Preface

Over a year ago, the Programming Systems Group at Columbia University (also known as ‘‘frodo’’), headed by Prof. Gail Kaiser, decided to upgrade MARVEL, a prototype process-based software development environment, from a single-task single-user system, into a multi-tasking multi-user system. After three years of research in programming environments, the notion of process programming was well understood in the context of a single user, and the group was ready to look into the problems and challenges of synchronizing multiple users while providing mechanisms for cooperation among them. Similar efforts in the general area of design environments are currently being made in the database and software-engineering research communities. Upgrading a single-task system into a multi-tasking system is apparently a hard task that requires both a major redesign and implementation of existing modules, and creation of new modules. In particular, the issue of modeling, managing, and controlling the access to data is crucial and much more complicated in the context of multiple users, especially when existing technologies from traditional databases are unsuitable. This thesis covers over a year of research work in an attempt to provide models, specifications and implementation for data management in such environments.

Two philosophical approaches with respect to system design are taken here. The first one exploits the principle of ‘‘componentization’’. According to this principle, a complex system is built from independent, loosely-connected and replaceable components. As a result, these components must have broad interfaces and must support a variety of different policies employed by components with the same functionality (i.e., components that provide the same services in different ways). This is in contrast to the monolithic approach in which components are tightly-connected and interdependent. In particular, when the components interconnect in a layered fashion, the higher the layer is, the more semantics it has about the domain. In programming environments, the highest layer is the human programmer, the middle layer is the environment itself, with limited knowledge, and the lowest layers that support the environment have minimal semantic knowledge about the domain. The various layers should have great flexibility in implementing their functionality. This is the intuition for the first model presented in chapter 2 and its implementation in MARVEL, presented in 6.

The second approach is to exploit the object-oriented paradigm in distribution, i.e., impose object-oriented decomposition on the participating nodes in a distributed system as a coarser layer of abstraction on top of the data itself. This improves performance and provides an underlying layer of cooperation among users that access shared partition. This is the intuition for the distributed model presented in chapter 3.

I hope you will enjoy reading the thesis.

-Issy Ben-Shaul.
4/30/91

1. Introduction

Design environments impose specific requirements on data management, that differ substantially from traditional business applications. The main differences are in three areas:

Data Modeling - complex, often nested, variable sized, and highly connected structures. (e.g, design of an airplane, or a large software project).

Data Access - the access method paradigm of load/work/save in multi-user design applications (as noted in [41]), is characterized by being (1) long - Thus traditional concurrency control methods might not be suitable; (2) interactive - Thus response time is a key issue, as opposed to general throughput; and (3) graph oriented search - Thus navigational access in addition to, but not instead of, associative access, is required.

Data Sharing - there is a growing demand to enable multiple designers to share and exchange information for collaborative work on large-scale projects. This implies sharing of ongoing work.

In contrast, traditional business applications are characterized by:
Data Modeling - flat, often loosely connected.
Data Access - short, batch oriented, and associative-based operations.
Data Sharing - high level, but always of completed work, and not for cooperation.

Recently, object-oriented data modeling has gained popularity in the area of design environments in general and Software Development Environments (SDEs) in particular. The ability to abstract complex and composite structures as well as simple data items as objects, and access any of them uniformly, simplifies the interface and management of the vast quantities of highly structured data normally associated with developing and maintaining a large design application. SDEs have their own unique characteristics. As noted in [13], these environments often interact in one way or another with the host file-system (e.g, source files, executables, etc.) so some form of interaction with the File Management System (FMS) is often required. In addition, many objects in an SDE that represent such entities vary in size even when they belong to the same class. This is in contrast to other environments (such as CAD) in which most objects of same type usually have more or less same size.

Object Oriented Data Base Management Systems (OODBMS) attempt to address these issues by providing both data abstraction and control over concurrent access to the inherently complex data associated with design applications. Although both the object-oriented paradigm in general, and object-oriented databases in particular, do not have yet a standard formalism, there are some concepts which form the core object-oriented data model:

Object - An object represents any real-world entity uniformly. It has state, behavior, and identity. The state of an object is the set of values for the attributes (*properties*) of the object, the behavior is the set of operations (*methods*) on the state of an object, and each object has a unique identifier. In OODBs, objects are persistent, that is, they retain their state in persistent storage independent of any application.

Classification - A set of objects with the same set of attributes are grouped into a class, and each object belongs to one and only one class as an instance of that class. A class, therefore, represents the structure (*schema*) of all objects that belong to it.

Inheritance - A "kind-of" relationship among classes wherein one class (the subclass) shares the attributes and methods of another class (the superclass) and redefines it by adding attributes, methods or both. When a class is allowed to inherit only from one superclass the relationship is called *single inheritance*, and when it can inherit from multiple superclasses it is called *multiple inheritance*. Single inheritance forms a class hierarchy while multiple inheritance forms a rooted directed graph. The infrastructure of classes is called the *class lattice*.

A multiuser Object Management System (OMS)[1] is defined here to be a system that provides the following:

- Object-oriented data model that provides data abstraction by objects, classification, and inheritance.
- Persistence to the objects and the class lattice.
- Protection mechanisms to prevent unauthorized access to objects.
- Query processing and objectbase navigation. The emphasis is on pointer traversal.
- Transaction management to control shared access to data. This includes concurrency control, transactions and crash recovery.
- Location transparency, when the data is distributed across multiple address spaces.
- Client interface from the application to the OMS.

---

[1]OMSs support the object abstraction but do not necessarily support the full object-oriented data model. Despite this difference, the terms OMS and OODBMS will be used interchangeably throughout the thesis.

The thesis mainly concentrates on the architecture of such systems, with emphasis on SDEs. It proposes both a centralized model and a distributed model that extends the centralized model. Both models address the above mentioned components separately as well as their integration in the overall design of the system. The emphasis is on the interfaces and interrelations among the different components of the system rather than on application of specific policies to certain modules, except for cases where these policies must be assumed for the system to abide with the overall proposed architecture. Although named after the distributed model, both models will be referred to as the Dynamic Hierarchical Object Management System (DHOMS).

1.1. Research Goals

The goals of this research are to design, implement, and evaluate a new model for a multi-user OMS specifically designed for use by SDEs, both in centralized and distributed systems. The primary considerations in the design of this model were:

Cooperation

The system should provide support for cooperation among users, both to encourage parallel development and to enable sharing and exchange of information. The goal is to provide architectural support for cooperation in addition to the support given by the transaction mechanism.

Fast Response Time

When the project is decomposed in an object-oriented manner,[2] a typical design application has a defined subset of the objectbase that is accessed most frequently (referred to as the application's *base*). As the data becomes more distant it is less relevant to the base, and therefore less likely to be accessed (e.g., in programming environments, an application programmer's base is the sub-project to which he/she is assigned). The goal is to optimize the accesses accordingly, that is, to make the data-access time an increasing function of its distance from the base. An application may have several bases, in which case a distance of a datum is taken from the closest base. Notice that while the distance of data items from each other is assumed to represent physical distance either in terms of hops in the network or in arrangement

---

[2]following the definition by Booch in [10].

in memory, the distance of a user application from the base might be only logically short. An application might be physically remote to its base, to accommodate cooperation among physically distant users. Still, the access to its base will be fast as far as the OMS is concerned.

Modularity

We want the system to be modular, and provide as much as possible "pluggable" modules, especially with regard to transaction management policy. Thus, for example, the system should enable implementation of non-serializable transactions such as in [22] and [23], as well as more traditional policies (2PL), without affecting other components of the system.

Support for Long Transactions

Long transactions have specific characteristics that make traditional transaction management unsuitable. In particular, serializability, and atomicity are properties that severely restrict the concurrency of long transactions and may have to be relaxed while still maintaining the integrity and consistency of the objectbase ([19] is an example of such a model). Besides the transaction manager, the goal is to provide support for long transactions in the underlying layers by providing flexible and reconfigurable data management that enables data sharing and convenient implementation of non-atomic and non-serializable transactions.

Distribution

Although distribution is commonplace in today’s design environments, there has been little work done in the area of distributed object-oriented databases ([25], [15]). In addition to the immaturity of OODBMSs, the complexity of the data and its structure (usually highly interconnected, and pointer-based) makes it hard to partition the objectbase and access it efficiently. The goal is to provide a distributed model that will utilize local resources and will provide a fertile ground for implementing cooperation. On the other hand, distribution for high availability and fault tolerance is not a goal of this research.

Dynamic Reconfiguration

Reconfiguration is defined here as the act of rearranging the data management responsibilities among the multiple data managers. The motivation is to be able to dynamically reconfigure the data managers in the system as needed, due to temporary or permanent change of the access patterns, the volume of the data, the number of users, or the cooperative needs. Moreover, it is desirable to provide means for explicit reconfiguration requests, to enable control by the database administrator. The goal is to localize as much as possible the impact of reconfiguration and make it an inexpensive operation.

1.2. Thesis Contents

The thesis consists of two main parts. The first part is theoretical. It presents a model for a centralized single-server and a model for a distributed multi-server system built on top of the single-server model. This presentation covers the specifications for each component module as well as for the system's overall functionality, presenting novel solutions both to the intrinsic problems arising from the characteristics of design applications and to the inherent difficulty of distributing an object-oriented database. A special consideration is given to the specific problems of programming environments.

The second part describes partial implementation of a subset of this model, namely a single server implementation, for MARVEL. MARVEL [7] [21], is an SDE that combines object-oriented data modeling with rule-based process modeling. MARVEL is considered as an instance of the *software process* environments [9], which are characterized as having a programmable process model. Thus, the implementation for MARVEL had to consider its specific characteristics as a rule-based SDE. In particular, the workload division between the client and the server was heavily influenced by MARVEL's rule-based process model. Nevertheless, an attempt was made to both generalize the implementation to any programming environment, and enable implementation of the distributed model as an extension. Therefore, many modules in MARVEL's OMS are ready to be used in a multi-server system. For instance, the object-id scheme conforms to the specifications for multi-server naming, the storage management module in MARVEL is designed so that it can be divided among multiple servers, and the IPC layer can be used for server-to-server communication in addition to the client-server communication in the single-server system. On the other hand, the current implementation does not fully comply with the model with respect to full separation, due to the monolithic nature of the previous single-user system. Thus, this is a first step in this direction. Future versions of MARVEL will exploit full separation.

The initial motivation for this thesis came from the need to upgrade MARVEL from a single-user to a multi-user system. However, upgrading MARVEL to a multi-user system required more than building the OMS. For instance, upgrading rule processing necessitated a major redesign of the rule processor and modifications to MARVEL's query processor; running the graphical user interface in the client process brought up visibility issues not directly related to the OMS. Hence, the second part of the thesis describes the whole process of upgrading MARVEL into a multi-user system, which includes, but is not limited to, building a single-server multi-user OMS.

In the thesis proposal, the plan was to implement the entire model for MARVEL while upgrading it to a multi-user system. However, as explained above, in the process of implementing it for MARVEL it was realized that the problems to be solved and the amount of work to be done required much more resources than anticipated. Thus, only the single-server model was implemented.

Finally, the long term objective of this research is to implement DHOMS as a generic OODBMS for design environments, independent of MARVEL or any specific environment. Chapter 8 addresses this issue. The remainder of this thesis is organized as follows:

Part A - The Model

Chapter 2 presents the centralized, single-server architecture. The key concept here is the *layered* architecture approach. Specifications for the interfaces between the various layers is given. Chapter 3 describes the multi-server distributed architecture. The main issues addressed are distribution, dynamic reconfiguration and support for cooperation. A novel approach to object and server naming is presented, that enables efficient dynamic reconfiguration. The distributed model extends the centralized model, i.e., each server is structured following the single-server specifications.

Part B - Implementation for MARVEL

Chapter 4 gives an overview of MARVEL and the introduction for part B. Chapter 5 presents the problems and the solutions in upgrading MARVEL to a multi-user system. In particular, dividing the workload between the client and the server, and upgrading rule processing are discussed extensively. Chapter 6 describes the implementation of an OMS for MARVEL, following the ideas and specifications from chapter 2. It describes in detail each component of MARVEL's OMS and the interfaces between them. Chapter 7 gives an overview of related work. Chapter 8 summarizes the thesis, and presents the current status of the implementation and future work in this direction.

2. A Single-Server Architecture

This section specifies the structure of a single-server in DHOMS. It addresses some of the problems stated in the introduction, while the distributed model complements it by addressing the remaining issues. Note that only specifications are given here, and no implementation details nor intra-module design details are provided here. In particular, module-specific problems (and the policies taken to solve them) are generally avoided, except in cases where they are crucial to the overall specification of the architecture.

2.1. Overview

One of the problems that many OMSs share is the lack of flexibility with regard to various policies such as concurrency control, data management and modeling, and the interface between the application and the OMS. In particular, transaction management policies are often hard coded in a way that prevents any change of policies without major modifications to other parts of the system. This approach also preempts a stronger property - the ability to tune and alter the system behavior without code modifications at all. In addition, implementing such systems is difficult due to their inherent complexity. The ability to construct and replace different components of the system independently is important both from the implementation and the utilization points of view. It eases the implementation, and enables to conduct cost-effective experimentation of the various components of the system. The latter is especially important for multi-user OMSs since the problem of concurrency-control in such systems is still considered a new research area.

In order to provide these features, a system must be built with an appropriate, modular architecture. However, modular architectures have their own problems. Firstly, it is not simple to define clean interfaces that isolate each component, especially if the various components interact with many other components. This implies that an attempt has to be made to minimize the inter-module dependencies, while providing all the possible interfaces that may be required between any two modules. Secondly, modularity can lead to performance degradation due to the ‘‘narrow’’ connection among the components and the hidden information. For example, in a non-modular OMS, a transaction that wants to access an object can set the locks, locate an object, and fetch it from disk or main memory at the same time. By having all the functionality at hand it can optimize those operations and avoid the overhead of interfacing with other modules. In contrast, a modular OMS would be required to go through multiple modules, and might require more processing. Performance in OMSs is in general a hard problem. One of the most important research issues in the OODBMS community is how to maintain comparable performance to a relational DBMS while providing the expressiveness and abstraction of object-orientation.

DHOMS attempts to address these issues by adopting a special kind of modularity to its architecture, namely the *layered* architecture. (See section 7 for related work in this area.) It defines an interface that all implemented layers must conform to, and restricts the inter-module communication only to modules that are one layer apart from each other. The main emphasis is on the separation between the data management and the transaction management. Thus, the DHOMS architecture enables different transaction policies to be implemented on top of the underlying layers. The transaction management is further separated into a conflict-detection layer and a conflict-resolution layer, and the data management layer is further separated into object management and persistent storage management layers. In order to accommodate full separation between transaction and data management, the data management layer must provide uniform access to objects regardless of their size, type, or lifetime (i.e volatile or non-volatile memory). This is also known as *single-level* store[3].

Single-server DHOMS adopts the basic client/server model, in which a centralized server manages the entire persistent objectbase and users access the objectbase by client processes which may or may not reside in the server's machine. Figure 2-1 illustrates the architecture of DHOMS within an environment. Interface specifications between all the layers of the system are given, including the interface to the environment-support and user-interface modules, although the emphasis is on the lower layers of the systems that control and manage the data. The layers of the system, in a bottom-up fashion, are:

Storage Manager (SM) - The bottom layer that provides persistent storage and controls the flow of data from main to secondary storage. This layer has no knowledge of the data model and therefore can be easily replaced. SM provides services for, and communicates with, the OM.

File Manager (FM) - As noted in the introduction, programming environments need to communicate with the host File Management System (FMS). FM serves as the mediator

---

[3]The term is adopted from [41].

Figure 2-1: Single-Server Architecture between OM and FMS.

Object Manager (OM) - This layer plays a central role in DHOMS, since it implements the object-oriented data model and provides the single-level store feature. The upper layers deal only with the object abstraction. OM communicates with SM to transform the raw data into objects, and provides services to LM. SM and OM together comprise the Data Management part of the system (DM).

Lock Manager (LM) - This layer, usually considered as part of the transaction manager, is treated as a separate layer, to support the complete separation between transactions and object management. LM is the *conflict-detection* layer, and it serves as a mediator between TM and OM and provides the handles for the transaction manager to enforce its policy.

Transaction Manager (TM) - This layer is responsible for controlling concurrent access to the data and maintain its consistency. It does so by enforcing concurrency control, transaction recovery, and crash recovery policies. TM communicates with LM when it needs to access the objectbase, and is called from ES and QP. A cleaner approach might restrict TM to be called only from QP and translate all ES requests into formal queries. However, the lack of specifications and formalism for a query language in OODB suggests to allow a direct interface between ES and the objectbase.

SM, OM, LM, and TM together can be viewed as the subsystem that provides and controls the access to the data and are the main subject of the thesis.

Query Processor (QP) - This layer translates the queries expressed in a query language into calls to the TM. The queries are posed either directly by the client application or indirectly by ES.

Environmental Support (ES) and (CES) - This is the environment's engine that is built on top of DHOMS. ES and CES - its counterpart component at the client - comprise the environment. ES communicates with TM directly or via QP and provides services to the client application, either directly or via CES.

Inter Process Communication (IPC) - This layer provides the communication between the client application and the server, and is responsible for receiving and sending messages. Like OM, IPC must preserve the object abstraction when transferring objects between the client and the server via a sequential medium. Therefore, much of the techniques resemble the way OM transforms connected objects into linear streams of bytes.

Client Server Interface (CSI) - This layer translates requests both from the users, via UI, and from CES, into requests to DHOMS, via IPC.

User Interface (UI) - This is the uppermost layer of DHOMS. It interfaces between the human user and the environment. This might suggest that UI may not be included as part of DHOMS at all. However, in order to provide the users with an image of the objectbase, this layer must communicate with DHOMS in order to get and update its image. Therefore a standard UI layer is specified, and it can be augmented by any environment-specific requirements. UI communicates with the user to accept application programs or ad hoc queries, and passes them to CSI.

The following sections describe each layer in detail. A detailed description of the interface function for each layer is given in appendix A.

2.2. Storage Manager (SM)

As mentioned above, SM is responsible for providing persistence to the objectbase, and is independent from the data model, therefore it operates as a byte store. SM consists of two subcomponents: An in-core buffer manager and a disk manager. The disk manager is responsible for storing and fetching the requested data from the disk to memory, and is controlled by the in-core manager that maintains a pool of cached information to speed up disk I/O (much like in Unix buffered I/O). Note, however, that SM's cache does not represent the in-memory objectbase, which is maintained by OM. Thus, SM must provide the option to pin objects immediately to disk. The interface to OM is provided by the basic operations open, read, write, and close.

The considerations in implementing this subsystem include:

- The system may include a large number of objects of varying size. Thus, SM must be able to store/fetch data items with optimal efficiency, regardless of their size.
- SM may provide a clustering mechanism, i.e. - store related objects in contiguous areas on disk to enable fast read/write of objects. There are various clustering techniques; some let the upper layers specify how to arrange data on disks, while others are transparent to the upper layers and done automatically based on predefined algorithms. Clustering is a separate research topic that is not addressed in this thesis.
- Optimizing the cache size as a function of the various system characteristics.

2.3. File Manager (FM)

FM is called by OM whenever a file object has to be read/written/added/removed. It is basically a set of functions that interact with FMS via system calls (See appendix A).

2.4. Object Manager (OM)

As mentioned above, OM provides the object-oriented data modeling, and single-level store. When OM wants to read an object, it translates the object-id into an address. In order to enable upper layers to participate in determining the clustering of objects, the translation is made at the OM level rather than at the SM level. The main characteristics of OM are:

The Data Model

The DHOMS data model is an extension of the core object-oriented data model (presented in the introduction). It is defined by a set of classes, each of which consists of a set of typed attributes, subclass-superclass relationships and multiple inheritance. An object has an identity, state, and behavior. In addition to the core model, the data model defines *composite objects*, which resemble those in ORION [27].

Objects are interconnected via two mechanisms:
Set-Of attributes - specify the class of objects that can be connected as subcomponents, to abstract *containment* relationships among objects. Thus, a composite object may have other objects as subcomponents imposing hierarchical structure on the objectbase.[4]
Link attributes - provide arbitrary semantic connections between objects as defined in the schema (horizontal connection). Links are useful when non-hierarchical pointer traversal is needed. However, they add complexity to the objectbase since they break the hierarchy, hence they should be used carefully. DHOMS distinguishes between these two types of attributes. In general, links are weaker than set-ofs and are considered as an augmentation to the hierarchical structure of the objectbase.

The operations on the objectbase are not part of the class definition as methods are in conventional object-oriented systems in order to provide flexibility with respect to the process model as defined by the environment. In addition, this separates the query language from the

[4]Kim [26] denotes this relationship as the object-composition hierarchy.

data management, a necessary property in OODBs.

Another property of the data model is that the principle of encapsulation is not fully kept, as attributes are exposed to the application, mainly to provide high visibility to end-users. For instance, in a programming environment, a programmer would like to know what the object he or she designs (programs) looks like internally, and what are the current values of the object's attributes, which may have been changed by other users. However, the exposure is read-only, and any change to a state of an object occurs only via methods.

Object Faulting

OM maintains an in-memory objectbase.[5] When OM receives a request to read an object, it first checks whether the object is resident in its pool. If the object is not resident an object fault occurs and OM has to be able to fetch it from SM. As noted in [20], the main two techniques to handle object faulting are *node marking* and *edge marking*. In node marking a non-resident object is marked with a stub (fault block) that encodes its location in secondary memory[6], while in edge marking, the links are tagged as pointing to non-resident objects. Each technique has its own advantages and disadvantages. Node marking requires additional space to hold the stub and additional indirection through the stub, but once a fault is detected all connected objects can access the object. In contrast, in edge marking each link has to be updated separately, but no additional in-memory space is required. DHOMS' approach to faulting is a variation of node faulting but without the space overhead. Using DHOMS' object naming scheme (see section 3.5), only a subset of the non-resident objects have to be represented by stubs. When the ancestor O2 of a non-resident object O1 is stubbed, the system infers that O1 is not resident and fetches from secondary storage the object along with the non-resident ancestors. Thus, an object cannot be resident without its parent being resident. The advantage of keeping a small subset of stubs in proportion to the number of non-resident objects outweighs the overhead of locating the resident ancestor, especially when considering the efficient naming scheme that makes this process inexpensive. (The actual way of locating an ancestor is similar to the way a local server is found as described in section 3.5.)

[5] the control over the size of the in-memory objectbase is not specified here.

[6] In DHOMS, SM may also retain the object in its own cache but this is transparent to OM.

Class Lattice

OM maintains the class lattice always resident in memory. It is read by the server on initialization and is not changed throughout the lifetime of the server. (Schema evolution is not addressed here.) Each class may specify default values for the various attributes, which are assigned on object creation.

Read/Write Operations

The read operation on a non-resident object involves typing the byte stream based on the object's class. Thus, the first few bytes of an object always encode its class, and the object can be interpreted regardless of the information known to the caller (e.g., during objectbase initialization, the server reads objects without knowing their classes in advance). The write operation (onto disk) is the opposite: the values of the object's attributes are written and the owner class is encoded into the beginning of the byte stream. OM does not prescribe a specific policy for the granularity of the read/write operation, thus, it provides the upper layers with read/write operations down to a granularity of a single object. However, a specific transaction manager could specify only operations on composite or sets of objects, in which case the granularity is always coarser than a single object. In the case of objects that contain file attributes, OM must distinguish between a request to read/write the contents of the file (via a call to FMS) and a request to read/write other attributes (via SM). This distinction is important since the difference in the amount of data involved might potentially be very large.

2.5. Lock Manager (LM)

Usually considered as a subcomponent of the Transaction Manager (TM), LM is treated as a separate module, to support the modular approach. The separation made here is between *conflict detection*, handled at the LM level, and *conflict resolution*, handled at the TM level. Conflict detection is the layer that is responsible for detecting multiple accesses to an object which violate the concurrency control policy as defined by TM, while conflict resolution must provide a solution to the apparent conflict, also, according to the concurrency-control policy. When a transaction T1 requests a lock L1 on object O1, LM fetches the current lock information associated with O1 and checks to see whether L1 can be set, i.e. - whether it is compatible with the current locks on O1. Detected conflicts are reported to TM along with the conflict information that includes other transactions holding locks, types of locks, and objects that caused the conflict. Otherwise, T1's request is granted, the lock information is updated, and the object can be fetched/stored. Conceptually, conflict detection is associated with the data, while conflict resolution is associated with the users requesting access to the data. This makes the separation between these two layers both natural and important for modularity considerations. LM and TM can be modified with minimal effect on each other. Changing the resolution method does not affect LM, while changing the detection mechanism hardly affects TM. In addition, this separation provides support for distribution, since LM is local to the data while TM is local to the request for that data, which can be remote (See section 3.6 for more discussion).

The Lock Set

An interesting feature of LM is its capability to support *any* set of locks on behalf of TM. This is contrast to other lock managers in which the lock set is both fixed and hard coded. The idea is to have the compatibility information external to LM. When the concurrency control policy has been determined, the compatibility information is specified and is fed to LM. Thus, LM operates according to external specifications, without actually having semantic knowledge about the type of the locks. Although no specific lock policy is provided here, the composition hierarchy in the objectbase gives rise to a policy that provides fine granularity with respect to locking, to enhance concurrency. This method, known as *granularity locking*, was implemented in MARVEL and will be discussed in section 6. An additional feature of LM is its ability to support multiple locks on an object that violate the compatibility information. This is necessary to enable non-conventional policies to be implemented. For instance, a cooperative transaction might allow two transactions that relate to each other in a special way to share uncommitted information. This gives architectural support for, and enables implementation of, *cooperation*. The support is provided by the LM_enforce_lock function, which implies that a lock on an object must be set regardless if the detection policy known to the LM. (see appendix I.9 for more details).

Finally, note that while LM seems to support only locking-based concurrency-control policies, other methods can be applied as well, as long as LM supports the right primitives. For example, an optimistic concurrency-control policy can be simulated by setting locks but ignoring conflicts (either at the LM, by defining an ‘‘empty’’ compatibility matrix, or at TM, by enforcing locks that conflict according to LM) until the validation phase.

2.6. Transaction Manager (TM)

Like other transaction managers, it is responsible for controlling concurrent access to the data and maintaining its consistency. As defined in [12], it must provide concurrency control, transaction recovery to maintain consistency in case of internal exceptions due to user interruption or conflict resolution, and crash recovery to maintain consistency in case of external system/network breakdown. This section focuses on concurrency control. Recovery is discussed extensively in section 6.

Concurrency-Control

One of the main goals of DHOMS is to provide flexibility with regard to TM's concurrency control policy. Thus, no specific policy is prescribed here but only interface specifications via transaction primitives are given. (Chapter 6 gives a detailed design and implementation of the transaction manager for MARVEL.) The idea is to provide primitives that enable flexible implementation. It is important to note that ''transactions'' in this context may not conform to their traditional definition. In particular, they need not be serializable or atomic. For example, when a long transaction is aborted, it may not be desirable to rollback all the changes made, hence partial UNDO may be allowed (non-atomicity). Cooperative transactions may be allowed to see intermediate results (non-serializability) while still preserving consistency. In addition to the traditional primitives BEGIN, READ, WRITE, COMMIT, and ABORT, DHOMS specifies additional primitives such as ADD, DELETE, STRONG_READ and STRONG_WRITE. These may imply a different set of locks to be requested from LM. One important primitive that is worth special consideration is the SHARE primitive. This primitive allows one transaction to share a copy of uncommitted data held by other transaction. It can be used to implement non-traditional transaction models such as transaction groups [37] and split-join transactions [32], by assigning multiple transactions a shared copy of objects. Groups could also be implemented at the LM level, by associating an optional group-id with lock requests. However, this would imply specific policy and violate the conceptual separation between data and transaction management. The full set of TM primitives is given in the appendix.

As mentioned earlier, SM, FM, OM, and TM form the core of the single-server model. The following layers are mentioned here to give a complete specification of the OMS, but are not discussed extensively. Part B gives a detailed implementation of those layers.

2.7. Query Processing (QP)

It is argued ([34]) that Object Oriented DataBases (OODB) usually do not support an independent query language, and the data is accessed directly, through low-level interaction with the object manager by means of *navigation*. In particular, no associative access is provided. Therefore, data independence, which is an inherent characteristic of the relational model, is harder to achieve in OODBs. However, DHOMS does specify a query language interface, both for associative and navigational access, where associative access is done against the class lattice and navigational access against the objectbase-hierarchy. Associative access is done using any combination of the logic operators EXISTS, FORALL, AND, and OR. Navigational access is done using MEMBER, ANCESTOR, LINK-TO, and LINK-FROM primitives. Queries may be formed by any combination of the above operators, together with a SUCHTHAT (expression) where expression may be constructed using equality/inequality operators (see figure 6-6 for an example of a query in MARVEL, and appendix I.7 for complete specifications). QP provides both interactive and batch query interfaces. In addition, it provides a set of functions (built-ins) to access the objectbase directly. This enables a richer interface to the ES module and does not restrict it to the query language.

2.8. Environmental Support (ES)

In essence, DHOMS is built to serve ES. Even though it provides direct access to the objectbase via QP, it is intended that most of the accesses will come through ES. In process-oriented environments, ES defines a process model, by which it communicates with DHOMS and with other entities such as external tools, for tool integration. Since ES is external to DHOMS, a sufficient interface has to be provided so that it can access the objectbase in a variety of ways. Built-ins is an example. A key issue is the division of ES between the server and the client when constructing an environment on top of DHOMS, and a thorough discussion is given in section 5.

2.9. Inter Process Communication (IPC)

DHOMS adopts the paradigm of message-passing to enable clients and servers to reside on remote hosts. In the single-server architecture the traditional client-server model is adopted, whereby the server is a passive entity that responds to requests coming from active clients. The IPC layer is composed of two sublayers:

- Low-level protocol for sending and receiving packets between processes.
- Object-level functions for transferring objects across address spaces.

As mentioned earlier, there is an analogy between IPC and DM (Data Manager), since both have to deal with linearization and reconstruction of the objectbase. Essentially, IPC adds another level of abstraction to the already multi-layered network communication protocol, namely the object-oriented abstraction. Some of the problems with IPC for an object-oriented model are:
(1) Linearization - As in any other pointer-intensive model, the problem is how to translate efficiently the pointers in one end so they can be recovered on the other end. Moreover, when an incomplete objectbase is transferred (the usual case), object faulting occurs.

(2) Variable sized transfers - due to the variable sized objects, the protocol has to be flexible and translate reasonably well to small as well as very large objects.

Part B presents the implementation of a protocol that attempts to address these problems. The main functions that IPC provides are send_object and recv_object. The notion of composite-objects implies that it must distinguish between sending an object with or without its descendants. However, links (see 2.4) are not followed and they have to be recovered at the receiving end, which introduces additional unavoidable overhead.

IPC Low-Level Functions

This layer is in charge of providing reliable transfer of the data. In this layer, the objects are fragmented into fixed sized packets, which are sent through the underlying reliable transmission protocol, either TCP sockets on the Internet domain, or Unix sockets on the Unix domain.

2.10. User Interface (UI)

This layer resides in the client application's address space, and is the uppermost layer of DHOMS. It is an application-independent layer that provides browsing capabilities and an interface to the server's Built-Ins (BI) module. An important characteristic of DHOMS is that it provides a visual image of the objectbase to the client. However, this image does not, by any means, represent a copy of the objectbase. The sole purpose of the image is to enable access and reference to arbitrary objects in the objectbase and provide visibility to the end-users. Thus, the client's image needs to contain just the structural information required to refer to an object in the objectbase, i.e. - its name and its id as well as the interconnections among the objects. Note that users should specify object and class *names* when accessing the objectbase image, and the CSI layer translates it into an object id and class id, respectively, based on resident mapping tables.

UI interfaces to CES via menus that are available for use by the environment and are passed to ES, via CSI (see figure 2-1). It communicates with CSI to pass all the information as entered by the user.

2.11. Client-Server Interface (CSI)

This layer provides low-level support to the client's application. It communicates with the application via UI and handles the requests to the server by interacting with the IPC layer. CSI provides the following: *Name Servers* - CSI maintains both class and object name servers that map objects and class names into their ids. The class table is static, but the object table may change when objects are added/deleted/renamed/moved. Thus the object name server is updated whenever the objectbase is updated in the client's space. Note that the server identifies objects by their id, while their name is merely a (built-in) attribute of that object.

Consistent View - DHOMS must provide the client application with a consistent view of the objectbase. The main problem here is to keep the image updated when the state of the objectbase is changed by other users. Any implementation must consider this problem with care.

Query preprocessing - While CES handles all environment-related preprocessing, there must be some preprocessing mechanism for direct interaction to DHOMS via Built-ins or direct queries. This may save an enormous amount of work at the server. Preprocessing may include checking that the command/query was formed correctly, and processing some of the services that do not require interaction with the objectbase, such as help and usage commands.

2.12. Summary

This section presented the layered approach to the architecture of an OMS for programming environments, concentrating on definition of the various layers as independent, replaceable components, and on the interfaces that enable such an architecture. The main characteristics are:

- Separation between data management and transaction management.
- Separation between conflict-detection and conflict-resolution, enabling implementation of multiple concurrency-control policies on top of the detection layer.
- Separation between object management and persistent storage management.
- Broad interface between layers to provide maximum flexibility in implementing upper layers on top of lower layers. (For example, the lock set)
- Encapsulation of files within objects, and the interaction between object

management and file-system management to handle file-attributes.

3. A Multi-Server Architecture

The single-server architecture presented in the previous chapter can be regarded as a restricted distributed system where the clients have their own processing power and can carry out computing tasks in their own address spaces. However, the entire objectbase is managed by a single, centralized server. In a fully distributed OODBMS the data management task is distributed among multiple servers in different sites. This chapter proposes a distributed model for OMSs by presenting a high-level design for the architecture and focusing on the method for data partitioning.

3.1. Background

Distributed databases can be roughly divided into data-sharing systems (DS) and data-partition systems (DP)[7]. In DS, the entire database is physically accessible from all participating nodes, while in data-partition the database is physically partitioned into disjoint partitions, each of which is managed by a single node. DS systems avoid the problem of static partitioning, and provide high reachability and higher availability since in the case of node failure the remaining n-1 nodes can still access the entire database. In DP, high availability can be achieved by replicating partitions. DS has the major drawback that it requires special hardware architecture (i.e. - multi-processor architecture), while data-partition systems can be built on top of any set of nodes connected by the network, a configuration that is commonplace in today's computing systems. To provide a DS-like model on top of a physically DP model, some protocols were developed (most notably Sun's Network File System (NFS) protocol), in which the entire database is logically shared, and remote data is accessed transparently through a low-level communication protocol. This does not solve the availability problem, but simulates a DS system and provides high reachability, a high level of distributed computing, and dynamic partitioning. This model will be referred to here as Virtual Data Sharing (VDS). Finally, there is another model that is a special case of DP. In this model, there are large number of nodes with small storage capacity and a small number of nodes with large storage capacity[8]. This does not imply necessarily two levels only. It could be devised as a hierarchy wherein top-level nodes are

---

[7]Another hybrid approach, taken by Tandem, maintains backup nodes with replicated database, for high availability.

[8]This configuration, in which many workstations operate with small local disks and access a file server is very popular and appears to be highly cost-effective.

larger and low-level nodes are smaller. It will be referred to as Hierarchical DPs or HDP. The HDP model implies that although the data is partitioned, it can still perform dynamic reconfiguration with a reasonable amount of overhead, to physically move small amounts of data between nodes.

DS and DP with replication present similar problems with regards to coherency control, since multiple nodes might have different versions of a data item either in a memory buffer (DS) or in disk (DP). This problem can be solved in two ways: (1) Centralized approach (e.g., Central Lock Manager (CLM) protocol), in which one node maintains global table; (2) Distributed approach (e.g., Primary Copy Locking (PCL) protocol), in which the database is logically partitioned into N disjoint parts, each of which is managed by a single node[9]. The centralized approach, however, preempts most of the distributed properties and is usually undesirable.

The discussion above leads to the conclusion that in order to exploit distribution, some form of data partitioning (i.e. - logical or physical) has to take place, regardless of the distribution model itself. The main work presented in this section is on *how* to partition the data. The model is mostly intended for DS and VDS systems (where the partition is logical), although it can be applied in full to HDP systems, and in part to any DP system (in which case the partition is physical). Thus, unless otherwise noted, DS/VDS models are assumed.

The main motivation for data partitioning in here is twofold:

1. Utilize local resources. This is common to all distributed systems.
2. Provide a platform for cooperation. This will be clarified later.

3.2. Overview

While distributed relational databases have reached maturity (e.g., R* [40] and INGRES* [39], the distributed extensions of R [4] and INGRES [38] respectively), the research on distributed OODBs is still in its infancy. (Examples of existing prototypes include distributed versions of ORION [25] and AVANCE [1].) As mentioned in section 1, the main problem with distributing an OODB is with the high inter-connection among many small objects. In the context of data partitioning, the problem is how to partition the data effectively and still maintain high degrees of distribution, location-transparency and high visibility. Moreover, assuming DS or VDS models, the problem is how to provide a high level of dynamic partitioning with minimal cost.

---

[9]see [33] for a discussion on coherency protocols.

DHOMS attempts to address these issues by presenting a partitioning scheme that corresponds to a specific kind of objectbase, namely - a hierarchical objectbase[10].

In essence, the idea is to partition the data along the objectbase hierarchy, that is, to allow partitioning of the objectbase on any non-partitioned subtree of the hierarchy, thereby creating a hierarchy of nodes that maintains the logical global structure of the objectbase and eases the task of partitioning. Additional ideas and advantages of this scheme:

High level of distribution - In order to exploit distribution, the idea is to enable each node to have sole control over its partition, without external intervention. By having an arbitrarily deep hierarchy of servers (rather than a fixed two-level hierarchy), a given node can further partition itself and delegate authority to other (newly created) nodes, i.e., to *manage the distribution* of its partition. The careful reader may suspect that this may imply very high cost in accessing remote objects. However, the idea here is to exploit object-oriented decomposition, and assume that the way the data is organized reflects the organization of the application. Therefore, close data is accessed frequently and as it gets more distant (in the hierarchy), it is less likely to be accessed. With this assumption in mind, it makes sense to make the data-access time to be an increasing function of the distance.

Scale - As a consequence of this partitioning, each node is required to know only how to connect to its parent and children nodes. (Although in practice it may be desired to know how to connect to more nodes.) In addition, when the data grows to a certain level such that further partitioning is required, a node can simply partition itself[11].

Cooperation - There are compelling reasons to believe that partitioning could be used not only for performance enhancements but also for cooperation purposes. With the object-oriented decomposition in mind (see above), all clients that are connected to a specific node are assumed to be associated with the specific partition managed by it (e.g., they all work on the same sub-project). In this case, the proposed infrastructure of nodes can serve as a convenient underlying platform for a cooperative model in which all "local" users can be treated in special ways that differ from accesses to the partition by ‘‘remote’’ clients. An example of such a concurrency

[10]Additional horizontal hooks are allowed, but just as augmentation to the hierarchy.

[11]In practice, control over partitioning should be imposed. see section 3.4.

control policy is transaction groups [37]. Note that partitioning for cooperation might have a negative effect on performance and vice-versa.

The main question is how to keep the cost reasonable with such high levels of distribution and dynamicity. The major problems are:

Naming Scheme - both for objects and nodes. The problem is that in the case of partitioning, all objects must still be accessible from any other partition. Local partitioning makes the problem harder, since it might be the case that not all nodes are aware of recent partitioning/joining. A novel solution to this problem is presented in section 3.5.

Visibility - This is particularly important in interactive environments such as SDEs. The problem here is to maintain a high level of visibility, i.e. - to enable a user to see any part of the objectbase, including remote partitions.

Other hard problems not addressed in this section are distributed transactions, distributed queries, and fault-tolerance.

A detailed description of the model is given below.

3.3. The Dynamic Hierarchy model

The model consists of three entities: A unique Primary Server (PS), an arbitrary number of Local Servers (LS), and clients (CL) that represent user sessions (see figure 3-1).

PS along with the LSs form a hierarchy of servers, each of which manages a disjoint partition of the objectbase whereby each server's partition is a subpartition of its parent server. The clients are the leaf objects, usually connected to a single LS although they may occasionally connect to multiple servers.

3.3.1. Primary Server (PS)

The primary server is the root server in the system, i.e, it is assumed to always exist while the system is running[12]. Besides having the functionality of a local server (see below), it has the additional functionalities :

1. An initial front-end to the system.

[12]This is a logical view. Obviously, any implementation has to take measures to recover from a PS's failure.

Figure 3-1: Multi-Server Architecture

2. A source of global information.
3. Global system initialization.

This is done by maintaining an updated persistent Global Distribution Table (GDT) that maps all existing partitions to servers. It is the only server that is known (i.e., it's address is known) to all the servers in the system regardless of their depth. Note that the motivation is to make PS as similar to LS as possible, to exploit distribution and autonomous management of LS's over their data. However, a conflicting motivation is to maintain *location transparency*, and therefore, a client application starts up a session with PS and specifies it's *base*. The PS directs the client application to the most specific LS, and from then on the client communicates (transparently) with the assigned LS. Another conflicting motivation is *recovery*, hence the compromise here is to enforce each LS to be registered at the GDT, so that its partition can be recovered in case of a (sub)system crash. Finally, PS is the only server that is required to be active in order for the system to be operational. Note that the configuration of the servers must be persistent and recorded at the PS.

3.4. Local Server (LS)

Local Servers perform the main tasks of data management. Each LS has a unique parent server, and an arbitrary number of children LSs, which can be viewed as subordinates of the LS. Assuming no failures, an LS has complete and exclusive authority over a partition of the objectbase. Any request for a data item within a partition is handled via it's LS. An LS can also delegate authority over a subpartition, in which case it loses its authority over that subpartition. In the same manner, it gains authority over a subpartition when one of its child LSs is destroyed. The important characteristics of LS are:

Dynamic Reconfiguration

As noted above, one of the goals is to provide dynamic reconfiguration in mapping servers to partitions (see section 1 for a definition of this term.) An important unique characteristic of DHOMS is that an LS is a dynamic entity, i.e., it can be dynamically created and destroyed on the fly. this operation can be invoked automatically from the OMS, or manually. In order to maintain location transparency, manual partitioning is allowed only to a system administrator (superuser), who controls the hierarchy of servers. Thus, as far as a client application is concerned, there is one objectbase, served by a single server. Internally, from a system point of view, the objectbase consists of a hierarchy of many small servers, each of which manages a fraction of the objectbase, where immediate LS's serve as gateways to the objectbase. The goal is then to devise the optimal hierarchy (in terms of performance, cooperation or both). Note that finding the optimal distribution in terms of performance is a known hard problem since it depends on many variables such as workload, resources and communication overhead[13]. This approach provides means to dynamically tune and control the distribution. However, dynamic configuration should cause minimal global impact on the rest of the servers and their users. This is addressed in section 3.5. The configuration of the servers also must be remembered so any LS has an independent existence once created.

Multiple LSs

Another unique characteristic of DHOMS is that a client can have more than one LS, in cases where frequent access is needed to multiple partitions. (e.g., an application programmer working on project A frequently accesses a utility library L that is shared with other projects). Thus, while the LS's form a strict hierarchy, the entire OMS is modeled as a DAG. This adds complexity to the client-server interface, since it has to distinguish between references to multiple LS's, and to identify the closest LS in case of requests for objects in remote LS's. Nevertheless, it is an important feature both in efficiency and cooperation aspects.

LS Functionality

The responsibilities of an LS over its partition are:

Persistence of the partition.

Consistency of the partition. This is gained by LS's Transaction Manager (see section 3.6).

Fetching remote data - By maintaining a Local Distribution Table (LDT), LS determines the path to follow in order to locate a requested object's server and to request from that LS to perform the fetch operation on that object. (See section 3.5 for an explanation).

Handling remote requests - When a request from another server arrives, LS determines whether the request applies to its own partition or whether further redirection is needed. For local data,

---

[13]An intelligent system could possibly have a controller daemon that automatically configures the system based on a specified set of system parameters, but this is beyond the scope of this thesis.

the LS performs the request with (possible) special considerations as defined by the transaction manager's policy towards external requests. One restriction might be the priority with which external requests are handled, as opposed to local requests.

Supervising children servers - This may include: Waking up an inactive child server, either when an external request arrives, or when a new client wants to access it directly. Another responsibility is to forward external requests to one of the children servers, recursively.

Partitioning the Objectbase

When the need arises to partition the objectbase, a superuser issues the command to the parent LS who owns that partition, specifying:

- The subpartition to be assigned a new LS.
- The identification of the target host.
- The kind of partition.

The sub-partition is specified by the top object of that partition. The target host specifies where this child LS will reside. Note that there is no restriction as to where to create the server. It can be created on the same machine - mainly for cooperation purposes (e.g., to create a group server), and it can be created on a different machine (e.g., at a client's machine) to utilize local and/or additional computing resources. Recall that the VDS/DS model is assumed here, thus partitioning does not mean physical partitioning. In a DP model, creation of a server on other machine requires physical transfer of the partition. In a general DP model this is impractical, since it would require an enormous amount of time and space if the partition is arbitrarily large. In this case the partition would have to be done statically. However, dynamic partitioning can still be applied in a HDP model to transfer small partitions from a large server into a small, local server.

Appendix II.2 presents a detailed description of the partitioning and rejoining procedures.

3.5. Naming Scheme

A crucial issue in the dynamic distributed hierarchy is the naming scheme for objects, since any local server must be able to locate any object in the objectbase given its id, thus, an object id must encode its owner server. Moreover, it is desired to encode the hierarchy of servers as well as the hierarchy of the objectbase in the id so that an object could be located remotely even when its most specific server is not known to the LS due to a later (dynamic) configuration of the servers or due to network overhead considerations. Hence, the only necessary requirement for an LS is to maintain its parent server and immediate children server's addresses. As will be seen, an optimization is made to speed up remote access, but the important point is that if an LS does not have the up-to-date configuration at hand, it can still navigate through the hierarchy of servers to locate the desired LS, and update its configuration. The final constraint on object ids is that they must not change due to a reconfiguration. An update of all object ids in a partition due to adding/removing a server would cause tremendous overhead, since all clients in all servers would have to be notified, and the objects would have to be modified in secondary storage.

Facing the above constraints, the key idea in the naming of the objects is to make them unique but provide for multiple interpretations of the same id in an orderly manner, i.e, the different interpretations are tried in a determined order, until an interpretation that the server ''understands'' is found.

An objectbase has a *breadth* number (BN) that determines the maximum number of children a given object may have, and is basically the base of the representation. When adding an object O2 as a child of O1, O2's id is computed by:

$$id(O2) = id(O1) \times BN + y + 1$$

where y is the number of existing children of O1. Note that this enables a unique decomposition based on the remainders from division by BN (equivalent to representing the id in base BN). For instance, if $BN = 16$ and $id(O1) = 50$ then the reminder decomposition of it is (3,2). Similarly, the decomposition of 302 is (1,2,14). Figure 3-2 illustrates the naming scheme in an objectbase with $BN = 16$. The vectors in brackets are the decompositions of the ids, and are used to locate the object's server as will be explained shortly. Note that this scheme imposes limits both on the depth and on the breadth of the forest. BN together with the length of the representation of an id (idlen) determines the maximum depth of the forest (DF) by:

$$DF = \log_{BN} 2^{idlen}.$$

Nevertheless, the enormous advantage gained here is that the object id encodes its location and the server naming scheme is built as an interpretation of the object id, thus changing the

Figure 3-2: Object Naming Scheme

configuration does not change the object id but only its interpretation.

Object and Server Location

Each LS maintains a Local Distribution Table (LDT) that consists of pairs of server ids and their addresses (PS maintains a Global Distribution Table (GDT), which is basically identical to an LDT except for additional global information). The server id is the remainder decomposition of the top object of the server's partition. As mentioned above, this table must keep entries at least for the parent and all children servers, although it is desirable to have more entries. Initially it gets an entry for each server in the system. However, since the configuration is likely to change, the server can still utilize an out-dated table and a new configuration does not require any overhead on the rest of the system as promised. When a request to access an object arrives at LS1, LS1 first checks whether the object is local. If it is remote, an attempt is made to locate the most specific server known to LS1, based on the information in the LDT. In the best case, the real LS of that object (LS2) is in the LDT, so the request is forwarded directly. Otherwise, an LS that is a few hops from LS2 is known (say LS3), and LS3 forwards the request based on its LDT. At the end of this transaction, LS1's LDT is updated to include LS2.

The method for finding the most specific server for object O1 with id OID is as follows: An object id is parsed using the remainder decomposition. The first interpretation is to look up a server with the full decomposition at the LDT. If that fails, a server id represented by (OID shifted right by BN) is looked up, etc. In the worst case, the Primary Server is called with the request. For example (see figure 3-3), if id(O1) = 802, its decomposition is (3,2,2), so the LS will interpret the server ids in the following order and will stop once a known server id is encountered in the LDT:

1. (3,2,2)
2. (3,2)
3. (3)
4. (0) - PS's id.

In the example in figure 3-3, the lookup will stop in the second interpretation (i.e., at 3,2) if LS has this server's id in its LDT. In this case the request will be directed to that server. If (3,2) is not in the LDT or (3,2) did not respond for some reason (it does not exist anymore, or is sleeping), the lookup will stop at 0 (i.e., PS) since there is no server named (3). PS in its turn will have an entry for LS (3,2) and the request will be forwarded to (3,2), while updating LS's

LDT to include (3,2) for future references. Again, we assume here that every server has an entry for PS in its LDT, and since PS is a static entity, it is guaranteed to always be there. As promised, this scheme enables to route through the objectbase when minimal distribution information is available as well as to speed up remote accesses as the information in the LDT is more up-to-date.

3.6. Distributed Transaction Management (DTM)

While no specific transaction model is described here, general requirements and guidelines are given in which all DTMs implemented on DHOMS must comply with:

- Each LS has an attached Transaction Manager (TM) that handles transactions originated locally by clients of that LS. These transactions might access remote data.
- All TMs must follow the same transaction model, although they may possibly have different options enacted.
- Lock Managers maintain only local lock information.
- Remote queries involve interaction between a TM and remote LMs. Recall that the conflict detection layer resides at LM, thus conflicts are detected without TM's intervention. Conflicts among transactions originating at different sites can be resolved without TM-TM interaction, assuming that every TM maintains the same policy. However, TM-TM interaction may be optionally desired in case of special TM policies such as in [8].
- Every DTM must implement an equivalent of the two-phase-commit protocol for the commitment of transactions.
- The model of execution can follow either Remote Procedure Call (RPC) i.e. - a request is executed at the remote server by spawning a local agent at the remote host to execute the request - or by the Copy-and-Execute approach, in which the remote data is fetched to the server for local execution. No requirement or preference is given here.

3.7. Visibility

This is an important issue from the end-user perspective. The goal is to enable an end-user to display objects that contain or link to objects in other partitions both directly by the user interface and indirectly, through queries.

The main problem is to propagate the updates of remote partitions into the client's space. Since locality of reference is assumed, there is no need to propagate changes from a remote partition unless that partition was previously referenced. The idea is then to keep in LS's LDT (see section 3.5) a timestamp for each remote LS (LS1). This timestamp represents the last time LS1 accessed LS. When LS1 requests data from LS, its timestamp (TS1) is checked against the last

Figure 3-3: Server Naming Scheme

time the partition changed (TS), and in the case that TS1 < TS, the updates of the partition are sent along with the original request back to the client, through LS. In cases where a remote partition's image is out of date (an inconsistent view see section 2.11), the remote LS will notify the client and will send the updated partition's image to the client.

3.8. Summary

This section presented an approach to distributed OMS, concentrating on partitioning and dynamic configuration. The main idea is to exploit the object-oriented paradigm into distribution, and partition dynamically the objectbase along its (hierarchical) structure, such that ‘‘close’’ data is fetched efficiently and as the data becomes ‘‘far’’ it is fetched with more cost. The main advantages of this approach are high level of distribution, scale, and underlying structure for implementation of cooperative concurrent engineering. Dynamic reconfiguration is achieved by the notion of Local-Servers which are dynamic entities, each of which has sole control over its partition, including sub-partitioning and delegation of authority over sub-partitions. In order to provide reasonable performance for accessing objects that might be managed by different servers at different times, a special object-naming scheme was presented.

Part B - Implementation for MARVEL

4. Introduction

This part presents the implementation of a single-server OMS for MARVEL, and the overall process of upgrading MARVEL into a multi-user environment. The problem domain is reduced to a specific kind of environment, namely - Rule-Based Development Environments (RBDE) - and the specific issues related to RBDEs are examined.

4.1. Overview of Single-User MARVEL

MARVEL 2.6 is a single-user process-oriented Software Development Environment (SDE) kernel. Its main characteristics are:

Tailorable - the environment is tailored to project-specific and/or user-specific requirements. A project administrator writes a specification of the project *data model* and *process model*. Both specifications are written in the MARVEL Strategy Language (MSL). The administrator loads these specifications into the kernel, generating a MARVEL environment that supports both the data management and process management requirements.

Object-Oriented Data Model - Software artifacts are abstracted as instances of classes and multiple inheritance is supported. Objects are interconnected by set-of attributes, which create the *composition hierarchy*, and by directed links, which allow arbitrary semantic connections as specified in the data model. Object persistence is provided through a simple object manager.

Rule-Based Process Model - The process model encapsulates the activities and the overall behavior of the environment. MARVEL's rule-based approach provides as part of the process model an assistance model, via both backward and forward chaining. The process is described in terms of rules that specify the behavior of the tailored MARVEL environment. As depicted in figure 4-1, each rule contains a *condition*, an *activity*, and multiple mutually-exclusive *effects*. Conditions are read-only queries on the state of the objectbase. They are formed by a query sub-language that supports the basic first-order logic operators and quantifiers[14]. A condition must be satisfied for the rule to fire. Activities encapsulate an action that involves invoking external tools via the *envelope* mechanism. Effects assert the result of the activity's execution

[14]Currently, there is no direct query interface to the user. It is intended to be added to MARVEL.

```
rule:
      condition
      { activity }
      effect1; effect2; . . .
```

Figure 4-1: A Generic Rule in MARVEL

into the MARVEL objectbase.

Rule Chaining - If a condition of rule R1 is not satisfied, MARVEL attempts to satisfy it by *backward chaining* to other rules that might satisfy R1. Alternatively, if R1's assertions change the objectbase in such a way that the conditions of other rules become satisfied, these rules are fired automatically, forming a *forward chain*. Both backward and forward chaining are recursive operations, that is, the process of satisfying conditions can backward-chain to any number of rules[15], and so is the process of forward-chaining. MARVEL supports two models of assistance based on chaining, namely - automation and consistency maintenance.[16] This distinction is supported in MSL, and therefore is user-defined, rather than system-defined. This distinction enhances the process model and enables to improve the concurrency control and recovery mechanisms, as will be seen in section 6.

Tool integration - is provided by the *envelope* mechanism. Envelopes (currently implemented as shell scripts) execute commercial off-the-shelf tools. Each envelope has knowledge of the proper interface to execute the tool, and returns information to MARVEL, which asserts specific effects on the objectbase based upon the results of the tool execution. Since the envelope is a ‘‘black box’’, the actual assertions are not known until its execution is complete.

Built-In Commands - MARVEL maintains a set of built-in commands to manipulate and browse through the objectbase. These can be considered as direct access to the objectbase, as opposed to indirect access, via rules. Although from abstraction and process-model aspects it may be better if all access is done only through rules, for practical reasons, it is important to maintain the low-level direct interface. However, in order to incorporate built-in operations within the process model, MARVEL has the novel feature of allowing to add conditions and effects to the

---

[15]provided that there is no infinite loop

[16]this feature exists only in MARVEL 3.0

operations.[17]

Graphical User Interface - The interesting feature provided is the visual image of the objectbase (see figure).

4.2. Requirements for Multi-User MARVEL 3.0

The long-term goal of the MARVEL project is to develop a kernel for multi-user development environments that allows teams of programmers to cooperate on developing large-scale software projects. The motivation is to retain (and possibly enhance) the general characteristics of single-user MARVEL as described in previous section, and extend them with a layer of support for multiple users by providing means for *cooperation* and ensuring data consistency. This extension implies major changes and enhancements. The most crucial change is that MARVEL shifts from being a single-thread program into a multi-thread program. MARVEL follows the single-server architecture and is built according to the client/server model. Some of the associated problems are: (1) How to divide the workload between the clients and the server, in particular with respect to rule-processing; (2) Provide a mechanism for interleaved execution of multiple rule-chains; (3) Efficient communication-protocols; and (4) How to maintain an updated version of the objectbase-image at each user. These are the main topics of section 5. The second major requirement is to provide a robust, persistent data management system. Since the MARVEL data-model is object-oriented, the database system that implements it is OODBMS. This is the main subject of the thesis, and the problems and solutions associated with the general case of OODBMS were discussed in previous sections. RBDE-specific problems include control over concurrent rule-chain execution, interaction with the host file system, and practical implementation problems. The OMS for MARVEL is the topic of section 6.

4.3. Overview

As mentioned above, the main two issues addressed here are (1) architectural solutions in upgrading MARVEL to a multi-thread program and providing a mechanism for concurrent execution at the server on behalf of multiple clients, and (2) transaction management to provide consistency in the case of concurrent execution. In general, these two issues might be tightly coupled i.e. - the implementation of the mechanism might be heavily influenced by the policy,

---

[17]also exists in 3.0 only

mainly for optimization reasons. However, in accordance with the componentization philosophy adopted in this thesis, the idea is to try to provide generic mechanisms that enable maximum flexibility with regard to specific management policies imposed on top of them, and to try to avoid preemption of any specific policy. Thus, these two issues are treated separately. Chapter 5 lays out the underlying mechanisms for concurrent rule execution, presents a mechanism for communication among processes, and a solution to the problem of maintaining consistent views of the objectbase.

Chapter 6 presents a complete OMS that implements MARVEL's data-model following the specifications given in chapter 2, including transaction management. However, currently there is no full separation between the various components, as much of the code in the single-user monolithic system is still used. As an example, the MARVEL query processor is dependent on the rule processor and is not separated from it. Future work on MARVEL should build MARVEL to be completely separated into independent, separate components. Chapter 7 presents related work in all areas that are related to this thesis, including full OODBMSs, object-oriented storage managers, transaction managers, programming environments, etc. Chapter 8 is a summary of the thesis, and indicates future directions.

5. Upgrading MARVEL into a Multi-User System

This chapter covers all aspects of upgrading MARVEL into a multi-user system that are not directly related to data-management (these are covered in chapter 6). The modules described here are:

- Environmental Support (ES), or MARVEL's Rule Processor (RP).
- Communication Protocol (IPC).
- User Interface (UI).

5.1. Rule Processing (RP)

One of the major design issues in upgrading MARVEL was upgrading the rule processing module (RP), which provides the environmental support for, and is a fundamental component of MARVEL. Note that a full model was already in existence in MARVEL so this involved integrating RP into the multi-user, single-server architecture. This section presents the architectural solutions for rule processing in a multi-user setting. The problem of maintaining concurrency control and consistency in the data is addressed in section 6.4.

A classical problem in a client/server architecture is how to divide the workload between the server and the clients to achieve optimal performance. The parameters that affect this decision are:

- Hardware parameters - Processing and storage capabilities of the clients and the servers and communication bandwidth.
- Software parameters - To what extent a client's application has to communicate with the server, e.g., whether a typical application is data intensive or not.

There is an inherent trade-off between the communication overhead implied by distribution and processing overhead (and a potential bottleneck) at the server in centralized systems. Dividing the workload in MARVEL involved determining where and how RP and Built-In commands (BI) will be executed.

Built-in commands, which constitute the complementary module to RP in accessing the objectbase and providing services to the clients, were dealt with on a case-by-case basis. For instance, any command that does not access the objectbase at all (e.g., help) is completely executed at the client. Other commands are filtered through a client's command pre-processor, which performs argument type checking and reformatting to ease the processing at the server and prevent incorrect invocations of a command to be passed to the server.

In contrast, the RP division was a major design decision since it determined where the chaining engine resides. The hardware configuration assumed in MARVEL (although not mandatory) is a powerful server with large storage and computing capacities, and clients with small storage capacity but still considerable computing capability (e.g., a workstation with a small local disk). This means that most of the data should reside at the server and data-intensive operations should be carried out at the server, while computing-intensive operations should be done at the client to avoid overhead at the server and utilize the client's resources. However, since there are no specific client/server hardware configuration requirements for MARVEL, software parameters were mainly considered.

At first glance, there are three alternatives for how to divide RP:

1. Do all processing at the server.
2. Do all processing at the client.
3. Do some processing at the server and some at the client.

Alternative 1, while possibly applicable in general rule-based systems (although probably not a good solution anyway since the client's resources are not being utilized at all), cannot be applied to a process-oriented rule-based system such as MARVEL since in this context rules may abstract some interactive activity (e.g., editing a file) that must be carried out by the client. Alternative 2 is not applicable as well since query processing (the condition part) as well as the effects on the objectbase must be executed at the server. So, some form of partial execution at the clients and the server has to take place. Observing rule execution in process-oriented rule-based systems, three levels of granularity can be identified:

1. The action part of a rule, which abstracts a single activity in the process.
2. A single rule, consisting of condition-action-effects, which abstracts the constraints for, and updates from, executing the activity.
3. A rule chain, consisting of set of rules, which abstracts complex tasks and performs automation and consistency activities.

In MARVEL there is a fourth level - *sessions* that abstract a set of rule chains. However, this level is not relevant here, and will be discussed in section 6.5. As mentioned above, actions must be executed at the clients, while conditions and effects are executed at the server. However, rule-chain management could be done either at the clients or at the server. So the problem reduces to whether to do rule-chain management at the clients while interacting with the server for conditions and effects or to do it at the server while interacting with the clients to carry out the activities.

The decision was to choose the latter, i.e., - have a server-based rule-chain engine that sends the activities to the clients.

The advantages of this approach are:

(1) Due to chaining, a user-initiated rule might take very long, access an arbitrary number of objects and therefore might produce many conflicts with other rule chains or user commands. Furthermore, due to the non-deterministic nature of chaining, there is no way to know a priori the read-set and write-set. In addition, even a single activity in MARVEL (e.g., editing, compiling) may take very long. This means that conflict detection/resolution is likely to be invoked frequently regardless of the concurrency control policy. Although server-based RP faces the same problems, most of the data objects are not distributed, so concurrency control is alleviated, and much of the communication overhead is avoided. Cache coherency problems are avoided as well, since all effects on the objectbase are done at the server. (Recall that the state of the objectbase is changed only through the effects of a rule.)

(2) Typically, the number of objects accessed by a rule's condition far exceeds the number of objects actually passed as arguments to an activity. Thus, significant communication overhead is saved by sending to the client only the objects that the activity actually needs.

(3) The server-based approach enables to execute rules in the context of transactions and by having knowledge of the concurrent rule-chain execution (as opposed to knowledge only about concurrent access to data in the case of the client-based approach) it can provide better concurrency control, especially for cooperative transaction models.

(4) With server-based RP the clients do not have to be modified when RP is upgraded or even when the system upgrades from single-server into multi-server. Thus there is better separation between the process and data management - and the application.

(5) It is possible to direct non-interactive activities to other clients, either for distributing the workload or because the activity requires special resources that are not present at the requesting client.

The main disadvantage of this approach is the processing overhead at the server, that might create a bottleneck while not utilizing the client's processing resources. However, in a multi-server architecture we anticipate that LS will usually be local to its clients (i.e., it will reside at the same machine) thus the distribution of processing is irrelevant in this case. Recall (section 3) that most queries are expected to be local thus server-server communication is minimal. To summarize, server-based rule processing avoids the problems of distribution among the clients and makes the clients more independent entities while enabling the distribution of rule processing among many servers. Distributed RP is not addressed in this thesis and may be a whole research direction.

Implementing server-based RP imposes the following problems:

1. A rule might be invoked simultaneously in different rule chains, thereby interfering with each other's runtime data.
2. In order to facilitate a reasonable degree of concurrency, it is obvious that rule chains should interleave with each other at the RP. Moreover, it is not realistic that even a single rule will be carried out atomically, since a rule's activity executed at the client might take long and would essentially leave the server idle waiting for the client to finish, thereby preempting concurrency.
3. As a consequence, since chaining is a recursive algorithm in nature, interrupting it would require some form of simulating the recursion and maintaining the information per chain, much like recursion is implemented in high-level programming languages.

The solutions to these problems are presented below.

5.1.1. Reentrant Rules

Since MARVEL supports *late-binding*, all arguments and condition-based bindings are determined at run-time. Each rule in MARVEL maintains a symbol table containing all the formal parameters and the precondition variables, and when a rule is invoked, its symbol table is filled with the run-time objects. Since all rules are executed in the context of one RP (i.e., one address space) concurrent invocations of a specific rule by multiple clients might interfere with each other.

A similar scenario occurs in memory-resident programs that are invoked from a shared-memory partition such as in IBM VM/CMS. These programs are characterized as being *reentrant*. The idea is then - to make rules *reentrant*. Each invocation of a rule entails creation of a rule-frame, which consists of a pointer to the (read-only) rule, and a dynamically allocated data section, initialized with the symbol-table containing the formal parameters of a rule. The symbol table is obtained by copying it from the rule's template symbol table. The symbol-table is then filled with the values of the actual parameters, and remains throughout the entire life cycle of a rule-chain. Multiple invocations of a rule within a chain are treated identically, i.e., - they are represented as multiple entries in the chain, each with its own private section.

5.1.2. Rule Context Switching

To address the second problem it was necessary to to build some kind of *context-switch* mechanism in the rule engine. Context switch in multi-tasking operating systems (OS) [5] is the mechanism that enables multiple tasks to be interleaved in their execution to create the illusion that they are being executed concurrently. The basic idea is to maintain for each task a *context* that holds its current state. When a task A is suspended temporarily (a decision made by the scheduler due to various reasons such as waiting for I/O, cpu quota, etc.) its context is kept and another task, B, can execute. When A is ready to continue execution it is inserted into the ready queue and on its turn, its context is restored and A resumes execution. Context switch is considered an expensive operation, since the state of a process consists of complex data structures to be restored (e.g., the state of the registers at the point of suspension), and the OS normally has to change its mode of execution.

Rule-chain context switching is analogous to OS context switching [5]. A rule-chain is initiated by a user-invoked rule. When this rule arrives at RP, RP creates a context that essentially consists of a special data-structure (the *rule-stack*), and some additional state information. When a rule-chain is suspended, its context is kept, and another rule-chain from the waiting queue is processed. The scheduling criteria is simple: Whenever a rule's condition in a chain **C** is satisfied (thus is ready to execute) the activity is sent to the client and **C** is suspended. When the activity has completed execution at the client, it sends a message to RP. RP then wakes up the suspended rule-chain **C** and queues it into the ready (FIFO) queue, to resume execution. Despite the conceptual similarities, rule context switching is much simpler than the OS mechanism described above since: (1) The scheduling is simple. RP is not interrupt-driven and the decision to switch contexts is under full control of RP (although a chain can be aborted as will be seen in section 6). (2) Context switch can only occur at one specific place in a rule as described above. Therefore, the rest of the rule must be executed serially, i.e., - it has a long critical-section. (3) The context holds only rule-chain related status information, which is a small amount of information relatively.

Maintaining Chaining-State Information

In designing the data structure for maintaining chaining-state information (the chain's *context*), the following properties have to be satisfied:

(1) Backward chaining in MARVEL is a multiply-recursive algorithm, i.e., it generates a tree of recursive invocations (as opposed to simple recursive algorithms that generate a list). Moreover, the generated tree is an AND-OR tree, since in some cases a condition may be satisfiable only by an application of a set of rules and in other cases one rule out of many optional rules is sufficient for satisfying the condition. This implies some form of a tree structure on the context.

(2) Forward Chaining can be implemented in two different ways:

1. FIFO - fire all rules in the set of satisfiable rules first, and for each fired rule queue the forward chained rules (aka as breadth-first).
2. LIFO - After firing each rule, push the forward chained rules and execute them first (aka as depth-first).

The second alternative is preferred because rules that chain to each other are much more conceptually related than rules that happen to be all in a forward chain, i.e., if the rules in the forward set { R1, R2, R3 } become satisfied as a result of applying rule R, and R4 is satisfied as a result of applying R1, than it is more reasonable to fire the rules in the order R1,R4,R2,R3 than R1,R2,R3,R4. For example, assume R abstracts linking a program, R1 abstracts executing a program, R2 abstracts archiving a program, R3 abstracts checking-in of a program, and R4 abstracts debugging a program. Assume that R is fired and generates the above list for forward chaining. Then R1 (execute) is called and crashes. This triggers R4 (the debugger) in the forward-chain set. It is obvious that the debugger rule has to be called before the other rules since it is related to the previous rule and generates a better process model. Moreover, the application of R4 may invalidate other rules in the forward set and prevent them from being needlessly fired.

A related issue is how to order the rules both in backward and forward chaining to produce a chaining model that is both optimized and reflects the process better. Possible approaches may include applying an ‘‘importance’’ function to a rule using heuristics based on the frequency of calls, number and type of parameters and so forth. This may be a future research direction in process-oriented rule-based environments.

The Chaining Stack

The chaining stack addresses the issues stated above. It is a multi-level stack, whereby each level in the stack consists of an ordered set of rules that correspond only to the first rule in the previous level, i.e., they may satisfy the precondition of that rule (but are not related to other rules in the previous level). Effectively, the stack implements an AND-OR tree. RP can execute in two modes depending on the context of the chain: backward or forward mode. Once a chain has entered forward mode it cannot revert to backward mode[18]. When a user-invoked rule, baseR, is entered to RP, its preconditions are evaluated. If satisfied, the activity is sent to the client, the chain mode is set to forward and is immediately suspended. Else, the chain status is set to backward mode; for each predicate in the condition, all rules that can potentially satisfy baseR (candidate rules) are pushed into the stack, and are evaluated. This process continues recursively until a rule R with satisfied preconditions is found. Once found, all candidate rules for a specific predicate are popped. If the level is an AND level, then all predicates would have to be evaluated and satisfied in the same manner whereas in case of an OR level, only one of the predicates has to be satisfied. Once a rule's condition is satisfied it is sent for execution at the client, and the chain is suspended. On return, the next rule in the stack is sent for execution (if indeed its precondition were satisfied by the previous rule). When baseR's activity returns from the client, RP executes in forward mode; it pushes all triggered rules onto the stack, pops a rule, sends it for execution and suspends the chain. The chaining process ends when the stack becomes empty or due to exceptional termination (this will be discussed in section 6.4).

5.2. IPC layer

The Inter Process Communication (IPC) layer was a new module in MARVEL that did not exist in the single user system. Thus, unlike rule processing, where the model existed and had to be upgraded, this layer required implementation from scratch. The IPC layer follows the specifications set forth in section 2.9. There are two main problems in implementing this level:

1. Linearization/delinearization of the objectbase - at the object-level layer.
2. Handling arbitrarily large objectbases - at the byte-level layer.

[18]Although this will change to accommodate consistency constraints in forward/backward chaining (see section 6.4.2).

5.2.1. Object-Level Functions

Transfer of (sub)objectbase from the server to the client usually triggered by the need to update the graphical-image at the client. This means that the *contents* of the objectbase need not be transferred but rather sufficient information that enables to draw the hierarchy and refer to any object. However, since the same mechanism will be used in the multi-server architecture (in which contents of objects will be passed as well), a general mechanism had to be built. The main problem as stated above is to linearize the pointer-based objectbase (or parts of it) for sending and rebuilding the structure when receiving. Although there are some well-known protocols that deal with pointer conversions, like SUN's RPC-XDR and Apollo's NCS, these are usually generic protocols not suited to specific types of structures or applications. Here we wanted to build a protocol specifically suited for the efficient transfer of MARVEL-like objectbases, i.e., primarily a forest with additional ‘‘weaker’’ links. The algorithm for the send/recv of any sub-objectbase starting at a given object, top_object, is as follows:

send_ob(top_object):

(1) Do a Breadth First Search (BFS) traversal of the objectbase starting from top_object.

(2) For each visited object, insert the required attributes' values into the linear buffer. Note that some types of transfer may request specific attributes to be sent. For instance, when sending the objectbase image in the graphical interface only the name and id of an object have to be transferred. Link attributes are passed as sets of object ids that represent the linked objects.

(3) When all children of a parent are visited, insert a separator into the buffer.

recv_ob(top_object):

This is a two pass process. The first pass constructs the hierarchy and the second pass reconnects the horizontal links (see 2.4).

First pass: read objects from the buffer and construct the hierarchy. Whenever an object is read, it is enqueued to the BFS queue and inserted into a hash table keyed by object id. Whenever a separator is encountered the next object is dequeued and all subsequently read objects (until the separator) are connected to it.

Second pass: Do a BFS traversal of the objectbase and for each object connect all objects linked via horizontal link by looking up the hash table for the address of the object that was generated in the first pass. The hash table, besides being used by the second pass in recv_ob is also used to identify the location of top objects when a subset of the objectbase is transferred. This is discussed in section 5.3.

Performance of send/recv

Let V denote the number of objects in the objectbase, E denote the number of hierarchical edges, and H denote the number of horizontal edges. Note that H will normally be much less than E since the objectbase is primarily hierarchical, and V and E are roughly the same (V - E = R, where R is the number of trees in the forest). The send algorithm being a BFS runs in O(V). The receive algorithm, ignoring the complexity of the hash function, runs in O(V + H), since each object and each horizontal link is visited only once. If we assume a reasonable hash function, on the average case this will produce small negligible overhead of O(H), whereas the worst case will give us $O(H^2)$ but in any case with small H we can conclude that both algorithms are linear in V and the overhead in each operation is small.

5.2.2. Low-Level Functions

In sending, the object-level IPC functions linearize the objects into an array and call the low-level send function to actually carry out the transfer. In receiving, the process is reversed: the low-level function is called to receive the buffer, which is passed to the object-level receive function. In MARVEL, a client can be connected to a server either locally or remotely. In the former case, they are connected via UNIX sockets, while in the latter they are connected via TCP sockets. In either case a reliable transmission is assumed, so no measures are taken to assure that a particular transfer was successful. The problem reduces to ensure that arbitrarily large objectbases can be sent/received correctly, without loss of information. The idea is to provide the object-level layer with an interface to send/receive an object, without being concerned with the actual transfer mechanism. The implementation is done at the low-level layer, which does the actual transmission/reception. The communication details are hidden much like the actual data representation and location is hidden in persistent object systems (thus it may be called *single-level transmission*). The implementation of the low-level functions is straight forward: they maintain a buffer to hold the incoming/outgoing data and when the buffer is full it is sent/received.

5.3. User Interface

As stated in section 4, MARVEL provides an enhanced graphical user interface that includes an image of the objectbase (see figure 5-1). Upgrading the UI module to multi-user MARVEL brought up the problem of maintaining an updated version of the objectbase image at the clients. Note that high visibility is an important characteristic of DHOMS as well as of MARVEL, so maintaining the same interface as in single user MARVEL was of great concern. This problem is analogous to the cache coherency problem in distributed databases with replication, where the replicated data has to be updated when some changes are made to the master copy. However, here the objectbase image is provides only handles to the data in the server, and does not represent a local copy of the objectbase, therefore there is no risk of violating consistency in the objectbase since any updates to the objectbase are done at the server. The only consideration is whether the client has a current view of the objectbase at hand. If it doesn't then some of the queries might fail due to the inconsistent view, but the objectbase itself would not be affected. The alternatives for solving this coherency problem are:

1. Any change made in the objectbase is propagated to all clients immediately by asynchronous messages from the server to its clients.
2. None of the changes made by other clients are propagated until an explicit request for a modified object is made.
3. Changes to the state of the objectbase are piggybacked with the next response from the server to a client.

The first alternative continuously keeps the clients up to date but is likely to cause tremendous communication overhead and slow both the clients and the server. Moreover, it is unnecessary to keep an idle client up to date just because it is connected to the server. The second alternative is too relaxed and does not maintain the high visibility property. Thus, the third alternative, being a popular technique in network protocols, seems like a reasonable compromise with low overhead since piggybacking the image has the advantage of not initiating a special message but rather adding information to a message that would have been sent anyway. The question is to determine when to piggyback a new image and what parts of it to send.

When to Transfer

To determine *when* to send an update, the server maintains for each client a virtual *timestamp* that represents the time of the most recent service request made by this client. The timestamp is part of the client's *session context* that exists throughout the lifetime of the client and holds software and hardware parameters about the client. It is discussed in section 6.5. In addition, the

Figure 5-1: MARVEL Graphical Interface server maintains a *server-session* that maintains global parameters. One of them is the *global timestamp*, which represents the last time any change to the objectbase occurred. When a client is being served by the server, its timestamp entry is checked against the global-timestamp and if it is smaller, an update is piggybacked with the response. Note that the computational overhead in this case is minimal, unlike timestamping as used for cache coherency, since the timestamp is per session rather than per object.

What to Transfer

Currently, MARVEL's server sends the entire objectbase in each update request, except in the case of updating the image of the client who actually caused the change. This client gets back only the portion that was affected during the service request. However, this method is expensive and does not scale up as the objectbase size grows. An improved method (probably to be implemented as part of MARVEL 3.0) follows :

To determine *what* to send in an update, the server maintains along with the global-timestamp an object-id of the top-most updated object. In addition, all the global-timestamp information is recorded. When it is determined to send an update to a client, all top objects accessed in timestamps that are marked later than the client's have to be sent. Then, a quick analysis is made in order to determine common ancestors to avoid sending the same object twice. Finally the selected set of objects is sent along with their descendants (unless single object update occurred, in which case only a single object is sent). The idea here is the same as used in data-partitioning, in the multi-server model: utilize the hierarchical structure of the objectbase and assume locality of reference to derive implicit information. Here, instead of marking each object with a timestamp and checking for updates across the entire objectbase, only top objects are marked and maintained while the descendants are implicitly marked (unless otherwise specified).

6. MARVEL Single-Server OMS

Single-user MARVEL had a crude object manager that provided persistence and other minimal services. While possibly adequate for small objectbases in a single user setting, it was very limited and could not scale up. All the objectbase information was kept in an ascii file in a way that enforced either writing or reading the entire objectbase for each read/write operation; since it was built for a single-user system there was no notion of concurrency control short of locking the entire objectbase when starting up a session. Crash recovery was implemented by maintaining a recent copy of the (entire) objectbase without a possibility to refine the granularity and no consideration was given to the hidden file system. However, the data model was well developed and is one of the fundamental concepts in MARVEL. Thus, the motivation was to retain the data model and implement it with a new, multi-user OMS.

The implementation of the data management subsystem followed the specifications set forth in section 2. Therefore, this section concentrates on MARVEL-specific concepts and problems, and presents the implementation of the OM. The modules discussed here are:

1. OM - Object manager.
2. SM - Storage manager.
3. LM - Lock manager.
4. TM - Transaction manager.
5. SEM - Session manager.
6. QP - Query processor.

With the exception of SEM, all other components were specified in section 2 as well as the interactions among them. SEM is more related to the Environmental Support module and therefore is not mentioned there.

Overview

The overall architecture of the OMS is best illustrated in figure 6-1, which is an instance of the general single-server architecture (figure 2-1) and follows the basic layered model. The server has three basic interfaces to the objectbase, namely - Built-Ins (BI), Rules (RP), and Queries (QP). BI supports mainly direct, navigational access to the objectbase through atomic operations that change the structure of the objectbase; RP provides environmental support by abstracting the activities via rule-chaining; and QP provides associative access to the objectbase. Currently,

Figure 6-1: MARVEL Client/Server Architecture there is no direct interface to QP and it is accessible only from RP in terms of pre-conditions. One level below resides the Transaction Manager (TM). Every operation acted upon the objectbase from any of the above interfaces is encapsulated in a transaction, which like in other DBMS's serves as a unit that is guaranteed to preserve the consistency of the objectbase according to the semantics defined on it, and a unit of recovery. Concurrency control is achieved by consulting the Lock Manager (LM), which is a separate, independent module. LM is responsible to detect conflicts as defined by TM and report them to TM with maximum diagnostic information so that TM's conflict resolution layer can make the correct and optimal (e.g., with respect to cooperation) decisions according to its concurrency control model. Below TM and LM is the Data Management module (DM), composed of the Object Manager (OM) and the Storage Manager (SM). OM implements a full object-oriented data model, and any access to any of the project's components is through the object abstraction, including file objects. Thus, OM hides the file system interface from the user. Persistence is provided through SM, the lowest layer in the system, which lacks any knowledge about the model, and performs an efficient byte-level storage and buffer management. Each component's implementation will be described in detail in the following sections.

6.1. MARVEL Object Manager (OM)

As suggested in [13], object management in programming environments must combine the capabilities of a DBMS and a File Management System (FMS). The alternatives are: (1) to extend an existing DBMS to handle file-like entities; (2) to integrate the FMS with an OMS. MARVEL adopts the latter approach, by exploiting the FMS ability to store files efficiently, and by providing a layer of object-oriented abstraction on top of it. Thus, the file system is used internally but is hidden from the users. Using composite-objects (see below) MARVEL's OM can map the hierarchical file system into the OMS in a natural way. For instance, most FMS's make a distinction between binary and text files (and maybe a few more distinctions). In MARVEL a text-file object can be an instance of DOCFILE (e.g., a latex file), a CFILE (e.g., a C source code file), an HFILE (a C header file) or just a FILE (generic file, superclass of all the above classes). It can basically be defined as an instance of any class as defined by the system administrator. This does not mean necessarily that each object maps into a file or part of a file in the file system. Many objects can represent arbitrary entities such as programmers, equipment, etc. However, it does mean that any file in the file system that is part of the environment is mapped in some way or another onto an object.

OM consists of two entities:

1. The objectbase.
2. The hidden file system.

6.1.1. Objectbase

MARVEL's objectbase consists of objects in the object-oriented sense, with hierarchical classification and inheritance that abstract all the entities in a software project such as sources, binaries, modules, libraries, documents, etc. In addition, it supports composite objects, that abstract the "composition hierarchy" ([26]) and arbitrary links among objects. Following the specifications in section 2 it consists of two main components:

- The *schema* consisting of all class definitions including the operations (aka as *behavior*). This comprises all the static properties of the objects.
- The *contents*, consisting of all instances represented as a collection of values of attributes that represent the state of the objects, preceded by an object unique identifier.

OM always maintains a copy of the schema in virtual memory. Persistence of the schema is provided simply by writing it to secondary storage when a change occurs. Although MARVEL allows to change the schema (by loading new schema), it is intended mainly to change the operations, not the classes (there are no means currently to ensure correct schema evolution), so it is rarely modified.

It is the *contents* entity that is of great concern, since it is a dynamic entity that constantly changes and must be kept in a consistent state. Persistence of the contents is provided by the Storage Manager.

OM provides the following interface to the objectbase:

- read_object(id,depth)
- write_object(id,depth)
- link_object(src_object, link_att,dest_object);
- unlink_object(src_object, link_att,dest_object);
- copy_object(src_object, dest_parent);
- move_object(src_object, dest_parent);
- rename_object(id, new_name);

- add_object(name,parent-id, att);
- delete_object(id);

The read/write operations perform navigational search to obtain the object's subcomponents and store/fetch each object using calls to SM.

Read/write and link/unlink affect only the objectbase, while the other commands affect both the objectbase and the file system. Therefore, they perform calls both to SM and FMS.

In addition, OM provides functions to navigate through the objectbase.

6.1.2. The Hidden File System

MARVEL uses the FMS (Unix file system) to store two kinds of files: (1) the actual files used by the software developer, such as the sources, binaries, etc. These are usually mapped to a single object that abstracts them although occasionally a single object may have multiple file attributes, in which case it encapsulates a set of files in the file system; (2) files that represent more abstract objects such as project, department, etc. These are usually directories in the FMS. Normally, a composite object will have file objects as its leaves and directories that are abstracted as internal nodes in the objectbase. However, the data model does not require the finest granularity of objects to be a single file. For example, an object could represent a function within a source file object, or even a statement in a program object. (Currently however, MARVEL does not support finer granularity than a file.) The file system is hidden from the user and is accessible only through the objectbase. An object that abstracts a file (or a set of files) has a *file system* attribute that encodes the location of that file in the file system, and accessing the file is made possible only through invocation of an operation (*method* in the OOP jargon, or rules in MARVEL) defined on that object. Thus, the approach here is to abstract the file system with an object-oriented layer, thereby enriching its limited type set, while still using the FMS to handle the storage management of files. As far as OM is concerned, a file is a stream of uninterpreted bytes referenced through an attribute of an object.

6.2. MARVEL Storage Manager (SM)

SM is responsible for providing persistence to the objectbase. Recall that SM is independent from the data model, therefore it serves as a byte store. This makes the SM replaceable. Although there are several operational storage managers that are specifically designed to support the object-oriented paradigm, ([28], [36], [41]), they all impose either a specific data model or a specific transaction model, thus eliminating modularity. It was decided to implement SM on top of the Unix dbm package. Dbm is a generic storage manager that performs fast fetch/store operations on variable-length data-items and maintains a cache of objects that are replaced by an LRU policy. However, SM is able to force a write to the disk, using the Unix fsync system call. The objects are indexed by a key (itself a variable-length object) and are stored in a hash table. Our implementation uses the object id as the key and all connections among objects are replaced by object ids in the data base. Thus, a call by OM to read a composite object amounts to reading the top object, obtaining the ids of the children, and than reading them, recursively. Dbm also enables to specify exclusive and shared access to the dbm file to ensure consistency in case of concurrent accesses to it, using the Unix file locking mechanism. The advantages of using dbm are:

- Simple and provides generic interface.
- Is a standard Unix library.
- Performs well.

The major drawback to dbm is that it does not provide a mechanism to specify *where* to place the objects, so there is no control over the physical or logical placement of objects on disk (it is determined internally, by the hash function). This rules out any clustering mechanism, and it also rules out the ability to read a set of (related) objects in a single fetch request. Another limitation is that the object size is limited to the system's block size to assure fast access time (no more than two I/O operations). For storing larger objects more work has to be done to split/join the objects after they were read. However, in MARVEL this is not a problem, since objects are small (typically 40 - 60 bytes long). Recall that the special file attributes are only pointers and do not contain the contents themselves. It may be the case that future versions of MARVEL will include an enhanced storage manager that will enable to specify where to store objects, in which case OM will have to adopt a clustering policy to utilize this capability.

6.3. MARVEL Lock Manager (LM)

Following the single-server architecture, LM is a separate component. One of the strengths of this model is its ability to support a variety of concurrency control policies with minimal effects on other modules. (Recall that concurrency control is divided between detection and resolution, where LM performs detection and TM performs resolution.) MARVEL LM does this by reading the specifications for, and performing, conflict detection based on external conflict information. Thus, one can define the conflict detection in a certain way that supports the concurrency control policy. Moreover, once that layer is defined, LM still does not enforce any lock policy nor does it take any action to prevent two incompatible locks to coexist. If the transaction manager approves an apparent conflict to exist (for instance if two transactions are executing in some cooperative mode) it can enforce any combination of locks on an object. For that matter, LM can hold multiple locks on an object regardless of their compatibility. The conflict information consists of the following components:

- Compatibility matrix.
- Ancestor lock table.
- Power matrix

The compatibility matrix is used to check whether two locks are compatible. The ancestor lock table indicates what lock to apply to the parent of the object being locked in a certain mode. This information is very useful in composition hierarchies, as will be seen below. The power matrix is used to determine which lock is more ‘‘powerful’’ than the other given two locks, i.e., which one has precedence and should be set on the object. This applies only when the locks are acquired on behalf of the same transaction. In any other case both locks will be held. The implementors of the concurrency control module are free to specify this information. Thus, change in the conflict policy can be achieved without code modifications. This feature was applied to MARVEL in trying to find an appropriate concurrency control model, and turned out to be an important feature both for testing and tuning the behavior of the system. The current detection layer in MARVEL (i.e., the conflict information) is described below.

Granularity Locking

The composition hierarchy in MARVEL implies that although each object is an independent, identifiable entity, an object that contains another object or is part of other object cannot always be treated in isolation from its ancestors and descendants since some of the operations on that object may affect the state of the connected objects. For instance, deleting an object implies deleting all its descendants, and adding an object as a child of another object adds a link to the parent object, thereby changing its state. However, many operations alter or access only the state of the object that they operate on, such as editing a source file or changing the name of an object. This problem does not exist in relational databases or object-oriented databases without a composition hierarchy. The trivial solution to this problem is to lock all descendants and ancestors of an object regardless of the semantics of the operation. However, this preempts concurrency and is much too restrictive, since it does not allow clients to operate together on two components of an object just because they share a common ancestor. Basically, it allows only one client per top object in the hierarchy. Another solution, implemented also in the ORION system [18], is to use *intention locks*, also known as *granularity locking*. When operating on an object O1, all O1's ancestors are locked in intention mode. When a client wants to apply an operation on an ancestor of O1, LM checks for conflicts against the intention lock. Intention locks are weaker than the locks put on the original object and their goal is to protect the original locked object from being affected by an operation acted upon an ancestor or other sibling. Thus, if object D is locked in L mode, all of its ancestors will have IL locks and IL may be compatible with any operation that does not affect D. In particular, it will be compatible with another IL lock.

MARVEL's LM implements a variation of ORION's locking scheme. In addition to ORION's exclusive and shared locks and their intention counterparts, MARVEL provides a way to distinguish between those operations that act upon an object itself, and those which also affect some of the object's descendants. Although this could be achieved by locking an object and all its descendants before performing the operation, this is highly inefficient. The solution is then to have another set of locks, namely super-exclusive/shared locks, that are not compatible with exclusive/shared intention locks, and are effectively implying exclusive/shared locks on all descendants of the object.

The complete lock information is given in figures 6-2, 6-4, and 6-3. It includes the compatibility-matrix, power-matrix, and the ancestor lock table. Note that in future versions of MARVEL the conflict detection information will be filled in by an administrator according to the coordination policy.

6.4. MARVEL Transaction Manager (TM)

According to the specifications in section 2, MARVEL Transaction Manager (TM) defines the conflict resolution layer and implements concurrency control, transactions and crash recovery. In MARVEL there are currently two methods to access the objectbase:

- Direct - via the Built-Ins module.
- Indirect - via rules. This includes both user initiated and chained rules.

```
# IS = Intention Shared; IX = Intention Exclusive;
# S = Shared;  X = Exclusive;
# SS = Super Shared; SX = Super Exclusive;
#
#          -------------------------------------------
#          | \   |  IS |  IX |  S  |  X  | SS  | SX  |
#          |-----|-----|-----|-----|-----|-----|-----|
#          |  IS | yes | yes | yes | yes | yes | no  |
#          |-----|-----|-----|-----|-----|-----|-----|
#          |  IX |     | yes | yes | yes | yes | no  |
#          |-----|-----|-----|-----|-----|-----|-----|
#          |  S  |     |     | yes | no  | yes | no  |
#          |-----|-----|-----|-----|-----|-----|-----|
#          |  X  |     |     |     | no  | no  | no  |
#          |-----|-----|-----|-----|-----|-----|-----|
#          | SS  |     |     |     |     | yes | no  |
#          |-----|-----|-----|-----|-----|-----|-----|
#          | SX  |     |     |     |     |     | no  |
#          ------------------------------------------|
#
```

Figure 6-2: Compatibility Matrix in MARVEL

```
#
#          -------------------------------------------
#          | \   |  IS |  IX |  S  |  X  | SS  | SX  |
#          |-----|-----|-----|-----|-----|-----|-----|
#          |  IS |     | IX  |  S  |  X  | SS  | SX  |
#          |-----|-----|-----|-----|-----|-----|-----|
#          |  IX |     |     |  S  |  X  | SS  |     |
#          |-----|-----|-----|-----|-----|-----|-----|
#          |  S  |     |     |     |  X  | SS  |     |
#          |-----|-----|-----|-----|-----|-----|-----|
#          |  X  |     |     |     |     |     |     |
#          |-----|-----|-----|-----|-----|-----|-----|
#          | SS  |     |     |     |     |     |     |
#          |-----|-----|-----|-----|-----|-----|-----|
#          | SX  |     |     |     |     |     |     |
#           ------------------------------------------|
```

Figure 6-3: Power Matrix in MARVEL

The direct interface via the query processor is not available yet. Built-Ins encapsulate single, short operations and are implemented as non-interleaved transactions, i.e., they are processed in their entirety at once at the server, and therefore do not cause problems. The method is straight

```
IS --> IS
IX --> IX
 S --> IS
 X --> IX
SS --> IS
SX --> IX
```

Figure 6-4: Ancestor Lock Table in MARVEL

forward: If there is a conflict in the read/write set (which is acquired at once), the transaction is aborted and the affects of the transaction are canceled. Otherwise it is committed.

Maintaining consistency while allowing concurrency in concurrent rule-chain execution is much harder. As already pointed out, from the transaction management point of view, rule chains in MARVEL can be regarded as (1) long (due to chaining and type of activities), (2) interactive, and (3) highly-concurrent transactions. Each of those properties as well as their combination is a source of difficulty for the transaction manager.

(1) implies that the abort primitive must minimize the undo operation, since it is not desirable to remove all work that has been going on for long time just because of a single conflict.

(2) implies that it is highly undesirable to block a transaction. Moreover, since transactions are long, blocking might lead to a live-lock situation.[19] In general, humans don't like to be blocked. They would rather be aborted and do other things.

(3) together with (1) implies that an optimistic approach is out of the question since this will probably cause endless rollback operations, which are not desirable as explained above.

In addition to the above constraints, chaining gives rise to some form of *nested transaction* model ([30]), whereby the user-invoked rule initiates a transaction and all chained rules (both in backward and forward chains) are treated as sub-transactions. Wherever applicable, the consequences of this model will be pointed out. The actual concurrency-control and recovery policies are explained below. Rules and subtransactions are used interchangeably as well as rule-chains and nested transactions. Also, backward and forward chains are treated the same by

[19]In live-lock a transaction may have to wait indefinitely, but it is not dead-locked, therefore dead-lock detection will not track it.

TM unless otherwise noted.

6.4.1. Concurrency Control

MARVEL's current concurrency control is a variation of two-phase locking (2PL) [14] as applied to nested transactions . A lock acquired by a subtransaction ST is also considered as a lock of its parent transaction PT. Like in 2PL, no locks can be acquired after any lock has been released with one exception: when a subtransaction is aborted while the parent transaction continues execution, the subtransaction releases its locks on objects that are not locked by the parent transaction. Another difference is that blocking is not permitted. Instead, a transaction that faces a conflict is aborted. This might produce a considerable amount of aborts, but is still preferable over blocking for the reasons given above. The idea is to try to minimize the amount of effects that have to be undone. This is shown in the next section. Note that MARVEL's current policy is quite simple and does not have any cooperation primitives. That is why it does not support the tx_share primitive mentioned in section 2. However, as pointed out along the thesis, the architecture of DHOMS was driven towards supporting a variety of concurrency control policies, and cooperative models in particular. Major research efforts are given towards enhancing transaction management in programming environments in general ([23], [37]) and rule-base environments in particular [8]. It is anticipated that future versions of MARVEL will incorporate more enhanced concurrency control policies.

6.4.2. Transaction Recovery

Tolerating transaction recovery is made through the following observation: Recall that rules in MARVEL chain for two different reasons: automation and consistency. In automation-based chaining we can treat each rule in a chain as a separate transaction with regard to recovery. However, consistency-based chaining implies that if rule R2 was fired due to consistency predicate P off rule R1, then if R2 aborts (and there is no other rule that can satisfy P), R1 must be aborted as well. In this case, rule-chains follow the *nested transaction* model. Thus, differentiating consistency and automation chains reduces the amount of information that has to be undone in case of an abort, since only the effects of rules in consistency chains have to be undone.

When a subtransaction is aborted due to a conflict or user intervention, TM enters into recovery mode. It does so by tracking all the consistency-based chaining that occurred up to that time in the parent transaction, and undoes their effects. Note that automation-based chaining need not be undone. The idea is to track subchains that are linked only by consistency predicates. The following definitions will be used later on:

- A *consistency chain* is a chain of nested transactions that are spawned due to consistency constraints. Likewise, an *automation chain* is a chain in which all subtransactions are spawned due to automation predicates. A rule-chain consists of interleaved consistency and automation subchains.
- A subtransaction T1 is *recoverable* if it starts a consistency chain, i.e., it spawns at least one subtransaction due to a consistency constraint.
- A subtransaction T2 is *committable* if all its consistency chains have been completed.

Assume T1 is a recoverable transaction, and C is the rule-chain spawned on behalf of T1. Whenever C turns from consistency mode to automation mode (due to an automation link in either forward or backward chaining), the spawned subtransaction T2 is independent in terms of recovery, i.e., T2 cannot affect the state of T1 so T1 is committable. If T1 spawned a single chain, then the effects of T1 will eventually be committed. However, if T1 spawns multiple consistency chains, it can only commit after all subchains have been committed. Therefore a committable subtransaction does not always commit its effects. Figure 6-5 illustrates these observations. C denotes consistency-chain and A denotes automation chain. In the single-list case, all effects of R1, R2, and R3 will be committed as soon as R3 completes execution, regardless of the outcome of applying R4. However, in the case of multiple chains, R1 is only committable after R3 has completed execution. If either of R5, R6, or R7 fails, the effects of the chain R1-R2-R3 will not be committed. Note that R1 need not be the beginning of a chain. It can be a beginning of a consistency chain that proceeds from an automation chain.

Aborting a Transaction

If a subtransaction T aborts, then: If the rule is executed in an automation chain only the immediate effects of the aborted transaction are not committed and the abort procedure ends. If the rule is in a consistency chain, TM must roll back and undo all the effects of the rules in that consistency chain, including all consistency chains that were spawned by any of the ancestors of that rule.

Optimizing Undo

One of the problems associated with nested transactions is that of of multiple copies of objects within one (nested) transaction. This problem intensifies if the copies are of large objects, such

```
  R1                             R1
  |                              /\
C |                           C /  \ C
  |                            /    \
  v                           R2    R5
  R2                          /      \
  |                       C  /        \ C
C |                         /          \
  |                        R3          R6
  v                       /            /\
  R3                   A /          C /  \ A
  |                     /            /    \
A |                    R4           R7    R8
  |
  V
  R4
```

Figure 6-5: Abort in Rule Chains

as files in programming environments. The motivation is then to: (1) try to minimize the number of copies maintained per object in a nested transaction, and (2) minimize space consumption of each copy.

Minimizing the number of copies can be done following a simple rule:

- In a consistency chain - maintain only the *first* and the current copy.
- In an automation chain - maintain only the *last* and the current copy.

If an abort occurs during a consistency chain, then all copies except the first one are discarded. In automation chains there is no reason to hold old copies that would never be addressed. Note that these rules apply to multiple copies of the same objects. If a subtransaction in a chain accesses new objects they are treated as the first copy. Finally, if a chain is purely an automation chain, or if a concurrency chain has finished and there is no other ‘‘sibling’’ concurrency chain, all effects can be committed immediately after execution. This both reduces the volume of copy, and it enhances shareability of objects.

Optimizing space consumption can be done by maintaining only copies of modified attributes, rather than copies of entire objects. This is extremely important in MARVEL, when file objects are modified in an attribute that is not the file attribute. In this case, only a small amount of information is copied rather then the entire file, which may be arbitrarily large.

6.4.3. Crash Recovery

Crash recovery was not considered in this thesis and is one of the topics for future research (see section 8). However, simple crash recovery mechanism is supported in MARVEL, which includes automatic restoration of all uncommitted transactions when the server starts up, checkpointing of the objectbase, and maintaining previous images of the objectbase on disk to restore an objectbase after a system crash that occurred before the buffers were written to disk.

6.5. MARVEL SEssion Manager (SEM)

This module is part of the Environmental Support module. Session management encapsulates an entire session between a human user and the environment [22]. The idea is to provide another layer in which user-specific parameters that are valid throughout the session can be defined once (at initialization) and last for all interactions of the user with the system. Sessions can be used to express a desired behavior that differs from the default one. They can also be used to define the focus or goals of sessions, thereby providing hints to the environment as to the optimal way it can help the user to achieve them. MARVEL currently has a basic SEM module. Each active user of the system has an *session-context* identified by a session-id (a user may have several sessions). This entry defines the environment parameters of the user, such as the type of interface (graphical or command-line), verbosity mode, system paths to lookup for information, etc. In addition, the session entry holds some active values such as the current-object, which represents the focus of the user in the objectbase and serves as a starting node in some object lookup algorithms. Whenever a transaction is processed at the server, its session context is exposed in addition to the transaction context. MARVEL also provides an interface to modify the session arguments on-line, via the SET built-in command. In the future, the session would be able to hold hints for the transaction manager such as group information in case of transaction groups or recovery mode, etc. Basically, any user-oriented information that may help any of the modules in the server should be added in SEM.

6.6. MARVEL Query Processor (QP)

QP provides associative and navigational access to the objectbase. As mentioned above, MARVEL does not currently support a direct interface to QP. Instead, queries are posed inside the condition of rules. MARVEL QP provides the basic logical propositional operators AND, OR, and NOT, and the quantifiers FORALL and EXIST, as well as any combination of them. In addition, MARVEL provides navigational access via the MEMBER, ANCESTOR, LINK-TO, LINK-FROM, operator that returns the parent or the child of a given object. The associative access is made against the class hierarchy while the navigational access is done against the objectbase hierarchy. A typical query is composed of two parts: (1) The characteristic function - which selects a set S of candidate objects based on their type and their location in the objectbase hierarchy, and (2) the property list - which selects a subset of S based on the actual values of their properties (attributes). Figure 6-6 is an example of a typical query in MARVEL.

```
#
# Characteristic function:
#
# Select : all objects ?l from class LIB
#               that share the same parent as p,
#          all objects ?m from class MODULE
#                that are members of p
#          all objects ?c from class CFILE
#                that are members of p
#
#

(and (forall PROJECT ?P
             suchthat (member [?P.programs ?p]))
     (forall LIB ?l
             suchthat (member [?P.libraries ?l]))
     (forall MODULE ?m
             suchthat (member [?p.modules ?m]))
     (forall CFILE ?c
             suchthat (member [?p.cfiles ?c])))
    :

  # Property list:
  #
  # Select : all ?c, ?m, ?l that satisfy the
  #          given property values
  #

  (and (?c.analyze_status = Analyzed)
       (?c.compile_status = Compiled)
       (?m.status = Archived)
       (?l.archive_status = Archived))
```

Figure 6-6: Query Processing in MARVEL

7. Related Work

This chapter presents research in related areas and its relevance to DHOMS. Section 7.1 presents work on OODBMSs, i.e., systems that support an object-oriented data model and provide all the required functionality defined in section 1. Section 7.2 presents work on systems and models that provide partial functionality such as storage, object and transaction management.

7.1. OODBMS

ORION ([6], [24], [18] [25]), was originally developed as a LISP-machine system that integrated object-oriented programming with a database system by extending the language to include primitives that manipulate its object-oriented database. It was later extended into a generic OODBMS, and was ported to UNIX. MARVEL OMS resembles ORION's in the support for composite-objects (which is a superset of MARVEL's composite objects), and in the granularity locking on the composite object hierarchy. However, MARVEL's Lock Manager (LM) is capable of supporting any set of locks and any compatibility among them by treating this information as external to LM. ORION-1SX resembles the single-server architecture in that it follows the client/server paradigm and it is modular. However, it is not layered like in DHOMS, and there is no clear-cut separation between data and transaction management. Orion-2 [25] is an attempt to construct distributed OODBMS and is still under development.

Cricket [41] is an object-based database, that is intended to provide a persistent platform for the implementation of design environments. It advocates *single-level* store. It uses the memory management primitives of the Mach operating-system (such as *external pager*) to control the virtual memory management. No specific data model is defined, but concurrency control is built-in, and there is no separation between data and transaction management.

Other work in this area include Gemstone, IRIS, O2.

7.2. Storage and Object Managers

The work of Brown [11] is closely related to single-server DHOMS in its layered approach to the architecture of an OMS, in particular in the separation of data-management from transactions. It is intended to provide the supporting platform for the implementation of persistent programming languages, such as Napier88 [29] and Galileo [2].

ObServer ([16], [36], [17]) is an object server specifically designed for SDEs. It assumes a distributed environment, and includes a rich set of locks including non-restrictive locks and notification-modes that support cooperation. The lock mechanism has, in addition to lock modes, communication modes that indicate whether or not a transaction wants to be notified when a locked object is updated, or when another transaction wants to acquire a lock on object that is already locked by the transaction. In particular, ObServer supports transaction-groups [37], which is one kind of cooperative transaction. ObServer does not define a data-model, which is an advantage as a storage manager, from DHOMS point of view. However, the support for transactions at the data management level violates the separation between data and process management. Another drawback of ObServer (based on [3]) is that there is no support for single-level store, and the client must know how and where to store objects.

EXODUS [35] is an extensible DBMS that supports development of application-specific DBMS, and resembles DHOMS in the sense that it is not restricted to specific policies. In particular it has a full separation between the storage module and the type module, to support abstract data types such as in an object-oriented data model. However, it does not provide multi-user support.

NSE ([31]) (a commercial product) is an environment that supports parallel development through an optimistic concurrency control mechanism that allows users to access objects without locking them. When multiple users modify the same file, a merge process helps the users to merge the multiple versions into one. File management in NSE is based on a concept of *environment*, a separate work space that has its own copy of a set of files, and that serves them to its members. These environments can nest to several levels, thus they resemble the hierarchy of DHOMS. However, NSE is not object-oriented, and, as noted in 6.4, optimistic concurrency control method is unsuitable to RBDEs with a high probability of conflicts.

8. Summary and Future Directions

In this thesis an attempt was made to define and specify a framework for design and implementation of an Object Management System that addresses the requirements of multi-user design environments. The first approach concentrated on a centralized approach, exploiting the ideas of componentization and layering. The main emphasis was on separating user-oriented transactions from data management, while trying to provide maximum support for upper layers to implement a variety of policies, including policies that aim at cooperation. As an example, the lock set can be arbitrarily defined, the TX_share primitive is a general-purpose cooperation-aid primitive, and the storage manager is independent from any data or transaction model.

The second approach aimed at a more complicated target, namely, distributed OMS, concentrating on a method for distribution of the data and on a method for dynamic reconfiguration of this distribution, both done in a way that preserves the overall semantic structure of the system. Motivation for partitioning was performance (obvious) and cooperation, in which partitioning could serve as a way to group users based on their need to work together on a partition, thus providing a mechanism for close coordination among them, by sharing a common server. The idea here was to exploit the object-oriented paradigm in distribution by imposing a structure of servers on the logical structure of the data. For dynamic reconfiguration, a naming scheme was invented that minimizes the impact of reconfiguration. The idea was to exploit the hierarchical configuration of the servers in order to maintain minimum information that is sufficient for accessing remote objects.

The third part (and initial motivation of the thesis) was the implementation and integration of an OMS in MARVEL using the theoretical scheme. The fact that MARVEL was already an existing system prevented full componentization since much of the code was still used, and the components, therefore are not completely separate. A preparation step was to upgrade MARVEL into a multi-tasking system, in particular with regard to concurrent execution of rules and defining the responsibilities of the clients and the server. Rationale for, and implementation of rule context-switching and rule-reentrancy were presented. Other practical solutions were presented for the implementation of various components of the OMS such as the lock manager, object manager, and transaction manager. As noted in the preface, the higher the layer is - the more semantic knowledge it contains. Therefore, transaction management had a lot of room for implementation following the specifications, and a simple concurrency control engine was built.

A unique implementation of transaction recovery was presented, in which transactions can abort partially (rather than entirely) due to semantic knowledge of the dependencies between their components.

The initial plan was to implement the centralized model, followed by the distributed extension. However, as pointed out, upgrading a single-task single-user system into multi-tasking multi-user system was a major task that required much more than implementation of multi-user OMS. In fact, it required major overhaul, and touched virtually (and physically) each component of the system.

8.1. Future Work

The main future work in the direction of DHOMS should be an implementation of an independent DHOMS system that will serve as a platform to support environments. This will exploit the componentization philosophy by treating the entire DHOMS system as a component that can be hooked with any environment. Another work in that direction should be to fully componentize MARVEL, into separate, replaceable components.

Another major direction would be to complete the specifications for a distributed model and implement a prototype that will enable to conduct performance analysis and evaluate the low-level support for cooperation.

Finally, much work has to be done in individual components and areas, some of them are current research topics. A partial list includes schema-evolution, crash recovery, implementation of non-traditional transaction managers on top of the primitives provided by lower levels, distributed transaction processing, formal query-language for OODBMS, standards for OODBMS, and clustering.

9. Acknowledgments

I would like to thank Gail Kaiser, my advisor, for her advice, guidance, and support along the entire period of implementing the system and writing the thesis. Thanks are also due to Naser Barghouti and George Heineman for their overall contribution to the MARVEL project and their many suggestions during long design sessions on transactions, rule processing, and overall design issues. Thanks to Tim Jones for his contribution to the design and implementation of the object and storage managers. Finally, thanks to Ouri Wolfson for his careful review of the thesis and his insightful comments.

I. Appendix A: Single-Server Interface Specifications

The interfaces are presented in the form of functions, in a C-like style, i.e. - return value (if significant) is given first, followed by the function name and parameters, followed by a short description.

I.1. Storage Manager (SM)

SM_open(ob) - Open the objectbase and make all necessary initializations such as storage allocation, etc.

datum *SM_read(address)* - read a datum from the given address.

SM_write(datum,address) - write datum into the given address.

SM_close(ob) - close the objectbase and make all necessary deallocations of storage.

I.2. File Manager (FM)

Interface to deal separately with file attributes. A UNIX-like hierarchical file system is assumed.

FM_file_add(path,name, type) - add a file/directory (depending on type) under the specified path.

FM_file_delete(path) - delete (recursively) directory and all its child directories.

FM_file_rename(path,newname,type, template) - Rename file. If directory, rename the children files according to the template. (e.g., in a $name.c template, it will be renamed to newname.c.)

FM_file_copy(src_path,dst_path) - copy (recursively) a file.

FM_file_move(src_path,dst_path) - move (recursively) a file.

I.3. Object Manager (OM)

object *OM_read_object(oid,<depth>)* - read an object specified by oid and return it. Depth specifies how many levels to follow for composite objects (by default all descendants are read).

OM_write_object(oid,<depth>) - write an object specified by oid. depth is defined as above.

OM_create_object(parent,class,name,<attribute values>) - generates an object id for the object and allocates the appropriate storage in memory. If attribute values are specified, the default values are taken from the class lattice. Also connects the new object to its parent.

OM_remove_object(oid) - free the object-id for reuse and deallocate storage.

OM_modify_attribute(oid,att,val,valtype) - write val into oid's att. Type checking is provided here to assure that att can get type of val.

OM_get_links(oid,att) - get the object(s) linked from oid by the att attribute.

I.4. Lock Manager (LM)

LM_set_lock(object,mode,tid) - Try to set the requested lock on behalf of transaction tid. In case of failure return diagnostic information including the transactions that conflict and their modes.

LM_reset_lock(object,tid) - Clear the lock on id on behalf of transaction tid.

LM_enforce_lock(object,tid) - Set the requested lock regardless of the detection policy.

I.5. Transaction Manager (TM)

TX_begin() - Begin a transaction. Return a tid

TX_read(tid,oid) - read an object denoted by oid.

TX_write(tid,oid) - write an object denoted by oid.

TX_strong_read(tid,object) - like read, except call LM with stronger lock.

TX_strong_write(tid,object) - like write, except call LM with stronger lock.

TX_add(tid,object) - add an object. Impose locks on parent.

TX_delete(tid,object) - delete an object. Impose locks on parent.

TX_commit(tid) - Commit, i.e., make all transient information permanent.

TX_abort(tid) - Abort, retaining the objectbase in a consistent state.

TX_share(tid1,tid2) - allow transaction T1 to share the context of transaction T2.

I.6. Built-In commands (BI)

BI_move_object(oid,target-parent) - move the specified object as son of the target-parent.

BI_copy_object(oid,target-parent) - copy the specified object as son of the target-parent.

BI_rename_object(oid,new-name) - self explanatory.

BI_connect_object(source,destination) - supports horizontal connection among objects (see section 2.4).

BI_disconnect_object(source,destination) - disconnect a link.

I.7. Query Processor (QP)

QP provides the following set of primitives that are used in constructing queries: (The queries are posed against classes.)

Quantifiers: EXISTS, FORALL

Propositional Operators: AND, OR, NOT

Navigational Operators: MEMBER, ANCESTOR, LINKED-TO, LINK-FROM (but not DESCENDANTS).

SUCHTHAT operators: <, >, <>, =, <=, >= . they can be performed on various types of operands that maintain partial order among them.

I.8. Inter Process Communication (IPC)

Object-Level Functions

In the following, oid stands for objectid, pid stands for process id, and cid is the class id of the object sent. The & sign represents the address of the object and is basically treated as an output variable, like in C programs. Note that pid is not necessarily the Unix process id, but rather an identification of the sending/receiving entity as defined in DHOMS.

send_object(pid,oid,cid) - Send a single object to the process identified by pid. This is a simple operation that involves sending the values of all attributes of that object, each prefixed by its class-id, for proper typing.

recv_object(&pid,&oid,&cid) - No input arguments. The receiving process resolves the pid of the sending process, the class id of the received object and its contents, including its oid. The receiving process then attempts to further connect the object with the rest of its objectbase image.

send_composite_object(pid,id,cid) -

recv_composite_object(&pid,&oid,&cid) -

Like *send/recv_object* except the whole subcomponents of the object are transferred. Since DHOMS relies on hierarchical structure of the objectbase, the traversal goes only through hierarchical edges, while other ("horizontal") edges are marked with stubs that represent the id of the referenced objects. Thus, hierarchically connected subobjects are guaranteed to be in the stream sent to *recv_com_object*. However, horizontally connected objects are sent only if they connect to objects within the subtree of oid. If they don't, the pointer is replaced by the stub that represents the referenced object's id. If the receiving process finds the referenced object in its image, it replaces the stub with a pointer to that object. When the transfer is complete, the client makes a second pass on the new composite object to look for unresolved links and tries to resolve them.

send_message(pid,buf,buflen) -

recv_message(&pid,&buf,buflen) - arbitrary long message that has to be sent between any two DHOMS processes. Messages could involve commands, error messages, etc.

byte-Level Functions

send(pid,buf,header) - send fix sized packets along with a DHOMS header that include various information such as the size of the data in the buffer, its sequence number, etc. (see section 5).

recv(pid,&buf,&header) - like send.

I.9. User Interface (UI)

display_object(name) - print the object's contents.

display_class(name) - print the class structure.

display_class_hierarchy() -

display_objectbase() - The standard image at the client.

zoomin/zoomout/pan objectbase - graphical browse through the hierarchy.

II. Appendix B: Partitioning/Rejoining Procedures

II.1. Partitioning Procedure

1. LS verifies that it can be partitioned. This includes checking that LS has no clients connected to it and that no remote requests are handled at that time. Another subtle issue to be verified is that an LS cannot be created on a partition which was already owned in part by other LS.
2. LS creates a new LS (say LS1) and passes to it the needed information regarding the new partition. (e.g., the id of the top object, the location of the partition in secondary storage and the owner machine if different. From now on LS1 is a subordinate server of LS.
3. LS reclaims the in-core storage of the subpartition, after checkpointing it. It then replaces the top of the subpartition's pointer with a stub that represents its object id. In a DAG objectbase, all further pointers from objects in LS to objects in LS1 are replaced with stubs.
4. LS adds an entry in its LDT specifying the partition and the address of LS1.
5. LS notifies PS about the new entry, and PS updates its GDT.
6. LS1 initializes itself and sends a ready message to LS once it has control over its partition.

II.2. Rejoining Procedure

1. LS1 verifies that no clients are connected to it at this moment.
2. LS1 verifies that it can be rejoined with its parent (e.g., if it is still alive and awaken).
3. LS1 checkpoints its partition, frees all the acquired in-core storage and sends a ready-to-die message to LS.
4. LS removes LS1's entry in LDT and notifies PS to update its GDT.
5. LS kills LS1.

References

[1] M. Ahlsen et al.
An Architecture for Object Management in OIS.
In *ACM Transactions on Office Information Systems*. July, 1984.

[2] A. Albano, L. Cardelli, and R. Orsini.
Galileo: A Strongly Typed, Interactive Conceptual Language.
In *ACM Transactions on Database Systems*, pages 230-260. 1985.

[3] Neil Arora.
Evaluation of ObServer for implementation in Marvel.
December, 1989
Columbia University.

[4] M. M. Astrahan et al.
System R: A Relational Approach to Database Management.
In *ACM Transactions on Database Systems*. 1976.

[5] Morris Bach.
The Design of the Unix Operating System.
Prentice Hall, 1986.

[6] Jay Banerjee, Haong-Tai Chou, Jorge F. Garza, Won Kim, Darrell Woelk, Nat Ballou, and Hyoung-Joo Kim.
Data Model Issues for Object-Oriented Applications.
ACM Transactions on Office Information Systems 5(1):3-26, January, 1987.

[7] Naser S. Barghouti and Gail E. Kaiser.
Implementation of a Knowledge-Based Programming Environment.
In *21st Annual Hawaii International Conference on System Sciences*, pages 54-63. IEEE Computer Society Press, Los Alamitos, CA, January, 1988.

[8] Naser S. Barghouti and Gail E. Kaiser.
Multi-Agent Rule-Based Software Development Environments.
In *5th Annual Knowledge-Based Software Assistant Conference*, pages 375-387. Syracuse NY, September, 1990.

[9] Gail E. Kaiser, Israel Z. Ben-Shaul and Naser S. Barghouti.
Preliminary Design of an Object Management System for Multi-User MARVEL.
In Takuya Katayama (editor), *6th International Software Process Workshop: Support for the Software Process*. IEEE Computer Society Press, Hakodate, Hokkaido, Japan, October, 1990.
Position paper. In press.

[10] Grady Booch.
Object Oriented Design.
The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1991, Chapter 1.

[11] Alfred Brown and John Rosenberg.
Persistent Object Stores: An Implementation Technique.
In Alan Dearle and Gail M. Shaw and Stanley B. Zdonik (editor), *Implementing Persistent Object Bases: Principles and Practice*, pages 199-212. The Fourth International Workshop on Persistent Object Systems, Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990.

[12] Stefano Ceri and Giuseppe Pelagatti.
Distributed Databases.
McGraw Hill, 1985.

[13] C. Godart and F. Charoy.
Computer Aided Software Engineering Object Management Systems: Requirements and ongoing Work.

[14] C. J. Date.
An Introduction to Data Base Systems.
Addison Wesley, 1986.

[15] Perasun Dewan and Eric Vasilik.
An Object Model for Conventional Operating Systems.
Computing Systems, the Journal of the USENIX Association 3(4):517-549, Fall, 1990.

[16] 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.

[17] Mary Fernandez.
ObServer II Server Interface Specification
Brown University, 1989.

[18] Jorge F. Garza and Won Kim.
Transaction Management in an Object-Oriented Database System.
In *SIGMOD International Conference on Data Management*, pages 37-45. Chicago IL, June, 1988.
Special issue of *SIGMOD Record*, 17(3), September 1988.

[19] Theo Haerder and Kurt Rothermal.
Concepts for Transaction Recovery in Nested Transactions.
In Umeshwar Dayal and Irv Traiger (editor), *SIGMOD International Conference on the Management of Data*, pages 239-248. ACM Press, San Francisco, CA, May, 1987.

[20] Anthony L. Hosking and J. Eliot B. Moss.
Towards Compile-Time Optimisations for Persistence.
In Alan Dearle and Gail M. Shaw and Stanley B. Zdonik (editor), *Implementing Persistent Object Bases: Principles and Practice*, pages 17-28. The Fourth International Workshop on Persistent Object Systems, Morgan Kaufmann Publishers, Inc., San Mateo, CA., 1990.

[21] Gail E. Kaiser, Naser S. Barghouti, Peter H. Feiler and Robert W. Schwanke.
Database Support for Knowledge-Based Engineering Environments.
IEEE Expert 3(2):18-32, May, 1988.

[22] Gail E. Kaiser.
A Marvelous Extended Transaction Processing Model.
In Ritter, G. (editor), *Proceedings of the 11th World Computer Conference IFIP Congress '89,* pages 707-712. Elsevier Science Publishers B.V., San Francisco CA, August, 1989.

[23] Gail E. Kaiser.
A Flexible Transaction Model for Software Engineering.
In *6th International Conference on Data Engineering*. IEEE Computer Society, Los Angeles CA, February, 1990.

[24] W. Kim, N. Ballou, H. Chou and J. Garza.
Integrating an Object-Oriented Programming System with a Database System.
In *OOPSLA '88 Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, pages 142-152. San Diego CA, September, 1988.
Special issue of *SIGPLAN Notices*, 23(11), November 1988.

[25] Won Kim, Jorge F. Garza, Nathaniel Ballou, and Darrell Woelk.
Architecture of the ORION Next-Generation Database System.
IEEE Transactions on Knowledge and Data Engineering II(1):109-123, March, 1990.

[26] Won Kim.
Object-Oriented Databases: Definitions and Research Directions.
IEEE Transactions on Knowledge and Data Engineering II(3):327-341, September, 1990.

[27] Kim, W., Banerjee, J., and Chou, H.
Composite Object Support in an Object-Oriented Database System.
In *Proceedings of the 2nd International Conference on Object-Oriented Programming Systems, Languages and Applications,* pages 118-125. Orlando Fa, October, 1987.

[28] 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.

[29] R. Morrison, A. L. Brown, R. Conor, and A. Dearle.
The Napier88 Reference Manual.
Technical Report PPRR-77, Universities of Glazgow and St. Andrews, Scotland, 1989.

[30] J. Eliot B. Moss.
Information Systems: *Nested Transactions: An Approach to Reliable Distributed Computing.*
The MIT Press, Cambridge MA, 1985.
PhD Thesis, MIT LCS TR-260, April 1981.

[31] Evan Adams.
Object Management in a CASE Environment.

[32] Calton Pu, Gail E. Kaiser and Norman Hutchinson.
Split-Transactions for Open-Ended Activities.
In Francois Bancilhon and David J. Dewitt (editor), *14th International Conference on Very Large Data Bases*, pages 26-37. Los Angeles CA, August, 1988.

[33] Erhard Rahm.
Design and Evaluation of Concurrency Control and Coherency Control Techniques for Database Sharing Systems.
Technical Report, Universitat Kaiserslautern, Kaiserslautern, Germany, 1988.

[34] The Committee for Advanced DBMS Function.
Third-Generation Database System Manifesto.
SIGMOD RECORD 19(3):31-44, September, 1990.

[35] Joel E. Richardson and Michael J. Carey.
Programming Constructs for Database System Implementation in EXODUS.
In Umeshwar Dayal and Irv Traiger (editor), *SIGMOD International Conference on the Management of Data*, pages 208-219. ACM Press, San Francisco, CA, May, 1987.

[36] Skarra, A. H., Zdonik, S. B., and Reiss, S. P.
An Object Server for an Object-Oriented Database System.
In *Proceedings of the 1986 International Workshop on Object-Oriented Database Systems,* pages 196-204. Pacific Grove CA, September, 1986.

[37] Andrea H. Skarra and Stanley B. Zdonik.
Concurrency Control and Object-Oriented Databases.
Object-Oriented Concepts, Databases, and Applications.
ACM Press, New York, 1989, pages 395-421, Chapter 16.

[38] M. Stonebraker, E. Wong, P. Kreps, and G. Held.
The Design and Implementation of INGRES.
In *ACM Transactions on Database Systems*. 1976.

[39] M. Stonebraker.
Homogeneous Distributed Data Base Systems.
Distributed Databases.
Cambridge University Press, Cambridge, 1980.

[40] R. Williams.
R*: An Overview of the Architecture.
In *International Conference on Database Systems*. Jerusalem, Israel, 1982.

[41] Eugene Shekita and Michael Zwilling.
Cricket: A Mapped, Persistent Object Store.
In Alan Dearle and Gail M. Shaw and Stanley B. Zdonik (editor), *Implementing Persistent Object Bases: Principles and Practice*, pages 89-102. The Fourth International Workshop on Persistent Object Systems, Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990.

Table of Contents

1. Introduction 2
1.1. Research Goals 4
1.2. Thesis Contents 6
2. A Single-Server Architecture 9
2.1. Overview 9
2.2. Storage Manager (SM) 13
2.3. File Manager (FM) 14
2.4. Object Manager (OM) 14
2.5. Lock Manager (LM) 16
2.6. Transaction Manager (TM) 18
2.7. Query Processing (QP) 19
2.8. Environmental Support (ES) 19
2.9. Inter Process Communication (IPC) 19
2.10. User Interface (UI) 20
2.11. Client-Server Interface (CSI) 21
2.12. Summary 21
3. A Multi-Server Architecture 23
3.1. Background 23
3.2. Overview 24
3.3. The Dynamic Hierarchy model 26
3.3.1. Primary Server (PS) 26
3.4. Local Server (LS) 28
3.5. Naming Scheme 30
3.6. Distributed Transaction Management (DTM) 34
3.7. Visibility 34
3.8. Summary 36
Part B - Implementation for MARVEL 37
4. Introduction 37
4.1. Overview of Single-User MARVEL 37
4.2. Requirements for Multi-User MARVEL 3.0 39
4.3. Overview 39
5. Upgrading MARVEL into a Multi-User System 41
5.1. Rule Processing (RP) 41
5.1.1. Reentrant Rules 44
5.1.2. Rule Context Switching 45
5.2. IPC layer 47
5.2.1. Object-Level Functions 48
5.2.2. Low-Level Functions 49
5.3. User Interface 50
6. MARVEL Single-Server OMS 53
6.1. MARVEL Object Manager (OM) 55
6.1.1. Objectbase 56
6.1.2. The Hidden File System 57
6.2. MARVEL Storage Manager (SM) 57
6.3. MARVEL Lock Manager (LM) 58
6.4. MARVEL Transaction Manager (TM) 60
6.4.1. Concurrency Control 63
6.4.2. Transaction Recovery 63
6.4.3. Crash Recovery 66
6.5. MARVEL SEssion Manager (SEM) 66

6.6. MARVEL Query Processor (QP) 66
7. Related Work 68
7.1. OODBMS 68
7.2. Storage and Object Managers 68
8. Summary and Future Directions 70
8.1. Future Work 71
9. Acknowledgments 72
I. Appendix A: Single-Server Interface Specifications 73
I.1. Storage Manager (SM) 73
I.2. File Manager (FM) 73
I.3. Object Manager (OM) 73
I.4. Lock Manager (LM) 74
I.5. Transaction Manager (TM) 74
I.6. Built-In commands (BI) 75
I.7. Query Processor (QP) 75
I.8. Inter Process Communication (IPC) 75
I.9. User Interface (UI) 76
II. Appendix B: Partitioning/Rejoining Procedures 78
II.1. Partitioning Procedure 78
II.2. Rejoining Procedure 78

List of Figures

Figure 2-1:	**Single-Server Architecture**	**11**
Figure 3-1:	**Multi-Server Architecture**	**27**
Figure 3-2:	**Object Naming Scheme**	**32**
Figure 3-3:	**Server Naming Scheme**	**35**
Figure 4-1:	**A Generic Rule in MARVEL**	**38**
Figure 5-1:	**MARVEL Graphical Interface**	**51**
Figure 6-1:	**MARVEL Client/Server Architecture**	**54**
Figure 6-2:	**Compatibility Matrix in MARVEL**	**61**
Figure 6-3:	**Power Matrix in MARVEL**	**61**
Figure 6-4:	**Ancestor Lock Table in MARVEL**	**62**
Figure 6-5:	**Abort in Rule Chains**	**65**
Figure 6-6:	**Query Processing in MARVEL**	**67**