A Single-Level Scheduler for the Replicated Architecture for Multilevel-Secure Databases

John P. McDermott
Naval Research Laboratory
Sushil Jajodia
Ravi S. Sandhu
George Mason University

The replicated architecture for multilevel secure database systems provides security by replicating data into separate untrusted single-level database systems. To be successful, a system using the replicated architecture must have a concurrency and replica control algorithm that does not introduce any covert channels. Jajodia and Kogan have developed one such algorithm that uses update projections and a write-all replica control algorithm. Here we describe an alternative algorithm. The new algorithm uses replicated transactions and a set of queues organized according to security class. A new definition of correctness is required for this approach, so we present one and use it to show that our algorithm is correct. The existence of this new algorithm increases the viability of the replicated architecture as an alternative to kernelized approaches.

1. Introduction

The architecture of a high-assurance multilevel secure database system is critical to its effectiveness. Conventional database system architectures do not include a trusted computing base and incorporating one requires significant change to the conventional architecture. Multilevel secure database system architectures can be characterized as either *kernelized* architectures or *frontend-backend* architectures [18]. Kernelized architectures incorporate the trusted computing base into some architectural layer of the machine that hosts the database system. Frontend-backend architectures incorporate the trusted computing base on a frontend machine and host the database system on one or more backend machines. There are many variations of either approach [10].

Kernelized architectures can place more or less trust in the database system software. For example, the Advanced Secure DBMS architecture [7] places more trust in the database system software by allowing it to process multilevel objects directly (most of the DBMS is part of the trusted computing base) and as a consequence of this potentially achieves higher performance than other approaches but costs more to assure. The Hinke-Schaefer approach [11,14,16] places less trust in the database system software and thus should cost less to build to the same level of assurance. It finesses the trust problem by storing multilevel objects in single-level fragments that are protected by the underlying trusted operating system. However, the Hinke-Schaefer approach needs to reconstruct logical multilevel objects from separate single-level objects and may give up some time performance in return.

In the frontend-backend architecture category, the integrity-lock architecture [8,5] uses a trusted frontend system to apply cryptographic check values to data stored in a single backend database system. It is generally not intended for use in high assurance systems. The SD-DBMS architecture [20,9] uses a trusted frontend to control multiple backends where the data is distributed but not replicated. Like the Hinke-Schaefer architecture, the SD-DBMS architecture places little trust in the database system software thus reducing assurance costs. Like the Hinke-Schaefer architecture, the SD-DBMS architecture can have time performance problems due to the need to materialize logical objects whose physical representation is fragmented.

Our paper is concerned with another frontend-backend architecture, which we call the *replicated architecture*. The Naval Research Laboratory, in its SINTRA project (Secure INformation Through Replicated Architecture), is investigating the replicated architecture. Like the Hinke-Schaefer architecture, little trust is placed in database system software but where the Hinke-Schaefer architecture fragments apparently-multilevel objects into single-level fragments the SINTRA architecture replicates apparently-multilevel objects as single-level copies. Lower-level data is replicated in the higher level databases (See Figure 1). The replication overcomes the object-materialization performance problem facing other architectures. Data that is apparently (to the user) multilevel is physically stored together as a single-level object in a single-level database system and thus does not need to be materialized.

The critical difficulty with the replicated approach is correct replication. Conventional approaches to replication and the related concurrency control issues are well understood. Unfortunately, the conventional solutions are either unworkable in the context of multilevel security or at best introduce covert channels. If data cannot be correctly replicated without introducing security flaws, then the replicated approach is not viable.

In this paper we show a new way of maintaining correct replication in a secure manner and relate it to the previously published algorithm of Jajodia and Kogan [12]. Our scheduler does not use any trusted components and uses conventional algorithms for the local scheduling. The new algorithm increases the viability of the replicated architecture as an alternative to the kernelized approach.

We continue in this section with a discussion of the SINTRA architecture and the problem of replicating data in a multilevel-secure database system. We also present the model and notation we use for the rest of the paper. Next, in section 2, we explain why conventional approaches to replica and concurrency control are not suitable for multilevel-secure database systems. In sections 3 and 4 we present two algorithms: write-all using update projections and a central queue and write-all using replicated transactions and multiple queues. Then we show the correctness of the new algorithm, using a new definition of correctness mandated by our replicated-transaction approach. In section 6, we compare the algorithms qualitatively and in section 7 we conclude our paper.

1.1 The SINTRA Architecture

The frontend-backend architecture with full replication has been around as a concept for some time [18]. In its SINTRA project, the Naval Research Laboratory is currently prototyping several frontend-backend architectures with full replication. What will be called in this paper the SINTRA architecture was first proposed for serious consideration by Froscher and Meadows [6].

The SINTRA architecture uses full replication to provide multilevel security. There is an untrusted backend database system for each security class. Data from dominated security classes is replicated in each backend system. Logically, the user is allowed to read down and write at the same class but physically the frontend reads all data at the same class and writes at the same class and up into dominating classes to maintain the replicas. It is important to remember that while the replicated architecture uses distributed database system technology, the replicated architecture is for centralized database systems and not for distributed systems. It could be implemented on a single machine such as LOCK [22].

Figure 1 illustrates the basic SINTRA architecture. Notice that the low data appears at all backends, blue data at the blue and high backends, etc.

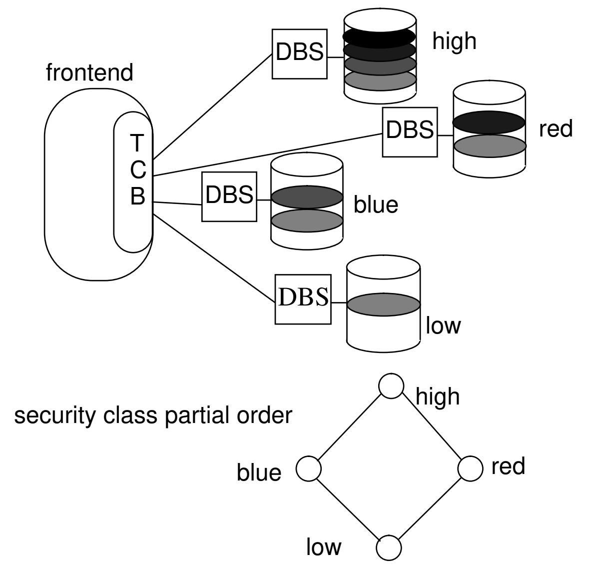

Figure 1. The Replicated Architecture

For this paper, our mathematical model of the SINTRA architecture consists of a finite set of passive entities $\boldsymbol{D}$, called *data items*, a finite set of active entities $\boldsymbol{T}$, called *transactions*, and a finite partial order (SC, $\leq$) of *security classes*.

We denote elements of $\boldsymbol{D}$ by lower case letters x, y, and z, and subscript them as x_A, x_B, x_C, etc. when emphasizing the fact that the data items in question are replicas of each other. We denote the elements of $\boldsymbol{T}$ by T or by T_i, T_j, and elements of SC by uppercase letters A, B, C, etc. We use a mapping $L{:}\boldsymbol{D}\cup\boldsymbol{T}\rightarrow SC$ to give the security class of every data element and every transaction and a similar mapping $S{:}\boldsymbol{D}\cup\boldsymbol{T}\rightarrow SC$ to give the apparent or *source security class* of every data element and every transaction. The mapping S models the fact that replicas of data may appear to the user to have a lower classification than the backend that holds the replica. For example, in the database system of Figure 1, if a transaction T with $L(T)=low$ writes a logical data item x, there will be, among others, replicas x_{red} and x_{high} with $L(x_{red})=red$ and $L(x_{high})=high$ but with $S(x_{red}) = S(x_{high}) = low$.

The model contains the following (partial) security policy:

1. Transaction T is not allowed to read data element $x \in \boldsymbol{D}$ unless $L(T)=L(x)$.
2. Transaction T is not allowed to write into data element $x \in \boldsymbol{D}$ unless $L(T)\leq L(x)$.
3. $L(T)$ and $S(T)$ do not change, and $L(x)$ and $S(x)$ do not change.

We model the replication by partitioning the data items $\boldsymbol{D}$ into a set of equivalence classes $[A]$ where $[A]=\{x \in \boldsymbol{D} \mid L(x)=A\}$. We will frequently denote equivalence class $[A]$ as D_A. The SINTRA architecture requires, for any security classes A and $B \in SC$ such that $B \leq A$, that there be a corresponding replica data item $x_A \in D_A$ if there is a data item $x_B \in D_B$.

1.2 The Problem of Maintaining Consistency Between Replicas

The concurrency control algorithm of a single-copy database system is considered to be correct if its execution of concurrent transactions is equivalent (we define equivalence more precisely in the following section) to some serial execution of those transactions on the same data items. This correctness criterion is called *serializability* [2, 21]. For replicated databases, the correctness criterion is generalized to require that an algorithm appear to execute concurrent transactions on replicas of data in a way that is equivalent to serial execution of those transactions on a single copy of the data. This is called *one-copy serializability* [2]. If the values stored in the replicas are produced by a one-copy serializable execution of the transactions then the replicas are *consistent*. Readers interested in serializability for kernelized architectures should see [16, 13].

A replicated database that has local serializability (i.e. in each back-end database system) cannot achieve one-copy serializability by simply copying everything, as the following example will show.

Example 1. Consider a hypothetical database system with the SINTRA architecture that has just two security classes A and B, with $B<A$. This database system acts as a server in a naval command and control system. One of the uses of this database system is to store *tracks*, which are collections of sensor reports associated with particular submarines, ships, or airplanes. Among the data items it stores are two logical data items x and y with $S(x)=B$, and $S(y)=A$. Data item x stores the best available position, course, and speed associated with a particular track and y stores the best available estimate of whether the track is hostile. The database has, among others, three predefined transactions: T_1, T_2, and T_3, that are used automatically by the command and control system, that is, under program control and not submitted by a human.

```
T1: write(x)
T2: read(x);
    x := EstimateX(x);
    write(x)
T3: read(x);
    read(y);
    y := EstimateY(x, y);
    write(y)
```

Transactions T_1 and T_2 originate at security class B and are also run at security class A to maintain consistency between the replicas of x. Transaction T_3 runs at security class A only. Transaction T_1 is used by an automatic-update program to update x from raw reports. Transaction T_2 at security class B is used by a program *EstimateX* to compute a best estimate of current position, course, and speed. Another program, *EstimateY*, uses transaction T_3 and sometimes changes the estimate of track hostility.

A locally correct scheduler for a backend could interleave the operations of T_1, T_2, and T_3 as long as the result was equivalent to some serial schedule. Instead of giving a locally correct interleaving of individual operations we will simply list the transactions below at each back-end database they act on and assume that each transaction is executed serially. Clearly, this execution represents local serializability at each security class.

At backend D_B: T_2; T_1

At backend D_A: T_1; T_2; T_3

This particular execution by the database concurrency control mechanism would cause the program *EstimateY* running at A to base its computation of y on *EstimateX*'s value for x (that is, T_2's value). The program *EstimateX* running T_2 at security class B would read the value of x that held before T_1 ran at security class B. Unfortunately, even though *EstimateY*, through its action in T_3, would use T_2's value for x, the value T_2 would compute for x could be different because at security class A transaction T_2 read T_1's value for x when it should have read the value of x before T_1 changed it.

The two expert systems could advise a user based on different views of the world that were an artifact of the replication. A user could see them giving contradictory results just because the local concurrency control mechanism in the database server made an arbitrary choice. If the user chose to use *EstimateX's* results at B and make his or her own estimate of track hostility, he or she might get a position, course, and speed that would indicate a hostile track. Meanwhile the program running at A *that followed the same rules as the user* would say the track was not hostile, because T_2's value of x was different at security class A. Although our backend systems were locally correct (serializable) the overall concurrency control approach did not give us consistent database results.

1.3 The Transaction Model of Concurrency Control

A *transaction* is an abstract unit of concurrent computation that executes atomically. The effects of a transaction do not interfere with the effects of other transactions that access the same data, and a transaction either happens with all of its effects made permanent or it doesn't happen and none of its effects are permanent. A useful model of a transaction must show how such properties can be achieved by composing smaller units of computation, when those smaller units are not necessarily guaranteed to compose into an atomic transaction. Thus the model must be concerned with showing potential interference between operations and with showing arbitrary orderings.

In this paper, we take one of the usual models [2, 21]: single transactions as total orders on finite sets of abstract *read*, *write*, *commit*, and *abort* operations, denoted $r[x]$, $w[x]$, c, and a, respectively. The total order establishes a relation between each operation in the transaction, i.e. either $p[x]<q[y]$ or $q[y]<p[x]$ if $p[x]$ and $q[y]$ are operations in transaction T. Each transaction has a greatest element that must be either c or a. This means that a transaction always either commits or aborts, and does nothing else after that.

The concurrent execution of a set of transactions is modeled as a partial order that contains the union of the transactions that execute concurrently, preserves the orders established for each transaction and further establishes some order between any two operations that *conflict*. Two operations $p_i[x]$ and $q_j[x]$ conflict if one of them is a write operation. We call this partial order a *history.* Notice that the partial order established by a history is not necessarily correct and that, unlike Example 1, the individual transactions do not necessarily run serially even when the history is correct.

Remark. We restrict our presentation to complete histories (i.e. the history consists only of committed transactions) [2] and simply call them histories. The extension to prefixes of complete histories is straightforward, but unnecessarily complicates our discussion.

A *serial history* H_s has all the operations of transaction T_i before transaction T_j or vice versa, for every pair of transactions T_i, T_j in H_s. If a history is equivalent to some serial history we say it is *serializable* and consider it to be correct. The kind of equivalence we use here is called *view equivalence* and is defined in terms of the *reads-from* and *final write* relationships between the transactions and operations in the two equivalent histories. Two complete histories are view equivalent if they have the same reads-froms and the same final writes over the same set of transactions. Thus we say that a history for a replicated database is one-copy serializable if it is view equivalent to some serial, single-copy history.

Definition 1. Transaction T_i *reads-x-from transaction* T_j if

1. $w_j[x]<r_i[x]$, that is, T_j's write of x is done before T_i's read of x,
2. there is no operation $w_k[x]$ such that $w_j[x]<w_k[x]<r_i[x]$, that is, no transaction T_k writes x between T_j and T_i.

Definition 2. A *final write* of data item x in history H, by transaction T_i, is a write operation $w_i[x]$ such that for any $j\neq i$, $w_j[x]<w_i[x]$, that is, all other writes of x come before T_i's write.

Normally we omit the reflexive relationships, set notation, etc. and just denote a single transaction as

$$T_2 = r_2[x]r_2[y]w_2[y]c_2$$

2. Conventional Write-All Protocols and Covert Channels

One straightforward way to implement a concurrency control algorithm for a replicated architecture is the simple immediate write-all protocol [2]. With this protocol, each write operation of a transaction is immediately replicated into all copies of the data. Reads are performed from the most convenient copy of the data. To avoid the problem of our Example 1 above, the database operations are usually synchronized by a coordinator, using a conventional concurrency control protocol (e.g. two-phase locking). Transaction commits are made atomic across replicas by some form of distributed commit protocol (e.g. two-phase commit). For instance, the transactions of Example 1 could, with two-phase locking and two-phase commit, be ordered:

at backend D_B:

$$r_2[x]r_2[y]w_2[y]c_2w_1[y]c_1$$

at backend D_A:

$$r_2[x]r_2[y]w_2[y]c_2w_1[y]c_1r_3[x]r_3[y]r_3[z]w_3[z]c_3$$

where the write operations are forced to occur in each back-end at the same time. Because the writes occur together and the commits are atomic across back-end databases, the transactions are forced to have the same reads-froms and final writes at each back-end database.

Both the concurrency control protocol and the commit protocol introduce covert channels and are thus unacceptable [12,15,4]. In the following sections we will show two algorithms that manage concurrent replicated transactions without introducing covert channels or violating the security policy, one that uses update projections and one that uses replicated transactions.

3. Write-All Using Update Projections and a Central Queue

In [12], Jajodia and Kogan presented the first concurrency and replica control algorithm for a replicated architecture. Their algorithm, which we call Algorithm J, has several desirable properties including one-copy serializability] and freedom from commit-protocol covert channels [15]. The algorithm uses update projections.

Definition 3. The *update projection* of transaction T is the transaction U obtained from transaction T by omitting the read operations of T.

Definition 4. A transaction T is an *update transaction* if T has a nonempty update projection. If T is not an update transaction then it is a query or *read-only transaction*.

Algorithm J. The write-all algorithm of [12] works by having the backend database in the same security class as a transaction T execute and commit T and then return its update projection U, if T is an update transaction. Figure 2 illustrates Algorithm J, with the arrow showing the progress of update projection U.

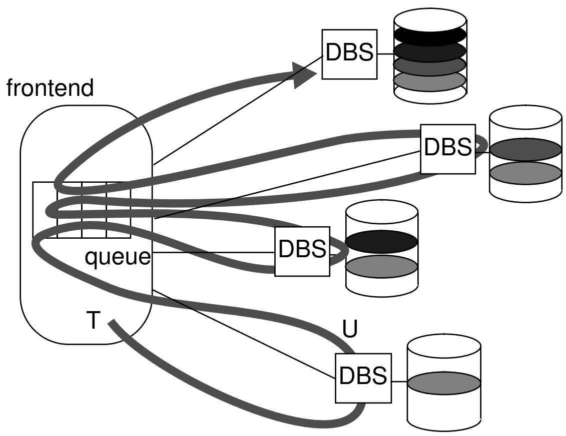

Figure 2. Algorithm J

The update projection U is subsequently synchronized in security-class-lattice order by means of a queue on the trusted frontend. For each security class that dominates the security class of T, transaction T's update projection U is merged with transactions from the next higher class, one class at a time from lowest class to highest class. As each update projection is committed, it is returned from a backend database system to the frontend in serialization order.

For those applications that have a high proportion of update transactions or with large update projections, Algorithm J might have poor performance due to the volume of message traffic and the overhead of computing update projections. For these kinds of applications it is desirable to have a replica control algorithm with less overhead due to message traffic and update projection generation.

Additionally, Algorithm J may require the use of a trusted mechanism, since the queue manager reads from all security classes and writes to all but the lowest security class. If possible, it is desirable to have the concurrency control done entirely by untrusted mechanisms.

In the following sections we present an alternative to Algorithm J, which we call Algorithm Q. The alternative algorithm sends transactions to each backend database system and does not receive and forward update projections to maintain replication. Instead of sending update projections, Algorithm Q replicates entire transactions, one for each backend where an update should be sent.

In our initial discussion of Algorithm Q we will assume each back end database system has its own *conservative timestamp ordering (CTO) scheduler* [2,21]. A *conservative* scheduler is one that never aborts a transaction for concurrency control reasons. Another reason we use conservative timestamp scheduling is that conservative timestamp ordering schedulers can easily be made to schedule transactions in the order they are received.

Remark. *Timestamp* schedulers assign timestamps (not necessarily from a real-time clock) to transactions and then schedule the individual operations of concurrent transactions by comparing timestamps. Any pair of operations from different transactions, at least one of which is a write, are scheduled in the order of the timestamps of their transactions. For database operations $p_i[x]$, $q_j[x]$ such that either $p = w$ or $q = w$ we have

$$ts(T_i) < ts(T_j) \text{ iff } p_i[x] < q_j[x]$$

where $ts(T_i)$ is the timestamp of transaction T_i.

4. Write-all with Replicated Transactions and Multiple Queues

Now we present our alternative algorithm. We will explain the algorithm first in terms of conservative timestamp ordering and then show how we can drop that requirement.

Algorithm Q replicates entire transactions into the appropriate backends. Instead of Algorithm J's single multilevel queue and scheduler, Algorithm Q uses multiple single-level queues and schedulers to order the transaction replicas in a consistent way and then send them as complete transactions to the backend database systems. The backend database systems may then schedule the transactions using any conservative scheduler that always serializes transactions in the order it receives them.

Our discussion will begin with an informal presentation of the algorithm, then we will look at some examples and conclude with a rigorous presentation of the full algorithm. The key to understanding Algorithm Q is the organization of the single-level queues. To understand that organization we need a definition.

Definition 5. Let $(SC, \leq)$ be a partial order and let a and b be elements of SC. We say that *a covers b* if

1. $b \leq a$ and $b \neq a$
2. there is no c such that $b \leq c \leq a$ and $c \neq b \neq a$.

We call b the *child* of a and a the *parent* of b. This definition can be extended to define ancestors and descendants in the obvious way. If $(SC, \leq)$ is represented as a directed acyclic graph with arcs (b, a) and a placed above b when a covers b then we have the standard Hasse diagram with the parent-child relationships made immediately clear. In the partial order of Figure 1 we would say that *high* covers *red* and *blue*.

On the frontend, at each security class, Algorithm Q has a queue for each child of that security class. Each of these queues is used to hold and order the transaction replicas propagated from the child security class to the parent security class. Figure 3 illustrates the flow of transactions through the queues.

On the frontend, at each security class, new update transactions are sent directly to the corresponding backend database system by the frontend scheduler. At the same time, replicas of the update transactions are sent to the frontend scheduler of each parent security class. The frontend scheduler at a parent security class puts the update transactions it receives from a child security class on the queue corresponding to the child security class. Because of the partial ordering of the security classes, an update transaction may arrive at a parent from more than one child. When a update transaction appears at the head of every queue it can appear in, that transaction is taken off and sent to the backend. Read-only transactions are not replicated.

At each backend, incoming transactions are timestamped and scheduled using a CTO scheduler. There is no need for the timestamps to be synchronized.

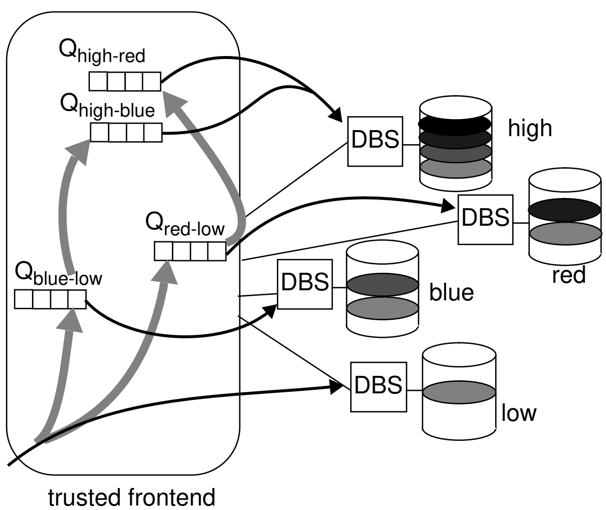

Figure 3. Algorithm Q for the Partial Order of Figure 1

Example 2. If our security class ordering (SC, ≤) is as shown in Figure 1 then we have four queues $Q_{parent\text{-}child}$: $Q_{red\text{-}low}$, $Q_{blue\text{-}low}$, $Q_{high\text{-}red}$, and $Q_{high\text{-}blue}$, as shown in Figure 3. A new transaction starting at security class *low* would go directly to the *low* backend and run to completion. At the same time, the algorithm would send a replica of the transaction to the queues $Q_{blue\text{-}low}$ and $Q_{red\text{-}low}$. When the replica reached the head of $Q_{blue\text{-}low}$ it would be sent to the *blue* backend and also to the queue $Q_{high\text{-}blue}$. When the corresponding replica reached the head of $Q_{red\text{-}low}$, it would be sent to the *red* backend and to the queue $Q_{high\text{-}red}$. When the replicas reached the heads of $Q_{high\text{-}blue}$ and $Q_{high\text{-}red}$, they would taken off together and one copy of the transaction sent to the *high* backend. In Figure 3, the shaded arrows show replicas of a transaction being sent to the queues and solid arrows show the paths to the backends.

How do we know we have the proper transactions in each queue? When there is a common transaction T at the head of every queue that T could ever be in, then we may remove T from all those queues and send it to the corresponding backend (and parents, if any). The set of security classes from which we can possibly get a transaction in a queue is determined by the paths from which a transaction can reach the parent security class.

Example 3. Suppose our security class ordering (SC, ≤) is as shown in Figure 4. The queues at A would be Q_{AB}, Q_{AC}, and Q_{AE}. Their corresponding sets P_B, P_C, P_E of security classes from which they could ever receive transactions are $P_B = \{B, D, F, H\}$, $P_C = \{C, D, G, H\}$, and $P_E = \{E, F, G, H\}$.

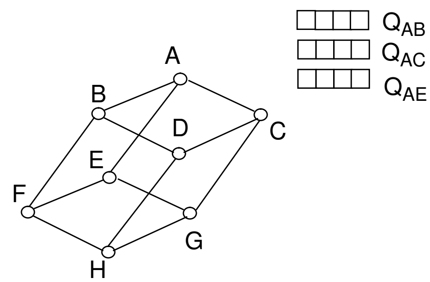

Figure 4. A More Complex Security Class Structure

The diagram in Figure 5 shows the three queues necessary for the partial order of Figure 4, assuming our algorithm started with one transaction from each class. The lines drawn through the transactions indicate where the algorithm is removing transaction replicas from the queues. The circled numbers represent one possible order in which the transactions could have been taken off and sent to the backend D_A.

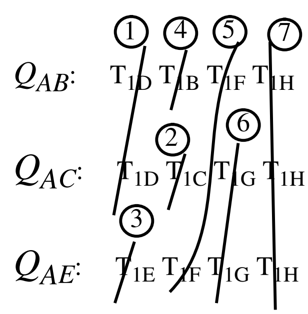

Figure 5. Removing Transactions from Queues

Example 4. Algorithm Q preserves the ordering of transactions originating at lower classes (i.e. descendants) because the queues at each ancestor in *SC* determine an ordering on the set of transactions submitted by its descendants. As long as each set of descendants has only one ancestor (i.e. no two parents share more than one child) the ordering will be preserved by simply collecting transactions in the queues. Unfortunately, some partial orders may have a directed acyclic graph where the same set of descendants have more than one ancestor. An example of such a graph is shown in Figure 6.

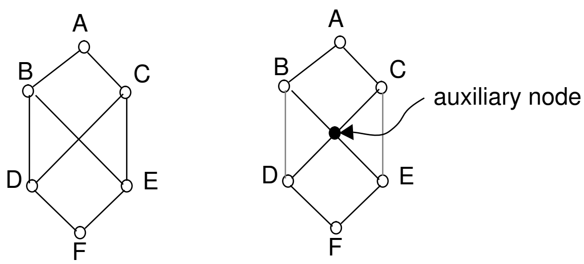

Figure 6. Descendants with More than One Parent

If one transaction is received at security classes *D* and *E*, and those arrive at different orders at *B* and *C*, then the resulting queues at *A* could look like this:

$$Q_{AB} = T_{1D}\, T_{1E}$$
$$Q_{AC} = T_{1E}\, T_{1D}$$

If this is allowed to occur, the algorithm will deadlock. It will be waiting for T_{1D} at the head of Q_{AC} but T_{1D} cannot reach the head of Q_{AC} because the algorithm is waiting for T_{1E} at the head of queue Q_{AB}.

Fortunately, the algorithm we have described above can cope with this kind of partial ordering, if we either add *auxiliary nodes* or we use *priority queues* at the common parents.

Remark. A priority queue is an abstract data type that acts like a queue, except the FIFO order is changed to one determined by some priority function that places highest priority elements at the head of the queue, etc. [1].

If auxiliary nodes were used they would not have a security class nor would they have a corresponding back end; they would just have the queues to collect transactions from the formerly common children. If the priority queue alternative is used then each queue would use a priority function that ordered the transactions at each common ancestor in the same order. Figure 6 shows an auxiliary node.

4.1 Reducing the Number of Queues

We can dispense with queues for parents with only one child. Since the ordering of transactions coming from such a child is fixed, they may be interleaved anywhere among the new transactions at the parent; there is no need for a queue to hold the single sequence of transactions received from the child. In the scheduler of Figure 3 only two queues $Q_{high\text{-}red}$ and $Q_{high\text{-}blue}$ at security class *high* are needed. A scheduler for Figure 5 would not need queues at classes *F*, *G*, or *H*.

4.2 Specification of Algorithm Q

Now we present Algorithm Q in a more rigorous way, with an eye to proving it correct. First we define a way for the algorithm to test whether an update transaction replica at the head of a queue may be taken off the queue. For this test we partition the update transactions into a system of disjoint sets of transactions. Each block of the partition contains all transactions that arrive at a security class via the same children. Then we also define a notation for the queues at the children and parents of a security class. Finally, we combine these notions into a brief specification of our algorithm.

Definition 6. First let $B_1, B_2, \ldots, B_k$ be the children of security class *A* and let $C_1, C_2, \ldots, C_m$ be the parents of security class *A*. Then for the children of *A* we define the partition *P* as follows: each block of *P* is a set of all transactions that could only reach security class *A* via paths ending at some distinct combination of these children. For example, define the block $[B_1]$ as the set of all transactions that could have only come to the queues at *A* via its child B_1 but not via any other children of *A*, define the block $[B_1 B_2]$ to be the set of all transactions that could have come to *A* via its children B_1 and B_2 but not from any other children of *A*, and so on up to block $[B_1 B_2 \ldots B_k]$ as the set of all transactions that must come via every child of *A*. Formally a block $[B_1 B_2 \ldots B_j]$ of *P* is the set

$$\{ T \in \mathbf{T} \mid S(T) < B_1 \textbf{ and } S(T) < B_2 \textbf{ and } \ldots$$
$$S(T) < B_j \textbf{ and } S(T) \not\leq B_{j+1} \textbf{ and } \ldots\ S(T) \not\leq B_k \}$$

Definition 7. Let $B_1, B_2, \ldots, B_k$ be the children of security class *A* and $C_1, C_2, \ldots, C_m$ be the parents of security class *A*. Then we denote the *queues at security class A for security classes* $B_1, B_2, \ldots, B_k$ as $Q_{AB_1}, Q_{AB_2}, \ldots Q_{AB_k}$. The queues to which transactions from security class *A* should be sent are $Q_{C_1A}, Q_{C_2A}, \ldots, Q_{C_mA}$, one at each parent security class.

Algorithm Q. Now that we have a way to talk about the partitions and queues we are going to use, we are ready to rigorously define the algorithm itself. Algorithm Q has two parts, the backend part and frontend part:

At each backend D_A: each transaction *T* is timestamped in the order in which it is received from the frontend. Transaction *T* is then scheduled using a conservative timestamp ordering scheduler. Transaction *T* commits without further communication with the frontend.

At the trusted frontend, at each security class *A*: if there is a transaction *T* such that $S(T) = A$, send it to the backend D_A and, if *T* is an update transaction and security class *A* has a parent, to each parent's queue Q_{C_1A}, Q_{C_2A}, $\ldots, Q_{C_mA}$. If A is the only child of its parent, then T is not placed in a queue, but sent directly to the parent.

If there is no transaction T such that $S(T) = A$ then find some T' in some block $[B_1B_2\ldots B_j]$ of the child set partition P such that T' appears at the head of every corresponding queue Q_{AB_1}, Q_{AB_2}, $\ldots Q_{AB_j}$. Take T' off the head of each queue and send it (once) to the backend D_A and, if A has a parent, to each parent's queue Q_{C_1A}, Q_{C_2A}, $\ldots, Q_{C_mA}$. If security class A has only one child B_1 then every T' from B_1 may be sent to D_A as soon as it arrives at A.

If a transaction T received at a backend D_A aborts it will be due to integrity problems or to a hardware or operating system fault. If the former happens, it will happen at each backend D_A; if the latter happens, the fault must be repaired and recovery procedures carried out.

5. Correctness

Since Algorithm Q is based on replicated transactions instead of replicated data, we will need a new theory to show that it is correct. Both [2] and [21] define correctness for a single schedule or history over a set of transactions. Bernstein et al. in [2] give a definition for correct scheduling over a set of transactions with replicated data. None of these theories describe our present approach, nor can they be used to prove it correct. We need a definition of correctness for replicated transactions.

5.1 Definition of Correctness

Our new algorithm replicates transactions, that is, each update transaction T is replicated into $T_A, T_B, \ldots, T_Z$ where $A, B, \ldots, Z$ are the security classes greater than or equal to $S(T)$. For each security class A, there is a corresponding history H_A that contains all of the update transaction replicas T_B with source security classes $S(T_B) \leq A$. The histories themselves should not be treated as replicated data histories [2] since no transaction replica acts outside its corresponding back end, that is, all operations of T_A are sent to backend database system D_A only.

A correct scheduling algorithm would assign transaction replicas to histories in such a way that the histories for each security class would be equivalent over the source class function S. To be completely correct, an algorithm should also result in transactions that map consistent database states to consistent database states, i.e. the resulting set of histories should be properly related to a serial history. We show what kind of equivalence is needed in the following discussion.

Definition 8. *A replicated-transaction history* $\mathcal{H}$ over a set of transactions $\boldsymbol{T}$ and a security class partial ordering SC is a set of histories over $\boldsymbol{T}$, one history H_A for each security class $A \in SC$. Each history $H_A \in \mathcal{H}$ contains all of the read-only transactions T in $\boldsymbol{T}$ such that $S(T) = A$ and contains a replica of every update transaction T' in $\boldsymbol{T}$ such that $S(T') \leq A$.

Definition 9. Let the history H be an element of $\mathcal{H}$ and A be a security class in the partial order $(SC, \leq)$. Then $\pi_A(H)$, *the security-class projection of* H, is the history obtained from H by omitting

1. all transactions whose source security class either dominates or is incomparable to security class A and
2. read-only transactions whose source security class is dominated by or is incomparable to security class A.

We will refer to $\pi_A(H)$ as *the A projection of* H. The notion of a security class projections can be extended to histories not in $\mathcal{H}$ and to sets and sequences.

With the above definitions in hand, it is now possible to define correctness for our replicated transaction scheduling approach.

Definition 10. Let $\mathcal{H}$ be a replicated-transaction history over $\boldsymbol{T}$. $\mathcal{H}$ is *security-class projection serializable* if for some serial history H_s over $\boldsymbol{T}$, and for any $H_A \in \mathcal{H}$, $H_A = \pi_A(H_s)$ where $=$ denotes view equivalence. This condition says that, for an algorithm to be correct, every history in the replicated-transaction history produced by that algorithm must be view equivalent to the corresponding security-class projection of the same serial history.

Why do we know that Definition 10 is a reasonable correctness criterion? Equivalence to serial execution is reasonable because individual transactions are designed to run as a single sequential program. Since individual transactions are designed this way, it is reasonable to expect a history containing a collection of them to be equivalent to some serial history. As we saw above in Example 1, in a replicated database, we get incorrect results if our replicated histories are not equivalent to the same serial history. In our definition we have used view equivalence. Readers who are interested in a justification of the reasonableness of view-equivalence should consult [2] or [21]. It is also possible to prove that an algorithm that always produces security-class-projection serializable replicated-transaction histories always produces one-copy serializable replicated-data histories.

5.2 Proof of Correctness for Algorithm Q

We will not give a full proof here (interested readers should see [17]), but instead we will discuss how a proof of correctness would run. To show that Algorithm Q produces a replicated-transaction history $\mathcal{H}$ that is security-class-projection serializable we must first find a suitable serial history H_s over $\boldsymbol{T}$ and then show that for any $H_A \in \mathcal{H}$ produced by Algorithm Q, $H_A = \pi_A(H_s)$.

Let A be the security class that is the greatest element of $(SC, \leq)$ or if $(SC, \leq)$ does not have a greatest element, add an auxiliary greatest element A to SC. It is easy to see that there is a serial history $H_{s'}$ such that $\pi_A(H_{s'}) = H_A$. If we only looked at update transactions we would be done but we need equivalence over all transactions in $\boldsymbol{T}$. We will now show how to construct another history H_A that is equivalent to a serial history H_s such that, for any history $H_B \in \mathcal{H}$ produced by Algorithm Q, $H_B = \pi_B(H_s)$.

To construct the history H_A we would have Algorithm Q also replicate the read-only transactions as though they were update transactions. Algorithm Q would then construct a history at security class A that contained all of the transactions in $\boldsymbol{T}$. This history H_A would also be equivalent to a serial history H_s. Then we could argue that backends that are always given replicas of transactions in the same order they were first scheduled together have histories that are security-class projection equivalent to H_s. We can appeal to the properties of conservative timestamp ordering to make this point.

To prove that the backends are always given replicas of transactions in the same order that they were first scheduled together, we argue that Algorithm Q always gives transactions to the backends in the same order. One way to do this is to use the *shuffle* of two or more sequences. The shuffle of two compatible sequences S_1 and S_2, denoted $S_1 * S_2$, is the set of all sequences that contain just the elements of $(image\ S_1) \cup (image\ S_2)$ and contain S_1 and S_2 as subsequences. Two sequences are *compatible* if they do not contain inconsistent orderings of elements common to S_1 and S_2. For an example of a shuffle, suppose $S_1 = abcde$ and $S_2 = wxdyze$. Then S_1 and S_2 are compatible and $wabxcdyze \in S_1 * S_2$. The extension to the shuffle $S_1 * S_2 * \ldots * S_k$ of more than two compatible sequences is straightforward.

We use the shuffle by showing how the queues at each security class work together to generate a shuffle of the transactions and how this shuffle is preserved over the partial order $(SC, \leq)$. This argument gives us a basis for the final arguments we sketched earlier.

6. Comparison of the Algorithms

Our new algorithm solves the problem of allowing concurrency while maintaining consistency between replicas, without introducing any covert channels. The algorithm hides both the data and transaction replication by maintaining consistency of reads-froms and final writes across security classes. It does this using replicated transactions instead of update projections as in the Jajodia-Kogan algorithm (Algorithm J). Because they work differently and have different structural properties, these algorithms give developers more design choices when solving the problem of multilevel-secure replica and concurrency control.

Algorithm J does not replicate transactions, a plus for databases processing transactions with large program texts. Algorithm Q does not thread update projections between backend databases and the trusted frontend, a plus for databases that process mostly updates. Both algorithms introduce little apparent delay with respect to use of the database during a single session because transactions take effect immediately at the session security class.

From the user's point of view at high session levels, Algorithm Q keeps its replicas of low data more current than Algorithm J. Algorithm J threads the effect of an update through each backend while Algorithm Q sends the transactions to the backends after relatively little processing, and no threading takes place at all. On the other hand, Algorithm Q performs more computation on the backend database systems, if we assume that running a replica of a transaction requires more computation that processing its corresponding update projection.

The Jajodia-Kogan algorithm does not require the use of conservative timestamp ordering schedulers in the backend database systems. As described, Algorithm Q depends on the use of a specific class of backend schedulers. However, it is possible to relax this requirement. Instead of requiring a conservative-timestamp-ordering scheduler in the backend database systems, we merely ask two things:

1. that the backend scheduler not abort transactions for concurrency control reasons, i.e. it must be conservative, and
2. that the backend scheduler always maintain a serialization order that preserves the order in which it receives conflicting transactions.

Both algorithms have the advantage of simple recovery from failures. Most of the work will be done by the untrusted backend database systems, using local mechanisms. In both Algorithm J and Algorithm Q, only a small amount of queue information must be kept in persistent storage on the frontend to support recovery from failures (of course allowing for caching, etc. as in normal database log management).

7. Conclusion

We believe that the replicated architecture has the potential for performance improvement over the kernelized approach. This potential depends on the replica and concurrency control algorithms used to keep the backend database systems consistent. Both Algorithm J and Algorithm Q solve the replica and concurrency control problems for multilevel secure database systems that enforce policies based on partial orders and have uniform integrity constraints across security classes. They do this without introducing any covert channels. Furthermore, Algorithm Q does not require any trusted code. Since Algorithm Q is structurally different from the only previous algorithm of this kind, it offers developers a choice more suited to applications with large numbers of large update transactions, or in applications where computation of update projections is undesirable. Interested readers should also be aware of the update-set-based algorithm due to Costich [3] as another alternative.

Acknowledgments

We thank Oliver Costich for his comments on our proof, Carl Landwehr and Myong Kang for their review and comments, and Judy Froscher for posing this problem. We would also like to thank the ACSAC referees for their comments.

References

1. A. Aho, J. Hopcroft, and J. Ullman, *Data Structures and Algorithms*, Addison-Wesley, 1983, ISBN 0-201-00023-7.
2. P. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987, ISBN 0-201-10715-5.
3. O. Costich, "Transaction Processing Using an Untrusted Scheduler in a Multilevel Database with Replicated Architecture", submitted for publication, 1991.
4. O. Costich and I. Moskowitz, "Analysis of a Storage Channel in the Two-Phase Commit Protocol", *Fourth Computer Security Foundations Workshop*, Franconia, NH, 1991.
5. D. Denning, "Commutative Filters for Reducing Inference Threats in Multilevel Database Systems", *Proceedings of 1985 Symposium on Security and Privacy*, Oakland, California, April 1985, pp. 134-146.
6. J. Froscher and C. Meadows, "Achieving a trusted database management system using parallelism", in *Database Security II: Status and Prospects*, ed. C. E. Landwehr, North-Holland, Amsterdam, 1989, ISBN 0-444-87483-6, pp. 253-261.
7. C. Garvey, N. Jensen, and J. Wilson, "The Advanced Secure DBMS: Making Secure DBMSs Usable", in *Database Security II: Status and Prospects*, ed. C. E. Landwehr, North-Holland, Amsterdam, 1989, ISBN 0-444-87483-6, pp. 187-195.
8. R. Graubart, "The Integrity Lock Approach to Secure Database Management", *Proceedings of 1984 Symposium on Security and Privacy*, Oakland, California, April 1984, pp.62-74.
9. J. Gray, N. Kelem, and L. Notargiacomo, "Secure Distributed Database Management System: Formal Model", *Final Technical Report, vol. 3, Rome Air Development Center TR-89-314*, Unisys Corporation, McLean, VA, December 1989.
10. T. Hinke, "DBMS Technology vs. Threats", in *Database Security: Status and Prospects*, ed. C. E. Landwehr, North-Holland, Amsterdam, 1988, pp. 57-87.
11. T. Hinke and M. Schaefer, *Secure Database Management System*, RADC-TR-75-266, Final Technical Report, System Development Corporation, November 1975.
12. S. Jajodia and B. Kogan, "Transaction Processing in Multilevel-Secure Databases Using Replicated Architecture", *Proceedings of 1990 IEEE Symposium on Security and Privacy, Oakland, CA*, pp. 360-368.
13. T. Keefe, W. Tsai, J. Srivastava, "Multilevel Secure Database Concurrency Control", *Proceedings of Sixth International Conference on Data Engineering*, Los Angeles, California, February 1990, pp. 337-344.
14. T. Lunt, D. Denning, R. Schell, M. Heckman, W. Shockley, "The SeaView Security Model", *IEEE Transactions on Software Engineering*, SE-16, 6, (June 1990), pp. 593-607.
15. G. MacEwen, "Effects of distributed system technology on database security: A survey", in *Database Security: Status and Prospects*, ed. C. E. Landwehr, North-Holland, Amsterdam, 1988, pp. 253-261.
16. W. Maimone and I. Greenberg, "Single-Level Multiversion Schedulers for Multilevel Secure Database Systems", *Proceedings of Sixth Annual Computer Security Applications Conference*, Tucson, AZ, December, 1990, pp. 137-147.
17. J. McDermott, S. Jajodia, and R. Sandhu, "Maintaining Consistency In Multilevel-secure Databases That Use A Replicated Architecture", submitted for publication, 1991.
18. "Multilevel Data Management", Committee on Multilevel Data Management, Air Force Studies Board, National Research Council, Washington, DC, 1983.
19. National Computer Security Center, *Trusted Database Management System Interpretation of the Trusted System Evaluation Criteria*, NCSC-TG-021, April 1991.
20. J. O'Connor and J. Gray, "A Distributed Architecture for Multilevel Database Security", *Proceedings of the 11th National Computer Security Conference,* Baltimore, Maryland, October 1989, pp. 179-187.
21. C. Papadimitriou, *The Theory of Database Concurrency Control*, Computer Science Press, 1986, ISBN 0-88175-027-1.
22. O. Saydjari, J. Beckman, and J.Leaman, "Locking Computers Securely", *Proceedings of the 10th National Computer Security Conference*, NBS, 1987, pp. 129-141.