The ODMG Lock Manager

by Thomas Mahler, Armin Waibel

1. What it does

The OJB ODMG implementation provides object level transactions as specified by the ODMG. This includes features like registering objects to transactions, persistence by reachability (a toplevel object is registered to a transaction, and also all its associated objects become registered implicitely) and as a very important aspect: object level locking.
Lockmanagement is needed to synchronize concurrent access to objects from multiple transactions (possibly from remote machines).
An example: There are two transactions `tx1` and `tx2` running on different physical machines. `Tx1` acquired a write lock on an object `obj` with the globally unique identity `oid`. Now also `tx2` tries to get a write lock on an object `obj'` (it's not the *same* object as it resides in a different VM!) with the *same* identity `oid` (an OJB Identity is unique accross VMs !). The OJB LockManager is responsible for detecting this conflict and doesn't allow `tx2` to obtain a write lock to prevent data inconsistency.

The ODMG Api allows transactions to lock an object `obj` as follows:

`org.odmg.Transaction.lock(Object obj, int lockMode)`,

where lockMode defines the locking mode:

```
/** Read lock mode.*/
public static final int  READ = 1;

/** Upgrade lock mode. */
public static final int  UPGRADE = 2;

/** Write lock mode. */
public static final int  WRITE = 4;
```

A sample session could look as follows:

```
// get odmg facade instance
Implementation odmg = OJB.getInstance();

//open database
Database db = odmg.newDatabase();
db.open(&quot;repository.xml&quot;, Database.OPEN_READ_WRITE);

// start a transaction
Transaction tx = odmg.newTransaction();
tx.begin();

MyClass myObject = ... ;

// lock object for read access
tx.lock(myObject, Transaction.READ);

// now perform read access on myObject ...

// lock object for write access
tx.lock(myObject, Transaction.UPGRADE);

// now perform write access on myObject ...
// finally commit transaction to make changes to myObject persistent
tx.commit();
```

The ODMG specification does not say if locks must be acquired explicitely by client applications or may be acquired implicitely. OJB provides implicit locking for the application programmers convenience: On commit of a transaction all read-locked objects are checked for modifications. If a modification is detected, a write lock is acquired for the respective object. If automatic acquisition of read- or write-lock failes, the transaction is aborted.

On locking an object to a transaction, OJB automatically locks all associated objects (as part of the *persistence by reachability feature*) with the same locking level. If application use large object nets which are shared among several transactions acquisition of write-locks may be very difficult. Thus OJB can be configured to aquire only read-locks for associated objects.
You can change this behaviour by modifying the file OJB.properties and changing the entry `LockAssociations=WRITE` to `LockAssociations=READ`.

The ODMG specification does not prescribe transaction isolationlevels or locking strategies to be used. Thus there are no API calls for setting isolationlevels. OJB provides four different isolationlevels that can be configured for each persistent class in the XML repository.
The isolationlevel of a class can be configured with the following attribute to a ClassDescriptor:

```
<ClassDescriptor isolation="read-uncomitted" ...>
    ...
</ClassDescriptor>
```

The four supported values are:

- read-uncommitted
- read-committed
- repeatable-read
- serializable

The semantics of these isolationlevels is defined below.

2. How it works

To provide Lockmanagement in a massively distributed environment as the OJB client/server architecture, OJB implements a LockManager that allows transaction coordination accross multiple threads, multiple VMs and even multiple physical machines running OJB ODMG transactions. The Default Implementation of this LockManager uses a database table to store locks. To make locks persistent allows to make them visible to all connected ODMG clients. Thus there is no need for an additional LockManager server that is accessed from all ODMG clients.

The LockManager interface provides the following API:

```
public interface LockManager
{
    /**
     * aquires a readlock for transaction tx on object obj.
     * Returns true if successful, else false.
     */
    public abstract boolean readLock(TransactionImpl tx, Object obj);

    /**
     * aquires a writelock for transaction tx on object obj.
     * Returns true if successful, else false.
     */
    public abstract boolean writeLock(TransactionImpl tx, Object obj);

    /**
     * upgrades readlock for transaction tx on object obj to a writelock.
     * If no readlock existed a writelock is acquired anyway.
     * Returns true if successful, else false.
     */
    public abstract boolean upgradeLock(TransactionImpl tx, Object obj);

    /**
     * releases a lock for transaction tx on object obj.
     * Returns true if successful, else false.
     */
    public abstract boolean releaseLock(TransactionImpl tx, Object obj);

    /**
     * checks if there is a readlock for transaction tx on object obj.
     * Returns true if so, else false.
     */
    public abstract boolean checkRead(TransactionImpl tx, Object obj);

    /**
     * checks if there is a writelock for transaction tx on object obj.
     * Returns true if so, else false.
     */
    public abstract boolean checkWrite(TransactionImpl tx, Object obj);
}
```

The lockmanager must allow and disallow locking according to the Transaction Isolationlevel specified for `obj.getClass()` in the XML RepositoryFile. It does so by applying a corresponding LockStrategy. LockStrategies are selected by the LockStrategyFactory:

```
private static LockStrategy readUncommitedStrategy =
                                new ReadUncommittedStrategy();
private static LockStrategy readCommitedStrategy =
                                new ReadCommittedStrategy();
private static LockStrategy readRepeatableStrategy =
                                new RepeatableReadStrategy();
private static LockStrategy serializableStrategy =
                                new SerializableStrategy();

/**
 * Obtains a LockStrategy for Object obj. The Strategy to be used is
 * selected by evaluating the ClassDescriptor of obj.getClass().
 *
 * @return LockStrategy
 */
public static LockStrategy getStrategyFor(Object obj)
{
    int isolationLevel = getIsolationLevel(obj.getClass());
    switch (isolationLevel)
    {
        case IsolationLevels.RW_READ_UNCOMMITTED:
            return readUncommitedStrategy;
        case IsolationLevels.RW_READ_COMMITTED:
            return readCommitedStrategy;
        case IsolationLevels.RW_REPEATABLE_READ:
            return readRepeatableStrategy;
        case IsolationLevels.RW_SERIALIZABLE:
            return serializableStrategy;
        default:
            return readUncommitedStrategy;
    }
}
```

The four LockStrategies implement different behaviour according to the underlying isolationlevel. The semantics of the strategies are defined by the following table:

Nr.	Name of TestCase	Transactions		Transaction-Isolationlevel			
		Tx1	**Tx2**	**ReadUncommitted**	**ReadCommitted**	**RepeatableRead**	**Serializable**
1	SingleReadlock	R		True	True	True	True

18	ReadThenRead	R		True	True	True	True
		R					
2	UpgradeReadLock	R		True	True	True	True
		U					
3	ReadThenWrite	R		True	True	True	True
		W					
4	SingleWriteLock	W		True	True	True	True
5	WriteThenRead	W		True	True	True	True
		R					
6	MultipleReadLock	R	R	True	True	True	False
7	UpgradeWithExistingReader	R	U	True	True	False	False
8	WriteWithExistingReader	R	W	True	True	False	False
9	UpgradeWithMultipleReaders	R	R	True	True	False	False
			U				
10	WriteWithMultipleReaders	R	R	True	True	False	False
			W				
11	UpgradeWithMultipleReadersOn1	R	R	True	True	False	False
		W					
12	WriteWithMultipleReadersOn1	R	R	True	True	False	False
		W					
13	ReadWithExistingWriter	W	R	True	False	False	False
14	MultipleWriteLocks	W	W	False	False	False	False
15	ReleaseReadLock	R		True	True	True	True
		Rel	W				
16	ReleaseUpgradeLock	U		True	True	True	True
		Rel	W				
17	ReleaseWriteLock	W		True	True	True	True

		Rel	W				
	Acquire ReadLock	R					
	Acquire WriteLock	W					
	Upgrade Lock	U					
	Release Lock	Rel					

The table is to be read as follows. The acquisition of a single read lock on a given object (case 1) is allowed (returns True) for all isolationlevels. To upgrade a single read lock (case 2) is also allowed for all isolationlevels. If there is already a write lock on a given object for tx1, it is not allowed (returns False) to obtain a write lock from tx2 for all isolationlevels (case 14).

The isolationlevels can be simply characterized as follows:

Uncommitted Reads
Obtaining two concurrent write locks on a given object is not allowed (case 14). Obtaining read locks is allowed even if another transaction is writing to that object (case 13). (Thats why this level is also called *dirty reads*)

Committed Reads
Obtaining two concurrent write locks on a given object is not allowed. Obtaining read locks is allowed only if there is no write lock on the given object (case 13).

Repeatable Reads
As commited reads, but obtaining a write lock on an object that has been locked for reading by another transaction is not allowed (case 7).

Serializable transactions
As Repeatable Reads, but it is even not allowed to have multiple read locks on a given object (case 6).

The proper behaviour of the LockStrategies is checked by JUnit TestCases that implement test methods for each of the 17 cases specified in the above table. (See code for classes `test.ojb.odmg.LockTestXXX`)

3. Locking in distributed environment

############ TODO ############

4. Implement you own lock manager

The LockManager default implementation uses a database table to make locks globally visible to all connected clients. This is a foolproof solution as it does not require a separate LockManager server. But it involves a lot of additional database traffic, as each lock check, acquisition or release results in database operations.
This may not be viable in some environments. Thus OJB allows to plug in user defined LockManagers implementing the `ojb.odmg.locking.LockManager` interface. OJB obtains its LockManager from the factory `ojb.odmg.locking.LockManagerFactory`. This Factory can be configured to generate instances of a specific implementation by changing the following entry in the configuration file OJB Properties file:

```
LockManagerClass=ojb.odmg.locking.LockManagerDefaultImpl
```

to:

```
LockManagerClass=acme.com.MyOwnLockManagerImpl.
```

Note:

Of course I'm interested in your solutions! If you have implemented something interesting, just contact me.