Transaction Processing Using an Untrusted Scheduler in a Multilevel Database with Replicated Architecture*

Oliver Costich

Center for Secure Information Systems, George Mason University, 4400 University Drive, Fairfax, Virginia 22030, USA

Software Architecture and Engineering, 13100 Worldgate Drive, Herndon, Virginia 22070, USA

Abstract

Replicated architecture has been proposed as a way to obtain acceptable performance in a multilevel secure database system. This architecture contains a separate database for each security level such that each contains replicated data from lower security classes. The consistency of the values of replicated data items must be maintained without unnecessarily interfering with concurrency of database operations. This paper provides a protocol to do this that is secure, since it is free of covert channels, and also ensures one-copy serializability of executing transactions. The protocol can be implemented with untrusted processes for both concurrency and recovery.

1. INTRODUCTION

In recent history, significant energy has been expended in attempts to develop database systems that protect classified information from unauthorized users based on the classification of the data and the clearances of the users. These are generally referred to as multilevel database systems.

---

*This work was supported by the Naval Research Laboratory under Contract N0001489-C-2389.

Earlier efforts to develop this kind of database system utilized the integrity lock approach [4] or the kernelized DBMS approach [8,10]. The former approach relies on a trusted front-end process that applies cryptographic check values to data in an untrusted back-end database. The latter approach relies on decomposing the multilevel database into single level databases that are stored separately. The data must be recomposed to obtain multilevel data in response to queries.

The integrity lock approach is computationally intensive and has a potential covert channel [4]. There is also a risk that service could be denied by forcing checksums to fail continuously. The kernelized approach can yield reduced performance due to the need to recombine single level data to produce multilevel data. More recently, a replicated architecture approach has been proposed with the speculation that performance degradation due to security requirements would not be as severe as with the other approaches [5].

Basically, the replicated architecture approach uses a physically distinct back-end DBMS for each security level, with each one containing all the data at its security level and below. The system's security is assured by a trusted front-end, that permits a user access to only the DBMS that matches their security level. In this approach, some of the less desirable features of the other approaches are avoided but at a cost. The cost is that of maintaining the consistency of the data that is contained in more than one back-end database (almost all of it) without compromising security or losing the potential for concurrent execution of database operations. The problem, then, is to develop a protocol that does this "correctly", i.e., so the user's view is that of a single database running only the user's transactions in some sequential order.

A number of protocols have been developed for kernelized architecture multilevel database systems that can be adapted to the replicated architecture ones. A protocol using a trusted scheduling process is proposed in [7], and an untrusted one in [9]. Both of these utilize a timestamp ordering technique with a global "clock" that makes them potentially susceptible to a covert channel, depending on how the "clock" is implemented.

For the replicated architecture multilevel database, a transaction processing protocol is presented in [6]. This protocol uses a trusted queue manager in its scheduler. In addition, the protocol fails unless the security levels are linearly related. It is interesting that this protocol has been adapted to a kernelized architecture model [10]. In [6], it is also noted that although the architecture appears to be that of a distributed database, the standard techniques used for consistency control in such systems fail due to the security requirements imposed. For example, the classical two-phase commit protocol has a covert channel, the capacity of which is computed in [3]. Concurrent with the work presented here, another approach to this problem using fully replicated transactions has been explored in [12].

In this paper, a scheduling protocol for the replicated architecture will be defined that relies solely on untrusted processes, does not require a linear ordering of the security levels, and does not rely on timestamp ordering

methods. The criterion for "correctness" that is brought to bear is drawn from the standards used in the theory of replicated database systems, namely one-copy serializability.

The presentation is organized as a description of the model used, followed by the specification of the protocol, the proof of its correctness, remarks about garbage collection and recovery, the conclusion, and directions of future work.

2. THE MODEL

The security model used here is based on the framework established by Bell and LaPadula [1]. The notation for the security and the replicated architecture database model is adapted from that in Jajodia and Kogan [6]. Both are sketched in the following sections.

2.1 Security Paradigm

The database system (**DBS**) consists of a finite set **D** of *data items* that are objects of the trusted system, and a finite set **T** of *transaction*s, that act on behalf of users and are subjects of the trusted system. There is a lattice[**] **S** of *security classes*, (**S**,<). If security classes **u** and **v** are in **S** then **u** *dominates* **v** if $\mathbf{v} \leq \mathbf{u}$. There is also a *labeling function* **L** that assigns unique security classes to data items and transactions:

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

The notion of security here only encompasses mandatory access control requirements. Discretionary access control issues are not discussed. The mandatory access control requirements are:

(1) If transaction $\mathbf{T_j}$ reads data item **x** then $\mathbf{L(T_j) \geq L(x)}$.

(2) If transaction $\mathbf{T_j}$ writes data item **x** then $\mathbf{L(T_j) = L(x)}$.

Enforcement of these two conditions guarantees that information concerning high security level data items cannot flow to lower security level

---

[**]It is not necessary that the security classes form a lattice. A partially ordered set is sufficient, though the lattice case is easier to comprehend intuitively. In the description of the protocol, the necessary addition to protocol to extend it to this case is noted.

transactions (and users). The second condition is more restrictive than the traditional ★-property in that $\mathbf{T_i}$ cannot write data item **x** if $\mathbf{L(T_i) < L(x)}$, i.e., no write-ups are permitted. In [6], it is argued that write-ups are undesirable in trusted database systems for integrity reasons and may permit covert channels. In any case, the restriction on writing-up is imposed to bound the complexity of the protocol, and can be removed at the cost of an increase in the complexity of the model definition and the attendant protocol.

2.2 DBS Architecture

The replicated database architecture is obtained by adding a collection of single-level, untrusted database systems to the security structure, one for each security class. That is, by adding a set $\{\mathbf{C_v} \mid \mathbf{v} \in \mathbf{S}\}$ of *back-end databases*. The system also contains a *front-end* processor, the trusted front end (**TFE**), that mediates the access of subjects (transactions) to objects (data items). The **TFE** will contain the trusted computing base (TCB), but not all of the **TFE** need be trusted. In particular, much of the scheduling mechanisms for the protocol will be in the untrusted portion of the **TFE**.

Each database $\mathbf{C_v}$ contains copies of all data items in all databases whose security level is dominated by **v**. The copy of data item **x** in the database $\mathbf{C_u}$ is denoted by $\mathbf{x_u}$. Alternatively, if $\mathbf{L(x)=u}$ so $\mathbf{x} \in \mathbf{C_u}$, there is a copy of **x** in each database whose security level dominates **u**.

2.3 Transaction Model and Concepts

A database transaction is the execution of a set of atomic operations on the data items of the database. The operations permitted on the data items are Read(x), that returns the value stored by the data item, and Write(x), that changes the value of the data item to a specified value. Other transaction operations such as Start, Commit, and Abort, while significant for the control of transaction processing [2], need not be made explicit for communicating this protocol. In fact, only committed transactions will be considered in defining the protocol. If $\mathbf{T_i}$ is a transaction, then a Read(x) operation by $\mathbf{T_i}$ is denoted $\mathbf{r_i[x]}$, and a Write(x) operation by $\mathbf{w_i[x]}$.

The atomicity of the transaction's operations ensures that the **DBS** behaves as if it executes the operations sequentially even though it may do so concurrently. The transaction model reflects the possibility of concurrent execution in the following definition.

Definition A *transaction* $\mathbf{T_i}$ is a partially ordered set with ordering relation $<_i$ where

(1) $\mathbf{T_i} \subseteq \{\mathbf{r_i[x]} \mid \mathbf{x} \in \mathbf{D}\} \cup \{\mathbf{w_i[x]} \mid \mathbf{x} \in \mathbf{D}\}$

(2) If $\mathbf{r_i[x]}, \mathbf{w_i[x]} \in \mathbf{T_i}$, then either $\mathbf{r_i[x]} <_i \mathbf{w_i[x]}$ or $\mathbf{w_i[x]} <_i \mathbf{r_i[x]}$.

The definition requires only that operations on the same data item be ordered. Two operations, from perhaps different transactions, *conflict* if they operate on the same data item and at least one of them is a Write. Operations of several transactions can be commingled so that concurrency of execution can be extended to sets of transactions, as reflected in the following.

Definition A *complete history* **H** over a set of transactions $\mathbf{T} = \{\mathbf{T_1}, \mathbf{T_2}, \text{---}, \mathbf{T_n}\}$ is a partial order with ordering relation $<_H$ where

(1) $\mathbf{H} \supseteq \mathbf{T_1} \cup \mathbf{T_2} \cup \text{---} \cup \mathbf{T_n}$

(2) $<_H \supseteq <_1 \cup <_2 \cup \text{---} \cup <_n$

(3) If **p**, **q** are operations of **H** that conflict, then either $\mathbf{p} <_H \mathbf{q}$, or $\mathbf{q} <_H \mathbf{p}$.

Since only complete histories will be considered, they will be referred to simply as *histories.*

The preceding view of transaction processing is the view of the user, to whom replicas or versions of data items are transparent. Histories of this type will be called *one-copy histories* when it is necessary to distinguish them from histories that represent the system's view of a transaction and that deal with copies of data items.

When a set of transactions is executed by a replicated **DBS**, an operation in a transaction must be translated into the equivalent operation on some or all of the copies of the data item. A *translation function* **h** performs the mapping. For a Read(x), **h** determines the copy of **x** to be read, i.e., $\mathbf{h}(\mathbf{r_i[x]})=\{\mathbf{r_i[x_u]}\}$ for some **u**. For a Write(x), **h** determines what copies of **x** are to be updated, i.e., $\mathbf{h}(\mathbf{w_i[x]}=\{\mathbf{w_i[x_a]}, \mathbf{w_i[x_b]}, \text{---}\}$. In the case at hand, if $\mathbf{L}(\mathbf{T_i})=\mathbf{u}$, then $\mathbf{h}(\mathbf{r_i[x]})=\{\mathbf{r_i[x_u]}\}$, and $\mathbf{h}(\mathbf{w_i[x]})=\{\mathbf{w_i[x_v]} \mid \mathbf{v} \geq \mathbf{u}\}$.

The concept of a *replicated data history* is needed to represent the actions of the translated transactions on the replicated data. In the replicated architecture, two operations on data items *conflict* if they operate on the same copy of the data item and at least one of them is a Write. In the following definition, $\mathbf{o_i[x]}$ represents either a Read(x) or a Write(x) operation.

Definition A *replicated data history* **H** over a set of transactions $\mathbf{T}=\{\mathbf{T_1}, \mathbf{T_2}, \text{---}, \mathbf{T_n}\}$ is a partial order with ordering relation $<_H$ such that

(1) $\mathbf{H}=\mathbf{h}(\mathbf{T_1}) \cup \mathbf{h}(\mathbf{T_2}) \cup \text{---} \cup \mathbf{h}(\mathbf{T_n})$

(2) If $\mathbf{r_i[x]} <_i \mathbf{o_i[y]}$ in $\mathbf{T_i}$, then $\mathbf{h}(\mathbf{r_i[x]}) <_H \mathbf{p}$ for all $\mathbf{p} \in \mathbf{h}(\mathbf{o_i[y]})$

(3)If $\mathbf{w_i[x]} <_i \mathbf{o_i[y]}$ in $\mathbf{T_i}$, then $\mathbf{w_i[x_u]} <_H \mathbf{o_i[y_u]}$ for all $\mathbf{u} \in \mathbf{S}$ such that $\mathbf{w_i[x_u]} \in \mathbf{h}(\mathbf{w_i[x]})$ and $\mathbf{o_i[x_u]} \in \mathbf{h}(\mathbf{o_i[y]})$

(4) If $\mathbf{p}, \mathbf{q} \in \mathbf{H}$ and they conflict, then either $\mathbf{p} < \mathbf{q}$ or $\mathbf{q} < \mathbf{p}$

It should be noted that this definition is not as restrictive as that in [2], and though not necessary for this paper, all the results there appear to hold under this weakened definition.

Replicated data histories represent the execution of a set of transactions as seen by the entire **MLS-DBS** rather than as seen by the user, to whom copies of data items are transparent. Notice that such histories preserve the orderings stipulated by the transactions (conditions (2) and (3)).

The concepts of *reads-from* and *final write* are essential to understanding the relationships among histories over the same set of transactions. These may be defined for one-copy or replicated data histories.

Definition Let **H** be a history (one-copy or replicated data) over a set of transactions **T**.

(1) $\mathbf{T_j}$ *reads-***x***-from* $\mathbf{T_i}$ in **H** if $\mathbf{w_i[x]} <_H \mathbf{r_j[x]}$ and there is no $\mathbf{T_k} \in \mathbf{T}$ for which $\mathbf{w_i[x]} <_H \mathbf{w_k[x]} <_H \mathbf{r_j[x]}$

(2) $\mathbf{w_i[x]}$ is a *final write* of **x** in **H** if there is no $\mathbf{T_k} \in \mathbf{T}$ for which $\mathbf{w_i[x]} <_H \mathbf{w_k[x]}$

This definition clearly makes sense for one-copy histories, and does also for replicated data histories if applied to a single copy of a data item. That is, "$\mathbf{T_j}$ reads-$\mathbf{x_u}$-from $\mathbf{T_i}$" and "$\mathbf{w_i[x_u]}$ is a final write of $\mathbf{x_u}$" are meaningful. These concepts are used to define the notion of equivalent histories.

Definition Let **H** and **G** be histories of the same type (one-copy or replicated data) over a set of transactions **T**. **H** and **G** are *view equivalent* if they have exactly the same reads from relationships and the same final writes.

Correct execution of a set of transactions should appear to the user as if the transactions were executed one at a time in some order. This concept of correctness is formalized as follows.

Definition A history **H** is *serial* if for every pair of transactions $\mathbf{T_i}$, $\mathbf{T_j}$ of **H**, either all operations of $\mathbf{T_i}$ appear before all those of $\mathbf{T_j}$, or vice versa. A history **H** is *serializable* if it is view equivalent to a serial history.

Any protocol for processing transactions on a replicated architecture database may give rise to a replicated data history. This history will represent a "correct" execution of the transactions if it appears to the user that the transactions executed serially on a one-copy database. Therefore a notion of equivalence between a one-copy history and a replicated data history is necessary. This requires the following definitions.

Definition If **H** is a replicated data history, say $\mathbf{T_j}$ *reads-***x***-from* $\mathbf{T_i}$ in **H** if for some $\mathbf{u \in S}$, $\mathbf{T_j}$ reads-$\mathbf{x_u}$-from $\mathbf{T_i}$.

Definition Let **H** and $\mathbf{H_{1C}}$ be replicated data and one-copy histories, respectively, over the same set of transactions **T**. **H** and $\mathbf{H_{1C}}$ are *equivalent* if

(1) **H** and $\mathbf{H_{1C}}$ have the same reads-from relationships, i.e., $\mathbf{T_j}$ reads-**x**-from $\mathbf{T_i}$ in **H** if and only if $\mathbf{T_j}$ reads-**x**-from $\mathbf{T_i}$ in $\mathbf{H_{1C}}$.

(2) For each final write $\mathbf{w_i[x]}$ in $\mathbf{H_{1C}}$, $\mathbf{w_i[x_u]}$ is a final write in **H** for some $\mathbf{u \in S}$.

Definition A replicated data history **H** is *one-copy serializable* (**1SR**) if it is equivalent to some one-copy serial history.

One-copy serializablility is the criterion for "correctness" that will be applied to protocols for transaction processing in a replicated architecture database. The protocol to be described herein yields replicated data histories that are one-copy serializable.

To complete the specification of the model, the *update projection* $\mathbf{U_i}$ of a transaction $\mathbf{T_i}$ is defined as $\{\mathbf{w_i[x]} \mid \mathbf{w_i[x] \in T_i}\}$. For each transaction $\mathbf{T_i}$, $\mathbf{U_i}$ can be regarded as a transaction that must be executed at each database $\mathbf{C_u}$ for which $\mathbf{u > L(T_i)}$, in order to propagate the updates generated by $\mathbf{T_i}$ to copies of the data items affected. In particular, each transaction $\mathbf{T_i}$ on the replicated database can be decomposed into a primary transaction that acts on $\mathbf{C_u}$ where $\mathbf{u = L(T_i)}$, and its update projection $\mathbf{U_i}$, that acts on $\mathbf{C_v}$ for $\mathbf{v > L(T_i)}$**.** It is often useful to think of a primary component or an update projection as a one-copy transaction acting on a single back-end database. Two update transactions, or an update transaction and a primary transaction *conflict* if their (nonreplicated) parent transactions conflict. It should be noted that write-up transactions could be allowed by modifying the definition of an update projection to include the write-up operations. The extension of the model and the protocol to this case is quite complicated and including it adds a measure of complexity that would only obscure this exposition, and will be given in detail in a future paper.

3. DESCRIPTION OF THE PROTOCOL

The problem is to define a protocol for executing the primary transactions and the update projections in the "correct" way. As previously mentioned, a protocol will be considered "correct" if the resulting replicated data history is one-copy serializable.

In the following, the symbol < will be used for all order relations (on the security lattice, transactions and histories). The intended order relation will be clear from the context in which it is used. The notation $\mathbf{T_i}$ will be used for both the entire transaction on the one-copy database and its primary component on the back-end database, with the context again the arbiter.

The protocol is a variant of the traditional *primary site* algorithm [2]. Each back-end $\mathbf{C_u}$ will have a scheduler $\mathbf{P_u}$ that produces view serializable schedules for the transactions executed there (primary components and update projections). In addition, $\mathbf{P_u}$ must preserve, in its serialization ordering, the order in which it receives update transactions. There are several types of schedulers that accomplish this, among which are variants of conservative two-phase locking and conservative timestamp ordering protocols [2]. $\mathbf{P_u}$ need not be trusted.

In fact, if one has decided on a primary site type of protocol, and has decided that correctness is represented by one-copy serializability, then one is locked into a particular approach to specifying the protocol, at least in the most abstract sense. Once a serialization order between transactions and/or update projections has been established at some back-end database, the same order must be maintained between the update projections at all higher security level back-end databases, else one-copy serializability is not enforced. If update projections could be executed a at high security level back-end before they were executed at some lower level back-end, a serialization order could be established at a high level. Since this serialization order cannot be communicated downward without violating the security policy, such behavior cannot be permitted. Therefore update projections must be propagated to higher level back-end databases in an order consistent with that of the security lattice. Thus the previous assumption on the behavior of schedulers (or some other means of preserving serialization orderings) is necessary. It follows from this discussion that, under these assumptions, specifying the protocol amounts to specifying the means by which update projections are propagated upward through the back-end databases.

A list $\mathbf{Q_u}$ is associated with each back-end database $\mathbf{C_u}$. The purpose of $\mathbf{Q_u}$ is to maintain a list of the primary transactions and update projections that have been executed and committed at $\mathbf{C_u}$. The list is ordered by the serialization order of the execution of these transactions, which need not agree with the order in which transaction are actually executed or committed. The $\mathbf{Q_u}$ are used to make the correct order of execution at lower security level back-end databases available to those at higher security levels.

In addition, there is, for each $\mathbf{u} \in \mathbf{S}$, an untrusted mechanism $\mathbf{R_u}$ that maintains $\mathbf{Q_u}$ and can read the contents of $\mathbf{Q_v}$ for all $\mathbf{v} \leq \mathbf{u}$ and is considered to be part of the global scheduler.

The actual location of the $\mathbf{Q_u}$ or $\mathbf{R_u}$ is not important for the correctness of the protocol, but since any access to them must be monitored by the TCB, it is most efficient for them to be within the untrusted part of the **TFE**.

One additional concept is necessary prior to describing the protocol. Say a back-end database $\mathbf{C_u}$ *covers* $\mathbf{C_v}$ if $\mathbf{L(C_u)}$ covers $\mathbf{L(C_v)}$ in the security lattice. (Recall that **u** covers **v** in a lattice if $\mathbf{u>v}$ and there is no **w** for which $\mathbf{u>w>v}$.)

The protocol works basically as follows. A transaction is submitted to the **TFE** which dispatches it for execution to the back-end database with the same security class as the transaction. The primary transaction is executed and committed there, and the update projection is added to $\mathbf{Q_u}$, where **u** is the security level of the transaction. The update projection is then promulgated to the back-end databases for which the security class strictly dominates that of the transaction. The distribution and timing of the update projections is controlled by the untrusted process $\mathbf{R_v}$ for $\mathbf{v>u}$. If $\mathbf{C_v}$ covers $\mathbf{C_u}$, $\mathbf{R_v}$ can scan $\mathbf{Q_u}$, retrieve an update projection, and dispatch it to the scheduler at $\mathbf{C_v}$. The crucial part of the protocol is in specifying the rules for retrieval of an update projection by $\mathbf{R_v}$. If not done correctly, the resulting histories will not be 1SR.

Notice that each $\mathbf{C_u}$, in isolation, can be considered to be a one-copy database. Primary transactions with security level **u**, and update projections from transactions whose security level is dominated by **u** can be considered as transactions on this one-copy $\mathbf{C_u}$. Therefore, the concepts of scheduling, execution, and commitment can be applied to these transactions locally (at $\mathbf{C_u}$). The scheduler $\mathbf{P_u}$ can generate a serializable schedule for these transactions at $\mathbf{C_u}$ and, as they commit, place the update projections into $\mathbf{Q_u}$ in the order of an equivalent serial schedule. The reader is referred to [6] for methods of placing update projections into $\mathbf{Q_u}$ in serialization order when that order differs from the commit-time ordering.

The protocol does not permit an update projection to be scheduled at higher security level back-end databases until its primary transaction has been committed locally. In fact, an update projection cannot be scheduled at $\mathbf{C_v}$ until it has been committed at all $\mathbf{C_u}$ for $\mathbf{u<v}$.

The protocol processes transactions as follows.

At each back-end database $\mathbf{C_u}$:

I.1 Primary transactions and update projections are received from the **TFE** and submitted to the local scheduler. Actions on data items are translated into the correct actions on local copies.

I.2 As local transactions (primary transactions and update projections) are committed, a report of their commitment is sent to the **TFE**. These reports are sent in an order consistent with the serialization order determined by the local scheduler. That is. if $\mathbf{V_i}$ and $\mathbf{V_j}$ are two local transactions, primary or update, at $\mathbf{C_u}$ and the scheduler $\mathbf{P_u}$ schedules $\mathbf{V_i}$ before $\mathbf{V_j}$ in the local serialization ordering, then the commitment report for $\mathbf{V_i}$ is sent to the **TFE** before that for $\mathbf{V_j}$. If there are no conflicts between $\mathbf{V_i}$ and $\mathbf{V_j}$, then

$\mathbf{P_u}$ need not impose a serialization order on them, and the reports can be sent in any order.

At the **TFE**:

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

II.2 Whenever a commitment report for $\mathbf{U_i}$ is received from $\mathbf{C_u}$, it is added to the end of $\mathbf{Q_u}$.

II.3 The $\mathbf{R_u}$ scan the lists $\mathbf{Q_v}$ for those **v** for which $\mathbf{C_u}$ covers $\mathbf{C_v}$. $\mathbf{R_u}$ will retrieve an update projection $\mathbf{U_i}$ from $\mathbf{Q_v}$ and send it to $\mathbf{C_u}$ to be executed when the following conditions are satisfied.

a. $\mathbf{R_u}$ has already retrieved and dispatched to $\mathbf{C_u}$ all $\mathbf{U_j}$ for which $\mathbf{U_j}$ was serialized before $\mathbf{U_i}$ by $\mathbf{P_v}$.

b. If $\mathbf{C_u}$ also covers $\mathbf{C_w}$, and $\mathbf{U_i}$ can eventually appear in $\mathbf{Q_w}$, then it does appear in $\mathbf{Q_w}$.

The crux of the protocol is II.3 because it controls the order in which the primary transactions and update projections are distributed to the back-end databases for processing by holding the submission of an update transaction until all preceding updates have been submitted. It is important to understand how this step can be performed by an untrusted $\mathbf{R_u}$.

II.3.a is possible because $\mathbf{U_i}$ and $\mathbf{U_j}$ arise from transactions $\mathbf{T_i}$ and $\mathbf{T_j}$ for which, say, $\mathbf{L(T_i)\leq L(T_j)=v}$. Thus the primary transaction $\mathbf{T_j}$ obtains a serialization order with either the primary transaction $\mathbf{T_i}$ or the update projection $\mathbf{U_i}$ from $\mathbf{P_v}$. In either case, $\mathbf{Q_v}$ lists $\mathbf{U_i}$ and $\mathbf{U_j}$ in the required relative serialization order and these lower security level lists are visible to $\mathbf{R_u}$. The order established at $\mathbf{Q_v}$ is maintained as the updates migrate upward following the security lattice ordering. II.3.b is possible because if $\mathbf{U_i}$ appear in some $\mathbf{Q_v}$ where $\mathbf{C_u}$ covers $\mathbf{C_v}$ and $\mathbf{C_u}$ also covers $\mathbf{C_w}$, then $\mathbf{R_u}$ can detect whether $\mathbf{U_i}$ will eventually appear at $\mathbf{C_w}$. Suppose $\mathbf{U_i}$ arises from the primary transaction $\mathbf{T_i}$. Then $\mathbf{U_i}$ will eventually appear at $\mathbf{C_w}$ if and only if $\mathbf{w\geq L(T_i)}$. $\mathbf{R_u}$ can know the structure of the security lattice below its own level, which is sufficient to detect the required condition.

Note: As previously noted, a partially ordered set (poset) of security

classes can be used rather than a lattice of security classes at the cost of adding additional processing to the protocol. To do this, topologically sort the security poset obtaining a global linear ordering of the security classes. For each C_u, this imposes a linear ordering on the C_v that are covered by C_u. We now add condition c to Step II.3

> II.3.c. If U_i in C_v and U_j in C_w can both be retrieved by R_u, and **v** precedes **w** in the global linear order, then R_u retrieves U_i before U_j.

This is necessary because with a security poset, the following situation could arise. Suppose both C_u and C_v cover C_y and C_z (this is not possible with a security lattice). Further suppose U_1 is in Q_y and U_2 is in Q_z, and they both satisfy conditions to be retrieved by both R_u and R_v. II.3.c then determines the order in which they will be retrieved, preventing the possibility of inconsistent serialization orders being forced at higher security levels.

As mentioned earlier, the replicated architecture must pay the price of maintaining the consistency of the replicated data. In this protocol, the price is paid in two ways. First, there is the overhead of maintaining the lists in the **TFE** in terms of both the space required and the added processing. Second, because updates are delayed, and the delay increases as the distance from the primary transaction's security level to higher levels increases, transactions at higher security levels may read data that may not be current.

4. PROOF OF CORRECTNESS

For the proof of correctness of the protocol, the definition of the protocol will be modified slightly for the sake of mathematical convenience. The redefinition does not affect the concurrency of operations specified by the protocol.

The modification eases the mathematical proof by explicitly accounting for the existence of read-only transactions in the execution of a set of transactions. Because read-only transactions have an empty update projection, there is no need to record them in the Q_u, since the information there is used only to correctly propagate updates. Consequently, the serialization information for read-only transactions is not maintained by the system, nor is it necessary for the correctness of the protocol. For purposes of the proof of correctness, it would be convenient if it did. Therefore, for the proof, Q_u will not only contain the update projections of transactions executed and committed at the security level of Q_u and below, but it will also contain a marker corresponding to read-only transactions that were executed and committed at the lower levels. For the sake of the proof, then, read-only transactions will

have update projections that serve solely to mark their position in the serialization ordering of all transactions. These update projections propagate through the $\mathbf{Q_u}$ just as if they were non-empty.

One other artifice is necessary if the security lattice **S** has no maximal element. One is simply created and added to the lattice, and will be designated by **t**. No back-end database need exist for this artificial security level, but only the list $\mathbf{Q_t}$. If **S** already has a maximal element, no additional element is required.

Let $\mathbf{U}=\{\mathbf{U_i} \mid \mathbf{T_i} \in \mathbf{T}\}$ be the set of all update projections, including the null ones for the read-only transactions, and let $\mathbf{U}^*$ 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 $\pi_{uv}: \mathbf{U}^* \rightarrow \mathbf{U}^*$ that is defined as follows.

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

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

(3) Extend π_{uv} to $\mathbf{U}^*$ by recursive definition.

Lemma. If **u** and **v** are in **S**, with $\mathbf{u} \geq \mathbf{v}$, then the serial histories associated with $\pi_{uv}(\mathbf{Q_u})$ and $\mathbf{Q_v}$ are view equivalent.

Proof. The proof is by induction on the minimal length **n** of a path from **u** to **v** in the lattice **S**. The result is trivial for $\mathbf{n}=1$, since if $\mathbf{C_u}$ covers $\mathbf{C_v}$, then the order of conflicting operations in $\mathbf{Q_v}$ is the same as the order in which they are inserted into $\mathbf{Q_u}$ by the condition imposed on the back-end schedulers. If the length of the minimal path from **u** to **v** is $\mathbf{n}>1$, find **w** for which $\mathbf{u}>\mathbf{w}>\mathbf{v}$. Then the path lengths from **u** to **w** and from **w** to **v** are smaller than **n**, so the induction hypothesis is true for these paths. Thus $\mathbf{Q_w}$ is view equivalent to $\pi_{uw}(\mathbf{Q_u})$, and $\mathbf{Q_v}$ is view equivalent to $\pi_{wv}(\mathbf{Q_w})$. Since $\pi_{uv}=\pi_{wv} \circ \pi_{uw}$, the result follows.

In particular, the serialization order for all transactions that is specified by $\mathbf{Q_t}$, where **t** is the maximal class in **S**, is consistent with the serialization orderings at each back-end database. The one-copy serial history that corresponds to this ordering, say **J**, will be shown to be equivalent to the replicated data history produced by the protocol.

As an intermediate step to facilitate the proof, let **H** be the replicated data history produced by the protocol, and let **G** be the history obtained from **H** by topologically sorting the security lattice and putting the serial histories of the $\mathbf{C_u}$, as specified by the $\mathbf{Q_u}$, sequentially in this topological order.

Lemma. **G** and **H** are view equivalent as replicated data histories.

Proof. Clearly **G** and **H** have the same operations. Since reads-from relationships and final writes are properties local to each back-end database,

these local properties are preserved in the structure of **H**. That is, if $\mathbf{T_j}$ reads-$\mathbf{x_u}$-from $\mathbf{T_i}$ in **H**, then $\mathbf{T_j}$ reads-$\mathbf{x_u}$-from $\mathbf{T_i}$ in **G** as well. A similar statement holds for final writes.

Theorem. **H** is 1SR.

Proof. Since **H** is view equivalent to **G**, it suffices to show that **G** is 1SR. Claim that **G** is equivalent to the one-copy history **J**.

1) Suppose $\mathbf{w_i}[x]$ is a final write of **x** in **J**. Then if $\mathbf{w_i}[\mathbf{x_t}]$ is not a final write of $\mathbf{x_t}$ in **G**, there is a $\mathbf{T_k}$ for which $\mathbf{w_i}[\mathbf{x_t}]<\mathbf{w_k}[\mathbf{x_t}]$. Then $\mathbf{U_i}$ precedes $\mathbf{U_k}$ in **G**, and so also in $\mathbf{Q_t}$. But then $\mathbf{T_i}$ precedes $\mathbf{T_k}$ in **J**, a contradiction. Hence $\mathbf{w_i}[\mathbf{x_t}]$ is a final write of $\mathbf{x_t}$ in **G**.

2) Suppose that $\mathbf{T_j}$ reads-**x**-from $\mathbf{T_i}$ in **G**, so that for some **u**, $\mathbf{w_i}[\mathbf{x_u}]<\mathbf{r_j}[\mathbf{x_u}]$ and no other transaction at $\mathbf{C_u}$ writes $\mathbf{x_u}$ between these operations. Notice that $\mathbf{u}=\mathbf{L}(\mathbf{T_j})$, but $\mathbf{L}(\mathbf{x})=\mathbf{L}(\mathbf{T_i})$ may be strictly dominated by **u**. If $\mathbf{T_j}$ does not read-**x**-from $\mathbf{T_i}$ in **J**, then there is a $\mathbf{T_k}$ for which $\mathbf{w_i}[x]<\mathbf{w_k}[x]<\mathbf{r_j}[x]$. Then in the serial history **J**, $\mathbf{T_i}$ precedes $\mathbf{T_k}$ which precedes $\mathbf{T_j}$. Since $\mathbf{T_i}$ and $\mathbf{T_k}$ both write **x**, $\mathbf{L}(\mathbf{T_i})=\mathbf{L}(\mathbf{T_k})$ which is dominated by **u**. Since $\boldsymbol{\pi}_{\mathbf{tu}}(\mathbf{Q_t})$ specifies a serialization order that is view equivalent to that of $\mathbf{Q_u}$, $\mathbf{U_i}$ precedes $\mathbf{U_k}$ precedes $\mathbf{U_j}$ in $\mathbf{Q_u}$, that contradicts that $\mathbf{T_j}$ reads-$\mathbf{x_u}$-from $\mathbf{T_i}$.

   Conversely, suppose $\mathbf{T_j}$ reads-**x**-from $\mathbf{T_i}$ in **J**. Let $\mathbf{u}=\mathbf{L}(\mathbf{T_j})$ and $\mathbf{v}=\mathbf{L}(\mathbf{T_i})=\mathbf{L}(\mathbf{x})$, and suppose $\mathbf{T_j}$ does not read-**x**-from $\mathbf{T_i}$ in **G**. Then there is a $\mathbf{T_k}$, $\mathbf{L}(\mathbf{T_k})=\mathbf{v}$, for which $\mathbf{w_i}[\mathbf{x_u}]<\mathbf{w_k}[\mathbf{x_u}]<\mathbf{r_j}[\mathbf{x_u}]$. It follows that $\mathbf{U_i}$ precedes $\mathbf{U_k}$ precedes $\mathbf{U_j}$ in $\mathbf{Q_u}$ and thus also in **J**, contradicting that $\mathbf{T_j}$ reads-**x**-from $\mathbf{T_i}$ in **J**.

5. GARBAGE COLLECTION AND RECOVERY

In implementing the protocol as described, the size of the lists, the $\mathbf{Q_u}$, can become arbitrarily large. This may be wasteful of storage as well as increase the execution time for scanning the $\mathbf{Q_u}$ by $\mathbf{R_u}$. In order to remedy this situation, some form of *garbage collection* should be included to maintain the $\mathbf{Q_u}$ at a reasonable size.

The security policy does not allow $\mathbf{R_u}$ to access information that would indicate whether or not a particular $\mathbf{U_i}$ must be kept for future use by the protocol, as this depends on information known only at higher security levels. Therefore, garbage collection requires that trusted mechanisms be used. The

earliest point at which a particular $\mathbf{U_i}$ may be discarded from $\mathbf{Q_u}$ is when $\mathbf{U_i}$ has been retrieved from it (and dispatched to the appropriate back-end database) by all the $\mathbf{R_v}$ for which $\mathbf{C_v}$ covers $\mathbf{C_u}$.

Trusted components could be built to perform the deletions at this point, but would be inordinately complex for the task. A more likely approach would be to wait until a particular $\mathbf{U_i}$ is inserted into $\mathbf{Q_t}$ (where t is the maximum class in the security lattice). In fact, this may be the only reason for maintaining $\mathbf{Q_t}$ (other than to simply the proof of correctness), since it is otherwise unnecessary. A relatively simple trusted mechanism could then remove $\mathbf{U_i}$ from all of the relevant $\mathbf{Q_u}$. Such a mechanism can be invoked at regular intervals. Doing garbage collection at the checkpoints taken for recovery purposes may be sufficient.

As for recovery, the back-end databases have their own recovery managers, so that the only concern is the recovery of the contents of the dynamic data structures in the **TFE**. The recovery scheme to accomplish this is straightforward and quite similar to what is generally used for databases. A log is maintained in the stable storage corresponding to security level **u**. The log for security level **u** can be located on the same hardware as the back-end database $\mathbf{C_u}$ to maintain security of the recovery logs. Alternatively, a collection of single level logs could be maintained in the stable storage dedicated to the **TFE**. Whenever $\mathbf{R_u}$ receives an update report from a back-end database and adds it to $\mathbf{Q_u}$, a log entry is created and written to the log in stable storage. If the **TFE** should fail, the logs can be used to reconstruct the $\mathbf{Q_u}$.

In addition, as for databases in general, a *checkpoint* can be taken, recording the state of the whole DBS. Performing the garbage collection function at this point and pruning the logs of any unnecessary entries reduces the amount of data that is stored.

After a crash, recovery is accomplished by restoring the state at the last checkpoint and using the logs to update the **DBS** to the point of failure. The proposed technique requires little trusted code.

6. CONCLUSION

The replicated architecture for **MLS-DBS**s has the potential to provide performance significantly better that the kernelized architecture design since the work performed for a user during a single log-in session takes place within a single, conventional database system. What is needed to make the replicated architecture work is an untrusted mechanism that maintains database consistency without giving up the potential for concurrent execution of database operations. The protocol presented here provides this. The additional overhead to provide the consistency control is not excessive and relatively straightforward to implement. Most of the concurrency control and recovery

processes are provided by the back-end databases themselves. Moreover the additional code required for the management of the lists in the front-end need not be trusted, reducing the effort required for system assurance.

7. FUTURE RESEARCH

An interesting question arose while doing the work for this paper concerning the notion of "correctness" for transaction processing in a multilevel database system. The usual concept of a database transaction precludes transactions from communicating with each other. In this context, "correctness" of execution is usually interpreted as looking to the user as if their transactions operated one at a time in some order. In untrusted (ordinary) replicated database systems, this position is satisfactory and one-copy serializability is an acceptable criterion for "correctness". In the replicated architecture multilevel case, however, something more seems appropriate.

Consider the following example. In the untrusted situation, a user submits the transaction **w[x]r[x]w[y]**. That is, the user writes new value for **x**, reads it and performs some process and writes a new value for **y**. The user is assured that the value of **x** used in the ensuing process is the one just written because **w[x]** and **r[x]** conflict and are therefore ordered within the transaction. Now suppose that **x** has a low security level and **y** a high one, and that the process that reads **x** and writes **y** is a high security level analysis. The user can no longer bind the write and read operations on **x** so that the desired ordering is enforced. In the multilevel situation, the original transaction must be broken into two transactions, **w[x]** and **r[x]w[y]**, that must be submitted at two different security levels. Moreover, one-copy serializability no longer guarantees that the user's expectations about the order of execution are realized. In fact either ordering of the two transactions satisfies that criterion, regardless of which of the two transactions is submitted first.

If one holds to the common practice of identifying a user with a single level log-on session, then a user is a single level entity for whom one-copy serializability gives a consistent view of the database. But users do not view themselves in this way in a multilevel situation. Rather, one usually thinks of a person who can log-in to the system at a number of security levels, so that the situation described above can actually occur. It appears that some notion of multilevel transaction and a corresponding concept of "correctness" are required to extend the ideas of this paper to what users expect to happen.

This problem seems to be related to a number of issues that arise from commingling database theory with that of computer security. The distinction

between database user and a subject in a secure system, as noted above, seems to be at least part of the problem. Investigation of this problem and related issues will be the subject of future work.

8. ACKNOWLEDGEMENTS

The author thanks Judy Froscher and John McDermott of the Naval Research Laboratory for reviewing this paper and making many helpful suggestions. A special thanks to Sushil Jajodia and Boris Kogan for introducing the author to the field of trusted database systems.

9. REFERENCES

1. D.E. Bell and L.J. LaPadula, "Secure Computer Systems:prUnified Exposition and Multics Interpretation," The Mitre Corp., March 1976.

2. P.A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.

3. Oliver L. Costich and Ira S. Moskowitz, "Analysis of a Storage Channel in the Two Phase Commit Protocol", *Proceedings of the Foundations of Computer Security Workshop IV*, Franconia, NH 1991.

4. D. Denning, "Commutative Filters for Reducing Inference Threats in Multilevel Database Systems," *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 134-146, Oakland, CA 1985.

5. Judith N. Froscher and Catherine Meadows, "Achieving a Trusted Database Management System using Parallelism," in *Database Security II:Status and Prospects*, ed. Carl Landwehr, pp.151-160, North-Holland, 1989.

6. Sushil Jajodia and Boris Kogan, "Transaction Processing in Multilevel-Secure Databases using Replicated Architecture" in *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 360-368, Oakland, CA May 1990.

7. T.F. Keefe and W.T. Tsai, "Multiversion Concurrency Control for Multilevel Secure Database Systems" in *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 369-383, Oakland, CA May 1990.

8. T. Lunt, D. Denning, R. Schell, M. Heckman, and W. Shockley, "The SeaView Security Model," *IEEE Transactions on Software Engineering*, SE-16,6 (June 1990), pp. 593-607.

9. William T. Maimone and Ira B. Greenberg, "Single-Level Multiversion Schedulers for Multilevel Secure Database Systems" in *Proceedings of the Sixth Annual Computer Security Applications Conference*, pp.137-147, Tucson, AZ December 1990.

10. Richard C. O`Brien, J.T. Haigh, and D.J. Thomsen, "Trusted Database Consistency Policy" Rome Air Development Center Technical Report RADC-TR-90-387, December 1990.

11. M. Tamer Öszu and Patrick Valduriez, *Principles of Distributed Database Systems*, Prentice-Hall 1991.

12. John McDermott, Sushil Jajodia, and Ravi Sandhu, "A Single Level Scheduler for the Replicated Architecture for Multilevel-Secure Databases" in *Proceedings of the Seventh Annual Computer Security Applications Conference*, San Antonio, TX December 1991