Building Interoperable Databases on Distributed Object Management Platforms

Asuman Dogac Cevdet Dengi M. Tamer Özsu

July 1, 1998

A common characteristic of today's information systems is the distribution of data among a number of autonomous and heterogeneous repositories. Increasingly, these repositories are database management systems (DBMS), but there is still a very large volume of data that is stored in file systems, spreadsheets and others. A fundamental challenge in building next generation information systems is to provide interoperability among these autonomous and potentially heterogeneous repositories. The commercial state-of-the-art in addressing the interoperability of DBMSs is to build gateways. This approach is quite restricted and provides only a partial solution. None of these systems properly deal with semantic or structural heterogeneity of the stored data. An alternative approach to achieving interoperability among DBMSs is the multidatabase approach [11]. A multidatabase system resides unabtrusively on top of existing database systems and presents a single database illusion to its users. In particular, a multidatabase system maintains a single global database schema against which its users issue queries and updates. It is suggested that this is the approach that should be preferred over gateways [5]. One restriction of multidatabase systems has been that they cannot handle repositories which do not have DBMS capabilities. The introduction of object-oriented technology into data management [3] has lifted this restriction. Object-orientation, with its encapsulation and abstraction capabilities, enable the development of wrappers which encapsulate a particular repository and provide a common DBMS-like interface to the rest of the system. The components of the interoperable system may be DBMSs, other types of repositories or legacy systems [1].

Fundamentally, interoperability of systems does not imply distribution. However, in practice, the systems that interoperate are usually distributed. A common and current example is the interoperation among data repositories on the World Wide Web. In such a distributed environment, capabilities and functionality of the distributed object management [9] platform have to be taken into account in designing the architecture of multidatabase systems.

This article discusses the role of distributed object computing platforms, in particular Object Management Group's (OMG) Object Management Architecture (OMA) [12] in providing an interoperability platform. We report our experiences in developing a distributed multidatabase system over such a platform.

Overview of OMA

Two new technologies, namely object oriented programming and distributed computing, have emerged in the last decade and influenced contemporary software to a great extent. Powerful workstations and PCs connected with high-speed networks and client-server software running on this distributed hardware is a reliable and relatively cheap alternative to centralized, mainframe based architectures. On the other hand object oriented technology dramatically reduces the complexity in software design and allows software reusability. Early client-server architectures such as Remote Procedure Call (RPC) do not have an object oriented model. A client needs to know how to access a service and also the location of the service. A client code must be changed whenever the client wants to use new services. The convergence of these two technologies, resulting in *distributed object systems* (DOMs) [9], is a natural step forward. The true benefits of distributed client-server computing can be realized by DOMs which permit the development of component software from which complex applications can be developed. Today, commercially available distributed systems support different levels of object-orientation. The RPC mechanism provided in Distributed Computing Environment (DCE) of the Open Software Foundation (OSF) supports encapsulation at the granularity of an individual server: A server accepts only the operations its interface defines. Yet DCE's RPC does not support abstraction, which is the ability to group associated entities according to common properties and polymorphism which is the ability of abstractions to overlap and intersect [7].

OMG's OMA is a promising attempt for a standard architecture that joins distributed computing and object-oriented programming technologies. OMA defines a Reference Model identifying and characterizing the components, interfaces, and protocols that compose a distributed object architecture. The OMA object model supports encapsulation, abstraction and polymorphism. The reference model has four basic components which are the Common Object Request Broker (CORBA) [2], Common Object Services (COSS), Common Facilities and Application Objects. CORBA is a middleware which enables distributed objects to operate on each other. COSS is a complementary standard for integrating distributed objects. CORBA and COSS together provide the basic infrastructure that can be used to provide database interoperability. They themselves, however, are not sufficient to provide multidatabase functionality.

It should be noted that Microsoft has a competing DOM architecture called Distributed Component Object Model (DCOM). DCOM and CORBA are the major players in the market today and there is ongoing work to provide the interoperability between these platforms.

CORBA

The key communication mechanism of OMA is CORBA where objects communicate with other objects via an ORB which provides brokering services between the clients and the servers. Brokering involves target object location, message delivery, and method binding. In this model, clients send requests to the ORB asking for certain services to be performed by whichever server can fulfill those needs. ORB finds the server, passes it the message from the client, receives the result which it then returns to the client.

In CORBA only the ORB knows about the implementation details and actual locations of the components in the system. Clients and servers only know the interfaces of the components. The only means of communication is the requests and their responses. In this way a distributed, heterogeneous environment becomes virtually local and homogeneous to the client. The changes in object implementation, or in object relocation has no effect on the clients. This reduces the complexity of client code dramatically and allows clients to discover new types of objects as they are added to the system and use them in a plug-and-play fashion without any change in the client code.

All a CORBA client knows about the target object is its object reference and its interface. An object reference should belong to an existing object which is generally created by an object factory. An object factory is itself a CORBA object. Clients may store the object references coming from an object factory themselves or may find them using the Naming Services. The interface determines the valid operations that can be performed on a particular object. Interfaces are defined using Interface Definition Language (IDL). An IDL interface declares a set of client accessible operations, exceptions, and typed attributes. IDL resembles a declarative subset of C++ but it is not a programming language, making ORB object development implementation language independent. To use or implement an interface, the interface must be translated, or mapped, into corresponding elements of a particular programming language. This means that an implementor will work with two different models: the "ORB" model when designing services and applications using IDL, and the implementation language model (C , C++, Java or Smalltalk) when implementing those ORB objects. Java mapping has a special importance since it allows Java scripts running on Web browsers access CORBA objects directly. IONA's OrbixWeb, SUNSoft's Joe, Visigenic's Visibroker and ORBeline's BlackWidow are examples of Java mappings.

Object implementations access most of the services provided by the ORB via object adapters, each of which is an interface to the ORB allowing the ORB to locate, activate and invoke operations on an ORB object. Until recently, only the Basic Object Adapter (BOA) was defined and had to be provided by all commercial ORBs. BOA was designed to be used with most of the object implementations and provides for generation and interpretation of object references, method invocation, registration, activation and deactivation of object implementations, selection of proper object implementation for a given object reference, and authentication. Recently, OMG s released a standard as an alternative to BOA. This standard, called the Persistent Object Adapter (POA), provides ORB portability. In the future OMG is expected to publish another standard for object DBMSs. Since ODBMSs provide some "ORB-like" services such as object reference generation and management, this adapter will be tuned to integrate ODBMSs with ORB distribution and communication. Library object adapter will be tuned for implementations resident in the client's process space.

OMG does not place any restriction on how ORBs are implemented. In most ORB implementations already existing IPC (Inter-Process Communication) methods such as UNIX socket libraries, shared memory and multi-thread libraries are used to achieve actual communication among clients, servers and the ORB. Yet, an ORB can be as simple as a library that supports communication among objects and their clients that actually resides in the same process space.

Common Object Services

The interfaces in OMA are categorized into three main groups: Interfaces of Object Services, interfaces of Common Facilities, and interfaces of Application Objects. Object Services provide the main functions for implementing basic object functionality using ORB. Each object service has a well defined interface definition and functional semantics that is orthogonal to other services. This orthogonality allows objects to use several object services at the same time without any confusion. For example, a file object that has transaction capabilities may use standard Transaction Service interface or a part of it.

It should be noted that it is not mandatory for objects to provide any of these interfaces. Yet, these standard interfaces provide "plug-and-play" reusability to objects. As an example, a client can move any object that supports Lifecycle Services by using the standard interface. If the object does not support the standard Lifecycle Services then the user needs to know "move semantics" for the object and its corresponding interface.

These services are at different phases of development. Standards on some important Common Object Services such as Naming, Lifecycle, Transaction, Trader, Security and Event Services are available. For others, requests for proposals have been released, but no standards are yet established. There are a few for which the request for proposals have yet to be released.

Common Facilities

Common facilities consist of components that provide services which are useful for the development of application objects in a CORBA environment. Two classes of facilities have been identified. Horizontal facilities consist of those facilities that are used by all (or many) application objects. Examples of these facilities include user interfaces, systems management and task management. Vertical facilities, on the other hand, are specialized components for selected application domains, such as the health care, transportation, manufacturing, electronic commerce or the telecommunications sector.

Implementing a Multidatabase System on CORBA

In a multidatabase system implementation, the main problem is the heterogeneity which basically exists at four levels: platform level, communication level, database system level and the semantic level. Database systems reside on different hardware and operating systems and use different communication protocols. CORBA provides implementation transparency which allows a client to access an object through its interface defined in IDL, independent of the hardware and software environment in which the object resides. This solves the platform heterogeneity problem. On the other hand CORBA provides location transparency which allows clients to access objects using their object identifiers independent of their location and communication protocols between the client and the object. This property of CORBA solves the communication level heterogeneity problem. The third level of heterogeneity is among the database management systems based on different data models and query languages. Finally, semantic conflicts are likely to be present among independently designed databases. This includes schema conflicts and data conflicts. The last two levels of heterogeneity in multidatabases on a DOM platform can be handled by developing a global layer which includes a global query manager, a global transaction manager and a schema integrator. Note that, schema integration is a hard problem that still requires further research.

Since CORBA provides an object-oriented framework for interoperability, implementing a multidatabase system on top of a distributed object management architecture like CORBA requires the definition of objects in IDL and providing their implementations. A fundamental design question, therefore, is the granularity of these objects. In registering a DBMS to CORBA, a row in a relational DBMS, an object in an ODBMS, a group of objects or a whole database can be an individual CORBA object. CORBA would accept all of these definitions. When fine granularity objects, like tables, are registered as objects, all the DBMS functionalities to process these tables, like querying, transactional control, etc., must be supported by the multidatabase system itself. However, when an entire relational DBMS, for example, is registered as an object, all the DBMS functionality needed to process these tables are left to that DBMS.

Another consideration regarding granularity has to do with the capabilities of the particular ORB that is used. In case of ORBs that only support BOA, each insertion and deletion of classes necessitates recompilation of the IDL code and rebuilding of the server. Thus, if the object granularity is fine, these ORBs incur significant overhead. If an ODBMS adapter is available, exporting fine granularity objects of ODBMSs will be more convenient since such an adapter relieves the overhead on the ORB by working in cooperation with the ODBMS to handle the objects it owns.

Yet another possible solution to this problem is to use dynamic server/skeleton interface. Although this interface is originally designed for inter-ORB interoperability, it prevents recompilation of the code and rebuilding of the server when used for dynamically adding new object types to the server side.

A second issue that needs to be addressed is the definition of the interfaces for database objects. Most commercial DBMSs support the basic transaction and query primitives either through their Call Level Interface (CLI) library routines or through their XA Interface library routines. This property makes it possible to define a generic database object interface through CORBA IDL to represent all the underlying DBMSs. CORBA allows multiple implementations of an interface. Hence it is possible to encapsulate each of the local DBMSs by providing a different implementation of the generic database object.

CORBA provides three alternatives for associating a client request with a server method: one interface to one implementation, one interface to one of many implementations and one interface to multiple implementations. If there exists only one implementation of an interface, all of the requests should be directed to a server that supports this single implementation. If there are more then one implementations of an interface, ORB can direct the requests to a server that supports any one of the existing implementations. In both cases, implementations handle all operations defined in the interface and after implementation selection, ORB always uses the same implementation for requests to a particular object. If each implementation of an interface does not handle all of the operations defined in the interface, that is, if each implementation provides only a part of the interface, the third method is used for associating a client request with a server method. In this case, ORB directs the requests to a server that support an implementation of the interface which handles the invoked operation. Since DBMSs that are registered to CORBA provide basic transaction management and query primitives for all of the operations that the interface definition specifies, the second alternative is usually sufficient.

CORBA defines three call modes, namely, synchronous, deferred synchronous, and one-way. In the synchronous mode, client waits for the completion of the requested operation. Synchronous mode can be restrictive for clients who issue operations that can be executed in parallel with multiple objects. In deferred synchronous mode, the client continues its execution after server selection and keeps on polling the server to get the results until the operation is completed. In one-way operation a client sends a request without any intention of getting a result. CORBA does not support asynchronous mode since the only way of communication is by a request. This implies that if a client is to receive asynchronous messages, it should also act as a server that implements an object that can receive requests. In other words, asynchronous mode of operation can be achieved between two CORBA objects sending one-way requests to each other. The only disadvantage of this peer-to-peer approach is the increased complexity of the client code. For objects of a multidatabase system synchronous call mode is generally sufficient. Deferred synchronous mode or peer-to-peer approach should be used when parallel execution is necessary. For example, in order to provide parallelism in query execution, the global query manager of a multidatabase should not wait for the query to complete after submitting it to a local DBMS by invoking the SendQuery method. Therefore SendQuery method should not be invoked using synchronous mode.

Another decision to be made is the invocation style for objects. CORBA allows both dynamic and stub-style invocation of the interfaces by the client. In stub-style interface invocation, the client uses generated code templates (stubs) that cannot be changed at run time. In dynamic invocation, on the other hand, the client defines and builds requests as it runs. When registering database objects, the stub-style interface

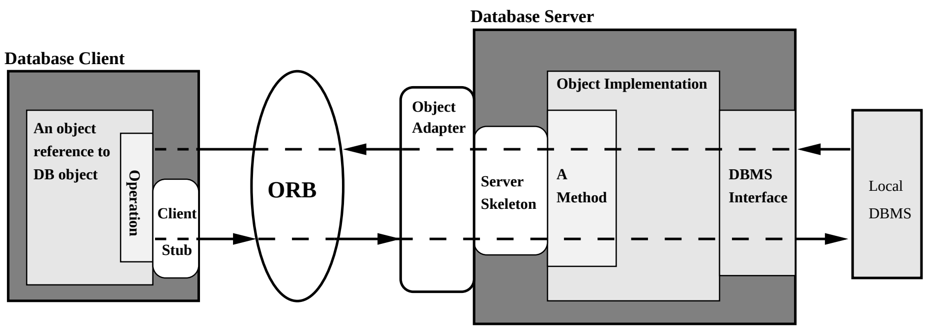


Figure 1: Invoking an operation to a Database Object instance through ORB

invocation is generally sufficient because all the objects are known and the interface of these database objects are not likely to change over time. Figure 1 illustrates invoking an operation on a database object instance.

When registering objects to CORBA, it is necessary to specify an activation policy for the implementation of each kind of object. This policy identifies how each implementation gets started. An implementation may support shared, unshared, server-per-method or persistent activation policies. While a server that uses shared activation policy can support more than one object, a server that uses unshared activation policy can support only one object at a time for an implementation. In server-per-method activation policy, a new server is used for each method invocation. Persistent activation policy is very similar to shared activation policy, except the server is never started automatically. Some of the objects in a multidatabase system need to be active concurrently. This can be achieved either by using threads on a server that uses shared activation policy or by using separate servers activated in the unshared mode for each object. Otherwise since a server can give service for one object at a time, other client requests to the objects owned by the same server should wait for the current request to complete. Furthermore all of the requests to an object must be serviced by the same server if the server keeps transient data for the object through its life cycle. For example, if a global transaction manager is activated in shared mode, it would be necessary to preserve the transaction contexts in different threads. However if global transaction manager is activated in unshared mode, the same functionality can be obtained with a simpler implementation at the cost of having one process for each active transaction. For certain objects in a multidatabase system, such as object factory, parallel execution is not necessary. These objects serve their clients for a short duration of time and thus do not create a bottleneck in the system. Therefore, there is no need to create a server for each activation which implies shared activation policy.

CORBA handles the heterogeneity at the platform and communication layers by providing an interoperability infrastructure. Thus, a multidatabase system design on CORBA focuses on the upper layers of the system such as schema integration, global query processing and global transaction management. It is clear that this reduces complexity in design and implementation of a multidatabase system dramatically. CORBA provides two more advantages in this respect: Using CORBA as a middleware, a multidatabase system becomes an integrated part of a broad range of distributed object systems that not only contain DBMSs, but also many objects of different kinds such as file systems, spreadsheets, workflow systems and legacy systems. Another advantage comes from the fact that CORBA is a standard. Therefore, the code will be portable among ORB implementations from different vendors. With CORBA 2.0, interoperability among foreign ORBs has become possible.

Using CORBA as the infrastructure affects the upper layers of a multidatabase system since CORBA and COSS together provide basic database functionality to manage distributed objects. The most important database related services included in COSS are the Object Transaction Services, Backup and Recovery Services, Concurrency Services, and the Query Services. If these services are available in the particular ORB that is used, it is possible to develop the global layers of multidatabase system on CORBA mainly by implementing the standard interfaces of these services for the involved objects. For example, by using an Object Transaction Service, implementing a global transaction manager occurs by implementing the interfaces defined in the Object Transaction Service specification for the involved DBMSs. It is also anticipated that the future commercial DBMSs will themselves support these interfaces. Object services that provide database management system functionality are briefly summarized in the following.

Object Transaction Service (OTS) specification describes a service that supports flat and nested transactions in a distributed heterogeneous environment based on the OMG CORBA architecture. OTS defines interfaces that allow multiple, distributed objects to cooperate to provide atomicity of transactions. These interfaces enable the objects to either commit all changes together or to rollback all changes together, in the presence of failures. In order to achieve this goal, OMG has defined some roles such as transactional client, transactional object, recoverable object, and transaction context. In a typical scenario, a transactional client initiates a transaction obtaining a Control object from a Factory provided by ORB. ORB associates a Transaction Context with each Control object. A transaction context contains all the necessary information to control and to coordinate a transaction. Transaction contexts are either explicitly passed as a parameter of the requests, or implicitly propagated by ORB, among the related transactional objects. The Control object is used in obtaining Terminator and Coordinator objects. Transactional client uses the Terminator to abort or to commit the transaction. Coordinator provides an interface for transactional objects to participate in two-phase-commit protocol. Transactional client sends requests to transactional objects. A transactional object is the one that supports transaction primitives as defined by the standard. If a transactional object has a resource that needs to be recovered in case of failures, it is called a recoverable object. Recoverable objects register a Resource object to the ORB using the Coordinator object. ORB interacts with the Resource object to either commit or to abort the changes in the Resource object. Thus ORB provides the atomicity of distributed transactions. It is anticipated that commercial DBMSs will implement the OTS-defined interfaces in the future.

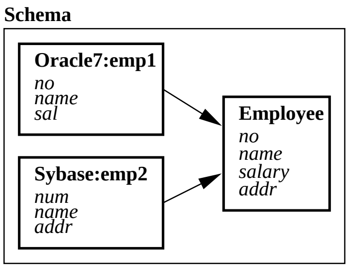
The Concurrency Control service (CCS) enables multiple clients to coordinate their access to shared resources. Coordinating access to a resource means that when multiple, concurrent clients access a single resource, any conflicting actions by the clients are reconciled so that the resource remains in a consistent state. The CCS does not define what a resource is. It is up to the clients of the CCS to define resources and to properly identify potentially conflicting uses of these resources. In a typical use, an object would be a resource and the object implementation would use the CCS to coordinate concurrent access to the object by multiple clients. The Concurrency Control service has been designed as a service that can be used in conjunction with the OMG's Object Transaction Service to coordinate the activities of concurrent transactions. CCS coordinates concurrent use of a resource using locks. The lock modes defined are Intention Read, Read, Upgrade, Intention Write and Write. A client can hold multiple locks on the same resource simultaneously. A collection of locks associated with a single resource is termed as a lock set. Typically, if an object is a resource, the object would internally create and retain a lock set. To manage the release of locks held by a transaction, the CCS defines a lock coordinator. Being able to retain multiple locks on a resource facilitates the use of locks with nested transactions. Thus, a child transaction can acquire a lock on a resource locked by its parent and then drop that lock without causing its parent to lose its lock. This approach is functionally equivalent to "delegation of locks" for nested transactions but supports simpler interfaces.

The Object Query Service (OQS) provides query operation on collections of objects. The queries are predicate-based and may return collections of objects. Although queries may be specified using object derivatives of SQL or other styles of object query languages, in order to provide query interoperability among the widest variety of query systems, an OQS provider must support either SQL Query or OQL. Objects may participate in the OQS in two ways. The simplest involves any CORBA object. The Query Evaluator is then responsible for evaluating the query predicate and performing all query operations by invoking operations directly on that object through its published IDL interfaces. This mechanism provides generality but no optimization. In the second approach, objects participate as members of a collection and the collection supports a specific query interface which passes the query predicate to the collection. Thus, it becomes possible to exploit the internal optimization mechanism of the collection.

Among other database related common object services Object Persistence Service provides a way to ensure persistence of an object regardless of the lifetime of the client applications and of the object implementation. Security Services provides authentication, encryption and audit mechanisms that can be used to provide security in the system. Backup/Restore Service provides a recovery mechanism to restore the objects to their previous states after a system crash or a user error. Finally we note that the basic object services provide much of the functionality of a componentized DBMS, including both Object DBMS and Relational DBMS functionality. However, most of these services are not available in the CORBA compliant products in the market today.

ODL definition

```
interface Employee {
    extent e_emp ;
    key  no ;
    attribute shortno ;
    attribute string < 20 >  name ;
    attribute float salary   ;
    attribute string < 40 >  addr ;
}
 mapping Employee {
    origin Oracle7:emp1 e1, Sybase:emp2 e2  ;
    def_ext  e_emp  as
        select   e1.*, e2.*
            from   e1, e2
            where  e1.no = e2.num ;
    def_att   no  as
        select  e1.no, e2.num
        from  e1, e2  ;
    def_att   salary  as  e1.sal ;
    def_att   addr  as  e2.addr ;
}
```



Global SQL Query

```
select   salary, addr
    from   Employee e
    where   e.no = 10 ;
```

Figure 2: ODL and SQL example

MIND Experience

MIND (METU Interoperable DBMS) [4] is a multidatabase system that is implemented at the Software R&D Center of Middle East Technical University (METU) in cooperation with Laboratory for Database Systems Research of University of Alberta[1]. MIND components are designed as CORBA objects communicating with each other through an ORB. The CORBA implementation used in MIND is DEC's ObjectBroker.

In MIND, local DBMSs to be integrated are encapsulated in a generic database object. MIND defines the interface of a generic database object in CORBA IDL and provides multiple implementations, one for each of the local DBMSs, which are called Local Database Agents (LDA). Currently supported systems include Oracle7[2], Sybase[3], Adabas D[4] and MOOD (METU Object-Oriented Database System). LDA objects are responsible for maintaining export schemas provided by the local DBMSs represented in the canonical data model, translating the queries received in the global query language to the local query language, and providing an interface to the local DBMSs. This layer provides a virtually homogeneous set of database objects. The global layer of MIND, which contains a global transaction manager, a global query processor and a schema integrator, is developed on top of this layer. Global Transaction Manager (GTM) component of MIND is responsible from the management of global, distributed transactions. It keeps track of sub-transactions, handles global commit or global abort using 2PC protocol over LDA objects and detects global deadlocks. In this respect, it acts as a mini TP monitor. At the time MIND was being implemented, OTS was not available and therefore, the distributed transaction mechanism in MIND does not comply with OTS. However, it is quite trivial to replace MIND's built-in transaction mechanism with an OTS implementation. Global Query Manager (GQM) component of MIND is responsible for parsing and decomposing the queries according to the information obtained from Schema Integration Service as well as for optimization of the global queries. After a global query is decomposed, the global sub-queries are sent to the involved LDA objects. MIND query optimization addresses the optimization of post-processing queries that combine results returned by the LDA objects. A dynamic query optimization scheme is developed which benefits from the location transparency provided by the DOM platform. The query optimization is performed at run-time by Query

[1]A prototype of the system is publically available at "ftp://srdc.metu.edu.tr/pub/mind/source/".
[2]Oracle7 is a trademark of Oracle Corp.
[3]Sybase is a trademark of Sybase Corp.
[4]Adabas D is a trademark of Software AG Corp.

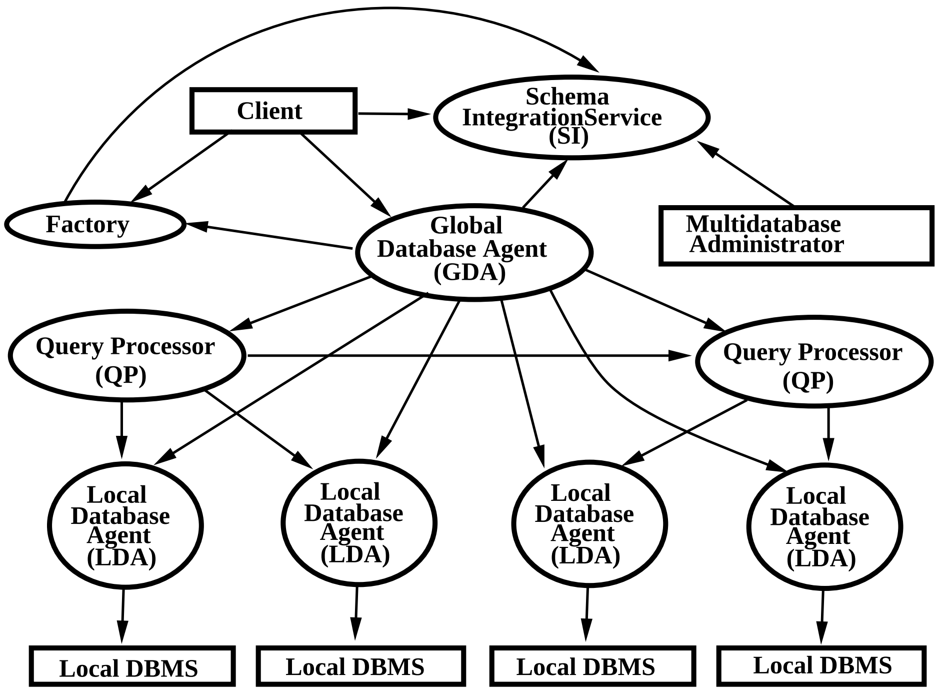

Client
Schema IntegrationService (SI)
Factory
Global Database Agent (GDA)
Multidatabase Administrator
Query Processor (QP)
Query Processor (QP)
Local Database Agent (LDA)
Local Database Agent (LDA)
Local Database Agent (LDA)
Local Database Agent (LDA)
Local DBMS
Local DBMS
Local DBMS
Local DBMS

Figure 3: An Overview of MIND Architecture

Processor (QP) objects. GQM may use as many of QP objects running in parallel as necessary. GQM uses a decision mechanism based on statistical inferencing to find out the execution order of sub-queries [8]. Since this scheme does not need the estimation of execution time of the involved global sub-queries (appearance times), it avoids the uncertainty problems of cost based query optimization schemes.

Schema Integration Service holds the global schema information. The integration of export schemas is currently performed by using an object definition language (ODL) which is based on OMG's interface definition language. The multidatabase administrator (DBA) builds the integrated schema as a view over export schemas. The functionalities of ODL allow selection and restructuring of schema elements from existing local schemas. MIND provides its users a common data model and a single global query language based on SQL. Figure 2 provides a simple example of ODL definition which integrates export schemas from two different DBMSs. A simple global query example is also given in Figure 2.

Figure 3 shows the major components of MIND architecture and their interactions. The direction of the arrows are from a requester to a CORBA object providing the related service. Note that, all the communication is handled by the ORB. A typical MIND client knows only the interface definitions of Global Database Agent, Factory, and Schema Integration Service objects. A client finds the Object Reference of the Factory using Naming Service provided by ORB. Then she requests a GDA object to be created from the Factory. GDA encapsulates the Global Transaction Manager and Global Query Manager in a CORBA object as depicted in Figure 4. The client may also query the names of available resources from the Schema Integration Service whose object reference is available through the Naming Service.

Since the CORBA implementation used supports only C mapping, mapping between CORBA objects and the C++ objects that actually implement the system is not transparent. This makes MIND implementation more complex. Support for C++ mapping would have reduced the complexity significantly.

MIND has a Tcl/Tk based GUI and a simple CGI (Common Gateway Interface) based Web interface. A more sophisticated Web access could be achieved using Java scripts if the CORBA implementation used supported Java mapping.

Our experience with MIND project has shown that CORBA offers a very useful methodology and a middleware to design and implement distributed object systems. To clarify the advantage of using CORBA in a multidatabase implementation, assume that CORBA is not available and we only have a client-server archi-

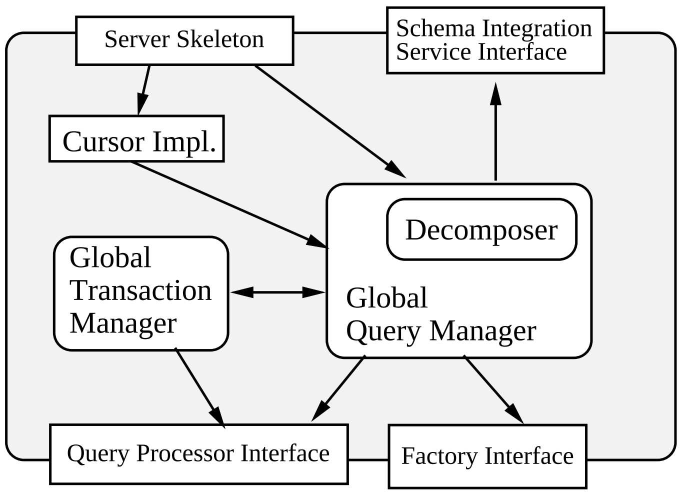

Server Skeleton
Schema Integration Service Interface
Cursor Impl.
Decomposer
Global Transaction Manager
Global Query Manager
Query Processor Interface
Factory Interface

Figure 4: Global Database Agent

tecture and TCP/IP as the infrastructure. In such such a case, implementing a distributed object-oriented architecture would require implementing the ORB-like functionality first. Since this requires tremendous amount of design and coding effort, it is difficult to justify the cost for one specific multidatabase system. A simpler and conventional architecture would be to design and implement local, communication and interoperable layers monolithically. In such a system, the local layer would provide access to different DBMSs. The communication layer would provide all necessary services for remote database access. The interoperable layer would contain global query processor, global transaction manager and a schema integrator. Note that, in such an architecture, the implementor would have to write code to resolve addresses. Although the monolithic design approach may reduce the implementation effort compared to the previous case, it will be difficult to maintain and extent the system. It is possible to move this architecture in the object-oriented direction again by providing the necessary code. Yet, such a system would suffer from interoperability problems with other information systems due to the lack of a standard.

CORBA is becoming a well accepted technology as predicted in [10]. The performance of a middleware, in fact, is dependent on the support from underlying layers such as operating systems and communication networks. The CORBA specification, itself, does not address the issues like the implementation of protocols that will determine scalability. Thus, CORBA compliant ORB implementations should solve these problems within their architectures.

The following information is presented to provide an idea on programmer productivity in developing code using a CORBA compliant ORB. Producing the first prototype of MIND required 24 man-months. Of this, 3 man-months were devoted to acquiring CORBA knowledge through registering the first DBMS to CORBA. Registering other DBMSs is performed in negligible amount of time both due to experience gained, and due to reusable code already generated. The rest of the time is spent in developing the global layer.

References

[1] Brodie, M., Stonebreaker, M. *Migrating Legacy Systems.* Morgan Kaufmann Publishers Inc., 1995.

[2] *The Common Object Request Broker: Architecture and Specification.* OMG Document Number 93.12.43, December 1993.

[3] Dogac, A., Özsu, M.T., Biliris, A. and Sellis T. (editors) *Advances in Object-Oriented Database Systems.* Springer-Verlag, 1994.

[4] Dogac, A., Halici, U., Kilic, E.,, Ozhan, G., Ozcan, F., Nural, S., Dengi, C., Mancuhan, S., Arpinar, B., Koksal, P., Evrendilek, C. METU Interoperable Database System. Demo Description, In *Proc. of ACM Sigmod Intl. Conf. on Management of Data*, Montreal, June 1996.

[5] W. Kim (editor), *Modern Database Systems.* ACM Press/Addison-Wesley, 1995.

[6] Mowbray, T.J., Zahavi, R., *The Essential Distributed Objects Survival Guide*, John Wiley & Sons, 1995.

[7] Nicol, J.R., Wilkes, C.T., Manola, F.A. Object Orientation in Heterogeneous Distributed Computing Systems. *IEEE Computer 26, 6*, (June 1993), 57-77.

[8] Ozcan, F., Nural, S., Koksal, P., Evrendilek, C., Dogac, A. Dynamic Query Optimization on a Distributed Object Management Platform, In *Proc. of Fifth International Conference on Information and Knowledge Management*, Maryland, USA, November 1996.

[9] Özsu, M.T., Dayal, U., Valduriez, P. *Distributed Object Management*, Morgan Kaufmann, 1994.

[10] Pancake, C.M. The Promise and the Cost of Object Technology: A Five Year Forecast. *Commun. ACM, 38, 10*, (Oct. 1995), 33-49.

[11] Sheth A. and Larson, J. Federated Database Systems. *ACM Computing Surveys, 22, 3*, (Sep. 1990), 183-236.

[12] Soley R.M. (ed.), Stone C.M. *Object Management Architecture Guide.* Third Edition, John Wiley & Sons, 1995.

About the Authors

Asuman Dogac is a professor of Computer Engineering and the director of Software Research and Development Center at the Middle East Technical University, Ankara, Turkey.
e-mail: asuman@srdc.metu.edu.tr; http://www.srdc.metu.edu.tr/~asuman/

Cevdet Dengi is the R&D Director of Bilgi Yonetim Sistemleri, A. S., Turkey.
e-mail: dengi@bys.com.tr; http://www.bys.com.tr/~dengi/

M. Tamer Özsu is a professor at the Department of Computing Science and the director of Laboratory for Database Systems Research at the University of Alberta.
e-mail: ozsu@cs.ualberta.ca; http://www.cs.ualberta.ca/~ozsu/

This work is partially being supported by the Middle East Technical University, the Graduate School of Natural and Applied Sciences, Project Number: AFP-97-07-02-08, and by the Scientific and Technical Research Council of Turkey Project No: EEEAG-Yazilim5.