A Nonblocking Atomic Transaction Processing Algorithm with Real-Time Property

I-Ling Yen
Lirong Dai
Computer Science Department
University of Texas at Dallas
MS EC-31, Box 830688
Richardson, TX 75083-0688

Ing-Ray Chen
Department of Computer Science
Virginia Polytechnic Institute and State University
7054 Haycock Road
Falls Church, VA 22043

Biao Chen
Computer Science Department
University of Texas at Dallas
MS EC-31, Box 830688
Richardson, TX 75083-0688

Abstract

Many multiple server systems are now being used for heavily accessed web services. Performance, availability, and real-time transaction processing are important requirements for many of these applications. In this paper, we apply the multi-invariant data structure (MIDS) concept to develop an atomic transaction processing algorithm. We show that our algorithm achieves nonblocking property. It also allows preemption at any point of transaction processing without suffering from inconsistency problems or high recovery costs. Thus, the algorithm is suitable for real-time transaction processing systems. Preliminary performance study results show that our algorithm incurs almost no overhead compared with two phase commit protocol.

1 Introduction

Advances in the worldwide network infrastructure have changed the usage patterns of many application systems. Many commercial systems, such as commercial web servers and transaction processing systems, are now accessed worldwide with a large number of concurrent accesses. Single processor platforms lack adequate power and reliability for implementing many such systems. Some commercial web service applications now use multiple server systems to offer high availability and load sharing. However, most of these systems are designed to handle file access requests or simple transactions rather than for complex distributed atomic transaction processing.

Atomic transaction processing algorithms [1, 3], such as the two-phase commit protocol and its variants [1, 3, 7, 6], have been studied extensively. It is a viable scheme for distributed transaction processing systems. However, the two-phase commit protocol incurs the problem of extended blocking in the event of a server failure while it is processing a transaction. Though the three-phase protocol [5, 6] can be used to eliminate the blocking problem, it incurs a much higher overhead. Replication has also been considered to eliminate the blocking problem in the two-phase commit protocol, but the full replication of the system is too expensive for many applications.

Besides the performance and availability concerns, real-time transaction processing is also an important requirement for many emerging E-commerce applications. More specifically, in some systems transactions with different levels of priorities are considered. When a high priority transaction arrives at the system, the system should be able to quickly preempt lower priority transactions and switch to the high priority transaction within a bounded delay. However, a low priority transaction can block a high priority transaction since it needs to complete its atomic actions to avoid causing inconsistencies in the database. To abort the low priority transaction can result in expensive recovery operations. Methods such as the three-phase protocol can eliminate the blocking problem [5, 6], but the priority inversion problem still remains unsolved.

As discussed in [8], the multi-invariant system approach can be used to achieve high reliability and responsiveness without incurring excessive overhead in multi server systems. The shared system state (database) is specified by two (or more) invariants, a strong invariant and a weak invariant. The system is designed to operate correctly even when only the weak invariant is true, though perhaps at a reduced level of performance. Normally, the system operates at peak performance with the strong invariant being true. Should a processor fail or should a task be interrupted by a higher priority task while updating the shared data structure, it is guaranteed that the weak invariant will continue to be true though the strong invariant may be violated. The strong invariant is restored in a finite number of steps to recoup the peak performance of the system. We have shown that the overhead of multi-invariant systems is low (8-15%) compared with naive systems where failure can cause violation of data integrity [9].

In this paper, we apply the multi-invariant system concept for atomic transaction processing. We use the MIDS scheme to obtain a highly available, reliable, and responsive transaction processing algorithm. In the next section, we discuss the background information, including the system model, assumptions, and the common protocol for atomic transaction processing, namely, the two-phase commit protocol. In Section 3, we present the MIDS approach for transaction processing. We will also address detailed implementation issues such as achieving reliable storage. The correctness and performance of the algorithm is analyzed in Section 4. We also discuss the major advantages and real-time processing capabilities of the MIDS transaction processing scheme. In Section 5, we present the details for implementing the MIDS-AT algorithm and some preliminary performance evaluation results. Finally, Section 6 states the conclusion of this paper.

2 Background

2.1 System Model

We consider a system that consists of N servers, S_1, S_2, ..., S_N, connected through a high speed network. The state of the system is represented by a shared database which is stored on a logically shared storage system consisting of M disks, D_1, D_2, ..., D_M. A client can issue a transaction to any of the N servers. Multiple transactions can be processed by the servers in parallel. While processing transactions, the servers access the shared system state space (the database) concurrently. In this paper, we consider implementing the logically shared storage on physically distributed disks where each server has its own disk and the disks are accessed through the corresponding servers. This facilitates a fair comparison between our algorithm and the implementation of conventional distributed database systems (two-phase commit protocol). Also, the physical system configuration is widely available.

We consider clean failures for processors and storage systems. A process will simply not function after a failure. No malicious behavior is considered. Also, we assume that the processor failure can be detected within a fixed time period. When a storage unit fails, we assume that no access is possible. In other words, we do not consider possible accesses to incorrect information in the storage system. Schemes such as parity or Hamming code can be used to detect or correct such failures. Finally, we assume the use of an ethernet like local area network. Thus, there is no network partitioning problem due to processor failures.

2.2 Two Phase Commit Protocol for Atomic Transaction Processing

Consider a transaction T. For simplicity, we assume that all the records accessed by T are evenly distributed on K servers. Let R_1, R_2, ..., R_K denote the K sets of records to be accessed by T, where R_i contains the records $r_{i,1}$, $r_{i,2}$, ..., $r_{i,k}$, that are stored on S_i. Let T_i denote the ith

subtransaction to be processed by S_i accessing records in R_i. Also, let S_c denote the coordinator for processing transaction T and assume that S_c does not process a subtransaction. Upon receiving the transaction T, S_c creates a transaction control block (TCB), *tcb*, for T. Each TCB contains the field *commitFlag* to indicate whether the transaction is committed. In the following, we give the simplified pseudo-code for the original two-phase commit protocol without considering failure recovery.

```
S_c upon receiving T:
  create tcb for T;
  divide T into subtransactions T_i and send T_i to S_i;
  receive response E_i from S_i or raise timeout exception;
  if E_i = commit, for all i, then tcb.commitFlag := commit;
  else tcb.commitFlag := abort
  endif;
  send decision in tcb.commitFlag to S_i, for all i;
  S_i upon receiving T_i:
    for j from 1 to k do
      lock (r_{i,j}); read (r_{i,j});
    endloop;
    compute T_i and log the updates for all records in R_i;
    if the computation succeeds then
      send commit to S_c;
    else send abort to S_c;
    endif;
    receive decision C_T from S_c or raise timeout exception;
    if C_T = commit then complete the update; endif;
    for j from 1 to k do unlock (r_{i,j}); endloop;
```

2.3 Performance of the Two Phase Commit Protocol

In the two phase commit algorithm, the coordinator S_c needs to access the disk once upon creating TCB, *tcb* and once for updating *tcb.commitFlag*. Each server S_i needs to access each record $r_{i,j}$ on the disk three times, once for reading and temporarily locking the record, once for locking the record (actually updating the lock on the disk) and logging the update (assume log is next to the record), and once for updating and unlocking the record. Other accesses, such as access to index table, waiting to acquire a lock, etc., can be performed in the memory. For communication among servers, there will be two messages from S_c to each S_i, for all i (once for sending T_i and once for sending C_T), and one message from each S_i to S_c (sending responses). Thus, the total overhead of the algorithm (without considering failure recovery) can be approximated as $3Kk+2$ disk accesses and $3K$ messages (note that Kk is the total number of records to be accessed for transaction T). In actual implementations, the overhead will be higher. For example, the servers S_i, for all i, will need to respond to S_c to confirm that they have updated the record so that the TCB entry can be removed.

3 Atomic Transaction Processing using the MIDS Approach

3.1 Multi-Invariant Data Structures

A data structure can be characterized by a representation invariant which specifies formally all possible legitimate states of the data structure. When the data structure is accessible by several processes, before every access a process can assume that the invariant assertion is true; this is called its rely condition [4]. Similarly, the process must ensure that the assertion is true after it completes the operation; this is called its guarantee condition [4].

When a processor fails while accessing the shared data structure, it will not be able to reestablish the invariant if it has violated the invariant. This can cause incorrect operation of the entire program. It may be possible to design data structures that maintain the invariant even in the presence of server failures. However, the performance of such data structures can be very poor. Our approach is to replace the invariant assertion by multiple assertions, namely, strong and weak invariants. During failure-free situations, the code should ensure that the strong invariant is true. Whenever the strong invariant is true, the performance of the system is at its peak. When a processor fails, the strong invariant may be violated, but the code should guarantee that the weak invariant will be true. When the strong invariant is violated, the performance of the system may degrade; however, the system will operate correctly provided that the weak invariant is true. If there are no further processor failures, then incomplete accesses can result in at most a finite number of recovery actions by other processors before the strong invariant is reestablished, either via data-structure integrity checks and recovery or incrementally in a manner similar to self-stabilization.

A formal definition of a multi-invariant data structure (MIDS) is given in the following.

Definition: Let I_1, I_2, ..., I_q denote a sequence of invariants where $I_1 \Rightarrow I_2 \Rightarrow \cdots \Rightarrow I_q$. A data structure D is a multi-invariant data structure if

1. D always satisfies I_q,
2. if D satisfies I_j but not I_{j-1} for $j > 1$, then it can be modified to obtain D' such that (a) D' satisfies I_{j-1}, (b) the output of any operation on D' is identical to the output of that operation on D, and (c) the performance of the program using D' is better than that using D.

In this paper, we consider $q = 2$ and refer to I_1 as the strong invariant of D and I_2 as the weak invariant of D.

The major advantage of using the MIDS design is the potential of low overhead for tolerating failures. Conventional fault tolerance approaches do not have the notion of multiple invariants . Thus, a fault-tolerant program always tries to maintain a strong invariant. This can imply extra computations for each action operating on the shared system state. In the MIDS, the system is

allowed to fall into a state that satisfies only a weaker invariant. Thus, the overhead of the system under failure-free situations is low, consisting only of simple integrity checks.

3.2 Data Structures and Invariants

To achive the real-time property, our goal is to develop a multi-invariant data structure that allows the transaction to be aborted at any time while recovery of each record can be done independently. Consider a TCB, *tcb*, for a transaction T. The states of $tcb.commitFlag$ can be: $commitFlag = active$ when the transaction is still active, $commitFlag = commit$ or *abort* when the transaction is committed or aborted, respectively. The $tcb.commitFlag$ of a newly created TCB is initialized to $active$.

A server status block is maintained for all servers to keep track of their status. For server S_i, $S_i.status$ is $active$ if S_i is working and $S_i.status$ is set to $inactive$ when failure of S_i is detected. An extra field $serverPtr$ is maintained in TCB which points to the server status block of the coordinator of the transaction. Thus, from TCB, we can directly trace back and determine the status of the coordinator for the transaction. When a server comes back from failure (inactive) to working (active) state, a different server status entry should be used. New transactions procesed by the reactivated server will have their $serverPtr$ pointing to the new entry and transactions that are processed before the server failure should have their $serverPtr$ pointing to the original entry. By doing so, the past $inactive$ status of the server will be maintained till the information is no longer needed.

We modify the lock field of each record in the database. A $TCBptr$ is used to indicate whether a record is locked. To lock a record, we assign $TCBptr$ to point to the TCB of the transaction. Assigning $TCBptr$ to nil indicates that the record is unlocked. Besides keeping track of the lock status, $TCBptr$ can be used to trace back to the task control block and its $commitFlag$ and, hence, can recover a record independently of other records involved in the transaction. This facilitates real-time processing since recovery can be done for each record independently. A $logPtr$ is also added to each record in the database. To update a record, a log is first created to store the current record information and then the record is updated. After the current state is copied to the log record, the pointer $logPtr$ is assigned to point to the log information. Subsequently, the record itself can be updated. $logPtr$ is used to facilitate one step update required in the MIDS approach. $logPtr$ is only assigned after a log record is completely copied and will be set back to nil when a commit decision is received. The data structure design of the system is illustrated in detail in Figure 1.

The strong invariant of the system is that for all r, such that record r is in the system database, we have $r.TCBptr = nil$ **and** $r.logPtr = nil$. In a failure free situation, all records accessed by a transaction will be released after the completion of the transaction. Thus, the strong invariant is

satisfied. When failure occurs, the on-going transactions may not be completed properly and some of the records will remain locked. In the two-phase commit protocol, this situation will result in blocking. Based on the MIDS design, the $TCBptr$ of each record needs to point to a TCB such that the locked records can be released independently. Thus, the system can continue to function without a specific recovery phase even if some records are still locked. The weak invariant that guarantees the correct operation of the system is as follows: For all r, such that record r is in the system database, $r.TCBptr = nil$ or there exists a TCB, tcb, where $r.TCBptr = tcb$ **and** $valid(r.logPtr)$. To define $valid(r.logPtr)$, we need to define the content of r. Let v denote the content of r and u denote the update function for transaction T on r, i.e., $u(r)$ denotes the updated value of r after executing transaction T. The predicate $valid(r.logPtr)$ can be defined as

$(r.logPtr = nil$ **and** $content(r) = v)$ **or**
$(r.logPtr \neq nil$ **and** $content(r.logPtr) = v$ **and**
  $(r.TCBptr.commitFlag = active$ **or**
    $r.TCBptr.commitFlag = abort$ **or**
    $(r.TCBptr.commitFlag = commit$ **and**
      $content(r) = u(r))$ $)$ $)$

If a failure occurs when the log record update is not completed or if there is no need to update r, then $r.logPtr = nil$ and the content of r should not have been touched. In this situation, the failure has no impact on r, thus, the system state is consistent in terms of r. If a failure occurs after $r.logPtr$ is assigned, then the content in $r.logPtr$ should be the original content of r. If a failure occurs before a decision is made, then the transaction will be aborted and the content in r does not matter since the content in $r.logPtr$ can recover r to its original state. If a failure occurs after a decision is made and the decision is abort, then the content of r does not matter. If the final decision is made to commit the transaction, then r should be completely updated to the state $u(r)$.

3.3 MIDS-AT Algorithm

Two phases are needed in the MIDS atomic transaction processing (MIDS-AT) algorithm. In the first phase, all the records to be accessed are locked, i.e., *TCBptr* of each record is set to point to the TCB of the transaction. For a record to be updated, a log is created to store the current record information. The pointer *logPtr* is assigned to point to the log information. After the assignment of *logPtr*, the original record is updated. At the end of the first phase, the *commitFlag* in the TCB is set to *commit* or *abort* if the transaction is to be committed or aborted. In the second phase, *TCBptr* and *logPtr* of each record are reset to *nil* if the transaction is committed. In the case of an aborted transaction, the log information is restored to the original record to return to the state before the transaction is processed.

The pseudo-code for MIDS-AT algorithm is given as follows. We consider a transaction T accessing K records r_1, r_2, ..., r_K. The task control block for T is denoted by *tcb*.

```
S_c upon receiving T:
  create tcb, for T;
  for i from 1 to K do
    lock (r_i, tcb); read (r_i);
    if r_i is to be updated then
      copy r_i to a log, log;
      r_i.logPtr := log;
      update (r_i) according to T;
    endif;
  endloop;
  if the computation succeeds then
    tcb.CommitFlg := commit;
  else tcb.CommitFlg := abort;
  endif;
  for i from 1 to K do
    r_i.TCBptr := nil; r_i.logPtr := nil;
  endloop;
```

The code presented above gives the general MIDS-AT algorithm for transaction processing. Some implementation details will vary when the algorithm is used for various architectures. For example, if we consider implementing this algorithm on distributed servers, then all the actions performed within the two **for** loops, which access the individual records, will actually be performed by the servers that hold the records. In other words, S_c needs to send requests to the other servers for record accesses.

Function $lock(r_i, tcb)$ in the above code can have different implementations for different architectures. If we consider implementing the database on a shared memory system, then the special locking algorithm designed for multi-invariant data structures (as discussed in [9]) has to be used. When distributed storage is considered, the *lock* function is simpler. It first checks $r_i.TCBptr$ to decide whether the record is locked. $r_i.TCBptr = nil$ indicates that the record r_i is not locked and we simply set $r_i.TCBptr$ to point to *tcb*. If $r_i.TCBptr \neq nil$, then the record is locked and we need to check the transaction control block $r_i.TCBptr$ points to to determine the execution status of the transaction that holds the lock. If $r_i.TCBptr.commitFlag$ is *active*, then we have to trace back to the server status block to determine whether the server had failed. If the server is still active, then record r_i is indeed locked. If the server had failed, then the recovery action for record r_i should be taken independently. First, $r.TCBptr.commitFlag$ should be changed from *active* to *abort* so that subsequent accesses to the same TCB will not require tracing back to the server record to find out the status. Subsequently, actions for aborting the subtransaction should be performed and the r_i can be released. If $r_i.TCBptr.commitFlag = commit$ or *abort*, then the incomplete operations on record r_i should be taken to commit or abort the transaction and then r_i can be released.

The pseudo-code for function $lock(r, tcb)$ is given in the following. Note that the recovery code for the case when r is locked by a failed server is a part of the function and is also given in the following.

```
lock(r, tcb):
  if (r.TCBptr = nil) then r.TCBptr := tcb;
  else
    tcb' := r.TCBptr;
    if tcb'.serverPtr.status = active then wait for the lock;
    else recovery(r, tcb');
    endif;
  endif;

recovery(r, tcb):
  if tcb.commitFlag = abort then
    if r.logPtr ≠ nil then copy r.logPtr to r; endif;
    r.TCBptr := nil; r.logPtr := nil;
  else if tcb.commitFlag = commit then
    r.TCBptr := nil; r.logPtr := nil;
  else -- tcb.commitFlag = active and tcb.serverPtr.status = inactive
    tcb.commitFlag := abort;
    if r.logPtr ≠ nil then copy r.logPtr to r; endif;
    r.TCBptr := nil; r.logPtr := nil;
  endif;
```

Note that failure can occur at any point of the execution of the recovery code without violating data integrity. However, the overhead can be high if the above code has to be executed whenever a server tries to lock a record and finds that the record is locked. Essentially, two extra references will be needed to check whether the server that locks the record is still active. To avoid this overhead, we can use the code switching mechanism used in the MIDS approach. When there is no server failure, there is no need to further check the status of a locked record but wait for the lock being released. When a server failure is detected, we switch the execution to execute the above code whenever an attempt is made to lock a record that is already locked. This can be done by dynamic binding or simply by using an extra check in the code.

As we can see, when there is a server failure, the only action that needs to be taken in the MIDS-AT algorithm is to set $S_i.status$ to $inactive$. No extensive recovery action is needed after the failure. The locked records can be recovered independently when they are accessed.

3.4 Reliable Storage in MIDS-AT

The failure assumption for multi-invariant data structure (MIDS) approach is fail-stop processor failure. In other words, a reliable storage is required for the MIDS system to work properly. Reliable storage can be achieved by using parallel disks with a certain level of redundancy, such as RAID [2]. In this paper, we consider the implementation on physically distributed storage system. We

divided the logical database into two segments, one for the manipulation of transaction control blocks and server status blocks, namely, the TCB space, and the other for storing the collection of data records (database). Here, we will focus on implementing a reliable TCB space since the TCB information has a crucial impact on the continuous operation of the system. For the data records, the RAID approach can be used to improve its availability.

To achieve reliable storage, we fully replicate the TCB space. A server S_i, upon receiving a transaction from a client, creates a TCB, *tcb*, for that transaction on S_i's local disk. It also replicates the TCB, tcb', on S_j. TCB update takes place when the final decision for a transaction is made and $tcb.commitFlag$ is set to *commit* or *abort*. Due to the existence of two copies of the TCB, an inconsistency problem can occur. In the case of TCB creation, *tcb* and tcb' can be created in parallel by S_i and S_j. For updating the $commitFlag$, S_i has to initiate the write to the backup storage first. S_i updates $tcb.commitFlag$ only after the update to $tcb'.commitFlag$ is confirmed. When a TCB is referenced, an attempt to reference the primary copy *tcb* will be made first. If *tcb* is not accessible due to server S_i's failure, then the backup copy tcb' will be referenced. If S_j fails while updating $tcb.commitFlag$, and $tcb'.commitFlag$, no access to $tcb'.commitFlag$ could have been performed and, hence, no inconsistency problem could occur. If S_i fails while updating $tcb.commitFlag$ and $tcb'.commitFlag$, then there is a possibility that two servers reference inconsistent values for $tcb.commitFlag$ and $tcb'.commitFlag$. However, the only possible update for $tcb.commitFlag$ is to change from *active* to *commit* or *abort*. As can be seen, this inconsistency does not hurt the correctness of the algorithm.

For actual implementation, it is better to implement the reliable TCB space in logically shared memory space (consisting of all the local memories of all servers). Each processor stores the TCBs in its local memory instead of its disk. Replication of the TCB space will also be done in the memory of a different processor. Without replication, it is not possible to store the TCBs in memory since the information will be lost after a failure. With replication and the assumption of at most one server failure at a time, the logically shared memory provides a highly efficient and reliable storage. Higher level of replication can be used if a more serious failure scenario is assumed.

Server status blocks can be replicated on all servers. There will be no need to update the server status block unless a server failure occurs. When it is necessary to make an update, the update request is broadcast to all servers. Since the server status block is highly replicated, each replica can be stored only in the main memory of each server instead of on the disk.

From each record, the pointer $TCBptr$ is used to locate the transaction control block and from each TCB, the pointer $serverPtr$ is used to locate the server status block. When a failure occurs, it is necessary to ensure the location of the backup copy of the control/status block via using these pointers. This can be done by maintaining a fixed mapping in terms of replication, i.e., S_i always

replicates its TCBs and server status block on S_j, where $j = f(i)$ and f is a fixed function known by all servers.

4 Analysis of the Algorithm

4.1 Correctness

We can show the correctness of a MIDS algorithm in two steps. First, we should show that after execution of the algorithm, the strong or weak invariant of the system will be satisfied. When there is no failure, the strong invariant should be satisfied. If a failure occurs, the weak invariant should be satisfied no matter at what execution point the failure occurred. Second, we need to show that as long as the weak invariant of the system is satisfied, the system can operate correctly. Instead of a formal proof, we will informally discuss the correctness of the algorithm in this section. Note that the correctness of individual database accesses (computation or update to records) during transactions processing is assumed.

When there is no failure, the MIDS-AT algorithm will execute the transaction to completion and then assign $TCBptr$ and $logPtr$ of all involved records to nil. Thus, the strong invariant will be satisfied after the complete execution.

A failure could occur at any point in the execution. Consider an arbitrary record r in the database. We use the same notation as discussed in Section 3. First, consider the satisfaction of $valid(r.logPtr)$ under the condition $r.TCBptr = tcb$ for some TCB tcb. If a failure occurs before we completely copy the content of r to a log, $r.logPtr = nil$. If $r.logPtr \neq nil$, then the copying action must have been completed and $content(r.logPtr) = v$ will be true. If a failure occurs after the assignment of $r.logPtr$ but before r is completely updated, then the $r.TCBptr.commitFlag$ could be *active* or *abort* and the predicate is satisfied. $r.TCBptr$ will not be set to *commit* since decision *commit* can only be set after the update to all records involved in the transaction has been done. If a failure occurs after the $commitFlag$ is set to *commit*, then we definitely have $content(r) = u(v)$. Thus, no matter when failure occurs, the execution of the MIDS-AT algorithm will bring the system to a state that satisfies the weak invariant of the system.

Next, we show that as long as the weak invariant is satisfied, the system will operate correctly. When $r.TCBptr = nil$, it is clear that no transaction is holding the record and the system can operate properly. If $r.TCBptr \neq nil$ then we know that there exists a TCB, tcb, where $r.TCBptr = tcb$. Let S_c be the coordinator of the transaction that holds r's lock. Also, let S_r denote the server that r resides on. From $tcb.commitFlag$, we can trace back and determine whether the coordinator S_c is alive. If S_c and S_r are alive, then the access to r can be done correctly and there is no problem.

Now, we consider that S_c had failed. If the server S_r also failed and it failed after sending a message to abort the subtransaction, then $tcb.commitFlag$ will either be *abort* or *active* (either

the coordinator S_c was active and set the *commitFlag* to *abort* or the coordinator failed and left the *commitFlag* *active* or the *commitFlag* was left to be *active* by the failed coordinator but was assigned to *abort* by an access and recovery to another record r' where $r'.TCBptr = tcb$). If S_r failed before sending any commit/abort decision, then *tcb.commitFlag* will also be either *abort* or *active* (either S_c, before failure, had a timeout on waiting for S_r's response and set *commitFlag* to *abort* or S_c had failed and left *commitFlag* *active* or the *commitFlag* was left to be *active* by the failed coordinator but was assigned to *abort* after an access and recovery to another record). In any of the cases discussed above, after access to r, *tcb.commitFlag* will be set to *abort*. Other records involved in the same transaction will also execute the abort action since the *tcb.commitFlag* has never been set to *commit* and its final flag is *abort*. Thus, the database as a whole is consistent. Correct recovery action for r then depends on its *logPtr*. If $r.logPtr = nil$, then r contains its original value and the record can be released. If $r.logPtr \neq nil$, we have shown that $content(r.logPtr) = v$ where v is the original value of r. Thus, r can be restored to its original value by simply copying $content(r.logPtr)$ to r. If a failure occurs during copying, $r.TCBptr$ still points back to *tcb*, $r.logPtr$ still points to $content(r.logPtr)$, $content(r.logPtr)$ is not changed, and $r.TCBptr.commitFlag$ will remain to be *abort*. Thus, recovery to r can be re-executed when r is accessed next time.

A final case is that S_r failed after sending a *commit* decision. It is possible again that the final decision is to abort the transaction. In that case, either the coordinator is alive and set *commitFlag* to *abort* or the transaction is aborted due to an inactive coordinator. In either of these cases, consistent abort actions will be taken as discussed above. If a failure of S_c occurs after it sets $commitFlag = commit$, then all the records involved in the transaction will take the *commit* action and the database as a whole is consistent. Consider record r. If $r.logPtr$ is set to nil, then r was not to be updated by the transaction and there is no problem. Thus, the only recovery action is set to $r.TCBptr$ to nil. If $r.logPtr \neq nil$, then we have shown that (when $tcb.commitFlag = commit$) $content(r) = u(r)$ which is what r supposed to be when the transaction is committed. In this case, $r.logPtr$ will simply be set to nil and then r is released by setting $r.TCBptr$ to nil. Consider the case where a failure occurs during recovery. If it occurs before $logPtr$ is set to nil, then nothing has been changed and the recovery can be re-done. If a failure occurs after $logPtr$ is set to nil and before $TCBptr$ is set to nil, then the record can still be recovered correctly since $logPtr$ is not useful here.

As we have shown above, as long as the weak invariant is maintained, the system will always work correctly. Also, the weak invariant is always satisfied no matter when a failure occurs during the execution of the MIDS-AT algorithm or the recovery algorithm.

4.2 Performance of MIDS-AT Algorithm

To simplify the performance analysis, we only consider the implementation of the MIDS-AT algorithm on a physically distributed storage system. We use the same assumptions and notation about the number of records (Kk) to be accessed, number of servers involved in the transaction (K), and record distribution as those stated in Section 2.2. The coordinator S_c needs to replicate each TCB and update both copies. This involves two extra messages for each TCB access (one from S_c to S_j requesting replication or update and one from S_j to S_c to report that the job is done) assuming that S_j keeps the TCB replica for S_c. Here, we consider storing the backup TCB and server status blocks in S_j's main memory, thus, the overhead for disk accesses is eliminated. Thus, for each transaction, 4 extra messages are required for the MIDS-AT algorithm compared with the conventional two-phase commit protocol due to the necessity for maintaining the backup copy of the TCB. Also, MIDS-AT requires 2 fewer disk accesses.

As in the case of the two-phase commit protocol, each server S_i needs to access each record on disk three times, once for reading and temporary locking, once for locking and logging, and once for final updating and unlocking. Here we do not consider database replication. Otherwise, the access to the records will require extra accesses to its backup copies and the comparison will not be fair since the availability of the records are provided if redundancy is considered. For communication among servers, there will be two messages from S_c to each S_i, for all i (one for S_c to send the subtransactions T_i to S_i and one for sending the final commitment decision to all S_i). Also, one message from each S_i to S_c is needed to send the commitment decision from S_i. Thus, the total overhead of the algorithm can be approximated as $3Kk$ disk accesses (consider storing TCB in memory) and $3K + 4$ messages. Thus, relative to the original two-phase commit protocol, there will be an overhead of 4 extra messages in the MIDS-AT algorithm for a single transaction processing.

4.3 Real-Time Property

In the MIDS-AT algorithm, when the coordinator of a transaction fails, each record locked by that transaction can be recovered and released independently. This property can be readily used for real-time applications. Consider the situation in which a high priority, real-time transaction T comes to the system. It is not a problem for a server S_i to switch from its current transaction execution to the execution of T since one server can execute many transactions concurrently. Assume that T needs to access records r while r is locked by S_j for a low priority transaction T'. In many conventional transaction processing approaches, T will have to wait for T' to finish execution and unlock r. In MIDS-AT approach, S_i can trace to S_j by checking $r.TCBptr.serverPtr$. To preempt the execution of T' and allow T to continue execution without being blocked, S_i can execute the

following code. In this code, tcb and tcb' denote the task control block of T and T', respectively.

```
S_i upon accessing r when executing T:
  if r.TCBptr ≠ nil then
    S_j := r.TCBptr.serverPtr;
    if S_j.status = active then
      signal S_j to discontinue the execution of T';
    endif;
    recover (r);
    r.TCBptr := tcb;
    .....
  endif;
```

If S_j is alive, it is necessary for S_i to signal S_j to stop accessing r since simultaneous access without protection can result in inconsistency. S_i, after signaling S_j, can proceed as though r is owned by an inactive server. In either cases, S_i executes the recover procedure on r as discussed in Section 3.2 and after recovery, reassigns $r.TCBptr$ to point to the new TCB.

Assume that the maximum time for accessing an unlocked record is t and the maximum time for recovery of a locked record is t_r. Then, the time bound for processing transaction T that accesses k records is $k(t + t_r)$. If a recovery phase is needed for preempting a transaction (such as the case of the two-phase commit protocol and its variants), then the time bound is not predicatable and, hence, is not suitable for real-time processing. As we have discussed above, MIDS-AT algorithm does not suffer from the recovery phase problem. It can handle priority preemption very easily and efficiently and, hence, is suitable for applications that requires high performance, real-time transaction processing.

4.4 Other Properties

Relative to the conventional two-phase commit protocol, the MIDS algorithm offers a non-blocking solution to atomic transaction processing. The cost of eliminating the blocking problem is small, specifically, 4 extra messages are required. One other major advantage of the MIDS approach is that the failure recovery procedure is extremely simple and records can be recovered and released independently. In contrast, the two-phase protocol has a complicated failure recovery phase and the successful completion of the recovery phase requires all servers involved in a transaction to be active. Compared to the three-phase commit protocol, the MIDS-AT scheme is much more efficient and can handle multiple failures if a higher level of replication of the TCB space is considered. For the approach where primary-backup replication is considered, a much higher overhead is incurred compared with the MIDS-AT algorithm. However, full replication does offer higher availability in terms of accesses to the database. The MIDS-AT algorithm also supports real-time processing, unlike those algorithms evolving from the two-phase commit protocol.

5 Implementation and Performance Evaluation

We implemented the MIDS-AT algorithm for experimental performance evaluation. The experimental environment consists of a cluster of Sun Workstations. Each server runs on an individual platform and communicates through sockets.

The simulation program consists of three components, database access simulation, robust distributed shared memory, and MIDS-AT algorithm. We use file accesses to simulate database accesses. Each file contains R records where R is a parameter to the system. When an access request arrives, the system computes the file location from the table in memory and then locates and accesses the record. To force an access to the disk instead of accessing the buffered content, we always flush the record to disk.

To facilitate the implementation of the robust TCB space, we implemented a robust shared memory on distributed platforms. The shared memory subsystem replicates and accesses the backup copies of records transparently. When a read request is received, the request is delivered to the primary site first. If the primary site is inactive, then the request is delivered to the backup site. When an update request is received, the request is delivered to the primary site and the primary site, after updating its own record, delivers the request to the backup site. If the primary site is inactive, then the request will be delivered to the backup site directly.

The MIDS-AT algorithm is implemented on top of the database access simulator and robust shared memory subsystem. When a client request arrives at a server, the server becomes the coordinator. It creates a TCB and issues a request to the robust shared memory subsystem to insert the TCB. The shared memory subsystem returns a pointer (index to the shared space) to the coordinator for future reference. The coordinator then sends subtransactions together with the TCB pointer to other servers involved in the transaction. As discussed in Section 4, each record on the disk has a $TCBptr$ field. If $TCBptr$ field of a record is nil, the record is not locked. To lock a record for a transaction, the server assigns the TCB pointer given by the coordinator to $TCBptr$ field of the record. If $TCBptr$ is not nil, then, the server will issue a request to the shared memory subsystem to reference the current $TCBptr$ and find out the status of the transaction that locked the record. The computation for the rest of the transaction processing is similar to conventional two-phase commit protocol.

We also implemented the two-phase commit protocol on the same set of platforms. The same set of requests are issued by the client and processed by both two-phase commit protocol and MIDS-AT algorithm. The average response time is measured for transaction processing. Some prelimiary performance results show that the difference in performance for MIDS-AT algorithm and conventional two-phase commit protocol is almost indistinguishable. Further performance data will be collected and presented in the final version of this paper.

6 Conclusion

In this paper, we discussed the application of the multiple invariant systems concept to real-time atomic transaction processing. The MIDS-AT algorithm we developed has many advantages. First, with a very insignificant overhead, namely, 4 extra messages relative to the two-phase commit protocol, the MIDS-AT algorithm achieves non-blocking transaction processing. Also, MIDS-AT is suitable for real-time processing due to its simple and inexpensive recovery operations that can be done on locked records independently. Thus, the MIDS-AT algorithm is a very promising approach for currently emerging E-commerce systems.

7 Acknowledgement

This work is supported in part by Texas Advanced Technology Program under Project Number 009741-0143-1999.

References

[1] A.J. Bernstein and PM. Lewis, *Concurrency in Programming and Database systems,* Jones and Bartlett, 1993.

[2] Garth A. Gibson, "Performance and reliability in redundant arrays of inexpensive disks," *Proc. Computer Measurement Group Annual Conference*, Reno, Nevada, Dec 1989.

[3] J. Gray, *Notes on Database Operating Systems*, vol. 60 of Lecture Notes in Computer Science, Springer-Verlag, 1978.

[4] C.B. Jones, "Tentative steps towards a development for interfering programs," *ACM Trans. Prog. Langs. and Sys.*, Vol. 5, No. 4, 1983, pp. 596-619.

[5] M. Raynal, *Distributed Algorithms and Protocols,* John Wiley and Sons, 1988.

[6] D. Skeen, "Nonblocking commit protocols," *Proceedings of ACM SIGMOD Conference on Management of Data*, June 1982.

[7] W. Weihl and B. Liskov, "Implementation of resilient, atomic data types," *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 2, April 1985, pp. 244-269.

[8] I.-L. Yen and F.B. Bastani, "Systematic incorporation of efficient fault tolerance in systems of cooperating parallel programs," *24th Intl. Symp. Fault-Tolerant Computing*, Austin, TX, June 1994, pp. 154-163.

[9] I-L. Yen and K. Karun, "Implementation and performance assessment of multilevel data structures," *21st International Computer Software and Applications Conference*, Bethesda, Maryland, August, 1997, pp. 153-157.

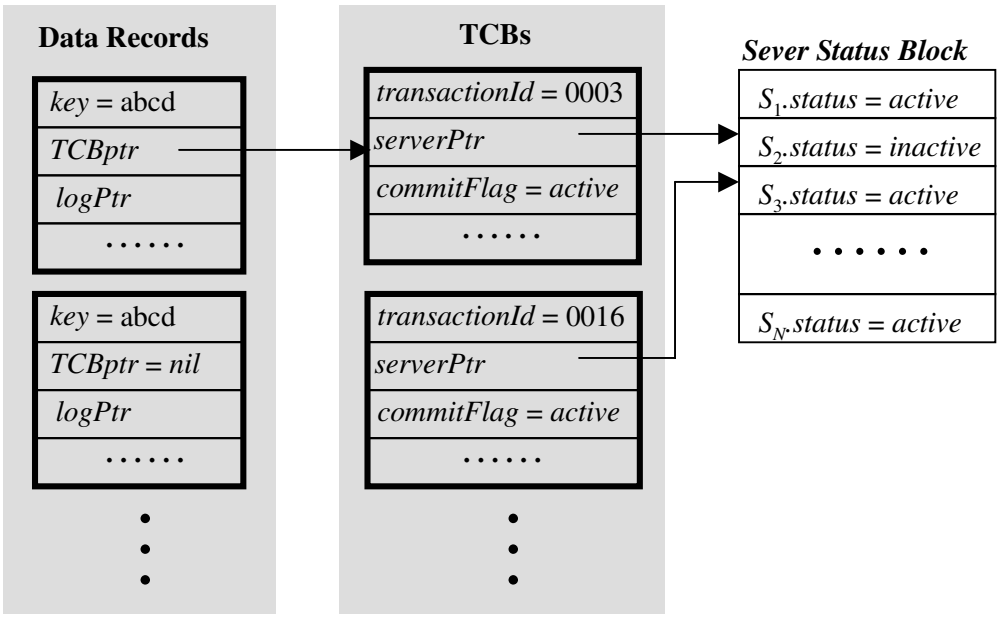

Figure 1. Data Structures for the MIDS transaction processing algorithm.