A secure concurrency control protocol for real-time databases

Ravi Mukkamala
Department of Computer Science, Old Dominion University
Norfolk, VA 23529-0162, USA

Sang H. Son
Department of Computer Science, University of Virginia
Charlottesville, VA 22903, USA

Abstract

Database systems for real-time applications must satisfy timing constraints associated with transactions, in addition to maintaining data consistency. Multilevel security requirements introduce a new dimension to transaction processing in real-time database systems. In this paper, we propose a novel concurrency control protocol that can meet the real-time, security, and serializability conditions using a primary copy and a secondary copy for each data object. First, we discuss the conflicting nature of the requirements and then present our protocol. We state and prove the properties characterizing our protocol. The implementation details of the data object representation and the additional processing overhead are also discussed. Finally, opportunities for tuning the protocol to suit different application domains are presented.

1 INTRODUCTION

Database security is concerned with the ability of a database management system to enforce a security policy governing the disclosure, modification or destruction of information. Most secure database systems use an access control mechanism based on the Bell-LaPadula model (Bell and LaPadula, 1976). This model is stated in terms of subjects and objects. An object is understood to be a data file, record or a field within a record. A subject is an active process that requests access to objects. Every object is assigned a classification and every subject a clearance. Classifications and clearances are collectively referred to as security classes (or levels) and they are partially ordered. The Bell-LaPadula model imposes the following restrictions on all data accesses:

a) Simple Security Property: A subject is allowed read access to an object only if the former's clearance is identical to or higher (in the partial order) than the latter's classification.

b) The *-Property: A subject is allowed write access to an object only if the former's clearance is identical to or lower than the latter's classification.

A real-time database management system (RTDBMS) is a transaction processing system where transactions have explicit timing constraints. Typically a timing constraint is expressed in the form of a deadline, a certain time in the future by which a transaction needs to be completed. In a real-time system, transactions must be scheduled and processed in such a way that they can be completed before their corresponding deadline expires. Conventional data models and databases are not adequate for time-critical applications. They are designed to provide good average performance, while possibly yielding unacceptable worst-case response times. As advances in multilevel security take place, MLS/DBMSs are also required to support real-time requirements. As more and more of such systems are in use, one cannot avoid the need for integrating real-time transaction processing techniques into MLS/DBMSs. In (Son 1993 and David 1995), the security impact on real-time database systems is studied, but to the best of our knowledge, no work has been reported on developing DBMSs that are multilevel secure and that support real-time requirements.

While the Bell-LaPadula model prevents direct flow of information from a higher access class to a lower access class, the conditions are not sufficient to ensure that security is not violated indirectly through what are known as covert channels (Lampson 1973). A covert channel allows indirect transfer of information from a subject at a higher access class to a subject at a lower access class. Especially, in the context of databases where concurrency control is used to manage the concurrent execution of operations by different subjects on the same data object, a covert channel arises when a resource or object in the database is shared between subjects with different access classes. The requirements of maintaining database consistency and the need to allow aborting a low-priority transaction when a high-priority transaction with conflicting data requirements arrives, have a potential to create a covert channel. For a more detailed description of the problem of concurrency control in secure databases, the reader is referred to (David and Son 1993, David et al. 1995). In this paper, we concern ourselves with concurrency control mechanisms that have to satisfy both security and real-time requirements. In particular, we deal with database security that allows read-down (Simple security property) but does not allow write-up (Ammann et al. 1992). In other words,

(1) A transaction T cannot read a data object d unless level(T)$\geq$level(d).
(2) Transaction T cannot write a data object d unless level(T)=level(d).

The main objective of this paper is to present a concurrency control protocol that satisfies the data security, the data consistency, and the real-time requirements without introducing covert channels. Our protocol is based on the basic two-phase locking (2PL) and supports the following. properties:

(1) All committed transactions are serializable.

(2) A low-level * transaction is neither delayed (delay security) nor otherwise affected (value security) due to data contention with high-level transactions.
(3) A high-priority transaction is neither blocked nor aborted by low-priority transactions due to data contention on low-level data.
(4) It is possible to trade-off serializability and non-interference for improved capability in meeting transaction deadlines.

The paper is organized as follows. In Section 2, we discuss the correctness issues for secure schedulers. The interaction between security and real-time system requirements is discussed in Section 3. The impact of the basic two-phase locking on both real-time and secure systems is illustrated in Section 4. In Section 5, we describe the database and transaction model adopted in the paper. Section 6 has the details of the proposed protocol, referred to as SRT-2PL. We state and prove some of the important properties of our protocol in Section 7. The system trade-offs stated in objective (4) above are discussed in Section 8. Finally, Section 9 summarizes the results and future work.

2 CORRECTNESS OF SECURE SCHEDULERS

Covert channel analysis and removal is the single most important issue in multilevel secure concurrency control. The notion of *non-interference* has been proposed as a simple and intuitively satisfying definition of what it means for a system to be secure (Goguen and J. Meseguer 1982). The property of *non-interference* states that the output as seen by a subject must be unaffected by the inputs of another subject at a higher-level. This means that a subject at a lower-level class should not be able to distinguish between the outputs from the system in response to an input sequence including actions from a higher level subject and an input sequence in which all inputs at a higher level have been removed (Keefe et al. 1990).

An extensive analysis of the possible covert channels in a secure concurrency control mechanism and the necessary and sufficient conditions for a secure, interference-free scheduler are given in (Keefe et al. 1990). Three of these properties are of relevance to the real-time secure two phase locking protocol discussed in this paper. For the following definitions, given a schedule s and an access level l, $purge(s,l)$ is the schedule with all actions at a level $> l$ removed from s.

1) *Value Security*: A scheduler satisfies this property if values read by a subject are not affected by actions with higher subject classification levels. Stated formally, for an input schedule p, the output schedule s is said to be *value secure* if $purge(s,l)$ is view equivalent to the output schedule produced for $purge(p,l)$.
2) *Delay Security*: This property ensures that the delay experienced by an action is not affected by the actions of a subject at a higher classification level. For an input schedule p and an output schedule s, a scheduler is *delay secure* if for all levels l in p, each of the actions a_l in $purge(p,l)$ is delayed in the output schedule produced for $purge(p,l)$ if and only if it is delayed in $purge(s,l)$.

*Here, the term *level* is used to refer to the security classification and *priority* to refer to the real-time classification.

3) *Recovery Security*: A set of transactions is in a deadlock state when every transaction in the set is waiting for an event that can only be caused by another transaction in the set (such as release of a lock). Deadlock is a problem unique to locking protocols and is not an issue in timestamp schedulers and optimistic concurrency control. Even these schedulers, however, can reach a state from which they cannot continue without aborting one or more transactions. For simplicity, these two conditions are lumped together and called as deadlock (Keefe et al. 1990).

The concurrency control protocol discussed in this paper satisfies all the three properties. The details are presented in Section 7.

3 SECURITY AND REAL-TIME REQUIREMENTS

The property of non-interference has the unfortunate effect of degrading performance for transactions at a higher level. In a secure environment, a transaction at a higher level:

- cannot cause the aborting of a transaction at a lower level. If it is allowed to do so, it is possible that it can control the number of times a lower level transaction is aborted, thereby opening a covert channel.
- cannot conflict with a transaction at a lower access class. If such a conflict does occur, the higher level transaction has to be blocked or aborted, not the low level transaction.
- cannot be granted greater priority of execution over a transaction at a lower access class.

There have been a number of papers in the real-time databases literature that have explored priority based scheduling approaches with respect to conventional databases (Abbot and Garcia-Molina 1992, Sha et al. 1990, Son et al. 1992). The priority usually reflects how close the transaction is to missing its deadline. Priority-based scheduling of real-time transactions, however, interacts with the property of non-interference which has to be satisfied by secure schedulers (Keefe et al. 1990). For example, take the sequence of transactions input to a scheduler as shown (the transactions arrived in the T_1, T_2, T_3, T_4 order):

T_1 (SECRET)	:	R(X)			
T_2 (UNCLASSIFIED)	:		W(X)		
T_3 (UNCLASSIFIED)	:			W(X)	
T_4 (UNCLASSIFIED)	:				R(X)

Assume that T_1, T_2, T_3 and T_4 have priorities 5, 7, 10 and 12 respectively and the priority assignment scheme is such that if $priority(T_2) > priority(T_1)$, then T_2 has greater criticalness and has to be scheduled ahead of T_1. In the above example, T_2 and T_3 are initially blocked by T_1 when they arrive. When T_1 completes execution, T_3 is scheduled ahead of T_2, since it has a greater priority than T_2 and the transaction execution order would be T_1,T_3,T_2,T_4. However, if the transaction T_1 is removed, the execution order would be T_2,T_3,T_4 because T_2 would have been scheduled as soon as it had arrived. The presence of the SECRET transaction T_1 thus changes the value read by the UNCLASSIFIED transaction T_4, which is a violation of *value security*. Delay security is also violated, since the presence of T_1 delays both T_2 and T_3.

Therefore, to satisfy the correctness properties discussed in Section 2 (to close all covert channels), we see that a very high performance penalty is being paid. The proposed concurrency control provides mechanisms by which both security and real-time system requirements can be met by the system, and thereby avoiding the security violation conditions discussed above.

4 IMPACT OF TWO-PHASE LOCKING ON REAL-TIME AND SECURE SYSTEMS

Two-phase locking (2PL) is the most widely used concurrency control in database systems. Here, the current lock holder is never aborted due to a conflicting request from another transaction. The new request is blocked until the current holders release their locks. In some cases, the requesting transaction is aborted, and regenerated after some delay.

The basic two-phase locking, however, does not work for secure databases because a transaction at a lower access class (say T_l) cannot be blocked due to a conflicting lock held by a transaction at a higher access class (T_h). If T_l were somehow allowed to continue with its execution in spite of the conflict, then non-interference would be satisfied. The basic principle behind the secure two-phase locking protocol is to try to simulate execution of Basic 2PL without blocking the lower access class transactions by higher access class transactions. Consider the transactions T_1 and T_2 in the above example. Since T_1 arrived first and obtained a read-lock on x, T_2 would be blocked waiting for T_1 to commit and release read-lock on x. Such an action has a potential for covert channel, and hence unacceptable in a secure environment. For example, if a high-level user intends to pass information to a low-level user, in violation of the $*$-property, the two users could adopt the following protocol:

(i) The high-level user generates a HIGH transaction and the low-level user generates a LOW transaction, each consisting of just one operation on a data object x at low-level.

(ii) At time t, the HIGH transaction submits a read on x when it wants to send bit 1, and submits none if it wants to send bit 0. Shortly thereafter, the LOW transaction submits a write on x.

(iii) In the absence of other transactions contending for x, when HIGH has read-locked x, LOW is either blocked or aborted. When HIGH has not submitted any lock request, LOW is granted the lock immediately. In this way, high-user can covertly send information to low-user due to the violation of the delay security.

Suppose we were to modify the basic 2PL for a real-time environment where meeting the deadlines is the main objective. In this case, whenever a high-priority job (probably with a closer deadline) has a conflicting lock request with a low-priority lock holder, the current holder is aborted and the lock is reassigned to the high-priority job. Once again, there is a potential for a covert channel.

(i) The high-level user generates a HIGH transaction at high-priority and the low-level user generates a LOW transaction at a low-priority, each consisting of just one operation on a data object x at low-level.

(ii) At time t, the LOW transaction submits a write on x. Shortly thereafter, the HIGH transaction submits a read on x if it wants to send bit 1; submits none if it wants to send bit 0.

(iii) In the absence of other transactions contending for x, a write-lock would be initially granted to LOW. However, on the arrival of HIGH's request, LOW is aborted. When HIGH has not submitted a lock request, and in the absence of other high-priority contentions for x, LOW completes its transaction. In this way, a high-user can covertly send information to low-user.

The proposed concurrency control protocol is intended to avoid both these situations.

5 DATABASE SYSTEM MODEL

Our model of the database system has three components: (i) Transaction manager (ii) Resource Scheduler, and (iii) Data manager.

5.1 Transaction Manager (TM)

All user transactions arrive at the transaction manager. It is responsible for

1. Requesting and obtaining all required locks for each transaction from the lock manager prior to the beginning of transaction execution (no new lock requests are generated by a transaction during its execution),
2. Requesting the scheduler for execution of individual operations of a transaction, and
3. Committing or aborting a transaction, and requesting for the release of the locks held by the transaction.

The lock manager component of TM is responsible for managing the locks. Here we assume that a lock status is: unlock, read-lock, or write-lock. In addition, the usual lock conflict rules are applied: (i) If a data object is unlocked, either read/write lock can be granted. (ii) If a data object is read-locked then only read-lock requests can be granted; write-locks will be blocked (or denied); (iii) If a data object is write-locked then both read and write lock requests are blocked (or denied).

The other functions of the transaction manager include obtaining resources from the resource scheduler during transaction execution. In order to enforce the three types of security discussed in Section 2, it is necessary that the transaction manager be a trusted component.

5.2 Resource Scheduler (RS)

Once a transaction has procured all the required locks, it can start its execution. Now it needs to do the actual read/write accesses to the database as well as obtain CPU and I/O time for the operations. The resource scheduler is responsible for these allocations. In particular, it should take the timing requirements (e.g., deadlines) of individual transactions into consideration while allocating the resources. Once again, the resource scheduler is to take into account both the real-time requirements and the non-interference properties of database security. For this reason, this component is assumed to be trusted.

5.3 Data Manager (DM)

The data manager is responsible for managing (i.e., providing the read and write operations) the data objects. To provide for non-interference between the high-level and low-level transaction requests, the data manager maintains a *primary* and a *secondary* copy for each data object. While the primary copy is used for read/write accesses by transactions that are at the same security level as the object, the secondary copy is used for read accesses by higher level transactions. For example, a data object X created by a confidential user will have the primary copy made available for read/write access by the confidential users. The secondary copy of X is used for read-only access by the secret and top-secret users. Accordingly, a read/write access on the primary copy is never blocked due to the presence of readers on the secondary copy. Similarly, a high-priority read-access operation on the secondary copy is never blocked due to low-priority writers on the primary copy.

6 SRT-2PL: SECURE REAL-TIME TWO-PHASE LOCKING PROTOCOL

As stated earlier, the proposed concurrency protocol only resolves the security and real-time system problems arising due to data contention. Accordingly, we concentrate only on these problems, and let the other resource allocation problems (i.e., CPU and I/O scheduling) be handled using existing methods (Kao and Garcia-Molina 1994, Rajkumar et al. 1995).

6.1 Data Object Representation

A key aspect of our protocol is the method used to represent the data objects so as to meet objectives (1)-(4) stated in Section 1. We describe the proposed representation here.

- Each data object has a primary and a secondary copy.
- The *primary copy* P_x of an object x is maintained as a normal data object. It is accessed (read/write) by transactions at the same security level as the object. (Since we do not allow write-up operations, it can only be updated by transactions at the same level.)
- The *secondary copy* is accessed (read) by transactions at a security level higher than the data object (i.e., for read-down operations). It is maintained as two data structures:

  (i) A normal data object, S_x, and
  (ii) a single queue of updates, Q_x.

  The queue Q_x contains updates that have been performed (in that order) on the primary copy (P_x) but yet to be performed on the secondary copy (S_x). Each update entry in

Q_x, say $Q_x[j]$, has a set of transaction identifiers (IDs) associated with it. We refer to the set as $TID_x[j]$.

In other words, the primary copy of a data object is read and updated by transactions at the same security level as the data object. The secondary copy, however, is accessed by transactions at higher security levels than the data object itself, and only for read operations. The management of secondary copies is the responsibility of the data manager. The details of the maintenance of these data structures is explained below.

6.2 The Protocol

Here, we describe our protocol for concurrency control. The rules according to which our protocol manages its locks and operations are as follows:

- Each transaction needs to procure all the required locks before starting its execution. In other words, strict static locking is assumed in the protocol. While this requirement may seem unduly restrictive, it is necessary to obtain strict 1-copy serializability. In environments where the strict serializability condition may be relaxed, the static lock requirement also may be relaxed. But in this paper, we do not deal with such extensions.
- The locking rules of the basic 2PL are followed on per copy basis. In other words, the primary and secondary copies of an object are treated independently for the purpose of resolving lock-conflicts. A read-lock request at the level of a data object is only blocked when the primary copy P_x is write-locked. Similarly, a write-lock request at this level is blocked only when the primary copy is read-locked or write-locked. Especially, it should be noted that a write-lock request on a primary copy is never blocked due to the presence of read-locks on the secondary copy. Obviously, the read-lock requests on the primary copy are never blocked due to the read-locks on the secondary.
  The locking rule at the primary copy can, however, be changed to meet the real-time requirements as follows: "Abort the current holder if the requester's priority is higher." It should be observed that when lock conflicts arise on the primary copy, both the current lock holder and the requester are at the same security level as that of the object. Hence, such aborts, if necessary will not result in interference or covert channels.
- Since the secondary copy is only accessed for read by high-level transactions, there are no data conflicts among high-level transactions. In other words, a high-level transaction is never blocked on its request for a read-lock on a secondary copy.
- Though the queue of updates Q_x is modified (indirectly) by transactions at the same level, these transactions are not required (in fact, not allowed) to obtain write-locks on the secondary copy. As described below, the updates are in fact placed in Q_x by the data manager. Since, eventually the updates need to be carried out on S_x, there may be an interval of time during which a high-level read operation (not a lock request) is temporarily blocked while S_x is being modified. While this does not violate the non-interference property, it may violate the real-time requirements of a high-priority high-level reader. We avoid this problem by a clever data structure management discussed below.

We now discuss the details of the database operations under our protocol.

- **Read-lock request $rl_i[x]$ from transaction T_i:** Two cases arise:

  - Level(x)= Level(T_i): Check the status of the primary copy of x. If it is unlock or readlock, then grant the lock. (Some complex real-time concurrency protocols such as OMP (Rajkumar et al. 1995) make further checks prior to granting the lock to T_i.) If the status is writelock, then enqueue the request, abort the request, or abort the current holder depending on the real-time scheduling policy. (For example, if T_i has an earlier deadline than the current holder or if it is more critical than the holder, then the current holder may be aborted.) As stated above, since all affected transactions are at the same level as x and T_i, no security interference is caused by any of these options.
  - Level(x)< Level(T_i): This is a high-level lock request for the secondary copy. Since the actual read operation may take place at a later time, and since it is possible that a secondary copy be updated during this interval, the current state of the secondary copy is marked by sending the current position of Q_x along with the lock grant to the TM. In addition, if the current size of Q_x is j, then the requester's transaction ID is added to $TID_x[j]$. For example, if the update queue currently has three entries, then the third entry is marked and its identification returned to T_i. In addition, the ID of T_i is added to $TID_x[3]$. Since there is no blocking due to data contention, there is no security violation due to interference.

- **Write-lock request $wl_i[x]$ from transaction T_i:** Obviously, both x and T_i are at the same level and hence the operation is on the primary copy. If the primary copy of x is unlocked, then grant the lock to T_i. (As before, some real-time policies may deny lock to T_i if they anticipate a higher-priority request to be arriving soon. This is especially relevant when aborting a current holder is disallowed.) If x is read/write locked, then as before, enqueue the request, abort the request, or abort the holder(s) (i.e. the readers or the writer) depending on the real-time scheduling policy.
- **Release read-lock request $rrl_i[x]$ from transaction T_i:** Two cases arise:

  - Level(x)= Level(T_i): Release the lock on the primary copy of x and check if any pending write-lock requests can be granted.
  - Level(x) < Level(T_i): This is an operation on the secondary copy. Accordingly, release the lock on S_x and remove the entry from the corresponding TID_x entry. For example, if T_i was informed of entry 3 at the time of lock grant, then the ID of T_i was also added to $TID_x[3]$. This entry is now removed. As discussed later, this entry may no longer be in the third position in Q_x.

- **Release write-lock request $rwl_i[x]$ from transaction T_i:** This pertains to the primary copy. Accordingly, release the lock on the primary copy and check if any pending requests can be granted a lock.

In addition to the concurrency control-related operations (request/release locks) discussed above, we need to describe the read/write operations in the context of primary and secondary copies. Here is a summary of these procedures.

- **Read request $r_i[x]$ from transaction T_i:**

- Level(x)= Level(T_i): The usual read operation is performed on the primary copy. Scheduling the operation is itself a function of the real-time resource scheduler (not discussed here).
- Level(x) < Level(T_i): The read operation is on the secondary copy. Accordingly, the corresponding read is executed as if the status of the object is the same as it was at the time of read-lock request grant. The read is executed as if all updates up to and including the identified entry (sent to T_i at the time of readlock) are carried out on the data object. In other words, read requests from transactions T_i and T_j, arriving at the same time, may be offered a different object view if they have obtained locks at different status values of Q_x. As shown later, this is required to maintain serializability of transactions.

- **Write request $w_i[x]$ from transaction T_i:** The primary copy is updated. Simultaneously, the latest update entry is added to the update queue Q_x of the secondary copy. Both these updates are done atomically. However, as discussed below, the current readers operating on the secondary copy may not be affected by it.

6.3 Design Issues related to the Secondary Copy

Since the secondary copy plays a key role in our protocol, it is important to discuss the design and implementation issues related to it. Here, we discuss some important issues and possible solutions.

(i) *When should the updates in the queue at a secondary copy be carried out on the data object?* Once a transaction manager has received a grant for a read-lock on a secondary copy (due to read-down), along with the marked entry, the data manager has to guarantee to the transaction that its view of the object will be unaltered until it is committed or aborted. At the time the transaction commits or aborts, the guarantee will expire. As mentioned above, this functionality is implemented using an update queue Q_x and a set TID_x of transaction IDs with each entry in Q_x. If at any time it is determined that all $TID_x[j]$ sets up to some point k are empty (i.e., $\forall j \; 1 \leq j \leq k \; TID_x[k] = \phi$) then these updates (up to j) may be carried out on the secondary version. All updates that have been carried out are automatically removed. In other words, as high-level readers commit/abort, the secondary copy is updated, and the updates are removed from the queue. These activities are coordinated by the data manager in coordination with the transaction manager.

(ii) *How is the superposition of update queue on the secondary copy achieved?* A related concern is the maintenance of the update queue and presenting the data for access to higher-level transactions. For this we need to know the type of the object under consideration. Suppose it is a simple data structure such as an integer or a floating-point variable, then a queue of integers/floating-points is maintained in Q_x and any read access can directly read its entry in the queue as the value. In fact, the operation of secondary copy update as discussed above also becomes quite simple. On the other hand, if is a large file, or a complex data structure such a 3-D image, then the queue could contain just the updates (in some predetermined format). When a transaction requires to read the data object with respect to a specific entry in Q_x, then somehow the secondary copy and subsequent updates should be taken into consideration.

Depending on the type of data object and the availability of memory, if in fact it is possible to create temporary images of data objects corresponding to each transaction (or update), then the problem is much more simplified. There is a trade-off among storage space, processing overhead, and meeting the real-time deadlines. Clearly, since the high-level and low-level transactions access different copies, the non-interference property is still maintained by the scheme.

7 PROPERTIES

In this section, we state and prove some of the important properties of the proposed concurrency control protocol. For the sake of brevity, the style of the proofs is more intuitive and less formal.

To prove the correctness properties (i.e., serializability), we use the simple definitions of history and serialization graph (SG). The formal definitions for these concepts can be found in (Bernstein et al. 1987). A history is a partial order of operations that represents the execution of a set of transactions. Any two conflicting operations must be comparable. Let H denote a history. The serialization graph for H, denoted by $SG(H)$, is a directed graph whose nodes are committed transactions in H and whose edges are all $T_i \rightarrow T_j$ $(i \neq j)$ such that one of T_i's operations precedes and conflicts with one of T_j's operations in H. To prove that $SG(H)$ is serializable, we only have to prove that $SG(H)$ is acyclic (Bernstein et al. 1987).

First, we focus on transactions executed at a single security level.

Lemma 1 *All transactions at a single level are serializable.*

Proof: Let the history H contain all transactions committed at level L. All transactions with read/write operations on data objects at level L (and no read operations at lower levels) are clearly serializable due to 2PL (Bernstein et al. 1987).

However, when transactions read data at lower levels, then there can be additional dependencies due to the reads. Whether or not such dependencies due to lower-level reads, when combined with the dependencies due to operations on primary copies, create cycles in $SG(H)$ is the question that needs to be resolved here.

We claim that no such cycles are possible due to our static locking policy where a transaction is required to procure all locks before it starts execution. Hence, if T_1 at level L precedes T_2 (also at level L) due to read/write operations at level L, then T_2 cannot precede T_1 due to read operations at levels lower than L. This implies that there cannot be a cycle in $SG(H)$ between T_1 and T_2. The same argument can be extended even when we consider more than two transactions. Hence H is serializable. ●

Property 1 *All committed transactions (independent of their level) are serializable.*

Proof: Intuitively, since we have a static locking policy where a transaction is required to obtain all locks (irrespective of levels) prior to the beginning of its execution, we claim that it is not possible for any two transactions to have contradicting dependency relationships at different security levels. Following is a semiformal proof of this argument.

Consider levels L and L' where $L > L'$ (i.e., L is HIGH and L' is LOW). Since transactions at each level are serialized (by Lemma 1), let $T_1 \rightarrow T_2 \rightarrow \ldots T_{m-1} \rightarrow T_m$ be the serialization order at level L and $t_1 \rightarrow t_2 \rightarrow \ldots t_{n-1} \rightarrow t_n$ be the order at level L' of committed transactions. (For simplicity, we assume total ordering.)

Suppose, by way of contradiction, there is no serialization order among all these transactions. Then there is a cycle in the corresponding serialization graph such that one of the following two cases is possible:

Case 1: $T_i \rightarrow t_j \rightarrow t_k \rightarrow T_i$: Since we allow only read-down operations for higher level transactions on lower level objects, $T_i \rightarrow t_j$ implies that there is a data object x (at level L') that was read by T_i and later modified by t_j. Similarly, $t_k \rightarrow T_i$ implies that there is a data object y (at level L') that was written by t_k and later read by T_i. Since all locks were obtained by T_i at the beginning of its execution, it had obtained locks on the secondary copies of x and y also at the beginning. Hence,

- $T_i \rightarrow t_j$ implies that t_j was committed after T_i started. (Note that the secondary update queue is atomically updated with the primary.)
- $t_k \rightarrow T_i$ implies that t_k was committed prior to T_i's start.

From the above, we can conclude that t_k was committed before t_j. However, this contradicts our assumption that $t_j \rightarrow t_k$. Thus, it is not possible to have a cycle such that $T_i \rightarrow t_j \rightarrow t_k \rightarrow T_i$.

Case 2: $t_i \rightarrow T_j \rightarrow T_k \rightarrow t_i$: $t_i \rightarrow T_j$ implies that there is a data object x (at level L') that was modified by t_i and later read by T_j. Similarly, $T_k \rightarrow t_i$ implies that there is a data object y (at level L') that was read by T_k and later modified by t_i. Using the same argument as above for locking and secondary copy management, we have the following.

- $t_i \rightarrow T_j$ implies that t_i was committed before T_j started.
- $T_k \rightarrow t_i$ implies that T_k started prior to t_i's commit.

From the above we can conclude that T_k started prior to T_j. Once again, since all locks are obtained at the beginning of a transaction execution, this implies that either T_k and T_j are concurrent or $T_k \rightarrow T_j$. However, this contradicts our assumption that $T_j \rightarrow T_k$. Thus, it is not possible to have a cycle such that $t_i \rightarrow T_j \rightarrow T_k \rightarrow t_i$.

Hence, since it is not possible to have a cycle in the serialization graph of committed transactions at levels L and L', the combined history is also serializable. The same proof can be extended even when transactions from more than two levels are considered. Thus, all committed transactions in the system are serializable. ●

Property 2 *A low-level transaction is neither delayed (delay security) nor otherwise affected (value security) due to data contention with high-level transactions.*

Proof: By multilevel security definition, a transaction can read/write lock data objects at its own level and readlock objects at lower levels. In our protocol, the former is achieved by locking the primary copy and the latter by locking the secondary copy. Suppose $Level(T_1) > Level(T_2)$, the only possible cases of data contention between T_1 and T_2 are as follows:

- T_1 and T_2 read-down x: This is the case when $Level(x) < Level(T_2)$. Since both operate on the secondary copy of x, and it is a read operation, there is no data contention.
- T_1 reads-down and T_2 writes x: This is the case when $Level(x) = Level(T_2)$. While T_1 operates on the secondary copy of x, T_2 operates on the primary copy. Hence, there is no data contention.
- T_1 reads-down and T_2 reads x: As before, $Level(x) = Level(T_2)$. Since both are read operations, there is no conflict. In addition, since T_1 operates on the secondary copy and T_2 on the primary copy, there is no interference.

Since high-level and low-level transactions are never in conflict with each other in regard to data access, neither delay-security nor value-security violations are possible, due to data contention.●

Property 3 *A high-priority transaction is neither blocked nor aborted by low-priority transactions due to data contention on low-level data.*

Proof. Let T_1 and T_2 be two transactions such that $Priority(T_1) > Priority(T_2)$. Three cases arise here.
Case 1: $Level(T_1) > Level(T_2)$. Suppose they both access data object x. We need to consider two subcases.
Case 1a: Suppose $Level(x) = Level(T_2)$. Then, T_1 is a read-down operation accessing the secondary copy and T_2 accesses the primary copy of x. Even though, the secondary queue is updated atomically with respect to primary copy updates, a reader on secondary copy is never blocked by such updates (due to the chosen data structures). Accordingly, T_2 can never block or delay T_1 due to data contention.
Case 1b: Suppose $Level(x) < Level(T_2)$. Here, both T_1 and T_2 are read-down operations on the secondary copy. Clearly, there is no data contention problem between T_1 and T_2. However, there may be an interference due to I/O or CPU scheduling. But other existing scheduling algorithms may be used to resolve such problems (Kao and Garcia-Molina 1994, Rajkumar et al. 1995).

Thus, Case 1 cannot result in blocking or delay of T_1 by T_2.

Case 2: $Level(T_1) = Level(T_2)$. Suppose they both access data object x. Here, we need to consider only the case where $Level(x) < Level(T_2)$. Clearly, both T_1 and T_2 are read-down operations and the nonblocking is argued the same way as in Case 1b above.
Case 3: $Level(T_1) < Level(T_2)$. If they both access a data object x, we have two subcases to consider.
Case 3a: Suppose $Level(x) = Level(T_1)$. Then, T_2 is a read-down operation accessing the secondary copy and T_1 accesses the primary copy of x. Since the low-level read/write operations are never blocked due to read-down on secondary, T_1 is never blocked by T_2 due to data contention.
Case 3b: Suppose $Level(x) < Level(T_1)$. Here, both T_1 and T_2 are read-down operations on the secondary copy. This is similar to Case 1b.

Since cases 1-3 cover all possibilities, and under each case, the high-priority transaction is never blocked or delayed due to data contention on lower data, the property is always valid. ●

8 REAL-TIME AND SECURITY TRADE-OFFS

As discussed previously, the requirements of real-time systems and the multilevel security systems combined with the restrictions imposed by the serializability criterion as correctness pose several design and implementation challenges. While real-time systems consider meeting the deadlines and hence giving importance to high-priority transactions, multilevel secure systems consider the non-interference with the low-level transactions as critical. The basic two-phase locking (where a requester waits until the conflicting lock holder releases the lock) with correctness condition of serializability, in fact, gets in the way of achieving both the above objectives.

As discussed earlier, and formally proved in Section 7, the proposed concurrency control achieves these objectives. However, it is not without additional cost. In order to enforce the *non-interference* property for multilevel security, we have implemented each object as a primary copy and a secondary copy pair. Due to the choice of the data structure for the secondary copy (once again dictated by the non-interference and serializability criterion), the operations on the secondary copy are much more expensive. Each read on this copy requires the superposition of update queue ($Q(x)$ entries on to the secondary data object copy (S_x) and presenting it to the high-level readers.

Real-time system designers generally argue that all operations should have predictable execution times. But the read-down operation is certainly not predictable in time since its execution time depends on the type of the object and the length of the update queue. In fact, in our protocol, no guaranteed bounds can be placed on the size of the secondary queue.

However, this does not mean that the protocol is not suitable for real-time systems. Instead, the implication is that we need to make trade-offs in terms of serializability, predictability, larger memory, and some interference to meet the needs of specific applications. Here are a few suggestions for such trade-offs.

- When meeting the real-time requirements is more important, and when it is semantically acceptable for a specific high-level (and high-priority) transaction to directly read the secondary copy, ignore the entries in the update queue, then let the transaction read S_x directly. The additional overhead and non-deterministic execution problems can thus be overcome to meet the high-priority transaction deadlines.
- When meeting the real-time deadlines of a critical high-level transaction is important and when it is also necessary that the transaction read the most up-to-date value, then let the high-level transaction read the primary copy directly, thereby reducing the overhead for read-downs. However, this is at the cost of interference and a possible covert-channel.
- The trade-off between real-time and basic 2PL arises when a high-priority transaction (requester) has a data conflict with a low-priority transaction (current lock holder). While the basic 2PL lets the requester wait until the lock is released, it is an unacceptable solution for real-time systems. For this reason, alternate versions of 2PL such as Hybrid 2PL (Son et al. 1992) have been suggested to handle the problem. However, to avoid the possibility of covert channels, such alternates should be adopted only in a controlled manner.
- Incorporate alternate correctness criteria such as the one in (Jajodia and Atluri 1992) and thereby more efficiently meeting the real-time and noninterference conditions.

While the above list is by no means exhaustive, it provides ideas of where certain types of trade-offs can be made and how the protocol can be implemented.

9 CONCLUSION

In this paper, we proposed a novel concurrency control protocol for multilevel secure real-time database systems. While the two-phase locking is the underlying concurrency control mechanism, we have introduced non-interference property by having a secondary copy for read-down operations. In addition, it is possible to include several existing scheduling algorithms for both data conflict resolution and other resource allocations within the protocol.

One of the main features of the protocol is its ability to be tuned to specific requirements. Trade-offs can be made in terms of serializability, limited interference, and limited missed deadlines. In fact, the fine tuning can be made by on-line monitoring of the deadline missing rate of important (high-level) transactions, and making dynamic decisions as to the read-down operations and other abort/commit decisions. This is similar to a feedback loop in a control system.

We are currently in the process of simulating the protocol and observing its behavior quantitatively. We plan to look at different trade-off situations under different application domains.

10 REFERENCES

Abbott, R. K. and Garcia-Molina, H. (1992) Scheduling Real-Time Transactions: A Performance Evaluation. *ACM Transactions on Database Systems*, **17**, 513-60.

Ammann, P. and Jajodia, S. (1992) A Timestamp Ordering Algorithm for Secure, Single-version, Multi-level Databases. in *Database Security, V: Status and Prospects*, (ed. C.E. Landwehr and S. Jajodia), Elsevier Science Publishers B.V., 191-202.

Bell, D.E. and LaPadula, L.J. (1976) Secure Computer Systems: Unified Exposition and Multics Interpretation. The Mitre Corp. Bernstein, P.A., Hadzilacos, V., and Goodman, N. (1987) Concurrency Control and Recovery in Database Systems. Addison-Wesley, Reading, MA.

David, R. and Son, S.H. (1993) A Secure Two Phase Locking Protocol. *Proc. the 12th Symposium on Reliable Distributed Systems*, Princeton, NJ.

David, R., Son, S.H., and Mukkamala, R. (1995) Supporting Security Requirements in Multilevel Real-time Databases. *Proc. 1995 IEEE Symp. Security and Privacy,* 199-210, Oakland, California.

Goguen, J.A. and Meseguer, J. (1982) Security Policy and Security Models. *Proc. the IEEE Symposium on Security and Privacy*, 11-20.

Jajodia, S. and Atluri, V. (1992) Alternative Correctness Criteria for Concurrent Execution of Transactions in Multilevel Secure Databases. *Proc. the IEEE Symposium on Security and Privacy*, Oakland, CA, May 1992.

Kao, B. and Garcia-Molina, H. (1994) An Overview of Real-Time Database Systems, in *Real-Time Computing,* (eds: W.A. Halang and A.D. Stoyenko) NATO ASI Series F, **127**, Springer-Verlag, 261-82.

Keefe, T.F., Tsai, W.T., and Srivastava, J. (1990) Multilevel Secure Database Concurrency Control. ***Proc. the Sixth International Conference on Data Engineering***, 337-44, Los Angeles, CA. Lampson, B.W. (1973) A Note on the Confinement Problem. ***Communications of the ACM***, **16**, 613-5.

Lee, J. and Son, S.H. (1995) Concurrency Control Algorithms for Real-Time Database Systems, in *Performance of Concurrency Control Mechanisms in Centralized Database Systems* (ed. Vijay Kumar), Prentice Hall, 429-60.

Rajkumar, R., Sha, L., Lehoczky, J.P., and Ramamritham, K. (1995) An Optimal Priority Inheritance Policy for Synchronization, in ***Advances in Real-Time Systems,*** (ed. S.H. Son), Prentice-Hall, 249-71.

Sha, L., Rajkumar, R., and Lehoczky, J.P. (1990) Priority Inheritance Protocol: An Approach to Real-time Synchronization. ***IEEE Trans. Computers,*** 1175-85.

Son, S.H., Lee, J., and Lin, Y. (1992) Hybrid Protocols Using Dynamic Adjustment of Serialization Order for Real-Time Concurrency Control. ***Real-Time Systems Journal***, **4**, 269-76.

Son, S.H. and Thuraisingham, B. (1993) Towards a Multilevel Secure Database Management System for Real-Time Applications. ***Proc. IEEE Workshop on Real-Time Applications***, New York, NY.

11 BIOGRAPHY

Ravi Mukkamala is an Associate Professor in the Department of Computer Science at the Old Dominion University. He received the B.S. degree in electronics and telecommunications engineering from Osmania University in 1976, and the M.Tech. degree in computer systems from Indian Institute of Technology, Kanpur in 1978. He earned the Ph.D. in computer science from the University of Iowa in 1987. Before moving to an academic environment, he worked as a systems analyst at Telco-Pune, India, from 1978-81 and as a systems consultant at ACCI-Hyderabad, India, from 1981-83.

Dr. Mukkamala's research interests include distributed database systems, data security, high-speed data communications, and performance analysis. He has published over 75 technical papers in conference proceedings and journals in these areas.

Sang Hyuk Son is an Associate Professor in the Department of Computer Science at the University of Virginia. He received the B.S. degree in electronics engineering from Seoul National University in 1976, and the M.S. degree in electrical engineering from Korea Advanced Institute of Science and Technology (KAIST) in 1978. He earned the Ph.D. in computer science form University of Maryland, College Park in 1986. He was a Visiting Professor at KAIST during 1994-1995.

His research interests include real-time computing, database systems, distributed systems, and database security. He has served on numerous program committees of international conferences on those areas, and published over 100 papers in journals and conference proceedings. He was an ACM National Lecturer for 1991-1993. He served as the Program Chair of the 10th IEEE Workshop on Real-Time Operating Systems and Software, and the General Chair of the same workshop in 1994. He is serving as the Program Chair for Workshop on Research Issues and Applications of Real-Time Database Systems, to be held in 1996. He also edited the book "Advances in Real-Time Systems," published by Prentice-Hall in 1995. Dr. Son is a member of the IEEE Computer Society and the ACM.