M.I.T. LABORATORY FOR COMPUTER SCIENCE
Computer Systems Research Division

November 22, 1978
Request for Comments No. 167

CONCURRENT AND RELIABLE UPDATES OF DISTRIBUTED DATABASES

by Akihiro Takagi

I. 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 the components (called entities hereafter) such as read and write, sequences of actions are grouped to form transactions, which are units of consistency. Each transaction transforms the database from a consistent state to a new consistent state[7][9][10]. Therefore transactions are also units of recovery. (The concept of a transaction is similar to the concept of a sphere of control[4][6], the concept of a conversation [23], or more generally, the concept of an atomic action[24][20]. Also it has close relation to the concept of a monitor[13][12].)

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[10]. To prevent these anomalies from occurring, it is usually required that ,for a 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.

Gray et al[7][9][10] 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[11]. In addition, it is subject to deadlock.

The consistent schedule adopted in SDD-1 directly controls the ordering of actions on each entity by utilizing timestamps assigned to transactions and the information about the necessary degree of synchronization that is acquired from the pre-analysis

of transactions[3]. The point is to eliminate unnecessary invocation of the synchronization mechanism and thereby reduce the run-time overhead. However SDD-1 does not seem to pursue maximum concurrent execution of transactions that need some degree of synchronization. In addition, the concurrency control mechanism of SDD-1 described in [3] does not assure the recoverability of the database from various kinds of failures. (1)

Reed[25] used a similar timestamp mechanism to implement a consistent schedule of transactions. He also introduced the concept of versions and tokens of mutable objects to reinforce the availability and recoverability of the database. Although this concept will make it easier to cope with availability and recoverability issues, the degree of concurrency will be much the same as the above approaches.

Montgomery's scheme[22] allows high degree of concurrency. But the interface to the database manager seems to become very different from conventional ones under his scheme since, for example, a read action returns multiple possible values of the data and the user has to perform computation on all of them.

The scheme for concurrent and reliable updates proposed in this paper is somewhat related to that of Reed[25] and Bernstein et al[3] in the sense that it also uses a timestamp mechanism,

---

(1)
However some work to built recoverability into SDD-1 is in progress.

although the scheduling policy is different. Moreover, the paper develops the concept of multiple uncommitted versions of objects. These uncommitted versions are somewhat similar to the tokens of Reed. However, there exists a crucial difference between the concept of uncommitted versions and the concept of tokens. Namely, in Reed's scheme, an object generally can not have more than one token at a time, whereas it can have multiple uncommitted versions in the scheme proposed here. This difference seems to result from the differences in motivation. The objective of the scheme discussed in this paper is to achieve almost maximum possible degree of concurrency as well as recoverability (although stress is mainly placed on crash resistence). (1) Therefore this scheme and Reed's scheme are almost disjoint and supplementary rather than competitive. In addition, it is proposed to use different schemes for concurrency control and recovery at different levels of the system although more than one level may share the same scheme. No other paper published so far explicitly mentioned this issue (especially in the context of distributed databases). Randell[23][24], Verhofstad[29] and Anderson et al[1] treated a different aspect of the recovery issue in a multi-level system.

---

(1)
To cope with more serious media failures such as a head crash, dust on magnetic media etc. (including serious failures of the operating system), we have to resort to such schemes as incremental dump, long-term checkpoint, differential files etc.[5][10][21][30] or multiple versions proposed by Reed.

2. ATOMIC ACTIONS --- A GENERAL DISCUSSION

This section discusses the characteristics of atomic actions in general (rather than transactions) and the shemes to implement them. The representation of an atomic action generally consists of a set of underlying objects and a sequence of (possibly atomic) actions on them. An example of an atomic action is a transaction in which case underlying objects are database entities accessed by the transaction.

Atomic actions 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[24][20]. These characteristics must be preserved even if several atomic actions are invoked concurrently or any kinds of failures occur during the processing of atomic actions. Therefore, our primary concerns are to ensure that:

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

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

Consistent Schedule of Atomic Actions

A schedule S for a set of atomic actions A1, A2, ..., An defines the binary relation < such that A1<A2 if atomic action A1 performs action a1 on object e at some step in S and atomic action A2 performs action a2 on e at a later step in S. 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 atomic actions must be such that the relation <* is a partial order[7][9][10] ". 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

The first one completely serializes any pair of atomic actions between which the relation < exists. This occurs in most conventional operating systems where (at least a part of) supervisor programs are executed serially. (1) 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 atomic actions can be executed concurrently even in this scheme).

The second one was explored by Gray et al[7][9][10] and is widely used in many database systems. Although an atomic action has to lock underlying objects in this scheme, it is possible to ensure that any legal schedule is consistent if each atomic

---

(1)
Such programs are called " serially reusable programs " in IBM OS/360 and were renamed " monitor procedures " by Hoare[13].

action observes a two-phase lock protocol[7] (i.e. an atomic action can not request new locks after releasing a lock). Therefore the degree of concurrency achieved by this scheme is supposed to be better than the first one. But it is not the best since the two-phase restriction is only a sufficient condition[7]. It is also subject to deadlock.

The third one is based on the observation that a consistent schedule of atomic actions is merely a sequencing of lower level actions performed on the underlying objects by these atomic actions such that the relation <* be a partial order. This sequencing is directly controled using a timestamp mechanism rather than a lock mechanism. Namely, each atomic action is assigned a globally unique timestamp, and thereby all atomic actions are totally ordered. The manager of each object schedules actions on the object in the timestamp order of atomic actions that request these actions. This scheme was used by Bernstein et al[3] and Reed[25]. (1) This distributed (i.e. per-object-based) scheduling algorithm guarantees that, for any pair of atomic actions A1 and A2, A1 < A2 if and only if the timestamp assigned to A1 is smaller than the timestamp assigned to A2. Therefore the relation <* defined by this scheduling algorithm is a partial order that can be extended to the timestamp order. Since this scheme imposes no more constraints

---

(1)
The schedule based on timestamps is somewhat similar to the methods that was devised to solve the mutual exclusion problem by Lamport[15], Rivest et al[26] etc..

than necessary (i.e. the relation <* be a partial order), it ensures the maximum degree of concurrency. In addition, it is deadlock free since <* is an acyclic relation. However, this scheme requires a non-trivial algorithm that ensures that atomic actions are eventually scheduled in the timestamp order even if components of the system fail or sequence anomalies occur because of communication delays, processing delays etc.. (1) Such an algorithm may induce a greater overhead than previous schemes do.

All or Nothing Property of Atomic Actions

In order to prevent a temporary inconsistency from becoming permanent when a failure is encountered , it must be always possible to decide whether or not to complete any outstanding atomic actions and perform the alternative thus selected. Unfortunately there exists no finite length protocol which ensures that each atomic action is either completed or backed out in a distributed system in which nodes or communication lines may fail at any time[22]. 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

---

(1)
For example, suppose that both of A1 and A2 performe actions on two objects O1 and O2. Then it may happen that O1 gets a request from A1 before that from A2, but O2 gets requests in the reverse order.

(2)
Commitment of an atomic action means deletion of the recovery data for this action. Therefore, after the atomic action is committed, there is no way of undoing the action.

(2) protocol that was first advocated by Lampson et al[17] and was refined by Gray[10] and Reed[25]. In the two-phase commit protocol, a commit point is established after the first phase of commitment is successfully completed. If something goes wrong before the commit point, the atomic action is backed out. On the other hand, the atomic action is completed no matter what happened after the commit point (it may cause an infinite delay).

(1) Backing out

In order to back out an atomic action:

1) the states of the underlying objects that were accessed by the atomic action so far must be restored to the states they were in when the atomic action was invoked
2) the information flow from the atomic action must be undone and all other atomic actions affected by this information flow must also be backed out (this is called cascading of backouts[10] or domino effect[24])

(a) Recovery of objects

Recovery of underlying objects accessed by an atomic action consists of two phases as follows.

1) deciding which objects the atomic action accessed
2) restoring the states of these objects to ones they were in before the atomic action was invoked

In this paper, a recovery scheme used to implement the

first phase is called a process-oriented scheme because the first phase associates objects on which a given atomic action (it may be a process) performed actions with each other. On the other hand, a recovery scheme used to implement the second phase is called an object-oriented scheme because the second phase deals with a history of actions performed on a given object by different atomic actions.

A process-oriented scheme basically remembers the identifiers of the objects accessed by the atomic action, 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)[5][10] and a recovery cache[14] can be used as a process-oriented scheme.

Before discussing object-oriented schemes, it is necessary to give a definition to recoverability of objects. Objects are classified into two categories: recoverable objects and non-recoverable objects. A recoverable object is one whose manager provides recovery. Namely, the manager saves recovery data and restores an earlier state when something goes wrong. On the other hand, a non-recoverable object is one whose manager does not provide recovery. The manager of a

non-recoverable object does not save recovery data. (1) But actually, there exist good possibilities that the user (the atomic action) can undo the effect of the earlier action by invoking an inverse or a compensating action[4] (if any) if he keeps sufficient recovery data.

In an object-oriented recovery scheme for a recoverable object, the manager of the object remembers each state of the object and identifiers of atomic actions (2) which depend on that state. How fully to remember the state changes depends on the kind of a failure to cope with and the kind of recovery (3) to provide. As was stated earlier, this paper mainly discusses crash resistance[28][30]. Crash resistance is provided if, after some kind of failure (4) , the system is always in the state it was in before the last set of atomic actions (i.e. the atomic actions interrupted by

---

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

(2)
As will be discussed later, this information is necessary in order to control the cascading of backout of atomic actions.

(3)
See Verhofstad[30] for further details.

(4)
Failure such as action failure and system failure excluding media failure.

the failure) were invoked. A careful replacement, multiple copies and differential files are examples of object-oriented recovery scheme used to implement crash resistance that were proposed so far.

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 data concerning the object are distributed among the users of the object. For example, suppose that several atomic actions are permitted to be executed concurrently. Then in order for a given atomic action to undo the action on a non-recoverable object, it has to consult all other atomic actions that may have accessed this object. (1) Saving recovery data as global data (e.g. an audit trail) is only a partial solution for this problem since handling of recovery data is still inefficient (in particular, this inefficiency will be intolerable in a distributed environment). Therefore a non-recoverable object is usefull only in a limited environment where the cascading of backout is not necessary, namely where an atomic action is not permitted to access objects until all previous atomic actions that

---

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

accessed them are completed. Unifying both of a process-oriented scheme and an object-oriented scheme into a single scheme is considered to be usefull. An audit trail and a recovery cache are such examples.

(b) New recovery schemes

This paper proposes two new recovery schemes for crash resistance: a backout/commit cache and multiple uncommitted versions of mutable objects.

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 limited classes of objects. In particular, none of them can be applied to a non-recoverable object whose manager does not provide an inverse or a compensating action for each action.

A backout/commit cache is associated with an atomic action. 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 poit is passed. (1) An atomic action performs each action in one of the following ways depending on the kind of the underlying object on which

---

(1)
A commit cache can be considered to be an extension of an intention list proposed by Lampson et al[17].

the action is performed. (Also, examples that were derived from Appendix B are shown in Table.1.)

Case 1: write the action into the commit cache, but do not perform the original action (when the object is non-recoverable and neither inverse nor compensating action is provided)

Case 2: write the inverse action or compensating action into the backout cache and perform the original action (when the object is non-recoverable, but either inverse or compensating action is provided)

Case 3: write the undo action (1) into the backout cache, write the commit action into the commit cache and perform the original action (when the object is recoverable)

The backout/commit cache associated with a given atomic action is deleted when this atomic action is committed. Under this scheme, backing out and committing an atomic action are merely 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 atomic actions explicitly. And this scheme is a general one in the sense that it can be applied to any kind of object.

---

(1) An action that requests the manager of a recoverable object to undo the last action performed by the requester.

No object-oriented recovery scheme proposed so far, except for Montgomery's scheme[22], permits several mutually dependent atomic actions to be executed concurrently. Therefore this paper proposes a new object-oriented recovery scheme for recoverable objects --- multiple uncommitted versions of mutable objects --- that is an extension of careful replacement. In this scheme, whenever an atomic action 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. An atomic action can perform actions on the underlying objects before the previous actions performed on these objects are committed. Each version contains the additional information such as the identifier of the atomic action that performed the action (i.e. created this version) and the time at which the action was performed. Each version continues to exist until the immediately succeeding version is committed. Therefore this new scheme records not only a complete history of the state change of the object caused by uncommitted actions but also the identifiers of the atomic actions that depend on each version. (1) This makes it possible to control the cascading of backout

---

(1)
Maintaining multiple committed versions and distributing them among different media and different nodes will be usefull as it was discussed in the previous section. But this is another subject and beyond the scope of this paper. See Reed[25].

caused by the backout of an uncompleted atomic action, and therefore atomic actions can be executed highly concurrently.

(c) Cascading of backout

The main problem of cascading of backout is to keep track of the information flow which originates from a given atomic action. There are two approaches to the problem.

1) preventing interactions

This approach prevents each atomic action from interacting with other atomic actions until the end of the atomic action. All known database systems follow this approach, partly because most of them use locking schemes to avoid conflicts among atomic actions (transactions) and therefore backing out of transactions may cause deadlock[10], and partly because they do not have elegant schemes to keep track of the information flow and to back out affected transactions. Obviously the drawback of this approach is that it seriously restricts concurrent execution of atomic actions. For example, prevention of interactions is achieved by holding all locks to the end of transaction in database systems which uses locking schemes.

2) Control the cascading

This approach is to devise a (new) scheme which keeps track of the information flow and backs out the affected atomic actions when a failure occurs. As was discussed above, the multiple uncommitted versions also serve as such a scheme. In the later sections, we show how simply the backout can be done using this scheme. Since each atomic action can perform actions on the underlying objects before the previous actions are committed, a highly concurrent schedule can be achieved.

(2) Forcing completion

As was stated before, an atomic action must be completed (i.e. commitment must be completed) no matter what happens after the commit point. In order to complete commitment by all means, actions performed after the commit point

1) must not be lost even if a failure occurs at any point of execution

2) must be repeatable (idempotent[10][17]). (1)

A commit cache proposed in this paper satisfies the first requirement because the cache is implemented in a stable storage[17]. There are few methods of satisfying the second

---

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

requirement. One method is to reduce the actions performed after the commit point to a sequence of write actions which are well known to be idempotent[17]. Another method is to prevent the actions from being performed more than once by using a mechanism which provides an unique identifier (such as a timestamp) for each invocation of atomic action.

3. BASIC STRATEGIES

Database systems are considered to be composed of multiple levels, and therfore there exist multiple levels of atomic actions. The highest atomic action is a transaction, and the lowest is a machine instruction or a micro instruction. One of my ideas is that it is most appropriate to apply different schemes for achieving atomicity to different levels although more than one level may share the same scheme. This is because different levels generally have different requirements.

Higher level atomic actions such as 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 atomic actions must be deadlock free since deadlock detection tends to be very expensive[10]. In addition, it is necessary to support recoverable objects at such a high level where a non-recoverable object is not useful, partly because the users (i.e. the atomic actions) 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 sceme 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 atomic actions since these atomic actions 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 users (atomic actions) are almost serially executed. A non-recoverable object might be even desirable at a lower level from the viewpoint of performance[29]. 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 as a process-oriented recovery scheme, this could be applied to most levels.

This paper assumes a simplified multi-level distributed system which consists of three levels. The top level provides transactions, 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 disk storage is a non-recoverable object and physical actions on disk storage are atomic. These are implemented by the underlying operating system.
2) In order to make logical actions on database entities atomic, use
   * a serial schedule as a consistent scheduling scheme
   * a backout/commit cache not only as a process-oriented recovery scheme but also as an object-oriented recovery scheme for 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 process-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.

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 processes transactions , one at a time, by communicating with data management subsystems. A data management subsystem maintains portions of the database(i.e. a set of entities) and controls accesses to them. It consists of monitors [13][12] (1) that define entities and data management processes which access entities by invoking monitor procedures 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 idle data management processes of the subsystem. Then this process accesses the entity by invoking a monitor procedure. 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). Access to the database via a readr message is called read-only access, and access via a pair of a readu message and a write

---

(1)
The term " monitor " can be used instead of " manager of entity " since logical actions on an entity are serially executed.

message is called 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 (either read-only or update) access to each entity at most once.

Timestamps

Each node has a clock used for generating globally unique timestamps. As was suggested by Thomas [27] , 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 [16] 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 timestamp as was assigned to the transaction which performs the action. In addition, the timestamp assigned to each transaction is maintained in the versions of the entities it accessed as version numbers.

Non-deterministic Schedule of Transactions

One of the key points of the concurrency control of transactions proposed here is that each monitor schedules accesses in the order of timestamps assigned to them. Therefore the problem is how to ensure this ordering. There are three alternatives: a deterministic approach, a semi-deterministic approach and a non-deterministic approach.

(1) Deterministic approach

In a deterministic approach, a monitor defers scheduling of an access to the entity it defines until it confirms that there exist no outstanding accesses to this entity that are assigned smaller timestamps. One of the drawbacks of a deterministic 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 monitor in the timestamp order. SDD-1, adopting a deterministic approach, tries to remedy such a drawback by inventing null writes and limiting the number of transaction management subsystems that have to be polled [3]. This limitation of polling range is based on the information acquired from the pre-analysis of transactions.

---

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

(2) Semi-deterministic approach

In a semi-deterministic approach, a monitor schedules each access immediately as far as it is assigned a greater timestamp than all accesses to the same entity that were already performed. On the other hand, if there exist some accesses that are assigned greater timestamps and already committed , this outdated access is rejected. This approach was proposed by Reed[25]. However, it is also inappropriate 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.

(3) Non-deterministic approach

In a non-deterministic approach, a monitor 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 assigned greater timestamps and already committed, then the outdated access is rejected.

This paper proposed a non-deterministic approach because of its simplicity and its low overhead in normal situations. (1) In addition, the introduction of the concept of multiple uncommitted versions makes a non-deterministic approach more attractive because:

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 a later section.

Management of Uncommitted Versions

Whenever each transaction tries to access a given entity, a new version of that entity is created and the access is performed to this version. (2) Each version is preserved until its immediately successor is committed.

It is assumed that each entity is represented in the storage in the way shown in Figure 1. An entry of a map 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

---

(1)
"normal" means that sequence anomaly does not occur frequently and access trafic to each entity moderately low so that each entity has at most a few version.

(2)
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.

number, which is equall 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 takes one of the following values:

1) dirty; already read , but not yet written (meaningless in the 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 which contains this version. (1) Entries of a descriptor are sorted in the timestamp order.

A transaction can access each 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 serialized in the timestamp order.

When a readu action is invoked, the monitor 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 version

---

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

of the entity. If it is not the latest one, the monitor discards the versions whose version numbers are greater than ts. (1) After that, it creates a new current version whose version number(v#), state(s), and access mode(acc) are "ts", "dirty", and "update" . Then it returns the content of the immediate predecessor. If it is the latest one , the monitor examines the state of the current version first. If the state is "dirty" , creation of a new version is deferred until it becomes "dependent". (2) Otherwise, the monitor 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 which 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.

When a write action is invoked, the monitor acquires a free storage cell for a new version and writes the content of the buffer into it, and change the state to "dependent" if the

---

(1)
Of course, if it is older than already committed actions, then it is rejected.

(2)
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 but one are eventually undone because of sequence anomalies.

current state is not "discarded". Otherwise, it deletes the current invalid version and returns as such. If other data management processes are waiting for its completion, then the monitor wakes up the oldest one. Each version is deleted when it and a newer version are committed.

Commitment of Transactions

The central principle of the commit protocol proposed in this paper is that no transaction can commit its accesses until the states of all previous versions of these entities become " committed " . The commit protocol proposed here is basically a two-phase commit protocol, but it is considerably different from others[10][17][25] because it must co-operate with the concept of multiple uncommitted versions.

In the first phase , a transaction management process sends prepare messages to data management subsystems to confirm that versions it accessed are eligible to be committed. When the monitor receives the request, it :

1) changes the state of the designated version to " prepared " and returns as such if the immediately previous version is already committed
2) defers the data management process until the state of the immediately previous version is changed to " committed " or "discarded" if it is not yet committed
3) returns as "discarded" and delete the version if it is "discarded".

If all return messages are "prepared", then the second phase begines. Otherwise it aborts the transaction.

In the second phase , commitment must be completed no matter what happens. The transaction management process sends commit messages to data management subsystems to complete commitment. This is done by executing the set of actions saved in the commit cache. When the monitor 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 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.

Cascading of Backout

Backout of a transaction occurs either when it is aborted because of failures or when it is involved in a sequence anomaly. (1) It is also backed out when transactions on which it depends are backed out. Note that deleting a read-only version from the chain of the versions does not affect the other transactions. The outline of the cascading algorithm is as follows.
Suppose a transaction that is backed out to be T.

---

(1)
See Non-deterministic Schedule of Transactions in this section.

1) The transaction management process executing T sends undo messages to all data management subsystems that maintain valid (i.e. not discarded) versions it accessed.This is done by executing the set of actions saved in the backout cache.

2) Each monitor deletes the version accessed by T . If it is an update version, it changes the states of all newer versions (if any) to "discarded".

3) When a write or a prepare action is later invoked on one of these discarded versions, the monitor deletes it and returns as "discarded".

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

5. CONCLUSION

The main goal of this paper is to develop a set of schemes that realize highly concurrent as well as reliable (especially in terms of crash resistance) execution of transactions in a distributed database system. This goal was achieved by a combination of several new schemes, i.e. a consistent schedule of transactions based on timestamps, multiple uncommitted versions and a backout/commit cache. The consistent schedule based on timestamps relaxes the two-phase lock constraint[7] imposed on most database systems. And the multiple uncommitted versions coupled with the backout/commit cache relax the more serious constraint that " no transaction can access any entity until all

previous transactions that accessed the entity are completed ", which is also imposed on all known database systems. A 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. A processing overhead may also be acceptable since, in a normal situation where sequence anomaly does not occur frequently, the overhead is almost comparable to one induced by careful replacement.

The secondary goal is to explore a method of building a distributed database system that achieve the above goal as a multi-level system. This paper indicated 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 algorithm.

References

[1] Anderson, T.; Lee, P.A.; and Shrivastava, S.K. " A model of recoverability in multi-level systems, " Tech. Rep. 115, Computing Laboratory, Univ. Newcastle upon Tyne, UK, Nov. 1977.

[2] Banatre, J.P.; and Shrivastava, S.K. " Reliable resource allocation between unreliable processes," Tech. Rep. 99, Computing Laboratory, Univ. Newcastle upon Tyne, UK, April 1977.

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

[4] Bjork, L.A. "Recovery scenario for a DB/DC system," Proc. ACM'73 28(1973), 142-147.

[5] Bjork, L.A. "Generalised audit trail requirements and concepts for database applications," IBM Syst. J. 14, 3(1975), 229-245.

[6] Davies, C.T. "Recovery semantics for a DB/DC system," Proc. ACM'73 28(1973), 136-141.

[7] Eswaran, K.P.; Gray, J.N.; Lorie, R.A.; and Traiger, I.L. "The notions of consistency and predicate locks in a database system," Commun. ACM 19, 11(Nov. 1976), 624-633.

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

[9] Gray, J.N.; Lorie, R.A.; Putzolu, G.R.; and Traiger, L.L. "Granularity of locks and degree of consistency in a shared database," IBM Research Rep. RJ1654, Sept. 1975.

[10] Gray, J.N. "Notes on data base operating systems," Advanced Course on Operating Systems, Technical Univ. Munich, 1977.

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

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

[13] Hoare, C.A.R. "Monitors:an operating system structuring concept," Commun. ACM 17, 10(Oct. 1974), 549-557.

[14] Horning, J.; Lauer, H.C.; Melliar-Smith, P.M.; and Randell, B. "A program structure for error detection and recovery," Proc. Conf. Operating Systems; Theoretical and Practical Aspects, IRIA, 1974, 177-193.

[15] Lamport, L. "A new solution of Dijkstra's concurrent programming problem," Commun. ACM 17, 8(Aug. 1974), 453-455.

[16] Lamport, L. "Time, clocks and ordering of events in a distributed system," Rep. CA-7603-2911, Mass. Comput. Assoc., Mar. 1976.

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

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

[19] Liskov, B.H.; Svobodova, L.; Greif, I.; and Clark, D.D. "Semantics of distributed computing," DSG Progress Rep., Laboratory for Computer Science, M.I.T., Cambridge, MA, July 1978.

[20] Lomet, D.B. "Process structuring, synchronisation and recovery using atomic actions," SIGPLAN Notices 12, 3(Mar. 1977), 128-137.

[21] Lorie, R.A. "Physical integrity in a large segmented database," ACM Trans. Database Syst. 2, 1(Mar. 1977), 91-104.

[22] Montgomery, W. PhD Thesis in preparation, Laboratory for Computer Science, M.I.T., Cambridge, MA, 1978.

[23] Randell, B. "System structure for software fault tolerance," IEEE Trans. Softw. Eng. SE-1, 2(June 1975), 220-232.

[24] Randell, B.; Lee, P.A.; and Treleaven, P.C. "Reliability issues in computing system design," Comput. Surv. 10, 2(June 1978), 123-165.

[25] Reed, D.P. PhD Thesis in preparation, Laboratory for Computer Science, M.I.T., Cambridge, MA, 1978.

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

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

1975.

[28] Verhofstad, J.S.M. "Recovery and crash resistence in a filing system," Proc. 1977 ACM SIGMOD Int. Conf. on Management of Data, ACM, NY, 158-167.

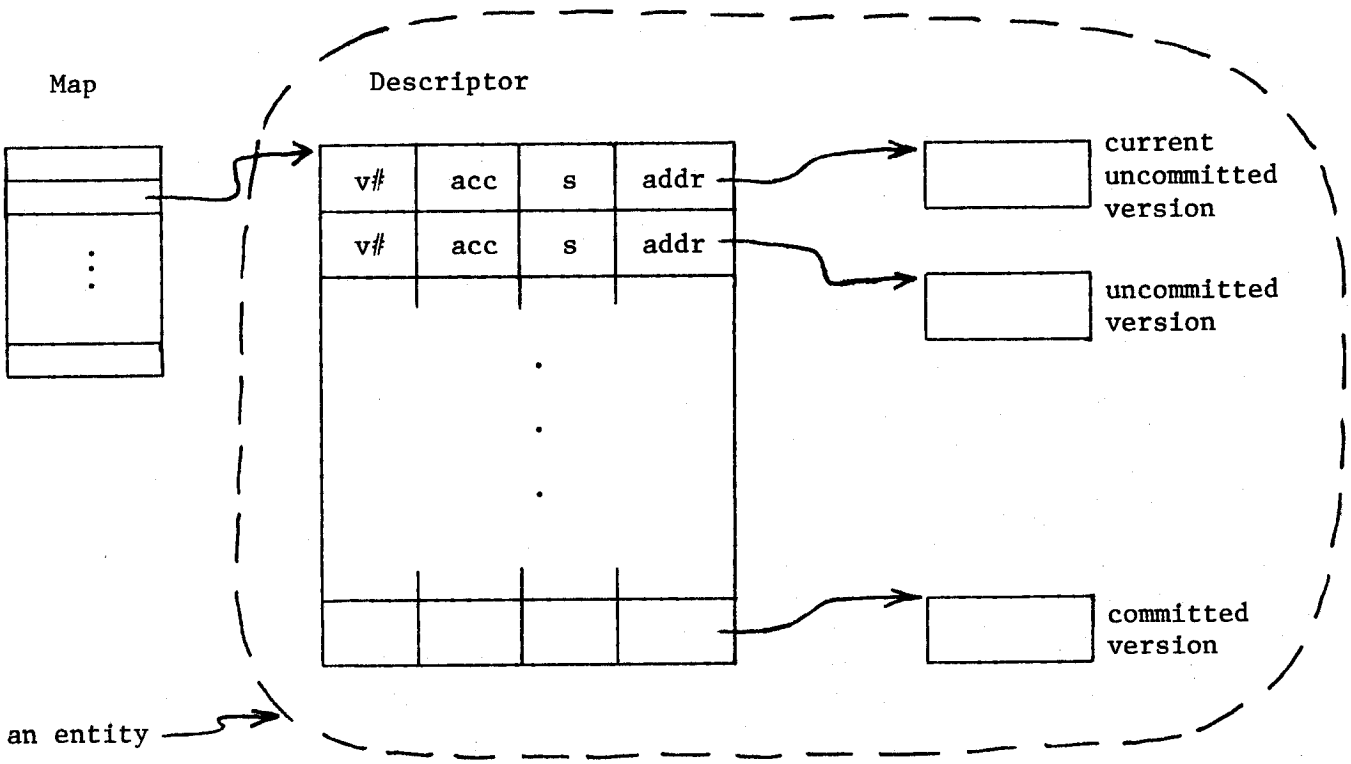
[29] Verhofstad, J.S.M. "On multi-level recovery: an approach using partially recoverable interfaces," Tech. Rep. 100, Computing Laboratory, Univ. Newcastle upon Tyne, UK, May 1977.

[30] Verhofstad, J.S.M. "Recovery techniques for database systems," Comput. Surv. 10, 2(June 1978), 167-195.

Table 1: Caching Into A Backout/Commit Cache

Type	Intended Action	Action Actually Performed	Action Cached In a Backout Cache	Action Cached In a Commit Cache
1.	continue(arg)	--	--	continue(arg)
2.	unit.create&set(arg)	unit.create&set(arg)	unit.delete(arg)	--
3.	send readu(arg)...	send readu(arg)...	send undo(arg)...	send commit(arg)...

Figure 1: Storage representation of entities



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. A Concurrent Pascal[20] was adopted as a base language, with three kinds of new constructs added.

(1) Exception handling

Constructs similar to those that were proposed by Liskov et al[18] were added for exception handling. Procedures, processes, monitors and classes have headings that contain the information about the ways in which they may terminate. For example, procedure p has the following heading.

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

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

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

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

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

```
end;
```

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

(2) Recovery

Several language constructs are added to support recovery. The recovery scheme using a backout/commit cache was built 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(args): do statement_1;
    action_2(args): do statement_2;
        .
        .
end;
```

(3) Message passing

The new language constructs that handle message passing between processes (over a network) are similar to those of Liskov[30] or Feldman[31]. The syntax of the send construct is:

```
send action(args) to recipient timeout limit
     response_1: do statement_1;
     response_2: 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(args): do statement_1;
   action_2(args): do statement_2;
      .
      .
end;
```

and

```
reply response(args) to recipient;
```

The recipient designated in send or reply 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. Several assumptions are made in order to make our algorithms simple. First, the disk storage management issues were ommitted. Second,it is assumed that all descriptors are of the same fixed size, although this results in low space efficiency. Third, it is assumed that each node provides virtual memory facilities.

Fig.B-1 and Fig.B-2 show the detailed algorithms of the data management subsystem and the transaction management subsystem, respectively.

Fig. B-1 Data Management Subsystem

```
type directory = array[1..file_length] of descriptor_address;

type descriptor_address = integer;

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 ordered_queue = class(max_length : integer)
   signals(failure);

 % queue elements are sorted in the order of version numbers %

   function entry arrival(version_number : integer)
      returns(integer) signals(failure);

    % returns the index of the next queue element in which arrival can take
      place %

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

    % returns the index of the next queue element from which departure can
      take place %

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

    % defines whether the queue is empty %

begin
 % initialization %
end;
```

```
type independent_queue = class(max_length : integer)
   signals(failure);

 % defines a set of independent single queues %

   function entry arrival(version_number : integer)
      returns(integer) signals(failure);

    % returns the index of a free queue element, in which arrival takes place
      %

   function entry departure(version_number : integer)
      returns(integer) signals(failure);

    % returns the index of the queue element which has the version_number ,
      and from which departure takes place %

   function entry empty(version_number : integer)
      returns(boolean) signals(failure);

    % defines whether the queue element which has the version_number exists %

begin
 % initialization %
end;


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

```
type r_disk = monitor(unit : disk) signals(failure);
   % defines a recoverable disk from a non-recoverable disk type
   object (that is provided by the underlying operating system)
   by use of backout/commit cache %

procedure entry create&set(var addr : integer; block : univ page
   signals(failure);

          .
          .
     begin
          .
          .
      cache unit.delete(addr) into backout-cache;
      unit.create&set(addr, block);
          .
          .
end
except when others : signal failure;

procedure entry write(addr : integer; block : univ page)
   signals(failure);
     .
     .
  begin
     .
   unit.read(addr, oldblock);
   cache unit.write(addr, oldblock) into backout-cache;
   unit.write(addr,block);
     .
     .
end
except when others : signal failure end;

procedure entry read(addr : integer; var clock : univ page)
   signals(failure);
     .
     .

procedure entry delete(addr : integer)
signals(failure);

begin
 % initialization %
end;
```

```
type entity = monitor(index : integer) signals(failure);

var map : directory; des : descriptor; new : version_descriptor;
  ordered : ordered_queue; independent : independent_queue;
  access_queue, commit_queue : process_queue; r_unit : r_disk;

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

 % returns the descriptor index whose version number is closest to and older
   than v_no, or returns "0" if not found%

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

 % returns the descriptor index whose version number is equal to v_no, or
   returns "0" is not found%

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

 % returns the descriptor index whose version number is closest to and newer
   than v_no, or returns "0" if not found%

function current returns(integer) signals(failure);

 % returns the current descriptor index %

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 %

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

 % delete the (i)th entry from the descriptor %

procedure commit_action;

% commits the logical action on the database entity,
  i.e. executes the actions cached in the commit cache%

var end_of_cache : boolean; ...
            .
            .
begin
            .
            .
       while not end_of_cache do
       begin
            .
            .
        case action in commit-cache of
         continue(q : queue : do continue(q);
            .
```

```
            .
          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%

            .
            .
        begin
            .
            .
          while not end_of_cache do
          begin
            .
            .
           case action in backout-cache of
             x.create&set(var y : integer; z : univ page): do
               x.create&set(y, z);
             x.write(y : integer; z : univ page): do
                x.write(y, z);
                  .
                  .
             end;
                  .
                  .
        end;
                .
                .
  end
  except when others : backout_action end;

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

  % discards all versions whose version numbers are greater than
    or equal to v_no %

  var i : integer;

  begin
    i := 1;
    repeat
      if des.list[i].s<>discarded then
        begin
          with des.list[i] do
```

```
          begin
            if acc=update then r_unit.delete(addr);
            if (s=dirty)and(not ordered.empty) then
            cache
            continue(access_queue[ordered.departure])
            into commit-cache
            else if not independent.empty(v#) then
              cache
              continue(commit_queue[independent.departure(v#)])
              into commit-cache;
            s := discarded
          end
        end;
      i := i + 1;
    until (des.list[i].v#<v_no)and(des.list[i].s<>discarded);
    r_unit.write(map[index],des)
  end
  except when others : signal failure end;

procedure entry readu(timestamp : integer; var block : univ page)
   signals(obsolete, congestion, failure);
  % reads the value of the immediately preceding version of the designated
    entity and creates a new version used for an update access %

  var l,m : integer;
  begin if des.length=max then signal congestion;
    l := older(timestamp);
    m := newer(timestamp);
    if (l=0) or ((m<>0)and(des.list[m].s = prepared))
    then signal obsolete;
    if des.list[l].s=dirty then
      begin
        delay(access_queue[ordered.arrival(timestamp)]);
        l := older(timestamp)
      end;
    if l<>current then discard(l);      *1
    with new do
    begin
      v# := timestamp;
      acc := update;      *2
      s := dirty;      *3
      addr := des.list[l].addr
      end;
    insert(l, new);
    r_unit.read(des.list[l].addr, block);
    r_unit.write(map[index),des);
                        *4
  commit_action
  end
  except when others : begin backout-action; signal failure end
  end;
```

```
procedure entry readr(timestamp : integer; var block : univ page)
   signals(obsolete, congestion, failure);
   % used for a read_only access.

 Same as readu except that;

   *1 ---> null
   *2 ---> acc := read_only;
   *3 ---> s := dependent;
   *4 ---> if (1=current)and(not ordered.empty) then
           cache
           continue(access_queue[ordered.departure])
         into commit-cache; %

procedure entry write(timestamp : integer; block : univ page)
   signals(discarded, failure);

   % updates the version created by the associated readu action %

  var k : integer;
  begin
    k := equal(timestamp);
    if des.list[k].s=discarded then
      begin
        delete(k);
        r_unit.write(map[index], des);
        signal discarded;
        commit_action
      end;
    r_unit.creat&set(des.list[k].addr, block);
    des.list[k].s := dependent;
    r_unit.write(map[index],des);
    if not ordered.empty then
    cache
      continue(access_queue[ordered.departure])
      into commit-cache;
  commit_action
end
except when others : begin backout-action; signal failure end
end;

procedure entry prepare(timestamp : integer)
   signals(discarded, failure);

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

  var k, l := integer;
  begin
    k := equal(timestamp);
    l := older(timestamp);
    while (des.list[l].s<>committed)and(des.list[k].s<>discarded)
      do
        begin
```

```
            delay(commit_queue[independent.arrival(timestamp)]);
            k := equal(timestamp);
            l := older(timestamp)
          end;
        if des.list[k].s=discarded then
          begin
            delete(k);
            r_unit.write(map[index], des);
            signal discarded;
            commit_action
          end
        else
          begin
            des.list[k].s := prepared;
            r_unit.write(map[index], des)
          end;
      end
      except when others : begin backout_action; signal failure end
      end;

  procedure entry commit(timestamp : integer) signals(failure);

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

    var k, l, m, nts : integer;
    begin
      k := equal(timestamp);
      l := older(timestamp);
      if l = 0 then commit_action;
      des.list[k].s := committed;
      if des.list[k].acc=update then r_unit.delete(des.list[l].addr);
      delete(l);
      r_unit.write(map[index], des);
      if k<>current then
        begin
          m := newer(timestamp);
          nts := des.list[m].v#;
          if not independent.empty(nts) then
          cache
          continue(commit_queue[independent.departure(nts)])
          into commit-cache
        end;
        commit_action
    end
    except when others : begin backout_action; signal failure end
    end;

  procedure entry undo(timestamp : integer) signals(discarded, failure);
  % discards the designated version and all subsequent versions that depend on
  this version %
```

```
var k, l, m, nts : integer;

begin
  l := older(timestamp);
  k := equal(timestamp);
  if k = 0 then commit_action;
  if des.list[k].s=discarded then
    begin
      delete(k);
      unit.write(map[index],des);
      signal discarded;
      commit_action
  end;
  if des.list[k].acc=update then
  begin
    if k<>current then
      begin
        discard(k);
        r_unit.delete(des.list[k].addr)
      end
    else
      begin
      if des.list[k].s=dirty then
        begin
          if not ordered.empty then
          cache
          continue(access_queue[ordered.departure])
          into commit-cache
        end
      else r_unit.delete(des.list[k].addr)
    end
  end
else
  begin
    if k<>current then
      begin
        m := newer(timestamp);
        nts := des.list[m].v#;
        if (des.list[l].s=committed)and(not independent.empty(nts)) then
        cache
        continue(commit_queue[independent.departure(nts)])
        into commit-cache
      end
  end;
delete(k);
r_unit.write(map[index], des);
commit_action
end
except when others : begin backout_action; signal failure end
end;
```

```
% initialization of monitor %

begin
  des.length := 1;
  with des.list[1] do
    begin
      v# := 0;
      acc := update;
      s := committed;
      r_unit.create&set(addr,nill)
    end;
  r_unit.create&set(map[index], des);
  commit_action
end
except when others ; begin backout_action; signal failure end
end;
```

```
type data_management_process = process;

  var block : univ page; . . . ;
  begin
    cycle
      case action in message of
        readu(x : entity; timestamp : integer) : do
          begin
            x.readu(timestamp, block)
            except
              when obsolete : reply obsolete;
              when congestion : reply congestion;
              when failure : reply failure
            end;
            reply normal(block)
          end
        readr(x : entity; timestamp : integer) : do
          begin
                .
                .
                .
        end
            .
            .
            .
    end;


  end;
```

Fig.B-2 Transaction Management Subsystem

```
type transaction_management_process = process;

var item : entity; inbuffer, outbuffer : univ page;
data_manager : set of data_management_process;
ts, lim1, lim2, lim3, lim4 :integer; . . . ;

procedure commit_transaction;

 % commits the transaction, i.e. executes the actions cached in
   the commit cache %

  var end_of_cache : boolean; ...

  begin
       .
       .
    while not end_of_cache do
    begin
         .
         .
      case action in commit-cache of
         send commit(x : integer) to y : do
            send commit(x) to y;
            .
            .
      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 %

  var end_of_cache : boolean; ...

  begin
       .
       .
    while not end_of_cache do
    begin
         .
         .
      case action in backout-cache of
         send undo(x : integer) to y : do
```

```
                send undo(x) to y;
                .
                .
          end;
             .
             .
       end;
          .
          .
    end
    except when others: backout_transaction end;

     .
     .
 begin
   cycle
        .
     ts := new_timestamp;
        .
        .
     cache
       send undo(ts) to data_manager
     into backout-cache;
     cache
       send commit(ts) to data_manager
     into commit-cache;
     send readu(item, ts) to data_manager timeout lim1
       normal(imput: univ page): do null;
       obsolete: do begin
                      backout_transaction;
                      restart_transaction
                    end;
       congestion: do begin
                       wait_for_a_while;
                       retry_sending
                     end;
       failure, timeout: do backout_transaction;
        .
        .
     send write(item, ts, output) to data_manager timeout lim2
        .
        .
     send prepare(item, ts) to data_manager timeout lim3
        .
        .
     commit_transaction
   end
 end;
```