rade locks reduce the likelihood of deadlocks.

Upgrade locks have the same compatibility as read locks, but two upgrade locks are not mutually compatible. Two update locks are not granted for an object as that would very likely result in a deadlock. Hence, the use of upgrade locks avoids deadlocks.

Figure 11.6
patibility of CORBA ency Control Service

	Intention Read	Read	Upgrade	Intention Write	Write
Intention Read	+	+	+	–	–
Read	+	+	+	–	–
Upgrade	+	–	–	–	–
Intention Write	+	–	–	–	–
Write	–	–	–	–	–

Locksets

The CORBA Concurrency Control service defines the interface `Lockset` in order to standardize operations for the acquisition and release of locks. Instances of this type are associated with any CORBA object to which concurrent access needs to be controlled. The interface is contained in the `CosConcurrencyControl` module, an excerpt of which is shown in Figure 11.7. In addition to the obvious operations to `lock` and `unlock`, `LockSet` also defines an operation `try_lock` that allows a transaction to check whether it can acquire a lock in a particular mode and `change_mode` in order to upgrade or downgrade the mode of a lock.

Figure 11.7
Excerpt of CORBA Concurrency Control Interfaces

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

In addition to the locking operations of `LockSet`, the Concurrency Control service also standardizes the operations of a factory to create locksets. The `create` operation creates a lockset and the `create_related` operation creates a lockset that is associated to a related lockset. Related locksets release their locks together and therefore unlocking of a set of related objects can be achieved with just one object request.

11.2.6 Summary

We have seen how concurrency control can be applied to distributed objects. We have identified the need for concurrency control when shared objects are accessed by concurrent transactions outside the control of a database. In these circumstances the designer of a server object has to apply two-phase locking in order to achieve serializability of the transactions. We have discussed hierarchical locking, a mechanism to efficiently implement the locking of large numbers of objects and we have seen how both two-phase locking and hierarchical locking are supported by the CORBA Concurrency Control service.

11.3 The Two-Phase Commit Protocol

In the previous section, we saw how the isolation property can be implemented. We have also already identified that the durability property of transactions will be achieved using a persistent state service, as discussed in Chapter 10. We have also seen that the implementation of consistency preservation is the responsibility of the client programmer. Hence we are only missing atomicity. In order to discuss how atomicity of distributed object transactions is achieved, we first present the roles that objects can adopt in distributed transactions. We then discuss that committing a distributed transaction involves distributed decision-making and that two phases are needed for implementing a commit. The communication between the objects involved with the transaction is therefore called the *two-phase commit protocol* (2PC).

11.3.1 Roles of Objects in Distributed Transactions

There are different distributed objects involved in transactions. It is beneficial to identify the roles that these distributed objects play for transaction management in order to understand their obligations for a successful implementation of the transaction properties. The roles that we identify now are the most basic ones that are commonly found in all distributed transactional systems. It should be noted that some middleware adds more specific roles.

We already identified that it is usually the application that starts and ends transactions. The objects executing these applications are generally referred to as *transactional clients*. In the examples that we have used above, objects of types `DirectBanking` and `Reporting` acted as transactional clients. They invoke the begin, commit and abort operations from a transaction coordinator.

A transactional client issues transaction commands.

Each transaction has a *transaction coordinator*. The coordinator knows the identity of all server objects that participate in a transaction. In distributed systems, there are usually many transaction coordinator objects, as otherwise an artificial bottleneck would be created. The

A transaction coordinator controls the execution of the transaction.

transaction coordinator is often provided by transaction services because, unlike transactional clients and transactional servers, it is application-independent.

Transactional servers are server objects that participate in transactions.

Transactional server objects are the application-specific server objects. They are often stateful. Prior to participating in the transaction they register their involvement with the transaction coordinator. The transactional server objects implement the protocols that are necessary to ascertain the ACID properties of the transaction. In our bank example, the different instances of `Account` objects are transactional server objects. In the examples used in this chapter, all instance are accounts but it is quite possible that instances of different types can participate in the same transaction.

11.3.2 Two-Phase Commit for Flat Transactions

There may be many concurrent transactions in a distributed object system. Every transaction coordinator manages a subset of those concurrent transactions. It is therefore necessary for a transactional client to identify the transaction that they wish to manipulate and for the transactional servers to indicate in which transactions they participate. This is achieved by system-wide *unique transaction identifiers*. The transaction identifier is acquired by the coordinator during the start of a new transaction and then passed to clients. Clients may either explicitly or implicitly pass the transaction identifier to transactional servers.

Distributed transaction commits can only be done in two phases.

A transaction involves the operation of multiple transactional servers. Each of these transactional servers may perform modifications to its state during the transaction. For the transaction to commit successfully every single participating transactional server has to be able to commit. It is therefore not possible for the transaction coordinator to decide for itself whether or not to commit the transaction. For the transaction to be atomic, every single transactional server that participated in the transaction has to be able to commit. The coordinator has to obtain agreement from every single participating transactional server to be able to commit. Only after that agreement has been given can the coordinator trigger the completion of the commit. It is therefore necessary to split the implementation of the commit into two phases: the voting phase and the completion phase.

Voting Phase

In the voting phase, servers decide whether they can commit.

The purpose of the *voting phase* of the two-phase commit protocol is to see whether or not the transaction can be committed. This is done by voting. Obtaining the votes is initiated by the transaction coordinator. We have to assume that the coordinator knows all the transactional servers that are participating in a transaction. Servers therefore have to register with the coordinator as soon as they participate in the transaction. To implement the voting phase, the transaction coordinator then asks every participating server for a vote. The transaction commits if every single transactional server has voted for the commit. If a single server has not voted in favour of committing, the transaction will be aborted.

There are many reasons why a transaction may have to be aborted. It could be because a transactional server detects a state of inconsistency from which it cannot recover. The server

would then vote against the commit. Another reason could be that the transactional server has crashed at some point after registering its participation but before returning its vote. The coordinator will then receive an exception, time-out or other failure indication saying that the transactional server is unavailable, which it will interpret as a vote against the commit. Finally, the transactional coordinator itself may decide that it is not a good idea to commit and vote against the commit itself.

Completion Phase

The purpose of the *completion phase* of the two-phase commit protocol is to implement the commit. After receiving votes from all participating transactional servers, the coordinator collates the votes and, if all servers voted for the commit, it requests every server to do the commit. This involves storing modified states on persistent storage. The transactional servers confirm completion of the commit with the transactional server.

In the completion phase, servers perform the commit.

From Example 11.15, we note that the implementation of transactions is transparent to the client. It only starts and commits the transaction. However, the implementation of transactions is not transparent to the designers of transactional servers. They have to implement the operations that are necessary for the two-phase commit protocol. These are, in particular, the `vote` and the `doCommit` operations.

Transaction implementation is transparent to the client, but not to the server.

Message Complexity

Let us now investigate the overhead of processing distributed transactions. The overhead is dominated by the number of messages that are needed for the object requests that implement the transaction. We can also observe that the only variable in the number of participating objects is the number of transactional servers. Let us therefore assume that N transactional servers participate in a transaction.

Every transactional server needs to register with the coordinator which leads to N registration requests. The transactional coordinator needs to request a vote from every participating server. This results in a total of N voting requests. Moreover, the coordinator needs to request N `doCommit` requests from the servers. This adds up to about $3 \times N$ requests in the case where there is no failure. Additional requests may be needed if a failure occurs, but we can note that the complexity of transaction processing is linear in relation to the number of participating transactional servers.

The complexity of two-phase commit is linear in relation to the number of participating servers.

Server Uncertainty

The sequence diagram in Example 11.15 slightly oversimplifies the picture. Transactions are not so easy to implement as it may seem from this example. The reason for this is the so-called *server uncertainty*. It denotes the period after a transactional server has voted in favour of a commit and before it receives the request to do the commit. During this period the server is ready to commit but does not know whether the commit is actually going to happen.