

A Java Application Programming Interface to a Multimedia Enhanced Object-Oriented DBMS

Susanne Boll, Jürgen Wäsch

German National Research Center for Information Technology (GMD),
Institute for Integrated Information and Publication Systems (GMD-IPSI),
{boll, waesch}@darmstadt.gmd.de
<http://www.darmstadt.gmd.de/~{boll, waesch}>

Abstract

Advanced multimedia applications call for integrated DBMS support, i.e., the integrated modeling, management, and interactive presentation of persistent multimedia documents. Generic services for the presentation of multimedia documents have to be available at the database clients. This includes continuous data delivery, realization of presentation layout, synchronization enforcement, and an efficient interaction handling. In this paper, we propose a Java application programming interface (Java API) to a multimedia enhanced OODBMS. We follow a two-step approach: First, we implement access to persistent data managed by the core OODBMS. Second, we present the design of generic multimedia presentation services in the Java. This Java API enables database client applications to access and to present multimedia documents persistently stored in the database. With our Java API platform-independent access to multimedia information stored in database systems is possible.

1 Introduction

Many Java¹ [GJS96] applications and applets need access to data that are persistently stored and managed by a database management system (DBMS). On the other hand, the object-oriented programming language Java seems to be a promising candidate to provide both platform-independent and easy-to-use access to DBMSs. Using Java, interactive graphical database front-ends can easily be built and downloaded from a WWW-server.

It seems obvious to us that in future DBMSs will provide the technologies to support multimedia applications. This includes the management of media types such as image, audio, and video as well as of multimedia documents that capture the spatial and temporal composition of different media objects

and specify interaction possibilities. It is accepted that object-oriented DBMSs (OODBMS) form a suitable basis for modeling, storage, and management of different media types especially continuous media types as well as of multimedia documents [AK92, BWAH96, CK95, RNL95, WA96]. As multimedia documents are subject to presentation, the database front-end has to implement services like presentation of composite multimedia documents, interaction handling, and continuous data delivery. If these services are available in Java world-wide platform-independent access to multimedia information stored in a DBMS is possible.

In this paper, we propose a Java application programming interface (Java API) to a multimedia enhanced OODBMS. The Java API enables an application programmer to access and to present media data and multimedia documents stored in the database. To prove our concepts, we use the client/server-based OODBMS VODAK [GMD95, KAN94] as a sample system. VODAK is currently being extended with multimedia capabilities mentioned above. The proposed Java API that can be used in both stand-alone Java applications and Java applets basically consists of two parts:

First, a database interface to traditional data managed by the core OODBMS: The Java API provides an application programmer with functionality equivalent to the data manipulation language of the OODBMS to enable application programming in Java. In case of VODAK, this means that the primitive and complex abstract data types of VODAK have to be implemented in the Java API. The Java API gives access to objects stored in a database by calling methods that are defined in the database schema. With the Java API it is possible to execute declarative queries written in the OODBMS's query language and to process their results. For programming dynamic applications it is possible to have access to meta data, i.e., information about the database schema.

¹Java is a trademark of Sun Microsystems Inc.

Second, multimedia presentation facilities: This part implements the efficient access to continuous data stored in the database. At the same time the Java API offers functionality to present discrete (time-independent) and continuous (time-dependent) media objects at the client. The Java API also gives an application access to the multimedia documents stored at the OODBMS server. For the presentation of multimedia documents, the Java API implements functionality for the synchronized playback of interactive multimedia documents exploiting the features of Java to present interactive multimedia documents, e.g., multi-threading and the abstract window toolkit.

In section 2, we briefly introduce VODAK as we use it as an example OODBMS. After that, we present the overall design of the Java API. In section 4, we describe the Java database interface to the core OODBMS. Section 5 introduces the multimedia extensions to VODAK. We outline multimedia presentation facilities that can be used by Java applications or applets to access and to present multimedia data and multimedia documents managed by a multimedia enhanced OODBMS.

2 Overview of the OODBMS VODAK

As we use VODAK as an example, its relevant concepts are introduced briefly in this section. VODAK [GMD95] is a client/server based OODBMS. It includes advanced features like an extensible data modeling and programming language [KAN94], an object-oriented query language [AF95], open nested transaction management [MRW⁺93], and distribution facilities [KFM⁺96].

2.1 VODAK data model

Primitive and complex data types. Similar to Java, VODAK distinguishes between *objects* and *data types*. The primitive data types supported by VODAK are `BOOL`, `INT`, `REAL`, `STRING`, `BYTESTRING`, and `OID`. Complex data types supported are `SET`, `ARRAY`, `DICTIONARY`, `TUPLE`, and `LIST`. Nesting of complex data types is possible, e.g., `SET` of `TUPLE`. All data types offer a rich set of built-in functions and operators [GMD95].

Objects with structure and behaviour. Objects are identified through unique object identifiers whereas values of data types have no identifier. The properties and methods of an object are described by the object's class definition in the database schema. Properties of objects may be of

primitive or complex data type, or may contain object references (OID values). Methods can take only values of data types as arguments and return values of data types. This implies that objects stay in the database and only references to objects in the database, i.e., OID values, are returned to an application program.

Classes. Classes define the structure and behaviour of objects and serve as containers for objects. An object is instance of only one class. In VODAK, classes are first-class objects, i.e., they can be treated like ordinary objects. A database schema is a collection of classes.

2.2 Languages for data definition, manipulation, and retrieval

VODAK provides a procedural modeling and programming language and a declarative query language for convenient and efficient database access [GMD95]. The VODAK Modeling Language (VML) consists of a DDL part which is used to define database schemas, and a procedural, computationally complete DML part for the implementation of database schemas and of application programs. With the VODAK Query Language (VQL) declarative SQL-like queries can be written. These queries can be executed within application programs. VQL currently supports a subset of the features that are defined in the OQL standard [Cat94]. In particular, this concerns the usage of method calls, of complex data types, and of path expressions in VQL queries.

2.3 VODAK transaction models

VODAK servers can run in different transaction modes [GMD95, MRW⁺93]. The transactional commands offered by VML are the same for all transaction modes. An application programmer can define her own atomic units by starting, committing, and aborting a top-level transaction.

2.4 Multimedia extensions to VODAK

A generic VML data type `CODUSTRING` is currently being integrated into VODAK. This data type supports the random access to continuous data stored at the database server. It serves as the basis for the implementation of continuous media types like audio and video at the OODBMS server.

We have developed a database schema that allows for the modeling and storage of multimedia documents by means of extended Object Composition Petri Nets [BKL96]. This database schema includes methods that map a multimedia document

into a presentation plan, i.e., an internal format that encodes all information relevant to the presentation of the document.

When a client application requests a multimedia document from the server, the respective presentation plan is generated at the server and sent to the client. The client software is then responsible for the interactive presentation of the document. This includes the interpretation of the presentation plan, the timely access to media data stored at the server, the presentation and synchronization of media data, as well as the handling of user interactions.

3 Overall design of the Java API

This section presents our design of a Java API to a multimedia enhanced OODBMS. Figure 1 illustrates the architecture of the Java API to the sample OODBMS VODAK. The Java API consists of four layers that are described below.

1. *Communication protocol.* The communication protocol between client and server implements the transfer of OODBMS commands from a client to a database server and the transfer of discrete and continuous data from and to the database server.
2. *Connection handling and database interface.* This layer consists of Java classes that realize the connection handling and provide a well-defined interface from a Java client to database servers.
3. *OODBMS data types.* This layer implements the data types of the OODBMS as a hierarchy of Java classes. This covers the primitive and complex data types for discrete data as well as a generic continuous data type. The Java continuous data type enables continuous data transport from a server to a client.
4. *Multimedia presentation facilities.* The Java classes constituting this layer enable the interaction with and the playback of both media objects and multimedia documents stored at the database server. This layer employs the continuous data type for the implementation of continuous media types like audio and video in Java.

Layers 1-3 are described in the next section. There, we focus on accessing discrete data managed by the OODBMS. The continuous data type is presented

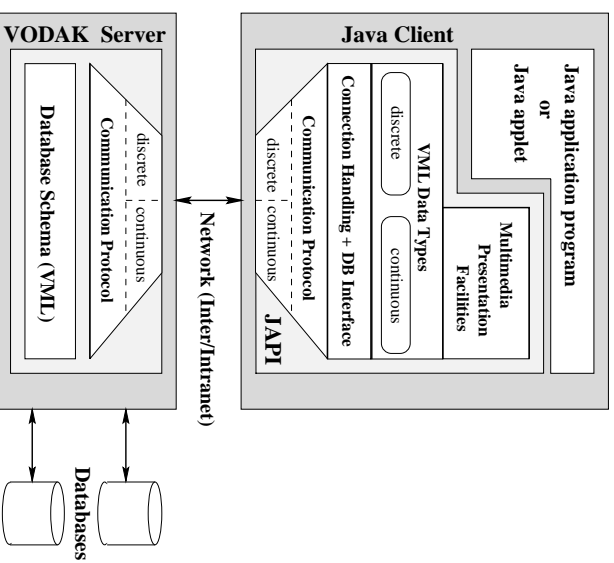


Figure 1: Overall architecture of the Java API to the OODBMS VODAK

together with the multimedia presentation facilities of the Java API in section 5.

With this layered approach the Java API tries to abstract from the details of the underlying OODBMS and its object-oriented data model. Layer 1 aims at hiding the details about the physical connection and the implementation of the data transfer between client and server. Layer 2 enables the submission of commands, queries and method calls to the database server within a Java program. Layer 3 provides the built-in OODBMS data types to make the full OODBMS functionality available in Java. Layer 2 together with layer 3 realize the automatic mapping of values of database data types to the corresponding Java objects and vice versa.

Of course, our prototypical implementation of the Java API is tailored to VODAK. The data model and the built-in data types of other OODBMSs differ to some degree from those of VODAK. Hence, it may be necessary to adapt some layers of our Java API prototype if we use another OODBMS, e.g., by adding new database data types to the Java API. However, the design of the Java API is general enough to cover the model differences of several OODBMSs. Layers 1-3 realize a “call-level” interface to an OODBMS (like JDBC [HR96] does for relational DBMSs), i.e., these layers only support the execution of methods and queries at the database server and the retrieval of their re-

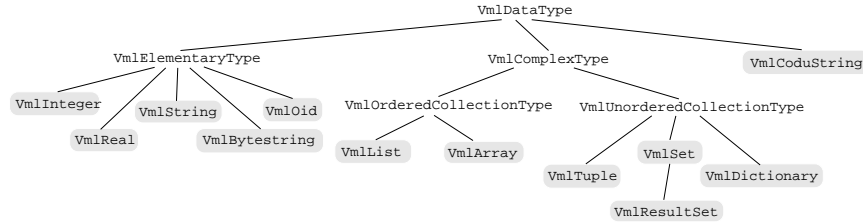


Figure 2: The Java class hierarchy implementing VML data types

sults. Method execution on database objects with this approach is possible as long as the OODBMS is able to return handles to database objects to the client (like VODAK’s OID data type). As we do not aim at transparent persistency of Java objects but at the implementation of multimedia presentation facilities, this “call-level” interface is sufficient.

With respect to the implementation of the continuous data type and the multimedia presentation facilities we do rely more closely on the specialities of the underlying OODBMS. The details of storage and transport of continuous data and the modeling of media types and multimedia documents in the specific OODBMS might influence the Java API design. Unfortunately, there is no commonly accepted approach for the modeling of media data and multimedia documents in OODBMSs.

4 Java API to discrete data managed by VODAK

4.1 VML data types

To enable database application programming in Java, the Java API must provide functionality equivalent to the DML part of OODBMS. Thus, the primitive and complex data types of VML have to be implemented in Java. Figure 2 shows the Java class hierarchy that implements VML’s discrete data types `BOOL`, `INT`, `REAL`, `STRING`, `BYTESTRING`, `OID`, `SET`, `ARRAY`, `DICTIONARY`, `TUPLE`, and `LIST`. The class `VmlResultSet` is used to represent VQL query results at the client. A `VmlResultSet` object contains `VmlTuple` objects and implements additional methods to access, e.g., “rows” in a query result set. `VmlCoduString` implements VML’s continuous data type and is further described in section 5. The grey shaded classes in the hierarchy can be instantiated, all other classes are abstract ones.

We decided to implement the complex VML data types by encapsulating objects of predefined Java classes. E.g., the implementation of `VmlSet` uses the Java class `Vector` to store the elements of the

set. Each Java class implements only those methods that are compliant with VML. This prohibits that a programmer circumvents the data type’s semantics, e.g., by accessing the 5th element of a `VmlSet` object. Moreover, the methods of the complex data types implement dynamic type checking. This guarantees homogeneous complex data types, i.e., all elements have to be of the same data type, as this is required by the VODAK data model.

Figure 3 indicates the implementation of the Java class `VmlSet`. The methods of this class implement the built-in functions for the VML data type `SET`.

4.2 Connection handling and database interface

Connection handling. When an application programmer wants to access a VODAK server she first has to obtain a `VodakConnection` object by calling a `newConnection` method of the class `VodakEnvironment` (similar to JDBC [HR96]). One connection object is concerned with exactly one connection to a VODAK server. Via the connection an application programmer can open a database with the `openDatabase` method (cf. Figure 4). Before opening another database at this connection the one already open must be closed with `closeDatabase`. To finally close the connection to a VODAK server the connection object offers a `close` method. Of course, a single application can maintain multiple database connections to one or more databases, one per connection object.

Database Interface. A `VodakConnection` object offers several methods that implement the well-defined interface to the OODBMS. Connection objects hide all details of connection handling and communication protocols from an application programmer. Figure 4 gives a representative list of method signatures of the class `VodakConnection`.

After a database has been opened, `sendMethod` can be used to call methods on objects in the database and to receive the respective results. The argument of `sendMethod` is an array of Java `VmlDataType` objects that represent values of VML

```

public class VmlSet extends VmlUnorderedCollectionType {
    private Vector vector; // The Vector object contains the elements of the VML set.
    ...
    // Methods implementing VML set functionality:
    public synchronized boolean insert(VmlDatatype anElement) {...}
    public synchronized boolean remove(VmlDatatype anElement) {...}
    public boolean includes(VmlDatatype anElement) {...}
    public int size() {...}
    public synchronized VmlSet union(VmlSet aSet) {...}
    public synchronized VmlSet intersection(VmlSet aSet) {...}
    public synchronized VmlSet difference(VmlSet aSet) {...}
    public synchronized boolean equals(Object obj) {...}
    public synchronized VmlDatatype copy() {...}
    // Other methods:
    public synchronized Object unwrap() {...}
    ...}

```

Figure 3: Outline of the Java class VmlSet

```

public class VodakConnection{
    // Create, open, close, delete a VODAK database at a server:
    public synchronized void createDatabase(String schemaName, String dbName) throws VodakException {...}
    public synchronized void openDatabase(String dbName) throws VodakException {...}
    public synchronized void closeDatabase() throws VodakException {...}
    public synchronized void deleteDatabase(String dbName) throws VodakException {...}

    // Get the oid of a class in the opened database from a server:
    public synchronized VmlOid getClassOid(String classname) {...}

    // Send a method call to a VODAK database (server) and receive the results:
    public synchronized VmlDataType
        sendMethod(VmlOid receiver, String methodname, VmlDataType arguments[]) throws VodakException {...}
    ...

    // Send a VQL query to a VODAK database (server) and receive the results of the query:
    public synchronized VmlResultSet execQuery(String queryString) throws VodakException {...}
    ...

    // Begin, commit, abort a top-level transaction at a VODAK database (server):
    public synchronized void beginTransaction() throws VodakException {...}
    public synchronized void endTransaction() throws VodakException {...}
    public synchronized void abortTransaction() throws VodakException {...}
    ... }

```

Figure 4: Representative list of methods offered by the class VodakConnection

data types. After the method has been executed at the server, the connection object automatically creates the respective Java objects from the result data received. Note, that the server never returns database objects — only data values or references to objects (represented by `VmlOid` objects) are returned to a database client. The same holds for `getClassOid` which returns a Java `VmlOid` object that represents a reference to a database class. Another way to access an opened database is to execute a VQL query at a database server. The result of `execQuery` is encoded as a Java `VmlResultSet` object (cf. section 4.1). Starting, committing, or aborting a top-level transaction has the usual semantics. A database transaction covers all method calls and queries between `beginTransaction` and `endTransaction` of a connection object.

For programming dynamic applications, access to meta data, e.g., information about attributes and methods that belong to a class or instances of a class in a database, is enabled by calling methods defined on the data dictionary classes of VODAK.

The methods described above are sufficient to

write application programs that exploit the entire database functionality. With these methods, however, more complex methods can be built and added to the Java API to provide an database application programmer with a higher-level API for more sophisticated database access. To get an intuitive understanding how to use the core Java API we give a sample program in Figure 5.

Error/exception handling. For the Java API we have to consider only those errors and exceptions that are not already handled by the database server. We distinguish between errors that are detected by the client and exceptions raised by the server. E.g., if we try to execute a query on a closed database we detect this already at the client and raise a `VodakClientException`. Dynamic type checking of complex data types may raise a `VmlDataTypeException`, e.g., if a `VmlInteger` object is inserted into a `VmlSet` of `VmlReal`. Break down of a connection can also be handled by a client (`VodakConnectionException`). Examples for server exceptions are unilateral transac-

```

// Open a connection to the VODAK database server at host "rombach", port 5000.
VodakConnection conn = VodakEnvironment.newConnection("rombach", Integer(5000));
// Open the database "GMDIPSI"
conn.openDatabase("GMDIPSI");
try{
// Start a transaction at the opened database "GMDIPSI".
conn.beginTransaction();
// Retrieve all persons [oid, name, name of department] who work in both the VODAK
// project and the Java project by executing a VQL query at the opened database.
VmlResultSet persons = conn.executeQuery(
"ACCESS p, p.name, p.department.name FROM p IN Persons
WHERE {'VODAK', 'JAVA'} IS-IN p->projectNames()");
// Select the oids of these persons
VmlSet personOids = persons.getRow("p");
// Get the oid of the VODAK class "Projects"
VmlOid projects = conn.getClassOid("Projects");
// Create a new project in the database with name "JAPI".
VmlOid newProject = conn.sendMethod(projects, "createProject", "JAPI");
// If the project was successfully created add the selected persons to the project and
// add the new project to the property "projects" of the selected persons. This is done
// by the method "addProjectMembers" of the class "Projects" in the database schema.
if (newProject.isNotNull())
{conn.sendMethod(newProject, addProjectMembers, personOids);}
// Commit the transaction.
conn.endTransaction();}
// If something went wrong, abort the transaction.
catch (VodakException e) {conn.abortTransaction();}
// Close the database "GMDIPSI"
conn.closeDatabase();
// Close the connection to the VODAK database server
conn.close(); }

```

Figure 5: Sample Java database application program

tion aborts due to deadlocks or non-existing object identifiers referred to by a client method call. In these cases, the server returns appropriate status information which is interpreted by the connection object and transformed to the respective `VodakServerException` object.

Transactions and multi-threading. VODAK offers only a synchronous interface to discrete data stored in its databases. To be multi-thread safe, all methods of class `VodakConnection` are declared as `synchronized`. To achieve parallelism, one can maintain multiple database connections to one or more databases, e.g., within different Java threads. This, however, can cause several problems: Transactions executed concurrently at the same database are subject to concurrency control at the database server. If the threads executing transactions are not independent, this may cause distributed deadlocks if the wait-for-dependencies of synchronized threads conflict with the wait-for-dependencies of their database transactions. It is left to the application programmer to resolve this kind of deadlocks. However, we are mainly concerned with multimedia presentation tasks on the client and for this we need only read access to the database from a multi-threaded Java applet or application.

4.3 Communication protocol

In our first prototype we use a proprietary communication protocol for sending commands and data

from a client to a server and vice versa. The protocol for discrete data exchange is specified in [BLW96]. It is based on the exchange of ASCII characters via sockets and is SGML compliant.

Conceptually, we could replace a proprietary communication protocol by a standard one, e.g., CORBA [OMG95]. A mapping of IDL to Java and Java object request broker (ORB) implementations are already available. The usage of CORBA implies that we have to define and implement a CORBA interface on the OODBMS server site.

As VODAK does not offer a CORBA interface yet and CORBA does not support continuous data transport, we decided to build our Java API prototype using a proprietary communication protocol. Switching over to CORBA does not affect layers 2-4 of the Java API. Only layer 1 has to be replaced by a CORBA-based implementation. However, implementing efficient continuous data delivery using CORBA seems to be difficult.

5 Multimedia presentation facilities

In this section, we present the multimedia presentation facilities of the Java API. With this layer of the Java API we want to put an application programmer into the position to access discrete and continuous media objects persistently stored in the database. Moreover, we want to provide the appli-

cation programmer with facilities to present these media objects as well as the multimedia documents that are stored at the server.

Though we are mainly concerned with the multimedia presentation facilities of a Java client, we first have a look on the concepts behind the modeling and management of multimedia data and multimedia documents in a multimedia enhanced OODBMS server. The multimedia presentation facilities of the Java API are presented in section 5. There, we explain how media objects, especially continuous media objects can be accessed by a Java client. Thereafter, we show how the presentation of media objects and of multimedia documents at a client can be implemented exploiting the features of Java.

5.1 The multimedia database server

The multimedia database server must efficiently manage different kinds of media data. Therefore, continuous as well as discrete media types must be modeled in the database schema. An efficient transport of continuous media data between server and clients is necessary. Moreover, the multimedia database server must support the modeling of multimedia documents. These documents capture the temporal and spatial composition of different media objects and specify interaction possibilities.

The AMOS research group at GMD-IPSI is extending VODAK with the mentioned multimedia capabilities. The lower part of Figure 6 shows the architecture of the multimedia database server. The server controls the concurrent access from the clients to discrete and continuous media data as well as to multimedia documents. The VODAK database management system is used for the management of discrete data. The multimedia database server employs additional External Media Servers (XMS) to efficiently manage continuous media data on specialized storage systems. The Continuous Object Manager (COM) provides multi-user access to continuous data via the XMS [RNL95, MKK95]. We concentrate on the issues of VML's continuous datatype and of the object-oriented modeling of media types and multimedia documents in the database schema, as this effects on the design of the Java API.

Continuous data type. The multimedia database server offers the generic continuous data type `CODUSTRING`. This data type can be considered as an array of continuous data units (CODU). The CODUS contain raw media data. Additionally `CODUSTRING` contains a meta-information about the continuous data, e.g., the number of the CODUS. The COM provides for random access to CODUS in a `CODUSTRING`.

Modeling of media objects. Discrete as well as continuous media types are modeled by means of VML classes in a database schema. This allows for the easy integration of new media types. Each class utilizes appropriate VML data types to store the respective media data, e.g., `BYTESTRING` for an image class and `CODUSTRING` for an audio class. The methods of the media classes provide for the media specific access to and the manipulation of the media objects at the multimedia database server.

Modeling of multimedia documents. Additionally, we have developed a VML database schema that enables the modeling and storage of multimedia documents. To describe multimedia documents, we use *presentation nets* [BKL96] which add several features to Object Composition Petri Nets [LG90]. Our database schema includes the mapping of a multimedia document into a *presentation plan*, i.e., an internal format that encodes all information relevant to the presentation of the document. Basically, a presentation plan contains references to the media objects in the database, information about spatial and temporal layout of the presentation as well as possible interactions with the end user.

5.2 Building a multimedia database client with Java

In the following, we describe the implementation of the Java classes that represent the database media objects on the client. For each of these media classes the corresponding presentation functionality must be offered by respective media presenters. When a multimedia document is requested by a client, the corresponding presentation plan is generated from the document and sent to the client. The Java API software is then responsible for the synchronized presentation of the document. The components that constitute the multimedia presentation facilities of the Java API are shown in the upper part of Figure 6.

Modeling media types in Java. For each media type defined in the multimedia database schema a corresponding Java class is implemented. The structure of each media class in Java is similar to its corresponding server class. Whereas the server classes are modeled by means of the VML data types, the client Java classes utilize the discrete and continuous Java `VmlDataTypes`. The methods of the client Java classes mainly are employed to give access to the media data at the client. The manipulation of media objects, however, is left to the server implementation.

We extend the database interface by appropriate methods to instantiate objects of these me-

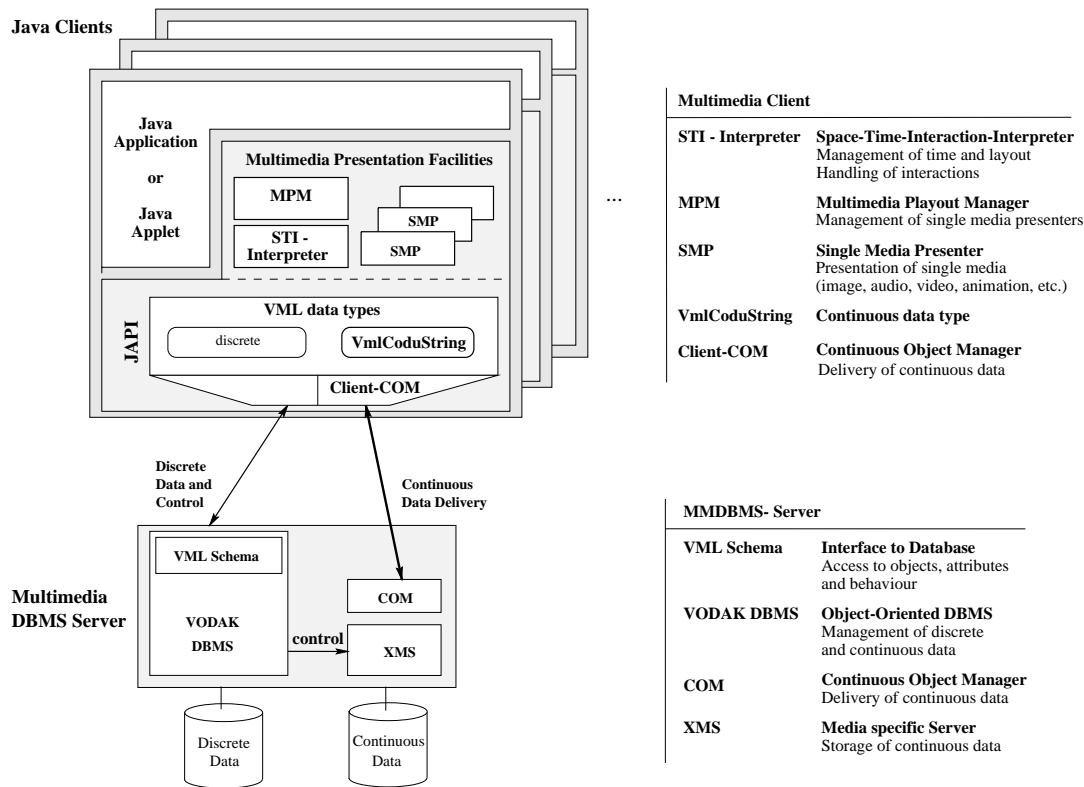


Figure 6: AMOS multimedia database management system architecture

media classes. These methods encapsulate the transport of media objects from server to client. E.g., a method `getImage(Vml0id anImage)` creates a Java `VodakImage` object that represents the corresponding database object at the client. In the case of discrete media types the client media object is a “proxy” of the server object and contains a copy of the raw media data. In case of continuous media types, the instantiated Java object contains a `VmlCoduString` object that is explained next.

Access to continuous data in Java. To give access to continuous data stored in the multimedia database server we implement a Java class `VmlCoduString`. As this kind of data can be of large size it is not feasible to transport the entire data from the server to the client at once — “just in time” delivery is necessary. Appropriate mechanisms for the preloading and buffering of CODUs have to be integrated into the client [MKK95]. These are encapsulated in the client’s Continuous Object Manager (Client-COM) that communicates with the server COM. The class `VmlCoduString` relieves the application from all details about continuous data transport and buffer management strategies. Figure 7 gives the signatures of some methods of `VmlCoduString`. With

`triggerContinuousDataFlow` the continuous data delivery to the client is initiated. Furthermore, `VmlCoduString` implements several methods to access single CODUs at the client.

Presenting persistent media objects in Java.

Each media type has its own requirements for presentation. The information available with a media object, however, is not sufficient for the object’s presentation. E.g., the presentation of an image takes place at a certain region on the screen or an audio is played back at a certain volume. These presentation parameters are provided by the presentation plan or supplied by the application program.

We implement a Single Medium Presenter (SMP) for each media type that encapsulates the necessary presentation information. SMPs provide methods for the control of the actual presentation of a single media object. SMPs are arranged in an inheritance tree which is indicated in Figure 8. The root class `SMP` encapsulates the global presentation information. A concrete media presenter, e.g., `SMP_Image`, then refers to the media object and encapsulates the specific information for the presentation of the media type.

At the same time the SMPs need specific presen-

```

public class VmlCoduString extends VmlDataType {...
    private int numberOfCodus;
    ...
    public boolean synchronized triggerContinuousDataFlow();
    VmlCodu public synchronized getNextCodu(); ...}

```

Figure 7: Sample method signatures of the Java class `VmlCoduString`

tation methods to present the corresponding media object at the designated device (screen, speaker, etc.). The presentation methods of SMPs are described by the interfaces outlined in Figure 8. The implementation of the single medium's presentation uses Java's abstract window toolkit that offers support for implementing graphical user interfaces. The SMPs for continuous media types trigger the continuous data transport prior to the media object's presentation. The actual presentation of the media object runs within a thread that is controlled by the SMP. The usage of threads enables simultaneous media presentations.

Presenting persistent multimedia documents in Java. Multimedia clients must be able to access multimedia documents and support the interactive presentation of these documents at the user's site. The latter calls for synchronization enforcement, the realization of presentation layout, and an efficient interaction handling.

As described in section 5.1, the modeling of multimedia documents is invisible to the client as only presentation plans are transferred to the client. When a client requests the presentation of a multimedia document from the multimedia database server the presentation plan for this document is returned. A presentation plan contains only references to the media objects in the database, not the media data itself.

What we need to carry out interactive presentations of multimedia documents is the interpretation of the presentation plan, the timely access to media objects in the database, spatial layout and temporal synchronization of the presentation, and the handling of user interactions during the presentation [BKL96]. We propose to separate the presentation tasks from the interpretation and the interaction handling. This allows to change the internal encoding of presentation plans without changing the actual presentation classes. Therefore, we implement two components that jointly carry out the presentation: A Space-Time-Interaction-Interpreter (STI-Interpreter) and a Multimedia Playout Manager (MPM) shown in Figures 6.

STI-Interpreter: The STI-Interpreter is responsible for the interpretation of a presentation plan. From that plan it extracts the temporal and spatial information about the presentation and generates

a *schedule* [BKL96]. The schedule includes additional information about the estimated time period needed for preparing a single medium's presentation. With this schedule the MPM is controlled. At the same time the STI-Interpreter is responsible for an efficient interaction handling. When a user interaction occurs the STI-Interpreter has to react to it, e.g., the entire presentation is stopped, paused, or resumed. By a user interaction another multimedia document can be requested for presentation, too. The multi-threading concepts of Java are of advantage as the interpretation of a presentation plan and handling of user interactions can be implemented in different synchronized threads that run in parallel.

MPM: The MPM controls the actual presentation of multimedia documents at the client. From the schedule created by the STI-Interpreter it prepares and triggers the sequential and parallel presentation of media objects.

For each single media object involved in a document's presentation the MPM creates at preparation time a SMP. The presentation parameters passed to the SMP include the `Vml0id` of the database media object to be presented. After its creation, the SMP runs through a preparation phase (`preparePresentation`). During this phase the SMP initiates the creation of the respective client media object. In case of continuous SMPs the continuous data transport is triggered.

Monitored by the MPM the SMPs carry out the respective single medium presentations. That is from the schedule of the STI-Interpreter the MPM starts, stop, pauses, and resumes the single medium presentations and synchronizes the single medium presentations. Parallel presentation of media objects is possible because each SMP runs its presentation in a thread.

The implementation of multimedia presentation facilities for a multimedia enhanced OODBMS in Java profits from the capabilities of the language. Java supports multi-threading and at the same time offers integrated support for building graphical user interfaces by its abstract window toolkit package. This is not obvious in programming languages. When we tried to use C++ with a thread-library together with a Motif class library we had

```

public interface PresentationInterface { ...
    public preparePresentation();
    public startPresentation();
    public stopPresentation(); ... }

public interface ContinuousPresentationInterface extends PresentationInterface { ...
    public pausePresentation();
    public resumePresentation(); ...}

public class SMP_Audio extends SMP implements ContinuousPresentationInterface { ...
    private VodakAudio anAudioObject;
    private int volume; ...}

public class SMP_GraphicalObject extends SMP { ...
    private int xPos; private int yPos;
    private int presentationHeight; private int presentationWidth; ...}

public class SMP_Image extends SMP_GraphicalObject implements PresentationInterface { ...
    private VodakImage anImageObject;
    private SMP_Image_Thread aPresentationThread;
    private String presentationStatus;
    ...
    public String getPresentationStatus() {
        return presentationStatus; }
    public boolean preparePresentation() { ...
        presentationStatus = "prepare";
        aPresentationThread = new SMP_Image_Thread (presentationStatus, this);
        aPresentationThread.start(); ... }
    public boolean startPresentation() { ...
        presentationStatus = "start";
        aPresentationThread = new SMP_Image_Thread (presentationStatus, this);
        aPresentationThread.start(); ... }
    ... }
... }

```

Figure 8: Sketch of SMP classes and interfaces

to cope with the problem that Motif is not MT-safe. Moreover, Java interpreters are available for almost every platform and, thus, with the Java API we get a platform-independent implementation of a client that is able to present multimedia documents managed by a multimedia database server.

6 Related work and conclusions

Currently there is a lot of work ongoing related to Java and persistency. The research and development activities in this area reach from accessing persistent data in pre-existing database systems up to adding transparent persistence to the Java programming language.

The most prominent and mature approach for accessing Relational DBMSs seems to be JDBC [HR96]. JDBC defines a database access API that supports basic SQL functionality and enables to access a wide range of RDBMS products. With JDBC Java can be used as the host language for writing database application programs. On top of JDBC higher-level APIs can be built, e.g., for the mapping of Java classes and objects to RDBMS tables and rows and/or vice versa.

With respect to accessing object-oriented DBMS, ODMG is working towards a standard API to use

ODMG-compliant OODBMS together with Java. The ODMG consortium promised to come up with its Java binding in the release 2.0 of the ODMG-93 specification [Cat94] beginning of 1997. The ODMG Java binding aims at transparent persistence and will implement persistence by reachability. In this sense, the ODMG Java binding is similar to Persistent Java. The Persistent Java project [Spe96] tries to integrate orthogonal persistence into the Java programming language.

None of those activities are specially concerned with accessing multimedia enhanced object-oriented database systems and presenting persistently stored multimedia documents using Java.

In this paper, we described how to implement such multimedia presentation services in Java by using the OODBMS VODAK as an example. First, we presented a Java interface to the core OODBMS that manages discrete data. With this interface Java can be used to write database application programs in which OODBMS commands, queries and method calls on database object are embedded. Then, we described how to access continuous data persistently stored in a DBMS. Moreover, we elaborated how to realize the synchronized playback of persistently stored multimedia documents at a Javaclient.

The resulting Java API can be used to build stand-alone Java application programs as well as

Java applets that can be easily downloaded via the network as part of an HTML document. Such multimedia database clients can interact with the multimedia database server and present the interactive multimedia documents stored at the server.

References

- [AF95] K. Aberer and G. Fischer. Semantic query optimization for methods in object-oriented database systems. In *Proceedings of the 11th IEEE International Conference on Data Engineering, Taipei, Taiwan*, March 1995.
- [AK92] K. Aberer and W. Klas. The impact of multimedia data on database management systems. Technical report, International Computer Science Institute (ICSI), Berkeley, CA, USA, 1992.
- [BKL96] S. Boll, W. Klas, and M. Löhr. Integrated Database Services for Multimedia Presentations. In *[Chu96]*, 1996.
- [BLW96] S. Boll, M. Löhr, and J. Wäsch. *Vodak Remote Application Programming Interface (Version 2.0) - Connecting VODAK to outer space*. Technical Report (Arbeitspapiere der GMD). GMD Sankt Augustin, September 1996.
- [BWAH96] A. Bapat, J. Wäsch, K. Aberer, and J. M. Haake. HyperStorM: An Extendable Object-Oriented Hypermedia Engine. In *Proc. of the Seventh ACM Conference on Hypertext (HT96)*, 1996.
- [Cat94] R. G. Catell, editor. *The Object Database Standard: ODMG-93*. Morgan Kaufmann, 1994.
- [Chu96] S. M. Chung. *Multimedia Information Storage and Management*. Kluwer Academic Publishers, 1996.
- [CK95] S. Christodoulakis and L. Koveos. Multimedia Information Systems: Issues and Approaches. In *[Kim95]*, pages 318–337. Addison-Wesley, 1995.
- [DOBS94] Asuman Dogac, M. Tamer Özsu, Alexandros Biliris, and Timos Sellis, editors. *Advances in Object-Oriented Database Systems*, volume 130 of *NATO ASI Series F*. Springer Verlag, Berlin, 1994.
- [GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.
- [GMD95] GMD-IPSI. VODAK V 4.0 User Manual. Technical report, GMD, April 1995.
- [HR96] G. Hamilton and R.G.Catell. JDBC: A Java SQL API. Technical report, SunSoft, 1996.
- [KAN94] W. Klas, K. Aberer, and E. Neuhold. Object-Oriented Modeling for Hypermedia Systems using the VODAK Modeling Language (VML). In *[DOBS94]*. Springer Verlag Berlin, 1994.
- [KFM⁺96] W. Klas, P. Fankhauser, P. Muth, T.C. Rakow, and E. J. Neuhold. Database integration using the object-oriented database system VODAK. In O.A. Bukhres and A.K. Elmagarmid, editors, *Object-Oriented Multidatabase Systems*, pages 472–532. Prentice Hall, 1996.
- [Kim95] Won Kim. *Modern Database Systems - The Object Model, Interoperability and Beyond*. Addison-Wesley, 1995.
- [LG90] T. D. C. Little and A. Ghafoor. Network Considerations for Distributed Multimedia Object Composition and Communication. *IEEE Network*, 4(6):32–49, November 1990.
- [MKK95] F. Moser, A. Kraiss, and W. Klas. L/MRP: A Buffer Management Strategy for Interactive Continuous Data Flows in a MMDBMS. In *Proceedings of the 21st Conference on Very Large Data Bases (VLDB'95)*, pages 275–286, September 1995. Zurich, Switzerland.
- [MRW⁺93] P. Muth, T. C. Rakow, G. Weikum, P. Brössler, and C. Hasse. Semantic concurrency control in object-oriented database systems. In *Proceedings of the 9th IEEE International Conference on Data Engineering (ICDE'93)*, pages 233–242, April 1993.
- [OMG95] OMG. The common object request broker: Architecture and specifications. Technical report, Object Management Group, 1993, 1995.
- [RNL95] T. Rakow, E. J. Neuhold, and M. Löhr. Multimedia Database Systems - The Notions and the Issues. In G. Lausen, editor, Tagungsband GI-Fachtagung Datenbanksysteme in Büro, Technik und Wissenschaft (BTW), Dresden März 1995, pages 1–29. Springer Verlag, Informatik Aktuell, 1995.
- [Spe96] S. Spence. Design issues for persistent java: a type-safe, object-oriented, orthogonally persistent system. In *Seventh International Workshop on Persistent Object Systems, Cape May, New Jersey*, May 1996.
- [WA96] J. Wäsch and K. Aberer. Flexible Design and Efficient Implementation of a Hypermedia Document Database System by Tailoring Semantic Relationships. In *Proceedings of the Sixth IFIP Conference on Data Semantics (DS-6), Atlanta, Georgia, USA, May 30 - June 2, 1995*, To appear in Meersman, Mark [Ed.]: *Database Application Semantics*. Chapman and Hall, 1996.