

## Chapter 8

# Inter-session Synchronization

So far we have looked at the issues of concurrency and scheduling within a single user session. We now focus on how objects can be shared across multiple concurrent user sessions. In the database literature, schemes to achieve this generally fall under the category of concurrency control and transaction management. Our purpose here is not to discuss a comprehensive concurrency control scheme, but only to give a basic usable secure solution to object sharing across user sessions. Discussion of a comprehensive transaction model for multilevel systems is beyond the scope of this dissertation. We address in detail inter-session concurrency for the kernelized and replicated architectures, and briefly discuss how some of these algorithms can be utilized in the trusted subject architecture in a secure (confidentiality preserving) manner.

### 8.1 Inter-session Concurrency for the Kernelized Architecture

Our approach to object sharing is based on a checkin/checkout access data model [LP83]. There exists a public single-version database from which user sessions check-out objects as needed. The objects are checked out into local workspaces (private databases) of individual user sessions. When all activity associated with an object has ceased, the object is checked back into the public database. Due to concurrent

activity in a user session, computations within a user session may view several versions of the same object. However, visibility across user sessions is limited to the public database which maintains only the latest version of every object.

### 8.1.1 Multilevel Checkin/Checkout of Objects

One of the considerations in designing an object sharing and transaction management scheme is that of formulating and maintaining some notion of inter-session correctness. Conventional database management schemes primarily support transactions that are short-lived and competitive. Interactions and visibility across such transactions are curtailed and the correctness of concurrent transactions is governed by serializability. However, if we examine the applications that are impelling the development of object-oriented database technology, we find that they are characterized by requirements that differ from those utilizing conventional databases. These applications are generally found in environments that call for cooperative work, such as computer-aided design. In such environments, serializability as a correctness criterion needs to be relaxed, and interactions between concurrent transactions have to be promoted rather than curtailed. In light of this, in our further discussions we do not assume that serializability is enforced.

We now discuss how a checkin/checkout scheme can be coupled with the model of r-transactions presented in the last chapter. Our choice of a checkin/checkout scheme as opposed to other conventional schemes directly follows from the above assumption that transactions are cooperative in nature. We provide the following commands to implement a checkin/checkout scheme:

1. **Public-checkout(R/W):** Checks out an object from the public database.
2. **Public-checkin:** Checks in an object into the public database.

3. **Local-checkout(R/W):** Checks out an object from the local workspace of a user session.
4. **Local-checkin:** Checks in an object into a session's local workspace.

The local commands differ from the public ones as their effects are internal to a session, and thus do not affect the visibility or availability of objects to other concurrent user sessions. A checkout operation can be requested in read (R) or write (W) mode. A checkout in W mode is permitted only if the computation generating the requesting subtransaction and the object to be checked out are at the same level. On the other hand, whenever a computation (or more precisely a subtransaction) requests a checkout of a lower level object, the request is granted in read (R) mode only. Multiple subtransactions may checkout the same object (or version of an object) in R mode. However, if a subtransaction first checks out a version in W mode, then no subtransaction at the same or higher levels may check out the version in either R or W mode (as checkouts in R mode conflict with those in W mode). On the other hand, when a high subtransaction first checks out a lower level object in R mode, it is understood that the high subtransaction is given a read-only snapshot of the object at the time. A low subtransaction will be allowed to checkout the same object in W mode before the high subtransaction has checked it in. Thus the checkout of low subtransactions are always given priority. While the checkin operation is necessary for any object checked out in W mode, it is redundant and can be ignored for any object checked out in R mode.

If a requested object has not been checked out by a user session so far, a public checkout request is issued. If however the object had been previously checked out from the public database by the session, it is simply checked out from the session's local workspace. In either case, when the subtransaction terminates, the object is checked

back into the local workspace of the session. A final version of every object that has been updated will eventually be checked back into the public database as explained in the remainder of this section.

When a subtransaction succeeds in checking out a version from a session's local workspace, it is guaranteed that the state of the object so read will never be invalidated in the future. This is because once a version becomes available for checkout in R mode to higher level subtransactions within the same session, we are guaranteed that such a version will never be updated again. To put it another way in transaction processing terminology, a checkout in R mode will always read *committed* values of objects. The implication of this is basically that high level subtransactions cannot develop *abort dependencies* on lower level ones, internally within a session. If such dependencies were possible, then a high level subtransaction would have to abort if a low level subtransaction from which it read, aborts.

As mentioned before, serializability is *not* enforced across user sessions. However, a subtransaction in a session will see only committed states of objects that are updated by other sessions. This is ensured by requiring all public checkin operations from a session to be deferred until the root computation in the session terminates. We consider a session to be logically and semantically committed at the point the root computation terminates normally (i.e., not due to an error or exception). This guarantees that no *abort dependencies* will develop across user sessions. The absence of such dependencies ensures that a session A would not have to abort because another session B from which it read, aborts.

### 8.1.2 Checkin/Checkout Variations

We now give two variations of a checkin/checkout scheme. They differ basically in how and when objects checked out from the public database are checked back

into the public database, for access by other user sessions. They thus offer different granularities of interactions across user sessions. These variations can be applied to both conservative and aggressive intra-session scheduling strategies. In a *level-by-level checkin/checkout* variation, an object that is updated at a level  $l$  by a session, is made visible to another session only when all updates to all objects at level  $l$ , by the session, have been completed. In the second *computation-by-computation checkin/checkout* variation, an object is made visible (checked in) as soon as all the subtransactions associated with a computation have terminated.

#### *Level-by-level Checkin/Checkout Schemes*

The basic idea is to checkin (commit) objects to the public database, one level at a time. Thus conceptually, we can implement this with processing and propagation of a *level-has-committed* message upwards in the security lattice. With a conservative scheduling scheme, the *level-has-committed* message can be piggybacked onto the *wake-up* message. On the other hand, with aggressive scheduling, the *level-has-committed* message has to be explicitly propagated. We describe both variations below.

The level-by-level checkin/checkout scheme can be combined with the conservative scheduling strategy as follows:

1. A subtransaction checks out the required objects from either its session's local workspace, or from the public database (the latter if any required object has not been previously checked out by the session).
2. When a subtransaction terminates all checked out objects are checked back in to the session's local workspace.
3. If a *wake-up/level-has-committed* message has been received from all immediate

lower levels, and all computations and associated subtransactions at a level say  $l$ , have terminated (i.e., when the level manager at  $l$  finds its local queue to be empty), then the level manager at  $l$  checks in the latest versions of all updated objects into the public database. This is followed by step 4.

4. After all updated objects at level  $l$  have been checked into the public database, a *wake-up/level-has-committed* message is sent to all immediate higher levels by the local level manager at level  $l$ .

In the conservative scheme above, the receipt of a *wake-up/level-has-committed* message from a lower level is a guarantee that no fork requests will be forthcoming from the lower level. However, in an aggressive level-by-level scheme, this is no longer true. In fact, a level may receive many wake-up messages from a lower level. A *level-has-committed* message can thus no longer be piggybacked onto a *wake-up* message, but rather has to be explicitly propagated, starting with the termination of the root computation. In addition to steps (1) and (2) given above for the conservative scheme, we require the following additional steps to achieve this:

- 3'. When the root computation terminates, we check in all updated objects to the public database and send a *level-has-committed* message to all immediate higher levels.
- 4'. When a *level-has-committed* message has been received from all immediate lower levels, and the local level manager finds its queue to be empty, it checks in all updated objects to the public database. The level manager then propagates the *level-has-committed* message to its immediate higher levels.

### *Computation-by-computation Checkin/Checkout Schemes*

A computation-by-computation checkin/checkout scheme releases (checks in) objects to the public database much earlier in comparison to the level-by-level scheme. Thus on the average, the availability of objects for checkout, across user sessions, is increased as waiting times are reduced. The scheme can be combined again with both conservative and aggressive scheduling. In either case the basic idea is the same. All objects checked out by a computation, or more precisely the set of subtransactions generated by the computation, are checked back into the *public* database as soon as the computation terminates. Contrast this with the level-by-level checkin scheme where we have to wait for all computations at the associated level to terminate. In other words, when the last subtransaction associated with a computation terminates, all checked out objects are checked back into the public database. However for objects checked out in W mode, only the latest version of every object is checked back in. It is obvious that this variation can result in objects being shuffled back and forth from the public database with much greater frequency than the level-by-level scheme.

## **8.2 Inter-session Concurrency for the Trusted Subject Architecture**

The level-by-level and computation-by-computation checkin/checkout schemes can also be implemented for the trusted subject architecture. However, we now have to demonstrate that the concurrency control schemes are secure in that they cannot be exploited for covert channels. In particular, we need assurance that a trusted multilevel subject such as the session manager cannot introduce any interference.

The arguments for a confidentiality proof for our checkin/checkout scheme can be built from the following:

- The checkout requests of low transactions never conflict with higher requests.

This means that the checkout requests of low transactions are never delayed or rejected due to the existence of higher level transactions. Intuitively, we can now establish noninterference by purging the requests of high level transactions and showing that they leave the order and timing of low level requests unaffected within a session, as well as across sessions.

### 8.3 Inter-session Concurrency for the Replicated Architecture

Having discussed the kernelized and replicated architectures, we now turn our attention to inter-session concurrency control for 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 8.1(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 8.1 (a), 8.1 (b), and 8.1 (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  $S_j$ .

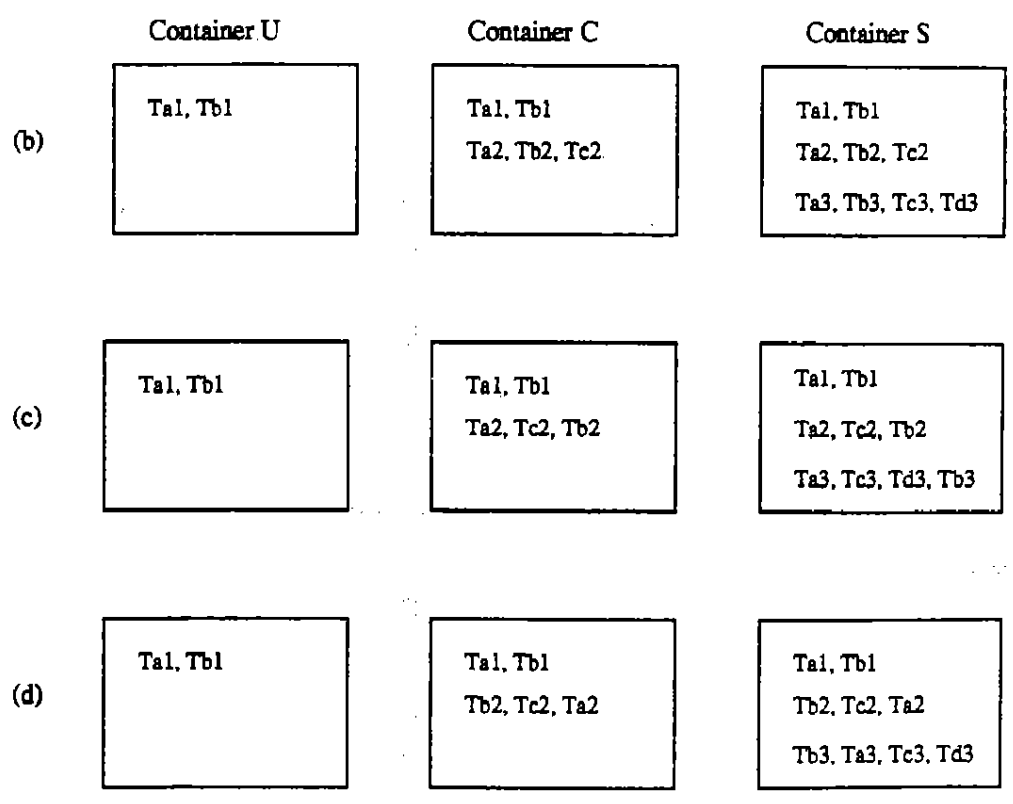
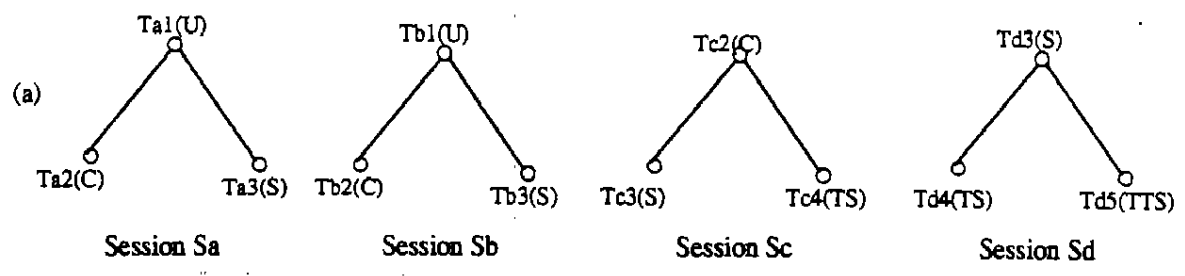


Figure 8.1: Illustrating histories generated with various inter-session synchronization schemes

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 8.1(b) shows a possible history at the various containers with protocol 1, and figure 8.1(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.<sup>1</sup> 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 8.1(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 transactions at the various containers to access data. When an individual transaction reaches its 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.

---

<sup>1</sup>We assume that such associations are kept in some data structure. We also assume that a transaction such as  $T_{c,2}$  in figure 8.1(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.

We now briefly discuss the correctness of the above protocols. A well known correctness criterion for replicated data is *one-copy serializability* [BHG87]. 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 the others and does not guarantee one-copy session serializability, but instead maintains one-copy serializability.