A Multidatabase System Implementation on CORBA*

A. Dogac, C. Dengi, E. Kilic, G. Ozhan, F. Ozcan, S. Nural, C. Evrendilek,
U. Halici, B. Arpinar, P. Koksal, S. Mancuhan
Software Research and Development Center of TUBITAK
Middle East Technical University (METU), Turkiye
email: asuman@srdc.metu.edu.tr

Abstract

METU INteroperable DBMS (MIND) is a multidatabase system based on OMG's distributed object management architecture. It is implemented on top of a CORBA compliant ORB, namely, DEC's ObjectBroker. 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. 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. 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

A multidatabase system (MDBS) is a database system that resides unobtrusively on top of existing database systems and allows the users to simultaneously access autonomous, heterogenous databases using a single data model and a query language.

A recent standard by OMG[1] , namely CORBA (The Common Object Request Broker Architecture) [12] 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 [13] is a complementary standard developed by the OMG for integrating distributed objects.

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[2], Sybase[3], Adabas D[4]and MOOD (METU Object-Oriented Database System) [2, 3, 4, 6]. 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 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.

The rest of the paper is organized as follows: The architecture of MIND 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

*This project is being partially supported by Motorola Inc., USA.

[1]OMG is a registered trademark, and CORBA, ORB, OMG IDL, Object Request Broker are trademarks of OMG.

[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.

the schema integration in MIND. The global query manager of the system is briefly summarized in Section 5. Finally, Section 6 provides a short description of the global transaction manager.

2 MIND Architecture

MIND architecture is based on OMG's Object Management Architecture (OMA), CORBA and COSS. The OMA defines a Reference Model identifying and characterizing the components, interfaces, and protocols that compose a distributed object architecture. CORBA is the core communication mechanism which enables distributed objects to operate on each other. COSS provides a set of standard functions to create objects, to control access to objects and to keep track of objects and object references.

In CORBA, clients ask for work to be done and servers do that work, all in terms of tasks called operations that are performed on entities called objects. Applications interact with each other without knowing where the other applications are on the network or how they accomplish their tasks. By using CORBA's model, it is possible to encapsulate applications as sets of distributed objects and their associated operations so that one can plug and unplug those client and server capabilities as they need to be added or replaced in a distributed system. These properties provide the means to handle heterogeneity at the database level. Thus, CORBA provides an infrastructure for implementing a multidatabase system. Semantic interoperability remains to be solved at the application programming level.

An overall view of MIND is provided in Figure 1.

Before we proceed with the architecture of MIND, we mention the fact that a common object server configuration in a CORBA environment supports two kinds of objects. The first kind of object, say X, has a standard interface with operations that change or query the object state; the second kind of object, say Y, is a factory object whose interface defines a single operation that creates objects of the X kind. A client uses interface X primarily to manipulate objects and interface Y to create objects. Thus, there is a factory object associated with each kind of object in the system except for the factory object itself which is created in the initialization part of the MIND system. Note the anology between an object factory and a class in an object-oriented programming language which is also an object factory. In the following discussion we choose to call object kinds as classes.

The basic components of MIND are a Global Database Agent (GDA) class (Object Factory in CORBA terminology) and a Local Database Agent (LDA) class:

1. 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.
2. A GDA class objects are responsible from:
    - parsing, decomposing, and optimizing the queries according to the information obtained from the SchemaInformationManager object,
    - global transaction management that ensures serializability of multidatabase transactions without violating the autonomy of local databases.

When a user wants to interact with MIND, a GDA object (GDAO) is created by sending a Create message to the Object Factory. ORB provides the location and implementation transparency for GDAOs. The locations of GDAOs are dynamically determined by the ORB using the information provided by the ORB administrator. The default site for the GDAO is usually the local host to prevent communication costs. A GDAO contains the objects of two classes, one responsible from global transaction management, namely Global Transaction Manager (GTM) class and the other from global query processing, namely Global Query Manager (GQM) class. A GTM object (GTMO) and a GQM object (GQMO) are created for each session of each client. GQMO obtains the global schema information necessary for the decomposition of the query from the SchemaInformation-Manager object. After decomposing, GQMO sends the subqueries to GTMO. GTMO is responsible from the correct execution of global transactions. For this purpose, it modifies the global subtransactions when necessary (e.g. enriching them with ticket operations) and controls the submission of global subtransations to the LDAOs. LDAOs control submission of operations to the LDBMSs and communicate with GTMO and GQMO to achieve atomic commitment of global transactions. Note that the LDAOs execute the global subqueries in parallel. The LDAO interface to the LDBMSs supports basic transaction and query primitives like BeginTrans, SendQuery, PrepareToCommit, CommitTrans, AbortTrans, GetNext, DeleteObj by using the Call Level Interface routines (CLI) of the component DBMSs.

In this architecture there is another class, called the Query Processor class responsible from processing the partial results returned by the LDAOs. As soon as two partial results that can be processed together appear at the LDAOs, a Query Processor Object(QPO) is created to process them. There could be as many QPOs running in parallel as needed to process the partial results. We think this is a highly dynamic scheme which provides maximum parallelism in the system that is possible without performing cost analysis.

In Figure 2, a screen snapshot of MIND graphical user interface is provided.

MIND is an evolving system (versions V.0.1, V.0.2 and V.1.0 have been produced upto now). As the system evolves, it provides clearer insights regarding the nature of issues in implementing a multidatabase system on a distributed object management architecture.

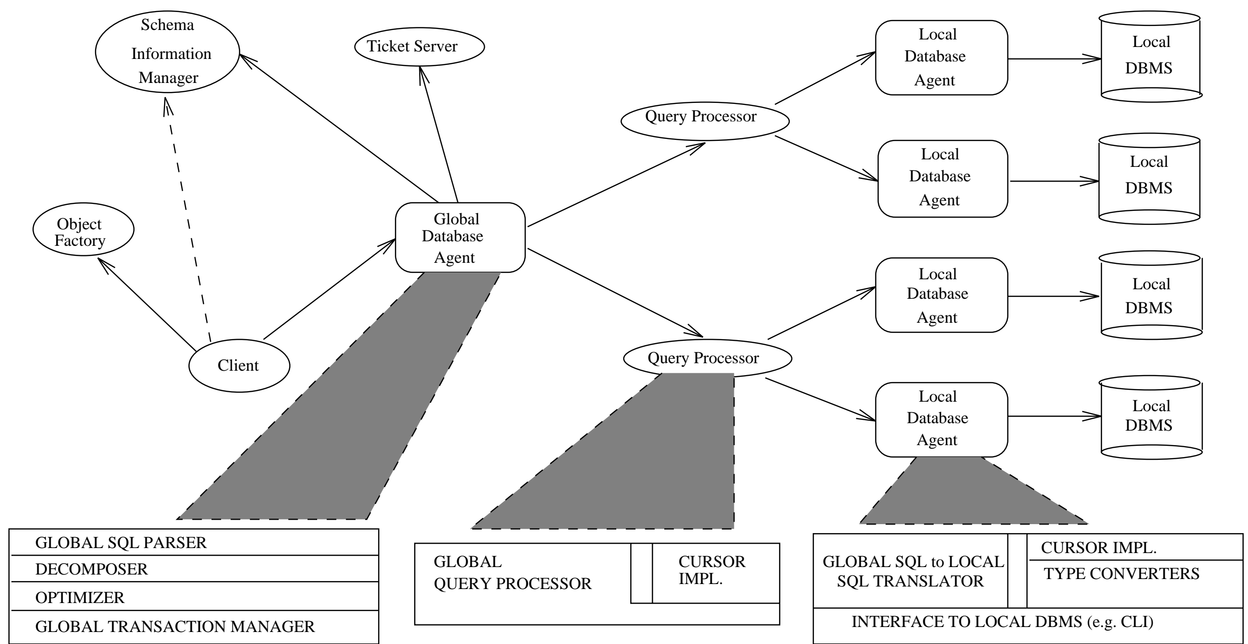

Figure 1: An Overview of MIND Architecture

3 The Infrastructure of MIND

As an initial step in implementing MIND, we encapsulated Oracle7, Sybase, Adabas D and MOOD DBMSs in multiple implementations of a generic Database Object [11]. The Database Object conveys requests from the client to the underlying DBMSs by using the CLIs of these DBMSs. The CLIs of these systems [14, 15, 1, 4] 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:

1) 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. Note that when more COSS functionality, like query services, transaction services are incorporated into CORBA implementations, most of the DBMS functionality will be available for database processing. In other words, the basic object services provide much of the functionality of a componentized DBMS, including both Object DBMS and Relational DBMS functionality.

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. In fact, it is necessary to recompile IDL code for ObjectBroker since it does not support dynamic server/skeleton interface yet. For other CORBA implementations (like SunSoft's DistributedObjectsEverywhere where dynamic server/skeleton interface is supported) such a recompilation may not be necessary.

Furthermore, the IDL code will be voluminous.

For the reasons stated above, we have defined each DBMS as an object.

2) Invocation style: 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*, 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 (like SendQuery primitive 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 as explained in section 5. Figure 3 illustrates invoking an operation to a database object instance.

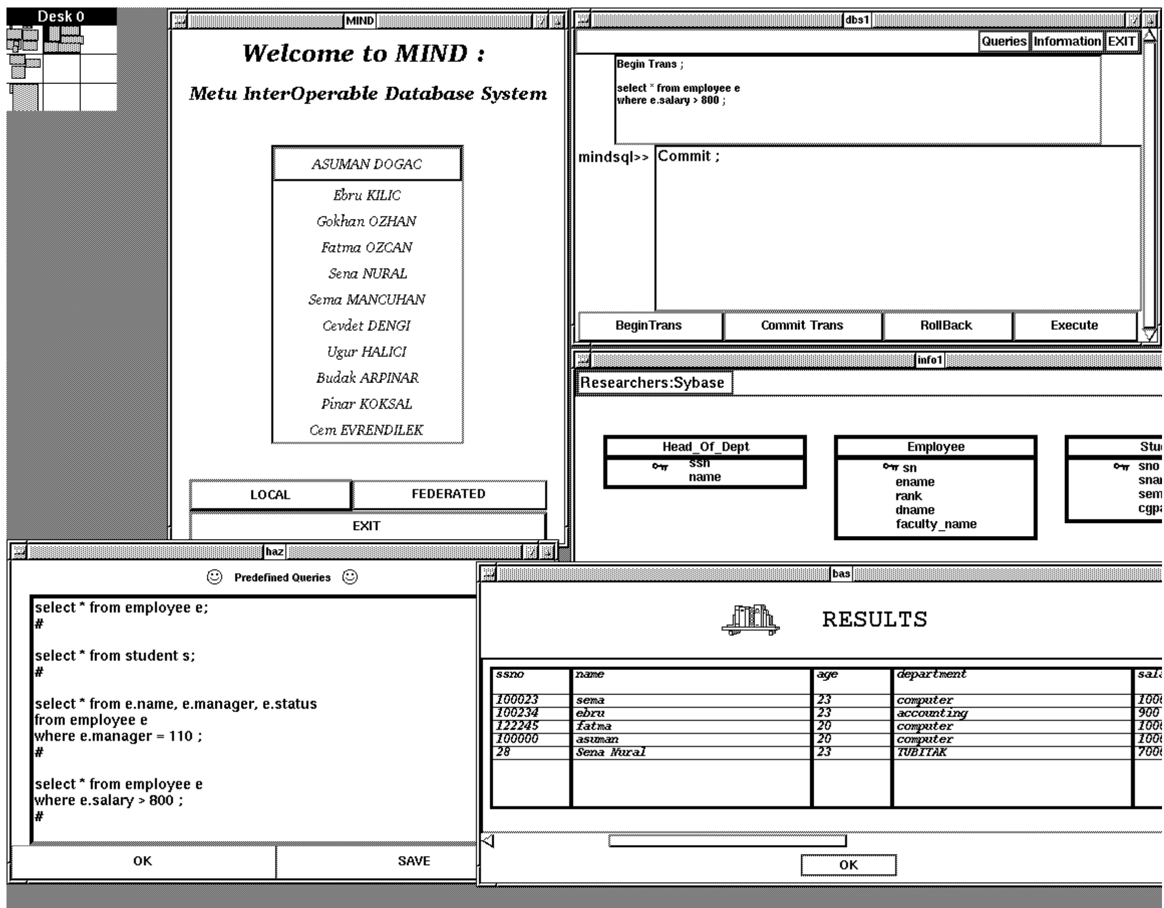

Figure 2: MIND GUI Screen snaphots

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

i. One interface to one implementation
ii. One interface to one of many implementations
iii. One interface to multiple implementations

When mapping one interface to one implementation, there is a direct one-to-one relationship between the operations in the interface and the methods in the implementation.

In the second alternative, for any one interface, it is possible to have several different implementations. Here again, there is a direct one-to-one relationship between the interface operations and the methods in the implementations.

The third alternative makes it possible to have multiple implementations associated with the same interface, with each implementation providing only a portion of the interface.

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.

4) Object Life Cycle: A client application needs an object reference to make a request. In CORBA, the initial object reference can be obtained in one of the following three ways:

i. The application developer can build the object reference into the client. This requires the client application to be compiled together with the server application. During compilation a built-in object reference is generated in both codes. Thus, the first request is statically bound to this built-in object reference.

ii. The client can get the object reference from an external source. This external source can be a file or a database containing initial object references in string form.

iii. ObjectBroker allows servers to store initial object references in the Advertisement Partition of Registery which corresponds to naming service of COSS. Then clients can access the Advertisement Partition to obtain object references.

The first approach makes the system inflexible since it is impossible to change a statically bound object reference without recompiling the system.

The second approach is not suitable either, since it is not feasible to maintain or replicate a single file or a database as an external source in a distributed

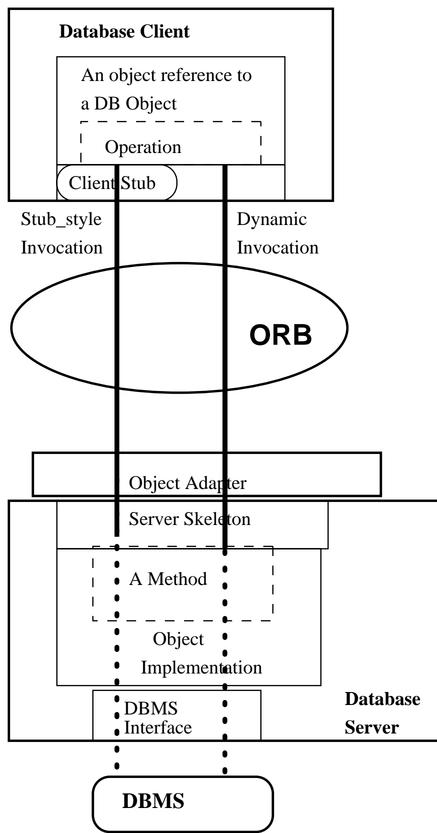

Figure 3: Invoking an operation to a Database Object instance through ORB

environment.

Therefore, we have chosen the third approach to get initial object references. The Advertisement Partition of Registry of ORB contains the object references of the following three objects: ObjectFactory object, TicketServer object and the SchemaInformationManager object. Since these objects serve the whole MIND system continuously, they are not created on demand; they are always active. Another point to note is the following: there is no need to have a different ObjectFactory for each kind of object, like GDA, LDA etc. Since the only function of Object Factory is to create these objects, having one ObjectFactory to create these objects does not create a bottleneck in the system.

The object references selected from the Advertisement Partition are used to create corresponding objects.

5) 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 :

i. Shared : The server can support more than one object for the implementation.

ii. Unshared : The server can support only one object at a time for the implementation.

iii. Server_per_method : A new server is used for each method invocation.

iv. 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 ObjectFactory, TicketServer, and the SchemaInformationManager object. There is one TicketServer object in MIND which provides a globally unique, monotonically increasing ticket number at its each invocation. The activation of SchemaInformationManager 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 ObjectFactory 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 sytem 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.

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:

1) 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.

2) 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.

3) 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.

4) 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.

4.1 Classification of Schema Conflicts

The potential schematic conflicts between the export schemas are identified according to the following classification framework:

1) Semantic Conflicts (Domain Conflicts): Two designers do not perceive exactly the same set of real world objects. For instance, a 'student' class may appear in one schema, while a more restrictive 'cs-student' class is in another schema. This is the first kind of conflict which relate to the domains of classes. The domain of a class is the set of objects in that class. According to their domains, relationships between classes are classified in four groups.

a) Identical Classes : Classes in export schemas represent the same set of objects (e.g. 'instructors' in one database and 'lecturers' in another database). These classes are merged into a single class in the global schema.

b) Intersecting Classes: Classes may represent overlapping sets of objects. For instance, 'student' and 'employee' classes in two databases may have common objects which represent research-assistants. Such classes are integrated in the global schema as classes having a common subclass that contains the set of common objects.

c) Inclusion Classes: The domain of one class is a subset of another class (e.g. Employee & Manager classes). In the global schema the more restrictive class is represented as the subclass of the other class.

d) Disjoint Classes: These are classes whose domains are completely different but related (e.g. Graduate vs Undergraduate students). Such classes are generalized into a superclass in the global schema.

2) Structural Conflicts (Descriptive Conflicts): When describing related sets of real-world objects, two designers do not perceive exactly the same set of properties. This second kind of conflict includes naming conflicts due to the homonyms and synonyms, attribute domain, scale, constraints, operations etc. For instance, different number of attributes or methods may be defined for semantically equivalent classes; or different names may be used for identical attributes. Such conflicts are resolved by providing a mapping between the class and attribute names in the global schema and class and attribute names in export schemas. This mapping is expressed in the object definition language described in Section 4.2.

4.2 MIND Schema Integrator

In MIND, LDAOs translate the local schemas into the export schemas using ODL, and then their export schemas are stored in the SchemaInformationManager. SchemaInformationManager integrates these export schemas to generate a global schema. Schema integration is a two phase process:

1) 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 given in Section 4.1.

2) 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. 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. In ODL, a schema definition is a list of interface definitions whose general form looks as follows:

```
interface classname:superclass_list {
     extent    extentname;
     keys      attr1;
     attribute attr_type attr1;
     ...
     relationship OtherClass relname
        inverse OtherClass::invrel;
     ...
}
```

where **classname** is the name of the class whose interface is defined; **superclass_list** includes all superclasses of the class; **extentname** provides access to the set of all instances of the class; **keys** allows to define a set of attributes which uniquely identifies each object of the class. **Attribute** and **relationship** constitute the signature of the class.

In addition to its interface definition, each class needs information to determine the extent and to map the attributes onto the local ones. The general syntax

for this mapping definition which is similar to [10] is provided in the following:

```
mapping classname {
   origin originname1: classname1 alias1 [,
          originname2: classname2 alias2,...];
   def_ext    extname  as
              select
              from alias1, alias2, ...
              where ...;
   def_attr attr1 as [alias1.attrname |
                       select alias1.attrname,
                              alias2.attrname
                       from alias1, alias2
                       where  ...; ]
   def_rel relname as
          select *
          from alias1, alias2 ...
          where ... ;];
}
```

The keyword **mapping** marks the block as a mapping definition for the derived class **classname**. The **origin** clauses define a set of private attributes that store the back-references to those objects, from which an instance of this class has been derived. The extent derivation clause starting with **def_ext** defines a query that provides full instantiation of the derived class. A list of **def_attr** lines defines the mapping for each attribute of the class. And finally, a set of **def_rel** lines express the relationships between derived classes as separate queries which actually represent the traversal path definitions.

After the global schema is obtained SchemaInformationManager provides the necessary information to the GQMO on demand. We are currently developing a graphical tool which will automatically generate textual specification of the global 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. The assertions 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

MIND V.0.2 [5] has a cost based optimizer [7]. In MIND V.1.0, we implemented a dynamic query processor. The main idea behind dynamic query processor is to exploit the inherent parallelism in the system as much as possible without performing a cost based analysis.

The dynamic MIND query processor is implemented as follows: When a user submits a global query to MIND, a GDAO, a GTMO and a GQMO are created as explained in Section 2. GQMO obtains the global schema information necessary for the decomposition of the query from GlobalSchemaInformation object. GQMO sends the subqueries to GTMO which controls the submission of global subtransations to the LDAOs. The subqueries are submitted to LDAOs in the deferred synchronous mode, i.e., after submitting the subquery the GTMO does not wait for it to complete. This provides different LDBMSs to execute the subqueries in parallel.

GQMO schedules the intersite operations dynamically as follows: GQMO keeps a waiting list of object identifiers of the LDAOs to which a subquery has been submitted and polls this list to see whether any of the partial results has appeared. When a partial result appears, if there is no matching operand to process this partial result, the object identifier of the LDAO containing the partial result is added to a success list. Whenever the object identifiers of two partial results that need to be processed together (e.g. a join operation, a union operation, etc.) appears in the success list, GQMO creates a QPO by sending a Create message to Object Factory. Activation of QPO is also in deferred synchronous mode and the object identifier of this QPO is added to the waiting list. It is clear that in this architecture two partial results that need to be processed together are executed as soon as they appear. This architecture is highly dynamic and obtains the maximum parallel execution that is possible without cost calculation.

6 MIND Transaction Management

In MIND V.0.2 [5], a ticket based algorithm has been implemented for flat transactions. 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 [9]. 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. In the following the NTNT algorithm is explained. In Section 6.1 an example is provided to clarify the technique.

We present the NTNT technique by referring to the pseudocode of the algorithm. To be able to provide a neat recursive algorithm, we imagine all the global transactions to be children of a virtual transaction called OMNI. When OMNI transaction starts

executing, it creates a siteTicket(OMNI) at each site whose default value is 0. Then we imagine that OMNI transaction executes forever. Since it is an imaginary transaction, it does not need to commit finally to make the updates of its children persistent.

GlobalBegin(T_i^G) assigns a globally unique and monotonically increasing ticket number denoted as TN to all transactions when they are initiated, that is, both the parent and the child transactions at all levels obtain a ticket. A Ticket Server object in MIND provides tickets and guarantees that a new subtransaction obtains a ticket whose value is greater than any of the previously assigned ticket numbers. Since any child is submitted after its parent, this automatically provides that any child has a ticket number greater than its parent's ticket. When the first Data Management(DM) read or DM write operation of a subtransaction T_i^G is to be executed at a local site, *LocalBegin*(T_i^G, k) is executed which starts all ancestors of the subtransaction if they are not initiated at this site yet. Next, each child transaction reads the local ticket created by its parent at this site (this ticket is created for the children of $parent(T_i^G)$, i.e. $siblings(T_i^G)$), and checks if its own ticket value is greater than the stored ticket value in the ticket for $siblings(T_i^G)$ at this site. If it is not, the transaction T_i^G is aborted at all related sites and resubmitted to MIND using the algorithms given in *GlobalAbort*(T_i^G) and *GlobalRestart*(T_i^G). As a result, all siblings of a subtransaction accessing to some site k are forced into conflict through a ticket item created by the parent of these siblings at site k. The pseudocode of the algorithm to check ticket values is presented in *LocalCheckTicket*(T_i^G, k). This mechanism makes the execution order of all siblings of a subtransaction to be consistent at all related sites by the use of tickets. In other words, the consistency of serialization order of the siblings are provided by guaranteeing them to be serialized in the order of their ticket numbers. If a transaction is validated using the *LocalCheckTicket*(T_i^G, k) algorithm then its read and write operations on any item x are submitted to related LDBMS by *LocalWrite(x)*, *LocalRead(x)* algorithms and committed by *GlobalCommit*(T_i^G). *GlobalCommit*(T_i^G) is executed after all children of T_i^G commit or abort. *GlobalCommit*(T_i^G) coordinates the 2PC protocol and if all LDBMSs replied Ready then commits the subtransaction.

In multidatabases when a local scheduler fails to commit a subtransaction, in order to provide the atomicity of the nested transaction, this subtransaction must be aborted at all related sites. This in turn necessitates commit operations of subtransactions to be achieved in two phases: PrepareToCommit, and Commit. In other words, LDBMSs should also support 2PC protocol for subtransactions. When this is not supported, a solution to remedy the situation is to abort the immediate parents at all related sites. This is possible because the commit message of the parent has not been sent yet.

If PrepareToCommit message is available at the user transaction level (i.e. immediate child of OMNI) this facility is used at this level. If the underlying DBMS does not support PrepareToCommit even at this level then PreparedToCommit state of a user transaction can be simulated as described in [8].

GarbageCollector algorithm which is not presented here is executed periodically to delete the ticket items created by the subtransactions.

NTNT Algorithm:

GlobalBegin(T_i^G):
begin
  Get global ticket for T_i^G *so that*
    $TN(T_i^G)$:=*lastTicketNo*+1;
    lastTicketNo:=$TN(T_i^G)$
end □

LocalBegin(T_i^G, k):
begin
  If parent(T_i^G, k) *has not started at site k yet then*
  begin
    LocalBegin(*parent*(T_i^G), k);
    Send Begin(T_i^G) *as child of parent*(T_i^G) *to Local Transaction Manager (LTM) at site k;*
  end
  LocalCheckTicket(T_i^G, k);
  If check FAILs then GlobalRestart(T_i^G);
end □

LocalCheckTicket(T_i^G, k):
begin
  If T_i^G *is not OMNI then*
  begin
    If siteTicket(*parent*(T_i^G)) > $TN(T_i^G)$ *then FAIL;*
    else
    begin
      siteTicket(*parent*(T_i^G)):=$TN(T_i^G)$;
      create(*siteTicket*(T_i^G)) *at site k with default value 0;*
    end
  end
end □

LocalWrite(x), LocalRead(x):
begin
  If the site(x) is to be visited first time by T_i^G *then*
    LocalBegin(T_i^G ,k);
  Forward the operation to Local Data Manager on behalf of T_i^G;
end □

GlobalAbort(T_i^G):
begin
for each related site k
  send LocalAbort(T_i^G) *message to LTM at site k;*
end □

GlobalRestart(T_i^G):
begin
  GlobalAbort(T_i^G);

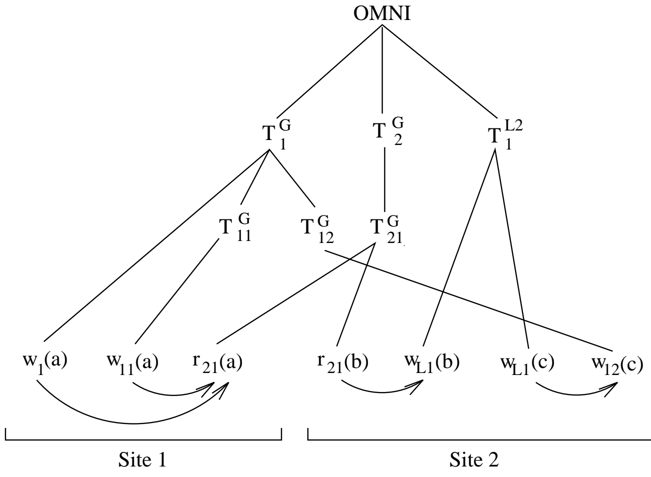

Figure 4: A Schedule of Nested Multidatabase Transactions

GlobalBegin(T_i^G);
end □

GlobalCommit(T_i^G):
begin
wait until all children(T_i^G) commits or aborts;
for each related site k
send PrepareToCommit(T_i^G) message to LTM at site k;
If all LTMs have replied Ready
for each related site k
send Commit(T_i^G) message to LTM at site k;
If any site fails to PrepareToCommit then
GlobalAbort(T_i^G);
end □

6.1 An Example

In the following, an example is provided to clarify the NTNT technique. Assume a multidatabase system with two LDBMSs at sites 1 and 2. User transactions can be arbitrarily nested and each (sub)transaction can issue read and write operations denoted as $r(a)$ and $w(a)$ respectively.

Figure 4 depicts the execution of two nested multidatabase transactions T_1^G and T_2^G, and a local transaction T_1^{L2}. Global transaction T_1^G has two subtransactions T_{11}^G and T_{12}^G, and T_2^G has one subtransaction T_{21}^G.

NTNT technique works for this example as follows: Assume the tickets obtained from the ticket server to be as follow:

$TN(OMNI) = 0$,
$TN(T_1^G) = 1$,
$TN(T_2^G) = 2$,
$TN(T_{11}^G) = 3$,
$TN(T_{21}^G) = 4$,
$TN(T_{12}^G) = 5$.

Execution at site 1:

T_1^G is accepted since
$siteTicket(parent(T_1^G)) =$
$TN(OMNI) = 0 < TN(T_1^G) = 1$
and $siteTicket(parent(T_1^G))$ is set to 1 and $siteTicket(T_1^G)$ is created with default value 0. Thus $w_1(a)$ is executed. Since
$siteTicket(parent(T_{11}^G)) = 0 < TN(T_{11}^G) = 3$,
$siteTicket(parent(T_{11}^G))$ is set to 3 and $w_{11}(a)$ is executed. Similarly
$siteTicket(parent(T_2^G)) = 1 < TN(T_2^G) = 2$,
T_2^G is accepted and $siteTicket(OMNI)$ becomes 2 and $siteTicket(T_2^G)$ is created with default value 0. $r_{21}(a)$ is executed because
$siteTicket(parent(T_{21}^G)) = 0 < TN(T_{21}^G) = 4$
and $siteTicket(parent(T_{21}^G))$ is set to 4.

Execution at site 2:

T_2^G is accepted since
$siteTicket(parent(T_2^G)) =$
$TN(OMNI) = 0 < TN(T_2^G) = 2$
and $siteTicket(parent(T_2^G))$ is set to 2. $siteTicket(T_2^G)$ is created with default value 0. T_{21}^G is accepted and $r_{21}(b)$ is executed since
$siteTicket(parent(T_{21}^G)) = 0 < TN(T_{21}^G) = 4$.
Yet T_1^G at site 2 is rejected and aborted at all sites since $siteTicket(parent(T_1^G)) = 2$ which is not less than $TN(T_1^G) = 1$.

The correctness proof of the NTNT technique is provided in [9]. The recovery manager of MIND is under development.

7 Conclusions

In this paper, we describe our experiences in building a multidatabase system, namely MIND, based on OMG's object management architecture. 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 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.

References

[1] ENTIRE SQL-DB Server Call Interface, ESD-311-316, SOFTWARE AG, April 1993.

[2] 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.

[3] 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.

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

[5] 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", *ACM SIGMOD Record*, Vol.24, No.3, September 1995.

[6] Dogac, A., Altinel, M., Ozkan, C., Durusoy, I., "Implementation Aspects of an Object-Oriented DBMS", *ACM SIGMOD Record*, Vol.24, No.1, March 1995.

[7] Evrendilek, C., Dogac, A., Nural, S., Ozcan, F., "Query Optimization in Multidatabase Systems", *Proc. of the Next Generation Information Technologies and Systems*, Israel, June 1995.

[8] 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.

[9] 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.

[10] Huck, G., Fankhauser, P., Busse, R., Klas, W., "IRO-DB : An Object-Oriented Approach towards Federated and Interoprable DBMSs", *Proc. of ADBIS '94*, Moscow, May 1994.

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

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

[13] Object Management Group, "The Common Object Services Specification, Volume 1", OMG Document Number 94.1.1, January 1994.

[14] Programmer's Guide to the Oracle Call Interfaces, Oracle Corporation, December 1992.

[15] Open Client DB-Library /C Reference Manual, Sybase Inc., November 1990.