Efficient Damage Assessment and Repair in Resilient Distributed Database Systems*

Peng Liu, Xu Hao

Department of Information Systems
University of Maryland, Baltimore County
Baltimore, MD 21250
pliu@umbc.edu, xhao1@gl.umbc.edu

Abstract

Preventive measures sometimes fail to defect malicious attacks. With cyber attacks on data-intensive applications becoming an ever more serious threat, intrusion tolerant database systems are a significant concern. The main objective of intrusion tolerant database systems is to detect attacks, and to assess and repair the damage caused by the attacks in a timely manner such that the database will not be damaged to such a degree that is unacceptable or useless. This paper focuses on efficient damage assessment and repair in resilient distributed database systems. The complexity of distributed database systems caused by data partition, distributed transaction processing, and failures makes damage assessment and repair much more challenging than in centralized database systems. This paper identifies the key challenges and presents an efficient algorithm for distributed damage assessment and repair.

Key Words: Damage Assessment and Repair, Intrusion Tolerance, Database Security

1 Introduction

1.1 The Problem

Recently, people have seen more and more successful attacks on data-intensive applications, especially web-based e-business and e-government applications. Expanding cyber attacks pressure database management systems to not only prevent unauthorized access, but also tolerate intrusions. In fact, we must recognize that not all attacks – even obvious ones – can be averted at their outset. Attacks that succeed, to some degree at least, are unavoidable. As a result, people realize that with a focus on *preventing* users from gaining access beyond their authorities, traditional database security techniques such as authorization [10, 26, 13], inference control [1], multilevel secure databases [31, 28], and multilevel secure transaction processing [4], are very limited to handle successful attacks, or *intrusions*, and *intrusion tolerant* database systems that can detect intrusions, isolate attacks, contain, assess, and repair the damage caused by intrusions are a significant concern.

The focus of this paper is on a critical aspect of intrusion tolerant database systems, namely, damage assessment and repair. The primary goal of the attacks on a database, in most cases, is

*Liu was supported by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Material Command, USAF, under agreement number F30602-00-2-0575.

to damage the data stored in the database (in an undetectable way) and use the damaged data to mislead people to make wrong decisions. Compared with the attack of completely shutting down a database system, damaging a set of data items (or objects) can be much more dangerous, not only because they are more difficult to detect, but also because the damage can *spread* seriously in a database through benign operations. In particular, in a database the results of one transaction can affect the execution of some other transactions. Informally, when a transaction T_i reads an object x updated by another transaction T_j, T_i is directly *affected* by T_j. If a third transaction T_k is affected by T_i, but not directly affected by T_j, T_k is indirectly affected by T_j. It is easy to see that when a (relatively old) transaction B_i that updates x is identified *malicious*, the damage on x can spread to every object updated by a transaction that is affected by B_i, directly or indirectly. The job of damage assessment and repair is to locate each affected transaction and recover the database from the damage caused on the objects updated by the transaction. This job gets even more difficult as the execution of new transactions continues because the damage can spread to new transactions and cleaned objects can be re-damaged by new transactions. Therefore, the main objective of an intrusion tolerant database system is to guarantee that damage spreading is (dynamically) controlled in such a way that the database will not be damaged to a degree that is unacceptable or useless. Being the two key techniques to control damage spreading, damage assessment and repair are a significant concern.

One key technique for building intrusion tolerant database systems is intrusion detection, which has attracted many researchers [8, 20, 23, 19]. Intrusion detection systems monitor system or network activity to discover attempts to disrupt or gain illicit access to systems. The methodology of intrusion detection can be roughly classed as being either based on (statistical) *profiles* [14, 15, 27] or on known patterns of attacks, called *signatures* [11, 9, 25, 12, 29]. Effective damage assessment and repair cannot be achieved without effective intrusion detection because intrusion detection can tell which users or operations can cause damage, however, intrusion detection makes the system attack-aware but not attack-resilient, that is, intrusion detection itself cannot maintain the integrity and availability of the database in face of attacks.

This paper focuses on the techniques to assess and repair the damage caused in a *distributed* database. Distributed databases are widely used in a variety of real world data intensive applications, and we need efficient damage assessment and repair algorithms to make them resilient to attacks. Several algorithms are developed to assess and repair the damage caused in a centralized database [3, 24, 17], however, the complexity of distributed database systems caused by data partition, distributed transaction processing, and failures suggests that they cannot be used in distributed database systems and new distributed damage assessment and repair algorithms are needed. First, since data are partitioned and stored on multiple *sites*, damage assessment and repair must be performed at multiple sites. Second, since one distributed transaction may access data stored at multiple sites, damage assessment, which is now based on the affecting relationships among distributed transactions, needs to coordinate and synthesize the affecting relationships among the subtransactions at each site. Third, distributed damage assessment and repair need to tolerate not only site failures but also communication failures.

Finally, it should be noticed that although our approach focuses on surviving attacks by malicious transactions, database intrusion tolerance can also be enforced at some lower levels such as *OS level*. However, although *transaction level* methods cannot handle OS level attacks, it is shown that in many applications where attacks are enforced mainly through *malicious* transactions transaction level methods can tolerate intrusions in a much more effective and efficient way. Moreover, it is shown that OS level intrusion tolerance techniques such as those proposed in [20, 19, 21, 5], can be directly integrated into a transaction level intrusion tolerance framework to complement it with the ability to tolerate OS level attacks.

1.2 Prior Work

Database recovery mechanisms are not designed to deal with malicious attacks. Undo and redo operations can only be performed on uncommitted or *active* transactions, and the durability property ensures that traditional recovery mechanisms never undo committed transactions [6]. However, the fact that a transaction commits does not guarantee that its effects are desirable. Specifically, a committed transaction may reflect malicious activity.

Although our assessment and repair model is related to the notion of *cascading abort* [6], cascading aborts only capture the *read-from* relation between active transactions, and in standard recovery approaches cascading aborts are avoided by requiring transactions to read only committed data [16].

Some specific damage assessment and repair techniques are developed in the context of centralized database systems [24, 3]. These techniques can be broken down into two categories: *data-oriented* methods [24], and *transaction-oriented* methods [3, 17]. Data-oriented methods use the read and write operations of transactions to trace the damage spread from one data object to another, and compose a specific piece of code to repair each damaged data object. Transaction-oriented methods assess and repair the damage by identifying and backing out affected transactions. Data-oriented methods are more flexible and better at handling *blind* writes, however, composing cleaning code for each data object can be time consuming and prone to errors. Transaction-oriented methods use a cleaning transaction, which can be easily composed, to repair multiple data objects at the same time, thus they are more robust and efficient.

1.3 Our Approach and Contribution

In this paper, we extend centralized damage assessment and repair techniques to distributed systems, in particular, we present an efficient on-the-fly transaction-oriented algorithm to assess and repair the damage caused in a distributed database. The algorithm has the following properties:

- It is totally distributed, thus it has no single point of failures and it can tolerate malicious attacks on end systems and communication links.
- It is on-the-fly. New transactions are continuously executed during the damage assessment and repair process.
- It is efficient. Damage assessment and repair are performed, in parallel, at each participating site.
- It works seamlessly with a commercial DDBMS, thus it can be directly applied to build resilient distributed database applications.
- It provides transparent damage assessment and repair.

The rest of the paper is organized as follows. In Section 2, we present the model for distributed damage assessment and repair. We present the damage assessment and repair algorithm in Section 3. In Section 4, we address some additional considerations of the algorithm. We conclude the paper in Section 5.

2 The Model

2.1 Distributed Database Model

A distributed database is a set of data *items* (or objects) stored across a set of *sites* connected by a network. Each data item is stored at one and only site. A distributed transaction is a partial order of *read* and *write* operations on these data items that either *commits* or *aborts*. Two operations *conflict* if one is write. Two transactions *conflict* if they have conflicting operations. The execution of a set of distributed transactions is modeled by a (distributed) *history*. The model of a distributed transaction execution is such that there is one process, called the *coordinator*, that is connected to the user application and a set of other processes, called the *subordinates*. Transactions are assumed to have globally unique identifiers. The processes are assumed to have globally unique identifiers, which also indicate the sites (or locations) of the corresponding processes. For simplicity, we assume that each site participating in a distributed transaction has only one process of that transaction.

We assume the distributed database system supports transaction serializability and uses the two-phase locking (2PL) protocol. We assume the two-phase commit (2PC) protocol is used to guarantee the atomicity of distributed transactions, despite intermittent site or communication link failures. We assume the conventional 2PC protocol is used here, and the protocol writes and force-writes log records as follows. When the user decides to commit a transaction, the coordinator will send *PREPARE* messages, in parallel, to the subordinates. Each subordinate that is willing to let the transaction be committed first force-writes a *prepare* log record and then sends a *YES VOTE* to the coordinator. Each subordinate that wants the transaction aborted force-writes an *abort* record, sends a *NO VOTE* to the coordinator, and aborts the transaction. After the coordinator receives the votes from all the subordinates, if all the votes were *YES VOTE*s, the coordinator will force-write a *commit* record and send *COMMIT* messages to all the subordinates. If the coordinator has received even one *NO VOTE*, it will force-write an *abort* record and sends *ABORT*s to the subordinates that vote *YES*. Each subordinate, after receiving a *COMMIT*, force-writes a *commit* record and sends an *ACK* message to the coordinator. Each subordinate, after receiving an *ABORT*, force-writes an *abort* record. The coordinator, after receiving the *ACK*s from all the subordinates that were sent a message in the second phase, writes an *end* record. It should be noticed that although our damage assessment and repair algorithm is specified based on the way log records are maintained, our algorithm can be easily extended to support other 2PC protocols such as presumed abort (PA) and presumed commit (PC) [22].

2.2 Damage Assessment and Repair Model

Our idea of damage assessment and repair is that only the malicious transactions and the innocent transactions affected by a malicious transaction, directly or indirectly, can cause damage and should be repaired. In order to make this idea more tangible, we first formally define the affecting relationship among transactions. Since damage can only be caused by committed transactions, our definitions care about only committed transactions. We denote the set of committed malicious transactions in a history by $\mathbf{B} = \{B_1, B_2, ..., B_m\}$. We denote the set of committed innocent transactions by $\mathbf{G} = \{G_1, G_2, ..., G_n\}$.

Definition 1 At a site, a committed subtransaction T_{jk} is *dependent upon* committed subtransaction T_{il} if there exists a data item x stored at the site such that T_{jk} reads x after T_{il} has updated x, and no other committed subtransactions update x between the time T_{il} updates x and T_{jk} reads x.

Definition 2 A committed distributed transaction T_j is *dependent upon* committed distributed transaction T_i in a history if one of T_j's subtransactions is dependent upon one of T_i's subtransactions.

Definition 3 In a history, a committed distributed transaction T_i *affects* committed distributed transaction T_j if the ordered pair $(T_j,\ T_i)$ is in the transitive closure of the *dependent upon* relation.

It is clear that the notation "affect" captures both direct and indirect affecting relationships. To illustrate, consider the following history over (B_1, G_1, G_2), which are assumed to be executed on two sites. Here r means a read, w means a write, and c means a commit.

At site A: $H_A = r_{B_{1A}}[x]w_{B_{1A}}[x]c_{B_{1A}}r_{G_{1A}}[x]r_{G_{1A}}[y]w_{G_{1A}}[y]c_{G_{1A}}r_{G_{2A}}[z]w_{G_{2A}}[z]c_{G_{2A}}$
At site B: $H_B = r_{B_{1B}}[u]w_{B_{1B}}[u]c_{B_{1B}}r_{G_{1B}}[v]w_{G_{1B}}[v]c_{G_{1B}}r_{G_{2B}}[v]r_{G_{2B}}[w]w_{G_{2B}}[w]c_{G_{2B}}$

Based on H_A, we find that G_{1A} is dependent upon B_{1A}, so G_1 is affected by B_1, and x and y are damaged. We find that G_{2A} is neither dependent upon B_{1A} nor G_{1A}. However, this does not ensure that G_2 will not be affected by B_1. Based on H_B, we find that G_{2B} is dependent upon G_{1B} (although G_{2B} is not dependent upon B_{1B}), so G_2 is dependent upon G_1. Since G_1 is dependent upon B_1, so $(G_2,\ B_1)$ is in the transitive closure of the dependent upon relation. Therefore, G_2 is actually affected by B_1 and w is damaged.

3 The Damage Assessment and Repair Algorithm

3.1 High-Level Description

We use two types of processes to do distributed damage assessment and repair (DAR): a *Local DAR Manager* on each site, and a *Local DAR Executor* on each site. The idea is to do distributed damage assessment and repair through local operations. At a site, the Local DAR Executor is responsible for scanning the local log to (a) identify the subtransactions that are affected by a malicious transaction and (b) compose the corresponding cleaning subtransactions for these subtransactions; the Local DAR Manager is responsible for coordinating the DAR process of (only) the distributed transactions that had their coordinators at the site and are affected by a malicious transaction. In particular, for each such transaction T_i, the Local DAR Manager will create the coordinator for T_i's cleaning transaction, and instruct (only) the Local DAR Executors at the sites where T_i's subordinates were located to assess the damage spread by T_i and to repair T_i. Cleaning transactions are executed just like an normal transaction.

The example described in Section 2 shows that in order to capture every transaction affected by a malicious transaction, global coordination among sites is necessary, since a transaction affected by a malicious transaction may be found unaffected at some of its subordinates' sites if only local information is used. Our algorithm does global coordination by making the Local DAR Managers and Executors collaborating with each other. In particular, after **B** is identified, it will be first sent, in parallel, to the Local DAR Managers at the sites where **B**'s coordinators were located. These Local DAR Managers then send an *ASSESS* message to ask each of the Local DAR Executors at the sites where **B**'s subordinates are located to assess the damage caused by **B** on their local data. The identifier of each affected (sub)transaction found by the Local DAR Executors then will be sent, in parallel, to the Local DAR Managers at the sites where the coordinators of these affected transactions were located, which will then send an *ASSESS* message to ask each of the Local DAR Executors at the sites where the other subordinates of these affected transactions were located to assess the damage indirectly caused by **B**. This kind of interaction between Local DAR managers and Executors will be *continuously* performed until all the damage is located.

The repair process is relatively simpler. The damage caused by each malicious or affected distributed transaction is repaired by a distributed cleaning transaction. The Local DAR Managers are responsible for creating the coordinators for these cleaning transactions. The Local DAR Executors are responsible for composing the cleaning subtransactions that comprise each cleaning transaction. It should be noticed that it is not guaranteed that a cleaning subtransaction will really clean each item it *wants* to clean. Sometimes, an item can be really cleaned after several cleaning subtransactions have updated the item. The reason is that cleaning subtransactions are composed only based on local information, which may not be complete.

3.2 Data Structures

Our algorithm is based on the transaction log maintained on each site. We assume every *read* (operation) is recorded in a log. Although this is not the case for most commercial distributed database systems, [3] shows that (a) in addition to logging reads there are several other ways to collect reads such as extracting read sets from transaction profiles, and (b) the algorithms based on in-log reads can be easily extended to use the reads maintained outside the logs. We assume each log has six types of records, namely, *read*, *write*, *abort*, *commit*, *prepare*, and *end* records. The other possible types of records such as *savepoints* and *checkpoints* are not considered here because they are irrelevant. We assume the log records at each site contain the type (*read*, *prepare*, *commit*, etc.) of the record, the identifier of the application, the identifier of the transaction, the identifier of the coordinator, the identities of the subordinates in the case of the *commit/abort* records written by the coordinator [22].

In addition to logs, our algorithm uses the following data structure:

- an *Cleaned Item Table* (CIT) at each site that contains the items that have been 'cleaned' at the site. Each item in this table is tagged with two numbers: $cLSN$, which indicates when the item is damaged, and $eLSN$, which indicates when the item is cleaned.

These data structures are described in Figure 1. Here $ItemID$ means the identifier of a data item. LSN, means log serial number, a unique number associated with each log record. LSN becomes larger as the log grows.

3.3 Damage Assessment and Repair Under Normal Operation

First, we describe the algorithm without considering failures. The algorithm is as follows. Note that here for clarity we assume during the DAR process no new malicious transactions will be added to **B**, although the algorithm can be easily extended to support newly identified malicious transactions.

Algorithm 1 Distributed On-the-fly Damage Assessment and Repair Algorithm
Input: **B**, the logs
Output: when the DAR process terminates, any prefix H_p of the history including the point where the process terminates results in the state that would have been produced by H'_p, where H'_p is H_p with all the transactions in **B** or affected by **B** removed
Initialization:
- make P, a pointer, pointed to the header of the log, which is before any log records;
- $assess\_list := \{\}$, $to\_assess := \{\}$, $undo\_list := \{\}$, $tmp\_undo\_list := \{\}$, $dirty\_item\_set := \{\}$, $tmp\_item\_set := \{\}$, $submitted\_item\_set := \{\}$; /* **See comment A** */

```
CIT: array[ItemID] of record /* Cleaned Item Table */
  cLSN;
      /* the LSN of the record that records the write that damages the item */
  eLSN;
      /* the LSN of the end record of the log when the item is cleaned */
end;
LF: persistent array[LSN] of record /* Stable Log File */
  LogRecType: (read, write, abort, commit, prepare, end);
  LogRecContents: array of char;
  AppID: integer;
  TransID: integer;
  CoordinatorID: integer;
  [For commit/abort records] Subordinates: array of integer;
end;
```

Figure 1: Major Data Structures

At the Local DAR Manager:
while TRUE
  if an *ASSESS* message arrives
    if the transaction (indicated by the message) is in the *assess_list*
      do nothing;
    if the transaction is not in the *assess_list*
      add the identifier of the transaction to the *assess_list*;
      create a coordinator (process) for the cleaning transaction of the transaction, and inform the
        coordinator the set of subordinates of the transaction;
      if the *ASSESS* message is not sent out by a Local DAR Executor
        send an *ASSESS* message, which contains the identifier of the coordinator process
          and the transaction identifier, in parallel, to (only) the Local DAR Executors at the sites
          where the transaction had a subordinate;
      otherwise the *ASSESS* message is sent out by a Local DAR Executor, denoted E_i
        send an *ASSESS* message, which contains the identifier of the coordinator process
          and the transaction identifier, in parallel, to (only) the Local DAR Executors, except for E_i, at
          the sites where the transaction had a subordinate;
        send a *CLEAN COORDINATOR* message, which contains the identifier of the coordinator process
          and the transaction identifier, to E_i; /* **See comment B** */
end while

At the Local DAR Executor:
while TRUE
  if a *CLEAN COORDINATOR* message arrives
    associate the identifier of the coordinator with the transaction kept in the *undo_list* ;
  if an *SUCCESS* message (sent from a coordinator of a cleaning transaction) arrives
    move each item x that has been cleaned from the *dirty_item_set* to the *CIT* table, and
      associate x with the *LSN* of the current end of the log (namely $x.eLSN$);
    remove x from the *submitted_item_set*;
  if an *ASSESS* message arrives
    if this is the first *ASSESS* message
      make P pointed to the first log record of the transaction;
    elseif the transaction is not in the *undo_list* /* **See comment C** */
      add the transaction (indicated by the message), together with the coordinator identifier, to *to_assess*;

if the first log record of the transaction is before the record pointed by P /* **See comment D** */
    if the transaction is not in the *tmp_undo_list*
        make P pointed to the first log record of the transaction;
    else remove the transaction from the *tmp_undo_list*, and move the items that are written by the transaction and kept in the *tmp_item_set* to the *dirty_item_set* /* **See comment E** */
if P points to a record and the termination conditions do not hold /* **See comment F** */
    Scan the log entry pointed by P:
    if the entry is for a cleaning transaction
        skip it;
    elseif the entry is for a transaction T_i in *to_assess*
        if the entry is a write record $[T_i, x, v_1, v_2]$
            if x is not in the CIT table
                add x, together with the LSN of the write record (denoted $x.cLSN$), to the *dirty_item_set*;
            elseif $x.eLSN < w.LSN$ or $w.LSN < x.cLSN$ /* **See comment G** */
                add $(x, x.cLSN)$ to the *dirty_item_set*;
        elseif the entry is a commit record $[T_i, commit]$
            compose the cleaning subtransaction for T_i; /* **See comment H** */
            send the composed cleaning subtransaction to its coordinator (the identifier of the coordinator can be found in *to_assess*);
        elseif the entry is of other types
            skip it;
    else
        case the entry is a write record $[T_i, x, v_1, v_2]$
            if x is not in the CIT table
                add $(x, x.cLSN)$ to the *tmp_item_set*;
            elseif $x.eLSN < w.LSN$ or $w.LSN < x.cLSN$
                add $(x, x.cLSN)$ to the *tmp_item_set*;
        case the entry is a read record $[T_i, x]$
            if x is in the *dirty_item_set* and $r.LSN > x.cLSN$ /* **See comment I** */
                add T_i to the *tmp_undo_list*;
            elseif x is in the CIT table and $r.LSN < x.eLSN$ /* **See comment J** */
                add T_i to the *tmp_undo_list*;
        case the entry is an abort record $[T_i, abort]$
            delete all the data items of T_i from the *tmp_item_set*;
            if T_i is in the *tmp_undo_list*, remove it;
        case the entry is a commit record $[T_i, commit]$
            if T_i is in the *tmp_undo_list*
                move all the items of T_i from the *tmp_item_set* to the *dirty_item_set*;
                move T_i from the *tmp_undo_list* to the *undo_list*;
                send an $ASSESS$ message, which contains the identifier of the transaction, to the Local DAR Manager at the site where the coordinator of T_i was located;
                compose the cleaning subtransaction for T_i;
                send the composed cleaning subtransaction to its coordinator (the identifier of the coordinator can be found in the *undo_list*);
            else delete all the items of T_i from the *tmp_item_set*;
        case the entry is of other types
            skip it;
    $P = P + 1$;
end while

At a Cleaning Transaction Coordinator:
while TRUE
    if a cleaning subtransaction arrives
        if the cleaning subtransaction is not *empty*
            send the cleaning subtransaction to its (corresponding) subordinate for execution;
        else
            do nothing;

```
    if all the cleaning subtransactions have arrived /* See comment K */
        exit;
end while
run the 2PC protocol to commit the cleaning transaction;
when the cleaning transaction commits, send a SUCCESS message to each Local DAR Executor
that submits an un-empty cleaning subtransaction;
```

Comments

A. Inside a Local DAR Manager, the *assess_list* contains (the identifiers of) the transactions that have been reported to the Local DAR Manager. Inside a Local DAR Executor, the *submitted_item_set* contains the items that are being cleaned but still not cleaned. The *to_assess* contains the set of transactions in **B** that has a subordinate at the site, and the set of innocent transactions that had a subordinate at the site, are affected by **B** but are not identified by the Local DAR Executor. The *undo_list* contains the transactions affected by **B** and identified by the Local DAR Executor. The *tmp_undo_list* contains the transactions that are found affected but may abort later on. The *dirty_item_set* contains the damaged items. The *tmp_item_set* contains the items that could have been damaged. When a Local DAR Executor scans an innocent transaction, it does not know whether or not the transaction will commit later on. Even if the transaction reads a damaged item, it will cause no damage if it aborts later on. Therefore, we use the *tmp_item_set* and the *tmp_undo_list* to contain the information about which transactions may have caused damage, and which items may have been damaged. In this way, we need not to re-collect this information when a transaction commits.

B. Although E_i does not need the *ASSESS* message, E_i still needs the identifier of the coordinator process to repair the damage caused by T_i at E_i's site.

C. If the transaction T is in the *undo_list*, it is already found affected at the site, and an *ASSESS* message M has already been sent to a Local DAR Manager. The reason that this site still receives an *ASSESS* message for T is because the Local DAR Manager receives another *ASSESS* message for T before M.

D. Given the transaction is not in the *undo_list*, if it is not in the *tmp_undo_list*, it is still not found affected at the site. Hence if its first log record is before P, then the Local DAR Executor had not assessed the damage caused by the transaction when it was scanned, so we need to rescan the transaction and its following transactions.

E. Although the transaction is in the *tmp_undo_list*, since the transaction is already added to *to_assess*, the items of the transaction kept in the *tmp_item_set* will no longer be handled. Hence we need to move them now to the *dirty_item_set*.

F. Since the algorithm allows users to continuously execute new transactions as the damaged items are identified and cleaned, new transactions can spread damage if they read a damaged but still unidentified item. So we face two critical questions: (1) Will the DAR process terminate? (2) If the DAR process terminates, can we detect the termination? Fortunately, [3] has answered these two questions in the context of centralized intrusion tolerant database systems. Here we give similar answers for distributed systems. First, when the damage spreading speed is quicker than the damage repair speed, the DAR process may never terminate. When the damage repair speed is quicker, the DAR process will terminate. Second, under the following conditions we can ensure that the DAR process terminates: (1) the cleaning transaction for every (reported) malicious transaction has been committed; (2) at each site, $dirty\_item\_set = \emptyset$; (3) at each site, $tmp\_undo\_list = \emptyset$, and $\forall x \in CIT$, $x.eLSN < l.LSN$. Here, $l.LSN$ denotes the LSN of the next log record for the Local DAR Executor to scan. (4) At each site, the Local DAR Manager has no $ASSESS$ message to send, and the Local DAR Executor has no $ASSESS$ message to respond. Condition 1 ensures that no malicious transaction is needed to repair. Condition 2 ensures that every identified damaged item is cleaned. Conditions 3 and 4 ensure that further scans will not identify any new damage. Please refer to [3] for more details about termination detection.

G. $w.LSN$ denotes the LSN of the write record. If $w.LSN > x.eLSN$, x is cleaned before the write operation, therefore, the write damages x again. If $w.LSN < x.cLSN$, this write is before the write that has been cleaned, so this write damages x before the cleaned write. So x is not really cleaned (by previous cleaning transactions).

H. The composing algorithm works as follows: for each item x written by T_i,

- **if** x is in the CIT table or the $submitted\_item\_set$, and $x.cLSN$ is larger than both the $cLSN$ value of x kept in the $clean\_item\_set$ (if any) and the $cLSN$ value of x kept in the $submitted\_item\_set$ (if any), then add no cleaning operation of x to the cleaning subtransaction of T_i (because x is already or to be 'better' cleaned);
- **otherwise**, add a write operation, which restores x to its previous value (before the write), to the cleaning subtransaction of T_i, and add $(x, x.cLSN)$ to the $submitted\_item\_set$.

It should be noticed that under some situations, for example, when T_i at the site is read-only, or when all the items written by T_i have been 'better' cleaned, an empty cleaning subtransaction may be composed.

I. $r.LSN$ denotes the LSN of the read record. If $r.LSN > x.cLSN$, x is read after x is damaged.

J. If $r.LSN < x.eLSN$, x is still not cleaned when x is read.

K. Since the coordinator was informed by its Local DAR Manager the list of subordinates of the transaction to clean, it knows the sites from which a cleaning subtransaction is expected.

3.3.1 An Example

We first use an example to show how the algorithm works, then we explain why the algorithm is correct in next section. Consider the following history. Assume the coordinator of B_1 is at site A, the coordinator of B_2 is at site B, the coordinator of G_1 is at site C, the coordinator of G_2 is at site D, the coordinator of G_3 is at site B. Assume the commit order is B_1, B_2, G_1, G_2, and G_3. For simplicity, only malicious transactions and the transactions affected by malicious transactions are included, and only the log records kept by subordinates are shown. Here p means a prepare record.

At site A: $H_A = r_{B_{1A}}[x]w_{B_{1A}}[x]p_{B_{1A}}c_{B_{1A}}r_{G_{1A}}[x]r_{G_{1A}}[y]w_{G_{1A}}[y]p_{G_{1A}}c_{G_{1A}}$
At site B: $H_B = r_{B_{2B}}[u]w_{B_{2B}}[u]p_{B_{2B}}c_{B_{2B}}r_{G_{3B}}[u]r_{G_{3B}}[v]w_{G_{3B}}[v]p_{G_{3B}}c_{G_{3B}}$
At site C: $H_C = r_{B_{2C}}[s]w_{B_{2C}}[s]p_{B_{2C}}c_{B_{2C}}r_{G_{1C}}[t]r_{G_{2C}}[s]r_{G_{2C}}[o]w_{G_{1C}}[t]w_{G_{2C}}[o]$
$p_{G_{1C}}p_{G_{2C}}c_{G_{1C}}c_{G_{2C}}$
At site D: $H_D = r_{B_{1D}}[p]w_{B_{1D}}[p]p_{B_{1D}}c_{B_{1D}}r_{G_{2D}}[q]r_{G_{3D}}[p]r_{G_{3D}}[r]w_{G_{2D}}[q]w_{G_{3D}}[r]$
$p_{G_{2D}}p_{G_{3D}}c_{G_{2D}}c_{G_{3D}}$

The process starts when two $ASSESS$ messages are sent, in parallel, to the Local DAR Managers located at sites A and B, respectively (Note that B_1 and B_2 can be simultaneously identified by a distributed intrusion detector). For efficiency the two $ASSESS$ messages can be sent out by the PEM or the Intrusion Detector.

After the Local DAR Manager at site A, where the coordinator of B_1 was located, receives the $ASSESS$ message, it will add B_1 to the *assess_list*, create a coordinator, denoted C_{B_1}, for the cleaning transaction of B_1, then send out two $ASSESS$ messages to the Local DAR Executors at sites A and D, where the two subordinates of B_1 were located, respectively. At the same time, the Local DAR Manager at site B will do similar things for B_2.

After the Local DAR Executor (Executor for short) at site A receives the $ASSESS$ message (for B_1), it will add the identifiers of B_1 and C_{B_1} to *to_assess*, which is previously empty. Since the $ASSESS$ message is the first $ASSESS$ message received by the Executor, the Executor will make P pointed to the first log record of B_1, then start to scan the log from P. The Executor will add $(x, x.cLSN)$ to the *dirty_item_set* when $w_{B_{1A}}[x]$ is scanned, compose the cleaning subtransaction, which contains only a $w[x]$ operation, and send it to C_{B_1} when $c_{B_{1A}}$ is scanned. Then the Executor will continue scanning the log, add G_1 to the *tmp_undo_list* since x is in the *dirty_item_set* when $r_{G_{1A}}[x]$ is scanned, add y to the *tmp_item_set* when $w_{G_{1A}}[y]$ is scanned. When $c_{G_{1A}}$ is scanned, the Executor will first move y from the *tmp_item_set* to the *dirty_item_set*, then move G_1 from the *tmp_undo_list* to the *undo_list*, then send an $ASSESS$ message for G_1 to the Local DAR Manager at site C where the coordinator of G_1 was located, then compose the cleaning subtransaction for G_{1A}. Since y is not in the CIT table, the subtransaction contains $w[y]$. Finally, the Executor wants to send the cleaning subtransaction to its coordinator. However, the coordinator is known to the Executor only after the Local DAR Manager at site C creates the coordinator and sends a $CLEAN$ $COORDINATOR$ message to the Executor.

After C_{B_1} receives the cleaning subtransaction from the Executor at site A, it will execute the subtransaction as part of the cleaning transaction for B_1 and use 2PC to commit the transaction. Note that the cleaning transaction cannot commit until the other subtransaction is composed at site D, sent to C_{B_1}, and executed.

At the same time, the Executor at site D will receive the $ASSESS$ message for B_1, the Executors at sites B and C will receive the $ASSESS$ message for B_2. And they will do similar things as what the Executor at site A does.

During the time when the Executor at site C is scanning its log, the $ASSESS$ message for G_1 sent out from the Local DAR Manager at site C can arrive. We assume the message arrives after $c_{G_{1C}}$ is scanned, so P is now pointed to $c_{G_{2C}}$. Since G_{1C} is already scanned, it is found not affected. However, we have to go back to the log record where G_{1C} starts and scan the log again to see if any transactions are affected by G_1. In particular, G_1 will be added to *to_assess*, and P will be pointed to first record of G_{1C}, then the log will be scanned (from P). After a while, t will be added to the *dirty_item_set*, a cleaning subtransaction for G_{1C} will be composed and sent out. When $c_{G_{2C}}$ is scanned, G_2 will be moved to the *undo_list*, o will be moved to the *dirty_item_set*, and a cleaning subtransaction for G_{2C} will be composed and sent to its coordinator at site D.

At the same time, similar things can be done at site D. As a result, a cleaning subtransaction for G_{2D} will be sent to its coordinator at site D, a cleaning subtransaction for G_{3D} will be sent to its coordinator at site B.

After a while, the cleaning transaction of G_1 will commit, and two $SUCCESS$ messages for G_1 will be sent to sites A and C, respectively. Then the Executor at site C will move t from its *dirty_item_set* to its CIT table; the Executor at site A will move y from its *dirty_item_set* to its CIT table. In a similar way, the cleaning transactions for B_1, B_2, G_2, and G_3 will finally commit.

After all the *SUCCESS* messages are received and processed, the whole DAR process will terminate.

3.3.2 Analysis of the Algorithm

Our algorithm has the following characteristics.

Claim 1. An ASSESS *message for transaction T is sent out by a Local Executor only if T is affected by a malicious transaction.*

Claim 2. For each innocent transaction affected by a malicious transaction (in **B***), the Local DAR Executor at every site where the transaction had a subordinate will assess and repair the damage caused by the transaction.*

Claim 3. When the DAR process terminates, every item that is updated by a transaction in **B** *or affected by* **B** *will be restored to the value before the item is damaged.*

Claim 4. If the DAR process terminates, the algorithm can detect the termination.

Based on these characteristics, the correctness of our algorithm can be specified as follows.

Theorem 1 Algorithm 1 is correct in the sense that (a) if the DAR process terminates, the algorithm can detect the termination and stop scanning the logs, and (b) when the DAR process terminates, any prefix H_p of the history including the point where the DAR process terminates results in the state that would have been produced by H'_p, where H'_p is H_p with all the transactions in **B** or affected by **B** removed.

3.4 Damage Assessment and Repair Under Failures and System Attacks

Let us now consider site and communication *failures*. We not only consider the failures caused by accidental *errors*, but also consider the failures caused by intentional attacks. In particular, during a DAR process, we assume a site can *crash* due to errors, we also assume a site can be attacked to crash. We assume due to errors or attacks, a message can be corrupted, lost (or dropped), faked, or replayed. We also assume a communication link can be broken for a while due to errors or attacks. We finally assume all failed sites ultimately recover, and all broken communication links are ultimately reconnected.

The impact of failures is two folds: First, corrupted, lost, faked, or replayed messages among the DAR Managers, the DAR Executors, and the cleaning transaction coordinators can cause a transaction to be assessed and repaired multiple times, can cause an innocent unaffected transaction to be "repaired" (rolled back), and can cause some malicious or affected transactions to stay at large. Second, crashed sites or broken communication links can disable some message interactions among the DAR Managers, the DAR Executors, and the cleaning transaction coordinators. In particular, when a site is unreachable during a period of time, its DAR Manager cannot receive (send) *ASSESS* messages from (to) DAR Executors at other sites; its DAR Executor cannot send *ASSESS* messages to DAR Managers at other sites; and its cleaning transaction coordinators cannot execute cleaning transactions at other sites. As a result, some information about affected transactions (generated at one site) cannot be distributed to other relevant sites in a timely manner, and some damaged items at one site can only be identified (and repaired) after another site becomes reachable.

The algorithm proposed in Section 3.3 can be easily extended to tolerate failures. First, our discussion in next section, namely Section 4, indicates that standard secure communication techniques such as IPSEC can be used to ensure that the messages between a Local DAR Manager and a Local DAR Executor or between a Local DAR Executor and a cleaning transaction coordinator will not be changed, dropped, faked, or replayed. Second, in order to tolerate failed sites and broken communication links, we assume that at each active site a *recovery process* exists and that it processes all messages from recovery processes at other sites and handles all the recovery tasks assigned by the Local DAR Manager and the Local DAR Executor. We assume that cleaning transaction coordinators (at each site) use standard distributed database fault tolerance services to execute cleaning transactions.

At an active site, when the Local DAR Manager or the Local DAR Executor find that a message cannot be sent out, they will forward the message, together with the information about who should receive the message, to the local recovery process, then "forget" the message and continue working. It is the responsibility of the recovery process to resend these unsuccessful messages when they can be sent out. When a site recovers, its recovery process will broadcast (or multicast) a $RESTART$ message to all the other recovery processes, which will then send out all the messages for the recovered site.

Since a site may crash at any point of time, we need to periodically keep state information for the DAR components so that they need not restart the DAR process from the very beginning when a site recovers. In particular, at each site we assume a message log is kept for each DAR component, including the Local DAR Manager, the Local DAR Executor, the local recovery process, and the local cleaning transaction coordinators. Whenever a DAR component receives a message, the message will be first logged. In addition, we assume each DAR component periodically takes a checkpoint of its current state. When a message for a DAR component is logged, the message is tagged with the latest checkpoint taken by the component. In this way, when a site recovers, each DAR component, starting from its latest checkpoint, knows exactly which messages should be reprocessed. Moreover, when a message for the Executor is logged, it will also be tagged with the LSN of the log record that is currently scanned. In this way, the Executor can synchronize messages and scan operations.

The above method can ensure that every message will be processed, but it can neither ensure that every message will be received only once nor ensure that messages will be received in their original (sending-out) orders. Fortunately, we can use the message log kept for each DAR component to filter off the messages that had been received. Moreover, our algorithm can tolerate mis-ordered messages. For each cleaning transaction coordinator, the arriving order of cleaning subtransactions does not matter. For each DAR Manager, if two $ASSESS$ messages do not arrive in the original order, they will be processed in the wrong order. As a result, at a site even if T_i is dependent upon T_j, the Local DAR Executor could receive the $ASSESS$ message for T_i before that for T_j (Note that this can happen even if each Local DAR Manager and Local DAR Executor receives $ASSESS$ messages in their original orders). Fortunately, the correctness of our Executor algorithm, which is based on the (absolute) time order indicated by log records, is not affected by the order of arriving messages.

Since our DAR system is basically *nondeterministic* (See that a Local DAR Executor could do different things after receiving an $ASSESS$ message and before receiving another $ASSESS$ message), distributed fault tolerance techniques for deterministic systems such as message logging cannot be directly used [7, 30, 2]. Our approach, which can be viewed as a *pessimistic* message logging approach tailored based on the semantics of DAR processes, is simple, does not need global coordination, and can tolerate nondeterministic events.

4 Additional Considerations

4.1 Security of Damage Assessment and Repair Subsystems

Intrusion tolerance components should certainly be secure and attack resilient by themselves, because otherwise the attacker can attack a system through attacking its intrusion tolerance components. As a critical component for distributed database intrusion tolerance, damage assessment and repair subsystems must be secure and attack resilient. Our damage assessment and repair subsystem, mainly composed of (only) two types of components, namely Local DAR Managers and Local DAR Executors, can be easily secured. First, we can use standard secure communication techniques such as IPSEC to ensure the messages between a Local DAR Manager and a Local DAR Executor or between a Local DAR Executor and a cleaning transaction coordinator will not be changed, dropped, faked, or replayed. Second, we can use standard integrity protection techniques such as message digests and digital signatures to ensure that the data (structures) used by the subsystem will not be corrupted. It should be noticed that here we address only external security threats and assume every DAR component works according to its specification and does not take malicious actions. Addressing internal security threats by malicious DAR components is out of the scope of this paper.

To make a damage assessment and repair subsystem attack (and failure) resilient is a more challenging. For this purpose, we use the design principle that the algorithm should be distributed in such a way that (a) there is no single point of failure; and (b) the dependency of one site upon other sites should be minimized. In particular, our algorithm avoids single points of failures by choosing to have no single point of control or coordination; our algorithm reduces the dependency of one site upon other sites by tolerating site and communication failures. As a result, when a (small) portion of sites are attacked to crash, the other sites can continue doing effective damage assessment and repair. The only impact of a crashed site A on another site B is that some damaged items at B may not be able to be identified or repaired. In most cases, especially when only a small portion of the sites are attacked to crash, most damaged items at healthy sites can still be correctly identified and repaired.

4.2 Performance Issues

Providing intrusion tolerance usually will affect the system performance. Therefore, a tradeoff between performance and intrusion tolerance should be (and has to be) made in most intrusion tolerant systems. Fortunately, our algorithm typically has very little impact on the performance of distributed transaction processing. The DAR components are totally separated from transaction processing components. In most cases when only few transactions are malicious the number of transactions affected by a malicious transaction is very small compared with the total number of transactions. We have done some performance study in centralized intrusion tolerant database systems using simulated data [18], the results show that on average only 2.72% transactions are affected by a malicious transaction. Although executing cleaning transactions makes the system busier, the number of cleaning transactions is typically very small compared with the total number of user transactions, and the Damage Confinement Manager can help prevent cleaning transactions from conflicting with user transactions. Our performance study in centralized intrusion tolerant database systems shows that on average only 0.32% items are damaged [18]. Although the DAR messages make the network busier, the number of DAR messages is typically very small compared with the number of transaction processing messages.

4.3 Other Issues

First, the proposed algorithm can be easily built upon a commercial DDBMS. In [18], we have implemented an on-the-fly damage assessment and repair system, denoted *ODAM*, on top of centralized Oracle servers. ODAM, which uses an algorithm similar to DAR Executors, can be extended to distributed commercial DBMSs. Second, it should be noticed that although repairing a damaged database can restore the data consistency, the proposed algorithm cannot (directly) recover applications from physical losses. For example, when a bad guy uses a good guy's credit card number to buy something, the amount of money is gone. Although the algorithm can make the credit card transaction database look like the fraud never happen, somebody, usually the credit card company, has to suffer the money loss if the bad guy cannot pay back the money (after being arrested). This kind of loss is usually managed by a separate internal database. Third, it should be noticed that in most cases the proposed algorithm can only be a part of real world attack recovery systems, which may involve many additional activities. In the credit card fraud example, if the repair happens after the good guy gets the monthly statement, then the credit card company, after reparing the database, may need to send a letter to inform the guy that the fraud is resolved.

5 Conclusion

We have analyzed the problem of damage assessment and repair in secure and resilient distributed database systems. Our algorithm, to our best knowledge the first method for distributed damage assessment and repair, has a number of unique and novel properties: (1) It is totally distributed, thus it has no single point of failures and it can tolerate malicious attacks on end systems and communication links. (2) It is on-the-fly. New transactions are continuously executed during the damage assessment and repair process. (3) It is efficient. Damage assessment and repair are performed, in parallel, at each participating site, and the sites are cooperating with each other using messages and message-driven mechanisms. The total time it uses to finish a distributed damage assessment and repair is (in most situations) much less than the summary of the time used by each site. (4) It works seamlessly with a commercial DDBMS, thus it can be directly applied to build resilient distributed database applications. (5) It provides transparent damage assessment and repair. It can mask database damage to the human user. We believe we have accomplished an important step towards providing "data integrity" guarantees to arbitrary distributed database applications in face of attacks.

References

[1] M. R. Adam. Security-Control Methods for Statistical Database: A Comparative Study. *ACM Computing Surveys*, 21(4), 1989.

[2] L. Alvisi and K. Marzullo. Message logging: Pessimistic, optimistic, causal and optimal. *IEEE Transactions on Software Engineering*, 24(2):149–159, 1998.

[3] P. Ammann, S. Jajodia, and P. Liu. Recovery from malicious transactions. Technical report, George Mason University, Fairfax, VA. Submitted for Journal publication.

[4] V. Atluri, S. Jajodia, and B. George. *Multilevel Secure Transaction Processing*. Kluwer Academic Publishers, 1999.

[5] D. Barbara, R. Goel, and S. Jajodia. Using checksums to detect data corruption. In *Proceedings of the 2000 International Conference on Extending Data Base Technology*, Mar 2000.

[6] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA, 1987.

[7] A. Borg, J. Baumbach, and S. Glazer. A message system supporting fault tolerance. *Operating System Review*, 17(5):90–99, October 1983.

[8] D. E. Denning. An intrusion-detection model. *IEEE Trans. on Software Engineering*, SE-13:222–232, February 1987.

[9] T.D. Garvey and T.F. Lunt. Model-based intrusion detection. In *Proceedings of the 14th National Computer Security Conference*, Baltimore, MD, October 1991.

[10] P. P. Griffiths and B. W. Wade. An Authorization Mechanism for a Relational Database System. *ACM Transactions on Database Systems*, 1(3):242–255, September 1976.

[11] K. Ilgun. Ustat: A real-time intrusion detection system for unix. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 1993.

[12] K. Ilgun, R.A. Kemmerer, and P.A. Porras. State transition analysis: A rule-based intrusion detection approach. *IEEE Transactions on Software Engineering*, 21(3):181–199, 1995.

[13] S. Jajodia, P. Samarati, V. S. Subrahmanian, and E. Bertino. A unified framework for enforcing multiple access control policies. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 474–485, May 1997.

[14] H. S. Javitz and A. Valdes. The sri ides statistical anomaly detector. In *Proceedings IEEE Computer Society Symposium on Security and Privacy*, Oakland, CA, May 1991.

[15] H. S. Javitz and A. Valdes. The nides statistical component description and justification. Technical Report A010, SRI International, March 1994.

[16] H.F. Korth, E. Levy, and A. Silberschatz. A formal approach to recovery by compensating transactions. In *Proceedings of the International Conference on Very Large Databases*, pages 95–106, Brisbane, Australia, 1990.

[17] P. Liu, P. Ammann, and S. Jajodia. Rewriting histories: Recovery from malicious transactions. *Distributed and Parallel Databases*, 8(1):7–40, 2000.

[18] P. Luenam and P. Liu. Odam: An on-the-fly damage assessment and repair system for commercial database applications. In *Proc. 15th IFIP WG11.3 Working Conference on Database and Application Security*, 2001. To appear.

[19] Teresa Lunt and Catherine McCollum. Intrusion detection and response research at DARPA. Technical report, The MITRE Corporation, McLean, VA, 1998.

[20] T.F. Lunt. A Survey of Intrusion Detection Techniques. *Computers & Security*, 12(4):405–418, June 1993.

[21] J. McDermott and D. Goldschlag. Towards a model of storage jamming. In *Proceedings of the IEEE Computer Security Foundations Workshop*, pages 176–185, Kenmare, Ireland, June 1996.

[22] C. Mohan, B. Lindsay, and R. Obermarck. Transaction management in the r* distributed database management system. *ACM Transaction on Database Systems*, 11(4):378–396, 1986.

[23] B. Mukherjee, L. T. Heberlein, and K.N. Levitt. Network intrusion detection. *IEEE Network*, pages 26–41, June 1994.

[24] B. Panda and J. Giordano. Reconstructing the database after electronic attacks. In *Proceedings of the 12th IFIP 11.3 Working Conference on Database Security*, Greece, Italy, July 1998.

[25] P.A. Porras and R.A. Kemmerer. Penetration state transition analysis: A rule-based intrusion detection approach. In *Proceedings of the 8th Annual Computer Security Applications Conference*, San Antonio, Texas, December 1992.

[26] F. Rabitti, E. Bertino, W. Kim, and D. Woelk. A model of authorization for next-generation database systems. *ACM Transactions on Database Systems*, 16(1):88–131, 1994.

[27] D. Samfat and R. Molva. Idamn: An intrusion detection architecture for mibile networks. *IEEE Journal of Selected Areas in Communications*, 15(7):1373–1380, 1997.

[28] R. Sandhu and F. Chen. The multilevel relational (mlr) data model. *ACM Transactions on Information and Systems Security*, 1(1), 1998.

[29] S.-P. Shieh and V.D. Gligor. On a pattern-oriented model for intrusion detection. *IEEE Transactions on Knowledge and Data Engineering*, 9(4):661–667, 1997.

[30] R. E. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transaction on Computer System*, 3(3):204–226, August 1985.

[31] M. Winslett, K. Smith, and X. Qian. Formal query languages for secure relational databases. *ACM Transactions on Database Systems*, 19(4):626–662, 1994.