

A Multi-Level Architecture for Distributed Object Bases

Markus Kirchberg, Klaus-Dieter Schewe, Alexei Tretiakov
Massey University, Dept of Information Systems
Private Bag 11 222, Palmerston North 5301, New Zealand
Email: [M.Kirchberg | K.D.Schewe | A.Tretiakov]@massey.ac.nz

Key words: Database architecture, distributed database, object data model, fragmentation, communicating agents, multi-level transactions, persistent object store, linguistic reflection, dialogues.

Abstract: The work described in this article arises from two needs. First, there is still a need for providing more sophisticated database systems than just relational ones. Secondly, there is a growing need for distributed databases. These needs are addressed by fragmenting schemata of a generic object data model and providing an architecture for its implementation. Key features of the architecture are the use of abstract communicating agents to realize database transactions and queries, the use of an extended remote procedure call to enable remote agents to communicate with one another, and the use of multi-level transactions. Linguistic reflection is used to map database schemata to the level of the agents. Transparency for the users is achieved by using dialogue objects, which are extended views on the database.

1 Introduction

The amount of data that is collected and stored in databases is steadily growing. In many cases the structure of these data is not adequately reflected in the database schema and query languages. E.g., many databases containing important biological data, are organized more or less as flat files. In addition, the centralization of data in a big central database becomes more and more a handicap for efficient data processing. As data arises at various locations and will be used at various other locations as well, the need for data distribution increases.

As to the structure of the data, the research in the last decade has investigated complex values, i.e., data constructed by various type constructors, and references / links between data — which in fact lead to infinite, yet finitely representable structures. The object oriented datamodel (OODM) from (Schewe and Thalheim, 1993) allows these different aspects to be combined. Starting from an arbitrary underlying type system a schema is defined as a set of classes, each of which combines complex values and references. Thus, the theory of that datamodel can be tailored according to the underlying type system. This has been exploited in (Schewe, 2001) to define a generic query algebra.

Thus, in order to satisfy the identified needs it is a natural idea to develop a distributed database system based on this OODM. The first problem that has to be addressed is the distribution of the data. The fragmentation of OODM schemata have been recently addressed in (Schewe, 2002); the problem of allocation is still open. The result of fragmentation and allocation will be still an OODM schema, but each class will be allocated to exactly one node — or in the case of replication several nodes — of a network. As a consequence the global objects corresponding to the original schema would be represented by several local objects for the fragmented schema.

However, the structure of logical local objects is still complex, whereas efficient storage and retrieval would require to provide just records stored on pages. This implies to further decompose objects as we move closer to the physical storage, so that we obtain multiple levels of objects.

The existence of multiple levels for objects suggests to exploit the concept of multi-level transactions (Beeri et al., 1989; Schewe et al., 2000) for the database functionality. Multi-level transaction scheduler exploit the fact that many low-level conflicts become irrelevant, if higher-level operation semantics are taken into account. The system uses the hybrid multi-level concurrency control protocol FoPL

(Schewe et al., 2000; Kirchberg and Schewe, 2001) as well as the more familiar two-phase locking protocol (Weikum, 1991; Kirchberg and Schewe, 2001). Both are combined with a multi-level recovery system (Schewe et al., 2000; Kirchberg, 2002) based on the ideas from the ARIES system (Mohan et al., 1992).

The multiple object levels are also reflected in the operational system, which exploits the ideas of two-stack machines (Subieta, 1991; Subieta et al., 1993) to implement database functionality, and in particular, to integrate the processing of queries and transactions. However, these machines have to be extended in a way that they can communicate with each other including communication via remote procedure call, run in parallel, are coupled with the transaction manager and the persistent object store, and reflect the operations on higher levels of the multi-level system.

Two more problems have to be addressed. The first one concerns the transformation of high-level queries and operations to the level of communicating agents. For this we exploit linguistic reflection (Stemple and Sheard, 1991; Stemple et al., 1992). We provide a macro language, in which the high-level constructs in transactions such as generic update operations and the high-level algebra constructs for querying can be formulated. In (Stemple et al., 1990) it has been shown how linguistic reflection can be used to expand such macros for the case of query algebra constructs. In (Schewe et al., 1994) linguistic reflection has been applied to expand macros for generic update operations.

The second problem concerns transparency at the user interface. We follow the idea to provide dialogue interfaces based on dialogue objects, which are defined by extended views (Schewe and Schewe, 2000).

Outline. In Section 2 we give a more detailed overview of the architecture. Section 3 then briefly describes the datamodel, the user interface integration, fragmentation and linguistic reflection. In Section 4 we describe more details of the communicating agents, the network communication, and the multi-level transaction management.

2 Architecture Overview

Figure 1 illustrates the multi-level architecture for distributed object bases on each site of a network. As virtually all modern database systems the proposed system has a layered internal architecture. Each layer has an interface that provides services invoked by the layer above it to implement their higher level services. We now introduce these layers in more detail starting with the physical storage.

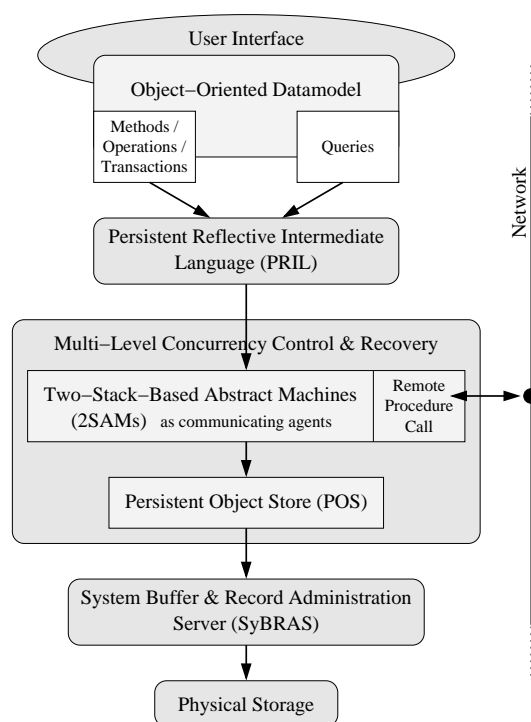


Figure 1: Architecture for Distributed Object Bases

The database can be seen as a collection of physical objects stored on a physical storage device. These objects can be accessed through the System Buffer and Record Administration Server (SyBRAS). It provides efficient access to physical objects (synchronized by latches) and physical data independency, guarantees persistency for objects, etc.

The Persistent Object Store (POS) sits on top of SyBRAS. It provides another level of abstraction by supporting storage objects. Storage objects are constructed from physical objects, e.g., records. POS maintains direct physical references between storage objects, and offers associative, object-related and navigational access to these objects. Access refers to the linkage between storage objects in order to reconstruct objects — in this case logical local objects — of a more complex structure. Associative access means the well-known access via key values. Object-related access refers to direct object access using object identifiers. Navigational access is related to the propagation along physical references. Please see (Zezula and Rabitti, 1993; Kuckelberg, 1998) for more details.

Communicating agents implement the functionality of the whole system. They integrate the processing of queries, methods assigned to objects and transactions. Furthermore, they are responsible for distributing requests to (remote) agents that store corresponding objects or process requests more efficiently.

In the presence of replication, they have to ensure data consistency by applying a certain replication schemata. These communicating agents are realized as two-stack abstract machines (2SAMs). Each agent consists of two levels. Lower levels link with POS and employ 2SAMs that deal with logical local objects. Higher level machines act as coordinators for transactions and, thus, deal with logical global objects that are constructed from logical 'local' objects¹. On both levels multiple 2SAMs may process concurrently. In order to take advantage of concurrent processing and the distributed architecture 2SAMs can distribute requests to 2SAMs employed on the same level. However, only those machines on the higher level are equipped with an extended RPC enabling them to distribute requests to remote agents.

Having these multiple object levels — the storage, the logical local and the logical global object level — we can take advantage of a more sophisticated transaction management system. It is based on the multi-level transaction model (Beeri et al., 1989) which is counted as the most promising transaction model in theory.

The transaction management system controls the execution of operations of communicating agents and POS. Hence, it ensures local and global serializability. It consists of two components, a transaction manager and a recovery manager. The transaction manager takes advantage of 1) benefits coming with the detection of pseudoconflict, and 2) the fact that serializability of a multi-level schedule can be achieved by serializing concurrent operations/(sub-)transactions level-by-level (Weikum, 1991). Hence different level-specific concurrency control protocols can be employed. We currently support the strict two-phase locking protocol (str-2PL) and FoPL — a hybrid protocol — with certain optimizations (Schewe et al., 2000). They can be used in all possible combinations depending on the expected conflict probability on the corresponding level. The corresponding multi-level recovery manager guarantees atomicity, durability and data consistency. Therefore, it has to maintain a log reflecting all updates to objects on all levels, support complete and partial undos of (sub-)transactions, redo of (sub-)transactions, crash recovery, etc. This functionality is provided by the ARIES/ML recovery mechanism (Schewe et al., 2000; Kirchberg, 2002) which is an extension of the well-known, sophisticated ARIES recovery algorithm (Mohan et al., 1992) to multi-level systems.

On the logical level, data is described in terms of a data model. Our system is based on the generic

¹'local' does not refer to the local machine in this case. It refers either to logical local objects stored on the local machine or to logical local objects provided by a remote agent

object-oriented data model presented in (Schewe and Thalheim, 1993). It considers objects as abstractions of real world objects. The OODM distinguishes between values and objects. Every object consists of a unique, immutable identifier, a set of (type-, value-)pairs, a set of (reference-, object-)pairs and a set of methods. The OODM is based on an underlying type system and supports any arbitrary one. Types are used to structure values. Classes serve as structuring primitive for objects having the same structure and behaviour. A schema is given by a collection of classes. The operations provided by the underlying type system plus a single join operator allow to define a corresponding generic query algebra (Schewe, 2001).

In order to support distribution certain fragmentation techniques are employed. These are splitting, horizontal fragmentation and vertical fragmentation (Schewe, 2002). For this purpose, classes are considered. Each class is assigned to exactly one node — or in case of replication to several nodes — of a network. Hence, fragmentation decomposes the global objects corresponding to the original schema into several local objects corresponding to the fragmented schema. Having fragmentation and a class \leftrightarrow node relationship, we still must allocate the fragments — including fragmented methods — to the corresponding nodes.

Local objects, resulting from the fragmentation process, do not correspond directly to logical global objects as processed by the communicating agents. Furthermore, high-level queries, transactions, object methods, etc need to be translated into code that can be interpreted by communicating agents. This conceptual gap between the logical OODM-level and the communicating agents is bridged by a persistent reflective intermediate language (PRIL). PRIL will support linguistic reflection (Stemple and Sheard, 1991; Stemple et al., 1992). We provide a macro language, in which the high-level constructs in transactions such as generic update operations and the high-level algebra constructs for querying can be formulated. In (Stemple et al., 1990) it has been shown how linguistic reflection can be used to expand such macros for the case of query algebra constructs. In (Schewe et al., 1994) linguistic reflection has been applied to expand macros for generic update operations.

The internal representation of objects, replication, data distribution, etc is hidden from the user. This is realized by the user interface. It provides dialogue interfaces based on dialogue objects, which are defined by extended views (Schewe and Schewe, 2000). These dialogue objects can be created anywhere in the network. Initiating an operation associated with such a dialogue object would create a master agent at the user's node, and start executing a top-level transaction.

3 The Datamodel and its Interfaces

In this section we briefly describe the OODM from (Schewe and Thalheim, 1993), a generic approach to query algebras (Schewe, 2001), an extension to integrate dialogue interfaces (Schewe and Schewe, 2000), an approach to fragment OODM schemata (Schewe, 2002), and the idea of linguistic reflection and its use in expanding macros for queries and transactions (Stemple et al., 1990; Schewe et al., 1994).

The Object Oriented Datamodel. In the OODM a *database schema* is a finite set of classes. Based on the fundamental distinction between general abstractions called values and application-dependant abstractions called objects, the OODM distinguishes classes from types (Beeri, 1990). *Types* can be roughly seen as denoting sets of values.² Thus, we provide an underlying type system, e.g., $t = b \mid x \mid (a_1 : t_1, \dots, a_n : t_n) \mid \{t\} \mid [t] \mid \langle t \rangle$ (using abstract syntax). Here, b represents any collection of base types including one type *ID* to be used for object identifiers, and at least one further type. x represents type variables. (\cdot) , $\{\cdot\}$, $[\cdot]$ and $\langle \cdot \rangle$ provide constructors for tuple, set, list and multiset types.

Given any type system, the structural part of a *class* C is defined by a *structure expression* exp_C , which results from a type without occurrence of *ID* by replacing all type variables x_i by references $r_i : C_i$ with reference names r_i and class names C_i , and by the set of *superclasses*. Obviously, the class names appearing in references and superclasses must be defined in a schema. The behavioural part of a class is defined by operations that are associated with the class. Such operations are defined using the usual control constructs of imperative languages.

In order to define *databases* over a given schema, we also need the *representation type* T_C for a class C , which is simply obtained from exp_C by replacing the references by the type *ID*. Then in a database \mathcal{D} each class C of the schema is represented by a value $\mathcal{D}(C)$ of type $\{(id : ID, val : T_C)\}$, i.e., by a finite set of identifier-value pairs. Obviously, we have to require some constraints to be satisfied: uniqueness of identifiers, inclusion integrity with respect to superclasses, and referential integrity with respect to references.

However, there is another important requirement on databases called *value-representability*, a necessary and sufficient condition for the existence of generic update operations and the unique identifiability of objects. To explain this property assume that for

²This is not completely correct according to the existence of type polymorphism as discussed in (Cardelli and Wegner, 1985; Mitchell, 1990). However, for our purposes here this view suffices.

each $(i, v) \in \mathcal{D}(C)$ we expand the value v into a rational tree, i.e., whenever an identifier i' occurs within v and this identifier corresponds to the reference $r : C'$, then we replace i' in v by the unique value v' such that $(i', v') \in \mathcal{D}(C')$. This of course results in an infinite, yet finitely representable tree. Value representability requires these rational tree values to be unique within each class. For formal details we refer the reader to (Schewe and Thalheim, 1993).

Query Algebra. As in principle the OODM can be based on any underlying type system, the definition of query languages including query algebras requires also some sophistication. In (Schewe, 2001) it has been shown that a query algebra can always be defined in a generic way consisting of operations induced from the underlying type systems and a single generalized join operator.

Dialogue Interfaces. In general, user-issued transactions will involve different operations on various classes. This remains true for distributed schemata. The work in (Schewe and Schewe, 2000) provides a mechanism to integrate the interface with the database. This is done by defining *dialogue classes*.

Fragmentation. In order to support distribution we have to fragment the underlying OODM schema. The work in (Schewe, 2002) generalizes horizontal and vertical fragmentation from the relational data model to the OODM. In addition, a third kind of fragmentation called *split fragmentation* is introduced.

Linguistic Reflection. User-issued transactions and queries would involve the complex operations of the query algebra, e.g., the generalized join, and generic update operations (insert, update, delete). Each of these operations involves to analyse the schema and to compute the required types. It is known that both joins and generic updates are not parametric operations (Stemple et al., 1990; Schewe et al., 1994). In order to realize such complex operations we need a technique called *linguistic reflection*.

Roughly speaking, we will consider these operations as macros in our language for operations. Then the purpose of linguistic reflection is to expand these macros and to replace them by ordinary operations.

The basic idea of linguistic reflection is to use *reflection types* such as $SCHEMA_{rep}$, $CLASS_{rep}$, $TYPE_{rep}$, $METHOD_{rep}$, etc for the representation of abstract syntax expressions representing schemata, classes, types, methods, commands (method bodies), etc respectively. For each of these, there exists a function *raise* associating with this syntactic expression a true schema, class, type, etc respectively.

So, in particular the macros for the complex query and update operations would first turn the classes, types, etc for which they are to be defined into values of the corresponding reflection types. This is the effect of applying the operation *drop*. Then the computation would be performed on these values of the representation types. Finally, the result would be raised resulting into actual operations. For technical details of reflection we refer the reader to (Stemple et al., 1990) for complex query operations and for updates to (Schewe et al., 1994).

4 The Operational Architecture

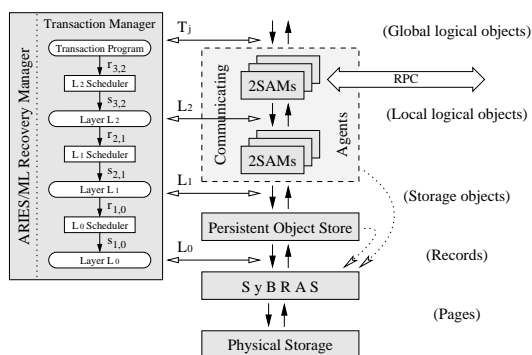


Figure 2: Operational & Physical Architecture.

In this section we focus on the operational architecture. A more detailed overview is shown in Figure 2. We now briefly describe an extension to two-stack abstract machines (Subieta et al., 1993), communication mechanisms for those machines, and an approach to multi-level concurrency control (Weikum, 1991; Schewe et al., 2000; Kirchberg and Schewe, 2001) and recovery (Schewe et al., 2000; Kirchberg, 2002).

4.1 Communicating Agents

The relationship between query languages and general-purpose programming languages has been studied since decades. The popular classification distinguishes between the embedded and integrated approach. The standard solution, namely embedding a query language in the programming language, suffers from problems collectively known as impedance mismatch. Alternative, integrated approaches such as Pascal/R, Napier88, O₂C, LOQIS (Subieta, 1991), etc circumvent these problems. Also, commercial products such as Oracle PL/SQL integrate query languages, in particular SQL, with imperative statements.

(Subieta et al., 1993) investigates the so-called 'seamless' integration of a query language (QL) with a programming language. Thus, a foundation of a QL-centralized programming language according to the traditional paradigms of the programming languages domain is built. An extended approach to two-stack abstract machine is proposed. In order to support query languages, the two stacks — the environment stack *ES* and the query result stack *QRES* — are modified and the semantics of query language operators are defined through operations on these stacks. The environment stack *ES*, as usual, determines scoping and binding. The query result stack *QRES* is a storage for intermediate query results, used either for the evaluation of query operators and for the evaluation of arithmetic-style expressions. However, this approach has some drawbacks:

1. It is not usable for distributed systems.
2. It does not support parallelism.
3. It is not suitable for large databases, as the stacks are main memory based.
4. It is not coupled with suitable concurrency control and recovery techniques.

We will briefly present an approach based on (Subieta et al., 1993) that removes these drawbacks. Instead of using a single two-stack machine we are using a network of machines for realizing a program, i.e., a transaction or a query, with the required number of machines being generated at run-time at sites distributed over a network. The communication between remote machines is based on an extended remote procedure call as described in Section 4.2.

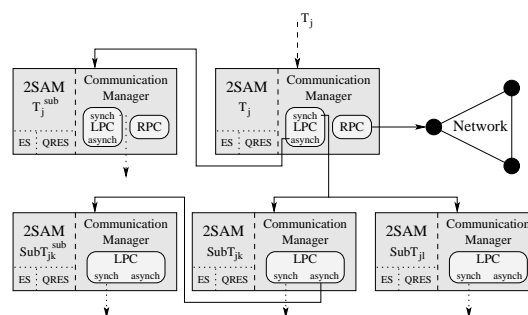


Figure 3: Architecture for Communicating Agents

The second drawback requires a more sophisticated, efficient architecture for local 2SAMs which makes it possible to distribute tasks among parallel 2SAMs. Additionally, the 2SAM language has to be extended with parallelization and synchronization commands, and new communication channels need to be created. Figure 3 shows a corresponding two-level architecture for communicating agents. Every

agent consists of a 2SAM and a communication module. Agents employed on the lower level link with POS and employ 2SAMs that deal with logical local objects. Agents on the higher level deal with logical global objects that are constructed from logical ‘local’ objects. ‘Local’ does not necessarily refer to the local machine since the higher level is capable of retrieving objects from remote agents.

On both levels multiple agents may process concurrently. An agent acts either as a manager for a higher-level request, e.g., a top-level transaction, or as a sub-manager, i.e., as a client, supporting a local or remote (sub-)manager employed on the same level. Every agent has the capability of creating sub-managers as required. It has to maintain a list of its sub-managers together with information enabling the agent to send further requests to a particular sub-manager, retrieve results, etc. The communication module provides the functionality to interact with other agents and layers, keeps track of sub-managers, and maintains a list of requests and results which have been received. Local agents can interact in two modes, asynchronous or synchronous. Asynchronous calls enable agents to process concurrently. The synchronous mode corresponds to sequential processing.

If we consider Figure 3, the 2SAM T_j acts as manager for the higher-level request T_j , in this case a top-level transaction. The request consists of a sequence of operations on logical global objects. The 2SAM T_j examines this sequence first. Afterwards, some operations are distributed to remote agents if necessary. Assume these remote operations can be executed independently from the rest. Thus, an asynchronous RPC call is issued. The remaining operations will be executed by local agents. Therefore, the 2SAM T_j tries to optimize their processing tasks by forwarding certain (sub-)tasks to local sub-managers. Assume that only one sub-manager 2SAM T_j^{sub} is created and both 2SAMs T_j and T_j^{sub} can process concurrently. Thus, an asynchronous call is issued and a sequence of (sub-)tasks is forwarded to 2SAM T_j^{sub} . Both agents then execute their tasks. Within this process several requests to the next lower level are issued. These requests are mainly synchronous ones. As soon as the sub-manager 2SAM T_j^{sub} has completed a (sub-)task, the corresponding partial result is returned to its manager. The manager is responsible for composing the final result for every set of partial results received from its sub-manager(s).

Since stacks are main memory based and the amount of data they process can get quite large, it might be necessary to export parts of the stacks to persistent storage. A stack can be accessed from the top only. Hence, we will keep the upper part of a stack in main memory and export only its lower part. In such a case, a 2SAM takes advantage of the services

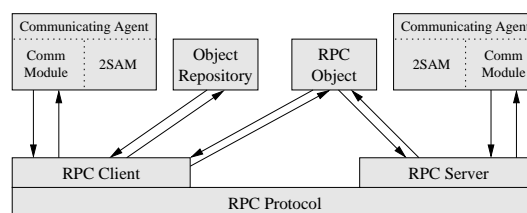


Figure 4: Network Communication Architecture

provided by the buffer manager³. It allows data to be exported to and efficiently accessed from a persistent storage through its page interface.

The last drawback to be removed addresses the coupling with suitable concurrency control and recovery techniques. Section 4.3 proposes a corresponding multi-level transaction management system.

4.2 Network Communication

Efficient communication capabilities are essential for distributed databases. We will briefly present a corresponding extended middleware component based on remote procedure call (RPC). Main differences to conventional RPCs are support of bulk type values, references and call-by-reference. Figure 4 shows the basic architecture for our network middleware. The major components are as follows:

- **RPC Client:** Provides interfaces to communicating agents to issue RPC calls.
- **RPC Server:** Matches the RPC call from the client to the corresponding communicating agent.
- **Object Repository:** Stores information about classes, in particular a class \leftrightarrow node mapping. It is frequently accessed by the RPC client to retrieve details on how to access remote objects. A copy of this repository is available on all nodes and kept in main memory for efficiency reasons.
- **RPC Protocol:** Provides a reliable and efficient way to communicate between RPC client and server.
- **RPC Object:** Is the external representation of a logical object that contains flattening and unflattening functions.

The RPC middleware provides access to remote objects and its methods. We support synchronous and asynchronous requests. One can think of a request as a block of sub-requests. All sub-requests of a block will be forwarded to the same remote agent. Such block requests allow sub-requests to access data, results, etc of previously executed sub-requests of the same block. Hence, the amount of data to be exchanged between nodes is decreased.

³The buffer manager is part of SyBRAS

4.3 Concurrency Control & Recovery

From an internal point of view, users access databases in terms of transactions. Fast response times and a high transaction throughput are crucial issues for all database systems. Hence, transactions are executed concurrently. The transaction management system ensures a proper execution of concurrent transactions. It implements concurrency control and recovery mechanisms to preserve the so-called ACID principles. A further increase in both response time and transaction throughput can be achieved by employing a more sophisticated transaction management system, e.g., a system based on the multi-level transaction model (Beeri et al., 1989; Weikum, 1991; Schewe et al., 2000; Kirchberg and Schewe, 2001). This multi-level model is counted as the most promising transaction model. It schedules operations of transactions based on information obtained from multiple levels. Since there are usually less conflicts on higher levels, lower level conflicts can be ignored. Hence, their detection increases the rate of concurrency.

The execution of concurrent transactions is described by a multi-level schedule. Corresponding one-level schedules, so-called level-by-level schedules, can be defined for each level. A multi-level schedule is multi-level serializable if it is equivalent to a serial multi-level schedule. According to (Beeri et al., 1989) a multi-level schedule is multi-level serializable, if all level-by-level schedules are conflict serializable. Thus, level-specific concurrency control protocols can be employed.

The layered system architecture as proposed in Section 2 allows to use the multi-level transaction model in a straightforward manner. Figure 2 shows the corresponding operational and physical architecture in more detail. It shows that the transaction management system controls the execution of operations of the communicating agents and POS. Since communicating agents consist of two levels a three-level scheduler is employed. This scheduler consists of three level-specific schedulers. Each of them employs a certain concurrency control protocol. On the highest level L_2 sequences of 2SAM operations on logical global objects are serialized. Such a sequence always corresponds to a top-level transaction. Since 2SAM operations on logical global objects are implemented by sequences of 2SAM operations on logical local objects, the L_2 scheduler creates an execution order of these operations on logical local objects. The L_1 scheduler then tries to optimize the output of the L_2 scheduler by serializing the corresponding operations on the lower level of the communicating agents. And so on. Finally, SyBRAS executes record operations that are serialized by the L_0 scheduler.

The general approach to concurrency control is the use of locking protocols, especially strict two-phase locking (str-2PL). However, locking suffers from some major problems affecting the transaction throughput. Two of those problems are transaction deadlocks and the impossibility to accept all (conflict-)serializable schedules. (Schewe et al., 2000) presents a hybrid protocol called FoPL (*forward oriented concurrency control protocol with preordered locking*), which is a provably correct protocol for multi-level transactions. FoPL does not suffer from any of those two mentioned problems. (Kirchberg and Schewe, 2001) describes first experimental results of the basic FoPL protocol in comparison to str-2PL. It is outlined that FoPL-based scheduler are most likely to outperform locking based protocols if some of the proposed optimizations to FoPL (Schewe et al., 2000) are realized. However, one has to consider that FoPL follows an optimistic idea. Hence, it is designed to be used on levels with a low conflict probability only.

Since we deal with distributed transactions guaranteeing serializability as described above is not sufficient. Global serializability and one-copy serializability (Bernstein and Goodman, 1985) have to be guaranteed when dealing with distributed transactions on replicated objects. (Kirchberg, 2002) outlines how str-2PL and FoPL can be used on the global level — which is level L_2 . Guaranteeing global serializability requires the use of a commit protocol, e.g., two-phase commit, to ensure atomicity. Additionally, global deadlocks detection mechanisms must be employed if str-2PL is used. FoPL requires only globally unique timestamps to be assigned at the start of the validation phase. Guaranteeing one-copy serializability requires a certain replication schema to be applied (Özsu and Valduriez, 1999). As far as FoPL is concerned only a few extension to this replication schema have to be done. (Kirchberg, 2002) also compares str-2PL and FoPL w.r.t. their efficiency in the presence of a distributed data. It is outlined that the necessary extensions to FoPL are much less expensive than the extensions to str-2PL.

Since concurrency control may force transactions to be aborted, recovery mechanisms are provided. So far only a few research projects have focussed on recovery mechanisms for multi-level transactions (Rothermel and Mohan, 1989; Lomet, 1992; Weikum, 1991). However, none of these approaches comes without major restrictions or disadvantages limiting its practical use. (Schewe et al., 2000; Kirchberg, 2002) outline an alternative approach, ARIES/ML. It overcomes those restrictions and disadvantages, preserves their major advantages and even supports all kinds of concurrency control protocols, both inverse and non-inverse operations, partial rollbacks etc.

5 Conclusion

This article proposes a multi-level architecture for distributed object bases. The proposed systems meets two steadily increasing demands. On the one hand it supports a distributed architecture that is flexible w.r.t. network architectures and communication mechanisms. On the other hand the demand for representing complex, real-world objects, relationships among those objects and even behaviour associated with them as real as possible is met by describing data on the basis of an object-oriented data model.

Key features of the architecture are the use of abstract communicating agents and an extended remote procedure call to integrate the processing of queries, methods assigned to objects and transactions, distribute requests to remote agents, exploit parallelism, and act as replication and transaction managers; the use of multi-level transactions to increased transaction throughput; and the use of linguistic reflection to map database schemata to the level of the agents.

REFERENCES

- Beeri, C. (1990). A formal approach to object-oriented databases. *Data & Knowledge Engineering*, 5(4):353–382.
- Beeri, C., Bernstein, P. A., and Goodman, N. (1989). A model for concurrency in nested transactions systems. *Journal of the ACM (JACM)*, 36(2):230–269.
- Bernstein, P. A. and Goodman, N. (1985). Serializability theory for replicated databases. *Journal of Computer and System Sciences*, 31(3):355–374.
- Cardelli, L. and Wegner, P. (1985). On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys (CSUR)*, 17(4):471–523.
- Kirchberg, M. (2002). Exploiting multi-level transactions in distributed database systems. In *Proc of the Workshop on Distributed Data & Structures*. Carleton Scientific.
- Kirchberg, M. and Schewe, K.-D. (2001). A comparison of multi-level concurrency control protocols. In *Proc of the 12th Australasian conf on Database technologies*, pages 153–160. IEEE Computer Society Press.
- Kuckelberg, A. (1998). The matrix-index coding approach to efficient navigation in persistent object stores. In *Workshop on Foundations of Models and Languages for Data and Objects*, pages 99–112.
- Lomet, D. B. (1992). MLR: a recovery method for multi-level systems. In *Proc of the ACM SIGMOD Conf on Management of Data*, pages 185–194. ACM Press.
- Mitchell, J. C. (1990). Type systems for programming languages. pages 365–458.
- Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., and Schwarz, P. (1992). ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS)*, 17(1):94–162.
- Özsu, M. T. and Valduriez, P. (1999). *Principles of distributed database systems (2nd ed.)*. Prentice-Hall.
- Rothermel, K. and Mohan, C. (1989). ARIES/NT: a recovery method based on write-ahead logging for nested transactions. In *Proc of the 15th Conf on Very Large Data Bases*, pages 337–346. Morgan Kaufmann Publishers Inc.
- Schewe, K.-D. (2001). On the unification of query algebras and their extension to rational tree structures. In *Proc of the 12th Australasian conf on Database technologies*, pages 52–59. IEEE Computer Society Press.
- Schewe, K.-D. (2002). Fragmentation of object oriented and semi-structured data. In *Proc of the Baltic Database and Information Systems Conf*.
- Schewe, K.-D., Ripke, T., and Drechsler, S. (2000). Hybrid concurrency control and recovery for multi-level transactions. *Acta Cybernetica*, 14:419–453.
- Schewe, K.-D. and Schewe, B. (2000). Integrating database and dialogue design. *Knowledge and Information Systems*, 2(1):1–32.
- Schewe, K.-D., Stemple, D. W., and Thalheim, B. (1994). Higher-level genericity in object-oriented databases. In *Conference on Management of Data*.
- Schewe, K.-D. and Thalheim, B. (1993). Fundamental concepts of object oriented databases. *Acta Cybernetica*, 11(1-2):49–84.
- Stemple, D., Fegaras, L., Sheard, T., and Socorro, A. (1990). Exceeding the limits of polymorphism in database programming languages. In *Proc of the Conf on Extending Database Technology on Advances in Database Technology*, pages 269–285. Springer-Verlag New York, Inc.
- Stemple, D. and Sheard, T. (1991). A recursive base for database programming primitives. *Lecture Notes in Computer Science*, 504:311–332.
- Stemple, D., Sheard, T., and Fegaras, L. (1992). Reflection: A bridge from programming to database languages. In *Proc of the Hawaii Conf on System Sciences*.
- Subieta, K. (1991). LOQIS: The object-oriented database programming system. *Lecture Notes in Computer Science*, 504:403–421.
- Subieta, K., Beeri, C., Matthes, F., and Schmidt, J. (1993). A stack-based approach to query languages. Technical Report 738, Institute of Computer Science Polish Academy of Sciences, Warszawa, Poland.
- Weikum, G. (1991). Principles and realization strategies of multilevel transaction management. *ACM Transactions on Database Systems (TODS)*, 16(1):132–180.
- Zeuzula, P. and Rabitti, F. (1993). Object store with navigation acceleration. information systems. *Information Systems*, 18(7):429–459.