An Introduction to Distributed Object Management*

M. Tamer Özsu
University of Alberta
Department of Computing Science
Edmonton, Alberta
Canada T6G 2H1

Umeshwar Dayal
Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94304
U.S.A.

Patrick Valduriez
INRIA Rocquencourt
78153 Le Chesnay Cedex
France

Abstract

Object-oriented data management systems (OODMSs) are expected to meet the requirements of new application domains which have a need for functionality beyond those provided by relational database systems. In addition, most of these application domains require the management of distributed artifacts. Therefore, next generation OODBMSs are likely to be distributed systems. This paper provides an overview of distributed object management issues and provides an introduction to the other papers included in this volume.

1 INTRODUCTION

Relational database systems have been very successful in meeting the requirements of traditional business applications. Some of the features of the relational systems which make them suitable for these applications become too restrictive when these systems are put to use in other application domains that are more complex than traditional business applications (e.g., office information systems, engineering databases, artificial intelligence applications,

*Published in:*Distributed Object Management*, M.T. Özsu, U. Dayal and P. Valduriez (eds.), Morgan-Kaufmann, 1994, pages 1–24.

medical imaging systems, geographical information processing systems, biological information systems). Some of the more important shortcomings of the relational model in meeting the requirements of these applications can be listed as follows:

1. Relational systems deal with a single object type: a relation. A relation is used to model different real-world objects, but the semantics of this association is not part of the database. Furthermore, the attributes of a relation may come only from simple and fixed data type domains (numeric, character, and, sometimes, date types). The advanced applications cited above require explicit storage and manipulation of more abstract types (e.g., images, design documents) and the ability for the users to define their own application-specific types . Therefore, a rich type system supporting user-defined abstract types is required.
2. The relational model structures data in a relatively simple and flat manner. These applications require more complex object structures with nested objects (e.g., a vehicle object containing an engine object).
3. Relational systems provide a declarative and (arguably) simple language for accessing the data. The applications mentioned above require richer languages which overcome the well-known "impedance mismatch" problem. This problem arises because of the differences in the level of abstraction between the relational languages and the programming languages with which they interact.

Object-oriented technology is now a topic of intense study as the major candidate to successfully meet the requirements of advanced applications that require data management services. However, a close study of these applications reveals that they are distributed in nature and require data management support in a distributed environment. Thus, these systems require the development of distributed object management systems (DOMs). The purpose of this paper is to provide an overview of the issues that need to be addressed in the development of this technology.

Even though the work on OODBMSs is relatively new, there has been more than a decade of work in relational distributed DBMSs. It is possible that the lessons learned in that process would be useful in developing distributed OODBMSs. In fact, we view distributed OODBMSs as the merger of the relational distributed DBMS and the OODBMS technologies (Figure 1). Therefore, we start with a review of distributed data management in the Section 2 and object-orientation in Section 3, before we embark on a discussion of issues that arise in developing distributed OODBMSs in Section 4.

2 DISTRIBUTED DATA MANAGEMENT

A distributed database is a collection of multiple, logically interrelated databases distributed over a computer network [Özsu and Valduriez, 1991a]. A distributed DBMS is then defined as the software system that permits the management of the distributed database and makes the distribution transparent to the users. This definition is based on the following implicit assumptions [Özsu and Valduriez, 1991b]:

1. Data is stored at a number of sites. Each site is assumed to logically consist of a single processor.
2. The processors at these sites are interconnected by a computer network rather than a multiprocessor configuration.

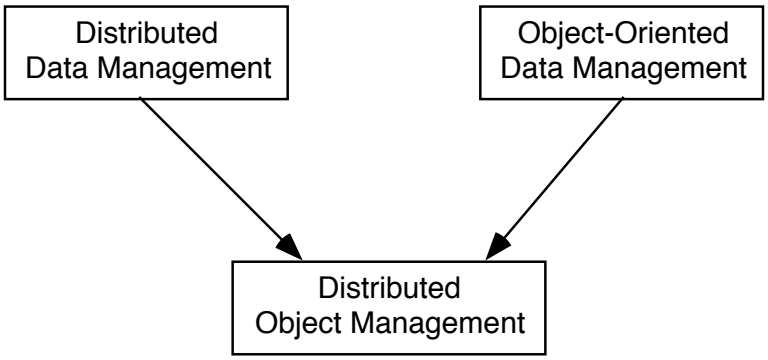

Figure 1: Technologies Contributing to DOM

3. The distributed database is a "database", not some "collection" of files that can be individually stored at each node of a computer network.

4. The system has the full functionality of a DBMS. It is neither, as indicated above, a distributed file system, nor purely a transaction processing system.

The last two assumptions indicate that the system has to be a fully-functional DBMS. The first two assumptions make the distinction between *distributed DBMS* and *parallel DBMS*. As indicated in [Özsu and Valduriez, 1993], next generation DBMSs will relax many of these assumptions due to technological developments — especially the emergence of affordable multiprocessors and high-speed networks — and the increasing use of client-server mode of computing. Thus, next generation systems will include parallel database servers connected to high speed networks which link them and other repositories to client machines that run application code and participate in the execution of database requests. It is within this context that object-oriented systems should be viewed. Even though our emphasis in this paper and in this volume is predominantly on distributed object-oriented DBMS, there has been some work on implementing these systems on top of parallel architectures as well (e.g., Bubba [Boral et al., 1990]).

Distributed DBMSs provide transparent access to a physically distributed database. User queries and transactions access the database based on *external schemas* (ES) defined over a *global conceptual schema* (GCS) which is partitioned and stored at different sites forming the *local conceptual schemas* (LCS) at those sites. The LCS at a given site is then mapped to a *local internal schema* (LIS) which describes the physical organization of data at that site (Figure 2).

Transparency , in this context, can be viewed as the extension of the well-known *data independence* concept to distributed systems. The following transparencies have to be provided by the distributed DBMS in order to achieve full data independence in a distributed environment: *network* (or *distribution*) *transparency* which hides the existence of the network and the distribution of data, *replication transparency* which masks the existence of physical copies of logical data items, and *fragmentation transparency* which prevents the user from having to deal with the partitioning of database objects.

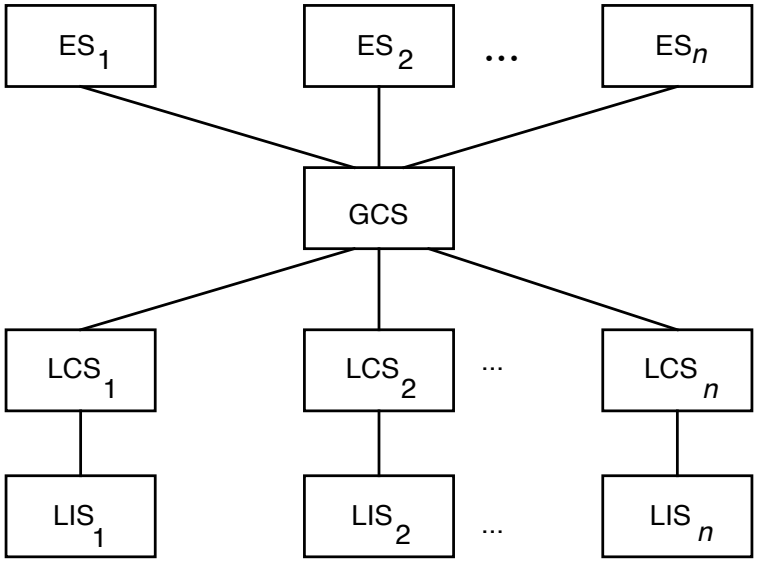

Figure 2: Datalogical Distributed DBMS Architecture

Various distributed DBMS modules cooperate to provide the functionality described above. **Distributed query processors** partition global queries to a set of sub-queries that can execute at each site. This partitioning is done transparently, using the location information in the distributed database directory. Distributed processors also optimize the distributed operation of binary operators (e.g., join, union). This partitioning of a given global query introduces *intra-query parallelism* that contributes to improved performance. The inherent parallelism in a distributed system enable queries to be submitted at a number of sites giving rise to *inter-query parallelism* . Further optimization (access path selection) of each sub-query is done at each site where it is executed.

Transactional accesses are managed by **global transaction managers** with the help of **schedulers** for synchronization and **local recovery managers** for reliability. The placement and operation of the schedulers are dependent upon the concurrency control algorithms that are implemented. The most common algorithm is some form of two-phase locking . Transaction managers are designed to specifically support a particular transaction model (see Section 4.4 for more discussion of models). They participate in the execution of an atomic commit protocol (usually two-phase commit) to provide transaction atomicity . Local recovery managers participate in the execution of the atomic commit protocol in addition to implementing reliability protocols (to ensure durability) and recovery routines.

3 OBJECT-ORIENTED DATA MANAGEMENT

What constitutes an object-oriented database system is a topic of considerable discussion. Contrary to the relational model, there is no universally accepted and formally specified object model. There are a number of features that are common to most model specifications,

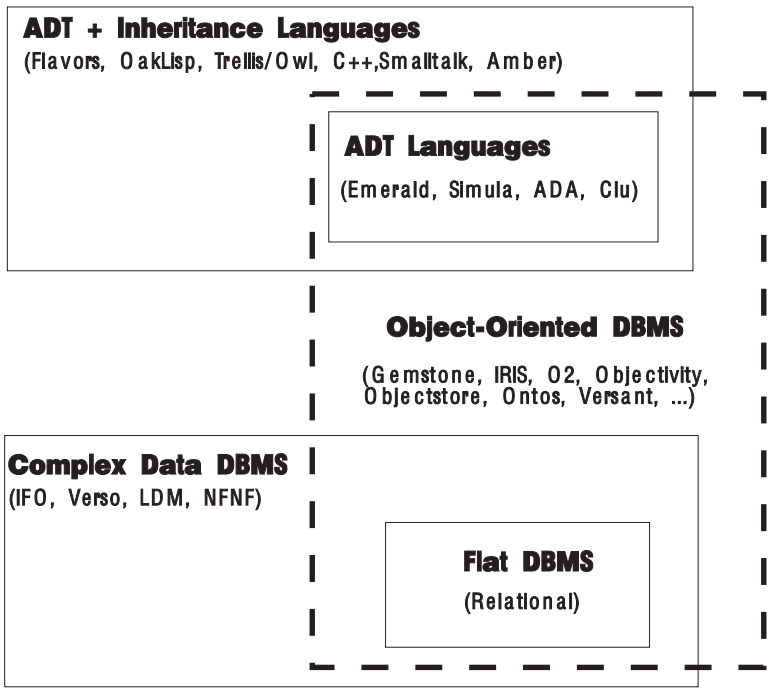

Figure 3: Merging of Technologies

but the exact semantics of these features are different in each model. In this paper, we will review the fundamental concepts assuming a fairly conventional object model.

Object-oriented database systems combine the features of object-oriented programming languages with those of database management systems. The former contributes the notions of types and instances, object identity , encapsulation , inheritance and navigational access, whereas the latter provides support for persistence , concurrency control , recovery , consistency , and query access. Figure 3 shows the merger of these technologies in producing an OODBMS.

All object-oriented DBMSs are built around the fundamental concept of an *object*. An object represents a real entity in the system that is being modeled. It is an abstraction defined by (a) a unique *object identifier* , (2) a set of *instance variables* (attributes) which define the *state* of the object, and (3) an interface in terms of *methods* (operations) which manipulate the object. Some object models (e.g., OODAPLEX [Dayal, 1989] and TIGUKAT [Peters et al., 1992]) take a behavioral approach and define a set of behaviors which are the only means of accessing and manipulating objects. In these models, there are no instance variables indexinstance variable or methods , there are only *behaviors* . In either case, the structure and implementation of objects are hidden and cannot be accessed by others. The only way of accessing objects is by their *interface* which is either the set of behaviors or the set of instance variables and methods. This is a fundamental feature of object-orientation and is called *encapsulation*. Encapsulation provides data abstraction and data independence.

Object identity (OID) is another fundamental feature of object models since they form the foundation of object comparisons and referencing. An OID is an *invariant* property of an object which distinguishes it logically and physically from all other objects. It is unique to each object and independent of the object's state . In some object models OID equality is the only comparison primitive; for other types of comparisons, the type definer is expected to specify the semantics of comparison. In other models, comparisons with respect to object states is allowed (e.g., shallow and deep equality). OIDs also form the basis of *referential sharing* where multiple objects contain the OID of another object as the value of one of their instance variables . These objects are also known as *composite objects* .

Object models also differ in how they deal with types, classes and collections. A *type* defines a template for all the objects of that type. It provides a common representation for all objects of that type and a set of operations on those objects. A *class* is a grouping of all object instances of a given type. It is the *extent* of the associated type. A *collection* is a user-defined grouping of objects which differs from a class in that it can be heterogeneous and objects need to be inserted into it explicityly by users. In most conventional systems a class is not separated from a type and serves both as a type specification and as an extent of that type. Beeri argues that there is use for both of classes and collections and that they should be supported in an object model [Beeri, 1990]. Collections provide for a clear closure semantics of the query models and facilitate definition of user views . Some of the recent models (e.g., TIGUKAT [Peters et al., 1992]) separate types and classes and also provide collections.

Object-oriented systems provide extensibility by allowing user-defined types to be defined that will be managed by the system. Typically, these types are defined based on predefined types. This is the process of subtyping. A type `A` is a *subtype* of another type `B` if it inherits all the behaviors (or instance variables and methods) of `B` and possibly more. It is possible for a type to be subtypes of more than one type, in which case the model supports *multiple subtyping*.

What we called subtyping above has commonly been called *inheritance* . In models where there is no distinction between a type and a class, subtyping and inheritance can be used interchangeably. In models which separate the two, however, inheritance is associated with code sharing, and is, therefore, treated as an implementation concept, whereas subtyping is considered a behavioral concept. Subtyping indicates a relationship between types. In the above example, `A` **is-a** `B`, resulting in *substitutability* : an instance of a subtype can be substituted in place of an instance of any of its supertypes in any expression. If this separation is made between subtyping (as a behavioral concept) and inheritance (as an implementation concept for code sharing), one gets a clear separation of concepts. However, merging the two makes sense in models where the class and type concepts are not separated.

The fundamental feature that database technology contributes to OODBMSs is *persistence* , which is the property of objects to exist beyond the duration of the process that creates them. Two important properties of persistence have been identified [Atkinson, 1993]:

1. **Persistence independence.** The semantics of a program is not changed by the longevity of the objects to which it is applied.
2. **Persistence orthogonality.** Any object of any type can be made persistent.

A number of different persistence models have been adopted in various OODBMSs. Most

straightforward persistence model is to attach persistence to objects, thereby making an object persistent only if it is declared as such. This model provides persistence orthogonality and it can also support persistence independence if the OODBMS treats persistent and transient objects uniformly. A second model is to associate persistence with the grouping construct (e.g., a collection). In this case, an object is persistent if it is placed in a collection which is declared to be persistent. A third model is to associate persistence with types. Thus an object is persistent if its type is declared to be persistent. To provide persistence orthogonality in this model, it is necessary to define the persistent (transient) dual of each transient (persistent) type (e.g., have both a `TransientInteger` and a `PersistentInteger` type). Finally, persistence can be associated with reachability. In this case, the type semilattice has a persistent root and any object that is reachable (by following the subtype and instance_of relationships) from that root is persistent.

Management of persistence is another important issue. One of the questions is the duration of persistence . There have been some research projects which treat all objects as persistent (without a clear deletion semantics). In these systems, objects are considered to be permanently persistent. More realistically, however, the two alternatives are (a) objects are persistent until they are explicitly deleted, or (b) objects are persistent until they are unreachable from any other object (i.e., no object holds reference to them any longer). In the latter case, garbage collection routines take care of reclaims the storage (distributed versions of these algorithms are discussed later in this paper). A second question that arises is with respect to how objects are shared. Two alternatives have been identified. First is what is called the workspace model where objects, when they are brought into memory, are copied into the requesting application's address space. Thus, the objects are checked-out of the database buffers and need to be checked-in later when the application releases them. In the second model, persistent objects reside in shared database buffers and the OODBMS accesses them on behalf of each application.

In this section we provided an overview of the more important object-oriented concepts. This was not meant to be a complete tutorial; much better ones are available. We have, hopefully, established a platform upon which the distributed object management concepts can be presented in the subsequent sections.

4 DISTRIBUTED OBJECT MANAGEMENT

Distributed object management has the same objective with regards to object-oriented data management as the traditional database systems had for relational databases: transparent management of "objects" that are distributed across a number of sites. Thus users have an integrated, "single image" view of the objectbase (Figure 4) while it is physically distributed among a number of sites (Figure 5).

Maintaining such an environment requires that problems similar to those of relational distributed databases be addressed, with the additional opportunities and difficulties posed by object-orientation. In the remainder, we will review these issues and indicate some of the approaches.

4.1 ARCHITECTURE

The first step that system developers seem to take, on the way to providing true distribution by peer-to-peer communication, is to develop client-server systems . This was true in

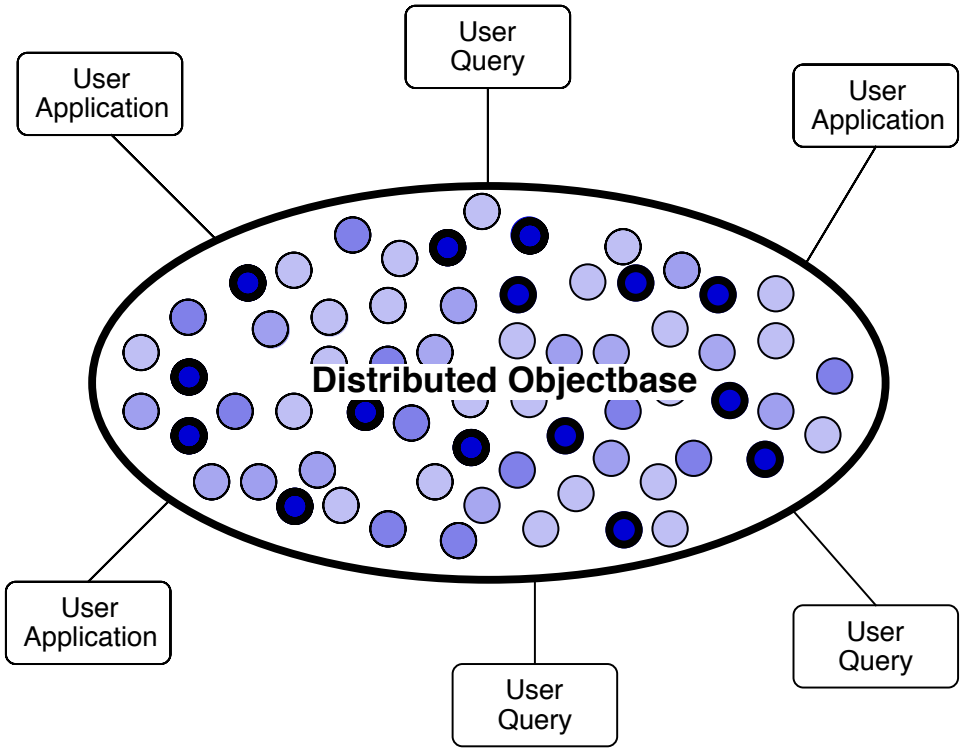

Figure 4: Transparent Access to Objectbase

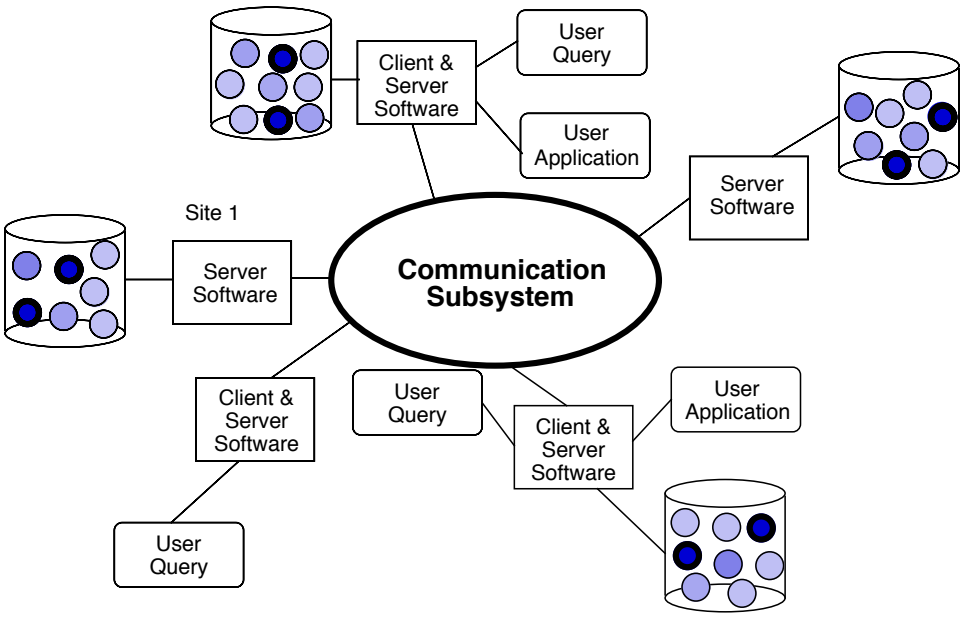

Figure 5: Distribution of Objectbase

relational distributed DBMSs, and it certainly seems to be true for OODBMSs as well. Most of the existing OODBMSs are *multiple client-single server* architectures where the objectbase is stored at and managed by a single server which provides services to a number of clients. Such systems are usually called *data shipping* systems [Franklin and Carey, 1993] to differentiate them from distributed systems that provide simple print server services, etc.

The design of data shipping client-server systems requires the resolution of a number of design issues. One issue is the communication mechanism between the clients and the server. The two alternatives are remote procedure calls (RPC) and message passing . The relative merits of these schemes have been discussed extensively in literature (e.g., see [Stankovic, 1982]). In Section 4.6, we provide an brief overview and comparison among them. A second important design issue is to determine the functions to be provided by the clients and the server. This is at least partly dependent upon the the resolution of a third issue, namely the unit of communication between the clients and the server. Two alternatives exist which result in two types of client-server architectures . The first alternative is that the clients request "objects" from the server which retrieves them from the database and returns them to the requesting client. These systems are called *object servers* (Figure 6). In object servers, the server undertakes most of the object management services with the client providing basically an execution environment for the applications as well as providing some level of object management functionality. The optimization of user queries and the synchronization of user transactions are all performed in the server with the client receiving the resulting objects. Object manager serves a number of functions. First and foremost, it provides a context for method execution. The replication of the object manager in both the server and the client enables methods to be executed at both the server and the clients. The execution of methods in the client may invoke the execution of other

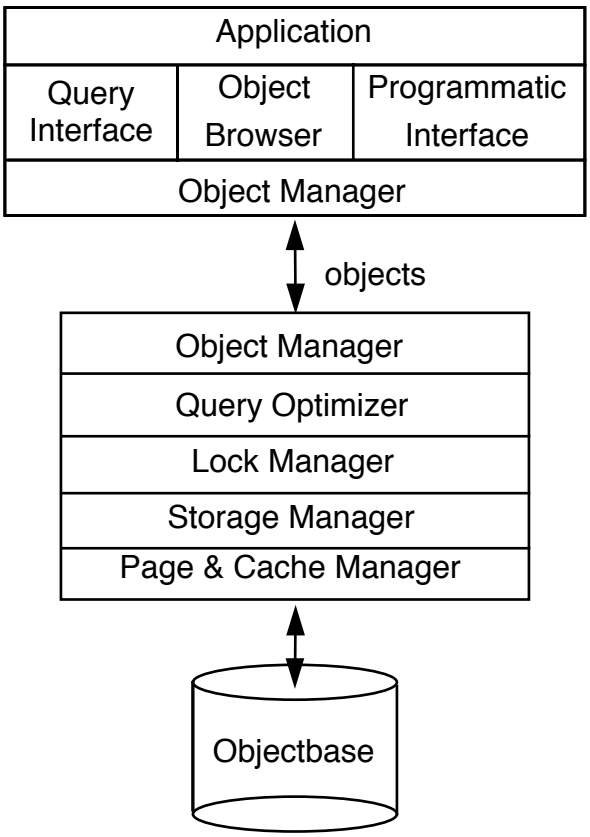

Figure 6: Object Server Architecture

methods which may not have been shipped to the server with the object. The optimization of method executions of this type is an important research problem which is addressed in [Maier et al., 1993]. Object manager also deals with the implementation of the object identifier (logical, physical, or virtual) , the deletion of objects (either explicit deletion or garbage collection). At the server, it also provides support for object clustering and access methods.

An alternative organization is a *page server* client-server architecture where the unit of transfer between the servers and the clients are physical units of data such as pages or segments rather than objects Figure 7). Page server architectures split the object processing services between the clients and the servers. In fact, the servers do not deal with objects anymore, instead acting as "value-added" storage managers.

Early performance studies (e.g., [DeWitt et al., 1990]) indicate that page server architectures perform better than object server architectures. Further studies on these designs are necessary before a definitive conclusion can be reached, however. It would be interesting to repeat the performance studies in multiple client-multiple server environments with differ-

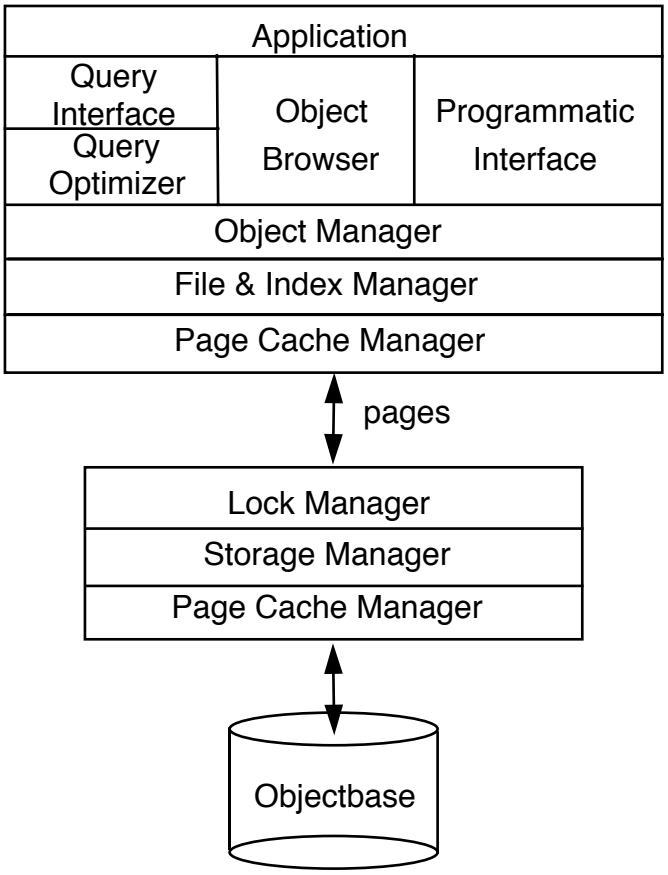

Figure 7: Page Server Architecture

ent job mixes (query versus navigational access) and with different storage system designs taking into account different buffer management policies.

A final design issue that we address relates to caching policies. There has been some work done in this regard ([Carey et al., 1991; Franklin and Carey, 1993]). Here we summarize the results of [Franklin and Carey, 1993]. This study considers data shipping, multiple client-single server, page server architectures. It compares the performance of three types of caching algorithms: (1) *server-based two-phase locking* where data pages are cached to the clients, but locks are managed by the server; (2) *optimistic two-phase locking* (O2PL) where both data pages and locks are cached and the detection of conflicts of locks cached at multiple clients are deferred until transaction commit time; and (3) *callback locking* (CL) where both data pages and locks are cached at the clients, but locks are obtained (if they are not already cached) from the server before the data pages can be accessed. Four versions of O2PL are analyzed: one which invalidates the cached data pages if there is a conflict (O2PL-I) , one which propagates the changes to the cached copies (O2PL-P) , one which dynamically chooses between invalidation and propagation (O2PL-D) , and an improved version of O2PL-D (O2PL-ND) which is proposed by the authors. For callback locking, two variants are considered: callback-read which allows for caching of read-locks only, and callback-all which allows caching of both the read- and write-locks. The throughput of these algorithms are compared with respect to four different workloads. Simply stated, the results indicate that callback-read performs as well or slightly better than callback-all in most situations. Furthermore, the O2PL-ND algorithm performs close to O2PL-I when invalidation is the right choice and close to O2PL-P when propagation is the right choice. It always performs better than O2PL-D and slightly better than the callback algorithms.

The client-server architectures that we have discussed above are relatively traditional ones. An interesting proposal that is included in [Atkinson, 1993] is to construct a distributed persistent object system as a substrate to the various components (operating systems, DBMSs, persistent programming languages , etc) that are used to develop large-scale, long-lived applications. The proposal is motivated by two observations: (1) development of these types of applications can be greatly aided if a persistent programming language is used, since this reduces the number of mappings that the programmer has to deal with, and (2) many of these components that are used in developing these systems, "despite the fact that they service very different needs, have a substantial number of sub-components serving common functional requirements." These activities are identified as persistence , stability, recovery , concurrency , scheduling, space administration, protection, accounting, resource allocation and control, naming , binding programs to data (see Figure 6 in [Atkinson, 1993]). The paper identifies four possible architectural approaches to supporting the development of persistent application systems: (1) ignore it and hope that it will be taken care of by market forces or that it will go away; (2) make sure that all of the components employ common standards for all the common activities; (3) provide a software layer on top of these components that hides their differences; and (4) provide a common substrate that provides these common functions. Atkinson's proposal is to follow the last approach and develop what is called a *scalable persistent foundation* (SPF). The architecture of such a system is depicted in Figure 8 of [Atkinson, 1993].

Open system architectures for DOMs have been the topic of some study. One such effort, carried out at Texas Instruments and supported by DARPA, it concentrates on the definition of an architectural framework of an open OODBMS and on the description of the design space for these systems [Wells and Blakeley, 1993]. The system employs a computational

model where objects exist in a universe of objects partitioned into environments according to certain criteria (e.g., address space) that are transparent to applications. The architecture consists of a *meta-architecture* consisting of a collection of glue modules and definitions providing the infrastructure for extending the operations of programming languages, and an extensible collection of policy performers (PPs). The meta-architecture contains a collection of address space managers, communications and translation managers for transferring objects between multiple address spaces (if there are more than one), a data dictionary, and a meta-architecture support module that provides the extension mechanism and defines the interface conventions to which policy modules must adhere. The PP modules that are instantiated in a particular open OODBMS may vary, but the full set includes transaction , distribution , replication , index, object query processor, version, configuration, dependency (for consistency and derived objects), persistence , access control, and gateway (for access to objects in foreign databases) policy performers. There is on-going work in most of these PPs except for the replication, access control and gateway PPs.

4.2 OBJECT DISTRIBUTION AND MIGRATION

The two important aspects of distribution design are fragmentation and allocation . Distribution design in the object-oriented world brings new complexities. Conceptually, objects encapsulate behaviors (methods) together with state . In reality, behaviors are implemented on types and shared by all instance objects of that type. Therefore, the location of objects with respect to their types becomes an issue. Objects can be co-located with their types, in which case different types of objects are located at different sites. Alternatively, objects of one type can be located at different sites. This is similar to horizontal partitioning in relational database systems (called *horizontal class partitioning* in [Karlapalem et al., 1993]). In this case, types may be duplicated at each site where there is an instance of that type, or the type specification may be stored at one site and remote processing of objects is supported as described below. The final alternative is to partition the type specification by dividing its behaviors (or instance variables and methods) into different fragments (called *vertical class partitioning* in [Karlapalem et al., 1993]) and distributing the objects in that manner. Again, in this case, there is a need for remote object processing.

Replication adds a new dimension to the design problem. Individual objects, classes of objects or collections of objects (or all) can be units of replication. Undoubtedly, the decision is at least partially object model dependent. Whether or not type specifications are located at each site can also be considered as a replication problem.

In horizontal and vertical partitioning , it is possible that the objects are not located at the site where the behaviors or methods (usually implemented as functions) that operate on them. This problem is addressed within the context of interoperable systems in [Fang et al., 1993], but the same techniques are applicable in general. Four alternatives can be identified: (1) local function – local argument, (2) local function – remote argument, (2) remote function – local argument, and (4) remote function – remote argument. The paper discusses solutions to all of these alternatives. Access to remote arguments is accomplished by creating surrogates in the local database that point to their remote counterparts. Local or remote functions operate on these surrogates. In the case of local stored functions applied to remote objects that do not have their counterparts in the remote component, new local function values are created as surrogates are created.

4.2.1 Storage Management and Object Clustering

An object model is essentially conceptual and should provide high physical data independence in order to increase programmer productivity. The mapping of this conceptual model to a physical storage is a classical database problem. In the case of OODBMSs, two kinds of relationships exist between types (or classes , depending upon the object model): subtyping or inheritance (e.g., an employee is a person) and composition (e.g., an employee works in a department). By providing a good approximation of object access, these relationships are essential to guide the physical clustering of persistent objects. Object clustering refers to the grouping of objects in physical containers (i.e., disk extents) according to common properties, such as the same value of an attribute or subobjects of the same object. Thus, fast access to clustered objects can be obtained.

Object clustering is difficult for two reasons. First, it is not orthogonal to object identity implementation, i.e, logical vs. physical OID . Logical OIDs incur more overhead (an indirection table) but enable vertical partitioning of classes. Physical OIDs yield more efficient direct object access but require each object to contain all inherited attributes. Second, the clustering of complex objects along the composition relationship is more involved because of object sharing (objects with multiple parents).

Given a class graph , there are three basic storage models for object clustering [Valduriez et al., 1986]. The *decomposition storage model* (DSM) partitions each object class in binary relations (OID, attribute) and therefore relies on logical OID. The advantage of DSM is simplicity. The *normalized storage model* (NSM) stores each class as a separate relation. It can be used with logical or physical OID. However, only logical OID allow the vertical partitioning of objects along the inheritance relationship [Kim et al., 1987] The *direct storage model* enables multi-class clustering of complex objects based on the composition relationship. This model generalizes the techniques of hierarchical and network databases and works best with physical OID [Benzaken and Delobel, 1991]. This model can capture object access locality and is therefore potentially superior for well-known access patterns. The major difficulty, however, is to be able to recluster an object whose parent it is grouped with has been deleted.

In a distributed system, both DSM and NSM are straightforward using horizontal partitioning. Goblin [Kersten et al., 1993] implements DSM as a basis for a distributed OODBMS with large main memory. DSM provides flexibility and its performance disadvantage is compensated by the use of large main memory and caching. Eos [Gruber and Amsaleg, 1993] implements the direct storage model in a distributed single-level store architecture where each object has a physical system-wide OID. The Eos grouping mechanism is based on the concept of most relevant composition links and solves the problem of multiparent shared objects. When an object moves to a different node, it gets a new OID. To avoid the indirection of forwarders, references to the object are subsequently changed as part of the garbage collection process without any overhead. The grouping mechanism is dynamic to achieve load balancing and cope with the evolutions of the object graph.

4.2.2 Distributed Garbage Collection

An advantage of object-based systems is that objects can refer to other objects using

object identity . As programs modify objects and remove references, a persistent object may become unreachable from the persistent roots of the system when there is no more reference to it. Such an object is garbage and should be de-allocated by the garbage collector . In relational DBMSs, there is no need for automatic garbage collection since object references are supported by join values. However, cascading updates as specified by referential integrity constraints are a simple form of "manual" garbage collection. In more general operating system or programming language contexts, manual garbage collection is typically error-prone. Therefore, the generality of distributed object-based systems calls for automatic distributed garbage collection .

There are two basic techniques for garbage collection : *reference counting* and *tracing* . With reference counting , each object carries a count of the number of references to it. When the number reaches zero, the object is garbage and can be de-allocated. Thus, garbage collection is done during object update. With tracing, garbage collection is a periodic process which walks the object graph and detects unreachable objects.

Reference counting seems to be the most suited for distributed systems [Bevan, 1987]. However, dealing with garbage cycles is difficult (the count of the cycle root can never be zero). Furthermore, the basic problem is that maintaining a strong invariant (equality of the count with the actual number of references) is difficult in the presence of failures.

Tracing is potentially more fault-tolerant because each execution of the collector starts from the root [Dijkstra et al., 1978]. However, most distributed tracing algorithms are usually very complex because they try to detect the minimal set of reachable objects. This is difficult and costly in the case of distributed cycles and a global service is needed to synchronize the local collectors.

However, when the assumptions are weakened, simpler protocols are possible. For instance, with the assumptions of fail-stop crashes only and that messages are either lost or delivered in finite time, [Shapiro et al., 1990] proposes an algorithm with tracing and local garbage collection but no global service. Nevertheless, as in most solutions, the detection of inter-space cycles of garbage remains an open issue.

4.2.3 Object Migration

One aspect of distributed systems is that objects move, from time to time, between sites. This raises a number of issues in object management systems. First is the unit of migration. In systems where the state is separated from the methods , it is possible to consider moving the object's state without moving the methods. The counterpart of this scenario in purely behavioral systems is the fragmentation of an object according to its behaviors . In either case, the application of methods to an object requires the invocation of remote procedures. This is an issue that we discussed above under object distribution. Other systems (e.g. [Dollimore et al., 1993]) consider the object as the unit of distribution. Even then, the migration of individual objects may move them away from their type specifications and one has to decide whether types are duplicated at every site where instances reside or whether the types are accessed remotely when behaviors or methods are applied to objects.

Another issue is that the movements of the objects have to be tracked so that they can be found in their new locations. A common way of tracking objects is to leave *surrogates* [Hwang, 1987; Liskov et al., 1993] or *proxy objects* [Dickman, 1993b]. These are place-holder objects left at the previous site of the object, pointing to its new location. Ac-

cesses to the proxy objects are directed transparently by the system to the objects themselves at the new sites. The migration of objects can be accomplished based on the state that they are in [Dollimore et al., 1993]. Objects can be in one of four states: ready, active, waiting, or suspended. Ready objects are those that are not currently invoked (or have not received a message) but are ready to be invoked (to receive a message). Active objects are those which are currently involved in an activity in response to an invocation or a message. Waiting objects are those which have invoked (or have sent a message to) another object and are waiting for a response and suspended objects are temporarily unavailable for invocation. Objects in active or waiting state are not allowed to migrate since the activity they are currently involved in would be broken. The migration involves two steps: (1) shipping the object from the source to the destination, and (2) creating a proxy at the source, replacing the original object.

Two related issues have to be addressed here. One relates to the maintenance of the system directory. As objects move, the system directory needs to be updated to reflect the new location. This may be done lazily whenever a surrogate or proxy object redirects an invocation rather than eagerly at the time of the movement. The second issue is that in a highly dynamic environment where objects move frequently, the surrogate or proxy chains may become quite long. It is useful for the system to transparently compact these chains from time to time. However, the result of compaction needs to be reflected in the directory and it may not be possible to accomplish that lazily.

Another important migration issue arises with respect to the movement of composite objects . The shipping of a composite object may involve shipping other objects referenced by the composite object (called the composition graph). A lazy migration strategy is employed in [Dollimore et al., 1993]. Three alternatives are considered for the migration of classes (types): (1) the source code is moved and recompiled at the destination, (2) the compiled version of a class is migrated just like any other object, or (3) the source code of the class definition is moved, but not its compiled operations for which a lazy migration strategy is used.

4.3 QUERY PROCESSING

In distributed relational database systems, query processing consists largely of generating, and executing, a low-cost plan for decomposing the query into operations that are shipped to various nodes, sequencing these operations, and piecing the results of these operations to construct the query's result. Since data movement across nodes of the distributed system is a key factor contributing to the cost of query processing, the plan consists of *bulk* operations that operate on and move sets of records, instead of one record at a time.

The same considerations carry over to distributed object systems. However, unlike relational systems, objects are not always implemented as flat records. Objects may have complex states consisting of many subobjects distributed over several nodes, and objects may reference other objects. Performing an operation over a collection of complex objects typically requires accessing these distributed pieces of state and following embedded references among the subobjects. A naive implementation of the operation, for example one that tries to access each piece of state for one object at a time, may incur unacceptable performance costs due to navigating across the network, following inter-object references, and making remote procedure calls . How to efficiently assemble the fragments of several distributed objects' state necessary to processing a query is addressed in [Maier et al., 1993].

In earlier work, the *assembly* operation had been introduced for use in centralized object systems [Keller et al., 1991]. This operation efficiently assembles the fragments of objects' states required for a particular processing step, and returns them as a complex object in memory. It translates the disk representations of complex objects into readily traversable memory representations. Efficiency is achieved by reordering disk accesses to exploit the physical locations of objects, sharing of subobjects, clustering and selectivity information, and data already present in database buffers. The assembly operation resembles a functional join in that it links objects based on inter-object references.

In [Maier et al., 1993], several strategies for implementing a distributed assembly operation are discussed. One strategy involves shipping all data to a central site for processing. This is straightforward to implement, but could be inefficient in general. A second strategy involves doing simple operations (e.g., selections, local assembly) at remote sites, then shipping all data to a central site for final assembly. This strategy also requires fairly simple control, since all communication occurs through the central site. The third strategy is significantly more complicated: perform complex operations (e.g., joins, complete assembly of remote objects) at remote sites, and then ship the results to the central site for final assembly. A distributed object system may include some or all of these strategies.

4.4 TRANSACTION MANAGEMENT

The transaction concept has been used effectively in traditional database systems to synchronize concurrent accesses to a shared database and to provide reliable access in the face of failures. A transaction is an atomic unit of work against the database. ACID properties of transactions (atomicity , consistency , isolation , and durability) guarantee correct concurrent execution as well as reliability [Bernstein et al., 1987; Gray and Reuter, 1992].

It is commonly accepted that the traditional flat transaction model would not meet the requirements of the advanced application domains that object-oriented data management technology would serve. Some of the considerations are that transactions in these domains are longer in duration requiring interactions with the user or the application program during its execution. In the case of object-oriented systems, transactions do not consist of simple read/write operations, necessitating, instead, synchronization algorithms that deal with complex operations on abstract (and possibly complex) objects . In some application domains, the fundamental transaction synchronization paradigm based on competition among transactions for access to resources has to change to one of cooperation among transactions in accomplishing a common task. This is the case, for example, in cooperative work environments.

In [Martin and Pedersen, 1993] four example applications are considered to argue the point. These examples are (1) collaborative work in an office environment (the "shared desk top" metaphor), (2) a concurrent `make` facility, (3) editing a large document, (4) a CASE environment, and (5) cooperative arrangement of meeting rooms. A number of extensions to the traditional transaction model are identified from these examples categorized as those that *preserve transaction isolation* and those that *isolate cooperation*. The former class of extensions provide transaction properties more flexibly by capturing application semantics while the latter allow localized cooperation between a group of transactions . To preserve isolation, three specific extensions are proposed:

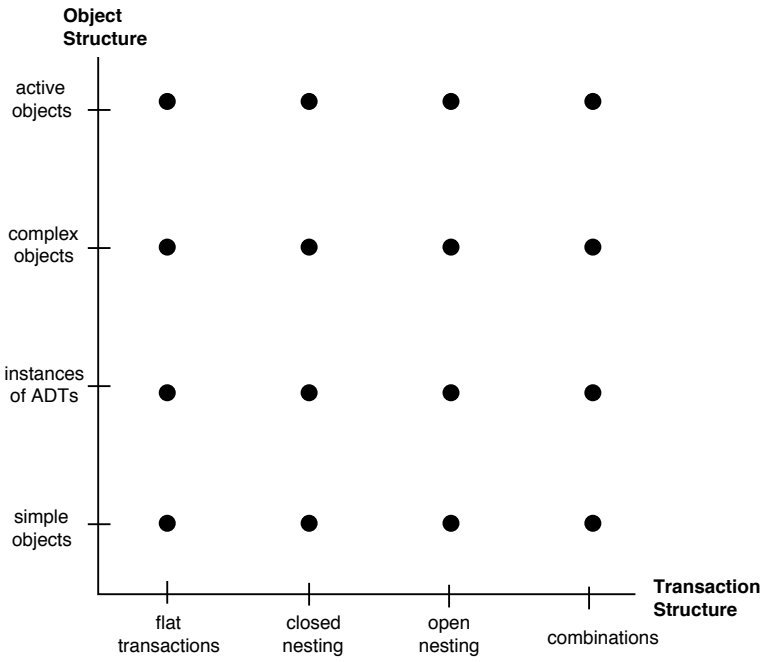

Figure 8: Transaction Model Alternatives

1. use of semantics-based concurrency control using commutativity of operations on objects;
2. identification of the layers of abstraction of a concurrent activity so that low level ordering dependencies can be ignored if higher level operations commute;
3. separation of the boundaries for failure atomicity and serializability to provide more flexibility.

To support isolation cooperation, the concept of *cooperative atomicity* is introduced. A group of concurrent activities is externally viewed as a single action and synchronized with respect to the group as a whole. Internally, the group of cooperating activities synchronize their own accesses to shared data.

There are a number of ways to characterize transaction management systems. In [Buchmann et al., 1992] these systems are characterized along two dimensions: transaction model and consistency definition. Transaction model is represented by transaction structure and the structure of objects that they operate on (Figure 8).

Each of the points in Figure 8 needs to be mapped to a correctness criterion. Serializability is the most common one of these, but there are others which provide more concurrency. It is not easy, however, to characterize these criteria. In [Ramamritham and Chrysanthis, 1993], the authors develop a taxonomy of correctness criteria and the formal specification of different classes using ACTA [Chrysanthis and Ramamritham, 1990]. The taxonomy follows two dimensions: *database consistency requirements*, and *transaction correctness*

properties . Under database consistency requirements, two issues are considered. The first is the unit of consistency, which specifies the sphere of application of the consistency criteria. If the consistency unit is the complete database, then all of the objects in the database have to be locally as well as mutually consistent. The consistency unit can be (location-independent or location-dependent) subsets of the database where mutual consistency is required only for objects in the same subset. Finally, the consistency unit may be specified as individual objects wherein each object in the database is expected to be locally consistent. The second issue that is covered under database consistency requirements is consistency maintenance. The questions that are addressed are "When is a consistency requirement expected to hold?" and "If a consistency requirement does not hold at a point it is supposed to, how is it restored?" Three alternative responses to the first question are identified: at activity boundaries, at specific points of time, and at specific database sates. Two responses to the second question are possible. One is that no restoration of consistency is possible if it does not hold at the point it is supposed to. The second is that consistency is restored in a deferred manner.

The second dimension of the taxonomy is transaction correctness properties . These properties can be presented with respect to four criteria: correctness of transaction results, correctness of transaction structure , correctness of data access related transaction behavior , and correctness of temporal behavior of transactions.

Using the characterization of Figure 8 and the classification of correctness criteria given in [Ramamritham and Chrysanthis, 1993], it is possible to identify a large number of the transaction management systems. For example, the point at the lower left-hand corner of the figure (flat transactions on simple objects) coupled with serializability characterizes many of the traditional transaction managers. This point is very well-studied and well-understood. However, object-oriented DBMSs, especially if they have active capabilities (see next section), require transaction managers that reside in the upper right-hand corner of Figure 8 and use correctness criteria that are significantly more relaxed than serializability . The move to that point is not straightforward.

There have been a number of studies along the axes of Figure 8. Of special interest are those studies that deal with nested transaction models and those that operate on abstract data types and on complex objects . Early studies on using a semantics-based correctness criterion to deal with operations on abstract data types are discussed in [Weihl, 1989a] and [Herlihy and Weihl, 1988]. These studies use the commutativity relationship among abstract operations defined over abstract data types to synchronize transactions . The consequence is that the degree of concurrency increases since read-write or write-write operations may commute and may, therefore, be allowed to execute concurrently (e.g., the `enqueue` and `dequeue` operations on a queue commute, and can execute concurrently, even though they correspond to a write-read pair). In [Nakajima, 1993], the work of Weihl is extended. Weihl defines two commutativity-based conflict relations (forward commutativity and backward commutativity) as well as describing concurrency control algorithms to enforce them. These two relations are somewhat difficult to work with since they are quite independent because one depends on the latest committed history while the other depends on the current history (consequently, one is not a subset of the other) and since they require different recovery algorithms . Thus, one or the other has to be chosen for enforcement even though each permit certain operation histories that the other one rejects. Nakajima defines a *general commutative relation* which is a superset of both the forward and the backward commutativity relations [Nakajima, 1993]. To support this generality, the paper argues for

the use of multiversion objects since they enable access, at the same time, to both the latest committed states of objects and to their current states . The paper describes concurrency control and recovery algorithms to enforce general commutativity. It also provides a high level discussion of the cost of the approach, but no detailed performance evaluation is provided.

A more general transactional model for supporting advanced applications in distributed object systems is described in [Dayal, 1993]. Instead of the cooperative applications considered in [Martin and Pedersen, 1993], the problem considered is that of automating long duration activities in complex enterprises. Such activities typically have weaker atomicity and consistency requirements than ACID transactions . Shared data resources cannot be locked for the entire duration of an activity. Also, complete rollback on failure and restart are unacceptable. Instead, an activity must be thought of as a collection of steps that can commit and release resources before the end of the activity. An activity may be partially rolled back, with committed steps being compensated for, and rolled forward. The *Activity/Transaction Model* of [Dayal et al., 1991; Dayal, 1993] describes the structure and semantics of activities.

4.5 ACTIVE DISTRIBUTED OBJECTS

Active systems are systems that monitor events of interest, and trigger an appropriate response when an event of interest is detected. Recently, active database systems have become a very active area of research and development [Dayal et al., 1988a; Stonebraker et al., 1990a; Widom and Finkelstein, 1990]. Traditional passive database systems only execute queries or transactions that are explicitly submitted by a user or application. Active database systems, by contrast, store and implement *Event-Condition-Action* rules : when the event is detected, a condition is evaluated; if the condition is satisfied, the action is executed. In active, object database systems, the events of interest typically involve the invocation of operations on objects. Some systems also support other types of events, such as temporal events and transaction events, and composite events that are constructed from simpler events using operations such as disjunction or sequence. Conditions are typically predicates over database states; and actions are arbitrary operations against the database or calls to application programs.

In a distributed active system, rule processing is distributed. The component events of a composite event may occur at different nodes; an event detected at one node may trigger conditions or actions at some other nodes; or, the condition and action may themselves involve distributed operations. Some work has been reported on the semantics of, and protocols for, distributed active databases [Ceri and Widom, 1992]. In that paper, a notion of correctness was defined, in which a distributed execution of rules is deemed correct if it is equivalent to a sequential execution at a single node, However, that work was restricted to relational database systems and simple events.

In [Jagadish and Shmueli, 1993], a formal model for specifying and reasoning about composite events in a distributed active object database system is introduced. A number of operations for constructing composite events are described. In this model, the basic events are points in time. An object event history is a sequence of basic events against a single object. Composite events then are predicates defined over object event histories. A trigger is an event (basic or composite) together with a procedure (the action). The notion of *s-correctness* is introduced as the correctness criterion for distributed execution. It specifies

that in a distributed execution, all objects behave as if they were part of a centralized system, i.e., all objects see the same object event histories as in some centralized execution. A number of protocols for ensuring s-correctness are described. These involve centralizing the checking for s-correctness or using the underlying transaction facilities (e.g., object-level locking, optimistic concurrency control), treating events as transactions , to enforce s-correctness.

4.6 OPERATING SYSTEM SUPPORT ISSUES

The problems of building relational DBMSs on top of current operating systems (OSs) has long been recognized within the database community (see, for example, [Gray, 1979; Stonebraker, 1981; Christmann et al., 1987]). The operating system community has also been discussing the difficulty of supporting, with a single set of operating system abstractions, multiple applications with potentially conflicting demands [Kiczales et al., 1992]. The problem is not that OSs provide one set of abstractions, but that they provide a single *implementation* for each of these abstractions. Thus, when a DBMS requires a different implementation for an OS abstraction (such as a file system), the options are limited: either the implementation has to be modified (if there is access to the source code), or the DBMS implementer has to re-implement the abstraction (in some closed OSs, the entire set of abstractions may need to be re-implemented). Clearly, these are not acceptable solutions. The recent movement towards microkernel architectures has the positive effect of reducing the set of abstractions that are implemented within the operating system kernel. The end result is that it is easier to re-implement some of the abstractions that are provided outside the kernel at the user level.

In distributed systems, the situation is further complicated by communication network considerations. Typically, distributed operating systems implement their own communication primitives (for performance reasons), ignoring the standard services offered by the communication protocols. However, a fundamental problem associated with avoiding the standardization efforts is the two-track approach that is being followed. Using the operating system in different networking environments can result in major re-work to accommodate the idiosyncrasies of each individual network architecture. On the other hand, network protocol standardization attempts typically treat the issue in isolation, without sufficient consideration for integration with operating systems. Thus, in distributed environments, the issue is considerably more complicated. We do not intend to go into details of all of these issues; interested readers can refer to above cited literature as well as [Özsu and Valduriez, 1991a]. The above discussion is sufficient to highlight the difficulties.

Object-orientation raises additional issues as well as providing new opportunities to deal with these issues. On the one hand, problems related to providing object management support have to be solved. On the other hand, the uniform use of object-oriented technology in both the DBMS and the OS can overcome some of the difficulties in integrating (or interfacing) them.

One of the major architectural issues is determining where the object management support will be located within the OS. There are three readily identifiable alternatives. The first alternative is to place object management support outside the OS kernel in the user space. The result is an object support layer not unlike the scalable persistent foundation that Atkinson proposes [Atkinson, 1993]. In this case, the OS kernel is not object-oriented and applications that do not require object support can access the kernel directly through ker-

nel calls. Applications (such as OODBMSs, object-oriented programming environments, object-oriented user interfaces, etc) that require a level of object support can be built on top of the object management layer. The second alternative is to put the object management support inside the operating system kernel without the kernel being object-oriented. In this case, the object management services are provided (without exception) to all OS users, but the OS kernel is not developed in an object-oriented fashion. This alternative uniformly provides object management support to all applications and requires them to access kernel services using the particular interface (e.g., messages, invocation, behavior application) the object model supports. A final alternative is to provide object management support at a much lower level and develop the OS kernel using the object-oriented paradigm.

Each of these alternatives have obvious advantages and disadvantages, particularly in terms of their performance implications and the ease with which they can support the development of object-oriented data managers. There are active research projects investigating the design and development of object-oriented operating systems (e.g., [Shapiro, 1993; Cabrera and Jul, 1992]). As a result, some of the issues are fairly well understood; but there are many others that the researchers are uncertain about.

Another fundamental question that needs to be addressed is the nature of object management support that needs to be provided by the object-oriented OS to OODBMSs. [Kulkarni et al, 1993] identifies the following OS requirements: (1) OS has to deal with differing and dynamic resource requirements of applications; (2) the storage hierarchy must be utilized effectively; (3) support should be provided for a variety of cooperation patterns that have different optimizations; (4) cooperation should not be impeded by the heterogeneity of the environment; and (5) the support for interface specification should enable integration of event management. How these requirements can be realized is an open research topic. Dickman argues that the following features of the Bellerophon [Dickman, 1993b] system architecture are well-suited to support an OODBMS: lightweight references , a class-structured object table , and a hybrid garbage collection algorithm. The references in Bellerophon are lightweight resulting in substantial reduction of the space overhead on reference passing while at the same time reducing the CPU overhead of marshalling a reference. It is claimed that this is advantageous for OODBMSs since "constructing additional indices is somewhat eased by having cheap and easily copied references. Furthermore, the transient objects involved in query handling will pass many references to persistent objects as arguments and results; the overhead involved in this is also reduced by the use of small references within the object pools." Bellerophon maintains a local object table at each site (with references to local objects and proxies) which is organized according to classes . Thus, each local object table implicitly contains a "complete set of local class indices" that may be used by the OODBMS in executing queries. Finally, Bellerophon implements a multi-layered hybrid garbage collection algorithm with local and distributed garbage collectors. The efficiency of this hybrid approach as well as the effective garbage collection of transient data is claimed to be advantageous for OODBMSs.

The use of object-orientation in the design and development of both the OS and the OODBMS is expected to assist in properly their "seamless" integration. Even though there is significant promise in this approach, their "object-orientation" may be quite different. For example, operating systems typically implement hardware and software resources as objects; thus objects are *system resources* which are large but have simple consistency specifications. There are few types in the system each usually having a few instances (e.g., there are only a few printers, a limited number of processes, etc). In databases, very large

amounts of simple and complex *data* are the objects; they are persistent and have more sophisticated integrity constraints.

The parallel work on object-oriented operating systems and object-oriented database systems should provide us with a better insight into these issues. On the one hand, we learn more about the object management support requirements of OODBMSs; on the other hand, we are learning what set of support services can be efficiently provided by operating systems.

4.7 MULTI-OBJECTBASES AND INTEROPERABILITY

Object-oriented technology plays a dual role in providing interoperability among autonomous, distributed and possibly heterogeneous data repositories. On the one hand, object-oriented DBMSs are yet another form of data repository adding to the complexity. On the other hand, object-orientation, with its encapsulation and abstraction capabilities, may assist in providing interoperability. It is the latter role that is the topic of discussion in this section.

Semantic heterogeneity among data objects is a long-standing problem in multidatabases . When object-oriented DBMSs are included as repositories in multidatabases, the issue surfaces as heterogeneity among types, requiring the definition of mappings between type definitions. In [Chomicki and Litwin, 1993], the problem is dealt with by providing a declarative language in which derived objects , derived types and derived functions can be defined from "base" types, objects and functions in various object-oriented DBMSs. These derived objects are like views , and are not instantiated.

A very similar problem is addressed in [Bratsberg, 1993] but somewhat differently. Bratsberg proposes to separate the extensional and intensional dimensions of classes and defining separate hierarchies for each. Consequently, a class has semantics similar to views (and in this sense there is some similarity with Chomicki and Litwin's work) and the problem of class integration becomes a subset of the problem of schema evolution (evolution of class hierarchies). Class integration is accomplished by connecting two extent graphs (corresponding to two classes) by proper extent propagations. A second problem is the integration of properties which defines the relationships between the properties of the classes that are to be integrated. The paper discusses various ways in which these relationships can be defined and relates this approach to those based on object-oriented views .

There are a number of ways that object-orientation can play a role in assisting interoperability . A common approach is to define a common object model that can be used to represent various objects from different repositories. In this context, it is useful to recall that this is one of the alternatives that Atkinson proposes to deal with the development of applications over heterogeneous and autonomous components (see Section 4.1). Even though the exact features of a "canonical" model that can serve this purpose are yet to be universally agreed upon, there have been a number of model proposals. There are a number of such models defined in this volume [Manola and Heiler, 1993; Castellanos et al., 1993; Kelter, 1993]. A popular approach is to define an object model that is powerful enough to model others (i.e., is a superset of other models) [Castellanos et al., 1993; Kelter, 1993]. An interesting modeling approach is to define a "RISC" object model [Manola and Heiler, 1993] which establishes a common denominator for the description of other data models and via which

integration and interoperability can be established. The potential components of such a model are defined in the paper. Some of these are explicit primitives for describing basic object model components such as methods , state , OIDs , and interfaces . The model should also permit the definition of various dispatching and delegation mechanisms, should contain type constructors, a generalized view mechanism and the ability to directly manipulate interface specifications (to describe various variants of type systems).

Another alternative discussed in [Tirri et al., 1993] is to use composite objects to integrate data from various data sources. Some of these data sources may be full-function DBMSs while others may not have DBMS functionality (file systems, etc). The data residing at each data source is viewed as a fragment of a composite object and a set of methods is defined to manipulate these fragments as a single entity. These objects are referred to as *fragment objects* (Figure 1 in [Tirri et al., 1993]) and allow loose integration of data in a bottom-up fashion. One fragment per data source is defined and a number of these fragments are integrated into a federated composite object. Two mappings (translation routines) are defined between a data source and the corresponding federated object fragment: `fragment_to_data` and `data_to_fragment` with obvious semantics. Once created, the federated objects can be accessed via methods that can read and/or update parts of the federated object.

The CORBA standard from Object Management Group (OMG) [OMG, 1991] is one of the standardization efforts to deal with interoperability. It establishes a shared conceptual base, similar to Atkinson's scalable persistent foundation, but CORBA has additional functionality such as a common object model and a common model of interaction between object-oriented components. The common object model is *abstract* and includes objects, values (object names and handles), operations, signatures, types and classes . It defines a high-level interface language to this base.

5 CONCLUSIONS

In this paper we provide an overview of distributed object management technology and the issues that need to be addressed in developing it. In addition to providing an introduction to the topic, the paper serves as a road map for the rest of the book providing a context for the rest of the papers in this volume.