An Architectural Framework for Object Management Systems (Area Paper)

Steven S. Popovich

Columbia University
Department of Computer Science
New York, NY 10027

CUCS-026-91
3 September 1991

Abstract

Much research has been done in the last decade in the closely related areas of object-oriented programming languages and databases. Both areas now seem to be working toward a common end, that of an *object management system*, or OMS. An OMS is constructed similarly to an OODB but provides a general purpose concurrent object oriented programming language as well, sharing the object base with the OODB query facilities. In this paper, we will define several different types of object systems (object servers, persistent OOPL's, OODB's and OMS's) in terms of their interfaces and capabilities. We will examine the distinguishing features and general architecture of systems of each type in the light of a general model of OMS architecture.

Popovich is supported in part by NASA grant NGT 50583. The Programming Systems Laboratory is supported by National Science Foundation grants CCR-9000930 and CCR-8858029, by grants from AT&T, BNR, DEC and SRA, by the New York State Center for Advanced Technology in Computers and Information Systems and by the NSF Engineering Research Center for Telecommunications Research.

1. Introduction: Object Management Systems

Object-oriented programming languages (OOPL's) have existed for more than twenty years, since the development of Simula-67 [Dahl 66], and have become popular during the 1980's, with the initial success of Smalltalk-80 [Goldberg 83] encouraging much subsequent research into object-oriented languages. The area has been extensively investigated, with many languages being developed, each taking a somewhat different approach to object orientation. Wegner [Wegner 87] has developed a taxonomy of these various approaches.

Recently, as the object-oriented model has become better understood, many researchers have applied it to the problem of reliable distributed computing. These projects fall into two main categories, *object servers* and *persistent OOPL's*, depending on whether they see the problem as one of providing a reliable repository for data objects (an object server), or of providing programming-language support for reliable computing (a persistent OOPL). Persistent OOPL's are sometimes built on top of an underlying object server. Persistent Owl [Christou 88], built at the University of Massachusetts on top of Mneme [Moss 88], is an example of this. Avalon [Detlefs 88; Clamen 90], built at Carnegie-Mellon University on top of the Camelot [Spector 88] server, is similar. In this case, however, the object server is separate from the Camelot transaction server. Both object servers and persistent OOPL's commonly make use of serializable atomic transactions, originally developed to ensure database consistency, for concurrency control and reliability support (crash recovery) [Eswaran 76].

Meanwhile, the database community has recognized a need for a new database model. The standard relational, hierarchical and network models have been found unsuitable for certain emerging application areas, such as engineering environments and applied AI, and they have adopted the object-oriented paradigm as a possible solution to their problem. Bancilhon [Bancilhon 88] gives a view of the state of the art in object-oriented database (OODB) research. These OODB's have object models similar to those of object servers and persistent OOPL's, but the main emphasis in OODB's is on queries and other data-oriented uses of objects rather than the more control-oriented use of persistent objects in programming languages.

In the last few years, with the maturation of the OOPL field and the development of OODB's, the idea of an *Object Management System* (OMS) has come into being. An OMS is a fusion of an OOPL and an OODB. It can be thought of as either an OOPL backed up with database support, or as an OODB with an extended object-oriented data manipulation language. In an OMS, programming language and database functionality are transparently integrated into a single system. An OMS commonly appears as a database within which objects are specified by use of an object-oriented language, and whose objects are available to its programming language. The similarity of the requirements expressed by the database community [Stonebraker 90] and the object-oriented research community [Atkinson 90] in their respective "manifestos" shows clearly that OMS's are an idea whose time has come, and which is becoming increasingly well-defined.

In this paper, we survey several object servers[1], persistent OOPL's, OODB's and OMS's

[1]We do not abbreviate this term, since shortening it to OS's would invite confusion with the common abbreviation for operating systems.

currently under development or in use. The systems we consider are primarily research systems, although we have included a few OODB's and an OMS that are now commercially available. Since the OMS field is so new, generally the commercial systems are outgrowths of earlier research systems. We begin by defining the distinguishing characteristics of the various types of systems. Each of these areas then serves as the subject of a section of this paper. Each section describes the common and the distinguishing features of its subject class of object systems and examines a few prominent examples in detail, paying particular attention to their concurrent and multi-user aspects. We conclude by summarizing the contributions of this survey.

2. Important Features of Object Management Systems

An OMS generally must have several important elements of each of the other types of systems — Object Server, OOPL and OODB. The requirements of the three other system types overlap to produce characteristics of an OMS. We first examine the main features of Object Servers, OOPL's and OODB's, then combine them to answer the question, "What is an OMS?"

2.1. Object Servers

An Object Server is the simplest of the three types of systems. Its main characteristics are that it must provide persistent object storage and some form of object management (at least the primitive creation, deletion, and updating of objects), sometimes allows object clustering or complex objects, and may have a data definition language (DDL) for describing the classes of objects it stores. An Object Server without a DDL is known as a *byte server*, since the objects it stores are simply streams of bytes, with no semantics attached to the bytes. An Object Server generally supports some level of concurrency, although single-user Object Servers do exist.

2.1.1. ObServer II

ObServer II [Hornick 87; Fernandez 89a] is perhaps the best known of the object servers. Its objects are simple, undifferentiated collections of bytes, so ObServer is also the best known byte server. In ObServer, all objects live on a central server, which manages the objects and controls access to them. Clients first bring the object over from the server, then make their modifications to the object, and finally commit their changes back to the server. Objects are completely passive in ObServer; there is no general trigger or active data mechanism, and no processes within objects. ObServer's "trigger" mechanism is actually a "soft lock" option for sending a notification message whenever an object is changed.

With this simple client/server model, and concurrency occurring only between clients, ObServer has a simple architecture, with six main modules and little interaction between them. Figure 2-1 shows these modules and their only documented interaction, aside from the common use of the `MESSAGES` module by all other modules in the system — a common lock table used by the `OBJECTS` and `TRANSACTIONS` modules. The operations available to the client through the server interface divide along the lines of these modules in the obvious way. The `SEGMENTS` module provides operations for manipulating storage segments, which contain sets of "related" objects and are the unit of transfer between main memory and disk; the `OBJECTS` module provides object management operations such as creating and deleting objects and moving them between storage segments; the `TRANSACTIONS` module provides transaction management operations. The `SYSTEM` module manages resources and scheduling within the server, and the `MESSAGES` module handles network communication with clients. All modules use the `MESSAGES` module, but these interconnections are not shown since they would complicate the interconnection graph. The second level of boxes in the graph are the major submodules and data structures used by the modules.

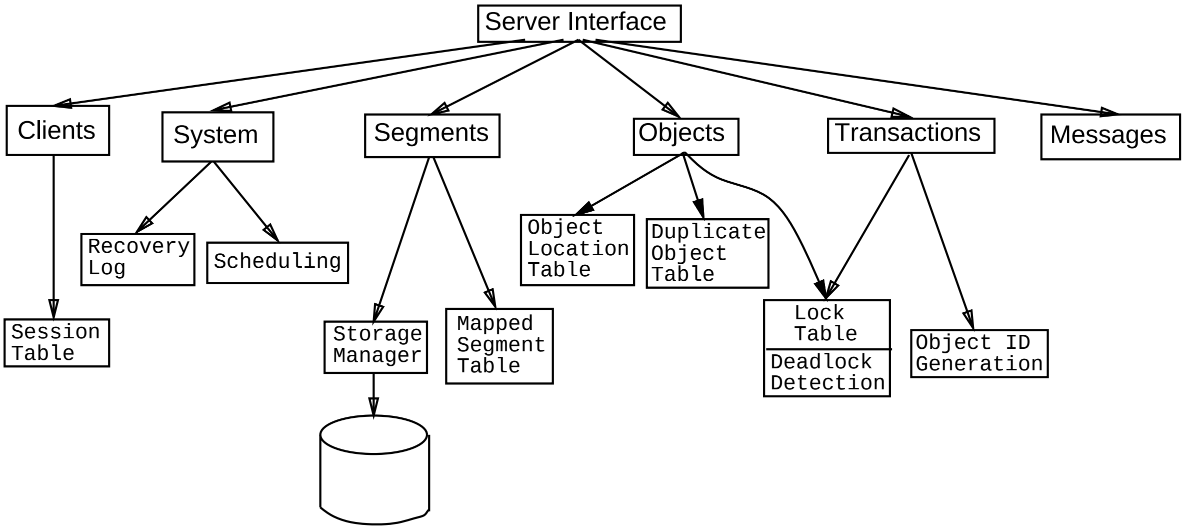

Figure 2-1: ObServer II Server Architecture

2.1.2. Mneme

Mneme is an Object Server intended to support software engineering environments [Moss 88]. It is designed to support a variety of programming languages in this role, and to allow using an object database like Exodus, or another object server such as ObServer II, as a back end. Thus, unlike most Object Servers, Mneme has two external interfaces, one at each end. This is shown in Figure 2-2, where multiple front ends (*e.g.*, Persistent C++ and Persistent Owl) and back ends (*e.g.*, Exodus and ObServer II) are shown interfacing to Mneme's object manager.

Mneme is not, however, currently implemented in this ambitious fashion. The current prototype, which we call "Basic" Mneme, includes a simple back end supporting a single-user system, supporting transactions only as a "session" mechanism where atomicity means only that transactions can be aborted and previous changes undone. So although Mneme is not yet implemented as a multi-user concurrent object system, it is designed as one, and our consideration of Mneme in this paper focuses on the interesting features of the design for a concurrent Mneme.

2.1.3. Camelot

Camelot [Spector 88] is a server that supports transactions on arbitrary collections of bytes. Camelot itself does not provide any object structure or persistent storage for data objects; these are provided by other servers in a Camelot transaction system. Camelot is more properly called a *transaction server*. A Camelot transaction system has a layered architecture, in which

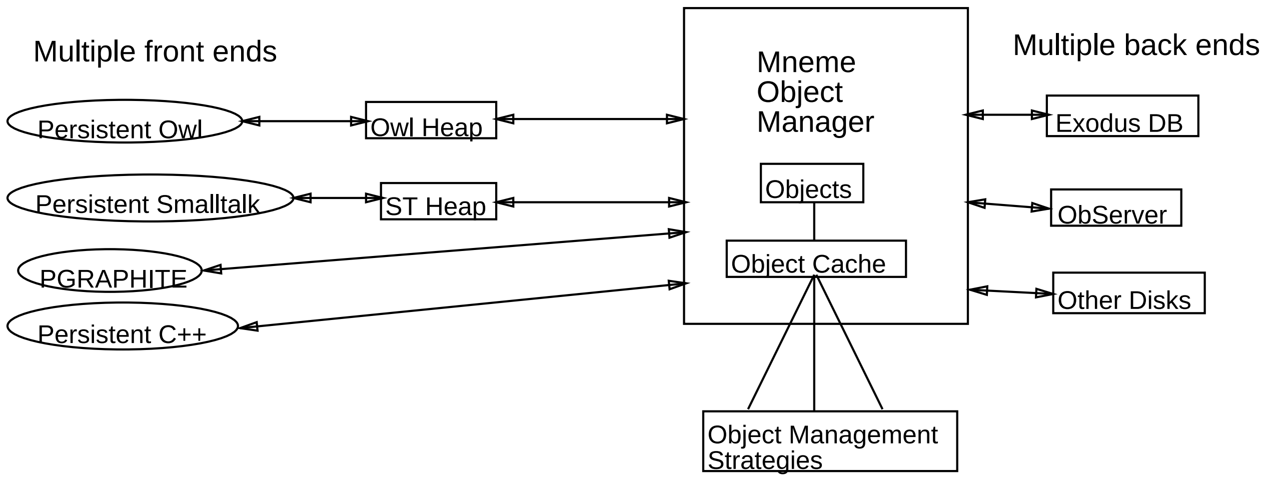

Figure 2-2: Mneme Object System Architecture

applications communicate with the transaction server through another server that provides an "object" abstraction. The Camelot transaction manager itself is built upon a modified version of the Mach [Rashid 87] OS's message passing facility, which depends in turn on other system facilities, as shown in Figure 2-3. Spector [Spector 89] has proposed a general architecture for distributed database systems based on the Camelot architecture. Camelot provides a locking-based transaction facility through a library [Bloch 89] with primitives for multi-threaded concurrency and remote procedure call access to the transaction server. The term *thread* is used in this paper to refer both to a general notion of lightweight processes, or multiple threads of control in a single address space, and the Mach operating system's particular implementation of this notion This library is used to implement "object" servers; the applications, as mentioned earlier, then use the "object" servers. We use the term "object", quoted here because the objects provided by the servers are not necessarily objects in the sense of an object-oriented programming language, although they are C++ objects in one version of Avalon (described in Section 2.3.2). The "objects", for example, might be byte strings, simple data items as in a conventional database, or entire files on disk, or might exist explicitly as an interface to a server.

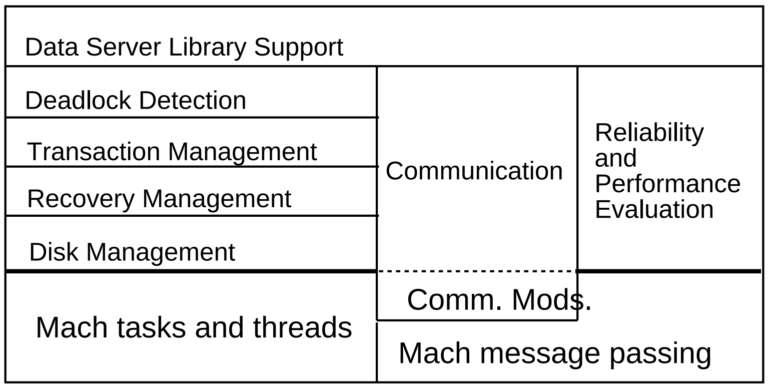

Figure 2-3: Architectural Layers in Camelot

2.1.4. DHOMS

DHOMS (Dynamic Hierarchical Object Management System) [Ben-Shaul 91] is the object server for the Marvel [Barghouti 90] rule-based software development environment. DHOMS has a layered architecture, as shown in Figure 2-4. It consists of three main modules: The transaction manager (TM), the lock manager (LM) and the data manager (DM), with an RPC client interface, a query processor and the Marvel rule processor (not discussed here) layered above them. The rule processor is located in the object server in order to minimize the need for transferring objects to the client. The client does not have an object cache; all objects always reside on the server.

The DM consists of three submodules: The storage manager and file manager, each of which is responsible for storage of a subset of the data, and the object manager, which is layered above the other two submodules and provides a uniform object abstraction for the two object implementations.

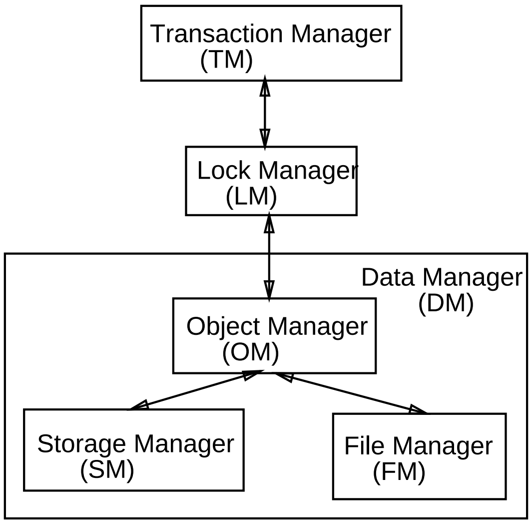

Figure 2-4: DHOMS Object Server Architecture

The OM is responsible for maintaining the association of object identities with disk storage, and also holds the locks on the objects; the LM has the lock compatibility matrix and makes the decision as to whether or not a lock request will be granted, but calls OM operations to actually grant and release locks. This was done so that the LM would not have to know about how the locks were stored. Instead, the OM is responsible for storing lock information, in the same way that it stores the other attributes of the objects. Above the LM is the TM, which implements two-phase locking transactions.

Access to objects is by navigation and fetching by OID only; there is no query language in DHOMS. Although the Marvel rule language has query-like constructs (*e.g.*, quantifiers), these are implemented in the rule processor rather than in DHOMS.

DHOMS is currently implemented as a single process serving many clients; design work is in progress on an implementation using a central server and a hierarchy of subsidiary servers. When implemented, this should transfer a substantial amount of the load from the single server to the client sites.

2.2. Object-Oriented Data Bases

An object-oriented data base (OODB) has the characteristics of an object server, and in fact will often be implemented using an object server as a back end. However, an OODB always has a data definition language and a query facility. The query facility generally takes the form of an SQL-like query language, similar to those provided by relational databases. In fact, we consider the query facility to be the primary distinction between OODB's and the other types of object systems.

An object-oriented database that supports concurrency preserves consistency by the means of a transactional model for concurrency control and crash recovery. This is most often a single, fixed transaction model, or a choice among a small number of algorithms may be possible by choosing lock modes. One OODB, Papyrus [Neimat 90], provides a table-based mechanism for specifying locking-based concurrency control schemes or optimistic algorithms based on "soft locks". Exodus, an OODB generation system described in Section 2.2.1, allows an implementor of an OODB instance to replace parts of its concurrency control code in order to implement a locking protocol appropriate to the data. Some OODB's provide a compiled data manipulation language, generally in the form of an OOPL, as well as ad-hoc query access, but other OODB's allow access only through the query facility.

2.2.1. EXODUS

EXODUS [Richardson 87; Carey 86] is an extensible object-oriented data base. It may be extended by adding new data types to the system. While many databases allow defining new types, EXODUS allows the implementor of a particular database system instance to add types at the same level as system types, specifying not only the structure and operators of the type but the low-level access method, locking protocol, procedures for processing queries on the type, and the optimizations that may be performed on the queries.

An EXODUS database is accessed solely through the query facility. A database is specified in EXODUS by means of a data definition language, which defines the database schema to the EXODUS type manager, and a data manipulation language, E [Richardson 89], in which various application-dependent parts of the database are written by the database implementor. E is an extended version of C. These include operator methods, which define the operations available in the instance's query language, as well as access methods for data stored on the disk and possibly concurrency control and recovery "stub" functions as well. Figure 2-5 shows the modules in a typical EXODUS database and their interconnections, including both the system modules used unchanged by the implementor and E modules written by the implementor. The type manager and storage object manager, along with the front-end parser and query compiler, form the kernel of the EXODUS system and cannot be changed or replaced by the user, although they are parameterized by the database schema.

2.2.2. Iris

Iris [Fishman 89] is an object-oriented data base being developed at Hewlett-Packard (HP) Laboratories using a relational database as its back end. The Iris architecture [Wilkinson 90], as shown in Figure 2-6, consists of three layers. The central layer is the Iris Kernel, which is

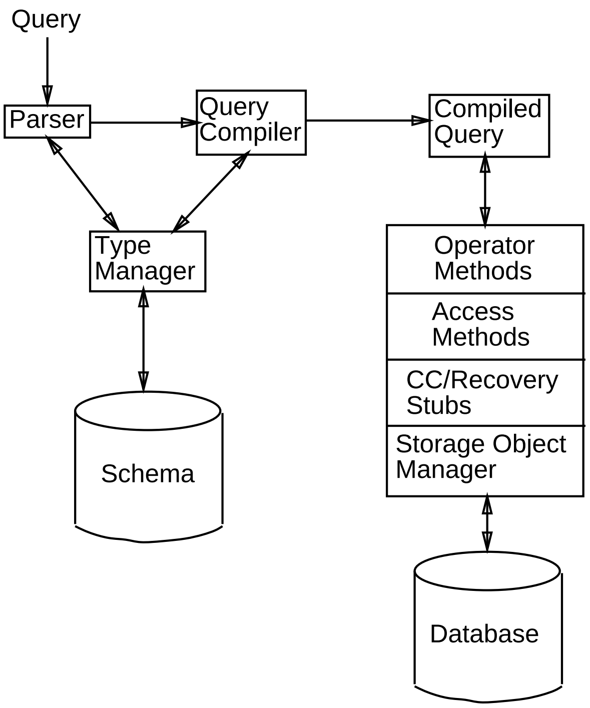

Figure 2-5: EXODUS Database System Architecture

further subdivided into several main modules. The Iris Kernel provides support for types, objects, functions, queries, updates, and versioning. It is the retrieval and update processor for the Iris data base. All object attributes, relationships, and computations are expressed in terms of functions. Functions, even those that serve to return attribute values, may have side effects; this allows Iris databases to provide the effect of active values to their users. There is, however, no predefined function providing any sort of default activity to the user. The kernel is extensible by allowing users to define foreign functions in C and other languages; these may be used in retrieval and update expressions. Below the kernel is the Storage Manager, which is actually the relational storage subsystem of the HP-SQL database, extended to support generation of unique object identifiers. The Storage Manager is very similar to System R's RSS [Blasgen 77]. Above the kernel are a set of interfaces to the DBMS, namely, the Object SQL ad-hoc query language, a graphical database editor, a C language interface, and an embedding of SQL at the statement level in various host languages.

The Iris Kernel consists of five main modules, shown inside the kernel box in Figure 2-6:

- The *Executive*, or client interface to the kernel. All requests come to the kernel in the form of queries, which are compiled into an intermediate form and then interpreted. Two buffers are returned from the Executive to the client for each query: a result buffer holding the actual result objects, and an error buffer with any error messages generated by the query. The Executive has two sets of entry points, one for clients and another for the kernel itself. The kernel entry points do less checking, to improve efficiency. They also provide access to additional functionality not provided to the user for security reasons.

- The *Query Translator*, which compiles queries into a tree format for processing. A

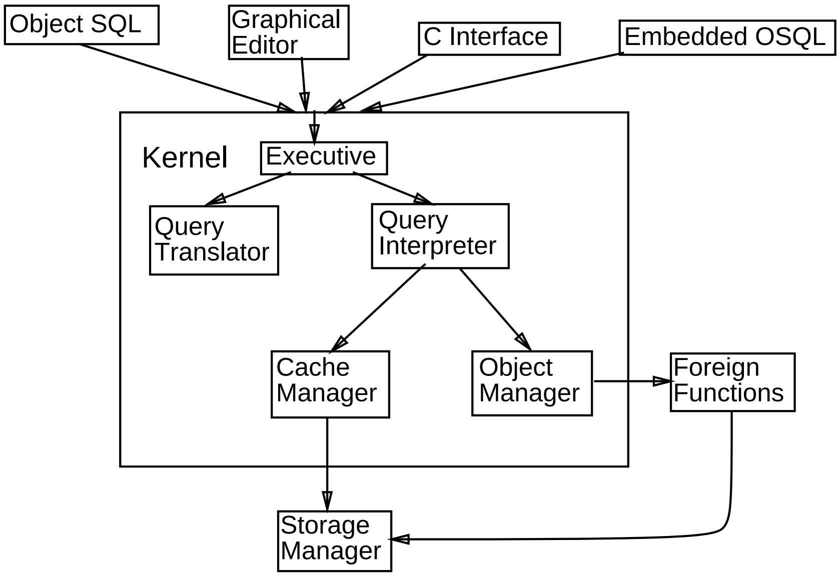

Figure 2-6: Iris Database System Architecture

query optimizer is included, and works on the tree.

- The *Query Interpreter*, which evaluates the query trees produced by the Query Translator.
- The *Object Manager*, providing access to tuples as objects.
- The *Cache Manager*, which holds the values of tables (rather than objects or functions) retrieved using the Storage Manager, and mediates between the kernel and the Storage Manager.

Iris seems to have the full capabilities of an OMS, but we have placed it with the OODB's because of the way in which it represents objects (as tuples in a relational database). It is difficult to consider a system to be an OMS when, at the lowest level, there is no single entity to which one can point and say, "This is an object". Because of Iris' non-object-based implementation strategy, spreading each object among several tables in a conventional relational database, we classify Iris as primarily a database.

2.2.3. Orion

Orion [Kim 89; Kim 90] was MCC's object-oriented database effort. A version of Orion is now sold as a commercial product by Itasca Systems, Inc. In this paper, we will consider the research version of Orion described in the published papers, rather than the commercial version sold as the Itasca database. Orion's architecture has six major components, in two layers, as shown in Figure 2-7. The lower layer has components both on the client and the server process; the upper layer resides entirely in the client.

The Message Handler is the interface between the user (or user program) and Orion. The Object Subsystem, in addition to object management functions, provides schema (type) management, query processing, and version management operations. The Transaction Manager provides serializable concurrency control and recovery. Finally, the Storage Manager provides

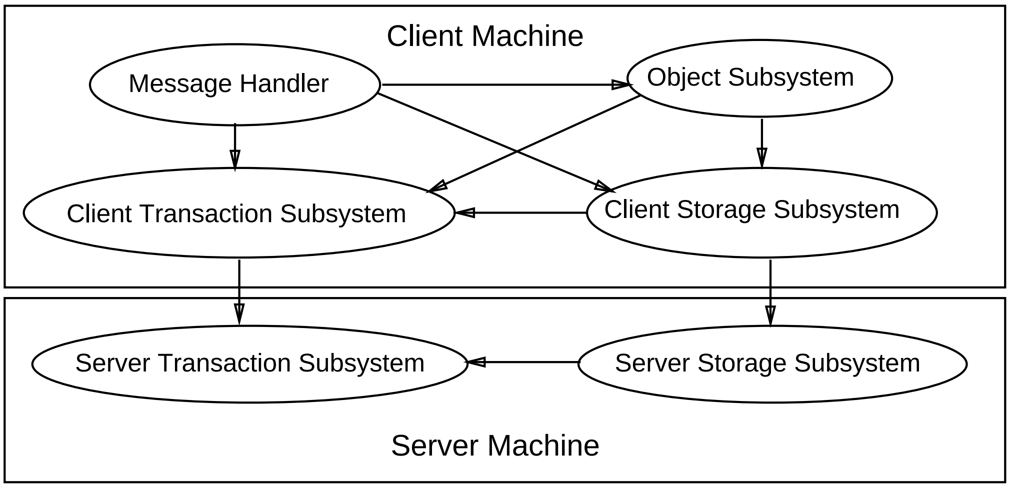

Figure 2-7: Orion OODB Architecture

access control, buffer management, and secondary storage management. The Transaction Manager and the Storage Manager are split between the client and the server, because objects in Orion are similarly divided. All objects reside permanently on the server, but may reside temporarily in the client while being accessed by a transaction. Objects are brought to the clients when accessed, modified on the client, and copied back at commit time. All locking and logging are performed on the server. This prevents difficulties maintaining consistency between multiple clients.

2.2.4. O_2

O_2 [Lecluse 90; Velez 90] is an object-oriented database system being developed by the *Altair* Group. We consider their V1 prototype version in this paper. O_2 is designed for use in a distributed environment, and its architecture is based on the client-server model, with a single server host and many workstation clients. There are multiple server processes, however; each workstation application creates a so-called "mirror process" on the server host to service its requests. There are object management components in both the client and server processes, as well as a level of caching in each. The server itself uses WiSS, the Wisconsin Storage System [Chou 85], as a storage manager. Figure 2-8 shows this situation.

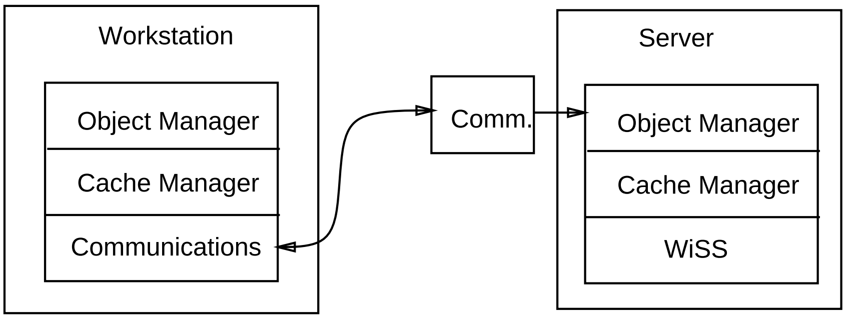

Figure 2-8: O_2 Database Architecture

Although O_2 runs in a distributed environment, and O_2 programs may run partially on the client and partially on the server, the server performs all lock management functions; its copy of each object is "primary" in a sense. Before an object may be updated, the server must be notified. This single-server model has the potential to become a bottleneck in the O_2 system.

2.3. Persistent Object-Oriented Programming Languages

A Persistent Object-Oriented Programming Language, like an Object-Oriented Data Base, also has the characteristics of an Object Server, and is often implemented using an Object Server as a back end. A persistent OOPL could also use an OODB as a back end, but the cost of using a query system to support the generally more restricted access methods of an OOPL (*e.g.*, navigation as opposed to more generalized queries) would most likely be prohibitive. The most commonly provided, and in many cases, the only, means access to objects is a simple fetch by object identifier; there is no associative access. Instead of a DDL and query facility, however, a persistent OOPL always has a DML (the programming language) for defining operations on the objects and a primitive DDL (the type declarations of the programming language) for defining the object types themselves. A "query" in most persistent OOPL's consists of navigating through the database from some starting point, retrieving a single object or a set of objects at a time by using their unique object identifiers (OIDs).

Often, both the language and the underlying persistence mechanism support concurrency. Persistent OOPL's distinguish themselves from OODB's by having a more or less conventional programming language as their DML's, by providing data abstraction features in their language, and by not providing a query language. While the OODB's DDL serves to specify the data format to the storage facility, the persistent OOPL's language also is intended to support software engineering of applications written in the language, and so provide some means of encapsulating objects. Generally, this is done by making a distinction between private and public methods and/or variables. OODB's have no such distinction, because their objects are merely data, freely manipulable from the query language and DML; code is either not stored in the database or is stored as "functions" rather than "methods", separately from the objects.

The language has a type structure, which often uses classes and single or multiple inheritance for code sharing. The type structure may, however, use delegation (for code sharing) and/or conformance (as a specification sharing mechanism). For example, Emerald [Black 86] is a persistent OOPL built on the unconventional notion of type conformance. In Emerald, an object is a member of an abstract type rather than an instance of a class. Instead of using an inheritance hierarchy to provide polymorphism, Emerald allows an object to be treated as any other type of object, providing that the types' interfaces are compatible (as determined at compile time). Basically, type *A* may be used in a context where type *B* is specified only if:

- *A* provides at least the operations that *B* provides, and
- for each operation that *B* provides, *A*'s type signature conforms to *B*'s.

Conformance between signatures is a little more complicated, and may best be illustrated by an example. Consider the case of a stack containing integers, with a *Push* operation taking an integer parameter and a *Pop* operation returning an integer. This is compatible with (conforms to) a generic source type containing only a *Pop* operation returning *any* type, since integer is a subtype of *any*; the integer stack may be used where a generic source is required. The integer stack does not, however, conform to a generic sink type containing only a *Push* operation, again because integer is a subtype of *any*; the generic sink requires that *Push* must take a parameter of *any* type, while the integer stack is more restrictive.

Emerald is interesting for this notion of type conformance, but its persistence support is very primitive, consisting only of a *checkpoint* operation to save a set of objects to disk. It also has no transaction processing capability, but only primitive synchronization operations using shared objects.

2.3.1. Argus

Argus [Liskov 88] is perhaps the best known persistent OOPL. In Argus, the underlying programming language is CLU [Liskov et al. 81]. There are two kinds of objects: small-scale CLU objects, which are not at all special in Argus, and larger *guardians*, which encapsulate a service provided by a collection of CLU objects. Guardians constitute the persistent part of the language. Guardians have *handlers*, rather than methods; a handler call is processed as an *action*. Actions are the unit of atomicity, and may be nested by calling them within other actions. Actions may also be created by using a statement-level language construct; top-level actions are usually created in this way. Argus internals used by guardians include transaction management, lock management and stable storage management. The architectural structure of Argus has not been discussed in the available literature, so we assume that each of these is simply a module within the Argus runtime system.

2.3.2. Avalon

Avalon is actually two persistent OOPL's implemented using Camelot as a back end. One uses C++ [Detlefs 88] as the base programming language, and the other uses the Common Lisp Object System [Clamen 90]. Avalon uses the Camelot primitives to provide object classes supporting atomicity and recoverability. The Avalon programmer can then use these as superclasses to implement persistent, recoverable, and atomic objects, while other objects that do not inherit from the Avalon classes are transient. Using Avalon is not quite as simple as that, however; what the Avalon classes actually provide is *primitive* support for recoverability and atomicity. The programmer using Avalon must program using these primitives, at the level of obtaining locks on individual objects, according to Avalon's protocol.

Avalon is built as a layer on top of Camelot. The language contains constructs for specifying data server interfaces, as well as for using the support in the Avalon library for Avalon's three standard classes: `RESILIENT`, `ATOMIC`, and `DYNAMIC`. The `RESILIENT` class implements recoverable objects with no concurrency control; the `ATOMIC` class is a subclass of `RESILIENT` with serializability added; and the `DYNAMIC` class is a subclass of `ATOMIC`, specializing it for two-phase read/write locking and automatic recovery. There is a considerable savings in code size and complexity, however, if the `DYNAMIC` class is used.

2.3.3. MELD

MELD [Kaiser 89; Popovich 91; Popovich 90] is a persistent OOPL that we are developing at Columbia University. MELD is a class-based language with multiple inheritance, in which classes (and thus all of their instances) may be declared persistent. Objects are multi-threaded, and concurrency is controlled by transactions. MELD's transactions are for concurrency control only; there are no recovery facilities in the current version of the system. MELD also has no query facility other than simple navigation.

MELD has two separate transaction models — distributed transactions and atomic blocks. Distributed transactions are exactly what the name implies, but use an optimistic algorithm rather than the more conventional locking approach. Atomic blocks are simple "lightweight" transactions that lock a single object for the duration of the block. Transactions may not be nested, but a single level of atomic blocks may be used within a distributed transaction to manage concurrency between the transaction's threads. Transactions are represented as a restricted form of "object" in the MELD system [Popovich 91], and work is underway on a general model of transaction objects that will allow interoperation of multiple concurrency control algorithms, each implemented in MELD as a class of transaction objects.

The MELD architecture follows the general model mentioned in Section 3, although MELD possesses only a primitive type manager. It stores enough information about the representation of each object type to allow it to check objects from the disk for validity in the current program. Its storage manager maintains a file with small pages, each of which may contain a single object or may be part of the file's index structure. The storage manager sees objects as sequences of bytes. The object manager implements object creation, message passing between objects, proxies for remote objects, and fetching of persistent objects through the storage manager, among other facilities. The lock manager is very small, and provides only mutual exclusion locks. Since MELD transactions use an optimistic algorithm, locks are not needed by transactions. The transaction manager is built on the object manager facilities for message passing between internal transaction ‘‘objects’’, and consists of a set of ‘‘methods’’ for these ‘‘objects’’.

2.3.4. Clouds

Clouds [Dasgupta 88] is actually an object-oriented *operating system*, rather than a language. However, Clouds incorporates a language, as well as transaction management on its objects, so we list it here under the heading of OOPL's. In an early version of Clouds, this language was Aeolus [Wilkes 86], developed by the Clouds project; the current language is C++-based.

The Clouds system consists of two parts, Clouds itself and the underlying Ra kernel. Each machine (or *node*) in Clouds runs the Ra kernel rather than, *e.g.*, Unix. The Ra kernel provides the basic operating system services of scheduling, segment management, interrupt processing, and virtual memory, while Clouds (outside of the kernel) provides transaction management (via a set of consistency-preserving thread types), naming services, device drivers, and network communications. Figure 2-9 shows this division of responsibilities.

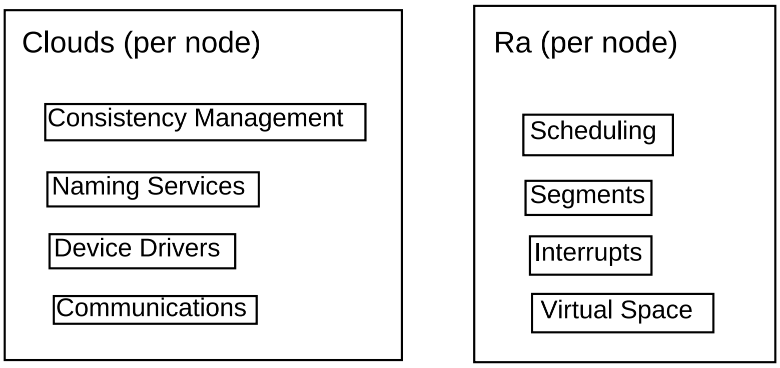

Figure 2-9: Clouds top-level modules

2.4. Object Management Systems

The term "Object Management System" (OMS) has been used to describe all of the preceding types of systems. However, in this paper, we consider an OMS to be a system with the combined capabilities of an Object-Oriented Data Base and a Persistent Object-Oriented Programming Language. An OMS has the characteristics of an Object Server, and often will be implemented, similarly to a Persistent OOPL, using an Object Server as a back end. It has both a database-like DDL and an OOPL-like DML, in addition to a query facility. An Object Management System is the goal toward which both Object-Oriented Data Bases and Persistent Object-Oriented Programming Languages seem to be working; we will describe Ontos [Ontologic 89] in this section because we believe that, of all current systems, it comes closest to meeting this goal. As previously stated, Iris also fits our description of an OMS, but was described primarily as a database because of its relational implementation.

The focus of our interest in Object Management Systems is on the issues common to all of the components of the system, most notably concurrency. The major problem with attempting to combine OOPL and OODB functionalities into an Object Management System is that existing OOPL's have one conception of what their back-end Object Servers should provide, while OODB's often have a very different idea of what their Object Server should be. For example, data in some OOPL's may contain active values; data in some OODB's may have consistency constraints placed on it. The functions of these mechanisms are very similar, but the mechanisms themselves often differ greatly in design and implementation. In an OOPL with active values, accessing an active variable generally results in a programmer-specified procedure, or other piece of code, being invoked, while in an OODB, constraints are generally stated in a declarative manner, giving the "programmer" very little control over the code that is eventually invoked to enforce the constraint. In an OMS, it is desirable to provide both functions. A rule formalism can encompass both functions in a single mechanism, and so the emerging consensus in the object-oriented and database communities, as stated in [Stonebraker 90] and [Atkinson 90], is that such a mechanism should be part of any OMS. There is not as yet, however, such an agreement on many concurrency issues.

2.4.1. Ontos

Ontos [Ontologic 89] is a commercially available OMS, built as an improvement on its predecessor OODB, Vbase [Andrews 87], by the same company, Ontologic, Inc. It provides access to objects through any of several language interfaces, C++ and Lisp among them, and interoperable locking and optimistic transactions on the objects. Ontos provides a transaction group mechanism based on that in ObServer II, thus allowing cooperative transactions. Ontos also provides a SQL-like query language for associative access to its object base; the query language is available both interactively to the end user and as a `QueryIterator` class in the programming language. Ontos is the only system (except Iris, which we "disqualify" as an OMS on the basis of its inefficient relational representation of objects) to provide both associative queries and a database navigation capability, and to allow both types of access through both a programming language and an ad-hoc query language interface. This flexibility of access to the objects, giving concurrent access from any of several programming languages as well as from a query language, is what makes us call Ontos a full OMS rather than an OODB.

Its major disadvantage is that its concurrency model is not extensible or customizable; the basic two-phase locking model (and the optimistic model, implemented as a set of "lock modes" which prevent non-optimistic access and remember how the object has been accessed) cannot be modified by the programmer to optimize performance in the application or to provide communication between cooperating transactions. The transactions must use the existing transaction algorithms, or facilities outside of Ontos, to manage concurrent access among themselves.

3. Architectures of Object Management Systems

When we examine the architectures of the systems we have discussed, it is readily apparent that they share several common components, although not all types of systems contain all of the components. For example, an Object Server or an OOPL does not have a query language processor, although such a component is a necessary part of an OODB or an OMS. It is also apparent that, in most systems, these components are connected in a similar fashion. Thus, it is natural to speak of a general architecture for object management systems. In this section, we first give a brief overview of this general architecture as we see it. We then, in a separate subsection for each common component, discuss the unusual features and the concurrency aspects of the components of each of the systems we have studied.

We have identified six top-level components of an object management system. In brief, these components are:

- A *type manager*. This is a central component of an object management system, concerned with defining all of the types of objects that are available to the system. It manages both predefined object types, if any, and programmer-defined object types. It is the type manager that defines the DDL of the system, and determines the in-memory format of the objects. Only OODB's and OMS's have type managers. If an OOPL were to provide persistent storage of class definitions as well as instances, it also would have a type manager. However, an OOPL of this sort would need only a query processor to qualify as an OODB, or perhaps even an OMS, and we have found no OOPL's without queries but with persistent class definitions.

- A *storage manager*. This component provides persistence for some of the objects or object types. The storage manager determines the on-disk format of the objects, and performs any necessary translation between memory and disk formats. The recovery facility, if present, is usually implemented by the storage manager and used by the transaction manager. The type manager also uses the storage manager to maintain persistent storage of the database schema (type specifications, *etc.*). All three classes of object systems have storage managers.

- An *object manager*. This component is responsible for creation and deletion of objects as instances of the types defined by the type manager, and for object memory management. The object manager decides when objects are to be read into memory or written to the disk, and uses the storage manager to perform these functions. To enable it to perform the memory management functions efficiently, the object manager often will take hints (from the type manager or from its own outside interface) as to object placement or clustering, which influence its calls to the storage manager. All three classes of object systems have object managers.

- A *lock manager*. This component provides primitive support for concurrency control. It is used by the object manager and the transaction manager to provide the capability to lock, for one thread or one process or one user, an object (or a set of objects, or part of an object) so as to give exclusive access. For increased concurrency or flexibility, the lock manager generally will provide other, non-exclusive, types of locks, but this is not always true. We include notification facilities, where a message is sent to tell an object's "owner" when it is accessed, as a non-exclusive type of lock. All of the systems we discuss that support concurrency have lock managers.

- A *transaction manager*. This component builds on the lock manager's facilities to provide atomic, serializable, and generally recoverable transactions on the managed objects. In some systems, the transaction manager also provides one or more extended transaction models. All of the systems we discuss have transaction managers.

- A *query processor*. This component interfaces to the transaction manager and the object manager, and provides *ad hoc* query services through a programming language and/or query language interface. This component often contains sub-components such as a query optimizer, but our focus here is on concurrency management rather than the internals of the query processing system. Only OODB's and OMS's have query processors.

Figure 3-1 summarizes the interconnections among the components, as we have presented them, in a "typical" object management system. The dotted line between the lock manager and the storage manager signifies a connection that would exist only in an OMS supporting long-lived persistent transactions, since only then would it be necessary for locks to be stored on the disk. None of the systems we studied do this.

Since the query processor has a minimal impact on concurrency except through its use of the transaction manager, and this paper is primarily concerned with concurrency, we do not discuss the query processor further here. We continue by describing the functionality and unusual features of each of the other five modules in each of the systems that have them.

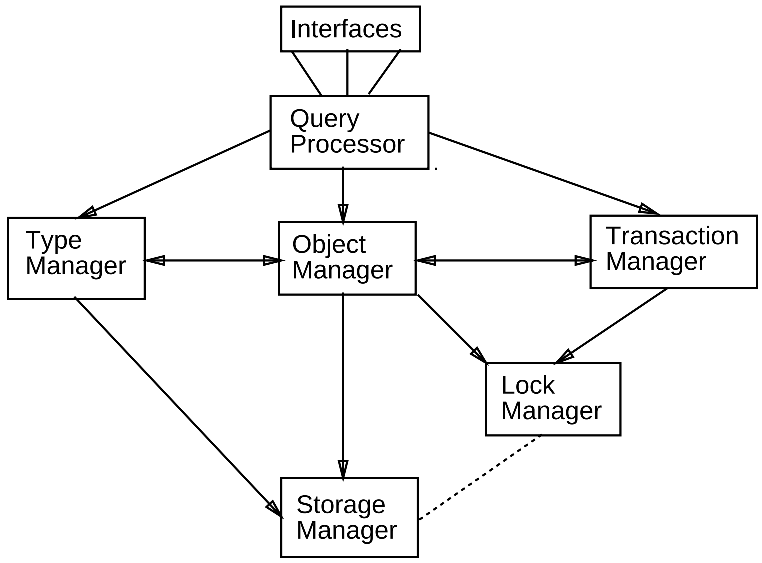

Figure 3-1: A "Typical" Object Management System Architecture

3.1. Type Manager

Type managers are present in OMS's, and also in OODB's (as database schema managers). They do not usually exist in Object Servers and OOPL's, since their main function is to provide the query processor with information about the structure of each object class. Although OOPL's have types, they generally do not store detailed information about the types on the disk, since there is no query facility with a need to know about the structure of each class of object. MELD does, however, have an extremely primitive memory for persistent types.

The type manager maintains a persistent record of the attributes, attribute types, methods, and method signatures of each persistent object class and the relations between the classes. It is used by the query processor and the object manager to determine the structure of objects that it reads from the disk. It uses the storage manager to maintain its class descriptions on disk, and in some cases represents the descriptions as objects themselves. In this last case, it also uses the object manager. Table 3-1 summarizes the main features of each object system's type manager.

The transaction server (Camelot [Spector 88]), the byte server (ObServer II [Hornick 87]) and the object-oriented operating system (Clouds [Dasgupta 88]) do not have type managers. The type manager in Mneme [Moss 88] would be part of its back end, and Basic Mneme has no type manager in its simple back end. The only type management in Argus [Liskov 88] is CLU's runtime type management; object type definitions are not stored persistently. Similarly, Avalon [Detlefs 88] provides no type management other than that normally provided by C++ (or Common Lisp). In each of these systems, the structure of persistent object classes is assumed not to change, and is compiled into each of the programs that use the classes. This prevents these systems from having any sort of schema evolution capability. We will not consider those systems further in this section.

Type Manager Features				
Feature	EXODUS	Iris	MELD	Ontos
Primitive types	Integer Real Character	Integer Real Boolean String Binary	Integer Real Char Boolean String	C++ Types
Inheritance	Multiple	Multiple	Multiple	Multiple
Constructors	record array set bag	list bag	array	Dictionary Set Array List Iterator
Definitions stored as objects?	Yes	Yes	No	Yes
Functions stored on disk?	No	Yes	No	Yes
Schema evolution support?	No	Yes	No	No
Extensible base type set?	Yes	No	No	No

Type Manager Features			
Feature	Orion	DHOMS	O_2
Primitive types	Integer Float String Boolean	Integer Character Boolean Float enum	Integer String Real Character Boolean
Inheritance	Single	Multiple	Conformance
Constructors	Set	Set List	tuple set list
Definitions stored as objects?	Yes	No	Unknown
Functions stored on disk?	Yes	No	Unknown
Schema evolution support?	Yes	No	No
Extensible base type set?	No	No	No

Table 3-1: Type Manager Summary

3.1.1. Exodus

The EXODUS type manager provides the object definitions that are used by all of the other modules in the system. The basic type structure of EXODUS is that of a classical object-oriented language with multiple inheritance. EXODUS provides the typical integer, real, character, and object ID base types, as well as constructors for record, array, set, and bag (multi-set) types. Any of these types may appear as a field in an object. An implementor of a particular EXODUS database may also add base types as primitives in the database. The EXODUS type manager considers subclassing as subtyping; instances of a subclass are considered as being members of their superclasses also, up to the root of the class hierarchy.

The type manager maintains two kinds of information about the database schema: the class hierarchy and the file catalog. The class hierarchy is rooted at the *Object* class; *Object* and the metaclass, *Class*, are predefined classes in EXODUS. Class hierarchy information is kept in objects of class *Class*. The file object, discussed further in the section on the EXODUS storage manager, is EXODUS's object grouping construct. The file catalog is simply a listing of all object groupings in the system. File objects are stored similarly to large storage objects, each of which represents any conceptual object whose representation takes more than a page of memory; large objects are discussed further in the section on the EXODUS object manager. In fact, the structural difference between file objects and large objects is a matter of a single tag bit in the object header. The B-tree index for file objects, however, is the disk page number rather than the byte number.

3.1.2. Iris

Types in Iris consist of the primitive types, known as *literal* types, and object types, known as *surrogate* types. Literal types represent themselves, while objects of surrogate types are represented by unique object identifiers. The literal types are the five base types: Integer, Real, Boolean, String, and Binary, and the two constructors, List and Bag. All other object types, and all user-defined classes, are surrogate types. The inheritance hierarchy in Iris is an acyclic graph, where a type may have multiple supertypes. This feature is not used, however, in the predefined system types, which use a single-inheritance hierarchy.

Iris is unusual in that object attributes, relations between objects, and operations defined on objects are all expressed in terms of functions. Iris functions may be many-valued and have side effects. A function has both a declaration (interface) and an implementation. This separation of declaration from implementation facilitates data independence and database evolution, by allowing types to have functions with similar or identical declarations, but very different implementations. This is the same basic concept as overloading in programming languages, but is used here as an abstraction mechanism to facilitate schema evolution.

Functions may be of any of three types: *stored*, *derived*, and *foreign*. A stored function is stored as a relation table in the database; a derived function is computed by evaluating an Iris (query) expression. A foreign function is one whose implementation is a subroutine written in a general-purpose programming language, such as C, and compiled separately from the database. There are a few restrictions on foreign functions: the user must specify to the system whether they have side effects (this information is used when compiling queries using the function), they cannot be updated by query statements (since the database cannot change a foreign function implementation, and their implementation cannot, for obvious reasons, be optimized). Some "foreign" functions are part of the system; for example, the Object Manager is implemented using this mechanism. The term "foreign" means only that the functions are not written as Iris tables or expressions. User-defined foreign functions would most likely be used to implement database extensions such as new primitive types.

The distinguishing feature of Iris' type management is its provision for database evolution. This is supported by allowing user objects to belong to any set of user defined types, and to gain and lose types dynamically. For example, an Employee object representing a given person may lose the Employee type and acquire a new type, Retiree, when the person retires from the company. At that time, the functions defined on the Employee type are no longer applicable to the object, but those defined on the Retiree type become applicable. Obviously, the Retiree type has to be very similar (in interface) to the Employee type for this transformation to be meaningful. Iris is unique among the databases we have studied in providing for schema evolution by allowing the type of an object to change dynamically. Orion also supports schema evolution, but by a different mechanism, allowing the definition of each type to change, rather than considering each type to be immutable, but allowing objects to change their types.

3.1.3. Meld

MELD's primitive types are, for the most part, similar to those in C. The C types int, float, double, char, and string (`char *` in C) are primitive types in MELD. There is one constructor, for array types. User objects are implemented in terms of these primitives.

MELD's type manager is currently in a very primitive state. MELD stores a representation of the data format of each persistent object class on disk, but uses it only to cross-check at program start that the object classes used by the program have the same format as those stored on the disk. There are no schema evolution facilities; when a class data format is changed, the objects of that class stored on disk are invalidated.

One advantage that MELD does allow is the ability to specify *constraints* on attributes of a user-defined object type. These may be expressed in terms of assignment statements (making a specified attribute change to reflect changes to other attributes), or in terms of functions or conditional statements (causing arbitrary code to be executed whenever one of a set of attributes changes). MELD constraints provide a primitive form of active data, as well as a means of expressing consistency constraints like those common in relational databases. The name, however, is somewhat deceptive, as these are not database consistency constraints. A MELD constraint is enforced in one direction only, rather than in both, like a consistency constraint. For example, a constraint "`a := b`" forces `a` to be set whenever `b` is changed, but not vice versa. Thus, we call MELD constraints "half-active" data. MELD constraints are not stored in the database, so the constraints on a class may change from time to time or from application to application.

3.1.4. Orion

In Orion, the primitive types are Integer, Float, String, and Boolean. There is one constructor type: Set. All other objects are composed of these primitives.

The database schema is represented in terms of objects. There are objects describing classes, and other objects describing instance variables and methods within the represented objects. The class objects also represent the class hierarchy. These schema objects facilitate schema evolution in the Orion system. When object type definitions are changed, new versions of the changed schema objects are created, but the old versions are kept, allowing older versions of instances of these classes to automatically be adapted to the new schema. The Orion schema evolution support allows for a wide range of simple schema changes, such as adding and deleting instance variables or changing their names, but does not easily handle basic representational changes. This is unlike Iris' support of database evolution, because while Iris considers the object as having changed its type, Orion does not allow an object's type to change, but does allow the definition of the type to change.

3.1.5. Ontos

Primitive types in Ontos are those of C++, as is the basic object model. A class library adds two constructors, for Aggregates (further divided into Dictionaries, Sets, Arrays and Lists) and Iterators (for query processing). In addition, the database aspect of Ontos makes it necessary to store the type model, or database schema, persistently. The Ontos type manager also contains the definitions of classes that represent object classes, their methods and attributes, and the multiple inheritance type hierarchy, similar in concept to those in Orion. These, like any Ontos objects, may exist in multiple versions as they are modified over time. Once an object (instance) is created, however, Ontos does not allow its type to change in the way that Orion does; any schema evolution must be performed manually, by iterating through the objects of each affected class and running code to transform each instance to an instance of the new version of the class.

Ontos represents the database schema using five object classes:

- *Type* - Represents a class definition. Contains a reference to the superclass, and to the *property* and *procedure* objects of the class.
- *Property* - Represents an instance variable. Contains the *type* of the variable.
- *Procedure* - Represents a method (or *member function,* in C++). Contains references to the method's code and argument list. Ontos uses this class to allow runtime binding of function calls.
- *ArgumentList* and *Argument* - Represent, respectively, a list of arguments and a single argument within a list. An *Argument* contains the *type* and default value of a single argument to a method.

3.1.6. DHOMS

The DHOMS type manager is part of its Object Manager (OM) module. DHOMS provides integer, character, boolean, floating point, enumerated types, and both text and binary files as basic types. Attributes may contain sets of either a basic type or an object type. The DHOMS type and object managers make a distinction that is not common to the other object systems we have studied. Because the DHOMS object base is stored and managed hierarchically, *attributes* are distinguished from *links*. The difference is that attributes specify containment of one object within another, and links do not. Thus, attributes specify the hierarchical structure; objects and their attributes form a forest, giving the database as a whole a tree structure. Links, on the other hand, specify semantic connections between objects, separate from the tree structure.

Attributes may be small, medium, or large. A small attribute may be of the usual primitive types or of an object type. An object-valued attribute implies containment; to use the terms we have used earlier, an object with object attributes is a "complex object". A sort of object-valued attribute is provided that does not imply containment; this is called a "link". There is, however, no construct for specifying links in the current Marvel DDL, so this capability has not seen practical use.

A medium attribute is an object that represents a file. It contains only the file type (*e.g.*, a binary object file). The name of the actual file is derived from context whenever it is needed. A large attribute consists of a set or sequence of attributes.

3.1.7. O_2

The usual basic types (strings, integers, reals, *etc.*) are provided as basic types in O_2. There are three non-basic types of values in O_2; *tuple-structured*, *set-structured*, and *list-structured*. Tuple-structured values are the familiar record-structured objects provided by all object systems, consisting of a set of name-value pairs. Set-structured values allow the association of an object identity with a set of objects, and may be used as attribute types in tuple-structured objects. List-structured values similarly support the use of (ordered) lists in objects. Values, as opposed to objects, do not have an identity in O_2; an object is created by taking any value and associating an object identity with it. Thus, any data structure may be constructed from either objects, if each of its component values is given an identity, or from non-object data items. This allows representation of complex objects in O_2.

There are no classes as such in O_2, and thus no inheritance; instead, there is a kind of "specification inheritance" related to the conformance concept discussed in Section 2.3. Instead of having methods with the "receiver" of the method distinguished from the other parameters of the method, O_2 has multi-methods — the types of all parameters are significant for the purposes of method dispatching. An object subtype may specialize its methods by adding new arguments and/or making the types of its arguments more restrictive. At runtime, the most restrictive method that fits the arguments is the one chosen to run.

Although the existence of an ad-hoc query language for O_2 implies that the database schema is represented on disk, we have found no information in the published literature to indicate how it is represented, or whether it is represented as O_2 objects. We cannot, therefore, comment on O_2's management of the database schema.

3.2. Storage Manager

The storage manager is responsible for disk storage and retrieval of a set of collections of data that comprise the objects stored by the system. These collections are stored either as undifferentiated sequences of bytes or as sets of attributes with known types. At this level, however, the collections are not considered to have any semantics as objects. Object semantics are implemented at a higher level, by the object manager; the storage manager's sole responsibility is the management of its memory space and disk storage space for object representations. Table 3-2 summarizes the features of the storage managers in the systems we have studied.

Camelot provides no storage manager of its own; it is designed to be used with storage management provided by another server. Thus, we will not consider Camelot in this section.

3.2.1. ObServer II

Objects in ObServer II are stored in *segments* on the disk. An object may be contained in more than one segment. Each segment is stored in consecutive pages on the disk, and space for segments is allocated by a first-fit algorithm. Disk fragmentation is kept to a minimum by compacting the segment partition whenever the server is idle, or at regular intervals. An extendible hash table is used to store the correspondence between segment IDs and segment locations, so as to guarantee access to any segment in two disk accesses.

The correspondence between objects and their unique identifiers is maintained in two files on the disk. One, the *Object Location Table* (OLT), maps each logical object ID to a (segment, size, offset in segment) triple. The other, the *Duplicate Object Table*, handles objects that are contained in more than one segment. It maps a logical object ID to a set of logical UIDs, one for each copy of the object, that is, one for each segment in which the object is contained. When an object is deleted, its space in the segment is freed and a special OLT entry, called a *tombstone,* is written for it. Object identifiers are never reused.

3.2.2. Mneme

The storage manager provided in Basic Mneme supports the notions of *files* and *logical* and *physical segments* [Moss 90]. A file is just that, a disk file containing a collection of physical segments. All information transfer between memory and disk is done in terms of physical segments. Each physical segment contains several logical segments. Logical segments are the means whereby the programmer groups related objects for storage near one another on the disk, while physical segments are the unit of transfer between memory and the disk. Logical segments consist of a table of objects plus space for the objects themselves. Object identifiers have three fields of ten bits each, one for the file, one for the logical segment, and one for the object. Objects are accessed by looking them up in a series of three tables organized on these fields: A global file table, a logical segment table within the file, and an object table within the logical segment. It is not clear how Mneme deals with fragmentation within logical segments.

Storage Manager Features						
Feature	ObServer	Mneme	EXODUS	Iris	Argus	Avalon
Representation	Bytes	Bytes	Bytes	Relational Tables	Log Records	Log Records
Storage Medium	Disk Partition	Files	Files	Relational Database	Log File	Log File
Object Space Reclamation	Compact When Idle	Unclear	May vary	Irrelevant	N/A	N/A
Fetch key	Unique ID	Unique ID	Unique ID	Associative Access	None	None
Recovery Method	None	None	Logging	Logging	Logging	Logging
Query Support	None	None	File Scan	Relational with tuple & predicate caches	None	None

Storage Manager Features						
Feature	MELD	Ontos	Clouds	Orion	DHOMS	O_2
Representation	Bytes	Proprietary	Virtual Memory	Bytes	Bytes	Records
Storage Medium	File	Files	Memory & Backing Disk	Disk Partition	File	Files
Object Space Reclamation	None	Proprietary	Free Pages	Best Fit	Handled by dbm	Handled by WiSS
Fetch key	Object and class name	Unique ID	None	Unique ID	Object and class name	Unique ID
Recovery Method	None	Logging	Shadow Pages	Logging	None	Unknown
Query Support	None	Proprietary	None	Indexes based on attributes	Iteration through class instances	Indexes

Table 3-2: Storage Manager Summary

3.2.3. Exodus

Storage objects in EXODUS are simple byte sequences of arbitrary length, similar to the objects provided by an Object Server to its users. At this level, storage objects are completely passive, since any semantics are defined at a higher level, in the object manager. Storage objects may be *large* objects, consisting of a tree structure of components; no cross-links are allowed. Large objects are discussed further in the Object Manager section. We have made an division in this paper between the Storage Manager and the Object Manager, and consider structuring of complex objects to be the province of object rather than storage management. EXODUS, however, combines these two modules into a single Storage Object Manager.

The EXODUS storage manager provides a predefined access method (B+-trees) and a set of "stub" functions for two-phase locking with a conventional logging scheme for recovery, but the implementor of an EXODUS database may replace both the storage access and concurrency control/recovery policies. Note that both must be replaced at the same time, since the two policies are combined in one set of functions. While their design, allowing replacement modules for these to be written in E, is no doubt quite general and should allow implementation of any imaginable access method or concurrency control scheme, Carey *et al.* acknowledge in their paper [Carey 86] that programming some of these modules (particularly concurrency control stubs) is likely to be very difficult and unlikely to be undertaken by users of the EXODUS system.

The EXODUS storage manager allows grouping objects together into a *file object*. There may be several reasons for creating a file object. One is that objects in the same file are located near one another on the disk. The other main reason for creating a file object is closely related; EXODUS provides a mechanism for sequencing through the objects in a file (*e.g.*, to process a query), and so sets of objects that are likely to be accessed in a single query should be grouped using a file object. A file object may contain objects of only a single class, but this is not seriously restricting to the database implementor, since EXODUS considers objects of a subclass to be objects of any of their superclasses as well. Thus, a file of *Object*s may contain any objects in the database.

3.2.4. Iris

The Iris storage manager, as previously stated, does not represent objects as such, but instead represents them as tuples in conventional relational database tables, with some extensions to support object identity. Tables may be created and dropped from the database dynamically. The storage manager provides commands to retrieve, update, insert and delete tuples. Tuples may be retrieved using indices allowing access to the tuples of a table in a predefined order, and a predicate can be defined to limit the tuples that will be retrieved. A scan operation is also provided, giving associative access to multiple tuples in a single request.

The Cache Manager, which is above the storage manager but below the object manager, could be considered to be part of either the storage manager or the object manager module. The Cache Manager provides two kinds of caches, *tuple caches* and *predicate caches*. Tuple caches, at most one per table, hold tuples from a particular table and are accessed by column. These are true caches, holding the most recently referenced tuples from the table. Many predicate caches, on the other hand, may exist for a single table. A predicate cache contains a set of tuples from a table that satisfy a particular predicate, and serves to hold intermediate results during query interpretation. The Cache Manager is not configurable, and the user cannot affect cache management policy.

3.2.5. Argus

Argus does not store objects as such on disk. To implement stable storage, Argus uses only logging. Argus makes a distinction between *data* log records, which record changes to data values, and *outcome* log records, which record the processing state of the two-phase commit protocol. After a crash, the log is processed in reverse order, starting at the last outcome record. Subsequent changes to data are ignored. The entire log is processed. This is kept from taking an excessive amount of time by periodically writing a snapshot of each guardian to its log; this begins a new log with the current state of the guardian. This housekeeping is performed in background mode when a guardian is idle. Thus, a very active guardian may take a large amount of time to recover after a crash, but a guardian that has been inactive will have had a snapshot written ‘‘recently’’ and so should take very little time to recover. The developers of Argus experienced no major problems due to this recovery strategy, but to our knowledge they never implemented any applications with a large stable state.

3.2.6. Avalon

In Avalon/C++, storage management is provided by the RESILIENT class. Its interface consists of two operations: `pin` and `unpin`. These implement a write-ahead locking protocol [Gray 78]. Before accessing an object, it is pinned in memory. The modifications are then made, and the unpin operation called. This logs the changes on stable storage (through a mechanism that is not clear in the published papers), and unpins the object so that its memory can be reused if necessary.

3.2.7. Meld

The MELD storage manager provides disk storage and retrieval of variable-length byte strings by a single key string. This simple facility is used by the object manager to provide object storage. There is a cache of recently referenced keys and their contents. This, along with the fact that MELD does not lock objects on disk, only in memory, presents a potential problem if multiple MELD programs should happen to read the same object from disk. Steps are taken in the object manager, however, to prevent this; if an object is in memory anywhere, this will be known by the network name service. The object will then be referenced remotely rather than reading in a second copy from the disk.

3.2.8. Ontos

The Ontos storage manager provides persistent storage for C++ objects. Since Ontos is a commercial product, and storage management is probably the single most crucial place where proprietary algorithms and optimizations can affect the performance of a database, Ontologic Inc. has said little about how their storage manager operates and the way that it indexes its data. Similarly, the available literature does not describe any caches that might be used or their management policies.

3.2.9. Clouds

In Clouds, the basic storage abstraction is the virtual address space. Some address spaces are persistent; these constitute Clouds' *object memory*. Storage management consists of a fairly conventional implementation of virtual memory, with the addition of shadow pages for use by the transaction processing subsystem. In contrast to more conventional operating systems, the disk is used only to provide backing for persistent virtual address spaces; there is no notion of a file in the system. There are three types of address spaces: *O space*, an object storage space; *P space*, where private data entities of a particular thread (*e.g.*, stacks) are stored; and *K space*, which holds the Ra kernel and system objects. Object management is built on top of this virtual storage support.

3.2.10. Orion

The Orion storage subsystem manages disk storage on a raw disk rather than in a single file. The disk has a header, log space for the recovery algorithm, an object directory, and a set of partitions containing the actual objects. Partitions are divided into segments, which in turn each contain a collection of pages. Each object lives on a single page, and a page may contain multiple objects. A single object may not be larger than a page; any conceptual object that would be larger must be represented as a composite object. Pages have a header, with information such as the amount of free space on the page, and the location within the page of the free space to be used for new objects. Pages also contain a location for each object, with a pointer to the object within the page. Access to objects is indirect through this pointer, allowing objects to be moved by the allocator. The page number and slot number of each object is stored in the object directory, providing fast access with one disk read (assuming the directory is already in memory) and an indirection to on-disk objects.

Object lookup is done in Orion using indices based on single attributes and rooted at a particular class in the class hierarchy. An index holds entries for all objects of that subclass and (recursively) its subclasses. This inclusivity contrasts with the more common case, where an index on an attribute contains only objects of a single class, and multiple indices must be searched if more than one class is to be considered. Including the subclasses in a single index saves space for the indices, and saves time on large queries. For small queries, however, this is more expensive than having a separate index for each subclass.

3.2.11. DHOMS

Storage management in DHOMS is performed by the Storage Manager (SM) and File Manager (FM) modules. The Storage Manager maintains disk storage for all DHOMS objects except file objects; these are the responsibility of the File Manager.

The database schema is stored in two parts within a single file. One part is the process model, expressed in a rule-based formalism. The other part is the data model, which consists of a list of classes, each with its attribute names and types and its superclass names.

The objects themselves are stored in a separate file. In the current implementation, this is a binary file in the format of the Unix *dbm* utility; DHOMS uses *dbm* to maintain its object base.

There is no faulting in of objects from disk in DHOMS; instead, the entire object file is read into memory at startup, with DHOMS asking *dbm* for each object individually. The objects remain in memory (although updates are also written to disk) throughout operation of the server. An upcoming version is expected to provide on-demand access to disk objects.

3.2.12. O_2

In O_2, the WiSS system functions as the storage manager, providing persistence, disk management and concurrency control support for O_2. In this section, we are concerned only with the first two of these areas, deferring discussion of concurrency control to later sections.

WiSS supports a single basic unit of persistence and I/O, the page, and persistent structures based on this support. These structures include record-structured sequential files, long data items and B-tree or hash-based indices.

WiSS maintains a cache of recently referenced pages. These, like the pages actually on the disk, are represented in a format which is more compact than the in-memory object format. The transformation of objects to the in-memory format is the responsibility of higher levels in the server, and the storage manager has no knowledge of O_2's object format.

The WiSS storage manager resides entirely on the server, and is hidden from the client by higher levels of the O_2 system.

3.3. Object Manager

Camelot, being purely a transaction server, has no object manager. It simply provides transactions on collections of bytes with no particular structure. All of the other systems, however, contain object management facilites.

The object manager implements an object abstraction using the facilities of the storage manager. It is responsible for providing all of the primitive operations on objects, as well as any storage management functions that depend upon semantic knowledge about the objects, such as clustering decisions. Table 3-3 summarizes the features of the object managers we studied.

3.3.1. ObServer II

In ObServer II, an object is simply a collection of data bytes. An object has an identity, but no internal structure. An object is always locked and accessed as a unit. As ObServer II has no type manager or type structure, there are no classes of objects. ObServer II object management is very simple, with operations provided to create, destroy, and update the objects. Objects are entirely passive, except that the transaction support may activate an object on access to shared data, through the "soft lock" notification mechanism.

3.3.2. Mneme

In Mneme, objects are treated in much the same way as in ObServer II. However, objects are not simply undifferentiated collections of data. A Mneme object has both *slots* and *bytes*. Slots hold references to other objects, while bytes hold non-object data. There are separate operations in the Mneme object management interface for accessing slots and bytes. Object IDs may only

Object Manager Features						
Feature	ObServer	Mneme	EXODUS	Iris	Argus	Clouds
Structure	Bytes	Slots and Bytes	Bytes	Relational Tuples	CLU Objects	Code & Data Segments, Persistent & Volatile Heaps
Active Objects?	No	No	No	No	Yes	Yes
Single or Multi-Threaded?	N/A	N/A	N/A	N/A	Multi	Multi
Distribution Support?	No	No	No	No	Yes	Yes
Multiple Versions?	No	No	Yes	No	No	Shadows
Clustering Method	Hand Prefetch	Pool Strategy	None	None	None	None
Configurable?	No	Via Strategy	Yes, but difficult	No	No	No
Complex object support?	No	No	"Large" Objects	Yes	No	No

Object Manager Features						
Feature	Avalon	MELD	Ontos	Orion	DHOMS	O_2
Structure	C++ or Lisp Objects	Attributes	Attributes	Attributes and Components	Attributes and Links	Attributes
Active Objects?	No	Half-Active	No	No	Rules	No
Single or Multi-Threaded?	Multi	Multi	N/A	N/A	Single	N/A
Distribution Support?	Camelot	Yes	No	Client/Server	No	Client/Server
Multiple Versions?	No	No	No	No	No	No
Clustering Method	None	None	Collections	By Class	None	Placement Trees
Configurable?	No	No	Yes	No	No	No
Complex object support?	No	No	Yes	Yes	Yes	Yes

Table 3-3: Object Manager Summary

be stored in slots, so unlike ObServer II, Mneme "knows" which objects refer to other objects and can use this information in its object management strategy and for garbage collection. Objects also have *attribute bits*, which when set signify special properties of an object, such as being read-only or being a *forwarder*. Forwarders are proxies that implement references to objects across file boundaries. Mneme objects, like ObServer II objects, are not active entities.

There is no single object management policy in Mneme; instead, there may be a set of *pools*. Each object is a member of exactly one pool. Each pool has a *strategy* associated with it. The strategy consists of a set of routines, one for each event that can happen to an object. These routines are called automatically whenever the specified event happens to an object in the pool. For example, strategies can include routines called on object creation, object deletion, faulting an object into memory from the disk, accessing a slot in an object, changing the value of a slot, and many other such operations. Strategies can facilitate clustering of related objects, where the strategy has knowledge about a certain class of objects and makes decisions as to which objects associated with the class instances should be clustered with the instances. Strategies might also be used to implement various memory management strategies, such as least-recently-used or least-frequently used, for the object memory. Finally, strategies are also used as the implementation mechanism for concurrency control, with locks being obtained as side effects of accessing objects. The default strategy is to have no side effects to any object events.

3.3.3. Argus

Object management in Argus is basically that provided by CLU, its underlying language. There is one important difference, however; Argus guardians are *active*. Each guardian contains not only data, but also multiple threads of control. The data in a guardian consists of a collection of (relatively) small CLU objects and the guardian's stable variables, all allocated on a stable heap (recoverable memory) provided by the storage manager. There are two ways that a thread can be created within a guardian. Each call to a *handler*, a method of a guardian, creates a thread. Additionally, a guardian may contain *background threads*, started at its creation and running indefinitely, independent of handler calls. This ability to have objects containing threads of control distinguishes Argus' object management from that of ObServer II and Mneme, as well as from that of conventional programming languages.

3.3.4. Exodus

There are two unusual features of object management in EXODUS. First, EXODUS objects may be either *small* or *large*. Small objects are those that fit on a single disk page; large objects are those that take more space to store. Large objects are stored as a B+-tree structure of pages, indexed by byte number within the object. Second, objects may be versioned; the size of versions of "large" objects is kept to a minimum by copying only the components of the object that have changed from one version to the next.

There are two types of versions, *working* and *frozen*. Frozen versions are immutable, while working versions may be changed. Once a working version is frozen, it cannot be made back into a working version, but new working versions may be *derived* from it. Deriving a version consists of copying its root node and marking all of its subcomponents as shared. Changes may then be made to the derived version. When changes are made to a shared node, a private copy is made of the node; the changes are actually made there, rather than in the shared copy, which remains unchanged until it is eventually deleted. This treatment of shared nodes as immutable avoids the problem of having a change to one version of an object show up in another version.

The EXODUS "storage manager" module provides operations to manage these facilities, as well as data access operations analogous to those provided by an object server, to the higher levels of each EXODUS database.

EXODUS storage objects are accessed by using an object identifier (OID) consisting of a page number on disk and a slot number within the page. This is, of course, a unique identifier, but it is also tied to the physical location of the object on the disk. An object cannot be moved without updating all references to it in the database, so EXODUS objects generally do not move, even if they grow larger than a page and so must be represented as large objects. In this case, the large object header replaces the original small object on disk and new pages are allocated to hold the content of the object. Physical OIDs were used in EXODUS to obtain a performance advantage over logical OIDs, which must be translated every time they are used.

3.3.5. Iris

In Iris, the Object Manager is a layer of system foreign functions, providing access to the underlying relational database implementation in terms of objects. It was implemented this way partly for efficiency reasons, and partly because the relational algebra was not sufficiently powerful to express some of its functions. The Iris group claims that work is underway on extending the relational algebra to allow many functions of the Object Manager to be implementable in it as database functions rather than foreign functions [Wilkinson 90].

The Object Manager provides a typical set of object management functions: Object creation and deletion, reading and updating object attributes, *etc.* Programming language access to objects, as opposed to associative query access, is provided by Object Manager operations, though the interface still goes through the query processor.

3.3.6. Clouds

A Clouds object consists of one or more global data segments, a persistent heap for dynamically allocated data, a volatile heap for a collection of locks and capabilities held by the object, and one or more code segments. The code segments contain both system services and user-defined operations. All of the persistent information is stored in an O space provided by the virtual memory support. The conventional object management operations are supported, and simply work in the O spaces provided by the storage management component.

3.3.7. Avalon

Avalon’s object management is simply that of the underlying language (C++ or Lisp). That is, object creation and deletion are provided by the language, and object naming is not provided by either Camelot or Avalon. The name service is a separate facility of the Mach operating system, and is used directly by implementors of object servers.

3.3.8. Meld

The MELD object manager provides only the most basic object management operations: Creating objects, and searching for remote and persistent objects by contacting a name service and by using the local storage manager if the name service fails to locate the object. An object name may be specified when an object is created; this, combined with the name of the object's class, forms the key string for the storage manager. It is also the name used for referencing the object through the name service when a `get` call is done to find the location of an object somewhere on the network. MELD does not generate unique object identifiers. It also does not currently allow deletion of objects, due to difficulties in determining when execution within an object has terminated. These difficulties are due to the use of a dataflow execution model; they will be removed (along with the dataflow model) in work currently in progress on the next version of MELD.

Objects in MELD are read into memory by the *create* and *get* operations. *Get* first queries the name service to locate the object; if not found, it searches the persistent database on disk. *Create* is like *get*, except that, while *get* returns a null result if the object cannot be found, *create* will create the object if it is not found. These operations are called as pseudo-methods of the object's class.

At the time an object is read into memory, all objects referenced by it are also read into memory. Actually, the transitive closure — all objects reachable from the object — are read into memory. This is necessary because MELD represents all references within objects as pointers, and there is no object faulting mechanism. It makes MELD potentially very slow on the first access to an object, but fast on subsequent accesses.

3.3.9. Ontos

The Ontos object manager provides the expected operations, plus a means for clustering objects. A cluster is an arbitrary collection of objects, with a name for use in referencing the cluster. It is not itself an object. The object manager provides three options to the application for bringing objects into memory: A single object may be read using the `VB_getObject` function, a cluster may be read using the `VB_getCluster` function, or finally, the closure of all objects reachable from an object may be read all at once using the `VB_getClosure` function. To support faulting in of objects not read when the `VB_getObject` or `VB_getCluster` functions were used, Ontos supports a special sort of *transparent reference* in addition to *direct references*, or pointers. If an object accessed through a transparent reference is not in memory at the time of the access. it is brought in automatically.

The object manager also allows the Ontos programmer to control the allocation of memory for objects. All functions that create objects or bring them into memory have arguments for a pointer to the memory to use to hold the object and the size of the block of memory. This allows the programmer, among other things, to allocate objects contiguously in memory, when desired for efficiency reasons.

3.3.10. Orion

Orion objects are small, residing on a single page of memory. However, objects in Orion may be composite. Composite objects consist of a hierarchy of component objects, which are like any other objects except that their existence depends on the existence of their containing (or parent) object. When a composite object is deleted, all of its components are deleted with it.

Clustering in Orion is simple. Objects of a single class are by default clustered in a segment on disk. There is a `Cluster` message defined by Orion; with this message, the programmer specifies a list of classes to be clustered in the same segment. This is useful, for example, in a parts hierarchy, where the programmer knows that a `Body` object will always be part of a `Vehicle` object, and so the `Body` objects should be stored with the `Vehicle` objects. Aside from this circumstance, however, Orion provides no way to cluster objects with respect to the relations between them.

The buffering of objects in Orion is divided between the object manager (on the client machine) and the storage manager (on the server machine). The storage manager buffers recently-used pages, while the object manager buffers recently-referenced objects. Interchange of objects between the client and the server is done in the disk format, and the conversion to and from in-memory format is done at the client. This gives the server faster response than in systems where the conversion is done on the server, and since the objects must be linearized in any case for transmission, would seem to be the best choice for performance. However, it does have the disadvantage of making the client more complex and spreading the knowledge of the disk format.

3.3.11. DHOMS

DHOMS objects consist of a list of named, typed attributes. The object manager contains a hash table which is used to convert key strings (object names) into object identities. There is also a lock table, which is a hash table of objects and locks. Multiple locks are kept in a list for each object. This resides in the object manager, rather than in the lock manager, in order to shield the lock manager from the details of conflict resolution; this is the object manager's responsibility. Each lock is represented as a pair, containing the object identifier and the owning transaction identifier. This maintains state information on the locks of all objects for use by the lock manager. There are no semantics associated with these locks by the object manager, however, since doing so would infringe on the transaction manager's authority to resolve conflicts between transactions.

3.3.12. O_2

O_2 objects, as stated earlier, may be *atomic*, or may be structured as *sets*, *lists*, or *tuples*. O_2 uses *physical* object identifiers, for efficiency reasons. There are two types of object identifiers in O_2. A disk object identifier consists of a volume identifier, a page number within the volume, and a slot number within the page, and represents a persistent object. A virtual memory object identifier consists of an address and a tag bit indicating whether the object is stored on the client or the server; these represent volatile (non-persistent) objects. Persistence is not an attribute of a type or of an object in O_2, but rather is conferred by reachability from a programmer-specified set of objects and values.

When a client wishes to access an object in O_2, the first step is to check its own local object cache. If the object is not found, a request for the object is sent to the server, and another layer of cache search is performed, after which it may be necessary to read the object from the disk. In any case, the object is copied from the server to the client, but the server retains concurrency control over the object. If the client later wishes to update the object, the object manager will make an explicit request to promote the lock on the page in which the object resides to a write lock. This is possible because, as stated above, the object identifier contains the exact location of the object.

O_2 allows the programmer (actually, one particular programmer -- the DBA) to specify clustering of object on disk by constructing *placement trees* [Benzaken 90]. These specify, for each class at the root of a placement tree, a set of attributes of specified classes that should be clustered with each instance of the class. Because it is a tree, it can also specify further clustering of the children's attributes, to any finite depth. Each class may be the root of at most one placement tree, so there can be no conflicts at the root level. Conflicts may occur at lower levels of the tree. It is not clear from the literature how they are resolved, but our intuition is that the largest cluster is chosen, based on their description of the algorithm as "a greedy tree pattern-matching algorithm."

3.4. Lock Manager

Avalon, being built on top of Camelot, uses Camelot's locking primitives rather than implementing its own set. Because of this, we will not discuss Avalon in this section. All of the other systems, however, provide lock management capabilities.

The lock manager provides exclusion primitives to the transaction manager, and often to the DML as well, as in ObServer II, MELD, and several other systems we have studied. These include, at the least, mutual exclusion locking (as in MELD), with shared read/exclusive write locks very common (particularly in OODB's like EXODUS, Iris and Orion). OMS's (such as Ontos) and other systems intended for advanced applications (such as ObServer II) generally provide "soft" or communication-based locks as well. Table 3-4 summarizes the capabilities of the lock managers in the systems we have studied.

3.4.1. ObServer II

In ObServer II, a lock has two attributes: a *lock mode* and a *communication mode*. The lock modes of ObServer II include the usual shared read/exclusive write locks, called here "restrictive read" and "restrictive write". There are also several more relaxed lock modes:

1. Non-restrictive read, which allows concurrent non-restrictive read and non-restrictive write operations.

2. Non-restrictive write, which allows only non-restrictive reads concurrently.

3. Multiple write, which is like non-restrictive write except that it also allows multiple writers (all using multiple write mode).

4. And a null lock mode, provided to allow use of communication modes without

Lock Manager Features						
Feature	ObServer	Mneme	EXODUS	Iris	Argus	Clouds
Mutual Exclusion?	Yes	No	Yes	Yes	Yes	Yes
Shared Read / Exclusive Write?	Yes	No	Yes	Yes	Yes	Yes
Soft Locks?	Yes	No	No	No	No	No
Granularity	Object	Slots or Larger	Byte Range Within Object	Tuples and Tables	Object	Page
Placement	Server	In Strategy	Server	Server	On Object	Page Table

Lock Manager Features						
Feature	MELD	Ontos	Orion	Camelot	DHOMS	O_2
Mutual Exclusion?	Yes	Yes	Yes	Yes	Yes	Yes
Shared Read/Excl. Write?	No	Yes	Yes	Yes	Yes	Yes
Soft Locks?	No	Yes	No	No	No	No
Granularity	Object	Object	Objects and Components	Object	Objects and Components	Object
Placement	On Object	Server	Server	Separate Server	Server	Server

Table 3-4: Lock Manager Summary

locking the object.

Communication modes facilitate cooperation between transactions by allowing a lock holder to be notified whenever certain events occur on the locked object. These include updating by another transaction as well as inability of another transaction to acquire a requested lock (further divided into read-lock, write-lock, and either-lock notification). Combinations of the three lock acquisition communication modes with the update communication modes are also available. Finally, there is a null communication mode, to be used when standard locking without notification is desired. These lock and communication modes are summarized in Table 3-5. Some combinations of lock and communication modes are semantically invalid, *e.g.*, a restrictive read lock with update notification. Valid pairings of lock and communication modes are shown as *V* in Table 3-6; invalid pairings are shown as *I*.

ObServer II also provides a non-blocking lock option. When a transaction attempts to acquire a lock, it states whether it is willing to wait if the lock is unavailable. If it will wait, the unsuccessful lock request is queued; otherwise, the request is denied immediately.

Lock Modes

N	No lock (notification only)
NR	Non-restrictive read; allows concurrent non-restrictive reads and non-restrictive writes.
RR	Restrictive read; allows concurrent reads, but no writes.
MW	Multiple write; allows concurrent non-restrictive reads and multiple writes (all using *MW* mode).
NW	Non-restrictive write; allows concurrent non-restrictive reads.
RW	Restrictive write; enforces exclusive access.

Communication Modes

N	No notification (lock only)
U	Notify lock holder on update.
R	Notify lock holder when an attempt to acquire a read lock fails.
W	Notify lock holder when an attempt to acquire a write lock fails.
RW	Combines *R* and *W* modes.
UR	Combines *U* and *R* modes.
UW	Combines *U* and *W* modes.
URW	Combines *U* and *RW* modes.

Table 3-5: ObServer II Lock and Communication Modes

	Communication Modes							
Lock Modes	N	U	R	W	RW	UR	UW	URW
N	I	V	I	I	I	I	I	I
NR	V	V	I	V	I	I	V	I
RR	V	I	I	V	I	I	I	I
MW	V	V	V	V	V	V	V	V
NW	V	I	V	V	V	I	I	I
RW	V	I	V	V	V	I	I	I

Table 3-6: ObServer II Lock and Communication Mode Pairings

The lock management features of ObServer II give a programmer considerable flexibility in managing concurrency. The price of this flexibility is that all locking calls are made directly by the using program, rather than being managed by the transaction subsystem (except for releasing locks when a transaction finishes). Although this does not prohibit using multiple transaction types in an application, it makes writing transaction code (particularly when using both hard and soft locks) rather cumbersome and unwieldy.

3.4.2. Mneme

Since Basic Mneme is a single-user system, it incorporates no locking mechanism. The Mneme design, however, intends to allow lock management to be implemented as part of an object management strategy. There are no explicit locking operations in the Mneme interface; instead, locking of an object is controlled by the object management strategy routines in the object's pool. Locks are intended to be acquired as side effects of accessing slots within objects. They are to be released as side effects of finishing a transaction on their object. Since there are many pools, there may be many different locking strategies in a Mneme application, with each in use on a separate set of objects. Moss and Sinofsky mention, in particular, the use of hard locks and two-phase locking on a pool of frequently accessed objects and soft locks and an optimistic algorithm on another pool of less frequently accessed objects.

So, while Basic Mneme has no lock manager, Mneme has the potential eventually to provide equal flexibility to ObServer II while maintaining the advantage of concealing lock management operations entirely from its client programs.

3.4.3. Camelot

The Camelot lock manager is part of the module that they call a transaction manager, but since it seems easily separable conceptually from the transaction manager, we describe Camelot lock management here rather than in the transaction manager section.

The locks provided by Camelot are the typical shared read/exclusive write locks. Both blocking and non-blocking operations are provided for acquiring locks. Camelot locking is distinguished from other systems by the fact that Camelot locks are not physically associated with the object being locked.

In Camelot, data is kept in data servers, separate processes from the transaction server. Locks, however, are kept with the transaction server. Accordingly, locks cannot be placed directly on the data objects. Camelot thus has the idea of a *lock name* in a *lock name space*. A lock name is a 32-bit integer, and uniquely identifies an object for the purpose of locking. The Camelot library includes a `LOCK_NAME` primitive for generating a lock name for an object that is guaranteed distinct from the lock name of any other object. Alternatively, the programmer may manage the lock name space by hand, generating lock names arbitrarily.

Ordinarily, locks are automatically released when the transaction holding them completes (whether it commits or aborts). In addition, the Camelot library provides an `UNLOCK` and a `DEMOTE_LOCK` (from a write lock to a read lock) primitive. This allows the programmer to relax Camelot's normally strict two-phase locking. To support nested transactions, Camelot uses a lazy lock propagation algorithm based on lock propagation in Argus. The Argus lock manager section (Section 3.4.6) provides more details on this algorithm.

3.4.4. Exodus

Locks in EXODUS are of the conventional shared-read/exclusive-write sort. EXODUS is unusual, however, in that a locks can be placed not only on an entire object, but on a range of bytes within an object as well. This range might, for example, include a set of related slots within the object. This increases concurrency by allowing two or more transactions simultaneous access to disjoint ranges within a single object.

Since EXODUS provides large (but not composite) objects, it must deal with the concurrency control problems created by insert, append, and delete operations on large objects, which may change their structure. EXODUS uses a well-known non-two-phase B+-tree locking protocol [Bayer 77] to preserve the consistency of its trees when the structure of a large object may be changed.

3.4.5. Iris

Since Iris is based on a relational back end, it does not lock objects in the sense that the other systems do. Locks on tuples and tables are provided by the storage manager, and used by the transaction processing facility, which is also part of the relational storage manager. Since the locks are the concern only of the relational back end, and not of the object-oriented front end, we omit further discussion of Iris locking.

3.4.6. Argus

Argus provides the conventional shared read and exclusive write locks. One unusual feature of Argus lock management is its lazy lock propagation algorithm. The term "lock propagation" describes the passing of locks between related nested transactions; in Argus, since messages sent when transactions are aborted are not reliably delivered, lock propagation also may come into play into when the transaction holding the lock aborts.

A conventional lock propagation algorithm for nested transactions would pass the locks held by a transaction to its parent transaction immediately upon the transaction’s termination. Locks held by a parent (or grandparent, *etc.*) transaction can then be inherited by child transactions, and passed back up the transaction tree as the children commit or abort. The effect of lock propagation is to keep a lock, once acquired, "in the family", so that no unrelated transaction can see inconsistent data, while allowing a single "family member" transaction at a time access to the data shared between them.

Argus’ lazy lock propagation algorithm works like the conventional algorithm when a transaction commits; its locks are passed to its parent in a reliably delivered commit message, or released if the transaction is a top-level transaction (*topaction*). When a transaction aborts, however, the notification messages it sends are not reliably delivered, so they cannot be used to give the transaction’s locks back to its parent. Instead, the lock will be lazily propagated the next time it is referenced.

When a transaction attempts to acquire a lock, the guardian responsible for the object checks locally whether the object is "locked". If it is not "locked", it is always safe to grant the lock to the requesting transaction. If it is "locked", however, it is necessary to check whether it is actually still locked. This process begins at the least common ancestor (in the transaction tree) of the transaction requesting the lock and the transaction recorded as holding the lock (or at the topaction above the transaction recorded as the holder if the two transactions are unrelated). The guardian where that ancestor transaction was initiated receives a query message. If the ancestor is found to have aborted, the lock can be granted to the requesting transaction immediately. If the ancestor has committed, or if it is a common ancestor and is still running but all subactions between the "holder" and the ancestor have committed, then the lock can also be granted. Otherwise, the requesting transaction must wait until the ancestor commits or aborts.

The actual Argus lock propagation algorithm includes some other minor cases (for reliability and implementation reasons) and represents a refinement of the above algorithm.

3.4.7. Clouds

Clouds provides the standard shared read and exclusive write locks. In keeping with its general model of a persistent virtual memory, locking in Clouds is performed at the page level as part of the virtual memory system. Each object must reside on a separate page (or pages) from any other object. This is reasonable in Clouds, since Clouds objects are very large and relatively few in number, but would be extremely inefficient if the average objects were much smaller than a page of in the presence of a large object population.

Whenever an action first accesses a page for read or for write, it is locked appropriately. Locking in Clouds is approximately two-phase; the only exception is that a read lock can be promoted to a write lock if a page that has previously been read is referenced for write. This exception means that deadlock can occur, since two transactions may hold read locks simultaneously (and subsequently attempt to write the same page). Deadlock is resolved simply by aborting one of the conflicting actions.

3.4.8. Meld

In MELD, there is only one kind of lock: A mutual exclusion lock on an entire object, used to implement a critical section construct called an atomic block. Transaction management is accomplished by other means. Thus, MELD's lock manager is very simple, with only operations to lock an object for exclusive access by a single thread of control and to unlock an object. Concurrency control for transactions in MELD is handled by an optimistic algorithm, which we will discuss further under the heading of MELD's transaction manager.

MELD can get away without using locks when updating objects, because its parallelism is simulated. In the current implementation, each MELD program runs in a single Unix process, with threads being scheduled at the statement level to give the illusion of concurrency. Only one statement is actually executing in the process at any given time. The result is that, when actually updating the object, MELD knows that there can be no conflict with another thread of control. The next version of MELD, MELDC, will no longer schedule at the statement level but will continue to simulate parallelism, running in a single Unix process. Thus, MELDC will, like MELD, not require locking of objects during updates.

3.4.9. Ontos

Ontos uses the standard shared read/exclusive write locks. The object manager calls to get an object, a cluster, and a closure all take a lock argument. They may be used to acquire read, write, or non-restrictive read locks. A non-restrictive read lock does not prohibit concurrent writes.

In the event that a lock cannot be obtained when requested, Ontos calls a user-defined function; this function is specified at the transaction level, rather than at the individual lock request level, but is included here since the calls to the function originate with the lock manager. System-defined functions are available to allow the user to choose any of the common options: abort the current transaction, wait until the lock is available, or simply continue processing while knowing that the lock request has failed. The user may also write handler functions.

3.4.10. Orion

Orion uses standard shared read/exclusive write locks on simple objects; on complex objects, it uses a granularity locking protocol with intention lock modes. This is also, for the most part, a standard protocol [Gray 75]. Three lock modes are added: ISO, IXO, and SIXO. They are described succinctly by two examples from Kim *et al.* [Kim 89]:

1. To access the Vehicle composite object Vi:
   a. Lock the Vehicle class object in IS (intention shared) mode.
   b. Lock the Vehicle composite instance Vi in S (shared) mode.
   c. Lock the class objects for the components in ISO (intention shared object) mode.
2. To update the Vehicle Vi or its components:
   a. Lock the Vehicle class object in IX (intention exclusive) mode.
   b. Lock the Vehicle composite instance Vi in X (exclusive) mode.
   c. Lock the class objects for the components in IXO (intention exclusive object) mode.

In short, the various O modes are used to lock component classes while accessing a complex object. Locking the classes prevents changing the definition of the class or adding new instances while an instance is being accessed. The SIXO mode (shared intention exclusive object) would seem to be intended for locking high-level component classes when updates are to be made only to components in lower levels of the class hierarchy, but the available literature is not clear on this point.

The same paper goes on to describe a design for adding shared component references to Orion’s composite objects, where the existence of a component is not dependent on the existence of its parent. This requires additional lock modes, as well as changes to the versioning scheme and schema evolution mechanism. However, it also states that shared component references have not yet been implemented.

3.4.11. DHOMS

DHOMS uses the standard granularity locking scheme (see above) on its objects. The lock compatibility matrix is stored in a file on disk and is read at server startup. This allows new lock modes to be added, when modifying or replacing the transaction manager, without changing any code in the lock manager.

The lock manager does not actually hold the locks on any object; these are maintained by the object manager. Instead, when a lock is requested by the transaction manager, the DHOMS lock manager queries the object manager for the status of all locks on the requested object. With this information, it then consults its compatibility matrix to decide whether to grant or deny the lock, and finally calls the object manager to enter the new lock into the table if the lock has been granted. If the lock cannot be granted, it is the transaction manager's responsibility to resolve the conflict.

3.4.12. O_2

Lock management in O_2 is handled by the WiSS storage manager. WiSS provides standard shared read/exclusive write locks at the page level. All lock information is kept on the server; the client has no knowledge of what locks exist on each object, and may only make requests to the server. As in Clouds, it is possible to cause a deadlock by having two clients with simultaneous read locks both attempt to update the object, requesting write locks. The available literature says nothing about how deadlocks are handled in O_2.

3.5. Transaction Manager

All of the systems under consideration provide some form of transaction, since they are all at least designed as concurrent object systems (although one system is only implemented as a single-user prototype). All of the concurrent systems provide at least the conventional atomic, serializable transactions. Among OODB's, this is often all that is provided. The other types of systems, often being intended to support design environments of one sort or another, will often provide a form of long-lived, non-serializable design transactions. Table 3-7 summarizes the capabilities of the transaction managers we have studied.

3.5.1. ObServer II

ObServer II provides a single level (with no nesting) of transactions using a (generally) two-phase locking algorithm. ObServer II transactions need not be serializable, but if the transactions adhere to strict two-phase locking using restrictive read and write locks, ObServer I transactions have the standard serializable semantics. If serializability is not necessary, though, a transaction may release some of its locks before it finishes, or some of the less restrictive lock modes may be used. Thus, although most of ObServer II's transactions are the usual atomic and serializable locking type, it is possible to represent optimistic concurrency control (using communication-based locking) and extended transaction models in ObServer II.

ObServer's distinguishing design feature is the *transaction group* [Fernandez 89b; Hornick 87] mechanism for cooperating transactions. A transaction group is a set of transactions (and possibly transaction groups) that share access to a set of objects. Objects may be locked on behalf of a group rather than an individual transaction. When a group holds a lock, all transactions within the group have simultaneous access to the object, uncontrolled with respect to one another. They may all hold copies of the object. Cooperation between transactions is achieved by using the previously mentioned soft lock mechanism, which allows transactions to be notified whenever another transaction within the group updates an object of concern on the server from its copy. By having the group hold a hard lock on the object, and using soft locks to

Transaction Manager Features							
Feature	Mneme	EXODUS	Iris	Argus	Clouds	Avalon	O_2
Algorithm	2PL	2PL	2PL	2PL	2PL Via Threads	Locking	2PL
Distributed?	No	No	No	Yes	Yes	Yes	No
Nested Transactions?	No	No	No	Yes	Yes	Yes	No
Cooperative Transactions?	No	No	No	No	No	No	No
Extensible?	No	To Other Locking Algs.	No	No	No	To Other Locking Algs.	No

Transaction Manager Features						
Feature	ObServer	MELD	Ontos	Orion	Camelot	DHOMS
Algorithm	Locking and/or Notification Based	Optimistic	Locking or Optimistic	2PL	Locking	2PL
Distributed?	No	Yes	No	No	Yes	No
Nested Transactions?	No	No	No	No	Yes	Pseudo Nesting
Cooperative Transactions?	Yes	No	Yes	No	No	In Design
Extensible?	To Other Locking Algs.	In Theory	No	No	To Other Locking Algs.	No

Table 3-7: Transaction Manager Summary

manage concurrency within the group, long-lived engineering design transactions can be implemented. The transaction group mechanism, however, has not yet been implemented.

3.5.2. Camelot

Camelot's transactions use a two-phase locking protocol and two-phase commit; they may be nested. The transaction manager is responsible for maintaining records of all servers participating in each transaction, but it does not actually hold any of the locks or data being used by the transaction. That is the responsibility of a separate data server. Both the data servers and the transaction manager communicate with a separate disk manager, which is responsible for maintaining stable storage both for objects and for log entries. This situation is illustrated in Figure 3-2, which also shows how the major components of Camelot communicate during a simple transaction with one participating data server [Duchamp 90].

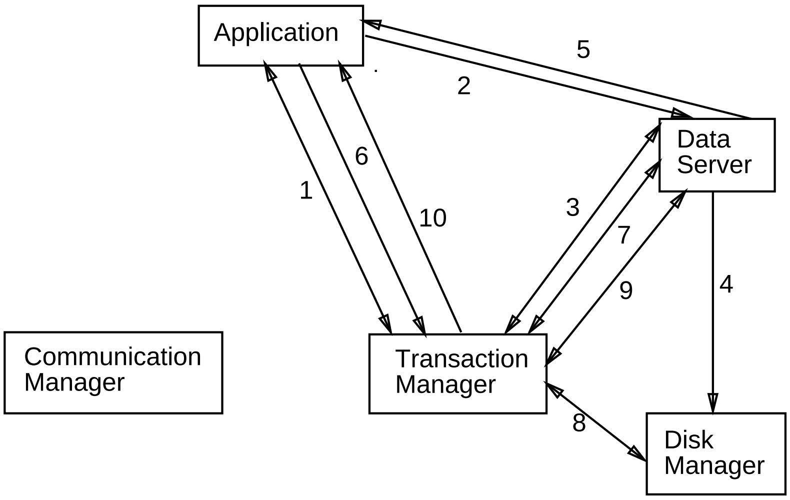

Figure 3-2: Example transaction in Camelot

The diagram shows the top-level components involved in the transaction (boxes), as well as the messages (arrows) passed between them. The communication manager is shown as disconnected from the other components because every one of the message shown passing from one component to another actually passes through the communication manager on the way. Showing the messages passing through the communication manager would unnecessarily muddy the waters. The numbers on the arrows (messages) in the diagram correspond to the following sequence of operations performed by the Camelot system:

1. The application begins the transaction by sending a message to the transaction manager. The transaction manager returns the identifier to the application.

2. The application sends a message to the data server to call an operation on the data residing at that server.

3. The server notifies the transaction manager that it will be participating in the transaction. The transaction manager acknowledges this.

4. The server acquires appropriate locks on the data using the Camelot library primitives for locking. It then performs the operation(s) on the data, and sends both the old and the new values of the object to the disk manager. Camelot is optimized so that it is often not actually necessary to force this record to disk, but it still must always be sent. This "message" is usually sent using shared memory facilities, rather than inter-process communication facilities.

5. The operation on the object done, the server acknowledges its completion to the application.

6. The application sends a request to commit to the transaction manager.

7. First phase of the commit protocol: The transaction manager asks the server if it is ready to commit, and the server replies that it is.

8. The transaction manager calls the disk manager to write a commit record to stable storage; the disk manager replies after it has done this.

9. The transaction manager tells the application that the transaction has committed.

10. Finally, the transaction manager tells the server to release the locks held by the completed transaction.

The foregoing assumes a top-level, rather than a nested, transaction. In the case of a nested transaction, the steps are the same, but some of the internal actions being performed differ (*e.g.*, the data server does not necessarily release the locks immediately).

3.5.3. Mneme

Mneme provides a single level of transactions using a two-phase locking algorithm. As previously stated, the locks are acquired transparently as slots and bytes within various Mneme objects are accessed by the transaction. Another object management strategy routine is called by the transaction manager to release the locks held by the transaction after it commits.

Since Mneme has a single object base or object server as its back end, no commit protocol is necessary. Mneme does not support distributed transactions. The prototype, Basic Mneme, is even simpler, since it does not even support concurrency. Its transactions are entirely an atomicity mechanism to allow aborting changes to the object base, and do not make use of any logging protocols for recoverability. Instead, shadow versions of changed objects are kept while a transaction is in progress, and at commit time these are written to the persistent object base as the current version.

3.5.4. Exodus

EXODUS, as an object-oriented database, supports the common database notion of atomic, serializable transactions using two-phase locking. It does not support nested transactions, or design transactions, or distributed transactions. Recoverability of small objects is accomplished by using before/after image logging (logging the values of all data in the object, or "images" of the object, before and after the change). Recoverability of large objects is supported by shadowing the changed parts of the object (using the version mechanism, as described in Section 3.3.4) until commit time. At commit, the object header of the object is overwritten by the new version, and before/after image logging is used. The old version of the object is deleted.

3.5.5. Iris

Since the Iris transaction manager is provided by its relational back end, it is similar to that of System R's RSS [Blasgen 77]. It provides atomic, serializable transactions using two-phase locking. Transactions may not be nested, and distributed transactions are not supported.

3.5.6. Argus

Argus supports nested transactions, which may cross guardian boundaries. Top-level transactions, or *topactions*, are the only transactions that are actually committed to stable storage; lower-level transactions, or *actions*, commit only to shadow versions of storage. This is done for performance reasons, so that committing actions is very fast. It has the disadvantage that, if a failure should occur inside of a topaction, even after the completion of large numbers of actions within it, the topaction will be rolled back. Argus uses conventional two-phase locking and two-phase commit protocols, with lock propagation between nested transactions. The lock propagation algorithm is lazy, in that no messages are sent on commit and delivery of messages sent on abort is not guaranteed by the system, so that if an object is found to be locked when a transaction attempts to access it, it becomes necessary to send queries to other concurrent actions (actually, to their guardians) to determine the actual status of the lock. This algorithm was described in more detail in Section 3.4.6.

We consider here an example transaction based on the description of a banking system in Argus presented in [Liskov 88]. The banking system consists of two kinds of guardians. *Branch* guardians implement operations such as *open*, *close*, *deposit* and *withdraw* on accounts. Each branch guardian maintains all of the accounts at a single bank branch. *Front end* guardians are input terminals representing tellers or ATMs. In addition, the banking system makes use of the standard *registry* guardian provided by the Argus system to allow front end guardians to locate branch guardians by giving their branch codes (names). The registry guardian is the Argus name service.

Branch guardians are implemented using a hash table kept in stable storage to store the account data. The hash table contains pairs, consisting of the account number (the hash key) and the account balance. Figure 3-3 shows the guardians and internal account data structures involved in a transfer between two accounts at different branches. Guardians are shown as solid boxes, containing dashed boxes which represent CLU objects maintained in the guardian's stable storage.

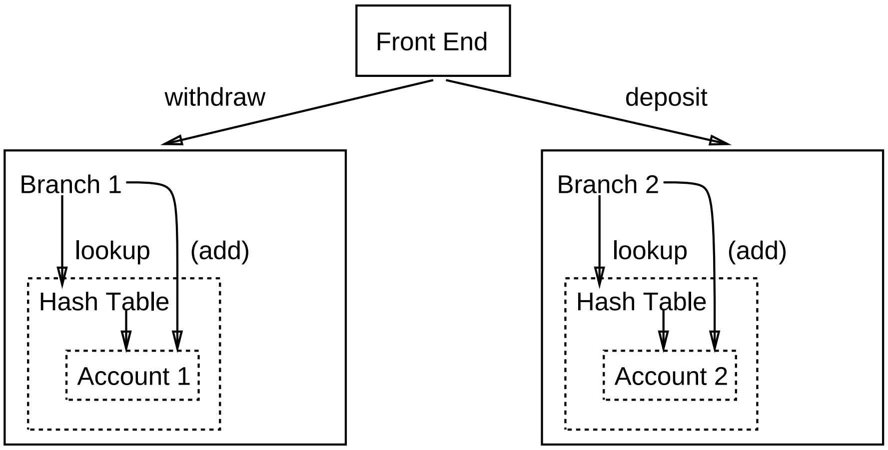

Figure 3-3: Example transaction in Argus

To perform the transfer, a user makes a request at a front end guardian. The first thing the guardian does is to begin a topaction. The guardian must then look up the branch codes involved in the transaction to locate the proper branch guardians. Conceptually, this involves sending messages to the registry guardian. In the example system, however, this process has been optimized by obtaining a complete copy of the registry guardian's branch information at startup (and again each time the front end guardian restarts after a crash). This information is then used whenever it is necessary to look up a branch code. This is why no registry guardian is shown in Figure 3-3. Of course, this only works because the set of branches is constant.

Next, a message is sent to *Branch 1* to withdraw the money from *Account 1*. When this message arrives, a subaction begins at *Branch 1*. The subaction calls a *lookup* procedure, which is not a handler, although it is defined at the same level as the handlers in the guardian. Since *lookup* is a procedure, it does not begin its own subaction, but continues as part of the existing subaction. It computes the hash function and reads the appropriate hash bucket, acquiring a read lock on the bucket. This bucket is an array containing the pairs mentioned earlier. It then steps through the array to find the proper account, and returns the location of the balance information.

After *lookup* returns, *withdraw* checks that the account's balance is sufficient, then subtracts the amount being transferred from the balance, acquiring a write lock on the hash bucket containing the account in the process. Had the balance not been sufficient, *withdraw* would instead have raised an exception, which would have resulted in the topaction being aborted. The *withdraw* subaction can then commit, and the *withdraw* handler returns.

With the withdrawal complete, the deposit proceeds similarly, with a *deposit* message being sent to *Branch 2*, which reads its hash table and writes its new balance in the same way that *Branch 1* did (except, of course, that *deposit* is *adding* to the balance).

After *deposit* returns, the transfer is complete and the topaction can commit.

3.5.7. Clouds

Clouds supports transactions by means of special types of threads, known as *consistency-preserving threads*, or *cp-threads*. There are two types of cp-threads: *global* cp-threads (*gcp-threads*), and *local* cp-threads (*lcp-threads*). Global cp-threads, as the name implies, implement distributed transactions and enforce global atomicity and consistency. They use two-phase locking and two-phase commit. Local cp-threads enforce only local atomicity and consistency, and commit changes to their single object when the lcp-thread exits a method.

Cp-threads are created by entering a method marked as global or local consistency preserving; this changes a normal simple thread (s-thread) to a gcp-thread or an lcp-thread, respectively. Most methods in Clouds are marked as making no change to the consistency status of the entering thread. Methods that do affect the consistency level enforce exactly the marked level of consistency, even if the entering thread previously maintained a higher consistency level. For example, if a gcp-thread enters an operation marked as local consistency-preserving, it becomes an lcp-thread, and its changes within that method will be committed at the completion of the method. Thus, global consistency is not preserved if the gcp-thread later aborts, since some of its effects have already been made public by the commit at the end of the local consistency-preserving method. Gcp-threads may call other global consistency preserving operations, but that does not give the effect of nested transactions; the result is one large transaction, committed at the end of the containing operation, since the gcp-thread is not transformed the second time it enters a global consistency preserving operation.

Figure 3-4 shows the steps involved in a simple example transaction involving two objects, one local and one remote. Only the steps for the local object are shown; the remote object acts similarly at its own site. The numbers on the arrows correspond to those in the text description below.

1. The transaction begins when an ordinary thread (*s-thread*) in the application calls an operation that is marked as a transaction (*global consistency preserving*). This

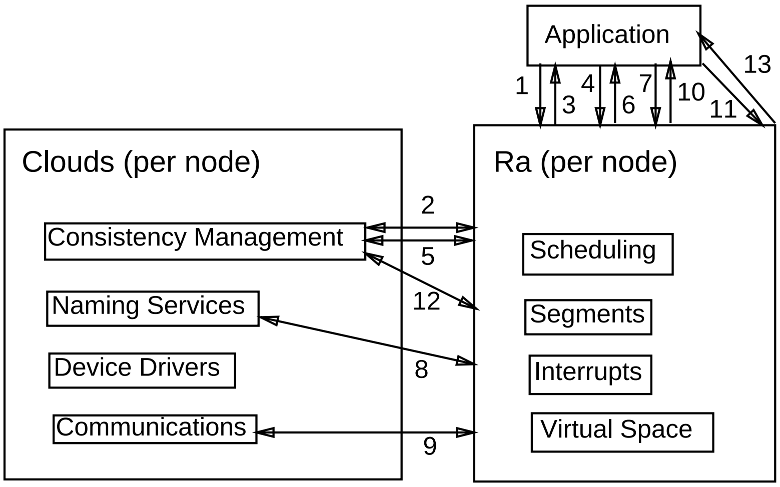

Figure 3-4: Clouds example transaction

causes a trap to the Ra kernel.

2. The kernel calls out to Clouds to change the *s-thread* into the desired *gcp-thread* type. Clouds makes the change and returns to Ra.
3. Ra returns from the trap; the thread is now treated as a transaction.
4. When the *gcp-thread* changes the data in an object, this causes another trap to Ra.
5. Ra calls a Clouds consistency management operation to create a shadow version of the object for the *gcp-thread*. With the shadow version created and associated with the thread, Clouds returns to Ra.
6. Ra returns to the application, which continues running and making other changes to its shadow version.
7. The call to an object not in memory also causes a trap to Ra.
8. Ra calls the Clouds name service to look up the name of the remote object. The name service returns the location of the object.
9. Ra sends the message to the remote object on behalf of the application, using the Clouds communication service. Eventually, when the remote message finishes execution, it receives the reply from the remote object.
10. Ra forwards the message to the application, which again continues running.
11. Later, the *gcp* operation exits. This traps one more time to Ra.
12. Ra calls a commit operation for the *gcp-thread*. This updates the object, frees the shadow version, changes the *gcp-thread* back to an *s-thread* and returns.
13. Finally, Ra returns to the application, which continues running in the non-transaction (*s-thread*) mode.

3.5.8. Avalon

Transaction processing in Avalon/C++ is provided by the ATOMIC class. This adds to the RESILIENT class (described in Section 3.2.6) three private and two public operations. The private operations are `seize()`, `release()`, and `pause()`. The `seize` and `release` operations lock and unlock the object for exclusive access during critical sections in the commit process; they are provided by Avalon. The two public operations are `commit(tid)` and `abort(tid)`. These are hooks that the Avalon programmer must use to provide the means to commit or abort the object when a transaction finishes. They are automatically called for all objects participating in a transaction whenever the transaction commits or aborts, respectively.

The ability to write explicit `seize()`, `release()`, and `pause()` operations is only necessary if fine control over concurrency is desired within the transaction. For most Avalon applications, the `DYNAMIC` class is sufficient, and is much more convenient. The `DYNAMIC` class limits the programmer to strict two-phase locking during transactions, but provides `read_lock()` and `write_lock()` operations rather than `seize`, `release`, and `pause`. These operations acquire locks which are held until the transaction finishes, at which time they are released automatically by the runtime system.

In writing a class to be used by transactional Avalon code, the programmer need not specify the `commit` and `abort` operations; default commit, abort, and recovery code is provided by the `DYNAMIC` class. To use the `ATOMIC` class, however, the programmer must provide these operations, as well as those that implement the functionality of the class. When the `DYNAMIC` class is used, the programmer simply writes all of the operations on the class as conventional C++ methods that call `read_lock` and `write_lock` as appropriate when accessing the data stored in the object. Recovery support is provided automatically by Avalon. Note, however, that it is necessary to call `read_lock` and `write_lock` before accessing the data in the object; otherwise, Avalon's concurrency control strategy will break down.

The example class in Figure 3-5 is taken from the Avalon manual [Eppinger 90][2]. It serves to illustrate that writing atomic classes in Avalon can be very simple. The same class, however, takes a full page of code to specify if written using the `SUBATOMIC` class. The class atomic_int has only two public functions. One redefines the assignment operator, replacing it with one that acquires a read lock before actually performing the assignment. The other redefines the conversion of an atomic_int to an int, which is performed automatically whenever an atomic_int appears in an expression. This allows an atomic_int to be used exactly like an int in an Avalon program.

A shortcoming of Avalon is that only variables of atomic types are handled properly by the transaction mechanism. Although it is easy to write atomic types, it is also easy to make a mistake. If a transaction accesses both `int` and `atomic_int` variables, only the changes

---

[2]This manual renames the classes used in the earlier literature, and in this paper. The `RESILIENT` class remains the same, but the `ATOMIC` class is renamed `SUBATOMIC` and the `DYNAMIC` class is renamed `ATOMIC`. Although these names are more intuitive than those previously used, this manual is (to our knowledge) not yet published, and so we have used the older names.

made to `atomic_int` variables will be recoverable. In fact, the changes made to `int` variables will be visible to other transactions while the transaction is still in progress! Non-atomic variables have completely uncontrolled concurrency, even within a transaction.

```
#include <avalon.h>

class atomic_int: public atomic {
  int val;
 public:
  int operator=(int rhs);
  operator int();
};

int atomic_int::operator=(int rhs) {
  write_lock();
  pinning () return val = rhs;
}

atomic_int::operator int() {
  read_lock();
  return val;
}
```

Figure 3-5: Example Avalon class: `atomic_int`

3.5.9. Meld

MELD's transactions use an optimistic, rather than a locking, concurrency control scheme. The transaction manager does not communicate directly with lock management, although atomic blocks are useful within transactions for managing the multi-threaded concurrency allowed by the language. MELD provides a single level of distributed transactions (but not nested transactions) using a two-phase commit protocol.

MELD's transaction manager is written in terms of internal second-class ‘‘objects’’. These communicate similarly to MELD's first-class objects, but are not actually instances of a MELD class, having their methods directly implemented in C instead. These methods are of two kinds: *local* operations, implementing the optimistic (or, we have proposed, pessimistic [Popovich 91]) concurrency control algorithm, and *global* operations, implementing the distributed commit protocol. The local methods of these transaction ‘‘objects’’ are called automatically by the runtime system as side-effects of accessing MELD objects, and maintain the read and write sets of the transaction. The global methods are called through the transaction manager. Two of them implement a simple termination detection protocol, with one message being sent to a coordinator object (located in the process where the transaction started) at the beginning and one at the end of each method. This is a rather expensive way to implement termination detection, but implementation considerations forced this small granularity. The rest of the global methods are called at commit time, and implement a standard two-phase commit protocol.

This simple example transaction, taken from a stock trading application used as a MELD demonstration, shows how MELD uses its transaction ‘‘objects’’. The application uses several distinct types of MELD objects: stocks, stock portfolios, checking accounts and users. To buy a stock, a user sends a `buy` message to the user’s stock portfolio. This method, implemented as a transaction block, sends a `quote` message to the stock to get its current price and a `withdraw` message to the user’s checking account to get the money to buy the stock. It then updates the user’s portfolio to reflect the stock that was purchased.

This activity thus involves three objects, which must be kept consistent, and up to three cohorts since the objects may reside in different processes (which may execute on different hosts), as depicted in Figure 3-6.

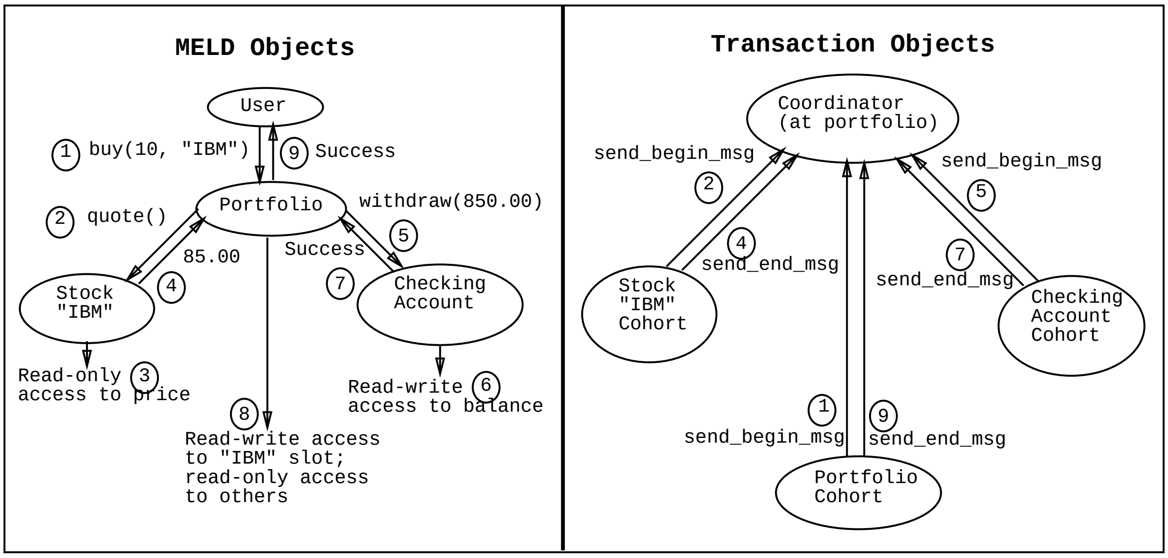

Figure 3-6: MELD Transaction Example: `buy(10, "IBM")`

We assume that the user already owns some shares of IBM and wants to buy 10 more. To do this, the user types a message on the terminal that causes the user object to send the message `buy(10, "IBM")`] to the corresponding stock portfolio. The events from this point onward are depicted in Figure 3-6; the numbers below correspond to those on the arrows in the diagram.

1. The `buy` message arrives at the portfolio object and the corresponding method is invoked. Entering the transaction block has the side effects of creating a coordinator object and a cohort object in the current process, and then the cohort sends a message to the coordinator (called a **send_begin_msg**) indicating that a method has begun execution as part of the transaction. When this message is received by the coordinator, it records the information in the message as part of MELD’s termination detection algorithm.

2. The portfolio then gets a handle on the persistent object representing the stock named `"IBM"`, which involves an exchange of messages with the name server (not shown in the diagram), and sends a `quote` message to the stock. This message implicitly carries the transaction identifier, which is the message passing address of the coordinator object. When the `quote` method is invoked, this has the side effects of creating a local cohort for the transaction in the stock object’s process and sending a **send_begin_msg** back to the coordinator to indicate that the method

is running as part of the transaction. Again, when this is received by the coordinator, the information it carries is recorded for use in termination detection.

3. When the `quote` method (not shown) reads the `price` instance variable of the stock object, the **do_read** operation is invoked to add this variable to the cohort's **read_set** and then the **pre_read** message accesses the appropriate version of this instance variable (the version written by the last visible transaction before the start time of the cohort).

4. The `quote` method sends a message returning the stock's price per share (for example, $85.00) back to its caller, the stock portfolio. As a side effect, a message to the coordinator is also generated. This message (called a **send_end_msg**) lets the coordinator know that the thread in the stock object has completed, and it destroys its record of that method. When the return message arrives, the `buy` method in the portfolio object continues.

5. The next step is to withdraw the money to buy the stock, so `buy` sends a `withdraw` message to the checking account object. When the `withdraw` method is invoked, it has the side effects of creating a cohort object in the checking account's process, since the transaction has not previously involved that process, and sends a **send_begin_msg** to the coordinator object. The `withdraw` method checks that the balance is sufficient; as a side effect of doing this, the `balance` instance variable is added to the cohort's **read_set**.

6. `withdraw` then subtracts the amount specified from the balance and writes the new balance back; `balance` is added to the **write_set** as a side effect, and a new shadow version is created for this instance variable.

7. The `withdraw` method concludes by sending a message back to the portfolio indicating that the withdrawal succeeded, and the `buy` method continues again. This return message has the usual side effect of a **send_end_msg** message to the coordinator, to tell it to destroy its record of this method.

8. The `buy` method next finds the slot for `"IBM"` in its portfolio and increments the number of shares by 10. This involves adding the number of shares to the read set and the write set and writing its shadow version. Other variables may also be read during the search.

9. Finally, when `buy` is finished, it sends a return message to the user object to say that it succeeded; this results in a **send_end_msg** to the coordinator. After processing this message, the coordinator realizes it has no more active threads, and so begins the commit protocol.

MELD's two-phase commit protocol actually has three conceptual phases; it avoids the necessity for three message exchanges at commit time by piggybacking the first phase on the earlier termination detection messages. The protocol is implemented by coordinator and cohort objects as follows:

1. The coordinator determines the global start number (the time the transaction as a whole is considered to have begun) and global transaction number (the time at which the transaction would appear in a serial execution of the system) by taking the minimum of the local start numbers and the maximum of the local transaction numbers, plus a small random safety factor. These numbers have already been collected, as they were available at the time the earlier termination detection

messages were sent, and they were piggybacked on these messages.

The coordinator sends this pair of numbers to all cohort objects in **validate_request** messages.

2. The cohorts reply with **validate_return** messages, which tell the coordinator whether or not the transaction could be validated at the cohort with these global transaction numbers.

3. If the transaction was valid at all cohorts, the coordinator then sends the cohorts a **real_write** message, and they update their current versions of all objects involved in the transaction from their shadow versions of those objects.

   Otherwise, the coordinator sends the cohorts a **destroy_version** message, and they discard their shadow versions for the transaction.

To support recoverability, acknowledgments would be required from every cohort to complete the second phase, but this is not implemented in the current version of MELD. The commit protocol messages for our example are shown in Figure 3-7.

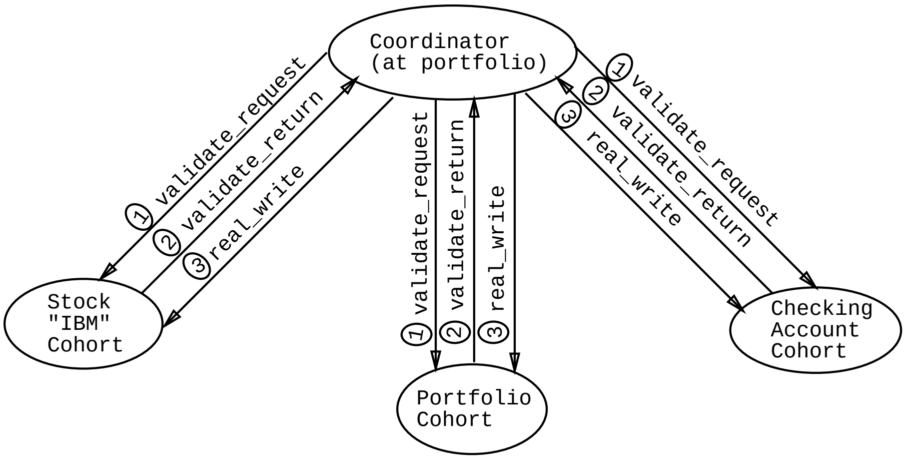

Figure 3-7: MELD Transaction Example Commit Protocol

3.5.10. Ontos

The Ontos transaction manager provides atomic, serializable transactions using two-phase locking. It also provides a simple facility for shared (cooperating) transactions. At transaction start, the programmer may specify a policy function to be called when a lock request fails, a buffering policy for objects brought into memory by the transaction, and a name for the transaction. The name is used by the shared transaction facility. If a process calls `TransactionStart` with a name, and the name is already in use by another transaction, then the calling process joins the existing transaction, rather than creating its own transaction. There is no access control; any process that knows a transaction's name can join it. If a transaction is created without specifying a name, however, no other process can ever join it. Data and locks are shared between all participating processes in the transaction, and at commit time, a two-phase commit protocol maintains atomicity from the viewpoint of a non-participant in the transaction.

Ontos also provides a facility for dealing with the potential inconsistencies that are created by transactions that store data outside of the object system (*e.g.*, in their processes' heap space). Both the commit and abort functions take a `cleanup` argument, which is a function that frees various resources that had been allocated by the transaction. If no cleanup function is used, a transaction's heap data will not be freed when the transaction finishes, and so an aborted transaction's intermediate results may be visible (on the heap) to subsequent transactions.

There are three system-defined cleanup functions. One of them may be used, or the user may provide a cleanup function. The default cleanup function for committing a transaction is `DeferCleanup`, and the default for aborting a transaction is `CleanCache`. The system-defined functions are:

- `CleanCache` - Invalidates implicit object references and frees the heap space that was used by the transaction. This is most appropriate after an abort, to make sure that none of the heap data are inadvertently treated as valid.
- `DeferCleanup` - Leaves the objects involved in the transaction valid and in memory, so that they may continue to be used. If another transaction updates the object before the next reference, however, the in-memory copy will be invalidated and the object will be re-faulted in from disk.
- `AbortNextIfDirty` - Provides an optimistic protocol for updating the objects on disk. Checking for conflicts is postponed until another transaction tries to update the object to disk. That transaction will then abort if a conflict is detected.

3.5.11. Orion

Orion's transaction management provides a single level of atomic, serializable transactions using the locking protocols previously described and a conventional logging algorithm (UNDO logging) for crash recovery. Orion's complex objects cause no problems (other than those already solved by its locking protocol) for transaction management, and components are simply treated as separate objects by the transaction manager.

3.5.12. DHOMS

DHOMS transaction management is a conventional two-phase locking scheme. It provides atomic, serializable transactions with semantics similar to a nested transaction model. The current transaction manager only provides concurrency control, but a standard logging approach is planned to provide recoverability.

The DHOMS transaction manager (TM), in addition to providing transaction support, is also responsible for resolving locking conflicts when they are detected by the lock manager. This was done to keep all knowledge of transaction semantics in the TM rather than the lock manager, so as to facilitate replacement of the TM by another transaction module by minimizing the amount of changes that must be made outside of the TM.

To replace the DHOMS TM, all that is necessary (outside of writing the new TM code) is to install a new lock mode description file for the Lock Manager. Since there are no semantic dependencies on the TM in either the OM or the LM, no code changes are necessary there.

The current implementation of the transaction manager does not resolve locking conflicts;

instead, when a conflict is detected, it aborts the transaction that discovered the conflict. The transaction manager design, however, provides for *control rules*, which specify the actions to be taken to resolve conflicts. For example, a control rule may state that conflicts are to be ignored if the conflicting transactions are owned by the same user, or by two members of a user group. A control rule might also specify that conflicts are to be ignored between operations that satisfy some property, *e.g.*, commutativity. This is not, however, inferred by the system; these properties must be specified by the programmer, by making the operations members of the same *troupe*.

The TM, as implemented, takes an unusual approach to the idea of aborting a transaction, due to the rule-based nature of Marvel and the fact that the actions of many Marvel rules (*e.g.*, invoking software tools) are irreversible. The processing of each rule is atomic with respect to the processing of any other rule. When a conflict is detected and a transaction must be aborted, processing of the current rule chain stops; no further rules that would ordinarily have been fired by the current rule will be processed. However, the previously processed rules in the chain do not have their effects undone; it is as though each rule were a nested transaction, and the chain as a whole the top-level transaction, which committed after the rule chain terminated.

3.5.13. O_2

O_2's transaction support is provided by WiSS. WiSS uses two-phase hierarchical (intention) locking on files and pages within files, and a non-two-phase locking protocol to ensure the consistency of indices. The literature does not indicate the exact non-two-phase algorithm being used. Since all objects live on a central server, O_2 needs no distributed transaction support. WiSS transactions may not be nested.

4. Conclusions

Much research has been done in the last few years in the closely related areas of object-oriented programming languages and databases. Both areas now seem to be working toward a common end, that of an *object management system*. We see OMS's as similar to OODB's, but providing a general purpose concurrent object oriented programming language as well, which shares the object base with the OODB query facilities. In this paper, we have defined the different types of object systems (object servers, persistent OOPL's, OODB's and OMS's) in terms of their interfaces and capabilities. We have presented a general OMS architecture and examined the distinguishing features and general architecture of systems of each type of system with respect to this model. We now review some of the contributions of the systems that we have studied. Following our architectural model, we will discuss each component, rather than each system, in the same order we have used throughout this paper.

Any OMS or OODB needs a type manager to maintain the persistent schema of the database; systems without type managers necessarily lack query facilities, since this information is needed for query processing. The OODB's and OMS's we studied represent their database schemata as objects within their own databases; future OMS's should do the same. Almost all of these systems also agree on multiple inheritance as a model for sharing between classes. EXODUS's facility for expanding the set of primitive types in the database offers offers the opportunity for the database implementor to maximize the efficiency of his system by customizing the system for his application. Iris and Orion both provide support for schema evolution, but using opposite approaches. Iris considers types as static and objects as able to change their type, while Orion considers an object's type attribute as permanent but allows type definitions to change. Neither one of these approaches has a clear advantage over the other.

The storage manager is, of course, essential to any persistent object system, but not all systems separate it from the object manager. We believe, as do the implementors of DHOMS, that there should be a clear distinction between storage management and object management, and that the storage manager should not know about object structure. The other system to clearly separate object management from storage management, Iris, does not necessarily share this view; its object manager is not layered on top of its storage manager. All of the systems except Orion and Clouds maintain their storage in a file or files on disk; this is clearly the simplest way to go, although Orion's use of a raw disk partition has equally clear advantages for applications that require maximum performance. Clouds is unique in treating the object base as virtual memory; while this is natural for an object-oriented operating system, it seems awkward for any other object-oriented application. The storage manager is responsible for recoverability support, and most of the systems provide this by using standard database logging techniques. This seems likely to continue in future systems.

The object manager is obviously central to an object management system. Objects in most systems consist of a set of attributes, some of which may be other objects. The OODB's are leading the way in supporting complex objects, with the idea of objects containing other objects rather than simply being linked to them by an attribute. All of the OODB's except EXODUS support complex objects to some degree. EXODUS supports very large single-level objects. Aside from being intuitively useful, containment provides a possible strategy for clustering objects on disk storage to take advantage of locality of reference. None of the systems have used this approach, although O_2's placement trees essentially select a subset of this relation for use in clustering. Ontos allows an arbitrary collection of objects to be clustered, and Orion clusters objects according to their class. Mneme provides a primitive level of support for clustering in its pool strategies, but does no clustering by default. The other systems provide no clustering support.

OOPL's, on the other hand, provide active objects; OODB's generally do not. An OMS should ideally provide both complex and active objects. Argus, Clouds, MELD and Avalon all provide objects with multi-threaded method execution. Argus and Clouds additionally allow objects to contain processes. MELD does allow code to run indefinitely within an object, but does not use a process abstraction. EXODUS provides some interesting features for advanced applications: multiple versions and configurability. Its multiple versions allow a history to be kept of changes to an object, which is important in engineering environments. Configurability, which Ontos also provides, allows the programmer to extend the functionality of the object manager. This is done in a conventional programming language in EXODUS, and in the database programming language in Ontos.

In most of the systems, the lock manager provides the primitive support for the concurrency control features of the transaction manager. The exception is MELD, where locks for mutual exclusion are peripheral to the optimistic transactions. All of the systems except the single-user Mneme provide locks of some sort, and even Mneme provides hooks in its strategies to allow the programmer to add locks. All of the other systems except MELD provide shared read/exclusive write locks, with MELD omitting them because they are not needed in its transaction algorithm. In most of the systems, the object is the only granularity of locking. Mneme and EXODUS allow locking only a portion of an object. The only departures are Iris, because of its relational back end, and Clouds, because of its virtual memory design and operating system nature. Locks are generally located with the objects on the object server; in the systems that have no central server (Argus, Clouds and MELD) they are collocated with the objects. The exception is Camelot, which maintains them on a separate server. Finally, two of the systems (ObServer and Ontos) provide "soft" or notification-based locks. Both build on this support in their transaction managers to provide cooperative transactions. These are very useful in advanced engineering applications, so "soft" locks should be provided by the lock managers of future OMS's.

We finish by considering the transaction manager. Transactions in most of these systems are based on the standard two-phase locking algorithm. ObServer, Avalon, Ontos and Camelot also allow non-two-phase locking for applications where serializability is not of the highest importance. Ontos additionally provides optimistic transactions using "soft locks", and ObServer has the capability to provide optimistic or other notification-based models, but (we believe) does not provide these as standard. The OODB's, with their central database servers, support only single-site transactions, with the client containing only a copy of the server's master data. The OOPL's and Camelot, with their inherently distributed nature, all provide distributed transactions. All of these systems except MELD also have nested transactions. It seems clear, then, that future OMS's should provide nested distributed transactions using at least a two-phase locking protocol, and that those intended for engineering applications should provide flexibility in transaction processing algorithm by using "soft" as well as hard locks. Finally, a few of the systems allow extension by the programmer of their transaction processing algorithms. EXODUS supports this, although in its underlying programming language rather than in the database programming language. Camelot, Avalon, and ObServer allow the use of non-two-phase locking and/or notification instead of locking, which constitutes a sort of extension to the transaction algorithm. Avalon allows customized transaction processing to be written into the objects that the transactions run on, while MELD takes the opposite approach. Finally, the DHOMS design for rule-based lock conflict resolution, along with its modular structure, should allow its transaction algorithm to be extended or even completely replaced relatively easily. DHOMS is the only system that we studied that was designed to facilitate total replacement, as opposed to extension, of the transaction manager.

References

[Andrews 87] Timothy Andrews and Craig Harris.
Combining Language and Database Advances in an Object-Oriented Development Environment.
In Norman Meyrowitz (editor), *Object-Oriented Programming Systems, Languages, and Applications Conference*, pages 430-440. ACM Press, October, 1987.
Special issue of SIGPLAN Notices, 22(12), December 1987.

[Atkinson 90] Malcolm Atkinson, Francois Bancilhon, David DeWitt, Klaus Dittrich, David Maier and Stanley Zdonik.
The Object-Oriented Database System Manifesto.
Deductive and Object-Oriented Databases.
Elsevier Science, 1990.

[Bancilhon 88] Francois Bancilhon.
Object-Oriented Database Systems.
In *7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 152-162. 1988.

[Barghouti 90] Naser S. Barghouti and Gail E. Kaiser.
Modeling Concurrency in Rule-Based Development Environments.
IEEE Expert 5(6):15-27, December, 1990.

[Bayer 77] R. Bayer and M. Schkolnick.
Concurrency of Operations on B-trees.
Acta Informatica (9), 1977.

[Ben-Shaul 91] Israel Ben-Shaul.
An Object Management System for Multi-User Programming Environments.
Technical Report CUCS-010-91, Columbia University, March, 1991.

[Benzaken 90] V. Benzaken and C. Delobel.
Enhancing Performance in a Persistent Object Store: Clustering Strategies in O_2.
In *4th International Workshop on Persistent Object Systems*, pages 375-384. Martha's Vineyard, MA, September, 1990.

[Black 86] Andrew Black, Norman Hutchinson, Eril Jul and Henry Levy.
Object Structure in the Emerald System.
In Norman Meyrowitz (editor), *Object-Oriented Programming Systems, Languages and Applications Conference*, pages 78-86. ACM, Portland OR, September, 1986.
Special issue of *SIGPLAN Notices*, 21(11), November 1986.

[Blasgen 77] M. W. Blasgen and K. P. Eswaran.
Storage and access in relation databases.
IBM Systems Journal 16(4):363-377, 1977.

[Bloch 89] Joshua J. Bloch.
The Camelot Library: A C Language Extension for Programming a General Purpose Distributed Transaction System.
In *9th International Conference on Distributed Computing Systems*, pages 172-180. IEEE Computer Society Press, Newport Beach CA, June, 1989.

[Carey 86] Michael J. Carey, David J. DeWitt, Daniel Frank, Goetz Graefe, M. Muralikrishna, Joel E. Richardson and Eugene J. Shekita.
The Architecture of the EXODUS Extensible DBMS.
In *1986 International Workshop on Object-Oriented Database Systems*, pages 52-65. Pacific Grove, CA, September, 1986.

[Chou 85] H.-T. Chou, David J. DeWitt, Randy H. Katz and Anthony C. King.
Design and Implementation of the Wisconsin Storage System.
Software — Practice and Experience 15(10), October, 1985.

[Christou 88] Vassiliki Christou and J. Eliot B. Moss.
Persistent Owl: Heap Management and Integration with Mneme.
Technical Report 88-78, University of Massachusetts, August, 1988.

[Clamen 90] Stewart M. Clamen, Linda D. Leibengood, Scott M. Nettles and Jeannette M. Wing.
Reliable Distributed Computing with Avalon/Common Lisp.
In *International Conference on Computer Languages*, pages 169-179. New Orleans LA, March, 1990.

[Dahl 66] Ole-Johan Dahl and Kristen Nygaard.
SIMULA— an ALGOL-Based Simulation Language.
Communications of the ACM 9(9):671-678, September, 1966.

[Dasgupta 88] Partha Dasgupta, Richard J. Leblanc Jr. and William F. Appelbe.
The Clouds Distributed Operating System: Functional Description, Implementation Details and Related Work.
In *8th International Conference on Distributed Computing Systems*, pages 2-9. San Jose CA, June, 1988.

[Detlefs 88] David Detlefs, Maurice Herlihy and Jeannette Wing.
Inheritance of Synchronization and Recovery Properties in Avalon/C++.
Computer 21(12):57-69, December, 1988.

[Duchamp 90] Dan Duchamp, Joshua J. Bloch, Jeffrey L. Eppinger, Alfred Z. Spector and Dean Thompson.
Design Rationale of the Camelot Transaction Processing Facility.
Technical Report CUCS-008-90, Columbia University, 1990.
Submitted for publication.

[Eppinger 90] Jeffrey L. Eppinger, Lily B. Mummert and Alfred Z. Spector.
Guide to the Camelot Distributed Transaction Facility including the Avalon Language.
July, 1990.

[Eswaran 76] K. P. Eswaran, J. N. Gray, R. A. Lorie and I. L. Traiger.
The Notions of Consistency and Predicate Locks in a Database System.
Communications of the ACM 19(11):624-632, November, 1976.

[Fernandez 89a] Mary F. Fernandez, Stanley B. Zdonik and Alan N. Ewald.
ObServer: A Storage System for Object-Oriented Applications.
September, 1989.

[Fernandez 89b] Mary F. Fernandez and Stanley B. Zdonik.
Transaction Groups: A Model for Controlling Cooperative Work.
In *3rd International Workshop on Persistent Object Systems: Their Design, Implementation and Use*, pages 128-138. Queensland, Australia, January, 1989.

[Fishman 89] D. H. Fishman, J. Annevelink, E. Chow, T. Connors, J. W. Davis, W. Hasan, C. G. Hoch, W. Kent, S. Leichner, P. Lyngbaek, B. Mahbod, M. A. Neimat, T. Risch, M. C. Shan and W. K. Wilkinson.
Overview of the Iris DBMS.
Object-Oriented Concepts, Databases, and Applications.
ACM Press, New York, 1989, pages 219-250, Chapter 10.

[Goldberg 83] Adele Goldberg and David Robson.
Smalltalk-80 The Language and its Implementation.
Addison-Wesley, Reading MA, 1983.

[Gray 75] J. Gray, R. Lorie and G. Putzolu.
Granularity of Locks and Degrees of Consistency in a Shared Database.
In *International Conference on Very Large Data Bases*, pages 428-451. Morgan Kaufmann, 1975.

[Gray 78] J. Gray.
Notes on Database Operating Systems.
Lecture Notes in Computer Science. Volume 60: Operating Systems: an Advanced Course.
Springer-Verlag, Berlin, 1978.

[Hornick 87] Mark F. Hornick and Stanley B. Zdonik.
A Shared, Segmented Memory System for an Object-Oriented Database.
ACM Transactions on Office Automation Systems 5(1):70-95, January, 1987.

[Kaiser 89] Gail E. Kaiser, Steven S. Popovich, Wenwey Hseush and Shyhtsun Felix Wu.
MELDing Multiple Granularities of Parallelism.
In Stephen Cook (editor), *3rd European Conference on Object-Oriented Programming*, pages 147-166. Cambridge University Press, Nottingham, UK, July, 1989.

[Kim 89] Won Kim, Elisa Bertino and Jorge F. Garza.
Composite Objects Revisited.
In James Clifford, Bruce Lindsay and David Maier (editors), *1989 ACM SIGMOD International Conference on the Management of Data*, pages 337-437. ACM Press, Portland OR, June, 1989.
Special issue of *SIGMOD Record*, 18(2), June 1989.

[Kim 90] Won Kim, Jorge F. Garza, Nathaniel Ballou and Darrel Woelk.
Architecture of the ORION Next-Generation Database System.
IEEE Transactions on Knowledge and Data Engineering 2(1):109-124, March, 1990.

[Lecluse 90] C. Lecluse, P. Richard and F. Velez.
O_2, an Object-Oriented Data Model.
Readings in Object-Oriented Database Systems.
Morgan Kaufman, 1990, pages 227-236.

[Liskov 88] Barbara Liskov.
Distributed Programming in Argus.
Communications of the ACM 31(3):300-312, March, 1988.

[Liskov et al. 81] Liskov, B., *et al.*.
CLU Reference Manual.
Springer-Verlag, Berlin, FRG, 1981.

[Moss 88] J. Eliot B. Moss and Steven Sinofsky.
Managing Persistent Data with Mneme: Designing a Reliable, Shared Object Interface.
In *Advances in Object-Oriented Database Systems*, pages 298-316. Springer-Verlag, September, 1988.

[Moss 90] J. Eliot B. Moss, Antony L. Hosking, Bojana Obrenic and Steven Sinofsky.
A Performance Study of the Mneme Persistent Object Store.
In Hector Garcia-Molina and H.V. Jagadish (editor), *1990 ACM SIGMOD International Conference on Management of Data*. Atlantic City, NJ, May, 1990.
Special issue of *SIGMOD Record*, 19(2), June 1990.

[Neimat 90] Marie-Anne Neimat and Kevin Wilkinson.
Extensible Transaction Management in Papyrus.
In Bruce Shriver (editor), *23rd Annual Hawaii International Conference on System Sciences,* pages 503-511. Kona HI, January, 1990.

[Ontologic 89] Ontologic Inc.
Product Description.
March, 1989

[Popovich 90] Steven S. Popovich, Gail E. Kaiser and Shyhtsun F. Wu.
MELDing Transactions and Objects.
In *Object-Based Concurrent Systems Workshop*. Ottawa, Canada, October, 1990.
Position paper. In press.

[Popovich 91] Steven S. Popovich, Shyhtsun F. Wu and Gail E. Kaiser.
An Object-Based Approach to Implementing Distributed Concurrency Control.
In *11th International Conference on Distributed Computing Systems*, pages 65-72. Arlington TX, May, 1991.

[Rashid 87] Richard Rashid, Avadis Tevanian, Michael Young, David Golub, Robert Baron, David Black, William Bolosky and Jonathan Chew.
Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures.
In *2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 31-39. Palo Alto CA, October, 1987.
Special issue of *SIGPLAN Notices*, 22(10), October 1987.

[Richardson 87] Joel E. Richardson and Michael J. Carey.
Programming Constructs for Database System Implementation in EXODUS.
In *ACM SIGMOD Internation Conference on Data Management*, pages 208-249. San Francisco CA, May, 1987.
Special issue of *SIGMOD Record*, 16(3), December 1987.

[Richardson 89] Joel E. Richardson, Michael J. Carey and Daniel T. Schuh.
The Design of the E Programming Language.
Technical Report, University of Wisconsin, January, 1989.

[Spector 88] A. Z. Spector, R. Pausch and G. Bruell.
Camelot, A Flexible, Distributed Transaction Processing System.
In *33rd IEEE Computer Society International Conference*, pages 432-437. March, 1988.

[Spector 89] Alfred Z. Spector.
Modular Architectures for Distributed and Database Systems.
In *8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 217-224. March, 1989.

[Stonebraker 90] Michael Stonebraker, Lawrence A. Rowe, Bruce Lindsay, James Gray, Michael Carey, Michael Brodie, Philip Bernstein and David Beech.
Third-Generation Database System Manifesto.
SIGMOD Record 19(3):31-44, September, 1990.

[Velez 90] F. Velez, V. Darnis, D. DeWitt, P. Futtersack, G. Harrus, D. Maier and M. Raoux.
Implementing the O_2 Object Manager: Some Lessons.
In *4th International Workshop on Persistent Object Systems*, pages 129-136. Martha's Vineyard, MA, September, 1990.

[Wegner 87] Peter Wegner.
Dimensions of Object-Based Language Design.
In Norman Meyrowitz (editor), *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, pages 168-182. ACM Press, Orlando FL, October, 1987.
Special issue of *SIGPLAN Notices*, 22(12), December 1987.

[Wilkes 86] C. Thomas Wilkes and Richard J. LeBlanc.
Rationale for the Design of Aeolus: A Systems Programming Language for an Action/Object System.
Technical Report GIT-ICS-86/12, Georgia Institute of Technology, December, 1986.

[Wilkinson 90] Kevin Wilkinson, Peter Lyngbaek and Waqar Hasan.
The Iris Architecture and Implementation.
IEEE Transactions on Knowledge and Data Engineering 2(1):63-75, March, 1990.