STUDIA INFORMATICA 2011
Volume 32 Number 3B (99)

Miłosz GÓRALCZYK, Jarosław KOSZELA
Military University of Technology, Informatics System Institute

ARCHITECTURE OF OBJECT DATABASE MUTDOD

Summary. This article contains overall description of architecture of distributed object database MUTDOD, which is created at Military University of Technology. This paper describes a metamodel schema and unified data model – which is shared between application and database environment. Also a division of query execution process is explained.

Keywords: architecture, metamodel, distributed object database, MUTDOD

ARCHITEKTURA OBIEKTOWEJ BAZY DANYCH MUTDOD

Streszczenie. Artykuł zawiera ogólny opis architektury rozproszonej obiektowej bazy danych, która jest tworzona w Wojskowej Akademii Technicznej. Materiał opisuje metamodel oraz ujednolicony model danych, który jest współdzielony pomiędzy środowisko aplikacji i bazy danych. Ponadto, został wyjaśniony podział etapów wykonania zapytania.

Słowa kluczowe: architektura, metamodel, rozproszona obiektowa baza danych, MUTDOD

1. Introduction

Based on an IDC's report [4]: in 2008 computer systems and their users produced about 427EB (Eksa ~ 10^{18}) of electronic data, but in 2012 they will produce over 2 ZB (Zetta ~ 10^{21}) – it is over 2 billion terabytes of data. It means that a volume of produced data will increase for about 100% every year. Probably a growth will be even higher in the following years. To handle this issue software engineers have to improve data storage and processing mechanisms to make them more effective. A simple way to solve this problem may be a **large-capacity distributed database systems**, which store and process a large data volume.

Current computer systems are going to be increasingly more complex. Time and a budget needed to create a computer system, may be reduced by using **object-oriented languages**, which allow for dividing system into modules. **OOP** (Object Oriented Programming) is also known of being developer-friendly. However using an object-oriented language may cause problems in data access, especially when system is cooperating with a **relationship database** - the most popular database kind nowadays. There is an **impedance mismatch of data structures** [5, 10] between object model in application and relational data model in database. That is why **ORM**s (Object-Relational Mapping) are used to transform data between these two different structures. Unfortunately, an additional mapping reduces system's performance and requires to prepare (or generate) a lot of additional code and configurations.

2. What is the MUTDOD?

In the future, **object database systems** [3, 10] may substitute relationship databases in most cases. At this moment there is no object database system, which serves full and natural (intuitional) objectivity to software engineers – there is more about this defects in following chapters. A try of creating such solution is **MUTDOD System**. MUTDOD stands for a **M**ilitary **U**niversity of **T**echnology **D**istributed **O**bject **D**atabase System, which is creating at title University in Warsaw, Poland. MUTDOD is mentioned to be a platform for creating (complex) object application with an object-data storage. This paper describes an object side of MUTDOD. More information about a distribution in MUTDOD is in [5, 14].

MUTDOD is a project which includes its own **object database engine** and **ODBMS** [6, 7] (Object Database Management System) which can be installed on distrusted (Windows) hosts. There are two kinds of MUTDOD's servers – a central server and a data server. The first one manages data hosts and a query execution. At this point MUTDOD works only with one instance of a central server, but in the future it will be divided and distributed to eliminate this performance bottleneck. The second one physically stores data and executes queries. There can be many data servers.

MUTDOD also contains an executive environment for client applications. At this point it is built as an ODBC-like provider for .NET applications, but in the future it might be required to create a separate runtime environment (like CLR[1] or JRE[2]).

All MUTDOD's elements are designed and implemented from beginning at Military University of Technology, including an object database engine. MUTDOD is build using .NET

[1] CLR – stands for a Common Languages Runtime; it is a runtime environment for Microsoft .NET Framework platform; more information about CLR you can find on [9].

[2] JRE – a Java Runtime Environment; it is a platform, which allows to run application created in Java.

Framework 4.0 in C# and C++. At this point it works only with .NET-based client applications. Support for other platforms and languages will be added in the future.

3. Unified data model

Von Neumann's architecture specifies [11] that application's executive code and data are kept in the same memory. It has been so up to this very day – all of mentioned elements are in RAM memory, but heavy data is stored on a hard disk or in the database – usually on remote host. In this way only objects in RAM are easy accessible for programmer (all operation can be performed on-the-fly). Stored data is loaded in disagreeable way – first of all there is a need to create a mapping from database's relationship structure to application's business objects. Even if some generator would be used, there is still a lot of effort to customize or optimize metadata. Secondly, while the programmer is attempting to use (remote) data from a database, he has to create a database connection, send query, check if execution do not generate some errors, etc. It is an additional effort, needles any way. The programmer should access the remote data in the same way as the local one.

Over the years software engineers got used to accessing database data in that tedious way. But there is no need to do it like that. For the programmer there is no difference between remote and local data (except their access latency). MUTDOD system is mentioned to serve **unified access** to both types of data. It means that for a programmer data access and processing are **transparent** – the developer does not have to care about data location. For example: there is a computer system to register all cars and their owners in Poland. In a database the data for the whole country is kept, probably a lot of GB. There is also an object application which serves functionality of **CRUD**[3] operation on all objects. To implement functionality of a new car registration a programmer has to get data about an owner and his other cars. He creates a new instance of a Car class, fills its attributes and… that is all, he does not have to do anything else. Programmer does not have to care about data synchronization with database – this is the role of MUTDOD. Please, notice that with MUTDOD programmer creates a database object as if it would be a local object. It works the same with update or delete operation – programmer just works on objects and does not trouble how to get an access to them.

To provide this functionality there is one key requirement – the **data model** in the application and the database has to be unified or at least a shared part of it – there will be more about this in a next paragraph. It means that objects in both **schemas** have the same attributes

[3] CRUD – stands for create, read, update, delete operations.

and methods, because objects have to be serialized and deserialized on both sides[4]. Of course using proper CASE tools will reduce time of system designing, because only one model of data will be designed, shared one object model for both – a database and an application. This is in opposition to using two different models – a relational one for the database and an object-oriented one for the application, plus additional ORM mappings.

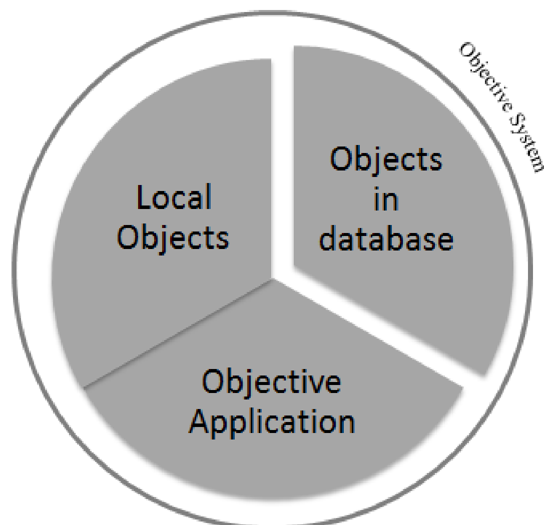

Fig. 1. Integral parts of objective computer system
Rys. 1. Integralne części obiektowego systemu komputerowego

What is more, because of a **unified data model** the programmer has only to create a configuration file to access objects in database – there is no need to create any database connector objects in code[5]. So MUTDOD allows software engineers to start thinking about systems, in which data from an object database is integral part of whole system (see Fig. 1) and which is easily accessible. They can think about a one unified data model.

It has to be mentioned that a unified object model for database and application does not imply that there is no difference between these two models. Some application's objects (attributes or methods) may be available for local use only, while some objects in a database may be secured for client applications. During creation of a data model programmer may operate with extended **encapsulation visibility modifiers**, beside a common: public, private, protected. A new one is: database (for elements only for database) and application (for elements only used in application). It will be only two additional encapsulation states. Both will be provided with MUTDOD.

[4] Notice that there will be no mapping, because both data models are objective.
[5] Like SQLProviders, ODBC, JDBC, ect for current solutions.

4. Easy data selecting

For the majority of computer systems the most common operation is data selection – **select queries**. However it is a rare case to select all stored data without any filtering or grouping. In a typical select operation there are also some join operations for many tables to get data across many relationships. For example to get VIN numbers of cars, whose owners' name is "Simpson" there is a need to join two tables – Car (which stores cars' information) and Person (which stores information about cars' owners).

Building such query in **OQL** (Object Query Language) requires: creating a database connector object, sending a query and results mapping[6] (database objects) to application's objects. In addition, OQL query is (usually) hard coded as a string in application's code, so there is no syntax check or semantic check of query during a code compilation. Even OQL syntax seems to be orthogonal to "objective world", because there are no (or just few) benefits with using OQL queries (and an object database) instead of SQL (and a relationship database). As long as developers will use OQL they will still have to think in a relational way during a queries creation.

Last but not least, a disadvantage of OQL is that there is no way to use application's language **build-in functions** (methods), e.g. .NET's or Java's string operation like: "To Lower Case", "Split", "Remove", "Replace", etc. High-level programmers got used to use them in an application code, so why they cannot use this functions in database query? Please consider a previously-mentioned situation, about cars' owners named "Simpson". Please consider a situation where there is no validation in application or data is migrated from many different systems. In that case in person's name attribute there can be values like: "Simpson", "simpson", "SIMPSON", etc. To get full results a developer has to write every combination of upper and lower cases of a name string in query's where section – please look on sample queries in the table 1.

Another way of selecting data from a database is using of a Linq[7,8]. It provides ORM, a **keyword-based syntax** and on-code-compilation errors checking. For programmers it is useful because they can use default syntax autocompleting tools and does not have to check if query has any syntax error in a runtime – a code compilation will stop if any error is found in a Linq query. Especially a good thing about the Linq is that programmers can use .NET objects' build-in functions. So in case of different spelling of car owner's name programmers

[6] Notice that even if there is no data structure impedance mismatch between models of an object application and a common object database, a mapping results to application's objects is required if a database works with OQL.

[7] LINQ – Language-INtegrated Query, an extension for .NET, which make access to data in database easier

[8] Of course there are many other equivalents of Linq for .NET and other languages.

can reduce query complexity with using a code like this: Owner.Name.ToLower("simpson"). For whole query, please look onto a table 1.

Table 1

Sample queries in object query languages

OQL	Linq	SBQL
select CAR.VIN from CAR join PERSON on CAR.OWNERID = PERSON.ID where PERSON. NAME = 'Simpson' Or PERSON.NAME = 'simpson' Or PERSON.NAME ='SIMPSON'	rom car in DB.Cars join person in DB.Persons on car.OWNERID equals person.ID where person.Name. ToLower(). Equals("simpson") select car.VIN	Car(WHERE Owner.Name .ToLower() .Equals("simpson")).VIN

As you can see in the Linq still has an OQL-like syntax. In other versions Linq[9] all references are in metamodel, so programmers do not have to use join operation. Instead, a Person object contains a **list of references** to owned cars and the Car object has a **reference** to its owner – an instance of a Person. It is a step in the right direction, because it reduces complexity of queries, so it also reduces queries developing time and chance of making mistakes.

A next step in object query language[10] is moving to a **full objective expression** (syntax). This kind of syntax is familiar for programmers, who use any of objective languages. Using a query language should be natural and intuitional for developers of objective applications. There should be no influence of SQL, because it is needless. As it is in Linq, an object query language should contain build-in keywords of a programming language, in which application is coded. A sample of this kind of language is SBQL (Stack-Based Query Language) [2, 8].

At this point MUTDOD provides most of the basic functionality of SBQL. In a near future the query language will provide some additional functionality to control distribution of query processing. MUTDOD's query language will also handle unified data models, so some stages of query execution will be moved to client's environment – more about this you can read in part 6.

5. Objective metamodel

MUTDOD's **data metamodel** is based on the **ODMG** [1] metamodel. It has to keep many kinds of information about stored data. The most important is the information about

[9] E.g.: Linq to Objects, Linq to Entities,ect.

[10] Linq is also available for finding objects in .NET collections, but still with the same, eclectic SQL-like syntax.

database schema, classes used and relations between objects. There are four kind of relation between objects:

- **generalization** – which allows to operate on object's dynamic roles, so in different contexts programmers can operate on different attributes of an object (e.g. cast Person object to Student object); for all objects base class is class Object;
- **aggregation** – allows software engineers to design relations between main object (container) and its parts; main object can exist without any part;
- **composition** – as above, but main object cannot exist without its parts. In MUTDOD a composition is also used when a class is divided horizontally[11] – some of object's attributes are stored in different location than the rest;
- **association** – reference to other object.

A relation in MUTDOD is an organized triple. It contains OID of object A, OID of object B and type of relation. **OID**s **O**bject **ID**entifier [1]. For any object in MUTDOD, OID is unique. Except primitive types (like int, double, OID, ect.) all attributes of a complex object are also objects.

Other thing which is stored in metamodel is information about elements encapsulation. In MUTDOD there are three common encapsulation states: public, protected and private. These are used to hide attributes and methods for inheritance. But as it was written above, due to unified data model for database and application there are two additional states: database and application. First one makes an element "visible" to database side, while second one works on application side. A use of new **encapsulation visibility modifiers** can be combined with the "old" modifiers, because you can create database protected attribute or application public methods. New modifiers allow creating objects which are used only on one side – like temporary object on client's side.

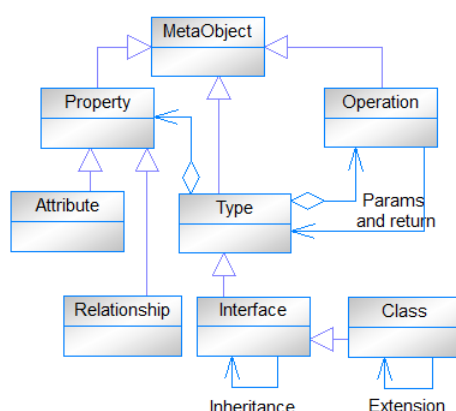

Fig. 2. MUTDOD's simplified metamodel schema without distribution elements
Rys. 2. Uproszczony schemat metadanych MUTDOD, bez elementów rozproszenia

[11] When a class is divided horizontally metamodel creates 2 or more subclasses and links them with composition to base class. An object division is transparent to programmers and has no influence for query's syntax if it is not declared by the programmer.

The metamodel keeps also **security** information, because some part of information could be secret or protected by law – like personal identity. MUTDOD data metamodel is designed to store all above information[12] – its simplified schema is shown on Fig. 2 as a modification of the ODMG metamodel schema.

MUTDOD's distribution requires that information about a **data location** has to be stored in the metamodel too. In MUTDOD data can be **divided vertically** [12] – some instances of a class can be stored on one host, while other objects of the same class are stored on other host. It may improve performance, e.g. host A is storing information about cars' owners, while host B contains objects of motorcycle owners.

Data can be also **divided horizontally** [13], what means that some attributes can be stored on a different (e.g. secured, faster) host. Please imagine a situation, when object Person contains personal information about cars' owner, list of owned car, login and image, as an avatar for web access. This object can be divided horizontally into two parts: a first part for a personal data and a car list for government use only, which are stored on secure, encrypted MUTDOD host; and a second part for an image and a login, on a second (less secure) MUTDOD host available for web application.

6. Divided execution

A main problem of object applications, which use ORM to operate on multiple rows from database, is that almost all operations (especially updates) on this data have to be processed in iterations – e.g. with foreach loops. It produces additional, yet unneeded operation cost and network traffic – an application has to select data, retrieve data (which is transformed into objects), for each element of collection updates object's attributes and send it back to the server. It is caused by an imperative nature of programming languages. But an object database environment should provide a query language, which combines features from both – **imperative and declarative languages** – so MUTDOD's one will do.

MUTDOD will provide a runtime environment for a client site, which will allow using a unified data model and make programmers able to use declarative operation between imperative operations in code of an application. So it will be possible to update an attribute for multiple objects with a one expression, and with no data transfer from the server – whole expression will be executed on server side. It could be something like this: *Employee*(Where Name==*"Simpson"*).*Salary* *= 0.1 – to increase salary for 10% for each employee, whose

[12] Not all features are implemented at the moment.

name is Simpson. Above query has to be differed from other application operations, which are coded in a programming language.

To handle with this MUTDOD's client environment is designed to be an extension of server environment. Unlikely a common database connection, MUTDOD's client has to take two additional states in **query execution**[13]. As you can see on fig. 3 this stages are: a query optimization and a query interpretation.

A **query optimization** is a stage which is responsible for choosing fastest way of query execution, division query to subqueries, ect. In MUTDOD it gets an additional role to identify, which operations require to return only references (OIDs) of selected objects and which ones have to return all the data. For example: if application will use selected collection only to modify value of each element, database may return only OIDs, because no value from objects will be used – there is no need to dereference anything. Other example is when all data will be displayed to end user in a grid – in this case whole objects (with their values) have to be returned. That is why MUTDOD's client environment has to take responsibility of query optimization (at least a part of it).

A **query interpretation** fills up the previous stage. It allows client applications to send a semi-processed (a compiled) query, so its execution time should be lower than for a query fully processed on the server side.

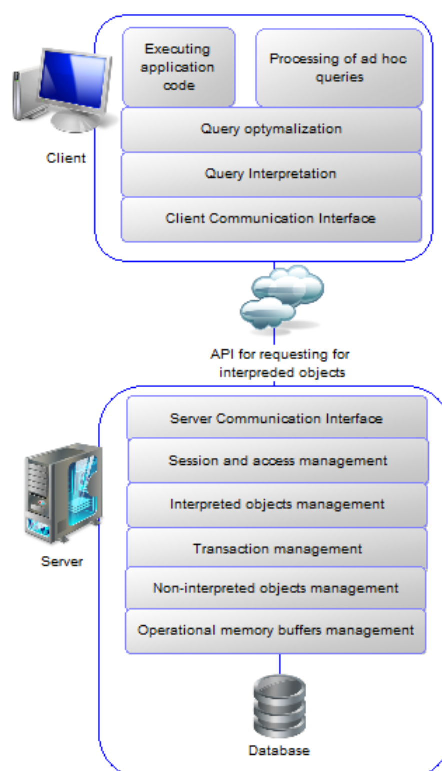

Fig. 3. Division of query execution process in application working with MUTDOD
Rys. 3. Podział procesu wykonywania w systemie wykorzystującym MUTDOD

[13] Of course these stages are automated and give no additional effort to a programmer.

Both stages require metadata from a database. The idea of a unified data model assumes that a database schema is shared with a client environment. There are also some other elements of metadata which have to be shared too, because they are required for an optimization and an interpretation process. Some of these elements are: indexes, elements' actual state and elements visibility (e.g. secured object should not be available at client side).

7. Summary

MUTDOD as a project is at the beginning of its research. The main idea – creating a platform for object systems with an object database – is constant, but minor ideas and its features are evolving. MUTDOD is a project, which is meant to find the answers to some of following questions – how object database platform should look like; how to make coding of complex application easier; how to join a code of operation with collection of data; what kind of features should object database server have to make it as popular and useful as relational databases; and many more.

Future steps of MUTDOD project research will be:

- deeper unification of client and database environment;
- making data processing more transparent for programmer;
- extending query language and trying to make it programming language independent (to run on .NET, JAVA, Ruby, etc.);
- working on local processing distribution and system effectiveness;
- discovering the best way of FAR strategy (Fragmentation Allocation and Replication) for data.

BIBLIOGRAPHY

1. Catell R. G. G.: The Object Database Standard: ODMG 3.0. Morgan Kaufmann, 2000.
2. Subieta K.: Teoria i konstrukcja obiektowych języków zapytań. Wydawnictwo PJWSTK, Warszawa 2004.
3. Lausen G., Vossen G.: Models and Languages of Object-Oriented Databases. Addison Wesley Longman Limited, 1997.
4. IDC Group: As the Economy Contracts, the Digital Universe Expands. http://www.emc-.com/collateral/demos/microsites/idc-digital-universe/iview.htm, valid on at 10th November 2009.

5. Brzozowska P., Góralczyk M., Jesionek Ł., Karpiński M., Kędzierski G., Kędzierski P., Koszela J., Wróbel E.: System obiektowy = obiektowa baza danych + obiektowa aplikacja. Studia Informatica, Vol. 31, No. 2B (90), Gliwice 2010.
6. Góralczyk M.: Projekt oprogramowania zarządzającego obiektową bazą danych. WAT, Warszawa 2010.
7. Karpiński M.: Projekt mechanizmu replikacji i synchronizacji elementów obiektowej bazy danych. WAT, Warszawa 2010.
8. Stack-Based Architecture (SBA) and Stack-Based Query Language (SBQL). www.sbql-.pl, valid on 1st April 2011.
9. Microsoft Corporation: Common Language Runtime (CLR). http://msdn.microsoft.com/pl-pl/library/8bs2ecf4.aspx, valid on 1st April 2011.
10. Subieta K.: Słownik terminów z zakresu obiektowości. Akademicka Oficyna Wydawnicza PLJ, Warszawa 1999.
11. Rojas R., Hashagen U.: The First Computers: History and Architectures. MIT Press, Cambridge MA 2000.
12. Bellatreche L., Simonet A., Simonet M.: Vertical fragmentation in distributed object database systems with complex attributes and methods. 7th International Workshop on Database and Expert Systems Applications (DEXA'96), 1996.
13. Bellatreche L., Simonet A.: Horizontal fragmentation in distributed object database systems. Lecture Notes in Computer Science, Volume 1127/1996, Springer, Berlin Heidelberg 1996.
14. Karpiński M., Koszela J.: Object-oriented distribution in MUTDOD. Studia Informatica, Vol. 32, No. 3B (99), Gliwice 2011.

Recenzenci: Dr inż. Ewa Płuciennik-Psota
Dr inż. Aleksandra Werner

Wpłynęło do Redakcji 16 stycznia 2011 r.

Omówienie

MUTDOD (ang. **M**ilitary **U**niversity of **T**echnology **D**istributed **O**bject **D**atabase) to rozproszona, obiektowa baza danych tworzona w Wojskowej Akademii Technicznej. MUTDOD jest projektowany jako platforma dla systemów tworzonych w obiektowych językach programowania.

Projekt ma na celu uproszczenie procesu tworzenia oprogramowania, m. in. poprzez ujednolicenie modelu danych w aplikacji i bazie danych. Celem jest traktowanie obiektów w bazie w sposób identyczny z obiektami lokalnymi trzymanymi w bazie danych. Baza danych powinna stanowić integralną część systemu, a nie być jej wyizolowanym fragmentem (rys. 1).

Ważnym elementem systemu jest język zapytań. Obecnie wykorzystywane języki wymagają warstwy ORM, aby zmapować tabele bazy danych na obiekty po stronie aplikacji. Z drugiej strony, OQL wydaje się trzymać zbyt silne więzy z SQL-em, przez co wydaje się nie pasować do obiektowych realiów. Celem MUTDOD jest integracja języka zapytań z językiem programowania w taki sposób, aby dla programisty problem składowania obiektów był transparenty. Język ma umożliwić korzystanie z lokalnych obiektów w ten sam sposób co obiektów w bazie danych, bez konieczności mapowania modelu klas bazy danych po stronie klienta. Różnice między językami zostały przedstawione w tabeli 1.

Kolejnym ważnym elementem jest metamodel danych. W MUTDOD bazuje na metamodelu zaproponowanym przez ODMG. Jednak jest on rozszerzany, aby zawierał informacje o bezpieczeństwie, rozproszeniu, a także, aby współpracował ze współdzielonym modelem danych. Uproszony model metamodelu został przedstawiony na rys. 2.

Unifikacja modelu danych wymaga, aby na stronę klienta przenieść dwa etapy przetwarzania zapytania: optymalizację i interpretację zapytań, co zostało zaprezentowane na rys. 3. Środowisko klienckie MUTDOD automatyzuje ten proces.

Addresses

Miłosz GÓRALCZYK: Wojskowa Akademia Techniczna, Wydział Cybernetyki,
ul. gen. Sylwestra Kaliskiego 2, 00-908 Warszawa, Poland, milosz.goralczyk@gmail.com.
Jarosław KOSZELA: Wojskowa Akademia Techniczna, Wydział Cybernetyki,
ul. gen. Sylwestra Kaliskiego 2, 00-908 Warszawa, Poland, jkoszela@wat.edu.pl.