Concurrency Control Service 7

7.1 Service Description

The purpose of the Concurrency Control Service is to mediate concurrent access to an object such that the consistency of the object is not compromised when accessed by concurrently executing computations.

The Concurrency Control Service consists of multiple interfaces that support both transactional and non-transactional modes of operation. The user of the Concurrency Control Service can choose to acquire locks in one of two ways:

- On behalf of a transaction (transactional mode.) The Transaction Service drives the release of locks as the transaction commits or aborts.
- By acquiring locks on behalf of the current thread (that must be executing outside the scope of a transaction). In this non-transactional mode, the responsibility for dropping locks at the appropriate time lies with the user of the Concurrency Control Service.

The Concurrency Control Service ensures that transactional and non-transactional clients are serialized. Hence a non-transactional client that attempts to acquire a lock (in a conflicting mode) on an object that is locked by a transactional client will block until the transactional client drops the lock.

7.1.1 Basic Concepts of Concurrency Control

Clients and Resources

The Concurrency Control Service enables multiple *clients* to coordinate their access to shared *resources*. Coordinating access to a resource means that when multiple, concurrent clients access a single resource, any conflicting actions by the clients are reconciled so that the resource remains in a consistent state.

The Concurrency Control Service does not define what a resource is. It is up to the clients

of the Concurrency Control Service to define resources and to properly identify potentially conflicting uses of those resources. In a typical use, an object would be a resource, and the object implementation would use the concurrency control service to coordinate concurrent access to the object by multiple clients.

Transactions as Clients

The Concurrency Control Service differentiates between two types of client: a transactional client and a non-transactional client. Conflicting access by clients of different types is managed by the Concurrency Control Service, thereby ensuring that clients always see the resource in a consistent state.

The Concurrency Control Service does not define what a transaction is. Transactions are defined by the Transaction Service. The Concurrency Control Service is designed to be used with the Transaction Service to coordinate the activities of concurrent transactions.

The Transaction Service supports two modes of operation: implicit and explicit. When operating in the implicit mode, a transaction is implicitly associated with the current thread of control. When executing in the explicit mode, a transaction is specified explicitly by the reference to the coordinator that manages the current transaction. To simplify the model of locking supported by the Concurrency Control Service when a transactional client is operating in the implicit transaction mode, transactional clients are limited to a single thread per transaction (nested transactions can be used when parallelism is necessary) and that thread can be executing on behalf of at most one transaction at a time.

Locks

The Concurrency Control service coordinates concurrent use of a resource using locks. A lock represents the ability of a specific client to access a specific resource in a particular way. Each lock is associated with a single resource and a single client. Coordination is achieved by preventing multiple clients from simultaneously possessing locks for the same resource if the activities of those clients might conflict. To achieve coordination, a client must obtain an appropriate lock before accessing a shared resource.

Lock Modes

The Concurrency Control Service defines several *lock modes*, which correspond to different categories of access. Having a variety of lock modes allows more flexible conflict resolution. For example, providing different modes for reading and writing allows a resource to support multiple concurrent clients that are only reading the data of the resource. The Concurrency Control Service also defines *intention locks* that support locking at multiple levels of granularity.

Lock Granularity

The Concurrency Control Service does not define the granularity of the resources that are locked. It defines a *lock set*, which is a collection of locks associated with a single resource. It is up to clients of the Concurrency Control Service to associate a lock set with

each resource. Typically, if an object is a resource, the object would internally create and retain a lock set. However, the mapping between objects and resources (and lock sets) is up to the object implementation; the mapping could be one to one, but it could also be one to many, many to many, or many to one.

Conflict Resolution

A client obtains a lock on a resource using the Concurrency Control Service. The service will grant a lock to a client only if no other client holds a lock on the resource that would conflict with the intended access to the resource. The decision to grant a lock depends upon the modes of the locks held or requested. For example, a read lock conflicts with a write lock. If a write lock is held on a resource by one client, a read lock will not be granted to another client.

Conflict Resolution for Transactions

The decision to grant a lock also depends upon the relationships among the transactions that hold or request a lock. In particular, if the transactions are related by nesting (nested transactions), a lock may be granted that would otherwise be denied.

Lock Duration

Typically, a transaction will retain all of its locks until the transaction is completed (either committed or aborted). This policy supports serializability of transactional operations. Using the two phase commit protocol, locks held by a transaction are automatically dropped when the transaction completes.

There are also situations where levels of isolation that are weaker than serializability are acceptable, such as when an application does not want other applications to change an object while reading it and does not refer to the object again within the transaction. In these circumstances, it is acceptable to release locks before the containing transaction completes, hence the duration will be shorter than the containing transaction.

To manage the release of the locks held by a transaction, the Concurrency Control service defines a lock coordinator. Lock sets that are related (for example, by being created by a resource manager for resources of the same type) and that should drop their locks together when a transaction commits or aborts may share a lock coordinator. It is up to clients of the concurrency control service to associate lock sets together and to release the locks when a transaction commits or aborts.

7.2 Locking Model

This section covers a number of important issues that relate to the locking model supported by the Concurrency Control Service. For a complete discussion of these issues the reader is directed to one of the standard texts on the subject[1].

The Lock Modes section applies to clients that operate in both transactional and non-transaction modes. The Multiple Possession Semantics, Two-Phase Transactional Locking, and Nested Transaction sections are relevant only to clients that operate in transactional mode.

7.2.1 Lock Modes

Read, Write, and Upgrade Locks

The Concurrency Control service defines *read* (R) and *write* (W) lock modes that support the conventional multiple readers, one writer policy. Read locks conflict with write locks, and write locks conflict with other write locks.

In addition, the Concurrency Control service defines an *upgrade* (U) mode. An upgrade mode lock is a read lock that conflicts with itself. It is useful for avoiding a common form of deadlock that occurs when two or more clients attempt to read and then update the same resource. If more than one client holds a read lock on the resource, a deadlock will occur as soon as one of the clients requests a write lock on the resource. If each client requests a single upgrade lock followed by a write lock, this deadlock will not occur.

Intention Read and Intention Write Locks

The granularity of the resources locked by an application determines the concurrency within the application. Coarse granularity locks incur low overhead (since there are fewer locks to manage) but reduce concurrency since conflicts are more likely to occur. Fine granularity locks improve concurrency but result in a higher locking overhead since more locks are requested. Selecting a suitable lock granularity is a balance between the lock overhead and the degree of concurrency required. Using the Concurrency Control service, an application can be developed to use coarse or fine granularity locks by defining the associated resources appropriately.

In addition, the Concurrency Control service supports variable granularity locking using two additional lock modes, *intention read* (IR) and *intention write* (IW). These additional lock modes are used to exploit the natural hierarchical relationship between locks of different granularity.

For example, consider the hierarchical relationship inherent in a database: a database consists of a collection of files, with each file holding multiple records. To access a record, a coarse grain lock may be set on the database, but at the cost of restricting other clients from accessing the database. Clearly, this level of locking is unsuitable. However, only setting a lock on the record is also inappropriate, because another client might set a lock on the file holding the record and delete or modify the file.

Using variable granularity locking, a client first obtains intention locks on the ancestor(s) of the required resource. To read a record in the database, for example, the client obtains an intention read lock (IR) on the database and the file (in this order) before obtaining the read lock (R) on the record. Intention read locks (IR) conflict with write locks (W), and intention write locks (IW) conflict with read (R) and write (W) locks.

1. See *Concurrency Control and Recovery in Database Systems* by P.A. Bernstein, V. Hadzilacos, and N. Goodman, or *Transaction Processing: Concepts and Techniques* by J.N. Gray and A. Reuter.

Lock Mode Compatibility

Table 1, “Lock Compatibility,” on page 5 defines the compatibility between the various

Table 1: Lock Compatibility

Granted Mode	Requested Mode				
	IR	R	U	IW	W
Intention Read (IR)					*
Read (R)				*	*
Upgrade (U)			*	*	*
Intention Write (IW)		*	*		*
Write (W)	*	*	*	*	*

locking modes (the symbol * is used to indicate when locks conflict). When a client requests a lock on a resource that cannot be granted because another client holds a lock on the resource in a conflicting mode, the client must wait until the holding client releases its lock. The Concurrency Control Service enforces a queueing policy such that all clients waiting for a new lock are serviced in a first in, first out order, and subsequent requests are blocked by the first request waiting to be granted the lock, unless the requesting client is a transaction that is a member of the same transaction family as an existing holder of the lock.

7.2.2 Multiple Possession Semantics

The Concurrency Control Service interface supports a locking model called multiple possession semantics. In this model, a client can hold multiple locks on the same resource simultaneously. The locks can be of different modes. In addition, a client can hold multiple locks of the same mode on the same resource; effectively, a count is kept of the number of locks of a given mode that have been granted to the client. When a client holds locks on a resource in more than one mode, other clients will not be granted a lock on the resource unless the requested lock mode is compatible with all of the modes of the existing locks.

In contrast, using the conventional locking model,[2] when a client holding a lock on a resource requests a lock on the same resource in a stronger mode, the existing lock is promoted from the weaker mode to the stronger mode (once the stronger lock can be granted without causing a conflict). Since lock modes form only a partial order, there will not

always be a stronger mode; in cases where neither mode is stronger, the lock will be promoted to the weakest mode that is at least as strong as either of the two modes.

7.3 Two-Phase Transactional Locking

The Concurrency Control Service provides primitives to support transaction-duration locking. Transaction duration locking is a special case of strict two-phase locking. In the first phase (the growing phase), a transaction obtains locks that are kept until the second phase (the shrinking phase), at which point they are released. A transaction must not release locks during the first phase, and must not obtain new locks during the second phase, otherwise concurrent computations may be able to view intermediate results of the transaction.

Two-phase locking is sufficient to guarantee serializability, hence this technique is used by transactions. During the normal execution of a transaction, no locks will be automatically dropped before the end of the transaction. When the transaction completes, the Concurrency Control Service must be informed so that the locks the transaction holds may be released. While releasing locks, no new locks may be obtained by the transaction.

When a transaction holds a lock that is no longer needed to ensure the transaction's serializability, or if a weaker level of isolation is acceptable, it is permissible to release the lock. The Concurrency Control Service therefore provides an operation that releases individual locks. This operation should be used with caution to ensure that the isolation level is appropriate for the application.

7.4 Nested Transactions

Lock conflicts within a transaction family are treated somewhat differently than conflicts between unrelated transactions. The underlying principle is the same for both: transactions must not be able to observe the effects of other transactions that might later abort. Unrelated transactions can abort independently; therefore, one transaction must not be permitted to acquire a lock that conflicts with a lock on the same resource held by an unrelated transaction.

Nesting imposes abort dependencies among related transactions. A parent transaction cannot abort without causing all of its children to abort. A child transaction that ends successfully cannot abort without causing its parent to abort. A transaction that cannot abort without causing another related transaction to abort (according to these guidelines and logical deductions) is said to be committed relative to that other transaction.

These dependencies make it possible to relax the rule that two transactions cannot acquire locks of conflicting modes on the same resource, without breaking the underlying principle. No partial effects can be observed and committed if all transactions that have done

2. See *Notes On Data Base Operating Systems* in *Operating Systems: An Advanced Course* (ed. Bayer, Graham, and Seegmuller) by J.N. Gray for further information.

work cannot abort without the observer being aborted. This property translates into a simple rule for nested locking: if all transactions holding locks on a resource are committed with respect to a transaction trying to acquire a lock on the resource, no conflict exists.

The multiple possession model (see previous section) facilitates the use of locks with nested transactions. In this model, multiple related transactions may hold locks of conflicting modes on a resource at the same time. When a nested transaction requests a lock, it is granted if all of the transactions holding locks on the resource are committed relative to the requestor. Both the requestor and previous holders are then considered to hold locks on the resource.

A child transaction can acquire a lock on a resource locked by its parent and then drop that lock without causing its parent to lose its lock. A transaction cannot drop a lock that it did not acquire itself. The lock possession semantics also require that each transaction acquire locks on its own behalf. It is improper to take locks on behalf of another transaction or to depend on locks held by other transactions.

Other approaches to nested transactions[3] treat a resource as being locked by a single transaction at a time. When a nested transaction requests a lock on a resource that is already locked by an ancestor transaction, the nested transaction becomes the new owner of the lock. When a nested transaction commits, ownership of all of its locks is transferred to its parent. When a nested transaction aborts, ownership of its locks reverts to the previous owners. The Concurrency Control service performs these lock transfers automatically. The multiple possession semantics model is functionally equivalent to this model, but it supports simpler interfaces.

7.5 CosConcurrencyControl Module

The Concurrency Control Service is defined by the CosConcurrencyControl module, which provides interfaces that support both transactional and non-transactional modes of operation. This section defines the interfaces and describes the operations they support.

- The interfaces provide two modes of operation that correspond to those supported by the Transaction Service; in both modes, locks are identified by the lock set they are associated with and the mode of the lock.
- A client of the Concurrency Control Service may operate in an implicit mode such that locks are acquired on behalf of the current transaction (for transactional clients) or current thread (for non-transactional clients).
- For transactional clients, a second alternative is possible that involves the client identifying the transaction by means of a reference to the transaction's coordinator object (the explicit mode of operation).

Locks are acquired on lock sets. Two sets of operations are provided by the *LockSetFactory* interface to create lock sets, one creates a lock set that can be used by clients operat-

3. See *Nested Transactions: An Approach To Reliable Distributed Computing* by J.E.B. Moss for further information.

ing in the implicit mode (the *LockSet* interface), the other creates a lock set for explicit mode transactional clients (the *TransactionalLockSet* interface). In addition, the *LockCoordinator* interface is provided to allow a client to release all locks held by a specific transaction.

The following sections define the types and exceptions common to both types of interface, the interfaces themselves, and describes the responsibilities of a user for managing transaction-duration locks.

OMG IDL for the CosConcurrencyControl module shown on the following page.

```
#include <CosTransactions.idl>
 module CosConcurrencyControl {

     enum lock_mode {
         read,
         write,
         upgrade,
         intention_read,
         intention_write
     };

     exception LockNotHeld{};

     interface LockCoordinator
     {
         void drop_locks();
     };

     interface LockSet
     {
         void lock(in lock_mode mode);
         boolean try_lock(in lock_mode mode);

         void unlock(in lock_mode mode)
             raises(LockNotHeld);
         void change_mode(in lock_mode held_mode,
                          in lock_mode new_mode)
            raises(LockNotHeld);
         LockCoordinator get_coordinator(
             in CosTransactions::Coordinator which);
     };

     interface TransactionalLockSet
     {
         void lock(in CosTransactions::Coordinator current,
                   in lock_mode mode);
         boolean try_lock(in CosTransactions::Coordinator current,
                          in lock_mode mode);
         void unlock(in CosTransactions::Coordinator current,
                     in lock_mode mode)
            raises(LockNotHeld);
         void change_mode(in CosTransactions::Coordinator current,
                          in lock_mode held_mode,
```

```
                        in lock_mode new_mode)
            raises(LockNotHeld);
        LockCoordinator get_coordinator(
            in CosTransactions::Coordinator which);
    };

    interface LockSetFactory
    {
        LockSet create();
        LockSet create_related(in LockSet which);
        TransactionalLockSet create_transactional();
        TransactionalLockSet create_transactional_related(in
            TransactionalLockSet which);
    };
};
```

7.5.1 Types and Exceptions

The types and exceptions described in this section apply to both the *Lockset* and *TransactionalLockset* interfaces.

TABLE 2.

```
module CosConcurrencyControl {
    enum lock_mode {
        read,
        write,
        upgrade,
        intention_read,
        intention_write
    };

    exception LockNotHeld{};
```

lock_mode

The `lock_mode` type represents the types of lock that can be acquired on a resource.

LockNotHeld

The `LockNotHeld` exception is raised when an operation to unlock or change the mode of a lock is called and the specified lock is not held.

7.5.2 LockCoordinator Interface

The *LockCoordinator* interface enables a transaction service to drop all locks held by a transaction. The *LockSet* and *TransactionalLockSet* interfaces create instances of the

LockCoordinator for each transaction. The *LockCoordinator* interface provides a single operation:

TABLE 3.

```
interface LockCoordinator {
    void drop_locks();
};
```

drop_locks

Releases all locks held by the transaction. This call is designed to be used by transactional clients when a transaction commits or aborts. For nested transactions, this operation must be called when the nested transaction aborts, but the call need only be made once for a transaction family when that family commits (recall that nested transaction commits are handled implicitly by the Concurrency Control service).

7.5.3 LockSet Interface

For clients operating in the implicit mode, locks are acquired and released on lock sets which are defined by means of the LockSet interface. The LockSet interface only provides operations to acquire and release locks on behalf of the calling thread or transaction. The interface does not provide support for transactional clients that use the explicit Transaction Service interfaces.

TABLE 4.

```
interface LockSet {
    void lock(in lock_mode mode);

    boolean try_lock(in lock_mode mode);

    void unlock(in lock_mode mode)
.       raises(LockNotHeld);

    void change_mode(in lock_mode held_mode,
                     in lock_mode new_mode)
        raises(LockNotHeld);

    LockCoordinator get_coordinator(in
        CosTransactions::Coordinator which);
};
```

When calls to acquire or release locks are made outside the scope of a transaction then it is assumed that the client is operating in the *non-transactional* mode (the concurrency control implementation must use the appropriate Transaction Service operation to determine

whether the current thread is executing on behalf of a transaction).

lock

Acquires a lock on the specified lock set in the specified mode. If a lock is held on the same lock set in an incompatible mode by another client then the operation will block the calling thread of control until the lock is acquired. If a call that is on behalf of a transactional client is blocked and the transaction is aborted then the call will return with the `Transactions::TransactionRolledBack` exception.

try_lock

Attempts to acquire a lock on the specified lock set. If the lock is already held in an incompatible mode by another client then the operation returns a FALSE result to indicate that the lock could not be acquired.

unlock

Drops a single lock on the specified lock set in the specified mode (recall that a lock can be held multiple times in the same mode). Calls to drop a lock that is not held result in the `LockNotHeld` exception being raised

change_mode

Changes the mode of a single lock (recall that multiple locks may be held on the same lock set). If the new mode conflicts with an existing mode held by an unrelated client, then the `change_mode` operation blocks the calling thread of control until the new mode can be granted. Like the `lock` call, if the client is a transaction and it aborts while the thread of control if blocked then the `Transactions::TransactionRolledBack` exception will be raised. Similarly, when a call is made to change the mode of a lock, but the lock is not held in the specified mode, the `LockNotHeld` exception will be raised.

get_coordinator

Returns the lock coordinator associated with the specified transaction.

7.5.4 TransactionalLockSet Interface

The *TransactionalLockSet* interface provides operations to acquire and release locks on a lock set on behalf of a specific transaction. The operations that make up the *Transactional-*

LockSet interface are:

TABLE 5.

```
interface TransactionalLockSet {
    void lock(in CosTransactions::Coordinator which,
              in lock_mode mode);

   boolean try_lock(in CosTransactions::Coordinator which,
                    in lock_mode mode);

    void unlock(in CosTransactions::Coordinator which,
                in lock_mode mode)
        raises(LockNotHeld);

   void change_mode(in CosTransactions::Coordinator which,
                    in lock_mode held_mode,
                    in lock_mode new_mode)
        raises(LockNotHeld);

    LockCoordinator get_coordinator(in
        CosTransactions::Coordinator which);
};
```

The operations provided by the *TransactionalLockSet* interface operate in an identical manner to the equivalent operations provided by the *LockSet* interface. The interfaces differ in that for the *TransactionalLockSet* interface the identity of the transaction is passed explicitly as a reference to the coordinator for the transaction instead of implicitly through an association with the calling thread.

7.5.5 LockSetFactory Interface

Lock sets are created using the *LockSetFactory* interface.

TABLE 6.

```
interface LockSetFactory {
    LockSet create();
    LockSet create_related(in LockSet which);

    TransactionalLockSet create_transactional();
    TransactionalLockSet
        create_transactional_related(in
            TransactionalLockSet which);
};
```

This interface provides two sets of operations that return new *LockSet* and *Transactional-LockSet* instances.

create

Creates a new lock set and lock coordinator.

create_related

Creates a new lock set that is related to an existing lock set. Related lock sets drop their locks together.

create_transactional

Creates a new transactional lock set and lock coordinator for explicit mode transactional clients.

create_transactional_related

Creates a new transactional lock set that is related to an existing lock set. Related lock sets drop their locks together.