A Practical Transaction Model and Untrusted Transaction Manager for a Multilevel-Secure Database System

Myong H. Kang[a], Oliver Costich[b]†, and Judith N. Froscher[a]

[a] Center for Secure Information Technology, Naval Research Laboratory, Washington, D.C. 20375

[b] Center for Secure Information Systems, George Mason University, Fairfax, Virginia 22030

Abstract

A new transaction model for multilevel-secure databases which use the replicated architecture is presented. A basic concurrency control algorithm and two variations are given based on this transaction model. We also present new correctness criteria for multilevel-secure databases which use the replicated architecture. Based on this criteria, we prove that our algorithms are correct.

Keyword Codes: H.2.1; K.6.5
Keywords: Database Management, Logical Design; Security and Protection

1. INTRODUCTION

There are several approaches for multilevel database systems which protect classified information from unauthorized users based on the classification of the data and the clearances of the users. One, the integrity lock approach [5], attempts to combine encryption techniques with off-the-shelf database management systems. The trusted frontend applies an encrypted check sum to data in untrusted backend databases. Another, the kernelized approach [11], relies on decomposing the multilevel database into single level databases which are stored separately, under the control of a security kernel enforcing a mandatory access control policy.

---

† Supported by the Naval Research Laboratory under contract N0001489-C-2389.

The integrity lock approach is computationally intensive and has a potential covert channel. The kernelized approach can yield reduced performance due to the need for recombining single level data to produce multilevel data. Motivated by performance concerns, a replicated architecture approach has been proposed.

The replicated architecture approach [6] uses a physically distinct backend database management system for each security level. Each backend database contains information at a given security level and all data from lower security levels. The system security is assured by a trusted frontend which permits a user to access only the backend database system which matches his/her security level.

The SINTRA[1] database system, which is currently being prototyped at the Naval Research Laboratory, is a multilevel trusted database management system based on this replicated architecture. The replicated architecture system contains a separate database system for each security level. The database at each security level contains data at its own security level, and replicated data from lower security levels.

The SINTRA database system consists of one trusted front end (TFE) and several untrusted backend database systems (UBD). The role of the TFE includes user authentication, directing user queries to the backend, maintaining data consistency among backends, etc. Each UBD can be any commercial off-the-shelf database system. Figure 1 illustrates the SINTRA architecture.

In the SINTRA project, we make the following assumptions:

(1) All UBD use the same database query language (e.g., SQL).

(2) The TFE changes the database states of the UBD only through database queries.

(3) Each UBD performs some type of scheduling which produces a serializable and recoverable history.

1.1. Merits and Problems for the Replicated Architecture

At first glance, a database mangement system for each security level may seem excessive. However, we think this approach has the following merits:

- The security policy can be easily enforced by carefully designing interfaces among different database systems.
- Development cost can be reduced because commercial database systems for backend computers are widely available.
- The amount of trusted software can be minimized.
- Performance can be improved by using optimization and parallelization techniques which have been developed for conventional databases. This is the case because the replicated architecture uses conventional database systems as backend database systems, and uniprocessor or multiprocessor computers can be chosen as backend computers without affecting the security policy.

Despite the above advantages, the replicated architecture has a unique problem. Since each UBD in a replicated architecture contains data from lower levels, update transactions have

---

1. Secure INformation Through Replicated Architecture

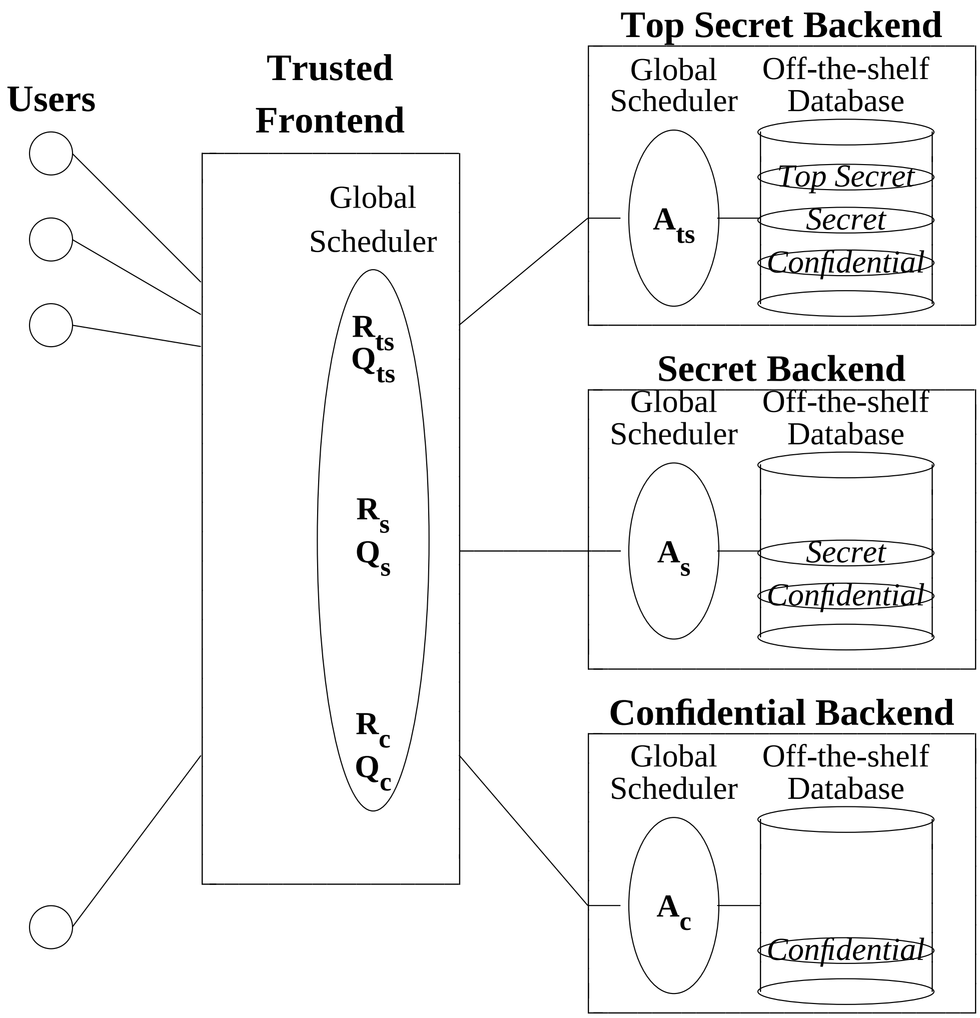


Figure 1: The SINTRA Architecture.

to be propagated up to higher security level databases. There are some problems which are related to this propagation.

[a] Since lower level update transactions propagate to higher level databases, high-level databases can be overloaded with lower level update transactions. This creates problems such as slow response to the request of a high-level user and longer propagation time for lower level update transactions.

[b] The propagation of update transactions has to be carefully controlled. If the propagation of update transactions is not carefully controlled, inconsistent database states among backend databases can be created. Consider this example. Two confidential level update transactions T_i and T_j are serialized in the order of $<T_i, T_j>$ at the confidential level backend database system. Since these two transactions are update transactions, these transactions have to be propagated to the secret level. If these two transactions are serialized in the order of $<T_j, T_i>$ at the secret level, an inconsistent database state between confidential and secret level backend databases may be created. Therefore, the serialization order introduced by the local scheduler at the user's session

level must be maintained at the higher level UBDs.

A possible solution to problem [a] is presented in [10]. In this paper we concentrate on solutions to problem [b].

1.2. Motivation for Another Concurrency Algorithm

Several concurrency control mechanisms which preserve database consistency and security for the replicated architecture have been proposed [4, 8, 12]. Those concurrency control algorithms assume that each UBD uses conservative scheduling or something similar to preserve the scheduled order of conflicting updates (i.e., never abort update transactions from lower security classes). In reality, off-the-shelf database systems do not generally guarantee this condition. Also such scheduling may either pass the burden to the user by asking him to predeclare read and write sets or remove the interactive query capability of database system. Hence, this assumption poses performance and usability problems for the SINTRA project.

Also the proposed algorithms use the conventional basic operations, read and write, to describe transactions. Traditionally, *r[x]* and *w[y]* are used to denote ‘‘read data item *x*’’ and ‘‘write data item *y*,’’ respectively. Data items *x* and *y* may be relations, fixed-sized pages, or tuples depending on the granularity of concurrency controllers. In this paper, we propose a new transaction model which is better suited for the SINTRA architecture.

The scheduler for the SINTRA architecture has two kinds of components; global and local. The global scheduler enforces data consistency among the UBDs. On the other hand, the local scheduler enforces serializability among transactions which are submitted to the backend database system. In theory, where the global scheduler executes is not important in this architecture. However, since we expect that I/O will be a bottle-neck for this type of architecture, we distribute much of the single level part of the global scheduler to the backend computers, depicted in figure 1. The local schedulers are the concurrency controllers of the off-the-shelf database systems.

The local schedulers typically use locks or timestamps based on the knowledge of actual data or physical layout of the data in each UBD. However, the global scheduler has very little knowledge about the behavior of the local scheduler or the physical layout of data. For example, the global concurrency controller has no knowledge about where a specific tuple is located or which physical page should be locked. Sometimes the tuples which will be modified are unknown until the computation based on existing data is completed. The above factors may force the global concurrency controller to use relations as basic units to detect conflicts among transactions. Such a scheduler will be too restrictive and inefficient because it ignores the fact that referencing only a few tuples or few attributes of a relation is not the same as referencing the entire relation.

Based on the observations above, we argue that the traditional transaction model is not sufficient to model transactions for this replicated architecture. In this paper, we introduce a layered view of transactions. In our model, a transaction can be viewed as a sequence of database queries, and each query can be viewed as a sequence of read and write operations. Based on this model, we introduce a concurrency control algorithm which makes no assumption that each UBD uses any particular scheduling technique.

This paper is organized as follows. Section 2 discusses a transaction model for the SINTRA architecture. A concurrency control mechanism based on this transaction model is presented in section 3. Finally, section 4 summarizes the contributions of this paper.

2. THE MODEL

In this section, models are presented for security, replicated architecture, and transaction processing. The transaction model, which is presented in this section, can alleviate the difficulties described above in section 1.2.

2.1. Security Model

The security model used here is based on that of Bell and LaPadula [1]. The database system consists of a finite set **D** of *objects* (data item) and a set **T** of *subjects* (transactions). There is a lattice **S** of security classes with ordering relation <. A class S_i *dominates* a class S_j if $S_i \geq S_j$. There is a *labeling function* **L** which maps objects and subjects to a security class:

$\mathbf{L}: \mathbf{D} \cup \mathbf{T} \rightarrow \mathbf{S}$

Security class **u** *covers* **v** in a lattice if **u** > **v** and there is no security class **w** for which **u** > **w** > **v**.

We consider two mandatory access control requirements:

(Simple Security Property)

If transaction T_i reads data item x then $L(T_i) \geq L(x)$.

(Restricted *-Property)

If transaction T_j writes data item x then $L(T_j) = L(x)$.

The simple security property allows a transaction to read data items if the security level of a transaction dominates the security level of data items. The restricted *-property allows a transaction to write if the security level of a transaction is the same as that of data items (i.e., no write-ups or write-downs are permitted). In [8], it is argued that write-ups (i.e., T_i cannot write to data item x if $L(T_i) < L(x)$) are undesirable in database systems for integrity reasons.

2.2. Replicated Architecture Model

The system has a TFE, which mediates the access of subjects to objects. The TFE contains the trusted computing base (TCB), but not all of the TFE need to be trusted. The system also contains a set of single level untrusted backend databases **C**, one for each element of the security lattice. Each backend database $\mathbf{C_u}$ contains copies of all data items in all databases whose security level is dominated by security level **u**. Alternatively, if $\mathbf{L(x)} = \mathbf{u}$ such that $\mathbf{x} \in \mathbf{C_u}$, then there is a copy of **x** in each database whose security level dominates **u**.

2.3. Transaction Model

We adopt a layered model of transactions, where a transaction is a sequence of queries, and each query can be considered as a sequence of reads and writes. For example, `replace` and `delete` queries can be viewed as a read operation followed by a write operation, `insert` can be viewed as a write operation, and `retrieve` can be viewed as a read operation. A layered view of two transactions T_1 and T_2 is shown in figure 2.

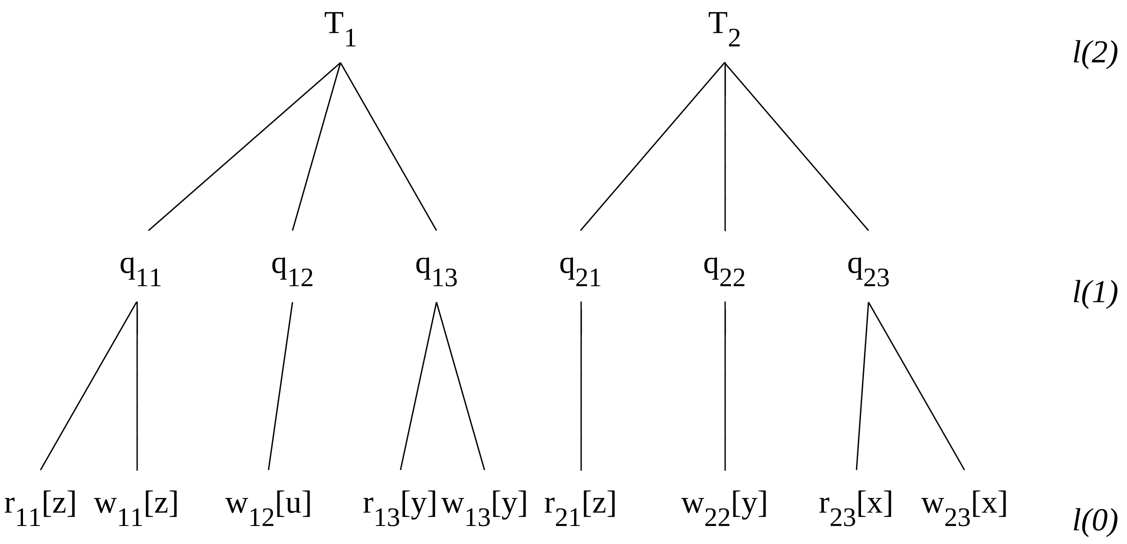


Figure 2: Layered model of two transactions.

Definition 1.

A **transaction** T_i is a sequence of queries terminated by either a *commit*(c_i) or an *abort*(a_i), i.e., $T_i = <q_{i1}, q_{i2}, ..., q_{in}, c_i>$. Each query, q_{ij}, is an atomic operation and is one of `retrieve`, `insert`, `replace`, or `delete`.

To model the propagation of updates produced by a given transaction to higher security level databases, *update projection* is defined.

Definition 2.

An **update projection** U_i, which corresponds to a transaction T_i, is a sequence of update queries, e.g., $U_i = <q_{i2}, q_{i5}, ..., q_{in}, c_i>$ obtained from transaction T_i by simply removing all `retrieve` queries.

Note that no aborted transaction will be propagated. Hence, update projections are always terminated by a commit. Also note that our update projections consist of read and write operations.

To describe concurrency control mechanisms, we adopt the following definition of *conflict*.

Definition 3.

Two operations at the same layer **conflict** if they operate on the same data item and one of them is either *write, insert, delete,* or, *replace*.

It is interesting to compare our transaction model to another multilevel transaction model which appears in [13]. In their model, the low-level conflicts impose constraints on the serialization orders for higher levels. However, in the SINTRA architecture, the global scheduler does not have enough information about the conflicts which may occur at the local scheduler. Therefore, the global scheduler has to make serialization decisions independent of the those of the local scheduler.

In the following section, we present a concurrency control algorithm using the transaction model above. In our concurrency control algorithm, the global scheduler works at the query level (i.e., *l(1)* in figure 2).

3. A CONCURRENCY CONTROL MECHANISM

In this section, a concurrency control algorithm is presented, which makes no assumption about UBD scheduling. In this algorithm, two types of schedulers can be identified, global and local schedulers. The global scheduler enforces data consistency among different security levels. On the other hand, the local scheduler enforces serializability among transactions, including update projections, which are submitted to the backend database system. The local scheduler deals with layer *l(0)* in figure 2, and the global scheduler deals with layer *l(1)* and upper layers. The global scheduler detects conflicts at level *l(1)*. Therefore, no knowledge of the specific items to be accessed or even the granularity of the lower level concurrency controller is required.

3.1. Algorithm C

To describe the concurrency control protocol, we need to define several mechanisms:

- A queue $\mathbf{Q_u}$ is associated with each backend database $\mathbf{C_u}$, where **u** is a security level. The purpose of $\mathbf{Q_u}$ is to maintain a list of update projections which have been executed and committed at $\mathbf{C_u}$. The queue is ordered by the serialization order of the execution of these transactions at $\mathbf{C_u}$.
- In addition, there is an untrusted mechanism $\mathbf{R_u}$ which maintains $\mathbf{Q_u}$ and can read the contents of $\mathbf{Q_v}$ for all **v** which are dominated by **u** in the security lattice.
- Another queue $\mathbf{A_u}$ is associated with each backend database $\mathbf{C_u}$. The purpose of $\mathbf{A_u}$ is to maintain a list of update projections which come from $\mathbf{Q_v}$, where **v** is covered by **u**, and are waiting to be sent to $\mathbf{C_u}$. The order of update projections in $\mathbf{A_u}$ is determined by the concurrency control algorithm which will be described later.

In our algorithm, $\mathbf{Q_u}$, $\mathbf{A_u}$, and $\mathbf{R_u}$ are considered parts of a global scheduler. Since mechanism $\mathbf{R_u}$ has to read the contents of $\mathbf{Q_v}$ for all **v** which are dominated by **u**, the $\mathbf{R_u}$ and the $\mathbf{Q_u}$ may be located in the TFE. However, $\mathbf{A_u}$ may be located in the backend system (see figure 1). Also in our algorithm, we say a backend database $\mathbf{C_u}$ *covers* $\mathbf{C_v}$ if **u** covers **v** in the security lattice. The protocol processes transactions as follows:

Algorithm C:

At each backend database $\mathbf{C_u}$:

[1] Primary transactions (that are submitted directly by the user) and update projections are received from the global scheduler and submitted to the local backend scheduler.

[2] As local transactions (primary transactions and update projections) are committed, a report of their serialization is sent to the global scheduler. These reports are sent in an order consistent with the serialization order determined by the local scheduler.

At the global scheduler:

[1] For each primary transaction $\mathbf{T_i}$ submitted to the TFE, $\mathbf{T_i}$ is dispatched to $\mathbf{C_u}$ for processing where $\mathbf{L(T_i) = u}$.

[2] Whenever a serialization report for $\mathbf{T_i}$ or $\mathbf{U_j}$ is received from $\mathbf{C_u}$, it is added to the end of $\mathbf{Q_u}$.

[3] The $\mathbf{R_u}$ scans the queue $\mathbf{Q_v}$ for those **v** for which $\mathbf{C_u}$ covers $\mathbf{C_v}$. The $\mathbf{R_u}$ will retrieve an update projection $\mathbf{U_i}$ from $\mathbf{Q_v}$ and add it to the end of $\mathbf{A_u}$ when the following condition is satisfied for all $\mathbf{v} \in \mathbf{S}$:

- If $\mathbf{C_u}$ covers $\mathbf{C_v}$, and $\mathbf{U_j}$ can eventually appear in $\mathbf{Q_v}$, then it does appear in the beginning of $\mathbf{Q_v}$.

[4] For update projections in the queue $\mathbf{A_u}$, update projections are sent to $\mathbf{C_u}$ one after another. Specifically, if $\mathbf{U_i}$ is before $\mathbf{U_j}$ in the queue $\mathbf{A_u}$, then send $\mathbf{U_i}$ and wait until $\mathbf{U_i}$ is committed at $\mathbf{C_u}$, and then send $\mathbf{U_j}$.

[5] An update projection is removed from $\mathbf{A_u}$ once it is committed.

[6] If an update projection, $\mathbf{U_i}$, is aborted then resubmit $\mathbf{U_i}$ to $\mathbf{C_u}$.

In algorithm C, we assume that local schedulers produce schedules such that the serialization order and the commit order of transactions are the same[2] (i.e., for any pair of transactions T_i and T_j, if T_i is committed before T_j then T_i also precedes T_j in the serialization order). However, most database schedulers do not guarantee the above condition. The *take-a-ticket* [7] operation can be used to force any *recoverable* scheduler [2] to produce schedules such that the serialization order and the commit order of transactions are the same. The *take-a-ticket* operation consists of reading the value of a ticket prior to commit time, and incrementing it through regular data manipulation operations. The value of a ticket determines the serialization order. All operations on tickets are subject to the local concurrency control.

Note that algorithm C does not slow down user (primary) transactions. The global scheduler of algorithm C concerns the serialization order of the update projections in $\mathbf{A_u}$ at each security level. Concurrency control among primary transactions and update projections is the responsibility of the local scheduler in the UBD.

Also note that $\mathbf{Q_u}$ and $\mathbf{R_u}$ are not needed if the security classes form a completely ordered set, since $\mathbf{A_u}$ satisfies all the requirement of the algorithm.

3.2. Proof of Correctness

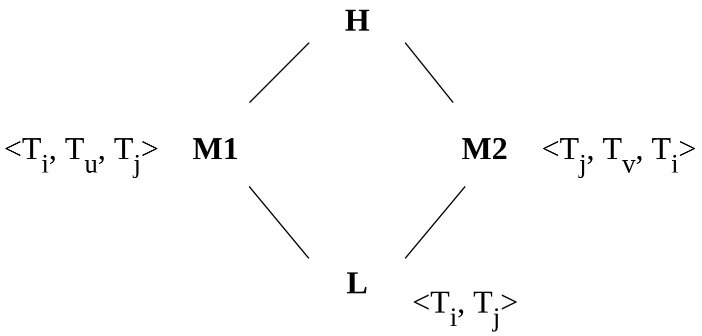
Many concurrency control algorithms have been proposed for the replicated architecture [4, 8, 12]. These papers use *one-copy-serializability* (or *1SR*) [2] as the correctness criteria for their concurrency control algorithms. In this paper, we present an alternative correctness criteria which we consider more intuitive. Based on this criteria, we will prove that algorithm C is correct. We will then show that our criteria imply one-copy-serializability.

For the sake of mathematical convenience, in this section, we assume read-only transactions have empty update projections which serve solely to mark their position in the serialization ordering of all transactions. Also, in this section, we do not distinguish the queue $\mathbf{Q_u}$ at the security class **u** and the contents of update projections in the queue $\mathbf{Q_u}$.

An example will help to clarify our approach. Consider the security lattice in figure 3, and two non-conflicting **L**-level transactions T_i and T_j. Also consider an **M1**-level transaction T_u, and an **M2**-level transaction T_v. Let's further assume that T_u conflicts with T_i and T_j, and T_v conflicts with T_i and T_j. Since two transactions, T_i and T_j, are not conflicting and

2. Consider the history of two transactions, T_i and T_j, **H**, where $\mathbf{H} = r_i[x]\ w_i[x]\ r_j[x]\ w_j[x]\ w_i[y]\ c_i\ c_j$. This history does not satisfy the rigorousness condition [3]. However, this history will be satisfactory for our purpose because the serialization order and the commit order of transactions are the same.

our security model does not allow *write-down*, an execution order $<T_i, T_u, T_j>$ at security class **M1** and an execution order $<T_j, T_v, T_i>$ at security class **M2** will generate the same result on *replicas* of security class **L** data. However, the reversed order between T_i and T_j at security classes **M1** and **M2** will create confusion in applying our update projection propagation rule. Specifically, at security class **H**, a consistent ordering among T_i, T_j, T_u, and T_v cannot be determined then *1SR* will be violated. Consequently, any global scheduler which does not enforce the same ordering among transactions at each relevant UBD may fail to produce consistent schedules. Thus any algorithm which gives 1SR schedules must preserve the orderings at lower levels.



H > **M1** > **L**
H > **M2** > **L**

Figure 3: A security lattice

In this paper, we use another criteria which says ‘‘preserve the order between update projections which is determined at lower security class (or preserve the relative order).’’ We also show that any schedule for this architecture which preserves the relative order of update projections satisfies *1SR*.

Let $\mathbf{U} = \{ U_i \mid T_i \in \mathbf{T}\}$ be the set of all update projections and $\mathbf{U}^*$ be the set of all strings from **U**. Then for each pair of security classes **u** and **v** of **S**, for which $\mathbf{u} \geq \mathbf{v}$, there is a projection $\pi_{uv} : \mathbf{U}^* \rightarrow \mathbf{U}^*$ which is defined as follows:

(a) $\pi_{uv}(U_i) = U_i$ if $L(T_i) \leq \mathbf{v}$.

(b) $\pi_{uv}(U_i) = \lambda$, the null string, otherwise.

Definition 4.

If **u** and **v** are in **S**, with $\mathbf{u} \geq \mathbf{v}$, and $\pi_{uv}(\mathbf{Q_u}) = \mathbf{Q_v}$ then we say that the **relative order** between $\mathbf{Q_u}$ and $\mathbf{Q_v}$ is preserved.

For example, if $\mathbf{Q_u} = <U_i, U_k, U_j>$ and $\mathbf{Q_v} = <U_i, U_j>$, and U_k is originated from security class **w**, where $\mathbf{u} \geq \mathbf{w} > \mathbf{v}$, then the relative order of update projections in $\mathbf{Q_u}$ and $\mathbf{Q_v}$ is preserved because $\pi_{uv}(\mathbf{Q_u}) = \mathbf{Q_v} = <U_i, U_j>$. We can now state our concept of correctness for transaction processing more precisely.

Definition 5.

For replicated architecture trusted database systems as above, let $\mathbf{Q_u}$ be the serialization order of the transactions and update projections committed at $\mathbf{C_u}$. Then a concurrency control algorithm is **correct** if it preserves relative orders for all elements of the security lattice.

Theorem 1

Algorithm C is correct.

Proof.

This is evident from the algorithm, since at each class, the update projections are executed and committed in the same relative order established at lower security class. □

Briefly, a schedule of transaction execution on a replicated database system is *one-copy-serializable* if it is *view equivalent* to a serial schedule on a one-copy database systems. View equivalence requires the two schedules to have the same reads-from and final-writes relationships. Details for this architecture may be found in [4]. Such schedules will be referred to as **ML-1SR**. Although in the following proofs, we use only read and write operations, we could instead use `retrieve, insert, replace`, and `delete` as in our query languages. We denote read and write operations on a data item x, or a copy of it x_n, of transaction T_i, by $r_i[x]$ and $w_i[x]$.

Theorem 2

With the architecture as specified above, if each UBD has a local scheduler which produces serializable schedules and there is a global scheduler such that for all **u** and **v** with $\mathbf{u} \geq \mathbf{v}$, the relative order between $\mathbf{Q_u}$ and $\mathbf{Q_v}$ is preserved, then the global schedule is **ML-1SR**.

Proof.

Let $\mathbf{S_m}$, where **m** is the maximal element of the security lattice, determine a serial execution order on the one-copy logical database corresponding to the replicated system, and let **H** be the schedule produced by an algorithm satisfying the hypotheses of the theorem. We assert that **H** is view equivalent to $\mathbf{S_m}$. Clearly, **H** and $\mathbf{S_m}$ have the same final-writes, by definition of $\mathbf{S_m}$. We will show that **H** and $\mathbf{S_m}$ have the same reads-from relationships using proof by contradiction.

(1) Suppose T_j reads-x-from T_i in $\mathbf{S_m}$, but not in **H**. Let $\mathbf{L}(T_j) = \mathbf{n}$. Then in $\mathbf{C_u}$, $w_i[x]$ precedes $w_k[x]$ and $w_k[x]$ precedes $r_j[x]$. But then at $\mathbf{Q_u}$, the serialization order is T_i precedes T_k and T_k precedes T_j. This is preserved in $\mathbf{S_m}$, by hypothesis, contradicting that T_j reads-x-from T_i in $\mathbf{S_m}$.

(2) Suppose T_j reads-x-from T_i in **H**, but not in $\mathbf{S_m}$. Then if $\mathbf{L}(T_j) = \mathbf{n}$, T_j reads-x-from T_i in $\mathbf{C_u}$. But if there is a T_k for which $w_i[x]$ precedes $w_k[x]$ and $w_k[x]$ precedes $r_j[x]$ in $\mathbf{S_m}$, then because $\mathbf{L}(T_k) = \mathbf{L}(T_i) = \mathbf{L}(x) \leq \mathbf{n}$, this relationship also holds in $\mathbf{C_u}$ and thus in **H**. This contradicts our assumption that T_j reads-x-from T_i in **H**. □

It is worthwhile to note that if the system has a completely ordered security lattice, the relative order between non-conflicting update projections does not have to be preserved. It is sufficient to preserve the ordering of conflicting update projections, rather than all update projections. This may permit greater concurrency.

Corollary 1

Algorithm C produces **ML-1SR** schedules.

Proof.

This follows immediately from the preceding theorems. □

3.3. Two Variations

Step [4] of algorithm C forces update projections to execute serially. However, if the global scheduler of the SINTRA system can detect conflicts among transactions, we can achieve better concurrency among update projections by taking advantage of this knowledge. Since the backend database system usually cannot report whether there were conflicts, an accurate analysis technique which can detect conflicts among transactions is needed. Data dependence analysis, introduced in [9], can detect conflicts among transactions without any knowledge of actual data or physical layout of data. Rather, it relies on analysis of the queries themselves, detecting conflicts by determining if common data is to be accessed.

The rest of the section introduces two variations of the algorithm C, *optimistic* and *semi-optimistic* approaches, which may achieve better concurrency. These variations are concerned with how update projections in $\mathbf{A_u}$ are submitted to the UBD, and therefore how committed user transactions and update projections are placed in $\mathbf{Q_u}$. In our two variations, transactions are executed hoping that the ‘‘correct’’ schedule is produced. When it is not, some amount of work already done will have to be undone. Hence, processing that is completed will have to be *certified* before it can be committed.

For the rest of the section, we assume that the serialization order is known at the end of each transaction (just before it might be committed). No transaction may actually be committed unless the global scheduler certifies it. If a user transaction or an update projection is committed, then it will be dispatched to $\mathbf{Q_u}$. However, if a user transaction or an update projection is executed but not yet committed, the global scheduler will store it in a queue $\mathbf{P_u}$. $\mathbf{P_u}$ holds candidates for insertion into $\mathbf{Q_u}$. Once the global scheduler certifies and commits a transaction then it will be moved from $\mathbf{P_u}$ to $\mathbf{Q_u}$.

3.3.1. Optimistic Approach

Generally, the optimistic variation of the algorithm C works as follows. User transactions and update projections at any security level are submitted to the backend as they arrive. If an update projection is completed out of the submission order, it is held in $\mathbf{P_u}$ awaiting certification, along with the completed user transactions submitted at that level. When an update projection which is submitted earlier completes and is placed in $\mathbf{P_u}$, data dependence analysis is used to determine whether the serialization order determined by $\mathbf{P_u}$ is equivalent (in a technical sense) to a correct (update order-preserving) one. If it is, $\mathbf{P_u}$ is rearranged to get a maximal prefix which is correct. The transactions in the prefix are certified and committed. If $\mathbf{P_u}$ cannot be rearranged, all transactions in $\mathbf{P_u}$ must be rolled back and re-submitted.

More specifically, the optimistic approach works as follows:

[a] Submit update projections in $\mathbf{A_u}$ to UBD as they arrive.

[b] Commit user transactions and update projections as long as update projections are serialized in the order that they are submitted to UBD up to that time (i.e., $\mathbf{P_u}$ is empty). Once those are committed then dispatch them to $\mathbf{Q_u}$.

[c] If U_i should be next to be serialized but U_j is already serialized then put U_j in $\mathbf{P_u}$ without committing it. Any user transactions or update projections which are executed will be put in $\mathbf{P_u}$ without committing them until U_i is executed. After U_i is executed, put U_i into $\mathbf{P_u}$.

[d] While the first and last transactions in $\mathbf{P_u}$ are update projections, do the following steps:

(1) Find an update projection in $\mathbf{P_u}$ which should be serialized before any other uncommitted update projections. Call that update projection T_n.

(2) Test if the first transaction in $\mathbf{P_u}$, T_1 (which must be an update projection), conflicts with T_n. If T_1 and T_n conflict then abort and remove all transactions in $\mathbf{P_u}$, re-submit them, and return to [a].

(3) Test if T_1 conflicts with any of the transactions from T_2 through T_{n-1}. If there is no conflict, then move T_1 to immediately after T_n in $\mathbf{P_u}$ and go to step (5).

(4) Test if T_n conflicts with any of the transactions from T_2 through T_{n-1}. If there is a conflict, then abort and remove all transactions in $\mathbf{P_u}$, re-submit them, and return to [a]. Otherwise move T_n to the beginning of $\mathbf{P_u}$.

(5) While the first transaction in $\mathbf{P_u}$, T_1, is either a user transaction or an update projection which is in the proper order to be serialized then do the following steps:

- Commit T_1, remove it from $\mathbf{P_u}$ and $\mathbf{A_u}$ if applicable, and dispatch it to $\mathbf{Q_u}$.

An example of this may be helpful. Consider a situation where the global scheduler submits update projections in the order of $<U_i, U_j>$ and the serialization order at the UBD is $<U_j, T_k, U_i>$ where T_k is a user transaction. Since U_j is serialized before U_i, U_j cannot be committed until U_i is committed. Now the task is to find if the order of either U_i or U_j can be rearranged. The only way this can happen is:

- There is no conflict between U_i and U_j, and either
  (a) There is no conflict between U_j and T_k, or
  (b) There is no conflict between T_k and U_i.

If situation (a) happens then $<T_k, U_i, U_j>$ will be the order which will be sent to $\mathbf{Q_u}$ by the algorithm. If situation (b) happens then $<U_i, U_j, T_k>$ will be the order which will be sent to $\mathbf{Q_u}$. This more complex approach may be justified if conflicts are likely to occur more frequently.

We could improve the performance of this algorithm by minimizing the number of transactions which must be aborted. For example, assume that T_1 and T_n do not conflict but T_1 conflicts with T_k and T_n conflicts with T_{k+1}. We could just abort T_{k+1} through T_n to ensure that T_n is serialized before T_{k+1}.

3.3.2. Semi-optimistic Approach

Since the SINTRA security policy prohibits *write-up* or *write-down*, the probability of conflict between a user transaction at security class **u** and an update projection from the security class **v** where **u** > **v** may be quite small[3]. Hence if conflicts among update projections are detected before those are submitted then there is less probability of aborting.

The semi-optimistic variations is similar to the optimistic one, but rather than submitting update projections as they arrive, it checks for conflicts first, thereby reducing the likelihood of having to roll back work already done. Specifically, the semi-optimistic approach replace the step [a] of the preceding optimistic approach with following:

[a1] Detect conflicts among update projections in $\mathbf{A_u}$.

[a2] If two update projections U_i and U_j conflict then submit them serially (i.e., submit one and then wait for a commit message before submitting another).

[a3] If there are update projections in $\mathbf{A_u}$ which do not conflict then submit one after another (i.e., submit one and then submit next update projection without waiting for commit message of previous update projection).

After applying steps [b] and [c] of the optimistic approach, $\mathbf{P_u}$ can be tested, as before (i.e., step [d] of optimistic approach). The only difference is that there is no need to detect conflicts among update projections because those have been already tested. Therefore, only steps (1), (3), (4), and (5) from the optimistic approach are required. This technique clearly falls between those of the previous two algorithms. The following table clarifies the different approaches of the three variations.

---

3. The SINTRA security policy allows *read-down*. Hence, a user transaction at level **u** can conflict with an update projection from level **v** where **u** > **v**.

Algorithm Variant	Submission Process for Update Projections	Mechanism for Insuring Consistent Update Projection Ordering
Pessimistic (Algorithm C)	One at a time	None required
Semi-optimistic	Check for conflicts before submitting. Maintain submission order for conflicting update projections.	Check for conflicts between update projections and primary transactions after execution. Roll back and redo as necessary.
Optimistic	As they arrive (No checking or delaying)	Check for conflicts among all local transactions after execution. Roll back and redo as necessary.

3.3.3. Correctness of the Variations

Corollary 2

The Optimistic and semi-optimistic variations of algorithm C produce **ML-1SR** schedules.

Proof.

This is evident from the algorithm, since the global scheduler certifies a schedule only if the relative order between $\mathbf{Q_u}$ and $\mathbf{Q_v}$ is preserved for all **u** and **v** with $\mathbf{u} \geq \mathbf{v}$. □

4. CONCLUSIONS

In this paper, we have presented arguments that the traditional transaction model is not sufficient to model transactions for multilevel-secure databases which use the replicated architecture. We have proposed a new transaction model for multilevel-secure databases.

Even though several concurrency control algorithms for the replicated architecture have been proposed, those algorithms assume that each UBD uses conservative scheduling or at least assumes schedules preserve the order of update projections without indicating how it is done. We present a concurrency control algorithm which does not assume that each UBD uses conservative scheduling. Our concurrency control algorithm is based on the transaction processing model which is proposed in this paper, which controls ordering through other means, outside the UBD.

Our basic concurrency algorithm, algorithm C, executes update projections serially. We also offer two variations of algorithm C, optimistic and semi-optimistic approaches, which may achieve better concurrency under a variety of likely conditions. Our directions for future research are performance comparison among different variations under different application scenario. These algorithms are, in fact, being implemented in our prototype for the SINTRA project.

REFERENCES

1 Bell, D. E., and LaPadula, L. J. Secure computer systems: Unified exposition and multics interpretation. The Mitre Corp, (1976).

2 Bernstein, P. A., et el. Concurrency controller and recovery in database systems. Addison-Wesley (1987).

3 Breitbart, Y., et el. On rigorous transaction scheduling. IEEE Transaction on Software Engineering, 17, 6 (1991).

4 Costich, O. Transaction processing using an untrusted scheduler in a multilevel database with replicated architecture. in Database Security V: Status and Prospects (North-Holland 1992)

5 Denning, D. Commutative filters for reducing inference threats in multilevel database systems. Proceedings of the IEEE symposium on Security and Privacy (1985).

6 Froscher, J. N., and Meadows, C. Achieving a trusted database management systems using parallelism. in Database Security II: Status and Prospects (North-Holland 1989)

7 Georgakopoulos, D., et el. On serializability of multidatabase transactions through forced local conflicts. Proceedings of Conference on Data Engineering (1991).

8 Jajodia, S., and Kogan, B. Transaction processing in multilevel-secure databases using replicated architecture. Proceedings of the IEEE symposium on Security and Privacy (1990).

9 Kang, M. H., et el. Data dependence analysis for an untrusted transaction manager in a multilevel database system. First International Conference on Information and Knowledge management (1992).

10 Kang, M. H., and Froscher, J. N. Architectural impact on performance for a multilevel database system. Submitted for publication.

11 Lunt, T., et el. The seaview security model. IEEE Transaction on Software Engineering, 16, 6 (1990).

12 McDermott, J., et el. A single level scheduler for the replicated architecture for multilevel-secure databases. Proceedings of the seventh annual computer security applications conference (1991).

13 Weikum, G. Principles and realization strategies of multilevel transaction management. ACM Transactions on Database Systems, 16, 1 (1991)