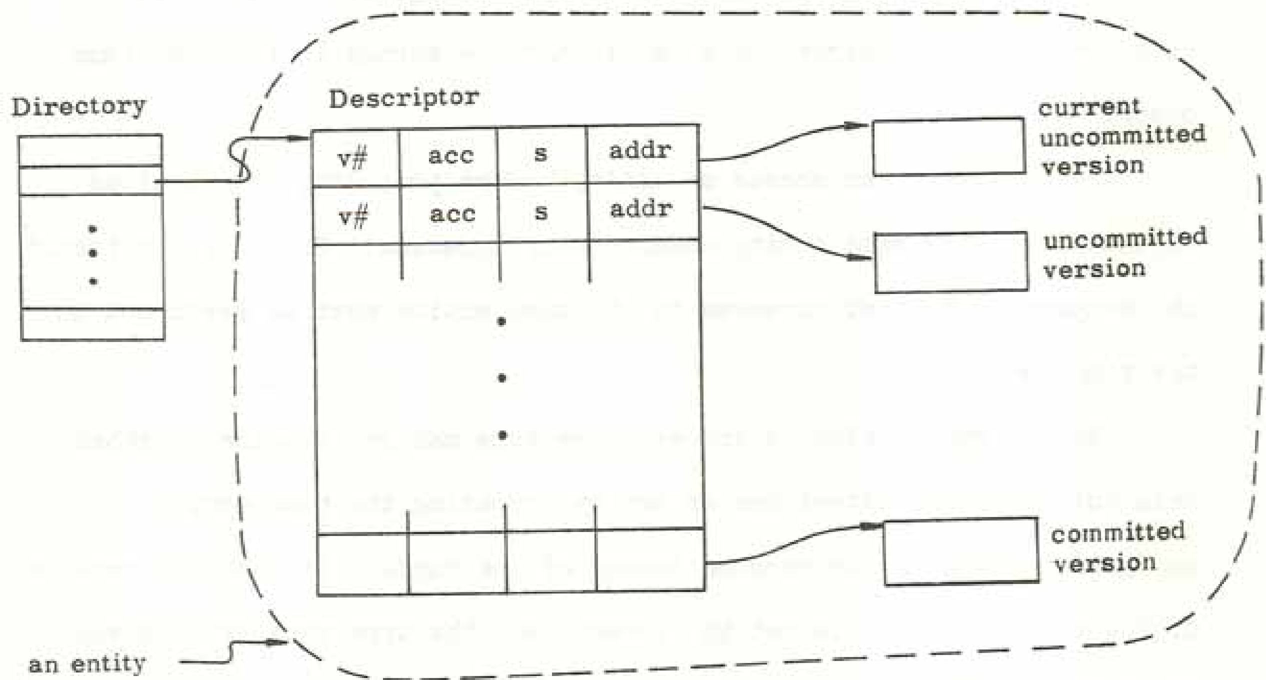


Figure 2 . Storage representation of entities

case of read-only access)

2) dependent : already accessed, but not yet prepared for commitment

3) prepared : prepared for commitment

4) committed : already committed

5) discarded : already discarded because of failures, sequence anomalies etc..

The addr field contains the address of the storage cell that contains this version [1]. Entries of a descriptor are sorted in the timestamp order.

A transaction can access an entity before preceding transactions that accessed the same entity commit their accesses. The only constraint on concurrency is that accesses to the same entity must be performed in the timestamp order.

When a readu action is invoked, the type manager examines whether this action is the latest one or not by comparing the timestamp ts assigned to it with the version number of the current (tentative) version of the entity. If it is not the latest one, the type manager discards the versions whose version numbers are greater than ts [2]. After that, it creates a new version (that is, creates a new descriptor entry) whose version number (v#), state (s), and access mode (acc) are "ts", "dirty",

---

(1) If the acc field is "read-only", this version shares the storage cell with the previous version.

(2) Of course, if it is older than the committed version, then the request is rejected. It is also rejected, however, if it is older than some version that is in the "prepared" state, since it is highly probable that such version will be committed. Moreover, according to the two-phase commit protocol, once a version is in the "prepared" state, only the transaction management process that originated the transaction may abort it (and thus discard the "prepared" version).

and "update". Then it returns the content of the immediate predecessor. If the requested action is the latest one, the type manager examines the state of the current version first. If the state of the current version is "dirty", creation of a new version is deferred (that is, the respective data management process is suspended) until the state of the current version becomes "dependent" (1). If the state is not "dirty", the type manager creates a new version and returns the content of the predecessor. Processing of a readr is similar to that of readu except that :

1) even if the action is not the latest one, it is not necessary to discard the versions that have greater version numbers. Instead, it is sufficient to insert a newly created, but outdated version immediately before these versions.
2) the state and the access mode are set to "dependent" and "read-only" respectively.
3) if the queue of waiting processes is not empty, then the type manager wakes up the process that has the smallest timestamp.

When a write action is invoked and the version created by the corresponding (preceding) readu action is not "discarded", the type manager acquires a free storage cell for the new version, writes the content of the buffer into it, and changes the state to "dependent". Otherwise, it deletes the invalid version and returns as such. If the

---

(1) It is possible to create a new version and permit an access immediately even if the state of the current version is "dirty". However this kind of concurrency proves fruitless since all accesses but one are eventually undone.

queue of waiting processes is not empty, the type manager wakes up the process that has the smallest timestamp. Each version, unless discarded, is deleted when it and a newer version are committed. At most one data management process can ever wait (directly) for the same version to change from "dirty" to "dependent" state. That is, although several processes may queue up as a result of the current version being "dirty", only one (the process with the smallest timestamp) is enabled. The next process on the queue will be awaken when the new version created by the first process becomes "dependent".

Note that it is possible that due to some serious failure the number of uncommitted versions will temporalily grow very large. In order to control the number of uncommitted versions, the number of entries in a descriptor is limited to some fixed value. Namely, if the number of entries reaches this value, the type manager of the corresponding object rejects new access requests, and the transaction that issued these requests have to try again at some later time.

4.4.1 Commitment of Transactions

The central principle of the commit protocol proposed in this paper is that no transaction can commit the versions it created until the states of all previous versions of these entities become "committed". The commit protocol is basically a two-phase commit protocol, but it is considerably different from others [9][16][23] because it must co-operate with the concept of multiple uncommitted versions.

In the first phase, a transaction management process sends prepare messages to all involved data management subsystems to confirm that versions it accessed are eligible to be committed. When the type manager

receives the request, it performs one of the following operations, depending on the state of the designated version and the state of the immediately preceding version :

1) changes the state of the designated version to "prepared" and returns as such if the designated version is "dependent" and the immediately preceding version is already committed.

2) suspends the data management process until the state of the immediately preceding version is changed to "committed" or "discarded" if the designated version is "dependent" and the immediately preceding version is not yet committed

3) returns as "discarded" and deletes the designated version if that version is "discarded".

If all return messages are "prepared", a commit point is established and then the second phase begins. Otherwise the transaction management process sends undo messages to all involved data management subsystems to abort the transaction. This is done by executing the set of actions (i.e. "send undo message" in this case) saved in the backout cache associated with this transaction.

The second phase must be completed no matter what happens. The transaction management process sends commit messages to all involved data management subsystems to commit the versions the transaction created. This is done by executing the set of actions (i.e. "send commit message" in this case) saved in the commit cache associated with this transaction. When the type manager receives the request, it:

1) changes the state of the designated version to "committed",

2) deletes the older committed version,

3) if a data management process is waiting for this version being committed, wakes it up.

Sending of a commit message is repeated until it is successfully processed. When the transaction management process confirms that all commit messages were successfully processed, the whole commitment process is completed.

4.4.2 Cascading of Backout

Backout of a transaction occurs either when it is aborted because of a failure[1] of any participants in the transaction processing or when it is involved in a sequence anomaly[2]. It is also backed out when transactions on which it depends are backed out. Suppose that T is the transaction that must be backed out. The backout of T causes all transactions that depend on T (in terms of the relation $<*$) to be backed out. The cascading of backout is done in the following way.

1) The transaction management process executing T sends undo messages to all involved data management subsystems. This is done by executing the set of actions saved in the transaction backout cache.

2) When a data management process receives the undo message, it requests the appropriate type manager to delete the version created by T by invoking an undo action. If the access mode of the version is "update", the type manager not only deletes the

---

(1) A failure that occurs during the second commit phase is excluded.

(2) See 4.3 Transaction Scheduling based on Timestamps.

version, but also changes the states of all newer versions (if any) to "discarded".

3) When a data management process invokes a write or prepare action on one of these discarded versions at the request of the transaction management process that created it, the type manager deletes it (i.e. the discarded version) and returns as "discarded". Then the data management process returns a "discarded" reply to the transaction management process that sent the write or prepare message to it.

4) Each transaction management process that has received a "discarded" reply must also be backed out by following the above procedures 1) 2) and 3).

5. CONCLUSION

The main goal of this paper was to develop a set of schemes that realize highly concurrent as well as reliable execution of transactions in a distributed database system. This goal was achieved by a combination of several new schemes : a consistent schedule of transactions based on timestamps, an object-oriented recovery using multiple uncommitted versions and a transaction(and object)-oriented recovery using a backout/commit cache. The consistent schedule based on timestamps minimizes the restraint of concurrency and, at the same time, eliminates the possibility of deadlock. The recovery scheme using multiple uncommitted versions coupled with a backout/commit cache realizes complete backout of transactions in case of failure without significantly sacrificing concurrency. It forms a striking contrast to other recovery schemes

proposed so far that suffer (to the considerable extent) from the serious restraint that "no transaction can access any entity until all previous transactions that accessed the entity are completed". The memory overhead induced by these new schemes may be sufficiently low since, at any point of time, each of a vast majority of entities is expected to have only one version. The processing overhead may also be acceptable since, in a normal situation where a sequence anomaly does not occur frequently, the overhead is almost comparable to one induced by careful replacement.

The secondary goal was to explore a method of building a distributed database system that achieves the above goal as a multi-level system. This paper suggested the necessity of using different schemes for concurrency control and recovery at different levels. In addition, this idea was actually applied to the implementation of the distributed updating algorithms.

Several extensions to these schemes will be possible. First, if a version is read-only, it is not necessary for a later version to wait until it becomes committed. Namely, a later version could be committed once the immediately preceding update version is committed. Second, if the older committed versions were not deleted, then an old read request could be processed (without creating a new version) long after subsequent versions have been committed; this is what the multiple versions of Reed's scheme [23] provide. Third, the assumption in Section 4.1 that any transaction performs an access to each entity at most once could be removed by slightly extending the timestamp mechanism. Namely, if each timestamp were composed of a transaction id part and an access id part, then it would be possible to decide whether two different accesses were

performed by the same transaction.

Finally, devising efficient and elegant language constructs that facilitate the implementation of various schemes proposed in this paper seems to be one of the most challenging future works.

APPENDIX A : LANGUAGE CONSTRUCTS

Since no language developed so far has constructs that sufficiently support distributed computation and recovery, it was necessary to devise them. The Concurrent Pascal [11] was adopted as a base language, with four kinds of new constructs added. The rest of Appendix A introduces these constructs briefly. Also, a few constructs of the Concurrent Pascal was slightly changed. However, it must be said that devising new language constructs is not the main objective of this paper and the language constructs added in this paper may be neither flawless nor very refined.

(1) Transaction

An atomic construct prefixed to the heading of a procedure (or function) means that the instance of this procedure (or function) is an atomic action (i.e. a transaction) and is implicitly assigned a backout/ commit cache.

(2) Exception handling

Constructs similar to those proposed by Liskov et al [17] were added for exception handling. Procedures and functions have headings that contain the information about the ways in which they may terminate. For example, procedure p has the following heading.

```
procedure p(formals) signals(conditions) ;
```

The syntax of the signal construct used to signal an exception to the calling routine is :

```
signal condition(args) ;
```

Also, the syntax of the exit construct used to raise an exception directly in the current routine is :

```
exit condition(args) ;
```

An exception handler is placed by means of except construct as follows.

```
statement except
          when condition_1(formals) : statement_1 ;
          when condition_2(formals) : statement_2 ;
            .
            .
        end ;
```

In addition, a reserved condition name others is used to handle "all remaining exceptions".

(3) Recovery

Several language constructs are added to support recovery. The recovery scheme using a backout/commit cache was added into the language because it is believed to be a considerably general scheme. The cache construct is used to save the action into the designated cache.

The syntax is :

```
cache action(args) into cache_name ;
```

where cache-name must be either commit-cache or backout-cache. The case action construct is used to execute the actions cached in the backout/commit cache. The syntax is :

```
case action in cache-name of
  action_1(formals) : do statement_1 ;
  action_2(formals) : do statement_2 ;
      .
      .
end ;
```

Each invocation of case action executes one action from the appropriate cache. The establish-commit-point construct puts a mark indicating that the commit point was already established on the backout/commit cache. On the other hand, the established? construct examines whether the commit point was already established or not.

(4) Message passing

This paper assumes the following language consturcts that handle

message passing between processes (over a network).[1] The syntax of the send construct is :

```
send action(args) to recipient timeout limit
     response_1(formals) : do statement_1 ;
     response_2(formals) : do statement_2 ;
          .
          .
     timeout : do statement_n
end ;
```

The case action construct and the reply construct are used to receive a message and to send a reply message, respectively, whose syntax is :

```
case action in message of
   action_1(formals) : do statement_1 ;
   action_2(formals) : do statement_2 ;
      .
      .
end ;
```

(1) These constructs are similar to those of Svobodova et al [25] and Feldman [7]. Although they sequentialize the individual steps of a transaction and thus introduce an unnecessary inefficiency, they make the program shown in Appendix B easy to understand. Devising more refined constructs that permit parallelism within a transaction is beyond the scope of this paper.

and

<u>reply</u> response(args) <u>to</u> recipient ;

The recipient designated in <u>send</u> or <u>reply</u> is usually a process name, but it can be a name of a process set, in which case the message is received by any member of the set.

APPENDIX B : DETAILED ALGORITHMS

Detailed algorithms for concurrent and reliable updates of a distributed database are presented here. It is assumed that the two-phase commit protocol is enforced by the operating system of each transaction management subsystem. Also, a few assumptions are made in order to make these algorithms simple. First, the disk storage management issues were omitted. Second, it is assumed that all descriptors are of the same fixed size, although this results in low space efficiency.

Sections B.1 and B.2 present the detailed algorithms of the data management subsystem and the transaction management subsystem, respectively.

B.1 Data Management Subsystem

Data Management Subsystem is composed of the following system components.

```
type descriptor = record
                    length : integer ;
                    list : array [1..max] of version_descriptor
                  end ;

type version_descriptor = record
                            v# : integer ;
                            acc : (read_only, update) ;
                            s : (dirty, dependent, prepared, discarded,
                                committed) ;
                            addr : integer
                          end ;

type process_queue = array [1..max] of queue ;

   % This paper assumes that the following standard procedures are defined
     for (single) queues.  This definition is somewhat different from that
     of Concurrent Pascal.

     atomic procedure delay(x : queue) signals(failure) ;

        The calling process is delayed in the queue x.

     atomic procedure continue(x : queue) signals(failure) ;

        If a process is waiting in the queue x, it resumes its execution
        of the monitor procedure.

     atomic procedure cancel(x : queue) signals(failure) ;

        If a process is waiting in the queue x, it is signaled that it
        has been canceled. %
```

```
+++++++++++++++++
+ ordered-queue +
+++++++++++++++++

type ordered_queue = class ;

   % defines an ordered set of single process queues (but does not contain
    the single queues themselves) : ordering is by the timestamp.(1)
    This ordering is realized without actually ordering the single
    process queues themselves since the queuing/dequeuing algorithm
    (that is implemented as class "ordered-queue") is separated from the
    queues (that are implemented as object "access-queue" of type
    "process-queue"). %

    atomic function entry arrival(timestamp : integer)
       returns(integer) signals(failure) ;

     % returns the index of the single queue to be used for the new
       arrival and assigns the designated timestamp to this single
       queue %

    atomic function entry departure returns(integer) signals(failure) ;

     % returns the index of the single queue that has the smallest
       timestamp and frees this index. %

    atomic function entry forced-departure(timestamp : integer)
       returns(integer) signals(failure) ;

     % returns the index of the single queue that has the designated
       timestamp and frees this index. %

    atomic function entry empty returns(boolean) signals(failure) ;

     % determines whether all of the single queues are empty %

    atomic function entry full returns(boolean) signals(failure) ;

     % determines whether all of the single queues are full %
```

(1) The idea of decomposing a multiprocess queue into a set of single queues and the mapping on them is due to Hansen [11].

```
atomic function entry find(timestamp : integer)
   returns(boolean) signals(failure) ;

  % determines whether the single queue that has the designated
    timestamp exists %

atomic procedure entry initialize(max-length : integer)
   signals(failure) ;


+++++++++++++++++++++
+ independent-queue +
+++++++++++++++++++++

type independent_queue = class ;

   % defines a set of independent single process queues (but does not
     contain the single queues themselves) %

     atomic function entry arrival(timestamp : integer)
        returns(integer) signals(failure) ;

       % returns the index of the single queue to be used for the new
         arrival and assigns the designated timestamp to this single
         queue %

     atomic function entry departure(timestamp : integer)
        returns(integer) signals(failure) ;

       % returns the index of the single queue that has the designated
         timestamp and frees the index %

     atomic function entry find(timestamp : integer)
        returns(boolean) signals(failure) ;

       % determines whether the single queue that has the designated
         timestamp exists %

     atomic procedure entry initialize(max-length : integer)
        signals(failure)  ;
```

```
+++++++++++++
+ directory +
+++++++++++++

type directory = monitor ;

    atomic function entry get returns(integer) signals(failure) ;

     % gets a free directory entry and returns its index %

    atomic procedure entry free(index : integer) signals(failure) ;

     % frees the designated directory entry %

    atomic procedure  entry set(index : integer ; descriptor_addr : integer)
       signals(failure) ;

     % sets the descriptor address in the designated directory entry %

    atomic function entry addr(index : integer) returns(integer)
       signals(failure) ;

     % returns the descriptor address set in the designated directory
       entry %

    atomic procedure entry intialize(unit : disk) signals(failure) ;
```

```
++++++++
+ disk +
++++++++

type disk = monitor ;

 % This monitor is assumed to be provided by the underlying operating
   system.  Also it is assumed that objects of type "disk" are non-
   recoverable and actions defined on them are atomic %

     atomic function entry get retruns(integer) signals(failure) ;

      % gets a free page and returns its absolute disk address, or returns
        "o" if not found %

     atomic procedure entry free(addr : integer) signals(failure) ;

      % frees a page identified by its absolute disk address %

     atomic procedure entry read(addr : integer ; var block : univ page)
        signals(failure) ;

      % reads a page identified by its absolute disk address %

     atomic procedure entry write(addr : integer ; var block : univ page)
        signals(failure) ;

      % writes a page identified by its absolute disk address %

     atomic procedure entry initialize signals(failure) ;
```

```
++++++++++
+ entity +
++++++++++

type entity = monitor ;

 % defines a database entity and controls access to it %

const max = ... ; qmax = ... ;

var des : descriptor ; des_addr : integer ;
    ordered : ordered_queue ;

     % defines an ordering of the elements in the access-queue, where the
       access-queue is a queue of data management processes waiting to
       create a new version %

    independent : independent_queue ;

     % together with the commit_queue, defines a queue of data management
       processes waiting for the immediately preceding versions to change
       either to "committed" or "discarded".  The independent defines a
       mapping between version numbers (timestamps) and positions of the
       corresponding processes on the commit-queue %

    access_queue, commit_queue : process_queue; ...

function older(v_no : integer) returns(integer) signals(failure) ;

 % returns the descriptor index of a valid version whose version number
   is closest to and older (less) than v_no, or returns "0" if not found %

function equal(v_no : integer) returns(integer) signals(failure) ;

 % returns the descriptor index  of a (valid or discarded) version whose
   version number is equal to v_no, or returns "0" if not found %

function newer(v_no : integer) returns(integer) signals(failure) ;

 % returns the descriptor index of a valid version whose version number
   is closest to and newer (greater) than v_no, or returns "0" if not
   found %

function current returns(integer) signals(failure) ;

 % returns the descriptor index of the latest valid version %
```

```
procedure insert(i : integer ; new_entry : version_descriptor)
   signals(failure) ;

 % inserts the new_entry between the (i-1)th entry and the (i)th entry of
   the descriptor and increases des.length by 1 %

procedure delete(i : integer) signals(failure) ;

 % deletes the (i)th entry from the descriptor and decreases des.length
   by 1 %

procedure discard(ind : integer) signals(failure) ;

 % discards all versions whose descriptor indices are greater than ind %

  var i : integer ;

  begin
   i : = current ;
   repeat
     if des.list[i].s<>discarded then
        begin
          with des.list[i] do
          begin
            if (acc = update)and(s<>dirty) then
            cache Free(unit, addr) into commit-cache ;
            if (s = dirty)and(not ordered.empty) then
               continue(access_queue[ordered.departure])
            else if independent.find(v#) then
               continue(commit_queue[independent.departure(v#)]) ;
            s : = discarded
          end
        end ;
      i : = i - 1 ;
    until i = ind
   end
   except when others : signal failure end ;
```

```
procedure commit_action ;

 % performs the commit phase of the logical action on the datbase entity,
   i.e. executes the actions cached in the commit cache that is associated
   with the logical action %

  var end_of_cache : bollean ; ...

  begin
      .
      .
  while not end_of_cache do
   begin
        .
        .
     case action in commit-cache of
       Free(var un : disk ; add : integer) :
         do un.free(add) ;
              .
              .
              .
              .
     end ;
       .
       .
   end ;
    .
    .
  end
  except when others : commit_action end ;
```

```
procedure backout_action ;

 % backs out the logical action on the database entity, i.e. executes the
   actions cached in the backout cache that is associated with the logical
   action %

  var end-of-cache : boolean ; ...
  begin
      .
      .
    while not end_of_cache do
       begin
           .
           .
         case action in backout-cache of
           Write(var un : disk ; add : integer ; var bl : univ page) :
           do un.write(add, bl) ;
                .
                .
                .
         end ;
           .
           .
       end ;
         .
         .
       unit.read(des-addr, des)
       % restores the value of "des" to the one it was in before the
         interrupted action was invoked %
  end
  except when others : backout_action end;
```

```
atomic procedure entry readu(timestamp : integer ; var block : univ page)
   signals(obsolete, congestion, discarded, canceled, duplicate,
   misused, failure) ;

  % reads the value of the immediately previous version of the designated
    entity and creates a new version.  Used for an update access %

   var k, l, m : integer ; olddess : descriptor ; new : version-descriptor ;
   begin
     olddes : = des;
     l : = older(timestamp) ;
     k : = equal(timestamp) ;
     m : = newer(timestamp) ;

     if k<>0 then
       if des.list[k].acc = update then              #1
         if des.list[k].s = discarded then
           begin
             delete(k) ;
             cache Write(unit, des_addr, olddes)
                   into backout-cache ;
             unit.write(des_addr, olddes) ;
             establish-commit-point ;
             commit_action ;
             signal discarded
           end
           % deletes the already discarded version %
         else
           begin
             unit.read(des.list[k].addr, block) ;
             % reads the designated entity %
             signal duplicate
           end
      else signal misused
    else if ordered.find(timestamp) then
        signal canceled ;
     % detects duplicate or wrong requests %

    if (l = 0)or((m<>0)and(des.list[m].s = prepared))
       then signal obsolete ;

    while des.list[l].s = dirty do
      begin
        if ordered.full then signal congestion ;
        delay(access-queue[ordered.arrival(timestamp)])
          except when canceled : signal canceled end ;
        l : = older(timestamp)
      end ;
     % waits until the immediately previous version becomes
       dependent" %
```

```
    if des.length = max then signal congestion ;
    if l<>current then discard(l) :      #2
      with new do
    begin
      v# : = timestamp ;
      acc : = update ;        #3
      s : = dirty ;        #4
      addr : = des.list[l].addr
    end ;
    insert(l, new) ;
    cache Write(unit, des_addr, olddes)
       into backout-cache ;
    unit.write(des_addr, des) ;
    % updates the descriptor %

    unit.read(des.list[l].addr, block) ;
    % reads the designated entity %

    establish-commit-point ;
    commit_action
  end
  except when others : begin backout-action ; signal failure end end ;


atomic procedure entry readr(timestamp : integer ; var block : univ page)
   signals(obsolete, congestion, discarded, canceled, duplicate, misused,
   failure) ;

 % used for a read-only access.  Same as readu except that :
  #1 --- if des.list[k].acc = read-only then
  #2 --- null
  #3 --- acc : = read_only ;
  #4 --- s : = dependent ;
  #5 --- if (l = current)and(not ordered.empty) then
             continue(access_queue[ordered.departure]) ; %
```

```
atomic procedure entry write(timestamp : integer ; var block : univ page)
   signals(discarded, duplicate, misused, failure) ;

 % updates the version created by the associated readu action %

  var k : integer ; olddes : descriptor ;
  begin
    olddes : = des ;
    k : = equal(timestamp) ;
    if k = 0 then signal misused ;
    if des.list[k].s<>dirty then
      if des.list[k].s<>discarded then

        if des.list[k].acc = update then
          signal duplicate
        else signal misused
        % detects duplicate or wrong requests %

      else

        begin
          delete(k) ;
          cache Write(unit, des-addr, olddes)
                into backout-cache ;
          unit.write(des_addr, des) ;
          establish-commit-point ;
          commit_action ;
          signal discarded
        end ;
        % deletes the version if already discarded %

     des.list[k].addr : = unit.get ;
     cache Free(unit, des.list[k].addr)
       into backout-cache ;
     unit.write(des.list[k].addr, block) ;
     % writes the designated entity %

     des.list[k].s : = dependent ;
     cache Write(unit, des-addr, olddes)
       into backout-cache ;
     unit.write(des_addr, des) ;
     % updates the descriptor %

     if not ordered.empty then
        continue(access-queue[ordered.departure]) ;
     establish-commit-point ;
     commit_action
  end
  except when others : begin backout_action ; signal failure end end ;
```

```
atomic procedure entry prepare(timestamp : integer)
   signals(discarded, canceled, duplicate, misused, failure) ;

 % confirms that the designated version is eligible to be committed %

  var k, l : = integer ; olddes : descriptor;
  begin
    olddes : = des ;
    k : = equal(timestamp) ;
    l : = older(timestamp) ;

    if k = o then signal misused ;
    if (des.list[k].s = prepared)or(des.list[k].s = committed) then
       signal duplicate ;
    if des.list[k].s = dirty then
       signal misused ;
    if independent.find(timestamp) then
       signal canceled ;
    % detects duplicate or wrong requests %

    while(des.list[l].s<>committed)and(des.list[k].s<>discarded)
    do
      begin
        delay(commit_queue[independent.arrival(timestamp)])
          except when canceled : signal canceled end ;
        k : = equal(timestamp) ;
        l : = older(timestamp)
      end ;
     % waits until the immediately previous version becomes "committed" %

    if des.list[k].s = discarded then
      begin
        delete(k) ;
        cache Write(unit, des_addr, olddes)
          into backout-cache ;
        unit.write(des-addr, des) ;
        establish-commit-point ;
        commit_action ;
        signal discarded
      end ;
      % deletes the version if already discarded %

    des.list[k].s : = prepared ;
    cache Write(unit, des_addr, olddes)
      into backout-cache ;
    unit.write(des_addr, des) ;
    % updates the descriptor %

    establish-commit-point ;
    commit_action
  end
  except when others : begin backout_action ; signal failure end end ;
```

```
atomic procedure entry commit(timestamp : integer)
   signals(duplicate, failure) ;

  % discards the previous version and makes the immediately
    subsequent version (if any) eligible for commitment %

   var k, l, m, nts : integer ; olddes : descriptor ;
   begin
     olddes : = des;
     k : = equal(timestamp) ;
     l : = older(timestamp) ;

     if des.list[k].s = committed then
       signal duplicate ;
     % detects duplicate requests.  Wrong requests do not occure
       because the two-phase commit protocol is enforced by the
       operating system of the transaction management subsystem. %

     if des.list[k].acc = update then
       cache Free(unit, des.list[l].addr)
         into commit-cache ;
     % frees the previous version at the end of the action %

     delete(l) ;
     des.list[k].s : = committed ;
     cache Write(unit, des_addr, olddes)
       into backout-cache ;
     unit.write(des_addr, des) ;
     % updates the descriptor %

     if k<>current then
       begin
         m : = newer(timestamp) ;
         nts : = des.list[m].v# ;
         if independent.find(nts) then
           continue(commit-queue[independent.departure(nts)])
       end ;
     % makes the immediately subsequent version eligible for commitment %

     establish-commit-point ;
     commit_action
   end
   except when others : begin backout_action ; signal failure end end ;
```

```
atomic procedure entry undo(timestamp : integer)
   signals(duplicate, failure) ;

 % discards the designated version and all subsequent versions
   that depend on this version %

  var k, l, m, nts : integer ; olddes : descriptor ;
  begin
    olddes : = des ;
    l : = older(timestamp) ;
    k : = equal(timestamp) ;

    if k = 0 then
      if ordered.find(timestamp) then
        cancel(access_queue[ordered.forced_departure(timestamp)])
      else if (l = current)and(des.list[l].s<>dirty)
              and(not ordered.empty) then
              continue(access_queue[ordered.departure])
           else signal duplicate
        % k = 0 means that readu/readr request for this version is
          waiting on the access-queue or that excution of readu/
          readr has failed or that undo has been already excuted. %

    else  % i.e. if this version exists %
      begin
        if des.list[k].s<>discarded then
          begin

            if des.list[k].acc = update then
              begin
                if k = current then
                  if not ordered.empty then
                    continue(access_queue[ordered.departure])
                else discard(k) ;
                cache Free(unit, des.list[k].addr) into commit-cache
              end
            % discards the newer versions, and frees this version at
              the end of the action %

            else  % i.e. if acc = read-only %
              if k<>current then
                begin
                  m : = newer(timestamp) ;
                  nts : = des.list[m].v# ;
                  if (des.list[l].s = committed)and
                        (independent.find(nts)) then
                     continue(commit_queue[independent.departure(nts)])
                end ;
             % simply deletes this version %
             if independent.find(timestamp) then
               cancel(commit_queue[independent.departure(timestamp)])
           end ;
```

```
        delete(k) ;
        cache Write(unit, des_addr, olddes)
          into backout-cache ;
        unit.write(des_addr, des)
        % updates the descriptor %

      end ;
    establish-commit-point ;
    commit_action
  end
  except when others : begin backout_action ; signal failure end end ;
```

```
atomic procedure entry initialize(unit : disk ; map : directory)
   signals(failure) ;

 % initializes the monitor %

  var index : integer ;
  begin
    des.length : = 1 ;
    with des.list[1] do
      begin
        v# : = 0 ;
        acc : = update ;
        s : = committed ;
        addr : = unit.get ;
        cache Free(unit, addr) into backout-cache
        % gets a page for the initial version %
      end ;

    des_addr : = unit.get ;
    cache Free(unit, des_addr) into backout-cache ;
    % gets a page for the descriptor %

    unit.write(des_addr, des) ;
    % sets up the descriptor %

    index : = map.get ;
    cache Free(map, index) into backout-cache ;
    map.set(index, des_addr) ;
    % sets up the directory entry %

    cache Initialize(ordered, qmax) into commit-cache ;
    cache Initialize(independent, qmax) into commit-cache ;
    establish-commit-point ;
    commit-action
  end
  except when others : begin backout_action ; signal failure end end ;
```

```
++++++++++++++++++++++++++++
+ data-management-process +
++++++++++++++++++++++++++++

type data_management_process = process ;

% A data management process, when it receives an access request (message)
  from one of transaction management subsystems, performs a designated
  action on a designated entity %

  var block : univ page ; ... ;
  begin
    cycle
      case action in message of
        Readu(ver item : entity ; timestamp : integer) : do
          begin
            item.readu(timestamp, block)
            except
              when obsolete : begin
                                reply obsolete ;
                                exit done
                              end ;
              when congestion : begin
                                  reply congestion ;
                                  exit done
                                end ;
              when discarded : begin
                                 reply discarded ;
                                 exit done
                               end ;
              when duplicate : begin
                                 reply duplicate(block) ;
                                 exit done
                               end ;
              when misused : begin
                               reply misused ;
                               exit done
                             end ;
              when failure : begin
                               reply failure ;
                               exit done
                             end ;
              when canceled : exit done
              % No reply is necessary because the readu request has
                been already undone by the requestor %
            end ;
            reply normal(block)
          end except when done : end ;

        Readr(var item : entity ; timestamp : integer) : do
          % same as Readu %
```

```
Write(var item : entity ; timestamp : integer ; bl : univ page) : do
  begin
    item.write(timestamp, bl)
    except
      when discarded : begin
                         reply discarded ;
                         exit done
                       end ;
      when duplicate : begin
                         reply duplicate ;
                         exit done
                       end ;
      when misused : begin
                       reply misused ;
                       exit done
                     end ;
      when failure : begin
                       reply failure ;
                       exit done
                     end ;
    end ;
    reply normal
  end except when done : end ;

Prepare(var item : entity ; timestamp : integer) : do
  begin
    item.prepare(timestamp)
    except
      when discarded : begin
                         reply discarded ;
                         exit done
                       end ;
      when duplicate : begin
                         reply duplicate ;
                         exit done
                       end ;
      when misused : begin
                       reply misused ;
                       exit done
                     end ;
      when failure : begin
                       reply failure ;
                       exit done
                     end ;
      when canceled : exit done
      % No reply is necessary because the prepare request has
        been already undone by the requestor %
    end ;
    reply normal
  end except when done : end ;
```

```
        Commit(var item : entity ; timestamp : integer) : do
          begin
            item.commit(timestamp)
            except
              when duplicate : begin
                                 reply duplicate ;
                                 exit done
                               end ;
              when failure : begin
                               reply failure ;
                               exit done
                             end ;
            end ;
            reply normal
          end except when done : end ;

        Undo(var item : entity ; timestamp : integer) : do
          begin
            item.undo(timestamp)
            except
              when duplicate : begin
                                 reply duplicate ;
                                 exit done
                               end ;
              when failure : begin
                               reply failure ;
                               exit done
                             end ;
            end ;
            reply normal
          end except when done : end
      end ;
    end ;
      .
      .
  end ;
```

B.2 Transaction Management Subsystem

Transaction Management Subsystem is composed of the following system components.

```
+++++++++++++++++++++++++++++++++++
+ transaction_management_process +
+++++++++++++++++++++++++++++++++++

type transaction_management_process = process ;

  % A transaction management process executes database transactions,
    one at a time, by communicating with data management subsystems %

const limit = ... ; period = ... ; max = ... ;
var  input_1, ... , input_p, output_1, ... , output_p : univ page ;

   % It is assumed that variables dms_1, ... , dms_n of type "data
     management subsystem" and item_1, ... , item_m of type "entity"
     are defined as system components outside of the transaction
     management processes. %

atomic procedure wait_for_a_while(time : integer) signals(failure) ;

  % suspends the requesting process for the designated period %

atomic function new_timestamp returns(integer) signals(failure) ;

  % acquires a timestamp from the local clock %
```

```
Procedure commit_transaction ;

  % executes the commit-phase of the transaction, i.e. executes the
    actions cached in the commit cache that is associated with the
    transaction %

  var m : integer ; end_of_cache, retry_action : boolean ; ...
  begin
      .
      .
    while not end_of_cache do
    begin
      .
      .
      case action in commit-cache of
        Send(procedure remote_action ;
             var dest : data_management_subsystem) : do
          begin
            m : = 1 ;
            repeat
              retry_action : = false ;
              send remote_action to dest timeout limit
                normal, duplicate : do ;
                timeout : do begin
                                if m = max then exit failure
                                else begin
                                       m : = m + 1 ;
                                       retry_action : = true
                                     end ;
                              end ;
                failure : do exit failure
                          % i.e. commit_transaction is re-entered %
              end ;
            until not retry_action
          end ;
            .
            .
            .
      end ;
        .
    end ;
      .
      .
  end
  except when others : commit_transaction end ;
```

```
procedure backout_transaction ;

  % backs out the transaction, i.e. executes the actions cached in the
    backout cache that is associated with the transaction %

  var m : integer ; end_of_cache, retry_action : boolean ; ...
  begin
      .
      .
    while not end_of_cache do
    begin
        .
        .
      case action in backout-cache of
        Send(procedure remote_action ;
             var dest : data_management_subsystem) : do
          begin
            m : = 1 ;
            repeat
              retry_action : = false ;
              send remote_action to dest timeout limit
                normal, duplicate : do ;
                timeout : do begin
                               if m = max then exit failure
                               else begin
                                      m : = m + 1 ;
                                      retry_action : = true
                                    end
                             end ;
                failure : do exit failure
                            % i.e. backout_transaction is re_entered %
              end
            until not retry_action
          end ;
            .
            .
            .
      end ;
        .
        .
    end ;
      .
      .
      .
   end
   except when others : backout_transaction end ;
```

```
atomic procedure database_transaction signals(failure) ;

 % executes a database transaction %

  var ts, m : integer ;
      restart_transaction, retry_action : boolean ; ...
  begin
    repeat
      begin
        restart_transaction : = false ;
        ts : = new_timestamp ;
          .
          .
        cache Send(undo(item_j, ts), dms_i)
           into backout-cache ;
        cache Send(commit(item_j, ts), dms_i)
           into commit-cache ;
        m : = 1 ;
        repeat
          retry_action : = false ;
          send readu(item_j, ts) to dms_i timeout limit
            normal(var input_k : univ page),
               duplicate(var input_k : univ page) : do ;
            obsolete, discarded : do begin
                                       backout_transaction ;
                                       restart_transaction : = true ;
                                       exit restart
                                     end ;
            congestion : do begin
                              backout_transaction ;
                              wait_for_a_while(period) ;
                              restart_transaction : = true ;
                              exit restart
                            end ;
            timeout : do begin
                             if m = max then signal external_failure
                             else begin
                                    m : = m + 1 ;
                                    retry_action : = true
                                  end
                           end ;
            misused : do signal failure ;
                         % indicates that the procedure " database_
                           transaction" is erroneous %
            failure : do signal external_failure
                         % indicates that dms_i or the communication
                           line has failed %
          end
        until not retry-action ;
            .
            .
      % reads entities that belong to read set of the transaction %
```

```
      .
      .
  m : = 1 ;
  repeat
    retry_action : = false ;
    send write(item_j, ts, output_k) to dms_i timeout limit
      normal, duplicate : do ;
      discarded : do begin
                      backout_transaction ;
                      restart_transaction : = true ;
                      exit restart
                    end ;
      timeout : do begin
                    if m = max then signal external-failure
                    else begin
                           m : = m + 1 ;
                           retry_action : = true
                         end
                  end ;
      misused : do signal failure ;
      failure : do signal external-failure
    end
  until not retry_action ;
      .
      .
% updates entities that belong to the update set of the
  transaction %
      .
      .
% for each entity in the read set of the transaction %
  m : = 1 ;
  repeat
    retry_action : = false ;
    send prepare(item_j, ts) to dms_i timeout limit
      normal, duplicate : do ;
      discarded : do begin
                      backout_transaction ;
                      restart_transaction : = true ;
                      exit restart
                    end ;
      timeout : do begin
                    if m = max then signal external_failure
                    else begin
                           m : = m + 1 ;
                           retry_action : = true
                         end
                  end ;
      misused : do signal failure ;
      failure : do signal external_failure
    end
  until not retry_action ;
```

```
          .
          .
      end except when restart : end
    until not restart_transaction ;
    establish-commit-point ;
    commit_transaction
  end
    except when failure, others : begin
                                    (notification to the user)
                                    backout_transaction ;
                                    signal failure
                                  end ;
           when external-failure : begin
                                     (notification to the user)
                                     backout_transaction
                                   end
    end ;
```

```
begin
  cycle
     database_transaction
  end
    except when failure : end
end ;
```

REFERENCES

[1] T.Anderson, P.A.Lee and S.K.Shrivastava, "A model of recoverablity in multi-level systems," IEEE Trans. Software Eng., vol.SE-4, pp. 486-494, Nov. 1978.

[2] P.A.Bernstein, D.W.Shipman, J.R.Rothnie and N.Goodman, "The concurrency control mechanism of SDD-1 : A system for distributed database (The general case)," Computer Corp. America, Cambridge, MA, Tech. Rep. CCA-77-09, Dec. 1977.

[3] L.A.Bjork, "Recovery scenario for a DB/DC system," in Proc. 1973 ACM Nat. conf., Aug. 1973.

[4] L.A.Bjork, "Generalised audit trail requirements and concepts for database applications," IBM Syst. J., vol.14, pp. 229-245, July 1975.

[5] C.T.Davies, "Recovery semantics for a DB/DC system," in Proc. 1973 ACM Nat. Conf. Aug. 1973.

[6] K.P.Eswaran, J.N.Gray, R.A.Lorie and I.L.Traiger, "The notions of consistency and predicate locks in a database system," Commun. Ass. Comput. Mach., vol.19, pp. 624-633, Nov. 1976.

[7] J.A.Feldman, "A programming methodology for distributed computing (among other things)," Department of Computer Science, Univ. Rochester, NY, Tech. Rep. 9, Jan. 1977.

[8] J.N.Gray, R.A.Lorie, G.R.Putzolu and I.L.Traiger, "Granularity of locks and degree of consistency in a shared database," IBM Research Rep. RJ 1654, Sept. 1975.

[9] J.N.Gray, "Notes on database operating systems," in Lecture Notes in Computer Science, vol.60, Springer-Verlag, New York, 1978.

[10] J.N.Gray, Talk at Laboratory for Computer Science, M.I.T., Mar. 1978.

[11] P.B.Hansen, The architecture of concurrent programs, Prentice-Hall, Inc. NJ. 1977.

[12] C.A.R.Hoare, "Monitors : an operating system structuring concept," Commun Ass. Comput. Mach. vol.17, pp. 549-557, Oct. 1974.

[13] J.J.Horning, H.C.Lauer, P.M.Melliar-Smith and B.Randell, "A program structure for error detection and recovery," in Lecture Notes in Computer Science, vol.16, Springer-verlag, New York, 1974.

[14] L.Lamport, "A new solution of Dijkstra's concurrent programming problem," Commun. Ass. Comput. Mach., vol.17, pp. 453-455, Aug. 1974.

[15] L.Lamport, "Time, clocks and ordering of events in a distributed system," Commun. Ass. Comput. Mach., vol.21, pp. 558-565, July 1978.

[16] B.Lampson and H.Sturgis, "Crash recovery in a distributed data storage system," Computer Science Laboratory, Xerox Palo Alto Research Center, CA, 1976.

[17] B.H.Liskov and A.Snyder, "Structured exception handling," Laboratory for Computer Science, M.I.T., Cambridge, MA, Computation Structures Group Memo 155, Dec. 1977.

[18] D.B.Lomet, "Process structuring, synchronization and recovery using atomic actions," SIGPLAN Notices, vol.12, pp. 128-137, Mar. 1977.

[19] R.A.Lorie, "Physical integrity in a large segmented database," ACM Trans. Database Syst., vol.2, pp. 91-104, Mar. 1977.

[20] W.A.Montgomery, "Robust concurrency control for a distributed information system," Laboratory for Computer Science, M.I.T., Cambridge, MA, TR-207, Dec. 1978.

[21] B.Randell, "System structure for software fault tolerance," IEEE Trans. Software Eng., vol.SE-1, pp. 220-232, June 1975.

[22] B.Randell, P.A.Lee and P.C.Treleaven, "Reliability issues in computing system design," ACM Comput. Surveys , vol.10, pp. 123-165, Jun 1978.

[23] D.P.Reed, "Naming and synchronization in a decentralized computer system," Laboratory for Computer Science, M.I.T., Cambridge, MA, TR-205, Sept. 1978.

[24] R.L.Rivest, V.R.Pratt, "The mutual exclusion problem for unreliable processes," Laboratory for Computer Science, M.I.T., Cambridge, MA, TM-84, Apr. 1977.

[25] L.Svobodova, B.H.Liskov and D.D.Clark, "Distributed computer systems : structure and semantics," Laboratory for Computer Science, M.I.T., Cambridge, MA., TR-215, Mar. 1979.

[26] R.H.Thomas, "A solution to the update problem for multiple copy databases which uses distributed control," Bolt Beranek and Newman Inc., Cambridge, MA, BBN Rep. 3340, July 1975.

[27] J.S.M.Verhofstad, "Recovery and crash resistance in a filing system," in Proc. 1977 ACM SIGMOD Int. Conf. on Management of Data, 1977.

[28] J.S.M.Verhofstad, "On multi-level recovery : an approach using partially recoverable interfaces," Computing Laboratory, Univ. Newcastle upon Tyne, UK, Tech. Rep. 100, May 1977.

[29] J.S.M.Verhofstad, "Recovery techniques for database systems," ACM Comput. Surveys, vol.10, pp. 167-195, June 1978.