MAKING ORACLE7, SYBASE, AND ADABAS D INTEROPERABLE THROUGH CORBA: MIND PROJECT

Gokhan Ozhan,
Researcher,
Prof. Dr. Asuman Dogac,
Director of Software R and D Center,
Ebru Kilic,
Fatma Ozcan,
Sena Nural,
Cevdet Dengi,
Assoc. Prof. Dr. Ugur Halici,
Budak Arpinar,
Pinar Koksal,
Sema Mancuhan,
Cem Evrendilek,
Researchers,
Software Research and Development Center of TUBITAK
06531, Ankara, Turkiye

Summary

METU Interoperable DBMS (MIND) is a multidatabase system which aims at achieving interoperability among heterogeneous, federated DBMSs. The architecture of MIND is based on OMG[1] distributed object management model. It is implemented on top of a CORBA compliant ORB, namely, ObjectBroker[2]. In MIND all local databases are encapsulated in a generic database object. The interface of the generic database object is defined in CORBA IDL and multiple implementations of this interface, one for each component DBMSs, namely, Oracle7[3], Sybase[4], Adabas D[5] and MOOD (METU Object-Oriented Database System) *(Asuman Dogac, 1994a-b)* are provided. MIND provides its users a common data model and a single global query language based on SQL. The main components of MIND are a global query manager, a global transaction manager, a schema integrator, interfaces to supported database systems and a graphical user interface. The integration of export schemas is currently performed by using an object definition language (ODL) which is based on OMG's interface definition language. It is the responsibility of the DBA to build 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 global query optimizer aims at maximizing the parallel execution of the intersite operations of the global subqueries. Through MIND global transaction manager, the serializable execution of the global transactions (both nested and flat) is provided.

1 Introduction

In today's world, information is typically distributed among multiple database management systems spread over heterogeneous platforms. Therefore there is a need to access and share data across these systems. Heterogeneity in underlying systems makes this integration very difficult if not impossible. The heterogeneity exists at three basic levels. The first is the platform level. Database systems reside on different hardware, use different operating systems and communicate with other systems using different communications protocols. The second level of heterogeneity is the database management system level. Data is managed by a variety of database management systems based on different data models and languages (e.g. file systems, relational database systems, object-oriented database systems etc.). Finally the third level of heterogeneity is that of semantics. Since different databases have been designed independently semantic conflicts are likely to be present. This includes schema conflicts and data conflicts.

Commercially available technology offers inadequate support both for integrated access to multiple databases and for integrating multiple applications into a comprehensive framework. Some products offer dedicated

---

[1] OMG is a registered trademark, and CORBA, ORB, IDL are trademarks of OMG.
[2] ObjectBroker is a trademark of DEC Corp.
[3] Oracle7 is a trademark of Oracle Corp.
[4] Sybase is a trademark of Sybase Corp.
[5] Adabas D is a trademark of Software AG Corp.

gateways to other DBMSs with limited capabilities. Thus, they require a complete change of the organizational structure of existing databases to cope with heterogeneity.

Another way of achieving interoperability among heterogeneous databases is through a multidatabase system. A multidatabase system (MDBS) is a system for the management of several databases. It allows the users to simultaneously access autonomous, heterogeneous databases using a single data model and a query language. The primary objective of a MDBS is to significantly enhance productivity in developing and executing applications that require simultaneous access against multiple independent databases. A multidatabase system provides a single global schema that represents an integration of the relevant portions of the underlying local databases. This in turn requires the support of a single common data model and a single data definition and manipulation language. The users may formulate queries and updates against the global schema.

A recent standard by OMG *(OMG, 1991)*, namely CORBA (The Common Object Request Broker Architecture) provides several advantages when used as the infrastructure of a multidatabase system. CORBA handles the heterogeneity at the platform level and in doing this it provides location and implementation transparency. In other words, the changes in object implementation, or in object relocation has no effect on the client. This reduces the complexity of the client code and allows clients to discover new types of objects added to the system and use them in plug-and-play fashion without any change in the client code. This feature of CORBA is very useful in registering new DBMSs to the system without affecting the already existing system and also this feature dramatically reduces the code that needs to be developed. Furthermore, CORBA and COSS (Common Object Specification Service) together provide much of the functionality to handle heterogeneity at the database level and some functionality to handle application interoperability. Note that COSS *(OMG, 1994)* is a complementary standard developed by the OMG for integrating distributed objects.

The rest of this paper is organized as follows. The architecture of the MIND system is described in Section 2. Section 3 presents the infrastructure of the system. The design decisions and experiences in developing generic Database Object implementations for various DBMSs are also discussed in this section. Section 4 describes the schema integration in MIND. The global query manager of the system is briefly summarized in Section 5. Section 6, describes in detail the creation process of a Local Database Agent. Section 7 provides a description of the transaction management process in MIND. In Section 8, registration process of DBMSs to CORBA is described in detail. Finally, in Section 9 conclusions about the implementation of MIND and future plans for the MIND project is stated.

2 MIND Architecture

In MIND, there is a generic Database Object defined in CORBA IDL and there are multiple implementations of this interface, one for each of the local DBMSs, namely Oracle7, Sybase, Adabas D, MOOD *(Asuman Dogac, 1995a-b).* The current implementation makes unified access possible to any combination of these databases through a global query language based on SQL. When a client application issues a global SQL query to access multiple databases, this global query is decomposed into global subqueries and these subqueries are sent to the ORB (CORBA's Object Request Broker) which transfers them to the relevant database servers on the network. On the server site, the global subquery is executed by using the corresponding Call Level Interface (CLI) routines of the local DBMSs and the result is returned back to the client again by the ORB. The results returned to the client from the related servers are processed if necessary. This approach hides the differences between local databases from the rest of the system. Thus, what the clients of this level see are homogeneous DBMS objects accessible through a common interface. An overall view of MIND is provided in **Figure 1**.

The basic components of MIND are **Global Database Agent** (GDA) and **Local Database Agent** (LDA) class:
A LDA class objects are responsible from:

- 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
- providing an interface to the LDBMSs.

A GDA class objects are responsible from:

- parsing, decomposing, and optimizing the queries according to the information obtained from the Schema Information Manager object,
- global transaction management that ensures serializability of multidatabase transactions without violating the autonomy of local databases.

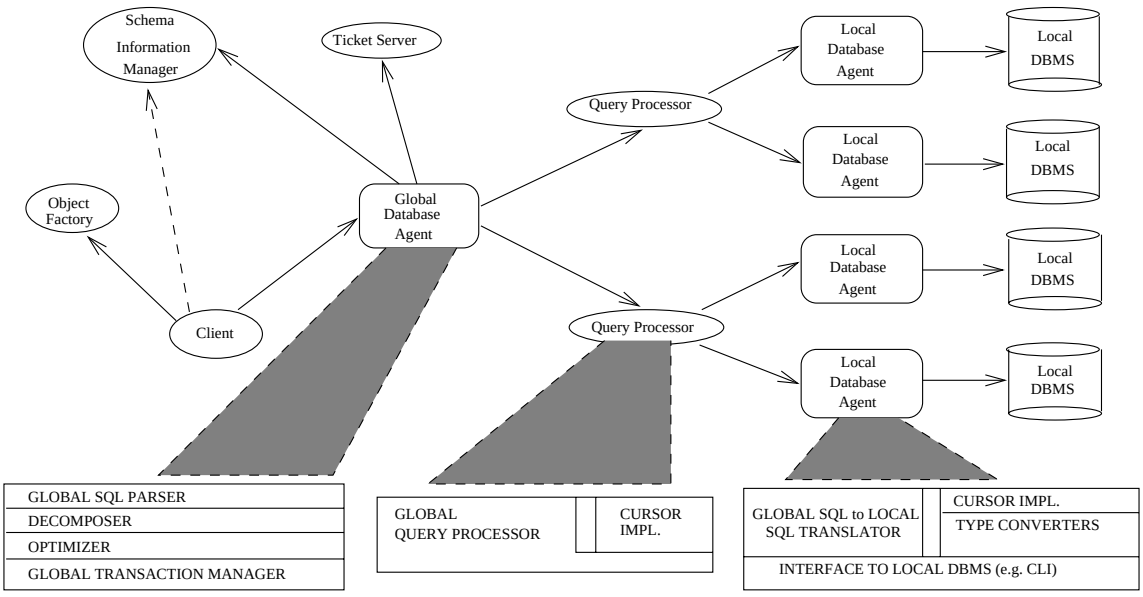

Figure 1: *An Overview of MIND Architecture*

In addition to these, MIND has the following complementary components:

Object Factory Server: This server, namely the *FactoryServer* is responsible from the creation of other MIND objects such as LDA Object (LDAO), Transaction Manager Object (TMO), Query Manager Object (QMO), and Query Processing Object (QPO). It provides the implementation of *DBfactory Interface* which has a single method, namely *CreateObj*. A client who needs an LDAO, TMO, or QPO just calls the CreateObj method of FactoryServer. Once the FactoryServer is started, it creates an object of its own implementation and writes its object reference to the Advertisement Partition of ObjectBroker's Registry. Since the only function of FactoryServer is to create objects, having one FactoryServer does not create a bottleneck in the system.

Ticket Server: This server provides a globally unique, monotonically increasing ticket number at each time it receives a request. These unique ticket numbers are used in the transaction management process of MIND. This server is started at the initialization of MIND system and it continues to serve the whole MIND system continuously.

Schema Information Manager: This server, namely the *SchemaServer* provides and manages the global schema information necessary for the decomposition of global queries into subqueries. It provides the implementation of *SchemaInt Interface* which has methods necessary for the treatment of schema information such as *ExportSchema*, etc. SchemaServer is also started at the initialization phase of MIND system and stays alive during the life-time of the system. Query Manager is the client of this server .

Transaction Manager: This server is started on demand by the ORB and is responsible from the execution and global serializability of both flat and nested transactions. It's infact a part of GDA and is embedded in GDA together with Query Manager.

Query Manager: This server is also started on demand by the ORB and is responsible from the decomposition of global queries into subqueries *(Sena Nural, 1996).* It gets the schema information necessary for the query decomposition from the SchemaServer. In fact, Query Manager is also a part of GDA. Query and transaction management processes are combined within GDA. In addition to the decomposition of global queries, Query Manager also performs query optimization *(Fatma Ozcan, 1996).*

Query Processor: This server is responsible for query processing. It minimizes the total query processing time by enabling parallel execution of subqueries. It performs the necessary operations (such as join, outer-join, and union) for processing partial results coming from the local DBMSs in order to get the final result of the global query. A Query Processors is started by the ORB as a result of a request from the Query Manager indicating that two partial results to be processed together are ready at the LDAs. Query Processor provides the implementation of *QPSrvc Interface* which has methods such as *ActivateQP* and *GetResult* for query processing.

3 The Infrastructure Of MIND

Oracle7, Sybase, Adabas D, and MOOD DBMSs were encapsulated in multiple implementations of a generic Database Object as the initial step of the implementation of MIND *(Asuman Dogac, 1995c).* The Database Object conveys requests from the client to the underlying DBMSs by using the CLIs of these DBMSs *(ORACLE,*

1992), (SYBASE 1990), (SOFTWARE AG, 1993). The CLIs of these systems support SQL data definition, data manipulation, query, and transaction control facilities. We have used C bindings of these CLIs to access the corresponding database servers. Results of the requests returned from the CLIs of underlying DBMSs are conveyed to the client through CORBA. Note that the clients of LDBMSs are LDAOs.

Our basic implementation decisions in registering different databases to CORBA are as follows:

Object granularity: In CORBA, objects can be defined in any granularity. In registering a DBMS to CORBA, an object can be a row in a relation or it can be a database itself. When fine granularity objects, like tables are registered, all the DBMS functionalities to process these tables, like querying, transactional control, etc., must be supported by the multidatabase system itself. However, when a relational DBMS, for example, is registered as an object, all the DBMS functionality needed to process these tables are left to the DBMS. Another disadvantage of registering fine granularity objects is the following: with each insertion and deletion of these classes, it is necessary to recompile the IDL code and rebuild the server if the dynamic server/skeleton interface is not supported by the CORBA vendor. Furthermore, the IDL code will be voluminous. For these reasons, we have defined each DBMS as an object.

Invocation style: CORBA allows clients to do both *dynamic* and *stub-style* invocation of the interfaces. In stub-style interface invocation, the client uses generated code templates (stubs) that cannot be changed at run time. In dynamic invocation, the client defines and builds requests as it runs.

We have chosen to use mostly the stub-style interface invocation because in our current implementation all the objects are known and the interface that the clients use is not likely to change over time. However, for certain operations in MIND (e.g., SendQuery method of LDA) *deferred synchronous* mode is necessary. In DEC's ObjectBroker deferred synchronous mode is supported only in dynamic interface invocation. Therefore, for such cases dynamic invocation is used.

Mapping client requests to servers: In associating a client request with a server method, CORBA provides the following alternatives:

- One interface to one implementation,
- One interface to one of many implementations,
- One interface to multiple implementations.

In the second alternative, for any one interface, it is possible to have several different implementations. There is a direct one-to-one relationship between the interface operations and the methods in the implementations. Since every database management system registered to CORBA provides methods for all of the operations that the interface definition specifies, the second alternative is sufficient for our purposes.

Object Life Cycle: A client application needs an object reference to make a request. ObjectBroker allows servers to store initial object references in the Advertisement Partition of Registry which corresponds to naming service of COSS. Then clients can access the Advertisement Partition to obtain object references.

The Advertisement Partition of ObjectBroker's Registry contains the object references of the following three objects: FactoryServer object, TicketServer object and the SchemaServer object. Since these objects serve the whole MIND system continuously, they are not created on demand and are always active.

Activation Policy: When registering different databases to CORBA, one has 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 one of the following activation policies :

Shared : The server can support more than one object for the implementation.
Unshared : The server can support only one object at a time for the implementation.
Server_per_method : A new server is used for each method invocation.
Persistent : The server is never started automatically. Once started, it is the same as the shared policy.

We have used the shared activation policy for activating FactoryServer, TicketServer, and the SchemaServer object. There is one TicketServer object in MIND which provides a globally unique, monotonically increasing ticket number at its each invocation. The activation of SchemaServer object is also shared, since it serves its clients for a short duration of time, there is no need to create a server for its each activation. Similarly, since one FactoryServer server is enough to meet the object creation demands within the system, its activation policy is also shared.

All the other objects in the MIND system are activated in the unshared mode. If GTMO were activated in the shared mode, it would be necessary to preserve the transaction contexts in different threads. Therefore, GTMO is activated in the unshared mode to obtain the same functionality with a simple implementation. GQMO is activated in the unshared mode for the same reason. Query Processor Objects (QPO) are activated in the unshared mode to provide parallelism in query execution. Each implementation is responsible for only one QPO and thus when a new QPO is activated a new QPO implementation is executed by the ORB dedicated to that object. In this way, QPOs accomplish their jobs in parallel without waiting for one another. If shared policy were used, one QP implementation would be responsible for more than one QPO. And these QPOs would have to accomplish their jobs sequentially using the same QP implementation.

LDAOs are activated in the unshared mode to be able to access the LDBMSs in a multiuser environment. Note that, if a LDAO were activated in the shared mode, the server created for this LDAO would not serve another user until it has completed its service with the current user. Such a scheme would reduce the LDBMS to a single user system.

MIND is an evolving user-friendly system (versions V.0.1, V.0.2 *(Ebru Kilic, 1995)*, V.1.0, and V.2.0 have been produced upto now). In **Figure 2**, a screen snapshot of MIND graphical user interface is provided As the system evolves, it provides clearer insights regarding the nature of issues in implementing a multidatabase system on a distributed object management architecture.

4 Schema Integration in MIND

MIND implements a four-level schema architecture that addresses the requirements of dealing with distribution, autonomy and heterogeneity in a multidatabase system. This schema architecture includes four different kinds of schemas:

- Local Schema: A local schema is the schema managed by the local database management system. A local schema is expressed in the native data model of the local database and hence different local schemas may be expressed in different data models.
- Export Schema: An export schema is derived by translating local schemas into a canonical data model. The process of schema translation from a local schema to an export schema generates mappings between the local schema objects and the export schema objects.
- Derived (Federated) Schema: A derived schema combines the independent export schemas to a (set of) integrated schema(s). A federated schema also includes the information on data distribution (mappings) that is generated when integrating export schemas. The global query manager transforms commands on the federated schema into a set of commands on one or more export schemas.
- External Schema: In addition, it should be possible to store additional information that is not derived from export databases. An external schema defines a schema for a user or an application. An external schema can be used to specify a subset of information in a federated schema that is relevant to the users of the external schema. Additional integrity constraints can also be specified in the external schema.

The classes in export and derived schemas behave like ordinary object classes. They consist of an interface and an implementation. But unlike ordinary classes, which store their objects directly, the implementations of the classes in these schemas derive their objects from the objects of other classes. In MIND, LDAOs translate the local schemas into the export schemas using ODL, and then their export schemas are stored in the server, SchemaInformationManager. SchemaInformationManager integrates these export schemas to generate a global schema. Schema integration is a two phase process:

- Investigation phase: First commonalities and discrepancies among the export schemas are determined. This phase is manual. That is, the DBA examines export schemas and defines the applicable set of inter-schema correspondences. The basic idea is to evaluate some degree of similarity between two or more descriptions, mainly based on matching names, structures and constraints. The identified correspondences are prompted according to the classification of schema conflicts.
- Integration phase: The integrated schema is built according to the inter-schema correspondences. The integration phase cannot be fully automated. Interaction with the DBA is required to solve conflicts among export schemas. In MIND, the integration of export schemas is currently performed by using an object definition language (ODL) which is based on OMG's interface definition language *(Asuman Dogac,1996).* The 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.

After the global schema is obtained SchemaServer provides the necessary information to the GQMO on demand. We have developed a graphical tool which will automatically generate textual specification of the global

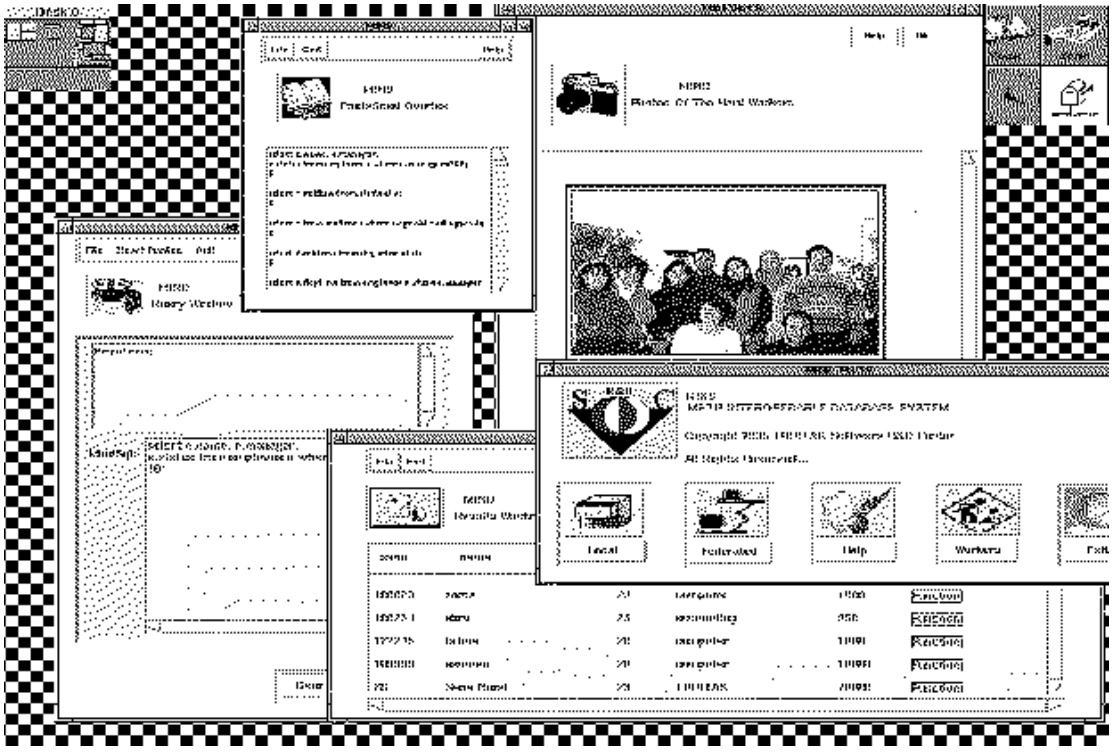

Figure 2: *MIND GUI Screen snaphots*

schema. Our ultimate aim is to establish a semi-automated technique for deriving an integrated schema from a set of assertions that state the inter-schema correspondences. Theassertions will be derived as a result of the investigation phase. For each type of assertion, there will correspond an integration rule so that the system knows what to do to build the integrated schema.

5 Query Processing In MIND

When a user wants to interact with MIND, a Query Manager Object (QMO) is created by sending *CreateObj* message to MIND's FactoryServer. These QMOs are created for each session of a client. QM obtains the global schema information necessary for the decompositon of the query from the SchemaServer. After decomposing the client's global query into subqueries, QM sends the subqueries to LDAs. LDAs then submit the incoming local subqueries to LDBMSs (Local Database Management Systems) so that they are executed in parallel.

Note that some operations such as outer join, join, or union are required for processing the partial results coming from LDBMSs in order to get the final result of the global query. These operations are processed by another object class called Query Processor in order to minimize the total query processing time by enabling parallelism. As soon as two partial results that can be processed together appear at the LDAs, Query Processor Object (QPO) is created to process them. There could be as many QPO's running in parallel as needed to process the partial results. This provides a highly dynamic schema with maximum parallelism in the system.

6 Local Database Agents In MIND

Local Database Agents (LDA) support the implementation of *DBservices Interface* which has methods for the execution subqueries on LDBMSs. Each LDA is in fact a connection between a client and the LDBMS server. It also stores login and cursor data structures in C++ objects which serve as the conduit for information between the client and the LDBMS.

QM invokes the *SendQuery* method of DBservices Interface for the execution of local subqueries. SendQuery method returns a success status upon the correct execution of the subquery. Whenever two SendQuery method calls for the subqueries whose results should be processed together are completed successfully, QM sends a request to FactoryServer for the creation of a QPO to process the partial results. QP is activated automatically by the ORB at the first time a request for processing the partial results is invoked by QM. QM passes the object identifiers of LDAOs (which have executed the local queries) and operation (such as join, outer-join, or union)

parameters to QP. QP invokes the *GetResult* method to LDAOs (whose identifiers are provided by QM as input parameters to QP) to get the partial results to be processed.

Such a complicated interaction among MIND system objects (e.g. QPO, QMO, and LDAO) requires a client-server architecture with the following properties:

- Each LDA responds to one and only one object's requests. After the execution of the query, the results are fetched using the cursor data structures.
- Each LDA must be capable of providing service to different clients which make requests to LDA's unique object. For example, a QM, as a client, may send request to an LDAO for the execution of a subquery, whereas, a QP, as another client, may send request to the same LDAO for retrieving the results of that query.

As it is mentioned in Section 3, ObjectBroker Version 1.2 supports four kinds of activation policies, namely shared, unshared, server-per-method and persistent. In addition to these activation policies, it provides *registration attributes* in order to specify different kinds of implementation servers. The activation policy of an implementation definition specifies what policy the server supports when dealing with objects of that implementation. When the activation policy is specified to be *shared*, the server can give service to multiple objects of a single implementation.

If the activation policy is specified to be *unshared* the server can support only one object of a single implementation at a time. It's possible for unshared servers to be started automatically by the ORB. The first time a request is invoked to an unshared implementation object, the ORB selects an active server with no active objects on it. If there is no such server, ORB starts one. After the first method invocation the object uses the same server. But once an unshared server is started by the ORB as a result of a request sent to an object by a client, that server does not respond to any requests coming from the other clients even if the requests are sent to the same object. This is because, ObjectBroker reserves such an unshared server to the client whose request caused the activation of it. So, the unshared server does not respond to any other client's requests.

We have chosen to use unshared activation policy for QM and LDA. This makes both type of servers to be responsible from one and only one object. Each QM is responsible for one end user's query requests. The user sends a transaction request to a QMO, gets the resulting output, if any, and then uses the same object to request for another transaction. Clients are identified by the identifiers of CORBA objects to which they send message. So that, no one's session context (e.g. its login information on different databases, transaction management and query processing information, etc) interfere to one another.

Since an unshared server does not respond to clients other than it is bounded to, we had to enrich the architecture of an unshared server, so that it responds to all client requests provided that they have been sent to the server's bounded object. Such kinds of unshared implementation servers are called *hidden* ObjectBroker terminology. If a server supports a hidden implementation it can only give service to the objects that are bounded to that server. Each client sending request to the bounded object of the implementation of a server can get service without problem. An object can be bounded to a server if and only if it's created by the server itself. Because of this, the objects of DBservices Implementations must be created by its own LDA, not by the FactoryServer. In MIND, all of the CORBA objects (e.g., QMO, QPO, LDAO, etc.) are created by the FactoryServer in as a standard. In order not to violate this standard, we have done the following:

We have defined another interface, namely *LDAFactory*, which has the method *CrLDA*. Thus LDA servers support two implementations, one for LDAFactory, another for DBservices interfaces. The CrLDA method is used to create DBservices object of LDA. When a request comes to FactoryServer for the creation of an LDAO, it first creates an object of LDAFactory implementation. Then, it invokes the CrLDA method of this implementation. This invocation makes ORB to start an LDA server. The activated LDA server executes the implementation of CrLDA method. CrLDA method creates an object of DBservices implementation which is specified to be *hidden* and *unshared*. Since the object creation is done on the LDA, this object is bounded to that LDA. CrLDA method returns the identifier of the bounded object. Any client making a request to that object will be served by the LDA bounded to that object. After the creation of DBservices implementation object, LDAFactory implementation object is discarded. All of these operations are transparent to the client. What the client does is to send a request to FactoryServer for the creation of an LDAO and get an object identifier as a result (which is infact the identifier of the DBservices implementation object). This object creation process is illustrated in **Figure 3**.

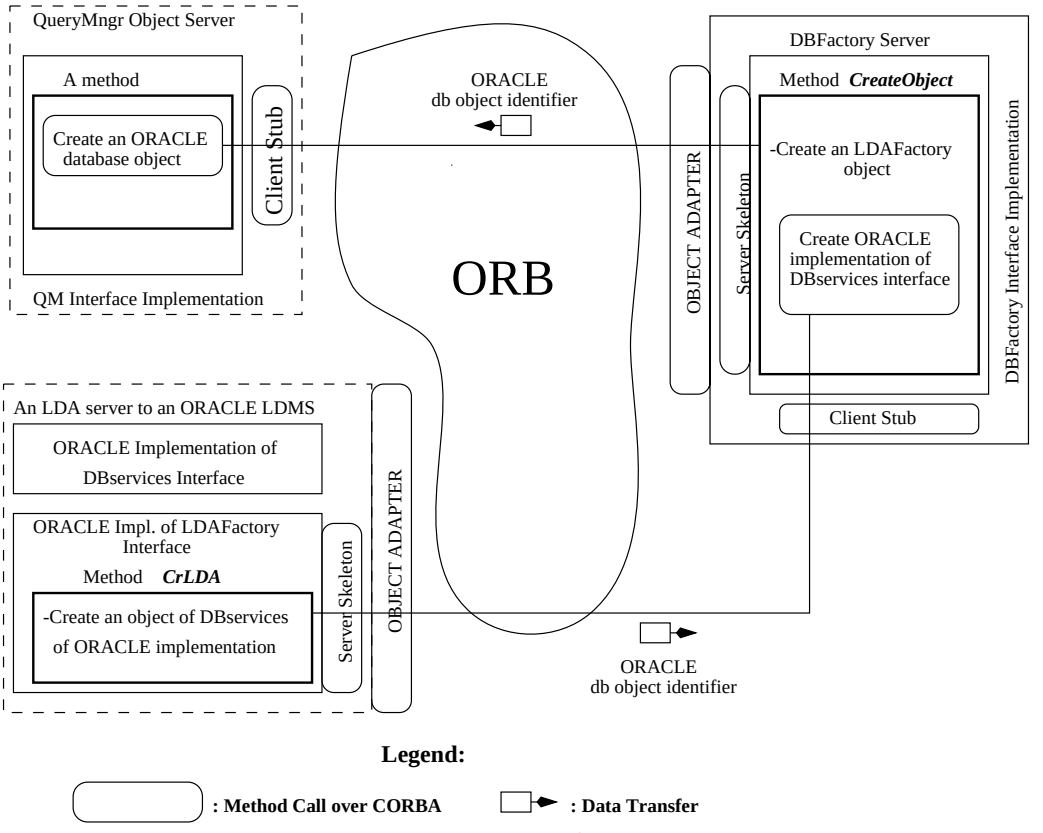

Figure 3: *Creation of database object*

7 Transaction Management In MIND

In MIND V.0.2, a ticket based algorithm *(D. Georgakopoulos, 1994)* has been implemented for flat transactions *(Asuman Dogac, 1995d).* For MIND V.1.0 a technique for global concurrency control of nested transactions in multidatabases, called Nested Tickets Method for Nested Transactions (NTNT) is developed *(Ugur Halici, 1995).* It should be noted that the concurrency control techniques developed for flat multidatabase transactions do not provide the correctness of nested transactions in multidatabases because for nested transactions a consistent order of global transactions is not enough; the execution order of siblings at all levels must also be consistent at all sites.

NTNT ensures global serializability of nested and flat multidatabase transactions without violating autonomy of LDBMSs. The main idea of NTNT technique is to give tickets to global transactions at all levels, that is, both the parent and the child transactions obtain tickets. Then each global (sub)transaction is forced into conflict with its siblings through its parent's ticket at all related sites. The recursive nature of the algorithm makes it possible to handle the correctness of different transaction levels smoothly.

Among the DBMSs incorporated to MIND, only Sybase supports nested transactions. Therefore, the parts of a global transaction submitted to Sybase servers can be nested transactions, the others must be flat transactions.

Even if a LDBMS does not support nested transactions, their effect with respect to the hierarchical domains of recovery can be simulated by using savepoints. The idea is simply that whenever a new subtransaction is started, a savepoint is generated and then rolling back to that save point has the same effect as an abort of the corresponding subtransaction *(J. Gray, 1993).* We are planning to implement nested transaction over ORACLE and ADABAS D as a future work using this technique.

8 Registration of DBMSs to CORBA

In order to register different DBMS to CORBA we used the Call Level Interfaces (CLI) of the corresponding DBMSs. Most of the commercial DBMSs provide CLIs in order to allow users to develop applications that take advantage of the non-procedural capabilities of SQL and the procedural capabilities of a third generation language. As it is mentioned in Section 3, we have registered Oracle7, Sybase, Adabas D, and MOOD DBMSs

to CORBA as multiple implementations of a generic Database Interface. In this Database Interface, we provide primitive methods necessary for data definition, data manipulation, query execution, and transaction control facilities. These methods have different implementations for each DBMS. In MIND, we have used C bindings of the CLIs to access the corresponding database servers. For example, in order to register Oracle7 to CORBA we used the ORACLE Call Interface functions for the programming language C. In the following sections, registration of Oracle7 to CORBA is described in detail.

8.1 ORACLE Call Interface

The ORACLE Call Interfaces (OCI) are a set of application programming interfaces that allow a user to manipulate data and schema in an ORACLE database. OCI supports all SQL data definition (e.g., CreateTable, DropTable, etc.), data manipulation (e.g., insert, update, etc.), query (i.e., select statement), and transaction control (e.g., commit, rollback, etc.) facilities that are available through the Oracle7 Server. A user can also take the advantage of PL/SQL, Oracle's procedural extensions to SQL, in an OCI application. When coding an application using the ORACLE Call Interfaces, there are certain steps that must be followed to insure that the program works properly. At least, the OCI application should include the following steps:

- allocate data structures that allow the connection to Oracle and process cursors,
- connect to one or more Oracle databases,
- open one or more cursors as needed by the program,
- process the SQL or PL/SQL statements required to perform the application's tasks,
- close the cursors,
- disconnect from the databases.

8.2 Common Interface For Registering DBMSs To CORBA

We have defined a virtual C++ class, namely *DatabaseImpl* which serves as a base class for the C++ classes defined for each DBMS. For example, C++ class defined for Oracle7, namely *OraImpl* inherits DatabaseImpl class. DatabaseImpl class has the following methods:

ConnectToDB: This method gets username and password as input arguments and builds a connection to the related DBMS. Username and password are retrieved from the SchemaServer.

DatabaseSendQuery: This method gets the query to be executed as an input argument, executes the query on the related DBMS and returns a status value as an output argument indicating whether the execution completed successfully or not.

GetNext: This method retrieves the results of the query executed by DatabaseSendQuery method in an output argument whose type is *DBresult* structure. DBresult structure is made up of two main parts. The first part, namely *description-part* is a structure for storing information such as number of columns selected, type of each column, maximum display size of each column, number of rows returned, etc. The second part, namely *data-part* is a structure of CORBA sequence type which holds the rows fetched in an one dimensional linear array. The maximum number of rows that can be returned by a single call of GetNext method is defined as a constant in IDL. Only a single call of GetNext method may not be enough to retrieve all of the resulting rows of a query. In order to avoid such a case, GetNext method should be called in a loop until the value of its status argument is returned as equal to NO_MORE_ROWS constant. Since the maximum number of rows that is returned by GetNext method is defined as a constant, it can easily be changed in order to optimize the network delay time.

BeginTrans: This method starts a new transaction on the related DBMS and returns a valid transaction identifier (TID) if CLI of the DBMS provides such a facility. Since Oracle Call Interface does not provide transaction identifiers, this method does not return a valid TID for Oracle.

PrepareToCommit: This method sends a PrepareToCommit message to the related DBMSs which provide such a facility in their CLIs. Since OCI does not provide facilities for 2 Phase Commit (2PC) protocol, this method is useless for Oracle.

CommitTrans: This method is responsible for committing the transaction in the related DBMS. It gets the identifier of the transaction to be commited as an input argument if it is needed by DBMS. OCI does not use transaction identifiers for committing the transactions so this argument is not used for Oracle.

AbortTrans: This method is responsible for aborting the transaction in the related DBMS. Just like the CommitTrans method, it gets the identifier of the transaction to be aborted. This argument is still useless for Oracle since OCI does not use TIDs for aborting the transactions.

CheckNestedTic: This method is responsible for the execution and management of both flat and nested transactions. For the implementation of nested transactions, we have developed a technique called Nested Tickets Method for Nested Transactions (NTNT) that provides for the correct execution of nested transactions in multidatabases *(Ugur Halici, 1995).*

8.3 Registration Of ORACLE To CORBA

As it is mentioned in Section 8.2, for each DBMS a C++ class is defined which inherits the virtual base class DatabaseImpl. OraImpl is the name of the C++ class which is defined for Oracle. In addition to the methods that are inherited from DatabaseImpl base class, it has some specific methods of its own, namely *describe_define*, *copy_describe*, *fetch_rows*, *oci_error*, and *do_exit*. These methods will be explained in more detail in this section in addition to the design decision made for the registration of Oracle to CORBA.

In Version 6 and earlier versions of ORACLE, each call to an OCI routine required a corresponding call to the RDBMS (Relational Database Management System) server. For example, when you parsed a SQL statement, the text of the SQL statement was transmitted to the RDBMS, and the statement was parsed and stored in the user's global area. A query required additional calls to the RDBMS to define the address of program variables to hold the select-list items. Finally, when the statement was executed another call was made to execute the statement and return the results. When the OCI application and the Oracle database are running on the same machine, multiple calls to ORACLE have only a slight impact on performance. In a networked client/server environment, in which OCI program and Oracle server are running on separate machines as it is in our case, multiple calls to the database server can result in decreased performance. In order to enhance the performance of MIND system, we used a new feature of Oracle7, namely *deferred execution*. Using this facility you can defer the processing of the step that parses the SQL statement, define output variables, until the statement is actually executed.

In an OCI program the following steps are necessary for the execution of a query:

- Connect to Oracle: An OCI program establishes communication with one or more Oracle databases by calling the *orlon* routine. ConnectToDB method calls this routine by providing username and password as input arguments.
- Open the Cursors: To process a SQL statement you must have an open cursor. A cursor data area provides a mapping between the user cursor in your program and the parsed representation of the SQL statement in Oracle. For openning cursors *oopen* routine is used.
- Parse the Statement: Every SQL statement must be parsed, using the *oparse* routine. Parsing the statement associates it with the cursor in your program. This routine has a flag called *defflg* for enabling deferred execution. In MIND, this flag is set to a non-zero value in order to use the deferred execution facility.
- Bind the Addresses of Input Variables: This step is necessary for interactive OCI programs in order to get input data from the user at run time. In MIND, we do not need such a facility because when a MIND user enters a global query, Query Manager in cooperation with the SchemaServer decomposes it into subqueries that can be directly executed by Oracle without the interaction with the user.
- Describe Select-List Items: If the SQL statement is a query, it is necessary to obtain information about the datatypes, column lengths, or display sizes of the select-list items. This information can be obtained using the *odescr* routine. In MIND, we have defined a constant indicating the maximum number of items that can be selected by a query in order to reduce the programming cost of processing dynamic select-lists.
- Define Select-List Items: For the execution of a query, address of the output variables in the OCI program must be associated with each select-list item in the query. For this purpose *odefin* routine is used. Description and definition of select-list items are performed by a member function of OraImpl class called *describe_define*.
- Execute the Statement: If the SQL statement is a data manipulation language statement (e.g., select, insert, update, delete, etc.) it must be executed using the *oexec* routine. Data definition language statements (e.g., CreateTable, DropTable, etc.) can be executed in the parse phase using the oparse routine if the deferred execution option is not used. Since we use the deferred execution facility, we have to call oexec routine in either case. Oracle implementation of the SendQuery method performs the parsing, description and definition of select-list items, and the execution of a query by calling the corresponding OCI routines and member functions of OraImpl class.
- Fetch the Rows for the Query: In order to fetch the rows resulting from the execution of a query *ofetch* routine is called. This routine is called by a member function of OraImpl class called *fetch_rows*. This function stores the columns of a fecthed row into the data-part of the DBresults structure. Oracle implementation of GetNext method calls this function in a loop until it fills up its buffer or the function returns a value equal to NO_MORE_ROWS flag. Prior to calling the fetch_rows function, GetNext method calls another member function of OraImpl class called *copy_describe* which stores the description data of the select-list items into the description-part of the DBresult structure.
- Close the Cursors: Before closing the connection each open cursor must be closed using the *oclose* routine.

• Disconnect From Oracle: For closing the connections to Oracle *ologof* routine is called. Another member function of OraImpl class, namely *do_exit* closes the open cursors and the connections to Oracle using the oclose and ologof routines respectively.

In MIND, error messages are transmitted to the client using the exception handling mechanism of CORBA. You can define user-exceptions in addition to the system exceptions of CORBA. We have defined a user-exception in IDL which consists of two parts. The first part is a flag indicating the name of MIND's component in which the error occured (e.g., ORACLE_ERROR, SYBASE_ERROR, etc.). The second part is a string which holds the description message of the error. For example, when an error occurs in an LDA for Oracle, first we set the user-exception flag to ORACLE_ERROR, then we fill in the user-exception message using a member function of OraImpl class called *oci_error*. This function uses the *oerhms* routine of OCI which returns the text of an Oracle error message of the given error code.

Call Interface of Oracle7 does not provide 2 Phase Commit (2PC) protocol routines. Neither it provides unique transaction identifiers (TID). If 2PC routines were supported by OCI, Oracle implementation of BeginTrans method would start a new transaction on Oracle database and return its unique identifier to the client. Unfortunately, BeginTrans method is useless for Oracle since 2PC routines are not available. PrepareToCommit method has no function in Oracle for the same reason. Oracle implementation of CommitTrans method uses the *ocom* routine of OCI in order to commit the current transaction. In contrast, Oracle implementation of AborTrans method uses the *orol* routine of OCI to abort the current transaction. In OCI terminology, current transaction is defined as the set of SQL statements executed since the *orlon* call or the last *ocom* or *orol* call. In other words, current transaction starts when you establish a connection with Oracle database and lasts until you commit or rollback the transaction using CommitTrans or AbortTrans method respectively.

In the next version of MIND, we are planning to overcome the problems due to the restricted transaction facilities of OCI by using *X/Open DTP*[6] (distributed transaction processing). An X/Open application is an application that operates in a distributed transaction processing environment. In an abstract model, applications call on *resource managers* to provide many different kinds of services. A database resource manager offers access to data in a database for an application. X/Open has a *transaction manager* component that registers transactions and manages their atomic commitment. Transaction processing monitors that use the standard X/Open interface to resource managers incorporate the transaction manager for transaction coordination. Oracle7 provides an interface called *XA Interface* which is specified in X/Open model. An XA-compliant library is provided by Oracle to be linked into the OCI applications. Using this library, we will modify Oracle implementations of MIND's transaction methods (e.g., BeginTrans, PrepareToCommit, CommitTrans, and AbortTrans) in order to support 2PC protocol.

In the current version of MIND, nested transactions are supported only if the underlying DBMS supports nested transactions. Since Oracle7 does not support nested transactions, a MIND user cannot send a query which contains nested transaction statements to Oracle. As it is mentioned in Section 7, we are planning to implement nested transactions on Oracle using the savepoints in the near future

9 Conclusion and Future Work

In this paper, we described our experiences in the implementation of a multidatabase system over CORBA. METU INteroperable Database System (MIND) is a multidatabase system that aims at achieving interoperability among heterogeneous, federated DBMSs. It has been implemented on top of DEC's CORBA, namely Object Broker. The basic component objects of MIND are Global Database Agents (GDA) and Local Database Agent (LDA). In MIND all local databases are encapsulated in a generic database object. The interface of the generic database object is defined in CORBA IDL and multiple implementations of this interface, one for each component DBMSs, namely, Oracle7, Sybase, Adabas D and MOOD are provided. All of the component objects (e.g., QMO, QPO, LDAO, etc.) of MIND are created by the FactoryServer as a standard.

Since CORBA handles heterogenity at the platform and communication layers, MIND development is focused on the upper layers of the system such as schema integration, global query execution and global transaction management, which reduced the required development effort dramatically. Our experience have indicated that CORBA offers a very useful methodology and a middleware to design and implement distributed object systems. Furthermore, using CORBA as a middleware made it possible for MIND to become an integral part of a broad

[6] X/Open DTP is a registered trademark of X/Open Company, Ltd.

distributed object system that not only contains DBMSs but may also include many objects of different kinds such as file systems, spreadsheets, workflow systems, etc.

As a future work, we are planning to implement nested transactions over ORACLE and ADABAS D using the technique briefly described in Section 7. Also in the next version of MIND, XA interface of Oracle will be used in order to overcome the problems due to the restricted transaction facilities of OCI. By the use of this interface, it will be possible to the manage the transactions sent to Oracle according to the 2 Phase Commit protocol.

BIBLIOGRAPHY

(Asuman Dogac, 1994a) Dogac, A., Evrendilek, C., Okay, T., Ozkan, C., "METU Object-Oriented DBMS", Advances in Object-Oriented Database Systems, edited by Dogac, A., Ozsu, T., Biliris, A., Sellis, T.,Springer-Verlag, 1994.

(Asuman Dogac, 1994b) Dogac, A., et. al., "METU Object-Oriented Database System", Demo Description, in the Proc. ACM SIGMOD Intl. Conf. on Management of Data, Minneapolis, May 1994.

(Asuman Dogac, 1995a) Dogac, A., Altinel, M., Ozkan, C., Durusoy, I., "Implementation Aspects of an Object-Oriented DBMS", in ACM SIGMOD Record, Vol.24, No.1, March 1995.

(Asuman Dogac, 1995b) Dogac, A., Altinel, A., Ozkan, C., Durusoy, I., Altintas, I., "METU Object-Oriented DBMS Kernel", in Proc. of Intl. Conf on Database and Expert Systems Applications, London, September 1995 (Lecture Notes in Computer Science, Springer-Verlag, 1995).

(Asuman Dogac, 1995c) Dogac, A., Kilic, E., Ozhan, G., Dengi, C., Kesim, N., Koksal, P., "Experiences in Using CORBA for a Multidatabase Implementation", 6th International Conference on Database and Expert Systems Applications Workshop presentation, London, September 1995.

(Asuman Dogac, 1995d) Dogac.,A., Dengi, C., Kilic, E., Ozhan, G., Ozcan, F., Nural, S., Evrendilek, C., Halici, U., Arpinar, B., Koksal, P., Kesim, N., Mancuhan, S., "METU Interoperable Database System", in ACM SIGMOD Record, Vol.24, No.3, September 1995.

(Asuman Dogac, 1996) Dogac.,A., Dengi, C., Kilic, E., Ozhan, G., Ozcan, F., Nural, S., Evrendilek, C., Halici, U., Arpinar, B., Koksal, P., Kesim, N., Mancuhan, S., "A Multidatabase System Implementation on CORBA", 6th Intl. Workshop on Research Issues in Data Engineering (RIDE-NDS '96), New Orleans, February 1996.

(D. Georgakopoulos, 1994) Georgakopoulos, D., Rusinkiewicz, M., and Sheth, A., "Using Tickets to Enforce the Serializability of Multidatabase Transactions", IEEE Trans. on Data and Knowledge Eng., Vol. 6, No.1, 1994.

(Ebru Kilic, 1995) Kilic, E., Ozhan, G., Dengi, C., Kesim, N., Koksal, P. and Dogac, A., "Experiences in Using CORBA in a Multidatabase Implementation", in Proc. of 6th Intl. Workshop on Database and Expert System Applications, London, Sept. 1995.

(Fatma Ozcan, 1996) F. Ozcan, "Dynamic Query Optimization on a Distributed Object Management Platform", Ms. Thesis, Dept. of Computer Engineering, METU, February 1996.

(J. Gray, 1993) J. Gray, and A. Reuter. Transaction Processing: Concepts and Techniques. Morgan Kuafmann, 1993.

(OMG, 1991) Object Management Group, "The Common Object Request Broker: Architecture and Specification", OMG Document Number 91.12.1, December 1991.

(OMG, 1994) Object Management Group, "The Common Object Services Specification, Volume 1", OMG Document Number 94.1.1, January 1994.

(ORACLE, 1992) Programmer's Guide to the Oracle Call Interfaces, Oracle Corporation, December 1992.

(Sena Nural, 1996) S. Nural, P. Koksal, F. Ozcan, A. Dogac, "Query Decomposition and Processing in Multidatabase Systems", in proc. of the Intl. Conf. on Eng. Systems Design & Analysis, Montpellier, July 1996.

(SOFTWARE AG, 1993) ENTIRE SQL-DB Server Call Interface, ESD 311-316, Software AG, April 1993.

(SYBASE, 1990) Open Client DB-Library/C Reference Manual, Sybase Inc., November 1990.

(Ugur Halici, 1995) Halici, U., Arpinar, B., and Dogac, A., "Serializability of Nested Transactions in Multidatabases", Technical report 95-10, Software R&D Center, Middle East Technical University, October 1995, (submitted to Sigmod Intl. Conf. on Management of Data '96).