Proc. Third European Symposium on Research in Computer Security, Brighton, UK, November 7-9, 1994, pages 403-428.

Supporting Object-based High-assurance Write-up in Multilevel Databases for the Replicated Architecture

Roshan K. Thomas and Ravi S. Sandhu[1]

Center for Secure Information Systems &
Department of Information and Software Systems Engineering
George Mason University, Fairfax, Virginia 22030-4444, USA
email: {rthomas, sandhu}@isse.gmu.edu

Abstract

We discuss the support of high-assurance write-up actions in multilevel secure object-oriented databases under the replicated architecture. In this architecture, there exists a separate untrusted single-level database for each security level. Data is replicated across these databases (or containers), as each database stores a copy of all the data whose class is dominated by that of the database. Our work utilizes an underlying message filter based object-oriented security model. Supporting message-based write-up actions with synchronous semantics directly impacts condidentiality, integrity, and performance issues. Also, an important concern in the replicated architecture is the maintenance of the mutual consistency of the replicated data. In this paper we offer solutions to support write-up actions while preserving the conflicting goals of confidentiality, integrity,and efficiency and at the same time demonstrate how the effects of updates arising from write-up actions are replicated correctly to guarantee such mutual consistency. Finally, we wish to emphasize that our elaboration of the message filter model demands minimum functionality from the TCB that is hosted within the trusted front end (TFE), and further requires no trusted subjects (i.e. subjects who are exempted, perhaps partially, from the usual mandatory controls). Collectively, these make verification of our solutions easier, since we have the assurance that covert channels cannot be introduced through the TFE.

Keyword Codes: D.1.5; D.4.6; K.6.5
Keywords: Object-oriented Programming, Security and Protection

1 Introduction

The replicated architecture for multilevel secure database management systems (mls DBMSs) has lately experienced a resurgence in the research community. It represents one of the three architectures identified by the Woods Hole study organized by the U.S. Air Force [19]. These architectures were motivated by the need to build multilevel secure DBMSs from existing untrusted DBMSs. The distinguishing feature of the replicated architecture is that lower level data is replicated at higher levels. To be more precise, for any given security level, a physically

[1]The work of both authors was partially supported by the National Security Agency through contract MDA904-92–C-5140. We are indebted to Pete Sell, Howard Stainer and Mike Ware for their support and encouragement in making this work possible.

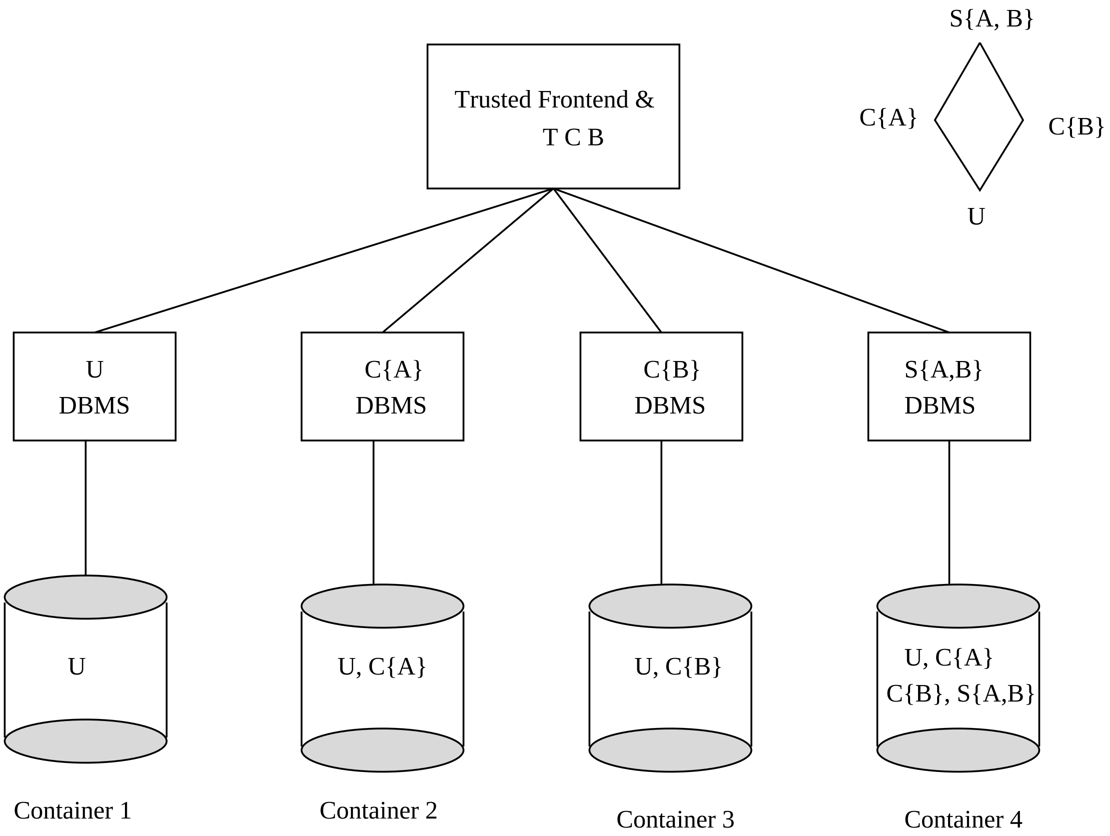


Figure 1: The replicated architecture illustrating containers for a simple lattice

separate DBMS is used to manage data at or below the level. In our further discussions, we use the term "container" to be synonymous with "database". These backend databases are untrusted, and rely on a trusted front end (TFE) that hosts the trusted computing base (TCB), for access mediation. The replicated architecture, as elaborated for a simple lattice, is shown in figure 1. Thus an object classified at U and stored in the first container is replicated across the other containers 2, 3, and 4. Such replicas when stored at containers 2, 3, and 4, are no longer considered to be at level U, rather are classified at the level of their respective containers. However, as far as applications are concerned, these replicas make up one logical object and is thus identified by a single object identifier.

The advantages and security of the replicated architecture stem from the fact that users (or subjects acting on their behalf) at different levels are physically isolated from one another, and that a user is able to accomplish all tasks (multilevel queries and updates at his/her level) from the data stored at a single DBMS. This is because a properly cleared user who logs in to the system at security level l, will be assigned to the DBMS at l. All data that is classified below levels l and stored at the lower level databases is replicated, and thus available, at the DBMS at l. Thus, for example, security threats from covert channels due to read-down operations in multilevel queries do not arise in this architecture.

The benefits of the replicated architecture come at the cost (and complexity) of the replica control schemes needed to keep the replicas of the data mutually consistent. To make this architecture commercially viable, these schemes would not only have to be efficient, but in addition must be secure (in that they do not introduce covert channels). It is important to note that covert channels can be introduced in this architecture only through the TFE.

Replica and concurrency control algorithms for relational databases under the replicated architecture have appeared in recent literature [1, 4, 5, 7, 11]. These algorithms have contributed to a better understanding of the complexity that arises due to the interaction between concurrency control, replication, and multilevel security. Perhaps the most significant advancement in this area was reported in [1]. Here it was observed that many algorithms for the replicated architecture could produce schedules that are not serializable, due to the distributed nature of decision making and synchronization. The authors in [1] pursue two approaches to address this. The first calls for global synchronization while the second restricts the security structure to avoid such synchronization.

In this paper, we turn our attention to object-oriented databases. With the ever increasing interest in object-oriented databases, we believe our effort here is timely, and one that we hope will provide impetus for further work in the area. The object-oriented security model that we utilize in this paper is based on a message filter component in the TCB that mediates messages sent between objects at various security levels [6]. The elaboration of this message filter object-oriented security model for trusted subject and kernelized architectures has been reported elsewhere in the literature [14, 15, 16]. In this paper, we elaborate the message-filtering functions under the replicated architecture, and in particular, focus on write-up actions. The solutions presented here are not prone to the problems reported in [1] since we use a forkstamping scheme for centralized decision making.

The rest of this paper is organized as follows. Section 2 presents some background material covering object-oriented databases and the message filter model. Section 3 explores the implications of the message filtering approach to security, for the replicated architecture. Section 4 discusses how the activity of a single user session is replicated across containers, while section 5 discusses how such sessions are synchronized across containers. Finally, section 6 concludes the paper.

2 Background

In this section we give some background to the message filter object-oriented security model and the concurrency and synchronization problems that arise when write-up actions are supported.

2.1 The Message Filter Model

The message filter model is one of several proposals addressing mandatory security in multilevel object-oriented databases that have appeared in the recent literature [6, 8, 9, 12, 13, 18]. The model [6, 14, 15, 16] is based on the view that the task of enforcing mandatory confidentiality essentially reduces to that of controlling and filtering the exchange of messages between objects. Objects and messages thus constitute the main entities in the model. Every object is assigned a single classification. The security policy is captured in a filtering algorithm, and enforced by a message filter component.

The message filter algorithm is given in figure 2. (In this and other algorithms, the % symbol is used to delimit comments.) Cases (1) through (4) deal with abstract messages, which are processed by application and user-defined methods. Cases (5) through (7) deal with primitive messages, which are directly processed by system defined methods. In case (1), the sender and receiver are at the same security level, and the message g_1 and its reply are allowed to pass. In

% *let* $g_1 = (h_1, (p_1, \ldots, p_k), r)$ *be the message sent from object* o_1 *to object* o_2 *where*

% h_1 *is the message name,* $p_1, \ldots, p_k$ *are message parameters,* r *is the return value*

if $o_1 \neq o_2 \vee h_1 \notin$ {**read, write, create**} **then case**
% *i.e.,* g_1 *is a non-primitive message*

(1) $L(o_1) = L(o_2)$: % *let* g_1 *pass, let reply pass*
  invoke t_2 **with** $rlevel(t_2) \leftarrow rlevel(t_1)$;
  $r \leftarrow$ reply from t_2; **return** r **to** t_1;

(2) $L(o_1) <> L(o_2)$: % *block* g_1*, inject NIL reply*
  $r \leftarrow$ NIL; **return** r **to** t_1;

(3) $L(o_1) < L(o_2)$: % *let* g_1 *pass, inject NIL reply, ignore actual reply*
  $r \leftarrow$ NIL; **return** r **to** t_1;
  invoke t_2 **with** $rlevel(t_2) \leftarrow \mathrm{lub}[L(o_2), rlevel(t_1)]$;
  % *where lub denotes least upper bound*
  discard reply from t_2;

(4) $L(o_1) > L(o_2)$: % *let* g_1 *pass, let reply pass*
  invoke t_2 **with** $rlevel(t_2) \leftarrow rlevel(t_1)$;
  $r \leftarrow$ reply from t_2; **return** r **to** t_1;

end case;

if $o_1 = o_2 \wedge h_1 \in$ {**read, write, create**} **then case**
% *i.e.,* g_1 *is a primitive message*

% *let* v_i *be the value that is to be bound to attribute* a_i

(5) $g_1 = (\mathbf{read}, (a_j), r)$: % *allow unconditionally*
  $r \leftarrow$ value of a_j; **return** r **to** t_1;

(6) $g_1 = (\mathbf{write}, (a_j, v_j), r)$: % *allow if status of* t_1 *is unrestricted*
  if $rlevel(t_1) = L(o_1)$
    then $[a_j \leftarrow v_j;\ r \leftarrow$ SUCCESS$]$
    else $r \leftarrow$ FAILURE;
  return r **to** t_1;

(7) $g_1 = (\mathbf{create}, (v_1, \ldots, v_k, S_j), r)$: % *allow if status of* t_1 *is unrestricted relative to* S_j
  if $rlevel(t_1) \leq S_j$
    then [CREATE i **with** values $v_1, \ldots, v_k$ **and** $L(i) \leftarrow S_j;\ r \leftarrow i$]
    else $r \leftarrow$ FAILURE;
  return r **to** t_1;

end case;

Figure 2: Message filtering algorithm

case (2) the levels are incomparable and thus the filter blocks the message from getting to the receiver object, and further injects a NIL reply. Case (3) involves a receiver at a higher level than the sender. The message is allowed to pass but the filter discards the actual reply, and substitutes a NIL instead. In case (4), the receiver object is at a lower level than the sender and the filter allows both the message and the reply to pass unaltered.

In cases (1), (3), and (4) the method in the receiver object is invoked at a security level given by the variable *rlevel*. The intuitive significance of *rlevel* is that it keeps track of the least upper bound (lub) of all objects encountered in a chain of method invocations, going back to the root of the chain. The value of *rlevel* needs to be computed for each receiver method invocation. In cases (1) and (4) the *rlevel* of the receiver method is the same as the *rlevel* of the sender method. In case (3), *rlevel* is the least upper bound of the *rlevel* of the sender method, and the classification of the receiver object. The purpose of *rlevel* is to implement the notion of restricted method invocations so as to prevent write-down violations. It is easy to see that if t_i is a method invocation in object o_i then $rlevel(t_i) \geq \mathrm{L}(o_i)$. We say that a method invocation t_i has a *restricted status* if $rlevel(t_i) > \mathrm{L}(o_i)$. When t_i is restricted, it can no longer update the state of the object o_i, it belongs to.

The message filtering algorithm presented above can be thought of as an abstract, non-executable, specification of the filtering functions. A close examination of the execution and implementation requirements for such a specification bring several issues to the forefront. In particular, dealing with the timing of replies to write-up messages (case 3 of the filtering algorithm) requires careful attention to potential downward signaling channels [14, 15, 16, 17]. Such channels are opened up if a low level sender method is resumed on the termination of the higher receiver method and receipt of the NIL reply. The solutions pursued in [15, 16, 17] have been to return instantaenous NIL replies and to execute the methods (computations) in the sender and receiver objects concurrently on issuing write-up messages. Now if the application being modeled calls for synchronous message passing semantics, the challenge then is to synchronize the concurrent computations to achieve equivalence to a sequential (synchronous) execution. When such equivalence can be guaranteed, we say that the concurrent computations preserve *serial correctness*. If serial correctness cannot be guaranteed, the integrity of the database may be compromised. Lastly, it should be noted that the signaling channel threat does not exist in kernelized architectures. But it turns out that synchronous semantics are not implementable in such architectures as there exists no trusted subjects.[2] Hence concurrent computations are still the most efficient way to process write-up messages requiring synchronous semantics.

2.2 Concurrency, Scheduling and Serial Correctness

We now elaborate on concurrency and serial correctness in more general terms. We can visualize the set of concurrent computations issued by a user as belonging to a *user session* and forming a tree such as the one shown in figure 3. The label on the arrows indicate the order in which the messages and the associated computations (methods) would be processed in a serial (synchronous) execution. Note that this order can be derived by a depth-first traversal of the tree. Serial correctness requires that a computation such as 3(TS) in the tree, see all the latest updates of lower level computations to its left, and no updates of lower level computations to its right. Thus 3(TS) should see the latest updates of 2(S) but not of 4(C) and 6(S). This is

[2] The term "trusted" is used often in the literature to convey one of two different notions of trust. In the first case, it conveys the fact that something is trusted to be correct. In the second case, we mean that some subject is exempted from mandatory confidentiality controls; in particular the simple-security and $\star$-properties in the Bell-Lapadula framework. It is the latter sense of trust that we refer to in this paper.

achieved with the help of a multi-version synchronization scheme that ensures that the versions of objects at levels C (confidential) and S (secret) that are available to 3(TS) are the ones that existed before 4(C) and 6(S) were created (forked). Further, serial correctness also mandates that a computation such as 3(TS) not get ahead of earlier forked ones to its left. Thus 3(TS) should not be started until 2(S) and its children (if any) have terminated.

If no system component has a global snapshot (such as that embedded in a tree) of the entire set of computations, then we need to explicitly capture the global serial order of messages and computations. This can be done by a scheme that assigns a unique forkstamp to each computation, as shown in figure 3. Starting with an initial forkstamp of 0000 for the root, every subsequent child of the root is given a forkstamp by progressively incrementing the most significant digit of this initial stamp by one. To generalize this for the entire tree, we require that with increasing levels, a less significant digit be incremented.

We can now succinctly state the requirements for serial correctness in terms of the following constraints that need to hold whenever a computation c is started at a level l:

- **Correctness-constraint 1:** There cannot exist any earlier forked computation (i.e. with a smaller forkstamp) at level l, that is pending execution;

- **Correctness-constraint 2:** All current non-ancestral as well as future executions of computations that have forkstamps smaller than that of c, would have to be at levels l or higher;

- **Correctness-constraint 3:** At each level below l, the object versions read by c would have to be the latest ones created by computations such as k, that have the largest forkstamp that is still less than the forkstamp of c. If k is an ancestor of c, then the latest version given to c is the one that was created by k just before c was forked.

From the above discussion it should be clear that we need to enforce some discipline on concurrent computations as arbitrary concurrency makes synchronization difficult and could lead to the violation of serial correctness (thereby affecting the integrity of objects). A scheduling strategy which guarantees serial correctness and at the same time enforces some discipline on concurrency, must take into account the following considerations.

- The scheduling strategy itself must be secure in that it should not introduce any signaling channels.

- The amount of unnecessary delay a computation experiences before it is started should be reduced.

The first condition above requires that a low-level computation never be delayed waiting for the termination of another one at a higher or incomparable level. If this were allowed, a potential for a signaling channel is again opened up. The second consideration admits a family of scheduling strategies offering varying degrees of performance. Informally, we say a computation is unnecessarily delayed if it is denied immediate execution on being forked, for reasons other than the violation of serial correctness.

We now consider two scheduling strategies that appear to approach the ends of a spectrum of secure (and correct) scheduling strategies, and a third one that lies somewhere in the middle of such a spectrum. These schemes that lie at the ends of this spectrum are referred to as *conservative* and *aggressive* schemes, and they are governed by the following invariants, respectively.

Inv-conservative: *A computation is executing at a level l only if all computations at lower levels, and all computations with smaller fork stamps at level l, have terminated.*

Inv-aggressive: *A computation is executing at a level l only if all non-ancestor computations (in the corresponding computation tree) with smaller fork stamps at levels l or lower, have terminated.*

Given a lattice of security levels, the conservative scheme essentially reduces to executing computations on a level-by-level basis in forkstamp order, starting at the lowest level in the lattice. At any point, only computations at incomparable levels can be concurrently executing. However, with the aggressive scheme, we are not following a level-by-level approach. Rather, a forked computation is denied immediate execution only if (at the time of fork) there exists at least one non-ancestral lower level computation with an earlier (smaller) forkstamp, that has not terminated. If denied execution, such a computation is queued and later released for execution when this condition is no longer true (as a result of one or more terminations). Figures 4 and 5 illustrate the progressive execution of the tree of concurrent computations in figure 3 under the conservative and aggressive strategies, respectively. In each of these figures, the termination of one or more computations (indicated by shaded circles) advances the tree to the next stage. As can be seen in these figures, the tree progresses to termination fastest under the aggressive scheme, since it induces no unnecessary delays. We conjecture that there exists several other variations of the above three scheduling schemes. Finally, it is important to note that the security of these schemes stem from the fact a low level computation is never suspended (delayed) because of a higher one.

3 Message-filtering in the Replicated Architecture

When we consider the implementation of message filtering in the replicated architecture, the very nature of the architecture poses a different and unique set of problems. We have to deal with security and integrity aspects of processing data within a single container as well as multiple containers.

3.1 Message-filtering Revisited

Consider first the issues pertinent to a singler container. The way objects are replicated and classified at the various containers, and the fact that only a subject cleared to the level of a container can access the data at the container, have the following implications:

1. There exists no need for message filtering between objects at a single container.
2. Method invocations resulting from messages sent between objects at a single container can be processed sequentially, as there exists no downward signaling channel threat.
3. There exists the need for integrity mechanisms to prevent replicas at a single container from being updated arbitrarily by subjects.

In other words, we do not enforce any message filtering or mandatory security controls between objects at a single container, since doing so would require access mediation mechanisms to be imported into the individual backend DBMS's. This clearly goes against the original spirit and motivation of the replicated architecture. Messages sent from low replicas to other objects within a single container result in method invocations which are processed sequentially according to RPC semantics. Hence there is no need to maintain multiple versions of objects. Also, covert channel threats do not exist, as only subjects cleared to the level of a container can observe the results of local computations. Finally, the lack of mandatory controls within a container has to be balanced with adequate integrity mechanisms giving us the assurance that such replicas will not be updated by the local subjects at a container.

In contrast to the above, dealing with objects residing at different containers does require message filtering so as to prevent illegal information flows. If we review the different filtering cases in the message filtering algorithm (as shown in figure 2), we now see that case (4) which deals with messages sent from higher level to lower level objects, is degenerate. This is because messages sent downwards in the security lattice to enable read-down operations do not cross the boundary of a container, and as mentioned before, involve no filtering. Messages sent to higher and incomparable levels will still need to be filtered. In particular, when messages are sent to higher objects (residing at higher level containers), concurrency may again arise.

Having discussed the message filtering and security issues in the replicated architecture, we now turn our attention to the trusted computing base (TCB) in the architecture. As mentioned before, the TCB is hosted within the trusted front end (TFE). A design objective in any secure architecture is to minimize the number of trusted functions that need to be implemented within the TCB. This enables the TCB to have a small size, and thereby making its verification and validation easier. In light of this, is it possible to implement the various coordination and replica control algorithms while keeping the size of the TCB small? In later sections, we present replica control and coordination schemes that require minimal functionality from the TCB. To be more precise, the role of the TCB reduces basically to that of a router of messages from lower level to higher containers. In particular, the TCB requires no trusted (multilevel) subjects or data structures. All scheduling and coordination is achieved through single-level subjects at the backend databases. In other words, this portion of the front-end TCB could be implemented using a kernelized architecture.

3.2 Serial Correctness and Replica Control

Recall from our previous discussion that sending messages between objects at a single container involves no message filtering, while sending messages to objects across containers does call for filtering. When filtering is involved concurrency is once again inevitable and we have to ensure that the concurrent computations executing across the various containers preserve serial correctness. We now investigate the interplay between serial correctness, the various scheduling algorithms, and replica control.

We had earlier presented three constraints as sufficient conditions to guarantee serial correctness of concurrent computations. Correctness constraints 1 and 2 are required to govern the scheduling of concurrent computations while the third constraint governs how versions should be assigned to process read-down requests. Constraints 1 and 2 would now have to be interpreted for computations executing across containers. For example, when a computation c is started at a level l (container C_l), constraint 2 would now read: *All current non-ancestral as well as future executions of computations that have forktamps smaller than that of c, would have to be at containers for level l or higher.* Also, the fact that there are no trusted subjects in our implementation means that there will no central coordination of the computations executing across the various containers. Hence the implementation of the various scheduling algorithms would have to be inherently distributed. Finally, correctness-constraint 3 also has to reinterpreted for the replicated architecture as we no longer maintain versions of objects. The original requirement that a computation c reading down obtain the versions of lower level objects consistent with a sequential execution, now maps to the requirement that the various updates (also called update projections in the literature) producing these different versions be shipped and applied to $c's$ container before it starts executing. This last constraint thus has a direct implication on the replica control schemes that would be utilized for the architecture.

In order to reason about update projections and their effect on serial correctness, we introduce the notion of *r-transactions*. This is done only for ease of exposition. Our solutions do not impose or mandate any particular model of transactions. Transactions allow us to conveniently group sequences of updates, and in particular those that need to be incrementally propagated to higher containers. We use the prefix "r" which stands for "replicated", to distinguish this notion of transactions from others in the literature. We drop the prefix when it is clear from the context that we are referring to r-transactions.

In the object model of computing, every message in is received at an object and results in the invocation of a method defined in that object. We refer to such an object as the home object of the method. The subsequent activity (reads and updates) within the boundary of a home object can be modeled as belonging to a r-transaction. Every message in a message chain can be mapped to a corresponding transaction. This leads to a hierarchical (tree) model of transactions for a user session. We consider the root message as starting a root transaction. The root transaction in turn issues other transactions which we see as its descendants in the tree. Figure 6 illustrates the transaction tree for a computation tree.

A depth-first (left-to-right) traversal of a transaction tree starting with the root transaction, will give the sequence in which the transactions are issued and started within a user session. To illustrate how serial correctness is to be maintained within a session and in the context of the replicated architecture, we need to zoom in and take a magnified look at the transaction tree. This is because a transaction may make its partial results visible to other transactions at different containers. Consider any subtree in figure 6 such as the one rooted at transaction T_2. A child of T_2, such as T_3, is allowed to see (read down) only part of the updates made by T_2. To be more precise, it is only those updates made by T_2 up to the point T_3 was issued. The second child T_4 will be allowed to see all the updates seen by T_6, and in addition those made by T_2 between the interval that T_3 and T_4 were issued.

To model and visualize partial visibilities within transaction boundaries, we introduce *r-subtransactions* as finer units of transactions. The second and larger tree in figure 6 illustrates a subtransaction tree derived from the original transaction tree. A transaction such as T_2 is now chopped up into three subtransactions $t_{2,1}$, $t_{2,2}$, $t_{2,3}$. The subtransaction $t_{2,1}$ represents all the updates by T_2 until transaction T_3 was issued. Subtransaction $t_{2,2}$ similarly represents the updates between the interval that transactions T_3 and T_4 were issued. Finally, the subtransaction $t_{2,3}$ accounts for all the remaining activity in T_2 before it committed. A subtransaction is seen as having a relatively short lifetime, and is required to commit before any sibling subtransactions to the right, or child transactions (and implicitly subtransactions) are started. The operations issued by a subtransaction are said to be *atomic operations*. Such operations never cross the boundary of their relevant home object and cannot lead to the sending of further messages (or the issuing of transactions) to other objects. Serial correctness requires that an individual transaction, such as T_4 see all the updates of all subtransactions below its level that will be encountered in a depth-first search of the subtransaction tree starting with the root and ending in T_4. Thus for T_4 this will include subtransactions $t_{1,1}$, $t_{2,1}$, and $t_{2,2}$. The updates of all these subtransactions except $t_{2,2}$ would have to be seen by the left sibling of T_4, which is transaction T_3, and thus would have already been applied logically at the relevant containers before T_4 was issued.

We formally define these and other notions below:

Definition 3.1 *We consider a* subtransaction *to be a totally ordered set of atomic operations. We define a* transaction T_i *to be a partial order* $(s_i, <_{T_i})$ *such that:*

1. s_i *is a set of operations, and each operation may be a subtransaction or another transaction* $T_j = (s_j, <_{T_j})$.
2. *The relation* $<_{T_i}$ *orders at least all conflicting atomic operations in* s_i.

Definition 3.2 *We define the* replica-set *of a transaction* T_j *at level* j *to be the set of updates of subtransactions at or below level* j *that will be seen by* T_j *in a sequential execution (or depth-first search) of the tree.*

Definition 3.3 *We define the* propagation-list *of a transaction* T_j *to be those updates in the replica-set of* T_j *that have not been seen by* T_i*, where* T_i *was the last transaction that was issued before* T_j *in a sequential execution. These updates are those made by subtransactions at levels lower than* j*, and to the right of* T_i *and to the left of* T_j *(in the subtransaction tree).*

In figure 6, the replica-set of transaction T_4 will consist of the updates issued by subtransactions $t_{1,1}$, $t_{2,1}$, and $t_{2,2}$. The propagation-list of T_4 will consist of the updates issued by subtransaction $t_{2,2}$.

Now in the replicated architecture, the transactions in a subtransaction tree execute across containers. Whenever an object in a low container issues a write-up request, a message will be sent upwards in the lattice, and routed by the TFE to the appropriate high level container. Such a message will be received by an object in the higher level container and eventually result in the invocation of a method. Before this method can be invoked (i.e., before the corresponding transaction can be started), we need to do the following:

1. Determine if it is safe to begin execution of the transaction;
2. Make sure that the propagation-list of the transaction has been applied at the local container.

The first consideration above arises from the fact that the transactions (methods) generated by a session execute across the various containers in a distributed fashion, and this may lead to transactions at higher containers starting prematurely (when compared to centralized sequential execution). We thus require the start-up of transactions to be governed by some invariant. Once a transaction is allowed to start (i.e., doing so would not violate the invariant), the replica control scheme should ensure that the relevant propagation-list (set of update projections) is applied at the local container of the transaction.

Before concluding our discussion on serial correctness and replica control, we note that in the replicated architecture serial correctness alone is insufficient to guarantee the mutual consistency of replicated data. This is because serial correctness can be guaranteed by shipping update projections only to the containers which have forked transactions for a session. In other words, if a transaction was not forked for a level, the replicas at the container for the level could be out-of-date, and we would still not violate serial correctness. The scheduling algorithms that we present in the next section not only guarantee serial correctness, but in addition ensure that when a session terminates, all containers will be mutually consistent. When such consistency is guaranteed, we say that the algorithms preserve the *final-state equivalence* of all the containers.

4 Intra-session Scheduling

In this section we discuss how we can combine replica control to ensure final-state equivalence with scheduling strategies. As in the kernelized architecture, the conservative scheme involves less complexity and is thus easier to implement. Due to the lack of space we discuss only the aggressive scheduling scheme. We begin by clarifying some aspects related to the execution and failure semantics of transactions, as well as some of the necessary data structures to be maintained by individual containers.

A r-transaction, as described here, is characterized by the property of failure atomicity. Hence if any of the subtransactions of a transactions fails or aborts, we have to abort the entire corresponding transaction. This would also require that we undo the effects of any committed earlier subtransactions. To avoid this, and still guarantee failure atomicity, we allow a transaction to commit only if all its subtransactions commit. The updates of committed subtransactions are made permanent in the database only when the parent transaction commits. Also, we take the commit of the root transaction to imply that the entire session has committed.

To implement our scheduling strategies and replica control schemes, every container C_j at level j maintains the following data structures for an active user session:

Activation-queue$_j$:	this is a priority queue of transactions that is maintained according to the forkstamp;
Projection-queue$_j$:	a queue which stores update projections (propagation-lists) by their forkstamps;
Transaction-history$_i$:	this is a list maintained for each level $i < j$, and maintains for every transaction forked from level i, its id, forkstamp, status and other information.

When transactions start issuing other transactions at higher levels, the relevant propagation-lists (update projections) are incrementally shipped to higher containers and stored in their projection queues. When a scheduling scheme calls for a transaction to be started, it is dequeued from the local activation queue and the relevant update projections are applied to the container just before transaction starts.

4.1 Implementing Aggressive Scheduling

We now briefly discuss the implementation of the aggressive scheduling scheme. A transaction history (listed above) is required to be maintained at every container and keeps track of the forked transactions at dominated levels. It is important to note that this history itself is a replicated data structure and snapshot. The need for the maintenance of this history arises from the fact that a container cannot read-down information at lower level containers. Recall that the front-end in the replicated architecture sends messages only in an upwards direction in the lattice. Hence the relevant information has to be gathered with the help of snapshots maintained by constantly sending messages upwards in the lattice. It is the sending of such messages and the maintenance of snapshots such as transaction histories that add to the complexity of implementing the aggressive scheduling scheme.

The aggressive scheduling algorithm is governed by the following invariant:
Inv-aggressive-replicated: *A transaction is executing at a container at level l only if all non-ancestor transactions (in the corresponding transaction tree) with smaller fork stamps at containers for levels l or lower, have terminated.*

The description of the scheduling algorithms is similar to that of the kernelized architecture [17], with the difference that we now have to post update projections at the right time to the appropriate containers. A container always looks at its transaction histories for dominated levels to see if the start-up of the next transaction would violate the above invariant. The detailed algorithms are presented in figures 7,8,11, and 12. In these algorithms % is a delimiter for comments.

When a write-up message is issued and a transaction is forked at a level, the transaction-history at this level is updated (see the fork procedure in figure 7). We then check to see if the update projections from the parent issuing the write-up can be applied to the local container and also if the forked transaction can be allowed to start. If doing so would violate serial correctness, the update projections are queued in the local projection-queue and the forked transaction is queued in the local activation-queue. A queued transaction is later started or "woken up" by the termination of a running transaction. When a transaction terminates (see figure 9), its updates are posted to the local as well as higher containers. If serial correctness is not violated, relevant update projections from the projection-queue may also be applied to the local container. We then check to see if transactions at the local and higher containers can be started as a result of this most recent termination. To release or start queued transactions at higher levels, a WAKE-UP message is sent to the higher level containers through the trusted front end (TFE). It is important to note that a WAKE-UP message is sent to higher containers only if there exists queued transactions and their release would not violate serial correctness and the invariant. As such when a container receives a WAKE-UP message from a lower level, it knows that its activation queue is not empty and proceeds unconditionally to start the next transaction at the head of the activation queue (see procedure for WAKE-UP processing in figure 8). Before a transaction is actually started, the projection queue is examined and all entries with a forkstamp less than that of the transaction are emptied and applied to the local container (as shown in figure 10).

Proofs

For brevity, we omit the proofs to demonstrate that that our algorithms preserve serial correctness. The arguments are similar to those made for the aggressive scheme under the kernelized architecture [17]. However, the requirement for these algorithms to preserve final-state equivalence is unique to this architecture. We state and prove this as a theorem.

Theorem 4.1 *The aggressive scheme preserves final-state-equivalence.*

Proof:
By induction on the number of possible terminations, n, in a session.
Basis: Consider the basis with $n = 1$. In this case we have only one termination, that of the

root transaction. The procedure **term-rep-agg** in figure 9 processes terminate requests, and calls for the update projection of the root transaction to be posted to the local container as well as all higher containers. Each higher container, on receiving the projection, will find that there are no lower level transactions with smaller forkstamps than the terminated root, and apply the update projection from its queue. Each higher container will thus be brought up-to-date with the updates of the root transaction and thus preserving final-state equivalence.
Induction Step: For the induction hypothesis, assume that when n is equal to m, final-state equivalence is guaranteed. For the induction step, let $n = m+1$. In other words, there are $m+1$ possible terminations, and given that the first m terminations preserve final-state equivalence, we have to show that the $m + 1^{th}$ termination preserves final-state equivalence. Consider the transaction t_{m+1} at container C_{m+1} that causes the $m + 1^{th}$ termination. By the induction hypothesis, we are guaranteed that C_{m+1} will receive all update projections from dominated containers. Some of these projections would be applied to the contents of C_{m+1} as soon as they are received, while others will be queued in the projection queue (as shown in procedure **post-update-rep-agg** of figure 11). When t_{m+1} starts, all the queued update projections originating from lower level transactions with smaller forkstamps than t_{m+1} would also be applied to C_{m+1}. Finally when t_{m+1} terminates all remaining update projections will be emptied and applied to C_{m+1} along with its last-updates. This guarantees the mutual consistency of container C_{m+1} will all lower level containers. It now remains to show that mutual consistency is preserved with containers higher than C_{m+1}. This follows from the fact when t_{m+1} terminates, all its update projection would be sent to all higher containers where they would be subsequently applied. Thus final-state equivalence is preserved across all $m + 1$ terminations, and this concludes the proof. □

5 Inter-session Synchronization

Having discussed various intra-scheduling schemes, we now turn our attention to inter-session concurrency control in the the replicated architecture. We do not address the issue of concurrency control between sessions at a single container, rather focus on multiple containers. Every container is assumed to provide some local concurrency control.

We assume the following:

- Every container C_j at level j, uses some local concurrency control scheme L_j.
- All containers share a system-low real-time clock. This is a reasonable assumption since the replicated architecture is not for a distributed system, but rather to be implemented on a single (central) machine. The value read from this clock is used to maintain a global serial order for sessions and transactions.

We discuss three approaches to inter-session synchronization and concurrency control that provide increasing degrees of concurrency across user sessions. To elaborate, consider the four sessions S_a, S_b, S_c, and S_d as shown in figure 13(a). Sessions S_a and S_b originate at container C_U at level U, while S_c and S_d originate at containers C_C and C_S at levels C and S respectively. The different transactions generated by these sessions are shown in the figure. For example, session S_a generates transactions T_{a1} at level U, T_{a2} at level C, and T_{a3} at level S. Figures 13 (a), 13 (b), and 13 (c) depict the histories that could be generated by the three inter-session schemes, at the various containers.

In the first scheme, sessions are serialized in a global order that is equivalent to the serialization events of the sessions. If L_j is based on two phase locking, we can use the lock point, which is the last lock step of the root transaction of the session, as its serialization event. If the local concurrency control scheme, L_j is based on timestamping, the timestamp assigned to S_j or the root transaction can be used for the serialization event. In the second approach, this serial order can be successively redefined to interleave incoming newer sessions without affecting the mutual consistency or correctness of the replicas and updates. In the third approach we relax the serial order for the sessions, and instead serialize transactions on a level-by-level basis.

Protocol 1: Globally serial sessions

When a session S_j starts at a container j (i.e., the root transaction executes C_j), the following protocol is observed:

1. S_j makes its resource requests to the local concurrency controller, and its transactions compete with other local sessions that start at C_j.
2. When S_j reaches its serialization event as governed by L_j, the real-time clock is read and its value used to form a *serial-stamp* for S_j.
3. The serial-stamp of S_j is broadcast to all higher level containers.
4. When S_j commits, a *commit-session* message is broadcast to all higher containers. This message may be piggy-backed with the *commit-transaction* message from the root transaction of S_j.

On receiving the serial-stamp from a container at a lower level, a container, C_k at level k, observes the following rules:

5. All local sessions originating at C_k, and having a smaller serial-stamp than that of S_j, are allowed to commit according to their serial-stamps, and subsequently propagate their updates to containers at levels higher than k.
6. The updates and transactions of S_j are allowed to proceed.
7. All local sessions at C_k having a greater serial-stamp than S_j are allowed to commit only after the *commit-session* notification of S_j is received, and its updates applied as in step 2 above, to C_k.

Several optimizations and variations on the above protocol are possible. It is obvious that the protocol provides minimum concurrency between sessions. In particular, the scheme offers very poor performance if transactions are of long durations. To elaborate, consider what happens if session S_b has sent its serial-stamp to container C but does not commit for a long time. If timestamping is used for the serialization events of sessions at container C, a local session S_c starting at container C_C after the serial-stamp of S_a has been received, will be assigned a greater serial-stamp. Hence, S_c will not be allowed to commit until S_b sends its *commit-transaction* message. The decrease in such concurrency is directly proportional to the size of the window between the serialization and commit events of session S_b.

We can easily improve the performance of the above scheme if S_c were allowed to go ahead and commit even if the *commit-session* message has not been received from S_b. This is possible if S_b has not updated the container C_C at level C so far. We can then re-assign to S_c an earlier serial-stamp than that of S_b. Figure 13(b) shows a possible history at the various containers with protocol 1, and figure 13(c) shows how the updates of session S_c can be placed ahead of S_b at container C by giving S_c an earlier serial-stamp than S_b. It is important to note that the relative order between the sessions S_a and S_b is still maintained, but only that S_c is now allowed to come between them. This idea is summarized in protocol 2 below.

Protocol 2: Globally serial sessions with successively redefinable serial-orders

Steps 1 through 6 of Protocol 1 still apply to Protocol 2, but step 7 is modified as below.
When a container C_k receives the serial-stamp from a session S_j at a lower container C_j, the following rules are followed:

7'. If there exists a session S_k that has the smallest serial-stamp among the sessions at C_k that have reached their serialization events but not yet committed, and such that S_k has a serial-stamp greater than S_j, then do:

   (a) If session S_j has not yet updated C_k, then reassign a serial-stamp to S_k that is smaller than the stamp of S_j.

   (b) Broadcast this new serial-stamp to all higher containers.

   (c) Allow S_k to update C_k and propagate its updates to higher containers.

The ability of protocols 1 and 2 above, to ensure the mutual consistency of the replicas at the various containers, can be attributed to the way updates are processed. To be more specific, the updates represented in the propagation-lists sent by various sessions, are processed at every container in strict serial-stamp order. A single serial-stamp is associated with the entire set of transactions (updates) that belong to a session.[3] In other words, a session is the basic unit of concurrency for interleaving updates from multiple sessions. To put it another way, the histories of the updates generated by protocols 1 and 2, guarantee that the individual transactions of two sessions, where each session starts at a different container, cannot be interleaved with each other in any of these histories.

A further improvement to protocol 2 and protocol 1 which we might call protocol 3, can be achieved if the global serial order that is maintained for sessions, is relaxed. Transactions are now serialized in some order on a level-by-level basis. This allows us to exploit more fine-grained concurrency within the structure of a session. The unit of concurrency now is no longer a session, but rather of finer granularity, and thus a transaction. Of course, the key here is exploit such fine-grained concurrency without compromising the mutual consistency of the replicas. The intuition behind this approach is illustrated in figure 13(d). Thus we see that the transactions at level U, namely $T_{a,1}$ and $T_{b,1}$ are serialized in the same order at all the containers. However, transactions at level C, namely $T_{a,2}$, $T_{c,2}$, $T_{b,2}$ are serialized in a different order. In particular, the updates from session S_b now come before sessions S_a and S_c. Protocol 2 can easily be modified so that the updates at each level are serialized independently, and made known to the higher containers. Unlike protocols 1 and 2, level-initiator transactions now have to compete with other

[3] We assume that such associations are kept in some data structure. We also assume that a transaction such as T_{c2} in figure 13(c), running at the container C_C, cannot update the local replicas of data stored at the lower container C_U. Protocols 1 and 2 can guarantee mutual consistency only to the extent that integrity safeguards are available to prevent such events.

transactions at the various containers to access data. When an individual transaction reaches it serialization event, the real-time clock is read to form a transaction-serial-stamp, which is subsequently broadcast to higher containers. Mutual consistency of the replicas is achieved by ensuring that updates in the propagation-lists are applied in strict transaction-serial-stamp order.

We now briefly discuss the correctness of the above protocols. Space constraints prevent us from giving a full-blown formal proof here. A well known correctness criterion for replicated data is *one-copy serializability* [2]. Protocols 1 and 2 guarantee what one might call *one-copy* **session** *serializability*. This gives the illusion that the sessions originating at the different containers execute serially on a one-copy, non-replicated, database. The interactions between transactions as governed by one-copy session serializability is much more restrictive in terms of concurrency and interleaving than one-copy serializability, but implicitly guarantees the latter. The final variation, ie., protocol 3, is less restrictive than and does not guarantee one-copy session serializability, but maintains one-copy serializability.

6 Summary and Conclusions

In this paper, we have addressed the issue of replica control for object-oriented databases. The elaboration of the message filter model for the replicated architecture required that we handle replica control for updates and computations generated within a session, as well as synchronization for multiple user sessions. Collectively, object-orientation and the support of write-up actions have impacted the solutions presented in this paper. These solutions thus differ from others presented for the replicated architecture within the context of traditional (relational) database systems.

The solutions presented here increase the commercial viability of the replicated architecture as well as object-oriented databases, for applications and environments that require multilevel security. The approach taken here requires minimum functionality from the TCB (that is hosted within the TFE) and more significantly, requires no multilevel (trusted) subjects. The potential for covert channels to be exploited within the TFE is thus eliminated.

There exists several avenues for future work. Inter-session synchronization and concurrency control warrant further investigation. In particular, it would be interesting to look at type-specific and semantic concurrency control across user sessions in the multilevel context. We would also like to investigate if it is possible to maintain atomicity of an entire session without violating confidentiality? That is, all of the component transactions in a session commit or abort without any impact on security (confidentiality). As observed by Mathur and Keefe in [10], atomicity and security seem to be conflicting requirements. If a session has component transactions at many levels, we cannot guarantee atomicity without introducing covert channels. At best we can only hope to reduce the bandwidth of such channels. Perhaps the approach would be to build a model that support write-up actions and at the same time minimizes such channels.

References

[1] P. Ammann and S. Jajodia. Planar lattice security structures for multi-level replicated databases. *Proc. of the Seventh IFIP 11.3 Workshop on Database Security*, Vancouver, Huntsville, Alabama, September 1993.

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

[3] B. Blaustein, S. Jajodia, C.D. McCollum, and L. Notargiacomo. A model of atomicity for multilevel transactions. *Proc. of the 1993 IEEE Symposium on Security and Privacy*, pp. 120–134, May 1993.

[4] O. Costich. Transaction processing using an untrusted scheduler in a multilevel database with replicated architecture, *Database Security V, Status and Prospects*, C.E Landwehr and S. Jajodia (Editors), Elsevier Science Publishers B.V. (North-Holland), Amsterdam, 1992.

[5] O. Costich and J. McDermott. A multilevel transaction problem for multilevel secure database systems and its solution for the replicated architecture. *Proc. of the 1992 IEEE Symposium on Security and Privacy*, pp. 192–203, May 1992.

[6] S. Jajodia and B. Kogan. Integrating an object-oriented data model with multi-level security. *Proc. of the 1990 IEEE Symposium on Security and Privacy*, pp. 76–85, May 1990.

[7] Sushil Jajodia and Boris Kogan, "Transaction processing in multilevel-secure databases using replicated architecture." *Proc. IEEE Symposium on Security and Privacy*, Oakland, California, May 1990, pages 360–368.

[8] T.F. Keefe and W.T. Tsai. Prototyping the SODA security model. *Proc. 3rd IFIP WG 11.3 Workshop on Database Security*, September 1989.

[9] T.F. Keefe, W.T. Tsai, and M.B. Thuraisingham. A multilevel security model for object-oriented systems. *Proc. 11th National Computer Security Conference*, pp. 1–9, October 1988.

[10] A.G. Mathur and T.F. Keefe. The concurrency control and recovery problem for multilevel update transactions in MLS systems. *To appear in the Proc. of the Computer Security Foundations Workshop*, Franconia, New Hampshire, 1993.

[11] J. McDermott, S. Jajodia, and R. Sandhu. A single-level scheduler for the replicated architecture for multilevel-secure databases. *Proc. of the Seventh Annual Computer Security Applications Conference*, San Antonio, TX, 1991.

[12] J.K. Millen and T.F. Lunt. Security for object-oriented database systems. In *Proc. of the 1992 IEEE Symposium on Security and Privacy*, pp 260-272, May 1992.

[13] M. Morgenstern A security model for multilevel objects with bidirectional relationships. *Database Security IV, Status and Prospects*, S. Jajodia and C.E Landwehr (Editors), Elsevier Science Publishers B.V. (North-Holland)

[14] R.S. Sandhu, R. Thomas, and S. Jajodia. A Secure Kernelized Architecture for Multilevel Object-Oriented Databases. *Proc. of the IEEE Computer Security Foundations Workshop IV*, pp. 139-152, June 1991.

[15] R.S. Sandhu, R. Thomas, and S. Jajodia. Supporting timing-channel free computations in multilevel secure object-oriented databases. *Proc. of the IFIP 11.3 Workshop on Database Security,* Sheperdstown, West Virginia, November 1991.

[16] R.K. Thomas and R.S. Sandhu. Implementing the message filter object-oriented security model without trusted subjects. *Proc. of the IFIP 11.3 Workshop on Database Security*, Vancouver, Canada, August 1992.

[17] R.K. Thomas and R.S. Sandhu. A Kernelized Architecture for Multilevel Secure Object-oriented Databases Supporting Write-up. Journal of Computer Security, Volume 2, No. 3, IOS Press, Netherlands, 1994.

[18] M.B. Thuraisingham. A multilevel secure object-oriented data model. *Proc. 12th National Computer Security Conference*, pp. 579–590, October 1989.

[19] Multilevel data management security. Committee on Multilevel Data Management Security, Air Force Studies Board, National Research Council, Washington, D.C., 1983.

Figure 3: A tree of concurrent computations with forkstamps

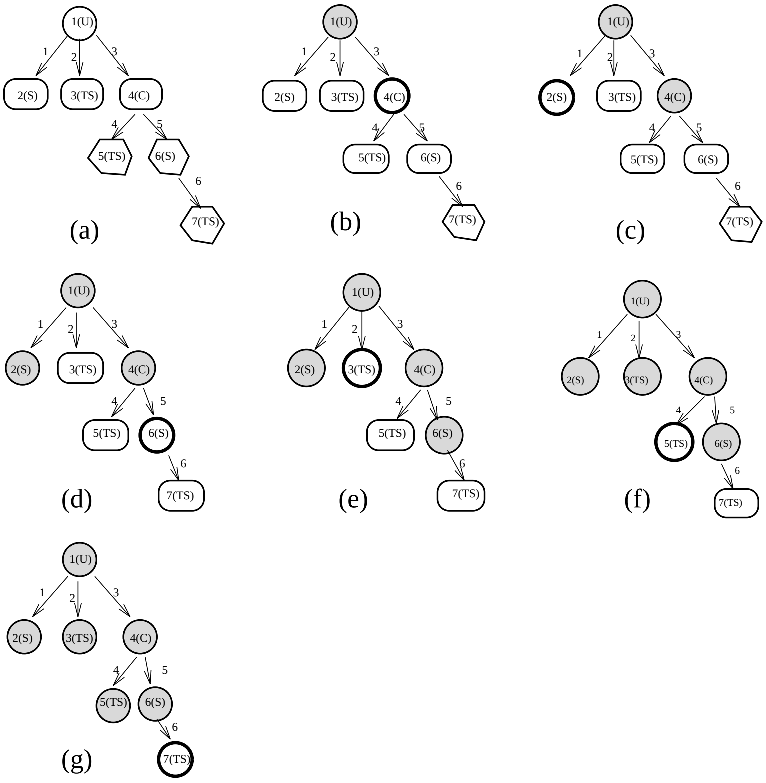


Figure 4: Conservative Scheduling

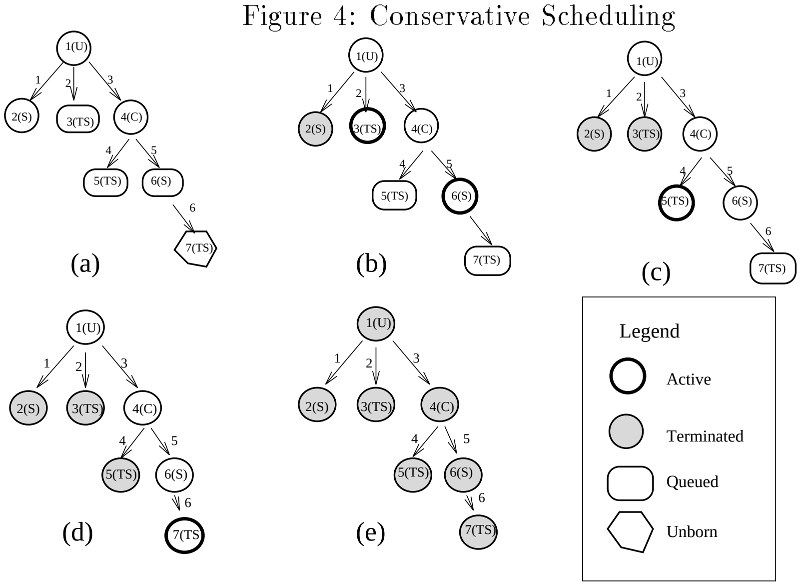
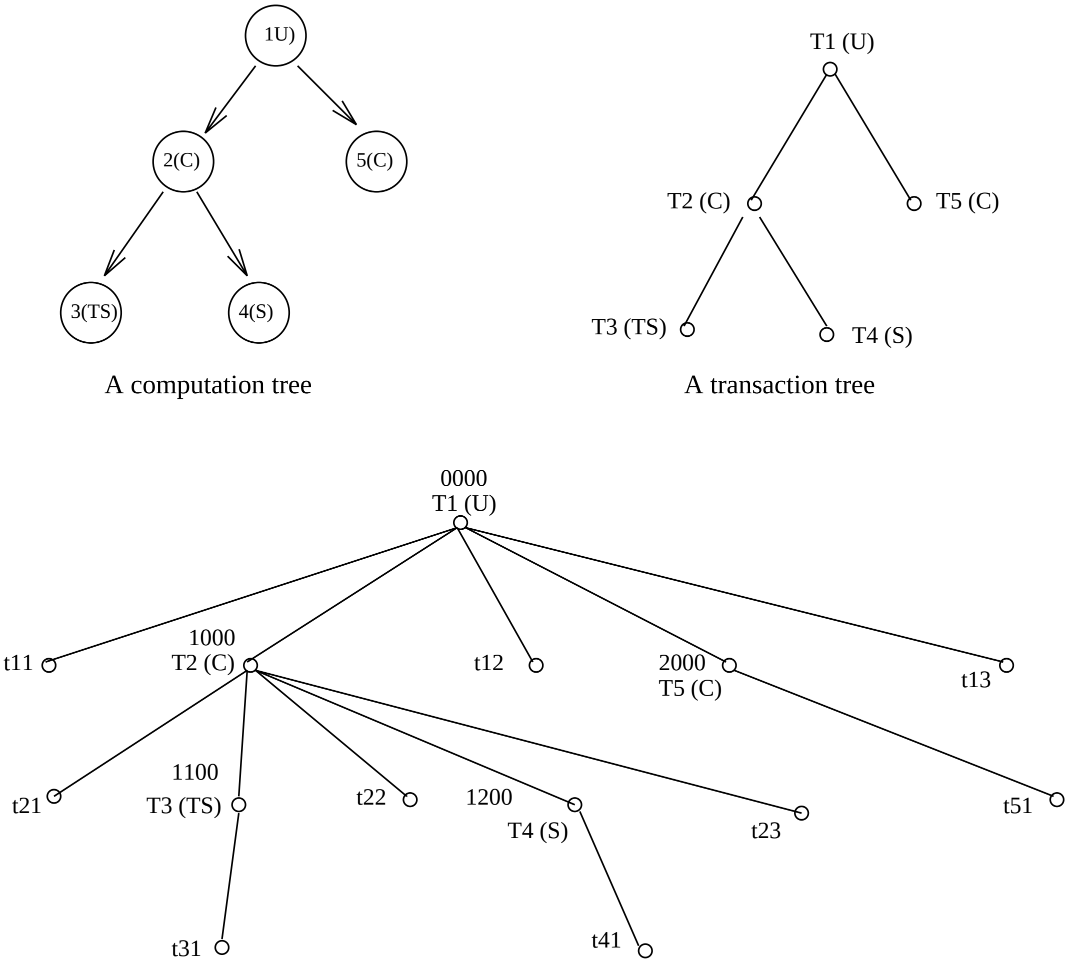


Figure 5: Aggressive Scheduling



The corresponding subtransaction tree

Figure 6: A transaction tree and its subtransaction mapping

```
Procedure fork-rep-agg(level-parent, level-create, forkstamp, update-projection)
{
%Let level-create be the level of the local transaction and container
Create a new transaction tt at level-create with identifier id;

%Initialize variables for tt
      tt.id ← id;
      tt.level-parent ← level-parent;
      tt.level-create ← level-create;
      tt.forkstamp ← forkstamp;
      tt.status ← 'non-terminated';

%Update local transaction history
append(transaction-history_{level-parent} , tt);

%See if the update projection from the parent can be applied at the local container
and if tt can be started immediately
If ∀ l ≤ level-create, ¬∃ any transaction c ∈ transaction-history_l:
      (c.level-create ≤ level-create ∧ c.forkstamp < tt.forkstamp
      ∧ c.status = 'non-terminated')
      then
         apply update-projection to local container;
         start-rep(tt);
      else
         %This is a priority queue maintained in forkstamp order
         enqueue(projection-queue, update-projection, forkstamp);
         %This is also a priority queue of transactions waiting to be activated
         enqueue(activation-queue, tt);
end-if
}
end procedure fork-rep-agg;
```

Figure 7: Processing **fork** requests under aggressive scheduling

```
Procedure wake-up-rep-agg
{
%Let tt be the transaction at the head of the local activation-queue
dequeue(activation-queue, tt);
start-rep(tt);
}
end procedure wake-up-rep-agg;
```

Figure 8: Processing **wake-up** requests under aggressive scheduling

```
Procedure term-rep-agg(level-term, last-update, term-forkstamp, last-forkstamp)
{
%Record the termination of transaction tt at level-term
For each level l < level-term do
   If (pp ∈ transaction-history_l ∧ pp.forkstamp = tt.forkstamp)
      then tt.status ← 'terminated'; End-If End-For

%Update local container with the last set of updates issued by tt
apply the updates in last-update to local container;
%Post these updates to higher levels
term-flag ← 'true';
For each level > level-term do
   post-update-rep-agg(level, last-update, increment(last-forkstamp), term-flag, tt);
End-For

%See if the update projections for last-updates from lower levels can be applied
quit-flag ← 'false';
For all levels l ≤ level-term do
   If ∃ any transaction q ∈ transaction-history_l :(q.status = 'not-terminated' ∧
   q.level-create ≤ level-term) then quit-flag ← 'true';exit for;end-If; end-For;

If quit-flag = 'false' then
   Repeat
      dequeue (projection-queue, update-projection);
      apply update-projection to local container;
   Until projection-queue = empty; exit procedure; End-If

%Check if a queued transaction at level level-term can be started
%Let mm be the transaction at the head of the activation queue
If the activation-queue is not empty
then If ∀ l, l ≤ level-term, ¬∃ any transaction c ∈ transaction-history_l:
        (c.forkstamp < mm.forkstamp ∧ c.status = 'not-terminated')
        then dequeue(activation-queue, mm); start-rep(mm); End-If End-If

%Check if a transaction at levels ≥ level-term can be started
For all levels l ≤ level-term do
   If ∃ a transaction c ∈ transaction-history_l with c.level-create > level-term ∧
   c.forkstamp > tt.forkstamp: ¬∃ any non-ancestor transaction k
   with (level(k) ≤ level(c) ∧ k.forkstamp < c.forkstamp ∧
   transaction-history_level(k).k.status = 'not-terminated')
   % We checked to see if c was not preceded by a lower-level active or queued
   % non-ancestor transaction in any of the transaction-histories searched
   then Send a WAKE-UP message to the container at level(c); End-If End-For
} end procedure term-rep-agg;
```

Figure 9: Processing **terminate** requests under aggressive scheduling

Procedure start-rep(*tt*)
{
%*Let tt be the transaction to be started*
counter $\leftarrow$ 1;
Repeat
%*Treat the projection-queue at the level of tt as a list and examine it element-wise*
  Read (projection-queue[counter], *pp*);
  If *pp*.forkstamp $\leq$ *tt*.forkstamp
    then
      apply *pp* to local container;
      delete (projection-queue, *pp*);
  End-If
  counter $\leftarrow$ counter + 1;
Until counter = length-of(projection-queue);

%*Begin executing tt*
execute(tt);
}
end procedure start-rep;

Figure 10: Updating the local container before starting a transaction

Procedure post-update-rep-agg(*local-level, update-projection, forkstamp, term-flag, tt*)
{
%*See if the posted update can be applied*
If $\forall\ l \leq$ local-level, $\neg\exists$ any transaction $c \in$ transaction-history$_l$:
    (c.level-create $\leq$ local-level $\wedge$ c.forkstamp $\leq$ tt.forkstamp
    $\wedge$ c.status = 'non-terminated')
    then
      apply update-projection to local container;
    else
      %*This is a priority queue maintained in forkstamp order*
      enqueue(projection-queue, update-projection, forkstamp);
end-if

If term-flag = 'true'
then %*Record the termination of transaction tt*
transaction-history$_{tt.level-parent}$.tt.status $\leftarrow$ 'terminated';
end-if
}
end procedure post-update-rep-agg;

Figure 11: Processing posted updates

```
Procedure record-new-transaction(transaction, level-parent)
{
% Update local transaction history for level level-parent
append(transaction-history_{level-parent} , transaction);
}
end procedure record-transaction;
```

Figure 12: Recording the fork of computations at lower levels

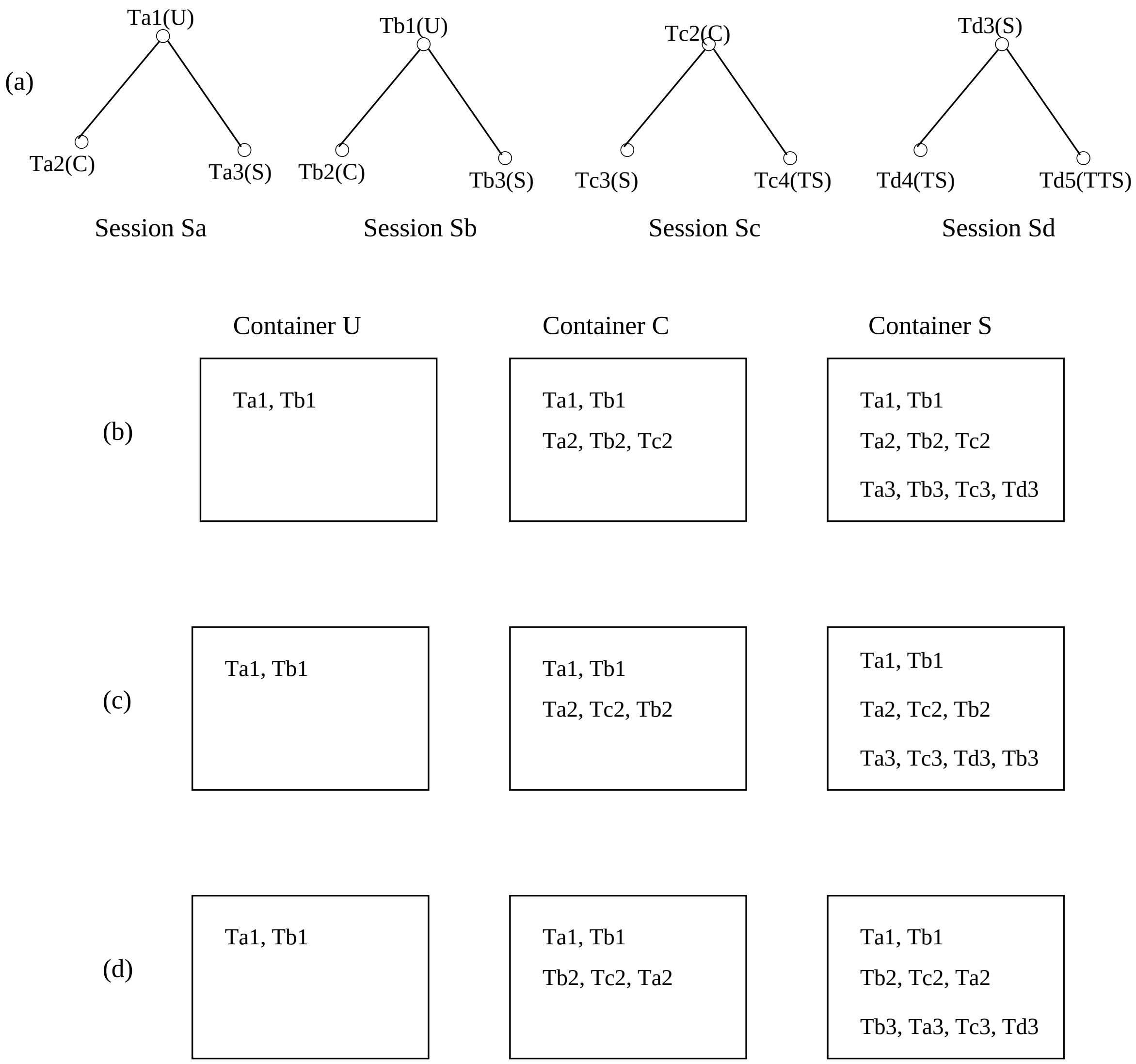


Figure 13: Illustrating histories generated with various inter-session synchronization schemes