Piotr Habela, Krzysztof Kaczmarski,
Hanna Kozankiewicz, Michał Lentner,
Krzysztof Stencel, Kazimierz Subieta

Data-Intensive Grid Computing Based on Updatable Views

Nr 974

Warszawa, maj 2004

Abstract

In this report we propose a new approach to integration of distributed heterogeneous resources on the basis of a canonical object-oriented database model, a query language and updateable database views. Views are used as wrappers/mediators on top of local servers and as a data integration facility for global applications. Views support location, implementation and replication transparency. Because views are defined in a high-level query language, the mechanism is much more abstract and flexible in comparison to e.g. CORBA or Web Services. The report presents a short introduction to the query language SBQL and updateable views. It also presents the architecture of the grid network based on updateable views, and simple examples illustrating the mechanism. In the report we also show how the mechanism can be used in peer-to-peer networks. We shortly describe the process of grid design and development. Finally, we present some issues related to grid metadata.

Key words: grid computing, Stack-Based Approach, updateable views

---

Data-intensive Grid Computing oparty na aktualizowalnych perspektywach

Streszczenie

W raporcie proponujemy nowe podejście do integracji rozproszonych, heterogenicznych zasobów. Podejście to oparte jest na kanonicznym obiektowym modelu danych, języku zapytań i aktualizowalnych perspektywach bazy danych. Perspektywy są tutaj używane jako osłony/mediatory, przez które lokalne serwery udostępniają swoje zasoby, oraz jako element, który integruje zasoby i udostępnia je globalnym aplikacjom. Perspektywy wspierają przezroczystość położenia, implementacji oraz replikacji. Perspektywy są zdefiniowane w języku wysokiego poziomu i w związku z tym są one bardziej uniwersalne niż np. CORBA czy Web Services. W raporcie przedstawiamy krótkie wprowadzanie do języka zapytań SBQL i aktualizowanych perspektyw zdefiniowanych na bazie tego języka. Prezentujemy także architekturę sieci grid opartą na aktualizowanych perspektywach i przedstawiamy parę przykładów ilustrujących jej działanie. W raporcie pokazujemy jak prezentowany mechanizm może być wykorzystany w sieciach typu peer-to-peer, a następnie krótko omawiamy jak wygląda proces projektowania i implementacji gridu. Na koniec w raporcie prezentujemy zagadnienia związane z metadanymi w gridzie.

Słowa kluczowe: grid computing, podejście stosowe, aktualizowalne perspektywy

1. Introduction

Grid computing is a new field concentrating on "flexible, secure, coordinated resource sharing among dynamic collections of individuals, institutions, and resources - what we refer to as virtual organizations" [Fost01]. Grid technology is presented as facilities that allow one to integrate many computers into one big virtual computer, which summarizes all the resources that particular computers possess: processing power, main memories, disc storages, data and services. A user of a grid can use idle computer resources of all the connected computers as a support of his/her individual desk-top computer. Sometimes computer resources are compared to electricity, which is supplied by many power stations to the global network and then distributed to many users according to their current needs.

Essentially, grid technology addresses *data-intensive* applications, where distribution of massive data implies distribution and parallelization of computations. Practically all business applications are data-intensive. We can imagine a lot of new applications of a data-intensive grid, such as, for example, integrated information on bank services and loans, integrated on-line hotel information and reservations, integrated health services information, etc. New business quality concerns *transparency of service providers*: the user of a grid first is looking for the service that he/she requires, then makes some transaction, and at last (if any) is interested in the service provider. Many new businesses can be based on the transparency of service providers and with they may create new kind of the information society culture and new opportunities in society development.

For serious businesses the distribution of data and services cannot be done ad hoc in response to the idle computational power. Responsibility, reliability, security and complexity of business applications cause distribution of data and services to be a planned phase of disciplined design processes. Any such process must be preceded by some agreement, contract, standard or law regulation which will determine all the rights and duties of data/services providers and users. This implies further issues for the grid technology such as security, safety, privacy, autonomy, transaction processing, secure data exchange, and other issues.

Data-intensive grid technology can be perceived as continuation of distributed or federated databases, the topic that has been developed for many years in the database domain, see e.g. [HKW90, WQ90, SL90, LM91, She91, SCG91, LM92, TSB92, SS95, GS97, Bell97, RM97, FGLM98, HKRS00]. The

domain of federated databases worked out many concepts that are very close to current grid research, such as various forms of transparency, heterogeneity, canonical data models, transaction procession within distributed federations, metamodels for federations, and others. There are many implemented prototypes, for instance Oasis [RMH99] or IRO/DB [BFHK94]. In this report we take the point of view that the concepts, ideas and solutions worked out by the distributed/federated databases domain must sooner or later be absorbed by the grid computing community.

The paradigm of distributed/federated databases as the basis for the grid technology is materialized in Oracle 10G [Oracle03]. Technologies and applications based on CORBA [OMG02] also follow this paradigm. It is based on *data-centric* (or *object-centric*) view, in which the designers start from developing a canonical federated data/object model and a data/object schema. In case of object-oriented models objects are associated with behavior, i.e. with operations acting on objects, events and other options. The next phase is development of applications acting on such federated (global) data/object resources.

In majority of cases of distributed/federated databases the design assumes the bottom-up approach, where existing heterogeneous distributed data and services are to be integrated into the virtual whole that follows some canonical or federated data/object model. This is just the idea of CORBA, which assumes the definition of a canonical model via IDL and skeletons/wrappers/adaptors that transform local server resources to the form required by the canonical model.

The key issue behind such integration is *transparency*, which means abstraction from secondary features of distributed resources. There are many forms of transparency, in particular location, concurrency, implementation, scaling, fragmentation, replication, and failure transparency. Due to transparency (implemented on the middleware level) some complex features of distributed data/service environment need not to be involved into the code of applications. Thus transparency much amplifies programmers' productivity and greatly supports flexibility and maintainability of software products.

While CORBA achieves some forms of transparency (location, implementation, etc.) there are situations in distributed heterogeneous environments that can hardly be handled by CORBA-oriented software. In terms of federated databases CORBA is based on horizontal partitioning of resources. Each local server, independently of other servers, supplies a bunch of its objects to the federation. Heterogeneity leads to more complex situations.

In some cases one must deal with vertical partitioning, e.g. when a virtual object is composed from parts stored at different servers. Another difficulty concerns redundancies that may appear during such composition, e.g. the attribute *EmployeeJob* for the same employee is stored at several servers. If only one of them is to be imported to the composed virtual object, there will be updating anomalies. Similar difficulty is with replications (mirrors), which are redundancies consciously introduced to increase accessibility, reliability and security. Various heterogeneous environments present more such difficulties.

Another point of view is taken by Open Grid Service Architecture (OGSA), which is based on Web Services. The goal is referred to as the Open Grid Service Infrastructure (OGSI). Transparency is not yet an issue of Web Services, however, some of its forms can be handled through routines acting on UDDI registers. Lack of the explicit transparency concept causes our doubt if the technology is able to support very large grids, which anyway have to hide boundaries between service providers. Although Web Services are based on a kind of a canonical data model, i.e. XML, it is burdened by too many syntactic and low-level details and is too simplistic in comparison to current objects models. Web Services require more concepts and functionalities to be added to cover all the issues that are necessary for business applications (transparency, security, interoperability and efficiency). These new functionalities may present a new layer which can fully cover the Web Services layer.

Other current trends, standards and technologies offer more possibilities to implement data-intensive distributed computing, in particular, component-oriented standards and technologies (e.g. DCOM, .NET, EJB, RMI and J2EE), Peer-to-Peer networks (P2P) and mobile software agents. In this report we do not discuss opportunities offered by the mentioned component-oriented standards and technologies. Concerning P2P networks, this is a very promising idea which can also be considered an architectural variant of a data-intensive grid (especially when there is no a central server). We have some doubts concerning mobile software agents since this idea is heavily mixed up with some controversial anthropomorphic rhetoric. If we remove this rhetoric and express the idea in normal technical terms, till now it is even unclear what is true advantage of the idea and what new quality it offers for designers, programmers and users. Up to now there are no sufficiently advanced business applications that prove its technical and economic significance.

In this report we consider the data/object-centric view of the grid technology, similarly to OMG CORBA. Procedural services, such as programs, procedures, functions and methods, will be considered attachments to objects,

as usual in object-oriented models. The algorithms need data and of course distributed algorithms need distributed data. We propose a universal architecture to handle the data aspect of a grid. In general, we focus on *transparency* issues of the grid network. As in CORBA, we also assume *autonomy* of local data/service providers. Autonomy means that to a big extent service providers do not need to change their current information systems in order to be connected to the grid. Data and services of a particular server are made visible for global applications through a *wrapper* that virtually maps the data/services to some assumed *canonical object model*. Then, all contributing services are integrated into the virtual whole by means of a *updatable views*. They create a virtual object/service store available for global applications. Integration is based on some protocol which makes transparent data transport issues, server locations, object and collection fragmentations, redundancies, replications, etc.

The rest of the report is organized as follows. In Section 2 we describe the issues that must be taken into account during establishment of the grid's architecture. In Section 3 we take a closer look at distributed databases. In Section 4 we present a simple and universal approach to query languages and an object model that is the common ontology for the proposed grid's architecture. In Section 5 we explain a solution to the problem of view updating, which inevitably must be a part of the architecture used to process distributed data. In Section 6 we define a general architecture for the data aspect of the grid. Section 7 presents examples that prove feasibility of this architecture. Section 8 is a special example — we show there how to create a peer-to-peer network in the proposed architecture. In Section 9 we discuss grid metadata. Section 10 concludes and enumerates research opportunities that stem from the proposed architecture.

2. General Issues of the Grid Technology

Grid technology should satisfy the following general requirements: transparency, security, interoperability, efficiency and pragmatic universality. The grid is transparent, if distributed resources can be accessed with the same ease as they reside inside one machine. The grid is secure, if it is immune from random failures and external aggression. The grid is interoperable, if it facilitates the collaboration of heterogeneous platforms, applications, data models and business logics. The grid is efficient, if the processing time inside the grid is acceptable for a wide range of users. Grid technology is

pragmatically universal if (from the designer and programmer points of views) it does not imply limitations with respect to the business that it has to support. The above general requirements are critical in the sense that unsatisfactory solution of any of them undermines feasibility of grid application goals.

Transparency reduces the complexity of design, programming and maintenance effort addressing distributed data and services. In situation when a grid application integrates thousands of heterogeneous resources lack of transparency forces the designers to take into account differences between distributed services which may lead to complexity of the global application, which can make the application goals unfeasible. Transparency allows the programmer to take no care for the location and the organization of distributed data and services. She can use them as if they were local. Thus transparency greatly decreases the cost of the development of an application and significantly increases its portability and maintainability of software products.

There are several forms of transparency, as follows:

- **location and access** transparency (the users need not to care for the geographical location of data and services),
- **concurrency** transparency (the users can access resources simultaneously and need not be aware of the existence of other users),
- **implementation** transparency (the users need not to know how data or service are implemented),
- **scaling** transparency (servers, data and services may be added or removed without any impact on the applications and the users),
- **fragmentation** transparency (the users need not be aware that data is partitioned; the fragments are integrated automatically),
- **replication** transparency (the users need not be aware that data is replicated; replicas may be transparently added or removed to improve the efficiency of processing),
- **failure** transparency (most users can still work after some of the nodes or communication links are broken),
- **indexing** transparency (like in relational databases, the users need not be aware of indices that are introduced to improve performance),
- **migration** transparency (the data and services can be moved without any impact on the applications and users).

Several forms of transparency are assumed in CORBA, in particular location and implementation transparency. Location, fragmentation and replication transparency is realized in distributed databases. Transparency is also the foundation of P2P networks. Transparency is not a property that can be reached ad hoc, but it must be carefully worked out and designed. It requires the establishment of standard APIs, canonical models of data and services and appropriate communication protocols.

The interoperability is less important when the grid is built top-down from scratch. In this case the implementation platform is usually homogeneous and the elements of the grid are interoperable, since they are alike. Interoperability is the critical issue in bottom-up projects. The usual goal of such enterprises is to integrate existing heterogeneous (legacy) systems with incompatible data and services. It is often easy to achieve the transport-level interoperability since most applications can talk over common protocols like shared memory, TCP/IP or HTTP. It is the foundation for higher-level interoperability, which is much harder to realize.

Interoperability and transparency can be easily developed when a common canonical model is simple and natural. For such a case wrappers have to be developed at reasonable cost and time. If the discrepancies among servers are significant, in order to achieve the interoperability the designer of the grid has to pass the information about the servers' local data/services model to the programmers. Such a grid is neither simple nor transparent. In such complex situation interoperability and transparency are in contradiction.

3. Distributed and Federated Databases

In distributed databases data can be fragmented and replicated. Fragmentation can be *horizontal*, i.e. every involved server has the same database schema but a different set of data items. In this case the integration is relatively simple - the programmers must have a way of the transparent access to the union of the data sets residing in the federated servers. *Vertical* fragmentation consists in a partition of objects such that objects' parts can be stored at different servers. Transparent access to a vertically fragmented data requires from the designer to implement the mechanism to glue together the dispersed fragments of an object.

The idea of *replication* is apparently very simple, but implementation of it can imply significant problems related to performance, consistency and

transaction processing. Replication means that the same data is redundantly stored in several places. In this way the access time may be decreased and data is better available. Data are also much more immune against damage. If integration of heterogeneous resources is done bottom-up, the replication is often inevitable and undesirable. In such cases the redundancy (as well as the fragmentation in fact) may be sophisticated and arbitrarily complex. The software of the grid must take advantage of desirable redundancies and neutralize redundancies. Replicas and redundancies seriously hinder transparency and especially transparent distributed data updates.

The integration of heterogeneous geographically distributed data, which is not necessarily under control of some DBMS, requires the development of a specialized middleware. *Wrappers* or *adaptors* (coming from the CORBA and Internet camps) are specialized program modules making it possible to map an internal API to the external API expected by some global application. Usually the mapping raises the level of abstraction. For instance, in CORBA applications the internal API is a collection of some C++ functions, while the external API is an IDL interface mapped to Java. Wrappers are considered non-sophisticated programs for changing data representation, names of functions, and so on. Wrappers are usually dealing with data units of low granularity (e.g. rows rather than tables) and mapping is 1:1, i.e. one "internal" data granule (e.g. a row plus subordinated rows) is conceptually mapped into one "external" data granule (e.g. a CORBA object). A wrapper can be considered a "physical bridge" between local databases and global applications.

Another concept is a *mediator* [Wied92]. It comes from the camp of data warehouses and the integration of semi-structured data. Till now, however, the concept is intuitive and defined pragmatically rather than semantically or theoretically. A mediator has to resolve (semantic, representational, structural, schematic) incompatibilities among data to be integrated. A similar role is assigned to database views. A *view* is a mapping of stored object into virtual ones. There are proposals considering mediators as virtual views [BRU96]. Perhaps the conceptual border between database views and mediators is difficult to define, but there are essential pragmatic differences, which allow considering mediators as a new quality:

- Mediators can act on irregular data and can produce irregular data, while database views usually assume and act on formatted data (e.g. tables) and produce formatted data.
- According to (false) stereotypes of the relational model, a database view is to be defined by a single query (e.g. in SQL). However, current query

languages have the power much below the algorithmic power. Mediators do not assume such limitations. The designers of mediators can use the full power of algorithmic programming languages as well as advanced higher-level methods, including methods attributed to artificial intelligence.

- Mediators can use any resources from the computer environment (e.g. the data written in files) while database views can use only resources stored in a database.
- Database views themselves are stateless: each invocation of a view cannot utilize information from previous invocations of the same view. (Stateful views are referred to as *capacity-augmenting views*.) Stateful mappings cannot be avoided in many global (federated) applications. Mediators can be stateful with no limitations.
- Database views are self-contained named entities defined in the syntax of a query language and stored in the database as identifiable persistent units. In contrast, mediators are programs written in a regular programming language and stored as a part of an application program in the main memory. Thus, mediators have much lower abstraction level and poorer maintenance potential than database views.

4. Stack-Based Approach (SBA)

4.1 Object Model

A successful integration of data and services must be driven by a common canonical data model. The model should include a simple, powerful, semantically clean and consistent query language. In SBA a query language is treated as a kind of a programming language. Thus evaluation of queries is based on mechanisms well known from programming languages. The approach precisely determines the semantics of query languages, their relationships with object-oriented concepts, constructs of imperative programming, and programming abstractions, including procedures, functional procedures, views, modules, etc.

SBA is defined for the general data store. In this report we explain our ideas for the simplest data store model (called M0 [Sub04]) where objects can contain other objects with no limitations on the level of the nesting of objects.

There are also relationships between objects. The presented approach can be easily extended to comply with other notions like classes, inheritance, dynamic roles, etc.

M0 is based on the principles of object relativism and internal identification. Therefore, in the store each object has the following properties:

- Internal identifier (OID) that neither can be directly written in queries nor printed,
- External name (introduced by a programmer or the designer of the database) that is used to access the object from an application,
- Content that can be a value, a link, or a set of objects.

Let I be the set of internal identifiers, N be the set of external data names, and V be the set of atomic values, e.g. strings, pointers, blobs, etc. Atomic values include also codes of procedures, functions, methods, views and other procedural entities. Formally, objects in M0 are triples defined below (i_1, $i_2 \in$ I, $n \in$ N, and $v \in$ V).

- Atomic objects have form $<i_1, n, v>$.
- Link objects have form $<i_1, n, i_2>$. An object is identified by i_1 and points at the object identified by i_2.
- Complex objects have form $<i_1, n, S>$, where S is a set of objects.

Note, that this definition is recursive and it models nested objects with an arbitrary number of hierarchy levels.

In SBA an object store consists of:

- The structure of objects defined above.
- Internal identifiers of *root objects* (they are accessible from outside, i.e. they are starting points for querying).
- Constraints (e.g. the uniqueness of the internal identifiers, referential integrities, etc.).

Example database schema and the corresponding data store are depicted respectively in Fig. 1 and Fig. 2. The store contains the information on departments and their employees. Each employee has personal identification number, name, salary, and job. Each department has the attribute dName. The employees work in departments. Some employees may manage departments.

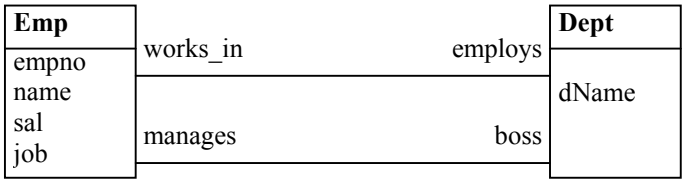

Fig. 1. Example database schema

Objects:

<i_1, *Emp*, {<i_2, *empno*, 123>, <i_3, *name*, "Smith">,
<i_4, *sal*, 3500>, <i_5, *job*, "consultant">, <i_6, *works_in*, i_{20}>} >
<i_7, *Emp*, {<i_8, *empno*, 234>, <i_9, *name*, "White">, <i_{10}, *sal*, 2900>,
<i_{11}, *job*, "developer">, <i_{12}, *works_in*, i_{20}> } >
<i_{13}, *Emp*, {<i_{14}, *empno*, 456>, <i_{15}, *name*, "Blake">,<i_{16}, *sal*, 3200>,
<i_{17}, *job*, "manager">, <i_{18}, *works_in*, i_{20}>,
<i_{19}, *manages*, i_{20}>} >
<i_{20}, *Dept*, {<i_{21}, *dName*, "IT">, <i_{22}, *employs*, i_1>, <i_{23}, *employs*, i_7>,
<i_{24}, *employs*, i_{13}>, <i_{25}, *boss*, i_{13}>} >

ROOT = {i_1, i_7, i_{13}, i_{20}}

Fig. 2. Example of SBA data store

4.2 Environment Stack and Name Binding

The basis of SBA is the environment stack (ES). It is one of the most basic auxiliary data structures in programming languages. It supports the abstraction principle, which allows the programmer to consider the currently written piece of code to be independent of the context of its possible uses. In SBA the environment stack consists of sections that contain sets of entities called binders. A *binder* is a construct that binds a name with a run-time object. Formally, it is a pair (n, i) (further written as $n(i)$) where n is an external name ($n \in$ N) and i is the reference to an object ($i \in$ I). In the following the binder concept will be extended to $n(x)$, where x is any result returned by a query.

SBA respects the naming-scoping-binding principle, which means that each name occurring in a query is bound to a run-time entity (an object, an attribute, a method a parameter, etc.) according to the scope of its name. The

principle is supported by means of the environment stack. The process of name binding follows the "search for the top rule" and it returns the second component of a binder with the given name that is the closest to the top of ES and is visible within the scope of the name. Because names associated with binders are not unique, the binding can return multiple identifiers. In this way we deal with collections.

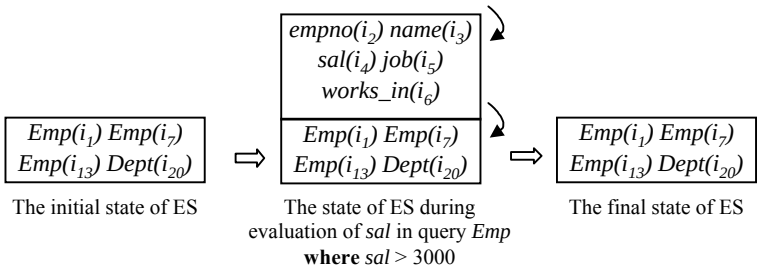

Fig. 3. Example of ES states during evaluation of a query

Some states of ES during evaluation of query *Emp* **where** *sal* > 3000 are depicted in Fig. 3. In this figure solid arrows indicate the order of name binding. At the beginning of the query evaluation ES consists only of the base section that contain binders to all root objects of the data store. There can be other base sections containing binders to the variables of the environment of the system, properties of the current session etc. If a query has no side effects, the state of the stack after the evaluation of the query is the same as before it. During query evaluation the stack grows and shrinks according to the nesting of the query.

New sections pushed onto ES are constructed by means of function *nested*. The function works in the following way:

- For the identifier of a link the function returns the set with the binder of the object the link points to.
- For a binder the function returns that the set with this binder.
- For a complex object the function returns the set of binders to the attributes of the object.
- For structures, nested(**struct**{x_1, x_2, ...}) = nested(x_1) ∪ nested(x_2) ∪ ...
- For other arguments the result of *nested* is empty.

Fig. 3 contains the result of *nested* for i_1 (it returned a set: *empno*(i_2), *name*(i_3), *sal*(i_4), *job*(i_5), *works_in*(i_6)).

4.3 Stack Based Query Language (SBQL)

In this section we present the basis of the Stack Based Query Language (SBQL) [Sub85, Sub87, SBM+93, SKL95, Sub04]. The language has been first implemented in LOQIS system [SMA90, Sub90, Sub91] then in several other systems. SBQL is based on the principle of compositionality, which means that more complex queries can be built of simpler ones. The description of the syntax of queries in SQBL follows.

- A name or a literal is a query; e.g. *name*, *Emp*, 2, "Black".
- $\sigma\ q$ is a query, where σ is a unary operator and q is a query; e.g. *sum*(*sal*), *sin*(*x*).
- $q_1\ \tau\ q_2$ is a query, where q_1 and q_2 are queries and τ is a binary operator; e.g. 2+2, *Emp*.*sal*, *Emp* **where** (*sal* > 2000).

Therefore, in SBQL we can construct complex queries by composing them from simpler ones using unary and binary operators. With the exception of typing constrains the operators are orthogonal.

In SBA we divide operators into two main groups: algebraic and non-algebraic. In the following we describe the difference.

Algebraic Operators. The operator is algebraic, if its evaluation does not require to use of the environment stack. The evaluation of the query $q_1\ \Delta\ q_2$ where Δ is algebraic operator looks as follows. Queries q_1 and q_2 are evaluated independently and the final result is a combination of these partial results depending on the semantics of operator Δ. Note that the key property of algebraic operators is that the order of evaluation of q_1 and q_2 does not matter.

Algebraic operators include string comparisons, Boolean and numerical operators, aggregate functions, operators on sets, bags and sequences (e.g. the union), comparisons, the Cartesian product, etc.

Non-algebraic Operators. If the query $q_1\ \theta\ q_2$ involves a non-algebraic operator θ, then q_2 is evaluated in the context determined by q_1. Thus, the order of evaluation of subqueries q_1 and q_2 is significant. The query $q_1\ \theta\ q_2$ is evaluated as follows. A subquery q_2 is evaluated for each element r of the collection returned by q_1. Before each such evaluation ES is augmented with a new scope determined by *nested*(*r*). After the evaluation the stack is popped to the previous state. A partial result of the evaluation is a combination of r and the result returned by q_2 for this value; the method of the combination depends

on θ. Finally, these partial results are merged into the final result depending on the semantics of operator θ.

Non-algebraic operators include selection (operator where), projection/navigation (the dot), dependent join, quantifiers (for all, for any) etc.

Examples of Queries in SBQL. Here we present examples of queries in SQBL addressing the example database shown in Fig. 2.

- Get employees earning more then 30000:

  Emp **where** *sal* > 30000

- Get the name of the boss of IT department:

  (*Dept* **where** *dName* ="IT")**.***boss***.***Emp***.***name*

- Get the departments that have employees that have salary higher than the boss of the department:

  (*Dept* **as** *d*) **where** ((*d***.***employs***.***Emp* **as** *e*) ∃ (*e***.***salary* > *d***.***boss***.***Emp***.***sal*))

4.4 Procedures in SBQL

SBA supports ordinary procedures and functional procedures (functions). Procedures can be defined with or without parameters, can have side effects, local environment, and can be recursive. There are no restrictions on the computational complexity of procedures. The mechanism of the procedure call in SBA is the same as the mechanism of the procedure call in programming languages. When a procedure is called, the environment stack is augmented by a new section with the local environment of the procedure and its parameters. Next, the body of the procedure is executed. Then, if the procedure is functional its result is returned. Finally, the section with the environment of the procedure is removed from the top of the environment stack. Results of functional procedures belong to the same category as results of queries and therefore, they can be called in queries.

Here is an example of a function in SBA.

```
proc wellPaidEmployeesOfDept(a) {
        create local avg_sal(avg((Dept
                                  where dName = a ) . employs . Emp . sal));
        return ((Dept where dName = a) . employs . Emp
                                                        where salary > avg_sal);
}
```

This function returns the information on well-paid employees of the given department. The name of the department is the parameter of the function. The function has local variable *avg_sal* equal to the average salary in the indicated department. This function can be called in the following query:

wellPaidEmployeesOfDept("IT") **.** (*name*, *sal*)

This query returns names and salaries of all employees of IT department that earn more then the average in this department.

In majority of approaches views are treated as first-class functional procedures and results of functions are used as l-values in updating statements. However, such an approach is inconsistent in many cases and may lead to warping user's intention. The problem of view updating was discussed in detail in [KLPS02]. In SBA an alternative approach is proposed, in which the definer of a view can explicitly determine how updating operations on this view are to be performed.

5. Updatable Views

5.1 The Model of Views

Updatable views are a very powerful tool that can help in achieving transparency. If they are universal, i.e. every view can be made updatable, then the virtual objects presented by the query are indistinguishable from the stored objects. The problem is known for decades and has a lot of limited solutions. In relational databases the only feasible technique to build updatable views is to use INSTEAD triggers. An advantage of SBA is that it facilitates the development of a much more universal solution to the problem of view updating. The presented idea is the very foundation of the architecture described in Section 6.

The approach to updatable views for SBA is presented in [KLS03a, KLS03b, KLS03c, KLS03d, KLPS02]. The idea is to augment the definition of a view with the information on users' intents with respect to updating operations. The first part of the definition of a view is the mapping of the stored objects onto the virtual objects, while the second part contains redefinitions of operations on virtual objects. The definition of a view may also contain other elements like definitions of subviews, procedures, etc.

The first part of the definition of a view has the form of a functional procedure. This functional procedure returns entities that we call *seeds*. Seeds unambiguously identify virtual objects and are parameters of operations on virtual objects. These operations are determined in the second part of the definition of the view. We distinguished four generic operations that can be performed on virtual objects:

- *delete* removes the given virtual object,
- *dereference* returns the value of the given virtual object,
- *insert* puts an object into the given virtual object; it has one parameter—the object to be inserted,
- *update* modifies value of the given virtual object; it has a parameter—the new value to be assigned.

Definitions of these operations are procedures that override the default operations that can be performed on a virtual object. The definer of a view has to decide which of the above-mentioned operations should be applied to given virtual objects. If an operation is undefined, it is forbidden. The operations are assigned fixed names. These are respectively *on_delete*, *on_retrieve*, *on_insert*, and *on_update*. The names of the parameters of operations can be arbitrary chosen by the definer of the view depending on the business meaning of operation.

The definition of a view can also contain definitions of other procedures that can be called by its user. Only procedures *on_delete*, *on_retrieve*, *on_insert*, and *on_update* have special semantics. Other procedures are treated like ordinary procedures inside a complex object. The definer of the view can also introduce other subobjects like internal state variables and nested views (see Section 5.3).

Here is the definition of an example view:

```
create view employeesOfITDeptDef {
        virtual objects employeesOfITDept     {
                        return (Emp where works_in . Dept . dName = ”IT”) as e; }
        on_retrieve do { return e . name;  }
        on_update new_name do{  e . name := new_name;  }
        on_delete do{  delete e;  }
}
```

The view returns information on all employees of the IT department. It defines the following operations: dereference, which returns the name of the

given employee, the update, which sets the name of an employee to the new value, and delete, which removes an employee from the database. The operation *on_insert* is not defined and thus it is not allowed. An example call of the view may be:

for each *employeesOfITDept* **as** *emp* **do** *emp* := *capitalize*(*emp*);

The call capitalizes names of all employees of the IT department. Note that the assignment (:=) in this expression calls procedure *on_update* of the view. When the parameter *emp* of *capitalize* is evaluated, the procedure *on_retrieve* is called.

In comparison to classical views (eg. SQL) SBA assumes that the name of the definition of a view is different from the name of virtual objects that are defined by this view. In this report we assume a simple naming convention where the name of the definition of a view always has suffix *Def*, e.g. *employeesOfITDeptDef*.

When a view is defined, on the bottom of ES two binders appear: one with the name of the view and the other with the name of virtual objects. Both these binders are necessary, because the first one is used to locate the definition of the view and the second is required to access virtual objects.

5.2 Virtual Identifiers and Calls of a View

The mechanism needs to distinguish between stored and virtual objects. Therefore we introduced the notion of a virtual identifier. It is defined as a triple:

<flag "I am virtual", the identifier of the def. of the view, seed>

This form of the virtual identifier tells the system that it deals with a virtual object. The system also knows the identifier of the view that owns this virtual object and the seed that was used to generate this virtual object. The notion of the virtual identifier is further generalized (see Section 5.3) to support nested views.

Let us now analyze a scenario of the updating operation on a view. First, a user invokes the view in a query. The invocation returns a set of virtual identifiers. Next, when the query engine tries to perform an update, it recognizes that it deals with the virtual object. Therefore, it takes the updating procedure from the definition of the view and performs it. Finally, the control is passed back to the user program.

5.3 Nested Views

The presented mechanism supports nesting of views. Again the approach is based on the principle of relativity. Hence, independently of the hierarchy level of a view, its definition has the same syntax, semantics and pragmatics (rules of use) as other views.

In a nested view there might be a need to access seeds of the surrounding views (in order of nesting). Therefore, we introduce an extension of the virtual identifier. Now, it has the following form:

<*flag "I am virtual"*, (*the identifier of the definition of the view*$_1$, *seed*$_1$),
(*the identifier of the definition of the view*$_2$, *seed*$_2$),
...,
(*the identifier of the definition of the view*$_n$, *seed*$_n$)>

where (*the identifier of the definition of the view*$_1$, *seed*$_1$) refers to the most outer view, (*the identifier of the definition of the view*$_2$, *seed*$_2$) refers to one of its subviews and (*the identifier of the definition of the view*$_n$, *seed*$_n$) refers to the currently processed view.

When a virtual identifier is processed by a non-algebraic operator, the sections with the results of *nested*(*seed*$_1$), *nested*(*seed*$_2$), ..., *nested*(*seed*$_n$) are pushed onto ES in this order (the section with *nested*(*seed*$_1$) is pushed as the first one) In this way we pass the seed to all the (dereference, update) procedures that are defined for the view. Next, the stack is augmented by another section containing binders to all subviews of the view identified by the identifier of the definition of the view.

Here is an example of a nested view:

```
create view EmpDeptDef {
        virtual objects EmpDept { return Emp as e; }
        create view EmpNameDef {
                virtual objects EmpName { return e as n; }
                on_retrieve do { return n.name; }
        }
        create view DeptNameDef {
                virtual objects DeptName { return e.works_in.Dept as d; }
                on_retrieve do { return d.dName; }
                on_update newDept do{
                        e . works_in := &(Dept where dName = newDept);}
        }}
```

The view returns information on employees and their departments. The example calls of this view might be:

- Get names of all employees of the IT department:

  (*EmpDept* **where** *DeptName* = "IT") **.** *EmpName*

- Move Smith to the "Help Desk" department:

  (*EmpDept* **where** *EmpName* = "Smith") **.** *DeptName* := "Help Desk"

- Change name of "Smiht" to "Smith":

  for each *EmpDept* **where** *EmpName* = "Smiht" **do** *EmpName* := "Smith";
  This call is incorrect since the subview *EmpNameDef* does not define an updating operation.

5.4 Views with Parameters and Recursive Views

The presented approach also supports views with parameters. Similarly to procedures in SBA the parameter of a view can be a query. Various methods of passing parameters (e.g. *call-by-value* and *call-by-reference*) may be adopted. The mechanism of passing parameters to the body of a view is analogical as passing parameters to the body of a SBA procedure (i.e. through the environment stack).

This approach also supports recursive views. It is a side effect of SBA, which abstracts pieces of code from contexts of their possible uses. In SBA views are entities similar to procedures. Similarly to procedures in case of views, all volatile data created by a view is put into sections of ES. In this way each call of a view is independent from calls of other views. That means that recurrence in fully supported.

6. Architecture of Grid Applications

6.1 Overview

Our strategic approach to grid technology assumes developing a universal idea and then optimize and specialize it in order to tailor it to the specific needs. The opposite approach i.e. to create niche solutions and then trying to generalize them into a common framework usually leads to eclectic, non-homogeneous and non-universal concepts.

We propose a universal architecture for the database aspect of a grid. The heart of this idea is *the global virtual object and service store* (we will use also a shorter term i.e. the *global virtual store*). The global virtual store consists of virtual objects that map objects/data stored at local servers. The store keeps also addresses of local servers. The store is defined through the query language SBQL and updatable views defined in SBQL. *Global clients* are applications that send requests and queries to the global virtual store.

The *global schema* is a collection of definitions of data and services provided by the global virtual store. Application programmers are the main users of these definitions (also the grid organizer uses it; see below). The programmers make use of them, while they are creating global clients. The global schema is agreed upon by a consortium (in particular, by a single company) that funds creation of the grid.

The grid offers services and data of *local servers* to these global clients. The *local schema* defines data and services inside a local server. The syntax and semantics of these schemata as well as the natures of the data and services can be very distinct at each local server. They can be written in e.g. OMG IDL, WSDL and ODL. However, this schema is invisible to the grid.

The first step of the integration of a local server into the grid is done by the administrators of local servers. They have to define *contributory schemata* which must conform to the global schema (we will describe this conformance later in this report). A contributory schema is the description of the data and services contributed by the local server to the grid. The local server's administrator also defines *contributory views* that constitute the mapping of the data and of the local server to the data and services assumed for the grid.

The second step of the integration of local servers into the grid is the creation of *global views*. These views are stored inside the global virtual store. The interface of them is defined by the global schema. They map the data and services provided by the local servers (in the contributory schema) to the data and services available to the global clients (in the global schema).

Such an idea of a grid would be useless if we had not had updatable views (see Section 5). The global clients not only query the global virtual store but also request updates of it. The same concerns the grid that not only queries the contributory views but also updates data of it.

Note that the view in SBA are just complex objects, thus they can contain methods and procedures as well as the local variables that store the state of these views. Therefore our views can offer the same facilities as CORBA

objects and Web Services. The are no limits for that, since the expressive power of SBQL (see Section 4.3) is the power of a universal programming language.

The *communication protocol* is the collection of routines used in the definition of the global views. It contains the functions to check e.g. the state (up or down) of a local server and the access time to a local server.

The global views are defined by the *grid designer*, which is a person, a team or software that generates these views upon the contributory schemata, the global schema and the integration schema. The *integration schema* contains additional information how the data and services of local servers are to be integrated into the grid. The integration schema does not duplicate the definitions from the contributory schemata. It holds only the items that cannot be included in the contributory schemata, e.g. the way to integrate pieces of a fragmented object the relationships among local servers that they are not aware of. The integration schema is used during the creation of views of the global virtual store.

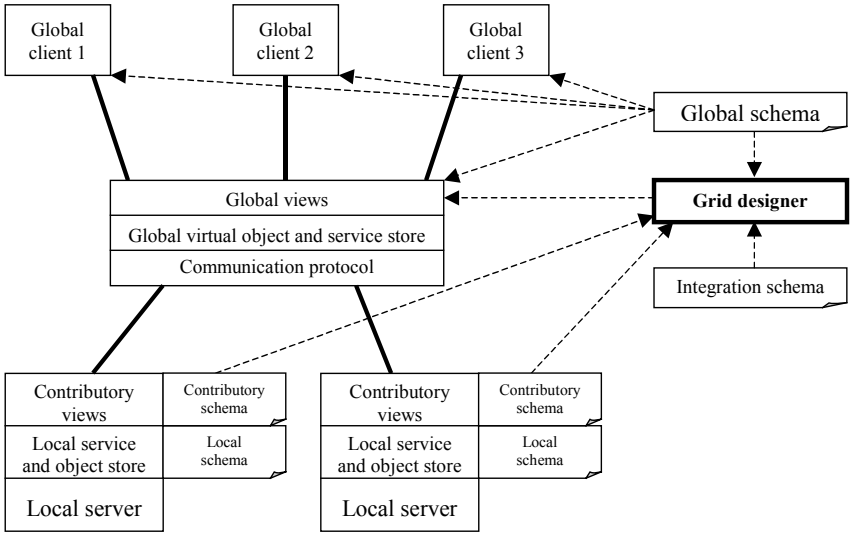

Fig. 4. The architecture of the grid

Fig. 4 presents this architecture. Solid lines represent run-time relationships (queries, requests and answers). Global clients request services of the global virtual store. The global virtual store requests services of the local

servers. These are the only run-time relationships. The association with the grid designer is not run-time, but it is in force only during creation (definition) of global views.

Dashed arrows stand for the definition relationships i.e. associations that are employed during creation of software (development and compilation) and the global views. The programmers of global applications (global clients) use the global schema. The global views conform to the global schema. The grid designer store uses the contributory schemata, the global schema and the integration schema.

6.2 Changes in Object Model

The global virtual store keeps the addresses of local servers in the form of triples:

(*oid*, *server's name*, *server's address*)

Such a triple is an add-on to the object model M0 (see Section 4.1): *oid* is the internal identifier inside the global virtual store, *server's name* is a human-readable name (e.g. Warsaw) and *the server's address* is the description of the physical location of the server (e.g. its IP or URL). We call such a triple a *server link object*.

A server link object looks like a link object. However, when someone navigates into it, it behaves like a complex object. Let us assume that we have server link object (i_W, *Warsaw*, *IPWarsaw*). Conceptually when such an object identifier i_W is navigated into (e.g. when the query *Warsaw.Emp.name* is evaluated), the environment stack is augmented with all root objects of the local server located at *IPWarsaw*. More formally the value of nested(i_W) is equal to the set of all root objects stored by the server identified by i_W. (in this case server *IPWarsaw*). Of course this process must be the subject of the careful optimization. Conceptually all the root objects migrate onto the global virtual store's environment stack. Physically however local query engines can evaluate certain parts of queries (like *Emp.name* in this case).

In most cases it is the grid's administrator who is entitled to define the server link objects.

When object references fall onto the top of the global store's environment stack, they are no longer sufficient to identify objects. The local object identifiers are unique only inside local servers. Thus when a local object identifier i_O finds its way onto the global store's environment stack, it is

implicitly converted to *global object identifier* i.e. pair (i_{LS}, i_O) where i_{LS} is the identifier of the server link object.

Therefore the binders on the global store's environment stack can hold the identifiers of server link objects, the global object identifiers and possibly references of necessary local objects of the global virtual store.

7. Examples

7.1 Horizontal Fragmentation

As the first simple example let us consider a horizontally fragmented database consisting of the local servers located in Warsaw, Cracow and Gdansk. The server link objects inside the global virtual store look as follows:

(i_W, *Warsaw*, *IPWarsaw*)

(i_C, *Cracow*, *IPCracow*)

(i_G, *Gdansk*, *IPGdansk*)

Local, Global and Contributory Schemata. The global schema and all the contributory schemata and the local schemata are the are the same. They are presented in Fig. 5. In this case creation of contributory views is trivial - they are just the identity mapping.

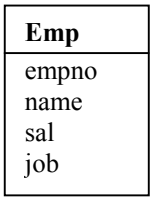

Fig. 5. The schema "Emp"

Integration Schema. Field *empno* is the unique identifier of all employees. There are no duplicates of *empno* in local databases. The global database is the virtual union of all three databases. This information is not included into particular contributory schemata. It is rather a global rule that can appear only in the integration schema.

To define the global view we take a look on the conceptual view object i_C. Physically it is stored as a triple shown above, but when one navigates into object i_C, it appears to be the complex object from Fig. 6.

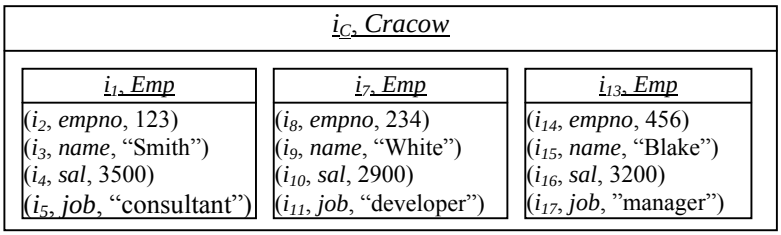

Fig. 6. The server link object as a complex object

The global views have access to the server link objects. Thus all the objects possessed by the local servers are available to the definer and the run-time environment of the views. Of course the definitions of the views will reference the names of the local servers, but the names will be invisible to global clients. The view hides these names from the global clients. They constitute an intermediate level of abstraction that separates concerns of the server and the global clients.

The Global View:

```
create view MyEmpDef {
    virtual objects MyEmp {
        return   ((Warsaw.Emp as p) ∪ (Cracow.Emp as p) ∪
                                                (Gdansk.Emp as p)) }
    on_retrieve do {
        return p.( deref(empno) as empno, deref(name) as name,
                   deref(sal) as sal, deref(job) as job) }
    on_delete do { delete p; }
    on_insert newObj do { insert newObj into p;}
    on_update newName do { p.name := newName; }
}
```

As one can expect the set of seeds of the virtual objects is the union of global object identifiers. The view implicitly delegates the updates to appropriate servers.

The definer of the view has used implicitly several routines of the communication protocol. These were the navigation ("**.**"), **insert**, **delete** and

update ("**:=**"). Note however that she has just called these operations as if they were local. Inside the global virtual store the implementations of the navigation, insert, delete and update must take into account that these routines are in fact elements of the communication protocol. It is true because some of the object identifiers are global (they are pairs of the identifier of a sever link object and the identifier of a local object) and the access to such objects is a remote operation.

The seeds of virtual objects are encapsulated. Therefore global clients cannot see the name of the server that owns the origin of the seed of a particular virtual object. The global clients just reference the name of the view and the name of the attributes, like in the following query:

MyEmp **where** *name* = "Smith"

7.2 Horizontal Fragmentation with a Replica

As in Section 7.1 we have three local servers in Warsaw, Cracow and Gdansk. The global schema and the contributory schemata are as previous, (Fig. 5), but now the local server in Cracow is a replica of the local server in Warsaw.

Integration Schema. It is a bit broader now:

- The local servers in Cracow and Warsaw contain the same data.
- Data should be retrieved from the replica with the shorter access time.
- Data in Cracow cannot be modified.
- Data in Warsaw can be modified. The changes are immediately reflected in Cracow.

Field *empno* is the unique identifier of all employees. There are no duplicates of *empno* in databases in Warsaw and Gdansk. The global database is the virtual union of the databases in Warsaw and Gdansk.

The information that the server in Cracow is a read-only replica of the server in Warsaw can be included in the contributory schema of either server or both of them as well. Since it was put into the integration schema, we can deduce that both servers are not aware of the replication, which is managed by some external mechanism.

The Global View:

```
create view MyEmpDef {
    virtual objects MyEmp {
        int timeToWarsaw := 1000000;
        int timeToCracow := 1000000;
        if alive(Warsaw) then timeToWarsaw := checkAccessTime(Warsaw);
        if alive(Cracow) then timeToCracow  :=  checkAccessTime(Cracow);
        if min(bag(timeToWarsaw, timeToCracow)) > 100 then {
                exception(AccessTimeToHigh);
                return ∅;
        }
        return   ((Gdansk.Emp as p) ∪
                if timeToWarsaw < timeToCracow   then (Warsaw.Emp as p)
                else  (Cracow.Emp as p) );
    }
    on_retrieve do {
        return p.(deref(empno) as empno, deref(name) as name,
                  deref(sal) as sal, deref(job) as job)
    }
    on_delete do {
        if server(p) = Cracow then
                              delete (Warsaw.Emp where empno = p.empno);
        else delete p;
    }
    on_insert newObj do {
        if server(p) = Cracow then
                insert newObj into (Warsaw.Emp where empno = p.empno);
        else insert newObj into p;
    }
    on_update newName do
        if server(p) = Cracow then
                (Warsaw.Emp where empno = p.empno).name := newName;
        else p.name := newName; } }
```

In this example apart from the core protocol routines (navigation, insert, delete and update) we have to use some auxiliary routines of the communication protocol. These calls are underlined. Functions *alive* and *checkAccessTime* are used to determine, which of the two local servers (Warsaw or Cracow) is to be used when constructing the seeds.

The last of these protocol routines (*server*) is slightly different. This routine reflects the subtlety that the query engine of the global store processes *global* object references (see Section 6.2) that are pairs consisting of the identifier of a server link object and the identifier of a local object. The call to *server* returns the first coordinate of the identifier of the global object, namely the reference to the server link object. The call to this routine must be made, because objects in Cracow cannot be updated. Instead one must update appropriate objects in Warsaw.

Note that we cannot just convert the local identifier of the updated object from Cracow to the local identifier of the equivalent object in Warsaw, because these local databases are autonomous, thus the local identifiers of the objects representing the same employee need not be the same. Both databases may have even distinct structure of the internal identifiers, because for example different vendors provided them. Therefore it might happen that it was impossible that these identifiers were the same. We circumvented it by using attribute *empno* to match equivalent objects from both servers.

7.3 Horizontal and Vertical Fragmentation with Replicas, Heterogeneity and Redundancy

This is the final and most complex example. Consider three local servers from Section 7.1 and another local server named *Kalisz.*

Global Schema. The global schema is presented in Fig. 7.

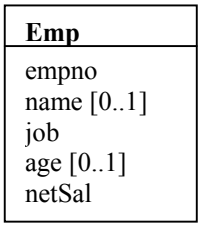

Fig. 7. The global schema

Contributory Schemata. All these schemata are different. They are presented in Fig. 8.

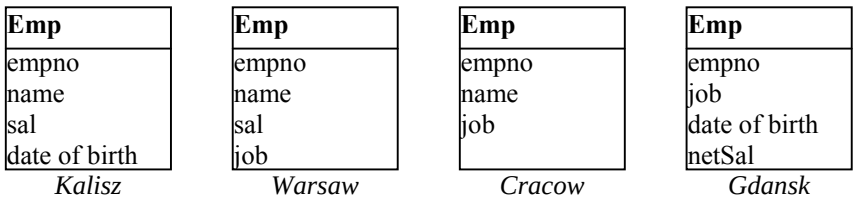

Fig. 8. Local schemata

Integration schema:

- Attribute *empno* uniquely identifies employees.
- Attribute *name* depends functionally on *empno*.
- Server *Cracow* holds a partial replica of local server *Warsaw*.
- An update of any data other than *sal* in *Warsaw* causes the equivalent update in the data held by *Cracow*.
- The users cannot modify data stored by *Cracow*.
- The only records held by *Kalisz* concern the employees of the branch in Kalisz.
- Server *Gdansk* keeps data on employees from Gdansk and Kalisz. It does not store *name* and *sal* but stores *job* and *netSal* that depends functionally on *sal*.

The Global View. In this case the definition of the view would be much more complex. The view definer must resolve all redundancies and replicas. She also must unify different data formats (*age* and *date of birth*) and data with distinct business meaning (*sal* and *netSal*). She must take into account dynamic differences in access times as well as the possibility and the semantics of updates.

SBA and SBQL are universal and we have a general solution to the problem of view updating, therefore with the help of the protocol routines we could code such a view.

The task of the grid designer could be much easier if some of the effort would be done by administrators of local servers. They can simplify the integration by presenting the data in a more uniform way through their contributory views.

8. Grid in Peer-to-Peer Networks

In this section we sketch how our grid concept could be used in peer-to-peer (P2P) networks. P2P is a network that connects nodes that are equal, in contrast to the classical client-server model. Recently P2P is a hot topic and many companies conduct the research on P2P applications. SUN has created JXTA [JXTA, JXTA2] – the set of open-source Java packages that can be used to develop P2P application. Intel supports P2P Working Group that creates standards concerning P2P The most widely-known applications of P2P are Internet file-sharing networks like Kazaa or Napster. In our research we target business applications of P2P like sharing data among big companies.

Usually P2P software is written in languages like Java [JXTA], or C. We suppose that such an application can be written in higher level query languages. Such an approach has the following qualities:

- clarity – one can quickly understand functions of such an application.
- flexibility – one can easily modify an application, so to quickly adapt it to particular requirement.
- shorter development and maintenance time.

In our data-grid approach to P2P, a network consists of peers that can share resources with others and can browse grid resources. Hence, each peer in the grid should install both: software to share resources and software to browse grid resources. In such a P2P network the membership changes dynamically and therefore the applications must automatically react to such changes. In is done in the following way. Each node in this network knows its nearest neighbors (in a special case each peer may know all peers in grid) for instance peers that are in the same LAN. If the grid membership changes (due to a peer failure or peer leave) a peer detects such a situation and update membership information. A peer uses neighbor information when it looks for resources.

In our approach each grid view at any peer stores some information on actual state of the whole grid – we call this set of information on a view state. The state must contain information on neighbors and it can also include other information, for instance, information on particular resource locations. A fragment of example P2P network is depicted in Fig. 9.

In the figure peer P2 has two neighbors: peer P1 and peer P3. The addresses of these two peers are stored as a part of the grid view's state and therefore, peers P1 and P3 can be queried by peer P2 for any resources.

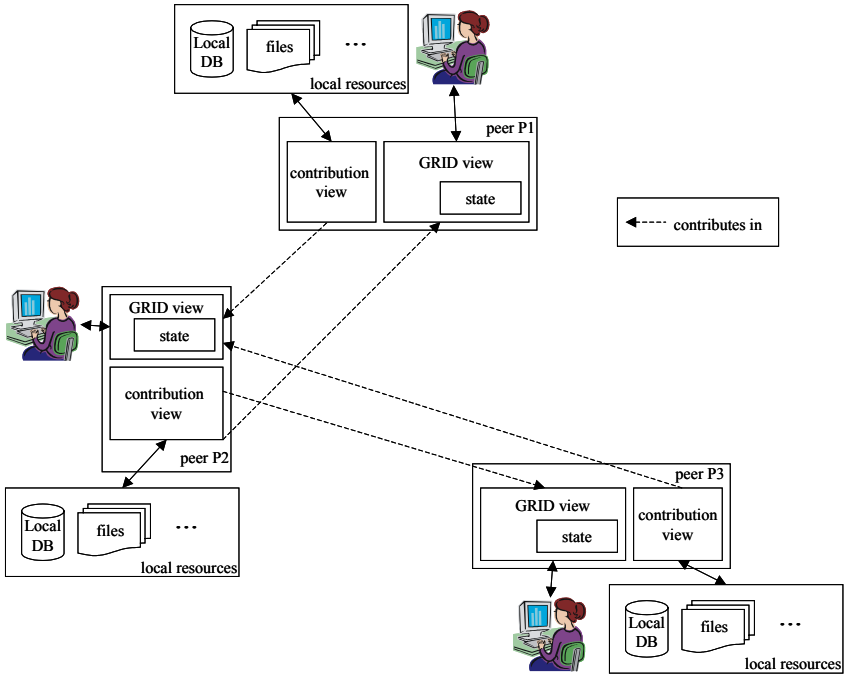

Fig. 9. A part of a grid in a P2P network

We use the term *grid view* instead of the term *global view*, because such a view is located at each peer. The *contributory view* is the view by means of which a peer shares resources with the others. A *peer* is a node in a peer-to-peer network. A *resource* is a data or service that a peer may share with the others

8.1 Views in Peer-to-Peer Grid

In our approach to P2P each peer in the network keeps two kinds of views: a *contribution view* and a *grid view*. A contribution view is a mean by which a peer shares its resources with the others. It may also offer some services to grid views. This view has another important feature: it can act as a client for the grid view at the same peer. In this way requests may propagate in the grid.

A grid view is a view through which a user sees all grid resources. This view may also contain description of services offered by grid like searching

P2P network. All these services are implemented as functional procedures located in the grid view body. The example service is shown in the following section of this report.

An important feature of the grid view is that it has state where peer can keep up-to-date information on (partial) grid state. A main part of peer state is a set containing information on current peer's neighbors. This set is kept as a part of the grid view definition. Elements of this set can be added, removed, or changed dynamically as the grid membership changes. This approach allows one to avoid hard-coding of server addresses into view definition (as it was in the classical grids described in the previous sections of this report).

The state of a grid view can also include other elements, for instance, a local cache with yellow pages (YP). Using this cache a peer would be able to quickly locate a resource/service in grid if it has been used before (and it is still stored in the YP cache). In the approach we assume that grid users do not have direct access to information on a grid view state.

8.2 Rendezvous Peer Concept

A rendezvous peer is a special peer that knows local grid members. This peer has the key role in process of grid joining. If a node wants to join grid it has to first contact Rendezvous Peer (see Section 8.4).

In grids that connect thousands of peers it is not possible that a single peer can handle all information. Therefore, in such large grids it is necessary to set up multiple Rendezvous peers that are responsible for local regions. A rendezvous peer can also handle communication between peers that are separated by firewall or NAT equipment. A local rendezvous should know the other rendezvous in the grid, so it could pass requests to them from local networks. Thanks to multiple rendezvous peers a grid can be scalable.

Division of a grid into local grids could be based on geographical location – e.g. in Fig. 10 a grid connects NY, Warsaw, and Dogville.

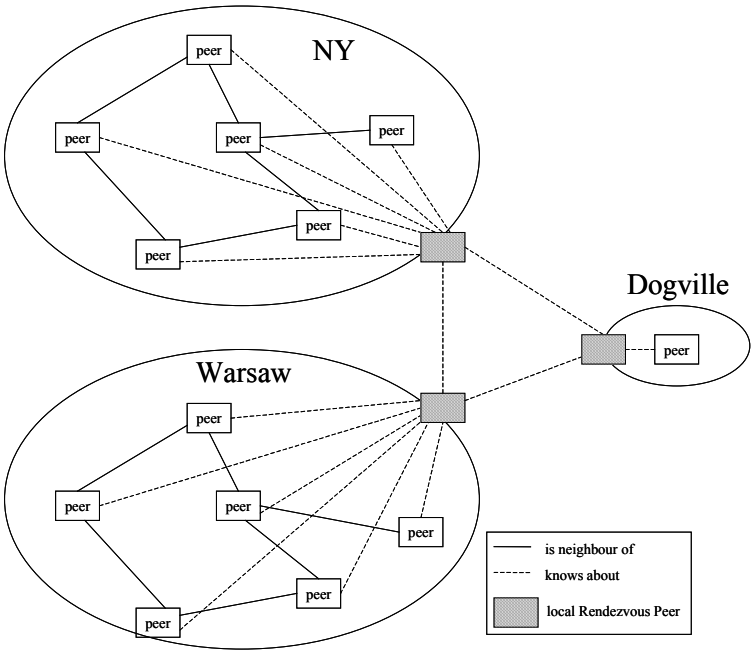

Fig. 10. Example PEER-TO-PEER network with Rendezvous points

8.3 Example Service : Searching the Grid

We show how resource requests propagate in the grid and how the example process of grid resource searching may look like. Assume peer P1 searches for some resource. It sends to all its neighbors search request containing description of the resource. If it wants to discover also resources located in other grid areas it can send this message to Rendezvous peer; the process of messages propagation between Rendezvous peers looks the same as the propagation between normal peers. Each neighbor checks whether it has the required resource. If so, it sends a proper message to P1. Next, the peer passes the request further to all its neighbors, and so on. Each peer that has the requested resource, or knows where it is located, answers directly to P1. Information on sender is a part of search request message. Each peer that gets a

search request may check if information on given resource is not already located in its yellow pages cache.

If peer P1 receives information on resource location, it stores it in its yellow pages in order to be able to use it later.

Search request message should contain the following elements:

- resourceDescription – a structure that describes resource
- sender – information on the peer that looks for given resource (this information is set up automatically)
- ttl – a counterpart of time-to-live used in protocols (this information is set up automatically depending on application settings).

A simplified version of search service description could be as follows (this search procedure is located into the body of a grid view):

```
procedure searchService( sender, resourceDescription, ttl ) {
    if ttl = 0 then return;
        if isInYPCache( resourceDescription ) then
            sender . send_resource_location_info(
                                getResourceLocation( resourceDescription ));
    for each s in GridViewDef . neighbors do
        if s != sender then s . send_search_request( sender,
                                                    resourceDescription, --ttl );
}
```

The procedure first checks if information on required resource is not already located in yellow pages. If it is, the information on resource location is sent to the message sender. Next, the message is passed to all neighbors (besides, the message sender). Due to search message propagation a peer is able to locate multiple locations of resource and can use the resource located in the optimal location. For example, a peer can download resource from multiple locations at once.

8.4 Joining Grid

A peer that wants to join a grid must know the address of the local Rendezvous peer. The peer sends to local Rendezvous JOIN_REQUEST message and as a response it receives JOIN_CONDITION_LIST message that contains requirements that must be fulfilled in order to join. Next, the nodes may negotiate these conditions. Finally, the peer gets from Rendezvous

JOIN_ACCCEPT message that contains a set of new possible peer-neighbors in the grid. The peer tries to connect to these peers by sending them NEIGHBOR_JOIN_REQUEST message. If the peer gets from any node NEIGHBOR_JOIN_ACCEPT message it adds the given node to list of its neighbors.

A peer that receives NEIGHBOR_JOIN_REQUEST message checks if it can accept a new neighbor. If so, it adds the peer to the list of neighbors and sends NEIGHBOR_JOIN_ACCEPT; otherwise it sends NEIGHBOR_JOIN_REFUSE message. It is assumed that if peer P1 is a neighbor of P2, peer P2 is a neighbor of P1.

8.5 Leaving Grid

In order to leave grid a peer should send LEAVE requests to local Rendezvous peer and to all its neighbors. After receiving all LEAVE_ACCEPT messages (or after a given timeout) the peer may leave the grid.

8.6 Resistance to Peer Failures

Such a data-grid has to be resistant to peer failures. Therefore, all peers in the grid periodically send PING messages to their neighbors. If any peer does not answer to this PING message X times in row it is removed from the list of neighbors. In this case a peer that has been removed has to again join the grid.

There could be also introduced some backup Rendezvous peers that is able to take over functions of Rendezvous peer in case of its failure.

9. Grid Modeling and Grid Metamodel

9.1 Grid Database Foundation

An institution or an organization having some business goals in data integration and data sharing can found a grid database. It may be a *consortium*[1]

[1] Later in this section we use term consortium as an institution responsible of grid founding giving necessary regulations and gathering participants for common purpose.

of companies possessing data stores or it may be a new initiative of higher-level cooperation. We recognize the following stages of grid database creation:

1. **Strategic phase**. An initiative of data integration appears for some important business purposes. It may be for example a decision of a consortium of companies or new government regulations for civil administration.
2. **Analysis phase**. It contains analysis of resources and available services. Domain experts and authorities recognize problems: redundancy, data structure mismatch, missing information, and geographical limitations.
3. **Design phase**. It involves negotiations and design of global canonical data structures. Involved institutions find a trade off between many possible solutions, existing and needed structures.
4. **Finalization phase.** Participants sign the final agreement. All of them have certain roles, privileges and duties and accept them. Participating databases will be called *nodes.* Organizations or companies using grid resources will be called *clients.* Often a node becomes a client in the same time.
5. **Implementation phase**. All involved nodes must transform their data to form required by the agreement. Global virtual view for grid clients is created. This phase although the last one from grid's point of view, is in fact more complex for nodes and may be the beginning of many data transformation activities. Nodes must design and create data views (contributory views) transforming their data to the required form. It is clear that for some nodes it may be very simple while complicated for others.

Summarizing, we assume here that cooperating clients or nodes, which must share their data, found a grid database. The type of participation for each node and its role is given from above. After a final agreement (or establishing other obligatory rules), all nodes must conform to their roles. Data from nodes participate in global virtual store in a form described by the global contract.

9.2 Grid Metadata

Metadata Purposes

From the above description of grid foundation process, we may derive two main tasks for metadata. Global data description aims the first one. The

description is created during the design phase and is accepted by all participants during the finalization phase. It describes global grid resources and therefore is called a *global schema*. All clients of a grid database will use it as their description of the main data storage. In addition, a global virtual view creator will use it as a specification of an output format of data served by the global virtual object and services store.

The second purpose is related to node participation. A final agreement establishes certain requirements for all participating nodes of different kinds. This data structure description is called a *contributory schema* because it defines contribution of a node to a grid.

Global Schema vs. Contributory Schema – Data Integration Module

Global schema contains information important from grid client's point of view. That is:

- Global data structures
- Globally available services for certain global objects

Each node shares some of its data that become a part of globally available virtual objects. Thus, it is responsible for data described by fragments of a global schema. Slices of the global schema express its participation. It means that contributory schema which defines node participation in a grid will be a part (more or less complex) of a global schema (see Fig. 11). The conclusion is that even if contributory schema contains some additional information regarding data fragmentation and replications it should have similar language with similar or even identical features to global schema language. This ability will not only allow us to cut global schema into local contributory schemas but will also let us simplify grids' embedding.

It is important to remind that grid's nodes are not just static data stores. They are autonomous databases with their clients and services. Some of these services through grid constructs may become available also for grid clients. Such a client calls an operation on virtual global object but in fact performs one or more operations on objects stored in distributed nodes. Thus, grid metamodel must support operations declarations.

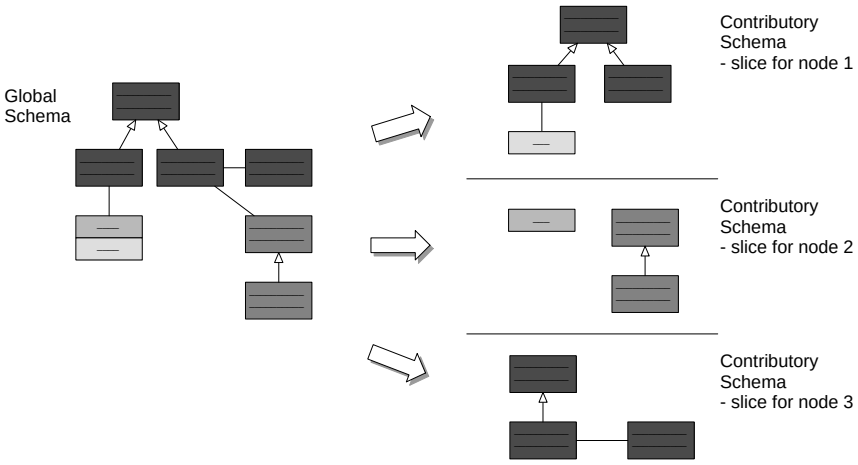

Fig. 11 Decomposition of a global schema into contributory schema slices. Contribution of node 1 is partially overlapping with contribution of node 3. Contribution of node 2 must be composed with contribution from node 1 to get a resulting object.

Global Grid Metadata

As we said earlier, consortium creates data structures and services upon analysis of locally available resources. During the implementation phase grid's designer and node integrators use it as a description of data structures they must adjust to. Such a minimal set of information useful for them should consists of:

- Grid architecture schema (topology of the grid, enumeration of participants, types of all nodes)
- Global schema (description of global virtual data offered for clients by global virtual store)
- Contributory schemas for all node types

A global virtual store, security system, and global updatable view machinery use runtime aspects of global metadata. To the previous design metadata structures, they need adding at least the following information:

- Global users' access privileges, security and authorization methods
- Communication routes, times and access points
- Optimization possibilities

We believe that metadata language should be able to express all of the above information in a short and convenient way. There should exist interfaces

to statistical information used in optimizations (necessary for internal grid engine) as well as programmers' interface offering reflection.

We focus on grid design information (global and contributory schema) since it seems to be the most difficult to define.

Contributory Schema

Apart from defining global data structures consortium assigns categories to all nodes. This decision immediately limits the way a node is participating in a grid. It gets certain data structure description and must transform its data to that form. Of course, some nodes may encounter many problems, while other may have quite an easy task.

A grid does not limit the way a node adjusts its data. It may be any of possible methods, which only fulfill global requirements. However, we propose to use updatable views, which are called contributory views. Output of this view must comply with contributory schema for particular node category.

Thus, contributory schema is an addition to local metadata used normally by node designer, node administrator and local node database clients. It includes information about data retrieval, update possibilities and limitations and should describe at least:

- Which local data structures are participating in the global virtual repository
  (shows output and input format for certain objects, declares available services)
- Possible redundancy of data in nodes (independent databases containing different data but concerning same business information, even not synchronized)
- Existing replications, their update, delete strategies and priorities
- Available strategies and privileges of local data update
- Dependencies between nodes
- Any additional constraints.

The contributory language should be able to express all these information. In fact, we would like it to be as close as possible to local data definition language and global metadata. That simplifies understanding of new concepts and allows us to reuse contributory schemas as local schemas in embedded grid architectures (see next section). However, one must remember that in simple cases contributory schema is only a description of node's output and input data and is highly different from local node's metadata. It is used for different purposes. Local node's data model describes the complete structure of the database. Contributory information is a requirement put on a node by a grid

designer. A node must hide and transform its local data structures in such a form that it complies with these requirements. The outside world should not even know what the transformations of the local data are.

It is important that requirements and constraints described in contributory schema for grid nodes of certain type limit a global virtual store and global view, too. It cannot dynamically and independently change (for example strengthen) any requirement violating node's contract. That could break node's local database, data transfer and make proper functioning impossible. Although, category is assigned to a node by a consortium, it is kind of a two-side contract obliging both a node (which is obligating to submit and accept certain data) and a global view (which is obligating not to change the rules of cooperation).

However, one may imagine a scenario in which a new unknown node but of a known category is attaching a grid in the runtime. This situation is acceptable if only allowed in advance by a grid designer, and described by final consortium agreement. For example, when the appropriate contributory schema and integration scenarios are formulated in more general terms rather than designating particular nodes by their name. This dynamical node attachment is more likely if a grid is using horizontal fragmentation. Such a new node must first introduce itself by showing its category. After attaching as any other node, it must execute services, send and receive data required by the global virtual store.

9.3 Embedded Grids – New Method of Data Integration

Currently, existing DBMS systems supporting grid solutions have only flat architecture. This is the first step in data integration and federated databases, which was already taken some time ago. Unfortunately, flat data integration has a big drawback. Let us imagine that we have several independent DBMS systems and each of them integrates several another distributed data stores. Now, if one would like to integrate these systems should be prepared to redesign all involved nodes. In addition, when some new nodes of a new category appear, they may require redesign of the global store and other participating nodes. This task may be risky, costly and time consuming.

Certain situations may require an incremental business modelling. In such cases, an integration task is decomposed into several levels, which are easier to design. This strategy may be more flexible than a 'big bang' modelling of all

integrations at once. Unfortunately, such modelling is impossible in flat grid architecture.

As an opposite to flat grid solutions we present a new layered grid architecture. This approach removes previous limitations by arranging grid databases in layers or integration levels (Fig. 12). The new grid structure is similar to a snowflake, where some nodes connect to a concentrating point, which becomes a node connected to another concentration point.

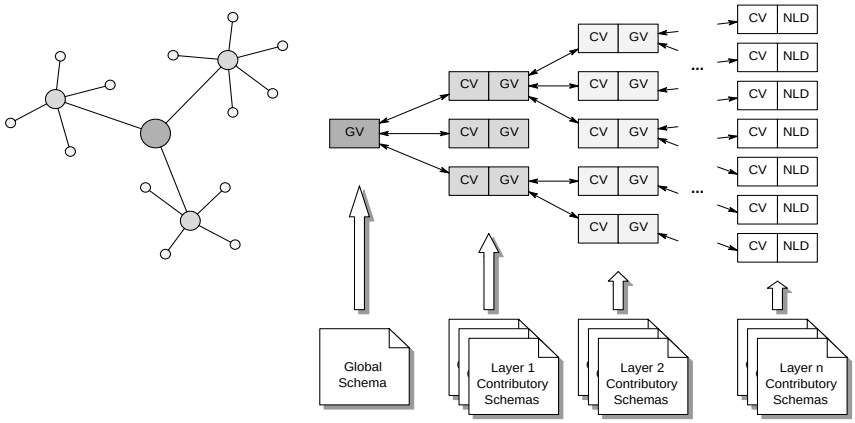

Fig. 12. A general idea of layered grids. CV stands for contributory view. GV stands for global view. NLD stands for node local data. A set of contributory schemas defines each layer.

This kind of architecture is known from data transmission and networking solutions. However, our approach is focused on pure data and services transfer, thus is operating on much higher level. It gives a programmer clear and fast means to integrate data and perform distributed operations. One grid is embedded as a node participating in another grid exactly in the same way as it concentrates its own nodes. Only available resources, market tradeoffs and business strategic decisions limit such integration.

This flexible solution is possible thanks to power of combined contributory view and global view, which constitute a node of a single layer. A global view gathers data from several contributory views of a lower (next) layer. Clients connect to such a concentration point and transparently get access to distributed data. They cannot access data of an upper (previous) layer. Lower layer nodes prepare data for a global view of a previous (upper) layer using

contributory views. Nodes communicate only with a concentration point of an upper layer. The first layer has no contributory view, since it does not participate in any other grid.

Fig. 13 presents a close-up to a single node's global view. A global schema of a node (white in the figure) defines contributory schemas for nodes of a next layer (gray in the figure).

Only a contributory schema from previous layer defines the form of a contributory view, as in previously explained architectures. Contributory view of a considered layer is transforming node's data into a form required by a previous layer, which is expressed by another contributory schema, and with which our layer must agree (white in the figure). It is then limited by participation in another grid. In this way, a single layer is defined upon contributory schemas from previous layer and creates new contributory requirements for the next layer.

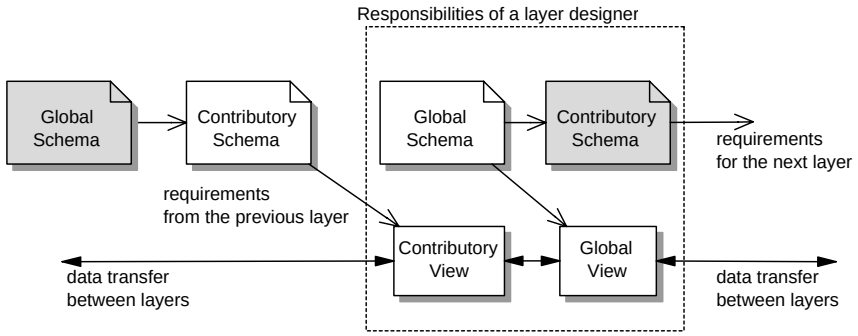

Fig. 13. A close-up to a single node integration – global and contributory schemas.

In some cases, there may be no direct clients and no need for global view in a layer. In such cases, the only usage for a global view is to prepare data for contributory view for the previous layer. Thus, a contributory view may be integrated with global view and global schema of a layer is not necessary (there are no clients to use this schema). Such a solution would be possible if contributory schema and global schema would be expressed in the same language. This is exactly what happens in our approach. All metadata are defined in a common modeling language. Thus, contributory schema may be reused as global schema for a global virtual view.

During grid database development, we may notice two strategies of integration in different phases: *bottom-up* and *top-down.* The first one appears when a consortium analyses existing resources. It gets existing DBMS, replicas and local constraints. Upon this information consortium works out and proposes a global data structure (global schema). It is then an analysis from bottom up. Once, an agreement is signed all participating nodes get global schema and contributory schemas they must fit to. Thus, during implementation phase we have responsibilities and modeling data going from top down.

We may also distinct these two approaches in the first strategic phase. An initiative of data integration may come from distributed nodes, which found a consortium or from a higher-level organization, which forces nodes to consolidate resources.

Summarizing, layered architecture compared to flat grid database has following advantages:

- Data integration may be divided into stages in which designer gathers selected nodes (with certain relationships) and organizes them around concentration points. Each group may be designed separately and independently from others.
- Grid integration layers have closed structure. There is no random access between layers or nodes. It means that a single layer communicates only with nodes from next layer and with one concentration point belonging to a previous layer. As in other closed layer architectures we may easily change the design of a single layer by touching only two closest layers. The impact of changes is highly minimized.
- We achieved a global relativism of grid nodes and concentration points. A concentration point may freely become a participating node in another layer. There is no limitation of grids' embedding.
- There is also no limitation in order of data integration. One may start from integrating only a few nodes. After the first layer is working properly, he or she may start to build another layer around new concentration points. Implementation and testing may be easily spread in time. It is also easy to separate certain milestones with concrete acceptance tests and clear assumptions for each development stage.
- Layered grid architecture may easily follow chosen methodology: bottom-up for existing data integration and top-down for establishing distributed resources.

9.4 Metamodel for Global and Contributory Schemas

This section discusses some concrete aspects of grid metamodel. As it was explained earlier, the most important part of the meta-data for us is the global data model and contributory language. These data will be used in a grid design and so far cannot be uniformly expressed by DDL languages. Identified features of needed metamodel are:

- Ability to declare object data structures
- Ability to declare operations
- Ability to be cut into slices defining contribution of nodes containing redundancy and replication information
- Ability to integrate global schema and contributory schema

The metamodel (based on notions developed for SBA) presented below has all of these features.

In our proposal we attempt to follow as far as possible the popular object models' notions (UML, Java, IDL/ODL), although we avoid introducing secondary notions and try to achieve a higher level of object relativism.[2] An important extension compared to traditional object models is the introduction of dynamic object role notion and dynamic inheritance among objects that is assumed by it.

To illustrate the discussed notions we follow the UML style of metamodel definition. Those constructs are necessary to describe contributing node's database. Note that we are only interested here in externally accessible features programmer needs to know in order to create a contributory view, which is transforming local data structures to required form. Particularly, this document does not deal with implementation details (e.g. the mechanisms providing operations' implementation are not discussed). Thus we do not use the term "class" and instead use the notion of interface to provide the necessary objects' description.

[2] A uniform treatment of all objects, no matter they are primitive or complex. An important consequence is the support for arbitrarily complex (nested) object compositions.

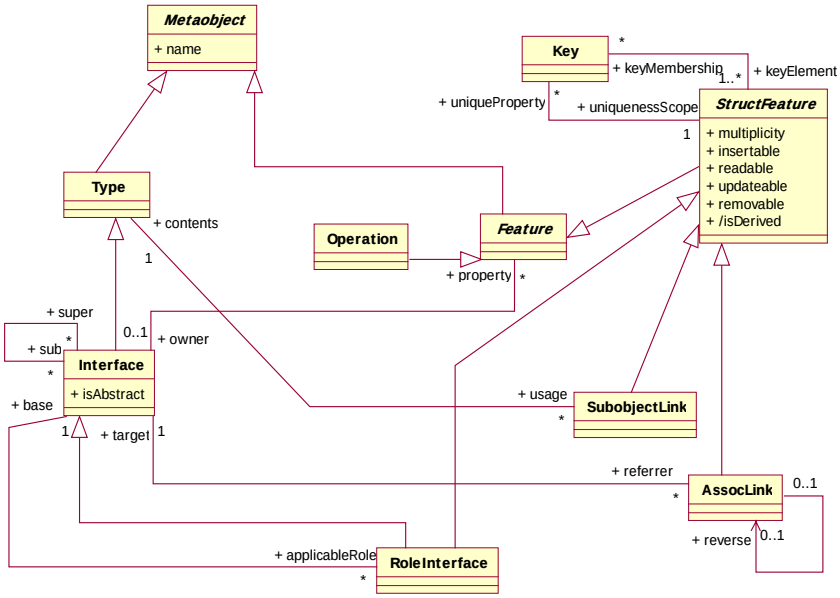

Fig. 14. Database metamodel: the core notions

Fig. 14. shows the core constructs describing database objects. All significant metadata notions that need to be named are derived from the abstract Metaobject [conceptual model] class.

Data items (objects) are described through their type (for primitive values) or by interface (in case of complex objects). Interface allows to define behavior[3] (supported operations) and structural features (subobjects and association links) of their object. The term "subobject" covers the notion of attribute and is used here to emphasize the ability to create compositions of complex objects. For each structural feature its multiplicity is specified. Other attributes describing structural feature indicate, whether the following generic operations: insertion, modification, removal, read are supported for this feature. Associations can be declared as unidirectional (although even in such case hidden opposite direction links would exist to help protecting database integrity).

[3] Operations signature is presented on the next figure.

An important assumption of the presented model is a separation of object definitions from their storage structures. That is, in order to allow storing say *Employee* objects in a database, one needs to provide *Employee* interface definition as well as to define global structural feature to store *Employee* instances. The second task is realized by the structural features distinguished by the lack of interface assigned to them. As a side effect, it also allows to define global (that is – database-scope) procedures.

For interfaces, traditional static generalization-specialization declarations are allowed. Although not further discussed in this document, the dynamic object role notion [JHPS02] was introduced here as one of the fundamental concepts of the assumed object model. Dynamic object role provides a dynamic inheritance mechanism among objects. From the data definition point of view is treated similarly to object's structural features (e.g. a multiplicity of roles connected to a single base object can be defined).

Uniqueness constraints allow specifying single and composite uniqueness keys. Note that if a given interface is used in different places of database structure, for each of this places [designated by appropriate *StructFeature* declarations] a different uniqueness constraint can be used. In other words, each uniqueness constraint concerning complex objects is scoped with respect to particular feature declaration rather than to all instances of a given interface. This definition will be further constrained by the concrete syntax of schema definition language.

Fig. 15 presents operation's signature. This is another place where our object model differs significantly from traditional object models, due to flexibility of queries that may be used as operations' parameters. Procedure's input and output data structures are treated uniformly and can take form of:

- Typed value: a value of predefined primitive type or an instance of schema-defined interface. Such result can be optionally specified as immutable. Moreover, the value can be referenced directly or through a reference link (pointer).
- Binder: a named contents (that may be a structure of any of kinds described in this list) accessible through this name.
- Constructed value: a structure containing one or more binders.

For each case the items may be specified as multiple and (if so) – as ordered or unordered. To follow a structural type matching paradigm, some adjustments would be necessary.

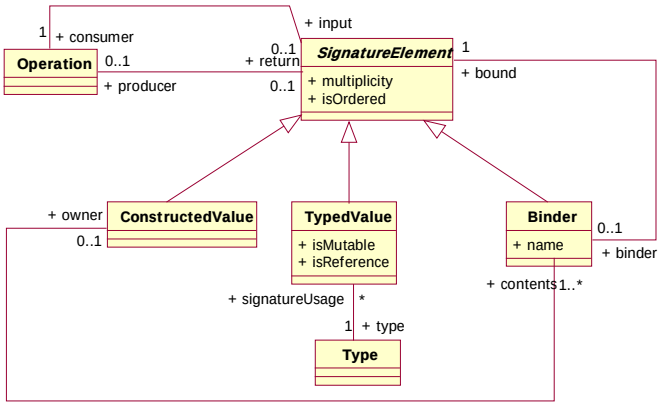

Fig. 15. Database metamodel: operation's signature details

Fig. 16 shows additional declarations that we find important in the context of databases' cooperation, namely – the replication paths. It is possible to identify the foreign databases involved. For each structural feature (global or interface-hosted) it is possible to indicate:

- The foreign databases, to which the request of updating a copy of a given feature is forwarded.
- The foreign databases, from which the requests of updating a given feature can be received.

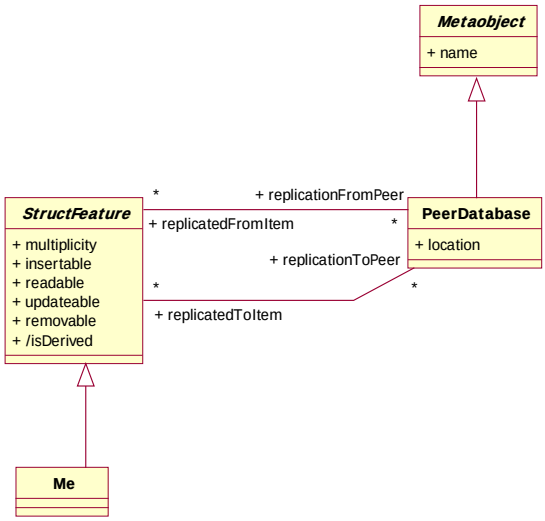

Fig. 16. Database metamodel: the replication paths description

9.5 Data Integration Issues

Procedural Power of the Global Updatable View and Virtual Attributes Descriptors

A grid user operates on global virtual objects according to global schema, which was created by a grid designer. However, global objects are composed of locally available objects. Data spread among nodes is fragmented. Object composition is performed by a global virtual view and is programmed upon global schema, data integration information and other constraints.

If one object is to be composed of several separated objects, there must be a well-defined key that is used to match all of them. This key is a chosen set of its attributes and is called a *composition key*.

From our point of view, it is not important how two objects are composed into one global object. We assume that such a transformation may be always expressed in SBQL queries used in procedural definition of a global updatable view. Integration based on unlimited expressions comparing composition keys is especially useful in case of gathering already existing resources.

A global virtual view programmer must be conscious about all possible types of conflicts between attributes that it is gathering in virtual objects. Some of the most obvious problems are:

- One attribute is existing in two or more places under the same name but contains different data
- Same attribute is existing in two or more places under different names but contain the same data
- A mandatory attribute is existing in only one of the locations and may be missing due to data transmission failures

In these cases the resulting attribute value may be:

- Copied from one selected location
- Required to be equal in all locations (otherwise an exception is thrown or no global object is created)
- Created as a collection of all available values (even having different names)
- Omitted and flagged as missing if no sensible decision can be done

A grid's designer and updatable view programmer must resolve all possible conflicts.

There is one important issue here. A virtual attribute value may be a combination of several values from many objects. In such cases, the view must keep all seeds for such a value to be able to track source objects. Even if all sources are well defined and known updating strategies and other services may be very complicated and in some cases may require a lot of manual programming.

Integration Modelling

During the design phase of a grid it may be useful to show data integration on graphical UML-like diagrams. However, neither UML, nor other graphical languages have enough notions to express such integration. The way objects are composed and description of virtual attributes require new key words.

In the Fig. 17 we present a simple example of such attribute description, which shows that this topic needs further investigations.

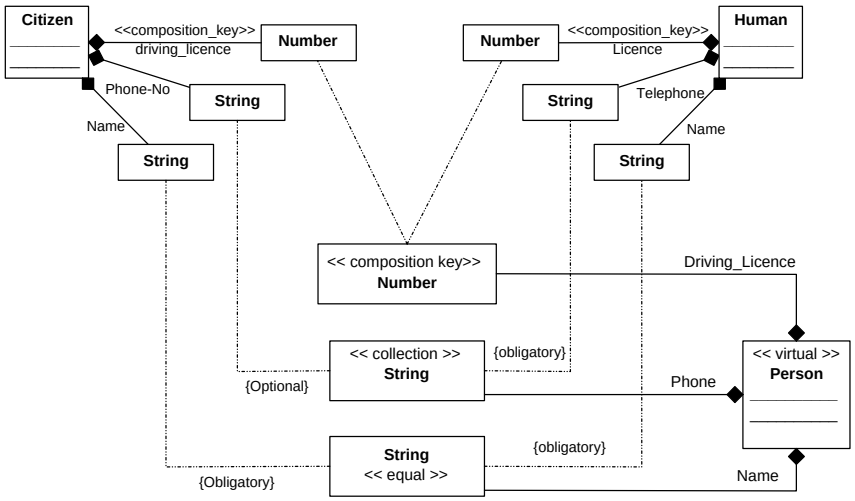

Fig. 17. Complication of data composition on UML-like schema

Business representatives may use a graphical integration schema during analysis and design phase. It is an informal but coherent and compact understandable way to show how distributed data is integrated.

An efficient way of representing data integration and virtual view in UML-like language is an open research problem.

10. Research Opportunities

In this report we proposed a universal architecture for the grid in the data-centered approach. This architecture allows integrating heterogeneous data sources with distinct schemata. To implement the architecture one has:

- to implement updatable views,
- to implement the grid's query engine (with two stacks, a store of server link objects and a translation of local and global object identifiers),
- to establish and to implement the routines of the communication protocol.

Further steps should boost the capabilities of the architecture:

- If we implement automatic synchronizations of the definition of the grid's views with the distributed schemata, we will obtain a bus like ORB or a peer-to-peer network.
- A registration system for data and services (like UDDI or CORBA trading service) is also needed.

The last set of research proposals is concerned with query optimization. In the presented architecture all processing is done inside the global virtual store's query engine. Most queries contain large fragments that can be evaluated by the local servers. The most wasteful operation is the initial navigation into a server link object when all its root objects migrate onto the top of the global store's environment stack. Inevitably the first task will be to optimize this element of the global query evaluation process. Here is not exhaustive list of possible optimizations that can be considered:

- To identify subqueries that a particular local server can execute (it could require to transfer some sections of the global stacks).
- To process subqueries in parallel.
- To partition queries into subqueries that will be evaluated independently by local servers, if the data is vertically fragmented.
- To use the result of the subquery from one server to optimize the processing on another local server.
- To augment views with proxies that speed up the evaluation of queries.
- To build global indices kept by the global virtual store (so far in the proposal the grid stores just the server link objects; after all it could also take care of indices). This optimization has the greatest potential. The idle time of the global virtual store can be filled with indexing and cataloguing the data held by local servers.

11. References

[Bell97] Zohra Bellahsene: Extending a View Mechanism to Support Schema Evolution in Federated Database Systems. DEXA 1997: 573-582

[BFHK94] R. Busse, P. Fankhauser, G. Huck, W. Klas. IRO-DB An object-oriented approach towards federated and interoperable DBMS. Proc. of ADBIS'94, Russia, 1994

[BRU96] Buneman, P., L. Rashid, J. Ullman. *Mediator Languages - a Proposal for a Standard*. Report of an I3/POB Working Group, University of Maryland, April 1996

[FGLM98] Peter Fankhauser, Georges Gardarin, M. Lopez, J. Muñoz, Anthony Tomasic: Experiences in Federated Databases: From IRO-DB to MIRO-Web. VLDB 1998: 655-658

[Fost01] I.Foster, C.Kesselman, S.Tuecke The Anatomy of the Grid: Enabling Scalable Virtual Organizations. International J. Supercomputer Applications, 15(3), 2001.

[GS97] Georges Gardarin, Fei Sha: Using Conceptual Modeling and Intelligent Agents to Integrate Semi-structured Documents in Federated Databases. Conceptual Modeling 1997: 87-99

[HKRS00] Wilhelm Hasselbring, Willem-Jan van den Heuvel, Geert-Jan Houben, Ralf-Detlef Kutsche, Bodo Rieger, Mark Roantree, Kazimierz Subieta: Research and Practice in Federated Information Systems, Report of the EFIS '2000 International Workshop. SIGMOD Record 29(4): 16-18 (2000)

[HKW90] David K. Hsiao, Magdi N. Kamel, C. Thomas Wu: The Federated Databases and System: A New Generation of Advanced Database Systems. DEXA 1990: 186-190

[JHPS02] Andrzej Jodłowski, Piotr Habela, Jacek Płodzień, Kazimierz Subieta: *Objects and Roles in the Stack-Based Approach*. DEXA 2002: 514-523.

[JXTA] www.jxta.org

[JXTA2] B.J. Wilson *Inside JXTA: Programming P2P Using the JXTA Platform*, New Riders, 2002

[KLPS02] Hanna Kozankiewicz, Jacek Leszczyłowski, Jacek Płodzień, Kazimierz Subieta. *Updatable Object Views.* Institute of Computer Science, Polish Academy of Sciences, Report 950, Warsaw, Poland, 2002

[KLS03a] Hanna Kozankiewicz, Jacek Leszczyłowski, Kazimierz Subieta. *Updatable XML Views.* Proc. of Advances in Databases and Information Systems (ADBIS), Springer LNCS 2798, pp. 385-399, Dresden, Germany, 2003

[KLS03b] Hanna Kozankiewicz, Jacek Leszczyłowski, Kazimierz Subieta. *Implementing Mediators through Virtual Updatable Views.* Proc. of the 5th International Workshop on Engineering Federated Information Systems (EFIS), IOS Press, pp. 52-62, Coventry, UK, 2003

[KLS03c] Hanna Kozankiewicz, Jacek Leszczyłowski, Kazimierz Subieta. *Updatable Views for an XML Query Language.* CAiSE FORUM, Klagenfurt/Velden, Austria, 2003

[KLS03d] Hanna Kozankiewicz, Jacek Leszczyłowski, Kazimierz Subieta. *New Approach to View Updates.* Proc. of the VLDB Workshop Emerging Database Research in Eastern Europe, Berlin, Germany, 2003

[LM91] Qing Li, Dennis McLeod: An Object-Oriented Approach to Federated Databases. RIDE-IMS 1991: 64-70

[LM92] Qing Li, Dennis McLeod: Managing Interdepencies among Objects in Federated Databases. DS-5 1992: 331-347

[OMG02] Object Management Group: OMG CORBA™/IIOP™ Specifications. http://www.omg.org/technology/documents/corba_spec_catalog.htm, 2002

[Oracle03] Oracle 10g: Infrastructure for Grid Computing, An Oracle White Paper, September 2003

[RM97] Mark Roantree, John Murphy: An Architecture for Federated Database Metadata. EFDBS 1997: 23-32

[RMH99] M. Roantree, J. Murphy, W. Hasselbring. The OASIS Multidatabase Prototype. ACM SIGMOD Record, Volume 28:1, March 1999.

[SBM+93] K. Subieta, C. Beeri, F. Matthes, J.W. Schmidt „A Stack-Based Approach to Query Languages", Institute of Computer Science, Polish Academy of Sciences, Report 738, Warsaw, 1993

[SCG91] Fèlix Saltor, Malú Castellanos, Manuel García-Solaco: Suitability of Data Models as Canonical Models for Federated Databases. SIGMOD Record 20(4): 44-48 (1991)

[She91] Amit P. Sheth: Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. VLDB 1991: 489

[SKL95] K. Subieta, Y. Kambayashi, J. Leszczyłowski "Procedures in Object-Oriented Query Languages", Proc. of VLDB, pp. 182-193, 1995

[SL90] Amit P. Sheth, James A. Larson: Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. ACM Comput. Surv. 22(3): 183-236 (1990)

[SMA90] K. Subieta, M. Missala, K. Anacki "The LOQIS System. Description and Programmer Manual", Institute of Computer Science, Polish Academy of Sciences, Report 695, Warsaw, November 1990

[SS95] Ingo Schmitt, Gunter Saake: Managing Object Identity in Federated Database Systems. OOER 1995: 400-411

[Sub04] K. Subieta. Theory and practice of object query languages, Polish-Japanese Institute of Information Technology, 2004 (in Polish)

[Sub85] K. Subieta "Semantics of Query Languages for Network Databases", ACM Transactions on Database Systems, Vol. 10, No. 3, pp. 347-394, 1985

[Sub87] K. Subieta "Denotational Semantics of Query Languages", Information Systems, Vol. 12, No. 1, 1987

[Sub90] K. Subieta "LOQIS: The Programming System Having Database Capabilities", Institute of Computer Science, Polish Academy of Sciences, Report 694, Warsaw, October 1990 (in Polish)

[Sub91] K. Subieta "LOQIS: The Object-Oriented Database Programming System", Proceedings of the 1st Intl. East/West Database Workshop on Next Generation Information System Technology, Springer LNCS 504, pp. 403-421, 1991

[TSB92] Henry Tirri, Jagannathan Srinivasan, Bharat K. Bhargava: Integrating Distributed Data Sources Using Federated Objects. IWDOM 1992: 315-328

[Wied92] Wiederhold, G. *Mediators in the Architecture of Future Information Systems*, IEEE Computer Magazine, March 1992

[WQ90] Gio Wiederhold, Xiolei Qian: Consistency Control of Replicated Data in Federated Databases. Workshop on the Management of Replicated Data 1990: 130-132

Content

Pracę zgłosił: Wojciech Penczek

Adresy autorów:

Piotr Habela[#]	habela@pjwstk.edu.pl
Krzysztof Kaczmarski[&]	kaczmars@mini.pw.edu.pl
Hanna Kozankiewicz[*]	hanka@ipipan.waw.pl
Michał Lentner[#]	michal.lentner@pjwstk.edu.pl
Krzysztof Stencel[+]	stencel@mimuw.edu.pl
Kazimierz Subieta[*#]	subieta@ipipan.waw.pl

[*]) Institute of Computer Science, Polish Academy of Sciences
ul. Ordona 21, 01-237 Warsaw, Poland

[#]) Polish-Japanese Institute of Information Technology
ul. Koszykowa 86, 02-008 Warsaw, Poland

[+]) Faculty of Mathematics, Informatics and Mechanics,
Warsaw University
ul. Banacha 2, 02-097 Warsaw, Poland

[&]) Faculty of Mathematics and Information Science,
Warsaw University of Technology
Pl. Politechniki 1, 00-661 Warsaw, Poland

Symbol klasyfikacji rzeczowej: H.2.3, H.2.4, H 2.5

Na prawach rękopisu

Printed as manuscript

Nakład 100 egzemplarzy. Papier kserograficzny klasy III. Oddano do druku w maju 2004r.

Wydawnictwo IPI PAN.

ISSN 0138-0648